

TURING 图灵程序设计丛书 Web开发系列



Pro HTML5 Programming

Powerful APIs for Richer Internet Application Development

# HTML5

# 高级程序设计

[荷] Peter Lubbers

Brian Albers 著

[美] Frank Salim

李杰 柳靖 刘淼 译

- 引领下一代Web开发潮流
- 全面涵盖开创性HTML5 API
- 真实示例，轻松上手

 人民邮电出版社  
POSTS & TELECOM PRESS

大家网

www.TopSage.com

TURING 图灵程序设计丛书 Web开发系列

大家网  
TopSage.com

Pro HTML5 Programming  
Powerful APIs for Richer Internet Application Development

# HTML5 高级程序设计

人民邮电出版社  
北京

大家网 [www.TopSage.com](http://www.TopSage.com)



# 目 录

第 1 章 HTML5 概述.....	1	2.1.2 canvas 是什么 .....	23
1.1 HTML5 发展史 .....	1	2.1.3 canvas 坐标 .....	23
1.2 关于 2022 年的那个神话.....	2	2.1.4 什么情况下不用 canvas.....	24
1.3 谁在开发 HTML5 .....	3	2.1.5 替代内容 .....	24
1.4 新的认识 .....	3	2.1.6 CSS 和 canvas .....	24
1.4.1 兼容性和存在即合理.....	3	2.1.7 浏览器对 HTML5 Canvas 的 支持 .....	25
1.4.2 效率和用户优先 .....	4	2.2 使用 HTML5 Canvas API .....	25
1.4.3 化繁为简 .....	4	2.2.1 检测浏览器支持情况.....	25
1.4.4 通用访问 .....	5	2.2.2 在页面中加入 canvas.....	26
1.5 无插件范式 .....	5	2.2.3 变换 .....	28
1.6 HTML5 的新功能 .....	8	2.2.4 路径 .....	30
1.6.1 新的 DOCTYPE 和字符集.....	8	2.2.5 描边样式 .....	32
1.6.2 新元素和旧元素 .....	9	2.2.6 填充样式 .....	33
1.6.3 语义化标记 .....	9	2.2.7 填充矩形区域 .....	34
1.6.4 使用 Selectors API 简化选取 操作 .....	15	2.2.8 绘制曲线 .....	35
1.6.5 JavaScript 日志和调试 .....	18	2.2.9 在 canvas 中插入图片.....	36
1.6.6 window.JSON .....	19	2.2.10 渐变 .....	37
1.6.7 DOM Level 3 .....	19	2.2.11 背景图 .....	39
1.6.8 Monkeys、Squirrelfish 和其他 JavaScript 引擎 .....	19	2.2.12 缩放 canvas 对象.....	41
1.7 小结 .....	20	2.2.13 Canvas 变换 .....	42
第 2 章 Canvas API .....	22	2.2.14 Canvas 文本 .....	43
2.1 HTML5 Canvas 概述 .....	22	2.2.15 应用阴影 .....	45
2.1.1 历史 .....	22	2.2.16 像素数据 .....	46
		2.2.17 Canvas 的安全机制 .....	48
		2.3 使用 HTML5 Canvas 创建应用.....	48

2.4 小结	52	4.5.2 处理 Geolocation 数据	85
<b>第 3 章 音频和视频</b>	<b>53</b>	4.5.3 最终代码	88
3.1 HTML5 Audio 和 Video 概述	53	4.6 进阶功能	90
3.1.1 视频容器	53	4.6.1 现在的状态是什么	91
3.1.2 音频和视频编解码器	54	4.6.2 在 Google Map 上显示“我在 这里”	92
3.1.3 HTML5 Audio 和 Video 的限制	55	4.7 小结	93
3.1.4 audio 元素和 video 元素的 浏览器支持情况	56	<b>第 5 章 Communication API</b>	<b>94</b>
3.2 使用 HTML5 Audio 和 Video API	56	5.1 跨文档消息通信	94
3.2.1 浏览器支持性检测	57	5.1.1 理解源安全	96
3.2.2 理解媒体元素	58	5.1.2 跨文档消息通信的浏览器支持 情况	97
3.2.3 使用 audio 元素	62	5.1.3 使用 postMessage API	97
3.2.4 使用 video 元素	63	5.1.4 使用 postMessage API 创建 应用	98
3.2.5 进阶功能	68	5.2 XMLHttpRequest Level 2	103
3.3 小结	70	5.2.1 跨源 XMLHttpRequest	104
<b>第 4 章 Geolocation API</b>	<b>72</b>	5.2.2 进度事件	105
4.1 位置信息	72	5.2.3 XMLHttpRequest Level 2 的 浏览器支持情况	106
4.1.1 纬度和经度坐标	73	5.2.4 使用 XMLHttpRequest API	106
4.1.2 位置信息从何而来	73	5.2.5 创建 XMLHttpRequest 应用	107
4.1.3 IP 地址地理定位数据	74	5.3 进阶功能	111
4.1.4 GPS 地理定位数据	74	5.3.1 结构化的数据	111
4.1.5 Wi-Fi 地理定位数据	74	5.3.2 Framebusting	111
4.1.6 手机地理定位数据	75	5.4 小结	112
4.1.7 用户自定义的地理定位数据	75	<b>第 6 章 WebSockets API</b>	<b>113</b>
4.2 HTML5 Geolocation 的浏览器支持 情况	75	6.1 HTML5 WebSockets 概述	113
4.3 隐私	76	6.1.1 实时和 HTTP	113
4.3.1 触发隐私保护机制	77	6.1.2 解读 HTML5 WebSockets	115
4.3.2 处理位置信息	78	6.2 HTML5 WebSockets 的浏览器支持 情况	120
4.4 使用 HTML5 Geolocation API	78	6.3 编写简单的 Echo WebSocket 服务器	120
4.4.1 浏览器支持性检查	78	6.4 使用 HTML5 WebSockets API	126
4.4.2 位置请求	79	6.4.1 浏览器支持情况检测	126
4.5 使用 HTML5 Geolocation 构建实时 应用	83		
4.5.1 编写 HTML 显示代码	84		



6.4.2	API 的基本用法	127	8.3.5	示例代码	163
6.5	创建 HTML5 WebSockets 应用程序	130	8.4	构建 HTML5 Web Workers 应用	164
6.5.1	编写 HTML 文件	131	8.4.1	编写 blur.js 辅助脚本	165
6.5.2	添加 WebSocket 代码	133	8.4.2	编写 blur.html 应用页面	166
6.5.3	添加 Geolocation 代码	133	8.4.3	编写 blurWorker.js	167
6.5.4	合并所有内容	134	8.4.4	与 Web Worker 通信	168
6.5.5	最终代码	136	8.4.5	运行程序	169
6.6	小结	138	8.4.6	示例代码	170
<b>第 7 章</b>	<b>Forms API</b>	<b>139</b>	8.5	小结	174
7.1	HTML5 Forms 概述	139	<b>第 9 章</b>	<b>Web Storage API</b>	<b>175</b>
7.1.1	HTML Forms 与 XForms	139	9.1	HTML5 Web Storage 概述	175
7.1.2	功能性表单	140	9.2	HTML5 Web Storage 的浏览器支持情况	176
7.1.3	HTML5 Forms 的浏览器支持情况	140	9.3	使用 HTML5 Web Storage API	176
7.1.4	输入型控件目录	141	9.3.1	检查浏览器的支持性	176
7.2	使用 HTML5 Forms API	144	9.3.2	设置和获取数据	178
7.2.1	新的表单特性和函数	144	9.3.3	封堵数据泄漏	179
7.2.2	表单验证	147	9.3.4	localStorage 与 session-Storage	180
7.2.3	验证反馈	151	9.3.5	Web Storage API 的其他特性和函数	180
7.3	构建 HTML5 Forms 应用	152	9.3.6	更新 Web Storage 后的通信	182
7.4	小结	157	9.3.7	探索 Web Storage	183
<b>第 8 章</b>	<b>Web Workers API</b>	<b>158</b>	9.4	构建 HTML5 Web Storage 应用	184
8.1	HTML5 Web Workers 的浏览器支持情况	158	9.5	浏览器数据库存储展望	195
8.2	使用 HTML5 Web Workers API	159	9.6	进阶功能	197
8.2.1	浏览器支持性检查	159	9.6.1	JSON 对象的存储	198
8.2.2	创建 HTML5 Web Workers	160	9.6.2	共享窗口	198
8.2.3	多个 JavaScript 文件的加载与执行	160	9.7	小结	200
8.2.4	与 HTML5 Web Workers 通信	160	<b>第 10 章</b>	<b>构建离线 Web 应用</b>	<b>201</b>
8.3	编写主页	161	10.1	HTML5 离线 Web 应用概述	201
8.3.1	处理错误	161	10.2	使用 HTML5 离线 Web 应用 API	203
8.3.2	HTML5 Web Workers	162	10.2.1	检查浏览器的支持情况	203
8.3.3	HTML5 Web Workers 的嵌套使用	162	10.2.2	搭建简单的离线应用程序	203
8.3.4	使用定时器	163	10.2.3	支持离线行为	204

10.2.4	manifest 文件	204
10.2.5	applicationCache API	205
10.3	使用 HTML5 离线 Web 应用构建应用	206
10.3.1	创建记录资源的 manifest 文件	208
10.3.2	创建构成界面的 HTML 和 CSS	208
10.3.3	创建离线 JavaScript	209
10.3.4	检查 applicationCache 的支持情况	210
10.3.5	为 Update 按钮添加处理函数	211
10.3.6	添加 Geolocation 跟踪代码	211
10.3.7	添加 Storage 功能代码	212
10.3.8	添加离线事件处理程序	212
10.4	小结	213
第 11 章	HTML5 未来展望	214
11.1	HTML5 的浏览器支持情况	214
11.2	HTML 未来的发展	214
11.2.1	WebGL	215
11.2.2	设备	217
11.2.3	音频数据 API	217
11.2.4	视频元素改进	218
11.2.5	触摸屏设备事件	218
11.2.6	P2P 网络	220
11.2.7	最终方向	221
11.3	小结	221



# HTML5 概述

**这**是一本关于 HTML5 编程的书。不过在学习之前，有必要先了解一下背景知识，什么是 HTML5？它经历了怎样的发展历程？HTML4 和 HTML5 有什么区别？

本章中，我们会集中讨论大家关注的一些实际问题。为什么是 HTML5？为什么它能掀起风潮？是什么设计理念使得 HTML5 真正具有革命性的进步？HTML5 如何在大幅改动的同时保持高度兼容？无插件范式意味着什么？HTML5 包含什么，不包含什么？HTML5 新增加了哪些特性，为什么能揭开整个 Web 开发新时代的序幕？下面我们一起来了解一下。

## 1.1 HTML5 发展史

HTML 的历史可以追溯到很久以前。1993 年 HTML 首次以因特网草案的形式发布。20 世纪 90 年代的人见证了 HTML 的大幅发展，从 2.0 版，到 3.2 版和 4.0 版（一年出了两个版本！），再到 1999 年的 4.01 版。随着 HTML 的发展，W3C（万维网联盟）掌握了对 HTML 规范的控制权。

然而，在快速发布了这四个版本之后，业界普遍认为 HTML 已经到了穷途末路，对 Web 标准的焦点也开始转移到了 XML 和 XHTML 上，HTML 被放在了次要位置。不过在此期间，HTML 体现了顽强的生命力，主要的网站内容还是基于 HTML 的。为能支持新的 Web 应用，同时克服现有的缺点，HTML 迫切需要添加新功能，制定新规范。

致力于将 Web 平台提升到一个新的高度，一小组人在 2004 年成立了 WHATWG（Web Hypertext Application Technology Working Group，Web 超文本应用技术工作组）。他们创立了 HTML5 规范，同时开始专门针对 Web 应用开发新功能——这被 WHATWG 认为是 HTML 中最薄弱的环节。Web 2.0 这个新词也就正是在那个时候被发明的。Web2.0 实至名归，开创了 Web 的第二个时代。旧的静态网站逐渐让位于需要更多特性的动态网站和社交网站——这其中的新功能真的是数不胜数。

2006 年，W3C 又重新介入 HTML，并于 2008 年发布了 HTML5 的工作草案。2009 年，XHTML2 工作组停止工作。又过了一年，也就到了现在。因为 HTML5 能解决非常实际的问题（随后可以看到），所以在规范还未定稿的情况下，各大浏览器厂家就已经按耐不住了，开始对旗下产品进行升级以支持 HTML5 的新功能。这样，得益于浏览器的实验性反馈，HTML5 规范也得到了持续地完善，HTML5 以这种方式迅速融入到了对 Web 平台的实质性改进中。



“大家好，我是 Brian<sup>①</sup>，HTML 的铁杆老粉丝。

1995 年，我创建了第一个属于自己的个人主页。那时候的‘主页’就是用来介绍自己的。上面的照片通常不清晰，代码中用了很多<blink>标签，页面上会告诉大家我住在哪儿、读过什么书、正在做什么跟计算机相关的工作。我和我的那些所谓‘万维网开发者’不是在大学里读书就是在大学里工作。

那时候的 HTML 非常初级，没有任何工具可用。Web 应用几乎没有，顶多有少量的文本处理脚本。页面代码都是用大家各自喜欢的文本编辑器写出来的。页面的更新频率基本上是数周或者数月。

不知不觉，我们已经走过了漫长的 15 个年头。

今天，用户对其在线资料一天更新很多次已经是很平常的事了。当然，如果没有在线工具持续稳定的更新换代，也不会有今天这样的交互方式。

提醒各位读者，大家在看这本书的时候心里要明白，我们的示例虽然现在看起来非常简单，但潜力是巨大的。就像 20 世纪 90 年代中期那些率先使用<img>标签的人一样，他们又怎么会知道在十年以后，很多人都已经在线编辑和储存照片了；而我们要有这样的前瞻性。

我们希望书中示例的基本思路能够激发读者无穷的创意，从而为 Web 的下个十年奠定新的基础。”

——Brian

## 1.2 关于 2022 年的那个神话

今天，我们看到的 HTML5 规范已经以工作草案的形式发布了——还不是最终版。那什么时候 HTML5 规范才能尘埃落定呢？现在就来了解一下几个关键时间点。第一个时间点是 2012 年，目标是发布候选推荐版。第二个时间点是 2022 年，目标是发布计划推荐版。哦！那等着吧，还早着呢！可能大家会这么想，然后就把书合上，扔到一边，等十年后再说。那就大错特错了，在明白这两个时间点的真正意义之前，可别急着下结论。

第一个，也就是最近的 2012 年，可以说是最重要的时间点，因为这个时间点一到就意味着 HTML5 规范编写完成了。想象一下，这并不久远，也就两年后的事情。计划推荐版（普遍认为距今还有点远）的重要性在于届时将会有两个对 HTML5 的互通实现，意味着将有两个浏览器会完全支持整个 HTML5 规范的所有功能——这个远大的目标让 2022 年这个时间点看起来又似乎太近了。毕竟，现在连 HTML 4 都还没有实现这个目标呢<sup>②</sup>。

关键是现在浏览器厂家已经着手支持 HTML5 中很多优秀的新功能了。只要用户有需求，现

<sup>①</sup> Brian，本书作者之一。——译者注

<sup>②</sup> HTML 4 最早于 1997 年成为 W3C 推荐标准，到现在 10 多年早已经过去了，仍然不存在两个完全支持这一规范的浏览器。——编者注



在就可以利用这些新功能进行 Web 应用的开发。虽然一些细节方面的改造还会持续进行，相应的 Web 应用可能需要改动，不过，相对于使用 HTML5 为用户带来的体验来讲，这点付出不算什么。当然，如果用户的浏览器是 IE6 的话，很多新功能是不支持的，需要模拟——不过这也不能成为抛弃 HTML5 的理由，毕竟这些用户最终都会升级浏览器版本，很多可能会直接选用 IE9，而且微软承诺在 IE9 中持续增加对 HTML5 的支持。实际上，通过使用新的浏览器和改进的模拟技术意味着用户现在和不久的将来便可以使用很多 HTML5 功能了。

### 1.3 谁在开发 HTML5

我们都知道开发 HTML5 需要成立相应的组织，并且肯定需要有人来负责。这正是下面这三个重要组织的工作。

- WHATWG：由来自 Apple、Mozilla、Google、Opera 等浏览器厂商的人组成，成立于 2004 年。WHATWG 开发 HTML 和 Web 应用 API，同时为各浏览器厂商以及其他有意向的组织提供开放式合作。
- W3C：W3C 下辖的 HTML 工作组目前负责发布 HTML5 规范。
- IETF (Internet Engineering Task Force, 因特网工程任务组)：这个任务组下辖 HTTP 等负责 Internet 协议的团队。HTML5 定义的一种新 API (WebSocket API) 依赖于新的 WebSocket 协议，IETF 工作组正在开发这个协议。

### 1.4 新的认识

HTML5 是基于各种各样的理念（在 WHATWG 规范中有详述）进行设计的，这些设计理念体现了对可能性和可行性的新认识。

- 兼容性
- 实用性
- 互通性
- 通用访问性

#### 1.4.1 兼容性和存在即合理

别担心，HTML5 并不是颠覆性的革新。相反，实际上 HTML5 的一个核心理念就是保持一切新特性平滑过渡。一旦浏览器不支持 HTML5 的某项功能，针对功能的备选行为就会悄悄进行。再说，互联网上有些 HTML 文档已经存在 20 多年了，因此，支持所有现存 HTML 文档是非常重要的。

HTML5 的研究者们还花费了大量的精力来研究通用行为。比如，Google 分析了上百万的页面，从中分析出了 DIV 标签的通用 ID 名称，并且发现其重复量很大。例如，很多开发人员使用 DIV id="header" 来标记页眉区域。HTML5 不就是要解决实际问题吗？那何不直接添加一个 <header> 标签呢？

尽管 HTML5 标准的一些特性非常具有革命性，但是 HTML5 旨在进化而非革命。毕竟没有从头再来的必要。（就算有必要的话，也不应该是 HTML5，起码也要发明一个更好的！）



## 1.4.2 效率和用户优先

HTML5 规范是基于用户优先准则编写的，其宗旨是“用户即上帝”，这意味着在遇到无法解决的冲突时，规范会把用户放到第一位，其次是页面作者，再次是实现者（或浏览器），接着是规范制定者（W3C/WHATWG），最后才考虑理论的纯粹性。因此，HTML5 的绝大部分是实用的，只是有些情况下还不够完美。

看看这个示例，下面的几种代码写法在 HTML5 中都能被识别。

```
id="prohtml5"  
id=prohtml5  
ID="prohtml5"
```

当然，肯定会有人反对这种不严格的语法，我们不去辩论对错，只去关心一个底线，那就是最终用户其实并不在乎代码怎么写。当然，我们并不提倡入门者一开始写代码就这么不严谨，毕竟归根结底，受害者还是最终用户，因为一旦由于开发人员的原因造成页面错误导致不能正常显示，那么被折磨的肯定是最终用户。

HTML5 也衍生出了 XHTML5（可通过 XML 工具生成有效的 HTML5 代码）。HTML 和 XHTML 两种版本的代码经过序列化应该可以生成近乎一样的 DOM 树。显然 XHTML 的验证规则严格得多，刚才示例中后两行代码是无法通过验证的。

### 1. 安全机制的设计

为保证 HTML5 足够安全，HTML5 在设计时就做了大量的工作。规范中的各个部分都有专门针对安全的章节，并且安全是被优先考虑的。HTML5 引入了一种新的基于来源的安全模型，该模型不仅易用，而且对各种不同的 API 都通用。这个安全模型可以让我们做一些以前做不到的事情，不需要借助于任何所谓聪明、有创意却不安全的 hack 就能跨域进行安全对话。在这方面，我们肯定不会怀念过去的“好”时光了。

### 2. 表现和内容分离

在清晰分离表现和内容方面，HTML5 迈出了巨大的步伐。HTML5 在所有可能的地方都努力进行了分离，也包括 CSS。实际上，HTML5 规范已经不支持老版本 HTML 的大部分表现功能了，但得益于先前提到的 HTML5 在兼容性方面的设计理念，那些功能仍然能用。表现和内容分离的概念也不是全新的，在 HTML 4 Transitional 和 XHTML 1.1 中就已经开始用了。Web 设计者把这个概念当做最佳实践使用了很久，不过现在清晰地分开表现和内容显得更为重要，否则会有如下弊端：

- 可访问性差；
- 不必要的复杂度（所有样式代码都放在页面中，代码可读性很差）；
- 文件变大（样式内容越多，文件越大），带来的后果就是页面载入变慢。

## 1.4.3 化繁为简

HTML5 要的就是简单、避免不必要的复杂性。HTML5 的口号是“简单至上，尽可能简化”。因此，HTML5 做了以下这些改进：



- 以浏览器原生能力替代复杂的 JavaScript 代码；
- 新的简化的 DOCTYPE；
- 新的简化的字符集声明；
- 简单而强大的 HTML5 API。

随后我们将详细讲解这些改进。

为了实现所有的这些简化操作，HTML5 规范已经变得非常大，因为它需要精确再精确。实际上要比以往任何版本的 HTML 规范都要精确。为了达到在 2022 年能够真正实现浏览器互通的目标，HTML5 规范制订了一系列定义明确的行为；任何歧义和含糊都可能延缓这一目标的实现。

另外，HTML5 规范比以往的任何版本都要详细，为的是避免造成误解。HTML5 规范的目标是完全、彻底地给出定义，特别是对 Web 应用。所以也难怪，整个规范超过了 900 页！

基于多种改进过的、强大的错误处理方案，HTML5 具备了良好的错误处理机制。非常有现实意义的一点是，HTML5 提倡重大错误的平缓恢复，再次把最终用户的利益放在了第一位。比如，如果页面中有错误的话，在以前可能会影响整个页面的显示，而 HTML5 不会出现这种情况，取而代之的是以标准方式显示“broken”标记，这要归功于 HTML5 中精确定义的错误恢复机制。

#### 1.4.4 通用访问

这个原则可以分成三个概念。

- 可访问性：出于对残障用户的考虑，HTML5 与 WAI (Web Accessibility Initiative, Web 可访问性倡议) 和 ARIA (Accessible Rich Internet Applications, 可访问的富 Internet 应用) 做到了紧密结合，WAI-ARIA 中以屏幕阅读器为基础的元素已经被添加到 HTML 中。
- 媒体中立：如果可能的话，HTML5 的功能在所有不同的设备和平台上应该都能正常运行。
- 支持所有语种：例如，新的<ruby>元素支持在东亚页面排版中会用到的 Ruby 注释。

## 1.5 无插件范式

过去，很多功能只能通过插件或者复杂的 hack (本地绘图 API、本地 socket 等) 来实现，但在 HTML5 中提供了对这些功能的原生支持。插件的方式存在很多问题：

- 插件安装可能失败；
- 插件可以被禁用或屏蔽 (例如 Apple 的 iPad 就不支持 Flash 插件)；
- 插件自身会成为被攻击的对象；
- 插件不容易与 HTML 文档的其他部分集成 (因为插件边界、剪裁和透明度问题)。

虽然一些插件的安装率很高，但在控制严格的公司内部网络环境中经常会被封锁。此外，由于插件经常还会给用户带来烦人的广告，一些用户也会选择屏蔽此类插件。这样的话，一旦用户禁用了插件，就意味着依赖该插件显示的内容也无法表现出来了。

在已经设计好的页面中，要想把插件显示的内容与页面上其他元素集成也比较困难，因为会引起剪裁和透明度等问题。插件使用的是自带的渲染模型，与普通 Web 页面所使用的不一样，所以当弹出菜单或者其他可视化元素与插件重叠时，会特别麻烦。此时，HTML5 却可以站出来，



挥舞着它的原生功能魔棒，对这类问题笑而不语，它可以直接用 CSS 和 JavaScript 的方式控制页面布局。实际上这是 HTML5 的最大亮点，显示了先前任何 HTML 版本都不具备的强大能力。HTML5 不仅仅是提供新元素支持新功能，更重要的是添加了对脚本和布局之间的原生交互能力，基于此，我们可以实现以前不能实现的效果。

以 HTML5 中的 canvas 元素为例，有很多非常底层的事情以前是没办法做到的（比如在 HTML4 的页面中就难画出对角线），而有了 canvas 就可以很轻易地实现了。更为重要的是新 API 释放出来的潜能，以及通过寥寥几行 CSS 代码就能完成布局的能力。基于 HTML5 的各类 API 的优秀设计，我们可以轻松地对它们进行组合应用。比如，从 video 元素中抓取的帧可以显示在 canvas 里面，用户点击 canvas 即可播放这帧对应的视频文件。这只是一个使用原生方法实现插件功能的示例。其实，当工作不再基于黑盒后，开发反而会变得更简单。HTML5 的不同功能组合应用为 Web 开发注入了一股强大的新生力量，这也是我们为什么决定写一本关于 HTML5 编程的书，而不单单是介绍那些新元素的原因。

## HTML5 包括什么，不包括什么

那么，HTML5 到底包括些什么？仔细阅读过规范的读者，可能会发现本书中讲解的很多功能其实在规范中是没有的。例如，Geolocation 和 Web Workers 就不在规范中。那为什么还要将它们纳入本书的讨论范围呢？炒作？当然不是！

很多 HTML5 的研究成果（如 Web Storage 和 Canvas 2D）起初都是 HTML5 规范的一部分，后来移出来列入了单独的规范中，这么做是为了让 HTML5 规范更好地突出重点。在成为官方规范之前，先单列出来进行讨论和编辑不失为一个好办法。这样的话，即使存在争议，也不会影响到整个规范。

在讨论某个功能的时候，特定领域的专家可通过邮件列表的方式共同探讨，不会因为喋喋不休而引起激辩。业界仍然倾向于把原始功能集都视为 HTML5，其中包括 Geolocation。这样的话，HTML5 不仅涵盖了核心的标记元素，同时也可以包括很多很酷的新 API。写这本书的时候，下面这些功能也属于 HTML5：

- Canvas (2D 和 3D)
- Channel 消息传送
- Cross-document 消息传送
- Geolocation
- MathML
- Microdata
- Server-Sent Events
- Scalable Vector Graphics (SVG)
- WebSocket API 及协议
- Web Origin Concept
- Web Storage



- Web SQL database
- Web Workers
- XMLHttpRequest Level 2

可以看到本书中所讨论到的 API 大多都在上面的列表中。为什么选择这些 API 呢？我们挑选的都是基本成熟的功能，也就是说这些功能已经得到了不止一种浏览器的支持，其他功能（不太成熟的）可能只在个别浏览器的某个 beta 版中可用，或者基本上只是个概念。

对于我们要讨论的 HTML5 功能，本书将提供各种浏览器的最新支持情况。不过，不管现在支持情况如何，不久的将来肯定会变，因为 HTML5 发展的速度非常快。不用担心，我们会推荐一些非常好的在线资源，用以查看当前（以及将来）浏览器的支持情况。www.canIuse.com 网站按照浏览器的版本提供了详尽的 HTML5 功能支持情况。若用户通过浏览器访问 www.html5test.com 的话，该网站会直接显示用户浏览器对 HTML5 规范的支持情况。

此外，本书的重点不是讨论使用某种模拟或者变通的方法让 HTML5 程序能够运行在旧浏览器上，而是着重关注 HTML5 规范本身，以及它的使用方法。也就是说，在本书中我们针对所讨论的每个 API 都会提供一些示例代码，开发人员可以直接拿来检测其可用性。因为检测用户代理的方式经常不可靠，所以我们使用特性检测。当然，还可以使用 Modernizr —— 一个 JavaScript 库，它提供了非常先进的 HTML5 和 CSS3 检测功能。我们强烈推荐使用 Modernizr，因为它是检测浏览器是否支持某些特性的最佳工具。

#### 对于 HTML 多说几句

“大家好，我是 Frank<sup>①</sup>，我喜欢画画。

我见过的第一个 HTML canvas 演示是一款简单的绘图程序，用户界面模仿的是微软的画图程序。尽管那落后于数码绘图几十年，并且当时只有个别浏览器支持，不过它却让我对其表现能力充满了期待。

当我开始数码绘图的时候，一般都使用安装在本地的桌面软件。不可否认，有些软件相当不错，但它们不具备 Web 应用的迷人特性。简而言之，这些软件都是离线的。想要共享数码作品，必须先从软件中将图像导出，然后上传至 Web。但是在 Web 的实时画布上协作讨论的话就不存在这种问题了。HTML5 应用可以省掉现在数码绘图流程中的导出环节，将整个创作过程都转移到线上，直至作品完成。

不能用 HTML5 实现的应用已经变得越来越少了。对于文本，Web 已然成为双向沟通的理想媒介。通过全 Web 的方式处理文本的应用程序比比皆是，而类似绘图、视频编辑、3D 建模等图形类程序才刚刚起步。

现在，我们已经开发出了许多功能强大的单机软件，用以创建和欣赏图片、音乐、电影等。更进一步，我们可以将这些软件移植到 Web 这个功能强大、无处不在的在线平台上。”

——Frank

<sup>①</sup> Frank，本书作者之一。——译者注

## 1.6 HTML5 的新功能

在讨论 HTML5 编程之前，让我们快速预览一下 HTML5 的新功能。

### 1.6.1 新的 DOCTYPE 和字符集

首先，根据 HTML5 设计准则的第 3 条——化繁为简，Web 页面的 DOCTYPE 被极大地简化了。以下面这段 HTML4 DOCTYPE 代码为例进行对比：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

谁能记得住？所以在新建页面的时候，我们往往只能通过复制粘贴的方式添加这么长的 DOCTYPE，同时脑子里还不确定复制的对不对。HTML5 干净利索地解决了这个问题：

```
<!DOCTYPE html>
```

现在的 DOCTYPE 好记多了。跟 DOCTYPE 一样，字符集的声明也被简化了。过去是这样的：

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

现在成了：

```
<meta charset="utf-8">
```

使用新的 DOCTYPE 后，浏览器默认以标准模式（standards mode）显示页面。例如，用 Firefox 打开一个 HTML5 页面，然后点击“工具>页面信息”（Tools > Page Info），会看到图 1-1 所示的画面。示例页面是以标准模式显示的。

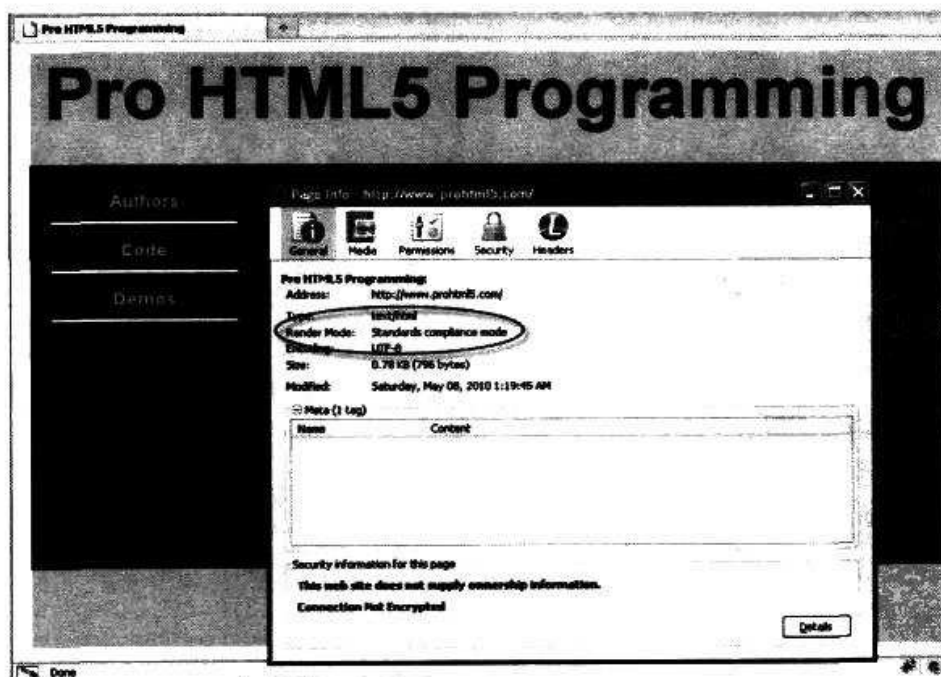


图 1-1 标准兼容模式下显示的页面



使用 HTML5 的 DOCTYPE 会触发浏览器以标准兼容模式显示页面。众所周知，Web 页面有多种显示模式，比如怪异模式 (Quirks)、近标准模式 (Almost Standards) 以及标准模式 (Standards)。其中标准模式也被称为非怪异模式 (no-quirks)。浏览器会根据 DOCTYPE 来识别该使用哪种模式，以及使用什么规则来验证页面。在怪异模式下，浏览器会尽量不中断页面显示，即使没有完全通过验证也会将其显示出来。HTML5 引入了新的标记元素和其他机制 (随后会详细讨论)，因此如果坚持使用已废弃的元素，那么页面将无法通过验证。

## 1.6.2 新元素和旧元素

HTML5 引入了很多新的标记元素，根据内容类型的不同，这些元素被分成了 7 大类。见表 1-1。

表1-1 HTML5的内容类型

内容类型	描 述
内嵌流	向文档中添加其他类型的内容，例如audio、video、canvas和iframe等
标题	在文档和应用的body中使用的元素，例如form、h1和small等
交互	段落标题，例如h1、h2和hgroup等
元数据	与用户交互的内容，例如音频和视频的控件、button和textarea等
短语	通常出现在页面的head中，设置页面其他部分的表现和行为，例如script、style和title等
片段	文本和文本标记元素，例如mark、kbd、sub和sup等
	用于定义页面片段的元素，例如article、aside和title等

上述所有类型的元素都可以通过 CSS 来设定样式。此外，虽然其中一些元素，如 canvas、audio 和 video，在使用时往往需要其他 API 来配合，以实现细粒度控制，但它们同样可以直接使用。我们在后续章节中详细讨论这类元素 API。

限于篇幅，本书讨论的内容无法涵盖所有新元素，不过片段类元素是全新的，我们会在下一节讨论，而 canvas、audio 和 video 作为 HTML5 新增的元素也会在后续章节中详细讨论。

同样地，对于旧的标签元素，网上的资料已经很多了，我们不会把所有旧的标签元素都罗列出来。不过需要注意的是，HTML5 中移除了很多在行内设样式的标记元素，如 big、center、font 和 basefont 等，以鼓励开发人员使用 CSS。

## 1.6.3 语义化标记

片段类的内容类型包含许多 HTML5 元素。HTML5 定义了一种新的语义化标记来描述元素的内容。虽然语义化标记不会让你马上感受到有什么好处，但是它可以简化 HTML 页面设计，并且将来搜索引擎在抓取和索引网页的时候，也绝对会利用到这些元素的优势。

前面我们说过，HTML5 的宗旨之一就是存在即合理。Google 分析了上百万的页面，从中发现了 DIV 标签的通用 ID 名称重复量很大。例如，很多开发人员喜欢使用 DIV id="footer" 来标记页脚内容，所以 HTML5 引入了一组新的片段类元素，在目前主流的浏览器中已经可以用了。表 1-2 列出了新增的语义化标记元素。

表1-2 HTML5中新的片段类元素

元素名	描述
header	标记头部区域的内容（用于整个页面或页面中的一块区域）
footer	标记脚部区域的内容（用于整个页面或页面中的一块区域）
section	Web页面中的一块区域
article	独立的文章内容
aside	相关内容或者引文
nav	导航类辅助内容

上面所有的元素都能用 CSS 设定样式。之前说到了 HTML5 效率优先的设计理念，它推崇表现和内容的分离，所以在 HTML5 的实际编程中，开发人员必须使用 CSS 来定义样式。代码清单 1-1 是一个 HTML5 页面的概貌，其中使用了新的 DOCTYPE、字符集和语义化标记元素——新的片段类元素。示例代码对应的源码在 code/intro 文件夹中。

#### 代码清单 1-1 HTML5 示例页面

```

<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8" >
  <title>HTML5</title>
  <link rel="stylesheet" href="html5.css">
</head>

<body>

  <header>
    <h1>Header</h1>
    <h2>Subtitle</h2>
    <h4>HTML5 Rocks!</h4>
  </header>

  <div id="container">
    <nav>
      <h3>Nav</h3>
      <a href="http://www.example.com">Link 1</a>
      <a href="http://www.example.com">Link 2</a>
      <a href="http://www.example.com">Link 3</a>
    </nav>

    <section>
      <article>
        <header>
          <h1>Article Header</h1>
        </header>
        <p>Lorem ipsum dolor HTML5 nunc aut nunquam sit amet, consectetur adipiscing
          elit. Vivamus at est eros, vel fringilla urna.</p>

```



```

    <p>Per inceptos himenaeos. Quisque feugiat, justo at vehicula
      pellentesque, turpis lorem dictum nunc.</p>
  <footer>
    <h2>Article Footer</h2>
  </footer>
</article>

<article>
  <header>
    <h1>Article Header</h1>
  </header>
  <p>HTML5: "Lorem ipsum dolor nunc aut nunquam sit amet, consectetur adipiscing
    elit. Vivamus at est eros, vel fringilla urna. Pellentesque odio</p>

  <footer>
    <h2>Article Footer</h2>
  </footer>
</article>

</section>

  <aside>
    <h3>Aside</h3>
    <p>HTML5: "Lorem ipsum dolor nunc aut nunquam sit amet, consectetur adipiscing
      elit. Vivamus at est eros, vel fringilla urna. Pellentesque odio rhoncus</p>
  </aside>

  <footer>
    <h2>Footer</h2>
  </footer>
</div>
</body>

</html>

```

没有样式的页面看起来有些枯燥乏味。代码清单1-2是一些可以用来设置内容样式的 CSS 代码。需要注意的是，这份样式表使用了 CSS3 的一些新特性，比如圆角 (border-radius) 和旋转变换 (transform: rotate();)。CSS3 同 HTML5 一样也正在开发过程中，并且为了便于浏览器逐步支持，也采用了模块化的方式发布子规范（例如变换 (transformation)、动画 (animation) 和过渡 (transition) 分别对应不同的子规范）。

CSS3 的规范很可能还会变动，CSS3 中的功能也处于实验期，因此为了避免命名空间冲突，这些功能都会加上浏览器厂商的前缀。要显示圆角、渐变 (gradients)、阴影 (shadows) 和变形 (transformations) 的话，需要在声明的部分加上前缀：-moz- (Mozilla 浏览器)、o- (Opera 浏览器) 和 -webkit- (Safari 和 Chrome 等基于 WebKit 核心的浏览器)。

#### 代码清单 1-2 HTML5 页面对应的 CSS 文件

```
body {
```

```
        background-color: #CCCCCC;
        font-family: Geneva, Arial, Helvetica, sans-serif;
        margin: 0px auto;
        max-width: 900px;
        border: solid;
        border-color: #FFFFFF;
    }

    header {
        background-color: #F47D31;
        display: block;
        color: #FFFFFF;
        text-align: center;
    }

    header h2 {
        margin: 0px;
    }

    h1 {
        font-size: 72px;
        margin: 0px;
    }

    h2 {
        font-size: 24px;
        margin: 0px;
        text-align: center;
        color: #F47D31;
    }

    h3 {
        font-size: 18px;
        margin: 0px;
        text-align: center;
        color: #F47D31;
    }

    h4 {
        color: #F47D31;
        background-color: #fff;
        -webkit-box-shadow: 2px 2px 20px #888;
        -webkit-transform: rotate(-45deg);
        -moz-box-shadow: 2px 2px 20px #888;
        -moz-transform: rotate(-45deg);
        position: absolute;
        padding: 0px 150px;
        top: 50px;
        left: -120px;
        text-align: center;
    }

    nav {
        display: block;
```



```
        width:25%;
        float:left;
    }
    nav a:link, nav a:visited {
        display: block;
        border-bottom: 3px solid #fff;
        padding: 10px;
        text-decoration: none;
        font-weight: bold;
        margin: 5px;
    }
    nav a:hover {
        color: white;
        background-color: #F47D31;
    }
    nav h3 {
        margin: 15px;
        color: white;
    }
    #container {
        background-color: #888;
    }
    section {
        display:block;
        width:50%;
        float:left;
    }
    article {
        background-color: #eee;
        display:block;
        margin: 10px;
        padding: 10px;
        -webkit-border-radius: 10px;
        -moz-border-radius: 10px;
        border-radius: 10px;
        -webkit-box-shadow: 2px 2px 20px #888;
        -webkit-transform: rotate(-10deg);
        -moz-box-shadow: 2px 2px 20px #888;
        -moz-transform: rotate(-10deg);
    }
    article header {
        -webkit-border-radius: 10px;
        -moz-border-radius: 10px;
        border-radius: 10px;
        padding: 5px;
    }
```

```
article footer {
    -webkit-border-radius: 10px;
    -moz-border-radius: 10px;
    border-radius: 10px;
    padding: 5px;
}

article h1 {
    font-size: 18px;
}

aside {
    display: block;
    width: 25%;
    float: left;
}

aside h3 {
    margin: 15px;
    color: white;
}

aside p {
    margin: 15px;
    color: white;
    font-weight: bold;
    font-style: italic;
}

footer {
    clear: both;
    display: block;
    background-color: #F47D31;
    color: #FFFFFF;
    text-align: center;
    padding: 15px;
}

footer h2 {
    font-size: 14px;
    color: white;
}

/* links */
a {
    color: #F47D31;
}

a:hover {
    text-decoration: underline;
}
```

图 1-2 是代码清单 1-1 中的页面应用了 CSS (包括部分 CSS3) 之后的显示效果。其实并不能把这个页面当成所谓的典型 HTML5 页面。因为计划赶不上变化, 这个示例使用了很多新标签只是为了演示而已。

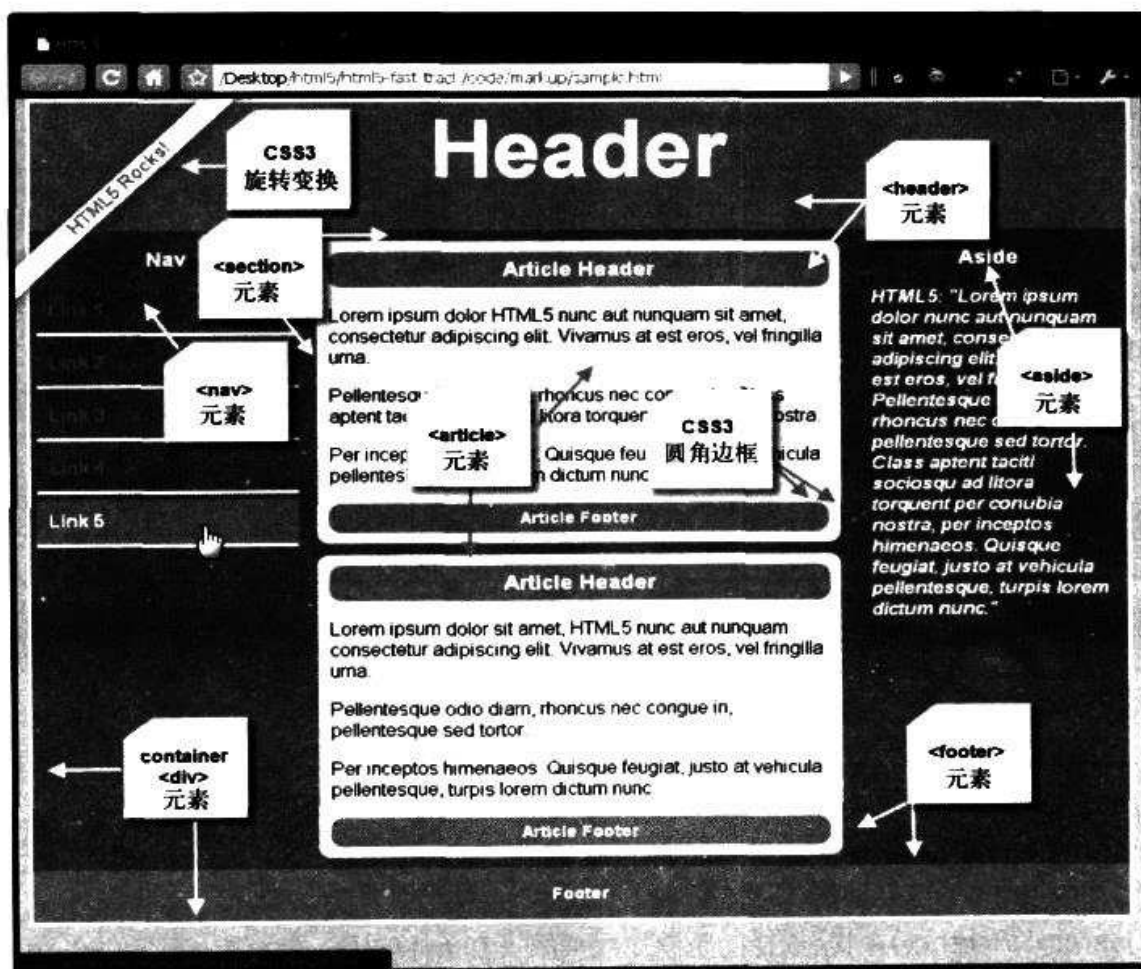


图 1-2 使用了所有新的语义化标记元素的 HTML5 页面

最后需要说明的是, 看起来好像浏览器是因为识别了新的元素, 所以显示出了对应的内容。其实不然, 事实上这些元素很可能是先被重命名为了 foo 或者 bar, 然后再应用样式, 最后才显示出来的 (当然, 对于搜索引擎优化来说没有任何好处)。IE 是个特例, 因为 IE 需要将这些元素都作为 DOM 的一部分, 所以要想在 IE 中看到这些元素, 必须用编程的方式把它们插入 DOM 中, 然后再以块元素 (block element) 的形式显示出来。

#### 1.6.4 使用 Selectors API 简化选取操作

除了语义化元素外, HTML5 还引入了一种用于查找页面 DOM 元素的快捷方式。表 1-3 列出了在 HTML5 出现之前, 用来在页面中查找特定元素的函数。



表1-3 以前用来查找元素的JavaScript方法

函 数	描 述	示 例
getElementById()	根据指定的id特性值查找并返回元素	<code>&lt;div id="foo"&gt; getElementById("foo");</code>
getElementsByName()	返回所有name特性为指定值的元素	<code>&lt;input type="text" name="foo"&gt; getElementsByName("foo");</code>
getElementsByTagName()	返回所有标签名称与指定值相匹配的元素	<code>&lt;input type="text"&gt; getElementsByTagName("input");</code>

有了新的 Selectors API 之后, 可以用更精确的方式来指定希望获取的元素, 而不必再用标准 DOM 的方式循环遍历。Selectors API 与现在 CSS 中使用的选择规则一样, 通过它我们可以查找页面中的一个或多个元素。例如, CSS 已经可以基于嵌套 (nesting)、兄弟节点 (sibling) 和子模式 (child pattern) 进行元素选择。CSS 的最新版除添加了更多对伪类 (pseudo-class) 的支持 (例如判断一个对象是否被启用、禁用或者被选择等), 还支持对属性和层次的随意组合叠加。使用表 1-4 中的函数就能按照 CSS 规则来选取 DOM 中的元素。

表1-4 新querySelector方法

函 数	描 述	示 例	结 果
querySelector()	根据指定的选择规则, 返回在页面中找到的第一个匹配元素	<code>querySelector("input.error");</code>	返回第一个CSS类名为 "error" 的文本输入框
querySelectorAll()	根据指定规则返回页面中所有相匹配的元素	<code>querySelectorAll("#results td");</code>	返回id值为results的元素下所有的单元格

可以为 Selectors API 函数同时指定多个选择规则, 例如:

```
// 选择文档中类名为 highClass 或 lowClass 的第一个元素
var x = document.querySelector(".highClass", ".lowClass");
```

对于 querySelector() 来说, 选择的是满足规则中任意条件的第一个元素。对于 querySelectorAll() 来说, 页面中的元素只要满足规则中的任何一个条件, 都会被返回。多条规则是用逗号分隔的。

以前在页面上跟踪用户操作很困难, 但新的 Selectors API 提供了更为便捷的方法。比如, 页面上有一个表格, 我们想获取鼠标当前在哪个单元格上。从代码清单 1-3 中可以看到使用 Selectors API 实现有多简单。这份源代码也可以从 code/intro 路径下找到。

#### 代码清单 1-3 使用 Selector API

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8" />
  <title>Query Selector Demo</title>
```

```
<style type="text/css">
  td {
    border-style: solid;
    border-width: 1px;
    font-size: 300%;
  }

  td:hover {
    background-color: cyan;
  }

  #hoverResult {
    color: green;
    font-size: 200%;
  }
</style>
</head>

<body>
  <section>
    <!-- create a table with a 3 by 3 cell display -->
    <table>
      <tr>
        <td>A1</td> <td>A2</td> <td>A3</td>
      </tr>
      <tr>
        <td>B1</td> <td>B2</td> <td>B3</td>
      </tr>
      <tr>
        <td>C1</td> <td>C2</td> <td>C3</td>
      </tr>
    </table>

    <div>Focus the button, hover over the table cells, and hit Enter to identify them
    using querySelector('td:hover').</div>
    <button type="button" id="findHover" autofocus>Find 'td:hover' target</button>
    <div id="hoverResult"></div>

    <script type="text/javascript">
      document.getElementById("findHover").onclick = function() {
        // 找到鼠标当前悬停的单元格
        var hovered = document.querySelector("td:hover");
        if (hovered)
          document.getElementById("hoverResult").innerHTML = hovered.innerHTML;
      }
    </script>
  </section>

</body>
</html>
```

从以上示例可以看到，仅用一行代码即可找到用户鼠标下面的元素：



```
var hovered = document.querySelector("td:hover");
```

**提示** Selectors API不仅仅只是方便，在遍历DOM的时候，Selectors API通常会比以前的子节点搜索API更快。为了实现快速样式表，浏览器对选择器匹配进行了高度优化。

不难理解为什么 W3C 中的 Selectors API 标准规范会从 CSS 规范中单独分离出来，从上面的代码也可以看出来，Selectors API 在样式应用以外同样大有作为。虽然本书不会深入讲解 Selectors API 的全部细节，但是对于希望优化 DOM 操作方式的 Web 开发人员来说，建议使用新的 Selectors API 以便快速查询应用架构。

### 1.6.5 JavaScript 日志和调试

JavaScript 日志和浏览器内调试从技术上讲虽然不属于 HTML5 的功能，但在过去的几年里，相关工具的发展出现了质的飞跃。第一个可以用来分析 Web 页面及其所运行脚本的强大工具是一款名为 Firebug 的 Firefox 插件。

现在，相同的功能在其他浏览器的内嵌开发工具中也可以找到：Safari 的 Web Inspector、Google 的 Chrome 开发者工具 (Developer Tools)、IE 的开发者工具 (Developer Tools)，以及 Opera 的 Dragonfly。图 1-3 是 Google 的 Chrome 开发者工具截图，显示了大量与当前 Web 页面相关的信息（使用快捷键 Ctrl+Shift+J 可以看到），包括调试控制台、资源视图、存储视图等。



图 1-3 Chrome 的开发者工具

很多调试工具支持设置断点来暂停代码执行、分析程序状态以及查看变量的当前值。



console.log API 已经成为 JavaScript 开发人员记录日志的事实标准。为了便于开发人员查看记录到控制台的信息，很多浏览器提供了分栏窗格的视图。console.log API 要比 alert() 好用很多，因为它不会阻塞脚本的执行。

### 1.6.6 window.JSON

JSON 是一种相对来说比较新并且正在日益流行的数据交换格式。作为 JavaScript 语法的一个子集，它将数据表示为对象字面量。由于其语法简单和在 JavaScript 编程中与生俱来的兼容性，JSON 变成了 HTML5 应用内部数据交换的事实标准。典型的 JSON API 包含两个函数，parse() 和 stringify()（分别用于将字符串序列化成为 DOM 对象和将 DOM 对象转换成字符串）。

如果在旧的浏览器中使用 JSON，需要 JavaScript 库（有些可以从 <http://json.org> 找到）。在 JavaScript 中执行解析和序列化效率往往不高，所以为了提高执行速度，现在新的浏览器原生扩展了对 JSON 的支持，可以直接通过 JavaScript 来调用 JSON 了。这种本地化的 JSON 对象被纳入了 ECMAScript 5 标准，成为了下一代 JavaScript 语言的一部分。它也是 ECMAScript 5 标准中首批被浏览器支持的功能之一。所有新的浏览器都支持 window.JSON，将来 JSON 必将大量应用于 HTML5 应用中。

### 1.6.7 DOM Level 3

事件处理是目前 Web 应用开发中最令人头疼的部分之一。除了 IE 以外，绝大多数浏览器都支持处理事件和元素的标准 API。早期 IE 实现的是与最终标准不同的事件模型，而 IE9 将会支持 DOM Level 2 和 DOM Level 3 的特性。如此，在所有支持 HTML5 的浏览器中，我们终于可以使用相同的代码来实现 DOM 操作和事件处理了，包括非常重要的 addEventListener() 和 dispatchEvent() 方法。

### 1.6.8 Monkeys、Squirrelfish 和其他 JavaScript 引擎

最新一轮的浏览器创新不仅仅是增加了新的标签和 API。最重要的变化之一是主流浏览器中 JavaScript/ECMAScript 引擎飞快的升级。新的 API 提供了很多上一代浏览器无法实现的功能，因而脚本引擎整体执行效率的提升，不论对现有的，还是使用了最新 HTML5 特性的 Web 应用都有好处。还记得浏览器在显示复杂图像、处理数据或者编辑长篇文章时，明显变得迟钝的情景吗？再好好想一想。

最近几年，浏览器厂商争相比拼，看谁能开发出更快的 JavaScript 引擎。过去的 JavaScript 纯粹是被解释执行，而最新的引擎则直接将脚本编译成原生机器代码，相比 2005 年前后的浏览器，速度的提升已经不在一个数量级上了。

大约从 2006 年 Adobe 将其 JIT 编译引擎和代号为 Tamarin 的 ECMAScript 虚拟机捐赠给 Mozilla 基金会开始，竞争的序幕就拉开了。尽管新版的 Mozilla 中 Tamarin 技术已经所剩无几，但 Tamarin 的捐赠促进了各家浏览器对新脚本引擎的研发，而这些引擎的名字就如同他们声称的性能一样有意思。



表1-5 Web浏览器的JavaScript引擎

浏览器	引擎名称	备注
Apple Safari 5	Nitro (也被称作Squirrel Fish Extreme)	Safari 4中发布, 在Safari 5中提升性能, 包括字节码优化和上下文线程的本地编译器
Google Chrome 5	V8	自从Chrome 2开始, 使用了新一代垃圾回收机制, 可确保内存高度可扩展而不会发生中断
Microsoft Internet Explorer 9	Chakra	注重于后台编译和高效的类型系统, 速度比IE8快10倍
Mozilla Firefox 4	JägerMonkey	从3.5版本优化而来, 结合了快速解释和源自追踪树(trace tree)的本地编译
Opera 10.60	Carakan	它采用了基于寄存器的字节码和选择性本地编译的方式, 声称效率比10.50版本提升了75%

总之, 得益于浏览器厂商间的良性竞争, JavaScript 的执行性能越来越接近于本地桌面应用程序。

#### 关于 HTML 再多说两句

“我的名字叫 Peter<sup>①</sup>, 说起竞争和疯狂的速度, 我非常喜欢跑步。

马拉松是一项伟大的运动, 从中可以发现伟大的人。当跑到百公里赛或者 165 公里长跑的最后阶段时, 你真的可以通过一种新的方式来认识一些志同道合的人。因为在这个时候, 人们放下了自己的架子, 为真正伟大友谊的诞生提供了机会。可以肯定, 竞争的因素仍然存在, 但更重要的是那份深切的情谊。哦, 我有点儿跑题了。

有的比赛我没有时间参加 (比如在写这本 HTML5 的书的时候), 但我想知道比赛中我的朋友们表现如何, 于是通常会上比赛网站去看。当然了, 网站中的‘实时跟踪’功能往往不那么可靠。

几年前, 我偶然间遇到一家为欧洲赛事建立的网站, 这个网站的思路是完全正确的。他们为跑在前面的运动员安装了 GPS 定位器, 然后在地图上显示出来 (本书中我们将使用 Geolocation 和 WebSocket 来模拟实现)。尽管事实上执行起来比较麻烦 (为了看到最新的数据, 用户必须频繁点击‘刷新’), 但是我从中仍然看到了难以置信的潜力。

现在, 仅仅过了几年, HTML5 便通过 API 的方式为我们提供了建立这类实时比赛网站所需要的工具, 例如位置服务应用所需的 Geolocation, 以及用来支持实时更新的 WebSocket。在我看来, 毋庸置疑, HTML5 冲破了终点线成为了赢家!”

——Peter

## 1.7 小结

本章概述了 HTML5 的重要特性。

我们讨论了 HTML5 的开发历史和即将迎来的几个重要时间点, 还讲述了 HTML5 时代的四

<sup>①</sup> Peter, 本书作者之一。——译者注

一个新设计准则：兼容性、实用性、互通性和通用访问性。每项设计准则都打开了一扇大门，同时也宣告了已经过时的惯例和约定的消亡。接着，我们介绍了 HTML5 令人意想不到的新的无插件范式，回答了每个人都挂在嘴边的问题——HTML5 规范到底包括什么，不包括什么，还回顾了 HTML5 的新特性，例如新的 DOCTYPE 和字符集声明，许多新的标记元素等，最后我们讨论了 JavaScript 引擎的竞争。

下一章起，我们开始探索 HTML5 编程，旅途的起点是 Canvas API。



**在**本章中，我们将探索如何使用 HTML5 的 Canvas API。Canvas API 很酷，可以通过它来动态生成和展示图形、图表、图像以及动画。本章将使用渲染 API (rendering API) 的基本功能来创建一幅可以放大缩小并自适应浏览器环境的图。还会演示如何基于用户输入来动态创建图像，生成热点图。当然，我们也会提醒你在使用 HTML5 Canvas 时需要注意的问题，并且分享解决这些问题的方法。

本章只涉及了最基本的图形知识，因此，你大可不必担心学不会而跳过本章。来吧，让我们一起来感受 HTML5 中这个强大的特性吧。

## 2.1 HTML5 Canvas 概述

关于 HTML5 Canvas API 完全可以写一本书（还不会是一本很薄的书）。由于只有一章的篇幅，所以我们将讨论 API 中那些我们认为是最常用的功能。

### 2.1.1 历史

Canvas 的概念最初是由苹果公司提出的，用于在 Mac OS X WebKit 中创建控制板部件 (dashboard widget)。在 Canvas 出现之前，开发人员若要在浏览器中使用绘图 API，只能使用 Adobe 的 Flash 和 SVG (Scalable Vector Graphics, 可伸缩矢量图形) 插件，或者只有 IE 才支持的 VML (Vector Markup Language, 矢量标记语言)，以及其他一些稀奇古怪的 JavaScript 技巧。

假设我们要在没有 canvas 元素的条件下绘制一条对角线——听起来似乎很简单，但实际上如果没有一套二维绘图 API 的话，这会是一项相当复杂的工作。HTML5 Canvas 能够提供这样的功能，对浏览器端来说此功能非常有用，因此 Canvas 被纳入了 HTML5 规范。

起初，苹果公司曾暗示可能会为 WHATWG (Web Hypertext Application Technology Working Group, Web 超文本应用技术工作组) 草案中的 Canvas 规范申请知识产权，这在当时引起了一些 Web 标准化追随者的关注。不过，苹果公司最终还是按照 W3C 的免版税专利权许可条款公开了其专利。

## SVG 和 CANVAS 对比

“Canvas 本质上是一个位图画布，其上绘制的图形是不可缩放的，不能像 SVG 图像那样可以被放大缩小。此外，用 Canvas 绘制出来的对象不属于页面 DOM 结构或者任何命名空间——这点被认为是一个缺陷。SVG 图像却可以在不同的分辨率下流畅地缩放，并且支持点击检测（能检测到鼠标点击了图像上的哪个点）。

既然如此，为什么 WHATWG 的 HTML5 规范不使用 SVG 呢？尽管 Canvas 有明显的不足，但 HTML Canvas API 有两方面优势可以弥补：首先，不需要将所绘制图像中的每个图元当做对象存储，因此执行性能非常好；其次，在其他编程语言现有的优秀二维绘图 API 的基础上实现 Canvas API 相对来说比较简单。毕竟，二鸟在林不如一鸟在手。”

——Peter

## 2.1.2 canvas 是什么

在网页上使用 canvas 元素时，它会创建一块矩形区域。默认情况下该矩形区域宽为 300 像素，高为 150 像素，但也可以自定义具体的大小或者设置 canvas 元素的其他特性。代码清单 2-1 是可放到 HTML 页面中的最基本的 canvas 元素。

### 代码清单 2-1 基本的 canvas 元素

```
<canvas></canvas>
```

在页面中加入了 canvas 元素后，我们便可以通过 JavaScript 来自由地控制它。可以在其中添加图片、线条以及文字，也可以在里面绘图，甚至还可以加入高级动画。

大多数主流操作系统和框架支持的二维绘制操作，HTML5 Canvas API 都支持。如果你在近年来曾经有过二维图像编程的经验，那么会对 HTML5 Canvas API 感觉非常顺手，因为这个 API 就是参照既有系统设计的。如果没有这方面经验，则会发现与这么多年来一直使用的图片加 CSS 开发 Web 图形的方式比起来，Canvas 的渲染系统有多么强大。

使用 canvas 编程，首先要获取其上下文 (context)。接着在上下文中执行动作，最后将这些动作应用到上下文中。可以将 canvas 的这种编辑方式想象成为数据库事务：开发人员先发起一个事务，然后执行某些操作，最后提交事务。

### 2.1.3 canvas 坐标

如图 2-1 所示，canvas 中的坐标是从左上角开始的，x 轴沿着水平方向（按像素）向右延伸，y 轴沿垂直方向向下延伸。左上角坐标为  $x=0$ ， $y=0$  的点称作原点。

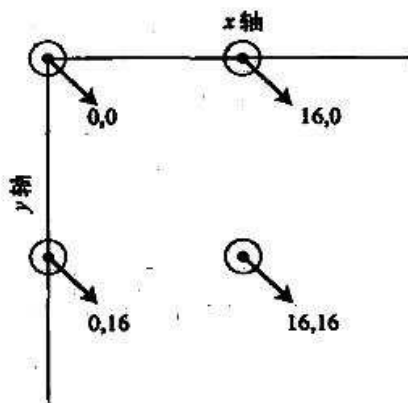


图 2-1 canvas 中的 x、y 坐标



## 2.1.4 什么情况下不用 canvas

尽管 canvas 元素功能非常强大，用处也很多，但在某些情况下，如果其他元素已经够用了，就不应该再使用 canvas 元素。例如，用 canvas 元素在 HTML 页面中动态绘制所有不同的标题，就不如直接使用标题样式标签（H1、H2 等），它们所实现的效果是一样的。

## 2.1.5 替代内容

访问页面的时候，如果浏览器不支持 canvas 元素，或者不支持 HTML5 Canvas API 中的某些特性，那么开发人员最好提供一份替代代码（2.1.7 节中的表 2-1 详细介绍了浏览器对 canvas 的支持情况）。例如，开发人员可以通过一张替代图片或者一些说明性的文字告诉访问者，使用最新的浏览器可以获得更佳的浏览效果。代码清单 2-2 展示了如何在 canvas 中指定替代文本，当浏览器不支持 canvas 的时候会显示这些替代内容。

### 代码清单 2-2 在 canvas 元素中使用替代内容

```
<canvas>
  Update your browser to enjoy canvas!
</canvas>
```

除了上面代码中的文本外，同样还可以使用图片，不论是文本还是图片都会在浏览器不支持 canvas 元素的情况下显示出来。

### canvas 元素的可访问性怎么样

“提供替代图像或替代文本引出了可访问性这个话题——很遗憾，这是 HTML5 Canvas 规范中明显的缺陷。例如，没有一种原生方法能够自动为已插入到 canvas 中的图片生成用于替换的文字说明。同样，也没有原生方法可以生成替代文字以匹配由 Canvas Text API 动态生成的文字。在写本书的时候，暂时还没有其他方法可以处理 canvas 中动态生成的内容，不过已经有工作组开始着手这方面的设计了。让我们一起期待吧。”

——Peter

## 2.1.6 CSS 和 canvas

同大多数 HTML 元素一样，canvas 元素也可以通过应用 CSS 的方式来增加边框，设置内边距、外边距等，而且一些 CSS 属性还可以被 canvas 内的元素继承。比如字体样式，在 canvas 内添加的文字，其样式默认同 canvas 元素本身是一样的。

此外，在 canvas 中为 context 设置属性同样要遵从 CSS 语法。例如，对 context 应用颜色和字体样式，跟在任何 HTML 和 CSS 文档中使用的语法完全一样。

### 2.1.7 浏览器对 HTML5 Canvas 的支持

除了 Internet Explorer 以外，其他所有浏览器现在都提供对 HTML5 Canvas 的支持。不过，随后我们会列出一部分还没有被普遍支持的规范，Canvas Text API 就是其中之一，但是作为一个整体，HTML5 Canvas 规范已经非常成熟，不会有特别大的改动了。从表 2-1 中可以看到，写本书的时候，已经有很多浏览器支持 HTML5 Canvas 了。

表2-1 浏览器对HTML5 Canvas的支持

浏览器	支持情况
Chrome	从1.0版本开始支持
Firefox	从1.5版本开始支持
Internet Explorer	不支持
Opera	从9.0版本开始支持
Safari	从1.3版本开始支持

从上面的表格中可以看出，在所有浏览器中，只有 Internet Explorer 不支持 HTML5 Canvas。如果需要在 Internet Explorer 中使用 canvas，可以选择使用名为 `explorercanvas` 的开源项目 (<http://code.google.com/p/explorercanvas>)。使用 `explorercanvas` 时，需要先判断当前浏览器是否是 Internet Explorer，如果是则在页面中嵌入 `script` 标签来加载 `explorercanvas`。写法如下：

