O'REILLY®

响应式Web设计性能优化

High Performance Responsive Design

[美] Tom Barker 著
余绍亮 丁一 叶磊 译
王建 审校

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

目 录

版权信息

书名：响应式Web设计性能优化

ISBN：978-7-115-39974-8

- 著　　[美]Tom Barker

  译　　余绍亮　丁　一　叶　磊

  责任编辑　赵　轩

- 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

  邮编　100164　电子邮件　315@ptpress.com.cn

  网址　http://www.ptpress.com.cn

- 读者服务热线：(010)81055410

  反盗版热线：(010)81055315

版权声明

作者简介

Tom Barker 资深Web技术专家，有20余年行业经验，专注于Web开发的各个方面。现为Comcast公司的Web开发高级经理，费城大学的兼职教授。痴迷于优雅的软件解决方案，软件持续改进，数据的提炼、分析以及可视化。 出版有 JavaScript性能优化：度量、监控与可视化（首部系统化阐述JavaScript性能优化的经典著作，Amazon全五星好评）

前言

现如今，虽然响应式设计已经成为一个热门话题，但它仍然被当作一门前端技术。在大多数开发人员的印象中，响应式设计通常都与媒体查询有着紧密的关系。在本书中，我更愿意把响应式设计称为一种哲学，而不仅仅是一门技术：一个可以从传统单一的前端处理转变为从多方位考虑的理想解决方案，因为每个HTTP请求中都携带了许多信息到Web服务器，这样Web服务器就可以根据信息做相应的响应。在某些场景中，在后台实现响应式是一种更好的解决方案。

正因为周围的设计者和开发者常常有构建响应式站点方面的困惑，而一些公司也敏锐地意识到Web性能会带来非常大的挑战，我便萌生了写一本关于响应式设计的书的想法。如果只是关注客户端的响应式，而不去寻找更多的性能方案，逐渐地，响应式的优势以及我们的自身的效率都会陷入瓶颈。

当我在写这本书的时候，它就开始有了自己的生命。当我们注意到响应式网站的性能已经严重影响到网站的正常运行时，我们该如何去面对？如果我们为网页性能创建SLA，在开发阶段持续集成环境中，该怎么对性能进行测试？

本书将为你一一解答。

目标读者

本书主要面向Web开发者，特别是那些只关注前端技术，而对后台技术涉足不深的Web开发者。这也是为什么我没有提那些老生常谈的前端性能，如CSS最佳实践之类的话题。如果你对这些感兴趣，你可以从其他很多地方获取相关的知识。同时这也是为什么我将JavaScript语言作为首选语言来写示例程序，尤其是使用NodeJS写后端示例代码。

也就是说，设计者、技术组长和任何层次的开发者都可以从本书丰富的示例与相关的知识中获益。

章节预览

第1章中，我使用了排名前50的站点作为示例数据展示了在响应式设计中使用的设计模式和反模式。这些模式和反模式作为指导原则贯穿于整本书。我们也探讨了m. 站点的使用以及它们的优点与缺点。

第2章初步探讨了Web性能相关概念、Web运行时性能和监测性能的相关工具。如果你还不熟悉Web性能的方方面面，本章内容将是一个很好的参考。同时它也能很好地帮助你对不常提及的知识进行回顾，如客户端的内存消耗等。

第3章着重探讨了如何将响应式加入到项目的规划和培训阶段，尤其是对响应式网站性能进行说明的SLA。

第4章主要探讨了后端的响应式性能实现的相关概念。我们使用NodeJS编写了为客户端专有的设备体验的功能。同时，我们也使用第三方设备库来获得更多客户端功能相关的上下文，而不是自己简单地检测User Agent字符串，然后推导出设备功能。

第5章关注前端解决方案。通过picture元素和secret属性来加载特定设备专用的的图片，以及根据客户端特性，对图片和整个页面片段进行延迟加载的相关概念。最后，展示了如何使用客户端设备库API来决定表单因子。

第6章使用了PhantomJS来完成自动测试和SLA性能验证，并且把这些测试用例整合到Jenkins持续集成环境中。

第7章着重介绍了当前构建响应式网页的各种框架情况，也对它们的易用性、使用的模式和反模式、第三方依赖和对页面负载的影响等方面都做了一一比较。同时也介绍了我的开源服务端模板框架—Ripple，它是基于第4章的示例代码开发的。

注意

技术更新的速度总会超过我们写书的速度，即使我们一直不断的进行更新也是无法跟上技术更新的脚步。但是我不得不说，出版社做的是一件非常伟大的工作，它总是在第一时间将它们的书送到读者的手中。也就是说，在第1章中，我们展示美国排名前50网站的例子其实2013年12月统计的，在那以后，Alexa的统计一直不断更新，其余的网站一直不断的更新他们的网页。一些浏览器也已经迭代发布更新了他们加载和预加载资源的方式。而对于书中讨论的任何标准也是一样的。在你阅读这段话的时候，也许它们就已经被更新或者已经排上日程了。

致谢

在这里我想感谢我最美丽的妻子，Lynn，在这一年的创作时间里，是她的耐心陪伴我度过了每个深夜和周末。同样感谢我可爱的孩子们，我原本想在他们熟睡以后才写本书，但是有时候计划赶不上变化，同样感谢他们的耐心和理解。

我也要感谢Mary Treseler能给我这次机会，让我能够创作本书，他的博学和指导也让我受益匪浅。我也想对Colleen Lobner、Nick Lombardi、Melanie Yarbrough和 Dianne Russell表达我的感激之情，没有他们的帮助，也就没有本书的出版。我也要感谢Ilya Grigorik、Lara Swanson、Clarissa Peterson和 Jason Pamental，他们的精益求精，他们的热情敬业，一直帮助我修改书中的各种问题，他们对本书的完成也是至关重要的。

第1章　响应式设计现状

1.1　响应式设计存在的问题

我曾与我的一个团队和产品经理参加了一个产品规划会议，讨论重新设计我们的视频区块，我们的团队主管提出了可以让这个区块具备响应式体验的方案。我们勾勒了这样一张页面，它会去加载默认的HTML5视频播放器，不过会根据用户所使用的设备来调整播放器的大小以及加载不同视频类型的资源与播放列表。这样我们的页面将会非常漂亮，能够适应更多的浏览设备，而且我们的视频就可以面向以前被拒之门外的各种设备的观众了。

这时我们的产品经理皱起鼻子说道：“我们的响应式首页出来之后，我就开始觉得我们的响应式设计思想有些问题。”

这让我甚感诧异。我们的响应式首页问题出在哪里？于是我开始着手做了些研究。

产品团队的印象是页面加载很缓慢。其实当开发人员在自己的笔记本电脑上演示时，一切看起来都很棒，然而当他们在真实设备上向高管们展示时，页面的加载时间实在太长太长了。

我查看了一下主页在智能手机和电脑渲染瀑布图。我所看到的就是我在许多其他站点中注意到的一些东西。

智能手机端在渲染的时候不仅加载了桌面端全部的资源文件，还加载了额外的CSS与sprite文件。图1-1表明相对于桌面端，在手机端的渲染增加了两个额外的HTTP请求，下载量也略微大一些（1.2MB vs 952KB）。

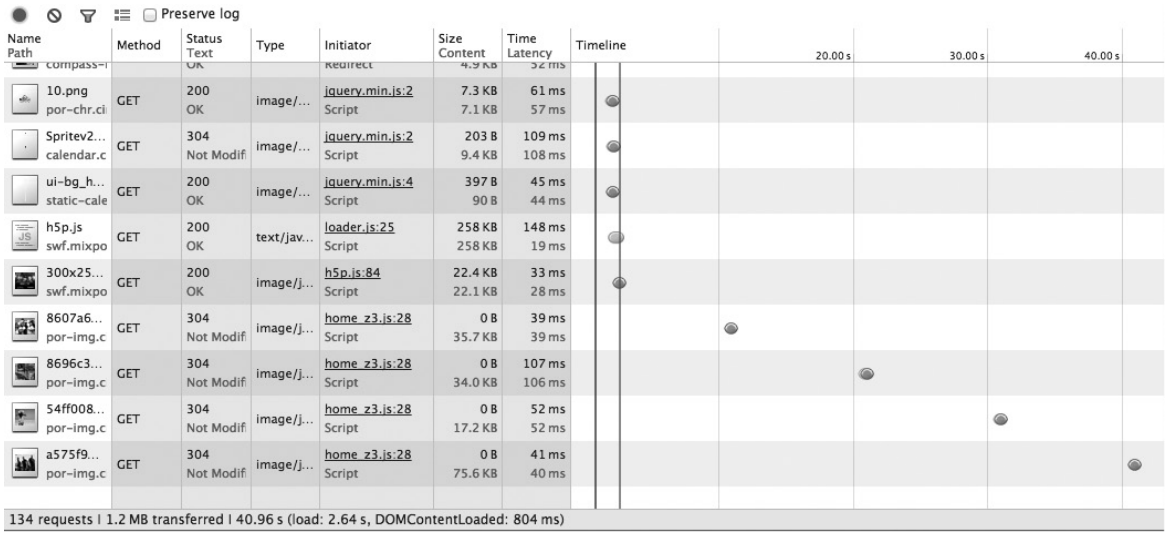

图1-1　主页在手机上渲染时的瀑布图

从图1-1中可以看到，共发送了134个HTTP请求，总的传输量是1.2MB。不过这是智能手机版的，通常情况下应该比桌面端的荷载要小。可事实并非如此，如图1-2所示。

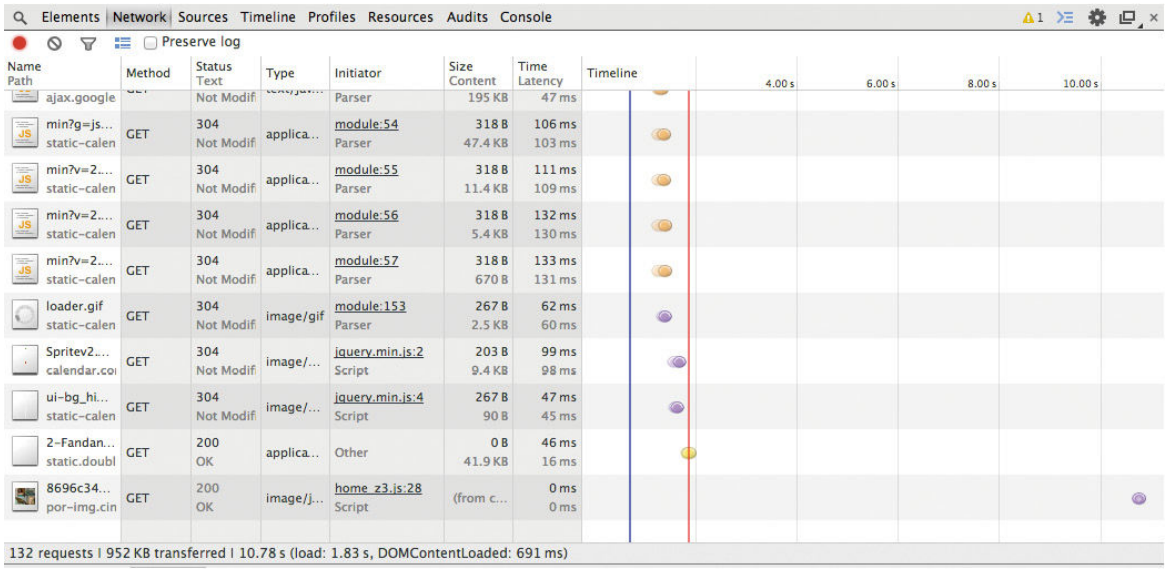

图1-2　桌面版主页渲染时的瀑布图

可以看到桌面版本总荷载是952KB，来自132个HTTP请求。无疑，手机版除装载所有桌面版同样的内容外，还增加了两个文件。显而易见，这加剧了移动体验中的带宽问题。这完全与我们创建手机页面的初衷背道而驰了。

无独有偶，我在我的笔记本上打开浏览器，同时查看我iPhone上的HTTPWatch，然后访问了alexa.com排名前50的网站，做了一些竞争分析。我发现这些网站中，有30%的手机版的荷载要高于其对应的桌面版——科技公司、银行、零售商均是如此。

除了我自己的研究，许多著名的报告也有相似的结论。The Search Agency（一个全球数字营销代理机构）分析了零售百强以及财富百强公司的站点，做出了下面的报告。

- “多重通路零售商”（*http://bit.ly/1vqYUPh*）。
- “财富一百强公司”（*http://bit.ly/1r1SDla*）。

> **提示**
>
> 要想访问这些报告，需要将你的E-mail地址提供给The Search Agency，然后The Search Agency会将报告发送给你。

在这些结论中，图1-3展示出使用了（确切地说是滥用）响应式设计的站点比简单、普通的桌面站点平均多耗时1.91秒。更让人诧异的是，它们比手机专用的站点多耗时10.74秒。

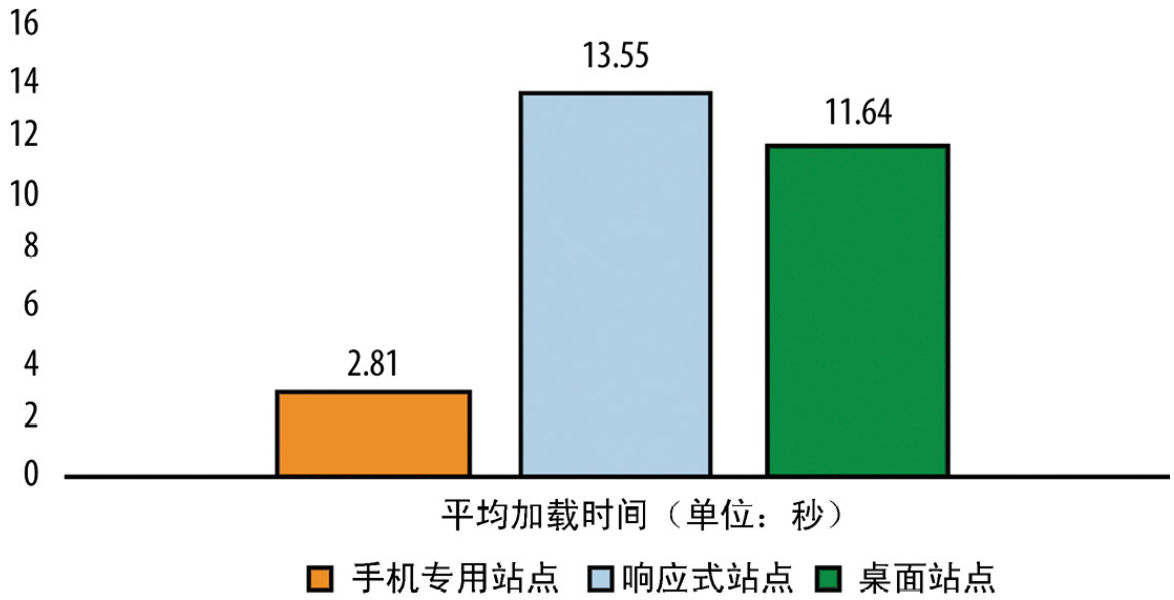

图1-3　The Search Agency对响应式站点与手机专用以及桌面专用站点的平均加载时间比较

Guy Podjarny——Akamai公司的CTO——在其博客上也发表过一段文章，详细描述了他在运行类似测试实验时的发现。他比较了多种分辨率下

的页面大小，发现其间的差别微乎其微。在*http://bit.ly/1tBv6cT*上可以找到他的文章。

我们都忘记了创建响应式体验的意义了吗？

竞争分析中的发现

我自己在对Alexa排名列表中的页面测试中也得到了一些有趣的数据。其中，我注意到了下面的情况。

- 在美国的热门网站中，47%仍然使用的是手机专用的“m.”类型的站点。花一分钟时间思考一下这个数字。这些都是互联网上流量最大的站点，可能是各行业的领头羊，包括YouTube、eBay以及Target公司，它们都没有采用响应式站点，而是使用了独立分开的站点。
- 平均来说，这些手机专用的站点的文件要比响应式站点小55%。使用“m.”的网站子集平均大小是383KB，而响应式站点的平均大小是851KB（见图1-4）。可见“理想很丰满，现实很骨感”。

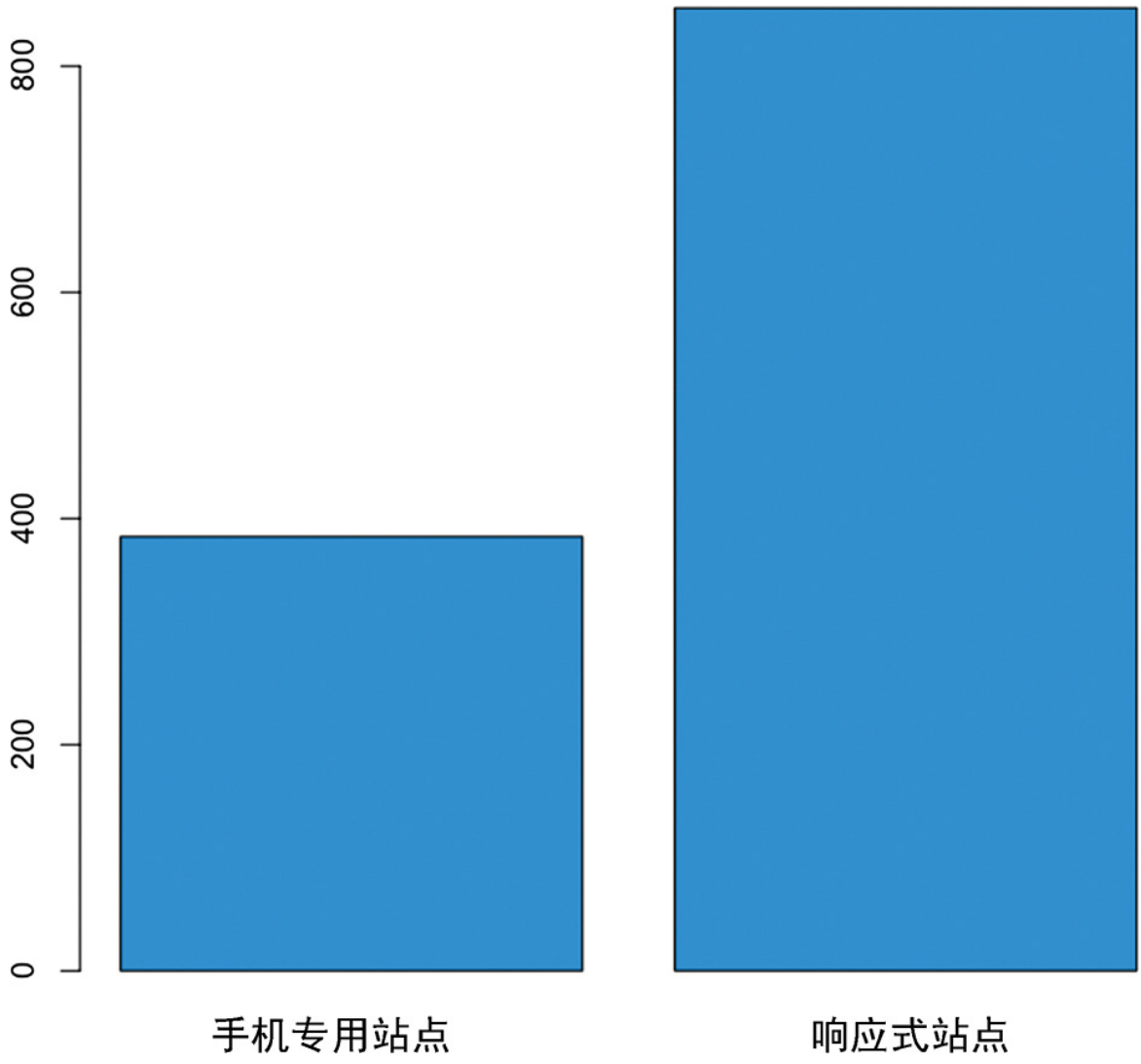

图1-4　手机专用站点与响应式站点平均文件大小对比（单位：KB）

- 响应式站点的荷载有长尾分布的特性，可达到4MB，而“m.”类型的站点的分布范围都在1MB以内。实际上，“m.”类型的站点大都密集分布在0～200KB以及200～400KB的范围内。我做了一个直方图，来研究下“m.”类型的站点与响应式站点中文件大小的分布情况，见图1-5和图1-6。

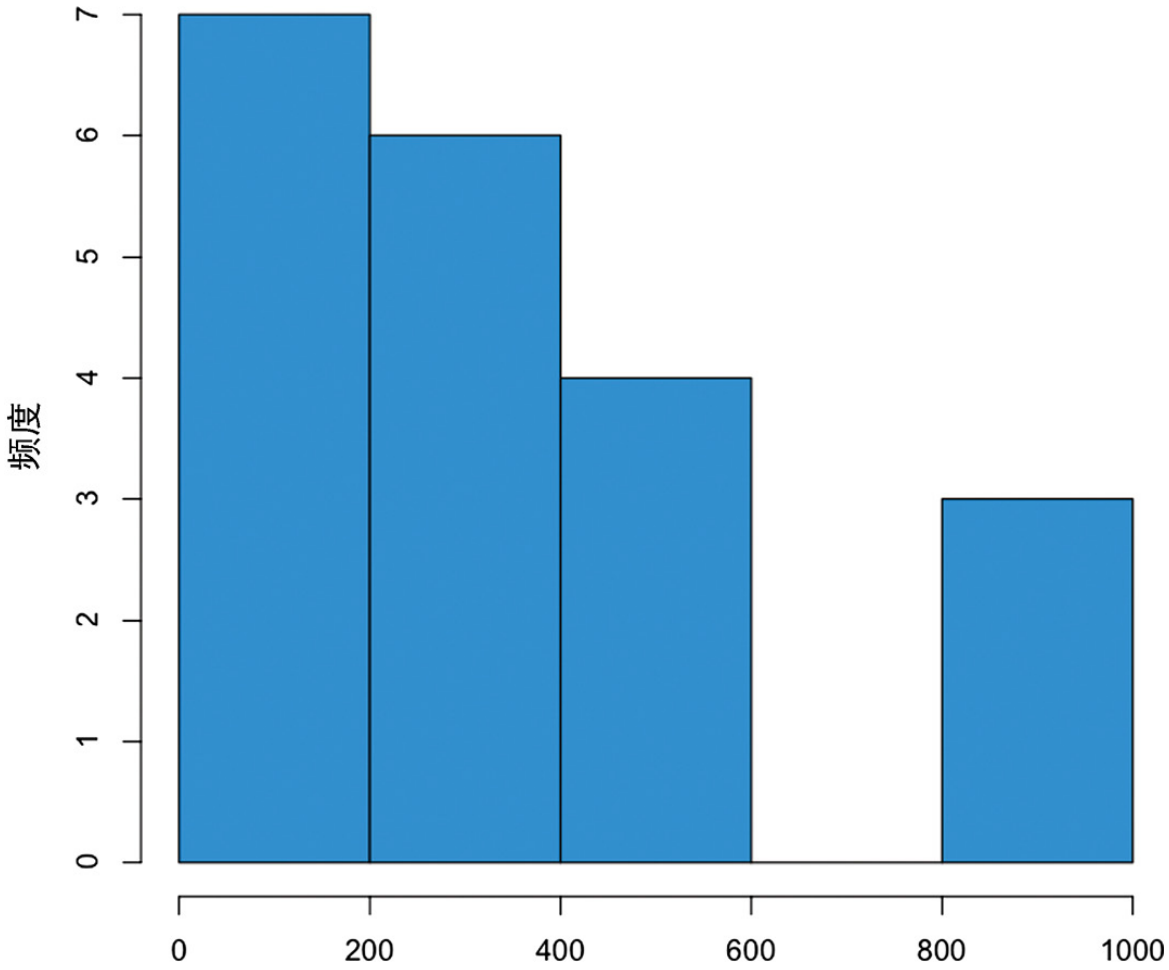

图1-5　手机专用的站点文件大小分布（单位：KB）

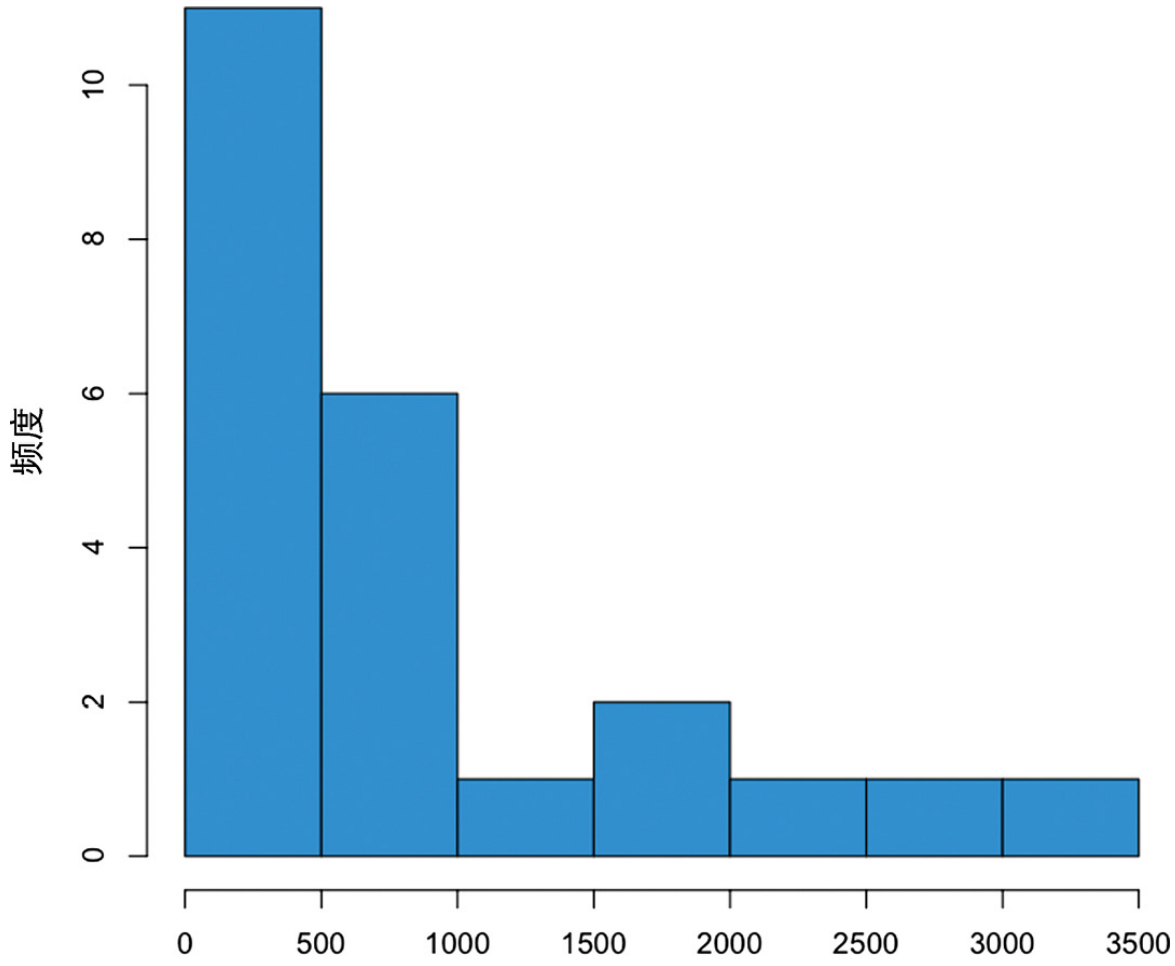

图1-6　响应式站点中文件大小的分布（单位：KB）

注意每个直方图中x轴的缩放比例。手机专用站点中有3个离群值接近于1MB；响应式站点中，1MB是第二大分组，且尾部一直延续到4MB。

- 对于响应式站点，43%的站点在手机体验中与桌面体验中的HTTP请求几乎相同，或是稍多一些。相比之下，专用站点中只有1.5%在手机体验中HTTP请求等同或高于其桌面体验版。图1-7和图1-8描绘了这种差异。

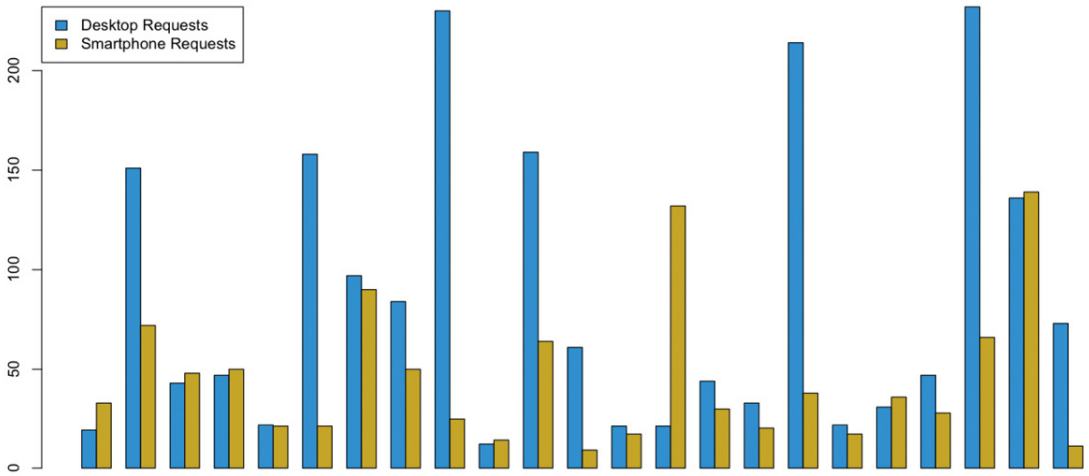

图1-7　响应式站点在桌面和手机体验中HTTP请求的分组条状图

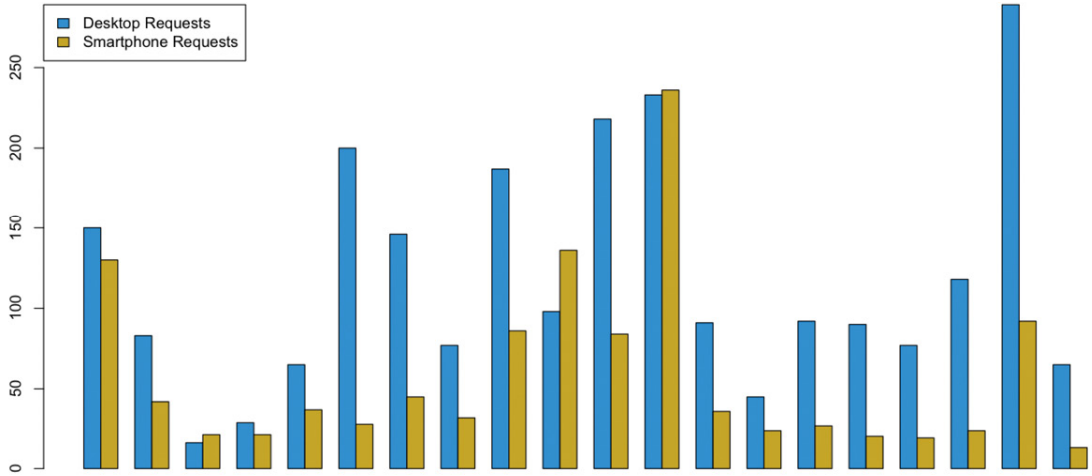

图1-8　桌面以及手机专用的“m.”站点体验中HTTP请求的分组条状图

在图1-7中的每个分组里，蓝条表示桌面体验中HTTP请求数目，黄条表示同一个页面在手机体验中的HTTP请求数。

同样，在图1-8中的每个分组中，蓝条表示桌面体验中HTTP请求数目，黄条表示手机体验中的HTTP请求数。

显然，我们的响应式设计的实现有问题。并且，提供一个专用体验的站点存在明显的优势，至少在HTTP请求数以及渲染一个页面需要的总传输荷载方面是这样的（不过也要注意，“m.”类型的站点也有其自身的一系列问题，我们后面会简短讨论）。应当注意到，贯穿本书，我的观点以及重复提到的主题自始至终都是响应式设计和专用体验并不是互斥的实现，而是同一理念的多个方面。

除了前面的指标，我还注意到我观察过的那些站点似乎是遵循了一些模式和反模式。

反模式

我在研究Alexa列表上的每个站点时，发现它们都存在一些共同的问题以及它们用到的一些反模式。接下来，我们来找出并研究一下这些反模式。

为所有设备加载同样的内容

这些站点中，有一些会为手机和桌面渲染加载完全一样的资源。它们加载同样的CSS文件，通过媒体查询给不同的分辨率设备提供不同的体验。加载同样的图片，通过缩放来解决不同分辨率下的显示需要。

这种错误做法的后果会在HTTP拥堵时暴露无遗。在不同的显示设备中HTTP请求数目完全一样的那些站点极可能就是用了这种做法。当开始考虑更大分辨率的显示设备如苹果的Retina显示屏以及超高清电视时，这种解决方案就不能很好的扩展显示。

加载额外的资源

虽然为所有设备加载同一组资源忽略了设备间的本质差异，但在手机体验时加载除了通用资源以外的资源，就完全与我们所熟知的移动体验相违背了。这些额外的资源一般都是一些CSS以及sprite文件。

手机体验中比桌面包含更多HTTP请求以及更大荷载的站点常会表现出这种行为。正如前面所提到的，我自己的站点正是用了这种反模式。

加载双倍的图片

最大的错误是，有些站点会为手机版本加载另外一组图片，如此一来图片文件的大小就是桌面版本图片的两倍了。这是常规桌面版图片之外的内容。

加载更大的图片然后再调整大小是为了让图片在小尺寸下显得更清晰。但这种做法的副作用就是会导致手机版站点的荷载比其桌面版大约要大30%。

所有这些问题都有几个共同的哲学思想。

- 它们明显都是着眼于桌面版本，并以此作为基础，在其上修改或新增元素，而不是从最小版本往上开发。
- 它们都没有利用各个平台的优势，也没有意识到各个平台的限制。
- 它们都试图仅从客户端来解决问题。

模式

并不是所有Alexa列表中的站点的做法都是错的——有些确实体验很好，为各种设备与分辨率做了优化。我们来看看它们用到的一些设计模式。

加载适合相应设备的资源

一些网站针对手机端加载了相比桌面端一半大小的图片（而不像之前所描述的反模式中加载两倍桌面端大小的图片），图1-9和图1-10展示了一个例子。

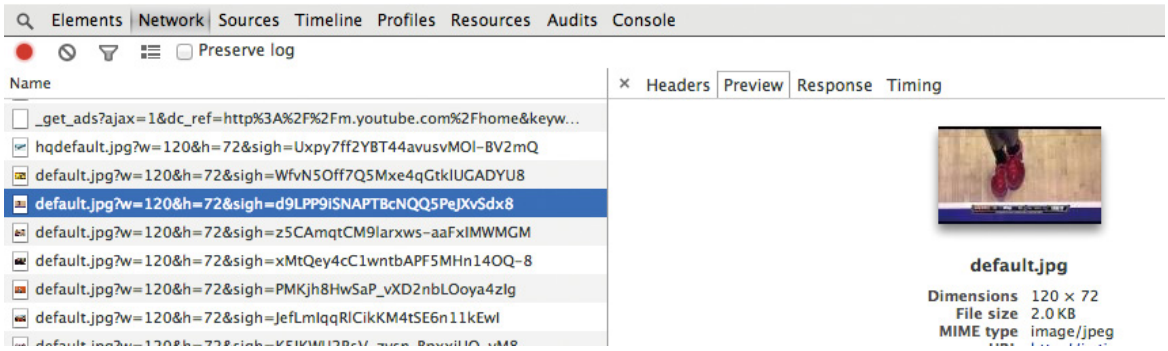

图1-9　针对手机端体验加载了专门为手机端准备的图片，120×70像素，2KB大小（截图自Chrome开发工具）

可以看到，在图1-9和图1-10中所展示的是同样的图片，它们的区别仅仅在于针对不同的客户端环境进行了不同程度的缩放。

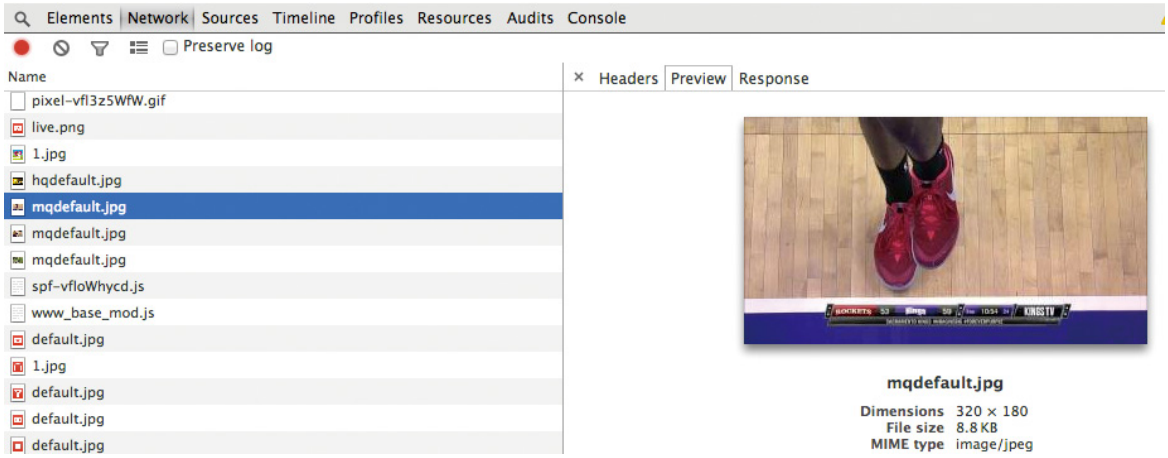

图1-10　针对桌面端体验加载了专门为桌面端准备的图片，120×180像素，8.8KB大小（截图自Chrome开发工具）

同样，有些站点使用了设备独有的sprite图和CSS——并不是在桌面程序的基础上为其他设备增加其他资源。这样需要适当考虑网络带宽限制和传输成本。在alexa名单中的大多数站点都是使用专有“m.”网络来完成此功能的。我们同样也可以建立使用此模式的响应式网站，这部分的内容会在第4章中出现。

从后端提供专有的体验

最好的浏览体验就是对于不同的设备提供完全不同的专有体验。有些站点提供了独立的“m.”网站，另外一些网站展示的是从服务端传输过来的基于设备独有布局和功能的页面。这种解决方案我们称之为RESS（响应式设计 + 服务端组件），但是对于我们常用来为预定义的分辨率断点来说，我们真的需要在一个“m.”站点中为每一部分功能传输加载适当的内容的同时，合并相同的业务逻辑。我们将在第4章讨论这种方案的更多细节。

为了更清晰地解释这个解决方案的架构，我们来看看图1-11所展示的这个架构的时序图。

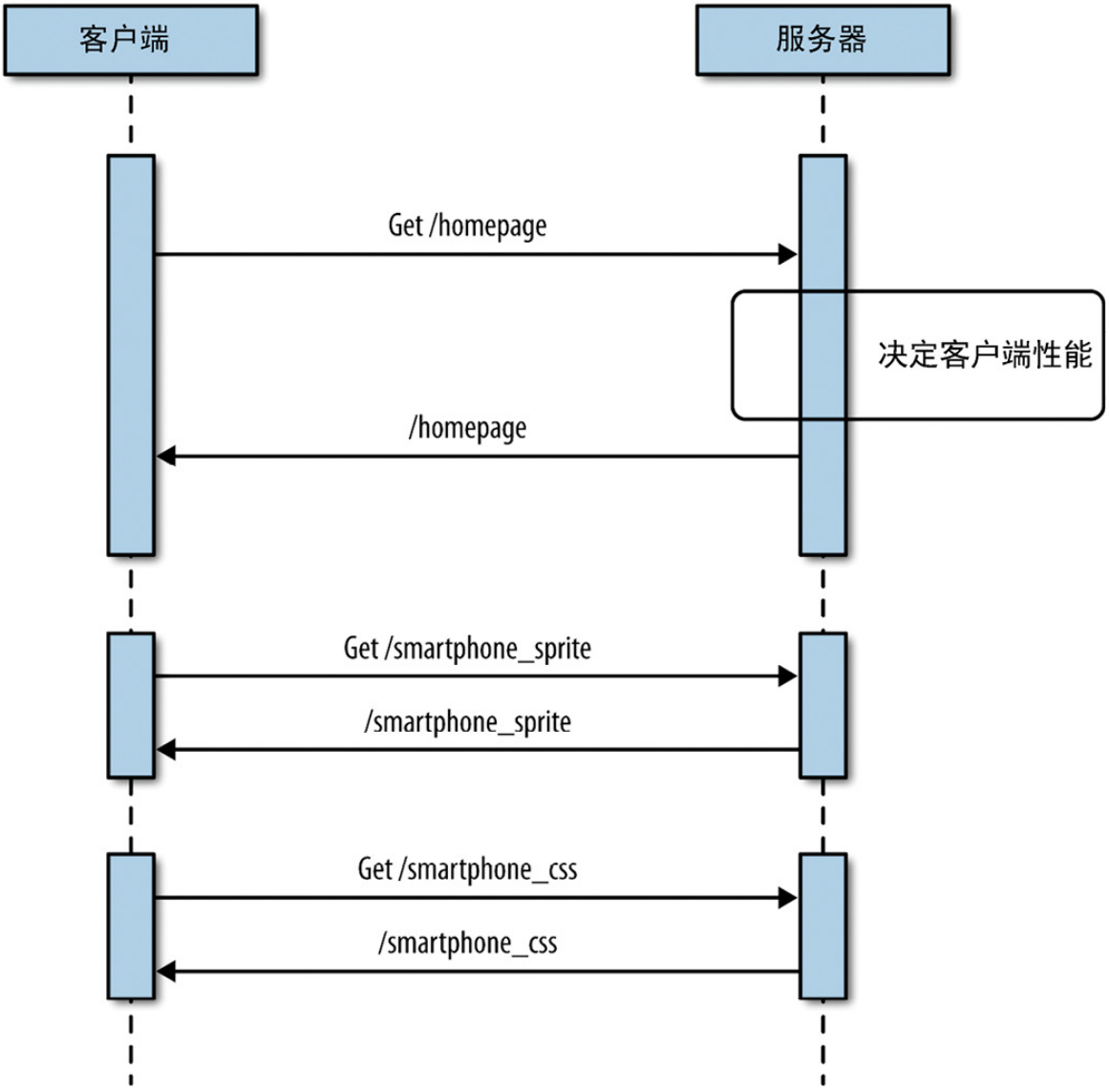

图1-11　后端提供基于设备的专有体验的时序图

请注意，这些站点在提供专有体验的同时，保证了最小的荷载和最大限度的加速。这也就是为什么47%的顶尖站点仍然提供专有的网页内容。

前端延迟加载的专有体验

有些站点不仅对图片进行延迟加载，而且对整个内容模块也进行延迟加载，包括上方和下方的折叠部分。通过这种方式，可以有效地避免为每个断点加载内容，智能地加载那些需要的内容，从而适应客户端性能，达到最佳的体验，但是是否延迟加载的决定权在客户端，而不是在服务端，我们将在第5章详细讨论相关细节。

