移动开发经典丛书

Professional iOS Network Programming: Connecting the Enterprise to the iPhone and iPad

iOS网络高级编程：iPhone和iPad的企业应用开发

Jack Cox
[美] Nathan Jones 著
John Szumski
张 龙 译

清华大学出版社

实现iOS应用与其他系统的无缝连接

iPhone SDK(现在称为iOS)的发布引发为iPhone创建应用的热潮。为了将iOS应用开发推向新高度，本书作者通过各种有效的方法、鲜活的示例与最佳实践实现了iOS应用与其他系统(如网络主机或其他移动设备)的无缝连接。无论是新手还是经验丰富的开发人员，都会从作者解决网络系统集成、安全与设备管理的方式中获益无穷，从而可以构建更棒、更可靠的应用。

主要内容

- 给出在客户端设备与服务器之间执行HTTP请求以及处理HTTP请求中错误的指导建议
- 分享保护网络通信以及改进请求性能的建议
- 为应用开发项目的规划与应用需求的制订打下坚实的技术基础
- 介绍高级的网络技术
- 使用Bonjour作为多个设备的应用间通信方式

作者简介

Jack Cox是CapTech Ventures公司的总监，在复杂系统设计、项目团队管理以及与业务方合作开发创新应用方面具有丰富经验。

Nathan Jones在企业级系统集成与移动开发方面具有丰富经验，包括策略定义、应用开发以及开发发布管理等。

John Szumski是一名企业软件开发人员，在中间件Web服务及各个主流平台的移动开发方面颇具经验。

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

ISBN 978-7-302-36411-5

9 787302 364115 >

定价：49.80元

移动开发经典丛书

iOS 网络高级编程：
iPhone 和 iPad 的企业应用开发

[美] Jack Cox
Nathan Jones
John Szumski
著

张 龙
译

清华大学出版社

北 京

Jack Cox, Nathan Jones, John Szumski
Professional iOS Network Programming: Connecting the Enterprise to the iPhone and iPad
EISBN: 978-1-118-36240-2

图书在版编目(CIP)数据

iOS 网络高级编程：iPhone 和 iPad 的企业应用开发/ (美)考克斯(Cox, J.)，(美)琼斯(Jones, N.)，(美)舒姆斯基(Szumski, J.) 著；张龙 译. —北京：清华大学出版社，2014
(移动开发经典丛书)
书名原文：Professional iOS Network Programming：Connecting the Enterprise to the iPhone and iPad
ISBN 978-7-302-36411-5

Ⅰ. ①i…　Ⅱ. ①考…　②琼…　③舒…　④张…　Ⅲ. ①移动终端－应用程序－程序设计　Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字(2014)第 098145 号

责任编辑：王　军　李维杰
装帧设计：孔祥峰
责任校对：成凤进
责任印制：王静怡
出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
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 装 者：三河市金元印装有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　印　　张：20　　字　　数：487 千字
版　　次：2014 年 7 月第 1 版　　印　　次：2014 年 7 月第 1 次印刷
印　　数：1～3200
定　　价：49.80 元

产品编号：053236-01

移动开发经典丛书

iOS 网络高级编程：
iPhone 和 iPad 的企业应用开发

[美] Jack Cox
Nathan Jones
John Szumski
著

张　龙
译

清华大学出版社

北　京

译 者 序

科技发展日新月异，以 iPhone 为标志的移动开发浪潮已经席卷全球，每天都有不计其数的 iOS 开发者向 App Store 提交新的应用。同时，学习 iOS 开发的技术人员也与日俱增，这其中有相当一部分技术人员是从其他平台转过来的。因此，优秀学习资源的价值也随之彰显出来。目前，市场上关于 iOS 开发的技术图书已经达到汗牛充栋的程度，不过令人略感遗憾的是，很多 iOS 开发图书介绍的都是一些入门知识，学习者在学习完后依然无法开发出高质量的应用。本书正是在这个时代背景下推出的，与那些泛泛而谈的图书不同的是，本书专注于 iOS 开发的一个领域——网络。

绝大多数 iOS 应用都依赖于网络，通过网络来访问服务器并与之交互。那么，该如何设计良好的服务、如何实现设备与服务器的无缝连接、如何分析通信传输的数据、如何进行错误处理、如何优化通信的性能、如何进行测试、如何实现应用间通信，这都是摆在每一个开发人员面前的问题。本书则对上述问题给出了详尽而又深入的解释，同时辅以相关代码示例。这使得学习者能够又快又好地掌握 iOS 网络的相关知识，并迅速应用到自己的应用当中。

此外，本书的一大特色就是深入，除了常见的 iOS 网络通信相关技术外，本书还详尽介绍了底层网络，这是构成高层网络功能的基石。掌握这些底层知识将有助于开发者更好地理解上层网络功能及使用方式，并选择最适合自己的网络技术。

值得一提的是，有大量的示例贯穿于本书。这些示例都是可以直接运行的，学习者完全可以先将示例运行出来，对相关技术有一个感性的认识后再来深入学习，这样可以做到事半功倍。

除了 iOS 网络技术的介绍外，本书还针对每一项技术给出了最佳实践，这些最佳实践可以帮助学习者构建出高质量、高性能的 iOS 网络应用。另外，书中的每一章都专门介绍 iOS 网络功能的一个主题，学习者也可以选择自己感兴趣的主题进行深入研究。

译者从事 Android 与 iOS 相关开发工作已经有 3 年多的时间，期间积累了不少经验，深谙移动开发的精髓。曾翻译过《Android Web 应用高级编程》、《iPhone SDK 编程入门经典：使用 Objective-C》、《Xcode 3 高级编程》、《iPhone 游戏开发》、《复杂性思考》、《设计原本》等书籍。译者目前担任 InfoQ 中文站翻译团队编辑、满江红开放技术研究组织成员，同时拥有 6 年以上的培训讲师经历。

感谢清华大学出版社的编辑们，你们的专业与认真都给我留下了深刻的印象，也非常感谢你们的理解与包容。翻译技术书籍是一项艰苦的劳动，在这里我要将我最真挚的谢意送给我的妻子张明辉，正是她无微不至的关怀与照顾才能让我忘却生活中的琐事而专心于翻译。本书的翻译工作还得到了张彤辉、张淑华、焦伟、张淑贤、李志芹、张兴国、马元贺、王辉、王凯、单会明、周锐、王冠、高鹏等的帮助，在此一并表示感谢。

对于译者来说，能将英文转换为中文并给读者带来切实的帮助是我最大的荣幸。因此，若您在阅读过程中发现问题请不吝赐教。由于译者水平有限，失误和遗漏之处在所难免，恳请读者批评指正。敬请广大读者提供反馈意见，读者可以将意见反馈到zhanglong217@163.com，我会仔细查阅读者发来的每一封邮件，以求进一步提高今后译著的质量。我的博客是 http://blog.csdn.net/ricohzhanglong，新浪微博是 http://weibo.com/fengzhongye，欢迎来访。

作 者 简 介

Jack Cox 是软件开发者、系统架构师以及 CapTech Ventures 公司的总监，负责公司的移动软件开发。他有着 30 年的各种业务软件开发经验、涉足过 3 家创业公司、拥有多项专利，并且经常在各种专业小组中发表演讲。他拥有位于印第安纳州阿普兰的泰勒大学的计算机科学学位。Jack 现在与妻子和孩子居住在弗吉尼亚州里士满。可以在 Twitter 上通过@jcox_mobile 联系到他。

Nathan Jones 是一位在 iOS 方面颇具经验的软件工程师，同时也有着丰富的移动 Web 技术开发经验。他的职业生涯从企业软件咨询开始，当 Apple 宣布可以为 iPhone 开发第三方应用时，他开始了对移动开发的探索。他毕业于弗吉尼亚理工学院和州立大学，拥有商业信息技术科学学士学位，主要关注于决策支持系统。目前，Nathan 与妻子 Jennifer 和儿子 Bryson 居住在弗吉尼亚州里士满。在工作、写作以及陪儿子玩耍的间隙，他喜欢打高尔夫，同时还是一位跑者。可以在 Twitter 上通过@nathanhjones 联系到他。

John Szumski 是软件工程师和移动咨询顾问，在 iOS、Android 与移动 Web 平台方面拥有丰富的经验。他曾为多家财富 500 强公司做过关于用户体验与技术设计方面的咨询。他以优异的成绩毕业于弗吉尼亚州夏洛茨维尔市的弗吉尼亚大学，拥有计算机科学学士学位。目前，John 与未婚妻居住在弗吉尼亚州里士满。可以在 Twitter 上通过@jszumski 联系到他。

技术编辑简介

Jonathan Tang 是一位高级开发者，现就职于 CapTech Consulting，专注于移动应用开发。他拥有 10 多年的开发经验，包括编写触摸屏界面、医疗设备以及 iOS 移动应用。在加入 CapTech 之前，John 在一家专注于医疗机器人技术的创业公司担任初级软件工程师。John 拥有约翰霍普金斯大学生物工程学士学位和乔治·华盛顿大学电子工程硕士学位。

致　谢

我要感谢 CapTech Ventures 公司的主管、管理层以及同事，特别是 Vinnie Schoenfelder，感谢你们鼓励并支持我编写这本书。将特别的感谢送给 Nathan Jones 与 John Szumski，感谢你们在这次旅程中的通力协作，使得我们能够完成人生的第一本书。我代表 Nathan、John 及我自己向 Wiley 的 Carol Long 和 Victoria Swider 表示深深的感谢，感谢对我们的容忍以及回答我们那些幼稚的问题。

我要对我的妻子和家人表示深深的感谢，感谢你们容忍我每天晚上和周末都将时间放在了写书上，谢谢你们支持我完成这个梦想。

最重要的是，我要感谢我的救星 Jesus Christ，他给予了我巨大的支持。要是没有他，我肯定一无所成。

—Jack Cox

我要感谢亲爱的妻子 Jennifer 和儿子 Bryson，感谢在我编写这本书时你们给予我的持续的支持与耐心。有时，我看 Xcode 的时间比看你们的时间还要长，那些夜晚与周末对于你们来说是很难熬的，我都看在眼里，谢谢你们。我还要感谢我的父母，感谢你们在这个过程中对我的鼓励，特别要感谢我的父亲，是你教会了我编写第一个程序，这为我的未来播下了种子。到现在我还保留着那张软盘，不过却没有驱动能够读它了。

—Nathan Jones

我要感谢我美丽的未婚妻 Caroline。感谢在我每天晚上写书和编辑时对我的理解与支持。我还要感谢我的家人，感谢你们在整个出版过程中对我的鼓励。

—John Szumski

前　言

现在，iPhone 与 iPad 在我们的生活与工作中已经无所不在，我们也越来越依赖于它们与 Internet 上的主机或房间里的其他手机无缝且正确的交互能力。本书介绍了实现这种连接功能的各种方法，并通过大量示例与最佳实践介绍了每一种方法的实现原理。

随着 iPhone SDK(现在叫做 iOS)的发布，很多经验丰富以及新手开发者都开始为 iPhone 开发应用。在这个潮流下，市场上出现了很多介绍如何为 iPhone 开发应用的图书。其中大多数图书都关注于如何开发用户界面。本书则不是这样，本书的焦点放在了连接 iOS 应用与其他系统(网络主机或是其他移动设备)的方法与最佳实践。如果你对学习 iOS 开发环境投入了时间和精力，并且现在想要通过经过实践证明的设计模式来构建企业级的应用，那么这本书就非常适合你。

在过去的 15 年间，网站开发在企业 IT 部门中处于主宰地位。人们的 HTML、CSS 与 JavaScript 技能水平不断增长，但小型设备之间的互联能力却呈现下降的趋势。由于移动软件开发在过去 4 年间呈现出爆发性的增长，开发社区(经验丰富的开发者与新手)要重新学习小型设备之间互联的实践。

作为为众多大型各户服务过的专业 iOS 开发者，本书的作者发现相对于设计、开发与验证应用来说，开发和打磨应用的交互部分会很花时间。此外，他们还发现现有的图书并没有很好地解决 iOS 开发的这个非常重要的方面。鉴于此，本书能够帮助新手与专业开发者构建出更好、更可靠的应用。

本书读者对象

企业 iOS 开发者(包括公司与组织中的开发者)会发现本书是非常有价值的资源，它提供了可运行的示例与指南来实现 iOS 网络应用与企业服务器的互联。本书所介绍的网络技术为每个编写 iOS 应用的开发者提供了强力武器。

从其他平台转向 iOS 的新手开发者可以通过本书全面了解到 iOS 的功能。此外，与这些功能对应的可运行的示例为你所开发的应用提供了基础网络特性。开发者应该具备一定的 Objective-C、Xcode 与 iOS 应用开发基础。

对于那些企业系统或是应用架构师来说，如果他们的高层设计包含了跨越多个企业系统的移动设备，那就会觉得本书是非常有价值的资源，可以从中了解到 iOS 设备强大的网络功能。第 1~5 章最适合企业架构师阅读。

项目技术经理与分析师可以通过本书了解到一些技术基础，从而规划应用开发项目并设定应用需求。第 1~5 章以及每一子章节的介绍性部分对于项目经理和分析师来说颇具价值。

对于技术读者来说，本书为新手提供了全新的想法，可以在应用中加入一些引人注目的特性。由于本书是从企业开发者的视角编写的，因此应用示例适合于传统的商业组织和应用。示例并没有涉及如何编写游戏；相反，这些示例都专注于企业中最为常见的那些任务。与休闲活动相关的一些网络技术(如点对点网络)在企业中也占据着一席之地，可以为移动设备开启新的、有价值的使用场景。

本书内容

本书重点关注于运行在 Apple 操作系统(iPhone、iPad 与 iPod，统称 iOS)之上的应用的网络编程技术，主要介绍了如下主题：

- 在客户端设备与服务器之间执行 HTTP 请求
- 管理客户端设备与服务器之间的数据负载
- 处理 HTTP 请求中的错误
- 保护网络通信
- 改进网络通信的性能
- 执行 Socket 层的通信
- 实现推送通知
- 单个设备上的应用间通信
- 多个设备上的应用间通信

本书的所有示例应用与代码片段都针对 iOS 5.0+编写。作者选择将重点放在 iOS 5+的原因在于 iOS 用户更倾向于快速升级；因此，早期 iOS 安装版本的基数会很小。其他移动操作系统对于新 OS 版本的使用率则慢很多，这是因为每个版本都必须经过无线运营商的审批，这延缓了它们的发布。

本书提供的服务器端代码示例是用 PHP 与 Perl 开发的，运行在 Apache 上。之所以选择这些技术是因为它们在 Mac OS X 上可以立刻使用，这也是运行 iOS 开发环境所必需的。

本书的组织结构

本书分为 4 个部分，每一部分都涵盖了 iOS 网络编程领域中的一个广泛主题。这几个部分涉及从对移动应用架构的高层次介绍到应用间通信的具体协议与解决方案，同时深入介绍了应用与服务器间通信的最为流行的方式。

第 I 部分：理解 iOS 与企业网络

大多数读者应该从这一部分开始阅读。这一部分从高层次概览了 iOS 网络以及针对移动网络架构的最佳实践。

第 1 章：iOS 网络功能介绍——本章回顾了用于将设备连接到服务器或是其他设备的网络编程基础知识以及 iOS 提供的 API。

第 2 章：设计服务架构——本章介绍了部署设备友好的网络应用的架构模式。

第 II 部分：HTTP 请求：iOS 网络功能

这部分深入介绍了 iOS 设备与服务器通信所需的最常见的设施。

第 3 章：构建请求——本章介绍了从 iOS 应用构建请求的方式，同时提供了使用 URL 加载 API 的代码示例。

第 4 章：生成与解析负载——本章介绍了编码 iOS 应用与服务器之间传递的信息的最为常见的方式，同时提供了 XML、JSON 以及 HTML 负载管理的代码示例。

第 5 章：错误处理——本章介绍了 HTTP 请求与响应中的错误处理。

第III部分：高级网络技术

这部分包含 5 章，重点介绍了 iOS 开发者可以使用的高级网络技术。

第 6 章：保护网络传输——本章介绍了除了基本的 SSL 通信外保护网络传输的方式，并提供了客户端与服务器端证书验证的示例。

第 7 章：优化请求性能——本章介绍了改进网络通信性能的几种方式。

第 8 章：底层网络——本章介绍了从 iOS 应用中通过底层网络 API 执行 Socket 与数据包通信的方式。

第 9 章：测试与操纵网络流量——本章介绍了拦截与修改设备和服务器间通信的方式，从而可以实现应用诊断与质量保证。

第 10 章：使用推送通知——本章介绍了如何使用推送通知实现服务器与应用间的异步通信。

第Ⅳ部分：应用间网络通信

这一部分包含 3 章内容，介绍了如何实现同一台设备上以及不同设备上的应用间通信。

第 11 章：应用间通信——本章介绍了同一台设备上应用间通信的各种方式。

第 12 章：使用 Game Kit 实现设备间通信——本章介绍了如何通过 Game Kit 实现设备间通信，这一章的目标并不在游戏，Game Kit 能够实现的特性有很多，本章并未介绍全部。

第 13 章：使用 Bonjour 实现自组织网络——最后一章介绍了 Bonjour，并通过它实现多台设备上的应用间通信。

学习本书的前提

要想从本书获得最大的收益，你应该对 iOS 编程有基本的理解，比如基本的 Xcode 使用、如何将应用部署到设备上等。要想运行示例应用，你需要如下软件与硬件：

- Apple Mac 电脑，安装有 OS X Lion(10.7)或更高版本
- XCode 4.3.2 或更高版本
- 一台 iOS 设备，iPhone 3GS 或更高版本，iPad 或 iPod Touch，安装 iOS 5.0 或更高版本
- 一个 Apple 开发者账号，地址是 https://developer.apple.com/programs/register/

源代码

读者在学习本书中的示例时，既可以手动输入所有的代码，也可以使用本书附带的源代码文件。本书使用的所有源代码都可以从站点 http://www.wrox.com/ 或 www.tupwk.com.cn/ downpage 上下载。登录到站点 http://www.wrox.com/，使用 Search 工具或使用书名列表就可以找到本书。接着单击本书细目页面上的 Download Code 链接，就可以获得所有源代码。

提示：由于许多图书的书名都很类似，所以按 ISBN 进行搜索是最简单的，本书英文版的 ISBN 是 978-1-118-31445-6。

可以将本书的所有源代码作为一个文件下载(可根据语言来选择版本：C#或Visual Basic)，然后用喜欢的解压缩工具对其进行解压缩。提取源代码时，请确保维持作为代码下载一部分的原始文件夹结构。不同的解压缩工具对这个功能有着不同的名称，不过应尽可能寻找一个像User Folder Names或Maintain Directory Structure这样的功能。从下载的代码中提取了文件之后，最后应有一个名为Source的文件夹以及一个名为Resources的文件夹。然后在C盘的根目录位置创建一个新文件夹，命名为BegASPNET，并将Source和Resources文件夹移到这个新文件夹中，最后得到类似 C:\BegASPNET\Source 和 C:\BegASPNET\Resources这样的文件夹。Source文件夹中包含本书 19 章中每一章的源代码文件，以及Planet Wrox网站的最终版本。Resources文件夹包含本书的一些练习中所需要的文件。如果一切正常，最后应看到如图 0-1 所示的结构。

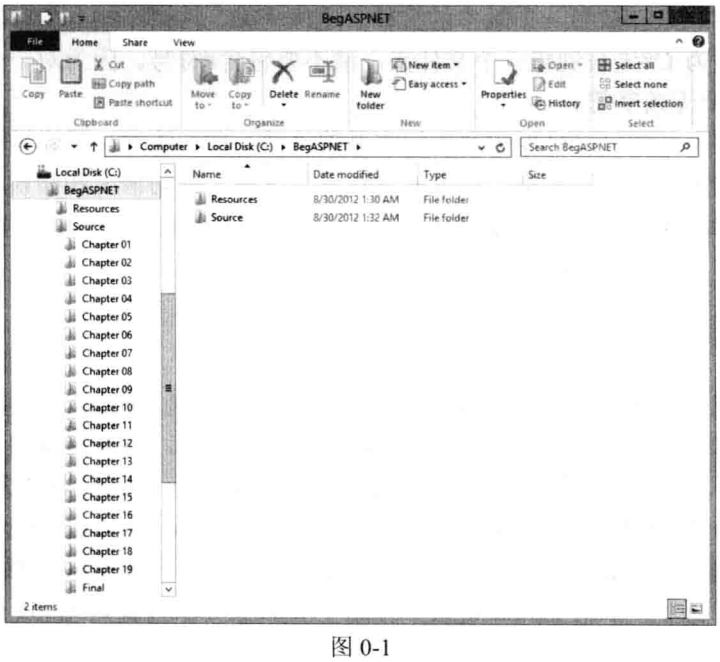


图 0-1

在以后的章节中，将在 C:\BegASPNET 文件夹中创建名为 Site 和 Release 的文件夹，从而文件夹结构将如图 0-2 所示。

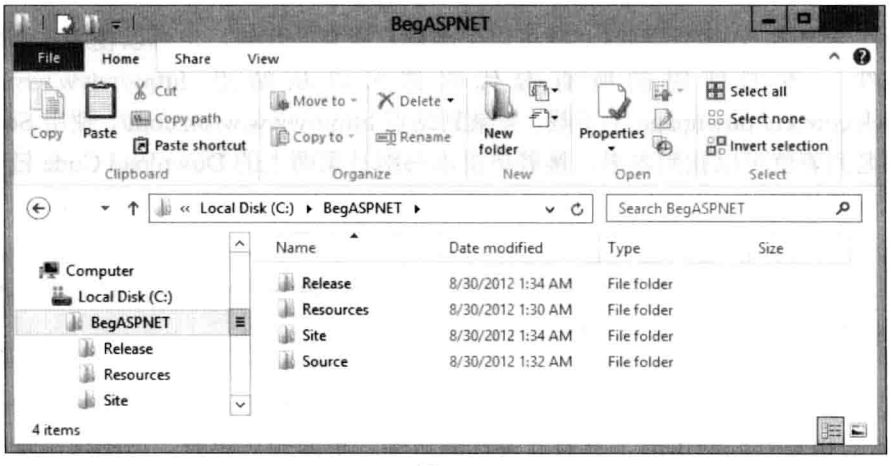


图 0-2

Site 文件夹包含本书将要构建的站点，而 Release 文件夹将包含本书末尾的站点的最终版本。每当做本书的一些练习受阻时，都可以打开 Source 文件夹查看一切最后应是什么样子。

如果要为特定章节运行站点来看看它是如何工作的，一定要在 Visual Studio 中打开那一章的文件夹作为一个网站。因此应直接打开诸如 C:\BegASPNET\Source\ Chapter 12 这样的文件夹，而不是打开它的父文件夹 C:\BegASPNET\Source。

如果想要使用两种编程语言完成操作，则创建第二个文件夹 C:\BegASPNETVB 或 C:\BegASPNETCS 来存放另一个版本的文件。这样一来，这两个站点就可以共存而不产生冲突。如果专门为 C#语言创建一个文件夹，请不要包含#符号。因为对一个网站来说，路径名中的#是一个无效字符。

坚持采用这个结构可以确保顺利执行本书的“试一试”练习。错误地混合或嵌套这些文件夹会使练习的完成变得困难，还可能导致发生预料之外的情况和错误。每当遇到本书中没有解释的问题或错误时，请确保站点结构仍然与这里提出的结构紧密相关。

勘误表

尽管我们已经尽了最大的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，如果你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走入误区，当然，这还有助于提供更高质量的信息。

要在网站上找到本书英文版的勘误表，可以登录 www.wrox.com/remtitle.cgi?isbn=1118311809，或者访问 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 Errata 链接。在这个页面上可以查看到 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

www.wrox.com/misc-pages/ booklist.shtml。

如果你在勘误表上没有找到错误，那么可以到 www.wrox.com/contact/techsupport.shtml 上，完成上面的表格，并把找到的错误发送给我们。我们将会核查这些信息，如果无误的话，会把它放置到本书的勘误表中，并在本书的后续版本中更正这些问题。

p2p.wrox.com

要与作者和同行讨论，请加入 p2p.wrox.com 上的 P2P 论坛。这个论坛是一个基于 Web 的系统，便于你张贴与 Wrox 图书相关的消息和相关技术，与其他读者和技术用户交流心得。该论坛提供了订阅功能，当论坛上有新的消息时，它可以给你传送感兴趣的论题。Wrox 作者、编辑和其他业界专家和读者都会到这个论坛上来探讨问题。

在 http://p2p.wrox.com 上，有许多不同的论坛，它们不仅有助于阅读本书，还有助于开发自己的应用程序。要加入论坛，可以遵循下面的步骤：

(1) 进入 p2p.wrox.com，单击 Register 链接。

(2) 阅读使用协议，并单击 Agree 按钮。

(3) 填写加入该论坛所需要的信息和自己希望提供的其他信息，并单击 Submit 按钮。

(4) 你会收到一封电子邮件，其中的信息描述了如何验证账户和完成加入过程。

不加入 P2P 也可以阅读论坛上的消息，但要张贴自己的消息，就必须加入该论坛。

加入论坛后，就可以张贴新消息，回复其他用户张贴的消息。可以随时在 Web 上阅读消息。如果要让该网站给自己发送特定论坛中的消息，可以单击论坛列表中该论坛名旁边的 Subscribe to this Forum 图标。

关于使用 Wrox P2P 的更多信息，可阅读 P2P FAQ，了解论坛软件的工作情况以及 P2P 和 Wrox 图书的许多常见问题的解答。要阅读 FAQ，可以在任意 P2P 页面上单击 FAQ 链接。

目　录

第Ⅰ部分　理解 iOS 与企业网络

第 1 章　iOS 网络功能介绍……3
1.1　理解网络框架……3
1.2　iOS 网络 API……4
1.2.1　NSURLConnection……4
1.2.2　Game Kit……5
1.2.3　Bonjour……5
1.2.4　NSStream……5
1.2.5　CFNetwork……6
1.2.6　BSD socket……6
1.3　运行循环……6
1.4　小结……8

第 2 章　设计服务架构……9
2.1　远程门面模式……10
2.1.1　门面服务示例……12
2.1.2　门面客户端示例……14
2.2　服务版本化……17
2.2.1　版本化服务示例……18
2.2.2　使用版本化服务的客户端示例……18
2.3　服务定位器……20
2.4　小结……24

第Ⅱ部分　HTTP 请求：iOS 网络功能

第 3 章　构建请求……27
3.1　HTTP 介绍……28
3.2　理解 HTTP 请求与响应……29
3.2.1　URL 结构……30
3.2.2　请求内容……31
3.2.3　响应内容……33
3.3　高层 iOS HTTP API……34
3.3.1　所有请求类型共用的对象……34
3.3.2　同步请求……38
3.3.3　队列式异步请求……41
3.3.4　异步请求……43
3.4　高级 HTTP 操作……51
3.4.1　使用请求方法……51
3.4.2　操纵 Cookie……53
3.4.3　头信息操作进阶……58
3.5　小结……61

第 4 章　生成与解析负载……63
4.1　Web Service 协议与风格……64
4.1.1　简单对象访问协议……64
4.1.2　表述性状态转移……65
4.1.3　选择一种方式……66
4.2　负载……67
4.2.1　负载数据格式简介……67
4.2.2　解析响应负载……70
4.2.3　生成请求负载……83
4.3　小结……90

第 5 章　错误处理……91
5.1　理解错误源……91
5.1.1　操作系统错误……93
5.1.2　HTTP 错误……98
5.1.3　应用错误……99
5.2　错误处理的经验法则……101
5.2.1　在接口契约中处理错误……101
5.2.2　错误状态可能不正确……101
5.2.3　验证负载……101
5.2.4　分离错误与正常的业务状况……102

5.2.5 总是检查 HTTP 状态 …… 102
5.2.6 总是检查 NSError 值 …… 102
5.2.7 使用一致的方法来处理错误 …… 102
5.2.8 总是设置超时时间 …… 102
5.3 优雅地处理网络错误 …… 102
5.3.1 设计模式介绍 …… 103
5.3.2 指挥调度模式示例 …… 107
5.4 小结 …… 112

第Ⅲ部分 高级网络技术

第 6 章 保护网络传输 …… 115
6.1 验证服务器通信 …… 116
6.2 HTTP 认证 …… 120
6.2.1 HTTP Basic、HTTP Digest 与 NTLM 认证 …… 120
6.2.2 客户端证书认证 …… 122
6.3 使用哈希与加密确保消息完整性 …… 126
6.3.1 哈希 …… 127
6.3.2 消息认证码 …… 130
6.3.3 加密 …… 134
6.4 在设备上安全地存储认证信息 …… 145
6.5 小结 …… 148

第 7 章 优化请求性能 …… 149
7.1 度量网络性能 …… 149
7.1.1 网络带宽 …… 150
7.1.2 网络延迟 …… 151
7.1.3 设备电量 …… 152
7.2 优化网络操作 …… 153
7.2.1 减少请求带宽 …… 153
7.2.2 降低请求延迟 …… 159
7.2.3 避免网络请求 …… 160
7.3 小结 …… 163

第 8 章 底层网络 …… 165
8.1 BSD Socket …… 165
8.1.1 配置 Socket 服务器 …… 167
8.1.2 Socket 客户端连接 …… 167
8.2 CFNetwork …… 172
8.3 NSStream …… 176
8.4 小结 …… 180

第 9 章 测试与操纵网络流量 …… 181
9.1 观测网络流量 …… 182
9.1.1 嗅探硬件 …… 182
9.1.2 嗅探软件 …… 183
9.2 操纵网络流量 …… 190
9.2.1 配置 Charles …… 192
9.2.2 HTTP 断点 …… 194
9.2.3 重写规则 …… 196
9.3 模拟实际的网络状况 …… 198
9.4 小结 …… 200

第 10 章 使用推送通知 …… 201
10.1 调度本地通知 …… 202
10.1.1 创建本地通知 …… 202
10.1.2 取消本地通知 …… 205
10.1.3 处理本地通知的到达 …… 207
10.2 注册并响应远程通知 …… 210
10.2.1 配置远程通知 …… 211
10.2.2 注册远程通知 …… 217
10.2.3 远程通知负载 …… 222
10.2.4 发送远程通知 …… 223
10.2.5 响应远程通知 …… 227
10.3 理解通知最佳实践 …… 231
10.4 小结 …… 232

第Ⅳ部分 应用间网络通信

第 11 章 应用间通信 …… 235
11.1 URL 方案 …… 235

11.1.1 实现自定义的URL方案……236
11.1.2 感知其他应用的存在……239
11.1.3 高级通信……240
11.2 共享钥匙串……244
11.2.1 企业SSO……244
11.2.2 检测应用之前的安装……250
11.3 小结……252

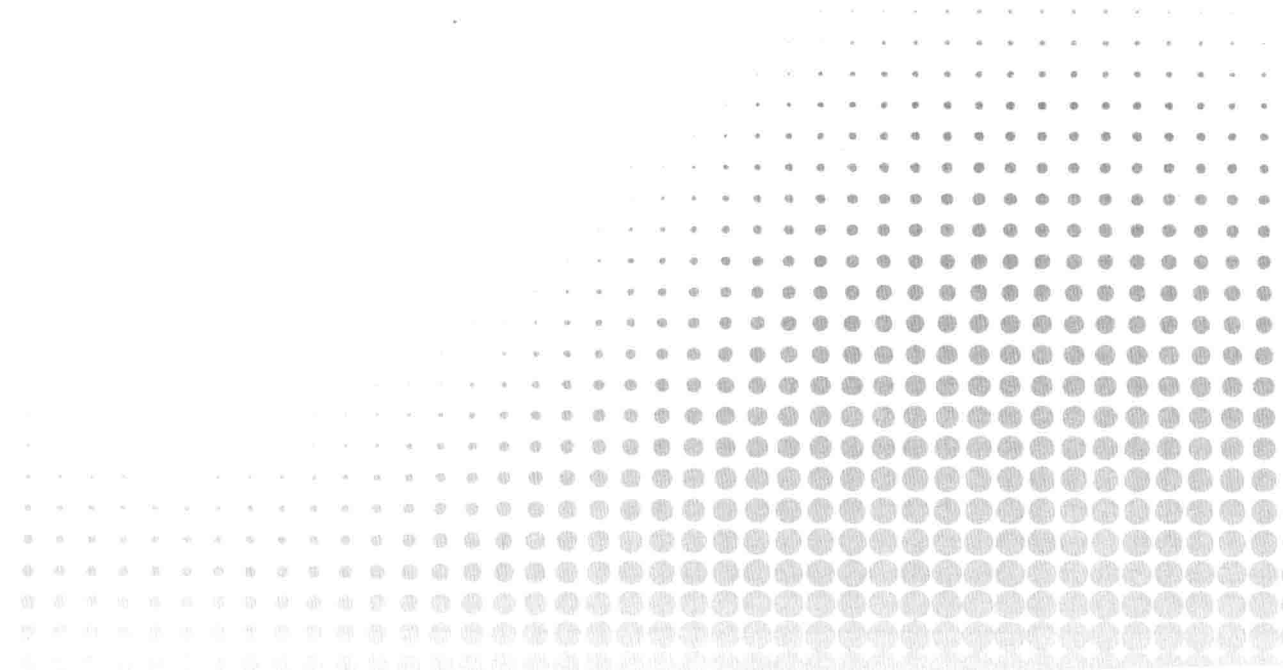
第12章 使用Game Kit实现设备间通信……253
12.1 Game Kit基础……254
12.2 点对点网络……257
12.2.1 连接到会话……257
12.2.2 向端点发送数据……260
12.3 客户端-服务器通信……264
12.4 小结……266

第13章 使用Bonjour实现自组织网络……267
13.1 zeroconf概览……268
13.1.1 寻址……268
13.1.2 解析……268
13.1.3 探测……269
13.2 Bonjour概览……270
13.2.1 发布服务……270
13.2.2 浏览服务……275
13.2.3 解析服务……278
13.2.4 与服务进行通信……280
13.3 实现基于Bonjour的应用……284
13.3.1 员工应用……285
13.3.2 顾客应用……293
13.4 小结……302

第 I 部分

理解iOS与企业网络

- 第 1 章　iOS 网络功能介绍
- 第 2 章　设计服务架构



第 1 章

iOS 网络功能介绍

本章内容：

- 理解 iOS 网络框架
- 面向开发者的关键网络 API
- 高效使用应用的运行循环

优秀的 iOS 应用需要简单、直观的用户界面。与之类似，与各种 Web Service 通信的优秀应用也需要一个良好架构的网络层。应用的架构必须具备适应需求变化的灵活性，以及能够优雅处理不断变化的网络情况的能力，同时又要保持着能够实现适当的可维护性与可伸缩性的核心设计原则。

在设计移动应用的架构时，必须精通一些核心概念，如运行循环、各种网络 API，以及如何将这些 API 与运行循环集成起来以创建响应式的网络应用框架。本章将会详细介绍运行循环以及如何在应用中高效使用它们。此外，本章还会概览关键 API 及其应用场景。

1.1 理解网络框架

在开始开发与网络交互的 iOS 应用前，需要理解 Objective-C 中网络层的组织形式，如图 1-1 所示。

每个 iOS 应用都位于某个网络框架栈之上，网络框架栈由 4 层构成。最上层是 Cocoa 层，包含了用于 URL 加载的 Objective-C API、Bonjour 与 Game Kit。Cocoa 层的下面是 Core Foundation 层，这是一套 C API，其中包含了 CFNetwork，这是大多数应用级别的网络代码的基础。CFNetwork 在 CFStream 与 CFSocket 之上提供了一个简单的网络接口。这两个类是针对 BSD socket 的轻量级封装，后者则形成了最下面的层，与无线硬件最为接近。BSD

socket 严格使用 C 来实现，向开发者提供了与远程设备或服务器进行通信的完全控制能力。

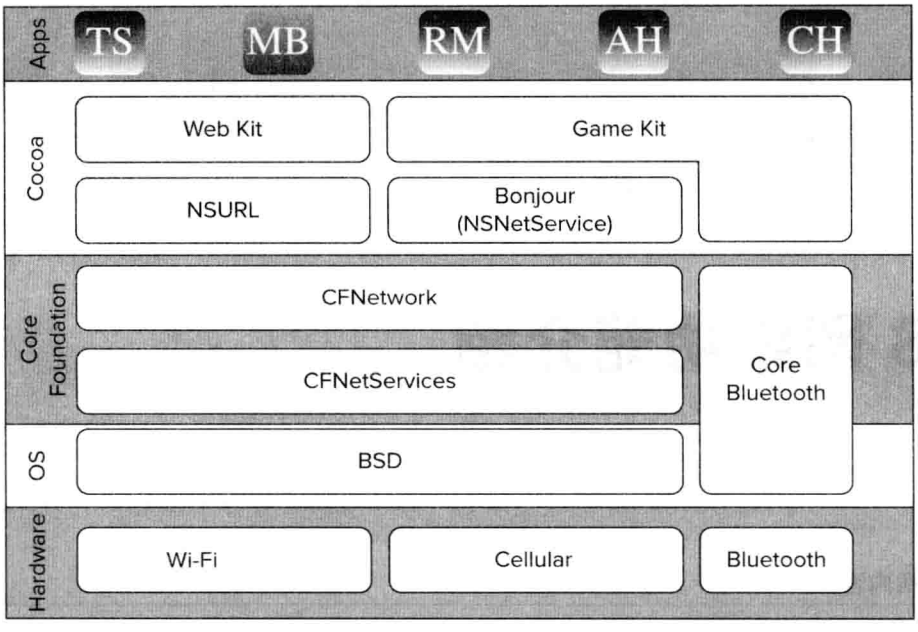


图 1-1

沿着框架栈向下移动每一层，你都会获得更为严密的控制，但却失去了上一层提供的易用性与抽象能力。虽然有时是可以的，不过 Apple 建议我们还是要位于 CFNetwork 层及其之上。BSD 层的原始 socket 无法访问系统范围的 VPN，也无法激活 Wi-Fi 和蜂窝无线电，而这些 CFNetwork 已经帮我们处理好了。

在开始设计应用的网络层之前，需要理解可用的各种 API 以及该如何使用它们。下一节将会介绍关键的 iOS 网络框架，并且简要介绍它们的使用方式。后面的章节将会对这里介绍的每一个 API 进行详细说明。

1.2　iOS 网络 API

框架栈的每一层都提供了一套关键 API，为开发者提供了各种功能与控制。每一层都比下一层提供了更高的抽象(参见图 1-1)。然而，这种抽象的代价就是丧失了某些控制。本节将会概览 iOS 中的关键 API 以及使用它们时的注意事项。

1.2.1　NSURLConnection

NSURLConnection 是一个 Cocoa 级别的 API，它提供了一个简单的方法来加载 URL 请求，可以与 Web Service 交互、获取图片或视频，或者只是简单地得到一个格式化的 HTML 文档。它构建在 NSStream 之上，并且在设计时针对如下 4 个常见的 URI 模式进行了优化支持：文件、HTTP、HTTPS 与 FTP。虽然 NSURLConnection 限制了你所能使用的协议，

但它对缓存读写的很多底层工作进行了抽象，包括对认证的内建支持，并且提供了一个健壮的缓存引擎。

NSURLConnection 接口中的内容并不多，它主要依赖于 NSURLConnectionDelegate 协议，应用可以凭借后者介入到连接生命周期的很多点上。在默认情况下，NSURLConnection 请求是异步的；不过有一个便捷方法可以发送同步请求。同步请求会阻塞调用线程，因此你也需要根据这一点来设计应用。第 3 章“构建请求”会详细介绍 NSURLConnection，还会介绍大量示例。

1.2.2　Game Kit

Game Kit 的核心功能在于为 iOS 应用提供了另一个点到点的网络选项。在传统的网络配置中，Game Kit 构建在 Bonjour 之上；然而，Game Kit 并不需要网络基础设施就能使用。它能创建自组(ad-hoc)的 Bluetooth Personal Area Networks(PAN)，这样在极少或是没有基础设施的地方，它就是非常棒的网络候选者了。

在创建网络时，Game Kit 只需要会话标识符、显示名与连接模式即可。不需要 socket 配置或是任何其他底层网络就可以实现连接点之间的通信。Game Kit 通过 GKSessionDelegate 协议进行通信。第 12 章“使用 Game Kit 实现设备间通信”会介绍如何将 Game Kit 集成到应用中。

1.2.3　Bonjour

Bonjour 是 Apple 对零配置(zeroconf)网络的实现。它提供了一种机制，可以检测并与网络中的设备或服务进行连接，同时无须了解设备的网络地址。Bonjour 通过名字、服务类型与域这个元组来引用服务。它对多播 DNS(mDNS)与基于 DNS 的服务探测(DNS-SD)所要求的底层网络进行了抽象。

在 Cocoa 层，NSNetService API 提供了一个接口，用于发布和解析 Bonjour 服务的地址信息。可以通过 NSNetServiceBrowser API 探测网络上可用的服务。发布 Bonjour 服务(甚至是使用 Cocoa 层的 API)需要理解 Core Foundation 才能配置好通信所需的 socket。第 13 章“使用 Bonjour 实现自组织网络”将会详细介绍零配置网络、Bonjour，此外还会通过示例介绍如何实现基于 Bonjour 的服务。

1.2.4　NSStream

NSStream 是一个 Cocoa 级别的 API，构建在 CFNetwork 之上，作为 NSURLConnection 的基础，旨在完成一些底层的网络任务。类似于 NSURLConnection，NSStream 提供了一种机制；用以与远程服务器或本地文件进行通信。不过，可以通过 NSStream 在诸如 telnet 或 SMTP 等 NSURLConnection 不支持的协议之上进行通信。

NSStream 提供的额外控制是有代价的。它并没有提供对处理 HTTP/S 响应状态码或认证的内建支持。它所发出与接收的数据都位于 C 缓冲中，Objective-C 开发者对此可能不太熟悉。它还无法管理多个外发请求，需要子类化才能添加这个特性。NSStream 是异步的，

通过 NSStreamDelegate 实现通信更新。第 8 章“底层网络”与第 13 章“使用 Bonjour 实现自组织网络”将会介绍 NSStream 的不同实现。

1.2.5　CFNetwork

CFNetwork API 位于基础的 BSD socket 之上，用在 NSStream、URL 加载系统、Bonjour 与 Game Kit API 的实现中。它为 HTTP 与 FTP 等高级协议提供了原生支持。CFNetwork 与 BSD socket 之间的主要差别在于运行循环集成。如果应用使用了 CFNetwork，那么输入与输出事件都会在线程的运行循环中进行调度。如果输入与输出事件发生在辅助线程中，就需要以恰当的模式开始运行循环。本章稍后的 1.3 节“运行循环”将会对此进行详细介绍。

CFNetwork 比 URL 加载系统提供了更多的配置选项，这个结果是喜忧参半的。在使用 CFNetwork 创建 HTTP 请求时可以使用这些配置选项。在创建请求时，需要手工将与请求一同发送的 HTTP 头和 Cookie 信息添加进去。但对于 NSURLConnection 来说，标准的头与 Cookie 信息会被自动添加进去。

CFNetwork 基础设施构建在 Core Foundation 层的 CFSocket 与 CFStream API 之上。CFNetwork 包含了针对特定协议的 API，比如用于与 FTP 服务器通信的 CFFTP、用于发送和接收 HTTP 消息的 CFHTTP、用于发布与浏览 Bonjour 服务的 CFNetServices 等。第 8 章将会详细介绍 CFNetwork，第 13 章则会概览 Bonjour。

1.2.6　BSD socket

BSD socket 构成了大多数 Internet 活动的基础，是网络框架层次体系中的最底层。BSD socket 使用 C 实现，但也可以用在 Objective-C 代码中。并不推荐使用 BSD socket API，因为它并没有在操作系统中插入任何钩子(hook)。比如，BSD socket 无法穿过系统范围的 VPN，如果 Wi-Fi 或是蜂窝无线电被关闭了，调用其 API 也无法自动激活。Apple 建议至少使用 CFNetwork 或是更高层的 API。第 8 章将会详细介绍 BSD socket 与 CFNetwork，并通过示例展示如何将其集成到应用中。

在实现各种网络 API 时，需要理解它们是如何与应用集成的。下一节将会介绍运行循环的概念，它会监控操作系统的网络事件并将这些事件转发给应用。

1.3　运行循环

运行循环是由类 NSRunLoop 表示的，有些线程可以让操作系统唤醒睡眠的线程以管理到来的事件，而运行循环则是这些线程的基础组件。运行循环是这样一种循环，可以在一个周期内调度任务并处理到来的事件。iOS 应用中的每个线程最多只有一个运行循环。对于主线程来说，运行循环会为你开始，在应用委托的 applicationDidFinishLaunchingWithOptions: 方法调用后就可以访问了。

不过，辅助线程必须显式运行自己的运行循环。在辅助线程中开始运行循环之前，你

至少要添加一个输入源或定时器；否则，运行循环就会立刻退出。运行循环向开发者提供了与线程交互的能力，不过有时并不是必需的。比如，没有任何其他交互而处理大数据集的线程可能就不会开始运行循环。然而，如果辅助线程与网络交互，就需要开启运行循环。

运行循环会从两类源中接收事件：输入源与定时器。输入源(通常是基于端口的或是自定义的)会异步向应用发送事件。这两类源的主要差别在于内核会自动发出基于端口的源信号，而自定义源就需要从不同的线程中手动发出。可以通过实现与 CFRunLoopSourceRef 相关的几个回调函数来创建自定义输入源。

定时器会生成基于时间的通知，它为应用(特别是线程)提供了一种机制以在未来的某个时间执行某个具体任务。定时器事件是同步发出的，并且与特定的模式有关，后面将会对此进行介绍。如果这个特定的模式当前并没有被监控，那么事件就会被忽略掉，线程也不会收到通知，直到运行循环“运行”在相应的模式下为止。

可以配置定时器以触发一次或是重复触发。重新调度是基于调度的触发时间而不是实际的触发时间。如果定时器触发，同时运行循环正在执行一个应用处理器方法，那么它会等待，直到下一次通过运行循环来调用定时器处理器为止，这一般是通过设定@selector()实现的。如果触发处理器被推迟到了下一次调用发生的那个点，那么定时器只会触发一个事件，之前延迟的事件则会被压制住。

运行循环也可以有观察者，它们不会被监控，这为对象提供了一种方式，使之可以在运行循环执行过程中的某个活动发生时收到回调。这些活动包括进入或退出运行循环、运行循环睡眠或唤醒、运行循环处理输入源或定时器之前等。CFRunLoopActivity 枚举的文档中对此有说明。观察者可以配置成触发一次，这样当触发后就会将其删除，也可以配置成重复的。要想添加运行循环观察者，请使用 Core Foundation 函数 CFRunLoopObserverRef()。

运行循环模式

每次通过运行循环都是在你所指定的模式下的一次运行。运行循环模式是由操作系统所用的一种约定，用于过滤监控的源并发布事件，比如调用委托方法等。模式包含了应该监控的输入源与定时器，以及当运行循环事件发生时应该通知的观察者。

在 iOS 中有两个预定义的运行循环模式。NSDefaultRunLoopMode(Core Foundation 中的 kCFRunLoopDefaultMode)是系统默认的，在开始运行循环及配置输入源时通常会使用它。NSRunLoopCommonModes(Core Foundation 中的 kCFRunLoopCommonModes)是个可配置的模式集合。通过在输入源实例上调用 scheduleInRunLoop:forMode:等方法，将 NSRunLoopCommonModes 赋给输入源会将其与当前组中的所有模式关联起来。

说明：

Mac OS X包含了 3 个额外的预定义运行循环模式，这可以在不同的文档中看到。NSConnectionReplyMode、NSModalPanelRunLoopMode与NSEventTrackingRunLoopMode提供了额外的过滤选项，不过iOS中并没有提供。

虽然NSRunLoopCommonModes是可配置的，但这是个底层过程，需要调用Core Foundation函数CFRunLoopAddCommonMode()。这会自动注册输入源、定时器与新模式的观察者，而不必手工将其添加到每个新模式中。可以通过指定自定义字符串(如@"CustomRunLoopMode")来定义自定义运行循环模式。要想提高自定义运行循环的效率，你至少需要添加一个输入源、定时器或是观察者。

虽然本节概览了运行循环，不过 Apple 提供了一些关于运行循环管理的颇有深度的资源，如果要开发高级的、基于网络的多线程应用，那么应该看看这些资源。开发者文档位于 https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Multithreading/RunLoopManagement/RunLoopManagement.html。第 8 章“底层网络”、第 13 章“使用 Bonjour 实现自组织网络”将会介绍因运行循环集成而获益的各种网络技术。

1.4　小结

理解 iOS 网络栈以及应用如何与运行循环交互是 iOS 开发者需要掌握的重要概念。良好架构的网络层为应用提供了极大的灵活性。与之类似，设计低劣的网络层则会对成功与可伸缩性带来不利影响。

本章介绍的工具概览了各种网络 API 并对它们进行了比较。对于它们的使用方式，本章虽然做了简单介绍，不过后续章节将会更深入地进行阐述。

第 2 章

设计服务架构

本章内容：

- 实现远程门面
- 使用服务定位器探测端点
- 使用服务版本化支持老版本应用

本章代码下载

可以在 www.wrox.com 的 Download Code 选项卡中下载本章代码，网址是 www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html。本章下载代码中包含一个示例项目与一套 Web Serive：

- Facade Tester.zip
- Facade PHP.zip

Web Service 是 iOS 网络应用的根本所在，其设计的灵活性与健壮性会对用户体验产生极大的影响。设计良好的服务 API 可以适应变化的后端数据源，同时为依赖它的应用提供不变的门面。服务定位器可以使应用能够动态探测到新的服务端点并使用它们，从而无须重新编译并重新向 App Store 提交应用。如果有必要重新提交应用，那么需要在过渡与升级期间支持应用的老版本，这是应用整个生命周期的一部分。如果想支持依旧被每天使用的老版本应用，同时又不妨碍为新版本增加新的特性，那么支持版本化的服务 API 就显得尤为重要了。本章将会在真实的业务场景中介绍这些重要的设计元素，同时会给出相应的示例实现。

2.1　远程门面模式

在设计应用的服务架构时，远程门面可以简化应用集成，并且可以让多个客户端共享相同的业务逻辑。门面模式用于抽象出客户端所用的底层系统的复杂性。比如，邮政系统包括大量的邮递员、卡车、飞机、配送中心和邮局；然而，用户所需的大多数任务都将这种复杂性隐藏起来了，只包含邮寄信件与接收包裹。用户无须了解信件是如何从纽约到达旧金山的，他们只需要支付邮资，然后等待邮件的到达即可。与之类似，应用 API 也可以将多个数据库查询或是后端系统请求抽象为一个单独的外部访问方法，该方法会返回操作的结果。只要门面的外部 API 契约保持不变，底层系统可以变化、升级或是完全移除而不会对使用该门面的任何客户端造成影响。

远程门面也使用了这种模式，并且将其用在了应用的 Web Service 层。它定义了一个不变的服务契约，应用可以通过该契约在外部创建、读取、更新或删除数据。API 通常用于与现有的业务系统进行交互，并提供具有相同功能的移动版本。图 2-1 展示了应用是如何直接查询各种端点的，图 2-2 展示了当门面与代表应用的各种后端服务进行交互时网络拓扑的变化情况。如果在设计最初的服务契约时能够保持谨慎并有预见性，那么相同的 API 就可以适应后端系统的大多数变化，这样应用就无须频繁更新来匹配服务基础设施，从而不会对应用的功能造成影响。

