



Professional XMPP
Programming with JavaScript and jQuery

XMPP高级编程 ——使用JavaScript和jQuery



(美) Jack Moffitt 著
杨明军 译

清华大学出版社

利用强大的XMPP协议快速创建实时的、高度交互式应用程序

XMPP是一个广泛用于即时通信、多用户聊天、语音和视频会议、协作空间、实时游戏、数据同步以及搜索领域的健壮协议。《XMPP高级编程——使用JavaScript和jQuery》将教您如何在自己的应用程序中发挥XMPP的强大威力，并向您展示如何利用XMPP构建下一代应用程序或向当前应用程序中添加新功能所需的所有工具。本书的特色是采用JavaScript语言进行讲解，并使用了jQuery库，书中的几个XMPP应用程序的复杂性随着内容展开而逐渐变大，它们是帮助学习的理想工具。

主要内容

- ◆ 学习XMPP的即时通信功能，比如花名册、出席和订阅以及个人聊天
- ◆ 涵盖XMPP节、节错误消息以及客户端协议语法和语义
- ◆ 讨论服务发现、数据表单以及发布-订阅协议
- ◆ 解决XMPP编程相关的主题，比如应用程序设计、事件处理以及现有协议的组合
- ◆ 详细讨论了如何部署基于XMPP的应用程序
- ◆ 讲解如何使用Strophe的插件系统以及如何创建自己的插件

作者简介

Jack Moffitt是Collecta公司的CTO，他在该公司领导一个团队致力于多个XMPP相关项目的研发，包括Strophe(一系列用于XMPP通信的库)、Palaver(群聊服务器)、Punjab(HTTP-XMPP网关服务)、Speeqe(简单的基于Web的群聊客户端)。他还在XSF董事会和XSF委员会中出任多个职位。

Wrox Professional guides are planned and written by working programmers to meet the real-world needs of programmers, developers, and IT professionals. Focused and relevant, they address the issues technology professionals face every day. They provide examples, practical solutions, and expert education in new technologies, all designed to help programmers do a better job.

源代码下载及技术支持

<http://www.wrox.com>

<http://www.tupwk.com.cn/downpage>

Wrox
An Imprint of
 WILEY

上架建议：网络编程/XMPP、实时通信
读者信箱：wkservice@vip.163.com
投稿信箱：bookservice@263.net



Wrox.com

Programmer Forums

Join our Programmer to Programmer forums to ask and answer programming questions about this book, join discussions on the hottest topics in the industry, and connect with fellow programmers from around the world.

Code Downloads

Take advantage of free code samples from this book, as well as code samples from hundreds of other books, all ready to use.

Read More

Find articles, ebooks, sample chapters and tables of contents for hundreds of books, and more reference resources on programming topics that matter to you.

ISBN 978-7-302-25630-4



9 787302 256304 >

定价：58.00元

XMPPT 高级编程—— 使用 JavaScript 和 jQuery

(美) Jack Moffitt 著

杨明军 译

清华大学出版社

北京



Jack Moffitt

Professional XMPP Programming with JavaScript and jQuery

EISBN: 978-0-470-54071-8

Copyright © 2010 by Wiley Publishing, Inc.

All Rights Reserved. This translation published under license.

本书中文简体字版由 Wiley Publishing, Inc. 授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字: 01-2010-5494

本书封面贴有 Wiley 公司防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话: 010-62782989 13701121933

图书在版编目(CIP)数据

XMPP 高级编程——使用 JavaScript 和 jQuery/ (美) 莫菲特(Moffitt,J.) 著；杨明军 译.

—北京：清华大学出版社，2011.6

书名原文：Professional XMPP Programming with JavaScript and jQuery

ISBN 978-7-302-25630-4

I. X… II. ①莫… ②杨… III. 网络通信—程序设计 IV. TN915

中国版本图书馆 CIP 数据核字(2011)第 096085 号

责任编辑：王军于平

装帧设计：孔祥丰

责任校对：胡雁翎

责任印制：何莘

出版发行：清华大学出版社

<http://www.tup.com.cn>

地 址：北京清华大学学研大厦 A 座

邮 编：100084

社 总 机：010-62770175

邮 购：010-62786544

投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者：北京鑫丰华彩印有限公司

装 订 者：三河市新茂装订有限公司

经 销：全国新华书店

开 本：185×260 印 张：27.25 字 数：697 千字

版 次：2011 年 6 月第 1 版 印 次：2011 年 6 月第 1 次印刷

印 数：1~4000

定 价：58.00 元

产品编号：038094-01

前言

采用 XMPP 的应用程序范围广泛，包括即时通信、多用户聊天、语音和视频会议、协作空间、实时游戏、数据同步，甚至包括搜索。虽然 XMPP 起初是作为专有即时通信系统(比如 ICQ 和 AOL Instant Messenger)的开放的标准化替代技术，但它已经成为一个极为健全的、适用于各类扣人心弦的创新应用程序的协议。

Facebook 在其聊天系统中使用了 XMPP 技术。Google 则采用 XMPP 来架构 Google Talk 以及它的令人兴奋的新型 Google Wave 协议。Collecta 基于 XMPP 的发布-订阅系统构建了一个实时搜索引擎。有几款 Web 浏览器正在试验使用 XMPP 作为同步和共享系统的基础。还有许多公司在它们的 Web 应用程序中使用 XMPP 来提供增强的用户体验和实时交互。

XMPP 的核心是小型结构化信息块的交换。与 HTTP 类似，XMPP 是一种客户端-服务器协议，但它与 HTTP 的不同之处在于，它允许任何一端向另一端异步发送数据。XMPP 使用长连接，数据以推(而不是拉)的形式发送。

由于 XMPP 的不同，使得它成为 HTTP 极佳的补充协议。采用 XMPP 的 Web 应用程序将能够实现 AJAX 提供给静态网站的所有功能，而且它们将实现进一步的交互性和动态性。JavaScript 和动态 HTML 已经将桌面应用程序功能带到 Web 浏览器，而 XMPP 将为 Web 带来新型通信机制。

由于其即时通信传统，XMPP 内置了许多常见的社交 Web 功能。联系人列表和订阅机制建立了社交图，出席更新机制可帮助用户了解别人在做什么，而个人通信机制可以保护用户之间通信的私密性。XMPP 还拥有将近 300 种扩展，这为我们构建复杂应用程序提供了范围广泛的、实用的工具。只须使用核心协议以及这些扩展中的少数几个，我们就可以构建出神奇的应用程序。

本书讲解如何在自己的应用程序中利用 XMPP 来构建具有社交功能的、协作式的、实时的应用程序。我们将开发一系列逐渐变复杂的 XMPP 应用程序，从“Hello, World!”开始，最后将开发出一个协作式文本编辑器、一个共享的涂鸦板以及一个实时的、多玩家游戏。最终，您将拥有使用 XMPP 构建下一代应用程序或向当前应用程序中添加新型的实时、推送或社交功能所需的所有工具。

目标读者

本书是为那些对开发 XMPP 应用程序感兴趣的开发人员而编写的。读者不需要具备任何 XMPP 经验，但如果先前有过这方面的经验，那么当然会对学习有所帮助。本书假设读者已经对 XMPP 有所耳闻，并希望深入研究。

本书中的所有应用程序均采用 JavaScript 语言开发，这是因为它易于理解，而且许多程序员比较熟悉它，每台带有 Web 浏览器的计算机都支持它。虽然本书使用 JavaScript，但所有的

概念和应用程序都可以使用任何语言开发，绝大多数“核心部分”与编程语言、所用的库以及 Web 浏览器没有任何关系。理解和处理本书中的代码并不要求读者是一位 JavaScript 专家。

本书假设读者理解基本的前端 Web 技术，即 CSS 和 HTML。如果您曾经手工编写过少量的 HTML 并修改过了 CSS 样式属性，那么您已经具有足够的经验了。

本书还使用两个库，即 jQuery 和 Strophe。如果以前曾经使用过 jQuery，那么会有所帮助，但如果从未用过，那么可以参见附录 A 提供的一个简短入门指南。在开发应用程序的过程中，我们将充分讲解 Strophe 库。

涵盖内容

XMPP 协议及其扩展涵盖面非常广泛。本书关注那些得到广泛应用的 XMPP 部分。以下主题得到更多关注：

- XMPP 的即时通信功能，比如花名册、出席和订阅以及个人聊天
- XMPP 节、节错误消息、客户端协议语法和语义
- 扩展 XMPP 节
- 服务发现(XEP-0030)
- 数据表单(XEP-0004)
- 多人聊天(XEP-0045)
- 发布-订阅(XEP-0060)

尽管这些主题全部位于客户端，但几乎所有这些主题也同样适用于 XMPP 机器人或服务器组件和插件。

本书还涵盖 XMPP 编程相关的主题，比如应用程序设计、事件处理以及将简单的协议元素组合成一个更强大的整体。在这个过程中，我们还将讨论几个 Web 编程主题，比如 Canvas API。

XMPP 问世已经超过了 10 年，现在已经相当成熟。本书内容涵盖核心协议的 1.0 版。对于 XMPP 协议的未来版本，本书的 XMPP 协议部分应该仍然有效，就像 HTTP 1.0 客户端能够很容易与 HTTP 1.1 服务器通信一样。

XMPP 有很多扩展，本书也同样讨论了其中的一些扩展。本书所关注的大多是处于稳定成熟状态的扩展。对于每一个用到的扩展，给出了它们的文档编号，这样如果有疑问就可以检查该扩展的最新版本，看看是否已经改变或已被取代。

本书中的应用程序采用 jQuery 的 1.3 系列版和 jQuery UI 的 1.7 系列版。通常这些库在很大程度上保持向后兼容。此外本书使用了 Strophe 库的 1.0 版，但未来的 1.X 版也应该能够运行。

本书结构

本书内容主要按照一系列示例 XMPP 应用程序的指导教程来进行组织。每个应用程序在难度上逐渐增加，分别用来展示 XMPP 协议及其扩展的一个或多个有用部分。出于清晰考虑，这些应用程序经过缩减，但它们确实是 XMPP 开发人员每天建立的应用程序类型。

全书划分为 III 个部分。

第 I 部分介绍 XMPP 协议、它的用途以及 XMPP 应用程序的设计。第 1 章讲解了 XMPP 的用例、该协议的历史以及它的组成部分。第 2 章解释 XMPP 适用于哪些场合并深入研究了 XMPP 应用程序的工作原理，特别是在 Web 环境下。

第 II 部分是本书的重点，它包含了 9 个解决不同问题的 XMPP 应用程序。每个应用程序均要比前一个应用程序更加复杂，并且构建在前几个应用程序的概念之上。从第 3 章的简单的“Hello, World!”示例开始，最终在第 11 章中构建一个实时的、多玩家游戏。

第 III 部分讲解了几个高级的但重要的主题。第 12 章讨论会话接入，这是一项有关安全、优化和持久化方面的技术。第 13 章深入研究如何最佳地部署和扩展基于 XMPP 的应用程序。第 14 章讲解如何使用 Strophe 的插件系统以及如何创建自己的插件。

使用本书的前提条件

本书使用 Web 技术，因此几乎不需要任何特殊的工具。我们可以在任何平台上使用、构建和运行本书中的应用程序。第 3 章讲解这些应用程序所需的库，而且大多数均可以使用而无须下载任何代码。

还将需要一种提供网页服务的方式，比如本地 Web 服务器或托管账号。如果不方便采用这些方式，那么可以使用 Tape 程序来提供文件服务。Tape 是一种简单的 Web 服务器，我们将在附录 B 中讲解。由于浏览器安全策略的需求，我们无法轻易地直接从本地文件系统中运行这些应用程序。

为了运行这些应用程序，还需要一个 XMPP 账号(如果要自行测试代码，那么在某些情况下还需要多个账号)。虽然可以借助任何公共 XMPP 服务器来完成这个任务，但还需要确保该服务器支持发布-订阅和多用户聊天功能，大多数服务器均支持这些功能。此外，还可以下载并运行自己的 XMPP 服务器，但本书并没有讲解这方面的内容。

第 12 章还需要在服务器端执行一些操作。该示例使用 Python 编程语言以及 Django 框架来完成这些操作。该章属于高级主题，本书中的普通应用程序并不需要它。

源代码

在研读本书示例的过程中，可以选择采用手工方式录入所有代码，也可以选择本书所附的源代码文件。本书中所用的所有源代码均可以从 <http://www.wrox.com> 和 <http://www.tupwk.com.cn/downpag> 下载。在该网站上，只要搜索本书的书名(可以通过搜索栏或使用书名列表)，然后在本书详细信息页面上单击 Download Code 链接来获取本书的源代码。

因为许多书籍都有着相似的书名，所以最简单的方式是按照 ISBN 搜索。本书的 ISBN 是 978-0-470-54071-8。

在下载代码之后，使用解压缩工具解压。或者，可以打开 Wrox 主代码下载页面 <http://www.wrox.com/dynamic/books/download.aspx>，查看本书以及所有其他 Wrox 书籍的代码。

勘误表

尽管我们已经尽了各种努力来保证文章或代码中不出现错误，但是错误总是难免的，如果您在本书中找到了错误，例如拼写错误或代码错误，请告诉我们，我们将非常感激。通过勘误表，可以让其他读者避免受挫，当然，这还有助于提供更高质量的信息。

请给 wkservice@vip.163.com 发电子邮件，我们就会检查您的反馈信息，如果是正确的，我们将在本书的后续版本中采用。

要在网站上找到本书的勘误表，可以登录 <http://www.wrox.com>，通过 Search 工具或书名列表查找本书，然后在本书的细目页面上，单击 Book Errata 链接。在这个页面上可以查看 Wrox 编辑已提交和粘贴的所有勘误项。完整的图书列表还包括每本书的勘误表，网址是 <http://www.wrox.com/misc-pages/booklist.shtml>。

p2p.wrox.com

P2P 邮件列表是为作者和读者之间的讨论而建立的。读者可以在 p2p.wrox.com 上加入 P2P 论坛。该论坛是一个基于 Web 的系统，用于传送与 Wrox 图书相关的信息和相关技术，与其他读者和技术用户交流。该论坛提供了订阅功能，当论坛上有新帖子时，会给您发送您选择的主题。Wrox 作者、编辑和其他业界专家和读者都会在这个论坛上进行讨论。

在 <http://p2p.wrox.com> 上有许多不同的论坛，帮助读者阅读本书，在读者开发自己的应用程序时，也可以从这个论坛中获益。要加入这个论坛，必须执行下面的步骤：

- (1) 进入 p2p.wrox.com，单击 Register 链接。
- (2) 阅读其内容，单击 Agree 按钮。
- (3) 提供加入论坛所需的信息及愿意提供的可选信息，单击 Submit 按钮。
- (4) 然后就可以收到一封电子邮件，其中的信息描述了如何验证账户，完成加入过程。

提示：

不加入 P2P 也可以阅读论坛上的信息，但只有加入论坛后，才能发送自己的信息。

加入论坛后，就可以发送新信息，回应其他用户的帖子。可以随时在 Web 上阅读信息。如果希望某个论坛给自己发送新信息，可以在论坛列表中单击该论坛对应的 Subscribe to this Forum 图标。

对于如何使用 Wrox P2P 的更多信息，可阅读 P2P FAQ，了解论坛软件的工作原理，以及许多针对 P2P 和 Wrox 图书的常见问题解答。要阅读 FAQ，可以单击任意 P2P 页面上的 FAQ 链接。

目 录

第 I 部分 XMPP 协议和架构	
第 1 章 了解 XMPP 协议	3
1.1 什么是 XMPP	3
1.2 XMPP 简史	5
1.3 XMPP 网络	6
1.3.1 服务器	6
1.3.2 客户端	6
1.3.3 组件	6
1.3.4 插件	7
1.4 XMPP 寻址	7
1.5 XMPP 节	8
1.5.1 通用属性	9
1.5.2 presence 节	10
1.5.3 message 节	12
1.5.4 IQ 节	14
1.5.5 error 节	15
1.6 连接生命周期	16
1.6.1 连接	17
1.6.2 流的建立	17
1.6.3 身份验证	18
1.6.4 连接断开	18
1.7 小结	19
第 2 章 设计 XMPP 应用程序	21
2.1 他山之石	21
2.2 XMPP 与 HTTP 的比较	24
2.2.1 XMPP 的优势	24
2.2.2 XMPP 的不足	25
2.3 桥接 XMPP 与 Web	27
2.3.1 长轮询	27
2.3.2 管理连接	28
2.3.3 让 JavaScript 理解 XMPP 协议	28
2.4 构建 XMPP 应用程序	29
2.4.1 浏览器平台	29
2.4.2 基本的基础设施	29
2.4.3 协议设计	30
2.5 小结	31
第 II 部分 应用程序	
第 3 章 Hello World: 第一个应用程序	35
3.1 应用程序预览	35
3.2 Hello 应用程序设计	36
3.3 准备	36
3.3.1 jQuery 与 jQuery UI	37
3.3.2 Strophe	38
3.3.3 flexHR	38
3.3.4 XMPP 账户	38
3.4 开始构建第一个应用程序	39
3.4.1 用户界面	39
3.4.2 应用程序代码	40
3.5 建立连接	43
3.5.1 连接生命周期	43
3.5.2 创建连接	43
3.5.3 连接 Hello	44
3.5.4 运行应用程序	48
3.6 创建节	48
3.6.1 Strophe 构建器	49
3.6.2 打招呼	51
3.7 处理事件	52
3.7.1 添加和删除处理器	52
3.7.2 节匹配	52
3.7.3 节处理器函数	53

3.7.4 处理 Hello 响应	54	第 6 章 与好友交谈：一对聊天	99
3.8 给 Hello 程序添加新功能	57	6.1 应用程序预览	99
3.9 小结	57	6.2 Gab 的设计	100
第 4 章 探索 XMPP 协议：一个调试		6.2.1 出席	100
控制台	59	6.2.2 消息	101
4.1 应用程序预览	59	6.2.3 聊天区域	101
4.2 设计 Peek	60	6.2.4 花名册区域	101
4.3 构建控制台	61	6.3 制作界面	101
4.3.1 用户界面	61	6.4 构建花名册	105
4.3.2 显示流量	63	6.4.1 请求花名册	106
4.3.3 美化 XML	66	6.4.2 处理 IQ	109
4.3.4 处理 XML 输入	69	6.4.3 更新出席状态	110
4.3.5 简化输入	71	6.4.4 添加新联系人	111
4.4 研究 XMPP	76	6.4.5 响应花名册变化	112
4.4.1 控制出席	76	6.4.6 处理订阅请求	114
4.4.2 探测版本	77	6.5 构建聊天对话	117
4.4.3 处理错误	78	6.5.1 处理标签页	117
4.5 更好的调试	80	6.5.2 创建新的聊天对话	118
4.6 小结	80	6.5.3 发送消息	119
第 5 章 实时微博：一个 Identica		6.6 即时通信最佳实践	122
客户端	83	6.6.1 理解消息路由	122
5.1 应用程序预览	83	6.6.2 更好地寻址消息	123
5.2 Arthur 的设计	84	6.7 添加活动通知	125
5.3 Identica 微博	84	6.7.1 理解聊天状态	125
5.3.1 建立账户	85	6.7.2 发送通知	126
5.3.2 开启 XMPP	85	6.7.3 接收通知	127
5.4 构建 Arthur	86	6.8 收尾工作	128
5.4.1 开始	86	6.9 更多 Gab 功能	138
5.4.2 接收消息	89	6.10 小结	138
5.5 XHTML-IM	90	第 7 章 探索服务：服务发现与浏览	141
5.5.1 将 XHTML-IM 添加到 Arthur	91	7.1 应用程序预览	141
5.5.2 发送消息	91	7.2 Dig 的设计	142
5.6 离线消息	93	7.3 查找信息	142
5.7 创建更好的微博客户端	96	7.3.1 disco#info 查询	142
5.8 小结	97	7.3.2 disco#items 查询	144

7.4	创建 Dig	145	9.2.1	一切都是 pubsub	196
7.4.1	初始 disco 查询	148	9.2.2	展示者的流程	197
7.4.2	浏览 disco 树	151	9.2.3	观众的流程	197
7.5	挖掘服务	156	9.3	填写表单	198
7.5.1	查找代理服务	156	9.3.1	Data Forms 扩展	198
7.5.2	发现功能	157	9.3.2	表单元素、字段和类型	199
7.5.3	寻找聊天对话	157	9.3.3	标准化的表单字段	201
7.6	服务发现的更多功能	158	9.4	处理 pubsub 节点	202
7.7	小结	158	9.4.1	创建节点	202
第 8 章	群聊：多人聊天客户端	159	9.4.2	配置节点	204
8.1	应用程序预览	159	9.4.3	pubsub 事件	206
8.2	Groupie 的设计	160	9.4.4	发布到节点	206
8.3	公开发言	161	9.4.5	订阅和退订	207
8.3.1	群聊服务	161	9.4.6	检索订阅情况	210
8.3.2	进入和离开房间	161	9.4.7	获取项	210
8.3.3	发送和接收消息	164	9.4.8	订阅管理	212
8.3.4	匿名性	165	9.5	使用 pubsub 广播绘图	213
8.3.5	创建房间	165	9.5.1	构建用户界面	214
8.3.6	理解角色和岗位	166	9.5.2	使用 Canvas 绘制草图	216
8.4	构建界面	168	9.5.3	登录并建立节点	219
8.5	加入房间	172	9.5.4	发布和接收绘图事件	225
8.6	处理出席和消息	175	9.6	改进 SketchPad	240
8.6.1	处理房间消息	176	9.7	小结	240
8.6.2	跟踪出席状态变化	178	第 10 章	与好友一同写作：协作式	
8.6.3	聊天历史	179	文本编辑器	243	
8.6.4	保持私密性	180	10.1	应用程序预览	243
8.6.5	描述动作	182	10.2	NetPad 的设计	244
8.7	管理房间	184	10.3	操作转换	245
8.7.1	更换主题	184	10.3.1	基本原理	245
8.7.2	处理麻烦制造者	185	10.3.2	算法细节	246
8.7.3	招募管理员	187	10.3.3	实现	248
8.8	改进 Groupie	194	10.4	扩展 XMPP 协议	258
8.9	小结	194	10.4.1	忽略未知数据	258
第 9 章	发布与订阅：共享画板简介	195	10.4.2	XML 命名空间	258
9.1	SketchCast 预览	196	10.4.3	扩展元素	259
9.2	SketchCast 的设计	196	10.4.4	扩展属性	261

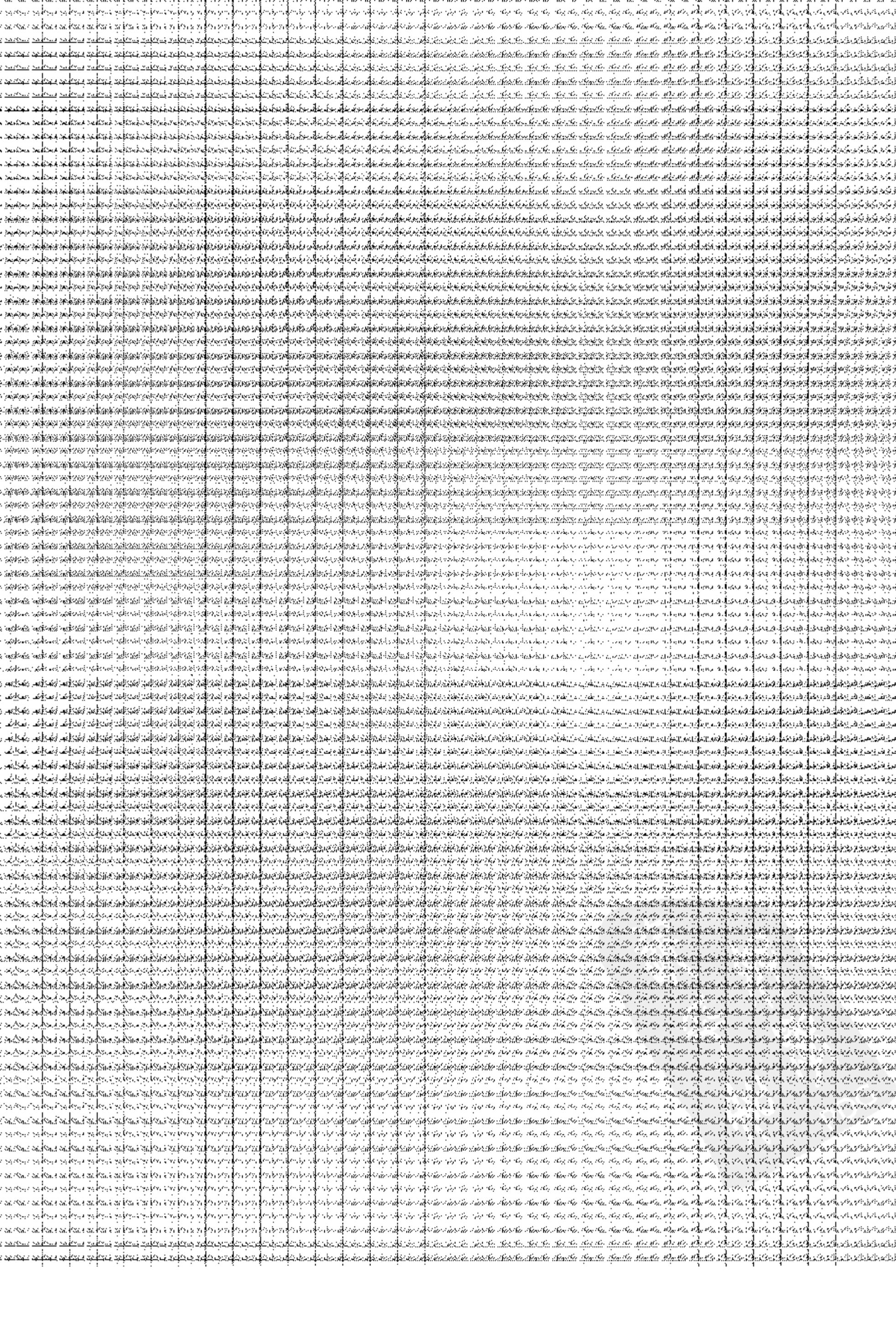
10.4.5 贡献扩展	261
10.5 设计协议	262
10.5.1 测试支持	262
10.5.2 请求和控制会话	263
10.5.3 编辑操作	264
10.6 构建编辑器	265
10.6.1 初始骨架	265
10.6.2 启动编辑会话	268
10.6.3 谈论工作	275
10.6.4 进行编辑	278
10.7 扩展 NetPad	288
10.8 小结	288
第 11 章 玩游戏：面对面的	
Tic-Tac-Toe	291
11.1 应用程序预览	291
11.2 Toetem 的设计	293
11.3 设计游戏协议	294
11.3.1 跟踪用户	294
11.3.2 管理玩家	295
11.3.3 管理游戏	297
11.3.4 玩游戏和观看游戏	299
11.4 Toetem 起步	302
11.5 实现会话和等待列表	308
11.5.1 裁判员(版本 1)	308
11.5.2 Toetem 客户端(版本 1)	313
11.6 实现游戏管理	316
11.6.1 裁判员(版本 2)	316
11.6.2 Toetem 客户端(版本 2)	324
11.7 实现游戏逻辑	329
11.7.1 Tic-Tac-Toe 库	329
11.7.2 裁判员(版本 3)	333
11.7.3 Toetem 客户端(版本 3)	336
11.8 让游戏更有趣	363
11.9 小结	363

第 III 部分 高级主题分构	
第 12 章 加入已有会话：引导 BOSH	367
12.1 会话接入	367
12.1.1 会话技术	368
12.1.2 用例	369
12.2 利用会话接入实现自动	
登录	370
12.3 小结	374
第 13 章 部署 XMPP 应用程序	375
13.1 横向扩展	375
13.1.1 多个连接管理器	376
13.1.2 XMPP 服务器集群	378
13.1.3 扩展组件	380
13.1.4 内部联合	380
13.1.5 成为服务器	380
13.2 纵向扩展	381
13.2.1 减少延迟	381
13.2.2 尽量减少 XML 序列化	383
13.2.3 优化 DOM 操作	384
13.3 小结	385
第 14 章 编写 Strophe 插件	387
14.1 使用插件	388
14.1.1 加载插件	388
14.1.2 访问插件功能	388
14.2 构建插件	389
14.3 创建花名册插件	390
14.3.1 保存联系人	390
14.3.2 获取并维护花名册	392
14.3.3 操纵花名册	397
14.4 试用插件	398
14.5 改进花名册插件	402
14.6 小结	402
附录 A jQuery 入门	405
附录 B 设置 BOSH 连接管理器	415

第 I 部分

XMPP协议和架构

- 第 1 章 了解 XMPP 协议
- 第 2 章 设计 XMPP 应用程序



第 1 章

了解 XMPP 协议

本章内容

- XMPP 历史
- XMPP 网络和连接
- XMPP 的三种构造块节

从最基本的层次来说, XMPP(eXtensible Messaging and Presence Protocol, 可扩展消息处理和现场协议)是一种用来在两个地点之间传递小型的结构化数据段的协议。在此基础上, XMPP 已被用来构建大规模即时通信系统、Internet 游戏平台、搜索引擎、协作空间以及语音和视频会议系统。每天都会出现更独特的应用程序, 这进一步说明了 XMPP 协议功能是如此多样和强大。

XMPP 由几个小的构造块组成, 并已经在这些原语的基础之上构建了许多更大的构造。在 XMPP 中有众多系统: 构建发布-订阅服务、多人聊天、表单检索和处理、服务发现、实时数据传输、隐私控制以及远程过程调用。XMPP 程序员通常会创建自己的、独特的、完全为手头上问题量身定做的构造。

大多数社交媒体构造(它们已经将网站(比如 Facebook、MySpace 和 Twitter)推向最前端)也都采用 XMPP 协议。在 XMPP 内部, 可以找到由联系人组成的花名册, 可以用它们来创建定向或无向社交图。当联系人上线或下线时自动发送出席通知, 而个人消息和公共消息是 XMPP 系统的基础应用。有时候开发人员之所以选择 XMPP 作为底层技术层只是因为它已经为开发人员提供了许多现成的社交功能, 他们只需要关注自己的应用程序的一些独特方面即可。

XMPP 为我们带来了无限的可能, 但在开始之前, 我们还需要了解 XMPP 的各种组成部分以及它们如何有机地构成一个整体。

1.1 什么是 XMPP

与所有协议一样, XMPP 定义了在两个或更多通信实体之间传递数据所采用的格式。对于 XMPP, 实体通常是客户端和服务器, 但它也允许两台服务器或两个客户端之间进行对等通信。Internet 上有许多 XMPP 服务器, 彼此之间能够相互访问, 从而构成了一个由互联系统组成的

联合网络。

在 XMPP 上交换的是 XML 数据，这让通信具有丰富的、可扩展的结构。为了获得更具人类可读性(因此更易于调试)这项更实用的功能，许多现代协议都放弃了二进制编码带来的带宽节省的优势。XMPP 选择使用 XML，这意味着，它能够充分利用大量与处理 XML 有关的知识和支持软件。

XMPP 通过使用 XML 而获得的一项主要功能就是 XML 的可扩展性。向 XMPP 协议中添加能同时保持向前和向后兼容的新功能是一件极其简单的事情。在已向 XMPP Standards Foundation 注册的 200 个协议扩展中，这种可扩展性得到了大量运用，它为开发人员提供了丰富的、实际上没有限制的工具。

XML 主要以文档格式闻名，但在 XMPP 中，XML 数据被组织成一对流，每个流分别用于通信的一个方向。每个 XML 流均由一个开始元素、后跟 XMPP 节和其他顶级元素、然后是一个结束元素组成。每个 XMPP 节(可带有子元素和属性)均是该流的一级子元素。在 XMPP 连接末尾，这两个流形成了一对有效的 XML 文档。

XMPP 节构成了该协议的核心部分，而 XMPP 应用程序则关注如何发送和响应各种类型的节。节可能包含网络上其他实体的信息、类似于电子邮件的个人消息或为计算机处理而设计的结构化通信数据。下面给出了一个示例节。

```
<message to='elizabeth@longbourn.lit'  
        from='darcy@pemberley.lit/dance'  
        type='chat'>  
    <body>What think you of books?</body>  
</message>
```

在一个典型的客户端-服务器 XMPP 会话中，一个与此类似的由 Elizabeth 发给 Darcy 的节将从 Elizabeth 的客户端传送到她的服务器。她的服务器将注意到该节的目的地是某个远程服务器上的一个实体，因此它与该远程服务器建立 XMPP 连接，并将该消息转发到该处。服务器之间的通信与电子邮件网络类似，但与电子邮件服务器不同的是，XMPP 服务器之间总是能够直接通信，而不需要借助中间服务器。

这种直接通信省去了垃圾消息和未经授权消息的常见传播媒介。这也是众多 XMPP 安全设计中的一环。它还支持通信端点之间通过使用 TLS(Transport Layer Security，传输层安全)来加密通信并通过 SASL(Simple Authentication and Security Layers，简单身份验证和安全层)实现强有力的身份验证机制。

XMPP 是为交换短小信息片段而设计的，而不是针对大型二进制数据块。但 XMPP 能够用来协商并建立可在端点之间传递大型数据块的带外或带内传输。对于这种传送而言，XMPP 相当于信令层。由于专注于小型结构化数据块，XMPP 协议的延迟极短，这使得它极其适用于实时应用程序。在开发人员试验实时 Web 时，这些应用程序(包括协作空间、游戏以及同步)正驱动着 XMPP 变得越来越流行。

我们将从本书中看到建立实时 Web 应用程序是多么简单的事情。当读完本书，读者应该对为什么如此多的人都被 XMPP 的强大功能所折服有一个透彻的理解。

1.2 XMPP 简史

XMPP 协议问世已经超过十年时间，从最初的简陋功能一路发展到今天，它已经取得了很大的进展。XMPP 的大部分设计均缘于 XMPP 创建时所处的环境，而 XMPP 的历史为我们了解开放协议如何被采用和创新提供了一个有趣的案例。

1996 年 Mirabilis 发布了 ICQ，这使得 Internet 用户之间能够进行快速的个人通信。ICQ 得以快速传播，而当时并没有其他公司发布类似的产品。1997 年 AOL 发布了 AOL Instant Messenger。Yahoo 也紧随其后于 1998 年发布了 Yahoo Pager(最终改名为 Yahoo Messenger)，而微软也最终于 1999 年凭借 MSN Messenger(现在的 Windows Live Messenger)加入竞争行列中。

所有这些即时通信应用程序都各自绑定到它们公司运营的专有协议和网络。ICQ 的用户不能与 Yahoo 用户交谈，反过来也一样。因此，常常出现这种情况：用户运行多种即时通信应用程序，这样才能够与自己的所有联系人聊天，因为尚无一家公司宣称占据 100% 的市场份额。

不久之后，开发人员期望为这些专有 IM 网络编写自己的客户端。有些开发人员希望编写能够联合两种或更多种 IM 网络的多协议客户端，还有一些开发人员希望将这些应用程序移植到微软的 Windows 以及 Apple 的 Mac OS 之外的操作系统。这些开发人员遇到了许多障碍，他们必须对这些没有文档资料的协议进行逆向工程，而 IM 网络则主动频繁地修改协议以挫败第三方开发人员的努力。

开放的、去中心化(decentralized)的 IM 网络和协议的思想正是在这样的气候中诞生了。

Jeremie Miller 于 1999 年 1 月发布了 Jabber 项目。Jabber 是一种基于 XML 的去中心化的即时通信协议，同时也是一个名为 jabberd 的服务器实现。立即形成了一个围绕该协议和实现的社区，并不断衍生出越来越多的客户端和新想法。在 2000 年 5 月，Jabber 的核心协议得以稳定，而 jabberd 被正式发布。

JSF(Jabber Software Foundation, Jabber 软件基金会)于 2001 年成立，旨在协调围绕 Jabber 协议及其实现而展开的各项工作。在 2002 年年底，JSF 已经向 IETF 标准化进程提交了核心协议规范，并成立了一个 IETF 工作小组。2004 年 10 月，这个标准化进程产生了 Jabber 协议的改进版，改名为 XMPP，其文档成为 RFC 标准，编号分别为 3920、3921、3922 和 3923。

在该协议的早期，开发人员持续通过向 JSF 提交协议扩展的方式来扩展它的可能性。这些扩展被称为 JEP(Jabber Extension Proposal, Jabber 扩展提议)。最终 JSF 及其扩展遵循 Jabber 到 XMPP 的名称改变，变成了 XSF(XMPP Standards Foundation, XMPP 标准基金会)和 XEP(XMPP Extension Proposal, XMPP 扩展提议)。

2005 年 XMPP 技术的大规模部署已经展开，比较突出的就是 Google Talk(Google 基于 XMPP 的 IM 服务)的发布。

现今 XMPP 生态系统已经相当庞大。已经有接近 300 种扩展被接受作为 XEP，并且创建了数十种客户端和服务器实现，既有商业产品也有开放源代码产品。实际上使用任何编程语言的软件开发人员都能够找到一个库来加速 XMPP 应用程序的开发进程。

XMPP 应用程序起初是以 IM 为中心的，这反映出它的出身，但开发人员已经发现 XMPP 能轻松胜任多种最初未能预见的应用程序，包括搜索引擎和同步软件。这种效用确实证明了开放系统和开放标准化进程的强大威力。

最近, IETF 已经组成了一个新的 XMPP 工作组来准备 XMPP 规范的下一版本, 把自从最初的 RFC 发布以来人们获取的所有认识整合起来。XMPP 不断得到精炼和扩展, 这样应用程序开发人员和 Internet 用户将总能够拥有一个开放的、去中心化的通信协议。

1.3 XMPP 网络

任何 XMPP 网络都是由若干角色组成的。这些角色可分为服务器、客户端、组件和服务器插件。XMPP 开发人员将编写代码来创建或修改这些角色类型中的一种。每种角色均在 XMPP 网络中发挥着各自的作用。

1.3.1 服务器

XMPP 服务器(更精确地说是使用服务器-服务器协议的 XMPP 实体或客户端-服务器协议中的服务器端)是任何 XMPP 网络的循环系统。服务器的任务就是为 XMPP 节提供路由, 无论这些节是从内部的一个用户发往另一个用户, 还是从本地用户发往远程服务器。

一组能够相互进行通信的 XMPP 服务器构成了一个 XMPP 网络。一组公开的 XMPP 服务器形成了全局的 XMPP 联合网络。如果某个服务器不能使用服务器-服务器协议, 那么它将变成一个孤岛, 不能与外部服务器通信。

XMPP 服务器总是允许用户连接到自己。但也可以编写直接使用服务器-服务器协议的应用程序或服务, 通过减轻路由开销来提高效率。

任何人都能够运行 XMPP 服务器, 而且几乎所有平台都有全功能的服务器可用。Ejabberd、Openfire 和 Tigase 是三种能够在 Windows、Mac OS X 和 Linux 系统上运行的流行的开放源代码服务器。还有几个商业 XMPP 服务器可用, 包括 M-Link 和 Jabber XCP。

1.3.2 客户端

大多数 XMPP 实体均是客户端, 它们通过客户端-服务器协议连接到 XMPP 服务器。这些实体中许多都是由人类驱动的、传统的 IM 用户, 但也有一些以机器人形式运行的自动化服务。

客户端必须向某个地方的 XMPP 服务器进行身份验证。服务器将该客户端发送的所有节路由到适当的目的地。服务器还负责管理客户端会话的其他几个方面, 包括它们的花名册以及裸地址, 稍后我们就会看到。

本书中的所有应用程序均被编写为客户端应用程序。这通常是大多数 XMPP 开发的起点。对于不需要用户参与或者有着苛刻要求的应用程序, 更好的做法通常是创建一个不同类型的实体, 比如服务器组件。

1.3.3 组件

并不只有客户端能够连接到 XMPP 服务器, 大多数服务器还支持外部服务器组件。这些组件通过添加某种新服务来增强服务器的行为。这些组件在服务器内部有各自的身份和地址, 但在外部运行并通过组件协议通信。

组件协议(由 XEP-0114 定义)可以让开发人员以一种服务器不可知的方式创建服务器扩展。

任何使用该协议的组件都能够在任何采用组件协议的服务器上运行(假设它不使用特定服务器的某些特殊功能)。一个典型的示例就是多人聊天服务，通常将这种服务作为组件来实现。

组件也需要向服务器进行身份验证，但这种验证要比客户端的完全的 SASL 验证简单。通常，可通过简单的口令来完成身份验证。

每个组件变成服务器内部的一个可单独寻址的实体，而在外界看来，它就像是一个子服务器。除了基本节之外，XMPP 服务器并不会代替已连接组件来管理其他节的路由。这给开发人员以极大的自由，他们可以完全按照自己的意愿完成工作，但当他们需要诸如花名册和出席管理这样的功能时，他们就要处理更多的工作。

服务器还允许组件在内部自行路由或管理节。因此组件创建可单独寻址的信息片段，比如房间、用户或开发人员需要的任何信息。这是客户端会话不能实现的功能，但可以用来建立真正优雅的服务。

最后，因为组件没有托管资源，所以那些涉及大量用户或很高流量的服务可以根据它们的作用来管理自己的资源。开发人员通常将服务作为客户端机器人，却不料后期发现服务器的花名册管理功能常常并不能很好地扩展以管理成千上万的联系人。组件可以管理花名册，并满足任务和伸缩性要求。

1.3.4 插件

有许多 XMPP 服务器还可以通过插件进行扩展。这些插件通常采用与服务器自身相同的编程语言编写，并在服务器的进程中运行。它们的作用在很大程度上与外部组件重叠，但插件还能够访问内部服务器数据结构并改变核心服务器行为。

服务器插件的功能实际上没有限制，但却是有代价的：插件不能在不同服务器之间移植。不同的服务器可能采用完全不同的语言编写，而它的内部数据结构也可能存在根本上的不同。撇开这个代价不论，插件有时候是完成特定任务的唯一方式。

与组件相比，插件已经减轻了开销，这是因为它们不需要通过网络套接字进行通信。它们也不需要分析或序列化 XML，而是可以直接操作各节的内部服务器表示。如果应用程序必须能够扩展，那么这可能会较大地提升所需的性能。

1.4 XMPP 寻址

XMPP 网络上的每个实体都有一个或多个地址(或称为 JID，jabber identifier)。JID 可以有多种形式，但它们通常看上去就像是电子邮件地址。比如 `darcy@pemberley.lit` 和 `elizabeth@longbourn.lit` 就是两个 JID。

每个 JID 由三部分组成：节点、域和资源。域部分总是必需的，但其他两部分是可选的，具体取决于它们所处的具体环境。

域是实体(服务器、组件或插件)的可解析 DNS 名称。仅由域组成的 JID 是有效地址，它表示服务器地址。指向域的节将由服务器自身处理，并可能被路由到某个组件或插件。

本地部分通常会识别域中的一个特定用户。它出现在 JID 的开头并位于域之前，它与 JID 剩余部分之间通过@字符隔开，就像是电子邮件地址的节点部分。本地部分还可以用来识别其

他对象，多人聊天服务将每个聊天室显示为一个 JID，而节点部分指向聊天室。

jid 的资源部分通常会标识一个特定的客户端 XMPP 连接。对于 XMPP 客户端而言，每个连接均被指派一个资源。如果 Darcy 先生(他的 JID 是 darcy@pemberley.lit)希望同时连接他的书房和图书馆，那么他的这些连接可以通过 darcy@pemberley.lit/study 和 darcy@pemberley.lit/library 来寻址。与本地部分一样，资源部分也可用来识别其他对象，在多人聊天服务中，jid 的资源部分被用来识别聊天室中的一个特定用户。

jid 划分为两种类型，即裸 jid 和完整 jid。完整 jid 总是特定实体的最具体的地址，而裸 jid 只是完整 jid 去除资源部分后的地址。例如，如果某个客户端的完整 jid 是 darcy@pemberley.lit/library，那么它的裸 jid 就是 darcy@pemberley.lit。在某些情况下，裸 jid 和完整 jid 是相同的，比如在寻址服务器或特定多人聊天室时。

客户端的裸 jid 有点特殊，这是因为服务器自己将处理发往客户端裸 jid 的节。例如，一条发往某个客户端的裸 jid 的消息将被转发到该用户的一个或多个已连接资源，如果该用户离线，就将该消息储存起来留待以后传送。但发送给完整 jid 的节通常会直接路由到该资源所在的客户端连接。可以将裸 jid 视为寻址用户的账户，而不是寻址该用户的某个已连接的客户端。

1.5 XMPP 节

在 XMPP 中，各项工作均是通过在一个 XMPP 流上发送和接收 XMPP 节来完成的。核心 XMPP 工具集由三种基本节组成。这些节分别为<presence>、<message>和<iq>。每种节都有各自的角色和用途，通过组合适当类型和数量的节就可以实现复杂的行为。

记住，XMPP 流由两份 XML 文档组成，通信的每个方向均有一份文档。这些文档有一个根元素<stream:stream>。这个<stream:stream>元素的子元素由可路由的节以及与流相关的顶级子元素构成。

每个节均是一个 XML 元素(包括它的子元素)。XMPP 通信的端点以节为单位来处理输入和产生输出。下面的示例给出了一段经过简化的简短的 XMPP 会话。

```

<stream:stream>
  <iq type='get'>
    <query xmlns='jabber:iq:roster' />
  </iq>

  <presence />

  <message to='darcy@pemberley.lit'
    from='elizabaeth@longbourn.lit/ballroom'
    type='chat'>
    <body>I cannot talk of books in a ball-room; my head is always full of
    something else.</body>
  </message>

  <presence type='unavailable' />
</stream:stream>

```

在这个示例中，Elizabeth 通过发送起始<stream:stream>标记创建了一个 XMPP 流。在打开

流之后，她将自己的第一节(一个`<iq>`元素)发送出去。这个`<iq>`元素请求 Elizabeth 的花名册，也就是她的所有已存储的联系人列表。接下来，她使用`<presence>`节来通知服务器她已在线并且可以访问。当注意到 Darcy 也在线时，她向他发送一条简短的`<message>`节，并拒绝他试图发起的聊天请求。最后，她又发送了一个`<presence>`节来告诉服务器自己不可访问并关闭`<stream:stream>`元素，然后结束会话。

现在我们已经看到每种 XMPP 节的实际使用示例。我们将更加详细地讨论每种 XML 节，但首先要学习它们都有哪些共有的属性。

1.5.1 通用属性

所有三种节均支持一组通用的属性。无论它们是`<presence>`、`<message>`或`<iq>`元素的属性，下面的属性的含义均相同。

1. from

XMPP 节几乎总拥有 `from` 属性。这个属性用于识别该节的起始 JID。并不建议在输出的节上手工设置 `from` 属性，服务器会在这些节通过时添加正确的 `from` 属性，而如果错误地设置 `from` 属性，那么服务器可能会拒绝整个节。

如果从客户端-服务器流中接收到的节上没有 `from` 属性，就意味着该节来源于服务器自身。而在服务器-服务器协议中，缺少 `from` 属性则被视为一种错误。

注意，本书中给出的示例节通常都包含 `from` 属性。这是出于清晰性和消除歧义的考虑。

2. to

XMPP 服务器把 XML 节发送到 `to` 属性中指定的 JID。与 `from` 属性类似，如果在客户端-服务器流中没有 `to` 属性，那么服务器将假设它是有意发给服务器自身的消息。建议在向服务器自身发送消息时忽略 `to` 属性。

如果 `to` 属性中指定的 JID 是一个用户，那么服务器有可能代表用户来处理该节。如果目的地是一个裸 JID，那么服务器将处理该节。对于三种不同的节类型来说，这种行为有所不同，我们在讲到每种类型时会逐一进行解释。如果目的地是一个完整 JID，那么服务器将直接把该节路由到该用户。

3. type

`type` 属性指定了`<presence>`、`<message>`或`<iq>`节的具体类型。对于 `type` 属性，这三种基本节均有几种可能的值，当详细讲解相关节的时候我们再具体阐述这些值。

所有三种节都可以将它们的 `type` 属性值设为 `error`。这表示该节是对已接收到的同一类型的节的错误响应。不要响应类型为 `error` 的节，以避免在网络上出现反馈循环。

4. id

可以为节指定 `id` 属性以辅助识别响应。对于`<iq>`节，这个属性是必需的，但对于其他两个节，该属性则是可选的。如果某节是为响应一个携带 `id` 属性的节而产生的，那么这个应答节必须包含一个携带相同值的 `id` 属性。

`id` 属性必须具有足够的唯一性，这样该节的发送者就可以使用它来甄别响应。通常，最简单做法是让 `id` 属性值在给定的流中保持唯一性，以避免出现任何歧义。

`<message>` 和 `<presence>` 节的应答节一般仅限于报告错误。`<iq>` 的应答节可以用来通知成功操作、确认命令或返回请求的数据。无论什么情况，客户端均可以使用应答节的 `id` 属性来识别与该节相关联的请求节。万一在一个短时间帧内发送大量相同类型的节，那么该功能将非常关键，这是因为这些节的应答可能会以乱序形式到达。

1.5.2 presence 节

`<presence>` 节控制并报告实体的可访问性。这里的可访问性涉及范围较广，既有简单的“在线”和“离线”，又有更复杂的“离开”和“请勿打扰”。此外，`<presence>` 节还用来建立和终止向其他实体发布出席订阅。

在传统的即时通信系统中，出席通知是主要的流量来源。为了能够进行即时通信，当一方可以进行通信时，另一方有必要获知该情况。当发送电子邮件时，发送者并不知道接收者当前是否正在查收并答复电子邮件，但借助即时消息和出席通知，他在发送消息之前就知道接收者是否在线。

对于其他领域中的应用程序，出席通知可用来通告相似类型的消息。例如，有些开发人员已经编写了机器人，当自己太忙不能接受更多的工作时，会将自己的出席状态设置为“请勿打扰”。基本的在线和离线状态可以让应用程序知道某项服务当前是否正常运行或因系统维护而下线。

1. 普通 presence 节

普通`<presence>` 节不含 `type` 属性，或者 `type` 属性的值为 `unavailable` 或 `error`。这些节为通信目的而设置，或提供某个实体的出席状态或可访问性。

`type` 属性没有 `available` 值，因为可以通过缺少 `type` 属性来指出这种情况。

用户通过发送不携带 `to` 属性、直接发往服务器的`<presence>` 节来操纵自己的出席状态。我们已经看过该节的两个简单示例，这些示例包含在下面几个更长的示例中。

```
<presence/>
<presence type='unavailable'/>
<presence>
  <show>away</show>
  <status>at the ball</status>
</presence>
<presence>
  <status>touring the countryside</status>
<priority>10</priority>
</presence>
<presence>
  <priority>10</priority>
</presence>
```

前两个节分别将用户的出席状态设置为在线和离线。这些节通常也是一个 XMPP 会话期间发送的第一个和最后一个 presence 节。

接下来的两个示例均通过<show>、<status>和<priority>子元素的形式来展示额外的出席信息。

<show>元素用来传达用户可访问性的性质。该元素之所以叫做 show 的原因是，它请求接收者的客户端使用这条消息来更新发送者出席状态的可视化指示器。<show>子元素只能用在<presence>节中，而且该元素只能包含如下几个可能的值：away、chat、dnd 和 xa。这些值分别用来传达该用户离开、有意聊天、不希望被打扰以及长期离开。

<status>元素是一个人类可读的字符串，用户可以将其设置为能够传达出席信息的任何值。在接收者的聊天客户端中，这个字符串一般会紧挨着联系人名字显示。

用户的每个已连接资源都有一个介于 -128~127 的优先级。该优先级默认值为 0，但可以通过在<presence>节中包含一个<priority>元素来设置它。同时拥有多个连接的用户可以使用这个优先级来指出哪一个资源应该接收到那些发往该用户裸 JID 的聊天消息。服务器将把这类消息传送给具有最高优先级的资源。负优先级有着特殊的含义：具有负优先级的资源绝不会接收到通过裸 JID 寻址方式传送给它们的消息。当人类用户正在进行常规聊天时，负优先级对于运行在同一个 JID 的自动化应用程序来说是极其有用的。

2. 扩展 presence 节

开发人员非常希望能够扩展<presence>节以包含更详细的信息，比如用户当前正在听的歌或个人的情绪。因为要将<presence>节广播给所有的联系人(即使它们可能对该信息并不感兴趣)，并且在 XMPP 网络流量中占据了很大的份额，所以不鼓励这种做法。这类扩展应该交给更紧密关注这种额外信息传递的协议来处理。

3. 出席订阅

用户的服务器会自动地将出席信息广播给那些订阅该用户出席信息的联系人。类似地，用户从所有他已经进行出席订阅的联系人那里接收到出席更新信息。出席订阅的建立和控制是通过使用<presence>节来完成的。

与一些社交网络和 IM 系统不同的是，在 XMPP 中，出席订阅是有方向的。如果 Elizabeth 订阅了 Darcy 的出席信息，那么这并不意味着 Darcy 也订阅了 Elizabeth 的出席信息。如果希望实现双向订阅，那么必须分别建立订阅。双向订阅通常是人类交流者的行为准则，但许多服务(甚至一些用户)只对其中一个方向感兴趣。

可以通过如下的值之一来识别出席订阅节：subscribe、unsubscribe、subscribed 或 unsubscribed。前两个值请求建立新的出席订阅或取消一个现有的订阅，而另外两个值则是对这类请求的应答。

下面的示例演示了 Elizabeth 和 Darcy 如何建立彼此的出席订阅。

```
<presence from='elizabeth@longbourn.lit/outside'
          to='darcy@pemberley.lit'
          type='subscribe'>

<presence from='darcy@pemberley.lit/library'
          to='elizabeth@longbourn.lit/outside'
          type='subscribed'>
```

```

<presence from='darcy@pemberley.lit/library'
          to='elizabeth@longbourn.lit'
          type='subscribe' />

<presence from='elizabeth@longbourn.lit/outside'
          to='darcy@pemberley.lit/library'
          type='subscribed' />

```

在交换节之后, Elizabeth 和 Darcy 都将发现对方已经出现在自己的花名册中, 并且会接到对方的出席更新信息。

第 6 章将研究一个相当传统的 IM 应用程序, 它能够建立和取消出席订阅, 并能够显示联系人的出席状态。

4. 定向出席

最后一种<presence>节是定向出席。定向出席节是一种直接发给另一个用户或其他某个实体的普通<presence>节。这些节可用来向那些没有进行出席订阅(通常因为只是临时需要出席信息)的实体传达出席状态信息。

定向出席的一个重要功能是, 当发送者变成不可访问状态时, 出席信息的接收者将自动得到通知, 即使发送者忘记显式地通知接收者。服务可以使用定向出席来临时地掌握某个用户的可访问性。

第 8 章将讲述定向出席, 因为对于多人聊天而言, 它是相当重要的。

1.5.3 message 节

<message>节用来从一个实体向另一个实体发送消息。这些消息可以是简单的聊天消息(就像读者熟悉的来自其他 IM 系统的消息), 但它们还可以用来传输任何类型的结构化信息。例如, 第 9 章中的 SketchCast 应用程序使用<message>节来传输绘制指令, 而在第 11 章中使用<message>节来传输游戏状态和新游戏变动情况。

<message>节属于“发射后不管”型, 没有内在的可靠性, 就像电子邮件消息一样。一旦消息已经发送出去, 发送者并不知道它是否传出去以及何时送达。在有些情况下(比如向不存在的服务器发送消息)发送者可能会接收到一个错误提示节, 他可以从中了解出现的问题。可以通过在应用程序协议中增加确认机制(有关这一点的示例请参见 XEP-0184 中的 Message Receipts)来实现可靠传送。

下面是<message>节的一些示例。

```

<message from='bingley@netherfield.lit/drawing_room'
          to='darcy@pemberley.lit'
          type='chat'>
  <body>Come, Darcy, I must have you dance.</body>
  <thread>4fd61b376fbc4950b9433f031a5595ab</thread>
</message>

<message from='bennets@chat.meryton.lit/mrs.bennet'
          to='mr.bennet@longbourn.lit/study'>

```

```

  type='groupchat'
<body>We have had a most delightful evening, a most excellent ball.</body>
</message>

```

第一个示例给出一个典型的个人聊天<message>节，其中包含一个线索标识符。第二个示例是一个多人聊天消息，Bennet夫人向 bennets@chat.meryton.lit 聊天室发送消息，Bennet先生接收到该消息。

1. 消息类型

<message>节有几种不同的类型。这些类型由 type 属性指出，而该属性的值可取 chat、error、normal、groupchat 或 headline。有时消息的类型用来提醒用户的客户端如何最佳地呈现该消息，但有些 XMPP 扩展(多人聊天就是最好的示例)使用 type 属性来区分上下文。

<message>节的 type 属性是可选的，但建议应用程序为其提供一个值。此外，任何应答<message>节应该将镜像接收到的 type 属性值。如果没有指定 type 属性，那么可以解释为这个<message>节的 type 属性被设为 normal。

chat 类型的消息是在一对一聊天对话上下文中发送的。这是那些主要关注私有的、一对一通信的 IM 应用程序中最常见的类型。

在答复某条导致错误的消息时可使用类型 error。在响应格式错误的寻址时通常会看到这些类型，向不存在的域或用户发送<message>节将产生一个 type 属性被设为 error 的应答节。

可在一对一聊天上下文之外发送 type 为 normal 的<message>节。这个类型实际上很少用到。

groupchat 类型用于在多人聊天中发送的消息。它用来区分多人聊天参与者发送的定向的私有消息与参与者发送给聊天室中所有人的广播消息。私有消息的 type 属性被设为 chat，而发送给聊天室中所有人的消息的 type 属性被设为 groupchat。

最后一种<message>节类型是 headline。大多数这些类型的消息被那些不希望或不支持应答的自动化服务使用。如果自动生成的电子邮件携带 type 属性，那么它的值将使用 headline 值。

2. 消息内容

尽管<message>节可以包含任意扩展元素，但<body>和<thread>元素是为向消息中添加内容而提供的正常机制。这两种子元素均是可选的。

<body>元素包含着该消息中人类可读的内容。可以包含多个<body>元素，只要它们包含不同的 xml:lang 属性即可，这可以让我们发送携带多种语言内容的<message>节。

对话(就像电子邮件一样)可以形成线索，线索中的每条消息都与相同的对话相关联。可以通过向<message>节中添加一个<thread>元素来创建线索。<thread>元素的内容是一个用来区分不同线索的唯一标识符。应答节应该包含与它所应答的节相同的<thread>元素。

在 IM(以及其他)上下文中，有几个常用的消息内容扩展。XHTML-IM(由 XEP-0071 定义)用来在消息中提供格式化、超链接以及富媒体。第 5 章的微博客户端 Arthur 使用 XHTML-IM 来提供增强的消息正文。另一个扩展 Chat State Notifications(XEP-0085)则允许用户通告对方自己正在撰写消息或有空闲。第 6 章中的 Gab 应用程序使用这些通知来提供很好的用户体验：当一方长时间输入时，接收方会被告知另一方仍然在积极地参与到该对话中。

1.5.4 IQ 节

<iq>节表示的是 Info/Query(信息与查询), 它为 XMPP 通信提供请求与响应机制。它与 HTTP 协议的基本工作原理非常相似, 允许获取和设置查询, 与 HTTP 的 GET 和 POST 动作类似。

每个<iq>节都必须有一个响应, 而且前面曾经提到过, 该节的必需的 id 属性将用来把响应与导致该响应的请求关联起来。<iq>节有四种, 通过该节的 type 属性区分。有两种<iq>节请求 (get 和 set)和两种响应(result 和 error)。在本书中, 这些节通常被缩写为 IQ-get、IQ-set、IQ-result 和 IQ-error。

每一个 IQ-get 或 IQ-set 节均必须接收响应的 IQ-result 或 IQ-error 节。下面的示例给出了一些常见的<iq>节以及它们可能的响应。注意, 与<message>和<presence>节(它们定义了子元素)不同, <iq>节通常只包含与它们功能相关的扩展元素。此外, 每一对<iq>节必须匹配 id 属性。

```
<iq from='jane@longbourn.lit/garden'
    type='get'
    id='roster1'>
  <query xmlns='jabber:iq:roster' />
</iq>

<iq to='jane@longbourn.lit/garden'
    type='error'
    id='roster1'>
  <query xmlns='jabber:iq:roster' />
  <error type='cancel'>
    <feature-not-implemented xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
  </error>
</iq>
```

Jane 向她的服务器发送了一个格式错误的花名册请求。服务器使用一个错误提示节作为响应。后面我们将详细讨论错误提示节。

```
<iq from='jane@longbourn.lit/garden'
    type='get'
    id='roster2'>
  <query xmlns='jabber:iq:roster' />
</iq>

<iq to='jane@longbourn.lit/garden'
    type='result'
    id='roster2'>
  <query xmlns='jabber:iq:roster'>
    <item jid='elizabeth@longbourn.lit' name='Elizabeth' />
    <item jid='bingley@netherfield.lit' name='Bingley' />
  </query>
</iq>
```

在重新发送正确的请求之后, 服务器将 Jane 的简短花名册响应给她。可以看到 Elizabeth 和 Bingley 均在 Jane 的联系人列表中。

```

<iq from='jane@longbourn.lit/garden'
  type='set'
  id='roster3'>
  <query xmlns='jabber:iq:roster'>
    <item jid='darcy@pemberley.lit' name='Mr. Darcy' />
  </query>
</iq>

<iq to='jane@longbourn.lit/garden'
  type='result'
  id='roster3' />

```

Jane 试图将 Darcy 添加到自己的花名册中, 服务器用一个空白 IQ-result 节来指出添加成功。如果应答节只是成功确认, 那么 IQ-result 节通常是空白的。

在任何需要结果数据或者需要简单确认的场合中, <iq>节都非常有用。大多数 XMPP 扩展协议混合使用<iq>和<message>节来实现它们的功能。<iq>节用于类似于配置和状态变化这样的信息, 而<message>节则用于常规通信。在某些场合中, <iq>节也用于通信, 这是因为节确认机制可实现限速功能。

1.5.5 error 节

所有三种 XMPP 节都有一个 error 类型, 而且错误提示节的每种类型的内容都是按照同一模式排列。错误提示节具有定义明确的结构, 通常包含原节(肇事节)的内容、通用错误信息以及应用程序特有的错误条件和信息(可选)。

所有错误提示节都必须将 type 属性设为 error, 而且必须携带一个<error>子元素。许多错误提示节还包含原节的内容, 但这并不是必需的, 而且在某些场合中这并不合适。

<error>子元素自身必须携带必要的 type 属性, 它可取值 cancel、continue、modify、auth 或 wait 之一。值 cancel 表示不应该重试该动作, 这是因为它总会失败。值 continue 通常代表一条警告信息, 这个值平时很少用到。错误类型 modify 表示发送的数据需要一些修改才会被接受。错误 auth 则通知实体在以某种方式进行身份验证之后重试该动作。最后, 值 wait 报告服务器临时遇到问题, 应该在稍后将原节原封不动地重新发送。

<error>子元素还必须包含一个错误条件(从已定义条件列表中选择一个)作为它的子元素。它还可以包含一个<text>元素来进一步指出有关该错误的详细信息。还可以在<error>元素的子元素中指定应用程序的特定错误条件(在它的命名空间里面)。

表 1-1 列出了最常见的已定义错误条件。有关这些错误条件的更多信息, 请参见 RFC 3920 的第 3.9.2 节。注意, 每个错误条件元素都必须位于 urn:ietf:params:xml:ns:xmpp-stanzas 命名空间里面。

表 1-1 常见的已定义错误条件

条件元素	描述
<bad-request/>	请求的格式错误或者包含非预期数据
<conflict/>	已存在另一个具有相同名称的资源或会话

(续表)

条件元素	描述
<feature-not-implemented/>	请求的功能尚未由服务实现
<forbidden/>	客户端没有得到授权来执行该请求
<internal-server-error/>	服务器遇到一个未定义的内部错误，阻止其处理该请求
<item-not-found/>	请求中涉及的项不存在；该错误等同于 HTTP 404 错误
<recipient-unavailable/>	预期的接收者临时不可访问
<remote-server-not-found/>	远程服务器不存在或不可到达
<remote-server-timeout/>	与远程服务器的通信已经被中断
<service-unavailable/>	系统并未提供请求的服务

下面这个 IQ-error 示例节给出了针对一个发布-订阅相关<iq>节的完整构造的错误提示响应。

```

<iq from='pubsub.pemberley.lit'
  to='elizabeth@longbourn.lit/sitting_room'
  type='error'
  id='subscribe1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <subscribe node='latest_books'
      jid='elizabeth@longbourn.lit'/'>
  </pubsub>
  <error type='cancel'>
    <not-allowed xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
    <closed-node xmlns='http://jabber.org/protocol/pubsub#errors' />
    <text xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'>
      You must be on the whitelist to subscribe to this node.
    </text>
  </error>
</iq>

```

该错误提示的类型是 `cancel`，该值表示不应该重试该动作，而条件<not-allowed/>指出了一个一般性故障。<text/>子元素包含了问题描述。最后，应用程序条件元素<closed-node/>给出了精确的应用程序错误提示信息。

1.6 连接生命周期

通过适当地组合我们已经学习的这三种节，我们实际上可以完成 XMPP 中的任何任务。但发送 XMPP 节通常要求建立一个经过身份验证的 XMPP 会话。本节描述了 XMPP 连接生命周期的其他部分，包括连接、流的建立、身份验证以及断开连接。

1.6.1 连接

在发送任何节之前，需要建立 XMPP 流。在 XMPP 流存在之前，必须建立通往 XMPP 服务器的连接。XMPP 为建立通往正确服务器的连接提供了复杂的支持。

通常，客户端和服务器利用域名系统(DNS)将服务器的域名解析成一个能够连接的地址。电子邮件服务特别地使用邮件交换台(MX)记录来提供处理特定域邮件的服务器列表，这样一来，一个知名服务器地址就不必处理每一项服务请求。电子邮件作为一个早期的 Internet 应用程序，它在 DNS 中得到特殊的对待。现如今，服务记录(SRV)可用来为任意服务提供类似的功能。

当 XMPP 客户端或服务器连接到另一个 XMPP 服务器时，它们要做的第一件事情就是在服务器域中查询适当的 SRV 记录。查询应答中可能包含多条 SRV 记录，这样就可以在多台服务器之间建立负载均衡连接。

如果未能找到一条合适的 SRV 记录，那么作为反馈，应用程序试着直接连接到给定域。大多数库也可以显式指定要连接的服务器。

1.6.2 流的建立

一旦建立通往给定 XMPP 服务器的连接，XMPP 流就启动了。通过向服务器发送起始元素<stream:stream>，就可打开 XMPP 流。服务器通过发送响应流的起始标记<stream:stream>进行响应。

一旦双向建立 XMPP 流，就可以来回发送各种元素。在连接生命周期的这个阶段，这些元素将与该流以及该流的功能有关。

服务器首先发送<stream:features>元素，详细列举 XMPP 流中支持的所有功能。这些功能大多数与可用的加密和身份验证选项有关。例如，服务器将指定加密(TLS)功能是否可用以及是否允许匿名登录。

通常并不需要知道有关这个 XMPP 连接阶段的太多细节，这是因为许多 XMPP 开发库将替用户处理这项工作，但下面的示例给出了典型的<stream:stream>元素交换以及服务器的功能列表。

首先，客户端向服务器发送起始元素。

```
<?xml version='1.0'?>
<stream:stream xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    version='1.0'
    to='pemberley.lit'>
```

服务器应答如下。

```
<?xml version='1.0'?>
<stream:stream xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    version='1.0'
    from='pemberley.lit'
    id='893ca401f5ff2ec29499984e9b7e8afc'>
```

```

 	xml:lang='en'>
<stream:features>
 	<stream:features>
<starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls' />
<compression xmlns='http://jabber.org/features/compress'>
 	<method>zlib</method>
</compression>
<mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
 	<mechanism>DIGEST-MD5</mechanism>
 	<mechanism>PLAIN</mechanism>
</mechanisms>
</stream:features>

```

根据这次交换，我们得知 pemberley.org 服务器支持 TLS、流压缩(通过 zlib)以及几种身份验证机制。

在两台服务器之间建立 XMPP 流的过程与此相同，但顶级命名空间是 `jabber:server` 而不是 `jabber:client`。

1.6.3 身份验证

XMPP 允许进行 TLS(Transport Layer Security, 传输层安全)加密，而且大多数客户端默认使用该功能。一旦服务器通告 TLS 支持后，客户端就启动 TLS 连接并将当前套接字升级为一个加密套接字而不断开连接。一旦 TLS 加密确立，就会创建一对新的 XMPP 流。

XMPP 中的身份验证使用 SASL(Simple Authentication and Security Layers, 简单身份验证与安全层)协议并支持多种身份验证机制(取决于涉及的服务器)。服务器通常提供明文验证和基于 MD5 摘要的验证，但有些服务器还支持通过 Kerberos 或特殊令牌来验证身份。

同样，这些加密和身份验证技术还用在许多其他协议中(比如电子邮件和 LDAP)，而且现有的一些用于支持 TLS 和 SASL 的公用库也同样可用于 XMPP。

一旦完成身份验证，客户端必须为该连接绑定一个资源并启动一个会话。如果用户正在线路上查看 XMPP 流量，就会看到`<bind>`和`<session>`元素(在`<iq>`节中)被发送出去以完成这些任务。如果客户端没有提供绑定的资源，那么服务器将为其选定一个(通常是随机地选择一个)。此外，即使客户端提供了一个资源，服务器也可能会修改用户选中的资源。

当两台服务器相互连接时，身份验证步骤稍有不同。服务器交换并验证 TLS 证书，或者接收服务器使用某种回拨协议通过 DNS 来验证发送者的身份。

1.6.4 连接断开

当用户结束 XMPP 会话时，他们终止会话并断开连接。最有礼貌的终止会话的方式是，首先发送无效出席信息，然后关闭`<stream:stream>`元素。

通过发送最后的无效出席信息，用户的联系人能够得知该用户离开的原因。显式地关闭流则可以让任何在路上的 XMPP 节安全地到达。

下面就是一个礼貌的断开连接。

```
<presence type='unavailable' />
```

```
</stream:stream>
```

然后，服务器终止发往客户端的流。

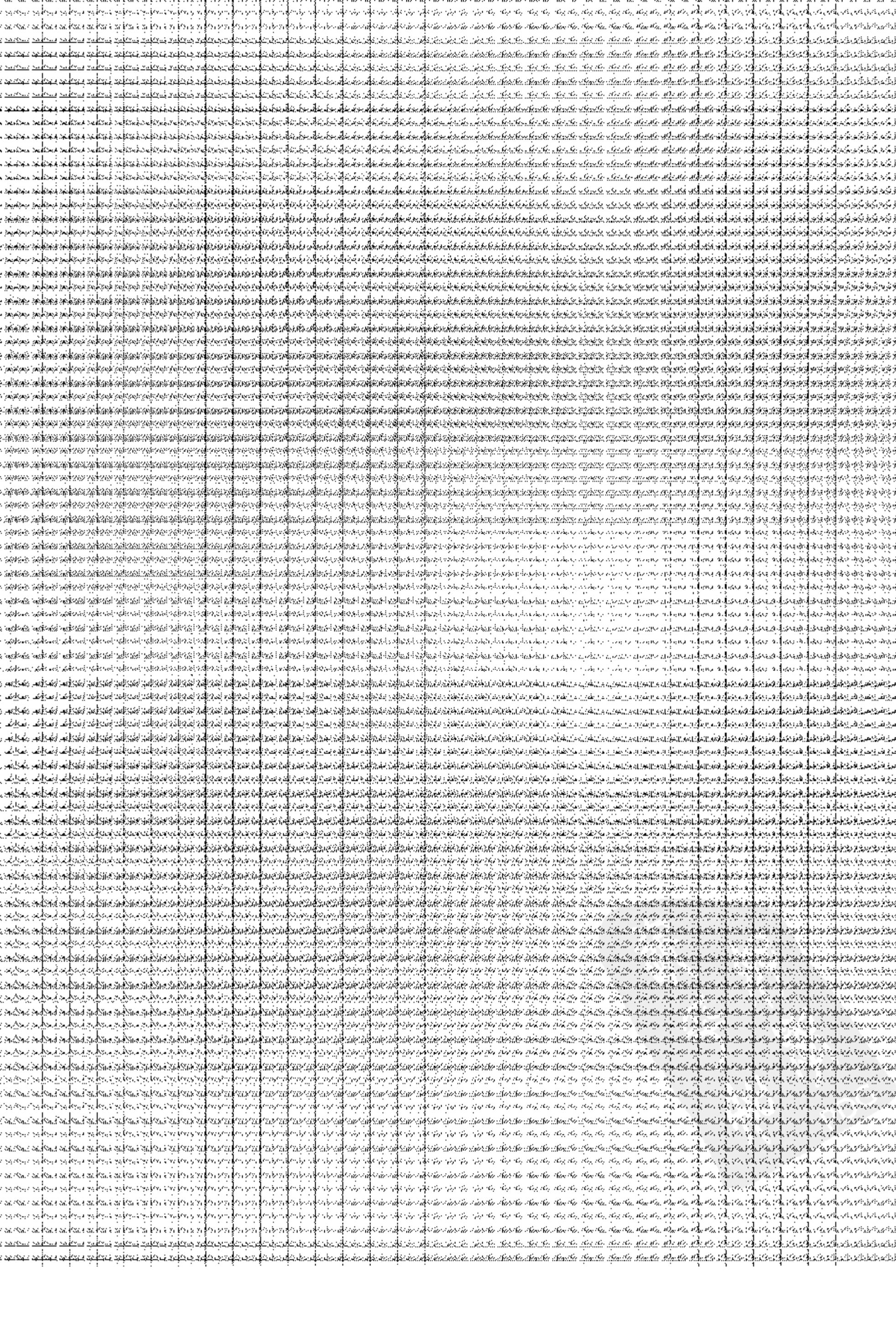
1.7 小结

本章介绍了 XMPP 协议并学习了它的历史、用例、寻址方式、词汇表以及连接生命周期。我们还看过几个 XMPP 节示例并学习了构成 XMPP 网络的不同实体。

现在应该能够理解如下内容：

- XMPP 是一种开放的、标准化的协议，最初用来替代专有 IM 网络。
- XMPP 协议历史已超过 10 年，已经相当成熟。
- XMPP 非常适于编写 IM 应用程序，但它还能胜任任何需要交换结构化消息的任务。
- 服务器、客户端、组件和插件组成了 XMPP 网络，均有各自的用途。
- XMPP 地址(名为 JID)与电子邮件地址类似，可分解为三部分：节点、域和资源。
- 完整 JID 是实体的最具体的地址，比如 `darcy@pemberley.lit/library`。
- 裸 JID 是完整 JID 不含资源的部分，比如 `darcy@pemberley.lit`。
- 服务器将处理发往客户端的裸 JID 的节，可能会将这些节路由到一个或更多已连接资源。
- 寻址到完整 JID 的节将直接传送给指定的资源。
- XMPP 主词汇表中共有 3 种节：`<message>`、`<presence>`和`<iq>`。
- `<message>`节用于在实体之间交换“发后不管”消息。
- `<presence>`节传达出席状态变化并用来操纵出席订阅。
- `<iq>`节提供了非常类似于 HTTP 协议的 GET 和 POST 操作的请求-响应语义。
- 所有 XMPP 会话均有一个由几个阶段构成的生命周期：连接、流的建立、身份验证、会话正文以及连接断开。

XMPP 的基本概念和协议语法只是冰山一角。在第 2 章中我们将学习如何使用这些概念来设计 XMPP 应用程序。



第 2 章

设计 XMPP 应用程序

本章内容

- HTTP 与 XMPP 之间的差异
- 使用 BOSH 来桥接 XMPP 和 HTTP
- 应用程序架构和协议设计

任何一种协议，不管看上去多么精妙绝伦，它都不可能成为所有问题的最佳解决方案。许多能够采用 XMPP 解决的问题若采用其他协议则很难解决，但即使 XMPP 也有其适用范围，若超出这个范围则应采用其他技术。为了编写尽可能完美的 XMPP 应用程序，必须首先理解 XMPP 擅长解决什么问题以及如何设计应用程序来最大程度地发挥出 XMPP 的优势。

XMPP 擅长于实时通信、协作以及数据交换。其他协议采取“拉”的方式获取数据，但 XMPP 采取“推送”的方式传送数据，从而可以更高效地通知并更快地响应新信息。XMPP 支持现在最流行的应用程序的社交功能，这使得开发人员很容易加入人际关系和通信并以此为基础构建新功能。XML 节丰富的、可扩展的词汇表为我们提供了一组稳固的、多种多样的工具，可用来构建许多常见模式或将其组合到自己的协议中。

尽管可以在多种上下文中(既可以在后端，也可以在前端)开发 XMPP 应用程序，但本书的应用程序使用 Web 浏览器作为它们的运行环境，并采用 Web 技术来实现。由于 Web 浏览器本身并不能理解 XMPP 协议(只能理解 HTTP 协议)，因此我们还需要研究如何让 XMPP 协议与 HTTP 协议合作以创建神奇的交互式应用程序。

2.1 他山之石

在启动一个项目之前获得一点激励总是有好处的，而现在已经存在许多极好的 XMPP 应用程序可以激励我们并让我们了解 XMPP 可以做出什么样的应用程序。虽然下面的几个应用程序只是肤浅地探讨 XMPP 协议的潜力，但它们确实可以作为了解借助 XMPP 协议能够做些什么(即使在现有应用程序的背景下)的较好示例。

由于 XMPP 最初是作为更好的即时通信协议，因此第一个应用程序就是一个 IM 客户端。

图 2-1 给出了在 Mac OS X 上运行的 Adium 客户端(<http://adium.im>)的一个截屏。这个客户端支持 XMPP 以及其他几种 IM 协议。每个能够想象得到的平台上都有类似的客户端存在。

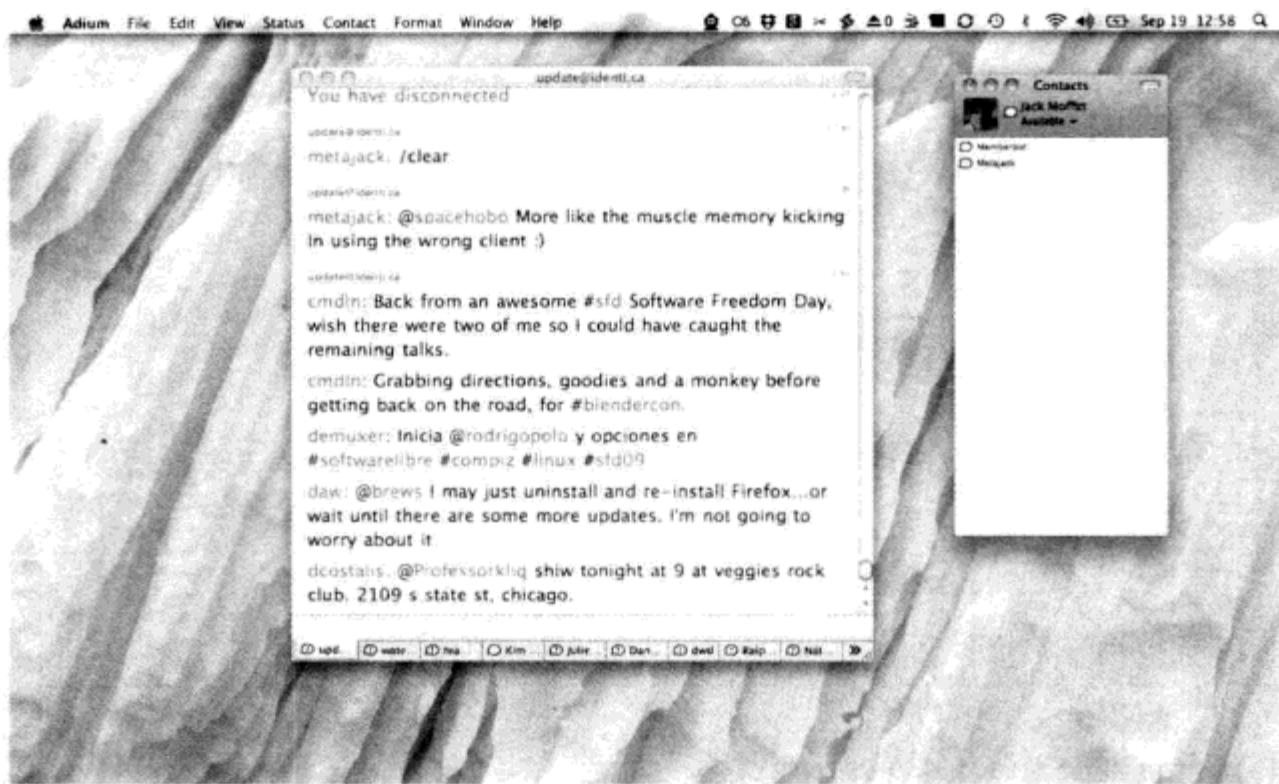


图 2-1

Google 已经在 XMPP 技术领域进行了巨大的投资。最初推出的是 Google Talk 服务，该服务已经历数年的改进。Google Talk 为任何拥有 Gmail 账户的人提供 IM 服务，同时还为所有 Google Apps for Domains 的客户提供该服务。Google 还曾参与 Jingle 最初的工作，这是一个进行语音和视频会议的 XMPP 扩展。Google 即将推出的 Wave 协议本身也是一种 XMPP 扩展，而它的云计算平台 AppEngine 也启用了 XMPP。图 2-2 给出了 Gmail 客户端(<http://gmail.com>)，它也包含了一个简单的基于 Web 的 IM 客户端。

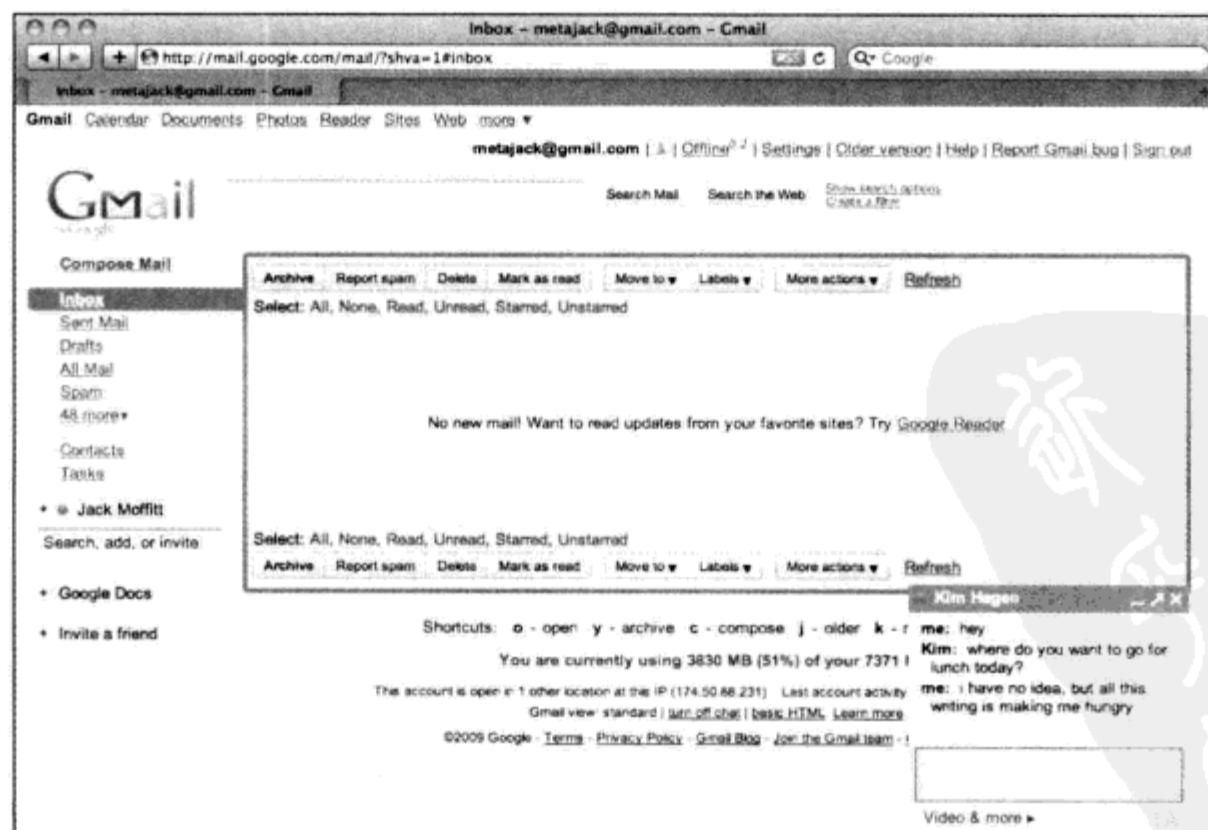


图 2-2

随着Twitter的出现，微博已经崛起，即使在企业环境中。像Socialcast、Yammer和Presently这样的服务均面向企业提供个人微博服务以改善公司的内部通信。因为低延迟通信是非常理想的，所以许多企业转向XMPP作为解决方案，这一点也不奇怪。图2-3展示了Presently应用程序(<http://presentlyapp.com>)允许用户与他们的同事保持联系。Presently使用Strophe库来处理通信，就像本书中的应用程序一样。它们还使用XMPP来支撑自己的后端基础设施。

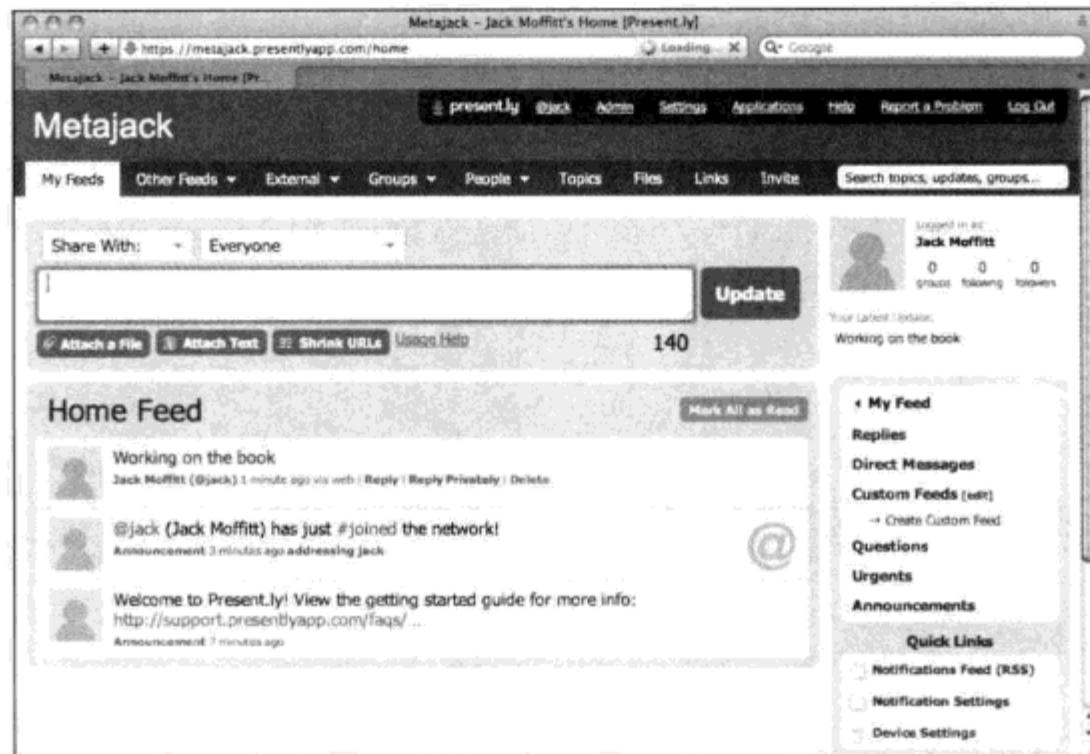


图 2-3

XMPP 极其擅长通信和协作，支持诸如群聊和文件共享这样的功能。Drop.io(<http://drop.io>)已经将XMPP的多人聊天室转换成一个丰富的、能够共享音频、视频、聊天和图片的协作空间。图2-4给出了这些协作空间之一在聊天模式下的一个示例。Strophe库还支撑起Drop.io应用程序。

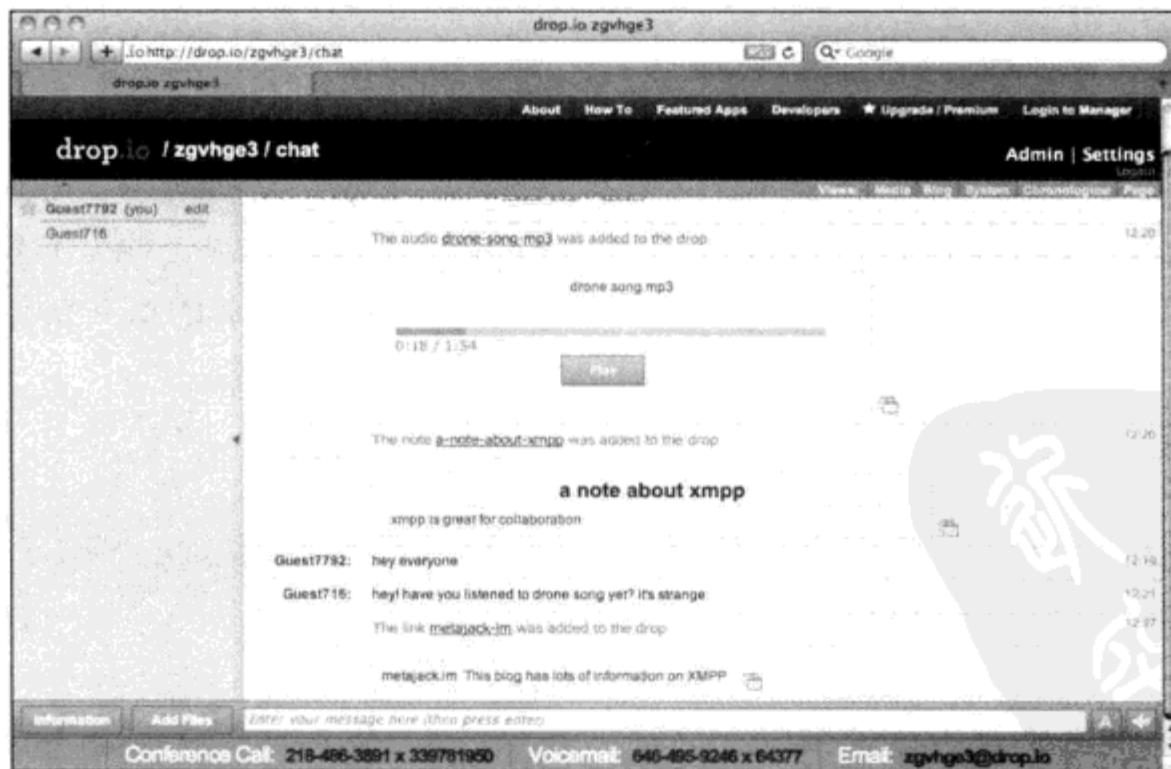


图 2-4

实际上，Strophe最初是为了在XMPP之上创建一个基于Web的、多玩家游戏系统而诞生

的。由于发现没有合适的通过 HTTP 传送 XMPP 的实现，我的团队着手建立一种实现，并在此基础上构建我们的应用程序。现在，不管是由于玩家深思熟虑而造成的游戏速度，还是下快棋时的电闪雷鸣般的速度，Chesspark 均可以让世界各地的玩家彼此实时地下棋。如果不借助 XMPP 和 Strophe，那么这项工作则要更加困难得多。图 2-5 给出了 Chesspark(<http://chesspark.com>) 的实际运行情况。

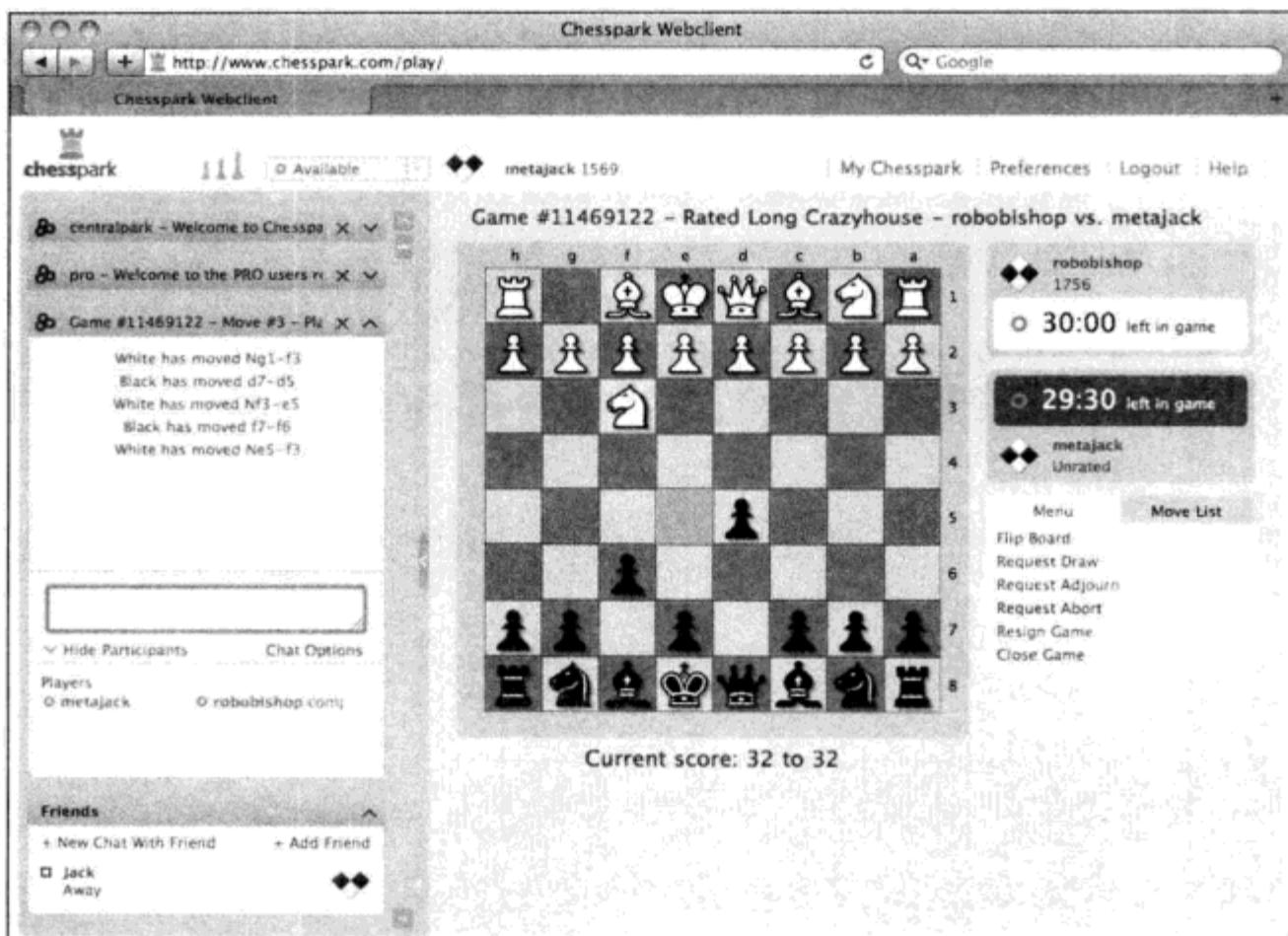


图 2-5

2.2 XMPP 与 HTTP 的比较

如果为项目选择了适当的工具，我们就会从中受益。因此，非常重要的一点就是要知道在什么场合下 XMPP 最能满足应用程序的需求。用户可能已经多少了解 XMPP 可能更适合自己的需求，或者对研究该协议的潜在用途感兴趣。本节将试着指出与 Web 的原生协议 HTTP 相比，XMPP 更适用于哪些场合。

2.2.1 XMPP 的优势

与 HTTP 相比，XMPP 具有如下的优势，下面将分别讲解每一点：

- 能够“推送”数据，而不是“拉”
- 防火墙友好
- 牢固的身份验证和安全机制
- 为许多不同的问题提供大量即开即用的工具

1. 推送数据

HTTP 客户端只能从服务器那里请求数据。除非服务器正在响应客户端请求，否则它不能向客户端发送数据。但 XMPP 连接是双向的。任何一方在任何时候都可以向另一方发送数据，只要该连接处于打开状态即可。

这种推送数据的能力极大地拓展了 Web 应用程序和协议设计的可能性。应用程序不必为获取更新而进行效率低下的轮询，而是能够在新信息可用时立即获得通知。这不仅减少了请求的总数量，而且让新信息变得可用的时刻与客户端获悉该信息可用的时刻之间的延迟几乎降为零。

2. 取悦防火墙

有些 Web 应用程序支持 HTTP 回调的使用，Web 服务器请求另一个 HTTP 服务器以发送数据。如果不是因为防火墙、NAT(network address translation，网络地址转换)以及 Internet 的其他现实情况，这本来是一个推送数据的便利功能。但实际上，很难从外部世界建立通往客户端的任何连接。

XMPP 连接是对防火墙和 NAT 友好的，这是因为服务器到客户端通信所用的连接是由客户端发起的。一旦建立连接，服务器就可以将其所需的所有数据推送给该客户端，就像在 HTTP 请求的响应中所做的那样。

3. 增强安全

XMPP 构建在 TLS 和 SASL 技术之上，它们为 XMPP 连接提供了牢固的加密和安全功能。虽然 HTTP 使用 SSL，但 HTTP 身份验证机制并没有大量地被开发人员实现和使用。相反，Web 中充斥着大量实现自己的身份验证方案(通常比较低劣)的站点。

4. 更大的工具箱

HTTP 仅限于简单的请求-响应语义而且只支持少数几项操作，即 GET、PUT、POST、DELETE，等。XMPP 包含三种不同的低级工具(<presence>、<message>和<iq>节)以及将近 300 种扩展(将这些低级工具组合成复杂的协议)。

当然有可能在 HTTP 之上构建许多同样的构造，但协议设计并非儿戏。XMPP 的扩展和协议非常牢靠而且经过广泛的评估，而类似的 HTTP 扩展往往不能彼此兼容，而且比较脆弱或比较简陋。

许多 XMPP 工具依靠并利用推送数据的能力。任何针对类似问题的 HTTP 解决方案都必须重新开发 XMPP 的大多数功能。

2.2.2 XMPP 的不足

每种协议都有各自的优缺点。在许多场合中 XMPP 并不是完成任务的最佳工具或受到某种限制。XMPP 的缺点包括：

- 有状态协议
- 社区和部署不及 HTTP 广泛
- 对于简单的请求，其开销比 HTTP 大

- 仍然需要专门的实现

1. 有状态

HTTP 是一种无状态协议，而 XMPP 则是有状态的。无状态协议更易于伸缩，这是因为每台服务器不需要知道整个状态就可以服务请求。XMPP 的这个缺点在实际中并不致命，这是因为大多数重要的 Web 应用程序毫无例外地均大量使用了 Cookie、后端数据库以及许多其他形式来存储状态。

虽然许多用来提高基于 HTTP 应用程序伸缩性的工具也同样可用来提高基于 XMPP 的应用程序的伸缩性，但由于 XMPP 资历相对较浅而且流行度相对较小，因此这类工具的数量和多样性更有限。

2. 较小的生态系统

HTTP 的年龄几乎是 XMPP 的两倍，而且作为 Web 的基石，它已经变得极其流行并得到人们很好的理解。这带来的结果就是，无论在哪一方面 HTTP 生态系统都要比 XMPP 生态系统更大。HTTP 库更多，HTTP 服务器更多，而且理解 HTTP 的工程师数目几乎要比理解所有其他现有协议(包括 XMPP)的工程师总数都要多。

这就是说，同样的情况也出现在 C 和 Java 编程语言与 Python、Ruby、Perl、Objective-C 以及其他语言的比较上。这些流行度较低的语言在完成工作方面并不逊色，而且通常是在许多任务的更好的解决方案。许多公司发现使用最强大的工具能够让它们在与其他只使用最流行工具的公司的激烈竞争中保持优势。

在开发 XMPP 应用程序的过程中，有时会遇到实现得并不好或没有按照自己需要实现的情况。通常很容易构建这些问题的解决方案，这是因为这些协议和交互已经有着很好的定义和规范的文档。对于 XMPP 扩展的自定义实现而言也同样如此，如果项目中需要使用它们，那么不必实现整个规范，只需实现它们所需的部分即可。

3. 更大的开销

XMPP 并未针对短期会话和简单请求进行优化。在这些领域中 HTTP 无疑是赢家。建立、维护和销毁 XMPP 会话都需要一些资源。XMPP 非常适于服务静态文档，但除非处在更大的 XMPP 应用背景中，否则这超出了 XMPP 的适用范围。

对于更长期的连接的或更复杂的交互，XMPP 开销与 HTTP 解决方案相比就可以忽略不计。任何 HTTP 解决方案也都需要支持状态、推送通知以及其他让 XMPP 如此有趣的功能。针对类似问题的基于 HTTP 的解决方案逐渐开始出现，但 XMPP 解决方案往往更加成熟而且更加优化。

4. 专门的实现

Apache Web 服务器曾经是所有人构建 Web 应用程序所需的全部。现在，多家公司正在构建自己的专门的系统来处理 Web 请求以提高处理性能并降低响应时间。

XMPP 刚刚进入应用程序和开发的黄金时期，各公司还没有多少时间针对特殊需求来开发专门的 XMPP 服务器和库。大多数 XMPP 服务器都是围绕传统的 IM 用例而设计的，但是许多 XMPP 服务器都可以被分拆并改作其他用途。

大多数时候，应用程序并不会如此苛求以至于需要专门的实现来支撑它。随着时间的推移，应用程序开发人员社区会将 XMPP 推向它的极限，就像 HTTP 社区所做的那样，开发专门的关注性能和大规模部署的工具。

2.3 桥接 XMPP 与 Web

虽然有几款浏览器正在试验一些功能以利用 XMPP，但主流浏览器目前还没有内置 XMPP 协议支持。但通过某种巧妙的编程和一些服务器端的帮助，我们可以在 HTTP 连接之上建立高效的 XMPP 会话通道。

使这种高效通道成为可能的是一项名为 HTTP 长轮询的技术。通过联合使用一个简单的基于 HTTP 的管理协议以及 XMPP 连接管理器，我们可以将 XMPP(以及它的所有功能)带入到 HTTP 应用程序中。

2.3.1 长轮询

早期的那些希望提供活动的不断变化的数据访问的网页使用定期刷新页面的方法。如果数据不断改变，那么这种直接的方法可以运行得很好，比如许多新闻站点使用定时刷新来获取活跃的博客重要事件。

微软最终在 Internet Explorer 中发布了一个名为 XMLHttpRequest 的 API，可以在网页中的 JavaScript 代码建立请求并处理返回的数据，而无须重新加载页面。最终所有主流浏览器都实现了该 API，程序员开始了解并使用这个便利的工具。这项技术最终叫做 AJAX(Asynchronous JavaScript and XML，异步 JavaScript 与 XML)。

使用 AJAX 可以在不刷新整个页面的情况下更新数据。这提高了效率而且几乎一切都变得更加动态，响应更加灵敏。

即使使用 AJAX，仍然是定期地请求(或轮询)数据。如果读者曾经在一个专横的老板下面工作过，他会不停地问“软件还没有编完吗？”这其实就是轮询。尽管服务器不会被激怒，但是如果太多客户端过快地轮询，服务器也可能变得缓慢。如果不被打扰，那么完成的工作量就会减少。但为了获取快速更新，轮询的间隔必须相当短，最低的延迟就是轮询间隔的长度。

轮询的另一个问题是大多数轮询请求并没有接收到新数据。为了在某个变化发生之后的一个合理的时间帧之内看到该变化，轮询间隔必须相当短，但实际的数据变化可能不会如此频繁。就像给专横的雇主(希望他是虚构的)的答复一样，服务器对“软件还没有编完吗？”问题的答案总是“还没有”。

一些聪明的人发明了一种巧妙的技术来解决这个问题。它们并不立即响应该请求，而是在新数据没有准备好时将其挂起一段时间。

例如，如果服务器上新数据已经准备就绪，那么服务器会立即应答。如果尚无新数据，那么服务器将保持连接的打开状态，并持有所有应答。一旦新数据到达，它最终会响应该请求。如果在一定时间之后没有新数据到达，服务器可以发回一个空的响应，这样就不会一次性锁住过多打开的连接。一旦一个请求返回，客户端就会立即发送一个新的请求，整个过程重新开始。

因为每个轮询请求都可能打开较长的时间，所以这种技术被称为长轮询。与普通轮询相比，

它有多种好处。

一旦新数据变得可用，客户端就会立即接收到该数据，这是因为它总是至少有一个连接打开并等待服务器的更新。通过将轮询的长度设置为某个较高的值，服务器必须处理的请求总数将保持在一个合理的水平，而更新的延迟则会降到最低。用户看到数据的即时变化，服务器运营商不用花高价扩充系统，这是一个多赢的局面。

唯一的真正改变之处是，不让客户端等待重新发送请求，而是让服务器等待直到其有新数据需要通告才响应请求。

人们已经设计了多个库和协议来利用长轮询技术，但 XMPP 是最早实现之一。在 XMPP 中，这种桥接被称为 BOSH(Bidirectional streams Over Synchronous HTTP，在同步 HTTP 之上传送双向数据流)。读者可能还听说过 Comet 或反向 HTTP，这些也都是构建在同样的技术之上的。

该技术的一个缺点是服务器需要更聪明才可以处理这些长轮询请求，而这正是连接管理器的作用所在。

2.3.2 管理连接

XMPP 连接可以持续任意长的时间，但 HTTP 请求却相当短命。连接管理器负责维护第三方的 XMPP 连接并通过 HTTP 长轮询技术来提供对连接的访问。

浏览器和连接管理器使用一种名为 BOSH 的简单协议通过 HTTP 进行通信。实际上，BOSH 帮助 HTTP 客户端建立一个新的 XMPP 会话，然后将 XMPP 节包装到一个特殊的<body>元素中通过 HTTP 来回传送。它还提供了一些安全功能以确保 XMPP 会话不会被轻易地劫持。连接管理器与 XMPP 服务器通信就像它是一个普通的客户端一样。

这样一来，HTTP 应用程序就能够控制一个真正的 XMPP 会话。由于长轮询技术提供的高效率和低延迟，因此它的性能相当好，足以匹敌原生连接。

为了将 XMPP 整合到浏览器中似乎需要完成大量的工作，但这不仅在实际中运行良好，而且事实证明，与直接的 XMPP 连接相比，这项技术甚至还有如下优势：

- 与连接管理器的交互是逐个请求进行的，这可以让客户端从一个网络移动到另一个网络。即使终端用户的 IP 地址改变多次，托管连接仍然一直保持有效。
- 因为一次失败请求并不会终止托管连接，所以这些托管会话非常稳固，能够容忍临时的网络故障。
- 因为连接管理器缓存并重发请求数据，所以不必担心连接中断时会丢失数据。
- HTTP 对防火墙极其友好，而且因为大多数连接管理器运行在标准 HTTP 端口之上，即使在不允许除 HTTP 之外的任何端口的受限网络中托管连接仍然能够运行。

这些优势使得托管连接非常适于某些场合，即使此时可以使用直接 XMPP 通信。

2.3.3 让 JavaScript 理解 XMPP 协议

利用 HTTP 长轮询，我们就拥有了从服务器获取低延迟数据更新的技术。将该技术与连接管理器组合起来，我们就能够通过一系列 HTTP 请求来发送和接收 XMPP 数据。最后，我们还需要简化该技术在 Web 的原生编程语言 JavaScript 中的实现。

Strophe 库的创建宗旨就是为了让使用 JavaScript 编写 XMPP 应用程序能够像采用任何其他语言一样简单，将托管连接的底层细节全部隐藏起来。就 Strophe 的用户而言，它看上去就跟

在任何其他环境中所使用的原生 XMPP 连接一样。

与普通 AJAX 应用程序相比，长轮询应用程序有一些特殊的需求。大多数 AJAX 库很少进行错误处理，这是因为失败的 AJAX 请求数目所占比重很小。但对于 XMPP 应用程序而言，任何失败的请求都可能导致延迟的巨幅增加并严重影响用户体验。由于这个原因，Strophe 尽可能多地优雅地处理各种错误，并使用提前超时来探测各种可能会被发现的问题。

这些功能使得 Strophe 相当可靠并且拥有高性能，即使面对不可避免的、偶然出现的错误也同样如此。Strophe 用户经常报告它们的应用程序能够持续运行数天而不会断开连接。

2.4 构建 XMPP 应用程序

XMPP 协议是多种应用程序领域的明智之选，而且我们已经了解了有关如何在 Web 中使用 XMPP 的技术细节。在第 3 章着手介绍第一个应用程序之前，让我们先来学习一些有关如何构建 XMPP 应用程序的细节。

2.4.1 浏览器平台

Web 浏览器可能是有史以来部署最广泛、使用最多的应用程序平台。Web 浏览器存在于每种计算机，甚至在许多移动电话中，而且更重要的是，这些设备的用户往往非常熟悉浏览器和 Web 应用程序。

随着 Web 应用程序需求越来越复杂，人们已经创造了许多新技术和抽象概念来改进该平台。XMPP 又带来了一套新技术和抽象概念，但它同时带来的是实时、交互式和协作式应用程序的巨大潜力。社交 Web 的崛起带来了社交应用程序的兴起，而如果开发人员希望朝着连接人类的方向更进一步，那么类似 XMPP 这样的技术将帮助他们实现自己的目标。