图1-12用时序图展示了这种方法的细节。

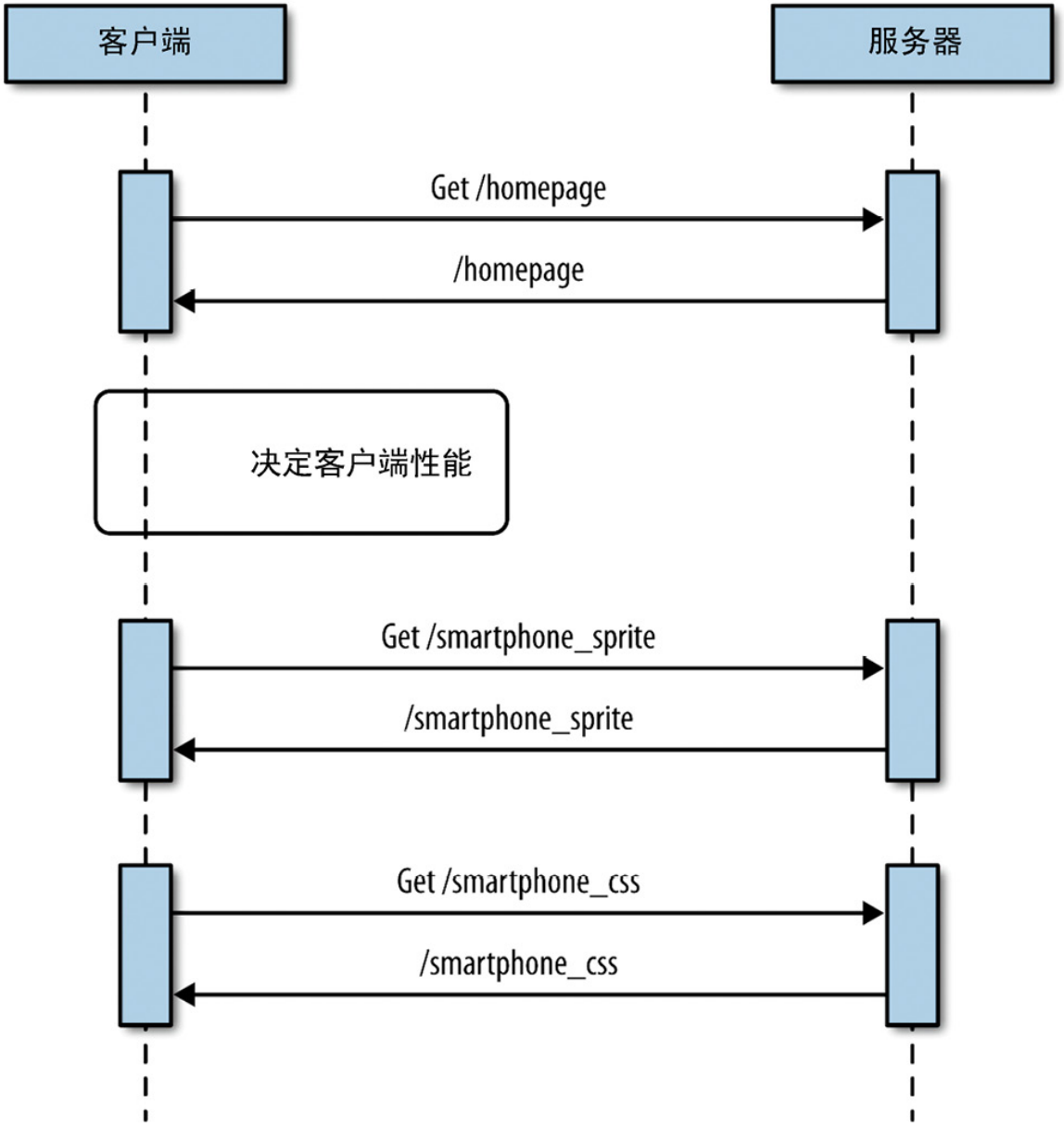

图1-12　从客户端加载和设备相匹配的资源

我们怎么没有感觉到

本章的前面部分我已经描述了我们是怎么向产品经理展示我们的响应式主页的。经过一个迭代的开发，我们已经可以在我们的笔记本电脑上打开这个页面了，现在我们的桌面屏幕上可以满屏展示这个页面，并且可以自定义我们的浏览器窗口大小来适应不同的屏幕大小场景。虽然看着我们

页面可以自适应屏幕大小是一件非常有趣的事情，但是在其他的设备上，它表现得一团糟。

我们在MacBook Pro上，也就是在我们的开发机上通过公司网络展示这个页面，当然看起来一点问题都没有。但是我们不能在我们预定的性能要求底限下工作（例如服务层协议或者SLA）。同时，我们甚至不能在非自己的设备上测试，最重要的是，我们不能违背SLA性能协议，至少我们不能比现有的主页性能低，毕竟在我们的性能监控平台上不能报警。在第3章中我们将详细讨论这个问题。

从最初到响应式设计到现在我们经历了哪些

早在2008年左右，也就是在响应式设计出现之前，我们需要维护两个URL：mysite.com和m.mysite.com（我们的“m.”站点），在同样的Web App中，每一个站点都需要有不同的页面甚至不同的App来适配。它们可能由不同的团队负责，在当时，我们的想法很前卫，也很罕见，并且已经有了手机页面在使用了。

然后，在2011年，Boston Globe项目重新启动，响应式设计和渐进增强的想法已经成为了每次发布的博客和头脑风暴会议的主题了。我们阅读了关于如何创建自适应用户设备的响应式页面的文章。而且我们都被这些概念和想法深深地迷住了。从2000年以来，那些利用相对高度和宽度创建流式布局的的忠实拥
趸，一开始并没有感觉什么不同，但是当他们看到字体大小和图片同样可以被缩放的时候，他们也被这种新的想法深深地吸引了。

业界关于这方面的探讨也如火如荼地开展起来，有很多书出版了，演讲会也开展了起来，每个人都开始制作响应式网站。我们也开始讨论和使用媒体查询来封装不同尺寸屏幕的样式，而且尝试使用多种方法缩放图片。

当我们在真实项目中运用这些新的想法时，都应该从最小的屏幕尺寸开始，并且在打下坚实的基础后，逐步提高。事实上，我们的老板只想看到最终完整的版本（比如桌面版），这样，他们就可以向公司领导层邀功了。所以设计团队首先开工了，我们所有人都为实现这个版本而努力。但是我们可以巧妙地使用媒体查询来控制CSS来达到降低视觉体验的差异，看起来一切都可以解决的，对吧？

我们基于CSS和JavaScript来完成桌面版（最多只有几百KB的大小），然后加载到智能手机和平板电脑上，通过前端的功能我们可以确定客户端

的详细信息。当一切都完成以后，我们可以向我们的老板展示，我们的老板也会向公司高层汇报，在一切就绪以后，项目就可以发布上线了。不可避免地，这样开发多少会造成代码的问题，以至于我们不得不花一些时间来进行重构。因为我们的CSS是基于桌面版的，是的，我们的所有link都最终连接到我们的桌面版代码中。然而，我们不能讳疾忌医，我们必须持续重构。因为我们产品的迭代速度非常快，我们需要资源来进行培训。

项目已经在运行中，但是现在的问题是，我们只关注前端样式是不是好看，页面是不是美观。媒体查询和图片自动缩放看起来很酷，但是，如果只关注这些表面的东西，我们就会从本质上错过了根据用户当前的设备来进行整体体验优化的机会，这不能被称作真正的响应式。我们不仅仅关注前端页面是怎么展现的，我们还把我们所有的判断逻辑都放置于前端。仅仅依赖于媒体查询来实现不同设备上的解决方案，或者通过JavaScript来测试前端，意味着我们向客户端传送了额外的文件。这种反模式的方法是我们已经察觉到的，如图1-13所示。

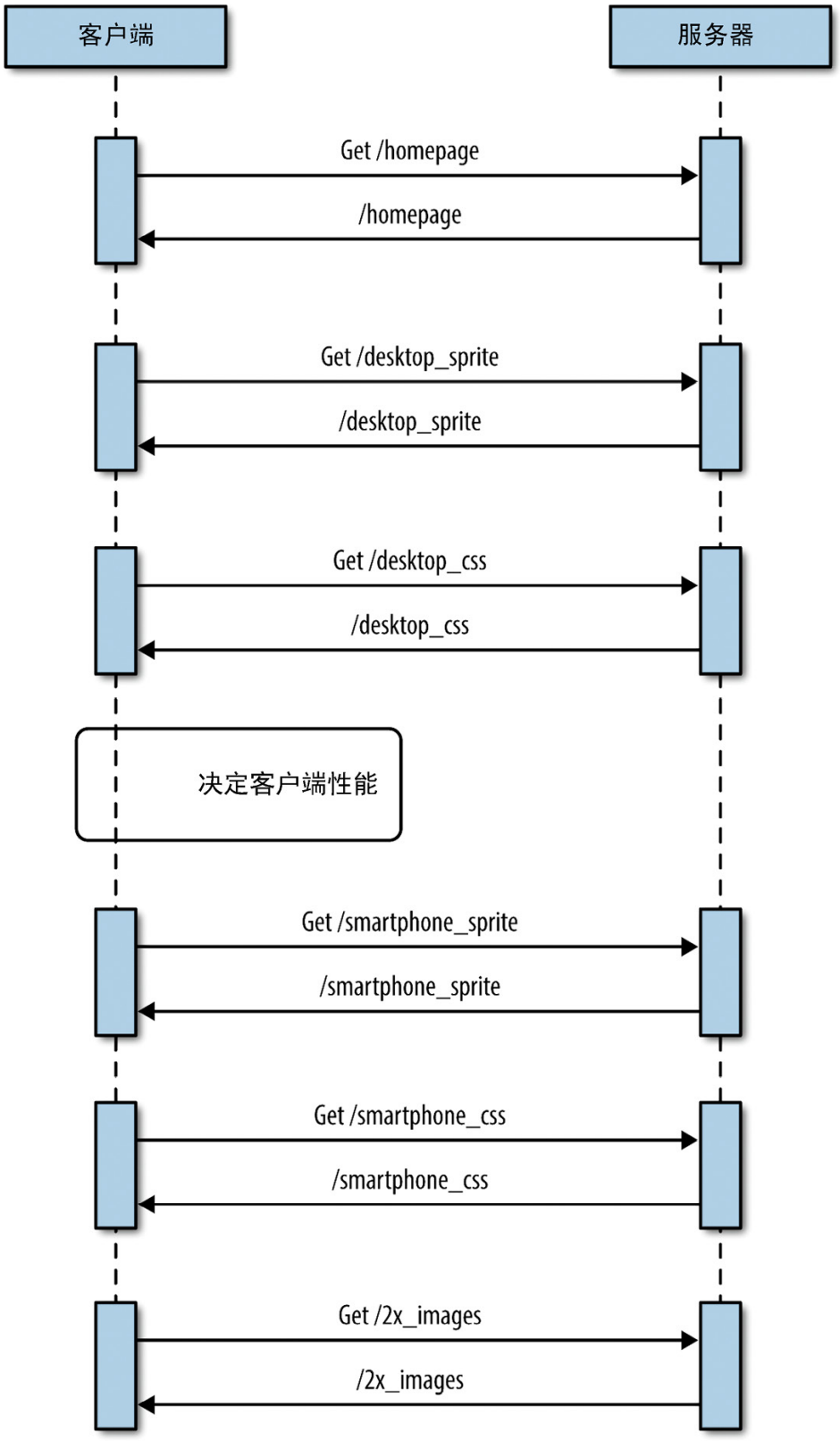

图1-13　反模式的时序图

不同设备之间的差异性，包括处理器能力、电池寿命和内存大小都会在我们将注意力集中在前端或者页面样式的时候被忽视。在实际项目中，这些都是我们需要在响应中应该注意的因素。这就是现在网络上主要的门户仍然维护着专有“m.”站点的原因。

为什么不使用“m.”专有站点

到目前为止，我们都在讨论“m.”站点的好处，那你也许要问了，既然它这么好，我们没有理由不用它啊。这里需要申明的一点是：我并不认同“m.”。它当前的确在人们使用响应式设计的时候带来显著的性能提升。但是它同时也有一些缺点。

资源开销

早在2000年的时候，当我创建我的第一个“m.”网站的时候，我必须让一个全新的工程师团队来开发和维护它。这主要是因为我们的产品团队并不想在建立主站的时候，影响移动设备的体验，这还因为移动站点是一项极其复杂和耗费精力的工作，我们不仅需要在主流的iOS和Android设备上支持它，还需要在其他操作系统的手机上支持它，这些手机都有不同大小的屏幕和特定的属性。包括对JavaScript甚至JavaScript的基本功能都支持的限制。

即使你不会使用一个单独的团队来维护，你仍然需要把你主站的相关工作，也就是“m.”网站，作为一个单独的项目来持续维护，事实上，某些功能在某些类型的手机上压根就不支持。

分散的代码

维护一个单独的网站也就意味着你需要维护一个单独的Web App和单独的一份代码。保持代码之间的同步是一个由来已久的问题，当前最主要的解决方法就是个人的自觉性和流程上保证，也就是说，最终它会变得混乱不堪和杂乱无章，当我们的代码没有办法进行同步，我们就失去了对代码的控制，我们最后很可能看到的是两个不同的网站，将来我们会花费更多的精力在更新上。

分段独立的URL

使用一个单独的“m.”网站也就意味着你需要创建和维护一个单独的URL，这和我们将一个资源作为一个统一的管理的整体思想相违背。对网站来说，一个“m.”就是第二个站点，此外，如果这样的话，我们的“m.”站点的未来将何去何从？它的底线在哪里呢？你会把它放到功能手机里面

吗？或者智能手机？又或者平板电脑？那么平板电脑会遇到什么情况呢？它们都会进入到同一个“m.”站点吗？或者你能为不同的尺寸和特性分别维护一个单独的网站？你很快会发现，这种分割很快就会让你疲于应付、苦不堪言。

毫无意义的重定向

使用一个完全独立的URL也就意味着客户端进入网站的时候需要进行重定向。额外增加一个重定向也就增加了不必要的延迟体验，因为服务器需要向客户端返回302或者304状态码，然后客户端必须重新向一个新的地址发送新的请求，就像图1-14展现的那样。

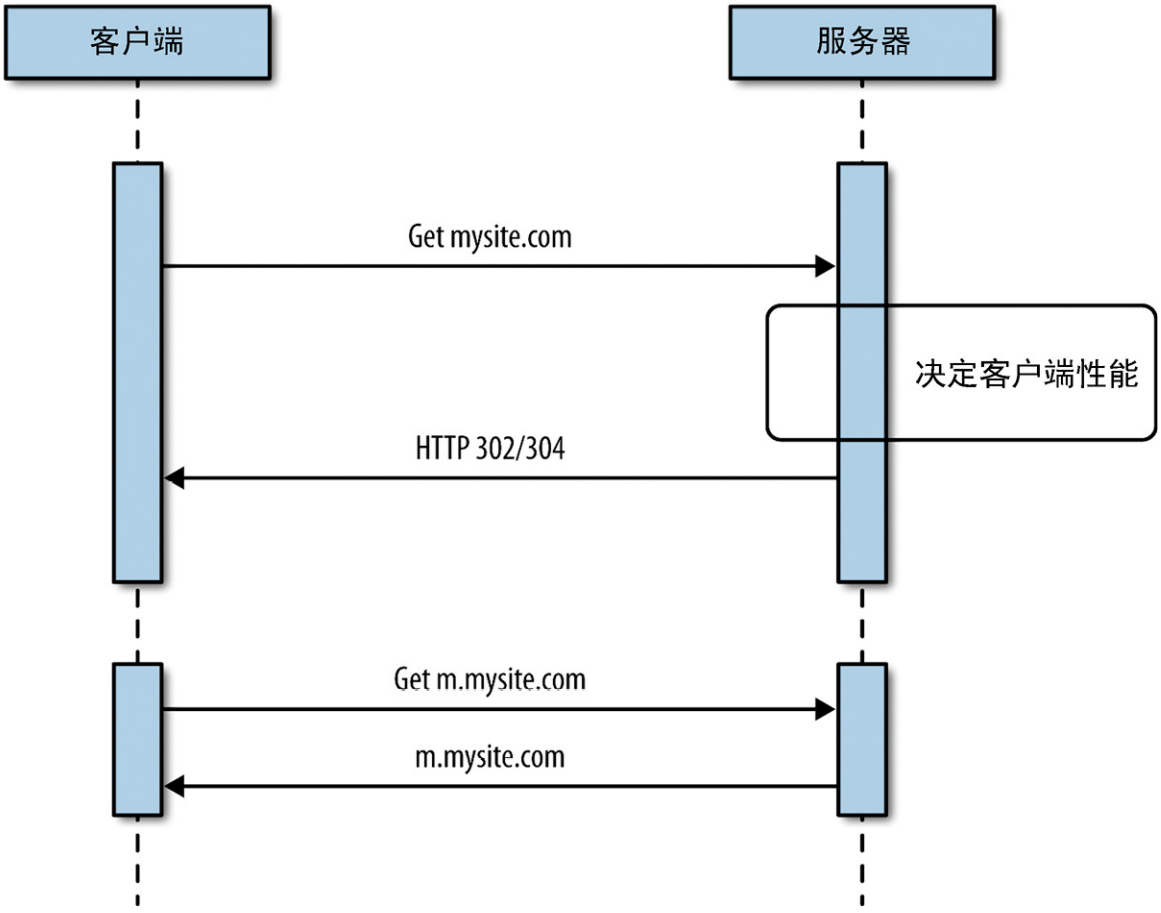

图1-14　在手机站点上引入HTTP重定向的单独URL

可扩展性带来的问题

到目前为止，我们主要都在讨论智能手机和桌面的体验问题，和平板一起，构成我们现在所知道的主要设备类型。但是工业的发展日新月异，

在过去的几年里，我们看到过非常多的新设备和它们特有的功能，包括网络基础设施和客户端的一系列的特色。

举个例子，当苹果公司发布了Retina屏幕的时候，我们必须和设计团队一起创建独特的图片，让它们在Retina屏上也有不俗的视觉效果。随着Web开发在电视指南和App中显山露水，以及分辨率为4K和8K的超高清电视机的屏幕不断增大，这一趋势仍将持续。

另外一个例子就是谷歌眼镜现在变得越来越普及，所以我们也要考虑我们的站点在眼镜上的体检。现在Google提供了一套名叫Mirror的API，从而使得客户端库和Mirror API可以整合到一起（*http://bit.ly/1rXkSpb*）。

这些只是一些处于科技前沿的便携设备，越来越多的新科技将会涌现。如果我们继续把响应式设计看作一个前端的工具的话，那么我们将会发现页面臃肿的问题会越来越严重，也许我们会看到越来越多的公司将重新回归到“m.”站点中去。

1.2 小结

工业的发展正在逐步影响响应式设计。在我统计过的站点中，几乎有一半的站点使用了自己专有的体验——和我们早在21世纪头几年使用的解决方案一样，而不是提供响应式站点。

响应式设计并不是一个有缺陷的方法论，只有在它被误用为一个附加的功能而非首要原则的时候，才会导致一种臃肿的、违反直觉的体验。同样，只有当我们将注意力集中到响应式的某一个方面，特别是前端特效时，我们才会失去对我们的响应式站点性能的敏锐观察力。然而，性能是响应式设计的一个重要的方面，作为交互的一个重要方面，需要慎重计划和设计。当我们在架构解决方案时，就要以此为基础。

在本章中，我们已经介绍了如何使用一些设计模式来影响响应式的性能。在接下来的章节中，我们会进一步介绍这些模式，并提出更多的模式。

如果我们不做这些功能并且在构建我们的响应式解决方案中考虑到性能问题，那么，当新的产品和设备带来越来越多的新特性后，越来越多的基于特定设备上的问题都需要解决，这会导致我们需要为每一种设备创建它独有的客户端交互方式。

第2章　初识Web应用性能

2.1　性能度量基础

如果你正在阅读本书，很可能你已经熟知性能的含义，或是至少曾经围绕你的Web应用做过一些性能相关的讨论。但在继续往下看之前，我们来确认下在术语方面我们的理解是一致的。

如若这是你首次听到Web性能优化一词，赶快去搞一本Steve Souders的*High Performance Web Sites*和*Even Faster Web Sites*（均由O'Reilly出版）读一读。这些都是Web性能的标准，也是所有Web开发者都应掌握的基础知识。

本章并不打算覆盖性能的每个细节。从前面提到的Steve Souders的著作开始，已经有大量的资料在讲这些东西。但是，本章的目标是对性能（Web性能和Web运行时性能）做个概述，包括一些性能度量的工具。这样一来，后面章节我们说到这些概念时，就不会再有歧义和混淆了。

当提及网站和Web应用性能的时候，我们说的要么是Web性能，要么是运行时性能。我们将Web性能定义为，一个终端用户从请求一段内容开始到这段内容显示在用户设备上这段时间的度量值。我们将运行时性能定义为，应用在运行时对用户输入响应能力的一个标示。

应当意识到，针对你的Web应用性能进行量化和标准的制定，是应用的一个关键方面。Web性能和运行时性能都有一些指标可以进行实证测度和量化。本章中，我们来看一看这些指标，以及可以用来量化这些指标的工具。

> **注意**
>
> 性能指标是组织机构用来定义一次尝试是成功或是失败的可量化目标。通常称作关键性能指标，缩写为KPI。

本章我们将谈到的性能指标类型如下所示。

定量指标

可以通过实验进行度量的一种目标（如某个东西的数量）。

定性指标

不能通过实验度量的一种目标（如某个东西的质量）。

先行指标

用于预测结果。

输入指标

用于度量某个过程中消耗的资源。

什么是Web性能？

想想每次浏览网页的过程。打开浏览器，键入URL，然后等待网页加载。从键入URL后按下回车键（或是从书签列表中点击某个书签，亦或是点击页面中的某个链接），到页面渲染，这之间消耗的时间就是所浏览页面的Web性能。若站点运行正常，这个时间甚至不应该被人感受到。

Web性能的定量指标数不胜数。

- 页面加载时间。
- 页面文件大小。
- HTTP请求数。
- 页面渲染时间。

Web性能的定性指标总结起来比较简单：速度感。

在看这些指标之前，先来讨论下页面是如何到达浏览器并展现给用户的。当通过浏览器请求一个Web页面，浏览器会创建一个线程去处理这个请求，随后开始远程dns查找，远程dns服务器会将你输入的URL对应的IP地址返回给浏览器。

接着，浏览器通过与远程Web服务端的三次握手来建立一个TCP/IP连接。这个握手由浏览器与远程服务端之间的SYN、SYN-ACK以及ACK消息组成。

TCP连接建立之后，浏览器通过连接发送一个HTTP GET请求到Web服务端。Web服务端找到请求的资源，然后在HTTP响应中将其返回，状态200表示响应正常。如果服务端找不到请求的资源或是解析资源的过程中出错，抑或是资源被重定向，HTTP响应状态也会反映出这些情况。这个页面可以

找到状态码的完整列表：*http://bit.ly/stat-codes*。下面是最常用的状态码。

- 200表示服务端成功响应。
- 301表示永久重定向。
- 302表示临时重定向。
- 403表示请求被拒绝。
- 404表示服务端找不到请求的资源。
- 500表示处理请求时出错。
- 503表示服务不可用。
- 504表示网关超时。

图2-1是TCP事务的时序图。

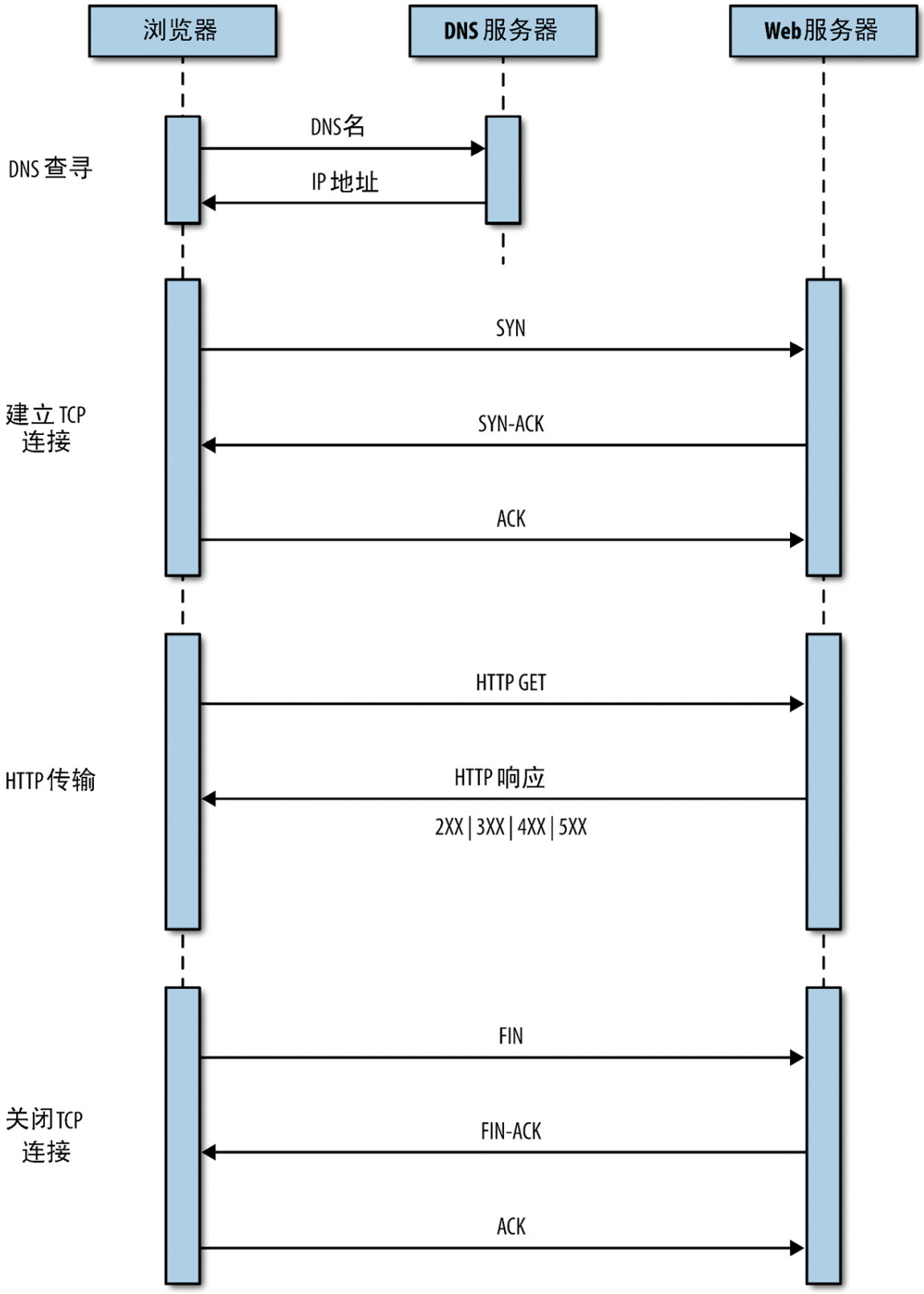

图2-1　浏览器和Web服务器的协商过程

要时刻记得，加载一个HTML页面不只需要一次这个过程，浏览器还要为页面链接的每个资源发起一个HTTP请求——所有的图片、链接的CSS和JavaScript文件以及其他类型的外部资源（但是要注意，只要后续的HTTP请求连接的是相同的源，浏览器就可以重用相应的TCP连接）。

当浏览器收到页面的HTML后，就开始解析并渲染页面内容。

浏览器用其渲染引擎来解析和渲染内容。现代的浏览器架构由几个关联的模块组成。

UI层

这一层为浏览器绘制界面。有些元素，如地址栏、刷新按钮以及用户界面上（UI）的其他元素是浏览器自身的。

网络层

该层处理网络连接，承担的职责有建立TCP连接以及处理HTTP的往返过程。网络层处理内容下载，然后将内容传递给渲染引擎。

渲染引擎

渲染引擎负责将内容绘制到显示器上。浏览器制造商会将他们的渲染引擎以及JavaScript引擎打上商标并对外授权。所以，相对流行的渲染引擎你可能已经听说过了。可以说，最流行的渲染引擎是WebKit，Chrome（Blink是它的译名）、Safari、Opera以及其他一些浏览器中都用到了WebKit。当渲染引擎遇到了JavaScript，会将其传递给JavaScript解释器。

JavaScript引擎

JavaScript引擎会解析并执行JavaScript。如同渲染引擎，浏览器制造商给他们的JavaScript打上商标进行授权，很可能你已经听说过它们了。一个流行的JavaScript引擎是Google的V8，Chrome、Chromium中都用到了它，Node.js就是用它作为引擎的。

图2-2展示了这样的架构。

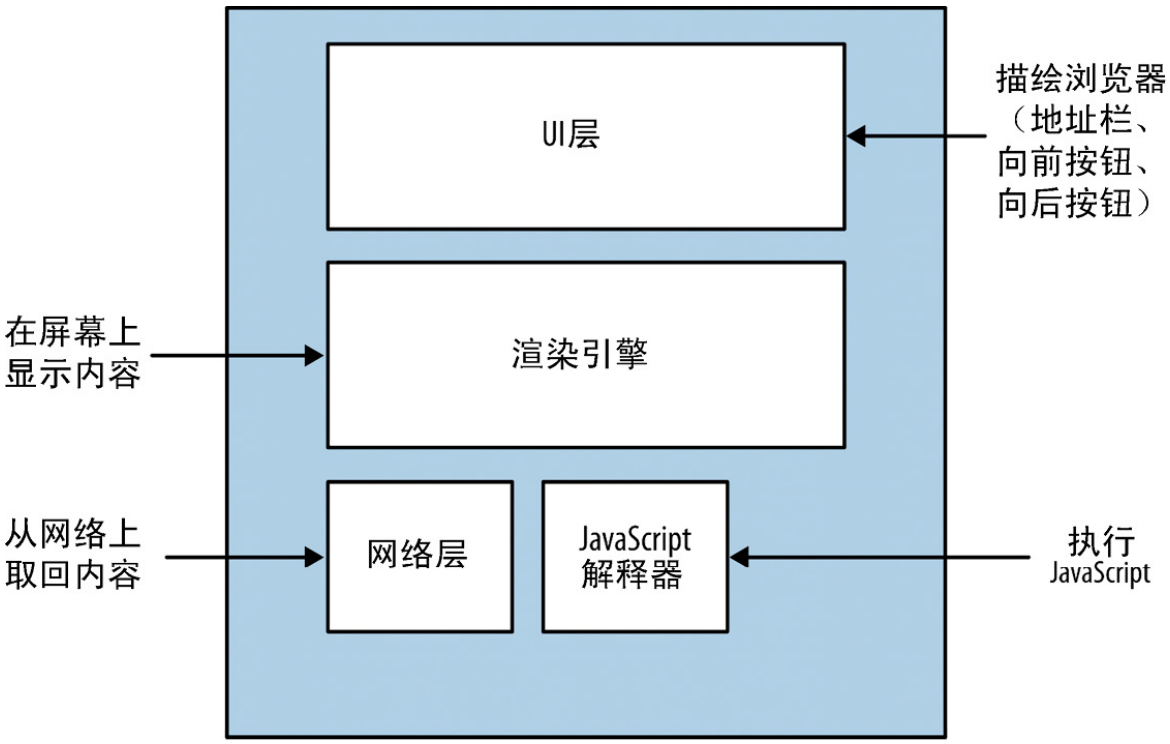

图2-2　分成多个模块组件的现代浏览器架构

设想这样一个用例，用户在浏览器地址栏里键入一个URL。UI层将这个请求传递到网络层，网络层随后建立连接，然后下载初始页面。当含有HTML块的数据包到达，就被传递给渲染引擎。渲染引擎将HTML组装成原始文本，然后对文本中的字符开始进行词法分析——或解析。这些字符会与一个规则集相比较——我们在HTML文档中指定的文档类型定义（DTD）——然后转换成基于规则集的符号。DTD规定了一系列标签，这些标签组成了我们将要使用的语言版本。这些标签就是由一些被分隔成有意义片段的字符组成。

这里有个网络层是如何处理并返回下列字符串的例子。

```
<!DOCTYPE html><html><head><meta charset="UTF-8"/>
```