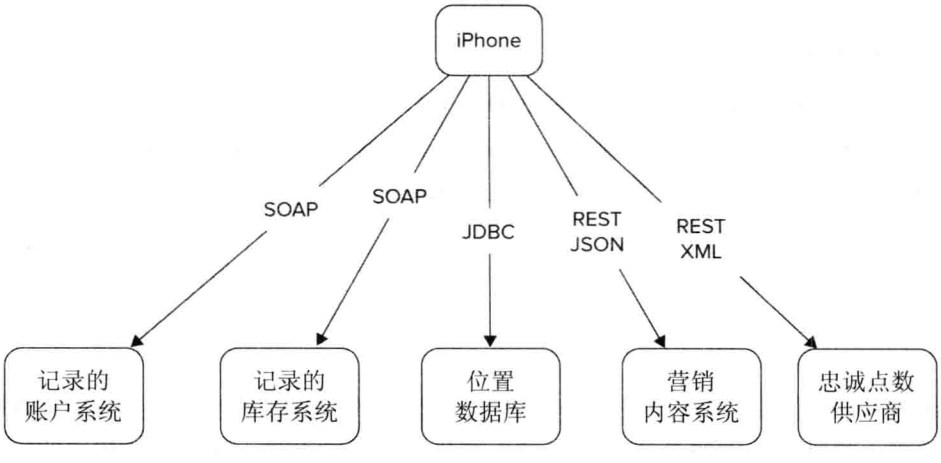


图 2-1

假设有家银行与其竞争对手合并了，该银行想将其现有的账户迁移到竞争对手的账户存储系统中。如果服务 API 的编写采用了抽象的银行功能，那么它就可以处理提供相同数据的任何后端数据库，即便存储在新的格式中也是如此。远程门面可以切换到新的数据源，转换与 API 契约不匹配的任何数据，然后将其返回给移动银行应用，而用户则完全不知道后面发生了什么事情。这种开发方式叫做契约编程，它能确保网络会话的两端遵循之前达成一致的输入与输出契约。只要契约依然有效，那么后端系统无论是重写、移植到其他语言还是升级，都不会对另一端造成任何负面影响。

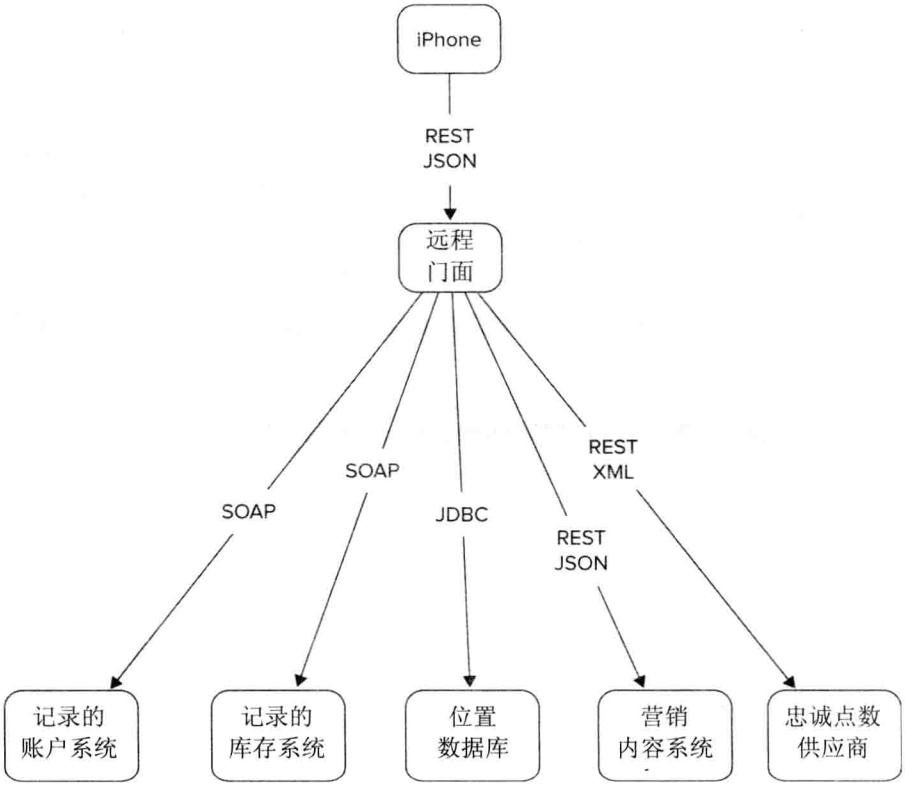


图 2-2

契约中应用一方的可维护性、可靠性与复杂性都会因门面模式而得到极大改善。随着应用中网络交互点数量的下降，需要支持未来门面版本的变化数量也会变得越来越少，它可以是自包含的。可靠性之所以会得到改进，是因为门面通常只有一个协议和一种消息格式，这样就降低了针对其他格式的第三方库或是单独的解析器的数量。这些变化都会降低应用的复杂性，节省开发成本，因为涵盖所有功能的单元测试数量会变少。在服务器端，我们只需要保护和向 Internet 公开一套端点即可，而不再是向众多不同的系统公开。

远程门面还会促使开发者将应用中的一些业务逻辑放到服务层中。某些变化频繁或是无法提前预估的函数可以在服务层中进行计算，然后只将最后的结果发送给客户端。在这种方式下，如果需要根据新的业务规则调整逻辑，那么应用无须更新即可生效。在银行合并这个示例中，这种调整可能是新的机构所采用的新的密码安全需求。如果应用只接收用户的密码，然后询问门面密码是否合法，那么逻辑就可以随意变化了。采用类似的模式来验证 e-mail 地址则可以轻松适应即将到来的自定义顶层域名(Top-Level Domain，TLD)；然而，如果有效的 TLD 列表是在应用中硬编码的，可能就会拒绝掉合法的 E-mail 地址，直到发布应用更新为止。在面对变化的业务流程时，远程门面可以在网络应用发布后确保企业拥有最大的灵活性。

相同的特性也适用于 API 的输入。门面可以将请求转换为后端系统所需的格式；比如，可以将 JSON 请求转换为 SOAP 请求，还可以对无法公开到 Internet 的其他系统强制施加

安全约束，在转发请求前追踪和验证 API key，或是限制发送给某些后端系统的请求数量。

2.1.1　门面服务示例

Facade Tester 应用使用两个 Web Service 装配视图：股票报价服务和天气服务。这两个服务从两个独立的源获取各自的数据，然后将数据转换为一种常见的输出格式。这么做模拟了一种门面服务，当应用在运行时，服务必须能在两个后端系统之间进行切换。这两个示例使用的都是版本 1 服务；版本 2 服务则用在 2.2 节“服务版本化”中。

股票报价服务将数据加载为逗号分隔的值(Comma-Separated Value，CSV)或是 XML 文档，如代码清单 2-1 所示。

代码清单 2-1　从两个股票报价源生成常见的输出(stockQuote_v1.php)

```
<?php
$useYahooResults = true;
$ticker = "AAPL";

if($useYahooResults) {
    $rawData = rtrim(file_get_contents("http://finance.yahoo.com/ d/
        quotes.csv?s=".$ticker."&f=snl1p2o"),"\r\n");

    $data = explode(",",$rawData);

    $symbol = trim($data[0],'"');
    $name = trim($data[1],'"');
    $currentPrice = trim($data[2],'"');

} else {
    $rawXML = file_get_contents("http://www.webservicex.net/stockquote.asmx/
        GetQuote?symbol=".$ticker);

        $wrapperData = simplexml_load_string($rawXML);

        $xmlData = simplexml_load_string($wrapperData);

        $symbol = (string)$xmlData->Stock->Symbol;
        $name = (string)$xmlData->Stock->Name;
        $currentPrice = (string)$xmlData->Stock->Last;
    }

    $response = array("symbol" => $symbol,
                      "name" => $name,
                      "currentPrice" => $currentPrice);

    // output final results:
    printjson_encode($response);

    ?>
```

下述逗号分隔的字符串具有关键的股票值：公司名、最近行市、开盘价以及开盘后的变动百分比。

```
{ticker symbol},{name},{last trade price},{percentage change},{opening price}

example:
"AAPL","Apple Inc.",530.12,"-2.92%",545.31
```

下述 XML 文档以更加结构化的格式展现出了相同的数据。该文档包含了 CSV 字符串中的所有信息，另外又加上了该服务忽略掉的一些额外数据。

```
<StockQuotes>
   <Stock>
      <Symbol>AAPL</Symbol>
      <Last>530.12</Last>
      <Date>5/17/2012</Date>
      <Time>4:00pm</Time>
      <Change>-15.955</Change>
      <Open>545.31</Open>
      <High>547.50</High>
      <Low>530.12</Low>
      <Volume>25614960</Volume>
      <MktCap>495.7B</MktCap>
      <PreviousClose>546.075</PreviousClose>
      <PercentageChange>-2.92%</PercentageChange>
      <AnnRange>310.50 - 644.00</AnnRange>
      <Earns>41.042</Earns>
      <P-E>13.31</P-E>
      <Name>Apple Inc.</Name>
   </Stock>
</StockQuotes>
```

如果变量$useYahooResults 为 true，就会加载 CSV 字符串，否则就会加载 XML。无论输入源是什么，门面都会以常见的 JSON 格式返回其数据，如下所示：

```
{"symbol":"AAPL","name":"AppleInc.","currentPrice":"-2.92%"}
```

门面所用的任何数据源都至少要提供所需的最低限度的字段，从而遵守它与 API 客户端所达成的契约。

示例门面还实现了一个 Web Service，从两个源之一提供弗吉尼亚州里士满当前的天气。这两个源都以 JSON 形式提供了天气状况，不过每个源具体的响应格式则大相径庭。该服务类似于这种情况：你想要将后端系统升级到新版本，新旧版本具有相同的基础数据，不过数据的组织形式是完全不同的。代码清单 2-2 展示了这个天气服务，它遵循与股票服务相同的基础结构。

代码清单 2-2　从两个天气服务生成常见的输出(weather_v1.php)

```
<?php
$useYahooResults = true;
```

```
if($useYahooResults) {
    $rawJSON = file_get_contents("http://query.yahooapis.com/v1/
                public/yql?q=select%20item%20from%20weather.forecast
                %20where%20location%3D%2248907%22&format=json");
    $rawData = json_decode($rawJSON);

$currentTemperature = $rawData->query->results->channel->item->
                                                    condition->temp;
$currentConditions = $rawData->query->results->channel->item->
                                                    condition->text;

} else {
    $rawJSON = file_get_contents("http://weather.yahooapis.com/forecastjson?
        w=12518996");
    $rawData = json_decode($rawJSON);

    $currentTemperature = (string)$rawData->condition->temperature;
    $currentConditions = $rawData->condition->text;
}

$response = array( "city" => "Richmond",
                   "state" => "Virginia",
                   "currentTemperature" => $currentTemperature);

/*
 * output final results:
 *
 * {"city":"Richmond","state":"Virginia","currentTemperature":"63"}
 */
printjson_encode($response);

?>
```

现在，消费客户端可以使用发布的每个服务，数据源也可以动态切换而无须修改其处理股票或天气数据的方式。

2.1.2　门面客户端示例

Facade Tester 应用说明了如何使用输出格式并在表视图中展现结果。代码清单 2-3 展示了如何通过 Grand Central Dispatch 在后台线程中加载门面天气服务。代码清单 2-4 展示了解析股票报价服务的相关代码。JSON 结果是通过 iOS 5 的 NSJSONSerialization 解析的，然后赋给了表视图所用的局部变量。这证明了门面模式确实能减轻我们的工作量，iOS 集成的关键代码只有区区几行。要想了解关于通过网络加载数据的更多信息，请参见第 3 章“构建请求”；要想了解关于 JSON 解析的更多信息，请参见第 4 章“生成与解析负载”。

代码清单 2-3　加载并解析天气服务(FTWeatherViewController.m)

```
NSString *v1_city;
```

```
NSString *v1_state;
NSString *v1_temperature;

- (void)loadVersion1Weather {
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

        FTAppDelegate *appDelegate = (FTAppDelegate*)
            [[UIApplicationsharedApplication] delegate];

        if(appDelegate.urlForWeatherVersion1 != nil) {
            NSError *error = nil;
            NSData *data = [NSData
                    dataWithContentsOfURL:appDelegate.urlForWeatherVersion1
                    options:NSDataReadingUncached
                    error:&error];
            if(error == nil) {
                NSDictionary *weatherDictionary = [NSJSONSerialization
                    JSONObjectWithData:data
                            options:NSJSONReadingMutableLeaves
                              error:&error];
                if(error == nil) {
                    v1_city = [weatherDictionaryobjectForKey:@"city"];
                    v1_state = [weatherDictionaryobjectForKey:@"state"];
                    v1_temperature = [weatherDictionaryobjectForKey:
                        @"currentTemperature"];

                    // update the table on the UI thread
                    dispatch_async(dispatch_get_main_queue(), ^{
                        [self.tableViewreloadData];
                    });

                } else {
                    NSLog(@"Unable to parse weather because of error: %@",
                                                                    error);
                      [selfshowParseError];
                }

            } else {
                [selfshowLoadError];
            }
        } else {
            [selfshowLoadError];
        }
    });
}
```

代码清单 2-4　加载并解析股票报价服务(FTStockViewController.m)

```
NSString *v1_symbol;
NSString *v1_name;
NSNumber *v1_currentPrice;

- (void)loadVersion1Stock {
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

        FTAppDelegate *appDelegate = (FTAppDelegate*)
            [[UIApplicationsharedApplication] delegate];

        if(appDelegate.urlForStockVersion1 != nil) {
            NSError *error = nil;
            NSData *data = [NSDatadataWithContentsOfURL:
                                appDelegate.urlForStockVersion1
                                options:NSDataReadingUncached
                                                error:&error];
            if(error == nil) {
                NSDictionary *stockDictionary = [NSJSONSerialization
                    JSONObjectWithData:data
                              options:NSJSONReadingMutableLeaves
                                error:&error];
                if(error == nil) {
                    v1_symbol = [stockDictionaryobjectForKey:@"symbol"];
                    v1_name = [stockDictionaryobjectForKey:@"name"];
                    v1_currentPrice = [NSNumbernumberWithFloat:
                        [[stockDictionaryobjectForKey:@"currentPrice"]
                        floatValue]];

                    // update the table on the UI thread
                    dispatch_async(dispatch_get_main_queue(), ^{
                        [self.tableViewreloadData];
                    });

                } else {
                    NSLog(@"Unable to parse stock quote because of error:
                        %@", error);
                    [selfshowParseError];
                }
            } else {
                [selfshowLoadError];
            }
        } else {
```

```
            [selfshowLoadError];
        }
    });
}
```

2.2　服务版本化

移动应用会经常更新以修复 Bug 和添加新特性，不过有些人常常觉得 Web Service 也需要维护并随之更新。服务版本化是这样一种技术，它更新与客户端的 API 契约，同时依然保留之前的版本以供现有的应用版本使用。通过 App Store 发布的应用无法强制用户升级到最新版本，这意味着在过渡期间现有的 Web Service 依然要保持功能的可用性。根据用户的升级行为，关闭现有服务而不影响旧版本的用户几乎是不可能的。一种方式就是加入检查逻辑，检查最小支持的应用版本，然后显示升级消息，直到用户同意升级为止。然而，在之前可以使用的版本上显示的这些烦人的消息可能会让一些用户感到心烦意乱，他们可能很快就会打爆你的支持热线，并在 App Store 评论中给出差评。由于这些潜在的负面效果，恰当的服务版本化确确实实是最佳解决方案。

API 版本化并不仅仅限于新的应用升级；它还可以用于向各种类型的设备发布不同的或是扩展的数据。比如，一个报表应用可能在 iPhone 上显示时需要一套数据，但在 iPad 上显示时则需要更加完整的数据集，因为 iPad 拥有更大的屏幕尺寸。如果这些额外的数据有着较大的后端或是网络负载，那么你肯定不想在并不需要它们的 iPhone 服务请求上浪费这些资源。

版本化系统的结构主要有两种方式：一种是主动系统，远程门面会接收到客户端的当前版本，然后选择正确的端点；另一种是被动系统，版本化服务端点是硬编码到客户端的每个新发布中。到底哪一种才是提供版本号作为输入的最佳方式，则是由每一家企业决定的，不过通常情况下，版本号会包含在 REST 端点的 URL 结构中或是作为查询参数进行传递。下述代码片段展示了这两种版本输入方式：

```
// a version given in the URL structure
http://example.com/api/1.0/stockquote/AAPL

// a version given as a query parameter
http://example.com/api/stockQuote.php?ticker=AAPL&apiVersion=1.0
```

被动系统是实现服务版本化最简单的方式。这种方式无须规划额外的服务器，在组织上的代价要大于技术上的。要想实现这种方式，只需要将版本号硬编码到客户端应用中已经定义好的端点 URL 中。由于在应用发布后，这些 URL 在功能上就是不变的了，因此可以确保硬编码使用该版本的应用总是会使用该版本。当新的客户端版本需要改变服务契约时，你只需要增加硬编码的 API 版本号，然后创建新的 Web Service 即可。

主动系统拥有被动系统的全部优势；然而就像所有的门面交互一样，它还具有未来改

变行为的能力。如果相同 Web Service 的两个不同版本具有相同的输入与输出契约时，但它们所执行的某些计算是不同的，这样兼容于两个版本的老客户端就可以在未来动态切换了。比如，如果某个在线零售商现在没有征收某个州的销售税，那么可以将客户端发送至价格检查服务的 1.0 版。然而，如果未来需要开始征收销售税，那么他只需要创建 2.0 版的服务，返回正常价格加上税就可以了。假设两种情况下返回的最终结果都是数字，那么服务契约就不会受到影响。要想实现主动版本化系统，门面需要将所有可能的客户端应用版本组合到兼容性桶(compatibility bucket)当中，然后为每个桶分配正确的 API 版本供其使用。为了简化未来的开发，请为高于门面所知道的最大版本号的客户端版本选择好的默认值，通常是最近的 API 版本。

2.2.1　版本化服务示例

两个示例 Web Service 都有两个版本，模拟了随着业务需求变化而扩展的服务契约。有些输出字段类型已经被修改，还有些新增的字段。这些示例使用了被动版本化系统，仅仅通过修改 URL 来指定新的版本。回忆一下天气服务的 1.0 版 weather_v1.php，它有如下输出格式：

```
{"city":"Richmond","state":"Virginia","currentTemperature":"63"}
```

currentTemperature 表示为字符串，这使得客户端在需要整型逻辑时变得复杂，比如设置用于对当前温度进行分类的寒冷、温和或炎热天气的阀值。服务的 2.0 版修复了这个疏忽，将返回值定义为数值类型。它还添加了 currentConditions 字段，这是对当前天气的文本说明。weather_v2.php 的输出格式如下所示：

```
{"city":"Richmond","state":"Virginia","currentTemperature":63,
 "currentConditions":"Mostly Cloudy"}
```

股票报价服务的 1.0 版与 2.0 版之间的变化也与此类似。stockQuote_v1.php 的第一个版本提供了基本的输出：

```
{"symbol":"AAPL","name":"Apple Inc.","currentPrice":"530.12"}
```

注意 currentPrice 在 1.0 版中是字符串，但在 stockQuote_v2.php 中却表示为数字，这样可以简化其在客户端的格式化：

```
{"symbol":"AAPL","name":"Apple Inc.","openingPrice":545.31,
 "currentPrice": 530.12,"percentageChange":"-2.92%"}
```

此外，还添加了两个新字段：openingPrice 与 percentageChange。

2.2.2　使用版本化服务的客户端示例

Facade Tester 中的天气视图控制器可以显示两个 API 版本的输出。loadVersion1Weather 与 loadVersion2Weather 会检查 API 端点的 URL 的应用委托，如代码清单 2-5 中的粗体代码所示。由于该例使用的是被动版本化，因此在这里直接硬编码 URL 看起来更加自然；然

而，在应用委托中定义可以使得在实现服务定位器时更具灵活性，下一节将会对此进行介绍。

代码清单 2-5　从应用委托中获取 API 端点(FTWeatherViewController.m)

```
- (void)loadVersion1Weather {
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

        FTAppDelegate *appDelegate = (FTAppDelegate*)
            [[UIApplicationsharedApplication] delegate];
        if(appDelegate.urlForWeatherVersion1 != nil) {
            NSError *error = nil;
            NSData *data = [NSData
dataWithContentsOfURL:appDelegate.urlForWeatherVersion1
                                options:NSDataReadingUncached
                                error:&error];

        // remaining code removed for brevity
}

- (void)loadVersion2Weather {
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

        FTAppDelegate *appDelegate = (FTAppDelegate*)
            [[UIApplicationsharedApplication] delegate];

        if(appDelegate.urlForWeatherVersion2 != nil) {
            NSError *error = nil;
            NSData *data = [NSData
dataWithContentsOfURL:appDelegate.urlForWeatherVersion2
                                options:NSDataReadingUncached
                                error:&error];

        // remaining code removed for brevity
}
```

应用在加载正确的 JSON 数据后，只是根据该版本的服务契约对其进行解析。天气服务的 1.0 版会加载城市、州与当前温度，如下所示：

```
v1_city = [weatherDictionaryobjectForKey:@"city"];
v1_state = [weatherDictionaryobjectForKey:@"state"];
v1_temperature = [weatherDictionaryobjectForKey:@"currentTemperature"];
```

2.0 版与之类似，不过会将当前温度解析为数字，还会查找当前的情况，代码如下所示：

```
v2_city = [weatherDictionaryobjectForKey:@"city"];
v2_state = [weatherDictionaryobjectForKey:@"state"];
v2_temperature = [[weatherDictionaryobjectForKey:@"currentTemperature"]
```

```
intValue];
    v2_conditions = [weatherDictionaryobjectForKey:@"currentConditions"];
```

在设置v2_temperature时，将之从NSNumber(NSJSONSerialization所用的数值类型)转换为视图控制器所用的整型。

2.3　服务定位器

服务定位器是一个帮助应用动态探测远程源 API 端点的工具，它可以解决应用硬编码的无效或不再存在的端点问题。应用开发者还可以通过服务定位器将之前发布的应用重新指向可用的新服务。这些新服务无须改变与客户端的 API 契约；比如，端点移到了不同的服务器或子域中、位于负载平衡器之后，或是移到了由 SSL 保证安全的 HTTPS 端点上。甚至可以为每个开发环境创建新的服务定位器文件以便轻松在开发、QA 或生产资源间切换，而这一切只需一次变换即可。

服务定位器的核心只是一个包含了 API 端点与关于这些端点的一些简要元数据的文件。应用通过这些元数据确定该使用哪个端点。比如 API 版本、输入或输出格式、设备类型以及安全级别等。它还需要包含端点的 URL 以及客户端应用用于匹配端点与其函数的键。由于该文件是静态的，不会频繁修改，因此可以轻松将其部署到 Web 服务器或内容分发网络(CDN)上。服务定位器的源需要高度可靠，因为它是应用失败的单点。虽然这看起来有点问题，但如果应用直接查询每个独立的后端服务，就会有很多个失败点存在，相比于此，单点失败会更好一些。在可能的情况下，服务定位器应该是负载平衡的，从而避免全部的用户请求发给一台单独主机。由于 CDN 的设计目的是针对静态文件的高可靠性，通常能比平常的 Web 服务器处理更高的持续带宽占有情况，因此我们推荐你在可能的情况下使用 CDN 来服务于服务定位器文件。

由于大多数 Web Service 都会将结果以 JSON 的形式输出，因此使用 JSON 来表示服务定位器会比较好。代码清单 2-6 展示了 Facade Tester 如何使用服务定位器探测天气与股票报价 API 端点。该结构将端点的所有版本都组合到了一个文件中；然而，你还可以为每个 API 版本创建单独的服务定位器文件。后者可以防止应用版本混合在一起并匹配不同的服务版本，不过对于某些业务场景来说这种约束并不是什么问题。

代码清单 2-6　示例服务定位器文件(serviceLocator.json)

```
{
    "services": [
        {
            "name": "stockQuote",
            "url": "http://example.com/api/stockQuote_v1.php",
            "version": 1
        },
        {
            "name": "stockQuote",
```

```
            "url": "http://example.com/api/stockQuote_v2.php",
            "version": 2
        },
        {
            "name": "weather",
            "url": "http://example.com/api/weather_v1.php",
            "version": 1
        },
        {
            "name": "weather",
            "url": "http://example.com/api/weather_v2.php",
            "version": 2
        }
    ]
}
```

实现了服务定位器模式的任何客户端的第一个动作通常都是加载并解析文件。由于所有的网络调用都需要端点，而端点只位于该文件中，因此在任何其他的网络动作发生之前必须先解析该文件。当应用返回到前台以确保端点数据是最新时，定位器文件也应该进行更新。应用可以停留在后台状态中一段时间，它之前可能已经加载了一个服务定位器文件，不过文件现在可能已经不是最新的了。在某些情况下，过时的端点可能会退出并超时，这会导致差劲的用户体验。通常情况下，当服务定位器加载时，应用会显示启动画面。

代码清单 2-7 展示了一个应用，它会在应用启动和返回到前台时加载服务定位器。它将 URL 存储为应用委托中的属性；然而，更加复杂的应用则需要专门的网络管理器，由它来处理服务定位器的加载，其他控制器也会使用它针对特定的网络调用查询端点。

代码清单 2-7　加载并解析服务定位器文件(FTAppDelegate.m)

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    // some code removed for brevity
    /*
     * load the service locator
     *
     * note: You should probably show a splash screen of some kind here
     * that waits for the SL to fully load. Currently a user could
     * try to start a network request before it knows which URL to use.
     */
    [selfloadServiceLocator];

    return YES;
}

- (void)applicationDidBecomeActive:(UIApplication *)application {
    // load the service locator
```

```
    [selfloadServiceLocator];
}

- (void)loadServiceLocator {
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{


        NSError *error = nil;
        NSData *data = [NSDatadataWithContentsOfURL:[NSURL URLWithString:
                        @"http://example.com/api/serviceLocator.json"]
                                            options:NSDataReadingUncached
                                              error:&error];

        if(error == nil) {
            NSDictionary *locatorDictionary = [NSJSONSerialization
                JSONObjectWithData:data
                           options:NSJSONReadingMutableLeaves
                             error:&error];
            if(error == nil) {
                self.urlForStockVersion1 = [self
                    findURLForServiceNamed:@"stockQuote"
                                   version:1
                              inDictionary:locatorDictionary];
                self.urlForStockVersion2 = [self
                    findURLForServiceNamed:@"stockQuote"
                                   version:2
                              inDictionary:locatorDictionary];
                self.urlForWeatherVersion1 = [self
                    findURLForServiceNamed:@"weather"
                                   version:1
                              inDictionary:locatorDictionary];
                self.urlForWeatherVersion2 = [self
                    findURLForServiceNamed:@"weather"
                                   version:2
                              inDictionary:locatorDictionary];

            } else {
                NSLog(@"Unable to parse service locator because of error:
                    %@", error);

                // inform the user on the UI thread
                dispatch_async(dispatch_get_main_queue(), ^{
                    [[[UIAlertViewalloc] initWithTitle:@"Error"
                                               message:@"Unable to parse
```

```
                                                    service locator."
                                                delegate:nil
                                        cancelButtonTitle:@"OK"
                                        otherButtonTitles:nil] show];
                });
            }

        } else {
            NSLog(@"Unable to load service locator because of error: %@", error);

            // inform the user on the UI thread
            dispatch_async(dispatch_get_main_queue(), ^{
                [[[UIAlertViewalloc] initWithTitle:@"Error"
                                           message:@"Unable to load service
                                                    locator. Did you remember
                                                    to update the URL to your own
                                                    copy of it?"
                                           delegate:nil
                                  cancelButtonTitle:@"OK"
                                  otherButtonTitles:nil] show];
            });
        }
    });
}

- (NSURL*)findURLForServiceNamed:(NSString*)serviceName
                         version:(NSInteger)versionNumber
                    inDictionary:(NSDictionary*)locatorDictionary {

    NSArray *services = [locatorDictionaryobjectForKey:@"services"];

    for(NSDictionary *serviceInfo in services) {
        NSString *name = [serviceInfoobjectForKey:@"name"];
        NSInteger version = [[serviceInfoobjectForKey:@"version"] intValue];

        if([name caseInsensitiveCompare:serviceName] == NSOrderedSame&&
            version == versionNumber) {

            return [NSURL URLWithString:[serviceInfoobjectForKey:@"url"]];
        }
    }

    return nil;
}
```

2.4　小结

灵活的服务架构需要在应用的第一个版本发布之前经过精心的规划与实现，这样才能获取最大的收益。如果某个版本采用了硬编码的端点或是业务逻辑，那么可以有效地支持该配置，即便业务发生了巨大变化也是如此。通过远程门面、API 版本化与服务定位器的融合，可以采用多种方式来改变已发布应用的业务逻辑与 API 设置。现在，对产品代码进行细微修改，以及在新版本中增加主要的新特性而无须破坏之前的应用版本已经变得很容易了。之前的服务基础设施的开发代价看起来可能没有必要，不过随着应用逐渐变大和演化，这种代价会带来很多倍的收益。

第 II 部分

HTTP请求：iOS网络功能

- 第 3 章　构建请求
- 第 4 章　生成与解析负载
- 第 5 章　错误处理

第 3 章

构建请求

本章内容：

- 理解 HTTP 请求的结构
- 从 iOS 应用中发出 HTTP 请求
- 使用 HTTP 请求的高级操作

本章代码下载

可以在 www.wrox.com 的 Download Code 选项卡中下载本章代码，网址是 www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html。代码位于第 3 章的压缩包中，并且根据每节的名字单独命名。

你可能从之前章节的介绍中已发现，iOS 中首选的通信方式是 HTTP。iOS 提供的最方便的网络 API 都是针对 HTTP 的，HTTP API 的文档描述也是最为详尽的，高层 HTTP API 可以很好地集成到基于运行循环的 iOS 应用架构中。毫无疑问，HTTP 与 HTTPS 是 iOS 网络通信中使用最多的协议。

本章将会介绍 HTTP 请求的结构以及应用是如何使用这些请求的。此外，本章还会就生成 HTTP 请求、接收 HTTP 响应的 3 个主要方法给出具体示例，同时会对何时该用何种方法给出一些建议。最后，本章还会探索使用 HTTP 协议的一些更高级的方式。

说明：

在本章及本书的其余章节中，术语 HTTP 指的是不安全的 HTTP 与安全的 HTTPS 请求。在这两个协议存在差异时会有相关的说明。

3.1　HTTP 介绍

Tim Berners-Lee 创造了 HTTP 协议的首个版本，HTTP 协议是作为 WorldWideWeb 项目的一部分在 1990 年开始的。该协议通过 HTML 定义，作为向位于瑞士日内瓦的 CERN 研究员们传输信息的一种方式，它使用标准的用户界面与标记语言。展现给用户的信息还可以包含指向其他相关信息的链接，可以通过激活文本中的链接访问。在该项目之前，信息是以多种格式存储的，只能通过基于该格式的不同工具访问，这使得查找、使用以及将相关历史研究关联到自己的研究变得异常困难。可以通过 http://www.w3.org/Proposal.html 阅读 WorldWideWeb 项目的最初提案。

说明：

关于 HTTP 与 HTML 的发明有个很有趣的插曲，那就是首个万维网服务器与浏览器是在一台 NeXTStep 计算机上编写的。在 1997 年，Apple 收购了 NeXT Computer 并将 NeXTStep 作为 Mac OS X 的基础。Apple 的 Mac OS X 后来成为 iOS 的基础。

Berners-Lee 最初的提案中有 3 个主要的创新：HTML、HTTP 与 URL。HTML 定义了向文本添加样式的一种方式、HTTP 定义了在服务端与客户端之间传输数据的一种方式、URL 定义了在网络机器中定位唯一资源的一种方式。

在 20 世纪 90 年代中期，随着 Web 浏览器与 HTTP 的使用从实验室转向业务与家庭领域，HTTP 很快成为恶意用户与组织的攻击目标。在响应中，工程师们开发出了用于安全传输 HTTP 的标准。一开始有两个彼此竞争的标准，分别是 HTTPS 与 S-HTTP。HTTPS 会加密整个 HTTP 消息，而 S-HTTP 则只会加密消息体，消息头则是未加密的文本。微软与 Netscape 决定对 HTTPS 实行标准化，这导致人们放弃了 S-HTTP。

HTTP一开始的设计目标是用于服务端到客户端浏览器之间传输的人类可读内容的通信。万维网的使用使得HTTP成为Internet上人类可读信息通信事实上的标准，并且进入用户的家庭。随着HTTP承载的HTML日渐增多，人们认识到HTTP还可以轻松地在机器与机器之间传输信息。

由于使用 HTML 与 HTTP 的 Web 浏览器日渐普及，使用 HTTP 传输机器可读的数据成为 Internet 上的系统间传递信息的最有效方式。如果应用所处的计算机已经联网，那么几乎可以保证通过 HTTP 可以与 Internet 上的另一台主机进行通信。公司网络代理与防火墙可以在安全的公司网络与 Internet 之间安全地传输 HTTP 请求，并在传输过程中执行过滤与安全验证。虽然 Internet 的设计目标是承载众多不同的应用层协议，但 HTTP 却成为对于最终用户来说配置量最少的协议。

3.2 理解 HTTP 请求与响应

要想高效使用 HTTP 进行客户端与服务器之间的通信，你应该理解该协议的原理。本节将会介绍现代应用中所使用的 HTTP 的主要原理与结构。

对于计算机通信来说，HTTP 请求遵循着客户端-服务器范式。图 3-1 展示了一个简单的 HTTP 请求的步骤序列。客户端建立一个到服务器的 TCP 连接，然后发送 HTTP 请求。服务器随后通过在同一个 TCP 连接上发送一个 HTTP 响应来响应该请求。接下来，客户端可以重用这个 TCP 连接，发送另一个请求或是将连接关闭。早期的 HTTP 协议版本只允许在一个 TCP 连接上发送一个请求。HTTP 1.1 则允许客户端重用连接。

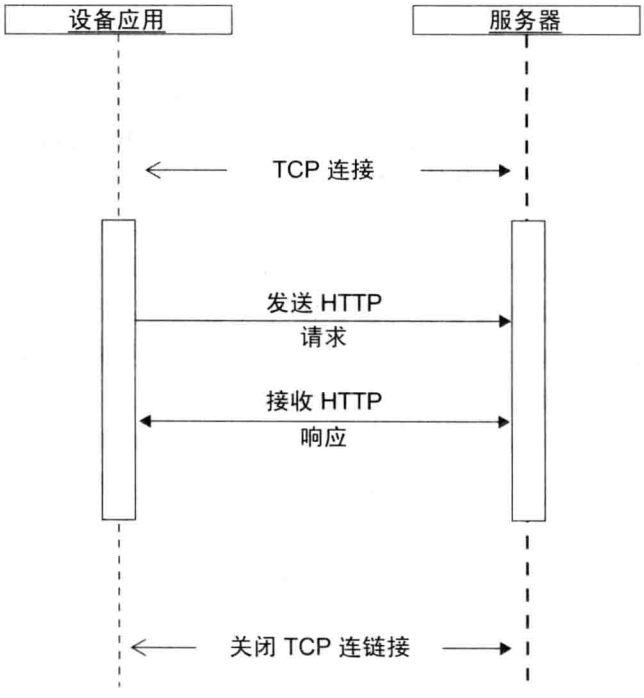


图 3-1

说明：

HTTP 的规范是 IETF RFC 2616。该 RFC 于 1999 年被采纳为标准，可以在 http://www.ietf.org/rfc/rfc2616.txt 上查阅该 RFC。

HTTP 与 HTTPS 之间的最重要差别在于会话的连接建立阶段。在 TCP 连接建立好、HTTP 请求发送前，客户端与服务器之间必须建立 SSL 会话。SSL 会话建立包含多个阶段：客户端与服务器协商使用何种密码、交换公钥、验证协商以及验证身份(可选)。当 SSL 会话建立完毕后，在 TCP 连接之上传输的所有数据都将是加密的。

3.2.1　URL 结构

从 iOS 开发者的角度来看，WorldWideWeb 项目的另一重要发明是 URL。URL 为 Internet 上的任何资源与内容提供了全局唯一的位置名。作为原则，单个资源可以通过多个 URL 定位，但单个 URL 不能引用不同的资源。该规则存在例外情况，比如主机名可以指向一台模糊的主机。在 iOS 的 URL 加载系统中，NSURL 对象用于管理 URL 对象。

通常情况下，URL 由 5 部分构成，如图 3-2 所示。

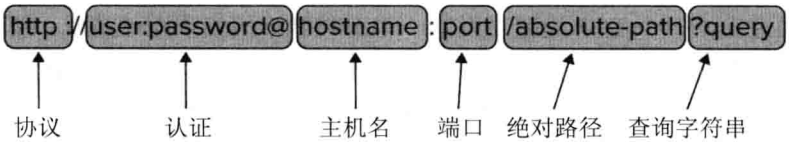


图 3-2

- 协议——协议部分指定了使用哪个应用层协议与服务器通信。如果上网有一段时间了，那么可能会记得除了 HTTP 外还可以使用 FTP 作为协议。HTTP 的优势导致 HTTP 协议之前所用的其他协议都几乎灭绝了。iOS 应用中常用的另一协议是 FILE 协议。FILE 请求用于在应用沙箱中取得本地文件系统中的资源。如果使用字符串而没有使用协议创建 NSURL 对象，那么默认就会使用 FILE 协议。
- 认证——某些 HTTP 服务器支持在 URL 中传输用户认证以实现 BASIC 认证。在图 3-2 中，认证部分包含了认证用户的用户名与密码。该格式使用得并不多，相比于其他认证方法，这种认证方式的安全性有些低。
- 主机名——URL 的主机名部分指定了包含请求资源的主机的 TCP 主机名或 IP 地址。如果 URL 协议是 FILE，那么这部分与端口部分必须省略掉。之前的规则提到单个 URL 要引用唯一资源，不过在使用相对或本地主机名时就会出现例外情况。比如，如果使用 localhost 作为主机名，URL 会引用本地机器；因此，不同机器上的相同 URL 可以引用不同的资源。
- 端口——URL 的端口部分指定了客户端连接的 TCP 端口。如果省略，那么客户端就会使用特定协议的默认端口：HTTP 是 80，HTTPS 是 443。如果应用运行的设备不在你所控制的网络中，那么最好使用这些端口值，因为有些网络代理和防火墙会出于安全或隐私等原因阻塞非标准的端口值。
- 绝对路径——绝对路径部分指定了网络资源的路径，就好像 HTTP 服务器可以钻取为目录树一样。绝对路径可以包含任意数量的路径部分，每部分使用斜杠字符(/)分隔。绝对路径不可以包含问号、空格、回车与换行符。很多 REST 服务使用路径来传递值，用来唯一标识数据库中存储的实体。比如，路径/customer/456/address/0 可以指定索引为 0 的地址，并且具有标识 456 的客户。
- 查询字符串——URL 的最后一部分是查询字符串。这个值与绝对路径部分是通过问号分隔的。根据约定，多个查询参数通过&字符分隔。查询字符串不可以包含回车、空格与换行符。

由于绝对路径与查询字符串的内容是受限的，因此URL通常会使用百分号进行编码。RFC 3986(http://tools.ietf.org/html/rfc3986)规定了URL百分号编码的细节信息。iOS为NSString对象提供了一个方法来执行URL的百分号编码。下述代码片段展示了如何对一个NSString对象进行百分号编码：

```
NSString *urlString = @"http://myhost.com?query=This is a question";
NSString *encoded = [urlString
    stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
```

编码的结果值是 http://myhost.com?query=This%20is%20a%20question。该编码不同于URL编码，由于不会对&字符编码，因此不会改变URL参数的分隔。URL编码会编码&、问号与其他标点符号。如果查询字符串包含了这些字符，那么需要实现一种更加彻底的编码方法。

3.2.2 请求内容

HTTP请求包含3部分：请求行、请求头与请求体。请求行与请求头是文本行，通过回车/换行符分隔(值为13的字节，或是0x0D/值为10的字节，或是0x0A)。在HTTP请求中这样使用文本值，使得它们很容易构建、解析和调试。空行(仅包含回车/换行符或是仅有换行符)将请求头与请求体划分开来。

下述代码片段包含一个HTTP请求示例，它来自于一个查询请求：

```
GET /search?source=ig&hl=en&rlz=&q=ios&btnG=Google+Search HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:11.0)…
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en,en-us;q=0.7,en-ca;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://www.google.com/ig?hl=en&source=webhp
Cookie: PREF=ID=fdf9979…
```

请求行是发送给服务器的第一行数据。请求行包含3方面主要信息：HTTP请求方法、请求URI与HTTP版本。

请求方法是个单词，标识了客户端请求的动作。由于区分大小写，因此表3-1中列出的标准方法都是大写的。在之前的代码片段中，请求方法是GET。

表3-1 常见的请求方法及用法

方 法	标 准 用 法
GET	从服务器获取一段内容，用HTTP术语来说就是实体。GET请求通常不包含请求体，不过也是可以包含的。有些网络缓存设施只会缓存GET响应。GET请求通常不会导致服务器端的数据发生变化

(续表)

方　法	标 准 用 法
POST	使用客户端提供的数据更新实体。POST 请求通常会在请求体中加入应用服务器所需的信息。POST 请求是非幂等的，这意味着如果处理多个请求，那么结果与处理单个请求是不同的
HEAD	获取响应的元数据而无须检索响应的全部内容。该方法通常用于检查服务器最近的内容变化而无须检索全部内容
PUT	使用客户端提供的数据添加实体。PUT 请求通常将应用服务器所需的信息放在请求体中来创建新的实体。通常情况下，PUT 请求是幂等的，这意味着多个请求的处理会产生相同的结果
DELETE	根据 URI 的内容或客户端提供的请求体来删除实体。DELETE 请求是 REST 服务接口中使用最为频繁的请求

请求行中的第 2 个域是 URI，URI 唯一标识请求的目标。如果请求使用的是 GET 方法，那么 URI 就会明确指定从目标主机检索的内容。URI 还可以包含查询参数，查询参数不可以包含空格与回车换行符。在之前的代码片段中，URI 包含了几个查询参数，每个参数都通过&字符进行分隔。注意，URI 并不包含用户常常在浏览器地址栏中提供的协议、主机与端口号。客户端使用 URL 的协议部分来确定如何与服务器进行连接。客户端指定的主机名与端口是在请求的 Host 头中提供的。

请求行中的最后一个域指定了所用 HTTP 协议的版本。在之前的 HTTP 请求代码示例中，版本值是 1.1，这表示服务器期望客户端使用针对 HTTP 协议 1.1 版的头与规则。

紧跟请求行的是请求头，它向服务器提供额外的元数据。这些元数据可以描述客户端、进一步描述请求或是从服务器请求某种类型的响应。每个请求可以提供一个或多个头。Host 头是 HTTP 1.1 请求中唯一必须提供的头。它提供了客户端指定的原始主机名，还可以包含最初请求的 URL 中提供的端口号。HTTP 服务器可以为多台主机提供内容。Host 头帮助 HTTP 服务器了解最初的请求是发给哪台主机的。

说明：

HTTP 规范允许 HTTP 客户端与服务器之间的中介添加、删除、重排序以及修改 HTTP 头。因此，应用发出的请求在到达服务器时可能会出现以下这些情况：添加新的头、修改已有的头或是删除某些头。

虽然使用了有状态的 TCP 传输层，但 HTTP 却是个无状态协议。这意味着 HTTP 服务器并不会保留关于某个请求的任何信息以用在未来的请求中。Cookie 提供了一种方式，可以将一些简单的状态信息存储到客户端，并在后续的请求中与服务器进行通信。

HTTP 头之后是可选的请求体。请求体可以是任意的字节序列，通过一个空行与头分隔开来。请求体必须遵循客户端与服务器之间预先确定的数据编码。对于 Web 浏览器来说，

这通常是表单编码数据，但对于移动应用来说，该编码通常是 XML 或 JSON 数据。第 4 章“生成并解析负载”会详细分析请求体与响应体的编码。

在 iOS 中，NSURLRequest 及其子类 NSMutableURLRequest 提供了必要的方法与属性来构建几乎任何的 HTTP 请求。接下来的 3.3 节“高层 iOS HTTP API”将会介绍这些对象。

3.2.3 响应内容

在 HTTP 服务器与应用服务器处理完请求后，HTTP 响应会通过同一个 TCP socket 返回给客户端。HTTP 响应的结构类似于 HTTP 请求，第 1 行也是状态行，后面是头，然后是响应体。下述代码片段展示了一个简单的 HTTP 响应：

```
HTTP/1.1 200 OK
Date: Tue, 27 Mar 2012 12:59:18 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Transfer-Encoding: chunked
Server: gws

<!doctype html><html itemscope="itemscope"
itemtype="http://schema.org/WebPage">
<head><meta itemprop="image" content="/images/google_favicon_128.png"/>
<title>ios - Google Search</title>
<script>window.google={kEI:"prlxT5qtNqe70AHh873aAQ",
getEI:function(a){var b;
while(a&&!(a.getAttribute&&(b=a.getAttribute("eid"
```

说明：

在上述代码中，出于简洁与清晰的目的，我们省略了很多行。

状态行包含 3 个域，域之间通过空格分隔。第 1 个域是响应的 HTTP 版本。接下来的两个域提供了表示请求结果的状态值。首先是一个 3 位的整数值，包含了请求的结果代码。最后是条说明短语，提供了关于代码的简短文本说明。在大多数情况下，数值能够完全说明状态。第 5 章“处理错误”将会分析错误状态、原因以及对错误的恰当响应方式。

紧跟状态行的是响应头；响应头之间通过回车/换行符进行分隔。每个头都包含了关于响应的元数据，包括数据的上一次修改时间、客户端可以缓存数据多长时间、数据的编码方式以及在随后的请求中提交的状态信息。

响应体是通过空行与响应头分隔开来的。响应体可以包含任意数量的二进制字符。与客户端通信的响应体的长度可以通过请求的 Content-Length 头或通过块编码体现。块编码响应包含 Transfer-Encoding 头，并且带有 chunked 值。块编码体包含一个或多个体片段。每个片段都有起始行，指定块中的字节数量。iOS URL 加载系统向应用隐藏了这种复杂性。

在 iOS 的 URL 加载系统中，NSURLResponse 及其子类 NSHTTPURLResponse 封装了

请求返回的数据。该继承体系中有两个对象，因为 URL 加载可以基于非 HTTP URL 实现数据的请求。比如，对 URL file://的请求就不会包含任何头信息。

3.3　高层 iOS HTTP API

本节将会介绍 iOS 应用与 HTTP 服务器进行 HTTP 通信时经常会用到的那些高层 API。在使用 URL 加载系统时，有 3 个主要的方法可以执行 HTTP 请求和接收响应：

- 同步——启动线程的代码会阻塞，直到整个响应加载完毕并返回到调用方法为止。该技术最容易实现，不过局限性也最大。
- 队列式异步——起始代码创建一个请求，并将其放到一个队列中以在后台线程中执行。该技术的实现稍微有些难度，不过却消除了纯同步式技术的诸多限制。
- 异步——起始代码开启一个请求，该请求运行在起始线程中，不过在请求处理时会调用委托方法。该技术的实现最为复杂，不过在处理响应时却提供了最大的灵活性。

每种请求都有自己的方式和最佳实践，但所有这 3 个请求都由相同的 4 个对象构成。本节首先会介绍其中的相似性，然后通过示例深入到每一个方法中，同时会针对每种技术给出相应的指导。

3.3.1　所有请求类型共用的对象

就像便宜的墨西哥餐厅菜单一样，所有的 URL 加载请求方法会共用几个对象。这 3 类请求会共用 4 类对象：NSURL、NSURLRequest、NSURLConnection 与 NSURLResponse 对象。

1. NSURL

可以通过 NSURL 对象轻松管理 URL 值并访问 URL 指向的内容。NSURL 可以指向文件资源，也可以指向网络资源，同时在这两类资源类型的使用上没有任何区别。下述代码片段展现了如何从 URL 加载数据：

```
NSURL *url = [NSURL urlWithString:mysteryString];
NSData *data = [NSData dataWithContentsOfURL:url];
```

mysteryString 的值可以引用文件或网络资源，而代码的行为是一样的。主要的差别在于加载 mysteryString 所引用资源的时间上。如果 URL 引用的是网络资源，就会在后台线程中执行代码，这样在数据加载时用户界面就不会暂停下来。

创建 NSURL 对象最常见的方式是使用类方法 URLWithString:进行实例化。该方法会创建一个 NSURL 对象，并使用提供的 NSString 对象的内容对其进行初始化，下述代码片段说明了这一点：

```
NSURL *url = [NSURL URLWithString:@"http://www.wiley.com/path1"];
```

NSURL 对象提供了很多访问器方法来读取 URL 各个部分的值。每个访问器都可以只读访问 URL 的某一部分。scheme 访问器会返回一个包含该 URL 所用协议的 NSString 对象。如果目标 URL 没有指定某个特定部分，那么返回值就为 nil。考虑之前创建的 url 对象，下述代码片段会打印出 Port is nil。

```
if(url.port == nil) {
    NSLog(@"Port is nil");
} else {
    NSLog(@"Port is not nil");
}
```

如果 URL 包含查询字符串，那么 query 访问器方法就会包含所有查询参数的值。根据 RFC 3986 的要求，在创建 NSURL 对象前，URL 字符串的内容需要以百分号编码。比如，如果执行下述代码片段，那么查询参数的值就为 q=iOS+Networking。

```
NSURL *url = [NSURL URLWithString:@"http://google.com?q=iOS+Networking"];
```

NSURL 对象是不可变的，这意味着无法先构建空的 NSURL 对象，然后通过调用对象的赋值方法(有时也叫做 setter 方法)来装配其属性。对象要么通过 NSString 对象，要么通过另一个 NSURL 对象实例化。如果用于实例化 NSURL 对象的字符串是不合法的，那么创建方法调用就会返回 nil。在使用 URL 对象进行网络请求前，应该先验证 URL 对象是正确的。

2. NSURLRequest

创建好 NSURL 对象后，接下来就需要完成执行 HTTP 请求所需的下一步了：创建 NSURLRequest 对象。NSURLRequest 对象包含了加载 URL 内容所需的信息，并且独立于 URL 中指定的协议。iOS 中的 URL 加载系统支持 HTTP、HTTPS、FTP 与 FILE URL 内容的加载。URL 加载系统提供了一种扩展方式以处理新的协议，方式是创建 NSURLProtocol 的子类，然后将返回结果提供给 URL 加载系统。

创建 NSURLRequest 对象最简单的方式是使用类方法和提供的 NSURL 对象。下述代码片段展示了使用默认值来创建请求对象的过程：

```
NSURL *url = [NSURL URLWithString:
     @"https://gdata.youtube.com/feeds/api/standardfeeds/top_rated"];
if(url == nil) {
    NSLog(@"Invalid URL");
    return;
}
NSURLRequest *request = [NSURLRequest requestWithURL:url];
if(request == nil) {
    NSLog(@"Invalid Request");
    return;
}
```

使用默认值表示请求使用 URL 协议指定的请求缓存规则，请求有着标准的请求超时。请参见第 7 章“优化请求性能”以深入了解关于控制请求缓存的各个选项。如果 URL 是 HTTP 或 HTTPS，那么请求方法将是 GET，并且使用操作系统提供的默认头。

下述示例展示了如何使用自定义的缓存与超时值来创建 NSURLRequest 对象。构建 URL 加载系统的代码忽略了所有缓存，如果完成请求连接的时间超过 20 秒，将会生成错误。

```
NSURL *url = [NSURL URLWithString:
     @"https://gdata.youtube.com/feeds/api/standardfeeds/top_rated"];
if(url == nil) {
   NSLog(@"Invalid URL");
   return;
}
NSURLRequest *request = [NSURLRequest
                 requestWithURL:url
                    cachePolicy:NSURLRequestReloadIgnoringCacheData
                timeoutInterval:20.0];
if(request == nil) {
   NSLog(@"Invalid Request");
   return;
}
```

NSURLRequest 对象提供了几个访问器方法来获取请求的属性，由于 NSURLRequest 类是不可变的，因此无法修改这些属性。除了 URL、缓存策略与超时值外，如果要修改其他属性，那么请使用 NSMutableURLRequest 类。

NSMutableURLRequest 是 NSURLRequest 的子类，提供了赋值方法以修改请求的属性。下述代码片段展示了使用一小段消息体来创建一个简单 POST 请求，它包含了以 UTF8 编码的一个 NSString 对象的字节。URL 加载系统会自动装配请求的 Content-Length 头。