```
<head>
<!--[if IE]><script src="excanvas.js"></script><![endif]-->
</head>
```

开发者迫切希望 Canvas 可以受到广泛支持，因此不断有项目启动来尝试解决老浏览器或者非标准浏览器不支持 Canvas 的问题。微软已经宣布 Internet Explorer 9 将支持 canvas，因此，所有主流浏览器都支持 canvas 已经指日可待了。

由于各家浏览器对 canvas 的支持程度有差异，所以最好在使用 API 之前，先测试一下 HTML5 Canvas 是否被支持。2.2.1 节会讲解怎样通过代码来检测浏览器支持 Canvas 的情况。

## 2.2 使用 HTML5 Canvas API

本节将深入探讨 HTML5 Canvas API。为此，我们将使用各种 HTML5 Canvas API 创建一幅类似于 LOGO 的图像，图像是森林场景，有树，还有适合长跑比赛的美丽跑道。虽然这个示例从平面设计的角度来看毫无竞争力，但却可以合理演示 HTML5 Canvas 的各种功能。

### 2.2.1 检测浏览器支持情况

在创建 HTML5 canvas 元素之前，首先要确保浏览器能够支持它。如果不支持，你就要为那些古董级浏览器提供一些替代文字。代码清单 2-3 就是检测浏览器支持情况的一种方法。

代码清单 2-3 检测浏览器支持情况

```
try {
```



```
document.createElement("canvas").getContext("2d");
document.getElementById("support").innerHTML =
    "HTML5 Canvas is supported in your browser.";
} catch (e) {
    document.getElementById("support").innerHTML = "HTML5 Canvas is not supported ←
        in your browser.";
}
```

上面的代码试图创建一个 canvas 对象，并且获取其上下文。如果发生错误，则可以捕获错误，进而得知该浏览器不支持 canvas。页面中预先放入了 ID 为 support 的元素，通过以适当的信息更新该元素的内容，可以反映出浏览器的支持情况。

以上示例代码能判断浏览器是否支持 canvas 元素，但不会判断具体支持 canvas 的哪些特性。写本书的时候，示例中使用的 API 已经很稳定并且各浏览器也都提供了很好的支持，所以通常不必担心这个问题。

此外，希望开发人员能够像代码清单 2-3 一样为 canvas 元素提供备用显示内容。

## 2.2.2 在页面中加入 canvas

在 HTML 页面中插入 canvas 元素非常直观。代码清单 2-4 就是一段可以被插入到 HTML 页面中的 canvas 代码。

### 代码清单 2-4 canvas 元素

```
<canvas height="200" width="200"></canvas>
```

以上代码会在页面上显示出一块 200×200 像素的“隐藏”区域。假如要为其增加一个边框，可以像代码清单 2-5 中的代码一样，用标准 CSS 边框属性来设置。

### 代码清单 2-5 带实心边框的 canvas 元素

```
<canvas id="diagonal" style="border: 1px solid;" width="200" height="200">
</canvas>
```

注意，上面的代码中增加了一个值为“diagonal”的 ID 特性，这么做的意义在于以后的开发过程中可以通过 ID 来快速找到该 canvas 元素。对于任何 canvas 对象来说，ID 特性都是特别重要的，因为对 canvas 元素的所有操作都是通过脚本代码控制的，没有 ID 的话，想要找到要操作的 canvas 元素会很难。

代码清单 2-5 在浏览器中的执行效果如图 2-2 所示。

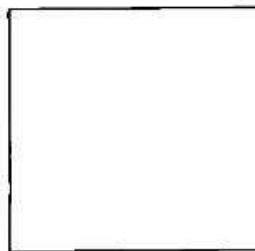


图 2-2 HTML 页面中的简单 canvas 元素

看起来好像没什么，但是就像那些艺术家说的，一张白纸可以画出最新最美的图画。现在，就让我们在这张“白纸”上作画吧。前面说过，在没有 HTML5 Canvas 的情况下，很难在页面上绘制一条对角线。现在我们来看看，有了 Canvas 以后，同样的事情会有多么简单。从代码清单 2-6 中可以看到，基于上面绘制的画布，仅仅使用几行代码就可以画出一条对角线。

#### 代码清单 2-6 在 canvas 中绘制一条对角线

```
<script>
function drawDiagonal() {
    // 取得 canvas 元素及其绘图上下文
    var canvas = document.getElementById('diagonal');
    var context = canvas.getContext('2d');

    // 用绝对坐标来创建一条路径
    context.beginPath();
    context.moveTo(70, 140);
    context.lineTo(140, 70);

    // 将这条线绘制到 canvas 上
    context.stroke();
}

window.addEventListener("load", drawDiagonal, true);
</script>
```

仔细看一下上面这段绘制对角线的 JavaScript 代码。虽然简单，它却展示出了使用 HTML5 Canvas API 的重要流程。

首先通过引用特定的 canvas ID 值来获取对 canvas 对象的访问权。这段代码中 ID 就是 diagonal。接着定义一个 context 变量，调用 canvas 对象的 getContext 方法，并传入希望使用的 canvas 类型。代码清单中通过传入“2d”来获取一个二维上下文，这也是到目前为止唯一可用的上下文。

**提示** 规范未来的某个版本中可能会增加对三维上下文的支持。

接下来，基于这个上下文执行画线的操作。在代码清单中，调用了三个方法——beginPath、moveTo 和 lineTo，传入了这条线的起点和终点的坐标。

方法 moveTo 和 lineTo 实际上并不画线，而是在结束 canvas 操作的时候，通过调用 context.stroke() 方法完成线条的绘制。图 2-3 显示了绘制结果。

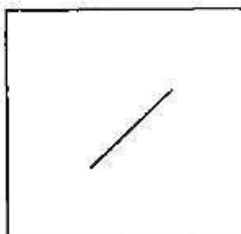


图 2-3 canvas 中的对角线



成功了！虽然从这条简单的线段怎么也想象不到最新最美的图画，不过与以前的拉伸图像、怪异的 CSS 和 DOM 对象以及其他怪异的实现形式相比，使用基本的 HTML 技术在任意两点间绘制一条线段已经是非常大的进步了。从现在开始，就把那些怪异的做法永远忘掉吧。

从上面的代码清单中可以看出，canvas 中所有的操作都是通过上下文对象来完成的。在以后的 canvas 编程中也一样，因为所有涉及视觉输出效果的功能都只能通过上下文对象而不是画布对象来使用。这种设计使 canvas 拥有了良好的可扩展性，基于从其中抽象出的上下文类型，canvas 将来可以支持多种绘制模型。虽然本章经常提到对 canvas 采取什么样的操作，但读者应该明白，我们实际操作的是画布所提供的上下文对象。

如前面示例演示的那样，对上下文的很多操作都不会立即反映到页面上。beginPath、moveTo 以及 lineTo 这些函数都不会直接修改 canvas 的展示结果。canvas 中很多用于设置样式和外观的函数也同样不会直接修改显示结果。只有当对路径应用绘制 (stroke) 或填充 (fill) 方法时，结果才会显示出来。否则，只有在显示图像、显示文本或者绘制、填充和清除矩形框的时候，canvas 才会马上更新。

### 2.2.3 变换

现在我们探讨一下在 canvas 上绘制图像的另一方式——使用变换 (transformation)。接下来的代码清单显示结果跟上面是一样的，只是绘制对角线的代码不一样。这个简单示例可能会让你误认为使用变换增加了不必要的复杂性。事实并非如此，其实变换是实现复杂 canvas 操作的最好方式。在后面的示例中将会看到，我们使用了大量的变换，而这对熟悉 HTML5 Canvas API 的复杂功能是至关重要的。

也许了解变换最简单的方法（至少这种方法不涉及大量的数学公式，也不需手足并用地去解释）就是把它当成是介于开发人员发出的指令和 canvas 显示结果之间的一个修正层 (modification layer)。不管在开发中是否使用变换，修正层始终都是存在的。

修正——在绘制系统中的说法是变换——在应用的时候可以被顺序应用、组合或者随意修改。每个绘制操作的结果显示在 canvas 上之前都要经过修正层去做修正。虽然这么做增加了额外的复杂性，但却为绘制系统添加了更为强大的功能，可以像目前主流图像编辑工具那样支持实时图像处理，所以 API 中这部分内容的复杂性是必要的。

不在代码中调用变换函数并不意味着可以提升 canvas 的性能。canvas 在执行的时候，变换会被呈现引擎隐式调用，这与开发人员是否直接调用无关。在接触最基本的绘制操作之前，提前了解系统背后的原理至关重要。

关于可重用代码有一条重要的建议：一般绘制都应从原点（坐标系中的 0,0 点）开始，应用变换（缩放、平移、旋转等），然后不断修改代码直至达到希望的效果。如图 2-4 所示。

代码清单 2-7 展示了如何使用最简单变换方法——translate 函数。

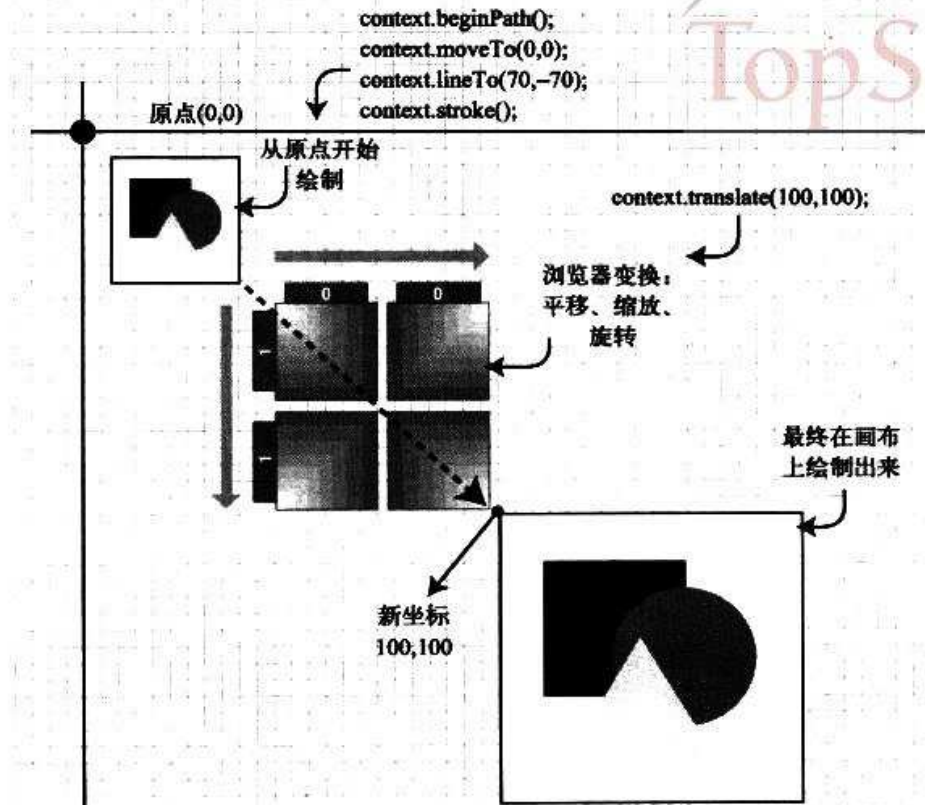


图 2-4 基于原点绘制和变换的示意图

## 代码清单 2-7 用变换的方式在 canvas 上绘制对角线

```

<script>
function drawDiagonal() {
    var canvas = document.getElementById('diagonal');
    var context = canvas.getContext('2d');

    // 保存当前绘图状态
    context.save();

    // 向右下方移动绘图上下文
    context.translate(70, 140);

    // 以原点为起点，绘制与前面相同的线段
    context.beginPath();
    context.moveTo(0, 0);
    context.lineTo(70, -70);
    context.stroke();

    // 恢复原有的绘图状态
    context.restore();
}

window.addEventListener("load", drawDiagonal, true);
</script>

```



我们详细研究一下上面这段通过平移方式绘制对角线的 JavaScript 代码。

(1) 首先, 通过 ID 找到并访问 canvas 对象。(ID 是 diagonal。)

(2) 接着通过调用 canvas 对象的 getContext 函数获取上下文对象。

(3) 接下来, 保存尚未修改的 context, 这样即使进行了绘制和变换操作, 也可以恢复到初始状态。如果不保存, 那么在进行了平移和缩放等操作以后, 其影响会带到后续的操作中, 而这不一定是我们所希望的。在变换前保存 context 状态可以方便以后恢复。

(4) 下一步是在 context 中调用 translate 函数。通过这个操作, 当平移行为发生的时候, 我们提供的变换坐标会被加到结果坐标(对角线)上, 结果就是将要绘制的对角线移动到了新的位置上。不过, 对角线呈现在 canvas 上是在绘制操作结束之后。

(5) 应用平移后, 就可以使用普通的绘制操作来画对角线了。代码清单中调用了三个函数来绘制对角线——beginPath、moveTo 以及 lineTo。绘制的起点是原点 (0,0), 而非坐标点 (70,140)。

(6) 在线条勾画出来之后, 可以通过调用 context.stroke() 函数将其显示在 canvas 上。

(7) 最后, 恢复 context 至原始状态, 这样后续的 canvas 操作就不会被刚才的平移操作影响了。图 2-5 显示了用这段代码绘制的对角线。

虽然新绘制的对角线看起来跟前面的一模一样, 但这次绘制使用了强大的变换功能。学习完本章接下来的内容, 就会明白变换的强大之处。

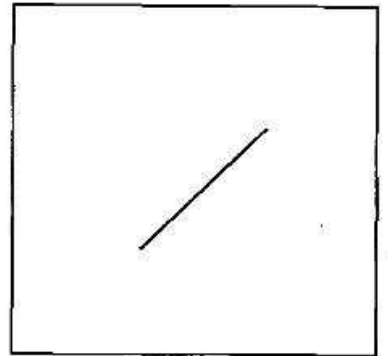


图 2-5 canvas 上平移过的对角线

## 2.2.4 路径

关于绘制线条, 我们还能提供很多有创意的方法。不过, 现在应该进一步学习稍复杂点的图形: 路径。HTML5 Canvas API 中的路径代表你希望呈现的任何形状。本章对角线示例就是一条路径, 你可能已经注意到了, 代码中调用 beginPath 就说明是要开始绘制路径了。实际上, 路径可以要多复杂有多复杂: 多条线、曲线段, 甚至是子路径。如果想在 canvas 上绘制任意形状, 那么你需要重点关注路径 API。

按照惯例, 不论开始绘制何种图形, 第一个需要调用的就是 beginPath。这个简单的函数不带任何参数, 它用来通知 canvas 将要开始绘制一个新的图形了。对于 canvas 来说, beginPath 函数最大的用处是 canvas 需要据此来计算图形的内部和外部范围, 以便完成后续的描边和填充。

路径会跟踪当前坐标, 默认值是原点。canvas 本身也跟踪当前坐标, 不过可以通过绘制代码来修改。

调用了 beginPath 之后, 就可以使用 context 的各种方法来绘制想要的形状了。到目前为止, 我们已经用到了几个简单的 context 路径函数。

□ moveTo(x, y): 不绘制, 只是将当前位置移动到新的目标坐标(x, y)。

□ lineTo(x, y): 不仅将当前位置移动到新的目标坐标(x, y), 而且在两个坐标之间画一条直线。

简而言之，上面两个函数的区别在于：`moveTo` 就像是提起画笔，移动到新位置，而 `lineTo` 告诉 canvas 用画笔从纸上的旧坐标画条直线到新坐标。不过，再次提醒一下，不管调用它们哪一个，都不会真正画出图形，因为我们还没有调用 `stroke` 或者 `fill` 函数。目前，我们只是在定义路径的位置，以便后面绘制时使用。

下一个特殊的路径函数叫做 `closePath`。这个函数的行为同 `lineTo` 很像，唯一的差别在于 `closePath` 会将路径的起始坐标自动作为目标坐标。`closePath` 还会通知 canvas 当前绘制的图形已经闭合或者形成了完全封闭的区域，这对将来的填充和描边都非常有用。

此时，可以在已有的路径中继续创建其他的子路径，或者随时调用 `beginPath` 重新绘制新路径并完全清除之前的所有路径。

跟了解所有复杂系统一样，最好的方式还是实践。现在，我们先不管那些线条的例子，使用 HTML5 Canvas API 开始创建一个新场景——带有长跑跑道的树林。权且把这个图案当成是我们长跑比赛的标志吧。同其他的画图方式一样，我们将从基本元素开始。在这幅图中松树的树冠最简单。代码清单 2-8 演示了如何在 canvas 上绘制一颗松树的树冠。

#### 代码清单 2-8 用于绘制树冠轮廓的函数

```
function createCanopyPath(context) {  
    // 绘制树冠  
    context.beginPath();  
  
    context.moveTo(-25, -50);  
    context.lineTo(-10, -80);  
    context.lineTo(-20, -80);  
    context.lineTo(-5, -110);  
    context.lineTo(-15, -110);  
  
    // 树的顶点  
    context.lineTo(0, -140);  
  
    context.lineTo(15, -110);  
    context.lineTo(5, -110);  
    context.lineTo(20, -80);  
    context.lineTo(10, -80);  
    context.lineTo(25, -50);  
  
    // 连接起点，闭合路径  
    context.closePath();  
}
```

从上面的代码中可以看到，我们用到的仍然是前面用过的移动和画线命令，只不过调用次数多了一些。这些线条表现的是树冠的轮廓，最后我们闭合了路径。我们为这棵树的底部留出了足够的空间，后面几节将在这里的空白处画上树干。代码清单 2-9 演示如何使用树冠绘制函数将树的简单轮廓呈现到 canvas 上。

#### 代码清单 2-9 在 canvas 上画树的函数

```
function drawTrails() {
```



```
var canvas = document.getElementById('trails');
var context = canvas.getContext('2d');

context.save();
context.translate(130, 250);

// 创建表现树冠的路径
createCanopyPath(context);

// 绘制当前路径
context.stroke();
context.restore();
}
```

这段代码中所有的调用想必大家已经很熟悉了。先获取 canvas 的上下文对象，保存以便后续使用，将当前位置变换到新位置，画树冠，绘制到 canvas 上，最后恢复上下文的初始状态。图 2-6 展示了我们的绘画技艺，一条简单的闭合路径表现了树冠。以后我们会详细扩展这段代码，现在算是一个好的开始。

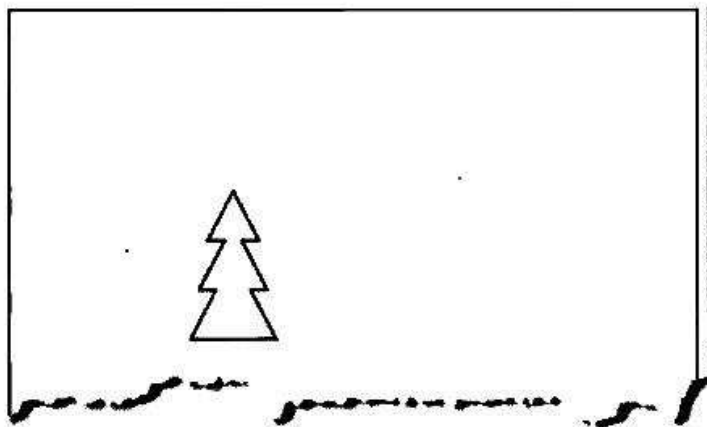


图 2-6 表现树冠的简单路径

## 2.2.5 描边样式

如果开发人员只能绘制直线，而且只能使用黑色，HTML5 Canvas API 就不会如此强大和流行。下面我们就使用描边样式让树冠看起来更像是树。代码清单 2-10 展示了一些基本命令，其功能是通过修改 context 的属性，让绘制的图形更好看。

### 代码清单 2-10 使用描边样式

```
// 加宽线条
context.lineWidth = 4;

// 平滑路径的接合点
context.lineJoin = 'round';

// 将颜色改成棕色
```

```
context.strokeStyle = '#663300';
```

```
// 最后，绘制树冠  
context.stroke();
```

设置上面的这些属性可以改变以后将要绘制的图形外观，这个外观起码可以保持到我们将 context 恢复到上一个状态。

首先，我们将线条宽度加粗到 4 像素。

接着，我们将 lineJoin 属性设置为 round，这是修改当前形状中线段的连接方式，让拐角变得更圆滑；也可以把 lineJoin 属性设置成 bevel 或者 miter（相应的 context.miterLimit 值也需要调整）来变换拐角样式。

最后，通过 strokeStyle 属性改变了线条的颜色。在这个例子中，我们使用了 CSS 值来设置颜色，不过在后面几节中，我们将看到 strokeStyle 的值还可以用于生成特殊效果的图案或者渐变色。

还有一个没有用到的属性——lineCap，可以把它的值设置为 butt、square 或者 round，以此来指定线条末端的样式。哦，示例中的线是闭合的，没有端点。图 2-7 就是我们加工过的树冠，与之前扁平的黑线相比，现在是一条更粗、更平滑的棕色线条。

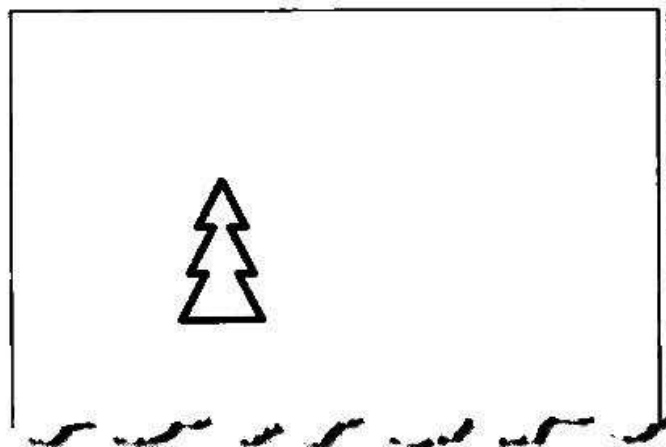


图 2-7 为树冠应用了描边样式

## 2.2.6 填充样式

正如你所期望的那样，能影响 canvas 的图形外观的并非只有描边，另一个常用于修改图形的方法是指定如何填充其路径和子路径。从代码清单 2-11 中可以看到，用宜人的绿色填充树冠有多么简单。

### 代码清单 2-11 使用填充样式

```
// 将填充色设置为绿色并填充树冠  
context.fillStyle = '#339900';  
context.fill();
```



首先，我们将 `fillStyle` 属性设置成合适的颜色。（在后面，我们将看到还可以使用渐变色或者图案填充。）然后，只要调用 `context` 的 `fill` 函数就可以让 `canvas` 对当前图形中所有的闭合路径内部的像素点进行填充。结果如图 2-8 所示。

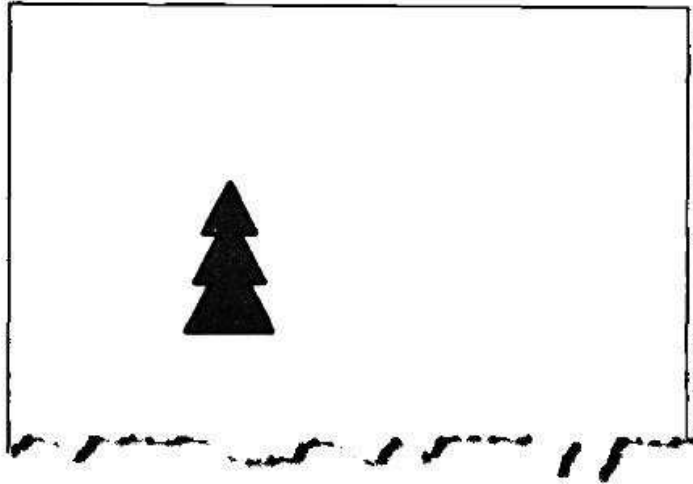


图 2-8 填充后的树冠

由于我们是先描边后填充，因此填充会覆盖一部分描边路径。我们示例中的路径是 4 像素宽，这个宽度是沿路径线居中对齐的，而填充是把路径轮廓内部所有像素全部填充，所以会覆盖描边路径的一半。如果希望看到完整的描边路径，可以在绘制路径（调用 `context.stroke()`）之前填充（调用 `context.fill()`）。

### 2.2.7 填充矩形区域

每棵树都有一个强壮的树干。我们在原始图形中为树干预留了足够的空间。从代码清单 2-12 中可以看到，通过 `fillRect` 函数可以画出树干。

#### 代码清单 2-12 调用 `fillRect` 函数

```
// 将填充色设为棕色
context.fillStyle = '#663300';

// 填充用作树干的矩形区域
context.fillRect(-5, -50, 10, 50);
```

在上面的代码中，再次将棕色作为填充颜色。不过跟上次不一样的是，我们不用 `lineTo` 功能显式画树干的边角，而是使用 `fillRect` 一步到位画出整个树干。调用 `fillRect` 并设置 `x`、`y` 两个位置参数和宽度、高度两个大小参数，随后，`Canvas` 会马上使用当前的样式进行填充。

虽然示例中没有用到，但与之相关的函数还有 `strokeRect` 和 `clearRect`。`strokeRect` 的作用是基于给出的位置和坐标画出矩形的轮廓，`clearRect` 的作用是清除矩形区域内的所有内容并将它恢复到初始状态，即透明色。

“在 HTML5 Canvas API 中，canvas 的清除矩形功能是创建动画和游戏的核心功能。通过反复绘制和清除 canvas 片段，就可能实现动画效果，互联网上有很多这样的例子。但是，如果希望创建运行起来比较流畅的动画，就需要使用剪裁 (clipping)<sup>①</sup> 功能了，有可能还需要二次缓存 canvas，以便最小化由于频繁的清除动作而导致的画面闪烁。本书中不会专门讲解动画，我们鼓励大家自己探索学习。”

——Brain<sup>②</sup>

图 2-9 显示的是基于树冠图形添加的、一次填充的树干。

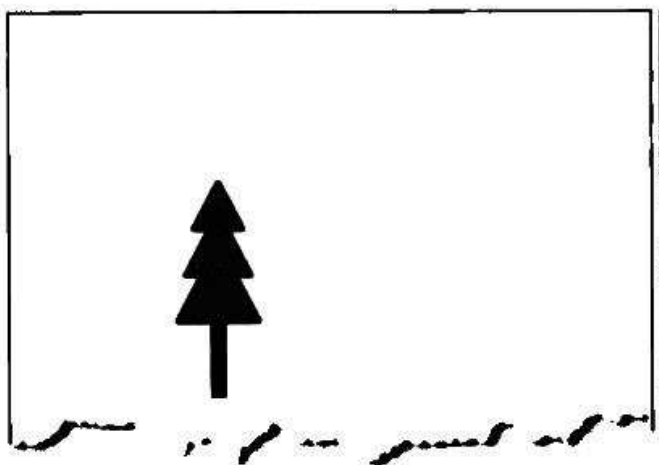


图 2-9 带有矩形树干的树

### 2.2.8 绘制曲线

这个世界，特别是自然界，并不是只有直线和矩形。canvas 提供了一系列绘制曲线的函数。我们将用最简单的曲线函数——二次曲线，来绘制我们的林荫小路。代码清单 2-13 演示了如何添加两条二次曲线。

#### 代码清单 2-13 绘制曲线

```
// 保存 canvas 的状态并绘制路径
context.save();

context.translate(-10, 350);
context.beginPath();
```

① 剪裁 (clipping)：本节没有介绍剪裁功能，此功能常用于创建动画，作者本意是希望读者能够自己探索相关的知识领域。——译者注

② Brain，本书的作者之一。——译者注



```
// 第一条曲线向右上方弯曲
context.moveTo(0, 0);
context.quadraticCurveTo(170, -50, 260, -190);

// 第二条曲线向右下方弯曲
context.quadraticCurveTo(310, -250, 410, -250);

// 使用棕色的粗线条来绘制路径
context.strokeStyle = '#663300';
context.lineWidth = 20;
context.stroke();

// 恢复之前的 canvas 状态
context.restore();
```

跟以前一样，第一步要做的事情是保存当前 canvas 的 context 状态，因为我们即将变换坐标系并修改轮廓设置。要画林荫小路，首先要把坐标恢复到修正层的原点，向右上角画一条曲线。

从图 2-10 中可以看到，quadraticCurveTo 函数绘制曲线的起点是当前坐标，带有两组 (x,y) 参数。第二组是指曲线的终点。第一组代表控制点 (control point)。所谓的控制点位于曲线的旁边 (不是曲线之上)，其作用相当于对曲线产生一个拉力。通过调整控制点的位置，就可以改变曲线的曲率。在右上方再画一条一样的曲线，以形成一条路。然后，像之前描边树冠一样把这条路绘制到 canvas 上 (只是线条更粗了)。

HTML5 Canvas API 的其他曲线功能还涉及 bezierCurveTo、arcTo 和 arc 函数。这些函数通过多种控制点 (如半径、角度等) 让曲线更具可塑性。图 2-11 显示了绘制在 canvas 上的两条曲线，看起来就像是穿过树林的小路一样。

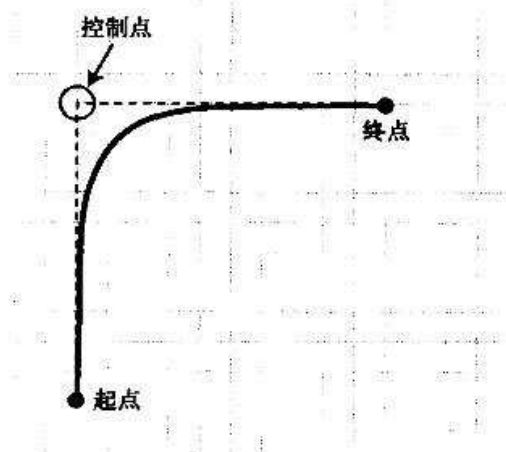


图 2-10 曲线的起点、终点和控制点

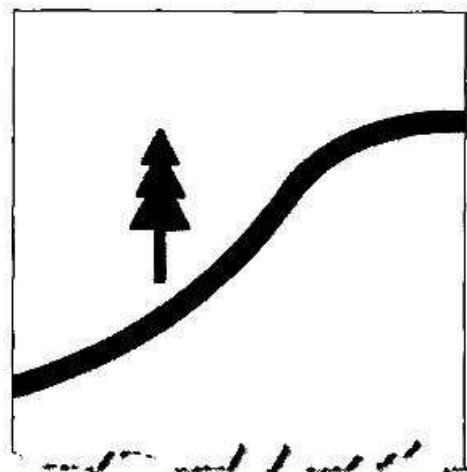


图 2-11 组成小路的曲线

## 2.2.9 在 canvas 中插入图片

在 canvas 中显示图片非常简单。可以通过修正层为图片添加印章、拉伸图片或者修改图片等，

并且图片通常会成为 canvas 上的焦点。用 HTML5 Canvas API 内置的几个简单命令可以轻松地为 canvas 添加图片内容。

不过，图片增加了 canvas 操作的复杂度：必须等到图片完全加载后才能对其进行操作。浏览器通常会在页面脚本执行的同时异步加载图片。如果试图在图片未完全加载之前就将其呈现到 canvas 上，那么 canvas 将不会显示任何图片。因此，开发人员要特别注意，在呈现之前，应确保图片已经加载完毕。

我们的示例将加载一张树皮纹理的图片作为树干以供 canvas 使用。为保证在呈现之前图片已完全加载，我们提供了回调，即仅当图像加载完成时才执行后续代码，如代码清单 2-14 所示。

#### 代码清单 2-14 加载图像

```
// 加载图片 bark.jpg
var bark = new Image();
bark.src = "bark.jpg";

// 图片加载完成后，将其显示在 canvas 上
bark.onload = function () {
    drawTrails();
}
```

从上面的代码中可以看到，我们为 bark.jpg 图片添加了 onload 处理函数，以保证仅在图像加载完成时才调用主 drawTrails 函数。这样做可以保证后续的调用能够把图片正常显示出来，如代码清单 2-15 所示。

#### 代码清单 2-15 在 canvas 上显示图像

```
// 用背景图案填充作为树干的矩形
context.drawImage(bark, -5, -50, 10, 50);
```

在这段代码里，我们用纹理贴图替换了之前调用 fillRect 函数的填充来作为新的树干。尽管替换的动作很小，但 canvas 上显示出来的树干更有质感。注意，在 drawImage 函数中，除了图片本身外，还指定了 x、y、width 和 height 参数。这些参数会对贴图进行调整以适应预定的 10×50 像素树干区域。我们还可以把原图的尺寸传进来，以便在裁切区域内对图片进行更多控制。

在图 2-12 中可以看到，同之前用矩形填充的方式相比，树干的变化不大。

### 2.2.10 渐变

对树干还是不满意？其实我也是。我们使用另一种可以让树干变得稍微好看点的绘制方法：渐变。渐变是指在颜色集上使用逐步抽样算法，并将结果应用于描边样式和填充样式中。使用渐变需要三个步骤：

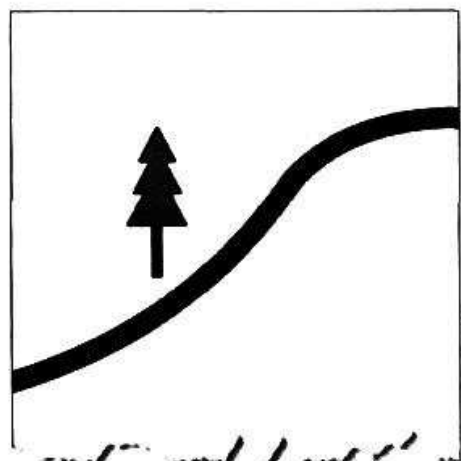


图 2-12 使用了树干贴图的树



- (1) 创建渐变对象;
- (2) 为渐变对象设置颜色, 指明过渡方式;
- (3) 在 context 上为填充样式或者描边样式设置渐变。

可以将渐变看做是颜色沿着一条线进行缓慢地变化。例如, 如果为渐变对象提供了 A、B 两个点, 不论是绘制还是填充, 只要从 A 移动到 B, 都会带来颜色的变化。

要设置显示哪种颜色, 在渐变对象上使用 `addColorStop` 函数即可。这个函数允许指定两个参数: 颜色和偏移量。颜色参数是指开发人员希望在偏移位置描边或填充时所使用的颜色。偏移量是一个 0.0 到 1.0 之间的数值, 代表沿着渐变线渐变的距离有多远。

假如要建立一个从点(0,0)到点(0,100)的渐变, 并指定在 0.0 偏移位置使用白色, 在 1.0 偏移位置使用黑色。当使用绘制或者填充的动作从(0,0)画到(0,100)后, 就可以看到颜色从白色(起始位置)渐渐转变成了黑色(终止位置)。

除了可以变换成其他颜色外, 还可以为颜色设置 alpha 值(例如透明), 并且 alpha 值也是可以变化的。为了达到这样的效果, 需要使用颜色值的另一种表示方法, 例如内置 alpha 组件的 CSS `rgba` 函数。

下面我们通过示例来详细了解如何使用两个渐变来填充(相应的函数为 `fillRect`)矩形区域, 并形成最终的树干, 见代码清单 2-16。

#### 代码清单 2-16 使用渐变

```
// 创建用作树干纹理的三阶水平渐变
var trunkGradient = context.createLinearGradient(-5, -50, 5, -50);

// 树干的左侧边缘是一般程度的棕色
trunkGradient.addColorStop(0, '#663300');

// 树干中间偏左的位置颜色要淡一些
trunkGradient.addColorStop(0.4, '#996600');

// 树干右侧边缘的颜色要深一些
trunkGradient.addColorStop(1, '#552200');

// 使用渐变色填充树干
context.fillStyle = trunkGradient;
context.fillRect(-5, -50, 10, 50);

// 接下来, 创建垂直渐变, 以用作树冠在树干上投影
var canopyShadow = context.createLinearGradient(0, -50, 0, 0);

// 投影渐变的起点是透明度设为 50% 的黑色
canopyShadow.addColorStop(0, 'rgba(0, 0, 0, 0.5)');

// 方向垂直向下, 渐变色在很短的距离内迅速渐变至完全透明, 这段长度之外的树干上没有投影
canopyShadow.addColorStop(0.2, 'rgba(0, 0, 0, 0.0)');

// 在树干上填充投影渐变
```

```
context.fillStyle = canopyShadow;  
context.fillRect(-5, -50, 10, 50);
```

如图 2-13 所示，使用了两个渐变后，最终绘制出来的树干有了平滑的光照效果。现在，树干看起来更平滑，同时树干上也有了轻微的阴影效果。我们把这幅图保存起来吧。



图 2-13 有渐变树干的树

除了我们刚才用到的线性渐变以外，HTML5 Canvas API 还支持放射性渐变，所谓放射性渐变就是颜色会介于两个指定圆间的锥形区域平滑变化。放射性渐变和线性渐变使用的颜色终止点是一样的，不过参数如代码清单 2-17 所示。

#### 代码清单 2-17 使用放射性渐变的示例

```
createRadialGradient(x0, y0, r0, x1, y1, r1)
```

代码中，前三个参数代表以(x0,y0)为圆心，r0为半径的圆，后三个参数代表以(x1,y1)为圆心，r1为半径的另一个圆。渐变会在两个圆中间的区域出现。

### 2.2.11 背景图

直接绘制图像有很多用处，但在某些情况下，像 CSS 那样使用图片作为背景也非常有用。我们已经了解了如何使用加粗的颜色描边和填充。在描边和填充的时候，HTML5 Canvas API 还支持图片平铺。

现在我们把林荫小路变得崎岖一点。这次不再对曲线跑道进行描边，而是使用背景图片填充的方法。为了达到预想的效果，我们将已经作废的树干图片（我们已经有了“渐变”树干）替换成砾石图片。我们将调用 createPattern 函数来替代之前的 drawImage 函数，如代码清单 2-18 所示。



## 代码清单 2-18 使用背景图片

```

// 加载砾石背景图
var gravel = new Image();
gravel.src = "gravel.jpg";
gravel.onload = function () {
    drawTrails();
}

// 用背景图替代棕色粗线条
context.strokeStyle = context.createPattern(gravel, 'repeat');
context.lineWidth = 20;
context.stroke();

```

从上面的代码中可以看到，绘制的时候还是使用 `stroke()` 函数，只不过这次我们先设置了 `context` 上的 `strokeStyle` 属性，把调用 `context.createPattern` 的返回值赋给该属性。再次强调一下，图片必须提前加载完毕，以便 `canvas` 执行后续操作。`context.createPattern` 的第二个参数是重复性标记，可以在表 2-2 中选择合适的值。

表2-2 重复性参数

平铺方式	意义
repeat	(默认值) 图片会在两个方向平铺
repeat-x	横向平铺
repeat-y	纵向平铺
no-repeat	图片只显示一次，不平铺

图 2-14 显示了应用背景图片方式画出的小路。

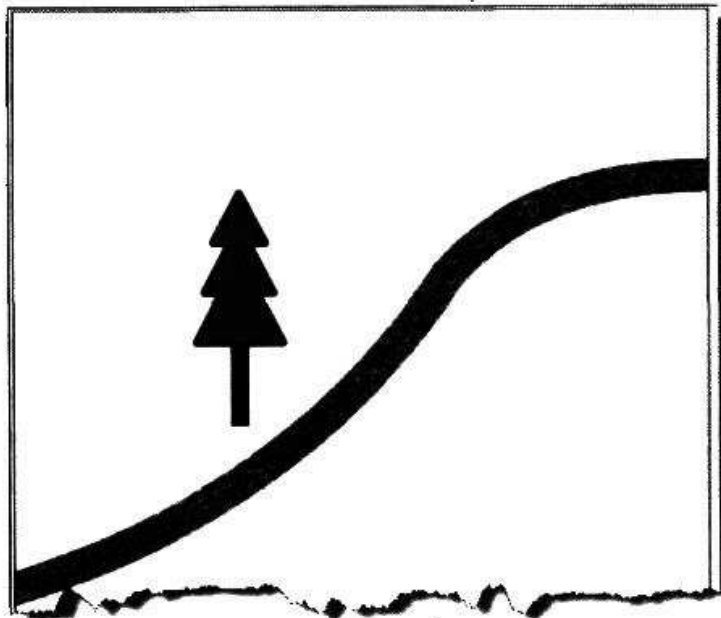


图 2-14 使用平铺图片作为背景的小路

## 2.2.12 缩放 canvas 对象

树林里怎么可能只有一棵树呢？现在我们来解决这个问题。为简单起见，我们计划把示例代码中用于绘制树的操作独立出来，当做一个单独的例程，称为 drawTree，见代码清单 2-19。

代码清单 2-19 绘制树对象的函数

```
// 创建树对象绘制函数，以便重用
function drawTree(context) {
    var trunkGradient = context.createLinearGradient(-5, -50, 5, -50);
    trunkGradient.addColorStop(0, '#663300');
    trunkGradient.addColorStop(0.4, '#996600');
    trunkGradient.addColorStop(1, '#552200');
    context.fillStyle = trunkGradient;
    context.fillRect(-5, -50, 10, 50);

    var canopyShadow = context.createLinearGradient(0, -50, 0, 0);
    canopyShadow.addColorStop(0, 'rgba(0, 0, 0, 0.5)');
    canopyShadow.addColorStop(0.2, 'rgba(0, 0, 0, 0.0)');
    context.fillStyle = canopyShadow;
    context.fillRect(-5, -50, 10, 50);

    createCanopyPath(context);
    context.lineWidth = 4;
    context.lineJoin = 'round';
    context.strokeStyle = '#663300';
    context.stroke();

    context.fillStyle = '#339900';
    context.fill();
}
```

可以看到，drawTree 函数包括了之前绘制树冠、树干和树干渐变的所有代码。为了在新的位置画出大一点的树，我们将使用另一种变换方式——缩放函数 context.scale，如代码清单 2-20 所示。

代码清单 2-20 绘制树对象

```
// 在(130, 250)的位置绘制第一棵树
context.save();
context.translate(130, 250);
drawTree(context);
context.restore();

// 在(260, 500)的位置绘制第二棵树
context.save();
context.translate(260, 500);

// 将第二棵树的宽高分别放大至原来的 2 倍
context.scale(2, 2);
drawTree(context);
context.restore();
```



scale 函数带有两个参数来分别代表在  $x$ 、 $y$  两个维度的值。每个参数在 canvas 显示图像的时候，向其传递在本方向轴上图像要放大（或者缩小）的量。如果  $x$  值为 2，就代表所绘制图像中全部元素都会变成两倍宽，如果  $y$  值为 0.5，绘制出来的图像全部元素都会变成之前的一半高。使用这些函数，就可以很方便的在 canvas 上创建出新的树，如图 2-15 所示。

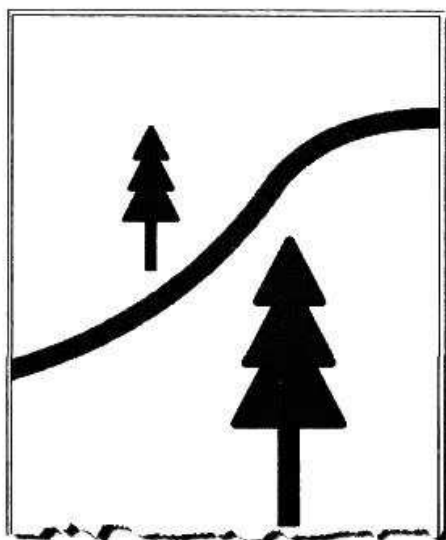


图 2-15 放大后的树

#### 始终在 origin 执行图形和路径的变换操作

“示例中演示了为什么要在 origin 执行图形和路径的变换操作，执行完后再统一平移。原因就是缩放 (scale) 和旋转 (rotate) 等变换操作都是针对 origin 进行的。

如果对一个不在 origin 的图形进行旋转变换，那么 rotate 变换函数会将图形绕着 origin 旋转而不是在原地旋转。与之类似，如果进行缩放操作时没有将图形放置到合适的坐标上，那么所有路径坐标都会被同时缩放。取决于缩放比例的大小，新的坐标可能会全部超出 canvas 范围，进而给开发人员带来困惑，为什么我的缩放操作会把图像删了？”

——Brian

### 2.2.13 Canvas 变换

变换操作并不限于缩放和平移，我们可以使用函数 `context.rotate(angle)` 来旋转图像，甚至可以直接修改底层变换矩阵以完成一些高级操作，如剪裁图像的绘制路径。如果想旋转图像，只需执行代码清单 2-21 所示的一系列操作即可。

#### 代码清单 2-21 旋转图像

```
context.save();
```

```
// 旋转角度参数以弧度为单位
context.rotate(1.57);
context.drawImage(myImage, 0, 0, 100, 100);

context.restore();
```

在代码清单 2-22 中，我们将演示如何对路径坐标进行随意变换，以从根本上改变现有树的路径显示，并最终创建一个阴影效果。

2

### 代码清单 2-22 一种变换的使用方法

```
// 创建用于填充树干的三阶水平渐变色
// 保存 canvas 的当前状态
context.save();

// X 值随着 Y 值的增加而增加，借助拉伸变换，可以创建一棵用作阴影的倾斜的树
context.transform(1, 0, -0.5, 1, 0, 0);

// 在 Y 轴方向，将阴影的高度压缩为原来的 60%
context.scale(1, 0.6);

// 使用透明度为 20% 的黑色填充树干
context.fillStyle = 'rgba(0, 0, 0, 0.2)';
context.fillRect(-5, -50, 10, 50);

// 使用已有的阴影效果重新绘制树
createCanopyPath(context);
context.fill();

// 恢复之前的 canvas 状态
context.restore();
```

你可以像上面那样直接修改 context 变换矩阵，前提是要熟悉二维绘图系统中的矩阵变换。分析这种变换背后的数学含义，可以看出我们通过调整与 Y 轴值相对应的参数改变了 X 轴的值，这样做目的是为了拉伸出一棵灰色的树做阴影。接下来，我们按照 60% 的比例将剪裁出的树缩小到了合适的尺寸。

注意，剪裁过的“阴影”树会先被显示出来，这样一来，真正的树就会按照 Z 轴顺序（canvas 中对象的重叠顺序）显示在阴影的上面。此外，树影的填充用到了 CSS 的 RGBA 特性，通过特性我们将透明度值设为正常情况下的 20%。至此，带有半透明效果的树影就做好了。将其应用于已经缩放过的树上，效果如图 2-16 所示。

#### 2.2.14 Canvas 文本

在作品即将完成之际，我们要在图像的上部添加一个别

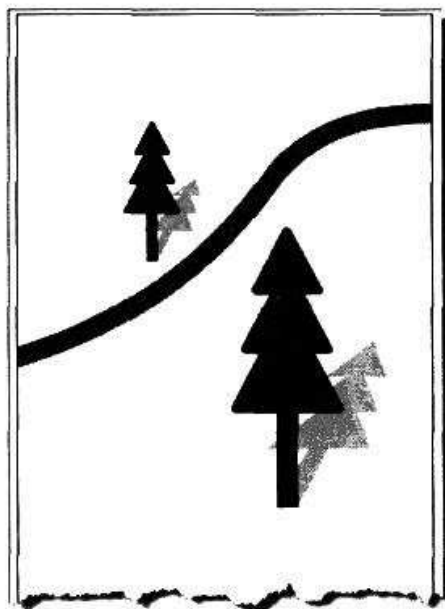


图 2-16 带有变换阴影的树



致的标题，以此来向大家演示 HTML5 Canvas API 强大的文本功能。需要特别注意的是，操作 canvas 文本的方式与操作其他路径对象的方式相同：可以描绘文本轮廓和填充文本内部，同时，所有能够应用于其他图形的变换和样式都能用于文本。

context 对象的文本绘制功能由两个函数组成：

- fillText(text,x,y,maxwidth)
- strokeText(text,x,y,maxwidth)

两个函数的参数完全相同，必选参数包括文本参数以及用于指定文本位置的坐标参数。maxwidth 是可选参数，用于限制字体大小，它会将文本字体强制收缩到指定尺寸。此外，还有一个 measureText 函数可供使用，该函数会返回一个度量对象，其中包含了在当前 context 环境下指定文本的实际显示宽度。

为了保证文本在各浏览器下都能正常显示，Canvas API 为 context 提供了类似于 CSS 的属性，以此来保证实际显示效果的高度可配置。如表 2-3。

表2-3 文本呈现相关的context属性

属 性	值	备 注
font	CSS字体字符串	例如: italic Arial, scan-serif
textAlign	start、end、left、right、center	默认是start
textBaseline	top、hanging、middle、alphabetic、ideographic、bottom	默认是alphabetic

对上面这些 context 属性赋值能够改变 context，而访问 context 属性可以查询到其当前值。在代码清单 2-23 中，我们首先创建了一段使用 Impact 字体的大号文本，然后使用已有的树皮图片作为背景进行填充。为了将文本置于 canvas 的上方并居中，我们定义了最大宽度和 center（居中）对齐方式。

#### 代码清单 2-23 使用 canvas 文本

```
// 在 canvas 上绘制标题文本
context.save();

// 字号为 60px，字体为 impact
context.font = "60px impact";

// 将文本填充为棕色
context.fillStyle = '#996600';
// 将文本设为居中对齐
context.textAlign = 'center';

// 在 canvas 顶部中央的位置，以大字体的形式显示文本
context.fillText('Happy Trails!', 200, 60, 400);
context.restore();
```

结果如图 2-17 所示，林间小路的美景马上就增添了几分欢快的味道。

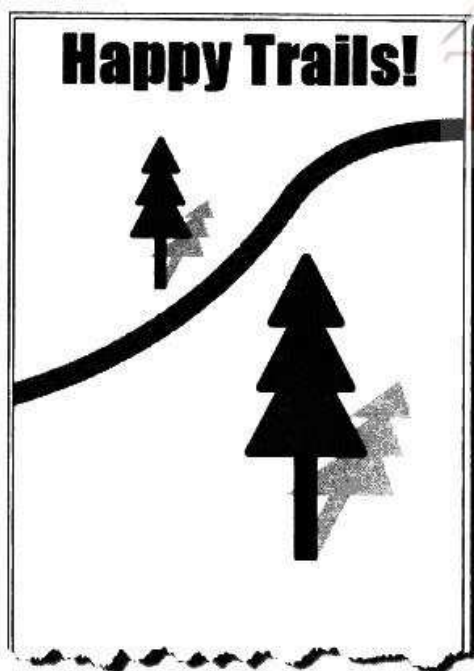


图 2-17 使用背景图片填充的文本

### 2.2.15 应用阴影

最后，我们将使用内置的 Canvas Shadow API 为文本添加模糊阴影效果。虽然我们能够通过 HTML5 Canvas API 将阴影效果应用于之前执行的任何操作中，但与很多图形效果的应用类似，阴影效果的使用也要把握好“度”。

可以通过几种全局 context 属性来控制阴影，见表 2-4。

表 2-4 阴影属性

属 性	值	备 注
shadowColor	任何 CSS 中的颜色值	可以使用透明度 (alpha)
ShadowOffsetX	像素值	值为正数，向右移动阴影；值为负数，向左移动阴影
shadowOffsetY	像素值	值为正数，向下移动阴影；值为负数，向上移动阴影
shadowBlur	高斯模糊值	值越大，阴影边缘越模糊

shadowColor 或者其他任意一项属性的值被赋为非默认值时，路径、文本和图片上的阴影效果就会被触发。代码清单 2-24 显示了如何为文本添加阴影效果。

#### 代码清单 2-24 应用阴影效果

```
// 设置文字阴影的颜色为黑色，透明度为 20%
context.shadowColor = 'rgba(0, 0, 0, 0.2)';

// 将阴影向右移动 15px，向上移动 10px
context.shadowOffsetX = 15;
```



```
context.shadowOffsetY = -10;
```

```
// 轻微模糊阴影  
context.shadowBlur = 2;
```

执行上述代码后，canvas 渲染器会自动应用阴影效果，直到恢复 canvas 状态或者重置阴影属性。添加阴影后的效果如图 2-18 所示。

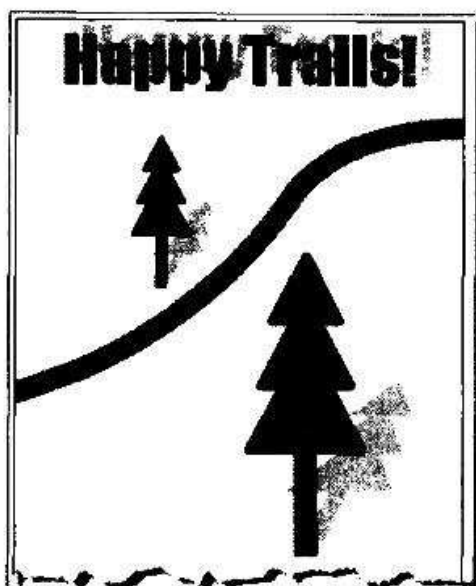


图 2-18 有阴影效果的标题

如你所见，由 CSS 生成的阴影只有位置上的变化，而无法与变换生成的阴影（树影）保持同步。为了一致起见，在 canvas 上绘制阴影时，应该尽量只用一种方法。

### 2.2.16 像素数据

Canvas API 最有用的特性之一是允许开发人员直接访问 canvas 底层像素数据。这种数据访问是双向的：一方面，可以以数值数组形式获取像素数据；另一方面，可以修改数组的值以将其应用于 canvas。实际上，放弃本章之前讨论的渲染调用，也可以通过直接调用像素数据的相关方法来控制 canvas。这要归功于 context API 内置的三个函数。

第一个是 `context.getImageData(sx, sy, sw, sh)`。这个函数返回当前 canvas 状态并以数值数组的方式显示。具体来说，返回的对象包括三个属性。

- width：每行有多少个像素。
- height：每列有多少个像素。
- data：一维数组，存有从 canvas 获取的每个像素的 RGBA 值。该数组为每个像素保存了四个值——红、绿、蓝和 alpha 透明度。每个值都在 0 到 255 之间。因此，canvas 上的每个像素在这个数组中就变成了四个整数值。数组的填充顺序是从左到右，从上到下（也就是先第一行再第二行，依此类推），如图 2-19 所示。

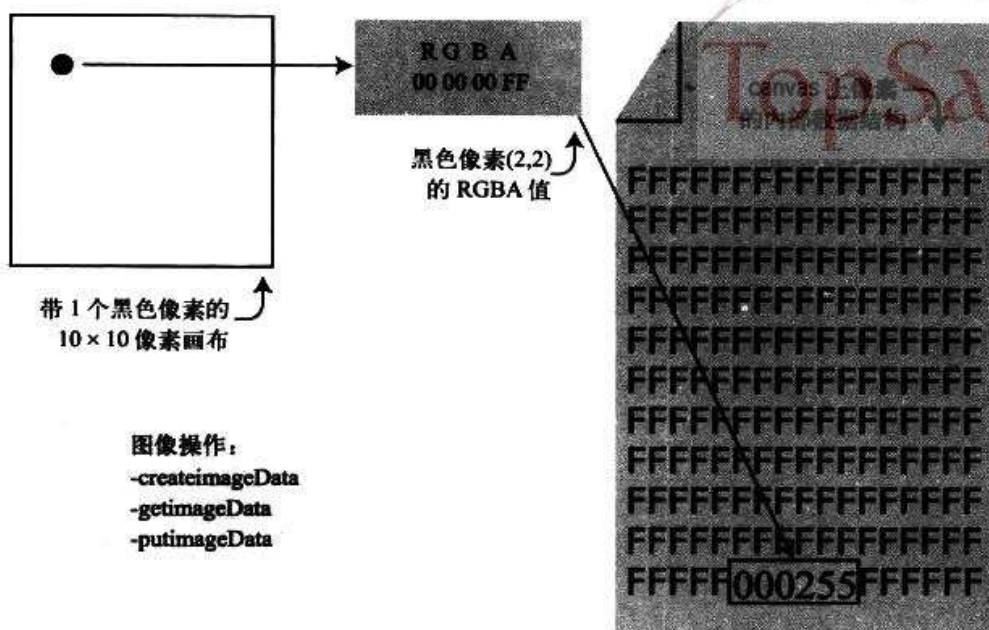


图 2-19 像素数据及其内部数据结构

`getImageData` 函数有四个参数，该函数只返回这四个参数所限定的区域内的数据。只有被 `x`、`y`、`width` 和 `height` 四个参数框定的矩形区域内的 `canvas` 上的像素才会被取到，因此要想获取所有像素数据，就需要这样传入参数：`getImageData(0, 0, canvas.width, canvas.height)`。

因为每个像素由四个图像数据表示，所以要计算指定的像素点对应的值是什么就有点头疼。不要紧，下面有公式。

在给定了 `width` 和 `height` 的 `canvas` 上，在坐标 `(x, y)` 上的像素的构成如下。

- 红色部分： $((width * y) + x) * 4$
- 绿色部分： $((width * y) + x) * 4 + 1$
- 蓝色部分： $((width * y) + x) * 4 + 2$
- 透明度部分： $((width * y) + x) * 4 + 3$

一旦可以通过像素数据的方式访问对象，就可以通过数学方式轻松修改数组中的像素值，因为这些值都是从 0 到 255 的简单数字。修改了任何像素的红、绿、蓝和 `alpha` 值之后，可以通过第二个函数来更新 `canvas` 上的显示，那就是 `context.putImageData(imagedata, dx, dy)`。

`putImageData` 允许开发人员传入一组图像数据，其格式与最初从 `canvas` 上获取来的是一样的。这个函数使用起来非常方便，因为可以直接用从 `canvas` 上获取数据加以修改然后返回。一旦这个函数被调用，所有新传入的图像数据值就会立即在 `canvas` 上更新显示出来。`dx` 和 `dy` 参数可以用来指定偏移量，如果使用，则该函数就会跳到指定的 `canvas` 位置去更新显示传进来的像素数据。

最后，如果想预先生成一组空的 `canvas` 数据，则可调用 `context.createImageData(sw, sh)`，这个函数可以创建一组图像数据并绑定在 `canvas` 对象上。这组数据可以像先前那样处理，只是在获取 `canvas` 数据时，这组图像数据不一定会反映 `canvas` 的当前状态。



### 2.2.17 Canvas 的安全机制

上面讨论了直接操纵像素数据的方法，在这里有必要重点提醒一下，大多数开发者都会合法使用像素数据操作。尽管如此，还是会有人出于某些邪恶的目的利用这种从 canvas 直接获取并且修改数据的能力。出于这个原因，origin-clean canvas 的概念应运而生，换句话说，如果 canvas 中的图片并非来自包含它的页面所在的域，页面中的脚本将不能取得其中的数据。

如图 2-20 所示，如果来自 <http://www.example.com> 的页面包含 canvas 元素，那么页面中的代码完全有可能在 canvas 里面呈现来自 <http://www.remote.com> 的图片。毕竟在任何 Web 页面中显示其他远程网站的图片都是完全可接受的。

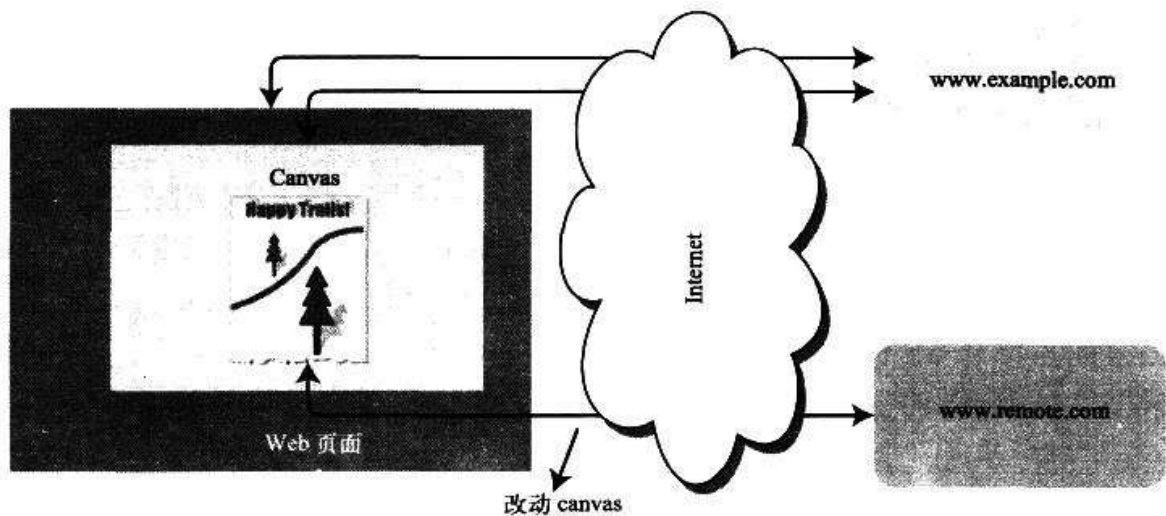


图 2-20 本地和远程图像来源

然而，在没有 Canvas API 以前，无法使用编程的方式获取下载图片的像素信息。来自其他网站的私有图片可以显示在本地，但无法被读取或者复制。如果允许脚本读取本地之外的图像数据，那么整个网络中的用户照片以及其他敏感的在线图片文档将被“无限制地共享”。

为了避免如此，在 `getImageData` 函数被调用的时候，如果 canvas 中的图像来自其他域，就会抛出安全异常。这样的话，只要不获取显示着其他域中图片的 canvas 的数据，那么就可以随意呈现这些远程图片。在开发的过程中要注意这个限制条件，使用安全的渲染方式。

## 2.3 使用 HTML5 Canvas 创建应用

使用 Canvas API 可以创建许多种应用：图形、图表、图片编辑等，然而最奇妙的一个应用是修改或者覆盖已有内容。最流行的覆盖图被称为热点图。虽然热点图听起来是度量温度的意思，不过这里的热度可以用于任何可测量的活动。地图上活跃程度高的部分使用暖色标记（例如红色、黄色或白色），活跃程度低的部分不显示颜色变化，或者显示浅浅的黑色或灰色。

举个例子，热点图可以用在城市地图上来标记交通路况，或者在世界地图上显示风暴的活动

情况。在 HTML5 中这些应用都非常容易实现，只需要将 canvas 叠放在地图上显示即可。实际上就是用 canvas 覆盖地图，然后再基于相应的活动数据绘制出不同的热度级别。

现在，我们使用已经学过的 Canvas API 知识来绘制一个简单的热点图。这个示例中，热度数据不是来源于外部，而是来源于我们的鼠标在地图上的移动情况。鼠标移动到某个区域，会使这个区域的“热度”增加。将鼠标放在特定区域不动会让该区域“温度”迅速增长至极限。为了示范，我们将在一个“难以名状”的地图上进行热点图的覆盖演示（见图 2-21）。

热点图

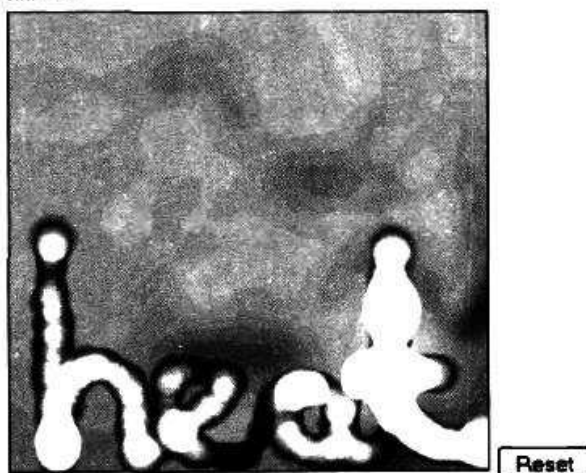


图 2-21 热点图应用

上面看到的是热点图应用的最终效果，下面我们深入分析实现代码。和往常一样，你可以在线查看或下载示例的源代码。

这份源码中，我们从 HTML 元素开始分析。为了演示效果，这个 HTML 中只包含了标题（Heatmap）、画布和按钮（Reset，用来复位热点图）。canvas 上显示的背景图片文件是 mapbg.jpg，通过代码清单 2-25 中的 CSS 代码应用到 canvas 中。

#### 代码清单 2-25 热点图的 canvas 元素

```
<style type="text/css">
  #heatmap {
    background-image: url("mapbg.jpg");
  }
</style>

<h2>Heatmap </h2>
<canvas id="heatmap" class="clear" style="border: 1px solid ; " height="300"
width="300"> </canvas>
<button id="resetButton">Reset</button>
```

我们还声明了一些变量，然后对其进行了初始化，以备后用。



```
var points = {};  
var SCALE = 3;  
var x = -1;  
var y = -1;
```

接下来，为了支持全局绘制操作，我们将为 canvas 设置一个高透明值，并且设置为混合模式，让新的绘制操作点亮底层的像素而不是替换它们。

然后，如代码清单 2-26 所示，我们会设置 addToPoint 函数，在鼠标移动的时候或者每隔 1/10 s 的时间调用它以改变显示效果。

代码清单 2-26 loadDemo 函数

```
function loadDemo() {  
    document.getElementById("resetButton").onclick = reset;  
  
    canvas = document.getElementById("heatmap");  
    context = canvas.getContext('2d');  
    context.globalAlpha = 0.2;  
    context.globalCompositeOperation = "lighter"  
  
    function sample() {  
        if (x != -1) {  
            addToPoint(x,y)  
        }  
        setTimeout(sample, 100);  
    }  
  
    canvas.onmousemove = function(e) {  
        x = e.clientX - e.target.offsetLeft;  
        y = e.clientY - e.target.offsetTop;  
        addToPoint(x,y);  
    }  
  
    sample();  
}
```

使用 canvas 的 clearRect 函数，就可以让用户在点击 Reset 按钮的时候，将整个 canvas 区域清空并重置回原始状态（见代码清单 2-27）。

代码清单 2-27 reset 函数

```
function reset() {  
    points = {};  
    context.clearRect(0,0,300,300);  
    x = -1;  
    y = -1;  
}
```

接下来我们建立一张颜色查找表，以便在 canvas 上执行绘制操作的时候使用。代码清单 2-28 中列出了颜色亮度由低到高的范围，不同的颜色值会被用来代表各种不同的热度。intensity 的值越大，返回的颜色越亮。

### 代码清单 2-28 getColor 函数

```
function getColor(intensity) {  
    var colors = ["#072933", "#2E4045", "#8C593B", "#B2814E", "#FAC268", "#FAD237"];  
    return colors[Math.floor(intensity/2)];  
}
```

不管什么时候，只要鼠标移过或者悬停在 canvas 的某个区域，就会有一个点被绘制出来。鼠标在特定区域中停留的时间越长，这个点就越大（同时越亮）。像代码清单 2-29 中所示，使用 context.arc 函数根据特定的半径值绘制圆，通过传到 getColor 函数中的半径值来判断，半径越大画出的圆越亮、颜色越热。

### 代码清单 2-29 drawPoint 函数

```
function drawPoint(x, y, radius) {  
    context.fillStyle = getColor(radius);  
    radius = Math.sqrt(radius)*6;  
  
    context.beginPath();  
    context.arc(x, y, radius, 0, Math.PI*2, true);  
  
    context.closePath();  
    context.fill();  
}
```

在 addToPoint 函数中（每次鼠标移动或者悬停的时候都会调用这个函数），canvas 特定点上的热度值会升高并保存下来。代码清单 2-30 中显示了最高的热度值是 10。给定像素点的当前热度值一旦被检测到，那么相应的像素以及相关的热度、半径值就会被传递到 drawPoint 函数中。

### 代码清单 2-30 addToPoint 函数