这串字符会被分割成下面这样有意义的块。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8"/>
```

渲染引擎拿到这些符号后将它们转换成文档对象模型（DOM）元素（DOM是页面元素的内存表现形式，也是JavaScript用于访问页面元素的API）。DOM元素被布局成一棵渲染树，渲染引擎会迭代该树。首次迭代中，渲染引擎会布局好DOM元素的位置；下一次迭代就将这些元素绘制到屏幕上。

如果渲染层在解析和符号化过程中发现了script标签，就会暂停下来然后评估下接下来要进行的处理。如果script标签指向一个外部JavaScript文件，解析过程暂停，随后网络层介入，下载JavaScript文件，然后初始化JavaScript引擎解析，执行该文件。如果script标签包含的是内嵌的JavaScript，渲染引擎暂停，JavaScript引擎被初始化，内嵌的JavaScript会被解析与执行。执行完成后，之前暂停的渲染过程会恢复运行。

这是一个很重要的细微差别，影响的不仅仅是DOM元素何时对JavaScript可见（我们的代码可能会尝试访问页面上的一个元素，但该元素可能还没有被解析和符号化，更不用提渲染了）和性能。例如，我们是想要阻塞页面的解析直到下载并运行了那一段代码吗，或是如果我们先展示内容再去加载页面的代码，页面的功能能否正常？

图2-3描绘了这种工作流程。

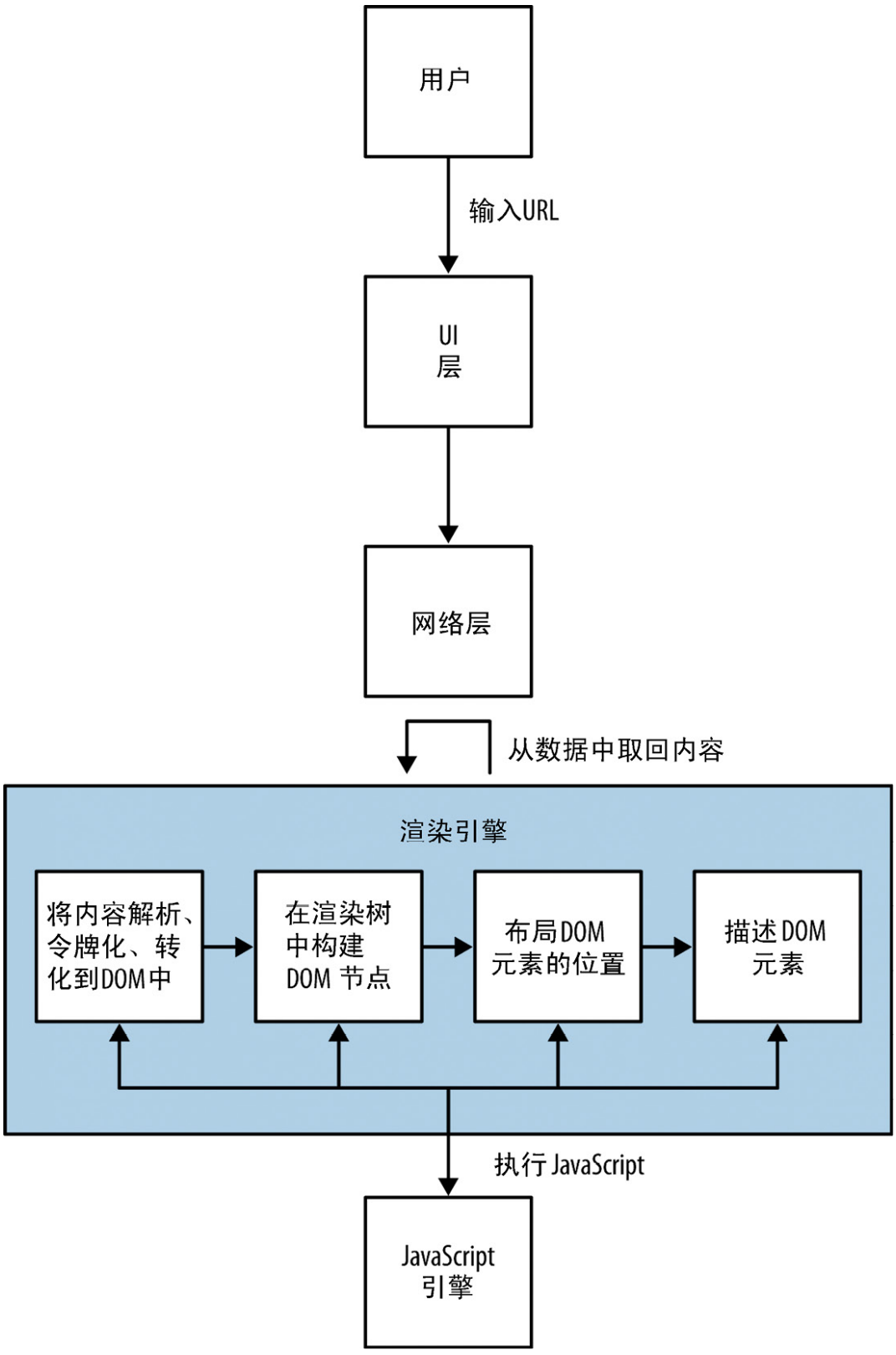

图2-3　浏览器中内容加载和渲染的工作流程

了解网页内容是怎样传递给浏览器对于理解Web性能影响因素是至关重要的。也要注意到，由于浏览器的快速更新，浏览器厂商可能会时常对这个工作流进行调整和优化。

既然我们已经了解了内容传递和展现的架构，那来看一看在这个传递工作流上下文中的性能指标。

HTTP请求数

时刻记住，当浏览器获取HTML页面时会创建一个HTTP请求，还会创建更多的HTTP请求来获取页面中链接的每个资源。根据网络延迟情况，每个HTTP请求都会使总的页面加载时间增加20～200毫秒（如果考虑可以并行加载资源的浏览器，这个数字会有所不同）。如果只是少量的资源，这些额外的加载时间可以忽略不计，但如果是100个或更多的HTTP请求，将会显著加大总体Web性能的延迟。

减少页面需要的HTTP请求数才有意义。开发者有很多办法可以做到，如将不同的CSS或JavaScript文件合并成单个文件，将常用的图片合并成单个的称之为sprite的图片文件。

页面负载

影响Web性能的因素之一是页面的总文件大小。总负载包括组成该页面的HTML、CSS以及JavaScript累计的文件大小。还包括所有的图片、cookie以及其他嵌在页面中的媒体。

页面加载时间

HTTP请求数以及总的页面负载本身只是输入，但Web性能方面需要关注的真正KPI是页面加载时间。

页面加载时间是最明显的性能指标，也是最容易量化的。简而言之，它是浏览器下载并渲染所有页面内容的时间。以前，度量的是从页面请求到页面（窗口加载，window onload）事件之间消耗的时间。最近，由于开发者越来越喜欢在页面完成加载之前就提供好的用户体验，这样度量结束的时间点就会移动，甚至完全改变。

特别是，在有一些用例中，window.onload事件触发之后，可以动态加载内容——比如，如果内容是延迟加载的，就会出现这种情况——并且

有一些用例，在window.onload事件触发之前页面就是可用的，看起来也是完整的（如先加载内容，然后再加载广告）。这些用例会降低依靠window.onload事件追踪特定页面加载时间的有效性。

有一些选择可以规避这个难题。WebPageTest的创建和维护者Pat Meenan，在WebPageTest中加入了一种度量方式，称为速度指标（Speed Index），其实质上是对页面内容渲染快慢计分。有一些开发团队创建了自己自定义的事件，来追踪他们觉得用户体验关键页面的各个部分是何时加载的。

无论你选择何种方式进行追踪，页面的加载（即，你的内容已经准备好接受用户交互）都是应当监控的关键性能指标。

2.2 追踪Web性能的工具

追踪Web性能最常用和最有用的工具非瀑布图莫属了。瀑布图非常直观，可以展示构成Web页面的所有资源、加载这些资源所需的所有HTTP事务，以及每个HTTP请求消耗的时间。所有这些HTTP请求都展示成条状，一般y轴是资源的名称或URL。有的时候，资源的大小和资源的HTTP响应状态也会展示在y轴。x轴，或显式或隐式地展示出所消耗的时间。

瀑布图中的条形是按请求发生的顺序绘制的（见图2-4），条形区块的长度表示完成事务耗时的长短。在瀑布图的底部有时候也会看到总页面加载时间以及总HTTP请求数。瀑布图的妙处之一是，从条形的排列和重叠我们也可以发现某些资源的加载是不是阻塞了其他资源的加载。

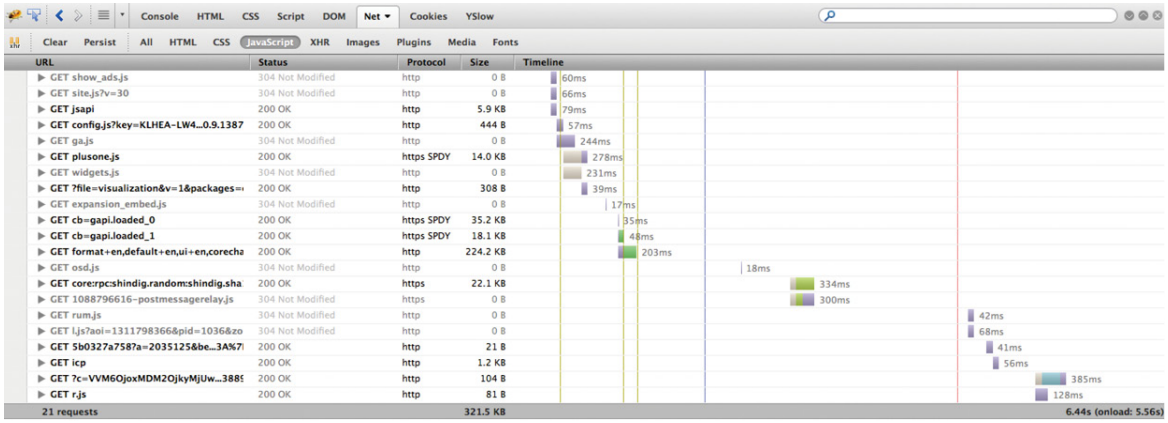

图2-4 Firebug生成的瀑布图

现今，有许多不同的工具可以为我们创建瀑布图。有些浏览器提供了内置的工具，比如Firefox中的Firebug，或是Chrome的开发者工具。也有一些免费、托管的解决方案，比如*webpagetest.com*。

我们来看几个这样的工具。

最简单的生成瀑布图的方法是用浏览器自带的工具。这类工具有好多种，但在这一点上或多或少都有些类同，至少在生成瀑布图的方式上是相似的（有些浏览器内置工具远比其他一些工具有用，这一点在我们讨论Web运行时性能的时候将会看到）。

Firebug是首个广为采用的浏览器内置开发工具。以Firefox插件形式存在，由Joe Hewitt初创，Firebug确立了一项标准，不仅仅可以创建瀑布

图来展示用于加载和渲染一个页面的网络活动，还让开发者可以通过控制台运行任意的JavaScript并展示错误，以及调试与单步调试浏览器中的代码。

如果你还不熟悉Firebug，可以访问*http://mzl.la/1vDXigg*进行安装。点击“Add to Firefox”按钮，然后按照说明操作即可成功安装这个附加组件。

> **注意**
>
> 其他浏览器也有可用的Firebug，但一般都是简化版，没有提供像在Firefox中那样完整的功能。

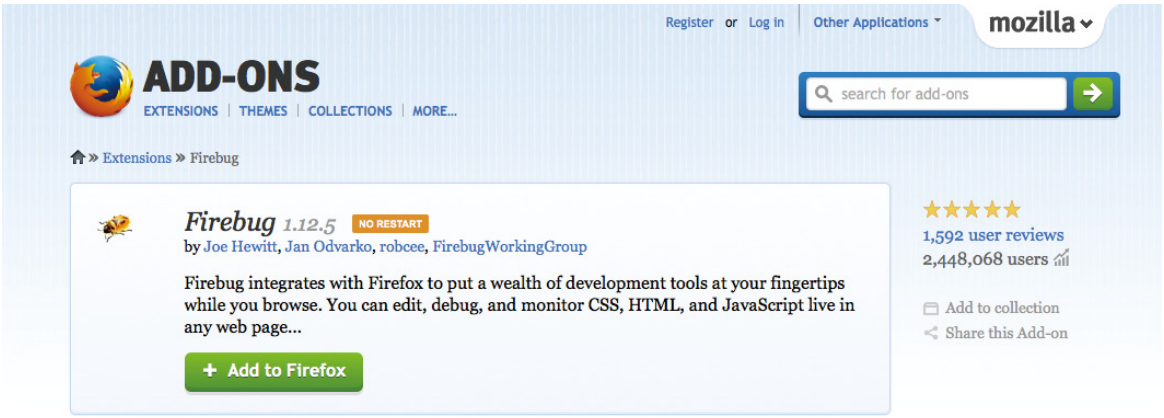

图2-5　Firebug下载页

要在Firebug中浏览瀑布图，点击“网络”（Net）选项卡。

自Firebug出现以来，浏览器一直在发展，到现在，最新发布的浏览器都内置了一些工具，至少可以度量性能的某些方面。Chrome有开发者工具，Internet Explorer也有自己的开发者工具，Opera有Dragonfly。

要访问Chrome中的开发者工具，可以点击Chrome菜单图标，选择“工具-开发者工具”，如图2-6所示。

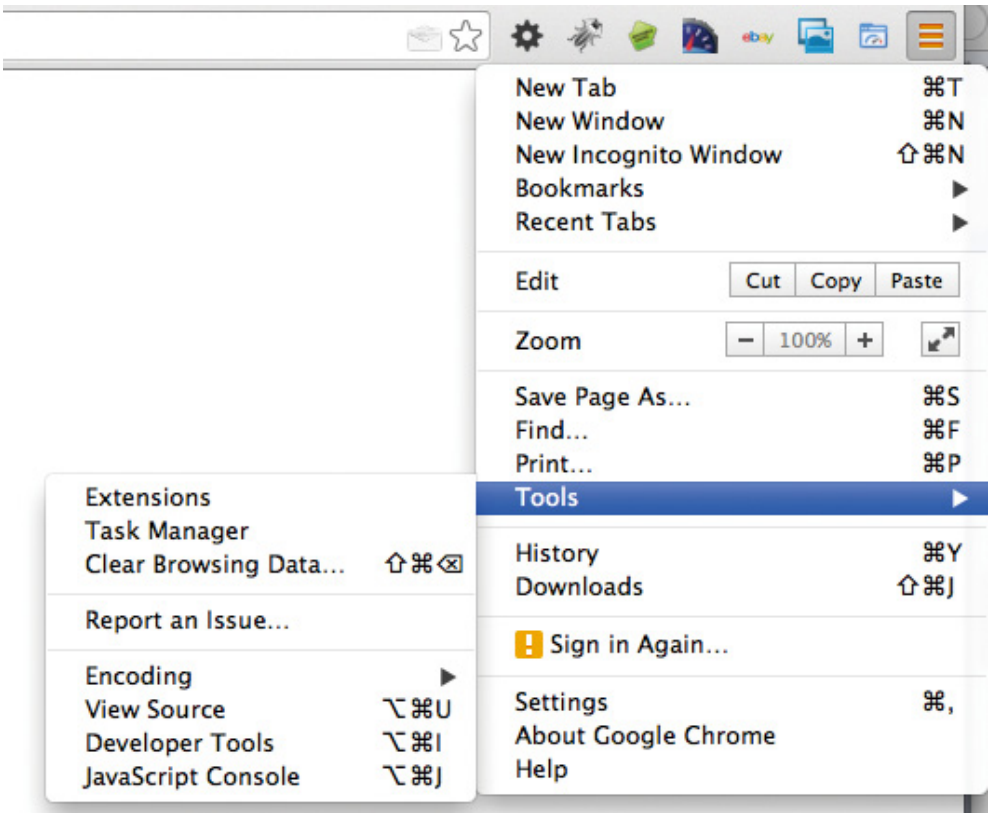

图2-6　访问Chrome中的开发者工具

在Internet Explorer中，选择“工具-开发人员工具”。

甚至在移动设备中，现在也有原生应用HTTPWatch，可以在其中运行一个浏览器，然后展示所有被加载资源的瀑布图。HTTPWatch可在*http://bit.ly/1rY322j*里下载。图2-7和图2-8可以让你对运行中的HTTPWatch有个粗略的认识。

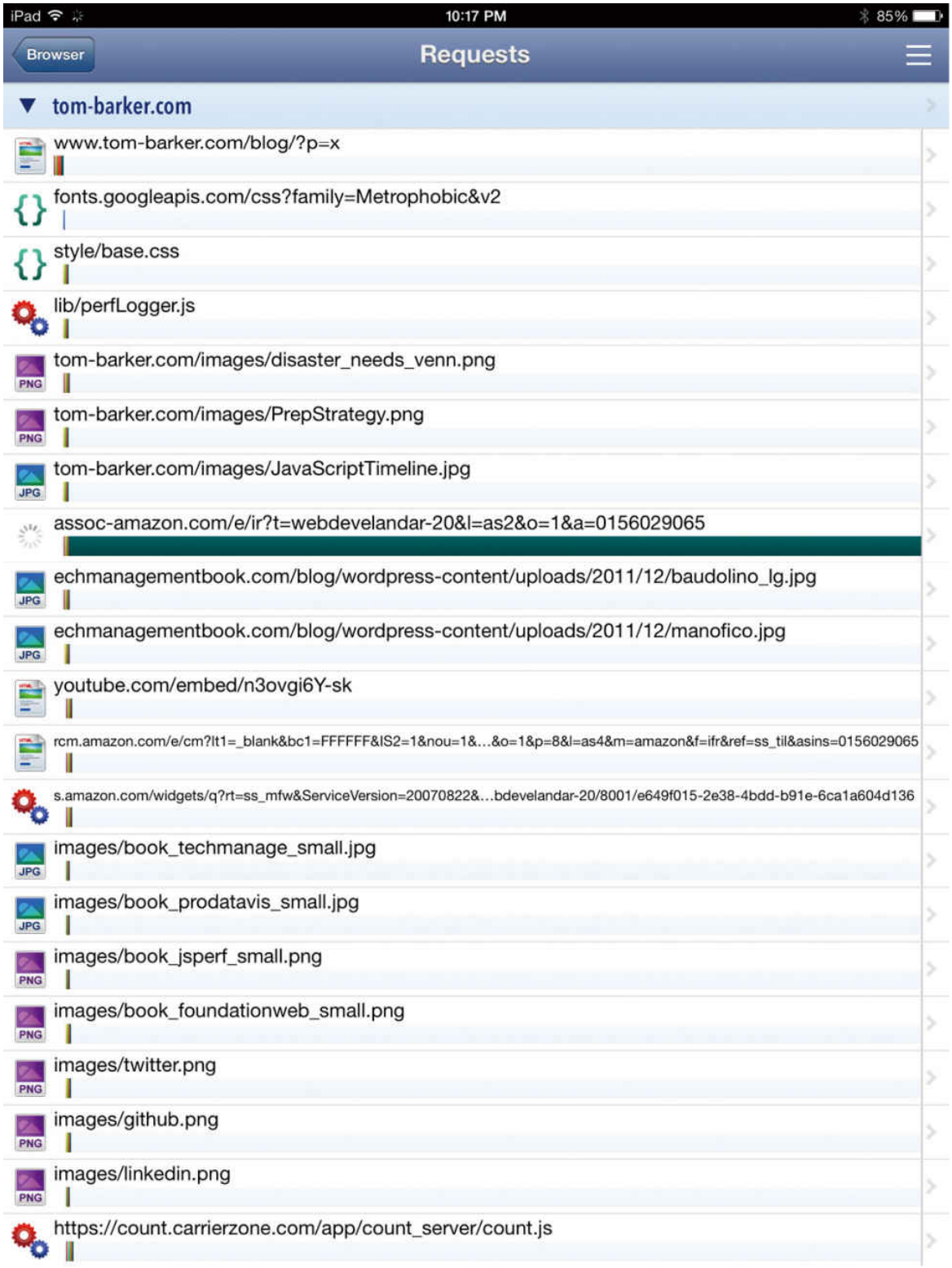

图2-7　iOS 7上HTTPWatch中的资源加载过程

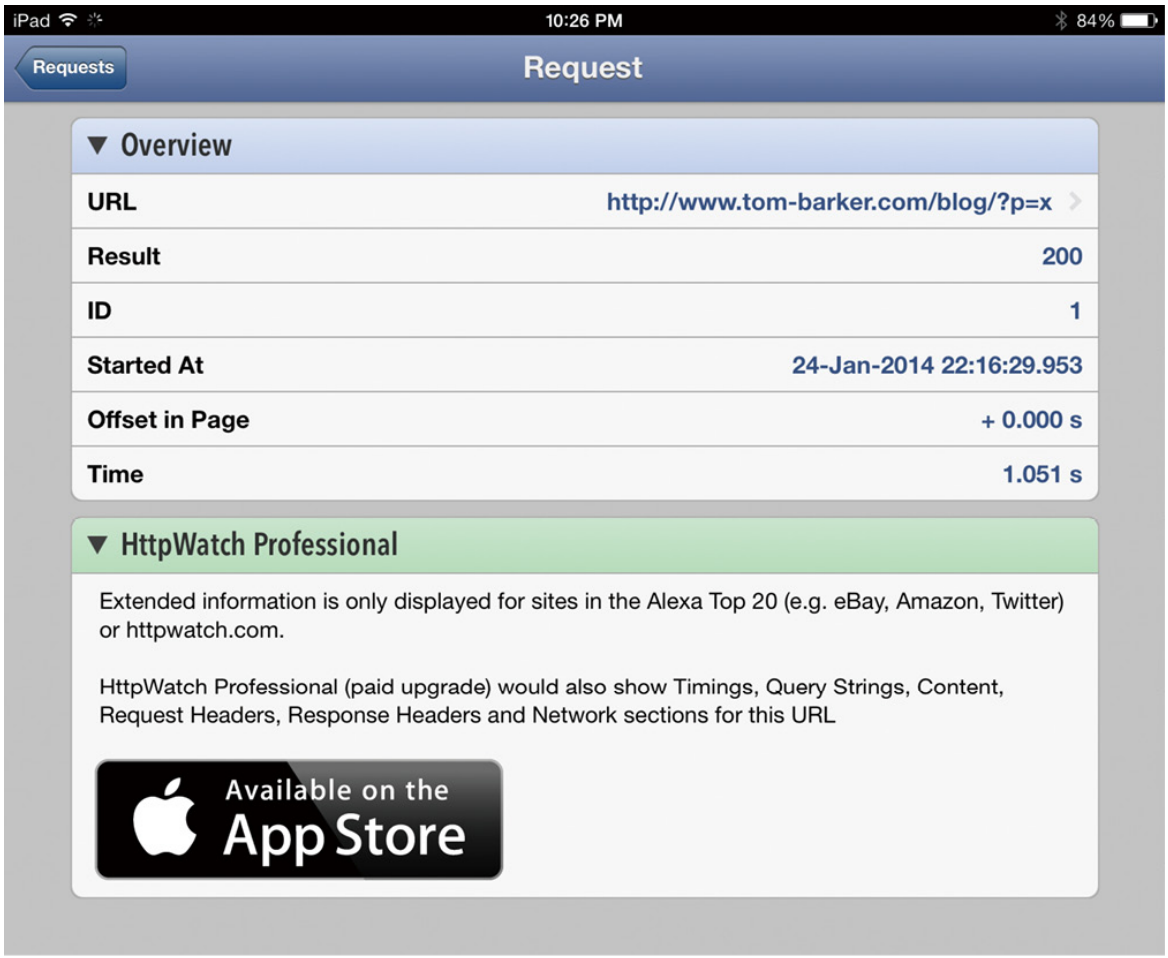

图2-8　iOS7上HTTPWatch中的Web性能信息

用浏览器内置的工具进行调试是非常不错的，但是如果你想找一些在持续集成环境中能用的自动化解决方案，就需要扩大选择范围了，包括平台解决方案或无外设的解决方案。

> **提示**
>
> 第6章我们将详细讲解无外设的测试以及持续集成。

如前面所提到的，WebPageTest（*www.webpagetest.org*），是处于领先地位的解决方案之一，由Pat Meenan创建并持续维护。WebPageTest是一种托管解决方案，或者开源工具，你可以安装然后在你的网络上作为一个本地副本运行来测试你的防火墙。下载和托管的代码库：*http://bit.ly/1wu4Zdd*。

WebPageTest是一个Web应用，它的输入是一个URL（以及一堆配置参数），然后对这个URL运行性能测试。我们能配置的WebPageTest参数的数

量相当多。

可以从一组全球范围的地点中选择一个来运行你的测试。每个地点都有一或多个浏览器供测试选择。还可以指定连接速度以及要运行的测试数量。

WebPageTest提供了有关站点整体性能的丰富信息，不仅包括瀑布图，还有展示给定页面的内容分布图表（负载的百分之多少由图片组成，多少由JavaScript组成等），模拟页面加载到终端用户的体验截屏，甚至还有CPU使用率，这个在本章后面会详讲。

最重要的是，WebPageTest是完全可编程的。它提供了一个API，可以获取所有这些信息。图2-9展示了WebPageTest中生成的瀑布图。

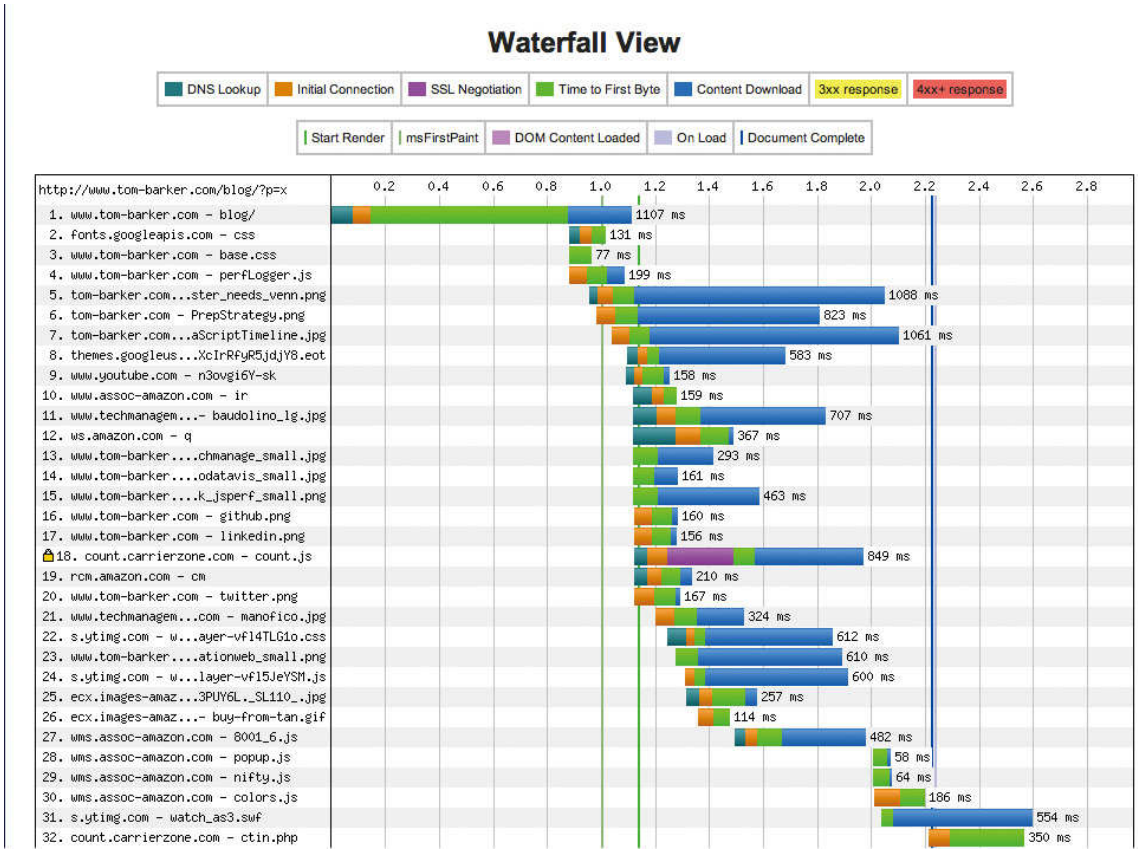

图2-9　WebPageTest生成的一个瀑布图

但在看Web性能指标时，要看的理想值是从你自己的用户那里得到的真实用户监控（Real User Monitoring，RUM）结果。要实现这个目标，需要一个完全可编程的解决方案，万维网联盟W3C已经制定了一个标准API，可

以用来捕获和报告浏览器内的性能数据。这是通过性能DOM对象（在所有现代浏览器中，这是一个属于window对象的对象）来实现的。

2010年末，W3C创建了一个新的工作组，简称为Web性能工作组（Web Performance Working group）。据其网站描述，这个工作组的任务是提供方法来度量用户代理特性和API的各方面应用性能。从战略意义上这意味着该工作组开发了一个API，可以用JavaScript通过这个API获取关键性能指标。

Google的Arvind Jain以及来自微软的Jason Weber担任这个工作组的主席。可以通过*http://bit.ly/1t87dJ0*访问其首页。

Web性能工作组已经创建了一些新的对象和事件，不仅可以用来量化性能指标，还可以用来优化性能。下面是对这些对象和接口高层面的概述。

性能对象

这个对象对外暴露了多个对象，比如Performancenavigation、PerformanceTiming、MemoryInfo，以及可以记录亚毫米级高精度时间的能力。

页面可见性API

这个接口让开发者具备检测指定页面是处于展现状态还是隐藏状态的能力，这样就可以针对如动画效果或是轮询操作中的网络资源优化内存利用率。

如果你在JavaScript控制台键入window.performance，将会看到返回的Performance类型的对象，这个对象对外暴露了几个对象和方法。截至本书写作时，标准对象集有PerformanceTiming类型的window.performance.timing、PerformanceNavigation类型的window.performance.navigation。Chrome还支持MemoryInfo类型的window.performance.memory。我们将在本章稍后的“Web运行时性能”部分讨论MemoryInfo。

PerformanceTiming对象对监控真实用户指标来说最为有用。图2-10所示为控制台中Performance对象和PerformanceTiming对象的一个截屏。

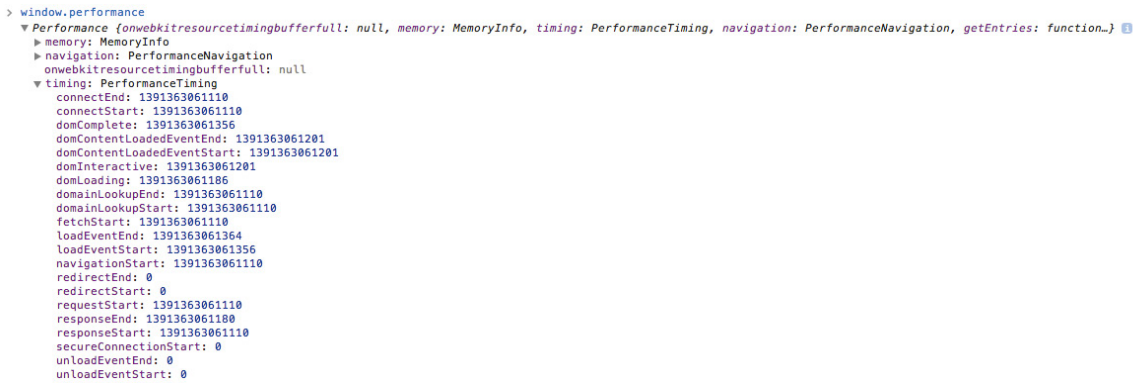

图2-10　控制台中展示的Performance对象及展开的Performance.Timing对象

要时刻记得，真实用户监控的目标是获取真实用户的实际性能指标，而不是实验室中人工测试或通过脚本测试获得的模拟的性能测试指标。RUM的好处在于能获取和分析基于实际用户的真实性能。

表2-1列出了PerformanceTiming对象的属性

表2-1　**PerformanceTiming**对象的属性

属性	描述
navigationStart	在导航开始时采集，要么是浏览器开始卸载前一个页面的时刻（如果有的话），要么是浏览器开始获取页面内容的时刻。这将包含unloadEventStart的时间或fetchStart的时间。想要追踪端到端的时间，通常会从这个时间开始
unloadEventStart/ unloadEventEnd	在浏览器开始卸载以及结束卸载前一个页面时采集（如果同域下有前一个页面需要卸载的话）
domainLookupStart/ domainLookupEnd	在浏览器开始以及完成所请求内容的DNS查找时采集
redirectStart/redirectEnd	在浏览器开始以及完成任一HTTP重定向时采集
connectStart/connectEnd	在浏览器开始以及完成当前页面到远程服务端的TCP连接建立时采集

属性	描述
fetchStart	在浏览器首次开始所请求资源的缓存时采集
requestStart	在浏览器为所请求的资源发送HTTP请求时采集
responseStart/responseEnd	在浏览器首次获取到以及完成接收服务端响应时采集
domLoading/domComplete	当文档开始以及结束加载的时候采集
domContentLoadedEventEnd/ domContentLoadedEventStart	当文档的DOMContentLoaded事件开始以及结束加载（浏览器完成加载所有的内容且运行了页面上所有包含进来的脚本）时采集
domInteractive	当页面的document.readyState属性变为'interactive'，引起readystatechange事件触发时采集
loadEventEnd/loadEventStart	在load事件被触发的时刻之前采集以及load事件触发之后立刻采集

图2-11展示了这些事件的发生顺序。

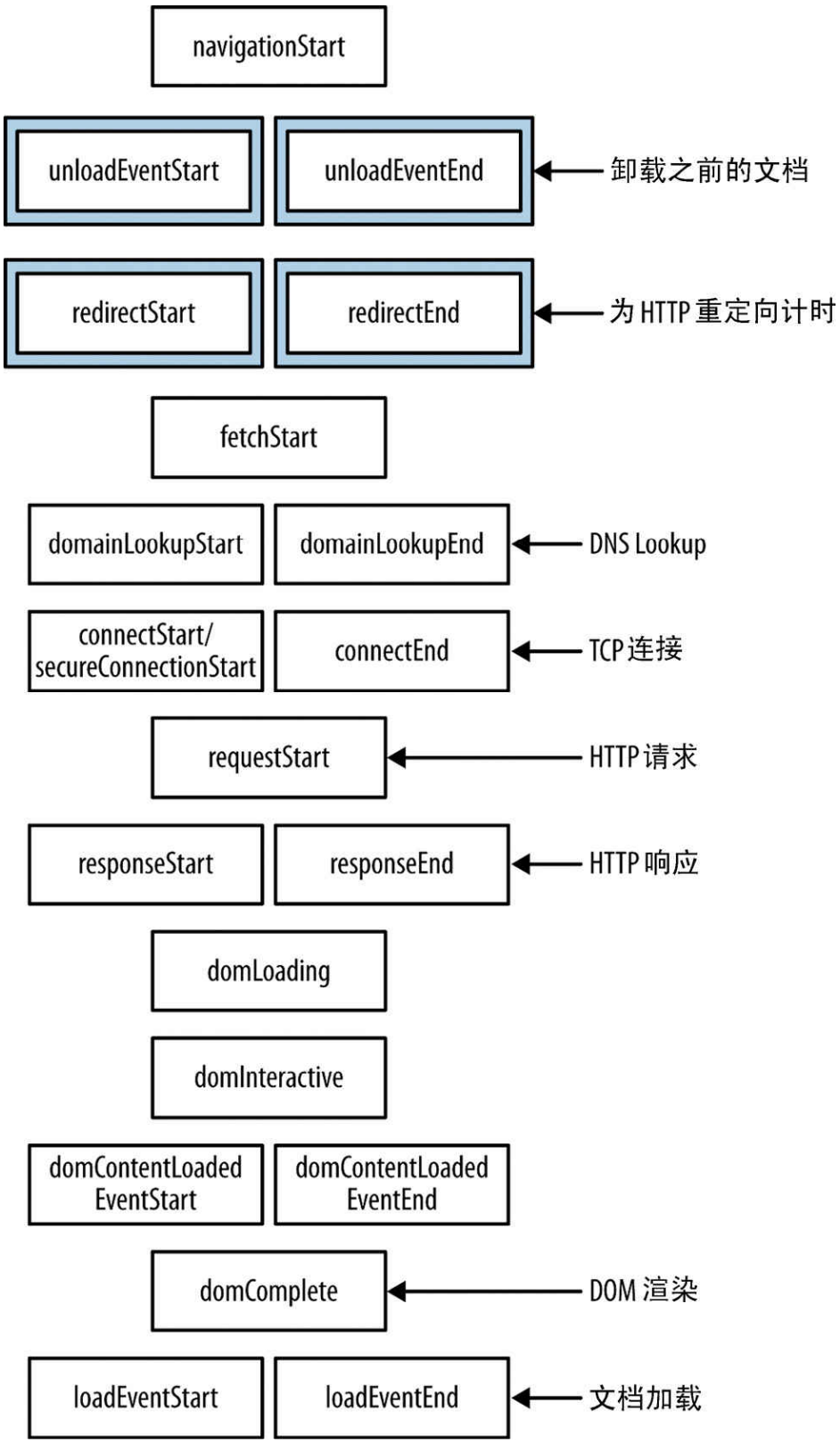
navigationStart
unloadEventStart
unloadEventEnd
卸载之前的文档
redirectStart
redirectEnd
为 HTTP 重定向计时
fetchStart
domainLookupStart
domainLookupEnd
DNS Lookup
connectStart/
secureConnectionStart
connectEnd
TCP 连接
requestStart
HTTP 请求
responseStart
responseEnd
HTTP 响应
domLoading
domInteractive
domContentLoaded
EventStart
domContentLoaded
EventEnd
domComplete
DOM 渲染
loadEventStart
loadEventEnd
文档加载

图2-11　性能计时事件

可以自己写一个JavaScript库嵌到页面中，然后从用户那里采集真实的RUM。本质上是通过JavaScript采集这些事件，再将它们发送到服务端，然后你就可以保存和分析这些指标了。我已经创建了这样一个库 *https://github.com/tomjbarker/perfLogger*，欢迎使用。

2.3 Web运行时性能

正如我们已经讨论过的，Web性能跟踪的是内容传递到用户的耗时。现在来看看Web运行时性能，它跟踪的是用户与应用交互时应用的行为。

对于传统的编译类型应用，运行时性能是有关内存管理、垃圾回收以及线程等各个方面的。这是因为编译类的应用运行在内核之上，直接使用系统资源。

在客户端运行Web应用与运行编译类应用是大不相同的。这是因为Web应用运行在沙盒中，或者，说得更具体点，是运行在浏览器中。当它们运行的时候，用的是浏览器的资源。而浏览器是运行在它事先从内核中分配的内存资源中的。所以，当我们提到Web运行时性能，我们实际上说的是应用是怎样在客户端的浏览器运行，以及让浏览器在虚拟内存中其自身的内存里执行。图2-12是Web应用在常驻内存中的浏览器内存里运行的情况。

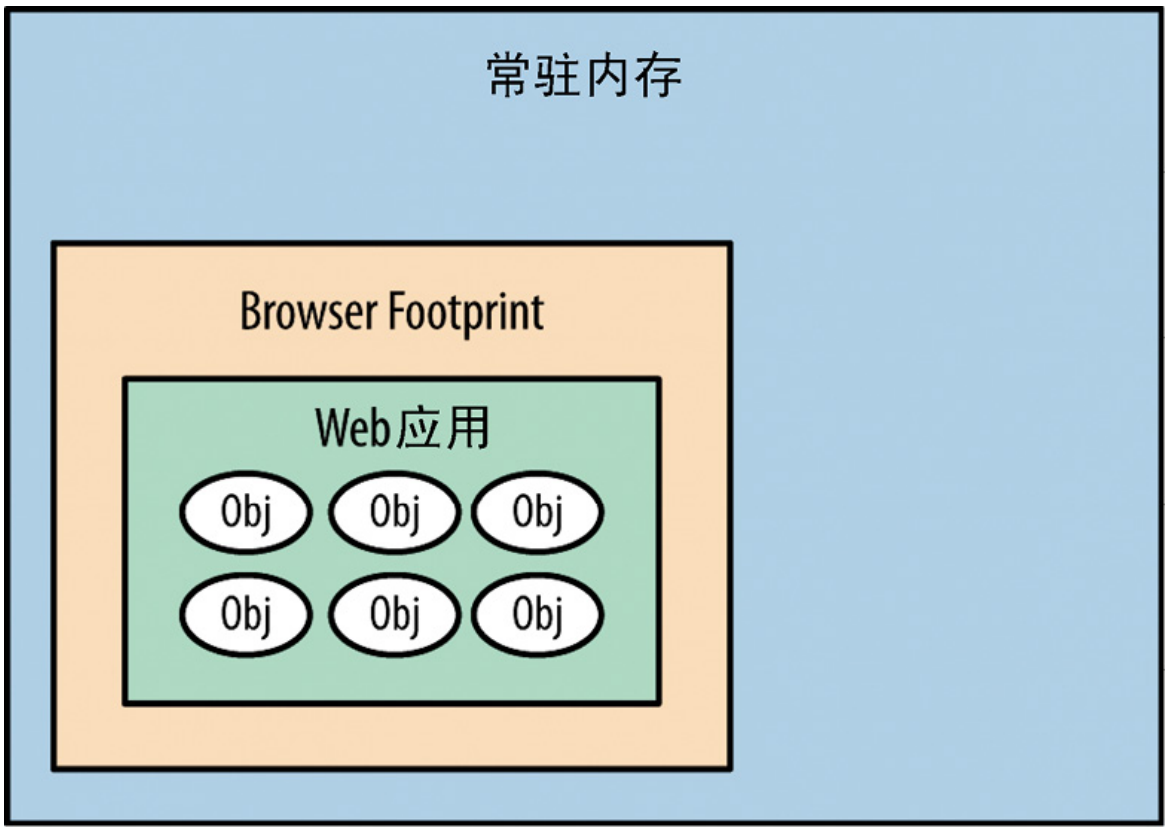

图2-12　一个Web应用运行在常驻内存中的浏览器预分配内存中

下面是我们需要考虑到的影响Web运行时性能的因素。

内存管理与垃圾回收

首先要看的是，我们有没有因为太多无用的对象以及创建更多对象时却仍保留这些无用对象而导致浏览器的内存分配被阻塞。随着时间推移，我们是否有什么机制限制JavaScript中的对象创建，或应用用的越多越久时，内存消耗是否也越多？是否存在内存泄露？

回收无用对象可能会导致浏览器在渲染或播放动画的时候暂停，容易在用户体验上出现锯齿现象。我们可以通过减少创建的对象数量以及尽可能重用已有对象来将垃圾回收次数降到最少。

布局

我们更新DOM的时候是否引发了页面重绘？这一般是由于大范围的样式变化，需要渲染引擎重新计算页面元素的大小与位置。

高代价的绘制

当用户滚动页面时，我们有没有因为绘制一些区域而加重浏览器的负担？动画效果或是更新除了位置、缩放、旋转或透明度之外的任意元素属性，都将引起渲染引擎重绘对应元素并消耗时间。位置、缩放、旋转以及透明度是渲染引擎最后配置的元素属性，所以，更新这些属性只需极小的开销。

如果我们在宽度、高度、背景或者其他属性上使用动画，渲染引擎就需要重新考虑页面的布局并且重绘那个元素，这就会在渲染和动画过程中消耗更多的时间。更糟的是，如果我们引起了父元素的重绘，渲染引擎就需要重绘所有的子元素，严重影响运行时性能。

同步调用

我们会在等待同步调用返回的时候阻塞用户的动作吗？当在操作复选框或其他方式接受输入后更新服务端的状态，再等待确认对应的更新操作已完成时，就会经常发生这样的事情。这会让页面感觉起来有些卡顿。

CPU占用率

浏览器渲染页面和执行客户端代码需要多大负载？

我们要查看的Web运行时性能指标是每秒的帧数和CPU的占用率。

每秒帧数

每秒帧数（FPS）是动画师、游戏开发者以及电影摄影师常用的一种度量单位。它是系统重绘屏幕的速率。按照Paul Bakaus的博客帖子“The Illusion of Motion”(*http://bit.ly/1ou97Zn*)中的说法，人类感觉动作平滑、逼真的理想帧率是60 FPS。

也有一个Web应用，叫每秒帧数（*http://frames-per-second.appspot.com*），在浏览器中以不同的帧率演示动画效果。看这个演示，感受下自己的眼睛对同一个动画在不同帧率下的反应，很有意思。

FPS还是浏览器的一个重要性能指标，因为其反映出了动画运行以及窗口滚动的平滑程度。滚动时出现锯齿（卡顿）已经是Web性能问题的一个明显标志。

在Google Chrome中监控FPS

Google在创建浏览器内置工具追踪运行时性能方面在当前已是领头羊。其内置开发工具中已经包含了追踪FPS的能力。点击Rendering选项卡，然后选中“Show FPS meter”复选框，就可以看到（见图2-13）。

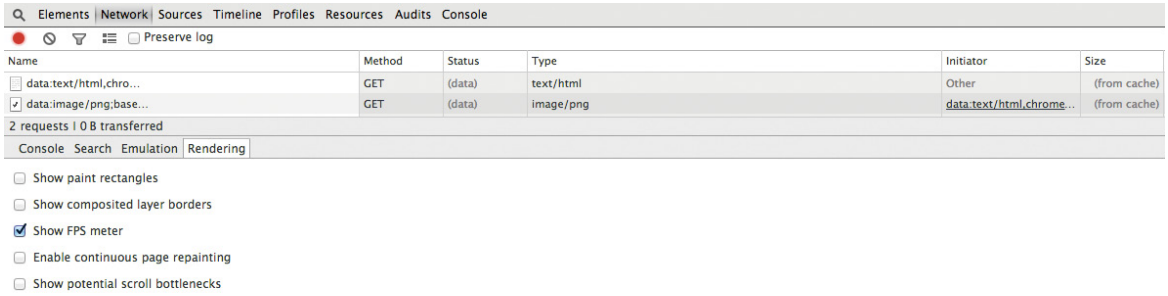

图2-13　在Chrome开发者工具中启用FPS meter

在浏览器的右上方会出现一个小的时间数列图，显示了当前FPS以及每秒帧数的趋势，如图2-14所示。使用这个工具，可以显式地追踪你的页面在实际使用过程中的执行情况。

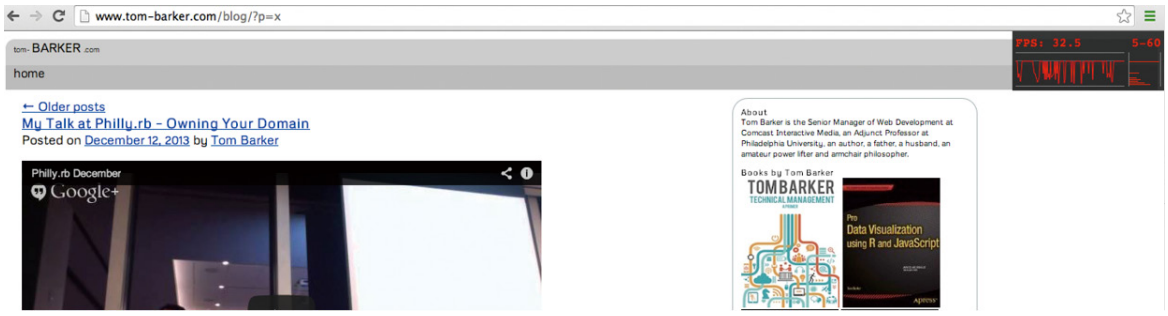

图2-14　Chrome的FPS meter，位于Web页面的右上角

虽然FPS meter是很好的追踪每秒帧数的工具，但迄今为止，对在帧率方面体验下降进行调试的最有用的工具还是Timeline工具，这个也是Chrome开发者工具中的一员（见图2-15）。

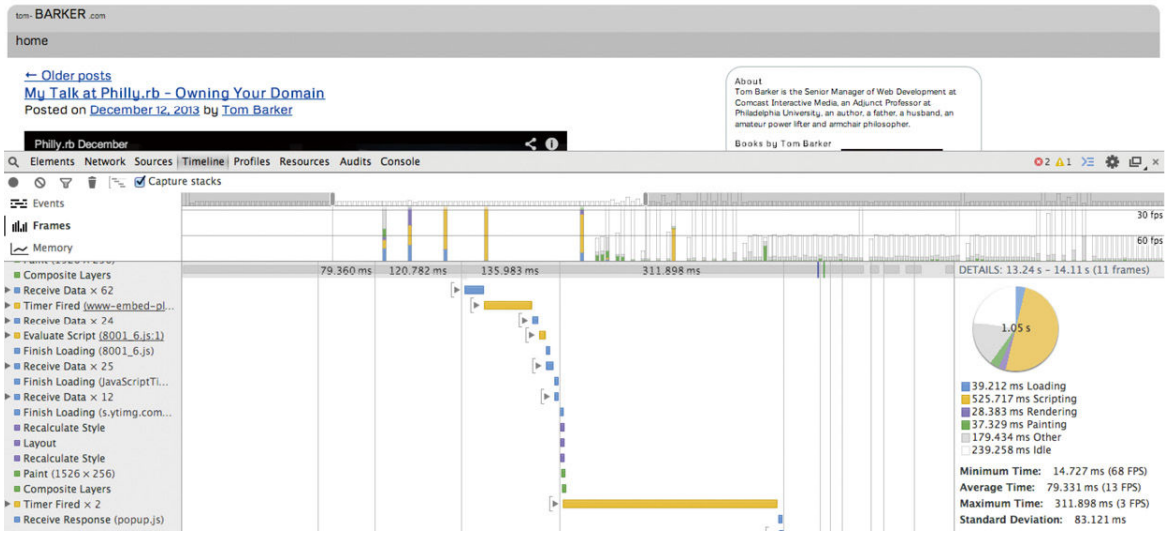

图2-15　Frames模式下的Chrome Timeline工具

用Timeline工具，可以追踪以及分析浏览器运行的时候都干了些什么。它提供了3个操作模式：Frames、Events以及Memory。我们来看一下Frames模式。

Frames模式

这个模式下，Timeline工具展示了Web应用的渲染性能。图2-15是Frames模式的屏幕布局。

在Timeline工具中可以看到两个不同的窗格。顶部窗格展示的是活动模式（位于左手边），里面包含了一系列代表帧的竖条。底部窗格是

Frames视图，这里展现的是形似瀑布的水平条状，标示了某个给定动作在帧里耗费的时长。在左边有对应动作的描述。在Frames视图的最右边是一个饼图，展示了在给定帧中最耗时动作的分类。所包含的动作如下所示。

- Layout
- Paint Setup
- Paint
- Recalculate Style
- Timer Fired
- Composite Layers

图2-15展现出运行JavaScript耗去了将近一半的时间，1.02秒中占了525毫秒。

使用Timeline工具，在Frames模式下，通过在Frames视图下找到最长的条，就能轻易地确定对帧率影响最大的动作。

内存分析

内存分析是监控我们应用所用到的内存消耗模式的一种方法。这对检测内存泄露与不会销毁的对象创建非常有用——JavaScript中，当我们用程序为DOM对象指定事件处理器，而后又忘记将事件处理器移除时尤为常见。更进一步，内存分析对优化内存占用也甚为有用。对象的创建、销毁与重用应当是智能的，要时刻注意，不要让剖析图中不断增加的一系列峰值呈上升趋势。图2-16描绘的是JavaScript堆。

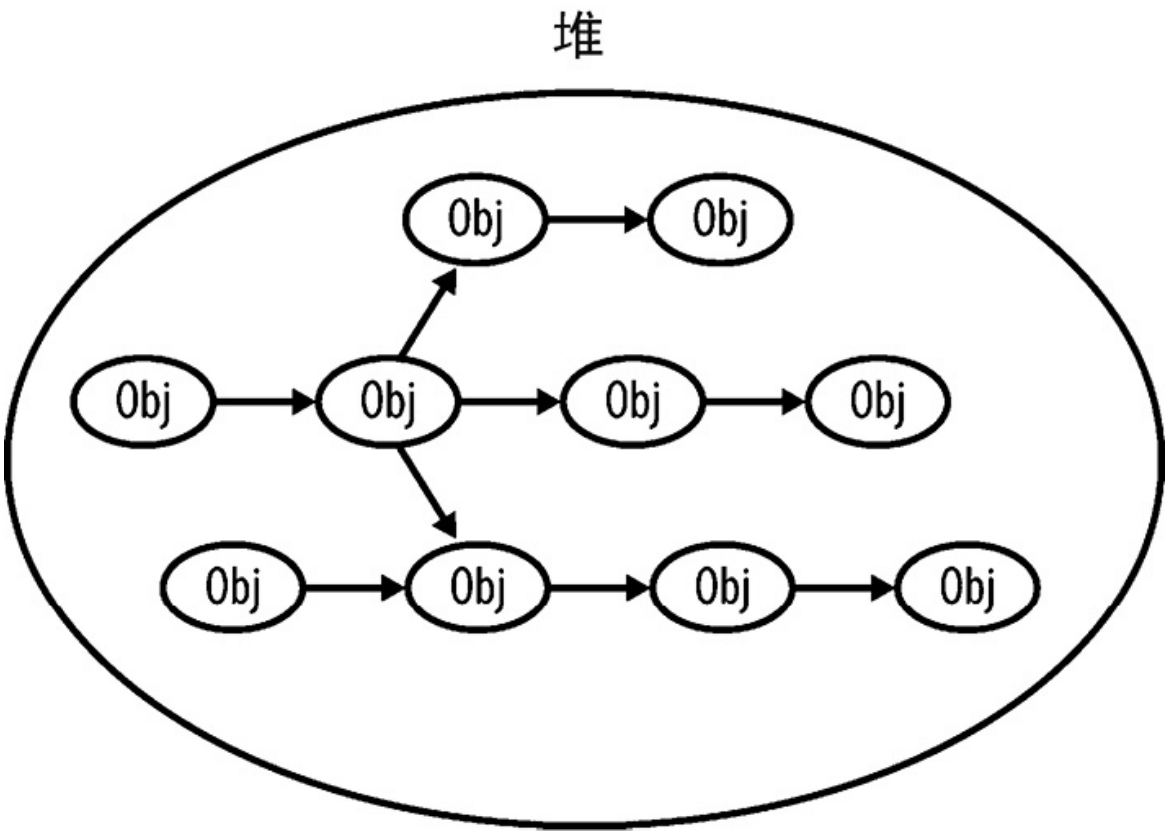

图2-16　JavaScript堆中的对象

虽然浏览器内置的功能远比之前强大，但这仍是一个需要扩大和规范化的领域。迄今为止，Google已经做了很多，让开发者可以用上浏览器内置的内存管理工具。

MemoryInfo对象

在Chrome已有的内存管理工具中，首先我们要看的是MemoryInfo对象，它存在于Performance对象中。图2-17的截图中展示了一个控制台视图。

