HZ BOOKS 华章科技

Apress®

HTML5 WebSocket 权威指南

[美] Vanessa Wang　Frank Salim　Peter Moskovits　著
姚军　等译

The Definitive Guide to HTML5 WebSocket

- WebSocket领域最全面和系统的著作，三位资深HTML5技术专家共同撰写
- 系统讲解WebSocket 的API、协议、消息传递、安全性和企业部署，并给出通过WebSocket协议通信的真实示例，可操作性强

机械工业出版社
China Machine Press

华章程序员书库

HTML5 WebSocket 权威指南

The Definitive Guide to HTML5 WebSocket

Vanessa Wang
（美） Frank Salim 著
Peter Moskovits
姚军 等译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

HTML5 WebSocket 权威指南 /（美）王（Wang, V.），（美）萨利姆（Salim, F.），（美）莫斯科维茨（Moskovits, P.）著；姚军等译 . —北京：机械工业出版社，2014.1
（华章程序员书库）
书名原文：The Definitive Guide to HTML5 WebSocket

ISBN 978-7-111-45641-4

Ⅰ. H… Ⅱ. ①王… ②萨… ③莫… ④姚… Ⅲ. 超文本标记语言 – 程序设计
Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2014）第 020324 号

本书版权登记号：图字：01-2013-9162

本书是 HTML5 WebSocket 领域最权威的著作之一，它系统、全面地讲解了 HTML5 WebSocket 的各个方面，是 Web 开发人员和架构师学习 WebSocket 的最佳选择。书中讨论了基于 WebSocket 的架构师如何减少不必要的网络开销和延迟层，如何通过 WebSocket 对广泛使用的协议（如 XMPP 和 STOMP）进行分层，如何保护 WebSocket 连接和在企业部署基于 WebSocket 的应用程序。主要内容包括：WebSocket API 和协议、WebSocket 协议通信的例子、WebSocket 的安全性和企业部署、内置即时通信和聊天应用程序的 WebSocket 与 XMPP、通过 WebSocket 的 STOMP 实现发布 / 订阅消息传递协议，以及用远程帧缓冲协议实现 VNC。

机械工业出版社（北京市西城区百万庄大街 22 号　　邮政编码　100037）
责任编辑：罗词亮
印刷
2014 年 3 月第 1 版第 1 次印刷
147mm × 210mm • 6.5 印张
标准书号：ISBN 978-7-111-45641-4
定　　价：49.00 元

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

译 者 序

万维网极大地改变了人类获取信息的方式。当今世界，不管是办公室里埋头工作的人们，还是路上行色匆匆的芸芸众生，都无时无刻不在通过面前的电脑、平板电脑甚至手机乐此不疲地获取信息、处理业务，而仅仅在 20 年之前，人们还从未想过有这番景象。

需求决定市场，人们在 Web 技术上投入了巨大的精力，从最初各种动态网页的渲染技术、令人头痛的跨浏览器兼容问题，到更具有类似桌面应用程序灵敏性的 AJAX 等，不一而足。在经历了大量艰辛的尝试之后，标准也在不断地发展，所有开发者和设计者都意识到，只有标准支持的技术才具备强大的生命力，也才能够真正得到广泛应用。

万维网的核心——HTML 也在不断地变化，从当初仅用于简单文档共享的超文本标记语言，发展到当今令人耳目一新，充实了更丰富语义的 HTML5。但是，HTTP 协议的局限性一直困扰着 Web 开发人员，阻碍他们开发媲美桌面应用程序的实时桌面应用程序，HTTP 无状态、半双工的特性使得它在许多方面显得力不从心，而为此开发的 Comet 等技术也无法完全提供和桌面上基于 TCP 协议的 Web 应用程序类似的体验。

WebSocket 的推出彻底扭转了这种局面，这种新技术基于标准、与语言无关，且能在其上灵活地增加标准协议层次，从而在 HTTP 架构中增加了一个全新的传输层。一经发布，Web 开发人员对之无不争先恐后，了解这种技术也成了所有从事 Web 工作的开

发人员、网络管理员的必修课。

本书用简单扼要的阐述、典型实用的例子，为读者提供了实施 WebSocket 技术所需要的主要基础知识，涵盖了对新旧 HTTP 架构的对比，在 WebSocket 上实施应用协议层次，WebSocket 应用程序部署以及安全性等丰富内容。可以毫不夸张地说，对于初涉这一领域的读者来说，通读本书，除了掌握相关的知识之外，还会对 WebSocket 开发充满信心，并在将来的实践中喜欢上这门技术。

本书的翻译工作主要由姚军完成，徐锋、陈志勇、刘建林、白龙、宁懿等也为翻译工作作出了贡献。由于译者水平所限，书中难免出现一些错误，请广大读者多加批评指正，在此也感谢机械工业出版社华章公司的编辑们对翻译工作的大力支持。

姚军

序

浏览器无疑是当今最流行和最普及的部署平台。实际上，每台电脑、智能手机、平板电脑以及几乎所有其他可以想到的电子产品现在都能执行 JavaScript、显示网页，当然也就能理解 HTTP。这本身就是一个了不起的成就，特别是在你意识到，这一切的发生只不过花费了 10 年多一点的时间时。然而，这还只是开始。昨天的浏览器和现在的完全不同，这要归功于 HTML5 的革新。

HTML5 WebSocket 的重要性再怎么强调都不过分：到现在为止，浏览器只能讲一种语言（HTTP），而这种语言不是为现代的实时 Web 所设计的。是的，我们用长轮询和 Flash socket 等过渡解决方案取得了进展，但是这些解决方案的复杂性和成本限制了我们的能力。WebSocket 改变了一切：它是全新设计、与数据无关（二进制和文本）、全双工的，在字节数和延迟上都为最小化开销作了优化。

WebSocket 是 Web 浏览器所用的 TCP，且具有更健壮、更易用的 API。突然之间，我们的客户可以在浏览器里直接使用任何网络协议，打开了一个全新的世界。一个 XMPP 聊天客户端？容易。需要将你的客户连接到在现有网络上部署的自定义二进制协议？没问题！更了不起的是，你可以在浏览器中使用你已经了解、喜爱和使用的 Web 创作工具在浏览器中直接编写脚本、设计样式和部署客户端。

昨日的浏览器与 HTTP 服务器通信。利用 WebSocket，浏览器可以与任何对象通信，实现任何协议：完成 HTTP 握手、升级连

接，都可以轻松做到。我们不再谈论建立稍好一点或者更有交互性的页面。有了 WebSocket，我们可以建立能够直接在用户的浏览器中交付的全新应用类型和体验。

本书对 WebSocket 的概念及其所能解决的问题进行了全面的讨论，并且提供了许多可以直接使用的实用示例。你将会惊喜地发现，使用 WebSocket 是如此容易，用很少的代码可以实现如此之多的功能。使用 WebSocket 是快乐的经历，而 Vanessa、Frank 和 Peter 的这本书就是合适的指南。尽情品读吧！

Ilya Grigorik

Google“建立快速 Web”项目的开发大使

致　　谢

衷心感谢 Peter Lubbers（Pro HTML5 Programming, 2nd ed.），他的指导和对 HTML5 的热心使本书成为可能。还要感谢 Ilya Grigorik，他对 Web 性能和实时技术的热情非常鼓舞人心。感谢 Steve Atkinson 和 Frank Greco 不知疲倦、夜以继日地为我们提供深刻的反馈意见。感谢 Jeff Mesnil 和 Dhurv Matani 提供的杰出代码（可以在 GitHub 找到他们），这为本书带来了前沿示例。特别感谢 Kaazing 公司和 Apress 出版社的支持，它们为我们提供了机会和大家分享对 WebSocket 的热情。

——Vanessa Wang，Frank Salim，Peter Moskovits

我要将最深切的感谢献给 Julian，感谢他的鼓励、支持和无尽的耐心。感谢了不起的 Camper 和 Tilson，还要感谢 Pins 在深夜和清晨陪伴着我。特别感谢我的家人为帮助我完成本书所作出的牺牲。

最后，将感谢和敬意献给我杰出的合著者和朋友们——出色、有创意、热情的 Frank 和 Peter。

——Vanessa Wang

我要衷心地感谢家人的支持，过去对你们的感谢还远远不够。我还要感谢 April 出色的建议和巨大的耐心。当然，我还要感谢合著者 Vanessa 和 Peter。

——Frank Salim

谢谢你，Anna，感谢你在本书编写过程中极大的支持和理解以及更多。我要感谢可爱的 Danka 和 Lea，他们非常体谅我，在我不得不坐下来工作时，他们没有伤心难过。特别要感谢 Aniko 不知疲倦的帮助。

最后但同样重要的是，很高兴能和两位合著者 Vanessa 和 Frank 一起工作，他们都是了不起的同行和朋友。感谢你们向我提供了这次机会，我享受合作中的每一个瞬间。

——Peter Moskovits

目　　录

第 1 章

HTML5 WebSocket 简介

本书是为所有想要学习如何构建实时 Web 应用程序的人而写的。你可能会对自己说：“我已经这样做了！”或者问：“这到底是什么意思？”让我们来澄清一下：本书将说明如何使用革命性和广受支持的新型开放行业标准技术——WebSocket 来构建真正的实时 Web 应用程序。这种技术能够在你的客户端应用程序和 Web 上的远程服务器之间实现全双工的双向通信——不需要插件！

仍然不得要领吗？几年以前我们也是这样，当时我们还没有开始使用 HTML5 WebSocket。在这本指南中，我们将解释你需要知道的 WebSocket 相关知识，以及在今天考虑使用 WebSocket 的原因。我们将告诉你如何在 Web 应用程序中实现一个 WebSocket 客户端，创建自己的 WebSocket 服务器，用 WebSocket 和更高级协议（如 XMPP 和 STOMP）协同工作，加强客户端与服务器之间的通信安全性，以及部署基于 WebSocket 的应用程序。最后，我们将解释，为什么你应该考虑现在就开始使用 WebSocket。

1.1 HTML5 是什么

首先，我们来解释“HTML5 WebSocket”中的“HTML5”部分。如果你已经是 HTML5 专家，比如说阅读过《Pro HTML5 Programming》，并且已经开发了现代化的响应式 Web 应用程序，可以跳过本节继续阅读后面的章节。但是，如果你是 HTML5 新手，下面的简单介绍适合你。

HTML 最初用于在 Internet 上共享的静态、基于文本的文档。随着时间的推移，Web 用户和设计师希望 HTML 文档更有交互性，他们开始添加表单功能和早期的“门户”类功能，以改进文档。现在，这些静态的文档集合（即网站）更像 Web 应用程序，根据富客户 / 服务器的桌面应用程序原则构建。这些 Web 应用程序可用于几乎所有设备：笔记本电脑、智能手机、平板电脑——所有上网设备。

HTML5 用于使富 Web 应用程序的开发更加简单、更加自然、更加符合逻辑，开发人员可以设计和构建一次，在任何地方部署。

HTML5 提供了 Web 应用程序的易用性，因为它不需要插件。通过 HTML5，你现在可以使用 <header> 这样的语义标记语言来代替 。多媒体也更容易编码，可以用 <audio> 和 <video> 等标签（tag）插入和指定合适的媒体类型。此外，通过语义化标签，HTML5 的可访问性更好，因为读屏器可以更容易地阅读标签。

HTML5 这一术语涵盖了 Web 技术中发生的大量改进和变化，包括了从用于网页的标记到 CSS3 样式、离线与存储、多媒体、连接性等一切变化。图 1-1 展示了不同的 HTML5 特性领域。

图 1-1　HTML5 特性领域（W3C，2011）

有许多资源深入研究 HTML5 的这些领域。在本书中，我们专注于连接性领域，也就是 WebSocket API 和协议。我们先来看看 HTML5 连接性的历史。

1.2　HTML5 连接性

HTML5 连接性领域包括 WebSocket、服务器发送事件和跨文档消息传递（Cross-Document Messaging）等技术。这些包含在 HTML5 规范中的 API 有助于简化浏览器所受限制使应用程序开发人员无法创建所需丰富功能，或者 Web 应用程序开发过于复杂的情况。HTML5 中的跨文档消息传递就是这种简化的一个例子。

在 HTML5 之前，浏览器窗口和框架之间的通信由于安全的原因而受到限制。然而，随着 Web 应用程序开始组合不同网站中的

内容和应用程序，这些应用程序的相互通信变得必不可少。为了解决这个问题，标准组织和主要浏览器供应商同意支持跨文档消息传递，后者能够确保在浏览器窗口、选项卡（tab）和 iFrame 之间跨源通信的安全。跨文档消息传递定义了 postMessage API，作为发送和接收消息的标准手段。利用来自不同主机和域的内容、在浏览器内部通信的用例有许多，例如地图、聊天和社会化网络。跨文档消息传递提供了不同 JavaScript 上下文之间的异步消息传递。

跨文档消息传递的 HTML5 规范还通过引入源（origin）的概念，澄清并提升了域安全性，这一概念由方案（scheme）、主机（host）和端口（port）来定义。根本上，两个 URI 当且仅当有相同的方案、主机和端口时才被认为是同源的。在源值中不考虑路径。

下面是方案、主机和端口不匹配（因而是不同源）的例子：

- https://www.example.com 和 http://www.example.com
- http://www.example.com 和 http://example.com
- http://example.com:8080 和 http://example.com:8081

下面的例子中 URL 是同源的：

http://www.example.com/page1.html 和 http://www.example.com/page2.html

跨文档消息传递通过允许消息在不同源之间交换，克服了同源策略的限制。当你发送消息时，发送者指定接收者的源；当你接收消息时，发送者的源会被作为消息的一部分。消息的源由浏览器提供，不会被伪造。在接收者一端，你可以决定处理哪些消息，忽略哪些消息。你还可以保留一个“白名单”，只处理来自具有信任源的文档的消息。

跨文档消息传递是 HTML5 规范利用非常强大的 API 简化 Web 应用程序之间通信的绝佳范例。但是，它的重点被限制在跨窗口、选项卡和 iFrame 的通信上。它不能解决协议通信中正在变得越来越严重的复杂性，这时我们就要求助 WebSocket。

HTML5 规范的主要编写者 Ian Hickson 在 HTML5 规范的通信部分中添加了我们今天所说的 WebSocket。WebSocket 最初称

作 TCPConnection，现已发展为一个独立的规范。虽然 WebSocket 目前存在于 HTML5 的领域之外，但是它对于实现现代（基于 HTML5 的）Web 应用程序中的实时连接性至关重要。那么，为什么 WebSocket 对今天的 Web 如此有意义呢？我们首先来看看旧的 HTTP 架构，在这种架构中协议通信非常重要。

1.3　旧的 HTTP 架构概览

为了理解 WebSocket 的重要性，我们首先来看看旧的架构，具体地说就是使用 HTTP 的架构。

1.3.1　HTTP 101（即 HTTP/1.0 和 HTTP/1.1）

在旧的架构中，连接性由 HTTP/1.0 和 HTTP/1.1 处理。HTTP 是客户端 / 服务器模式中请求 – 响应所用的协议，在这种模式中，客户端（一般是 Web 浏览器）向服务器提交 HTTP 请求，服务器响应请求的资源（例如 HTML 页面）和关于页面的附加信息。HTTP 也用来读取文档；HTTP/1.0 对于从服务器请求单个文档来说已经足够。但是，随着 Web 的成长超出了简单的文档共享，并开始包含更多的交互性，连接性需要进行微调，以缩短浏览器请求和服务器响应之间的时间。