```
NSURL *url = [NSURL urlWithString@"http://server.com/postme"];
NSMutableURLRequest *req = [NSMutableURLRequest requestWithURL:url];
[req setHTTPMethod:@"POST"];
[req setHTTPBody:[@"Post body"
                        dataUsingEncoding:NSUTF8StringEncoding]];
```

有两种方式可以向 NSURLRequest 提供 HTTP 体：在内存中(就像之前的示例一样)或是通过 NSInputStream。代码可以通过输入流提供请求体而无须将整个体加载到内存中。如果发送诸如照片或视频等大容量内容，那么使用输入流是最佳选择。下述代码片段展示了如何通过输入流创建 POST 方法，需要事先将 NSString srcFilePath 设定为应用包或是沙箱中的文件路径：

```
NSMutableURLRequest *request =
[NSMutableURLRequest requestWithURL:url];
NSInputStream *inStream =
   [NSInputStream
      inputStreamWithFileAtPath:srcFilePath];
[request setHTTPBodyStream:inStream];
```

```
[request setHTTPMethod:@"POST"];
```

由于 NSURLRequest 对象包含 HTTP 与非 HTTP 请求的属性，因此访问非 HTTP URL 的代码需要将特定于 HTTP 的属性的值设为 nil。

说明：

在 iOS 6 中，NSURLRequest 有一个新属性用于标识请求是否可以通过蜂窝网络发送。使用该属性可以使应用利用第 5 章“错误处理”中介绍的 Reachability 框架，而无须显式将框架添加到应用中。

3. NSURLConnection

NSURLConnection 对象是 URL 加载系统活动的中心，但提供的接口却不多，只提供了用于初始化、开启与取消连接的方法。

回到上面提及的用于执行 HTTP 请求和获取响应的主要方法上来，NSURLConnection 类通过 3 种不同的操作模式发挥作用：同步、异步与队列式异步。同步模式是最易于使用的，不过却有很多限制，这使得它不太适合更加高级的交互。异步模式提供了很大的灵活性，不过其代价就是增加了代码的复杂性。队列式异步模式提供了异步模式的后台操作，同时又保持了同步模式的简单性。下一节将会介绍这 3 种模式。

在异步模式下操作时，NSURLConnection 对象会调用委托对象来指引连接流、处理到来的数据、处理认证并对错误做出响应。本章的 3.3.4 节“异步请求”将会深入探索 NSURLConnection 委托。

4. NSURLResponse

NSURLResponse 对象会在 URL 加载请求完毕后返回。响应对象的内容根据请求的类型及成功与否会有较大变化。如下列表介绍了从请求返回的各种对象。还有两个对象也可能来自于 URL 加载请求：NSError 对象与 NSData 对象。如果请求有问题或是客户端无法连接到服务器，就会产生 NSError 对象。请参见第 5 章以深入了解关于出错状态信息。如果有响应返回，那么生成的 NSData 对象就会包含响应体。如果生成了 NSError 对象，就不会再生成 NSURLResponse 与 NSData 对象。

- MIMEType——结果数据的 MIME 类型。该值来源于服务器，如果客户端框架认为服务器有错，那么可以修改；如果服务器没有提供，还可以由客户端框架提供。
- expectedContentLength——该值可能由请求返回，也可能不返回，返回值可能与返回内容的实际大小不同。如果返回内容的大小未知，那么该值将等于 NSURLResponseUnknownLength。
- suggestedFilename——要么是服务器提供的内容文件名，要么来自于 URL 和 MIME 类型。
- URL——返回内容的 URL。由于重定向和标准化等原因，该 URL 可能与请求提供的 URL 不同。

- textEncodingName——最初的数据源所用的文本编码名。如果响应中没有使用文本编码，那么该值将为 nil。

URL 加载系统提供了一个名为 NSHTTPURLResponse 的 NSURLResponse 子类，它包含特定于 HTTP 请求的属性。该类对于确定 HTTP 请求的结果是必需的。它有如下额外参数：

- 响应头——该属性返回响应头值的 NSDictionary 对象。字典的键是头的名字，每个键的值是头的值。HTTP 规范允许一个请求有多个同名的头。NSHTTPURLResponse 通过返回一个包含所有头值的 NSString 对象(头值之间用逗号分隔)来处理这一点。
- HTTP 状态码——来自于响应状态行的整数状态码。NSHTTPURLResponse 类有一个类方法可以针对任意状态码返回本地化的字符串说明。

接下来的几节将会通过示例来说明如何使用每一种 URL 加载系统的请求方法。除了示例外，我们还会就每一种技术的使用给出最佳实践。

3.3.2　同步请求

同步请求是 iOS 中最简单的请求类型，不过简单的代价则是缩减的功能与灵活性的降低。在发出同步请求时，请求所处的线程就会阻塞，直到请求完成或失败为止。同步请求通常用于创建 HTTP GET 请求在后台线程中获取已知大小的资源。比如，使用同步请求在后台线程中可以轻松获取图片并显示在单元格中。

说明：

本节的示例来自于本章提供的示例程序。该示例程序是个简单的 RSS 客户端程序，会从 NASA 获取 RSS 源，然后向 iOS 设备提供视频文件下载，供稍后观看。

图 3-3

示例程序 VideoDownloader 会使用同步请求下载 RSS 源的 XML 内容。图 3-3 展示了该示例程序的截屏。通过视图顶部的分隔控件可以在获取源 XML 的队列式异步请求与同步请求间切换。如果选择同步方法并按下右侧的刷新按钮，就会发现刷新按钮被冻结住了，在请求发送时应用变得没有响应。在 Wi-Fi 网络下，这个过程很快，但如果在 EDGE 网络下，延迟就很明显了。

同步请求展示了获取 URL 数据的最简单方式，在 iOS API 中有很多辅助方法的底层使用的都是同步请求。比如，NSString stringWithContentsOfURL: 方法会创建一个 NSString 实例，然后根据 URL 的内容从任意服务器获取这些内容。如果 URL 使用了 FILE(比如 file://foo.txt)协议，就会从

本地文件系统获取内容。如果 URL 使用了 HTTP(比如 http://www.wiley.com)协议，就会从远程服务器获取内容。因此，除非知道 URL 是 FILE URL，否则在使用这些辅助方法从 URL 中获取内容时务必小心。

代码清单 3-1 展示了示例应用中的 doSyncRequest 方法，它会执行一个同步请求来加载 XML 源。

代码清单 3-1 VideoDownloader/FeedLoader.m 的 doSyncRequest 方法

```
- (NSArray *) doSyncRequest:(NSString *)urlString {
    // make the NSURL object from the string
    NSURL *url = [NSURL URLWithString:urlString];

    // Create the request object with a 30 second timeout and a
    // cache policy to always retrieve the
    // feed regardless of cachability.
    NSURLRequest *request =
        [NSURLRequest requestWithURL:url
          cachePolicy:NSURLRequestReloadIgnoringLocalAndRemoteCacheData
          timeoutInterval:30.0];

    // Send the request and wait for a response
    NSHTTPURLResponse    *response;
    NSError              *error = nil;
    NSData *data = [NSURLConnection sendSynchronousRequest:request
                                         returningResponse:&response
                                                     error:&error];

    // check for an error
    if (error != nil) {
        NSLog(@"Error on load = %@", [error localizedDescription]);
        return nil;
    }

    // check the HTTP status
    if([response isKindOfClass:[NSHTTPURLResponse class]]) {
        NSHTTPURLResponse *httpResponse = (NSHTTPURLResponse *)response;
        if (httpResponse.statusCode != 200) {
            return nil;
        }
    }

    // Parse the data returned into an NSDictionary
    NSDictionary *dictionary =
        [XMLReader dictionaryForXMLData:data
                                  error:&error];
    // Dump the dictionary to the log file
    NSLog(@"feed = %@", dictionary);

    NSArray *entries =[self getEntriesArray:dictionary];
```

```
    // return the list if items from the feed.
    return entries;

}
```

代码清单3-1从包含URL的调用者中接收一个NSString对象。当URL构建完毕后，会实例化一个NSURLRequest对象。在该例中，代码重写了默认的缓存策略与超时时间。注意，这里通过NSURLRequestReloadIgnoringLocalAndRemoteCacheData将缓存策略设为永不缓存。这样可以更好地表现出同步请求对UI的影响，因为UI线程会被阻塞。通常情况下，代码不会重写所有缓存，不过重写默认的超时时间倒是比较常见的，这里为请求指定30秒的超时时间，正如代码清单3-1所示。

创建好请求后，代码会调用NSURLConnection的类方法sendSynchronousRequest:returningResponse:error来执行请求。该方法将请求和两个指针作为输入参数：一个指向NSURLResponse对象，它会由服务器的响应进行装配；另一个指向NSError对象，如果请求失败，该对象中就包含详细的错误信息。response指针指向NSURLResponse的一个实例；然而，它将是用于处理所有HTTP请求的NSHTTPURLResponse子类实例。如果NSError不为nil，那就说明请求在某个底层失败了；然而，如果为nil，那就表明请求没有因为网络错误或是不合法的URL而失败。但请求仍旧可能在语义上失败，比如服务器响应说遇到了内部服务器错误等。error指针指向的NSError对象的内容包含了关于错误的详细说明信息，第5章将会对此进行详细介绍。

代码清单3-1中的代码会检查NSError对象是否存在，以及NSHTTPURLResponse对象的状态码。如果二者都标识为成功，那么方法就会继续执行。

sendSynchronousRequest:returningResponse:error方法会以NSData对象的形式返回HTTP响应体。由于源以XML形式表示，因此成功请求的NSData对象会被XML读取器解析到NSDictionary中。接下来会遍历该字典，将RSS条目列表返回给调用者。

创建同步调用相当简单，仅需几行代码就可以成功从服务器获取数据，不过这种简单性是以有限的使用场景和增加的缺陷风险为代价的。

同步请求的最佳实践

- 只在后台线程中使用同步请求，除非确定请求访问的是本地文件资源，否则请不要在主线程上使用。
- 只有在知道返回的数据不会超出应用的内存时才使用同步请求。记住，整个响应体都会位于代码的内存中。如果响应很大，那么可能导致应用出现内存溢出问题。此外，当代码将响应解析为所需的格式时可能需要复制返回的数据，这会导致内存增加一倍。
- 在处理返回的数据前，验证错误与调用返回的HTTP响应状态码。

- 如果源 URL 需要验证，那么不要使用同步请求，因为同步框架并不支持对认证请求作出响应。唯一的例外是 BASIC 认证，因为这时认证信息可以通过 URL 或请求头进行传递。以这种方式执行认证会增加应用与服务器之间的耦合度，从而导致整个应用变得更加脆弱。如果请求不使用 HTTPS 协议，那么还会在明文中传递认证信息。请参见第 6 章“保护网络传输”以了解关于响应认证请求的更多信息。
- 如果需要向用户提供进度条，那么不要使用同步请求，因为请求是原子的，无法提供中间的进度指示信息。
- 如果需要通过流解析器来渐进解析响应数据，那么不要使用同步请求。
- 如果在请求完成前需要取消，那么不要使用同步请求。

3.3.3 队列式异步请求

队列式异步请求类似于同步请求。程序提供 NSURLRequest 对象，URL 加载系统尝试加载请求而不会与调用代码之间存在任何其他的交互。这两种方式之间的主要差别在于 URL 加载系统执行的队列式异步请求位于队列中，可能位于后台线程上。队列式异步请求的概念是在 iOS 5.0 中增加的。

iOS 提供了一种叫做操作队列的设施，名为 NSOperationQueue。这些队列可以让程序描述待执行的操作，然后以先进先出的顺序提交操作供队列执行。队列框架提供了优先级顺序以及根据操作依赖的顺序，不过 URL 加载系统并没有使用这些设施。

在代码构建队列式异步请求前，首先需要创建队列，里面是执行的请求。下述代码片段展示了如何创建操作队列：

```
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
```

操作队列可以并发执行多个操作。在默认情况下，并发操作的数量是由 iOS 根据系统情况决定的。可以通过调用创建队列的 setMaxConcurrentOperationCount:方法来重写默认值。当应用启动时，一个队列就会自动创建，可以通过调用 NSOperationQueue 的 mainQueue 类方法来获取该队列。不要使用该队列执行网络请求，因为会在主线程上操作，长时间的操作会冻结用户界面。如果从测试应用的分隔控件中选择队列选项并轻拍刷新按钮，那么会发现刷新按钮会立刻返回到默认状态，同时表视图会被清空。之所以会出现这种情况是因为请求是排队的，主运行循环会继续执行而不会等待请求完成。

代码清单 3-2 展示的方法用于创建和处理排队请求的结果。与同步请求一样，首先会创建一个 NSURL 对象，然后向其传递一个新的 NSURLRequest 对象。请求创建完毕后，如果不存在，那么代码会创建一个名为 queue 的 NSOperationQueue 对象。该变量在 FeedLoader 类的实现中被声明为静态变量。通常情况下，应用会在启动时在应用委托中创建一个队列，然后在整个应用中都使用该队列。由于知道队列已经存在，因此代码会调用 NSURLConnection 来执行队列中的请求，并在操作完成或失败后调用一个块。当请求位于队列中时，doQueueRequest:delegate 方法会返回到调用者。由于方法会在 URL 加载完毕前返回，因此当加载完毕时需要由一个委托类去调用。由于这里使用了异步完成模式，因此

代码需要实现委托或通知模式，从而将接收到的数据传递给最初的请求对象。

代码清单 3-2　VideoDownloader/FeedLoader.m 的 doQueuedRequest 方法

```
- (void) doQueuedRequest:(NSString *)urlString delegate:(id)delegate {
    // make the NSURL object
    NSURL *url = [NSURL URLWithString:urlString];

    // create the request object with a no cache policy and a 30 second timeout.
    NSURLRequest *request = [NSURLRequest requestWithURL:url
      cachePolicy:NSURLRequestReloadIgnoringLocalAndRemoteCacheData
      timeoutInterval:30.0];

    // If the queue doesn't exist, create one.
    if(queue == nil) {
        queue = [[NSOperationQueue alloc] init];
    }

    // send the request and specify the code to execute when the
    // request completes or fails.
    [NSURLConnection sendAsynchronousRequest:request
                                       queue:queue
                           completionHandler:^(NSURLResponse *response,
                                               NSData *data,
                                               NSError *error) {

            if(error != nil) {
                NSLog(@"Error on load = %@", [error localizedDescription]);
            } else {
                // check the HTTP status
                if([response isKindOfClass:[NSHTTPURLResponse class]]) {
                    NSHTTPURLResponse *httpResponse = (NSHTTPURLResponse
                                                       *)response;
                    if(httpResponse.statusCode != 200) {
                        return;
                    }
            }

            // parse the results and make a dictionary
            NSDictionary *dictionary =
                [XMLReader dictionaryForXMLData:data
                                          error:&error];
            NSLog(@"feed = %@", dictionary);

            // get the dictionary entries.
            NSArray *entries =[self getEntriesArray:dictionary];

            // call the delegate
            if([delegate respondsToSelector:@selector(setVideos:)]) {
            [delegate performSelectorOnMainThread:@selector(setVideos:)
```

```
                                                withObject:entries
                                             waitUntilDone:YES];
                }
            }
        }];
}
```

待执行的代码块通过 sendAsynchronousRequest:queue:completionHandler 方法的 completionHandler 参数传递。completion 块验证请求没有生成错误，并且 HTTP 状态码为 200，这表示成功。如果请求是成功的，那么返回的数据就会被解析到 NSDictionary 中。接下来，代码会验证所提供的委托类支持 setVideos:方法。如果支持，就会在主线程上调用该方法，并提供 RSS 源返回的条目数组。setVideos:之所以要在主线程上调用，是因为 completion 块是在后台线程上执行的。如果委托方法在该上下文中执行并操纵用户界面，那么结果就是不可预知的，大多数情况下都是不正确的。

队列式异步请求的最佳实践

- 只有在知道返回的数据不会超出应用的内存时才使用队列式异步请求。记住，整个响应体都会位于代码的内存中。如果响应很大，那么可能导致应用出现内存溢出问题。此外，当代码将响应解析为所需的格式时可能需要复制返回的数据，这会导致内存增加一倍。
- 为所有操作使用单一的 NSOperationQueue，根据服务器的能力以及预期的网络状况控制当前操作的最大数量。
- 在处理返回的数据前验证错误与调用返回的 HTTP 响应状态码。
- 如果源 URL 需要验证，那么不要使用队列式异步请求，因为该功能并不支持对认证请求做出响应。如果服务需要这种认证，那么可以将 BASIC 认证信息放在提供给请求的 URL 中。
- 如果需要向用户提供进度条，那么不要使用队列式异步请求，因为请求是原子的，无法提供中间的进度指示信息。
- 如果需要通过流解析器来渐进解析响应数据，那么不要使用队列式异步请求。
- 如果请求在完成前需要取消，那么不要使用队列式异步请求。

3.3.4 异步请求

异步请求使用与同步和队列式异步请求相同的对象，只不过又增加了另一个对象，即 NSURLConnectionDelegate 对象。

图 3-4 展示了与 HTTP 请求过程相关的委托调用序列。协议处理器在 HTTP 协议过程中处理时，会在连接的重要阶段调用委托方法。

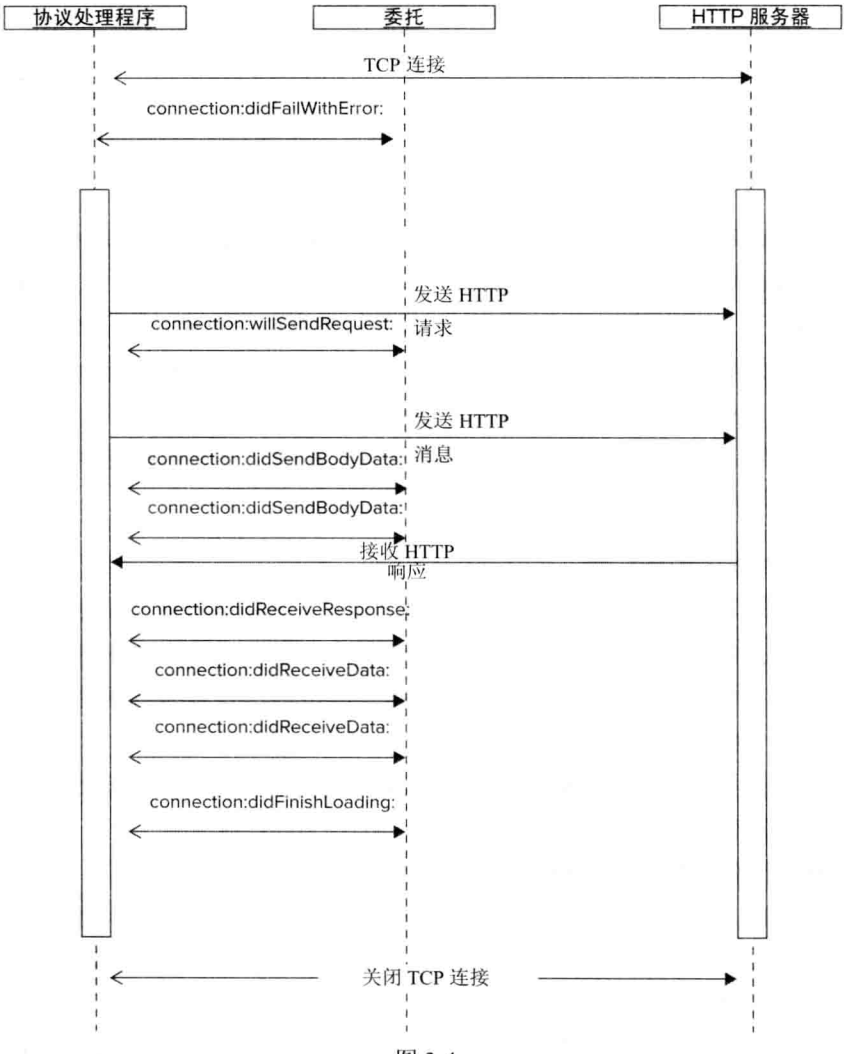


图 3-4

协议处理器在调用方法前会先验证委托实现了该方法。如果没有实现，那么协议处理器就会假定一个默认值并在连接中继续处理。说明委托方法操作的最佳方式就是尝试一下示例应用。

代码清单 3-3 包含了使用异步技术初始化 URL 加载请求的代码。一开始，该方法与之前的技术很类似：创建 NSURL 对象，然后用来构建请求。当请求创建完毕后，代码会创建 NSURLConnection 对象并将自身作为委托对象。在 URL 内容加载时，协议处理器会调用委托类并提供关于请求状态的信息。借助于这些回调，委托类可以调整协议处理器的行为。

创建好连接后，代码会开始请求。在连接创建与开启连接之间，应用可以修改委托消息传递给委托类的方式。代码可以指定不同的运行循环或操作队列来传递回调。该例并没有做这些修改。

代码清单 3-3 VideoDownloader/AsyncDownloader.m 的 start 方法

```
- (void) start {
    NSLog(@"Starting to download %@", srcURL);

    // create the URL
    NSURL *url = [NSURL URLWithString:srcURL];

    // Create the request
    NSURLRequest *request = [NSURLRequest requestWithURL:url];

    // create the connection with the target request and this
    // class as the delegate
    self.conn =
        [NSURLConnection connectionWithRequest:request
                                      delegate:self];

    // start the connection
    [self.conn start];
}
```

示例应用实现了几个委托方法供调用，同时又有几个方法没有实现。委托方法是由两个协议定义的：NSURLConnectionDelegate 与 NSURLConnectionDataDelegate。接下来的内容首先会回顾已实现的方法，然后再来看看未实现的方法。

1. connection:willSendRequest:redirectResponse:委托方法

如代码清单 3-4 所示，实现的第一个委托方法是 connection:willSendRequest:redirectResponse:。该方法只会在连接接收到 HTTP 重定向响应时得到调用。

代码清单 3-4 VideoDownloader/AsyncDownloader.m 的重定向委托方法

```
- (NSURLRequest *)connection:(NSURLConnection *)connection
             willSendRequest:(NSURLRequest *)request
            redirectResponse:(NSURLResponse *)redirectResponse {

    // Dump debugging information
    NSLog(@"Redirect request for %@ redirecting to %@",
        srcURL, request.URL);
    NSLog(@"All headers = %@",
         [(NSHTTPURLResponse*) redirectResponse allHeaderFields]);

    // Follow the redirect
    return request;
}
```

如果协议处理器接收到来自服务器的重定向请求，就会调用该方法。HTTP 重定向指的是这样一种 HTTP 响应，它们通知客户端要寻找的内容位于另一不同的 URL 中。如果应

用从内容分发网络加载内容，那么重定向请求就是很常见的。该委托方法总是在其他传递响应或体数据的方法前得到调用。由于请求可以执行多个重定向，因此对于单个请求来说，该方法可能不会被调用，也可能会被调用多次。如果委托没有实现该方法，那么协议处理器就会将重定向转向新的位置。通过实现该方法，代码可以拦截重定向，并且根据重定向的特点终止连接或修改请求。在该例中，代码会执行调试功能，将重定向请求头信息的日志打印出来，然后再执行重定向。

2. connection:didReceiveResponse:委托方法

实现的第2个方法是connection:didReceiveResponse:，如代码清单3-5所示。在协议处理器从响应头构建出响应对象后，会调用该方法。

代码清单3-5　VideoDownloader/AsyncDownloader.m的响应接收委托方法

```
- (void) connection:(NSURLConnection *)connection
didReceiveResponse:(NSURLResponse *)response {
    NSLog(@"Received response from request to url %@", srcURL);

    NSHTTPURLResponse *httpResponse = (NSHTTPURLResponse *)response;
    NSLog(@"All headers = %@", [httpResponse allHeaderFields]);

    if(httpResponse.statusCode != 200) {// something went wrong,
                                        //abort the whole thing
        // reset the download counts
        if(downloadSize != 0L) {
            [progressView addAmountToDownload:-downloadSize];
            [progressView addAmountDownloaded:-totalDownloaded];
        }
        [connection cancel];
        return;
    }

    NSFileManager *fm = [NSFileManager defaultManager];

    // If we have a temp file already, close it and delete it
    if(self.tempFile != nil) {
        [self.outputHandle closeFile];

        NSError *error;
        [fm removeItemAtPath:self.tempFile error:&error];
    }

    // remove any pre-existing target file
    NSError *error;
    [fm removeItemAtPath:targetFile error:&error];

    // get the temporary directory name and make a temp file name
    NSString *tempDir = NSTemporaryDirectory();
```

```
    self.tempFile = [tempDir stringByAppendingPathComponent:
    [self createUUID]];

    NSLog(@"Writing content to %@", self.tempFile);

    // create and open the temporary file
    [fm createFileAtPath:self.tempFile contents:nil attributes:nil];
    self.outputHandle = [NSFileHandle fileHandleForWritingAtPath:
                                                    self.tempFile];

    // prime the download progress view
    NSString *contentLengthString = [[httpResponse allHeaderFields]
                                     objectForKey:@"Content-length"];
    // reset the download counts
    if(downloadSize != 0L) {
        [progressView addAmountToDownload:-downloadSize];
        [progressView addAmountDownloaded:-totalDownloaded];
    }
    downloadSize = [contentLengthString longLongValue];
    totalDownloaded = 0L;

    [progressView addAmountToDownload:downloadSize];
}
```

当协议处理器接收到足够的数据来创建URL响应对象时会调用didReceiveResponse方法。如果在接收到足够的数据来构建响应对象前出现了错误，就不会调用该方法。在示例代码中，委托方法会验证响应对象的HTTP状态。如果状态不是200，那么请求的加载就会被取消，提供下载进度的视图会被重置。如果状态是200，那么代码会更新进度视图，方式是将所需的数据量加起来，然后创建临时文件来接收HTTP响应体，临时文件稍后会被传递给另一个委托方法。

该方法可能会被协议处理器调用多次；因此，代码必须处理重新开始这一场景。在示例中，重新开始逻辑包括重置进度指示器。如果之前响应的临时文件存在，那么还需要将其删除。

3. connection:didReceiveData:委托方法

下一个实现的委托方法是connection:didReceiveData:，如代码清单3-6所示。当协议处理器接收到部分或全部响应体时会调用该方法。该方法可能不会调用，也可能会调用多次，并且调用总是跟在最初的connection:didReceiveResponse:之后。如果需要渐进地解析响应，那么流处理器应该充分使用该方法。

代码清单3-6 VideoDownloader/AsyncDownloader.m的didReceiveData方法

```
- (void)connection:(NSURLConnection *)connection
    didReceiveData:(NSData *)data {
    // figure out how many bytes in this chunk
```

```
    totalDownloaded+=[data length];

    // Uncomment the following lines if you want a packet by
    // packet log of the bytes received.
    NSLog(@"Received %lld of %lld (%f%%) bytes of data for URL %@",
          totalDownloaded,
          downloadSize,
          ((double)totalDownloaded/(double)downloadSize)*100.0,
          srcURL);

    // inform the progress view that data is downloaded
    [progressView addAmountDownloaded:[data length]];

    // save the bytes received
    [self.outputHandle writeData:data];
}
```

该方法可能不会被调用，也可能会被调用多次，这取决于响应体的大小。在每次调用时，协议处理器会在data参数中传递部分体数据。该委托方法负责聚集所提供的数据对象，然后处理它们或是将其存储起来。所提供的数据块大小可能与应用协议的语法边界不一致。换句话说，如果代码接收的是XML文档，那么数据对象可能与文档中的元素边界不一致。在示例中，应用首先将connection:didReceiveResponse:方法中接收到的字节追加到数据文件中。

4. connection:didFailWithError:委托方法

如代码清单3-7所示，当连接失败时会调用这个委托方法。该方法可能会在请求处理的任何阶段得到调用。

代码清单3-7　VideoDownloader/AsyncDownloader.m的didFailWithError方法

```
- (void)connection:(NSURLConnection *)connection
  didFailWithError:(NSError *)error {
    NSLog(@"Load failed with error %@",
          [error localizedDescription]);

    NSFileManager *fm = [NSFileManager defaultManager];

    // If you have a temp file already, close it and delete it
    if(self.tempFile != nil) {
        [self.outputHandle closeFile];

        NSError *error;
        [fm removeItemAtPath:self.tempFile error:&error];
    }

    // reset the progress view
    if(downloadSize != 0L) {
```

```
        [progressView addAmountToDownload:-downloadSize];
        [progressView addAmountDownloaded:-totalDownloaded];
    }
}
```

如果被调用，那么该方法将是为该连接调用的最后一个方法。示例应用只是在连接失败时打印出错误消息。如果临时下载文件存在，那么将会关闭临时文件，并调整进度指示器以反映出中断的下载。一旦该方法返回，协议处理器将取消请求。这个方法很适合分析器采用，这样就可以对应用调用的端点失败率做出定量度量。

5. connectionDidFinishLoading 委托方法

如代码清单 3-8 所示，该例实现的最后一个委托方法是 connectionDidFinishLoading。当整个请求完成加载并且接收到的所有数据都被传递给委托后，就会调用该委托方法。

代码清单 3-8 VideoDownloader/AsyncDownloader.m 的 connectionDidFinishLoading 方法

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection {
    // close the file
    [self.outputHandle closeFile];

    // Move the file to the target location
    NSFileManager *fm = [NSFileManager defaultManager];
    NSError *error;
    [fm moveItemAtPath:self.tempFile
                toPath:self.targetFile
                 error:&error];
    // Notify any concerned classes that the download is complete
    [[NSNotificationCenter defaultCenter]
    postNotificationName:kDownloadComplete
    object:nil
    userInfo:nil];
}
```

该方法是为连接调用的最后一个方法，并且此调用与 connection:didFailWithError:的调用是互斥的。示例应用会将聚集所有接收到数据的文件关闭掉，根据最初请求的 URL 将文件移到某个位置，然后通过 NSNotificationCenter 通知视图控制器下载完毕。

连接委托还可以实现其他几个方法来增加可用信息并实现对连接的控制。本节的剩余内容将会介绍这些方法。

6. connection:needNewBodyStream:委托方法

该方法是可选的，只是用于重新向请求体的输入流发出请求。如果协议处理器由于出现错误或是认证等原因需要重新传递请求体就会调用该方法。如下面的代码片段所示，该方法签名会接收 NSURLConnection 对象(用于请求新的数据流)以及触发该委托回调的 request：

```
- (NSInputStream *)connection:(NSURLConnection *)connection
needNewBodyStream:(NSURLRequest *)request;
```

该委托应该返回一个未打开且自动释放的 NSInputStream 对象，该对象引用的是待上传的文件。如果代码没有实现该方法并且使用输入流来引用请求体，那么协议处理器在开始请求前首先会创建整个输入流的副本。如果代码发送一个大文件，那么应该实现该方法以避免复制文件的代价。

7. connection:didSendBodyData:totalBytesWritten:totalBytesExpectedToWrite:委托方法

如下代码片段所示，该可选的委托方法向委托对象提供了上传进度信息：

```
-(void)connection:(NSURLConnection *)connection
          didSendBodyData:(NSInteger)bytesWritten
        totalBytesWritten:(NSInteger)totalBytesWritten
  totalBytesExpectedToWrite:(NSInteger)totalBytesExpectedToWrite;
```

协议处理器会在不确定的时间间隔内调用该委托以报告上传进度。bytesWritten与totalBytesWritten值可能不会一直增加，这是因为如果出现错误或是认证问题，就需要重新传递请求体。如果希望向用户提供上传进度指示器，那么应该实现该方法。

8. connection:willCacheResponse:委托方法

如下代码片段所示，该可选方法向委托提供了一种方式来检测与修改协议控制器所缓存的响应：

```
-(NSCachedURLResponse *)connection:(NSURLConnection *)connection
       willCacheResponse:(NSCachedURLResponse *)cachedResponse;
```

NSCachedURLResponse 对象包含了 NSURLResponse 对象与请求返回来的以 NSData 形式存在的数据。该对象还包含了响应保持所需的存储策略，包括持久化存储、仅内存存储及不允许存储。缓存下来的响应对象还包含目录 userInfo，可被应用用来存储缓存请求的元数据。如果该委托实现返回 nil，那么响应就不会缓存下来。

9. 认证委托方法

有 5 个委托方法与 URL 请求的客户端认证有关。这些方法的使用将在第 6 章进行详细介绍。

10. 异步请求与运行循环

异步请求需要运行循环。当数据传递到服务器或是被客户端接收时，运行循环用于实现事件与委托对象之间的通信。异步请求在发出时，会在当前线程的运行循环上操作。这个实现细节是很重要的，因为在 Grand Central Dispatch 块中或是通过 NSOperationQueue 创建的线程并没有运行循环。因此，如果在后台线程上发出了异步请求，那么还需要确定线程是有运行循环还是使用了别的运行循环。如下代码片段展示了如何显式地将请求处理

指定给运行循环：

```
NSURLConnection connection = [[NSURLConnection alloc]
                              initWithRequest:request
                                     delegate:self
                             startImmediately:NO];

[connection scheduleInRunLoop:[NSRunLoop mainRunLoop]
                      forMode:NSDefaultRunLoopMode];
[connection start];
```

第一个操作创建了 NSURLConnection 对象，不过并没有立刻启动方法，这样就可以进一步初始化了。下一行代码获取到主线程的运行循环，然后将它提供给连接，作为其运行循环。最后，连接通过 start 方法开始处理。如果不想在主运行循环中执行异步请求，那么需要在另一个线程上创建运行循环，然后针对这个新创建的运行循环调度连接。

异步请求的最佳实践

- 对于大的上传或下载来说，请使用异步请求以减少应用的内存占用量。
- 在需要认证的情况下请使用异步请求。
- 如果需要向用户提供进度反馈，那么请使用异步请求。
- 在后台线程上使用异步请求时要小心，请提供一个运行循环。
- 对于可以在后台线程的请求队列中轻松调度和完成的简单请求来说，这时使用异步请求有些过犹不及。
- 如果使用输入流来上传数据，请实现 connection:newBodyStream:方法以避免对输入流的复制。

3.4 高级 HTTP 操作

HTTP 头在提供可修改服务器响应的元数据以及向 HTTP 客户端提供额外信息方面扮演着重要的角色。基于这一点，iOS 开发者常常需要操纵请求头或是查看响应头。比如，有些服务器需要自定义的认证头来提供关于用户身份的信息。标准的 URL 加载系统并不会自动添加这些头，不过它提供了通过代码添加的方法。

本节将会介绍更多的 HTTP 操作以及可以通过 iOS 的 URL 加载系统执行的操作。并且将会分别介绍如何创建与使用其他的 HTTP 请求方法、如何处理 HTTP Cookie 以及关于 HTTP 头的高级用法。

3.4.1 使用请求方法

本章一开始引用了表 1-1 中介绍的几个额外的 HTTP 方法。最常用的请求类型是 GET 请求，不过它有时会被滥用，在需要诸如 PUT、HEAD 或 DELETE 等方法的场合下使用了 GET。

根据定义，GET 请求不应该包含 HTTP 体，而只应该包含请求行与请求头。HTTP 服务器会返回由 URL 指定的资源内容。网络设备常常会假定 GET 请求完整的上下文位于请求行中，并根据这些数据来缓存响应。如果 GET 请求包含会修改请求所返回内容的请求体，那么由于中间网络设备的缓存行为，你可能会得到错误的结果。根据约定，GET 请求不应该导致服务器上的数据发生任何变化。

HTML 浏览器最早通过 POST 请求来发送或提交 HTML 表单，使用的是一种特定的数据编码，通过 application/x-www-form-urlencoded 这种 Content-Type 来指定。iOS 应用通常都会使用 POST 请求向服务器发送 XML 或 JSON 数据。下述代码片段展示了如何创建 JSON 对象并将其作为请求体：

```
NSError *error;
NSDictionary *dict =
    [NSDictionary dictionaryWithObjectsAndKeys:
                    @"dog", @"animal",
                    @"fido", @"name",
                    @"20", @"weight", nil];
NSData *jsonData = [NSJSONSerialization
                   dataWithJSONObject:dict
                   options:NSJSONWritingPrettyPrinted
                   error:&error];

if(error != nil) { // encoding succeeded
    NSLog(@"Error on encoding dictionary");
    return;
}
NSLog(@"Json = %@", [[NSString alloc]
                   initWithData:jsonData
                   encoding:NSUTF8StringEncoding]);
NSMutableURLRequest *request = [NSMutableURLRequest
                                    requestWithURL:url];
[request setHTTPMethod:@"POST"];
[request setHTTPBody:jsonData];
```

上述代码首先创建了一个简单的 NSDictionary 对象，里面放置了一些虚构出来的动物的名值对。然后通过内建的 JSON 库，创建一个 NSData 对象来表示该字典。接下来将该 NSData 对象提供给 NSMutableURLRequest 作为请求体。该代码产生的 JSON 如下所示(第 4 章将会详细介绍构建与解析请求和响应负载的过程)：

```
{
    "weight" : "20",
    "animal" : "dog",
    "name" : "fido"
}
```

使用 HEAD 方法的请求会指示 HTTP 服务器只返回关于所请求资源的 HTTP 头信息。HEAD 请求通常没有请求体，也没有响应体返回。它们常常用于验证缓存的数据与服务器

上的数据，同时又不必获取缓存资源的整个内容。

PUT 请求类似于 POST，因为它总是有请求体，但从语义上来说，两者有如下重要差别：PUT 请求用于向服务器添加新的资源，而 POST 只用于更新服务器上的资源。在使用 RESTful 服务时，这种语义上的差别是非常重要的，第 4 章将会介绍这一点。

3.4.2 操纵 Cookie

Cookie 是 HTTP 协议在首个版本之后加入的一个重要组件。它向服务器提供了追踪会话状态的能力，同时又无须维持客户端与服务器之间的连接。在浏览器客户端，Cookie 值是由服务器通过请求提供的，然后被放到随后的请求中。由于设计 Cookie 的目的是追踪会话状态，因此它们通常都会非常小，基本上是几十到几百个字节。

从服务器发送的 Cookie 有几个属性用于确定 Cookie 值、何时返回到服务器以及客户端应该保留 Cookie 的时间。这些属性有：

- name——Cookie 的名字，从同一 DNS 域返回的所有 Cookie 名都是唯一的。只有 name 和 value 这两个属性才会在后续的请求中发送给服务器。
- value——由向服务器发送的下一请求返回的值。
- domain——后续请求在 Cookie 中包含的 DNS 域。比如，拥有域值.domain1.com 的 Cookie 不应该返回给.domain2.com。如果省略掉，那么客户端就会将 URL 的主机名当作域。如果域的最前面是个圆点(.)，那么 Cookie 就会返回给发送到该域及其子域的任何请求。如果没有最前面的圆点，那么 Cookie 就只会包含在发送给该域而非其子域的请求中。
- path——path 限制发送给请求的 Cookie 都是针对指定的 URL 路径。如果与 DNS 域搭配使用，那么 path 属性就可以限制只会将 Cookie 发送给服务器上有限且精确的 URL 集合。
- Expiration Date——Cookie 不再随请求发送的日期与时间，Cookie 会在这个时间点从客户端存储中删除。
- Session Only——指定 Cookie 是在当前浏览器会话时间内返回还是一直持续到过期日期，以二者之间先到的时间为准。在 iOS 应用中，会话指的是 OS 加载应用到终止应用之间的应用生命周期。
- Secure——指定 Cookie 只会用在 HTTPS 连接而非 HTTP 连接上。
- Comment——用于向用户说明 Cookie 目的的注释值。
- Comment URL——URL 值，向用户提供了一个 HTML 文档，用于说明 Cookie 的目的。
- HTTP Only——指示器，告诉客户端不要与 JavaScript 应用共享 Cookie 以防止跨站脚本攻击。
- Version——Cookie 遵循的 HTTP Cookie 规范版本。

虽然不是浏览器，但 iOS 应用依然可以在 HTTP 连接中方便地使用 Cookie。URL 加载

框架帮我们做了大量繁杂的工作以利用协议的这个特性。下面是应用经常要使用到Cookie的3个地方：检索Cookie值、显式删除Cookie以及手工将Cookie添加到请求中。

URL加载设施会自动处理所有HTTP与HTTPS请求的Cookie。会将返回的Cookie保存在响应中，然后按照Cookie处理规则将其添加到随后的请求中。只有在Cookie的domain属性所提供的DNS域中的主机才会返回Cookie。方便的是，非会话Cookie值会在应用启动过程中进行持久化。因此，如果应用检索到Cookie，然后又终止了，那么检索到的Cookie依然会出现在应用随后的调用中。URL加载系统只会在Cookie没有过期并且针对目标域是有效的情况下才会发送Cookie。

URL加载系统提供了两个重要对象以管理Cookie：NSHTTPCookie与NSHTTPCookieStorage。NSHTTPCookie通过所有必要以及可选属性来表示Cookie。NSHTTPCookieStorage则是单例对象，用于管理应用的Cookie。注意，与所有其他的应用数据一样，NSHTTPCookieStorage Cookie也是沙箱的，无法在应用间共享。

NSHTTPCookieStorage可以控制保存哪些Cookie，不过在默认情况下，会存储响应中返回的所有Cookie，无论Cookie的域是否匹配请求的域。通过使用Cookie接受策略，代码可以控制保存Cookie的程度。存储策略值如下所示：

- NSHTTPCookieAcceptPolicyAlways——这是默认值，表示任何返回的Cookie都应该被保存下来。
- NSHTTPCookieAcceptPolicyNever——这个值表示不应该存储Cookie。
- NSHTTPCookieAcceptPolicyOnlyFromMainDocumentDomain——该策略告诉NSHTTPCookieStorage对象只保存域值与请求域相匹配的Cookie。

如下代码片段会阻止应用保存Cookie。

```
[[NSHTTPCookieStorage sharedHTTPCookieStorage]
setCookieAcceptPolicy:NSHTTPCookieAcceptPolicyNever];
```

代码还可以对每个请求停止使用自动化的Cookie处理，这是通过调用setHTTPShouldHandleCookies:，同时在提交NSMutableURLRequest前为其指定值NO来实现的。这样可以阻止URL加载系统处理返回的请求。

1. 从响应中获取Cookie

从响应中获取Cookie，然后根据名字查找特定的Cookie是很常见的事情。如下代码片段展示了如何创建请求，然后获取由JavaEE Web服务器返回的会话ID：

```
NSMutableURLRequest *req =  [NSMutableURLRequest
                             requestWithURL:url];

NSHTTPURLResponse *response;
NSError *error;

// make a request
NSData *data = [NSURLConnection
```

```
                    sendSynchronousRequest:req
                        returningResponse:&response
                                    error:&error];

// get the full set of headers and display them
NSDictionary *headers = [response allHeaderFields];
NSLog(@"Headers = %@", headers);

// extract the set-cookie headers and split them into cookies
NSArray *cookies = [NSHTTPCookie cookiesWithResponseHeaderFields:headers
                                forURL:url];

// look for the cookie with the name JSESSIONID
for(NSHTTPCookie *cookie in cookies) {
    NSLog(@"Cookie: %@", cookie);
    if([[cookie name] isEqualToString:@"JSESSIONID"]) {
        NSLog(@"Found the session id");
    }
}
```

请求模式与其他请求一样。response 对象声明为 NSHTTPURLResponse，后面无须再进行类型转换。allHeaderFields 方法会返回一个字典，里面包含着所有头的名字与值。如果某个头在响应中出现过多次，那么其值会拼接起来并使用逗号分隔。cookiesWithResponseHeaderFields 会返回一个 NSHTTPCookie 对象的数组，其中的每个对象都包含在 SetCookie:头中找到的 Cookie。记住，这个 Cookie 集合可能与提供给下一个请求的 Cookie 并不完全一致。在下一次请求前有些 Cookie 可能会过期，而 NSHTTPCookieStorage 还可能包含了将会添加到下一个请求的新的 Cookie。

2. 删除 Cookie

有时，你可能需要从持久化 Cookie 存储中删除 Cookie，以便 Cookie 不会再被放到发出的请求中。由于 Cookie 是在通过异步的 start 或 sendSynchronousRequest 方法开启连接之后被添加到请求中的，因此无法阻止已经位于 Cookie 存储中的 Cookie 被添加到请求中。相反，代码首先需要在开始请求前从应用的 NSHTTPCookieStorage 对象中删除 Cookie。

如下代码片段展示了如何从 Cookie 存储中删除所有 Cookie：

```
- (void)deleteAllCookies
{
    // get the shared cookie jar
    NSHTTPCookieStorage *jar = [NSHTTPCookieStorage
                                sharedHTTPCookieStorage];
    // get all the cookies
    NSArray *storedCookies = [jar cookies];

    // delete them all
    for(NSHTTPCookie *cookie in storedCookies) {
        [jar deleteCookie:cookie];
```

```
    }
    [[NSUserDefaults standardUserDefaults] synchronize];
}
```

方法的前两行代码获取到应用的单例Cookie存储对象。接下来的一行代码获取到由所有存储的Cookie构成的一个NSArray对象。接下来的循环会遍历数组中的每个Cookie并将其删除。

代码可以组合使用Cookie属性，有选择地删除Cookie。如下代码片段会针对特定的URL，根据名字来删除Cookie。记住，对于目标域来说，Cookie的名字是唯一的。

```
- (void)deleteCookie:(NSString *)cookieName url:(NSURL *)url {
    // get the shared cookie storage object
    NSHTTPCookieStorage *jar = [NSHTTPCookieStorage
                                sharedHTTPCookieStorage];

    // get the cookies for the supplied URL
    NSArray *storedCookies = [jar cookiesForURL:url];

    // iterate over the list of cookies
    for(NSHTTPCookie *cookie in storedCookies) {

        // if the cookie name matches the target, delete it
        if([cookieName isEqualToString:[cookie name]]) {
            NSLog(@"Deleting cookie %@", cookie);
            [jar deleteCookie:cookie];
        }
    }
}
```

代码使用提供的URL来过滤从Cookie存储返回的Cookie集合。当Cookie列表返回后，方法会遍历列表以查找与某个名字匹配的Cookie，如果找到，就将其删除。

deleteCookie:url:方法接收两个参数，分别是待删除的Cookie名字以及URL(请求发出后返回Cookie的URL)。确定针对某个域返回哪些Cookie会考虑该域的全局cookies以及特定主机的Cookie。表3-2展示了域与子域的映射规则。

表3-2 Cookie域映射

Cookie域	Cookie是否随请求提交到http://www.host.com	Cookie是否随请求提交到http://host.com
.host.com	是	是
.www.host.com	是	否
host.com	否	是
.www.nothost.com	否	否

如果请求发给http://www.host.com，那么域为.host.com或.www.host.com的Cookie就

会被添加到请求中。如果请求发给 http://host.com，那么域为.host.com 的 Cookie 就会被添加到请求中，不过域为.www.host.com 的 Cookie 则不会被添加进去。

3. 创建 Cookie

我们可以创建 Cookie 并以编程的方式添加到请求或 Cookie 存储中。如果想要手工创建 Cookie 以遵循 Web 应用的独有协议，那么这就是十分必要。有时还需要接收域的 Cookie，然后创建相同的 Cookie 并发送给另一个域。本节将会介绍两种实现方法。

第一个代码片段会创建一个 Cookie，然后将其添加到请求中。

```
// create a dictionary of properties
NSDictionary *properties = [NSDictionary
                            dictionaryWithObjectsAndKeys:
                            @"FOO",NSHTTPCookieName,
                            @"This is foo", NSHTTPCookieValue,
                            @"/", NSHTTPCookiePath,
                            url, NSHTTPCookieOriginURL,
                            nil];
// using the properties make a cookie
NSHTTPCookie *cookie = [NSHTTPCookie
                        cookieWithProperties:properties];

// create a mutable request
NSMutableURLRequest req = [NSMutableURLRequest
                           requestWithURL:url];

// make an array of 1 cookie
NSArray *newCookies = [NSArray arrayWithObject:cookie];

// make a dictionary of the headers required to send the cookie
NSDictionary *newHeaders = [NSHTTPCookie
                       requestHeaderFieldsWithCookies:newCookies];
// make the cookie headers the headers of the request
[req setAllHTTPHeaderFields:newHeaders];

// send the request
[NSURLConnection sendSynchronousRequest:req
                       returningResponse:&response
                                   error:&error];
```

代码的第一部分创建了一个字典，里面包含了新 Cookie 的属性，其中的属性名都是 NSHTTP 常量。我们通过该字典(代码中叫做 properties)来实例化新的 NSHTTPCookie。接下来使用默认属性来创建一个 NSMutableURLRequest 对象。

接下来的代码要创建出添加到请求中的正确结构。首先创建一个 NSArray 对象，里面包含着一个新的 Cookie。然后通过该数组创建一个 NSDictionary 对象，头信息会被放到 HTTP 请求中以表示 Cookie。接下来会使用包含 Cookie 信息的新头替换掉默认的头内容。

如果代码需要添加其他头信息，那么需要在调用 setAllHTTPHeaderFields:之前将其添加到 newHeaders 字典中。

如下代码所示，将 Cookie 添加到请求中的第二种方式更加简洁，不过这会将 Cookie 添加到随后所有的请求中：

```
// create the properties for a cookie
NSDictionary *properties = [NSDictionary
                            dictionaryWithObjectsAndKeys:
                            @"FOO",NSHTTPCookieName,
                            @"This is foo", NSHTTPCookieValue,
                            @"/", NSHTTPCookiePath,
                            url, NSHTTPCookieOriginURL,
                            nil];
// create the cookie from the properties
NSHTTPCookie *cookie = [NSHTTPCookie
                        cookieWithProperties:properties];

// add the cookie to the cookie storage
[[NSHTTPCookieStorage sharedHTTPCookieStorage] setCookie:cookie];

// create and issue a request
req = [NSMutableURLRequest requestWithURL:url];
[NSURLConnection sendSynchronousRequest:req
                      returningResponse:&response
                                  error:&error];
```

与上一个示例一样，代码首先也要创建 Cookie，然后将其存储到应用的 Cookie 存储中。请记住，如果 Cookie 存储接受策略有限制，那么 Cookie 可能不会被存储进去。存储完 Cookie 之后，新的请求在开始执行时会自动将其从 Cookie 存储中取出来。

3.4.3　头信息操作进阶

除了操纵请求体与随请求发送的 Cookie 外，我们常常还会向请求添加头，或是从请求中删除头以及查看响应中的头信息。

下面将会介绍如何操纵 HTTP 请求与响应头，并谈到为何说管理从应用中发出的头信息是非常重要的。

1. 添加请求头

当代码需要修改请求头时，需要创建 NSMutableURLRequest 对象而非不可变的 NSURLRequest 对象。NSMutableURLRequest 类提供了两种方式来向请求添加头：一次一个头以及替换所有头。

setAllHTTPHeaderFields:方法提供了一种方式以通过一次调用替换掉所有的请求头。之前的 Cookie 示例就通过该方法向请求添加 Cookie。如果代码需要根据动态的数据集来添加头，那么该方法就很有用了。如果数据集位于字典中，那么只需通过一次调用即可将

其添加到请求中。

还可以将头逐个添加到请求中。如下代码片段展示了如何向请求添加头：

```
NSMutableURLRequest *req = [NSMutableURLRequest requestWithURL:url];
[req addValue:@"en" forHTTPHeaderField:@"Content-Language"];
[req addValue:@"da" forHTTPHeaderField:@"Content-Language"];
```

第一行代码实例化了一个名为 req 的请求对象。第二行代码向请求添加了一个头，该头的名字为 Content-Language，值为 en。HTTP 协议规定头的名字要以冒号结束，不过如果省略的话，该方法会附加一个冒号。如果相同的头名被添加多次，那么头的值就是多个值的拼接并使用逗号分隔。在上述示例中，最后生成的头如下所示：

```
Content-Language: en,da
```