```
function addToPoint(x, y) {  
    x = Math.floor(x/SCALE);  
    y = Math.floor(y/SCALE);  
  
    if (!points[[x,y]]) {  
        points[[x,y]] = 1;  
    } else if (points[[x,y]]==10) {  
        return  
    } else {  
        points[[x,y]]++;  
    }  
    drawPoint(x*SCALE,y*SCALE, points[[x,y]]);  
}
```

最后，还注册了一个 loadDemo 函数，用来在窗口加载完毕的时候调用。

```
window.addEventListener("load", loadDemo, true);
```

总而言之，这些 100 行左右的代码可以向大家证明：使用 HTML5 Canvas API，在短短的时间内，不用任何插件或者外部技术就可以实现非常高级的功能。另外，我们身边的各种数据源又是无穷无尽的，既然将它们可视化如此简单方便，那我们还等什么呢？



## 进阶功能之全页玻璃窗

在前面的示例中，我们看到了如何把 canvas 应用于图片上。其实，我们还可以把 canvas 应用于整个浏览器窗口或者其中的一部分之上——这种技术通常被称作“玻璃窗”（glass pane）。在 Web 页面中放置玻璃窗后，我们可以做很多之前意想不到的事情。

例如，可以编写函数来获取页面中所有 DOM 元素的绝对位置，然后创建循序渐进的帮助功能，从而引导 Web 应用的用户，一步一步地教他们学会操作。

另外，可以借助 canvas 玻璃窗并利用鼠标事件让用户在 Web 页面上绘制反馈。不过，使用此功能时，请记住以下三点。

- 需要将 canvas 的 CSS 属性 position 设置为 absolute，并且指定 canvas 的位置、宽度和高度。如果没有明确的宽度和高度值，那么 canvas 将保持默认尺寸——0 像素。
- 别忘了将 canvas 的 CSS 属性 z-index 的值设置得大一些，使其能够盖在所有显示内容的上面。如果 canvas 被其他内容覆盖在最下面，就毫无用武之地了。
- 设置 canvas 玻璃窗会阻塞后续的事件访问，因此需要提醒开发人员，不需要时要记得“关闭窗口”。

## 2.4 小结

到目前为止，我们看到了 HTML5 Canvas API 提供的强大功能，利用它可以直接修改 Web 应用的外观，而不必再像以前那样借助于各种第三方技术。HTML5 Canvas API 还可以通过自由组合图像、渐变和复杂路径等方式来创建你能想到的几乎所有效果。需要注意的是，绘制工作通常应以原点为起点，在展现图像之前要先完成加载，而在使用外部来源的图片时则要留心。如果能学会驾驭 canvas，那么你就能在网页上创建出前所未见的应用。

**本章**，我们将探索 HTML5 的两个重要元素——audio 和 video，并演示如何利用它们创建引人注目的应用。audio 和 video 元素的出现让 HTML5 的媒体应用多了新选择，开发人员不必使用插件就能播放音频和视频。对于这两个元素，HTML5 规范提供了通用、完整、可脚本化控制的 API。

首先，我们会讨论音频 (audio) 和视频 (video) 的容器文件和编解码器，以及为什么支持这些编解码器。接着将分析没有支持哪些常用的编解码器，这种缺陷也是使用这两个媒体元素的最大弊端。不过，我们也会讨论将来有可能怎样解决这个问题。此外，我们还会演示 HTML5 Audio 和 Video<sup>①</sup>的一种切换机制，通过该机制，可以让浏览器在所列媒体文档中挑选最合适的文件类型进行播放。

然后，我们会演示如何通过 API 编程的方式来控制页面中的音频和视频。最后，我们会探讨 HTML5 Audio 和 Video 在实际中的应用。

## 3.1 HTML5 Audio 和 Video 概述

本节，我们要讨论与 HTML5 audio 元素和 video 元素相关的两个关键概念：容器 (container) 和编解码器 (codec)。

### 3.1.1 视频容器

不论是音频文件还是视频文件，实际上都只是一个容器文件，这点类似于压缩了一组文件的 ZIP 文件。从图 3-1 可以看出，视频文件 (视频容器) 包含了音频轨道、视频轨道和其他一些元数据。视频播放的时候，音频轨道和视频轨道是绑定在一起的。元数据部分包含了该视频的封面、标题、子标题、字幕等相关信息。

主流视频容器支持如下视频格式：

- Audio Video Interleave (.avi)
- Flash Video (.flv)

<sup>①</sup> 本章中，HTML5 Audio 和 Video 表示 HTML5 Audio 和 Video 规范的简写，该规范中最核心的内容就是 audio 元素和 video 元素，以及与之相关的控制机制。——译者注



- MPEG 4 (.mp4)
- Matroska (.mkv)
- Ogg (.ogv)

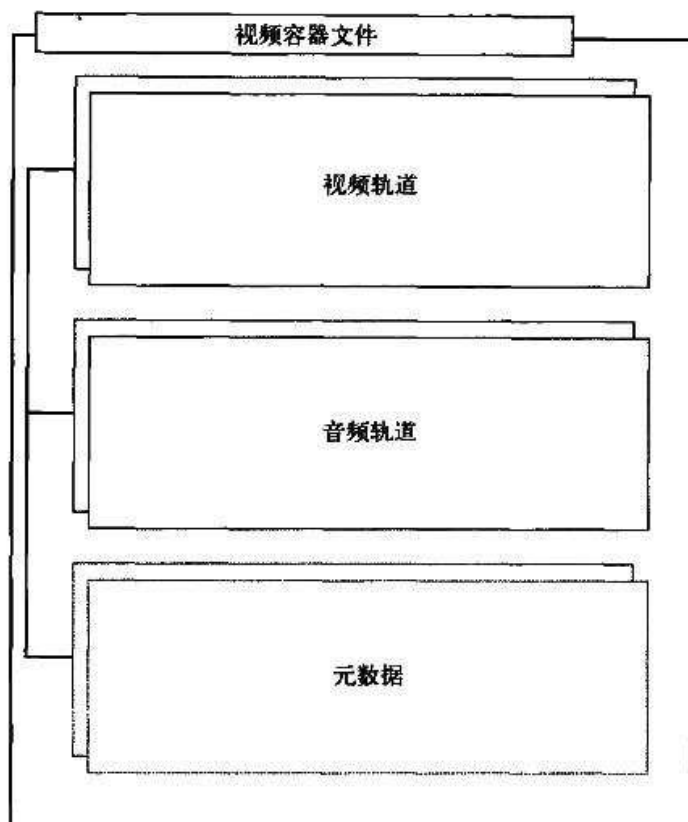


图 3-1 视频容器概貌

### 3.1.2 音频和视频编解码器

音频和视频的编码/解码器是一组算法，用来对一段特定音频或视频流进行解码和编码，以便音频和视频能够播放。原始的媒体文件体积非常大，假如不对其编码，那么构成一段视频和音频的数据可能会非常庞大，以至于在因特网上传播需耗费无法忍受的时间。若没有解码器的话，接收方就不能把编码过的数据重组为原始的媒体数据。编解码器可以读懂特定的容器格式，并且对其中的音频轨道和视频轨道解码。

下面是一些音频编解码器：

- AAC
- MPEG-3
- Ogg Vorbis

下面是一些视频编解码器：

- H.264

- VP8
- Ogg Theora

#### 编解码器的战争和暂时停战

有一些编解码器受专利保护，有一些则是免费的。例如 Vorbis 音频编解码器和 Theora 视频编解码器可以免费使用，而要想使用 MPEG-4 和 H.264 的话就需要支付相关费用了。

最初，HTML5 规范本来打算指定编解码器，但实施起来困难重重。比如 Ogg Theora，有的厂商考虑到他们已有的硬件或软件不支持，所以不希望 HTML5 规范中包含 Ogg Theora。例如 Apple 的 iPhone 使用的硬件解码器就是 H.264 而不是 Theora。另一方面，免费系统若采用要支付专利费用的编解码器将不可避免地影响其下游分销业务。而专有编解码器的性能指标又是采用免费编解码器的浏览器厂商需要考虑的一个重要因素。这种情况导致了现在的僵局，没有任何一种编解码器可以被所有浏览器厂商接受并在其产品中提供支持。

目前，HTML5 规范放弃了对编解码器的要求，不过未来很有可能重新考虑。现在，开发人员能做的只能是熟悉各种浏览器的支持情况，针对不同的浏览器环境对媒体文件进行重编码（你可能已经这么做了）。

我们非常希望随着时间的推移，不同编解码器的支持程度越来越高并最终汇集成统一版本，这样可以简化将来对通用媒体类型的选择。当然，也有可能某个编解码器逐渐强大起来并最终成为领域内的事实标准。此外，HTML5 的媒体标签内建了一套机制来挑选最适合在浏览器中播放的内容类型，进而实现不同的环境下媒体类型的简单切换。

#### WebM 来了

“2010 年 5 月，Google 发布了 WebM 视频格式。现在主流的 Web 视频都比较模糊，作为新的音频和视频格式，WebM 旨在改善这种现状，让 Web 视频清晰化。WebM 的后缀名是.webm，是基于 Matroska 修改的封装格式，视频使用 VP8 编码，音频使用 Ogg Vorbis 编码。Google 以不受限许可的方式发布了 WebM 的规范和软件，包括源码和专利权。WebM 是一种高清视频格式，对二次开发者和发布者均免费，它代表了编解码器的重大进步。

随着浏览器的不断发展，至少 Firefox、Opera 和 Chrome 宣称会支持 WebM，其中 Opera 10.60 已经实现了对它的支持。Mozilla 和 Google 已经宣布在他们的下一个浏览器版本中提供对 WebM 的支持。”

——Frank<sup>①</sup>

### 3.1.3 HTML5 Audio 和 Video 的限制

有些功能是 HTML5 规范所不支持的，如下所示。

<sup>①</sup> Frank，本书作者之一。——译者注





- 流式音频和视频。因为目前 HTML5 视频规范中还没有比特率切换标准，所以对视频的支持只限于加载的全部媒体文件，但是将来一旦流媒体格式被 HTML5 支持，则肯定会有相关的设计规范。
- HTML5 的媒体受到 HTTP 跨源（cross-origin）资源共享的限制。关于跨源资源共享的更多信息，请参考第 5 章。
- 全屏视频无法通过脚本控制。从安全性角度来看，让脚本元素控制全屏操作是不合适的。不过，如果要让用户在全屏方式下播放视频，浏览器可以提供其他控制手段。
- 对 audio 元素和 video 元素的访问尚未完全加入规范中。基于流行的字幕格式 SRT 的字幕支持规范（WebSRT）仍在编写中。

### 3.1.4 audio 元素和 video 元素的浏览器支持情况

从表 3-1 中可以看到，在写这本书的时候，很多浏览器已经实现了对 HTML5 中 audio 元素和 video 元素的支持。表中还显示了对编解码器的支持情况。

表3-1 audio元素和video元素的浏览器支持情况

浏览器	支持情况	编解码器、容器
Chrome	3.0及以上版本支持	Theora和Vorbis、Ogg容器 H.264和AAC、MPEG 4容器
Firefox	3.5及以上版本支持	Theora和Vorbis、Ogg容器
Internet Explorer	不支持	无
Opera	10.5及以上版本支持	Theora和Vorbis、Ogg容器（10.5及以上版本） VP8 和 Vorbis、WebM格式（10.6及以上版本）
Safari	3.2及以上版本支持	H.264和AAC、MPEG 4容器

按照惯例，在使用 audio 元素和 video 元素之前，第一件事还是检测浏览器是否支持。在 3.2.1 节我们会演示如何通过代码的方式检测浏览器支持情况。

## 3.2 使用 HTML5 Audio 和 Video API

本节，我们会探讨如何在 Web 应用中使用 HTML5 Audio 和 Video。在页面中播放视频的典型方式是使用 Flash、QuickTime 或者 Windows Media 插件向 HTML 中嵌入视频，相对这种方式，使用 HTML5 的媒体标签有两大好处，可以极大地方便用户和开发人员。

- 作为浏览器原生支持的功能，新的 audio 元素和 video 元素无需安装。尽管有的插件安装率很高，但是在控制比较严格的公司环境下往往被屏蔽。因为有的插件绑定广告，所以很多用户选择了禁用此类插件，于是在摆脱广告干扰的同时也丢弃了其媒体播放的能力。插件也是安全问题的一个来源。此外，插件很难与页面其他内容集成，往往在设计好的页面中导致剪裁、透明度等问题。由于插件使用的是独立的渲染模型，与基本的 Web 页



面使用的不同,所以在开发的过程中,如果遇到弹出式菜单或者其他需要跨越插件边界显示重叠元素的时候,开发人员就会面临很大的困难。

- 媒体元素向 Web 页面提供了通用、集成和可脚本化控制的 API。对于开发人员来说,使用新的媒体元素之后,可以轻易地使用脚本来控制和播放内容,本章后面会有很多相关的示例。

使用媒体标签最大的缺点在于缺少通用编解码器支持。这个我们已经在前面讨论过了,但是我们希望编解码器的支持情况越来越好,随着时间的推移能够最终统一起来,到时候通用媒体类型就能轻松实现并且流行起来了。此外,媒体标签内建了一套机制来挑选最适合在浏览器中播放的内容类型,关于这点,我们将在本章随后的部分进行详细讨论。

### 3.2.1 浏览器支持性检测

检测浏览器是否支持 audio 元素或 video 元素最简单的方式是用脚本动态创建它,然后检测特定函数是否存在:

```
var hasVideo = !!!(document.createElement('video').canPlayType);
```

这段脚本会动态创建一个 video 元素,然后检查 canPlayType() 函数是否存在。通过“!!”运算符将结果转换成布尔值,就可以反映出视频对象是否已创建成功。

如果检测结果是浏览器不支持 audio 或 video 元素的话,则需要针对这些老的浏览器触发另外一套脚本来向页面中引入媒体标签,虽然同样可以用脚本控制媒体,但使用的是诸如 Flash 等其他播放技术了。

另外,可以在 audio 元素或 video 元素中放入备选内容,如果浏览器不支持该元素,这些备选内容就会显示在元素对应的位置。可以把以 Flash 插件方式播放同样视频的代码作为备选内容。如果仅仅只想显示一条文本形式提示信息替代本应显示的内容,那就简单了,在 audio 元素或 video 元素中按下面这样插入信息即可:

```
<video src="video.ogv" controls>
  Your browser does not support HTML5 video.
</video>
```

如果是要为不支持 HTML5 媒体的浏览器提供可选方式来显示视频,可以使用相同的方法,将以插件方式播放视频的代码作为备选内容,放在相同的位置即可:

```
<video src="video.ogv">
  <object data="videoplayer.swf" type="application/x-shockwave-flash">
    <param name="movie" value="video.swf"/>
  </object>
</video>
```

在 video 元素中嵌入显示 Flash 视频的 object 元素之后,如果浏览器支持 HTML5 视频,那么 HTML5 视频会优先显示,Flash 视频作后备。不过在 HTML5 被广泛支持之前,可能需要提供多种视频格式。



“让 Web 应用做到所有人都能访问不仅仅是一件正确的事，这更是业务上的需求，有时还是一种法律约束！针对视觉和听觉有障碍的用户，Web 应用应该提供他们需要的备选内容。

对于视频和音频可访问性方面，设计 HTML5 标准的组织敏锐地注意到了规范固有支持性的缺乏，例如未启用字幕等，并且相应的方案也正在计划制定中。在此期间，开发人员应该至少提供视频字幕全文的链接，并且考虑使用 Video API 脚本来同步文本显示，或者显示在视频旁边。

需要注意的是，位于 audio 元素和 video 元素中的备选内容只有在浏览器不支持 HTML5 audio 元素和 video 元素的时候才会起作用，因此无障碍显示的内容不能放在这儿，否则当浏览器支持 HTML5 媒体的时候，用户就看不到了。”

——Brian

### 3.2.2 理解媒体元素

得益于出色的设计思路，HTML5 中的 audio 元素和 video 元素有很多相同之处。两者都支持的操作有播放、暂停、静音/消除静音、加载等，因此通用的动作被从 video 和 audio 元素中剥离出来并放到了媒体 (media) 元素部分。接下来我们就从这些共同点出发，来了解媒体元素。

#### 1. 基本操作：声明媒体元素

为了举例说明，我们会用 audio 元素来演示 HTML5 媒体元素的通用动作。本章的示例涉及大量的媒体文件，这些素材会在随书的示例支持文件中找到。

在下面的示例中 (示例文件是 audio.html)，我们创建了一个音频播放页面，播放的是舒缓、安详，可让公众自由播放的巴赫的《空气》：

```
<!DOCTYPE html>
<html>
  <title>HTML5 Audio </title>
  <audio controls src="johann_sebastian_bach_air.ogg">
    An audio clip from Johann Sebastian Bach.
  </audio>
</html>
```

这段代码假定 HTML 文档和音频文件 (johann\_sebastian\_bach\_air.ogg) 的位置是在同一路径下。图 3-2 是在支持 audio 元素的浏览器中的页面显示效果。从图中可以看到一个带有播放按钮的播放控制条。点击播放按钮，就可以播放音频文件。

代码中的 controls 特性告诉浏览器显示通用的用户控件，包括开始、停止、跳播以及音量控制。如果不指定 controls 特性，用户将无法播放页面上的音频。

audio 元素之间的内容，是为不支持 audio 元素的浏览器准备的替换内容，如果用户用的是老版本浏览器，页面上就会显示这些文本信息。替换内容不局限于文本信息，还可以换成 Flash 播放器等视频播放插件，或者直接给出媒体文件的链接地址。

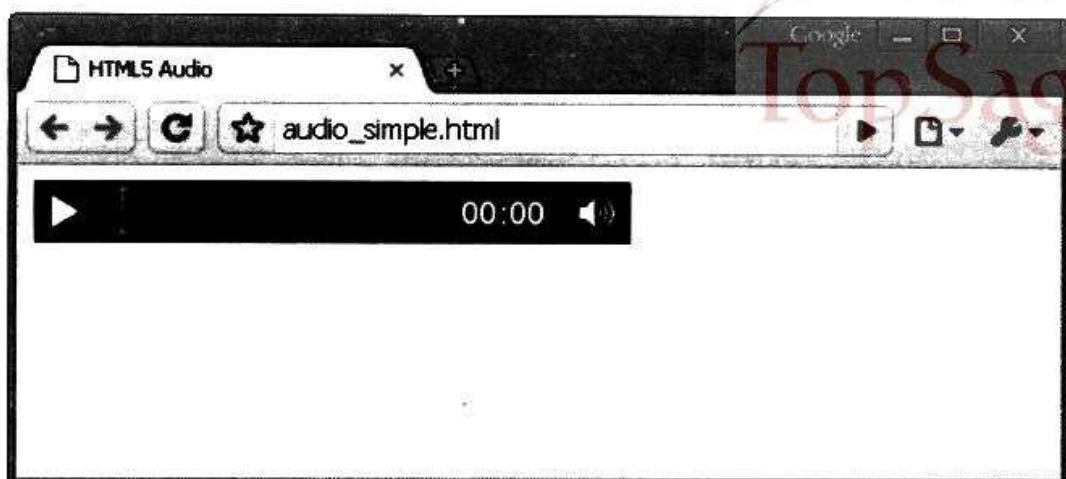


图 3-2 简单的音频控制

## 2. 使用source元素

最后，介绍最重要的特性：`src`。最简单的情况下，`src` 特性直接指向媒体文件就可以了。但是，万一浏览器不支持相关容器或者编解码器呢（比如 Ogg 和 Vorbis）？这就需要用到备用声明了。备用声明中可以包含多种来源，浏览器可以从这么多的来源中进行选择（见示例 `audio_multisource.html`）：

```
<audio controls>
  <source src="johann_sebastian_bach_air.ogg">
  <source src="johann_sebastian_bach_air.mp3">
  An audio clip from Johann Sebastian Bach.
</audio>
```

从上面的代码可以看到，在 `audio` 元素中，我们使用两个新的 `source` 元素替换了先前的 `src` 特性。这样可以让浏览器根据自身播放能力自动选择，挑选最佳的来源进行播放。对于来源，浏览器会按照声明顺序判断，如果支持的不止一种，那么浏览器会选择支持的第一个来源。

---

**提示** 注意来源列表的排放顺序，要按照用户体验由高到低或者服务器消耗由低到高列出。

---

在支持 `audio` 元素的浏览器中运行上述代码，页面的显示结果与添加可选来源列表之前没有任何变化。但是如果浏览器不支持 Ogg Vorbis 格式，而是支持 MP3 格式，那上面的代码就起作用了。这种声明模型的妙处在于当开发人员在与媒体文件进行交互操作的时候，不需要考虑实际上调用的是哪个容器和编解码器。浏览器给开发人员提供操作媒体文件的统一接口，这与所选择播放的文件来源无关。

此外，还有一种暗示浏览器应该使用哪个媒体来源的方式。我们知道一个媒体容器可能会支持多种类型的编解码器，因而就不难理解声明的源文件的扩展名可能会让浏览器误以为自己支持或不支持某种类型了。如果 `type` 特性中指定的类型与源文件不匹配，浏览器可能就会拒绝播放。指定类型的方式就是为 `source` 元素添加 `type` 特性。如果我们明确知道浏览器支持某种类型，将



它写明显然更明智。否则，最好是省略 type 特性，让浏览器自己检测编码方式。此外还需要注意的是，WebM 格式只支持一种音频编解码器和一种视频编解码器。也就是说只要看到 .webm 扩展名或者 video/webm 的内容类型，就可以知道来源的全部情况了。如果浏览器支持 WebM，那么就可以播放所有 .webm 文件，代码如下（示例代码文件为 audio\_type.html）：

```
<audio controls>
  <source src="johann_sebastian_bach_air.ogg" type="audio/ogg; codecs=vorbis">
  <source src="johann_sebastian_bach_air.mp3" type="audio/mpeg">
  An audio clip from Johann Sebastian Bach.
</audio>
```

从上面的代码中可以看到，容器和编解码器都可以在 type 特性中声明。代码中的值分别代表 Ogg Vorbis 和 MP3。所有的支持列表见 RFC 4218，RFC 4218 是由 IETF (Internet Engineering Task Force, Internet 工程任务组) 维护的一套文档。表 3-2 中列出了一些常用组合。

表3-2 媒体类型和特性值

类 型	特 性 值
在Ogg容器中的Theora视频和Vorbis音频	type='video/ogg; codecs="theora, vorbis"'
在Ogg容器中的Vorbis音频	type='audio/ogg; codecs=vorbis'
在MP4容器中的simple baseline H.264视频和low complexity AAC音频	type='video/mp4; codecs="avc1.42E01E, mp4a.40.2"'
在MP4容器中的MPEG-4 简单视频和简单 AAC音频	type='video/mp4; codecs="mp4v.20.8, mp4a.40.2"'

### 3. 媒体的控制

从前面的示例中我们看到，在 audio 元素或 video 元素中指定 controls 特性即可在页面上以默认方式进行播放控制。正像你想的那样，如果不加这个特性，那么在播放的时候就不会显示控制界面。假如播放的是音频，那么页面上任何信息都不会出现，因为音频元素的唯一可视化信息就是它对应的控制界面。假如播放的是视频，那么视频内容会照常显示。即使不添加 controls 特性也不能影响页面正常显示。有一种方法可以让没有 controls 特性的音频或视频正常播放，那就是在 audio 元素或 video 元素中设置另一个特性：autoplay（见 audio\_no\_control.html 示例文件）：

```
<audio autoplay>
  <source src="johann_sebastian_bach_air.ogg" type="audio/ogg; codecs=vorbis">
  <source src="johann_sebastian_bach_air.mp3" type="audio/mpeg">
  An audio clip from Johann Sebastian Bach.
</audio>
```

通过设置 autoplay 特性，不需要任何用户交互，音频或视频文件就会在加载完成后自动播放。不过大部分用户对这种方式会比较反感，所以应慎用 autoplay 特性。在无任何提示的情况下，播放一段音频通常有两种用途，第一种是用来制造背景氛围，第二种是强制用户接收广告。这种方式的问题在于会干扰用户本机播放的其他音频，尤其会给依赖屏幕阅读功能进行 Web 内容导航的用户带来不便。

如果内置的控件不适应用户界面的布局, 或者希望使用默认控件中没有的条件或者动作来控制音频或视频文件, 那么可以借助一些内置的 JavaScript 函数和特性。表 3-3 列出了一些常用的函数。

表3-3 常用的控制函数

函 数	动 作
load()	加载音频/视频文件, 为播放做准备。通常情况下不必调用, 除非是动态生成的元素。用来在播放前预加载
play()	加载 (有必要的) 并播放音频/视频文件。除非音频/视频已经暂停在其他位置了, 否则默认从开头播放
pause()	暂停处于播放状态的音频/视频文件
canPlayType(type)	测试video元素是否支持给定MIME类型的文件

canPlayType(type)函数有一个特殊的用途: 向动态创建的 video 元素中传入某段视频的 MIME 类型后, 仅仅通过一行脚本语句即可获得当前浏览器对相关视频类型的支持情况。例如, 下面的代码可以快速判断当前浏览器是否支持播放 fooType 类型的视频, 而无需在浏览器窗口中显示任何内容:

```
var supportsFooVideo = !(document.createElement('video').canPlayType('fooType'));
```

表 3-4 列出了媒体元素中的部分只读特性。

表3-4 只读的媒体特性

只读特性	值
duration	整个媒体文件的播放时长, 以s为单位。如果无法获取时长, 则返回NaN
paused	如果媒体文件当前被暂停, 则返回true。如果还未开始播放, 默认返回true
ended	如果媒体文件已经播放完毕, 则返回true
startTime	返回最早的播放起始时间, 一般是0.0, 除非是缓冲过的媒体文件, 并且一部分内容已经不在缓冲区
error	在发生了错误的情况下返回的错误代码
currentSrc	以字符串形式返回当前正在播放或已加载的文件。对应于浏览器在source元素中选择的文件

表 3-5 列出了可被脚本修改并直接影响播放的部分媒体元素特性。它们的作用类似于函数。

表3-5 可脚本控制的特性值

特 性	值
autoplay	将媒体文件设置为创建后自动播放, 或者查询是否已设置为autoplay
loop	如果媒体文件播放完毕后能重新播放则返回true, 或者将媒体文件设置为循环播放 (或者不循环播放)
currentTime	以s为单位返回从开始播放到现在所用的时间。在播放过程中, 设置currentTime来进行搜索, 并定位到媒体文件的特定位置



(续)

特 性	值
controls	显示或隐藏用户控制界面，或者查询用户控制界面当前是否可见
volume	在0.0到1.0之间设置音频音量的相对值，或者查询当前音量相对值
muted	为音频文件设置静音或者消除静音，或者检测当前是否为静音
autobuffer	通知播放器在媒体文件开始播放前，是否进行缓冲加载。如果媒体文件已设置为autoplay，则忽略此特性

开发人员可以基于上述功能和特性自行开发媒体播放用户界面，进而实现对浏览器支持的音频和视频的控制。

### 3.2.3 使用 audio 元素

如果你已经了解了 audio 元素和 video 元素的共享特性，那就基本认识了 audio 元素所能提供的功能。我们直接看一个简单的示例，它演示了如何通过脚本来控制 audio 元素。

#### 激活 audio 元素

如果需要在用户交互界面上播放一段音频，同时不想被时间轴和控制界面影响页面显示效果，则可创建一个隐藏的 audio 元素——不设置 controls 特性，或将其设置为 false——然后用自行开发的控制界面控制音频的播放。参考下面这段代码，代码文件为 audioCue.html：

```

<!DOCTYPE html>
<html>
  <link rel="stylesheet" href="styles.css">
  <title>Audio cue</title>

  <audio id="clickSound">
    <source src="johann_sebastian_bach_air.ogg">
    <source src="johann_sebastian_bach_air.mp3">
  </audio>

  <button id="toggle" onclick="toggleSound()">Play</button>

  <script type="text/javascript">
    function toggleSound() {
      var music = document.getElementById("clickSound");
      var toggle = document.getElementById("toggle");

      if (music.paused) {
        music.play();
        toggle.innerHTML = "Pause";
      }
      else {
        music.pause();
        toggle.innerHTML = "Play";
      }
    }
  </script>
</html>

```

我们再次用 audio 元素播放了巴赫的那首曲子。不过在示例中，我们隐藏了用户控制界面，也没有将其设置为加载后自动播放，而是创建了一个具有切换功能的按钮，以脚本的方式控制音频的播放：

```
<button id="toggle" onclick="toggleSound()">Play</button>
```

按钮在初始化时会提示用户单击它以播放音频。每次单击时，都会触发 toggleSound() 函数。在 toggleSound() 函数中，我们首次访问了 DOM 中的 audio 元素和 button 元素：

```
if (music.paused) {
    music.play();
    toggle.innerHTML = "Pause";
}
```

通过访问 audio 元素的 paused 特性，可以检测到用户是否已经暂停播放。如果音频还没开始播放，那么 paused 特性默认为 true，这种情况会在用户第一次单击按钮的时候遇到。此时，需要调用 play() 函数来播放音频，同时修改按钮上的文字，告诉用户再次单击就会暂停：

```
else {
    music.pause();
    toggle.innerHTML = "Play";
}
```

相反，如果音频没有暂停（仍在播放），我们会使用 pause() 函数将它暂停，然后变换按钮上的文字，让用户知道下次单击的时候音频将继续播放。挺容易，不是吗？这是 HTML5 媒体元素的关键：建立简单的显示和控制机制支持多种媒体类型，以淘汰媒体播放插件。简单是 HTML5 媒体元素最大的优点。

### 3.2.4 使用 video 元素

我们已经讨论了很多简单的应用。后面的示例中，我们将尝试提高一些复杂度。HTML5 video 元素同 audio 元素非常类似，只是比 audio 元素多了一些特性，详见表 3-6。

表3-6 video元素的额外特性

特 性	值
poster	在视频加载完成之前，代表视频内容的图片的URL地址，可以想象一下“电影海报”。该特性不仅可读，而且可以修改，以便更换图片
width、height	读取或设置显示尺寸。如果设置的宽度与视频本身大小不匹配，可能导致居中显示，上下或左右可能出现黑色条状区域
videowidth、videoHeight	返回视频的固有或自适应的宽度和高度。只读

video 元素还有一个 audio 元素不支持的关键特性：可被 HTML5 Canvas 的函数调用（更多关于 HTML5 Canvas 的内容见第 2 章）。

#### 1. 创建视频时序查看器

在下面这个稍显复杂的示例中，我们将演示如何抓取 video 元素中的帧并显示在动态 canvas



上。为了演示这个功能，我们将创建一个简单的视频时序查看器。当视频播放时，定期从视频中抓取图像帧并绘制到旁边的 canvas 上，当用户单击 canvas 上显示的任何一帧时，所播放的视频会跳转到相应的时间点。只需寥寥几行代码即可创建时序查看器，它支持用户在较长的视频中随意跳转。

示例中选用的视频是 20 世纪中期电影院的优惠广告，一起来感受一下吧（见图 3-3）。

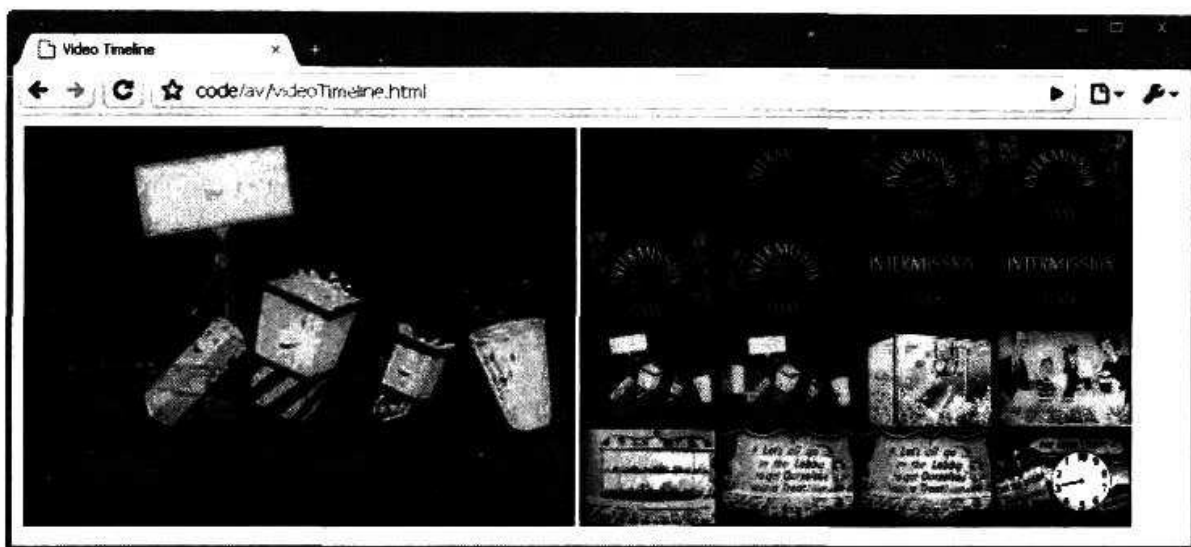


图 3-3 视频时序查看器

## 2. 添加video元素和canvas元素

先用一段简单的代码来显示视频：

```
<video id="movies" autoplay oncanplay="startVideo()" onended="stopTimeline()"
  autobuffer="true" width="400px" height="300px">
  <source src="Intermission-Walk-in.ogv" type='video/ogg; codecs="theora, vorbis"'>
  <source src="Intermission-Walk-in_512kb.mp4"
    type='video/mp4; codecs="avc1.42E01E, mp4a.40.2"'>
</video>
```

其中大多数标记与 audio 元素的示例代码中类似，不再赘述。我们着重讨论不同的地方。很明显 <audio> 元素替换成了 <video> 元素，并且 <source> 元素分别指向 Ogg 和 MPEG 的视频以供浏览器选择。

video 元素声明了 autoplay 特性，这样一来，页面加载完成后，视频马上会被自动播放。此外还增加了两个事件处理函数。当视频加载完毕，准备开始播放的时候，会触发 oncanplay 函数来执行我们预设的动作。类似地，当视频播放完后，会触发 onended 函数以停止帧的创建。

接下来我们将创建 id 为 timeline 的 canvas，之后会以固定的时间间隔在上面绘制视频帧。

```
<canvas id="timeline" width="400px" height="300px">
```

## 3. 添加变量

接下来我们要为示例编写脚本代码，在脚本中声明一些有助于调整示例的变量，同时增强代

码可读性。

```
// 抓取帧的时间间隔：单位是 ms
var updateInterval = 5000;

// 时序中帧的尺寸
var frameWidth = 100;
var frameHeight = 75;

//时序的总帧数
var frameRows = 4;
var frameColumns = 4;
var frameGrid = frameRows * frameColumns;
```

updateInterval 用来控制抓取帧的频率——代码中是每 5 s 一次。frameWidth 和 frameHeight 两个参数用来指定在 canvas 中展示的视频帧的大小。类似地，frameRows、frameColumns 以及 frameGrid 三个参数决定了在时序中总共显示多少帧。

```
//当前帧
var frameCount = 0;

// 播放完后取消计时器
var intervalId;

var videoStarted = false;
```

为了跟踪当前播放的是哪一帧，我们引入了 frameCount 变量。frameCount 变量可被示例中的所有函数调用。（出于演示目的，示例视频每隔 5 s 会取出一帧。）intervalId 用来停止控制抓取帧的计时器。最后，我们还添加了 videoStarted 标志以确保每个示例只创建一个计时器。

#### 4. 添加updateFrame函数

整个示例的核心功能是抓取视频帧并绘制到 canvas 上，它是视频与 canvas 相结合的部分，其代码如下：

```
//把帧绘制到画布上
function updateFrame() {
    var video = document.getElementById("movies");
    var timeline = document.getElementById("timeline");

    var ctx = timeline.getContext("2d");

    //根据帧数计算出当前播放位置
    //然后以视频为输入参数绘制图像
    var framePosition = frameCount % frameGrid;
    var frameX = (framePosition % frameColumns) * frameWidth;
    var frameY = (Math.floor(framePosition / frameRows)) * frameHeight;
    ctx.drawImage(video, 0, 0, 400, 300, frameX, frameY, frameWidth, frameHeight);

    frameCount++;
}
```



在第2章我们已经了解到，在操作 canvas 前，首先需要做的就是获取 canvas 的二维上下文对象：

```
var ctx = timeline.getContext("2d");
```

因为我们希望按从左到右、从上到下的顺序填充 canvas 网格，所以需要精确计算从视频中截取的每帧应该对应到哪个 canvas 网格中。根据每帧的宽度和高度，可以计算出它们的起始绘制坐标(X, Y)：

```
var framePosition = frameCount % frameGrid;  
var frameX = (framePosition % frameColumns) * frameWidth;  
var frameY = (Math.floor(framePosition / frameRows)) * frameHeight;
```

最后是将图像绘制到 canvas 上的关键函数调用。在第2章的示例中，我们已经使用过位置和缩放参数，但这里我们向 drawImage() 函数中传入的不是图像，而是视频对象：

```
ctx.drawImage(video, 0, 0, 400, 300, frameX, frameY, frameWidth, frameHeight);
```

canvas 的绘图程序可以将视频源当做图像或者图案进行处理，这样开发人员就可以方便地修改视频并将其重新显示在其他位置。

---

**提示** 当 canvas 使用视频作为绘制来源时，画出来的只是当前播放的帧。canvas 的显示图像不会随着视频的播放而动态更新，如果希望更新显示内容，需要在视频播放期间重新绘制图像。

---

### 5. 添加 startVideo 函数

最后，更新 frameCount，这表示我们开始在时序查看器上绘制新的视频截图。现在示例只剩下一个功能要实现，那就是定时更新时序查看器上的帧：

```
function startVideo() {  
    // 只在视频第一次播放时设置计时器。  
    if (videoStarted)  
        return;  
  
    videoStarted = true;  
  
    // 计算初始帧，然后以规定时间间隔创建其他帧  
    updateFrame();  
    intervalId = setInterval(updateFrame, updateInterval);  
}
```

别忘了，一旦视频加载并可以播放就会触发 startVideo() 函数。因此，我们首先要保证每次页面加载都仅触发一次 startVideo()，除非视频重新播放。

```
// 只在视频第一次播放时设置计时器  
if (videoStarted)  
    return;  
  
videoStarted = true;
```

视频开始播放后，我们将抓取第一帧，接着会启用间隔计时器来定期调用 `updateFrame()` 函数。所谓间隔计时器是以指定时间间隔不断重复的计时器。示例中的结果是每 5s 抓取一个新截图：

```
//计算初始帧，然后定期创建其他帧
updateFrame();
intervalId = setInterval(updateFrame, updateInterval);
```

## 6. 处理用户输入

用户单击时序查看器上的某一帧时，系统该怎么处理？下面我们就来回答这个问题：

```
//创建事件处理函数，用来在用户单击某帧后定位视频
var timeline = document.getElementById("timeline");
timeline.onclick = function(evt) {
    var offX = evt.layerX - timeline.offsetLeft;
    var offY = evt.layerY - timeline.offsetTop;

    //计算以零为基准索引的网格中哪帧被单击
    var clickedFrame = Math.floor(offY / frameHeight) * frameRows;
    clickedFrame += Math.floor(offX / frameWidth);

    //视频开始后已经播放到多少帧
    var seekedFrame = (((Math.floor(frameCount / frameGrid)) *
        frameGrid) + clickedFrame);

    //如果用户单击的帧在当前帧之前，则假定是上一轮的帧
    if (clickedFrame > (frameCount % 16))
        seekedFrame -= frameGrid;

    //不允许跳出当前视频
    if (seekedFrame < 0)
        return;
```

代码稍显复杂。我们获取了 id 为 `timeline` 的 `canvas`，并对其设置了用于处理用户单击的函数。该函数通过单击事件来确定用户单击位置的  $X$  和  $Y$  坐标：

```
var timeline = document.getElementById("timeline");
timeline.onclick = function(evt) {
    var offX = evt.layerX - timeline.offsetLeft;
    var offY = evt.layerY - timeline.offsetTop;
```

然后，我们利用帧的尺寸计算出用户单击的是 16 个帧中的哪一个：

```
//计算以零为基准索引的网格中哪帧被单击
var clickedFrame = Math.floor(offY / frameHeight) * frameRows;
clickedFrame += Math.floor(offX / frameWidth);
```

用户单击的帧一定是刚刚播放的视频帧中的一个，所以通过对应的网格索引能够计算出最近播放的帧：

```
//视频开始后已经播放到多少帧
var seekedFrame = (((Math.floor(frameCount / frameGrid)) *
    frameGrid) + clickedFrame);
```





如果用户单击的帧在当前帧之前,需要向前跳跃一个完整的网格周期,以确定实际播放时间:

```
//如果用户单击的帧在当前帧之前,那么假定这是上一轮的帧
if (clickedFrame > (frameCount % 16))
    seekedFrame -= frameGrid;
```

最后,我们必须添加安全控制代码,以免用户单击的帧由于某种原因比视频初始帧还早:

```
// 不允许跳出当前视频
if (seekedFrame < 0)
    return;
```

现在我们已经知道了用户想要跳转到的时间点,接下来是实现跳转播放。虽然这是示例的核心函数,但是代码却很简单:

```
//计算出这一帧对应的视频(以s为单位)
var video = document.getElementById("movies");
video.currentTime = seekedFrame * updateInterval / 1000;

// 设置目标帧
frameCount = seekedFrame;
```

通过设置 video 元素的 currentTime 特性,可以让视频自动跳转到指定时间,并将当前帧数 (frameCount) 设置为新选择的帧。

---

**提示** 与JavaScript计时器以ms为单位不同, video中的currentTime以秒为单位。

---

### 7. 添加stopTimeline函数

整个视频时序查看器示例中,最后要做的工作是在视频播放完毕时,停止帧的抓取。虽然不是必须的,但是如果不这么做,示例会在现有代码基础上不停地抓取帧,过段时间就会让时序查看器变成一片空白:

```
//停止绘制时序的帧
function stopTimeline() {
    clearInterval(intervalId);
}
```

视频播放完毕时会触发 onended()函数, stopTimeline 函数会在此时被调用。

我们的视频时序查看器的功能可能还不够强大,无法满足高端用户的要求,不过别忘了它仅用了很少的代码啊。好,继续吧。

## 3.2.5 进阶功能

尽管有些技巧我们在示例中用不上,但这并不妨碍它们在多种类型的 HTML5 Web 应用中发挥作用。本节,我们就来介绍一些简单实用的进阶功能。

### 1. 页面中的背景噪音

很多网站为所有访问者自动播放音乐,希望以这种方式来“款待”用户。虽然有时候我们不大喜欢这样的方式,不过用 HTML5 的 audio 元素实现起来倒是非常便捷:

```

<!DOCTYPE html>
<html>
  <link rel="stylesheet" href="styles.css">
  <title>Background Music</title>

  <audio autoplay loop>
    <source src="johann_sebastian_bach_air.ogg">
    <source src="johann_sebastian_bach_air.mp3">
  </audio>

  <h1>You're hooked on Bach!</h1>
</html>

```

从上面的代码中可以看到，实现循环播放一首背景音乐非常简单，只需在 audio 元素中设置 autoplay 和 loop 特性即可（见图 3-4）。



图 3-4 设置 autoplay 特性以便在页面加载时播放音乐

—眨眼（<blink>）用户就会流失

“所谓能力越强责任越大，但有些事情可以做，并不代表着应该做。<blink>元素就是一个活生生的例子<sup>①</sup>！”

audio 元素和 video 元素可以通过简单的方式实现强大的播放功能，但不要仅仅因为这个便利性就在不恰当的地方肆意使用它们。如果开发人员有足够的理由对页面的音频或视频使用 autoplay（可能是媒体浏览器，用户期望一打开页面就播放内容），一定要确保明确提供关闭自动播放的功能。当用户发现页面上有烦人的内容而无法轻易关闭时，会断然离开。想让用户流失？没有比这种方式更快的了。”

——Brian

① <blink>标签的作用是闪屏。业界普遍不推荐使用此标签，因为使用<blink>会造成字体的闪烁，不利于用户了解文字内容。——译者注



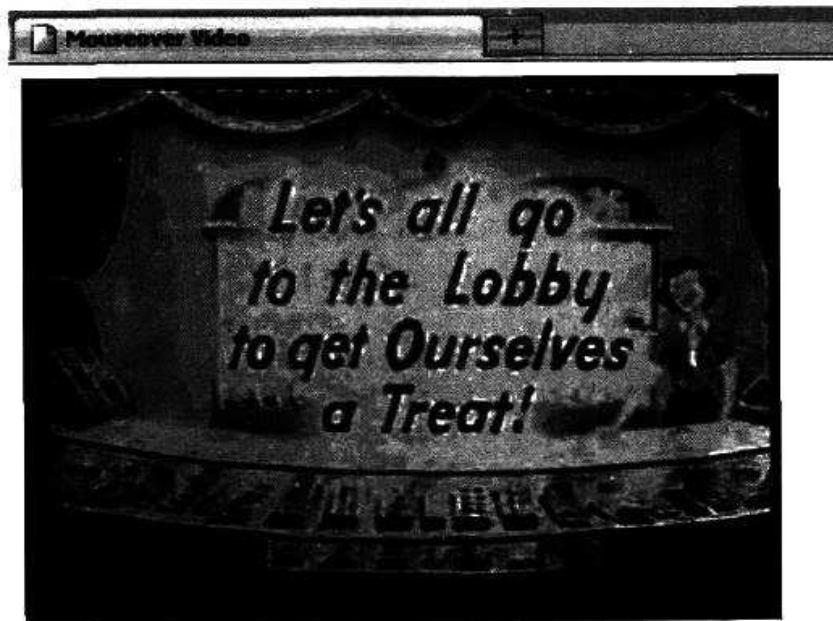
## 2. 鼠标悬停播放视频

video 元素中, 另一种简单高效的用法是通过在视频上移动鼠标来触发 play 和 pause 功能。页面包含多个视频, 且由用户来选择播放某个视频时, 这个功能就非常适用了。比如在用户鼠标移到某个视频上时, 播放简短的视频预览片段, 用户单击后, 播放完整的视频。上述效果可以通过下面的代码轻松实现 (示例文档为 mouseoverVideo.html):

```
<!DOCTYPE html>
<html>
  <link rel="stylesheet" href="styles.css">
  <title>Mouseover Video</title>

  <video id="movies" onmouseover="this.play()" onmouseout="this.pause()"
    autobuffer="true"
    width="400px" height="300px">
    <source src="Intermission-Walk-in.ogv" type='video/ogg; codecs="theora, vorbis"'>
    <source src="Intermission-Walk-in_512kb.mp4" type='video/mp4; codecs="avc1.42E01E,
      mp4a.40.2"'>
  </video>
</html>
```

只需额外设置几个特性, 即可实现鼠标悬停预览视频的效果, 如图 3-5 所示。



**Point at the video to play it!**

图 3-5 鼠标悬停播放视频

## 3.3 小结

本章我们介绍了 HTML5 audio 元素和 video 元素的用法, 演示了如何使用它们构建引人注

目的 Web 应用。audio 元素和 video 元素的引入，让 HTML5 应用在对媒体的使用上多了一种选择：不用插件即可播放音频和视频。此外，audio 元素和 video 元素还提供了通用的、集成化的、可用脚本控制的 API。

我们首先了解了 audio 和 video 的容器文件和编解码器，以及为什么 HTML5 支持这些编解码器，然后演示了一种让浏览器自动选择最合适媒体类型进行播放的机制。

接下来，我们演示了如何通过编程方式使用 API 控制 audio 元素和 video 元素。最后，我们探讨了 HTML5 Audio 和 Video 的实际应用。

下一章，我们将演示如何在 Web 应用中基于地理定位信息来更新用户所在的位置。



**让**我们设想一个场景，有一个 Web 应用程序，它可以向用户提供附近不远处某商店运动鞋的打折优惠信息。使用 HTML5 Geolocation API（地理定位 API），你可以请求用户共享他们的位置，如果他们同意，应用程序就可以向其提供相关信息，告诉用户去附近哪家商店可以挑选到打折的鞋子。

HTML5 Geolocation 技术的另一个应用场景是构建计算行走（跑步）路程的应用程序。想象一下，在开始跑步时通过手机浏览器启动应用程序的记录功能。在用户移动过程中，应用程序会记录已跑过的距离，还可以把跑步过程对应的坐标显示在地图上，甚至可以显示出海拔信息。如果用户正在和其他选手一起参加跑步比赛，应用程序甚至可以显示其对手的位置。

再有一种 HTML5 Geolocation 应用场景是基于 GPS 导航的社交网络应用，可以用它看到好友们当前所处的位置，比如知道了好友的方位，就可以挑选合适的咖啡馆。此外，还有很多特殊的应用。

本章，我们会探讨 HTML5 Geolocation API，它允许用户在 Web 应用程序中共享他们的位置，使其能够享受位置感知服务。首先，我们了解一下 HTML5 Geolocation 位置信息的来源：纬度、经度和其他特性，以及获取这些数据的途径（GPS、Wi-Fi 和蜂窝站点等）。然后，我们将讨论 HTML5 地理定位数据的隐私问题，以及浏览器如何使用这些数据。

在此之后，我们将深入探讨 HTML5 Geolocation API 在实际中的应用。目前有两种类型的定位请求 API：单次定位请求和重复性的位置更新请求。我们会分别介绍这两种请求的使用方式和适用场景。接下来，我们会演示如何使用它们构建实用的 HTML5 Geolocation 应用程序。最后，我们讨论其他一些用例和使用技巧。

## 4.1 位置信息

HTML5 Geolocation API 的使用方法相当简单。请求一个位置信息，如果用户同意，浏览器就会返回位置信息，该位置信息是通过支持 HTML5 地理定位功能的底层设备（例如，笔记本电脑或手机）提供给浏览器的。位置信息由纬度、经度坐标和一些其他元数据组成。有了这些位置信息就可以构建引人注目的位置感知类应用程序。

### 4.1.1 纬度和经度坐标

位置信息主要由下例所示的一对纬度和经度坐标组成。以美国最美丽的高山湖泊——塔霍湖边的塔霍城坐标为例：

纬度：39.17222，经度：-120.13778

在这里，纬度（距离赤道以北或以南的数值表示）是 39.172 22，经度（距离英国格林威治以东或以西的数值表示）是-120.137 78。

经纬度坐标可以用以下两种方式表示：

- 十进制格式（例如，39.172 22）；
- DMS（Degree Minute Second，角度）格式（例如，39° 10' 20'）。

---

提示 HTML5 Geolocation API返回坐标的格式为十进制格式。

---

除了纬度和经度坐标，HTML5 Geolocation 还提供位置坐标的准确度。除此之外，它还可能提供其他一些元数据，具体情况取决于浏览器所在的硬件设备，这些元数据包括海拔、海拔准确度、行驶方向和速度等。如果这些元数据不存在则返回 null。

### 4.1.2 位置信息从何而来

HTML5 Geolocation API 不指定设备使用哪种底层技术来定位应用程序的用户。相反，它只是用于检索位置信息的 API，而且通过该 API 检索到的数据只具有某种程度的准确性。它并不能保证设备返回的实际位置是精确的。

#### 关于位置

“有一个很有趣的例子。我家用的是无线网络，当我在 Firefox 中打开本章中的 HTML5 地理定位示例应用程序时，它显示我在萨克拉门托（约偏离我的实际位置 121km (1 mile=1.6 km)）。这个结果是错的，但也不足为奇，因为我的 ISP（Internet Service Provider，互联网服务提供商）位于萨克拉门托市中心。

然后，我让我的儿子，Sean 和 Rocky，在他们的 iPod Touch 上（使用同一个 Wi-Fi 网络）浏览同一页面。在 Safari 浏览器中，显示他们在加利福尼亚州一个叫做 Marysville 的小镇，距离萨克拉门托 48km。想想为什么会出现这种情况吧！”

——Peter

---

设备可以使用下列数据源。

- IP 地址
- 三维坐标
  - GPS（Global Positioning System，全球定位系统），



- 从 RFID、Wi-Fi 和蓝牙到 Wi-Fi 的 MAC 地址；
- GSM 或 CDMA 手机的 ID。
- 用户自定义数据

为了保证更高的准确度，许多设备使用 1 个或多个数据源的组合。这些方式各有优缺点，下面会详细介绍。

### 4.1.3 IP 地址地理定位数据

过去，基于 IP 地址的地理定位方法是获得位置信息的唯一方式，但其返回的位置信息通常并不靠谱。基于 IP 地址的地理定位的实现方式是：自动查找用户的 IP 地址，然后检索其注册的物理地址。因此，如果用户的 IP 地址是 ISP 提供的，其位置往往就由服务供应商的物理地址决定，该地址可能距离用户数千米。表 4-1 是基于 IP 地址的地理定位数据的优缺点。

表4-1 基于IP地址的地理位置数据的优缺点

优 点	缺 点
任何地方都可用 在服务器端处理	不精确（经常出错，一般精确到城市级） 运算代价大

许多网站会根据由 IP 地址得到的位置信息来做广告，所以在实际中可能会遇到这样的情况：你到其他国家旅行，访问非本地网站时却突然看到了本地广告（基于你访问网站所在国家或地区的 IP 地址）。

### 4.1.4 GPS 地理定位数据

只要可以看到天空的地方，GPS 就可以提供非常精确的定位结果。GPS 定位是通过收集运行在地球周围的多个 GPS 卫星的信号实现的。但是，它的定位时间可能较长，因此它不适合需要快速响应的应用程序。

因为获取 GPS 定位数据所需时间可能较长，所以开发人员可能需要异步查询用户位置。可以添加一个状态栏以显示正在重新获取应用程序用户的位置。表 4-2 是基于 GPS 的地理定位数据的优缺点。

表4-2 基于GPS的地理定位数据的优缺点

优 点	缺 点
很精确	定位时间长，用户耗电量 室内效果不好 需要额外硬件设备

### 4.1.5 Wi-Fi 地理定位数据

基于 Wi-Fi 的地理定位信息是通过三角距离计算得出的，这个三角距离指的是用户当前位置

到已知的多个 Wi-Fi 接入点（几乎都在城市里）的距离。不同于 GPS，Wi-Fi 在室内也非常准确。表 4-3 是基于 Wi-Fi 的地理定位数据的优缺点。

表4-3 基于Wi-Fi的地理定位数据的优缺点

优 点	缺 点
精确 可在室内使用 可以简单、快捷定位	在乡村这些无线接入点较少的地区效果不好

#### 4.1.6 手机地理定位数据

基于手机的地理定位信息是通过用户到一些基站的三角距离确定的。这种方法可提供相当准确的位置结果。这种方法通常同基于 Wi-Fi 和基于 GPS 的地理定位信息结合使用。表 4-4 是基于手机的地理定位数据的优缺点。

表4-4 基于手机的地理定位数据的优缺点

优 点	缺 点
相当准确 可在室内使用 可以简单、快捷定位	需要能够访问手机或其modem的设备 在基站较少的偏远地区效果不好

#### 4.1.7 用户自定义的地理定位数据

除了通过编程计算出用户的位置外，也可以允许用户自定义其位置。应用程序可能允许用户输入他们的地址、邮政编码和其他一些详细信息。应用程序可以利用这些信息来提供位置感知服务。表 4-5 是用户自定义的地理定位数据的优缺点。

表4-5 用户自定义的地理定位数据的优缺点

优 点	缺 点
用户可以获得比程序定位服务更准确的位置数据 允许地理定位服务的结果作为备用位置信息 用户自行输入可能比自动检测更快	可能很不准确，特别是当用户位置变更后

## 4.2 HTML5 Geolocation 的浏览器支持情况

各浏览器对 HTML5 Geolocation 的支持程度不同，并且还在不断更新。在 HTML5 的所有功能中，HTML5 Geolocation 是第一批被全部接受和实现的功能之一，这对于开发人员来说是个好消息。相关规范已达到一个非常成熟的阶段，不大可能做太大改变。如表 4-6 所示，在本书编写时，很多浏览器已经支持 HTML5 Geolocation。



表4-6 浏览器对HTML5 Geolocation的支持情况

浏览器	支持情况
Chrome	在带有Gears的第2版Google Chrome中被支持
Firefox	3.5及以上版本支持
Internet Explorer	通过Gears插件支持
Opera	计划在版本10中支持，已经在开发版本中进行测试
Safari	在版本4中支持以实现在iPhone上可用

由于浏览器对它的支持程度不同，在使用之前最好先检查浏览器是否支持 HTML5 Geolocation API。4.4.1 节将讲解如何以编程方式检查浏览器是否支持此功能。

### 4.3 隐私

HTML5 Geolocation 规范提供了一套保护用户隐私的机制。除非得到用户明确许可，否则不可获取位置信息。

这个隐私处理机制很合理，也解释了用户一直问到的 HTML5 Geolocation 应用程序中的老大难问题。不过，从可接触到的 HTML5 Geolocation 应用程序示例中可以看到，通常会鼓励用户共享这些信息。例如，用户正在某咖啡厅喝咖啡，如果应用程序可以让他们得知该咖啡店附近的商店中有少见的五折跑鞋，那么用户就会觉得共享他们的位置信息是可以接受的。让我们仔细看看图 4-1 所示浏览器和设备的隐私架构。

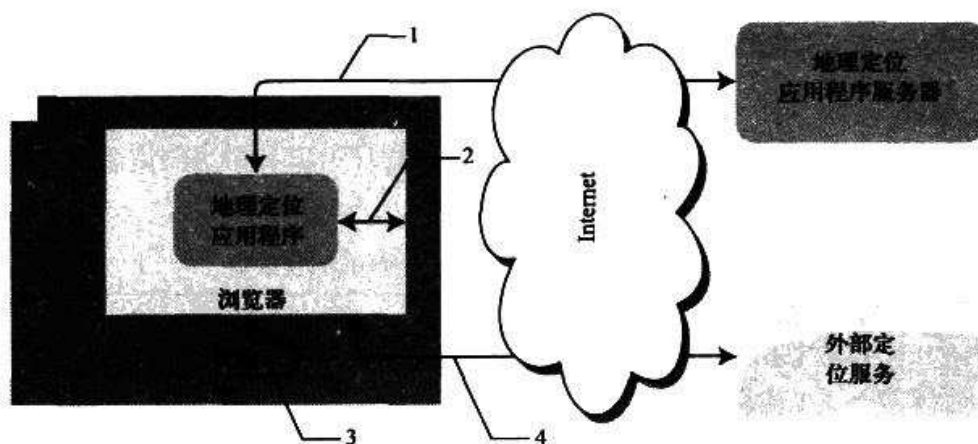


图 4-1 HTML5 地理定位浏览器和设备的隐私架构

图中标识的步骤如下。

- (1) 用户从浏览器中打开位置感知应用程序。
- (2) 应用程序 Web 页面加载，然后通过 Geolocation 函数调用请求位置坐标。浏览器拦截这一请求，然后请求用户授权。我们假设用户同意。
- (3) 浏览器从其宿主设备中检索坐标信息。例如，IP 地址、Wi-Fi 或 GPS 坐标。这是浏览器的内部功能。

(4) 浏览器将坐标发送给受信任的外部定位服务，它返回一个详细位置信息，并将该位置信息发回给 HTML5 Geolocation 应用程序。

**重要提示** 应用程序不能直接访问设备，它只能请求浏览器来代表它访问设备。

### 4.3.1 触发隐私保护机制

访问使用 HTML5 Geolocation API 的页面时，会触发隐私保护机制。图 4-2 显示了在 Firefox 3.5 中触发隐私保护机制的页面。

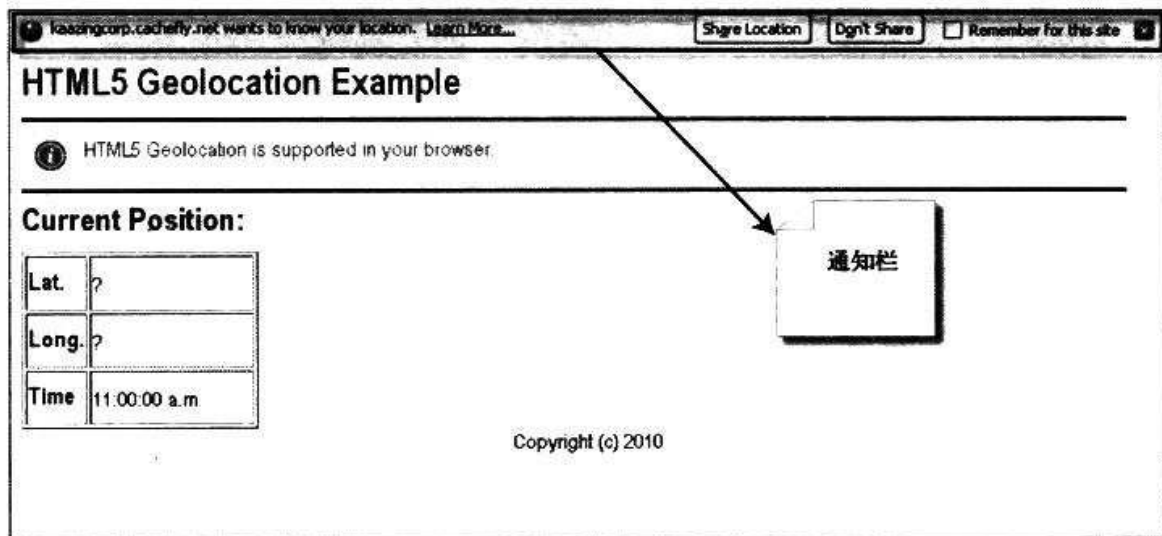


图 4-2 在 Firefox 中调用 HTML5 Geolocation API 时触发的通知栏

执行 HTML5 Geolocation 代码时会触发这一机制。如果仅仅是添加 HTML5 Geolocation 代码，而不被任何方法调用（例如，在 onload 方法中调用），则不会触发隐私保护机制。只要所添加的 HTML5 Geolocation 代码被执行，浏览器就会提示用户应用程序要共享他们的位置。执行 HTML5 Geolocation 的方式很多，例如调用 navigator.geolocation.getCurrentPosition 方法等（后面将会有更详细的解释）。图 4-3 显示了 iPhone 中 Safari 上的运行结果。

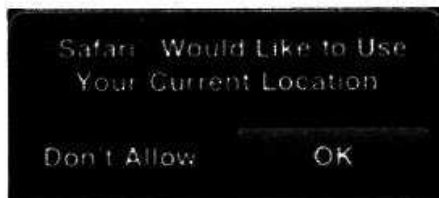


图 4-3 HTML5 Geolocation API 调用时在 Safari 中触发的通知对话框

除了询问用户是否允许共享其位置之外，Firefox 等一些浏览器还可以让用户选择记住该网站的位置服务权限，以便下次访问的时候不再弹出提示框，类似于在浏览器中记住某些网站的密码。



**提示** 在Firefox中，如果用户已经允许其向某网站发送位置信息，但后来又改变主意了，那么很简单，返回到该网站，选择Tools菜单的Page Info，更改Permission选项卡中的Share Location设置即可。

### 4.3.2 处理位置信息

因为位置数据属于敏感信息，所以接收到之后，必须小心地处理、存储和重传。如果用户没有授权存储这些数据，那么应用程序应该在相应任务完成后立即删除它。

如果要重传位置数据，建议先对其进行加密。在收集地理定位数据时，应用程序应该着重提示用户以下内容：

- 会收集位置数据；
- 为什么收集位置数据；
- 位置数据将保存多久；
- 怎样保证数据的安全；
- 位置数据怎样共享（如果同意共享）；
- 用户怎样检查和更新他们的位置数据。

## 4.4 使用 HTML5 Geolocation API

本节将更详细地探讨 HTML5 Geolocation API 的使用方法。为了更好地解释说明，我们创建了一个简单的浏览器页面 geolocation.html。所有代码都可以从 apress.com 本书页面中下载。

### 4.4.1 浏览器支持性检查

开发人员在调用 HTML5 Geolocation API 函数前，需要确保浏览器支持其所要完成的所有工作。这样，当浏览器不支持时，就可以提供一些替代文本，以提示用户升级浏览器或安装插件（如 Gears）来增强现有浏览器功能。代码清单 4-1 是浏览器支持性检查的一种途径。

#### 代码清单 4-1 检查浏览器支持性

```
function loadDemo() {  
    if(navigator.geolocation) {  
        document.getElementById("support").innerHTML = "HTML5 Geolocation supported.";  
    } else {  
        document.getElementById("support").innerHTML = "HTML5 Geolocation is not supported in your browser.";  
    }  
}
```

在示例中，loadDemo 函数测试了浏览器的支持情况，这个函数是在页面加载的时候被调用的。如果存在地理定位对象，navigator.geolocation 调用将返回该对象，否则将触发错误。页面上预先定义的 support 元素会根据检测结果显示支持情况的提示信息。

## 4.4.2 位置请求

目前，有两种类型的位置要求：

- 单次定位请求；
- 重复性的位置更新请求。

### 1. 单次定位请求

在许多应用中，只检索或请求一次用户位置即可。例如，如果要查询在接下来的一个小时内放映某大片的最近的电影院，就可以使用代码清单 4-2 所示的简单的 HTML5 Geolocation API。

#### 代码清单 4-2 单次定位请求

```
void getCurrentPosition(in PositionCallback successCallback,  
                        in optional PositionErrorCallback errorCallback,  
                        in optional PositionOptions options);
```

让我们详细分析下这个核心函数调用。

首先，因为这个函数是通过 navigator.geolocation 对象调用的，所以在脚本中需要先取得此对象。如前所述，应确保有一个合适的后备函数，用来应对浏览器不支持 HTML5 Geolocation 的情况。这个函数接受一个必选参数和两个可选参数。

- 函数参数 successCallback 为浏览器指明位置数据可用时应调用的函数。因为像获取位置数据这样的操作可能需要较长时间才能完成，所以这个参数很重要。没有用户希望在检索位置时浏览器被锁定，也没有开发人员希望他的程序无限期暂停（特别是要成功取得位置信息，经常必须等待用户的许可）。successCallback 是收到实际位置信息并进行处理的地方。
- 跟绝大多数编程场景一样，最好提前准备出错处理。位置信息请求很可能因为一些不可控因素失败，对于这些情况，你可能需要提供一个用于跟用户解释或者提示其重试的 errorCallback 函数。虽然此参数是可选的，不过建议选用。
- 最后，options 对象可以调整 HTML5 Geolocation 服务的数据收集方式。这是一个可选参数，我们随后会详细介绍它。

假设在页面上已经创建了一个名为 updateLocation() 的 JavaScript 函数，它使用最新的位置数据更新页面内容。同样地，也创建了一个 handleLocationError() 函数来处理错误情况。接下来，我们将研究这些函数的细节，而请求访问用户位置的核心代码如下所示：

```
navigator.geolocation.getCurrentPosition(updateLocation, handleLocationError);
```

#### ● updateLocation() 函数

updateLocation() 函数是做什么的呢？实际上非常简单。只要浏览器具备访问位置信息的条件，就会调用 updateLocation() 函数，该函数只接受一个参数：位置对象。这个对象包含坐标 (coords 特性) 和一个获取位置数据时的时间戳。在实际开发中不一定需要时间戳，重要的位置数据都包含在了 coords 特性中。

坐标总是有多个特性，但是浏览器和用户的硬件设备会决定这些特性值是否有意义。以下是



前三个特性：

- latitude (纬度)
- longitude (经度)
- accuracy (准确度)

毋庸置疑，这些特性的数据是必需的。latitude 和 longitude 将包含 HTML5 Geolocation 服务测定的最佳的十进制用户位置。accuracy 将以 m 为单位指定纬度和经度值与实际位置间的差距，置信度为 95%。局限于 HTML5 Geolocation 的实现方式，位置只能是粗略的近似值。在呈现返回值前请一定要检查返回值的准确度。如果推荐的所谓“附近的”鞋店，其实要耗费用户几小时的路程，可能会产生意想不到的后果。

坐标还有一些其他特性，不能保证浏览器都为其提供支持，但如果不支持就会返回 null：

- altitude —— 用户位置的海拔高度，以 m 为单位；
- altitudeAccuracy —— 海拔高度的准确度，也是以 m 为单位，如果不支持 altitude 特性也会返回 null；
- heading —— 行进方向，相对于正北而言；
- speed —— 地面速度，以 m/s 为单位。

除非确定用户的设备能够访问这些信息，否则建议应用程序不要过于依赖于它们。全球定位设备可能提供这种细节信息，而网络三角定位则不会。

现在，让我们了解一下 updateLocation() 函数的实现代码（参见代码清单 4-3），该函数根据坐标信息执行具体的更新操作。

#### 代码清单 4-3 updateLocation() 函数使用示例

```
function updateLocation(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    var accuracy = position.coords.accuracy;

    document.getElementById("latitude").innerHTML = latitude;
    document.getElementById("longitude").innerHTML = longitude;
    document.getElementById("accuracy").innerHTML = "This location is accurate within " +
    accuracy + " meters. "
}
```

示例中，updateLocation() 用来更新 HTML 页面上三个控件元素的文本内容。我们把 longitude 参数的值赋给经度 (longitude) 元素，将 latitude 参数的值赋给纬度 (latitude) 元素。

#### ● handleLocationError() 函数

因为位置计算服务很可能出差错，所以对于 HTML5 Geolocation 应用程序来说错误处理非常重要。幸运的是，该 API 定义了所有需要处理的错误情况的错误编号。错误编号设置在错误对象中，错误对象作为 code 参数传递给错误处理程序。让我们依次来看看这些错误编号。

- UNKNOWN\_ERROR (错误编号为 0) —— 不包括在其他错误编号中的错误。需要通过 message 参数查找错误的更多详细信息。

- PERMISSION\_DENIED (错误编号为 1) ——用户选择拒绝浏览器获得其位置信息。
- POSITION\_UNAVAILABLE (错误编号为 2) ——尝试获取用户位置数据，但失败了。
- TIMEOUT (错误编号为 3)：设置了可选的 timeout 值。尝试确定用户位置的过程超时。

在这些情况下，你可能想让用户知道应用程序运行出了问题，而在获取失败和请求超时的时

候，你可能希望重试。  
代码清单 4-4 是一个错误处理示例程序。

代码清单 4-4 使用错误处理函数

```
function handleLocationError(error) {
    switch(error.code){
        case 0:
            updateStatus("There was an error while retrieving your location: " +
                error.message);
            break;

        case 1:
            updateStatus("The user prevented this page from retrieving a location.");
            break;

        case 2:
            updateStatus("The browser was unable to determine your location: " +
                error.message);
            break;

        case 3:
            updateStatus("The browser timed out before retrieving the location.");
            break;
    }
}
```

访问 error 对象的 code 参数可以得到错误编号，而 message 特性可提供更详细的问题说明。在所有情况下，我们都调用自己的程序来使用自定义的信息更新页面状态。

#### ● 可选的地理定位请求特性

如果要同时处理正常情况和错误情况，就应该把注意力集中到三个可选参数 (enableHighAccuracy、timeout 和 maximumAge) 上，将这三个可选参数传递给 HTML5 Geolocation 服务以调整数据收集方式。请注意，这三个参数可以使用 JSON 对象传递，这样更便于添加到 HTML5 Geolocation 请求调用中。

- enableHighAccuracy：如果启用该参数，则通知浏览器启用 HTML5 Geolocation 服务的高精度模式。参数的默认值为 false。如果启用此参数，可能没有任何差别，也可能会导致机器花费更多的时间和资源来确定位置，所以应谨慎使用。

---

**提示** 需要注意的是，高精度的设置只是一个切换操作：true或false。API不允许把精度设置为不同的等级或一个数字范围。也许，这将在未来版本的规范中得到解决。

---



- timeout: 可选值, 单位为 ms, 告诉浏览器计算当前位置所允许的最长时间。如果在这个时间段内未完成计算, 就会调用错误处理程序。其默认值为 Infinity, 即无穷大或无限制。
- maximumAge: 这个值表示浏览器重新计算位置的时间间隔。它也是一个以 ms 为单位的值。此值默认为零, 这意味着浏览器每次请求时必须立即重新计算位置。

**提示** 大家可能会感到困惑, timeout和maximumAge两者有什么区别呢? 虽然它们的名称相似, 但用法不同。timeout是指计算位置数据所用的时间, 而maximumAge涉及计算位置数据的频率。任何超过timeout的单次计算时间, 都会触发错误。不过, 若浏览器没有在maximumAge设定时间之内更新过的数据, 它就必须重新获取。这里的极限情况: 如果将maximumAge设置为“0”则浏览器在每次请求时都需要重新获取数据, 如果将其设置为Infinity则意味着不再重新获取数据。

但是, 请注意 API 不允许我们为浏览器指定多长时间重新计算一次位置信息。这是完全由浏览器的实现所决定的。我们所能做的就是告诉浏览器 maximumAge 的返回值是什么。实际频率是一个我们无法控制的细节。

现在更新我们的位置请求, 让其包含一个使用 JSON 对象表示的可选参数, 如下所示:

```
navigator.geolocation.getCurrentPosition(updateLocation, handleLocationError,  
                                         {timeout:10000});
```

这个调用告诉 HTML5 Geolocation, 任何处理时间超过 10 s (10 000 ms) 的位置请求都应该触发一个错误, 在这种情况下, 将调用 handleLocationError 函数处理编号对应 TIMEOUT 的错误。

## 2. 重复性的位置更新请求

有时候, 仅更新一次是不够的。还好, Geolocation 服务的设计者使应用程序可以在单次请求用户位置和以既定间隔多次请求用户位置间相互转换。事实上, 转换的方式很简单, 只需要变换请求函数即可, 如下所示:

```
navigator.geolocation.getCurrentPosition(updateLocation, handleLocationError);  
navigator.geolocation.watchPosition(updateLocation, handleLocationError);
```

通过对代码进行简单的修改, 只要用户位置发生变化, Geolocation 服务就会调用 updateLocation 处理程序。它的效果就像是程序在监视用户的位置, 并会在其变化时及时通知用户一样。

这样做有什么意义呢?

假设有一个页面, 随着观察者在城镇周围的移动, 网页上的方向指示也随之改变。再假设一个关于加油站的页面, 随着用户开车在高速公路上持续行驶, 页面不断更新显示最近的加油站。另外, 还可以在一个页面中记录和传送用户位置来实现回溯已走路线。如果位置信息一发生变化就能传递给应用程序, 那么前面所假设的所有服务都会变得很容易实现。

关闭更新也很简单。如果应用程序不再需要接收有关用户的持续位置更新, 则只需调用 clearWatch() 函数, 如下所示:



```
navigator.geolocation.clearWatch(watchId);
```

这个函数会通知 Geolocation 服务，程序不想再接收用户的位置更新。但是，watchID 是什么，它从何而来呢？其实它是 watchPosition() 函数的返回值。watchID 表示一个唯一的监视请求以便将来取消监视。所以，如果应用程序要停止接收位置更新信息，可以参照代码清单 4-5 编写代码。

#### 代码清单 4-5 watchPosition 的使用

```
var watchId = navigator.geolocation.watchPosition(updateLocation, handleLocationError);
// 基于持续更新的位置信息实现一些功能

// .....