在 HTTP/1.0 中，每个服务器请求需要一个单独的连接，这种方法至少可以说没有太好的伸缩性（scalability）。HTTP 的下一个修订版本 HTTP/1.1 增加了可重用连接。由于可重用连接的推出，浏览器可以初始化一个到 Web 服务器的连接，以读取 HTML 页面，然后重用该连接读取图片、脚本等资源。HTTP/1.1 通过减少客户端到服务器的连接数量，降低了请求的延迟。

HTTP 是无状态的，也就是说，它将每个请求当成唯一和独立的。无状态协议具有一些优势，例如，服务器不需要保存有关会话的信息，从而不需要存储数据。但是，这也意味着在每次 HTTP 请求和响应中都会发送关于请求的冗余信息。

我们来看看客户端对服务器请求的一个例子。代码清单 1-1 展

示了包含多个 HTTP 首标的完整 HTTP 请求。

代码清单 1-1　客户端对服务器请求的 HTTP/1.1 首标

```
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://localhost:8080/PollingStock/
Cookie: showInheritedConstant=false; showInheritedProtectedConst
ant=false; showInheritedProperty=false; showInheritedProtectedPr
operty=false; showInheritedMethod=false; showInheritedProtectedM
ethod=false; showInheritedEvent=false; showInheritedStyle=false;
showInheritedEffect=false;
```

代码清单 1-2 展示了服务器对客户端响应的一个示例。

代码清单 1-2　服务器对客户端响应的 HTTP1.1 首标

```
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: 321
Date: Wed, 06 Dec 2012 00:32:46 GMT
```

在代码清单 1-1 和代码清单 1-2 中，仅仅首标信息的开销就有 871 字节（这还没有实际的数据）。这两个例子说明，不管服务器是否向客户端发送实际的数据或者信息，请求的首标信息都必须在两个方向上传输：从客户端到服务器和从服务器到客户端。

在 HTTP/1.0 和 HTTP/1.1 中，低效的根源主要是：

- HTTP 用于文档共享，而不是丰富的交互性应用程序，我们在桌面上习以为常的这种应用程序现在已经进入 Web；
- 随着客户端和服务器之间交互的增加，HTTP 协议在客户端和服务器之间通信所需要的信息量快速增加。

从根本上讲，HTTP 还是*半双工*的协议，也就是说，在同一时刻流量只能单向流动：客户端向服务器发送请求（单向），然后服务器响应请求（单向）。半双工的效率很低。想象一下这样的电话

交谈：当你想要交流时，你必须按下一个按钮，说话，再按下另一个按钮完成。同时，你的通话对象必须耐心等待你讲完，按下按钮，最后回复。是不是很熟悉？我们儿时小规模地进行过这种形式的通信，美国军队始终采用这种方式：步话机。虽然步话机有绝对的好处和很好的应用，但是并不总是最有效的通信形式。

工程师们多年来一直致力于解决这个问题，他们使用各种著名的方法：轮询、长轮询和 HTTP 流化（streaming）。

1.3.2　绕道而行：HTTP 轮询、长轮询和流化

通常，浏览器访问网页时，会向页面所在的服务器发送一个 HTTP 请求。Web 服务器确认请求并向浏览器发回响应。在许多情况下，返回的信息（如股价、新闻、交通图、医疗设备读数和天气信息）到达浏览器显示页面时已经过时。如果用户需要得到最新的实时信息，他们可以不断刷新页面，但是这显然并不实际，也不是特别精妙的解决方案。

当前对提供实时 Web 应用程序的尝试多半围绕“轮询”（polling）技术进行，这种技术模拟其他服务器端“推”技术（最流行的是 Comet），本质上就是推迟完成 HTTP 响应，向客户端提交信息。

轮询是一种定时的同步调用，客户端向服务器发送请求查看是否有可用的新信息。请求以固定的时间间隔发出，不管是否有信息，客户端都会得到响应：如果有可用信息，服务器发送这些信息；如果没有可用信息，服务器返回一个拒绝响应，客户端关闭连接。

如果你知道信息交付的精确间隔，轮询就是一个好的解决方案，因为你可以同步客户端，只在你知道服务器上有可用信息的时候发送请求。然而，实时数据并不总是可以预测的，发出不必要的请求、因而打开过多连接是不可避免的。结果是，在低信息率的情况下，你可能打开或者关闭许多不必要的连接。

长轮询（long polling）是另一种流行的通信方法，客户端向服务器请求信息，并在设定的时间段内打开一个连接。服务器如

果没有任何信息，会保持请求打开，直到有客户端可用的信息，或者直到指定的超时时间用完为止。这时，客户端重新向服务器请求信息。长轮询也称作 Comet（前面已经提到过）或者反向 AJAX。Comet 延长 HTTP 响应的完成，直到服务器有需要发送给客户端的内容，这种技术常常称作“挂起 GET”或“搁置 POST”。重要的是要知道，当信息量很大时，长轮询相对于传统轮询并没有明显的性能优势，因为客户端必须频繁地重连到服务器以读取新信息，造成网络的表现和快速轮询相同。长轮询的另一个问题是缺乏标准实现。

在流化技术中，客户端发送一个请求，服务器发送并维护一个持续更新和保持打开（可以是无限或者规定的时间段）的开放响应。每当服务器有需要交付给客户端的信息时，它就更新响应。看起来，流化是能够适应不可预测的信息交付的极佳方案，但是服务器从不发出完成 HTTP 响应的请求，从而使连接一直保持打开。在这种情况下，代理和防火墙可能缓存响应，导致信息交付的延迟增加。因此，许多流化的尝试对于存在防火墙和代理的网络是不友好的。

上述方法提供了近乎实时的通信，但是它们也涉及 HTTP 请求和响应首标，包含了许多附加和不必要的首标数据与延迟。此外，在每一种情况下，客户端都必须等待请求返回，才能发出后续的请求，而这显著地增加了延迟。

图 1-2 展示了 Web 上这些连接的半双工特性，它们整合到架构中，其中在内联网中具有通过 TCP 的全双工连接。

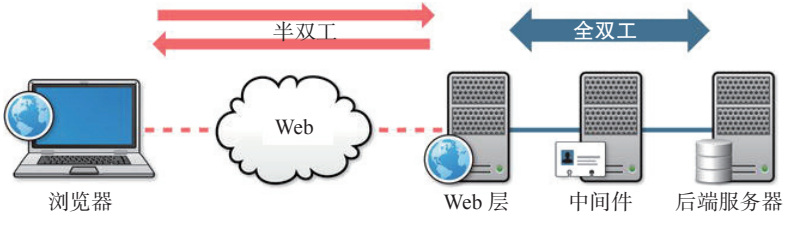

图 1-2　在 Web 上是半双工；在后端是通过 TCP 的全双工

1.4　WebSocket 入门

那么，我们该往何处去？为了消除这些问题，HTML5 规范的连接性部分包含了 WebSocket。WebSocket 是一种自然的全双工、双向、单套接字连接。使用 WebSocket，你的 HTTP 请求变成打开 WebSocket 连接（WebSocket 或者 WebSocket over TLS（Transport Layer Security，传输层安全性，原称“SSL”））的单一请求，并且重用从客户端到服务器以及服务器到客户端的同一连接。

WebSocket 减少了延迟，因为一旦建立起 WebSocket 连接，服务器可以在消息可用时发送它们。例如，和轮询不同，WebSocket 只发出一个请求。服务器不需要等待来自客户端的请求。相似地，客户端可以在任何时候向服务器发送消息。相比轮询不管是否有可用消息，每隔一段时间都发送一个请求，单一请求大大减少了延迟。

图 1-3 比较了轮询和 WebSocket 方案。

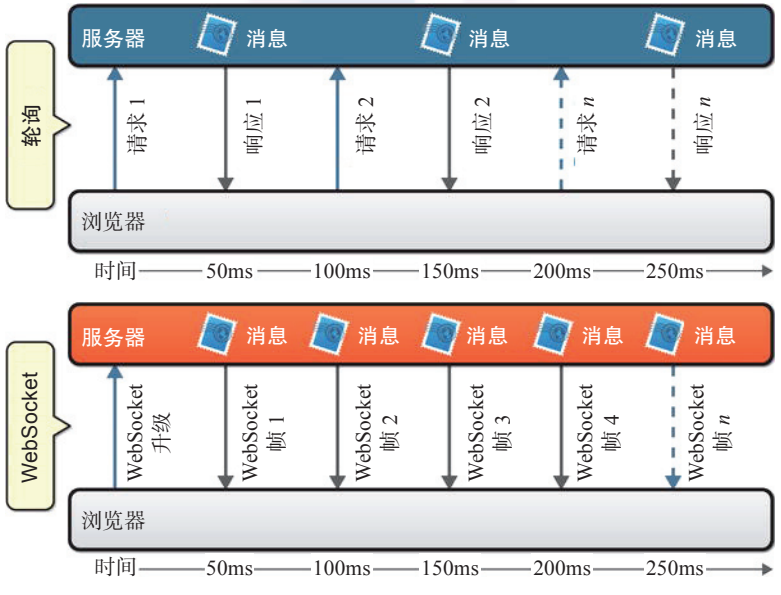

图 1-3　轮询与 WebSocket

本质上，WebSocket 和 HTML5 语义及简单化范式融为一体。它不仅消除了复杂的变通方法需求以及延时，而且简化了架构。接下来我们更深入地探讨这方面的理由。

1.5 为什么需要 WebSocket

我们已经研究了 WebSocket 的历史，现在再来看看使用 WebSocket 的一些理由。

1.5.1 WebSocket 与性能相关

WebSocket 使实时通信更加有效。

你总是可以在 HTTP 上使用轮询（有时候甚至是流化），通过 HTTP 接收通知。然而，WebSocket 能节约带宽、CPU 资源并减少延迟。

WebSocket 是性能上的一个革新。

1.5.2 WebSocket 与简洁性相关

WebSocket 使 Web 上客户端和服务器之间的通信变得更加简单。

用 WebSocket 之前的架构建立过实时通信的人们就会知道，HTTP 上的实时通知技术过于复杂。在无状态的请求之间维护会话状态更增加了复杂度。跨源 AJAX 十分难以理解，用 AJAX 处理有序的请求需要特殊考虑，而 AJAX 通信也很复杂。将 HTTP 扩展到它不适用的用例的每个尝试都会增加软件的复杂度。

WebSocket 可以显著简化实时应用程序中面向连接的通信。

1.5.3 WebSocket 与标准相关

WebSocket 是一个低层网络协议，你可以在它的基础上构建其他标准协议。

许多 Web 应用程序实际上很庞大。大部分 AJAX 应用程序通常都包含了紧密耦合的客户端和服务器组件。WebSocket 自然支持高层应用协议的概念，你可以更加灵活地独立发展客户端和服务器。支持较高层的协议使模块化成为可能，鼓励开发可重用组件。

例如，你可以使用相同的 XMPP over Websocket 客户端登录不同的聊天服务器，因为所有 XMPP 服务器都理解相同的标准协议。

Websocket 是可互操作 Web 应用程序的一个革新。

1.5.4　WebSocket 与 HTML5 相关

WebSocket 是为 HTML5 应用程序提供高级功能，以便与其他平台竞争所作努力的一部分。

每种操作系统都需要网络功能。应用程序打开套接字以及与其他主机通信的能力是每个主要平台提供的核心特性。HTML5 在许多方面倾向于使浏览器成为一个与操作系统相仿的全功能应用程序平台。低级网络 API（如套接字）无法处理源安全模型或者 Web 的 API 设计风格。WebSocket 为 HTML5 应用程序提供了 TCP 风格的网络，没有破坏浏览器安全性，而且有一个现代的 API。

WebSocket 是 HTML5 平台的关键组件，也是开发人员的强大工具。

1.5.5　你需要 WebSocket

简单地说，你需要 WebSocket 来构建世界级的 Web 应用程序。WebSocket 解决了使 HTTP 不适合于实时通信的主要不足之处。WebSocket 的异步、双向通信模式是对 Internet 上的传输层协议提供的总体灵活性的回报。

想想 WebSocket 的各种出色应用，以及在应用程序中构建真正的实时功能，例如聊天、协作文档编辑、大型多人在线游戏（Massively Multiplayer Online, MMO）、股票交易应用程序等。我们将在本书后面介绍具体的应用程序。

1.6　WebSocket 和 RFC 6455

WebSocket 是一个协议，但是还有一个 WebSocket API，应用程序可以用它控制 WebSocket 协议，响应服务器触发的事件。这个 API 由 W3C（World Wide Web Consortium，万维网联盟）开发，

而协议由 IETF（Internet Engineering Task Force，互联网工程任务组）开发。WebSocket API 现在得到现代浏览器的支持，包含了使用全双工、双向 WebSocket 连接所需的方法和特性（attribute）。利用这个 API，你可以执行必要的操作，例如打开和关闭连接、发送和接收消息、监听服务器触发的事件。第 2 章将更加详细地描述这个 API，并提供使用它的例子。

WebSocket 协议能够通过 Web 进行客户端和远程服务器之间的全双工通信，并支持二进制数据和文本字符串的传输。这个协议由开始的握手和之后的基本消息框架组成，在 TCP 上添加层次。第 3 章将更加详细描述协议，并说明如何创建自己的 WebSocket 服务器。

1.7 WebSocket 的世界

WebSocket API 和协议都有十分繁荣的社区，这通过各种 WebSocket 服务器选择、开发人员社区和大量广泛使用的 WebSocket 应用程序得到了反映。

1.8 WebSocket 的选择

WebSocket 服务器的实现多种多样，例如 Apache mod_pywebsocket、Jetty、Socket.IO 和 Kaazing 的 WebSocket Gateway。

写作本书的想法来源于我们渴望分享在 Kaazing 多年使用 WebSocket 及相关技术中得到的知识、经验和意见。Kazzing 花费 5 年多的时间，构建了一个企业级的 WebSocket 网关服务器及其客户端程序库。

1.8.1 非常活跃的 WebSocket 社区

我们已经列出了使用 WebSocket 的一些原因，并将探索自己动手实现 WebSocket 的实用示例。除了各种可用的 WebSocket 服务器之外，WebSocket 社区也十分兴旺，特别是在 HTML5 游戏、企业级消息传递和在线聊天方面。每天都有许多会议和编码研讨

会，不仅讨论 HTML5 的特定领域，而且讨论实时通信方法，尤其是 WebSocket。连许多构建广为使用的企业级消息服务的公司也将 WebSocket 集成到它们的系统中。因为 WebSocket 是基于标准的，所以很容易改进现有架构，标准化和扩展实现，也很容易构建过去不可能或者难以建立的新服务。