```
> window.performance
▼ Performance {onwebkitresourcetimingbufferfull: null, memory: MemoryInfo, timing: PerformanceTiming, navigation: PerformanceNavigation, getEntries: function…}
  ▼ memory: MemoryInfo
      jsHeapSizeLimit: 793000000
      totalJSHeapSize: 26000000
      usedJSHeapSize: 17100000
    ▶ __proto__: MemoryInfo
  ▶ navigation: PerformanceNavigation
    onwebkitresourcetimingbufferfull: null
  ▶ timing: PerformanceTiming
  ▶ __proto__: Performance
```

图2-17　MemoryInfo对象

可以像这样访问MemoryInfo对象。

```
>>performance.memory
MemoryInfo {jsHeapSizeLimit: 793000000, usedJSHeapSize:
37300000, totalJSHeapSize: 56800000}
```

表2-2展示出了与MemoryInfo相关的堆属性。

表2-2 **MemoryInfo**对象属性

对象属性	描述
jsHeapSizeLimit	堆大小的上限
usedJSHeapSize	堆中当前所有对象使用的内存总量
totalJSHeapSize	堆的总大小，包括没有被对象使用的空闲空间

这些属性指出了可用和已用的JavaScript堆。堆是解释器保存在驻留内存中的JavaScript对象集合。在堆中，每个对象都是互有关联的节点，它们是通过诸如原型链或组合对象等属性连接起来的。浏览器中运行的JavaScript是通过对象引用来使用堆中对象的。当要销毁JavaScript中的一个对象，实际要做的就是销毁那个对象的引用。当解释器发现堆中对象不再有对象引用时，垃圾收集器将会从堆中移除这些对象。

用MemoryInfo对象，我们可以获取用户群与内存消耗相关的RUM数据，也可以在实验室里追踪这些指标，好在代码产品化之前发现潜在的内存问题。

Timeline工具

除了提供Frames模式来调试Web应用帧率之外，Chrome的Timeline工具还有Memory模式（如图2-18所示），能可视化地观察随时间变化的应用内存使用情况，并且会显示文档、DOM节点以及留在内存中的事件监听器的数量。

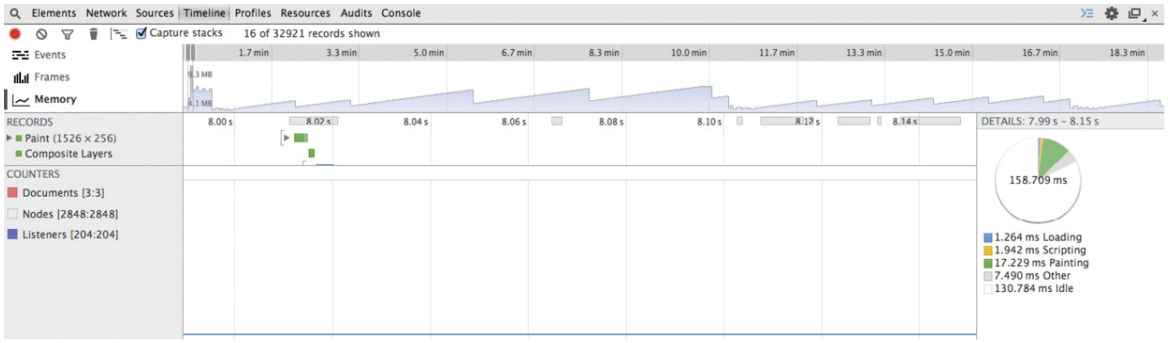

图2-18　Memory模式下的Chrome Timeline工具

顶部窗口展示的是内存剖析图，最底部的窗口展示了文档、节点以及监听器的数量。注意看，蓝色阴影区显示了内存使用率，可视化地展示了堆内存使用量。随着更多对象被创建，内存使用率也一直上升；当这些对象被销毁并被垃圾收集掉后，内存使用率就下降了。

可以在Mozilla开发网*http://mzl.la/1r1RzOG*找到一篇有关内存管理的很好的文章。

Firefox也开始开放内存使用数据，可以通过“about:memory”页面看到，Firefox的实现更多的还是通过静态信息页的方式而不是暴露一组API。正因为如此，它无法容易地插入程序中生成经验数据，about:memory页面更像是为Firefox用户（尽管是高级用户）设计的，而不是作为运行时性能管理的开发者工具集的一部分。

要在Firefox中访问“about:memory”页面，在浏览器的地址栏里键入about:memory。图2-19展示了这个页面的样子。

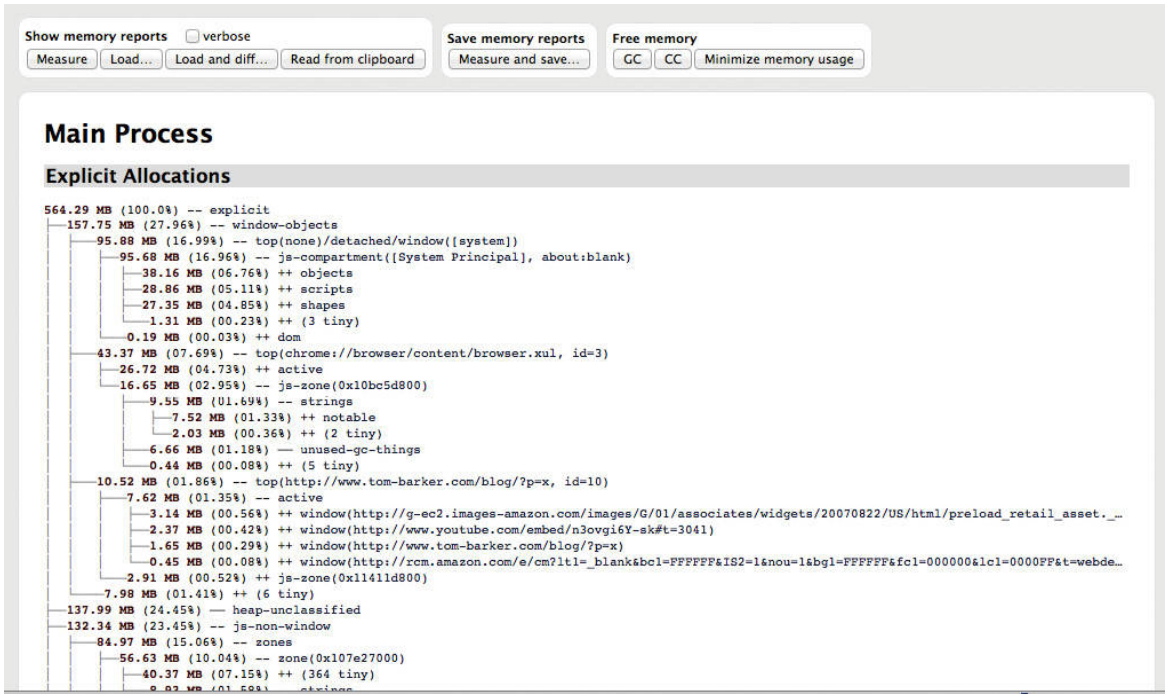

图2-19　Firefox的about:memory页面

如图2-19所示，可以看到以操作系统级别展示的浏览器分配的内存，以及浏览器打开的每个页面的堆分配情况。

2.4 小结

本章探讨了Web性能以及Web运行时性能。我们关注了页面内容是怎样从Web服务器到浏览器的，以及在传输与内存渲染过程中存在的潜在瓶颈。还关注了在运行时表示Web应用执行情况的性能指标，这是性能的另一个关键点：不仅要看内存传输到终端用户有多快，还要看传输过去后应用的可用性。

我们使用了一些工具来量化和追踪两种类型的性能。

最重要的是，我们对本书剩余部分将要着重讲到的概念有了共同的认知。当我们谈及诸如降低页面负载以及HTTP请求数或者避免页面重绘这些概念时，可以翻回本章再看上下文。

第3章我们来看看如何让响应式成为业务方法论以及软件开发周期的一部分。

第3章　千里之行始于计划

3.1　滑坡谬误的一段经历

犹记得我第一次启动一个期望做成响应式的项目。团队里的每个人都参与进来了：产品经理、设计团队、工程师。我们一起在探索有关什么是响应式这个问题，我们的集体观念一遍遍地刷新。对于所有的可能性以及能尝试一些新的东西我们都激动不已。

在那之前，我们维护了一个“m.”类型的站点，单独要一个开发人员对其更新，以期与主站保持一致。从工程管理上来讲，我们一直希望能把这个开发人员调回主团队，并且我们很享受与设计团队的合作。

我们已经投入几个星期了，但仍然没有什么东西可以向管理层演示，甚至连能看到的东西都没有，尽管如此，我们仍然为我们能拥有这极好的学习经验而感到高兴。当然，管理层并不满意，他们需要有一些具体成果，可以向他们的领导团队以及同事说说。设计团队的部分人从工作组中拆分出去做网站桌面版可能的原型，仅是作为一个能说得出口的东西。当然，原型做完展示之后，就通过了，突然就变成了我们要做的最终设计，并且要在结束日期前完成。

虽然我们知道应该先搞一个移动版的展现，然后以此为基础进行开发，我们很快就将所有想法推迟，为后续迭代构建出了一个小的视图，随后开始关注最终产品的视图创建。仅在一年后，我们就开始构想站点在其他设备上可能的体验，但这个时候主要的桌面版功能已经非常丰富了，所以响应式项目进展非常缓慢，并且开始变成一个个人“宠物”项目，花费了几个月来仿制了一个响应式站点模型。

这时候已经太迟了；这个模型的页面负载几乎无异于桌面版，在实际设备上展示时表现得也很差劲。所以站点依然仅支持桌面版。

是不是很像你自己上个项目或当前项目的经历？ 问题都出在哪里？为此我思考了一阵：我应该从中吸取什么样的教训，好在未来的项目上不再翻跟头？

从高的层面看，是我们自己搞死了自己。从团队内部来看，整个项目看起来像是有趣的探索与协作，摸清对我们来说全新的边界。从团队外部

来看，看着像是没有计划，没有目标——也的确如此。从长远看，我们计划的缺乏，削弱了管理层对团队的信任，开了干预团队的先例，并给我们定了粗略的最终目标。

本章，我将概述下怎样为团队制定一个计划，好快速产出可交付产品，以成为领导层的谈话要点，而同时仍然能坚持构建响应式、高性能站点的目标。

3.2 项目计划

响应式项目与任何其他项目没有任何差别，因为它们都会受益于项目计划。从程序或项目管理学来讲，有好几种类型的项目计划，视方法、组织、商业部门以及你所问的人（其他因素）而定，但一般来说，项目计划包含下面几个步骤。

1. 评估/总结整个任务。

2. 确定粗略的目标与时间表。

3. 列出资源和风险。

4. 列出衡量成功的关键性能指标（KPI）。

响应式项目唯一的不同在于，能证明各种设备体验的需求在上述的每个步骤中都应当是明确的。我们来更详细地看看每个步骤。

评估和总结整个任务

评估整个任务涉及收集需求以及确定项目的内容策略。这可能意味着需要与干系人或产品经理进行一次讨论，以确定站点的理念与愿景，以及想要实现的用例。这可能也意味着要与他们一起来做大量的用户测试与竞争分析来确定内容策略。

评估任务的有一部分是去回答一定的相关问题。比如，是否要为10英尺的屏幕提供视觉体验？或是否准备提供纯文本内容？正要做的是在为电视产品实现同等的体验？或是为锁定的一群用户实现一个局域网能用的站点？

你的项目真的需要响应式？若干年前，我做过一个Web应用项目，该应用旨在协助施工经理识别出显而易见的风险，譬如没有被围网围起来的污泥。出于这个用例的性质，这个项目永远也不需要一个桌面版，所以，我们做了一个手机版，桌面访问时就随它按比例缩放（当时还没有平板电脑）。

这个用例以及整个项目愿景应该可以清楚地回答这个问题了：这个项目我的目标视口是什么？这些视口应该是你需求的一部分，并且随着项目计划中每个步骤的进行，我们还将重新提到它们，但再一次说一下，第一

步是为了确认哪些是我们明确的目标。图3-1是潜在目标视口的几个代表，以及它们相对大小的差异。

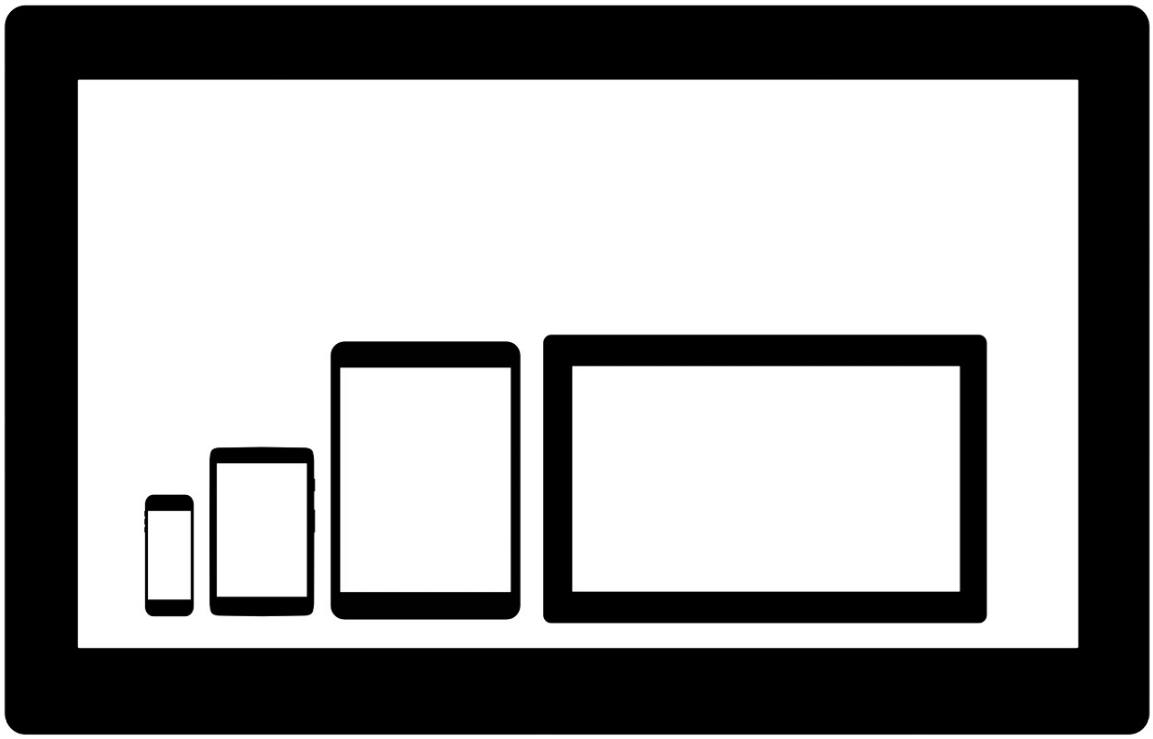

图3-1　几个视口的代表，从智能手机、平板、笔记本到高清电视，其中的差异不仅仅有尺寸，还有方向

除了尺寸方面的差异，还需要关注每个设备的观看距离、电池寿命以及网速和可靠性。

研究显示，从用户面部到智能手机屏幕的平均距离仅为12.6英寸；而笔记本为25英寸，电视为96英寸（见图3-2）。

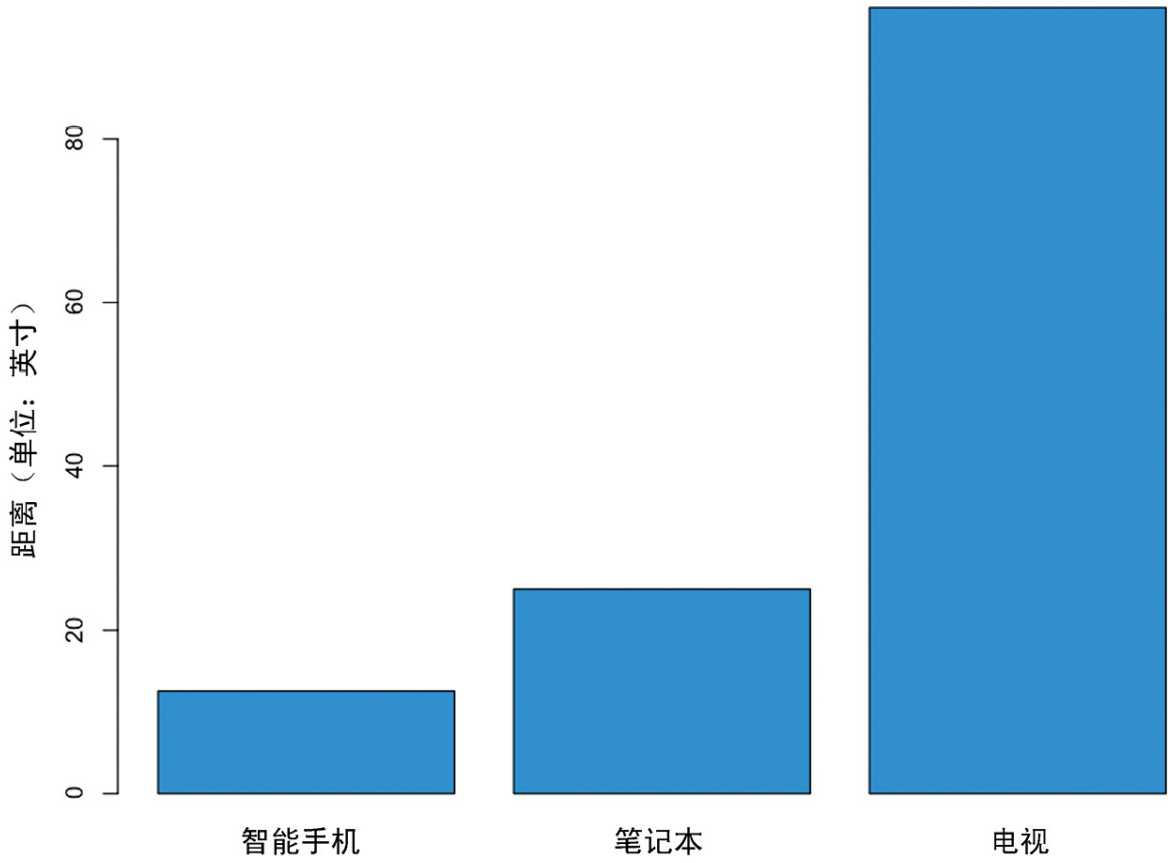

图3-2　不同设备的平均观看距离

观看距离的变化意味着很多地方的不同，包括图片和字体大小，每个都要请求不同的CSS规则，还可能为不同的设备请求不同的图片。在评估整个任务规模时需要考虑到这些。

不同设备间的平均网速差距也甚大。据Akamai在2013年发布的报告“State of the Internet”（*http://bit.ly/1tDGysM*）：在美国，平均宽带连接速度为11.6 Mbit/s，而平均手机连接速度为5.3Mbit/s，见图3-3。

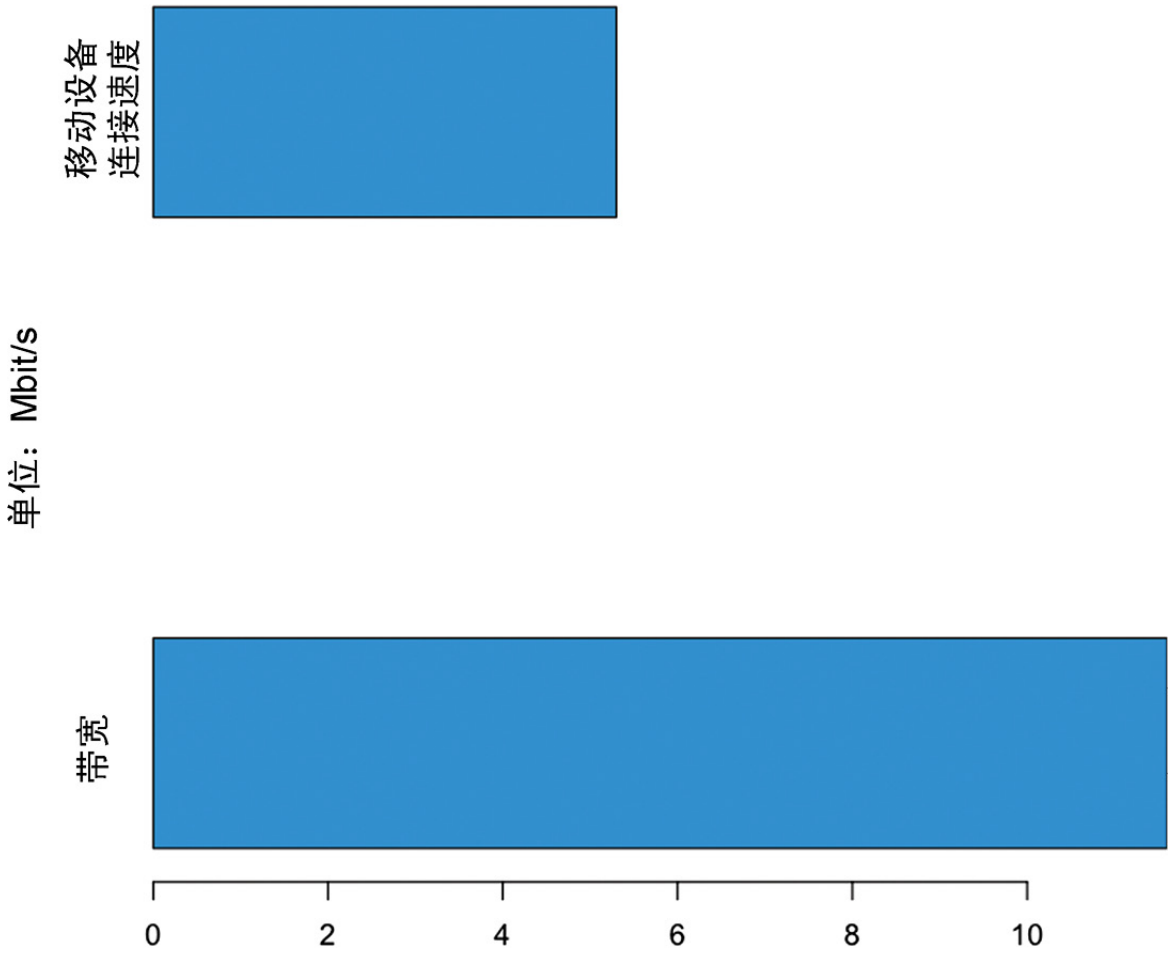

图3-3　2013年美国平均连接速度

连接速度上的差异凸显出了一个问题，要多久才能传递、渲染内容到设备上。这意味着，你需要相应地设计你的功能集以及运营预算。

建立粗略的目标与时间表

莫要空谈计划。在确定了目标视口之后，就应该做竞争分析了。用心去调查有相似功能的内部和外部应用，并基于竞争分析给出每个设备上的性能基准。为当前的性能现状做个水平划分，然后明智地决定你的应用该处在哪个水平上。

图3-4是手机版页面加载时间一个假设的竞争分析的结果。在这个假设的数据集中，我们可以看到，主要的内部和外部竞争者都落在500毫秒到1秒的范围里。这对于我们的应用是否是一个可接受的范围，或者我们是否需要做性能的领头羊，以500毫秒以内作为目标范围？在这个范围内的站点

都有一些什么样的功能，我们可以减小功能集来达到这么低的页面加载时间吗？

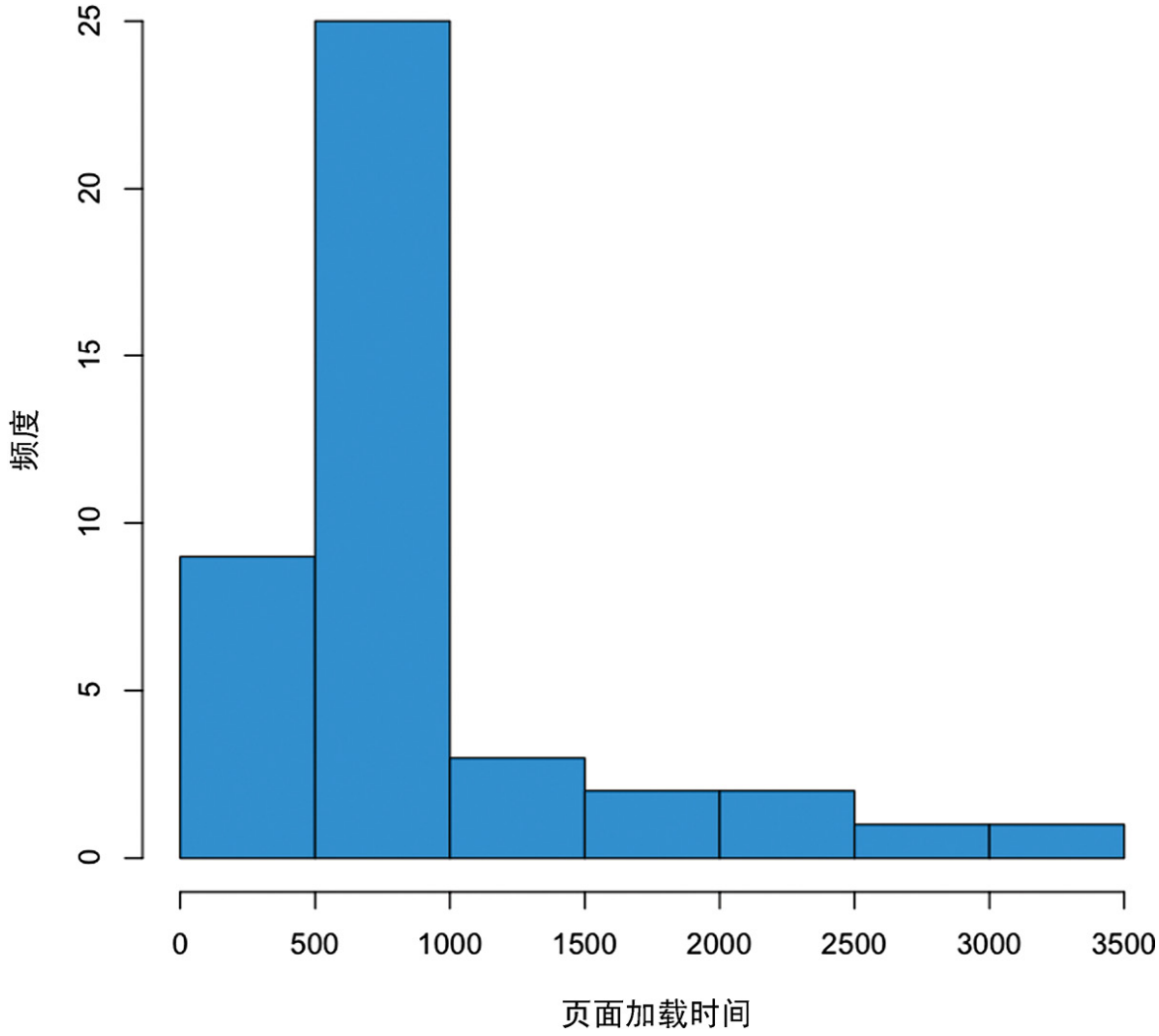

图3-4　假设的竞争者页面加载时间结果的直方图

在图3-4中，极端值已经高达3.5秒。需要做决定的是你想将自己的应用放在这个性能基线的什么位置。这就是你的性能服务水平协议。

确定一个性能服务水平协议

服务水平协议（SLA）是服务提供者的一个质量保证，通常保证的方面有响应时间、可服务时间以及错误率。作为一个网站的所有者，站点就是我们提供的一项服务，我们有必要向我们的终端用户以及内部干系人提供有关网站性能的一个SLA。

你的性能SLA应当明确所表述内容以及其度量方式。一个好的性能SLA内容可能是像下面这个样子。

在小屏幕上，我们网站的页面加载时间95%的时候保持在1秒或更低，在大屏上，至多3秒，以上数据通过综合测试得出。

一旦确定了性能SLA，这将影响不同版本上所提供的功能以及这些功能的展示形式。你也应该将这个SLA写进文档，让团队和干系人知晓。

确定粗略的里程碑与时间表

既然已经理解了所做产品的需求以及性能层面真正包含的内容，就可以开始具体实现了。这就像用户需求层次树结构一样丰富而复杂，也可以像一张T恤尺码表一样高级。

但是，所有的设备/分辨率/特定视口都应该明确写下来，并作为整个时间表的指标，如图3-5所示。

User Story	Risks	Dependencies	T-Shirt Size
Create environments			1 sprint
Identify Endpoints			1 sprint
Create 1024 × 768 view of homepage (for older laptops and Ipad 1 and 2s)			1 sprint
Create 2048 × 1536 view of homepage (for Ipad 3 and up)			1 sprint
Create 768 × 1024 view of homepage (for iPad minis)			1 sprint
Create 2560 × 1440 view of homepage (for laptops)			1 sprint
Create 640 × 1136 view of homepage (for iPhone 5s)			1 sprint

图3-5　一个概略计划样本，每种分辨率与目标设备都设有里程碑

要清楚，图3-5中所列出的高层面的需求（创建1024×768的视图，创建2560×1440的视图）并不能就认为它们是不同的页面——这只是一个将要完成的指标集（如果你乐意，也可以称为目标）；对于该如何实现，则没有体现。

注意

> Radu Chelariu在Smashing Magazine发表一篇非常好的文章，简要地列出了一堆设备的分辨率，参见*http://bit.ly/zqcgUb*.

还有一点：高层面的用户需求定义了基础架构的方案以及监控我们SLA的过程，既然我们已经承诺遵守性能SLA，就应当将这些高层面的用户需求包含进来。让我们把这些需要保障的点加到已有列表中吧，见图3-6。

User Story	Risks	Dependencies	T-Shirt Size
Create environments			1 sprint
Identify Endpoints			1 sprint
Create 1024 × 768 view of homepage (for older laptops and Ipad 1 and 2s)			1 sprint
Create 2048 × 1536 view of homepage (for Ipad 3 and up)			1 sprint
Create 768 × 1024 view of homepage (for iPad minis)			1 sprint
Create 2560 × 1440 view of homepage (for laptops)			1 sprint
Create 640 × 1136 view of homepage (for iPhone 5s)			1 sprint
Create environments for internal WebPageTest instance			1 sprint
Set up and configure internal WebPageTest instance			1 sprint
Integrate SLA check into CI workflow			1 sprint

图3-6　高层面的需求列表，加入了跟踪SLA相关的内容

列出依赖和风险

当我们有了高层面的需求描述，对每个需求都做了时间估算之后，就可以开始列出每个需求的风险与依赖。这些应当都是非常简单且是常识性的东西，但是，你仍然需要将它们列出来以说明完成这个需求需要的措施，并向干系人表明这些措施是已经被采用了的。图3-7是上个例子的延续，这次列出了依赖与风险。

User Story	Risks	Dependencies	T-Shirt Size
Create environments		Infrastructure	1 sprint
Identify Endpoints		API Team, Connectivity to API endpoints	1 sprint
Create 1024 × 768 view of homepage (for older laptops and Ipad 1 and 2s)	Must adhere to SLA for tablets	Need designs or wireframes for this resolution, need SLA for this device	1 sprint
Create 2048 × 1536 view of homepage (for Ipad 3 and up)	Must adhere to SLA for tablets	Need designs or wireframes for this resolution, need SLA for this device	1 sprint
Create 768 × 1024 view of homepage (for iPad minis)	Must adhere to SLA for tablets	Need designs or wireframes for this resolution, need SLA for this device	1 sprint
Create 2560 × 1440 view of homepage (for laptops)	Must adhere to SLA laptop/desktop	Need designs or wireframes for this resolution, need SLA for this device	1 sprint
Create 640 × 1136 view of homepage (for iPhone 5s)	Must adhere to SLA for smartphones	Need designs or wireframes for this resolution, need SLA for this device	1 sprint
Create environments for internal WebPageTest instance		Infrastructure	1 sprint
Set up and configure internal WebPageTest instance		need environments set up	1 sprint
Integrate SLA check into CI workflow			1 sprint

图3-7　整体项目需求计划中的依赖与风险

图3-7解释了我们是如何看到这些依赖里包含了设计或线框图，环境的准备，以及定义好的性能SLA。通过明确地列出这些内容，我们就能看出哪些需求需要建立在其他需求之上。这也让我们避开这些需求来创建一个有意义的时间表成为可能。

制定时间表

既然我们已经知道了完成该项任务将要涉及的步骤，就可以建一个粗略的时间表了。通过为每个任务使用高层次的T恤尺寸表示法，我们就能对它们进行有意义地分组，并把它们横放到时间表中。

对于这个例子，假定我们是两周一个迭代。假设我们熟知我们团队的开发速度，就能够粗略构想出每个迭代应当做的东西。我们可以将所有的研究与准备工作放到单独的一个迭代中。然后将若干个需求放到另一个迭代，剩下的放到第三个迭代。

通过下面的方法，我们可以看到这个任务可能至少是一个六周的项目，如图3-8所示。

Week 1	Week 2	Week 3	Week 4
Sprint A		Sprint B	
Create environments		Create 640 × 1136 view of homepage (for iPhone 5s)	
Identify Endpoints		Create 768 × 1024 view of homepage (for iPad minis)	
Create environments for internal WebPageTest instance		Create 1024 × 768 view of homepage (for older laptops and Ipad 1 and 2s)	
Set up and configure internal WebPageTest instance		Integrate SLA check into CI workflow	

Week 5	Week 6	Week 7	Week 8
Sprint C			
Create 2048 × 1536 view of homepage (for Ipad 3 and up)			
Create 2560 × 1440 view of homepage (for laptops)			

图3-8　高层面需求的粗略时间安排

这里的重点是，这些都是非常粗的时间表。其实，说得文雅点，就是瞎猜或粗略估计的。只要你让干系人清楚地知道，当有更多信息的时候，这个时间表会经常变化，且在有新进展时会及时传达，这样你就会做得很好。

衡量成功的关键性能指标（KPI）

现在我们已经对这个任务做了评估，创建了粗略的时间表，且列出了达成这个时间表的所涉及的依赖。接下来，我们需要确保已经清晰地定义了成功的标准。实际上，衡量这个项目成功与否的关键性能指标在我们产品或业务团队找我们完成这个任务之前就已经存在了，但是，我们需要与他们一起确认清楚，首先，这些KPI对整个团队都是明确的，其次，我们针对这个任务的解决方案与预期标准是一致的。

如果这时KPI还没定下来，我们就需要与干系人一起把它确定掉。不然我们怎么知道项目是否成功了，又怎么在迭代中进行优化呢？

遵守性能SLA

现在我们已经有了一个要做的事情的计划，确定了阶段目标，达成每个目标时会保持沟通。每个版本我们都有一个性能SLA，现在我们可以开始做事了。

但在开发过程中，务必遵守我们的性能SLA。需要确保性能测试是持续集成工作流程的一部分，当背离SLA时，需要警觉。第6章中我们会着重讲怎么去做。

当评估新的功能时，用你的SLA作为一个讨论点。这些新功能是否会影响性能？业务规则中的细微改动是否会为产品带来更高的性能？

3.3 小结

本章并没有打算讲如何管理一个项目，而是在于探讨一种将响应式和性能整合到一个项目计划中的方式。有了快速响应的项目计划，我们可以向干系人传达有意义的阶段目标，而且不需要牺牲任何一种设备体验，因为这原本就是我们最终产品的一个部分。

第4章　响应式服务端实现

本章讨论的核心，也是全书的核心，就是不赞同把响应式Web设计作为一个前端独有的技术。因为如果我们这样认为了，它就会限制我们的思考广度，我们能做什么？我们可以用什么工具？但是作为Web开发者，我们必须精益求精，必须有能力把控Web全栈中任何一个细节。本章将讨论我们如何基于后端来思考如何更好地实现响应式。

4.1 Web栈

在我们开始本章内容之前，我必须先给Web栈一个定义：Web栈是一系列栈的集合。而在讨论Web栈之前，让我们先从网络栈开始。

网络栈

网络栈是由不同网络系统相互通信的一系列协议的统称，它由下面一些层构成。

数据链路层

数据链路层对应的是硬件连接到网络的标准方式。对我们而言，通常是以以太网的形式，具体是支持IEEE 802.3标准的物理互联设备(*http://bit.ly/ethernet-standards*)；或者也可以通过WiFi，具体是支持802.11标准的无线互联设备(*http://bit.ly/1p8UW6P*)。

网络层

网络层对应的是网络中不同网络节点之间相互识别和通信的标准，具体是IP协议或者互联网协议。它是通过IP地址在网络中识别网络节点，并且在主机之间通过数据包的形式传递数据。IETF RFC 749主要记录了标准的互联网协议，你可以在*http://bit.ly/11j3ouQ*上阅读到详细信息。

传输层

传输层通常相当于TCP，全称是Transmission Control Protocol，也就是传输控制协议。在IETF RFC 793 (*http://www.ietf.org/rfc/rfc793.txt*)中定义。

TCP是用来在主机间建立链接的协议。IP协议负责将数据以数据包的方式进行传输，而TCP将数据包分成多个段，并且为每个段创建含有目标IP地址的header，重新组装并且对接受到的这些网络传输数据进行校验。

应用层

应用层是这一系列协议的最上层，它对应的是像HTTP这样的协议，HTTP也称为超文本传输协议。HTTP的标准是IETF RFC 2616，你也可以在*http://tools.ietf.org/html/rfc2616*上看到详细信息。HTTP是一种Web语言，它主要是由request（请求）/response（响应）两个动作组成。

把这些栈组织起来就得到了数据在Internet中发送和接收的整个流程步骤了，如图4-1所示。

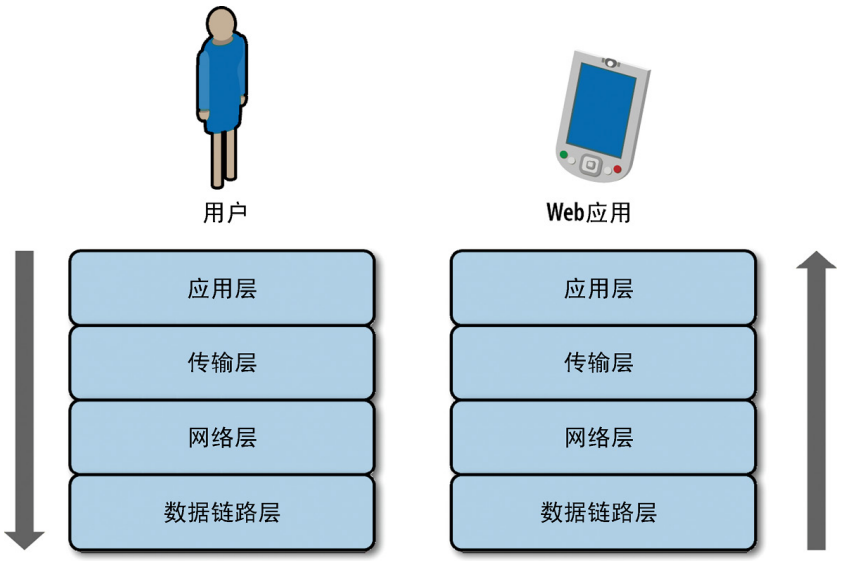

图4-1　用户通过TCP/IP栈发送一个请求，然后这个请求同样通过TCP/IP栈和远端服务器上的Web应用进行通信

应用层

了解所有的底层知识是非常重要的，但是对Web开发者来说，最主要的也就是我们每天都会面对的，并且可以通过编码方式控制的一层，就是应用层，也就是HTTP协议。第2章向我们展示了一个HTTP传输在TCP的连接中都要经历哪些步骤。一般来说由两个部分组成：从客户端发送一个请求，然后从服务端返回一个响应。很容易理解，对吧？不过深入了解HTTP内容对我们的工作大有裨益，现在我们来看看一个请求和一个响应的细节性问题，更深入地探讨它们的内部机制。

HTTP请求

一个HTTP请求由两个部分组成：请求正文和一系列请求header。其中，请求正文包括请求的HTTP方法或者动作，以及请求远程资源的URI。更简单来说，它表明了当前要执行的动作（获取文件，发送文件或者获取一个文件的信息）以及这个动作在哪执行（文件或者资源的地址路径），下面就是一些在HTTP1.1中支持的方法。

OPTIONS

服务器支持的HTTP请求方法的功能选项。

GET

当你请求远程资源时，如果你在HTTP header中指定了If-Modified-Since、If-Unmodified-Since、If-Match、If-None-Match或者If-Range里的一项，那么它就变成一个有条件的GET了。因为这个时候，服务器只返回符合这些请求条件的资源。一般来说，如果你想控制获取资源的方式，是获取最新的资源还是使用当前缓存的时候，就可以使用这种有条件的
GET。

HEAD

只请求远程资源的HTTP header，它主要是用来检查最后更改的日期或者确认一个URI是否可用。

POST

请求服务器对一个资源进行更新或者修改。

PUT

请求服务器创建一个新的资源。

DELETE

请求服务器删除一个资源。

请求header允许客户端为请求指定或者增加参数，这种行为和向一个函数传递参数非常类似。下面是一些非常有趣的请求header。

Host

在URI中指定的域名名称。

If-Modified-Since

这个参数在Request header中传给服务器，指示服务器只返回在指定的时间里有过更新的资源。如果资源已经更新过了，那么服务器就返回这个资源和200状态码。如果没有，那么服务器将返回304状态码。

User-Agent

这个参数标识客户端的具体特征信息。在本章中的内容里，我们在很多地方都使用了这个参数。

你可以通过一些网络探查工具——例如Charles或者Fiddler——检查HTTP请求的内容。下面的例子我们展示了一个HTTP请求的详细信息。