// OK, 现在我们可以停止接收位置更新信息了
navigator.geolocation.clearWatch(watchId);
```

## 4.5 使用 HTML5 Geolocation 构建实时应用

到目前为止，我们主要讨论了单次定位请求。接下来我们使用多次请求特性构建一个简单有用的 Web 应用程序——距离跟踪器，通过此应用程序可以了解到 HTML5 Geolocation API 的强大之处。

要想快速确定在一定时间内的行走距离，通常可以使用 GPS 导航系统或计步器这样的专用设备。基于 HTML5 Geolocation 提供的强大服务，开发人员可以创建一个网页来跟踪从网页被加载的地方到目前所在位置所经过的距离。虽然它在台式机上不大实用，但是对于手机是很理想的，而目前可接入网络的支持全球定位的手机有数百万部。只要在手机浏览器中打开这个示例页面并授予其位置访问的权限，每隔几秒钟，应用程序就会根据刚才走过的距离更新，并将其增加到总距离中（如图 4-4 所示）。

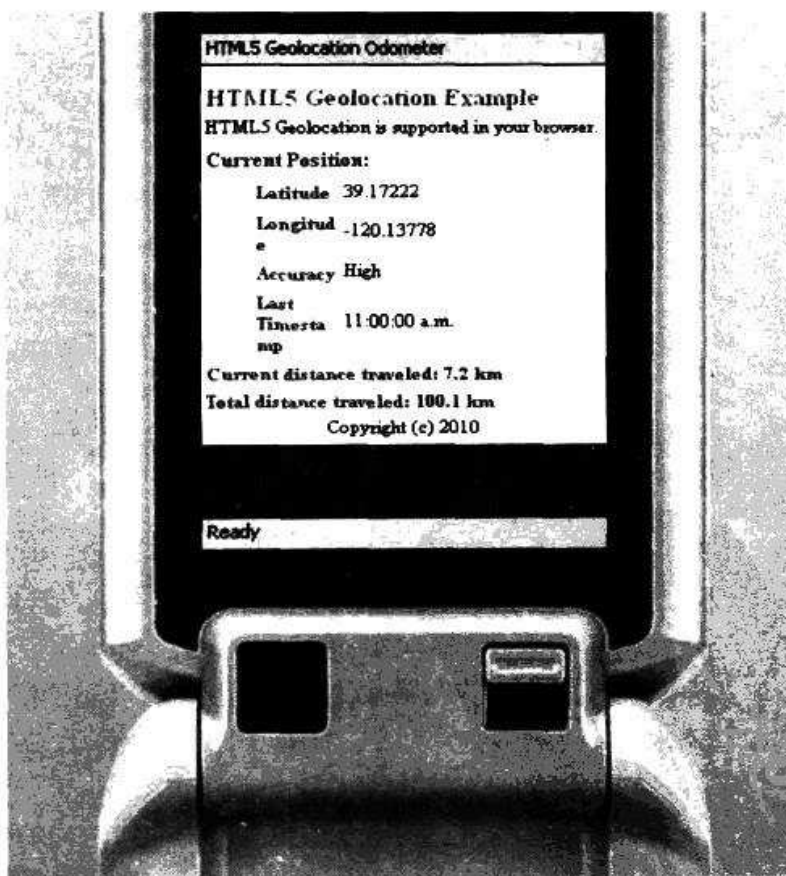


图 4-4 HTML5 Geolocation 示例应用程序运行界面





此示例使用的 `watchPosition()` 函数刚刚讨论过。每当有新的位置返回，就将其与最后保存的位置进行比较以计算距离。距离计算使用了众所周知的 Haversine 公式来实现，这个公式能够根据经纬度来计算地球上两点间的距离。如代码清单 4-6 所示：

代码清单 4-6 Haversine 公式

$$\Delta\sigma = 2\arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos\phi_1 \cos\phi_2 \sin^2\left(\frac{\Delta\lambda}{2}\right)}\right)$$

本书不会介绍 Haversine 公式的原理，不过，我们会演示公式的 JavaScript 实现，你可以使用这段 JavaScript 代码来计算地球上两个位置之间的距离（见代码清单 4-7）。

代码清单 4-7 JavaScript 实现的 Haversine 公式

```
function toRadians(degree) {
    return degree * Math.PI / 180;
}

function distance(latitude1, longitude1, latitude2, longitude2) {
    // R 是地球的半径，以 km 为单位
    var R = 6371;

    var deltaLatitude = toRadians(latitude2 - latitude1);
    var deltaLongitude = toRadians(longitude2 - longitude1);
    latitude1 = toRadians(latitude1);
    latitude2 = toRadians(latitude2);

    var a = Math.sin(deltaLatitude/2) *
        Math.sin(deltaLatitude/2) +
        Math.cos(latitude1) *
        Math.cos(latitude2) *
        Math.sin(deltaLongitude/2) *
        Math.sin(deltaLongitude/2);

    var c = 2 * Math.atan2(Math.sqrt(a),
        Math.sqrt(1-a));
    var d = R * c;
    return d;
}
```

对于这个公式的原理，请查阅中学数学教材。针对目标，我们编写了将角度转换成弧度的方法，还提供了 `distance()` 函数来计算两个经纬度表示的位置间的距离。

如果定期检查用户的位置并计算移动距离，就可以给出以时间为坐标轴的合理的近似移动距离。这需要基于一种假设，即用户在每个区间上都是直线移动。我们的示例就是这样假设的。

#### 4.5.1 编写 HTML 显示代码

首先，我们要创建示例程序的 HTML 显示代码。这部分代码我们做了简化，因为我们真正感兴趣的是处理数据的脚本。我们把所有最新的数据以简单的表格形式展示，分行显示纬度、经

度、准确度和以 ms 为单位的时间戳。此外，我们还会显示一些文本类统计信息，以使用户能够看到走过距离的概要情况（见代码清单 4-8）。

代码清单 4-8 距离跟踪器的 HTML 网页代码

```
<h1>HTML5 Geolocation Distance Tracker</h1>

<p id="status">HTML5 Geolocation is <strong>not</strong> supported in your browser.</p>

<h2>Current Position:</h2>
<table border="1">
  <tr>
    <th width="40" scope="col">Latitude</th>
    <td width="114" id="latitude">?</td>
  </tr>
  <tr>
    <td> Longitude</td>
    <td id="longitude">?</td>
  </tr>
  <tr>
    <td>Accuracy</td>
    <td id="accuracy">?</td>
  </tr>
  <tr>
    <td>Last Timestamp</td>
    <td id="timestamp">?</td>
  </tr>
</table>

<h4 id="currDist">Current distance traveled: 0.0 km</h4>
<h4 id="totalDist">Total distance traveled: 0.0 km</h4>
```

表中的数值都是默认值。开始接收到数据后，应用程序会进行相应的更新。

## 4.5.2 处理 Geolocation 数据

在 Geolocation 数据处理部分，第一段 JavaScript 代码应该看起来很熟悉了。之前我们设置过一个处理程序 `loadDemo()`，它会在页面加载完成的时候执行。这个脚本会检测浏览器是否支持 HTML5 Geolocation，然后将检测结果显示在页面上。最后，代码会请求监测用户位置，如代码清单 4-9 所示。

代码清单 4-9 添加 `loadDemo()` 方法

```
var totalDistance = 0.0;
var lastLat;
var lastLong;

function updateStatus(message) {
  document.getElementById("status").innerHTML = message;
}
```



```
function loadDemo() {
    if(navigator.geolocation) {
        updateStatus("HTML5 Geolocation is supported in your browser.");
        navigator.geolocation.watchPosition(updateLocation,
            handleLocationError,
            {maximumAge:20000});
    }
}
```

```
window.addEventListener("load", loadDemo, true);
```

对于错误处理，我们使用之前提到过的那段代码（对于距离跟踪器而言，这些出错处理已经足够用了）。我们将检查收到的所有错误编号，并更新页面上的状态信息，如代码清单 4-10 所示。

#### 代码清单 4-10 添加出错处理代码

```
function handleLocationError(error) {
    switch (error.code) {
        case 0:
            updateStatus("There was an error while retrieving your location: " + error.message);
            break;

        case 1:
            updateStatus("The user prevented this page from retrieving a location.");
            break;

        case 2:
            updateStatus("The browser was unable to determine your location: " + error.message);
            break;

        case 3:
            updateStatus("The browser timed out before retrieving the location.");
            break;
    }
}
```

需要注意的是，我们将位置监测中的 `maximumAge` 选项设置为：`{maximumAge:20000}`。这将告诉定位服务，所有缓存的位置数据的生命期都不能大于 20 s（或 20 000 ms）。设置此选项可以使页面处于定期更新状态。我们可以随意调整参数的大小，并试着同时调整缓存的大小。

我们的大部分工作都将在 `updateLocation()` 函数中实现，此函数中我们将使用最新数据来更新页面并计算路程，如代码清单 4-11 所示。

#### 代码清单 4-11 添加 `updateLocation()` 函数

```
function updateLocation(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    var accuracy = position.coords.accuracy;
    var timestamp = position.timestamp;

    document.getElementById("latitude").innerHTML = latitude;
    document.getElementById("longitude").innerHTML = longitude;
```

```
document.getElementById("accuracy").innerHTML = accuracy;
document.getElementById("timestamp").innerHTML = timestamp;
```

和期望的效果一样，收到一组最新坐标数据后，首先要做的是记录所有信息。我们收集纬度、经度、准确度和时间戳，然后将这些数据更新到表格中。

应用程序可能选择不显示时间戳。时间戳主要供程序使用，它对最终用户没有多大意义，所以可以替换成更便于用户识别的时间指示器，或将其完全删除。

准确度是以 m 为单位的，乍一看它似乎没有什么使用价值。然而，任何数据都依赖于其准确度。即使不显示给用户，也应该在代码中考虑准确度。显示不准确的值会向用户提供错误的位置信息。因此，我们将过滤掉所有低精度的位置更新数据，如代码清单 4-12 所示。

#### 代码清单 4-12 忽略不准确的更新

```
//合理性检测……，如果 accuracy 值太大，就不要计算距离
if (accuracy >= 500) {
    updateStatus("Need more accurate values to calculate distance.");
    return;
}
```

#### 最简单的旅行方式

“保持位置跟踪的精度至关重要。作为开发人员，无法选择浏览器计算位置所使用的方法，但是可以保证数据的准确度。所以推荐使用 accuracy 特性！”

慵懒的下午，我坐在后院吊床上，在支持地理定位功能的手机浏览器上监测我的位置。我很惊讶地看到，在几分钟内，我只是身体倾斜了一下，但手机上却显示我已经以不同的速度行走了半公里。这听起来可能很可笑，但却提醒我们，数据的准确性与其来源密切相关。”

——Brian

最后，我们将计算移动的距离。假设我们前面已经至少收到了一个准确的位置。我们将更新移动的总距离并将其显示给用户，同时还将储存当前值以备后面做比较。为了使我们的界面不会太混乱，在计算数值时可采用四舍五入或截断的方式，如代码清单 4-13 所示。

#### 代码清单 4-13 添加计算距离的代码

```
// 计算距离
if ((lastLat !== null) && (lastLong !== null)) {
    var currentDistance = distance(latitude, longitude, lastLat, lastLong);
    document.getElementById("currDist").innerHTML =
        "Current distance traveled: " + currentDistance.toFixed(4) + " km";

    totalDistance += currentDistance;

    document.getElementById("totalDist").innerHTML =
        "Total distance traveled: " + totalDistance.toFixed(4) + " km";
}
```



```
        lastLat = latitude;  
        lastLong = longitude;  
  
        updateStatus("Location successfully updated.");  
    }  
}
```

示例代码的内容就这么多！包括错误处理在内，代码总共不到 200 行，在这么简短的 HTML 和脚本中，我们构建了一个能够持续监测用户位置变化的示例应用程序，几乎完整地演示了 Geolocation API 的使用。尽管示例不适用于台式机，但是不妨在支持地理定位功能的手机或移动设备上实践一下，看看你一天大概能走多远的路。

### 4.5.3 最终代码

代码清单 4-14 显示了完整的示例代码。

代码清单 4-14 完整的距离跟踪器代码

```
<!DOCTYPE html>  
<html>  
<title>HTML5 Geolocation Odometer</title>  
<h1>HTML5 Geolocation Distance Tracker</h1>  
  
<p id="status">HTML5 Geolocation is <strong>not</strong> supported in your browser.</p>  
  
<h2>Current Position:</h2>  
<table border="1">  
<tr>  
<th width="40" scope="col">Latitude</th>  
<td width="114" id="latitude">?</td>  
</tr>  
<tr>  
<td> Longitude</td>  
<td id="longitude">?</td>  
</tr>  
<tr>  
<td>Accuracy</td>  
<td id="accuracy">?</td>  
</tr>  
<tr>  
<td>Last Timestamp</td>  
<td id="timestamp">?</td>  
</tr>  
</table>  
  
<h4 id="currDist">Current distance traveled: 0.0 km</h4>  
<h4 id="totalDist">Total distance traveled: 0.0 km</h4>  
  
<script type="text/javascript">  
  
    var totalDistance = 0.0;  
    var lastLat;
```

```
var lastLong;

function toRadians(degree) {
    return degree * Math.PI / 180;
}

function distance(latitude1, longitude1, latitude2, longitude2) {
    //R 是地球的半径, 单位为 km
    var R = 6371;
    var deltaLatitude = toRadians(latitude2-latitude1);
    var deltaLongitude = toRadians(longitude2-longitude1);
    latitude1 = toRadians(latitude1);
    latitude2 = toRadians(latitude2);

    var a = Math.sin(deltaLatitude/2) *
    Math.sin(deltaLatitude/2) +
    Math.cos(latitude1) *
    Math.cos(latitude2) *
    Math.sin(deltaLongitude/2) *
    Math.sin(deltaLongitude/2);

    var c = 2 * Math.atan2(Math.sqrt(a),
    Math.sqrt(1-a));
    var d = R * c;
    return d;
}

function updateStatus(message) {
    document.getElementById("status").innerHTML = message;
}

function loadDemo() {
    if(navigator.geolocation) {
        updateStatus("HTML5 Geolocation is supported in your browser.");
        navigator.geolocation.watchPosition(updateLocation, handleLocationError,
        {maximumAge:20000});
    }
}

window.addEventListener("load", loadDemo, true);

function updateLocation(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    var accuracy = position.coords.accuracy;
    var timestamp = position.timestamp;

    document.getElementById("latitude").innerHTML = latitude;
    document.getElementById("longitude").innerHTML = longitude;
    document.getElementById("accuracy").innerHTML = accuracy;
    document.getElementById("timestamp").innerHTML = timestamp;

    //合理性检测……如果 accuracy 值太大, 就不要计算距离
    if (accuracy >= 500) {
        updateStatus("Need more accurate values to calculate distance.");
    }
}
```



```
        return;
    }

    //计算距离
    if ((lastLat !== null) && (lastLong !== null)) {
        var currentDistance = distance(latitude, longitude, lastLat, lastLong);
        document.getElementById("currDist").innerHTML =
            "Current distance traveled: " +
            currentDistance.toFixed(4) + " km";

        totalDistance += currentDistance;

        document.getElementById("totalDist").innerHTML =
            "Total distance traveled: " + totalDistance.toFixed(4) + " km";
    }

    lastLat = latitude;
    lastLong = longitude;

    updateStatus("Location successfully updated.");
}

function handleLocationError(error) {
    switch (error.code) {
        case 0:
            updateStatus("There was an error while retrieving your location: " +
                error.message);
            break;

        case 1:
            updateStatus("The user prevented this page from retrieving a location.");
            break;

        case 2:
            updateStatus("The browser was unable to determine your location: " +
                error.message);
            break;

        case 3:
            updateStatus("The browser timed out before retrieving the location.");
            break;
    }
}
</script>
</html>
```

## 4.6 进阶功能

尽管有些技巧我们在示例中用不上，但这并不妨碍它们在多种类型的 HTML5 Web 应用中发挥作用。本节，我们就来介绍一些简单实用的进阶功能。

### 4.6.1 现在的状态是什么

大家可能已经注意到了，HTML5 Geolocation API 通常涉及时计。不足为奇，因为手机三角测量、GPS、IP 查询等定位技术都可能需要较长时间才能完成。幸运的是，API 为开发人员提供了大量的信息，可以利用这些信息为用户创建一个合理的状态显示栏。

如果开发人员在定位函数中设置了可选的 timeout 值，那么地理定位服务获取位置信息超时的时候，就会向用户弹出错误提示信息。更合理的效果是当请求还在进行时，在用户页面上告知用户当前的状态信息。状态从提出请求开始，不论请求成功或失败，状态的结束都与 timeout 的值对应。

在代码清单 4-15 中，我们启用 JavaScript 间隔计时器以定期使用新的进度值更新显示的状态。

代码清单 4-15 添加状态栏

```
function updateStatus(message) {
    document.getElementById("status").innerHTML = message;
}

function endRequest() {
    updateStatus("Done.");
}

function updateLocation(position) {
    endRequest();
    // 处理位置数据
}

function handleLocationError(error) {
    endRequest();
    // 处理错误
}

navigator.geolocation.getCurrentPosition(updateLocation, handleLocationError,
                                         {timeout:10000});
// 超时时间为 10s

updateStatus("Requesting location data...");
```

我们详细分析一下这段代码。同以前一样，我们已经有了用于更新页面状态值的函数，如下所示。

```
function updateStatus(message) {
    document.getElementById("status").innerHTML = message;
}
```

这里使用了简单的文本方式来显示状态信息（参见代码清单 4-16），实际应用中可以使用更吸引人的图形来显示状态。



**代码清单 4-16 显示状态**

```
navigator.geolocation.getCurrentPosition(updateLocation, handleLocationError,  
                                        {timeout:10000});  
  
    //超时时间为 10s  
  
    updateStatus("Requesting location data...");
```

我们再次使用 Geolocation API 来获取用户的当前位置，但此时将 timeout 设置为 10 s。10 s 后，由于 timeout 选项的原因，结果可能是成功或者失败。

我们会马上更新状态的文本显示，告诉用户该位置请求正在进行中。然后，如果请求在 10 s 内完成了，或者请求未完成但时间超过了 10 s，都会使用回调方法来重置状态文本，如代码清单 4-17 所示。

**代码清单 4-17 重置状态文本**

```
function endRequest() {  
    updateStatus("Done.");  
}  
function updateLocation(position) {  
    endRequest();  
  
    //处理位置数据  
}
```

这个示例功能虽然简单，但易于扩展。

调用 getCurrentPosition() 会启动单次定位查找请求。开发人员可以轻易地决定何时发起位置查找请求。不过，在通过 watchPosition() 重复查找位置的情况下，开发人员就无法控制何时发起每个位置请求了。

此外，计时是从用户授权访问位置数据开始的。由于无法在用户授权时立即获知其准确时间，所以也就无法做到精确显示状态。

## 4.6.2 在 Google Map 上显示“我在这里”

一个非常常见的利用位置数据的应用是在诸如 Google Map 中显示用户的位置。由于实际需求量很大，Google 将 HTML5 Geolocation 内置到了其用户界面中。只需按下“Show My Location”按钮（见图 4-5），Google Map 就将使用 Geolocation API（如果可用）在地图上检测并显示用户的位置。

开发人员也可以自己实现同样的功能。Google Map API 超出了本书范围，但是我们只需知道它的经纬度位置表示方式也是按照十进制格式设计的（并非巧合）。因此，开发人员可以将查找到的用户位置结果传递给 Google Map API，如代码清单 4-18 所示。更多此方面的内容可以阅读 *Beginning Google Maps Application Second Edition* (Apress, 2010)。



图 4-5 Google 地图中的“Show My Location”按钮

#### 代码清单 4-18 将位置传递给 Google Map API

```
// 创建 Google Map……详细的使用说明请参考 Google API
var map = new google.maps.Map(document.getElementById("map"));

function updateLocation(position) {
    // 将位置信息传递给 Google Map, 并将其居中
    map.setCenter(new google.maps.LatLng(position.coords.latitude, position.coords.longitude));
}

navigator.geolocation.getCurrentPosition(updateLocation, handleLocationError);
```

## 4.7 小结

本章讨论了 HTML5 Geolocation, 讲述了 HTML5 Geolocation 的位置信息——纬度、经度和其他特性, 以及获取它们的途径。我们还探讨了由 HTML5 Geolocation 引发的隐私问题。最后演示了如何使用 HTML5 Geolocation API 创建引人注目的位置感知 Web 应用程序。

下一章, 我们将演示如何基于 HTML5 Communication 实现标签页间及窗口间通信, 以及如何实现页面和服务器的跨域通信。



**本章**，我们将探讨用于构建实时（real-time）跨源（cross-origin）通信的两个重要模块：跨文档消息通信（Cross Document Messaging）和 XMLHttpRequest Level 2。通过它们，我们可以构建引人注目的 Web 应用。作为 HTML5 应用新的通信手段，这两个构建块可以让不同域间的 Web 应用安全地进行通信。

首先，我们要讨论 HTML5 通信规范中的两个主要元素：postMessage API 和源安全（origin security）概念，之后将会展示如何用 postMessage API 在 iframe、标签页和窗口间进行通信。

接下来，我们会探讨 XMLHttpRequest Level 2——XMLHttpRequest 的改进版，介绍 XMLHttpRequest 在哪些方面得到了改进，并会特别介绍如何用 XMLHttpRequest 发起跨源请求以及如何使用新的进度事件（progress event）。

## 5.1 跨文档消息通信

出于安全方面的考虑，运行在同一浏览器中的框架、标签页、窗口间的通信一直都受到了严格的限制。例如，在浏览器内部共享信息对某些站点可能比较方便，但是同时也增加了受到恶意攻击的可能性。如果浏览器允许程序访问加载到其他框架和标签的内容，某些网站就能够利用脚本来窃取其他网站的某些信息。浏览器厂商合理地限制了这类访问，当尝试检索或修改从其他源加载的内容时，浏览器会抛出安全异常，并阻止相应的操作。

然而，现实中存在一些合理的让不同站点的内容能在浏览器内进行交互的需求。Mashup 就是最典型的一个例子，它是各种不同应用的结合体，把来自不同站点的地图、聊天、新闻等应用全部整合到一起形成一个新的元应用（meta-application）。在这种情形下，如果浏览器内部能提供直接的通信机制，就能更好地组织这些应用。

为了满足上述需求，浏览器厂商和标准制定机构一致同意引入一种新功能：跨文档消息通信。跨文档消息通信可以确保 iframe、标签页、窗口间安全地进行跨源通信。它把 postMessage API 定义为发送消息的标准方式。利用 postMessage 发送消息非常简单，代码如下所示：

```
chatFrame.contentWindow.postMessage('Hello, world', 'http://www.example.com/');
```

接收消息时仅需在页面中增加一个事件处理函数。当某个消息到达时，通过检查消息的来源来决定是否对这条消息进行处理。代码清单 5-1 描述了一个可以把消息传递给 messageHandler()

函数的事件监听器。

#### 代码清单 5-1 消息事件的监听器

```
window.addEventListener("message", messageHandler, true);
function messageHandler(e) {
    switch(e.origin) {
        case "friend.example.com":
            // 处理消息
            processMessage(e.data);
            break;
        default:
            // 消息来源无法识别
            // 消息被忽略
    }
}
```

消息事件是一个拥有 data（数据）和 origin（源）属性的 DOM 事件。data 属性是发送方传递的实际消息，而 origin 属性是发送来源。有了 origin 属性，接收方能轻易地忽略掉来自不可信源的消息；根据可信源的列表能方便地判断来源是否可靠。

如图 5-1 所示，postMessage API 提供了一种交互方式，使得不同源的 iframe（源为 http://chat.example.net 网站的聊天部件）可以与其父页面（源为 http://portal.example.com，包含这里提及的聊天部件）进行通信。

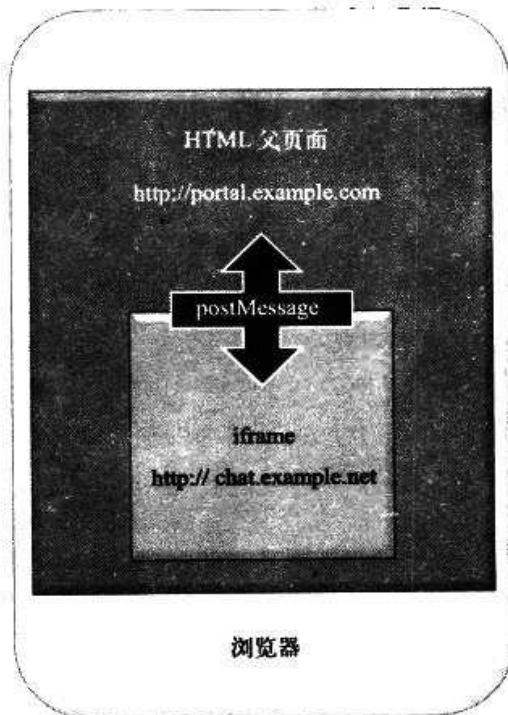


图 5-1 iframe 与其父页面间的 postMessage 通信

这个示例中，聊天部件被包含在一个 iframe 中，因此不能直接访问父窗口。当聊天部件接收



到聊天消息时，它可以用 `postMessage` 向父页面发送一个消息，以便父页面提醒聊天部件的用户接收到了新消息。同理，父页面也能将用户的状态发送给聊天部件。父页面和部件通过把彼此的源加到可信源的白名单中，就都能收到来自对方的信息。

在 `postMessage` 之前，`iframe` 间的通信有时会通过直接写脚本来实现。通过执行一个页面中的脚本来尝试操纵另一个文件。这种方式可能会因为安全限制而被禁止，因而 `postMessage` 取代了直接编程访问，它提供了 Javascript 环境中的异步通信机制。如图 5-2 所示，假如没有 `postMessage`，跨源通信将导致安全错误，为防止跨站点（cross-site）脚本攻击，浏览器会强制引发这种安全错误。

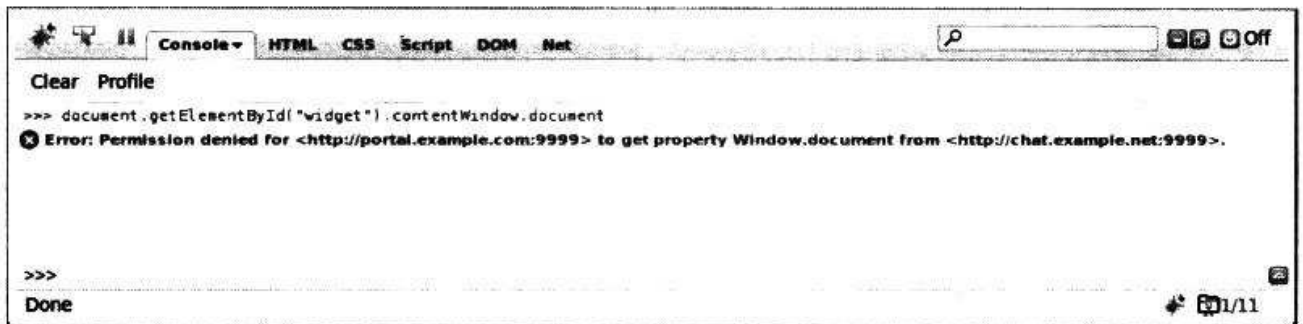


图 5-2 Firefox 和 Firebug 早期版本中的跨站点脚本错误

`postMessage` API 不仅可以胜任同源文档间的通信，而且在浏览器不允许非同源通信的情况下，`postMessage` API 也很有用。鉴于它的一致性和易用性，在同源文档间通信时也推荐使用 `postMessage`。在 JavaScript 环境的通信中始终应使用 `postMessage` API，例如使用 HTML5 Web Workers 通信时。

### 5.1.1 理解源安全

HTML5 通过引入源（origin）的概念对域安全进行了阐明和改进。源是在网络上用来建立信任关系的地址的子集。源由规则（scheme）、主机（host）、端口（port）组成。例如，由于规则不同（如 `https` 与 `http`），所以 `https://www.example.com` 页面与 `http://www.example.com` 页面的源是不同的。源的概念中不考虑路径，如 `http://www.example.com/index.html` 与 `http://www.example.com/page2.html` 有相同的源，因为它们只是路径不同，其他完全相同。

HTML5 定义了源的序列化。源在 API 和协议中以字符串的形式出现。这对于使用 `XMLHttpRequest` 进行跨源 HTTP 请求是非常重要的，对于 `WebSocket` 也一样。

跨源通信通过源来确定发送者，这就使得接收方可以忽略或者拒绝来自不可信源的消息。同时，各种应用必须加入事件监听器以接收消息，从而避免被不可信应用程序的信息所干扰。

`postMessage` 的安全规则确保了消息不会被传递到非预期的源页面中。当发送消息时，由发送方指定接收方的源。如果发送方用来调用 `postMessage` 的窗口不具有特定的源（例如用户跳转到了其他站点），浏览器就不会传送消息。

类似地，接收信息的时候，发送方的源也被作为消息的一部分。为避免伪造，消息源由浏览器提供。接收方可以决定处理哪些消息，以及忽略哪些消息。我们可以保留一份白名单，告诉浏览器仅仅处理可信源的消息。

### 谨慎对待外部输入

“在处理跨源通信的消息时，一定要验证每个消息的源。此外，处理消息中的数据时也应该谨慎。即使消息来自可信源，也应该像对待外部输入时一样仔细。下面是两个关于内容注入的示例，一个是可能带来麻烦的方法，而另一个则是较为安全的方法。”

```
//危险：e.data 会被当成标记！
element.innerHTML = e.data;
```

```
//相对安全
element.textContent = e.data;
```

最好永远不要对来自第三方的字符串求值。再者，要避免使用 eval 方法处理应用内部的字符串。可以通过 window.JSON 或者 json.org 解析器使用 JSON。JSON 是一种数据格式，它能在 JavaScript 中安全使用，而且 json.org 解析器也设计得非常严谨。”

——Frank

5

## 5.1.2 跨文档消息通信的浏览器支持情况

在本书编写的时候，HTML5 的跨文档消息通信已经被很多浏览器支持了，如表 5-1 所示。

表5-1 HTML5跨文档消息通信的浏览器支持情况

浏览器	支持情况
Chrome	2.0及以上版本支持
Firefox	3.0及以上版本支持
Internet Explorer	8.0及以上版本支持
Opera	9.6及以上版本支持
Safari	4.0及以上版本支持

在使用前，我们还是应该先检测浏览器是否支持 HTML5 的跨文档消息通信。本章后面的“浏览器支持情况检测”部分将演示如何编写代码来检查浏览器的支持性。

## 5.1.3 使用 postMessage API

本节将更详细地介绍 HTML5 postMessage API 的使用。

### 1. 浏览器支持情况检测

在调用 postMessage 前，应该首先检测浏览器是否支持它。下面就是一种检测是否支持 postMessage 的方法：



```
if (typeof window.postMessage === undefined) {  
    // 该浏览器不支持 postMessage  
}
```

## 2. 发送消息

通过调用目标页面 window 对象中的 postMessage() 函数可发送消息，代码如下：

```
window.postMessage("Hello, world", "portal.example.com");
```

第一个参数包含要发送的数据，第二个参数是消息传送的目的地。要发送消息给 iframe，可以在相应 iframe 的 contentWindow 中调用 postMessage，代码如下：

```
document.getElementsByTagName("iframe")[0].contentWindow.postMessage("Hello, world",  
"chat.example.net");
```

## 3. 监听消息事件

脚本可以通过监听 window 对象中的事件来接收信息。在事件监听函数中，接收方可以决定接收或者忽略消息，见代码清单 5-2。

代码清单 5-2 监听消息事件并通过白名单鉴定源

```
var originWhitelist = ["portal.example.com", "games.example.com", "www.example.com"];  
  
function checkWhitelist(origin) {  
    for (var i=0; i<originWhitelist.length; i++) {  
        if (origin === originWhitelist[i]) {  
            return true;  
        }  
    }  
    return false;  
}  
  
function messageHandler(e) {  
    if(checkWhitelist(e.origin)) {  
        processMessage(e.data);  
    } else {  
        //忽略来自未知源的消息  
    }  
}  
window.addEventListener("message", messageHandler, true);
```

---

**提示** HTML5 定义的 MessageEvent 接口也是 HTML5 WebSockets 和 HTML5 Web Workers 的一部分。HTML5 的通信功能中用于接收消息的 API 与 MessageEvent 接口是一致的。其他通信类 API，如 EventSource API 和 Web Workers，也都使用 MessageEvent 接口来传递消息。

---

### 5.1.4 使用 postMessage API 创建应用

现在，我们将创建前面提到过的包含跨源聊天部件的门户应用。可以利用跨文档消息通信来实现门户页面和聊天部件之间的交互，如图 5-3 所示。

## Cross-Origin Portal

Origin: `http://portal.example.com:9999`

Status: `Out for a run!`

This uses `postMessage` to send a status update to the widget iframe contained in the portal page.

### Widget iframe

Origin: `http://chat.example.net:9999`

Status set to: **Out for a run!** by containing portal.

Runtime:

This will ask the portal to notify the user. The portal does this by flashing the title. If the message comes from an origin other than `http://chat.example.net:9999`, the portal page will ignore it.

图 5-3 门户页面和跨源的聊天部件 iframe

这个示例演示了如何在门户页面中以 `iframe` 方式嵌入第三方的部件。示例中使用了一个来自 `chat.example.net` 的部件。门户页面和部件通过 `postMessage` 来通信。`iframe` 中的聊天部件通过父页面标题内容的闪烁来通知用户。这是后台接收事件的应用中常见的 UI 技术。不过，由于部件不在父页面中，而是被隔离在一个来自不同源的 `iframe` 中，所以直接改变标题会引发安全冲突。因此，部件使用 `postMessage` 请求父页面修改标题。

示例中，门户页面还发送消息到 `iframe` 以通知部件：用户改变了其状态。以这样的方式使用 `postMessage` 可以实现门户与部件之间的协作。由于信息发送时会检查目标源，接收时会检查事件源，所以数据不会意外泄露或被伪造。

**提示** 示例中的聊天部件没有与在线聊天系统连接，发送通知是通过用户单击“Send Notification”按钮来驱动的。在线聊天应用可以通过 Web Sockets 来实现，这将在第 6 章中介绍。

为了便于演示，我们编写了两个简单的 HTML 页面：`postMessagePortal.html` 和 `postMessageWidget.html`。下面将分步演示建立门户页面和聊天部件页面的要点。示例代码位于 `code/communication` 文件夹下。

### 1. 创建门户页面

首先，添加来自不同源的 `iframe` 以包含聊天部件：



```
<iframe id="widget" src="http://chat.example.net:9999/postMessageWidget.html"></iframe>
```

接下来，添加事件监听器 `messageHandler` 监听来自聊天部件的消息事件。从下面的示例代码中可以看到，部件会请求门户去通知用户——闪烁标题。为了确保消息的来源是聊天部件，我们会验证消息源，如果消息不是来源于 `http://chat.example.net:9999`，门户页面会直接忽略它。

```
var targetOrigin = "http://chat.example.net:9999";

function messageHandler(e) {
  if (e.origin === targetOrigin) {
    notify(e.data);
  } else {
    // 忽略来自其他源的消息
  }
}
```

最后，添加一个与聊天部件通信的函数。它用 `postMessage` 发送一个状态，更新门户页面中的部件 `iframe`。对于在线的聊天应用，可以用这种方式来更新用户状态（在线、离开等）。

```
function sendString(s) {
  document.getElementById("widget").contentWindow.postMessage(s, targetOrigin);
}
```

## 2. 创建聊天部件页面

首先，添加事件监听器 `messageHandler` 监听来自门户页面的消息事件。如下面的示例代码所示，聊天部件监听发过来的状态变更消息。为了确保消息来源于门户页面，我们会验证消息源，如果不是来源于 `http://portal.example.com:9999`，部件将直接忽略它。

```
var targetOrigin = "http://portal.example.com:9999";
function messageHandler(e) {
  if (e.origin === "http://portal.example.com:9999") {
    document.getElementById("status").textContent = e.data;
  } else {
    // 忽略其他源发来的消息
  }
}
```

其次，编写用于与门户页面通信的函数。在接收到新消息时，部件将请求门户页面通知用户，并且用 `postMessage()` 函数向门户页面发送消息，代码如下：

```
function sendString(s) {
  window.top.postMessage(s, targetOrigin);
}
```

## 3. 完整代码

代码清单 5-3 是 `postMessagePortal.html` 文件的完整代码。

### 代码清单 5-3 `postMessagePortal.html` 的内容

```
<!DOCTYPE html>
<title> Portal [http://portal.example.com:9999]</title>
<link rel="stylesheet" href="styles.css">
```

```
<style>
  iframe {
    height: 400px;
    width: 800px;
  }
</style>
<link rel="icon" href="http://apress.com/favicon.ico">
<script>
var defaultTitle = "Portal [http://portal.example.com:9999]";
var notificationTimer = null;

var targetOrigin = "http://chat.example.net:9999";

function messageHandler(e) {
  if (e.origin == targetOrigin) {
    notify(e.data);
  } else {
    //忽略其他源发来的消息
  }
}

function sendString(s) {
  document.getElementById("widget").contentWindow.postMessage(s, targetOrigin);
}

function notify(message) {
  stopBlinking();
  blinkTitle(message, defaultTitle);
}

function stopBlinking() {
  if (notificationTimer != null) {
    clearTimeout(notificationTimer);
  }
  document.title = defaultTitle;
}

function blinkTitle(m1, m2) {
  document.title = m1;
  notificationTimer = setTimeout(blinkTitle, 1000, m2, m1)
}

function sendStatus() {
var statusText = document.getElementById("statusText").value;
  sendString(statusText);
}

function loadDemo() {
  document.getElementById("sendButton").addEventListener("click", sendStatus, true);
  document.getElementById("stopButton").addEventListener("click", stopBlinking, true);
  sendStatus();
}
window.addEventListener("load", loadDemo, true);
window.addEventListener("message", messageHandler, true);
```



```

</script>

<h1>Cross-Origin Portal</h1>
<p><b>Origin</b>: http://portal.example.com:9999</p>
Status <input type="text" id="statusText" value="Online">
<button id="sendButton">Change Status</button>
<p>
This uses postMessage to send a status update to the widget iframe contained in the portal
page.
</p>
<iframe id="widget" src="http://chat.example.net:9999/postMessageWidget.html"></iframe>
<p>
<button id="stopButton">Stop Blinking Title</button>
</p>

```

代码清单 5-4 是 postMessageWidget.html 页面的完整代码。

#### 代码清单 5-4 postMessageWidget.html 的内容

```

<!DOCTYPE html>
<title>widget</title>
<link rel="stylesheet" href="styles.css">
<script>

var targetOrigin = "http://portal.example.com:9999";

function messageHandler(e) {
  if (e.origin === "http://portal.example.com:9999") {
    document.getElementById("status").textContent = e.data;
  } else {
    //忽略其他源发来的消息
  }
}

function sendString(s) {
  window.top.postMessage(s, targetOrigin);
}

function loadDemo() {
  document.getElementById("actionButton").addEventListener("click",
    function() {
      var messageText = document.getElementById("messageText").value;
      sendString(messageText);
    }, true);
}

window.addEventListener("load", loadDemo, true);
window.addEventListener("message", messageHandler, true);

</script>
<h1>Widget iframe</h1>
<p><b>Origin</b>: http://chat.example.net:9999</p>
<p>Status set to: <strong id="status"></strong> by containing portal.</p>

```

```
<div>
  <input type="text" id="messageText" value="Widget notification.">
  <button id="actionButton">Send Notification</button>
</div>
```

```
<p>
This will ask the portal to notify the user. The portal does this by flashing the title. If
the message comes from an origin other than http://chat.example.net:9999, the portal page
will ignore it.
</p>
```

#### 4. 部署应用

示例应用的运行依赖于两个先决条件：首先，页面需要部署在 Web 服务器上；其次，两个页面必须来自不同的域。如果可以访问位于不同域的多个 Web 服务器（如两台 Apache HTTP 服务器），那么将示例文件部署到服务器之后，示例就能顺利运行了。另外，如果要在本机部署运行示例应用，需要使用 Python 的 SimpleHTTPServer Web 服务器，其安装步骤如下。

(1) 更新 Windows 系统的 hosts 文件（位于 C:\Windows\system32\drivers\etc\hosts）或 Linux 系统的 hosts 文件（位于/etc/hosts），增加两条指向 localhost（IP 地址为 127.0.0.1）的记录，如下所示：

```
127.0.0.1 chat.example.net
127.0.0.1 portal.example.com
```

**提示** hosts文件修改完成后，必须重启浏览器以确保该DNS记录生效。

(2) 安装 Python，其中包括轻量级 SimpleHTTPServer Web 服务器。

(3) 打开示例文件（postMessageParent.html 和 postMessageWidget.html）所在目录。

(4) 按如下方式启动 Web 服务器<sup>①</sup>：

```
python -m SimpleHTTPServer 9999
```

(5) 打开浏览器，输入http://portal.example.com:9999/postMessagePortal.html。现在应该就可以看到图 5-3 所示的页面了。

## 5.2 XMLHttpRequest Level 2

XMLHttpRequest API 使得 Ajax 技术的实现成为了可能。现在市面上有许多关于 XMLHttpRequest 和 Ajax 的图书。更多关于 XMLHttpRequest 编程的知识，建议阅读由 John Resig 撰写的 *Pro JavaScript Techniques*（Apress, 2006）。

作为 XMLHttpRequest 的改进版，XMLHttpRequest Level 2 在功能上有了很大的改进。在本章中，我们将介绍 XMLHttpRequest Level 2 的这些改进，主要集中在以下两个方面：

- 跨源 XMLHttpRequests，
- 进度事件（Progress events）。

<sup>①</sup> 在执行下面这行命令之前，请读者先确认已将 Python 根目录（即 Python.exe 所在目录，如 C:/python27）添加到了系统的 Path 环境变量中。——编者注



## 5.2.1 跨源 XMLHttpRequest

过去, XMLHttpRequest 仅限于同源通信。XMLHttpRequest Level 2 通过 CORS (Cross Origin Resource Sharing, 跨源资源共享) 实现了跨源 XMLHttpRequests。其中源的概念曾在 5.1 节中讨论过。

跨源 HTTP 请求包括一个 Origin 头部, 它为服务器提供 HTTP 请求的源信息。头部由浏览器保护, 不能被应用程序代码更改。从本质上讲, 它与跨文档消息通信中消息事件的 origin 属性作用相同。Origin 头部不同于早先的 Referer [sic] 头部, 因为后者中的 Referer 是一个包括了路径的完整 URL。由于路径可能包含敏感信息, 为了保护用户隐私, 浏览器并不一定会发送 Referer, 而浏览器在任何必要的时候都会发送 Origin 头部。

使用跨源 XMLHttpRequest 可以构建基于非同源服务的 Web 应用程序。例如, 如果 Web 应用程序使用了一个源的静态文本和另一个源的 Ajax 服务, 那么它可以借助跨源 XMLHttpRequest 请求实现在两个源之间的通信, 如果没有跨源 XMLHttpRequest 则只能进行同源通信, 而且部署方式也会受到限制。也许不得不将 Web 应用程序部署在一个单独域中或者再为其建立一个子域。

如图 5-4 所示, 通过跨源 XMLHttpRequest 可以从客户端整合来自不同源的内容。如果目标服务器允许, 可以使用用户证书访问受保护的内容, 进而让用户直接访问个人的数据。反之, 如果通过服务器端对不同源进行整合, 则所有内容都要穿过一个服务器端的基础层, 因而可能会形成瓶颈。

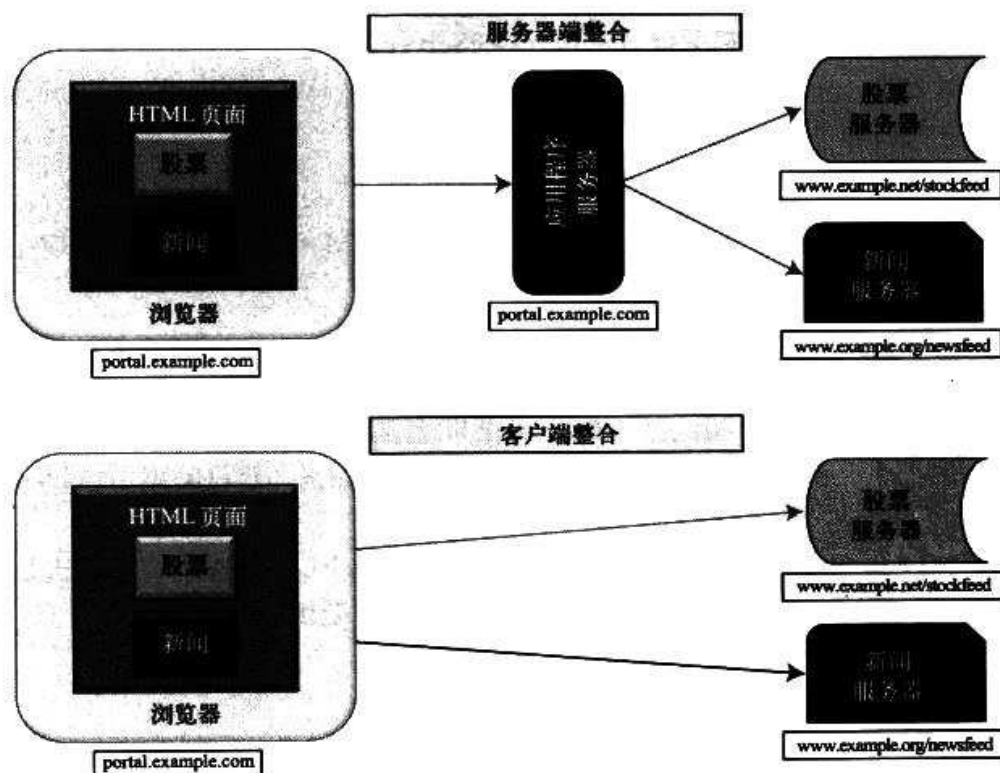


图 5-4 客户端整合与服务器端整合的差异

CORS 规范要求,对一些敏感行为——如申请证书的请求或除了 GET 和 POST 以外的 OPTIONS 预检 (preflight) 请求,必须由浏览器发送给服务器,以确定这种行为能否被支持和允许,这意味着成功通信的背后或许需要由具备 CORS 处理能力的服务器来支持。代码清单 5-5 和代码清单 5-6 展示了托管在 www.example.com 的页面与 www.example.net 的服务之间用于跨源交换的 HTTP 头部。

#### 代码清单 5-5 请求的头部示例

```
POST /main HTTP/1.1
Host: www.example.net
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.9.1.3) Gecko/20090910 Ubuntu/9.04
(jaunty) Shiretoko/3.5.3
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.example.com/
Origin: http://www.example.com
Pragma: no-cache
Cache-Control: no-cache
Content-Length: 0
```

#### 代码清单 5-6 响应的头部示例

```
HTTP/1.1 201 Created
Transfer-Encoding: chunked
Server: Kaazing Gateway
Date: Mon, 02 Nov 2009 06:55:08 GMT
Content-Type: text/plain
Access-Control-Allow-Origin: http://www.example.com
Access-Control-Allow-Credentials: true
```

## 5.2.2 进度事件

新版 XMLHttpRequest 中最重要的 API 改进之一是增加了对进度的响应。在 XMLHttpRequest 之前的版本中,仅有 readystatechange 一个事件能够被用来响应进度。更糟糕的是,浏览器对该事件的实现并不兼容,如在 Internet Explorer 中永远都不会触发 readyState 3<sup>①</sup>。此外,readyState 的更改事件缺乏与上传进程通信的方法。在这种情况下,想要实现上传进度条是一件相当复杂的事情,而且还要牵扯到服务器端的编程开发。

XMLHttpRequest Level 2 用了一个有意义的名字 Progress 进度来命名进度事件。表 5-2 是新的进度事件名称。通过为事件处理程序属性设置回调函数,

表 5-2 XMLHttpRequest Level 2 中的进度事件

进度事件的名称
loadstart
progress
abort
error
load
loadend

① 当 readyState 状态值为 3 时,所有响应头部都已经接收到。响应体开始接收但未完成。——译者注



CORS 规范要求,对一些敏感行为——如申请证书的请求或除了 GET 和 POST 以外的 OPTIONS 预检 (preflight) 请求,必须由浏览器发送给服务器,以确定这种行为能否被支持和允许,这意味着成功通信的背后或许需要由具备 CORS 处理能力的服务器来支持。代码清单 5-5 和代码清单 5-6 展示了托管在 `www.example.com` 的页面与 `www.example.net` 的服务之间用于跨源交换的 HTTP 头部。

代码清单 5-5 请求的头部示例

```
POST /main HTTP/1.1
Host: www.example.net
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.9.1.3) Gecko/20090910 Ubuntu/9.04
(jaunty) Shiretoko/3.5.3
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.example.com/
Origin: http://www.example.com
Pragma: no-cache
Cache-Control: no-cache
Content-Length: 0
```

代码清单 5-6 响应的头部示例

```
HTTP/1.1 201 Created
Transfer-Encoding: chunked
Server: Kaazing Gateway
Date: Mon, 02 Nov 2009 06:55:08 GMT
Content-Type: text/plain
Access-Control-Allow-Origin: http://www.example.com
Access-Control-Allow-Credentials: true
```

## 5.2.2 进度事件

新版 XMLHttpRequest 中最重要的 API 改进之一是增加了对进度的响应。在 XMLHttpRequest 之前的版本中,仅有 `readystatechange` 一个事件能够被用来响应进度。更糟糕的是,浏览器对该事件的实现并不兼容,如在 Internet Explorer 中永远都不会触发 `readyState 3`<sup>①</sup>。此外, `readyState` 的更改事件缺乏与上传进程通信的方法。在这种情况下,想要实现上传进度条是一件相当复杂的事情,而且还要牵扯到服务器端的编程开发。

XMLHttpRequest Level 2 用了一个有意义的名字 `Progress` 进度来命名进度事件。表 5-2 是新的进度事件名称。通过为事件处理程序属性设置回调函数,

表 5-2 XMLHttpRequest Level 2 中的进度事件

进度事件的名称
<code>loadstart</code>
<code>progress</code>
<code>abort</code>
<code>error</code>
<code>load</code>
<code>loadend</code>

① 当 `readyState` 状态值为 3 时,所有响应头部都已经接收到。响应体开始接收但未完成。——译者注

```
document.getElementById("support").innerHTML =  
    "Your browser <strong>does</strong>support cross-origin XMLHttpRequest";  
}
```

## 2. 构建跨源请求

为了构建跨源 XMLHttpRequest，首先要创建一个新的 XMLHttpRequest 对象，代码如下：

```
var crossOriginRequest = new XMLHttpRequest()
```

接下来，通过指定不同源的地址来构造跨源 XMLHttpRequest，代码如下：

```
crossOriginRequest.open("GET", "http://www.example.net/stockfeed", true);
```

在请求过程中务必确保能够监听到错误。请求不成功有很多原因，如网络故障、访问被拒、目标服务器缺乏对 CORS 支持等。

## 3. 使用进度事件

在表示请求和响应的不同阶段方面，XMLHttpRequest Level 2 不再使用数值型状态表示法，而是提供了命名进度事件。为事件处理程序属性设置相应的回调函数后，就可以对这些事件进行监听了。

代码清单 5-8 显示了如何用回调函数来处理进度事件。进度事件使用了多个文本域，分别记录待发送数据的总量、已发送数据的总量，以及用于标识数据总量是否已知的布尔值（流式 HTTP 中可能无法获知数据总量）。XMLHttpRequest.upload 调度事件也用到了这些文本域。

### 代码清单 5-8 使用 onprogress 事件

```
crossOriginRequest.onprogress = function(e) {  
    var total = e.total;  
    var loaded = e.loaded;  
  
    if (e.lengthComputable) {  
        //利用进度信息做些事情  
    }  
}  
  
crossOriginRequest.upload.onprogress = function(e) {  
    var total = e.total;  
    var loaded = e.loaded;  
  
    if (e.lengthComputable) {  
        //利用进度信息做些事情  
    }  
}
```

## 5.2.5 创建 XMLHttpRequest 应用

在这个示例中，我们要将赛事位置坐标上传到非同源的 Web 服务器端，并使用新的进度事件监控包括上传进度在内的 HTTP 请求的状态。示例运行效果如图 5-5 所示。



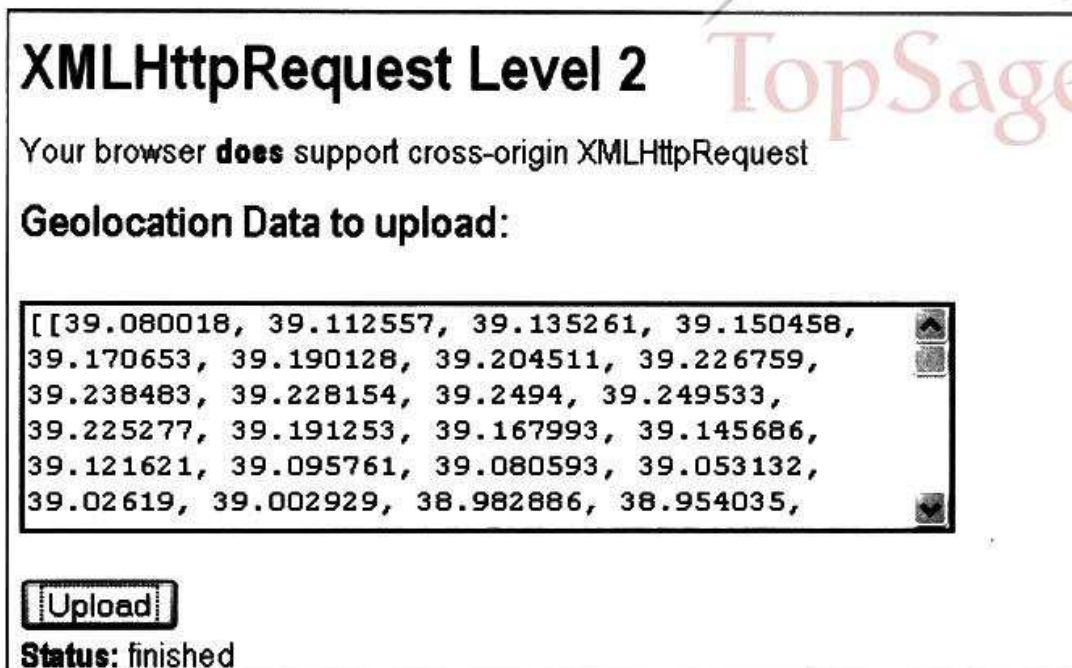


图 5-5 上传位置数据的 Web 应用

为方便说明，我们创建了 HTML 文件 `crossOriginUpload.html`。下面的步骤揭示了图 5-5 所示的跨源上传页面的重要组成部分。示例代码位于 `code/communication` 文件夹下。

首先，创建一个新的 XMLHttpRequest 对象：

```
var xhr = new XMLHttpRequest();
```

接下来，检查浏览器是否支持跨源 XMLHttpRequest，代码如下：

```
if (typeof xhr.withCredentials === undefined) {
    document.getElementById("support").innerHTML =
        "Your browser <strong>doesnot</strong> support cross-origin XMLHttpRequest";
} else {
    document.getElementById("support").innerHTML =
        "Your browser <strong>does</strong> support cross-origin XMLHttpRequest";
}
```

然后，设置回调函数以处理进度事件，并计算上传和下载的完成率。

```
xhr.upload.onprogress = function(e) {
    var ratio = e.loaded / e.total;
    setProgress(ratio + "% uploaded");
}

xhr.onprogress = function(e) {
    var ratio = e.loaded / e.total;
    setProgress(ratio + "% downloaded");
}

xhr.onload = function(e) {
```

```
    setProgress("finished");  
  }  
  
  xhr.onerror = function(e) {  
    setProgress("error");  
  }  
}
```

最后，打开请求并发送包含编码后的地理位置数据的字符串。由于目标位置的 URL 与当前页面不同源，所以这是一个跨源请求。

```
var targetLocation = "http://geodata.example.net:9999/upload";  
xhr.open("POST", targetLocation, true);  
  
geoDataString = dataElement.textContent;  
xhr.send(geoDataString);
```

### 1. 完整代码

代码清单 5-9 是 crossOriginUpload.html 文件的完整代码。

#### 代码清单 5-9 crossOriginUpload.html 文件

```
<!DOCTYPE html>  
<title>Upload Geolocation Data</title>  
<link rel="stylesheet" href="styles.css">  
<link rel="icon" href="http://apress.com/favicon.ico">  
<script>  
function loadDemo() {  
  var dataElement = document.getElementById("geodata");  
  dataElement.textContent = JSON.stringify(geoData).replace(", ", ", ", "g");  
  
  var xhr = new XMLHttpRequest()  
  if (typeof xhr.withCredentials == undefined) {  
    document.getElementById("support").innerHTML =  
      "Your browser <strong>does not</strong> support cross-origin XMLHttpRequest";  
  } else {  
    document.getElementById("support").innerHTML =  
      "Your browser <strong>does</strong> support cross-origin XMLHttpRequest";  
  }  
  
  var targetLocation = "http://geodata.example.net:9999/upload";  
  
  function setProgress(s) {  
    document.getElementById("progress").innerHTML = s;  
  }  
  
  document.getElementById("sendButton").addEventListener("click",  
    function() {  
      xhr.upload.onprogress = function(e) {  
        var ratio = e.loaded / e.total;  
        setProgress(ratio + "% uploaded");  
      }  
  
      xhr.onprogress = function(e) {  
        var ratio = e.loaded / e.total;
```



```

        setProgress(ratio + "% downloaded");
    }

    xhr.onload = function(e) {
        setProgress("finished");
    }

    xhr.onerror = function(e) {
        setProgress("error");
    }

    xhr.open("POST", targetLocation, true);

    geoDataString = dataElement.textContent;
    xhr.send(geoDataString);
}, true);
}
window.addEventListener("load", loadDemo, true);