2. 删除请求头

有时，需要从请求中删除头。比如，如果应用想要禁用对返回数据的压缩，那么可以覆写默认的 iOS 头(默认的头支持 gzip 或 DEFLATE 压缩)：

```
Accept-Encoding: gzip, deflate
```

这会告诉服务器：可以对返回的响应体进行压缩。如下代码覆写了请求的 Accept-Encoding 头：

```
NSMutableURLRequest *req = [NSMutableURLRequest requestWithURL:url];
[req setValue:@"" forHTTPHeaderField:@"Accept-Encoding"];
```

URL 加载系统 API 并未提供完全删除请求中标准头的方式。最好的方式就是使用空值覆写默认值。

3. 查看响应头

当 HTTP 请求完成且没有错误时，可能不包含头，也可能包含多个头。如下代码片段展示了对于成功完成的 HTTP 请求，如何将头信息存储到日志文件中并获取到响应的 MIME 类型：

```
NSHTTPURLResponse *response;
NSError *error;
[NSURLConnection sendSynchronousRequest:req
                      returningResponse:&response
                                  error:&error];
NSDictionary *headers = [response allHeaderFields];
NSLog(@"Headers = %@", headers);

NSString *contentType = [headers objectForKey:@"Content-Type"];
```

如果请求 URL 使用 HTTP 或 HTTPS 协议，那么这么做就是没问题的。allHeaderFields 方法会返回由所有响应头构成的字典，其中包括 Set-Cookie 头。最后一行代码会获取到

Content-Type 头的值。如果响应中省略了头，那么返回值就是 nil。如果响应中包含了相同类型的多个头，那么只会返回一个值。这个返回值是由所有响应值拼接而成的，并使用逗号分隔。

4. 主要的请求头

有几个 HTTP 头使用的比较多。下面将介绍这些头的使用情况。

很多 REST 服务器实现会通过检查 Accept 头来确定对返回负载的数据编码。比如，值为 application/xml 的 Accept 头会告诉服务器返回 XML 文档，而 application/json 则会返回 JSON 文档。

Authorization 头可以与认证证书一起来避免对来自服务器的响应进行认证检查。如下代码片段展示了如何添加 BASIC 认证头：

```
NSString *basicBody = [NSString stringWithFormat:@"%@:%@",
                       username, password];
NSData *authData = [basicBody
                       dataUsingEncoding:NSASCIIStringEncoding];

// Base64 encode authData into a string called b64Data
// code omitted to perform the Base64 encoding

NSString *authValue = [NSString stringWithFormat:@"Basic %@",
                       b64Data];

[theRequest setValue:authValue
  forHTTPHeaderField:@"Authorization"];
```

上述代码创建了名为 basicBody 的认证数据体，其中包含用户名与密码并使用冒号分隔。接下来将 basicBody 转换为 NSData 对象，其中包含字符串的 ASCII 值。然后将其以 Base64 编码为名为 b64Data 的字符串。然后，将单词 Basic 放在该字符串的前面，作为 Authentication 头的值。

你可能已经知道，BASIC 认证的安全性不如 Base64 编码的密码，这意味着只能在 HTTPS 连接上使用这种方式。该例并未使用 Base64 编码，不过网络上有很多库实现了 Base64 编码数据。第 11 章“应用间通信”进一步介绍了 Base64 编码。

另一个有必要修改的头是 User-Agent，创建它的目的是为 HTTP 服务器识别出浏览器类型。在很多企业网络中，user agent 值用于根据浏览器类型将访问转向特定的服务器。有些 HTTP 服务器会根据 user agent 值修改响应内容。在默认情况下，来自应用的 user agent 会包含应用产品名与版本、网络框架与版本，以及 OS 名与版本。VideoDownloader 应用的 User-Agent 值如下所示：

```
VideoDownloader/1.0 CFNetwork/548.1.4 Darwin/11.0.0
```

应用的 User-Agent 头值会根据设备 iOS 版本的不同而不同，当运行在 iOS 模拟器中时

也会不同。如果网络基础设施会根据 user agent 进行路由，那么它必须能处理这些差别才行。处理这些差别有两种方式：

- 覆写 User-Agent 头以在各种平台上都提供相同的值。
- 根据 User-Agent 头的子集进行路由，比如只根据应用名。头的剩余部分会根据应用的版本以及所运行的设备而发生变化。

代码并非只能对请求使用标准的头。可以使用几乎任何头名来自定义头；然而，这些自定义头名不能包含冒号或空格。头名区分大小写，需要与 Web 服务开发者协调来确定准确的头名与值。自定义的头有助于传输认证与控制数据，应用基础设施可以通过它们来支持应用的业务逻辑。比如，通过自定义的头提供会话标识符，这样应用服务器就可以在将请求负载发送给应用逻辑前将会话上下文与其关联起来。

将自定义的头添加到请求中的方式与标准头一样，如下代码片段所示：

```
NSMutableURLRequest *req = [NSMutableURLRequest requestWithURL:url];
[req setValue:@"myValue" forHTTPHeaderField:@"My-Custom-Header"];
```

3.5 小结

HTTP 请求是想要连接到企业服务的应用开发人员采用的主要手段。iOS 提供了一个简单却健壮的 API 来执行绝大多数的请求，无须探究底层 API。虽然最初是为浏览器创造的，不过 HTTP 协议是非常灵活的，足以适合应用与远程服务器之间的几乎任何请求与响应交互。

借助 URL 加载系统与 NSURLRequest、NSURLResponse 和 NSURLConnection，只需几行代码就可以发出请求。随着通信变得越来越复杂，可以继续使用这个工具集来发出请求，只不过这样会与应用代码紧密耦合。借助 URL 加载系统 API，可以控制负载与请求头，从而完全掌控应用与服务器之间的通信。

第4章

生成与解析负载

本章内容

- 使用 Web Service 协议与风格，理解它们对移动应用产生的影响
- 理解常见的负载格式
- 解析 XML、HTML 与 JSON 负载
- 生成 JSON 与 XML 负载

本章代码下载

可以 www.wrox.com 的 Download Code 选项卡中下载本章代码，网址是 www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html。代码位于第 4 章的压缩包中，是个独立的 Xcode 项目，主要内容如下：

- 一个新闻阅读应用，可以解析 XML、HTML 与 JSON 数据并生成 JSON 与 XML 负载。该应用的源代码位于 App.zip 归档文件中。
- 示例应用所用的一个 CNN 文章归档，该归档保存了截至应用开发出来的那一天的文章。文章内容位于 App.zip 归档的“News Content”目录中。
- 一个服务端的 PHP 脚本，用于处理应用生成的 XML 与 JSON 负载。服务端组件的源代码位于 Server.zip 归档中。

随着越来越多的公司开始在工作移动化方面发力，将企业服务集成到应用中变得越来越常见。要想成功实现与 Web Service 的通信，无论服务是否是组织内部的，都需要有生成并解析传输的结构化数据的能力。这种结构化数据构成了应用与 Web Service 之间的契约，只要结构不变，那么应用与 Web Service 就可以实现各自的更新而不会对彼此产生影响。

本章将会比较两种流行的 Web Service 实现，并给出在移动中使用它们时的一些注意

事项。此外，本章还会介绍常见的数据交换格式，如何通过原生的 iOS API 组合来创建并解析负载。所谓负载指的是数据体，表示将要处理的传输意图。

对于那些可以控制服务设计的场合，第 2 章“设计服务架构”中已经介绍了几种 Web Service 设计模式。这些模式旨在优化移动设备的服务并最大化服务层业务逻辑的重用性。这样就会得到更加流线化的负载解析过程(本章将会对此进行介绍)，可以更快地实现应用的多渠道部署，比如说部署到 Web。

4.1　Web Service 协议与风格

协议指的是在与其他系统交换结构化信息时所要遵循的一套格式、过程与规则。此外，协议定义了在传输过程中所要使用的数据格式。这样，接收系统就能正确地解释结构化信息并做出相应的回应。本节将会对 SOAP(企业应用中所用的一种流行的协议)与 REST(一种架构设计风格，非常适合于面向移动设备的 Web Service)进行比较。

4.1.1　简单对象访问协议

简单对象访问协议(Simple Object Access Protocol，SOAP)是个轻量级协议，用于通过可扩展标记语言(Extensible Markup Language，XML)实现系统间的结构化数据交换。SOAP 最初是由微软在 1998 年提出的，旨在使用常用编程语言以一种更方便的方式实现定义、方法与过程，进而简化数据交换。SOAP 1.2 版在 2003 年 6 月成为 W3C 推荐标准，并在 2007 年 4 月对“Part 2”部分进行了增强。

SOAP 消息构成了 Web Service 栈的基础，很多企业建立的服务层都以 SOAP 作为面向服务架构(Service-Oriented Architecture，SOA)的基石，可以服务于防火墙内外的客户端。虽然 SOAP 结点间的消息传输通常都是单向的，不过协议本身是支持双向通信的。SOAP 结点是逻辑上的处理单元，可以发出、接收、处理或是中继 SOAP 消息。

SOAP 消息中包含信封，信封中包含头和体，如图 4-1 所示。头是可选的，如图 4-1 中的虚线对象所示，其中包含服务级的信息，通常是认证与会话管理数据。体是消息的主要内容，包含消息最终接收者所要处理的信息。SOAP 可以使用各种传输层，比如说 HTTP、E-Mail 及异步队列等。这样，SOAP 就成为很多交互问题的解决方案，不过它是在移动设备

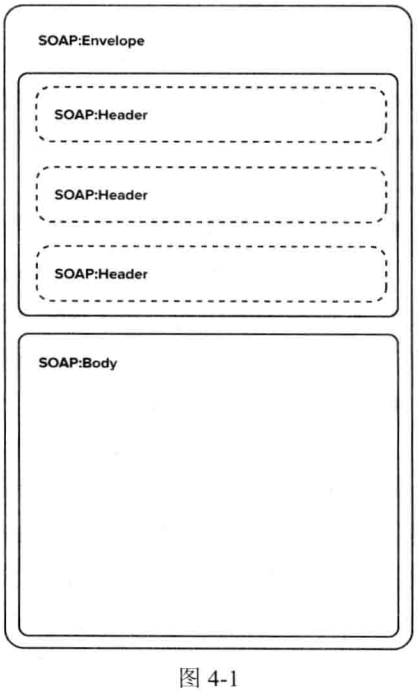


图 4-1

爆发之前设计的。

消息体的内容与结构取决于接收系统。一个体可以包含多个子元素，每个元素都可以有命名空间。每个子元素包含接收者需要执行的操作以及必要的参数值。下述代码片段展示了一个示例 SOAP 体：

```
<soap:Body>
  <m:GetWeather xmlns:m="http://www.<domain>.com/weather">
    <m:ZipCode>95014</m:ZipCode>
  </m:GetWeather>
</soap:Body>
```

上述代码包含了一个 SOAP 体，告诉接收者执行 GetWeather 操作。该操作是有命名空间的，由 xmlns:m 属性指定。这个操作需要 ZipCode 参数以得到响应。

可以为具体的操作参数指定数据类型，方式是在参数的开始标签中加入额外的属性，如下所示：

```
<m:ItemNumber xsi:type='xsd:int'>95014</m:ItemNumber>
```

上述代码指定与元素内容 95014 关联的数据类型是整型。表 4-1 列出了 SOAP 协议支持的常见数据类型以及对应的属性名。

表 4-1　常见的 SOAP 数据类型

数 据 类 型	属　性　名
Integer	xsd:int
Boolean	xsd:boolean
String	xsd:string
Float	xsd:float
Double	xsd:double
Array 或 Dictionary	xsd:struct

根据 Web Service 描述语言(Web Service Description Language，WSDL)与 XML 模式定义(XML Schema Definition，XSD)来生成 SOAP 客户端代码是很复杂的。如果要为多个平台和设备进行开发，那么这种复杂性还会不断加剧。SOAP 的另一个陷阱就是在移动环境下的变更将会非常困难。如果是两台内部的服务器进行通信，那么控制这些服务器进行的变化就是很简单的。不过要是完全依赖于用户来更新应用，结果就完全是另外一回事了。这些困难导致另外一种 Web Service 设计风格的出现，这种风格更加适合于面向移动设备的服务。

4.1.2　表述性状态转移

表述性状态转移(Representational State Transfer，REST)是由 Roy Fielding 于 2000 年作为其博士论文的一部分而提出的，他是 RFC 超文本传输协议的主要作者。虽然人们经常将

REST 看作标准或协议，不过它实际上并不是；REST 是一种架构设计风格，可以应用到 Web Service。虽然 REST 是与 HTTP 1.1 同时发展起来的，也经常与该协议发生关联，不过它并不仅仅限于 HTTP 这一种应用层协议。REST 设计风格最大规模的实现就是万维网，实现了 REST 风格接口的服务通常叫做 RESTful Service。

REST 的一项中心议题就是资源的概念，资源具有全局标识符。统一资源标识符(Uniform Resource Identifier，URI)这一概念将 REST 与其他架构风格区分开来。可以将资源看作独立于表示的任何东西。比如，/user/accounts 可能是获取账户资源的 JSON 列表的一个端点。此外，/user/account/123 可能是获取号码为 123 的这一账户资源特定镜像表示的一个端点。

REST 强调模式设计，从资源的角度来实现，而不像 SOAP 那样根据动作或服务来实现。可以将资源标识符看作完整的句子；有主语(比如/user/account/123)和谓语(比如，请求中使用的 HTTP 方法——POST、GET、PUT 或 DELETE)。这样，REST 就可以实现机器与人类可读了。

RESTful 架构还有两个关键属性：无状态与可缓存。无状态交互要求请求包含所有必要的信息(通常情况下这些信息可能位于会话中)来理解传输上下文。每个请求传输的信息开销抵消了 REST 响应负载的一些好处，不过通常情况下要比其他服务模式更加轻量级。此外，客户端可以更加轻松地缓存响应，因为每个资源都有全局唯一的标识符。诸如图片等静态资源可以使用内容分发网络(Content Delivery Network，CDN)进行托管，因为它们可以缓存到广泛的服务器网络，并且对请求作出快速响应。RESTful 服务中的端点可以返回不同的数据类型。比如，有些端点可能会以 JSON 格式的数据返回资源表示，有些则可能会返回一张图片。

4.1.3　选择一种方式

基于 SOAP 的服务现在依然部署在很多企业中，特别是那些使用着更加流行的套装软件解决方案(如 ERP 软件)的企业。SOAP 与 REST 尝试着通过不同的方式来解决相同的问题。虽然没有一种协议是适合于所有场景的完美解决方案，不过对于移动优化的服务层来说，REST 风格的服务架构是最佳设计。

一个错误的假定是 SOAP 要比 REST 更加安全。之所以会出现这种假设是因为整个 SOAP 中包含了具体的安全方法，名字叫做 WS-Security。然而，创建 WS-Security 的主要原因在于 SOAP 规范是独立于传输的，我们无法对传输层的安全做出任何假设。正如任何的安全应用一样，安全必须被设计到架构中，而且要遵循恰当的原则。

在移动世界中，另一个不成立的假定是远程设备是可靠的。在将 SOAP 用于网络中已知应用服务器之间的通信时，这个假定可能是正确的。不过对于移动设备来说(很容易被盗用)，这个假定就不正确了。需要假定远程设备是不可靠的。

与其他数据协议一样，安全需要良好的设计和原则，这会贯穿于开发的生命周期中；REST 也不例外。在设计 REST 的安全时一定要考虑到应用数据。你必须仔细考虑清楚传

输的数据，确保只传递最少量的数据就能让应用的功能正常使用。

在使用 REST 时可能会出现个人身份信息暴露(Personally Identifiable Information，PII)的问题，在使用 HTTP 或 SOAP 时也面临着相同的问题。除非绝对必要或是收益大于风险，否则绝对不要将 PII 发送给移动设备。虽然这稍微偏离了 REST 风格的约束，不过来自于设备的请求应该利用服务器会话中的信息，以验证任何请求负载的语义正确性。

如果实现得比较好，那么 REST 风格的架构将是向移动信道发送资源的最佳方式。除了本身的好处外，REST 服务还提供了如下最佳组合：

- 开发者熟知及生产力
- 性能
- 网络效率
- 解决安全问题
- 健壮
- 接口灵活

除此之外，一项最佳实践就是将所有的外部服务调用合并成为单个以 REST 风格构建的移动门面，并以 JSON 格式传递资源，这将在 4.2 节“负载”中进行介绍。要想了解关于移动门面架构模式的更详尽信息，请参考第 2 章。

4.2　负载

负载指的是在服务的请求响应事务中交换的数据。比如，在 POST 请求中，负载指的就是请求体。负载并不包含其他数据，如请求头或是请求的 HTTP 方法，例如 POST。如果应用要向 Web Service 发送信息或是接收来自于 Web Service 的信息，那么你需要清楚地理解请求与响应的负载格式。

本节将会介绍常见的负载格式，包括 XML、JSON 与 HTML。一旦理解这些典型的数据交换格式，你就可以进行实践了，掌握如何使用 Web Service 响应以及如何将接收到的数据集成到应用中。很多应用还需要向 Web Service 发送结构化的负载数据，因此本节的最后一个主题将会介绍如何创建 XML 与 JSON 输出，并通过示例来说明如何组织这些请求。

4.2.1　负载数据格式简介

进入与发出的负载数据存在很多形式与大小。比如，有些开发者会使用原生的字符串或是以分隔符分开的数据与 Web Service 进行通信。这么做虽然简单，不过技术上却不具备可扩展性，难以处理复杂的数据结构，可能会导致很多问题。本节将会介绍用于发送和接收结构化数据的 3 种标准方式，分别是可扩展标记语言(XML)、JavaScript 对象符号(JSON)以及超文本标记语言(HTML)。

1. XML

XML 是一种标记语言，用于编码和组织数据。XML 规范(标准通用标记语言 SGML 的扩展)开始于 1996 年，由万维网联盟(W3C)制订。第 5 次修订于 2008 年 11 月发布。最初，XML 的关注点在于文档，不过现在已经被广泛用作 Web Service 中传递结构化数据的格式。XML 已经被扩展为很多标记语言和协议，比如用于组织声音参数控制的 VoiceXML、用于交换财务数据的 Open Financial Exchange(OFX)以及用于发布媒体内容的 Really Simple Syndication(RSS)等。上一节曾提到过，SOAP 协议也通过 XML 来交换结构化数据。

XML 文档包含标记与内容。标记由标签、属性与元素构成。有 3 种类型的标签：起始标签(<person>)、结束标签(</person>)以及空元素标签(<noContact />)。空元素标签也叫做自关闭标签。属性指的是起始标签或空元素标签中的键值对，它们提供了关于元素的附加信息。

元素指的是构成 XML 文档的组件。元素是标签、属性与内容的集合。元素包含起始标签与结束标签或是空元素标签。起始标签与结束标签之间的数据就是内容。内容可以包含标记与其他元素，这样就可以在数据结构中构建父子关系了。下述代码片段展示了一个 XML 元素示例：

```
<person>
   <firstName>Nathan</firstName>
   <lastName>Jones</lastName>
   <emailAddress primary='true'>email@domain.com</emailAddress>
   <noContact medium='email' />
</person>
```

上述代码描述了一个名为 person 的元素，该元素包含几个子元素：firstName、lastName、emailAddress 与 noContact。emailAddress 元素包含一个属性，用于表明该元素的内容是这个人的主 E-Mail 地址。noContact 元素(表示这个人不想被联系)也包含了一个属性，用于表明不应该通过哪种方式联系这个人。

2. JSON

JSON是一种用于交换结构化信息的轻量级数据格式。有人认为JSON是在 2001 年提出并开始使用的，不过记载了application/json媒体类型的RFC4627(http://tools.ietf.org/html/rfc4627)直到 2006 年 7 月才发布出来。从那以后，JSON得到了广泛的应用并开始蓬勃发展，这在一定程度上是由移动应用的爆发引起的，因为移动应用需要高效、易于理解且紧凑的方式在蜂窝数据网络上进行数据交换。

JSON 拥有小巧的格式规则定义集合，在创建负载时需要严格遵守。下面是 JSON 支持的数据类型以及与之关联的格式规则：

- 数字：无双引号
- 布尔：取值为 true 或 false，无双引号
- 字符串：双引号括起

- 数组：方括号包围的以逗号分隔的列表
- 对象：花括号包围的键值对集合。Objective-C 中的对象是通过 NSDictionary 表示的
- null：无双引号

格式良好的 JSON 文档的根类型要么是数组，要么是对象。如下代码片段使用 JSON 表示之前用 XML 表示的 person 示例：

```
{
    "person": {
        "firstName": "Nathan",
        "lastName": "Jones",
        "email": {
            "emailAddress": "email@domain.com",
            "primary": true
        },
        "noContact": "email"
    }
}
```

上述示例将 person 表示为对象，该对象拥有键 firstName、lastName、email 与 noContact。在代码中，email 表示为子对象，从而保证 primary 属性的可见性。该 JSON 文档已经被格式化，实现了更好的可读性，不过看起来不太节省空间。然而，在去掉所有空白与换行后，之前的示例就变成了下面这个更加紧凑的版本，可以被机器轻松解析：

```
{"person":{"firstName":"Nathan","lastName":"Jones","email":{"
emailAddress":"email@domain.com","primary":true},"noContact":"email"}}
```

3. HTML

HTML 是一种标记语言，用于组织网页上的数据，这样浏览器就可以解析页面了。HTML 是由 Tim Berners-Lee 在 20 世纪 90 年代初创建的，用于实现 CERN 的同事交换文档所用。HTML 规范的首个提案发布于 1993 年年中，从那之后已经有了 5 个主要的修订版。在本书撰写之际，第 5 个主要修订版(又叫做 HTML5)还处于草案状态，因为还需要通过规范审查与批准流程。

HTML 文档结构类似于 XML 文档，它们都起源于 SGML。然而，HTML 新的草案(HTML5)并不像之前的版本那样基于 SGML。HTML 文档包含 doctype 定义(DTD)、元素、属性、数据类型与字符实体引用。HTML 与 XML 文档结构的主要差别在于 HTML 文档拥有预先定义好的标签与属性名的集合。本节主要关注于数据结构与内容，因此并不会介绍数据类型与字符实体引用。HTML 中定义了大量的类型，如果考虑 ID、语言、颜色、长度单位等属性的类型，那就更多了。

doctype 定义位于 HTML 文档的第一行，它告诉浏览器当前页面使用的是 HTML 规范的哪个版本。虽然预先定义好了，但 HTML 元素仍就包含起始标签和结束标签(<html>与</html>)，或是空元素标签(
)。元素的属性是位于起始标签中的键值对。属性是预先定义好的，比如 id、name 与 class，不过 HTML5 还支持自定义属性。这些属性以 data-作

为前缀，并且不应该包含大写字符。自定义属性旨在存储不适合现有属性存储的特定于应用的数据。下述代码片段展示了一份HTML文档：

```
<!DOCTYPE html>
<html>
    <head>
      <title>Person: Nathan Jones</title>
    </head>
    <body>
        <div id='firstName'>Nathan</div>
        <div id='lastName'>Jones</div>
        <div id='emailAddress' data-primary='true'>
            email@domain.com
        </div>
        <div id='noContact' data-medium='email' />
    </body>
</html>
```

上述示例使用HTML展示了之前3个代码示例中使用的person对象。doctype定义告诉浏览器文档使用HTML5编写。根元素html有两个子元素：head与body。head标签包含关于页面的元数据，如标题、关键字与页面样式等。body标签包含屏幕上显示的内容。字段名使用id属性定义，其值则是位于起始标签与结束标签之间的内容。注意这里使用HTML5的自定义属性来表示主E-Mail与非联系方式。

4.2.2　解析响应负载

Web Service可以通过多种格式返回结构化数据，不过大多数时候使用的是XML与JSON。也可以让应用只接收HTML结构的数据。实现了这些Web Service或是接收HTML文档的应用必须能解释这种结构化数据并将其转换为对于应用上下文有意义的对象。第6章“保护网络传输”将会介绍如何处理加密的负载。

本节将会介绍如何通过原生iOS API解析和转换响应数据。为了强化本节介绍的概念，这里采用的示例应用是个轻量级的RSS阅读器。应用会聚合CNN头条RSS种子(http://rss.cnn.com/rss/cnn_topstories.rss)的文章内容与Twitter信息。

警告：

主流网站通常都会优化和重新组织站点的标记结构。出于这个原因，从wrox.com下载的本章代码包含了本章编写那天的RSS种子与链接文章的副本。

1. XML

在介绍如何解析XML文档之前，需要理解两种解析方式：Simple API for XML(SAX)与Document Object Model(DOM)。SAX解析器是事件驱动的，它会顺序解析XML文档中的元素，一次处理一个元素。DOM解析器则会将整个XML文档以可遍历的结点树的形式读取到内存中。

iOS 自带了两种原生 XML 解析器，分别是 NSXMLParser 与 libxml。NSXMLParser 是个 Objective-C SAX 解析器，在遇到元素、属性、CData 块、注释与文档起始和结束事件时会调用各种委托方法。libxml 是个开源、基于 C 语言的 API，支持 SAX 与 DOM 解析。libxml SAX 解析类似于 NSXMLParser，因为在遇到某些事件时会进行大量回调。libxml DOM 会将整个 XML 文档读取为结点树，可以通过 XML Path Language(XPath)遍历与查询。这里的示例使用了 NSXMLParser，不过在稍后解析 HTML 时会使用 libxml。

还有很多第三方的 XML 库可供使用，下面列出其中比较知名的 XML 库：

- TBXML(https://github.com/71squared/TBXML)
- TouchXML(https://github.com/TouchCode/TouchXML)
- KissXML(https://github.com/robbiehanson/KissXML)
- GDataXML(http://code.google.com/p/gdata-objectivec-client/source/browse/trunk/Source/XMLSupport/)

与任何原生的 XML 解析器一样，这里的每个库都有优点与缺点。比如，有些库速度快，内存占用少，但却缺乏创建 XML 的能力；有些库速度很快，但却占用更多的内存。在决定选择解析器时，预期的 XML 文档大小是个重要的考虑因素；有些解析器对于小文档来说很适合，有些解析器则支持大型的 XML 文档。使用原生解析库的另一个考虑因素就是它们是由 Apple 发布并提供支持的。这意味着它们会针对 iOS 操作系统未来的每次发布都进行完全的测试以确保向后兼容性。在评估应用中到底该使用哪个解析器时，这些都是要考虑的因素。

这个新闻阅读器的目标是展示文章列表，还可以查看文章全文。就像很多媒体一样，CNN 不会在 RSS 种子中发布文章内容。这意味着获取整个文章数据需要两个步骤。首先，需要获取包含随后元数据的 RSS 种子，这包括指向全文的链接，这样才能获取到每篇文章。其次，需要从文章的 HTML meta 标签中获取到实际的文章内容与额外的元数据。

在创建 XML 解析器之前，你得知道要获取的数据以及从哪里获取。代码清单 4-1 定义了 Post 对象的接口。

说明：

为了更清楚地加以说明，无论是从 RSS 种子还是从 HTML 文章内容中获得的属性都列在了属性的旁边。

代码清单 4-1　Post 对象接口定义(/Application/topstories/topstories/Post.h)

```
@interface Post : NSObject

@property(nonatomic,strong) NSString *title;              //rss
@property(nonatomic,strong) NSString *postDescription;    //rss
@property(nonatomic,strong) NSString *content;            //html
@property(nonatomic,strong) NSString *author;             //html
@property(nonatomic,strong) NSString *section;            //html
@property(nonatomic,strong) NSString *contentURL;         //rss
```

```
@property(nonatomic,strong) NSDate *pubDate;                    //rss
@property(nonatomic,strong) NSMutableArray *keywords;           //html
@property(nonatomic,strong) NSMutableArray *tweets;
@property(nonatomic,assign) BOOL contentFetched;
@property(nonatomic,assign) BOOL tweetsLoading;

- (NSDictionary*)dictionaryRepresentation;


@end
```

定义好Post对象后，就可以开始创建RSS解析器了。代码清单4-2展示了解析器的接口定义。有两点需要注意，一是自定义委托，它会返回获取到的Post对象数组；二是解析器遵循了NSXMLParserDelegate。

代码清单4-2 头条新闻RSS解析器接口定义(/Application/topstories/topstories/TopStoriesParser.h)

```
#import "Post.h"

@protocol TopStoriesDelegate <NSObject>
@required
-(void)topStoriesParsedWithResult:(NSMutableArray*)posts;
@end

@interface TopStoriesParser : NSObject <NSXMLParserDelegate>

@property(nonatomic,strong) NSData              *feedData;
@property(nonatomic,strong) NSMutableArray      *posts;

@property(assign) id<TopStoriesDelegate> delegate;

- (id)initWithFeedData:(NSData*)data;
- (void)parseTopStoriesFeed;

@end
```

代码清单4-3展示了RSS解析器的实现。NSXMLParser是个SAX解析器，因此当某些解析器事件发生时，它会接收到大量的委托消息。

代码清单4-3 头条信息解析器实现(/Application/topstories/topstories/TopStoriesParser.m)

```
#import "TopStoriesParser.h"
#import "Utils.h"

@interface TopStoriesParser () {
    Post            *post;
    NSMutableString *currentValue;
    BOOL            parsingItem;
}
```

```
@end

@implementation TopStoriesParser

@synthesize posts = _posts;
@synthesize feedData = _feedData;
@synthesize delegate = _delegate;

- (id)initWithFeedData:(NSData*)data {
    self = [super init];
    if(self != nil) {
        self.feedData = data;
    }
    return self;
}

- (void)parseTopStoriesFeed {
    // create and start parser
    NSXMLParser *parser = [[NSXMLParser alloc]
                           initWithData:_feedData];
    parser.delegate = self;
    [parser parse];
}

#pragma mark - NSXMLParserDelegate
- (void)parserDidStartDocument:(NSXMLParser *)parser {
    _posts = [[NSMutableArray alloc] init];
}

- (void)parserDidEndDocument:(NSXMLParser *)parser {
    if([_delegate respondsToSelector:
        @selector(topStoriesParsedWithResult:)]) {

        [_delegate topStoriesParsedWithResult:_posts];
    }
}

- (void)parser:(NSXMLParser *)parser
didStartElement:(NSString *)elementName
  namespaceURI:(NSString *)namespaceURI
 qualifiedName:(NSString *)qName
    attributes:(NSDictionary *)attributeDict {

    // if you were expecting an attribute, it would be
    // handled here in the attributeDict by using
    // objectForKey: using the attribute name

    // started a new post, create a fresh object
    if([elementName isEqualToString:@"item"]) {
        post = [[Post alloc] init];
```

```
        parsingItem = YES;
    }
}

- (void)parser:(NSXMLParser *)parser
foundCharacters:(NSString *)string {
    // capture the current element value
    NSString *tmpValue =
        [string stringByTrimmingCharactersInSet:
        [NSCharacterSet whitespaceAndNewlineCharacterSet]];
    if(currentValue == nil) {
        currentValue = [[NSMutableString alloc] initWithString:tmpValue];
    } else {
        [currentValue appendString:tmpValue];
    }
}

- (void)parser:(NSXMLParser *)parser
 didEndElement:(NSString *)elementName
  namespaceURI:(NSString *)namespaceURI
 qualifiedName:(NSString *)qName {

    // reached the end of a post
    if([elementName isEqualToString:@"item"]) {
        [_posts addObject:post];
        post = nil;
        parsingItem = NO;
    }

    // make sure we're parsing a post item and not header data
    if(parsingItem == YES) {
        if([elementName isEqualToString:@"title"]) {
            post.title = currentValue;

        } else if([elementName isEqualToString:@"description"]) {
            post.postDescription = currentValue;

        } else if([elementName isEqualToString:@"pubDate"]) {
            post.pubDate = [Utils publicationDateFromString:currentValue];

        } else if([elementName isEqualToString:@"feedburner:origLink"]) {
            post.contentURL = currentValue;
        }
    }

    // reset the current element value
    currentValue = nil;
}
```

当解析器开始解析RSS种子时，它会在初始化NSMutableArray(用于存储处理过的Post

对象)时调用 parserDidStartDocument:方法。与之类似，当解析器到达文档末尾时，parserDidEndDocument:方法会得到调用。这时，解析器会有完整的 Post 对象列表，从而通知委托。

当开始处理新元素时会调用 parser:didStartElement:namespaceURI:qualifiedName:attributes:方法。这是处理属性的地方；记住，属性是起始标签的一部分。当从元素中读取内容时会调用 parser:foundCharacters:方法，当元素关闭时会调用 parser:didEndElement:namespaceURI:qualifiedName:方法。当所有元素都关闭时就可以安全地处理和存储任何累积下来的内容了。

既然解析器已经完成，我们就可以在应用接收到 RSS 种子数据时调用它了。代码清单 4-4 详细说明了如何初始化解析器、开始解析过程以及处理委托方法。FetchTopStoriesOperation 被注册为 RSS 解析器的委托。当调用 topStoriesParsedWithResult:方法时，操作会遍历每个接收到的 Post 对象，然后发出调用来开始过程的第 2 步，即接收文章内容。

代码清单4-4　获取头条新闻操作的实现(/Application/topstories/topstories/FetchTopStoriesOperation.m)

```
#import "FetchTopStoriesOperation.h"
#import "FetchPostContentOperation.h"
#import "TopStoriesParser.h"

//#define kURL @"file:/<path to folder>/cnn_topstories.rss"
#define kURL @"http://rss.cnn.com/rss/cnn_topstories.rss"
#define kTimeout 30.0

@implementation FetchTopStoriesOperation

- (void)main {

    [self postNotification:kTopStoriesStartNotification];
    [self startNetworkActivityIndicator];

    // create the and issue request
    NSMutableURLRequest *req = [[NSMutableURLRequest alloc]
                                initWithURL:[NSURL URLWithString:kURL]
                                cachePolicy:NSURLCacheStorageAllowed
                                timeoutInterval:kTimeout];

    NSHTTPURLResponse *response = nil;
    NSError *error = nil;
    NSData *data = [NSURLConnection sendSynchronousRequest:req
                                        returningResponse:&response
                                                    error:&error];
    // check response got data and process data accordingly
    if(data != nil) {
        TopStoriesParser *parser = [[TopStoriesParser alloc]
                                    initWithFeedData:data];
        parser.delegate = self;
```

```
        [parser parseTopStoriesFeed];

    // there was an error getting the feed, alert the presses
    } else {
        [self postNotification:kTopStoriesErrorNotification];
    }

    [self stopNetworkActivityIndicator];
}

#pragma mark - TopStoriesDelegate
- (void)topStoriesParsedWithResult:(NSMutableArray *)posts {
    // add the parsed results to the model
    [Model sharedModel].posts = posts;

    // trigger the content fetch (low priority)
    // handled here vs Model.m to adjust priority
    for(Post *post in posts){
        FetchPostContentOperation *op = [[FetchPostContentOperation alloc]
                                         init];
        op.post = post;
        op.queuePriority = NSOperationQueuePriorityLow;
        [op enqueueOperation];
    }

    [self postNotification:kTopStoriesSuccessNotification];
}

@end
```

借助现在收集到的信息，应用的显示界面应该如图4-2所示。可以从wrox.com下载整个源代码以查看完整实现的表视图。

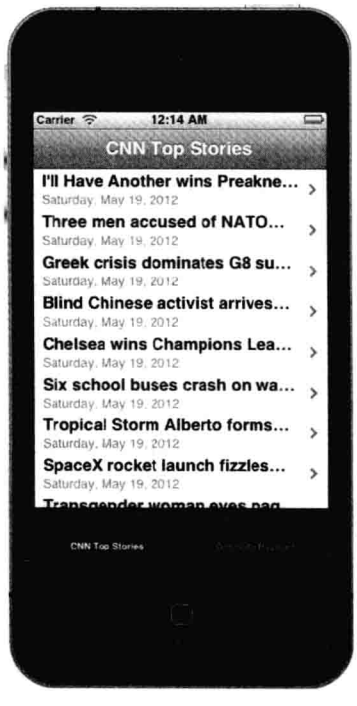


图4-2

目前，应用可以获取并解析RSS种子，然后开始文章内容下载流程，现在是时候解析文章内容了。

2. HTML

在之前的介绍中曾提到过，HTML文档的结构类似于XML。不过，XML文档的结构要求发送端与接收端遵循某种服务契约。HTML文档本身通常不带契约，它们可以频繁地发生巨大的变化而不必通知用户。要想减少频繁变化的HTML内容对应用的影响，一种解决方案就是实现远程门面，参见第2章的介绍。这个Web Service要求与应用之间保持严格的内容结构契约。应用的变化需要App Store的审批，用户也需

要更新其应用，不过处于控制下的 Web Service 的变更则是很容易部署的，而且在适应对之前内容的变化上更加灵活。

警告：

考虑到 HTML 文档很容易变化这个问题，你不应该在应用中解析 HTML。变更会极大地影响应用的正常使用。

本节将会使用 Ben Reeves 创建的 libxml 封装器(https://github.com/zootreeves/Objective-C-HMTL-Parser)来解析从 RSS 种子获取到的每一篇文章的内容。要想在应用中使用该封装器，需要遵循如下步骤：

(1) 将 HTMLNode.h /.m 与 HTMLParser.h /.m 文件添加到项目中。

(2) 将 libxml2.dylib 添加到项目中。

(3) 将$(SDKROOT)/usr/include/libxml2 添加到目标构建设置的 Header Search Paths 域中。

(4) 禁用 HTMLNode 与 HTMLParser 文件的 ARC。在目标构建阶段，将编译器标记 -fno-objc-arc 添加到 HTMLNode.m 与 HTMLParser.m 中。在本书撰写之际，该封装器还没有对 ARC 提供支持。

代码清单 4-5 详细展示了应用生成与处理文章内容请求的过程。当网络调用完毕时，自定义方法 processContentData:会得到调用。该方法创建了一个新的 HTMLParser，它会处理 head 与 body 标签，获取相关的元数据与文章内容。获取文章内容的逻辑非常有趣，因为 CNN 使用特殊的类来表示某个段落标记是文章的一部分。

代码清单 4-5　获取文章内容的实现(/Application/topstories/topstories/FetchPostContentOperation.m)

```
#import "FetchPostContentOperation.h"
#import "HTMLParser.h"

#define kTimeout 30.0

@interface FetchPostContentOperation ()
- (void)processContentData:(NSData*)content;
@end

@implementation FetchPostContentOperation

@synthesize post = _post;

- (void)main {

    [self postNotification:kPostContentStartNotification];
    [self startNetworkActivityIndicator];

    NSURL *url = [NSURL URLWithString_post.contentURL];
    NSMutableURLRequest *req = [[NSMutableURLRequest alloc]
```

```
                                    initWithURL:url
                                    cachePolicy:NSURLCacheStorageAllowed
                                    timeoutInterval:kTimeout];
    NSHTTPURLResponse *response = nil;
    NSError *error = nil;
    NSData *data = [NSURLConnection sendSynchronousRequest:req
                                          returningResponse:&response
                                                      error:&error];

    // check response got data and process data accordingly
    if(data != nil) {
        [self processContentData:data];
        _post.contentFetched = YES;

        [self postNotification:kPostContentSuccessNotification];

    // there was an error getting the post content, alert the presses
    } else {
        [self postNotification:kPostContentErrorNotification];

    }

    [self stopNetworkActivityIndicator];
}

#pragma mark - Private Methods
- (void)processContentData:(NSData*)content {

    NSError *error = nil;
    HTMLParser *parser = [[HTMLParser alloc]
                          initWithData:content error:&error];
    if(error) {
        return;
    }

    // get html doc head and meta tags
    HTMLNode *head = [parser head];
    NSArray *metaTags = [head findChildTags:@"meta"];

    // retrieve article meta data and add to the post
    for(HTMLNode *meta in metaTags) {
        NSString *name = [meta getAttributeNamed:@"name"];

        // keywords
        if([name isEqualToString:@"keywords"]) {
            NSString *keywordContent = [meta getAttributeNamed:@"content"];
            NSMutableArray *keywords = (NSMutableArray*)
            [keywordContent
            componentsSeparatedByString:@","];
```

```
            if([keywords count]>0) {
                _post.keywords = keywords;
            }

        // author name
        } else if([name isEqualToString:@"author"]) {
        NSString *author = [meta getAttributeNamed:@"content"];
        if(author.length > 0) {
            _post.author = author;
        }

        // article section
        } else if([name isEqualToString:@"section"]) {
            NSString *section = [meta getAttributeNamed:@"content"];
            if(section.length > 0) {
                _post.section = section;
            }
        }
    }

    // get html doc body and paragraph tags
    HTMLNode *body = [parser body];
    NSArray *paragraphTags = [body findChildTags:@"p"];

    // iterate through all paragraphs saving the appropriate story content
    NSMutableString *storyContent = [[NSMutableString alloc] init];
    for(HTMLNode *para in paragraphTags) {

        // only save the 'story paragraphs' - class=cnn_storypgraphtxt
        NSString *class = [para getAttributeNamed:@"class"];
        NSRange storyParaTest = [[class lowercaseString]
                                 rangeOfString:@"cnn_storypgraphtxt"];
        if((storyParaTest.location != NSNotFound) && (class != nil)) {
            [storyContent appendString:[para rawContents]];
        }
    }

    _post.content = storyContent;
}

@end
```

如你所见，解析 HTML 非常麻烦。在这个过程中有很多可能出现问题的地方，每处都会导致应用不可用。对文章内容的类名的简单改动就会导致应用无法获取到任何文章内容，只能显示空白视图。

不过，如果一切顺利，那么文章内容就会被获取到，可以看到每篇文章并阅读所有内容。虽然是个简单的布局，但图 4-3 还是展示了文章内容视图的样子。图中所示的内容表明已经获取到所有 CNN 相关的文章。

下一特性是添加轻量级的 Twitter 搜索集成来搜索从 meta 标签中获取到的文章关键字。

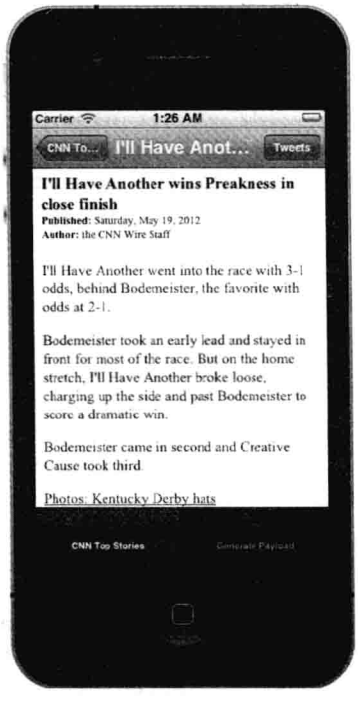


图 4-3

3. JSON

从 iOS 5 开始，Apple 通过 NSJSONSerialization 类提供了原生的 JSON 解析支持。在 iOS 5 的原生支持之前，解析 JSON 数据需要使用第三方库，比如 JSON framework(https://github.com/stig/json-framework)或 JSON-Kit(https://github.com/johnezang/JSONKit)。虽然这些库支持很好，使用起来也很简单，不过 Apple 支持的原生 API 也是很受大家欢迎的。

NSJSONSerialization提供了两个方法来解析 JSON 数据：JSONObjectWithData:options:error: 与 JSONObjectWithStream:options:error。JSONObjectWithData:options:error: 会根据传递进来的 JSON 数据创建 Foundation 对象。JSONObjectWithStream:options:error:的行为类似于JSONObjectWithData:options:error:，只不过接收的源是JSON数据种子。这两个方法都有options参数，可以接收如下值的组合来配置解析器解析输入的方式：

- NSJSONReadingAllowFragments：告诉解析器处理既不是 NSArray 也不是 NSDictionary 的顶层对象。这个选项可以处理诸如{"user": null}这样的简单 JSON 结构的转换。
- NSJSONReadingMutableContainers：告诉解析器生成 NSMutableArray 与 NSMutableDictionary 对象。可变对象意味着可以通过方法修改它们，比如 NSMutableArray 的 addObject:与 NSMutableDictionary 的 setObject:forKey:。这可以用于如下场景：有主要和次要的结果集，你要在进一步处理前将次要结果中的值添加到主要结果中。
- NSJSONReadingMutableLeaves：告诉解析器生成 NSMutableString 对象。如果要在进一步处理前操纵被解析的响应中的某个特定值，那么可以使用该选项。

示例应用的一项特性就是可以基于文章关键字获取相关的推文。这是通过 Twitter 的搜索 API(http://search.twitter.com/search.json?q=<query>)实现的，它会返回 JSON 编码的搜索结果。代码清单 4-6 展示了 Tweet 对象的接口定义。Tweet 对象使用从 Twitter 返回的字段的一个小的子集。在获取时，每个 Post 对象维护一个相关推文的数组。

代码清单 4-6　Tweet 对象接口定义(/Application/topstories/topstories/Tweet.h)

```
@interface Tweet : NSObject
```

```
@property(nonatomic,strong) NSString *identifier;
@property(nonatomic,strong) NSString *fromUser;
@property(nonatomic,strong) NSString *fromUserDisplay;
@property(nonatomic,strong) NSString *profileImageURL;
@property(nonatomic,strong) NSString *text;
@property(nonatomic,strong) NSDate *createdDate;
@property(nonatomic,strong) UIImage *profileImage;

- (id)initWithDictionary:(NSDictionary*)tweetData;

@end
```

说明：

就在开始这个示例应用的开发后不久，CNN 停止了在文章中装配关键字 meta 标签。考虑到这个变化，作者在 wrox.com 上的文章中创建了关键字 meta 数据。虽然这个变化并不会对应用造成影响，不过这还是说明了在应用中解析 HTML 内容是有潜在风险的。

定义好 Tweet 对象后，应用就可以搜索 Twitter 并开始解析 JSON 响应了。代码清单 4-7 展示了如何创建搜索请求并解析生成的响应。当接收到响应时，操作会调用 JSONObjectWithData:options:error:以创建代表搜索结果的 NSDictionary。接下来，操作会遍历推文数据，为每个结果创建 Tweet 对象，然后将其添加到 Post 对象中。

代码清单 4-7　获取相关推文(/Application/topstories/topstories/FetchPostTweetsOperation.m)

```
#define kTimeout 30.0

@implementation FetchPostTweetsOperation

@synthesize post = _post;

- (void)main {

    [self postNotification:kTweetsStartNotification];
    [self startNetworkActivityIndicator];
    _post.tweetsLoading = YES;

    // create the twitter query
    NSMutableString *query = [[NSMutableString alloc] init];
    for(int i=0; i<[_post.keywords count]; i++) {
        // prepend comma for all but first keyword
        if(i != 0) {
            [query appendString:@","];
        }

        [query appendString:[_post.keywords objectAtIndex:i]];
    }
```

```
// create the and issue request - separate variables for line size
NSString *searchEndpoint = @"http://search.twitter.com/search.json";
NSString *querystring = [NSString
                         stringWithFormat:@"q=%@&rpp=15",
                         [Utils urlEncode:query]];

NSString *url = [NSString stringWithFormat:@"%@?%@",
                 searchEndpoint,
                 querystring];
NSMutableURLRequest *req = [[NSMutableURLRequest alloc]
                            initWithURL:[NSURL URLWithString:url]
                            cachePolicy:NSURLCacheStorageAllowed
                            timeoutInterval:kTimeout];

NSHTTPURLResponse *response = nil;
NSError *error = nil;
NSData *data = [NSURLConnection sendSynchronousRequest:req
                                     returningResponse:&response
                                                 error:&error];

// check response got data and process data accordingly
if(data != nil) {

    // convert response to JSON
    NSError *error = nil;
    NSDictionary *searchResults = [NSJSONSerialization
                                   JSONObjectWithData:data
                                   options:NSJSONReadingAllowFragments
                                   error:&error];

    // create tweets
    NSMutableArray *tweets = [[NSMutableArray alloc] init];
    NSArray *results = [searchResults objectForKey:@"results"];
    for(NSDictionary *tweetData in results) {
        Tweet *tweet = [[Tweet alloc] initWithDictionary:tweetData];
        [tweets addObject:tweet];
    }

    _post.tweetsLoading = NO;
    if([tweets count] > 0) {
        _post.tweets = tweets;
        [self postNotification:kTweetsSuccessNotification];

    // no tweets were retrieved
    } else {
        [self postNotification:kTweetsErrorNotification];
    }

// there was an error getting the post content, alert the presses
} else {
```

```
        _post.tweetsLoading = NO;
        [self postNotification:kTweetsErrorNotification];

    }

    [self stopNetworkActivityIndicator];
}

@end
```

当连接好 TweetsTableViewController(请从 wrox.com 下载)及操作完成后，你应该会看到类似于图 4-4 所示的推文视图。

图 4-4

4.2.3　生成请求负载

将复杂的 Web Service 集成到应用中通常需要以一种结构化的格式来发送负载。这些格式通常是 JSON 或 XML，也可能是 XML 的某个变种，如 SOAP 等。本节将会介绍如何通过应用中使用的 Foundation 对象来创建这些常见的交换格式。本节将会构建新闻阅读器，将聚合的文章内容发送给一个简单的服务器端脚本。该脚本位于从 wrox.com 下载的本章代码中。

1. JSON

生成JSON数据与解析一样简单。Apple随iOS 5发布了NSJSONSerialization，iOS 5提供了一个原生API，用于从Foundation对象创建JSON数据。NSJSONSerialization公开了两个JSON数据创建方法，分别是dataWithJSONObject:options:error:与writeJSONObject:toStream:options:error:。每个方法都包含options参数，用来配置方法的输出。在本书撰写之际，只有选项NSJSONWritingPrettyPrinted，用于告诉方法通过添加空白来生成更易读的JSON。不指定该选项则会生成尽可能紧凑的JSON。

NSJSONSerialization 还提供了 isValidJSONObject:来验证尝试转换的 Foundation 对象是否可以转换成 JSON。对于能够转换为 JSON 的对象来说，必须满足如下规则：

- 顶层对象是 NSArray 或 NSDictionary。
- 所有的对象必须是 NSString、NSNumber、NSArray、NSDictionary 或 NSNull。
- 所有的 NSDictionary 键必须是 NSStrings。
- NSNumbers 不能为 NaN 或无穷大。

由于新闻阅读器应用以自定义类 Post 的格式存储文章，因此在使用 NSJSONSerialization 时还需要做些额外的处理。之所以要做这种处理是因为 Post 类不满足方才提及的转换规则。应用能够将文章转换为 JSON 的一种方式是在 Post 中实现一个返回 NSDictionary 的方法，

如代码清单 4-8 所示。我们只添加类属性的一个子集，就可以降低示例的负载大小。

代码清单 4-8　生成 Post NSDictionary 表示(/Application/topstories/topstories/Post.m)

```
...
- (NSDictionary*)dictionaryRepresentation {
    NSString *pubDateString =
    [NSString stringWithFormat:@"%@", self.pubDate];

    // content, tweets, and keywords were left off to limit size
    return [NSDictionary dictionaryWithObjectsAndKeys:
            [Utils urlEncode:self.title], @"title",
            [Utils urlEncode:self.postDescription], @"description",
            [Utils urlEncode:self.author], @"author",
            [Utils urlEncode:self.section], @"section",
            [Utils urlEncode:self.contentURL], @"contentURL",
            [Utils urlEncode:pubDateString], @"pubDate", nil];
}
```