```
GET /style/base.css HTTP/1.1
Host: www.tom-barker.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7;
rv:27.0) Gecko/20100101 Firefox/27.0
Accept: text/css,*/*;q=0.1
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.tom-barker.com/blog/?p=x
Connection: keep-alive
```

HTTP响应

当服务器接收并处理了一个请求以后，它将会返回客户端一个响应信息。和HTTP请求一样，HTTP响应也由两个部分组成：状态行和响应header的属性。

状态行中有协议版本号（HTTP 1.1），状态码和描述请求状态的文本短语，状态码由三位数字组成，并且把响应分成了五个层次类别。状态码的第一位数字表面它的类别。你可以在*http://bit.ly/rfc-http*上查到W3C的HTTP详细规范，具体的类别如下所示。

1xx：信息

请求已收到，正在处理中。

2xx：成功

请求已经成功接收、解析并执行了。

3xx：重定向

需要进一步的跳转和更多的操作来完成当前请求。

4xx：客户端错误

请求包含了语法错误，不能被执行。

5xx：服务端错误

服务端在处理一个有效的请求时失败。

响应header里的属性值和请求header里面的属性值很像，服务端通过指定这一系列的键值对表明当前响应的具体信息。下面就是我们常用到的一些响应header。

Age

标识请求的资源从创建或者更新时到请求发生时的大概时间。

ETag

列出了服务器为这个资源分配的唯一实体标记符，这个参数在一些需要额外匹配验证的时候非常有用。

Vary

这个参数标明当前请求的哪个请求头用来表示该请求是否可被缓存。在本章的后半节中，我们将模拟基于不同的User Agent，服务器发送不同的响应。Vary这个参数非常重要，因为它让我们可以将User Agent头信息作为一个请求是否可以被缓存的决定因素之一。

下面是HTTP Response的示例。

```
HTTP/1.1 200 OK
Date: Sat, 29 Mar 2014 19:53:24 GMT
Server: Apache
Last-Modified: Sat, 05 May 2012 22:11:12 GMT
Content-Length: 2599
Keep-Alive: timeout=10, max=100
Connection: Keep-Alive
Content-Type: text/css
```

Charles

现在有非常多的工具可以观察到你的网络传输情况，这些开发者工具是以浏览器插件方式存在的（详见第2章）。还有更多可以深入分析网络传输的工具：它们中的一个佼佼者就是Charles，它在Web开发者中非常受欢迎（见图4-2）。

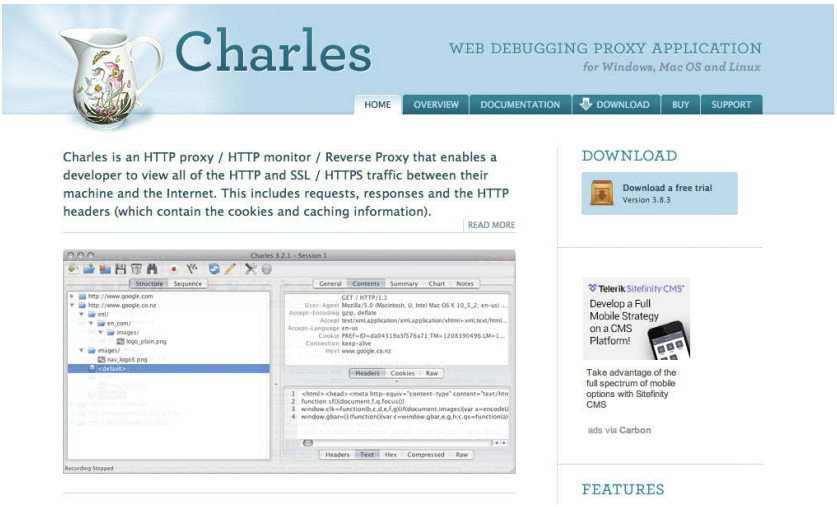

图4-2　Charles主页

Charles是一个HTTP监测工具，利用Charles，你可以观察和编辑网络传输内容；Charles同样也是个HTTP代理工具，你可以使用它控制带宽流量，降低连接延迟，拦截请求,DNS欺骗，甚至映射到本地文件，使得访问本地文件可以像访问远程服务端文件一样。从*http://www.charlesproxy.com/*上可以下载最新的Charles。

图4-3展示了Charles的界面。这个截图按顺序展示了在一给定访问中的所有HTTP事务，我们可以看到，在展示的结果中，包含了HTTP状态、HTTP方法、主机名、传输负载和持续时间等。

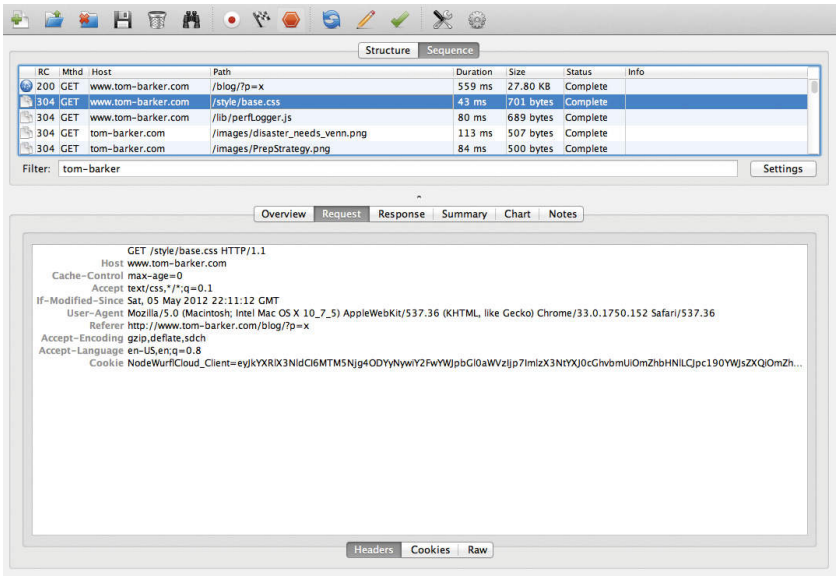

图4-3　Charles记录的HTTP传输信息

4.2 Web应用栈

到目前为止，我们一直在讨论Web应用运行所需要的基础设施和网络协议。现在我们要理解Web应用运行所需要的软件栈。绝大部分情况下，Web应用是基于客户端-服务器模式的，也可以将其看成是一个分布式计算方式。如果用一句话来总结，就是客户端向服务器端请求数据，服务器处理请求和响应。通常情况下，为了可扩展性，这些服务器分布在整个网络中。

为了符合这种模式，在下面的具体示例中，首先让我们假设浏览器就是客户端，Web服务器就是server。当我在说Web服务器的时候，指的是像Apache（*https://httpd.apache.org/*）和微软的IIS（*http://www.iis.net/*）这样的应用软件，或者是运行这些软件的硬件。

言归正传，这些Web服务器监听某些特定的端口——表示应用端口的数字，也就是HTTP请求的端口。通常情况下，HTTP请求的是80端口，HTTPS请求的是443端口。当Web服务器接收到这个请求的时候，它就会把这个请求发送给对应的资源。

类似于Ruby或者PHP，或者像HTML页面这样的静态内容，这些资源可以在服务端以代码的方式进行评估和解析。无论哪种方式，这个被发送的请求都会获得服务端的响应。

如果服务端响应的正文是HTML文件， HTML将在客户端设备上进行解析和渲染。如果内容中包含了任何JavaScript文件，它们也会被客户端设备解释运行。

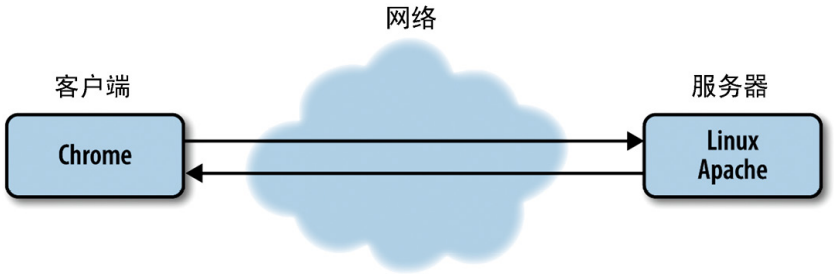

图4-4 客户端-服务器传输示例

4.3 服务端响应

现在你已经对协议和软件栈在整个Web栈中的定位和作用有了一个大概的了解，所以在整栈中，你应该做的第一件事，就是尽可能早地判断出客户端的相关特性。现在，响应式设计是客户端获取服务器端发送HTTP响应后，在客户端根据获取到的相应的客户端特性，然后根据这些特性对响应的内容进行接收、解析和渲染等一系列操作。

浏览器请求页面的架构如图4-5所示。Web服务器在80端口接收到了请求，然后传递给Web应用，Web应用随后处理这个请求并发送响应。客户端接收到这个响应以后，解析页面内容，渲染页面，在客户端设备上运行获取客户端设备特性的代码，最后根据这些特性做相应的页面显示。

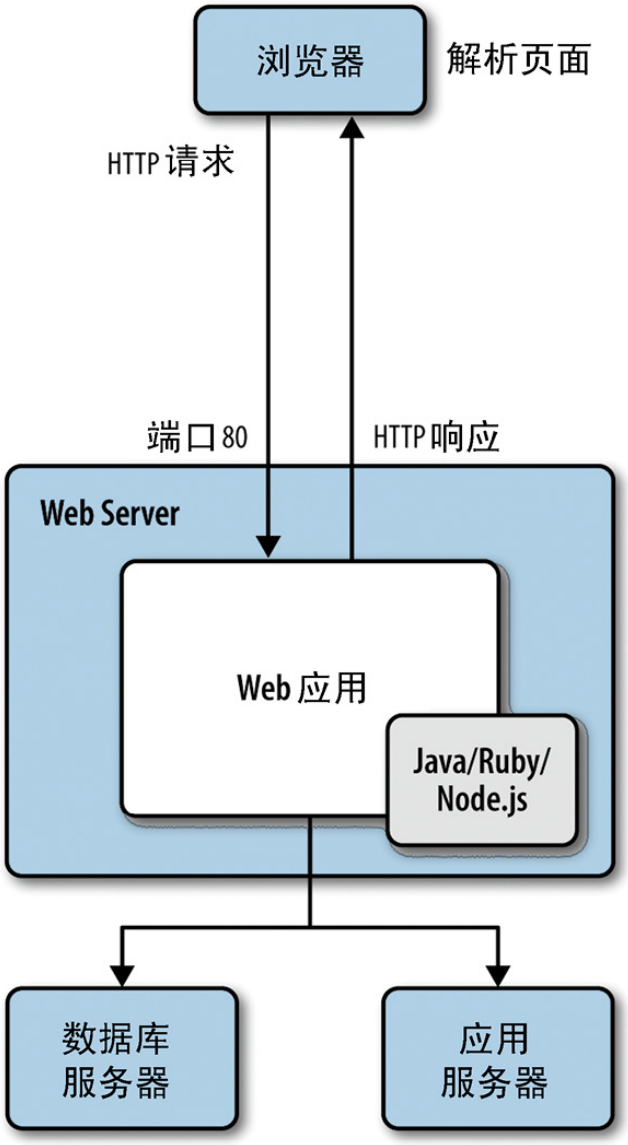

图4-5 在客户端确定特性

即使用文字写把所有这些步骤列出明细，也会让人感觉过度和不需要如此复杂。

我们可以从HTTP请求的描述信息中收集客户端的一些特征信息，还记得之前说过的User Agent吗？是的，这些描述客户端的特征信息正是通过User Agent传递给Web服务器和Web应用的。我们可以在服务端，而不是客户端确定客户端的特征和能力，这将有效地简化我们发送给客户端的内容——发送针对设备专有的代码而不是全部的代码（图4-6为修改过的架构）。

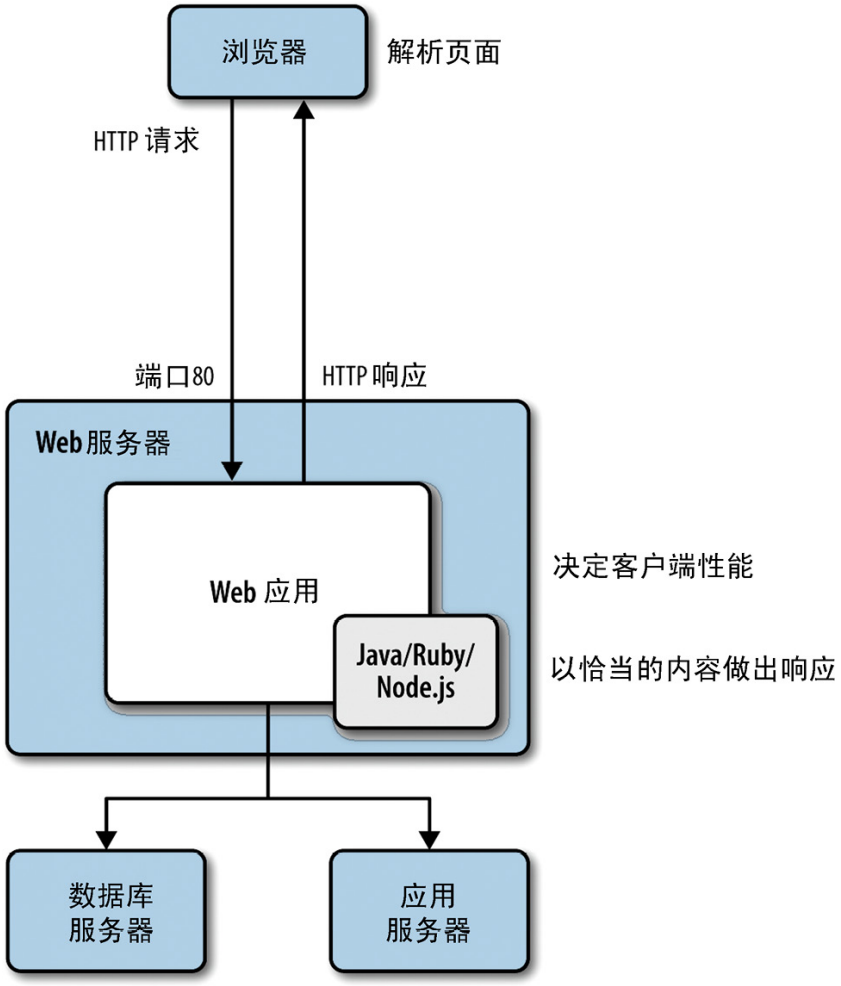

图4-6　在服务端确定客户端的能力并且响应对应的客户端内容

看到这里，你肯定非常想知道我们是如何利用User Agent确定客户端能力的。所以首先我们来看一看User Agent。

检查User Agent

你可以在RFC 2616的14.43小节查看User Agent的字段规范，具体的HTTP规范可以在*http://bit.ly/1tDGOZO*上找到详细信息。

User Agent是由不同的令牌组成的一个字符串，它描述了浏览器、浏览器版本、操作系统及系统版本等一系列系统信息，见表4-1。

表4-1　User Agent的一些示例字符串

浏览器	User Agent字符串

浏览器	User Agent字符串
Chrome34Mac版	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/34.0.1847.116 Safari/537.36
Safari OS7+iPhone版	Mozilla/5.0 (iPhone; CPU iPhone OS 7_0 like Mac OS X) AppleWebKit/537.51.1 (KH TML, like Gecko) Version/7.0 Mobile/11A465 Safari/9537.53
Safari OS6+iPad版	Mozilla/5.0 (iPad; CPU OS 6_0 like Mac OS X) AppleWebKit/536.26 (KHTML , like Gecko)Version/6.0 Mobile/10A5355d Safari/8536.25
Chrome Android手机版，并且运行在Ice Cream Sandwich版本上	Mozilla/5.0 (Linux; U; Android 4.0.3; ko-kr;LG -L160L Build/IML 74K) AppleWebkit/534.30(KHTML , like Gecko) Version/4.0 Mobile Safari/534.30

你可以很容易获取这些字段并通过正则表达式解析得到相关的信息。下面这个例子，通过一个函数来确定客户端设备，从而建立起响应的客户端功能。这个例子非常简单，我们将使用JavaScript创建一个函数，像下面代码那样检查移动设备。

```
function detectMobileDevice(ua){
    var re = new RegExp(/iPhone|iPod|iPad|Android/);
    if(re.exec(ua)){
        return true;
    }else{
        return false;
    }
}
```

这里需要注意的是，我们将User Agent作为参数传递给detectMobileDevice函数，并且使用正则在User Agent中搜索匹配iPhone、iPad或者Android的字符串；一旦匹配，我们将返回true。

这是一个相当简单而基础的示例，它只关注了当前的客户端设备的平台或者操作系统。我们还可以使用更强大的功能检查客户端的特性，例如触摸支持和设备的最大宽度。

Google和Apple分别在*http://bit.ly/1u0cHqv*和*http://bit.ly/ZXVAhT*上发布了它们的User Agent标准。

在探讨User Agent可靠性的时候，有个词需要特别注意：当你在阅读规范的时候，你会注意到客户端SHOULD（应该）在发送的请求中包含User Agent信息。在规范中，这是一个非常明确的声明。事实上，SHOULD在IETF的说明中是作为关键字列出的。针对关键字的含义，也有一个规范说明，你可以在*http://tools.ietf.org/html/rfc2119*上看到这段声明。对于SHOULD这个关键字，有如下规定：

……现实中确实存在一些特殊的情况，在这些特殊的情况下需要忽略一个特殊的选项，但是必须可以理解全部的含义，并且在选择一种不同的方式之前，一定要权衡利弊。

毫不掩饰地说，这句话也就是说，客户端没有义务使用User Agent字段或者使用正确的User Agent正确标识它们自己。如果用户愿意，它甚至可以伪造User Agent进行欺骗。机器人或者爬虫经常导致一些意想不到的结果。但是这些都是非正常的情况，当我们针对普通用户进行开发的时候，相信这些User Agent的信息并不会有什么不妥之处。使用User Agent最大的痛点就是你要和所有新发行的设备保持同步，并且可以将User Agent和已知的特性和效果集联系起来。这也是为什么我们需要一个设备检测服务的原因。

设备检测服务

如果我们的客户端设备只是一些已知的设备，那么前面一个例子完全可以说明问题了。但是如果我们想检查客户端的特征和尺寸大小，该怎么做呢？我们可以根据User Agent自定义一张客户端能力对应设计

表，这样我们就可以在这张定义的设计表里搜寻；或者使用一个更高级的服务，这个服务为我们提供了表和查询的能力。

设备检测服务，通过传递请求，可以为我们提供获取客户端能力的功能。这种解决方案的架构如图4-7所示，当客户端通过网络发送请求时，我们的服务接收到这个请求，在服务器层我们开启一个后门调用设备检测服务。

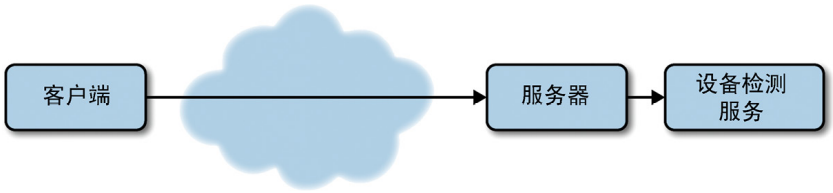

图4-7　在服务器端使用设备检测服务

使用最早也是最广泛的一个设备检测服务就是Wurfl。

Wurfl

Wurfl全称是无线通用资源文件（Wireless Universal Resource FiLe）。在2011年之前，它是一个列出了设备对应相应能力的免费开放的XML文件，其结构如下所示。

```
<device id="generic_android_ver3_0" user_agent="DO_NOT_MATCH_ANDROID_3_0" fall_back="generic_android_ve
   <group id="product_info">
      <capability name="is_tablet" value="true"/>
      <capability name="device_os_version" value="3.0"/>
      <capability name="can_assign_phone_number" value="false"/>
      <capability name="release_date" value="2011_february"/>
   </group>
   <group id="streaming">
      <capability name="streaming_preferred_protocol" value="http"/>
   </group>
   <group id="display">
      <capability name="columns" value="100"/>
      <capability name="physical_screen_height" value="217"/>
      <capability name="dual_orientation" value="true"/>
      <capability name="physical_screen_width" value="136"/>
      <capability name="rows" value="100"/>
      <capability name="max_image_width" value="980"/>
      <capability name="resolution_width" value="1280"/>
      <capability name="resolution_height" value="768"/>
      <capability name="max_image_height" value="472"/>
   </group>
   <group id="sms">
      <capability name="sms_enabled" value="false"/>
   </group>
   <group id="xhtml_ui">
      <capability name="xhtml_send_mms_string" value="none"/>
      <capability name="xhtml_send_sms_string" value="none"/>
   </group>
   </device>
```

从2011年开始，Wurfl的开发者们创建了自己的公司Scientiamobile，开始围绕Wurfl提供服务，并且停止了支持个人消费方面的开放文档。他们围绕着Wurfl提供了一系列的产品，包括Wurfl Cloud——提供API访问设备数据库；Wurfl Onsite——本地安装的设备数据库；Wurfl Infuze——在服务端通过环境变量保证Wurfl数据库的可用性。

理论上，最好的解决方案就是Wurfl Infuze，因为当查询设备数据的时候没有产生文件I/O和传输延迟的额外消耗。然而门槛最低的解决方案是WurflCloud，因为它不需要内部主机，不需要安装基础设施，甚至有免费的选项。基于这些原因，我们在本章剩下的内容中讨论如何在我们的应用中集成Wurfl Cloud。

在开始前，我们应该先到Scientiamobile的主页*http://www.scientiamobile.com/*上查看相关的信息，图4-8是Scientiamobile网站主页的截图。

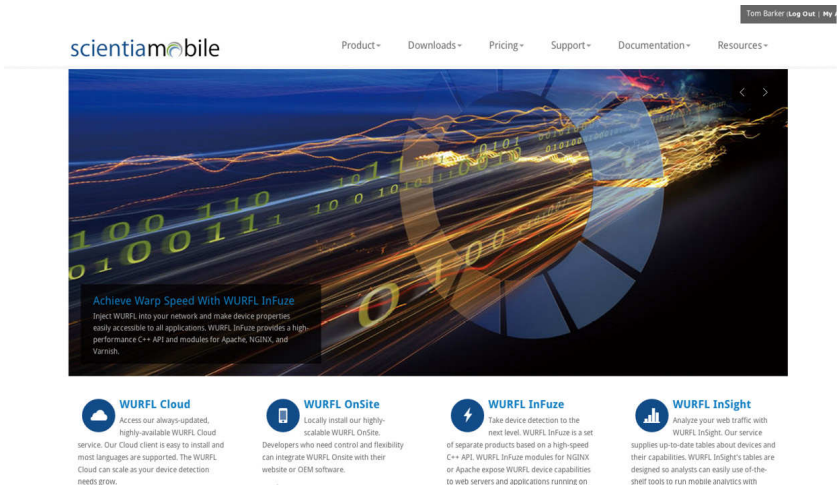

图4-8　Scientiamobile主页

我们可以点击页面底部的Wurfl Cloud，然后会被导航到价格页面。不用担心，在Sign Up链接下面有个免费选项，点击以后，我们会被导航到图4-9所示的页面。在这个页面我们可以创建我们自己的账号。详细信息请移步*http://bit.ly/1x34Psg*。

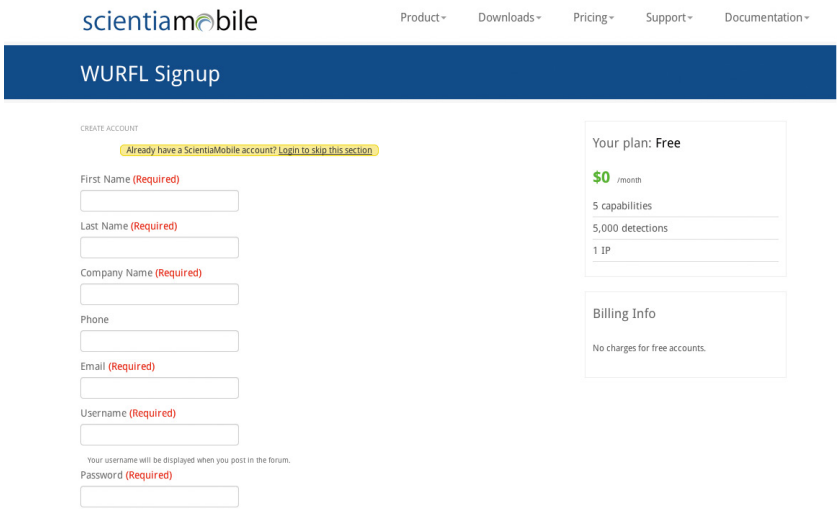

图4-9　注册一个新的账号

在建立账号以后，你需要获取一个API key，你可以在Account Settings页面上获取key，如图4-10所示。

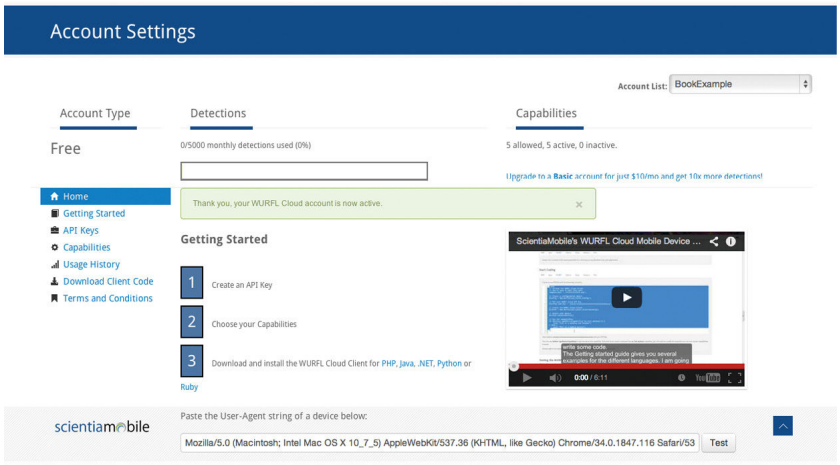

图4-10　在Account Settings页面上配置账号

在Account Settings（账户设置）页面，同样可以选择你需要测试哪一种设备能力（免费版本只提供5个选项）。你可以从可用能力列表中选择需要的，然后拖拽它们到自己的能力列表中，这些列表也决定了如何在你的代码中引用，如图4-11所示。

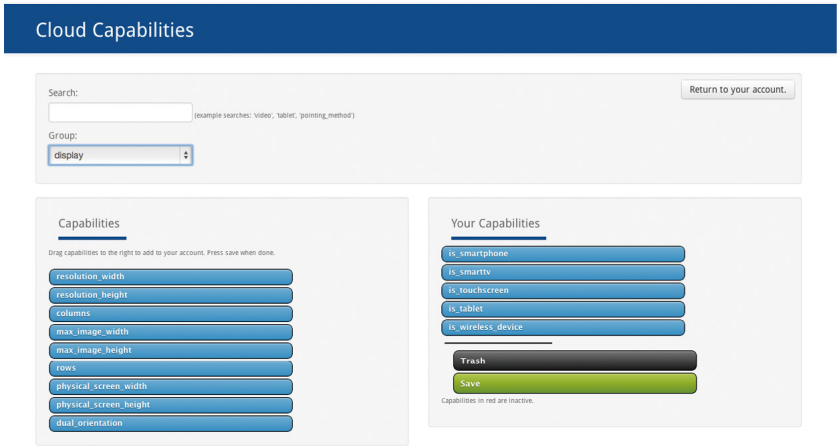

图4-11　从Wurfl Cloud中选择你需要的能力

最后，你需要下载Wurlf Cloud的客户端，然后根据你使用的语言开始写代码，在写本书的时候，Wurfl客户端代码支持下面的语言和技术。

- Java
- PHP
- Microsoft.Net
- Python
- Ruby
- Node.js
- Perl

图4-12展示了Wurfl Cloud客户端的下载界面。

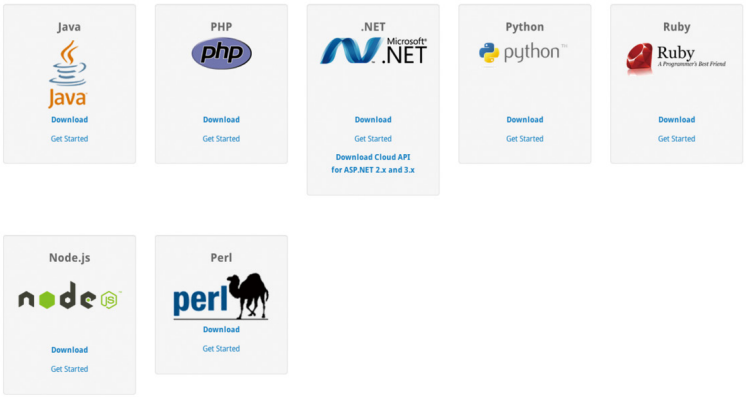

图4-12　根据你的项目情况选择Wurfl Cloud客户端

Wurfl Cloud客户端以一个ZIP文件的形式下载下来，它包含了一些可以在项目中使用的可以和Wurfl Cloud交互的类。

示例代码

下面让我们创建一个使用Wurfl Cloud的应用。在开始编码之前，先假设有如下场景。

你的项目使用的是Node.js，所以应该选择下载Node.js客户端。在下载了Wurfl Cloud客户端ZIP文件以后，解压缩到你的工程可以访问的路径上。像大多数的Node.js应用一样，你已经准备好server.js用来接收请求了，同样，已经准备好router.js路由对应的请求了。index.js，把server.js和应用逻辑（命名为responsiveApp.js）整合起来。下面就是index.js的内容。

```
var server = require('./server/server.js');
var router = require('./server/router.js');
var responsiveApp = require("./responsiveApp.js");

var handle = {}
handle["/"] = responsiveApp.start;
handle["/start"] = responsiveApp.start;
handle["/favicon.ico"] = responsiveApp.favicon;
server.start(router.route, handle);
```

Index.js加载了server.js和router.js，同时也加载了responsiveApp.js（即使你还没有创建它）。创建了一个调用处理器的对象，它会被传递给服务器，服务器会指明如何根据不同的路由URL调用不同的处理函数。在这个例子中，我们只是简单地将所有的请求（除了请求favicon的）映射到responsiveApp.js中的start函数。最后，调用server.start启动。

server.start的作用只是创建一个事件处理器，当接收到HTTP请求的时候就触发事件。随后事件处理器就将请求传递给router.js。在这里，router.js会做一层逻辑判断，先检查请求，然后根据handler对象调用相应的处理函数。

如何深入地讲解Node.js，就超出了本书的范围。如果你想深入了解Node.js,你可以阅读Shelley Powers编写的Learning Node一书。接下来，让我们在responsive-App.js中编码应用的逻辑。首先，加载HTTP模块。然后加载两个我们之前下载的主要文件（如WurflCloudClient.js和Config.js）。

```
var http = require('http');
var wurfl_cloud_client = require("./NodeWurflCloudClient/WurflCloudClient");
var config = require("./NodeWurflCloudClient/Config");
```

接下来，我们创建start函数，在这个函数体里我们只调用了一个函数——getCapabilities，同时，如果我们有favicon，那么我们就创建一个favicon函数用作返回我们的favicon。

```
function start(response, request) {
   getCapabilities(response, request);
}
function favicon(response) {
   response.writeHead(200, {
   'Content-Type': 'image/x-icon'
} );
//write favicon here
  response.end();
}
```

现在我们来看看这个函数的具体逻辑。我们创建了一个getCapabilities函数，将start函数中的两个参数请求和响应也传递给这个函数。

```
function getCapabilities(response, request) {
}
```

接下来我们需要创建两个变量：一个对象——result_capabilities；另一个是数组——request_capabilities。request_capabilities数组存储了我们想要检查的功能。那这些功能从哪里来的呢？这就是我们之前在Wurfl账号里配置的功能。

```
function getCapabilities(response, request) {
var result_capabilities = {};
var request_capabilities = ['is_smartphone','is_tablet','is_touchscreen', 'is_wireless_device']
```

创建一个名为api_key的变量，可以从Wurfl的配置界面获取，这个值也是我们使用API的凭证。再创建一个名为configuration的变量，这个变量存储着调用config.WurflCloudConfig返回的值。

```
var api_key = "XXXXX ";
var configuration = new config.WurflCloudConfig(api_key);
```

下面我们将初始化一个WurflCloudClient的实例，WurflCloudClient的构造函数有三个构造参数：request、response和之前已经初始化的configuration实例，我们把这个实例命名为WurflCloudClientObject。

这个对象是我们访问Wurfl并获取能力的关键。接下来我们需要调用这个对象的detectDevice方法，给这个对象传递三个参数：请求、request_capabilities和一个在查询结果返回时需要触发的回调函数。

```
WurflCloudClientObject.detectDevice(request, request_capabilities,
function(err, result_capabilities){
```

这个匿名回调函数的业务逻辑是根据查询结果为我们定制适当的体验，并且渲染相应的HTML、CSS和JavaScript。在这个简化了的例子中，我们只是简单地调用函数然后输出正确的数据（drawSmartphoneHomepage等）。

基于这种方式，我们无需将所有和设备或者体验相关的代码都放到媒体查询和客户端中，我们仅仅在服务器端返回和设备或者体验相关的代码。

```
if(err!=null){
     console.log("<br>Error: " + err + " <br/>");
}
else{
     if(result_capabilities['is_smartphone']){
          drawSmartphoneHomepage(response);
     }else if(result_capabilities['is_tablet']){
          drawTabletHomepage(response);
     }else{
          drawDesktopHomepage(response);
     }
}
```

作为参考，整个示例代码如下面所示。