对于 XMPP 开发人员而言，将 Web 浏览器定位为目标平台有着巨大的意义。Web 应用程序是跨平台、易于部署而且有着巨大的、有着熟练使用经验的用户基础。此外，Web 技术大量使用 HTML，而用于 HTML 的工具通常也都能很好地用于 XML，因而也能用于 XMPP。

jQuery 库就是这样的一个工具，它已为众多 Web 开发人员熟知。jQuery 使许多普通的 HTML 和 CSS 操作变得简单有趣。该功能也同样适用于 XML 数据，因为它们共享非常类似的结构。本书的应用程序使用 jQuery 来同时处理和操作用户界面(由 HTML 和 CSS 组成)以及接收到的 XMPP 数据。

Web 技术有其擅长的领域，但从实践角度来看，Web 开发人员和 XMPP 开发人员都在寻求一种更好的创建新型有趣应用程序的平台。

2.4.2 基本的基础设施

就像 Web 通常需要一个 Web 服务器和应用程序服务器或框架，XMPP 应用程序也需要一些基本的基础设施。XMPP 连接要求有一台 XMPP 服务器，而且通常在该服务器上会有一个账户。XMPP 应用程序还需要与连接管理器进行通信，这是因为浏览器目前还无法原生地理解 XMPP 协议。最后，应用程序使用的任何服务也必须由 XMPP 服务器提供。

XMPP 服务器需求并不是非常难以满足的。在几乎所有平台上下载、安装和运行 XMPP 服

服务器都是一件相当简单的事情，而且有多种选择。还可以使用公共 XMPP 服务器，用户可能已经拥有该服务器的账户，但这取决于应用程序使用的数据量。许多公共 XMPP 服务器针对 IM 流量进行了优化，并限制高吞吐量客户端。

许多用户已经拥有 XMPP 账户，这要归功于 Google、LiveJournal、具有前瞻思维的 ISP 以及其他大型组织所做的努力。如果运行自己的服务器并要求用户到该服务器注册，那么通常很容易以编程方式在普通 Web 应用程序后端中创建 XMPP 账户。

所有主流 XMPP 服务器都内置 HTTP 托管连接支持(或 BOSH 支持)。通常只需稍微修改配置文件将该功能开启即可(如果没有默认开启的话)。如果希望用户能够连接到任意 XMPP 服务器，那么需要一个独立的连接管理器。有关使用和配置连接管理器的更多信息，请参见附录 B。

每种 XMPP 服务器都支持一组核心功能和若干 XMPP 扩展。通常，XMPP 服务器的网站上会给出扩展的相关资料。大多数服务器支持那些成熟的而且流行的扩展，比如多人聊天以及发布-订阅。如果应用程序使用了某些 XMPP 扩展，那么一定要检查自己使用的服务器是否支持这些扩展。

许多 XMPP 应用程序不需要借助任何特殊 Web 应用程序服务器或其他基础设施就能够实现。本书中的应用程序都是独立的，而且能够在可公开访问的 XMPP 服务器上运行。此外，为了进行开发，我们还提供了一个连接管理器，因此在研究本书的应用程序过程中不需要自行建立。

2.4.3 协议设计

如果不是在创建现有 XMPP 服务(比如多人聊天或传统 IM 功能)的一种新的更好的版本，那么用户可能正在进行某种协议设计来实现自己的梦想。XMPP 提供了大量的工具可作为工作基础，而且通常将这些工具进行简单的组合就足够满足大多数应用程序的需要。下面的指南影响了本书中的应用程序，或许对用户自己的协议设计有所帮助。

1. 组合现有协议

如果能够将现有的协议扩展进行组合来实现应用程序，那么最好这样做。充分利用已发布的将近 300 种扩展，通常会有一个很好的起点。

即使用到的特定扩展没有实现，那么实现一种协议也远比重新设计一个协议容易得多。扩展的作者们一直以浓厚的兴趣在思考着该问题域，他们撰写文档来描述解决方案。此外，XMPP 扩展是社区工作成果，用户的反馈将有助于为所有的人改进该扩展。

第 11 章中开发的游戏就是一个具体示例。一种可能的解决方案是为所有游戏交互创建一个新的协议。或者相反，应用程序将游戏语义置于多人聊天室之上。这涉及少量的协议设计以便处理游戏特有的内容，但复用了已经经过重重检验的现有扩展的大量工作。

使用协议组合节省了工作量，还使得新协议更容易被他人理解。为了理解新协议，他们只需要理解这些协议如何组合在一起以及一些少量的新事物。

2. 保持简洁

保持简洁几乎在每个领域中都是极好的建议，它也同样适用于 XMPP 协议设计。如果不创建一些全新的内容，那么试着让它尽可能简单。

不仅协议更易于理解，而且由于没有复杂性将导致更少的 bug 和更短的开发时间。一个复杂的协议往往说明已经忽视了某个更简单的方法。

为了解决新用例，许多 XMPP 扩展本身也会得到扩展。没有必要(也不希望)在一开始就将所有需要的功能打包到协议中。实际上，许多较为复杂的扩展一旦变得较大时，它们最终都会被拆分成一个核心部分和若干相关的扩展。

3. 避免扩展出席

<presence>节经常会占服务器流量最大的比重。大多数<presence>节扩展并不适于一般情况。与直接扩展<presence>节相比，更好的做法是利用 XEP-0163 中定义的 PEP(Personal Eventing Protocol)协议，可以让用户订阅他们感兴趣的额外数据。

PEP 扩展以及 Entity Capabilities(XEP-0114)和 Service Discovery(XEP-0015)使得提供扩展的出席类型的信息非常有效并可选择加入。

其他节并不需要类似的优化，这是因为它们通常并不会一次性广播给许多人。

4. 参与社区

XMPP 社区中有众多拥有协议设计经验的人，而且他们通常非常友好。如果用户认为自己的新协议会得到普通应用，那么用户还可能希望为其编写文档并提交作为 XMPP 扩展。即使用户并不打算在自己的项目之外使用自己的扩展，让社区对用户的设计进行反馈仍然会非常有帮助。

协议设计讨论通常会在标准邮件列表中进行，可以通过访问 XSF 网站上的讨论页面 <http://xmpp.org/about/discuss.shtml> 来订阅该列表。可以随意询问问题、共享自己的协议设计思想并向其他协议的讨论和反馈做出贡献。社区总是欢迎新成员。

如果希望将自己的协议作为正式扩展提交，就会希望使用扩展提交页面 <http://xmpp.org/extensions/submit.shtml> 中提供的模板和指南按照 XEP 格式来撰写文档。XSF 委员会会考虑是否接受新扩展。新协议要获得接受只需该扩展的用途具有普遍性并且尚未涉及现有工作。

用户还可以作为会员加入 XSF，XSF 是一个开放组织，由其会员运营。每季度进行一次选举，XSF 一直在寻找新会员，他们既可以是开发人员，也可以是爱好者。更多信息请参见会员页面 <http://xmpp.org/xsf/members/>。

2.5 小结

在本章中，我们学习了 XMPP 是胜任什么任务的最佳工具以及它与 HTTP 协议的不同之处。此外我们还研究了如何让 XMPP 协议在只能理解 HTTP 的 Web 浏览器中运行以及如何构建 XMPP 应用程序。

在这个过程中我们讲解了如下主题：

- 从其他公司的 XMPP 应用程序示例中受到启发
- XMPP 与 HTTP 的优缺点
- HTTP 长轮询

- 托管的 XMPP 连接(BOSH)
- Strophe 库
- XMPP 应用程序必需的基础设施
- 如何设计 XMPP 协议

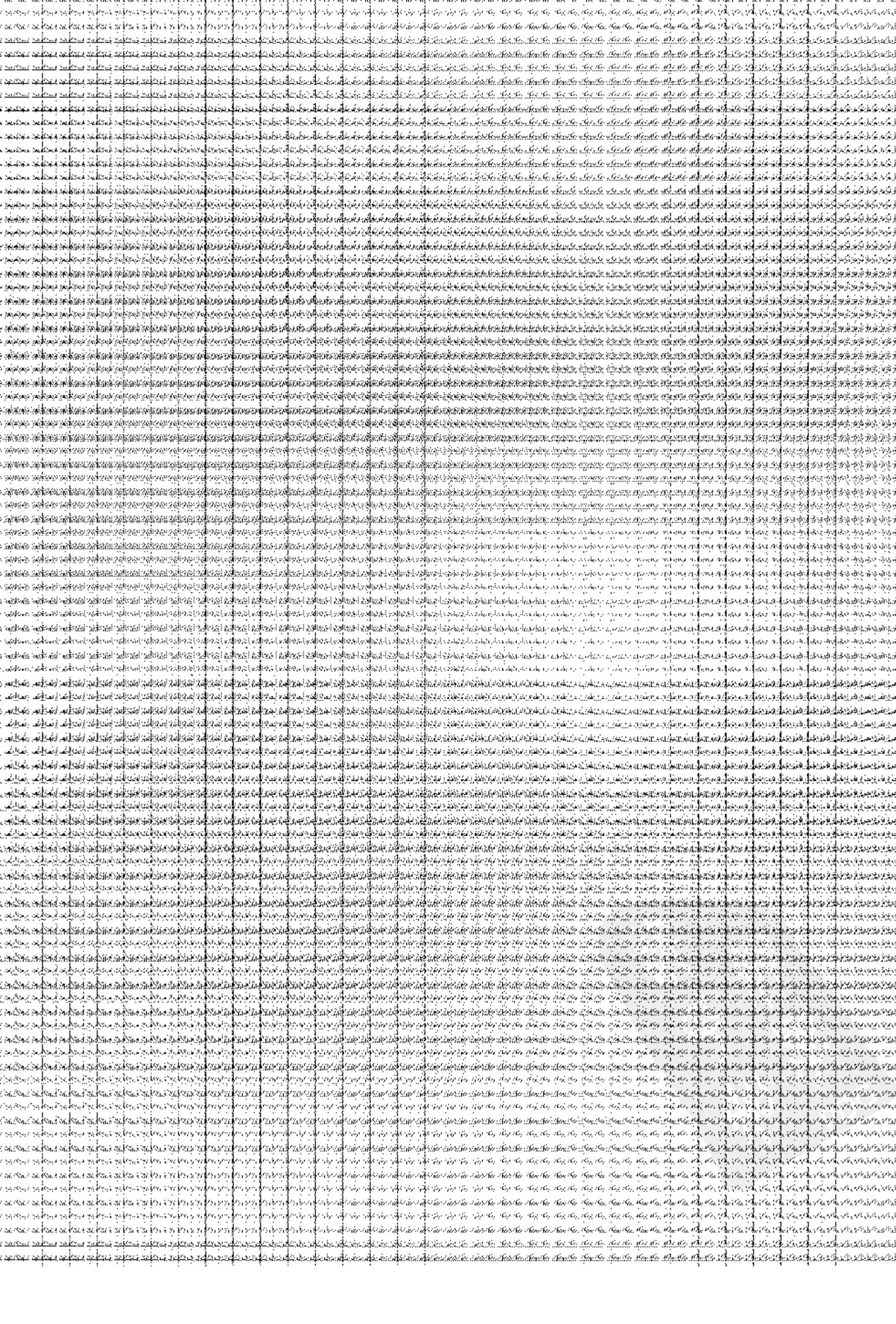
在第 3 章中，我们将创建第一个应用程序从而开始 XMPP 开发工作。

第 II 部分

应 用 程 序

- 第 3 章 Hello World: 第一个应用程序
- 第 4 章 探索 XMPP 协议: 一个调试控制台
- 第 5 章 实时微博: 一个 Identica 客户端
- 第 6 章 与好友交谈: 一对一天聊
- 第 7 章 探索服务: 服务发现与浏览
- 第 8 章 群聊: 多人聊天客户端
- 第 9 章 发布与订阅: 共享画板简介
- 第 10 章 与好友一同协作: 协作式文本编辑器
- 第 11 章 玩游戏: 面对面的 Tic-Tac-Toe





第 3 章

Hello World: 第一个应用程序

本章内容

- 创建和组织 XMPP 项目
- 建立到 XMPP 服务器的连接
- 构建并操纵 XMPP 节
- 向服务器发送数据
- 处理 XMPP 事件

XMPP 最初是作为一种开放的、联合的即时通信协议，但它现在已经演变成一个可以构建多种类型应用程序的强大协议。本书的这部分将介绍各种有趣应用程序的构建过程，并在这个过程中研究 XMPP 协议及其巨大的潜力。这些应用程序都采用 JavaScript、HTML 和 CSS 编写，尽管都很简单，但它们将展示使用 XMPP 编写强大的程序是一件多么简单的事情。

我们要编写的第一个应用程序是著名的“Hello, World”示例的 XMPP 实现。它向 XMPP 服务器发送一条消息并将响应显示出来。虽然听起来非常简单(而且事实上也确实简单)，但这仍然包含大量关于如何设置用户界面、获取必需的库以及学习 Strophe 库的内容。

当本章结束时，读者应该已经准备好开始构建更有趣的应用程序。读完本书，读者将会构建一些引人注目的项目，而如果不使用 XMPP 就很难完成这些任务。

本章可能是本书最困难的部分之一，只是因为它里面包含大量崭新的内容。

本书中的应用程序假设读者具有一定的 jQuery 库知识。如果尚不熟悉 jQuery，那么请参见附录 A 给出的简明手册。

3.1 应用程序预览

在开始编写一个应用程序之前，我们可以通过一些预览截屏图来了解最终的结果是什么样子。

图 3-1 和图 3-2 给出了未来的 XMPP 编程成果。通过这些截屏图我们可以看到应用程序最终的模样，同时它们

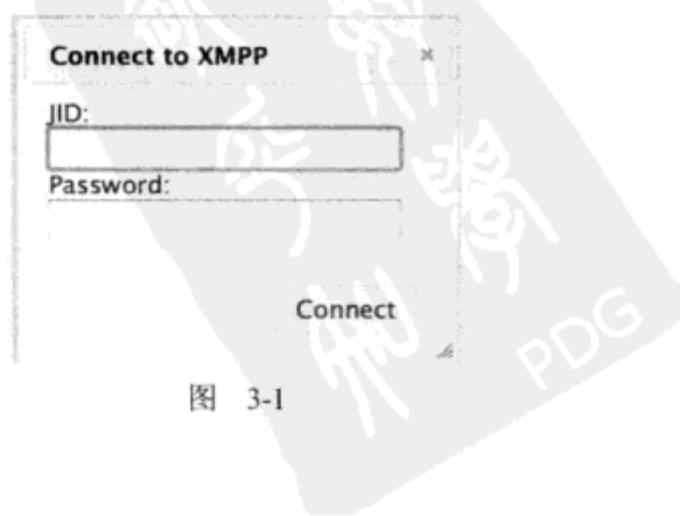


图 3-1

还提供了一个参考点，可用来与自己的工作进行比较。

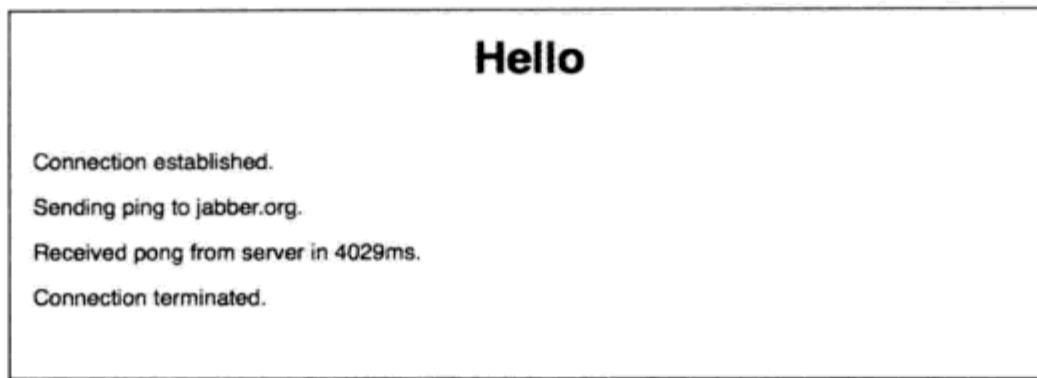


图 3-2

当应用程序首次启动时，它会打开图 3-1 中给出的登录对话框并接受用户输入的 XMPP 地址和口令。一旦应用程序向服务器发送一条消息之后，服务器的响应就会被显示出来，如图 3-2 所示。

我们故意让用户界面保持简洁，其目的是为了让读者将注意力放在 XMPP 协议和 JavaScript 代码上。

3.2 Hello 应用程序设计

每个应用程序都是由若干个小的部分通过特定方式有机地组合在一起的。这些部分可能是用户界面元素、XMPP 协议处理程序或某种过程的实现。在每一章中，我们会首先学习这一章的应用程序中涉及的各个不同部分以及它们如何有机地构成一个紧密的整体。

第一个应用程序名为 Hello，而且我们已经在图 3-1 和图 3-2 中给出了两个可见部分：登录对话框和服务器响应屏幕。还有几个看不见的部分：XMPP 连接的处理、XMPP 节的生成以及事件处理。

当用户首次打开这个应用程序时，他会看到让其输入 XMPP 用户名和口令的登录对话框。当用户单击 Connect 按钮，Hello 程序必须建立一条通往 XMPP 服务器的连接，并对该用户进行身份验证。

一旦建立了连接，Hello 程序的任务就是向用户的服务器发送一个简单的 XMPP 节并等待服务器的应答。Strophe 库能够让我们将处理程序附加到连接以响应特定的 XMPP 节。我们将会看到，这些处理程序将使得对传入的请求和响应的处理变得非常容易。

最后，Hello 程序将显示服务器应答的一个比较具有人类可读性的版本，并断开连接。

Hello 并不是一个多么复杂的应用程序，但在整本书中我们都会看到每个应用程序都在重复着这里用到的相同的 XMPP 工具。

3.3 准备

在开始编写代码之前，必须首先收集构建应用程序要用到的一些 JavaScript 库。我们需要如下部分：

- **jQuery** jQuery 库使得 HTML 和 CSS 的处理变得异常简单, 它在 XML(XMPP 节)的操作方面也非常方便。
- **jQuery UI** jQuery UI 库提供了我们所需的一些常见的用户界面构造块, 包括对话框和标签页。
- **Strophe** 该库使得编写 XMPP 客户端应用程序变得极其简单, 而且它已有多种编程语言版本。当然, 我们将使用 JavaScript 版本。
- **flXHR** Strophe 能够使用 flXHR(标准 XMLHttpRequest API 的 Flash 替代技术)来简化 JavaScript 同源策略的处理。通常, JavaScript 应用程序不能与外部服务器通信, 但借助 Flash 和 flXHR, Strophe 能够克服这个限制。

此外, 如果还没有在公共服务器拥有一个 XMPP 账户, 那么需要创建一个。

同源策略

出于安全考虑, JavaScript 代码在一个沙箱环境中运行。虽然 Web 应用程序能够从任何地方加载 JavaScript 代码, 但所有的代码必须将其通信限制到该 Web 应用程序所在的宿主服务器。这种限制被称为同源策略。

最近, 为了能够构建更有趣的应用程序, 人们常常寻找各种方法来绕开该策略。大多数 Web 应用程序库(包括 jQuery)已经提供了一些方法让我们实现跨域请求。但普通的针对 HTTP GET 操作的变通方法并不适用于 XMPP, 这些方法必须使用 HTTP POST。

本书使用 flXHR 库来实现跨域请求, 但也可以使用其他解决方案。最常见的替代方法是使用反向 HTTP 代理, 让 BOSH 连接管理器看上去就像是应用程序的本地服务器一样。附录 B 讨论了这种替代技术。

3.3.1 jQuery 与 jQuery UI

jQuery 和 jQuery UI 库可以分别从 <http://jquery.com> 和 <http://ui.jquery.com> 下载。本书示例用到的版本分别是 jQuery 1.3.2 和 jQuery UI 1.7.2。这些库的更新版本也应该能够运行。

jQuery 库(与许多 JavaScript 库类似)有普通版和缩减版的区别。两种都能运行, 但在开发期间推荐使用普通版, 这是因为缩减版 JavaScript 会妨碍调试。

Google 已经让人们能够通过它的 AJAX Library API 来访问许多 JavaScript 库(包括 jQuery 和 jQuery UI)。这意味着如果我们正在一台连接 Internet 的计算机上进行开发, 那么我们甚至不必下载这些库, 而是直接连接到 Google 的超快服务器即可。

本书中的样例代码使用 Google 托管的版本。要想在 HTML 中包含 jQuery、jQuery UI 和 UI 主题 CSS, 那么请将下面几行代码放入<head>元素中。

```
<link rel='stylesheet' href='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/themes/cupertino/jquery-ui.css'>
```

```
<script src='http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/jquery.js'></script>
<script src='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/jquery-ui.js'>
</script>
```

如果希望使用这些库的缩减版，那么只需要将 js 扩展名改为 min.js 即可。

如果已经自行下载了这些库并且希望使用自己的本地副本，那么使用下面的几行代码应该就能奏效(假设 jQuery 和 jQuery UI 安装到应用程序的 scripts 子目录中，而 jQuery UI 主题安装在 styles/themes 下)。

```
<link rel='stylesheet' href='styles/themes/cupertino/jquery-ui-1.7.2.custom.css'>
<script src='scripts/jquery-1.3.2.js'></script>
<script src='scripts/jquery-ui-1.7.2.custom.js'></script>
```

jQuery UI 带有数十种主题，而本书中的应用程序可以使用其中的任何一种主题。示例代码和截屏均使用了 `cupertino` 主题。可以在 jQuery UI 网站上浏览所有可用主题，并将代码中找到的所有 `cupertino` 替换成自己喜欢的主题。

3.3.2 Strophe

可以从 <http://code.stanziq.com/strophe> 网页下载 Strophe 库。要确保自己下载的是最新的 JavaScript 版。一定不要下载 `libstrophe`，这是 C 语言库，当然除非您打算采用 C 语言来编写 XMPP 代码。

在本书中，我们假设 `strophe.js` 文件以及所有 Strophe 插件文件均放在 `scripts` 目录下。这样在 HTML 文件中就可以使用下面这行代码将其包含进来。

```
<script src='scripts/strophe.js'></script>
```

3.3.3 f1XHR

`f1XHR` 库位于 <http://flxhr.flensed.com>。该库为 Strophe 提供了跨域请求支持，通过一个特殊的 Strophe 插件即可启用。我们将用到的 `f1XHR` 最低版本是 1.0.4，更新的版本也是可用的。

一旦下载 `f1XHR` 并解压缩，就可以将 `flensed-1.0/deploy` 目录中的内容放进应用程序所在目录的 `scripts` 子目录中。

在应用程序中启用 `f1XHR` 非常简单，只需要加载两个额外的 JavaScript 文件即可。

```
<script src='scripts/f1XHR.js'></script>
<script src='scripts/strophe.flxhr.js'></script>
```

第一个脚本 `f1XHR.js` 用来加载 `f1XHR` 库。第二个脚本是一个特殊的 Strophe 插件，可以让 Strophe 使用 `f1XHR` 来进行跨域请求。

3.3.4 XMPP 账户

如果还没有 XMPP 账户，那么可以在众多公共 XMPP 服务器中选择一个来建立自己的账户。<http://xmpp.org/services/> 给出了一个很长的公共 XMPP 服务列表。`jabber.org` 服务器一直是较为流行的选择，可以通过 <http://register.jabber.org> 建立账户。

请注意, 如果已经有 XMPP 账户, 那么大多数 XMPP 账户通常是为了典型的即时通信用途而提供的。每种服务器可能启用了不同的一组功能, 因此需要确认自己的服务器是否支持我们正在构建的应用程序所需的功能。例如, jabber.org 服务器支持本书所需的所有功能。

现在, 读者应该已经拥有开始构建 XMPP 应用程序所需的所有的库, 以及使用这些应用程序所需的 XMPP 账户。下面就让我们开始构建应用程序。

3.4 开始构建第一个应用程序

本书中的所有应用程序都是由一个 HTML 文件、一个 CSS 文件以及一个或多个 JavaScript 文件组成, 此外还有前一节中讨论过的必需的库。HTML 和 CSS 文件构成了应用程序的用户界面, 并包含所用的对话框、与用户交互的各种控件以及让这些元素极具吸引力的样式信息。我们的主要注意力将放在 JavaScript 文件上, 它们包含了所有的应用程序代码。

3.4.1 用户界面

本书的每一章均从应用程序的基本 HTML 和 CSS 布局开始。在每一章中, 我们会向这些文件中添加新标记, 但大部分时间, 最开始的 HTML 和 CSS 是不变的。

对于第一个应用程序, 我们更加详细地讲解所用的 HTML 和 CSS 标记, 但如果读者已经非常精通 Web 技术, 那么可以略过该代码, 直接阅读下一节。

程序清单 3-1 给出了 Hello 应用程序的 HTML 标记。



程序清单 3-1 hello.html

可从
Wrox.com
下载源代码

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"  
    "http://www.w3.org/TR/html4/strict.dtd">  
  
<html>  
    <head>  
        <title>Hello - Chapter 3</title>  
  
        <link rel='stylesheet' href='http://ajax.googleapis.com/ajax/libs/  
        jqueryui/1.7.2/themes/cupertino/jquery-ui.css'>  
        <script src='http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/  
        /jquery.js'></script>  
        <script src='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/  
        /jquery-ui.js'></script>  
        <script src='scripts/strophe.js'></script>  
        <script src='scripts/flXHR.js'></script>  
        <script src='scripts/strophe.flxhr.js'></script>  
  
        <link rel='stylesheet' href='hello.css'>  
        <script src='hello.js'></script>  
    </head>  
    <body>  
        <h1>Hello</h1>
```

```

<div id='log'>
</div>

<!-- login dialog -->
<div id='login_dialog' class='hidden'>
    <label>JID:</label><input type='text' id='jid'>
    <label>Password:</label><input type='password' id='password'>
</div>
</body>
</html>

```

在<head>节中，除了必要的库之外，还会加载 hello.css 和 hello.js 文件，它们是应用程序的 CSS 样式和 JavaScript 代码文件。

在<body>节接近结束的地方会看到登录对话框。jQuery UI 对话框是从普通的<div>元素创建而来的：在希望变成对话框的特定<div>元素上调用 jQuery UI 的 dialog()函数。因为我们要通过 JavaScript 代码来创建、显示和隐藏该对话框，所以它的<div>元素被赋予 CSS 类 hidden，这样只有在需要的时候它才会出现。

id 属性为 log 的空<div>元素将用来显示服务器的响应以及应用程序运行过程中的状态更新信息。这可以让我们非常容易地了解正在发生什么，并可精确掌握自己编写的代码正在做些什么。

程序清单 3-2 给出了 Hello 的 CSS 样式。该文件包含了一些让应用程序更美观的样式。



程序清单 3-2 hello.css

可从
Wrox.com
下载源代码

```

body {
    font-family: Helvetica;
}

h1 {
    text-align: center;
}

.hidden {
    display: none;
}

#log {
    padding: 10px;
}

```

在讲解了 HTML 和 CSS 文件之后，我们就可以开始编写代码了。

3.4.2 应用程序代码

本书中的 JavaScript 代码由 3 个基本部分组成。首先是应用程序的命名空间对象，应用程序的状态和函数都在此定义。位于命名空间对象之后的是文档准备就绪事件处理程序，一旦浏览器准备就绪它就会初始化应用程序。最后是自定义事件处理程序，它负责处理那些不是由元

素或用户交互触发的内部事件。

1. 命名空间

我们使用命名空间对象，从而尽可能地避免使用全局变量。虽然不能指望所有的第三方代码库都会坚持该原则，但通过将自己的应用程序状态和全局变量放进一个单独的对象(命名空间对象)中我们可以将问题降到最少。

与其像下面这样定义变量和函数：

```
var some_global = 0;
var another_global = true;

function my_adder(x, y) {
    return x + y;
}
```

还不如将同样的代码放入一个全局对象 `MyNamespace` 中：

```
var MyNamespace = {
    some_global: 0,
    another_global: true,
    my_adder: function (x, y) {
        return x + y;
    }
};
```

这里使用的语法稍有不同，这是因为每个符号都变成了 `MyNamespace` 对象的一个属性，而该对象则是使用 `{ }` 语法定义的一个对象字面值。现在，如果另一个库也定义了 `some_global`，那么我们的代码不会受到影响。

可以像访问任何其他对象的属性那样来访问命名空间对象的属性。表达式 `MyNamespace.some_global` 将访问 `some_global` 属性，而调用 `MyNamespace.my_adder(1, 2)` 则会返回 3。

对于命名空间对象，唯一需要引起注意的是，最后一个属性的末尾不要带有逗号。如果忘记了这个细节，那么该代码在 Internet Explorer 浏览器中就不会被正确加载。下面的代码是错误的：

```
var MyNamespace = {
    some_global: 0,
};
```

0 后面的逗号将告诉解释器其后还有另一个属性定义，但实际上并没有。在其他语言中，这些额外的逗号通常会被忽略，而且大多数浏览器也同样会在 JavaScript 中将其忽略，但 Internet Explorer 浏览器在这方面却相当吹毛求疵。

2. 文档就绪事件处理程序

一旦 DOM(document object model，文档对象模型)可供 JavaScript 代码使用，就会立即引发文档准备就绪事件。这个事件会在整个页面的内容加载完毕之前引发，这与文档载入事件不同

(一旦所有的 CSS、脚本和图像均获取完毕并准备就绪才会引发该事件)。一般而言，最好将初始化代码放在这里，这是因为此时已经可以访问DOM，而准备就绪事件引发的时机足够早，用户不必等待很长的时间才启动初始化。

jQuery 库使得编写在文档准备就绪事件引发时执行的函数变得极其容易。可以使用 jQuery 的 ready() 方法来创建一个文档准备就绪事件处理函数。下面的代码在文档准备就绪事件引发时会立即将相关情况告知用户。

```
$(document).ready(function () {
    alert("document ready");
});
```

注意，这个初始化函数并没有名称，而且这个函数直接被传入 ready() 函数。这样的函数被称为匿名函数，在使用诸如 jQuery 和 Strophe 这样的基于回调的库时，它们非常常见。

本书中的应用程序通常会在文档准备就绪事件处理程序中放置一些初始化代码，比如创建对话框以及设置交互事件处理程序。当文中说要向文档准备就绪事件处理程序中添加一些代码时，只需将相关代码放到这个初始化函数的末尾即可。

3. 自定义事件处理程序

jQuery 库使得创建和使用自定义事件变得非常容易。这些自定义事件通常用来提高代码的可读性并减少组件之间的耦合度。例如，我们并不是将连接 XMPP 服务器的全部代码放入用来处理用户单击 Connect 按钮事件的函数中，而是让单击事件处理程序触发一个名为 connect 的自定义事件，并将连接代码放入这个 connect 事件的处理程序中。

使用自定义事件使得以后扩展代码变得容易，而无须多处修改处理流程。如果为了响应刚刚描述的 connect 事件需要执行多项操作，那么只需要为该事件再添加一个包含新的处理逻辑的处理程序即可，而不需要增加登录对话框的代码的复杂性。

为了引发自定义事件，可以调用：

```
$(document).trigger('event_name', event_data);
```

事件名称可以是任意值，但最好不要使用与普通 DOM 事件相同的名称，比如 click 和 keypress。可以将 event_data 参数设置为任何希望传入事件处理程序的数据，或者如果不希望传入任何额外信息，可以彻底将其忽略。

使用 bind() 方法创建一个处理这些事件的处理程序。

```
$(document).bind('event_name', function (e, data) {
    var event_data = data;
});
```

这段代码将指定的函数设置为 event_name 事件的处理程序。传入的第一个参数是事件对象，与传给普通 DOM 事件的参数类似。第二个参数与传入 trigger() 的 event_data 相同。

稍后在开发 Hello 的初始代码时，我们将会看到命名空间对象、文档准备就绪事件处理程

序以及自定义事件处理程序如何进行交互。

3.5 建立连接

在 Hello 程序能够向服务器发送任何数据之前，它必须首先建立连接。我们将需要从用户那里收集 XMPP 身份凭证，然后使用 Strophe 连接到 XMPP 服务器。一旦建立连接，Strophe 就会使用指定的凭证进行身份验证并创建会话，从而可以在 XMPP 之上发送和接收数据。

3.5.1 连接生命周期

在第 1 章中我们曾经讨论过，XMPP 连接从被创建之后到被销毁之前要历经多个生命周期阶段。理解这些阶段非常重要，这是因为阶段转换通常需要重要的应用程序逻辑。例如，如果应用程序需要被其他对象连接才能完成有价值的操作，那么它会等待转换到已连接阶段。

当要求 Strophe 连接到服务器时，我们还为其提供了一个回调函数，每次连接阶段改变时 Strophe 都会调用该函数。这就允许应用程序来处理诸如连接启动、身份验证失败以及连接断开这样的事件。

表 3-1 给出了这些阶段变化和状态的完整列表。状态名称对应于 Strophe.Status 对象中定义的常量。例如，CONNECTED 状态就是 Strophe.Status.CONNECTED。

表 3-1 Strophe 连接状态

状态	描述
CONNECTING	Strophe 已经开始尝试建立通往 XMPP 服务器的连接
AUTHENTICATING	连接已经建立，Strophe 现在正尝试进行身份验证并建立会话
CONNECTED	会话已经建立，而且用户数据现在可以自由地流动了
DISCONNECTING	终止已经启动的连接
DISCONNECTED	连接被完全终止
CONNFAIL	Strophe 在试图建立连接时遇到问题
AUTHFAIL	在身份验证过程中发生错误

普通 XMPP 连接会依次经历前 5 个阶段，而应用程序将主要关注 CONNECTED 和 DISCONNECTED。通过观察 CONNFAIL 和 AUTHFAIL 状态可以处理连接中出现的错误。那些-ING 状态并不常用，但可以用在 UI 中提供可见的更新信息，这样用户就能够知道在他们等待的过程中正在发生什么事情。

3.5.2 创建连接

XMPP 连接通过 Strophe.Connection 对象进行管理。在第 2 章中，我们知道 BOSH 连接管理器在 HTTP 和 XMPP 之间提供了一座桥梁。BOSH 连接管理器以 URL 形式提供给 HTTP 协议客户端访问，而我们创建的 Strophe.Connection 对象需要知道这些 URL。

许多 XMPP 服务器都内置了对 BOSH 的支持，而且它们通常以 URL <http://example.com>:

5280/http-bind 或 <http://example.com:5280/xmpp-httpbind> 来提供该服务。有些 BOSH 连接管理器能够处理任意 XMPP 服务器的通信，但通常内置连接管理器只能与运行它们的服务器通信。

对于本书中应用程序的开发而言，可以随意使用 <http://bosh.metajack.im:5280/xmpp-httpbind> 的 BOSH 连接管理器。这个 BOSH 连接管理器能够与任何公开 XMPP 服务器通信。这个服务器专门提供给本书的读者在开发时使用，以避免每个人都要搭建自己的 BOSH 服务。

可以使用 `new` 关键字来创建一个新的 `Strophe.Connection` 对象，就像任何其他 JavaScript 对象一样。

```
Var conn=new Strophe.Connection("http://bosh.metajack.im:5280/xmpp-httpbind");
```

一旦有了连接对象，就可以调用 `connect()` 和 `disconnect()` 来启动和关闭与服务器的通信。

```
// starting a connection to example.com
conn.connect("user@example.com", "mypassword", my_callback);

// disconnecting
conn.disconnect();
```

`connect()` 的前两个参数是用来验证该会话的 JID 和口令，最后一个参数是前面曾经讨论过的回调函数。该回调函数被调用时只携带一个参数，该参数被设为前一节中描述的那些状态中的一个。下面给出了一个简单的回调函数，一旦连接到达 CONNECTED 阶段，它就会断开连接。

```
function my_callback(status) {
    if (status === Strophe.Status.CONNECTED) {
        conn.disconnect();
    }
}
```

每次连接改变其状态时，都会执行这个回调函数。`my_callback()` 函数简单地忽略除 CONNECTED 之外的所有状态，一旦连接到达该状态，就立即断开连接。

启动和关闭连接并不涉及太多工作。可以运用所学的这个新知识来实现 Hello 程序的登录对话框。

3.5.3 连接 Hello

为了建立连接，首先必须收集用户的身份凭证。登录对话框就是为此而设计的，但它最初是被隐藏起来的，并且需要创建并显示出来。可以使用 jQuery UI 的 `dialog()` 函数来显示和隐藏这个对话框，并利用自定义事件使用 `Strophe` 来启动连接。

在希望转换成对话框的元素上调用 `dialog()` 函数，并为其提供一个用来定义该对话框行为的属性列表。虽然这里并不完整地讲解大多数属性，但是从它们的名称中非常容易弄明白它们的功能。jQuery UI 文档 <http://ui.jquery.com/> 完整地讲解了所有的属性。

创建 `hello.js` 文件并向其中添加如下代码。



可以从
Wrox.com
下载源代码

```

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Connect to XMPP',
        buttons: {
            "Connect": function () {
                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val()
                });

                $('#password').val('');
                $(this).dialog('close');
            }
        }
    });
});

```

code snippet hello.js

最重要的属性是 `buttons`, 它定义了对话框的按钮以及这些按钮被单击时执行的动作。这里只定义了一个名为 `Connect` 的按钮。当单击 `Connect` 按钮时, 就会触发一个名为 `connect` 的自定义事件, 而 JID 和口令将随之传入该事件的处理程序。一旦该事件被触发, 口令字段就会被清空, 并关闭该对话框。当执行为 `Connect` 按钮定义的函数时, `this` 对象将被设为对话框的主元素 `#login_dialog`。可以利用 `jQuery` 将其包装起来以便容易地访问对话框的其他方法, 就像上面代码中的 `close` 方法。

接下来, 需要为 `connect` 事件创建一个处理程序: 创建一个新的 `Strophe.Connection` 对象并调用 `connect()` 方法。还需要提供一个能够响应连接状态变化的回调函数。

将下面的自定义事件处理程序添加到前面编写的文档就绪事件处理函数中。



可以从
Wrox.com
下载源代码

```

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        "http://bosh.metajack.im:5280/xmpp-httpbind");
    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });
});

$(document).bind('connected', function () {
    // nothing here yet
});

```

```

$(document).bind('disconnected', function () {
    // nothing here yet
});

```

code snippet hello.js

在 `connect` 事件的处理程序中，我们创建一个新的连接对象，调用 `connect()` 方法，并为其提供了一个回调函数，该函数只是触发新的自定义事件。每当 Strophe 通知该回调函数进入 `CONNECTED` 状态时，就会引发 `connected` 事件，而进入 `DISCONNECTED` 状态时引发 `disconnected` 事件。

接下来的两个事件处理程序被绑定到这两个新的自定义事件，但是什么工作也不做。

这种连接事件设计是本书中的应用程序使用的一种模式。自定义事件已经使得我们能够轻易地将连接过程的不同部分区分开，并且在不影响其他部分的情况下修改任何一个处理程序。

在继续 Hello 的下一部分之前，我们还需要添加最后一点内容。日志显示区域用来提醒用户在应用程序运行时发生的事情，因此应该创建一个函数向这个区域写入消息，并在 `connected` 和 `disconnected` 事件触发时记录日志。

首先为 Hello 程序创建一个命名空间对象，并添加 `connection` 属性和 `log()` 函数。`connection` 属性将用来存放活动连接对象，这样后面就能够访问该对象。`log()` 函数简单地使用消息来更新日志显示区域。下面的代码在该文件顶部的文档准备就绪事件处理程序之前运行。



可从
Wrox.com
下载源代码

```

var Hello = {
    connection: null,
    log: function (msg) {
        $('#log').append("<p>" + msg + "</p>");
    }
};

```

code snippet hello.js

`connection` 属性被初始化为 `null`，但我们将需要将创建好的连接对象指派给它，当连接终止时再次将其设为 `null`。我们还应该将日志消息添加到 `connected` 事件处理程序。下面给出了修改之后的事件处理程序，其中修改部分已被突出显示。



可从
Wrox.com
下载源代码

```

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        "http://bosh.metajack.im:5280/xmpp-httpbind");
    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });
    Hello.connection = conn;
});

```

```

});
```

```

$(document).bind('connected', function () {
    // inform the user
    Hello.log("Connection established.");
});
```

```

$(document).bind('disconnected', function () {
    Hello.log("Connection terminated.");

    // remove dead connection object
    Hello.connection = null;
});
```

code snippet hello.js

Hello 程序并没有完成太多工作，但它应该已经为第一次测试运行做好准备了。如果需要检验是否所有的代码均已就位，那么可以参照程序清单 3-3 给出的目前已经构建的完整的 hello.js 文件。



可从
Wrox.com
下载源代码

程序清单 3-3 hello.js(初始版本)

```

var Hello = {
    connection: null,

    log: function (msg) {
        $('#log').append("<p>" + msg + "</p>");
    }
};

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Connect to XMPP',
        buttons: {
            "Connect": function () {
                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val()
                });

                $('#password').val('');
                $(this).dialog('close');
            }
        }
    });
});
```

```

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        "http://bosh.metajack.im:5280/xmpp-httpbind");
    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });
    Hello.connection = conn;
});
$(document).bind('connected', function () {
    // inform the user
    Hello.log("Connection established.");
});
$(document).bind("disconnected", function () {
    Hello.log("Connection terminated.");
    // remove dead connection object
    Hello.connection = null;
});

```

3.5.4 运行应用程序

因为 Hello 程序是一个仅由 HTML、CSS 和 JavaScript 代码组成的 Web 应用程序，所以运行和测试它是相当简单的。尽管 fIXHR 库可用来执行跨域请求而无需任何服务器设置，但它并不允许我们在 file://这样的 URL 下面运行，却请求 http:// URL 的应用程序。这意味着我们将需要使用 Web 服务器通过 HTTP 来服务 Hello 程序，并将浏览器指向 http:// URL。

这并不复杂，任何 Web 浏览器都能胜任，而且任何廉价的托管服务也都应该很好地胜任该工作。任何平台都有易于安装的 Web 服务器，而且有些操作系统已经将它们内置其中。还可以使用 Tape 程序来处理文件。附录 B 描述了 Tape。

一旦有一个正在运行的 Web 服务器或已将代码上传至中意的 Web 主机，那么可以让浏览器打开 hello.html 在该服务器上的 URL，并试图登录。应该会看到一条内容为 Connection established 的消息。

如果出现问题，那么请检查 Web 浏览器的错误控制台，确保代码没有抛出一条没有被看到的错误。此外，应确信已经安装了所有的依赖文件并且该文件已被放入了正确的位置。

3.6 创建节

构建 XMPP 节是编写 XMPP 应用程序最重要的部分之一。即使可以使用插件来处理大多数 XMPP 协议工作，但可能还是需要创建一些自己的 XMPP 节。Strophe 有协议强大的 XMPP 节构建工具可以让这个工作尽可能变得轻松。

XMPP 节只是部分 XML 文档。当然可以使用浏览器自己的 DOM 操纵函数来构建 XMPP 节, 但 DOM API 相当乏味, 而且在不同浏览器之间还稍有不同。Strophe 将这些 API 抽象出来(与 jQuery 抽象出同样 API 所用的方式相同), 因此可以关注创建自己所需的 XMPP 节, 而无须担心全部细节。

3.6.1 Strophe 构建器

构建 XMPP 节是 `Strophe.Builder` 对象的任务。这些对象受到 jQuery 的启发, 使得构建 XMPP 节的速度极快而且非常容易。此外, Strophe 还提供了几个快捷函数来完成常见的构造操作。

创建构建器时携带两个参数: 一个元素名称和一组属性。在下面的代码中, 第一行创建 `<presence>` 节, 而第二行创建 `<presence to='example.com'>` 节。

```
var pres1 = new Strophe.Builder("presence");
var pres2 = new Strophe.Builder("presence", {to: "example.com"});
```

因为构建 XMPP 节是一个相当常见的操作, 而且输入 `new Strophe.Builder` 需要多次按下按键, 所以 Strophe 为 XMPP 节的创建提供了四个全局别名: `$build()`、`$msg()`、`$pres()` 和 `$iq()`。这些函数的代码就是像上面的示例那样创建 `Strophe.Builder` 对象。函数 `$build()` 携带的参数与 `Strophe.Builder` 的构造方法的参数一样。其他三个函数分别负责创建 `<message>`、`<presence>` 和 `<iq>` 节, 均可携带一个表示预期属性的可选实参。

虽然可以使用 `$build()` 或 `$pres()` 来缩短上面的代码, 但后者最常用。

```
var pres1 = $build("presence");
var pres2 = $build("presence", {to: "example.com"});
var pres3 = $pres();
var pres4 = $pres({to: "example.com"});
```

可以通过链式函数调用来构建更复杂的 XMPP 节。就像 jQuery 一样, 当调用 Strophe 的构建快捷方法时, 它们返回的是构建器对象。构建器对象上的大多数方法(稍后我们就会看到)也会返回构建器对象。这意味着, 我们可以将方法调用放在一个链条中, 就像在 jQuery 中那样。如果这听起来有点困惑, 请不要担心, 我们将会看到大量的示例。

构建器对象的所有方法都有一个缩写名称以节省键盘录入工作。返回构建起对象的链式方法分别是 `c()`、`cnode()`、`t()`、`attrs()` 和 `up()`。还有两个并不返回构建器对象的方法, 它们是 `toString()` 和 `tree()`。

`toString()` 方法将 XMPP 节序列化成文本。这对于调试而言非常有用。例如, `$pres().toString()` 返回字符串 “`<presence>`”。`tree()` 方法所做的工作与此类似, 但返回的是位于 XMPP 节元素树顶部的 DOM 元素。通常并不会用到这个方法, 但如果需要访问在构建器对象内部创建的 DOM 元素, 就可以使用这个方法。

可以使用 `c()` 和 `cnode()` 方法向 XMPP 节中添加新的子元素。前者携带的参数与 `$build()` 相同, 它将新的子元素追加到当前元素后面。后一个方法所做工作与此相同, 但它只携带一个 DOM 元素作为输入参数。大多数时候 `c()` 将更加高效, 但 `cnode()` 偶尔可用来复制或重用已经构建的

XMPP 节片段。

因为这些方法返回构建器对象，所以可以轻易地将它们链接起来。

```
var stanza = $build("foo").c("bar").c("baz");
```

调用 `stanza.toString()` 将返回如下内容。

```
<foo><bar><baz/></bar></foo>
```

每当添加一个子元素时，构建器的当前元素就会变成这个新的子元素。如果希望在同一个元素上创建多个子元素，那么必须在每次调用 `c()` 之后在元素树中向上后退一级。可以使用 `up()` 方法来完成这件工作。

可以将 `up()` 方法添加到前面的示例中来构建一个稍有不同的 XMPP 节。

```
var stanza = $build("foo").c("bar").up().c("baz");
```

这段代码产生的 XMPP 节如下所示。

```
<foo><bar/><baz/></foo>
```

文本子元素是通过 `t()` 方法添加的。与类似的 `c()` 方法不同的是，`t()` 方法并不会改变当前元素。使用 `$msg()`、`c()` 和 `t()` 方法可以创建一条典型的 XMPP 消息，如下面的示例所示。

```
var message = $msg({to: "darcy@pemberley.lit", type: "chat"})
  .c("body").t("How do you do?");
```

这个构建器生成的 XMPP 节如下所示。

```
<message to='darcy@pemberley.lit'
  type='chat'>
  <body>How do you do?</body>
</message>
```

构建器对象的最后一个方法是 `attrs()`，它携带一个属性集合，并用它来增强当前元素的属性集合。如果部分 XMPP 节是由其他的代码构建的，而且我们在通过连接将其发送出去之前需要再添加一些最终的属性，就可以使用这个方法。这个方法并不常用，但在抽象 XMPP 节构建功能时这个方法相当方便。

下面的代码给出了几个构建器，它们以及它们生成的 XMPP 节都更加复杂一些。

```
var iq = $iq({to: "pemberley.lit", type: "get", id: "disco1"})
  .c("query", {xmlns: "http://jabber.org/protocol/disco#info"});
  // produces:
  // 
  // <iq to='pemberley.lit'
  //   type='get'
  //   id='disco1'
  //   xmlns='http://jabber.org/protocol/disco#info'>
  //   <query xmlns='http://jabber.org/protocol/disco#info'/>
  // </iq>
```

```

//      type='get'
//      id='disco1'
// <query xmlns='http://jabber.org/protocol/disco#info' />
// </iq>

var presence = $pres().c("show").t("away").up()
  .c("status").t("Off to Meryton");

// produces
//
// <presence>
//   <show>away</show>
//   <status>Off to Meryton</status>
// </presence>

```

3.6.2 打招呼

Hello 应用程序需要向服务器发送一个 XMPP 节向其道一声 hello，我们可以通过使用 Strophe 构建器函数来完成这件工作。hello 节是一个包含一个 ping 请求的 IQ-get 节。

一旦连接准备好接受数据，就会用状态 CONNECTED 来调用该连接的回调函数。这会触发已经连接的事件并调用附加的处理程序。而这正是向服务器发送 ping 请求的最好时机。

下面给出了修改后的事件处理程序。



可从
Wrox.com
下载源代码

```

$(document).bind('connected', function () {
  // inform the user
  Hello.log("Connection established.");

  var domain = Strophe.getDomainFromJid(Hello.connection.jid);

  Hello.send_ping(domain);
});

```

code snippet hello.js

还需要将 send_ping() 函数添加到命名空间对象中。



可从
Wrox.com
下载源代码

```

send_ping: function (to) {
  var ping = $iq({
    to: to,
    type: "get",
    id: "ping1"
  }).c("ping", {xmlns: "urn:xmpp:ping"});

  Hello.connection.send(ping);
}

```

code snippet hello.js

这里使用了一些以前没有见过的新技术。连接对象的 send() 方法向服务器发送一个 XMPP

节。连接对象的 `jid` 属性包含着与该连接相关的完整 JID。Strophe 对象中有几个方法可以让 JID 的处理更加简单：`getUserFromJid()`、`getDomainFromJid()`、`getResourceFromJid()` 和 `getBareJidFromJid()`。 JID 辅助器函数返回 JID 的不同部分。下面的代码给出如何使用这些函数以及它们的返回值。

```
Strophe.getUserFromJid("darcy@pemberley.lit/library"); // "darcy"
Strophe.getDomainFromJid("darcy@pemberley.lit/library"); // "pemberley.lit"
Strophe.getResourceFromJid("darcy@pemberley.lit/library"); // "library"
Strophe.getBareJidFromJid("darcy@pemberley.lit/library"); // "darcy@pemberley.li"
```

Hello 程序现在向用户的服务器发送一个 ping，但它还没有对服务器的响应进行任何处理。在本章最后部分我们将学习如何处理传入的 XMPP 节。

3.7 处理事件

大多数 XMPP 应用程序都是由事件驱动的。有些事件是由用户的交互活动所触发的，比如单击鼠标或按下键盘按键，而有些事件是由传入的 XMPP 节触发的。例如，当接收到一条消息时，应用程序进行处理，将其显示给用户。处理传入的 XMPP 节可能是任何 XMPP 应用程序最重要的部分，而 Strophe 使这件工作变得相当简单。

3.7.1 添加和删除处理程序

可以使用 `addHandler()` 来添加新的 XMPP 节处理程序，而使用 `deleteHandler()` 则可将其删除。下面的代码演示了使用这些函数的基本知识。

```
var ref = conn.addHandler(my_handler_function, null, "message");
// once the handler is no longer needed:
connection.deleteHandler(ref);
```

`addHandler()` 函数返回一个处理程序引用。这个引用只用来传入 `deleteHandler()` 以识别待删除的特定处理器。

`deleteHandler()` 函数并不常用，这是因为处理程序函数有办法在不再需要时将自己删除。在某些情况下，在处理程序中系统并不知道什么时候删除处理程序，此时就可以使用 `deleteHandler()` 函数。

3.7.2 节匹配

`addHandler()` 函数携带一个或多个参数。第一个参数是当接收到匹配的 XMPP 节时调用的函数。剩余的参数是匹配条件。下面这段摘自 Strophe 源代码的经过精简的函数定义给出了这些参数的完整列表。

```
addHandler: function (handler, ns, name, type, id, from) {
  // implementation omitted
}
```

如果任何一个条件为 `null` 或未定义, 那么匹配所有 XMPP 节。否则, 只有当它们满足条件(XMPP 节中特定部分满足字符串相等条件)时才会匹配 XMPP 节。最后四种条件(`name`、`type`、`id` 和 `from`)指定 XMPP 节元素名称和类型的过滤器。只有在顶级元素(而不是该元素的任何子元素)上才会检查这四个条件。第一个条件 `ns` 稍有不同, 它用于检查顶级元素以及它的直接子元素。读者稍后就会明白其中的缘由。

`name` 条件几乎总是为 `null` 以便匹配任何 XMPP 节, 否则只能是 `message`、`presence` 或 `iq` 之一。在接收到任何`<message>`节时, 我们在 `addHandler()`示例中设置的处理程序都会被调用。

`type`、`id` 和 `from` 条件匹配`<message>`、`<presence>`和`<iq>`节的主要属性。可以使用 `type` 来区分普通聊天消息和群聊消息, 或者将 IQ-result 节与 IQ-error 节独立开。`id` 条件通常用来处理特定请求的应答, 比如与特定 IQ-get 请求相关联的 IQ-result。匹配 `from` 属性会限制处理程序只处理来自特定 JID 的消息。应该慎用 `from` 条件, 这是因为我们并不会总能知道其他用户或服务会使用哪一个资源来与我们进行通信。如果将裸 JID 作为 `from` 条件, 那么不会匹配任何完整 JID, 这是因为这里采用的是精确匹配方式。

匹配 `ns`(命名空间)条件基本上是针对 IQ 节的。IQ 节通常包含一个子元素, 这个子元素的命名空间是根据它服务的功能类型确定的。

例如, 前面发送的 `ping` 节就是一个携带了一个位于 `urn:xmpp:ping` 命名空间下面的`<ping>`子元素的 IQ-get 节。下面的代码设置了一个处理程序来捕获所有传入的 `ping` 请求。

```
conn.addHandler(my_ping_handler, "urn:xmpp:ping", "iq");
```

每当连接接收到一个携带着一个位于 `urn:xmpp:ping` 命名空间下面的子元素的 IQ 节时, 都会调用 `my_ping_handler()`函数。它将获得所有这些 XMPP 节, 而不管它们的 `type`、`id` 或 `from` 属性, 这是因为这些条件都未被指定。

3.7.3 节处理程序函数

每当找到某个处理程序的匹配节时, 都会调用该处理程序函数, 并将该节作为它的实参传入。除非该函数返回 `true` 或者其他计算结果为 `true` 的表达式, 否则一旦完成就会将该处理函数删除。

下面示例中的 XMPP 处理程序被称为一次性处理程序, 这是因为它返回 `false`。当这个处理程序完成第一个节的处理之后, 它将会被删除, 而且不会被再次调用, 除非再次调用 `addHandler()` 明显式地将其再次添加进来。

```
function my_ping_handler(iq) {
    // do something interesting
    return false;
}
```

如果该函数完全没有使用 `return` 语句, 那么它返回 `undefined`, 这与返回 `false` 具有同样的效果。如果发现自己的处理程序停止运行, 就应检查它们的返回值。

下面的示例给出了一个返回 `true` 的处理程序, 它能够处理接收到的所有 XMPP 节。`handle_incoming_ping()`函数用 `pong` 来响应传入的 `ping` 请求。

```

function handle_incoming_ping(iq) {
    // conn is assumed to be a global pointing to a valid
    // Strophe.Connection object
    var pong = $iq({to: $(iq).attr('from'), type: "result", id: $(iq).attr('id')});
    conn.send(pong);

    return true;
}

```

3.7.4 处理 Hello 响应

Hello 程序所需的最后一部分是处理服务器对 ping 请求的应答。可以运用我们学到的 XMPP 节处理程序的新知识来实现这部分功能。与大多数 ping 请求一样，我们可以用它来测量接收一个响应需要花多长时间，因此我们也可以向 Hello 程序中添加一些计时用的代码。

通常，在发送初始请求之前就应该添加响应处理程序。这有助于避免下面形成的竞争条件：服务器过快地产生响应，以至于我们的代码尚未及时地添加处理程序来捕获该响应。

修改 connected 事件处理程序以匹配下面给出的代码。



可从
Wrox.com
下载源代码

```

$(document).bind('connected', function () {
    // inform the user
    Hello.log("Connection established.");

    Hello.connection.addHandler(Hello.handle_pong, null, "iq", null, "ping1");

    var domain = Strophe.getDomainFromJid(Hello.connection.jid);

    Hello.send_ping(domain);
});

```

code snippet hello.js

还需要对 send_ping() 函数进行如下的修改。



可从
Wrox.com
下载源代码

```

send_ping: function (to) {
    var ping = $iq({
        to: to,
        type: "get",
        id: "ping1"
    }).c("ping", {xmlns: "urn:ietf:params:xml:ns:xmpp-ping"});

    Hello.log("Sending ping to " + to + ".");
    Hello.start_time = (new Date()).getTime();
    Hello.connection.send(ping);
}

```

code snippet hello.js

在代码发送 XMPP 节之前添加处理函数，并且将发送请求的时间记录在 Hello 命名空间对

象的 `start_time` 属性中。这个新版本还向日志中添加了另一条消息，这样用户就能够看到正在发生什么情况。

还需要将这个新属性和 `handle_pong()` 函数添加到 `Hello` 命名空间对象中。下面的代码给出了这些新加的部分。



```
start_time: null,
handle_pong: function (iq) {
    var elapsed = (new Date()).getTime() - Hello.start_time;
    Hello.log("Received pong from server in " + elapsed + "ms");
    Hello.connection.disconnect();
    return false;
}
```

code snippet hello.js

在添加最后这部分之后，`Hello` 程序就应该能够完整地运行了。使用浏览器打开 `hello.html` 以运行 `Hello` 程序。服务器的响应速度如何？程序清单 3-4 给出了最终版本的 `hello.js`。



程序清单 3-4 hello.js(最终版本)

```
var Hello = {
    connection: null,
    start_time: null,
    log: function (msg) {
        $('#log').append("<p>" + msg + "</p>");
    },
    send_ping: function (to) {
        var ping = $iq({
            to: to,
            type: "get",
            id: "ping1"
        }).c("ping", {xmlns: "urn:xmpp:ping"});
        Hello.log("Sending ping to " + to + ".");
        Hello.start_time = (new Date()).getTime();
        Hello.connection.send(ping);
    },
    handle_pong: function (iq) {
        var elapsed = (new Date()).getTime() - Hello.start_time;
        Hello.log("Received pong from server in " + elapsed + "ms.");
        Hello.connection.disconnect();
        return false;
    }
}
```

```

};

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Connect to XMPP',
        buttons: {
            "Connect": function () {
                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val()
                });
                $('#password').val('');
                $(this).dialog('close');
            }
        }
    });
});

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        "http://bosh.metajack.im:5280/xmpp-httpbind");
    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });
    Hello.connection = conn;
});

$(document).bind('connected', function () {
    // inform the user
    Hello.log("Connection established.");

    Hello.connection.addHandler(Hello.handle_pong,
        null, "iq", null, "ping1");

    var domain = Strophe.getDomainFromJid(Hello.connection.jid);

    Hello.send_ping(domain);
});

$(document).bind('disconnected', function () {
    Hello.log("Connection terminated.");

    // remove dead connection object
    Hello.connection = null;
});

```

3.8 给 Hello 程序添加新功能

对于简单的应用程序而言，最好的地方在于它们最容易改进。试着向 Hello 程序中添加如下的功能：

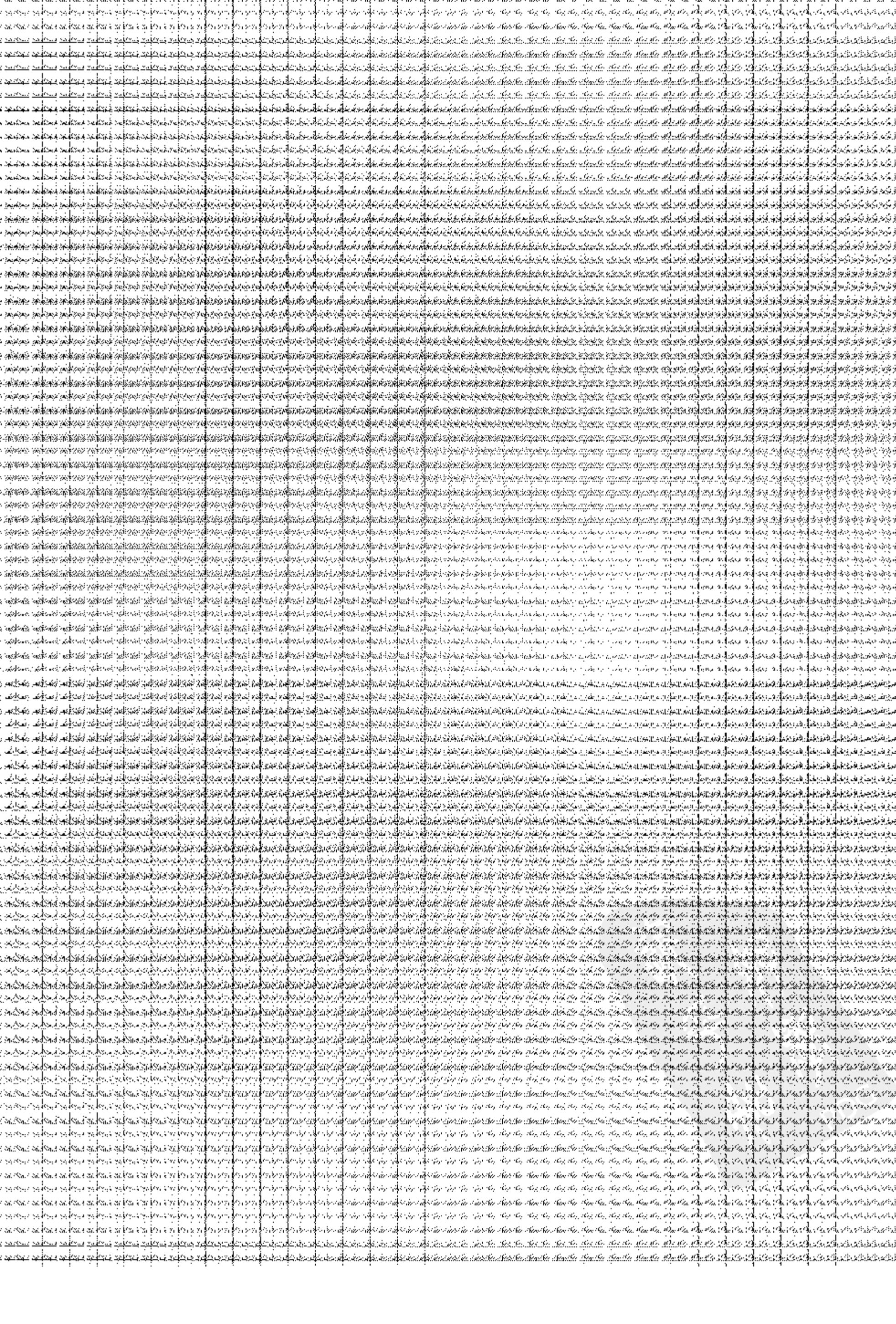
- 单次测量并不具有代表性，修改 Hello 程序以执行多次 ping 请求并测量平均响应时间。
- 如果在前面的任务中使用了串行方法测量，那么试着使用并行方法。如果已经建立并行版本，那么试着让它串行运行。

3.9 小结

我们已经编写了第一个 Strophe 应用程序，而且这可能也是第一个 XMPP 应用程序。可以看到，利用 Strophe 和 jQuery 来处理 XMPP 是相当简单的。我们已经学习了如下内容：

- 如何获取基于 Strophe 的 XMPP 应用程序所需的所有文件。
- 如何测试和运行 XMPP 应用程序。
- 如何建立连接和终止连接。
- 如何向服务器发送数据。
- 如何处理传入的 XMPP 节。
- 如何使用 jQuery 自定义事件。

在掌握这些基础知识之后，我们就可以在第 4 章中继续编写第一个有用的 XMPP 应用程序。



4

第 4 章

探索 XMPP 协议：一个调试控制台

本章内容

- 如何钩入 Strophe 的日志设施
- 分析 XML
- 操纵出席信息
- 查询软件版本
- 处理 XMPP 错误

开发人员总是喜欢不断地加工并改善自己的工具。在开发 XMPP 应用程序的过程中，我们将需要一款工具来辅助研究和查看协议流量。要是不使用查看源代码(view source)命令或者不能轻易加工 URL 来测试远程站点的功能，那么很少有 Web 开发人员能够轻松工作。对于 XMPP 节，这样的工具可用来查看协议流量并轻易地创建要发送的节。在本章中我们将构建一个名为 `Peek` 的协议调试控制台，以后我们将使用它来研究一些 XMPP 服务以及典型的协议条件。

`Peek` 在整本书中都非常有用。每当遇到一个示例节时，都可以加载 `Peek`，输入 XMPP 节，然后观察会有什么响应。这样一来，即使没有开始构建应用程序，我们仍然可以试验各种 XMPP 功能。

构建 `Peek` 应用程序所需的许多部分都已经介绍过了：建立连接、发送 XMPP 节、设置基本的处理程序来处理传入的流量。但 `Peek` 还需要用到 `Strophe` 的一些新功能，而且一旦 `Peek` 构建起来之后，我们就可以用它来研究一些新的 XMPP 概念。

4.1 应用程序预览

图 4-1 给出了完工之后的应用程序，这是一个拥有彩色终端以及一个带有精美的突出显示功能的代码编辑器。

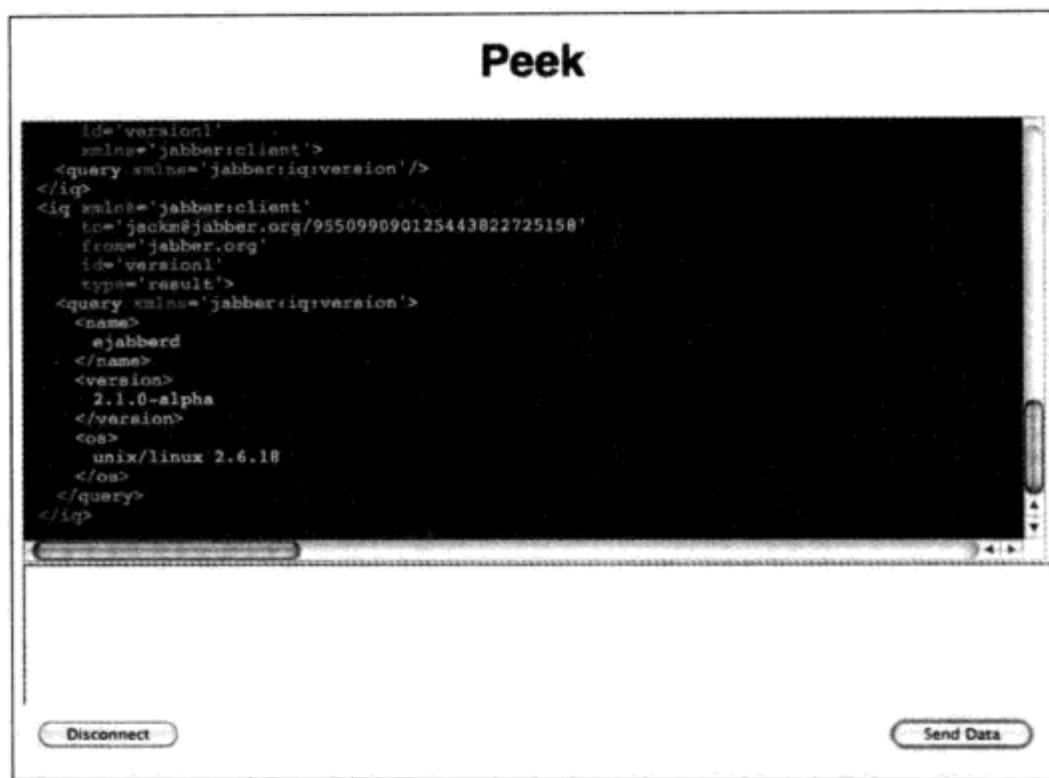


图 4-1

带有黑色背景的顶部区域用来显示所有发送到服务器以及从服务器那里接收到的 XMPP 节。它下面的输入域用来接受 XML 输入或 Strophe 的 XMPP 节构建命令。

4.2 设计 Peek

调试控制台是使用 XMPP 构建的最简单的程序之一。Peek 需要能够发送流量以及显示流量，但它只需要一点点最简单的流量处理逻辑。设法美化用户界面是主要工作。

Strophe 中的每个连接都有一些特殊的函数来钩入正在发送和接收的协议数据。这些函数有两种形式——`xmlInput()`和`xmlOutput()`——用于结构化流量，而`rawInput()`和`rawOutput()`用于实际的字符数据。通常，这些函数不执行任何操作，但 Peek 将覆盖这些函数以获取进入和离开连接的数据流视图。

当一个连接首次建立时，Strophe 会在后台发送并接收几个XMPP 节以处理身份验证和会话建立工作。只有当会话成功启动时才会获得 CONNECTED 状态。我们将首先处理流量显示，这样当这些 XMPP 设置节出现时就能够看到它们。

XMPP 流携带着 XML 数据，但这个 XML 数据并没有为显示而进行格式化处理。实际上，大多数 XMPP 服务器、客户端以及库发送的 XML 数据都已经将其中的不必要的空格符去掉。

通常 XMPP 节非常像一个巨大的文本字符串。下面是一个带有良好格式的示例 XMPP 节，而其后则是一段通常会看到的 XMPP 节。

```
<message to='darcy@pemberley.lit/meryton'
         from='bingley@netherfield.lit/meryton'
         type='chat'>
<body>Come, Darcy, I must have you dance. I hate to see you standing about by
         yourself in this stupid manner. You had much better dance.</body>
</message>
```

```
<message to='darcy@pemberley.lit/meryton' from='bingley@netherfield.lit/meryton'
type='chat'><body>Come, Darcy, I must have you dance. I hate to see you standing
about by yourself in this stupid manner. You had much better dance.</body>
</message>
```

对于较短的 XMPP 节来说，这种差异并不太大，但对于较长的 XMPP 节以及大量的 XMPP 节来说，后者几乎不具有可读性。

我们将使用 `xmlInput()` 和 `xmlOutput()` 来获取结构化的流量，然后使用 HTML 和 CSS 将其包装起来以便使用语法突出显示和额外的空格符来显示 XMPP 流量。我们并不会详细地讲解这里的显示转换代码，这是因为它与我们的目标无关，但该代码易于理解，如果感兴趣的话，那么可以对其进行修改。

在搭建好控制台并接收协议流量之后，我们将注意力转向用户输入的处理。首先让用户手工输入 XML 节。因为 `Strophe` 只接受真正的、要通过连接发送的 XML 数据，所以必须将输入文本分析成 XML，然后将其通过连接发送出去。我们将使用 Web 浏览器的原生 XML 分析功能来完成这件工作。

输入 XML 代码是一件相当无聊的事情，因此我们添加一个功能，以便让用户能够使用 `Strophe` 自带的 XMPP 节构建命令，比如 `$msg()`、`$pres()` 和 `$iq()`。`Peek` 将使用 JavaScript 的 `eval()` 函数来执行该代码。

这听起来非常简单，实际上也确实如此。但在试验和调试应用程序以及与我们交互的服务器时，`Peek` 极其有用。

4.3 构建控制台

`Peek` 将使用与第 3 章中的 `Hello` 应用程序相同的应用程序结构。我们首先需要创建用户界面，构建 `peek.html` 和 `peek.css`，然后采用 JavaScript 来创建应用程序逻辑。本节末尾给出了最终的源代码，万一遇到问题可以查阅该源代码。

4.3.1 用户界面

`Peek` 的用户界面极其简单。它由一个用来显示协议流量的区域、一个提供给用户创建输出 XMPP 节的文本区域输入框以及几个按钮组成。同时还包含一个登录对话框，但跟第 3 章中介绍的一样，它最初是被隐藏起来的。程序清单 4-1 给出了包含这些元素的最初版本的 HTML 代码。



程序清单 4-1 peek.html

可从
Wrox.com
下载源代码

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
  .
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=UTF-8" />
    <title>Peek - Chapter 4</title>
```

```

<link rel='stylesheet' href='http://ajax.googleapis.com/ajax/libs/
    /jqueryui/1.7.2/themes/cupertino/jquery-ui.css'>
<script src='http://ajax.googleapis.com/ajax/libs/jquery/1.3.2
    /jquery.js'></script>
<script src='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2
    /jquery-ui.js'></script>
<script src='scripts/strophe.js'></script>
<script src='scripts/flXHR.js'></script>
<script src='scripts/strophe.flxhr.js'></script>

<link rel='stylesheet' type='text/css' href='peek.css'>
<script type='text/javascript' src='peek.js'></script>
</head>
<body>
<h1>Peek</h1>

<div id='console'></div>
<textarea id='input' class='disabled'
    disabled='disabled'></textarea>

<div id='buttonbar'>
    <input id='send_button' type='submit' value='Send Data'
        disabled='disabled' class='button'>
    <input id='disconnect_button' type='submit' value='Disconnect'
        disabled='disabled' class='button'>
</div>

<!-- login dialog -->
<div id='login_dialog' class='hidden'>
    <label>JID:</label><input type='text' id='jid'>
    <label>Password:</label><input type='password' id='password'>
</div>
</body>
</html>

```

程序清单 4-2 给出了最初的 CSS 代码。唯一不太显而易见的样式是.incoming，它用于控制台中，以区分传入的流量与输出的流量。



程序清单 4-2 peek.css

可从
Wrox.com
下载源代码

```

body {
    font-family: Helvetica;
}

h1 {
    text-align: center;
}

#console {
    padding: 10px;
}

```

```

height: 300px;
border: solid 1px #aaa;

background-color: #000;
color: #eee;
font-family: monospace;

overflow: auto;
}

#input {
width: 100%;
height: 100px;
font-family: monospace;
}

.incoming {
background-color: #111;
}

textarea.disabled {
background-color: #bbb;
}

#buttonbar {
margin: 10px;
}

#disconnect_button {
float: left;
width: 100px;
}

#send_button {
float: right;
width: 100px;
}

```

本节稍后我们将向 `peek.css` 中添加几个新样式。

4.3.2 显示流量

既然我们已经构建了一个简单的用户界面，下面就让我们把登录对话框加进来并钩入 `Strophe` 的日志函数以便让 XMPP 节出现在控制台中。

首先，需要创建登录对话框，并且在页面加载时将其打开。我们可以在响应文档准备就绪事件时来完成这件工作，就像在第 3 章中所做的那样。将下面的代码放进 `peek.js` 文件中。



可从
Wrox.com
下载源代码

```

$(document).ready(function () {
  $('#login_dialog').dialog({
    autoOpen: true,
    draggable: false,
    modal: true,
  });
});

```

```

        title: 'Connect to XMPP',
        buttons: {
            "Connect": function () {
                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val()
                });

                $('#password').val('');
                $(this).dialog('close');
            }
        });
    });
}

```

code snippet peek.js

jQuery UI 的 `dialog()` 函数将指定的`<div>`元素转换成一个会自动打开的模态对话框。我们添加了一个 `Connect` 按钮，它将引发 `connect` 事件，然后将该对话框关闭。

在文档准备就绪事件处理程序之后添加下面的处理程序。



可从
Wrox.com
下载源代码

```

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        'http://bosh.metajack.im:5280/xmpp-httpbind');

    conn.xmlInput = function (body) {
        Peek.show_traffic(body, 'incoming');
    };
    conn.xmlOutput = function (body) {
        Peek.show_traffic(body, 'outgoing');
    };

    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });
    Peek.connection = conn;
});

```

code snippet peek.js

这里的大部分代码我们在第 3 章中都已经看到过。被突出显示的代码行是新代码，它们将连接对象的空白的、不执行任何操作的默认日志函数改写为我们自己的版本。这里并没有指向两个几乎相同的函数，而是使用行内函数携带正确的参数来调用 `show_traffic()`。

将下面的 `Peek` 对象的实现(包括 `show_traffic()` 和 `xml2html()` 函数)添加到文档准备就绪事件

处理程序的后面。



```

var Peek = {
    connection: null,
    show_traffic: function (body, type) {
        if (body.childNodes.length > 0) {
            var console = $('#console').get(0);
            var at_bottom = console.scrollTop >= console.scrollHeight -
                console.clientHeight;

            $.each(body.childNodes, function () {
                $('#console').append("<div class='" + type + "'>" +
                    Peek.xml2html(Strophe.serialize
                    (this)) + "</div>");
            });

            if (at_bottom) {
                console.scrollTop = console.scrollHeight;
            }
        }
    },
    xml2html: function (s) {
        return s.replace(/&/g, "&")
            .replace(/</g, "<")
            .replace(/>/g, ">");
    }
};

```

code snippet peek.js

Peek对象将作为所有应用程序代码和状态的命名空间容器。这是一种很好的编程实践，它将我们的代码与其他应用程序隔离开。目前，该应用程序拥有的唯一的应用程序状态是connection变量。

Web浏览器自身并不能理解XMPP协议(至少目前不会)，因此XMPP连接必须通过HTTP请求建立隧道。这种隧道技术的一个副作用就是要把XMPP节放入到<body>包装器中，该包装器包含了一些有关请求的元数据。这种隧道技术(名为BOSH)正是文档XEP-0124和XEP-0206的主题，而且我们在第2章中曾经接触过。show_traffic()函数忽略这个<body>包装器而处理它的子元素，也就是XMPP节本身。

我们使用一个新的Strophe函数Strophe.serialize()将所有XMPP节从XML转换成文本(这被称为序列化)。Strophe的serialize()函数产生的XML字符串必须首先经过转义之后才能插入到HTML文档中，否则XML元素会被解释为HTML元素。这种转义是通过xml2html()函数完成的，它将特殊字符替换成它们的转义后版本。然后该文本将作为一个<div>元素追加到控制台的内容后面。传入的XMPP节将被赋予CSS类incoming，而输出的XMPP节则被赋予CSS类outgoing。这些都是jQuery的典型用法。

但当新数据进入时，控制台应该滚动到底部，因此该元素的 scrollTop 属性被设为与它的 scrollHeight 属性相同。但如果在向上滚动以阅读顶部的信息时却被移到了底部，那么会让用户感到沮丧。这里的代码在改变滚动位置之前检查控制台当前是否一直处于底部。如果用户正在查看以前的信息，那么当新数据到来时该窗口就不会跳到底部。当用户返回到底部，自动滚动行为将再次启动。

最后，添加一些逻辑以实现 Disconnect 按钮的功能。一旦 Peek 连接，我们就要启用 Disconnect 按钮，而当 Peek 断开连接时，要将该按钮禁用。可以通过绑定 connected 和 disconnected 事件(当 Strophe 报告这些状态时连接回调会引发这些事件)来完成这项工作。将下面的处理程序添加到 peek.js 的末尾。



可从
Wrox.com
下载源代码

```
$(document).bind('connected', function () {
    $('#disconnect_button').removeAttr('disabled');
});

$(document).bind('disconnected', function () {
    $('#disconnect_button').attr('disabled', 'disabled');
});
```

code snippet peek.js

要让该按钮能够执行一些操作，必须处理它的单击事件。将下面的代码添加到文档准备就绪事件处理程序中。



可从
Wrox.com
下载源代码

```
$('#disconnect_button').click(function () {
    Peek.connection.disconnect();
});
```

code snippet peek.js

如果在 Web 浏览器中加载应用程序并登录到自己的 XMPP 服务器，就会看到在身份验证期间发送的 XMPP 节出现在控制台中。唯一的问题是，它们显示为长长的、不带任何格式化的字符串。下面就让我们来美化它们，让它们更加易于阅读。

4.3.3 美化 XML

查看长长文本字符串是一件相当费时费力的苦差事，因此我们希望重新格式化 XML 以产生更加美观、可读性更佳的显示结果。一种典型的做法是缩进。每个子元素都应该在其父元素下面缩进，而它自己的子元素也要进行类似的缩进。属性也可以进行缩进，这样它们就与所有其他属性排成一列，每个属性占据一行。最后，可以为不同的部分指派不同的颜色：标点、标记名称、属性、属性值以及内容。

首先，我们希望添加一些适当的 CSS 样式供转换使用。将下面的样式添加到 peek.css 文件中。



可从
Wrox.com
下载源代码

```
.xml_punc { color: #888; }
.xml_tag { color: #e77; }
.xml_aname { color: #55d; }
.xml_avelue { color: #77f; }
.xml_text { color: #aaa; }
.xml_level0 { padding-left: 0; }
.xml_level1 { padding-left: 1em; }
.xml_level2 { padding-left: 2em; }
.xml_level3 { padding-left: 3em; }
.xml_level4 { padding-left: 4em; }
.xml_level5 { padding-left: 5em; }
.xml_level6 { padding-left: 6em; }
.xml_level7 { padding-left: 7em; }
.xml_level8 { padding-left: 8em; }
.xml_level9 { padding-left: 9em; }
```

code snippet peek.css

每级子元素均缩进1em，最多达9级子元素。表4-1解释了其他的样式。

表4-1 XML样式

CSS类	用途
.xml_punc	像<、>、/、=这样的标点
.xml_tag	元素标记名称
.xml_aname	属性名称
.xml_avelue	属性值
.xml_text	元素的文本子元素

接下来，我们需要修改 `show_traffic()`，使用 `Strophe.serialize()`之外的函数来生成数据表示。在下面的代码中，`Peek.pretty_xml()`已经将突出显示行中的旧的序列化代码替换掉了。



可从
Wrox.com
下载源代码

```
show_traffic: function (body, type) {
    if (body.childNodes.length > 0) {
        var console = $('#console').get(0);
        var at_bottom = console.scrollTop >= console.scrollHeight -
            console.clientHeight;

        $.each(body.childNodes, function () {
            $('#console').append("<div class='" + type + "'>" +
                Peek.pretty_xml(this) +
                "</div>");
        });
    }
}
```

code snippet peek.js

最后，我们需要实现 `pretty_xml()`。程序清单 4-3 中的实现是递归的。首先，它将开头的标记以及它的属性进行样式化，然后该函数针对每个子标记调用自己，最后它将结束标记样式化。还有一些额外的情形(文本子元素和空白标记)需要处理。每行输出都位于自己的`<div>`元素中，而文本子元素则位于自己的、可能占用多行的`<div>`元素中。



可从
Wrox.com
下载源代码

程序清单 4-3 peek.js 中的 `pretty_xml()` 函数

```
pretty_xml: function (xml, level) {
    var i, j;
    var result = [];
    if (!level) {
        level = 0;
    }

    result.push("<div class='xml_level" + level + "'>");
    result.push("<span class='xml_punc'>&lt;"/></span>");
    result.push("<span class='xml_tag'>\"");
    result.push(xml.tagName);
    result.push("</span>");
    // attributes
    var attrs = xml.attributes;
    var attr_lead = []
    for (i = 0; i < xml.tagName.length + 1; i++) {
        attr_lead.push(" ");
    }
    attr_lead = attr_lead.join("");

    for (i = 0; i < attrs.length; i++) {
        result.push(" <span class='xml_aname'>");
        result.push(attrs[i].nodeName);
        result.push("</span><span class='xml_punc'>='</span>");
        result.push("<span class='xml_alue'>");
        result.push(attrs[i].nodeValue);
        result.push("</span><span class='xml_punc'>'</span>");

        if (i !== attrs.length - 1) {
            result.push("</div><div class='xml_level" + level + "'>");
            result.push(attr_lead);
        }
    }

    if (xml.childNodes.length === 0) {
        result.push("<span class='xml_punc'>/&gt;</span></div>");
    } else {
        result.push("<span class='xml_punc'>&gt;</span></div>");

        // children
        $.each(xml.childNodes, function () {
            if (this.nodeType === 1) {
                result.push(Peek.pretty_xml(this, level + 1));
            } else if (this.nodeType === 3) {

```

```

        result.push("<div class='xml_text xml_level" +
                    (level + 1) + "'>");
        result.push(this.nodeValue);
        result.push("</div>");
    }
});

result.push("<div class='xml xml_level" + level + "'>");
result.push("<span class='xml_punc'>&lt;/</span>");
result.push("<span class='xml_tag'>");
result.push(xml.tagName);
result.push("</span>");
result.push("<span class='xml_punc'>&gt;</span></div>");
}
return result.join("");
}

```

读者可能好奇的是，为什么这里的代码将所有的字符串(逐部分地)放入一个数组中，而不是使用字符“+”将它们串接起来。使用数组来存放较长字符串的每个部分，然后将它们一次性连接起来，这是 JavaScript 中一种常见的优化模式。因为 JavaScript 字符串是不可变的，当把两个字符串串接起来时必须创建一个新的字符串。等待所有的小字符串全部创建完毕之后再执行最终的拼接工作，将为解释器省去大量的中间字符串创建工作。类似的字符串拼接优化模式也存在于其他使用不可变字符串的编程语言中，比如 Python 和 Java。

如果在完成这些修改之后再次加载 Peek 应用程序，那么我们将会看到美观的 XML 输出结果，就像图 4-1 中所示的那样。

4.3.4 处理 XML 输入

现在控制台能够显示美观的、彩色的 XML 节，但一旦初始的身份验证和会话建立完成之后，就没有更多的 XML 节可供呈现。现在我们要向 Peek 中添加用户输入，这样就能够与控制台交互。

首先，需要一旦连接准备好接受 XMPP 节就立即让输入域和 Send 按钮变得可用。我们在 `connected` 和 `disconnected` 事件处理程序中对 `Disconnect` 按钮做了上述操作。因为两个按钮都有 `button` 类，所以我们可以同时处理这两个按钮。对于文本域，我们还需要将 `disabled` 类(用来更改背景色)删除。下面给出了新的处理程序，修改部分已被突出显示。



可以从
Wrox.com
下载源代码

```

$(document).bind('connected', function () {
    $('.button').removeAttr('disabled');
    $('#input').removeClass('disabled').removeAttr('disabled');
});
$(document).bind('disconnected', function () {
    $('.button').attr('disabled', 'disabled');
    $('#input').addClass('disabled').attr('disabled', 'disabled');
});

```

code snippet peek.js

只要建立的连接存在, 用户现在就可以在文本域中输入 XML。我们只需要在用户单击 Send 时执行操作即可。

Strophe 的 send() 函数只接受有效 XML DOM 对象或 Strophe.Builder 对象。这使得我们很难通过 XMPP 连接发送无效 XML。发送无效 XML 将导致服务器立即终止连接。但用户只能输入文本, 因此我们必须首先创建一个函数, 把文本分析为 XML。Web 浏览器都内置了 XML 分析器。将下面的 text_to_xml() 函数添加到 Peek 对象中。



可从
Wrox.com
下载源代码

```
text_to_xml: function (text) {
    var doc = null;
    if (window['DOMParser']) {
        var parser = new DOMParser();
        doc = parser.parseFromString(text, 'text/xml');
    } else if (window['ActiveXObject']) {
        var doc = new ActiveXObject("MSXML2.DOMDocument");
        doc.async = false;
        doc.loadXML(text);
    } else {
        throw {
            type: 'PeekError',
            message: 'No DOMParser object found.'
        };
    }

    var elem = doc.documentElement;
    if ($(elem).filter('parsererror').length > 0) {
        return null;
    }
    return elem;
}
```

code snippet peek.js

text_to_xml() 函数创建一个 XML 分析器, 并分析字符串。Internet Explorer 6 浏览器中并没有 DOMParser 类, 因此必须使用 ActiveX 对象。但 Firefox、Safari 和 Opera 浏览器均实现了 DOMParser。ActiveX 对象与 DOMParser API 稍有不同, 但对于 Peek 的需要而言, 所需的修改非常细微。

有些 DOMParser 对象也会为无效输入数据输出 XML 文档, 而且这些错误文档将拥有一个顶级的<parsererror>元素。我们必须检查这一点, 这样就不会意外地将这些错误文档当作 XMPP 节发送出去。

下面就是要把 Send 按钮与 text_to_xml() 函数连接起来并发送分析结果。可以将下面的代码添加到文档准备就绪事件处理程序中来完成这项工作。



可从
Wrox.com
下载源代码

```
$('#send_button').click(function () {
    var xml = Peek.text_to_xml($('#input').val());
    if (xml) {
```

```

Peek.connection.send(xml);
$('#input').val('');
}
});

```

code snippet peek.js

注意，不必把 XML 添加到控制台。Strophe 会自动把 XMPP 节传给 `xmlInput()` 和 `rawInput()` 日志函数，而这些函数已经负责将美观的 XML 数据添加到控制台。

最后我们还需要处理输入错误。目前，如果用户输入一些无效的输入，比如`<<presence>>`，那么单击按钮不会发生任何事情。若是能给用户某种反馈就更好了。JQuery 可以轻松完成这件工作。修改后的 Send 按钮单击事件处理程序会在探测到输入错误时通过动画将背景色渐变为红色。



可从
Wrox.com
下载源代码

```

$('#send_button').click(function () {
    var xml = Peek.text_to_xml($('#input').val());
    if (xml) {
        Peek.connection.send(xml);
        $('#input').val('');
    } else {
        $('#input').animate({backgroundColor: "#faa"}, 200);
    }
});

```

code snippet peek.js

现在，一旦用户开始纠正他的错误，我们还必须将背景色重置。可以使用 `keypress` 事件处理程序来完成这项工作。应该将下面的代码添加到文档准备就绪事件处理程序中。



可从
Wrox.com
下载源代码

```

$('#input').keypress(function () {
    $(this).css({backgroundColor: '#fff'});
});

```

code snippet peek.js

Peek 现在已经是一个可以运行的 XMPP 调试器了。

4.3.5 简化输入

手工输入所有 XML 可能有些无聊。Strophe 对此进行了补偿：我们可以利用易用的 `Builder` 对象以及它的辅助器函数`$msg()`、`$pres()` 和 `$iq()`。很容易扩展 `Peek` 以允许用户输入代码和 XML，如果他们了解一些 JavaScript 知识，那么他们的工作就会变得轻松得多。

首先，必须探测用户输入的是代码还是 XML。最简单的实现方法是查看第一个字符。如果该字符是“`<`”，那么它非常有可能是 XML；而如果该字符是“`$`”，那么它很可能是 `Builder` 对象的三个辅助器函数之一。如果用户输入的可能是代码，那么可以使用 JavaScript 的 `eval()` 函数来执行该代码。可以将 Send 按钮的单击事件处理程序替换成以下的新逻辑。



```

        $('#send_button').click(function () {
            var input = $('#input').val();
            var error = false;
            if (input.length > 0) {
                if (input[0] === '<') {
                    var xml = Peek.text_to_xml(input);
                    if (xml) {
                        Peek.connection.send(xml);
                        $('#input').val('');
                    } else {
                        error = true;
                    }
                } else if (input[0] === '$') {
                    try {
                        var builder = eval(input);
                        Peek.connection.send(builder);
                        $('#input').val('');
                    } catch (e) {
                        error = true;
                    }
                } else {
                    error = true;
                }
            }
            if (error) {
                $('#input').animate({backgroundColor: "#faa"});
            }
        });
    
```

code snippet peek.js

如果第一个字符串是“<”，那么处理逻辑完全相同。但当第一个字符是“\$”时，Peek 将输入作为代码进行计算，而且如果没有错误出现，就试着将该输入作为 XMPP 节发送出去。如果代码抛出一个异常(例如，如果该代码包含一个语法错误或者没有产生 Builder 对象)，就表明出现一个错误。

Peek 现在能够帮助我们深度研究 XMPP。程序清单 4-4 给出了最终的 peek.js 文件。



程序清单 4-4 peek.js(最终版本)

可从
Wrox.com
下载源代码

```

var Peek = {
    connection: null,

    show_traffic: function (body, type) {
        if (body.childNodes.length > 0) {
            var console = $('#console').get(0);
    
```

```

var at_bottom = console.scrollTop >= console.scrollHeight
  - console.clientHeight;

$.each(body.childNodes, function () {
  $('#console').append("<div class='"
    + type + "'>" +
    Peek.pretty_xml(this) +
    "</div>");

});

if (at_bottom) {
  console.scrollTop = console.scrollHeight;
}

},
pretty_xml: function (xml, level) {
  var i, j;
  var result = [];
  if (!level) {
    level = 0;
  }

  result.push("<div class='xml_level" + level + "'>");
  result.push("<span class='xml_punc'>&lt;</span>");
  result.push("<span class='xml_tag'>\"");
  result.push(xml.tagName);
  result.push("</span>");

  // attributes
  var attrs = xml.attributes;
  var attr_lead = []
  for (i = 0; i < xml.tagName.length + 1; i++) {
    attr_lead.push(" ");
  }
  attr_lead = attr_lead.join("");

  for (i = 0; i < attrs.length; i++) {
    result.push(" <span class='xml_aname'>\"");
    result.push(attrs[i].nodeName);
    result.push("</span><span class='xml_punc'>='</span>");
    result.push("<span class='xml_alue'>\"");
    result.push(attrs[i].nodeValue);
    result.push("</span><span class='xml_punc'>'</span>");

    if (i !== attrs.length - 1) {
      result.push("</div><div class='xml_level" + level + "'>");
      result.push(attr_lead);
    }
  }
  if (xml.childNodes.length === 0) {
    result.push("<span class='xml_punc'>/&gt;</span></div>");
  } else {
    result.push("<span class='xml_punc'>&gt;</span></div>");
  }
}

```

```

        // children
        $.each(xml.childNodes, function () {
            if (this.nodeType === 1) {
                result.push(Peek.pretty_xml(this, level + 1));
            } else if (this.nodeType === 3) {
                result.push("<div class='xml_text xml_level" +
                           (level + 1) + "'>");
                result.push(this.nodeValue);
                result.push("</div>");
            }
        });

        result.push("<div class='xml xml_level" + level + "'>");
        result.push("<span class='xml_punc'>&lt;/</span>");
        result.push("<span class='xml_tag'>");
        result.push(xml.tagName);
        result.push("</span>");
        result.push("<span class='xml_punc'>&gt;</span></div>");
    }

    return result.join("");
},
text_to_xml: function (text) {
    var doc = null;
    if (window['DOMParser']) {
        var parser = new DOMParser();
        doc = parser.parseFromString(text, 'text/xml');
    } else if (window['ActiveXObject']) {
        var doc = new ActiveXObject("MSXML2.DOMDocument");
        doc.async = false;
        doc.loadXML(text);
    } else {
        throw {
            type: 'PeekError',
            message: 'No DOMParser object found.'
        };
    }
    var elem = doc.documentElement;
    if ($(elem).filter('parsererror').length > 0) {
        return null;
    }
    return elem;
}
};

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Connect to XMPP',

```

```
buttons: {
    "Connect": function () {
        $(document).trigger('connect', {
            jid: $('#jid').val(),
            password: $('#password').val()
        });
        $('#password').val('');
        $(this).dialog('close');
    }
});
});

$('#disconnect_button').click(function () {
    Peek.connection.disconnect();
});

$('#send_button').click(function () {
    var input = $('#input').val();
    var error = false;
    if (input.length > 0) {
        if (input[0] === '<') {
            var xml = Peek.text_to_xml(input);
            if (xml) {
                Peek.connection.send(xml);
                $('#input').val('');
            } else {
                error = true;
            }
        } else if (input[0] === '$') {
            try {
                var builder = eval(input);
                Peek.connection.send(builder);
                $('#input').val('');
            } catch (e) {
                console.log(e);
                error = true;
            }
        } else {
            error = true;
        }
    }
    if (error) {
        $('#input').animate({backgroundColor: "#faa"});
    }
});

$('#input').keypress(function () {
    $(this).css({backgroundColor: '#fff'});
});
});
```

```

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        "http://bosh.metajack.im:5280/xmpp-httpbind");

    conn.xmlInput = function (body) {
        Peek.show_traffic(body, 'incoming');
    };

    conn.xmlOutput = function (body) {
        Peek.show_traffic(body, 'outgoing');
    };

    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });
    Peek.connection = conn;
});

$(document).bind('connected', function () {
    $('.button').removeAttr('disabled');
    $('#input').removeClass('disabled').removeAttr('disabled');
});

$(document).bind('disconnected', function () {
    $('.button').attr('disabled', 'disabled');
    $('#input').addClass('disabled').attr('disabled', 'disabled');
});

```

4.4 研究 XMPP

Peek 可用来帮助我们研究 XMPP 扩展如何运转以及为何有些扩展并不能按照预期执行操作。我们可以从应用程序中剪切并粘贴 XMPP 节构建代码，看看服务器的响应究竟是什么。如果不熟悉特定的协议扩展，那么可以输入示例，看看服务器如何响应不同的输入。

本书中含有大量的可以在 Peek 中进行试验的示例。一定要适当地调整服务器名称和 JID。本书中的示例使用的是虚构的服务器域和虚构人物的 JID。如果试着使用第 4.4.3 节中的那些虚构的示例，就会看到发生什么情况。

4.4.1 控制出席

在第 1 章中我们学习过，出席信息和出席控制是 XMPP 最基本的功能。出席也是该协议最简单的部分之一。

打开 Peek 应用程序，登录到 XMPP 服务器，输入下面的代码行，然后单击 Send。

```
<presence/>>
```

在XMPP连接上发送的第一个`<presence>`元素被称为初始出席。通常，服务器将您的出席信息广播到您的 JID 的所有已连接资源以及所有订阅出席通知的用户。初始出席还会让服务器向您的花名册中已经订阅了出席通知的所有用户发送出席探测。初始出席使得您能够从自己的联系人那里接收传入的 presence 节。

如果花名册为空，那么在响应中我们只能看到稍微经过修改的 XMPP 节反馈回来。如果花名册中还有其他联系人，那么您将可能几乎立即接收到他们的出席通知。只要服务器认为您在线，您就会持续地接收到联系人出席状态改变时发送过来的出席更新信息。

因为没有在`<presence>`元素上指定任何属性，所以它将通知服务器您在线。试着将下面的内容输入 Peek，将自己的出席状态设为已离开。

```
$pres().c('show').t("away").up().c('status').t("reading");
```

4.4.2 探测版本

大多数 XMPP 客户端(甚至服务器)都支持 Software Version 扩展(XEP-0092)。这个简单的扩展要求实体报告自己的软件和版本号。服务器和 XMPP 服务经常使用这个协议扩展来收集统计信息，而我们可以使用 Peek 来试验如何请求软件版本。

为了请求服务器的软件版本，请发送一个携带有使用 `jabber:iq:version` 命名空间的`<query>`元素的 IQ-get 节。可以将下面两种代码之一输入 Peek 来询问 Jabber.org 服务器它运行的是什么软件。

```
$iq({type: "get", id: "version1", to: "jabber.org"})
  .c("query", {xmlns: "jabber:iq:version"})
```

或

```
<iq type='get' id='version1' to='jabber.org'><query xmlns='jabber:iq:version' /></iq>
```

服务器将响应与下面 XMPP 节类似的信息。因为在您阅读本书的时候 Jabber.org 服务器软件可能已经升级，所以您接收到的响应可能不同。

```
<iq xmlns='jabber:client'
  from='jabber.org'
  to='darcy@pemberley.lit/library'
  id='version1'
  type='result'>
<query xmlns='jabber:iq:version'>
  <name>ejabberd</name>
  <version>2.1.0-alpha</version>
  <os>unix/linux 2.6.18</os>
</query>
```

```
</iq>
```

试着探测自己的一些联系人或者其他服务器，看看他们使用的是什么软件。

4.4.3 处理错误

错误处理是所有应用程序中的重要一环。XMPP 协议有一套统一的错误报告机制，核心协议以及几乎所有扩展都使用了该机制。第 1 章中讲过 IQ-error 节，但我们可以使用 Peek 来更详细地研究这些节以及其他错误条件。

1. iq 节错误

我们发送的 IQ-get 和 IQ-set 节总是应该能够接收到 IQ-result 或 IQ-error 应答。我们应该试着构建一些无效的节来看看不同的实体如何响应。

gmail.com 的 Google Talk 服务不支持前面讨论的 Software Version 扩展。向 gmail.com 发送一个请求来询问它的软件版本。

```
$iq({type: "get", id: "version2", to: "gmail.com"})
  .c("query", {xmlns: "jabber:iq:version"})
```

服务器应该立即响应一个错误提示信息。

```
<iq from='gmail.com'
  to='darcy@pemberley.lit/library'
  id='version1'
  type='error'>
  <query xmlns='jabber:iq:version' />
  <error code='503' type='cancel'>
    <service-unavailable xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
  </error>
</iq>
```

IQ-error 节将原始请求的内容作为第一个子元素包含在其中。我们接收到的许多错误提示节都包含原始请求的内容，但在某些情况下，如果该节非常庞大或包含敏感信息，那么这部分可能会被忽略。<error>元素是必需的，而且它应该包含一个<text>元素以及另一个位于 urn:ietf:params:xml:ns:xmpp-stanzas 命名空间下面的唯一的子元素。前面的错误提示节并没有包含<text>子元素(它是可选的)，但在下一节中我们将会看到这样的一个示例。后一种类型的子元素的 name 属性将指出已发生的错误的类型。在这里，它是 service-unavailable 错误。这正是我们期待的错误提示，因为 gmail.com 并不支持该扩展。

此外还要注意的是，<error>元素的 type 属性是 cancel。这意味着我们不应该继续尝试该操作。有些错误会把 type 属性设为 modify，这意味着应用程序应该用纠正后的输入再次尝试。其他的错误类型也都有可能，它们有 continue、auth 和 wait。可以将下面的 XMPP 节发送到多人聊天服务中的一个不存在的房间，从而产生一个错误。

```
<iq type='get' to='bad-room-123@conference.jabber.org' id='info1'>
  <query xmlns='http://jabber.org/protocol/disco#info' />
</iq>
```

应该会接收到一条类似于下面的内容的响应。注意，服务器已经提供了一个人类可读的<text>元素以及一个普通的错误条件元素。

```
<iq to='darcy@pemberley.lit/library'
  from='bad-room-123@conference.jabber.org'
  id='info1'
  type='error'>
  <query xmlns='http://jabber.org/protocol/disco#info' />
  <error code='404'
    type='cancel'>
    <item-not-found xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
    <text xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'>
      Conference room does not exist
    </text>
  </error>
</iq>
```

<error>元素可能还包含应用程序特定的错误条件，它的命名空间是该服务特有的。可以在 RFC 3920 的第 9.3 节中找到更多有关<error>元素和 IQ-error 节的信息。

2. message 节错误

<message>节也可能导致错误，而且这些错误的结构与 IQ-error 节非常类似。与 IQ-error 节类似，<message>节的错误的 type 属性值为 error，通常包含原始消息而且还包含相同的<error>元素。

最常见的消息错误之一是未能传送给用户。例如，试着向一个虚构的服务器上的一位虚构的用户发送一条消息。

```
$msg({to: 'elizabeth@longbourn.lit', type: 'chat'}).c('body')
  .t('What think you of books?')
```

因为域 longbourn.lit 并不存在，所以服务器将响应一条类似下面的消息错误提示。

```
<message to='darcy@pemberley.lit'
  from='elizabeth@longbourn.lit'
  type='error'>
  <body>What think you of books?</body>
  <error code='404'
    type='cancel'>
    <remote-server-not-found xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
  </error>
</message>
```

这些类型的传送错误提示相当常见，用户经常会输错地址或者具有已经改变服务器的联系人。

3. presence 节错误

与消息错误一样，`<presence>`节错误通常会在远程服务器不可达(可能是因为它们本不存在或网络掉线)时出现。它们偶尔也会突然出现在其他地方。与 IQ-error 和消息错误类似，presence 节错误的 `type` 属性值为 `error`，并含有`<error>`元素。

我们常常会看到它们会作为服务器出席探测的结果而出现。当发送初始出席信息时，XMPP 服务器会向所有联系人发送出席探测，而如果这些联系人的任何一个服务器不能到达，这就会产生一个出席错误并传送给您。下面给出了这种类型错误的一个示例：

```
<presence to='darcy@pemberley.lit/library'
          from='elizabeth@longbourn.lit'
          type='error'>
  <error code='404'
         type='cancel'>
    <remote-server-not-found xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
  </error>
</presence>
```

由于某种原因，pemberley.lit 服务器(Darcy 的 XMPP 账户所在的服务器)不能到达 longbourn.lit 服务器，因此将出席错误提示作为出席探测的结果返回。

每种 XMPP 节都有匹配的错误提示节，它们的工作方式均相同。应用程序必须注意的错误取决于应用程序以及故障的重要性。例如，客户端会忽略这些出席错误提示，这是因为联系人的状态与用户看到的状态相同，不管该联系人位于一个不可达服务器还是已经离线。但在试图加入某个多人聊天室(参见第 8 章)时，应该将收到的出席错误提示指出的真正问题告知用户。

4.5 更好的调试

Peek 已经相当有用，但它可以更好。试着添加一些额外的功能：

- 让控制台中的 XMPP 节可折叠——单击元素使其子元素隐藏或显示。
- 用户常常希望再次运行同一命令，添加对命令历史的支持。

4.6 小结

在本章中，我们构建了一个应用程序来查看协议流量并辅助研究和调试 XMPP 应用程序和服务。在这个过程中我们学习了如下的内容：

- 钩入 Strophe 的结构化日志工具
- 分析用户的 XML 输入
- 将 XML 格式化为美观的 HTML

- 将用户输入作为 JavaScript 代码进行计算

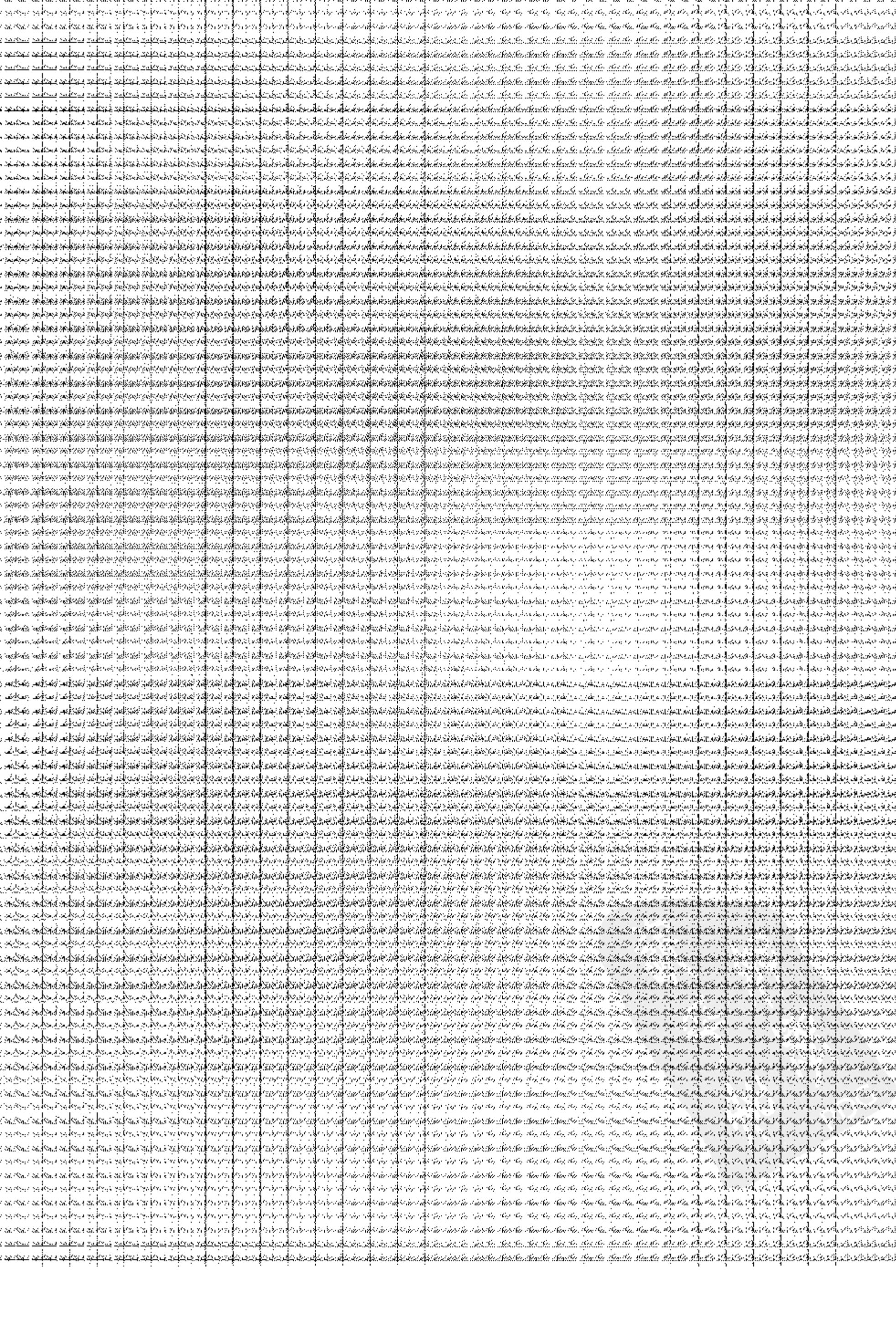
一旦构建 Peek，就可用它来研究 XMPP 协议的一些常见部分：

- 操纵自己的出席信息
- 从其他系统中检索软件版本信息
- 处理各种常见节错误

阅读本书剩余部分，读者会发现 Peek 在诊断问题以及研究 XMPP 协议方面非常有用。

在下一章中，我们将构建一个 Identi.ca 微博服务的通信客户端。





5

第 5 章

实时微博：一个 Identica 客户端

本章内容

- 使用 XMPP <message> 节
- 使用 Identica XMPP API
- 使用 XHTML-IM 来改进消息
- 处理存储的离线消息

成千上万的用户通过 Twitter、Identica、Jaiku 以及其他微博服务与好友和世界各地的朋友进行通信。这些服务要求用户回答简单的问题：“您在做什么？”用户的更新产生的数据流与动态的、全球的聊天室非常类似：每个用户通过他们感兴趣的人来定义他们身处的聊天室。