既然每个 Post 对象都有 NSDictionary 表示，所以它们可以被转换为 JSON 并发送给服务端脚本，如代码清单 4-9 所示。

代码清单 4-9　JSON 生成与发送(/Application/topstories/topstories/ShareArticlesOperationJSON.m)

```
#import "ShareArticlesOperationJSON.h"

@implementation ShareArticlesOperationJSON

@synthesize posts, shareType;

- (void)main {

    [self startNetworkActivityIndicator];

    // create url and issue request
    NSString *urlString =
    [NSString
     stringWithFormat:@"<server>/parse.php?parseMethod=%d",shareType];
    NSURL *url = [NSURL URLWithString:urlString];

    NSMutableURLRequest *req =
    [[NSMutableURLRequest alloc] initWithURL:url
                                 cachePolicy:NSURLCacheStorageAllowed
                             timeoutInterval:30.0];

    [req setHTTPMethod:@"POST"];
    [req setValue:@"application/json" forHTTPHeaderField:@"Accept"];

    // convert array of POST objects to array of dictionary
    // objects so NSJSONSerialization can handle it
```

```
    NSMutableArray *articles = [[NSMutableArray alloc] init];
    for(Post *post in posts) {
    // dictionaryRepresentation is a custom method
    // that creates an NSDictionary of a few key fields
    [articles addObject:post.dictionaryRepresentation];
}
NSDictionary *articleData =
[NSDictionary dictionaryWithObject:articles forKey:@"articles"];

// validate object
if([NSJSONSerialization isValidJSONObject:articleData] == NO) {
    [self postNotification:kShareArticleErrorNotification];
    [self stopNetworkActivityIndicator];
    return;
}

// convert dictionary to JSON data and set the body
NSError *jsonWriteError = nil;
NSData *payload =
[NSJSONSerialization dataWithJSONObject:articleData
                                options:NSJSONWritingPrettyPrinted
                                  error:&jsonWriteError];

    [req setHTTPBody:payload];


    NSHTTPURLResponse *response = nil;
    NSError *error = nil;
    NSData *data = [NSURLConnection sendSynchronousRequest:req
                                         returningResponse:&response
                                                     error:&error];

    // check response got data and process data accordingly
    // you would also typically check the status code here too
    if(data != nil) {
        NSError *jsonParseError = nil;
        NSDictionary *responseDict =
        [NSJSONSerialization JSONObjectWithData:data
                                        options:0
                                          error:&jsonParseError];

    // successfully transmitted articles
    if([responseDict objectForKey:@"articleCount"] != nil) {
        [self postNotification:kShareArticleStartNotification];

        // tell the user how many articles were sent
        NSInteger articleCount =

            [[responseDict objectForKey:@"articleCount"] intValue];
```

```
            NSString *msg =
                [NSString stringWithFormat:@"%d articles shared via JSON.",
                articleCount];

            dispatch_async(dispatch_get_main_queue(), ^{
                [[[UIAlertView alloc] initWithTitle:@"Great Success"
                                            message:msg
                                            delegate:nil
                                  cancelButtonTitle:@"OK"
                                  otherButtonTitles:nil] show];
            });

            // server was not able to properly interpret the content
            } else {
                [self postNotification:kShareArticleErrorNotification];
            }

        }

        [self stopNetworkActivityIndicator];
    }
```

图 4-5

应用创建了一个NSDictionary对象(表示每一个Post对象)的数组，然后使用 articles 的一个键值对来创建NSDictionary。这样生成的 JSON 顶层就是个集合了。

尝试将不合法的类型转换为 JSON 会导致应用崩溃。因此，在调用 dataWithJSONObject: options:error:之前最好先调用 isValidJSONObject:以优雅地处理潜在的错误。

在执行完 ShareArticlesOperationJSON 后，用户应该会看到如图 4-5 所示的警告。

2. XML

有多种方式可以创建 XML 文档，包括字符串格式化、使用之前提到的支持写入 XML 的第三方库，或是使用 libxml 等。libxml 是 iOS 自带的用于生成 XML 的原生唯一 API，它是基于 C 语言的 API，这意味着对于之前没有使用过 C 函数的用户来说，使用起来会觉得有些繁琐。

这里的示例使用 libxml，特别是 xmlwriter 接口。最终，选择的方式取决于具体的需求。如果所有的服务通信都是通过 JSON 完成的，只有一个第三方调用使用包含两个字段的 XML 文档，那么使用 libxml 就有点小题大做了。不过，更复杂的需求则需要使用第三方库或是对 libxml 更灵活的封装器，而不是使用此处介绍的方式。

根据本章之前介绍 XML 解析时使用的步骤进行构建，一个额外的步骤是使用 libxml

创建 XML 文档；必须将<libxml/xmlwriter.h>导入每个创建 XML 的类中。在新闻阅读器示例应用中，XML 的生成是通过 Utils 中的 postXMLDataFromDictionary:方法实现的，如代码清单 4-10 所示。

代码清单 4-10　文章 XML 的生成(/Application/topstories/topstories/Utils.m)

```
#import "Utils.h"
#import <libxml/xmlwriter.h>

@implementation Utils
...
+ (NSData*)postXMLDataFromDictionary:(NSDictionary*)dictionary {

    xmlTextWriterPtr _writer;
    xmlBufferPtr _buffer;
    xmlChar *_elementName;
    xmlChar *_elementValue;

    _buffer = xmlBufferCreate();
    _writer = xmlNewTextWriterMemory(_buffer, 0);

    xmlTextWriterStartDocument(_writer, "1.0", "UTF-8", NULL);
    xmlTextWriterStartElement(_writer, BAD_CAST "articles");

    NSArray *posts = [dictionary objectForKey:@"articles"];
    for(NSDictionary *post in posts) {

        // start the post element
        xmlTextWriterStartElement(_writer, BAD_CAST "article");

        // create elements for each post property
        NSArray *keys = [post allKeys];
        for(NSString *key in keys) {
            // you could optionally check class types here
            // and do additional processing, however, the
            // types being processed can be cast as xmlChar*

            // xmlChar pointer to element name and value
            _elementName = (xmlChar*)[key UTF8String];
            _elementValue = (xmlChar*)[[post objectForKey:key]
                                       UTF8String];

            // write the element
            xmlTextWriterStartElement(_writer, _elementName);
            xmlTextWriterWriteString(_writer, _elementValue);
            xmlTextWriterEndElement(_writer); // </_elementName>
        }

        xmlTextWriterEndElement(_writer); // </article>
```

```
    }

    xmlTextWriterEndElement(_writer); // </articles>
    xmlTextWriterEndDocument(_writer);
    xmlFreeTextWriter(_writer);

    // convert buffer to NSData and cleanup
    NSData *_xmlData = [NSData dataWithBytes:(_buffer->content)
    length:(_buffer->use)];

    xmlBufferFree(_buffer);

    return _xmlData;
}

@end
```

调用 xmlTextWriterStartDocument()函数会将 XML 版本号与编码定义添加到文档中。接下来根据需要开始调用 xmlTextWriterStartElement()、xmlTextWriterWriteString()与xmlTextWriterEndElement()函数来创建 XML 结构。xmlTextWriterEndElement()函数不需要关闭元素名，它会帮你完成。

该例没有用到的两个重要的 libxml 功能是向 XML 文档中添加注释与元素属性。与元素类似，可以通过调用 xmlTextWriterStartComment()、xmlTextWriterWriteComment()与xmlTextWriterEndComment()函数来添加注释。属性的处理则不同，有一个便捷的函数xmlTextWriterWriteAttribute()，它接收属性名与属性值，并帮你处理整个过程的开始与结束部分。

当方法遍历完所有文章后，它会最后调用 xmlTextWriterEndElement()函数一次来关闭父元素(本例中就是 articles)，然后调用 xmlTextWriterEndDocument()函数以完成处理过程。处理完毕后，应用会将 XML 缓存转换为 NSData 以便在请求 post 体中发送出去。

现在应用已经能够生成必要的 XML 了，代码清单 4-11 展示了如何将数据发送给服务器。

代码清单 4-11 XML 请求创建与发送(/Application/topstories/topstories/ShareArticlesOperationXML.m)

```
#import "ShareArticlesOperationXML.h"

@implementation ShareArticlesOperationXML

@synthesize posts, shareType;

- (void)main {

    [self startNetworkActivityIndicator];

    // create the and issue request
    ...
```

```
    // convert array of POST objects to array of dictionary
    // objects so the XML writer can handle it
    NSMutableArray *articles = [[NSMutableArray alloc] init];
    for(Post *post in posts) {
        [articles addObject:post.dictionaryRepresentation];
    }
    NSDictionary *articleData =
    [NSDictionary dictionaryWithObject:articles forKey:@"articles"];

    // convert dictionary to XML data and set body
    NSData *payload = [Utils postXMLDataFromDictionary:articleData];
    [req setHTTPBody:payload];

    // issue network request
    NSHTTPURLResponse *response = nil;
    NSError *error = nil;
    NSData *data = [NSURLConnection sendSynchronousRequest:req
                                         returningResponse:&response
                                                     error:&error];
    // check response got data and process data accordingly
    // you would also typically check the status code here too
    if(data != nil) {
        ...
    }

    [self stopNetworkActivityIndicator];
}

@end
```

代码清单 4-11 中的重要部分看起来应该很熟悉了，它类似于代码清单 4-9。出于简洁的目的，这里省略掉了一些代码，不过遵循着类似的处理。为了简化 XML 生成逻辑，Post 对象数组转换为 NSDictionary 的过程类似于代码清单 4-9 中的步骤。在生成 XML 文档后，它会被发送给服务器。

在执行完 ShareArticlesOperationXML 后，用户会看到如图 4-6 所示的警告。

图 4-6

4.3　小结

在设计应用该如何与 Web Service 通信时有几个因素需要考虑。如果实现恰当，那么将数据发送给移动设备的最佳架构就是 REST。对于 iOS 应用来说，最佳的数据交换格式是 JSON。虽然 XML 也是原生支持的，不过 JSON 负载更易于使用，可以更好地映射到 Foundation 类型，对于蜂窝网络也更加友好。

下一章将介绍在网络通信过程中哪里会出现错误以及如何优雅地处理这些错误。

第 5 章

错 误 处 理

本章内容

- iOS 应用中的网络错误源
- 检测网络的可达性
- 错误处理的经验法则
- 处理网络错误的设计模式

本章代码下载

可以在 www.wrox.com 的 Download Code 选项卡中下载本章代码，网址是 www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html。代码位于第 5 章的压缩包中，并且根据每节的名字单独命名。

到目前为止，我们所介绍的 iPhone 与其他系统的网络交互都是基于一切正常这个假设。本章将会放弃这个假设，并深入探究网络的真实世界。在真实世界中，事情是会出错的，有时可能是非常严重的错误：手机进入与离开网络、包丢掉或是延迟；网络基础设施出错；偶尔用户还会出错。如果一切正常，那么编写 iOS 应用就会简单不少，不过遗憾的是现实并非如此。本章将会探讨导致网络操作失败的几个因素，介绍系统如何将失败情况告知应用，应用又该如何优雅地通知用户。此外，本章还将介绍如何在不往应用逻辑中添加错误处理代码的情况下，以一种整洁且一致的方式处理错误的软件模式。

5.1 理解错误源

早期的 iOS 有个很棒的天气预报应用。它在 Wi-Fi 和信号良好的蜂窝网络下使用正常，不过当网络质量不那么好时，这个天气预报应用就像感冒似的，在主屏幕上崩溃。有不少

应用在出现网络错误时表现很差劲，会疯狂弹出大量 UIAlertView 以告诉用户出现了“404 Error on Server X”等类似信息。还有很多应用在网络变慢时界面会变得没有响应。这些情况的出现都是没有很好地理解网络失败模式以及没有预期到可能的网络降级或是失败。如果想要避免这类错误并能够充分地处理网络错误，那么你首先需要理解它们的起源。

考虑一个字节是如何从设备发往远程服务器以及如何从远程服务器将这个字节接收到设备，这个过程只需要几百毫秒的时间，不过却要求网络设备都能正常工作才行。设备网络与网络互联的复杂性导致了分层网络的产生。分层网络将这种复杂环境划分成了更加易于管理的模块。虽然这对程序员很有帮助，不过当数据在各个层之间流动时可能会产生之前提到的网络错误。图 5-1 展示了 Internet 协议栈的各个层次。

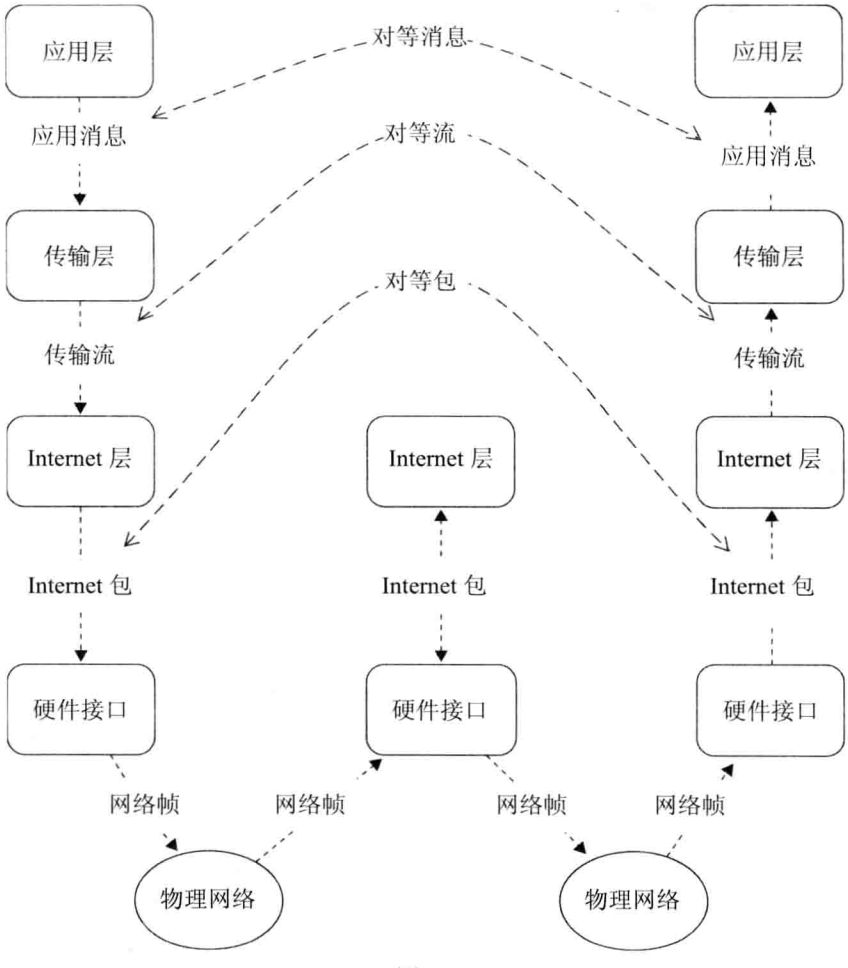


图 5-1

每一层都会执行某种错误检测，这可能是数学意义上的、逻辑意义上的，或是其他类型的检测。比如，当网络接口层接收到某一帧时，它首先会通过错误校正码来验证内容，如果不匹配，那么错误就产生了。如果这个帧根本就没有到达，那就会产生超时或是连接重置。错误检测出现在栈的每一层，自下而上直到应用层，应用层则会从语法和语义上检

查消息。

在使用 iOS 中的 URL 加载系统时，虽然手机与服务器之间的连接可能会出现各种各样的问题，不过可以将这些原因分成 3 种错误类别，分别是操作系统错误、HTTP 错误与应用错误。这些错误类别与创建 HTTP 请求的操作序列相关。图 5-2 展示了向应用服务器发出的 HTTP 请求(提供来自于企业网络的一些数据)的简单序列图。每块阴影区域都表示这 3 种错误类型的错误域。典型地，操作系统错误是由 HTTP 服务器问题导致的。HTTP 错误是由 HTTP 服务器或应用服务器导致的。应用错误是由请求传输的数据或应用服务器查询的其他系统导致的。

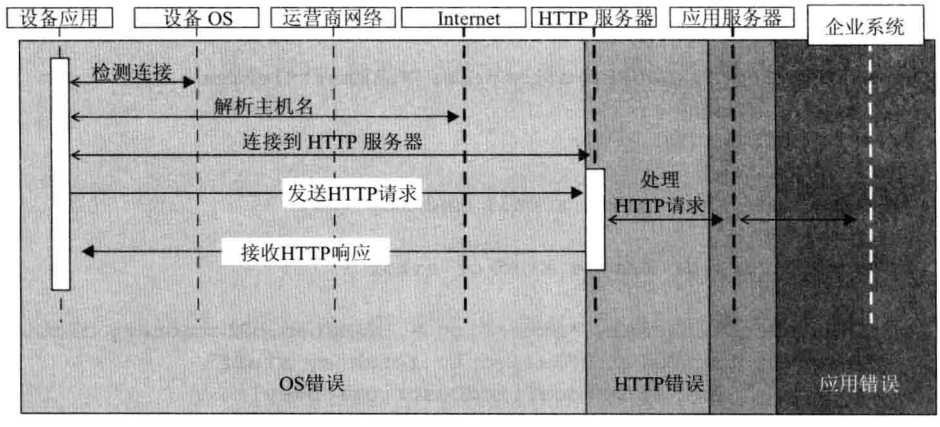


图 5-2

如果请求是安全的 HTTPS 请求，或是 HTTP 服务器被重定向客户端，那么上面这个序列的步骤将会变得更加复杂。上述很多步骤都包含着大量的子步骤，比如在建立 TCP 连接时涉及的 SYN 与 SYN-ACK 包序列等。下面将会详细介绍每一种错误类别。

5.1.1 操作系统错误

操作系统错误是由数据包没有到达预定目标导致的。数据包可能是建立连接的一部分，也可能位于连接建立的中间阶段。OS 错误可能由如下原因造成：

- 没有网络——如果设备没有数据网络连接，那么连接尝试很快就会被拒绝或是失败。这些类型的错误可以通过 Apple 提供的 Reachability 框架检测到，本节后面将会对此进行介绍。
- 无法路由到目标主机——设备可能有网络连接，不过连接的目标可能位于隔离的网络中或是处于离线状态。这些错误有时可以由操作系统迅速检测到，不过也有可能导致连接超时。
- 没有应用监听目标端口——在请求到达目标主机后，数据包会被发送到请求指定的端口号。如果没有服务器监听这个端口或是有太多的连接请求在排队，那么连接请求就会被拒绝。

- 无法解析目标主机名——如果无法解析目标主机名，那么URL加载系统就会返回错误。通常情况下，这些错误是由配置错误或是尝试访问没有外部名字解析且处于隔离网络中的主机造成的。

在iOS的URL加载系统中，操作系统错误会以NSError对象的形式发送给应用。iOS通过NSError在软件组件间传递错误信息。相比简单的错误代码来说，使用NSError的主要优势在于NSError对象包含了错误域属性。

不过，NSError对象的使用并不限于操作系统。应用可以创建自己的NSError对象，使用它们在应用内传递错误消息。如下代码片段展示的应用方法使用NSError向调用的视图控制器传递回失败信息：

```
- (id)fetchMyStuff:(NSURL*)url error:(NSError**)error
{
    BOOL errorOccurred = NO;

    // some code that makes a call and may fail

    if(errorOccurred) //some kind of error
    {
        NSMutableDictionary *errorDict = [NSMutableDictionary dictionary];
        [errorDictsetValue:@"Failed to fetch my stuff"
                    forKey:NSLocalizedDescriptionKey];
        *error = [NSErrorerrorWithDomain:@"myDomain"
                                    code:kSomeErrorCode
                                userInfo:errorDict];
        return nil;
    } else {
        return stuff
    }

}
```

域属性根据产生错误代码的库或框架对这些错误代码进行隔离。借助域，框架开发者无须担心覆盖错误代码，因为域属性定义了产生错误的框架。比如，框架A与B都会产生错误代码1，不过这两个错误代码会被每个框架提供的唯一域值进行区分。因此，如果代码需要区分NSError值，就必须对NSError对象的code与domain属性进行比较。

NSError对象有如下3个主要属性：

- code——标识错误的NSInteger值。对于产生该错误的错误域来说，这个值是唯一的。
- domain——指定错误域的NSString指针，比如NSPOSIXErrorDomain、NSOSStatusErrorDomain及NSMachErrorDomain。
- userInfo——NSDictionary指针，其中包含特定于错误的值。

URL加载系统中产生的很多错误都来自于NSURLErrorDomain域，代码值基本上都来自于CFNetworkErrors.h中定义的错误代码。与iOS提供的其他常量值一样，代码应该使用针对错误定义好的常量名而不是实际的错误代码值。比如，如果客户端无法连接到主机，

那么错误代码是1004，并且有定义好的常量kCFURLErrorCannotConnectToHost。代码绝不应该直接引用1004，因为这个值可能会在操作系统未来的修订版中发生变化；相反，应该使用提供的枚举名kCFURLError。

如下是使用URL加载系统创建HTTP请求的代码示例：

```
NSHTTPURLResponse *response=nil;
NSError *error=nil;
NSData *myData=[NSURLConnectionsendSynchronousRequest:request
                                    returningResponse:&response
                                                error:&error];

if (!error) {
// No OS Errors, keep going in the process
   ...
} else {
   // Something low level broke
}
```

注意，NSError对象被声明为指向nil的指针。如果出现错误，那么NSURLConnection对象只会实例化NSError对象。URL加载系统拥有NSError对象；如果稍后代码会用到它，那么应该保持这个对象。如果在同步请求完成后NSError指针依然指向nil，那就说明没有产生底层的OS错误。这时，代码就知道没有产生OS级别的错误，不过错误可能出现在协议栈的某个高层。

如果应用创建的是异步请求，那么NSError对象就会返回到委托类的下面这个方法：

```
- (void)connection:(NSURLConnection *)connection
   didFailWithError:(NSError *)error
```

这是传递给请求委托的最终消息，委托必须能识别出错误的原因并作出恰当的反应。在如下示例中，委托会向用户展示UIAlertView：

```
- (void) connection:conndidFailWithError:error {
   UIAlertView *alert = [UIAlertViewalloc] initWithTitle:@"Network Error"
                                          message:[error description]
                                          delegate:self
                                cancelButtonTitle:@"Oh Well"
                                otherButtonTitles:nil];
   [alert show];
   [alert release];
}
```

上述代码以一种生硬且不友好的方式将错误展现给了用户。在iOS人机界面指南(HiG)中，Apple建议不要过度使用UIAlertViews，因为这会破坏设备的使用感受。5.3节“优雅地处理网络错误”中介绍了如何通过良好的用户界面以一种干净且一致的方式处理错误的模式。

iOS设备通信错误的另一主要原因就是由于没有网络连接而导致设备无法访问目标服

务器。可以在尝试发起网络连接前检查一下网络状态，这样可以避免很多 OS 错误。请记住，这些设备可能会很快地进入或是离开网络。因此，在每次调用前检查网络的可达性是非常合情合理的事情。

iOS 的 SystemConfiguration 框架提供了多种方式来确定设备的网络连接状态。可以在 SCNetworkReachability 参考文档中找到关于底层 API 的详尽信息。这个 API 非常强大，不过也有点隐秘。幸好，Apple 提供了一个名为 Reachability 的示例程序，它为 SCNetworkReachability 实现了一个简化、高层次的封装器。Reachability 位于 iOS 开发者库中。

Reachability 封装器提供如下 4 个主要功能：

- 标识设备是否具备可用的网络连接
- 标识当前的网络连接是否可以到达某个特定的主机
- 标识当前使用的是哪种网络技术：Wi-Fi、WWAN 还是什么技术都没用
- 在网络状态发生变化时发出通知

要想使用 Reachability API，请从 iOS 开发者库中下载示例程序，地址是 http://developer.apple.com/library/ios/#samplecode/Reachability/Introduction/Intro.html，然后将 Reachability.h 与 Reachability.m 添加到应用的 Xcode 项目中。此外，还需要将 SystemConfiguration 框架添加到 Xcode 项目中。将 SystemConfiguration 框架添加到 Xcode 项目中需要编辑项目配置。图 5-3 展示了将 SystemConfiguration 框架添加到 Xcode 项目中所需的步骤。

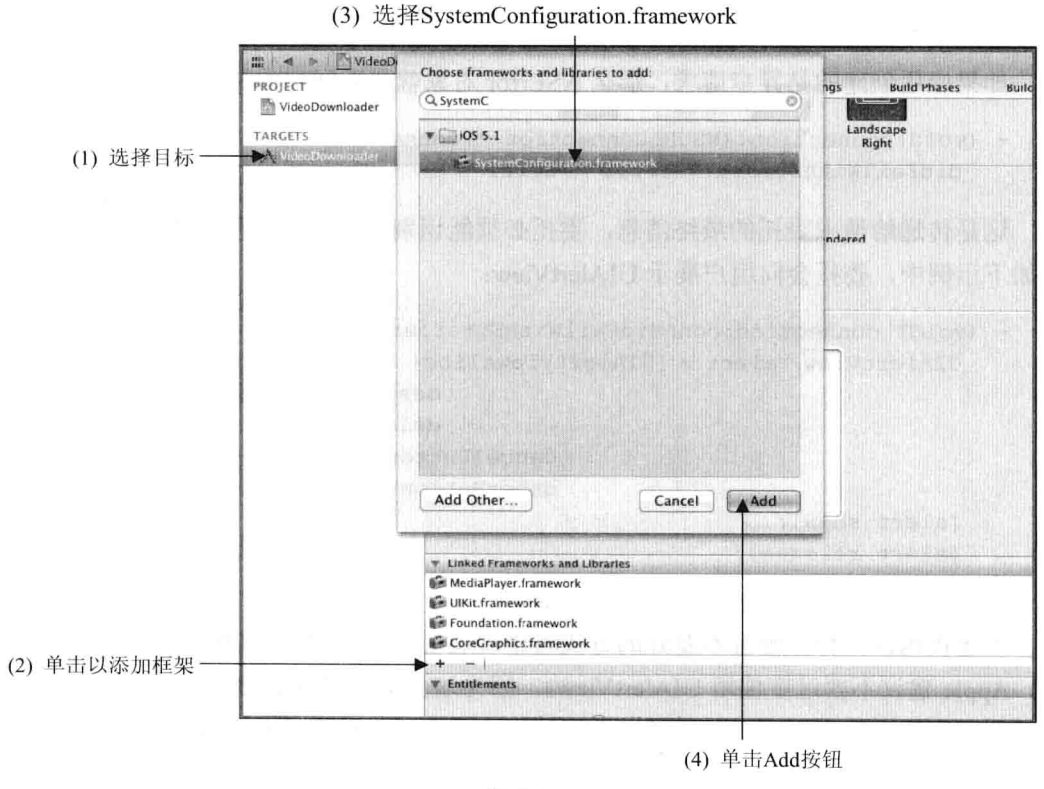


图 5-3

选定好项目目标后，找到设置中的 Linked Frameworks and Libraries，单击+按钮添加框架，这时会出现框架选择界面。选择 SystemConfiguration 框架，单击 add 按钮将其添加到项目中。

如下代码片段会检查是否存在网络连接。不保证任何特定的主机或 IP 地址是可达的，只是标识是否存在网络连接。

```
#import "Reachability.h"
...
if([[Reachability reachabilityForInternetConnection]
                              currentReachabilityStatus] == NotReachable) {
    // handle the lack of a network
}
```

在某些情况下，你可能想要修改某些动作、禁用 UI 元素或是当设备处于有限制的网络中时修改超时值。如果应用需要知道当前正在使用的连接类型，那么请使用如下代码：

```
#import "Reachability.h"
...
NetworkStatus reach = [[Reachability reachabilityForInternetConnection]
                                                  currentReachabilityStatus];
  if(reach == ReachableViaWWAN) {
     // Network Is reachable via WWAN (aka. carrier network)
  } else if(reach == ReachableViaWiFi) {
     // Network is reachable via WiFi
  }
```

知道设备可达性状态的变化也是很有必要的，这样就可以主动修改应用行为。如下代码片段启动对网络状态的监控：

```
#import "Reachability.h"
...
[[NSNotificationCenterdefaultCenter]
                               addObserver:self
                                  selector:@selector(networkChanged:)
                                     name:kReachabilityChangedNotification
                                   object:nil];
Reachability *reachability;
reachability = [[Reachability reachabilityForInternetConnection] retain];
[reachability startNotifier];
```

上述代码将当前对象注册为通知观察者，名为 kReachabilityChangedNotification。NSNotificationCenter 会调用当前对象的名为 networkChanged:的方法。当可达性状态发生变化时，就向该对象传递 NSNotification 及新的可达性状态。如下示例展示了通知监听者：

```
- (void) networkChanged: (NSNotification* )notification
{
   Reachability* reachability = [notification object];
```

```
    if(reachability == ReachableViaWWAN) {
        // Network Is reachable via WWAN (a.k.a. carrier network)
    } else if(reachability == ReachableViaWiFi) {
        // Network is reachable via WiFi
    } else if(reachability == NotReachable) {
        // No Network available
    }
}
```

可达性还可以确定当前网络上某个特定的主机是否是可达的。可以通过该特性根据应用是处于内部隔离的网络上还是公开的 Internet 上调整企业应用的行为。如下代码示例展示了该特性：

```
Reachability *reach = [Reachability
                reachabilityWithHostName:@"www.captechconsulting.com"];
if(reachability == NotReachable) {
    // The target host is not reachable available
}
```

请记住，该特性对目标主机的访问有个来回。如果每个请求都使用该特性，那就会极大增加应用的网络负载与延迟。Apple 建议不要在主线程上检测主机的可达性，因为尝试访问主机可能会阻塞主线程，这会导致 UI 被冻结。

OS 错误首先就表明请求出现了问题。应用开发者有时会忽略掉它们，不过这样做是有风险的。因为 HTTP 使用了分层网络，这时 HTTP 层或是应用层可能会出现其他类型的潜在失败情况。

5.1.2　HTTP 错误

HTTP 错误是由 HTTP 请求、HTTP 服务器或应用服务器的问题造成的。HTTP 错误通过 HTTP 响应的状态码发送给请求客户端。

404 状态是常见的一种 HTTP 错误，表示找不到 URL 指定的资源。下述代码片段中的 HTTP 头就是当 HTTP 服务器找不到请求资源时给出的原始输出：

```
HTTP/1.1 404 Not Found
Date: Sat, 04 Feb 2012 18:32:25 GMT
Server: Apache/2.2.14 (Ubuntu)
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 248
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1
```

响应的第一行有状态码。HTTP 响应可以带有消息体，其中包含友好、用户可读的信息，用于描述发生的事情。你不应该将是否有响应体作为判断 HTTP 请求成功与否的标志。

一共有 5 类 HTTP 错误：

- 信息性质的100级别——来自于HTTP服务器的信息，表示请求的处理将会继续，不过带有警告。
- 成功的200级别——服务器处理了请求。每个200级别的状态都表示成功请求的不同结果。比如，204表示请求成功，不过没有向客户端返回负载。
- 重定向需要的300级别——表示客户端必须执行某个动作才能继续请求，因为所需的资源已经移动了。URL 加载系统的同步请求方法会自动处理重定向而无须通知代码。如果应用需要对重定向进行自定义处理，那么应该使用异步请求。
- 客户端错误400级别——表示客户端发出了服务器无法正确处理的错误数据。比如，未知的URL或是不正确的HTTP头会导致这个范围内的错误。
- 下游错误500级别——表示HTTP服务器与下游应用服务器之间出现了错误。比如，如果Web服务器调用了JavaEE应用服务器，Servlet出现了NullPointerException，那么客户端就会收到500级别的错误。

iOS中的URL加载系统会处理HTTP头的解析，并可以轻松获取到HTTP状态。如果代码通过HTTP或HTTPS URL发出了同步调用，那么返回的响应对象就是一个NSHTTPURLResponse实例。NSHTTPURLResponse对象的statusCode属性会返回数值形式的请求的HTTP状态。如下代码演示了对NSError对象以及从HTTP服务器返回的成功状态的验证：

```
NSHTTPURLResponse *response=nil;
NSError *error=nil;
NSData *myData = [NSURLConnectionsendSynchronousRequest:request
                                        returningResponse:&response
                                                    error:&error];
//Check the return
if((!error) && ([response statusCode] == 200)) {
    // looks like things worked
} else {
    // things broke, again.
}
```

如果请求的URL不是HTTP，那么应用就应该验证响应对象是否是NSHTTPURLResponse对象。验证对象类型的首选方法是使用返回对象的isKindOfClass:方法，如下所示：

```
if([response isKindOfClass:[NSHTTPURLResponse class]]) {
    // It is a HTTP response, so we can check the status code
    ...
```

要想了解关于HTTP状态码的权威信息，请参考W3 RFC 2616，网址是http://www.w3.org/Protocols/rfc2616/rfc2616.html。

5.1.3 应用错误

本节将会介绍网络协议栈的下一层(应用层)产生的错误。应用错误不同于OS错误或

HTTP 错误，因为并没有针对这些错误的标准值或是原因的集合。这些错误是由运行在服务层之上的业务逻辑和应用造成的。在某些情况下，错误可能是代码问题，比如异常，不过在其他一些情况下，错误可能是语义错误，比如向服务提供了无效的账号等。对于前者来说，建议生成 HTTP 500 级别的错误；对于后者来说，应该在应用负载中返回错误码。

比如，如果用户尝试从账户中转账的金额超出了账户的可用余额，那么手机银行就应该报告应用错误。如果发出了这样的请求，那么 OS 会说请求成功发送并接收到了响应。HTTP 服务器会报告接收到了请求并发出了响应，不过应用层必须报告这笔交易失败。

报告应用错误的最佳实践是将应用的负载数据封装在标准信封中，信封中含有一致的应用错误位置信息。在上述资金转账示例中，成功的转账响应的业务负载应该如下所示：

```
{ "transferResponse":{
    "fromAccount":1,
    "toAccount":5,
    "amount":500.00,
    "confirmation":232348844
   }
}
```

响应包含了源账号与目标账号、转账的资金数额及确认号。直接将错误码与错误消息放到 transferResponse 对象中会导致错误码与错误消息的定位变得困难。如果每个动作都将错误信息放到自己的响应对象中，就无法在应用间重用错误报告逻辑了。使用如下代码中的数据包结构可以让应用快速确定是否出现了错误，方式是检查响应的 JSON 负载中是否存在“error”对象：

```
{"error":{
       "code":900005,
       "messages":"Insufficient Funds to Complete Transfer"
   },
   "data":{
       "fromAccount":1,
       "toAccount":5,
       "amount":500.00
   }
}
```

报告错误的 UI 代码是很容易重用的，因为错误信息总是位于响应负载的 error 属性中。此外，实际的交易负载处理得到了简化，因为它总是位于相同的属性名之下。

无论请求失败的原因是什么，OS、HTTP 层还是应用，应用都必须能知道如何作出响应。你应该在开发时就提前考虑好应用所有的失败模式，并设计好一致的方式来检测并响应错误。

5.2 错误处理的经验法则

错误可能是由多种原因造成的，最佳处理方式也随编写的应用不同而不同。虽然很复杂，不过有一些经验法则可以帮助处理错误原因不可控的本质。

5.2.1 在接口契约中处理错误

在设计服务接口时，只指定输入、输出与服务操作的做法是不正确的。接口契约还应该指定如何向客户端发送错误信息。服务接口应该使用业界标准方式在可能的情况下传递错误信息。比如，服务器不应该为服务端失败定义新的 HTTP 状态值；相反，应该使用恰当的 500 级别的状态。如果使用了标准值，那么客户端与服务端开发者就能对如何传递错误信息达成共识。应用绝不应该依赖于非标准的状态或是其他属性值来确定错误出现与否。

应用开发者也不应该依赖于当前服务器软件栈的行为来决定该如何处理错误。在部署了 iOS 应用后，服务器软件栈可能会由于未来的升级或替换而改变行为。

5.2.2 错误状态可能不正确

移动网络有如下有别于传统 Web 应用错误的不那么明显的行为：模糊不清的错误报告。从移动设备发往服务器的任何网络请求都有 3 种可能的结果：

- 设备完全能够确认操作是成功的。比如，NSError 与 HTTP 状态值都表明成功，返回的负载包含语义上正确的信息。
- 设备完全能够确认操作是失败的。比如，返回的应用负载包含来自于服务器的特定于本次操作的失败标识。
- 设备模糊地确认操作是失败的。比如，移动应用发出 HTTP 请求以在两个账户间转账。请求被银行系统接收并正确地处理；然而，由于网络失败应答却丢失了，NSURLConnection 报告超时。超时发生了，但却是在转账请求成功之后发生的。如果重试该操作，那就会导致重复转账，可能还会造成账户透支。

第 3 种场景会导致应用出现意外和检测不到的错误行为。如果应用开发者不知道第 3 种场景的存在，那么他们可能就会错误地假设操作失败，然后不小心重试已经成功的操作。知道整个操作失败还不够，开发者必须考虑导致请求失败的原因，以及自动重试每个失败的请求是否是恰当的。

5.2.3 验证负载

应用开发者不应该认为如果没有 OS 错误或 HTTP 错误，负载就是有效的。在很多场景下，请求似乎是成功的，不过负载却是无效的。客户端与服务器之间传递的负载都有一种验证机制。JSON 与 XML 就是具备了验证机制的负载格式，不过以逗号分隔的值文件与 HTML 就没有这种机制。

5.2.4 分离错误与正常的业务状况

服务契约不应该将正常的业务状况报告为错误。比如有个用户，由于可能的欺诈导致账户被锁定，锁定状态应该在数据负载中进行报告而不应该当作错误情况。分离错误与正常的业务状况会让代码保持恰当的关注分离。只有当出现问题时才应该将之看成错误。

5.2.5 总是检查HTTP状态

总是检查HTTP响应中的HTTP状态，理解成功的状态值，甚至向相同的服务发出重复的调用也是如此。服务器的状态可能随时会发生变化，甚至在并行的调用间也是如此。

5.2.6 总是检查NSError值

应用代码应该总是检查返回的NSError值来确保OS层没有出现问题。即便知道应用总是运行在信号良好的Wi-Fi网络下也应该这样做。任何东西都有出错的可能性，代码在处理网络时也需要做好防御工作。

5.2.7 使用一致的方法来处理错误

网络错误的产生原因是非常多的，很难一一列举出来，影响的多样性及范围也是非常大的。在设计应用时，请不要只关注于一致的用户界面模式或是一致的命名模式。你还应该设计一致的模式来处理网络错误。该模式应该考虑到应用可能会遇到的所有类型的错误。如果应用的内部没有以一致的模式处理这些错误，那么应用就无法以一致的方式向用户报告这些错误。

5.2.8 总是设置超时时间

在iOS中，HTTP请求的默认超时时间间隔是4分钟，这对于移动应用来说过长了，大多数用户都不会在任何应用中等待4分钟。开发者需要选择合理的超时时间，方式是评估网络请求的可能响应时间，然后将最差的网络场景下的网络延迟考虑进去。如下示例展示了如何创建具有20秒超时时间的请求：

```
- (NSMutableURLRequest *) createRequestObject:(NSURL *)url {
    NSMutableURLRequest *request = [[[NSMutableURLRequestalloc]
                                     initWithURL:url
                                     cachePolicy:NSURLCacheStorageAllowed
                                 timeoutInterval:20
                                autorelease];
    return request;
}
```

5.3 优雅地处理网络错误

iOS简化了网络通信，不过对可能发生的所有类型的错误与边界条件作出响应则不是

那么轻松的事情。常见的做法是在网络代码中放置钩子来快速查看结果，接下来再对所有的错误情况进行处理。对于非移动应用来说，通常可以使用这种方式，因为来自工作站的网络连接是可预测的。如果在应用加载时有网络，那么当用户加载下一个页面时基本上也会有网络。绝大多数情况都是这样的，开发者可以依赖浏览器向用户显示消息。如果在移动应用中没有及时添加异常处理，那么当后面遇到新的错误源时就需要大幅重构网络代码。

本节将会介绍一种设计模式，用来创建一个优雅且健壮的异常处理框架，并且在未来遇到新的错误时几乎不需要做什么工作就能很好地进行扩展。

考虑如下3个移动通信中的主要异常情况：

- 由于设备没有充分的网络连接导致远程服务器不可达。
- 由于OS错误、HTTP错误或是应用错误导致远程服务器返回错误响应。
- 服务器需要认证，而设备尝试发出未认证的请求。

随着可能的异常数量呈现出线性增长，处理这些异常的代码量则呈指数级增长。如果代码要在每一类请求中处理所有这些错误，那么代码的复杂性与数量就会呈指数级增长。本节将要介绍的模式会将这种指数级的曲线压成线性曲线。

5.3.1 设计模式介绍

本节介绍的模式联合使用了指挥调度模式与广播通知。该模式包含如下类型的对象：

- 控制器
- 命令对象
- 异常监听器
- 命令队列

下面从高层次来介绍每一类对象的行为。

1. 对象说明

下面介绍构成指挥调度模式的对象的特性及属性。

控制器

控制器通常指的是视图控制器，用来请求数据并处理结果。在该设计模式中，控制器无须包含任何异常处理逻辑。唯一需要控制器处理的错误情况就是成功完成或是完全不可恢复的服务失败。在不可恢复失败这个场景中，控制器通常会将自己从视图栈中弹出，因为用户这时已经收到接下来要介绍的异常监听器对象发出的失败通知了。控制器会创建命令并监听命令的完成情况。

命令对象

命令对象与应用执行的不同网络交易相关。检索图片、从指定的REST端点处获取JSON数据或是向服务发出POST信息等都是命令对象请求。命令对象是NSOperation的子类。由于命令对象中的大多数逻辑都与其他类型的命令对象相同，因此可以创建父类命令对象来处理，让特定的命令继承该逻辑。命令对象具有如下属性：

- 完成通知名——在iOS中，控制器会将自身注册为该通知名的观察者。当服务调用成功返回时，命令对象会通过NSNotificationCenter使用该名字来广播通知。虽然该名字对于命令类来说通常是唯一的，不过在某些情况下，如果有多个控制器发出相同类型的命令(区分不同的响应)，那么这个名字针对于特定的实例将是唯一的。
- 服务器错误异常通知名——特定的异常处理器对象会监听该通知。当服务器超时、返回与认证相关的OS错误或HTTP错误时，命令对象会通过NSNotificationCenter并使用该名字来广播消息。通常情况下，所有的命令类会共享相同的异常名，因此也会共享相同的异常监听器。不过不同的命令类可能需要使用不同的异常监听器，并且有不同的服务器错误异常名。
- 可达性异常通知名——当检测到无法到达Internet或目标主机时，命令对象会生成该类型的通知。另一个异常监听器可以监听该类型的异常。在某些应用中，这类异常是不需要的，因为服务器错误异常监听器会处理可达性异常。
- 认证异常通知名——如果确定用户没有认证或是服务器报告了未认证状态，那么命令对象可能会产生该类型的通知。第3个异常监听器会等待该类型通知的出现。认证通知名通常会在应用的所有通知中共享。
- 自定义属性——这些属性特定于发出的请求。控制器通常会提供这些值，因为它们特定于服务调用所需的业务数据，而且不同的调用数据也是不同的。

异常监听器

一般来说，每个异常监听器都是由应用委托实例化的，位于后台并等待着特定类型的通知。在很多情况下，异常监听器在接收到通知时会显示模态视图控制器，这将在“异常监听器行为”部分进行介绍。

命令队列

控制器会将命令提交到命令队列进行处理，应用可能有一个或多个命令队列。在iOS中，命令队列是NSOperationQueue的子类。不应该将主队列用作命令队列，因为它的操作运行在用户界面线程上，在执行长时间运行的操作时会影响到用户体验。NSOperationQueues提供了管理活动操作以及操作间依赖的内置功能。

2. 对象行为

上述每一个对象都在成功完成网络交易的过程中扮演着各自不同的角色。下面介绍它们在该模式中各自的角色。

控制器行为

控制器重点关注于执行UI与业务逻辑。当控制器想要从服务获取数据时，应该采取如下动作：

(1) 创建一个网络命令对象。

(2) 针对命令对象的具体属性初始化请求。

(3) 注册为命令完成的观察者。

(4) 将命令推送到操作队列中准备执行。

(5) 等待 NSNotificationCenter 发送完成通知。

当操作完成时，控制器会接收到完成通知并采取如下动作：

(1) 检查操作状态，看看操作是否成功。

(2) 如果成功，那么控制器会处理接收到的数据。接收到的数据是通过 NSNotification 对象的 userInfo 属性提供给控制器的。NSOperationQueues 会在自己的线程上执行 NSOperation 对象。当操作完成后，会通过 NSNotificationCenter 发送 NSNotification。该通知回调方法会在 NSOperation 运行的线程上得到调用，在该例中这会确保它不会进入主线程中。如果控制器操纵 UI，那么需要在主线程上做这些改变，通常是通过 Grand Central Dispatch(GCD)实现的。

(3) 如果不成功，那么控制器根据应用需求可以有很多选择。比如，可以将自己从视图栈中弹出或是更新 UI，表明数据不可用。控制器不应该重试或是显示模态警告，因为这些动作是异常监听器的职责。

(4) 控制器应该将自身从命令完成通知的观察者中移除。在某些情况下，如果控制器想要监控来自于相同类型命令的其他数据，那就没必要这么做了。

注意，控制器不包含处理重试、超时、认证或可达性的任何逻辑；这些逻辑都是由命令与异常监听器实现的。

如果控制器想要确保只有它才能接收到返回的数据，那么就应该在将其放到队列中之前改变通知名，将其改为针对该命令对象实例唯一的值，然后监听这个名字的通知。

命令对象行为

命令对象负责调用目标服务并将服务调用的结果广播出去。一般来说，命令对象需要执行如下步骤：

(1) 检查可达性。如果网络不可达，那就广播一条可达性异常通知。

(2) 如果需要，检查认证状态。如果用户尚未授权，那就广播一条可达性异常通知。

(3) 使用控制器提供的自定义属性构建网络请求。通常情况下，端点 URL 是命令对象类的静态属性或是从配置子系统中加载。

(4) 使用同步请求方式发出网络请求。参见第 3 章的 3.3.2 节“同步请求”以了解详情。

(5) 检查请求状态。如果状态是 OS 错误或 HTTP 错误，那就广播一条服务器异常通知。如果是认证错误，那就广播一条认证异常通知。

(6) 解析结果，参见第 4 章。

(7) 广播一条成功状态的完成通知。

当命令对象广播通知时，无论是成功还是其他通知，都需要创建字典对象，字典对象中包含自身的副本、调用状态与返回的数据，以此作为调用的结果。自身复制是有必要的，因为 NSOperation 实例只能执行一次。稍后将会介绍，在监听器处理异常时，命令可能会被再次提交。

同步请求API非常适合于该模式，因为命令是在后台线程而非主线程中执行的。如果请求发出或是返回的数据量超出期望在内存中处理的数据量，应用就需要使用异步请求了。因为NSOperation的主要功能是一个方法，操作必须实现并发锁来阻塞main方法，直到异步调用完成为止。

异常监听器行为

这种模式之所以如此强大，异常监听器功不可没。这些对象通常都是由应用委托创建的，驻留在内存中并监听着通知。当接收到通知时，监听器会通知用户，还可能会接收来自用户的响应。当异常发生时，通知中包含了触发该异常的命令副本，当用户作出响应后，监听器通常会再次将命令发回到队列中并重试。关于异常监听器有趣的一点是：由于多个命令可能同时发生，因此在用户响应第一个异常时可能还会同时产生多个异常通知。出于这一点，异常监听器必须收集异常通知，然后在用户响应完第一个异常后重新提交所有的触发命令。这个错误集合可以避免一种常见的应用行为不当——用户被相同问题触发的多个UIAlertView连续轰炸。

服务器异常的流程如下所示：

(1) 呈现一个漂亮的模态对话框，列出错误信息并让用户选择取消或是重试。

(2) 收集可能被广播的其他服务器异常。

(3) 如果用户选择重试，那么关闭对话框并重新提交所有收集到的命令。

(4) 如果用户选择取消，那么关闭对话框。监听器应该将所有收集到的命令的完成状态设为失败，然后让每个命令广播一条完成通知。

可达性异常的流程如下所示：

(1) 呈现一个漂亮的模态对话框，通知用户需要网络连接。

(2) 收集可能会被广播的其他服务异常。

(3) 监听可达性变更。当网络可达时，关闭对话框并重新提交收集到的命令。

认证异常的流程稍微有点复杂。请记住，命令之间是独立的，在任意时刻可能会有多个命令同时发生。认证流程并不会生成认证异常通知，流程如下所示：

(1) 呈现一个模态登录视图。

(2) 继续收集由于认证错误导致的失败命令。

(3) 如果用户取消，那么监听器应该针对收集到的命令使用失败状态发送一条完成通知。

(4) 如果用户提供了认证信息，那么创建一个登录命令，将其放到命令队列中。

(5) 等待登录命令的完成通知。

(6) 如果由于用户名/密码不匹配而导致登录失败，那么回到步骤(2)，否则关闭登录视图控制器。

(7) 如果登录命令成功，那么重新向命令队列提交触发命令。

(8) 如果登录命令失败，那么让触发命令使用失败状态发送一条完成通知。

命令队列行为

命令队列是原生的 iOSNSOperationQueue 对象。在默认情况下，命令队列遵循着先进先出(FIFO)的顺序。在代码向 NSOperationQueue 中添加命令对象后，执行如下动作：

(1) 保持命令对象，这样其内存就不会被释放掉。

(2) 等待，直到队列头有可用位置。

(3) 当命令对象到达队列头时，命令对象的 start 方法会被调用。

(4) 命令对象的 main 方法得到调用。

请参考 iOS API 文档中关于 NSOperation 与 NSOperationQueue 对象的介绍来了解队列与命令对象之间交互的详细信息。

5.3.2 指挥调度模式示例

本节通过调用 YouTube 的一项认证服务来介绍指挥调度模式。在此类通信过程中需要考虑很多失败模式：

- 用户可能没有提供有效的身份信息。
- 设备可能无法联网。
- YouTube 可能没有及时响应或是出于某些原因失败了。

应用需要以一种优雅且可靠的方式处理每一种情况。该例将会阐述主要的代码组件并介绍一些实现细节。

项目中的应用是个示例应用，只用于演示目的。

1. 前提条件

要想成功运行该应用，你需要准备好如下内容：

- 一个 YouTube 账号。
- 至少向你的 YouTube 账号上传一个视频(无须公开，只要上传到该账号即可)。
- 从 Wrox 网站上下载的项目压缩文件。

该项目使用 Xcode 4.1 与 iOS 4.3 开发，应用使用的是截止到 2011 年 10 月份的 YouTube API，不过该 API 处于 Google 的控制下，而且可能会发生变化。

2. 主要对象

下载好项目并在 Xcode 中加载后，你会看到如下类：

1) 命令

命令分组中有如下一些类。

BaseCommand

BaseCommand 是所有命令对象的父类。它提供了每个命令类所需的众多方法，这些方法有：

- 发送完成、错误与登录通知的方法。

- 用于让对象监听完成通知的方法。
- 用于支持实际的 NSURLRequests 的方法。

BaseCommand 继承了 NSOperation，因此所有的命令逻辑都位于该类的每个子类对象的 main 方法中。

GetFeed

如代码清单 5-1 所示，该类的 main 方法会调用 YouTube 并加载当前登录用户上传的视频列表。YouTube 通过请求 HTTP 头中的令牌来确定登录用户的身份。如果没有这个头，YouTube 就会返回 HTTP 状态码 0 而不是更加标准的 4xx HTTP 错误。

代码清单 5-1　CommandDispathDemo/service-interface/GetFeed.h

```
- (void)main {
    NSLog(@"Starting getFeed operation");
    // Check to see if the user is logged in
    if([self isUserLoggedIn]) { // only do this if the user is logged in


        // Build the request
        NSString *urlStr =
           @"https://gdata.youtube.com/feeds/api/users/default/uploads";
        NSLog(@"urlStr=%@",urlStr);
        NSMutableURLRequest *request =
              [ self createRequestObject:[NSURL URLWithString:urlStr]];

        // Sign the request with the user's auth token
        [self signRequest:request];

        // Send the request
        NSHTTPURLResponse *response=nil;
        NSError *error=nil;
        NSData *myData = [self sendSynchronousRequest:request
                                          response_p:&response
                                               error:&error];

        // Check to see if the request was successful
        if([super wasCallSuccessful:responseerror:error]) {
            [self buildDictionaryAndSendCompletionNotif: myData];
        }
    }
}
```