WebSocket 的激动人心之处还反映在 GitHub 等在线社区上，每天社区中都会创建更多的 WebSocket 相关服务器、应用程序和项目。其他兴旺的社区有 http://www.websocket.org/，我们在后续章节中将使用它提供的一个 WebSocket 服务器示例，还有 http://webplatform.org 和 http://html5rocks.com，它们都是鼓励分享所有与 HTML5（包括 WebSocket）相关信息的开放社区。

说明　附录 B 中列出了更多的 WebSocket 服务器。

1.8.2　WebSocket 应用程序

在写作本书时，WebSocket 正被用在各种各样的应用程序中。有些应用程序可能使用了以前的“实时”通信技术（如 AJAX），但是它们已经显著地改进了性能。外币兑换和股票报价应用程序也从 WebSocket 提供的降低带宽需求和全双工连接特性中得到了好处。第 3 章将研究如何检查 WebSocket 流量。

随着部署到浏览器的应用程序的增加，HTML5 游戏的开发也有了迅猛的发展。WebSocket 从本质上很适合于 Web 游戏，因为游戏及其交互很依赖响应能力。使用 WebSocket 的 HTML5 游戏实例包括在线赌博应用程序、集成 WebGL over WebSocket 的游戏控制器应用程序以及游戏中的在线聊天。还有一些很有趣的大型多人在线游戏（MMO）广泛地用于各种移动及桌面设备的浏览器中。

1.9　相关技术

你可能吃惊地发现，有一些技术可以和 WebSocket 结合使用或者代替 WebSocket。下面是其他一些新兴的 Web 通信技术。

1.9.1 服务器发送事件

如果架构需要双向全双工通信，那么 WebSocket 是个好的选择。然而，如果你的服务主要向其客户端广播或者推送信息，而不需要任何交互（如新闻摘要、天气预报等），那么使用服务器发送事件（Server-Sent Event，SSE）提供的 EventSource API 是个好的选择。SSE 是 HTML5 规范的一部分，加强了某些 Comet 技术。可以将 SSE 当作一种 HTTP 轮询、长轮询和流化的公用可互操作语法使用。利用 SSE，你可以得到自动重连、事件 ID 等功能。

说明 尽管 WebSocket 和 SSE 连接都以 HTTP 请求开始，但是你所看到的性能优势和功能可能有很大的不同。例如，SSE 不能从客户端向服务器上传流化数据，且只支持文本数据。

1.9.2 SPDY

SPDY（音同“Speedy”）是 Google 开发的一种网络协议，得到越来越多的浏览器支持，包括 Google Chrome、Opera 和 Mozilla Firefox。本质上，SPDY 扩充了 HTTP，通过压缩 HTTP 首标和多路复用（multiplexing）等手段改进 HTTP 请求性能。它的主要目的是改进网页的性能。WebSocket 的重点是改进 Web 应用程序前端和服务器之间的通信，而 SPDY 优化的是应用程序内容和静态页面的交付。HTTP 和 WebSocket 之间的不同是架构性的，而不是增量的。SPDY 是 HTTP 的修改形式，所以它有与 HTML 相同的架构风格和语义。它改正了许多 HTTP 的非本质问题，添加了多路复用、工作管道（working pipling）和其他有用的改进。WebSocket 去除了请求 – 响应风格的通信并启用实时交互和替代的架构模式。

WebSocket 和 SPDY 是相互补充的，你可以将 SPDY 扩充的 HTTP 连接升级为 WebSocket，从而在 SPDY 上使用 WebSocket，从两个领域获得利益。

1.9.3　Web 实时通信

Web 实时通信（Web Real-Time Communication，WebRTC）是增强现代 Web 浏览器通信能力的另一种努力。WebRTC 是 Web 的点对点技术。浏览器可以直接通信，而不需要通过服务器传输所有的数据。WebRTC 包含可以让浏览器相互之间实时通信的 API。在写作本书时，WebRTC 由万维网联盟（W3C）草拟，仍处于草案阶段，可以在 http://www.w3.org/TR/webrtc/ 上找到。

WebRTC 的第一批应用是实时语音和视频聊天。WebRTC 对于媒体应用程序已经是具有吸引力的新技术，网上有许多示例应用程序，你可以用它们测试 Web 上的视频和音频传输。

WebRTC 以后将会添加数据通道。这些数据通道计划使用和 WebSocket 类似的 API 以保持一致性。此外，如果你的应用程序使用了流媒体和其他数据，可以结合使用 WebRTC 和 WebSocket。

1.10　小结

本章介绍了 HTML5 和 WebSocket，以及一些 HTTP 演变到 WebSocket 的历史。我们希望，你现在已经和我们一样兴奋地学习 WebSocket，研究代码，并且梦想着用它所能实现的宏伟目标。

后续章节将更深入地研究 WebSocket API 和协议，说明如何将 WebSocket 与标准的、更高级的应用程序协议相结合，讨论 WebSocket 的安全性，并描述企业级特性和部署。

第 2 章

WebSocket API

本章介绍 WebSocket 应用程序编程接口（API），你可以用它来控制 WebSocket 协议，创建 WebSocket 应用程序。我们研究 WebSocket API 的各个组成部分，包括事件、方法和特性。为了学习使用 API，我们写了一个简单的客户端应用程序。将其连接到现有的公共服务器（http://websocket.org），这样我们就可以通过 WebSocket 发送和接收消息。通过使用现有的服务器，我们可以将重点放在学习用于创建 WebSocket 应用程序的易用 API 上。我们还将循序渐进地说明如何使用 WebSocket API 驱动使用二进制数据的 HTML5 媒体。最后，我们将讨论浏览器支持和连接性。

本章重点研究 WebSocket 的客户端应用程序方面，你可以将 WebSocket 协议扩展到自己的 Web 应用程序中。后续章节将更加详细地描述 WebSocket 协议，以及在具体环境中使用 WebSocket 的方法。

2.1　WebSocket API 概览

正如第 1 章所提到的，WebSocket 由网络协议和帮助建立客户端应用程序与服务器之间 WebSocket 连接的 API 组成。第 3 章将更加详细地描述 WebSocket 协议，现在先来看看 API。

WebSocket API 是一个接口，使应用程序能使用 WebSocket 协议。通过在应用程序中使用这个 API，可以控制应用程序用于发送和接收消息的全双工通信信道。WebSocket 接口非常简单易用。要连接到远程主机，只要创建一个新的 WebSocket 对象实例，并为新对象提供一个代表所要连接端口的 URL 就可以了。

WebSocket 连接通过在客户端和服务器之间第一次握手时将 HTTP 协议升级到 WebSocket 协议来完成，这一工作在相同的底层 TCP 连接上进行。一旦建立，WebSocket 消息可以在 WebSocket 接口定义的方法之间来回传送。在应用程序代码中，可以使用异步事件监听器处理连接生命周期的每个阶段。

WebSocket API 完全是（真正的）事件驱动的。一旦建立全双工连接，当服务器有需要发送到客户端的数据，或者你所关心的

资源将要改变状态时，它会自动发送数据或者通知。有了事件驱动的 API，你就不需要轮询服务器以得到目标资源的最新状态，客户端只要监听需要的通知和更改就行了。

在后面几章讨论更高层协议（如 STOMP 和 XMPP）时，我们将看到使用 WebSocket API 的不同示例。但是现在，我们要更仔细地研究 API。

2.2 WebSocket API 入门

WebSocket API 使你可以通过 Web，在客户端应用程序和服务器端进程之间建立全双工的双向通信。WebSocket 接口规定了可用于客户端的方法以及客户端与网络的交互方式。

首先，你要调用 WebSocket 构造函数（constructor），创建一个 WebSocket 连接。构造函数返回 WebSocket 对象实例。你可以监听该对象上的事件，这些事件告诉你何时连接打开，何时消息到达，何时连接关闭以及何时发生错误。你可以与 WebSocket 对象交互，发送消息或者关闭连接。下面来研究 WebSocket API 的各个方面。

2.2.1 WebSocket 构造函数

为了建立到服务器的 WebSocket 连接，使用 WebSocket 接口，通过指向一个代表所要连接端点的 URL，实例化一个 WebSocket 对象。WebSocket 协议定义了两种 URL 方案（URL scheme）——ws 和 wss，分别用于客户端和服务器之间的非加密与加密流量。ws（WebSocket）方案与 HTTP URI 方案类似。wss（WebSocket Secure，WebSocket 安全）URI 方案表示使用传输层安全性（TLS，也叫 SSL）的 WebSocket 连接，使用 HTTPS 采用的安全机制来保证 HTTP 连接的安全。

说明 第 7 章将详细讨论 WebSocket 安全性。

WebSocket 构造函数有一个必需的参数 URL（指向连接目标的 URL）和一个可选参数 protocols（为了建立连接，服务器必须在其响应中包含的一个或一组协议名称）。在 protocols 参数中可以使用的协议包括 XMPP（eXtensible Messaging and Presence Protocol，可扩展消息处理现场协议）、SOAP（Simple Object Access Protocol，简单对象访问协议）或者自定义协议。

代码清单 2-1 展示了 WebSocket 构造函数的必需参数，它必须是以 ws:// 或者 ws:// 开始的一个完全限定的 URL。在这个例子中，完全限定 URL 是 ws://www.websocket.org。如果 URL 有语法错误，构造函数将抛出异常。

代码清单 2-1　WebSocket 构造函数示例

```
// Create new WebSocket connection

var ws = new WebSocket("ws://www.websocket.org");
```

连接到 WebSocket 服务器时，可以选择使用第二个参数列出应用程序支持的协议，用于协议协商。

为了确保客户端和服务器发送与接收双方都能理解的消息，它们必须使用相同的协议。WebSocket 构造函数允许你定义客户端用于与服务器通信的协议。服务器反过来选择使用的协议（在客户端和服务器之间只能使用一种协议）。这些协议在 WebSocket 协议之上使用。WebSocket 的最大好处之一是能在 WebSocket 上建立广泛使用的协议层次（将在第 3 章 ~ 第 6 章介绍），使你可以完成许多出色的工作，例如将传统的桌面应用程序带到 Web 中。

说明　WebSocket 协议（RFC 6455）将与 WebSocket 一起使用的协议称作“子协议”，尽管这些协议是更高级、结构完整的协议。在本书中，我们一般简单地将和 WebSocket 一起使用的协议称作“协议”，以免引起混淆。

我们离题太远了，现在回到 API 中的 WebSocket 构造函数。在第一次 WebSocket 连接握手时（你将在第 3 章中学到更多的内容），客户发送带有协议名称的 Sec-WebSocket-Protocol 首标。服

务器选择 0 个或者 1 个协议，响应一个带有和客户请求相同的协议名称的 Sec-WebSocket-Protocol 首标；否则，服务器关闭连接。

协议协商对于确定 WebSocket 服务器支持的协议及版本很有用。应用程序可能支持多个协议，使用协议协商选择与特定服务器通信的协议。代码清单 2-2 展示了支持假想协议“myProtocol”的 WebSocket 构造函数。

代码清单 2-2　带有协议支持的 WebSocket 构造函数示例

```
// Connecting to the server with one protocol called myProtocol

var ws = new WebSocket("ws://echo.websocket.org", "myProtocol");
```

说明　在代码清单 2-2 中，假想的协议“myProtocol”是一个精心定义、甚至可能是注册和标准化的协议名称，客户端应用程序和服务器都能理解这个协议。

WebSocket 构造函数还可以包含一组客户端支持的协议，让服务器决定使用其中的一个。代码清单 2-3 展示了一个 WebSocket 构造函数的例子，以数组的形式表示其支持的协议列表。

代码清单 2-3　带有协议支持的 WebSocket 构造函数示例

```
// Connecting to the server with multiple protocol choices

var echoSocket = new
WebSocket("ws://echo.websocket.org", ["com.kaazing.echo",
"example.imaginary.protocol"])

echoSocket.onopen = function(e) {
   // Check the protocol chosen by the server
   console.log(echoSocket.protocol);
}
```

在代码清单 2-3 中，由于 WebSocket 服务器 ws://echo.websocket.org 只理解 `com.kaazing.echo` 协议，而不理解 `example.imaginary.protocol`，该服务器在触发 WebSocket open 事件的时候选择 `com.kaazing.echo` 协议。使用数组为你提供了让应用程序对不同服务器使用不同协议的灵活性。

第 3 章将深入讨论 WebSocket 协议，但是实际上，可以在

protocols 参数中指定的协议有三种类型：

- 注册协议：已经根据 RFC 6455（WebSocket 协议），向注册协议的正式管理实体 IANA（Internet Assigned Numbers Authority，互联网编号分配机构）正式注册的标准协议。注册协议的例子之一是 Microsoft 的 SOAP over WebSocket 协议。更多信息参见 http://www.iana.org/assignments/websocket/websocket.xml。
- 开放协议：广泛使用的标准化协议，如 XMPP 和 STOMP，它们尚未注册为正式的标准协议。我们将在后续章节里研究这些协议类型的使用。
- 自定义协议：你自己编写并且和 WebSocket 一起使用的协议。

本章重点关注将 WebSocket API 用于你的自定义协议，后续的章节将研究开放协议的使用。我们首先单独地了解事件、对象和方法，然后将它们组合为一个可行的示例。

2.2.2　WebSocket 事件

WebSocket API 是纯事件驱动的。应用程序代码监听 WebSocket 对象上的事件，以便处理输入数据和连接状态的改变。WebSocket 协议也是事件驱动的。客户端应用程序不需要轮询服务器来得到更新的数据。消息和事件将在服务器发送它们的时候异步到达。

WebSocket 编程遵循异步编程模式，也就是说，只要 WebSocket 连接打开，应用程序就简单地监听事件。客户端不需要主动轮询服务器得到更多的信息。要开始监听事件，只要为 WebSocket 对象添加回调函数。也可以使用 addEventListener() DOM 方法为 WebSocket 对象添加事件监听器。

WebSocket 对象调度 4 个不同的事件：

- open
- message
- error
- close

和所有 Web API 一样，可以用 on< 事件名称 > 处理程序属性

监听这些事件，也可以使用 addEventListener(); 方法。

1. WebSocket 事件：open

一旦服务器响应了 WebSocket 连接请求，open 事件触发并建立一个连接。open 事件对应的回调函数称作 onopen。

代码清单 2-4 说明建立 WebSocket 连接时如何处理该事件。

代码清单 2-4 open 事件处理程序示例

```
// Event handler for the WebSocket connection opening
ws.onopen = function(e) {
   console.log("Connection open...");
};
```

到 open 事件触发时，协议握手已经完成，WebSocket 已经准备好发送和接收数据。如果应用程序接收到一个 open 事件，那么可以确定 WebSocket 服务器成功地处理了连接请求，并且同意与应用程序通信。

2. WebSocket 事件：message

WebSocket 消息包含来自服务器的数据。你也可能听说过组成 WebSocket 消息的 WebSocket 帧（Frame）。第 3 章将详细讨论消息和帧的概念。为了理解消息使用 API 的方式，WebSocket API 只输出完整的消息，而不是 WebSocket 帧。message 事件在接收到消息时触发，对应于该事件的回调函数是 onmessage。

代码清单 2-5 展示了一个接收文本消息并显示消息内容的消息处理程序。

代码清单 2-5 文本消息 message 事件处理程序示例

```
// Event handler for receiving text messages
ws.onmessage = function(e) {
   if(typeof e.data === "string"){
      console.log("String message received", e, e.data);
   } else {
      console.log("Other message received", e, e.data);
   }
};
```