```
var http = require('http');
var wurfl_cloud_client = require("./NodeWurflCloudClient/WurflCloudClient");
var config = require("./NodeWurflCloudClient/Config");
function start(response, request) {
   getCapabilities(response, request);
}
function favicon(response) {
       response.writeHead(200, {
       'Content-Type': 'image/x-icon'
   } );
   //write favicon here
   response.end();
}
function getCapabilities(response, request) {
var result_capabilities = {};
var request_capabilities = ['is_smartphone','is_tablet',
'is_touchscreen', 'is_wireless_device']
var api_key = "XXXXX ";
var configuration = new config.WurflCloudConfig(api_key);
var WurflCloudClientObject = new wurfl_cloud_client.Wurfl-CloudClient(configuration, request, response)
WurflCloudClientObject.detectDevice(request, request_capabilities,
function(err, result_capabilities){
   console.log(result_capabilities);
   if(err!=null){
        console.log("<br>Error: " + err + " <br/>");
   }
   else{
      if(result_capabilities['is_smartphone']){
               drawSmartphoneHomepage(response);
      }else if(result_capabilities['is_tablet']){
               drawTabletHomepage(response);
      }else{
               drawDesktopHomepage(response);
      }
               }
   });
}
exports.start = start;
exports.favicon = favicon;
exports.getCapabilities = getCapabilities;
```

4.4 缓存的影响

在开发大型网站的时候，我们常常会依赖于缓存减轻我们的服务器压力。当我们把响应式移到服务端但又缓存了响应时，问题便出现了。不管从客户端传来什么样的User Agent信息，我们将只看到当前缓存的版本而无法看到最新的响应。

为了解决这个问题，我们可以在发送响应的时候使用Vary这个HTTP响应header参数。当有请求进来的时候，服务端会基于User Agent的值做一些判断，然后决定哪些响应需要缓存起来。

> **提示**
>
> 在本书创作期间，大多数的CDN在使用Vary Response header的时候都不会缓存。如果你的CDN也是这样的，你需要针对这种情况有相应的替代方案。可能需要使用Edge Side Includes将User Agent的检查逻辑移到CDN的边缘层。

4.5 Edge Side Include

使用类似于Akamai这样的CDN来缓存内容是一种非常好的方式，它可以有效减少服务器的传输压力，这也就意味着可以有效减少维护的硬件，从而降低成本。不仅如此，CDN可以让内容更快地传递给用户。图4-13展示了这种模式下的整体架构图。

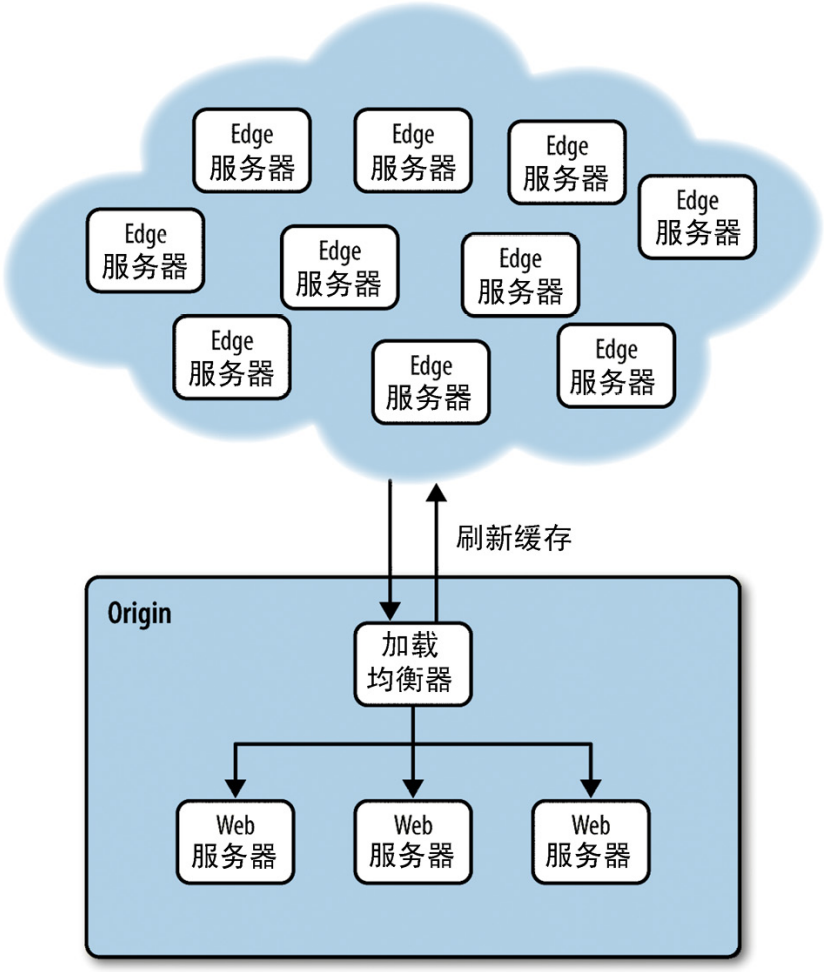

图4-13　通过边缘网络传输缓存内容

就在一个月以前，我们的CDN运营商不允许我们缓存User Agent相关的内容（再一次说明一下，是使用Vary HTTP header），所有的客户端得到的都是同一份缓存，并不是设备相关的内容。所以解决这种问题的方法就是使用Edge Side Include（ESI）语言。ESI由一些大公司制定的，包括Akamai和Oracle，并且已经提交给了W3C。你可以在*http://bit.ly/1rY5WUO*获取ESI的详细规范。

ESI是一种标识语言，嵌入到了HTML文档中。EDGE服务器都有一个ESI处理逻辑，用来读取ESI标签，解析逻辑，把输出渲染到HTML中。和PHP类似，ESI非常类似一种服务端脚本语言，它可以在服务端进行解析，然后输出到HTML中。而且和PHP更相似的是，ESI标签不会在客户端展示出来，它只会展示渲染以后的内容。

下面的代码是ESI的示例脚本，根据请求的User Agent数据，加载合适的内容。

```
<html>
<head></head>
```

```
<body>
<esi:choose>
     <esi:when test="$(HTTP_USER_AGENT{'os'})=='iPhone'">
     <esi:comment text="Include iPhone specific resources here" />
…
     </esi:when>
     <esi:when test="$(HTTP_USER_AGENT{'os'})=='iPad'">
          <esi:comment text="Include iPad specific resources here"/>
…
     </esi:when>
     <esi:when test="$(HTTP_USER_AGENT{'os'})=='Android'">
          <esi:comment text="Include Android specific resources here" />
…
     </esi:when>
     <esi:otherwise>
          <esi:comment text="Include desktop specific resources here" />
…
     </esi:otherwise>
</esi:choose>
</body>
</html>
```

4.6 小结

本章从多个角度全面地探讨了我们的应用程序。从应用底层探讨了协议和软件栈，并且探讨了在整个请求流程中，我们应用有哪些步骤需要处理。站在一个更高的视角上，我们不得不问自己一个问题：如何尽快从最早的请求中获取客户端设备功能？最重要的是如何尽快对响应进行处理？

为了回答这个问题，我们重点对请求的HTTP请求中的User Agent进行检测，甚至使用了第三方的设备检测服务，比如Wurfl。

不过这个解决方案的一个潜在风险就是如何处理缓存内容。一个解决方式就是使用响应header中的Vary参数。缓存服务器通过User Agent决定哪一种响应需要缓存起来。另外一种方式就是通过使用ESI，把我们的设备或者能力检测逻辑从我们的服务器移到CDN edge服务器上。

不管哪一种解决方案，只要我们将响应式的处理在服务端解决，避免客户端加载所有的响应式相关代码，避免提供两份相同内容或者无效内容的反模式，就可以减少向客户端传送的负载，取而代之的是一种更精简、更合理的响应方式，所以这种实现方式更好。而做这些复杂技术和架构的原因就是，这种实现方式对带宽、电池寿命、CPU限制等这些终端用户的设备存在的种种限制，都是一种有效的优化手段。

第5章　响应式前端实现

第4章展示了如何将响应式范式从客户端迁移到服务端。这个过程包括两个方面：第一，从服务端加载符合设备特性的内容和提供专有的体验；第二，避免加载所有设备内容的反模式。这样做可以有效降低页面的负载，减少客户端设备渲染页面的整体时间。

但是如果你的基础设施、商业模式或者团队技术深度不能够有效地支持服务端的解决方案，那该如何做呢？在这种情况下，有一些方法可以从单纯的客户端解决方案中就能够获得相近的性能提升。在本章中，我们将焦点放在前端上，讨论一些其他的方式来实现相同的模式。

5.1 图片操作

Steve Souders向我们展示了用他的Interesting Stats页面所收集到的HTTP统计信息，结果显示，对页面负载影响最大的就是页面上的图片了（见图5-1）。可以肯定地说，要在客户端提高响应式性能，优化移动设备客户端的图片传输非常重要。

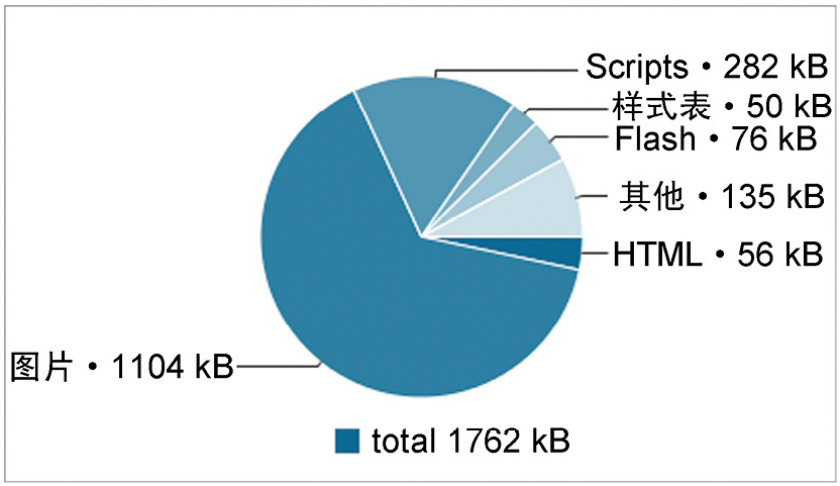

图5-1 HTTP Achieve显示的页面负载组成的资源类型分布图

在过去，响应式图片被看成需要和窗口页面大小保持同比例。第1章展示了几种方式之间的横向比较：通过CSS调整图片大小，或者在一些需要的场景中，下载两倍的图片尺寸，然后通过CSS将它们缩小。这里需要再次重申的是，这些解决方案都确实是性能的反模式。实现方式是：加载设备所需的所有资源或者加载两倍尺寸的资源。

为了真正解决用户在响应式方面的性能需要——包括带宽限制、电池寿命，像素密度和视窗大小——我们必须遵循只加载适合于设备资源的模式。

响应式的各个方面，特别是响应式图片，很明显是一个需要标准化的领域，一些推荐的解决方案正在起草。下面让我们来看看具体内容和其他一些技术选项。

SRCSET属性

当前草案里的一个选项是使用srcset属性，它是用在<img>标签中为加载响应式图片服务的，它也是不久前才被W3C增加进来的。草案中对srcset属性的具体描述可以在*http://bit.ly/1tDH5Lr*中找到。详细信息更高级的是，通过srcset属性，<img>标签可以为不同客户端设备的不同像素比指定不同的图片。下面让我们看看具体是怎么使用的。

```
<img src="1x.jpg" srcset="2x.jpg 2x">
```

设备像素比

正如前面的代码所示，<img>标签中有个默认的图片1x.jpg，这主要是为了向后兼容，防止浏览器不支持srcset。如果设备像素比是2，那么我们设置的srcset属性和它指定的图片就会起作用了——在本示例中我们指定了2x.jpg。设备像素比是物理像素和设备独立像素之间的比例。典型的例子就是iPad Retina屏，iPad Retina屏是1024物理像素宽，但是因为它使用了Retina屏，包含更多信息的像素或者设备独立像素，实际上它的真实像素宽度是2048。所以，iPad Retina屏显示的设备像素比的计算公式为[设备独立像素]除以[物理像素]，也就是2048/1024 = 2。

如果你对这些信息感兴趣，可以在*http://bit.ly/1uBP6RI*找到Peter-Paul Koch对于这方面的详细讲解。

浏览器使用的设备像素比的值可以使用window.devicePixelRatio属性来获取。图5-2提供了srcset的屏幕截图。在这个例子中，我们使用Google Chrome对Motorola Droid Razr HD的仿真性展示，在这里，我们使用的devicePixelRatio是1。

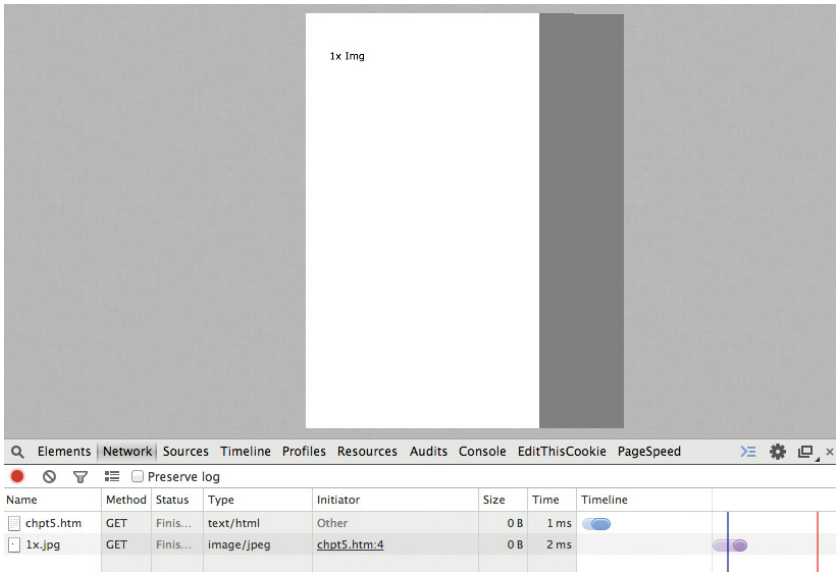

图5-2　模拟Motorola Droid Razr HD

Motorola Droid Razr HD使用的是720×1280的分辨率显示，它的devicePixelRatio是1，在这种情况下，我们的1x.jpg就会被加载。下面是我们的User Agent值。

```
Mozilla/5.0 (Linux; U; Android 2.3; en-us; DROID RAZR 4G
Build/6.5.1-73_DHD-11_M1-29) AppleWebKit/533.1 (KHTML, like
Gecko) Version/4.0 Mobile Safari/533.1
```

图5-3展示了Chrome对iPad 4的仿真，iPad 4使用了Retina屏，它的devicePixelRatio是2。

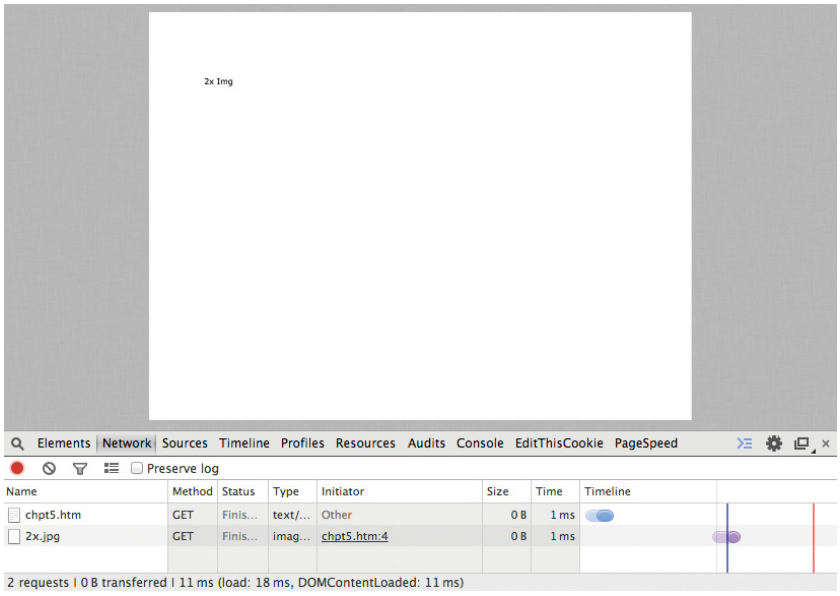

图5-3　对Apple iPad4的模拟显示

iPad 4使用了2048×1536的显示器分辨率并且它的devicePixelRatio是2，所以加载了图片2x，下面是iPad的User Agent。

```
Mozilla/5.0 (iPad; CPU OS 7_0 like Mac OS X) AppleWebKit/
537.51.1 (KHTML, like Gecko) Version/7.0 Mobile/11A465 Safari/
9537.53.
```

在上面使用的两个例子中，在Developer Tools的Network标签下显示它们都只加载了需要的图片。在这里需要注意的是，这两个屏幕截图中，我们都是使用了模拟的设备。目前srcset属性还没有得到全面支持，所以在使用的时候仍然需要检查你的使用日志，获取你的应用上使用最为广泛的设备列表，测试并确保在这些设备上已经支持了srcset属性。

使用srcset属性的缺点是，你必须指定我们可能需要的所有不同的图片，这会额外增加发送的字节数，从而增加额外的负载。如果你想进行更深入的优化，Ilya Grigorik在其*High Performance Browser Networking*一书中，提供了一种非常简练的方式，把像素比的映射关系放到服务端来完成，你可以在*http://bit.ly/1qnPSeY*上查看到详细信息。

srcset属性的优点，除了可以让你指定设备的特定使用的多个图像，而无需加载多余的图像之外，还有就是现代的浏览器已经开始支持这个属性了。而我们接下来要讨论的主题——picture元素——则不是这样。

picture元素

处理响应式图片的另外一个可选性技术是使用<picture>元素。你可以*http://www.w3.org/TR/html-picture-element/*查看W3C的工作草案。

<picture>元素是HTML5中新增的一个元素。从概念上讲，它是一个容器元素，包含了多种不同的源标签，基于不同的视口宽度像素密度指定不同的图片，它也可以容纳一个img标签进行优雅降级。<source>元素支持media属性，使用它可以指明要关注的媒体类型和目标CSS属性。source元素还有src属性，使用这个属性来指明针对目标的媒体类型和CSS属性所对应要下载的图片。

针对高像素显示的平板电脑和手机，我们使用<picture>元素重写先前的srcset示例，会得到如下示例代码。

```
<picture>
   <source media="(min-width: 640px, min-device-pixel-ratio:2)" src=" hi-res_small.jpg ">
   <source media=”(min-width: 2048px, min-device-pixel-ratio:2)” src=” hi-res_large.jpg ">
   <img src="1x.png ">
</picture>
```

让更有趣的是,它同样可以支持srcset属性。这两种标签的功能组合代码如下所示。

```
<picture>
   <source srcset="big.jpg 1x, big-2x.jpg 2x, big-3x.jpg 3x" type="image/jpeg" media="(min-width: 40em)
   <source srcset="med.jpg 1x, med-2x.jpg 2x, med-3x.jpg 3x" type="image/jpeg" />
   <img src="fallback.jpg" alt="fancy pants" />
</picture>
```

不管是srcset属性还是<picture>元素，它们都是我们可能用到的解决方案。如果从性能的角度来比较两种解决方案，我们会发现，在理论上，它们都会只下载最适合客户端设备特性的资源，但是<picture>元素很明显比使用了srcset属性的<img>标签更冗长。如果我们想对它们使用的字节数进行量化比较，以本章中我们使用的代码为例，使用srcset属性的<img>标签使用了95个字节，而<pictrue>元素使用了231个字节，srcset属性的方式比<picture>元素的方式节约了近60%的字节数。

图5-4展示了二者之间全面的横向比较。当我们看到这些比较时，似乎有这样一种感觉，95和231个字节数似乎无伤大雅。但是这仅仅是一个<img>标签的字节数，如果是整个网页或者大流量的访问情况下

呢？现在让我们温习一下第1章中Alexa顶级站点的数据集。如果我们使用这些站点的数据集，只是把<img>标签放到这些站点页面中，那么它们所消耗的字节数如表5-1所示（注意它们是以K为单位的）。

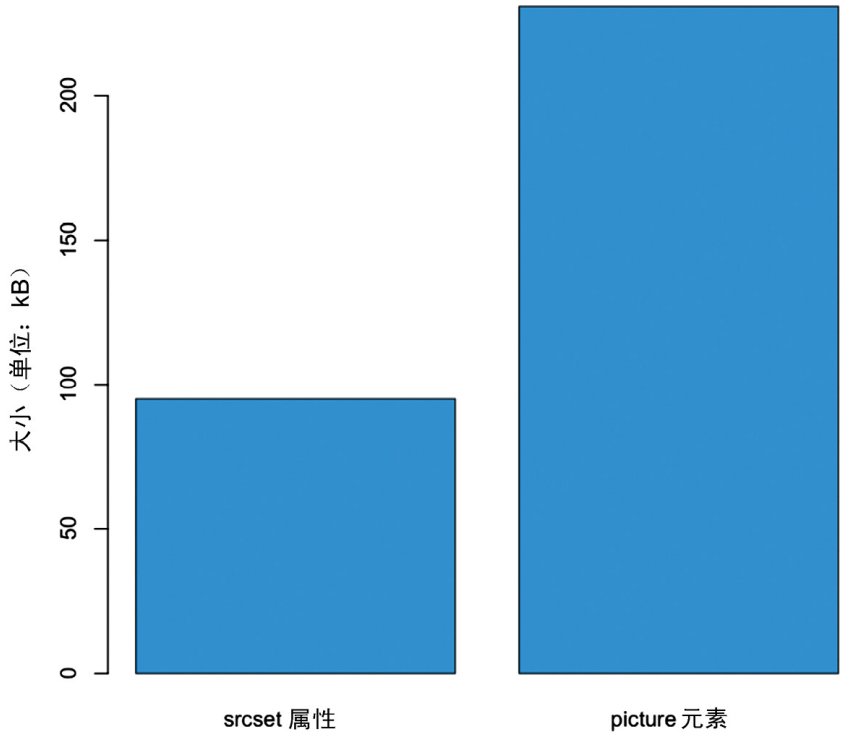

图5-4　使用了srcset的IMG标签和picture元素所消耗的不同字节数之间的比较

表5-1　数据集字节数小结

MIN	0.000
1 ST QUARTILE	0.305
MEDIAN	3.650
MEAN	56.507
3 RD QUARTILE	62.125
MAX	371.100

最坏的情况下，仅仅<img>标签文本就消耗了371KB，而不包含其他页面元素，如HTML、CSS或者JavaScript等。当然，有一些文件的大小非常有可能是被用作追踪的标识或者间隔符而产生的，它们不会为不同的设备请求不同的版本。从这些数字进行推断，就会发现图5-5所示的使用<picture>元素代替<img>元素对性能产生的影响。

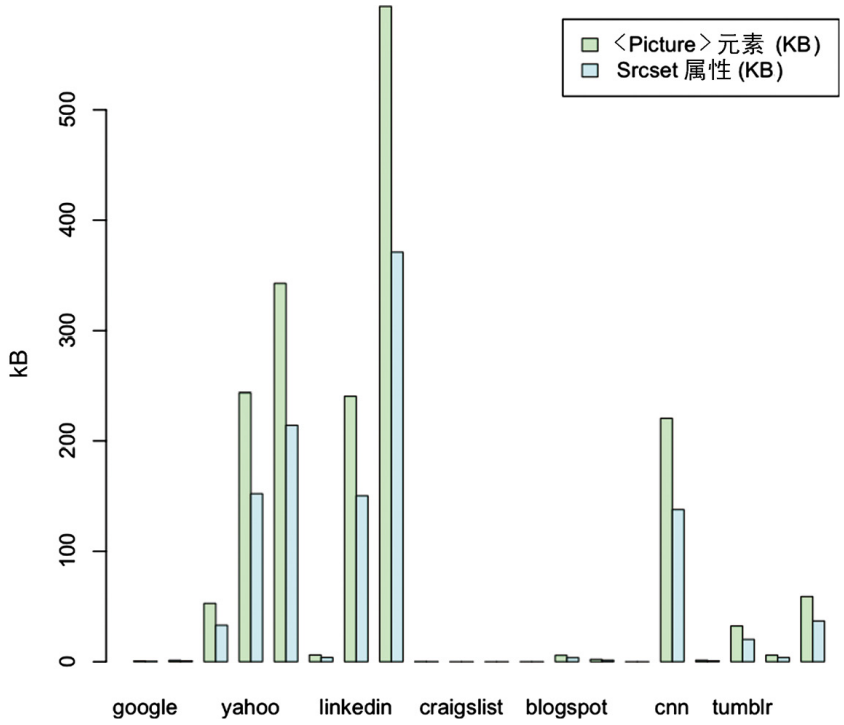

图5-5　推断出不同的字节数

根据推断，使用<picture>元素的总结如表5-2所示。

表5-2　使用<picture>元素的总结

MIN	0.000
1 ST QUARTILE	0.488 (+0.183 KB)
MEDIAN	5.840 (+2.19 KB)
MEAN	90.411 (+33.904 KB)
3 RD QUARTILE	99.400 (+37.275 KB)
MAX	593.760 (+222.66 KB)

实事求是地说，测试的站点有了75%的流量增长，这远比我们加载额外图片所带来的流量要小，不过值得关注的是它的极限值。我们来看看数据集中最大的流量——现在几乎已经有600KB了。很明显，虽然<picture>标签在未来某个时候会被现代的浏览器全面支持，它也会提供可靠的方式来加载响应式图片，因为页面传输文件大小产生的影响，它应该作为一个可选的解决方案，而不是解决加载任何响应式图片问题的默选方案。

重要的是，在这些示例中使用了<picture>元素将增加数以百计的标记字节数，而不使用<picture>元素将增加数以万计的图片字节数，当然你也可以使用压缩来减少这种方式对负载带来的影响。

5.2 延迟加载

到目前为止，我们已经在本章中讨论图片相关的内容。现在让我们重新思考一个问题：如何在客户端使用某一个策略——在页面渲染时只加载适应于该设备的资源？第1章展示了客户端的相关方法，这个方法包含了延迟加载。

使用延迟加载，它会在真正需要的时候才会去加载相关的内容。一个延迟加载的例子就是无限滚动：只有真正需要显示的内容才会被显示到“显著位置”（设备中展示在屏幕范围内的内容），当用户向下滑动后，更多的内容才会被下载并渲染显示到屏幕上。因此，我们第一次需要加载HTML语义结构的骨架，然后根据客户端的特性，延迟加载相应的CSS和JavaScript。

整个架构如图5-6所示。

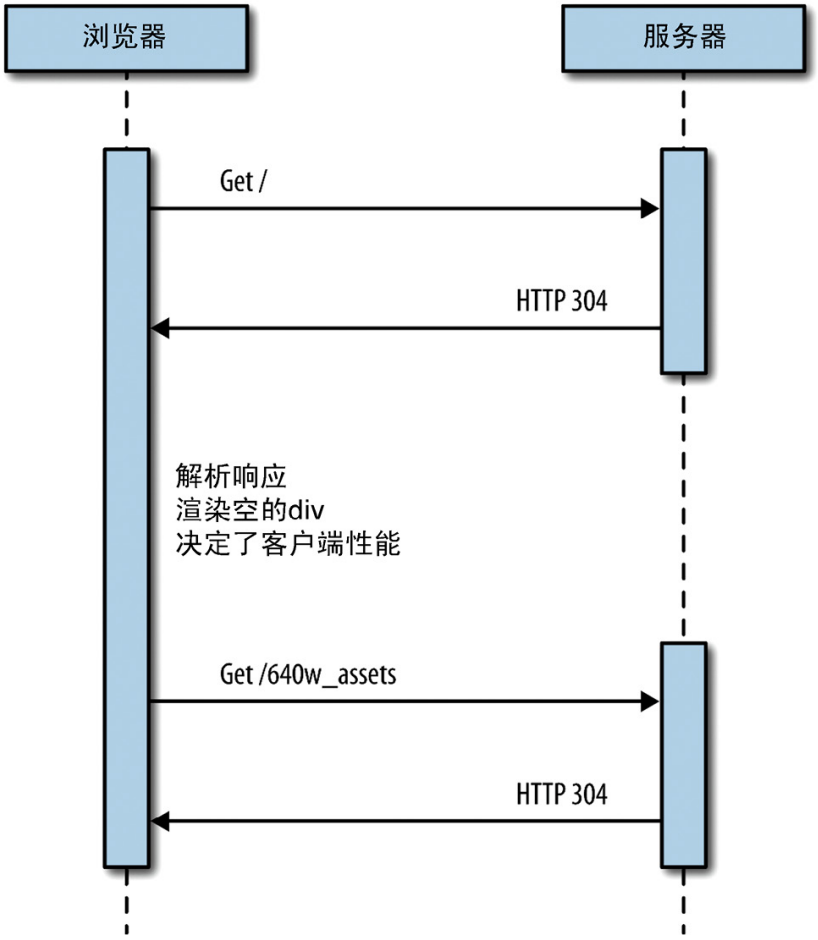

图5-6 延迟加载适合于客户端的相关内容

让我们来看一个具体的例子。首先，我们将加载基本的HTML骨架结构，只包含最少的没有格式化的内容，内容中有包含了id的 标签，id是用来表明什么内容需要被加载到这些标签中（head、body、footer和响应式）。

```
<html>
<head></head>
<body>
<h2>Lazy Loading Example</h2>
<div id="head"></div>
```

```
<div id="body">
        Loading Content
     </div>
<div id="footer">
</div>
</body>
</html>
```

接下来，我们将<script>标签放在body的底部，创建一个名为determineClient()的函数，在这个函数中，我们创建了一个名为client的对象，这个对象同样包含了一个名为sectionURL的对象，这个对象里，我们有一些属性——head、body和footer，它们的命名取决于我们页面中div的id属性。

```
<script>
function determineClient(){
        var client = {
             sectionURLs: {
                head: "/components/head/",
                body: "/components/body/",
                footer: "/components/footer/"
             }
        };
   }
</script>
```

这样做主要是因为需要在确定客户端的特性或者特有的体验之后，才根据这些属性加载合适的页面内容。为了防止无法确定客户端的特性，在这里我们加上了一些默认的数据。

现在，我们将增加一些分支逻辑来测试window.innerWidth和window.devicePixelRatio，然后根据它们来填充sectionURL。在我们的示例中，假设我们的目录结构设置是基于尺寸大小的，如图5-7所示。

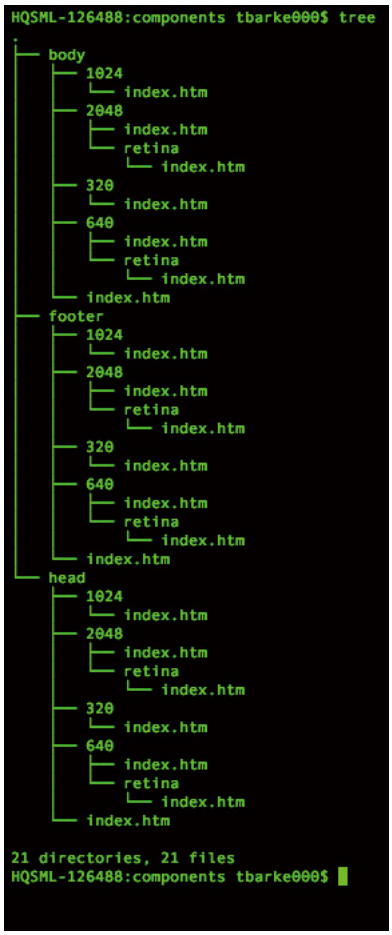

图5-7　示例网站的目录结构

在图5-7中，目录结构是根据视窗宽度来定义的，包括像素密度设备的目录。同样需要注意的是，每一个顶级目录（head、body、footer）都有它独有的index.htm文件，这样也是为了确保客户端可以加载到默认的内容。当然，图中的目录并不一定需要物理文件，它们可以通过Apache的mode_rewrite规则或者其他任何类型的URL处理功能来实现。

在我们的分支逻辑填充完client对象后，determineClient()函数返回了client对象，如下所示。

```
<script>
   function determineClient(){
      var client = {
           sectionURLs: {
              head: "/components/head/",
              body: "/components/body/",
              footer: "/components/footer/"
        }
   };
     if(window.innerWidth == 320){
           client.sectionURLs.head = "/components/head/320/";
           client.sectionURLs.body = "/components/body/320/";
           client.sectionURLs.footer = "/components/footer/320/";
     }else if(window.innerWidth == 640){
           if(window.devicePixelRatio == 1){
                client.sectionURLs.head = "/components/head/640/";
                client.sectionURLs.body = "/components/body/640/";
                client.sectionURLs.footer = "/components/footer/640/";
        }else if(window.devicePixelRatio >=2){
                client.sectionURLs.head = "/components/head/640/retina/";
```

```
                client.sectionURLs.body = "/components/body/640/retina/";
                client.sectionURLs.footer = "/components/footer/640/retina/";
        }
    }else if((window.innerWidth == 1024) || (window.innerWidth == 1440)){
            client.sectionURLs.head = "/components/head/1024/";
            client.sectionURLs.body = "/components/body/1024/";
            client.sectionURLs.footer = "/components/footer/1024/";
    }else if(window.innerWidth == 2048){
       if(window.devicePixelRatio ==2){
              client.sectionURLs.head = "/components/head/2048/retina/";
              client.sectionURLs.body = "/components/body/2048/retina/";
              client.sectionURLs.footer = "/components/footer/2048/retina/";
       }
       }
     return client;
}
</script>
```

如果将client对象在控制台展示的话，内容如下所示。

```
Object {sectionURLs: Object}
sectionURLs: Object
body: "/components/body/1024/"
footer: "/components/footer/1024/"
head: "/components/head/1024/"
```

接下来，我们将创建一个名为loadSection的函数， 将client对象和我们指定的 作为参数传递进去。这个函数有非常典型的XMLHttpRequest对象模板代码，它是为了从服务端获取相应的内容。主要的步骤如下所示。

- 创建一个基于xhr的一个临时对象，然后将它用于传递给函数的section参数。
- 一旦数据从服务端成功返回，回调函数将被触发，匹配到ID的innerHTML元素将会被xhr对象中的responseText重写。

```
function loadSection(section, client){
        var xhr = new XMLHttpRequest();
        xhr.open("get", client.sectionURLs[section], true);
        xhr.section = section;
        xhr.send();
        xhr.onload = function(){
           document.getElementById(xhr.section).innerHTML = xhr.responseText;
           }
}
```

现在所有的功能都有了，剩下的就是如何将所有的逻辑整合到一起了。接下来，我们将创建一个函数，在window.load()事件触发的时候，这个函数将被执行。这个函数同时作为一个控制器，创建一个变量保存determineClient函数调用获取的client对象，然后为所有的section调用loadSection()函数。

```
window.onload = function(){
     var client = determineClient();
     var sections = ["head", "body", "client"];
     for(var n=0;n<sections.length(),n++){
            loadSection(n, client);
     }
}
```

在浏览器中运行我们的示例代码，运行结果可以在Network标签下查看，如图5-8所示。

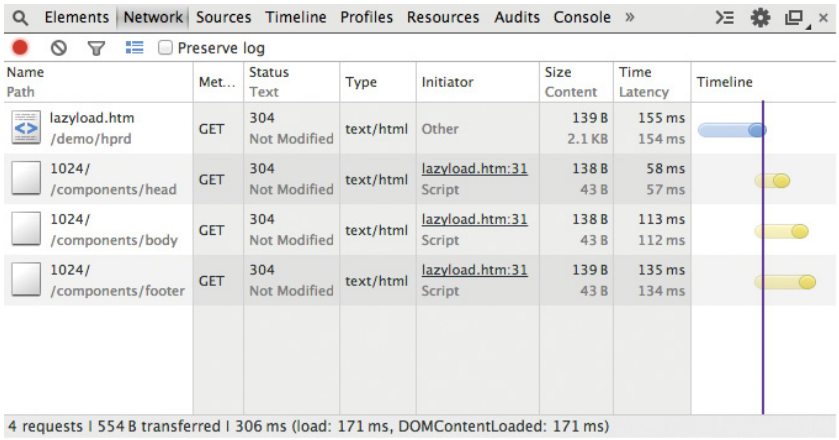

图5-8　瀑布图显示了head、body和footer在页面加载完成以后的延迟加载情况

我们可以看到基本的页面渲染在171毫秒内完成，延迟加载的内容在131毫秒内完成。

你可以从*http://tom-barker.com/demo/hprd/lazyload.htm*下载完整的示例代码。

```
<html>
<head></head>
<body>
<h2>Lazy Loading Example</h2>
<div id="head"></div>
<div id="body">
          Loading Content ...
     </div>
     <div id="footer">
     </div>
     <script>
       window.onload = function(){
         var client = determineClient();
         var sections = ["head", "body", "client"];
         for(var n=0;n<sections.length(),n++){
         loadSection(n, client);
           }
     }
     function loadSection(section, client){
        var xhr = new XMLHttpRequest();
        xhr.open("get", client.sectionURLs[section], true);
        xhr.section = section;
        xhr.send();
        xhr.onload = function(){
            document.getElementById(xhr.section).innerHTML =
            xhr.responseText;
            }
   }
   function determineClient(){
      var client = {
         sectionURLs: {
            head: "/components/head/",
            body: "/components/body/",
            footer: "/components/footer/"
            }
      };
     if(window.innerWidth == 320){
          client.sectionURLs.head = "/components/head/320/";
          client.sectionURLs.body = "/components/body/320/";
          client.sectionURLs.footer = "/components/footer/320/";
     }else if(window.innerWidth == 640){
       if(window.devicePixelRatio == 1){
            client.sectionURLs.head = "/components/head/640/";
            client.sectionURLs.body = "/components/body/640/";
            client.sectionURLs.footer = "/components/footer/640/";
     }else if(window.devicePixelRatio >=2){
            client.sectionURLs.head = "/components/head/640/retina/";
            client.sectionURLs.body = "/components/body/640/retina/";
```

```
            client.sectionURLs.footer = "/components/footer/640/retina/";
          }
    }else if((window.innerWidth == 1024) || (window.inner - Width == 1440)){
         client.sectionURLs.head = "/components/head/1024/";
         client.sectionURLs.body = "/components/body/1024/";
         client.sectionURLs.footer = "/components/footer/1024/";
    }else if(window.innerWidth == 2048){
              if(window.devicePixelRatio ==2){
                   client.sectionURLs.head = "/components/head/2048/retina/";
                   client.sectionURLs.body = "/components/body/2048/retina/";
                   client.sectionURLs.footer = "/components/footer/2048/retina/";
              }
    }
    return client;
}
</script>
</body>
</html>
```

注意

本例中我们同时对格式和内容都进行了延迟加载，这是一个极端的例子。在某些场景中，你可能只需要延迟加载格式或者功能。但是在某些场景下，你会发现延迟加载内容也非常有用。或许你在尝试一些新的图像格式，例如WebP或者JEPG XR，现在的浏览器还没有全面支持这些格式，你可以只为支持它们的浏览器加载更轻量级的内容（指的是使用WebP、JPEG XR代替JPG）。也许，像我的一个团队最近正在做的那样，在电视机顶盒上开发web内容，并且不同的机顶盒支持不同的视频播放模式。在这种情况下，你需要为每个特定的机顶盒延迟加载它所支持格式的那些视频。

有些规则我们必须牢记于心：浏览器将自动延迟加载CSS背景图片。如果display属性被设置成none，那么背景图片只有当display属性设置成visible的时候才会被加载。这是页面中延迟加载默认图片的另外一种使用策略。

设备检测库

在客户端测试设备特性是一件自然而容易的事情，但如果想获取精确的形状和准确的设备类型还是非常困难。你可以把所有知道的设备特性收集起来作为参数，这样我们就无需考虑形状要素，但也不会去考虑诸如网络稳定性等类似问题。我们可以解析User Agent来获取形状要素相关的数据，不过我们仍然需要一个映射表，通过这个映射表的关联标记，从User Agent中找到指定的设备和形状要素。

如果不想维护这张映射表，依靠于第三方组织有什么优势？依赖于第三方组织，就可以在无需维护User Agent和设备数据库的情况下，精确地获取目标的特征形状要素。Wurfl和Device Atlas都提供了获取设备特性的JavaScript的客户端组件库。Device Atlas打包了他的JavaScript库和client。Scientiamobile有一个站点，*http://wurfl.io/* 致力于推广它们的客户端解决方案：wurfl.js。图5-9展示了wurfl.io的主页。

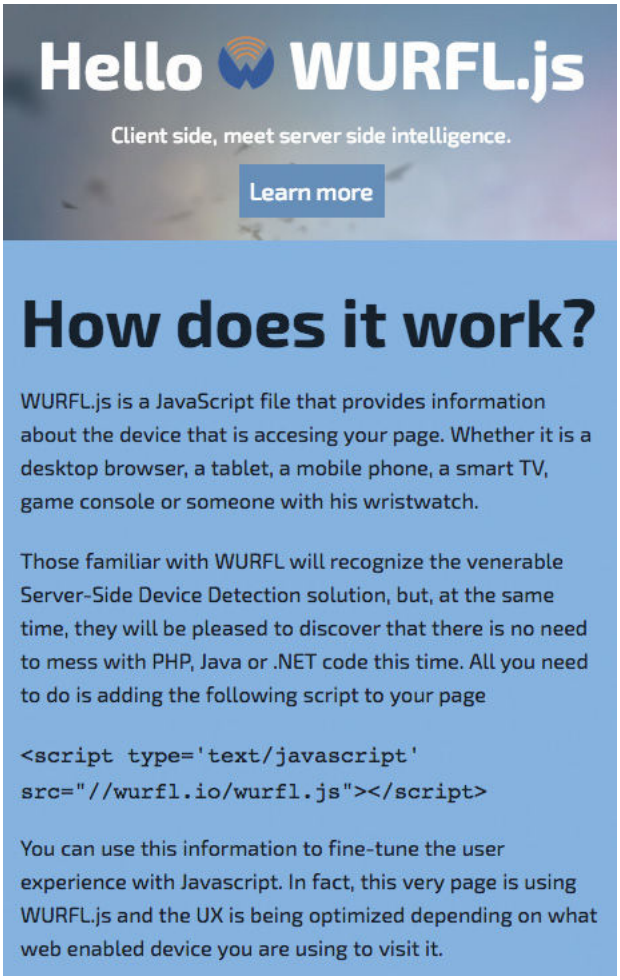

图5-9　Scientiamobile的wurfl.io网页

只要简单地引入一个标签就可以使用wurfl.js的功能了。

```
<script type='text/javascript' src="http://wurfl.io/wurfl.js">
</script>
```

创建一个全局变量名叫WURFL。如果将WURFL对象输出到控制台，将看到如下信息。