在上述代码清单中，通过 self 调用的很多方法都是在 BaseCommand 父类中实现的。GetFeed 命令就是指挥调度模式的原型。main 方法会对用户登录进行检查，因为如果这个调用失败了，那就没必要再调用服务器了。如果用户已经登录，那么代码就会构建请求，将认证头添加到请求中，然后发送一条同步请求。代码的最后一部分会调用一个父类方法来确定调用是否成功。该方法使用来自于 NSHTTPURLResponse 对象的 NSError 对象

与 HTTP 状态码来确定是否成功。如果调用失败，就会广播一条错误通知或是需要登录的通知。

LoginCommand

该命令会向 YouTube 发出对用户进行认证的请求。该命令比较独立，因为并没有使用 BaseCommand 对象的辅助方法。

之所以没有使用这些方法，是因为如果登录失败，就不应该生成需要认证的失败消息，而只会报告正常完成或是失败的状态。

登录监听器会处理来自于登录失败的错误。要想了解关于 YouTube 所需协议的详细信息，请参考 http://code.google.com/apis/youtube/2.0/developers_guide_protocol_understanding_video_feeds.html。

2) 异常监听器

监听器分组中有视图控制器，当错误发生或是用户需要登录时会呈现出来。NetworkErrorViewController 与 LoginViewController 都继承了 InterstitialViewController，后者提供了几个常用的辅助方法。这两个视图控制器都会以模态视图控制器的形式呈现出来。

- NetworkErrorViewController：向用户提供重试或是放弃失败操作的选择。如果用户选择重试，那么失败命令就会放回到操作队列中。
- LoginViewController：向用户请求用户名与密码。位于视图栈的顶部，直到用户成功登录为止。
- InterstitialViewController：作为其他异常监听器的父监听器，提供了一些支持功能，比如收集多个错误通知以及当错误解析完毕时重新分发错误的代码等。

监听器的关键代码位于 viewDidDisappear:方法中(如代码清单 5-2 所示)，当视图完全消失时会调用该方法。如果在视图完全消失前命令已进入队列中，那么其他错误就有可能导致再一次呈现视图，这会导致应用出现严重的错误。iOS 5 提供了处理这个问题的更好方式，因为在视图消失时用户可以指定执行的代码块。在处理触发命令前，代码并不需要确定消失的原因。

代码清单 5-2 CommandDispatchDemo/NetworkErrorViewController.m

```
- (void) viewDidDisappear:(BOOL)animated {
    if(retryFlag) {
        // re-enqueue all of the failed commands
        [self performSelectorAndClear:@selector(enqueueOperation)];
    } else {
        // just send a failure notification for all failed commands
        [self performSelectorAndClear:
                        @selector(sendCompletionFailureNotification)];
    }
    self.displayed = NO;
}
```

应用委托会将自身注册为网络错误与需要登录通知的监听器(如代码清单 5-3 所示)，收集异常通知并在错误发生时管理正确的视图控制器的呈现。

上述代码展示了需要登录通知的通知处理器。由于要处理用户界面，因此其中的内容必须使用 GCD 在主线程中执行。

代码清单 5-3　CommandDispatchDemo/CommandDispatchDemoAppDelegate.m

```
/**
* Handles login needed notifications generated by commands
**/
- (void) loginNeeded:(NSNotification *)notif {
    // make sure it all occurs on the main thread
    dispatch_async(dispatch_get_main_queue(), ^{
        // make sure only one thread adds a command at a time
        @synchronized(loginViewController) {
            [loginViewController addTriggeringCommand:
                                [notif object];
            if(!loginViewController.displayed) {
                // if the view is not displayed then display it.
                [[self topOfModalStack:self.window.rootViewController]
                          presentModalViewController:loginViewController
                          animated:YES];
            }
            loginViewController.displayed = YES;
        }
    }); // End of GC Dispatch block
}
```

3) 视图控制器

在这个简单的应用中有个主要的视图控制器。RootViewController(参见下面的代码)继承了 UITableViewController。当该控制器加载时，会创建并排队命令以加载用户的视频列表(又叫做 YouTube 种子)，并且会将控制流放回到主运行循环中以耐心等待命令的完成。第一次调用总是失败的，因为这时用户还没有登录。

CommandDispatchDemo/RootViewController.m 的 requestVideoFeed 方法会启动加载视频列表的过程，如下所示：

```
(void)requestVideoFeed {
    // create the command
    GetFeed *op = [[GetFeedalloc] init];

    // add the current authentication token to the command
    CommandDispatchDemoAppDelegate *delegate =
           (CommandDispatchDemoAppDelegate *)[[UIApplication
                                        sharedApplication] delegate ];
    op.token = delegate.token;

    // register to hear the completion of the command
```

```
    [op listenForMyCompletion:self selector:@selector(gotFeed:)];

    // put it on the queue for execution
    [op enqueueOperation];
    [op release];
}
```

注意，代码并不需要检查用户是否已经登录；在执行时命令会做检查。

gotFeed:方法会处理来自于 YouTube 的最终返回数据。在此例中，requestVideoFeed:方法会将 gotFeed:方法注册为完成通知的目标方法。如果调用成功，该方法会将数据加载到表视图中，否则显示 UIAlertView：

```
- (void) gotFeed:(NSNotification *)notif {
    NSLog(@"User info = %@", notif.userInfo);
    BaseCommand *op = notif.object;
    if(op.status == kSuccess) {
        self.feed = op.results;

        // if entry is a single item, change it to an array,
        // the XML reader cannot distinguish single entries
        // from arrays with only one element
        id entries = [[feed objectForKey:@"feed"] objectForKey:@"entry"];
        if([entries isKindOfClass:[NSDictionary class]]) {
            NSArray *entryArray = [NSArrayarrayWithObject:entries];
            [[feed objectForKey:@"feed"] setObject:entryArrayforKey:@"entry"];
        }
        dispatch_async(dispatch_get_main_queue(), ^{
            [self.tableViewreloadData];
        });
    } else {
        dispatch_async(dispatch_get_main_queue(), ^{
            UIAlertView *alert = [[UIAlertViewalloc]
                initWithTitle:@"No Videos"
                      message:@"The login to YouTube failed"
                     delegate:self
            cancelButtonTitle:@"Retry"
            otherButtonTitles:nil];
            [alert show];
            [alert release];
        });
    }
}
```

YouTubeVideoCell 是 UITableViewCell 的子类，它会异步加载视频的缩略图。它通过 LoadImageCommand 对象完成加载处理：

```
/**
 * Start the process of loading the image via the command queue
```

```
 **/
- (void) startImageLoad {
    LoadImageCommand *cmd = [[LoadImageCommandalloc] init];
    cmd.imageUrl = imageUrl;
    // set the name to something unique
    cmd.completionNotificationName = imageUrl;
    [cmd listenForMyCompletion:self selector:@selector(didReceiveImage:)];
    [cmdenqueueOperation];
    [cmd release];
}
```

这个类会改变完成通知名，这样它(也只有它)就可以接收到特定图片的通知了。否则，它还需要检查返回的通知来确定是否是之前发出的命令。

指挥调度模式的优雅之处在于能将应用中所有凌乱的异常处理逻辑和登录呈现逻辑与主视图控制器分离开来。当视图控制器发出命令时，会忽略掉所有的异常处理与认证处理，只是完成请求而已。只是发出请求，等待响应，然后处理响应。并不关心用户注册的请求是不是重试了5次才成功。此外，服务请求代码并不需要知道请求来自于哪里，结果去向哪里；只是关注于执行调用并广播结果。

指挥调度模式还有其他优势，开发者一开始会编写一些代码并论证结果，如果顺利，那么会添加异常处理器，而这对之前的代码不会造成任何影响。此外，如果设计恰当，那么所有的网络服务调用都会使用相同的基础命令类，这会减少命令类的数量。

在通用应用中，可以通过异常监听器调整展示的视图，这样iPhone上的错误显示界面就会适配于该平台，iPad上的错误显示界面也会适配于更大的平台。

这种模式可以快速展示结果，对业务逻辑与异常处理进行关注分离，减少重复代码以及提供更好的用户体验。

5.4 小结

代码使用网络时会出现很多错误源，理解错误源有助于快速诊断并解析网络问题。借助于Reachability框架，代码可以主动对变化的网络状况作出响应，从而避免不必要的网络错误的出现。在发出网络请求以及处理成功与失败的结果时遵循一致的模式，可以确保代码更加整洁、更加具有可维护性。

第 III 部分

高级网络技术

- 第 6 章　保护网络传输
- 第 7 章　优化请求性能
- 第 8 章　底层网络
- 第 9 章　测试与操纵网络流量
- 第 10 章　使用推送通知

第 6 章

保护网络传输

本章内容

- 如何验证应用与正确的服务器通信
- 使用 HTTP 与客户端证书对服务进行认证
- 如何生成密码散列并通过其验证负载的一致性
- 在 iOS 应用中加密与解密数据
- 使用设备的钥匙串存储认证信息的提示

本章代码下载

可以在 www.wrox.com 的 Download Code 选项卡中下载本章代码，网址是 www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html。代码位于第 6 章的压缩包中，并且分成两个主要部分：

- 一个 Xcode 项目，包含了一个基本的手机银行应用，它会与一个简单的 Web Service 通信。
- 一套 PHP 脚本，作为手机银行应用的 Web Service，它会处理认证、获取账号详细信息以及资金转账。

2011 年，与移动相关的数据泄漏的平均代价是每条记录 194 美金，每次事件中组织需要付出的平均代价是 550 万美金(援引自“2011 Cost of Data Breach Study”，Ponemon Institute© Research Report，2012 年 3 月，http://www.symantec.com/content/en/us/about/media/pdfs/b-ponemon-2011-cost-of-data-breach-us.en-us.pdf)。考虑到这个世界的网络本质，在应用生命周期的每个阶段都要强调安全的重要性。

为了满足安全相关的需求，Apple 向开发者提供了 Security 框架与 CommonCrypto 接口，可以在应用中使用。Security 框架是一套 C API 的集合，用于管理证书、信任策略以

及对设备安全数据存储的访问。CommonCrypto 是一套接口的集合，用于数据的加解密、生成常见的密码散列(比如 MD5 和 SHA1)，计算消息认证码，生成密码或是基于密码的密钥等。

本章将会介绍如何使用 Security 框架与 NSURLConnection 来验证客户端与服务器的身份，还会介绍常见的认证模式，并提供一个关于加密传输数据的示例。最后，本章将会介绍如何解密服务器响应，并使用设备的钥匙串安全地存储认证信息。

下载的手机银行应用有助于阐述本章介绍的各个知识点。该应用包含一个服务端组件，出于简化的目的使用 PHP 开发。PHP 非常直接，即便之前没什么经验也能很轻松地理解。本章将会介绍重要的服务端代码片段，完整的服务端代码位于本书第 6 章的压缩包中。

6.1　验证服务器通信

应用的用户很可能会到处走动，毕竟这是移动应用，而且也没法确保应用到 Internet 的连接是安全的，不会被其他人窥探。大多数咖啡馆都会向顾客提供免费 Wi-Fi，不过这些网络很容易被窃听，用户可能都感知不到。确保用户只与期望的服务器进行通信是开发者的职责。

最佳实践是使用NSURLProtectionSpace验证手机银行应用的用户与安全的银行服务器进行通信，特别是在发出的请求会操纵后端数据时更是如此。NSURLProtectionSpace 表示需要认证的服务器或域，是所有进来的 NSURLAuthenticationChallenges 的一个属性。

如下代码片段展示了如何创建保护空间，可以将其与 challenge 中包含的信息做对比：

```
NSURLProtectionSpace *defaultSpace =
[[NSURLProtectionSpace alloc]
initWithHost:@"yourbankingdomain.com"
port:443
protocol:NSURLProtectionSpaceHTTPS
realm:@"mobile"
authenticationMethod:NSURLAuthenticationMethodDefault];
```

注意这里指定的端口号是 443，对应于 NSURLProtectionSpaceHTTPS 协议。表 6-1 列出了其他支持的协议及常见的端口号。如果不确定服务器配置的是哪个端口号，可以将进来的 challenge 的属性打印到控制台。

表 6-1 支持的 NSURLProtectionSpace 协议

协 议 常 量	默 认 端 口
NSURLProtectionSpaceHTTP	80 或 8080
NSURLProtectionSpaceHTTPS	443
NSURLProtectionSpaceFTP	21 或 22

应用指定了 NSURLAuthenticationMethodDefault 的认证方法。NSURLProtectionSpaceHTTP 协议的默认认证方式是 Basic 认证，因此在这种情况下，指定 nil 或 NSURLAuthenticationMethodHTTPBasic 与 NSURLAuthenticationMethodDefault 是一回事。下面列出所有支持的认证方法：

- NSURLAuthenticationMethodDefault
- NSURLAuthenticationMethodHTTPBasic
- NSURLAuthenticationMethodHTTPDigest
- NSURLAuthenticationMethodHTMLForm
- NSURLAuthenticationMethodNTLM
- NSURLAuthenticationMethodNegotiate
- NSURLAuthenticationMethodClientCertificate
- NSURLAuthenticationMethodServerTrust

既然已经使用服务器的属性创建了保护空间，你需要确保将其用于验证连接。在代码向需要认证的服务器请求资源时，服务器会使用 HTTP 状态码 401 进行响应，即访问拒绝。NSURLConnection 会接收到该响应并立刻使用认证 challenge 的一份副本来发送一条 willSendRequestForAuthenticationChallenge:委托消息。图 6-1 展示了基本的挑战-响应的过程。

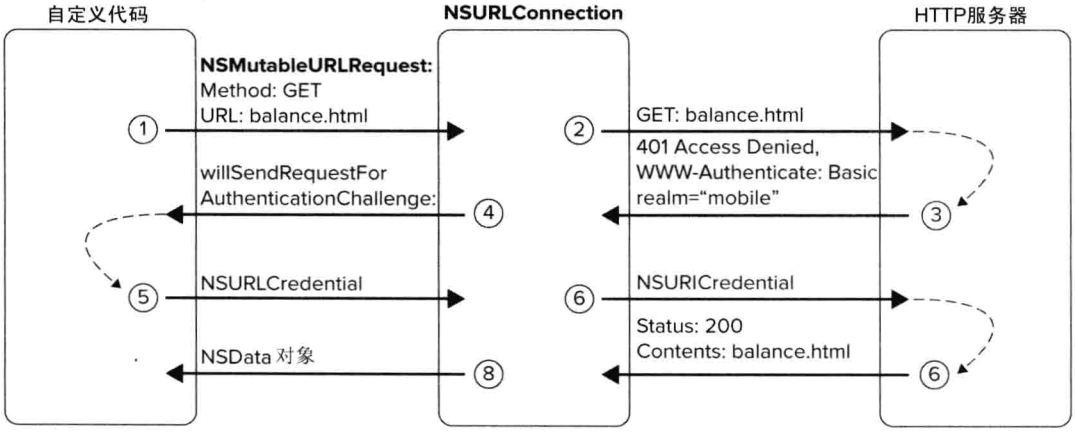


图 6-1

实现 willSendRequestForAuthenticationChallenge:可以检查挑战，确定是否想要响应服务器的认证挑战，同时发出恰当的挑战响应。认证挑战响应是 NSURLCredential 的实例，可用于创建信任、用户名/密码组合和客户端证书，下一节将会对此进行详细介绍。在创建服务器信任的 NSURLCredential 时，委托需要对信任做出判断。

如下示例在 willSendRequestForAuthenticationChallenge:中实现了一种可能的保护空间验证：

```
- (void)connection:(NSURLConnection *)connection
     willSendRequestForAuthenticationChallenge:
```

```
    (NSURLAuthenticationChallenge *)challenge {

    // create an array of protection spaces for confirmation
    NSURLProtectionSpace *defaultSpace =
        [[NSURLProtectionSpace alloc]
            initWithHost:@"yourbankingdomain.com"
                    port:443
                protocol:NSURLProtectionSpaceHTTPS
                   realm:@"mobilebanking"
    authenticationMethod:NSURLAuthenticationMethodDefault];

    NSURLProtectionSpace *trustSpace =
        [[NSURLProtectionSpace alloc]
            initWithHost:@"yourbankingdomain.com"
                    port:443
                protocol:NSURLProtectionSpaceHTTPS
                   realm:@"mobilebanking"
    authenticationMethod:NSURLAuthenticationMethodClientCertificate];
    NSArray *validSpaces =
        [NSArray arrayWithObjects:defaultSpace, trustSpace, nil];
    // validate that the authentication challenge
    // came from a whitelisted protection space
    if(![validSpaces containsObject:challenge.protectionSpace]) {

        // dispatch alert view message to the main thread
        NSString *msg =
         @"We're unable to establish a secure connection.
            Please check your network connection and try again.";
        dispatch_async(dispatch_get_main_queue(), ^{
            [[[UIAlertView alloc] initWithTitle:@"Unsecure Connection"
                                        message:msg
                                       delegate:nil
                              cancelButtonTitle:@"OK"
                              otherButtonTitles:nil] show];
        });
        // cancel authentication attempt
        [challenge.sender cancelAuthenticationChallenge:challenge];
    }
    ...
}
```

本节已经介绍了如何创建保护空间，不过上述代码片段已添加额外的保护空间，这为后端提供了一些灵活性。当确定要支持的保护空间后，请先创建它们，然后将它们添加到数组中以便与进来的认证挑战相比较。实际上，你应该定义有效的保护空间作为模型层的一部分，这样就可以在所有网络操作中重用它们了。如果认证挑战中的保护空间与所有支持的空间不匹配，那么你应该通知用户并取消认证挑战。

上述示例中的代码会通过 Grand Central Dispatch 向主线程发出 UIAlertView。这么做是必要的，因为应用执行的网络活动的每个逻辑单元都是 NSOperation 的子类，通常会在后

台线程中处理。然而，手机银行应用会在主线程上异步发出请求，这样应用就可以响应willSendRequestForAuthenticationChallenge:委托方法。在该场景中，对 Grand Central Dispatch 的使用是一种安全保障手段。第 7 章“优化请求性能”将会介绍一种更加适合的网络模式，并提供一种高效的方式来通知视图控制器哪些情况需要用户操作。

既然已经实现了服务器验证，那它是如何保护应用的用户呢？这种特定的验证会确保应用只与指定的服务器进行通信。如果发现处于恶意网络之上，传输被重新路由到第三方服务器(比如 yourbankingdomain.phishing.com)，那么保护空间验证就会因不匹配的主机而失败，后续的通信也会终止。更为重要的是，登录认证信息、银行账号等信息是绝不会传输的。

复杂的 iOS 应用常常会与 Web Service 通信，这些服务可能经常会发生变化。但遗憾的是，iOS 应用的修改需要经由 Apple 审批，这个结果是不可预知的。组织肯定不希望遇到必须立刻修改 Web Service 认证的情况，否则应用就无法正常运行。要想解决这个风险问题，应用包含的保护空间需要考虑到与备份认证服务器或其他备选服务器通信的问题。包含多个保护空间可以实现一定程度的灵活性，不过这最终是每个组织都需要评估的安全决策。

实现后端灵活性的另一种方式是只验证认证挑战中的某些属性，比如主机、端口与协议是否与预先定义好的相匹配。比如，可以验证挑战是从使用 SSL、端口号为 443 的特定主机发出的。只要不满足其中任何一个条件，代码就会立刻向用户发出警告，告诉用户无法建立安全的连接。

```
if(![challenge.protectionSpace.host
       isEqualToString:@"yourbankingdomain.com"] ||
   !challenge.protectionSpace.port == 443 ||
   ![challenge.protectionSpace.protocol
       isEqualToString:NSURLProtectionSpaceHTTPS]) {

   // if ANY of our challenge verifications fail, alert the user
   dispatch_async(dispatch_get_main_queue(), ^{
       NSString *msg = @"We're unable to establish a secure connection."
       [[[UIAlertView alloc] initWithTitle:@"Unsecure Connection"
                                   message:msg
                                   delegate:nil
                         cancelButtonTitle:@"OK"
                         otherButtonTitles:nil] show];
   });

   // cancel authentication
   [challenge.sender cancelAuthenticationChallenge:challenge];
}
```

服务器验证是非常重要的，不过尚不足以阻止所有的攻击。比如，无法阻止中间人攻击，这指的是有人窃取网络通信的情况。要想确保用户数据的安全，还需要仔细评估和考虑其他一些安全度量，比如消息完整性与数据加密等。

6.2　HTTP 认证

认证指的是确认访问系统的用户身份的过程。对于手机银行应用的服务层来说，最重要的事情就是要能分辨出真实用户与虚假用户之间的差别。本节将会介绍常见的认证模式以及如何在 iOS 应用中处理这些挑战。

手机银行应用有两种认证模式：标准验证与快速验证。标准验证只是提示用户输入用户名与密码，而快速验证则让用户注册设备，然后使用 PIN 进行验证，每次验证时无需用户名与密码。要想确保快速认证的安全性，如果用户选择在给定的认证请求中注册设备，那么服务器响应就需要包含一个额外的属性，即用户的证书。应用会存储这个证书，并在随后启动时检查，从而确定应该显示哪个认证视图。

手机银行应用的标准认证模式使用 HTTP Basic 认证，而快速认证则使用从 Web Service 下载的客户端证书。接下来将会介绍每一种方式。

6.2.1　HTTP Basic、HTTP Digest 与 NTLM 认证

Basic、Digest 与 NTLM 认证都是基于用户名/密码的认证。这意味着可以通过相同的逻辑来处理所有这 3 种类型的认证挑战。Basic 与 Digest 认证要比 NTLM 认证用得多一些，不过 NTLM 认证也有人在用。

HTTP Basic 认证是由 RFC 1945(http://tools.ietf.org/html/rfc1945)定义的，顾名思义，这是一种很基本的认证方式。明文传送的用户名与密码信息很容易被拦截和篡改。然而，如果搭配 SSL，那么这些弱点也是可以接受的，这种组合也是一种常见的认证模式。

HTTP Digest 认证最初是由 RFC 2069(http://tools.ietf.org/html/rfc2069)定义的，是一种更加安全的认证形式，在传输密码前会对其进行 MD5 哈希处理，同时会配以密码随机数。密码随机数是个随机数或伪随机数，用于对消息进行签名，不过每个值只能使用一次。由于每个随机数只会使用一次，然后就会被标记为过期，因此可以防止重放攻击(重新发送之前的密文)。HTTP Basic 与 HTTP Digest 认证现在已经合并成一个标准，即 RFC 2617(http://tools.ietf.org/html/rfc2617)。

NTLM 是微软推出的安全协议，提供了认证、完整性与加密服务。NTLM 认证是一项类似于 HTTP Basic 与 HTTP Digest 认证的挑战——响应协议。它在很大程度上已经被 Kerberos 系统取代，不过依然还继续用于认证 Web 上的远程用户。Kerberos 是由 MIT 基于“tickets”想法而开发出来的认证协议，可以在不安全的网络上实现安全的身份识别。

幸好，NSURLConnection 会完成各种认证方法的随机数与哈希值的处理工作，这样只需以 NSURLCredential 对象的形式指定认证信息就可以了。NSURLCredential 适合于大多数的认证请求，因为它可以表示由用户名/密码组合、客户端证书以及服务器信任创建的认证信息。认证信息有各种持久化选项：不持久化、只对当前会话持久化以及永久持久化。应用只能访问由它们创建的认证信息，用户可以授权要访问的应用，这一点与传统的 Mac 开发是类似的。

HTTP Basic、HTTP Digest 与 NTLM 认证的响应逻辑是相同的。代码清单 6-1 展示了通过 willSendRequestForAuthenticationChallenge:方法响应用户名与密码的认证挑战过程。

代码清单 6-1 处理 Basic 认证挑战(/App/Mobile-Banking/AuthenticateOperation.m)

```
- (void)connection:(NSURLConnection *)connection
      willSendRequestForAuthenticationChallenge:
         (NSURLAuthenticationChallenge *)challenge {


...

   // respond to basic authentication requests
   // DIGEST and NTLM authentication follow this pattern
   if(challenge.protectionSpace.authenticationMethod ==
         NSURLAuthenticationMethodHTTPBasic) {
      // proceed with authentication
      if(challenge.previousFailureCount == 0) {
         NSURLCredential *creds =
            [[NSURLCredential alloc]
               initWithUser:_username
                  password:_password
               persistence:NSURLCredentialPersistenceForSession];

         [challenge.sender useCredential:creds
            forAuthenticationChallenge:challenge];

      // authentication has previously failed.
   // depending on authentication configuration, too
      // many attempts here could lead to a poor user
   // experience via locked accounts

      } else {

         // cancel the authentication attempt
         [[challenge sender] cancelAuthenticationChallenge:challenge];

         // alert the user that his credentials are invalid
         // this would typically be handled in a cleaner
   // manner such as updating the styled login view
   NSString *msg = @"Invalid username / password.";
         dispatch_async(dispatch_get_main_queue(), ^{
            [[[UIAlertView alloc]
               initWithTitle:@"Invalid Credentials"
                     message:msg
                    delegate:nil
            cancelButtonTitle:@"OK"
            otherButtonTitles:nil] show];
         });
      }
```

```
    }
...
}
```

在确定挑战是针对 HTTP Basic 或另一种支持的挑战类型后，应该确保挑战没有失败，并使用用户输入的用户名与密码创建 NSURLCredential 对象。如果挑战失败，那就警告用户并取消挑战。这是非常重要的，因为 willSendRequestForAuthenticationChallenge:可能会被调用多次。根据配置，如果用户的认证信息不合法而又没有恰当地进行检查，那就有可能在用户只提交一次不合法的认证信息后，账户就被锁定了。如果进来的挑战认证方法并不是应用所能处理的类型，那就不要发出响应。这会告诉 NSURLConnection，应用无法处理这种认证方法。

6.2.2　客户端证书认证

既然用户已经认证成功，现在我们假设用户在这个认证请求中注册了设备。在设备注册过程中，应用必须存储认证服务返回来的证书。如下代码展示了包含着证书数据的成功的服务层响应：

```
{
    "result": "SUCCESS",
    "additional_info": "Authentication Successful",
    "certificate": "<BASE64 Encoded Certificate>
}
```

上述代码片段中返回的证书数据被编码成了 PKCS #12(.p12)文件格式，这是由 RSA Laboratories 发布的一种常用标准，用于与客户端应用交换证书数据。代码清单 6-2 与 6-3 展示了如何解码 Base 64 .p12 数据，提取出标识与证书信息，并将它们存储起来以供后续的认证请求使用。

代码清单 6-2　处理证书数据的认证响应(/App/Mobile-Banking/AuthenticateOperation.m)

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection {
    ...

    // unpack service response
    NSError *error = nil;
    NSDictionary *response = [NSJSONSerialization
                              JSONObjectWithData:self.responseData
                                         options:0
                                           error:&error];


    ...

    // create our client certificate if necessary,
    // and store in the credential store
    if(_registerDevice == YES) {
```

```
        NSString *certString = [response objectForKey:@"certificate"];
        NSData *certData =
            [NSData dataWithBase64EncodedString:certString];

        // retrieve the identity and certificate for our decoded data
        SecIdentityRef identity = NULL;
        SecCertificateRef certificate = NULL;
        [Utils identity:&identity
         andCertificate:&certificate
         fromPKCS12Data:certData
         withPassphrase:@"test"];

        // store the certificate for future authentication challenges
        if(identity != NULL) {

              // store the certificate and identity in
              // the keychain as the default
              NSURLProtectionSpace *certSpace =
               [[NSURLProtectionSpace alloc]
                  initWithHost:@"yourbankingdomain.com"
                          port:443
                      protocol:NSURLProtectionSpaceHTTPS
                         realm:@"mobilebanking"
    authenticationMethod:
                        NSURLAuthenticationMethodClientCertificate];

        NSArray *certArray = [NSArray arrayWithObject:
                              (__bridge id)certificate];
        NSURLCredential *credential =
            [NSURLCredential
            credentialWithIdentity:identity
                      certificates:certArray
                      `persistence:
                            NSURLCredentialPersistencePermanent];

        [[NSURLCredentialStorage sharedCredentialStorage]
          setDefaultCredential:credential
            forProtectionSpace:certSpace];
        }
    }
    ...
}
```

代码清单 6-2 的大部分内容都是非常直接的，不过在检索服务返回的.p12 证书数据中的身份与证书时可能会出现异常。代码清单 6-3 详尽阐述了如何使用 Security Framework 中的 SecPKCS12Import()函数导入身份和信任，然后提取出证书的过程。

代码清单 6-3　从.p12 数据中获取身份与证书(/App/Mobile-Banking/Utils.m)

```
+ (void)identity:(SecIdentityRef*)identity
```

```
andCertificate:(SecCertificateRef*)certificate
fromPKCS12Data:(NSData*)certData
withPassphrase:(NSString*)passphrase {

    // bridge the import data to foundation objects
    CFStringRef importPassphrase = (__bridge CFStringRef)passphrase;
    CFDataRef importData = (__bridge CFDataRef)certData;

    // create dictionary of options for the PKCS12 import
    const void *keys[] = { kSecImportExportPassphrase };
    const void *values[] = { importPassphrase };
    CFDictionaryRef importOptions = CFDictionaryCreate(NULL, keys,
                                                       values, 1,
                                                       NULL, NULL);

    // create array to store our import results
    CFArrayRef importResults = CFArrayCreate(NULL, 0, 0, NULL);
    OSStatus pkcs12ImportStatus = errSecSuccess;

    pkcs12ImportStatus = SecPKCS12Import(importData,
                                         importOptions,
                                         &importResults);

    // check if import was successful
    if(pkcs12ImportStatus == errSecSuccess) {
        CFDictionaryRef identityAndTrust =
                                CFArrayGetValueAtIndex (importResults, 0);

        // retrieve the identity from the certificate imported
        const void *tempIdentity = NULL;
        tempIdentity = CFDictionaryGetValue (identityAndTrust,
                                             kSecImportItemIdentity);
        *identity = (SecIdentityRef)tempIdentity;

        // extract the certificate from the identity
        SecCertificateRef tempCertificate = NULL;
        OSStatus certificateStatus = errSecSuccess;
        certificateStatus = SecIdentityCopyCertificate (*identity,
                                                        &tempCertificate);
        *certificate = (SecCertificateRef)tempCertificate;
    }

    // clean up
    if(importOptions) {
        CFRelease(importOptions);
    }
}
```

作为服务器与客户端 SSL 握手的一部分，willSendRequestForAuthenticationChallenge: 会在服务器信任与客户端证书认证挑战的过程中收到多个回调。你需要确定应用该处理哪

个挑战。如下代码示例是对代码清单 6-1 的扩展，用于确定应用是否应该发出客户端证书或是标准的用户凭证来进行认证：

```
// if this is a client certificate authentication request AND
// the user has already registered this device, attempt to issue
// the certificate to the service tier
if(challenge.protectionSpace.authenticationMethod ==
     NSURLAuthenticationMethodClientCertificate
     && devicePreviouslyRegistered) {

   // proceed with authentication
   if(challenge.previousFailureCount == 0) {

   // retrieve the default credential specifically for
   // client certificate challenges
   NSURLCredential *credential =
      [[NSURLCredentialStorage sharedCredentialStorage]
        defaultCredentialForProtectionSpace:
          [[Model sharedModel] clientCertificateProtectionSpace]];
      if(credential) {
         [challenge.sender useCredential:credential
            forAuthenticationChallenge:challenge];
      }

   // authentication has previously failed.
   // depending on authentication configuration, too many attempts
   // here could lead to a poor user experience via locked accounts
   } else {
      // cancel the authentication attempt
      [[challenge sender] cancelAuthenticationChallenge:challenge];

      // alert the user that his credentials are invalid
      // this would typically be handled in a cleaner
      // manner such as updating the styled login view

   }

// either the user has not registered this device or
// this is not a client certificate challenge
} else {
      ...
      // perform authentication based on Listing 6-1
}

// if nothing catches this challenge,
// attempt to connect without credentials
 [challenge.sender
 continueWithoutCredentialForAuthenticationChallenge:challenge];
```

在服务层，可以通过 openssl_x509_parse()函数检索到证书的属性。在获取到证书属性

后，你还可以使用服务层的很多认证选项。其中一个选项是验证请求发起人，然后从已知的私钥列表中查找该用户。另一个选项是在应用中使用 PIN 机制，从而在向认证挑战发出客户端证书之前进行验证。

```
if(array_key_exists('SSL_CLIENT_CERT', $_SERVER)){
    $clientCertData = openssl_x509_parse($_SERVER['SSL_CLIENT_CERT']);
    // using certificate attributes and encrypted PIN
    // verify identity and issue authentication tokens
} else {
    // issue failed authentication message
}
```

NSURLConnection 会拦截带有不受信任证书的服务器响应，这包括自签的 SSL 证书。如果使用自签的 SSL 证书来测试本章介绍的认证机制，那么大多数基于网络的代码都将无法执行。不过，将服务器证书(.cer 文件扩展)以邮件的形式发送给设备上配置好的 E-Mail 账户，就可以单击并安装证书了。安装完毕后，NSURLConnection 请求就会将服务器证书看作受信的证书并会继续处理。

6.3　使用哈希与加密确保消息完整性

既然应用已经验证在与正确的服务器通信并已被成功认证，那么用户就可以开始发出服务请求了。应用必须确保传输的数据在传输过程中是安全且未被修改的。本节将会介绍能够满足这两个需求的技术，包括密码散列、消息认证码(Message Authentication Code，MAC)以及加密移除。

这些主题将会通过手机银行应用的资金转移功能来实现(可以在本书的网站上下载)。资金转移请求会联合使用密码哈希与加密的方式来确保消息是难以理解的并且在传输阶段是没有经过修改的。虽然本节将会生成和介绍很多负载示例，不过每个示例都会使用如下代码定义的 JSON 负载结构：

```
{
    "mac": "Message Authentication Code",
    "iv": "Initialization Vector",
    "payload": {
        "toAccount": "123123456456",
        "fromAccount": "654654321321",
        "amount": 23.23,
        "transferDate": "2012-02-27",
        "transferNotes": "Book advance to savings."
    }
}
```

payload 属性是请求体中唯一被加密的元素。服务层要想正确解密负载，就得要求应用使用 Initialization Vector(IV)来对数据进行加密。通常情况下，IV 会随着被它加密的数据一

起发送。虽然这么做似乎降低了消息的安全性，但实际上并不会，因为单凭 IV 本身不足以解密消息。当服务层解密负载后，它会使用同样的预定义的负载属性集合来生成 MAC，然后将其与进来的请求中的 MAC 进行对比以验证消息的完整性。如果消息在传输过程中被修改，那么代码就不会匹配，被修改的消息则会产生错误。本节示例应用使用的属性集是 To Account、From Account、Amount 与 Transfer Date 的拼接，在后面介绍消息认证码时你就会看到。

图 6-2 概览了请求、响应交易中发生的步骤，以及在交易每一端的加密与解密过程中使用的组件。交易双方都知道用于加密与解密的重要值和 MAC 算法。此外，请求体部分与之前示例中定义的 JSON 负载结构是匹配的。

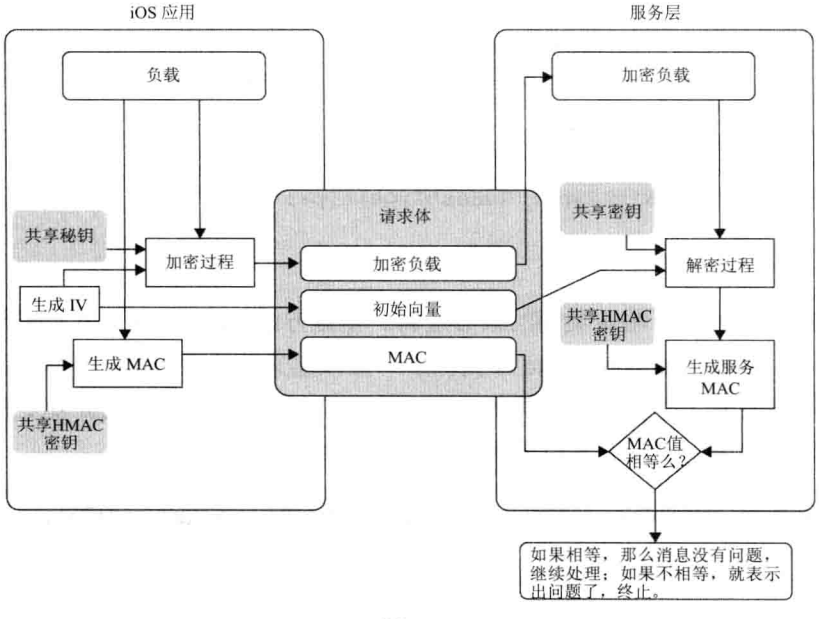


图 6-2

6.3.1　哈希

对于给定的数据块，密码哈希与摘要会生成固定大小的位序列。这些哈希值可以简化数据块的比较与排序。哈希的常见使用场景包括追踪文件变更、下载校验和、数据混淆以进行数据库存储，以及验证请求数据的完整性等。本节后面将会介绍一种使用 MAC 的更为健壮的方式。

iOS CommonCrypto 库提供了对 MD5、SHA-1、SHA-256 摘要以及其他不太常用的加密算法的支持。本章的示例将会专注于 MD5、SHA-1 与 SHA-256，不过对于所有的 CommonCrypto 摘要算法，生成哈希值的过程是一致的。可以通过一系列函数调用创建摘要上下文来手工创建哈希值，然后使用待处理的数据更新上下文，最后获取摘要计算的值；也可以使用每个摘要算法提供的便捷函数。

为了简化应用的实际使用，在 NSString 上创建一个类别来实现哈希方法。

NSString+Hashing.h/.m 中放置了所有与哈希相关的类别方法，其接口定义如代码清单 6-4 所示。

代码清单 6-4　NSString Hashing 类别定义(/App/Mobile-Banking/NSString+Hashing.h)

```
#import <CommonCrypto/CommonDigest.h>

enum {
    NJHashTypeMD5 = 0,
    NJHashTypeSHA1,
    NJHashTypeSHA256,
}; typedef NSUInteger NJHashType;

@interface NSString (Hashing)

- (NSString*)md5;
- (NSString*)sha1;
- (NSString*)sha256;
- (NSString*)hashWithType:(NJHashType)type;

@end
```

代码清单 6-5 展示了哈希类别的核心逻辑。便捷方法 md5、sha1 与 sha256 都会调用 hashWithType:。虽然 CommonCrypto 库提供了一些预定义的哈希枚举，应用可以直接使用，不过这些枚举都没有定义在摘要计算的上下文中。相对于依赖未来可能会发生变化的枚举，应用使用自定义的值可以限制支持的摘要算法，这也是额外的好处。

代码清单 6-5　hashWithType 中的核心摘要计算逻辑(/App/Mobile-Banking/NSString+Hashing.m)

```
- (NSString*)hashWithType:(NJHashType)type {

    // Create pointer to the string as UTF8 - this is NULL terminated
    const char *ptr = [self UTF8String];

    // Create buffer with length for chosen digest
    NSInteger bufferSize;
    switch(type) {
        case NJHashTypeMD5:
            // 16 bytes
            bufferSize = CC_MD5_DIGEST_LENGTH;
            break;

        case NJHashTypeSHA1:
            // 20 bytes
            bufferSize = CC_SHA1_DIGEST_LENGTH;
            break;

        case NJHashTypeSHA256:
            // 32 bytes
```

```
            bufferSize = CC_SHA256_DIGEST_LENGTH;
            break;

        default:
            return nil;
            break;
    }

    unsigned char buffer[bufferSize];

    // Perform hash calculation and store in buffer
    switch(type) {
        case NJHashTypeMD5:
            CC_MD5(ptr, strlen(ptr), buffer);
            break;

        case NJHashTypeSHA1:
            CC_SHA1(ptr, strlen(ptr), buffer);
            break;

        case NJHashTypeSHA256:
            CC_SHA256(ptr, strlen(ptr), buffer);
            break;

        default:
            return nil;
            break;
    }

    // Convert buffer value to pretty printed NSString
    // this will match the servers hash calculation
    NSMutableString *hashString = [NSMutableString
                                    stringWithCapacity:bufferSize * 2];
    for(int i = 0; i < bufferSize; i++) {
        [hashString appendFormat:@"%02x",buffer[i]];
    }

    return hashString;

}
```

hashWithType:实现中唯一不太直接的地方就是最后一步。代码清单 6-5 的最后一步会循环摘要计算的字节输出，然后将其转换为十六进制，即可读的输出。完成好核心的哈希逻辑并将其放到一个方法中后，实现每个便捷方法就只需要一行代码即可，如代码清单 6-6 所示。

代码清单 6-6　哈希便捷方法的实现(/App/Mobile-Banking/NSString+Hashing.m)

```
- (NSString*)md5 {
```

```
    return [self hashWithType:NJHashTypeMD5];
}

- (NSString*)sha1 {
    return [self hashWithType:NJHashTypeSHA1];
}

- (NSString*)sha256 {
    return [self hashWithType:NJHashTypeSHA256];
}
```

这种方式的额外好处就是可以轻松扩展以支持更多的摘要计算。下面调用每个便捷方法并显示相应的输出(美国国家标准协会(NIST)提供了测试向量来验证摘要计算的输出，地址是 http://www.nsrl.nist.gov/testdata/)：

```
NSLog(@"MD5: %@", [@"test string" md5]);
NSLog(@"SHA1: %@", [@"test string" sha1]);
NSLog(@"SHA256: %@", [@"test string" sha256]);

Output:
MD5: 6f8db599de986fab7a21625b7916589c
SHA1: 661295c9cbf9d6b2f6428414504a8deed3020641
SHA256: d5579c46dfcc7f18207013e65b44e4cb4e2c2298f4ac457ba8f82743f31e930b
```

在服务层生成哈希值的过程是类似的，因为 PHP 支持代码清单 6-6 中实现的每一种摘要算法，此外还支持一些这里没有列出的算法。生成哈希值的标准函数是 hash()，它接收待执行的算法以及算法所需要的值。此外，PHP 还提供了一些便捷函数来生成 MD5 与 SHA1 哈希值。下面展示了如何在 PHP 中生成每一种摘要的哈希值：

```
echo "MD5: ".md5("test string")."</br>";
echo "SHA1: ".sha1("test string")."</br>";
echo "SHA256: ".hash("sha256", "test string")."</br>";

Output:
MD5: 6f8db599de986fab7a21625b7916589c
SHA1: 661295c9cbf9d6b2f6428414504a8deed3020641
SHA256: d5579c46dfcc7f18207013e65b44e4cb4e2c2298f4ac457ba8f82743f31e930b
```

上述示例展示了如何生成字符串对象的哈希值，不过也可以很轻松地生成 NSData 对象的哈希值，方式是在 NSData 上创建一个类似的类别。然而，如果有更为复杂的哈希需求或是想在 iOS 应用中比较哈希值，可以考虑创建自定义类，从而优化初始化并通过重写 isEqualTo:来简化哈希值的比较。不过，使用哈希算法可以检测到内容的变化，同时消息认证码与密钥是配对的，并且更加安全。

6.3.2 消息认证码

消息认证码(MAC)是这样一种机制：可以检测到负载是否被修改并验证其真实性。实

现方式是对进来的请求数据(或是预先设定好的请求数据的子集)生成哈希值，然后将哈希值与随负载一同发送的预先计算好的 MAC 进行对比。MAC 类似于之前介绍的哈希函数，但却更加安全，这是因为它们总是与一个密钥配对。参见之前的图 6-2，应用会计算随请求发送的 MAC 值。接下来，将进来的 MAC 与服务层使用相同密钥和数据集计算出的 MAC 进行比较。如果两个 MAC 不同，我们就认为消息被修改了。另一种方式是生成密文的 MAC。虽然结果都是一样的，不过这样可以在执行代价比较高昂的解密处理前判断消息是否已经被修改过。

虽然还有其他的 MAC 算法，不过本节介绍的示例专注于基于哈希的消息认证码(HMAC)，这是因为 HMAC 是 iOS 和大多数服务层平台原生支持的。HMAC 也经常被称作密钥消息认证码，它是由 RFC 2104(http://tools.ietf.org/html/rfc2104)定义的。HMAC 可以使用任何哈希函数，通常会使用 MD5 或 SHA-1，不过其功能主要依赖于底层哈希函数的功能与密钥。虽然 MD5 哈希算法存在一些弱点，SHA-1 的加密性更强一些，不过这并不会对它们在 HMAC 中的使用造成影响。

iOS HMAC 实现支持 MD5、SHA-1、SHA-224、SHA-256、SHA-384 及 SHA-512 摘要算法。HMAC 的输出长度总是与使用的哈希算法的摘要长度一样。与之前介绍的其他哈希函数一样，HMAC 既可以手工生成，也可以使用便捷方法生成，代码清单 6-7 演示了如何使用便捷方法生成 HMAC。

HMAC 示例基于之前创建的 NSString+Hashing 类别构建。首先，该类别还需要方法定义并导入两个库，如代码清单 6-7 所示。

代码清单 6-7　HMAC 哈希新增方法定义(/App/Mobile-Banking/NSString+Hashing.h)

```
...
#import <CommonCrypto/CommonHMAC.h>
#import <CommonCrypto/CommonCryptor.h>

@interface NSString (Hashing)
...
- (NSString*)hmacWithKey:(NSString*)key;

@end
```

导入 CommonCryptor 似乎没有必要，不过需要在 hmacWithKey:的实现中访问密钥长度常量 kCCKeySizeAES256，如代码清单 6-8 所示。

代码清单 6-8　hmacWithKey:实现(/App/Mobile-Banking/NSString+Hashing.m)

```
- (NSString*)hmacWithKey:(NSString*)key {
    // Pointer to UTF8 representations of strings
    const char *ptr = [self UTF8String];
    const char *keyPtr = [key UTF8String];

    // Implemented with SHA256, create appropriate buffer (32 bytes)
```

```
    unsigned char buffer[CC_SHA256_DIGEST_LENGTH];

    // Create hash value
    CCHmac(kCCHmacAlgSHA256, // algorithm
           keyPtr, kCCKeySizeAES256, // key and key length
           ptr, strlen( ptr ), // data to hash and length
           buffer); // output buffer

    // Convert HMAC buffer value to pretty printed NSString
    NSMutableString *output =
        [NSMutableString
        stringWithCapacity:CC_SHA256_DIGEST_LENGTH * 2];
    for(int i = 0; i < CC_SHA256_DIGEST_LENGTH; i++) {
        [output appendFormat:@"%02x",buffer[i]];
    }

    return output;

}
```

代码清单 6-8 包含 hmacWithKey:的实现，它类似于之前介绍过的 hashWithType:方法。不过有两个差别，分别是增加了密钥以及只能使用 SHA-256 摘要算法的限制。密钥与输入数据的创建采用 UTF8 编码，然后传给 CCHmac()函数来创建 HMAC。该函数会使用 HMAC 值来装配缓存，然后 hmacWithKey:以字符串的形式将其返回。

你可能想知道密钥的值来自哪里，是如何选择的。密钥对于密码安全来说是必不可少的，密钥长度的重要性怎么强调也不过分！如果缺乏安全且随机的密钥，应用就会暴露给各种攻击！根据需要保护的数据的不同，你可能还会遭受到可能的诉讼问题，比如数据外泄或是违反美国健康保险流通与责任法案(HIPAA)等。回忆一下图 6-2 中展示的安全概览图：在每一步中，只有加密(E-Key)与 MAC(M-Key)密钥没有随请求一同发送。保护好这些密钥是确保传输数据安全的唯一保障措施。

由于密钥是安全模型的关键所在，因此必须好好选择。密钥的最终长度应该等于加密算法或 HMAC 的密钥长度。短于算法密钥长度的都需要补 NULL，直到等于算法密钥的长度为止，这会削弱缺少的那些字符的随机性。如果必须将用户的输入数据作为密钥的基础，那么可以在输入前追加随机或伪随机的值，这叫做加盐。如果需要再生成这个密钥，那么确保将用于生成最终密钥值的盐也存储起来。最后，使用之前介绍过的算法(如 MD5 或 SHA-1 等)运行几千次哈希计算来得到加盐后的值，然后去除一定的字节数作为最终使用的密钥。

如果应用要求以用户输入作为密钥的基础，那么可以考虑使用 CommonCrypto/CommonKeyDerivation 库的 CCKeyDerivationPBKDF()函数。CCKeyDerivationPBKDF()会返回盐的密钥值、推导算法、推导次数以及用户指定的输出密钥长度。还可以通过 SecRandomCopyBytes()函数生成随机的字节数组。

虽然对于开发者来说，将各种处理细节输出到日志中是种很常见的做法，不过绝不应该将生成的密钥打印到控制台。日志文件可以非常容易地从设备上获取，如果被攻击者发现，那就是非常严重的安全问题了。

共享密钥的缺点在于应用需要规划好密钥版本。不可避免地，你会遇到必须修改共享密钥的情况，这就要求部署应用的新版本，然后更新服务层。然而，iOS 用户并不总是会安装应用更新，也没有任何机制可以强制用户这么做。对于使用旧版本应用的用户来说，该怎么处理呢？需要确保来自旧版本的用户交易也能被正确地解密、验证并被服务处理。密钥版本化可以在不发布应用更新的情况下解决这个问题；不过，要从一开始就将其放到应用的开发当中。

如下代码展示了如何使用代码清单 6-8 中的方法来生成 HMAC：

```
#import "NSString+Hashing.h"
...
// create mac input with to account, from account, amount, and transfer date
NSString *macCandidate = [NSString stringWithFormat:@"%@%@%@%@",
                                @"1234",                    // to account
                                @"4321",                    // from account
                                @"2300.00",                 // amount
                                @"2012-12-25 00:00:00"];    // transfer date
// generate mac
NSString *mac = [macCandidate
                      hmacWithKey:@"065a62448fb75fce3764dcbe68f9908d"];
...
Output:
51e66ca8fd8eb4bbe02fc6421e0dda1deb94f0c9518996a55bc7a4c242f1c8a9
```

该例将资金转移负载的子集作为 MAC；不过，也可以使用整个负载。MAC 的拼接顺序与属性(即想要哈希的值)必须与服务层共享，这样才能确保解密的正确性。如果客户端或服务层在不同的假设下运作，那么消息完整性检查就会失败，什么都不会得到处理。

既然已经传输了MAC，并且服务层也对负载进行了解密，那么服务必须生成与之相伴的MAC。如下代码片段展示了如何使用与客户端相同的拼接输入字符串在PHP中生成HMAC(PHP函数hash_hmac()与iOS中的HMac()函数接收类似的输入，并且可以指定所用的算法、哈希的内容以及要使用的密钥)：