除了文本，WebSocket 消息还可以处理二进制数据，这种数据作为 Blob 消息（见代码清单 2-6）或者 ArrayBuffer 消息处理（见代码清单 2-7）。因为设置 WebSocket 消息二进制数据类型的应用

程序会影响二进制消息，所以必须在读取数据之前决定用于客户端二进制输入数据的类型。

代码清单 2-6　Blob 消息 message 事件处理程序示例

```
// Set binaryType to blob (Blob is the default.)
ws.binaryType = "blob";

// Event handler for receiving Blob messages
ws.onmessage = function(e) {
   if(e.data instanceof Blob){
      console.log("Blob message received", e.data);
      var blob = new Blob(e.data);
   }
};
```

代码清单 2-7 展示了一个检查和处理 ArrayBuffer 消息的处理程序。

代码清单 2-7　ArrayBuffer 消息 message 事件处理程序示例

```
// Set binaryType to ArrayBuffer messages
ws.binaryType = "arraybuffer";

// Event handler for receiving ArrayBuffer messages
ws.onmessage = function(e) {
   if(e.data instanceof ArrayBuffer){
      console.log("ArrayBuffer Message Received", + e.data);
      // e.data is an ArrayBuffer. Create a byte view of that object.
      var a = new Uint8Array(e.data);
   }
};
```

3. WebSocket 事件：error

error 事件在响应意外故障的时候触发。与该事件对应的回调函数为 onerror。错误还会导致 WebSocket 连接关闭。如果你接收一个 error 事件，可以预期很快就会触发 close 事件。close 事件中的代码和原因有时候能告诉你错误的根源。error 事件处理程序是调用服务器重连逻辑以及处理来自 WebSocket 对象的异常的最佳场所。代码清单 2-8 展示了监听 error 事件的一个例子。

代码清单 2-8 error 事件处理程序示例

```
// Event handler for errors in the WebSocket object
ws.onerror = function(e) {
   console.log("WebSocket Error: " , e);
   //Custom function for handling errors
   handleErrors(e);
};
```

4. WebSocket 事件：close

close 事件在 WebSocket 连接关闭时触发。对应于 close 事件的回调函数是 onclose。一旦连接关闭，客户端和服务器不再能接收或者发送消息。

说明 WebSocket 规范还定义了 ping 和 pong 帧，可以用于持续连接（keep-alive）、心跳、网络状态检测、延迟测量等，但是 WebSocket API 目前没有输出这些特性。尽管浏览器接受 ping 帧，但是不会触发对应 WebSocket 上的 ping 事件。相反，浏览器将自动响应 pong 帧。然而，浏览器实例化的 ping 如果在一段时间内没有得到 pong 应答，可能会触发连接的 close 事件。第 8 章将详细介绍 WebSocket 的 ping 和 pong。

当调用 close() 方法终止与服务器的连接时，也会触发 onclose 事件处理程序，如代码清单 2-9 所示。

代码清单 2-9 close 事件处理程序示例

```
// Event handler for closed connections
ws.onclose = function(e) {
   console.log("Connection closed", e);
};
```

WebSocket close 事件在连接关闭时触发，这可能有多种原因，比如连接失败或者成功的 WebSocket 关闭握手。WebSocket 对象特性 readyState 反映了连接的状态（2 为正在关闭，3 为已关闭）。

close 事件有 3 个有用的属性（property），可以用于错误处理和恢复：wasClean、code 和 error。wasClean 属性是一个布尔属性，表示连接是否顺利关闭。如果 WebSocket 的关闭是

对来自服务器的一个 close 帧的响应，则该属性为 `true`。如果连接是因为其他原因（例如，因为底层 TCP 连接关闭）关闭，则该属性为 `false`。`code` 和 `reason` 属性表示服务器发送的关闭握手状态。这些属性和 `WebSocket.close()` 方法中的 `code` 和 `reason` 参数一致，我们将在本章后面详加介绍。在第 3 章中，我们将在讨论 WebSocket 协议时讨论关闭的代码和它们的含义。

说明　关于 WebSocket 事件的更多细节，参见 WebSocket API 规范：http://www.w3.org/TR/websockets/。

2.2.3　WebSocket 方法

WebSocket 对象有两个方法：`send()` 和 `close()`。

1. WebSocket 方法：`send()`

使用 WebSocket 在客户端和服务器之间建立全双工双向连接后，就可以在连接打开时（也就是说，在调用 `onopen` 监听器之后，调用 `onclose` 监听器之前）调用 `send()` 方法。使用 `send()` 方法可以从客户端向服务器发送消息。在发送一条或者多条消息之后，可以保持连接打开，或者调用 `close()` 方法终止连接。

代码清单 2-10 是向服务器发送文本消息的一个例子。

代码清单 2-10　通过 WebSocket 发送一条文本消息

```
// Send a text message
ws.send("Hello WebSocket!");
```

`send()` 方法在连接打开的时候发送数据。如果连接不可用或者关闭，它抛出一个有关无效连接状态的异常。人们开始使用 WebSocket API 时常犯的一个错误是试图在连接打开之前发送消息，如代码清单 2-11 所示。

代码清单 2-11　试图在打开连接之前发送消息

```
// Open a connection and try to send a message. (This will not work!)
var ws = new WebSocket("ws://echo.websocket.org")
ws.send("Initial data");
```

代码清单 2-11 不能正常工作，因为连接尚未打开。你应该等待 open 事件触发，才能在新构造的 WebSocket 上发送第一条消息，如代码清单 2-12 所示。

代码清单 2-12　在发送消息之前等待 open 事件

```
// Wait until the open event before calling send().
var ws = new WebSocket("ws://echo.websocket.org")
ws.onopen = function(e) {
   ws.send("Initial data");
}
```

如果想发送消息响应另一个事件，可以检查 WebSocket readyState 属性，并选择只在套接字打开时发送数据，如代码清单 2-13 所示。

代码清单 2-13　检查 readyState 属性了解 WebSocket 是否打开

```
// Handle outgoing data. Send on a WebSocket if that socket is open.
function myEventHandler(data) {
   if (ws.readyState === WebSocket.OPEN) {
      // The socket is open, so it is ok to send the data.
      ws.send(data);
   } else {
      // Do something else in this case.
      //Possibly ignore the data or enqueue it.
   }
}
```

除了文本（字符串）消息之外，WebSocket API 允许发送二进制数据，这对于实现二进制协议特别有用。这样的二进制协议可能是 TCP 上层的标准互联网协议，这些协议的载荷可能是 Blob 或 ArrayBuffer。代码清单 2-14 是通过 WebSocket 发送二进制消息的一个示例。

代码清单 2-14　通过 WebSocket 发送二进制消息

```
// Send a Blob
var blob = new Blob("blob contents");
ws.send(blob);

// Send an ArrayBuffer
var a = new Uint8Array([8,6,7,5,3,0,9]);
ws.send(a.buffer);
```

说明　第 6 章将展示一个通过 WebSocket 发送二进制数据的例子。

Blob 对象在与 JavaScript File API 结合使用以发送和接收文件时特别有用，这些文件主要有多媒体文件、图像、视频和音频。本章最后的示例代码结合 WebSocket API 和 JavaScript File API，读取文件内容，并将其作为 WebSocket 消息发送。

2. WebSocket 方法：close()

使用 close() 方法，可以关闭 WebSocket 连接或者终止连接尝试。如果连接已经关闭，该方法就什么都不做。在调用 close() 之后，不能在已经关闭的 WebSocket 上发送任何数据。代码清单 2-15 展示了 close() 方法的一个例子。

代码清单 2-15　调用 close() 方法

```
// Close the WebSocket connection
ws.close();
```

可以向 close() 方法传递两个可选参数：code（数字型的状态代码）和 reason（一个文本字符串）。传递这些参数能够向服务器传递关于客户关闭连接原因的信息。我们将在第 3 章中介绍 WebSocket 关闭握手时详细讨论状态代码和原因。代码清单 2-16 展示了调用带参数的 close() 方法的一个例子。

代码清单 2-16　带有原因的 close() 方法调用

```
// Close the WebSocket connection because the session has ended successfully
ws.close(1000, "Closing normally");
```

代码清单 2-16 使用了代码 1000，正如代码中所描述的，这表示连接正常关闭。

2.2.4　WebSocket 对象特性

可以使用多种 WebSocket 对象特性提供关于 WebSocket 对象的更多信息：readyState、bufferedAmount 和 protocol。

1. WebSocket 对象特性：readyState

WebSocket 对象通过只读特性 readyState 报告其连接状态，

你在前面的几节中已经学到了一点相关的知识。这个属性根据连接状态自动变化，并提供关于 WebSocket 连接的有用信息。

表 2-1 描述了用于描述连接状态的 `readyState` 特性的 4 个不同值。

表 2-1 `readyState` 特性、取值和状态描述

特性常量	取值	状　态
WebSocket.CONNECTING	0	连接正在进行中，但还未建立
WebSocket.OPEN	1	连接已经建立。消息可以在客户端和服务器之间传递
WebSocket.CLOSING	2	连接正在进行关闭握手
WebSocket.CLOSED	3	连接已经关闭，不能打开

信息来源：万维网联盟，2012 年。

正如 WebSocket API 所描述的，当 WebSocket 对象第一次创建时，`readyState` 为 0，表示套接字正在连接。了解 WebSocket 连接的当前状态有助于应用程序的调试，例如，确保在尝试开始向服务器发送请求之前已经打开了 WebSocket 连接。这一信息对于了解连接的生命周期也很有用。

2. WebSocket 对象特性：`bufferedAmount`

设计应用程序时，你可能想要检查发往服务器的缓冲数据量，特别是在客户端应用程序向服务器发送大量数据的时候。尽管调用 `send()` 是立即生效的，但是数据在互联网上的传输却不是如此。浏览器将为你的客户端应用程序缓存出站数据，从而使你可以随时调用 `send()`，发送任意数量的数据。然而，如果你想知道数据在网络上传送的速率，WebSocket 对象可以告诉你缓存的大小。你可以使用 `bufferedAmount` 特性检查已经进入队列，但是尚未发送到服务器的字节数。这个特性报告的值不包括协议组帧开销或者操作系统、网络硬件所进行的缓冲。

代码清单 2-17 展示一个使用 `bufferedAmount` 特性每秒发送更新的例子。如果网络无法承受这一速率，它会相应地作出调整。

代码清单 2-17　bufferedAmount 示例

```
// 10k max buffer size.
var THRESHOLD = 10240;

// Create a New WebSocket connection
var ws = new WebSocket("ws://echo.websocket.org/updates");

// Listen for the opening event
ws.onopen = function () {
   // Attempt to send update every second.
   setInterval( function() {
      // Send only if the buffer is not full
      if (ws.bufferedAmount < THRESHOLD) {
         ws.send(getApplicationState());
      }
   }, 1000);
};
```

对于限制应用向服务器发送数据的速率，从而避免网络饱和，bufferedAmount 特性很有用。

专家提示　你可以在试图关闭连接之前检查对象的 bufferedAmount 特性，确定是否有些数据还没有从应用中发送到服务器。

3. WebSocket 对象特性：protocol

在前面关于 WebSocket 构造函数的讨论中，我们提到了 protocol 参数，它让服务器知道客户端理解并可在 WebSocket 上使用的协议。WebSocket 对象的 protocol 特性提供了另一条关于 WebSocket 实例的有用信息。客户端和服务器协议协商的结果可以在 WebSocket 对象上看到。protocol 特性包含在打开握手期间 WebSocket 服务器选择的协议名，换句话说，protocol 特性告诉你特定 WebSocket 上使用的协议。protocol 特性在最初的握手完成之前为空，如果服务器没有选择客户端提供的某个协议，该特性保持空值。

2.3　全部组合起来

现在，我们已经简单了解了 WebSocket 构造函数、事件、特性和方法，可以将所学的 WebSocket API 相关知识组合起来了。这

里，我们创建了一个客户端应用程序，通过 Web 与一台远程服务器通信，用 WebSocket 交换数据。我们的 JavaScript 客户端示例使用 ws://echo.websocket.org 上的“回显”服务器，它可以接收并返回你发送到服务器的任何消息。使用回显服务器对于纯粹的客户端测试很有用，特别是用于理解 WebSocket API 与服务器的交互方式。

首先，我们创建连接，然后在网页上显示我们的代码触发的事件，这来自于服务器。页面将显示连接到服务器的客户端的相关信息，向服务器发送消息并从服务器接收消息，然后从服务器断开。

代码清单 2-18 展示了服务器通信和消息传递的一个完整示例。

代码清单 2-18　使用 WebSocket API 的完整客户端应用程序

```
<!DOCTYPE html>
<title>WebSocket Echo Client</title>
<h2>Websocket Echo Client</h2>

<div id="output"></div>
<script>

// Initialize WebSocket connection and event handlers

function setup() {
   output = document.getElementById("output");
   ws = new WebSocket("ws://echo.websocket.org/echo");

// Listen for the connection open event then call the sendMessage function
   ws.onopen = function(e) {
      log("Connected");
      sendMessage("Hello WebSocket!")
   }

// Listen for the close connection event
   ws.onclose = function(e) {
      log("Disconnected: " + e.reason);
   }

// Listen for connection errors
   ws.onerror = function(e) {
      log("Error ");
   }

// Listen for new messages arriving at the client
   ws.onmessage = function(e) {
      log("Message received: " + e.data);
      // Close the socket once one message has arrived.
      ws.close();
```

```
    }
}

// Send a message on the WebSocket.
function sendMessage(msg){
   ws.send(msg);
      log("Message sent");
   }

// Display logging information in the document.
function log(s) {
   var p = document.createElement("p");
   p.style.wordWrap = "break-word";
   p.textContent = s;
   output.appendChild(p);

   // Also log information on the javascript console
   console.log(s);
}

// Start running the example.
setup();
</script>
```

在运行网页之后，输出类似于下面这样：

```
WebSocket Sample Client


Connected


Message sent


Message received: Hello WebSocket!


Disconnected
```

如果你看到这一输出，那么恭喜你，你已经成功创建并执行了第一个 WebSocket 客户端应用程序。如果这个例子不能正常工作，你需要调查失败的原因。你可以在浏览器的 JavaScript 控制台上找到有用的信息。你的浏览器可能不支持 WebSocket，尽管这种情况越来越少了。虽然所有主流浏览器的最新版本都包含了对 WebSocket API 及协议的支持，但是仍然有一些在用的旧浏览器没有这方面的支持。下面说明如何确保浏览器支持 WebSocket。

2.4 检查 WebSocket 支持

因为并非所有浏览器都原生支持 WebSocket（令人吃惊！），所以在代码中包含确定浏览器支持的方法是很好的做法，如果有可能，要提供某种变通的办法。大部分现代浏览器都支持 WebSocket，但是根据用户的不同，你仍然可能应该使用某种技术来覆盖用户群。