</script>

<h1>XMLHttpRequest Level 2</h1>
<p id="support"></p>

<h4>Geolocation Data to upload:</h4>
<textarea id="geodata">
</textarea>
</div>

<button id="sendButton">Upload</button>

<script>
geoData = [[39.080018000000003, 39.112557000000002, 39.135261, 39.150458,
39.170653000000001, 39.190128000000001, 39.204510999999997, 39.226759000000001,
39.238483000000002, 39.228154000000004, 39.249400000000001, 39.249533, 39.225276999999998,
39.191253000000003, 39.167993000000003, 39.145685999999998, 39.121620999999998,
39.095761000000003, 39.080593, 39.053131999999998, 39.02619, 39.002929000000002,
38.982886000000001, 38.954034999999998, 38.944926000000002, 38.919960000000003,
38.925261999999996, 38.934922999999998, 38.949373000000001, 38.950133999999998,
38.952649000000001, 38.969692000000002, 38.988512999999998, 39.010652, 39.033088999999997,
39.053493000000003, 39.072752999999999], [-120.15724399999999, -120.15818299999999, -
120.156004000000001, -120.14564599999999, -120.141285, -120.108899000000001, -
120.095285000000002, -120.077596, -120.045428, -120.0119, -119.988971000000002, -
119.951240999999998, -119.932700999999998, -119.927131, -119.92685999999999, -
119.926362000000001, -119.928446000000001, -119.911036, -119.942834, -119.944130000000002, -
119.945552000000001, -119.954110000000001, -119.941327, -119.946059000000001, -
119.975275999999999, -119.99445, -120.028998, -120.066335, -120.078673000000001, -120.089985,
-120.112227, -120.097907000000001, -120.108810000000001, -120.116692, -120.117847, -
120.117278999999998, -120.143981999999999]];
</script>

<p>
    <b>Status: </b> <span id="progress">ready</span>
</p>

```

## 2. 部署应用

示例代码依赖于两个前提条件：首先，页面不能同域；其次，目标页面必须由能够解析 CORS 头部的 Web 服务器来提供。本章的示例代码中包括支持 CORS 的 Python 脚本，它可以处理传入的跨源的 XMLHttpRequests。你可以在本机按以下步骤进行配置。

(1) 更新主机文件（Windows 中的 C:\Windows\system32\drivers\etc\hosts，Unix/Linux 中的 /etc/hosts 文件），添加如下两个指向 localhost（IP 地址为 127.0.0.1）的项：

```
127.0.0.1 geodata.example.net
127.0.0.1 portal.example.com
```

**提示** hosts文件修改完成后，必须重启浏览器以确保该DNS记录生效。

(2) 如果在学习之前的示例时没有安装 Python，则需要安装包括了轻量级 SimpleHTTPServer Web 服务器的 Python 环境。

(3) 打开包含示例文件（crossOriginUpload.html）和 CORS 系统服务器 Python 脚本（CORSServer.py）的目录。

(4) 运行 Python 脚本：

```
python CORSServer.py 9999
```

(5) 打开浏览器，输入 `http://portal.example.com:9999/crossOriginUpload.html` 后即可看到图 5-5 所示的页面。

## 5.3 进阶功能

尽管有些技巧我们在示例中用不上，但这并不妨碍它们在多种类型的 HTML5 Web 应用中发挥作用。本节，我们就来介绍一些简单实用的进阶功能。

### 5.3.1 结构化的数据

早期版本的 `postMessage` 仅支持字符串。后来的版本支持 JavaScript 对象、`canvas imageData` 和文件等其他数据类型。由于不同浏览器对规范支持的差异，对不同的对象类型的支持情况也不同。

在一些浏览器中，对借由 `postMessage` 发送的 JavaScript 对象的限制同对 JSON 数据的限制是相同的。具体来讲，可能不允许循环数据结构，如循环链表。

### 5.3.2 Framebusting

Framebusting 技术可以用来保证某些内容不被加载到 `iframe` 中。应用程序首先检测其所在的窗口是否为最外层的窗口（`window.top`），若不是则跳脱包含它的框架，代码如下所示：

```
if (window !== window.top) {
    window.top.location = location;
}
```



不过，你可能会希望借助 `iframe` 导入一些确定的合作网站页面来充实自身的内容。这里有一种解决方案，即使用 `postMessage` 实现 `iframe` 与其父页面间的握手通信，如代码清单 5-10 所示。

代码清单 5-10 在 `iframe` 中使用 `postMessage` 实现互信页面握手通信

```
var framebustTimer;
var timeout = 3000; // 超时时间设为 3 s

if (window !== window.top) {
    framebustTimer = setTimeout(
        function() {
            window.top.location = location;
        }, timeout);
}

window.addEventListener("message", function(e) {
    switch(e.origin) {
        case trustedFramer:
            clearTimeout(framebustTimer);
            break;
    }
}, true);
```

## 5.4 小结

本章主要介绍了如何使用跨文档消息通信和 `XMLHttpRequest Level 2` 创建引人注目的应用，这些应用能够安全地进行跨源通信。

首先，我们讨论了 `postMessage` 和源安全的概念，这是 `HTML5 Communication` 规范的两个关键元素。接下来，我们演示了如何利用 `postMessage` API 实现 `iframe`、标签页和窗口间通信。

随后，我们讨论了 `XMLHttpRequest Level 2`。作为 `XMLHttpRequest` 的升级版，它对 `XMLHttpRequest` 进行了多方面的改进，而最为重要的是对于 `readystatechange` 事件方面的改进。然后，我们介绍了如何使用 `XMLHttpRequest` 创建跨源请求，以及如何使用新的进度事件。

最后，我们用一些实例结束了本章的讲解。下一章，我们将介绍 `HTML5 WebSockets`，基于它能够轻松高效地将实时数据流推送给 Web 应用。

**本**章我们将研究 HTML5 WebSockets 的使用方法。HTML5 WebSockets 是 HTML5 中最强大的通信功能，它定义了一个全双工通信信道，仅通过 Web 上的一个 Socket 即可进行通信。WebSockets 不仅仅是对常规 HTTP 通信的另一种增量加强，它更代表着一次巨大的进步，对实时的、事件驱动的 Web 应用程序而言更是如此。

过去要在浏览器中实现全双工连接，必须通过迂回的“hacks”来模拟实现，相比之下 HTML5 WebSockets 带来的改进是如此之大，以至于 HTML5 规范的领军人物——Google 工程师 Ian Hickson 都说：

“数据从几千字节减少到了两字节，延迟从 150ms 减少到了 50ms——这不可小看。实际上，仅仅这两个因素已经足以引起 Google 对 WebSockets 的兴趣了。”

——[www.ietf.org/mail-archive/web/hybi/current/msg00784.html](http://www.ietf.org/mail-archive/web/hybi/current/msg00784.html)

HTML5 WebSockets 为何能够带来如此显著的改善？接下来我们会详细讨论，同时还会看到 HTML5 WebSockets 是如何一举淘汰传统的 Comet 和 Ajax 轮询 (polling)、长轮询 (long-polling) 以及流 (streaming) 解决方案的。

## 6.1 HTML5 WebSockets 概述

让我们与 HTTP 解决方案比一比，看看在全双工实时浏览器通信中，HTML5 WebSockets 是如何减少不必要的网络流量并降低网络延迟的。

### 6.1.1 实时和 HTTP

正常情况下，浏览器访问 Web 页面时，一般会向页面所在的 Web 服务器发送一个 HTTP 请求。Web 服务器识别请求，然后返回响应。大多数情况下，如股票价格、新闻报道、余票查询、交通状况、医疗设备读取数据等，当内容呈现在浏览器页面上时，可能已经没有时效性。如果用户想要获得最新的实时信息，就需要不断地手动刷新页面，这显然不是一个明智的做法。

目前实时 Web 应用的实现方式，大部分是围绕轮询和其他服务器端推送技术展开的，其中最著名的是 Comet。Comet 技术可以让服务器端主动以异步方式向客户端推送数据，它会使针对



传输消息到客户端的响应延迟完成。

使用轮询时，浏览器会定期发送 HTTP 请求，并随即接收响应。这项技术是浏览器在实时信息传送方面的首次尝试。显然，如果知道消息传递的准确时间间隔，轮询将是一个很好的办法，因为可以将客户端的请求同步为只有服务器上的信息可用时才发出。但是，实时数据往往不可预测，不可避免会产生一些不必要的请求，在低消息率情况下会有很多无用的连接不断地打开和关闭。

使用长轮询时，浏览器向服务器发送一个请求，服务器会在一段时间内将其保持在打开状态。如果服务器在此期间收到一个通知，就会向客户端发送一个包含消息的响应。如果时间已到却还没收到通知，服务器会发送一个响应消息来终止打开的请求。然而，最关键的是，当信息量很大时，与传统轮询方式相比，长轮询方式并无实质上的性能改善。

使用流解决方案时，浏览器会发送一个完整的 HTTP 请求，但服务器会发送并保持一个处于打开状态的响应，该响应持续更新并无限期（或是一段时间内）处于打开状态。每当有消息可发送时该响应就会被更新，但服务器永远不会发出响应完成的信号，这样连接就会一直保持在打开状态以便后续消息的发送。但是，由于流仍是封装在 HTTP 中，其间的防火墙和代理服务器可能会对响应消息进行缓冲，造成消息传递的时延。因此，当检测到缓冲代理服务器时，许多流解决方案就回退到长轮询方式。此外，可利用 TLS (SSL) 连接来保护响应不被缓冲，但在这种情况下，每个连接的创建和清除会消耗更多的服务器资源。

综上所述，所有这些提供实时数据的方式都会涉及 HTTP 请求和响应报头，其中包含有大量额外的、不必要的报头数据，会造成传输延迟。最重要的是，全双工连接需要的不仅仅是服务器到客户端的下行连接。为了在半双工 HTTP 的基础上模拟全双工通信，目前的许多解决方案都使用了两个连接：一个用于下行数据流，另一个用于上行数据流。这两个连接的保持和协作也会造成大量的资源消耗，并增加了复杂度。简而言之，HTTP 技术并不是为了实现实时全双工通信设计的。从图 6-1 也可以看出来，基于半双工 HTTP，构建一个采用发布/订阅模式来利用后端数据源显示实时数据的 Web 应用程序是比较复杂的。

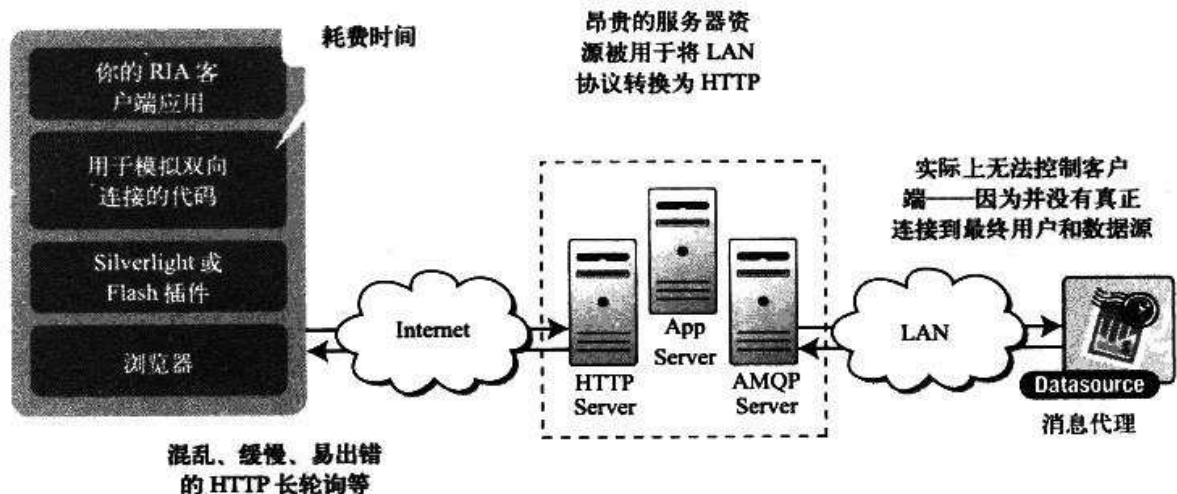


图 6-1 实时 HTTP 应用程序的复杂性



当开发人员试图对上述方案继续扩展时，情况会变得更糟。模拟基于 HTTP 的双向浏览器通信是非常复杂和易错的，而且复杂度不可控。虽然最终用户感觉 Web 应用像是实时的，但是这种“实时”体验的代价非常高，包括额外的时间延迟、不必要的网络流量和 CPU 性能消耗。

## 6.1.2 解读 HTML5 WebSockets

一开始 HTML5 规范的首席作者 Ian Hickson 将 WebSockets 在规范的 Communications 章节中定义成了 TCPConnection，随着规范的演进，后来才改名为 WebSockets。现在，就像 Geolocation、Web Workers 一样，为了明确主题，WebSockets 已成为一个独立的规范。

### WebSocket 与火车模型有何联系？

“Ian Hickson 是火车模型的爱好者，他一直在寻找一种通过计算机控制火车模型的方法。这种想法自 1984 年 Marklin 首次推出了数字控制器就开始了，比 Web 的出现时间还要早很多。

在 Ian 将 TCPConnection 引入到 HTML5 规范时，他正致力于通过浏览器来控制火车模型，通过经典的 pre-WebSocket “hanging GET” 和 XHR 技术来实现浏览器和火车之间的通信。如果当时浏览器中有 socket 通信可以实现（就像“胖”客户端上那样的异步客户端/服务器通信模式）的话，火车控制器程序就会简单得多。因此，来自一切皆有可能的灵感，（火车）车轮已经开动，而且 WebSocket 火车已经离站。下一站：实时 Web。”

——Peter

6

### 1. WebSocket握手

为了建立 WebSocket 通信，客户端和服务端在初始握手时，将 HTTP 协议升级到了 WebSocket 协议，如代码清单 6-1 所示。请注意，该连接是在草案 76 (Draft 76) 中描述的，它在后续规范修订中还可能会发生变化。

#### 代码清单 6-1 WebSocket 升级握手

```
从客户端到服务器：
GET /demo HTTP/1.1
Host: example.com
Connection: Upgrade
Sec-WebSocket-Key2: 12998 5 Y3 1 .P00
Sec-WebSocket-Protocol: sample
Upgrade: WebSocket
Sec-WebSocket-Key1: 4@1 46546xW%01 1 5
Origin: http://example.com

[8-byte security key]

从服务器到客户端：
HTTP/1.1 101 WebSocket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
```



```
WebSocket-Origin: http://example.com
WebSocket-Location: ws://example.com/demo
WebSocket-Protocol: sample
[16-byte hash response]
```

一旦连接建立成功，就可以在全双工模式下在客户端和服务端之间来回传送 WebSocket 消息。这就意味着，在同一时间、任何方向，都可以全双工发送基于文本的消息。在网络中，每个消息以 0x00 字节开头，以 0xFF 结尾，中间数据采用 UTF-8 编码格式。

## 2. WebSocket 接口

除了对 WebSocket 协议的定义外，该规范还同时定义了用于 JavaScript 应用程序的 WebSocket 接口，如代码清单 6-2 所示。

代码清单 6-2 WebSocket 接口

```
[Constructor(in DOMString url, in optional DOMString protocol)]
interface WebSocket {
  · readonly attribute DOMString URL;

  //就绪状态
  const unsigned short CONNECTING = 0;
  const unsigned short OPEN = 1;
  const unsigned short CLOSED = 2;
  readonly attribute unsigned short readyState;
  readonly attribute unsigned long bufferedAmount;

  //网络
  attribute Function onopen;
  attribute Function onmessage;
  attribute Function onclose;
  boolean send(in DOMString data);
  void close();
};
WebSocket implements EventTarget;
```

WebSocket 接口的使用很简单。要连接远程主机，只需要新建一个 WebSocket 实例，提供希望连接的对端 URL。注意，ws://和 wss://前缀分别表示 WebSocket 连接和安全的 WebSocket 连接。

基于同一底层 TCP/IP 连接，在客户端和服务端之间的初始握手阶段，将 HTTP 协议升级至 WebSocket 协议，WebSocket 连接就建立完成了。连接一旦建立，WebSocket 数据帧就可以以全双工模式在客户端和服务端之间进行双向传送。连接本身是通过 WebSocket 接口定义的消息事件和 send 函数来运作的。在代码中，采用异步事件侦听器来控制连接生命周期的每一个阶段。

```
myWebSocket.onopen = function(evt) { alert("Connection open ..."); };
myWebSocket.onmessage = function(evt) { alert("Received Message: " + evt.data); };
myWebSocket.onclose = function(evt) { alert("Connection closed."); };
```

## 3. 大幅削减不必要的网络流量和时延

那么，WebSocket 效率究竟有多高？我们通过逐项对比轮询应用程序和 WebSocket 应用程序来说明。这里以一个简单 Web 应用程序为例来演示轮询方式，页面使用传统的发布/订阅模式从 RabbitMQ 消息代理（message broker）请求实时股票信息，通过轮询 Web 服务器上的 Java Servlet

来运作。RabbitMQ 消息代理从一个虚拟的、股票价格持续更新的数据源接收数据。网页连接并注册一个特定的股票频道（消息代理中的一项），使用 XMLHttpRequest 进行轮询，更新频率为每秒一次。当收到更新时，经过一些计算，股票数据就会显示在表格中，如图 6-2 所示。










COMPANY	SYMBOL	PRICE	CHANGE	SPARKLINE	OPEN	LOW	HIGH
THE WALT DISNEY COMPANY	DIS	27.65	0.56		27.09	24.39	29.80
GARMIN LTD.	GRMN	35.14	0.35		34.79	31.31	38.27
SANBISK CORPORATION	SNOK	20.11	-0.13		20.24	18.22	22.26
GOODRICH CORPORATION	GR	49.99	-2.35		52.34	47.11	57.57
NVIDIA CORPORATION	NVDA	13.92	0.07		13.85	12.47	15.23
CHEVRON CORPORATION	CVX	67.77	-0.53		68.30	61.49	75.11
THE ALLSTATE CORPORATION	ALL	30.88	-0.14		31.02	27.92	34.12
EXXON MOBIL CORPORATION	XOM	65.66	-0.86		66.52	59.87	73.17
METLIFE INC.	MET	35.58	-0.15		35.73	32.16	39.30

图 6-2 一个 JavaScript 股票行情应用程序

看起来似乎不错，但从另一个角度考察，就会发现这个应用程序其实存在很严重的问题。例如，在带有 Firebug 插件的 Firefox 浏览器中，你可以看到每秒都会有一个 GET 请求发往服务器。从下方的 HTTP 报头数据中可以发现，每个请求关联的报头开销相当惊人。代码清单 6-3 和代码清单 6-4 分别是某次请求和响应的 HTTP 报头数据。

#### 代码清单 6-3 HTTP 请求报头

```
GET /PollingStock//PollingStock HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.1.5) Gecko/20091102
  Firefox/3.5.5
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.example.com/PollingStock/
Cookie: showInheritedConstant=false; showInheritedProtectedConstant=false;
  showInheritedProperty=false; showInheritedProtectedProperty=false;
  showInheritedMethod=false; showInheritedProtectedMethod=false;
  showInheritedEvent=false; showInheritedStyle=false; showInheritedEffect=false
```

#### 代码清单 6-4 HTTP 响应报头

```
HTTP/1.x 200 OK
X-Powered-By: Servlet/2.5
Server: Sun Java System Application Server 9.1_02
Content-Type: text/html;charset=UTF-8
Content-Length: 21
```





Date: Sat, 07 Nov 2009 00:32:46 GMT

做一个有趣的统计(哈哈),我们来统计一下字符数。HTTP 请求和响应报头信息总共包含有 871B 的额外开销,这些开销中不含任何数据。当然,这只是一个示例,实际中的报头数据可能少于 871B,但超过 2 000B 也很常见。在本示例应用程序中,典型的股票消息大约只占到 20 个字符。但从代码中可以看到,这部分信息已经被太多不必要的报头信息所淹没,而这些报头信息完全没必要在第一时接收。

那么,把这种应用程序部署后供大量用户使用会有什么后果呢?让我们看看在三种不同应用场景下,这种轮询应用关联的 HTTP 请求和响应报头数据所需要的网络开销有多大。

- 场景 A: 1 000 客户,每秒轮询一次,网络流量为  $871 \times 1\,000 = 871\,000\text{B} = 6\,968\,000\text{bit/s}$  (6.6Mbit/s)。
- 场景 B: 10 000 客户,每秒轮询一次,网络流量为  $871 \times 10\,000 = 8\,710\,000\text{B} = 69\,680\,000\text{bit/s}$  (66Mbit/s)。
- 场景 C: 100 000 客户,每秒轮询一次,网络流量为  $871 \times 100\,000 = 87\,100\,000\text{B} = 696\,800\,000\text{bit/s}$  (665Mbit/s)。

可见,不必要的网络开销相当庞大。现在我们换用 HTML5 WebSockets 来重构应用程序。在页面中添加一个事件处理程序,用于异步监听来自消息代理的股票更新消息(再加上一点点其他的改动)。每个消息都是一个 WebSocket 帧,只有 2B 的开销(而不是 871B)。让我们再来看看三个应用场景中引入了 WebSocket 后网络开销的变化。

- 场景 A: 1 000 客户,每秒接收一个消息,网络流量为  $2 \times 1\,000 = 2\,000\text{B} = 16\,000\text{bit/s}$  (0.015Mbit/s)。
- 场景 B: 10 000 客户,每秒接收一个消息,网络流量为  $2 \times 10\,000 = 20\,000\text{B} = 160\,000\text{bit/s}$  (0.153Mbit/s)。
- 场景 C: 100 000 客户,每秒接收一个消息,网络流量为  $2 \times 100\,000 = 200\,000\text{B} = 1\,600\,000\text{bit/s}$  (1.526Mbit/s)。

如图 6-3 所示,与轮询方式相比,HTML5 WebSockets 对不必要的网络流量的减少幅度相当惊人。

时延减少情况呢?图 6-4 中上半部分表示半双工轮询方案的时延。举例来说,假设消息从服务器发送到浏览器需要 50 ms。轮询应用程序会引入许多额外的时延,因为只能在响应完成时将新的请求发送到服务器。这个新请求又需要 50 ms,在此期间服务器无法向浏览器发送任何消息,进而造成额外的服务器内存消耗。

从图 6-4 中下半部分可以看出 WebSocket 方案降低时延的情况。一旦连接升级为 WebSocket,消息会在到达服务器后立即返回到浏览器。虽然消息发送到浏览器仍需要 50 ms,但由于 WebSocket 连接始终保持在打开状态,就不需要重复向服务器发送请求了。

HTML5 WebSockets 在实时 Web 的扩展性方面取得了巨大的进步。正如在本章中看到的,根据 HTTP 消息头大小的影响,HTML5 WebSockets 可以实现 500:1 甚至 1 000:1 的 HTTP 消息头流量缩减,并实现 3:1 的通信延迟缩减。

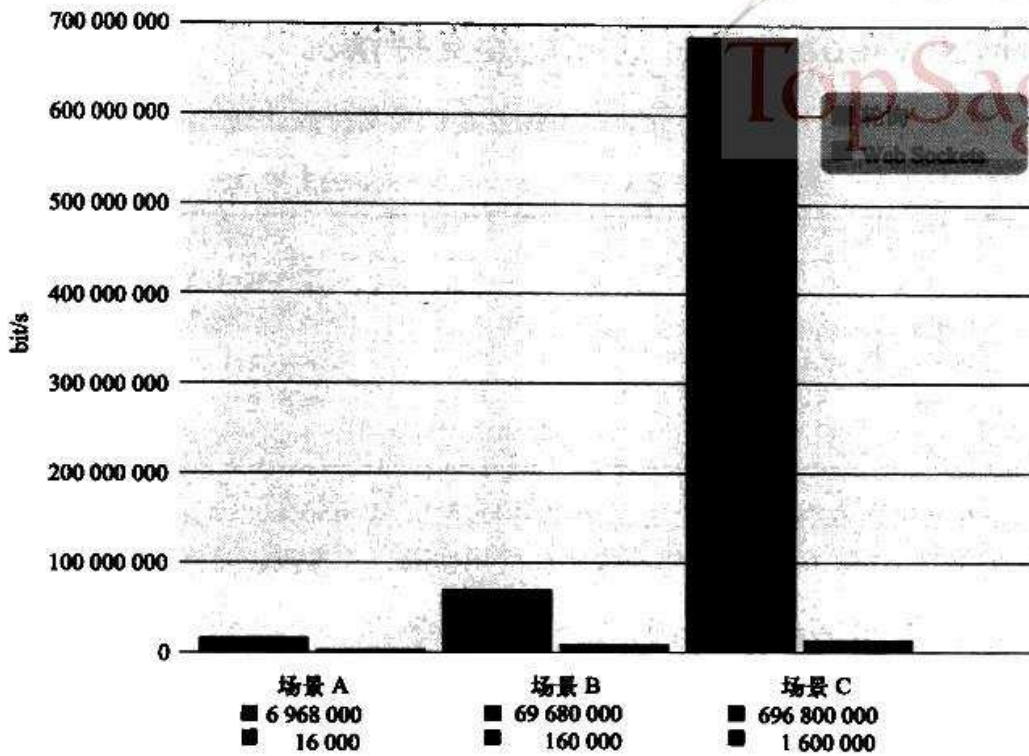


图 6-3 轮询和 WebSocket 应用程序间不必要网络流量的对比

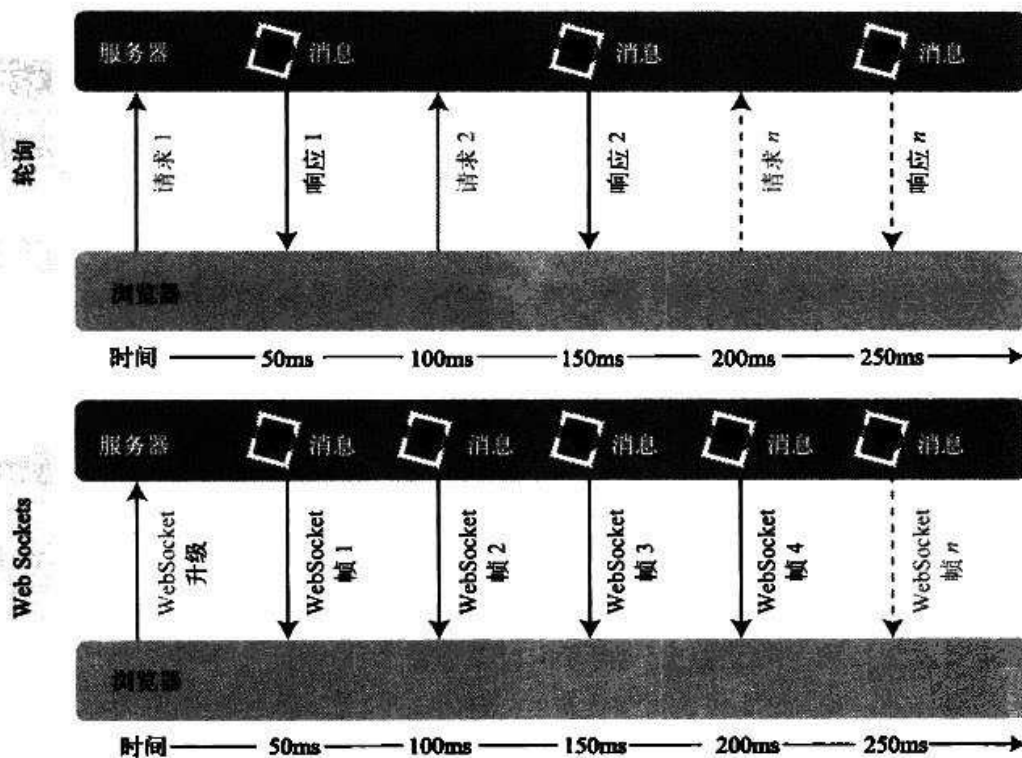


图 6-4 轮询和 WebSocket 应用程序间时延的对比



## 6.2 HTML5 WebSockets 的浏览器支持情况

如表 6-1 所示，在编写本书的时候，以下浏览器已经支持或计划支持 HTML5 WebSockets。

表6-1 浏览器对HTML5 WebSockets的支持情况

浏览器	支持情况
Chrome	版本4以上支持
Firefox	版本4以上支持
Internet Explorer	暂不支持
Opera	暂不支持
Safari	版本5以上支持

由于浏览器支持程度有所不同，所以在使用 HTML5 WebSockets 元素之前，最好先检查一下浏览器是否支持它。本章 6.4.1 小节中将介绍如何用代码方式检测浏览器的支持情况。

## 6.3 编写简单的 Echo WebSocket 服务器

在使用 WebSockets API 之前，开发人员还需要一个支持 WebSocket 的服务器。本节我们会演示如何通过简单的步骤完成 WebSocket “Echo” 服务器的编写。为了运行本章中的示例，我们引入了一个简单的 WebSockets 服务器，它是用 Python 编写的。下面示例的代码位于本书网站上的 WebSockets 部分。

### 现有 WebSocket 服务器

现成的 WebSocket 服务器有很多，还在开发中的就更多了。以下是几种现有的 WebSocket 服务器。

- Kaazing WebSocket Gateway：一种基于 Java 的 WebSocket 网关。
- mod\_pywebsocket：一种基于 Python 的 Apache HTTP 服务器扩展。
- Netty：一种包含 WebSocket 的 Java 框架。
- Node.js：一种驱动多个 WebSocket 服务器的服务器端 JavaScript 框架。

对于非原生支持 WebSocket 的浏览器来说，Kaazing 的 WebSocket 网关包含了完整的客户端浏览器 WebSocket 模拟支持，这样，你就可以在其基础上进行 WebSocket API 编程了，同时生成的代码可以运行在所有浏览器上。

为接收 ws://localhost:8080/echo 上的连接，需要启动 Python WebSocket Echo 服务器。打开命令行窗口，转到该文件所在的文件夹，然后执行以下命令：

```
python websocket.py
```

我们还引入了一个广播服务器 (broadcast server)，用以接收 ws://localhost:8080/broadcast 上的连接。与 Echo 服务器不同，发往这个特定服务器端的任何 WebSocket 消息都会被广播给所有

当前连接的客户端。将消息广播给多个监听者非常简单。为了启动广播服务器，打开命令行窗口，转到该文件所在的文件夹，然后执行以下命令：

```
python broadcast.py
```

以上两个脚本都用到了 `WebSocket.py` 中的 `WebSocket` 协议库，要实现其他的服务器端行为，可以在相应的路径下添加相应的处理代码。

**提示** 这个服务器只能处理 `WebSocket` 协议，无法响应 `HTTP` 请求。因为 `WebSocket` 连接衍生于 `HTTP` 的子集，而且使用升级过的报头，所以其他很多服务器都可以使用同一端口响应 `WebSocket` 和 `HTTP` 请求。

接下来，看看当浏览器与这个服务器通信时会怎么样。浏览器向 `WebSocket URL` 发出一个请求，服务器会返回报头来完成 `WebSocket` 握手。`WebSocket` 握手响应必须以 `HTTP/1.1 101 WebSocket Protocol Handshake` 开头。实际上，握手报头的内容和顺序要比 `HTTP` 报头定义得更加严格。

**提示** 如果你想要自己实现 `WebSocket` 服务器，可以参考 <http://tools.ietf.org/html/draft-hixie-thewebsocketprotocol> 中公布的 IETF 协议草案或是最新技术规范。

```
# 填写我们自己的响应报头
self.send_bytes("HTTP/1.1 101 Web Socket Protocol Handshake\r\n")
self.send_bytes("Upgrade: WebSocket\r\n")
self.send_bytes("Connection: Upgrade\r\n")
self.send_bytes("Sec-WebSocket-Origin: %s\r\n" % headers["Origin"])
self.send_bytes("Sec-WebSocket-Location: %s\r\n" % headers["Location"])

if "Sec-WebSocket-Protocol" in headers:
    protocol = headers["Sec-WebSocket-Protocol"]
    self.send_bytes("Sec-WebSocket-Protocol: %s\r\n" % protocol)

self.send_bytes("\r\n")
# 回写哈希处理后的响应 token
self.send_bytes(response_token)
```

握手完成之后，客户端和服务器就可以随时发送消息了。在服务器上，每个连接都通过一个 `WebSocketConnection` 实例来表示。`WebSocketConnection` 中的 `send()` 函数会把字符串转换成 `WebSocket` 协议字符串。`UTF-8` 格式编码的字符串以 `0x00` 和 `0xFF` 字节作为帧边界。在该服务器上，每个 `WebSocket` 连接都是一个支持缓冲发送的异步 `socket` 封装程序 (`asyncore.dispatcher_with_send`)。

**提示** `Python` 等语言还有很多其他的异步 I/O 框架。之所以选 `Asyncore`，是因为它包含在 `Python` 标准库当中。另外要注意的是，这个实现中可支持 `WebSocket` 协议的 75 和 76 版本。严格来说，不允许同时支持两个版本。这里只是为了做个简单的演示。



WebSocketConnection 继承自 `asyncore.dispatcher_with_send`，并将 `send()` 函数重写以支持 UTF-8 编码字符串，并且添加了 WebSocket 字符串帧。

```
def send(self, s):
    if self.readyState == "open":
        self.send_bytes("\x00")
        self.send_bytes(s.encode("UTF8"))
        self.send_bytes("\xFF")
```

`websocket.py` 文件中 `WebSocketConnections` 的处理程序沿用了简易的调度接口。连接请求到达时，调用处理程序中的 `dispatch()` 方法来为各帧分发有效载荷。`EchoHandler` 方法会将各条消息返回给发送者。

```
class EchoHandler(object):
    """
    The EchoHandler repeats each incoming string to the same WebSocket.
    """
    def __init__(self, conn):
        self.conn = conn
    def dispatch(self, data):
        self.conn.send("echo: " + data)
```

基本广播服务器 `broadcast.py` 的工作原理大致相同，但当广播程序接收到字符串时，会返回给所有连接着的 `WebSocket`，如下所示。

```
class BroadcastHandler(object):
    """
    The BroadcastHandler repeats incoming strings to every connected
    WebSocket.
    """
    def __init__(self, conn):
        self.conn = conn
    def dispatch(self, data):
        for session in self.conn.server.sessions:
            session.send(data)
```

`broadcast.py` 中的处理程序提供了一种轻量级的消息广播器，它只负责简单地发送和接收字符串，对我们的示例而言这已经足够了。需要注意，广播服务器并不对输入进行任何验证，但产品化的消息服务器中这种验证是必需的。产品化的 `WebSocket` 服务器至少要对输入数据的格式进行验证。

最后，代码清单 6-5 和代码清单 6-6 分别提供了 `websocket.py` 和 `broadcast.py` 的完整代码。不过代码只是用来演示一个服务器的实现，并不适用于实际部署。

#### 代码清单 6-5 `websocket.py` 的完整代码

```
#!/usr/bin/env python

import asyncore
import socket
```

```
import struct
import time
import hashlib

class WebSocketConnection(asyncore.dispatcher_with_send):
    def __init__(self, conn, server):
        asyncore.dispatcher_with_send.__init__(self, conn)

        self.server = server
        self.server.sessions.append(self)
        self.readystate = "connecting"
        self.buffer = ""

    def handle_read(self):
        data = self.recv(1024)
        self.buffer += data
        if self.readystate == "connecting":
            self.parse_connecting()
        elif self.readystate == "open":
            self.parse_frametype()

    def handle_close(self):
        self.server.sessions.remove(self)
        self.close()

    def parse_connecting(self):
        header_end = self.buffer.find("\r\n\r\n")
        if header_end == -1:
            return
        else:
            header = self.buffer[:header_end]
            # 从缓冲中移除报头, 以及行结束符占用的 4B
            self.buffer = self.buffer[header_end+4:]
            header_lines = header.split("\r\n")
            headers = {}

            # 验证 HTTP 请求和构建路径
            method, path, protocol = header_lines[0].split(" ")
            if method != "GET" or protocol != "HTTP/1.1" or path[0] != "/":
                self.terminate()
                return

            # 解析报头
            for line in header_lines[1:]:
                key, value = line.split(": ")
                headers[key] = value

            headers["Location"] = "ws://" + headers["Host"] + path

            self.readystate = "open"
            self.handler = self.server.handlers.get(path, None)(self)

            if "Sec-WebSocket-Key1" in headers.keys():
                self.send_server_handshake_76(headers)
            else:
```



```

        self.send_server_handshake_75(headers)

def terminate(self):
    self.ready_state = "closed"
    self.close()

def send_server_handshake_76(self, headers):
    """
    Send the WebSocket Protocol v.76 handshake response
    """

    key1 = headers["Sec-WebSocket-Key1"]
    key2 = headers["Sec-WebSocket-Key2"]
    # 从 buffer 中再读 8B
    key3, self.buffer = self.buffer[:8], self.buffer[8:]

    response_token = self.calculate_key(key1, key2, key3)

    # 回写响应报头
    self.send_bytes("HTTP/1.1 101 Web Socket Protocol Handshake\r\n")
    self.send_bytes("Upgrade: WebSocket\r\n")
    self.send_bytes("Connection: Upgrade\r\n")
    self.send_bytes("Sec-WebSocket-Origin: %s\r\n" % headers["Origin"])
    self.send_bytes("Sec-WebSocket-Location: %s\r\n" % headers["Location"])

    if "Sec-WebSocket-Protocol" in headers:
        protocol = headers["Sec-WebSocket-Protocol"]
        self.send_bytes("Sec-WebSocket-Protocol: %s\r\n" % protocol)

    self.send_bytes("\r\n")
    # 回写哈希过的响应令牌
    self.send_bytes(response_token)

def calculate_key(self, key1, key2, key3):
    # 通过提取数值字符来解析 key1 和 key2
    num1 = int("".join([digit for digit in list(key1) if digit.isdigit()]))
    spaces1 = len([char for char in list(key1) if char == " "])
    num2 = int("".join([digit for digit in list(key2) if digit.isdigit()]))
    spaces2 = len([char for char in list(key2) if char == " "])

    combined = struct.pack(">II", num1/spaces1, num2/spaces2) + key3
    # md5 将组合后的字节相加
    return hashlib.md5(combined).digest()

def send_server_handshake_75(self, headers):
    """
    Send the WebSocket Protocol v.75 handshake response
    """

    self.send_bytes("HTTP/1.1 101 Web Socket Protocol Handshake\r\n")
    self.send_bytes("Upgrade: WebSocket\r\n")
    self.send_bytes("Connection: Upgrade\r\n")
    self.send_bytes("WebSocket-Origin: %s\r\n" % headers["Origin"])
    self.send_bytes("WebSocket-Location: %s\r\n" % headers["Location"])
    if "Protocol" in headers:

```

```

        self.send_bytes("WebSocket-Protocol: %s\r\n" % headers["Protocol"])

    self.send_bytes("\r\n")

    def parse_frametype(self):
        while len(self.buffer):
            type_byte = self.buffer[0]
            if type_byte == "\x00":
                if not self.parse_textframe():
                    return

    def parse_textframe(self):
        terminator_index = self.buffer.find("\xFF")
        if terminator_index != -1:
            frame = self.buffer[1:terminator_index]
            self.buffer = self.buffer[terminator_index+1:]
            s = frame.decode("UTF8")
            self.handler.dispatch(s)
            return True
        else:
            # 帧不完整
            return False

    def send(self, s):
        if self.readystate == "open":
            self.send_bytes("\x00")
            self.send_bytes(s.encode("UTF8"))
            self.send_bytes("\xFF")

    def send_bytes(self, bytes):
        asyncio.dispatcher_with_send.send(self, bytes)

class EchoHandler(object):
    """
    The EchoHandler repeats each incoming string to the same Web Socket.
    """

    def __init__(self, conn):
        self.conn = conn

    def dispatch(self, data):
        self.conn.send("echo: " + data)

class WebSocketServer(asyncio.dispatcher):

    def __init__(self, port=80, handlers=None):
        asyncio.dispatcher.__init__(self)
        self.handlers = handlers
        self.sessions = []
        self.port = port
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.set_reuse_addr()
        self.bind(("", port))
        self.listen(5)

```



```
def handle_accept(self):
    conn, addr = self.accept()
    session = WebSocketConnection(conn, self)

if __name__ == "__main__":
    print "Starting WebSocket Server"
    WebSocketServer(port=8080, handlers={"/echo": EchoHandler})
    asyncore.loop()
```

从代码中可以看到，WebSocket 握手中有一个特殊的关键值计算。这是为了防止跨协议（cross-protocol）攻击。简而言之，这种机制能够阻止恶意的 WebSocket 客户端代码连接到非 WebSocket 服务器。截止草案 76，握手中的这部分设计仍在讨论中。

#### 代码清单 6-6 broadcast.py 的完整代码

```
#!/usr/bin/env python

import asyncore
from websocket import WebSocketServer

class BroadcastHandler(object):
    """
    The BroadcastHandler repeats incoming strings to every connected
    WebSocket.
    """

    def __init__(self, conn):
        self.conn = conn

    def dispatch(self, data):
        for session in self.conn.server.sessions:
            session.send(data)

if __name__ == "__main__":
    print "Starting WebSocket broadcast server"
    WebSocketServer(port=8000, handlers={"/broadcast": BroadcastHandler})
    asyncore.loop()
```

现在我们有可用的 Echo 服务器，下一步需要编写客户端程序。

## 6.4 使用 HTML5 WebSockets API

本节我们将更为详细地探讨 HTML5 WebSockets 的使用方法。

### 6.4.1 浏览器支持情况检测

在使用 HTML5 WebSockets API 之前，首先需要确认浏览器的支持情况。如果浏览器不支持，我们可以提供一些替代信息，提示用户升级浏览器。代码清单 6-7 是检测浏览器支持情况的一种方法。

#### 代码清单 6-7 检测浏览器支持情况

```
function loadDemo() {
```

```

if (window.WebSocket) {
    document.getElementById("support").innerHTML = "HTML5 WebSocket is supported in your
        browser.";
} else {
    document.getElementById("support").innerHTML = "HTML5 WebSocket is not supported in
        your browser.";
}
}

```

上面的示例代码使用 `loadDemo` 函数检测浏览器支持性，该函数会在页面加载时被调用。若存在 `WebSocket` 对象，调用 `window.WebSocket` 就会将其返回，否则将触发异常失败处理。然后，根据检测结果更新页面显示。由于页面代码中预定义了 `support` 元素，将适当的信息显示在此元素中就可以从页面上反映出浏览器的支持情况。

检测浏览器是否支持 HTML5 WebSockets 的另一种方法是使用浏览器控制台（如 Firebug 或 Chrome 开发工具）。图 6-5 是在 Google Chrome 中检测自身是否支持 WebSockets（若不支持，`window.WebSocket` 命令将返回“undefined”）。

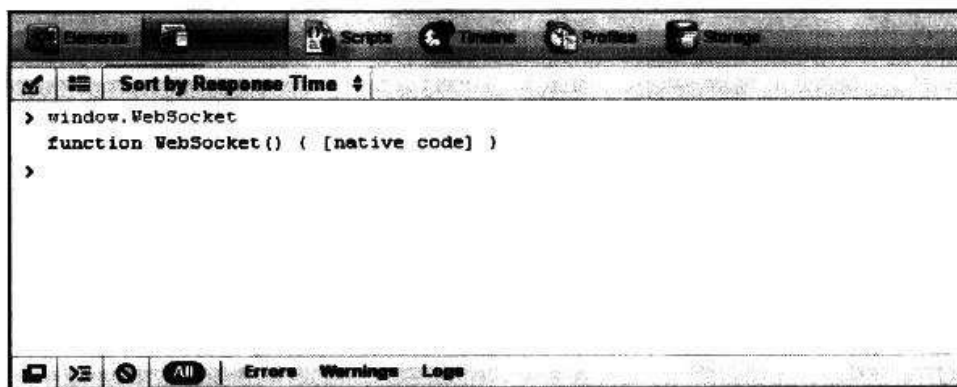


图 6-5 通过 Google Chrome 开发工具来检测 WebSocket 支持性

## 6.4.2 API 的基本用法

以下示例代码均位于本书网站上的 WebSockets 部分。`websocket.html` 文件、`broadcast.html` 文件（还包括下面一节中用到的 `tracker.html` 文件），以及前面提到的可在 Python 中启动的 WebSocket 服务器的代码都在同一个文件夹中。

### 1. WebSocket 对象的创建及其与 WebSocket 服务器的连接

WebSocket 接口的使用非常简单。要连接通信端点，只需要创建一个新的 `WebSocket` 实例，并提供希望连接的对端 URL。`ws://`和 `wss://`前缀分别表示 `WebSocket` 连接和安全的 `WebSocket` 连接。

```

url = "ws://localhost:8080/echo";
w = new WebSocket(url);

```

### 2. 添加事件监听器

WebSocket 编程遵循异步编程模型，打开 socket 后，只需要等待事件发生，而不需要主动向服务器轮询，所以需要在 `WebSocket` 对象中添加回调函数来监听事件。



WebSocket 对象有三个事件：`open`、`close` 和 `message`。当连接建立时触发 `open` 事件，当收到消息时触发 `message` 事件，当 WebSocket 连接关闭时触发 `close` 事件。同大多数 JavaScript API 一样，事件处理时会调用相应的 (`onopen`、`onmessage`、`onclose`) 回调函数。

```
w.onopen = function() {
    log("open");
    w.send("thank you for accepting this websocket request");
}
w.onmessage = function(e) {
    log(e.data);
}
w.onclose = function(e) {
    log("closed");
}
```

### 3. 发送消息

当 socket 处于打开状态（即调用 `onopen` 监听程序之后，调用 `onclose` 监听程序之前），可以采用 `send` 方法来发送消息。消息发送完成之后，可以调用 `close` 方法来终止连接，当然也可以不这么做，让其保持打开状态。

```
document.getElementById("sendButton").onclick = function() {
    w.send(document.getElementById("inputMessage").value);
}
```

浏览器双向通信就这么简单。为完整起见，代码清单 6-8 列出了带有 WebSocket 的整个 HTML 页面代码。

代码清单 6-8 websocket.html 代码

```
<!DOCTYPE html>
<title>WebSocket Test Page</title>
<script>

    var log = function(s) {
        if (document.readyState !== "complete") {
            log.buffer.push(s);
        } else {
            document.getElementById("output").innerHTML += (s + "\n");
        }
    }
    log.buffer = [];

    url = "ws://localhost:8080/echo";
    w = new WebSocket(url);
    w.onopen = function() {
        log("open");
        w.send("thank you for accepting this WebSocket request");
    }
    w.onmessage = function(e) {
        log(e.data);
    }
}
```

```
w.onclose = function(e) {
    log("closed");
}

window.onload = function() {
    log(log.buffer.join("\n"));
    document.getElementById("sendButton").onclick = function() {
        w.send(document.getElementById("inputMessage").value);
    }
}
</script>

<input type="text" id="inputMessage" value="Hello, WebSocket!"><button
id="sendButton">Send</button>
<pre id="output"></pre>
```

#### 4. 运行WebSocket页面

为了测试带有 WebSocket 代码的 websocket.html 页面，请打开命令行窗口，转到 WebSocket 代码所在目录下，输入以下命令启动服务器：

```
python -m SimpleHTTPServer 9999
```

下一步，打开另一命令行窗口，转到 WebSocket 代码所在目录，输入以下命令来启动 Python WebSocket 服务：

```
python websocket.py
```

最后，打开支持 WebSockets 的浏览器，浏览 <http://localhost:9999/websocket.html>。

图 6-6 为运行起来的页面。



图 6-6 websocket.html 运行页面

示例代码文件夹中还包含一个 Web 页面，可以连接到之前创建的 broadcast 服务。操作步骤如下：先将正在运行 WebSocket 服务器的命令行窗口关闭，然后转到 WebSocket 代码所在目录，输入以下命令启动 Python WebSocket 服务器。



```
python broadcast.py
```

分别打开两个支持 WebSockets 的浏览器，统一转到 `http://localhost:9999/broadcast.html`。

图 6-7 所示为 broadcast WebSocket 服务器在两个独立的 Web 页面中的运行情况。

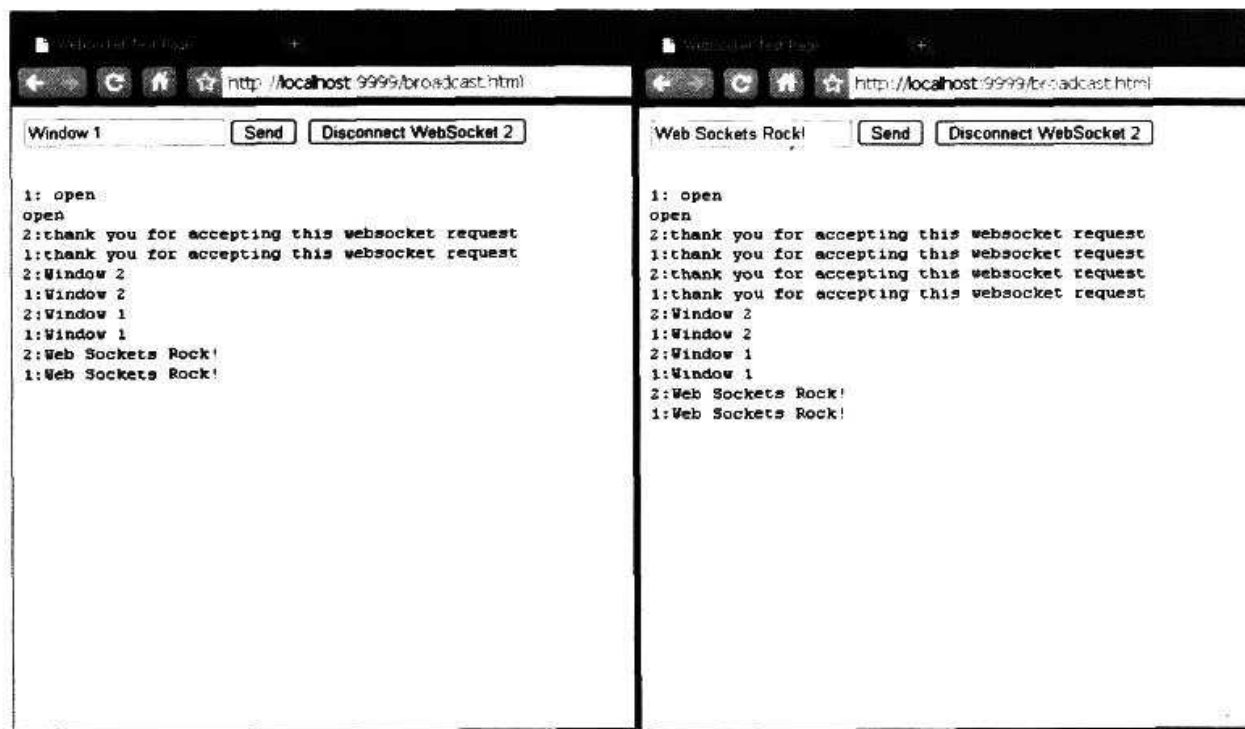


图 6-7 两个浏览器中运行的 broadcast.html

## 6.5 创建 HTML5 WebSockets 应用程序

我们已经了解 WebSocket 的基本知识，现在可以动手实践一下了。之前，我们使用 HTML5 Geolocation 接口创建了一个直接在 Web 页面中计算距离的应用。我们可以利用同样的 Geolocation 技术，结合对 WebSockets 的支持，创建一个支持多方通信的简单应用：位置跟踪器。

---

**提示** 我们需要用到上面介绍的 broadcast WebSocket 服务器，如果对此还不是很熟悉，则建议先花一些时间来学习相关的基础知识。

---

在这个示例应用中，我们会将 WebSocket 和 Geolocation 技术相结合，以便确定用户位置并将其广播给所有有效的监听者。所有加载该应用并连接到同一 broadcast 服务器的用户都将通过 WebSocket 来定期发送他们的位置信息。同时，该应用还将监听所有来自服务器的消息，并将其向所有监听者实时地更新显示。在长跑比赛中，此类应用可以使参赛者知道所有参赛选手的位置，并促使他们加快速度（或是减慢速度）。

除经度和纬度位置之外，这个小应用不包含任何个人信息。姓名、生日、喜欢的冰淇淋口味等都是严格保密的。

### 敬告用户

“这是一个关于个人信息共享的应用程序。在应用首次访问 Geolocation API 的时候，浏览器会为用户弹出一个提示框。诚然，这里只共享了位置信息。但是，如果用户没明白浏览器警告的意思，那么这个应用将会非常严肃地给他上一课，他会意识到将敏感信息传送到远端有多么容易。请确保用户充分理解位置信息提交的后果。

如果用户对此有疑问，我们需要在应用中多做一些相关的工作，提前告知用户他们的敏感数据会被如何使用。告知方式应尽量简洁。”

—Brian

敬告过后让我们深入探讨代码。同样地，完整代码示例都在网上，以供细读。这里我们将关注其中最重要的部分。应用程序的最终效果如图 6-8 所示，理论上还可以通过叠加在地图上来对其进行优化。

#### HTML5 WebSocket / Geolocation Tracker

**Geolocation**  
Location updated at Sun Jan 17 2010 23:37:04 GMT-0800 (Pacific Standard Time)

**WebSocket**  
Updated location from Me

Lat: 37.399305 | Lon: -122.076307

图 6-8 位置跟踪器应用

### 6.5.1 编写 HTML 文件

示例中的 HTML 标签专门进行了简化，这样有助于突出手边的数据。有多简单呢？

```
<body onload="loadDemo()">

<h1>HTML5 WebSocket / Geolocation Tracker</h1>

<div><strong>Geolocation</strong>: <p id="geoStatus">HTML5 Geolocation is
<strong>not</strong> supported in your browser.</p></div>
<div><strong>WebSocket</strong>: <p id="socketStatus">WebSockets are <strong>not</strong>
supported in your browser.</p></div>

</body>
```

简单到只包含一个标题和一些状态区域：一个用于更新 Geolocation，另一个用于记录 WebSocket 行为日志。这样，当收到实时消息时，就可将实际位置数据插入到页面中。

默认情况下，状态信息会显示用户的浏览器不支持 Geolocation 或 WebSockets。一旦检测到可以支持这两种 HTML5 技术，就采用更为友好的方式来更新状态信息。



```
<script>

    // WebSocket 的引用
    var socket;

    // 为该会话生成唯一的随机 ID
    var myId = Math.floor(100000*Math.random());

    // 当前显示的行数
    var rowCount = 0;
```

我们再次通过脚本实现了应用功能。首先，定义几个变量。

- socket：全局变量，方便各种函数访问。
- myId：用于在线表示位置数据，介于 0~100 000 之间的随机数。这个数字仅用来表示相对位置变化，而无需使用名字之类的其他个人信息。足够大的数据池可保证每个人的标识唯一。
- rowCount：表示有多少用户将其位置数据发送给我们，主要用于可视化显示。

下面两个函数看起来应该比较眼熟。与其他示例应用一样，它们用来辅助更新状态信息。有两个状态信息需要更新。

```
function updateSocketStatus(message) {
    document.getElementById("socketStatus").innerHTML = message;
}

function updateGeolocationStatus(message) {
    document.getElementById("geoStatus").innerHTML = message;
}
```

当获取位置出现异常时，有必要设置对用户友好的异常提醒信息。如需了解更多的 Geolocation 异常处理信息，请查阅第 4 章。

```
function handleLocationError(error) {
    switch(error.code)
    {
        case 0:
            updateGeolocationStatus("There was an error while retrieving your location: " +
                error.message);
            break;
        case 1:
            updateGeolocationStatus("The user prevented this page from retrieving a
                location.");
            break;
        case 2:
            updateGeolocationStatus("The browser was unable to determine your location: " +
                error.message);
            break;
        case 3:
            updateGeolocationStatus("The browser timed out before retrieving the location.");
            break;
    }
}
```

## 6.5.2 添加 WebSocket 代码

现在，一起来看一下核心功能代码。页面初始加载时会调用 loadDemo 函数，它是应用程序的起点。

```
function loadDemo() {
    //进行检测，确保浏览器支持 socket
    if (window.WebSocket) {

        // broadcast WebSocket 服务器的位置
        url = "ws://localhost:8080";
        socket = new WebSocket(url);
        socket.onopen = function() {
            updateSocketStatus("Connected to WebSocket tracker server");
        }
        socket.onmessage = function(e) {
            updateSocketStatus("Updated location from " + dataReturned(e.data));
        }
    }
}
```

首先是创建 WebSocket 连接。同其他 HTML5 技术一样，在使用之前，需要进行浏览器支持情况检测，因此代码一开始测试了浏览器是否支持 window.WebSocket 对象。

一旦验证通过，就可以使用之前提到的连接字符串格式来与远程广播服务器建立连接。连接会保存在我们声明的 socket 全局变量中。

最后，我们建立了两个函数，用来在 WebSocket 收到更新消息时执行相关操作。onopen 函数只负责更新状态信息以告知用户已成功建立连接。类似地，onmessage 函数只负责更新状态信息以告知用户消息已送达。onmessage 函数同时还会调用 dataReturned 函数以便在页面上显示送达的消息内容。dataReturned 函数稍后再进行讨论。

## 6.5.3 添加 Geolocation 代码

读过第 4 章后，下面的代码应该就不陌生了。这里，我们要检测 Geolocation 服务的浏览器支持情况，并对状态信息做相应更新。

```
var geolocation;
if(navigator.geolocation) {
    geolocation = navigator.geolocation;
    updateGeolocationStatus("HTML5 Geolocation is supported in your browser.");
}

//使用 Geolocation API 注册位置更新处理函数
geolocation.watchPosition(updateLocation,
                           handleLocationError,
                           {maximumAge:20000});
}
```

和第 4 章中一样，我们监控当前位置的变化，并注册处理函数保证在位置发生变化时，updateLocation 函数会被调用。错误信息会发送至 handleLocationError 函数，位置信息则设为



每 20s 更新一次。

当新位置可用时，浏览器将调用以下函数。

```
function updateLocation(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    var timestamp = position.timestamp;

    updateGeolocationStatus("Location updated at " + timestamp);

    //通过 WebSocket 发送我的位置
    var toSend = JSON.stringify([myId, latitude, longitude]);
    sendMyLocation(toSend);
}
```

这部分看起来与第 4 章中非常类似，但要简单得多。这里，我们从浏览器提供的位置信息中提取了经度、纬度和时间戳信息，然后更新状态消息，表明新的位置信息已经到达。

#### 6.5.4 合并所有内容

最后一段代码得出了用于发送给远程 broadcast WebSocket 服务器的消息字符串，它采用 JSON 编码格式：

```
"[<id>, <latitude>, <longitude>]"
```

其中 id 是一个用来标识用户的随机数，latitude 和 longitude 是由 Geolocation 位置对象提供的。我们将 JSON 格式编码的字符串信息直接发送给服务器。

通过 sendMyLocation() 函数可将位置信息发送给服务器。

```
function sendMyLocation(newLocation) {
    if (socket) {
        socket.send(newLocation);
    }
}
```

在 socket 已成功创建（并为后续访问保留）的情况下，通过这个函数将消息字符串发送到服务器是安全的。消息到达后，WebSocket 消息广播服务器将把位置信息分发给处于连接状态且正在监听消息的每一个浏览器。这样，每个人都能知道你在哪里，至少通过随机数标识出的大量匿名用户能让人了解你的动向。

发送消息之后，让我们看看这些相同消息抵达时，浏览器将如何处理。回想一下，我们已在 socket 上注册了 onmessage 函数，其作用是将所有输入数据传送到 dataReturned() 函数。下一步，我们来详细分析这个函数。

```
function dataReturned(locationData) {
    //从数据中拆分出 ID、经度和纬度
    var allData = JSON.parse(locationData);
    var incomingId = allData[1];
    var incomingLat = allData[2];
    var incomingLong = allData[3];
}
```

dataReturned()函数有两个功能。它将在页面中创建（或更新）显示元素，以便呈现收到的消息字符串中的位置信息，同时返回标明了消息来源于哪个用户的文本信息。随后 socket.onmessage 函数会将其中的用户名显示在页面上方状态信息中。

数据处理函数的第一步操作是使用 JSON.parse 对收到的数据进行分解。尽管从健壮性角度考虑，还应该检查数据格式是否非法，但是作为示例，此处假设所有消息格式都是合法的。因此，字符串会被干净利落地分解成随机 ID、纬度和经度。

```
//根据 ID 定位到 HTML 元素
//如果不存在，就创建
var incomingRow = document.getElementById(incomingId);
if (!incomingRow) {
    incomingRow = document.createElement('div');
    incomingRow.setAttribute('id', incomingId);
}
```

当收到一条消息时，用户界面会为每个随机 ID 创建一个可见的<div>。用户的 ID 也会显示在里面，换句话说，当前用户自己的数据发送出去并从 WebSocket 广播服务器返回后，也会被显示在页面上。

因此，我们首先要通过消息字符串中的 ID 来定位它所对应的行元素。如果不存在，就创建一个，并将其 id 值设为从 socket 服务器返回的 id 值，以备后用。

```
incomingRow.userText = (incomingId == myId) ?
    'Me' :
    'User ' + rowCount;
rowCount++;
```

数据行中显示什么样的用户文本内容很容易判断。如果 ID 与用户 ID 一致，就显示为“me”。否则，用户名由一个公共字符串和一个每次加 1 的行数组合而成。

```
document.body.appendChild(incomingRow);
}
```

新的显示元素就绪后，就会被插入到页面最后。不管显示元素是新建的还是已经存在的（如果不是某个用户的第一次位置更新，那相应的显示元素就已经存在了），都需要根据当前的文本信息更新它的内容。

```
//使用新的值更新对应的文本
incomingRow.innerHTML = incomingRow.userText + " \ \ Lat: " +
    incomingLat + " \ \ Lon: " +
    incomingLong;

return incomingRow.userText;
}
```

在这个示例中，我们通过一个反斜杠（当然，经过了转义处理）来分隔用户名文本信息和经、纬度信息。最后，为了更新状态行，用于显示的用户名文本将返回给调用函数。

WebSocket 和 Geolocation 的组合程序已经完成。但要注意，只有当多个浏览器同时访问应用时，你才能看到多条更新信息。读者不妨做个练习，改造示例代码，让位置信息显示在 Google 地图上。然后，你就会明白 HTML5 的奥妙了。



### 6.5.5 最终代码

代码清单 6-9 给出了 tracker.html 文件中的源代码。

代码清单 6-9 tracker.html

```
<!DOCTYPE html>
<html lang="en">

<head>
<title>HTML5 WebSocket / Geolocation Tracker</title>
<link rel="stylesheet" href="styles.css">
</head>

<body onload="loadDemo()">
<h1>HTML5 WebSocket / Geolocation Tracker</h1>

<div><strong>Geolocation</strong>: <p id="geoStatus">HTML5 Geolocation is
<strong>not</strong> supported in your browser.</p></div>
<div><strong>WebSocket</strong>: <p id="socketStatus">WebSockets are <strong>not</strong>
supported in your browser.</p></div>

<script>

    // WebSocket 的引用
    var socket;

    //为该会话生成唯一的随机 ID
    var myId = Math.floor(100000*Math.random());

    //当前显示的行数
    var rowCount = 0;

    function updateSocketStatus(message) {
        document.getElementById("socketStatus").innerHTML = message;
    }

    function updateGeolocationStatus(message) {
        document.getElementById("geoStatus").innerHTML = message;
    }

    function handleLocationError(error) {
        switch(error.code)
        {
            case 0:
                updateGeolocationStatus("There was an error while retrieving your location: " +
                    error.message);

                break;
            case 1:
                updateGeolocationStatus("The user prevented this page from retrieving a
                    location.");

                break;
        }
    }
</script>
```

```
case 2:
    updateGeolocationStatus("The browser was unable to determine your location: " +
        error.message);
    break;
case 3:
    updateGeolocationStatus("The browser timed out before retrieving the location.");
    break;
}
}

function loadDemo() {
    //进行测试, 确保浏览器支持 socket
    if (window.WebSocket) {
        // broadcast WebSocket 服务器的位置
        url = "ws://localhost:8080";
        socket = new WebSocket(url);
        socket.onopen = function() {
            updateSocketStatus("Connected to WebSocket tracker server");
        }
        socket.onmessage = function(e) {
            updateSocketStatus("Updated location from " + dataReturned(e.data));
        }
    }
}

var geolocation;
if(navigator.geolocation) {
    geolocation = navigator.geolocation;
    updateGeolocationStatus("HTML5 Geolocation is supported in your browser.");
}

//使用 Geolocation API 注册位置更新处理函数
geolocation.watchPosition(updateLocation,
    handleLocationError,
    {maximumAge:20000});
}

function updateLocation(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    var timestamp = position.timestamp;

    updateGeolocationStatus("Location updated at " + timestamp);

    //通过 WebSocket 发送我的位置
    var toSend = JSON.stringify([myId, latitude, longitude]);
    sendMyLocation(toSend);
}

function sendMyLocation(newLocation) {
    if (socket) {
        socket.send(newLocation);
    }
}
```



```
    }  
    function dataReturned(locationData) {  
        //从数据中拆分出 ID、经度和纬度  
        var allData = JSON.parse(locationData)  
        var incomingId = allData[1];  
        var incomingLat = allData[2];  
        var incomingLong = allData[3];  
  