```
echo hash_hmac( "sha256",
                "123443212300.002012-12-25 00:00:00",
                "065a62448fb75fce3764dcbe68f9908d");
Output:
51e66ca8fd8eb4bbe02fc6421e0dda1deb94f0c9518996a55bc7a4c242f1c8a9
```

注意之前两个示例的输出值是一样的。由于这两个值相同，因此服务层可以相信接收到的是未被修改的请求，并且可以安全地进行资金转移。

6.3.3　加密

既然已经生成了消息认证码，客户端现在就可以加密请求负载并将其发送给服务器进行处理。虽然本节会介绍两种特定的对称加密算法——高级加密标准与数据加密标准，以及它们在移动设备上的适用性，不过这里并不会对加密理论进行广泛讨论。然而，好的做法就是相信专家，使用标准化、经过验证的加密算法，而不是自己编写。

使用公钥密码学的非对称加密

本章没有涉及的一个主题就是使用公钥密码学的非对称加密。如果使用公钥密码学，应用会在预先定义好的时间间隔内通过 Web Service 下载公钥，用来对通信进行加解密。这么做的好处在于无须事先共享密钥就可以正常完成加密过程。这种方式提供更高的部署灵活性与长期的可维护性，但却导致加密过程变得更加复杂，设备与 Web Service 都必须维护运行期的密钥列表和各自的标识符，这样才能正确地对加密的消息进行解密。

数据加密标准(Data Encryption Standard，DES)是 1976 年到 1999 年的美国国家加密算法。在 1999 年到 2002 年这几年间，标准升级了，使用称为 Triple-DES 的更为安全的变体。2002 年，NIST 官方选择高级加密标准(Advanced Encryption Standard，AES)取代 DES 与 Triple-DES。不过，如今 DES 实现还是在广泛应用着，只不过是 Triple-DES 的更新实现。虽然 AES 是官方标准，不过 Triple-DES 已经被 NIST 批准为 2030 年前对敏感政府信息的加密标准。

Triple-DES 使用两个 56 位的密钥，第一个密钥用于加密数据，第二个密钥用于加密第一次加密的结果，然后又使用第一个密钥对结果进行加密。执行 3 次独立的加密过程是资源密集型的，因此 Triple-DES 并不适合于移动设备的加密。即便处理器的性能在不断改进，现在也还是有其他更为高效的加密方法。

另外，AES 在设计时考虑的是速度、强度和对资源更为高效的利用，这使得它非常适合于移动设备。只需要一个步骤就能加密数据，并且提供了 256 艾字节(2560 亿吉字节)的上限，这个上限指的是算法对一条消息加密的容量限制。AES 的另一好处是从 iOS 4.3 开始，超过 1024 字节的消息可以被硬件加速(对于 SHA-1 摘要计算来说，超过 4096 字节的消息会应用硬件加速)。考虑到当前设备的存储限制，在 iOS 中使用 AES 比较合适。

手机银行应用使用的是 AES 加密算法。然而，为了能够彻底支持之前尚在使用的标准，本书专门通过一些示例来介绍如何使用 Triple-DES。本节介绍的方法要在 NSData 与 NSString 上创建类别。

Objective-C 中的加解密类似于上一节中介绍的生成密码哈希。加密方法被打包在 CommonCryptor(这是一个 C 函数库)中来对数据进行加解密，并且支持多种加密算法，如表 6-2 所示。该库提供了一个名为 CCCrypt()的便捷函数，它会执行无状态的加解密操作。该函数非常健壮，能够满足大多数加密需求，同时也是本章要介绍的方式。该库还提供了一套函数，可以让开发者对加密过程的每一步进行更加细粒度的控制。

表 6-2　支持的加密算法

加 密 算 法	算 法 常 量
高级加密标准(AES)，128 位	kCCAlgorithmAES128
数据加密标准(DES)	kCCAlgorithmDES
Triple-DES、3 密钥、EDE 配置	kCCAlgorithm3DES
CAST	kCCAlgorithmCAST
RC4 Stream Cipher	kCCAlgorithmRC4
RC2 Block Cipher	kCCAlgorithmRC2
Blowfish Block Cipher	kCCAlgorithmBlowfish

完整的 iOS 加密过程包括创建加密上下文、处理消息、接收剩余的数据以及释放上下文。虽然本章并不会介绍这个过程，不过还有一个额外的步骤可能会在接收最终的输出以及释放上下文之间执行，从而改进某些情况下的性能。这个额外的步骤会重置上下文，这样就可以处理额外的消息，甚至在需要时更新初始化向量。如果需要处理大量的消息，那么可以考虑该方式。

与之前的示例一样，可以在 NSString 与 NSData 上创建一个类别来持有加密方法。代码清单 6-9 与 6-10 展示了每个接口的定义。NSData+Encryption 的接口定义中包含了用于 Base 64 编解码的方法定义。

代码清单 6-9　NSData+Encryption 接口定义(/App/Mobile-Banking/NSData+Encryption.h)

```
@interface NSData (Encryption)

- (NSData*)encryptedWithAESUsingKey:(NSString*)key andIV:(NSData*)iv;
- (NSData*)decryptedWithAESUsingKey:(NSString*)key andIV:(NSData*)iv;

- (NSData*)encryptedWith3DESUsingKey:(NSString*)key andIV:(NSData*)iv;
- (NSData*)decryptedWith3DESUsingKey:(NSString*)key andIV:(NSData*)iv;

+ (NSData*)dataWithBase64EncodedString:(NSString*)string;
- (id)initWithBase64EncodedString:(NSString*)string;

- (NSString*)base64Encoding;
- (NSString*)base64EncodingWithLineLength:(NSUInteger)lineLength;


@end
```

代码清单 6-10　NSString+Encryption 接口定义(/App/Mobile-Banking/NSString+Encryption.h)

```
@interface NSString (Encryption)

- (NSString*)encryptedWithAESUsingKey:(NSString*)key andIV:(NSData*)iv;
- (NSString*)decryptedWithAESUsingKey:(NSString*)key andIV:(NSData*)iv;
```

```
- (NSString*)encryptedWith3DESUsingKey:(NSString*)key andIV:(NSData*)iv;
- (NSString*)decryptedWith3DESUsingKey:(NSString*)key andIV:(NSData*)iv;

@end
```

定义好接口后，下面来看看 NSString+Encryption 的实现。代码清单 6-11 展示了每个字符串编解码方法的实现。

代码清单 6-11　NSString 编解码方法的实现(/App/Mobile-Banking/NSString+Encryption.m)

```
#import "NSData+Encryption.h"
#import "NSData+Base64.h"
...
- (NSString*)encryptedWithAESUsingKey:(NSString*)key andIV:(NSData*)iv {
    NSData *encrypted =
    [[self dataUsingEncoding:NSUTF8StringEncoding]
      encryptedWithAESUsingKey:key
                         andIV:iv];
    NSString *encryptedString = [encrypted base64Encoding];

    return encryptedString;
}

- (NSString*)encryptedWith3DESUsingKey:(NSString*)key andIV:(NSData*)iv {
    NSData *encrypted =
    [[self dataUsingEncoding:NSUTF8StringEncoding]
      encryptedWith3DESUsingKey:key
                          andIV:iv];

    NSString *encryptedString = [encrypted base64Encoding];

    return encryptedString;
}

- (NSString*)decryptedWithAESUsingKey:(NSString*)key andIV:(NSData*)iv {
    NSData *decrypted =
    [[NSData dataWithBase64EncodedString:self]
      decryptedWithAESUsingKey:key
                         andIV:iv];

    NSString *decryptedString =
    [[NSString alloc] initWithData:decrypted
                          encoding:NSUTF8StringEncoding];

    return decryptedString;
}

- (NSString*)decryptedWith3DESUsingKey:(NSString*)key andIV:(NSData*)iv {
    NSData *decrypted =
```

```
    [[NSData dataWithBase64EncodedString:self]
      decryptedWith3DESUsingKey:key
                        andIV:iv];

    NSString *decryptedString =
    [[NSString alloc] initWithData:decrypted
                        encoding:NSUTF8StringEncoding];

    return decryptedString;
}
```

每个 NSString 方法都是类似的；主要的差别在于 Base 64 编码的方向以及在返回前如何对结果进行编码。现在是时候在 NSData+Encryption 类别中实现核心编码逻辑了，如代码清单 6-12 与 6-13 所示。

代码清单 6-12　AES 加密(/App/Mobile-Banking/NSData+Encryption.m)

```
#import <CommonCrypto/CommonCryptor.h>
#import "NSData+Base64.h"
...
- (NSData*)encryptedWithAESUsingKey:(NSString*)key andIV:(NSData*)iv {

    NSData *keyData = [key dataUsingEncoding:NSUTF8StringEncoding];

    size_t dataMoved;

    NSMutableData *encryptedData =
        [NSMutableData dataWithLength:self.length + kCCBlockSizeAES128];

    CCCryptorStatus status = CCCrypt(kCCEncrypt,
                                     kCCAlgorithmAES128,
                                     kCCOptionPKCS7Padding, //CBC Padding
                                     keyData.bytes,
                                     keyData.length,
                                     iv.bytes,
                                     self.bytes,
                                     self.length,
                                     encryptedData.mutableBytes, //data out
                                     encryptedData.length,
                                     &dataMoved); // total data moved
    if(status == kCCSuccess) {
        encryptedData.length = dataMoved;
        return encryptedData;
    }

    return nil;

}
```

代码清单 6-13　Triple-DES 加密(/App/Mobile-Banking/NSData+Encryption.m)

```
- (NSData*)encryptedWith3DESUsingKey:(NSString*)key andIV:(NSData*)iv {

    NSData *keyData = [key dataUsingEncoding:NSUTF8StringEncoding];

    size_t dataMoved;
    NSMutableData *encryptedData =
        [NSMutableData dataWithLength:self.length + kCCBlockSize3DES];

    CCCryptorStatus result = CCCrypt(kCCEncrypt,
                                     kCCAlgorithm3DES,
                                     kCCOptionPKCS7Padding, // CBC Padding
                                     keyData.bytes,
                                     keyData.length,
                                     iv.bytes,
                                     self.bytes,
                                     self.length,
                                     encryptedData.mutableBytes,// data out
                                     encryptedData.length,
                                     &dataMoved); // total data moved

    if(result == kCCSuccess) {
        encryptedData.length = dataMoved;
        return encryptedData;
    }

    return nil;
}
```

代码清单 6-12 与 6-13 使用 CCCrypt()函数分别实现了 AES 与 Triple-DES 加密。CCCrypt()函数非常直接明了，不过需要为 encryptedData 指针分配合适的空间。CCCrypt()函数的输出长度绝对不会超过输入加上一个额外的块的长度。块大小是根据实现的加密算法确定的。

在示例中，如果没有获得成功的结果，那么方法会返回 nil。根据需求的不同，需要实现额外的错误处理。CCCrypt()函数有 3 种可能的错误结果：kCCBufferTooSmall、kCCAlignmentError 与 kCCDecodeError。kCCBufferTooSmall 与 kCCDecodeError 最常见。kCCBufferTooSmall 表示输出缓存(在该例中就是 encryptedData)的大小不够。kCCDecodeError 只能在解密操作中获取到，最有可能与无效的密钥相关。

完成加密方法后，代码清单 6-14 展示了应用该如何进行加密。代码清单 6-14 还详细介绍了 MAC 的生成过程。

代码清单 6-14　AES 加密(/App/Mobile-Banking/FundsTransferOperation.m)

```
...
    // build our transfer data
    NSDictionary *transfer = [NSDictionary dictionaryWithObjectsAndKeys:
```

```
                                _toAccount, @"toAccount",
                                _fromAccount, @"fromAccount",
                                date, @"transferDate",
                                _transferNotes, @"transferNotes",
                                amount, @"amount", nil];

    // create the json representation of our transfer data using ios5 API
    NSError *error = nil;
    NSData *transferData = [NSJSONSerialization dataWithJSONObject:transfer
                                                           options:0 error:&error];
    NSString *transferString =
        [[NSString alloc] initWithData:transferData
                              encoding:NSUTF8StringEncoding];

    // generate our initialization vector
    NSData *iv = [Utils
                  blockInitializationVectorOfLength:kCCBlockSizeAES128];

    // because we generate random bytes,
    // it may not be proper UTF8 encoding.
    // because of this, we can't just init a string with data. instead,
    // we encode it for transmission. the IV is then decoded on the service
    // and used in the decryption process
    NSString *ivString = [iv base64Encoding];

    // encrypt our transfer data using AES and a randomly generated
    // IV (this IV needs to be what we send to the service)
    NSString *encryptedString =
        [transferString encryptedWithAESUsingKey:kAESEncryptionKey
                                           andIV:iv];

    // calculate our message authentication code
    NSString *macCandidate = [NSString stringWithFormat:@"%@%@%@%@",
                              _toAccount, // to account
                              _fromAccount, // from account
                              amount, // amount
                              _transferDate]; // transfer date
    NSString *mac = [macCandidate hmacWithKey:kMACKey];

    // construct our payload
    NSMutableDictionary *payload = [NSMutableDictionary
                                    dictionaryWithObjectsAndKeys:
                                    ivString, @"iv",
                                    mac, @"mac",
                                    encryptedString, @"payload", nil];
...
```

创建好资金转移数据结构并生成初始化向量后，需要加密转移指令并计算出 MAC。虽然 MAC 的生成已经在上一节中介绍过了，不过代码清单 6-14 还是加入了这部分代码以

提供额外的上下文。

到现在为止，作为输入的初始化向量已经有了。在代码清单 6-14 中，你要负责在加密前生成初始化向量。代码清单 6-15 展示了如何使用 Security Framework(下一节将会介绍)提供的 SecRandomCopyBytes()函数来创建加密安全的随机字节数组。

代码清单 6-15　使用 SecRandomCopyBytes()函数创建初始化向量

```
+ (NSData*)blockInitializationVectorOfLength:(size_t)ivLength {
    // default to AES block size
    if(ivLength == 0) {
        ivLength = kCCBlockSizeAES128;
    }

    NSMutableData *iv = [NSMutableData dataWithLength:ivLength];

    int ivResult = SecRandomCopyBytes(kSecRandomDefault,
                                      ivLength,
                                      iv.mutableBytes);

    if(ivResult == noErr) {
        return iv;
    }

    return nil;
}
```

如果没有指定向量的长度，那么该方法会使用 AES 块长度作为默认值。在生成向量后，确保没有收到错误并返回结果。唯一的错误是-1，表示失败。要想使用 SecRandomCopyBytes()函数，就必须在项目中引入 Security Framework。

在从一个系统向另一个系统传输加密数据以进行解密时，你需要知道各种参数的用途是什么。对于小于特定算法定义的块长度的初始化向量来说，CCCrypt()函数会以 NULL 进行补齐。根据接收系统处理补齐方式的不同，这可能会造成一些集成问题。

代码清单 6-16 展示了如何使用 PHP 对代码清单 6-14 中生成的加密负载进行解密和验证 MAC。对于 AES 和 Triple-DES 来说，解密过程是一致的，只不过传递给 mcrypt_decrypt()函数调用的算法和密钥不同。

代码清单 6-16　使用 PHP 进行 AES 解密(/Service/index.php)

```
// retrieve the request payload
$postData = json_decode(@file_get_contents('php://input'));
$inboundMac = $postData->mac;
$iv = $postData->iv;
$payload = $postData->payload;

// decrypt the payload
$decrypted = mcrypt_decrypt(MCRYPT_RIJNDAEL_128,
```

```
                                $AES_KEY,
                                base64_decode($payload),
                                MCRYPT_MODE_CBC,
                                $iv);
$decLength = strlen($decrypted);
$padding = ord($decrypted[$decLength-1]);
$decrypted = substr($decrypted, 0, -$padding);

// decode decrypted payload, split into components
$decryptedPayloadJSON = json_decode($decryptedPayload);
$toAccount = $decryptedPayloadJSON->toAccount;
$fromAccount = $decryptedPayloadJSON->fromAccount;
$amount = $decryptedPayloadJSON->amount;
$transferDate = $decryptedPayloadJSON->transferDate;
$transferNotes = $decryptedPayloadJSON->transferNotes;

// grab toAccount, fromAccount, amount, transferDate (in that order)
// and create message auth code (hmac)
$macCandidate = $toAccount.$fromAccount.$amount.$transferDate;
$derivedMac = hash_hmac("sha256", $macCandidate, $HMAC_KEY);

// validate inbound hmac matches your derived value
if($inboundMac != $derivedMac) {
    sendAPIResponse
       (400,
        json_encode(buildErrorResponse("Message Integrity Error"))
       );
       return;
}

// here you would perform your actual transfer and
// validate it was successful prior to issuing 200

sendAPIResponse(200);
return;
```

如下代码片段展示了为了解码 Triple-DES 加密的 blob 对算法所做的变更：

```
...
// decrypt the payload
$decrypted = mcrypt_decrypt(MCRYPT_3DES,
                                $DES_KEY,
                                base64_decode($payload),
                                MCRYPT_MODE_CBC,
                                $iv);
$decLength = strlen($decrypted);
$padding = ord($decrypted[$decLength-1]);
$decrypted = substr($decrypted, 0, -$padding);

// decode decrypted payload, split into components
...
```

加密完毕后，解密、响应解释以及负载解密相对来说就比较容易了。解密之所以很容易，是因为只有一行代码，代码清单 6-17 和 6-18 类似于代码清单 6-12 与 6-13。CCCrypt() 函数可用于加密与解密操作，意图是通过第一个参数指定的。

代码清单 6-17 与 6-18 中的方法看起来很熟悉；它们与相应的加密过程几乎是一样的。唯一的差别在于告诉 CCCrypt()函数执行解密操作而非加密操作。

代码清单 6-17　AES 解密(/App/Mobile-Banking/NSData+Encryption.m)

```
- (NSData*)decryptedWithAESUsingKey:(NSString*)key andIV:(NSData*)iv {

    NSData *keyData = [key dataUsingEncoding:NSUTF8StringEncoding];

    size_t dataMoved;
    NSMutableData * decryptedData =
        [NSMutableData dataWithLength:self.length + kCCBlockSizeAES128];

    CCCryptorStatus result = CCCrypt(kCCDecrypt,
                                     kCCAlgorithmAES128,
                                     kCCOptionPKCS7Padding, // CBC Padding
                                     keyData.bytes,
                                     keyData.length,
                                     iv.bytes,
                                     self.bytes,
                                     self.length,
                                     decryptedData.mutableBytes, //data out
                                     decryptedData.length,
                                     &dataMoved); // total data moved

    if(result == kCCSuccess) {
        decryptedData.length = dataMoved;
        return decryptedData;
    }

    return nil;

}
```

代码清单 6-18　Triple-DES 解密(/App/Mobile-Banking/NSData+Encryption.m)

```
- (NSData*)decryptedWith3DESUsingKey:(NSString*)key andIV:(NSData*)iv {

    NSData *keyData = [key dataUsingEncoding:NSUTF8StringEncoding];

    size_t dataMoved;
    NSMutableData *decryptedData =
        [NSMutableData dataWithLength:self.length + kCCBlockSize3DES];

    CCCryptorStatus result = CCCrypt(kCCDecrypt,
                                     kCCAlgorithm3DES,
```

```
                                        kCCOptionPKCS7Padding, // CBC Padding
                                        keyData.bytes,
                                        keyData.length,
                                        iv.bytes,
                                        self.bytes,
                                        self.length,
                                        decryptedData.mutableBytes,//data out
                                        decryptedData.length,
                                        &dataMoved); // total data moved

    if(result == kCCSuccess) {
        decryptedData.length = dataMoved;
        return decryptedData;
    }

    return nil;

}
```

应用实现了账户列表功能以说明客户端解码。当用户认证成功后，应用会立刻发出请求来检索用户的账户。账户列表包含了名字、号码及余额，并以加密的形式传给应用。可以对账户列表进行解密，验证消息完整性，创建账户对象并在视图层中显示出来。

代码清单 6-19 展示了如何创建服务层响应，这包括使用 AES 加密负载并计算 MAC。根据之前对加密的介绍，你现在应该对代码清单 6-19 的结构感到很熟悉。

代码清单 6-19　使用 PHP 生成负载并加密(/Service/index.php)

```
// create array of bank accounts for the user
$accounts = array();

// fill our accounts array with data
...

// generate the IV
$iv = generateInitVectorOfLength(16);

// create MAC from accounts data
$mac = hash_hmac("sha256", json_encode($accounts), $hmacKey);

// encrypt our payload
$encryptedPayload = mcrypt_encrypt(MCRYPT_RIJNDAEL_128,
$AES_Key,
addEncryptionPadding(json_encode($accounts)),
MCRYPT_MODE_CBC,
$iv);
// generate service response
$response['iv'] = $iv;
$response['mac'] = $mac;
$response['payload'] = base64_encode($encryptedPayload);
...
```

虽然代码清单 6-19 介绍的是 AES 加密，但其实 Triple-DES 加密也是类似的。你只需将初始化向量的长度由 16 改为 8，将加密算法由 MCRYPT_RIJNDAEL_128 改为 MCRYPT_3DES，并修改密钥($AES_Key)即可。

代码清单 6-20 详细介绍了如何在应用中解释和解密代码清单 6-19 生成的响应。

代码清单 6-20　处理 AES 加密响应(/App/Mobile-Banking/GetAccountsOperation.m)

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection {

    // unpack service response
    NSError *error = nil;
    NSDictionary *response =
        [NSJSONSerialization JSONObjectWithData:self.responseData
                                        options:0
                                          error:&error];
        // decrypt the payload
        NSString *inboundMAC = [response objectForKey:@"mac"];
        NSData *ivData =
            [[response objectForKey:@"iv"]
               dataUsingEncoding:NSUTF8StringEncoding];

        NSString *encryptedResponse = [response objectForKey:@"payload"];
        NSString *decryptedResponse =
            [encryptedResponse decryptedWithAESUsingKey:kAESEncryptionKey
                                                  andIV:ivData];

        if(decryptedResponse != nil) {
            // create JSON array of account info
            NSError *accountError = nil;
            NSArray *accounts =
                [NSJSONSerialization JSONObjectWithData:
                   decryptedResponse dataUsingEncoding:NSUTF8StringEncoding]
                                              options:0
                                                error:&accountError];

            // validate the MAC
            NSString *generatedMAC = [decryptedResponse hmacWithKey:kMACKey];
            if([inboundMAC isEqualToString:generatedMAC]) {

                // validation passed, create accounts
                for(NSDictionary *accountData in accounts) {
                Account *account = [[Account alloc] initWithData:accountData];
                [[Model sharedModel].accounts addObject:account];
                }

            } else {
      // post error notification
        }
    } else {
```

```
    // post error / unable to decrypt notification
        }
        ...
}
```

本节介绍了如何使用原生 Objective-C 库对数据进行加解密。虽然数据加密在用户的安全中扮演着重要角色，不过密码还是会受到联邦出口管控。这些管控在 App Store 提交过程中会遇到。图 6-3 展示了 App Store 的加密确认界面。对于企业部署的应用来说，应该咨询法律部门。

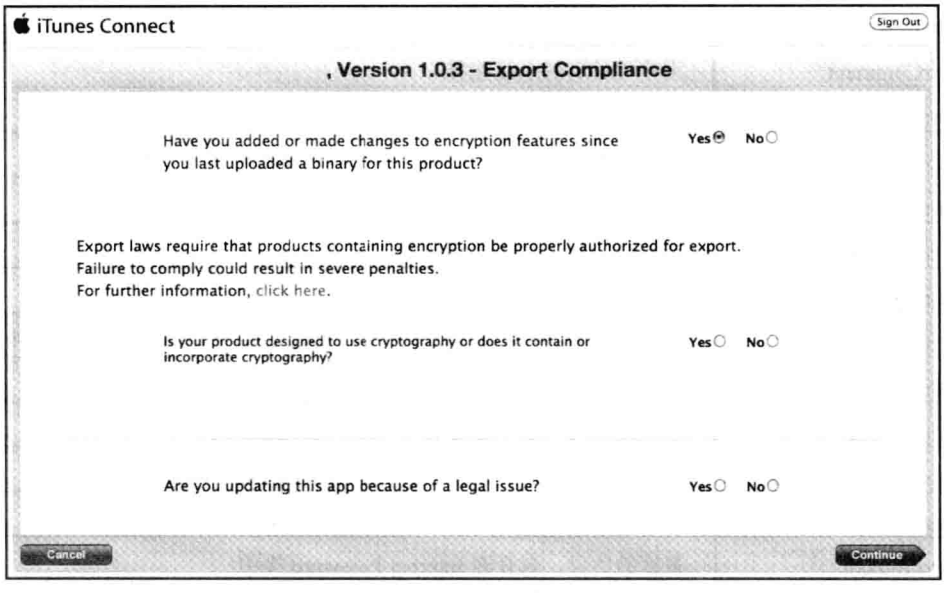


图 6-3

6.4　在设备上安全地存储认证信息

既然应用能安全地与服务层进行通信，那就需要在设备上安全地存储信息。Apple 提供了 Keychain Services API(作为 Security Framework 的一部分)来完成这项工作。Keychain 是一种在设备上安全存储少量数据的机制，比如密码、密钥、证书和身份信息等。Keychain 并不适合于通用目的的加密和数据存储，而是用来存储需要保护的信息，比如密码与私钥就会以加密的形式存储起来。诸如证书(不需要这种级别的保护)等就不会加密存储。本节将会介绍 Keychain 的概念，第 11 章“应用间通信”将会更加深入地介绍 Keychain 的实现。

在 iOS 中，每个应用都可以访问它所创建的 Keychain 条目而无须请求许可。这与传统的 Mac 开发不同，后者的应用可以访问用户同意的任何 Keychain 条目。从技术上来说，Keychain 数据存储在应用沙箱之外，这样就可以通过应用委托事件来持久化数据了。iOS Keychain 的权限依赖于用于签名应用的配置文件。当应用在其版本的生命周期中不断演进时，你需要一直使用相同的配置文件。

应用的 Keychain 可以包含任意数量的条目，每个条目都包含待存储的数据及属性。每个 Keychain 条目的属性都依赖于在存储过程中选择的条目类。条目类之间存在很多通用的条目属性。请参考 Apple 的文档以了解完整列表，表 6-3 列出了应用可以设置的属性。

表 6-3　可编辑的 Keychain 条目属性

条目属性	说明
kSecAttrAccessible	表示条目何时可以被访问，下一章将会对此做进一步介绍
kSecAttrAccessGroup	表示条目属于哪个访问组，下一章将会对此做进一步介绍
kSecAttrDescription	用户可见的字符串说明条目
kSecAttrComment	用户可编辑的条目注释
kSecAttrCreator	条目的创建者，表示为由 4 个字符代码构成的无符号整数
kSecAttrType	条目的类型，表示为由 4 个字符代码构成的无符号整数
kSecAttrLabel	用户可见的条目标签
kSecAttrIsInvisible	布尔值，表示是否显示条目
kSecAttrIsNegative	表示条目是否存在有效的密码
kSecAttrAccount	与该条目相关的账户名，包含在 Generic 与 Internet Password 类中
kSecAttrService	与该条目相关的服务名，包含在 Generic Password 类中
kSecAttrGeneric	用户定义的属性，包含在 Generic Password 类中
kSecAttrSecurityDomain	条目的 Internet 安全域，包含在 Generic Password 类中
kSecAttrServer	条目的服务域或 IP 地址，包含在 Internet Password 类中
kSecAttrProtocol	条目的协议，包含在 Internet Password 类中
kSecAttrAuthenticationType	条目的认证模式，包含在 Internet Password 类中
kSecAttrPort	Internet 端口号，包含在 Internet Password 类中
kSecAttrPath	路径，通常是 URL 路径，包含在 Internet Password 类中
kSecAttrApplicationLabel	条目的标签，用于以编程的方式查找密钥，包含在 Key 类中
kSecAttrIsPermanent	布尔值，表示密钥是否已经被永久存储，包含在 Key 类中
kSecAttrKeyType	与密钥关联的算法，包含在 Key 类中
kSecAttrKeySizeInBits	密钥中总的位数，包含在 Key 类中
kSecAttrEffectiveKeySize	密钥中有效的位数，包含在 Key 类中
kSecAttrCanEncrypt	布尔值，表示密钥是否可用于加密，包含在 Key 类中
kSecAttrCanDecrypt	布尔值，表示密钥是否可用于解密，包含在 Key 类中
kSecAttrCanDerive	布尔值，表示密钥是否可以导出另一个密钥，包含在 Key 类中
kSecAttrCanSign	布尔值，表示密钥是否可以创建数字签名，包含在 Key 类中
kSecAttrCanVerify	布尔值，表示密钥是否可以验证数字签名，包含在 Key 类中
kSecAttrCanWrap	布尔值，表示密钥是否可以封装另一个密钥，包含在 Key 类中
kSecAttrCanUnwrap	布尔值，表示密钥是否可以展开另一个密钥，包含在 Key 类中

在创建过程中，有两个重要的属性需要注意(对于所有类都是一样的)，它们是 kSecAttrAccessible 与 kSecAttrAccessGroup。可以通过 kSecAttrAccessible 判断应用何时能够访问 Keychain 条目。你应该使用让应用能够满足其目的的最严格的选项。表 6-4 列出了 kSecAttrAccessible 属性的所有可能值。最低限度上，你应该考虑将 kSecAttrAccessible 设为以 ThisDeviceOnly 结尾的值，这样就限制 Keychain 条目不能传给新的设备。kSecAttrAccessGroup 表示 Keychain 条目属于哪个访问组。应用可以属于多个访问组，你在第 11 章定义的 Entitlements.plist 文件中将会看到这一点。多个访问组可以进一步划分 Keychain 数据。访问组还可以用于在应用间共享数据。第 11 章的示例将会详细介绍这一点，包括 Entitlements.plist 文件。

表 6-4　kSecAttrAccessible 属性的可能值

可能的取值	说　明
kSecAttrAccessibleWhenUnlocked	当设备解锁时就可以访问条目数据。建议当应用处于前台并且需要条目时使用该值
kSecAttrAccessibleAfterFirstUnlock	当设备重启的第一次解锁后可以访问条目数据
kSecAttrAccessibleAlways	条目数据总是可以访问的，不推荐开发者使用该值，它应由系统使用
kSecAttrAccessibleWhenUnlockedThisDeviceOnly	类似于 kSecAttrAccessibleWhenUnlocked，只不过条目数据无法迁移到新的设备
kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly	类似于 kSecAttrAccessibleAfterFirstUnlock，只不过条目数据无法迁移到新的设备
kSecAttrAccessibleAlwaysThisDeviceOnly	条目数据总是可以访问的，类似于 kSecAttrAccessibleAlways，但并不推荐使用该值，因为它应由系统使用。具有该值的条目无法迁移到新的设备

由于实现简单并且存储灵活，对于刚开始接触 Keychain 概念的开发者来说，Generic Password 是最常见的起点。Generic Password 类是安全存储非 Internet 密码的理想位置，比如本章的服务层使用的认证令牌等。Generic Password 类也可以用于存储检测应用之前是否安装过的指示器。

与 Keychain Services 的交互需要理解如何组织 Keychain 搜索。最重要的步骤是为 kSecClass 属性指定恰当的条目类值。这是在 Keychain 条目创建时设定的，然后划分搜索。kSecAttrAccessGroup 对于搜索结构也是非常重要的。如前所述，应用可以包含多个访问组。如果应用指定错误的访问组进行搜索，那就无法定位到你要搜索的 Keychain 条目。

在 Keychain 中创建条目时，最佳实践是首先判断条目是否已经存在，然后根据结果进行添加或更新。其余动作(检索、更新与删除)都接收一个查询参数，它是 CFDictionaryRef 的一个实例。动作会在与查询匹配的每个 Keychain 条目上执行。查询参数可以是任意数量

的条目属性(表 6-3 中介绍的属性的一个子集)与表 6-5 中定义的搜索属性的组合。

表 6-5　预定义的 Keychain 搜索属性

搜索属性	说　明
kSecMatchPolicy	限制证书与身份必须遵循该策略，值是 SecPolicyRef 对象
kSecMatchIssuers	限制证书与身份，要求身份对应的证书链包含一个或多个 Issuers，这个 Issuers 是由 X.509 名字构成的 CFArray 中的元素
kSecMatchEmailAddressIfPresent	限制证书与身份，要求包含指定的地址或不包含地址
kSecMatchSubjectContains	限制证书与身份，要求主题中包含指定的 CFStringRef
kSecMatchCaseInsensitive	指定搜索的匹配要区分大小写
kSecMatchTrustedOnly	布尔值，限制可验证的证书必须是可信任的锚点。如果为假，那么受信与非受信的证书都会返回
kSecMatchValidOnDate	限制密钥、证书与身份要在指定的日期前为有效的，传递 kCFNull 可以使用当前日期
kSecMatchLimit	指定返回的结果数量。默认值是 kSecMatchLimitOne，另一个预定义的常量为 kSecMatchLimitAll
kSecMatchLimitOne	限制结果为第一个匹配的条目
kSecMatchLimitAll	指定返回所有匹配的结果

说明：

如前所述，第 11 章将会介绍如何将 Keychain 集成到应用中，包括如何在多个应用间共享数据。

6.5　小结

确保网络通信的安全性是非常重要的，形式也多种多样，比如验证用户与正确的服务器通信，或是在授权用户访问系统前对其进行认证。你还需要选择使用 AES 或 Triple-DES 等加密算法对网络流量的全部或部分进行加密。为了确保请求在传输过程中没有被他人操纵，你需要 MAC 或密码哈希，这需要与服务层设定好传输规划。在安全地传输数据后，可以选择使用设备 Keychain 将其安全地存储起来。

虽然 Apple 提供了这些安全库，不过作为开发者还是需要对如何保护应用中有价值的个人信息有所了解。这十分必要，通常是合约或法律要求的，特别是在处理来自金融机构、健康组织或政府部门的数据时。对用户透明是最重要的，你应该清楚所要访问的信息并根据用户的意愿进行传输。只存储应用所需的绝对必要的信息以满足这个目标，并且总是以恰当且安全的方式来传输这些信息。

下一章将会介绍通过 HTTP 缓存、压缩与管道实现请求优化的各种模式。

第 7 章

优化请求性能

本章内容

- 理解网络带宽与延迟
- 通过压缩减少请求带宽
- 通过管道降低请求延迟
- 使用缓存最小化请求带宽

本章代码下载

可以在 www.wrox.com 的 Download Code 选项卡中下载本章代码，网址是 www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html。代码位于第 7 章的压缩包中，并且根据章节名字单独命名。

正如之前的章节所述，iOS 设备提供的基础网络功能是强大且易用的。iOS 拥有功能完善的 API，支持众多业界标准协议；不过，纵使你开发出优秀的应用，但网络通信的质量还是要取决于设备与外界的连接。

本章将会介绍度量网络性能的各种维度，可以通过这些知识来改进应用的网络通信。本章还将介绍一些最佳实践，如减少应用消耗的网络带宽、改进响应性，以及延长运行着应用的设备的电池寿命。

7.1 度量网络性能

本节将会从高层次概览用于描述网络性能的各种度量指标。虽然有很多度量指标可用于描述无线网络的性能，不过本节将会关注其中最为重要的 3 个指标：带宽、延迟与电量功耗。

7.1.1 网络带宽

用于描述无线网络性能的最常见度量指标就是带宽。在数字无线通信中，网络带宽可以描述为两个端点之间的通信通道每秒钟可以传输的位数。现代无线网络所能提供的理论带宽是很高的，不过请记住，运营商与网络设备提供商引用的带宽数字常常是该项技术的理论最大值，网络设备使用的实际带宽可能与这个最大值之间存在很大的偏差。图 7-1 展示了以对数尺度描述的当前无线技术的理论能力。

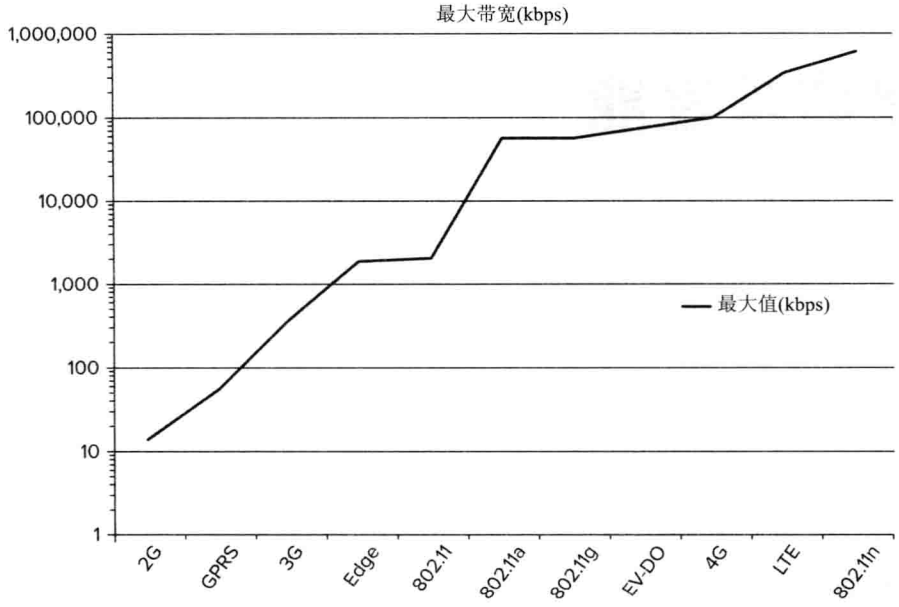


图 7-1

有很多原因会导致设备实际使用的带宽与理论最大值有明显的差别。首先，无线网络带宽是共享的。虽然无线技术在鼓吹自己有很高的带宽，不过通道可能会被很多无线设备共享，每个设备都会为带宽展开竞争。因此，每个设备的带宽消耗与传输都会对设备的带宽产生影响。图 7-2 展示了应用发出的请求在到达目的服务器之前要经过的路由。在路由的每一步中，请求都会与其他数据包展开带宽竞争。

考虑这样一种情况：你在堪萨斯州，你的应用在使用 LTE 基站，不过基站有个小的回程线路，连接到了 Internet。应用到基站的数据传输率应该很不错，但回程线路却限制了可用的带宽。任何网络连接的最大速率取决于通信路径上的最慢链路。虽然有充足的回程带宽，但运营商的网络设备却对通过网络的每个数据包应用了服务质量(Quality of Service，QoS)优先级值。设备通常将简单数据包的优先级排在语音数据包之后；因此，网络请求必须在对时效性敏感的语音数据之后等待。

诸如到基站的距离、大气状况以及网络干扰等其他因素也会对设备使用的带宽产生影响。这些因素都是无法避免的，应用只会使用到可用带宽的一小部分，并且要充分利用这一小部分带宽。本章后面的 7.2.1 节“减少请求带宽”将会对此给出一些建议。

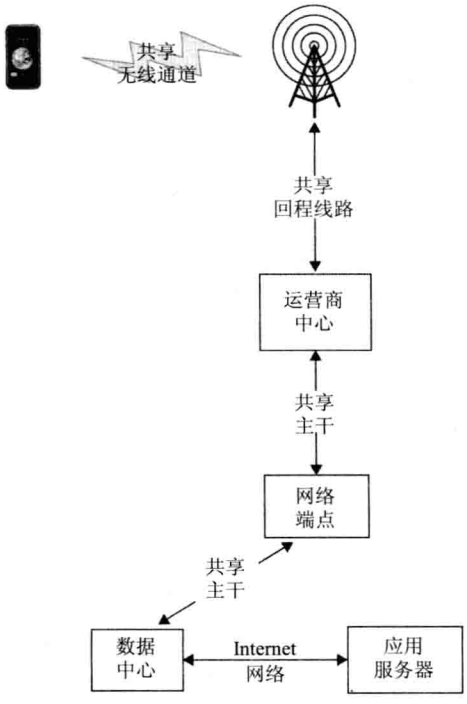


图 7-2

7.1.2　网络延迟

度量网络性能的第 2 个因素是网络延迟，指的是网络包在两个端点间一次往返所需的时间。无线运营商很少会提到网络的延迟数据，不过延迟却会对应用的实际性能造成很大影响。与带宽一样，有很多因素会影响到应用遭受的延迟情况。主要因素就是用于将设备连接到外界的无线网络技术本身的延迟。图 7-3 展示了针对公共网络标准的最佳延迟数据。

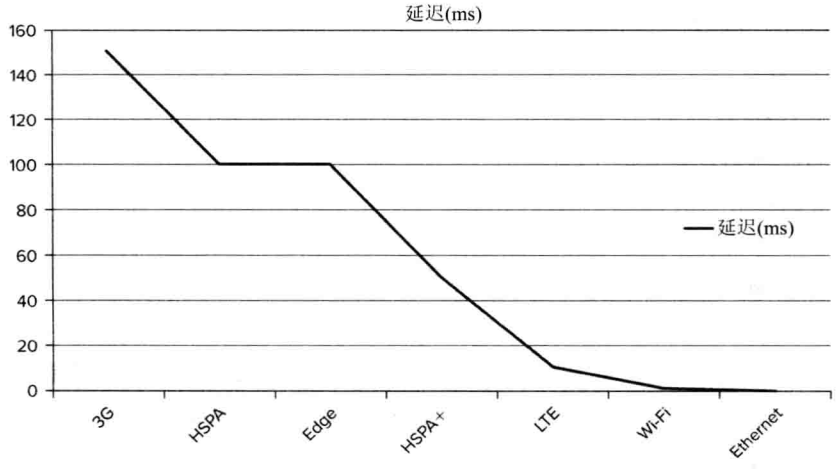


图 7-3

延迟也取决于数据包在设备与目标服务器之间传输时选择的特定路由。图 7-2 展示了一个示例路由。还有其他很多因素会对网络延迟造成显著影响，如下所示：

- 共享相同基站的其他设备会导致数据包出现延迟
- 网络数据包的 QoS 优先级会导致数据包等待其他优先级更高的数据包
- 回程通道会增加应用的延迟
- 服务器响应时间会增加数据包的延迟

在 iOS 设备上可以通过很多工具来度量设备与网络主机之间的延迟情况，比如 SpeedTest.net 应用。如果使用它或是类似的工具，那么你会发现每次在同样的网络上发出请求时，延迟时间都会出现较大范围的变化，甚至那些连续发出的请求也是如此。所有这些因素都导致我们很难一致且精确地进行度量。

延迟对应用造成的影响主要就是响应性。根据网络的情况，用户等待某个动作的响应时间从几毫秒到几秒变化不等。这意味着你需要在开发过程中尽早且频繁地在各种网络上对应用进行测试。第 9 章“测试与操纵网络流量”将会介绍如何使用 Max OS X 的 Network Link Conditioner 来模拟 iOS 模拟器遇到的各种网络情况。延迟的影响与应用产生的请求大小成反比(一个例外就是当请求小于 1500 字节时，网络延迟的影响将会降为 0 字节)。如果应用发出的请求的负载很小，那么延迟就会造成很大的影响。然而，如果应用处理的是流媒体内容或是执行了需要大量负载的活动，那么延迟相对于总的带宽来说就微不足道了。

本章后面的 7.2.2 节“降低请求延迟”将会介绍可以采用哪些度量手段来降低延迟时间。

7.1.3 设备电量

iOS 设备的运行通常使用的都是电池，应用采取的每个动作都会消耗电量。iOS 设备上电量的使用大户主要有：

- 屏幕显示与背光
- 位置服务
- Wi-Fi 无线电
- WWAN(蜂窝)无线电
- 图形处理器
- CPU
- 蓝牙无线电
- 声音处理器

除了屏幕显示之外，应用能够控制上面的大多数。在优化程序以实现更好的网络性能时，你还会得到另一个好处，那就是改进应用的电量消耗。如果应用将传输的数据量降到最小、优化对现有的 TCP 连接的使用、避免不必要的请求，那就可以减少设备无线电的开启时间。

减少应用的电池使用时间的另一种方式是使用智能加密。最近，Web 应用中的一项最

佳实践是对所有数据进行加密，这对于那些运行在台式机或笔记本(通常都会接电源)电脑上的应用来说是一条好建议。不过，对于电量有限的移动设备来说，你应该考虑要对哪些请求进行加密。在传输时，图片以及其他公开的内容就不需要加密了。使用加密会增加 CPU 的负载，并且会激活通常情况下处于休眠状态的硬件资源。

7.2　优化网络操作

承认网络性能存在问题是解决问题的第一步。如果不采取措施解决网络性能问题，那么用户将会遭遇到非常不好的体验。本节将会介绍如何通过压缩 HTTP 体的内容来减少应用使用的网络带宽、可以采用哪些度量手段来最小化高延迟连接产生的影响，同时还会介绍如何通过响应缓存而不去使用网络。

7.2.1　减少请求带宽

在计算机的早期时代，计算机之间的互联仅仅为 300 位/秒是很常见的事情。在这个低速率下，传输的每一位都需要经过深思熟虑，计算机科学家们设计了紧凑和稀疏的协议，能够保持带宽，但却难以实现、验证和支持。

随着现代网络速率增加到了每秒上百万位，每一位都变得不那么相关了，这些连接使用的协议也变得更加详尽了。这些详尽的协议更易于维护和管理。诸如 SOAP 之类的协议会消耗掉大量的带宽，但却提供了足够的信息，使得每条消息几乎都是自描述的。

随着 GSM 与 CDMA 等共享无线网络的出现，工程师们不得不重新考虑他们为应用选择的协议所使用的带宽，并通过一些手段来减少带宽的使用量，而且还不会牺牲应用的可靠性与可维护性。有一些手段可以减少应用所消耗的带宽量：

- 使用高效的数据交换格式——为客户端与服务器之间传输的数据选择高效的编码。第 4 章“生成与解析负载”介绍了选择正确的负载格式的标准。
- 在可能的情况下使用预先压缩的数据——使用专用算法对诸如音频、视频和图像进行压缩或按比例缩放以适应通道与设备。
- 压缩每一个请求与响应负载——压缩文本负载以减少带宽，同时又不太影响服务器与客户端代码。

实际上，可以通过压缩服务器响应或是客户端请求为非媒体负载开启负载压缩。压缩响应的方式与压缩请求完全不同。接下来将会介绍针对响应与请求的压缩方法。

为请求与响应体使用 JSON 与 XML

JSON 与 XML 是用于请求与响应体的常见数据编码方式。压缩模式的效率在很大程度上取决于待压缩的数据，不过通常情况下 JSON 都是一种更为高效的模式。诸如 CJSON 与 EXI 等方法可用于对 JSON 或 XML 执行特定的编码压缩。这些压缩编码模式要求服务器与客户端模块能够压缩与解压缩数据，但可能会导致无法重用服务。比如，假设开发了一个使用 CJSON 的服务，可以在 iPhone 上使用数据，但却无法在移动 Web 应用上重用该服务。

说明：

可以通过如下 RFC 文档了解 HTTP 响应与请求格式，以及压缩的官方规范：

- RFC 2616——超文本传输协议：http://www.ietf.org/rfc/rfc2616.txt
- RFC 1951——DEFLATE 压缩数据格式规范：http://www.ietf.org/rfc/rfc1951.txt
- RFC 1952——GZIP 文件格式规范：http://www.ietf.org/rfc/rfc1952.txt
- RFC 1950——ZLIB 压缩数据格式规范：http://www.ietf.org/rfc/rfc1950.txt

1. 响应压缩

响应负载压缩是最简单的一种 HTTP 负载压缩形式。HTTP 响应由返回给客户端的用于响应上一个 HTTP 请求的响应头与响应体构成。响应压缩会对响应体应用数据压缩算法，但不会操纵 HTTP 头。

文本数据的响应压缩会对返回给客户端的数据大小产生很大的影响。借助于 JSON 或 XML(请参见第 4 章以了解如何创建与处理 JSON 和 XML 响应)，压缩后的负载可能不到原来负载大小的 10%，不过结果会根据原来负载的简洁程度而有很大变化。如果原来的源使用了 JSON，并且名字只用一个字符，去除了所有空白，那么你会发现相对于打印出冗长元素名的 XML 格式来说，JSON 方式的结果压缩效果就很有限。通常情况下，负载越大，压缩率就越高。一些小的负载由于压缩查找表的缘故可能还会出现大小增加的情况。

在 iOS HTTP 负载中，默认情况下，所有的 HTTP NSURLConnection 请求都是开启压缩的。接收到的负载会自动解压缩并以最初的格式呈现在代码中。解压缩的计算代价要比传输 10 倍字节的通信代价低；因此，激活响应压缩几乎总是有益无害的。

在默认情况下，NSURLConnection 会向每个请求添加如下 HTTP 头：

```
Accept-Encoding: gzip, deflate
```

Accept-Encoding 头告诉服务器，客户端可以接收使用 gzip 或 DEFLATE 压缩的负载，不过服务器可以自己选择是否压缩响应。这样，通过响应负载压缩来提升性能的关键就在于配置服务器以支持压缩。

说明：

有些浏览器无法正确处理 DEFLATE 压缩，因此最常用的压缩其实是 gzip。

比如，配置 Apache Web 服务器的过程涉及加载压缩模块并针对特定的文档类型激活输出过滤器。首先，Apache 配置需要加载两个模块：

```
LoadModulefilter_module library-path/mod_filter.so
LoadModuledeflate_module library-path/mod_deflate.so
```

说明：

library-path 的值会根据 Apache 的安装位置而发生变化。

filter_module 是个常用的模块，并且可能已经被加载了。deflate_module 则不太常用，不过也是 Linux、OS X 与 Windows 上标准 Apache 安装的组成部分。

警告：
deflate_module 的名字容易引起误解，因为它支持的是 gzip 而非 DEFLATE 压缩。

接下来需要定义压缩的内容。最直接的方式就是添加应用于所有内容的输出过滤器。如下代码片段就使用了一个全局的 DEFLATE 输出过滤器：

```
SetOutputFilter DEFLATE
```

这并不是推荐的做法，除非知道 Web 服务器只使用文本数据，它能通过压缩获益。将压缩应用于预先压缩的内容，如图片、音频和视频等会将 CPU 资源消耗在压缩上，但对于负载的大小却没有多少，或是完全没有正面的影响。更加明确的方式是只为那些能够从压缩中获益的内容类型添加输出过滤器。如下代码为几种常见的内容类型应用了压缩：

```
AddOutputFilterByType DEFLATE text/plain
AddOutputFilterByType DEFLATE text/xml
AddOutputFilterByType DEFLATE application/xhtml+xml
AddOutputFilterByType DEFLATE text/css
AddOutputFilterByType DEFLATE application/xml
AddOutputFilterByType DEFLATE application/atom_xml
AddOutputFilterByType DEFLATE application/x-javascript
AddOutputFilterByType DEFLATE text/html
AddOutputFilterByType DEFLATE application/json
```

启用响应压缩对于其他应用与基于浏览器的用户应该是透明的，因为客户端软件(无论是应用还是浏览器)都必须显示声明可以接收带有 Accept-Encoding 头的压缩负载，大多数现代浏览器都是这样做的。压缩的唯一缺点是在开发阶段使用网络嗅探器分析流量时会导致负载难以阅读和调试。推荐的做法是在测试环境中开启压缩，但在开发环境中不启用。

很多其他的 HTTP 服务器(包括微软的 IIS)都支持响应压缩。要想了解如何在 IIS 中开启压缩的相关信息，请参考 http://www.iis.net/ConfigReference/system.webServer/httpCompression。很多负载均衡工具(如 BIG-IP)都支持通过硬件压缩子系统来增强压缩。

如果想要禁用负载压缩，应用可以通过清除自动设定的 Accept-Encoding 头来实现。如下代码示例演示了如何清除这个头：

```
NSMutableURLRequest *request = [[[NSMutableURLRequestalloc]
                                   initWithURL:url
                                   cachePolicy:NSURLCacheStorageAllowed
                               timeoutInterval:20] autorelease];
[request addValue:@"" forHTTPHeaderField:@"Accept-Encoding"];
```

对请求的响应不会再被服务器压缩。响应压缩是优化应用网络带宽使用的一种简单手段，只需要对服务层做很小的修改即可实现。

2. 请求压缩

与响应压缩不同，请求压缩的实现更为复杂，因为它既需要客户端实现，也需要服务端实现。在执行请求压缩时，HTTP 客户端会在将请求体发送给服务器之前对其应用某种类型的数据压缩。Web 浏览器对请求压缩的支持并不太好，因为浏览器不知道目标服务器是否能够支持对请求的解压缩。如果服务器无法理解压缩模式，那么请求就会被丢弃，客户端应用将永远无法得到响应。

由于这个难以解决的问题的存在，请求压缩需要应用与服务端开发者提前做好协调。另一种方式是设定一种模式，iOS 应用首先查询服务器来判断是否支持压缩，然后根据服务器的响应来调整其行为。

请求压缩会为移动应用带来很大的好处，因为广域无线传输速率通常是非对称的，为发送给设备的数据提供了更大的带宽，而对设备发出的数据则提供了很小的带宽。之所以使用这种非对称的带宽，原因在于大多数 Web 流量都是非对称的。如果应用定义了标准的非对称模式，那么你绝对应该考虑使用请求压缩。比如，如果应用收集数据，然后上传到服务器，那么应用就会通过上传负载的压缩而获益。本章的示例应用包含一个简单、中等大小的 XML 文件(40KB)。如果不压缩，那么在传输到服务器以及回传给设备时就会消耗 80KB 的带宽。如果对请求与响应开启了压缩，那么同样一次往返，数据只会消耗 12KB 的带宽。