与聊天系统类似，微博系统通常具有低延迟特点，便于在参与者之间进行实时通信。但这种低延迟通道受到传统的、高延迟用户界面的限制。许多微博服务的高级用户使用能够提供更快体验的第三方客户端与这些系统交互。

Identica 就是这样一种能够完全支持 XMPP 客户端的服务。在本章中，我们将为 Identica 系统构建一个实时的微博客户端。我们将开始了解如何让一个 XMPP 支撑的 Web 应用程序拥有更加动态的、低延迟的实时体验。

在前两章中，我们曾经学习 Strophe 库的基础知识并构建一个便于试验和调试的简单应用程序。在本章中，我们将开始做一些比较有趣的事情，我们将编写使用 XMPP 的有用软件，如果采用其他方法则很难完成同样的工作。

5.1 应用程序预览

Identica 服务中的更新叫做 dent。本章我们构建的应用程序叫做 Arthur(指 Arthur Dent，他是《银河系漫游指南》中的主角)。图 5-1 给出了 Arthur 运行时的情况。

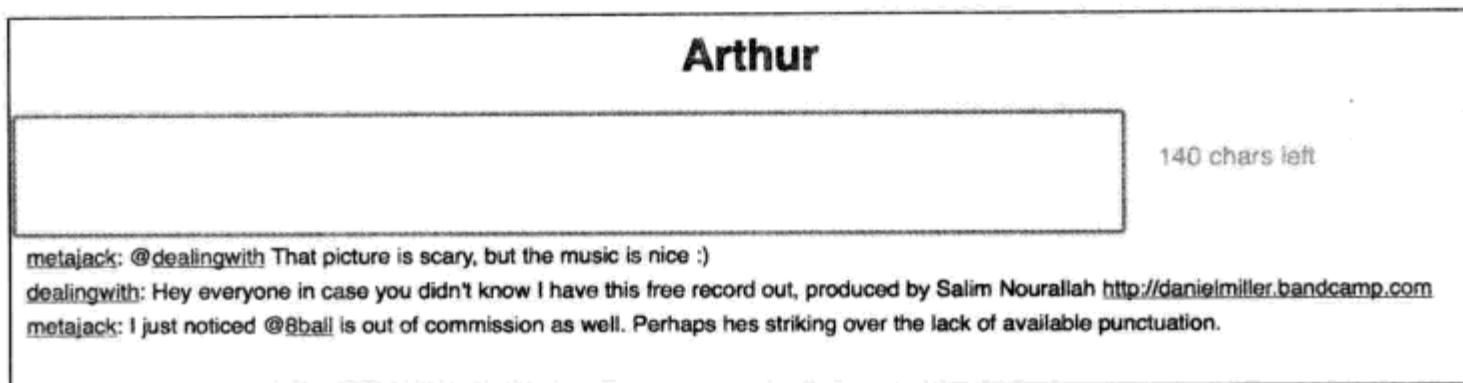


图 5-1

输入域位于应用程序的顶部。用户在这里输入并发送他们的 dent。在输入域下面，按照最近时间顺序显示传入的消息。这些消息通常包含可单击链接，这是因为它们以 XHTML 格式传送。

5.2 Arthur 的设计

Identica 的 XMPP API 使用 XMPP <message> 节来与系统进行交互。用户的 dent 将作为消息发送给 update@identi.ca，而同样的地址会用来发送从被关注用户那里传入的更新(在更新发生时)。这正是 XMPP 中普通的一对一通信的工作方式。

尽管这个 API 对除了新 dent 之外的其他用户动作来说有点不够灵活，但它确实具有天生的优势：任何 XMPP 客户端不经修改都能够使用它。XMPP 客户端均支持一对一聊天，因此能够进行基本的 Identica 集成。

Arthur 模拟了 Web 应用程序中的这种基本的一对一聊天整合。Arthur 的需求要比一个完整的聊天客户端更加简单，这是因为它只需要支持与单个实体的聊天即可。因为另一端是一个计算机程序，而不是一个人类。Arthur 不必支持像输入通知或出席更新这样的功能。另一方面，由于应用程序的关注点比较集中，最终的用户体验应该会更好一点，这是因为它是专门为微博而设计的。

Arthur 需要处理来自 Identica 系统的传入消息，并适当地显示出来。这些消息也携带 XHTML 标记，可用于显著地改进它们的显示效果。同时还需要将来自用户的新 dent 发送给 Identica。最后，Arthur 必须处理用户离线时发送的消息。

5.3 Identica 微博

Identica 虽然只是众多微博服务中的一个，但它有一些重要的属性使得它成为最理想的处理对象，这不仅仅是因为它对 XMPP 的支持。虽然 Twitter 是这些服务中最知名的，但它早期对 XMPP 的支持实验已经中止，因为它主要工作转向了让网站保持可用。Twitter 也是一个“用高墙围起来的花园”，也就是说它并不能与其他微博服务进行互操作。

Identica 是构建在 StatusNet 微博框架之上的最知名的系统。StatusNet 是一个开放源代码、联合的、任何人都能够运行的微博平台。实际上，StatusNet 不仅仅是开放源代码，它还是一项 Free Network Service(见 <http://autonomo.us/2008/07/franklinstreet-statement/>)，它是一项构建在自

由软件之上的服务，将更多重点放在了让用户控制自己的数据上。

与 Twitter 和其他封闭的微博服务不同，StatusNet 系统形成了一个能够彼此进行通信的微博网络。这种联合非常类似于第 1 章中讨论的 XMPP 网络或 Internet 电子邮件服务器网络。任何 StatusNet 站点都能够使用 OMB(Open Microblogging, 开放微博)协议从其他微博站点那里接收和发送更新。

OMB

OMB 是 Evan Prodromou 为了建立一个开放的、联合的微博世界的需要而创建的协议。Evan 同时领导着 StatusNet 项目，它是实现 OMB 协议的最著名的软件。

OMB 目前版本是 0.1，可以在下面的网页中找到它：

<http://openmicroblogging.org/protocol/0.1/>

OMB 目前只定义在 HTTP 传输之上，但 OMB 社区中的许多人正希望在 OMB 下一个版本中添加 XMPP 传输。如果有兴趣参与进来或阅读更多有关 OMB 的内容，那么可以加入下面的邮件列表或关注 OMB 博客：

<http://lists.openmicroblogging.org/mailman/listinfo/omb>

<http://openmicroblogging.org/>

OMB 和 StatusNet 项目的思想与 Internet 的思想匹配得很好。如此多 StatusNet 站点(包括 Identica)均很好地支持了 XMPP，这只是一个额外的结果。XMPP 支持也使得我们很容易在 StatusNet 服务器之上构建实时应用程序。

5.3.1 建立账户

如果尚未取得 Identica 账户，也没有拥有其他启用 XMPP 的 StatusNet 服务的账户，那么现在就应该建立一个账户，这是因为我们需要它来测试 Arthur。只需要访问 Identica 的网站 <http://identi.ca>(或者其他启用 XMPP 的 StatusNet 系统)并注册一个账户即可。

如果您希望自己的新账户关注一个人的话，那么可以关注我的账户 <http://identi.ca/metajack>。一旦建立了自己的账户，就需要进行设置以获取 XMPP 通知。

5.3.2 开启 XMPP

要配置 Identica 账户的 XMPP 支持，需要遵循以下步骤：

- (1) 在任何 Identica 页面上，单击页面右上角的导航条中的 Connect 链接。
- (2) 选择 IM 标签页。
- (3) 在 IM Address 域中，输入自己的 Jabber ID 并单击 Add。
- (4) 确认已选择 Send Me Notices through Jabber/Gtalk(“通过 Jabber/Gtalk 向我发送通知”)选项。您可能会希望选择选项 Send Me Replies through Jabber/GTalk from People I'm Not Subscribed To(“将来自我尚未订阅的人的应答通过 Jabber/GTalk 发送给我”)。

(5) 单击 Save 按钮。

(6) Identica 将向您的 XMPP 账户发送一条确认消息。单击它提供的超链接就可以完成设置。

现在，每当您关注的人贴出一则更新消息，您的 XMPP 账户就应该接收到消息。

可以向 8ball 发送一条更新信息来测试上述功能是否可以运行。8ball 是一个 Identica 机器人，它会向所提问题随机地给出 yes 或 no 响应，就像著名的魔法 8 号球玩具。可向 8ball 发送一条类似“@8ball Is it working?”的消息。记住，如果没有选择 replies from people you aren't subscribed to (“来自尚未订阅的人的应答”) 选项，那么需要首先访问 <http://identi.ca/8ball> 并单击 Subscribe 按钮来订阅 8ball。

如果一切顺序，那么 8ball 的随机响应将出现在聊天客户端中，我们就准备好开始构建 Arthur 了。

5.4 构建 Arthur

Identica 已经开始向您的 XMPP 账户发送通知。这些通知可能出现在您最喜欢的聊天客户端中，看上去非常像来自于普通账户的普通个人聊天消息。我们的首个目标是编写足够多的代码让 Arthur 能够连接到 XMPP 账户、从 Identica 那里接收更新，并将它们显示出来。将这些实现之后，就可以花一些精力使用 XHTML-IM 扩展来美化传入的消息。最后，我们还需要让用户将自己的更新发送到 Identica。

5.4.1 开始

这里所需的 HTML 和 CSS 代码遵循着与前两章相同的模式。程序清单 5-1 给出了 HTML 代码，而程序清单 5-2 给出了 CSS 代码。



程序清单 5-1 arthur.html

可从
Wrox.com
下载源代码

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

<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=UTF-8">
    <title>Arthur - Chapter 5</title>

    <link rel='stylesheet' href='http://ajax.googleapis.com/ajax/libs/
    queryui/1.7.2/themes/cupertino/jquery-ui.css'>
    <script src='http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/
    jquery.js'></script>
    <script src='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/
    jquery-ui.js'></script>
    <script src='scripts/strophe.js'></script>
    <script src='scripts/flXHR.js'></script>
    <script src='scripts/strophe.flxhr.js'></script>

    <link rel='stylesheet' type='text/css' href='arthur.css'>
    <script type='text/javascript' src='arthur.js'></script>
```

```

</head>
<body>
    <h1>Arthur</h1>

    <textarea id='input' rows='3'></textarea>
    <div id='counter'><span class='count'>140</span> chars left</div>
    <div id='stream'>
    </div>

    <!-- login dialog -->
    <div id='login_dialog' class='hidden'>
        <label>JID:</label><input type='text' id='jid'>
        <label>Password:</label><input type='password' id='password'>
    </div>
</body>
</html>

```



程序清单 5-2 arthur.css

可从
Wrox.com
下载源代码

```

body {
    font-family: Helvetica;
}

h1 {
    text-align: center;
}

#input {
    width: 75%;
    font-size: 16pt;
}

#counter {
    width: 20%;
    float: right;
    padding: 25px;
    font-size: 14pt;
    color: #bbb;
}

.hidden {
    display: none;
}

```

这个 UI 由一个用于输入的文本区域和一个字符计数器组成。与 Twitter 一样，Identica 将更新信息的字符数限制为 140 个字符。我们希望让用户知道它们还可输入多少个字符，这样他们就不会试图发送过多字符。在输入域下面是一个

元素，它将用来显示传入的更新信息。隐藏的登录对话框被放在了最后。

与前几章类似，我们将需要启用登录对话框并设置连接流事件。将下面的代码添加到新文

件 arthur.js 中：



```

var Arthur = {
    connection: null
};

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Connect to XMPP',
        buttons: {
            "Connect": function () {
                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val()
                });

                $('#password').val('');
                $(this).dialog('close');
            }
        }
    });
});

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        'http://bosh.metajack.im:5280/xmpp-httpbind');
    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });
    Arthur.connection = conn;
});

$(document).bind('connected', function () {
    // nothing here yet
});

$(document).bind('disconnected', function () {
    // nothing here yet
});

```

code snippet arthur.js

Arthur 用户现在应该能够登录到自己的 XMPP 账户。

5.4.2 接收消息

在第 1 章中我们学到 XMPP 会根据每个资源的可访问性以及优先级将<message>节传送到一个或多个资源。在 Arthur 能够从服务器那里接收到任何消息之前，它必须首先发送一个初始<presence>节把自己标记为可访问。一旦可访问，该资源就开始接受发送到该用户裸 JID(假设它具有最高优先级或刚好是唯一的已连接资源)或 Arthur 的连接的完整 JID 的消息。

Arthur 只对<message>节感兴趣，因此我们将为这些 XMPP 节建立一个处理程序，这样它们就会被传送给合适的函数用于显示。要注意在发送初始出席之前建立该处理程序，否则可能会出现竞争条件：在发送初始出席信息与建立处理程序这段时间的消息将会被忽略。

在下面的 `connected` 事件处理程序中新增加的代码行用于建立一个消息处理程序，发送初始出席，并将传入的消息定向到 `handle_message()` 函数，稍后我们将实现这个函数。



```
$(document).bind('connected', function () {
    Arthur.connection.addHandler(Arthur.handle_message,
        null, "message", "chat");
    Arthur.connection.send($pres());
});
```

code snippet arthur.js

建立该处理程序来搜索携带 `type` 属性值 `chat` 的<message>节。我们在第 1 章中曾经学过，在 XMPP 中，`chat` 类型用于普通的个人消息。

现在必须实现 `handle_message()` 函数。这个函数需要将消息正文提取出来并将其插入到用来显示更新信息的<div>元素的顶部。jQuery 使得这项工作变得非常容易。注意检查消息的发送者是否正是您所认为的人，否则您可能会看到错误的消息。



```
handle_message: function (message) {
    if ($(message).attr('from').match(/^update@identi.ca/)) {
        var body = $(message).children('body').text();
        $('#stream').prepend("<div>" + body + "</div>");
    }

    return true;
}
```

code snippet arthur.js

该代码使用 `children()` 只查找<body>元素(它是<message>节的直接子元素)。稍后就会看到，Identica 消息有一个额外的<body>元素，因此这里的代码需要显式地找到正确的元素。

Arthur 现在应该可以接收更新并将更新显示出来。

5.5 XHTML-IM

没过多久 XMPP 社区就开始厌倦了普通文本消息。于是建立了 XHTML-IM(XEP-0071)扩展来重用 XHTML 现有的格式化和行内 CSS 样式以便让消息更加结构化同时在视觉上更加美观。许多 XMPP 客户端均支持发送和接收 XHTML-IM 消息，而像 Identica 这样的 XMPP 服务也经常使用 XHTML-IM 向消息内容中增加结构化信息。

XHTML-IM 是为适于小型聊天消息用例而设计的 XHTML 缩减版。嵌入的对象、脚本、样式表以及内容转换功能均被删除，只留下一些基本的标记和行内样式。下面是一个使用 XHTML-IM 的`<message>`节示例。注意，这里同时提供了普通的`<body>`元素和 XHTML-IM 命名空间下面的`<html>`元素，这样一来如果不支持 XHTML-IM 也可正常工作。

```

<message from='bingley@netherfield.lit'
          to='jane@longbourn.lit'
          type='chat'>
  <body>I hope my dear friends will join me at Netherfield for a ball
  this Tuesday.</body>
  <html xmlns='http://jabber.org/protocol/xhtml-im'>
    <p style='font-weight: bold'>I hope my dear friends will join me at Netherfield
    for a ball this Tuesday.</p>
  </html>
</message>

```

XEP

XMPP 的定义文档是 IETF RFC(Request for Comment, 请求评议)3920 和 3921。这些文档涵盖了 XMPP 流的基本语义、XMPP 联合的工作原理以及主要 XMPP 节类型的细节。所有未被视为 XMPP 核心但被接受作为普遍有价值的补充的部分均以 XEP(XMPP Extension Proposal, XMPP 扩展提议)的形式定义。

任何人都可以提议新的 XEP，将其提交给 XSF(XMPP Standards Foundation, XMPP 标准基金会)。XSF 委员会评审所有新提议的 XEP，并投票决定是否接受它们进入标准化进程。一旦被接受，XEP 就会被分配一个编号并且状态被设为“试验性质”(experimental)。经过一段时间，当人们精炼并实现该 XEP 后，它的状态可能会变为“草案”，而且可能最终成为“定稿”。处于草案或定稿状态的 XEP 被认为是经过严格评审并已准备好广泛实现和部署。在某些情况下，当 XEP 被取代或未能普遍使用时，它们会变成“过时的”或“历史的”。

有些 XEP 非常小，例如 Software Version(XEP-0092)，还有一些则相当大，就像 Publish-Subscribe(XEP-0060，我们将在第 9 章详细讨论)。在 XMPP 出现后的最初十年中接受了将近三百个 XEP，它们覆盖的功能范围非常广泛：从远程命令到发布歌曲信息。每年 4 月 1 日，XSF 甚至会发布一则诙谐的 XEP(我个人比较喜欢 XEP-0239)。

本书中涵了几个重要的 XEP，我们可以在下面找到 XEP 的完整列表以及它们的规范。

<http://xmpp.org/extensions/>

5.5.1 将 XHTML-IM 添加到 Arthur

如果使用 Peek(我们在第4章开发的调试控制台)并查看 Identica 传入的消息，就会看到该服务在每条消息中追加了 XHTML-IM 负载。这些结构化的负载把用户名、标签、组和回复引用转换成可单击的超链接。如果用户在 Arthur 中看到一条更新信息引用他们从来没有听说过的人，那么他们可以轻易地单击超链接以找出更多信息。

因为 Arthur 是一个 Web 应用程序，所以很容易呈现 XHTML-IM 消息。我们可以使用 jQuery 的 `contents()` 方法来获取所有的 XHTML 子元素，然后将它们插入到合适的地方。由于应用程序使用 HTML DOM 而 XMPP 连接使用 XML DOM，因此必须使用 `importNode()` 在文档之间传送元素。然而，IE 并不支持 `importNode()`，因此对于这款浏览器，我们必须使用 `xml` 属性来获取原始的文本信息。在修改后的 `handle_message()` 函数中，这些改变之处都已被突出显示。



可从
Wrox.com
下载源代码

```
handle_message: function (message) {
    if ($(message).attr('from').match(/^update@identi.ca/)) {
        var body = $(message).find('html > body').contents();
        var div = $("<div></div>");

        body.each(function () {
            if (document.importNode) {
                $(document.importNode(this, true)).appendTo(div);
            } else {
                // IE workaround
                div.append(this.xml);
            }
        });
        div.prependTo("#stream");
    }
    return true;
}
```

code snippet arthur.js

请注意 Arthur 并没有核查它接收到的 XHTML。有可能将`<script>`元素和其他有害代码注入到消息正文中。如果是在真实应用程序中，就一定要将这种可能性考虑进去。

Arthur 现在应该显示经过格式化的美观的 HTML 消息而不是普通文本。剩下的工作就是让用户将自己的更新发回到 Identica。

5.5.2 发送消息

读者可能已经预料到，向 Identica 发送消息是通过普通 XMPP 消息完成的。不必担心发送 XHTML-IM 内容或处理消息中的超链接，这是因为 Identica 系统将翻译用户的输入并链接所有的合适项之后才会将其传送给订阅者。在发送消息之前，应该创建字符计数器，这样 Arthur 就能够在 Identica 拒绝用户输入的过长消息之前给他们一些警告。

在 JavaScript 中统计字符数算是小事一桩, 因为 String 对象支持 length 属性, 就像数组一样。唯一的难点是测量文本的长度然后更新计数器。

每当用户在输入域或文本框中输入时, 浏览器就会依次触发 keydown、keypress 和 keyup 事件。麻烦之处在于, keypress 事件(这是放入字符计数逻辑的最显而易见的地方)是在输入域的值修改之前触发的。当触发第一个 keypress 事件时, 输入域的值仍然是空的。但在调用 keyup 事件之前, 这个域的值已经被修改过了, 因此可以将适当的逻辑放在这里来更新计数器。

可以将下面的 keyup 事件处理程序添加到文档准备就绪事件处理函数中, 就放在登录对话框初始化代码之后。



```
$('#input').keyup(function () {
    var left = 140 - $(this).val().length;
    $('#counter .count').text('' + left);
});
```

code snippet arthur.js

现在当用户进行输入操作时 Arthur 就会更新输入计数。

一旦用户对自己编写的更新信息感到满意, 他就可以按下 Enter 键来提交相关信息。为了将这些事件发送到 Identica, 我们只需要将它们包装到一个发送到 update@identi.ca 的普通的<message>节中。这次, 我们希望使用 keypress 事件, 这是因为 Enter 键并不属于消息。

将下面的代码添加到文档准备就绪事件处理程序中。



```
$('#input').keypress(function (ev) {
    if (ev.which === 13) {
        ev.preventDefault();

        var text = $(this).val();
        $(this).val('');
        var msg = $msg({to: 'update@identi.ca', type: 'chat'})
            .c('body').t(text);
        Arthur.connection.send(msg);
    }
});
```

code snippet arthur.js

传入处理程序的事件对象包含一个 which 属性, 它存放着被按下的按键的 ASCII 码, 而 Enter 键的代码为 13。处理程序使用 preventDefault() 来阻止默认处理程序处理 Enter 键(将该字符添加到该域的值中)。然后, 处理程序向 update@identi.ca 发送一个<message>节。

不必担心如何向用户显示更新。Identica 也会把这些用户自己的更新通知给他们, 因此这条更新消息的超链接版本将自动出现在数据流中。

Arthur 现在已经是一个全功能的微博客户端了。

5.6 离线消息

在将 Arthur 交付给用户之前还需要添加最后一项功能，即支持离线消息。

自从 XMPP 早期(当时它还叫做 Jabber)，服务器已经将发送给离线用户的消息保存起来留作以后传送。这个功能(尽管简单)仍然没有在一些专有网络上得以实现。奇怪的是，这项功能在 XMPP 创建后三年之内都没有在 XEP 中指定。延迟传送最后编写成 XEP-0091，之后又被替换成目前的 Delayed Delivery(XEP-0203)。

消息发送者不需要做任何特殊处理就能够支持离线消息。如果接收到一位离线用户的消息，那么一台典型的服务器会将该消息排队，直到该用户下次上线。接收者也只需稍作处理。当离线消息的接收者下次发送初始出席，通知服务器她现在可以访问了。接下来，服务器将几条消息传送给她，但最后一条消息是在她上线之前发出的。

看看下面的示例。首先，Jane 发送她的初始出席，通知服务器她现在可以访问了。接下来，服务器将几条消息传给她，但最后一条消息是在她上线之前发出的。

```

JANE: <presence/>

SERVER: <message from='elizabeth@longbourn.lit/bedroom'
         to='jane@longbourn.lit/bedroom'
         type='chat'>
    <body>Miss Bingley sees that her brother is in love with you, and
         wants him to marry Miss Darcy.</body>
    <delay xmlns='urn:xmpp:delay'
          from='longbourn.lit'
          stamp='1809-09-22T12:44:13Z'>Offline storage</delay>
</message>

<message from='elizabeth@longbourn.lit/bedroom'
         to='jane@longbourn.lit/bedroom'
         type='chat'>
    <body>She follows him to town in the hope of keeping him there, and tries
         to persuade you that he does not care about you.</body>
    <delay xmlns='urn:xmpp:delay'
          from='longbourn.lit'
          stamp='1809-09-22T12:44:24Z'>Offline storage</delay>
</message>

<message from='elizabeth@longbourn.lit/bedroom'
         to='jane@longbourn.lit/bedroom'
         type='chat'>
    <body>No one who has ever seen you together, can doubt his
         affection.</body>
</message>

```

请注意在 Jane 离线期间接收到的那些消息中有一个额外的<delay>元素。这些额外的元素

就是延迟传送标记，它指出该消息是在最初接收到它们后的一段时间之后才传送的。stamp 属性指出 Jane 的服务器最初接收到该消息的时间。

可以通过在用户界面中指派特殊的颜色来指出哪些消息是在用户离线期间出现在数据流中的。在 `handle_message()` 中，我们可以通过在处理 `<message>` 节时为那些包含 `<delay>` 元素的消息指派一个额外的 CSS 类来实现。

首先，向 `arthur.css` 中添加如下 CSS 类。



可从
Wrox.com
下载源代码

```
.delayed {
    color: #a99;
}
```

code snippet arthur.css

接下来，下面被突出显示的代码行给出了需要对 `handle_message()` 函数做出的修改。



可从
Wrox.com
下载源代码

```
handle_message: function (message) {
    if ($(message).attr('from').match(/^update@identi.ca/)) {
        var delayed = $(message).find('delay').length > 0;
        var body = $(message).find('html > body').contents();

        var div = $("<div></div>");
        if (delayed) {
            div.addClass('delayed');
        }

        body.each(function () {
            if (document.importNode) {
                $(document.importNode(this, true)).appendTo(div);
            } else {
                // IE workaround
                div.append(this.xml);
            }
        });
        div.prependTo('#stream');
    }

    return true;
}
```

code snippet arthur.js

Arthur 将用粉红色来显示延迟消息，这样用户就会知道在他们离线期间错过了什么消息。

在添加了最后这项功能之后，就可以将 Arthur 交付给真正的用户了。程序清单 5-3 给出了最终的 JavaScript 代码。



程序清单 5-3 arthur.js

可从
Wrox.com
下载源代码

```

var Arthur = {
    connection: null,

    handle_message: function (message) {
        if ($(message).attr('from').match(/^update@identi.ca/)) {
            var delayed = $(message).find('delay').length > 0;
            var body = $(message).find('html > body').contents();

            var div = $("<div></div>");

            if (delayed) {
                div.addClass('delayed');
            }

            body.each(function () {
                if (document.importNode) {
                    $(document.importNode(this, true)).appendTo(div);
                } else {
                    // IE workaround
                    div.append(this.xml);
                }
            });
            div.prependTo('#stream');
        }
        return true;
    }
};

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Connect to XMPP',
        buttons: {
            "Connect": function () {
                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val()
                });

                $('#password').val('');
                $(this).dialog('close');
            }
        }
    });

    $('#input').keyup(function () {
        var left = 140 - $(this).val().length;
    });
});

```

```

        $('#counter .count').text('' + left);
    });

    $('#input').keypress(function (ev) {
        if (ev.which === 13) {
            ev.preventDefault();

            var text = $(this).val();
            $(this).val('');

            var msg = $msg({to: 'update@identi.ca', type: 'chat'})
                .c('body').t(text);
            Arthur.connection.send(msg);
        }
    });
});

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        'http://bosh.metajack.im:5280/xmpp-httpbind');

    conn.connect(data.jid, data.password, function (status)
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    );
    Arthur.connection = conn;
});

$(document).bind('connected', function () {
    Arthur.connection.addHandler(Arthur.handle_message,
        null, "message", "chat");
    Arthur.connection.send($pres());
});

$(document).bind('disconnected', function () {
    // nothing here yet
});

```

5.7 创建更好的微博客户端

Identica 的 XMPP API 还支持许多其他功能，包括直接消息、收藏、用户订阅与退订、查看个人简介信息。可以发送消息“help”来获取完整的可用命令列表。

试着将以下功能添加到 Arthur 中：

- 显示用户的当前订阅列表。
- 使用用户个人简介将它的用户名转换成他的真实姓名。

- 在每个 dent 旁边添加一个按钮，可让用户将其标记为收藏。

5.8 小结

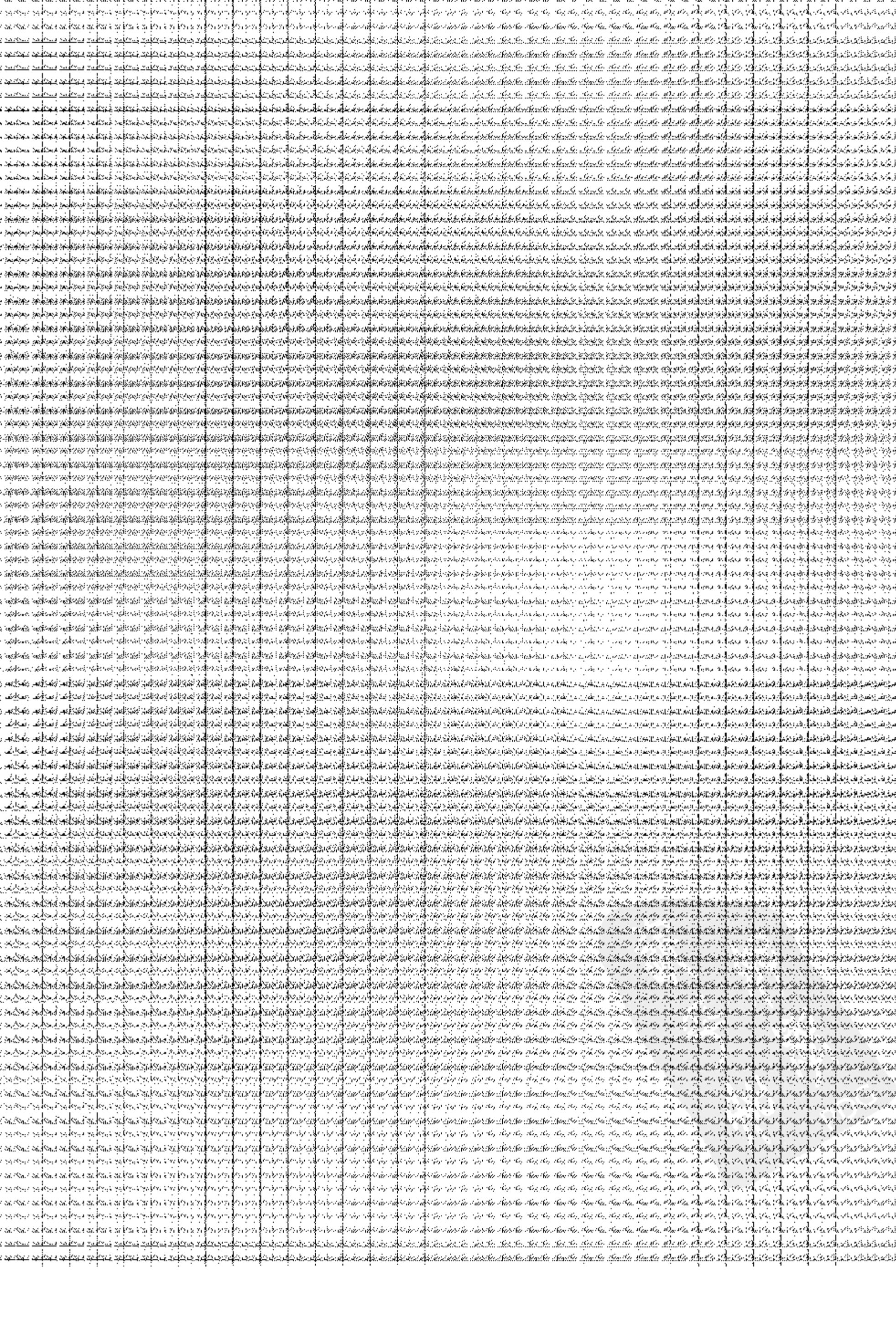
在本章中，我们不再局限于 Strophe 应用程序的基本设置以及使用调试器简单地查看和加工数据流。我们创建了一个实时的微博客户端 Arthur，它可让我们非常方便地与 Identica 上的好友保持联系。

在创建 Arthur 的过程中，我们完成了如下工作：

- 学习 Identica、StatusNet 和 OMB 标准。
- 使用 Identica XMPP API 与微博服务交互。
- 创建一个<message>节处理程序来处理传入的 dent。
- 向应用程序中添加对 XHTML-IM 的支持。
- 发现离线消息以及如何探测和处理它们。

Arthur 是一个使用 XMPP 处理个人的、一对一聊天的非常简单的示例。在下一章中，我们将扩展这个思想，使其变成一个关注于与多人进行个人通信的通用聊天客户端。在第 8 章中我们将创建更加复杂的可进行群聊的应用程序。





第 6 章

与好友交谈：一对一聊天

本章内容

- 出席订阅
- 管理花名册和联系人
- 消息路由
- 一对一通信最佳实践
- 聊天状态通知

很多年以来，XMPP 主要用于它最初的用途，也就是即时通信。该协议的其他用途(在后面几章中我们将会看到)曾经只是少数几个富有想象力的黑客的实验内容。现如今，虽然即时通信部分不像发布-订阅、群聊以及协作应用程序那么流行，但是 XMPP 的 IM 基础仍然是极其重要的。

即时通信系统是一等社交网络。每个成员都有一个社交图(各自的花名册)并能够与其他成员通信和分享。这些社交工具在底层均采用 XMPP 协议，它们使得构建社交应用程序变得相当容易。与许多流行的社交网络不同，XMPP 网络还是联合的，可以将许多不同的社区连接在一起。

应用程序的社交和社区方面正在变得越来越重要，因此理解 XMPP 的基本社交工具势在必行。本章中的聊天应用程序名为 Gab，虽然它可能并不是最先进的软件，但可以使用它的组成部分向任何类型的应用程序流加重要的社交功能。事实上，当许多用户已经越来越习惯于这些功能之后，他们就会希望在其他工具中也能够实现这些功能。

即时通信是 XMPP 最早的用途，而在本章中我们将自行构建一个可用来与好友聊天的、具有多标签页功能的基本的通信应用程序。如果用户以前使用过即时通信应用程序，那么对每项功能似乎都会比较熟悉，它们构成了开发社交软件的坚实基础。

6.1 应用程序预览

本章的应用程序的界面要比我们在前几章中开发的稍微复杂一些。图 6-1 给出了构建完毕的应用程序的外观。

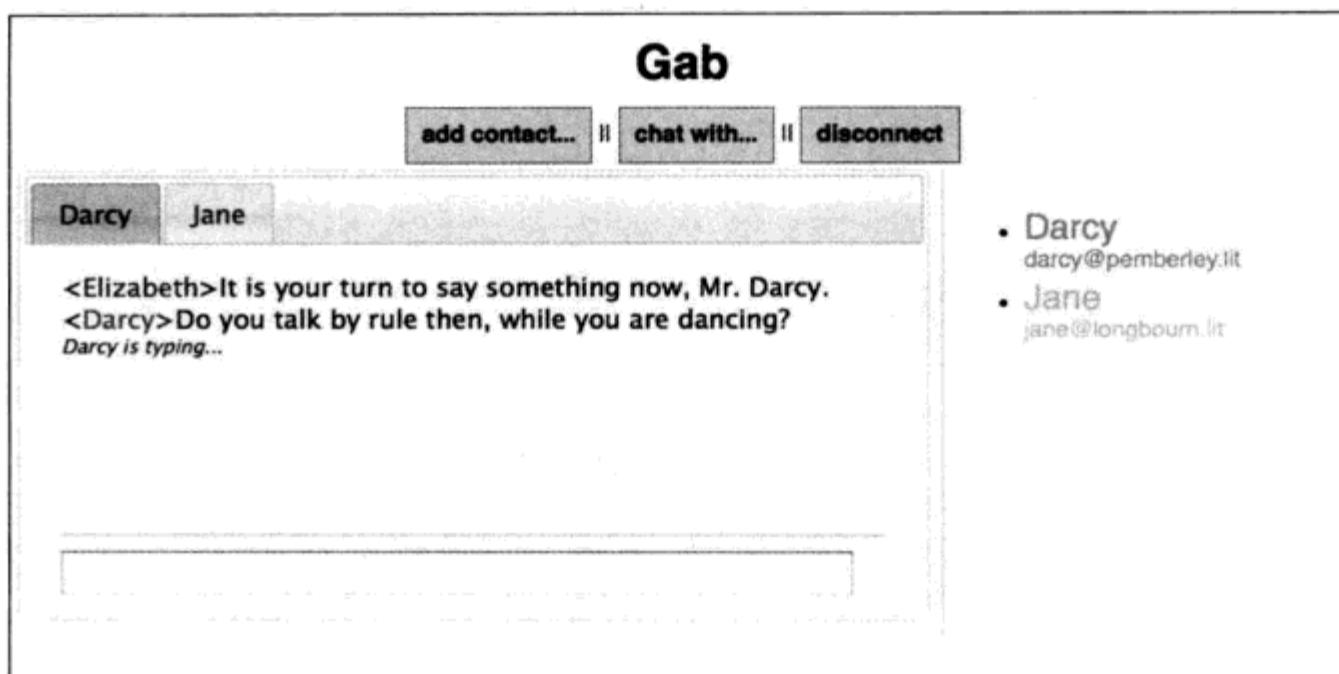


图 6-1

聊天区域位于左侧。每个聊天对话均出现在界面中的一个独立标签页中，从而让用户能够同时保持多个对话。花名册出现在右侧，它显示用户所有联系人的名字和状态。在这些区域上面是几项常见功能的按钮。

这个界面涉及大量工作——多个标签页、多个控件以及 XMPP 协议的多个新内容。

6.2 Gab 的设计

XMPP 即时通信大量使用了<message>和<presence>节。每当一个用户与另一个用户通信时都要发送<message>节。我们在第 5 章中已经看到这些 XMPP 节在实际使用时的简单示例，Identica API 只是一个模拟用户的计算机程序。每当联系人上线、状态变为离开或离线时，都会发送<presence>节。

这两种 XMPP 节休戚相关。<presence>节通告用户的聊天可访问性。如果用户离线或离开，那么可能并不是尝试进行对话的最佳时间。而在线用户很可能愿意立刻与他人进行交谈。一旦用户决定通信，<message>节就负责在对话参与者之间传递对话片段。

6.2.1 出席

XMPP 的设计者对隐私问题非常敏感，因此出席信息是通过订阅进行控制的。为了让 Elizabeth 接收到来自 Wickham 的出席更新信息，她必须首先订阅这些更新信息。此外，Wickham 必须批准她的订阅请求。

出席订阅是非对称的。如果 Elizabeth 订阅了 Wickham 的出席更新信息，那么并不一定意味着 Wickham 也订阅了 Elizabeth 的出席更新信息。在大多数情况下，一位用户向某个人发送订阅请求，他就会自动批准来自同一个人的订阅请求。

在构建 Gab 的花名册区域的功能时，我们就会看到出席订阅功能。

6.2.2 消息

在第5章中我们看过，消息通常非常简单。但它们确实具有一些特殊的传送语义，作为 XMPP 程序员必须加以考虑。这些语义集中在如何寻址以及将一条消息寻址到联系人的裸 JID(elizabeth@longbourn.lit)还是完整 JID(elizabeth@longbourn.lit/library)。

消息还可以包含消息文本之外的内容。它们可以携带格式化信息(就像在第 5 章中使用的 XHTML-IM)、元数据或应用程序特有的有效载荷。稍后就会看到，活动通知有时候会出现在没有包含任何消息文本的消息中。

Gab 有两个主要的功能中心，即聊天区域和花名册区域，以及一个用于实现常见功能的按钮条。每个区域用于处理一种类型的 XMPP 节。聊天区域处理<message>节，而花名册区域处理<presence>节。

6.2.3 聊天区域

Gab 必须支持与多个联系人同时聊天。每个聊天对话拥有一个标签页，这在所有对话之间建立了一种隔离，同时使得用户很容易找到他想要交谈的联系人。

每一条传入的消息都携带有发送该消息的人的地址。每个标签页分别负责处理一个特定发送者传入的消息。每个标签页的底部有一个输入域可用来键入输出的消息。

当从一个尚未打开标签页的联系人那里接收到一条新消息时，就会创建一个新的聊天标签页。还可以通过单击聊天区域上方的动作条中的 **New Chat With** 按钮或在花名册区域单击联系人的名字来建立标签页。

每个聊天对话需要显示消息、状态变化以及相关联用户的活动通知。

6.2.4 花名册区域

花名册区域用于显示用户的所有联系人以及他们当前的出席状态。这个列表应该被排序，这样可访问的联系人就会出现在列表顶部，然后才是那些处理离开状态的联系人。而离线联系人则位于列表的底部。

传入的出席信息应该更新花名册区域，这样用户总是能够知道每个联系人的最新状态。

在动作条中单击 **New Contact** 按钮，这会向花名册中添加一个新的联系人并请求出席订阅。还必须处理一种情况：当新联系人请求订阅用户的出席更新信息时，该用户能够决定是批准还是拒绝订阅请求。

在阐明基本设计之后我们开始编写 **Gab** 应用程序。

6.3 制作界面

程序清单 6-1 给出了 **Gab** 界面的 HTML 代码。它包含动作条以及一个 **chat-area** <div>元素和一个 **roster-area** <div>元素。在它们下面是 **Gab** 需要用到的几个对话框，其中的登录对话框是我们在前几章中已经看到过的，联系人对话框用来向花名册中添加新联系人，而聊天对话框用来启动新的聊天对话，批准对话框用来处理传入的出席订阅请求。



程序清单 6-1 gab.html

可从
Wrox.com
下载源代码

```

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

<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=UTF-8">
    <title>Gab - Chapter 6</title>

    <link rel='stylesheet' href='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/themes/cupertino/jquery-ui.css'>
    <script src='http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/jquery.js'>
    </script>
    <script src='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/jquery-ui.js'></script>
    <script src='scripts/strophe.js'></script>
    <script src='scripts/flXHR.js'></script>
    <script src='scripts/strophe.flxhr.js'></script>

    <link rel='stylesheet' type='text/css' href='gab.css'>
    <script type='text/javascript' src='gab.js'></script>
  </head>
  <body>
    <h1>Gab</h1>

    <div id='toolbar'>
      <span class='button' id='new-contact'>add contact..</span> ||
      <span class='button' id='new-chat'>chat with..</span> ||
      <span class='button' id='disconnect'>disconnect</span>
    </div>

    <div id='chat-area'>
      <ul></ul>
    </div>

    <div id='roster-area'>
      <ul></ul>
    </div>

    <!-- login dialog -->
    <div id='login_dialog' class='hidden'>
      <label>JID:</label><input type='text' id='jid'>
      <label>Password:</label><input type='password' id='password'>
    </div>

    <!-- contact dialog -->
    <div id='contact_dialog' class='hidden'>
      <label>JID:</label><input type='text' id='contact-jid'>
      <label>Name:</label><input type='text' id='contact-name'>
    </div>

    <!-- chat dialog -->
    <div id='chat_dialog' class='hidden'>
  
```

```

<label>JID:</label><input type='text' id='chat-jid'>
</div>

<!-- approval dialog -->
<div id='approve_dialog' class='hidden'>
    <p><span id='approve-jid'></span> has requested a subscription
       to your presence. Approve or deny?</p>
</div>
</body>
</html>

```

聊天区域只有一个元素，它将变成标签页条。每个可见的标签页均对应于这个元素的一个子元素(含有该标签页的名称)。每个标签页的内容将作为一个<div>元素插入到元素之后。jQuery UI 函数 tabs()将完成将这个简单的 HTML 结构转换成一组全功能的聊天标签页所涉及的复杂工作。

花名册区域最初同样也是一个空列表。一旦 Gab 从服务器那里获取到花名册联系人信息，就会将该信息动态地填充到这个列表中。

程序清单 6-2 给出了 Gab 的 CSS 代码。虽然样式非常简单，但在美化 Gab 界面方面已经取得了很大的进步。



程序清单 6-2 gab.css

可从
Wrox.com
下载源代码

```

body {
    font-family: Helvetica;
}

h1 {
    text-align: center;
}

.hidden {
    display: none
}

.button {
    padding: 10px;
    background-color: #ddd;
    border: solid 1px #666;
    font-weight: bold;
}

.button:hover {
    background-color: #ddf;
}

#toolbar {
    text-align: center;
}

```

```
        margin-bottom: 15px;
    }

    #chat-area {
        float: left;
        width: 600px;
        height: 300px;
    }

    .chat-messages {
        height: 180px;
        border-bottom: solid 2px #ddd;
        overflow: auto;
    }

    #chat-area input {
        margin-top: 10px;
        width: 95%;
    }

    #roster-area {
        float: right;
        border-left: solid 2px #ddd;
        padding: 10px;
        width: 250px;
        height: 300px;
    }

    .chat-name {
        color: #c33;
    }

    .chat-message .me {
        color: #33c;
    }

    .chat-event {
        font-style: italic;
        font-size: 75%;
    }

    .roster-contact {
        padding: 3px;
    }

    .roster-contact:hover {
        background-color: #aaa;
        color: white;
    }

    .roster-name {
        font-size: 150%;
    }
```

```

.online {
    color: #3c3;
}

.away {
    color: #c33;
}

.offline {
    color: #ccc;
}

```

这个界面的布局和样式相对简单，但借助 HTML 和 CSS 这样的强大工具实现更好的效果并非难事。

现在已经建立好用户界面，下面我们首先编写花名册区域的代码。

6.4 构建花名册

XMPP 聊天客户端通常会在启动时执行以下动作：

- (1) 连接服务器并进行身份验证。
- (2) 请求花名册。
- (3) 发送初始出席。

最后一步会让服务器向该用户已经订阅出席更新信息的联系人发送出席探测。这些探测将导致每一位在线的联系人向该用户发回一个<presence>节。

Gab 将需要重复这个动作序列并处理最终的花名册数据和出席更新信息。首先，编写登录对话框，就像我们在前几章中所做的那样，同时要构建基本的连接事件处理程序。创建一个名为 gab.js 的文件并插入下面的代码，经过前几章的学习，我们应该已经非常熟悉这些代码。



可以从
Wrox.com
下载源代码

```

var Gab = {
    connection: null
};

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Connect to XMPP',
        buttons: {
            "Connect": function () {
                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val()
                });
            }
        }
});

```

```

        $('#password').val('');
        $(this).dialog('close');
    }
}
});

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        'http://bosh.metajack.im:5280/xmpp-httpbind');

    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });
    Gab.connection = conn;
});

$(document).bind('connected', function () {
    // nothing here yet
});

$(document).bind('disconnected', function () {
    // nothing here yet
});

```

code snippet gab.js

一旦用户单击 Connect 按钮，该代码就开始连接到 XMPP 服务器。如果连接成功就会引发 connected 事件，而任何连接断开(或连接失败)都会引发 disconnected 事件。

现在可以在 connected 事件处理程序中请求花名册。

6.4.1 请求花名册

花名册是通过携带一个位于 `jabber:iq:roster` 命名空间下面的`<query>`元素的 IQ-get 和 IQ-set 节来进行操纵的。可以使用这些简单的 XMPP 节来检索花名册、修改和删除联系人。我们马上就会看到添加新联系人的具体方法，但现在我们只关注检索。

下面的 XMPP 节用来从服务器那里请求 Elizabeth 的花名册。

```

<iq from='elizabeth@longbourn.lit/library'
    type='get'
    id='roster1'>
    <query xmlns='jabber:iq:roster' />
</iq>

```

她的服务器将响应如下内容。

```
<iq to='elizabeth@longbourn.lit/library'
    type='result'
    id='roster1'>
  <query xmlns='jabber:iq:roster'>
    <item jid='darcy@pemberley.lit' name='Mr. Darcy' subscription='both' />
    <item jid='jane@longbourn.lit' name='Jane' subscription='both' />
  </query>
</iq>
```

这个花名册包含两个联系人。jid 属性是联系人的地址，而 name 属性是该联系人的用户指定别名。subscription 属性的值是根据联系人的出席订阅状态设置的。如果双向都有订阅，那么它的值为 both；如果 Elizabeth 有订阅但另一方没有订阅，那么该值为 to；如果 Elizabeth 没有订阅该联系人的出席信息但该用户订阅她的出席信息，那么该值为 from。一般而言，用户只希望看到花名册中 subscription 值为 both 或 to 的联系人。

将下面的 JavaScript 代码添加到 connected 事件处理程序中以检索花名册。



可从
Wrox.com
下载源代码

```
$(document).bind('connected', function () {
  var iq = $iq({type: 'get'}).c('query', {xmlns: 'jabber:iq:roster'});
  Gab.connection.sendIQ(iq, Gab.on_roster);
});
```

code snippet gab.js

接下来，将 on_roster() 的实现添加到 Gab 对象中。



可从
Wrox.com
下载源代码

```
on_roster: function (iq) {
  $(iq).find('item').each(function () {
    var jid = $(this).attr('jid');
    var name = $(this).attr('name') || jid;

    // transform jid into an id
    var jid_id = Gab.jid_to_id(jid);

    var contact = $("<li id='" + jid_id + "'>" +
      "<div class='roster-contact offline'>" +
      "<div class='roster-name'>" +
      name +
      "</div><div class='roster-jid'>" +
      jid +
      "</div></div></li>");
    Gab.insert_contact(contact);
  });
}
```

code snippet gab.js

可能已经注意到，在上面的事件处理程序中有一个新的 Strophe 库函数 sendIQ()。IQ 节的大部分常见处理方式均是由这个函数负责完成的，我们将在下一节中更完整地描述它。

`on_roster()` 函数遍历花名册项(每一项都有各自的`<item>`元素)，并携带适当的 HTML 来调用 `insert_contact()` 函数。为了以后查找联系人，有必要为它的元素指定一个 `id` 属性。这里的代码使用联系人的裸 JID 的稍作变形的版本作为 `id`。下面给出了 `jid_to_id()` 函数，应该将其添加到 `Gab` 对象中。



可从
Wrox.com
下载源代码

```
jid_to_id: function (jid) {
    return Strophe.getBareJidFromJid(jid)
        .replace("@", "-")
        .replace(".", "-");
}
```

code snippet gab.js

`insert_contact()` 函数用来让联系人列表保持正确排序。联系人的可访问性越好，他在花名册中的位置越高。如果两个联系人具有相同的可访问性，那么他们将按照 JID 排序。应该将下面的 `insert_contact()` 实现以及它的辅助函数 `presence_value()` 添加到 `Gab` 对象中。



可从
Wrox.com
下载源代码

```
presence_value: function (elem) {
    if (elem.hasClass('online')) {
        return 2;
    } else if (elem.hasClass('away')) {
        return 1;
    }

    return 0;
},

insert_contact: function (elem) {
    var jid = elem.find('.roster-jid').text();
    var pres = Gab.presence_value(elem.find('.roster-contact'));

    var contacts = $('#roster-area li');

    if (contacts.length > 0) {
        var inserted = false;
        contacts.each(function () {
            var cmp_pres = Gab.presence_value(
                $(this).find('.roster-contact'));
            var cmp_jid = $(this).find('.roster-jid').text();

            if (pres > cmp_pres) {
                $(this).before(elem);
                inserted = true;
                return false;
            } else {
                if (jid < cmp_jid) {
                    $(this).before(elem);
                }
            }
        });
    }
}
```

```

        inserted = true;
        return false;
    }
}

if (!inserted) {
    $('#roster-area ul').append(elem);
}
else {
    $('#roster-area ul').append(elem);
}
}

```

code snippet gab.js

如果运行当前状态的 Gab 程序，那么我们应该能够登录到服务器并会看到花名册显示出来并且已经排序。

6.4.2 处理 IQ

<iq>节在 XMPP 中比较特殊，它们是唯一需要有响应的 XMPP 节。每个 IQ-get 或 IQ-set 节都必须接收到相应的 IQ-result 或 IQ-error 节，就像在 HTTP 协议中任何 GET 或 POST 请求都必须接收到一个响应一样。所有的<iq>节也都必须携带一个 id 属性，它要有足够的唯一性以便能够在一个会话中识别它们。这是因为相同的 id 属性将用来标识响应，这样代码就知道特定的节 IQ-result 或 IQ-error 所对应的 IQ-get 或 IQ-set 节。

每次发送 IQ-get 或 IQ-set 节时，通常都要处理成功和错误响应。这意味着必须为我们发送的每个 IQ-get 或 IQ-set 节设置 XMPP 节处理程序。此外，还需要负责 id 属性的唯一性，这样才能区分传入的响应。

sendIQ()函数将所有这些需求以及相关的行为包装起来放入一个易用的接口中。sendIQ()函数并没有创建一个唯一的 id 值、设置成功和错误响应处理程序然后将<iq>节发送出去，而是接受一个<iq>节，确保它有一个唯一的 id，自动用我们提供的成功和错误回调来设置处理程序，并确保在回调执行完毕之后这些处理程序会得到适当的清除。

sendIQ()的调用方法如下：

```
Connection.sendIQ(iq_stanza, success_callback, error_callback);
```

成功和错误响应回调参数均是可选的。如果接收到 IQ-result 节，就引发成功回调；如果接收到 IQ-error 节，就引发错误响应回调。两种回调都只传递一个参数，即响应节。

此外，sendIQ()还接受一个超时值作为可选的第 4 个参数。如果在超时时间段内没有接收到 IQ-get 或 IQ-set 节的响应，就会自动触发错误响应回调。在面临导致响应延迟或不能发送的网络或服务错误时，这种超时机制可能非常有用，它会使我们的代码非常坚固。

sendIQ()使得 IQ 节的处理变得极其简单，在后面几章中我们还会看到更多示例。

6.4.3 更新出席状态

在检索到花名册之后，代码应该发送初始出席来启动出席探测并获取来自该用户的联系人的出席更新信息。为了不错过任何更新，我们总是应该在发送那些将要触发我们感兴趣的事件之前设置好合适的处理程序。修改 `on_roster()` 处理程序，添加下面被突出显示的代码。



可从
Wrox.com
下载源代码

```

on_roster: function (iq) {
    $(iq).find('item').each(function () {
        var jid = $(this).attr('jid');
        var name = $(this).attr('name') || jid;

        var jid_id = Gab.jid_to_id(jid);

        var contact = $("<li id='"
            + jid_id + "'>" +
            "<div class='roster-contact offline'>" +
            "<div class='roster-name'>" +
            name +
            "</div><div class='roster-jid'>" +
            jid +
            "</div></div></li>");

        Gab.insert_contact(contact);
    });

    // set up presence handler and send initial presence
    Gab.connection.addHandler(Gab.on_presence, null, "presence");
    Gab.connection.send($pres());
}

```

code snippet gab.js

接下来，实现 `on_presence()` 函数以相应地更新联系人列表。应该将下面的代码添加到 `Gab` 对象中。



可从
Wrox.com
下载源代码

```

on_presence: function (presence) {
    var ptype = $(presence).attr('type');
    var from = $(presence).attr('from');

    if (ptype !== 'error') {
        var contact = $('#roster-area li#' + Gab.jid_to_id(from))
            .removeClass("online")
            .removeClass("away")
            .removeClass("offline");
        if (ptype === 'unavailable') {
            contact.addClass("offline");
        } else {
            var show = $(presence).find("show").text();
            if (show === "" || show === "chat") {
                contact.addClass("online");
            } else {

```

```

        contact.addClass("away");
    }
}

var li = contact.parent();
li.remove();
Gab.insert_contact(li);
}

return true;
}

```

code snippet gab.js

出席节处理程序只是更新列表中的联系人的 CSS 类，这会让浏览器更新屏幕，之后将联系人删除，然后再将其重新插入，从而使列表正确排序。`on_presence()`处理程序使用与 `on_roster()` 相同的 `jid_to_id()` 转换来查找正确的待修改元素。

下面测试 Gab，使用它来登录您的账户。应该看到自己的花名册加载，并且当联系人上线、改变状态以及离线时您能够观察到颜色的变化。

6.4.4 添加新联系人

Gab 现在已经能够很好地看到用户当前的联系人，但还必须添加一项功能，让用户能够创建新联系人。新联系人的创建方式有两种：用户可以单击 **Add Contact** 并输入该联系人的信息，或者他们可以接收并接受来自某个新联系人的出席订阅请求。

首先，应该建立新联系人对话框并绑定适当按钮上的单击事件以打开对话框。将下面的代码添加到文档准备就绪处理函数中。



可从
Wrox.com
下载源代码

```

$( '#contact_dialog' ).dialog({
    autoOpen: false,
    draggable: false,
    modal: true,
    title: 'Add a Contact',
    buttons: {
        "Add": function () {
            $( document ).trigger('contact_added', {
                jid: $( '#contact-jid' ).val(),
                name: $( '#contact-name' ).val()
            });
            $( '#contact-jid' ).val('');
            $( '#contact-name' ).val('');
            $( this ).dialog('close');
        }
    }
});

$( '#new-contact' ).click(function (ev) {

```

```
$( '#contact_dialog' ).dialog( 'open' );
});
```

code snippet gab.js

当用户单击 Add 按钮时, 对话框会触发 contact_added 事件。我们需要处理这个事件, 向服务器发送适当的 XML, 从而将该联系人真正地添加到该用户的花名册中。应该将下面的处理程序添加到 gab.js 末尾处, 放在其他的文档准备就绪事件绑定的后面。



可从
Wrox.com
下载源代码

```
$(document).bind('contact_added', function (ev, data) {
    var iq = $iq({type: "set"}).c("query", {xmlns: "jabber:iq:roster"})
        .c("item", data);
    Gab.connection.sendIQ(iq);

    var subscribe = $pres({to: data.jid, "type": "subscribe"});
    Gab.connection.send(subscribe);
});
```

code snippet gab.js

contact_added 事件处理程序创建一个增加花名册用的 IQ-set 节并将其发送给服务器。然后, 它向这个新联系人发送一个类型为 subscribe 的出席节, 请求订阅该联系人的出席更新信息。注意, 这里没有代码来利用新的花名册状态来更新 UI。我们并不在这里处理这项工作, 而是创建另一个处理程序, 当用户的花名册状态改变时就会通知该处理程序。

6.4.5 响应花名册变化

用户可以多次连接到他们的 XMPP 服务器, 每个连接均被分配一个自己的资源(有关客户端资源的更多细节请参见第 1 章)。这些资源中的任何一个都可以修改花名册。为了让每个资源看到的花名册保持同步, XMPP 服务器将把花名册状态变化广播给所有已连接资源。

每个已连接资源都会得到花名册增加、删除以及修改的通知, 即使该操作是由该资源自身执行的。有些动作(比如请求出席订阅)也可能附带地导致花名册变化。这些变化也会以同样的方式广播。

假设 Jane 刚刚使用了 Gab 的新联系人对话框将 Bingley 添加到自己的花名册中。与刚才看到的代码类似, Jane 的客户端将发送一个如下所示的 IQ-set 节。

```
<iq from='jane@longbourn.lit/sitting_room'
    type='set'
    id='add1'>
    <query xmlns='jabber:iq:roster'>
        <item jid='bingley@netherfield.lit' name='Mr. Bingley' />
    </query>
</iq>
```

她的服务器应该用 IQ-result 应答来响应她的请求。

```
<iq to='jane@longbourn.lit/sitting_room' type='result' id='add1'/>
```

她的服务器还会用一个非常相似的 IQ-set 节将这个花名册状态变化广播出去。

```
<iq to='jane@longbourn.lit/sitting_room'
    type='set'
    id='changed1'>
  <query xmlns='jabber:iq:roster'>
    <item jid='bingley@netherfield.lit' name='Mr. Bingley' subscription='none' />
  </query>
</iq>
```

服务器的应答可能含有 Jane 发送的原始 IQ-set 节中没有指定的一些属性，就像在前面的代码中的 `subscription` 属性。注意，与任何其他 IQ 节一样，Jane 必须用 IQ-result 或 IQ-error 节来响应。

```
<iq type='result' id='changed1'/>
```

如果 Jane 已经被多个资源连接，那么所有那些连接都会接收到相同的花名册增加通知。

因为服务器向所有资源广播状态变化，所以许多 XMPP 客户端会等到接收到这个通知后才更新 UI。这还带来一个好处，那就是花名册不会显示不一致的状态。如果在接收到响应之前立即更新花名册，那么当服务器实际拒绝该操作时 UI 反映的却可能是增加成功。

现在我们可以向 Gab 中对添加这些花名册状态变化进行支持，不仅在用户添加新联系人之后显示修改后的花名册，而且还显示来自该用户其他资源的所有修改(当操作完成时)。下面被突出显示代码行给出了必须对 `connected` 事件处理程序做出的修改。



可从
Wrox.com
下载源代码

```
$(document).bind('connected', function () {
  var iq = $iq({type: 'get'}).c('query', {xmlns: 'jabber:iq:roster'});
  Gab.connection.sendIQ(iq, Gab.on_roster);

  Gab.connection.addHandler(Gab.on_roster_changed,
    "jabber:iq:roster", "iq", "set");
});
```

code snippet gab.js

还需要将下面定义的 `on_roster_changed()` 函数添加到 Gab 对象中。



可从
Wrox.com
下载源代码

```
on_roster_changed: function (iq) {
  $(iq).find('item').each(function () {
    var sub = $(this).attr('subscription');
    var jid = $(this).attr('jid');
    var name = $(this).attr('name') || jid;
    var jid_id = Gab.jid_to_id(jid);

    if (sub === 'remove') {
```

```

        // contact is being removed
        $('#' + jid_id).remove();
    } else {
        // contact is being added or modified
        var contact_html = "<li id=''" + jid_id + "'>" +
            "<div class=''" +
            ("'" + jid_id).attr('class') || "roster-contact offline") +
            "'>" +
            "<div class='roster-name'" +
            name +
            "</div><div class='roster-jid'" +
            jid +
            "</div></div></li>";

        if ("'" + jid_id).length > 0) {
            ("'" + jid_id).replaceWith(contact_html);
        } else {
            Gab.insert_contact(contact_html);
        }
    }
});

return true;
}

```

code snippet gab.js

每当接收到表示花名册更新的 IQ-set 节时都会触发该处理程序。如果<item>子元素的 subscription 属性值为 remove，那么对应的花名册项就会被删除。否则，添加一个新的花名册项会导致旧项被替换掉。在第 14 章中我们还将更多地讲解花名册操作方面的内容。

Gab 现在已经让花名册与其他客户端保持同步，显示联系人的出席更新信息，并让用户添加新联系人列表中。

6.4.6 处理订阅请求

用户接受新的出席订阅请求也会导致花名册增加的结果。回忆前面的示例，我们看到 Jane 将 Bingley 添加为联系人。在将其添加到花名册之后，她将发送一个出席订阅请求。在前面看到的事件处理程序中，Gab 会执行同样的步骤。

Jane 发送给 Bingley 的订阅请求如下所示。

```

<presence from='jane@longbourn.lit/sitting_room'
    to='bingley@netherfield.lit'
    type='subscribe' />

```

Bingley 可以通过回复一个携带 type 属性值为 subscribed 或 unsubscribed 的<presence>节来相应地批准或拒绝她的请求。他的响应(批准 Jane 的订阅请求)如下所示。

```
<presence from='bingley@netherfield.lit/parlor'
          to='jane@longbourn.lit/sitting_room'
          type='subscribed'/>
```

如果 Bingley 希望拒绝 Jane 的订阅请求，那么这个 XMPP 节的类型值为 `unsubscribed`。

如果 Bingley 的花名册中还没有把 Jane 作为联系人，那么服务器会立即将 Jane 加入进来并向他发送一条包含订阅状态的花名册更新信息。

```
<iq to='bingley@netherfield.lit/parlour'
    type='set'
    id='newcontact1'>
  <query xmlns='jabber:iq:roster'>
    <item jid='elizabeth@longbourn.lit'
          subscription='from' />
  </query>
</iq>
```

`subscription` 属性的值为 `from`，这是因为 Bingley 还没有订阅 Jane 的出席更新信息。通常情况下，他现在会请求订阅她的出席更新信息，而且如果 Jane 批准该请求，那么最终的订阅状态将是 `both`。

前面创建的花名册更新处理程序也可以用于批准订阅请求触发的变化通知，但仍然必须处理请求本身。应该将下面的代码添加到文档准备就绪事件处理程序中以启用批准对话框。



可从
Wrox.com
下载源代码

```
$('#approve_dialog').dialog({
  autoOpen: false,
  draggable: false,
  modal: true,
  title: 'Subscription Request',
  buttons: {
    "Deny": function () {
      Gab.connection.send($pres({
        to: Gab.pending_subscriber,
        "type": "unsubscribed"}));
      Gab.pending_subscriber = null;
      $(this).dialog('close');
    },
    "Approve": function () {
      Gab.connection.send($pres({
        to: Gab.pending_subscriber,
        "type": "subscribed"}));
      Gab.pending_subscriber = null;
      $(this).dialog('close');
    }
  }
});
```

code snippet gab.js

Approve 和 Deny 按钮分别用来发送携带 type 属性值 subscribed 和 unsubscribed 的<presence>响应节。如果批准请求，那么服务器产生一条花名册修改消息；如果请求被拒绝，就会将该拒绝消息通知给发送者，而且不会修改花名册。

接下来，必须修改前面编写的出席处理程序来处理传入的订阅请求。下面被突出显示的代码行给出了需要对 Gab 对象做出的修改。



可从
Wrox.com
下载源代码

```

pending_subscriber: null,
on_presence: function (presence) {
    var ptype = $(presence).attr('type');
    var from = $(presence).attr('from');

    if (ptype === 'subscribe') {
        // populate pending_subscriber, the approve-jid span, and
        // open the dialog
        Gab.pending_subscriber = from;
        $('#approve-jid').text(Strophe.getBareJidFromJid(from));
        $('#approve_dialog').dialog('open');
    } else if (ptype !== 'error') {
        var contact = $('#roster-area li#' + Gab.jid_to_id(from))
            .removeClass("online")
            .removeClass("away")
            .removeClass("offline");
        if (ptype === 'unavailable') {
            contact.addClass("offline");
        } else {
            var show = $(presence).find("show").text();
            if (show === "" || show === "chat") {
                contact.addClass("online");
            } else {
                contact.addClass("away");
            }
        }
        var li = contact.parent();
        li.remove();
        Gab.insert_contact(li);
    }
    return true;
}

```

code snippet gab.js

注意，Gab 对象已经增加了 pending_subscriber 属性。新的订阅请求处理代码将使用该属性。当 on_presence() 处理程序接收到一个订阅请求时，它只是生成该属性，然后准备并打开对话框。用户在对话框中的动作将决定产生哪一种响应。

Gab 现在处理传入的订阅请求，但通常当该请求得到批准时用户将希望发送一个补充的订

阅请求，这样最终的订阅将是双向的。修改 Approve 按钮的动作以匹配下面代码中被突出显示的代码行。



可从
Wrox.com
下载源代码

```

"Approve": function () {
    Gab.connection.send($pres({
        to: Gab.pending_subscriber,
        "type": "subscribed"}));

    Gab.connection.send($pres({
        to: Gab.pending_subscriber,
        "type": "subscribe"}));

    Gab.pending_subscriber = null;
}

```

code snippet gab.js

现在，每当用户批准订阅请求时，Gab 将发回一个订阅请求。Gab 的花名册区域此时已经完工，下面要处理聊天区域。

6.5 构建聊天对话

聊天是 Gab 的核心，而该动作完全发生在聊天区域。我们将使用 jQuery UI 的标签页控件来极大地简化多个聊天对话的显示以及相互切换。每个聊天对话标签页均代表一个与特定联系人之间的对话，而且每个标签页均还有一个输入区域来发送消息。每当接收到一条新的来自某位尚未有标签页的用户发来的消息时，都会建立一个新的聊天对话标签页。单击花名册中的联系人将选择一个现有的聊天对话标签页，或者如果还不存在这样的标签页，那么就为其创建一个新的标签页。

如果以前没有见过 jQuery UI 标签页控件，那么在开始编写代码之前，应该回顾一下该控件的基本知识。

6.5.1 处理标签页

jQuery UI 的标签页控件功能相当强大。它提供了熟悉的标签页功能，您以前很可能已经在各种桌面和 Web 应用程序中看到过该功能。它对元素 id 和结构有一些特殊的约束，而且它还提供了大量的选项。

标签区域包含一个条状物，它存放着所有带标签的标签页。这个条状物在 HTML 标记中就是一个元素。Gab 应用程序的 HTML 标记已经包含这个元素(在本章开头的程序清单 6-1 中可以看到)。每个标签页都有一个子元素，它的名字是一个携带特殊的 href 属性的超链接。

下面就是一个含有两个标签页的标签区域所需的 HTML 标记示例。

```

<div id='tab-example'>
<ul>
<li><a href='#tab-example-1'>First Tab</a></li>

```

```

<li><a href="#tab-example-2">Second Tab</a></li>
</ul>

<div id='tab-example-1'>
  <p>This is the first tab. Neat!</p>
</div>

<div id='tab-example-2'>
  <p>And here is the second tab.</p>
</div>
</div>

```

每个标签页在元素中都有一个项，以及一个对应的包含标签页内容的<div>元素。href 属性和<div>元素的 id 属性必须相同(href 属性中开头的“#”除外)。

可以调用 tabs()来初始化标签区域。例如，为了将上面的 HTML 标记转换成一个标签页控件，请运行下面的代码。

```
$('#tab-example').tabs();
```

可以通过 tabs()的 add 子命令来添加新的标签页。可以使用下面的代码向前面的标签区域中添加一个新的标签页。

```

$('#tab-example').tabs('add', '#tab-example-3', 'Third Tab');
$('#tab-example-3').html("<p>Finally, a third tab!</p>");

```

还可以使用 select 子命令通常编程方式来更改选中的标签页。下面的代码用来切换到第二个标签页。

```
$('#tab-example').tabs('select', '#tab-example-2');
```

标签页还支持其他几个子命令以及用来控制它们行为的各种选项。如果希望研究其他功能，请参阅 jQuery UI 的文档。

在下一节中，我们将运用这里学到的标签页新知识来创建一些聊天窗口。

6.5.2 创建新的聊天对话

最容易开始创建聊天对话的方法就是让用户单击花名册中的联系人来创建或选择聊天对话窗口。但在实现这些动作之前，我们必须首先初始化聊天区域。将下面的代码添加到文档准备就绪事件处理程序中。



可从
Wrox.com
下或源代码

```
$('#chat-area').tabs().find('.ui-tabs-nav').sortable({axis: 'x'});
```

code snippet gab.js

这段代码将聊天区域初始化为一个多标签页控件，然后使用 jQuery UI 的 sortable()函数让

标签条中的标签可排列。这可以让用户在标签页创建之后重新排列它们，这样它们就能够严格按照用户的喜好来排列。

现在我们已经有了一个真正的标签页区域，下面我们需要响应花名册联系人上的单击事件。到目前为止，我们总是使用 `bind()` 或 `click()` 来设置事件处理程序，但现在我们不得不使用 `live()`。与 `bind()` 类似，`live()` 将一个处理程序函数附加到一个事件，但它还为那些在调用 `live()` 之后创建的所有元素完成该操作。因为花名册联系人是动态添加的，所以必须使用 `live()` 来确保任何未来的花名册联系人都仍然获得正确的事件处理程序。

将下面的单击事件处理程序放入 Gab 的文档准备就绪事件处理程序中。



可从
Wrox.com
下载源代码

```
$('.roster-contact').live('click', function () {
    var jid = $(this).find('.roster-jid').text();
    var name = $(this).find('.roster-name').text();
    var jid_id = Gab.jid_to_id(jid);

    if ($('#chat-' + jid_id).length > 0) {
        $('#chat-area').tabs('select', '#chat-' + jid_id);
    } else {
        $('#chat-area').tabs('add', '#chat-' + jid_id, name);
        $('#chat-' + jid_id).append(
            "<div class='chat-messages'></div>" +
            "<input type='text' class='chat-input'>");
    }

    $('#chat-' + jid_id).data('jid', jid);
}

$('#chat-' + jid_id + ' input').focus();
});
```

code snippet gab.js

这个单击事件处理程序首先查找该联系人的 JID 和名字。它确定这个联系人是否已经存在一个聊天对话标签页，如果有的话，那么应选中它。否则，它为该联系人建立一个新的聊天对话标签页并初始化其内容。最后，将输入焦点定位到文本输入框，这样用户就能够立即开始输入消息了。请注意，这里使用了 jQuery 的 `data()` 函数利用`<div>`元素来保存该联系人的 JID。后面的其他代码还将使用这个 JID。

这些新建的聊天对话标签页目前还不是非常理想。我们需要让用户能够发送消息，并在接收到新消息时更新内容。

6.5.3 发送消息

每当用户在输入框内按下Enter键时，我们应该发送该用户的消息。我们仍然可以使用jQuery 的活动事件处理程序来完成这项工作：



```
$('.roster-input').live('keypress', function (ev) {
    if (ev.which === 13) {
        ev.preventDefault();
```

```

var body = $(this).val();
var jid = $(this).parent().data('jid');

Gab.connection.send($msg({
  to: jid,
  "type": "chat"
}).c('body').t(body));

$(this).parent().find('.chat-messages').append(
  "<div class='chat-message'>&lt;" +
  "<span class='chat-name me'>" +
  Strophe.getNodeFromJid(Gab.connection.jid) +
  "</span>&gt;<span class='chat-text'>" +
  body +
  "</span></div>");
}

Gab.scroll_chat(Gab.jid_to_id(jid));

$(this).val('');
}
);

```

code snippet gab.js

首先，阻止 Enter 键的默认动作(将该键添加到文本框的数据中)。检索消息的文本以及保存的 JID，然后将它们用来构建待发送的消息。同时还将该文本回显到显示屏上，并清空输入框，为用户输入新文本消息做好准备。

现在我们必须处理传入的消息，以便将它们的内容放入到适当的标签页中。下面被突出显示的代码行给出了需要对 connected 事件处理程序所做的修改，以建立新的<message>节处理程序。



可从
Wrox.com
下载源代码

```

$(document).bind('connected', function () {
  var iq = $iq({type: 'get'}).c('query', {xmlns: 'jabber:iq:roster'});
  Gab.connection.sendIQ(iq, Gab.on_roster);

  Gab.connection.addHandler(Gab.on_roster_changed,
    "jabber:iq:roster", "iq", "set");
  Gab.connection.addHandler(Gab.on_message,
    null, "message", "chat");
});

```

code snippet gab.js

接下来，必须在 Gab 对象中实现 on_message() 以及它的辅助函数 scroll_chat()。



可从
Wrox.com
下载源代码

```

on_message: function (message) {
  var jid = Strophe.getBareJidFromJid($(message).attr('from'));
  var jid_id = Gab.jid_to_id(jid);

  if ($('#chat-' + jid_id).length === 0) {

```

```

        $('#chat-area').tabs('add', '#chat-' + jid_id, jid);
        $('#chat-' + jid_id).append(
            "<div class='chat-messages'></div>" +
            "<input type='text' class='chat-input'>");
        $('#chat-' + jid_id).data('jid', jid);
    }

    $('#chat-area').tabs('select', '#chat-' + jid_id);
    $('#chat-' + jid_id + ' input').focus();

    var body = $(message).find("html > body");

    if (body.length === 0) {
        body = $(message).find('body');
        if (body.length > 0) {
            body = body.text()
        } else {
            body = null;
        }
    } else {
        body = body.contents();

        var span = $("<span></span>");
        body.each(function () {
            if (document.importNode) {
                $(document.importNode(this, true)).appendTo(span);
            } else {
                // IE workaround
                span.append(this.xml);
            }
        });
        body = span;
    }

    if (body) {
        // add the new message
        $('#chat-' + jid_id + ' .chat-messages').append(
            "<div class='chat-message'>" +
            "&lt;<span class='chat-name'>" +
            Strophe.getNodeFromJid(jid) +
            "</span>&gt;<span class='chat-text'>" +
            "</span></div>");

        $('#chat-' + jid_id + ' .chat-message:last .chat-text')
            .append(body);
        Gab.scroll_chat(jid_id);
    }

    return true;
},
scroll_chat: function (jid_id) {

```

```

var div = $('#chat-' + jid_id + '.chat-messages').get(0);
div.scrollTop = div.scrollHeight;
}

```

code snippet gab.js

`on_message()`处理程序稍微有点复杂。首先，它探测这个联系人是否已经打开一个聊天对话标签页，如果没有就创建一个标签页。然后，它选中该标签页，并将输入焦点放到输入框中。一旦正确显示该标签页，那么它就会提取消息正文(查找 XHTML-IM 负载，就像我们在第 5 章中看过的那样)、为该消息创建一个新的

元素，然后追加该正文。最后，如果将来它还会继续被调用来处理未来的消息，那么它返回 `true`。

Gab 现在是一个简单但有能力的即时通信客户端。花些时间对它进行测试，确保它能够按照预期运行。

6.6 即时通信最佳实践

Gab 目前有点过于简单，而且它并没有遵循标准的通信最佳实践。例如，输出的消息总是发送给用户的裸 JID，即使知道他们的完整地址。需要小心处理消息寻址，这是因为它们的传送受到服务器路由规则的影响。

6.6.1 理解消息路由

发送给已连接客户端的 XMPP <message> 节均专门由 XMPP 服务器路由。具体说来就是，服务器将根据 XMPP 节的 `to` 属性是裸 JID 还是完整 JID 来执行不同的操作。

寻址到用户裸 JID 的消息将被传送给具有最高优先级的已连接资源。如果多个资源并列具有最高优先级，那么服务器可能选择将消息传送给所有这些资源，或者选择最佳的一个资源。服务器如何选择最佳的传送资源取决于具体的服务器。大多数服务器会把消息传送给所有的并列资源，但也可能遇到下面这种情况，即服务器试图通过评估近期最活跃资源来猜测最佳资源。

寻址到完整 JID 的消息只会被传送给特定的资源，而如果该资源离线，那么有可能将该消息离线保存直到用户下次上线。

因为向裸 JID 发送消息可能导致多次传送，所以最好的做法是，一旦从另一方接收到完整地址，就将消息寻址到完整 JID。

例如，如果最近没有与某位用户通信，就应该将消息寻址到它们的裸 JID。一旦它们做出响应，就知道它们正在使用哪个资源与您进行交谈，然后就可以使用该目的地为未来发送给它们的消息进行寻址。

在聊天期间，用户可能上线和离线或离开然后又回来。因为在这些事件之后他们可能希望在不同位置接收消息，所以每当它们的出席状态改变时最好不再使用用户的完整地址。一旦得到他们的首次应答，就可以再次开始将消息寻址到(可能是新的)完整地址。

现在我们更好地理解了消息路由机制，下面我们更新 **Gab**，在发送消息时完成正确的操作。

6.6.2 更好地寻址消息

修改 Gab 将消息寻址到完整 JID 非常容易。由于我们已经将要发送消息的目的地址存放在聊天对话标签页的

元素中，因此我们只需将保存在那里的裸 JID 替换成用户的完整 JID 即可。当然，我们希望只有在 on_message() 中知道完整地址才来做这件事情。在 on_message() 处理程序中修改下面被突出显示的代码行。



可从
Wrox.com
下载源代码

```

on_message: function (message) {
    var full_jid = $(message).attr('from');
    var jid = Strophe.getBareJidFromJid(full_jid);
    var jid_id = Gab.jid_to_id(jid);

    if ($('#chat-' + jid_id).length == 0) {
        $('#chat-area').tabs('add', '#chat-' + jid_id, jid);
        $('#chat-' + jid_id).append(
            "<div class='chat-messages'></div>" +
            "<input type='text' class='chat-input'>");
    }

    $('#chat-' + jid_id).data('jid', full_jid);

    $('#chat-area').tabs('select', '#chat-' + jid_id);
    $('#chat-' + jid_id + ' input').focus();

    var body = $(message).find("html > body");

    if (body.length === 0) {
        body = $(message).find('body');
        if (body.length > 0) {
            body = body.text();
        } else {
            body = null;
        }
    } else {
        body = body.contents();

        var span = $("<span></span>");
        body.each(function () {
            if (document.importNode) {
                $(document.importNode(this, true)).appendTo(span);
            } else {
                // IE workaround
                span.append(this.xml);
            }
        });
        body = span;
    }

    if (body) {
        // add the new message
        $('#chat-' + jid_id + '.chat-messages').append(

```

```

    "<div class='chat-message'>" +
    "&lt;<span class='chat-name'>" +
    Strophe.getNodeFromJid(jid) +
    "</span>&gt;<span class='chat-text'>" +
    "</span></div>");

    $('#chat-' + jid_id + '.chat-message:last .chat-text')
        .append(body);

    Gab.scroll_chat(jid_id);
}

return true;
}

```

code snippet gab.js

每当从某位用户那里接收到一条消息时，就会把存放在<div>元素中的该用户的 JID 数据更新为完整地址。现在剩下的就是要在用户的出席状态改变时将该数据切换回裸 JID。在 `on_presence()` 处理程序中修改如下被突出显示的代码行。



可从
Wrox.com
下载源代码

```

on_presence: function (presence) {
    var ptype = $(presence).attr('type');
    var from = $(presence).attr('from');

    if (ptype === 'subscribe') {
        // populate pending_subscriber, the approve-jid span, and
        // open the dialog
        Gab.pending_subscriber = from;
        $('#approve-jid').text(Strophe.getBareJidFromJid(from));
        $('#approve_dialog').dialog('open');
    } else if (ptype !== 'error') {
        var contact = $('#roster-area li#' + Gab.jid_to_id(from))
            .removeClass("online")
            .removeClass("away")
            .removeClass("offline");
        if (ptype === 'unavailable') {
            contact.addClass("offline");
        } else {
            var show = $(presence).find("show").text();
            if (show === "" || show === "chat") {
                contact.addClass("online");
            } else {
                contact.addClass("away");
            }
        }
        var li = contact.parent();
        li.remove();
        Gab.insert_contact(li);
    }
}

```

```

// reset addressing for user since their presence changed
var jid_id = Gab.jid_to_id(from);
$('#chat-' + jid_id).data('jid', Strophe.getBareJidFromJid(from));

return true;
},

```

code snippet gab.js

当用户的出席状态改变并且了解它们的已连接资源时 Gab 现在应该会随之在完整寻址和裸寻址之间切换。

6.7 添加活动通知

许多客户端支持发送和接收活动通知。这些通知会在某人输入、关注聊天对话或者已离开或关闭聊天对话时通告聊天的另一方。这些事件的定义文档是 Chat State Notifications (XEP-0085)。为了让 Gab 更加用户友好，我们希望添加对显示和发送正在输入通知的支持。

6.7.1 理解聊天状态

以下是 5 种已定义的聊天状态。

- 活跃：用户正关注这个聊天对话。
- 不活跃：用户没有关注这个聊天对话。
- 撰写：用户正在为这个聊天对话输入一条消息。
- 暂停：用户曾经在输入，但已经短暂停止了。
- 离开：用户已经关闭或离开这个聊天对话。

每个状态都可以转变到另一个或多个状态。例如，一位处于撰写状态的用户可能转变成暂停或活跃状态，而一位处于离开状态的用户则只能转变成活跃状态。XEP-0085 文档给出了完整的状态转变列表。

有些客户端可能不支持活动通知。Gab 假设所有客户端都理解这些消息，这虽然不是最优的，但应该不会引起任何问题，因为那些不被理解的元素会被忽略。在实际的应用程序中，我们会希望探测对这个功能的支持情况，如果他们的客户端不能处理这些消息，就禁止向另一方发送通知。规范详细说明了进行这种探测的几种方法。

通知是通过<message>节来通告的。有些状态会随着消息正文一起通告，但有些状态则是通过自身的不携带任何正文的消息来发送的。下面的示例节显示了一个用户正在进入撰写状态，然后发送写好的消息，并转变为活跃状态。

```

<message from='darcy@pemberley.lit/roings'
  to='elizabeth@longbourn.lit/hunsford'
  type='chat'>

```

```

<composing xmlns='http://jabber.org/protocol/chatstates' />
</message>

<message from='darcy@pemberley.lit/rosings'
  to='elizabeth@longbourn.lit/hunsford'
  type='chat'>
<body>In vain have I struggled. It will not do. My feelings will not be repressed.
  You must allow me to tell you how ardently I admire and love you.</body>
<active xmlns='http://jabber.org/protocol/chatstates' />
</message>

```

Darcy 的客户端首先向 Elizabeth 发送一条通知告诉他已经开始输入消息。几秒钟之后，接收到一条普通消息，开始他的提议，以及第二条通知他正在关注这个聊天对话。Elizabeth 的客户端可以使用这些通知来适当地修改显示信息。

6.7.2 发送通知

Gab 只支持活跃状态和撰写状态。当用户开始输入时，就会将撰写状态通告给有意获知该信息的接收者。当写完消息时，就会发送活跃状态。这就是前一个示例中所演示的一系列事件。

首先，应该修改聊天标签页输入框的 keypress 事件处理程序来发送撰写状态通知。此外，一旦消息发送出去后，还需要追加活跃状态通知。下面给出了修改后的处理程序。



可从
Wrox.com
下载源代码

```

$('.chat-input').live('keypress', function (ev) {
  var jid = $(this).parent().data('jid');

  if (ev.which === 13) {
    ev.preventDefault();

    var body = $(this).val();

    var message = $msg({to: jid,
                       "type": "chat"})
      .c('body').t(body).up()
      .c('active', {xmlns:"http://jabber.org/protocol/chatstates"});
    Gab.connection.send(message);

    $(this).parent().find('.chat-messages').append(
      "<div class='chat-message'>&lt;" +
      "<span class='chat-name me'>" +
      Strophe.getNodeFromJid(Gab.connection.jid) +
      "</span>&gt;<span class='chat-text'>" +
      body +
      "</span></div>");

    Gab.scroll_chat(Gab.jid_to_id(jid));

    $(this).val('');
    $(this).parent().data('composing', false);
  } else {

```

```

        var composing = $(this).parent().data('composing');
        if (!composing) {
            var notify = $msg({to: jid, "type": "chat"})
                .c('composing', {xmlns: "http://jabber.org/protocol/
                chatstates"});
            Gab.connection.send(notify);
            $(this).parent().data('composing', true);
        }
    });
}

```

code snippet gab.js

当用户开始输入时，用户的聊天搭档现在就可以得到撰写通知了。如果运行 Gab 并在另一个客户端中启动一个与自己的对话，那么就可以看到这种情况。

6.7.3 接收通知

我们还希望向 Gab 中添加传入的撰写通知支持。为此，我们要观察传入消息中的撰写通知。将下面被突出显示的代码添加到 `on_message()` 处理程序中。



可从
Wrox.com
下载源代码

```

on_message: function (message) {
    var full_jid = $(message).attr('from');
    var jid = Strophe.getBareJidFromJid(full_jid);
    var jid_id = Gab.jid_to_id(jid);

    if ($('#chat-' + jid_id).length === 0) {
        $('#chat-area').tabs('add', '#chat-' + jid_id, jid);
        $('#chat-' + jid_id).append(
            "<div class='chat-messages'></div>" +
            "<input type='text' class='chat-input'>");
    }

    $('#chat-' + jid_id).data('jid', full_jid);
    $('#chat-area').tabs('select', '#chat-' + jid_id);
    $('#chat-' + jid_id + ' input').focus();

    var composing = $(message).find('composing');
    if (composing.length > 0) {
        $('#chat-' + jid_id + ' .chat-messages').append(
            "<div class='chat-event'>" +
            Strophe.getNodeFromJid(jid) +
            " is typing..</div>");
        Gab.scroll_chat(jid_id);
    }

    if (body.length === 0) {
        body = $(message).find('body');
        if (body.length > 0) {

```

```

        body = body.text();
    } else {
        body = null;
    }
} else {
    body = body.contents();

    var span = $("<span></span>");
    body.each(function () {
        if (document.importNode) {
            $(document.importNode(this, true)).appendTo(span);
        } else {
            // IE workaround
            span.append(this.xml);
        }
    });
    body = span;
}

if (body) {
    // remove notifications since user is now active
    $('#chat-' + jid_id + '.chat-event').remove();

    // add the new message
    $('#chat-' + jid_id + '.chat-messages').append(
        "<div class='chat-message'>" +
        "&lt;<span class='chat-name'>" +
        Strophe.getNodeFromJid(jid) +
        "</span>&gt;<span class='chat-text'>" +
        "</span></div>");

    $('#chat-' + jid_id + '.chat-message:last .chat-text')
        .append(body);
    Gab.scroll_chat(jid_id);
}

return true;
}

```

code snippet gab.js

每次另一方开始输入时，Gab 现在都会显示一条短小的消息来通知用户这个活动。当另一方最终发送他们的消息时，就会显示撰写通知。

6.8 收尾工作

还有两项小功能需要处理，也就是工具条中的另外两个按钮。

Disconnect 按钮相当简单。将下面的单击事件处理程序添加到文档准备就绪事件处理函数中。



可从
Wrox.com
下载源代码

```
$('#disconnect').click(function () {
    Gab.connection.disconnect();
});
```

code snippet gab.js

在 disconnected 事件处理程序中还必须重置 UI。在这个处理程序中修改如下被突出显示的代码行。



可从
Wrox.com
下载源代码

```
$(document).bind('disconnected', function () {
    Gab.connection = null;
    Gab.pending_subscriber = null;

    $('#roster-area ul').empty();
    $('#chat-area ul').empty();
    $('#chat-area div').remove();

    $('#login_dialog').dialog('open');
});
```

code snippet gab.js

为了实现 New Chat 对话框，我们必须首先初始化这个对话框，然后将按钮单击事件绑定到调用 dialog('open')。为此，将下面的代码添加到文档准备就绪事件处理函数中。



可从
Wrox.com
下载源代码

```
$('#chat_dialog').dialog({
    autoOpen: false,
    draggable: false,
    modal: true,
    title: 'Start a Chat',
    buttons: {
        "Start": function () {
            var jid = $('#chat-jid').val();
            var jid_id = Gab.jid_to_id(jid);

            $('#chat-area').tabs('add', '#chat-' + jid_id, jid);
            $('#chat-' + jid_id).append(
                "<div class='chat-messages'></div>" +
                "<input type='text' class='chat-input'>");
            $('#chat-' + jid_id).data('jid', jid);

            $('#chat-area').tabs('select', '#chat-' + jid_id);
            $('#chat-' + jid_id + ' input').focus();

            $('#chat-jid').val('');
            $(this).dialog('close');
        }
    }
});
```

```
});

$('#new-chat').click(function () {
    $('#chat_dialog').dialog('open');
});
```

code snippet gab.js

将最后这些代码添加进来之后，该应用程序已经完工了。Gab 已经不再是一个简单的聊天客户端，它还支持通知、可排列标签页、多聊天对话以及花名册。程序清单 6-3 给出了最终版本的代码。



可从
Wrox.com
下载源代码

程序清单 6-3 gab.js

```
var Gab = {
    connection: null,

    jid_to_id: function (jid) {
        return Strophe.getBareJidFromJid(jid)
            .replace("@", "-")
            .replace(".", "-");
    },

    on_roster: function (iq) {
        $(iq).find('item').each(function () {
            var jid = $(this).attr('jid');
            var name = $(this).attr('name') || jid;

            // transform jid into an id
            var jid_id = Gab.jid_to_id(jid);

            var contact = $("<li id='" + jid_id + "'>" +
                "<div class='roster-contact offline'>" +
                "<div class='roster-name'>" +
                name +
                "</div><div class='roster-jid'>" +
                jid +
                "</div></div></li>");

            Gab.insert_contact(contact);
        });

        // set up presence handler and send initial presence
        Gab.connection.addHandler(Gab.on_presence, null, "presence");
        Gab.connection.send($pres());
    },

    pending_subscriber: null,

    on_presence: function (presence) {
```

```

var ptype = $(presence).attr('type');
var from = $(presence).attr('from');
var jid_id = Gab.jid_to_id(from);

if (ptype === 'subscribe') {
    // populate pending_subscriber, the approve-jid span, and
    // open the dialog
    Gab.pending_subscriber = from;
    $('#approve-jid').text(Strophe.getBareJidFromJid(from));
    $('#approve_dialog').dialog('open');
} else if (ptype !== 'error') {
    var contact = $('#roster-area li#' + jid_id + '.roster-contact')
        .removeClass("online")
        .removeClass("away")
        .removeClass("offline");
    if (ptype === 'unavailable') {
        contact.addClass("offline");
    } else {
        var show = $(presence).find("show").text();
        if (show === "" || show === "chat") {
            contact.addClass("online");
        } else {
            contact.addClass("away");
        }
    }
    var li = contact.parent();
    li.remove();
    Gab.insert_contact(li);
}

// reset addressing for user since their presence changed
var jid_id = Gab.jid_to_id(from);
$('#chat-' + jid_id).data('jid', Strophe.getBareJidFromJid(from));

return true;
},
on_roster_changed: function (iq) {
    $(iq).find('item').each(function () {
        var sub = $(this).attr('subscription');
        var jid = $(this).attr('jid');
        var name = $(this).attr('name') || jid;
        var jid_id = Gab.jid_to_id(jid);

        if (sub === 'remove') {
            // contact is being removed .
            $('#' + jid_id).remove();
        } else {
            // contact is being added or modified
            var contact_html = "<li id='"
                + jid_id + "'>" +
                "<div class='"

```

```

        +">" +
        "<div class='roster-name'>" +
        name +
        "</div><div class='roster-jid'>" +
        jid +
        "</div></div></li>";

        if ($('#' + jid_id).length > 0) {
            $('#' + jid_id).replaceWith(contact_html);
        } else {
            Gab.insert_contact(contact_html);
        }
    }
});

return true;
},
on_message: function (message) {
    var full_jid = $(message).attr('from');
    var jid = Strophe.getBareJidFromJid(full_jid);
    var jid_id = Gab.jid_to_id(jid);

    if ($('#chat-' + jid_id).length === 0) {
        $('#chat-area').tabs('add', '#chat-' + jid_id, jid);
        $('#chat-' + jid_id).append(
            "<div class='chat-messages'></div>" +
            "<input type='text' class='chat-input'>");
    }

    $('#chat-' + jid_id).data('jid', full_jid);
    $('#chat-area').tabs('select', '#chat-' + jid_id);
    $('#chat-' + jid_id + ' input').focus();

    var composing = $(message).find('composing');
    if (composing.length > 0) {
        $('#chat-' + jid_id + ' .chat-messages').append(
            "<div class='chat-event'>" +
            Strophe.getNodeFromJid(jid) +
            " is typing..</div>");
        Gab.scroll_chat(jid_id);
    }

    var body = $(message).find("html > body");

    if (body.length === 0) {
        body = $(message).find('body');
        if (body.length > 0) {
            body = body.text()
        } else {
            body = null;
        }
    }
}
});
```

```

        }
    } else {
        body = body.contents();

        var span = $("<span></span>");
        body.each(function () {
            if (document.importNode) {
                $(document.importNode(this, true)).appendTo(span);
            } else {
                // IE workaround
                span.append(this.xml);
            }
        });
        body = span;
    }

    if (body) {
        // remove notifications since user is now active
        $('#chat-' + jid_id + '.chat-event').remove();

        // add the new message
        $('#chat-' + jid_id + '.chat-messages').append(
            "<div class='chat-message'>" +
            "&lt;<span class='chat-name'>" +
            Strophe.getNodeFromJid(jid) +
            "</span>&gt;<span class='chat-text'>" +
            "</span></div>");

        $('#chat-' + jid_id + '.chat-message:last .chat-text')
            .append(body);

        Gab.scroll_chat(jid_id);
    }
}

return true;
},
scroll_chat: function (jid_id) {
    var div = $('#chat-' + jid_id + '.chat-messages').get(0);
    div.scrollTop = div.scrollHeight;
},
presence_value: function (elem) {
    if (elem.hasClass('online')) {
        return 2;
    } else if (elem.hasClass('away')) {
        return 1;
    }
    return 0;
},
insert_contact: function (elem) {

```

```
var jid = elem.find('.roster-jid').text();
var pres = Gab.presence_value(elem.find('.roster-contact'));

var contacts = $('#roster-area li');

if (contacts.length > 0) {
    var inserted = false;
    contacts.each(function () {
        var cmp_pres = Gab.presence_value(
            $(this).find('.roster-contact'));
        var cmp_jid = $(this).find('.roster-jid').text();

        if (pres > cmp_pres) {
            $(this).before(elem);
            inserted = true;
            return false;
        } else {
            if (jid < cmp_jid) {
                $(this).before(elem);
                inserted = true;
                return false;
            }
        }
    });
    if (!inserted) {
        $('#roster-area ul').append(elem);
    }
} else {
    $('#roster-area ul').append(elem);
}
}

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Connect to XMPP',
        buttons: {
            "Connect": function () {
                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val()
                });

                $('#password').val('');
                $(this).dialog('close');
            }
        }
    });
});
```

```
$('#contact_dialog').dialog({  
    autoOpen: false,  
    draggable: false,  
    modal: true,  
    title: 'Add a Contact',  
    buttons: {  
        "Add": function () {  
            $(document).trigger('contact_added', {  
                jid: $('#contact-jid').val(),  
                name: $('#contact-name').val()  
            });  
  
            $('#contact-jid').val('');  
            $('#contact-name').val('');  
  
            $(this).dialog('close');  
        }  
    }  
});  
  
$('#new-contact').click(function (ev) {  
    $('#contact_dialog').dialog('open');  
});  
  
$('#approve_dialog').dialog({  
    autoOpen: false,  
    draggable: false,  
    modal: true,  
    title: 'Subscription Request',  
    buttons: {  
        "Deny": function () {  
            Gab.connection.send($pres({  
                to: Gab.pending_subscriber,  
                "type": "unsubscribed"}));  
            Gab.pending_subscriber = null;  
  
            $(this).dialog('close');  
        },  
  
        "Approve": function () {  
            Gab.connection.send($pres({  
                to: Gab.pending_subscriber,  
                "type": "subscribed"}));  
  
            Gab.connection.send($pres({  
                to: Gab.pending_subscriber,  
                "type": "subscribe"}));  
  
            Gab.pending_subscriber = null;  
  
            $(this).dialog('close');  
        }  
    }  
});
```

```

});
```

```

$('#chat-area').tabs().find('.ui-tabs-nav').sortable({axis: 'x'});
```

```

$('.roster-contact').live('click', function () {
    var jid = $(this).find(".roster-jid").text();
    var name = $(this).find(".roster-name").text();
    var jid_id = Gab.jid_to_id(jid);

    if ($('#chat-' + jid_id).length === 0) {
        $('#chat-area').tabs('add', '#chat-' + jid_id, name);
        $('#chat-' + jid_id).append(
            "<div class='chat-messages'></div>" +
            "<input type='text' class='chat-input'>");
        $('#chat-' + jid_id).data('jid', jid);
    }
    $('#chat-area').tabs('select', '#chat-' + jid_id);
    $('#chat-' + jid_id + ' input').focus();
});
```

```

$('.chat-input').live('keypress', function (ev) {
    var jid = $(this).parent().data('jid');

    if (ev.which === 13) {
        ev.preventDefault();

        var body = $(this).val();

        var message = $msg({to: jid,
                           "type": "chat"})
                     .c('body').t(body).up()
                     .c('active', {xmlns: "http://jabber.org/protocol/chatstates"});

        Gab.connection.send(message);

        $(this).parent().find('.chat-messages').append(
            "<div class='chat-message'>&lt;" +
            "<span class='chat-name me'>" +
            Strophe.getNodeFromJid(Gab.connection.jid) +
            "</span>&gt;<span class='chat-text'>" +
            body +
            "</span></div>");
        Gab.scroll_chat(Gab.jid_to_id(jid));

        $(this).val('');
        $(this).parent().data('composing', false);
    } else {
        var composing = $(this).parent().data('composing');
        if (!composing) {
            var notify = $msg({to: jid, "type": "chat"})
                        .c('composing', {xmlns: "http://jabber.org/protocol/chatstates"});
            Gab.connection.send(notify);
        }
    }
});
```

```

        $(this).parent().data('composing', true);
    }
})
));

$('#disconnect').click(function () {
    Gab.connection.disconnect();
    Gab.connection = null;
});

$('#chat_dialog').dialog({
    autoOpen: false,
    draggable: false,
    modal: true,
    title: 'Start a Chat',
    buttons: {
        "Start": function () {
            var jid = $('#chat-jid').val();
            var jid_id = Gab.jid_to_id(jid);

            $('#chat-area').tabs('add', '#chat-' + jid_id, jid);
            $('#chat-' + jid_id).append(
                "<div class='chat-messages'></div>" +
                "<input type='text' class='chat-input'>");
            $('#chat-' + jid_id).data('jid', jid);

            $('#chat-area').tabs('select', '#chat-' + jid_id);
            $('#chat-' + jid_id + ' input').focus();
            $('#chat-jid').val('');

            $(this).dialog('close');
        }
    }
});

$('#new-chat').click(function () {
    $('#chat_dialog').dialog('open');
});
});

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        'http://bosh.metajack.im:5280/xmpp-httpbind');

    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });
    Gab.connection = conn;
});

```

```

});
```

```

$(document).bind('connected', function () {
  var iq = $iq({type: 'get'}).c('query', {xmlns: 'jabber:iq:roster'});
  Gab.connection.sendIQ(iq, Gab.on_roster);

  Gab.connection.addHandler(Gab.on_roster_changed,
    "jabber:iq:roster", "iq", "set");
  Gab.connection.addHandler(Gab.on_message,
    null, "message", "chat");
});
```

```

$(document).bind('disconnected', function () {
  Gab.connection = null;
  Gab.pending_subscriber = null;

  $('#roster-area ul').empty();
  $('#chat-area ul').empty();
  $('#chat-area div').remove();

  $('#login_dialog').dialog('open');
});
```

```

$(document).bind('contact_added', function (ev, data) {
  var iq = $iq({type: "set"}).c("query", {xmlns: "jabber:iq:roster"})
    .c("item", data);
  Gab.connection.sendIQ(iq);

  var subscribe = $pres({to: data.jid, "type": "subscribe"});
  Gab.connection.send(subscribe);
});
```

6.9 更多 Gab 功能

您可能熟悉的大多数客户端的功能均要比 Gab 的功能多得多。可以将这些功能中的一些添加到 Gab 中使其更具竞争力。

试着添加如下这些简单功能：

- 添加对其他聊天通知状态(比如暂停和离开)的支持。
- 允许用户使用基本文本的标记(比如 Markdown 或 Textile)来发送内容丰富的 XHTML-IM 消息。
- 使用 Personal Eventing Protocol(XEP-0163)扩展，在聊天区域显示用户的联系人正在听什么音乐。其他几款聊天客户端可发送这类通知。

6.10 小结

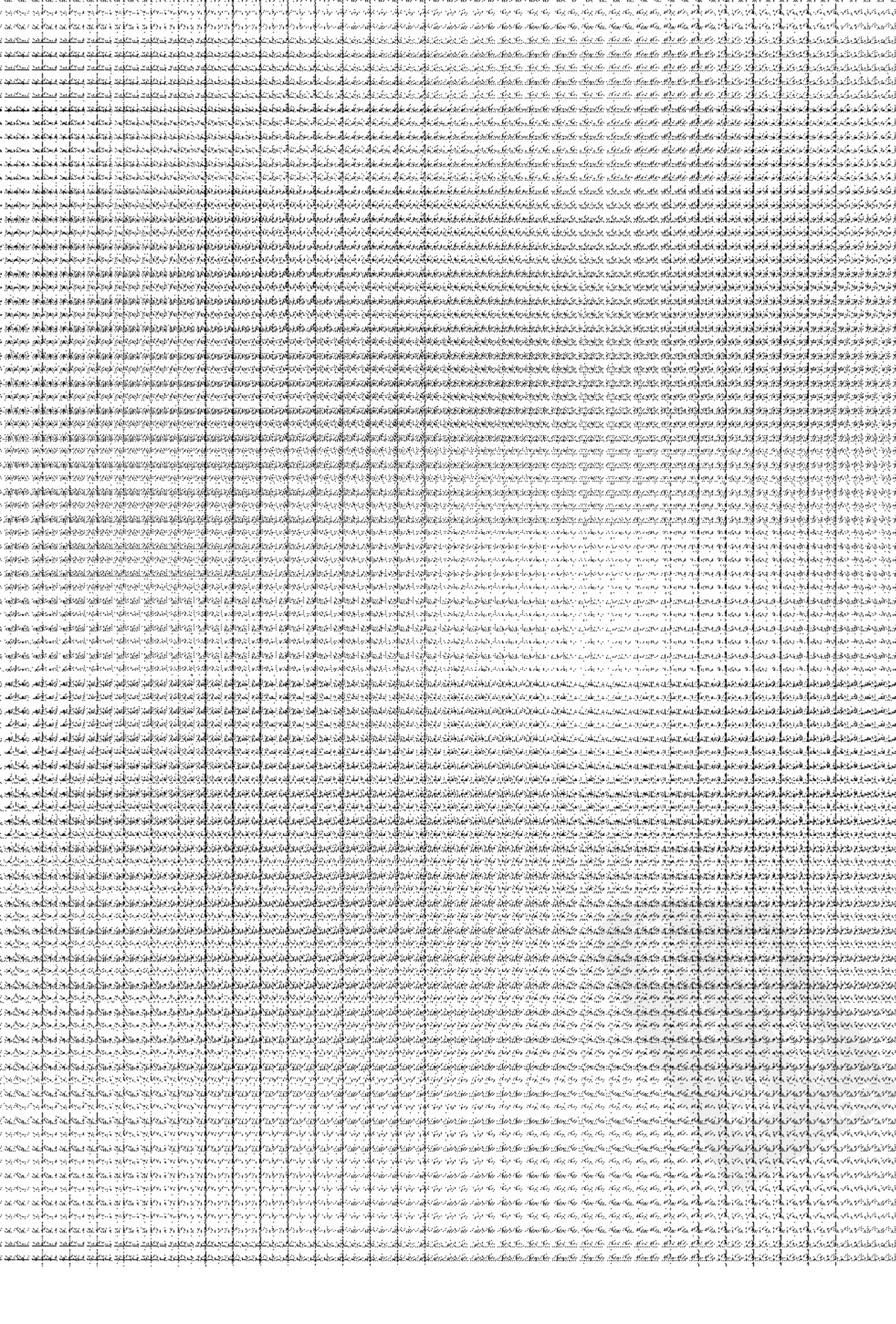
即时通信是最初的 XMPP 应用领域。通过创建 Gab 应用程序，我们已经开发出了一个有能

力的 IM 客户端，它能够管理花名册和多聊天会话。Gab 要比前几章的应用程序复杂得多，通过坚持开发工作并实现 Gab 我们学到了：

- 如何获取和处理花名册
- 如何处理花名册变化
- 如何添加和编辑联系人
- 如何发送和响应出席订阅请求
- XMPP 服务器如何路由消息
- 了解 jQuery UI 的标签页控件
- 如何发送和解释聊天对话的活动通知

在第 7 章中我们将学习如何利用 XMPP 服务发现机制来发现服务以及它们的功能。





7

第 7 章

探索服务：服务发现与浏览

本章内容

- 服务发现的基本知识
- 翻译服务发现结果
- 服务发现树
- 其他 XMPP 扩展如何使用服务发现机制

XMPP 服务器和服务的联合网络世界是一个庞大的场所，而且它每天都在不断在壮大。早期，XMPP 开发人员就明白他们需要某种方式来查找有关这些服务器以及它们提供哪些服务的信息。XMPP 服务发现系统就是为了满足这个需要而建立的。

我们可以向网络上几乎任何一个 XMPP 可寻址的实体发送服务发现请求。应用程序能够弄明白哪些是服务器、哪些是发布-订阅系统以及哪些是多人聊天服务。服务信息通常包含支持功能列表，因此应用程序代码绝不需要猜测支持什么功能。

大多数 XMPP 扩展协议都使用了服务发现系统，通常用来通告对功能的支持。有些还依靠它来进行服务浏览。例如，可以查询多人聊天服务有哪些聊天室，给定聊天室应用了哪些设置，甚至有哪些用户出席。

服务发现非常简单但却是许多 XMPP 协议的重要组成部分，我们的应用程序也经常会在各种任务中使用它。在本章中，我们将构建一个名为 Dig 的服务浏览程序，可以让用户来研究不同的 XMPP 服务。

7.1 应用程序预览

Dig 应用程序类似于您可能比较熟悉的其他目录浏览界面，比如 Windows 资源管理器或 Mac OS X 的 Finder。图 7-1 给出了最终的应用程序的外观。

左侧是给定实体的服务树。右侧的窗格显示关于该树中选中项的信息。用户可以在应用程序的顶部输入想要查询服务的实体。

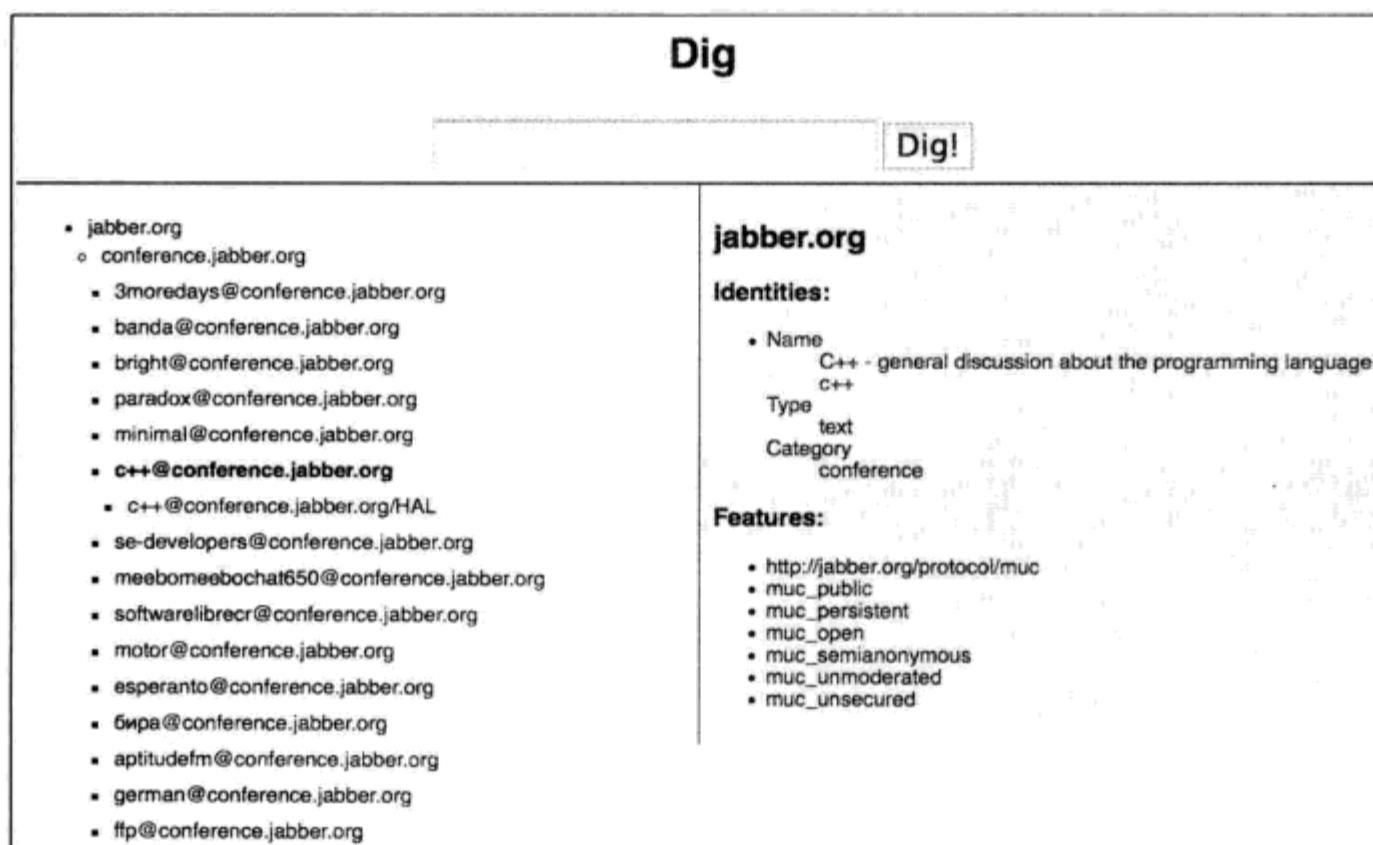


图 7-1

图 7-1 给出的树只是 jabber.org 多人聊天服务器 conference.jabber.org 的服务树的一部分。

7.2 Dig 的设计

Dig 必须在实体上执行两种服务发现查询, 即 disco#info 和 disco#items。名称中的“#”取自它们的底层命名空间。disco#info 查询返回实体的基本身份信息以及支持的功能。disco#items 查询枚举实体的子节点。

一旦用户输入他们想要查看的实体, 就要运行这两种查询, 查询结果开始在左侧窗格中形成一棵树。在该树的叶节点上单击会导致进一步的服务发现查询并进一步扩展该树。disco#info 查询的结果出现在右侧信息窗格中, 而 disco#items 查询结果用来扩展树。

应用程序的 UI 相当简单, 但用户要学习的有关 XMPP 服务器和服务的信息总量却是非常庞大的。

7.3 查找信息

服务发现(由 XEP-0030 定义, 常被称为 disco)由与某个实体或服务相关的信息层次结构以及一对查询 disco#info 和 disco#items(分别用来请求信息和遍历树)组成。

7.3.1 disco#info 查询

我们要交互的大多数 XMPP 实体均会响应 disco#info 查询。一般说来, 这些查询涉及两类信息: 实体和功能。实体信息包含实体的名称和作用, 而且某些服务可能有多个实体。功能信

息用于发现某项支持哪些特定的功能。例如，发布-订阅服务可能报告它支持 Publish-Subscribe 规范(XEP-0060)中定义的所有必需功能以及一些可选功能，此外还支持其他几项功能。

disco#info 查询非常简单。它们由一个使用 <http://jabber.org/protocols/disco#info> 命名空间的 IQ-get 节组成。