        //根据 ID 定位到 HTML 元素  
        //如果不存在,就创建  
        var incomingRow = document.getElementById(incomingId);  
        if (!incomingRow) {  
            incomingRow = document.createElement('div');  
            incomingRow.setAttribute('id', incomingId);  
  
            incomingRow.userText = (incomingId == myId) ?  
                'Me' :  
                'User ' + rowCount;  
  
            rowCount++;  
  
            document.body.appendChild(incomingRow);  
        }  
        //使用新的值更新对应的文本  
        incomingRow.innerHTML = incomingRow.userText + " \\ Lat: " +  
            incomingLat + " \\ Lon: " +  
            incomingLong;  
  
        return incomingRow.userText;  
    }  
    </script>  
</body>  
</html>
```

## 6.6 小结

本章,我们演示了如何使用 HTML5 WebSockets 创建引人注目的实时应用,见证了其开发有多么简单,功能又是多么强大。

首先,我们了解了 HTML5 WebSockets 协议本身的特性,以及它对现有 HTTP 流量方面的影响。我们从网络开销方面对比了基于轮询的通信策略和 WebSocket 通信技术。

然后,我们搭建了一个简单的 WebSocket 服务器来演示 WebSocket 的运行情况,从整个搭建过程中可以看到 WebSocket 协议在实际使用中非常简单。类似地,我们还讨论了客户端的 WebSockets API,发现它也很容易与现有 JavaScript 程序集成。

最后,我们创建了一个相对复杂的示例应用,它综合了 Geolocation 和 WebSocket 的能力,这也说明两种技术完全能够良好地协同工作。

现在,我们已经了解了 HTML5 为浏览器带来的 socket 式的网络编程技术。下一步,我们会把注意力转移到收集更多有用的数据上,而不仅仅是用户的当前位置信息。下一章,我们将深入了解 HTML5 在表单控件方面的改进。

**本**章我们将探讨一种旧貌换新颜的技术——HTML 表单。HTML 表单自问世以来，一直是 Web 的核心技术之一。如果没有表单，那么很多应用，如在线交易、论坛以及搜索等，根本无法实现。

尽管 HTML5 Forms 已经出现多年，但无论从规范还是从具体实现来说，它始终都是最不稳定的一块内容。有好消息也有坏消息，好消息是这部分的发展速度非常快，坏消息是开发人员需要仔细选择新的表单控件，以确保兼容所有浏览器。HTML5 Forms 规范详细规定了大量的表单 API，新发布的兼容 HTML5 的主流浏览器都会或多或少地添加对表单控件的支持，这其中不乏一些有用的验证功能。

我们将利用一章的篇幅梳理表单控件，为你介绍哪些可以使用，哪些即将发布。

## 7.1 HTML5 Forms 概述

如果熟悉 HTML 表单（对 HTML 高级编程感兴趣的人通常如此），那么你会很容易适应 HTML5 Forms 的新增功能。如果不熟悉表单的基本知识，我们建议你先阅读一些表单创建和处理方面的教程。本章用到的旧表单知识在相关教程中都能找到，而作为开发人员的你一定乐于看到下面的几点：

- 表单仍然使用<form>元素作为容器，我们可以在其中设置基本的提交特性；
- 当用户或开发人员提交页面时，表单仍然用于向服务器端发送表单中控件的值；
- 所有之前熟悉的表单控件，如文本框、单选按钮、复选框等，都沿用原来的使用方法（尽管增加了新的功能）；
- 仍然可以使用脚本操作表单控件，喜欢自己修改和编写处理函数的开发人员大可放心。

### 7.1.1 HTML Forms 与 XForms

在过去的数年间（远在 HTML5 受到普遍关注以前），你可能听说过 XForms 的事。XForms 是一个以 XML 为核心、功能强大却略显复杂的标准，它用于规范客户端表单的行为，而专门的 W3C 工作组研究这些行为已经近十年。XForms 充分利用了 XML Schema，制订了针对验证和格式化的精确准则。不过，很遗憾，在没有安装插件的情况下，主流浏览器均不支持 XForms。



### 7.1.2 功能性表单

HTML5 表单规范更加注重对现有的简单 HTML 表单功能的改进，力求使之包含更多控件类型，同时它还着力解决 Web 开发人员今天面对的实际难题。不过，在开发过程中有一点需要牢记于心，那就是时刻关注表单的跨浏览器表现。

**提示** 一定要领会HTML5 Forms的核心设计理念：规范的核心是功能性动作和语义，而非外观和显示效果。

规范虽然详细描述了相关功能性 API 的使用方法，如颜色、日历、数值选择器、邮件地址等，但是它并没有规定浏览器应该以何种方式将这些元素呈现给用户。这种做法带来了多方面的好处。首先，浏览器会竞相改善用户交互方式；其次，它分离了样式和语义；再次，在未来或面对专用用户输入设备时，可以根据实际情况灵活调整交互方式。不过，除非目标浏览器平台能够支持 Web 应用中用到的所有表单控件，否则，你要向用户提供充足的上下文信息，以确保他们能够通过其他方式完成与应用间的交互。有了正确的提示和描述后，用户在使用过程中才不会产生疑问，即使浏览器不支持未知表单输入控件而采用了备选处理方案，用户也不会因此而受到影响。

HTML5 表单包含了大量新的 API 和元素类型，它们已经得到众多浏览器的支持。为了便于理解，我们将其分成两类：

- 新的输入型控件；
- 新的函数和特性。

在正式讨论之前，让我们了解一下各浏览器对 HTML5 Forms 的支持情况。

### 7.1.3 HTML5 Forms 的浏览器支持情况

虽然兼容 HTML5 Forms 的浏览器在逐渐增多，但支持的程度还参差不齐。除了 Opera 浏览器之外，很多浏览器厂商还没有投入太多精力去支持新的表单控件。最近，可能是由于 iPhone 及其屏幕键盘显示的缘故，WebKit 内核的浏览器添加了对 HTML5 Forms 的支持，这点随后我们还会详细讨论。诸如验证等高级功能也刚刚被引进。表 7-1 总结了 HTML5 Forms 的浏览器支持现状。

表7-1 HTML5 Forms的浏览器支持情况

浏览器	支持情况
Chrome	5.0.x支持的输入型控件有email、number、tel、url、search和range。还支持大部分的验证功能
Firefox	3.6版本不支持，4.0版本开始支持。不支持的新输入型控件有url、email和range，这些控件会退化为普通文本输入框
Internet Explorer	不支持，不过新的输入型控件会退化为普通文本输入框
Opera	当前的多个版本完全支持最初的规范，包括验证功能
Safari	4.0.x版支持的输入型控件有email、number、tel、url、search和range。支持大部分验证功能。在Safari浏览器移动版中，有些类型的支持情况更好



对于新的 HTML5 Forms 而言，检测浏览器支持情况其实意义不大，因为根据新的设计原则，在旧的浏览器中新的表单控件会平滑降级。换言之，这意味使用 HTML5 Forms 的新元素完全没有后顾之忧，因为浏览器不支持新的输入型控件时，会把它们呈现为简单的文本输入框。不过，也正因为如此，多层 (multi-tier) 表单验证显得尤为重要 (随后我们会详细讨论)，即使现代浏览器都支持对表单控件的数据类型进行验证，我们也不能完全依赖其自身提供的验证器。

现在，我们已经了解了浏览器的支持情况。接下来，让我们逐一了解 HTML5 规范中新增的表单控件。

### 7.1.4 输入型控件目录

从 W3C 网站上可以找到 HTML5 中所有新增和修改的元素，具体的元素目录位于 <http://dev.w3.org/html5/markup/>。

目录列出了 HTML 页面中当前已有以及未来可能发布的所有元素。新增的和修改过的元素也在目录清单中。然而，列表中所谓的“新”是相对于 HTML4 规范而言——并不代表该元素已经获得浏览器支持或者出现在了最终规范中。明确了这一点后，让我们开始讨论 HTML5 引入的新的表单元素。表 7-2 列出了 type 特性对应的新的特性值。Web 开发人员都会非常熟悉 `<input type="text">` 和 `<input type="checkbox">`。新特性值的使用方式与现有的模式类似。

表7-2 浏览器中出现的新HTML5 表单元素

类 型	用 途
tel	电话号码
email	电子邮件地址文本框
url	网页的 URL
search	用于搜索引擎，比如在站点顶部显示的搜索框
range	特定值范围内的数值选择器，典型的显示方式是滑动条

新的输入型控件有什么用呢？用于 API 编程吗？貌似没多大用处。事实上，就 tel、email、url 和 search 来说，我们无法通过合适的特性值把它们与简单输入类型 text 区分开来。

将输入控件指定为特殊的类型会得到什么呢？能够得到专用的输入型控件。(适用范围可能有限，而且很多桌面浏览器可能也不支持。)

以 email 为例：

```
<input type="email">
```

这与以往的使用方式不同，以前只是声明单纯的 text 类型：

```
<input type="text">
```

这样做的好处是，可以让浏览器在支持控件的情况下，为用户呈现不同的输入方式或用户界面，而且浏览器还可以在表单提交以前对其做进一步的验证，我们会在本章后面讨论验证。

移动设备浏览器向来是最先支持这些表单控件类型的浏览器。通过识别不同的类型，浏览器



会显示不同的输入界面。当类型为 text 时，iPhone 手机会显示标准的屏幕键盘，如图 7-1 所示。当类型为 email 时，iPhone 手机会改变屏幕键盘，显示 @ 符号，如图 7-2 所示。

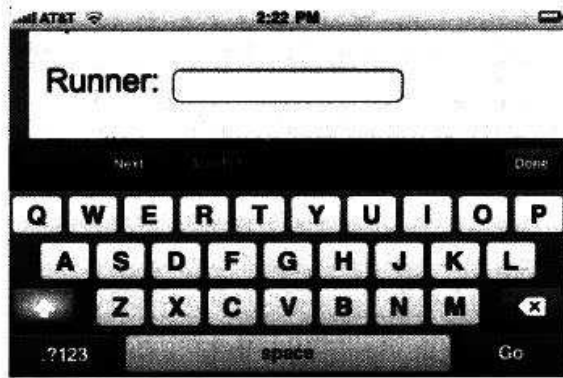


图 7-1 类型为 text 时的屏幕键盘

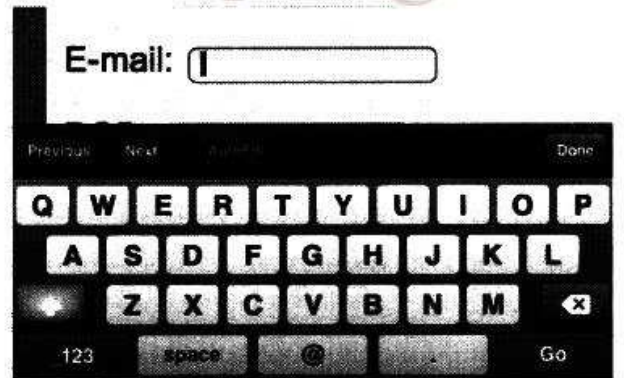


图 7-2 类型为 email 时的屏幕键盘

请注意二者在空格键区域的细微差别，类型为 email 时，在空格键区插入了 @ 和 . 符号以便于输入电子邮件地址。url 和 search 类型与此类似，键盘上会有局部微调。不过，桌面版的 Safari 浏览器，以及其他没有明确支持 email、url、search 和 tel 类型的浏览器，只会显示普通文本输入框。未来的浏览器，即使是桌面版本，也可能会针对不同类型的输入框向用户提供直观的提示。以 Opera 浏览器为例，它会在输入框旁边显示一个小信封图标，表明应该在此输入框中输入电子邮件地址。尽管目前浏览器的支持情况有限，我们还是可以在 Web 应用程序中使用这些类型，因为浏览器会针对这些类型进行优化，或者干脆将其降级为普通文本输入框。

还有一种逐渐得到浏览器青睐的类型——`<input type="range">`。设计 range 类型的目的是为了用户指定的数值范围中进行选择。例如，在表单中使用 range 输入型控件来限制年龄选择的范围（如限定最小年龄为 18）。通过创建 range 控件并为其指定 min 和 max 值，开发人员能够在页面上显示出一个数值范围选择器，用户只能选择指定范围内的数值。以 Opera 浏览器为例，如下代码：

```
<input type="range" min="18" max="120">
```

会生成一种便捷的控件，让用户从受限的年龄范围内选择合适的值。在最新的 Opera 浏览器中，其显示效果如下：



有点遗憾，浏览器没有在滑动条上显示刻度对应的数值大小。这样，用户也就无从得知自己选择的数值。解决这个问题很简单，添加 onchange 处理函数以便显示区域能够基于 range 控件值的改变而改变，具体实现见代码清单 7-1。

#### 代码清单 7-1 用于更新数值显示的 onchange 处理函数

```
<script type="text/javascript">
```

```
function showValue(newVal) {
    document.getElementById("ageDisplay").innerHTML = newVal;
}
</script>

<label for="age">Age</label>
<input id="age" type="range" min="18" max="120" value="18" onchange="showValue(this.value)">
<span id="ageDisplay">18</span>
```

显示效果好多了，如下图所示。

Age  44

Opera 和基于 WebKit 的浏览器 (Safari 和 Chrome) 现已支持 range 类型的 input 元素。Firefox 浏览器有对 range 类型的支持计划，但在本书写作过程中还未见进展。在 Firefox 浏览器中使用 range 类型时，目前只会显示常规的文本输入框。

#### HERE BE DRAGONS

“有句俚语叫 ‘Here be dragons’，历史上人们会用它在地图上标出存在未知潜在危险的区域。表 7-3 所示的表单元素亦是如此。虽然规范描述得很清晰，而且它们也已经存在一段时间了，但其中的大多数元素仍然缺乏在实践中的应用。

期待有一天，浏览器开发者可以改进设计、弥合缺憾并响应反馈意见。届时，表 7-3 所示的表单元素会发生翻天覆地的变化。与其在开发中应用这些元素，倒不如将它们看做是 HTML5 表单发展的风向标。如果你坚持现在就用它们，那么后果自负……”

——Brian

还有一些已列入规范但尚未得到广泛支持的表单元素，见表 7-3。

表7-3 未来的HTML5 Forms元素

类 型	用 途
number	只能包含数值的文本框
color	颜色选择器，基于调色盘或者取色板进行选择
datetime	显示完整的日期和时间，包括时区，如图7-3所示
datetime-local	显示日期和时间，不含时区
time	不含时区的时间选择器和指示器
date	日期选择器
week	某年中的周选择器
month	某年中的月选择器

尽管一些先进的浏览器已经初步支持这些元素 (例如图 7-3 中是 Opera 的 datetime 元素的显示效果)，不过，本章不会把重点放在它们身上，因为这些元素有可能发生巨大的变化。请继续关注未来的版本吧！



Tel#:

E-mail:

DOB: 1944-06-06

T-shirt Size

Week	Mon	Tue	Wed	Thu	Fri	Sat	Sun
22	29	30	31	1	2	3	4
23	5	6	7	8	9	10	11
24	12	13	14	15	16	17	18
25	19	20	21	22	23	24	25
26	26	27	28	29	30	1	2
27	3	4	5	6	7	8	9

Shirt style:

图 7-3 datetime 输入框显示效果

## 7.2 使用 HTML5 Forms API

熟悉了新的表单元素类型之后，接下来我们介绍新旧表单控件中都有的特性和 API。开发功能强大的 Web 应用程序的用户界面时，代码量通常会很大，而表单控件的很多特性和 API 旨在有效减少代码量。使用某些新特性，你甚至可以开发出前所未有的界面效果，即使在最差的情况下，也可以省去页面中大量的脚本代码。

### 7.2.1 新的表单特性和函数

我们先来讨论 HTML5 新增的特性、函数和元素。如同新增的输入型控件一样，不管目标浏览器是否支持新增特性，我们都可以放心使用它们。因为市面上的浏览器在不支持这些特性时，会直接忽略它们，而不是报错。

#### 1. placeholder

当用户还没有输入值的时候，输入型控件可以通过 placeholder 特性向用户显示描述性说明或者提示信息。这在目前流行的用户界面框架中很常见，而主流 JavaScript 框架也提供了类似的功能。不过，有了 placeholder 特性以后，现代浏览器就可以内置这项功能了。

使用 placeholder 特性只需将说明性文字作为该特性值即可。除了普通的文本输入框外，email、number、url 等其他类型的输入框也都支持 placeholder 特性。

```
<label>Runner: <input name="name" placeholder="First and last name" required></label>
```

在谷歌 Chrome 等支持 placeholder 特性的浏览器中，特性值会以浅灰色的样式显示在输入框中，当页面焦点切换到输入框中，或者输入框中有值了以后，该提示信息就会消失。

Runner:

在不支持 placeholder 的浏览器中运行时，此特性会被忽略，以输入型控件的默认方式显示。

Runner:

类似地，在输入值的时候，placeholder 文本也不会出现。

Runner:

## 2. autocomplete

早在 Internet Explorer 5.5 就已经引入的 autocomplete 特性如今终于标准化了。万岁！（尽管浏览器从它出现时就一直提供支持，但规范化的行为对开发人员都有好处。）

浏览器通过 autocomplete 特性能够知晓是否应该保存输入值以备将来使用。例如不保存的代码如下：

```
<input type="text" name="creditcard" autocomplete="off">
```

autocomplete 特性应该用来保护敏感用户数据，避免本地浏览器对它们进行不安全地存储。表 7-4 显示了 autocomplete 的不同行为。

表 7-4 输入型控件的 autocomplete 行为

类 型	作 用
on	该字段无需受到保护，值可以被保存和恢复
off	该字段需要受到保护，值不可以被保存
unspecified	包含<form>的默认设置。如果没有被包含在表单中或没有指定值，则行为与设置为on时相同

## 3. autofocus

页面载入时，开发人员通过 autofocus 特性可以指定某个表单元素获得输入焦点。每个页面上只允许出现一个 autofocus 特性。如果设置了多个 autofocus 特性，则相当于未指定此行为。

**提示** 如果页面内容的呈现依赖于门户页面或者共享内容页面，那么很难做到每个页面只有一个 autofocus 控件。所以，如果你无法完全控制整个页面，就不要指望基于 autofocus 特性获取焦点。

为了让控件（如搜索文本输入框）自动获取焦点，只需在元素中设置 autofocus 特性即可：

```
<input type="search" name="criteria" autofocus>
```

与其他布尔类型特性一样，使 autofocus 特性生效不需要专门将其值设为 true。





**提示** 在用户不希望焦点转移的情况下使用 autofocus 会惹恼用户。很多用户习惯于用键盘导航，这时切换表单控件的焦点会适得其反。最佳方案是仅当表单取全部默认值的情况下，才使用 autofocus 特性。

#### 4. list 特性和 datalist 元素

通过组合使用 list 特性和 datalist 元素，开发人员能够为某个输入型控件构造一张选值列表，其使用方法如下。

(1) 创建 id 特性值唯一的 datalist 元素，该元素可插入文档的任意位置。

(2) 添加若干 option 元素作为 datalist 元素的子元素，它们表示某控件推荐选值的全集。例如，datalist 元素若表示电子邮件列表，那么每个备选的 e-mail 地址都应该放入 option 元素中，代码如下：

```
<datalist id="contactList">
  <option value="x@example.com" label="Racer X">
  <option value="peter@example.com" label="Peter">
</datalist>
```

(3) 将 input 元素的 list 特性值设为 datalist 元素的 id 值，可以实现二者的关联。

```
<input type="email" id="contacts" list="contactList">
```

在支持 list 特性和 datalist 元素的浏览器（如 Opera）中，自定义列表控件显示效果如下：

Contacts	
x@example.com	Racer X
peter@example.com	Peter

#### 5. min 和 max

在之前的 `<input type="range">` 示例中我们看到，通过设置 min 和 max 特性，可以将 range 输入框的数值输入范围限定在最低值和最高值之间。这两个特性既可以只设置一个，也可以两个都设置，还可以都不设置，输入型控件会根据设置的参数对值范围做出相应调整。例如，创建一个表示信心大小的 range 控件，值范围从 0% 至 100%，代码如下所示：

```
<input id="confidence" name="level" type="range" min="0" max="100" value="0">
```

上述代码会创建一个最小值为 0、最大值为 100 的 range 控件，无巧不成书，默认值恰好也是 0 和 100。

#### 6. step

对于输入型控件，设置其 step 特性能够指定输入值递增或递减的粒度。例如，按以下方式将表示信心大小的 range 控件的 step 特性设置为 5：

```
<input id="confidence" name="level" type="range" min="0" max="100" step="5" value="0">
```

设置完成后，控件可接受的输入值只能是初始值与 5 的倍数之和。换句话说，只能输入 0、

5、10……100，至于是由输入框还是滑动条输入则由浏览器决定。

step 特性的默认值取决于控件的类型。对于 range 控件，step 默认值为 1。为了配合 step 特性，HTML5 引入了 stepUp 和 stepDown 两个函数对其进行控制。

不出所料，两个函数的作用分别是根据 step 特性的值来增加或减少控件的值。如此一来，用户不必输入就能够调整输入型控件的值了。

### 7. valueAsNumber 函数

valueAsNumber 函数的作用是完成控件值类型在文本与数值间的相互转换。它既是 getter 函数又是 setter 函数。作为 getter 函数调用时，valueAsNumber 函数将文本类型转换成 number 类型，以方便计算。如果转换失败，则会返回 NaN 值 (Not-a-Number)。

valueAsNumber 也可用于为输入型控件设置 number 值。例如，上述表示信心大小的 range 控件可以这样设置：

```
document.getElementById("confidence").valueAsNumber(65);
```

输入的数值必须同时满足 min、max 和 step 要求，否则会抛出错误。

### 8. required

一旦为某输入型控件设置了 required 特性，那么此项必填，否则无法提交表单。以文本输入框为例，要将其设置为必填项，按照如下方式添加 required 特性即可：

```
<input type="text" id="firstname" name="first" required>
```

如果此文本框中没有值，则无论以编程方式还是用户操作都将无法提交表单。required 特性是最简单的一种表单验证方式，不过表单验证功能远不止于此。接下来，我们就详细的讨论。

## 7.2.2 表单验证

在深入探讨表单验证之前，让我们先思考一下表单验证的真实含义。就其核心而言，表单验证是一套系统，它为终端用户检测无效的控件数据并标记这些错误。换言之，表单验证就是在表单提交服务器前对其进行一系列的检查并通知用户纠正错误。

但是真正的表单验证是什么？

是一种优化。

之所以说表单验证是一种优化，是因为仅通过表单验证机制不足以保证提交给服务器的表单数据是正确和有效的。另一方面，设计表单验证是为了让 Web 应用更快地抛出错误。换句话说，最好利用浏览器内置的处理机制来告知用户网页内包含无效的表单控件值。过去，数据在网络上传一圈，仅仅是为了让服务器通知用户他输入了错误的的数据。如果浏览器完全有能力让错误在离开客户端之前就被捕获到，那么我们应该利用这个优势。

不过，浏览器的表单检查还不足以处理所有的错误。

### 恶意还是误解

“尽管 HTML5 规范大幅提升了浏览器内置表单检测方面的能力，但是仍然不能完全取代服



务器端验证。或许永远都不可能。

显然，许多错误条件是需要与服务器端交互才能验证，比如信用卡是否被授权，甚至是基本的验证。然而，即使是普通验证也不能只依赖客户端。用户使用的浏览器可能本身就不支持表单验证。少数用户会屏蔽脚本功能，导致基于特性值的基本验证外的验证功能都无法使用。还有一些用户会利用诸如 Firefox Greasemonkey 等插件修改页面内容，进而导致表单验证功能失效。归根结底，重要数据的验证不能仅依靠客户端验证一种方式。只要数据存储在客户端，它就可以被篡改。

HTML5 的表单验证可让用户快速获得重要反馈，但正确性方面绝对不应依赖于它！”

——Brian

话虽如此，HTML5 还是引入了八种用于验证表单控件的数据正确性的方法。让我们依次了解一下，不过先要介绍一下用于反馈验证状态的 `ValidityState` 对象。

在支持 HTML5 表单验证的浏览器中，可以通过表单控件来访问 `ValidityState` 对象：

```
var valCheck = document.myForm.myInput.validity;
```

这行代码获取了名为 `myInput` 的表单元素的 `ValidityState` 对象。对象包含了对所有八种验证状态的引用，以及最终验证结果。

调用方式如下：

```
valCheck.valid
```

执行完毕，我们会得到一个布尔值，它表示表单控件是否已通过了所有的验证约束条件。可以把 `valid` 特性看做是最终验证结果：如果所有八个约束条件都通过了，`valid` 特性就是 `true`。否则，只要有一项约束没通过，`valid` 标志都是 `false`。

---

**提示** `ValidityState` 对象是一个实时更新的对象。获得某表单元素的 `ValidityState` 对象后，当表单元素内容发生变化时，你可以通过它来获得更新后的检测结果。

---

如前所述，任何表单元素都有八个可能的验证约束条件。每个条件在 `ValidityState` 对象中都有对应的特性名，可以用适当的方式访问。让我们逐一分析，看看它们是如何与表单控件关联的，以及如何基于 `ValidityState` 对象来对它们进行检查：

#### `valueMissing`

目的：确保表单控件中的值已填写。

用法：在表单控件中将 `required` 特性设置为 `true`。

示例：`<input type="text" name="myText" required>`

详细说明：如果表单控件设置了 `required` 特性，那么在用户填写或者通过代码调用方式填值之前，控件会一直处于无效状态。例如，空的文本输入框无法通过必填检查，除非在其中输入任意文本。输入值为空时，`valueMissing` 会返回 `true`。

### typeMismatch

目的：保证控件值与预期类型相匹配（如 number、email、URL 等）。

用法：指定表单控件的 type 特性值。

示例：`<input type="email" name="myEmail">`

详细说明：特殊的表单控件类型不只是用来定制手机键盘！如果浏览器能够识别出来表单控件中的输入不符合对应的类型规则——比如 email 地址中没有 @ 符号，或者 number 型控件的输入值不是有效的数字——那么浏览器就会把这个控件标记出来以提示类型不匹配。无论哪种出错情况，typeMismatch 将返回 true。

### patternMismatch

目的：根据表单控件上设置的格式规则验证输入是否为有效格式。

用法：在表单控件上设置 pattern 特性，并赋予适当的匹配规则。

示例：`<input type="text" name="creditcardnumber" pattern="[0-9]{16}" title="A credit card number is 16 digits with no spaces or dashes">`

详细说明：pattern 特性向开发人员提供了一种强大而灵活的方式来为表单的控件值设定正则表达式验证机制。当为控件设置了 pattern 特性后，只要输入控件的值不符合模式规则，patternMismatch 就会返回 true 值。从引导用户和技术参考两方面考虑，你应该在包含 pattern 特性的表单控件中设置 title 特性以说明规则的作用。

### tooLong

目的：避免输入值包含过多字符。

用法：在表单控件上设置 maxLength 特性。

示例：`<input type="text" name="limitedText" maxLength="140">`

详细说明：如果输入值的长度超过了 maxLength，tooLong 特性会返回 true。虽然表单控件通常会在用户输入时限制最大长度，但在有些情况下，如通过程序设置，还是会超出最大值。

### rangeUnderflow

目的：限制数值型控件的最小值。

用法：为表单控件设置 min 特性，并赋予允许的最小值。

示例：`<input type="range" name="ageCheck" min="18">`

详细说明：在需要做数值范围检查的表单控件中，数值很可能会暂时低于设置的下限。此时，ValidityState 的 rangeUnderflow 特性将返回 true。

### rangeOverflow

目的：限制数值型控件的最大值。

用法：为表单控件设置 max 特性，并赋予允许的最大值。

示例：`<input type="range" name="kidAgeCheck" max="12">`

详细说明：与 rangeUnderflow 类似，如果一个表单控件的值比 max 更大，特性将返回 true。

### stepMismatch

目的：确保输入值符合 min、max 及 step 设置。



用法：为表单控件设置 step 特性，指定数值的增量。

示例：`<input type="range" name="confidenceLevel" min="0" max="100" step="5">`

详细说明：此约束条件用来保证数值符合 min、max 和 step 的要求。换句话说，当前值必须是最小值与 step 特性值的倍数之和。例如，范围从 0 到 100，step 特性值为 5，此时就不允许出现 17，否则 stepMismatch 返回 true 值。

#### customError

目的：处理应用代码明确设置及计算产生的错误。

用法：调用 setCustomValidity(message) 将表单控件置于 customError 状态。

示例：`passwordConfirmationField.setCustomValidity("Password values do not match.");`

详细说明：浏览器内置的验证机制不适用时，需要显示自定义验证错误信息。当输入值不符合语义规则时，应用程序代码应设置这些自定义验证消息。

自定义验证消息的典型用例是验证控件中的值是否一致。例如，密码和密码确认两个输入框的值不匹配（在“进阶功能”一节我们将深入研究此例）。只要定制了验证消息，控件就会处于无效状态，并且 customError 返回 true。要清除错误，只需在控件上调用 setCustomValidity("") 即可。

#### 其他用于验证的特性及函数

总之，这八个约束可以让开发人员找到表单验证失败的确切原因。或者，如果你不关心失败的具体原因，那么只访问 ValidityState 对象的 valid 布尔值即可，它代表了验证的最终结果。如果所有八个约束条件都返回 false，那么 valid 特性将返回 true。此外，在编写验证代码时，表单控件还支持其他一些有用的特性和函数。

#### willValidate 特性

willValidate 特性仅用来说明某表单控件是否将进行验证。如果 required 特性、pattern 特性等设置在了控件上，那么通过 willValidate 特性，你可以得知验证将会执行。

#### checkValidity 函数

即使没有用户输入，也可以使用 checkValidity 函数对表单进行验证。一般情况下，表单验证发生在用户或者脚本提交表单时，checkValidity 函数却能在任何时间对表单进行验证。

---

**提示** 在表单控件上调用 checkValidity 函数不仅会进行验证，还会触发所有结果事件和 UI 触发器，就好像表单已经提交了一样。

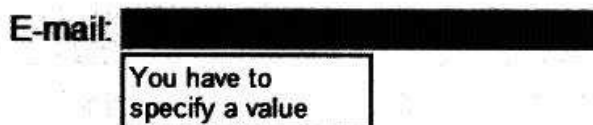
---

#### validationMessage 特性

撰写此书的时候还没有一个浏览器支持 validationMessage 特性，不过在你阅读本书的时候，很可能已经有了。validationMessage 特性允许你通过编程方式查询本地化错误信息，浏览器基于当前验证状态显示的也是这些信息。例如，如果 required 特性没有值，则浏览器可能会提示用户：“这是必填字段”（This field requires a value）。如果浏览器支持 validationMessage 特性，这段提示信息会由 validationMessage 返回并依据控件当前验证状态进行调整。

### 7.2.3 验证反馈

关于验证反馈,有一个到目前为止我们一直回避的问题——浏览器怎样以及何时将验证错误信息反馈给用户呢?规范中没有规定用户界面如何展示错误信息,而且目前各浏览器的展现方式也不尽相同。以 Opera 为例,在 Opera 10.5 中,浏览器会通过弹出提示消息并将字段置红的方式来标记产生错误的字段。



相比之下,谷歌 Chrome 浏览器只会在发现错误时将焦点定位到出问题的控件。哪种做法正确呢?

二者均非标准。不过,如果开发人员想控制反馈给用户的错误消息,可以使用一个合适的处理函数:invalid 事件。

只要发生表单验证(不管是在提交表单时,还是直接调用 checkValidity 函数),所有未通过验证的表单都会接收到一个 invalid 事件。invalid 事件可以被忽略、观察、甚至取消。可以为接收此通知的字段添加 invalid 事件处理函数,见代码清单 7-2:

代码清单 7-2 为 invalid 事件添加处理函数

```
//invalid 事件的事件处理函数
function invalidHandler(evt) {
    var validity = evt.srcElement.validity;

    //检测有效性,看是否某个特定的验证条件没有通过
    if (validity.valueMissing) {
        // 提示用户必填项中没有填值
    }

    // 可能检测更多的约束条件……

    // 如果不希望浏览器提供默认的验证反馈,按照下面的方式取消事件
    evt.preventDefault();
}

//注册 invalid 事件的监听器
myField.addEventListener("invalid", invalidHandler, false);
```

现在让我们分析一下上面的代码。

首先,声明了一个事件处理函数来接收 invalid 事件。在该函数内,首先要做的是检查事件来源。之前我们已经了解到,表单控件验证错误触发了 invalid 事件。因此,事件的 srcElement 应该是发生错误的表单控件。

从事件来源中,我们得到了 validity 对象。使用 validityState 实例,通过检查各个约束条



件，确定错误的出处。在示例中，因为我们知道字段有 `required` 特性，所以首先要检查是否满足 `valueMissing` 约束条件。

如果验证成功，我们可以修改用户界面，提示用户在必填字段中输入值。开发人员可以决定出错信息的显示方式——置于弹出警告框中或者显示在页面特定区域上。

在我们告知用户出了什么错，如何改正后，接下来要决定是否需要浏览器显示内置的反馈信息。默认情况下，浏览器会自动显示内置的反馈信息。为了阻止浏览器显示默认的错误信息，我们调用了 `evt.preventDefault()` 以阻止浏览器的默认行为，并自行处理错误信息的显示。

从这份代码中我们再次看到，开发人员的选择余地很大。HTML5 Forms API 让你的开发更具灵活性，既可以使用定制的 API，又可以使用默认的浏览器行为。

#### 关闭验证

不管验证 API 背后的功能有多么强大，但还是有一些原因，促使你关闭对某个控件或者整个表单的验证。最常见的情况是暂存表单内容或者供将来查阅，这时，即使内容无法通过验证也没关系。

想象一下这样的场景，用户进入了一个复杂的订单录入表单，填写到一半时忽然有急事需要处理。理想的情况下，你可能会向用户提供一个“保存”按钮，点击后将已填的信息提交至服务器保存。但如果表单没有全部完成，验证规则就会阻止内容的提交。用户会非常失望，因为他必须选择要么继续完成表单，要么放弃已经填完的部分。

为了解决类似问题，表单本身可以通过代码方式设置 `noValidate` 特性。这样一来，所有的验证逻辑都会被放弃，只会单纯地提交表单。这个特性可以通过脚本设置，也可以直接标记。

关闭验证更好的办法是在如表单提交按钮 (`type` 特性值为 `submit`) 这样的控件上设置 `formNoValidate` 特性。下面的示例在名为“save”的提交按钮上设置了 `formNoValidate` 特性：

```
<input type="submit" formnovalidate name="save" value="Save current progress">  
<input type="submit" name="process" value="Process order">
```

乍一看，代码创建了两个普通的提交按钮。第二个按钮会正常提交表单。但是，第一个按钮设置了 `noValidate` 特性，所以点击该按钮会绕过所有的验证。换句话说，在数据被提交到服务器之前，不会再对其进行正确性检查了。当然，服务器端还需要做一些相应的设置以处理未经验证的数据，但最佳做法是任何时候都应做这些设置。

## 7.3 构建 HTML5 Forms 应用

现在，让我们利用本章描述的工具，创建一个简单的注册页面，以此来演示 HTML5 Forms 的新特性。再回到我们所熟悉的跑步俱乐部应用，我们将为这家俱乐部创建一个比赛注册页面，其中会包含本章介绍的新的表元素和验证功能。

一如既往，示例文件的源代码位于 `code/forms` 文件夹。我们无需花费太多精力关注 CSS 和次要的标记，重点关注页面的核心部分即可。照例，我们先看看图 7-4 所示的最终页面，然后将其拆成一系列片段分别讨论。

图 7-4 比赛注册表单页面示例

上面的注册页面包含了本章所探讨的众多元素和 API，包括验证功能。虽然不同的浏览器会导致页面实际显示效果上的差异，但是在浏览器不支持某个功能时，页面会平滑退化。

现在来看看代码。

在之前的示例中，我们已经讨论过页眉、导航、页脚。现在，这个页面包含了一个<form>元素。

```
<form name="register">
  <p><label for="runnername">Runner:</label>
    <input id="runnername" name="runnername" type="text"
      placeholder="First and last name" required</p>
  <p><label for="phone">Tel #:</label>
    <input id="phone" name="phone" type="tel"
      placeholder="(xxx) xxx-xxx"></p>
  <p><label for="emailaddress">E-mail:</label>
    <input id="emailaddress" name="emailaddress" type="email"
      placeholder="For confirmation only"></p>
  <p><label for="dob">DOB:</label>
    <input id="dob" name="dob" type="date"
      placeholder="MM/DD/YYYY"></p>
```

在这一部分中，我们可以看到四个主要的 input 标签：姓名 (name)、电话 (phone)、电子邮件 (email) 和生日 (birthday)。对于每个 input 标签，我们都为其设定了一个带描述信息的 <label>，并通过 for 特性将它们绑定到了实际控件上。我们还设置了 placeholder 特性以向用户



示范应该填写什么样的内容。

对于姓名文本输入框，我们为其设定了 `required` 特性来使其成为必填项。如果未输入任何值，将会触发表单验证的 `valueMissing` 约束条件。在电话文本输入框中，我们将其声明为 `tel` 类型。有些浏览器支持 `tel` 类型，有些则不支持，还有些浏览器能够提供优化过的键盘<sup>①</sup>。

同样，电子邮件文本输入框被标记为 `email` 类型。处理方式取决于浏览器版本。若输入值不是一个有效的 `email` 地址，则某些浏览器会抛出 `typeMismatch` 约束异常。

最后，生日文本输入框被声明成了 `date` 类型。现在还没有多少浏览器支持这一特性，但将来一旦支持的话，浏览器会自动呈现一个日期选择控件供用户使用。

```
<fieldset>
  <legend>T-shirt Size: </legend>
  <p><input id="small" type="radio" name="tshirt" value="small">
    <label for="small">Small</label></p>
  <p><input id="medium" type="radio" name="tshirt" value="medium">
    <label for="medium">Medium</label></p>
  <p><input id="large" type="radio" name="tshirt" value="large">
    <label for="large">Large</label></p>
  <p><label for="style">Shirt style:</label>
    <input id="style" name="style" type="text" list="stylelist" title="Years of
      participation"></p>
  <datalist id="stylelist">
    <option value="White" label="1st Year">
    <option value="Gray" label="2nd - 4th Year">
    <option value="Navy" label="Veteran (5+ Years)">
  </datalist>
</fieldset>
```

在上面的代码段中，我们设置了一些控件，用于选择 T-shirt。第一组控件是标准的单选按钮，用于选择 T-shirt 尺寸。

随后的代码更为有趣。我们用到了 `list` 特性及其关联的 `<datalist>` 元素。在 `<datalist>` 中，我们声明了一系列用于列表显示的 T-shirt 类型，并使用了不同的 `value` 特性和 `label` 特性来区分它们，不同的 T-shirt 类型代表了不同的参赛年龄段。尽管这个列表非常简单，但其中的技术同样适用于列表长度动态变化的情况。

```
<fieldset>
  <legend>Expectations:</legend>
  <p>
    <label for="confidence">Confidence:</label>
    <input id="confidence" name="level" type="range"
      onchange="setConfidence(this.value)"
      min="0" max="100" step="5" value="0">
    <span id="confidenceDisplay">0%</span></p>
  <p><label for="notes">Notes:</label>
    <textarea id="notes" name="notes" maxLength="140"></p>
</fieldset>
```

<sup>①</sup> 主要指 iPhone 等产品上使用的触摸屏键盘。——译者注

最后一段代码中，我们为用户创建了一个滑动条，来表示他对在比赛中取得名次的信心。为此，我们使用了 range 类型的输入框。示例中信心以百分比的方式加以衡量，所以我们在输入控件上设置了 min、max 和 step 特性。这样，用户的信心比例就会被强制约束在正常范围内。此外，我们将信心比例变化的步长设为 5%，如果浏览器支持 range 特性并以滑动条形式展现，那么你可以很直观地看到其效果。尽管简单的控件交互不太可能触发它们，但控件上仍有一些验证约束，如 rangeUnderflow、rangeOverflow 和 stepMismatch。

因为 range 控件默认无法显示其所代表的数值，所以我们需要为应用添加此功能。range 控件的 onchange 处理程序会操纵 id 为 confidenceDisplay 的 span 标签以显示数值，不过，想看效果还要再等一会<sup>①</sup>。

在这段代码的最后，我们添加了一个 <textarea> 标签，用于填写注册中的附加信息。通过在 textarea 元素上设置 maxLength 特性，我们为 textarea 元素添加了 tooLong 约束条件检测，因为你无法保证用户不会将一大段文本粘贴到 textarea 元素中。

```
<p><input type="submit" name="register" value="Register"></p>
</form>
```

最后，添加一个用以提交注册表单的按钮来完成整个控件部分的编写。示例中，注册信息实际上不会发往任何服务器。

还差一些脚本代码没有编写：如何更新信心滑动条的显示，以及如何覆盖浏览器内置的表单验证反馈。尽管浏览器默认的表单异常处理机制能够满足开发的要求，但了解其他选择也是有必要的。

```
<script type="text/javascript">
  function setConfidence(newVal) {
    document.getElementById("confidenceDisplay").innerHTML = newVal + '%';
  }

  function invalidHandler(evt) {
    // 获取表单控件对应的 label 标签
    var label = evt.srcElement.parentElement.getElementsByTagName("label")[0];

    // 将 label 标签的文本设置为红色
    label.style.color = 'red';

    // 阻止事件冒泡
    evt.stopPropagation();

    // 停止浏览器默认的验证处理函数
    evt.preventDefault();
  }

  function loadDemo() {
    // 在表单上注册事件处理函数，以处理所有未通过验证的控件通知
    document.register.addEventListener("invalid", invalidHandler, true);
  }
</script>
```

<sup>①</sup> 作者的意思是随后会添加相应的脚本代码，那时才会看到效果。——译者注



```
window.addEventListener("load", loadDemo, false);  
</script>
```

上面这段代码包含下面几部分。首先是 `setConfidence()` 函数，用于处理 `range` 控件的 `change` 事件。每次滑动条值发生变化，都会找到 `id` 为 `confidenceDisplay` 的 `<span>` 并更新相应文本。从示例代码中可以看到，配置非常简单。

---

**提示** 为什么 `range` 控件默认不显示数值呢？可能是为了让用户界面设计师能够定制精确的显示位置和显示外观。虽然把显示工作分离出来增加了工作量，但却更灵活了。

---

代码的最后展示了如何覆盖浏览器默认的验证异常处理。首先，注册事件监听器以监听 `invalid` 事件。为了在所有表单控件范围内捕获 `invalid` 事件，我们把处理函数注册在了表单上。再次提醒，一定要注册事件监听器，否则无法监听到 `invalid` 事件。

```
// 在表单上注册事件处理函数，以处理所有未通过验证的控件通知  
document.register.addEventListener("invalid", invalidHandler, true);
```

现在，只要表单元素触发了验证约束条件，`invalidHandler` 就会被调用。为了向用户提供比浏览器默认设置更加细致的反馈，我们将不合法的表单控件对应的 `label` 文本标为红色。实现这种效果先要通过表单控件的父节点找到相关的 `<label>` 标签。

```
// 获取表单控件对应的 label 标签  
var label = evt.srcElement.parentElement.getElementsByTagName("label")[0];  
  
// 将 label 字体颜色设置为 red  
label.style.color = 'red';
```

将 `label` 标签中的说明文本设置为红色后，我们要阻止浏览器或其他 `invalid` 事件处理函数对 `invalid` 事件重复处理。利用 DOM 的强大功能，我们调用了 `preventDefault()` 函数来停止浏览器默认的事件处理程序，并调用 `stopPropagation()` 阻止其他处理程序处理 `invalid` 事件。

```
// 阻止事件冒泡  
evt.stopPropagation();  
  
// 停止浏览器默认的验证错误处理函数  
evt.preventDefault();
```

只是寥寥几步，我们就用定制的前端验证代码创建了一个带有验证功能的表单！

## 进阶功能

尽管有些技巧我们在示例中用不上，但这并不妨碍它们在多种类型的 HTML5 Web 应用中发挥作用。本节，我们就来介绍一些简单实用的进阶功能。

**密码是：Validation!**

HTML5 表单验证为常见的密码确认功能提供了一种便捷的实现方式。以往，标准的做法是

在页面上提供两个密码输入框，只有当二者一致时方能提交表单。现在，我们使用一种新方法，在表单提交之前，调用 `setCustomValidation` 函数来确保两个密码输入框中的内容相匹配。

之前我们提到，当标准约束规则不适用时，`customError` 可以用来处理表单控件中的错误。特别是当验证依赖于多个控件并行触发时，`customError` 约束条件的应用优势更为明显，密码确认无疑是一个很好的示例。

`ValidityState` 对象引用被获取后，会一直处于激活状态，所以当密码字段不匹配时，应在 `ValidityState` 对象上设置自定义错误信息，而当密码字段相匹配时，立刻清除自定义错误信息。通过为密码字段添加 `onchange` 事件处理函数可以实现上述功能。

```
<form name="passwordChange">
  <p><label for="password1">New Password:</label>
  <input type="password" id="password1" onchange="checkPasswords()"></p>
  <p><label for="password2">Confirm Password:</label>
  <input type="password" id="password2" onchange="checkPasswords()"></p>
</form>
```

从上面的代码中可以看到，在包含两个密码字段的表单中，只要其中任何一个密码值发生变化，就会触发我们注册的事件。

```
function checkPasswords() {
  var pass1 = document.getElementById("password1");
  var pass2 = document.getElementById("password2");

  if (pass1.value != pass2.value)
    pass1.setCustomValidity("Your passwords do not match. Please recheck that your
      new password is entered identically in the two fields.");
  else
    pass1.setCustomValidity("");
}
```

这里提供了一种处理密码匹配的方法。取得两个密码字段的值，加以比较，如果它们不匹配，则抛出一个自定义错误。出于验证的目的，代码只需对其中一个密码字段设置错误信息。如果两个密码字段匹配，自定义错误信息会被空字符串所替换，这样就能够消除自定义错误。

在表单控件上设置错误信息后，你可以使用本章之前描述的方法向用户显示反馈信息，要求用户修改密码以使其符合匹配条件。

## 7.4 小结

本章，我们看到 HTML 表单旧貌换新颜，这一切要归功于 HTML5 带来的新元素、新特性和新 API。我们看到了一些新的输入型控件，以后还会陆续增多。我们还看到了如何直接在表单控件中集成客户端验证功能，以避免错误数据在客户端和服务端来回传输。总之，在创建包含全面功能的 Web 应用界面方面，HTML5 Forms 提供了多种能够有效降低脚本代码开发量的途径。下一章，我们将研究 HTML5 Web Workers，在支持它的浏览器中，可以将长时间运行的任务放到多个独立的运行环境中去处理。



# Web Workers API

# 8

HTML5 Web Workers 可以让 Web 应用程序具备后台处理能力。它对多线程的支持性非常好，因此，使用了 HTML5 的 JavaScript 应用程序可以充分利用多核 CPU 带来的优势。将耗时长任务分配给 HTML5 Web Workers 执行，可以避免弹出脚本运行缓慢的警告，见图 8-1。图中显示的是 JavaScript 程序循环持续几秒钟后弹出的警告窗口。

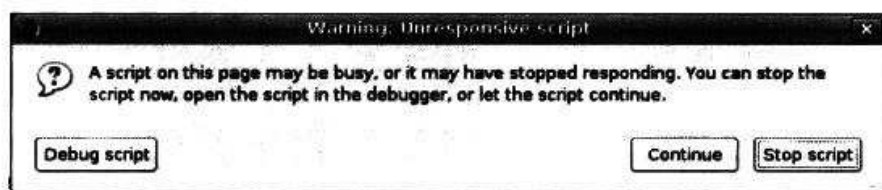


图 8-1 Firefox 中的脚本运行缓慢警告

尽管 Web Workers 功能强大，但也不是万能的，有些事情它还做不到。例如，在 Web Workers 中执行的脚本不能访问该页面的 window 对象 (window.document)，换句话说，Web Workers 不能直接访问 Web 页面和 DOM API。虽然 Web Workers 不会导致浏览器 UI 停止响应，但是仍然会消耗 CPU 周期，导致系统反应速度变慢。

如果开发人员创建的 Web 应用程序需要执行一些后台数据处理，但又不希望这些数据处理任务影响 Web 页面本身的交互性，那么可以通过 Web Workers 生成一个 Web Worker 去执行数据处理任务，同时添加一个事件监听器去监听它发出的消息。

Web Workers 的另一个用途是监听由后台服务器广播的新闻信息，收到后台服务器的消息后，将其显示在 Web 页面上。像这种与后台服务器对话的场景中，Web Workers 可能会使用到 Web Sockets 或 Server-Sent 事件。

在本章中，我们将探讨如何使用 Web Workers。首先，讨论 Web Workers 的工作机制和在编写本书时各浏览器的支持情况。接下来，将讨论如何使用 API 创建新的 Worker 以及如何在 Worker 和生成该 Worker 的上下文之间进行通信。最后，我们会演示如何建立一个 Web Workers 应用。

## 8.1 HTML5 Web Workers 的浏览器支持情况

浏览器对 Web Workers 的支持情况各不相同，并且仍在持续更新发展中。如表 8-1 所示，在

编写本书时，Web Workers 已经得到了很多浏览器的支持。

表8-1 浏览器支持情况

浏览器	支持情况
Chrome	版本3及以上支持
Firefox	版本3.5及以上支持
Internet Explorer	暂不支持
Opera	版本10.6及以上支持
Safari	版本4及以上支持

## 8.2 使用 HTML5 Web Workers API

本节，我们将从细节上探讨 Web Workers API 的使用。为便于说明，我们创建了一个简单的页面：echo.html。Web Workers 的使用方法非常简单，只需创建一个 Web Workers 对象，并传入希望执行的 JavaScript 文件即可。另外，在页面中再设置一个事件监听器，用来监听由 Web Worker 发来的消息和错误信息。如果想要在页面与 Web Workers 之间建立通信，数据需通过 postMessage 函数传递。对于 Web Worker JavaScript 文件中的代码也是如此：必须通过设置事件处理程序来处理发来的消息和错误信息，通过调用 postMessage 函数实现与页面的数据交互。

### 8.2.1 浏览器支持性检查

在调用 Web Workers API 函数之前，首先要确认浏览器是否支持。如果不支持，可以提供一些备用信息，提醒用户使用最新的浏览器。代码清单 8-1 是可以用来测试浏览器支持性的代码。

代码清单 8-1 检查浏览器支持性的代码

```
function loadDemo() {
  if (typeof(Worker) !== "undefined") {
    document.getElementById("support").innerHTML =
      "Excellent! Your browser supports HTML5 Web Workers";
  }
}
```

这个示例中，使用 loadDemo 函数来检测浏览器的支持情况，可在页面加载时调用该函数。调用 typeof(Worker) 会返回全局 Window 对象的 Worker 属性，如果浏览器不支持 Web Workers API，返回结果将是 "undefined"。这段代码在检测了浏览器支持性之后，会将检测结果反映到页面上。通过更新页面中预先定义好的 support 元素中的内容，为用户显示适当的提示，如图 8-2 所示。

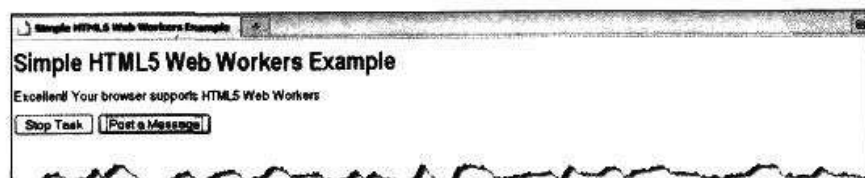


图 8-2 检测 HTML5 Web Workers 支持性的示例



## 8.2.2 创建 HTML5 Web Workers

Web Workers 初始化时会接受一个 JavaScript 文件的 URL 地址,其中包含了供 Worker 执行的代码。这段代码会设置事件监听器,并与生成 Worker 的容器进行通信。JavaScript 文件的 URL 可以是相对或者绝对路径,只要是同源(相同的协议、主机和端口)即可:

```
worker = new Worker("echoWorker.js");
```

## 8.2.3 多个 JavaScript 文件的加载与执行

对于由多个 JavaScript 文件组成的应用程序来说,可以通过包含<script>元素的方式,在页面加载的时候同步加载 JavaScript 文件。然而,由于 Web Workers 没有访问 document 对象的权限,所以在 Worker 中必须使用另外一种方法导入其他的 JavaScript 文件——importScripts:

```
importScripts("helper.js");
```

导入的 JavaScript 文件只会在某一个已有的 Worker 中加载和执行。多个脚本的导入同样也可以使用 importScripts 函数,它们会按顺序执行:

```
importScripts("helper.js", "anotherHelper.js");
```

## 8.2.4 与 HTML5 Web Workers 通信

Web Worker 一旦生成,就可以使用 postMessage API 传送和接收数据。postMessage API 还支持跨框架和跨窗口通信。大多数 JavaScript 对象都可以通过 postMessage 发送,但含有循环引用的除外。

假设要建立这样一个简单的 Web Workers 示例:用户可以向 Worker 发送信息,然后 Worker 把信息返回来。也许这个示例并没有太大的实际意义,但是其理念是构建复杂应用时所必需的。图 8-3 显示的是这个示例的 Web 页面和正在执行中的 Web Workers。对应的程序代码列于本节的末尾。

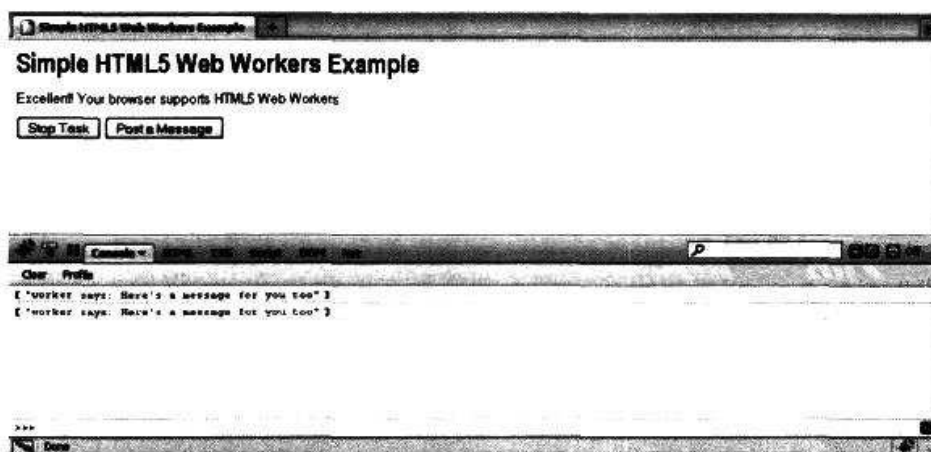


图 8-3 一个简单的使用了 HTML5 Web Workers 的 Web 页面<sup>①</sup>

<sup>①</sup> 本示例需在安装了 Firebug 的 Firefox 浏览器中测试,并请启用 Firebug 控制台。——编者注

为了能与 Web Workers 成功通信，除了要在主页（调用 Web Workers 的页面）中添加代码以外，Worker JavaScript 文件中也需要添加相应代码。

## 8.3 编写主页

为实现页面到 Web Workers 的通信，我们将调用 `postMessage` 函数以传入所需数据。同时，我们将建立一个监听器，用来监听由 Web Workers 发送到页面的消息和错误信息。

为建立主页和 Web Workers 之间的通信，首先在主页中添加对 `postMessage` 函数的调用，如下所示：

```
document.getElementById("helloButton").onclick = function() {  
    worker.postMessage("Here's a message for you");  
}
```

用户点击“Post a Message”按钮后，相应信息会被发送给 Web Workers。然后，我们将事件监听器添加到页面中，用来监听从 Web Workers 发来的信息：

```
worker.addEventListener("message", messageHandler, true);
```

```
function messageHandler(e) {  
    //处理 Worker 发来的消息  
}
```

### 编写HTML5 Web Workers JavaScript文件

在 Web Workers JavaScript 文件中，也需要添加类似的代码：必须添加事件监听器以监听从发来的消息和错误信息，并且通过调用 `postMessage` 函数实现与页面之间的通信。

为了完成页面与 Web Workers 之间的通信功能，首先，我们添加代码调用 `postMessage` 函数。例如，在 `messageHandler` 函数中可以添加如下代码：

```
function messageHandler(e) {  
    postMessage("worker says: " + e.data + " too");  
}
```

接下来，在 Web Workers JavaScript 文件中添加事件监听器，以处理从主页发来的信息：

```
addEventListener("message", messageHandler, true);
```

在这个示例中，接收到信息后会马上调用 `messageHandler` 函数以保证信息能及时返回。

### 8.3.1 处理错误

HTML5 Web Workers 脚本中未处理的错误会引发 Web Workers 对象的错误事件。特别是在调试用到了 Web Workers 的脚本时，对错误事件的监听就显得尤为重要。下面显示的是 Web Workers JavaScript 文件中的错误处理函数，它将错误记录在控制台上：

```
function errorHandler(e) {  
    console.log(e.message, e);  
}
```



为了处理错误，还必须在主页上添加一个事件监听器：

```
worker.addEventListener("error", errorHandler, true);
```

### 8.3.2 HTML5 Web Workers

Web Workers 不能自行终止，但能够被启用它们的页面所终止。开发人员都希望在不再需要 Web Workers 时回收其所占资源，比如当 Web Workers 通知主页它已执行完成的时候。另外，还有可能在用户干预的情况下取消一个运行耗时较长的任务，等等。这些情况下我们都需要终止 Web Workers，可以调用 `terminate` 函数来实现。被终止的 Web Workers 将不再响应任何信息或者执行任何其他的计算。终止之后，Worker 不能被重新启动，但可以使用同样的 URL 创建一个新的 Worker。

```
worker.terminate();
```

### 8.3.3 HTML5 Web Workers 的嵌套使用

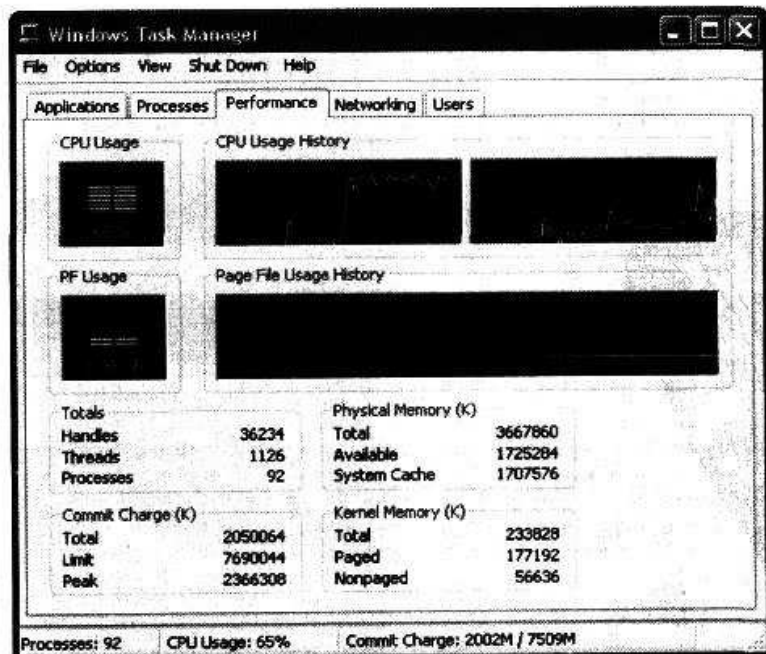
Worker 的 API 能够在 Web Workers 脚本中嵌套使用，以创建子 Worker：

```
var subWorker = new Worker("subWorker.js");
```

#### 大量的 Worker

“如果递归生成的多个 Worker 都包含了同一个 JavaScript 源文件，保守估计，你将看到一些有趣的结果，如下图所示。”

——Peter



### 8.3.4 使用定时器

虽然 HTML5 Web Workers 不能访问 window 对象,但是它可以与属于 window 对象的 JavaScript 定时器 API 协作:

```
var t = setTimeout(postMessage, 2000, "delayed message");
```

### 8.3.5 示例代码

为完整起见,代码清单 8-2 与代码清单 8-3 中展示了上述页面及其 Web Workers JavaScript 文件的源代码。

代码清单 8-2 调用 HTML5 Web Workers 的 HTML 页面代码

```
<!DOCTYPE html>
<title>Simple HTML5 Web Workers Example</title>

<link rel="stylesheet" href="styles.css">

<h1>Simple HTML5 Web Workers Example</h1>

<p id="support">Your browser does not support HTML5 Web Workers.</p>

<button id="stopButton" >Stop Task</button>
<button id="helloButton" >Post a Message</button>

<script>
  function stopWorker() {
    worker.terminate();
  }

  function messageHandler(e) {
    console.log(e.data);
  }

  function errorHandler(e) {
    console.warn(e.message, e);
  }

  function loadDemo() {
    if (typeof(Worker) !== "undefined") {
      document.getElementById("support").innerHTML =
        "Excellent! Your browser supports HTML5 Web Workers";

      worker = new Worker("echoWorker.js");
      worker.addEventListener("message", messageHandler, true);
      worker.addEventListener("error", errorHandler, true);
      document.getElementById("helloButton").onclick = function() {
        worker.postMessage("Here's a message for you");
      }
      document.getElementById("stopButton").onclick = stopWorker;
    }
  }
</script>
```



```
    }  
  }  
  window.addEventListener("load", loadDemo, true);  
</script>
```

#### 代码清单 8-3 HTML5 Web Workers JavaScript 文件

```
function messageHandler(e) {  
  postMessage("worker says: " + e.data + " too");  
}  
  
addEventListener("message", messageHandler, true);
```

## 8.4 构建 HTML5 Web Workers 应用

刚才我们讨论的是 Web Workers API 的各种用法。Web Workers API 到底有多强大？现在我们通过建立一个 HTML5 Web Workers 应用来演示。我们将在这个应用中实现一个带有图像模糊过滤器的 Web 页面，过滤动作将由多个 Web Workers 并行执行。图 8-4 显示了该应用程序的执行效果。

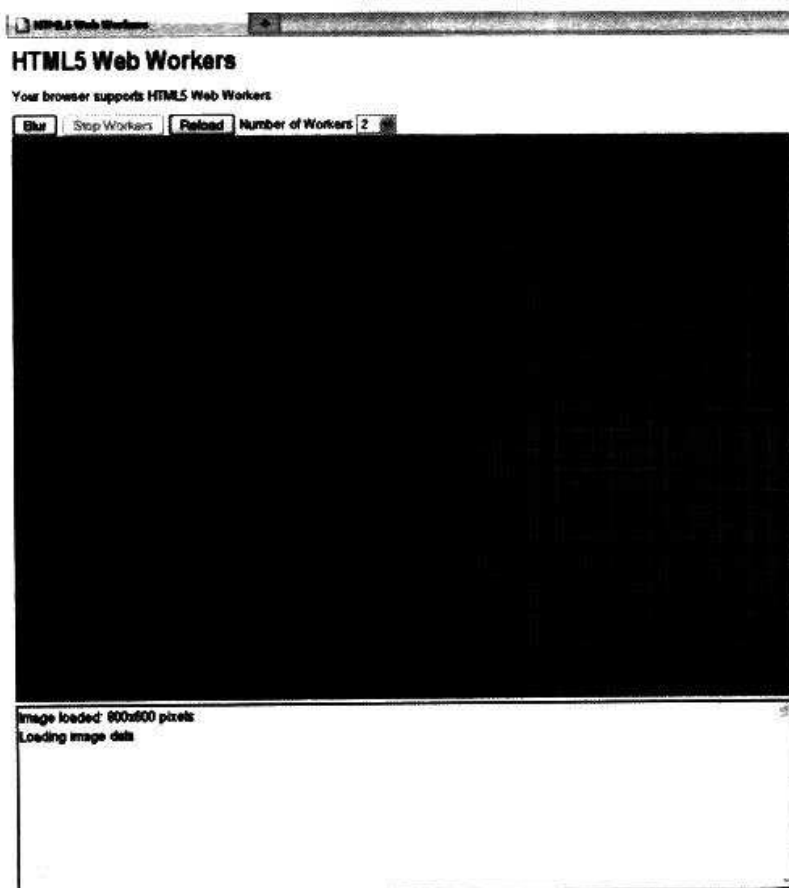


图 8-4 基于 HTML5 Web Workers 的图像模糊过滤器页面

这个应用程序首先从 canvas 向多个 Web Workers (可以指定数量) 发送图像数据, 然后 Web Workers 使用 box-blur 过滤器对这些图像数据进行处理。处理时间大约是几秒钟, 取决于图像大小和可用的计算资源 (即使电脑配备高速 CPU, 也可能因为已加载了其他进程, 导致需要更多的时钟周期来执行 JavaScript)。

Web Workers 承担了所有繁重的任务, 因此不存在弹出脚本运行缓慢警告的风险, 也不需要手动将任务分割成多份执行——在不能使用 Web Workers 的情况下, 这是必须得考虑的事情。

### 8.4.1 编写 blur.js 辅助脚本

如代码清单 8-4 所示, 在 blur.js 应用脚本中, 我们可以直接使用模糊过滤器, 使之一直循环运行直到将输入数据全部处理完。

代码清单 8-4 blur.js 文件中 JavaScript box-blur 过滤器的实现

```
function inRange(i, width, height) {
    return ((i >= 0) && (i < width*height*4));
}

function averageNeighbors(imageData, width, height, i) {
    var v = imageData[i];

    // 主方向
    var north = inRange(i-width*4, width, height) ? imageData[i-width*4] : v;
    var south = inRange(i+width*4, width, height) ? imageData[i+width*4] : v;
    var west = inRange(i-4, width, height) ? imageData[i-4] : v;
    var east = inRange(i+4, width, height) ? imageData[i+4] : v;

    // 相邻对角线
    var ne = inRange(i-width*4+4, width, height) ? imageData[i-width*4+4] : v;
    var nw = inRange(i-width*4-4, width, height) ? imageData[i-width*4-4] : v;
    var se = inRange(i+width*4+4, width, height) ? imageData[i+width*4+4] : v;
    var sw = inRange(i+width*4-4, width, height) ? imageData[i+width*4-4] : v;

    // 平均
    var newVal = Math.floor((north + south + east + west + se + sw + ne + nw + v)/9);

    if (isNaN(newVal)) {
        sendStatus("bad value " + i + " for height " + height);
        throw new Error("NaN");
    }
    return newVal;
}

function boxBlur(imageData, width, height) {
    var data = [];
    var val = 0;
    for (var i=0; i<width*height*4; i++) {
        val = averageNeighbors(imageData, width, height, i);
        data[i] = val;
    }
}
```



```
    return data;  
}
```

简而言之，该算法通过求附近像素值的平均值对图像进行模糊处理。对于一个有着百万级像素的大图像而言，需要执行相当长的时间。在 UI 线程中，运行这样的循环是极不合适的。即使不弹出脚本运行缓慢的警告，在循环终止之前，用户界面也无法响应用户的其他操作。不过，利用 Web Workers 在后台执行计算倒是可以接受的。

## 8.4.2 编写 blur.html 应用页面

代码清单 8-5 是调用 Web Workers 的 HTML 页面代码。为了方便说明，该 HTML 示例也化繁为简了。我们的目的不在于建立漂亮的界面，而是通过搭建一个简洁的框架，演示如何控制 Web Workers 并实际运行。应用程序的页面嵌入了 canvas 元素来显示输入的图像。页面上还有一组按钮，包括开始模糊 (Blur)、停止模糊 (Stop Workers)、重置图像 (Reload) 和指定生成的 Worker 数量 (Number of Workers)。

代码清单 8-5 blur.html 页面代码

```
<!DOCTYPE html>  
<title>HTML5 Web Workers</title>  
<link rel="stylesheet" href = "styles.css">  
  