说明 第 8 章讨论各种 WebSocket 变通方法和模拟选项。

确定浏览器是否支持 WebSocket 有多种方法，Web 浏览器的 JavaScript 控制台是可以用来研究代码的一个便捷工具。每个浏览器启动 JavaScript 控制台的方法各不相同。例如，在 Google Chrome 中，你可以选择“视图”→“开发者”→“开发者工具”[⊖]，然后点击“控制台”（console）选项。关于 Chrome 开发者工具的更多信息，参见 https://developers.google.com/chrome-developer-tools/docs/overview。

专家提示 Google 的 Chrome 开发者工具还可以检查 WebSocket 流量。在“开发者工具”面板中，单击“网络”（Network）选项，然后在面板底部单击“WebSockets”选项即可。附录 A 将详细介绍 WebSocket 的实用调试工具。

打开浏览器的交互式 JavaScript 控制台，并求取 `window.WebSocket` 表达式的值。如果你看到 WebSocket 构造函数对象，就意味着浏览器对 WebSocket 有原生支持。如果浏览器支持 WebSocket，但是你的示例代码无法正常工作，就需要进一步调试代码。如果同一个表达式的求值返回空白或者“未定义”，浏览器对 WebSocket 就没有原生支持。

为了确保 WebSocket 应用程序在不支持 WebSocket 的浏览

⊖ 在 Google Chrome 28.0.1500.72 中文版中，可选择右上角的“自定义及控制 Google Chrome”图标→“工具”→“开发者工具”选项。——编辑注

器中正常工作，你必须研究变通或者模拟策略。你可以自己编写（很复杂），使用填充插件（Polyfill，用于旧的浏览器、复制标准 API 的一个 JavaScript 程序库），或者使用 Kaazing 等 WebSocket 供应商，这些供应商支持 WebSocket 模拟，可以使任何浏览器（包括和 Microsoft Internet Explorer 6 同时代的）支持 HTML5 WebSocket 标准 API。第 8 章将进一步讨论这些选项，作为在企业部署 WebSocket 应用程序的一部分。

作为应用程序的一部分，可以添加 WebSocket 支持的条件检查，如代码清单 2-19 所示。

代码清单 2-19　确定浏览器 WebSocket 支持的客户端代码

```
if (window.WebSocket){
    console.log("This browser supports WebSocket!");
} else {
    console.log("This browser does not support WebSocket.");
}
```

说明　许多在线资源描述了 HTML5 和 WebSocket 与浏览器的兼容性——包括移动浏览器。http://caniuse.com/ 和 http://html5please.com/ 就是两个这样的资源。

2.5　在 WebSocket 中使用 HTML5 媒体

作为 HTML5 和 Web 平台的一部分，WebSocket API 可以很好地和所有 HTML5 特性（feature）配合。这个 API 所能发送和接收的数据类型广泛地用于传输应用程序数据和媒体。字符串当然可以表示 XML 和 JSON 等 Web 数据格式。二进制类型可以和拖放（Drag-and-Drop）、FileReader、WebGL 和 Web Audio API 等集成。

我们来看看如何结合 WebSocket 使用 HTML5 媒体。代码清单 2-20 展示了一个结合 WebSocket 使用 HTML5 媒体的完整客户端应用程序。你可以根据这些代码创建自己的 HTML 文件。

说明 为了构建（或者只是理解）本书中的例子，你可以选择使用我们创建的虚拟机，这个虚拟机包含了我们在示例中使用的所有代码、程序库和服务器。下载、安装和启动这个虚拟机的说明参见附录 B。

代码清单 2-20 结合 WebSocket 使用 HTML5 媒体的完整客户端应用程序

```
<!DOCTYPE html>
<title>WebSocket Image Drop</title>
<h1>Drop Image Here</h1>
<script>

// Initialize WebSocket connection
var wsUrl = "ws://echo.websocket.org/echo";
var ws = new WebSocket(wsUrl);
ws.onopen = function() {
 console.log("open");
}

// Handle binary image data received on the WebSocket
ws.onmessage = function(e) {
   var blob = e.data;
   console.log("message: " + blob.size + " bytes");
   // Work with prefixed URL API
  if (window.webkitURL) {
      URL = webkitURL;
   }

   var uri = URL.createObjectURL(blob);
   var img = document.createElement("img");
   img.src = uri;
   document.body.appendChild(img);
}
// Handle drop event
document.ondrop = function(e) {
   document.body.style.backgroundColor = "#fff";
   try {
      e.preventDefault();
     handleFileDrop(e.dataTransfer.files[0]);
     return false;
   } catch(err) {
     console.log(err);
   }
}

// Provide visual feedback for the drop area
document.ondragover = function(e) {
   e.preventDefault();
   document.body.style.backgroundColor = "#6fff41";
}
```

```
document.ondragleave = function() {
  document.body.style.backgroundColor = "#fff";
}

// Read binary file contents and send them over WebSocket
function handleFileDrop(file) {
   var reader = new FileReader();
   reader.readAsArrayBuffer(file);
   reader.onload = function() {
     console.log("sending: " + file.name);
     ws.send(reader.result);
   }
}
</script>
```

在你所喜爱的现代浏览器中打开这个文件。在 WebSocket 连接打开时查看浏览器的 JavaScript 控制台。图 2-1 展示了在 Mozilla Firefox 中客户端应用程序的运行情况。注意，在该图的底部，我们显示了 Firebug（一个强大的开发和调试工具，可以从 http://getfirebug.com 获得）中的 JavaScript 控制台。

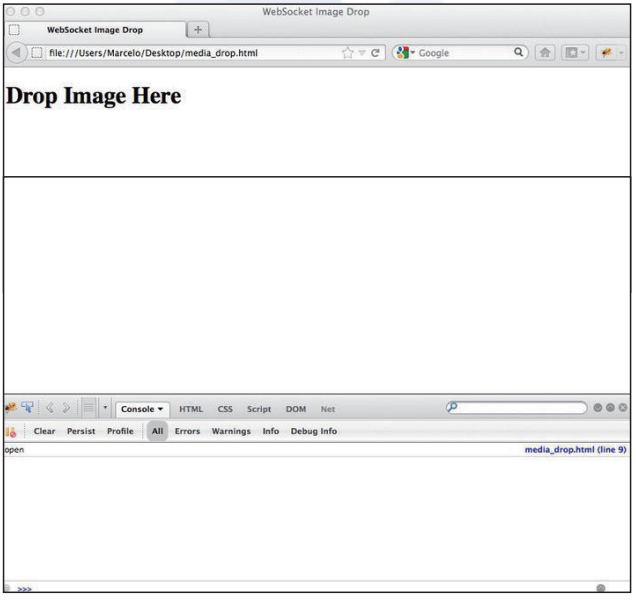

图 2-1　结合 WebSocket 使用 HTML5 媒体的客户端应用程序在 Mozilla Firefox 中的显示

现在，试着将一个图像文件拖放到这个页面上。在完成图像文件的拖放后，应该看到图像显示在网页上，如图 2-2 所示。注意 Firebug 对添加到页面的图像文件的显示方式。

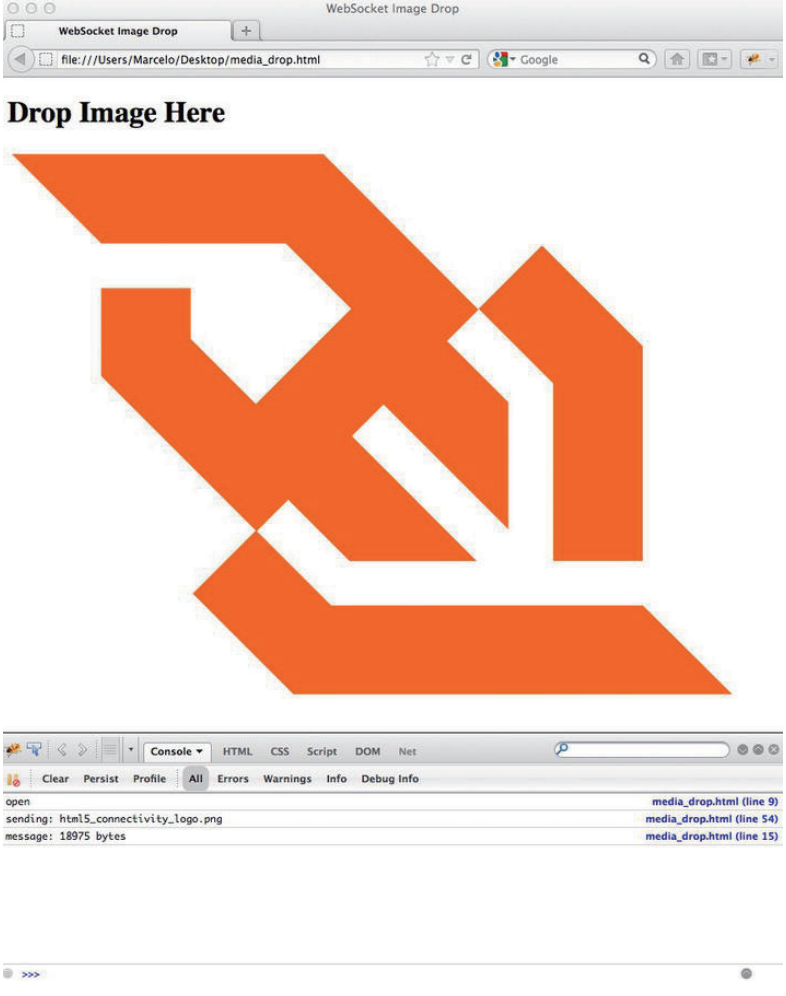

图 2-2　Mozilla Firefox 中，结合 WebSocket 使用 HTML5 媒体的客户端应用程序显示的图像（PNG）

说明　websocket.org 服务器目前只接受小消息，所以这个例子只适用于小于 65 KB 的图像文件，但是这个限制可能会变化，你可以在自己的服务器上试验更大的图像。

这个演示程序的绝妙特性因为媒体来源于最终显示它的同一浏览器而被削弱。你可以用 AJAX、甚至完全不需要网络就能够实现同样的视觉效果。当客户端或者服务器发送一些媒体数据，由不同的浏览器（甚至是其他数千个浏览器）显示时，情形就变得有趣了。在广播方案中使用的二进制图像数据读取和显示机制与这个简单的“回显”演示相同。

2.6　小结

在本章中，你学习了 WebSocket API 各个方面的相关知识，这个 API 使你能够从浏览器中运行的一个客户端应用程序中启动一个 WebSocket 连接，并通过 WebSocket 连接在客户端和服务器之间发送消息。你学习了 WebSocket API 背后的概念，包括事件、消息和特性，并看到了几个 API 的实用示例。你还学习了如何用公开可用的 WebSocket 回显服务器创建自己的 WebSocket 应用程序，你可以用这个服务器进一步测试自己的应用程序。这个接口的权威定义可以参见 http://www.w3.org/TR/websockets/ 上的完整 WebSocket API 规范。

在第 3 章中，你将学习 WebSocket 协议并逐步构建自己的简单 WebSocket 服务器。

第 3 章

WebSocket 协议

WebSocket 是定义服务器和客户端如何通过 Web 通信的一种网络协议。协议是通信的议定规则。组成互联网的协议组由 IETF（互联网工程任务组）发布。IETF 发布评议请求（Request for Comments，RFC），精确地规定了协议（包括 RFC 6455）：WebSocket 协议。RFC 6455 于 2011 年 12 月发布，包含了实现 WebSocket 客户端和服务器时必须遵循的规则。

在前一章中，我们研究了 WebSocket API，该 API 允许应用程序与 WebSocket 协议交互。本章，我们将带你回顾互联网和协议的历史，了解 WebSocket 协议创建的原因及其工作原理。我们使用网络工具观测和了解 WebSocket 网络流量。使用以 JavaScript 和 Node.js 编写的示例 WebSocket 服务器，我们将研究 WebSocket 握手建立 WebSocket 连接的方法、消息的编码和解码方式，以及持续连接和关闭连接的手段。最后，我们将使用示例 WebSocket 服务器同时远程控制多个浏览器。

3.1　WebSocket 协议之前

为了更好地理解 WebSocket 协议，我们通过快速浏览，看看相关的历史背景，从而了解 WebSocket 是如何融入重要的协议家族之中的。

协　　议

协议是计算中最重要的部分之一。它们跨越编程语言、操作系统和硬件体系结构，允许不同人编写、不同机构操作的组件跨越房屋甚至在全球范围内通信。开放、可互操作系统中的许多成功范例都归功于精心设计的协议。

在万维网及其基础技术 HTML、HTTP 等推出之前，互联网和现在完全不同。一方面，它比现在小得多；另一方面，它实际上是一个对等网络。当时互联网主机之间通信的两个流行协议现在仍然盛行：互联网协议（Internet Protocol，IP）和传输控制协议（Transmission Control Protocol，TCP），前者负责在互联网的两台

主机之间传送数据封包，后者可以看作跨越互联网，在两个端点之间可靠地双向传输字节流的一个管道。两者结合起来的 TCP/IP 历史上是无数网络应用程序使用的核心传输层协议，这种情况仍在持续。

3.1.1 互联网简史

一开始，互联网主机之间采用 TCP/IP 通信。在这种情况下，任一台主机都可以建立新的连接。一旦 TCP 连接建立，两台主机都可以在任何时候发送数据，如图 3-1 所示。

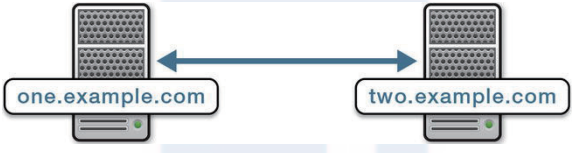

图 3-1 互联网主机之间的 TCP/IP 通信

你想在网络协议中实现的其他功能必须在传输协议基础上构建。这些更高的层次被称作*应用层协议*。例如，在 Web 之前出现的用于聊天的 IRC 和用于远程终端访问的 Telnet 就是两个重要的应用层协议。IRC 和 Telnet 显然需要异步的双向通信。客户端必须在另一个用户发送聊天消息或者远程应用程序打印一行输出时接收到提示通知。由于这些协议一般在 TCP 之上运行，异步双向通信总是可用。IRC 和 Telnet 会话维持客户端和服务器之间的持续连接，使客户端和服务器可以在任何时候自由地相互发送数据。TCP/IP 还是其他两个重要协议的基础：HTTP 和 WebSocket。按照顺序，我们先来简单地了解一下 HTTP。

3.1.2 Web 和 HTTP

1991 年，万维网（World Wide Web）项目第一次公布。Web 是使用统一资源定位符（URL）链接的超文本文档系统。当时，URL 是一个重大的发明。“URL”中的“U”是“universal”（统一）的缩写，说明了当时的一个革命性想法——所有超文本文档

可以相互连接。Web 上的 HTML 文档通过 URL 相互链接。更有意义的是，Web 协议经过裁剪，用于读取资源。HTTP 是一个用于文档传输的简单同步请求 – 响应式协议。

最早的 Web 应用程序使用表单和全页刷新。每当用户提交信息，浏览器将提交一个表单并读取新页面。每当有需要显示的更新信息，用户或者浏览器必须刷新整个页面，使用 HTTP 读取整个资源。