```
<iq to='pemberley.lit'
    from='elizabeth@longbourn.lit/lambton'
    type='get'
    id='info1'>
  <query xmlns='http://jabber.org/protocol/disco#info' />
</iq>
```

在这个示例查询中，Elizabeth 正在探测 pemberley.lit 服务器的信息。这个服务器将使用一个身份和功能列表进行响应。

```
<iq to='elizabeth@longbourn.lit/lambton'
    from='pemberley.lit'
    type='result'
    id='info1'>
  <query xmlns='http://jabber.org/protocol/disco#info'>
    <identity name='Pemberley XMPP Server'
              category='server'
              type='im' />
    <feature var='http://jabber.org/protocol/disco#info' />
    <feature var='jabber:client' />
    <feature var='jabber:iq:roster' />
    <feature var='jabber:iq:version' />
    <feature var='msgoffline' />
  </query>
</iq>
```

pemberley.lit 服务器报告它的名称并说明自己是一个即时通信服务器。它还列举了该服务器支持的功能，包括对 disco#info 的支持、标准 XMPP 客户端的支持、对花名册命令的支持、对版本查询的支持、对离线用户消息存储的支持。

XMPP 服务器对 disco#info 查询有不同的响应，但其响应形式是一样的。通常，XMPP 服务器将支持范围很广的功能，身份和功能列表都要比虚构的 pemberley.lit 服务器的列表长很多。disco#info 响应必须至少包含一个身份并且至少包含 disco#info 功能。

每个<identity>元素包含一个可选的 name 属性以及必需的 category(类别)和 type(类型)属性。XMPP 注册机构负责维护一个标准的类别和类型值列表，可以在 <http://xmpp.org/registrar/disco-categories.html> 网页中找到它。那些在单个地址上提供多项服务的实体将返回多个<identity>元素，通常每项服务对应一个元素。

<feature>元素列出了实体提供的基本功能。必需属性 var 携带着该功能的标识符。这些标识符通常对应于各种 XMPP 协议扩展使用的命名空间，XMPP 注册机构负责维护一个已注册功能列表，可以在 <http://xmpp.org/registrar/disco-features.html> 网页中找到。在前一个示例中，除了

msgoffline 功能之外均使用相对命名空间作为标识符。

应用程序可以使用该信息来测试特定实体提供什么类型的服务，或者确定特定服务是否支持某种具体的功能。例如，并不是所有的会议服务器都支持群聊的服务器端日志功能。那些支持该功能的服务器就可以使用一个`<feature>`元素来通告这类支持。

7.3.2 disco#items 查询

服务查询信息以树的形式组织。disco#items 查询请求实体的子节点列表。这些子节点中有一些是其他的实体，而有些是内部节点。通过在每个子节点上继续执行 disco#items 查询，就可以展开整棵服务树。

在 XMPP 服务器上请求 disco#items，将返回构成该服务器的各种服务器端组件，每一个都是独立的可寻址的实体。例如，发布-订阅服务将返回顶级节点列表，每个节点还可能有子节点。多人聊天服务器回答 disco#items 查询时返回聊天室列表。

disco#items 查询与 disco#info 查询几乎相同。

```
<iq to='pemberley.lit'
  from='elizabeth@longbourn.lit/lambton'
  type='get'
  id='items1'>
  <query xmlns='http://jabber.org/protocol/disco#items' />
</iq>
```

可以看到，只有命名空间的最后部分有所不同。这里给出了来自 pemberley.lit 的响应。

```
<iq to='elizabeth@longbourn.lit/lambton'
  from='pemberley.lit'
  type='result'
  id='items1'>
  <query xmlns='http://jabber.org/protocol/disco#items'>
    <item jid='pubsub.pemberley.lit'
      name='The Pemberley Pubsub System' />
    <item jid='chat.pemberley.lit'
      name='The Pemberley Multi-User Chat System' />
    <item jid='pemberley.lit'
      node='statistics'
      name='Pemberley Server Statistics' />
  </query>
</iq>
```

实体的每个子项均由响应中的一个`<item>`元素表示。上面的示例给出了 3 个子项，其中两个是服务器组件，另一个是内部节点。所有这些子项也都响应 disco#info 和 disco#items 查询。

7.3.3 disco 节点

服务树的内部暴露为节点。前一节中看到的 disco#items 查询包含一个 statistics 节点，它是 pemberley.lit 子节点之一。通过在请求的`<query>`元素中包含一个 node 属性，disco 节点是可寻

址的：

```

<iq to='pemberley.lit'
    from='elizabeth@longbourn.lit/lambton'
    type='get'
    id='info2'>
<query xmlns='http://jabber.org/protocol/disco#info'
    node='statistics' />
</iq>

```

statistics 节点的应答将包括一个或多个<identity>元素以及一个或多个<feature>元素，就像任何其他实体一样。对 statistics 节点进行 disco#items 查询还可能让更多子节点显现出来。

除了各种错误情况和其他一些小的细节，这些简单的部分就是服务发现的全部内容。我们已经拥有构建 Dig 所需的全部知识。

7.4 创建 Dig

与前面的应用程序一样，开始构建 Dig 时我们首先来创建它的用户界面。图 7-1 中的截屏显示了三个主要部分：服务输入控件、树形窗格以及信息窗格。程序清单 7-1 给出了 Dig 的 HTML 标记，程序清单 7-2 给出了 CSS 样式。



程序清单 7-1 dig.html

可从
Wrox.com
下载源代码

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
    <head>
        <meta http-equiv="Content-type" content="text/html; charset=UTF-8">
        <title>Dig - Chapter 7</title>

        <link rel='stylesheet' href='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/themes/cupertino/jquery-ui.css'>
        <script src='http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/jquery.js'>
        </script>
        <script src='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/jquery-ui.js'></script>
        <script src='scripts/strophe.js'></script>
        <script src='scripts/flXHR.js'></script>
        <script src='scripts/strophe.flxhr.js'></script>

        <link rel='stylesheet' type='text/css' href='dig.css'>
        <script type='text/javascript' src='dig.js'></script>
    </head>
    <body>
        <h1>Dig</h1>

```

```
<div id='input-bar'>
  <input id='service' type='text'>
  <input id='dig' type='button' value='Dig!' disabled='disabled'>
</div>

<div class='clear'></div>

<div class='left'>
  <ul id='tree'>
  </ul>
</div>

<div class='right'>
  <h2 id='selected-name'></h2>
  <div id='disco-info'>
    <h3>Identities:</h3>
    <ul id='identity-list'>
    </ul>

    <h3>Features:</h3>
    <ul id='feature-list'>
    </ul>
  </div>
</div>

<!-- login dialog -->
<div id='login_dialog' class='hidden'>
  <label>JID:</label><input type='text' id='jid'>
  <label>Password:</label><input type='password' id='password'>
</div>
</body>
</html>
```



可以从
Wrox.com
下载源代码

程序清单 7-2 dig.css

```
body {
  font-family: Helvetica;
}

h1 {
  text-align: center;
}

.hidden {
  display: none
}

.left {
  padding: 10px;
  width: 400px;
  float: left;
}
```

```

.right {
    padding: 10px;
    border-left: solid 1px black;
    width: 500px;
    float: right;
    background-color: #eee;
}

.clear {
    clear: both;
}

.item {
    padding: 4px;
}

.selected {
    background-color: #ff6;
    font-weight: bold;
}

#input-bar {
    padding: 10px;
    text-align: center;
    border-bottom: solid 2px black;
}

input {
    font-size: 150%;
}

#tree ul {
    line-height: 175%;
    margin-left: 0;
    padding-left: 10px;
}

```

输入控件包含一个文本输入框和一个按钮，树形窗格包含一个空白的无序列表，信息窗格包含一部分用来显示实体，另一个部分用来显示功能。样式只是让这些不同的元素的默认呈现稍微美观一点。树中选中的实体将突出显示为黄色和粗体，而不同的 UI 组件有一些额外的空白或更大的字体。

程序清单 7-3 包含了 Dig 的 JavaScript 代码骨架。我们在前几章中已经见过所有这些元素，因此在这里不再赘述。



可从
Wrox.com
下载源代码

程序清单 7-3 dig.js(初始骨架)

```

var Dig = {
    connection: null,
};

```

```

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Connect to XMPP',
        buttons: {
            "Connect": function () {
                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val()
                });

                $('#password').val('');
                $(this).dialog('close');
            }
        }
    });
});

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        'http://bosh.metajack.im:5280/xmpp-httpbind');
    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });
    Dig.connection = conn;
});

$(document).bind('connected', function () {
    // nothing here yet
});

```

将这些基本代码准备好之后，下面就可以开始处理应用程序的实质问题。

7.4.1 初始 disco 查询

一旦用户已经把一个 JID 输入到文本输入框并单击 Dig! 按钮，代码就必须将初始的 disco#info 和 disco#items 查询发送给指定的服务。

首先，绑定 Dig! 按钮的单击事件并发送这两种 disco 查询。应该将下面的代码放到文档准备就绪事件处理程序中。



```

$('#dig').click(function () {
    var service = $('#service').val();
    $('#service').val('');

```

```

// set up disco info pane
$('#selected-name').text(service);
$('#identity-list').empty();
$('#feature-list').empty();

// clear tree pane
$('#tree').empty();

$('#tree').append("<li><span class='item selected'>" +
    service +
    "</span></li>");

Dig.connection.sendIQ(
    $iq({to: service, type: "get"})
    .c("query", {xmlns:
        "http://jabber.org/protocol/disco#info"}),
    function (iq) {
        Dig.on_info(iq, $('.selected')[0]);
    });

Dig.connection.sendIQ(
    $iq({to: service, type: "get"})
    .c("query", {xmlns:
        "http://jabber.org/protocol/disco#items"}),
    function (iq) {
        Dig.on_items(iq, $('.selected')[0]);
    });
);

```

code snippet dig.js

这里的代码首先获取服务的名称，并清空文本输入框。然后该代码清空信息窗格并创建初始的 disco 树。最后，它向该服务发送 disco#info 和 disco#items 查询。

注意，当前选中的元素也会被传给 on_info() 和 on_items()。之所以传入这个额外信息是因为，这样一来处理程序就知道应该将查询结果指派给该树的哪一个分支。如果没有提供该信息，那么当用户在接收到查询的响应之前单击另一个不同的分支，他可能会看到一个分支的结果出现在另一个分支的下面。

现在必须实现 on_info() 和 on_items() 函数。下面的代码将第一个函数添加到 Dig 对象中。



可从
Wrox.com
下载源代码

```

on_info: function (iq, elem) {
    // do nothing if the response is not for the selected branch
    if ($.selected.length > 0 &&
        elem !== $('.selected')[0]) {
        return;
    }

    $('#feature-list').empty();
    $(iq).find("feature").each(function () {
        $('#feature-list').append("<li>" +

```

```

        $(this).attr('var') +
        "</li>");

    $('#identity-list').empty();
    $(iq).find("identity").each(function () {
        $('#identity-list').append("<li><dl><dt>Name</dt><dd>" +
            ($(this).attr('name') || "none") +
            "</dd><dt>Type</dt><dd>" +
            ($(this).attr('type') || "none") +
            "</dd><dt>Category</dt><dd>" +
            ($(this).attr('category') || "none") +
            "</dd></dl></li>");
    });
}

```

code snippet dig.js

`on_info()`函数简单地提取每个功能和每个身份并将它们格式化成列表元素。很少用到的定义列表(`<dl>`元素)有助于格式化每个身份的属性。

`on_items()`函数也相当简单。



可从
Wrox.com
下载源代码

```

on_items: function (iq, elem) {
    var items = $(iq).find("item");
    if (items.length > 0) {
        $(elem).parent().append("<ul></ul>");

        var list = $(elem).parent().find("ul");

        $(iq).find("item").each(function () {
            var node = $(this).attr('node');
            list.append("<li><span class='item'>" +
                $(this).attr("jid") +
                (node ? ":" + node : "") +
                "</span></li>");
        });
    }
}

```

code snippet dig.js

如果返回任何`<item>`元素，那么这些新项就会被添加到树形窗格中的树的选中分支中。注意，如果一些项指向 `disco` 节点而不是 `JID`，那么它们显示为 `jid:node`。这可能并不是显示信息的最佳方式，但当后面我们必须处理树中子项的单击事件时它将有所帮助。

最后，需要在 `connected` 事件处理程序中启用 `Dig!`按钮。对该事件处理程序进行如下的修改。



可从
Wrox.com
下载源代码

```
$(document).bind('connected', function () {
    $('#dig').removeAttr('disabled');
});
```

code snippet dig.js

此时运行 Dig 我们应该能够看到顶级子节点和有关实体的基本信息。接下来，我们需要添加浏览和扩展树的功能。

7.4.2 浏览 disco 树

到目前为止，Dig 还只是获取服务的高层视图。我们需要将其展开以便让用户单击该树的各种分支来改变信息窗格中显示的信息，并展开该树位于当前分支下面的部分。

因为 `on_items()` 和 `on_info()` 函数如此通用，所以我们唯一必须添加的就是一个用来处理树形窗格中的单击事件的处理程序。将下面的处理程序代码添加到文档准备就绪事件处理程序中。



可从
Wrox.com
下载源代码

```
$('#tree .item').live('click', function () {
    if ($(this).hasClass("selected")) {
        return;
    }

    $(".selected").removeClass("selected");
    $(this).addClass("selected");

    var serv_node = $(this).text();
    var service, node;
    var idx = serv_node.indexOf(":");
    if (idx < 0) {
        service = serv_node;
        node = null;
    } else {
        service = serv_node.slice(0, idx);
        node = serv_node.slice(idx + 1);
    }

    var queryAttrs;
    if (node) {
        queryAttrs = { node: node };
    } else {
        queryAttrs = {};
    }

    var elem = this;
    queryAttrs["xmlns"] = "http://jabber.org/protocol/disco#info";
    Dig.connection.sendIQ(
        $iq({to: service, type: "get"})
            .c("query", queryAttrs),
        function (iq) {
```

```

        Dig.on_info(iq, elem);
    });

    if ($.selected).parent().find("ul").length === 0) {
        queryAttrs["xmlns"] = "http://jabber.org/protocol/disco#items";
        Dig.connection.sendIQ(
            $iq({to: service, type: "get"})
                .c("query", queryAttrs),
            function (iq) {
                Dig.on_items(iq, elem);
            });
    }
});

```

code snippet dig.js

jQuery 的 `live()` 函数用来将一个事件处理程序绑定到匹配的元素。与使用 `click()` 或 `bind()` 来完成这件工作不同之处在于, `live()` 甚至可以在那些在该绑定创建之后动态添加进来的元素上进行绑定。我们在第 6 章中已经使用过 `live()`。

该代码首先检查被单击的是否是选中的分支, 如果是的话就返回。在这里我们不需要重做查询。如果接收到单击事件的分支并不是选中分支, 就把旧的选择去掉, 被单击的分支现在变成了选中的分支。

接下来的代码块将分析选中``元素中的文本, 以确定这个分支的树项的服务和节点。我们需要这两种信息来完成 `disco` 查询。

最后, 向正确的服务和节点发送 `disco#info` 查询, 而且如果前面没有针对该树的这部分进行 `disco#items` 查询, 那么还会发送一个 `disco#items` 查询。因为 `on_items()` 为相关的树分支创建子列表, 所以检查子元素``就足以确定 `on_items()` 是否已经执行过。这两个 `disco` 查询都使用我们前面创建的 `on_items()` 和 `on_info()` 处理程序。

`Dig` 已经完工了, 它现在能够用来研究大量的公共 XMPP 服务。程序清单 7-4 给出了最终的源代码。在第 7.5 节中, 我们将使用 `Dig` 来浏览一些知名的服务。



可从
Wrox.com
下载源代码

程序清单 7-4 dig.js(最终版)

```

var Dig = {
    connection: null,
    on_items: function (iq, elem) {
        var items = $(iq).find("item");
        if (items.length > 0) {
            $(elem).parent().append("<ul></ul>");
            var list = $(elem).parent().find("ul");
            $(iq).find("item").each(function () {
                var node = $(this).attr('node');

```

```
list.append("<li><span class='item'>" +
            $(this).attr("jid") +
            (node ? ":" + node : "") +
            "</span></li>");
        });
    },
    on_info: function (iq, elem) {
        // do nothing if the response is not for the selected branch
        if ($.selected.length > 0 &&
            elem !== $('.selected')[0]) {
            return;
        }

        $('#feature-list').empty();
        $(iq).find("feature").each(function () {
            $('#feature-list').append("<li>" +
                $(this).attr('var') +
                "</li>");
        });

        $('#identity-list').empty();
        $(iq).find("identity").each(function () {
            $('#identity-list').append("<li><dl><dt>Name</dt><dd>" +
                ($(this).attr('name') || "none") +
                "</dd><dt>Type</dt><dd>" +
                ($(this).attr('type') || "none") +
                "</dd><dt>Category</dt><dd>" +
                ($(this).attr('category') || "none") +
                "</dd></dl></li>");
        });
    }
};

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Connect to XMPP',
        buttons: {
            "Connect": function () {
                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val()
                });

                $('#password').val('');
                $(this).dialog('close');
            }
        }
});
```

```

        }
    });

    $('#dig').click(function () {
        var service = $('#service').val();
        $('#service').val('');

        // set up disco info pane
        $('#selected-name').text(service);
        $('#identity-list').empty();
        $('#feature-list').empty();

        // clear tree pane
        $('#tree').empty();

        $('#tree').append("<li><span class='item selected'>" +
                           service +
                           "</span></li>");
    });

    Dig.connection.sendIQ(
        $iq({to: service, type: "get"})
            .c("query", {xmlns:
                           "http://jabber.org/protocol/disco#info"}),
        function (iq) {
            Dig.on_info(iq, $('.selected')[0]);
        });
}

Dig.connection.sendIQ(
    $iq({to: service, type: "get"})
        .c("query", {xmlns:
                           "http://jabber.org/protocol/disco#items"}),
    function (iq) {
        Dig.on_items(iq, $('.selected')[0]);
    });
});

$('#tree .item').live('click', function () {
    if ($(this).hasClass("selected")) {
        return;
    }

    $(".selected").removeClass("selected");
    $(this).addClass("selected");

    var serv_node = $(this).text();
    var service, node;
    var idx = serv_node.indexOf(":");
    if (idx < 0) {
        service = serv_node;
        node = null;
    } else {

```

```

        service = serv_node.slice(0, idx);
        node = serv_node.slice(idx + 1);
    }

    var query_attrs;
    if (node) {
        query_attrs = { node: node };
    } else {
        query_attrs = {};
    }

    var elem = this;
    query_attrs["xmlns"] = "http://jabber.org/protocol/disco#info";
    Dig.connection.sendIQ(
        $iq({to: service, type: "get"})
            .c("query", query_attrs),
        function (iq) {
            Dig.on_info(iq, elem);
        });
    if ($.(".selected").parent().find("ul").length === 0) {
        query_attrs["xmlns"] = "http://jabber.org/protocol/disco#items";
        Dig.connection.sendIQ(
            $iq({to: service, type: "get"})
                .c("query", query_attrs),
            function (iq) {
                Dig.on_items(iq, elem);
            });
    }
});
});

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        'http://bosh.metajack.im:5280/xmpp-bind');

    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });
    Dig.connection = conn;
});

$(document).bind('connected', function () {
    $('#dig').removeAttr('disabled');
});

```

7.5 挖掘服务

服务发现可以提供各种有用的功能。应用程序可以使用或实现服务发现以便：

- 确定对特定软件功能的支持。
- 使用 disco#items 结果来生成资源列表，比如公共多人聊天室。
- 遍历 disco 树搜索诸如专有 IM 网络网关或代理服务之类的服务。
- 向其他请求服务发现的实体通告有关应用程序的元数据。

在下一节中，我们将使用 Dig 来完成类似的任务。我们将试着弄清楚服务器是否提供代理服务，该服务器是否支持注册，以及可以加入它的多人聊天服务中的哪些公共聊天室。

7.5.1 查找代理服务

XMPP 客户端通常支持联系人之间的点到点文件传送。但由于网络限制，许多对等体位于 NAT(网络地址转换设备)或企业防火墙后面。在这些场合中，有必要让双方使用第三方代理服务来完成传送。

可能碰巧双方都不知道要使用什么代理服务，但借助服务发现浏览功能，他们的客户端软件可以确定用户的服务器是否提供这种服务。通常，这种检查过程对最终用户是透明的，但这里我们使用 Dig 手工查找答案。

在 Web 浏览器中打开 Dig 应用程序，在文本输入框中输入 jabber.org，然后单击 Dig! 按钮。您应该会看到与图 7-2 中所示类似的结果，但如果 jabber.org 服务改变或者升级了它的一些服务，那么您看到的结果可能有所不同。

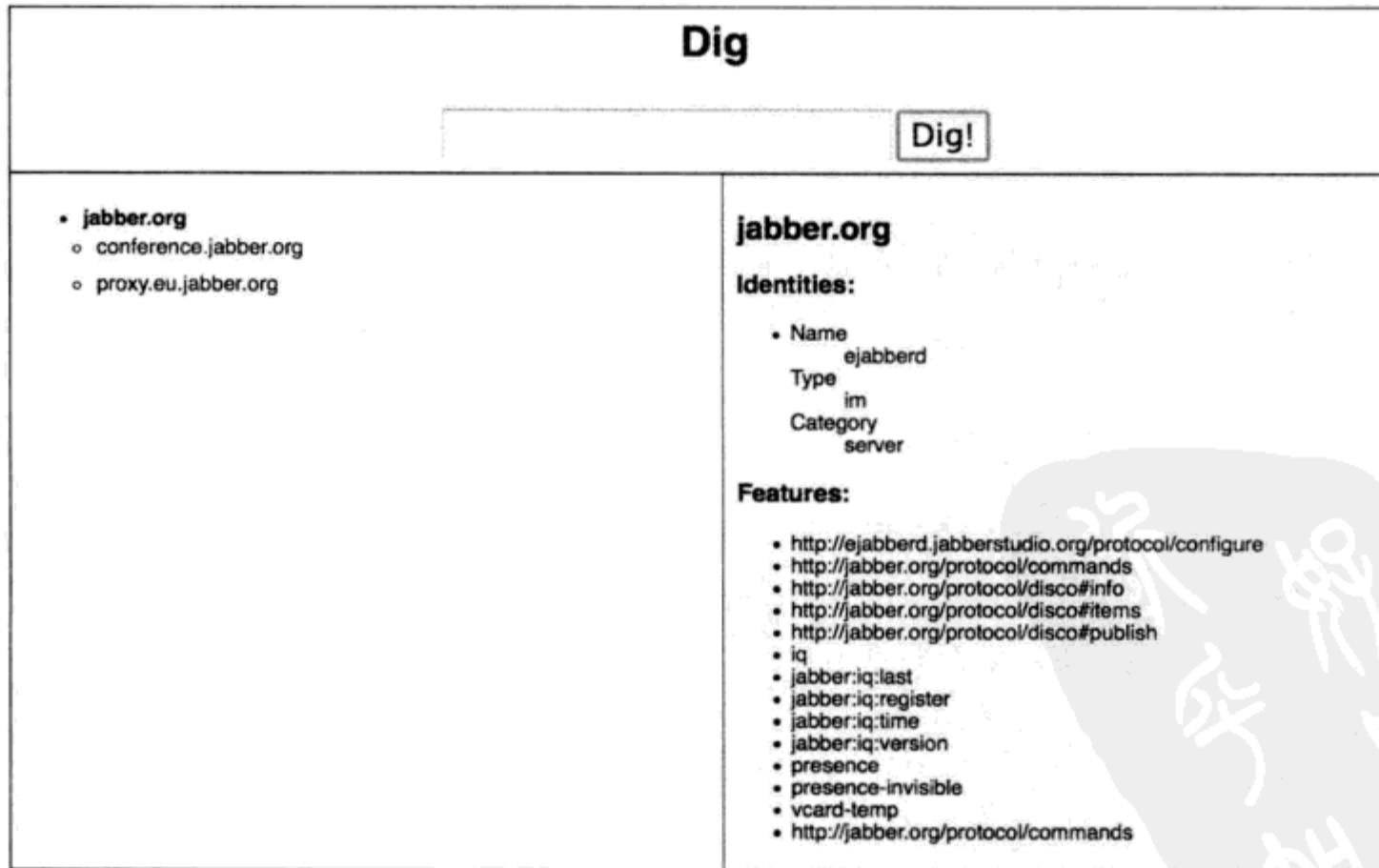


图 7-2

应该看到潜在目标 proxy.eu.jabber.org 项。单击该项以获取更多信息，应该会在信息窗格中看到与图 7-3 所示类似的信息。

XMPP Bytestreams(更多信息请参见 XEP-0065)代理正是完成点到点文件传送所需的。

通常，应用程序将按照类型、类别或功能而不是按照名称来搜索特定的服务项。在这种情况下，由于人类的直觉是从服务的名称来推导可能的功能，因此这让搜索无法进行。

注意，proxy.eu.jabber.org 项并没有任何子节点，因此在它之下该树不再是可展开的。

7.5.2 发现功能

有些服务器支持通过 XMPP 创建账户。如果应用程序用到这项功能，那么最好使用服务发现机制来检查 XMPP 服务器是否支持该功能。

如果前一个示例中用到的 Dig 应用程序还没有关闭，那么只需单击该树的根节点 jabber.org。否则，打开 Dig 应用程序，并浏览 jabber.org 的 disco 树。应该会看到如图 7-4 所示的功能列表。

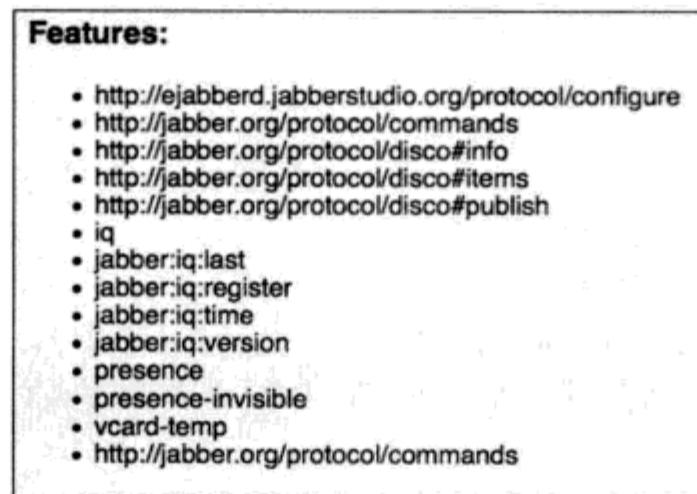


图 7-4

可以看到，jabber:iq:register 功能出现在该列表中，因此可以确信该服务器支持注册。

7.5.3 寻找聊天对话

服务发现机制还可用来列举可公开访问的资源，可能是发布-订阅节点、多人聊天室或共享文件列表。从前两个示例可以看出，jabber.org 服务器提供 conference.jabber.org 服务。可以浏览该服务来获取公共房间列表。

使用 Dig 浏览 jabber.org 的 disco 树并单击 conference.jabber.org 项。应该看到如图 7-5 所示的长长的聊天室列表。

如果单击任何一个房间，就会在右侧窗格中看到有关该房间的更多信息。可能会注意到，在单击特定房间之后，位于它下面的树就展开了。这些子项是这个聊天对话的参与者，

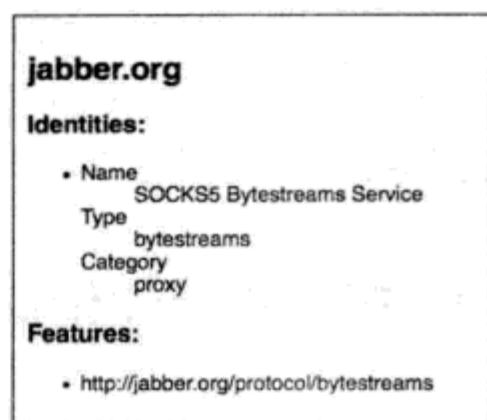


图 7-3



图 7-5

每个 JID 的资源部分就是该参与者的别名。

现在我们已经完成 Dig 的构建并看到服务发现机制的实际运行情况，我们已经准备好在自己的应用程序中使用它了。

7.6 服务发现的更多功能

Dig 已经可以用来浏览 disco 树，但我们可以试着添加一些新的功能，让它变得更好：

- 添加按照类别或类型(而不是节点名称)来分组节点的功能。
- 允许用户针对他们感兴趣的特定功能来过滤树。

7.7 小结

本章讲解了一个名为服务发现的简单的但非常重要的 XMPP 扩展，它常被称为 disco。使用服务发现，应用程序可以浏览可访问的资源、确定特定的功能以及查找有关各种服务的元数据。应用程序还可以实现服务发现以便让其他实体来收集有关该应用程序的资源的信息。

我们创建了一个名为 Dig 的简单的服务发现浏览器，可用来研究 jabber.org 服务树的多个领域。在这个过程中我们学习了如下内容：

- 如何进行 disco#info 和 disco#items 查询
- 如何解释查询结果并使用结果项来构建 disco 树
- 如何使用 disco 来确定属性支持情况
- 如何列举公共聊天室和其他公共资源
- 如何查找应用程序所需的特定服务

第 8 章整章内容都是关于我们这里探索过的服务(多人聊天)的。

第 8 章

群聊：多人聊天客户端

本章内容

- 加入群聊房间
- 创建和配置房间
- 在群聊中交换出席状态和消息
- 多人聊天角色和权限
- 主持房间

XMPP 的多人聊天扩展最初受到 IRC(Internet Relay Chat, 互联网中继聊天)协议的启发。有些聪明的 XMPP 协议设计者希望在几个方面改进 IRC, 与此同时让群聊成为 XMPP 的原生功能。

XMPP 的 MUC(multi-user chat, 多人聊天)与它的前辈相比具有一个很大的优点, 那就是结构化有效载荷。IRC 和其他聊天协议通常来回传送不带结构的普通文本。由于 XMPP 的可扩展性, MUC 消息即使在普通的聊天消息中也可以携带任意复杂的有效载荷。

MUC 还是一种消息广播形式。发送给房间的单条消息会自动地重新广播到所有的参与者。这使它与发布-订阅优点相似, 我们将在第 9 章中讨论。但与发布-订阅不同的是, MUC 提供了群聊服务特有的大量的高级管理功能(比如主持房间), 而且每个参与者通常也可以向房间广播消息。

结构化通信、自动化广播以及群发布功能为许多 MUC 替代用法打开了大门。它已经被用作丰富的协作空间(比如 Drop.io, 我们已在第 2 章讲过), 而在第 11 章中, 我们将基于 MUC 构建一个多人游戏 Tic-Tac-Toe。

多人聊天是一个庞大的协议, 而一个功能齐全的客户端可能非常复杂。在本章中, 我们将构建一个名为 Groupie 的简单群聊客户端, 它将用到 MUC 协议的最重要的几部分。

8.1 应用程序预览

图 8-1 给出了完工之后的 Groupie 应用程序的截屏。与前几章一样, 我们让这个应用程序的界面尽可能简单。

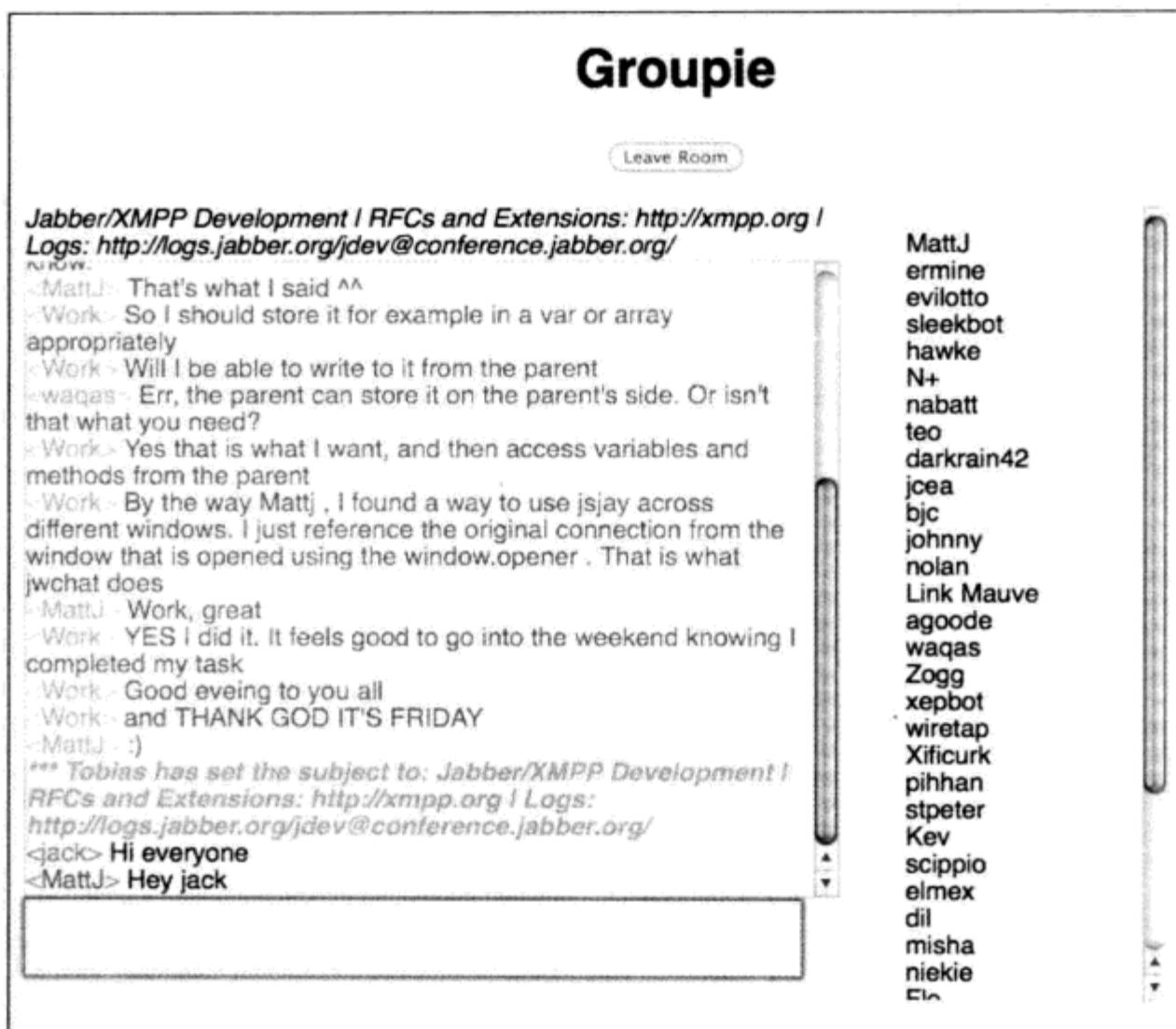


图 8-1

在这个屏幕的上方是聊天和参与者区域。聊天区域包含了人们发送到房间的所有消息以及用户发送和接收到的个人消息。参与者区域包含了由正在与用户交谈的当前房客组成的列表。在聊天区域下面是用户用来输入和发送消息的文本框。

8.2 Groupie 的设计

尽管 Groupie 的 UI 设计看似简单，但想要做出一个有用的群聊客户端还需要完成一些复杂的工作。Groupie 将需要支持像房间创建与配置、主题更换、参与者之间的个人通信以及基本的主持功能这样的功能。

当应用程序启动时，用户将看到我们在前几章中看到过的登录对话框的扩展版本。这个新的登录对话框包含了一些新的表单字段：用户希望加入的 MUC 房间以及用户希望在该房间中使用的别名。如果该房间已经存在，那么用户将加入到该房间中的其他参与者中，而如果该房间尚不存在，就会创建该房间。

一旦进入房间，来自其他参与者的消息就会出现在聊天区域中，而用户也可以在文本输入框中输入自己的消息。当人们进入或离开房间时，聊天区域右侧的参与者列表会随之变化。在文本框中使用特殊的命令可以向其他参与者发送个人消息，而发送给该用户的个人消息将在正常的聊天区域中显示，但呈现方式有所不同。

我们还将实现另外两个特殊的命令，即用户动作和主题更换。用户动作可以让用户描述自己的动作，类似于某个人以第三人称的方式来谈论自己。主题更换功能可以让用户修改房间的主题(如果他们在该房间中具有足够的权限)。

群聊房间有不同的角色和岗位，它们分别控制着对房间的短期和长期权限和访问。当房间内的某些用户失控时，有必要通过操作他们的角色让他们闭嘴或将他们请出房间，而更改参与者的岗位则可以永久性地让他们成为管理员或禁止入内。Groupie 通过一些特殊的文本命令来支持这些操作。我们将在第 8.3.6 节中更多地讨论这些主题。

这看似需要实现大量的功能，但这些功能只是 MUC 房间各种可能的功能的冰山一角。在开始编写代码之前，我们应该熟悉一下构成多人聊天协议的各种 XMPP 节。

8.3 公开发言

群聊功能可以让多人聚集在同一个地方来讨论一个主题。这些虚拟的会议场所被称为房间，与真实世界不同的是，用户可以在同一时间待在多个群聊房间中。房间具有访问控制、主持人、管理员，甚至还有群通信的自动日志和归档功能。在开始启动应用程序的相关工作之前，我们必须首先学习如何在协议层面上完成这些群聊操作。

8.3.1 群聊服务

群聊通常作为一项服务随普通 XMPP 服务器提供。群聊服务有自己的域，例如，jabber.org 服务器的群聊服务器运行在 conference.jabber.org。

群聊服务中的每个房间都有自己的地址，看上去就像是用户的 JID 一样。XMPP 开发人员在 conference.jabber.org 服务上的房间是 jdev@conference.jabber.org，而一般性的与 XMPP 相关的聊天室在 jabber@conference.jabber.org。

许多 XMPP 服务器都会运行群聊服务，而这些服务上散布着成千上万的房间。在第 7 章中曾经看到过，我们可以使用服务发现机制来查找和定位群聊服务以及构建于它们之上的公开房间。

8.3.2 进入和离开房间

在能够在群聊房间中执行任何操作之前，必须首先进入该房间。这通常被称为加入(join)房间。当完成参与之后，离开房间。因为这反映了用户进入、上线以及离线的概念，所以 MUC 设计者决定使用<presence>节来为这部分协议建模。

用户只需向群聊房间发送可访问(available)出席信息就可以加入该房间(注意他们能否理解 MUC 协议)。直接向 JID(而不是用户的服务器)发送出席状态信息，这被称为定向出席(directed presence)。与此类似，要想离开，就向该房间发送不可访问(unavailable)的出席信息。



定向出席

直接向 JID(而不是用户的服务器)发送出席状态信息，这被称为发送定向出席。定向出席在 XMPP 协议和各种扩展中都相当有用，因为它有一些特殊性质。

可以向用户或服务发送定向出席而不需要建立出席订阅。这可用来让另一个用户或外部服务临时访问出席状态信息。

定向出席的另一个性质是服务器记录哪些接收者已经接收到定向出席通知。服务器使用该信息来确保当发送者离线(甚至发送者在注销前忘记发送不可访问出席信息)时接收者得到通知。

在使用定向消息时有一点需要注意，只有不可访问出席信息是将被自动发送的。出席状态从可访问变为离开或从离开变为可访问状态均不会自动为发送者广播。

因为群聊服务需要记录参与者的出席状态，所以定向出席实现了这个关键角

群聊房间中的每个参与者也都有自己的地址。每个参与者针对该房间挑选一个别名，他们在这个房间内的 JID 就是房间的 JID 加上包含他们的别名的资源部分。例如，Darcy 在 Meryton 舞会聊天室中的别名是 darcy，因此他的群聊 JID 就是 ball@chat.meryton.lit/darcy。

如果 Bingley 和 Jane 希望加入 Meryton 舞会群聊房间，那么他们都需要向他们在 ball@chat.meryton.lit 房间中的预期身份发送定向出席消息。下面给出了他们的 XMPP 节。

```
<presence to='ball@chat.meryton.lit/bingley'
          from='bingley@netherfield.lit/meryton'>
  <x xmlns='http://jabber.org/protocol/muc' />
</presence>

<presence to='ball@chat.meryton.lit/jane'
          from='jane@longbourn.lit/meryton'>
  <x xmlns='http://jabber.org/protocol/muc' />
</presence>
```

一旦加入房间，群聊服务就会将所有其他参与者的出席状态信息广播给他们。在将所有其他参与者的出席节发送完毕之后，服务器最终通过将到来的参与者的出席状态信息发送给每个人(包括新到者)来结束出席广播。因此，当新参与者看到他们自己的出席状态广播回来时，他们知道自己已经完全加入到该房间了。

下面就是 Jane 的客户端在加入房间时接收到的信息。

```
<presence to='jane@longbourn.lit/meryton'
          from='ball@chat.meryton.lit/elizabeth'>
  <x xmlns='http://jabber.org/protocol/muc'>
    <item affiliation='member' role='participant' />
  </x>
</presence>

<presence to='jane@longbourn.lit/meryton'
          from='ball@chat.meryton.lit/bingley'>
  <x xmlns='http://jabber.org/protocol/muc' />
```

```

<item affiliation='member' role='participant'/>
</x>
</presence>

<presence to='jane@longbourn.lit/meryton'
          from='ball@chat.meryton.lit/jane'>
<x xmlns='http://jabber.org/protocol/muc'>
<item affiliation='member' role='participant'/>
<status code='110' />
</x>
</presence>

```

房间会将每位参与者的岗位和角色与出席状态信息一同发送。Jane 自己的出席状态广播还包括状态码 110，这说明这条出席状态信息指的是该用户自己。与来自 Jane 花名册的出席更新不同，当人们离开或新人加入时，Jane 还会接收到来自该房间的出席更新。

有可能房间中的某个人已经使用了用户想用的别名。当出现这种情况时，群聊服务将使用一条出席错误提示来响应定向出席，以指出别名冲突。

Lydia 试着加入她妹妹的房间，但别人已经使用了她想用的别名。

```

<presence to='ball@chat.meryton.lit/lydia'
          from='lydia@longbourn.lit/meryton'>
<x xmlns='http://jabber.org/protocol/muc' />
</presence>

```

服务器响应别名冲突错误。

```

<presence to='lydia@longbourn.lit/meryton'
          from='ball@chat.meryton.lit'
          type='error'>
<x xmlns='http://jabber.org/protocol/muc' />
<error type='cancel'>
<conflict xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
</error>
</presence>

```

Lydia 必须选择一个新的别名并试着再次加入该房间。

离开房间是通过向房间 JID 发送不可访问出席信息来实现的。Darcy 厌倦了房间中的交谈，于是决定离开。他发送如下 XMPP 节。

```

<presence to='ball@chat.meryton.lit/darcy'
          from='darcy@pemberley.lit/meryton'
          type='unavailable' />

```

服务器将这条信息广播到其他房客。例如，Jane 的客户端就会接收到下面这条标记 Darcy 离开的 XMPP 节。

```

<presence to='jane@longbourn.lit/meryton'
          from='ball@chat.meryton.lit/darcy'
          type='unavailable'>
  <x xmlns='http://jabber.org/protocol/muc'>
    <item affiliation='member' role='none' />
  </x>
</presence>

```

Darcy 也会从该房间接收到该广播消息。注意，该消息还携带了状态码 110，这是因为该消息引用的 Darcy 自己的出席信息。

```

<presence to='darcy@pemberley.lit/meryton'
          from='ball@chat.meryton.lit/darcy'
          type='unavailable'>
  <x xmlns='http://jabber.org/protocol/muc'>
    <item affiliation='member' role='none' />
    <status code='110' />
  </x>
</presence>

```

除了规范中定义的更多错误情况之外，这就是加入和离开群聊房间涉及的全部内容。

8.3.3 发送和接收消息

一旦进入房间，用户就可以通过发送和接收消息彼此进行通信。这与个人聊天消息(我们已经在前几章中看到)非常类似。

发送到房间的消息有一个特殊的类型 `groupchat`。定向到房间裸 JID(比如 `ball@chat.meryton.lit`)的消息将广播给所有的房客。发送给房间中某个房客的完整 JID 的消息属于个人消息，它们会通过房间转发给该用户的真实 JID。

在下面的示例中，Bingley 显示寻址到房间中的 Darcy。首先，Bingley 向该房间发送自己的消息。

```

<message to='ball@chat.meryton.lit'
          from='bingley@netherfield.lit/meryton'
          type='groupchat'>
  <body>Come, Darcy, I must have you dance. I hate to see you standing
        about by yourself in this stupid manner. You had much better
        dance.</body>
</message>

```

他的消息将广播给所有的房客，包括 Darcy。Darcy 的客户端将接收到如下内容。

```

<message to='darcy@pemberley.lit/meryton'
          from='ball@chat.meryton.lit/bingley'
          type='groupchat'>
  <body> Come, Darcy, I must have you dance. I hate to see you standing

```

```

    about by yourself in this stupid manner. You had much better
    dance.</body>
</message>

```

在大多数群聊房间中，消息都与此类似，可能携带 XHTML-IM 有效载荷以便更加美观。群聊消息可以进行扩展，就像 XMPP 的其他部分一样，通过使用扩展，消息可以携带任意结构化信息。我们将在第 11 章中使用群聊功能构建一个游戏服务时实际地看到这一点。

8.3.4 匿名性

XMPP 多人聊天室有多级的匿名性，这与其他许多群聊系统不同。规范中目前定义了 3 个等级：不匿名、半匿名和完全匿名。

在不匿名的房间中，每位房客都能够看到其他房客的真实 JID。该房间会在房客的出席更新中广播额外的 `jid` 属性。

在半匿名的房间中，只有所有者和管理员能够看到房客的真实 JID。半匿名和不匿名房间是最常见的类型，大多数群聊服务都配置使用这两种类型之一作为新创建房间的默认值。

完全匿名房间非常稀少，只有服务器管理员能够访问这些房间的房客的真实 JID。甚至该房间的所有者都不能访问真实 JID。

在半匿名和完全匿名的房间中不能向参与者发送普通的个人消息，因为发送者不能访问该参与者的真实 JID。这就是为什么群聊中的个人消息要发送给参与者的房间 JID 的原因所在。

8.3.5 创建房间

虽然 XMPP 联合网络中已经有成千上万房间可供参与，但有时候还是会发现自己正在查找的房间并不存在。创建房间非常简单，它的实现方式大致上与加入房间的方式相同。

实际上，通过加入不存在的房间就可以创建房间。假设服务允许用户创建新房间，向新房间的预期房间 JID 发送定向出席信息就会导致创建该房间，而该用户将被设为该房间的所有者。下面，Bingley 为 Netherfield 聚会创建了一个新房间：

```

<presence to='chatter@chat.netherfield.lit/bingley'
          from='bingley@netherfield.lit/drawing_room'>
  <x xmlns='http://jabber.org/protocol/muc' />
</presence>

```

chat.netherfield.lit 服务进行响应，向该房间新的唯一的房客发送出席广播。

```

<presence to='bingley@netherfield.lit/drawing_room'
          from='chatter@chat.netherfield.lit/bingley'>
  <x xmlns='http://jabber.org/protocol/muc'>
    <item affiliation='owner' role='moderator' />
    <status code='110' />
    <status code='201' />
  </x>
</presence>

```

注意, Bingley 已经拥有所有者(owner)岗位和主持人(moderator)角色。这些属性将让 Bingley 在该房间中拥有特殊的权力, 稍后我们将会更多地看到这些。这里发送了 110 状态码(这与前面一样), 此外还发送了新的状态码 201。这个新的状态码指示新房间已经建立完毕。

一旦房间建立完毕, 所有者通常就可以按照预期对其进行配置。群聊房间支持许多配置选项, 包括:

- 房间持久性, 当所有参与者全部离开后该房间是否仍然继续存在
- 房间描述
- 该房间的消息是否记录日志
- 是否允许参与者更换该房间的主题
- 最大房客人数
- 类似成员列表一样的访问控制

房间配置是通过 Data Forms(XEP-0004)来完成的, 我们将在第 9 章中学习它。Groupie 的需求很少, 因此默认的房间配置就足够了。

8.3.6 理解角色和岗位

我们已经看到群聊房间中的每位用户都被指派一个角色和一个岗位。房客通常都有一个参与者(participant)角色和一个成员(member)岗位, 但我们在房间创建过程中曾经看到过, 房间的创建者的角色为主持人(moderator), 岗位为所有者(owner)。我们要更多地学习角色和岗位, 这是因为它们对于聊天室的最关键部分之一社区管理而言非常重要。

角色和岗位都用来允许或限制功能, 但它们用于不同的时间范畴。岗位是一种长期的性质, 它会保存起来跨越对该房间的多次访问, 而角色只用于当前访问。例如, 当房间的所有者加入房间时, 他们的角色是 moderator, 而当他们离开时, 他们的角色变成了 none, 但即使在他们离开后, 他们的岗位仍然为 owner。

角色和岗位大多数为层次结构, 每一等级都具有前一等级的所有性质, 此外还增加了一些新的性质。表 8-1 列出了已定义的角色以及他们的普通含义, 表 8-2 列出了所有可能的岗位。

表 8-1 群聊角色

角 色	特 权
无	没有权限, 不在房间中
游客	可以查看对话, 但不能交谈
参与者	可以完全参与公开对话
主持人	可以将用户从房间中踢出, 或者将参与者降为游客

表 8-2 群聊岗位

被 排 斥 者	禁入该房间
无	可以加入该房间
成员	可以加入该房间, 即使该房间只有成员可加入, 并且可以获取成员列表
管理员	可以禁止成员或非岗位用户, 可以添加和移除成员岗位或主持人角色
所有者	可以添加和移除管理员和所有者, 可以配置或销毁房间

利用表 8-1 和表 8-2 中的信息，我们可以开始理解如何进行社区管理。“踢出”粗暴的参与者是通过将他们在房间中的角色设为无(none)来实现的。如果他们回来后继续制造麻烦，那么可以将他们的房间岗位设为被排斥者(outcast)，这样就不再允许他们加入该房间。房间的所有者能够通过将某些参与者的岗位设置为管理员(admin)，从而将他们招募为新的房间管理员，每当他们加入该房间时，就会把他们的角色修改为 moderator。

角色和岗位的操作是通过 IQ-set 和 IQ-get 节来完成的。修改参与者的角色或岗位通常会导致针对受影响用户向该房间广播新的出席信息。

Wickham 撤下 militia 聊天室与 Lydia 走了。Forster 不得不禁止 Wickham 进入 militia 的聊天室。Forster 发送一个 IQ-set 节来更改 Wickham 的岗位。

```
<iq to='militia@chat.emdashshire.lit'
    from='forster@militia.lit/headquarters'
    type='set'
    id='ban1'>
  <query xmlns='http://jabber.org/protocol/muc#admin'>
    <item jid='wickham@emdashshire.lit'
          affiliation='outcast' />
    <reason>AWOL</reason>
  </query>
</iq>
```

chat.emdashshire.lit 服务响应一个 IQ-result 节，指示操作成功(Forster 是该房间的所有者)。如果 Wickham 当前不在房间中，那么该房间就不必向其他的房客广播任何信息；但如果 Wickham 被禁止进入时正好在房间中，那么他将被强制离开，而且所有的房间房客都会得到他的出席状态变化通知。

当 Wickham 在驱逐之后被迫离开该房间时，他将接收到如下 XMPP 节。

```
<presence to='wickham@emdashshire.lit/london'
          from='militia@chat.emdashshire.lit'
          type='unavailable'>
  <x xmlns='http://jabber.org/protocol/muc#user'>
    <item affiliation='outcast' role='none'>
      <actor jid='forster@militia.lit' />
      <reason>AWOL</reason>
    </item>
    <status code='301' />
  </x>
</presence>
```

<actor>元素将让 Wickham 知道谁造成了岗位修改，而状态码 301 指示他已经被禁止入内。其他的 militia 成员将接收到类似的出席节。

角色变化也是通过与岗位相同的方式进行的。如果 Catherine 希望向 Elizabeth 授予她在自己房间中的发言权，那么她将发送一个携带预期角色变化的 IQ-set 节。

```

<iq to='chatter@chat.rosings.lit'
    from='lady_catherine@rosings.lit/parlor'
    type='set'
    id='voicel'>
  <x xmlns='http://jabber.org/protocol/muc#admin'>
    <item nick='elizabeth' role='participant' />
  </x>
</iq>

```

Elizabeth 的角色从游客(visitor)变成了参与者(participant)，而该房间将向所有房客发送出席状态变化信息。注意，nick 属性用来指定 Elizabeth 的别名。由于角色适用于单次房间访问，因此更改角色是通过别名完成的而不是 JID。

可以看到，MUC 是一个相当有深度的协议。该扩展的规范中还列举了其他的用例、配置选项以及错误流，但本节讲解的主题对于构建 Groupie 来说已经足够了。

8.4 构建界面

Groupie 的界面非常简单，我们从图 8-1 中可以看出。它由一个稍微经过扩展的登录对话框、一个聊天区域、一个参与者区域以及聊天输入框组成。程序清单 8-1 给出了构建这个界面所需的 HTML，而程序清单 8-2 给出了它使用的 CSS 样式。



可从
Wrox.com
下载源代码

程序清单 8-1 groupie.htm

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=UTF-8">
    <title>Groupie - Chapter 8</title>

    <link rel='stylesheet' href='http://ajax.googleapis.com/ajax/libs/jqueryui
    /1.7.2/themes/cupertino/jquery-ui.css'>
    <script src='http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/jquery.js'>
    </script>
    <script src='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2
    /jquery-ui.js'></script>
    <script src='scripts/strophe.js'></script>
    <script src='scripts/flXHR.js'></script>
    <script src='scripts/strophe.flxhr.js'></script>

    <link rel='stylesheet' type='text/css' href='groupie.css'>
    <script type='text/javascript' src='groupie.js'></script>
  </head>
  <body>
    <h1>Groupie</h1>

```

```

<div id='toolbar'>
  <input id='leave' type='button' value='Leave Room'
         disabled='disabled'>
</div>

<div>
  <div>
    <span id='room-name'></span>
    <span id='room-topic'></span>
  </div>
  <div id='chat'>
  </div>

  <textarea id='input'></textarea>
</div>

<div id='participants'>
  <ul id='participant-list'>
  </ul>
</div>

<!-- login dialog -->
<div id='login_dialog' class='hidden'>
  <label>JID:</label><input type='text' id='jid'>
  <label>Password:</label><input type='password' id='password'>
  <label>Chat Room:</label><input type='text' id='room'>
  <label>Nickname:</label><input type='text' id='nickname'>
</div>
</body>
</html>

```



可从
Wrox.com
下载源代码

程序清单 8-2 groupie.css

```

body {
  font-family: Helvetica;
}

h1 {
  text-align: center;
}

.hidden {
  display: none
}

#toolbar {
  text-align: center;
  padding: 5px;
  margin-bottom: 15px;
}

#room-name {

```

```
        font-size: 150%;  
        font-weight: bold;  
    }  
  
    #room-topic {  
        font-style: italic;  
    }  
  
    #chat-area {  
        float: left;  
        width: 500px;  
    }  
  
    #chat {  
        overflow: auto;  
        height: 400px;  
        border: solid 1px #ccc;  
    }  
  
    #participants {  
        float: left;  
        width: 200px;  
        height: 500px;  
        overflow: auto;  
    }  
  
    #input {  
        width: 95%;  
        font-size: 120%;  
    }  
  
.notice {  
    font-style: italic;  
    font-weight: bold;  
    color: #3d3;  
}  
  
.error {  
    color: #d33;  
}  
  
.message {  
    color: #aaa;  
}  
  
.nick {  
    color: #66f;  
}  
  
.self {  
    color: #f66;  
}  
  
.body {
```

```

        color: #000;
    }

.delayed {
    opacity: 0.5;
}

.private {
    background-color: #fdd;
}

.action {
    color: #333;
}

```

我们希望应用程序一打开就显示登录对话框，就像前几章中一样。程序清单 8-3 中的 JavaScript 代码骨架已经经过稍微的修改，在登录对话框中增加了一些新字段，但其余部分与我们前面看到过的一样。



程序清单 8-3 groupie.js(骨架)

可从
Wrox.com
下载源代码

```

var Groupie = {
    connection: null,
    room: null,
    nickname: null
};

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Join a Room',
        buttons: {
            "Join": function () {
                Groupie.room = $('#room').val();
                Groupie.nickname = $('#nickname').val();

                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val()
                });

                $('#password').val('');
                $(this).dialog('close');
            }
        }
    });
});

$(document).bind('connect', function (ev, data) {

```

```

Groupie.connection = new Strophe.Connection(
  'http://bosh.metajack.im:5280/xmpp-httpbind');
Groupie.connection.connect(
  data.jid, data.password,
  function (status) {
    if (status === Strophe.Status.CONNECTED) {
      $(document).trigger('connected');
    } else if (status === Strophe.Status.DISCONNECTED) {
      $(document).trigger('disconnected');
    }
  });
});

$(document).bind('connected', function () {
  // nothing here yet
});

$(document).bind('disconnected', function () {
  // nothing here yet
});

```

在完成样板代码之后，就可以开始添加对加入群聊房间的支持。

8.5 加入房间

在第 8.3 节中我们了解到，加入群聊房间是一个简单的事情，只需向预期的房间 JID 发送定向出席即可。但当您加入该房间时，它会立即向您发送有关所有其他参与者的出席广播，因此必须处理这些信息。还需要添加对离开房间(万一用户希望加入不同的房间)的支持。

首先，应该向 `Groupie` 对象中添加一个命名空间常量，这样就不必重复地输入 MUC 命名空间 `http://jabber.org/protocol/muc`。此外，还需要用一个标记来指示用户是否已经加入某个房间。最后，还需要一个表示房间房客的字典对象。将下面的属性添加到 `Groupie` 对象中。



可从
Wrox.com
下载源代码

```

NS_MUC: "http://jabber.org/protocol/muc",
joined: null,
participants: null

```

code snippet groupie.js

接下来，修改 `connected` 事件处理程序，向服务器发送初始出席信息(以便上线)并向用户希望加入的房间 JID 发送定向出席(以加入该房间)。



可从
Wrox.com
下载源代码

```

$(document).bind('connected', function () {
  Groupie.connection.send($pres().c('priority').t('-1'));
  Groupie.connection.send(
    $pres({
      to: Groupie.room + "/" + Groupie.nickname
    })
  );
});

```

```
});.c('x', {xmlns: Groupie.NS_MUC}));  
});
```

code snippet groupie.js

定向出席节看上去跟我们前面看过的出席节相似，但初始出席有一些新信息，即一个优先级子节点。

出席优先级为用户的服务器提供信息来说明哪些已连接资源对于消息传送更加重要。消息将被路由给那些具有最高的正的出席优先级的资源，而如果出现并列优先级的情况，那么消息将被传送给这些具有并列优先级的资源中的一个。

Groupie 中的负的出席优先级有着特殊的含义。任何具有负优先级的资源都不会接收到寻址到裸 JID 的消息。这对于不打算使用个人消息的 XMPP 应用程序来说非常有用，这是因为负优先级将确保应用程序不会传送它不会处理的消息。它还确保用户不会丢失发送给那些不能进行个人聊天的资源的个人消息。

因为 Groupie 不处理个人消息(除了我们后面会看到的一些特殊的群聊个人消息之外)，所以它将出席优先级设为-1，以避免意外地从用户那里接收到个人消息。

既然 Groupie 启动了加入群聊房间的进程，那么我们应该添加对处理其他参与者出席广播以及自己的出席广播(指示房间加入过程结束)的支持。

将下面的被突出显示的代码行添加到 connected 事件处理程序中，以设置传入的出席的处理程序。



可从
Wrox.com
下载源代码

```
$document.bind('connected', function () {  
    Groupie.joined = false;  
    Groupie.participants = {};  
  
    Groupie.connection.send($pres().c('priority').t('-1'));  
  
    Groupie.connection.addHandler(Groupie.on_presence,  
        null, "presence");  
  
    Groupie.connection.send(  
        $pres({  
            to: Groupie.room + "/" + Groupie.nickname  
        }).c('x', {xmlns: Groupie.NS_MUC}));  
});
```

code snippet groupie.js

现在必须实现 on_presence()。这个处理程序在这个阶段需要完成两件事。首先，它必须利用该房间的当前的房客信息来生成参与者区域。其次，当用户自己的出席信息出现时，它将触发 room_joined 事件。下面给出了修改后的 Groupie 对象。



可从
Wrox.com
下载源代码

```
var Groupie = {  
    connection: null,  
    room: null,
```

```

nickname: null,
NS_MUC: "http://jabber.org/protocol/muc",
joined: null,
participants: null,
on_presence: function (presence) {
    var from = $(presence).attr('from');
    var room = Strophe.getBareJidFromJid(from);

    // make sure this presence is for the right room
    if (room === Groupie.room) {
        var nick = Strophe.getResourceFromJid(from);

        if ($(presence).attr('type') === 'error' &&
            !Groupie.joined) {
            // error joining room; reset app
            Groupie.connection.disconnect();
        } else if (!Groupie.participants[nick] &&
            $(presence).attr('type') !== 'unavailable') {
            // add to participant list
            Groupie.participants[nick] = true;
            $('#participant-list').append('<li>' + nick + '</li>');
        }
        if ($(presence).attr('type') !== 'error' &&
            !Groupie.joined) {
            // check for status 110 to see if it's our own presence
            if ($(presence).find("status[code='110']").length > 0) {
                // check if server changed our nick
                if ($(presence).find("status[code='210']").length > 0) {
                    Groupie.nickname = Strophe.getResourceFromJid(from);
                }
                // room join complete
                $(document).trigger("room_joined");
            }
        }
    }
    return true;
}
};


```

code snippet groupie.js

该代码记录是否加入该房间，以便知道什么时候引发 room_joined 事件。只有当首次接收到用户自己的出席信息(由状态码 110 指示)时，它才应该引发 room_joined 事件。如果出现出席错误，那么通常说明出现别名冲突。在这种情况下 Groupie 简单地重置应用程序。注意，群聊服务可能会更改用户请求的别名，这种修改会通过状态码210指出，因此在这种情况下，nickname 属性会被更新以反映该变化。

可以为 `room_joined` 事件添加一个处理程序，让用户知道他们的操作已经成功执行，并启用 `Leave Room` 按钮。将下面的处理程序添加到 `groupie.js` 文件末尾。



可从
Wrox.com
下载源代码

```
$ (document).bind('room_joined', function () {
    Groupie.joined = true;
    $('#leave').removeAttr('disabled');
    $('#room-name').text(Groupie.room);

    $('#chat').append("<div class='notice'>*** Room joined.</div>");
});
```

code snippet groupie.js

剩下的工作就是将 `Leave Room` 按钮连接起来以便让用户能够离开该房间。将下面的代码添加到文档准备就绪事件处理程序中。



可从
Wrox.com
下载源代码

```
$('#leave').click(function () {
    Groupie.connection.send(
        $pres({to: Groupie.room + "/" + Groupie.nickname,
               type: "unavailable"}));
    Groupie.connection.disconnect();
});
```

code snippet groupie.js

上面的代码向该房间发送类型为 `unavailable` 的定向出席信息以友好地退出，然后终止 XMPP 连接。修改 `disconnected` 事件处理程序，以确保登录对话框再次出现。



可从
Wrox.com
下载源代码

```
$ (document).bind('disconnected', function () {
    Groupie.connection = null;
    $('#participant-list').empty();
    $('#room-name').empty();
    $('#room-topic').empty();
    $('#chat').empty();
    $('#login_dialog').dialog('open');
});
```

code snippet groupie.js

8.6 处理出席和消息

一旦用户进入群聊房间，他就需要能够发送和接收消息，而当人们加入和离开该房间时，用户的参与者列表应该保持更新。发送给用户的个人消息应该以一种特别的方式显示，而且用户还应该能够向其他参与者发送个人消息。我们可以轻易地向 `Groupie` 中添加这些功能。

8.6.1 处理房间消息

为了显示来自房间中其他人的消息，必须首先添加一个处理程序来处理传入的<message>节，然后显示它们。可以在 connected 事件处理程序中(紧随<presence>节的 addHandler()函数之后)添加适当的处理程序。



可从
Wrox.com
下载源代码

```
Groupie.connection.addHandler(Groupie.on_public_message,
    null, "message", "groupchat");
```

code snippet groupie.js

注意，发送到房间的每条消息的 type 属性均被设为 groupchat。这个属性值将公开消息与个人消息区分开来。后面我们还将添加一个用于处理个人消息的处理程序，这些消息的 type 属性值等于 chat 或 normal。

现在必须实现 on_public_message()，它应该显示寻址到该房间的传入消息。将下面的代码添加到 Groupie 对象中。



可从
Wrox.com
下载源代码

```
on_public_message: function (message) {
    var from = $(message).attr('from');
    var room = Strophe.getBareJidFromJid(from);
    var nick = Strophe.getResourceFromJid(from);

    // make sure message is from the right place
    if (room === Groupie.room) {
        // is message from a user or the room itself?
        var notice = !nick;

        // messages from ourself will be styled differently
        var nick_class = "nick";
        if (nick === Groupie.nickname) {
            nick_class += " self";
        }

        var body = $(message).children('body').text();

        if (!notice) {
            Groupie.add_message("<div class='message'>" +
                "&lt;span class='" + nick_class + "'>" +
                nick + "</span>&gt; <span class='body'>" +
                body + "</span></div>");
        } else {
            Groupie.add_message("<div class='notice'>*** " + body +
                "</div>");
        }
    }
}
```

```

        return true;
    }

```

code snippet groupie.js

这个函数要比它看起来的稍微简单一些。首先，这个函数分辨出房间的房客发送的消息与房间自身发送的消息，并适当地打印消息。它还使用 CSS 样式来区分用户发送的消息与其他房客发送的消息。

最后，与前面的应用程序一样，该函数确保当使用 `add_message()` 函数添加新行时，内容窗口将滚到底部。下面给出了辅助函数 `add_message()`，应该将其添加到 `Groupie` 对象中。



可从
Wrox.com
下载源代码

```

add_message: function (msg) {
    // detect if we are scrolled all the way down
    var chat = $('#chat').get(0);
    var at_bottom = chat.scrollTop >= chat.scrollHeight -
        chat.clientHeight;

    $('#chat').append(msg);

    // if we were at the bottom, keep us at the bottom
    if (at_bottom) {
        chat.scrollTop = chat.scrollHeight;
    }
}

```

code snippet groupie.js

接下来我们要把文本输入框连接起来，这样用户就可以向房间发送消息。我们的实现方式是添加一个 `keypress` 事件处理程序，并观察对应于 `Enter` 键的按键编码。将下面的代码添加到文档准备就绪事件处理程序中。



可从
Wrox.com
下载源代码

```

$('#input').keypress(function (ev) {
    if (ev.which === 13) {
        ev.preventDefault();

        var body = $(this).val();

        Groupie.connection.send(
            $msg({
                to: Groupie.room,
                type: "groupchat"}).c('body').t(body));

        $(this).val('');
    }
});

```

code snippet groupie.js

添加这个最新的功能之后，Groupie 现在已经有了一定的用处了。用户可以加入群聊房间并参与到对话中。但是参与者列表现在还没有跟踪房间的真实状态，下面我们就来解决这个问题。

8.6.2 跟踪出席状态变化

我们对前面编写的 `on_presence()` 处理程序做了大量工作，每当新人加入房间时，它都会更新参与者列表，但当人们离开时，它并没有保持更新。我们需要向 `on_presence()` 中添加逻辑，当某个人离开房间时会将这个人从列表中移除。

下面的代码给出了修改后的 `on_presence()` 函数，其中新加的代码和修改过的代码均被突出显示。注意，我们分别为人们加入房间和离开房间添加了事件。我们稍后将使用这些事件。



```

on_presence: function (presence) {
    var from = $(presence).attr('from');
    var room = Strophe.getBareJidFromJid(from);

    // make sure this presence is for the right room
    if (room === Groupie.room) {
        var nick = Strophe.getResourceFromJid(from);

        if ($(presence).attr('type') === 'error' &&
            !Groupie.joined) {
            // error joining room; reset app
            Groupie.connection.disconnect();
        } else if (!Groupie.participants[nick] &&
            $(presence).attr('type') !== 'unavailable') {
            // add to participant list
            Groupie.participants[nick] = true;
            $('#participant-list').append('<li>' + nick + '</li>');

            if (Groupie.joined) {
                $(document).trigger('user_joined', nick);
            }
        } else if (Groupie.participants[nick] &&
            $(presence).attr('type') === 'unavailable') {
            // remove from participants list
            $('#participant-list li').each(function () {
                if (nick === $(this).text()) {
                    $(this).remove();
                    return false;
                }
            });
            $(document).trigger('user_left', nick);
        }
    }

    if ($(presence).attr('type') !== 'error' &&
        !Groupie.joined) {
        // check for status 110 to see if it's our own presence
        if ($(presence).find("status[code='110']").length > 0) {
            // check if server changed our nick
        }
    }
}

```

```

        if ($(presence).find("status[code='210']").length > 0) {
            Groupie.nickname = Strophe.getResourceFromJid(from);
        }

        // room join complete
        $(document).trigger("room_joined");
    }
}

return true;
}

```

code snippet groupie.js

通过针对这两个新事件编写处理程序，当有人加入和离开房间时就可以向用户打印有用的消息。下面的事件处理程序向聊天区域添加有关这些事件的提示消息，将这些代码添加到 groupie.js 文件的末尾。



可从
Wrox.com
下载源代码

```

$(document).bind('user_joined', function (ev, nick) {
    Groupie.add_message("<div class='notice'>*** " + nick +
        " joined.</div>");
});

$(document).bind('user_left', function (ev, nick) {
    Groupie.add_message("<div class='notice'>*** " + nick +
        " left.</div>");
});

```

code snippet groupie.js

现在 Groupie 可以记录谁在房间中以及人们何时加入和离开。

8.6.3 聊天历史

读者可能已经注意到，通常一旦加入群聊房间，立即就会出现大量的消息。这些消息表示最近的聊天历史，XMPP MUC 房间会向新加入的房客发送一定数量(可以配置)的历史消息，这样他们就对接下来要讨论的内容获取了一定的背景信息。如果曾经使用过 IRC，那么您可能比较熟悉这种情况：当中途加入到某个活跃的对话中来时，您会感到非常困惑。

Groupie 最好能够将这种聊天历史消息按照不同的方式显示。我们可以修改 `on_public_message()` 处理程序，向这些消息中添加一个特殊的 CSS 类。房间使用`<delay>`元素为历史聊天消息带上特殊标记，就像我们在第 5 章中讲到的存储的个人消息一样。有可能会遇到群聊服务器使用老版本的延迟指示(在 XEP-0091 中定义)，而不是 XEP-0203 中定义的最新版本。为了探测旧式延迟指示器，只需要检查是否存在 `jabber:x:delay` 命名空间下面的`<x>`元素。下面是修改后的 `on_public_message()` 函数，将 CSS 类 `delayed` 添加到那些包含延迟提示(两种类型之一)的消息中。

```

on_public_message: function (message) {
    var from = $(message).attr('from');
    var room = Strophe.getBareJidFromJid(from);
    var nick = Strophe.getResourceFromJid(from);

    // make sure message is from the right place
    if (room === Groupie.room) {
        // is message from a user or the room itself?
        var notice = !nick;

        // messages from ourself will be styled differently
        var nick_class = "nick";
        if (nick === Groupie.nickname) {
            nick_class += " self";
        }

        var body = $(message).children('body').text();

        var delayed = $(message).children("delay").length > 0 ||
            $(message).children("x[xmlns='jabber:x:delay']").length > 0;

        if (!notice) {
            var delay_css = delayed ? " delayed": "";
            Groupie.add_message("<div class='message" + delay_css + "'>" +
                "&lt;<span class=' + nick_class + '>" + nick + "</span>&gt; <span class='body'>" +
                body + "</span></div>");
        } else {
            Groupie.add_message("<div class='notice'>*** " + body +
                "</div>");
        }
    }
    return true;
}

```

code snippet groupie.js

在处理完聊天历史之后，现在我们要继续处理最后一个与通信相关的功能，即个人房间消息。

8.6.4 保持私密性

当发送那些给所有的人看的消息时，Groupie 将消息寻址到房间本身，该房间将该消息重新广播到所有的房客。我们还可以将消息寻址到特定的房客，这些消息将通过私密方式发送，不会在房间内的其他房客中共享。为了完成 Groupie，我们还希望添加对在房间内发送和接收个人消息的支持。

为了发送个人房间消息，只需要将该消息直接寻址到这个房客的房间 JID，确保将 type 属性设为 `chat` 而不是 `groupchat`。我们在第 8.3.4 节中曾经讨论过，我们使用的是参与者的房间 JID，这是因为该房间可能被配置为匿名的或半匿名的。

如果没有不同的 type 属性值，那么接收者就没有办法区分公共消息和个人消息。为了接收个人消息，我们必须观察来自房间房客的、type 属性值为 chat 的传入消息。

首先，在 connected 事件处理程序中(在其他处理程序旁边)添加一个新的<message>节处理程序来处理个人消息。



可从
Wrox.com
下载源代码

```
Groupie.connection.addHandler(Groupie.on_private_message,
    null, "message", "chat");
```

code snippet groupie.js

接下来，将 on_private_message()函数添加到 Groupie 对象中。



可从
Wrox.com
下载源代码

```
on_private_message: function (message) {
    var from = $(message).attr('from');
    var room = Strophe.getBareJidFromJid(from);
    var nick = Strophe.getResourceFromJid(from);

    // make sure this message is from the correct room
    if (room === Groupie.room) {
        var body = $(message).children('body').text();
        Groupie.add_message("<div class='message private'>" +
            "@@ <span class='nick'>" +
            nick + "</span>&gt; <span class='body'>" +
            body + "</span> @@</div>");

    }
    return true;
}
```

code snippet groupie.js

Groupie 在将消息显示给用户之前，确保它来自该房间内的另一个参与者。

最后，需要修改文本输入框的 keypress 事件处理程序以探测个人消息并将其发送。个人消息使用/msg 命令发送，用户会在文本框中输入该命令。如果以前曾经使用过 IRC 客户端，那么应该非常熟悉这些。新的处理程序代码必须能够分析这种特殊的命令，检查是否有错误或非法实参，然后执行预期的动作。

下面给出了 keypress 处理程序，其中修改过的代码行已被突出显示。



可从
Wrox.com
下载源代码

```
$('#input').keypress(function (ev) {
    if (ev.which === 13) {
        ev.preventDefault();

        var body = $(this).val();

        var match = body.match(/^\/(.*)?:(.*)?$/);
        var args = null;
```

```

if (match) {
    if (match[1] === "msg") {
        args = match[2].match(/^(.*) (.*)$/);
        if (Groupie.participants[args[1]]) {
            Groupie.connection.send(
                $msg({
                    to: Groupie.room + "/" + args[1],
                    type: "chat"}).c('body').t(body));
            Groupie.add_message(
                "<div class='message private'>" +
                "@@ &lt;span class='nick self'>" +
                Groupie.nickname +
                "</span>&gt; <span class='body'>" +
                args[2] + "</span> @@</div>");
        } else {
            Groupie.add_message(
                "<div class='notice error'>" +
                "Error: User not in room." +
                "</div>");
        }
    } else {
        Groupie.add_message(
            "<div class='notice error'>" +
            "Error: Command not recognized." +
            "</div>");
    }
} else {
    Groupie.connection.send(
        $msg({
            to: Groupie.room,
            type: "groupchat"}).c('body').t(body));
}

$(this).val('');
});
});

```

code snippet groupie.js

上面的代码将命令分析逻辑抽离出来,这是因为我们在第 8.6.5 节中还将添加更多的特殊命令。对于/msg 命令, Groupie 将个人消息发送给用户指定的房客。如果所指的房客并不在房间中或指定的是非法命令,那么会显示一条错误提示信息。

个人消息现在显示在“@@”标记之间而且拥有淡红色背景。添加对一个命令的支持并不太难,而额外的命令也可以轻易实现。作为本节的最后一个功能,我们应该添加一个 action 命令。

8.6.5 描述动作

许多聊天系统支持动作描述,用户描述自己或他们正在做什么事情。因为以第三人称谈论

自己通常很奇怪，所以这些聊天系统增加了一些命令，让这个操作变得更加自然一些。通常，这些命令写作/**action** 或/**me**，而用户输入类似下面的信息。

/me writes another XMPP application

我们可以轻易地向 Groupie 中添加对这些命令的支持。我们只需要在 keypress 事件处理程序中的命令逻辑中添加一个新的子句，并以一种特殊方式来显示以/me 开头的消息。

下面的代码给出了命令解释子句中所需的修改。



```

if (match[1] === "msg") {
    args = match[2].match(/^(.*) (.*)$/);
    if (Groupie.participants[args[1]]) {
        Groupie.connection.send(
            $msg({
                to: Groupie.room + "/" + args[1],
                type: "chat"}).c('body').t(body));
        Groupie.add_message(
            "<div class='message private'>" +
            "@@ <span class='nick self'>" +
            Groupie.nickname +
            "</span>&gt; <span class='body'>" +
            args[2] + "</span> @@</div>");
    } else {
        Groupie.add_message(
            "<div class='notice error'>" +
            "Error: User not in room." +
            "</div>");
    }
} else if (match[1] === "me" || match[1] === "action") {
    Groupie.connection.send(
        $msg({
            to: Groupie.room,
            type: "groupchat"}).c('body')
            .t('/me ' + match[2]));
} else {
    Groupie.add_message(
        "<div class='notice error'>" +
        "Error: Command not recognized." +
        "</div>");
}

```

code snippet groupie.js

最后，修改 **on_public_message()** 的相关部分来匹配修改后的部分。



```

if (!notice) {
    var delay_css = delayed ? " delayed": "";
    var action = body.match(/\b/me (.*)$/);
    if (!action) {

```

```

Groupie.add_message(
    "<div class='message' + delay_css + "'>" +
    "&lt;<span class='nick' + nick_class + "'>" +
    nick + "</span>&gt; <span class='body'>" +
    body + "</span></div>");

} else {
    Groupie.add_message(
        "<div class='message action' + delay_css + "'>" +
        "* " + nick + " " + action[1] + "</div>");

}
else {
    Groupie.add_message("<div class='notice'>*** " + body +
        "</div>");

}

```

code snippet groupie.js

Groupie 接近完工，而且它已经是一个相当有用的群聊客户端。但如果您是一个房间主持人，那么还需要一些额外的功能来管理房客。

8.7 管理房间

我们将要向 Groupie 中添加的最后的一些功能全部与房间管理有关。这些是用户成功运行自己的群聊房间所需的工具，包括操作房间的主题、踢出和禁止用户以及管理房间的管理员。

8.7.1 更换主题

每个房间都有一个主题，这通常显示在房间的顶部。根据房间的配置情况，用户或许能够修改主题消息，或者将这个动作限制为只有管理员可以执行。Groupie 还要显示当前主题并允许授权用户修改它。

主题变化将作为一个无正文消息发送出去，该消息携带一个`<subject>`元素，这个`<subject>`元素的内容将成为新的房间主题。可以在 `on_public_message()` 处理程序中观察这些消息。应该将下面的代码插入到包含“`if (!notice) {`”的代码行之前。



可从 Wrox.com 下载源代码

```

// look for room topic change
var subject = $(message).children('subject').text();
if (subject) {
    $('#room-topic').text(subject);
}

```

code snippet groupie.js

可以向 `keypress` 事件处理程序中添加`/topic` 命令，从而允许用户来更换主题。下面给出了修改后的命令逻辑，新增加的代码行被突出显示。



可从 Wrox.com 下载源代码

```

} else if (match[1] === "me" || match[1] === "action") {
    Groupie.connection.send(
        $msg({

```

```

        to: Groupie.room,
        type: "groupchat"}).c('body')
        .t('/me ' + match[2]));
} else if (match[1] === "topic") {
    Groupie.connection.send(
        $msg({to: Groupie.room,
            type: "groupchat"}).c('subject')
            .text(match[2]));
} else {
    Groupie.add_message(
        "<div class='notice error'>" +
        "Error: Command not recognized." +
        "</div>");
}

```

code snippet groupie.js

Groupie 的用户现在可以看到房间主题并能够修改它(假设该房间的配置允许他们这样做)。

8.7.2 处理麻烦制造者

令人感到悲哀的是，Internet 上的每个聚集场所都会成为一些肆意扰乱和平对话的人的猎物。但 MUC 协议提供了一些工具来处理这些麻烦制造者，我们将会向 Groupie 中添加其中的两个工具：踢出和禁入。

将用户从房间中踢出只是将他们暂时从房间中移出，就像是一次严厉的警告。我们在第 8.3 节中曾经看到过，踢出用户是通过针对一个房客发送一个 IQ-set 节将该房客的角色修改为 none 来完成的。为了让用户踢出某个房客，该用户必须具有主持人角色。

添加/kick 命令，将下面的子句添加到命令逻辑中。



可从
Wrox.com
下载源代码

```

} else if (match[1] === "topic") {
    Groupie.connection.send(
        $msg({to: Groupie.room,
            type: "groupchat"}).c('subject')
            .text(match[2]));
} else if (match[1] === "kick") {
    Groupie.connection.sendIQ(
        $iq({to: Groupie.room,
            type: "set"})
            .c('query', {xmlns: Groupie.NS_MUC + "#admin"})
            .c('item', {nick: match[2],
                role: "none"}));
} else {
    Groupie.add_message(
        "<div class='notice error'>" +
        "Error: Command not recognized." +
        "</div>");
}

```

code snippet groupie.js

可以按照下面的方法测试这项功能：创建一个新房间，这样您就是该房间的所有者，然后用另一个 XMPP 账户加入该房间。一旦加入第二个账户，就可以用第一个账户输入命令/kick nickname 将第二个账户踢出该房间。

禁止某个用户的操作基本上与此相同，但我们必须修改该用户的岗位而不是角色。被禁止的用户的岗位为 outcast，他将不再能够加入该房间，直到该禁令被撤销。

将下面的子句添加到命令逻辑中，以添加对/ban 的支持。



可以从
Wrox.com
下载源代码

```

} else if (match[1] === "kick") {
  Groupie.connection.sendIQ(
    $iq({to: Groupie.room,
      type: "set"})
    .c('query', {xmlns: Groupie.NS_MUC + "#admin"})
    .c('item', {nick: match[2],
      role: "none"}));
} else if (match[1] === "ban") {
  Groupie.connection.sendIQ(
    $iq({to: Groupie.room,
      type: "set"})
    .c('query', {xmlns: Groupie.NS_MUC + "#admin"})
    .c('item', {jid: Groupie.participants[match[2]],
      affiliation: "outcast"}));
} else {
  Groupie.add_message(
    "<div class='notice error'>" +
    "Error: Command not recognized." +
    "</div>");
}

```

code snippet groupie.js

必须使用用户的裸 JID 而不是他们的房间别名来禁止他们。还可以禁止整个域，但通常主持人会尝试限制该禁令以避免无意中惩罚到无辜者。上面的代码使用 participants 字典来获取房客别名对应的 JID。在前面，Groupie 只在这个字典中保存了值 true，我们必须将下面被突出显示的代码行添加到 on_presence() 处理程序中，如果 JID 消息可用就将其保存起来。



可以从
Wrox.com
下载源代码

```

} else if (!Groupie.participants[nick] &&
  $(presence).attr('type') !== 'unavailable') {
  // add to participant list
  var user_jid = $(presence).find('item').attr('jid');
  Groupie.participants[nick] = user_jid || true;
  $('#participant-list').append('<li>' + nick + '</li>');

  if (Groupie.joined) {
    $(document).trigger('user_joined', nick);
  }
} else if (Groupie.participants[nick] &&

```

code snippet groupie.js

Groupie 现在可以帮助房间管理员和所有者维持和平(只要他们中的一个在场)。因为管理员总需要时间休息，所以通常需要招募新的管理员来帮忙。

8.7.3 招募管理员

我们要向 Groupie 中添加的最后一项功能是授权和撤销管理员特权。这是通过/op 和/deop 命令(以它们的 IRC 对等物来命名)来实现的。

添加和移除管理员的实现方式与禁止用户的实现方式非常类似，我们只需要修改某个人对该房间的岗位即可。将下面的被突出显示的子句添加进来以实现新命令。



可从
Wrox.com
下载源代码

```

} else if (match[1] === "ban") {
    Groupie.connection.sendIQ(
        $iq({to: Groupie.room,
            type: "set"})
        .c('query', {xmlns: Groupie.NS_MUC + "#admin"})
        .c('item', {jid: Groupie.participants[match[2]],
            affiliation: "outcast"}));

} else if (match[1] === "op") {
    Groupie.connection.sendIQ(
        $iq({to: Groupie.room,
            type: "set"})
        .c('query', {xmlns: Groupie.NS_MUC + "#admin"})
        .c('item', {jid: Groupie.participants[match[2]],
            affiliation: "admin"}));

} else if (match[1] === "deop") {
    Groupie.connection.sendIQ(
        $iq({to: Groupie.room,
            type: "set"})
        .c('query', {xmlns: Groupie.NS_MUC + "#admin"})
        .c('item', {jid: Groupie.participants[match[2]],
            affiliation: "none"}));

} else {
    Groupie.add_message(
        "<div class='notice error'>" +
        "Error: Command not recognized." +
        "</div>");
}

```

code snippet groupie.js

Groupie 最终完工了，可以提供给普通参与者和主持人使用。程序清单 8-4 给出了最终的 JavaScript 代码。



可从
Wrox.com
下载源代码

程序清单 8-4 groupie.js(最终版本)

```
var Groupie = {
    connection: null,
```

```

room: null,
nickname: null,
NS_MUC: "http://jabber.org/protocol/muc",
joined: null,
participants: null,
on_presence: function (presence) {
  var from = $(presence).attr('from');
  var room = Strophe.getBareJidFromJid(from);

  // make sure this presence is for the right room
  if (room === Groupie.room) {
    var nick = Strophe.getResourceFromJid(from);

    if ($(presence).attr('type') === 'error' &&
        !Groupie.joined) {
      // error joining room; reset app
      Groupie.connection.disconnect();
    } else if (!Groupie.participants[nick] &&
               $(presence).attr('type') !== 'unavailable') {
      // add to participant list
      var user_jid = $(presence).find('item').attr('jid');
      Groupie.participants[nick] = user_jid || true;
      $('#participant-list').append('<li>' + nick + '</li>');

      if (Groupie.joined) {
        $(document).trigger('user_joined', nick);
      }
    } else if (Groupie.participants[nick] &&
               $(presence).attr('type') === 'unavailable') {
      // remove from participants list
      $('#participant-list li').each(function () {
        if (nick === $(this).text()) {
          $(this).remove();
          return false;
        }
      });
      $(document).trigger('user_left', nick);
    }
  }

  if ($(presence).attr('type') !== 'error' &&
      !Groupie.joined) {
    // check for status 110 to see if it's our own presence
    if ($(presence).find("status[code='110']").length > 0) {
      // check if server changed our nick
      if ($(presence).find("status[code='210']").length > 0) {
        Groupie.nickname=Strophe.getResourceFromJid(from);
      }
    }

    // room join complete
    $(document).trigger("room_joined");
  }
}

```

```

        }
    }

    return true;
},
on_public_message: function (message) {
    var from = $(message).attr('from');
    var room = Strophe.getBareJidFromJid(from);
    var nick = Strophe.getResourceFromJid(from);

    // make sure message is from the right place
    if (room === Groupie.room) {
        // is message from a user or the room itself?
        var notice = !nick;

        // messages from ourself will be styled differently
        var nick_class = "nick";
        if (nick === Groupie.nickname) {
            nick_class += " self";
        }

        var body = $(message).children('body').text();

        var delayed = $(message).children("delay").length > 0 ||
            $(message).children("x[@xmlns='jabber:x:delay']").length > 0;

        // look for room topic change
        var subject = $(message).children('subject').text();
        if (subject) {
            $('#room-topic').text(subject);
        }

        if (!notice) {
            var delay_css = delayed ? " delayed": "";
            var action = body.match(/\bme (.*)$/);
            if (!action) {
                Groupie.add_message(
                    "<div class='message" + delay_css + "'>" +
                    "&lt;<span class='nick" + nick_class + "'>" +
                    nick + "</span>&gt; <span class='body'>" +
                    body + "</span></div>");
            } else {
                Groupie.add_message(
                    "<div class='message action " + delay_css + "'>" +
                    "* " + nick + " " + action[1] + "</div>");
            }
        } else {
            Groupie.add_message("<div class='notice'>*** " + body +
                "</div>");
        }
    }
}

```

```

        return true;
    },

    add_message: function (msg) {
        // detect if we are scrolled all the way down
        var chat = $('#chat').get(0);
        var at_bottom = chat.scrollTop >= chat.scrollHeight -
            chat.clientHeight;

        $('#chat').append(msg);

        // if we were at the bottom, keep us at the bottom
        if (at_bottom) {
            chat.scrollTop = chat.scrollHeight;
        }
    },

    on_private_message: function (message) {
        var from = $(message).attr('from');
        var room = Strophe.getBareJidFromJid(from);
        var nick = Strophe.getResourceFromJid(from);

        // make sure this message is from the correct room
        if (room === Groupie.room) {
            var body = $(message).children('body').text();
            Groupie.add_message("<div class='message private'>" +
                "@@ &lt;span class='nick'>" +
                nick + "</span>&gt; <span class='body'>" +
                body + "</span> @@</div>");
        }
        return true;
    }
};

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Join a Room',
        buttons: {
            "Join": function () {
                Groupie.room = $('#room').val();
                Groupie.nickname = $('#nickname').val();

                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val()
                });

                $('#password').val('');
                $(this).dialog('close');
            }
        }
});

```

```

        }
    });
});

$('#leave').click(function () {
    $('#leave').attr('disabled', 'disabled');
    Groupie.connection.send(
        $pres({to: Groupie.room + "/" + Groupie.nickname,
            type: "unavailable"}));
    Groupie.connection.disconnect();
});

$('#input').keypress(function (ev) {
    if (ev.which === 13) {
        ev.preventDefault();

        var body = $(this).val();

        var match = body.match(/^\/(.*)\?:(.*)?$/);
        var args = null;
        if (match) {
            if (match[1] === "msg") {
                args = match[2].match(/^(.*)\?$/);
                if (Groupie.participants[args[1]]) {
                    Groupie.connection.send(
                        $msg({
                            to: Groupie.room + "/" + args[1],
                            type: "chat"}).c('body').t(body));
                    Groupie.add_message(
                        "<div class='message private'>" +
                        "@@ &lt;span class='nick self'>" +
                        Groupie.nickname +
                        "</span>&gt; <span class='body'>" +
                        args[2] + "</span> @@</div>");
                } else {
                    Groupie.add_message(
                        "<div class='notice error'>" +
                        "Error: User not in room." +
                        "</div>");
                }
            } else if (match[1] === "me" || match[1] === "action") {
                Groupie.connection.send(
                    $msg({
                        to: Groupie.room,
                        type: "groupchat"}).c('body')
                        .t('/me ' + match[2]));
            } else if (match[1] === "topic") {
                Groupie.connection.send(
                    $msg({to: Groupie.room,
                        type: "groupchat"}).c('subject')
                        .text(match[2]));
            } else if (match[1] === "kick") {
        }
    }
});

```

```

        Groupie.connection.sendIQ(
            $iq({to: Groupie.room,
                  type: "set"})
            .c('query', {xmlns: Groupie.NS_MUC + "#admin"})
            .c('item', {nick: match[2],
                        role: "none"}));
    } else if (match[1] === "ban") {
        Groupie.connection.sendIQ(
            $iq({to: Groupie.room,
                  type: "set"})
            .c('query', {xmlns: Groupie.NS_MUC + "#admin"})
            .c('item', {jid:Groupie.participants[match[2]],
                        affiliation: "outcast"}));
    } else if (match[1] === "op") {
        Groupie.connection.sendIQ(
            $iq({to: Groupie.room,
                  type: "set"})
            .c('query', {xmlns: Groupie.NS_MUC + "#admin"})
            .c('item', {jid:Groupie.participants[match[2]],
                        affiliation: "admin"}));
    } else if (match[1] === "deop") {
        Groupie.connection.sendIQ(
            $iq({to: Groupie.room,
                  type: "set"})
            .c('query', {xmlns: Groupie.NS_MUC + "#admin"})
            .c('item', {jid:Groupie.participants[match[2]],
                        affiliation: "none"}));
    } else {
        Groupie.add_message(
            "<div class='notice error'>" +
            "Error: Command not recognized." +
            "</div>");
    }
} else {
    Groupie.connection.send(
        $msg({
            to: Groupie.room,
            type: "groupchat"}).c('body').t(body));
}
$(this).val('');
});
});
});

$(document).bind('connect', function (ev, data) {
    Groupie.connection = new Strophe.Connection(
        'http://bosh.metajack.im:5280/xmpp-httpbind');
    Groupie.connection.connect(
        data.jid, data.password,
        function (status) {

```

```

        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });
}

$(document).bind('connected', function () {
    Groupie.joined = false;
    Groupie.participants = {};

    Groupie.connection.send($pres().c('priority').t('-1'));

    Groupie.connection.addHandler(Groupie.on_presence,
        null, "presence");
    Groupie.connection.addHandler(Groupie.on_public_message,
        null, "message", "groupchat");
    Groupie.connection.addHandler(Groupie.on_private_message,
        null, "message", "chat");

    Groupie.connection.send(
        $pres({
            to: Groupie.room + "/" + Groupie.nickname
        }).c('x', {xmlns: Groupie.NS_MUC}));
});

$(document).bind('disconnected', function () {
    Groupie.connection = null;
    $('#room-name').empty();
    $('#room-topic').empty();
    $('#participant-list').empty();
    $('#chat').empty();
    $('#login_dialog').dialog('open');
});

$(document).bind('room_joined', function () {
    Groupie.joined = true;

    $('#leave').removeAttr('disabled');
    $('#room-name').text(Groupie.room);

    Groupie.add_message("<div class='notice'>*** Room joined.</div>");
});

$(document).bind('user_joined', function (ev, nick) {
    Groupie.add_message("<div class='notice'>*** " + nick +
        " joined.</div>");
});
$(document).bind('user_left', function (ev, nick) {
    Groupie.add_message("<div class='notice'>*** " + nick +
        " left.</div>");
});

```

8.8 改进 Groupie

Groupie 是一个简单的群聊客户端，但它还有很大的改进空间。试着添加如下的功能以提供更好的用户体验：

- 添加对同一时刻在多个房间中聊天的支持。
- 让房间所有者为新创建的房间配置一些基本设置。
- 将第 7 章中的浏览器与 Groupie 合并，让用户查找新聊天室并轻易地加入它们。

8.9 小结

XMPP 的 MUC 协议可以让一群人创建一个共享的通信空间。通常，它用于人类用户之间的文本通信，但它还用于许多其他用途。有些人已经使用群聊功能让机器人交换数据。还有些人使用它来创建丰富的协作式环境，或为游戏系统提供基础环境(我们将在第 11 章中看到)。

在本章中，我们创建了一个简单的群聊客户端，它实现了 MUC 功能集合中的一些重要部分。最后我们学习了如下的内容：

- 如何加入、离开以及创建群聊房间
- 如何参与对话
- 如何发送和接收房间内的个人消息
- 如何管理房间的主题
- 角色和岗位的工作原理以及它们被赋予的不同特权
- 如何处理和禁止麻烦制造者
- 如何建立新的房间管理员
- 如何处理聊天历史

在第 9 章中，我们将研究一个名为发布-订阅的类似协议，它主要用于通知和一对多广播系统。

9

第 9 章

发布与订阅：共享画板简介

本章内容

- 发布-订阅系统的工作原理
- 使用数据表单
- 创建和配置 pubsub 节点
- 订阅 pubsub 节点和退订
- 发布和接收事件
- 使用 HTML5 的<canvas>元素

在线聊天虽然是一种很好的团队通信方式，但它在表达可视化想法方面通常并不十分有效。在面对面交流环境中，发言者可以设置一个白板，然后将他们的想法通过绘图方式表达出来。在这一章中，我们将开发一个名为 SketchCast 的应用程序，它允许展示者将白板会话广播给实际数目不受限的多个参与者。我们使用 XMPP 的最强大扩展即发布-订阅(常被称为 pubsub)来开发 SketchCast。

SketchCast 是一个非常简单的矢量绘图程序示例。几乎所有的图形编程或 GUI 编程入门书籍中都会讲解类似的程序。SketchCast 应用程序将这个功能带入共享环境中，让所有观众都能够看到展示者正在绘制的内容。

为了实现这个应用程序，除了简单的绘图工作之外，还需要完成大量的功能。为了让其他人都能够看到展示者正在绘制的内容，我们必须捕获绘图动作，将它们转换成某种适于传送的格式，将这些动作通过网络发送出去，然后在观众的计算机上重建这些动作。此外，展示者还需要某种机制来设置整个系统、记录参与者，并确保所有的人都看到相同的数据。

XMPP 和 pubsub 使得这组复杂的动作能够很容易实现。它负责处理繁重的细节工作，而让我们把注意力只放在少数几个关键部分。

本章展示了 XMPP pubsub 系统的许多核心思想，并介绍了如何处理 XMPP 的 Data Forms，许多 XMPP 扩展都会使用它。

9.1 SketchCast 预览

在开始之前，先看看图 9-1 给出的最终的应用程序的外观。

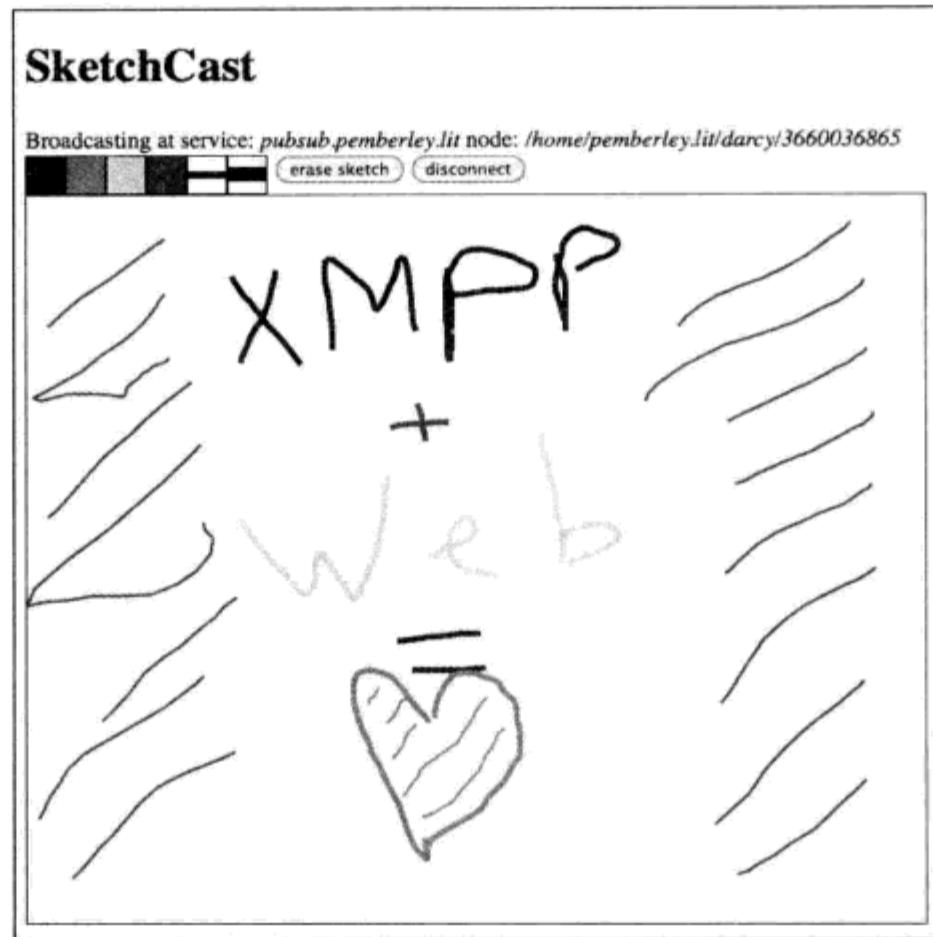


图 9-1

SketchCast 的用户界面非常简单，它包含一个用来放置各种绘图工具的工具栏和一个很大的绘图区域。展示者可以从几种颜色或线宽中挑选，还可以擦除画板。对于观众而言，这些控件会被禁用。

9.2 SketchCast 的设计

SketchCast 具有许多功能，但我们会看到，它实际上是最简单的 XMPP pubsub 应用程序之一。这个应用程序证实了 pubsub 系统强大的功能和巨大的灵活性。下面就让我们来研究 pubsub 都提供了哪些功能。

9.2.1 一切都是 pubsub

发布-订阅系统无处不在，报纸、博客、电视，甚至电子邮件列表。pubsub 的名字已经道出了它的工作原理。该系统中有一个通信通道，订阅者对通过该通道发送出来的数据感兴趣，而发布者可以将数据通过该通道发送出去。

假设我们创办了一份名为《XMPP 爱好者》的杂志。这份杂志相当受欢迎，吸引了成千上万的订阅者，他们将自己的订金和地址告诉我们。当新的一期杂志内容准备就绪后，我们将其印刷并邮寄到每位订阅者的地址。

印刷杂志是份苦差事。想想看，我们还必须花时间收集订金和地址，打印每种单据，将满卡车的杂志运往邮局。如果我们只需关心编写下一期杂志内容，然后将其发给一家负责处理剩余工作的公司，那该多好。黄页上刚好有一家 Pubsub, Inc.公司。

XMPP pubsub 将构建发布-订阅系统过程中的所有手工劳动全部抽离出来(就像 Pubsub, Inc.公司让发行杂志变得更简单一样)。计算机使得计算工作变得相当容易，复杂的应用程序(比如电子表格和 Web 浏览器)都构建在它们上面。同样，XMPP pubsub 也可以让新型应用程序以它为基础进行构建。

在了解 pubsub 系统之后，我们可以学习如何使用它们来支撑起 SketchCast 应用程序。

9.2.2 展示者的流程

SketchCast 应用程序必须为想要成为展示者的人做的第一件事情就是，为他们发布绘图信息建立一个通道。在 XMPP pubsub 中，这些通道被称为节点。

pubsub 应用程序的需求各不相同，因此系统配备了配置工具以提供最大的灵活性。SketchCast 需要对它的 pubsub 节点的默认配置进行一些调整。许多不同的 XMPP 扩展也都有类似的配置系统，因此，以一种标准化的方式来操作这些系统是有意义的。XMPP 的 Data Forms 提供了这种机制，而 SketchCast 将需要使用这个机制来请求和提交配置表单。

一旦创建并配置节点之后，应用程序就需要向该节点发布用户的绘制动作。这些事件一经发布，pubsub 将接管并确保将它们传送给感兴趣的成员。

当展示者正在绘图时，他们希望知道有多少人正在观看。SketchCast 需要从 pubsub 节点那里获取订阅者列表，这样它就能够将该数据展示给这些用户。

一旦完成绘图会话，事后最好通过删除该 pubsub 节点将其清除。

9.2.3 观众的流程

一旦展示者建立节点并准备好开始，那么他们将该节点的存在通告给潜在的观众成员。在这里我们不必担心如何完成这项工作(或许可以将其放在一封公司内部的电子邮件中发送给所有的员工)。

每位观众成员将把该节点输入到 SketchCast 中，该应用程序将订阅绘图用的 pubsub 节点。此后，pubsub 系统将确保任何新发布的数据都将发送给他们。

有些观众成员可能会忙于玩扫雷游戏，并没有注意到展示者的电子邮件，直到他们已经开始绘制了。这些用户需要赶上该组的其他成员，因此 SketchCast 将需要使用 pubsub 来获取他们已经错过的绘制动作。

每一个绘制动作都需要在绘图区域呈现出来，就好像它出现在展示者的屏幕上一样。SketchCast 将把绘制事件翻译成屏幕上的图形，从而重建绘图过程。

最后，一旦绘图完成或观众成员感到疲劳，他们会希望停止观看展示。应用程序将必须从 pubsub 节点中退订，以确保它不会继续获取用户不再感兴趣的事件。

在学习 Data Forms 和 Publish-Subscribe 扩展的过程中要始终记住 SketchCast 的这种设计。一旦学习了这些新的 XMPP 协议，我们就开始构建 SketchCast。

9.3 填写表单

每个应用程序的数据都有所不同。电子表格按照一种格式来组织数据，而方法记录程序则采用完全不同的组织方式。同一个应用程序的每个不同版本也可能有不同的需求。与其为每个应用程序建立新格式，不如建立一种灵活的、能够用于各种不同应用程序的格式。Data Forms 就是 XMPP 工具箱中使用最广泛的这类格式。

我们将需要使用表单来配置 SketchCast 的 pubsub 节点。

9.3.1 Data Forms 扩展

Data Forms 从 HTML 表单中受到一定的启发。它可以让应用程序定义一个含有不同类型的字段(比如文本字段、列表字段、地址字段等)的表单。它还在这些表单之上提供了一种轻量级的工作流，让应用程序能够请求、提供、提交和取消表单。

Pubsub 协议中多处用到表单，包括配置 pubsub 节点以及处理订阅选项。许多 XMPP 协议依靠表单来完成类似的功能。MUC 协议(XEP-0045)使用表单来完成房间配置以及其他设置，Ad-Hoc Commands(XEP-0050)使用表单来完成命令输入和输出，而 Service Discovery Extensions (XEP-0128)依靠表单作为轻易地向其他协议中添加扩展性的方式。

下面给出了一个示例表单。这个表单有一个值为 form 的 type 属性，这意味着该系统希望该表单的内容得到填充。当从某个 XMPP 服务那里请求表单时，就会得到一个这样的表单。如果现在还不理解某些属性的作用，那么请不要担心，我们将在第 9.3.2 节中更详细地了解这些属性。