<h1>HTML5 Web Workers</h1>  
  
<p id="status">Your browser does not support HTML5 Web Workers.</p>  
  
<button id="startBlurButton" disabled>Blur</button>  
<button id="stopButton" disabled>Stop Workers</button>  
<button onclick="document.location = document.location;">Reload</button>  
  
<label for="workerCount">Number of Workers</label>  
<select id="workerCount">  
  <option>1</option>  
  <option selected>2</option>  
  <option>4</option>  
  <option>8</option>  
  <option>16</option>  
</select>  
  
<div id="imageContainer"></div>  
<div id="logOutput"></div>
```

接下来在 blur.html 中添加创建 Worker 的代码。我们通过传递 JavaScript 文件的 URL 来实例化每个 Worker 对象。每个实例化的 Worker 运行的代码相同，分别负责处理输入图像的不同部分：

```
function initWorker(src) {  
    var worker = new Worker(src);
```

```

worker.addEventListener("message", messageHandler, true);
worker.addEventListener("error", errorHandler, true);
return worker;
}

```

最后为 blur.html 文件增加错误处理的代码。这样，在 Worker 发生错误的时候，页面不会毫无反应，而是显示相应的错误信息。我们的示例应该不会遇到任何问题，但在实际开发中监听错误事件是个好习惯，而且对调试工作的意义也很大。

```

function errorHandler(e) {
    log("error: " + e.message);
}

```

### 8.4.3 编写 blurWorker.js

现在，我们将 Worker 用来与页面通信的代码添加到 blurWorker.js 文件中（如代码清单 8-6 所示）。Web Workers 完成运算即可使用 postMessage 通知页面。我们将利用这个信息更新主页上的图像。Web Workers 创建完成后，会等待包含图像数据和开始模糊指令的消息，此消息是一个 JavaScript 对象，其中包含有消息类型和数值数组形式的图像数据。

代码清单 8-6 在 blurWorker.js 文件中发送和处理图像数据

```

function sendStatus(statusText) {
    postMessage({"type": "status",
                "statusText": statusText}
    );
}

function messageHandler(e) {
    var messageType = e.data.type;
    switch (messageType) {
        case ("blur"):
            sendStatus("Worker started blur on data in range: " +
                      e.data.startX + "-" + (e.data.startX+e.data.width));
            var imageData = e.data.imageData;
            imageData = boxBlur(imageData, e.data.width, e.data.height, e.data.startX);

            postMessage({"type": "progress",
                        "imageData": imageData,
                        "width": e.data.width,
                        "height": e.data.height,
                        "startX": e.data.startX
                        });
            sendStatus("Finished blur on data in range: " +
                      e.data.startX + "-" + (e.data.width+e.data.startX));
            break;
        default:
            sendStatus("Worker got message: " + e.data);
    }
}

```



```
    }  
  }  
  addEventListener("message", messageHandler, true);  
}
```

#### 8.4.4 与 Web Worker 通信

在 blur.html 文件中，我们可以通过给 Worker 发送一些代表模糊任务的数据和参数来使用 Worker。方法是使用 postMessage 函数发送一个 JavaScript 对象。这个 JavaScript 对象包含了 Worker 负责处理的 RGBA 图像数据阵列、图像的尺寸和像素范围。每个 Worker 根据接收到的信息分别对图像的不同部分进行处理：

```
function sendBlurTask(worker, i, chunkWidth) {  
  var chunkHeight = image.height;  
  var chunkStartX = i * chunkWidth;  
  var chunkStartY = 0;  
  var data = ctx.getImageData(chunkStartX, chunkStartY,  
                              chunkWidth, chunkHeight).data;  
  
  worker.postMessage({'type' : 'blur',  
                     'imageData' : data,  
                     'width' : chunkWidth,  
                     'height' : chunkHeight,  
                     'startX' : chunkStartX});  
}
```

#### CANVAS 图像数据

“postMessage 可以对 imageData 对象进行高效序列化，以便通过 canvas API 使用。不过一些支持 Worker 和 postMessage API 的浏览器也许还不能支持 postMessage 的这种扩展的序列化能力。例如，Firefox 3.5 不能通过 postMessage 传送 imageData 对象，不过未来版本可能会提供此支持。

因此，在本章中介绍的图像处理示例中，以传送 imageData.data（数据序列化方式同 JavaScript 数组一样）的方式代替传送 imageData 对象本身。在 Web Workers 执行运算任务时，它们同时将其状态和结果返回到页面上。代码清单 8-6 显示了数据如何在经模糊过滤器处理之后从 Worker 传送到页面上。同样地，消息包含了一个 JavaScript 对象，这个对象中含有图像数据和标记处理范围的坐标信息。”

——Frank

HTML 页面上的消息处理程序接收上述数据，并用新的像素值更新 canvas。处理后的图像数据到达时，HTML 页面即时显示结果。现在，我们创建了一个可以处理图像的示例应用，并且具有利用多核 CPU 的潜在优势。此外，Web Workers 执行时，程序也不会锁定用户界面，不会让用户界面停止响应。应用执行时的状态见图 8-5。

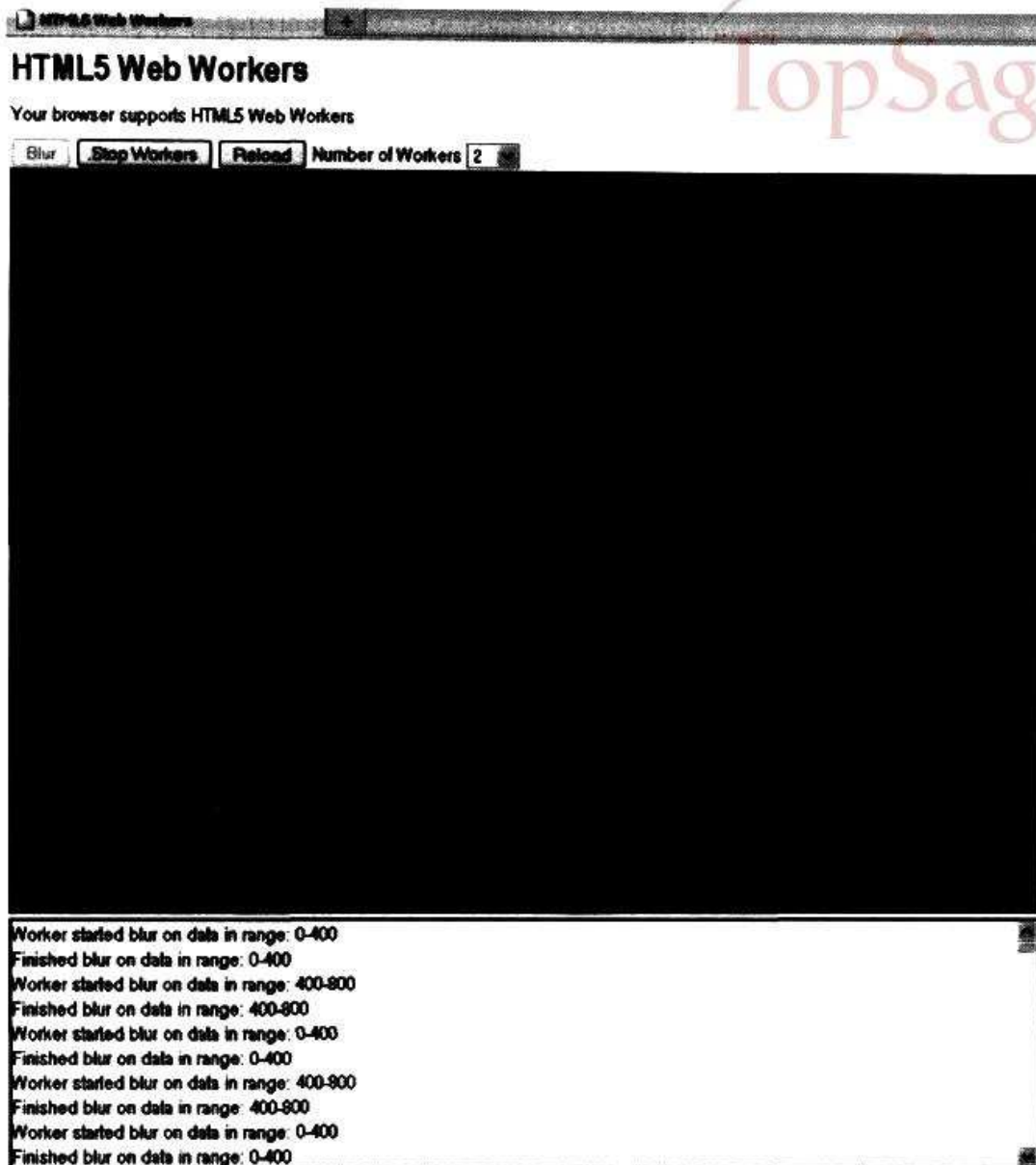


图 8-5 执行中的模糊程序

### 8.4.5 运行程序

为了让示例运行起来,blur.html 页面需要部署在一个 Web 服务器中(例如 Apache 或者 Python 的 SimpleHTTPServer)。以 Python 的 SimpleHTTPServer 为例,其部署步骤如下。

- (1) 安装 Python。
- (2) 打开包含示例文件 (blur.html) 的目录。
- (3) 输入命令启动 Python:

```
python -m SimpleHTTPServer 9999
```



(4) 打开浏览器，输入 `http://localhost:9999/blur.html`。应该就可以看到图 8-5 所示的页面了。

#### 8.4.6 示例代码

为完整起见，代码清单 8-7、代码清单 8-8 和代码清单 8-9 列出了示例的完整代码。

代码清单 8-7 blur.html 文件的内容

```
<!DOCTYPE html>
<title>HTML5 Web Workers</title>
<link rel="stylesheet" href = "styles.css">

<h1>HTML5 Web Workers</h1>

<p id="status">Your browser does not support HTML5 Web Workers.</p>

<button id="startBlurButton" disabled>Blur</button>
<button id="stopButton" disabled>Stop Workers</button>
<button onclick="document.location = document.location;">Reload</button>

<label for="workerCount">Number of Workers</label>
<select id="workerCount">
  <option>1</option>
  <option selected>2</option>
  <option>4</option>
  <option>8</option>
  <option>16</option>
</select>

<div id="imageContainer"></div>
<div id="logOutput"></div>
<script>

var imageURL = "example2.png";
var image;
var ctx;
var workers = [];

function log(s) {
  var logOutput = document.getElementById("logOutput");
  logOutput.innerHTML = s + "<br>" + logOutput.innerHTML;
}

function setRunningState(p) {
  //在运行的时候，stop 按钮可用，start 按钮不可用
  document.getElementById("startBlurButton").disabled = p;
  document.getElementById("stopButton").disabled = !p;
}

function initWorker(src) {
  var worker = new Worker(src);
  worker.addEventListener("message", messageHandler, true);
}
```

```
worker.addEventListener("error", errorHandler, true);
return worker;
}

function startBlur() {
    var workerCount = parseInt(document.getElementById("workerCount").value);
    var width = image.width/workerCount;

    for (var i=0; i<workerCount; i++) {
        var worker = initWorker("blurWorker.js");
        worker.index = i;
        worker.width = width;
        workers[i] = worker;

        sendBlurTask(worker, i, width);
    }
    setRunningState(true);
}

function sendBlurTask(worker, i, chunkWidth) {
    var chunkHeight = image.height;
    var chunkStartX = i * chunkWidth;
    var chunkStartY = 0;
    var data = ctx.getImageData(chunkStartX, chunkStartY,
                                chunkWidth, chunkHeight).data;

    worker.postMessage({'type': 'blur',
                        'imageData': data,
                        'width': chunkWidth,
                        'height': chunkHeight,
                        'startX': chunkStartX});
}

function stopBlur() {
    for (var i=0; i<workers.length; i++) {
        workers[i].terminate();
    }
    setRunningState(false);
}

function messageHandler(e) {
    var messageType = e.data.type;
    switch (messageType) {
        case ("status"):
            log(e.data.statusText);
            break;
        case ("progress"):
            var imageData = ctx.createImageData(e.data.width, e.data.height);

            for (var i = 0; i<imageData.data.length; i++) {
                var val = e.data.imageData[i];
                if (val == null || val> 255 || val< 0) {
                    log("illegal value: " + val + " at " + i);
                    return;
                }
            }
        }
    }
}
```



```
        imageData.data[i] = val;
    }
    ctx.putImageData(imageData, e.data.startX, 0);

    // 模糊相同窗口
    sendBlurTask(e.target, e.target.index, e.target.width);
    break;
default:
    break;
}
}
function errorHandler(e) {
    log("error: " + e.message);
}

function loadImageData(url) {

    var canvas = document.createElement('canvas');
    ctx = canvas.getContext('2d');
    image = new Image();
    image.src = url;

    document.getElementById("imageContainer").appendChild(canvas);

    image.onload = function(){
        canvas.width = image.width;
        canvas.height = image.height;
        ctx.drawImage(image, 0, 0);
        window.imgdata = ctx.getImageData(0, 0, image.width, image.height);
        n = ctx.createImageData(image.width, image.height);
        setRunningState(false);
        log("Image loaded: " + image.width + "x" + image.height + " pixels");
    };
}

function loadDemo() {
    log("Loading image data");

    if (typeof(Worker) !== "undefined") {
        document.getElementById("status").innerHTML = "Your browser supports HTML5 Web Workers";

        document.getElementById("stopButton").onclick = stopBlur;
        document.getElementById("startBlurButton").onclick = startBlur;

        loadImageData(imageURL);

        document.getElementById("startBlurButton").disabled = true;
        document.getElementById("stopButton").disabled = true;
    }
}
```

```
window.addEventListener("load", loadDemo, true);
</script>
```

#### 代码清单 8-8 blurWorker.js 文件内容

```
importScripts("blur.js");

function sendStatus(statusText) {
    postMessage({"type" : "status",
                "statusText" : statusText}
    );
}

function messageHandler(e) {
    var messageType = e.data.type;
    switch (messageType) {
        case ("blur"):
            sendStatus("Worker started blur on data in range: " +
                e.data.startX + "-" + (e.data.startX+e.data.width));
            var imageData = e.data.imageData;
            imageData = boxBlur(imageData, e.data.width, e.data.height, e.data.startX);

            postMessage({"type" : "progress",
                        "imageData" : imageData,
                        "width" : e.data.width,
                        "height" : e.data.height,
                        "startX" : e.data.startX
                    });
            sendStatus("Finished blur on data in range: " +
                e.data.startX + "-" + (e.data.width+e.data.startX));
            break;
        default:
            sendStatus("Worker got message: " + e.data);
    }
}

addEventListener("message", messageHandler, true);
```

#### 代码清单 8-9 blur.js 文件内容

```
function inRange(i, width, height) {
    return ((i>=0) && (i < width*height*4));
}

function averageNeighbors(imageData, width, height, i) {
    var v = imageData[i];

    // 主方向
    var north = inRange(i-width*4, width, height) ? imageData[i-width*4] : v;
    var south = inRange(i+width*4, width, height) ? imageData[i+width*4] : v;
    var west = inRange(i-4, width, height) ? imageData[i-4] : v;
    var east = inRange(i+4, width, height) ? imageData[i+4] : v;
```



```
// 相邻对角线
var ne = inRange(i-width*4+4, width, height) ? imageData[i-width*4+4] : v;
var nw = inRange(i-width*4-4, width, height) ? imageData[i-width*4-4] : v;
var se = inRange(i+width*4+4, width, height) ? imageData[i+width*4+4] : v;
var sw = inRange(i+width*4-4, width, height) ? imageData[i+width*4-4] : v;

// 平均
var newVal = Math.floor((north + south + east + west + se + sw + ne + nw + v)/9);

if (isNaN(newVal)) {
    sendStatus("bad value " + i + " for height " + height);
    throw new Error("NaN");
}
return newVal;
}

function boxBlur(imageData, width, height) {
    var data = [];
    var val = 0;
    for (var i=0; i<width*height*4; i++) {
        val = averageNeighbors(imageData, width, height, i);
        data[i] = val;
    }

    return data;
}
)
```

## 8.5 小结

本章，我们讨论了如何使用 Web Workers 搭建具有后台处理能力的 Web 应用程序。首先，我们了解了 Web Workers 的工作机制，说明了编写本书时各浏览器的支持情况。然后，我们讨论了如何使用 API 创建 Worker，以及如何在一个 Worker 与生成它的上下文之间通信。最后，我们演示了如何使用 Web Workers 构建 Web 应用程序。下一章中，我们将演示如何通过 HTML5 保存数据的本地副本，以此来减少应用的网络开销。

# Web Storage API

# 9

**本**章，我们来探索 HTML5 的 Web Storage（有时候也称为 DOMStorage）API，看看如何方便地在 Web 请求之间持久化数据。在 Web Storage API 出现之前，远程 Web 服务器需要存储客户端和服务端交互使用的所有相关数据。随着 Web Storage API 的出现，开发者可以将需要跨请求重复访问的数据直接存储在客户端的浏览器中，还可以在关闭浏览器很久后再次打开时恢复数据，以减小网络流量。

本章首先介绍 Web Storage 和 cookie 的区别，然后探讨存储和取出数据的方法。接着，我们会介绍 localStorage 和 sessionStorage 的区别，存储接口的特性、功能以及如何处理 Web Storage 事件。最后，我们会简要介绍 Web SQL Database API 和一些实用功能。

## 9.1 HTML5 Web Storage 概述

为了解释 Web Storage API，最好先回顾其前身，也就是名字很有趣的 cookie（小甜饼）。浏览器的 cookie，其名字源自一种由来已久的编程技术，即在程序间传递小数据值的 magic cookie。cookie 很神奇，它是一个在服务器和客户端间来回传送文本值的内置机制。服务器可以基于其放在 cookie 中的数据在不同 Web 页面间追踪用户的信息。用户每次访问某个域时，cookie 数据都会被来回传送。例如，cookie 可以存储会话标识符，使得 Web 服务器能够通过 cookie 中存储的同服务器端购物车数据库对应的唯一 ID，来识别哪个购物车属于当前用户。这样，用户在页面间切换时，购物车可以同步更新，保持一致。cookie 的另一个用途是将本地个性化数据存储在应用程序中，以便后续网页加载时使用。

cookie 的值也可用于用户不感兴趣的操作，例如跟踪用户浏览过的网页以挖掘广告目标客户。正因为如此，一些用户要求浏览器内置手动阻止或删除特定网站 cookie 的功能。

喜欢也好，讨厌也罢，早在 20 世纪 90 年代中期，cookie 就得到了 Netscape 浏览器的支持。cookie 也是少数几个自 Web 早期到现在浏览器厂商一直支持的功能之一。随着数据在服务器和浏览器之间传递，cookie 可以在多个请求中追踪数据。尽管 cookie 无处不在，但它还是有一些众所周知的缺点。

- cookie 的大小受限。一般来说，一个 cookie 只能设置大约 4KB 的数据，这意味着它不能接受像文件或邮件那样的大数据。



- 只要有请求涉及 cookie, cookie 就要在服务器和浏览器间来回传送。一方面, 这意味着 cookie 数据在网络上可见的, 它们在不加密的情况下有安全风险; 另一方面, 也意味着无论加载哪个相关 URL, cookie 中的数据都会消耗网络带宽。因此, 从目前情况来看, 相对较小的 cookie 意义更大。

许多情况下, 即使不使用网络或远程服务器也能达到同样的目的, 这正是 HTML5 Web Storage API 的由来。使用这一简单的 API, 开发者可以将数据存储在 JavaScript 对象中, 对象在页面加载时保存, 并且容易获取。通过使用 sessionStorage 或 localStorage, 在打开新窗口或新标签页以及重新启动浏览器时, 开发人员可以选择是否激活这些数据。存储的数据不会在网络上传输, 重新浏览网页时也容易获取到。此外, 使用 Web Storage API 可以保存高达数兆字节的大数据。因此, Web Storage 适用于存储超出 cookie 大小限制的文档和文件数据。

## 9.2 HTML5 Web Storage 的浏览器支持情况

在 HTML5 的各项特性中, Web Storage 的浏览器支持度是非常好的。其实, 目前所有主流浏览器版本都在一定程度上支持 Web Storage。从表 9-1 中可以看出, 很多浏览器已经支持 Web Storage 了。

表9-1 支持HTML5 Web Storage的浏览器

浏览器	细节
Chrome	3.0及以上版本支持
Firefox	3.0及以上版本支持
Internet Explorer	8.0及以上版本支持
Opera	10.5及以上版本支持
Safari	4.0及以上版本支持

HTML5 Web Storage 因其广泛的支持度而成为 Web 应用中最安全的 API 之一。尽管如此, 最好还是像往常一样, 在使用之前先检测浏览器是否支持 Web Storage。9.3.1 小节中将介绍如何以编程方式检查浏览器是否支持 Web Storage。

## 9.3 使用 HTML5 Web Storage API

Web Storage API 简单易用。我们先介绍数据的简单存储和获取, 然后分析 localStorage 和 sessionStorage 之间的差异。最后, 简单了解一下 Web Storage API 的高级特性, 例如数据变化时的事件通知等。

### 9.3.1 检查浏览器的支持性

在 Web Storage API 中, 特定域下的 storage 数据库可直接利用 window 对象访问。因此, 确定用户的浏览器是否支持 Web Storage API, 只要检查它是否存在 window.localStorage 或 window.sessionStorage 就行了。代码清单 9-1 可以检测浏览器是否支持 Web Storage API, 程序运行后将显示一条浏览器是否支持 Web Storage API 的消息。这段代码属于 code/storage 文件夹下的

browser-test.html 文件。除此之外，开发人员还可以使用 Modernizr 来处理一些可能会导致错误的情况，如 Chrome 匿名模式不支持 storage 等。

代码清单 9-1 检测浏览器是否支持 Web Storage

```
function checkStorageSupport() {  
    //sessionStorage  
    if (window.sessionStorage) {  
        alert('This browser supports sessionStorage');  
    } else {  
        alert('This browser does NOT support sessionStorage');  
    }  
  
    //localStorage  
    if (window.localStorage) {  
        alert('This browser supports localStorage');  
    } else {  
        alert('This browser does NOT support localStorage');  
    }  
}
```

程序运行结果如图 9-1 所示。

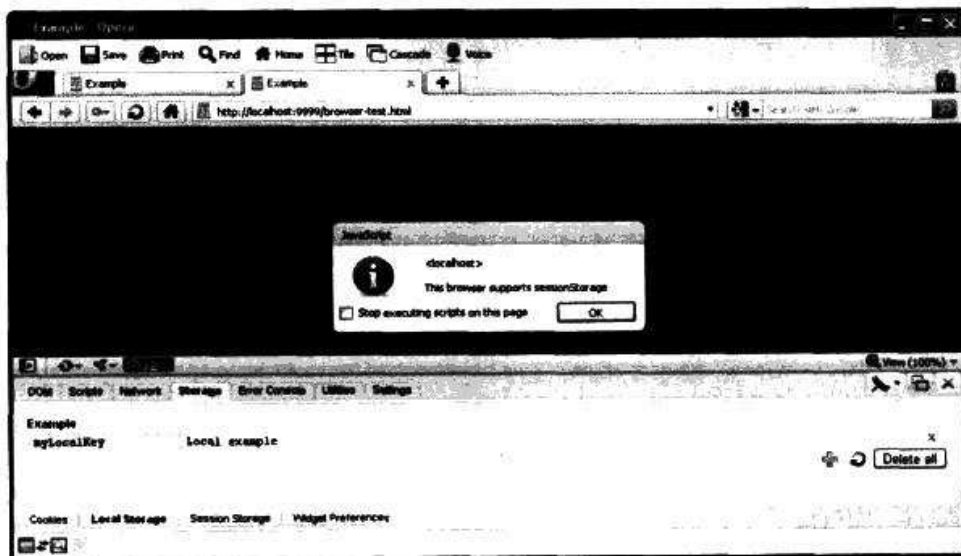


图 9-1 检查浏览器的支持性

许多浏览器不支持从文件系统直接访问文件式的 sessionStorage。所以，在运行本章示例之前，你应当确保是从 Web 服务器上获取页面。例如，可以通过下面的命令来运行位于 code/storage 文件夹中的 Python HTTP 应用服务器：

```
python -m SimpleHTTPServer 9999
```

运行上述指令后，就可以访问 <http://localhost:9999/> 上的文件了，例如 <http://localhost:9999/browser-test.html>。



**提示** 对于很多API来说，特定的浏览器可能只支持其部分功能，但是因为Web Storage API非常小，所以它已经得到了相当广泛地支持。不过出于安全考虑，即使浏览器本身支持Web Storage，用户仍然可自行选择是否将其关闭。

### 9.3.2 设置和获取数据

现在，我们将重点放在 `sessionStorage` 的功能上，你将学会如何设置和获取网页中的简单数据。设置数据值很简单，只需执行一条语句即可，下面给出了完整的语句声明：

```
window.sessionStorage.setItem('myFirstKey', 'myFirstValue');
```

在上面的存储访问语句中，需要注意三点。

- 实现 Web Storage API 的对象是 `window` 对象的子对象，因此 `window.sessionStorage` 包含了开发人员需要调用的函数。
- `setItem` 方法需要一个字符串类型的“键”和一个字符串类型的“值”来作为参数。虽然官方的 Web Storage API 支持传递非字符串数据，但是目前浏览器可能还不支持其他数据类型。
- 调用结果是将字符串 `myFirstValue` 设置到 `sessionStorage` 中，这些数据随后可以通过键 `myFirstKey` 获取。

获取数据需要调用 `getItem` 函数。例如，如果我们把下面的声明语句添加到前面的示例中：

```
alert(window.sessionStorage.getItem('myFirstKey'));
```

浏览器将创建一个 JavaScript 警告框来显示文本 `myFirstValue`。可以看出，使用 Web Storage API 设置和获取数据非常简单。

不过，访问 `Storage` 对象还有更简单的方法。你可以使用 `expando` 属性<sup>①</sup>设置数据。使用这种方法，可完全避免调用 `setItem` 和 `getItem`，而只是根据键值的配对关系，直接在 `sessionStorage` 对象上设置和获取数据。使用这种方法，我们的设置数据调用代码可以改写为：

```
window.sessionStorage.myFirstKey = 'myFirstValue';
```

同样，获取数据的代码可以改写为：

```
alert(window.sessionStorage.myFirstKey);
```

这只是 Web Storage API 的基础知识。现在，我们已经了解了如何在应用程序中使用 `sessionStorage`。但有读者可能还存在疑问，`sessionStorage` 对象有什么特别之处？毕竟，JavaScript 本来也允许开发人员设置和获取几乎任何对象的属性。其实，二者之间最大的不同在于作用域。你可能还没有意识到，在我们的示例中，设置和获取的调用不必出现在同一个网页中。只要网页是同源的（包括规则、主机和端口），基于相同的键，我们都能够在其他网页中获得设

<sup>①</sup> `expando` 属性：开发人员因开发需要而为对象额外定义的属性。——译者注



置在 `sessionStorage` 上的数据。在对同一页面后续多次加载的情况下也是如此。大部分开发者对页面重新加载时丢失脚本数据的问题已经习以为常了，但通过 Web Storage API 保存的数据不再如此了，重新加载页面后这些数据仍然还在。

### 9.3.3 封堵数据泄露

数据能够保存多久呢？对于设置到 `sessionStorage` 中的对象，只要浏览器窗口（或标签）不关闭它们就会一直存在。当用户关闭窗口或浏览器，`sessionStorage` 数据将被清除。`sessionStorage` 中的数据在某种程度上有点像便条，其中的数据不会保存很久，所以开发人员不应该把真正有价值的东西放在里面，因为不能保证不论什么时候查询这些数据都会存在。

那么，为什么还要在 Web 应用程序中使用 `sessionStorage` 呢？`sessionStorage` 非常适合用于短时存在的流程中，如对话框和向导。如果数据需要存储在多个页面中，同时又不希望用户下一次访问应用程序时重新部署，则可将这些数据存储在 `sessionStorage` 中。以前，这类数据可能需要通过表单和 cookie 提交，并在页面加载时来回传递，而使用 `Storage` 可以避免这种开销。

`sessionStorage` API 还有另外一种特殊用法，它解决了一个一直困扰着诸多 Web 应用程序的问题：数据作用域，以购买机票的购物应用程序为例。在这个应用程序中，诸如理想的出发返回日期这样的用户偏好数据，可能会在浏览器和服务器间使用 cookie 来回传送，为的是在用户使用应用时（如挑选座位和选餐），服务器应用程序能记住用户先前选择的偏好数据。

不过，用户打开多个窗口是很常见的，它们可能在查看旅游产品时同时打开多个窗口，比较不同代理商同一时间起飞的航班。这会导致 cookie 系统出现问题，因为如果一个用户在比较价格和是否有票等情况时在浏览器窗口之间来回切换，它们很可能在其中一个窗口设置 cookie 值，而在其他窗口中意外地将这些值应用到 URL 相同的另一个网页的后续操作中。这一现象也被称为数据泄露，其产生的根本原因在于 cookie 能够被同源网页共享。图 9-2 揭示了出现数据泄露的情况。

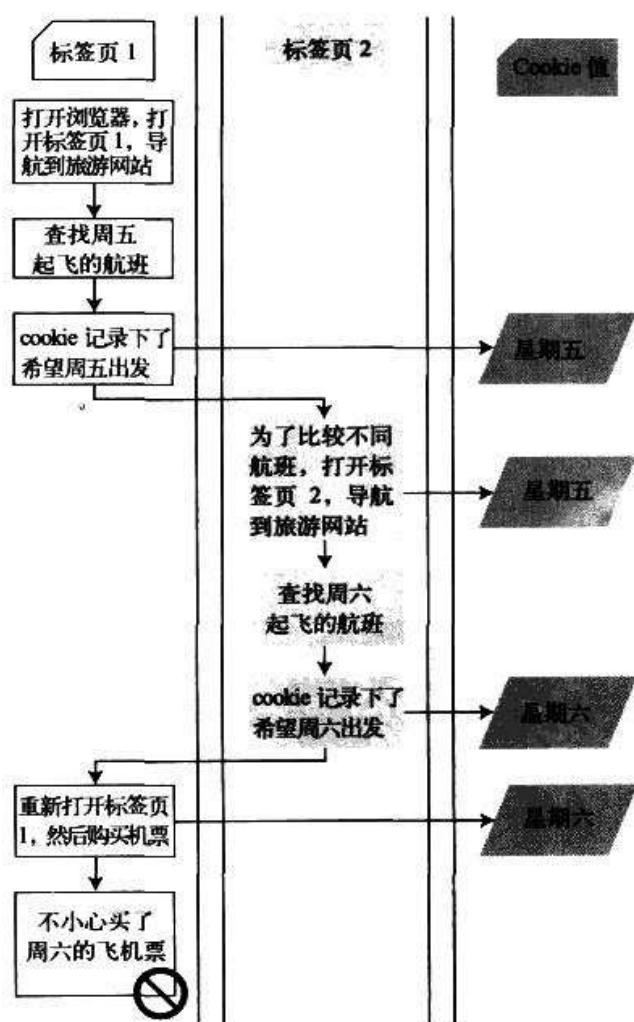


图 9-2 在旅游网站中比较机票价格时产生的信息泄露



而使用 `sessionStorage` 能够跨页面（使用该应用的页面）暂存如启程日期这样的临时数据，又不会将其泄漏到用户仍在浏览其他航班信息的窗口中。这样，不同的偏好信息就会被隔离在预订相应航班的窗口中。

### 9.3.4 localStorage 与 sessionStorage

有时候，一个应用程序会用到多个标签页或窗口中的数据，或多个视图共享的数据。在这种情况下，比较恰当的做法是使用 HTML5 Web Storage 的另一种实现方式：`localStorage`。照猫画虎，你应该已经猜到如何使用 `localStorage` 了。`localStorage` 和 `sessionStorage` 在编程上唯一的区别是访问它们的名称不同，分别是通过 `localStorage` 和 `sessionStorage` 对象来访问它们的。二者在行为上的差异主要是数据的保存时长及它们的共享方式。表 9-2 展示了二者的区别。

表9-2 `sessionStorage`和`localStorage`的区别

<code>sessionStorage</code>	<code>localStorage</code>
数据会保存到存储它的窗口或标签页关闭时	数据的生命期比窗口或浏览器的生命期长
数据只在构建它们的窗口或者标签页内可见	数据可被同源的每个窗口或者标签页共享

### 9.3.5 Web Storage API 的其他特性和函数

HTML5 Web Storage API 是 HTML5 中最简单的 API 之一。我们已经知道了如何基于 `sessionStorage` 和 `localStorage` 来显式和隐式地设置和获取数据的方法。接下来，我们将探讨该 API 的所有其他特性和函数。

在使用了 `sessionStorage` 或 `localStorage` 对象的文档中，我们可以通过 `window` 对象来获取它们。除了名字和数据的生命周期外，它们的功能完全相同且均实现了代码清单 9-2 所示的 Storage 接口。

代码清单 9-2 Storage 接口

```
interface Storage {
    readonly attribute unsigned long length;
    getter DOMString key(in unsigned long index);
    getter any getItem(in DOMString key);
    setter creator void setItem(in DOMString key, in any data);
    deleter void removeItem(in DOMString key);
    void clear();
};
```

下面详细介绍接口中的特性和函数。

- `length` 特性表示目前 `Storage` 对象中存储的键-值对的数量。请记住，`Storage` 对象是同源的，这意味着 `Storage` 对象的项数（和长度）只反映同源情况下的项数。
- `key(index)` 方法允许获取一个指定位置的键。一般而言，最有用的情况是遍历特定 `Storage`

对象的所有键。键的索引从零开始，即第一个键的索引是 0，最后一个键的索引是 index (长度-1)。获取到键后，你可以用它来获取其相应的数据。除非键本身或者在它前面的键被删除，否则其索引值会在给定 Storage 对象的生命周期内一直保留。

- getItem(key)函数是根据给定的键返回相应数据的一种方式。另一种方式是将 Storage 对象当做数组，而将键作为数组的索引<sup>①</sup>。在这两种情况下，如果 Storage 中不存在指定键，则返回 null。
- 与 getItem(key)函数类似，setItem(key, value)函数能够将数据存入指定键对应的位置，如果值已存在，则替换原值。需要注意的是设置数据可能会出错。如果用户已关闭了网站的存储，或者存储已达到其最大容量，那么此时设置数据将抛出 QUOTA\_EXCEEDED\_ERR 错误，如图 9-3 所示。因此，在需要设置数据的场合，务必保证应用程序能够处理此类异常。

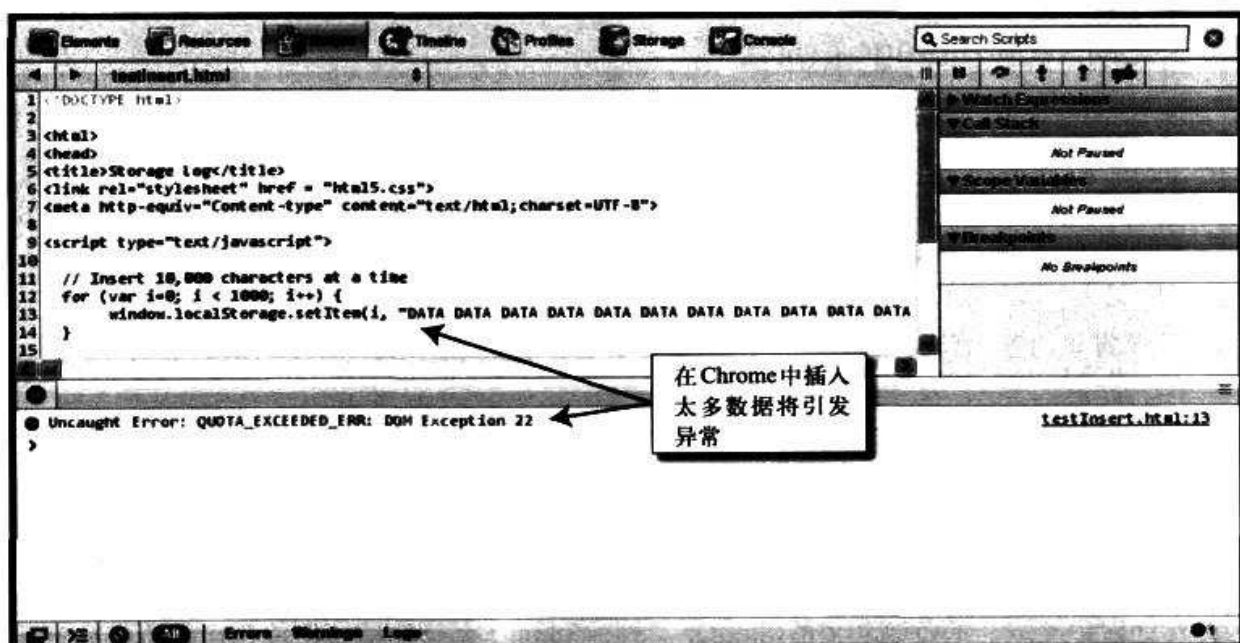


图 9-3 Chrome 抛出的 QUOTA\_EXCEEDED\_ERR 错误

- removeItem(key)函数的作用当然是删除数据项了，如果数据存储在建参数下，则调用此函数会将相应的数据项删除。如果键参数没有对应数据，则不执行任何操作。

**提示** 跟某些集和或数据框架不同，删除数据项时不会将原有数据作为结果返回。在删除操作前请确保已经存储了相应数据的副本。

<sup>①</sup> 由于 JavaScript 对象是关联数组，所以可以将对象当做数组来访问。当然，将键作为 Storage 对象的属性来访问也没有问题。——编者注





- 最后是 `clear()` 函数，它能删除存储列表中的所有数据。空的 `Storage` 对象调用 `clear()` 方法也是安全的，此时的调用不执行任何操作。

#### 磁盘空间配额

“HTML5 规范中建议浏览器允许每组同源页面使用 5MB 的空间。当达到空间配额时，浏览器应提示用户分配更多空间，同时还应提供查看每组同源页面已用空间的方法。

实际的实现方式多少有些出入。有些浏览器在不告知用户的前提下，允许页面使用更多的空间，有些浏览器则向用户提示已经扩展了空间，而另外一些则只是抛出 `QUOTA_EXCEEDED_ERR` 错误，如图 9-3 所示。图 9-3 所示示例的测试文件 `testQuota.html` 位于 `code/storage` 文件夹中。”

——Peter

### 9.3.6 更新 Web Storage 后的通信

某些复杂情况下，多个网页、标签页或者 `Worker` 都需要访问存储的数据。此时，应用程序可能会在存储数据被修改后触发一系列操作。对于这种情况，HTML5 Web Storage API 内建了一套事件通知机制，它可以将数据更新通知发送给感兴趣的监听者。无论监听窗口本身是否存储过数据，与执行存储操作的窗口同源的每个窗口的 `window` 对象上都会触发 `Web Storage` 事件。

**提示** 同源窗口间可以使用 `Web Storage` 事件进行通信，9.6 节将对此做更加深入的探讨。

像下面这样，添加事件监听器即可接收同源窗口的 `Storage` 事件：

```
window.addEventListener("storage", displayStorageEvent, true);
```

代码中事件类型参数是 `storage`，表明我们感兴趣的是 `Storage` 事件。这样一来，只要有同源的 `Storage` 事件发生（包括 `SessionStorage` 和 `LocalStorage` 触发的事件），已注册的所有事件侦听器作为事件处理程序就会接收到相应的 `Storage` 事件。`Storage` 事件的接口形式如代码清单 9-3 所示。

#### 代码清单 9-3 StorageEvent 接口

```
interface StorageEvent : Event {
    readonly attribute DOMString key;
    readonly attribute any oldValue;
    readonly attribute any newValue;
    readonly attribute DOMString url;
    readonly attribute Storage storageArea;
};
```

`StorageEvent` 对象是传入事件处理程序的第一个对象，它包含了与存储变化有关的所有必要信息。

- `key` 属性包含了存储中被更新或删除的键。

- oldValue 属性包含了更新前键对应的数据，newValue 属性包含更新后的数据。如果是新添加的数据，则 oldValue 属性值为 null。如果是被删除的数据，则 newValue 属性值为 null。
- url 属性指向 Storage 事件发生的源。
- storageArea 属性是一个引用，它指向值发生改变的 localStorage 或 sessionStorage。如此一来，处理程序就可以方便地查询到 Storage 中的当前值，或者基于其他 Storage 的改变而执行其他操作。

代码清单 9-4 是一个简单的事件处理程序，它以警告框的形式来显示在当前页面上触发的 Storage 事件的详细信息。

代码清单 9-4 显示 Storage 事件内容的事件处理程序

```
// 显示 Storage 事件的详细信息
function displayStorageEvent(e) {
    var logged = "key:" + e.key + ", newValue:" + e.newValue + ", oldValue:" +
        e.oldValue + ", url:" + e.url + ", storageArea:" + e.storageArea;

    alert(logged);
}

// 添加 Storage 事件监听器
window.addEventListener("storage", displayStorageEvent, true);
```

### 9.3.7 探索 Web Storage

由于 HTML5 Web Storage 在功能上同 cookie 非常相似，所以最新浏览器在二者处理方式上的雷同也就不足为奇了。存储在 localStorage 或 sessionStorage 中的数据在最新的浏览器中可以像 cookie 一样浏览，如图 9-4 所示。

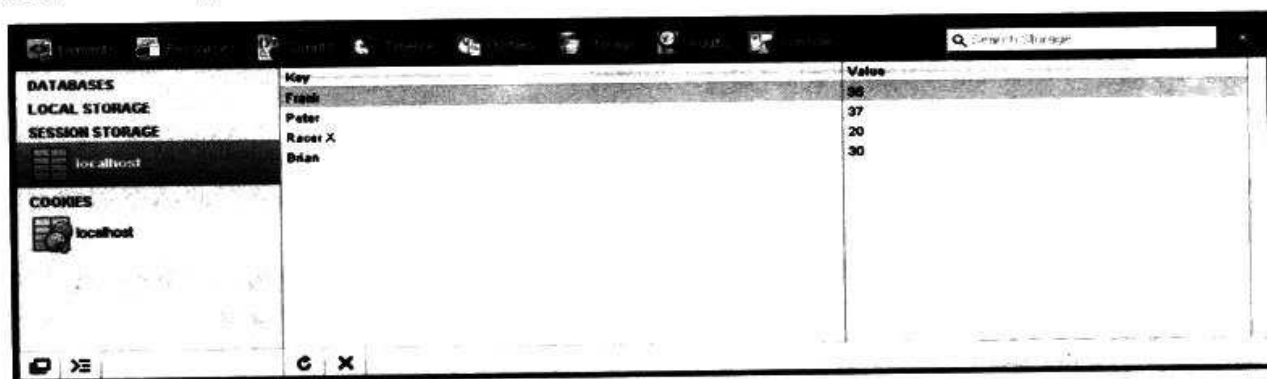


图 9-4 谷歌 Chrome 浏览器 Storage 面板中的存储数据

Chrome 提供的界面允许用户根据需要删除存储数据，而在访问网页时，用户可以轻易地浏览到网站记录了哪些数据。苹果公司的 Safari 浏览器和 Chrome 一样，都是基于 WebKit 渲染引擎，它也像 Chrome 那样统一显示 cookie 和 Storage。图 9-5 显示了 Safari 存储面板。



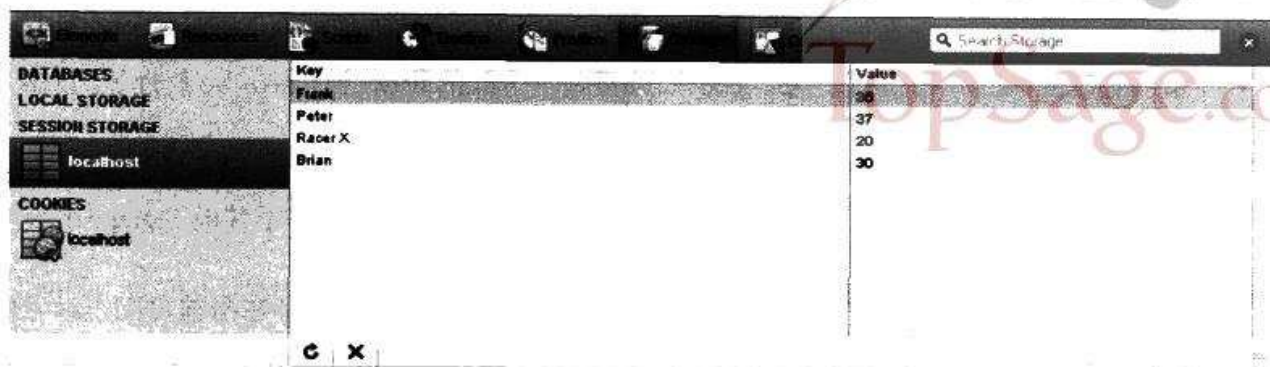


图 9-5 Safari 的 Storage 面板中的存储数据

Opera 浏览器则更进一步，它不仅允许用户浏览和删除存储数据，还允许创建数据，如图 9-6 所示。



图 9-6 Opera 浏览器 Storage 面板中的存储数据

随着众多浏览器开发者的努力，Web Storage 的应用范围变得日益广阔。可以预见 Web Storage 提供给用户和开发人员的特性和工具也会越来越多，越来越强大。

## 9.4 构建 HTML5 Web Storage 应用

现在，让我们将前面所学知识应用到 Web 应用程序中。随着应用变得越来越复杂，无需服务器交互而管理尽可能多的数据变得越来越重要。将数据存储在本地的客户端，进而从本地而不是远程获取数据，既可降低网络流量，又可提升浏览器响应能力。

一个困扰开发人员的常见问题是，当用户从应用程序的一个页面切换到另一个页面时如何管理数据。传统实现方式是由服务器存储数据，当用户在网页间切换时来回传递数据。还有一种做法是应用程序尽可能地让用户停留在一个动态更新的网页上。不过，用户更习惯于在页面间切换，当用户返回到应用程序的某个页面时，如果能够快速获取数据并加以显示，对于增强用户体验来说无疑是非常好的方式。

在示例应用程序中，我们将演示用户在一个网站的页面间切换时，如何将应用程序的临时数据存储在本地的，以及如何从每个页面的 Storage 中快速加载。我们将对前面章节中的示例进行改造。在第 4 章中，我们发现收集用户的当前位置很容易。随后，在第 6 章中，我们演示了如何获



取位置数据并将其发送到远程服务器，以供所有感兴趣的用户浏览。本节，我们将更进一步：监听从 WebSocket 广播的位置数据并将其存储在本地 Storage 中，以便用户在页面间切换时可以立即获取应用程序所需的数据。

试想，我们的跑步俱乐部程序可以获取参赛选手的实时位置信息，这些信息由选手的移动设备广播出来，通过 WebSocket 服务器实现共享。由于参赛选手在比赛中不断上传新的位置信息，所以对于 Web 应用程序来说，要实时显示每个参赛选手的位置还是比较简单的。优秀的网站会将参赛选手的位置信息存入缓存，当用户在网站的页面间切换时，可以快速显示位置信息。这正是我们接下来要构建的。

为了达到这一目的，我们需要引进一个示例网站，它应该可以保存和恢复参赛选手数据。我们已经做好了包含 3 个页面的跑步比赛网站，并将其放在了在线资源的 code/storage 文件夹下。当然，你可以选择任意网站来做演示。关键是网站要包含多个页面，且能让用户在它们之间轻松切换。我们会在这些网页中插入少量动态内容来表示实时排行榜，排行榜的内容实际上就是一张所有参赛选手距离终点还有多远的距离列表。图 9-7 显示了跑步网站包含的三个页面。

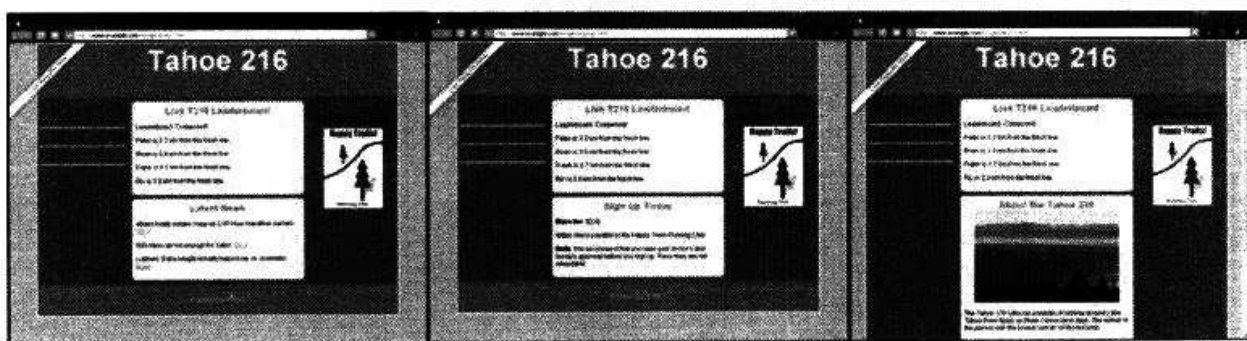


图 9-7 跑步网站示例

三个页面的共同点是都包含了一张排行榜。排行榜中的每项都会显示参赛选手的名字以及他到终点的距离。加载任一页面时，都会为排行榜建立一个到比赛广播服务器的 WebSocket 连接，监听参赛选手的位置信息。反过来，参赛选手将向同一个广播服务器发送他们目前的位置信息，并最终将位置信息实时传递到页面。

所有这些都与前面章节中提到的 Geolocation 和 WebSockets 有关。事实上，这里大部分的演示代码是本书前面的示例代码。不过，这个示例与前面的示例有一个关键的不同点：当数据到达页面后，我们会将其存储在 sessionStorage 中，以供后期获取。然后，无论何时用户转向一个新页面，在新的 WebSocket 连接创建之前，存储的数据都将被获取和显示。这样一来，在页面之间传输临时数据就不必依赖 cookie 或 Web 服务器了。

正如我们在第 6 章中所做的那样，我们将参赛选手的位置信息以容易读取和解析的格式通过 Web 传送。参赛选手的位置信息是一个字符串，使用分号 (;) 作为数据块的分隔符，将姓名、纬度和经度分隔开。例如，参赛选手 Racer X 位于纬度 37.20，经度 -121.53，可以使用下面的字符串：



```
;Racer X;37.20;-121.53
```

现在，我们来分析代码。每个网页都包含功能相同的 JavaScript 代码，具体功能包括连接到 WebSocket 服务器、处理和显示排行榜信息以及使用 sessionStorage 保存和恢复排行榜。因此，在构建真实的应用程序时，JavaScript 库中可以考虑包含这些代码。

首先，我们将构建一些通用函数。要计算任意参赛选手当前位置与终点间的距离，我们需要代码清单 9-5 的程序。

#### 代码清单 9-5 距离计算程序

```
//函数通过计算两个地理位置的经纬度来计算二者之间的距离
function toRadians(num) {
    return num * Math.PI / 180;
}
function distance(latitude1, longitude1, latitude2, longitude2) {
    // R 是以 km 为单位的地球半径
    var R = 6371;

    var deltaLatitude = toRadians((latitude2-latitude1));
    var deltaLongitude = toRadians((longitude2-longitude1));
    latitude1 = toRadians(latitude1), latitude2 = toRadians(latitude2);

    var a = Math.sin(deltaLatitude/2) *
        Math.sin(deltaLatitude/2) +
        Math.cos(latitude1) *
        Math.cos(latitude2) *
        Math.sin(deltaLongitude/2) *
        Math.sin(deltaLongitude/2);

    var c = 2 * Math.atan2(Math.sqrt(a),
        Math.sqrt(1-a));
    var d = R * c;
    return d;
}

// 终点的经纬度
var finishLat = 39.17222;
var finishLong = -120.13778;
```

这些函数我们已经非常熟悉，早在第 6 章就用过了，distance 函数用来计算两点之间的距离。函数的细节和它在比赛中能否最准确地表示距离都不重要，重要的是它可以用在我们的示例中以演示效果。

在终点线，我们确定了比赛终点位置的经度和纬度。从代码中可以看到，我们将其与输入的参赛选手位置坐标进行比较，以确定参赛选手到终点线的距离，从而得到他们在这场比赛中的排名顺序。

现在，让我们来看看用于显示网页的 HTML 代码片段。

```
<h2>Live T216 Leaderboard</h2>
<p id="leaderboardStatus">Leaderboard: Connecting...</p>
<div id="leaderboard"></div>
```

虽然 HTML 网页中大部分内容与我们的阐述目的无关，但在这几行中，我们声明了 ID 为 `leaderboardStatus` 和 `leaderboard` 的命名元素。`leaderboardStatus` 命名元素用于显示 WebSocket 的连接信息。我们将在 `leaderboard` 命名元素中插入表示位置信息的 `div` 元素。调用代码清单 9-6 所示的通用函数，可以从 WebSocket 消息获得这些位置信息。

#### 代码清单 9-6 位置信息实用函数

```
// 在页面中显示参赛选手的名字以及他到终点的距离
function displayRacerLocation(name, distance) {
    // 寻找 ID 为 name (参赛选手名字) 的 HTML 元素, 如果不存在, 则创建之
    var incomingRow = document.getElementById(name);
    if (!incomingRow) {
        incomingRow = document.createElement('div');
        incomingRow.setAttribute('id', name);
        incomingRow.userText = name;

        document.getElementById("leaderboard").appendChild(incomingRow);
    }

    incomingRow.innerHTML = incomingRow.userText + " is " +
        Math.round(distance*10000)/10000 + " km from the finish line";
}
```

这是一个简单的显示函数，它接受参赛选手的名字及其到终点的距离。图 9-8 是 `index.html` 网页上排行榜的展示效果。

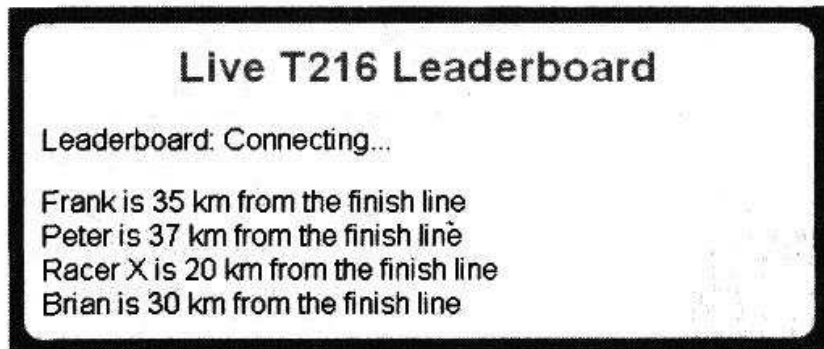


图 9-8 跑步比赛排行榜

参赛选手的名字有两个用途：它不仅显示在参赛选手的状态信息中，还用来引用存储参赛选手状态信息的唯一 `div` 元素。如果与参赛选手相对应的 `div` 元素存在，那么我们可以使用标准语句 `document.getElementById()` 来查找到它。如果与参赛选手相对应的 `div` 元素不存在，我们将创建相应的 `div` 元素并将其插入到 `leaderboard` 命名元素中。无论哪种情况，我们随后都会根据参赛选手到终点线的最新距离更新 `div` 元素，实现它在页面中的实时更新。如果你已经阅读过第 6 章，那么对我们在这里构建的示例程序应该已经很熟悉了。



我们的下一个函数是消息处理器，它将在 WebSocket 服务器返回数据时被调用，如代码清单 9-7 所示。

#### 代码清单 9-7 WebSocket 消息处理函数

```
// 当从 WebSocket 获得新的位置数据时进行回调
function dataReturned(locationData) {
    // 将数据解析成 ID、经度和纬度
    var allData = locationData.split(";");
    var incomingId = allData[1];
    var incomingLat = allData[2];
    var incomingLong = allData[3];
    // 使用新数据更新每行的显示
    var currentDistance = distance(incomingLat, incomingLong, finishLat, finishLong);

    // 在 Storage 中存储新的参赛选手名字和他到终点的距离
    window.sessionStorage[incomingId] = currentDistance;

    // 在页面中显示更新后的信息
    displayRacerLocation(incomingId, currentDistance);
}
```

这个函数的参数是一个前面介绍过的格式化字符串，它将参赛选手的名字、经度和纬度以分号分隔。我们的第一步是使用 JavaScript 的 `split()` 函数将它分成各个部分，以便对 `incomingId`、`incomingLat`、`incomingLong` 单独处理。

然后，它将参赛选手的经度和纬度、终点线的经度和纬度信息传递给之前定义的 `distance()` 函数，并将计算得到的距离存储在 `currentDistance` 变量中。

现在，我们有了一些需要存储的数据，来看一下如何实现 HTML Web Storage 的调用。

```
// 在 storage 中存储新的参赛选手名字和他到终点的距离
window.sessionStorage[incomingId] = currentDistance;
```

上面这行代码中，我们使用 `sessionStorage` 对象来存储参赛选手到终点线的当前距离，其对应的键是参赛选手的名字（或 ID）。换句话说，我们在 `sessionStorage` 中设置了一组数据，数据以参赛选手的名字作为键，以参赛选手到终点的距离作为值。我们很快就会用上这些保存在 `Storage` 中的数据，当用户在网站的页面间切换时，数据会从 `Storage` 中恢复。在函数的末尾，我们调用了已经定义过的 `displayLocation()` 函数，以确保最新的位置能够显示在当前页面中。

现在，我们来看看 `Storage` 示例中的最后一个函数——用户任何时候访问页面都会触发的加载函数，如代码清单 9-8 所示。

#### 代码清单 9-8 初始化页面加载程序

```
// 页面加载时，为赛事广播服务器建立一个 socket 连接
function loadDemo() {
    // 确定浏览器是否支持 sessionStorage
    if (typeof(window.sessionStorage) === "undefined") {
        document.getElementById("leaderboardStatus").innerHTML = "Your browser does not support HTML5 Web Storage";
    }
    return;
}
```

```

    }

    var storage = window.sessionStorage;

    // 遍历 Storage 数据库中的所有“键”，在页面中显示每名参赛选手的位置信息
    for (var i=0; i < storage.length; i++) {
        var currRacer = storage.key(i);
        displayRacerLocation(currRacer, storage[currRacer]);
    }
    // 测试浏览器是否支持 WebSockets
    if (window.WebSocket) {
        // WebSocket 广播服务器的位置
        url = "ws://websockets.org:7999/broadcast";
        socket = new WebSocket(url);
        socket.onopen = function() {
            document.getElementById("leaderboardStatus").innerHTML = "Leaderboard:

                Connected!";
        }
        socket.onmessage = function(e) {
            dataReturned(e.data);
        }
    }
}

```

代码清单 9-8 所示的函数相对较长，以后还会遇到很多这么长的代码。让我们逐步分析 loadDemo() 函数。如代码清单 9-9 所示，我们首先进行基本的错误检查，通过检测浏览器页面的 window 对象中是否存在 sessionStorage 对象，来判断浏览器是否支持 Web Storage。如果无法访问 sessionStorage，则更新 leaderboardStatus 来提示用户，并提前结束加载程序。本示例中，当浏览器不支持 Storage 时，我们不提供备选方案。

#### 代码清单 9-9 检查浏览器是否支持 sessionStorage

```

// 确定浏览器是否支持 sessionStorage
if (typeof(window.sessionStorage) === "undefined") {
    document.getElementById("leaderboardStatus").innerHTML = "Your browser does
        not support HTML5 Web Storage";
    return;
}

```

**提示** 在实际项目中，如果浏览器不支持 Storage，很可能要重写程序，以便在页面切换时放弃保存持久化数据，并在页面加载时显示空白排行榜。不过，这里就不需要了，因为我们的目标只是演示 Storage 如何优化用户的体验和减轻网络负担。

页面加载时，要做的下一件事情是获取保存在 Storage 中的所有参赛选手到终点的距离结果，这些结果已经在网站的页面上出现过了。回想一下，网站中的每个页面都运行了一段相同的脚本，从而使用户浏览不同页面时都能看到排行榜。这时，其他页面上的排行榜可能已经向 Storage 中写入了数据，所以，调用代码清单 9-10 中的代码，在页面加载时直接获取数据并加以显示即可。



只要用户不关闭窗口、标签页或浏览器，不清除 `sessionStorage`，保存过的数据就会在用户浏览时一直有效。

#### 代码清单 9-10 显示保存的参赛选手数据

```
var storage = window.sessionStorage;

// 遍历 Storage 数据库中的所有键，在页面中显示每名参赛选手的位置信息
for (var i=0; i < storage.length; i++) {
    var currRacer = storage.key(i);
    displayRacerLocation(currRacer, storage[currRacer]);
}
```

上面的代码是整个示例应用中的核心部分。我们首先查询了 `sessionStorage` 的长度，也就是 `Storage` 中有多少个键。然后，调用 `storage.key()` 获取每一个键，并将其存储在 `currRacer` 变量中。最后，调用 `storage[currRacer]` 方法，传入 `currRacer` 变量，获取当前键所对应的值。键及其对应的值分别代表参赛选手的名字和他到终点的距离，这些信息都是在访问之前的页面时存储的。

一旦获取了已经储存的参赛选手姓名和距离，我们就可以使用 `displayRacerLocation()` 函数将它们显示出来。页面加载时，一切都发生在电光火石间，网页会在瞬间用先前传递过来的数据填充排行榜。

---

**提示** 我们的示例应用程序只依赖于将数据存储到 `sessionStorage` 的应用。如果应用程序需要与其他数据共享 `Storage` 对象，那么你需要使用更加细致的键存储策略，而不再是将键简单地存放在根级别下。我们将在 9.6 节中了解另一种存储策略。

---

最后一块工作是使用 `WebSocket` 将页面连接到跑步比赛广播服务器上，如代码清单 9-11 所示。

#### 代码清单 9-11 连接到 WebSocket 广播服务

```
// 测试浏览器是否支持 WebSocket
if (window.WebSocket) {

    // WebSocket 广播服务器的位置
    url = "ws://websites.org:7999/broadcast";
    socket = new WebSocket(url);
    socket.onopen = function() {
        document.getElementById("leaderboardStatus").innerHTML = "Leaderboard:↔  
Connected!";
    }
    socket.onmessage = function(e) {
        dataReturned(e.data);
    }
}
```

正如我们在第 6 章中所做的那样，首先检查 `window.WebSocket` 对象是否存在，以确定浏览器

是否支持 WebSockets。一旦确认，我们就连接到运行着 WebSocket 服务器的 URL。服务器会广播以分号分隔格式组织的参赛选手位置信息。在 socket.onmessage 回调函数接收到消息时，我们会调用先前讨论的 dataReturned() 函数来处理和显示它。此外，我们还使用了 socket.onopen 方法，以将简单的诊断信息更新到 leaderboardStatus 命名元素中，来表明成功打开了套接字。

下面该轮到页面加载了。脚本中最后声明的一块代码用于注册事件监听器，这样一来，网页加载成功后就会自动调用 loadDemo() 函数：

```
// 在页面加载和卸载时添加监听器
window.addEventListener("load", loadDemo, true);
```

前面提到多次了，只有当页面完全加载后，才能通过事件监听器调用 loadDemo() 函数。

不过，要怎么做才能将参赛选手的数据从比赛路线传输到 WebSocket 广播服务器，继而传入我们的页面呢？这就要用到第 6 章所演示的跟踪器示例了。很简单，只需要将连接 URL 指向前面列出的广播服务器即可。毕竟，字符串格式都是一样的。代码清单 9-12 是我们构建的一个非常简单的参赛选手广播来源页面，能够达到类似目的。理论上，这个页面可以运行在参赛选手的移动设备上。尽管文件本身不包括任何 HTML5 Web Storage 代码，但在支持 WebSockets 和 Geolocation 的浏览器上，它却是一条传输格式化数据的便捷方式。racerBroadcast.html 文件可以从本书的示例源代码文件中找到。

#### 代码清单 9-12 racerBroadcast.html 文件内容

```
<!DOCTYPE html>

<html>

<head>
<title>Racer Broadcast</title>
<link rel="stylesheet" href="styles.css">
</head>

<body onload="loadDemo()">

<h1>Racer Broadcast</h1>

Racer name: <input type="text" id="racerName" value="Racer X"/>
<button onclick="startSendingLocation()">Start</button>

<div><strong>Geolocation</strong>: <p id="geoStatus">HTML5 Geolocation not
started.</p></div>
<div><strong>WebSocket</strong>: <p id="socketStatus">HTML5 Web Sockets are
<strong>not</strong> supported in your browser.</p></div>

<script type="text/javascript">
// 引用 Web Socket
var socket;

var lastLocation;
```



```
function updateSocketStatus(message) {
    document.getElementById("socketStatus").innerHTML = message;
}

function updateGeolocationStatus(message) {
    document.getElementById("geoStatus").innerHTML = message;
}

function handleLocationError(error) {
    switch(error.code)
    {
        case 0:
            updateGeolocationStatus("There was an error while retrieving your location: " +
                error.message);
            break;
        case 1:
            updateGeolocationStatus("The user prevented this page from retrieving a
                location.");
            break;
        case 2:
            updateGeolocationStatus("The browser was unable to determine your location: " +
                error.message);
            break;
        case 3:
            updateGeolocationStatus("The browser timed out before retrieving the location.");
            break;
    }
}

function loadDemo() {
    // 测试是否支持 WebSockets
    if (window.WebSocket) {

        // WebSocket 广播服务器的位置
        url = "ws://websites.org:7999/broadcast";
        socket = new WebSocket(url);
        socket.onopen = function() {
            updateSocketStatus("Connected to WebSocket race broadcast server");
        }
    }
}

function updateLocation(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    var timestamp = position.timestamp;
    updateGeolocationStatus("Location updated at " + timestamp);

    // 处理消息以便使用 WebSocket 将其发送
    var toSend = ";" + document.getElementById("racerName").value
        + ";" + latitude + ";" + longitude;
    setTimeout("sendMyLocation('" + toSend + "')", 1000);
}
```

```

function sendMyLocation(newLocation) {
    if (socket) {
        socket.send(newLocation);
        updateSocketStatus("Sent: " + newLocation);
    }
}

function startSendingLocation() {
    var geolocation;
    if(navigator.geolocation) {
        geolocation = navigator.geolocation;
        updateGeolocationStatus("HTML5 Geolocation is supported in your browser.");
    }
    else {
        geolocation = google.gears.factory.create('beta.geolocation');
        updateGeolocationStatus("Geolocation is supported via Google Gears");
    }

    // 使用 Geolocation API 注册位置更新
    geolocation.watchPosition(updateLocation,
                              handleLocationError,
                              {maximumAge:20000});
}
</script>
</body>
</html>

```

它与第 6 章的跟踪器示例几乎一模一样，不再赘述。主要的区别在于，这个文件包含了参赛选手名字的文本输入框：