利用 JavaScript 和 XMLHttpRequest API，人们开发出了一组称为 AJAX 的技术，这项技术能够使应用程序在每次交互期间不会有不连贯的过渡。AJAX 使应用程序只读取感兴趣的资源数据，并在没有导航的情况下更新页面。AJAX 使用的网络协议仍然是 HTTP；尽管名为 XMLHttpRequest，数据也只是有时使用 XML 格式，而不是始终使用该格式。

Web 已经变得非常流行，以至于许多人混淆了 Web 和互联网这两个概念，因为 Web 常常是他们唯一重要的互联网应用程序。NAT（Network Address Translation，网络地址转换）、HTTP 代理和防火墙也越来越常见。今天，许多互联网用户都没有公开的 IP 地址，用户并不都有唯一的 IP 地址，这有许多原因，其中包括安全措施、过度拥挤和缺乏必要性。地址的短缺妨碍了可寻址性，例如，需要公开地址的蠕虫无法访问没有编址的用户。此外，IPv4 地址已经不足以供所有 Web 用户使用。NAT 使用户可以共享 IP 地址进行 Web 冲浪。最后，占统治地位的 HTTP 协议并不要求客户端可寻址。HTTP 可以很好地使用客户端应用程序驱动的交互，因为所有 HTTP 请求都由客户端发起，如图 3-2 所示。

本质上，HTTP 用其内置的文本支持（从而支持互相连接的 HTML 页面）、URL 和 HTTPS（通过传输层安全（TLS）的安全 HTTP）使 Web 成为可能。然而，在某种程度上，HTTP 的流行也造成了互联网的退化。因为 HTTP 不需要可寻址的客户端，Web 世界的寻址变成不对称的。浏览器能够通过 URL 寻找服务器资源，但是服务器端应用程序却无法主动地向客户端发送资源。客户端只能发起请求，而服务器只能响应未决的请求。在这个非对

称的世界中，要求全双工通信的协议无法正常工作。

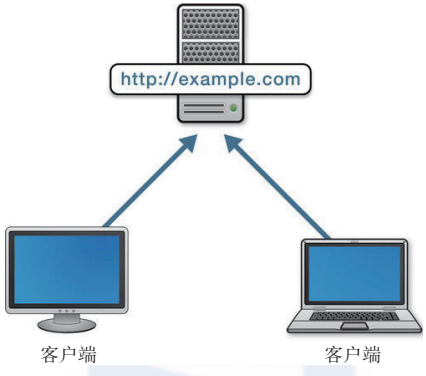

图 3-2 连接到 Web 服务器的 HTTP 客户端

解决这一局限性的方法之一是由客户端发出 HTTP 请求，以防服务器有需要共享的更新。使用 HTTP 请求颠倒通知流程的这一过程用一个伞形术语“Comet”来表示。正如我们在前面几章中讨论过的，Comet 本质上是一组利用轮询、长轮询和流化开发 HTTP 潜力的技术。这些技术实际上模拟了 TCP 的一些功能，以便处理上述的服务器 – 客户端用例。因为同步的 HTTP 和这些异步应用程序之间不匹配，Comet 复杂、不标准且低效。

说明 在服务器 – 服务器通信中，每台主机都可以寻址其他主机。一台服务器可以在有新数据可用时向其他服务器发出 HTTP 请求，这就是服务器间 feed（摘要）更新通知所用的 PubSubHubbub 协议的通信方式。PubSubHubbub 是一种扩展 RSS 和 Atom，以及允许 HTTP 服务器之间发布 / 订阅通信的开放协议。虽然可以使用 WebSocket 进行服务器 – 服务器通信，但是本书的重点在于实时 Web 应用程序中的客户端 – 服务器通信。

3.2 WebSocket 协议简介

简短的互联网历史课之后，我们回到当前。现在，Web 应

用程序相当强大，具有重要的客户端状态和逻辑。现代 Web 应用程序常常需要双向通信。更新的即时通知与其说是例外，不如说是惯例，用户也越来越期待反应灵敏的实时交互。我们来看看 WebSocket 带来了什么。

3.2.1　WebSocket：Web 应用程序的互联网能力

WebSocket 为 Web 应用程序保留了我们所喜欢的 HTTP 特性（URL、HTTP 安全性、更简单的基于数据模型的消息和内置的文本支持），同时提供了其他网络架构和通信模式。和 TCP 一样，WebSocket 是异步的，可以用作高级协议的传输层。WebSocket 是消息协议、聊天、服务器通知、管道和多路复用协议、自定义协议、紧凑二进制协议和用于与互联网服务器互操作的其他标准协议的很好基础。

WebSocket 为 Web 应用程序提供了 TCP 风格的网络能力。寻址仍然是单向的，服务器可以异步发送客户端数据，但是只在 WebSocket 连接打开时才能做到。在客户端和服务器之间 WebSocket 连接始终打开。WebSocket 服务器也可以作为 WebSocket 客户端。但是，利用 WebSocket，Web 客户端（如浏览器）不能接受不是由它们建立的连接。图 3-3 展示了连接到一台服务器的 WebSocket 客户端，客户端和服务器都可以在任何时候发送数据。

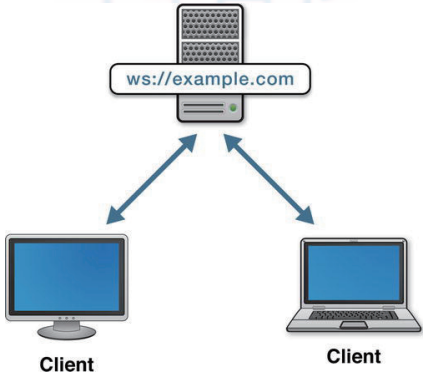

图 3-3　连接到服务器的 WebSocket 客户端

WebSocket 是连通 Web 世界和互联网世界（更具体地说是 TCP/IP）的桥梁。过去不容易用于 Web 应用程序的异步协议现在可以轻松地使用 WebSocket 通信。表 3-1 比较了 TCP、HTTP 和 WebSocket 的主要特性。

表 3-1 TCP、HTTP 和 WebSocket 的对比

特　性	TCP	HTTP	WebSocket
寻址	IP 地址和端口	URL	URL
并发传输	全双工	半双工	全双工
内容	字节流	MIME 消息	文本和二进制消息
消息定界	否	是	是
连接定向	是	否	是

TCP 只能传送字节流，所以消息边界只能由更高层的协议来表现。使用 TCP 的套接字编程新手常犯的一个错误是，假定每次调用 `send()` 都会得到一次成功的 `receive`。在简单的测试中，这一假定可能恰好没错。当负载和延迟变化时，TCP 套接字上发送的字节可能会有不可预测的碎片。TCP 数据可能根据操作系统的喜好分布到多个 IP 封包或者合并为较少的封包。对于 TCP 来说，唯一可以保证的是，到达接收端的单个字节将会按顺序到达。和 TCP 不同，WebSocket 传输一序列单独的消息，在 WebSocket 中，和 HTTP 一样，多字节的消息作为整体、按照顺序到达。因为 WebSocket 协议内置了消息边界，所以它能够发送和接收单独的消息并避免常见的碎片错误。

值得一提的是，在互联网出现之前，人们遵循的是另一个网络模型：开放系统互连（Open Systems Interconnection，OSI），它包含 7 个层次：物理层、数据链路层、网络层、传输层、会话层、表示层和应用层。然而，尽管术语类似，OSI 设计时没有考虑互联网。为互联网设计的 TCP/IP 协议模型只由 4 个层次组成：链路层、互联网层、传输层和应用层，这一模型推动着当今的互联网。

IP 处于互联网层，而 TCP 处于 IP 之上的传输层。WebSocket 的层次在 TCP 之上（也在 IP 之上），因为你可以在 WebSocket 上构建应用级协议，所以它也被看作是传输层协议。

3.2.2　检查 WebSocket 流量

在第 2 章中，我们使用了 WebSocket API，但没有真正地看到网络层上发生了什么。如果你想看到网络上的 WebSocket 流量，可以使用 Wireshark（http://www.wireshark.org/）或 tcpdump（http://www.tcpdump.org/）等工具，检查通信栈中的内容。Wireshark 能够剖析 WebSocket 协议，让你在一个方便的 UI 中（见图 3-4）看到我们在本章后面所要讨论的 WebSocket 协议各个部分（例如操作码、标志和载荷）。它甚至能够显示从 WebSocket 客户端发出的消息的无屏蔽版本。我们将在本章后面讨论屏蔽（masking）。

说明　附录 A 详细介绍了 WebSocket 流量调试工具。

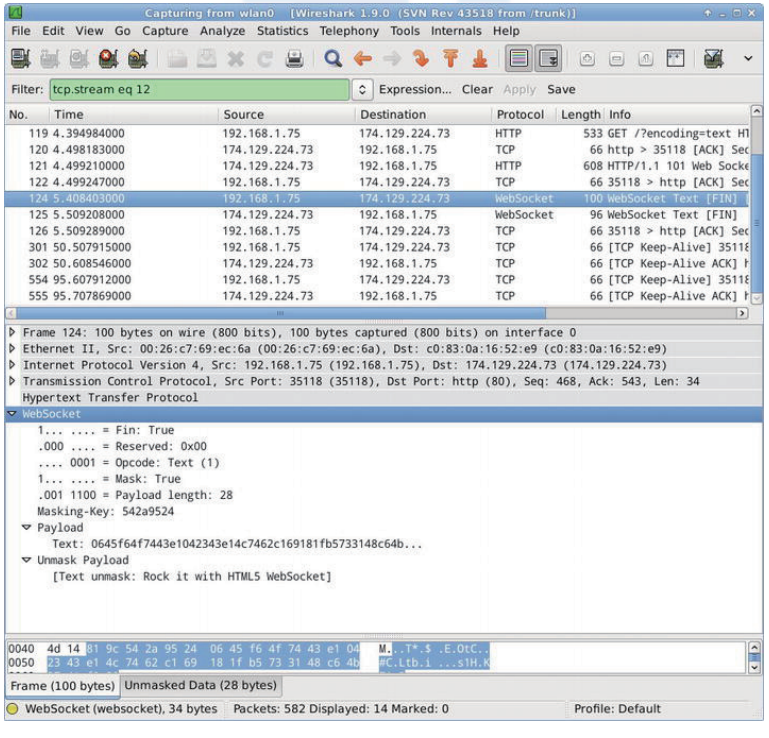

图 3-4　在 Wireshark 中查看 WebSocket 会话

WebKit（驱动 Google Chrome 和 Apple Safari 的浏览器引擎）最近也添加了对检查 WebSocket 流量的支持。在最新版本的 Chrome 浏览器中，可以在开发者工具的“网络”（Network）选项卡中查看 WebSocket 消息。图 3-5 展示了这个 WebSocket 开发必备的工具。

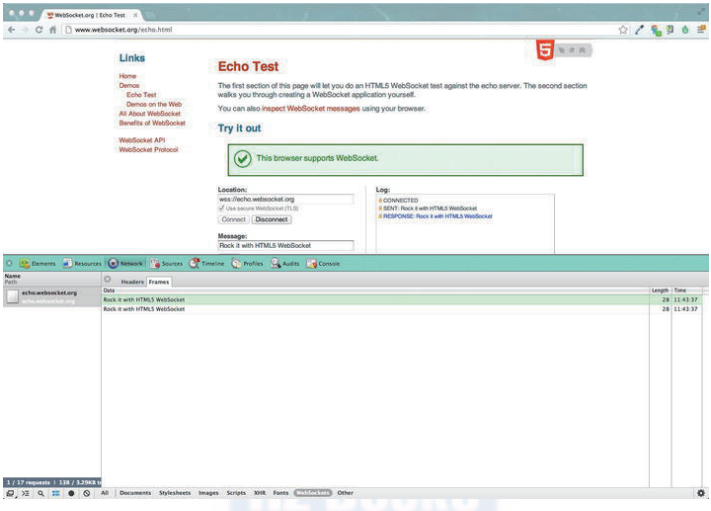

图 3-5　使用 Google Chrome 开发者工具查看 WebSocket 会话

强烈建议使用这些工具查看 WebSocket 的实际运行情况，这不仅能够学习协议的相关知识，还能更好地理解 WebSocket 会话期间发生的一切。

3.3　WebSocket 协议

我们来更加详细地看看 WebSocket 协议。对于协议的每个部分，我们将研究处理特定语法的 JavaScript 代码。之后，我们将把这些代码片段组合成一个服务器程序库示例和两个简单的应用程序。

3.3.1　WebSocket 初始握手

每个 WebSocket 连接都始于一个 HTTP 请求。该请求和其他请求很相似，但是包含一个特殊的首标——Upgrade。Upgrade 首标表示客户端将把连接升级到不同的协议。在这种情况下，这个不同的协议就是 WebSocket。

我们来看看连接到 ws://echo.websocket.org/echo 期间记录的一个握手示例。在握手完成之前，WebSocket 会话遵循 HTTP/1.1 协议。客户端发送的 HTTP 请求如代码清单 3-1 所示。

代码清单 3-1　客户端发起的 HTTP 请求

```
GET /echo HTTP/1.1
Host: echo.websocket.org
Origin: http://www.websocket.org
Sec-WebSocket-Key: 7+C6OOxYybOv2zmJ69RQsw==
Sec-WebSocket-Version: 13
Upgrade: websocket
```

代码清单 3-2 展示了服务器发回的响应。

代码清单 3-2　从服务器发出的 HTTP 响应

```
101 Switching Protocols
Connection: Upgrade
Date: Wed, 20 Jun 2012 03:39:49 GMT
Sec-WebSocket-Accept: fYoqiH14DgI+5ylEMwM2sOLzOiO=
Server: Kaazing Gateway
Upgrade: WebSocket
```

图 3-6 展示了从客户端发往服务器，升级为 WebSocket 的 HTTP 请求，这也称作 WebSocket 初始握手（opening handshake）。

图 3-6 说明了要求和可选的首标。有些首标是必需的，必须存在并且要精确才能保证 WebSocket 连接成功。这一握手中的其他首标是可选的，但因为握手是一次 HTTP 请求和响应，所以也是允许的。在成功升级之后，连接的语法切换为用于表示 WebSocket 消息的数据帧格式。除非服务器响应 101 代码、Upgrade 首标和 Sec-WebSocket-Accept 首标，否则 WebSocket 连接不能成功。Sec-WebSocket-Accept 响应首标的值从 Sec-WebSocket-Key 请求首标继承而来，包含一个特殊的响应键值，必须与客户

端的预期精确匹配。

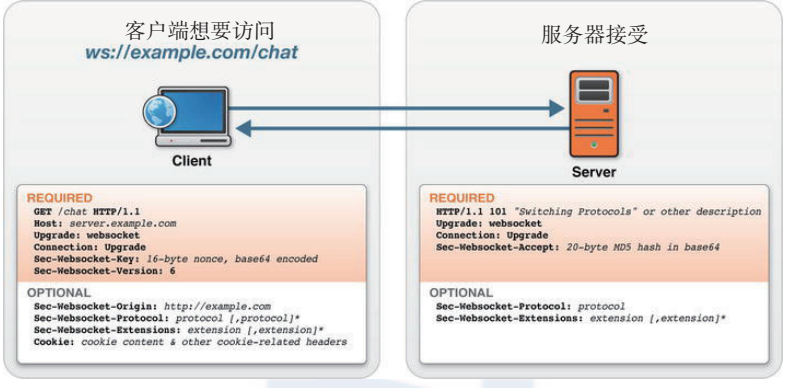

图 3-6 WebSocket 初始握手示例

3.3.2 计算响应键值

为了成功地完成握手，WebSocket 服务器必须响应一个计算出来的键值。这个响应说明服务器理解 WebSocket 协议。没有精确的响应，就可能哄骗一些轻信的 HTTP 服务器意外地升级一个连接。