```
<x xmlns='jabber:x:data' type='form'>
  <title>A Simple Form</title>
  <instructions>Fill out this simple form!</instructions>
  <field type='text-single'
    label="What's your favorite color?"
    var='favorite_color'>
    <required/>
  </field>
</x>
```

这种类型的表单通常带有标签，而且对于有些特定的字段还包含选项列表。当表单填写完毕并返回时，通常只发送响应数据，这些字段的元数据将被忽略。

上述表单的完整版本就像下面这样。

```
<x xmlns='jabber:x:data' type='submit'>
  <field type='text-single'
    var='favorite_color'>
    <value>orange</value>
  </field>
</x>
```

表单很容易扩展。如果以后决定这个表单应该用来询问用户最喜欢的食物，那么可以将这个问题添加到表单中，而完全不需要修改协议。一个经过精心设计的用户界面可以只向对话框中添加新字段，并将用户的应答发回给服务器。

9.3.2 表单元素、字段和类型

在前一节中我们看到，表单由一些元数据和一些字段组成。这些元素都包含在一个表单元素内，就像`<x xmlns='jabber:x:data' type='form'>`这样。`<x>`元素是一个历史产物，如果协议设计者现在构建 Data Forms，那么他们可能会使用更具描述性的`<form>`。

在`<x>`元素中有一些元数据(就像我们在第 9.3.1 节中看到的`<instructions>`元素)和表单字段。

1. `<x>`元素的 `type` 属性

表单中的 `type` 属性有 4 种可能的值：`form`、`submit`、`cancel` 和 `result`。这些值与表单在特定工作流中所处的位置有关。

值 `form` 意味着这是一个需要填写的空表单。如果正试着打开一个新的银行账户，银行职员交给您一个上面带有表单的写字板，那么这个表单的 `type` 属性值就是 `form`。

当返回一个填好的表单时，它的值就是 `submit`。一旦在银行填好新账户表单，就可以将其交还给银行职员，此时该表单的 `type` 值为 `submit`。

如果在填写表单时改变主意，可以将空表单送回去，此时它的 `type` 值为 `cancel`。当您坐在银行中填写新账户表单时，可能注意到街道对面的那家银行有一个巨大的宣传更低费率的标记。于是您不再填写表单，将其交还给银行职员，并走到街道对面的那家银行。那个可怜的银行职员手里拿着的表单的 `type` 值就是 `cancel`。

最后还有一个 `result` 类型。当表示一般数据集时或表示提交表单动作(生成一些响应数据)的结果时，就可以使用这个类型。在那家更便宜的银行里填完新账户表单之后，银行职员建立该账户，并交给您一份包含账户详细信息的文档。这份文档就是一个 `type` 为 `result` 的表单。

2. 表单元数据

除了表单字段之外，表单还可以包含一个标题和一组指令。这些是直接在用户界面中展示表单时提供给人类用户阅读的信息。在第 9.3.1 节中我们看到过，在最喜欢颜色的那个示例表单中曾经使用这些元素。

3. 表单字段

每个表单的核心均是等待填写或传递结果数据的表单字段。与 HTML 表单不同，XMPP 的 Data Forms 有一个非常丰富的数据类型集合可供指定。甚至可以将表单字段标记为必填或可选，并包含其他元数据来辅助改进显示给用户的结果。

每个字段都可以有一些与之相关联的元数据。我们在第一个示例中看到 `label` 属性。表单字段还可以包含`<required>`元素，这意味着在提交该表单时这个字段必须包含一个值。与表单自身的`<instructions>`元素相似，每个字段也可以有一个`<desc>`元素来携带一段人类可读的、描述该字段作用的说明文字。

每个表单字段必须有一个 `var` 属性，它用唯一标识该字段。它的值可以是任何标识符，但

大多数 XMPP 扩展使用 Field Standardization for Data Forms(XEP-0068)中定义的标准化表单字段。下一节中我们将更详细地讲解这一点。

每个表单字段都有一个 `type` 属性, 它描述了该字段的数据的类型。协议文档定义了如下类型。

- `text-single`: 单行文本, 类似于 HTML 中的`<input type='text'>`
- `text-private`: 在输入时遮蔽的单行文本, 类似于 HTML 中的`<input type='password'>`
- `text-multi`: 多行文本, 类似于 HTML 中的`<textarea>`
- `list-single`: 预定义选项列表中的单个值, 类似于 HTML 中的`<select>`
- `list-multi`: 预定义选项列表中的多个值, 类似于 HTML 中的`<select multiple='multiple'>`
- `jid-single`: 单个 JID
- `jid-multi`: 多个 JID
- `boolean`: `true` 或 `false`
- `hidden`: 一个对用户隐藏的表单字段, 它的值通常是不经修改地返回
- `fixed`: 人类可读的描述, 用于表单中的每节的标题

`list-single` 和 `list-multi` 字段的选项是通过该字段包含的`<option>`元素子节点来指定的。每个`<option>`元素都可以有一个 `label` 属性, 而该选项的值则在`<value>`子节点中指定。下面是一个有三个选项的 `list-single` 字段示例。

```
<field type='list-single' var='animals'
       label='Pick an animal'>
  <option label='Fox'>
    <value>fox</value>
  </option>
  <option label='Hare'>
    <value>hare</value>
  </option>
  <option label='Tortoise'>
    <value>tortoise</value>
  </option>
</field>
```

字段还可以通过包含一个`<value>`元素作为直接子节点来指定一个默认值。如果该字段可以有多个值(通过 `list-multi`、`jid-multi` 和 `text-multi` 字段携带), 那么它可以出现多个`<value>`子节点。当表单返回时, 同样的`<value>`元素保存着填好的字段的值。下面是一个用来询问有关 XMPP 服务器管理员的已提交表单字段的示例。

```
<field type='jid-multi' var='admins'>
  <value>alice@example.com</value>
  <value>bob@example.com</value>
</field>
```

表单可用来操作或返回丰富的、结构化的数据集。在图 9-2 中, 我们可以看到一个 XMPP 客户端正在呈现与多人聊天室配置有关的表单时的截屏。这个表单可以让用户轻易地修改配置。即使服务器支持某种未在 MUC 规范中定义的新特性, 由于表单的灵活性, 用户仍然能够定制它。

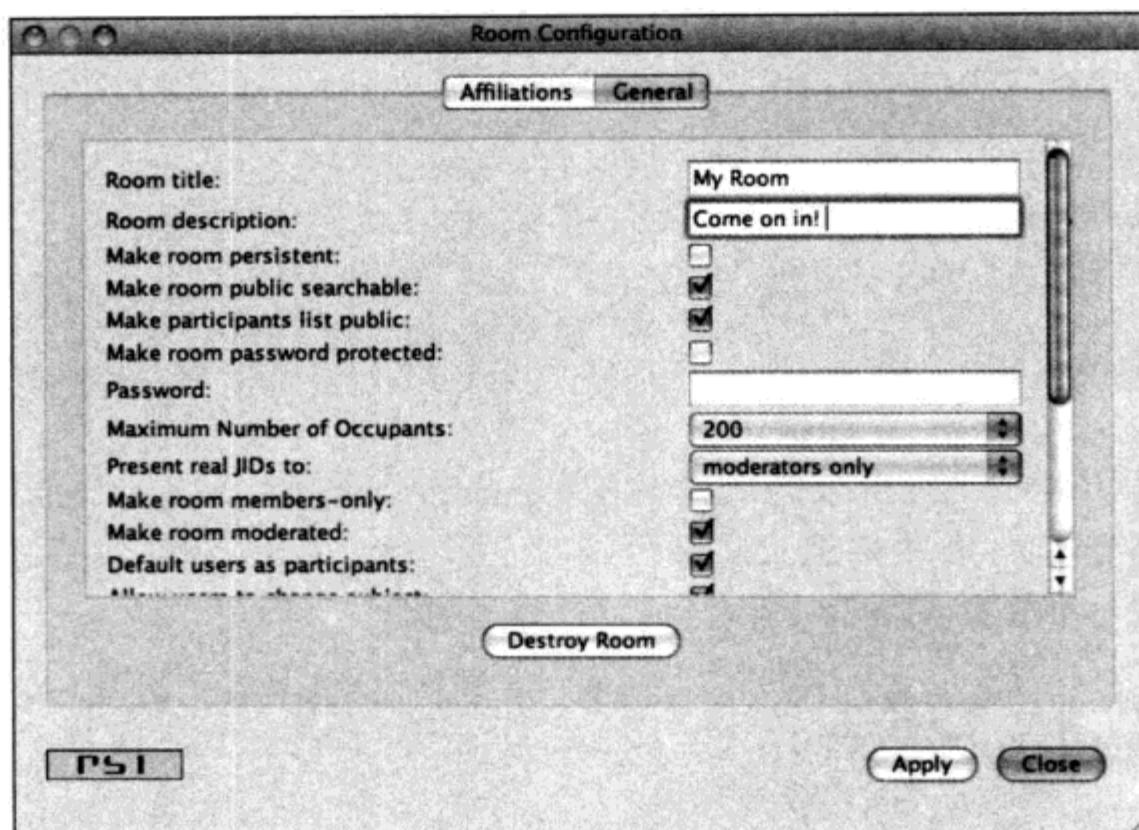


图 9-2

9.3.3 标准化的表单字段

灵活性是一项极好的特性，但让表单如此具有可塑性也带来了一个不好的问题。如何让所有人在常用表单中应该包括哪些字段这个问题上取得一致呢？很快 XMPP 社区就通过 Field Standardization for Data Forms(XEP-0068)解决了这个问题。

每种标准化表单均有一个表单类型，并用一个特殊的字段对它的值进行编码。XMPP 注册机构负责管理 XMPP 扩展中使用的标准类型列表，但定制协议的开发人员不必注册他们的表单类型。因为我们在 SketchCast 应用程序中将使用标准的 pubsub 相关的表单，所以应该会看到这类表单的示例。下面就是一个被提交用来修改 pubsub 节点标题的节点配置表单。

```

<x xmlns='jabber:x:data' type='submit'>
  <field var='FORM_TYPE' type='hidden'>
    <value>http://jabber.org/protocol/pubsub#node_config</value>
  </field>
  <field var='pubsub#title'>
    <value>Best Node Ever</value>
  </field>
</x>

```

标准化表单类型是一个 var 属性值等于 FORM_TYPE 的特殊隐藏字段。这个字段的值就是该表单类型的标识符。pubsub 扩展定义了几种不同的表单类型，查看 XEP-0060 中的 pubsub 规范，可以发现 http://jabber.org/protocol/pubsub#node_config 是节点配置表单的标识符。

标准化表单中的字段的 var 属性均在表单规范中定义。XEP-0060 的第 16.4.2 节指定了 pubsub 的节点配置表单，下面给出了一段包含 pubsub#title 字段的摘要。

```

<form_type>
  <name>http://jabber.org/protocol/pubsub#node_config</name>

```

```

<!-- parts omitted -->
<field var='pubsub#title'
    type='text-single'
    label='A friendly name for the node' />
<!-- parts omitted -->
</form_type>

```

<name>元素的内容就是将要出现在 FORM_TYPE 字段中的值，而<field>元素指定了它的字段类型。

SketchCast 应用程序将使用标准化表单来配置 pubsub 节点，而使用定制表单把绘图信息从发布者那里传给订阅者。如果将 SketchCast 协议本身发布为一个 XEP，那么它可能为绘图信息定义并注册一个标准的表单类型。

9.4 处理 pubsub 节点

pubsub 节点是发布-订阅系统的通信枢纽。用户和应用程序可以订阅感兴趣的节点，当有数据发布到该节点，该节点就会立即将该数据广播给所有订阅者。

这些节点由 pubsub 服务提供。几乎所有 XMPP 服务器都为其用户准备了 pubsub 服务。例如，著名的 XMPP 服务器 jabber.org 就有一项公开的 pubsub 服务，它位于 pubsub.jabber.org。要想开始使用 pubsub，只需知道 pubsub 服务的位置即可。我们可以创建一个有趣的 pubsub 节点，而用户为了订阅该节点，他们需要知道该服务的地址和该节点的名称。

pubsub 节点支持多种动作和配置。SketchCast 应用程序将使用许多基本动作：创建节点、配置节点、发布到节点、删除节点、订阅节点以及从节点接收事件。这看似需要完成大量工作，但稍后就会看到，pubsub 节点相当简单。

9.4.1 创建节点

如果没有一个可供发布和订阅的 pubsub 节点，那么 SketchCast 的用户就不能实现多少功能。正在展示的用户需要创建一个节点，所有的观察者都能够订阅它。

向 pubsub 服务发送一个 IQ-set 节，就可以创建一个 pubsub 节点。

```

<iq to='pubsub.pemberley.lit'
    from='darcy@pemberley.lit/library'
    type='set'
    id='create1'>
    <pubsub xmlns='http://jabber.org/protocol/pubsub'>
        <create node='latest_books' />
    </pubsub>
</iq>

```

pubsub 节点中的大多数动作看上去均与这个相似。在前面几章中我们已经看到几个与此相似的 XMPP 节，这里的不同之处是<pubsub>元素。

与服务发现查询(参见第 7 章)不同的是，pubsub 相关的动作均包含一个由 pubsub 命名空间

之一进行限制的<pubsub>元素。在这里，我们使用了命名空间 `http://jabber.org/protocol/pubsub`，这是主要的pubsub命名空间，但有些动作(比如节点配置)要求像`http://jabber.org/protocol/pubsub#owner`这样的相关命名空间。在阅读这些示例的时候，我们只需简单地跳过所用的命名空间并假设它们都是相同的。如果遇到 pubsub 节错误，那么仔细检查是否针对相关的动作使用了正确的命名空间。

在<pubsub>元素中是请求的动作，即<create>。节点属性中指定了要创建的节点。如果不存在一个有着相同名称的节点，而且允许在这项服务中创建节点，那么服务器将确认请求成功。

```
<iq from='pubsub.pemberley.lit'
  to='darcy@pemberley.lit/library'
  type='result'
  id='create1'>
```

如果该节点已经存在，或者如果没有得到授权来创建它，那么服务器将返回一个携带有<conflict>或<forbidden>条件的 IQ-error 节。

现在 pubsub 节点已经建立完毕并准备好接收事件。

在创建节点时可以让服务器来选择名称。有时候我们并不关心怎么称呼节点，而且如果用户并没有必要挑选一个名称，那么应用程序将变得更加易用。在 SketchCast 中我们将使用这些速成节点，下面的示例演示了如何创建一个这样的节点。

```
<iq from='darcy@pemberley.lit/library'
  to='pubsub.pemberley.lit'
  type='set'
  id='instant1'>
<pubsub xmlns='http://jabber.org/protocol/pubsub'>
  <create/>
</pubsub>
</iq>
```

创建速成节点要比命名节点更加容易。但有一个问题：服务器创建的到底是什么节点？当我们创建速成节点时，服务器会把该节点的名称包含在它的响应中。

```
<iq from='pubsub.pemberley.lit'
  to='darcy@pemberley.lit/library'
  type='result'
  id='instant1'>
<pubsub xmlns='http://jabber.org/protocol/pubsub'>
  <create node='1390361429' />
</pubsub>
</iq>
```

服务器的响应看上去就像是最初的创建命名节点的请求。它为节点名称生成了一个随机数，因为我们告诉过它不关心该节点叫什么名字。pubsub 服务可以按照不同的方式来命名速成节点，有的使用随机的字符串为节点命名，而有的则可能使用数字计数器。

9.4.2 配置节点

新创建的 pubsub 节点有一个可用于多种目的的默认配置。一般情况下，这个默认配置适用于可公开访问的节点，它将最近发布的一些项储存起来，而且只允许节点的创建者(又被称为拥有者)在它上面进行发布。使用的具体配置取决于服务以及它的配置。

一些有着特殊需求的应用程序可能需要适当地配置 pubsub 节点以确保满足这些需求。例如，SketchCast 应该将最近的一些绘画动作保存起来，否则，新的订阅者可能会被不完整的绘图过程弄糊涂。pubsub 节点的配置还包含像节点标题、访问控制以及事件通知是否应该包含原始数据这样的信息。

pubsub 节点配置是通过在一个<configure>动作中提交一个表单来完成的。节点配置必须由该节点的拥有者来完成，因此<pubsub>元素的命名空间与前一个示例稍有不同。首先，请求一个配置表单，看看有哪些选项可用。

```
<iq from='darcy@pemberley.lit/library'
  to='pubsub.pemberley.lit'
  type='get'
  id='configure1'>
<pubsub xmlns='http://jabber.org/protocol/pubsub#owner'>
  <configure node='latest_books' />
</pubsub>
</iq>
```

服务器返回一个空的配置表单。在下面的表单中，我们忽略了许多表单字段，这样就能够将注意力放在协议的结构上而不是它的细节方面。此外，XMPP 服务器对 pubsub 特性集合的支持各不相同，因此服务器有可能并不会返回其中的某些字段。

```
<iq from='pubsub.pemberley.lit'
  to='darcy@pemberley.lit/library'
  type='result'
  id='configure1'>
<pubsub xmlns='http://jabber.org/protocol/pubsub#owner'>
  <configure node='latest_books'>
    <x xmlns='jabber:x:data' type='form'>
      <field var='FORM_TYPE' type='hidden'>
        <value>http://jabber.org/protocol/pubsub#node_config</value>
      </field>
      <field var='pubsub#title' type='text-single'>
        <label>A friendly name for the node</label>
      </field>
      <field var='pubsub#persist_items' type='boolean'>
        <label>Persist items to storage</label>
        <value>true</value>
      </field>
      <field var='pubsub#max_items' type='text-single'>
        <label>Max # of items to persist</label>
        <value>10</value>
      </field>
    </x>
  </configure>
</pubsub>
</iq>
```

```

<!-- more fields -->
</x>
</configure>
</pubsub>
</iq>

```

这个配置表单就像我们在第 9.3 节中曾经看到的那些表单一样。这个表单中的第一个字段是表单类型标识符，它后面则是普通的字段。这个表单中给出的 3 个字段分别是用来控制节点标题、是否保存前面发布的项以及如果保存的话那么保存多少项。

因为我们想修改节点的设置，所以必须向服务器提交一个包含预期配置的完整的表单。这里的表单提交节与前一个示例中的几乎完全相同，但这里将字段标签和类型移除，填入字段值，并将表单的 `type` 属性修改为 `submit`。这里只修改了示例中的 3 项设置，但我们还可以一次只修改一项设置，或者同时修改所有可能的设置：

```

<iq from='darcy@pemberley.lit/library'
    to='pubsub.pemberley.lit'
    type='set'
    id='configure2'>
<pubsub xmlns='http://jabber.org/protocol/pubsub#owner'>
<configure node='latest_books'>
<x xmlns='jabber:x:data' type='submit'>
<field var='FORM_TYPE'>
<value>http://jabber.org/protocol/pubsub#node_config</value>
</field>
<field var='pubsub#title'>
<value>Books I've Read Lately</value>
</field>
<field var='pubsub#persist_items'>
<value>true</value>
</field>
<field var='pubsub#max_items'>
<value>100</value>
</field>
</x>
</configure>
</pubsub>
</iq>

```

服务器应该用一个成功应答来响应提交：

```

<iq from='pubsub.pemberley.lit'
    to='darcy@pemberley.lit/library'
    type='result'
    id='configure2' />

```

现在，Darcy 先生有了一个 `pubsub` 节点，该节点有一个可以吸引来订阅者的醒目标题，它

还能够保存有关 Darcy 先生最近已经阅读的 100 本书的数据。

9.4.3 pubsub 事件

pubsub 节点以及它们的配置非常必要而且有用，但它们本身并没有完成什么具体的功能。pubsub 节点的真正价值体现在发布到它们的事件以及广播到订阅者的事件。

pubsub 事件中可以包含任何数据。pubsub 服务并不知道也不关心事件中包含的数据，它只是将该数据广播给节点的订阅者。pubsub 事件的内容被称为有效载荷。一般而言，订阅者知道给定节点发布的是什么有效载荷而且知道如何解释该有效载荷。有一些常见的事件有效载荷，比如采用 Atom 内容联合格式的博客帖子或者采用 User Tune 格式(XEP-0118)的关于正在播放乐曲的信息。

在发布时，该事件将被封装到<pubsub>元素中的<publish>动作中，而当接收事件时，同样的事件被放在<message>节中。在两种情况下，事件的有效载荷均是相同的。下面是一个携带 User Tune 有效载荷的事件示例。

```
<tune xmlns='http://jabber.org/protocol/tune'>
  <artist>Elizabeth Bennet</artist>
  <title>A Piano Song for Lady Catherine</title>
</tune>
```

表单非常灵活足以表示许多类型的有效载荷，而 Darcy 先生选择使用表单对自己已经读过的书籍进行编码。每当他读完一本书之后，他就会使用下面的基于表单的有效载荷向他的 pubsub 节点发布一个事件。

```
<x xmlns='jabber:x:data' type='result'>
  <field var='title'>
    <value>A History of Pemberley</value>
  </field>
  <field var='author'>
    <value>Sir Lewis de Bourgh</value>
  </field>
</x>
```

上面的示例都包含事件有效载荷，但我们还可以配置节点只发送事件通知。在这种情况下，可以通过查询 pubsub 节点来获取有效载荷。在下面几节中我们将了解如何发布、接收事件以及如何获取有效载荷。

9.4.4 发布到节点

Darcy 先生刚刚读完最近的一本书，因此现在必须向他的 pubsub 节点发布一个新事件。在前一节中我们曾经看过 Darcy 先生的事件有效载荷，下面则演示了如何发布这些有效载荷。

```
<iq from='darcy@pemberley.lit/library'
  to='pubsub.pemberley.lit'
  type='set'>
```

```

    id='publish1'
<pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish node='latest_books'>
        <item>
            <x xmlns='jabber:x:data' type='result'>
                <field var='title'>
                    <value>A History of Pemberley</value>
                </field>
                <field var='author'>
                    <value>Sir Lewis de Bourgh</value>
                </field>
            </x>
        </item>
    </publish>
</pubsub>
</iq>

```

在这个示例中，事件有效载荷被封装到一个<item>元素中，然后放入<publish>动作中。如果 Darcy 先生刚刚读完两本书，那么他只需将第二本书(放入单独的<item>元素中)追加进来即可。服务器将接受发布请求，并返回一个成功响应。

```

<iq from='pubsub.pemberly.lit'
    to='darcy@pemberley.lit/library'
    type='result'
    id='publish1'>
    <pubsub xmlns='http://jabber.org/protocol/pubsub'>
        <publish node='latest_books'>
            <item id='821b576dfabfc6b358b6ec4139b87f5c' />
        </publish>
    </pubsub>
</iq>

```

注意，服务器的响应还包含一个<item>元素。因为 Darcy 先生的<item>元素并不包含 id 属性，所以服务器替他创建了一个 id 属性(这是因为每项都必须有一个标识符)。这些标识符将用于通知、项目获取以及撤销。

Darcy 还可以在他的原始发布节中指定一个 id 属性，这样服务器就会使用该 id。这与前面的普通节点和速成节点的创建示例非常相似。

9.4.5 订阅和退订

Elizabeth 和 Georgiana 都是热心的读者，她们非常希望了解 Darcy 先生最近阅读过哪些书籍。这两位女士均可以通过发送一个携带<subscribe>动作的 IQ-set 节来订阅 Darcy 先生的 pubsub 节点。

```

<iq from='elizabeth@longbourn.lit/outside'
    to='pubsub.pemberley.lit'
    type='set'>

```

```

    id='subscribe1'>
<pubsub pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <subscribe node='latest_books'
               jid='elizabeth@longbourn.lit/outside' />
</pubsub>
</iq>

```

Elizabeth 同时指定了她希望订阅的节点以及将事件传送的目的地址。只要她一直是订阅者，那么每当 Darcy 先生读完一本书并发布相应的事件，她就立即会接到通知。

服务器针对 Elizabeth 的请求发送一个成功响应。

```

<iq from='pubsub.pemberley.lit'
    to='elizabeth@longbourn.lit/outside'
    type='result'
    id='subscribe1'>
<pubsub pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <subscribe node='latest_books'
               jid='elizabeth@longbourn.lit/outside'
               subscription='subscribed' />
</pubsub>
</iq>

```

subscription 属性用于向 Elizabeth 通告有关她的订阅的状态，在这个示例中它告诉她已经完成订阅了。还可以为 pubsub 节点配置多种访问控制模型，其中有些模型要求节点的拥有者来手工批准订阅请求。如果 Darcy 先生的节点是按照这种方式配置的，那么 subscription 属性的值就会是 pending。

如果由于某种原因 Darcy 先生配置自己的节点只允许一组特定的人订阅(白名单访问模型)，那么 Elizabeth 的请求可能会返回一个 not-allowed 错误。

```

<iq from='pubsub.pemberley.lit'
    to='elizabeth@longbourn.lit/outside'
    type='error'
    id='subscribe1'>
<pubsub pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <subscribe node='latest_books'
               jid='elizabeth@longbourn.lit/outside' />
</pubsub>
<error type='cancel'>
    <not-allowed xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
    <closed-node xmlns='http://jabber.org/protocol/pubsub#errors' />
</error>
</iq>

```

既然 Elizabeth 已经订阅，那么当 Darcy 先生读完一本书时她将接收到事件广播。事件广播是通过一个含有位于 <http://jabber.org/protocol/pubsub#event> 命名空间下面的<event>子元素的

<message>节来发送的。<event>元素的<items>子元素中将含有一个或多个<item>子元素，而有效载荷被放在<item>元素中(如果包含有效载荷的话)。在这个示例中，只有一项，而该项的id属性将匹配服务器前面返回给 Darcy 的值。这里的有效载荷也与 Darcy 发布的有效载荷相同。

```

<message from='pubsub.pemberley.lit'
          to='elizabeth@longbourn.lit/outside'>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='latest_books'>
      <item id='821b576dfabfc6b358b6ec4139b87f5c'>
        <x xmlns='jabber:x:data' type='result'>
          <field var='title'>
            <value>A History of Pemberley</value>
          </field>
          <field var='author'>
            <value>Sir Lewis de Bourgh</value>
          </field>
        </x>
      </item>
    </items>
  </event>
</message>

```

与 Elizabeth 不同，Georgiana 每天都能见到 Darcy，这种经常性的联系使得她即使不订阅 pubsub 也知道 Darcy 的阅读习惯。在订阅 Darcy 的 pubsub 节点数天之后，她觉得发布给自己的信息对于她来说没有多大用处。于是，她通过发送一个携带<unsubscribe>动作的 IQ-set 节来取消订阅。

```

<iq from='georgiana@pemberley.lit/piano'
      to='pubsub.pemberley.lit'
      type='set'
      id='unsubscribe1'>
  <pubsub pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <unsubscribe node='latest_books'
                 jid='georgiana@pemberley.lit/piano' />
  </pubsub>
</iq>

```

服务器响应一个成功的结果。

```

<iq from='pubsub.pemberley.lit'
      to='georgiana@pemberley.lit/piano'
      type='result'
      id='unsubscribe1' />

```

以后任何来自 Darcy 的读书更新信息都不会再发送给 Georgiana。

9.4.6 检索订阅情况

Darcy 富有好奇心，虽然他本身不愿意承认这一点，但是他经常希望了解一下有多少人关注他的读书更新信息。当他发布一次新的更新时，他并没有得到通知告诉它有多少人已经接收到他的事件。但 Darcy 可以轻易地查询到他的 pubsub 节点的订阅者列表。

```
<iq from='darcy@pemberley.lit/library'
    to='pubsub.pemberley.lit'
    type='get'
    id='subscribers1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub#owner'>
    <subscriptions node='latest_books' />
  </pubsub>
</iq>
```

可以看到 Darcy 使用了命名空间 `http://jabber.org/protocol/pubsub#owner`，这是因为检索订阅者列表是一个仅限于节点拥有者的动作。在普通的`<pubsub>`元素中，他放置了一个指向他的 pubsub 节点的`<subscriptions>`动作。服务器返回一个相当短的订阅者列表。

```
<iq from='pubsub.pemberley.lit'
    to='darcy@pemberley.lit/library'
    type='result'
    id='subscribers1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub#owner'>
    <subscriptions node='latest_books'>
      <subscription jid='elizabeth@longbourn.lit/outside'
                    subscription='subscribed' />
      <subscription jid='bingley@netherfield.lit/house'
                    subscription='subscribed' />
    </subscriptions>
  </pubsub>
</iq>
```

只有他的好友 Charles 和 Elizabeth 关注他在读什么书。

9.4.7 获取项

Elizabeth 只订阅了 Darcy 的 `latest_books` 节点，而她错过了本周早些时候的事件广播。如果她希望更多地了解他，那么她必须研究他过去的阅读材料。

Darcy 将配置他的节点保存项。任何人都可以查询他的节点来获取最近发布的项。在这里，通过向该节点发送一个携带`<items>`动作的 IQ-get 节来请求最近的 5 项。

```
<iq from='elizabeth@longbourn.lit/outside'
    to='pubsub.pemberley.lit'
    type='get'
    id='items1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
```

```

<items node='latest_books' max_items='3' />
</pubsub>
</iq>

```

<items>元素包含一个 node 属性，就像我们前面看过的其他动作一样。Elizabeth 还将 max_items 属性设为 3，这是因为她只对最近的历史感兴趣。如果她忽略 max_items，那么服务器将认为她请求发送自它被配置以来保存的所有历史数据。如果将 max_items 设为 500(这远大于为该节点配置的最大值)，那么服务器将发送尽可能多的可用数据。

服务器响应请求的项以及它们的有效载荷。

```

<iq from='pubsub.pemberley.lit'
  to='elizabeth@longbourn.lit/outside'
  type='result'
  id='items1'>
<pubsub xmlns='http://jabber.org/protocol/pubsub'>
  <items node='latest_books'>
    <item id='4f900045977f0ccd372c4a670bcba27f'>
      <x xmlns='jabber:x:data' type='result'>
        <field var='title'>
          <value>Of Acquaintances and Persuasion</value>
        </field>
        <field var='author'>
          <value>Daleforth Carnham</value>
        </field>
      </x>
    </item>
    <item id='16ddab0d5b3572388446c552d1bdf793'>
      <x xmlns='jabber:x:data' type='result'>
        <field var='title'>
          <value>Managing Temperment</value>
        </field>
        <field var='author'>
          <value>Sarah Pratt</value>
        </field>
      </x>
    </item>
    <item id='e4139c9d583558c172a28f68ec036c6c'>
      <x xmlns='jabber:x:data' type='result'>
        <field var='title'>
          <value>The Haunting at Hertfordshire</value>
        </field>
        <field var='author'>
          <value>Sir William Lucas</value>
        </field>
      </x>
    </item>
  </items>
</pubsub>
</iq>

```

还可以配置节点只发送通知，这样就会缺少有效载荷。如果 Darcy 先生的节点按照这样配置，那么服务器的响应将会是下面这个样子。

```
<iq from='pubsub.pemberley.lit'
  to='elizabeth@longbourn.lit/outside'
  type='result'
  id='items1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <items node='latest_books'>
      <item id='4f900045977f0ccd372c4a670bcba27f' />
      <item id='16ddab0d5b3572388446c552d1bdf793' />
      <item id='e4139c9d583558c172a28f68ec036c6c' />
    </items>
  </pubsub>
</iq>
```

必须向服务器单独发送一个请求来检索各项以及它们的有效载荷。Elizabeth 将像前面一样发送一个携带<items>动作的 IQ-get 节，但<items>元素将包含她感兴趣的项列表。下面给出了一个示例。

```
<iq from='elizabeth@longbourn.lit/outside'
  to='pubsub.pemberley.lit'
  type='get'
  id='items2'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <items node='latest_books'>
      <item id='4f900045977f0ccd372c4a670bcba27f' />
      <item id='16ddab0d5b3572388446c552d1bdf793' />
      <item id='e4139c9d583558c172a28f68ec036c6c' />
    </items>
  </pubsub>
</iq>
```

然后服务器将以有效载荷作为响应，就像最初的那个示例一样。

如果一个节点被配置为只发送通知，那么订阅者将接收到包含新项列表的<message>。也需要类似的检索过程来获取这些项的事件有效载荷。

9.4.8 订阅管理

我们看到，当订阅一个 pubsub 节点时，必须包含希望从哪里接收事件通知的 ID。这个 JID 可以是一个裸 JID(如 elizabeth@longbourn.lit)或一个完整 JID(如 darcy@pemberley.lit/library)。因为事件的传送是通过<message>节来完成的，所以根据订阅是针对裸 JID 还是针对完整 JID，传送的语义是不同的。应用程序应该使用裸 JID 还是完整 JID 取决于具体的环境。

通常，只要 pubsub 节点存在或者直到用户退订，所有订阅都将一直持续。pubsub 节点还可以有不同的订阅生命周期，该规范包含了几个规定订阅期满的示例(示例请参见 XEP-0060 的第 12.18 节)。例如，当匿名连接的用户的会话终止时，他们的订阅也将被移除，即使他们从来没

有显式地退订。有些聪明的开发人员甚至在研究基于出席的订阅：一旦订阅者离线，就可以立即取消该订阅。

考虑到可用的 pubsub 服务器和节点的数目庞大，Elizabeth 可能忘记自己订阅了哪些，以及应该将事件发送到哪里。如果 Elizabeth 记得她可能订阅过的 pubsub 服务，那么她可以向它们询问自己的订阅列表。

```
<iq from='elizabeth@longbourn.lit/outside'
    to='pubsub.pemberley.lit'
    type='get'
    id='mysubs1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <subscriptions/>
  </pubsub>
</iq>
```

Elizabeth 要求 pubsub.pemberley.lit 上的服务列出她在该服务中的所有节点上的订阅。她还可以在<subscriptions>动作上指定一个 node 属性，将自己的查询限制到特定的节点。在这里，服务器告诉她，除了订阅 Darcy 的最新读书消息之外，她还有一项被遗忘的订阅，即 Georgiana 的公开日记。

```
<iq from='pubsub.pemberley.lit'
    to='elizabeth@longbourn.lit/outside'
    type='result'
    id='mysubs1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <subscriptions>
      <subscription node='latest_books'
                    jid='elizabeth@longbourn.lit/outside'
                    subscription='subscribed' />
      <subscription node='public_diary'
                    jid='elizabeth@longbourn.lit'
                    subscription='subscribed' />
    </subscriptions>
  </pubsub>
</iq>
```

我们已经看过了 pubsub 中许多重要的用例，这已经为我们创建本章的应用程序提供了足够的协议知识。

9.5 使用 pubsub 广播绘图

现在是时候将理论付诸实践了。在本章前面我们使用了很大的篇幅来熟悉 Data Forms 和 Publish-Subscribe，而这些新知识将使得构建 SketchCast 应用程序变得容易。

展示用户必须创建一个 pubsub 节点，将绘图事件转换成基于表单的有效载荷，将事件发布

到节点，一旦完成就删除该节点。观众用户需要订阅展示者的 pubsub 节点，检索过去的绘图事件，将事件转换成绘图命令，处理新的事件通知并最终退订该节点。在整个过程中应用程序应该用户友好，可处理常见错误并让用户可视地通信。

9.5.1 构建用户界面

SketchCast 用户界面只需要少数几个主要部分。首先需要一些空间来绘图。还需要一些按钮来改变所用的颜色、改变线条的粗细、擦除绘画以及断开连接。还需要一个小的状态条向用户报告正在发生的事情。最后，需要一个登录对话框和一个错误提示对话框。

程序清单 9-1 给出了应用程序的 HTML 标记，它包含了所有这些界面元素。



可从
Wrox.com
下载源代码

程序清单 9-1 sketchcast.html

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

<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=UTF-8" />
    <title>SketchCast - Chapter 9</title>

    <link rel='stylesheet' href='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/themes/cupertino/jquery-ui.css'>
    <script src='http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/jquery.js'>
    </script>
    <script src='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/jquery-ui.js'></script>
    <script src='scripts/strophe.js'></script>
    <script src='scripts/flXHR.js'></script>
    <script src='scripts/strophe.flxhr.js'></script>

    <link rel='stylesheet' type='text/css' href='sketchcast.css'>
    <script type='text/javascript' src='sketchcast.js'></script>
  </head>
  <body>
    <h1>SketchCast</h1>

    <!-- status bar -->
    <div id='status'></div>

    <!-- drawing tool buttons. The ids are the button values -->
    <div class='button disabled color' id='color-000'></div>
    <div class='button disabled color' id='color-f00'></div>
    <div class='button disabled color' id='color-0f0'></div>
    <div class='button disabled color' id='color-00f'></div>
    <div class='button disabled linew' id='color-2'><div></div></div>
    <div class='button disabled linew' id='color-4'><div></div></div>
    <input id='erase' type='button' value='erase sketch' disabled>
    <input id='disconnect' type='button' value='disconnect' disabled>
    <div class='clear'></div>
```

```
<!-- drawing area -->
<canvas id='sketch' class='disabled' width='600'
        height='500'></canvas>

<!-- login dialog -->
<div id='login_dialog' class='hidden'>
  <label>JID:</label><input type='text' id='jid'>
  <label>Password:</label><input type='password' id='password'>
  <label>Pubsub service:</label><input type='text' id='service'>
  <label>Pubsub node:</label><input type='text' id='node'>
</div>

<!-- empty error dialog -->
<div id='error_dialog' class='hidden'>
  <p></p>
</div>
</body>
</html>
```

程序清单 9-2 给出了这个应用程序的 CSS。



可从
Wrox.com
下载源代码

程序清单 9-2 sketchcast.css

```
.clear {
  clear: both;
}

.button {
  border: solid 1px black;
  width: 25px;
  height: 25px;
  float: left;
}

div.disabled {
  -moz-opacity: 0.25;
  opacity: 0.25;
}

.hidden {
  display: none;
}

#sketch {
  border: solid 1px black;
}

#color-000 {
  background-color: #000;
}
```

```

#color-f00 {
    background-color: #f00;
}

#color-0f0 {
    background-color: #0f0;
}

#color-00f {
    background-color: #00f;
}

#width-2 div {
    background-color: #000;
    height: 5px;
    margin-top: 10px;
}

#width-4 div {
    background-color: #000;
    height: 10px;
    margin-top: 7px;
}

#erase {
    margin-left: 5px;
}

```

这两个对话框起初处于隐藏状态，而状态条开始时为空。所有其他控件都显示出来但均被禁用。当展示者开始发布绘画时，将启用工具条和绘图区域。观看者的工具条始终被禁用，但一旦订阅绘画，绘图区域将变得可用。

9.5.2 使用 Canvas 绘制草图

在关注应用程序的 XMPP 部分之前，我们应该将绘图区域和绘图工具连接起来，这样就可以进行一些绘画测试。

绘画是通过一根虚拟的画笔来完成的，它类似于许多消费级图形应用程序中的画笔。该笔的端点是鼠标指针，当鼠标按钮按下时它就会在绘图区域留下墨迹。工具栏按钮用来更改墨水的颜色、产生的线条的粗细以及擦除整个画面。

首先，必须记录该笔是收起还是按下，并将其连接到 `mouseup` 和 `mousedown` 事件。接下来，当鼠标移动时，应该检查该笔是否按下，如果是，就从它原来的位置画一条连接当前位置的直线。创建一个名为 `sketchcast.js` 的文件，并将下面的代码添加到该文件中。



```

var SketchCast = {
    pen_down: false,
    old_pos: null
};

$(document).ready(function () {

```

```

$( '#sketch' ).mousedown( function () {
    SketchCast.pen_down = true;
});

$( '#sketch' ).mouseup( function () {
    SketchCast.pen_down = false;
});

$( '#sketch' ).mousemove( function (ev) {
    // get the position of the drawing area, our offset
    var offset = $( this ).offset();
    // calculate our position within the drawing area
    var pos = { x: ev.pageX - offset.left,
                y: ev.pageY - offset.top };

    if ( SketchCast.pen_down ) {
        if ( !SketchCast.old_pos ) {
            SketchCast.old_pos = pos;
            return;
        }

        // render the line segment
        var ctx = $( '#sketch' ).get( 0 ).getContext( '2d' );
        ctx.beginPath();
        ctx.moveTo( SketchCast.old_pos.x, SketchCast.old_pos.y );
        ctx.lineTo( pos.x, pos.y );
        ctx.stroke();

        SketchCast.old_pos = pos;
    } else {
        SketchCast.old_pos = null;
    }
});
});

```

code snippet sketchcast.js

一旦应用程序加载完毕，它就绑定 mousedown、mouseup 和 mousemove 事件。mousedown 和 mouseup 事件简单地修改画笔的状态，而 mousemove 事件完成了所有实际工作。

为了进行绘图，需要确切地知道当前位于绘图区域的什么位置。这个问题有点棘手，这是因为<canvas>元素的坐标系统是从它的左上角开始，但 mousemove 事件中鼠标的坐标则是相对于浏览器视区的左上角。jQuery 的 offset() 函数帮助解决了这个问题，它可以提供元素相对于视区的坐标。如果从 mousemove 事件坐标中减去这些坐标，那么运算结果就是相对于绘图区域左上角的一组坐标。

HTML5 <canvas> 元素能够完成一些非常神奇的功能，但这个应用程序的需求相当简单。稍后我们将讲解 SketchCast 所需的全部绘图逻辑，但如果对画布元素能够完成哪些功能感兴趣，那么可以参考 Opera 开发人员网站上的一份极好的手册 <http://dev.opera.com/articles/view/html-5-canvas-the-basics/>。

所有的画布操作都要有一个绘图上下文，它包含了像当前颜色和线宽这样的绘图信息。如果以前曾经做过图形编程工作，那就很可能已经了解与此类似的绘图环境。因为 SketchCast 只能实现二维绘图，所以使用画布 API 的 `getContext()` 函数来创建一个上下文 2d，如下所示。

```
var ctx = $('#sketch').get(0).getContext('2d');
```

在该上下文上调用 `beginPath()` 函数，开始定义一条路径。路径可以相当复杂，但在 SketchCast 中我们只使用简单的线段。`moveTo()` 用来将画笔移到鼠标的最后已知位置，但不标记画布，而 `lineTo()` 用来添加一个从前一个位置到当前位置的线段。路径已经完成，因此可以调用 `stroke()` 让其显示在画布上。

在 Web 浏览器中加载 SketchCast，试着画几幅绘画。

虽然试验很有意思，但很快就会发现绘制细细的黑线条没有多大意思。我们可以通过启用工具栏上的按钮(从而能够修改画笔的颜色和宽度)来让绘画更加有趣。绑定工具栏上所有按钮的单击事件，其实现如下所示。对 `sketchcast.js` 文件完成如下突出显示的部分的修改。



可从
Wrox.com
下载源代码

```
var SketchCast = {
    pen_down: false,
    old_pos: null,
    color: '000',
    line_width: 4
};

$(document).ready(function () {
    $('#sketch').mousedown(function () {
        SketchCast.pen_down = true;
    });

    $('#sketch').mouseup(function () {
        SketchCast.pen_down = false;
    });

    $('#sketch').mousemove(function (ev) {
        // get the position of the drawing area, our offset
        var offset = $(this).offset();
        // calculate our position within the drawing area
        var pos = {x: ev.pageX - offset.left,
                  y: ev.pageY - offset.top};

        if (SketchCast.pen_down) {
            if (!SketchCast.old_pos) {
                SketchCast.old_pos = pos;
                return;
            }

            // render the line segment
            var ctx = $('#sketch').get(0).getContext('2d');
            ctx.strokeStyle = '#' + SketchCast.color;
            ctx.lineWidth = SketchCast.line_width;
        }
    });
});
```

```

    ctx.beginPath();
    ctx.moveTo(SketchCast.old_pos.x, SketchCast.old_pos.y);
    ctx.lineTo(pos.x, pos.y);
    ctx.stroke();

    SketchCast.old_pos = pos;
} else {
    SketchCast.old_pos = null;
}
});

$('.color').click(function (ev) {
    SketchCast.color = $(this).attr('id').split('-')[1];
});

$('.linew').click(function (ev) {
    SketchCast.line_width = $(this).attr('id').split('-')[1];
});

$('#erase').click(function () {
    var ctx = $('#sketch').get(0).getContext('2d');
    ctx.fillStyle = '#fff';
    ctx.strokeStyle = '#fff';
    ctx.fillRect(0, 0, 600, 500);
});
});
});

```

code snippet sketchcast.js

这里已经添加了两个新的状态变量(`color` 和 `line_width`)，而且已经修改代码在绘图时使用这些值。擦除按钮的实现非常简单，它在整幅画布的上面绘制一个白色的矩形。但我们还需要对工具栏按钮稍作解释。

所有的颜色按钮的工作方式基本相同，只是它们设置的颜色值不同而已。我们并没有为每个按钮编写一个不同的处理程序，而是使用按钮的 `id` 属性来保存按钮将要设置的颜色值。所有的颜色按钮都有 CSS 类 `color`，而且每当这些按钮中的一个被按下时，我们都能够从它的 `id` 属性中获取颜色值来设置绘图颜色。同样的技术也用于线宽按钮。这节省了大量的代码输入工作。

在 Web 浏览器中重新加载 SketchCast，并练习各种新的创作选项。

9.5.3 登录并建立节点

要想让 SketchCast 变得实用，下一步就要开始构建 pubsub 支持，这样用户就能够与他人分享绘图。首先，我们想要启用登录对话框并确保当应用程序开始时将其弹出。然后连接到 XMPP 服务器，如果用户进行展示，那么创建并配置一个 pubsub 节点，而如果他只是观看，那么就要订阅一个现有的节点。

登录对话框除了用来输入用户名和口令的普通字段之外，它还包含两个分别用来指定 pubsub 服务和 pubsub 节点的字段。如果展示者希望创建一个新的绘画，就把 pubsub 节点留空，而如果观看者希望观看绘画，就要输入该绘画的节点。将下面的代码放到文档准备就绪事件处

理程序的顶部。



可从
Wrox.com
下载源代码

```
$('#login_dialog').dialog({
    autoOpen: true,
    draggable: false,
    modal: true,
    title: 'Connect to a SketchCast',
    buttons: {
        "Connect": function () {
            $(document).trigger('connect', {
                jid: $('#jid').val(),
                password: $('#password').val(),
                service: $('#service').val(),
                node: $('#node').val()
            });
            $('#password').val('');
            $(this).dialog('close');
        }
    }
});
```

code snippet sketchcast.js

与其他应用程序一样，SketchCast 需要记录连接。它还必须保存 pubsub 服务和节点。SketchCast 还需要一些用于 pubsub 和表单的命名空间常量。将下面的属性添加到 SketchCast 对象中。



可从
Wrox.com
下载源代码

```
connection: null,
service: null,
node: null,

NS_DATA_FORMS: "jabber:x:data",
NS_PUBSUB: "http://jabber.org/protocol/pubsub",
NS_PUBSUB_OWNER: "http://jabber.org/protocol/pubsub#owner",
NS_PUBSUB_ERRORS: "http://jabber.org/protocol/pubsub#errors",
NS_PUBSUB_NODE_CONFIG: "http://jabber.org/protocol/pubsub#node_config"
```

code snippet sketchcast.js

还需要钩入错误提示对话框。为了简洁起见，我们将所有的错误视作重大错误，当它们发生时终止连接。我们并不想让用户混淆，因此当他们关闭错误提示对话框时，我们必须重新打开登录对话框，这样他们就能够重新开始。稍后，我们将添加一个函数来报告错误(从而打开这个对话框)。将下面的代码添加到文档准备就绪事件处理程序中。



可从
Wrox.com
下载源代码

```
$('#error_dialog').dialog({
    autoOpen: false,
```

```

draggable: false,
modal: true,
title: 'Whoops! Something Bad Happened!',
buttons: {
    "Ok": function () {
        $(this).dialog('close');
        $('#login_dialog').dialog('open');
    }
}
);

```

code snippet sketchcast.js

将熟悉的连接逻辑事件处理程序添加到 sketchcast.js 文件的末尾。



可从
Wrox.com
下载源代码

```

$(document).bind('connect', function (ev, data) {
    $('#status').html('Connecting..');

    var conn = new Strophe.Connection(
        'http://bosh.metajack.im:5280/xmpp-httpbind');

    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });

    SketchCast.connection = conn;
    SketchCast.service = data.service;
    SketchCast.node = data.node;
});

$(document).bind('connected', function () {
    // nothing here yet
});

```

code snippet sketchcast.js

一旦 SketchCast 连接，它必须发送初始出席并设置或订阅一个 pubsub 节点。修改 connected 事件处理程序来完成这项工作。



可从
Wrox.com
下载源代码

```

$(document).bind('connected', function () {
    $('#status').html("Connected.");

    // send negative presence since we're not a chat client
    SketchCast.connection.send($pres().c('priority').t('-1'));

    if (SketchCast.node.length > 0) {

```

```

// a node was specified, so we attempt to subscribe to it

// first, set up a callback for the events
SketchCast.connection.addHandler(
    SketchCast.on_event,
    null, "message", null, null, SketchCast.service);

// now subscribe
var subiq = $iq({to: SketchCast.service,
                  type: "set"})
.c('pubsub', {xmlns: SketchCast.NS_PUBSUB})
.c('subscribe', {node: SketchCast.node,
                  jid: SketchCast.connection.jid});
SketchCast.connection.sendIQ(subiq,
                             SketchCast.subscribed,
                             SketchCast.subscribe_error);

} else {
    // a node was not specified, so we start a new sketchcast
    var createiq = $iq({to: SketchCast.service,
                        type: "set"})
.c('pubsub', {xmlns: SketchCast.NS_PUBSUB})
.c('create');
SketchCast.connection.sendIQ(createiq,
                             SketchCast.created,
                             SketchCast.create_error);
}
}

```

code snippet sketchcast.js

这里使用了负的出席优先级，这是因为 SketchCast 不能处理普通的聊天消息。我们在第 8 章中也曾经使用过这个技术，目的是为了避免意外地获取到属于该用户的其他已连接资源的个人消息。

如果用户指定了一个节点，那么应用程序将准备一个 pubsub 事件处理程序，构建一个 subscription 节(就像第 9.4 节中的一样)并向服务器发送请求。如果接收到成功回复，就调用 subscribed() 函数，否则就说明出现错误并调用 subscribe_error()。添加一个<message>处理程序，这样当接收到事件时，on_event() 就能够处理它。

如果没有指定节点，那么 SketchCast 构建并发送一个类似的 IQ-set 节来创建一个节点。一旦节点创建完毕，该代码就调用 created()，否则出现任何 IQ-error 响应，都调用 create_error()。

SketchCast.on_event() 的初始实现什么也不做。我们稍后将向它里面添加更多代码。将这个占位符函数添加到 SketchCast 对象中。



```

on_event: function (msg) {
    // blank for now
}

```

可从
Wrox.com
下载源代码

code snippet sketchcast.js

现在将 `subscription` 处理程序添加到 `SketchCast` 对象中。



```
subscribed: function (iq) {
    $(document).trigger("reception_started");
},
subscribe_error: function (iq) {
    SketchCast.show_error("Subscription failed with " +
        SketchCast.make_error_from_iq(iq));
}
```

code snippet sketchcast.js

一旦成功订阅，该代码就会引发一个自定义事件 `reception_started`。如果订阅失败，就会给出一个错误提示信息。我们需要实现这两个错误处理程序函数 `show_error()` 和 `make_error_from_iq()`，将下面的代码添加到 `SketchCast` 对象中。



```
make_error_from_iq: function (iq) {
    var error = $(iq)
        .find('*[xmlns="' + Strophe.NS.STANZAS + '"]')
        .get(0).tagName;
    var pubsub_error = $(iq)
        .find('*[xmlns="' + SketchCast.NS_PUBSUB_ERRORS + '"]');
    if (pubsub_error.length > 0) {
        error = error + "/" + pubsub_error.get(0).tagName;
    }
    return error;
},
show_error: function (msg) {
    SketchCast.connection.disconnect();
    SketchCast.connection = null;
    SketchCast.service = null;
    SketchCast.node = null;
    $('#error_dialog p').text(msg);
    $('#error_dialog').dialog('open');
}
```

code snippet sketchcast.js

`make_error_from_iq()` 函数查看 IQ-error 节的 `<error>` 元素，将通用错误类型和应用程序特有错误类型(如果有的话)提取出来。它将这些错误提示合并到一个要展示给用户的简单字符串中。`show_error()` 断开连接，重置 XMPP 状态，并弹出携带指定消息的错误提示对话框。这些错误提示消息并非用户友好，但它们易于生成而且能够完成任务。在实际的应用程序中，应该花更多精力来呈现更有帮助的错误提示。

订阅工作已经处理完毕，因此下面要继续完成为展示者创建节点的任务。下面是节点创建回调的实现，应该将其添加到 `SketchCast` 对象中。



```

created: function (iq) {
    // find pubsub node
    var node = $(iq).find("create").attr('node');
    SketchCast.node = node;

    // configure the node
    var configiq = $iq({to: SketchCast.service,
                        type: "set"})
        .c('pubsub', {xmlns: SketchCast.NS_PUBSUB_OWNER})
        .c('configure', {node: node})
        .c('x', {xmlns: SketchCast.NS_DATA_FORMS,
                  type: "submit"})
        .c('field', {"var": "FORM_TYPE"})
        .c('value').t(SketchCast.NS_PUBSUB_NODE_CONFIG)
        .up().up()
        .c('field', {"var": "pubsub#deliver_payloads"})
        .c('value').t("1")
        .up().up()
        .c('field', {"var": "pubsub#send_last_published_item"})
        .c('value').t("never")
        .up().up()
        .c('field', {"var": "pubsub#persist_items"})
        .c('value').t("true")
        .up().up()
        .c('field', {"var": "pubsub#max_items"})
        .c('value').t("20");
    SketchCast.connection.sendIQ(configiq,
                                 SketchCast.configured,
                                 SketchCast.configure_error);
},
create_error: function (iq) {
    SketchCast.show_error("SketchCast creation failed with " +
                          SketchCast.make_error_from_iq(iq));
}

```

code snippet sketchcast.js

pubsub 节点创建节请求一个速成节点。created()回调必须做的第一件事情就是将创建好的节点的名称保存起来。

当展示者开始绘画时，任何到来的观看者都将看到新的绘画事件，但他们在订阅之前不会获取到任何事件。为了解决这个问题，观看者必须从 pubsub 节点检索最近的项，以便为绘画事件提供一些上下文。

因为我们通常并不知道任意给定 pubsub 服务的配置默认值，所以代码必须配置 pubsub 节点来保存合理数目的历史事件。

后面的 created()回调为 pubsub 节点构建一个配置表单并提交。注意，我们不需要从服务器那里询问表单，这是因为我们已经知道需要修改什么值。该回调发送配置表单并指派 configured()

和 `configure_error()` 来分别处理成功情况和错误情况。

`create_error()` 只负责向用户显示错误消息。

下面给出了这两个回调。



可从
Wrox.com
下载源代码

```
configured: function (iq) {
    $(document).trigger("broadcast_started");
},
configure_error: function (iq) {
    SketchCast.show_error("SketchCast configuration failed with " +
        SketchCast.make_error_from_iq(iq));
}
```

code snippet sketchcast.js

当服务器确认节点成功配置时，SketchCast 触发自定义 `broadcast_started` 事件。如果在配置期间发生错误，那么将打开错误提示对话框。

这里给出了 `broadcast_started` 事件的一个空的占位符处理程序，应该将其添加到 `sketchcast.js` 文件的末尾，它负责更新状态区域并启用绘图控件。



可从
Wrox.com
下载源代码

```
$(document).bind('broadcast_started', function () {
    $('#status').html('Broadcasting at service: <i>' +
        SketchCast.service + '</i> node: <i>' +
        SketchCast.node + "</i>");

    $('.button').removeClass('disabled');
    $('#sketch').removeClass('disabled');
    $('#erase').removeAttr('disabled');
});
```

code snippet sketchcast.js

9.5.4 发布和接收绘图事件

下一个要实现的任务是根据展示者的绘图动作生成绘画事件，将这些事件发布到节点，在订阅者一方接收事件，然后将事件有效载荷翻译成绘图动作。一旦编写完这些部分，那么应用程序的剩余工作就是进一步完善它。

1. 绘图事件

表单是保存绘画事件的便利方法，但首先必须确定需要将什么数据编码以便重建绘图动作。回顾我们前面编写的 `mousemove` 事件处理程序中控制绘图的代码行。



可从
Wrox.com
下载源代码

```
// render the line segment
var ctx = $('#sketch').get(0).getContext('2d');
ctx.strokeStyle = '#' + SketchCast.color;
```

```

ctx.lineWidth = SketchCast.lineWidth;
ctx.beginPath();
ctx.moveTo(SketchCast.old_pos.x, SketchCast.old_pos.y);
ctx.lineTo(pos.x, pos.y);
ctx.stroke();

```

code snippet sketchcast.js

绘制线段只需要少量的数据，即颜色、线宽以及鼠标指针的新旧坐标。

要构建 pubsub 事件有效载荷，可以将这四部分数据编码成如下所示的表单。

```

<x xmlns='jabber:x:data' type='result'>
  <field var='color'>
    <value>f00</value>
  </field>
  <field var='line_width'>
    <value>6</value>
  </field>
  <field var='from_pos'>
    <value>50,123</value>
  </field>
  <field var='to_pos'>
    <value>72,89</value>
  </field>
</x>

```

2. 发布绘图事件

必须修改绘图代码，采用前面概述的格式将事件发布到节点。此外，只有展示者应该能够绘图，而且必须是在启用绘画区域的情况下才可以。修改 `mousemove` 事件处理程序以匹配下面给出的代码。



可从
Wrox.com
下载源代码

```

$( '#sketch' ).mousemove( function ( ev ) {
  // get the position of the drawing area, our offset
  var offset = $( this ).offset();
  // calculate our position within the drawing area
  var pos = { x: ev.pageX - offset.left,
              y: ev.pageY - offset.top };

  if ( SketchCast.pen_down ) {
    if ( !SketchCast.old_pos ) {
      SketchCast.old_pos = pos;
      return;
    }

    if ( !$( '#sketch' ).hasClass( 'disabled' ) &&
        ( Math.abs( pos.x - SketchCast.old_pos.x ) > 2 ||


```

```

Math.abs(pos.y - SketchCast.old_pos.y) > 2)) {
    // render the line segment
    var ctx = $('#sketch').get(0).getContext('2d');
    ctx.strokeStyle = '#' + SketchCast.color;
    ctx.lineWidth = SketchCast.line_width;
    ctx.beginPath();
    ctx.moveTo(SketchCast.old_pos.x, SketchCast.old_pos.y);
    ctx.lineTo(pos.x, pos.y);
    ctx.stroke();

    SketchCast.publish_action({
        color: SketchCast.color,
        line_width: SketchCast.line_width,
        from: SketchCast.old_pos,
        to: pos
    });
}

SketchCast.old_pos = pos;
}
} else {
    SketchCast.old_pos = null;
}
});

```

code snippet sketchcast.js

注意，任何线段必须至少有两个像素高或两个像素宽，这样新版代码才能够绘制它。之所以这样做是为了防止发布过量的绘图事件。在某些浏览器中，可能在同一个像素中发送多次`mousemove`事件。

接下来，将下面的代码添加到`SketchCast`对象中来实现`publish_action()`。



可从
Wrox.com
下载源代码

```

publish_action: function (action) {
    SketchCast.connection.sendIQ(
        $iq({to: SketchCast.service,
            type: "set"})
        .c('pubsub', {xmlns: SketchCast.NS_PUBSUB})
        .c('publish', {node: SketchCast.node})
        .c('item')
        .c('x', {xmlns: SketchCast.NS_DATA_FORMS,
            type: "result"})
        .c('field', {"var": "color"})
        .c('value').t(action.color)
        .up().up()
        .c('field', {"var": "line_width"})
        .c('value').t('' + action.line_width)
        .up().up()
        .c('field', {"var": "from_pos"})
        .c('value').t('' + action.from.x + ',' + action.from.y)
        .up().up()
    );
}

```

```

    .c('field', {"var": "to_pos"})
    .c('value').t('' + action.to.x + ',' + action.to.y));
}

```

code snippet sketchcast.js

这个函数看起来比较复杂，但它只是构建我们前面看过的表单并将其封装到一个携带<publish>动作的 pubsub IQ-set 节中，然后将其发送到服务器。一旦将该数据发布到节点，pubsub 服务就会将其广播到所有订阅者。

3. 接收绘图事件

我们已经创建了一个空的 on_event()处理程序，每当新的事件到达时就会执行该函数。我们现在必须补上遗漏的部分：从事件中提取数据并将其转换成画布上的一条线段。

我们将构建一个与前一节中用于 publish_action()中的似动作对象，然后将该对象发送给 render_action()函数。下面给出了新的 on_event()代码。



可从
Wrox.com
下载源代码

```

on_event: function (msg) {
    if ($(msg).find('x').length === 0) {
        // this message wasn't for us!
        return true;
    }

    var color = $(msg).find('field[var="color"] value').text();
    var line_width = $(msg).find('field[var="line_width"] value').text();
    var from_pos = $(msg).find('field[var="from_pos"] value').text()
        .split(',');
    var to_pos = $(msg).find('field[var="to_pos"] value').text()
        .split(',');

    var action = {
        color: color,
        line_width: line_width,
        from: {x: parseFloat(from_pos[0]),
                y: parseFloat(from_pos[1])},
        to: {x: parseFloat(to_pos[0]),
              y: parseFloat(to_pos[1])}
    };

    SketchCast.render_action(action);

    return true;
}

```

code snippet sketchcast.js

现在实现 render_action()并将其添加到 SketchCast 对象中。



可从
Wrox.com
下载源代码

```
render_action: function (action) {
  // render the line segment
  var ctx = $('#sketch').get(0).getContext('2d');
  ctx.strokeStyle = '#' + action.color;
  ctx.lineWidth = action.line_width;
  ctx.beginPath();
  ctx.moveTo(action.from.x, action.from.y);
  ctx.lineTo(action.to.x, action.to.y);
  ctx.stroke();
}
```

code snippet sketchcast.js

如果这段代码看起来比较熟悉，那么不要感到意外，它只是 `mousemove` 事件处理程序中的绘图代码稍微经过修改过的版本。到处复制稍微经过修改的代码是一个不好的做法，因此将旧的绘图代码替换成对 `render_action()` 函数的调用。这会把所有的绘图逻辑放在一个地方，这样更易于查找和修改，而且不会出现不同步的情况。修改 `mousemove` 事件处理程序，将原有的绘图代码替换掉。



可从
Wrox.com
下载源代码

```
$('#sketch').mousemove(function (ev) {
  // get the position of the drawing area, our offset
  var offset = $(this).offset();
  // calculate our position within the drawing area
  var pos = {x: ev.pageX - offset.left,
             y: ev.pageY - offset.top};

  if (SketchCast.pen_down) {
    if (!SketchCast.old_pos) {
      SketchCast.old_pos = pos;
      return;
    }

    if (!$('#sketch').hasClass('disabled') &&
        (Math.abs(pos.x - SketchCast.old_pos.x) > 2 || 
         Math.abs(pos.y - SketchCast.old_pos.y) > 2)) {
      SketchCast.render_action({
        color: SketchCast.color,
        line_width: SketchCast.line_width,
        from: {x: SketchCast.old_pos.x,
                y: SketchCast.old_pos.y},
        to: {x: pos.x,
              y: pos.y});

      SketchCast.publish_action({
        color: SketchCast.color,
        line_width: SketchCast.line_width,
        from: SketchCast.old_pos,
        to: pos
      });
    }
  }
});
```

```

    });
    SketchCast.old_pos = pos;
}
} else {
    SketchCast.old_pos = null;
}
});

```

code snippet sketchcast.js

现在我们已经拥有一个可以运行的 SketchCast 了。

4. 后期加入并赶上进度

当后期加入的用户到来时，他们需要从 pubsub 服务那里请求自己已经错过的绘图事件。前面曾经演示过，这很容易通过一个携带<items>命令的 IQ-get 节来实现。将下面被突出显示的代码行添加到 reception_started 事件处理程序中。



可从
Wrox.com
下载源代码

```

$(document).bind('reception_started', function () {
    $('#status').html('Receiving SketchCast.');
    // get missed events
    SketchCast.connection.sendIQ(
        $iq({to: SketchCast.service, type: "get"})
            .c('pubsub', {xmlns: SketchCast.NS_PUBSUB})
            .c('items', {node: SketchCast.node}),
        SketchCast.on_old_items);
});

```

code snippet sketchcast.js

现在，将 on_old_items() 的实现添加到 SketchCast 对象中。



可从
Wrox.com
下载源代码

```

on_old_items: function (iq) {
    $(iq).find('item').each(function () {
        SketchCast.on_event(this);
    });
}

```

code snippet sketchcast.js

因为 on_event() 处理程序值查找包含绘图动作的表单，所以可以只把所有的<item>子元素发送给它。据 on_event() 所知，它就像任何其他事件一样来接收动作。

当一个后到者加入一个正在进行中的绘画会话中时，他们将接收到过去的绘图动作，这样他们就有一定的上下文来帮助理解他们看到的新的绘画部分。在这里，他们将接收到前面的 20 个动作，这是因为节点配置中设置存储最多 20 项。注意，有些 XMPP 服务器限制能够存储的项数，因此如果需要大量存储，那么可能必须特别地配置服务器来支持该应用程序。

5. 优雅地终止会话

为了完成 SketchCast，应该确保展示者和观看者能够优雅地退出。为了在退出后进行清理，应用程序必须将 pubsub 节点删除(如果该用户是展示者的话)和退订(如果该用户是观看者的话)。在这两种情况下都还应该断开连接并将应用程序重置到开始时的状态。

首先处理发布者情况，实现工具栏中的 Disconnect 按钮。为该按钮添加处理程序的最佳时机是当获得 broadcast_started 事件时。当接收到该事件时，我们知道该用户是一个发布者，而且刚刚创建了一个 pubsub 节点。下面给出了修改过的事件处理程序。



可从
Wrox.com
下载源代码

```

$(document).bind('broadcast_started', function (ev, data) {
    $('#status').html('Broadcasting at service: <i>' +
        SketchCast.service + '</i> node: <i>' +
        data.node + "</i>");
    $('.button').removeClass('disabled');
    $('#sketch').removeClass('disabled');
    $('#erase').removeAttr('disabled');
    $('#disconnect').removeAttr('disabled');

    $('#disconnect').click(function () {
        $('.button').addClass('disabled');
        $('#sketch').addClass('disabled');
        $('#erase').attr('disabled', 'disabled');
        $('#disconnect').attr('disabled', 'disabled');

        SketchCast.connection.sendIQ(
            $iq({to: SketchCast.service,
                  type: "set"})
                .c('pubsub', {xmlns: SketchCast.NS_PUBSUB_OWNER})
                .c('delete', {node: SketchCast.node}));

        SketchCast.disconnect();
    });
}) ;
}
)

```

code snippet sketchcast.js

该代码绑定 Disconnect 按钮的事件，使用<delete> pubsub 动作来删除 pubsub 节点，然后断开连接。现在必须编写 disconnect() 函数并将其添加到 SketchCast 对象中。



可从
Wrox.com
下载源代码

```

disconnect: function () {
    $('#erase').click();
    SketchCast.connection.disconnect();
    SketchCast.connection = null;
    SketchCast.service = null;
    SketchCast.node = null;
    $('#login_dialog').dialog('open');
}

```

code snippet sketchcast.js

在断开 XMPP 流之后，该代码重置应用程序状态，清除绘图区域，并打开登录对话框。然后 SketchCast 准备好运行下一回合。

观看者的断开连接顺序稍微复杂一些。必须从 pubsub 节点中退订并断开连接，这与发布者的动作非常相似，但绘画的发布者有可能已经在所有订阅者之前断开连接。在这种情况下，pubsub 服务会把节点的删除通知给用户，而我们必须扩展 on_event() 函数来处理这种新的事件类型。

首先，一旦接收到 reception_started 事件就立即启用断开连接按钮，就像我们为发布者一方所做的那样。下面给出了修改后的事件处理程序。



可从
Wrox.com
下载源代码

\$(document).bind('reception_started', function (ev, data) {
 \$('#status').html('Receiving SketchCast.');//
 \$('#disconnect').removeAttr('disabled');//
 \$('#disconnect').click(function () {
 \$('#disconnect').attr('disabled', 'disabled');//
 SketchCast.connection.sendIQ(
 \$iq({to: SketchCast.service,
 type: "set"})
 .c('pubsub', {xmlns: SketchCast.NS_PUBSUB_OWNER})
 .c('unsubscribe', {node: SketchCast.node
 jid: SketchCast.connection.jid});
 SketchCast.disconnect();
 });

 // get missed events
 SketchCast.connection.sendIQ(
 \$iq({to: SketchCast.service, type: "get"})
 .c('pubsub', {xmlns: SketchCast.NS_PUBSUB})
 .c('items', {node: SketchCast.node}),
 SketchCast.on_old_items);
});

code snippet sketchcast.js

接下来，必须修改 pubsub 事件处理程序 on_event()。如果 pubsub 节点被删除而用户仍然处理订阅状态，那么显示一条错误提示信息。将旧的处理程序替换成下面的代码。



可从
Wrox.com
下载源代码

```
on_event: function (msg) {  
    if ($(msg).find('x').length > 0) {  
        var color = $(msg).find('field[var="color"] value').text();  
        var line_width = $(msg).find('field[var="line_width"] value').text();  
        var from_pos = $(msg).find('field[var="from_pos"] value').text()  
            .split(',');  
        var to_pos = $(msg).find('field[var="to_pos"] value').text()  
            .split(',');  
  
        var action = {  
            color: color,
```

```

        line_width: line_width,
        from: {x: parseFloat(from_pos[0]),
                y: parseFloat(from_pos[1])},
        to: {x: parseFloat(to_pos[0]),
              y: parseFloat(to_pos[1])}
    };

    SketchCast.render_action(action);
} else if ($(msg).find('delete[node="' + SketchCast.node + '"]')
    .length > 0) {
    SketchCast.show_error("SketchCast ended by presenter.");
}

return true;
}

```

code snippet sketchcast.js

新版本的代码首先查找绘画绘制事件，如果没有找到，那么它将查找节点删除通知。全部工作已经完成，现在可以正式推出它了。程序清单 9-3 给出了完整的 JavaScript 文件。



可从
Wrox.com
下载源代码

程序清单 9-3 sketchcast.js

```

var SketchCast = {
    // drawing state
    pen_down: false,
    old_pos: null,
    color: '000',
    line_width: 4,

    // xmpp state
    connection: null,
    service: null,
    node: null,

    // namespace constants
    NS_DATA_FORMS: "jabber:x:data",
    NS_PUBSUB: "http://jabber.org/protocol/pubsub",
    NS_PUBSUB_OWNER: "http://jabber.org/protocol/pubsub#owner",
    NS_PUBSUB_ERRORS: "http://jabber.org/protocol/pubsub#errors",
    NS_PUBSUB_NODE_CONFIG: "http://jabber.org/protocol/pubsub#node_config",

    on_event: function (msg) {
        if ($(msg).find('x').length > 0) {
            var color = $(msg).find('field[var="color"] value').text();
            var line_width = $(msg)
                .find('field[var="line_width"] value').text();
            var from_pos = $(msg).find('field[var="from_pos"] value').text()
                .split(',');

```

```

        var to_pos = $(msg).find('field[var="to_pos"] value').text()
                    .split(',');
        var action = {
            color: color,
            line_width: line_width,
            from: {x: parseFloat(from_pos[0]),
                    y: parseFloat(from_pos[1])},
            to: {x: parseFloat(to_pos[0]),
                  y: parseFloat(to_pos[1])}
        };
        SketchCast.render_action(action);
    } else if ($(msg).find('delete[node=' + SketchCast.node + ']')
               .length > 0) {
        SketchCast.show_error("SketchCast ended by presenter.");
    }
    return true;
},
on_old_items: function (iq) {
    $(iq).find('item').each(function () {
        SketchCast.on_event(this);
    });
},
// subscription callbacks
subscribed: function (iq) {
    $(document).trigger("reception_started");
},
subscribe_error: function (iq) {
    SketchCast.show_error("Subscription failed with " +
        SketchCast.make_error_from_iq(iq));
},
// error handling helpers
make_error_from_iq: function (iq) {
    var error = $(iq)
        .find('*[xmlns=' + Strophe.NS.STANZAS + '"]')
        .get(0).tagName;
    var pubsub_error = $(iq)
        .find('*[xmlns=' + SketchCast.NS_PUBSUB_ERRORS + '"]');
    if (pubsub_error.length > 0) {
        error = error + "/" + pubsub_error.get(0).tagName;
    }
    return error;
},
show_error: function (msg) {
    SketchCast.connection.disconnect();
    SketchCast.connection = null;
}

```

```

SketchCast.service = null;
SketchCast.node = null;

$('#error_dialog p').text(msg);
$('#error_dialog').dialog('open');
},

// node creation callbacks
created: function (iq) {
    // find pubsub node
    var node = $(iq).find("create").attr('node');
    SketchCast.node = node;

    // configure the node
    var configiq = $iq({to: SketchCast.service,
                        type: "set"})
        .c('pubsub', {xmlns: SketchCast.NS_PUBSUB_OWNER})
        .c('configure', {node: node})
        .c('x', {xmlns: SketchCast.NS_DATA_FORMS,
                  type: "submit"})
        .c('field', {"var": "FORM_TYPE"})
        .c('value').t(SketchCast.NS_PUBSUB_NODE_CONFIG)
        .up().up()
        .c('field', {"var": "pubsub#deliver_payloads"})
        .c('value').t("1")
        .up().up()
        .c('field', {"var": "pubsub#send_last_published_item"})
        .c('value').t("never")
        .up().up()
        .c('field', {"var": "pubsub#persist_items"})
        .c('value').t("true")
        .up().up()
        .c('field', {"var": "pubsub#max_items"})
        .c('value').t("20");
    SketchCast.connection.sendIQ(configiq,
                                 SketchCast.configured,
                                 SketchCast.configure_error);
},

create_error: function (iq) {
    SketchCast.show_error("SketchCast creation failed with " +
                         SketchCast.make_error_from_iq(iq));
},

configured: function (iq) {
    $(document).trigger("broadcast_started");
},

configure_error: function (iq) {
    SketchCast.show_error("SketchCast configuration failed with " +
                         SketchCast.make_error_from_iq(iq));
},

```

```

publish_action: function (action) {
  SketchCast.connection.sendIQ(
    $iq({to: SketchCast.service, type: "set"})
      .c('pubsub', {xmlns: SketchCast.NS_PUBSUB})
      .c('publish', {node: SketchCast.node})
      .c('item')
      .c('x', {xmlns: SketchCast.NS_DATA_FORMS,
                type: "result"})
      .c('field', {"var": "color"})
      .c('value').t(action.color)
      .up().up()
      .c('field', {"var": "line_width"})
      .c('value').t('' + action.line_width)
      .up().up()
      .c('field', {"var": "from_pos"})
      .c('value').t('' + action.from.x + ',' + action.from.y)
      .up().up()
      .c('field', {"var": "to_pos"})
      .c('value').t('' + action.to.x + ',' + action.to.y));
  },
  render_action: function (action) {
    // render the line segment
    var ctx = $('#sketch').get(0).getContext('2d');
    ctx.strokeStyle = '#' + action.color;
    ctx.lineWidth = action.line_width;
    ctx.beginPath();
    ctx.moveTo(action.from.x, action.from.y);
    ctx.lineTo(action.to.x, action.to.y);
    ctx.stroke();
  },
  disconnect: function () {
    $('#erase').click();
    SketchCast.connection.disconnect();
    SketchCast.connection = null;
    SketchCast.service = null;
    SketchCast.node = null;
    $('#login_dialog').dialog('open');
  }
};

$(document).ready(function () {
  $('#login_dialog').dialog({
    autoOpen: true,
    draggable: false,
    modal: true,
    title: 'Connect to a SketchCast',
    buttons: {
      "Connect": function () {
        $(document).trigger('connect', {

```

```

        jid: $('#jid').val(),
        password: $('#password').val(),
        service: $('#service').val(),
        node: $('#node').val()
    });

    $('#password').val('');
    $(this).dialog('close');
}
}

$('#error_dialog').dialog({
    autoOpen: false,
    draggable: false,
    modal: true,
    title: 'Whoops! Something Bad Happened!',
    buttons: {
        "Ok": function () {
            $(this).dialog('close');
            $('#login_dialog').dialog('open');
        }
    }
});

$('#sketch').mousedown(function () {
    SketchCast.pen_down = true;
});

$('#sketch').mouseup(function () {
    SketchCast.pen_down = false;
});

$('#sketch').mousemove(function (ev) {
    // get the position of the drawing area, our offset
    var offset = $(this).offset();
    // calculate our position within the drawing area
    var pos = {x: ev.pageX - offset.left,
               y: ev.pageY - offset.top};

    if (SketchCast.pen_down) {
        if (!SketchCast.old_pos) {
            SketchCast.old_pos = pos;
            return;
        }

        if (!$('#sketch').hasClass('disabled') &&
            (Math.abs(pos.x - SketchCast.old_pos.x) > 2 ||

            Math.abs(pos.y - SketchCast.old_pos.y) > 2)) {
            SketchCast.render_action({
                color: SketchCast.color,
                line_width: SketchCast.line_width,
                from: {x: SketchCast.old_pos.x,

```

```

                y: SketchCast.old_pos.y},
                to: {x: pos.x,
                      y: pos.y}});
        SketchCast.publish_action({
            color: SketchCast.color,
            line_width: SketchCast.line_width,
            from: SketchCast.old_pos,
            to: pos
        });
        SketchCast.old_pos = pos;
    }
} else {
    SketchCast.old_pos = null;
}
});

$('.color').click(function (ev) {
    SketchCast.color = $(this).attr('id').split('-')[1];
});

$('.linew').click(function (ev) {
    SketchCast.line_width = $(this).attr('id').split('-')[1];
});

$('#erase').click(function () {
    var ctx = $('#sketch').get(0).getContext('2d');
    ctx.fillStyle = '#fff';
    ctx.strokeStyle = '#fff';
    ctx.fillRect(0, 0, 600, 500);
});
});

$(document).bind('connect', function (ev, data) {
    $('#status').html('Connecting..');

    var conn = new Strophe.Connection(
        'http://bosh.metajack.im:5280/xmpp-httpbind');
    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });
    SketchCast.connection = conn;
    SketchCast.service = data.service;
    SketchCast.node = data.node;
});

$(document).bind('connected', function () {
    $('#status').html("Connected.");
});

```

```

// send negative presence send we're not a chat client
SketchCast.connection.send($pres().c('priority').t('-1'));

if (SketchCast.node.length > 0) {
    // a node was specified, so we attempt to subscribe to it
    // first, set up a callback for the events
    SketchCast.connection.addHandler(
        SketchCast.on_event,
        null, "message", null, null, SketchCast.service);
    // now subscribe
    var subiq = $iq({to: SketchCast.service,
                    type: "set"})
        .c('pubsub', {xmlns: SketchCast.NS_PUBSUB})
        .c('subscribe', {node: SketchCast.node,
                        jid: SketchCast.connection.jid});
    SketchCast.connection.sendIQ(subiq,
                                 SketchCast.subscribed,
                                 SketchCast.subscribe_error);
} else {
    // a node was not specified, so we start a new sketchcast
    var createiq = $iq({to: SketchCast.service,
                        type: "set"})
        .c('pubsub', {xmlns: SketchCast.NS_PUBSUB})
        .c('create');
    SketchCast.connection.sendIQ(createiq,
                                 SketchCast.created,
                                 SketchCast.create_error);
}
});

$(document).bind('broadcast_started', function () {
    $('#status').html('Broadcasting at service: <i>' +
                      SketchCast.service + '</i> node: <i>' +
                      SketchCast.node + "</i>");
    $('.button').removeClass('disabled');
    $('#sketch').removeClass('disabled');
    $('#erase').removeAttr('disabled');
    $('#disconnect').removeAttr('disabled');

    $('#disconnect').click(function () {
        $('.button').addClass('disabled');
        $('#sketch').addClass('disabled');
        $('#erase').attr('disabled', 'disabled');
    });
});

```

```

        $('#disconnect').attr('disabled', 'disabled');

        SketchCast.connection.sendIQ(
            $iq({to: SketchCast.service,
                  type: "set"})
                .c('pubsub', {xmlns: SketchCast.NS_PUBSUB_OWNER})
                .c('delete', {node: SketchCast.node}));
        SketchCast.disconnect();
    });
});

$(document).bind('reception_started', function () {
    $('#status').html('Receiving SketchCast.');

    $('#disconnect').removeAttr('disabled');
    $('#disconnect').click(function () {
        $('#disconnect').attr('disabled', 'disabled');
        SketchCast.connection.sendIQ(
            $iq({to: SketchCast.service,
                  type: "set"})
                .c('pubsub', {xmlns: SketchCast.NS_PUBSUB_OWNER})
                .c('unsubscribe', {node: SketchCast.node,
                                   jid: SketchCast.connection.jid}));
        SketchCast.disconnect();
    });

    // get missed events
    SketchCast.connection.sendIQ(
        $iq({to: SketchCast.service, type: "get"})
            .c('pubsub', {xmlns: SketchCast.NS_PUBSUB})
            .c('items', {node: SketchCast.node}),
        SketchCast.on_old_items);
});
});

```

9.6 改进 SketchPad

软件总是可以不断改进。您是否希望向 SketchCast 中添加一些新功能？市场营销部门告诉我们前三个用户需求是：

- 显示在浏览器订阅之前发生的绘图事件。
- 使用户能够采用不同的字体在白板上书写。
- 显示可用的 SketchCast 服务，而不是输入它们的名称。

9.7 小结

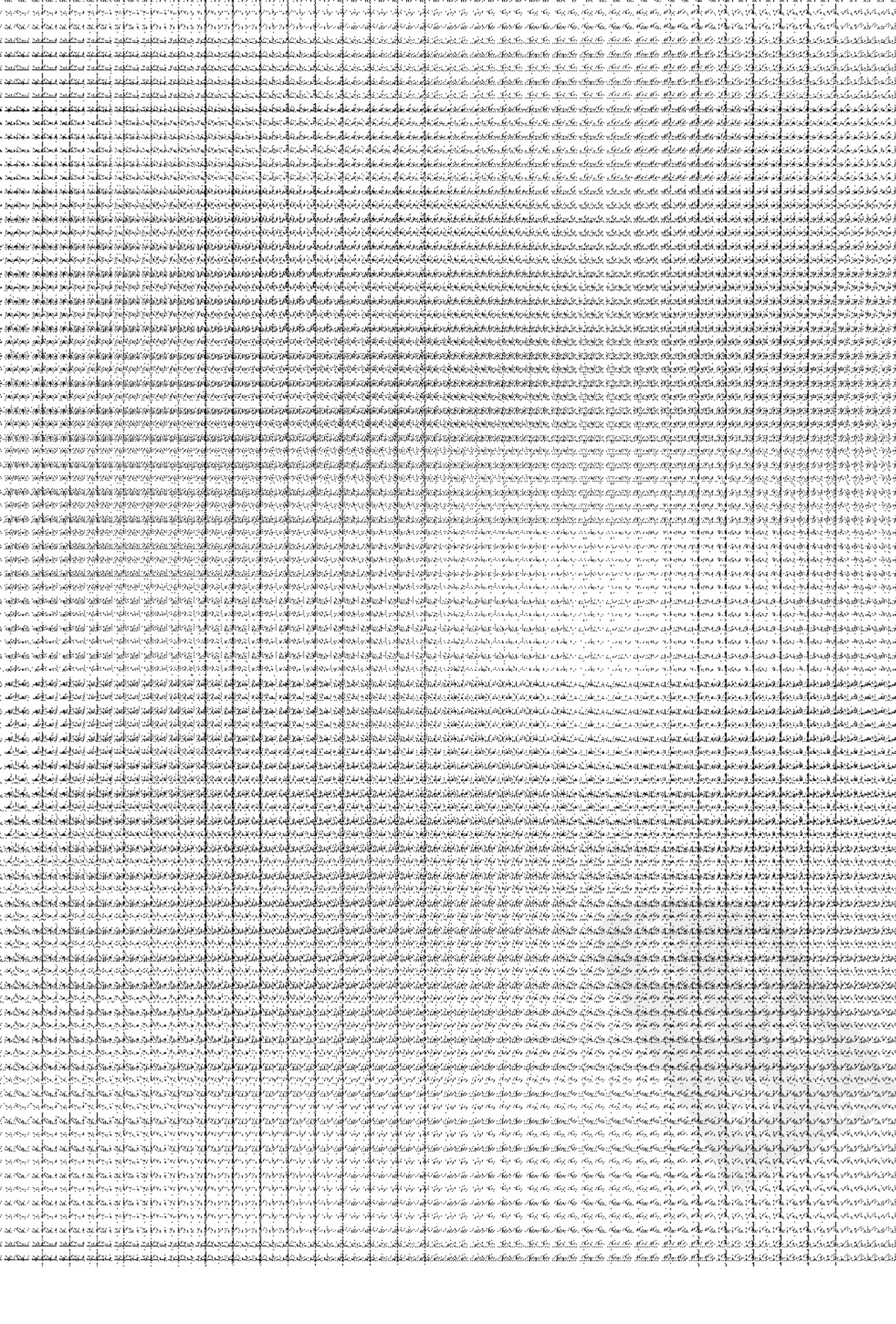
XMPP pubsub 是一种极为强大的工具。即使是最简单的形式，它也能够让创建类似

SketchCast这样的应用程序变得相当简单。在本章中，我们使用 XMPP pubsub 来开发一个绘图展示应用程序，并学习了如下内容：

- pubsub 基础知识
- 创建和使用表单
- 从 pubsub 节点订阅、退订和获取项
- 创建、配置和删除 pubsub 节点
- 发布和处理 pubsub 事件
- HTML5 的<canvas>元素

在第 10 章中，我们将看到一个真正的协作式应用程序，它可以让两个用户编辑一份共享的文档，我们将从中学习到如何利用自己的元素来扩展 XMPP 节。





第 10 章

与好友一同写作：协作式文本编辑器

本章内容

- 学习操作转换算法
- 使用服务发现来探测功能
- 扩展 XMPP 节

人们曾经通过来回邮寄手稿来进行协作。多年来，随着技术的改进，这变成了通过硬盘的文件共享方式，然后是通过电子邮件进行共享。现在的技术已经达到一个高度，多人(甚至在大型组织中)能够通过网络同时编辑同一份文档，而且人们对 Web 的熟悉使得它成为非常容易和常见的事情。有了前面几次 XMPP 应用程序的经历，再看到 XMPP 能够为协作式文档编辑提供理想的媒介，您应该不会感到奇怪。

这类应用程序最新的一个示例就是 Google Wave，这是一个丰富的、充满各种媒体的协作式的、可供一组人同时处理共享文档的空间。当 Google 需要一种协议来促进这类平台时，它转向了 XMPP。在 XMPP 之上，Google 为基于操作转换(operational transformation)原理的协作式编辑构建了一个梦幻般的平台。它还选择以开放的方式(<http://www.waveprotocol.org>)来完成 Wave 协议中的协议设计工作。

在本章中，我们将基于与 Google Wave 一样的原理(操作转换)构建一个协作式文本编辑器。尽管这个应用程序是为一对用户之间的写作而设计的，但底层的算法可以扩展到任意数目的协作者。通过这个应用程序以及其他应用程序，我们可以看出在 XMPP 的核心技术之上构建复杂的软件是一件多么简单的事情。

10.1 应用程序预览

图 10-1 给出了 NetPad 应用程序的最终版本。它的 UI 非常简单，但其功能却非常强大。



图 10-1

一个硕大的编辑区域占据了屏幕的大部分空间。顶部有一个状态显示区域和一个“断开连接”按钮，而聊天区域出现在底部。两个协作者不仅能够处理同一段文本，而且他们还能够彼此交谈以交流各自的想法。

10.2 NetPad 的设计

与任何文本编辑器类似，NetPad 的主要功能是编辑文本。虽然 NetPad 很容易接受来自本地的对文本的编辑操作，但它还必须支持来自远程用户的编辑操作。构建 NetPad 所涉及的大部分工作是支持这些远程操作。聊天区域是第 6 章 Gab 应用程序功能的精简版，而且到目前为止，您应该已经熟悉该应用程序所需的个人通信部分的功能。

为了让用户以协作方式编辑同一段文本，NetPad 运用了操作转换理论。本地的编辑动作被表示为对文本的操作并发送给远端的另一方。接收远程操作并转换成稍微经过修改的版本，在应用这些修改后的版本之后，双方最终会看到该文本的一致的视图。

NetPad 必须在一对用户之间建立一个编辑会话，将本地操作发送给另一方，并处理传入的远程操作。这些操作将编码为 XML 数据(作为 XMPP 协议扩展)。应用程序使用<presence>节来建立会话，使用<message>节来通告编辑操作。

当然，并不是每位用户都能够以协作方式编辑 NetPad 文档，因此 NetPad 必须首先确定潜在的协作者的客户端是否支持这种编辑功能。这可以通过服务发现(在第 7 章中已经讲过)来完成。

NetPad 的设计非常简单，但操作转换和 XMPP 协议扩展都值得深入研究。一旦讲解了这些概念，我们就可以为设计 NetPad 协议并构建 NetPad 应用程序做好准备。

10.3 操作转换

Ellis 和 Gibbs 在他们的论文 *Concurrency Control in Groupware Systems* 中首先提出了操作转换背后的概念。这篇论文是学术论文的杰出典范，极具可读性，它将一个复杂的主题转换成简单的描述。NetPad 使用的算法直接基于这些优秀研究者的工作之上。

协作编辑是实时群件(groupware)的一种形式。这类应用程序有许多有趣的重要性质。首先，它们是高度交互式、实时的分布式网络应用程序。这些性质也都体现在本书中的大多数应用程序中。群件应用程序本质上是自组织的而且是不稳定的，人们无法弄清楚用户将要采取哪些动作或者他们将按照什么顺序来执行这些动作，而且用户来去自由。最后，这些类型的应用程序相当专注，系统的每位用户都在与同一个数据进行交互，而且它们所做的修改通常会彼此冲突。

群件应用程序的主要困难在于一致性。一个用户看到的文档视图应该尽可能接近于其他用户看到的相同信息的视图。当编辑操作停止时，最终的文档应该完全相同。如果用户会话终止时每个人得到的结果却不同，那么协同工作就没有带来任何好处。

虽然并发问题并不是群件应用程序特有的，但这些系统的实时性和分布式功能使得那些针对并发问题的最常见的解决方案变得不再可行。例如，数据库管理系统通常支持不同形式的并发控制，但它们是通过完全禁止可能引起冲突的操作来实现的。其他系统则只允许一次一位编辑者，并将该权限从一个用户传给另一个用户。群件要求的低延迟和交互性迫使我们寻求一种不同的解决方法。

10.3.1 基本原理

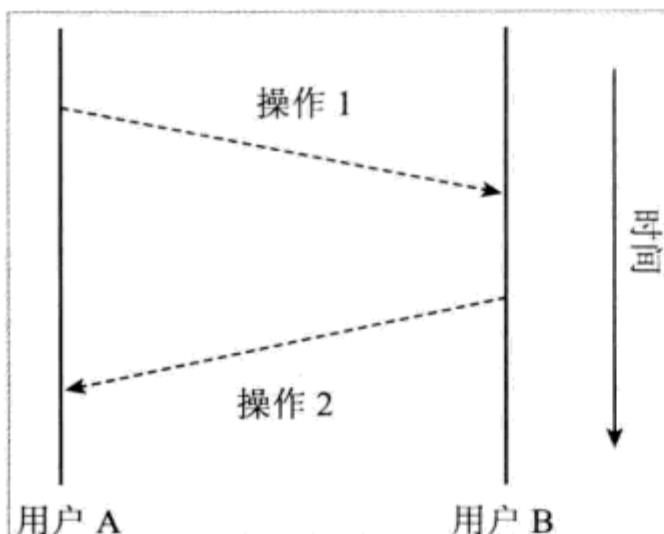
操作转换算法涉及一组用户、这些用户能够执行的操作以及每个用户按照什么顺序来执行这些操作。

NetPad 所需的操作集合非常简单，即插入一个字符和删除一个字符。插入操作携带两个实参，分别是待插入的字符和将其放入的位置。这个操作写作 `insert(pos, char)`。删除操作只需要一个位置参数(被删除字符所在的位置)，写作 `delete(pos)`。

每个用户的编辑动作都会生成一个由这两种操作组成的序列。重要的是这组操作要按照适当的顺序执行。例如，考虑 4 个操作：`insert(0,'c')`、`insert(0,'b')`、`insert(0,'a')`和 `delete(0)`。交换这些操作的顺序将产生不同的字符串。按照前面列出的顺序，这些操作将产生字符串 `bc`。而如果交换中间两个操作的顺序，那么最终的字符串将变成 `ac`。

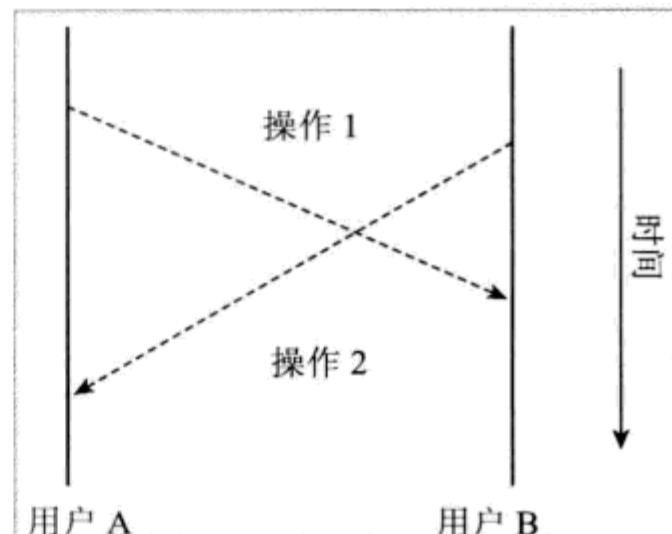
每位用户的操作都是有序的，而其他用户将按照相同的顺序来执行它们。此外，总的操作集合也有一个顺序，但与本地操作不同的是，这种排序并不完全。有可能两个用户同时在同一个字符串上执行不同的操作。这些操作是重叠的，不能按照彼此的关系进行排序。

图 10-2 和图 10-3 演示了这两种情况。在两幅图中，每个用户(u_1 和 u_2)都各自产生了一个操作 o_1 和 o_2 。在图 10-2 中，在他们产生 o_2 之前， o_1 已由 u_2 执行。但在图 10-3 中， u_1 和 u_2 执行了重叠的同时操作。



改编自 Ellis 和 Gibbs 的 *Concurrency Control in groupware Systems*(ACM Sigmod Record, ACM 18(2), 1989)中的图 2。

图 10-2



改编自 Ellis 和 Gibbs 的 *Concurrency Control in groupware Systems*(ACM Sigmod Record, ACM 18(2), 1989)中的图 2。

图 10-3

操作重叠的情况是协作编辑问题真正的困难之处。通过记录操作的顺序并强制要求按照正确顺序执行就可以轻易地处理非重叠的情况。但重叠操作没有顺序。那么如何执行这些操作呢？

必须将重叠操作转换成新的可以排序的操作，而这正是名称“操作转换”的由来。因为 u_1 知道 o_2 操作的状态，所以他能够将 o_2 转换成一个考虑本地操作 o_1 的操作。类似地， u_2 能够转换 o_1 以说明 o_2 所做的修改。

假设 u_1 和 u_2 的编辑器中都有字符串 ace。每个用户同时执行一个插入操作，其中 o_1 (由 u_1 执行)为 $insert(1, 'b')$ ， o_2 (由 u_2 执行)为 $insert(2, 'd')$ 。

首先，观察在不转换这些操作的时候会发生什么情况。第一位用户执行 o_1 ，然后接收并执行 o_2 ，最终的字符串是 abdce。而第二位用户执行 o_2 ，然后 o_1 ，最终得到字符串 abcde。每位用户现在得到的数据视图不一致，而只有它们中的一个版本是正确的。

现在，观察当每位用户对它们接收到的操作进行转换之后的结果。第一位用户执行 o_1 ，然后接收到 o_2 。因为该用户已经向字符串中添加了一个字符，所以他应用一次转换将 o_2 转换成 o_2' ，将插入操作的位置加 1， $insert(2, 'd')$ 变成了 $insert(3, 'd')$ 。 u_2 接收到的 o_1 操作并不需要转换，这是因为它引用的位置并没有被 o_2 改变， o_1' 与 o_1 一样。应用 o_1 、 o_2' 或 o_2 、 o_1' 最终得到相同的结果，即字符串 abcde。

类似的转化可以在任何的一对操作之间进行。

10.3.2 算法细节

通过上面的学习我们得知，操作转换以一种相对简单和直接的方式解决了一个复杂的问题。但在 NetPad 中实现该算法之前，还有几个重要的细节需要考虑。我们必须定义操作请求的结构。

我们需要特别注意一组操作的执行，而该算法运用优先级概念解决了这个问题。此外，每位用户有必要去维护一个待执行操作队列，目的是为了知道给定操作施加的状态，并追踪已经执行过的操作的历史记录。

1. 状态

每位用户必须维护一个状态向量，参与会话的每位用户分别对应一个元素。每个NetPad 会话中将有两个参与者，因此两个用户的状态向量均有两个元素。状态向量的元素是简单的从零开始的计数器。对于每个执行过的操作，状态向量中的操作发起者都将递增 1。例如，如果状态向量中的第一个位置代表本地用户，而第二个位置代表远程用户，那么状态向量 $<10, 23>$ 表示在本地文档上曾经执行了 10 个本地产生的操作和 23 个远程产生的操作。

每位用户在状态向量中的位置必须在所有状态向量中保持一样。这很容易通过对他们的 JID 按照字典顺序排序来实现。

2. 操作请求

当任何一个用户执行了一个本地操作，他们都会向其他用户广播一个相应的操作请求。操作请求可以表示为四元素向量 $<j, s, o, p>$ 的形式。第一个元素 j 是发起者在状态向量中的位置。下一个元素 s 表示在该操作产生时第 j 个用户的状态向量。元素 o 是请求的操作，而 p 是该操作的优先级。

每个请求不仅携带操作及其参数，而且还携带状态向量(当该操作第一次执行时)以及它的优先级。这种额外的信息可以让其他参与者来确定他们什么时候能够执行该操作。例如，一位用户可能在第二位用户之后产生了一个操作，而第三位用户(他首先接收到第二位用户的操作)必须等接收到第一位用户的操作之后才能执行其中一个操作。如果不知道操作的相应状态，就无法正确为执行排序。

3. 请求队列和执行日志

每位用户必须为传入的请求维护一个队列。所有请求(包括本地和远程)都将追加到该队列，而用户必须遍历这个队列来执行那些已经准备好的操作。如果用户的当前状态向量小于或等于某个排队的请求的状态向量，那么这个操作就准备好执行了。

状态向量的比较方式较为特殊，我们将在第 10.3.3 节描述。如果两个状态向量相等，那么该操作是在同一个状态上进行，而且可以执行而无须任何转换。如果请求的状态向量小于当前本地状态向量，那么某些重叠操作已经被执行了而且该请求的操作在执行之前必须经过转换。

当任何请求的操作执行之后，该请求将被追加到用户的执行日志中。该日志用于转换重叠操作(被不同的用户同时执行的操作)，将考虑已经被执行过的操作。将按照字典倒序遍历执行日志来查找那些影响到给定请求的操作。

4. 优先级

每个请求都有一个优先级，这是该操作在它的起源地执行时计算得到的值。由于这些优先级必须把前面在同一位置上执行过的那些操作的优先级考虑进去，因此这样它们由两部分组成。第一部分是执行日志中最近的请求(它的操作的起始位置等于当前操作的位置)的优先级。第二部分是用户在状态向量中的位置。如果执行日志中没有适用的请求，那么该优先级将只由用户在状态向量中的位置组成。

这听起来可能有点复杂，但为了让算法完美地运行，这是有必要的。使用某种更简化的优

先级方案并不能适用于所有情况，因此需要这种方案。上述论文中提供了这种方法的正确性证明的概要，该论文还说明更简化的方案是不可行的。

10.3.3 实现

操作转换的实现直接取自前面提到的那篇论文。这里配合源代码讲解了实现细节，但如果需要了解更多细节或对相关内容感到好奇，可以阅读原论文。

1. 初始代码

首先设置 OpTrans 对象以及它的基本属性和初始化函数。将下面的代码放入新文件 optrans.js 中。



可从
Wrox.com
下载源代码

```
var OpTrans = {
    log: null, // the request log
    queue: null, // the request queue
    state: null, // our state vector
    jid: null, // our jid
    jid_map: null, // maps jids to positions in state vectors
    buffer: null, // the text buffer being affected by operations
    update_func: null, // callback function

    init: function (jids, buffer, update_func) {
        this.log = [];
        this.queue = [];
        this.state = [];
        this.jid_map = {};
        this.buffer = buffer.split('');
        this.jid = jids[0];
        this.update_func = update_func;

        $.each(jids.sort(), function (i) {
            OpTrans.jid_map[this] = i;
            OpTrans.state.push(0);
        });
    }
};
```

code snippet optrans.js

函数的参数是参与者 JID 列表、初始缓冲区内容以及一个回调函数。JID 列表中的第一项必须是本地用户的 JID。初始缓冲区通常是一个空字符串，但如果在其他用户连接进来之前，主持人用户已经完成了部分编辑工作，那么可以将这个部分完成的文档作为初始缓冲区。每当操作执行时都会调用这个回调函数，同时传入新的缓冲区以及一个标记用来指示更新来自本地用户还是远程用户。

init() 函数将各种属性设置为各自的默认值。执行日志和请求队列开始时为空。传入该函数的 JID 列表用来创建 jid_map 属性，它将每个 JID 映射到状态向量中的对应索引。每个状态向量的长度均与 JID 列表相同，而初始状态向量在开始时每个值均设置为 0。jid 属性保存着本地

用户的 JID。

会话中的每位用户在会话开始时都将初始化各自的 OpTrans 对象。除了 jid 属性(每个用户的值都不一样)之外，所有的参与者在开始时都有相同的初始状态。

2. 比较函数

为了确定操作的优先顺序，需要将一个状态向量与另一个状态向量进行比较。类似地，优先级值也需要进行比较。因为这两种对象并不是简单的数字或字符串，所以我们将为它们编写专用的比较函数。

将下面的 compare_state()和 compare_priority()函数代码添加到 ObTrans 对象中。



可以从
Wrox.com
下载源代码

```

compare_state: function (s1, s2) {
    var i, smaller = false;
    for (i = 0; i < s1.length; i++) {
        if (s1[i] > s2[i]) {
            return 1;
        } else if (s1[i] < s2[i]) {
            smaller = true;
        }
    }
    if (smaller) {
        return -1;
    }
    return 0;
},

compare_priority: function (p1, p2) {
    if (p1.length > p2.length) {
        return -1;
    } else if (p1.length < p2.length) {
        return 1;
    } else {
        var i;
        for (i = 0; i < p1.length; i++) {
            if (p1[i] > p2[i]) {
                return -1;
            } else if (p1[i] < p2[i]) {
                return 1;
            }
        }
    }
    return 0;
}

```

code snippet optrans.js

状态向量是被逐个元素进行比较的。状态向量 s_i 等于另一个状态向量 s_j 的条件是它们的所有元素均相等。如果 s_j 的所有元素均小于或等于 s_j ，但 s_i 中至少有一个元素小于 s_j 中对应的元素，那么 s_i 小于 s_j 。否则， s_i 就被认为是大于 s_j 。

优先级也是按照元素来进行比较的。虽然状态向量总是等长的，但优先级却可以无限长。两个优先级相同的条件是它们的长度相同而且每个元素相同。否则，其中一个优先级是另一个优先级的子列表，或者它们在某个特定元素上不同。在前一种情况下，较长的优先级较大。在后一种情况下，第一个不同元素的比较结果就是优先级比较的结果。

3. 处理请求

操作转换的第一步就是处理请求。请求可能是本地用户操作的结果，有可能是通过网络从另一位用户那里接收到的。对于本地操作，必须计算优先级并构造一个请求。远程操作在被发送之前被封装到一个请求中。在这两种情况下，都必须把请求添加到请求队列中，然后就进入执行阶段。

将下面的 `do_local()` 和 `do_remote()` 函数添加的 `ObTrans` 对象中。



可以从
Wrox.com
下载源代码

```

do_local: function (op, pos, chr) {
    // calculate p
    var l, maxp = [];
    for (l = 0; l < this.log.length; l++) {
        if (this.log[l][4] === pos) {
            if (this.compare_priority(maxp, this.log[l][3]) == 1) {
                maxp = this.log[l][3];
            }
        }
    }
    var p = maxp.concat(this.jid_map[this.jid]);
    // append request to queue
    var req = [this.jid_map[this.jid],
               this.state.concat(),
               [op, pos, chr],
               p];
    this.queue.push(req);
    this.execute();
    return req;
},
do_remote: function (jid, state, op, pos, chr, pri) {
    // append request
    var req = [this.jid_map[jid], state, [op, pos, chr], pri];
    this.queue.push(req);
    this.execute();
}

```

code snippet optrans.js

对于 `do_local()`，只传入了操作的细节，但对于 `do_remote()` 来说，我们必须传入请求的所有必要组成部分，它们将用来创建一个请求对象。

由于要计算优先级，`do_local()` 函数要比 `do_remote()` 函数更加复杂。`do_local()` 函数还必须将构造好的请求返回给调用者，以便将该请求广播给远程用户。在第 10.3.2 节中我们已经描述过如何计算优先级。

NetPad 在编辑会话启动时立即调用 `init()`，然后又调用 `do_local()` (当本地用户执行编辑操作时) 和 `do_remote()` (当接收到来自远程用户的请求时)。在执行每次请求之后，就会调用回调函数来通知 NetPad 有关新文本变化的情况。`OpTrans` 对象的所有其他函数都用于内部。

4. 执行

`execute()` 函数包含着操作转换算法的大部分核心逻辑。它在请求队列中查找已经准备好执行的请求，根据执行日志中已经执行的操作，可能需要对这些请求执行转换，然后将操作应用到缓冲区中。应该将下面的 `execute()` 函数添加到 `OpTrans` 对象中。



可以从
Wrox.com
下载源代码

```

execute: function () {
    var r, cmp, new_queue = [];
    for (r = 0; r < this.queue.length; r++) {
        var remstate = this.queue[r][1];
        var cmp = this.compare_state(remstate, this.state);
        if (cmp < 1) {
            var op = this.queue[r][2];
            var orig_pos = op[1];
            if (cmp < 0) {
                var l = -1;
                while (l = this.find_prev(remstate, l) >= 0) {
                    var k = this.log[l][0];
                    if (remstate[k] <= this.log[l][1][k]) {
                        op = this.transform_op(
                            op,
                            this.log[l][2],
                            this.queue[r][3],
                            this.log[l][3]);
                    }
                }
            }
            var remote = this.queue[r][0] !== this.jid_map[this.jid];
            this.perform_op.apply(this, [remote].concat(op, orig_pos));
            this.log.push(this.queue[r].concat(orig_pos));
            this.state[this.queue[r][0]] += 1;
        } else {
            new_queue.push(this.queue[r]);
        }
    }
    this.queue = new_queue;
}

```

code snippet optrans.js

`execute()`函数按照请求添加进去的顺序依次遍历请求队列，查找已经准备好执行的请求。每个状态向量小于或等于本地用户的当前状态向量的请求都可以被执行。如果请求的状态向量严格小于本地用户的状态向量，那么必须将该请求转换之后才能够执行它。本地缓冲区已经被那些不是在请求的来源上执行的操作所修改。如果两个状态向量相同，那么请求可以立即被执行。

为了转换一个请求，搜索执行日志查找那些状态向量小于或等于当前请求的请求。`find_next()`函数返回执行日志中下一个这样的请求，如果没有找到这样的请求，就返回-1。此外，这些日志中的请求只有部分会影响到该转换，将日志记录的请求的状态向量和当前请求的状态向量进行比较(比较日志记录的请求的来源所对应的元素)，如果日志记录请求的状态向量中的元素大于或等于当前请求的状态向量的元素，那么必须转换该请求。在执行日志中找到的每一个这样的请求均会触发一个转换(由 `transform_op()`完成)，有可能出现多次转换。

将下面的 `find_prev()`函数添加到 `OpTrans` 对象中。



可以从 Wrox.com 下载源代码

```

find_prev: function (state, last_idx) {
    if (last_idx < 0) {
        last_idx = this.log.length;
    }

    var k;
    for (k = last_idx; k >= 0; k--) {
        if (this.compare_state(this.log[k][1], state) < 1) {
            break;
        }
    }

    return k;
}

```

code snippet optrans.js

一旦潜在的转换操作准备好执行，那么 `perform_op()`函数完成缓冲区操作，然后将该请求添加到执行日志中。注意，该操作的原始位置也被保存到执行日志中，这是因为在我前面给出的 `do_local()`函数中计算优先级时需要用到。

应该将下面的 `perform_op()`函数添加到 `OpTrans` 对象中。



```

perform_op: function (remote, op, pos, chr) {
    if (op === 'insert') {
        this.buffer.splice(pos, 0, [chr]);
    } else if (op === 'delete') {
        this.buffer.splice(pos, 1);
    }

    this.update_func(this.buffer.join(''), remote);
}