```
>WURFL
Object {is_mobile: true, complete_device_name: "Apple iPad",
form_factor: "Tablet"}
```

WURFL对象标出了客户端是否在移动设备上、设备的名称和设备的形状要素。很显然这不是一个完整的特性列表，它只是对我们已知的客户端信息的一个增强。

缺点当然也很明显，就是增加了一个额外的外部调用，在我们将页面最终展示给用户的时候，增加了一些潜在的页面负载和延迟。

5.3 小结

本章重点展示了Web站点所需的前端软件栈。首先我们探讨了响应式图片存在的问题以及新的HTML5标准新增的定位响应式图片的工作草案。我们比较了<img>标签中新增的srcset属性和即将到来的<picture>元素，同时我们也分别探讨了使用它们对页面产生的负载影响。

然后我们又探讨了延迟加载在整个页面区块中的使用，避免加载暂时不需要的样式和内容。这些示例程序其实和第4章内容很相似，在第4章中，我们采用了只加载设备相关的内容和格式的策略。不过第4章中这种延迟加载是通过服务端来实现的，本章中我们是通过客户端来实现的。

这些方式都有其优点和缺点。当解析从服务端获取的体验内容时，我们需要格外小心缓存语义，因为不同的体验内容可能会从同一个URI中获得。当从客户端来实现不同的体验内容的时候，你需要在客户端运行代码，并且通过网络连接加载额外的资源。

在第6章中，我们将深入探讨持续集成相关知识，并且探讨如何将网站的响应式和性能检查纳入持续集成环境中。

第6章　持续测试Web性能

6.1　保持一个稳定的过程

任何一个学习过系统理论知识的学生都知道，当你为系统做一些优化的时候，你需要反复且持续不断地检查优化点的状态和过程的正确性。就像一个温度控制器调节一个区域的温度一样，在系统新特性的开发阶段，你需要把Web性能指标保持在SLA中的一个合理范围内。一般来说，在控制系统中，通过循环反馈工具访问系统的输出，如果有必要，应及时更正。在更高的层次上，它们就像图6-1所示流程那样工作，对一个过程的处理结果进行评估，然后将这个评估反馈结果作为一个输入再次按照上面流程进行处理。

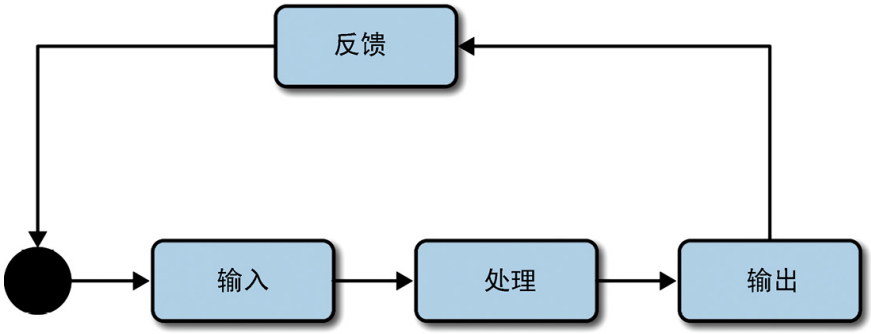

图6-1　基本的循环反馈机制图

对软件工程来说，在持续反馈过程中的一个非常有效的程序环节就是持续集成。持续集成（CI）作为一种非常好的机制，它可以在新代码提交以后，检查各种软件指标和阻止构建——有效遏制了问题代码的提交和部署，直到各种指标达到预期。图6-2展示了在持续集成工作流模式下，新的持续反馈图。

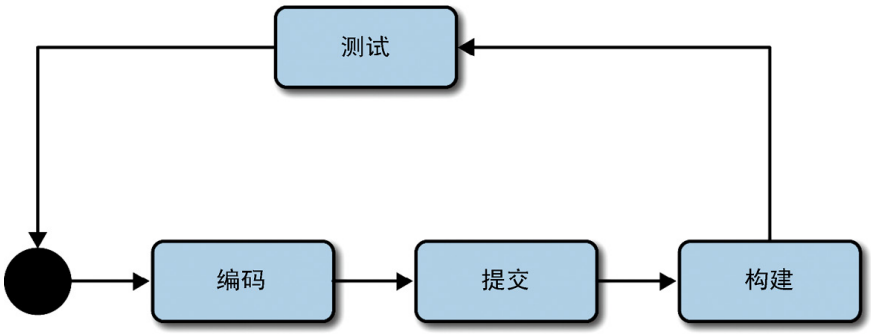

图6-2　CI模式下的持续反馈流程图

此时，你可能非常渴望在你的部门建立起CI环境，并且通过它为你的程序构建运行自动测试。或许你已经开始使用像Jenkins、Anthill，或者其他各式各样可用的CI工具。但是我敢打赌，你的自动测试套件并不能够测试Web性能，或者不能测试在不同视窗尺寸和不同体验下的Web性能。所以接下来，让我们来改变这个局面。

6.2 Web响应式性能自动测试

如果我们在5～10年之前讨论Web开发，关于Web测试驱动开发的概念很少有人知道。但是在最近5年多的时间内，围绕着Web测试驱动开发，关于什么是可行的、什么又是主流的相关讨论已经如火如荼地开展起来了。

单元测试框架，像Pivotal Labs的Jasmine已经发布了，Web开发者也开始在他们的JavaScript中进行单元测试了。headless web browser测试框架（headless web browser是一个没有图形用户接口的Web浏览器。通过它，我们就能以编程的方式——比如测试和自动化——访问Web任务页面）已经发行，并提供了可以在集成测试中完成的一些功能。

如果我们想测试Web响应式的性能，headless web browsers是一个极好的选择，因为它提供了下面的一系列功能。

- 允许从终端运行浏览器脚本。
- 集成进了CI软件。
- 允许自动调整视窗大小。
- 允许以编程方式分配User Agent。
- 可以看到页面加载资源的内部细节。

最流行的一种headless browser测试库是PhantomJS（*http://phantomjsorg*）。PhantomJS是由Ariya Hidayat创建的一个JavaScript API，它可以通过提供的接口访问Webkit（或者更为特别的QtWebkit）。最棒的是，你可以通过命令行来运行PhantomJS，所以就可以把你的测试用例集成到CI工作流中。现在，让我们来学习一下如何在不同的视窗尺寸和User Agent下使用PhantomJS测试你的网站性能。

headless browser自动测试

首先你需要安装PhantomJS，通过在控制台或者命令行输入以下命令，你就可以轻松地完成。

```
sudo npm inst
all -g phantomjs
```

PhantomJS将被安装在全局目录，所以不管在任何目录下，我们都可以使用它。为了确保PhantomJS已经被成功安装，我们需要在命令行上输入命令，检查版本，例如：

```
phantomjs --version
1.9.7
```

使用PhantomJS时的核心流程就是创建一个页面对象，并且使用这个对象加载和分析页面。

```
var page = require('webpage').create();
page.open('http://localhost:8080/', function (status) {
});
```

使用PhantomJS运行上面代码的一般步骤是，先将你的代码保存到一个文件中，然后在命令行上运行这个文件：

```
>phantomjs filename.js
```

基于功能关注点的不同，PhantomJS被分成了一系列的API模块。PhantomJS中包含的模块有如下几种。

系统模块

这个模块提供了通过命令行连接其他模块并获取参数等功能，你可以让你的脚本更灵活，可以不需要在脚本中硬编码视窗尺寸User Agent路径列表等一系列参数，而这一切仅仅通过传递参数的方式——例如一个URL列表——就可以实现。我们也可以使用系统模块访问环境变量和获取系统信息。你可以使用如下代码使用系统模块。

```
var system = require('system');
console.log(system.args, system.env);
```

网页模块

使用这个模块，你可以下载和评估网页。网页模块的优雅之处在于当它创建这个网页的时候，你有了检查网页和网络传输各方面信息的能力。同时你也可以向这个页面中插入自定义内容，比如当你请求这个页面的时候，向请求中插入HTTP header信息，下面代码就是使用网页模块的示例代码。

```
var page = require('webpage').create();
page.open('http://localhost:8080/', function (status) {
});
```

Web服务器模块

使用这个模块，你可以在页面和远程资源之间监听和代理它们的连接事务。同样可以使用Web服务器模块向一个本地端口输出，以下是示例代码。

```
var webserver = require('webserver');
var server = webserver.create();
var service = server.listen('127.0.0.1', function(request,response){
});
```

文件系统模块

文件系统模块提供了访问本地文件系统的功能，例如读写文件或者目录。你可以使用如下代码使用这个模块。

```
var fs = require('fs');
var file = fs.open('[local file] ', '[Open mode]')
```

同时你可以在*http://bit.ly/13DeMD2*找到PhantomJS的完整文档。自从有了PhantomJS，你就可以对下面列出的各个方面进行测试。

- 是否基于客户端的能力加载了合适的资源。
- 每种体验的负载能力是否在我们预先定义的SLA范围之内。

让我们看一下如何使用PhantomJS达到上面所述的要求。

评估资源加载

第一个测试用例是用来确保我们的页面加载了正确的资源，在前面我们已经详细探讨过了为什么需要让客户端设备加载合适的资源（为了减少尺寸负载，对于不同的带宽质量和可用性等级，提供不同的视窗尺寸）。我们也详细讨论了为了在前后端达到这样的目的，迄今为止我们所用的方式方法。但是现在，我们将要以编程的方式来验证这一切是否达到预期目的。

我们可以通过伪造视窗大小或者User Agent,然后评估在该条件下加载的资源。在下面的例子中，我们将使用网页模块创建一个模拟的页面，设置视窗属性（用来接收宽度和高度的JSON对象数据），然后模拟UserAgent属性，设置到这个网页中，以便让Web服务器认为这个请求是从iPhone5发起的。

```
var page = require('webpage').create();
//simulating an iPhone 5
page.viewportSize = {
     width: 640,
     height: 1136
};
page.settings.userAgent = 'Mozilla/5.0 (iPad; CPU OS 4_3_5
like Mac OS X; en-us) AppleWebKit/533.17.9 (KHTML, like Gecko)
Version/5.0.2 Mobile/8L1 Safari/6533.18.5';
page.zoomFactor = 1;
page.open('http://localhost:8080/', function (status) {
});
```

如果想验证这个页面是否按照预期来渲染和加载了适当的内容，我们可以用几种方式实现这个目的。

- 把渲染的页面截图，然后肉眼查看这个页面是不是正确渲染了。下面代码段的功能就是打开一个页面：首先检查页面是不是成功打开了，然后使用页面渲染功能截图并保存。

```
page.open('http://localhost:8080/', function(status) {
   if(status == 'success'){
   page.render('./screenshots/iPhone5.png');
}
       });
```

- 以可编程的方式检查已经渲染的页面中的元素和我们期望中的页面的元素是否一致。下面的代码段展示了这一细节：在成功打开页面的基础上，使用page.evaluate获取了一个URI值，它在节点id为description-image的src属性里。假设我们想继续评估iPhone5的用户体验，那么我们就可以检查页面是否从我们控制尺寸的资源目录中下载文件了。

```
page.open('http://localhost:8080/', function (status) {
  if(status == 'success'){
       var image_source = page.evaluate(function(s) {
            return document.querySelector(s).src;
       }, 'description-image');
       if (image_source){
  …
  }
  }
               });
```

- 检查请求了这个页面的网络请求，确保所有需要的资源都正确下载了，并且没有下载额外多余的资源。在下面的代码段中，为了获取HTTP请求的详细信息，我们在请求页面的时候，使用了回调函数。

在这里，我们仍然使用iPhone5作为使用场景。对于每一个请求，我们都会注册一个匿名函数来检查当前请求返回的资源路径，然后检查这个路径是否保留了不合适设备图片的指定路径，比如，这个路径中是否包含目录/nav/320/?

```
page.onResourceRequested = function (request) {
//check request to see if the requested resource is
coming from a known device
// inappropriate directory
     };
page.open(address, function (status) { … });
```

验证Web性能

到目前为止，我们只是将注意力集中在如何验证已渲染的页面中是否已经加载了需要的资源。接下来，我们将注意力放在如何验证页面在每一种专有体验页面的Web性能上。

有几种方式可以实现这个功能。

- 使用Phantom，可以测量请求发出和页面完整渲染之间所消耗的时间。下面的代码段展示了我们在请求页面的时候保存当前时间的快照，当页面加载完以后，我们同样保存一个时间快照，用后一个时间快照的时间减去前一个，就可以算出页面加载的时间。

```
var startTime = Date.now(),
        loadTime;
page.open(address, function (status) {
    if (status == 'success') {
        loadTime = Date.now() - startTime;
        console.log("page load time: " + loadTime + "ms")
    }
});
```

- 使用PhantomJS中的YSlow模块生成YSlow报告。为了可以在命令行中使用YSlow的服务，Yahoo!已经开发出他们自己的PhantomJS JavaScript文件。这个文件就是yslow.js,你可以从*http://yslow.org/phantomjs/*上查看详细信息。使用yslow.js，我们可以传递指定的User Agent和视窗大小。我们同样可以传递数据输出的格式信息和数据的详细信息。图6-3提供了yslow.js简洁帮助文档部分的截图。

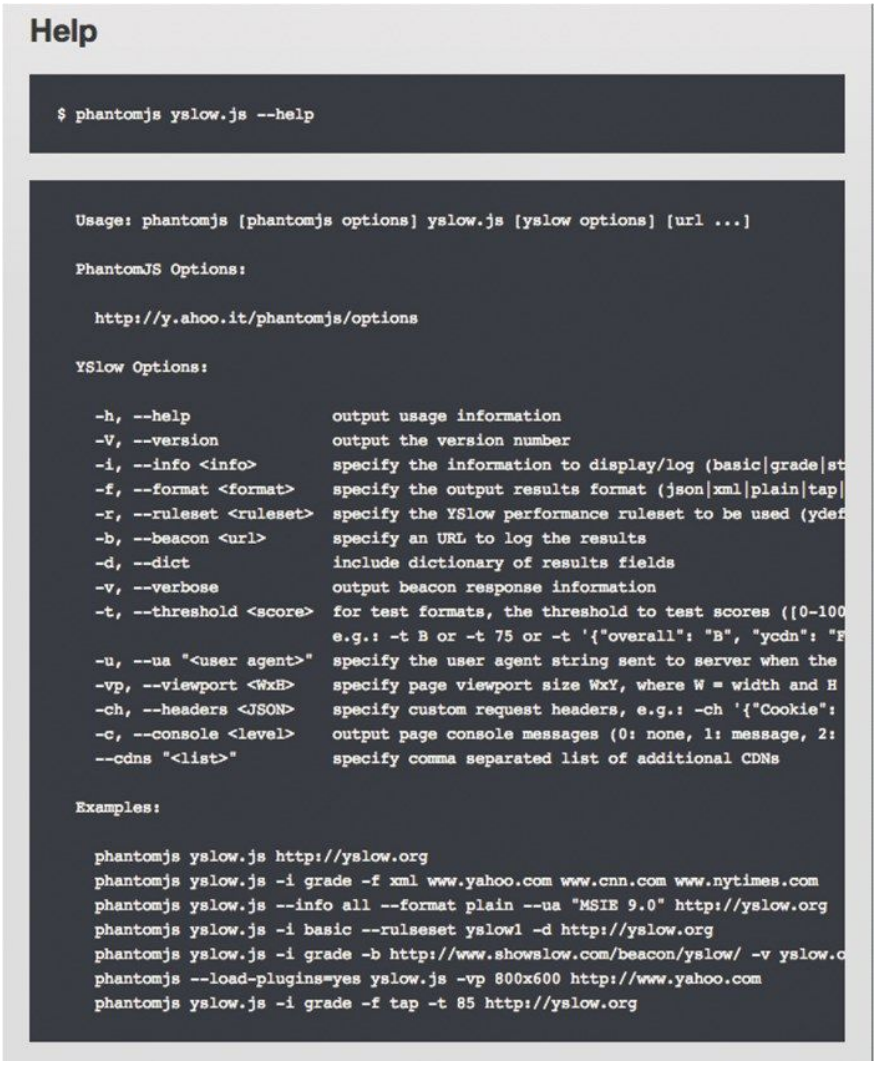

图6-3　YSlow.js的部分文档

图6-3展示了脚本支持的所有参数和一些例子使用方式。通过在命令行输入phantomjs yslow.js -help，你也可以获取同样的帮助文档。

继续我们之前测试iPhnoe体验的例子，现在让我们传递我们自定义的User Agent和视窗的高度和宽度，下面的示例中将展示这一细节。

```
> phantomjs yslow.js --info stats --format plain --vp 640x1136
--ua 'Mozilla/5.0 (iPad; CPU OS 4_3_5 like Mac OS X; en-us)
AppleWebKit/533.17.9 (KHTML, like Gecko) Version/5.0.2 Mobile/
8L1 Safari/6533.18.5' http://localhost:8080
version: 3.1.8
size: 846.4K (846452 bytes)
overall score: B (86)
url: http://localhost:8080/
# of requests: 46
ruleset: ydefault
page load time: 187
page size (primed cache): 10.2K (10290 bytes)
# of requests (primed cache): 1
statistics by component:
    doc:
        # of requests: 1
        size: 10.2K (10290 bytes)
    css:
        # of requests: 8
        size: 154.7K (154775 bytes)
    js:
        # of requests: 20
        size: 617.0K (617056 bytes)
    cssimage:
        # of requests: 6
        size: 32.6K (32694 bytes)
    image:
        # of requests: 10
        size: 14.0K (14095 bytes)
    favicon:
        # of requests: 1
        size: 17.5K (17542 bytes)
statistics by component (primed cache):
    doc:
        # of requests: 1
        size: 10.2K (10290 bytes)
```

注意展示的详细程度：我们获取了页面的整体负载、HTTP的请求数、失败的HTTP请求数和content type带来的总体负载。

也有很多其他框架能做和YSlow.js一样的事情，它们的工作方式和原理也是一样的（例如James Pearce的confess.js，你可以从*http://bit.ly/lofAru5*获取详细信息）。

在这两种用例中，请牢记我们的目的是在所有可罗列的设备上运行所有不同的体验场景。让我们停下来，设想一下，如果我们将刚刚讨论的测试用例集成到你的CI工作流中会怎么样呢？任何破坏了服务层允许范围的改动，你的团队都会收到警告和通知。现在让我们具体来看看如何将这些验证步骤都集成到CI工作流中。

6.3 持续集成

CI是实时代码合并和提交测试的最佳实践。它源自于Kent Beck的极限编程方法论，不过目前在项目组开发人员中，它已经发展为整合代码变动的事实上的实践。它遵循着和Beck提出的另外一个广为人知的实践理论相同的准则——测试驱动开发。把反馈循环离解析者更近（在开发者提交代码的阶段）会节省两者（开发和测试）的时间，并且使反馈循环更加接近流程的下游。

CI的核心工作流就是提交代码，然后完成下面的步骤。

1．确保工作成功构建（确保成功编译，或者压缩静态内容文件，或者为了清除缓存，使用时间戳指纹信息重命名相关资源）；如果没有，那么构建失败。

2．运行集成和单元测试。如果它们失败了，构建失败。一旦构建失败，那么就需要把相关的失败信息汇报给整个团队，那么接下来将有相应的开发人员确定构建失败的原因，并提交相关代码修复这个问题。图6-4展示了整个工作流过程。

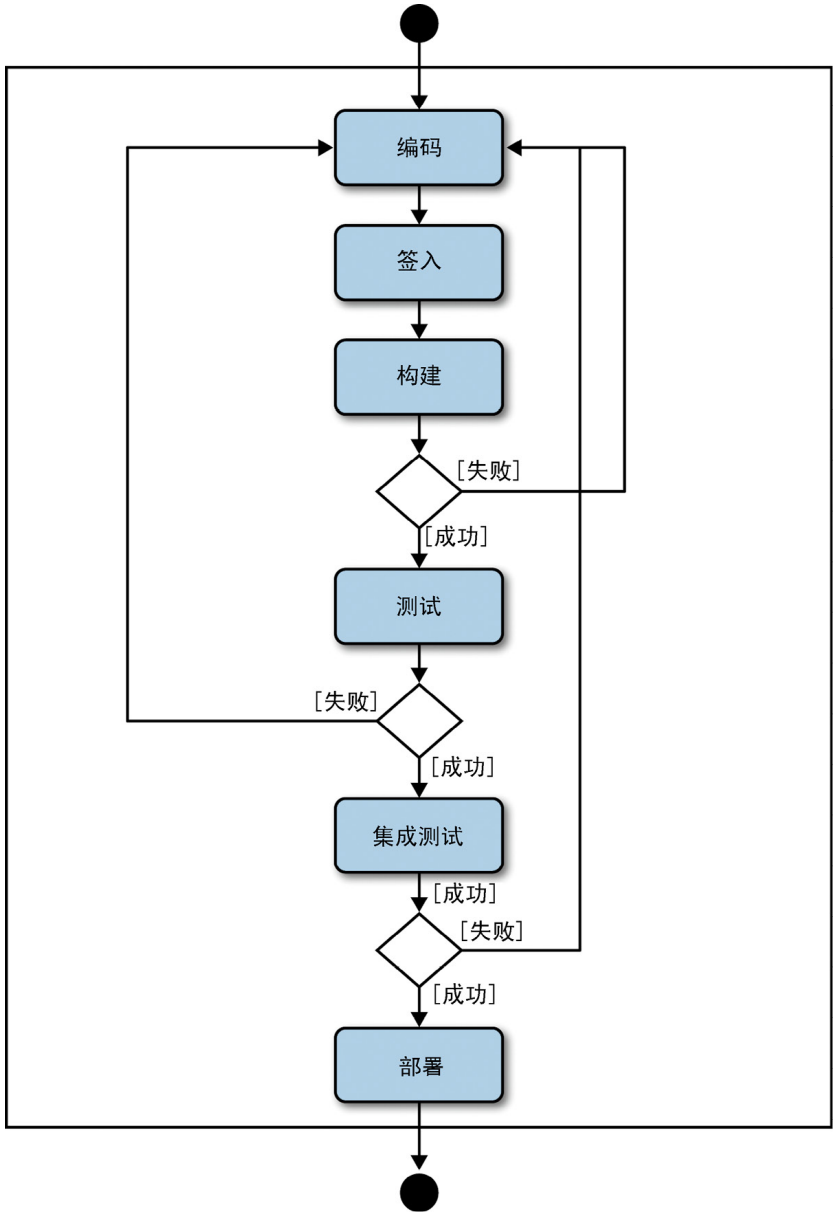

图6-4　CI工作流

现在有很多软件解决方案可以解决图6-4展示的工作流。其中最有名的软件就是Jenkins，它的强大之处在于它是开源的，并且非常容易安装和配置。在本章的后半部分，我们将学习如何将我们的PhantomJS脚本集成到Jenkins的工作流中。但是首先，我们必须将之前所说的内容用脚本运行起来，只有这样，代码在Jenkins中才能顺利实现。

PhantomJS脚本示例

为了把你的性能测试整合到Jenkins中，我们需要做好前期准备工作。首先，你必须创建一个JavaScript脚本来评估你的SLA性能。为了让Jenkins也可以解析相关内容，我们必须以JSunit xml格式输出报告数据。这样，在每一次的构建过程中，Jenkins将运行这个脚本并且产生这个XML文件。在构建结束以后，它将读取这个XML文件作为测试结果输出。

提示

> 在我们创建这个脚本之前，我们需要在工程中增加async.js文件。这样我们就可以以异步的方式运行测试了。当我们试着去获取页面加载时间的时候，我们同时也在下载每个断言的代码，增加了额外的字节数，这些可以增加我们页面加载的时间，从而导致我们获取的测试时间是不准确的。

安装async.js，首先到工程目录下输入：npm install async，这个命令在我们的工程中创建了图6-5所示的文件结构，这样我们的代码就能使用异步模式了。

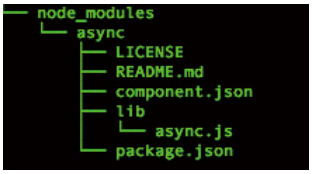

图6-5　异步模块在工作目录中的树形结构图

好，让我们开始吧。首先创建一些将要使用的变量。我们创建一个名叫async的变量，并向这个变量里加载async。接下来，创建一个名叫testsToRun的数组，这个数组里将保存我们将要运行的所有测试用例的名字——例如rendertime和payload。最后，创建一个名叫results的对象，我们用这个对象来保存每个测试的值，包含易于理解的每个测试的显示名称、每个测试的阈值、每个测试最终的真实结果。

为了保证可读性，这个示例中，我们将把一些资源硬编码。在真实的生产脚本中，你应该移除所有硬编码的值，让这些值是可配置的，并且在运行的程序中加载这些值。下面来看下具体的代码。

```
var async = require('async'),
testsToRun = ["rendertime","payload"],
results = {
    testnames:{
        rendertime:"Time to Render",
        payload: "Total Page Payload"
    },
    threshold: {
        rendertime: 500,
        payload: 1000
    },
    actual: {
        rendertime: 0,
        payload:0
    },
    test_results: {
        rendertime: "fail",
        payload: "fail"
    }
}
```

接下来，我们将创建一个名为test的方法，它将封装我们所有将要执行的测试用例。我们将传递一个测试类型和一个回调方法。传递测试类型是因为这样就可以确定运行哪个测试用例。传递回调函数是因为这样一旦测试用例执行完成以后，这个回调函数就可以被调用。在这种方式下，async知道哪个方法被调用完成了。

在这个测试方法中，首先定义一些变量和值。我们将首先获取当前时间的一个快照，定义一个名为page的变量，加载WebPage模块，然后硬编码视窗尺寸和User Agent。再次重申，硬编码的值需要在运行时可以配置，但就本例来说，为了更容易理解，我们在这里对这些值进行了硬编码。

```
function test(testType, callback){
    var startTime = Date.now(),
    loadTime;
    var page = require('webpage').create();
    page.viewportSize = {
         width: 640,
         height: 1136
```

```
    };
    page.settings.userAgent = 'Mozilla/5.0 (iPad; CPU OS
    4_3_5 like Mac OS X; en-us) AppleWebKit/533.17.9 (KHTML, like
    Gecko) Version/5.0.2 Mobile/8L1 Safari/6533.18.5';
    page.zoomFactor = 1;
}
```

接下来，在这个测试方法中，为这个page对象创建onResourceReceived事件处理器。一旦之前请求的远程资源返回的话，这个处理器将会执行。在这个方法中，我们将检查其他测量页面加载的测试是否正确执行（所以我们不需要为这些测试增加延迟）。为了确保功能的正确性，我们需要在result.actual对象中增加payload属性。

```
page.onResourceReceived = function (resp) {
//increment the payload by the size of the resource received
    if(testType == "payload"){
         if(resp.bodySize != undefined){
         results.actual.payload += resp.bodySize
         }
    }
};
```

现在继续这个测试方法。调用page.open来加载用于性能测试的Web页面。不过为了方便起见，我们将硬编码一个本地地址，不过在现实中，你仍然需要保证这个地址在运行时是可配置的。在页面加载完成以后，我们获取当前时间，然后减去开始时间得到最终的渲染时间，然后需要马上调用我们将要为之定义的方法，这个方法名称为calculateResults。最后，我们关闭这个页面并且调用回调函数，通知async我们的函数运行完成了。

```
page.open('http://localhost:8080/', function (status) {
    if(status == 'success'){
         results.actual.rendertime = Date.now() - startTime;
    }
    calculateResults()
    page.close();
    llback.apply();
});
```

在将注意力从这个测试方法转移出去之前，让我们定义一个名为calculateResults的方法。在这个方法中，我们将使用testType作为索引和真正的测试结果阈值进行比较，然后分配一个通过或者失败的评估结果给结果属性。

```
function calculateResults(){
    if(results.actual[testType] <= results.threshold[testType]){
         results.test_results[testType] = "pass";
}
}
```

好，返回到脚本的底部，让我们增加一些控制逻辑，使用async.each，我们调用存储在testsToRun数组中的每一个测试方法的值。当这个方法调用完成以后，之前作为第三个参数传递给asynch.each的匿名方法将会被执行。这个方法调用了一个名为formatOutput的方法，然后退出PhantomJS。这个方法的代码如下所示。

```
async.each(testsToRun,test,
    function(err){
        formatOutput();
        phantom.exit();
    }
);
```

最后，让我们定义这个名为formatOutput的方法。这个方法的主要功能是把测试的结果按照JunitXML的格式来输出，以便于Jenkins解析。我们可以在*http://bit.ly/Ze98o9*找到对应的XSD。

为了完成这个功能，我们可以创建一个套件（suite）保存所有的测试，每个测试创建一个用例，最后运行这个suite。我们为每个测试创建一个测试用例，并且把这个方法映射到testsToRun的数组上。在我们收集完所有的输出结果后，把它们输出到控制台上。

```
function formatOutput(){
   var output = '<?xml version="1.0" encoding="utf-8"?>\n'+
   '<testsuite tests="'+ testsToRun.length +'">\n'
   testsToRun.map(function(t){
        output += '<testcase classname="'+ t +'" name="'+results.testnames[t] +'">\n'
   if(results.test_results[t] == "fail"){
        output += '<failure type="fail"> threshold: '+
        results.threshold[t] + ' result: '+ results.actual[t] +' </failure>\n'
   }
   output += '</testcase>\n'
})
output += '</testsuite>'
console.log(output)
}
```

下面是展示示例的完整代码（你也可以在GitHab上找到，地址为*https://github.com/tomjbarker/HP_ResponsiveDesign*）。

```
//simulating an iPhone 5
var async = require('async'),
       testsToRun = ["rendertime","payload"],
       results = {
           testnames:{
               rendertime:"Time to Render",
               payload: "Total Page Payload"
           },
        threshold: {
               rendertime: 500,
                  payload: 1000
        },
           actual: {
               rendertime: 0,
               payload:0
           },
           test_results: {
               rendertime: "fail",
               payload: "fail"
      }
}
function test(testType, callback){
      var startTime = Date.now(),
      loadTime;
      var page = require('webpage').create();
      page.viewportSize = {
               width: 640,
               height: 1136
      };
      page.settings.userAgent = 'Mozilla/5.0 (iPad; CPU OS
      4_3_5 like Mac OS X; en-us) AppleWebKit/533.17.9 (KHTML, like
      Gecko) Version/5.0.2 Mobile/8L1 Safari/6533.18.5';
      page.zoomFactor = 1;
      page.onResourceReceived = function (resp) {
      //increment the payload by the size of the resource received
      if(testType == "payload"){
               if(resp.bodySize != undefined){
                        results.actual.payload += resp.bodySize
               }
        }
};
page.open('http://localhost:8080/', function (status)
{
      if(status == 'success'){
                results.actual.rendertime = Date.now() - start-
                Time;
      }
```

```
      calculateResults()
      page.close();
      callback.apply();
});
     function calculateResults(){
           var output = "";
           if(results.actual[testType] <= results.threshold[testType]){
                     results.test_results[testType] = "pass";
           }
     }
}
     function formatOutput(){
           var output = '<?xml version="1.0" encoding="utf-8"?>\n'+
               '<testsuite tests="'+ testsToRun.length +'">\n'
           testsToRun.map(function(t){
                    output += '<testcase classname="'+ t +'" name="'+ results.testnames[t] +'">\n'
           if(results.test_results[t] == "fail"){
                     output += '<failure type="fail"> threshold: '+
                     results.threshold[t] + ' result: '+ results.actual[t] +'</failure>\n'
           }
                output += '</testcase>\n'
     })
     output += '</testsuite>'
     console.log(output)
}
    async.each(testsToRun,test,
          function(err){
               formatOutput();
               phantom.exit();
    }
);
```

把这个脚本保存成一个文件，并命名为iphone5test.js，然后从控制台运行。你就可以看到如图6-6所示的日志信息。

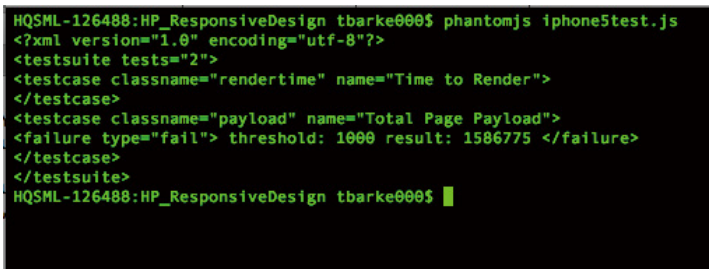

图6-6　控制台运行脚本的输出（注意输出格式为Junit XML）

接下来，我们将安装Jenkins并且在项目的构建过程中运行我们的脚本文件。

Jenkins

Jenkins起源于Hudson，是Kohsuke Kawaguchi还在Sun公司的时候创建的一个开源CI工具。在Oracle收购Sun之后，Jenkins从Hudson脱离。Hudson继续属于Oracle（Oracle已经将这个项目移交给了Eclipse基金会了）。而Jenkins CI则继续由社区提供支持。你可以从*http://jenkins-ci.org/*上获取Jenkins的最新版本，创建自己的拷贝，提交bug或者查阅Jenkins的相关文档。图6-7展示了Jenkins CI的主页。

图6-7　Jenkins CI主页

在Jenkins的主页上，你可以下载安装包进行安装。如图6-8所示，你可以看到Jenkins的Mac OS版本的安装包。

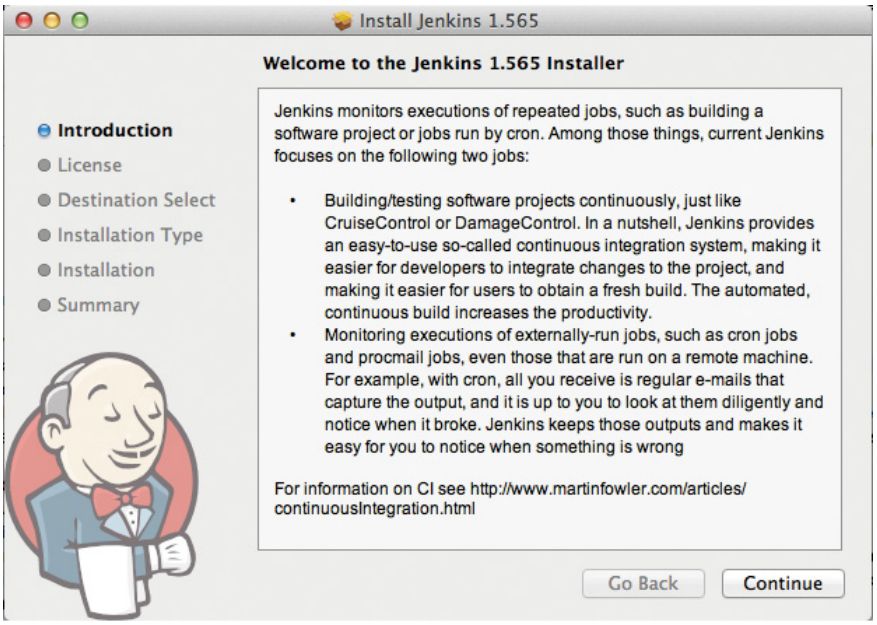

图6-8　Jenkins的Mac OS安装包

完成安装过程之后，可以通过访问*http://localhost:8080/*访问Jenkins，如图6-9所示。

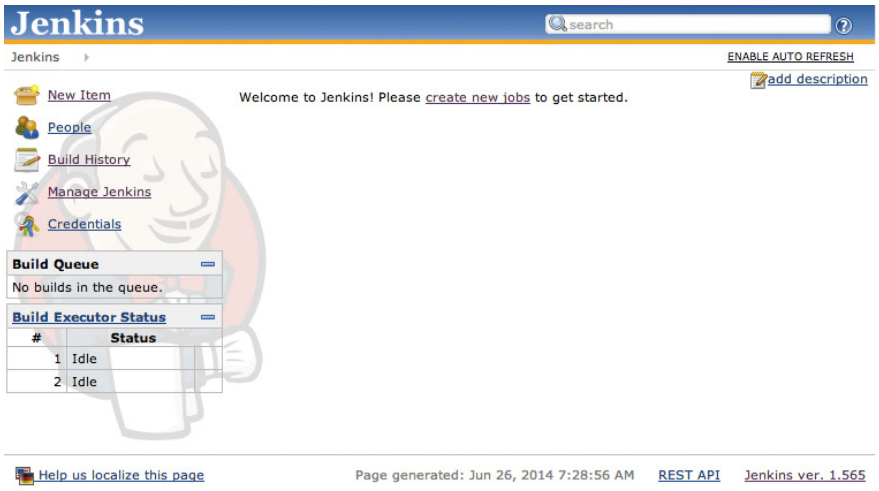

图6-9　刚安装成功的Jenkins的主页

在这个例子中，假设GitHub插件已经安装完成了（如果没有，进入Manage Jenkins，点击Manage Plugins，然后安装），我们使用GitHub作为代码管理工具。

一开始需要在Jenkins中创建一个工程。你可以在Jenkins主页上创建一个新的工程：点击New Item，会打开一个类似于图6-10所示的窗口。对于我们的例子来说，我们创建一个freestyle的工程，然后给这个工程取个好听的名字。

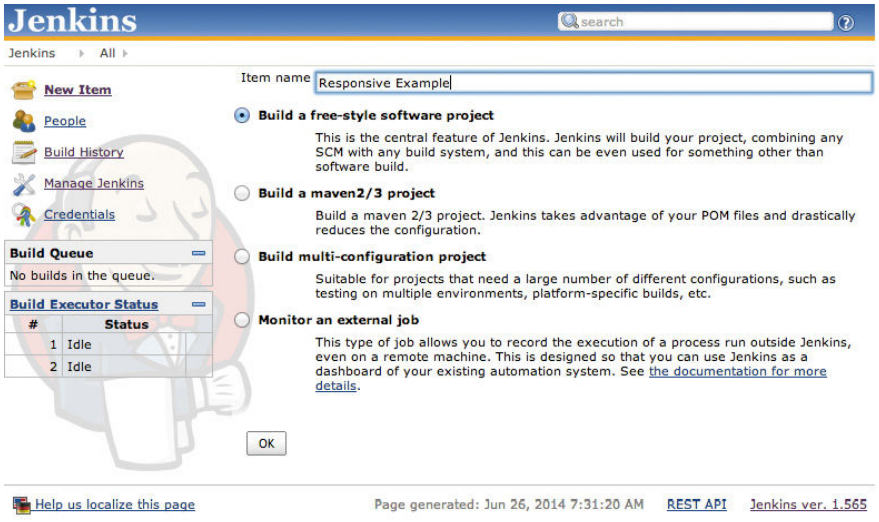

图6-10　在Jenkins中创建一个新工程

现在可以配置这个新的工程了。在Source Code Management页面上，选择Git作为代码管理工具，然后键入我们的工程在GitHub上的地址，如图6-11所示。

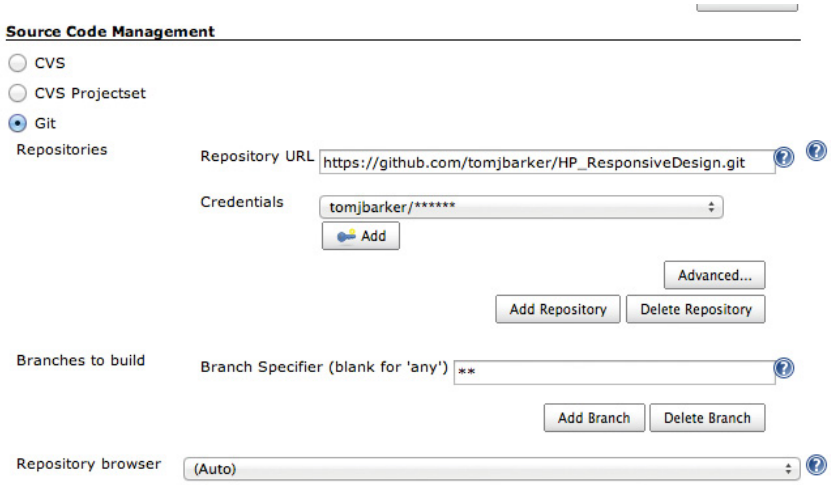

图6-11　指明Jenkins工程和GitHub工程

接下来，为我们的PhantomJS脚本增加一个构建执行步骤，将结果输出到一个名为result.xml的XML文件中（见图6-12）。每次工程构建的时候，这个步骤都会运行我们的脚本，然后生成一个新的XML文件。最后，仍然在Source Code Management页面，增加一个post-build动作，这一步是为了将Junit测试结果以报表形式发布出去，需要明确指定脚本文件生成的result.xml的具体路径。

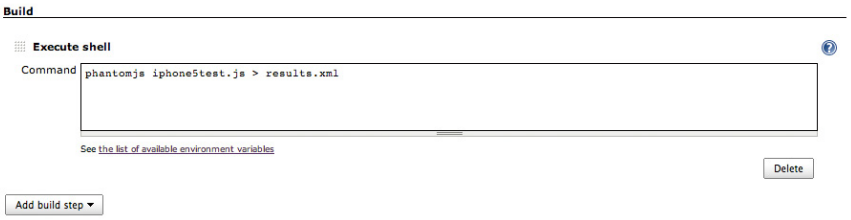

图6-12　在构建过程中运行PhantomJS脚本，然后输出到一个XML文件

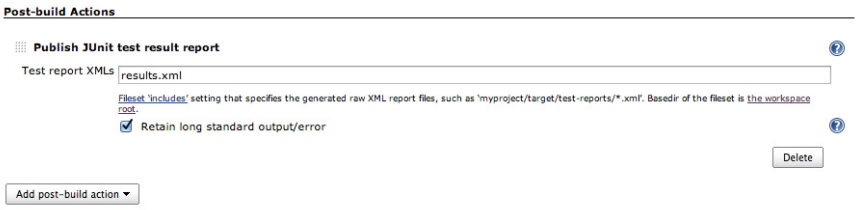

图6-13　在构建过程的post-build步骤中，读取这个生成的XML文件

从这里开始，我们可以在Jenkins上完成构建了。我们的脚本运行完成并且测试报告已经生成。如果希望每一次提交到GitHub上的改动都同步到Jenkins上，也就是每一次提交到GitHub代码之后，Jenkins都需要自动构建一次，我们需要在GitHub上配置一个Web钩子来向Jenkins发送请求。在构建完成之后，可以看到Jenkins上的Web性能测试结果，如图6-14所示。

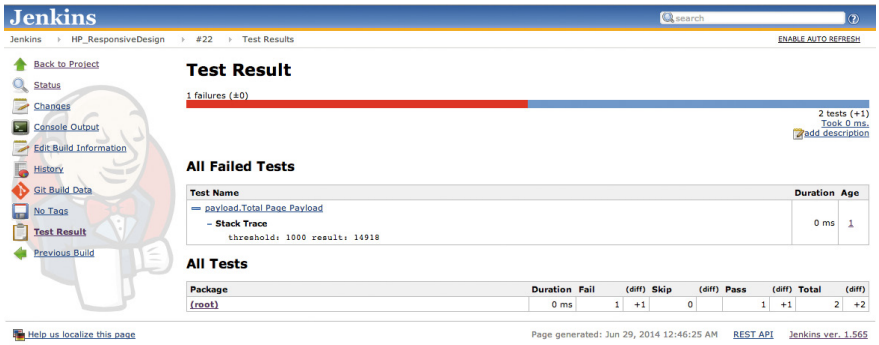

图6-14　Jenkins输出测试结果

到目前为止，我们的代码对Web性能产生的所有影响，都可以被自动检测出来，并且我们可以得到一个实时的反馈了。

6.4 小结

本章对Web性能持续测试这一课题进行了探索。我们讨论了如何使用PhantomJS创建headless browser测试。我们使用大量篇幅讲解了如何验证最佳体验模式。比如在前面的章节中建立的相关知识，例如维护、只加载设备特有的资源、维护页面负载和渲染的SLA时间等。

最后，我们学习了如何使用Jenkins把相关逻辑集成到CI工作流中。

第7章是对解决响应式Web站点问题的一系列框架现状的一次调查。

第7章　响应式设计框架

7.1　响应式设计框架之现状

到目前为止，我们已经探讨了在响应式设计中Web性能模式和反模式，也探讨了如何同时在客户端和服务端使用自己的解决方案来实现这些最佳实践的模式。第6章主要讲述了如何通过PhantomJS创建自动测试以持续验证响应式设计模式性能，以及如何将它们集成进Jenkins的持续集成工作流中。

本章中，我们将讨论当前一些常见的框架，并且学习它们是如何优化Web性能的。这些响应式框架有着不同的实现方式：有的是提供定义好的页面布局的样本文件；有的只是定义了响应式网格布局的网格系统；还有的则提供了一整套完整的解决方案，包括可复用模块、Web字体和JavaScript库的不同页面布局。

如果注意观察这些框架的整体架构和技术，会发现一些很有趣的事情。首先需要注意的就是，它们都是为了前端而生。在这些框架中，一般来说，都是预先定义了CSS，然后渲染某个模块的样式，比如按钮或者网格，或者更加复杂的UI元素，比如折叠模块、滑动模块和导航模块。只要在页面中把这些class样式分配到相应的元素上，我们就可以使用这些模块。而有些框架提供了一些JavaScript API ，通过使用这些API，可以在页面上以编程的方式创建各式风格的元素。

在撰写本书的时候，最有名的框架就是Twitter的BootStrap和ZURB的Foundation。事实上，在Google Trend上比较BootStrap和Foundation与其他框架的相关搜索热度时，我们需要创建两种不同的图表，因为和其他框架比起来，BootStrap的搜索热度整整多出了一个数量级。图7-1和图7-2所示充分说明了这点。

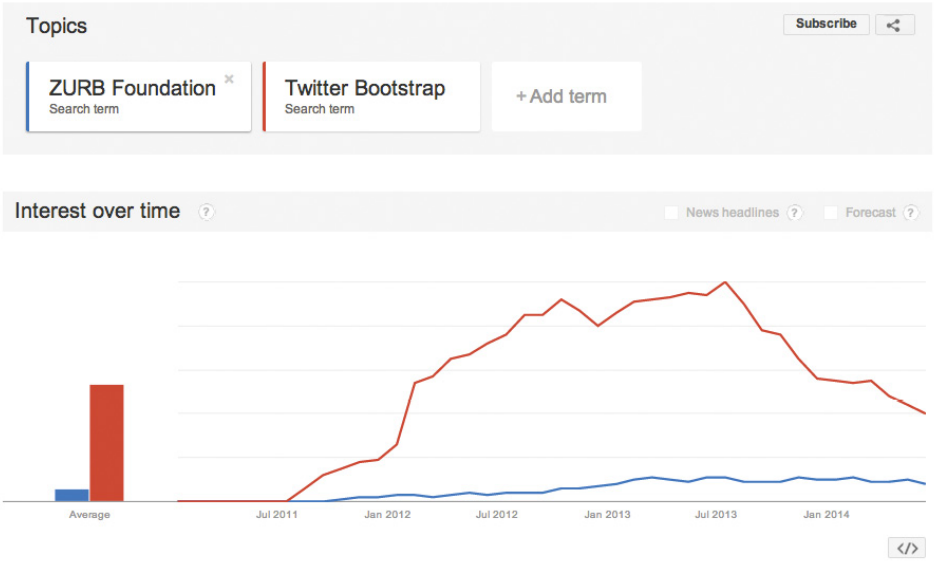

图7-1　在Twitter的BootStrap框架和ZURB的Foundation框架之间的相对搜索热度比较

在图7-1和图7-2中，请注意在这两个Google Trend图片在规模上的不同，在这两张图表中，都含有Foundation的相关连接。下面我们将建立一些标准来评估这些框架。

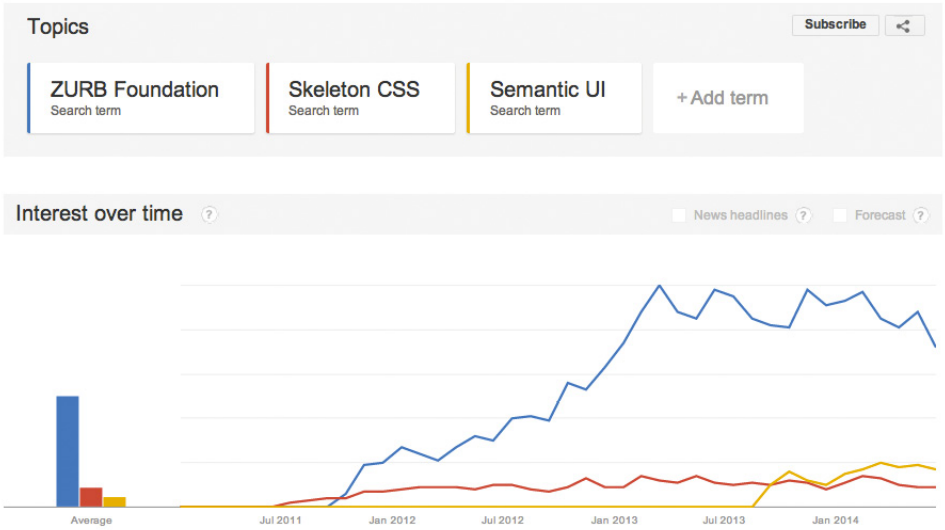

图7-2　在ZURB的Foundation、Skelenton和Semantic UI之间相对搜索热度比较

下面就是我们的评估标准。

- 框架用了哪些模式或者反模式？
- 框架易用性怎么样？
- 框架的大小怎么样？依赖第三方库了吗？
- 如果依赖了，那么框架依赖了什么或者有多少依赖，包括其他框架或者库？

让我们首先来评估Twitter的BootStrap。

7.2 Twitter Bootstrap

Bootstrap是一个前端开源框架，由Twitter的Mark Otto和Jacob Thornton于2011年创建。可以在*http://getbootstrap.com/*上找到Twitter的详细信息。图7-3展示的是BootStrap主页。

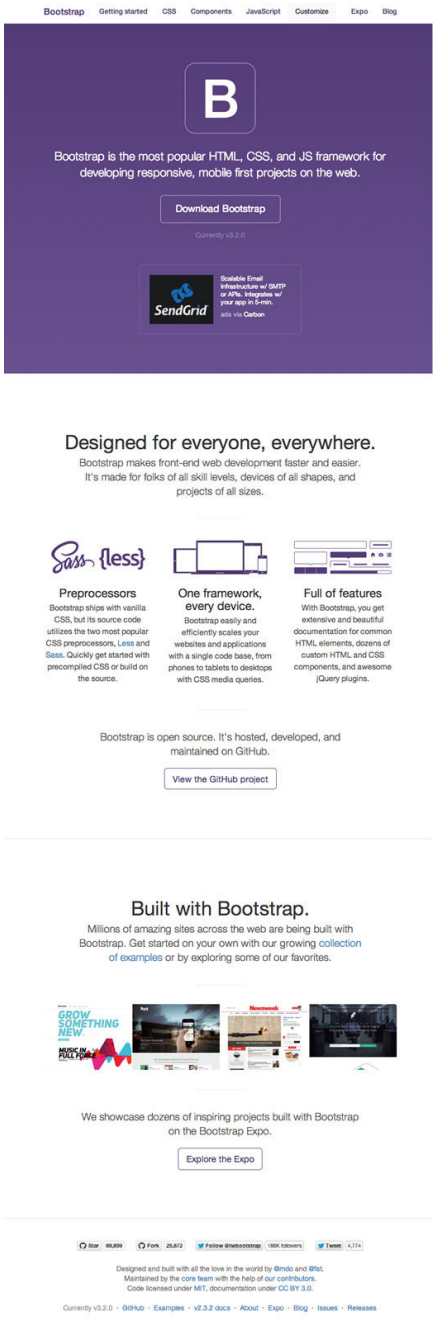

图7-3 BootStrap的主页

BootStrap提供了已定义好的CSS和JavaScript来实现响应式前端组件。这些组件包含了按钮、标签、进度条、网格系统、提示样式和更加特殊的页面布局。

在图7-4中，你可以看到安装目录的组成结构。

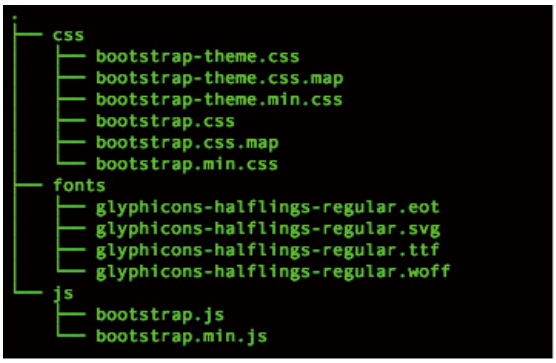

图7-4　BootStrap的核心安装

使用BootStrap非常简单，方式就和你在页面中引入CSS和JavaScript文件一样。引入BootStrap之后，你就可以使用那些预定义的模块了。不过值得一提的是，BootStrap依赖于JQuery。