响应函数从客户端发送的 `Sec-WebSocket-Key` 首标中取得键值，并在 `Sec-WebSocket-Accept` 首标中返回根据客户端预期计算的键值。代码清单 3-3 使用 Node.js 加密 API 计算键值和后缀组合的 SHA1 散列值。

代码清单 3-3 用 Node.js 加密 API 计算响应键值

```
var KEY_SUFFIX = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";
var hashWebSocketKey = function(key) {
   var sha1 = crypto.createHash("sha1");
   sha1.update(key + KEY_SUFFIX, "ascii");
   return sha1.digest("base64");
   }
```

说明 代码清单 3-3 中的 `KEY_SUFFIX` 是一个协议规范中包含的固定键值后缀，每个 WebSocket 服务器都必须知道。

在 WebSocket 初始握手和响应键值的计算中，WebSocket 协议依靠 RFC 6455 中定义的 Sec- 首标。表 3-2 描述了 WebSocket Sec- 首标。

表 3-2　WebSocket Sec- 首标和描述（RFC 6455）

首　标	描　述
Sec-WebSocket-Key	只能在 HTTP 请求中出现一次 用于从客户端到服务器的 WebSocket 初始握手，避免跨协议攻击。参见 Sec-WebSocket-Accept
Sec-WebSocket-Accept	只能在 HTTP 请求中出现一次 用于从客户端到服务器的 WebSocket 初始握手，确认服务器理解 WebSocket 协议
Sec-WebSocket-Extensions	可能在 HTTP 请求中出现多次（从逻辑上说，这和包含所有值的单一 Sec-WebSocket-Extensions 首标相同），但是在 HTTP 响应中只能出现一次 用于从客户端到服务器的 WebSocket 初始握手，然后用于从服务器到客户端的响应。这个首标帮助客户端和服务器商定一组连接期间使用的协议级扩展
Sec-WebSocket-Protocol	用于从客户端到服务器的 WebSocket 初始握手，然后用于从服务器到客户端的响应。这个首标通告客户端应用程序可使用的协议。服务期使用相同的首标，在这些协议中的最多选择一个
Sec-WebSocket-Version	用于从客户端到服务器的 WebSocket 初始握手，表示版本兼容性。RFC 6455 的版本总是 13。服务器如果不支持客户端请求的协议版本，则用这个首标响应。在那种情况下，服务器发送的首标中列出了它支持的版本。这只发生在 RFC 6455 之前的客户端中

3.3.3　消息格式

当 WebSocket 连接打开时，客户端和服务器可以在任何时候相互发送消息。这些消息在网络上用标记消息之间边界并包括简洁的类型信息的二进制语法表示。更准确地说，这些二进制首标标记另一个单位——帧（frame）——之间的边界。帧是可以合并组成消息的部分数据。你可能在 WebSocket 的相关讨论中将“帧”和“消息”互换使用，这是因为很少有一个消息使用超过一个帧

（至少目前如此）。而且，在协议帧的早期草案中，帧就是消息，消息在线路上的表示被称作“组帧”（framing）。

回忆第 2 章的内容，WebSocket API 没有向应用程序暴露帧级别的信息。尽管 API 按照消息工作，但是可以在协议级别上处理子消息数据单元。虽然消息一般只有一个帧，但是它可以由任意数量的帧组成。服务器可以使用不同数量的帧，在全体数据可用之前开始交付数据。

我们来更详细地看看 WebSocket 帧的特征。图 3-7 展示了 WebSocket 帧头。

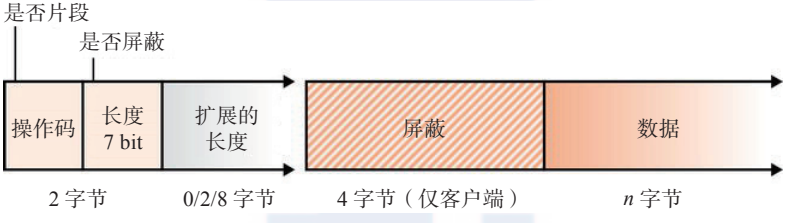

图 3-7　WebSocket 帧头

WebSocket 帧化代码负责：

- 操作码
- 长度
- 解码文本
- 屏蔽
- 多帧消息

1. 操作码

每条 WebSocket 消息都有一个指定消息载荷类型的操作码（opcode）。操作码由帧头的第一个字节中最后 4 bit 组成，取数字值，说明见表 3-3。

表 3-3　操作码定义

操　作　码	消息载荷类型	描　　述
1	文本	消息的数据类型为文本
2	二进制	消息的数据类型为二进制的

（续）

操　作　码	消息载荷类型	描　　述
8	关闭	客户端或者服务器向对方发送关闭握手
9	ping	客户端或者服务器向对方发送 ping（ping 和 pong 的使用详见第 8 章）
10（十六进制 0xA）	pong	客户端或者服务器向对方发送 pong（ping 和 pong 的使用详见第 8 章）

4 bit 的操作码有 16 种可能取值，WebSocket 协议只定义了 5 种操作码，剩余的操作码保留用于未来的扩展。

2. 长度

WebSocket 协议使用可变位数来编码帧长度，这样，小的消息就能使用紧凑的编码，协议仍然可以携带中型甚至非常大的消息。对于小于 126 字节的消息，长度用帧头前两个字节之一表示。对于 126 ~ 216 字节的消息，使用两个额外的字节表示长度。对于大于 216 字节的消息，长度为 8 字节。该长度编码保存于帧头第二个字节的最后 7 位。该字段中 126 和 127 两个值被当作特殊的信号，表示需要后面的字节才能完成长度编码。

3. 解码文本

WebSocket 文本消息用 8 位 UCS 转换格式（UTF-8）编码。UTF-8 是用于 Unicode 的变长编码，向后兼容 7 位的 ASCII，也是 WebSocket 文本消息允许的唯一编码。坚持使用 UTF-8 编码避免了大量的“普通文本”格式以及协议中的不同编码对互操作性的妨害。

在代码清单 3-4 中，`deliverText` 函数使用来自 Node.js 的 `buffer.toString()` API 将入站消息的载荷转换为 JavaScript 字符串。UTF-8 是 `buffer.toString()` 的默认编码，这里指定格式是为了清晰。

代码清单 3-4　UTF-8 文本编码

```
case opcodes.TEXT:
   payload = buffer.toString("utf8");
```

4. 屏蔽

从浏览器向服务器发送的 WebSocket 帧内容进行了“屏蔽”，以混淆其内容。屏蔽的目的不是阻止窃听，而是为了不常见的安全原因，以及改进和现有 HTTP 代理的兼容性。第 7 章会进一步解释屏蔽所能防止的跨协议攻击种类。

帧头第二个字节的第一位表示该帧是否进行了屏蔽：WebSocket 协议要求客户端屏蔽发送的所有帧。如果有屏蔽，所用的掩码将占据帧头扩展长度部分后的 4 个字节。

WebSocket 服务器接收的每个载荷在处理之前首先被解除屏蔽，代码清单 3-5 展示了一个简单的函数，用 4 个字节的掩码解除 WebSocket 帧载荷部分的屏蔽。

代码清单 3-5　解除载荷屏蔽

```
var unmask = function(mask_bytes, buffer) {
   var payload = new Buffer(buffer.length);
   for (var i=0; i<buffer.length; i++) {
      payload[i] = mask_bytes[i%4] ^ buffer[i];
      }
   return payload;
}
```

解除屏蔽之后，服务器得到原始消息内容：二进制消息可以直接交付；文本消息将进行 UTF-8 解码，并通过服务器 API 输出字符串。

5. 多帧消息

帧格式中的 fin 位考虑了多帧消息或者部分可用消息的流化，这些消息可能不连续或者不完整。要发送一条不完整的消息，你可以发送一个 fin 位设置为 0 的帧。最后一个帧的 fin 位设置为 1，表示消息以这一帧的载荷作为结束。

3.3.4　WebSocket 关闭握手

本章前面已经研究了 WebSocket 初始握手。在人类的交往中，我们往往在刚刚会面的时候握手。有时候，我们在分别的时候也握手。在这个协议中情况也是如此。WebSocket 连接总是以初始握手开始，因为这是初始化互联网和其他可靠网上对话的唯一手段，

连接可以在任何时候关闭，所以不可能总是以关闭握手结束。有时候，底层的 TCP 套接字可能突然关闭。关闭握手优雅地关闭连接，使应用程序能够知道有意中断和意外终止连接之间的差异。

当 WebSocket 关闭时，终止连接的端点可以发送一个数字代码，以及一个表示选择关闭套接字原因的字符串。代码和原因编码为具有关闭操作码（8）的一个帧的载荷。数字代码用一个 16 位无符号整数表示，原因则是一个 UTF-8 编码的短字符串。RFC 6455 定义了多种特殊的关闭代码。代码 1000 ~ 1015 规定用于 WebSocket 连接层。这些代码表示网络中或者协议中的某些故障。表 3-4 列出了这一系列代码、描述和每个代码适用的情况。

表 3-4　定义的 WebSocket 关闭代码

代　码	描　述	何时使用
1000	正常关闭	当你的会话成功完成时发送这个代码
1001	离开	因应用程序离开且不期望后续的连接尝试而关闭连接时，发送这一代码。服务器可能关闭，或者客户端应用程序可能关闭
1002	协议错误	当因协议错误而关闭连接时发送这一代码
1003	不可接受的数据类型	当应用程序接收到一条无法处理的意外类型消息时发送这一代码
1004	保留	不要发送这一代码。根据 RFC 6455，这个状态码保留，可能在未来定义
1005	保留	不要发送这一代码。WebSocket API 用这个代码表示没有接收到任何代码
1006	保留	不要发送这一代码。WebSocket API 用这个代码表示连接异常关闭
1007	无效数据	在接收一个格式与消息类型不匹配的消息之后发送这一代码。如果文本消息包含错误格式的 UTF-8 数据，连接应该用这个代码关闭
1008	消息违反政策	当应用程序由于其他代码所不包含的原因终止连接，或者不希望泄露消息无法处理的原因时，发送这一代码

（续）

代　码	描　述	何时使用
1009	消息过大	当接收的消息太大，应用程序无法处理时发送这一代码（记住，帧的载荷长度最多为64字节。即使你有一个大服务器，有些消息也仍然太大。）
1010	需要扩展	当应用程序需要一个或者多个服务器无法协商的特殊扩展时，从客户端（浏览器）发送这一代码
1011	意外情况	当应用程序由于不可预见的原因，无法继续处理连接时，发送这一代码
1015	TLS失败（保留）	不要发送这个代码。WebSocket API用这个代码表示TLS在WebSocket握手之前失败

说明　第2章说明了WebSocket API使用关闭代码的方式。更多WebSocket API的有关信息参见http://www.w3.org/TR/websockets/。

其他代码保留以用于特殊用途。表3-5列出了RFC 6455定义的4类关闭代码。

表3-5　WebSocket关闭代码范围

代　码	描　述	何时使用
0 ~ 999	禁止	1000以下的代码是无效的，不能用于任何目的
1000 ~ 2999	保留	这些代码保留以用于扩展和WebSocket协议的修订版本。按照标准规定使用这些代码，参见表3-4
3000 ~ 3999	需要注册	这些代码用于“程序库、框架和应用程序”。这些代码应该在IANA（互联网编号分配机构）公开注册
4000 ~ 4999	私有	在应用程序中将这些代码用于自定义用途。因为它们没有注册，所以不要期望它们能被其他WebSocket广泛理解

3.3.5　对其他协议的支持

WebSocket协议支持更高级的协议和协议协商。RFC 6455古怪地将用于WebSocket的协议称作“子协议”，而它们实际上是

更高级、完整的协议。正如我们在第 2 章中所提到的，在本书中，我们将把和 WebSocket 一起使用的协议简单地称作“协议”，以避免混淆。

在第 2 章中，我们说明了用 WebSocket API 协商更高层协议的方法。在网络层上，这些协议用 `Sec-WebSocket-Protocol` 首标协商。协议名称在客户端发送初始升级请求时用首标值表示：

```
Sec-WebSocket-Protocol: com.kaazing.echo, example.protocol.name
```

上述首标表示客户端可以使用任一个协议（`com.kaazing.echo` 或 `example.protocol.name`），服务器可以选择适用的协议。如果你在对 ws://echo.websocket.org 发送的升级请求中发送了这个首标，那么服务器响应中将包含如下首标：

```
Sec-WebSocket-Protocol: com.kaazing.echo
```

这个响应表示，服务器选择使用 `com.kaazing.echo` 协议。选择协议不会改变 WebSocket 协议本身的语法。相反，这些协议在 WebSocket 协议之上的层次，为框架和应用程序提供更高级的语义。我们将在后续章节中研究 WebSocket 上层的 3 个广泛使用、基于标准的协议用例。

要简单地扩展 WebSocket 协议，可以使用另一种机制——扩展（extension）。

3.3.6　扩展

和协议一样，扩展也使用 `Sec-` 首标协商。连接的客户端发送一个 `Sec-WebSocket-Extensions` 首标，包含所支持的扩展名称。

说明　不能同时协商多个协议，但是可以同时协商多个扩展。

例如，Chrome 可以发送如下的首标，表明它将接受一个试验性的压缩扩展：

```
Sec-WebSocket-Extensions: x-webkit-deflate-frame
```

扩展的得名是因为它们扩展了 WebSocket 协议。扩展可以为帧格式添加新的操作码和数据字段。你可能发现，部署新的扩展

比新协议（“子协议”）更困难，因为浏览器供应商必须明确地为这些扩展提供支持。你还可能发现，编写一个 JavaScript 程序库实现协议，比等待所有浏览器供应商标准化扩展、所有用户将浏览器更新到支持扩展的版本更容易一些。

3.4 用 Node.js 编写 JavaScript WebSocket 服务器

我们已经研究了 WebSocket 协议的精华部分，现在我们来编写一个自己的 WebSocket 服务器。WebSocket 协议有许多现成的实现，你可以选择在自己的应用程序中使用这些实现，然而，你可能因为必要性或者展现自己的才能，而需要编写新的服务器或者修改现有服务器。编写自己的 WebSocket 协议实现可能是有趣且具有启发意义的，能够帮助你理解和评估其他服务器、客户端和程序库。最好的一点是，它能够为你带来对网络、通信和 Web 的领悟。

本章中的示例服务器用 JavaScript 编写，利用了 Node.js 提供的 IO API。我们选择这些技术只是为了将本书中的代码例子限制在一种语言内。因为你可能正在使用 JavaScript 和 HTML5 进行前端开发，你也就很有可能顺利地阅读这些代码。当然，你并不一定要用 JavaScript 编写服务器，选择其他语言也有很好的理由。WebSocket 是一个与语言无关的协议，这意味着你可以选择任何能够监听套接字的语言来创建服务器。