```

code snippet optrans.js

最终的操作非常容易执行。利用 JavaScript 的 Array 类的 `splice()` 方法，我们非常容易在任意位置插入或删除字符。一旦操作完成，就会用新的缓冲区来通知回调函数。注意在 `OpTrans` 对象内部，该缓冲区是作为一个数组保存的，但 `perform_op()` 将其作为一个字符串传给回调函数。

5. 转换

该算法唯一剩下的部分是执行操作转换。`transform_op()` 函数同时携带待转换的操作和转换对应的操作。第一个操作总是当前操作，而转换对应的操作将来自于执行日志。一旦完成转换，该函数就会返回新的、转换之后的操作。

应该将下面的 `transform_op()` 函数添加到 `OpTrans` 对象中。



可从
Wrox.com
下载源代码

```
transform_op: function (op1, op2, pri1, pri2) {
    var idx1 = op1[1];
    var idx2 = op2[1];

    if (op1[0] === 'insert' && op2[0] === 'insert') {
        if (idx1 < idx2) {
            return op1;
        } else if (idx1 > idx2) {
            return [op1[0], idx1 + 1, op1[2]];
        } else {
            if (op1[2] === op2[2]) {
                return null;
            } else {
                var cmp = this.compare_priority(pri1, pri2);
                if (cmp === -1) {
                    return [op1[0], idx1 + 1, op1[2]];
                } else {
                    return op1;
                }
            }
        }
    } else if (op1[0] === 'delete' && op2[0] === 'delete') {
        if (idx1 < idx2) {
            return op1;
        } else if (idx1 > idx2) {
            return [op1[0], idx1 - 1, op1[2]];
        } else {
            return null;
        }
    } else if (op1[0] === 'insert' && op2[0] === 'delete') {
        if (idx1 < idx2) {
            return op1;
        } else {
            return [op1[0], idx1 - 1, op1[2]];
        }
    } else if (op2[0] === 'delete' && op2[0] === 'insert') {
        if (idx1 < idx2) {
            return op1;
        }
    }
}
```

```

        } else {
            return [op1[0], idx1 + 1, op1[2]];
        }
    }
}

```

code snippet is part of optrans.js

共有 4 种可能的操作组合，分别是 insert() 对 insert() 或 delete() 转换以及 delete() 对 insert() 或 delete() 转换。在所有组合中，转换对应的操作都可能前后移动位置、被丢弃或保持不变。

考虑一个 delete() 操作被前面的 delete() 操作转换的情形。如果当前 delete() 操作所在的索引小于旧的 delete() 操作，那么不需要修改。如果索引相同，那么这个删除动作已经发生过，因此该操作是多余的。否则，该索引必须递减 1，以说明已删除一个字符。其他的操作转换与此类似。

稍微复杂一点的情况是两个操作都是 insert()。在这种情况下，如果两个 insert() 操作出现在同一索引处，那么该算法必须利用这些操作的优先级来确定最终的转换。如果没有进行优先级比较，就有可能让用户按照不同的顺序执行这些操作，最终导致不同的缓冲区。

程序清单 10-1 给出了最终的 optrans.js 文件。尽管 NetPad 使用该文件来实现两位用户之间的协作编辑，但操作转换算法(以及这里的特定实现)并不仅限于两个用户。可以非常容易地实现任意多个参与者同时处理同一个缓冲区的功能。



程序清单 10-1 optrans.js

可以从
Wrox.com
下载源代码

```

var OpTrans = {
    log: null, // the request log
    queue: null, // the request queue
    state: null, // our state vector
    jid: null, // our jid
    jid_map: null, // maps jids to positions in state vectors
    buffer: null, // the text buffer being affected by operations
    update_func: null, // callback function

    init: function (jids, buffer, update_func) {
        this.log = [];
        this.queue = [];
        this.state = [];
        this.jid_map = {};
        this.buffer = buffer.split('');
        this.jid = jids[0];
        this.update_func = update_func;

        $.each(jids.sort(), function (i) {
            OpTrans.jid_map[this] = i;
            OpTrans.state.push(0);
        });
    },
}

```

```
getState: function (jid) {
  ,

  do_local: function (op, pos, chr) {
    // calculate p
    var l, maxp = [];
    for (l = 0; l < this.log.length; l++) {
      if (this.log[l][4] === pos) {
        if (this.compare_priority(maxp, this.log[l][3]) == 1)
          maxp = this.log[l][3];
      }
    }
  }

  var p = maxp.concat(this.jid_map[this.jid]);

  // append request to queue
  var req = [this.jid_map[this.jid],
             this.state.concat(),
             [op, pos, chr],
             p];
  this.queue.push(req);

  this.execute();

  return req;
  ,

  do_remote: function (jid, state, op, pos, chr, pri) {
    // append request
    var req = [this.jid_map[jid], state, [op, pos, chr], pri];
    this.queue.push(req);

    this.execute();
  }

  compare_state: function (s1, s2) {
    var i, smaller = false;
    for (i = 0; i < s1.length; i++) {
      if (s1[i] > s2[i]) {
        return 1;
      } else if (s1[i] < s2[i]) {
        smaller = true;
      }
    }
    if (smaller) {
      return -1;
    }

    return 0;
  }

  compare_priority: function (p1, p2) {
    if (p1.length > p2.length) {

```

```

        return -1;
    } else if (p1.length < p2.length) {
        return 1;
    } else {
        var i;
        for (i = 0; i < p1.length; i++) {
            if (p1[i] > p2[i]) {
                return -1;
            } else if (p1[i] < p2[i]) {
                return 1;
            }
        }
    }
    return 0;
},
execute: function () {
    var r, cmp, new_queue = [];
    for (r = 0; r < this.queue.length; r++) {
        var remstate = this.queue[r][1];
        var cmp = this.compare_state(remstate, this.state);
        if (cmp < 1) {
            var op = this.queue[r][2];
            var orig_pos = op[1];
            if (cmp < 0) {
                var l = -1;
                while (l = this.find_prev(remstate, l) >= 0) {
                    var k = this.log[l][0];
                    if (remstate[k] <= this.log[l][1][k]) {
                        op = this.transform_op(op,
                                              this.log[l][2],
                                              this.queue[r][3],
                                              this.log[l][3]);
                    }
                }
            }
            var remote = this.queue[r][0] !== this.jid_map[this.jid];
            this.perform_op.apply(this, [remote].concat(op, orig_pos));
            this.log.push(this.queue[r].concat(orig_pos));
            this.state[this.queue[r][0]] += 1;
        } else {
            new_queue.push(this.queue[r]);
        }
    }
    this.queue = new_queue;
},
perform_op: function (remote, op, pos, chr) {
    if (op === 'insert') {

```

```
        this.buffer.splice(pos, 0, [chr]);
    } else if (op === 'delete') {
        this.buffer.splice(pos, 1);
    }
    this.update_func(this.buffer.join(''), remote);
},
find_prev: function (state, last_idx) {
    if (last_idx < 0) {
        last_idx = this.log.length;
    }

    var k;
    for (k = last_idx; k >= 0; k--) {
        if (this.compare_state(this.log[k][1], state) < 1) {
            break;
        }
    }
    return k;
},
transform_op: function (op1, op2, pri1, pri2) {
    var idx1 = op1[1];
    var idx2 = op2[1];

    if (op1[0] === 'insert' && op2[0] === 'insert') {
        if (idx1 < idx2) {
            return op1;
        } else if (idx1 > idx2) {
            return [op1[0], idx1 + 1, op1[2]];
        } else {
            if (op1[2] === op2[2]) {
                return null;
            } else {
                var cmp = this.compare_priority(pri1, pri2);
                if (cmp === -1) {
                    return [op1[0], idx1 + 1, op1[2]];
                } else {
                    return op1;
                }
            }
        }
    } else if (op1[0] === 'delete' && op2[0] === 'delete') {
        if (idx1 < idx2) {
            return op1;
        } else if (idx1 > idx2) {
            return [op1[0], idx1 - 1, op1[2]];
        } else {
            return null;
        }
    } else if (op1[0] === 'insert' && op2[0] === 'delete') {
```

```

        if (idx1 < idx2) {
            return op1;
        } else {
            return [op1[0], idx1 - 1, op1[2]];
        }
    } else if (op2[0] === 'delete' && op2[0] === 'insert') {
        if (idx1 < idx2) {
            return op1;
        } else {
            return [op1[0], idx1 + 1, op1[2]];
        }
    }
};

}

```

10.4 扩展 XMPP 协议

在前面的应用程序中，我们已经看过许多核心 XMPP 协议以及几个重要的扩展，但我们尚未建立自己的扩展。创建协议扩展是 XMPP 开发的重要组成部分，有时候我们甚至需要建立另一个扩展的扩展。

借助命名空间，XML 文档非常容易扩展。每个属性或元素都附属于某个命名空间，而且可以向不同的命名空间下面添加新元素和属性。XMPP 系统被设计忽略那些位于它们不能识别的命名空间下面的 XML，但他们会继续把这些有效载荷转发到它们的最终目的地。这种功能组合使得 XMPP 易于扩展，而 XMPP 扩展开发社区已经运用这些功能创建了将近 300 种协议扩展。

10.4.1 忽略未知数据

XMPP 扩展之所以能够运行是因为 XMPP 系统不要求理解通过的所有 XML 数据。可以在不破坏现有 XMPP 软件的情况下添加新功能，而任何未知数据不仅会被忽略，而且还将被保留直到传送给它的目的地。

XMPP 协议栈的每一部分都遵循着这条忽略未知数据的原则。例如，服务器并不了解有关输入通知的任何内容，但这些信息仍然可以由一个客户端发往另一个客户端。与此类似，客户端可能并不理解其他客户端的某些功能，这并不会导致它们不能运行，这些未能被理解的功能会被简单地忽略。

很难知道开发人员甚至最终用户如何使用协议。因此，最好尽可能地灵活，为未来的提升和改进预留空间。如果 XMPP 不能忽略未知的协议元素，那么很可能它仍然局限于它最初设计时的少数几种用途。

10.4.2 XML 命名空间

许多人熟悉 XML，但似乎很少有人接触过 XML 命名空间。命名空间对于定于 XMPP 扩展而言非常重要，因为它们可用来增加一些将在新的上下文中进行解释的元素和属性。

XML 命名空间只是一个 URI(uniform resource identifier, 统一资源标识符)。重要的是，尽管 URI 可能有时候看似 URL，但它不必引用 Web 上的实际位置。

在 XMPP 中，URI 有几种形式。最初使用的是 `jabber:foo` 或 `jabber:x:foo` 这样的形式。最终社区采用了更像 URL 的 URI，比如 `http://jabber.org/protocol/muc`。现在，XMPP URI 看似 `urn:xmpp:jingle:1`。这些区别是由于社区对 URI 的不断增长的认识以及 URI 本身的复杂性造成的。

官方 XMPP 扩展把它们的 URI 注册到 XMPP 注册管理机构，这属于标准化过程的一部分。那些并不打算进行标准化的、与应用程序相关的命名空间通常使用 URL 式的 URI，比如 `http://metajack.im/ns/netpad`。如果打算把自己的协议扩展提交到 XSF，那么可以联系 XMPP 注册管理机构获取一个临时的命名空间以供使用，直到自己的扩展达到协议草案的状态，届时将被分配一个永久性的命名空间。

XML 元素可以有默认的命名空间，而这个命名空间会被那些没有声明自己的命名空间的子元素继承。例如，XMPP 客户端流使用默认的命名空间 `jabber:client`，流中的`<message>`元素并不需要声明自己的命名空间 `jabber:client`，这是因为它将从默认命名空间继承这个属性。这种继承关系非常方便，因为不必为每个元素声明命名空间。大多数时候，元素均位于一个公共的、继承而来的命名空间。

10.4.3 扩展元素

扩展元素只是那些位于不同命名空间下面的新的子元素。我们已经看过这类元素的几个示例，包括 XHTML-IM 的`<html>`子元素以及本书中的每个 IQ-get 和 IQ-set 节。

XMPP 对于在哪里放置扩展元素并没有任何限制，但通常并不能将扩展元素放在 XMPP 节级别。XMPP 扩展甚至例行公事地将扩展元素添加到其他 XMPP 扩展的元素中。

特定协议扩展的扩展元素通常只能出现在几个特定的位置上。例如，如果不把 `jabber:iq:roster` 元素作为`<iq>`节的首个子元素，那么没有什么意义。有些扩展确实有一些更特别的性质，具体说来，Data Forms 就可以放在任何地方。

扩展元素以两种方式出现。可以在元素中通过携带 `xmlns` 属性来声明一个新的默认命名空间，或者可以使用命名空间前缀。

1. 改变默认命名空间

改变默认命名空间是扩展元素最常使用的方法，本书中的所有示例均使用这种方法。任何元素都可以声明默认命名空间，只需把 `xmlns` 属性的值设置为要用的命名空间即可。下面的示例给出了一个`<message>`节，它携带着一个位于新的默认命名空间下面的扩展`<event>`子元素。`<event>`元素下面的某些子元素也位于新的命名空间下面，这是因为它们没有特别地声明命名空间，而`<x>`元素为表单及其子元素声明了另一个默认命名空间。

```
<message to='elizabeth@longbourn.lit/bedroom'
         from='pubsub.pemberley.lit'>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='latest_books'>
      <item id='821b576dfabfc6b358b6ec4139b87f5c'>
        <x xmlns='jabber:x:data' type='result'>
```

```

<field var='title'>
  <value>A History of Pemberley</value>
</field>
<field var='author'>
  <value>Sir Lewis de Bourgh</value>
</field>
</x>
</item>
</items>
</event>
</message>

```

2. 命名空间前缀

除了修改默认命名空间，XML 元素还可以定义命名空间前缀。我们在第 1 章已经见过这种方法，开启 XMPP 流的<stream:stream>元素就是一个使用前缀命名空间的元素。

必须在使用前缀之前或在使用时定义它们，而且定义了前缀的元素的子孙元素也都将继承这个前缀的定义。通过包含 xmlns:foo 属性就定义了一个前缀，其中 foo 是希望使用的前缀。这个属性的值就是该前缀被绑定到的命名空间。前缀出现在元素名称之前，与名称之间以冒号隔开。

下面的示例给出了与前面一样的<message>节，但这次使用的是前缀而不是修改默认命名空间。

```

<message to='elizabeth@longbourn.lit/bedroom'
  from='pubsub.pemberley.lit'>
<pubsub:event xmlns:pubsub='http://jabber.org/protocol/pubsub#event'>
  <pypubsub:items node='latest_books'>
    <pubsub:item id='821b576dfabfc6b358b6ec4139b87f5c'>
      <form:x xmlns:form='jabber:x:data' type='result'>
        <form:field var='title'>
          <form:value>A History of Pemberley</form:value>
        </form:field>
        <form:field var='author'>
          <form:value>Sir Lewis de Bourgh</form:value>
        </form:field>
      </form:x>
    </pubsub:item>
  </pypubsub:items>
</pubsub:event>
</message>

```

在这里，使用前缀要比修改默认命名空间更加繁琐。但是，如果将<pubsub:event>元素的子元素替换成位于 jabber:client 命名空间下面的元素，那么使用前缀就能够有所改善。这就是为什么 XMPP 流的<stream:stream>元素使用前缀的原因，它所包含的元素大多数属于一个不同的命名空间。

注意，修改默认命名空间与使用前缀这两种方法可以同时用于同一个元素中。实际上，这

正是<stream:stream>元素所采用的方法。

```
<stream:stream xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    version='1.0'
    from='pemberley.lit'
    id='893ca401f5ff2ec29499984e9b7e8afc'
    xml:lang='en'>
```

10.4.4 扩展属性

虽然我们还可以创建扩展属性，但在协议扩展中很少用到这种方法。在 XML 中对待属性的方式与元素有些不同。不携带前缀的属性并不属于默认命名空间，而属于它所在的元素。元素如何解释这样的属性是由应用程序自己定义的。由于这个原因，建议只有在用于扩展 XMPP 时才使用前缀属性。

前缀属性的运行方式与前缀元素非常类似，而且必须按照与前缀元素相同的方式来定义该前缀。下面的示例在消息的<body>元素上使用一个前缀属性，以便向文本中添加一个明确的情绪(emotion)。

```
<message to='darcy@pemberley.lit'
    from='elizabaeth@longbourn.lit/ballroom'
    type='chat'>
    <body emote:emotion='annoyed' xmlns:emote='http://metajack.im/ns/emote-0'>
        I cannot talk of books in a ball-room; my head is always full of
        something else.
    </body>
</message>
```

我们还可以将 emotion 属性变成一个元素，或许这样就可以让这种情绪适于整个节而不仅仅是<body>元素。相反，元素是扩展 XMPP 的首选方式。很少有协议扩展使用属性，但如果需要这样做，那么还是能够做到。对于这两种方法的实际用法，请参见 XEP-0072 中的 SOAP over XMPP。

有一个前缀总是会被定义，那就是 xml，这个前缀常常用于向各种 XMPP 元素中添加属性。此外，任何以 xml(或 XML、xMl，等等)开头的前缀均是保留前缀。很多时候我们看到 xml:lang 属性用来指派使用哪一种语言来显示文本段。它还可用来提供可选的翻译。例如，可以使用两个<body>元素来发送一条消息，每个元素使用一个不同的 xml:lang 值。与其他前缀不同的是，xml 前缀并不需要定义，它在 XML 中总是隐含地定义，xmlns 属性就是这样。

10.4.5 贡献扩展

如果您开发出一个有趣的扩展，而且认为该扩展可能对社区普遍有用，那么应该考虑将其提交给 XSF 并参与标准化过程。XSF 欢迎来自 XMPP 社区内外的新的有趣的创意。

提交提议非常容易。<http://xmpp.org/extensions/submit.shtml> 提供了 XEP 文档模板，可以修改

该文档来描述自己的扩展。XEP-0001(XMPP Extension Protocols)描述了整个标准化过程，而XEP-0143中的Guidelines for Authors of XMPP Extension Proposals提供了一组有帮助的指南。

一旦提交，您的提议就会被发送到XMPP扩展编辑手中，而且该提议会被放入XEP收件箱中。XSF每年会选举一组成员来组成XMPP委员会，这个委员会将投票决定是否接受新提议或修改现有的扩展。

XMPP委员会和XMPP扩展编辑都非常友好，而且他们更乐于通过提供反馈以及回答您的所有问题来帮助您改进提议。

另一个参与方式是加入XSF列出的众多讨论列表中的一个。可以在<http://xmpp.org/about/discuss.shtml>找到这些列表。standards@xmpp.org列表就是讨论大多数协议的地方。列表成员提议新的创意，指出现有协议中的错误或遗漏的功能，并询问和解答与协议相关的问题。此外还有关于各种特殊主题的列表，甚至还有一个有关实时Web服务的列表ws-xmpp@xmpp.org。

还可以申请成为XSF会员。XSF每季度召开会员选举会议，XSF会员选举出XSF委员会、理事会和新的成员。他们还对组织结构和规则的变化进行投票。

在过去的几年中，XSF每年召开两次最高级别会议，一次在美国，一次在欧洲。这些最高级别会议是对所有人员开放的，许多最活跃的社区成员经常出席这些会议，同时还有来自各个公司的开发人员。可以在<http://xmpp.org/summit/>找到更多信息。

10.5 设计协议

NetPad协议是由其他XMPP协议加上一组应用程序特有的新元素组合而成的。这些新元素将属于一个特殊的命名空间<http://metajack.im/ns/netpad>。首先，通过使用标准的服务发现查询来支持NetPad协议的可发现功能。然后，用户通过在定向出席节中包含新元素的方式来建立会话。最后，使用普通的<message>节来收发启动和停止会话命令以及编辑操作本身等数据。

应用程序真正特有的部分比较少，而且可以利用现有XMPP协议的非常标准的部分来实现它们。这是一种典型的协议构造方法，这同时也说明了为什么如此多的开发人员能够轻易地构建出各种神奇功能。

10.5.1 测试支持

客户端可以通过适当地响应服务发现请求来指示是否支持NetPad协议。客户端必须等待disco#info查询并返回一个携带var属性值<http://metajack.im/ns/netpad>的<feature>元素。

下面是一个来自宣称支持NetPad协议的客户端的示例响应。

```

<iq to='darcy@pemberley.lit/library'
    from='bingley@netherfield.lit/drawing_room'
    type='result'
    id='netpad1'>
  <query xmlns='http://jabber.org/protocol/disco#info'>
    <identity category='client'
              type='pc'
              name='NetPad' />
  </query>
</iq>

```

```
    <feature var='http://jabber.org/protocol/disco#info'/>
    <feature var='http://metajack.im/ns/netpad'/>
  </query>
</iq>
```

一旦应用程序知道远程一方支持 NetPad 协议，它就可以开始使用该协议与远程进行交互。

10.5.2 请求和控制会话

一旦建立一个编辑会话，那么当另一方离开时，主持人用户将希望得到通知。定向出席非常适于为这种情况提供基础协议。远程用户可以随着(请求建立编辑会话的)NetPad 新协议元素一起发送定向出席消息。

这个新元素名为<collaborate>，下面给出了一个编辑会话请求的示例。

```
<presence to='bingley@netherfield.lit/drawing_room'  
         from='darcy@pemberley.lit/library'>  
  <collaborate xmlns='http://metajack.im/ns/netpad'/>  
</presence>
```

Bingley 不仅接收到协作请求, 他还将在 Darcy 离线(没有明确地终止任何会话)时接收到通知。

Bingley 有可能已经处在与另一位用户的会话之中，因此在这种情况下必须返回一种适当的错误提示消息。XMPP 节错误提示可以包含与应用程序相关的错误条件，因此我们可以针对这种情况创建`<already-collaborating>`条件。下面是 Bingley 返回的错误提示响应示例。

```
<presence to='darcy@pemberley.lit/library'
          from='bingley@netherfield.lit/drawing_room'
          type='error'>
  <collaborate xmlns='http://metajack.im/ns/netpad' />
  <error type='wait'>
    <recipient-unavailable xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
    <already-collaborating xmlns='http://metajack.im/ns/netpad' />
  </error>
</presence>
```

这条错误提示的通用条件是<recipient-unavailable>，这是这种情况下最适当的选择。该错误提示的类型为 wait，这说明若将来重试则可能会成功。失败请求的原始内容也包含在错误提示中。

或者，如果会话请求成功，那么 Bingley 应该向 Darcy 发送一条确认信息。此外，Bingley 可能已经开始编辑他的文档，因此必须将当前文本发送给 Darcy，这样 Bingley 未来的编辑命令才有意义。

可以使用<message>节来响应会话请求并将文档内容放入其中。这里需要一个新元素来指示这个<message>节属于 NetPad 协议，而且包含文档内容。由于这是编辑会话的正式开始，因此将这个元素称为<start>似乎比较合理。下面的示例给出了 Bingley 对 Darcy 的会话请求的响应。

```
<message to='darcy@pemberley.lit/library'  
        from='bingley@netherfield.lit/drawing room'
```

```

    type='chat'>
<start xmlns='http://metajack.im/ns/netpad'>
    Dearest Jane
</start>
</message>

```

Darcy 接收到<start>确认以及一段看上去像是一封写给 Jane 的信的开头部分的内容。如果尚未书写任何内容，那么<start>元素就是空白的。

Darcy 在任何时候都可以通过发送不可访问出席信息或离线来终止编辑会话，但 Bingley 也需要能够停止该会话。我们可以向<message>节中添加一个<stop>元素来实现这个功能，下面的示例给出了这个元素。

```

<message to='darcy@pemberley.lit/library'
    from='bingley@netherfield.lit/drawing_room'
    type='chat'>
    <stop xmlns='http://metajack.im/ns/netpad' />
</message>

```

现在双方都能够终止会话。

10.5.3 编辑操作

一旦会话建立完毕，参与方就需要向对方发送编辑操作。这些操作可以通过<message>节中的新元素来承载。这种新元素叫做<op>，它必须携带操作的名称及其参数。此外，操作请求必须包含状态向量和操作的优先级。

因为操作的名称、位置和字符属性均是单一值，所以它们被放入<op>元素的属性中。状态向量和优先级值是数字列表，因此这些值均被编码为子元素容器，每个子元素保存着单个单元的值。状态向量被编码放入<state>元素中，而优先级则放入<priority>元素中。每个单元值放在<cell>元素中。这与第 9 章的 SketchCast 应用程序使用的编码过程非常类似，但这里并没有使用表单，而是采用了一种自定义协议。

下面的节给出了实际运行的操作。

```

<message to='bingley@netherfield.lit/drawing_room'
    from='darcy@pemberley.lit/library'
    type='chat'>
    <op name='insert' pos='12' char='.' xmlns='http://metajack.im/ns/netpad'>
        <state>
            <cell>0</cell>
            <cell>0</cell>
        </state>
        <priority>
            <cell>1</cell>
        </priority>
    </op>
</message>

```

```

<message to='darcy@pemberley.lit/library'
         from='bingley@netherfield.lit/drawing_room'
         type='chat'>
  <op name='delete' pos='4' xmlns='http://metajack.im/ns/netpad'>
    <state>
      <cell>0</cell>
      <cell>1</cell>
    </state>
    <priority>
      <cell>0</cell>
    </priority>
  </op>
</message>

```

Darcy 添加了一个逗号，而 Bingley 开始删除 Dearest 中的 est。

这些就是 NetPad 协议所需的所有数据片段。通过将该协议构建在现有原语之上，新元素仍然简单但却能够完成大量任务。稍后我们将会看到这种协议不仅易于理解，而且还易于实现。

10.6 构建编辑器

在掌握了操作转换基础和 XMPP 协议扩展后，我们可以开始构建 NetPad 协作编辑器。我们分三个阶段来构建这个编辑器。首先，必须确定功能支持并设置好用户之间的编辑会话。接下来，添加两个用户之间的聊天功能。最后，让两个用户能够在编辑窗口中协作编辑同一份文档。

10.6.1 初始骨架

在开始编写代码之前，NetPad 需要一个用户界面、一些样式和初始的代码骨架。这些与我们在前几章中看到过的其他应用程序完全类似。

程序清单 10-2 给出了应用程序的基本布局。除了熟悉的标题和登录对话框之外，这个应用程序还包括状态区域、一个 Disconnect 按钮、一个编辑区域和一个聊天区域。



程序清单 10-2 netpad.html

可从
Wrox.com
下载源代码

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>NetPad - Chapter 10</title>
    <link rel='stylesheet' href='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/themes/cupertino/jquery-ui.css'>
    <script src='http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/jquery.js'>
    </script>
    <script src='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/jquery-ui.js'>
    </script>
    <script src='scripts/strophe.js'></script>

```

```

<script src='flxhr/flXHR.js'></script>
<script src='scripts/strophe.flxhr.js'></script>

<link rel='stylesheet' href='netpad.css'>
<script src='optrans.js'></script>
<script src='netpad.js'></script>
</head>
<body>
<h1>NetPad</h1>

<div id='status' class='no-collab'>
    Not collaborating.
</div>
<input id='disconnect' type='button' value='disconnect'
       disabled='disabled'>
<div class='clear'></div>

<textarea id='pad' disabled='disabled'></textarea>

<div id='chat-area'>
    <div id='chat'></div>
    <input id='input' type='text' disabled='disabled'>
</div>

<!-- login dialog -->
<div id='login_dialog' class='hidden'>
    <label>JID:</label><input type='text' id='jid'>
    <label>Password:</label><input type='password' id='password'>
    <label>Collaborator:</label><input type='text' id='collaborator'>
</div>
</body>
</html>

```

登录对话框包含一个名为 **Collaborator** 的域。如果这个域为空，那么 NetPad 将启动一个没有协作者的会话，它将允许另一个用户加入编辑会话。如果指定了一个协作者，那么它将试着加入给定用户的编辑会话。

这里包含了 **optrans.js** 文件，这是因为后面在使用我们创建的操作转换库来实现编辑功能时将用到这个文件。

程序清单 10-3 给出了 NetPad 应用程序的 CSS，并没有什么特别之处。



可从
Wrox.com
下载源代码

程序清单 10-3 netpad.css

```

body {
    font-family: Helvetica;
}

h1 {
    text-align: center;
}

```

```

.hidden {
    display: none;
}

.clear {
    clear: both;
}

#pad {
    width: 500px;
    height: 300px;
}

#disconnect {
    margin: 5px 5px 5px 50px;
}

#status {
    padding: 5px 15px 5px 15px;
    float: left;
}

.no-collab {
    background-color: #fcc;
}

.try-collab {
    background-color: #ccf;
}

.collab {
    background-color: #cfc;
}

```

程序清单 10-4 给出了最初的 JavaScript 骨架代码。到目前为止，读者应该已经相当熟悉初始连接代码。与前几章的应用程序的唯一不同之处在于额外添加的协作者字段。



可从
Wrox.com
下载源代码

程序清单 10-4 netpad.js(骨架)

```

var NetPad = {
    connection: null,
    collaborator: null
};

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Connect to XMPP',
        buttons: {

```

```

    "Connect": function () {
        $(document).trigger('connect', {
            jid: $('#jid').val(),
            password: $('#password').val(),
            collaborator: $('#collaborator').val()
        });

        $('#password').val('');
        $(this).dialog('close');
    }
});

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        "http://bosh.metajack.im:5280/xmpp-httpbind");

    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });

    NetPad.connection = conn;
    NetPad.collaborator = data.collaborator || null;
});

$(document).bind('connected', function () {
    // nothing here yet
});

$(document).bind('disconnected', function () {
    // nothing here yet
});

```

10.6.2 启动编辑会话

一旦用户单击 Connect 按钮，那么他要么等待另一位用户为自己启动一个编辑会话，要么试着为远程用户启动一个编辑会话。因为并不是每位用户都将运行 NetPad(或兼容 NetPad)编辑器，所以必须首先测试是否支持协作编辑。如果支持该功能，那么就会启动一个编辑会话。

我们在本章的前半部分开发的 NetPad 协议包含一些基本的会话建立支持。重要的是要知道协作者何时进入和离去，因此会话建立工作由定向出席完成，就像加入和离开多人聊天室一样。

1. 确定功能支持情况

功能支持是通过在答复 disco#info 查询时响应 NetPad 命名空间的方式来指出的。我们需要向 NetPad 应用程序中添加一个命名空间常量来记录这个命名空间。

除此之外，我们还需要知道应用程序是否作为主编辑器在运行。虽然主编辑器在协作者中

并没有什么特殊的权力，但主编辑器是会话的联系点。如果用户启动 NetPad 时没有指定协作者，那么应用程序将成为主编辑器并等待协作者。

将下面的属性添加到 NetPad 对象中。



可从
Wrox.com
下载源代码

```
NS_NETPAD: 'http://metajack.im/ns/netpad',
master: null
```

code snippet netpad.js

当指定协作者时，NetPad 必须向该协作者发送一个服务发现请求来确定他的客户端是否支持 NetPad 协议。当以主编辑器身份运行时，NetPad 必须侦听这些请求并响应正确的功能以确认它支持该功能。可以修改 `connected` 事件处理程序以添加该功能。



可从
Wrox.com
下载源代码

```
$(document).bind('connected', function () {
    if (NetPad.collaborator) {
        NetPad.master = false;

        $('#status')
            .text('Checking feature support for ' + NetPad.collaborator + '.')
            .attr('class', 'try-collab');

        // check for feature support
        NetPad.connection.sendIQ(
            $iq({to: NetPad.collaborator, type: 'get'})
                .c('query', {xmlns: Strophe.NS.DISCO_INFO}),
            function (iq) {
                var f = $(iq).find(
                    'feature[var=' + NetPad.NS_NETPAD + '"]');
                if (f.length > 0) {
                    $('#status')
                        .text('Establishing session with ' +
                            NetPad.collaborator + '.')
                        .attr('class', 'try-collab');

                    // request editing session
                } else {
                    $('#status')
                        .text('Collaboration not supported with ' +
                            NetPad.collaborator + '.')
                        .attr('class', 'no-collab');

                    NetPad.connection.disconnect();
                }
            });
    } else {
        NetPad.master = true;

        // handle incoming discovery requests
        NetPad.connection.addHandler(NetPad.on_disco_info,
```

```

        Strophe.NS.DISCO_INFO, "iq", "get");
    }
});
```

code snippet netpad.js

connected 事件处理程序检查是否已经指定协作者，如果没有指定，那么将 `master` 属性设为 `false`。它更新状态区域以通知用户发生的事件，然后它向协作者发送服务发现请求。

一旦协作者已经回复服务发现请求，那么 NetPad 检查响应结果中是否包含期望的功能。如果支持该功能，那么更新状态区域，并请求编辑会话。如果不支持该功能，那么状态区域中将显示一条适当的消息，并终止连接。

如果指定了协作者，那么 `master` 将被设置为 `true`，并添加一个处理程序来处理服务发现请求。我们需要实现这个处理程序 `on_disco_info()`。应该将下面的代码添加到 NetPad 对象中。



可从
Wrox.com
下载源代码

```

on_disco_info: function (iq) {
    NetPad.connection.sendIQ(
        $iq({to: $(iq).attr('from'),
              id: $(iq).attr('id'),
              type: "result"})
            .c('query', {xmlns: Strophe.NS.DISCO_INFO})
            .c('identity', {category: 'client',
                             type: 'pc'}).up()
            .c('feature', {'var': NetPad.NS_NETPAD})));
    return true;
}
```

code snippet netpad.js

这个处理程序发送携带适当的`<identity>`和`<feature>`元素的 IQ-result 节。NetPad 将自己识别为一个支持 NetPad 协议的聊天客户端。注意，该响应的 `id` 属性值必须与原始请求的 `id` 属性值匹配。

如果现在运行 NetPad，那么应该能够看到功能协商的实际运行情况。试着分别与另一个 NetPad 用户和一个普通的 XMPP 用户进行协作，观察不同的行为。

但这里尚不能断开连接，因此我们应该把 `Disconnect` 按钮连接起来。

首先，修改 `disconnected` 事件处理程序来重置应用程序。



可从
Wrox.com
下载源代码

```

$(document).bind('disconnected', function () {
    NetPad.connection = null;
    NetPad.collaborator = null;

    $('#login_dialog').dialog('open');
});
```

code snippet netpad.js

接下来，一旦建立连接就立即启用 `Disconnect` 按钮。将下面的代码行作为第一行添加到 `connected` 事件处理程序中。



可从
Wrox.com
下载源代码

```
$('#disconnect').removeAttr('disabled');
```

code snippet netpad.js

最后，添加这个按钮的 `click` 事件处理程序。应该将下面的代码添加到文档准备就绪事件处理程序中。



可从
Wrox.com
下载源代码

```
$('#disconnect').click(function () {
    $('#disconnect').attr('disabled', 'disabled');
    NetPad.connection.disconnect();
});
```

code snippet netpad.js

现在应用程序将按照用户的预期运行。

2. 建立 NetPad 会话

一旦 NetPad 确定协作者支持该协议，它就请求编辑会话。在我们设计的 NetPad 协议中，这是通过携带特殊的`<collaborate>`元素的定向出席节来实现的。

在发送定向出席节来请求编辑会话之后，主编辑器响应一个携带`<start>`元素的特殊消息来开始会话，或响应一个`<already-collaborating>`错误。主编辑器在终止会话时还可以一个使用`<stop>`元素的会话终止消息。如果另一个用户终止了他的连接，那么主编辑器需要停止该协作，它通过侦听不可访问出席节来处理这种情况。

首先，我们将修改 `connected` 事件处理程序，添加一个用来处理会话请求的处理程序。与此同时，主编辑器可以开始操作文本，因此需要启用编辑区域。下面的突出显示的部分是 `connected` 事件处理程序中的新增部分。



可从
Wrox.com
下载源代码

```
    } else {
        NetPad.master = true;
        $('#pad').removeAttr('disabled');

        // handle incoming discovery and collaboration requests
        NetPad.connection.addHandler(NetPad.on_disco_info,
            Strophe.NS.DISCO_INFO, "iq", "get");
        NetPad.connection.addHandler(NetPad.on_collaborate,
            NetPad.NS_NETPAD, "presence");
    }
```

code snippet netpad.js

现在将下面的 `on_collaborate()` 函数添加到 `NetPad` 对象中。



可从
Wrox.com
下载源代码

```
on_collaborate: function (presence) {
    var from = $(presence).attr('from');

    if (NetPad.collaborator) {
        // we already have a collaborator
        NetPad.connection.send(
            $pres({to: from, type: 'error'})
                .c('error', {type: 'wait'})
                .c('recipient-unavailable', {xmlns: Strophe.NS.STANZAS})
                .up()
                .c('already-collaborating', {xmlns: NetPad.NS_NETPAD}));
    } else {
        NetPad.collaborator = from;
        NetPad.start_collaboration(true);
    }

    return true;
}
```

code snippet netpad.js

`on_collaborate()` 处理程序检查是否已经有一个会话正在运行。如果找到一个现有的会话，那么它将返回一条适当的错误提示`<presence>`节，就像我们在协议设计示例中给出的 XMPP 节一样。否则，设置协作者并调用 `start_collaboration()` 开始会话。

下面给出了 `start_collaboration()` 函数，前面曾经讨论过，它向协作者发送会话开始消息，以通知他请求成功。



可从
Wrox.com
下载源代码

```
start_collaboration: function () {
    $('#status')
        .text('Collaborating with ' + NetPad.collaborator + '.')
        .attr('class', 'collab');

    if (NetPad.master) {
        // set up and send initial collaboration state
        NetPad.connection.send(
            $msg({to: NetPad.collaborator, type: 'chat'})
                .c('start', {xmlns: NetPad.NS_NETPAD}));
    } else {
        $('#pad').removeAttr('disabled');
    }
}
```

code snippet netpad.js

更新状态文本以反映新的编辑会话。主编辑发送一个携带 `NetPad` 命名空间下面的`<start>` 子

元素的<message>节，以确认会话已经开始。因为双方都将调用这个函数，所以有必要只针对主编辑器发送该消息。当另一个参与者调用该函数时，它只启用自己的编辑区域。

协作用户需要处理这条消息以便也能够启动该会话。修改 connected 事件处理程序，添加下面被突出显示的代码行。



可从
Wrox.com
下载源代码

```
$ (document).bind('connected', function () {
    $('#disconnect').removeAttr('disabled');

    NetPad.connection.addHandler(NetPad.on_message, null, "message");

    if (NetPad.collaborator) {
        NetPad.master = false;
```

code snippet netpad.js

还需要将下面的 on_message() 函数添加到 NetPad 对象中。



可从
Wrox.com
下载源代码

```
on_message: function (message) {
    var from = $(message).attr('from');

    if (NetPad.collaborator === from) {
        var collab = $(message)
            .find('*[xmlns="' + NetPad.NS_NETPAD + '"]');
        if (collab.length > 0) {
            if (NetPad.master) {
                // handle state changes
            } else {
                var command = collab[0].tagName;
                if (command === "start") {
                    NetPad.start_collaboration();
                } else if (command === "stop") {
                    NetPad.stop_collaboration();
                } else {
                    // handle state changes
                }
            }
        }
    }

    return true;
}
```

code snippet netpad.js

on_message() 处理程序将留意 NetPad 命名空间下面的元素。有可能接收到三种元素，分别是：<start>元素(指示会话开始)、<stop>元素(指示会话被主编辑器终止)和真正的编辑操作(稍后将实现)。

当主编辑器的客户端断开连接时，它首先通过发送会话终止消息来通知另一个参与者会话

被终止了。将下面被突出显示的修改部分应用到 Disconnect 按钮的 click 事件处理程序中。



可从
Wrox.com
下载源代码

```
$('#disconnect').click(function () {
    if (NetPad.collaborator) {
        NetPad.stop_collaboration(true);
    }

    $('#disconnect').attr('disabled', 'disabled');

    NetPad.connection.disconnect();
});
```

code snippet netpad.js

`stop_collaboration()`函数的实参控制着是否向协作者发送会话终止消息。当启动会话终止操作时，主编辑必须发送这条消息，但其他时候不需要发送通知。

将下面的 `stop_collaboration()` 函数添加到 NetPad 对象中。



可从
Wrox.com
下载源代码

```
stop_collaboration: function (notify) {
    $('#status')
        .text('Not collaborating.')
        .attr('class', 'no-collab');

    if (notify) {
        NetPad.connection.send(
            $msg({to: NetPad.collaborator, type: 'chat'})
                .c('stop', {xmlns: NetPad.NS_NETPAD}));
    }
}
```

code snippet netpad.js

这个函数只是更新状态区域，而且如果设置了 `notify` 参数，那么将向协作者发送会话终止消息。

主编辑器现在能够终止会话，但它还没有探测另一个参与者什么时候断开连接。因为另一个参与者将发送定向出席节，所以主编辑器可以留意不可访问<presence>节。通过对 `connected` 事件处理程序进行如下的修改，添加一个处理这些节的处理程序。



可从
Wrox.com
下载源代码

```
} else {
    NetPad.master = true;
    $('#pad').removeAttr('disabled');

    // handle incoming discovery and collaboration requests
    NetPad.connection.addHandler(NetPad.on_disco_info,
        Strophe.NS.DISCO_INFO, "iq", "get");
    NetPad.connection.addHandler(NetPad.on_collaborate,
        NetPad.NS_NETPAD, "presence");
```

```

    NetPad.connection.addHandler(NetPad.on_unavailable,
        null, "presence", "unavailable");
}

```

code snippet netpad.js

现在将下面的 `on_unavailable()` 处理程序添加到 `NetPad` 对象中。



可从
Wrox.com
下载源代码

```

on_unavailable: function (presence) {
    var from = $(presence).attr('from');

    if (from === NetPad.collaborator) {
        NetPad.stop_collaboration();
    }

    return true;
}

```

code snippet netpad.js

当其他参与者离开时，这个处理程序调用 `stop_collaboration()` 函数。因为另一个参与者已经断开连接，所以没有理由发送消息通知该会话已经结束。

如果此时测试 `NetPad` 应用程序，那么应该看到当一方加入和断开连接时协作式编辑会话会随之启动和结束。既然可以启动和停止会话，那么我们只需要发送和处理编辑命令，就可以实现一个实用的编辑器。但在此之前我们还应该添加另一个方便的功能。

10.6.3 谈论工作

每种协作流程的主要组成部分之一就是通信。两个合作处理同一份文档的人将有许多想法要讨论，因此我们需要向 `NetPad` 用户提供彼此进行聊天的功能。

我们已经构建了几种类型的聊天功能，而 `NetPad` 的聊天实际上要比第 6 章中的个人聊天客户端 `Gab` 更简单。稍后会看到通过利用 XMPP 的现有工具添加一项重要功能是多么简单的事情。

输入框在启动时是被禁用的，因此需要在会话启动后将其启用，并在会话结束时将其禁用。下面被突出显示的部分是必须对 `start_collaboration()` 和 `stop_collaboration()` 函数所做的修改。



可从
Wrox.com
下载源代码

```

start_collaboration: function () {
    $('#status')
        .text('Collaborating with ' + NetPad.collaborator + '.')
        .attr('class', 'collab');

    $('#input').removeAttr('disabled');

    if (NetPad.master) {
        // set up and send initial collaboration state
        NetPad.connection.send(
            $msg({to: NetPad.collaborator, type: 'chat'})
                .c('start', {xmlns: NetPad.NS_NETPAD}));
    }
}

```

```

    } else {
        $('#pad').removeAttr('disabled');
    }
},
stop_collaboration: function (notify) {
    $('#status')
        .text('Not collaborating.')
        .attr('class', 'no-collab');

    $('#input').attr('disabled', 'disabled');

    if (notify) {
        NetPad.connection.send(
            $msg({to: NetPad.collaborator, type: 'chat'})
                .c('stop', {xmlns: NetPad.NS_NETPAD}));
    }
}
}

```

code snippet netpad.js

接下来，连接输入框，当用户按下 Enter 键时发送消息。将下面的 keypress 事件处理程序添加到文档准备就绪事件处理程序中。



可从
Wrox.com
下载源代码

```

$('#input').keypress(function (ev) {
    if (ev.which === 13) {
        ev.preventDefault();

        var body = $(this).val();
        $('#chat').append("<div class='message'>" +
            "&lt;<span class='nick self'>" +
            Strophe.getBareJidFromJid(
                NetPad.connection.jid) +
            "</span>&gt; " +
            "<span class='message'>" +
            body +
            "</span>" +
            "</div>");

        NetPad.scroll_chat();

        NetPad.connection.send(
            $msg({to: NetPad.collaborator, type: 'chat'})
                .c('body').t(body));

        $(this).val('');
    }
});

```

code snippet netpad.js

这个函数将聊天文本格式化并将其添加到聊天区域，然后向协作者发送一个<message>节。一旦将该消息发送出去，该函数就清空输入框，并准备好处理下一条消息。

一旦添加了内容之后聊天区域必须滚动显示，以便让新消息正确显示。将下面的 scroll_chat() 函数添加到 NetPad 对象中，以实现这个效果。



可从
Wrox.com
下载源代码

```
scroll_chat: function () {
    var chat = $('#chat').get(0);
    chat.scrollTop = chat.scrollHeight;
}
```

code snippet netpad.js

最后，NetPad 必须留意传入的消息并将其显示出来。修改 on_message() 函数以匹配下面给出的代码。



可从
Wrox.com
下载源代码

```
on_message: function (message) {
    var from = $(message).attr('from');

    if (NetPad.collaborator === from) {
        var collab = $(message)
            .find('*[xmlns="' + NetPad.NS_NETPAD + '"]');
        if (collab.length > 0) {
            if (NetPad.master) {
                // handle state changes
            } else {
                var command = collab[0].tagName;
                if (command === "start") {
                    NetPad.start_collaboration();
                } else if (command === "stop") {
                    NetPad.stop_collaboration();
                } else {
                    // handle state changes
                }
            }
        } else {
            // add regular message to the chat
            var body = $(message).find('body').text();
            $('#chat').append("<div class='message'>" +
                "<span class='nick'>" +
                Strophe.getBareJidFromJid(from) +
                "</span>&gt; " +
                "<span class='message'>" +
                body +
                "</span>" +
                "</div>");

            NetPad.scroll_chat();
        }
    }
}
```

```

        return true;
    }

```

code snippet netpad.js

在进行这些少量修改之后，NetPad 现在支持协作者之间的聊天功能。

10.6.4 进行编辑

我们还剩下一项任务，即添加协作编辑这一主要功能。通过创建 OpTrans 对象已经完成大部分的工作，但还需要编写少量的“胶水”代码将一切有机地结合起来。

为了完成这些工作，我们需要初始化 OpTrans 对象，生成并广播本地操作，接收并处理远程操作，并用最新的修改来更新编辑区域。

1. 初始化 OpTrans

在会话启动时，所有参与者都需要调用 OpTrans 的 init()方法。可以修改 start_collaboration()函数来轻易地实现这一点。



```

start_collaboration: function () {
    $('#status')
        .text('Collaborating with ' + NetPad.collaborator + '.')
        .attr('class', 'collab');

    $('#input').removeAttr('disabled');

    var buffer = $('#pad').val();
    OpTrans.init([NetPad.connection.jid, NetPad.collaborator],
                buffer,
                NetPad.update_pad);

    if (NetPad.master) {
        // set up and send initial collaboration state
        var msg = $msg({to: NetPad.collaborator, type: 'chat'})
            .c('start', {xmlns: NetPad.NS_NETPAD});
        if (buffer) {
            msg.t(buffer);
        }

        NetPad.connection.send(msg);
    } else {
        $('#pad').removeAttr('disabled');
    }
}

```

code snippet netpad.js

将编辑区域的初始内容和相关的 JID 传入 init()方法。同时还修改会话启动消息，将会话开

始前主编辑器已经创建的文本包含在该消息中。

还必须修改 `on_message()` 处理程序(负责处理会话启动消息)来处理这些初始数据。在 `on_message()` 函数中进行如下被突出显示部分的修改。



```
if (command === "start") {
    $('#pad').val(collab.text());
    NetPad.start_collaboration();
} else if (command === "stop") {
    NetPad.stop_collaboration();
} else {
```

code snippet netpad.js

`start_collaboration()` 函数还将 `update_pad()` 函数传入 `OpTrans.init()`，每当编辑操作造成缓冲区变化时都会调用这个函数。将下面的 `update_pad()` 实现添加到 `NetPad` 对象中。



```
update_pad: function (buffer, remote) {
    var old_pos = $('#pad')[0].selectionStart;
    var old_buffer = $('#pad').val();
    $('#pad').val(buffer);

    if (buffer.length > old_buffer.length && !remote) {
        old_pos += 1;
    }

    $('#pad')[0].selectionStart = old_pos;
    $('#pad')[0].selectionEnd = old_pos;
}
```

code snippet netpad.js

`update_pad()` 函数的大部分代码处理的是光标位置的保存和恢复，这样它在更新时就能始终位于同一位置上。否则，在每次编辑之后，光标就会跳到缓冲区的末尾。但 IE6 不支持 `selectionStart` 和 `selectionEnd` 属性，因此该代码在该浏览器中并不能正常运行。但它在 IE 浏览器的后续版本以及其他浏览器中应该可以很好地运行。

2. 本地和远程操作

下一步是生成并广播本地编辑事件。当用户位于编辑区域时，特定的按键操作会导致缓冲区内容发生变化。这些键包括普通的数字字母键、Delete 键和 Backspace 键以及复制和粘贴的快捷键。但处理各种可能性需要大量的工作，因此出于简洁考虑，`NetPad` 仅限最基本的操作并不允许其他的操作。

下面的 `keypress` 事件处理程序将处理本地编辑事件。在确定发生的事件之后，它调用 `send_op()` 函数来更新 `OpTrans` 对象并将该操作广播到另一个用户。将下面的代码添加到文档准备就绪事件处理程序中。



可以从
Wrox.com
下载源代码

```

$( '#pad' ).keypress( function ( ev ) {
    if ( NetPad.collaborator ) {
        var idx = this.selectionStart;
        var handled = true;
        if ( ev.which === 8 ) {
            this.selectionStart = idx - 1;
            this.selectionEnd = idx - 1;
            NetPad.send_op( 'delete', idx - 1 );
        } else if ( ev.which === 46 ) {
            NetPad.send_op( 'delete', idx );
        } else if ( ( ev.which >= 32 && ev.which <= 127 ) ||
                    ev.which >= 256 ) {
            NetPad.send_op( 'insert', idx, String.fromCharCode( ev.which ) );
        }
        ev.preventDefault();
    }
});
```

code snippet netpad.js

按键编码 8 和 46 分别表示 Backspace 键和 Delete 键。最后一种情形所指定的键编码范围处理的是普通的可打印 ASCII 字符和可打印的非 ASCII 字符。这个处理程序将操作、该操作发生时所在的索引位置以及(对于插入操作)待插入的字符传入 sendop() 函数。任何没有在此得到明确处理的字符都将被忽略，这是因为这里调用了 preventDefault() 函数来关闭默认的行为。

send_op() 函数获取本地操作，按照前面设计的 NetPad 协议的 XML 格式进行编码，并将其发送给协作者。它对应的 process_op() 函数处理 NetPad 协议节的解码，并将操作请求传给 OpTrans 的 do_remote() 函数。将下面的两个函数添加到 NetPad 对象中。



可以从
Wrox.com
下载源代码

```

send_op: function ( op, pos, chr ) {
    var req = OpTrans.do_local( op, pos, chr );
    var op_attrs = { xmlns: NetPad.NS_NETPAD,
                    name: op,
                    pos: pos };
    if ( chr ) {
        op_attrs['char'] = chr;
    }

    var msg = $msg({ to: NetPad.collaborator, type: 'chat' })
        .c('op', op_attrs)
        .c('state');
    var i;
    for ( i = 0; i < req[1].length; i++ ) {
        msg.c('cell').t( '' + req[1][i] ).up();
    }

    msg.up().c('priority');
    for ( i = 0; i < req[3].length; i++ ) {
```

```

        msg.c('cell').t('' + req[3][i]).up();
    }

    NetPad.connection.send(msg);
}

process_op: function (op) {
    var name = op.attr('name');
    var pos = parseInt(op.attr('pos'), 10);
    var chr = op.attr('char');
    var pri = [];
    var state = [];

    op.find('state > cell').each(function () {
        state.push(parseInt($(this).text(), 10));
    });

    op.find('priority > cell').each(function () {
        priority.push(parseInt($(this).text(), 10));
    });

    OpTrans.do_remote(NetPad.collaborator,
                      state,
                      name, pos, chr,
                      pri);
}

```

code snippet netpad.js

最后一步是每当接收到远程操作时调用 `process_op()`。因为它们是在`<message>`节中接收到的，所以可以修改 `on_message()` 处理程序，添加如下被突出显示的代码行。



可从
Wrox.com
下载源代码

```

if (collab.length > 0) {
    if (NetPad.master) {
        NetPad.process_op(collab);
    } else {
        var command = collab[0].tagName;
        if (command === "start") {
            $('#pad').val(collab.text());
            NetPad.start_collaboration();
        } else if (command === "stop") {
            NetPad.stop_collaboration();
        } else {
            console.log("got remote op");
            NetPad.process_op(collab);
        }
    }
} else {

```

code snippet netpad.js

NetPad 现在可以发送和接收编辑操作，使用 OpTrans 对象来处理它们，并更新缓冲区内容以反映本地和远程修改。可与一位好友一起来测试这个应用程序，或者自己运行这个应用程序的多个副本。在编辑会话结束时，应该会注意到两位用户都具有相同的最终文本。

程序清单 10-5 给出了最终版本的 netpad.js 文件。



可从
Wrox.com
下载源代码

程序清单 10-5 netpad.js(最终版本)

```
var NetPad = {
    connection: null,
    collaborator: null,
    NS_NETPAD: 'http://metajack.im/ns/netpad',
    master: null,

    on_disco_info: function (iq) {
        NetPad.connection.sendIQ(
            $iq({to: $(iq).attr('from'),
                  id: $(iq).attr('id'),
                  type: "result"})
                .c('query', {xmlns: Strophe.NS.DISCO_INFO})
                .c('identity', {category: 'client',
                                 type: 'pc'}).up()
                .c('feature', {'var': NetPad.NS_NETPAD}));

        return true;
    },

    on_collaborate: function (presence) {
        var from = $(presence).attr('from');

        if (NetPad.collaborator) {
            // we already have a collaborator
            NetPad.connection.send(
                $pres({to: from, type: 'error'})
                    .c('error', {type: 'wait'})
                    .c('recipient-unavailable', {xmlns: Strophe.NS.STANZAS})
                    .up()
                    .c('already-collaborating', {xmlns: NetPad.NS_NETPAD}));
        } else {
            NetPad.collaborator = from;

            NetPad.start_collaboration(true);
        }

        return true;
    },

    start_collaboration: function () {
        $('#status')
            .text('Collaborating with ' + NetPad.collaborator + '...')
            .attr('class', 'collab');
    }
};
```

```

        $('#input').removeAttr('disabled');

        var buffer = $('#pad').val();
        OpTrans.init([NetPad.connection.jid, NetPad.collaborator],
                    buffer,
                    NetPad.update_pad);

        if (NetPad.master) {
            // set up and send initial collaboration state
            var msg = $msg({to: NetPad.collaborator, type: 'chat'})
                .c('start', {xmlns: NetPad.NS_NETPAD});
            if (buffer) {
                msg.t(buffer);
            }

            NetPad.connection.send(msg);
        } else {
            $('#pad').removeAttr('disabled');
        }
    },

    on_message: function (message) {
        var from = $(message).attr('from');

        if (NetPad.collaborator === from) {
            var collab = $(message)
                .find('*[xmlns="' + NetPad.NS_NETPAD + '"]');
            if (collab.length > 0) {
                if (NetPad.master) {
                    NetPad.process_op(collab);
                } else {
                    var command = collab[0].tagName;
                    if (command === "start") {
                        $('#pad').val(collab.text());
                        NetPad.start_collaboration();
                    } else if (command === "stop") {
                        NetPad.stop_collaboration();
                    } else {
                        NetPad.process_op(collab);
                    }
                }
            } else {
                // add regular message to the chat
                var body = $(message).find('body').text();
                $('#chat').append("<div class='message'>" +
                    "&lt;<span class='nick'>" +
                    Strophe.getBareJidFromJid(from) +
                    "</span>&gt; " +
                    "<span class='message'>" +
                    body +
                    "</span>" +
                    "</div>");

            }
        }
    }
}

```

```

        NetPad.scroll_chat();
    }
}

return true;
},

stop_collaboration: function (notify) {
    $('#status')
        .text('Not collaborating.')
        .attr('class', 'no-collab');

    $('#input').attr('disabled', 'disabled');

    if (notify) {
        NetPad.connection.send(
            $msg({to: NetPad.collaborator, type: 'chat'})
                .c('stop', {xmlns: NetPad.NS_NETPAD}));
    }
},

on_unavailable: function (presence) {
    var from = $(presence).attr('from');

    if (from === NetPad.collaborator) {
        NetPad.stop_collaboration();
    }

    return true;
},

scroll_chat: function () {
    var chat = $('#chat').get(0);
    chat.scrollTop = chat.scrollHeight;
},

update_pad: function (buffer, remote) {
    var old_pos = $('#pad')[0].selectionStart;
    var old_buffer = $('#pad').val();
    $('#pad').val(buffer);

    if (buffer.length > old_buffer.length && !remote) {
        old_pos += 1;
    }

    $('#pad')[0].selectionStart = old_pos;
    $('#pad')[0].selectionEnd = old_pos;
},

send_op: function (op, pos, chr) {
    var req = OpTrans.do_local(op, pos, chr);
    var opAttrs = {xmlns: NetPad.NS_NETPAD,
                  name: op,
                  pos: posK};
}
}

```

```
if (chr) {
    op_attrs['char'] = chr;
}

var msg = $msg({to: NetPad.collaborator, type: 'chat'})
    .c('op', op_attrs)
    .c('state');

var i;
for (i = 0; i < req[1].length; i++) {
    msg.c('cell').t('' + req[1][i]).up();
}

msg.up().c('priority');
for (i = 0; i < req[3].length; i++) {
    msg.c('cell').t('' + req[3][i]).up();
}

NetPad.connection.send(msg);
},

process_op: function (op) {
    var name = op.attr('name');
    var pos = parseInt(op.attr('pos'), 10);
    var chr = op.attr('char');
    var pri = [];
    var state = [];

    op.find('state > cell').each(function () {
        state.push(parseInt($(this).text(), 10));
    });

    op.find('priority > cell').each(function () {
        priority.push(parseInt($(this).text(), 10));
    });

    OpTrans.do_remote(NetPad.collaborator,
                      state,
                      name, pos, chr,
                      pri);
}
};

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Connect to XMPP',
        buttons: {
            "Connect": function () {
                $(document).trigger('connect', {
                    jid: $('#jid').val(),

```

```
        password: $('#password').val(),
        collaborator: $('#collaborator').val()
    });

    $('#password').val('');
    $(this).dialog('close');
}

});

$('#disconnect').click(function () {
    if (NetPad.collaborator) {
        NetPad.stop_collaboration(true);
    }

    $('#disconnect').attr('disabled', 'disabled');

    NetPad.connection.disconnect();
});

$('#input').keypress(function (ev) {
    if (ev.which === 13) {
        ev.preventDefault();

        var body = $(this).val();
        $('#chat').append("<div class='message'>" +
            "&lt;<span class='nick self'>" +
            Strophe.getBareJidFromJid(
            NetPad.connection.jid) +
            "</span>&gt; " +
            "<span class='message'>" +
            body +
            "</span>" +
            "</div>");

        NetPad.connection.send(
            $msg({to: NetPad.collaborator, type: 'chat'})
                .c('body').t(body));
    }

    $(this).val('');
});

$('#pad').keypress(function (ev) {
    if (NetPad.collaborator) {
        var idx = this.selectionStart;
        var handled = true;
        if (ev.which === 8) {
            this.selectionStart = idx - 1;
            this.selectionEnd = idx - 1;
            NetPad.send_op('delete', idx - 1);
        } else if (ev.which === 46) {
            NetPad.send_op('delete', idx);
        }
    }
});
```

```

        } else if ((ev.which >= 32 && ev.which <= 127) ||
                    ev.which >= 256) {
            NetPad.send_op('insert', idx, String.fromCharCode(ev.which));
        }

        ev.preventDefault();
    }
});

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        "http://bosh.metajack.im:5280/xmpp-httpbind");

    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });

    NetPad.connection = conn;
    NetPad.collaborator = data.collaborator || null;
});

$(document).bind('connected', function () {
    $('#disconnect').removeAttr('disabled');

    NetPad.connection.addHandler(NetPad.on_message, null, "message");

    if (NetPad.collaborator) {
        NetPad.master = false;

        $('#status')
            .text('Checking feature support for ' + NetPad.collaborator +
            .attr('class', 'try-collab');

        // check for feature support
        NetPad.connection.sendIQ(
            $iq({to: NetPad.collaborator, type: 'get'})
                .c('query', {xmlns: Strophe.NS.DISCO_INFO}),
            function (iq) {
                var f = $(iq).find(
                    'feature[var="' + NetPad.NS_NETPAD + '"]');

                if (f.length > 0) {
                    $('#status')
                        .text('Establishing session with ' +
                            NetPad.collaborator + '.')
                        .attr('class', 'try-collab');

                    NetPad.connection.send(
                        $pres({to: NetPad.collaborator})
                );
            }
        );
    }
});

```

```

        .c('collaborate', {xmlns: NetPad.NS_NETPAD})));
    } else {
        $('#status')
            .text('Collaboration not supported with ' +
                  NetPad.collaborator + '.')
            .attr('class', 'no-collab');
        NetPad.connection.disconnect();
    }
});
} else {
    NetPad.master = true;

    $('#pad').removeAttr('disabled');
    // handle incoming discovery and collaboration requests
    NetPad.connection.addHandler(NetPad.on_disco_info,
        Strophe.NS.DISCO_INFO, "iq", "get");
    NetPad.connection.addHandler(NetPad.on_collaborate,
        NetPad.NS_NETPAD, "presence");
    NetPad.connection.addHandler(NetPad.on_unavailable,
        null, "presence");
}
});
});

$(document).bind('disconnected', function () {
    NetPad.connection = null;

    $('#login_dialog').dialog('open');
});
}

```

10.7 扩展 NetPad

NetPad 的功能已经相当丰富，但它还缺少一些很好的功能。应该试着通过向 NetPad 添加下列功能中的一部分，让它成为一款更具竞争力的产品。

- 当主编辑器没有发送会话终止消息的情况下断开连接时，协作用户并没有接收到通知。修改协议以包含这种情况，并实现它。
- 允许两个以上的协作者。甚至可以使用多用户聊天室来广播编辑操作。
- 修正编辑区域，让它能够处理更高级的功能，比如方向键移动和复制粘贴功能。
- 向应用程序中添加一个共享白板(可以基于第 9 章的 SketchCast)，这样协作者就能够绘制图形。每位用户可以采用不同的颜色进行绘制。

10.8 小结

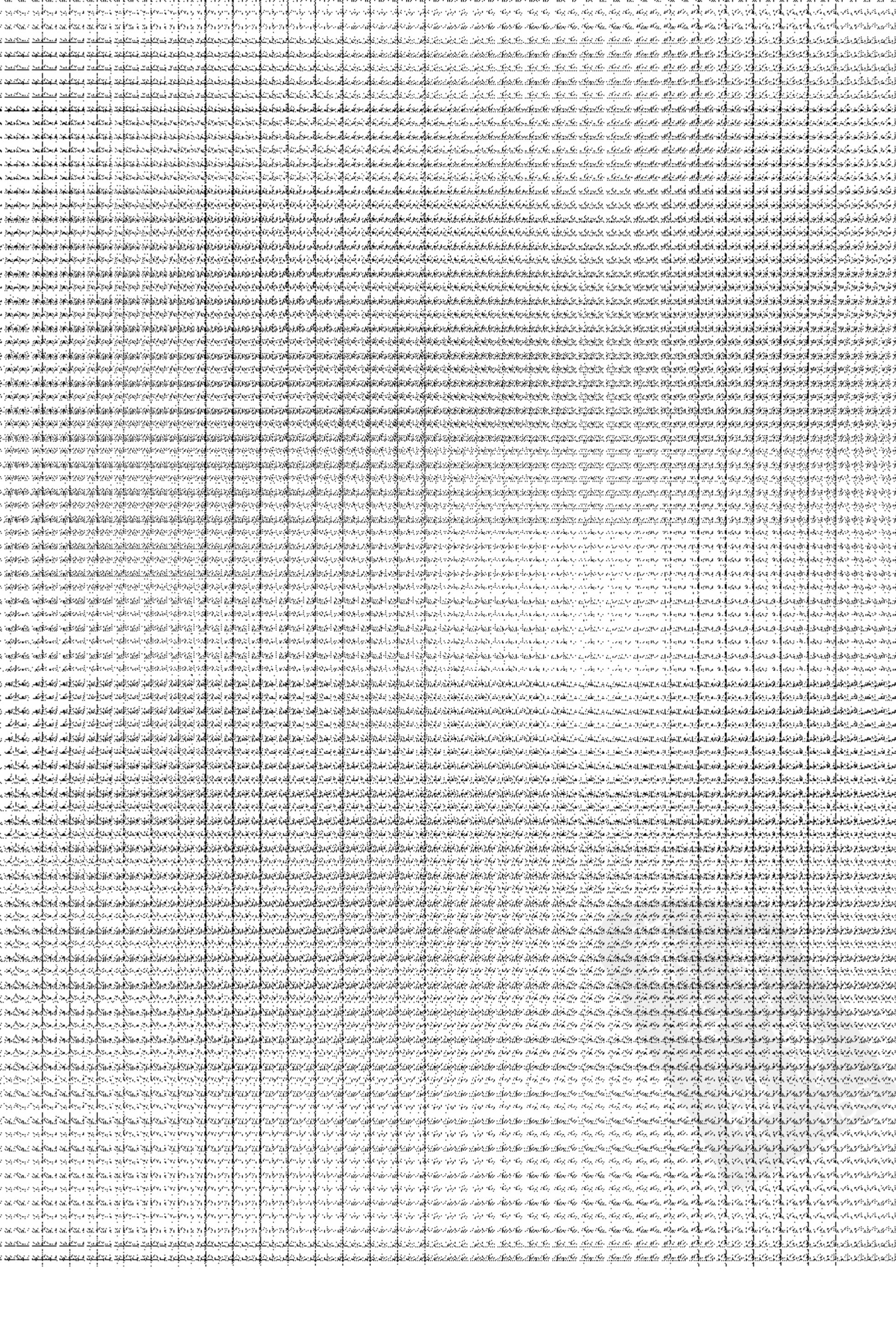
在本章中我们构建了一个复杂的应用程序，可以让两位用户以协作方式实时地编辑同一份文档。像 NetPad 这样的应用程序变得越来越常见，而 XMPP 协议使得构建它们并非难事。

希望您能够从本章中学到许多知识，包括：

- 操作转换背后的原理
- 各种扩展 XMPP 节以完成自定义协议的方式
- 使用定向出席来实现自己的会话管理
- 轻松地向现有应用程序中添加聊天和其他社交功能

在第 11 章中，我们将构建本书中最大的应用程序，一个为 Tic-Tac-Toe 玩家准备的多用户、实时游戏系统。





第 11 章

玩游戏：面对面的 Tic-Tac-Toe

本章内容

- 编写 XMPP 机器人
- 创建复杂的协议
- 基于 MUC 的协作空间

几乎所有人都喜欢玩游戏，而像《魔兽世界》这样的大型多人在线角色扮演游戏(MMORPG)的成功说明，人们喜欢与别人一起玩游戏。许多游戏依赖于玩家之间的低延迟交互，还有些则需要具备让玩家找到彼此并进行通信的功能。XMPP 提供了大量的非常适于游戏的内置功能，而更复杂的功能则只需要扩展 XMPP 节即可。

XMPP 极擅长于来回发送小片的结构化信息，而游戏可以利用它来传播游戏世界或状态的变化。与其他形式的协作相比，这实际上并不是一个多么不同的问题，唯一不同的就是数据的用途。只需要将第 10 章的 NetPad 应用程序的完全由文本构成的缓冲区替换成到处是怪兽的地下城即可。对于 XMPP 而言，这是一些 XML 数据而已。

只要玩家不是在独自一人玩游戏，那么通信就是玩游戏过程中的重要一环。在阅读本书的过程中我们已经发现，XMPP 拥有用来实现所有类型通信的多种工具，而且这些工具中的大多数是内置的，不需要做太多工作就可以使用它们。是否还记得向 NetPad 应用程序中添加聊天功能是多么简单的事情吗？在游戏方面我们也同样可以做到。在玩家的出席信息中添加一些与游戏有关的信息，以前的聊天花名册现在变成了同道冒险者公会。

我们将在本章中将 XMPP 运用到这种快乐的工作中，我们将开发一个 Tic-Tac-Toe 游戏，世界各地的用户都可以玩这个游戏。

11.1 应用程序预览

Toetem 的 UI 由大量的不同部分组成。图 11-1 给出了初始屏幕，其中显示了当前游戏列表和等待玩家列表。

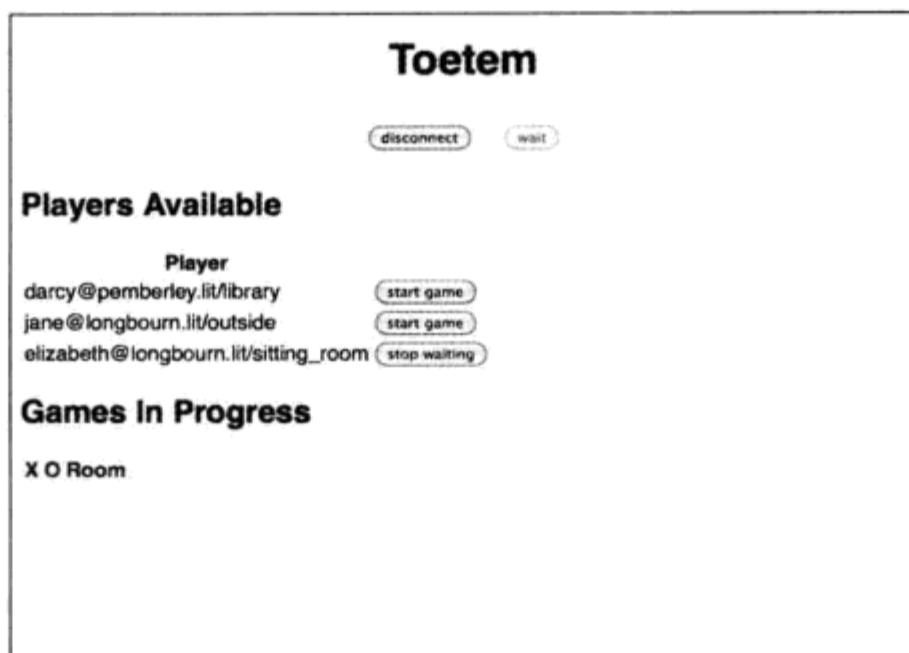


图 11-1

在界面的顶部排列着几个按钮。我们在前面已经看过 Disconnect 按钮，但另一个按钮是全新的。Wait 按钮将用户添加到等待玩游戏的玩家列表中。

在按钮条的下方是等待玩家列表。这些是那些已经准备好玩 Tic-Tac-Toe 游戏的玩家。在每位玩家旁边有一个 Play 按钮，用户可以单击这个按钮来启动一个游戏供他与该玩家一起玩，如果该玩家就是用户本人，那么会有一个 Stop Waiting 按钮。

在界面底部是活跃游戏列表。每当有一个新游戏启动时，它都会出现在这个列表中，而当该游戏结束时，它就会从该列表中消失。每个游戏的旁边有一个 Watch 按钮，可以让用户作为一名观众加入该游戏以观看赛事。

图 11-2 给出了游戏板和聊天区域。当轮到玩家时，他通过单击游戏板与之交互。顶部的按钮可用来放弃游戏。聊天区域可让所有玩家和观众彼此交谈。

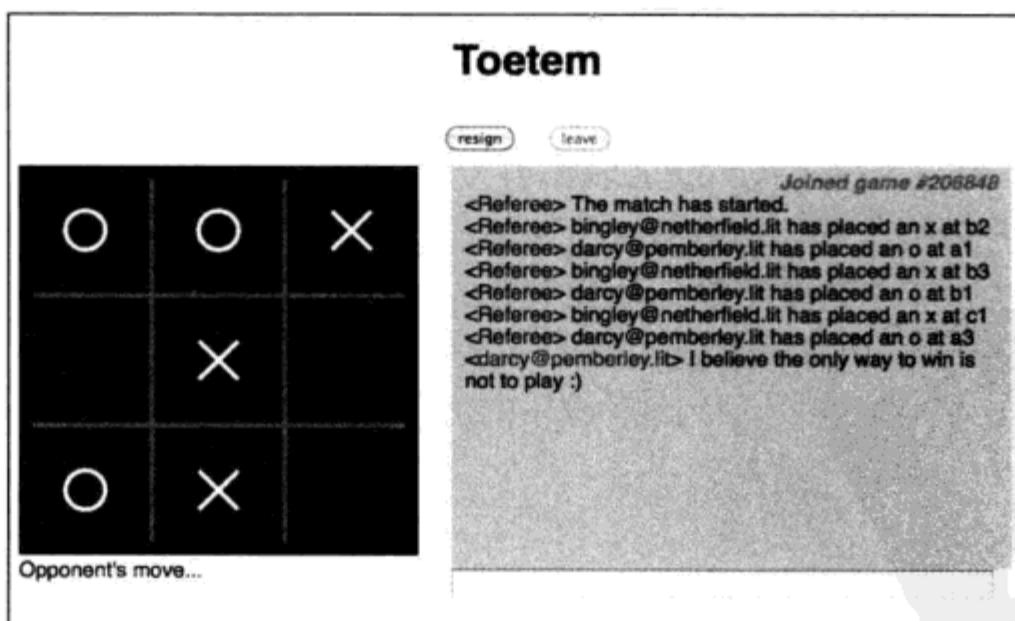


图 11-2

这是我们在本书中构建的最大的应用程序，但我们很快就会看到，它是由我们已经完成的各部分组合起来的。

11.2 Toetem 的设计

Toetem 由两个主要部分组成：裁判员和游戏客户端。

裁判员是一个负责执行游戏规则并记录活跃游戏和等待玩家的 XMPP 机器人。用户只通过自己在游戏客户端中的动作间接地与裁判员交互。系统中的所有玩家只需要一个裁判员，但如果缺少裁判员，那就没有办法阻止恶意玩家的作弊和滥用行为。

游戏客户端是每位玩家用来访问系统和玩游戏的应用程序。用户的动作将发送给裁判员，然后广播到其他玩家和观众。

因为裁判员是一个机器人，所以它不需要用户界面。它只负责维护一个连接并侦听来自用户的命令，检查它们的合法性，然后将它们广播给其他用户或更新它的内部状态。

裁判员理解多种命令，我们将在第 11.3 节中进行详细讲解。我们创建的协议是所有 3 种 XMPP 节（`<iq>`、`<presence>` 和 `<message>`）的有机组合，并利用了现有扩展，比如 MUC。

游戏被组织成多人聊天室的形式。每个游戏均有一个相关联的房间，所有玩家和观众均加入这个房间来玩这个游戏或观看该游戏。这个聊天室为该游戏的所有参与者提供了通信区域，还用来将玩家的移动广播给所有用户。多人聊天室已经提供了绝大部分所需的功能，因此它们为 Toetem 协议提供了极佳的构建块。

机器人和组件

Toetem 使用机器人作为裁判员，但机器人有一些潜在的缺陷。因为机器人是服务器上的普通用户，所以它们会有完整的花名册和所有路由到客户端的普通消息。通常，这种额外的功能并不是必需的，而且可能会妨碍机器人的可伸缩性。

在更为健壮的系统中，裁判员将被替换成一个执行同样功能的组件。该组件将对自己的资源和路由进行更多的控制，而且它不会很快地达到尺寸上限。有关组件和伸缩性方面的更多信息，请参见第 1.3.3 节和第 13.1.3 节。

希望玩游戏的用户在连接时需要知道裁判员的 JID。一旦连接，他们就能够查看当前游戏列表以及等待玩家列表。他们可以选择与正在等待的玩家之一来启动一个游戏，或者等待其他玩家来玩游戏，或者观看其他用户正在玩的游戏之一。

如果启动一个游戏或作为观众加入一个游戏，那么他们将切换到游戏视图。在游戏视图中，用户将看到 Tic-Tac-Toe 游戏板以及游戏的聊天区域。每当玩家移动时，所有玩家和观众都能够实时地看到游戏板更新。观众任何时候都能够离开游戏，但玩家必须等到游戏结束后才能离开。

大部分工作是定义并实现 Toetem 协议并创建游戏板控件。我们在本章中设计的协议并不仅限于 Tic-Tac-Toe 游戏，实际上，它是 Chesspark 象棋游戏使用的协议的简化版。只需要修改游戏板和特定移动类型和状态，我们就可以将其转换成任何其他类型的游戏。之所以选择 Tic-Tac-Toe 游戏是因为它的规则极其简单，这样我们就可以将注意力放在协议以及用户的交互方面，而不是复杂的游戏逻辑。

11.3 设计游戏协议

在第 10 章中，我们设计了一个简单的协议，而在本章中我们设计了一个更复杂的用来管理和运行游戏系统的协议。这个协议的所有部分均构建在我们在前几章中已经讨论过的协议块之上，但把这些简单部分组合起来就形成了一个复杂的系统。

Toetem 协议由 4 个主要部分组成。首先，需要某种方式来跟踪系统中的玩家，而且就像在第 10 章中介绍一样，我们使用定向出席来完成这项任务。接下来，需要管理正在等待玩游戏的玩家列表。还需要管理活跃游戏以及它们对应的聊天室。最后，必须处理每个游戏内部的玩家交互。

11.3.1 跟踪用户

跟踪用户是一个相当重要的功能。对于启动者来说，其他玩家将希望知道谁可以玩游戏以及谁正在玩游戏。而更重要的是，裁判员必须适当地处理玩家意外离开的情况。

不在玩游戏或不在等待列表中的用户离开，那么这并不是我们要关心的情形。但如果正在玩游戏的玩家或在等待列表中的玩家离开，那么其他用户就需要了解这带来的结果。如果用户单击某个正在等待的玩家的 Play 按钮，然后发现该玩家已经不在那里，那么这会让人觉得生气。更糟糕的是玩家在游戏期间退出。

第一种情况很容易处理，只需要将该玩家从等待玩家列表中删除即可。在第二种情况下，必须决定如何处理该情况。在 **Toetem** 中，我们采用了最简单的解决方案，就是惩罚离开游戏的玩家。虽然这种解决方案并不是最优的选择，但它防止了最常见的滥用情形，也就是即将失败的玩家通过离开游戏实现作弊。

Toetem 协议使用定向出席来跟踪玩家。为了与裁判员交互，用户必须首先向它发送定向出席节。此后，当该用户不再可访问时裁判员将获知该信息，然后就可以采取适当的动作。这是第 10 章中的 **NetPad** 应用程序和多人聊天室采用的相同技术。

下面的<presence>示例节显示 **Darcy** 正在把自己注册到裁判员。注意协议使用的命名空间是 <http://metajack.im/ns/toetem>。

```
<presence to='referee@pemberley.lit/toetem'
          from='darcy@pemberley.lit/library'
          <register xmlns='http://metajack.im/ns/toetem' />
</presence>
```

当玩家离开时，裁判员将接收到与下面类似的 XMPP 节。

```
<presence to='referee@pemberley.lit/toetem'
          from='darcy@pemberley.lit/library'
          type='unavailable' />
```

一旦玩家向裁判员注册，裁判员就通过一对<message>节将当前的等待玩家列表和活跃游戏列表发送给他。在注册之后接收到下面的<message>节。

```

<message to='darcy@pemberley.lit/library'
         from='referee@pemberley.lit/toetem'>
  <waiting xmlns='http://metajack.im/ns/toetem'>
    <player jid='elizabeth@longbourn.lit/sitting_room' />
  </waiting>
</message>

<message to='darcy@pemberley.lit/library'
         from='referee@pemberley.lit/toetem'>
  <games xmlns='http://metajack.im/ns/toetem'>
    <game x-player='jane@longbourn.lit/outside'
          o-player='bingley@netherfield.lit/drawing_room'
          room='toetem-1045@games.pemberley.lit' />
  </games>
</message>

```

<waiting>元素中为每位正在等待的玩家携带了一个包含其 JID 的<player>元素。与此类似, <games>元素为每个活跃游戏携带了一个包含其玩家和房间的<game>元素。

11.3.2 管理玩家

玩家需要发出信号说明他们正在等待开始游戏, 而在他们停止等待时也同样需要发出信号。借助<iq>节很容易完成这些任务。玩家还需要处理当前等待玩家列表的变化(裁判员通过<message>节发送)。

如果用户希望自己添加到等待列表中, 那么他们通过 IQ-set 节来发送<waiting>命令。为了将自己从该列表中移除, 他们通过 IQ-set 节发送<stop-waiting>命令。Elizabeth 通过如下的 XMPP 节将自己添加到该列表中, 然后又从中删除。

```

<iq to='referee@pemberley.lit/toetem'
     from='elizabeth@longbourn.lit/sitting_room'
     type='set'
     id='waiting2'>
  <waiting xmlns='http://metajack.im/ns/toetem' />
</iq>

<iq to='elizabeth@longbourn.lit/sitting_room'
     from='referee@pemberley.lit/toetem'
     type='result'
     id='waiting2' />

<iq to='referee@pemberley.lit/toetem'
     from='elizabeth@longbourn.lit/sitting_room'
     type='set'
     id='waiting3'>
  <stop-waiting xmlns='http://metajack.im/ns/toetem' />
</iq>

<iq to='elizabeth@longbourn.lit/sitting_room'
     from='referee@pemberley.lit/toetem'
     type='result'
     id='waiting3' />

```

```
type='result'
id='waiting3'/>
```

这些 XMPP 节均是只有空响应的相当简单的命令，但它足以实现 Toetem 的需求。

当玩家已经处于等待状态时也可能发送<waiting>命令，或者玩家尚未开始等待时发送<stop-waiting>命令。Toetem 协议也需要处理这些情况。在第一种情况下，没有理由抛出错误，裁判员只需要假装它已经将该用户添加到列表中并返回成功响应。而在另一种情况下，裁判员必须返回一条合适的错误消息。在下面的示例中，裁判员发送一个 bad-request 条件来响应 Jane 发出的奇怪的<stop-waiting>命令。

```
<iq to='jane@longbourn.lit/outside'
    from='referee@pemberley.lit/toetem'
    type='error'
    id='stop1'>
<stop-waiting xmlns='http://metajack.im/ns/toetem'/>
<error type='cancel'>
    <bad-request xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
</error>
</iq>
```

每当等待列表改变时，裁判员都将该更新信息发送给所有已注册玩家。每个玩家都将收到类似下面的<message>节。

```
<message to='darcy@pemberley.lit/library'
    from='referee@pemberley.lit/toetem'>
<waiting xmlns='http://metajack.im/ns/toetem'>
    <player jid='jane@longbourn.lit/outside' />
</waiting>
</message>

<message to='darcy@pemberley.lit/library'
    from='referee@pemberley.lit/toetem'>
<not-waiting xmlns='http://metajack.im/ns/toetem'>
    <player jid='jane@longbourn.lit/outside' />
</not-waiting>
</message>
```

第一个 XMPP 节与玩家首次注册时接收到的节非常相似，不同之处仅在于更新信息只包含一个<player>元素。有关玩家离开等待列表的更新信息也基本相同，不同之处在于<not-waiting>元素而不是<waiting>。

用户看到的总是精确的等待玩家列表，这是因为他们最初得到的是当前列表，而且每当该列表改变时他们都会接收到更新信息。没有必要让所有用户轮询等待列表，更新信息将实时地被传送并得到处理。

这些简单的交互(除了前面描述的不可访问出席通知)就是裁判员管理等待玩家列表所需的全部工作。

11.3.3 管理游戏

游戏的管理稍微复杂一些。从用户的角度来看，需要与等待玩家列表相同的三种动作，即处理活跃游戏列表的更新信息、启动游戏和放弃游戏。但游戏开始和结束时裁判员必须创建和销毁多人聊天室，而这涉及的工作量更多一些。

一旦游戏开始，该游戏的玩家将接收到裁判员发送的一条消息，该消息中包含该游戏房间的名称。之后这些玩家必须加入该房间，游戏才能开始。在游戏结束之后，玩家可以从容地离开，他们可能希望在赛后继续与对方玩家聊天。

活跃游戏列表更新的处理方式与等待列表更新的处理方式完全类似。当游戏开始和结束时，将向所有已注册玩家发送一条消息。当 Jane 和 Elizabeth 启动和结束他们的游戏时，Darcy 将看到如下的 XMPP 节。

```

<message to='darcy@pemberley.lit/library'
         from='referee@pemberley.lit/toetem'>
  <games xmlns='http://metajack.im/ns/toetem'>
    <game x-player='jane@longbourn.lit/outside'
          o-player='elizabeth@longbourn.lit/sitting_room'
          room='toetem-123@games.pemberley.lit'/>
  </games>
</message>

<message to='darcy@pemberley.lit/library'
         from='referee@pemberley.lit/toetem'>
  <game-over xmlns='http://metajack.im/ns/toetem'>
    <game x-player='jane@longbourn.lit/outside'
          o-player='elizabeth@longbourn.lit/sitting_room'
          room='toetem-123@games.pemberley.lit'/>
  </game-over>
</message>

```

与等待列表相同，游戏通知是立即传送的，这样每位玩家都总能看到最新的信息，而不需要不停地询问。

如果 Elizabeth 希望观看游戏，那么她只需要加入该游戏的房间即可。然后裁判员就开始将游戏移动信息广播给她，我们将在下一节中进行描述。

为了启动自己的游戏，Elizabeth 必须向裁判员发送<start>命令，并将她希望一起玩游戏的玩家名称随该命令一同发送过去。下面的示例给出了一次成功的请求及其响应。

```

<iq to='referee@pemberley.lit/toetem'
     from='elizabeth@longbourn.lit/sitting_room'
     type='set'
     id='start1'>
  <start xmlns='http://metajack.im/ns/toetem'
         with='jane@longbourn.lit/outside'/>
</iq>

<iq to='elizabeth@longbourn.lit/sitting_room'

```

```
from='referee@pemberley.lit/toetem'
type='result'
id='start1'/>
```

然后裁判员创建游戏房间并邀请玩家。这些邀请函是由该房间代表裁判员发送的，在下面的示例中可以看到。XEP-0045 的第 7.5.2 节(Mediated Invitation)定义了这些邀请函的格式。

```
<message to='elizabeth@longbourn.lit/sitting_room'
         from='toetem-456@games.pemberley.lit'>
  <x xmlns='http://jabber.org/protocol/muc#user'>
    <invite from='referee@pemberley.lit/toetem' />
  </x>
</message>

<message to='jane@longbourn.lit/outside'
         from='toetem-456@games.pemberley.lit'>
  <x xmlns='http://jabber.org/protocol/muc#user'>
    <invite from='referee@pemberley.lit/toetem' />
  </x>
</message>
```

一旦接收到邀请函，两位玩家必须加入该房间。此后的协议交互将在第 11.3.4 节中讲解。

游戏启动时可能存在几种可能的错误情况。如果玩家试图与一位目前并不在等待列表中的用户开始游戏，那么服务器将返回一条类似下面的 **item-not-found**(项不存在)错误节。

```
<iq to='jane@longbourn.lit/outside'
     from='referee@pemberley.lit/toetem'
     type='error'
     id='start2'>
  <start xmlns='http://metajack.im/ns/toetem'
         with='elizabeth@longbourn.lit/library' />
  <error type='modify'>
    <item-not-found xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
  </error>
</iq>
```

此外，被请求的玩家有可能已经开始与别人玩游戏并且目前正在忙于玩游戏。在这种情况下，裁判员返回一条类似下面的 **not-allowed**(操作不允许)错误消息。

```
<iq to='jane@longbourn.lit/outside'
     from='referee@pemberley.lit/toetem'
     type='error'
     id='start3'>
  <start xmlns='http://metajack.im/ns/toetem'
         with='darcy@pemberley.lit/library' />
  <error type='cancel'>
    <not-allowed xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
  </error>
</iq>
```

当游戏进行过程中，玩家可能希望放弃而不是等待显而易见的最终结果。放弃游戏是通过向裁判员发送一个`<resign>`命令(通过 IQ-set 节)完成的。在下面的示例中，由于被聪明的妹妹打败，Jane 放弃了游戏。

```

<iq to='referee@pemberley.lit/toetem'
    from='jane@longbourn.lit/outside'
    type='set'
    id='resign1'>
  <resign xmlns='http://metajack.im/ns/toetem' />
</iq>

<iq to='jane@longbourn.lit/outside'
    from='referee@pemberley.lit/toetem'
    type='result'
    id='resign1' />

```

如果目前没有参与游戏的玩家试图放弃，那么他将接收到与前一个示例中相同的bad request 错误提示。

Toetem 协议仅剩下一部分尚未完成，即玩家在游戏房间中彼此之间的交互。

11.3.4 玩游戏和观看游戏

当接收到游戏启动请求时，裁判员为该游戏创建一个房间，加入该房间，然后邀请这两个玩家。一旦这两个玩家都加入该房间，游戏就开始了。裁判员负责管理游戏状态、接收玩家的移动，然后根据需要将该游戏的移动和状态广播出去。

裁判员的一项任务就是在游戏启动和结束时通知房间中每一个人。这很容易完成：向该房间发送一条消息，其中携带一些 Toetem 协议元素以及普通的消息内容。下面的示例节给出了`<game-started>`和`<game-ended>`元素。

```

<message to='elizabeth@longbourn.lit/sitting_room'
    from='toetem-789@games.pemberley.lit/referee'
    type='groupchat'>
  <body>The match has started.</body>
  <game-started xmlns='http://metajack.im/ns/toetem'
    x-player='elizabeth@longbourn.lit/sitting_room'
    o-player='jane@longbourn.lit/outside' />
</message>

<message to='elizabeth@longbourn.lit/sitting_room'
    from='toetem-789@games.pemberley.lit/referee'
    type='groupchat'>
  <body>elizabeth has won the match.</body>
  <game-ended xmlns='http://metajack.im/ns/toetem'
    winner='elizabeth@longbourn.lit/sitting_room' />
</message>

```

一旦游戏开始，执X的玩家可以率先移动。每位玩家通过向裁判员发送一个携带一条`<move>`

命令的 IQ-set 节来完成一次移动。玩家向裁判员发送移动而不是将这些消息广播到房间，这样裁判员就可以检查每次移动是否合法。如果移动被接受，那么裁判员将其转发给房间的每个人。如果移动非法，那么裁判员将返回一个 IQ-error 节并且不会把该消息转发到该房间。

下面的示例给出了 Elizabeth 的首次移动。

```
<iq to='referee@pemberley.lit/toetem'
  from='elizabeth@longbourn.lit/sitting_room'
  type='set'
  id='move1'>
  <move xmlns='http://metajack.im/ns/toetem'
    col='b' row='2' />
</iq>
```

裁判员判断这次移动是合法的，并用一个 IQ-result 节来响应 Elizabeth。然后，裁判员将该移动转发给房间中的其他玩家和观众。

```
<iq to='elizabeth@longbourn.lit/sitting_room'
  from='referee@pemberley.lit/toetem'
  type='result'
  id='move1'>
  <message to='toetem-789@games.pemberley.lit'
    from='referee@pemberley.lit/toetem'
    type='groupchat'>
    <body>elizabeth has marked an x at b2.</body>
    <move xmlns='http://metajack.im/ns/toetem'
      col='b' row='2' />
  </message>
```

游戏的剩余部分继续按照该方式运行直到分出胜负。

Elizabeth 可能试图违反顺序移动或发送非法的移动。裁判员在这两种情况下都必须返回适当的错误提示。在前一种情况下，正确的错误提示是“非预期的请求”，而在后一种情况下是“不可接受的请求”。如果玩家在游戏正式启动之前发送移动，那么也可以使用非预期的请求，而如果游戏已经结束，那么返回“操作不允许”错误提示。这些错误差不多与我们已经看过的错误提示节相同，只是错误条件元素改变了。

注意，重要的是游戏房间中的所有参与者都要检查游戏状态消息的发送者。只有裁判员广播的游戏状态变化才是合法的。如果参与者没有注意到消息的起源，那么恶意玩家就能够伪装成裁判员。

这些简单的协议处理了游戏内部的大多数交互。但还有一个问题，如果新的观众在游戏中途加入房间，那么他们不知道游戏的当前状态。裁判员必须将当前游戏状态发送到所有加入该房间的人。一旦用户已经加入并接收到状态，那么该用户就能够像其他参与者一样查看游戏状态广播。

游戏的状态由游戏的阶段、玩家以及他们的块、当前的游戏板位置以及游戏的结果(如果有的话)组成。游戏的阶段可以是“正在等待”、“正在玩”或“已结束”。如果状态是“正在等待”，

那么只需要将玩家及其块通告出去。当处于“正在玩”状态时，需要通告玩家、块以及游戏板位置。而当处于“已结束”状态时，则需要所有信息。

裁判员可以在一条直接房间消息的`<game>`元素中将该信息发送给已经加入该房间的玩家。这个`<game>`元素可以使用一个 `phase` 属性来表示游戏的阶段。我们前面曾经看过的 `x-player` 和 `o-player` 属性也可以用来通告玩家及其块(如果需要的话)。最后，可以将游戏板位置保存到 `pos` 属性中(以字符串形式编码)。

Tic-Tac-Toe 游戏板由 9 个方格组成，可以在这些方格中填入 X 或 O。可以将该信息转换成一个由 9 个字符组成的字符串，每个字符串表示一个空格、X 或 O。填充游戏板方格的字符从左上角开始向右移动，然后在其他两行中继续按照该方式移动。

例如，图 11-3 中给出的位置的字符串就是“O X X O X X”。

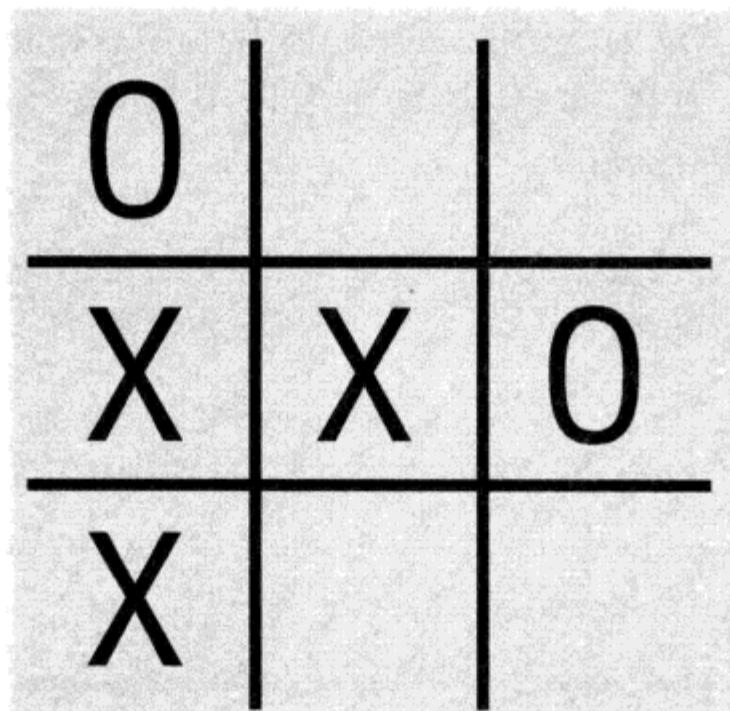


图 11-3

下面的示例给出了广播给 Bingley(他在游戏已经结束之后才加入该游戏)的游戏状态节。在其他游戏阶段加入的参与者也会看到与此十分类似的消息，只是信息量更少一些。

```

<message to='toetem-789@games.pemberley.lit/bingley'
  from='referee@pemberley.lit/toetem'
  type='chat'>
  <game-state xmlns='http://metajack.im/ns/toetem'
    phase='finished'
    x-player='elizabeth@longbourn.lit/sitting_room'
    o-player='jane@lonbourn.lit/outside'
    pos='oxoxxooxx'
    winner='elizabeth@longbourn.lit/sitting_room' />
</message>

```

如果游戏中止或平局，那么 `winner` 属性就不会出现。当游戏结束之后到来的观众可以通过查看 `pos` 属性是否完全填满移动来判断这两种状态。如果所有移动都用完了，那么游戏平局。否则，游戏中止。

现在所有加入的参与者都知道当前状态并能够跟上任何新的移动。

Toetem 协议已经完成了。该协议由大量的不同部分组成，但每一部分都非常简单。而最终结果是一个灵活的 Tic-Tac-Toe 游戏平台。

11.4 Toetem 起步

与本书中的其他应用程序不同的是，Toetem 由两大部分代码组成，即裁判员和游戏客户端。每一部分都要有自己的 HTML、CSS 和 JavaScript 文件。为了玩游戏，需要三个用户，两个玩家作为游戏客户端，一个玩家作为裁判员，为了托管每个游戏的房间，还需要多人聊天服务。

在开始实现 Toetem 协议之前，需要搭建应用程序的骨架。

程序清单 11-1 和程序清单 11-2 给出了裁判员的 HTML 和 CSS，而程序清单 11-3 给出了 JavaScript 文件的初始骨架。注意，裁判员的 UI 非常少，它只包含一个够用的 UI 以便让运行裁判员的人能够看到正在发生什么情况。



可从
Wrox.com
下载源代码

程序清单 11-1 referee.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"  
    "http://www.w3.org/TR/html4/strict.dtd">  
  
<html>  
    <head>  
        <title>Toetem Referee - Chapter 3</title>  
  
        <link rel='stylesheet' href='http://ajax.googleapis.com/ajax/libs/jqueryu  
i/1.7.2/themes/cupertino/jquery-ui.css'>  
        <script src='http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/jquery.js'>  
        </script>  
        <script src='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2  
/jquery-ui.js'></script>  
        <script src='scripts/strophe.js'></script>  
        <script src='flxhr/flXHR.js'></script>  
        <script src='scripts/strophe.flxhr.js'></script>  
  
        <link rel='stylesheet' href=referee.css'>  
        <script src='referee.js'></script>  
        <script src='board.js'></script>  
    </head>  
    <body>  
        <h1>Toetem Referee</h1>  
  
        <div id='log'>  
        </div>  
  
        <!-- login dialog -->  
        <div id='login_dialog' class='hidden'>  
            <label>JID:</label><input type='text' id='jid'>  
            <label>Password:</label><input type='password' id='password'>  
        </div>
```

```

</div>
</body>
</html>

```



可从
Wrox.com
下载源代码

程序清单 11-2 referee.css

```

body {
    font-family: Helvetica;
}

h1 {
    text-align: center;
}

.hidden {
    display: none;
}

#log {
    padding: 10px;
}

```



可从
Wrox.com
下载源代码

程序清单 11-3 referee.js(骨架)

```

var Referee = {
    connection: null
};

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Connect to XMPP',
        buttons: {
            "Connect": function () {
                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val()
                });

                $('#password').val('');
                $(this).dialog('close');
            }
        }
    });
});

$(document).bind('connect', function (ev, data) {

```

```

var conn = new Strophe.Connection(
    "http://bosh.metajack.im:5280/xmpp-httpbind");

conn.connect(data.jid, data.password, function (status) {
    if (status === Strophe.Status.CONNECTED) {
        $(document).trigger('connected');
    } else if (status === Strophe.Status.DISCONNECTED) {
        $(document).trigger('disconnected');
    }
});

Referee.connection = conn;
});

$(document).bind('connected', function () {
    // nothing here yet
});

```

程序清单 11-4、程序清单 11-5 和程序清单 11-6 给出了游戏客户端的 HTML、CSS 和骨架 JavaScript 代码。登录对话框有一个额外的 Referee 输入框用来向客户端提供该服务的裁判员位置。UI 有两个主要部分，一个用于用户尚未玩游戏或观看游戏的时候，一个用于他们正在参与游戏时。两个界面都非常简单。参与者 UI 有游戏板和聊天区域，而非参与者 UI 则包含正在等待玩家列表和活跃游戏列表。图 11-1 和图 11-2 给出了这两个布局。



程序清单 11-4 totem.html

可从
Wrox.com
下载源代码

```

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

<html>
    <head>
        <title>Toetem - Chapter 11</title>

        <link rel='stylesheet' href='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/themes/cupertino/jquery-ui.css'>
        <script src='http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/jquery.js'>
        </script>
        <script src='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/jquery-ui.js'></script>
        <script src='scripts/strophe.js'></script>
        <script src='flxhr/flXHR.js'></script>
        <script src='scripts/strophe.flxhr.js'></script>

        <link rel='stylesheet' href='totem.css'>
        <script src='totem.js'></script>
    </head>
    <body>
        <h1>Toetem</h1>

        <div id='browser'>

```

```
<div class='buttons'>
  <input id='disconnect' class='button' type='button' value='disconnect'>
  <input id='wait' class='button' type='button' value='wait'>
</div>

<h2>Players Available</h2>
<table id='waiting'>
  <thead>
    <tr>
      <th>Player</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    </tbody>
</table>

<h2>Games In Progress</h2>
<table id='games'>
  <thead>
    <tr>
      <th>X</th>
      <th>O</th>
      <th>Room</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    </tbody>
</table>
</div>

<div id='game' class='hidden'>
  <div class='buttons'>
    <input id='resign' class='button' type='button'
           value='resign' disabled='disabled'>
    <input id='leave' class='button' type='button'
           value='leave' disabled='disabled'>
  </div>

  <div id='board-area'>
    <canvas id='board' width='300' height='300'>
    </canvas>
    <div id='board-status'></div>
  </div>

  <div id='chat'>
    <div id='messages'>
    </div>
    <input type='text' id='input'>
  </div>
</div>
```

```

<!-- login dialog -->
<div id='login_dialog' class='hidden'>
    <label>JID:</label><input type='text' id='jid'>
    <label>Password:</label><input type='password' id='password'>
    <label>Toetem Referee</label><input type='text' id='referee'>
</div>
</body>
</html>

```



可从
Wrox.com
下载源代码

程序清单 11-5 toetem.css

```

body {
    font-family: Helvetica;
}

h1 {
    text-align: center;
}

.hidden {
    display: none;
}

.buttons {
    text-align: center;
}

input.button {
    margin: 10px;
}

#board {
    width: 300px;
    height: 300px;
    background-color: #000;
}

#board-area {
    float: left;
}

#chat {
    padding-left: 25px;
    float: left;
}

#messages {
    padding: 5px 10px;
    width: 400px;
    height: 300px;
    background-color: #ddd;
}

```

```

.messages .system {
    text-align: right;
    font-style: italic;
    font-weight: bold;
    color: #999;
}

#input {
    width: 400px;
}

.nick {
    color: #00c;
}

.me {
    color: #c00;
}

```



可从
Wrox.com
下载源代码

程序清单 11-6 toetem.js(骨架)

```

var Toetem = {
    connection: null,
    referee: null
};

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Connect to XMPP',
        buttons: {
            "Connect": function () {
                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val(),
                    referee: $('#referee').val()
                });

                $('#password').val('');
                $(this).dialog('close');
            }
        }
    });
});

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        "http://bosh.metajack.im:5280/xmpp-httpbind");

    conn.connect(data.jid, data.password, function (status) {

```

```

        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });

    Toetem.connection = conn;
    Toetem.referee = data.referee;
});

$(document).bind('connected', function () {
    // nothing here yet
});

$(document).bind('disconnected', function () {
    // nothing here yet
});

```

当把这些初始的工作做好之后，我们就可以开始实现该协议的最初部分。

11.5 实现会话和等待列表

实现 Toetem 协议的第一步是实现会话跟踪和等待玩家列表。在玩家能够与裁判员交互之前需要进行会话跟踪，而等待列表是 Toetem 中最容易实现的部分。因为我们在本章中同时构建服务器和客户端，所以我们同时实现两部分。

裁判员必须跟踪玩家的出席信息并维护一个正在等待游戏的玩家列表。裁判员将根据玩家的请求将其添加到该列表中以及从中删除。每当玩家注册时必须将这个列表发送给所有玩家，而且该列表发生变化时，也必须将变化信息广播到所有玩家。

用户必须向裁判员发送定向出席消息来建立游戏会话并获取游戏功能的访问权。一旦裁判员跟踪玩家的出席状态，该玩家就可以通过单击 UI 中的不同按钮来查询该列表，将自己添加到该列表或从该列表中删除。

11.5.1 裁判员(版本 1)

首先，裁判员需要数据结构来跟踪玩家的出席状态并保存等待列表。此外，顺便创建一个 Toetem 协议命名空间常量以减少输入。将下面的代码添加到 Referee 对象中。



```

waiting: [],
presence: {},
NS_TOETEM: "http://metajack.im/ns/toetem"

```

code snippet referee.js

等待列表将被保存到一个数组中，而玩家的出席信息则被保存到一个字典对象中(以它们的 JID 为键)。

为了跟踪出席信息并回答玩家的查询，裁判员需要用来处理<presence>和<iq>节的处理程序。对 connected 事件处理程序进行如下的修改以建立这些处理程序。



可从
Wrox.com
下载源代码

```
$(document).bind('connected', function () {
    var conn = Referee.connection;
    $('#log').prepend("<p>Connected as " + conn.jid + "</p>");

    conn.addHandler(Referee.on_presence, null, "presence");
    conn.addHandler(Referee.on_iq, null, "iq");

    conn.send($pres());
});
```

[code snippet referee.js](#)

这两个处理程序都需要实现。

将下面的 on_presence() 函数添加到对象中。



可从
Wrox.com
下载源代码

```
on_presence: function (pres) {
    var from = $(pres).attr('from');
    var bare_from = Strophe.getBareJidFromJid(from);
    var type = $(pres).attr('type');
    var bare_jid = Strophe.getBareJidFromJid(Referee.connection.jid);

    // only look for direct presence from other users
    if ((!type || type === "unavailable") && from !== bare_jid) {
        if (type === "unavailable") {
            delete Referee.presence[bare_from];

            // remove from lists
            Referee.remove_waiting(from);

            $('#log').prepend("<p>Unregistered " + bare_from + ".</p>");
        } else if ($(pres).find('register').length > 0) {
            Referee.presence[bare_from] = from;

            $('#log').prepend("<p>Registered " + bare_from + ".</p>");

            Referee.send_waiting(from);
        }
    }

    return true;
}
```

[code snippet referee.js](#)

on_presence() 函数忽略任何来自裁判员自身或它的其他资源的<presence>节。对于包含<register>的可访问<presence>节，它将把发送者添加到被跟踪玩家列表中并发送等待玩家列表。

对于不可访问节，裁判员将发送者从被跟踪玩家列表中删除并将它们从等待列表中删除(万一他们忘记将自己删除)。无论是哪一种情况，裁判员都必须确保用美观的日志消息让其用户得到通知。

注意，等待列表和出席跟踪使用的是裸 JID。这限制用户只使用单个已连接资源与 Toetem 系统进行交互。可以扩展裁判员程序以支持单个用户多个游戏会话，但这可能会让交互更加复杂。这里选择了简单的解决方案。

`remove_waiting()`函数的功能正如其名称所暗示的那样，并将等待列表变化广播给所有已注册玩家。如果被请求的 JID 并不在列表中，那么该函数不做任何处理。应该将下面的实现添加到 `Referee` 对象中。



可从
Wrox.com
下载源代码

```
remove_waiting: function (jid) {
    var bare_jid = Strophe.getBareJidFromJid(jid);

    var i;
    for (i = 0; i < Referee.waiting.length; i++) {
        var wjid = Strophe.getBareJidFromJid(Referee.waiting[i]);
        if (wjid === bare_jid) {
            break;
        }
    }

    if (i < Referee.waiting.length) {
        Referee.waiting.splice(i, 1);

        Referee.broadcast(function (msg) {
            return msg.c('not-waiting', {xmlns: Referee.NS_TOETEM})
                .c('player', {jid: jid});
        });
    }

    $('#log').prepend("<p>Removed " + bare_jid + " from " +
        "waiting list</p>");
}
```

code snippet referee.js

该代码简单地扫描整个列表直到找到匹配的 JID，而当找到匹配项时，将该项从数组中删除，并使用 `broadcast()` 来通告所有玩家。`broadcast()` 函数携带一个函数，它用来向即将发送给所有已注册玩家的

message

节中添加内容。这样一来，`broadcast()` 就使得我们很容易发送通知。将 `broadcast()` 函数添加到 `Referee` 对象中。



可从
Wrox.com
下载源代码

```
broadcast: function (func) {
    $.each(Referee.presence, function () {
        var msg = func($msg({to: this}));
        Referee.connection.send(msg);
    });
}
```

code snippet referee.js

裁判员还需要向发出非法请求或非预期请求的客户端发送错误提示。创建下面的函数 `send_error()`，并将其添加到 `Referee` 对象中。



可从
Wrox.com
下载源代码

```
send_error: function (iq, etype, ename, app_error) {
    var error = $iq({to: $(iq).attr('from'),
                     id: $(iq).attr('id'),
                     type: 'error'})
        .cnode(iq.cloneNode(true)).up()
        .c('error', {type: etype})
        .c(ename, {xmlns: Strophe.NS.STANZAS}).up();

    if (app_error) {
        error.c(app_error, {xmlns: Referee.NS_TOELEM});
    }

    Referee.connection.send(error);
}
```

code snippet referee.js

这个函数相当方便，这是因为裁判员将必须处理大量的错误情况。

因为裁判员经常需要检查玩家是否在等待列表中，所以 `is_waiting()` 函数也有用处。将下面的代码添加到 `Referee` 对象中以实现该功能。



可从
Wrox.com
下载源代码

```
is_waiting: function (jid) {
    var bare_jid = Strophe.getBareJidFromJid(jid);

    var i;
    for (i = 0; i < Referee.waiting.length; i++) {
        var wjid = Strophe.getBareJidFromJid(Referee.waiting[i]);
        if (wjid === bare_jid) {
            return true;
        }
    }

    return false;
}
```

code snippet referee.js

就像 `remove_waiting()` 函数一样，`is_waiting()` 通过遍历该列表来搜索玩家的 JID。

现在可以实现 `on_iq()` 函数，它负责处理许多与玩家的交互。在这个版本中，只实现了 `waiting` 命令和 `stop-waiting` 命令。将下面的代码添加到 `Referee` 对象中。