```
<link href="css/bootstrap.min.css" rel="stylesheet">
<script src="js/bootstrap.min.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>
```

可以很容易看出来为什么BootStrap如此流行：使用BootStrap，在20分钟之内，就可以使用BootStrap预设的组件和样式来创建自己的站点，如图7-5所示（详见*http://tomjbarker.github.io/*）。

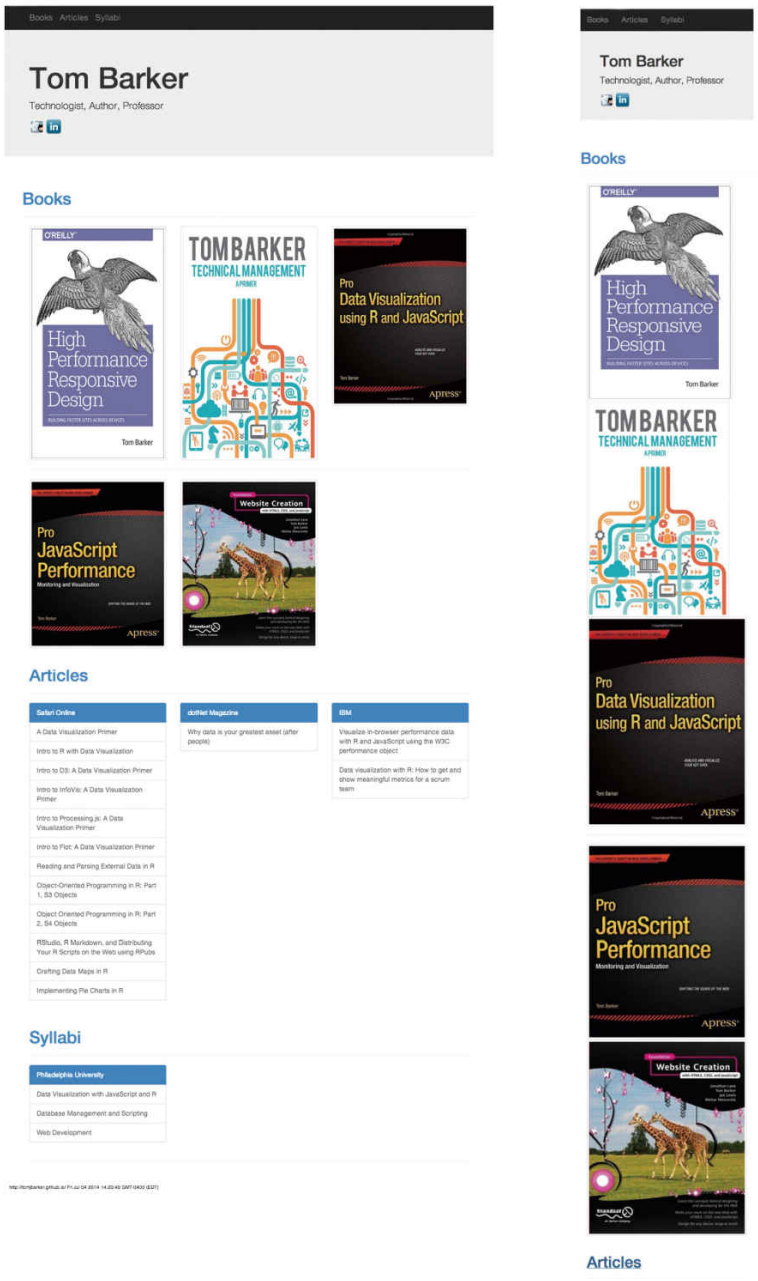

图7-5　使用BootStrap创建的网站

评估

从表7-1中可以看到BootStrap的具体表现。

表7-1　Bootstrap评估一览图

模式/	与众不同的是，Bootstrap为每种体验加载相同的资源。图片将在客户端重新调整大小来适配视窗的大小。可以用一些JQuery插件来解决这些问题。一个比较著名的插件就是HiSRC（详见*https://github.com/teleject/hisrc*），这个插件首先加载更小并且对移动设备更友好的图片，

反模式	然后根据连接的传输速度和客户端的设备像素比，加载相应的大图。虽然这样做可以解决在小屏场景中的问题，小屏中加载的是设备相关的资源，但是大屏中需加载更多的资源
易用性	使用Bootstrap现有的模块和样式，可以在20分钟内构建一个响应式网站
依赖	JQuery
框架大小（包含依赖的大小）	最小安装版本包含Bootstrap的CSS和JavaScript文件，也包含依赖的JQuery文件，所有依赖的文件包括： bootstrap.min.css: 107 KB jquery.min.js: 82.6 KB bootstrap.min.js: 31KB ---------- 总计：220.6 KB 请注意这只是最小的安装版本，如果你想使用Bootstrap的主题和Web字体，需要另外增加

7.3 ZURB Foundation

下一个评估的框架是ZURB的Foundation，ZURB是一个加拿大的设计公司。Foundation其实在2011年就创建和开源了。可以在*http://foundation.zurb.com/*上下载Foundation。图7-6展示了Foundation的主页。

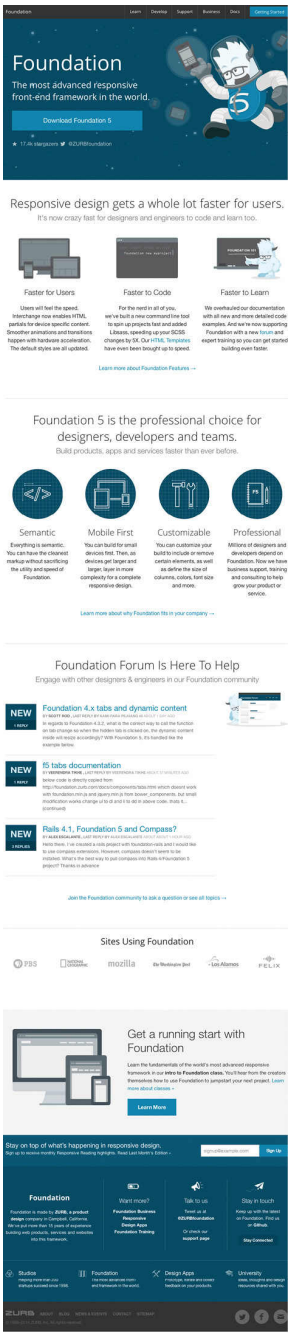

图7-6 ZURB Foundation的主页

下载并解压Foundation的压缩包，解压后框架的目录结构如图7-7所示。

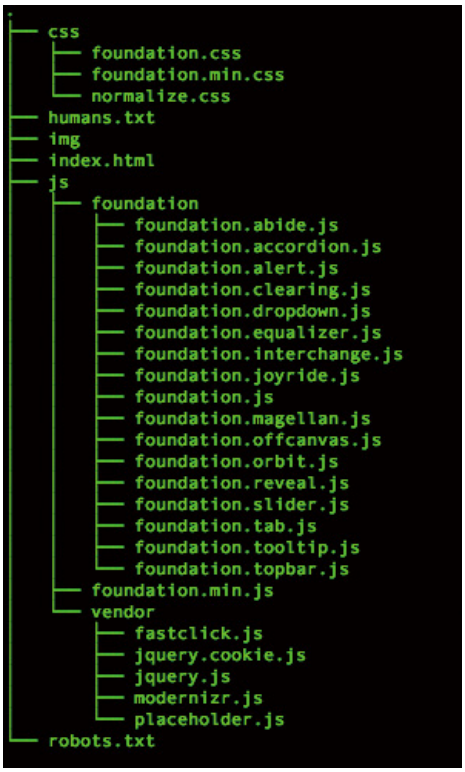

图7-7　Foundation安装目录的树形结构图

和BootStrap一样，Foundation也是由一些预置的组件构成的，包括处理不同视窗尺寸的媒体查询。和BootStrap一样，页面被排列成行与列，将css class作用到 标签上，以指定某一网格结构以及要加载的组件。

通过使用Foundation内置的组件，我建立了一个网站，如图7-8所示。你也可以访问*http://bit.ly/1ORjT1n*查看站点。表7-2提供了具体的评估数据。

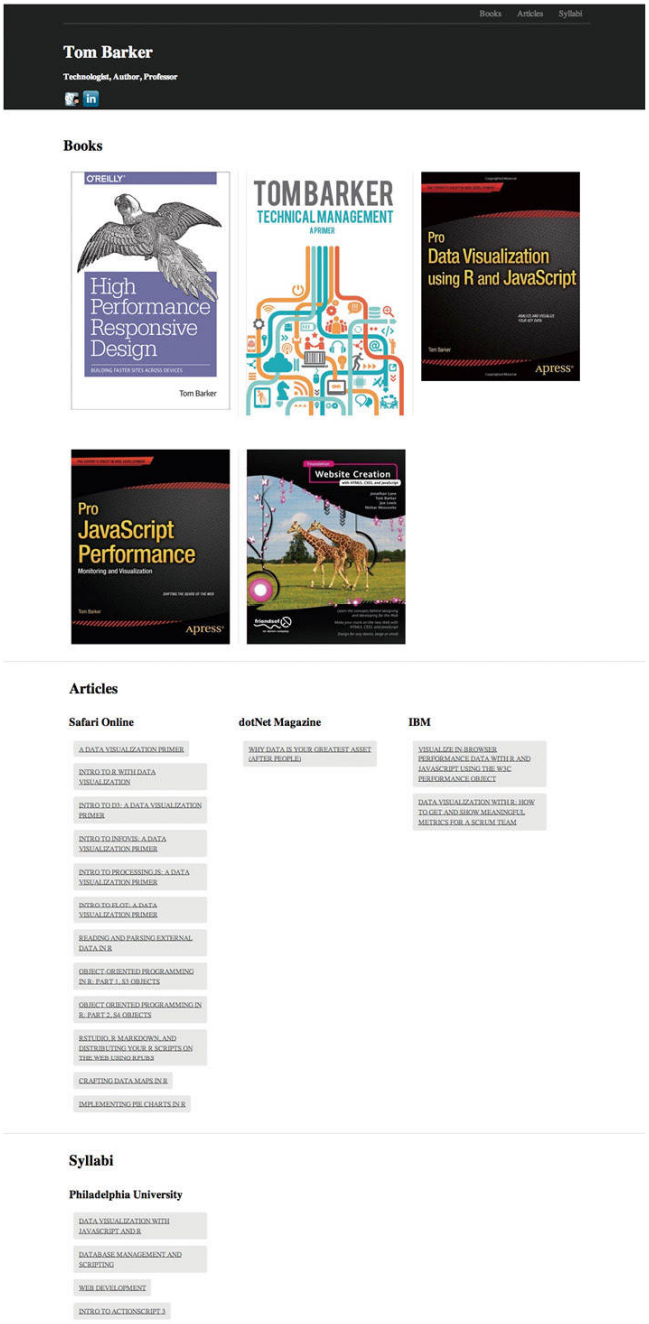

图7-8　使用Foundation创建的网站

让我们看看Foundation的具体表现，如表7-2所示

表7-2　Foundation的评估一览图

模式/反模式	为每一种体验都加载相同的资源，在客户端改变图像尺寸
易用性	和Bootstrap一样，使用预打包模块
依赖	Modernizr, JQuery
框架大小（包含依赖的大小）	foundation.css: 153.6 KB modernizr.js: 11 KB

	jquery.js: 82.6 KB foundation.js: (minified) 89.9 KB ---------- 总计: 337.1 KB

7.4 Skeleton

Skeleton是Twitter的Dave Gamache在2011年创建并发布的框架，可以从*http://www.getskeleton.com/*下载。图7-9展示了Skeleton的主页。

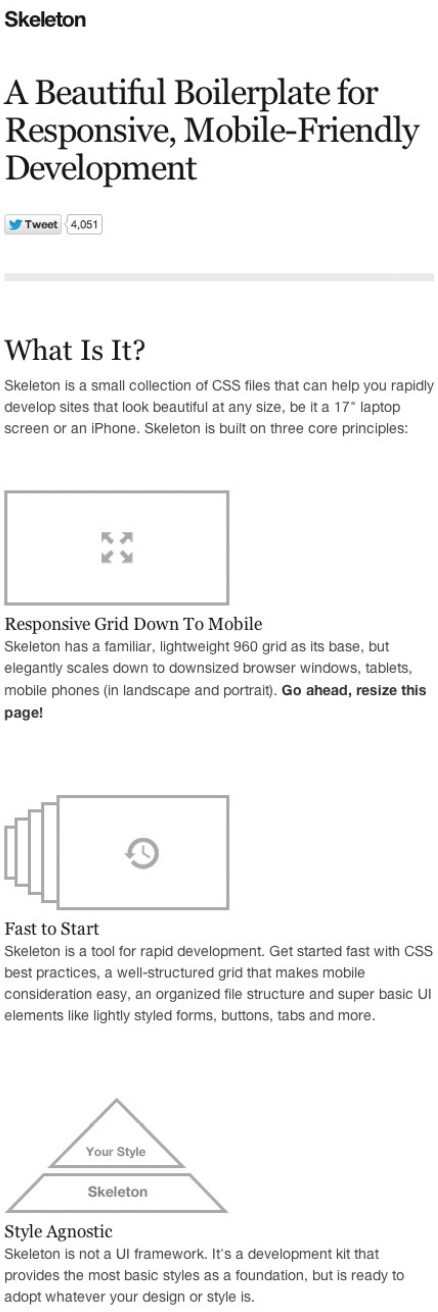

图7-9 Skeleton主页，主页上有相关的操作指南和代码示例

下载并解压Skeleton框架，可以看到Skeleton其实更像是一个模版。它提供了一个index.html供我们编辑，还有一个包含了必要的CSS和图片的目录层次结构，代码可以直接进行引用。图7-10展示了解压后的Skeleton目录树层次结构。

图7-10　Skeleton解压缩以后的样本文件

BootStrap提供了诸如Jumbotron等内置组件，但Skeleton采用了一种更简单的做法。它几乎没有任何样式可言。其主要是提供按钮、表单、排版以及布局定义。使用Skeleton通常是为了降低布局依赖，而在自己的样式中运用这些组件。

使用提供的样板，可以构建一个和前面例子非常类似的网站，这个网站保持了Skeleton的简约风格。如图7-11所示，可以从*http://tomjbarker.github.io/skeleton/*上获取Skeleton。

图7-11　使用Skeleton创建的网站

评估

让我们看看Skeleton在我们评估标准上的表现，详情请见表7-3。

表7-3　Skeleton评估一览图

模式/反模式	Skeleton为所有的设备体验加载相同的资源，这样做的好处是可以保持框架的最小化
易用性	易于使用，预置了样式，不过如果想增加任何样式，则必须自己添加

依赖	无
框架大小（包含依赖的大小）	Skeleton是真正的最小化安装。我们只需要两个CSS文件：base.css和skeleton.css。这些文件没有经过压缩，但是在我的示例代码中，我压缩了它们。在写作本书的时候，它们的总计大小是： base.css: (压缩版) 6.1 K B skeleton.css: (压缩版) 5.4 K B layout.css: 1.7 KB --- 总计: 13.2 KB 请注意这个总数没有将基于Skeleton上的样式包含进来，除非只想要一个最小最简单的设计（一般你不会遇到这种情况），否则都需要增加新的样式

7.5 Semantic UI

Semantic UI是另一个Web框架，同样也是一个前端框架，它提供了预置的客户端响应式UI组件。你可以在*http://semantic-ui.com/*上看到具体信息。图7-12展示了Semantic UI主页的屏幕截图。

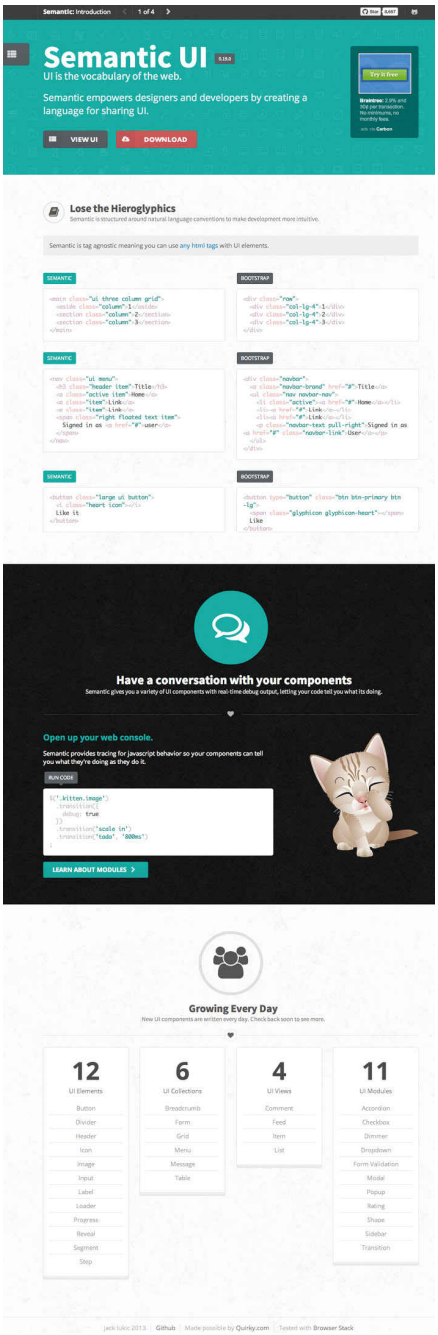

图7-12 Semantic UI主页

可以从主页上下载ZIP文件，解压后的目录结构如图7-13所示，可以看到，在目录中有一个示例文件夹包含了一些示例页面，可以参考这些页面学习如何使用这些框架。名为less的文件夹包括了每个模块各自的LESS文件，minified目录里存放了每个组件已缩小的特有CSS文件。同时，还有个打包目录，包含了所有UI组件与JavaScript API，它们被聚合到了单个CSS和JavaScript文件中。图7-14展示了打包目录的内容。最后，有一个未压缩的目录存放了所有模块各自的（未经压缩过的）CSS文件。

图7-13　Semantic UI目录的树形结构视图

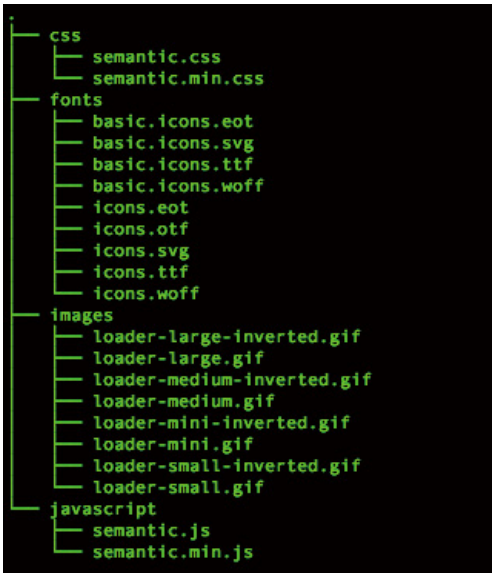

图7-14　Semantic UI的packaged文件目录下的树形结构视图

在图7-14中可以看到，在下载的文件中也包含了各模块的CSS文件，这样就可以只为我们使用的模块加载对应的文件了。

使用打包过的CSS，然后基于下载文件中的homepage.html示例代码，就可以创建一个如图7-15所示的示例网站了。例子可以在*http://tomjbarker.github.io/semantic/* 上找到。

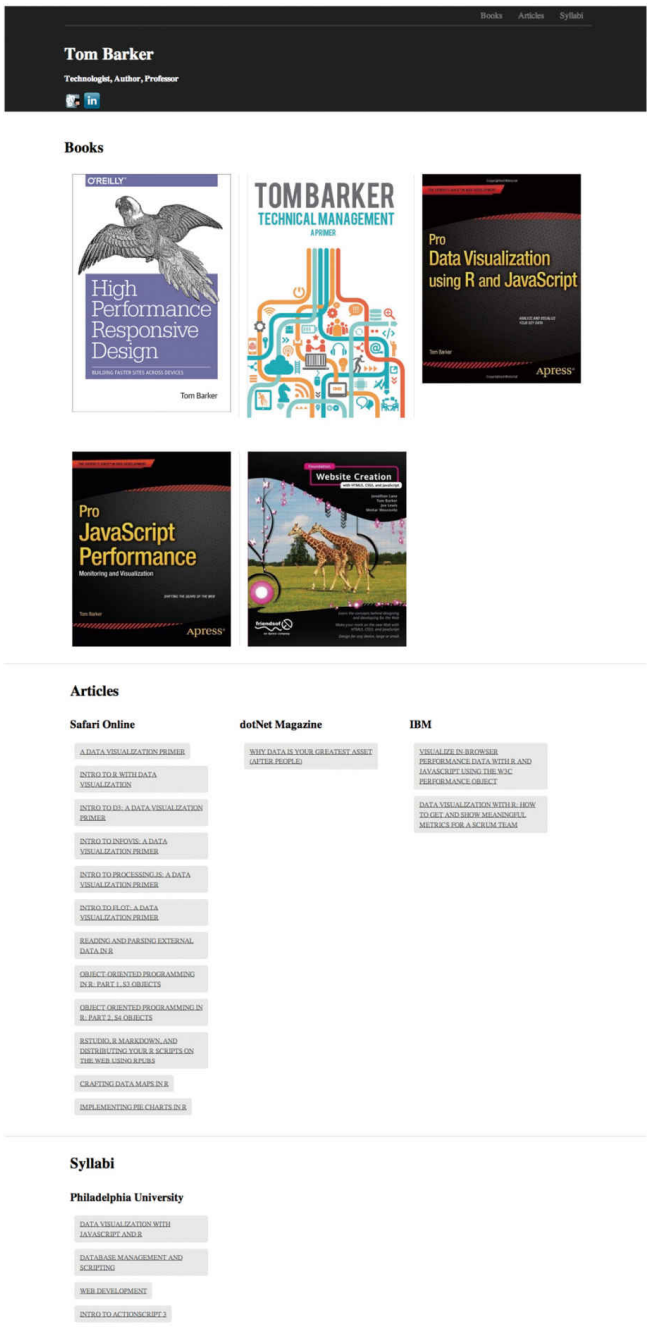

图7-15　使用Semantic UI创建的网站

评估

让我们看看Semantic UI在我们评估标准上的表现，详情请见表7-4。

表7-4 Semantic UI评估一览图

模式/反模式	再次重申，Semantic是一个前端框架，它具有我们已熟知的所有相同的反模式
易用性	和Bootstrap、Foundation相同
依赖	无
框架大小（包含依赖的大小）	semantic.css: (minified) 231 K B jquery.js: 82.6 KB semantic.js: (minified) 134.4 K B ---------- 总计: 448 KB

7.6　各种前端框架之间的比较

如果仔细查看各种数据，就会发现所有这些我们探讨过的框架中，Semantic是最重量级的——如果使用的是打包文件，且没有选择要包含的组件。图7-16提供了各框架大小方面的比较。

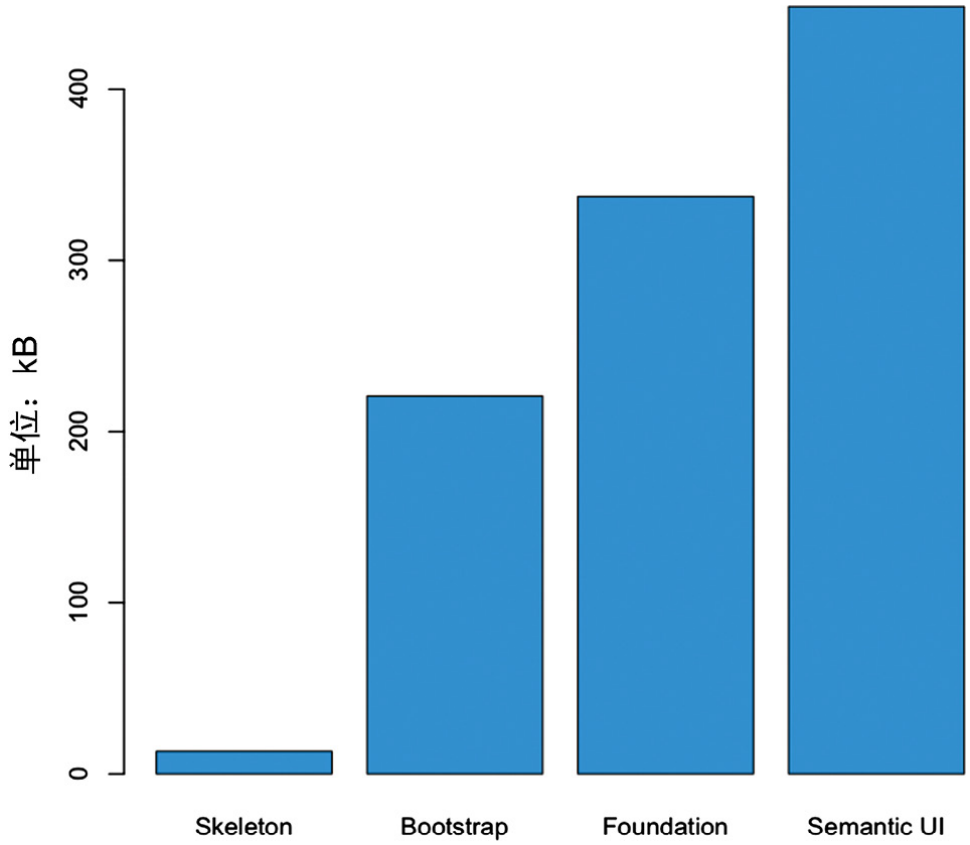

图7-16　框架负载比较

图7-16清晰地指出了框架大小的巨大变化。Skeleton是13KB；Semantic则有448KB之多。更进一步，如果看看使用了这些框架的示例站点，再看看每个站点的总负载，每种资源类型的总负载开始爆发，可以看到，页面大小从Skeleton例子中的460KB，膨胀到Semantic UI示例中的907KB。图7-17展示了这种膨胀。

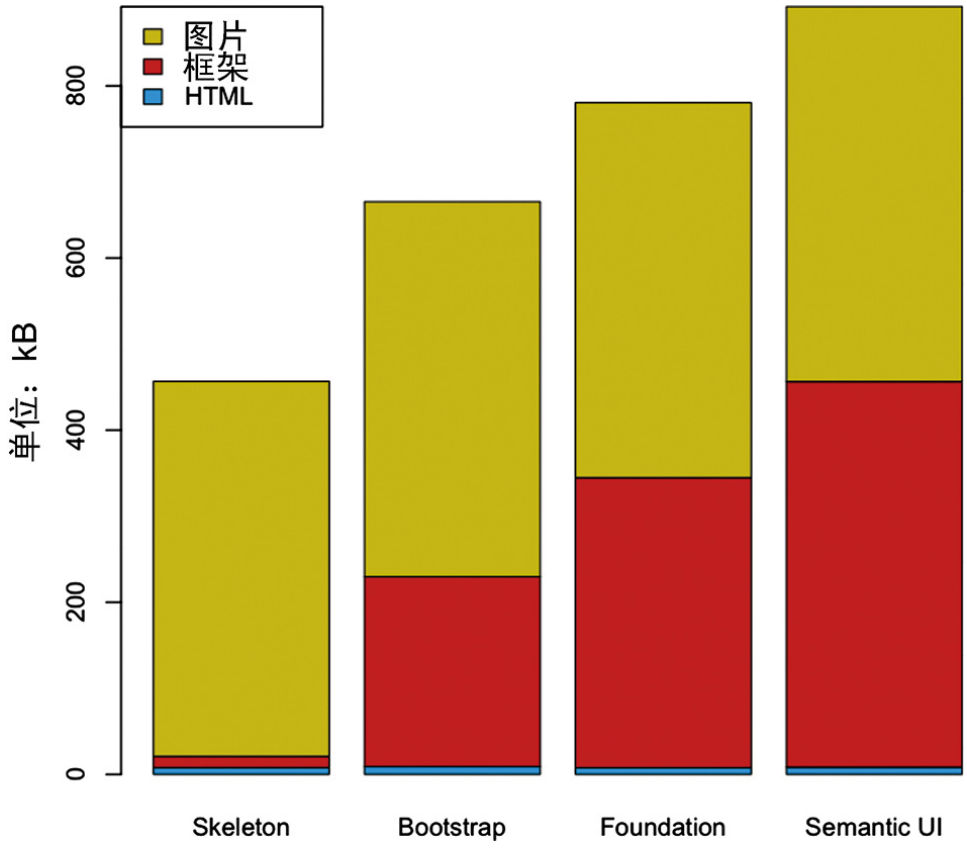

图7-17　框架的大小给页面负载带来的影响

7.7 Ripple

在比较完响应式框架现状之后，它们的相同点和差异点已经显而易见了。是的，它们都是前端框架，除了Skeleton之外，它们最初都不是为了性能而生的。有了这方面的相关知识作为基础，再加上我们之前讨论过的知识点，我决定使用NodeJS建立一个简单的样板，继而创建一个全栈式的响应式网站。我给它起了个名字叫Ripple。更多关于Ripple的信息请参考*https://github.com/tombarker/Ripple*，下面就是Ripple的源代码。

```
var http = require("http");
var url = require("url");
var handle = {}
handle["/"] = checkUA;
handle["/favicon.ico"] = favicon;
var uaViewPortCategories = {
        "320": new RegExp(/Nexus S|iPhone|BB10|Nexus 4|Nexus 5|HTC|LG|GT/),
        "640": new RegExp(/Nexus 7/),
        "1024": new RegExp(/Silk|iPad|Android/)
    };
var assetPath = {
        "css": "assets/css/1024/",
        "img": "assets/img/1024/",
        "js": "assets/js/1024/"
    };
var serv = http.createServer(function (req, res) {
        var pathname = url.parse(req.url).pathname;
        route(pathname, res, req);
    });
function route(path, res, req){
        console.log("routing " + path)
        handle[path](res, req);
    }
function checkUA(res, req) {
        var ua = req.headers["user-agent"]
        var re = new RegExp(/iPhone|iPod|iPad|Mobile|Android/);
        if(re.exec(ua)){
                  getMobileCapabilities(ua, res);
        }
                  renderExperience(res);
    }
function getMobileCapabilities(ua, res){
        res.writeHead(200, { "Content-Type": "text/html" });
        var viewPortWidth = 1024;
        if(uaViewPortCategories["320"].exec(ua)){
                viewPortWidth = 320
        }else if(uaViewPortCategories["640"].exec(ua)){
                viewPortWidth = 640
        }else if(uaViewPortCategories["1024"].exec(ua)){
                viewPortWidth = 1024
     }
assetPath.css = "assets/css/"+viewPortWidth+"/";
assetPath.img = "assets/img/"+viewPortWidth+"/"
assetPath.js = "assets/js/"+viewPortWidth+"/"
}
function renderExperience(res){
       res.writeHead(200, { "Content-Type": "text/html" });
       res.write(assetPath.css + "<br/>");
       res.write(assetPath.img + "<br/>");
       res.end(assetPath.js);
    }
function favicon(res, req){
```

```
        res.writeHead(200, {
                'Content-Type': 'image/x-icon'
    } );
        res.end();
}
serv.listen(80);
```

如果你想使用这些样板文件，非常简单，从GitHub上下载这个工程，放到项目的目录中去，然后在node上运行engine.js，如下所示。

```
node engine.js
```

这个engine文件检查HTTP请求中的User Agent，然后运行一系列的固定表达式来获取User Agent中的值，之后检查当前客户端的具体类型，并基于具体类型为静态资源创建路径。当然，这些资源都是和客户端设备的视窗尺寸相匹配的。

7.8 小结

在本书的撰写过程中，所有和响应式设计相关的框架都是前端框架。除了Skeleton以外，大多是重量级的框架——甚至有些可称之为臃肿。但是它们都遵循着相同的反模式，为每一种不同的设备体验加载相同的资源。

同样值得一提的是，在本书撰写的过程中，还没有主流的服务端框架或者样板文件可用。如果你对上述相关的知识感兴趣，并且有一探究竟的好奇心，我建议你学习Ripple相关知识，探索在服务端使用响应式而带来的巨大性能优势。

看完了

如果您对本书内容有疑问，可发邮件至contact@epubit.com.cn，会有编辑或作译者协助答疑。也可访问异步社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@epubit.com.cn。

在这里可以找到我们：

- 微博：@人邮异步社区
- QQ群：368449889