```
Racer name: <input type="text" id="racerName" value="Racer X"/>
```

参赛选手的名字会作为数据字符串的一部分发送到广播服务器：

```
var toSend = ";" + document.getElementById("racerName").value
            + ";" + latitude + ";" + longitude;
```

如果要测试的话，可以在像 Google Chrome 这样支持 HTML5 Web Storage、Geolocation 和 WebSocket 的浏览器中打开两个窗口。首先，在第一个窗口中加载跑步俱乐部的 index.html 页面。你会看到它使用 WebSocket 连接到了赛事广播网站，然后等待参赛选手数据通知。其次，在第二个窗口中打开 racerBroadcast.html 文件。同样，页面打开后也会连接到 WebSocket 广播网站。输入参赛选手的名字，然后点击“Start”按钮。你会看到 WebSocket 服务器将这名参赛选手的位置信息发送出去，然后它会在浏览器另一个窗口的排行榜中出现。显示效果如图 9-9 所示。

可以通过页面左侧的“Signup”和“About the Race”链接导航到跑步俱乐部的其他页面。由于这些页面都加载了我们的脚本，它们会立即使用已有的参赛选手数据加载和填充页面中的排行榜。（广播页面）会发出更多的参赛选手状态通知，这样一来，不管我们切换到网站的哪个页面，都能看到这些信息。





图 9-9 跑步页面和 racerBroadcast.html 页面

现在，代码已经全部完成了，让我们回顾一下都做了些什么。我们构建了一个适合包含在共享 JavaScript 库中的简单功能块，它连接到一个 WebSocket 广播服务器并监听参赛选手数据更新信息。收到更新信息后，脚本会在页面中显示位置数据，并将其使用 HTML5 sessionStorage 存储起来。当页面加载时，它会检查所有已经存储的参赛选手数据，从而保证用户在浏览网站时所有状态都是正确的。这么做有哪些优势呢？

- 减少网络流量：赛事信息存储在本地的浏览器中。一旦获取到赛事信息，每个页面加载时都会从本地获取数据，而不是使用 cookie 或服务器请求获取。
- 即时显示数据：因为页面的动态部分（当前的排行榜状态）是本地数据，所以浏览器网页本身可以缓存而不是从网络上加载。无需耗费任何网络加载所需的时间就可迅速显示这些数据。
- 临时存储：数据在比赛结束后就没用了。因此，我们把它存在了 sessionStorage 中，这意味着窗口或标签页关闭后它就会被丢弃，不再占用任何空间。

### 无懈可击

“我们在示例中只用了寥寥几行脚本代码就完成了很多事。但是，不要以为在现实中开发可供公开访问的网站也会这么简单。在实际应用中直接使用这里的示例代码是不现实的。

例如，我们的信息格式不支持重名的参赛选手名字，所以最好每个参赛选手有一个唯一的标识符。我们计算到终点的距离是‘直线距离’，不能用于真正的越野赛事。更本地化、更多的错误检查以及更注重细节会让你的网站适用于所有参赛选手。这也算是个免责声明吧。”

——Brian

示例中用到的技术可以应用到各种类型的数据上，像聊天、电子邮件、体育计分都是很好的示例，都可以像我们所演示的那样，使用 localStorage 或 sessionStorage 在页面间缓存并显示信息。如果应用程序需要定期在浏览器和服务器间发送特定的用户数据，不妨考虑使用 HTML5 Web Storage 来简化流程。

## 9.5 浏览器数据库存储展望

键-值对形式的 Storage API 在数据持久化方面已经很强大了，但是如果为 Storage 建立查询索引又会如何呢？HTML5 的应用同样可以访问索引数据库。数据库 API 的具体细节仍在完善，并有多个方案。Web SQL Database 是其中之一，并已经在 Safari、Chrome 和 Opera 中实现了。表 9-3 显示了浏览器对于 Web SQL Database 的支持情况。

表9-3 浏览器对于Web SQL Database的支持情况

浏览器	细节
Chrome	3.0及以上的版本均支持
Firefox	不支持
Internet Explorer	不支持
Opera	10.5及以上的版本均支持
Safari	3.2及以上的版本均支持

Web SQL Database 允许应用程序通过一个异步 JavaScript 接口访问 SQLite 数据库。虽然它既不是常见 Web 平台的一部分，也不是 HTML5 规范最终推荐的数据库 API，但当针对如 Safari 移动版这样的特定平台时，SQL API 会很有用。在任何情况下，SQL API 在浏览器中的数据库处理能力都是无可比拟的。跟其他 Storage API 一样，浏览器能够限制同源页面可用 Storage 的大小，并且当用户数据被清除时，Storage 中的数据也会被清除。

### Web SQL Database 的命运

“虽然 Web SQL Database 已经在 Safari、Chrome 和 Opera 中实现，但是 Firefox 中并没有实现它，而且它在 WHATWG 维基中也被列为‘停滞’状态。HTML5 规范中定义了一个执行 SQL 命令的 API，接收字符串形式的输入，遵从 SQLite 对 SQL 语句的解析规范。由于标准认定直接执行 SQL 语句不可取，Web SQL Database 已被较新的规范——索引数据库（Indexed Database，原为 WebSimpleDB）所取代。索引数据库更简便，而且也不依赖于特定的 SQL 数据库版本。目前浏览器正在逐步实现对索引数据库的支持。”

——Frank

因为 Web SQL Database 事实上已被各大浏览器厂商支持，所以我们将介绍一个基本示例，其中省略了对 Web SQL Database API 细节的完整说明。示例演示了 Web SQL Database API 的基本用法。它打开名为 mydb 的数据库，创建 racers 表（如果表不存在的话），向其中填充预定义的



名字。图 9-10 是在 Safari 的 Web Inspector 面板中显示的带有 racers 表的数据库。

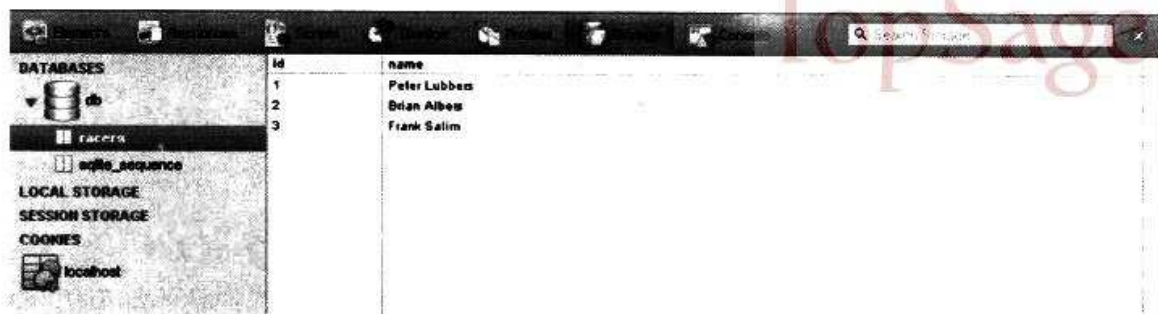


图 9-10 Safari 的 Web Inspector 面板中显示的包含 racers 表的数据库

我们通过数据库名打开相应数据库。与数据库交互时，`window.openDatabase()` 函数会返回 Database 对象。`openDatabase()` 函数将数据库名作为必选参数，将版本说明作为可选参数。打开数据库后，应用程序代码就可以对其进行读写操作了。`transaction.executeSql()` 函数用来在事务上下文中执行 SQL 语句。在这个简单示例中，我们将使用 `executeSQL()` 函数创建一张数据表，并向其中插入参赛选手的名，随后通过查询数据库的方式获取数据，进而创建一张用于显示的 HTML 表格。图 9-11 显示了输出的 HTML 文件，其中包含从数据表中检索出来的姓名列表。

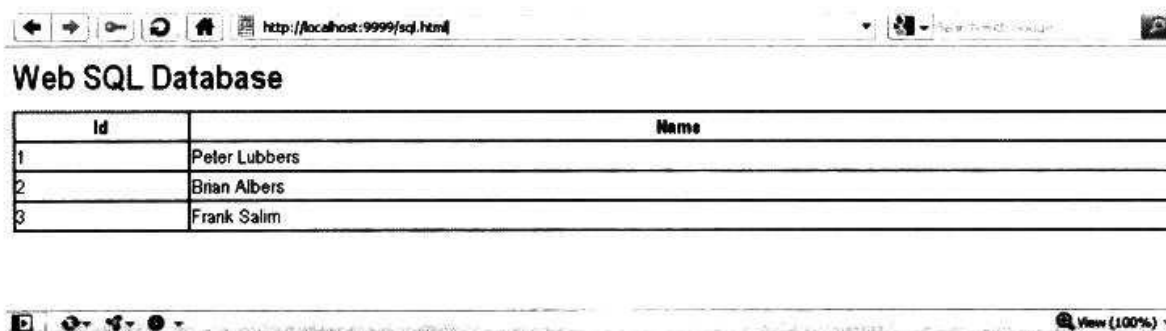


图 9-11 sql.html 显示了 SQL 语句“SELECT \* FROM racers;”的执行结果

数据库操作可能需要花点时间才能完成。不过，在获得查询结果集之前，查询操作会在后台运行，以避免阻塞脚本的执行。`executeSQL()` 的第三个参数是回调函数，查询得到的事务和结果集将作为参数供此回调函数使用。

代码清单 9-13 是存放在 `code/storage` 文件夹下的 `sql.html` 文件的完整代码。

#### 代码清单 9-13 使用 Web SQL Database API

```
<!DOCTYPE html>
<title>Web SQL Database</title>
<script>

// 通过名字打开数据
var db = openDatabase('db', '1.0', 'my first database', 2 * 1024 * 1024);
```

```

function log(id, name) {
    var row = document.createElement("tr");
    var idCell = document.createElement("td");
    var nameCell = document.createElement("td");
    idCell.textContent = id;
    nameCell.textContent = name;
    row.appendChild(idCell);
    row.appendChild(nameCell);

    document.getElementById("racers").appendChild(row);
}

function doQuery() {
    db.transaction(function (tx) {
        tx.executeSql('SELECT * from racers', [], function(tx, result) {
            // 记录 SQL 结果集
            for (var i=0; i<result.rows.length; i++) {
                var item = result.rows.item(i);
                log(item.id, item.name);
            }
        });
    });
}

function initDatabase() {
    var names = ["Peter Lubbers", "Brian Albers", "Frank Salim"];

    db.transaction(function (tx) {
        tx.executeSql('CREATE TABLE IF NOT EXISTS racers (id integer primary key ←
autoincrement, name)');

        for (var i=0; i<names.length; i++) {
            tx.executeSql('INSERT INTO racers (name) VALUES (?)', [names[i]]);
        }

        doQuery();
    });
}

initDatabase();
</script>
<h1>Web SQL Database</h1>
<table id="racers" border="1" cellspacing="0" style="width:100%">
  <th>Id</th>
  <th>Name</th>
</table>

```

## 9.6 进阶功能

尽管有些技巧我们在示例中用不上，但并不妨碍它们在多种类型的 HTML5 Web 应用中发挥



作用。本节，我们就来介绍一些简单实用的进阶功能。

### 9.6.1 JSON 对象的存储

虽然 HTML5 Web Storage 规范允许将任意类型的对象保存为键-值对形式，实际情况却是一些浏览器将数据限定为文本字符串类型。不过，既然现代浏览器原生支持 JSON (JavaScript Object Notation)，我们额外有了一种解决这个问题的可行方案。

JSON 是一种将对象与字符串可以相互表示的数据转换标准。十余年来，JSON 一直是通过 HTTP 将对象从浏览器传送到服务器一种常用格式。现在，我们可以通过序列化复杂对象将 JSON 数据保存在 Storage 中，以实现复杂数据类型的持久化。脚本如代码清单 9-14 所示。

代码清单 9-14 存储 JSON 对象

```
<script>
  var data;

  function loadData() {
    data = JSON.parse(sessionStorage["myStorageKey"])
  }

  function saveData() {
    sessionStorage["myStorageKey"] = JSON.stringify(data);
  }

  window.addEventListener("load", loadData, true);
  window.addEventListener("unload", saveData, true);
</script>
```

上面的脚本中包含了用于绑定页面加载事件和卸载事件的事件监听器。在这里，处理程序会分别调用 loadData() 和 saveData() 函数。

在 loadData() 函数中，首先查询 sessionStorage，获取与键相对应的值，并将这个值传给 JSON.parse() 函数。作为 JSON.parse() 函数参数的值实际上是用于表示某个对象的字符串，函数会据此重新构建一个原始对象的副本。通过事件绑定，页面每次加载时都会调用 loadData() 函数。

与之类似，saveData() 函数操作 data 全局变量，它调用 JSON.stringify()，以将保存在 data 变量中的对象副本转换为字符串形式，进而将该字符串保存回 Storage 中。将 saveData() 函数绑定到页面卸载事件后，我们就能确保用户每次关闭浏览器或窗口时，都会调用 saveData() 函数。

上述两个函数的实际结果是，任何保存在 Storage 中的对象，不管它是否为复杂对象类型，我们都能做到在用户浏览时加载，在用户离开时保存。根据这种处理非文本数据的方法，开发人员可以进一步再扩展。

### 9.6.2 共享窗口

正如前面提到的，在任何同源浏览器窗口中触发 HTML5 Web Storage 事件的能力具有深远的意义。这意味着即使不是所有的窗口都使用了 Storage 对象，也可以基于 Storage 在窗口间发送消

息。这同样也意味着，我们可以在窗口间共享同源的数据。

让我们通过示例代码进行说明。要监听跨窗口消息，只需在脚本中绑定 Storage 事件处理程序即可。假设在 `http://www.example.com/storageLog.html` 网页中包含代码清单 9-15 所示的代码（这个示例文件 `storageLog.html` 也在 `code/storage` 文件夹中）。

#### 代码清单 9-15 使用 Storage 进行跨窗口通信

```
// 显示 Storage 事件中的记录
function displayStorageEvent(e) {
    var incomingRow = document.createElement('div');
    document.getElementById("container").appendChild(incomingRow);

    var logged = "key:" + e.key + ", newValue:" + e.newValue + ", oldValue:" +
        e.oldValue + ", url:" + e.url + ", storageArea:" + e.storageArea;
    incomingRow.innerHTML = logged;
}

// 为 Storage 事件添加监听器
window.addEventListener("storage", displayStorageEvent, true);
```

为 storage 事件类型绑定事件监听器后，窗口就能收到其他页面修改 Storage 后发出的更新通知了。例如，浏览器窗口打开了页面 `http://www.example.com/browser-test.html`，此时同源页面修改了 Storage 中的值，那么 `storageLog.html` 页面将会收到更新通知。为了将消息发送到接收窗口，发送窗口只需修改 Storage 对象，然后新旧数据值就会作为通知的一部分发送出去。举例来说，如果使用 `localStorage.setItem()` 更新了 Storage 数据，同源的 `storageLog.html` 页面的 `displayStorageEvent()` 函数将会收到一个事件。通过匹配事件名称和数据，两个页面就可以通信了，这在 HTML5 出现之前是非常困难的。图 9-12 显示了正在运行的 `storageLog.html` 页面，记录的是它接收到的 storage 事件。

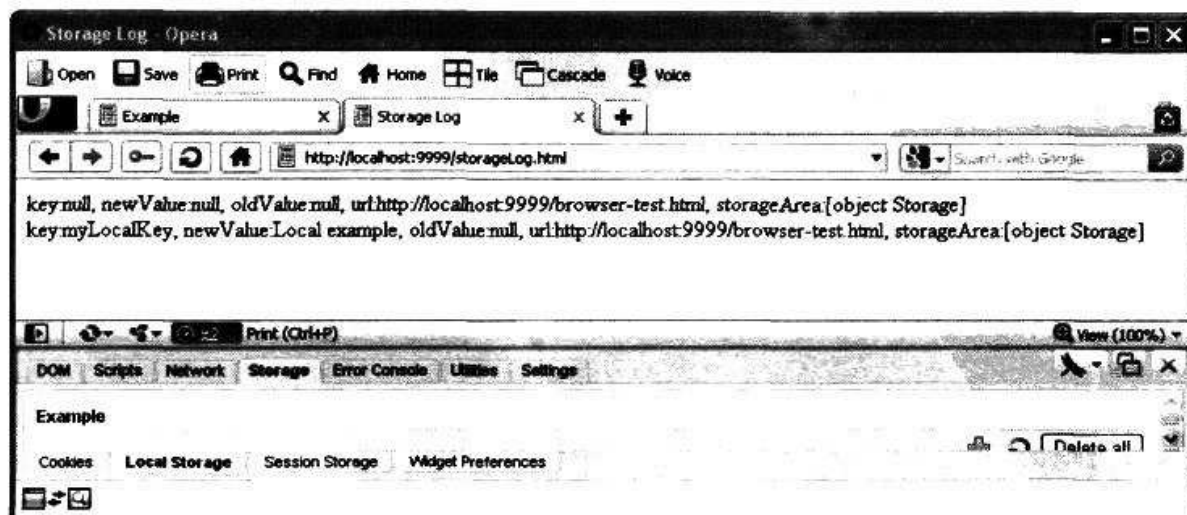


图 9-12 记录了 Storage 事件概况的 `storageLog.html` 页面



## 9.7 小结

本章中，我们讲解了如何用 HTML5 Web Storage 代替浏览器 cookie，在跨窗口、跨标签页甚至是（用 localStorage）跨浏览器重启的情况下，保存本地数据。随后，我们了解了如何使用 sessionStorage 将数据适当地隔离到各个窗口，以及如何基于 storage 事件实现跨窗口数据共享。在本章的示例中，我们演示了当用户浏览网站时，使用 Storage 在页面间跟踪数据的实用方法，这个方法很容易应用于其他数据类型。我们还演示了在页面加载或卸载时，如何存储非文本类型的数据，以保存和恢复跨页访问时网页的状态。

下一章，我们将介绍如何利用 HTML5 创建离线应用。

## 构建离线 Web 应用

## 10

**本章**，我们将探讨如何创建 HTML5 离线 Web 应用。HTML5 应用并不需要始终保持网络连接，而现在，开发人员可以更加灵活地控制缓存资源的加载。

## 10.1 HTML5 离线 Web 应用概述

很明显，在 Web 应用中使用缓存的原因之一是为了支持离线应用。在全球互联的时代，离线应用仍有其实用价值。当无法上网的时候，你会做些什么呢？有人可能会说如今网络无处不在，而且非常稳定，不存没有网络的情况。但事实果真如此吗？下面这些问题，你考虑到了吗？

- 我们乘坐的所有航班都有 Wi-Fi 吗？
- 我们的移动网络设备的信号好吗（最后一次遇到无信号是什么时候）？
- 我们去做讲演时，一定能够上网吗？

越来越多的应用移植到了 Web 上，我们倾向于认为用户拥有 24 小时不间断的网络连接。但事实上，网络连接中断时有发生，例如在乘坐飞机的情况下，可预见的中断时间一次就可能达到好几个小时。

间断性的网络连接一直是网络计算系统致命的弱点。如果应用程序依赖于与远程主机的通信，而这些主机又无法连接时，用户就无法正常使用应用程序了。不过当网络连接正常时，Web 应用程序可以保证及时更新，因为用户每次使用，应用程序都会从远程位置更新加载相关数据。

如果应用程序只需要偶尔进行网络通信，那么只要在本机存储了应用资源，无论是否连接网络它都可用。随着完全依赖于浏览器的设备的出现，Web 应用程序在不稳定的网络状态下还能够持续工作就变得更加重要。在这方面，不需要持续连接网络的桌面应用程序历来被认为比 Web 应用程序更有优势。

HTML5 的缓存控制机制综合了 Web 应用和桌面应用两者的优势：基于 Web 技术构建的 Web 应用程序，可在浏览器中运行并在线更新，也可在脱机情况下使用。然而，因为目前的 Web 服务器不为脱机应用程序提供任何默认的缓存行为，所以要想使用这一新的离线应用功能，你必须在应用中明确声明。

HTML5 的离线应用缓存使得在无网络连接状态下运行应用程序成为可能。这类应用程序用处很多，比如在起草电子邮件草稿时就无需连接因特网。HTML5 中引入了离线应用缓存，有了



它 Web 应用程序就可以在没有网络连接的情况下运行。

应用程序开发人员可以指定 HTML5 应用程序中，具体哪些资源（HTML、CSS、JavaScript 和图像）脱机时可用。离线应用的适用场景很多，例如：

- 阅读和撰写电子邮件；
- 编辑文档；
- 编辑和显示演示文档；
- 创建待办事宜列表。

使用离线存储，避免了加载应用程序时所需的常规网络请求。如果缓存清单（cache manifest）文件是最新的，浏览器就知道自己无需检查其他资源是否最新。大部分应用程序可以非常迅速地从本地应用缓存中加载完成。此外，从缓存中加载资源（而不必用多个 HTTP 请求确定资源是否已经更新）可节省带宽，这对于移动 Web 应用是至关重要的。目前，加载速度慢是 Web 应用比不上应用的一个地方。缓存则可以解决这一问题。

开发人员可以直接控制应用程序缓存。利用缓存清单文件可将相关资源组织到同一个逻辑应用中。这样一来，Web 应用就拥有了本来只属于桌面应用的特性。你可以充分发挥想象力，尝试用一些更巧妙的方式利用这些特性。

缓存清单文件中标识的资源构成了应用缓存（application cache），它是浏览器持久性存储资源的地方，通常在硬盘上。有些浏览器向用户提供了查看应用程序缓存中数据的方法。例如，在最新版本的 Firefox 中，`about:cache` 页面会显示应用程序缓存的详细信息，提供了查看缓存中的每个文件的方法，如图 10-1 所示。

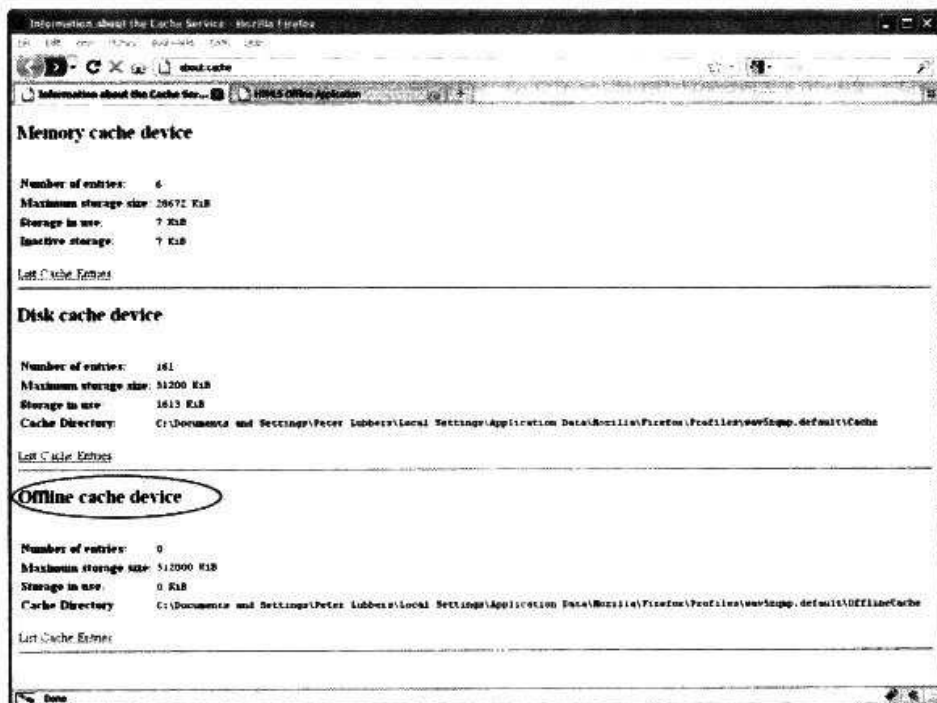


图 10-1 在 Firefox 中查看离线缓存

## HTML5 离线 Web 应用的浏览器支持情况

编写本书时，各浏览器对 HTML5 离线 Web 应用的支持情况见表 10-1。从下表中可以看出，许多浏览器已经支持 HTML5 离线 Web 应用。

表10-1 支持HTML5离线应用程序缓存的浏览器

浏览器	描述
Chrome	4.0及以上的版本均支持
Firefox	3.5及以上的版本均支持
Internet Explorer	不支持
Opera	10.6及以上的版本均支持
Safari	4.0及以上的版本均支持

由于目前不同的浏览器对于 HTML5 离线应用的支持程度不同，所以在使用之前最好先测试浏览器的支持情况。

## 10.2 使用 HTML5 离线 Web 应用 API

本节，我们将更加细致地探讨如何使用 HTML5 离线 Web 应用 API。

### 10.2.1 检查浏览器的支持情况

使用离线 Web 应用 API 前最好先检查浏览器是否支持它。代码清单 10-1 演示了检测方法。

代码清单 10-1 检查浏览器是否支持离线 Web 应用 API

```
if(window.applicationCache) {
    //浏览器支持离线应用
}
```

### 10.2.2 搭建简单的离线应用程序

假设开发人员希望搭建一个包含 HTML 文档、样式表和 JavaScript 文件的单页面应用程序，同时要为这个 HTML5 应用程序添加离线支持，那么可参考代码清单 10-2，在 html 元素中加入 manifest 特性。

代码清单 10-2 HTML 元素中的 manifest 特性

```
<!DOCTYPE html>
<html manifest="application.manifest">
.
.
.
</html>
```

修改完 HTML 文档，接下来需要提供缓存清单文件，用以指明哪些资源需要存储在缓存中。代码清单 10-3 是一个缓存清单文件的内容示例。



### 代码清单 10-3 缓存清单文件内容示例

```
CACHE MANIFEST
example.html
example.js
example.css
example.gif
```

## 10.2.3 支持离线行为

HTML5 引入了一些新的事件，用来让应用程序检测网络是否正常连接。应用程序处于在线状态和离线状态会有不同的行为模式。是否处于在线状态可以通过检测 `window.navigator` 对象的属性来做判断。首先，`navigator.onLine` 是一个标明浏览器是否处于在线状态的布尔属性。当然，`onLine` 值为 `true` 并不能保证 Web 应用程序在用户的机器上一定能访问到相应的服务器。而当其值为 `false` 时，不管浏览器是否真正联网，应用程序都不会尝试进行网络连接。代码清单 10-4 演示了如何查看页面状态是在线还是离线，它甚至可以在 Internet Explorer 中使用。

### 代码清单 10-4 检查在线状态

```
//页面加载的时候，设置状态为 online 或 offline
function loadDemo() {
    if (navigator.onLine) {
        log("Online");
    } else {
        log("Offline");
    }
}

// 添加事件监听器，在线状态发生变化时，触发相应动作
window.addEventListener("online", function(e) {
    log("Online");
}, true);

window.addEventListener("offline", function(e) {
    log("Offline");
}, true);
```

## 10.2.4 manifest 文件

离线应用包含一个 `manifest` (清单) 文件，此文件中列出了浏览器为离线应用缓存的所有资源。`manifest` 文件的 MIME 类型是 `text/cache-manifest`。Python 标准库中的 `SimpleHTTPServer` 模块对扩展名为 `manifest` 的文件能配以头部信息 `Content-type: text/cache-manifest`。配置方法是打开 `PYTHON_HOME / Lib / mimetypes.py` 文件并添加一行代码：

```
'manifest': 'text/cache-manifest manifest',
```

不同的 Web 服务器都有其独特的配置方法。例如，要配置 Apache HTTP 服务器，开发人员需要将下面一行代码添加到 `conf` 文件夹的 `mime.types` 文件中：

```
text/cache-manifest manifest
```

manifest 文件的写法是先写 CACHE MANIFEST，然后换行，每行单列资源文件。每行的换行符可以是 CR、LF 或者 CRLF——格式很灵活——但文本编码格式必须是 UTF-8。UTF-8 是多数文本编辑器经常输出的编码格式。

接下来，我们了解一下上面这个 manifest 文件示例的各个部分。

如果没有指定标题，默认就是 CACHE MANIFEST 部分。下面的 manifest 文件示例中指定了两个要缓存的文件：

```
CACHE MANIFEST
application.js
style.css
```

添加到 CACHE MANIFEST 区块中的文件，无论应用程序是否在线，浏览器都会从应用程序缓存中获取该文件。没有必要在这里列出应用程序的主 HTML 资源，因为最初指向 manifest 文件的 HTML 文档会被隐含包含进来。但是，如果希望缓存多个 HTML 文件，或者希望将多个 HTML 文件作为支持缓存的应用程序的可选入口，则需将这些文件都列在 CACHE MANIFEST 中。

FALLBACK 部分提供了获取不到缓存资源时的备选资源路径。代码清单 10-5 所示的 manifest 文件表明，当无法获取/app/ajax/\*时，所有对/app/ajax/及其子路径的请求都会被转发给 default.html 文件来处理。

#### 代码清单 10-5 manifest 文件示例（包含所有可能出现的区块）

```
#注释以#号开头
CACHE MANIFEST
#要缓存的文件
about.html
html5.css
index.html
happy-trails-rc.gif
lake-tahoe.JPG

#不缓存登录页面
NETWORK
signup.html

FALLBACK
signup.html      offline.html
/app/ajax/       default.html
```

NETWORK 部分罗列的资源，无论缓存中存在与否，均从网络获取。

### 10.2.5 applicationCache API

applicationCache API 是一个操作应用缓存的接口。新的 window.applicationCache 对象可触发一系列与缓存状态相关的事件。该对象有一个数值型属性 window.applicationCache.status，代表了缓存的状态。缓存状态共有 6 种，见表 10-2。



表10-2 缓存的6种状态

数值型属性	缓存状态
0	UNCACHED (未缓存)
1	IDLE (空闲)
2	CHECKING (检查中)
3	DOWNLOADING (下载中)
4	UPDATEREADY (更新就绪)
5	OBSOLETE (过期)

目前 Web 上大部分的页面都没有指定缓存清单, 所以这些页面的状态就是 UNCACHED (未缓存)。IDLE (空闲) 是带有缓存清单的应用程序的典型状态。处于空闲状态说明应用程序的所有资源都已被浏览器缓存, 当前不需要更新。如果缓存曾经有效, 但现在 manifests 文件丢失, 则缓存进入 OBSOLETE (过期) 状态。对于上述各种状态, API 包含了与之对应的事件 (和回调特性)。例如, 当缓存更新完成进入空闲状态时, 会触发 cached 事件。此时, 可能会通知用户, 应用程序已处于离线模式可用的状态, 可以断开网络连接了。表 10-3 是一些与缓存状态有关的常见事件。

表10-3 常见事件及其关联的缓存状态

事 件	关联的缓存状态
onchecking	CHECKING
ondownloading	DOWNLOADING
onupdateready	UPDATEREADY
onobsolete	OBSOLETE
oncached	IDLE

此外, 没有可用更新或者发生错误时, 还有一些表示更新状态的事件:

- onerror
- onnoupdate
- onprogress

window.applicationCache 有一个 update() 方法。调用 update() 方法会请求浏览器更新缓存。包括检查新版本的 manifest 文件并下载必要的新资源。如果没有缓存或者缓存已过期, 则会抛出错误。

### 10.3 使用 HTML5 离线 Web 应用构建应用

在下面的示例中, 我们将跟踪跑步者出发后的位置 (包括间断性网络连接或无连接的情况)。比如, Peter 在跑步时将一款具有 HTML5 Geolocation 功能和支持 HTML5 Web 浏览器的手机带在身上, 不过房子附近树林中的信号不是很好, 所以他希望即使无法联网也能定位并记录自身位置。

离线状态下, Geolocation API 可以在使用硬件地理定位的设备 (例如 GPS) 上继续工作, 但在使用 IP 定位的设备上不可以, 因为 IP 地理定位设备需要连接网络, 以便将客户端的 IP 地址映射为坐标。离线应用程序还可以通过像本地存储或者 Web SQL Database 这样的接口访问本地存储。图 10-2 是该应用运行时的截图。

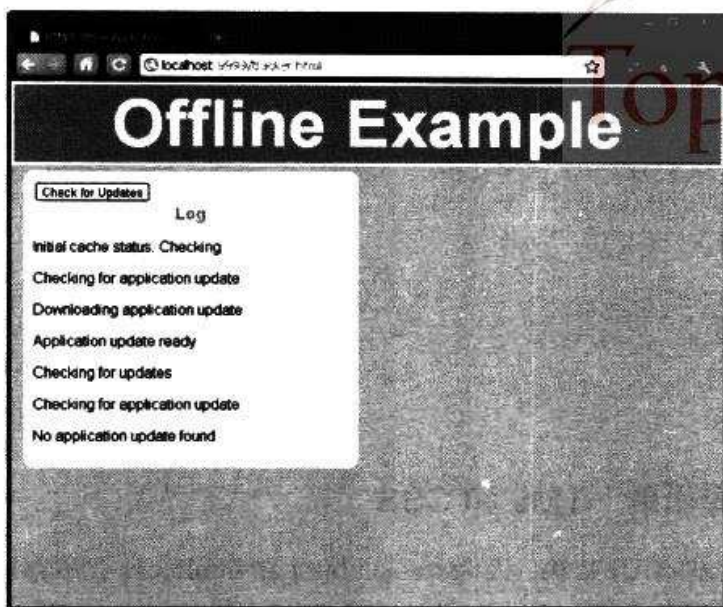


图 10-2 运行中的应用程序

运行此应用程序需要一台 Web 服务器来提供所需静态资源。需要注意，manifest 文件的内容类型必须配置为 text/cache-manifest 发送到浏览器。如果文件类型不正确，即使浏览器支持应用缓存也会返回缓存错误。

要运行此应用程序的全部功能，服务器还需要具备接收地理位置数据的功能。本例中，服务器端的主要任务是存储、分析和提供这些数据。静态应用程序中，数据的获取不局限于同源。图 10-3 是应用程序在 Firefox 中以离线模式运行时的截图。应用程序的示例文件在本书网站上的“Offline”部分。

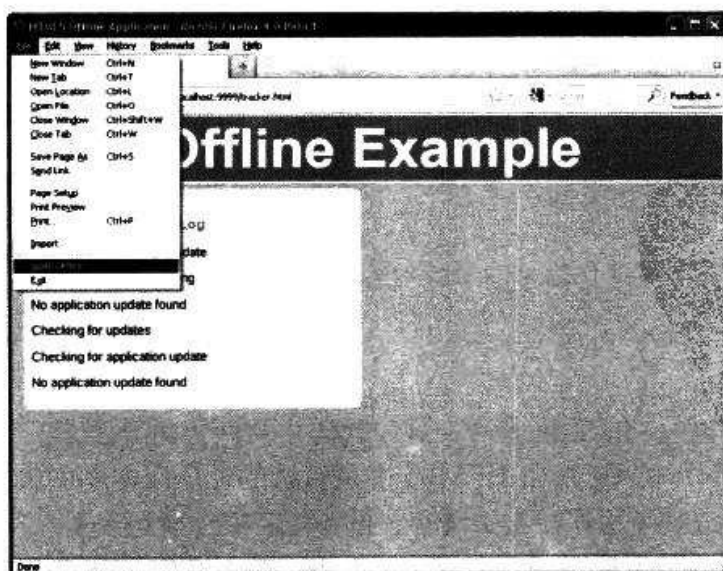


图 10-3 在离线模式下运行的应用程序



### 10.3.1 创建记录资源的 manifest 文件

首先,按照以下方式创建 `tracker.manifest` 文件。文件中列出了应用程序需要缓存的资源:

```
CACHE MANIFEST
# JavaScript
./offline.js
# ./tracker.js
./log.js

# stylesheets
./html5.css

# images
```

### 10.3.2 创建构成界面的 HTML 和 CSS

下面是本章示例的基本 UI 结构。因为 `tracker.html` 和 `html5.css` 会被存储到缓存中,所以应用程序将从缓存中提取这两个文件。

```
<!DOCTYPE html>
<html lang="en" manifest="tracker.manifest">
<head>
  <title>HTML5 Offline Application</title>
  <script src="log.js"></script>
  <script src="offline.js"></script>
  <script src="tracker.js"></script>
  <link rel="stylesheet" href="html5.css">
</head>

<body>
  <header>
    <h1>Offline Example</h1>
  </header>

  <section>
    <article>
      <button id="installButton">Check for Updates</button>
      <h3>Log</h3>
      <div id="info">
      </div>
    </article>
  </section>
</body>
</html>
```

从实现应用程序的离线功能角度来看,这段代码中有两点需要注意。第一点是 HTML 元素的 `manifest` 特性。因为 `<html>` 元素在 HTML5 中是可选的,所以本书中大部分的 HTML 示例都省略了它。不过,如果希望应用程序支持缓存,就不能省略 `<html>` 元素,并需要在该元素中设置 `manifest` 特性,因为应用程序是否缓存离线文件取决于是否指定了 `manifest` 文件。

第二点要注意的是按钮,它的作用是让用户能够手动安装 Web 应用程序,以支持离线情况。

### 10.3.3 创建离线 JavaScript

示例中，JavaScript 文件由多个通过<script>标签包含进来的.js 文件组成。这些 js 脚本会同 HTML 和 CSS 文件一起存储到缓存中。

```
<offline.js>
/*
 * 记录 window.applicationCache 触发的每个事件
 */

window.applicationCache.onchecking = function(e) {
    log("Checking for application update");
}

window.applicationCache.onnoupdate = function(e) {
    log("No application update found");
}

window.applicationCache.onupdateready = function(e) {
    log("Application update ready");
}

window.applicationCache.onobsolete = function(e) {
    log("Application obsolete");
}

window.applicationCache.ondownloading = function(e) {
    log("Downloading application update");
}

window.applicationCache.onsuccess = function(e) {
    log("Application cached");
}

window.applicationCache.onerror = function(e) {
    log("Application cache error");
}

window.addEventListener("online", function(e) {
    log("Online");
}, true);

window.addEventListener("offline", function(e) {
    log("Offline");
}, true);

/*
 * 将 applicationCache 状态代码转换成消息
 */
showCacheStatus = function(n) {
    statusMessages = ["Uncached", "Idle", "Checking", "Downloading", "Update Ready", "Obsolete"];
    return statusMessages[n];
}
```



```
install = function() {
    log("Checking for updates");
    try {
        window.applicationCache.update();
    } catch (e) {
        applicationCache.onerror();
    }
}

onload = function(e) {
    // 检测所需功能的浏览器支持情况
    if (!window.applicationCache) {
        log("HTML5 Offline Applications are not supported in your browser.");
        return;
    }

    if (!navigator.geolocation) {
        log("HTML5 Geolocation is not supported in your browser.");
        return;
    }

    if (!window.localStorage) {
        log("HTML5 Local Storage not supported in your browser.");
        return;
    }

    log("Initial cache status: " + showCacheStatus(window.applicationCache.status));
    document.getElementById("installButton").onclick = checkFor;
}

<log.js>
log = function() {
    var p = document.createElement("p");
    var message = Array.prototype.join.call(arguments, " ");
    p.innerHTML = message;
    document.getElementById("info").appendChild(p);
}
}
```

### 10.3.4 检查 applicationCache 的支持情况

除了离线应用缓存，示例中还使用了地理定位和本地存储。在页面加载前应确保浏览器支持这两种功能。

```
onload = function(e) {
    // 检测所需功能的浏览器支持情况
    if (!window.applicationCache) {
        log("HTML5 Offline Applications are not supported in your browser.");
        return;
    }

    if (!navigator.geolocation) {
        log("HTML5 Geolocation is not supported in your browser.");
    }
}
```

```
        return;
    }

    if (!window.localStorage) {
        log("HTML5 Local Storage not supported in your browser.");
        return;
    }

    if (!window.WebSocket) {
        log("HTML5 WebSocket is not supported in your browser.");
        return;
    }

    log("Initial cache status: " + showCacheStatus(window.applicationCache.status));
    document.getElementById("installButton").onclick = install;
}
}
```

### 10.3.5 为 Update 按钮添加处理函数

接下来，下面的代码用来处理更新行为，其作用是更新应用缓存：

```
install = function() {
    log("Checking for updates");
    try {
        window.applicationCache.update();
    } catch (e) {
        applicationCache.onerror();
    }
}
```

单击按钮后将检查缓存区，并更新需要更新的缓存资源。当所有可用更新都下载完毕之后，将在用户界面显示一条消息，告诉用户应用程序已安装成功，可以在离线模式下运行了。

### 10.3.6 添加 Geolocation 跟踪代码

下面的代码修改自第 4 章的地理定位示例代码。它是 tracker.js 文件的一部分。

```
/*
 * 定位并报告当前位置
 */
var handlePositionUpdate = function(e) {
    var latitude = e.coords.latitude;
    var longitude = e.coords.longitude;
    log("Position update:", latitude, longitude);
    if(navigator.onLine) {
        uploadLocations(latitude, longitude);
    }
    storeLocation(latitude, longitude);
}

var handlePositionError = function(e) {
    log("Position error");
}
```



```
var uploadLocations = function(latitude, longitude) {
    var request = new XMLHttpRequest();
    request.open("POST", "http://geodata.example.net:8000/geoupload", true);
    request.send(localStorage.locations);
}

var geolocationConfig = {"maximumAge":20000};

navigator.geolocation.watchPosition(handlePositionUpdate,
                                     handlePositionError,
                                     geolocationConfig);
```

### 10.3.7 添加 Storage 功能代码

当应用程序处于离线状态时，需要将数据更新写入本地存储，接下来，我们就添加这方面的代码。

```
var storeLocation = function(latitude, longitude) {
    // 加载 localStorage 的位置列表
    var locations = JSON.parse(localStorage.locations || "[]");
    // 添加地理位置数据
    locations.push({"latitude" : latitude, "longitude" : longitude});
    // 保存新的位置列表
    localStorage.locations = JSON.stringify(locations);
}
```

在第 9 章中我们介绍了 HTML5 的 localStorage，本章的这个示例应用程序就是通过它来保存坐标的。因为 localStorage 可以将数据存储在本地浏览器中，所以它特别适用于具有离线功能的应用程序。本地存储中的数据在将来的会话中可用。当网络连接恢复正常后，应用程序就可以与远程服务器进行数据同步。

这里使用 Storage 还有一个好处，那就是当上传请求失败后可以通过 Storage 得到恢复。如果应用程序遇到某种原因导致的网络错误，或者应用程序被关闭（可能是用户关闭浏览器、浏览器或操作系统崩溃以及页面导航跳转等）的时候，数据会被存储以便下次再进行传输。

### 10.3.8 添加离线事件处理程序

位置更新处理程序运行时，会去检查网络连接状态。如果应用程序在线，事件处理函数会存储并上传当前坐标。如果应用程序离线，事件处理函数只存储不上传。当应用程序重新连接到网络后，事件处理函数会在 UI 上显示在线状态，并在后台上传之前存储的所有数据。

```
window.addEventListener("online", function(e) {
    log("Online");
}, true);

window.addEventListener("offline", function(e) {
    log("Offline");
}, true);
```

网络连接状态在应用程序没有真正运行的时候可能会发生改变。例如用户关闭了浏览器、刷新页面或跳转到了其他网站。为了应对这些情况，我们的离线应用程序在每次页面加载时都会检查与服务器的连接情况。如果连接正常，会尝试与远程服务器同步数据。

```
//如果浏览器在线，与服务器同步
if(navigator.onLine) {
    uploadLocations();
}
```

## 10.4 小结

本章，我们学习了如何基于 HTML5 离线 Web 应用创建即使没有因特网连接也可使用的应用程序。为确保应用中所需的文件能够成功缓存，需要将这些文件指定在 manifest 文件中，随后在应用程序的主页面中进行引用。然后，添加监听器监听在线和离线状态的变化，进而基于因特网连接与否让网站执行不同的操作。

最后一章，我们将对 HTML5 编程的前景进行展望。



# HTML5 未来展望

本书前面的章节中已经介绍了 HTML5 所提供的强大编程功能。此外，我们还讨论了 HTML5 的发展历程和新的无需插件的开发模式。本章，我们将讨论一些还不够成熟但却极具潜力的 HTML5 特性。

## 11.1 HTML5 的浏览器支持情况

随着版本的不断升级，浏览器对 HTML5 功能的支持度越来越高。在写作本书期间，前面章节中讨论过的许多功能都已经得到了浏览器正式支持，HTML5 在浏览器中的发展无疑获得了巨大的推动力。

微软意外地发布了下一代浏览器的一系列开发者预览版本。IE9 将支持一些关键的 HTML5 功能并大幅提升运行速度。一旦 IE9 正式版发布，诸如多媒体、存储和通信等 HTML5 的大部分高级功能就将成为主流浏览器的必备功能。

如今，许多开发人员还在拼命开发所谓“健壮”的 Web 应用，耗费大量精力以兼容旧浏览器。截止 2010 年，在因特网中仍然存在着大量过时的浏览器，IE6 就是这种残酷现状的代表。即便如此，IE6 的寿命也是有限的，因为支持 IE6 的操作系统越来越难找了。总有一天，IE6 的用户数会接近零。越来越多的 IE 用户已经升级到了新版本，而 IE9 将是 Web 发展的一个分水岭。无论何时，总会存在一个相对最旧的浏览器，而随着时间的推移，最旧的浏览器终会被淘汰。在编写本书的时候，IE6 的市场占有率已降至 20% 以下，而且还在持续下滑。很多用户直接升级到了最先进的浏览器。未来的主流浏览器最起码应该支持 HTML5 的 Video、Canvas 和 WebSocket，以及所有原来需要 hack 才能实现跨浏览器兼容的功能。

本书前面讨论的 HTML5 功能都已经基本稳定且被多数浏览器支持。除此之外，还有一些其他的 HTML 扩展功能及 API 尚处在开发初期。本章，我们就来看一些尚未发布的功能。这其中有些处于前期试验阶段，而有些只需要进行细微修改即可形成最终规范并广泛推广。

## 11.2 HTML 未来的发展

我们在本节讨论的几项激动人心的功能很可能在不久的将来就会出现在浏览器中。这些功能



的实现可能也无需等到 2022 年。将来很可能没有 HTML6 规范，因为 WHATWG 曾暗示未来的规范将被统一简称为“HTML”。HTML 将以增量的形式发展而非一蹴而就，其中特定的功能及其对应规范会单独发展。大家就这些功能达成共识之后，浏览器才会采纳并加以实现。在 HTML5 稳定之前，一些即将发布的功能很可能已经在浏览器中大规模使用了。负责推动 Web 发展的组织为了满足用户和开发人员的需求，一直致力于 Web 平台的开发升级。

### 11.2.1 WebGL

WebGL 是针对 Web 上 3D 图像的 API。历史上，Mozilla、Opera、Google 等浏览器厂商曾分别提供了试验性的 JavaScript 3D API。今天，WebGL 正向规范化方向迈进，而越来越多的 HTML5 浏览器加入了支持它的阵营。WebGL 的规范化进程由浏览器厂商和 The Khronos Group（负责 OpenGL 的组织，OpenGL 是 1992 年创建的跨平台 3D 绘图标准）共同推进。OpenGL 规范的当前版本为第 4 版，它已被广泛应用于游戏和计算机辅助设计中，并成为了微软 Direct3D 的有力竞争对手。

我们在第 2 章看到了在 canvas 元素中调用 `getContext("2d")`，可以获得 2D 绘图的上下文。毫无疑问，这为其他类型的绘图上下文提供了方便之门。WebGL 使用的同样是 canvas 元素，只不过获取的是 3D 上下文。由于还在试验期，所以调用 `getContext()` 时，传入的参数需要使用浏览器厂商指定的名称作为前缀（`moz-webgl`、`webkit-3d` 等）。例如，在支持 WebGL 的 Firefox 版本中，可以在 canvas 元素中调用 `getContext("moz-webgl")` 来获取 3D 上下文。这里调用 `getContext()` 返回的 API 对象与 2D canvas 的有所不同，因为它提供的是 OpenGL 绑定，而非绘图操作。WebGL 版本的 canvas 上下文管理的是纹理和顶点缓冲区，而不是调用函数来绘制线条和填充形状。

#### 1. 3D HTML

与其他 HTML5 元素一样，WebGL 将会成为 Web 平台不可或缺的一部分。因为 WebGL 通过 canvas 元素来渲染，所以它属于 document 对象。你可以像操作图像或 2D canvas 那样，在页面 3D canvas 元素中应用定位和变换。实际上，任何在 2D canvas 上能做的事情，在 3D canvas 上都能做，比如叠放文本和视频、执行动画等。与纯粹的 3D 显示技术相比，结合其他文档元素，3D canvas 不仅可以用于创建 HUD（Heads-Up Display，平视显示器），而且使得 2D 与 3D 的混合界面变得更易于开发了。想象一下，在 3D 场景中利用 HTML 标签构造一个 Web 用户界面会是怎样的效果。许多 OpenGL 应用程序使用了非原生的菜单和控件，而 WebGL 则可以使用原生的样式美观的 HTML5 表单元素。

现有的 Web 网络架构也为 WebGL 的开发提供了便利。WebGL 应用程序可以通过 URL 加载纹理、模型等资源。多人游戏可以基于 WebSocket 进行通信。例如，Google 最近使用 HTML5 WebSocket、Audio、WebGL 等技术将经典的 3D 游戏 Quake II 移植到了 Web 上，并加入了多人竞争机制，如图 11-1 所示。游戏逻辑和图形使用 JavaScript 实现，页面呈现通过调用 WebGL canvas 完成。游戏使用持久化的 WebSocket 连接来保持与服务器间的通信，从而实现对不同玩家位置的调整。





图 11-1 Quake II

## 2. 3D着色器

WebGL 是 OpenGL ES 2 与 JavaScript 的结合，因此，它可以使用 OpenGL 中标准化的编程图形管道，包括着色器 (shader)。着色器可将高度灵活的渲染效果应用于 3D 场景，让显示效果更真实。WebGL 着色器是用 GLSL (GL Shading Language, GL 着色语言) 编写的，这是 Web 中又一种新的专用语言。HTML5 的 WebGL 应用程序使用 HTML 搭建框架，用 CSS 控制样式，用 JavaScript 处理逻辑，用 GLSL 进行着色。开发人员可以借鉴在 OpenGL 着色器方面的开发经验，按照类似的方式使用 WebGL API。

WebGL 可以当做是 Web 3D 图形的基础层。与对 DOM 进行抽象并提供若干强大功能的 JavaScript 库类似，有一些库以 WebGL 为基础，提供了若干额外功能。正在开发的库涵盖了多个方面，有场景图，有 3D 文件格式 (如 COLLADA)，还有游戏开发的完整引擎。图 11-2 显示的是 Shader Toy，它是由 Inigo Quilez 搭建的 WebGL 着色工作台，其上提供的场景渲染效果是他与其他 9 名场景渲染艺术家一起设计的。这张截图显示的是 Rgba 的 Leizex。我们希望在不久的将来，随着高级渲染库的蓬勃发展，即便是 Web 编程新手也能轻松地创建出 3D 场景来。

---

**提示** WebGL 不仅将 OpenGL 引入到 JavaScript 中，它还在尝试分拆出一个通用的二进制数据 API。对于快速的 3D API 来说，二进制缓冲是必要的。对此，有一个新的 API 提案，提案中引入了名为 TypedArray 的 API 专门用来操作二进制数据。若该提案生效的话，TypedArray 将在 HTML5 中发挥很大作用。

---



图 11-2 Shader Toy 是一个 WebGL 着色工作台

### 11.2.2 设备

Web 应用很可能需访问多种硬件，如网络摄像头、麦克风或是已连接的存储设备。为此，HTML5 设计了 device 元素，以便让应用程序访问所连接硬件的数据流。当然，对于隐私的保护是非常严格的，所以不是每段脚本都可以随意使用用户的摄像头。如同 Geolocation 和 Storage API 一样，当应用程序请求提升权限时，浏览器就会弹出提示，并请求用户确认。与网络摄像头相关的应用主要是视频会议，不过还有许多其他令人惊奇的 Web 应用功能，比如增强现实 (augmented-reality) 和头部跟踪。

### 11.2.3 音频数据 API

可编程的音频 API 与 <audio> 的关系类似于 <canvas> 与 <img> 间的关系。在 canvas 标签出现之前，Web 图像对脚本而言基本是不透明的。图像的创建和操作只能发生在服务器端。如今，已经有工具可以基于 canvas 元素来创建和操作可视化媒体了。同样地，在 HTML5 应用中可以使用音频数据 API 创建音乐。这将有助于丰富 Web 应用程序的内容创建能力，推进媒体创作工具与消费平台一体化 (self-hosting) 时代的到来。可以想象，将来会有一天，用户仅仅通过浏览器就能完成音频的创建和编辑。

简单的音频播放功能可使用 <audio> 元素来实现。但是，应用程序对音频的操作、分析或联



机创建都需要底层 API。若无法访问音频数据,那么诸如文本到语音的转换、语音到语音的转换、合成器、音乐可视化等功能都是空谈。

将来,标准音频 API 不仅有望操作数据元素中的麦克风输入,而且还有可能操作<audio>标签内引入的文件。通过<device>和音频数据 API,可以构建一个允许用户在页面内录制和编辑声音的 HTML5 应用程序。音频片段可存储在本地浏览器缓存中以便重用,还可与基于 canvas 的编辑工具相结合。

对此,目前 Mozilla 的每夜构建版本中有一种试验性的实现方式。Mozilla 音频数据 API 可被当做实现标准跨浏览器音频编程功能的基础。

### 11.2.4 视频元素改进

本书前面章节详细介绍了<video>元素的现有功能。在 HTML5 仍然迟迟未能指定必选视频编解码器的时候,Google 最近发布了一种面向 Web 的高质量、免费的网络视频编解码器——WebM。虽然并不是所有厂商都支持 WebM,但这种格式背后的支持力量还是很强大的。

此外,对当前 HTML5 视频控件进行改进的需求也与日俱增。为了全面取代基于插件的视频展现方式,HTML 自身控件需要引入更好的用于流媒体的 API,期待这种能力能在下一代媒体标签中有所发挥。不支持 Flash 的设备越来越多,如 Apple iPad 和 iPhone,所以要让大众用户正常浏览网络视频,只能依靠 HTML5,这是 HTML5 发展的重要契机。

### 11.2.5 触摸屏设备事件

Web 访问方式越来越多地从台式机和笔记本电脑转换到了手机和平板电脑上,因此 HTML5 的交互处理方式也在逐渐发生变化。Apple 在推出 iPhone 的同时,也将一系列特殊事件引入到了其浏览器当中,这些事件用来处理多点触摸输入和设备旋转。尽管还未被标准化,但这些事件已被其他移动设备厂商选用。学会此类事件,就可以针对当今最先进的设备开发出合适的 Web 应用程序。

#### 1. 方向事件

方向事件是移动设备中最简单的事件,它可以加入到页面 body 标签中:

```
<body onorientationchange="rotateDisplay();">
```

在方向事件处理程序中,可以引用 window.orientation 属性。该属性可选的值如表 11-1 所示,它们以页面首次加载时设备的方向为基准。

表 11-1 方向值及其含义

方向值	含 义
0	页面当前方向与首次加载时的原始方向一样
-90	与原始方向相比,设备顺时针旋转了 90 度(向右)
180	与原始方向相比,设备旋转了 180 度(垂直翻转)
90	与原始方向相比,设备逆时针旋转了 90 度(向左)

获得方向值后,就可对设备的内容显示进行相应的调整了。

## 2. 手势事件

移动设备支持的另一种事件相对高级一些，称为手势事件。手势事件可以理解为通过多点触摸引发的缩放或旋转。当用户有两个或多个手指同时在触摸屏上挤压 (pinch) 或扭转 (twist) 时，就会触发手势事件。扭转表示旋转，挤压 (pinch in) 和伸展 (pinch out) 分别表示缩小和放大。为了接收到手势事件，代码需要注册表 11-2 中所示的事件处理程序。

表11-2 手势事件处理程序

事件处理程序	描 述
ongesturestart	用户将多个手指放在触摸屏上，并开始滑动
ongesturechange	用户正在使用手指动作进行缩放或是旋转操作
ongestureend	用户移开手指，缩放或旋转操作已经完成

在用户做手势的过程中，事件处理程序会灵活检测事件的缩放或旋转属性，并对显示效果进行相应更新。代码清单 11-1 是手势处理函数的示例。

代码清单 11-1 手势处理函数示例

```
function gestureChange(event) {
    //获取用户手势生成的缩放量
    //1.0 代表初始大小，小于1.0 表示缩小，大于1.0 表示放大
    //基于尺寸大小按比例放大和缩小
    var scale = event.scale;

    //获取用户手势生成的旋转量
    //旋转值介于0 度到 360 度之间
    //正值代表顺时针旋转，负值代表逆时针方向旋转
    var rotation = event.rotation;

    //基于旋转操作更新页面显示
}

//为文档节点添加手势变换监听程序
node.addEventListener("gesturechange", gestureChange, false);
```

在需要操作物件或者以显示为主的应用程序（如绘图工具和导航工具等）中，经常需要用到手势事件。

## 3. 触摸事件

如果需要在低层次上处理设备事件，可以通过触摸事件获取所需信息。表 11-3 所示为不同的触摸事件。

表11-3 触摸事件

事件处理程序	描 述
ontouchstart	已经在触摸设备表面放置了一个手指。当多个手指放在设备上时，会发生多点触摸事件
ontouchmove	在拖动操作中，一个或多个手指发生了移动
ontouchend	一个或多个手指离开设备表面
ontouchcancel	意外中断停止了触摸操作



与其他移动设备事件不同，触摸事件需要考虑多点数据（多个手指可能会同时触摸）同时出现的情况。因此，用于处理触摸事件的 API 会相对复杂一些，如代码清单 11-2 所示。

代码清单 11-2 用于处理触摸事件的 API

```
function touchMove(event) {  
  // touches 变量是一个列表，包含了当前每个手指的触摸点信息  
  var touches = event.touches;  
  
  // changedTouches 列表包含当前触摸状态发生变化的手指的触摸点信息，如添加、移开或重放手指  
  var changedTouches = event.changedTouches;  
  
  // 监听器注册在哪个节点上，targetTouches 就包含哪个节点上发生的触摸操作的触摸点信息  
  var targetTouches = event.targetTouches;  
  
  // 在取得预定的触摸点之后，可以取得能够从其他事件对象中正常获取的大部分信息  
  var firstTouch = touches[0];  
  var firstTouchX = firstTouch.pageX;  
  var firstTouchY = firstTouch.pageY;  
}  
  
// 为我们的示例注册一个触摸事件监听器  
node.addEventListener("touchmove", touchMove, false);
```

不难发现，设备的原生事件处理机制可能会干扰手势事件和触摸事件的处理。在这种情况下，可调用：

```
event.preventDefault();
```

上述代码会覆盖浏览器界面的默认行为，让开发人员自行处理事件。在所有移动事件被标准化之前，建议开发人员先仔细阅读相应设备的开发文档，再进行开发。

### 11.2.6 P2P 网络

在 Web 应用程序中，高级网络技术始终在向前发展。不论是 HTTP 还是 WebSocket，都有客户端（浏览器或其他用户代理）和服务端（URL 主机）。P2P 网络允许客户端之间直接通信。通常情况下，这比从服务器发送数据更高效，而且有助于降低托管成本，提高应用的性能。要想开发更快的多人游戏和协作软件，P2P 技术是不二之选。

在 HTML5 中，P2P 结合 device 元素可以构建高效视频聊天应用。在 P2P 视频聊天应用中，对话双方能够直接互传数据，无需通过中心服务器。在 HTML5 之外，P2P 视频聊天已广泛应用于诸如 Skype 等应用程序中。由于流媒体视频对带宽要求比较高，如果没有 P2P 通信技术的话，很可能那些流媒体应用程序一个都无法实现。

针对 P2P 网络，浏览器厂商也进行了一些试验，例如 Opera 的 Unite 技术，该技术会直接在浏览器中部署一个简易的 Web 服务器。Opera Unite 允许用户为别人创建和提供服务，从而实现聊天、文件共享和文档协作等功能。

当然，面向 Web 的 P2P 网络不仅需要考虑如何为开发人员提供编程 API，还需要引入一个兼顾安全和网络媒介的协议。



## 11.2.7 最终方向

到目前为止，我们一直着眼于如何让开发人员构建强大的 HTML5 应用程序。我们考虑到的另一种视角是基于 HTML5 的 Web 应用如何能让用户获得上佳的体验。许多 HTML5 功能旨在消除或降低脚本的复杂性，提供之前需要插件才能实现的功能。拿 HTML5 的 video 元素来说，无需任何 JavaScript 即可指定控制界面、自动播放、缓冲行为和生成并显示缩略图。而通过 CSS3 可以直接在样式文件中实现以前用脚本才能实现的动画效果。这些声明式代码让应用程序更容易被用户理解，并最终让用户获得更多掌控。

前面的章节中，你已经见过 Firefox 和 WebKit 的开发工具是怎样揭示 HTML5 各种功能信息的，包括存储功能，以及至关重要的 JavaScript 调试、性能分析、命令行求值等功能。因此，简单的声明性代码、浏览器内置或 Web 应用程序自带的轻量级工具将是 HTML5 未来的发展方向。

Google 显然对 HTML 的未来发展充满信心——它宣布即将发布 Google Chrome 操作系统。这是一种围绕浏览器和媒体播放器开发的精简版操作系统。到 2010 年底，Google 操作系统的目标是包含丰富的基于 HTML API 实现的功能，以提供完美的用户体验，同时使其上运行的应用程序完全符合标准的 Web 体系架构。

## 11.3 小结

本书主要讲解了如何使用强大的 HTML5 API，希望各位读者能够发挥自己的智慧，创建出色的应用！

这是本书的最后一章，我们介绍了一些即将纳入 HTML5 规范的内容，如 3D 图形、device 元素、触摸事件、P2P 网络等。HTML5 还在快速发展，非常值得我们密切关注。

各位读者可能接触 Web 或者从事 Web 开发超过十年的时间了，回想一下最近几年 HTML 技术带来了多大的变化。十年前，“专业 HTML 编程”意味着学习 HTML4 的新功能。当时顶尖的开发者刚刚发现动态页面更新和 XMLHttpRequest。虽然 Ajax 技术得到了广泛关注，但那也是在发明“Ajax”这个术语几年以后的事了。当时，浏览器中的大多数专业编程还只是限于讨论框架和操作图像地图而已。

曾经需要数页脚本代码才能实现的功能，如今只需一个标签就能轻松搞定。只要愿意，你随时都可以下载一个支持 HTML5 的浏览器，打开自己喜欢的文本编辑器，运用我们讲到的那些通信与交互手段，亲自动手尝试一下令人心驰神往的 HTML5 编程吧！

本书针对 Web 开发进行了一系列的探讨，希望你能喜欢这些内容，同时期待我们的抛砖引玉能够激发你的创造性。十年后，我们盼望能够再写一本书，而书里会介绍到你用 HTML5 创造的伟大应用。



## 图书在版编目 (CIP) 数据

HTML5高级程序设计 / (荷) 柳伯斯 (Lubbers, P.), (美) 阿伯斯 (Albers, B.), (美) 萨利姆 (Salim, F.) 著; 李杰, 柳靖, 刘森译. — 北京: 人民邮电出版社, 2011. 1

(图灵程序设计丛书)

书名原文: Pro HTML5 Programming: Powerful APIs for Richer Internet Application Development  
ISBN 978-7-115-24487-1

I. ①H… II. ①柳… ②阿… ③萨… ④李… ⑤柳… ⑥刘… III. ①超文本标记语言, HTML—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字(2010)第236844号

## 内 容 提 要

本书首先介绍了HTML5的历史背景、新的语义标签及与以往HTML版本相比的根本变化,同时揭示了HTML5背后的设计原理。从第2章起,分别围绕构建令人神往的富Web应用,逐一讨论了HTML5的Canvas、Geolocation、Communication、WebSocket、Forms、Web Workers、Storage等API的使用,辅以直观明了的客户端和服务端示例代码,让开发人员能够迅速理解和掌握新一代Web标准所涵盖的核心技术。本书最后探索了离线Web应用并展望了HTML5未来的发展前景。

本书面向有一定经验的Web应用开发人员,对HTML5及未来Web应用技术发展抱有浓厚兴趣的读者也可以学习参考。

## 图灵程序设计丛书 HTML5高级程序设计

- 
- ◆ 著 [荷] Peter Lubbers  
[美] Frank Salim
  - 译 李杰 柳靖 刘森
  - 责任编辑 朱 巍
  - 执行编辑 毛倩倩
  - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号  
邮编 100061 电子函件 315@ptpress.com.cn  
网址 <http://www.ptpress.com.cn>  
北京艺辉印刷有限公司印刷
  - ◆ 开本: 800×1000 1/16  
印张: 14.75  
字数: 357千字 2011年1月第1版  
印数: 1-4 000册 2011年1月北京第1次印刷
- 著作权合同登记号 图字: 01-2010-6674号  
ISBN 978-7-115-24487-1
- 

定价: 45.00元

读者服务热线: (010)51095186 印装质量热线: (010)67129223

反盗版热线: (010)67171154



“三位作者共同操刀，为我们呈献了功能强大且实用的HTML5代码。本书打破了HTML5在近几年尚不可用的魔咒，深入讲解了具有主流浏览器支持的API的使用方法，解答了我关于HTML5标准的许多疑问，建议每位熟悉HTML并想使用新标准的人员都阅读本书。”

——亚马逊读者评论

# HTML5高级程序设计

今天，大多数现代浏览器都已经支持HTML5。学习体验HTML5带给Web开发的便捷、快速和强大功能，是每一位Web设计和开发人员的当务之急。

本书由旧金山HTML5用户组创建人联合另两位资深Web开发专家共同打造，为读者清晰解读了HTML5规范的缘由、发展和现状，全面展示了如何使用WebSocket、Geolocation、Web Storage、Canvas及音频视频等前所未有的新特性构建先进的Web应用，并以大量的示例涵盖全部HTML5 API。此外，还介绍了当今浏览器对HTML5特性的支持情况。

**Pro HTML5 Programming**  
Powerful APIs for Richer Internet Application Development

Apress®

图灵网站：[www.turingbook.com](http://www.turingbook.com) 热线：(010)51095186

反馈/投稿/推荐信箱：[contact@turingbook.com](mailto:contact@turingbook.com)

有奖勘误：[debug@turingbook.com](mailto:debug@turingbook.com)

分类建议 计算机/Web开发

人民邮电出版社网址：[www.ptpress.com.cn](http://www.ptpress.com.cn)

ISBN 978-7-115-24487-1



9 787115 244871 >

ISBN 978-7-115-24487-1

定价：45.00元