可从
Wrox.com
下载源代码

```
on_iq: function (iq) {
    var id = $(iq).attr('id');
    var from = $(iq).attr('from');
    var type = $(iq).attr('type');

    // make sure we know the user's presence first
```

```

if (!Referee.presence[Strophe.getBareJidFromJid(from)]) {
    Referee.send_error(iq, 'auth', 'forbidden');
} else {
    var child = $(iq).find('*[xmlns="" + Referee.NS_TOETEM +
                           ""]:first');
    if (child.length > 0) {
        if (type === 'get') {
            Referee.send_error(iq, 'cancel', 'bad-request');
            return true;
        } else if (type !== 'set') {
            // ignore IQ-error and IQ-result
            return true;
        }
    }
    switch (child[0].tagName) {
        case 'waiting':
            Referee.on_waiting(id, from, child);
            break;
        case 'stop-waiting':
            Referee.on_stop_waiting(id, from, child);
            break;
        default:
            Referee.send_error(iq, 'cancel', 'bad-request');
    }
} else {
    Referee.send_error(iq, 'cancel', 'feature-not-implemented');
}
}

return true;
}

```

code snippet referee.js

这个函数首先检查裁判员是否正在跟踪发出请求的用户，如果不是这样的话就发送一条适当的错误提示消息。如果请求来自于一个被跟踪的客户端，那么该函数在 IQ-set 节中搜索 Toetem 命令，如果没有找到，就响应一条错误提示消息。对于 waiting 命令，on_iq() 将该行为委托给 on_waiting()，而 stop-waiting 命令则委托给 on_stop_waiting()。

下面的代码实现了最后两个函数 on_waiting() 和 on_stop_waiting()。

```

on_waiting: function (id, from, elem) {
    // if they were already waiting, remove them so their resource
    // can be updated
    if (Referee.is_waiting(from)) {
        Referee.remove_waiting(from);
    }

    Referee.waiting.push(from);
}

```



可以从
Wrox.com
下载源代码

```

Referee.connection.send($iq({to: from, id: id, type: 'result'}));

Referee.broadcast(function (msg) {
    return msg.c('waiting', {xmlns: Referee.NS_TOETEM})
        .c('player', {jid: from});
});

$('#log').prepend("<p>Added " +
    Strophe.getBareJidFromJid(from) + " to " +
    "waiting list.</p>");

on_stop_waiting: function (id, from, elem) {
    if (Referee.is_waiting(from)) {
        Referee.remove_waiting(from);
    }

    Referee.connection.send($iq({to: from, id: id, type: 'result'}));
}

```

code snippet referee.js

`on_waiting()` 函数将用户添加到等待列表中，并注意到如果他们已经在等待，那么将他们删除。它还将等待列表的变化广播到所有玩家。最后，`on_stop_waiting()` 函数将该用户从等待列表中删除。

注意，这些函数总是返回成功的响应。如果能够简单地将动作忽略，那就没有必要向用户发送错误提示信息。如果出于某种原因，用户的客户端未能同步，那么返回成功响应通常能够纠正该问题。裁判员对它接受的信息的检查比较宽松，而对于它发送的消息则非常严格，就像任何优秀的 Internet 服务或客户端应用程序应该做的那样。

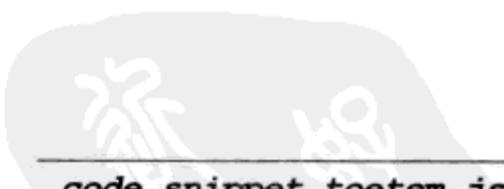
裁判员的第一个版本现在已经可以运行了，但我们首先必须实现初始的 Toetem 客户端。

11.5.2 Toetem 客户端(版本 1)

就像裁判员一样，Toetem 客户端也需要一个协议常量，在发出的每个 XMPP 节中几乎都用到了该协议的命名空间。将下面的 `NS_TOETEM` 属性添加到 `Toetem` 对象中。



`NS_TOETEM: "http://metajack.im/ns/toetem"`

**code snippet toetem.js**

当客户端连接时，它必须向裁判员发送定向出席消息来启动一个游戏会话。一旦向裁判员注册完毕，它还必须处理传入的等待玩家列表。在断开连接之后它必须清除该列表，重置应用程序状态，然后重新打开登录对话框。将下面的被突出显示代码添加到 `connected` 和 `disconnected` 事件处理程序中。



```

$(document).bind('connected', function () {
    $('#disconnect').removeAttr('disabled');
    $('#wait').removeAttr('disabled');

    Toetem.connection.addHandler(Toetem.on_message, null, "message");

    // tell the referee we're online
    Toetem.connection.send(
        $pres({to: Toetem.referee})
        .c('register', {xmlns: Toetem.NS_TOETEM}));
});

$(document).bind('disconnected', function () {
    Toetem.referee = null;
    Toetem.connection = null;

    $('#waiting tbody').empty();
    $('#games tbody').empty();

    $('#login_dialog').dialog('open');
});

```

code snippet toetem.js

`on_message()`处理程序侦听初始的等待列表以及变化通知。下面给出了它的实现，应该将这些代码添加到 `Toetem` 对象中。



```

on_message: function (message) {
    var from = $(message).attr('from');

    if ($(message).find('waiting').length > 0) {
        $(message).find('waiting > player').each(function () {
            $('#waiting tbody').append(
                "<tr><td class='jid'>" +
                $(this).attr('jid') +
                "</td><td>" +
                ($(this).attr('jid') === Toetem.connection.jid ?
                    "<input type='button' class='stop_button' " +
                    "value='stop waiting'>" :
                    "<input type='button' class='start_button' " +
                    "value='start game'>") +
                "</td></tr>");
        });
    } else if ($(message).find('not-waiting').length > 0) {
        $(message).find('not-waiting > player').each(function () {
            var jid = $(this).attr('jid');
            $('#waiting td.jid').each(function () {
                if ($(this).text() === jid) {
                    $(this).parent().remove();
                    return false;
                }
            });
        });
    }
}

```

```

        });
    });

    return true;
}

```

code snippet toetem.js

当接收到<waiting>或<not-waiting>消息时，必须更新等待列表。注意，在等待玩家的 JID 旁边显示了一个动作按钮。如果列表中的玩家就是该用户，那么单击这个按钮就会把该用户从等待列表中删除。否则，这个按钮将启动与这个玩家的游戏。可以将下面的事件处理程序添加到文档准备就绪事件处理程序中以实现 Stop Waiting 按钮。



可从
Wrox.com
下载源代码

```

$('input.stop_button').live('click', function () {
    Toetem.connection.sendIQ(
        $iq({to: Toetem.referee, type: "set"})
            .c('stop-waiting', {xmlns: Toetem.NS_TOETEM}));
});

```

code snippet toetem.js

还应该实现这个屏幕上的 Disconnect 和 Wait 按钮的处理程序。将下面的代码添加到文档准备就绪事件处理程序中。



可从
Wrox.com
下载源代码

```

$('#disconnect').click(function () {
    Toetem.connection.disconnect();
});

$('#wait').click(function () {
    Toetem.connection.sendIQ(
        $iq({to: Toetem.referee, type: "set"})
            .c("waiting", {xmlns: Toetem.NS_TOETEM}));
});

```

code snippet toetem.js

Wait 按钮用来向裁判员发送携带<waiting>命令的适当 IQ-set 节。这不需要更新屏幕上的等待列表，这是因为裁判员将会广播该变化、触发 on_message() 处理程序并导致这条新信息的出现。

这里完成了裁判员和游戏客户端的第一个版本。如果启动一个裁判员实例，并打开几个指向它的游戏客户端，那么应该能够监视并操纵等待玩家列表。

11.6 实现游戏管理

为了完成游戏系统，下一步需要做的就是实现游戏管理功能以及游戏聊天。

对于裁判员而言，这包括创建玩家之间的新游戏、为这些游戏建立聊天室以及邀请玩家到房间中进行交互。裁判员还必须处理活跃游戏列表并在游戏开始和结束时让这个新列表、等待玩家列表以及玩家保持同步。

需要激活 Toetem 客户端的游戏页面，也要启用它的各种小部件。还必须实现我们在前一节中看到的 Start Game 按钮。最后，客户端需要能够发送和接收房间中的普通聊天消息。

11.6.1 裁判员(版本 2)

增强的裁判员现在必须跟踪活跃游戏，而且还将与多人聊天服务器交互以建立游戏房间。将下面的新属性添加到 Referee 对象中，这些代码添加了 games 字典以及 MUC 协议命名空间常量。



可从
Wrox.com
下载源代码

```
games: {},  
NS_MUC: "http://jabber.org/protocol/muc",  
NS_MUC_USER: "http://jabber.org/protocol/muc#user",  
NS_MUC_OWNER: "http://jabber.org/protocol/muc#owner",  
MUC_SERVICE: 'games.pemberley.lit'
```

code snippet referee.js

注意，必须将 MUC_SERVICE 的值修改为适当的多人聊天服务，这里使用的值只是一个占位符。您喜欢的 XMPP 服务器可能在 conference.domain.com 运行服务。

Referee 的 on_iq() 处理程序(已在前面建立)需要响应两种新命令，即 start 和 resign。这些新命令将分别用于启动新游戏和结束游戏。将下面的新 case 语句添加到 on_iq() 处理程序的语句中。



可从
Wrox.com
下载源代码

```
case 'start':  
    Referee.on_game_start(iq, id, from, child);  
    break;  
  
case 'resign':  
    Referee.on_resign(iq, id, from);  
    break;
```

code snippet referee.js

现在实现 on_game_start() 和 on_resign()。

下面的 on_game_start() 实现非常简单，但这是因为它依靠其他函数来完成具体的工作。该函数首先检查涉及的玩家还没有玩游戏，将它们从等待列表中删除，并建立游戏房间。将下面的代码添加到 Referee 对象中。



```

on_game_start: function (iq, id, from, elem) {
    var with_jid = $(elem).attr('with');
    var with_bare = Strophe.getBareJidFromJid(with_jid);

    // check that the players are available
    if (!Referee.is_waiting(with_jid)) {
        Referee.send_error(iq, 'modify', 'item-not-found');
        return;
    }

    if (Referee.is_playing(with_jid) ||
        Referee.is_playing(from)) {
        Referee.send_error(iq, 'cancel', 'not-allowed');
        return;
    }

    Referee.connection.send($iq({to: from, id: id, type: 'result'}));

    // remove players from waiting list
    Referee.remove_waiting(from);
    Referee.remove_waiting(with_jid);

    // create game room and invite players
    Referee.create_game(from, with_jid);
}

```

code snippet referee.js

在上面的代码中引用了两个新函数，即 `is_playing()` 和 `create_game()`。

与 `is_waiting()` 函数类似，`is_playing()` 检查用户是否已经在玩游戏。将它的实现添加到 `Referee` 对象中。



```

is_playing: function (jid) {
    var bare = Strophe.getBareJidFromJid(jid);

    var found = false;
    $.each(Referee.games, function () {
        if (Strophe.getBareJidFromJid(this.x_player) === jid ||
            Strophe.getBareJidFromJid(this.o_player) === jid) {
            found = true;
            return false;
        }
    });
    return found;
}

```

code snippet referee.js

`create_game()`函数更加复杂一些。因为裁判员没有与多人聊天服务结合起来，所以有可能裁判员试图建立的房间已经存在。为了处理这种可能性，裁判员应选取一个随机的房间名称并试着建立该房间。如果该房间已经存在，那么裁判员应离开该房间并再次开始。一旦新房间已经建立，裁判员就初始化游戏并要求玩家加入房间。

将下面的 `create_game()` 代码添加到 `Referee` 对象中。



可以从
Wrox.com
下载源代码

```

create_game: function (player1, player2) {
    // generate a random room name, and make sure it
    // doesn't already exist to our knowledge
    var room;
    do {
        room = "" + Math.floor(Math.random() * 1000000);
    } while (Referee.games[room]);

    var room_jid = room + "@" + Referee.MUC_SERVICE + "/Referee";
    Referee.connection.addHandler(function (presence) {
        var game;

        if ($(presence).find('status[code="201"]').length > 0) {
            // room was freshly created
            game = Referee.new_game();
            game.room = room;

            // create initial game state with randomized sides
            if (Math.random() < 0.5) {
                game.x_player = player1;
                game.o_player = player2;
                Referee.games[room] = game;
            } else {
                game.x_player = player2;
                game.o_player = player1;
                Referee.games[room] = game;
            }

            // invite players to start the game
            Referee.invite_players(game);

            // notify everyone about the game
            Referee.broadcast(function (msg) {
                return msg.c('games', {xmlns: Referee.NS_TOELEM})
                    .c('game', {'x-player': game.x_player,
                                'o-player': game.o_player,
                                'room': Referee.game_room(room)});
            });

            $('#log').prepend("<p>Created game room " + room + ".</p>");
        } else {
            // room was already in use, we need to start over
            Referee.connection.send(
                $pres({to: room_jid, type: 'unavailable'}));
            Referee.create_game(player1, player2);
        }
    });
}

```

```

        }
        return false;
    }, null, "presence", null, null, room_jid);

Referee.connection.send(
    $pres({to: room_jid})
        .c("x", {xmlns: Referee.NS_MUC}));
}

```

code snippet referee.js

最初的游戏状态是通过调用 `new_game()` 来建立的。一旦游戏对象存在，裁判员就可以随机地将两个玩家分别指派到一方(要么是 X 要么是 O)，邀请玩家加入游戏，并将该游戏已经存在的消息广播给所有已注册玩家。

在这个函数中定义的`<presence>`处理程序检查方面的特殊的 201 状态码。这个状态(之前在第 8 章讨论过)是新建房间时发送的。这个处理程序只需要检查这个首个`<presence>`节，因此它返回 `false` 并在完成后被删除。

看看 `new_game()` 的代码。我们已经见过 `room`、`x_player` 和 `o_player` 属性，但这里还有其他几个属性。



可从
Wrox.com
下载源代码

```

new_game: function () {
    return {
        room: null,
        waiting: 2,
        status: 'starting',
        x_player: null,
        o_player: null,
        winner: null
    };
}

```

code snippet referee.js

这些属性大都是自描述性的。`waiting` 属性的值从 2 开始，这是因为裁判员等待两位玩家加入该房间才能开始该游戏。每次玩家加入该房间时，这个值就会递减 1。当 `waiting` 值变为 0 时，游戏就正式开始了。`Status` 属性描述游戏的状态。它可能包含的值为 `starting`、`playing`、`finished` 和 `aborted`。当我们实现裁判员的更多功能时将详细讨论这些值。最后，`winner` 属性用来在游戏结束时存放赢得游戏的玩家的 JID。

在建立游戏房间和对象之后，必须邀请两位玩家加入该房间。将下面的代码添加到 `Referee` 对象中，该代码用来发送邀请函。



可从
Wrox.com
下载源代码

```

invite_players: function (game) {
    // send room invites
    $.each([game.x_player, game.o_player], function () {
        Referee.connection.send(

```

```

        $msg({to: game.room + "@" + Referee.MUC_SERVICE})
            .c('x', {xmlns: Referee.NS_MUC_USER})
            .c('invite', {to: this}));
    });
}

```

code snippet referee.js

当每位玩家加入游戏房间时，裁判员将会从他们那里接收到<presence>节。我们需要修改裁判员的 `on_presence()` 处理程序来查找这些 XMPP 节，而且一旦两位玩家都已加入就立即启动游戏。此外，如果某位玩家在游戏结束之前离开，那么必须惩罚该玩家并结束游戏。因此，`on_presence()` 处理程序必须查看房间中玩家的可访问出席信息和不可访问出席信息。

修改裁判员的 `on_presence()` 处理程序以匹配下面的实现。修改过的代码行已被突出显示。



可从
Wrox.com
下载源代码

```

on_presence: function (pres) {
    var from = $(pres).attr('from');
    var bare_from = Strophe.getBareJidFromJid(from);
    var type = $(pres).attr('type');
    var bare_jid = Strophe.getBareJidFromJid(Referee.connection.jid);
    var domain = Strophe.getDomainFromJid(from);

    if (domain === Referee.MUC_SERVICE) {
        // handle room presence
        var room = Strophe.getNodeFromJid(from);
        var player = Strophe.getResourceFromJid(from);
        var game = Referee.games[room];

        // make sure it's a game and player we care about
        if (game &&
            (game.status === 'starting' || game.status === 'playing') &&
            (player === game.x_player || player === game.o_player)) {
            if (game.status === 'starting') {
                if (type !== 'unavailable') {
                    // waiting for one less player; if both are
                    // now present, the game is started
                    game.waiting -= 1;

                    $('#log').prepend("<p>Player " + bare_from +
                        " arrived to game " + game.room +
                        ".</p>");

                    if (game.waiting === 0) {
                        Referee.start_game(game);
                    }
                } else {
                    // one of the players left before the game even
                    // started, so abort the game
                    Referee.end_game(game, 'aborted');
                }
            } else {

```

```

        // during play, forfeit a player if they leave the room
        if (type === 'unavailable') {
            if (player === game.x_player) {
                game.winner = game.o_player;
            } else {
                game.winner = game.x_player;
            }

            Referee.end_game(game, 'finished');
        }
    }
}

} else if ((!type || type === "unavailable") &&
    bare_from !== bare_jid) {
    // handle directed presence from players
    if (type === "unavailable") {
        delete Referee.presence[bare_from];

        // remove from lists
        Referee.remove_waiting(bare_from);

        $('#log').prepend("<p>Unregistered " + bare_from + ".</p>");
    } else {
        Referee.presence[bare_from] = from;

        $('#log').prepend("<p>Registered " + bare_from + ".</p>");

        Referee.send_waiting(from);
        Referee.send_games(from);
    }
}

return true;
}

```

code snippet referee.js

新的 if 子句查找来自于多人聊天服务所在域的<presence>节。从 JID 中提取房间和发送者，并查找游戏对象。如果游戏已经存在，处于活跃状态，而且该节来自其中一位玩家，那么该节说明需要更进一步的处理。

当处于 starting 状态时，裁判员正等待玩家加入游戏。一旦两位玩家都已加入，它就调用 start_game()，状态变为 playing。如果一位玩家加入然后在另一位玩家到来之前离开，那么调用 end_game() 中止游戏。

当处于 playing 状态时，裁判员将观察提前退出的玩家，该玩家将受到惩罚。

在实现 start_game() 和 end_game() 之前，必须实现 send_games() 来发送初始游戏列表(就像当玩家注册时发送的当前等待列表一样)。将 send_games() 函数以及下面的辅助函数添加到 Referee 对象中。这些辅助函数用来构造游戏房间的 JID 并向该房间发送消息。



```

send_games: function (jid) {
    var msg = $msg({to: jid})
        .c('games', {xmlns: Referee.NS_TOETEM});

    $.each(Referee.games, function (room) {
        msg.c('game', {'x-player': this.x_player,
                      'o-player': this.o_player,
                      'room': Referee.game_room(room)}).up();
    });

    Referee.connection.send(msg);
},
game_room: function (game) {
    return game.room + "@" + Referee.MUC_SERVICE;
},
muc_msg: function (game) {

    return $msg({to: Referee.game_room(game.room), type: "groupchat"});
}

```

code snippet referee.js

`send_game()`构建当前活跃游戏列表并将其发送给玩家。`game_room()`函数携带基本的房间名称并为该游戏的房间构造适当的 JID。这个函数由 `muc_msg()` 使用，它返回一个部分构建的节(用来与房间的房客通信)。因为裁判员会向游戏房间发送多种不同的消息，所以这个函数可以帮助减少编程工作量。

`start_game()`函数将`<game-started>`消息广播到房间并将游戏的状态设为 `playing`。将该函数的实现添加到 `Referee` 对象中。



```

start_game: function (game) {
    game.status = 'playing';
    Referee.connection.send(
        Referee.muc_msg(game)
            .c('body').t('The match has started.').up()
            .c('game-started', {xmlns: Referee.NS_TOETEM,
                                'x-player': game.x_player,
                                'o-player': game.o_player}));

    $('#log').prepend("<p>Started game " + game.room + ".</p>");
}

```

code snippet referee.js

裁判员同时包含`<game-started>`元素和普通的`<body>`元素。这在房间中提供了人类可读的消息，同时还为客户端代码提供了机器友好的触发器。

`end_game()`函数稍微有点长，这是因为它需要向房间发送一条类似的消息，清除游戏并离开房间。将下面的 `end_game()` 代码添加到 Referee 对象中。



可从
Wrox.com
下载源代码

```

end_game: function (game, status) {
    game.status = status;

    // let room know the result of the game
    var attrs = {xmlns: Referee.NS_TOETEM};
    if (game.winner) {
        attrs.winner = game.winner;
    }

    var msg = "";
    if (game.winner) {
        msg += Strophe.getBareJidFromJid(game.winner) +
            " has won the match.";
    } else if (status === 'finished') {
        msg += "The match was tied.";
    } else {
        msg += "The match was aborted.";
    }

    Referee.connection.send(
        Referee.muc_msg(game)
            .c('body').t(msg).up()
            .c('game-ended', attrs));

    // delete the game
    delete Referee.games[game.room];

    // leave the room
    Referee.connection.send(
        $pres({to: game.room + "@" + Referee.MUC_SERVICE + "/Referee",
            type: "unavailable"}));

    // notify all the players
    Referee.broadcast(function (msg) {
        return msg.c('game-over', {xmlns: Referee.NS_TOETEM})
            .c('game', {'x-player': game.x_player,
                'o-player': game.o_player,
                'room': Referee.game_room(game.room)}));
    });

    $('#log').prepend("<p>Finished game " + game.room + ".</p>");
}

```

code snippet referee.js

游戏的赢家(如果说有的话)也包含在发送到房间的`<game-ended>`元素中。

注意，即使裁判员已经离开而且游戏已经结束，只要玩家愿意，他们仍然可以继续在该房间中聊天。

这个版本的裁判员的最后一项任务是实现 `on_resign()` 以处理玩家放弃游戏的情况。记得在本节前面我们在 `on_iq()` 处理程序中添加的三个新函数中就有 `on_resign()` 函数。

裁判员必须首先找到玩家希望放弃的游戏。这是通过辅助函数 `find_game()` 实现的。将下面的 `find_game()` 和 `on_resign()` 代码添加到 `Referee` 对象中。



可从
Wrox.com
下载源代码

```

find_game: function (player) {
    var game = null;
    $.each(Referee.games, function (r, g) {
        if (g.x_player === player || g.o_player === player) {
            game = g;
            return false;
        }
    });
    return game;
},
on_resign: function (iq, id, from) {
    var game = Referee.find_game(from);
    if (!game || game.status === 'finished' ||
        game.status === 'aborted' ||
        game.status === 'starting') {
        Referee.send_error(iq, 'cancel', 'bad-request');
    } else {
        if (from === game.x_player) {
            game.winner = game.o_player;
        } else {
            game.winner = game.x_player;
        }
        Referee.end_game(game, 'finished');
        Referee.connection.send($iq({to: from, id: id, type: 'result'}));
        $('#log').prepend("<p>" + Strophe.getBareJidFromJid(from) +
                           " resigned game " + game.room + ".</p>");
    }
}

```

code snippet referee.js

`on_resign()` 函数确保游戏存在并处于 `playing` 状态。否则，它发送一条错误提示消息。另一位玩家被设为该游戏的赢家，然后调用之前实现的 `end_game()` 函数来结束游戏。注意，该玩家的放弃游戏的 IQ-set 节需要一个对应的 IQ-result 应答，在该函数末尾发送了该应答。

裁判员现在支持游戏管理功能了。唯一缺少的就是游戏逻辑和游戏中的事件。

11.6.2 Toetem 客户端(版本 2)

因为裁判员支持创建游戏，所以可以实现等待玩家列表以及活跃游戏列表中的 Start Game

按钮。开始游戏将导致裁判员要求该玩家加入新游戏房间，而客户端需要侦听这些邀请并加入房间。然后客户端必须转入游戏模式，而我们将让游戏页面的按钮和聊天区域发挥作用。

客户端需要记录它目前参与的游戏。将 game 属性添加到 Toetem 对象中。



可从
Wrox.com
下载源代码

game: null

code snippet toetem.js

启动游戏的主要路径是通过 Start Game 按钮。当单击按钮时，该按钮的处理程序向裁判员发送一条<start>命令。将下面的处理程序添加到客户端的文档准备就绪事件处理程序中。



可从
Wrox.com
下载源代码

```
$('input.start_button').live('click', function () {
    Toetem.connection.sendIQ(
        $iq({to: Toetem.referee, type: "set"})
            .c('start', {xmlns: Toetem.NS_TOETEM,
                "with": $(this).parent().prev().text()});
});
```

code snippet toetem.js

注意，玩家的名字是从 HTML 等待玩家表格的前一个<td>元素中获取的。

最终裁判员将向游戏房间发送一封发给该玩家的邀请函。因为这封邀请函是作为<message>节发送的，所以我们将修改 on_message() 处理程序来处理它。与此同时，我们还可以添加对裁判员广播的活跃游戏通知的支持。对 on_message() 函数进行如下的修改(被突出显示部分)。



可从
Wrox.com
下载源代码

```
on_message: function (message) {
    var from = $(message).attr('from');

    if ($(message).find('waiting').length > 0) {
        $(message).find('waiting > player').each(function () {
            $('#waiting tbody').append(
                "<tr><td class='jid'>" +
                $(this).attr('jid') +
                "</td><td>" +
                ($(this).attr('jid') === Toetem.connection.jid ?
                    "<input type='button' class='stop_button' " +
                    "value='stop waiting'>" :
                    "<input type='button' class='start_button' " +
                    "value='start game'") +
                "</td></tr>");
        });
    } else if ($(message).find('not-waiting').length > 0) {
        $(message).find('not-waiting > player').each(function () {
            var jid = $(this).attr('jid');
            $('#waiting td.jid').each(function () {
```

```

        if ($(this).text() === jid) {
            $(this).parent().remove();
            return false;
        }
    });
});

} else if ($(message).find('games').length > 0) {
    $(message).find('games > game').each(function () {
        if ($(this).attr('x-player') !== Toetem.connection.jid &&
            $(this).attr('o-player') !== Toetem.connection.jid) {
            $('#games tbody').append(
                "<tr><td>" +
                $(this).attr('x-player') +
                "</td><td>" +
                $(this).attr('o-player') +
                "</td><td class='jid'>" +
                $(this).attr('room') +
                "</td><td>" +
                "<input type='button' class='watch_button' " +
                "value='watch game'>" +
                "</td></tr>");
        }
    });
}

} else if ($(message).find('game-over').length > 0) {
    $(message).find('game-over > game').each(function () {
        var jid = $(this).attr('room');
        $('#games td.jid').each(function () {
            if ($(this).text() === jid) {
                $(this).parent().remove();
                return false;
            }
        });
    });
}

} else if ($(message)
    .find('x > invite').attr('from') === Toetem.referee) {
    Toetem.game = from;

    $('#messages').empty();
    $('#messages').append("<div class='system'>" +
        "Joined game #" +
        Strophe.getNodeFromJid(from) +
        "</div>");

    Toetem.scroll_chat();

    $('#wait').removeAttr('disabled');
    $('#browser').hide();
    $('#game').show();

    var nick = Toetem.connection.jid;
    Toetem.connection.send(
        $pres({to: Toetem.game + '/' + nick}))
}

```

```

        .c('x', {xmlns: Toetem.NS_MUC})));
    }

    return true;
}

```

code snippet toetem.js

当接收到游戏通知时，程序相应地更新屏幕上的活跃游戏列表。当接收到游戏邀请函时，`on_message()`将 `game` 属性设为该房间的 JID，将客户端切换到游戏屏幕，并加入该房间。在切换到游戏屏幕之前，清除消息区域并添加一条简短的消息告诉用户发生的情况。将浏览器屏幕隐藏起来，游戏屏幕将显示出来，而且因为用户不再等待游戏，所以应该把 `Wait` 按钮重新启用。

玩家加入房间时将他们的别名设为各自的 JID。对用户而言这并不是非常友好的做法，但这可以简化代码。在真实的游戏系统中，应该更优雅地处理别名。

最后，两位玩家都应该加入房间，裁判员将发送`<game-started>`元素。`on_message()`处理程序需要观察发送到房间的消息中的这些元素。将下面最后一条 `else` 子句添加到 `on_message()` 中的主 `if` 语句中。



可从
Wrox.com
下载源代码

```

} else {
    var body = $(message).children('body').text();
    if (body) {
        var who = Strophe.getResourceFromJid(from);
        var nick_style = 'nick';
        if (who === Toetem.connection.jid) {
            nick_style += ' me';
        }

        $('#messages').append(
            "<div><span class='"
            + nick_style + "'>" +
            Strophe.getBareJidFromJid(who) +
            "</span>&gt; " +
            body + "</div>");
        Toetem.scroll_chat();
    }

    if ($(message).find('delay').length > 0) {
        // skip command processing of old messages
        return true;
    }

    var cmdNode = $(message)
        .find('*[xmlns="' + Toetem.NS_TOELEM + '"]');
    var cmd = null;
    if (cmdNode.length > 0) {
        cmd = cmdNode.get(0).tagName;
    }
    if (cmd === 'game-started') {
        $('#resign').removeAttr('disabled');
    }
}

```

```

    } else if (cmd === 'game-ended') {
        $('#resign').attr('disabled', 'disabled');
        $('#leave').removeAttr('disabled');
    }
}

```

code snippet toetem.js

新代码首先将<body>元素的内容添加到聊天区域中。如果接收到来自裁判员的任何命令，它将采取适当的动作，但它会忽略任何来自聊天历史的延时命令。对于 game-started 命令，它只是启用按钮，而对于 game-ended 命令，它将禁用 Resign 按钮并启用 Leave 按钮。

这里的代码还显示游戏房间的其他房客发送的消息。当然，用户也希望进行通信并能够使用这些新按钮。将下面的事件处理程序添加到客户端的文档准备就绪事件处理程序中。



可从
Wrox.com
下载源代码

```

$('#input').keypress(function (ev) {
    if (ev.which === 13) {
        ev.preventDefault();

        var input = $(this).val();
        $(this).val('');

        Toetem.connection.send(
            $msg({to: Toetem.game, type: 'groupchat'})
                .c('body').t(input));
    }
});

$('#resign').click(function () {
    Toetem.connection.sendIQ(
        $iq({to: Toetem.referee, type: 'set'})
            .c('resign', {xmlns: Toetem.NS_TOETEM}));
});

$('#leave').click(function () {
    Toetem.connection.send(
        $pres({to: Toetem.game + '/' + Toetem.connection.jid,
            type: 'unavailable'}));
    $('#game').hide();
    $('#browser').show();
});

```

code snippet toetem.js

keypress 处理程序基本上与我们在前几章中创建用来让用户输入和发送文本的处理程序相同。单击 Resign 按钮向裁判员发送<resign>命令，这会导致向游戏房间发送<game-ended>命令。Leave 按钮用来退出房间并将客户端退回到浏览器屏幕。

最后还需要 scroll_chat()函数，我们在前面也讨论过它。将这个函数添加到 Toetem 对象中。



可从
Wrox.com
下载源代码

```
scroll_chat: function () {
    var div = $('#messages').get(0);
    div.scrollTop = div.scrollHeight;
}
```

code snippet totem.js

Toetem 客户端的第二个版本已经完成了。试着运行新的裁判员以及几个客户端，并创建、放弃和离开游戏。甚至还可以跟自己聊天。

11.7 实现游戏逻辑

完成 Tic-Tac-Toe 游戏的最后一步是现实游戏规则、玩家移动、显示 Tic-Tac-Toe 游戏板并允许观众观看游戏。

首先，裁判员需要管理每个 Tic-Tac-Toe 游戏的游戏板，我们将建立一个小型库来记录游戏板上以前的移动并确保新的移动是合法的。这个库还判断游戏何时结束(可能是一位玩家赢了，也可以是所有可能的移动都已经完成而没有任何玩家实现同一行中有三个 X 或 O)。

一旦完成游戏板的逻辑，就可以扩展裁判员来使用它，并对从玩家那里接收到的传入移动做出反应。

裁判员还必须向任何加入该房间的观众发送即时的游戏状态。

最后，需要向客户端添加 Tic-Tac-Toe 游戏板的图形显示，并允许玩家通过直接在游戏板上单击鼠标来完成移动。观众能够在游戏过程中加入游戏并观看赛事。

11.7.1 Tic-Tac-Toe 库

Tic-Tac-Toe 库将所有与游戏相关的逻辑抽离出来放入一个地方。裁判员将为每个游戏创建一个游戏板并将移动发送给游戏板。在一次移动之后，它可以弄明白游戏是否结束以便采取适当的动作。在本节中，我们创建一个实现这些功能的 Board 类。

首先，创建一个名为 board.js 的文件，并将 Board 构造器函数添加到该文件中。



可从
Wrox.com
下载源代码

```
Referee.Board = function () {
    this.board = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']];
```

code snippet board.js

注意，Board 对象是在 Referee 对象下面定义的。裁判员可以使用 new Referee.Board() 来创建新的游戏板。

游戏板的内部状态只有一个属性 board。这个属性只是一个三乘三的字符矩阵，字符的值可以是 'X'、'O' 或 ' '。

移动将以行和列的形式发送给裁判员。列是字母，可以是 a、b 或 c，而行是数字，可以是

1、2 或 3。Board 类需要将这些值转换成能够用来索引游戏板矩阵的内部坐标。建立下面的辅助函数 `moveToCoords()`。将该代码添加到构造器函数之后。



可从
Wrox.com
下载源代码

```
Referee.Board.prototype = {
    moveToCoords: function (col, row) {
        var map = {'a': 0, 'b': 1, 'c': 2, '1': 0, '2': 1, '3': 2};
        var coords = {row: null, col: null};
        coords.col = map[col];
        coords.row = map[row];
        return coords;
    }
};
```

code snippet board.js

通过将 JavaScript 类的原型对象设置为一个包含着预期方法的对象，我们可以定义 JavaScript 类方法。当创建 Board 对象时，新对象将从原型中继承这些方法。在本节剩余部分，我们通过更多的方法来增强这个原型。

`moveToCoords()` 函数携带行和列值并创建一个简单的对象，将 `col` 和 `row` 属性设置为转换后的值。`map` 变量定义了一个用户指定坐标到数字索引之间的简单映射，这使得转换极其简单。

还需要另一个便利的辅助函数 `toString()`，它将游戏板的状态转换成一个简短的字符串。将下面的代码添加到原型中。



可从
Wrox.com
下载源代码

```
toString: function () {
    var r, s = '';
    for (r = 0; r < 3; r++) {
        s += this.board[r].join('');
    }
    return s;
}
```

code snippet board.js

为了判断移动是否合法以及游戏是否结束，Board 类必须知道轮到哪个玩家移动。将下面的 `currentSide()` 函数添加到原型中。这个函数计算 X 和 O 的个数以判断下一步轮到谁移动。



可从
Wrox.com
下载源代码

```
currentSide: function () {
    var r, c;
    var x = 0, o = 0;

    for (r = 0; r < 3; r++) {
        for (c = 0; c < 3; c++) {
            if (this.board[r][c] === 'x') {
                x += 1;
            } else if (this.board[r][c] === 'o') {
```

```

        o += 1;
    }
}

if (x === o) {
    return 'x';
}

return 'o';
}

```

code snippet board.js

由于一次移动由 X 或 O 以及目标坐标组成，因此可以轻易地判断该移动是否合法。X 移动只有轮到 X 方移动时才能够进行，这同样适用于 O 移动。此外，已经包含移动的方格不能再次移动。这些规则已经在下面的 `validMove()` 实现中得以体现，应该将该代码添加到原型中。



可从
Wrox.com
下载源代码

```

validMove: function (side, col, row) {
    var curSide = this.currentSide();
    if (side !== curSide) {
        return false;
    }

    var coords = this.moveToCoords(col, row);
    if (this.board[coords.row][coords.col] !== ' ') {
        return false;
    }

    return true;
}

```

code snippet board.js

`move()` 函数调用 `validMove()` 确保移动得到允许。如果该移动合法，那么游戏板矩阵将更新以反映这次新的移动。将 `move()` 函数添加到原型中。



可从
Wrox.com
下载源代码

```

move: function (side, col, row) {
    if (this.validMove(side, col, row)) {
        var coords = this.moveToCoords(col, row);
        this.board[coords.row][coords.col] = side;
    } else {
        throw {
            name: "BoardError",
            message: "Move invalid"
        };
    }
}

```

code snippet board.js

如果移动不合法，那么会出现异常，裁判员将捕获该情况，并向肇事客户端发送错误提示消息。

一旦移动完成，裁判员就必须判断游戏是否结束。建立一个名为 `gameOver()` 的函数来处理这项任务，并将其添加到原型中。



可从
Wrox.com
下载源代码

```
gameOver: function () {
    var r, c;

    // check for row wins
    for (r = 0; r < 3; r++) {
        if (this.board[r][0] === 'x' &&
            this.board[r][1] === 'x' &&
            this.board[r][2] === 'x') {
            return 'x';
        } else if (this.board[r][0] === 'o' &&
                   this.board[r][1] === 'o' &&
                   this.board[r][2] === 'o') {
            return 'o';
        }
    }

    // check for column wins
    for (c = 0; c < 3; c++) {
        if (this.board[0][c] === 'x' &&
            this.board[1][c] === 'x' &&
            this.board[2][c] === 'x') {
            return 'x';
        } else if (this.board[0][c] === 'o' &&
                   this.board[1][c] === 'o' &&
                   this.board[2][c] === 'o') {
            return 'o';
        }
    }

    // check for diagonal wins
    if (this.board[0][0] === 'x' &&
        this.board[1][1] === 'x' &&
        this.board[2][2] === 'x') {
        return 'x';
    } else if (this.board[0][0] === 'o' &&
               this.board[1][1] === 'o' &&
               this.board[2][2] === 'o') {
        return 'o';
    } else if (this.board[0][2] === 'x' &&
               this.board[1][1] === 'x' &&
               this.board[2][0] === 'x') {
        return 'x';
    } else if (this.board[0][2] === 'o' &&
               this.board[1][1] === 'o' &&
               this.board[2][0] === 'o') {
        return 'o';
    }
}
```

```

    }
    // check for a tie
    var tie = true;
    for (r = 0; r < 3; r++) {
        if (this.board[r].indexOf(' ') >= 0) {
            tie = false;
            break;
        }
    }
    if (tie) {
        return "=";
    }
    return false;
}

```

code snippet board.js

这个函数虽长却非常简单。它只是在游戏版矩阵中检查所有的能够取胜 Tic-Tac-Toe 的模式。如果没有探测到这种模式，那么它检查是否所有的方格都已包含移动，这样游戏就平局了。如果游戏结束，那么它将返回 x、o 或=，否则返回 false。

这就是 Tic-Tac-Toe 简单游戏所需的全部逻辑。本章末尾的程序清单 11-8 给出了 board.js 文件的完整代码。

11.7.2 裁判员(版本 3)

借助新的 Board 类，我们很快就可以实现全功能的裁判员程序。大部分工作是添加新的 <move> 命令。

首先，修改 new_game() 函数，初始化游戏的 Board 对象。



可从
Wrox.com
下载源代码

```

new_game: function () {
    return {
        room: null,
        board: new Referee.Board(),
        waiting: 2,
        status: 'starting',
        x_player: null,
        o_player: null,
        winner: null
    };
}

```

code snippet referee.js

接下来，向裁判员的 on_iq() 处理程序中添加新的 switch 语句来处理 <move> 命令。



可以从
Wrox.com
下载源代码

```
case 'move':
    Referee.on_move(iq, id, from, child);
    break;
```

code snippet referee.js

与其他命令类似，这条命令将它的实现委托给另一个函数 `on_move()`。将下面的函数添加到 `Referee` 对象中。



可以从
Wrox.com
下载源代码

```
on_move: function (iq, id, from, elem) {
    var game = Referee.find_game(from);
    var row = elem.attr('row');
    var col = elem.attr('col');

    if (!game) {
        Referee.send_error(iq, 'cancel', 'not-allowed');
    } else if (!row || !col) {
        Referee.send_error(iq, 'modify', 'bad-request');
    } else if (!game || game.status !== 'playing' ||
        (game.board.currentSide() === 'x' &&
        from === game.o_player) ||
        (game.board.currentSide() === 'o' &&
        from === game.x_player)) {
        Referee.send_error(iq, 'wait', 'unexpected-request');
    } else {
        var side = null;
        if (from === game.x_player) {
            side = 'x';
        } else {
            side = 'o';
        }

        try {
            game.board.move(side, col, row);

            Referee.connection.send(
                $iq({to: from, id: id, type: 'result'}));

            Referee.connection.send(
                Referee.muc_msg(game)
                    .c('body').t(
                        Strophe.getBareJidFromJid(from) +
                        ' has placed an ' + side + ' at ' +
                        col + row).up()
                    .c('move', {xmlns: Referee.NS_TOETEM,
                        col: col,
                        row: row}));

            $('#log').prepend("<p>" + Strophe.getBareJidFromJid(from) +
                " moved in game " + game.room + ".</p>");
        }
    }
}
```

```

        // check for end of game
        var winner = game.board.gameOver();
        if (winner) {
            if (winner === 'x') {
                game.winner = game.x_player;
            } else if (winner === 'o') {
                game.winner = game.o_player;
            }

            Referee.end_game(game, 'finished');
        }
    } catch (e) {
        Referee.send_error(iq, 'cancel', 'not-acceptable');
    }
}
}

```

code snippet referee.js

该函数进行一些检查以确保该游戏已经存在可供发送该移动命令的玩家访问，并且该游戏处于正确的状态中允许该玩家移动。如果这些条件均被满足，那么调用游戏板的 move() 函数来进行移动并使用 gameOver() 函数来判断游戏是否结束以及是否有赢家。如果遇到错误，那么向该玩家发送适当的响应。如果一切顺利，那么裁判员将把这次移动广播到整个房间。

最后，裁判员必须向所有加入该房间的观众发送即时的游戏状态信息。这些观众可能在游戏已经进行了几次移动之后才进入该房间，因此他们需要了解当前的游戏板移动以及其他游戏状态。当观众通过向房间发送出席信息来加入游戏时，裁判员会得到通知，因此必须修改 on_presence() 处理程序来发送游戏状态。将下面的 if else 子句添加到 on_presence() 中以 if (game && (game.status === starting) 开头的 if 语句的末尾。



可从
Wrox.com
下载源代码

```

    } else if (game && type !== 'unavailable') {
        // handle observers joining
        var msg = $msg({to: from, type: 'chat'});
        if (game.status === 'starting') {
            msg.c('body').t('Waiting for players..').up()
                .c('game-state', {xmlns: Referee.NS_TOETEM,
                    'phase': game.status,
                    'x-player': game.x_player,
                    'o-player': game.o_player});
        } else if (game.status === 'playing') {
            msg.c('body').t('Game in progress.').up()
                .c('game-state', {xmlns: Referee.NS_TOETEM,
                    'phase': game.status,
                    'x-player': game.x_player,
                    'o-player': game.o_player,
                    'pos': game.board.toString()});
        } else {
            msg.c('body').t('Game over.').up()
        }
    }
}

```

```

        .c('game-state', {xmlns: Referee.NS_TOETEM,
                           'phase': 'finished',
                           'x-player': game.x_player,
                           'o-player': game.o_player,
                           'pos': game.board.toString()});
        if (game.winner) {
            msg.attr({winner: game.winner});
        }
    }

    Referee.connection.send(msg);

    $('#log').prepend("<p>Sent state to observer " + bare_from +
                      " in game " + game.room + ".</p>");
}

```

code snippet referee.js

任务已经完成了，裁判员现在已经准备好担任 Tic-Tac-Toe 游戏裁判了。

11.7.3 Toetem 客户端(版本 3)

最终的 Toetem 客户端需要一个美观的 Tic-Tac-Toe 游戏板来显示正在进行中的游戏并让玩家进行移动。Toetem 还需要侦听裁判员发送到房间的<move>命令，以便让游戏板保持同步，而如果用户正在观看游戏，那么当用户加入时响应游戏状态信息。

我们将使用 HTML5 <canvas> API 来绘制图形板，在第 9 章的 SketchPad 应用程序中也使用了该 API。与 SketchPad 类似，最终的 Toetem 客户端也不能在 Internet Explorer 中运行，这是因为它不支持<canvas>元素。

Tic-Tac-Toe 游戏板由九个被四条直线彼此分隔的方格组成。将下面的 draw_board()函数添加到 Toetem 对象中来绘制游戏板。



可从
Wrox.com
下载源代码

```

draw_board: function () {
    var ctx = $('#board')[0].getContext('2d');

    // clear board
    ctx.fillStyle = '#000';
    ctx.beginPath();
    ctx.fillRect(0, 0, 300, 300);

    // draw grid lines
    ctx.strokeStyle = '#999';
    ctx.lineWidth = 4;

    ctx.beginPath();

    ctx.moveTo(100, 10);
    ctx.lineTo(100, 290);
    ctx.moveTo(200, 10);

```



```

    ctx.lineTo(200, 290);
    ctx.moveTo(10, 100);
    ctx.lineTo(290, 100);
    ctx.moveTo(10, 200);
    ctx.lineTo(290, 200);
    ctx.stroke();
}

}

```

code snippet toetem.js

首先，用黑色填充整个区域。然后采用灰色绘制四条直线以建立九个方格。<canvas> API 调用的自描述性相当好，可以在第 9 章中找到有关它们的更多细节。

当玩家移动时，必须在适当的位置上放置 X 或 O。我们使用两条对角线来绘制 X，而使用圆圈来绘制 O。将下面的 `draw_piece()` 函数添加到 `Toetem` 对象中。



可从
Wrox.com
下载源代码

```

draw_piece: function (piece, x, y) {
    var ctx = $('#board')[0].getContext('2d');

    ctx.strokeStyle = '#fff';

    var center_x = (x * 100) + 50;
    var center_y = (y * 100) + 50;

    ctx.beginPath();

    if (piece === 'x') {
        ctx.moveTo(center_x - 15, center_y - 15);
        ctx.lineTo(center_x + 15, center_y + 15);

        ctx.moveTo(center_x + 15, center_y - 15);
        ctx.lineTo(center_x - 15, center_y + 15);
    } else {
        ctx.arc(center_x, center_y, 15, 0, 2 * Math.PI, true);
    }

    ctx.stroke();
}

```

code snippet toetem.js

计算适当方格的中心位置，然后相对于该位置来绘制正确的图形。

虽然现在已经可以绘制美观的图形，但还需要将其与游戏客户端结合起来。为了绘制初始的游戏板并正确地设置 `watching` 属性，将下面的被突出显示的代码行添加到 `on_message()` 处理程序中，就放在显示游戏区域的代码之后。



可从
Wrox.com
下载源代码

```

Toetem.game = from;
Toetem.watching = false;
$('#messages').empty();

```

```

$( '#messages' ).append( "<div class='system'>" +
    "Joined game #" +
    Strophe.getNodeFromJid( from ) +
    "</div>" );
Toetem.scroll_chat();

$( '#wait' ).removeAttr( 'disabled' );
$( '#browser' ).hide();
$( '#game' ).show();
Toetem.draw_board();
$( '#board-status' ).html( 'Waiting for other player...' );

```

code snippet toetem.js

Watch Game 按钮也需要类似的代码来切换到游戏视图中并绘制游戏板以及加入游戏房间的代码。将下面的代码添加到文档准备就绪事件处理程序中。



可从
Wrox.com
下载源代码

```

$( 'input.watch_button' ).live( 'click', function () {
    // join the game room
    Toetem.game = $( this ).parent().prev().text();
    Toetem.watching = true;

    $( '#browser' ).hide();
    $( '#game' ).show();

    Toetem.draw_board();
    $( '#board-status' ).html( '' );

    Toetem.connection.send(
        $pres( { to: Toetem.game + '/' + Toetem.connection.jid } ) );
});
```

code snippet toetem.js

为了进行新的移动，玩家单击游戏板上的一个空白格子。这个动作会导致向裁判员发送一条适当的<move>命令。没有理由让玩家违反顺序游戏，因此只有轮到玩家时才允许移动。

为了跟踪轮到哪一方移动、玩家使用什么块以及玩家是在观看还是在游戏中，应将下面的新属性添加到 Toetem 对象中。



可从
Wrox.com
下载源代码

```

x_player: null,
o_player: null,
turn: null,
my_side: null,
watching: false
```

code snippet toetem.js

当开始游戏或加入游戏时初始化这些新属性，而当玩家移动时进行维护。

将下面的处理程序添加到文档准备就绪事件处理程序中，让玩家进行移动。



可从
Wrox.com
下载源代码

```
$('#board').click(function (ev) {
    if (Toetem.turn && Toetem.turn === Toetem.my_side) {
        var pos = $(this).position();
        var x = Math.floor((ev.pageX - pos.left) / 100);
        var y = Math.floor((ev.pageY - pos.top) / 100);

        Toetem.connection.sendIQ(
            $iq({to: Toetem.referee, type: 'set'})
                .c('move', {xmlns: Toetem.NS_TOETEM,
                            col: ['a', 'b', 'c'][x],
                            row: y + 1}));
    }
});
```

code snippet toetem.js

如果轮到玩家移动，那么鼠标单击事件的坐标将用来计算玩家单击的游戏板方格。因为鼠标事件的 `pageX` 和 `pageY` 坐标是相对于页面(而不是元素)的，所以必须通过减去游戏板左上角的坐标来转换该坐标。因为每个方格每一条边是 100 像素，所以该代码将最终的坐标除以 100 来获得介于 0~2 之间的整数，指示每个方向上的三个可能的方格。

一旦知道了方格，就向裁判员发送`<move>`命令。注意，此时并不更新屏幕上的游戏板，一旦裁判员向游戏房间广播经过验证的移动之后，才会更新游戏板。

在能够玩真正的 Tic-Tac-Toe 游戏之前，我们还需要完成游戏启动、游戏状态以及移动处理逻辑。裁判员广播游戏开始消息，并移到房间，但直接向观众发送游戏状态，所有这些通知都会到达 `on_message()` 处理程序。将下面被突出显示的修改代码提交到 `on_message()` 内部的命令处理逻辑中。



可从
Wrox.com
下载源代码

```
var cmdNode = $(message)
    .find('*[xmlns="' + Toetem.NS_TOETEM + '"]');
var cmd = null;
var row, col;
if (cmdNode.length > 0) {
    cmd = cmdNode.get(0).tagName;
}
if (cmd === 'game-started') {
    var me = Toetem.connection.jid;
    Toetem.x_player = cmdNode.attr('x-player');
    Toetem.o_player = cmdNode.attr('o-player');
    Toetem.turn = 'x';

    if (Toetem.x_player === me) {
        Toetem.my_side = 'x';
        $('#board-status').html('Your move..');
    } else if (Toetem.o_player === me) {
        Toetem.my_side = 'o';
    }
}
```

```

        $('#board-status').html("Opponent's move..");
    }

    if (!Toetem.watching) {
        $('#resign').removeAttr('disabled');
    } else {
        $('#leave').removeAttr('disabled');
    }
} else if (cmd === 'game-ended') {
    $('#resign').attr('disabled', 'disabled');
    $('#leave').removeAttr('disabled');
    var winner = cmdNode.attr('winner');
    if (winner === Toetem.connection.jid) {
        $('#board-status').html('You won!');
    } else if (winner && !Toetem.watching) {
        $('#board-status').html('You lost!');
    } else if (!Toetem.watching) {
        $('#board-status').html('You tied!');
    }
} else if (cmd === 'move') {
    var map = {'a': 0, 'b': 1, 'c': 2, '1': 0, '2': 1, '3': 2};
    col = cmdNode.attr('col');
    row = cmdNode.attr('row');

    Toetem.draw_piece(Toetem.turn, map[col], map[row]);

    if (Toetem.turn === 'x') {
        Toetem.turn = 'o';
    } else {
        Toetem.turn = 'x';
    }

    if (!Toetem.watching) {
        Toetem.my_turn = Toetem.turn === Toetem.my_side;

        if (Toetem.my_turn) {
            $('#board-status').html("Your move..");
        } else {
            $('#board-status').html("Opponent's move..");
        }
    }
} else if (cmd === 'game-state') {
    Toetem.x_player = cmdNode.attr('x-player');
    Toetem.o_player = cmdNode.attr('o-player');

    var pos = cmdNode.attr('pos');
    var blanks = 0;
    if (pos) {
        var idx = 0;
        for (row = 0; row < 3; row++) {
            for (col = 0; col < 3; col++) {
                if (pos[idx] !== ' ') {

```

```

        Toetem.draw_piece(pos[idx], col, row);
    } else {
        blanks += 1;
    }

    idx += 1;
}
}

if (blanks % 2 === 0) {
    Toetem.turn = 'o';
} else {
    Toetem.turn = 'x';
}
}

```

code snippet toetem.js

game-started 命令现在设置 x_player、o_player、turn 和 my_side 属性并适当地更新游戏板的状态区域。

当接收到<move>命令时，它们被转换成游戏板坐标并用来在游戏板上绘制块。然后反转 turn 属性，并更新游戏板的状态区域。

game-ended 命令更新玩家按钮的状态并向游戏板的状态区域中打印出一条消息告知游戏赢家是谁。

最后，game-state 命令导致前面的移动被绘制到游戏板上并设置游戏的属性。

Toetem 客户端完工了。启动一个裁判员并邀请几位好友一起在线玩 Tic-Tac-Toe 游戏。将新游戏服务发布并观看其他玩家的游戏。

裁判员、Board 类以及客户端的最终代码分别在程序清单 11-7、程序清单 11-8 和程序清单 11-9 中给出。



可从
Wrox.com
下载源代码

程序清单 11-7 referee.js(最终代码)

```

var Referee = {
    connection: null,
    games: {},
    waiting: [],
    presence: {},
    NS_TOETEM: "http://metajack.im/ns/toetem",
    NS_MUC: "http://jabber.org/protocol/muc",
    NS_MUC_USER: "http://jabber.org/protocol/muc#user",
    NS_MUC_OWNER: "http://jabber.org/protocol/muc#owner",
    MUC_SERVICE: 'chat.cactus.local',
}

```

```

is_waiting: function (jid) {
    var bare_jid = Strophe.getBareJidFromJid(jid);

    var i;
    for (i = 0; i < Referee.waiting.length; i++) {
        var wjid = Strophe.getBareJidFromJid(Referee.waiting[i]);
        if (wjid === bare_jid) {
            return true;
        }
    }
    return false;
},

is_playing: function (jid) {
    var bare = Strophe.getBareJidFromJid(jid);

    var found = false;
    $.each(Referee.games, function () {
        if (Strophe.getBareJidFromJid(this.x_player) === jid ||
            Strophe.getBareJidFromJid(this.o_player) === jid) {
            found = true;
            return false;
        }
    });
    return found;
},

remove_waiting: function (jid) {
    var bare_jid = Strophe.getBareJidFromJid(jid);

    var i;
    for (i = 0; i < Referee.waiting.length; i++) {
        var wjid = Strophe.getBareJidFromJid(Referee.waiting[i]);
        if (wjid === bare_jid) {
            break;
        }
    }

    if (i < Referee.waiting.length) {
        Referee.waiting.splice(i, 1);

        Referee.broadcast(function (msg) {
            return msg.c('not-waiting', {xmlns: Referee.NS_TOITEM})
                .c('player', {jid: jid});
        });

        $('#log').prepend("<p>Removed " + bare_jid + " from " +
            "waiting list</p>");
    }
},

send_error: function (iq, etype, ename, app_error) {

```

```

var error = $iq({to: $(iq).attr('from'),
                 id: $(iq).attr('id'),
                 type: 'error'})
.cnode(iq.cloneNode(true)).up()
.c('error', {type: etype})
.c(ename, {xmlns: Strophe.NS.STANZAS}).up();

if (app_error) {
    error.c(app_error, {xmlns: Referee.NS_TOETEM});
}

Referee.connection.send(error);
},

on_presence: function (pres) {
    var from = $(pres).attr('from');
    var bare_from = Strophe.getBareJidFromJid(from);
    var type = $(pres).attr('type');
    var bare_jid = Strophe.getBareJidFromJid(Referee.connection.jid);
    var domain = Strophe.getDomainFromJid(from);

    if (domain === Referee.MUC_SERVICE) {
        // handle room presence
        var room = Strophe.getNodeFromJid(from);
        var player = Strophe.getResourceFromJid(from);
        var game = Referee.games[room];

        // make sure it's a game and player we care about
        if (game &&
            (game.status === 'starting' || game.status === 'playing') &&
            (player === game.x_player || player === game.o_player)) {
            if (game.status === 'starting') {
                if (type !== 'unavailable') {
                    // waiting for one less player; if both are
                    // now present, the game is started
                    game.waiting -= 1;

                    $('#log').prepend("<p>Player " + bare_from +
                                      " arrived to game " + game.room +
                                      ".</p>");

                    if (game.waiting === 0) {
                        Referee.start_game(game);
                    }
                } else {
                    // one of the players left before the game even
                    // started, so abort the game
                    Referee.end_game(game, 'aborted');
                }
            } else {
                // during play, forfeit a player if they leave the room
                if (type === 'unavailable') {
                    if (player === game.x_player) {

```

```

        game.winner = game.o_player;
    } else {
        game.winner = game.x_player;
    }

    Referee.end_game(game, 'finished');
}

}

} else if (game && type !== 'unavailable') {
    // handle observers joining
    var msg = $msg({to: from, type: 'chat'});
    if (game.status === 'starting') {
        msg.c('body').t('Waiting for players..').up()
            .c('game-state', {xmlns: Referee.NS_TOETEM,
                'phase': game.status,
                'x-player': game.x_player,
                'o-player': game.o_player});
    } else if (game.status === 'playing') {
        msg.c('body').t('Game in progress.').up()
            .c('game-state', {xmlns: Referee.NS_TOETEM,
                'phase': game.status,
                'x-player': game.x_player,
                'o-player': game.o_player,
                'pos':game.board.toString()});
    } else {
        msg.c('body').t('Game over.').up()
            .c('game-state', {xmlns: Referee.NS_TOETEM,
                'phase': 'finished',
                'x-player': game.x_player,
                'o-player': game.o_player,
                'pos':game.board.toString()});
        if (game.winner) {
            msg.attr({winner: game.winner});
        }
    }
}

Referee.connection.send(msg);
$('#log').prepend("<p>Sent state to observer " + bare_from +
    " in game " + game.room + ".</p>");
}

} else if ((!type || type === "unavailable") &&
    bare_from !== bare_jid) {
    // handle directed presence from players
    if (type === "unavailable") {
        delete Referee.presence[bare_from];
        // remove from lists
        Referee.remove_waiting(bare_from);
        $('#log').prepend("<p>Unregistered " + bare_from + ".</p>");
    } else {
        Referee.presence[bare_from] = from;
    }
}

```

```

        $('#log').prepend("<p>Registered " + bare_from + ".</p>");

        Referee.send_waiting(from);
        Referee.send_games(from);
    }
}

return true;
},
broadcast: function (func) {
    $.each(Referee.presence, function () {
        var msg = func($msg({to: this}));
        Referee.connection.send(msg);
    });
},
send_waiting: function (jid) {
    var msg = $msg({to: jid})
        .c('waiting', {xmlns: Referee.NS_TOETEM});

    $.each(Referee.waiting, function () {
        msg.c('player', {jid: this}).up();
    });

    Referee.connection.send(msg);
},
on_iq: function (iq) {
    var id = $(iq).attr('id');
    var from = $(iq).attr('from');
    var type = $(iq).attr('type');

    // make sure we know the user's presence first
    if (!Referee.presence[Strophe.getBareJidFromJid(from)]) {
        Referee.send_error(iq, 'auth', 'forbidden');
    } else {
        var child = $(iq).find('*[xmlns="' + Referee.NS_TOETEM +
                               '"]':first');
        if (child.length > 0) {
            if (type === 'get') {
                Referee.send_error(iq, 'cancel', 'bad-request');
                return true;
            } else if (type !== 'set') {
                // ignore IQ-error and IQ-result
                return true;
            }
        }

        switch (child[0].tagName) {
        case 'waiting':
            Referee.on_waiting(id, from, child);
            break;

        case 'stop-waiting':
    }
}

```

```

        Referee.on_stop_waiting(id, from, child);
        break;

    case 'start':
        Referee.on_game_start(iq, id, from, child);
        break;

    case 'resign':
        Referee.on_resign(iq, id, from);
        break;

    case 'move':
        Referee.on_move(iq, id, from, child);
        break;

    default:
        Referee.send_error(iq, 'cancel', 'bad-request');
    }
} else {
    Referee.send_error(iq, 'cancel', 'feature-not-implemented');
}
}

return true;
},

on_waiting: function (id, from, elem) {
    // if they were already waiting, remove them so their resource
    // can be updated
    if (Referee.is_waiting(from)) {
        Referee.remove_waiting(from);
    }

    Referee.waiting.push(from);

    Referee.connection.send($iq({to: from, id: id, type: 'result'}));

    Referee.broadcast(function (msg) {
        return msg.c('waiting', {xmlns: Referee.NS_TOELEM})
            .c('player', {jid: from});
    });

    $('#log').prepend("<p>Added " +
        Strophe.getBareJidFromJid(from) + " to " +
        "waiting list.</p>");
},
on_stop_waiting: function (id, from, elem) {
    if (Referee.is_waiting(from)) {
        Referee.remove_waiting(from);
    }

    Referee.connection.send($iq({to: from, id: id, type: 'result'}));
},

```

```

send_games: function (jid) {
  var msg = $msg({to: jid})
    .c('games', {xmlns: Referee.NS_TOETEM});

  $.each(Referee.games, function (room) {
    msg.c('game', {'x-player': this.x_player,
      'o-player': this.o_player,
      'room': Referee.game_room(room)}).up();
  });

  Referee.connection.send(msg);
},

new_game: function () {
  return {
    room: null,
    board: new Referee.Board(),
    waiting: 2,
    status: 'starting',
    x_player: null,
    o_player: null,
    winner: null
  };
},

on_game_start: function (iq, id, from, elem) {
  var with_jid = elem.attr('with');
  var with_bare = Strophe.getBareJidFromJid(with_jid);

  // check that the players are available
  if (!Referee.is_waiting(with_jid)) {
    Referee.send_error(iq, 'modify', 'item-not-found');
    return;
  }

  if (Referee.is_playing(with_jid) ||
    Referee.is_playing(from)) {
    Referee.send_error(iq, 'cancel', 'not-allowed');
    return;
  }

  Referee.connection.send($iq({to: from, id: id, type: 'result'}));

  // remove players from waiting list
  Referee.remove_waiting(from);
  Referee.remove_waiting(with_jid);

  // create game room and invite players
  Referee.create_game(from, with_jid);
},

create_game: function (player1, player2) {
  // generate a random room name, and make sure it
  // doesn't already exist to our knowledge
}

```

```

var room;
do {
    room = "" + Math.floor(Math.random() * 1000000);
} while (Referee.games[room]);

var room_jid = room + "@" + Referee.MUC_SERVICE + "/Referee";
Referee.connection.addHandler(function (presence) {
    var game;

    if ($(presence).find('status[code="201"]').length > 0) {
        // room was freshly created
        game = Referee.new_game();
        game.room = room;

        // create initial game state with randomized sides
        if (Math.random() < 0.5) {
            game.x_player = player1;
            game.o_player = player2;
            Referee.games[room] = game;
        } else {
            game.x_player = player2;
            game.o_player = player1;
            Referee.games[room] = game;
        }

        // invite players to start the game
        Referee.invite_players(game);

        // notify everyone about the game
        Referee.broadcast(function (msg) {
            return msg.c('games', {xmlns: Referee.NS_TOETEM})
                .c('game', {'x-player': game.x_player,
                            'o-player': game.o_player,
                            'room': Referee.game_room(room)});
        });

        $('#log').prepend("<p>Created game room " + room + ".</p>");
    } else {
        // room was already in use, we need to start over
        Referee.connection.send(
            $pres({to: room_jid, type: 'unavailable'}));
        Referee.create_game(player1, player2);
    }

    return false;
}, null, "presence", null, null, room_jid);

Referee.connection.send(
    $pres({to: room_jid})
        .c("x", {xmlns: Referee.NS_MUC}));
},
invite_players: function (game) {

```

```

    // send room invites
    $.each([game.x_player, game.o_player], function () {
        Referee.connection.send(
            $msg({to: game.room + "@" + Referee.MUC_SERVICE})
                .c('x', {xmlns: Referee.NS_MUC_USER})
                .c('invite', {to: this}));
    });
}

game_room: function (room) {
    return room + "@" + Referee.MUC_SERVICE;
},

muc_msg: function (game) {
    return $msg({to: Referee.game_room(game.room), type: "groupchat"});
},

start_game: function (game) {
    game.status = 'playing';
    Referee.connection.send(
        Referee.muc_msg(game)
            .c('body').t('The match has started.').up()
            .c('game-started', {xmlns: Referee.NS_TOELEM,
                'x-player': game.x_player,
                'o-player': game.o_player}));
    $('#log').prepend("<p>Started game " + game.room + ".</p>");
},

end_game: function (game, status) {
    game.status = status;

    // let room know the result of the game
    var attrs = {xmlns: Referee.NS_TOELEM};
    if (game.winner) {
        attrs.winner = game.winner;
    }

    var msg = "";
    if (game.winner) {
        msg += Strophe.getBareJidFromJid(game.winner) +
            " has won the match."
    } else if (status === 'finished') {
        msg += "The match was tied.";
    } else {
        msg += "The match was aborted.";
    }

    Referee.connection.send(
        Referee.muc_msg(game)
            .c('body').t(msg).up()
            .c('game-ended', attrs));
}

```

```

// delete the game
delete Referee.games[game.room];

// leave the room
Referee.connection.send(
    $pres({to: game.room + "@" + Referee.MUC_SERVICE + "/Referee",
           type: "unavailable"}));
// notify all the players
Referee.broadcast(function (msg) {
    return msg.c('game-over', {xmlns: Referee.NS_TOETEM})
        .c('game', {'x-player': game.x_player,
                    'o-player': game.o_player,
                    'room': Referee.game_room(game.room)}));
});

$('#log').prepend("<p>Finished game " + game.room + ".</p>");
},
find_game: function (player) {
    var game = null;
    $.each(Referee.games, function (r, g) {
        if (g.x_player === player || g.o_player === player) {
            game = g;
            return false;
        }
    });
    return game;
},
on_resign: function (iq, id, from) {
    var game = Referee.find_game(from);
    if (!game || game.status === 'finished' ||
        game.status === 'aborted' ||
        game.status === 'starting' ) {
        Referee.send_error(iq, 'cancel', 'bad-request');
    } else {
        if (from === game.x_player) {
            game.winner = game.o_player;
        } else {
            game.winner = game.x_player;
        }
        Referee.end_game(game, 'finished');
        Referee.connection.send($iq({to: from, id: id, type: 'result'}));
        $('#log').prepend("<p>" + Strophe.getBareJidFromJid(from) +
                           " resigned game " + game.room + ".</p>");
    }
},
on_move: function (iq, id, from, elem) {
    var game = Referee.find_game(from);

```

```

var row = elem.attr('row');
var col = elem.attr('col');

if (!game) {
    Referee.send_error(iq, 'cancel', 'not-allowed');
} else if (!row || !col) {
    Referee.send_error(iq, 'modify', 'bad-request');
} else if (!game || game.status !== 'playing' ||
    (game.board.currentSide() === 'x' &&
     from === game.o_player) ||
    (game.board.currentSide() === 'o' &&
     from === game.x_player)) {
    Referee.send_error(iq, 'wait', 'unexpected-request');
} else {
    var side = null;
    if (from === game.x_player) {
        side = 'x';
    } else {
        side = 'o';
    }
    try {
        game.board.move(side, col, row);

        Referee.connection.send(
            $iq({to: from, id: id, type: 'result'}));

        Referee.connection.send(
            Referee.muc_msg(game)
                .c('body').t(
                    Strophe.getBareJidFromJid(from) +
                    ' has placed an ' + side + ' at ' +
                    col + row).up()
                .c('move', {xmlns: Referee.NS_TOETEM,
                            col: col,
                            row: row}));
        $('#log').prepend("<p>" + Strophe.getBareJidFromJid(from) +
                           " moved in game " + game.room + ".</p>");

        // check for end of game
        var winner = game.board.gameOver();
        if (winner) {
            if (winner === 'x') {
                game.winner = game.x_player;
            } else if (winner === 'o') {
                game.winner = game.o_player;
            }

            Referee.end_game(game, 'finished');
        }
    } catch (e) {
        Referee.send_error(iq, 'cancel', 'not-acceptable');
    }
}

```

```

        }
    }
};

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Connect to XMPP',
        buttons: {
            "Connect": function () {
                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val()
                });
                $('#password').val('');
                $(this).dialog('close');
            }
        }
    });
});

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        "http://bosh.metajack.im:5280/xmpp-httpbind");

    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });
    Referee.connection = conn;
});

$(document).bind('connected', function () {
    var conn = Referee.connection;

    $('#log').prepend("<p>Connected as " + conn.jid + "</p>");

    conn.addHandler(Referee.on_presence, null, "presence");
    conn.addHandler(Referee.on_iq, null, "iq");

    conn.send($pres());
});

$(document).bind('disconnected', function () {
    // nothing here yet
});

```



可从
Wrox.com
下载源代码

程序清单 11-8 board.js

```
Referee.Board = function () {
    this.board = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']];
};

Referee.Board.prototype = {
    validMove: function (side, col, row) {
        var curSide = this.currentSide();
        if (side !== curSide) {
            return false;
        }

        var coords = this.moveToCoords(col, row);
        if (this.board[coords.row][coords.col] !== ' ') {
            return false;
        }

        return true;
    },
    move: function (side, col, row) {
        if (this.validMove(side, col, row)) {
            var coords = this.moveToCoords(col, row);
            this.board[coords.row][coords.col] = side;
        } else {
            throw {
                name: "BoardError",
                message: "Move invalid"
            };
        }
    },
    moveToCoords: function (col, row) {
        var map = {'a': 0, 'b': 1, 'c': 2, '1': 0, '2': 1, '3': 2};
        var coords = {row: null, col: null};
        coords.col = map[col];
        coords.row = map[row];
        return coords;
    },
    currentSide: function () {
        var r, c;
        var x = 0, o = 0;

        for (r = 0; r < 3; r++) {
            for (c = 0; c < 3; c++) {
                if (this.board[r][c] === 'x') {
                    x += 1;
                } else if (this.board[r][c] === 'o') {
                    o += 1;
                }
            }
        }
    }
};
```

```
}

if (x === o) {
    return 'x';
}

return 'o';
},

gameOver: function () {
    var r, c;

    // check for row wins
    for (r = 0; r < 3; r++) {
        if (this.board[r][0] === 'x' &&
            this.board[r][1] === 'x' &&
            this.board[r][2] === 'x') {
            return 'x';
        } else if (this.board[r][0] === 'o' &&
                   this.board[r][1] === 'o' &&
                   this.board[r][2] === 'o') {
            return 'o';
        }
    }

    // check for column wins
    for (c = 0; c < 3; c++) {
        if (this.board[0][c] === 'x' &&
            this.board[1][c] === 'x' &&
            this.board[2][c] === 'x') {
            return 'x';
        } else if (this.board[0][c] === 'o' &&
                   this.board[1][c] === 'o' &&
                   this.board[2][c] === 'o') {
            return 'o';
        }
    }

    // check for diagonal wins
    if (this.board[0][0] === 'x' &&
        this.board[1][1] === 'x' &&
        this.board[2][2] === 'x') {
        return 'x';
    } else if (this.board[0][0] === 'o' &&
               this.board[1][1] === 'o' &&
               this.board[2][2] === 'o') {
        return 'o';
    } else if (this.board[0][2] === 'x' &&
               this.board[1][1] === 'x' &&
               this.board[2][0] === 'x') {
        return 'x';
    } else if (this.board[0][2] === 'o' &&
```

```

        this.board[1][1] === 'o' &&
        this.board[2][0] === 'o') {
            return 'o';
        }

        // check for a tie
        var tie = true;
        for (r = 0; r < 3; r++) {
            if (this.board[r].indexOf(' ') >= 0) {
                tie = false;
                break;
            }
        }

        if (tie) {
            return "=";
        }

        return false;
    },
    toString: function () {
        var r, s = '';

        for (r = 0; r < 3; r++) {
            s += this.board[r].join('');
        }

        return s;
    }
};

```



可从
Wrox.com
下载源代码

程序清单 11-9 toetem.js(最终代码)

```

var Toetem = {
    connection: null,
    referee: null,
    NS_TOETEM: "http://metajack.im/ns/toetem",
    game: null,
    x_player: null,
    o_player: null,
    turn: null,
    my_side: null,
    watching: false,

    on_message: function (message) {
        var from = $(message).attr('from');

        if ($(message).find('waiting').length > 0) {
            $(message).find('waiting > player').each(function () {
                $('#waiting tbody').append(
                    "<tr><td class='jid'>" +

```

```

        $(this).attr('jid') +
        "</td><td>" +
        ($(this).attr('jid') === Toetem.connection.jid ?
        "<input type='button' class='stop_button' " +
        "value='stop waiting'>" :
        "<input type='button' class='start_button' " +
        "value='start game'") +
        "</td></tr>");

    });

} else if ($(message).find('not-waiting').length > 0) {
    $(message).find('not-waiting > player').each(function () {
        var jid = $(this).attr('jid');
        $('#waiting td.jid').each(function () {
            if ($(this).text() === jid) {
                $(this).parent().remove();
                return false;
            }
        });
    });
}

} else if ($(message).find('games').length > 0) {
    $(message).find('games > game').each(function () {
        if ($(this).attr('x-player') !== Toetem.connection.jid &&
            $(this).attr('o-player') !== Toetem.connection.jid) {
            $('#games tbody').append(
                "<tr><td>" +
                $(this).attr('x-player') +
                "</td><td>" +
                $(this).attr('o-player') +
                "</td><td class='jid'>" +
                $(this).attr('room') +
                "</td><td>" +
                "<input type='button' class='watch_button' " +
                "value='watch game'>" +
                "</td></tr>");
        }
    });
}

} else if ($(message).find('game-over').length > 0) {
    $(message).find('game-over > game').each(function () {
        var jid = $(this).attr('room');
        $('#games td.jid').each(function () {
            if ($(this).text() === jid) {
                $(this).parent().remove();
                return false;
            }
        });
    });
}

} else if ($(message)
    .find('x > invite').attr('from') === Toetem.referee) {
    Toetem.game = from;
    Toetem.watching = false;
}

```

```

$( '#messages' ).empty();
$( '#messages' ).append( "<div class='system'>" +
    "Joined game #" +
    Strophe.getNodeFromJid( from ) +
    "</div>" );

Toetem.scroll_chat();

$( '#wait' ).removeAttr( 'disabled' );
$( '#browser' ).hide();
$( '#game' ).show();
Toetem.draw_board();
$( '#board-status' ).html( 'Waiting for other player..');

var nick = Toetem.connection.jid;

Toetem.connection.send(
    $pres( { to: Toetem.game + '/' + nick } )
    .c( 'x', { xmlns: Toetem.NS_MUC } ) );
} else {
    var body = $( message ).children( 'body' ).text();
    if ( body ) {
        var who = Strophe.getResourceFromJid( from );
        var nick_style = 'nick';
        if ( who === Toetem.connection.jid ) {
            nick_style += ' me';
        }

        $( '#messages' ).append(
            "<div>&lt;span class='"
            + nick_style + "'>" +
            Strophe.getBareJidFromJid( who ) +
            "</span>&gt; " +
            body + "</div>" );

        Toetem.scroll_chat();
    }

    if ( $( message ).find( 'delay' ).length > 0 ) {
        // skip command processing of old messages
        return true;
    }

    var cmdNode = $( message )
        .find( '*[xmlns=' + Toetem.NS_TOETEM + ']' );
    var cmd = null;
    var row, col;
    if ( cmdNode.length > 0 ) {
        cmd = cmdNode.get( 0 ).tagName;
    }
    if ( cmd === 'game-started' ) {
        var me = Toetem.connection.jid;
        Toetem.x_player = cmdNode.attr( 'x-player' );
        Toetem.o_player = cmdNode.attr( 'o-player' );
    }
}

```

```

Toetem.turn = 'x';

if (Toetem.x_player === me) {
    Toetem.my_side = 'x';
    $('#board-status').html('Your move..');
} else if (Toetem.o_player === me) {
    Toetem.my_side = 'o';
    $('#board-status').html("Opponent's move..");
}

if (!Toetem.watching) {
    $('#resign').removeAttr('disabled');
} else {
    $('#leave').removeAttr('disabled');
}

} else if (cmd === 'game-ended') {
    $('#resign').attr('disabled', 'disabled');
    $('#leave').removeAttr('disabled');
    var winner = cmdNode.attr('winner');
    if (winner === Toetem.connection.jid) {
        $('#board-status').html('You won!');
    } else if (winner && !Toetem.watching) {
        $('#board-status').html('You lost!');
    } else if (!Toetem.watching) {
        $('#board-status').html('You tied!');
    }
} else if (cmd === 'move') {
    var map = {'a': 0, 'b': 1, 'c': 2, '1': 0, '2': 1, '3': 2};
    var col = cmdNode.attr('col');
    var row = cmdNode.attr('row');

    Toetem.draw_piece(Toetem.turn, map[col], map[row]);

    if (Toetem.turn === 'x') {
        Toetem.turn = 'o';
    } else {
        Toetem.turn = 'x';
    }

    if (!Toetem.watching) {
        Toetem.my_turn = Toetem.turn === Toetem.my_side;

        if (Toetem.my_turn) {
            $('#board-status').html("Your move..");
        } else {
            $('#board-status').html("Opponent's move..");
        }
    }
} else if (cmd === 'game-state') {
    var pos = cmdNode.attr('pos');
    if (pos) {
        var idx = 0;

```

```
        for (row = 0; row < 3; row++) {
            for (col = 0; col < 3; col++) {
                if (pos[idx] !== ' ') {
                    Toetem.draw_piece(pos[idx], col, row);
                }
            }
            idx += 1;
        }
    }
}

return true;
},
scroll_chat: function () {
    var div = $('#messages').get(0);
    div.scrollTop = div.scrollHeight;
},
draw_board: function () {
    var ctx = $('#board')[0].getContext('2d');

    // clear board
    ctx.fillStyle = '#000';
    ctx.beginPath();
    ctx.fillRect(0, 0, 300, 300);

    // draw grid lines
    ctx.strokeStyle = '#999';
    ctx.lineWidth = 4;

    ctx.beginPath();

    ctx.moveTo(100, 10);
    ctx.lineTo(100, 290);
    ctx.moveTo(200, 10);
    ctx.lineTo(200, 290);
    ctx.moveTo(10, 100);
    ctx.lineTo(290, 100);
    ctx.moveTo(10, 200);
    ctx.lineTo(290, 200);

    ctx.stroke();
},
draw_piece: function (piece, x, y) {
    var ctx = $('#board')[0].getContext('2d');

    ctx.strokeStyle = '#fff';
    var center_x = (x * 100) + 50;
    var center_y = (y * 100) + 50;
```

```

        ctx.beginPath();

        if (piece === 'x') {
            ctx.moveTo(center_x - 15, center_y - 15);
            ctx.lineTo(center_x + 15, center_y + 15);

            ctx.moveTo(center_x + 15, center_y - 15);
            ctx.lineTo(center_x - 15, center_y + 15);
        } else {
            ctx.arc(center_x, center_y, 15, 0, 2 * Math.PI, true);
        }

        ctx.stroke();
    }
};

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Connect to XMPP',
        buttons: {
            "Connect": function () {
                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val(),
                    referee: $('#referee').val()
                });

                $('#password').val('');
                $(this).dialog('close');
            }
        }
    });

    $('#disconnect').click(function () {
        $(this).attr('disabled', 'disabled');

        Toetem.connection.disconnect();
    });

    $('#wait').click(function () {
        $(this).attr('disabled', 'disabled');

        Toetem.connection.sendIQ(
            $iq({to: Toetem.referee, type: "set"})
                .c("waiting", {xmlns: Toetem.NS_TOETEM}));
    });

    $('input.stop_button').live('click', function () {
        Toetem.connection.sendIQ(
            $iq({to: Toetem.referee, type: "set"})
                .c('stop-waiting', {xmlns: Toetem.NS_TOETEM}),

```

```

        function () {
            $('#wait').removeAttr('disabled');
        });
    });

    $('input.start_button').live('click', function () {
        Toetem.connection.sendIQ(
            $iq({to: Toetem.referee, type: "set"})
                .c('start', {xmlns: Toetem.NS_TOETEM,
                             "with":$(this).parent().prev().text()});
    });

    $('input.watch_button').live('click', function () {
        // join the game room
        Toetem.game = $(this).parent().prev().text();
        Toetem.watching = true;
        $('#browser').hide();
        $('#game').show();
        Toetem.draw_board();
        $('#board-status').html('');

        Toetem.connection.send(
            $pres({to: Toetem.game + '/' + Toetem.connection.jid}));
    });

    $('#input').keypress(function (ev) {
        if (ev.which === 13) {
            ev.preventDefault();

            var input = $(this).val();
            $(this).val('');

            Toetem.connection.send(
                $msg({to: Toetem.game, type: 'groupchat'})
                    .c('body').t(input));
        }
    });

    $('#resign').click(function () {
        Toetem.connection.sendIQ(
            $iq({to: Toetem.referee, type: 'set'})
                .c('resign', {xmlns: Toetem.NS_TOETEM}));
    });

    $('#leave').click(function () {
        Toetem.connection.send(
            $pres({to: Toetem.game + '/' + Toetem.connection.jid,
                   type: 'unavailable'});
        $('#game').hide();
        $('#browser').show();
    });
}

```

```

    $('#board').click(function (ev) {
        if (Toetem.turn && Toetem.turn === Toetem.my_side) {
            var pos = $(this).position();
            var x = Math.floor((ev.pageX - pos.left) / 100);
            var y = Math.floor((ev.pageY - pos.top) / 100);

            Toetem.connection.sendIQ(
                $iq({to: Toetem.referee, type: 'set'})
                    .c('move', {xmlns: Toetem.NS_TOETEM,
                                col: ['a', 'b', 'c'][x],
                                row: y + 1}));
        }
    });
});

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        "http://bosh.metajack.im:5280/xmpp-httpbind");

    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });
    Toetem.connection = conn;
    Toetem.referee = data.referee;
});

$(document).bind('connected', function () {
    $('#disconnect').removeAttr('disabled');
    $('#wait').removeAttr('disabled');

    Toetem.connection.addHandler(Toetem.on_message, null, "message");
    // tell the referee we're online
    Toetem.connection.send(
        $pres({to: Toetem.referee})
            .c('register', {xmlns: Toetem.NS_TOETEM}));
});

$(document).bind('disconnected', function () {
    Toetem.referee = null;
    Toetem.connection = null;

    $('#waiting tbody').empty();
    $('#games tbody').empty();

    $('#login_dialog').dialog('open');
});

```

11.8 让游戏更有趣

Toetem 是一个简单的采用 XMPP 协议的游戏系统示例。可以试着建立一个更好的、更有趣的版本。您可以：

- 添加对另一款游戏(比如德州扑克或棋类)的支持。
- 添加系统能够处理多种游戏类型的功能。
- 允许玩家同时观看多场游戏。
- 允许玩家同时玩多场游戏。棋类游戏高手很轻松就可以同时与数十个实力稍逊的赛手进行较量。

11.9 小结

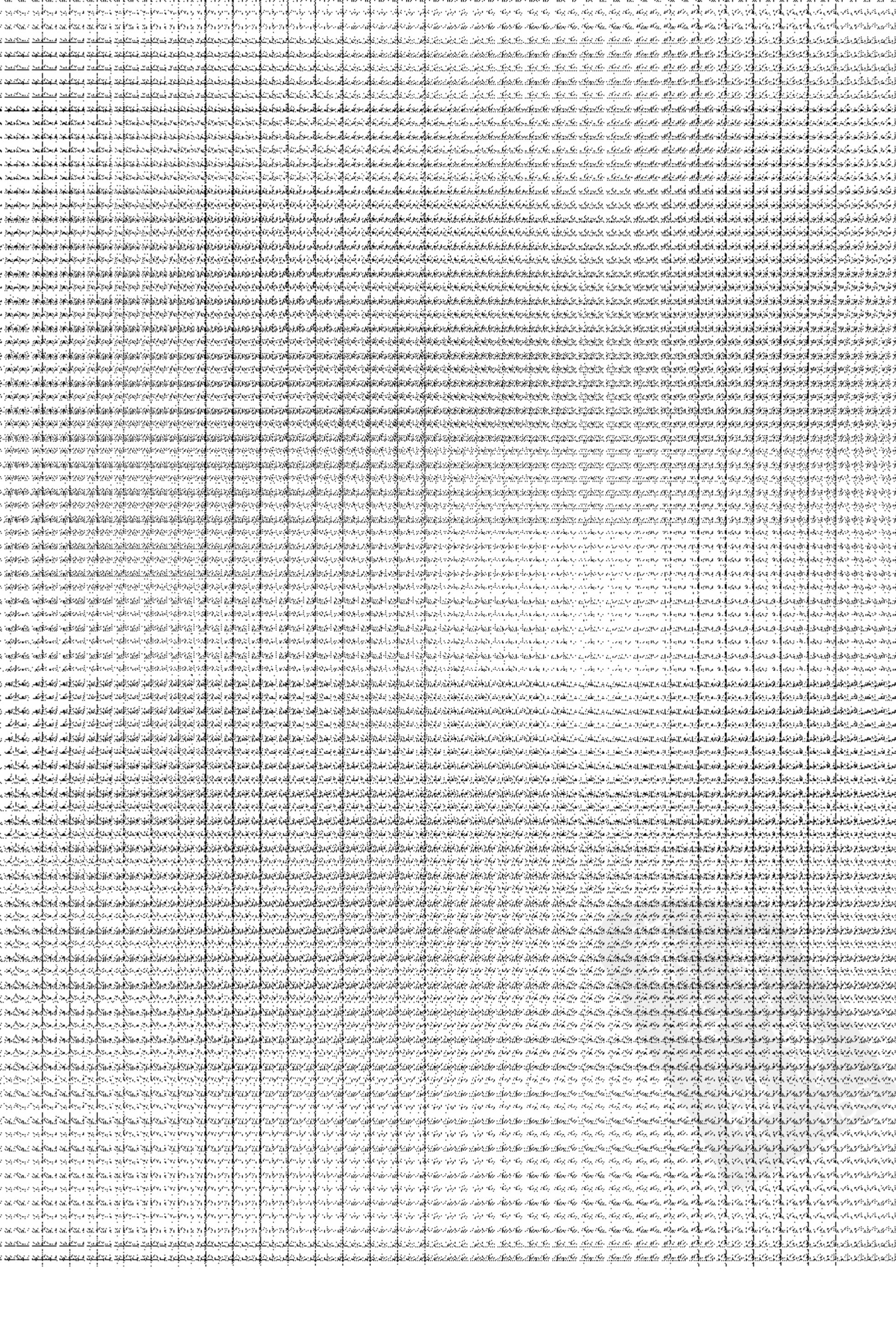
在最后这个应用程序中，我们创建了一个实时的、多玩家 Tic-Tac-Toe 游戏系统。来自世界各地的玩家都可以使用他们的 XMPP 账户连接到该系统，查找其他的玩家，并通过一起玩游戏而获得乐趣。与前面的应用程序不同的是，Toetem 要求同时实现服务逻辑和客户端逻辑。还必须设计一个合适的协议利用多人聊天协议将它有机地组合起来。

在本章中我们学习了如下内容：

- 创建 XMPP 机器人
- 代替用户来管理外部资源
- 设计一个构建在多用户聊天和简单交互之上的游戏协议
- 使用我们在前几章中学到的工具实现一个复杂的应用程序

虽然不借助 XMPP 分析也可以建立类似 Toetem 的系统，但到现在您应该确信那样做会遇到更多的问题。当我们借助 XMPP 的威力，像 Toetem 这样的交互式应用程序就会变得非常易于设计和构建。

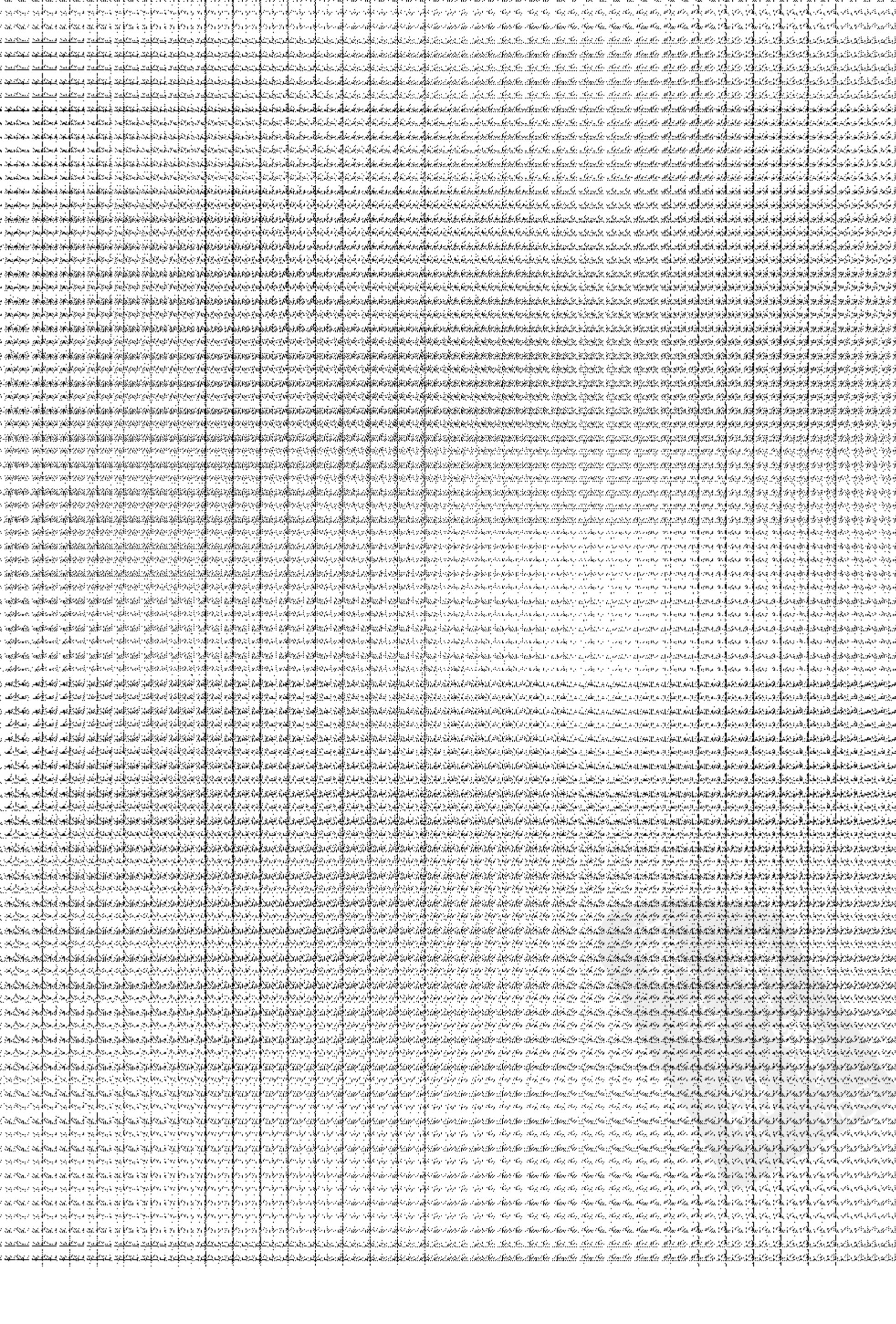
本书的第III部分也是最后的一部分涵盖了几个高级主题，包括 XMPP 会话接入、XMPP 应用程序部署以及 Strophe 插件的编写。



第III部分

高级主题分构

- 第 12 章 加入已有会话：引导 BOSH
- 第 13 章 部署 XMPP 应用程序
- 第 14 章 编写 Strophe 插件



12

第 12 章

加入已有会话：引导 BOSH

本章内容

- BOSH 如何跟踪会话
- 使用 Strophe 接入会话
- 利用会话接入提高性能
- 利用会话接入实现持久化
- 利用会话接入增强安全性

Strophe 较为高级的用法之一是会话接入。当应用程序开始时，我们并不启动一个全新的 XMPP 连接，而是提供了已经建立好的连接，并为 Strophe 提供少量额外信息让它可以附着到这个连接。而这个连接可能是通过服务器端逻辑建立的，也可能是应用程序中前一个页面建立的。

会话接入可以解决与安全、会话持久化以及性能相关的多种问题。大多数时候并不需要会话接入，但当应用程序确实需要它时，它确实非常顺手。

尽管会话接入能够提供更短的应用程序启动时间或增强的安全特性，但它的代价是需要完成更多的工作来准备和管理连接。

知道什么时候使用会话接入与了解如何使用它同样重要。在本章中，我们将介绍常见的用例以及接入已建立会话的技术。

与本书的其他章不同的是，本章不会只使用纯粹的 JavaScript 代码来演示会话接入。本章将使用一些服务器端代码来处理会话创建工作(但在正文中不会完整地对其进行讲解)。

12.1 会话接入

会话接入属于那种作用有限但却不可或缺的功能。它的核心非常简单，我们并不需要启动一个新连接并到服务器进行身份验证，而只是附着到已经建立好的连接上去。但是，这是如何完成的以及它能够用来实现什么功能并不是那么显而易见。

12.1.1 会话技术

XMPP 是一种基于 TCP 的协议，就像 HTTP 一样，通信数据是通过两个端点之间一个已建好的、通常非常可靠的套接字来传送的。回顾第 2 章，XMPP 的 BOSH 扩展在这种双向的有状态协议与 HTTP 协议(单向而且无状态)之间建立了桥接。为了理解会话接入的工作原理，有必要稍微深入一点了解 BOSH 如何实现这两种协议之间的转换。

因为 Web 浏览器不能直接连接到 XMPP 服务器，所以 BOSH 连接管理器响应浏览器的 HTTP 请求，并利用它们为用户管理 XMPP 连接。除了 XMPP 通信所需的套接字之外，每个托管连接均有两个与之相关联的数据：SID 和 RID。

SID 的含义是会话标识符(Session Identifier)。它唯一标识托管的 XMPP 连接，而且通常是一个较长的、无意义的、由字母和数字组成的字符串。虽然它足以标识会话，但它本身并无多大用处。

RID 的含义是请求标识符(Request Identifier)，表示与 BOSH 托管连接相关联的特殊 HTTP 协议请求。在连接建立之前，客户端在它的第一个请求中向连接管理器发送一个随机的 RID。每个后续的请求都会将这个 RID 的值递增 1。

SID 和 RID 一起为与底层 XMPP 连接交互提供了足够的信息。因为 RID 是从一大堆数字中随机生成的，所以实际上如果不知道前面的 RID 数字，那么很难猜出它。此外，连接管理器会拒绝超出当前请求的狭窄窗口值之外的 RID。这样一来，BOSH 托管连接既可以容忍诸如乱序到达这样的小错误，却又可以抵御像连接劫持这样的攻击。

因为这两个标识符足够定位和使用托管的 XMPP 会话，所以如果一个应用程序知道 SID 和 RID，那么它就能够接管或接入底层的会话。我们所要做的就是使用未经修改的 SID 和递增 1 之后的 RID 向 BOSH 连接管理器发送一个请求。

注意，按照这种方式来共享会话非常困难，这是因为两个应用程序(或者同一个应用程序的两个实例)需要在任何时候都知道当前 RID。如果这是一个很容易解决的问题，那么我们一开始就不需要 XMPP 来回传送数据。

当然，接入会话并不会让应用程序访问在接入之前在该连接上上传送的历史数据。为此，通常在会话生命期的最初阶段或其他经过严格定义的、已知的时间点接入会话。

如果应用程序代码知道希望接入的会话的 SID 和 RID，那么可以使用 Strophe 的 attach() 函数来接管该会话。这里并不传入用户名和口令，而是传入 SID 和 RID，但与 connect() 函数类似，它仍然携带一个连接状态回调函数。attach() 函数还携带了会话中使用的 JID，但这不是非常必要的。如果不知道底层会话使用的 JID，那么一旦接入该会话，就可以找到它。

下面的代码演示了 attach() 函数，但在看过前几章的 connect() 函数之后，再看这个函数应该非常熟悉。

```
var connection = new Strophe.Connection(BOSH_URL);
connection.attach(jid, sid, rid, callback);
```

利用 Strophe 来实现会话接入非常容易，但比较棘手的地方在于首先要把 SID 和 RID 传给 Strophe 并在应用程序之外建立连接。

通常情况下，SID 和 RID 是作为 AJAX 调用(用来建立会话)的结果而传回的数据，或者嵌

入到后端 Web 应用程序返回的 HTML 标记中。底层连接通常是通过完成最初的 HTTP 请求的后端代码建立的(一般使用 JavaScript 代码)。

即使了解了会话接入的工作原理，您可能仍然好奇我们为什么需要这样的功能。毕竟，虽然本书中构建的所有应用程序均没有使用它，但都能正常运行。

12.1.2 用例

会话接入技术通常用来解决安全、性能和持久化问题。新的技术用例每天都在出现，因此这里的简短列表无法尽数包含，但这些用法已经展现了会话接入技术出现几年来的一些主要动机。

1. 增强安全性或简化安全功能

BOSH 会话可以进行加密，而且底层的 XMPP 会话通常也同样进行加密。因为 XMPP 使用了 SASL，所以它的身份验证机制往往相当健壮。那么，会话接入技术又是如何在这种情况下进行增强的呢？

在 Web 应用程序中，身份验证最初是由用户输入用户名和口令。服务器分配并返回一个会话标识符，然后所有发送和接收的 HTTP 请求都必须包含这个标识符，它可用来验证客户端的身份，而不需要反复地询问用户凭证。这个用例在启用了 XMPP 的 Web 应用程序中也仍然令人满意。

缓存用户凭证的相同用例与基于令牌的身份验证机制类似。只需要想象一下 SID 和 RID 是令牌，我们就能够明白会话接入为何如此有用。

如果应用程序中启用 XMPP 的部分属于一个更大的网站的一部分，那么若用户不必同时登录网站和 XMPP 应用程序，则用起来要更加方便。在这种情况下，最好是让网站能够为 XMPP 应用程序提供已经嵌入用户凭证的页面。为了避免通过将用户口令放入网页中(因而也位于浏览器的缓存中)将其泄漏，我们可以嵌入会话和请求标识符，并使用它们接入一个已经由服务器预先经过身份验证的正在运行的会话中。在最糟糕的情况下，也只有一个会话会被泄漏。

2. 提高性能

另一个常见的用例是提高应用程序启动性能。通常，需要多次请求和响应才能启动并建立 BOSH 上的 XMPP 连接。若预先在服务器端建立连接，则可消除宝贵的往返(如果启动延迟非常重要)，而且如果实现得当，它实际上可以实现即时连接。

启动时更少的请求也使得 Web 应用程序的 HTTP 基础设施的负载更轻，特别是在大量客户端在连接中断或故障之后重新连接时。想象一下，一万个客户端突然发起连接而且每个客户端还必须发送 6 次或 7 次请求才能建立连接，这会造成很大的流量尖峰。

如果 XMPP 会话是连接到后端 Web 基础结构的本地服务器，那么会话创建性能还会得到极大的提高。每次往返都不需要经过 Internet，它们只须通过局域网传送。假设打开的 Internet 连接的平均 ping 时间为 100ms，那么 6 次请求将耗费多于 0.5s 的时间。而典型的局域网 ping 时间为 10ms，因此启动延迟立即降低了一个数量级。而更好的情况是，会话创建完全可以在 XMPP 服务器内部完成，完全不涉及任何往返延迟和额外的请求。

3. 持久化连接

会话接入还可用来持久化连接，让其跨越多个页面。但这种用例更加复杂，这是因为无法阻止用户在第二个窗口或标签页中打开同一个网页，因此必须小心一次只有一个页面在使用底层会话。

实现会话持久化的方法之一是把 SID 和 RID 存储在 cookie 中，并在加载下一个页面时重新将其接入会话中。这样一来，当用户从一个页面跳到另一个页面时，XMPP 会话就总能够找到他，该会话将多次被分离并重新接入。

在熟悉了会话接入技术的主要用例并了解了它的基本工作原理之后，下面就让我们来看一个它实际运行的示例。

12.2 利用会话接入实现自动登录

会话接入的大部分工作并不是 JavaScript 应用程序代码。本章与本书中其他只使用 JavaScript 的示例不同，它还采用 Python 在 Django 框架之上实现了一个后端 Web 应用程序。同样的工作还可以使用 Ruby on Rails 应用程序或采用 Java 编写的应用程序来完成，但 Python 代码的可读性相当好，即使对于那些不熟悉它的人来说也是如此，用它作为示例非常合适。

首先，必须为这个示例安装几个依赖包。我们需要：

- Django 版本 1.1 或更新的版本，<http://www.djangoproject.com>
- Twisted Python 版本 8.2.0 或更新版本，<http://twistedmatrix.com>
- Punjab 版本 0.13 或更新版本，<http://code.stanziq.com/pubjab>

注意，尽管 Punjab 是一个全功能的 BOSH 连接管理器，但这个示例只用到了它的 BOSH 库例程。没有必要运行 Punjab 服务器，只需要安装该代码，这样 Django 应用程序内部的会话建立代码就能够找到它。

有关特定平台的安装说明，请参见这些包提供的说明。附录 B 也给出了 Punjab 的一些基本说明。一旦安装了软件，就可以创建项目。

建立 Django 项目

使用 Django 内置的 `django-admin.py` 命令来创建一个名为 `attach` 的 Django 项目。将项目放到一个合适目录，并运行该项目：

```
django-admin.py startproject attach
```

该命令应该会创建一个名为 `attach` 的目录，该目录中包含了一个项目骨架。这个目录的内容应该包括文件 `settings.py`、`__init__.py`、`urls.py` 和 `manage.py`。

接下来，必须在这个项目中创建一个名为 `attacher` 的 Django 应用程序。可以使用 `manage.py` 命令来完成这项任务。在 `attach` 目录下面，运行：

```
python manage.py startapp attacher
```

与 `startproject` 类似，这个命令创建了一个名为 `attacher` 的目录，它包含了应用程序的骨架文件。

接下来，必须设置 `urls.py`，将应用程序中的 URL 映射到代码中的特定视图。程序清单 12-1 给出的 `urls.py` 文件建立了一个映射，将根 URL 映射到 `attacher` 应用程序的 `index` 视图。



程序清单 12-1 urls.py

可从
Wrox.com
下载源代码

```
from django.conf.urls.defaults import *
urlpatterns = patterns('',
    (r'^$', 'attach.attacher.views.index'),
    (r'^static/(?P<path>.*)$', 'django.views.static.serve',
    {'document_root':
        '/path/to/attach/media/'}),
)
```

切记要将`/path/to/attach/media/`替换成 `attach` 目录的绝对路径并追加`/media/`。该目录存放着应用程序的静态资源，稍后我们将创建这个目录。

`index` 视图的定义位于 `attacher/views.py` 文件中，程序清单 12-2 给出了该文件的内容。



程序清单 12-2 views.py

可从
Wrox.com
下载源代码

```
from django.http import HttpResponseRedirect
from django.template import Context, loader

from attach.settings import BOSH_SERVICE, JABBERID, PASSWORD
from attach.boshclient import BOSHClient

def index(request):
    bc = BOSHClient(JABBERID, PASSWORD, BOSH_SERVICE)
    bc.startSessionAndAuth()

    t = loader.get_template("attacher/index.html")
    c = Context({
        'jid': bc.jabberid.full(),
        'sid': bc.sid,
        'rid': bc.rid,
    })
    return HttpResponseRedirect(t.render(c))
```

这个视图使用项目设置文件中定义的用户凭据和配置通过 `BOSHClient` 对象创建一个 XMPP 会话。一旦创建，它就利用 `index.html` 模板来呈现一个响应页面。`Context` 对象提供的 `jid`、`sid` 和 `rid` 属性将用来替换模板中的对应数据。当浏览器接收到呈现好的页面时，它将能够访问到已建立连接的真正 JID、SID 和 RID。

`boshclient.py` 文件负责处理会话建立工作，我们将需要从本书的网站下载该文件并将其复制到项目目录中。由于它的源代码太长了，无法在正文中列出，因此这里没有给出它。

为了创建 HTML 模板，在 `attach` 项目目录下面创建一个名为 `templates` 的目录。在 `templates` 目录下面创建 `attacher` 目录，将程序清单 12-3 中的 HTML 代码放到 `index.html` 文件中。



可从
Wrox.com
下载源代码

程序清单 12-3 `index.html`

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

<html>
  <head>
    <title>Strophe Attach Example</title>
    <script src='http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/jquery.js'>
    </script>
    <script src='scripts/strophe.js'></script>
    <script src='scripts/flXHR.js'></script>
    <script src='scripts/strophe.flxhr.js'></script>

    <script type='text/javascript'>
      <!--
      var Attacher = {
        JID: '{{ jid }}',
        SID: '{{ sid }}',
        RID: '{{ rid }}'
      };
      // -->
    </script>

    <link rel='stylesheet' type='text/css' href='attacher.css'>
    <script type='text/javascript' src='attacher.js'></script>
  </head>
  <body>
    <h1>Strophe Attach Example</h1>

    <p>This example shows how to attach to an existing BOSH session with
    Strophe.</p>

    <h2>Log</h2>

    <div id='log'>
    </div>
  </body>
</html>
```

当模板呈现时，`Attacher` 对象的属性将替换成已建立会话的真正的 JID、SID 和 RID 值。该 JavaScript 代码将能够访问这些数据来调用 `Strophe` 的 `attach()` 函数。

必须编辑项目目录中的 `settings.py` 文件，提供有关您自己的首选 XMPP 服务器和 BOSH 服务器的配置信息。将下面的配置行添加到 `settings.py` 文件的末尾，但切记要使用真正的域名

和值。

```
BOSH_SERVICE = 'http://pemberley.lit/xmpp-httpbind'
JABBERID = 'darcy@pemberley.lit/bosh'
PASSWORD = 'lizzy'
```

应该将程序清单 12-4 和程序清单 12-5 中的 CSS 和 HTML 文件放到 attach 项目目录的 media 目录中。需要将 MEDIA_ROOT 设定值设置为这个目录的绝对路径，并在末尾加上斜杠字符。例如，在 Windows 系统中，这个值如下所示：

```
MEDIA_ROOT = 'c:\\projects\\attach\\media\\'
```

在 Mac OS X 或 GNU/Linux 系统中，这可能如下所示：

```
MEDIA_ROOT = '/projects/attach/media/'
```



程序清单 12-4 attacher.css

可从
Wrox.com
下载源代码

```
body {
    font-family: Helvetica;
}

h1, h2 {
    text-align: center;
}

#log {
    background-color: #ccc;
}
```



程序清单 12-5 attacher.js

可从
Wrox.com
下载源代码

```
$(document).ready(function () {
    Attacher.connection = new Strophe.Connection(
        "http://bosh.metajack.im:5280/xmpp-httpbind");

    Attacher.connection.attach(
        Attacher.JID, Attacher.SID, Attacher.RID, null);

    $('#log').append("<div>Session attached!</div>");

    Attacher.connection.sendIQ(
        $iq({to: Strophe.getDomainFromJid(Attacher.JID),
              type: "get"})
        .c('query', {xmlns:
                     'http://jabber.org/protocol/disco#info'}));
    
```

```
function () {
    $('#log').append("<div>Response received " +
                      "from server!</div>");  
});  
});
```

现在应用程序已经完成创建，可以使用 Django 的内置 Web 服务器来测试它。使用下面的命令启动该服务器。