我们编写的这个例子可以用于 Node.js 0.8，无法在更早的版本上运行，如果未来 Node API 变化，可能需要作一些修改。`websocket-example` 模块合并了前面的代码片断和一些附加的代码，以组成一个 WebSocket 服务器。这个例子并不健壮，也不能用于实际生产，但是它可以作为该协议的一个简单而完备的例子。

说明 要构建（或者只是理解）本书中的例子，你可以选择使用我们创建的包含所有用于例子中的代码、程序库和服务器的虚拟机（VM）。下载、安装和启动 VM 的指南参见附录 B。

3.4.1　构建简单的 WebSocket 服务器

代码清单 3-6 以构建一个简单 WebSocket 服务器作为我们的开始。你也可以打开文件 websocket-example.js 查看代码示例。

代码清单 3-6　用 JavaScript 和 Node.js 编写的 WebSocket 服务器 API

```
// The Definitive Guide to HTML5 WebSocket
//  Example WebSocket server

// See The WebSocket Protocol for the official specification
// http://tools.ietf.org/html/rfc6455

var events = require("events");
var http = require("http");
var crypto = require("crypto");
var util = require("util");

// opcodes for WebSocket frames
// http://tools.ietf.org/html/rfc6455#section-5.2

var opcodes = { TEXT  : 1
   , BINARY: 2
   , CLOSE : 8
   , PING  : 9
   , PONG  : 10
};

var WebSocketConnection = function(req, socket, upgradeHead) {
   var self = this;

   var key = hashWebSocketKey(req.headers["sec-websocket-key"]);
   // handshake response
   // http://tools.ietf.org/html/rfc6455#section-4.2.2

   socket.write('HTTP/1.1 101 Web Socket Protocol Handshake\r\n' +
      'Upgrade: WebSocket\r\n' +
      'Connection: Upgrade\r\n' +
      'sec-websocket-accept: ' + key +
      '\r\n\r\n');

   socket.on("data", function(buf) {
      self.buffer = Buffer.concat([self.buffer, buf]);
      while(self._processBuffer()) {
      // process buffer while it contains complete frames
      }
   });

   socket.on("close", function(had_error) {
      if (!self.closed) {
         self.emit("close", 1006);
         self.closed = true;
```

```
      }
   });

   // initialize connection state

   this.socket = socket;
   this.buffer = new Buffer(0);
   this.closed = false;
}
util.inherits(WebSocketConnection, events.EventEmitter);

// Send a text or binary message on the WebSocket connection

WebSocketConnection.prototype.send = function(obj) {
   var opcode;
   var payload;
   if (Buffer.isBuffer(obj)) {
      opcode = opcodes.BINARY;
      payload = obj;
   } else if (typeof obj == "string") {
   opcode = opcodes.TEXT;
// create a new buffer containing the UTF-8 encoded string
   payload = new Buffer(obj, "utf8");
   } else {
      throw new Error("Cannot send object. Must be string or Buffer");
   }
   this._doSend(opcode, payload);
}
// Close the WebSocket connection

WebSocketConnection.prototype.close = function(code, reason) {
   var opcode = opcodes.CLOSE;
   var buffer;

// Encode close and reason

if (code) {
   buffer = new Buffer(Buffer.byteLength(reason) + 2);
   buffer.writeUInt16BE(code, 0);
   buffer.write(reason, 2);
   } else {
      buffer = new Buffer(0);
   }
   this._doSend(opcode, buffer);
   this.closed = true;
}

// Process incoming bytes

WebSocketConnection.prototype._processBuffer = function() {
   var buf = this.buffer;

   if (buf.length < 2) {
      // insufficient data read
```

```
      return;
   }

   var idx = 2;

   var b1 = buf.readUInt8(0);
   var fin = b1 & 0x80;
   var opcode =  b1 & 0x0f;      // low four bits
   var b2 = buf.readUInt8(1);
   var mask = b2 & 0x80;
   var length = b2 & 0x7f;      // low 7 bits

   if (length > 125) {
      if (buf.length < 8) {
      // insufficient data read
      return;
   }

   if (length == 126) {
      length = buf.readUInt16BE(2);
      idx += 2;
      } else if (length == 127) {
      // discard high 4 bits because this server cannot handle huge lengths
      var highBits = buf.readUInt32BE(2);
      if (highBits != 0) {
      this.close(1009, "");
      }
      length = buf.readUInt32BE(6);
      idx += 8;
      }
   }

   if (buf.length < idx + 4 + length) {
      // insufficient data read
      return;
   }

   maskBytes = buf.slice(idx, idx+4);
   idx += 4;
   var payload = buf.slice(idx, idx+length);
   payload = unmask(maskBytes, payload);
   this._handleFrame(opcode, payload);

   this.buffer = buf.slice(idx+length);
   return true;
}

WebSocketConnection.prototype._handleFrame = function(opcode, buffer) {
var payload;
switch (opcode) {
   case opcodes.TEXT:
   payload = buffer.toString("utf8");
  this.emit("data", opcode, payload);
   break;
```

```
    case opcodes.BINARY:
    payload = buffer;
    this.emit("data", opcode, payload);
    break;
    case opcodes.PING:
    // Respond to pings with pongs
    this._doSend(opcodes.PONG, buffer);
    break;
    case opcodes.PONG:
    // Ignore pongs
     break;
     case opcodes.CLOSE:
    // Parse close and reason
    var code, reason;
    if (buffer.length >= 2) {
    code = buffer.readUInt16BE(0);
    reason = buffer.toString("utf8",2);
     }
     this.close(code, reason);
     this.emit("close", code, reason);
     break;
     default:
     this.close(1002, "unknown opcode");
     }
}

// Format and send a WebSocket message

WebSocketConnection.prototype._doSend = function(opcode, payload) {
   this.socket.write(encodeMessage(opcode, payload));
}

var KEY_SUFFIX = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";
var hashWebSocketKey = function(key) {
   var sha1 = crypto.createHash("sha1");
   sha1.update(key+KEY_SUFFIX, "ascii");
   return sha1.digest("base64");
}

var unmask = function(maskBytes, data) {
   var payload = new Buffer(data.length);
   for (var i=0; i<data.length; i++) {
      payload[i] = maskBytes[i%4] ^ data[i];
   }
   return payload;
}

var encodeMessage = function(opcode, payload) {
   var buf;
   // first byte: fin and opcode
   var b1 = 0x80 | opcode;
   // always send message as one frame (fin)

// Second byte: mask and length part 1
```

```
// Followed by 0, 2, or 8 additional bytes of continued length
var b2 = 0; // server does not mask frames
var length = payload.length;
if (length<126) {
   buf = new Buffer(payload.length + 2 + 0);
   // zero extra bytes
   b2 |= length;
   buf.writeUInt8(b1, 0);
   buf.writeUInt8(b2, 1);
   payload.copy(buf, 2);
} else if (length<(1<<16)) {
   buf = new Buffer(payload.length + 2 + 2);
   // two bytes extra
   b2 |= 126;
   buf.writeUInt8(b1, 0);
   buf.writeUInt8(b2, 1);
   // add two byte length
   buf.writeUInt16BE(length, 2);
   payload.copy(buf, 4);
} else {
   buf = new Buffer(payload.length + 2 + 8);
   // eight bytes extra
   b2 |= 127;
   buf.writeUInt8(b1, 0);
   buf.writeUInt8(b2, 1);
   // add eight byte length
   // note: this implementation cannot handle lengths greater than 2^32
   // the 32 bit length is prefixed with 0x0000
   buf.writeUInt32BE(0, 2);
   buf.writeUInt32BE(length, 6);
   payload.copy(buf, 10);
 }
return buf;
}

exports.listen = function(port, host, connectionHandler) {
   var srv = http.createServer(function(req, res) {
});

srv.on('upgrade', function(req, socket, upgradeHead) {
   var ws = new WebSocketConnection(req, socket, upgradeHead);
   connectionHandler(ws);
});

srv.listen(port, host);
};
```

3.4.2　测试简单的 WebSocket 服务器

现在，我们测试自己的服务器。回显（echo）是网络中的“Hello, World”，所以我们用新的服务器 API 做的第一件事是创建

一个服务器，如代码清单 3-7 所示。回显服务器简单地响应连接客户端所发送的任何信息。在这个例子中，我们的 WebSocket 回显服务器将响应接收到的任何 WebSocket 消息。

代码清单 3-7　用你的新服务器 API 构建一个回显服务器

```
var websocket = require("./websocket-example");

websocket.listen(9999, "localhost", function(conn) {
   console.log("connection opened");
   conn.on("data", function(opcode, data) {
      console.log("message: ", data);
      conn.send(data);
   });

   conn.on("close", function(code, reason) {
      console.log("connection closed: ", code, reason);
   });
});
```

可以用 node 在命令行上启动这个服务器。一定要将 websocket-example.js 放在同一目录下（或者作为模块安装）。

```
> node echo.js
```

以后，如果从浏览器发起对一个回显服务器的连接，就会看到从客户端发出的每条消息都被服务器返回。

说明　当服务器监听本地主机时，浏览器必须在同一台机器上。也可以用第 2 章的回显客户端示例尝试。

3.4.3　构建远程 JavaScript 控制台

JavaScript 最好的特征是对交互式开发的改善。Chrome 开发工具和 Firebug 内置的控制台是 JavaScript 开发如此高效的原因之一。控制台也称为 REPL（Read Eval Print Loop，读取 – 求值 – 打印循环），你可以输入一个表达式查看结果。我们将利用 Node.js 的 `repl` 模块，添加一个自定义的 `eval()` 函数。添加 WebSocket，我们可以通过互联网远程控制一个 Web 应用程序。使用这个 WebSocket 驱动的控制台，我们将能从一个命令行接口远程求取表达式的值。更好的是，我们可以输入一个表达式，并看到每个并

发连接的客户端求得的表达式值。

在这个例子中，你将使用代码清单 3-6 所示的同一个服务器，然后构建两个小的片段：一个用作远程控制，另一个作为你控制的对象。图 3-8 展示了下一个例子达到的效果。

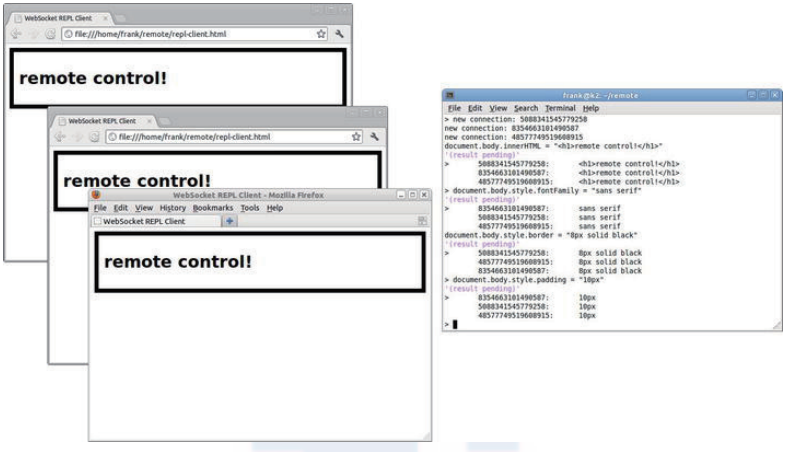

图 3-8　远程 JavaScript 控制台

在你构建这个例子之前，一定要完成代码清单 3-6 中示例的构建。如果你还构建了回显服务器（见代码清单 3-7），则必须在测试下面的代码片段之前将其关闭。代码清单 3-8 包含了远程控制所用的 JavaScript 代码。

代码清单 3-8　websocket-repl.js

```
var websocket = require("./websocket-example");
var repl = require("repl");

var connections = Object.create(null);

var remoteMultiEval = function(cmd, context, filename, callback) {
   for (var c in connections) {
   connections[c].send(cmd);
   }
   callback(null, "(result pending)");
}

websocket.listen(9999, "localhost", function(conn) {
   conn.id = Math.random().toString().substr(2);
```

```
    connections[conn.id] = conn;
    console.log("new connection: " + conn.id);

    conn.on("data", function(opcode, data) {
    console.log("\t" + conn.id + ":\t" + data);
    });
    conn.on("close", function() {
    // remove connection
    delete connections[conn.id];
    });
});

repl.start({"eval": remoteMultiEval});
```

我们还需要一个简单的网页来进行控制。这个页面上的脚本简单地打开一个到控制服务器的连接，计算接收到的任何消息，并响应结果。这个客户端还将在 JavaSript 控制台上记录输入的表达式。如果你打开浏览器的开发人员工具，就会看到这些表达式。代码清单 3-9 展示了包含脚本的网页。

代码清单 3-9　repl-client.html

```
<!doctype html>
<title>WebSocket REPL Client</title>
<meta charset="utf-8">
<script>
var url = "ws://localhost:9999/repl";
var ws = new WebSocket(url);
ws.onmessage = function(e) {
   console.log("command: ", e.data);
   try {
      var result = eval(e.data);
      ws.send(result.toString());
   } catch (err) {
      ws.send(err.toString());
   }
}
</script>
```

现在，如果你运行 node websocket-repl.js，就会看到一个交互式解释器。如果你在两个浏览器中加载 repl-client.html，就会看到每个浏览器计算你的命令。代码清单 3-10 显示了两个表达式的输出——`navigator.userAgent` 和 `5+5`。

代码清单 3-10　控制台表达式输出

```
> new connection: 5206121257506311
new connection: 6689629901666194
```

```
navigator.userAgent
'(result pending)'
>     5206121257506311:    Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:13.0)
Gecko/20100101 Firefox/13.0.1
   6689629901666194:    Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.1
(KHTML, like Gecko) Chrome/21.0.1180.15 Safari/537.1
5+5
'(result pending)'
>     6689629901666194:    10
   5206121257506311:    10
```

3.4.4　扩展建议

远程 JavaScript 控制台是某些有趣项目很好的起点。下面是扩展这一示例的几种途径。

- 为远程控制台创建一个 HTML5 用户界面。使用 WebSocket 在用户界面和控制服务器之间通信。考虑与 HTTP/AJAX 之类的通信策略相比，使用套接字如何简化管道命令的发送以及延迟响应的接收。
- 读完第 5 章后，可以将远程控制服务器修改为使用 STOMP。你可以用一个主题，向每个连接的浏览器会话广播命令，并接收队列上的回复。考虑将远程控制服务等新功能组合到消息驱动应用程序的方法。

3.5　小结

本章中，我们研究了互联网和协议的简史，以及创建 WebSocket 协议的原因。我们详细研究了 WebSocket 协议，包括线路流量、初始和关闭握手以及帧格式。我们使用 Node.js 构建了一个示例 WebSocket 服务器，驱动一个简单的回显演示和一个远程控制台。本章提供了 WebSocket 协议的一个很好的概览，你可以在 http://tools.ietf.org/html/rfc6455 上看到完整的协议规范。

在下一章中，我们将使用 WebSocket 之上的高层协议构建特性丰富的实时应用程序。