```
python manage.py runserver
```

这条命令启动了一个可以通过 Web 浏览器(<http://127.0.0.1:8000/>)访问的 Web 服务器。如果一切设置正常，那么应该会在网页上看到日志消息，该消息说明它已经连接到会话并与服务器通信。

可以看到，会话接入设置非常复杂，但从 JavaScript 应用程序的视角来看，它极其简单。如果现在还不需要预先建立会话以及会话接入技术，那么只要记得，当有这样的需要时，可以使用该技术即可。

12.3 小结

会话接入是 Strophe 和 XMPP 的一种高级用法，用来从 JavaScript 中接入到预先建立的会话。通过接入以带外方式创建的会话，XMPP 应用程序可以在方便用户的情况下增强安全、提高启动性能，并跨越多次页面加载实现会话持久化。

在本章中我们学习了如下内容：

- 探索会话接入技术的用途
- 学习会话接入的工作原理
- 学习如何将 Strophe 接入已有的会话
- 使用 Django 来构建一个自动登录示例作为后端程序

在第 13 章中，我们将研究一些注意事项、最佳实践以及部署和扩展 XMPP 应用程序的指导方针。

13

第 13 章

部署 XMPP 应用程序

本章内容

- 连接管理器负载均衡
- XMPP 服务器集群
- 分布式组件
- 减少延迟和序列化
- 优化客户端操作

使用 XMPP 开发应用程序只是开始。一旦完成最近的创新工作，那么可能希望得到大规模应用。因为规模非常大，可能存在非常多的用户，所以希望确保应用程序得到适当部署，从而能够尽可能地扩展并高效运行。

我们可以按照多种方式扩展应用程序，既可以横向扩展(扩展到多台计算机上)，也可以纵向扩展(优化应用程序并在性能更高的硬件上运行)。举例来说，许多 XMPP 服务器都具有集群操作模式，可以让多台服务器在单个域下面提供服务。

如果应用程序的用户或流量较少，那么可以跳过本章内容。毕竟，当应用程序变得非常流行时，总是可以回到这里继续阅读。

13.1 横向扩展

应用程序常常会用光单台计算机上的可用资源。对于某些应用程序，这个限制可能是内存，而对于其他一些程序，该限制则可能是处理器速度。尽管向计算机中添加更多资源是一种简单的解决方法，但能够向一台计算机中添加的资源总量终究是有限的。通常更好的做法是通过横向方式进行扩展，也就是把应用程序分布到多台物理计算机上去。

XMPP 应用程序可以利用多种技术来实现该方案。哪种技术合适取决于需要扩展的资源以及特定应用程序的特性。应用程序可以按照以下方式进行横向扩展：

- 在不同的计算机上运行多个连接管理器并在它们之间进行负载均衡
- 组成 XMPP 服务器集群，将连接分摊到多台计算机

- 在多台计算机上运行单个服务器组件
 - 通过使用内部联合将应用程序分摊到 XMPP 服务器网络中
 - 直接采用服务器-服务器协议
- 聪明的程序员肯定还会发现其他方法，但以上是目前的做法。

13.1.1 多个连接管理器

BOSH 连接管理器是一种 HTTP 协议服务，而与大多数 HTTP 协议服务类似的是，我们很容易将其扩展到多台计算机上。如果应用程序拥有成千上万个同时连接的用户，就可能需要将他们的连接分摊到多个连接管理器上。

对于独立版连接管理器来说，非常容易运行多个管理器，但如果使用内置版管理器，就需要服务器支持集群。一旦所有的连接管理器都在运行，那么我们希望它们负载均衡，并提供一个共同的端点来访问它们。

有多种开放源代码的和商业的应用程序以及专门的硬件可用来实现 HTTP 服务的负载均衡和汇聚。大多数大型网站都已经为他们的标准 Web 应用程序部署了这些工具。本章使用的是两款专为这些目的构建的开源工具：nginx 和 Haproxy。

在开始深入了解相关细节之前，对于 BOSH 连接管理器负载均衡还有一件事情需要注意，连接管理器保存的会话状态数很少，即 XMPP 连接及其标识符。当用户首次发送 HTTP 请求时，负载均衡器必须挑选一台服务器来进行响应。对于所有后续的请求，负载均衡器都必须继续选择相同的服务器，否则该连接就可能被中断。但多数负载均衡器都能够通过特殊的 HTTP cookie 来跟踪服务器分配情况，这是因为支持会话的 Web 应用程序通常需要类似的功能。

每一种产品的配置和搭建过程都不同，但下面的几个示例中描述的许多概念均适用于这些产品。

1. nginx

nginx 是一款为高性能和稳定性而设计的轻量级 Web 服务器。它对反向 HTTP 代理以及负载均衡的支持极佳，这使得它特别适合基于 XMPP 的 Web 应用程序。本书描述了独立版连接管理器 Punjab 的负载均衡设置的基本知识。更多有关 Punjab、独立版连接管理器以及如何使用 nginx 作为简单代理的信息，请参见附录 B。

可以到 nginx 的网站 <http://nginx.net> 了解更多详情。

首先，需要在多台计算机上配置和搭建 Punjab，或者在同一台计算机上的不同 IP 地址和端口上运行多个实例。为了进行演示，这里假设 Punjab 在下列计算机上的 5280 端口上运行，服务的路径为/xmpp-httplibbind：

- bosh-1.pemberley.lit
- bosh-2.pemberley.lit
- bosh-3.pemberley.lit

我们将把 nginx 配置成反向代理，并在本地路径/xmpp-httplibbind 下面对这三台 Punjab 服务器进行负载均衡。

在 http 节下面，需要添加一个 upstream 配置节(就像下面的代码那样)来定义 Punjab 服务器集合。

```
upstream punjab {
    ip_hash;
    server bosh-1.pemberley.lit:5280;
    server bosh-2.pemberley.lit:5280;
    server bosh-3.pemberley.lit:5280;
}
```

指令 `ip_hash` 指示 `nginx` 使用用户的 IP 地址散列来挑选服务器，以确保来自同一个 IP 地址的所有请求都会发送到同一台服务器。绝不会意外地将用户发送到一台没有保存他的相应状态的 `Punjab` 实例。读者可能已经注意到这并不是真正的负载均衡，但在许多情况下，这种基于散列的方法也同样有效。

在 `server` 节下面，需要添加 `location` 配置节来设置反向代理。这里并没有使用被代理服务器的主机名，而是使用前面设置的 `upstream` 配置(名为 `punjab`)。

```
location /xmpp-httplib {
    proxy_pass http://punjab;
}
```

现在，`nginx` 将根据用户的 IP 地址的散列值将传入的/`xmpp-httplib` 请求代理给这三台配置好的 `Punjab` 服务器之一。请注意，在添加或删除服务器会改变散列函数时，如果按照这种方式来重新配置服务器，那么用户将可能体验到临时断开连接。

这就是配置的全部工作。当然，`nginx` 还有许多配置选项并没有办法在这里的基本示例中讨论。有关 `nginx` 配置的更多信息，请参见 <http://wiki.nginx.org> 上面的 `nginx` 文档。

2. HAProxy

`HAProxy` 是一种用于 TCP 和 HTTP 服务的高性能、高可用负载均衡器。它能够支持复杂的负载均衡配置，但在这里我们为 `BOSH` 连接管理器使用了一种基本配置。

可以到 `HAProxy` 的网站 <http://haproxy.1wt.eu/> 上访问它。

与 `nginx` 示例一样，这个示例假设同样也有三台 `Punjab` 服务器可用，所有服务器都在侦听端口 5280，可通过/`xmpp-httplib` 来访问 `BOSH` 服务。这些服务器分别叫做 `bosh-1.pemberley.lit`、`bosh-2.pemberley.lit` 和 `bosh-3.pemberley.lit`。

首先，在 `haproxy.cfg` 文件中，需要定义一个新的后端来列出 `Punjab` 服务器集合，如下所示。

```
backend punjab
    mode http
    balance leastconn
    cookie BOSHSRV insert indirect nocache
    server bosh-1 bosh-1.pemberley.lit:5280 cookie bosh-1
    server bosh-2 bosh-2.pemberley.lit:5280 cookie bosh-2
    server bosh-3 bosh-3.pemberley.lit:5280 cookie bosh-3
```

这一节定义一个名为 `punjab` 的、带有三台服务器的 HTTP 后端。负载均衡策略 `leastconn` 指示选择当前连接数量最少的服务器。这里使用了一个 `cookie` 来确保每个客户端持续地使用同

一台 Punjab 服务器来服务多次 HTTP 请求。

为了使用这个新的后端，我们需要向相关的前端配置节中添加一些配置信息。下面的配置行演示了这种配置。

```
frontend www
  # normal config would be here
  option httpclose
  acl bosh path_beg /xmpp-httpbind
  use_backend punjab if bosh
```

`httpclose` 选项是有必要的，这是因为 Haproxy 并不支持 `keep-alive` 模式。如果忽略该选项，那么 Haproxy 只能处理初始请求，而这可能导致一些非预期的行为。`httpclose` 选项通过禁用 `keep-alive` 模式避开了这个问题。

另外两行告诉 Haproxy，当路径以/xmpp-httpbind 开头时将使用 `punjab` 后端。

当 Haproxy 看到/xmpp-httpbind 路径时，它将请求转发到为 `punjab` 后端配置的三台服务器之一。它插入一个名为 `BOSHSRV` 的 cookie，其值设为该服务器的标识符，或者如果该 cookie 已经存在，那么使用它将该请求路由到正确的服务器。因为 Haproxy 使用 cookie 来识别服务器，而不是 IP 散列，所以可以在不影响用户的情况下修改该配置。

可以很容易使用 Haproxy 来建立更复杂的配置。举例来说，可以限制每台 Punjab 服务器只能服务特定数量的连接，设置自定义超时时间以及持续地监测可用性。

有关这里给出的特定配置的信息以及其他更高级用法的信息，请参见 Haproxy 文档。该文档地址为 <http://haproxy.lwt.eu/download/1.3/doc/configuration.txt>。

13.1.2 XMPP 服务器集群

有几款 XMPP 服务器支持集群。在集群配置中，几个 XMPP 服务器实例为单个 XMPP 域提供服务。举例来说，`pemberley.lit` 服务器可以解析成由 10 台集群服务器组成的服务器组，但无论用户选中哪一台服务器，它都将显示为 `pemberley.lit`。

每种 XMPP 服务器都有自己的集群配置需求。在本节中，我们将看到一个演示如何在两台独立的计算机上配置 `ejabberd` 服务器集群的示例。可以通过 <http://www.ejabberd.im> 网站获取和学习有关 `ejabberd` 的更多内容。

首先，需要在两台独立的服务器上安装 `ejabberd`。对于这个示例而言，我们假设这些计算机分别叫做 `xmpp-1.pemberley.lit` 和 `xmpp-2.pemberley.lit`，而且这两台计算机同时提供 `pemberley.lit` 域的 XMPP 服务。基本的配置说明请参见 `ejabberd` 文档。

要配置这两台计算机，必须编辑配置文件 `ejabberd.cfg`。

首先，必须将服务器的主机名修改为正确的域，在这里它是 `pemberley.lit`。在 `xmpp-1.pemberley.lit` 和 `xmpp-2.pemberley.lit` 上都应该是相同的。找到如下所示的配置行，并将其修改为 `pemberley.lit`。

```
{hosts, ["pemberley.lit"]}
```

接下来，可以进行所有其他的普通配置修改，就像通常对单台服务器所做的修改一样。

现在，可以在 xmpp-1.pemberley.lit 上启动 XMPP 服务器。运行如下命令。

```
ejabberdctl start
```

可以通过运行如下命令来检查服务器是否正确启动。

```
ejabberdctl status
```

几秒钟之后它应该提示节点已经启动完毕。

一旦 ejabberd 运行，它就会建立一个 Erlang cookie 文件。需要确保这个 Erlang cookie 文件在两台计算机上是匹配的。ejabberd 的 Erlang cookie 文件名为.erlang.cookie，它与 ejabberd 的其他数据文件位置相同。在类 UNIX 系统(包括 Mac OS X 和 GNU/Linux)上，通常可以在/var/lib/ejabberd 目录中找到该文件。只需要将该文件从 xmpp-1.pemberley.lit 复制到 xmpp-2.pemberley.lit 的相同位置上即可。

在开始配置第二个节点之前，必须首先在该节点上初始化集群的数据库。根据 ejabberd 节点命名方式的不同，这一步可能有点繁琐。在默认情况下，ejabberd 使用短名称，但还可以配置它来使用全限定域名。在这两种情况下，各自的命令稍有不同，这里给出的是默认设置的命令。

要初始化第二个节点的数据库，请运行下面的命令。

```
erl -sname ejabberd@xmpp-2 -mnesia dir ''/var/lib/ejabberd'' \
-mnesia extra_db_nodes "['ejabberd@xmpp-1']" -s mnesia
```

一定要使用正确的引号，这是因为它们非常重要，如果没有正确使用引号，该命令就无法运行。行尾的反斜杠符号只是为了在下一行继续输入命令。如果在一行上输入整条命令，那么可以将字符忽略掉。如果在自己的系统中节点名称以及数据库路径不同，那么还可能必须编辑它们。最后，如果使用的是全限定域名，就需要将-sname 修改为-name。

上面的命令会让您进入 Erlang 命令 shell。可以通过输入下面的命令并按 Enter 键来检验两个节点是否均已经连接到数据库。

```
mnesia:info()
```

应该看到一行以“running db nodes”开头的消息，其中包含两个 ejabberd 节点。

如果一切运行正常，那么现在可以将模式表复制到新节点。输入下面的命令并按 Enter 键。

```
mnesia:change_table_copy_type(schema, node(), disc_copies)
```

现在可以输入如下命令退出 Erlang。

```
q()
```

第二个 ejabberd 节点已经准备好启动了。按照与第一个节点相同的方式启动它。

```
ejabberdctl start
```

如果希望运行两个以上的节点，那么可以重复用来配置第二个节点的指令来配置所有其他节点。有人运行横跨数十台计算机的 ejabberd 集群。

注意，如果正在运行 ejabberd 的内部 BOSH 连接管理器，那么可以运行第 13.1.1 节中的相同比例在这些计算机之间进行负载均衡。如果希望对本地 XMPP 连接进行负载均衡，那么可以依靠 DNS 在适当的 SRV 记录中列出所有可用 XMPP 节点来轮转请求。甚至可以使用 Haproxy 或另一个 TCP 负载均衡器解决方案为来自单个端点的 XMPP 连接提供负载均衡。

13.1.3 扩展组件

在第 1 章中当描述各种不同 XMPP 实体类型时，我们曾经学过一点有关组件的知识。本书中的所有 XMPP 代码均作为客户端来访问服务器，但 XMPP 服务通常编写为服务器组件的形式。有几款 XMPP 服务器(包括 ejabberd)支持负载均衡组件，这为我们扩展应用程序提供了另一种途径。

组件负载均衡背后的基本思想是多个组件以相同的标识连接到服务器。举例来说，如果该服务以 library.pemberley.lit 形式提供，那么每台计算机都将运行该组件并以 library.pemberley.lit 进行连接。

对于 ejabberd，这种负载均衡是通过轮询方式将请求发送到每台计算机来实现的。在集群设置中，ejabberd 首先尝试本地连接的计算机，然后才是远程计算机。这种负载均衡策略并不适用于所有应用程序，但在几种情况下它是有用的。目前虽然还不支持其他策略，但有志向的 Erlang 开发人员很快就会将它们增加进来。

需要查看服务器的文档以了解是否支持组件负载均衡并了解它采用了什么策略。如果可以编写自己的服务以便利用该功能，那么该服务将会很好地扩展。

13.1.4 内部联合

常常没有理由必须让 XMPP 服务的每个用户或每个组件都位于同一台服务器或域中。在这些情况下，我们可以放弃服务器集群，转而使用内部联合来获取扩展性。内部联合服务器是通过服务器-服务器协议进行通信的内部服务器，就像公共 XMPP 服务器彼此的通信方式一样。

我们还可以将内部联合与集群技术联用。举例来说，某项全球 XMPP 服务可能在美国和欧洲都有集群，每个集群通过常规的服务器联合彼此进行通信。

与集群相比，联合的好处之一就是服务器之间没有共享服务器，因此能够取得的扩展性要高得多。能够共存而且可以彼此进行通信的 XMPP 服务器(或电子邮件、HTTP、DNS 服务器)的总数实际上没有限制。

自然的应用程序边界通常是服务器联合边界所在的地方。有些服务可能使用地理位置作为指导，而有些则可能使用 IP 散列。如果该服务在任何位置上都是相同的，那么可以采用任何技术将用户放到最快的或最合适的服务器上去。

13.1.5 成为服务器

并非只有 XMPP 服务器才能使用服务器到服务器协议。最高效的扩展应用程序的方式可能

是完全绕开服务器，让自己的服务直接使用服务器到服务器协议。

服务器-服务器协议要比客户端-服务器版稍微复杂一些，这是由于它依赖 TLS 证书或回拨验证。但一旦建立 XMPP 流，那么来回发送的 XMPP 节就是相同的。

如果 XMPP 服务不太需要诸如用户账户、花名册或其他传统 IM 服务器特性，那么更好的做法是不要服务器。举例来说，XMPP pubsub 服务可能与任何拥有大量的用户的特定服务器无关，因此可以让这类 XMPP pubsub 服务直接作为服务器来寻址。

目前能够简化服务器-服务器协议编程的库较少，但随着越来越多的非 IM 相关的 XMPP 服务的涌现，这种状况正在快速改变。这可能很快成为大多数 XMPP 服务的表现形式，就像 Web 服务已经成为它们自己的 HTTP 服务器一样。

13.2 纵向扩展

横向扩展有时候很难实现。不仅必须处理应用程序的常规关注，而且还需要操心涉及的各种计算机如何分担工作并处理内部通信。如果不打算实现这种扩展性，那么很难改进应用程序。另一方面，购买更快的计算机只是一个资金问题，而优化应用程序代码则是时间和精力问题。

所有应用程序都可以通过性能剖析和计时分析来找出执行较慢的操作。具体说来，XMPP 应用程序有一些常见的地方会出现性能问题，而本节将关注这些问题。主要关注的领域是通信延迟、XML 序列化和 HTML DOM 操作。

13.2.1 减少延迟

所有网络应用程序都必须以某种方式处理延迟问题。一个操作本身的速度可能快如闪电，但如果把请求发送到服务器并接收响应所花的时间却很长，那么用户将体验到恼人的延迟。通常，减少延迟可以为用户可感知的速度带来巨大的提升，因此我们总是应该想方设法让应用程序的延迟尽可能小。

延迟可能来自多个地方，但作为应用程序开发人员，只有几个地方在自己的直接控制之下。XMPP 应用程序中两个最大的延迟来源是连接(以及其他资源)的初始化和建立以及过大的 XMPP 节。根据应用程序的特定需要，这两种延迟的处理方式有多种。

1. 连接流水线及设置

当 XMPP 应用程序启动时，它们需要连接到服务器。这个过程需要花一些时间并涉及多次请求和响应回合才能完成。一旦建立连接，应用程序通常需要执行一组初始化操作，比如获取花名册以及加入聊天室。当使用本地 XMPP 连接时，这些过程并不是太大问题，但如果通过 HTTP 传送，那么情况将大不相同。

在一次发送到连接管理器的 HTTP 请求中可以包含任意多个 XMPP 节，对于响应而言也同样如此。初始化节的性质决定了它们通常必须顺序地发送和接收，这是因为下一个 XMPP 节依赖于前一个 XMPP 节的结果。如果假设每个请求-响应的延迟为 100ms，那么多个初始化节很快就会累积起来，成为显著的延迟并导致用户体验变差。

为了解决这些问题，首先要尽可能地并行处理。如果能够同时启动多个序列，那么完成初

始化所需的 HTTP 往返总数就会减少。如果有三个能够并行地运行的初始化任务，那么完成所有这些任务所需的往返总数等于完成最长的任务所需的往返次数。举例来说，如果一款群聊应用程序在启动时加入多个房间，那么它应该一次性加入所有房间，而不是逐个加入。最终，该应用程序的启动时间将会快得多。

在默认情况下，在调用 `send()` 函数时，`Strophe` 会自动地立即发送新的 XMPP 节。如果应用程序将要在短时间内发送大量 XMPP 节，那么这种行为可能并不是我们想要的，这是因为它可能导致使用过多的、不必要的 HTTP 往返。在代码中可以让这些 XMPP 节形成一个批次，然后一次性发送出去，但如果初始化代码横跨多个模块的话，那么这很难实现。

`Strophe` 提供了两个函数可以帮助我们来优化该应用程序领域：`pause()` 和 `resume()`。当为某个连接调用 `pause()` 之后，`Strophe` 将停止向服务器发送任何数据，直到调用 `resume()`。大量 XMPP 节的传送优化是一个简单的工作，只需要先暂停，执行所有的单个 `send()` 调用，然后恢复即可。最终的结果是向服务器发送一个很大的请求，而不是发送多次小的请求。

我们已经在第 12 章中讲过的会话接入是另一个降低连接延迟的有用工具。因为连接的所有初始设置都发生在原生 XMPP 连接上(通常是在同一个网络上的计算机作为 XMPP 服务器)，所以启动连接只需要一个 HTTP 往返即可。除了让连接和身份验证过程变得更快之外，还可以将常见的初始化代码放在服务器端会话接入服务中，也会让这些操作变得更快。这可以轻易地将数秒的延迟大大缩短。

一旦应用程序已经在运行，那么会话接入技术就没有太大用处了，但还有其他技术也同样可用于应用程序会话的其他几个阶段。举例来说，游戏客户端可能需要向某项中心服务发送多次请求，加入一个房间并查询初始状态。组合使用 `pause()` 和 `resume()` 以及并行化技术可以让程序执行速度尽可能快。

当然，所有这些延迟减少措施都可以按照各种方式进行组合。为特定应用程序选择最为高效和合适的措施。Amazon 和 Google 在减少页面请求延迟所带来的好处方面进行了大量的研究，发现它们能够显著地影响到它们的盈利。用户肯定会喜欢应用程序灵敏的响应能力。

2. 处理大型 XMPP 节

XMPP 协议是为交换小型数据片段而设计的，因此大型 XMPP 节可能会导致一些问题，这一点也不奇怪。在大部分情况下，很容易避开大型 XMPP 节，但即使不能避开，也有几款工具能够帮助缓解它们带来的问题。

为了理解为什么大型 XMPP 节会带来问题，请记住每个 XMPP 会话均由两个 XMPP 流组成。在一个流上面一次只能发送一个 XMPP 节，而且每个 XMPP 节都要花一点时间才能从源传送到目的地。后面的任何需要发送的节都必须等待它们前面的节传送完毕。

通常，这些 XMPP 节都很小，因此每节只需要等待非常短的时间就会被真正发送出去。但在发送大型 XMPP 节时，传送时间可能相当长，而在此期间，没有办法在相同方向上发送任何其他的 XMPP 节。

假设正在通过 XMPP 玩一种快速的棋类游戏，在游戏进行当中，就在您移动之前，某一位联系人发送了一个巨大的头像文件。您的移动会立即被发送给服务器，而且对家立即响应，但客户端现在必须等待直到这个头像文件完全传送完毕，您才能看到对家的移动。与此同时，您的移动计时器仍然在计算着宝贵的时间。

处理大型 XMPP 节的方法主要有两种：将它们切割成较小的 XMPP 节以及在带外处理。

XMPP 提供了多种方法将大量数据切割成多份较小的 XMPP 节。其中一种方法是 In-Band Bytestreams(XEP-0047)，它将大的有效载荷划分成较小的片段，并通过一系列<iq>或<message>节来发送它们。一旦接收到该有效载荷的所有 XMPP 节，就可以组合出原始数据并进行处理。与使用大型数据块发送数据不同的是，我们将其划分为多个 XMPP 节进行传送，这样就留下一些缝隙可供发送其他 XMPP 节，从而改善了整体的延迟问题。In-Band Bytestreams 可以很好地处理诸如图片和二进制数据这类的数据。

有些 XMPP 节是由长长的项列表或结果列表组成。举例来说，发给某个繁忙的 pubsub 系统的服务发现请求可能返回成千上万的可用节点，从而形成一个硕大的、笨重的 XMPP 节。对于这些类型的 XMPP 节，通常更好的做法是分页返回结果集。Results Set Management(XEP-0059) 可用来处理任意数据的分页返回，而 Roster Versioning(XEP-0237)专门为花名册提供了一个类似的解决方案。这些解决方案并非总是构建到每个扩展中，但显然我们可以将其添加到自己的程序中。随着时间的推移，我相信 XMPP 社区将把一个类似这两个方法的解决方案应用到所有可能产生的大型 XMPP 节的扩展中。

尽管 XMPP 可能不是传送大型信息块的最佳协议，但还有大量的其他协议可以轻易地完成这个工作。我们的应用程序可以使用这些协议在 XMPP 连接之外传送大型的带外数据(OOB)。

对于基于 Web 的 XMPP 应用程序而言，最常见的 OOB 技术是使用普通的 HTTP 请求。举例来说，并不通过 XMPP 来传送头像或图片文件，而是可以通过 XMPP 传送该数据的 URL，并通过 HTTP 协议传送真正的数据。XMPP 连接将保持必要的低延迟特性，而大量的数据则可以通过一种接近最优的传输机制进行发送。同时，浏览器缓存许多请求，因此，可以应答对于同一数据的多次请求，而不用浪费带宽来传送重复的数据。

Jingle 扩展(XEP-0166、XEP-0234、XEP-0247 以及其他)为 OOB 连接的协商、建立和通信提供了一个非常灵活的框架。这项技术最初是 Google 为语音和视频聊天而开发的，但已经经过 XMPP 社区的标准化和扩展，能够处理各种用例。

如果有足够的中间服务，那么 Jingle 甚至能够穿越 NAT 和防火墙来协商和建立连接。如果没有 OOB 通道可用，那么 Jingle 还能够协商带内通道。尽管 Jingle 是为多媒体传输协商和管理需要而建立的，但它很快就成为 XMPP 最重要的工具之一，而且已经有扩展使用 Jingle 来实现任何文件传送以及安全的端到端通信。

13.2.2 尽量减少 XML 序列化

应用程序采用便利的数据结构来处理 XMPP 节，而当准备传送时，这些数据结构将被序列化成 XML 字节并通过网络发送出去。繁忙的应用程序将序列化大量 XML 数据，但这通常并不是一种开销很大的操作，但它可能成为性能方面的关键因素。

除了自行优化 XML 序列化例程之外，改进性能的唯一方式就是减少需要完成的序列化操作数量。可以组织应用程序尽可能减少序列化操作数量，但我们首先必须知道序列化操作位于哪些地方。

每个 XMPP 连接都涉及至少两个序列化步骤。一个实体必须序列化 XML 才能发送给另一方，而接收者必须将它发回的任何响应序列化。

对于不同服务器上的两个客户端，总共有高达 6 个序列化步骤。首先，Darcy 必须将他的

XMPP 节序列化并将它们发送到 pemberley.lit 服务器。他的服务器可能需要分析这些节并将它们序列化并发送到 longbourn.lit 服务器。longbourn.lit 服务器可能需要序列化 XMPP 节以传送给 Elizabeth 的客户端。最后, Elizabeth 的响应将沿着反方向传送。如果两个客户端都使用 HTTP 连接管理器, 那么可能会发生更多的序列化操作。

有些高度优化的 XMPP 服务器能够在不完全分析 XMPP 节的情况下路由它们(因而避免重新序列化), 而连接管理器也可以使用类似的技术来避免不必要的序列化。如果序列化变成应用程序的瓶颈, 那么可以考虑使用这些已有的、经过优化的、实现中的一种。

在某些场合中, 我们自己的应用程序也可以避免序列化。举例来说, 就像一台经过优化的服务器一样, 如果访问原始 XML 字节而且只需要发送 XMPP 节的副本, 那么可以传送相同的原始字节。但大多数有用的应用程序通常都不会重新发送复制的 XMPP 节。

但在一个实例中序列化更成问题, 这就是组件。XMPP 服务器组件通过 XMPP 流进行通信, 就像客户端一样, 而且因为它们通常既是通信的终点又是通信的起点, 它们不能简单地路由预先序列化的 XMPP 节。组件必须处理分析和序列化工作。

缓解这种序列化的唯一办法是在实体之间传送表示 XMPP 节的内部数据结构, 但因为这些实体通常是采用不同的编程语言、由不同的开发人员或小组编写的, 这并不总是可行的解决方案。

服务器插件并没有这类序列化问题, 这是因为它们共享数据表示(而且通常共享相同的内存或进程), 而且不需要序列化。此外, 因为它们属于服务器的一部分, 通信通常是通过函数调用完成的, 而不是 TCP 套接字连接。

如果有一个组件遇到了 XML 序列化瓶颈, 那么将其转换成一个服务器插件可能是最可行的选择。

不管什么情况, 理解序列化发生的地点以及在哪些地方并非完全需要是找出如何优化应用程序这部分的问题的第一步。一旦知道额外的序列化步骤位于何处, 那么尽量减少应用上面的技术将成为一件相当简单的事情。

13.2.3 优化 DOM 操作

动态 Web 应用程序必须完成大量的 DOM 操作。当处理新数据或新的用户交互时, 必须改变应用程序的用户界面。那些操纵 DOM 的操作通常是开销最大的操作。

我们可以采用一些技术来减少或优化浏览器 DOM 的访问和操纵: 批处理操作、克隆以及限制范围。

1. 批量操作

如果曾经编写过图形处理代码, 那么读者可能知道双缓冲技术。双缓冲图形应用程序并不是直接在屏幕上绘制, 而是在一个屏幕外的缓冲区上绘制。一旦绘制完成, 将把这个缓冲区与屏幕缓冲区对换。最终实现无闪屏的帧修改。

批量 DOM 操作与双缓冲技术非常相似。我们并不是一次操作一个 DOM 元素, 而是在屏幕外构建 DOM 树的一个更新部分, 然后将整个部分一次性插入。这可以获得性能提升, 这是因为在完成最终的结构之前, 浏览器不必呈现任何内容。

另一项技术(也广泛地应用到计算机图形学中)是定期地更新用户界面。人类无法察觉非常细小的时间片, 任何能够以足够快的速度完成的操作都会被认为是瞬即完成的。我们可以将一

小片时间内的多次中间更新省去，然后在最后一次性呈现它们。用户无法看出两者之间的差别，但计算机必须完成的屏幕更新(开销较大)要少很多。

下面的示例对于演示这些技术可能有所帮助。第 6 章的 Gab 应用程序在屏幕上显示的花名册由一个列表组成，每个列表项本身又由多个元素组成。当把联系人添加到花名册或从中删除联系人时，为了高效地操纵 DOM，我们一次性操作整个项(而不是逐个地进行)。但是，有可能有多次花名册更新几乎在同时发生。如果 Gab 能够将花名册变化累积到一个批次中，然后每 1/10 秒对这些变化进行处理，那么效率要高很多。用户不会注意到应用程序在响应性能上的任何变化，但浏览器每秒最多更新 UI 十次，即使每秒接收到成百上千次更新也同样如此。

2. 克隆而不是创建

与重新创建一个元素相比，浏览器能够以快得多的速度克隆一个现有的元素。通过克隆然后修改模板元素，我们能够优化新 UI 对象的插入操作。还可以使用克隆技术来简化批量操作。

如果应用程序需要插入许多相似类型的元素，它们之间只有少数不同之处，那么可以通过使用模板元素获取一定的效率提升。可以在 HTML 代码中包含一个模板元素，或者在应用程序启动时创建一个。当需要插入类似元素时，我们克隆该元素，进行必要的修改，然后插入克隆的元素。jQuery 提供了 `clone()` 函数来简化该操作。

还可以将克隆技术与批量更新联合起来使用。首先，将接收修改信息的 UI 部分克隆出来。接下来，对这个克隆的元素执行所有更新。当操作完成后，可以将原始元素替换成克隆的元素，一次性展现全部 DOM 操作。

3. 限制操作的作用域

许多 DOM 操作是针对元素集合，而且在该操作完成之前，必须枚举该元素集合。在使用 jQuery 或类似的库时，通过仔细选取选择器，可以限制目标元素的搜索范围。这带来的变化相当大，代码只需要遍历一个较小的元素集合就可以找到想要的那些元素，而不需要遍历整个文档。

限制搜索范围的最简单方法是使用元素标识符作为选择器的第一部分。举例来说，如果希望操作所有带有 CSS 类 `roster-item` 的元素，而这些元素刚好都是一个带有 `roster id` 的元素的子节点，那么可以使用选择器 `#roster .roster-item`，而不是更低效的 `.roster-item`。浏览器只需要搜索 `roster` 元素的所有子节点，而不是整个文档的所有元素。对于大型文档而言，这可以取得显著的速度提升。

13.3 小结

虽然网络让我们感觉世界非常小，但它却可能存在数以百万计的潜在用户。当应用程序需要服务大量的用户群时，我们必须考虑可扩展性。类似地，如果应用程序属于严重的资源密集型程序，那么在部署时通常需要特别注意优化和扩展性。

在本章中，我们学习了如何使用如下的技术来解决这类部署问题：

- 横向和纵向扩展应用程序
- 分散 HTTP 连接管理器

- XMPP 服务器集群
- 服务器组件负载均衡
- 内部联合服务器
- 采用服务器-服务器协议
- 尽可能减少初始化延迟
- 处理大型 XMPP 节
- 减少 XML 序列化
- 优化浏览器中的 DOM 操作

在第 14 章中，我们将讲解开发人员如何轻易地扩展 Strophe 库以添加新功能。



第 14 章

编写 Strophe 插件

本章内容

- 使用 Strophe 插件
- 创建插件
- 向 Strophe 中添加命名空间
- 构建一个简单的花名册插件

本书中的应用程序全部采用 Strophe 的原始函数编写来构建和发送 XMPP 节。按照这种方式开发代码能够让我们了解 XMPP 的各种扩展的工作原理。但在实际的应用程序中，最好开发一些抽象过程来减少某些繁琐的工作。Strophe 允许用户构建和加载插件来扩展它的功能，这样开发人员就能够构建这类抽象过程并提供给应用程序使用。

作为示例，在第 9 章中我们曾构建了应用程序 SketchCast，它将绘制事件广播到 pubsub 节点让订阅者查看。应用程序的代码建立订阅 XMPP 节、节点创建节以及配置节来完成它的任务。假如 Strophe 有一个 pubsub 插件替我们实现这个逻辑，那么使用这个插件的代码就将如下所示。

```
SketchCast.connection.pubsub.create(  
    SketchCast.node, SketchCast.created, SketchCast.create_error);  
  
SketchCast.connection.pubsub.configure(  
    SketchCast.node, {"max_items": 20},  
    SketchCast.configured, SketchCast.configure_error);
```

这个版本的代码依然正确，但要简短得多，而且清晰得多。如何构建操作所需的各种 XMPP 节的详细信息被隐藏到插件接口的后面，从而让我们能够专心思考自己的应用程序逻辑而不是协议的细枝末节。

Strophe 的网站上已经有几个 Strophe 插件可供使用，而且它的用户一直在不停地创建更多的插件。在启动下一个 XMPP 项目之前，或许可以看看有哪些可用的插件可满足应用程序的需求。用户还可以向社区共享插件。

插件是创建运行速度更快的代码并充分利用他人工作成果的极佳方式。它们还能够让我们

将自己的代码模块化，将应用程序逻辑与底层的协议语义分离。在本章中，我们将了解如何使用现有的插件以及创建自己的插件。

14.1 使用插件

使用 Strophe 插件非常简单：加载插件代码，然后通过 `Strophe.Connection` 实例访问新功能。每个插件在加载时自动地将自己安装到 Strophe 库中，然后，每次建立新连接时，该插件将作为该连接的一个属性以提供给代码访问。

14.1.1 加载插件

Strophe 插件必须在 Strophe 库之后加载。下面的 HTML 代码在 Strophe 主代码之后加载 Strophe pubsub 插件(包含在 Strophe 库中)。

```
<script src='strophe.js'></script>
<script src='strophe.pubsub.js'></script>
```

按照约定，插件被命名为 `strophe.plugin-name.js`。插件一旦载入就会自动地将自己注入到 Strophe 库中(在第 14.2 节中我们将看到这是如何实现的)。如果熟悉 jQuery 插件，那么这应该看起来非常熟悉。

14.1.2 访问插件功能

一旦插件载入，它们就成为 `Strophe.Connection` 对象实例的属性可供使用。如果从 `strophe.myplugin.js` 文件中加载插件，就可以通过连接对象的 `myplugin` 属性来访问该插件的接口。如果像前面的示例那样加载 Strophe pubsub 插件，那么可以像下面这样访问它。

```
var connection = new Strophe.Connection(BOSH_URL);
connection.connect(. . .);
. . .
connection.pubsub.subscribe("pubsub.pemberly.lit", "latest_books",
    function (iq) { . . . },
    function (iq) { . . . });
```

也可以按照相同的方式来访问其他 pubsub 插件功能。

除了提供新函数之外，插件还可以向 Strophe 中添加新的命名空间。我们可以随 `Strophe.NS` 对象中的那些内置命名空间一道来访问这些命名空间。例如，Strophe pubsub 插件向 Strophe 中添加了许多与 pubsub 有关的命名空间，下面就是其中一些。

- `Strophe.NS.PUBSUB`，扩展到 `http://jabber.org/protocol/pubsub`
- `Strophe.NS.PUBSUB_OWNER`，扩展到 `http://jabber.org/protocol/`
- `pubsub#owner`
- `Strophe.NS.PUBSUB_EVENT`，扩展到 `http://jabber.org/protocol/`
- `pubsub#event`

如果由于某种原因该插件没有直接提供我们想要的功能，那么这些命名空间使得我们可以非常方便地自行构建 pubsub 节。它们还减少了输入以及规范搜索工作。

每个插件都提供了不同的函数，但均可以按照相同的方式来访问这些函数。如果希望学习有关某个特定插件的知识以及它提供的操作，那么请查阅它的文档。即使该插件文档不全或者根本没有提供文档，那么通过本书学到的工具应该能够很容易理解最复杂的插件。毕竟，大多数插件只是为常见的协议操作提供便利的包装器。

现在我们已经了解了如何加载和使用插件，下面请继续阅读，学习如何构建自己的插件。

14.2 构建插件

插件的创建过程如下：建立定义插件功能的原型对象，然后将该原型注册到 Strophe 库。当建立 Strophe 新连接时，它会根据插件原型建立一个插件对象并调用该对象的 `init()` 函数来初始化该插件。在初始化之后，应用程序代码就可以使用该插件。

`init()` 函数还有另一个作用。大多数插件需要通过连接发送数据以及设置 XMPP 节处理程序，而对于这些任务，插件需要访问 `Strophe.Connection` 对象。Strophe 将该连接对象作为 `init()` 的第一个参数传入，然后该插件就能够将这个引用保存起来留作后用。

大多数插件常用的另一个函数是 `statusChanged()`。当连接状态改变时，Strophe 调用每个插件的 `statusChanged()` 函数。这与我们在前几章中多次看到的连接回调完全一样。这可以让插件响应像 CONNECTED 和 DISCONNECTED 这样的事件。这个函数是可选的，因为有些插件只需要响应直接调用。

程序清单 14-1 中的示例插件是我们能够做出的最小的 Strophe 插件。除了前面讨论的 `init()` 函数之外，它还提供了改变出席状态的 `online()` 和 `offline()` 函数。



程序清单 14-1 `strophe.simple.js`

可以从
Wrox.com
下载源代码

```
Strophe.addConnectionPlugin('simple', {
    init: function (connection) {
        this.connection = connection;
    },
    online: function () {
        this.connection.send($pres());
    },
    offline: function () {
        this.connection.send($pres({type: "unavailable"}));
    }
});
```

通过 `addConnectionPlugin()` 函数可以完成 Strophe 新插件的注册工作。这个函数携带插件名称和插件原型作为实参。这些插件被称为连接插件，因为它们增强了 `Strophe.Connection` 对象。

在未来，Strophe 可能支持用来增强该库其他部分的其他插件类型。

最后，Strophe 插件可以使用`.addNamespace()`来增强`Strophe.NS`对象中可用的命名空间。下面的代码向 Strophe 中添加服务发现命名空间。这个简单的示例插件没有新的命名空间，但更复杂的插件通常需要增强命名空间以方便插件用户的使用。

```
Strophe.addNamespace('DISCO_INFO', 'http://jabber.org/protocol/disco#info');
Strophe.addNamespace('DISCO_ITEMS', 'http://jabber.org/protocol/disco#items');
```

前面示例插件的用户可以编写`connection.simple.online()`来发送可访问出席信息，编写`connection.simple.offline()`来发送不可访问出席信息而不是手工构建 XMPP 节。就这个具体情况而言，这并不能节省多少工作量，但我们很快就会看到，更复杂的插件可以让 XMPP 编程任务变得简单得多。

14.3 创建花名册插件

为了真正体验插件能够做什么，我们将构建一个具有某种不平凡功能的插件。在本节中，我们将开发一个花名册管理插件，它将常见的花名册操作抽离出来并将花名册数据放进 JavaScript 友好的数据结构中提供访问。

管理花名册涉及几个基本操作。首先，需要查询花名册并将其保存起来。然后，需要某种方式来添加、编辑和删除花名册项。因为其他连接资源可能也会对花名册进行修改，所以还需要侦听这些修改并相应地更新花名册。最后，还希望当联系人发送出席信息时能够让花名册保持更新。

14.3.1 保存联系人

在开始花名册修改操作之前，应该首先从如何将花名册状态保存到插件中开始。这将是该代码不同部分之间的主要交互点。

假设我们已经创建了一个应用程序，它在用户界面中显示用户的花名册。为此，我们的代码应该能够为花名册更新操作查询花名册并添加处理程序。当应用程序的另一部分需要花名册数据时，可以再次查询花名册(这样做的效率不高)，或者让 UI 的一部分与另一部分通信，而这些将功能紧紧耦合在一起。这两种方法都不好，最好的做法是将花名册状态解释放在一个地方，而应用程序的各个部分能够与之交互。

下面是一个采用 JavaScript 字面值表示的花名册样例，我们的插件中的花名册状态也将以此结构作为基础。

```
contacts = {
  "darcy@pemberley.lit": {
    name: "Darcy",
    resources: {
      "library": {
        show: "away",
        status: "reading"
      }
    }
  }
}
```

```

        }
    },
    subscription: "both",
    ask: "",
    groups: ["Family"]
},
"bingley@netherfield.lit": {
    name: "Charles",
    resources: {},
    subscription: "both",
    ask: "",
    groups: ["Friends"]
}
);

```

这个花名册样例包含两个联系人(Darcy 和 Bingley)，他们分别处于在线和离线状态。resources 属性指出联系人当前正在使用哪一个资源在线以及它们的适当的元数据，因为 Darcy 至少有一个资源，所以他被认为是在线的，而因为 Bingley 没有资源，所以他被认为是离线的。

联系人的 name、subscription 和 ask 属性以及资源的 status 属性均以它们所表示的协议属性命名。我们在第 6 章中详细讲过这些属性。show 和 status 属性来自于联系人的<presence>节，而 name、subscription 和 ask 属性则属于给定联系人的用户花名册状态的一部分。

但关于 show 和 status 属性我们稍微做了一点简化。当联系人的出席信息中没有<show>元素时，他们被认为是在线的并可访问，因此 show 属性被设为 available。类似地，如果联系人的出席信息没有<status>元素，那么 status 属性被设为空字符串。

最后，groups 属性指示联系人属于哪些花名册组。

让插件的用户来编写代码枚举 resources 属性来判断联系人是否在线是一件相当乏味的工作。我们可以让这些程序员的生活变得更加轻松，即提供一种实用工具方法根据 resources 属性值来计算出这个属性。



可从
Wrox.com
下载源代码

```

online: function () {
    var result = false;
    for (var k in this.resources) {
        result = true;
        break;
    }
    return result;
}

```

code snippet strophe.roster.js

现在我们已经有足够的信息来首次勾勒插件代码的骨架。下面的版本虽然尚未提供多少功能，但在下面几节中我们将对其进行扩展。将下面的代码添加到 strophe.roster.js 文件中。



```

// Contact object
function Contact() {
    this.name = "";
    this.resources = {};
    this.subscription = "none";
    this.ask = "";
    this.groups = [];
}

Contact.prototype = {
    // compute whether user is online from their
    // list of resources
    online: function () {
        var result = false;
        for (var k in this.resources) {
            result = true;
            break;
        }
        return result;
    }
};

Strophe.addConnectionPlugin('roster', {
    init: function (connection) {
        this.connection = connection;
        this.contacts = {};

        Strophe.addNamespace('ROSTER', 'jabber:iq:roster');
    }
});

```

code snippet strophe.roster.js

14.3.2 获取并维护花名册

我们在第 6 章中曾经了解过如何获取和处理花名册，现在我们将再次运用这些技能在插件中生成花名册状态。首先，一旦连接建立必须从服务器那里获得花名册最初副本。之后，当联系人出席状态改变时需要更新他的信息。该插件还必须触发一个事件，这样当新的花名册发生变化时，用户的代码会得到通知。

在第 6 章中，我们学习了如何从服务器那里获取花名册。每当连接建立时插件都必须这样做。毕竟，当一个客户端中的用户断开连接时，另一个客户端可能已经做出修改。此外，当插件接收到断开连接通知时，它需要将花名册表示置入一个合适的状态。

下面的代码实现了 `statusChanged()` 函数来处理这些任务(针对 CONNECTED 和 DISCONNECTED 状态)。它还通知 `roster_changed` 事件的所有处理程序花名册已经更新。我们应该将这段代码插入到插件的原型中，放在 `init()` 函数之后。



```

statusChanged: function (status) {
    if (status === Strophe.Status.CONNECTED) {
        this.contacts = {};
        // build and send initial roster query
        var roster_iq = $iq({type: "get"})
            .c('query', {xmlns: Strophe.NS.ROSTER});

        var that = this;
        this.connection.sendIQ(roster_iq, function (iq) {
            $(iq).find("item").each(function () {
                // build a new contact and add it to the roster
                var contact = new Contact();
                contact.name = $(this).attr('name') || "";
                contact.subscription = $(this).attr('subscription') ||
                    "none";
                contact.ask = $(this).attr('ask') || "";
                $(this).find("group").each(function () {
                    contact.groups.push(this.text());
                });
                that.contacts[$(this).attr('jid')] = contact;
            });

            // let user code know something happened
            $(document).trigger('roster_changed', that);
        });
    } else if (status === Strophe.Status.DISCONNECTED) {
        // set all users offline
        for (var contact in this.contacts) {
            this.contacts[contact].resources = {};
        }

        // notify user code
        $(document).trigger('roster_changed', this);
    }
}

```

code snippet strophe.roster.js

该插件现在能够记录基本的花名册信息，但当花名册改变时或当联系人改变出席状态时，它并不能让该信息保持更新。我们可以通过为携带来自服务器的花名册更新信息的 IQ-set 节设置 XMPP 节事件处理程序来解决第一个问题。第二个问题则可以通过添加出席节处理程序来解决。

在第 6 章中，我们研究了如何通过添加、更新和删除联系人来修改花名册。服务器还通知其他已连接资源有关花名册的变化，这样每个客户端的状态就保持一致。例如，如果 Darcy 在连接到资源 library 时将 Wickham 从他的花名册移除，那么他的另一个资源 drawing_room 就会得到有关该变化的通知。首先，Darcy 将 Wickham 从他的花名册中删除。

```

<iq from='darcy@pemberley.lit/library'
    type='set'

```

```

    id='delete1'>
<query xmlns='jabber:iq:roster'>
    <item jid='wickham@militia.lit' subscription='remove' />
</query>
</iq>

```

服务器将删除 Wickham 并通知 Darcy 的所有已连接资源有关该花名册变化的信息。Darcy 的 drawing_room 资源将接收到如下信息。

```

<iq to='darcy@pemberley.lit/drawing_room'
    type='set'
    id='deleted1'>
<query xmlns='jabber:iq:roster'>
    <item jid='wickham@militia.lit' subscription='remove' />
</query>
</iq>

```

Darcy 的 library 资源现在也会接收到同样的更新，尽管正是它请求的修改。服务器总是将花名册状态变化通知给所有的资源。

```

<iq to='darcy@pemberley.lit/library'
    type='set'
    id='deleted1'>
<query xmlns='jabber:iq:roster'>
    <item jid='wickham@militia.lit' subscription='remove' />
</query>
</iq>

```

两个客户端都必须用 IQ-result 来确认 IQ-set 节。此时，所有客户端都会拥有花名册状态的相同视图。

可以扩展 statusChanged() 来设置花名册更新的处理程序并向插件原型中添加一个新的函数来处理这些变化。下面的代码给出了 statusChanged() 的修改版，新加的代码行被突出显示，其后是新的 rosterChanged() 函数。因为服务器每次总是发送单条更新，所以 rosterChanged() 只需要处理一个<item>子元素。



可以从
Wrox.com
下载源代码

```

statusChanged: function (status) {
    if (status === Strophe.Status.CONNECTED) {
        this.contacts = {};

        this.connection.addHandler(this.rosterChanged.bind(this),
        Strophe.NS.ROSTER, "iq", "set");
        // build and send initial roster query
        var roster_iq = $iq({type: "get"})
            .c('query', {xmlns: Strophe.NS.ROSTER});

        var that = this;
        this.connection.sendIQ(roster_iq, function (iq) {

```

```

$(iq).find("item").each(function () {
    // build a new contact and add it to the roster
    var contact = new Contact();
    contact.name = $(this).attr('name') || "";
    contact.subscription = $(this).attr('subscription')
        "none";
    contact.ask = $(this).attr('ask') || "";
    $(this).find("group").each(function () {
        contact.groups.push(this.text());
    });
    that.contacts[$(this).attr('jid')] = contact;
});

// let user code know something happened
$(document).trigger('roster_changed', that);
});

} else if (status === Strophe.Status.DISCONNECTED) {
    // set all users offline
    for (var contact in this.contacts) {
        this.contacts[contact].resources = {};
    }

    // notify user code
    $(document).trigger('roster_changed', this);
}
}

rosterChanged: function (iq) {
    var item = $(iq).find('item');
    var jid = item.attr('jid');
    var subscription = item.attr('subscription') || "";

    if (subscription === "remove") {
        // removing contact from roster
        delete this.contacts[jid];
    } else if (subscription === "none") {
        // adding contact to roster
        var contact = new Contact();
        contact.name = item.attr('name') || "";
        item.find("group").each(function () {
            contact.groups.push(this.text());
        });
        this.contacts[jid] = contact;
    } else {
        // modifying contact on roster
        var contact = this.contacts[jid];
        contact.name = item.attr('name') || contact.name;
        contact.subscription = subscription || contact.subscription;
        contact.ask = item.attr('ask') || contact.ask;
        contact.groups = [];
        item.find("group").each(function () {
            contact.groups.push($(this).text());
        });
    }
}
}

```

```

        });
    }

    // acknowledge receipt
    this.connection.send($iq({type: "result", id: $(iq).attr('id')}));

    // notify user code of roster changes
    $(document).trigger("roster_changed", this);

    return true;
}

```

code snippet strophe.roster.js

处理花名册更新之后，现在可以继续处理出席更新。我们需要向 `statusChanged()` 中添加一个新的出席节处理程序并向插件的原型中添加一个新函数 `presenceChanged()`。`presenceChanged()` 函数只需要根据出席节信息添加、修改或删除联系人的 `resources` 属性中的适当属性即可。

将下面的代码行添加到 `statusChanged()` 前一版本的 `addHandler()` 代码行之后。



可从
Wrox.com
下载源代码

```

this.connection.addHandler(this.presenceChanged.bind(this),
    null, "presence");

```

code snippet strophe.roster.js

现在，将 `presenceChanged()` 的实现添加到原型中。



可从
Wrox.com
下载源代码

```

presenceChanged: function (presence) {
    var from = $(presence).attr("from");
    var jid = Strophe.getBareJidFromJid(from);
    var resource = Strophe.getResourceFromJid(from);
    var ptype = $(presence).attr("type") || "available";

    if (!this.contacts[jid] || ptype === "error") {
        // ignore presence updates from things not on the roster
        // as well as error presence
        return true;
    }

    if (ptype === "unavailable") {
        // remove resource, contact went offline
        delete this.contacts[jid].resources[resource];
    } else {
        // contact came online or changed status
        this.contacts[jid].resources[resource] = {
            show: $(presence).find("show").text() || "online",
            status: $(presence).find("status").text()
        };
    }

    // notify user code of roster changes

```

```

$(document).trigger("roster_changed", this);
}

```

code snippet strophe.roster.js

现在，该插件在连接的整个生命周期中负责维护花名册状态。插件用户在任何时候都可以通过 `connection.roster.contacts` 来访问该信息。现在，只剩下帮助开发人员修改花名册这项任务了。

14.3.3 操纵花名册

在第 14.3.2 节中，我们处理三种花名册操纵的通知：添加、修改和删除联系人。现在我们来实现这项功能的另一面，即让插件的用户通过简单的 `addContact()`、`deleteContact()` 和 `modifyContact()` 调用来修改花名册。还有两种情况会改变花名册：每当接收到出席订阅时添加联系人，而每当用户退订某个联系人的出席信息时通常会将其删除。我们将使用 `subscribe()` 和 `unsubscribe()`，使得订阅和退订就像其他花名册修改操作一样容易。

直接的花名册操纵辅助器非常简单。因为插件已经处理变化通知，所以用户代码完全没有必要修改 `contacts` 属性。一旦服务器已经处理花名册修改工作，它就会生成变化通知，而该通知将触发我们编写的处理程序来完成对 `contacts` 属性的适当修改。我们只需要向服务器发送适当的 IQ-sets 节来启动事件链条即可。在此之上，花名册项修改操作与添加操作完全一样，因此我们可以重用同样的代码。

下面的代码中实现了这三个函数。该代码与我们在第 6 章中编写用来完成同样操作的代码非常相似。



可从
Wrox.com
下载源代码

```

addContact: function (jid, name, groups) {
    var iq = $iq({type: "set"})
        .c("query", {xmlns: Strophe.NS.ROSTER})
        .c("item", {name: name || "", jid: jid});
    if (groups && groups.length > 0) {
        $.each(groups, function () {
            iq.c("group").t(this).up();
        });
    }
    this.connection.sendIQ(iq);
}

deleteContact: function (jid) {
    var iq = $iq({type: "set"})
        .c("query", {xmlns: Strophe.NS.ROSTER})
        .c("item", {jid: jid, subscription: "remove"});
    this.connection.sendIQ(iq);
}

modifyContact: function (jid, name, groups) {
    this.addContact(jid, name, groups);
}

```

code snippet strophe.roster.js

回忆一下，订阅和退订某个联系人的出席信息是一个两步过程。对于订阅而言，客户端首先添加一个新联系人，然后发送出席订阅请求。对于退订请求而言，客户端首先发送正确的出

席节，然后将该联系人从花名册中删除。下面的代码使用我们刚刚编写的花名册操纵函数，应将其添加到插件原型中。



可从
Wrox.com
下载源代码

```
subscribe: function (jid, name, groups) {
    this.addContact(jid, name, groups);

    var presence = $pres({to: jid, "type": "subscribe"});
    this.connection.send(presence);
},

unsubscribe: function (jid) {
    var presence = $pres({to: jid, "type": "unsubscribe"});
    this.connection.send(presence);

    this.deleteContact(jid);
}
```

code snippet strophe.roster.js

现在新的花名册插件已经完工。下面就来看看它能够做些什么。

14.4 试用插件

Strophe 插件应该让开发人员的工作要比直接处理繁琐的细节更加轻松。即使我们在第 14.3 节中开发的简单插件也非常有用。为了说明这一点，我们构建一个小应用程序，当其他资源正在操纵联系人时，用它展示当前的花名册状态和更新。

程序清单 14-2 和程序清单 14-3 给出了 RosterWatch 应用程序的 HTML 代码和 CSS 样式。我们在前面已经看过所有这些代码。在 HTML 文件中需要注意的主要是插件文件 strophe.roster.js 以及 Strophe 库。



可从
Wrox.com
下载源代码

程序清单 14-2 rosterwatch.html

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

<html>
    <head>
        <meta http-equiv="Content-type" content="text/html; charset=UTF-8">
        <title>RosterWatch - Chapter 14</title>

        <link rel='stylesheet' href='http://ajax.googleapis.com/ajax/libs/jqueryui
        /1.7.2/themes/cupertino/jquery-ui.css'>
        <script src='http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/jquery.js'>
        </script>
        <script src='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/jquery-
        ui.js'></script>
```

```

<script src='scripts/strophe.js'></script>
<script src='scripts/flXHR.js'></script>
<script src='scripts/strophe.flxhr.js'></script>

<script src='strophe.roster.js'></script>

<link rel='stylesheet' type='text/css' href='rosterwatch.css'>
<script src='rosterwatch.js'></script>
</head>
<body>
<h1>RosterWatch</h1>

<div class='toolbar'>
<input id='disconnect' type='button' value='disconnect'
disabled='disabled'>
</div>

<div id='roster'>
</div>

<!-- login dialog -->
<div id='login_dialog' class='hidden'>
<label>JID:</label><input type='text' id='jid'>
<label>Password:</label><input type='password' id='password'>
</div>
</body>
</html>

```



程序清单 14-3 rosterwatch.css

可从
Wrox.com
下载源代码

```

body {
    font-family: Helvetica;
}

h1 {
    text-align: center;
}

.toolbar {
    text-align: center;
}

#roster {
    width: 500px;
    margin: auto;
    border: solid 1px black;
}

.hidden {
    display: none;
}

```

```

.contact {
    padding: 10px;
}

.name {
    font-size: 150%;
    font-weight: bold;
}

.jid {
    font-size: 80%;
    font-style: italic;
}

.online {
    background-color: #7f7;
}

.away {
    background-color: #f77;
}

.offline {
    background-color: #777;
}

```

这个应用程序的实际 JavaScript 代码非常简单，程序清单 14-4 列出了该代码。因为所有与花名册更新处理的繁重工作已经交由插件处理，所以该代码只处理了非常少量的 XMPP 工作，即建立连接并发送初始出席信息。roster_changed 事件处理程序只是遍历花名册联系人并输出 HTML，就像所有现代的动态 Web 应用程序一样。



可从
Wrox.com
下载源代码

程序清单 14-4 rosterwatch.js

```

RosterWatch = {
    connection: null
};

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        modal: true,
        title: 'Connect to XMPP',
        buttons: {
            "Connect": function () {
                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val()
                });
            }
        }
});

```

```
        $('#password').val('');
        $(this).dialog('close');
    }
}

$('#disconnect').click(function () {
    $('#disconnect').attr('disabled', 'disabled');
    RosterWatch.connection.disconnect();
});

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        'http://bosh.metajack.im:5280/xmpp-httpbind');
    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });
    RosterWatch.connection = conn;
});

$(document).bind('connected', function () {
    $('#disconnect').removeAttr('disabled');

    RosterWatch.connection.send($pres());
});

$(document).bind('disconnected', function () {
    RosterWatch.connection = null;

    $('#roster').empty();
    $('#login_dialog').dialog('open');
});

$(document).bind('roster_changed', function (ev, roster) {
    $('#roster').empty();

    var empty = true;
    $.each(roster.contacts, function (jid) {
        empty = false;

        var status = "offline";
        if (this.online()) {
            var away = true;
            for (var k in this.resources) {
                if (this.resources[k].show === "online") {
                    away = false;
                }
            }
        }
    })
});
```

```

        status = away ? "away": "online";
    }

    var html = [];
    html.push("<div class='contact " + status + "'>");
    html.push("<div class='name'>");
    html.push(this.name || jid);
    html.push("</div>");

    html.push("<div class='jid'>");
    html.push(jid);
    html.push("</div>");

    html.push("</div>");

    $('#roster').append(html.join(''));
});

if (empty) {
    $('#roster').append("<i>No contacts:(</i>\"");
}
});

```

我们创建的花名册插件确实很好地将花名册管理的细节抽离出来，这正是我们要实现的目标。

14.5 改进花名册插件

这里创建的花名册插件非常简单，虽然它仍然非常有用。下面是一些可以尝试改进的地方：

- 在 `roster_changed` 事件中发送更详细的更新信息，这样用户的代码就不必每次都要刷新整个花名册。
- 扩展插件以触发更多像传入的出席订阅和联系人上线和离线这样的事件。
- 添加对 Personal Eventing Protocol(XEP-0163)以及 Entity Capabilities(XEP-0115)的支持以获取延伸信息，比如联系人正在欣赏什么音乐。

14.6 小结

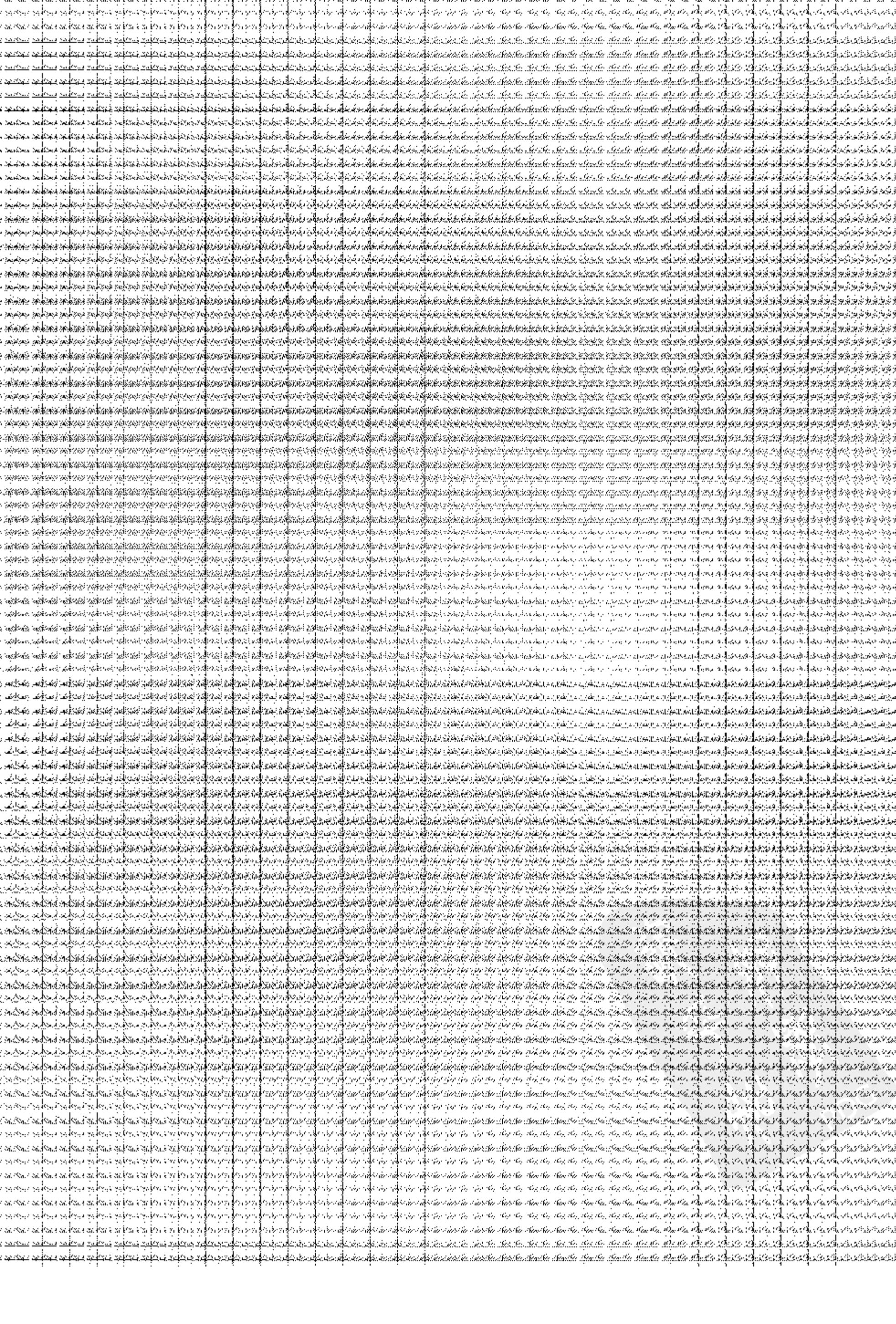
通过将协议细节抽离出来并让开发人员重用并在他人工作基础之上轻易地构建，插件使开发人员的工作变得更加轻松。Strophe 库提供了一种简单但极为高效的插件系统可让它的用户添加高级功能。在本章中我们学习了如下的内容：

- 学习如何加载并使用 Strophe 插件
- 研究如何创建自己的插件

- 创建一个花名册管理插件，真正简化花名册、花名册更新以及出席变化的处理
- 当使用插件来处理繁重的工作之后应用程序的逻辑可以变得极为简洁

到目前为止我们已经学习了如此多的内容，我希望您能够对这些内容感兴趣。我们构建了九个应用程序，其中包括几个非常复杂的应用程序，并学习如何扩展 XMPP 应用程序。现在您应该已经精通 XMPP 协议和 Strophe 库，或许还学到了一些新的 JavaScript 和 jQuery 技术。希望您已经掌握了优秀的 XMPP 应用程序所需的精彩技术。





附录 A

jQuery 入门

很长时间以来 JavaScript 程序员一直在处理 DOM(Document Object Model, 文档对象模型)API 的种种风格。许多人创建了辅助函数将特定的操作抽象出来, 这样他们就能够跨越所有浏览器运行, 而有些最佳的抽象已经足够优雅, 能够普遍地应用于许多应用程序。jQuery 库是一种轻量级的、跨浏览器的 JavaScript 库, 它使用灵敏的基于 CSS 选择器的 API 来取代 DOM API 中丑陋的部分, 能够快速完成大多数任务。

jQuery 非常擅长于操纵 HTML, 但它对 XML 文档的处理也非常在行。因为 XMPP 节只是一些 XML 文档片段, 所以 jQuery 使得 XMPP 的操纵要比使用原始的 DOM API 更加简单。

Strophe 本身受到了 jQuery 中函数链式调用、回调的使用以及插件架构的启发, 因此这两个库非常自然地组合在一起。

如果以前没有用过或看过 jQuery, 那么这个附录将向您讲授一些基本知识, 并让您了解一些背景知识以便能够在自己的 XMPP 应用程序中高效地使用 jQuery。jQuery 库还为 Web 编程提供了其他许多有用的函数, 比如 AJAX 请求和动画, 但这里并没有涉及, 这是因为本书中并没有用到这些方面。

A.1 寻找 jQuery

可以在 <http://jquery.com> 网站找到库。jQuery 的网站还包含了一些手册、参考文档、示例以及一个长长的以各种有趣方式扩展 jQuery 的第三方插件列表。

可以下载这个库以便在自己的应用程序中使用, 或者按照第 3 章中的做法, 可以使用 Google 作为它的 AJAX Libraries API 的一部分而提供的托管版本。本书中的应用程序使用的是 Google 托管版本。

可以通过在页面中包含<script>元素来加载 jQuery, 如下所示。

```
<script src='http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/jquery.js'>
</script>
```

在生成应用程序中，可能希望使用缩减版库，这会减少该库的尺寸，从而让应用程序的启动时间更快。缩减版通常名为 `jquery.min.js`，而不是 `jquery.js`。

一旦 `jQuery` 加载到应用程序中，就可以通过 `$()` 函数访问它的功能。这个样子有趣的函数有着简短的名称，从而可以节省大量的输入，因为它通常会成为应用程序中使用最多的函数。

A.2 jQuery 与 DOM

在深入研究 `jQuery` 的细节之前，让我们看看下面的示例，将 `DOM` 和 `jQuery` 做一下比较。这个示例应该能够让我们感受一下 `jQuery` 代码的样子以及它与直接使用 `DOM API` 的不同之处。

下面的代码有一个简单的任务。它查找当前页面中的所有 `<p>` 元素，并将它们的背景色修改为亮红色。首先给出 `DOM API` 版本。

```
var p_elems = document.getElementsByTagName('p');
for (var i = 0; i < p_elems.length; i++) {
    p_elems[i].style.backgroundColor = '#faa';
}
```

现在使用 `jQuery` 来完成同样的任务。

```
$( 'p' ).css({backgroundColor: '#faa'});
```

`jQuery` 代码不仅要短得多，而且也要清晰得多。它使用 `CSS` 选择器语法选取一组元素并将一个新的背景色应用到所有匹配的元素。

上面示例中的 `DOM` 代码能够在所有浏览器中运行，但 `jQuery` 实际上真正的闪光之处在于使用那些在不同浏览器上不一致的 `DOM` 功能时。对于不一致的 `DOM API` 函数，`jQuery` 代码同样简短，但若不使用 `jQuery` 的同等代码则大量充斥用于检查兼容性的 `if` 语句。

A.3 使用 `jQuery`

`jQuery` 的设计非常简单却很强大：查找一组元素并在结果集合上执行某种操作。

A.3.1 查找元素

在前一节中看到，`jQuery` 的 `$()` 函数接受一个扩展版 `CSS` 选择器语法，而不使用 `DOM` 的 `getElementById()` 或 `getElementsByName()`。这使得元素的查找速度非常快而且非常简单。一旦完成最初的选取，`jQuery` 就包含了一些辅助函数来增强该选集(如果需要的话)。

除了 `CSS` 选择器之外，`jQuery` 的 `$()` 函数还接受 `DOM` 元素。给定的 `DOM` 元素将变成 `jQuery` 选集。

1. 选择器

与 CSS 类似, jQuery 支持以下基本选择器。

- `anelement`: 选取带有名字 `anelement` 的元素
- `.aclass`: 选取带有类名 `aclass` 的元素
- `#anid`: 选取带有唯一 ID `andi` 的元素

这些选择器可以像 `p.error` 和 `input#textbox` 这样组合使用以形成更具体的选择器。多个选择器之间可以用逗号隔开, 最终的选择器将找出匹配这些选择器中任何一个的元素。

还可以通过将多个选择器用空格隔开来查找特定层次结构中的元素。例如, 选择器 `#chat p` 查找标识为 `chat` 的元素中的所有 `<p>` 子元素。还可以使用 `>`、`+` 和 `-` 来指定更具体的关系。选择器 `div > p` 只查找属于 `<div>` 元素的直接子节点的 `<p>` 元素。使用 `h1 + p` 只查找紧跟 `<h1>` 兄弟节点之后的 `<p>` 元素, 而与此类似, `h1 - p` 选取紧靠在 `<h1>` 兄弟节点之前的 `<p>` 元素。

选择器还可以进行过滤。添加 `:first` 或 `:last` 只选取匹配剩余部分选择器的那些元素中的第一个或最后一个元素。过滤器 `:visible` 和 `:hidden` 与此类似, 但将选取限制到用户能够在屏幕上看到的或看不到的元素。

最后, jQuery 为属性提供了几种过滤器。主要的变种是 `[anattr]`(匹配携带 `anattr` 属性的元素) 以及 `[anattr='somevalue']`(匹配携带 `anattr` 属性而且该属性的值为 `'somevalue'` 的元素)。

看看您能否判断出下面的选择器将匹配什么元素:

- `p:first`
- `ul > li .name`
- `input[type='text'], input[type='password']`

2. 增强选集

一旦选取之后, jQuery 就包含了大量的方法来增强我们需要的元素。这里讨论的是最常见的方法。

`filter()` 方法用于从给定选集中挑选出元素子集。如果需要对很大的元素选集执行一个常见的操作, 但其中只有更小的子集需要执行第二个操作, 那么在第一次操作之后可以使用 `filter()` 来缩小选集, 然后执行第二个操作。

`children()` 函数用来过滤列表以找出匹配元素的所有直接子节点。这个方法经常用来匹配 XMPP 节的一级子节点。例如, 为了从 XMPP 节中取出 `<error>` 元素, 可以使用 `$(stanza).children('error')`。

`find()` 用来从当前选集的子节点中查找匹配元素。因为 XMPP 节通常只从选中的顶级元素开始, 所以 `find()` 通常用来选择该元素下面的子元素。

`not()` 将匹配的元素从选集中删除。这个函数可用来将一两个元素从一个选集中删除。

A.3.2 在元素上执行操作

jQuery 支持的在选中元素上执行的操作包括修改 CSS 属性、获取或设置元素及其属性的内容以及绑定事件处理程序。jQuery 还支持几个受到函数式编程启发的方法, 比如 `each()` 和 `map()`, 这些方法可用来对整个选集执行遍历和函数应用。我们稍后将会讲解这些函数。

A.3.3 链式调用

函数链式调用进一步增强了 jQuery 的可用性和功能。链式调用可以让我们在同一选集上轻易地执行多项操作。

jQuery 的大多数方法会将 jQuery 对象本身以及它对应的选集一起返回。这意味着该对象上的方法调用可以一个接一个地链接起来。

下面的代码给出了一个典型的调用对象方法的示例。

```
var obj = new SomeObject();
obj.foo();
obj.bar(2);
obj.baz('XMPP');
```

如果 SomeObject 的每个方法返回该对象的实例，那么这些方法就可以链接起来。下面给出了链式调用。

```
var obj = new SomeObject();
obj.foo().bar(2).baz('XMPP');
```

如果这是您第一次看到这样的构造，那么这看起来可能有点奇怪。事实证明，对于专门为
此设计的库(比如 jQuery)来说，这实际上非常有用。Strophe 使用了链式调用来简化 XMPP 节的
构建工作，我们在第 3 章中曾经讨论过。

下面的 jQuery 示例易于理解而且相当具有可读性，即使用户从前从未见过该库。

```
$(‘p.error’).css({backgroundColor: ‘#f00’, fontHeight: ‘14pt’}).show();
$(‘#lists ul’).append(‘- New item
’).append(‘- Another new item
’);
```

A.4 操纵元素

本书中我们使用 jQuery 完成的大部分工作与从元素中提取数据、添加新元素或修改现有元素有关。本节讲解 jQuery 支持的最常见的操作以及本书的应用程序使用的操作。

A.4.1 数据提取

在处理 XMPP 节时，jQuery 主要用来从 XML 中提取有用的信息，就像是 XPath 的编程版本。

attr() 用来获取和设置元素的属性。为了获取一个属性，要传入该属性的名称，而为了设置一个属性，需要同时传入名称和预期的值。如果当前的 jQuery 选集包含多个元素，那么调用 attr(name) 时只影响到第一个元素，而当调用 attr(name, value) 时则会影响所有元素。下面的示例是 attr() 的典型用法。

```
$(stanza).find(‘item’).attr(‘jid’); // get the first item’s jid attribute
```

```
// set the form's input elements to buttons
$('#someform input').attr('type', 'button');
```

`text()`和`html()`获取元素的文本内容或 HTML 内容。因为我们使用 Strophe 的构建器来构造大多数 XMPP 节，所以本书大多数时候都在使用这些方法来获取值而不是设置值。

```
var body = $(stanza).find('body').text(); // get the message body contents
$('#chat_input').text('Type your message here..'); // set the text box prompt
```

A.4.2 元素样式化

大多数动态 Web 应用程序通过操作各种元素的样式来完成许多 UI 任务。元素可以显示或隐藏，改变颜色和大小，或者通过编程方式操纵它们的 CSS 类。下面的 jQuery 方法提供了这种行为。

`show()`和`hide()`可以根据它们的名称来推断它们的功能。它们改变所有选中元素的可见性。通常用来让对话框弹出到视图中，或者将一个标签页的内容替换成另一个标签页的内容。

`css()`可以检索元素的特定 CSS 样式，或改变它的一项或多项样式属性。下面的代码给出了该方法的几种典型用法。

```
var c = $('#tab1').css('color'); // get the tab's font color
$('.username').css('background-color', '#fff'); // set a single css style
$('.errorMsg').css({fontSize: '150%', color: 'red'}); // change multiple styles at once
```

注意，样式的名称可以写作`fontSize`或`font-size`。jQuery 允许我们省去引号和连接符。

`addClass()`和`removeClass()`改变对选中元素起作用的 CSS 类。这些方法提供了一种一次性修改大量 CSS 属性(而不必单独地指定它们)的快速方法。

A.4.3 添加和移除元素

显示和隐藏元素可以完成大量的 UI 工作，但有时候必须向 DOM 中添加元素或将其彻底从 DOM 中移除。例如，在第 6 章的聊天应用程序中，当从服务器接收到花名册项时，向联系人列表中添加新项。

可以使用`append()`和`prepend()`来添加内容，或者使用前面提到的`html()`函数。所有这些函数均携带一个包含着待追加、前置或替换的新 HTML 内容的字符串。

`remove()`将选中的元素从 DOM 中彻底删除。`empty()`将选集中每个元素的所有子元素移除，它经常用来清空列表或整个`<div>`。

A.4.4 遍历选集

应用程序经常需要对选集中的所有元素执行某个函数。在大多数流行的语言中，这通常通过`for`或`while`循环的帮助来完成的，但 jQuery 可以让我们借助函数式编程语言的模式来轻

易地完成这项工作。

jQuery 对象支持 `length` 属性，就像普通的 JavaScript 数组一样，而且选集中的所有元素都能够通过普通的数组方括号表示法访问。下面的代码利用这些功能，遵照传统的遍历模式向列表中添加联系人。

```
var i;
var items = $(stanza).find('item');
for (i = 0; i < items.length; i++) {
    $('#contact_list').append('<li>' + $(items[i]).attr('jid') + '</li>');
}
```

该代码查找 XMPP 节中的所有`<item>`元素，然后从每个元素中提取 `jid` 属性并使用该属性向联系人列表中添加新元素。注意，我们用`$()`将 `items[i]` 包装起来，这是因为它是普通的 DOM 元素，而不是 jQuery 选集。在该元素上调用`$()`之后，它就成为一个普通的只有一个元素的 jQuery 选集。

同样的动作可以通过在最初的选集上使用 jQuery 的 `each()`方法来完成：

```
$(stanza).find('item').each(function () {
    $('#contact_list').append('<li>' + $(this).attr('jid') + '</li>');
});
```

`each()`携带一个函数作为其实参，而该函数将为每项执行预期的动作。每次调用该函数时，`this` 变量引用遍历的当前项。因为该项是普通的 DOM 元素，所以为了使用 `attr()`方法，必须首先使用`$()`函数将其包装起来。

请注意，这里消除了大量不必要的代码，比如变量 `i`，从而让代码非常具有可读性。

`each()`还可用在普通的 JavaScript 数组上，通过使用该方法的全局版本，并作为全局`$`对象的一个属性来访问。

```
var list = [1, 2, 3, 4, 5];
$.each(list, function () {
    $('body').append('called for item ' + this);
});
```

A.5 处理事件

除了处理 DOM 元素本身以外，Web 应用程序还必须处理和触发文档和特定元素上的事件。两种不同浏览器上的事件模型的行为差异可能相当巨大，但 jQuery 为事件提供了一个一致的接口来简化它们的使用。

除了基本的用户交互事件，比如单击按钮或页面完成载入，jQuery 还提供了对自定义事件的支持。自定义事件可用来实现各种有趣的功能。可以让应用程序中的一部分 UI 触发一个 `contact_added` 事件，或者将一系列简单的时间转换成一个在特定应用程序中有着特定意义的组

合事件。

自定义事件还为应用程序代码解耦提供了简单的方法。我们并不让一部分代码直接调用另一部分代码，相反，可以触发一个让其他部分代码能够侦听的事件。如果应用程序的多个部分需要参与到同一逻辑中，那么它们只需侦听并响应相同的事件即可，而不必修改其他的代码来调用新函数。

自定义事件的处理方式与普通的 DOM 事件完全相似，唯一的不同之处在于事件名称不同，jQuery 并没有为自定义事件提供方便命名的辅助函数。

A.5.1 基本的事件方法

基本的事件处理方法是 `bind()` 和 `trigger()`。`bind()` 用来侦听事件，而 `trigger()` 通知侦听者已经发生一个事件。

`bind()` 接受两个参数：事件名称和该事件触发时执行的函数。

下面的示例代码将添加一个按钮的单击事件的事件处理程序。

```
$('#go_button').bind('click', function () {
    // do something here
});
```

我们的代码可以使用 `trigger()` 函数来触发自定义事件和普通的 DOM 事件。`trigger()` 的第一个实参是事件名称，可选的第二个实参是能够传给事件处理程序的额外数据。

```
$('#go_button').trigger('click'); // click the button programmatically
// trigger a custom event with some event specific data
$(document).trigger('contact_added', {jid: 'darcy@pemberley.lit'});
```

普通事件的处理程序会传入一个实参，即事件对象。这里的 `on_click()` 处理程序演示了这一点。

```
function on_click(ev) {
    // handler code
}
```

如果通过第二个实参为 `trigger()` 传入一些数据，那么该数据将放入第二个参数中传给处理程序。

```
function on_new_contact(ev, data) {
    // add the new contact
}
```

A.5.2 便利事件方法

为了减少输入并改善可读性，jQuery 为常见的 DOM 事件提供了几个便利的方法。

大多数 DOM 事件在 jQuery 中都有一个与事件具有相同名称的对应方法。例如，`click()` 方

法触发单击事件。当不带实参调用这些方法时，就会触发相应的事件，但当传入实参给函数时，它们会为该事件绑定新的处理程序。第 A.5.1 节的按钮单击事件处理程序示例可以重新编写。

```
$('#go_button').click(function () {
    // do something here
});
```

jQuery 提供的方法包括 `keypress()`、`keyup()`、`keydown()`、`mouseup()`、`mousedown()`、`mousemove()`、`click()`、`load()` 和 `ready()`。

A.5.3 文档准备就绪事件

文档准备就绪事件有点特殊。一旦 DOM 可供 JavaScript 代码访问和操作就立即触发该事件。Web 应用程序的大多数初始化和启动代码通常都会放入一个文档准备就绪事件处理程序。

要把一个处理程序绑定到文档准备就绪事件，可以使用如下方法。

```
$(document).bind('ready', function () {
    // init code here
});
```

或：

```
$(document).ready(function () {
    // init code here
});
```

因为这个构造非常常见，所以 jQuery 也提供了一个更为简单的快捷方式。只需将处理程序直接传给 `$()` 函数。

```
$(function () {
    // init code here
});
```

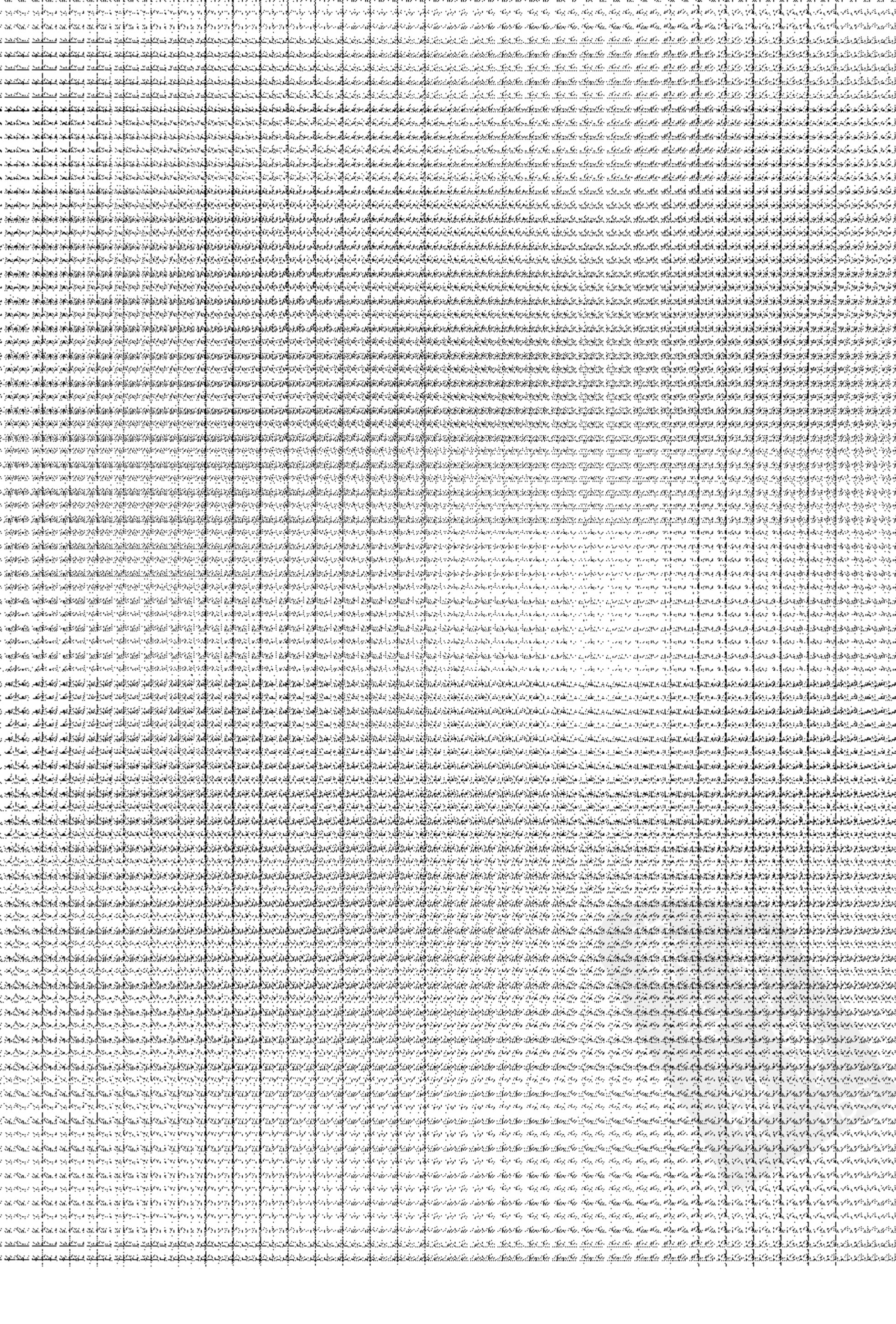
许多 jQuery 程序员似乎在使用最短的构造，但本书使用 `ready()` 以便尽可能让代码易于理解。我们将会看到其他 DOM 事件也使用各自的特殊命名的方法进行处理，而自定义事件是通过 `bind()` 和 `trigger()` 来进行管理的。

A.6 进一步阅读

所有的 jQuery 方法在它的极佳的参考文档中都有完整的讲解，可以在 <http://docs.jquery.com> 网页中找到该文档。文档维基站点也有一些指向大量手册的指针，可以在 <http://docs.jquery.com/Tutorials> 网页中找到这些指针的列表。

jQuery 还有一个庞大的程序员社区在不断地通过插件来扩展该库，非常像第14 章中讨论的 Strophe 插件。本书中使用的 jQuery UI 插件是这些插件中最著名的插件之一，它为 Web 应用程序提供了丰富的用户界面组件。本书中用到的 jQuery UI 功能已经在它们所在的章中讲解过了。可以在 <http://ui.jquery.com> 找到更多有关 jQuery UI 的信息，而在 <http://plugins.jquery.com> 可以找到 jQuery 插件的完整列表。





附录 B

设置 BOSH 连接管理器

Web 浏览器尚不能理解 XMPP 协议。为了在 Web 应用程序中使用 XMPP 协议，需要一位中间人将无状态的 HTTP 请求转换成有状态而且长寿命的 XMPP 连接。围绕在这个过程周围的一些细节被称为 BOSH(Bi-directional streams Over Synchronous HTTP, 通过同步 HTTP 传送双向流)，而提供这种服务的服务器则被称为连接管理器。

本附录将讲解为 XMPP 应用程序选择和安装 BOSH 连接管理器所需要知道的信息。本书中的应用程序全部使用了一款专为测试目的而提供的公共 BOSH 连接管理器。如果打算让自己的应用程序大规模使用，那么可能希望设置自己的服务器。

由于 JavaScript 的同源策略(参见第 3.3 节)，与连接管理器直接通信通常被视为安全违例。为了绕开这项策略，本书中的应用程序使用了 Flash 库 flXHR 让我们能够发送跨域请求。此外，还可以使用反向代理，从而让连接管理器就像是自己的 Web 服务器的一部分。

本附录的最后一部分将讨论如何配置反向代理连接管理器以应对那些不能或不希望使用 Flash 的场合。

B.1 选择连接管理器

连接管理器有两种主要形式：内置在 XMPP 服务器中或独立的管理器。每种类型都有各自的优缺点，根据特定应用程序的需求它们可能合适也可能不合适。

B.1.1 独立管理器

最初的连接管理器曾经都是独立的，这是因为 XMPP 服务器团队还没有来得及将 BOSH 服务整合到它们的系统中。独立的连接管理器作为单独的服务或守护进程运行，可能是在远离 XMPP 服务器的一台计算机上运行。

独立的管理器通常可以连接到 Internet 上任何可达的 XMPP 服务器。当接收到建立 XMPP 连接的请求时，连接管理器代表请求方向期望的服务器建立一个普通的 XMPP 客户端连接。当应用程序的用户已经在各种不同的 XMPP 服务器上拥有 XMPP 账户时，这种方式让独立管理

器成为很好的选择。

因为独立管理器必须代表用户维护 XMPP 连接，所以每个连接会有一些额外的开销。除了普通的到达 XMPP 服务器的普通 TCP 连接之外，还会有两个或更多个 HTTP 连接。这意味着，连接管理器的每位用户至少需要建立 3 个 TCP 套接字。

这种开销带来的好处是独立 BOSH 连接管理器通常要比内置管理器更易于扩展。这是 BOSH 服务与 XMPP 其他部分解耦所带来的结果。扩展独立的连接管理器通常可以通过普通的扩展其他 HTTP 服务的方法来实现，而扩展内置管理器则依赖于底层 XMPP 服务器的扩展性能。

B.1.2 内置管理器

大多数 XMPP 服务器现在都有 BOSH 内置支持。这些内置的连接管理器非常易于设置，它们通常只需要对服务器的配置进行很少的修改来启用它们即可。这意味着，当程序员为他们的应用程序寻找连接管理器时，这些内置管理器将成为他们首先尝试的目标。

与独立的管理器不同的是，内置管理器不能与外部服务器建立 XMPP 连接，而只能与它们运行所在的服务器建立连接。如果所有的用户在连接管理器所在的服务器上都有账户或者所有的用户均匿名连接，那么内置管理器将是这些应用程序的理想选择。

内置管理器的开销较低，这是因为它们与 XMPP 服务器耦合在一起。不需要建立到 XMPP 服务器的 TCP 连接，这是因为连接管理器已经作为服务器的一部分在运行。内置管理器能够将 HTTP 请求直接转换成内部路由的数据包，而不需要经历额外的网络层。

通过内置管理器获得的另一项效率提升就是降低了序列化开销。通常，应用程序需要将 XML 数据保存在特殊的数据结构中从而让该数据易于操作和获取。当把 XML 写入到套接字中以传送到服务器时，必须将该数据结构序列化成实际的 XML 数据。在高性能场合中，这种序列化开销可能非常显著。独立的连接管理器必须经常性地将它们接收到的数据再次序列化以传送给外部 XMPP 服务器，而内置管理器可以避免这额外的一步。

最后，内置管理器已经针对它们对应的服务器进行了良好的测试，这是因为它们是作为一个单元进行开发的。独立的连接管理器通常会针对许多服务器进行测试，但很难企及由编写 XMPP 服务器的同一个程序员团队直接整合所带来的稳定性。

B.1.3 连接管理器权衡

每种连接管理器都有各自的长处和弱点。

1. 支持外部服务器

独立的连接管理器能够连接到 Internet 上任何可达的 XMPP 服务器，但内置的连接管理器只能连接到它们所在的服务器。

2. 设立与配置的复杂性

内置服务器易于设置，这是因为它们只需要对现有 XMPP 服务器的配置文件进行一点点改动即可。但必须对独立的服务器单独安装并配置，这让我们的工作会稍微复杂一些。

3. 可扩展性

内置服务器由于省去了额外的序列化开销，这使得它们更高效，但其扩展性取决于底层的服务器。独立的服务器通常能够像其他 Web 服务那样扩展，但它们的序列化开销稍高。

没有一种解决方案能够完美地适用于所有应用程序。只有通过仔细地思考自己的应用程序的具体需求才能做出正确的选择。

B.2 Punjab: 独立管理器

如果应用程序需要一个能够连接到任何 XMPP 服务器的独立连接管理器，那么 **Punjab** 就能够实现。**Punjab** 是一个采用 Twisted Python 网络库编写的 Python 守护进程。它经过良好的测试而且可以实现很高的性能。

B.2.1 获取 Punjab

可以从 **Punjab** 的网站 <http://code.stanziq.com/punjab> 获取 **Punjab**。它要求 Python 2.5 或更新版本以及 Twisted Python 2.5 或更新版本，但推荐使用 Twisted Python 8.1 或更新版本。

可以通过 Python 网站 <http://www.python.org> 找到 Python。如果正在运行 Linux 或使用 Mac OS X，那么您可能已经安装了 Python 的合适版本。

可以在 <http://www.twistedmatrix.com> 找到 Twisted Python。Linux 用户可能会在发行的软件包列表中找到 Twisted Python 的某个版本，而 Mac OS X 用户的操作系统中已经预先安装了一个副本。

B.2.2 安装与设置

可以采用熟悉的 Python 方式通过使用 `setup.py` 文件来安装 **Punjab**。首先，将发行中的压缩文件提取到选中的目录中。接下来，运行如下命令来安装 **Punjab**。

```
python setup.py install
```

一旦安装完毕，就可以通过为其创建 `.tac` 文件来配置 **Punjab**。`.tac` 文件包含着应用程序的配置，而 Twisted Python 的 `twistd` 命令在启动 **Punjab** 时会使用该文件。

Punjab 软件包自带了一个默认的 `punjab.tac` 文件，它配置 **Punjab** 在端口 5280 上运行，BOSH URL 为 `/xmpp-httpbind`。这个默认的配置通常已经不错了，但如果应用程序需要某些不同的配置，那么可以随意修改。当查看该文件时，用户想修改的值就一目了然了。

B.2.3 启动并测试 Punjab

通过 `twistd` 命令来启动 **Punjab**。想要将 **Punjab** 作为后台进程或守护进程启动，那么运行如下命令。

```
twistd -y punjab.tac
```

如果希望在前台运行 `Punjab`，那么可以在命令中添加参数`-n`，如下所示。

```
twistd -ny punjab.tac
```

无论选择何种方式，`Punjab` 都应该正常启动并响应端口 5280(或者在配置中指定的其他端口)。如果正在本地计算机上运行 `Punjab`，那么可以访问 `http://localhost:5280/xmpp-httpbind` 来测试 `Punjab` 是否正在运行。如果它在另一个主机上运行，那么应使用该主机的普通域名而不是 `localhost`。

如果 `Punjab` 正确运行，那么当访问配置好的 URL 时应该看到如下消息。

```
XEP-0124 - BOSH
```

注意，除非在使用 `Strophe` 的 `flXHR` 库进行跨域访问，否则需要将 `Punjab` 代理到 Web 应用程序目录树中，以遵守 JavaScript 的同源策略。

B.3 ejabberd 和 mod_http_bind：内置管理器

`ejabberd` 是目前最流行的 XMPP 服务器之一，而且毫无疑问它自带了一个 BOSH 连接管理器，从而可以通过 HTTP 连接来提供 XMPP 访问。如果已经运行 `ejabberd` 作为自己的 XMPP 服务器，那么很容易启用 BOSH 支持。

B.3.1 获取 mod_http_bind

`ejabberd` 在其发行文件中包含 `mod_http_bind` 已经有一段时间，但如果碰巧使用的版本低于这里连续使用的版本，那么可以单独从 `ejabberd` 模块页面 <http://www.ejabberd.im/ejabberd-modules> 中获取 `mod_http_bind`。

如果需要获取 `ejabberd` 本身的副本或升级旧版本，那么可以在 <http://www.ejabberd.im> 找到最新代码。

B.3.2 配置 mod_http_bind

`ejabberd` 中的模块通过 `ejabberd.cfg` 文件来启用。这些文件的内容实际上是真正的 Erlang 语法数据结构，因此它们并不像您曾经用过的大多数配置文件。如果知道查看什么内容，那么修改起来仍然非常简单。

默认的 `ejabberd.cfg` 文件已经启动了一些 Web 服务器选项，因此只需要稍作修改以添加连接管理器支持即可。

在 `ejabberd.cfg` 文件中搜索，直到找到一个如下的节为止。

```
{listen,
 [
 {5222, ejabberd_c2s, [
```

配置文件的这一节负责处理 ejabberd 将要在哪个端口上侦听连接请求。再往下几行代码之后，应该看到端口 5280 的一些配置。

```
{5280, ejabberd_http, [
    http_poll,
    web_admin
]}
```

我们将对此节稍作修改以添加对 http_bind 的支持而不是 http_poll。将该节修改成如下所示的形式。

```
{5280, ejabberd_http, [
    web_admin,
    {request_handlers, [{["xmpp-httpbind"], mod_http_bind}}]
]}
```

如果配置文件还没有包含端口 5280 的配置节，那么可以轻易地将上面的配置节添加到该文件中。只需确保在 listen 块的最后一节之后放置一个逗号，然后添加新的配置节即可。例如，看看下面的被突出显示的配置行，这里已经向 listen 块(以前没有包含 web 节)中添加一个 web 节。

```
listen,
[
%% some sections omitted
{5269, ejabberd_s2s_in, [
    {shaper, s2s_shaper},
    {max_stanza_size, 131072}
]},
{5280, ejabberd_http, [
    web_admin,
    {request_handlers, [{["xmpp-httpbind"], mod_http_bind}}]
]}
].
```

注意，向端口 5269 的配置节添加一个逗号，然后将端口 5280 的配置节(包含新的 mod_http_bind 节)添加进来。

除了修改 listen 节之外，还必须将 mod_http_bind 添加到 modules 节。搜索 modules 节，如下所示。

```
{modules
[
  {mod_adhoc, []},
```

在该节的末尾，插入 `mod_http_bind` 配置行，确保在这个新配置行的前一行的末尾放置一个逗号。

```
{mod_disco, []},
{mod_roster, []},
{mod_http_bind, []}
].
```

再次向含有 `mod_roster` 的配置行添加一个逗号，然后添加新配置行。

B.3.3 重启并测试 ejabberd

一旦更新配置使其具有 `mod_http_bind` 支持，就可以运行如下命令来重启 ejabberd 服务器。

```
ejabberdctl stop
ejabberdctl start
```

ejabberd 应该会正常启动，而且应该能够访问 `http://server.example.com:5280/xmpp-bind` 并看到一条有关 BOSH 支持的消息。将 `server.example.com` 替换成 XMPP 服务器的正确主机名。

还需要将这个 URL 代理到服务应用程序的目录树下面，除非正在使用 Strophe 的 flXHR 插件来实现跨域请求支持。

B.4 代理和安全策略

当 JavaScript 在 Web 浏览器中运行时会受到多种限制。其中之一就是同样策略，它限制 JavaScript 只能向加载应用程序的同一服务器建立 HTTP 请求。如果应用程序的 `index.html` 文件是从 `example.com` 上的 80 端口加载的，那么 JavaScript 代码就只能向同一个域和端口发送请求。

像 jQuery 这样的 AJAX 库，往往有一些部分变通措施来绕开这项限制，但这些措施只有在您需要建立的是 GET 请求而且远程服务器能够以特殊的 JSON-P 格式返回数据的情况下有用。但 BOSH 请求是通过 HTTP POST 完成的，而且因为数据是以 XML 格式收发，所以这些变通措施派不上用场。

对于这个问题的一个解决办法是利用 Flash 的跨域能力来建立 HTTP 连接，而且本书中的应用程序就是通过 Strophe 的 flXHR 插件来采用该方法的。另一种解决办法是使用反向代理来让一个指向应用程序的服务器的本地 URL 来代理远程 BOSH 服务。对于 JavaScript 代码而言，请求是发往同一服务器，但在后端，数据实际上被 Web 服务器发送到远程服务，然后响应又被转发回请求者。

B.4.1 利用 Flash 实现跨域请求

与 JavaScript 类似，Flash 也有着严格的安全策略，但与 JavaScript 不同的是，它允许向那些接受这类跨域请求的域发送跨域请求。Flash 通过读取目标域中的一个特殊文件 `crossdomain.xml` 进行这项访问许可检查。

crossdomain.xml 文件必须位于应用程序需要建立跨域请求的那个网站的根目录中。例如，如果应用程序托管在 `http://pemberley.lit` 域，而它需要访问 `http://longbourn.lit:5280/xmpp-htpbind` 的连接管理器，那么 Flash 就会搜索 `http://longbourn.lit:5280/crossdomain.xml` 文件来检查是否具有所需跨域请求的访问许可。如果找到 `crossdomain.xml` 文件而且包含适当的配置信息，那么跨域请求就会获得许可。

为了让 BOSH 连接管理器能够服务任意基于 Flash 的客户端，它需要提供如程序清单 B-1 所示的 `crossdomain.xml` 许可文件。注意，这个文件允许来自任何域的跨域请求。如果这项宽松的策略并不适于自己的应用程序，那么可以使用更严格的配置。要学习更多有关如何自定义 `crossdomain.xml` 文件的信息，请参见 http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html。



程序清单 B-1: crossdomain.xml

可从
Wrox.com
下载源代码

```
<!DOCTYPE cross-domain-policy SYSTEM
"http://www.macromedia.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <!--
    Cross domain policy file for allow everything. If you need more
    information on these, please see:
    http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html
  -->
  <site-control permitted-cross-domain-policies="all"/>
  <allow-access-from domain="*" />
  <allow-http-request-headers-from domain="*" headers="*" />
```

1. 使用 Punjab 服务 `crossdomain.xml` 文件

Punjab 的默认配置文件 `punjab.tac` 包含对静态内容服务的支持。在默认情况下，它让自己运行所在的目录下面的 `html` 子目录能够通过 HTTP 访问。

为了提供 `crossdomain.xml` 文件的访问，只需要将该文件放到 `html` 目录中即可。可以通过访问 `http://bosh.example.com:5280/crossdomain.xml` 来验证 Punjab 服务器是否在 `bosh.example.com` 的 5280 端口上运行。

2. 使用 ejabberd 服务 `crossdomain.xml` 文件

ejabberd 服务器还利用 `mod_http_fileserver` 模块对静态内容提供访问支持。可以启用该模块，然后将适当的请求处理程序放入 ejabberd 配置文件中。

首先，添加下面的 `modules` 节的配置行。

```
{mod_http_fileserver, [{docroot, "/path/to/html"}
  {content_types, [{".xml", "text/xml"}]}]}
```

`docroot` 选项指定了 `ejabberd` 寻找待服务的静态文件的路径。`content_types` 选项只将适当的 `text/xml` 内容类型添加到已知类型列表中。这是必须的，因为 `ejabberd` 在默认情况下并没有包含这种特定的映射。

既然要加载该模块，那么我们必须添加一个新的请求处理程序来告诉 `ejabberd` 如何服务以及到哪里寻找静态文件。修改配置文件的 `listen` 配置节中的 `ejabberd_http` 节，如下所示。

```
5280, ejabberd_http, [
    web_admin,
    {request_handlers, [{["xmpp-httpbind"],
        mod_http_bind},
        {"", mod_http_fileserver}]}]
```

采用粗体显示的配置行给出了新的请求处理程序，它将作为已配置文档根(即 `ejabberd web` 服务的根)。

下面只需将 `crossdomain.xml` 文件放入文档根目录中即可。一旦把该文件放入正确的位置，就可以通过访问 `http://example.com:5280/crossdomain.xml`(将 `example.com` 替换成服务器的名称)来检查是否一切就绪。

B.4.2 代理 BOSH

代理需要 Web 服务器的配合以及一些额外的配置。为了让本书的材料尽可能简单，我们使用了 Flash 插件，但有时候代理解决方案更好。例如，iPhone 的 Web 浏览器不支持 Flash，因此针对该平台编写的 XMPP 应用程序就需要使用代理 BOSH 连接管理器。

下面几节中的示例假设您已经搭建好一个可以运行的 BOSH 连接管理器。如果希望试验代理 BOSH 服务而又不必搭建 BOSH 连接管理器，那么可以使用 `http://bosh.metajack.im:5280/xmpp-httpbind` 中的 BOSH 进行测试。

1. 配置 Apache 来代理 BOSH

Apache HTTP 服务器一直是服务网页和应用程序的最常见的选择，而且它通过其 `mod_proxy` 模块提供了完善的代理支持。如果 Web 应用程序使用 Apache，那么可以为 BOSH 连接管理器添加一个代理。

我们还需要确保已经构建 `mod_proxy` 模块或者该模块可用，尽管在默认情况下它通常已经编译而且包含在内。

首先，必须允许访问远程 BOSH 服务器。可以将下面的`<Proxy>`节放在任何地方，但通常将其放在`<VirtualHost>`节内部，或放在主配置文件 `httpd.conf` 中。

```
<Proxy http://bosh.metajack.im:5280/xmpp-httpbind>
    Order allow,deny
    Allow from all
</Proxy>
```

这里使用的 BOSH URL 是 `http://bosh.metajack.im:5280/xmpp-httpbind`，这可以用于测试，但应该将其替换成自己的 BOSH 服务 URL。

一旦在代理配置中允许 BOSH URL，那么就可以将相关的 `ProxyPass` 和 `ProxyPassReverse` 配置行添加进来。

```
ProxyPass /xmpp-httpbind http://bosh.metajack.im:5280/xmpp-httpbind
ProxyPassReverse /xmpp-httpbind http://bosh.metajack.im:5280/xmpp-httpbind
```

这两行代码将本地服务器上的/xmpp-httpbind URL 代理到远程 BOSH 服务 URL。

一旦配置好 Apache，那么就可以重启 Apache 服务器(就像通常所做的那样)。可以通过访问 Apache 服务器上的/xmpp-httpbind URL 来检查新配置是否工作，应该会看到 XEP-0124 或 BOSH 的提醒。

2. 配置 nginx 来代理 BOSH

nginx(发音为“engine x”)是由 Igor Sysoev 开发的小巧快速的 Web 服务器，由于其出色的性能而被众多繁忙的 Internet 创业公司采用。nginx 还为代理资源提供了极佳的支持，而这使得它成为服务 XMPP Web 应用程序的极佳选择。

在 nginx 中设置代理非常容易。只需在相关的 `server` 块中添加一个 `location` 块即可。下面的示例添加了 `location /xmpp-httpbind`，这会代理 `http://bosh.metajack.im:5280/xmpp-httpbind` 中的 BOSH 服务。

```
server {
    listen 80;

    # more configuration..

    location /xmpp-httpbind {
        proxy_pass http://bosh.metajack.im:5280;
    }

    # more configuration..
}
```

一旦修改好配置并重启服务器，就可以通过访问 nginx 服务器上的/xmpp-httpbind 资源来测试代理设置能否工作。

3. 使用 Tape 进行本地开发

如果正在开发多个 XMPP 应用程序或者正在测试同一个应用程序的不同版本，那么不停地修改 Web 服务器配置来代理正确的 BOSH 服务就有点不现实了。为了简化这类开发工作，Tape 程序应运而生。Tape 是一个真正简单的即时 Web 服务器。

Tape 通常在一个目录中运行并通过 Web 提供该目录及其子目录中的文件的服务。除了服务静态文件之外，它还能够代理任意位于本地目录树下面的 URL。可以利用这个功能来代理 BOSH 服务以提供给 XMPP 应用程序使用，而不必搭建并设置传统的 Web 服务器。

为了获取最新的 Tape 副本，可以访问 <http://github.com/metajack/tape>。与本附录前面描述过的 Punjab 类似，Tape 依赖于 Twisted Python 框架。它不需要特殊的安装程序，只需要将 Tape 文件放在 OS X 或 Linux 的/usr/local/bin 目录中，或放在 Windows 的系统路径中的某个地方。

将当前目录修改成存放应用程序的目录并启动 Tape。

```
cd awesome_project
tape
```

在默认情况下，Tape 会通过 <http://localhost:8273>(在电话拨号盘上 8273 拼出来就是 T-A-P-E) 来服务文件。可以在命令行上使用-P 选项来轻易地指定 Tape 代理 BOSH 服务。

```
tape -P /xmpp-httpbind=http://bosh.metajack.im:5280/xmpp-httpbind
```

如果不使用 8273 端口，那么还可以使用-p 选项来修改该端口。

```
tape -p 8000 -P /xmpp-httpbind=http://bosh.metajack.im:5280/xmpp-httpbind
```

要停止运行 Tape，只需按 **Ctrl+C** 组合键或杀死该进程即可。

Tape 还支持简单的配置文件，可以将该文件放在 XMPP 应用程序的目录中。这个文件名为.taperc。下面的.taperc 示例与前面使用-p 和-P 选项的示例效果相同。

```
[server]
port = 8000

[proxies]
/xmpp-httpbind = http://bosh.metajack.im:5280/xmpp-httpbind
```

Tape 使得我们搭建一个代理正确 URL 的即时 Web 服务器的工作变得简单。在每个 XMPP 应用程序的目录中放置一个.taperc 文件，我们就可以切换到新项目并输入 tape，然后立即就可以使用正确的配置来访问应用程序。

Tape 还可用来服务借助 Flash 的 Strophe，这是因为 Flash 的安全策略要求要么针对 file:// URL 访问编译 Flash 代码，要么针对 http:// URL 访问编译代码。为了使用带有 http:// URL 的对象，该对象必须通过 http:// URL 来提供服务。如果还没有搭建好 Web 服务器，那么 Tape 可以在开发期间提供一个很好的替代品。

B.5 更多 BOSH 连接管理器

除了本附录重点提及的这些 BOSH 连接管理器之外还有更多其他管理器可用。下面列出了其他几种常见的选择，可以在相应的网站上找到更多相关信息。

- Openfire 是一款采用 Java 编写的带有 BOSH 支持的 XMPP 服务器，它的网站是 <http://www.igniterealtime.org/projects/openfire/index.jsp>

- Tigase 也是一款采用 Java 编写的带有 BOSH 支持的 XMPP 服务器, 它的网站是 <http://www.tigase.org/>
- Prosody 是一款采用 Lua 编写的轻量级的、易于使用的 XMPP 服务器, 它的网站是 <http://prosody.im/>
- JabberHTTPBind 是一款基于 Java Servlet 的独立连接管理器, 它的网站是 <http://blog.jwchat.org/jhb/>
- Rhb 是一个采用 Ruby 编写的独立连接管理器, 它的网站是 <http://rubyforge.org/projects/rhb/>