要想创建请求压缩，首先需要在 Web 服务器上定义好输入过滤器。该例展示了如何在 Apache Web 服务器上做到这一点。

警告：

Apache 并不会在通过 PHP 或 mod_jk 模块等资源过滤器发送数据前对其解压缩。因此，如果通过资源过滤器向 Web 应用传递了压缩数据，那么目标 Web 应用就要负责负载的解压缩。

说明：

与响应压缩一样，客户端应用不应该将 CPU 时间浪费在压缩诸如 PDF、加密数据、图像、音频及视频等已经压缩的内容上。然而，代表预先压缩的数据的 Base64 数据常常会从请求压缩中获益。比如，如果要以 Base64 格式上传 JPEG 文件，那么可以对 Base64 数据进行压缩，相较于未压缩的 Base64 数据，压缩后的数据体积会降低 30%左右。

在 Apache 中，用于响应压缩的模块也可以执行请求压缩。如下配置片段会加载所需模块：

```
LoadModulefilter_module library-path/mod_filter.so
LoadModuledeflate_module library-path/mod_deflate.so
```

说明：

library-path 的值会根据 Apache 的安装位置而发生变化。

接下来需要为 DEFLATE 模块定义输入过滤器。如下代码片段定义了一个输入过滤器和一个 CGI 别名：

```
SetInputFilter DEFLATE
SetOutputFilter DEFLATE
ScriptAlias /cgi/ <html-directory>/cgi-bin/
```

说明：

html-directory 的位置会根据主机系统的配置而发生变化。

如果带有 Content-Encoding:gzip 头的请求到达 HTTP 服务器，HTTP 服务器就会尝试解压缩请求体并将其传给过滤器链中的下一个过滤器。出于说明的目的，这个示例请求压缩应用带有一段简单的 Perl 脚本(参见代码清单 7-1)，它会将接收到的响应体负载回显出来。脚本并不关心接收到的负载是不是压缩过的。

代码清单 7-1　Decomp.cgi

```
#!/usr/bin/perl
print "Content-type: text/html\n\n";
print "Hello, World.\n";
print "Body=\n";
foreach(<>) {
      print;
}
```

接下来，为 iOS 应用添加压缩代码。需要压缩负载并向请求添加 Content-Encoding 头。示例压缩代码使用了 libz.dylib 框架，这需要项目在编译任何压缩代码前先链接到该框架。代码清单 7-2 展示了示例程序中用于压缩 HTTP 请求体的方法：

代码清单 7-2　PostCompress/CompressRequest.m

```
- (NSData *)compressNSData:(NSData *)myData {
    NSMutableData *compressedData = [NSMutableData dataWithLength:16384];

    z_streamcompressionStream;
    // setup the compression stream
    compressionStream.next_in=(Bytef *)[myData bytes];
    compressionStream.avail_in = [myData length];
    compressionStream.zalloc = Z_NULL;
    compressionStream.zfree = Z_NULL;
    compressionStream.opaque = Z_NULL;
    compressionStream.total_out = 0;

    // start the compression of the stream using default compression
    if(deflateInit2(&compressionStream,
                    Z_DEFAULT_COMPRESSION,
                    Z_DEFLATED,
                    (15+16),
```

```
                8, Z_DEFAULT_STRATEGY) != Z_OK) {
        // Something failed
        errorOccurred = YES;
        return nil;
    }


    // loop over the input stream writing bytes into
    // the compressedData buffer in 16K chunks
    do {
        // for every 16K of data compress a chunk into
        // the compressedData buffer
        if(compressionStream.total_out>= [compressedData length]) {
            // increase the size of the output data buffer
            [compressedData increaseLengthBy:16386];
        }

        compressionStream.next_out = [compressedDatamutableBytes] +
                                        compressionStream.total_out;
        compressionStream.avail_out = [compressedData length] -
                                        compressionStream.total_out;
        // compress the next chunk of data
        deflate(&compressionStream, Z_FINISH);
      // keep going until no more compressed data to copy out
    } while(compressionStream.avail_out == 0);

    // end the compression run
    deflateEnd(&compressionStream);
    // set the actual length of the compressed data object
    // to match the number of bytes
    // returned by the compression stream
    [compressedDatasetLength: compressionStream.total_out];
    returncompressedData;
}
```

代码清单 7-3 展示了用于将 Content-Encoding 头添加到请求中的代码。如果没有这个头，DEFLATE 模块就不知道请求内容已经被压缩了。

代码清单 7-3 PostCompress/CompressRequest.h

```
[requestaddValue:@"gzip" forHTTPHeaderField:@"Content-Encoding"];
NSData *compressed = [self compressNSData:myData];
[requestsetHTTPBody:compressed];
reqSize = [compressed length];
```

示例应用实现了压缩并提供了一种方式以执行多个请求(包含了针对你所选择的 URL 的示例负载)。除了请求所消耗的平均时间与总时间外，还会显示出负载的大小。时间包含了计算出的压缩负载所需的时间。图 7-4 展示了使用不同的请求与响应压缩组合时，响应时间的变化。这里的结果是通过 Network Link Conditioner 工具计算出来的，相对于不受控

制的蜂窝网络来说，它会提供更加一致的性能指标。

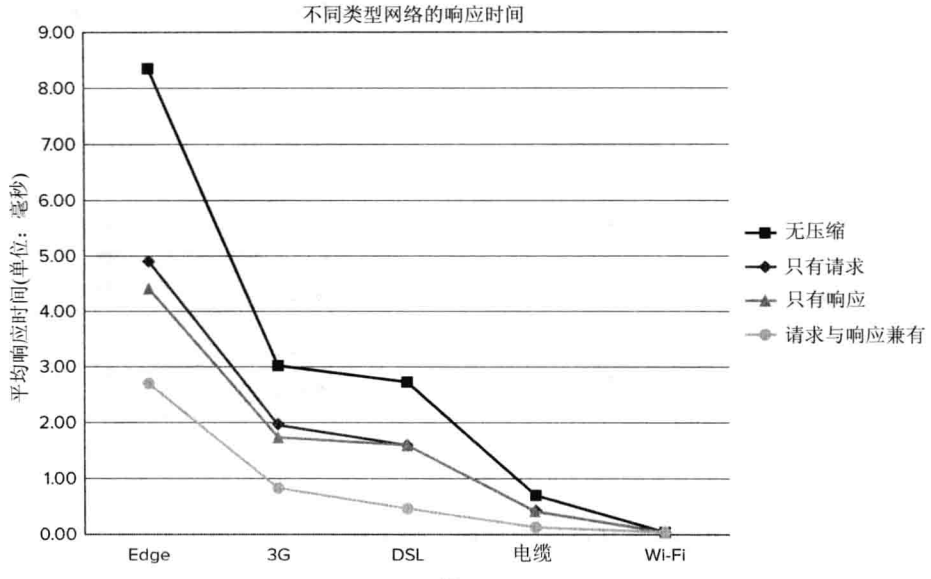


图 7-4

通过 DSL 可以看到收益，因为 DSL 连接对于上下游的速率来说通常是非对称的。值得注意的是，在 Edge 网络上要 10 秒钟才能完成的请求，如果对请求和响应进行了压缩，处理时间会降到 3 秒以下。

7.2.2　降低请求延迟

网络延迟包括在手机与运营商网络间建立连接的时间、建立 TCP 连接的时间，可能还有协商 SSL 连接的时间，以及发送与接收 HTTP 请求的时间。实际上，在 iOS 设备上，我们没有办法降低单个网络请求的延迟，不过可以通过一些技术来降低多个请求的延迟。本节将会介绍如何降低应用网络请求所消耗的潜在时间。

本节介绍的技术提供了一种方式以避免重复消耗这个潜在时间，就好比每次去杂货店只购买购物单上的一件商品是件很傻的事情一样；对于应用所需的一小块数据来说，每次都建立并关闭一条 TCP 连接是非常不明智的。降低请求延迟有两项最佳实践：在单个 TCP 连接上发送 HTTP 请求，以管道的形式发送 HTTP 请求，从而优化全双工 TCP 连接的使用。

你的应用可能已经在使用 HTTP 请求集群了，因为 iOS 默认情况下就是这样做的。当应用使用完 NSURLConnection 对象后，操作系统在关闭连接前会保持它开启几秒钟，通常是 10 秒。这项技术也可以在更高的层次上使用，保持不重要的更新，直到积累了足够的批量数据或是某些用户动作需要网络活动为止。接下来，应用可以按照顺序执行所有的队列请求，在这个过程中一直保持激活相同的连接，避免建立多个 TCP 连接的开销。

另一种方式是使用单个服务端点来架构服务层，它会将请求代理给组织内外的其他服务。这种方式可以通过让应用对不同的活动重用单个连接而避免延迟。

HTTP 管道是重用现有 TCP 连接的第三种方式。它使得 HTTP 客户端能够在对第一个

请求的响应返回前在相同的 TCP Socket 上发送第二个请求。响应返回的顺序与请求发起的顺序保持一致。图 7-5 展示了管道与非管道下的通信中请求与响应流。由于 POST 与 PUT 命令会修改服务器上的实体，因此我们建议不要对这样的请求使用管道。

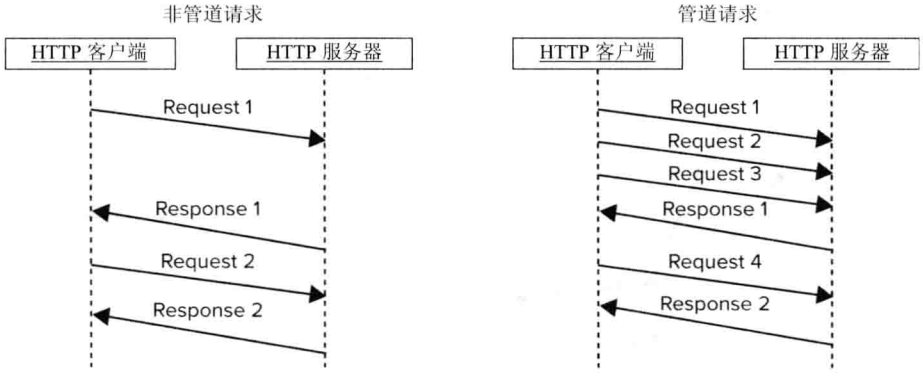


图 7-5

可以轻松为 NSURLRequest 开启管道支持，如下所示：

```
NSMutableURLRequest *request = [[NSMutableURLRequestalloc]
                                initWithURL:[NSURL URLWithString:url]];
[requestsetHTTPShouldUsePipelining:YES;
```

说明：
上述请求是可变的。

使用这种方式时需要对目标服务器进行广泛测试，因为并非所有的服务器都支持 HTTP 管道。Apache 与 IIS 都支持管道，无需任何额外配置。

7.2.3 避免网络请求

除了减小请求大小以减少带宽以及将其分组以避免延迟外，改进网络性能的最佳方式其实是不使用网络。本节将会介绍 HTTP 缓存的基础知识，如何在 iOS 应用中利用这些规则，从而在本地缓存内容以避免不必要的网络流量。

IETF 在 RFC 2616 中明确定义了 Web 浏览器与 Web 服务器之间的 HTTP 缓存的工作方式，可以在 http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html 上找到相关信息。HTTP 被设计为针对浏览器与服务器之间的通信，缓存机制也是针对这种使用模式的。iOS 提供了一种机制来利用标准 HTTP 缓存，以及采取相应的行为。通过 NSURLRequest 发出的每个请求都会经过缓存组件。该组件是 NSURLCache 或其子类的实例。这个对象是 iOS 采用的管理来自服务器的响应缓存的标准机制。

1. 默认缓存行为

在默认情况下，NSURLRequest 遵照 RFC 2616 来管理缓存。该默认行为指定缓存要返回大多数当前的内容副本。如果返回的内容不是最新的，就会发出警告；如果无法返回内

容，则报告错误。

在 iOS 上，这意味着只有在响应头表明响应能够缓存的情况下，当第一次返回时才会将其缓存到内存中。对于后续发送给相同 URL 的请求来说，iOS 会使用 If-Modified-Since 头(包含缓存响应的修改日期与时间)向服务器发送请求。如果服务器确定自从请求头提供的时间开始内容没有被修改，就会返回状态为 304 的响应，并且没有响应体。通过这个响应，iOS 能够确定它所缓存的副本是最新的内容，并使用 200 状态码返回，这很有效地对应用代码隐藏了缓存活动。在内容来自于内容分发网络(CDN)的网络配置中，源 URL 对于不同的请求来说可能是不同的，因此对于 HTTP 标准定义的缓存机制是不适用的。

这些标准的缓存规则的设计专门针对与 Web 浏览器的交互。使用 HTTP 作为传输协议的移动应用可以适当修改这些默认规则来改进性能并满足应用的需求。iOS 中的 URL 加载系统向客户端应用提供了一种方式来覆写默认行为。在覆写默认行为时，你需要花一些时间来充分理解可能会导致应用出现缺陷的边际行为。

可以通过为请求设定缓存策略来覆写默认的缓存规则，如下所示：

```
NSMutableURLRequest *request=[[NSMutableURLRequestalloc]
                                    initWithURL:[NSURL URLWithString:url]];
[requestsetCachePolicy:NSURLRequestUseProtocolCachePolicy];
```

说明：

注意上面的请求是 NSMutableURLRequest，这样代码就可以修改请求参数了。

iOS 提供了 6 种不同的设置，使得开发者能够控制响应缓存的方式：

- NSURLRequestUseProtocolCachePolicy——该设置告诉系统遵照 RFC 2616 指定的规则。
- NSURLRequestReloadIgnoringLocalCacheData——该设置告诉请求略过本地缓存，从网络上检索新的内容。如果某些网络设备(如缓存网络代理)介于应用与数据源之间，并且持有内容的缓存副本，就会返回缓存副本。
- NSURLRequestReloadIgnoringLocalAndRemoteCacheData——该设置告诉请求略过本地缓存并将头添加到请求中，同时告诉中间设备也略过缓存，提供源服务器上的最新数据。
- NSURLRequestReturnCacheDataElseLoad——该设置会让缓存系统返回一份内容的缓存副本而不去验证服务器上是否有最新的副本。如果请求的缓存副本在缓存中存在，就会将其返回。如果缓存副本不存在，那就通过网络请求检索内容。该设置提供了最快的响应时间，但却最有可能返回过期的数据。要想通过该设置带来收益，请使用该类型的请求向用户提供最初的快速响应，然后在后台线程中发出请求，使用服务器的最新数据来刷新缓存。
- NSURLRequestReturnCacheDataDontLoad——该设置指定只返回缓存中的内容。如果内容不在缓存中，那就会返回错误而不是从服务器上获取内容。

- NSURLRequestReloadRevalidatingCacheData——该设置总是会重新验证数据。在某些情况下，缓存的响应可能会有过期时间，到了这个时间后系统就会检查最新的数据。如果使用该设置，那就会忽略掉过期时间，并且总是验证服务器有没有最新的内容。

除了配置每个请求使用缓存的方式外，还可以通过配置应用的 NSURLCache 对象来指定应用所能缓存的数据量。

2. 配置 NSURLCache

在应用使用任何标准的 iOS 类创建网络请求时，系统都会创建 NSURLCache 实例。在默认情况下，该实例只会将数据缓存在 RAM 中，这意味着当程序退出时，其缓存的请求就会被清空。当设备处于低内存状态时也会清空 RAM 缓存。

iOS 提供了一种方式来重新定义默认的缓存，并指定了更大的内存容量与持久化存储，以便缓存在应用重启后依然可以使用。如下代码片段就展示了重新定义的默认缓存：

```
NSURLCache *cache = [[NSURLCachealloc]
        initWithMemoryCapacity:1024*1024
                  diskCapacity:1024*1024*20
                      diskPath:@"URLCache"];
  [NSURLCachesetSharedURLCache:cache];
```

该例创建 1MB 的内存缓存和 20MB 的持久化缓存。缓存数据库的位置位于应用的沙箱，在 Library/Caches 目录下，文件名为 URLCache。示例代码的第 2 行将应用的缓存实例设定为上一行创建的那一个。

iOS 中有一种奇怪的现象，即在某些情况下，应用中的系统组件会将缓存的内存容量设为 0MB，这就禁用了缓存。解决这个无法解释的行为的一种方式就是通过自己的实现子类化 NSURLCache，拒绝将内存缓存大小设为 0。如下代码展示了防止这种行为的一个子类实现：

```
@interface NonZeroingCache :NSURLCache

@end

@implementation NonZeroingCache

-(void)setMemoryCapacity:(NSUInteger)newMemSize
{
    if(newMemSize == 0) {
        NSLog(@"Attempt to set cache size to 0");
        return;
    }
    [supersetMemoryCapacity: newMemSize];
}

@end
```

setMemoryCapacity:方法中的代码会验证大小不为 0，并调用父类，从而将新的大小设为除了 0 之外的其他值。

可以通过压缩数据及管道化请求以最大化地提升应用的性能，不过最快的请求实际上是没有发出的请求。通过仔细考虑应用需求以及服务器的行为，可以将数据保留在缓存中，只有当服务器上的数据发生变化时才刷新，从而避免发出这些请求。

7.3　小结

iOS 用户都希望应用能够立刻响应每个请求。移动产业有这样一条原则，即屏幕越小，用户越没耐心。提供让用户乐于使用的应用意味着要珍惜用户的时间，就像珍惜你自己的时间一样。通过压缩请求与响应来优化应用所使用的带宽，通过管道化请求避免不必要的延迟，甚至通过缓存响应来避免冗余的网络请求都会加速应用并改进用户体验。

第 8 章

底层网络

本章内容

- 解密 BSD Socket
- 实现 CFNetwork API
- 使用 NSStream 进行开发

本章代码下载

可以在 www.wrox.com 的 Download Code 选项卡中下载本章代码，网址是 www.wrox.com/remtitle.cgi?isbn=9781118362402。本章代码位于一个示例项目中，压缩为 Low-Level Networking.zip。

每个复杂的计算机系统都是构建在一个或多个抽象层之上，底层网络也不例外。网络的根本是 Berkley 或 BSD Sockets。它执行大多数基础的网络任务：发送与接收一系列的二进制位。由于需要使用相当数量的代码才能恰当地发送一个字节，而且相同的逻辑对于每个 Socket 都要重复执行，因此人们构建了库来封装该逻辑，这样就能被其他人重用了。在 iOS 上，这个库叫做 Core Foundation networking 或 CFNetwork，它是对原始 Socket 的轻量级封装，不过它很快对于大多数常见场景来说就变得非常笨重了。最后，添加了另一层(NSStream)来封装 CFNetwork，并且作为最基础的 Objective-C 网络 API。大家更熟悉的类(比如 NSURLConnection 与 UIWebView)都是非常易于使用的，而且能够通过很少的代码完成很多事情，这都是由于这些底层库所提供的坚实基础而实现的。本章将会通过示例介绍每一个库，连接到相同的服务器，执行相同的任务，这样就能比较它们的能力与复杂度了。

8.1 BSD Socket

BSD Socket API 最初是由加利福尼亚大学伯克利分校的研究人员在 20 世纪 80 年代末

实现的。最终，它被美国电气和电子工程师协会(IEEE)标准化为 Portable Operating System Interface (POSIX) Socket API，并作为 UNIX 与类 UNIX 操作系统的标准。其流行程度与易于使用的特性也激发了另一个类似的实现，即 Winsock API for Microsoft Windows。在某个层次上，BSD Socket 担负起几乎所有的 Internet 流量，并且可以让应用程序开发人员完全控制到远程设备或服务器的通信。Socket 指的是两个端点之间的单向连接，因此它们通常情况下都是成对出现的；一个用于读，一个用于写。类似于 UNIX 系统上的几乎所有其他资源，Socket 也表示为文件，并且在创建时会被分配文件描述符。表 8-1 总结了 9 个最常见的 Socket API 调用。

表 8-1 BSD Socket API 调用

API 调用	说 明
int socket(int addressFamily, int type, int protocol)	创建并初始化新的 Socket。如果成功，就返回一个文件描述符，失败则返回 0。
int bind(int socketFileDescriptor, sockaddr *addressToBind, int addressStructLength)	为addressToBind结构体指定的地址与端口号分配 Socket
int accept(int socketFileDescriptor, sockaddr *clientAddress, int clientAddressStructLength)	接受连接请求，将客户端地址存储到 clientAddress 中
int connect(int socketFileDescriptor, sockaddr *serverAddress, int serverAddressLength)	连接到 serverAddress 指定的服务器
hostent* gethostbyname(char *hostname)	使用 DNS 查找与指定主机名对应的 IP 地址
int send(int socketFileDescriptor, char *buffer, int bufferLength, int flags)	在 Socket 上从 buffer 发送最多 bufferLength 个字节
int receive(int socketFileDescriptor, char *buffer, int bufferLength, int flags)	在Socket上将最多bufferLength个字节读取到buffer中
int sendto(int socketFileDescriptor, char *buffer, int bufferLength, int flags, sockaddr *destinationAddress, int destinationAddressLength)	从 buffer 向 destinationAddress 发送最多 bufferLength 个字节
int recvfrom(int socketFileDescriptor, char *buffer, int bufferLength, int flags, sockaddr *fromAddress, int *fromAddressLength)	在 Socket 上将最多 bufferLength 个字节读取到 buffer 中，然后将发送者的地址存储到 fromAddress 中

BSD Socket 完全使用 C 语言实现，并且可以在 Objective-C 代码中使用。如果想要重用现有的网络库，使用别的平台移植过来的代码而又不想做过多工作，之前有 Socket 经验并且不想学习 Apple 的高层框架，那么这是非常方便的。不过 Apple 并不推荐这种方式，这是因为原始的 Socket 无法访问操作系统内建的网络特性(比如系统范围的 VPN)。更糟糕的是，初始化 Socket 连接并不会自动打开设备的 Wi-Fi 或是蜂窝无线电。无线电会智能地

关闭以节省电池电量，任何通信连接都会失败，除非其他网络进程激活了无线电。CFNetwork 对 BSD Socket 的封装可以激活设备的无线电，因此在几乎所有场景中，我们都建议使用 CFNetwork 而非 BSD Socket。

要想创建 Socket，请调用 socket(int addressFamily, int type, int protocol)并提供 socket.h 中定义的网络域、Socket 类型与协议枚举值。通常情况下，对于 iOS 应用发出的流量，addressFamily 值为 IPv4(AF_INET)或 IPv6(AF_INET6)；不过，也可以打开到本地文件的 Socket。Socket 类型通常会被设定为流(SOCK_STREAM)或数据包(SOCK_DGRAM)。这两个值非常重要，因为调用 socket()时常常会提供值为 0 的协议，这表示系统可以通过域和类型值自动选择恰当的协议。对于流式 Socket 来说，自动选择的值为传输控制协议(IPPROTO_TCP)；对于数据包 Socket 来说，自动选择的值为用户数据报协议(IPPROTO_UDP)。第 12 章“使用 GameKit 实现设备间通信”将会详细介绍这两个协议的语义。如果成功创建了 Socket，那么返回值就是新文件说明符的号码；不过，如果由于某个原因导致调用失败，那么返回值就为-1。这时，通信尚未开始，Socket 也没有被指定为输入或输出 Socket(直到首次使用 Socket 时才会指定)。客户端现在可以开始与服务器建立连接了；不过，在开始通信前服务器需要一个或多个调用。

8.1.1 配置 Socket 服务器

BSD Socket 服务器必须通过调用 bind(int socketFileDescriptor, sockaddr *addressToBind, int addressStructLength)与具有唯一地址的 Socket 关联。它会接收一个 Socket 并将其分配或是绑定到某个特定的地址与端口。绑定成功返回 0，否则返回-1。绑定 Socket 后，接下来的步骤取决于你在 socket()调用中指定的连接类型，即 UDP 或 TCP：

- 对于 UDP Socket 来说，现在就可以开始向外界传输数据了，因为 UDP 是个无连接的协议，不需要在另一端监听。
- TCP Socket 是个面向连接的协议，需要在 Socket 的另一端有参与者。要想建立 TCP 连接，你需要调用 listen(int socketFileDescriptor, int backlogSize)以建立好缓冲区队列的数据结构。

作为第一个参数传递进去的Socket会成为只读Socket，不能用于发送消息。backlogSize 表示有多少个挂起的连接在排队的同时等待服务端代码的使用。在监听时，服务器会等待进来的连接请求并调用 accept(int socketFileDescriptor, sockaddr *clientAddress, int clientAddressStructLength)来接收请求。这会将挂起的请求从缓冲队列中移除，并使用客户端的地址信息(最主要的是IP地址与端口)来装配clientAddress结构体。接受了挂起的请求后，服务器就可以从客户端接收消息了。

8.1.2 Socket 客户端连接

客户端的第一个动作取决于 Socket 使用的协议。对于 TCP Socket 来说，客户端首先要通过 connect(int socketFileDescriptor, sockaddr *serverAddress, int serverAddressLength)协商一个到服务器的连接。在 TCP 握手时该调用会阻塞，成功返回 0，失败返回-1。对于 UDP

Socket 来说，connect()调用是可选的；不过，调用它会为所有的 UDP 传输 Socket 设定默认地址。这样会方便 UDP 数据包的发送与接收。如果设备通过主机名而非 IP 地址进行连接，那么它可能不清楚该如何继续，因为 sockaddr 结构体只包含一个 IP 地址。人们通过域名解析系统(Domain Name System，DNS)来解决将主机名转换为 IP 地址的问题。hostent* gethostbyname(char *hostname)函数会对指定的主机名执行阻塞 DNS 查询。hostent 结构体包含与主机对应的 IP 地址列表，格式兼容于 Socket API 中所用的 sockaddr 结构体。如果主机名包含以点分法表示的 IP 地址，那么调用只是使用给定的 IP 地址来装配返回的 hostent 结构体的第一个结果并立刻返回。借助于这种行为，用户只需提供 IP 地址或主机名即可，不会干扰到连接逻辑。如果根据给定的主机名无法找到 DNS 条目，那么 gethostbyname()调用就会返回 NULL。

建立好 Socket 连接后，设备就可以通过它发送或接收消息了。有两对 Socket API 调用可供使用，使用哪一对取决于所用的 Socket 类型。TCP Socket 使用 int send(int socketFileDescriptor, char *buffer, int bufferLength, int flags)与 int receive(int socketFileDescriptor, char *buffer, int bufferLength, int flags)对。在发送时，由 socketFileDescriptor 描述的 Socket 会将缓存中介于 0 与 bufferLength 之间的字节发送出去。如果成功，那么会返回成功发送出去的字节数量，失败则返回-1。在接收时，缓存会通过从 Socket 读取的第一个 bufferLength 长度的字节副本来装配。类似于 send()调用，如果成功，receive()调用也会返回成功读取的字节数量，失败则返回-1。之前使用 connect()调用来设定默认地址的 UDP Socket 也可以像 TCP Socket 那样使用 send()调用与 receive()调用。否则，UDP Socket 必须使用第 2 对针对无连接协议的 API 调用。

UDP Socket 可以通过 int sendto(int socketFileDescriptor, char *buffer, int bufferLength, int flags, sockaddr *destinationAddress, int destinationAddressLength) API 调用使用相同的 Socket 连接发送给多个地址。这种调用行为类似于 send()，只不过它为目标地址提供了额外的参数。由于 UDP 并不会确保消息的传递，因此它可以通过另一个 sendto()调用使用 Socket 发送给另一个地址。相应的无连接的接收调用是 int recvfrom(int socketFileDescriptor, char *buffer, int bufferLength, int flags, sockaddr *fromAddress, int *fromAddressLength)，其行为类似于 sendto()。请注意如下重要差别：最后一个参数是指向整数的指针，它的值是 fromAddress 结构体的最终长度。由于 UDP Socket 通常并不会连接到单个端点，因此接收数据包的代码需要知道数据包来自于何处，recvfrom()则会将该信息赋给 fromAddress。

既然我们现在已经可以连接到其他设备，可以发送与接收数据，那么如下示例就将这一切连接起来。该示例应用连接到仓库中的一台监控服务器，该服务器会控制警报与温控系统。示例应用必须使用底层网络框架进行连接，这是因为仓库中的服务器会通过 Telnet 协议报告结果，而诸如 NSURLConnection 等高层对象并不直接支持该协议。3 个独立的网络控制器都会加载相同的 Telnet 结果，不过使用的是不同的底层网络框架并将结果展示给用户。仓库服务器会根据如下代码片段使用格式化的字符串进行响应：

```
84,60,+67,1,1,0,0,0,1
```

```
{room temperature},{outlet temperature},{coil temperature},
{compressor status},{air switch status},{auxiliary heat status},
{front door status},{system status},{alarm status}
```

每个控制器都在后台线程中获取数据，从而在网络通信过程中不会阻塞用户界面，如代码清单 8-1 所示。

代码清单 8-1 在后台线程中获取结果(LLNNetworkingController.m)

```
- (void)start {
    NSURL *url = [NSURL
            URLWithString:[NSStringstringWithFormat:@"telnet://%@:%i",
        self.urlString, self.portNumber]];

    NSThread *t = [[NSThreadalloc] initWithTarget:self
                                    selector:@selector(loadCurrentStatus:)
                                      object:url];
    [t start];
}
```

网络控制器会通过两条委托消息向用户界面报告结果：networkingResultsDidLoad:将网络结果作为参数，networkingResultsDidFail:则接收一条只读的消息，表示错误。加载仓库服务器结果的完整 Socket 实现如代码清单 8-2 所示。

代码清单 8-2 使用 BSD Socket 加载结果(LLNBSDSocketController.m)

```
- (void)loadCurrentStatus:(NSURL*)url {
    if([self.delegaterespondsToSelector:@selector
                                        (networkingResultsDidStart)]) {
        [self.delegatenetworkingResultsDidStart];
    }

    // create a new Internet stream socket
    socketFileDescriptor = socket(AF_INET, SOCK_STREAM, 0);

    if(socketFileDescriptor == -1) {
        if([self.delegaterespondsToSelector:
                @selector(networkingResultsDidFail:)]) {
            [self.delegatenetworkingResultsDidFail:
                @"Unable to allocate networking resources."];
    }

    return;
}


// convert the hostname to an IP address
structhostent *remoteHostEnt = gethostbyname([[url host] UTF8String]);

if(remoteHostEnt == NULL) {
```

```
if([self.delegaterespondsToSelector:
        @selector(networkingResultsDidFail:)]) {
    [self.delegatenetworkingResultsDidFail:
        @"Unable to resolve the hostname of the warehouse server."];
    }

    return;
}

structin_addr *remoteInAddr = (structin_addr
                                    *)remoteHostEnt->h_addr_list[0];

// set the socket parameters to open that IP address
structsockaddr_insocketParameters;
socketParameters.sin_family = AF_INET;
socketParameters.sin_addr = *remoteInAddr;
socketParameters.sin_port = htons([[url port] intValue]);

// connect the socket; a return code of -1 indicates an error
if(connect(socketFileDescriptor, (structsockaddr *) &socketParameters,
        sizeof(socketParameters)) == -1) {

    NSLog(@"Failed to connect to %@", url);

    if([self.delegaterespondsToSelector:
            @selector(networkingResultsDidFail:)]) {

        [self.delegatenetworkingResultsDidFail:
            @"Unable to connect to the warehouse server."];
    }

    return;
}

NSLog(@"Successfully connected to %@", url);

NSMutableData *data = [[NSMutableDataalloc] init];
BOOL waitingForData = YES;

// continually receive data until you reach the end of the data
while(waitingForData){
const char *buffer[1024];
int length = sizeof(buffer);

// read a buffer's amount of data from the socket; the number
// of bytes read is returned
int result = recv(socketFileDescriptor, &buffer, length, 0);

// if you got data, append it to the buffer and keep looping
if(result > 0){
```

```
            [dataappendBytes:bufferlength:result];

            // if you didn't get any data, stop the receive loop
            } else {
                waitingForData = NO;
            }
        }
        // close the stream since you are done reading
        close(socketFileDescriptor);


        NSString *resultsString = [[NSStringalloc] initWithData:data
                                                encoding:NSUTF8StringEncoding];

        NSLog(@"Received string: '%@'", resultsString);

        LLNNetworkingResult *result = [self parseResultString:resultsString];

        if(result != nil) {
            if([self.delegaterespondsToSelector:
                    @selector(networkingResultsDidLoad:)]) {

                [self.delegatenetworkingResultsDidLoad:result];
            }
        } else {
            if([self.delegaterespondsToSelector:
                    @selector(networkingResultsDidFail:)]) {

                [self.delegatenetworkingResultsDidFail:
                    @"Unable to parse the response from the warehouse server."];
            }
        }
    }
```

Socket 控制器首先会通过 socket()创建一个新的 Internet 流式 Socket。接下来接收服务器的主机名并通过 gethostbyname()获取其 IP 地址。如果这两个调用成功，那么应用就可以使用主机名与端口来装配 sockaddr_in 结构体了，然后使用它来连接到服务器。在设置端口时，整数值会通过 htons()转换为网络字节序，这样可以确保它能够被大端与小端系统读取。在 connect()调用中，sockaddr_in 会被转换为 sockaddr，之所以可以这样，是因为当 sockaddr_in.sin_family 为 AF_INET 时这两个结构体都有相同的布局。当 Socket 连接上之后，应用就可以将任何可用数据读取到NSMutableData中以供后续处理了。会读取1024字节大小的块，直到读调用表明没有更多的可用数据时为止。当下载完所有数据后，数据会被转换为字符串，被解析到每个环境数据中，并传递给 UI 以展示给用户。清理 Socket 只需调用 Socket 文件描述符的 close()即可。

8.2 CFNetwork

CFNetwork 位于框架层次的更上一层，是对 BSD Socket 的一层轻量级封装。你会发现两者在回调方法与逻辑流方面存在很多相似性，这都是因为底层 BSD 基础的结果。如前所述，相对于原始的 Socket 来说，CFNetwork 的主要优势在于被集成到系统级的设置与主运行循环中。沿着框架层次越往上就越能获得更好的系统服务，比如必要时开启无线以及通过系统范围的 VPN 进行路由等，并且没有什么严重的缺陷。

运行循环集成对于线程来说是必不可少的，也是 iOS 中事件处理流的基础。每个线程都有自己的运行循环，主线程循环叫做主运行循环或 UI 运行循环。每个循环都会调度异步事件的处理，如果没有事件发生，线程就会睡眠。键盘与手势输入、网络事件与定时器等都是事件，每个事件都需要由自定义的应用逻辑进行处理。正如接下来的代码示例所演示的，每个线程都有自己的运行循环；不过，每个次级线程都必须显式启动与停止自己的循环。CFNetwork 使用了一种类似于高层次 Objective-C 框架所用的委托的回调系统，并且只能在支持运行循环的框架中使用。

C函数用于创建和打开Socket，回调系统用于处理源自于它的所有事件。CFNetwork包含一个便捷的创建函数CFStreamCreatePairWithSocketToHost()，可以针对给定的主机名与端口创建一对Socket，一个用于读，另一个用于写。框架会负责将主机名转换为IP地址，将端口号转换为网络字节序。如果不需要其中一个Socket，那么只需将NULL作为读或写流参数，这样就不会创建它了。正像原始Socket一样，必须在使用前显式通过CFReadStreamOpen()或CFWriteStreamOpen()打开流。这两个调用都是异步的，并且在成功打开后会通过kCFStreamEventOpenCompleted调用回调函数。代码清单 8-3 展示了如何创建与打开流，它所使用的示例与上一个示例一样。

代码清单 8-3 使用 CFNetwork 创建 Socket(LLNCFNetworkController.m)

```
- (void)loadCurrentStatus:(NSURL*)url {
    if([self.delegaterespondsToSelector:@selector
                                        (networkingResultsDidStart)]) {
    [self.delegatenetworkingResultsDidStart];
    }

    // keep a reference to self to use for controller callbacks
    CFStreamClientContextctx = {0, (__bridge void *)(self), NULL, NULL, NULL};

    // get callbacks for stream data, stream end, and any errors
    CFOptionFlagsregisteredEvents = (kCFStreamEventHasBytesAvailable |
                                     kCFStreamEventEndEncountered |
                                     kCFStreamEventErrorOccurred);

    // create a read-only socket
    CFReadStreamRefreadStream;
```

```
CFStreamCreatePairWithSocketToHost(kCFAllocatorDefault,
                                   (__bridge CFStringRef)[url host],
                                   [[url port] integerValue],
                                   &readStream,
                                   NULL);

// schedule the stream on the run loop to enable callbacks
if(CFReadStreamSetClient(readStream, registeredEvents,
                         socketCallback, &ctx)) {

    CFReadStreamScheduleWithRunLoop(readStream,
                                    CFRunLoopGetCurrent(),
                                    kCFRunLoopCommonModes);
} else {
    NSLog(@"Failed to assign callback method");

    if([self.delegaterespondsToSelector:
          @selector(networkingResultsDidFail:)]) {

        [self.delegatenetworkingResultsDidFail:
          @"Unable to respond to data from the warehouse server."];
    }

    return;
}


// open the stream for reading
if(CFReadStreamOpen(readStream) == NO) {
    NSLog(@"Failed to open read stream");

    if([self.delegaterespondsToSelector:
          @selector(networkingResultsDidFail:)]) {
       [self.delegatenetworkingResultsDidFail:
          @"Unable to read data from the warehouse server."];
    }

    return;
}

CFErrorRef error = CFReadStreamCopyError(readStream);

if(error != NULL) {
    if(CFErrorGetCode(error) != 0) {
       NSLog(@"Failed to connect stream; error '%@' (code %ld)",
          (__bridge NSString*)CFErrorGetDomain(error),
          CFErrorGetCode(error));
    }

    CFRelease(error);
```

```
        if([self.delegaterespondsToSelector:
                @selector(networkingResultsDidFail:)]) {

            [self.delegatenetworkingResultsDidFail:
                @"Unable to connect to the warehouse server."];

        }

        return;
    }

    NSLog(@"Successfully connected to %@", url);


    // start processing
    CFRunLoopRun();
}
```

初始化最后剩下的一部分内容是注册 Socket 回调函数，这需要如下 3 个步骤：

(1) 首先，创建一个回调函数并传递给流。读写流有相同的回调签名，区别只在于第一个参数表示的流引用的类型。比如，回调函数可以是具有方法签名 socketCallback (CFReadStreamRef stream, CFStreamEventType event, void *info)的任何函数。第一个参数是产生事件的流，几乎用于处理流的每个函数都有一个指向它的引用。产生回调的事件作为 event 被传递进去，它可以是 CFStreamEventType 值之一，如表 8-2 所示。最后一个值是在注册流的上下文时设置的参数，它可以是指向任何数据的指针，只要这些数据有助于你处理流数据即可。

(2) 接下来，为流装配一个 CFStreamClientContext 结构体来创建之前提到的上下文，它会持有一个指针，指向你自己的 info 对象与回调函数(用于保持、释放或复制该对象)。通常情况下，你会传递一个 self 引用，对于 info 来说，它是管理流的对象，对于每个可选的回调来说就是 NULL。

(3) 最后，选择代码想要接收的流事件。只需要存储 CFStreamEventType 的一个按位或的值即可，如表 8-2 所示，这表示你希望回调函数处理的那些事件。

完成上面这 3 步后，调用 CFReadStreamSetClient(CFReadStreamRef stream, CFOptionFlags streamEvents, CFReadStreamClientCallBack callbackFunction, CFStreamClientContext *context)，将回调注册到流上。代码清单 8-4 将这些概念整合到了一个针对只读流的回调函数中。

表 8-2 CFStream 事件类型

事 件 常 量	说 明
kCFStreamEventOpenCompleted	Socket 被成功打开
kCFStreamEventHasBytesAvailable	Socket 有可以读取的字节
kCFStreamEventCanAcceptBytes	Socket 缓存中可以写入字节

(续表)

事件常量	说明
kCFStreamEventErrorOccurred	操作出现错误。CFReadStreamCopyError()或 CFWriteStreamCopyError()会提供关于错误的更多细节信息
kCFStreamEventEndEncountered	Socket 到达字节流的末尾
kCFStreamEventNone	该默认值不表示任何事件

代码清单 8-4 只读流回调示例(LLNCFNetworkController.m)

```
voidsocketCallback(CFReadStreamRef stream, CFStreamEventType event,
                                                    void *myPtr) {
    LLNCFNetworkController *controller = (__bridge
                                        LLNCFNetworkController*)myPtr;
    switch(event) {
        case kCFStreamEventHasBytesAvailable:
            // read bytes until there are no more
            while(CFReadStreamHasBytesAvailable(stream)) {
                UInt8 buffer[kBufferSize];
                intnumBytesRead = CFReadStreamRead(stream, buffer,
                                                    kBufferSize);

                [controllerdidReceiveData:[NSDatadataWithBytes:buffer
                                                length:numBytesRead]];
            }

            break;

        case kCFStreamEventErrorOccurred: {
            CFErrorRef error = CFReadStreamCopyError(stream);

        if(error != NULL) {
            if(CFErrorGetCode(error) != 0) {
                NSLog(@"Failed while reading stream; error '%@' (code %ld)",
                      (__bridge NSString*)CFErrorGetDomain(error),
                      CFErrorGetCode(error));
              }

              CFRelease(error);
        }

        if([controller.delegaterespondsToSelector:
                  @selector(networkingResultsDidFail:)]) {

              [controller.delegatenetworkingResultsDidFail:
                  @"An unexpected error occurred while reading from
                  the warehouse server."];
            }
```

```
            break;
        }

        case kCFStreamEventEndEncountered:
            [controllerdidFinishReceivingData];

            // clean up the stream
            CFReadStreamClose(stream);

            // stop processing callback methods
            CFReadStreamUnscheduleFromRunLoop(stream,
                                              CFRunLoopGetCurrent(),
                                              kCFRunLoopCommonModes);

            // end the thread's run loop
            CFRunLoopStop(CFRunLoopGetCurrent());

            break;

        default:
            break;
    }
}
```

即便成功注册了回调，也只有在运行循环得到恰当处理后才会被调用。幸好，这只需要简单的两个步骤即可：将回调注册到运行循环以及启动运行循环。顾名思义，CFReadStreamScheduleWithRunLoop(CFReadStreamRef stream, CFRunLoopRef runLoop, CFStringRef runLoopMode)以及与其对应的写入流会根据给定的运行循环来调度流。runLoop 几乎总是 CFRunLoopGetCurrent()，它会返回当前线程的默认运行循环，runLoopMode 几乎总是 kCFRunLoopCommonModes，它包含了默认模式和为当前线程手工添加的模式。在将流集成到运行循环中后，你只需要通过 CFRunLoopRun()开启循环并等待事件发生即可。

8.3 NSStream

沿着框架层次再往上就是 NSStream 了，它是针对 CFNetwork API 的 Objective-C 封装器。NSStream 使用委托协议 NSStreamDelegate，几乎完全模仿了 CFNetwork 流回调函数的功能，所有同样的运行循环需求也都适用于 NSStream。NSStream 有两个具体类，分别是 NSInputStream 与 NSOutputStream，它们分别与 CFReadStream 和 CFWriteStream 完成相同的功能。

实现 NSStream 以进行读或写是相当直接的，只有一处需要注意：iOS SDK 并不支持 NSHost，但使用 NSStream 的指定初始化器 getStreamsToHost:port:inputStream:outputStream:

却需要用到 NSHost。由于这是几乎每个开发者的一个痛点，特别是那些从 Mac OS X 移植代码的开发者，因此 Apple 发布了一份技术说明(Technical Q&A QA1652，http://developer.apple.com/library/ios/#qa/qa1652/_index.html)，通过一个 NSStream 类别来模拟这个缺失的功能，代码清单 8-5 中的示例应用就使用到了它。该解决方案利用 CFReadStreamRef 与 NSInputStream 之间的 Toll-Free Bridging，使用 CFNetwork 框架来创建流，代码清单 8-3 介绍了其创建方式。

Toll-Free Bridging

如果两个对象是 Toll-Free Bridged 的，就意味着对 Core Foundation 对象的引用可以替换为 Objective-C Foundation 对象。比如，NSString、NSArray、NSDictionary 与 NSInputStream 等常见类型分别可以桥接为 CFStringRef、CFArrayRef、CFDictionaryRef 与 CFReadStreamRef。虽然互换桥接类型看起来不可思议，不过这却是可能的，这是由于 Foundation 对象的具体类在内存中的布局方式以及各种 Core Foundation C 调用的强力检查的原因。复杂性已经被隐藏掉了，你只需要使用便捷的桥接类型即可。

代码清单 8-5 无需 NSHost 创建到某主机的 NSStream(NSStream+StreamsToHost.m)

```
+ (void)readStreamFromHostNamed:(NSString *)hostName
                           port:(NSInteger)port
                     readStream:(out NSInputStream **)readStreamPtr {

    assert(hostName != nil);
    assert((port > 0) && (port < 65536));
    assert((readStreamPtr != NULL));

    CFReadStreamRef readStream = NULL;

    CFStreamCreatePairWithSocketToHost(NULL,
                                       (__bridge CFStringRef) hostName,
                                       port,
                                       ((readStreamPtr != NULL) ? &readStream :
                                          NULL),
                                       NULL);

    if(readStreamPtr != NULL) {
        *readStreamPtr = CFBridgingRelease(readStream);
    }
}
```

在轻松创建好流之后，你只需要按照之前的示例实现采取相同步骤即可。代码清单 8-6 展示了如何创建一个只读流、设置委托、在运行循环中调度，然后打开它。对于 BSD 或 CFNetwork 实现来说，这些步骤可能要编写不少代码；不过，这里只需要 6 行代码即可！

代码清单 8-6 初始化 NSInputStream(LLNNSStreamController.m)

```
- (void)loadCurrentStatus:(NSURL *)url {
    if([self.delegate respondsToSelector:
          @selector(networkingResultsDidStart)]) {

       [self.delegate networkingResultsDidStart];
    }

    NSInputStream *readStream;
    [NSStream readStreamFromHostNamed:[url host]
                                 port:[[url port] integerValue]
                           readStream:&readStream];

    [readStream setDelegate:self];
    [readStream scheduleInRunLoop:[NSRunLoopcurrentRunLoop]
                          forMode:NSDefaultRunLoopMode];

    [readStream open];

    [[NSRunLoopcurrentRunLoop] run];
}
```

实现 NSStreamDelegate 协议类似于 CFReadStream 回调函数并包含了相同的事件。代码清单 8-7 展示了一个示例实现，它会将仓库服务器种子读取到一个 NSMutableData 对象中，然后对其进行解析。这部分代码唯一的复杂之处在于 NSInputStream 的 read:maxLength: 调用；不过，基本概念类似于之前将字节读取到缓存中的示例。NSStream 是个轻量级且强大的对象，这都是因为背后那些优秀的基础类在起作用。

代码清单 8-7 示例 NSStreamDelegate 实现(LLNNSStreamController.m)

```
- (void)stream:(NSStream *)stream handleEvent:(NSStreamEvent)eventCode {
    switch(eventCode) {
       case NSStreamEventHasBytesAvailable: {
          if(receivedData == nil) {
             receivedData = [[NSMutableDataalloc] init];
          }

          uint8_tbuf[1024];
          int numBytesRead = [(NSInputStream *)stream read:buf
                                                 maxLength:1024];

          if(numBytesRead> 0) {
             [receivedData appendBytes:(const void *)buf length:
                                          numBytesRead];

          } else if(numBytesRead == 0) {
             NSLog(@"End of stream reached");
```

```
        } else {
            NSLog(@"Read error occurred");
        }

        break;
    }
        case NSStreamEventErrorOccurred: {
            NSError *error = [stream streamError];
        NSLog(@"Failed while reading stream; error '%@' (code %d)",
            error.localizedDescription, error.code);

        if([self.delegate respondsToSelector:
                @selector(networkingResultsDidFail:)]) {
            [self.delegate networkingResultsDidFail:
                @"An unexpected error occurred while reading from
                the warehouse server."];
        }

        [self cleanUpStream:stream];
    }

    case NSStreamEventEndEncountered: {
        NSString *resultsString = [[NSStringalloc]
                                            initWithData:receivedData
            encoding:NSUTF8StringEncoding];
        NSLog(@"Received string: '%@'", resultsString);

        LLNNetworkingResult *result = [self
                                    parseResultString:resultsString];

        if(result != nil) {
            if([self.delegate respondsToSelector:
                    @selector(networkingResultsDidLoad:)]) {
                [self.delegate networkingResultsDidLoad:result];
            }
        } else {
            if([self.delegate respondsToSelector:
                    @selector(networkingResultsDidFail:)]) {

                [self.delegate networkingResultsDidFail:
                    @"Unable to parse the response from the warehouse
                    server."];
            }
        }

        [self cleanUpStream:stream];

        break;
    }
    default:
```

```
            break;
        }
    }

- (void)cleanUpStream:(NSStream*)stream {
    [stream removeFromRunLoop:[NSRunLoop currentRunLoop]
                      forMode:NSDefaultRunLoopMode];
    [stream close];

    stream = nil;
}
```

8.4 小结

本章介绍了 iOS 中的底层网络框架，并谈到了每个框架的优缺点。框架层次越往上，代码就会越短且越简单，不过你却失去了一些能力，因为每个抽象层都隐藏了实际执行通信的原始网络 Socket。仓库服务器示例提供了一种简单的方式来为应用确定恰当的框架，因为每种实现都可以直接从复杂性、开发时间以及易用性方面进行对比。

第 9 章

测试与操纵网络流量

本章内容

- 观测网络流量
- 通过代理操纵网络流量
- 模拟实际的网络状况

本章代码下载

可以在 www.wrox.com 的 Download Code 选项卡中下载本章代码，地址是 www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html。代码位于第 9 章的下载压缩包中，并且根据章节名字单独命名。

你会出错，事实上这个世界上每天都有人会出错。由于你和其他人会出错，因此你所编写的软件也可能出错。正因为这一点，对软件行为进行诊断、找出错误、纠正错误，然后避免将来再出错是非常必要的。

在编写网络应用时，系统的组件之间会存在着很多软件与硬件层。我们有必要检测这些组件之间的交互，从高层次上了解系统所发生的事情的精确全貌，这样才能更好地检测任何已有的错误。

本章将会介绍观测这些网络交互、操纵它们以及模拟实际状况的方法。通过观测网络交互，可以精确地了解设备发出了哪些请求，接收了哪些数据。通过操纵网络流量，可以在开发环境中创建出只有用户在使用应用时才会出现的状况。模拟网络状况还可以验证在各种网络状况下应用的行为。

9.1　观测网络流量

在编写网络应用时，无法完全控制设备发出或接收的数据包。设备上的代码与远程服务器上的代码之间存在着很多软件与硬件层。由于代码与底层网络层之间的交互的文档语焉不详或是难以理解，因此有时需要检查设备发出的原始数据，这样才能确定应用发出的到底是什么。与之类似，服务器与应用之间也存在着类似的层次，有时需要知道从服务器接收的精确的数据。比如，如果应用向请求添加了 HTTP 头，那么你需要确切地知道 iOS 是如何进一步处理这些头的以及最终发给服务器的到底是什么。

观测网络流量的行为叫做嗅探或数据包分析。数据包分析器从网络早期就已经存在了，并且可以用于几乎每一种类型的物理互联与协议。本节将会介绍如何在运行着 OS X Lion 的开发系统中嗅探使用了 TCP 传输与 HTTP 应用协议的以太网与 Wi-Fi 连接。

9.1.1　嗅探硬件

在从 iOS 设备开始捕获网络流量前，首先要有能够方便于数据包捕获的网络拓扑。运行着捕获软件的电脑必须与 Wi-Fi 设备位于同一网络，或是来自 Wi-Fi 网络的数据包要能传播到捕获电脑。因此，需要做一些硬件方面的编排以从设备捕获网络流量。

说明：

从 iOS 模拟器捕获数据包不需要做特别的硬件或网络配置。如果需要捕获这些数据包，那么可以使用下一节列出的嗅探软件来监听回送设备(lo0)或是用于连接网络的接口。

图 9-1 展示了推荐的配置——使用了嗅探电脑与 iOS 设备共享的 Wi-Fi 网络。

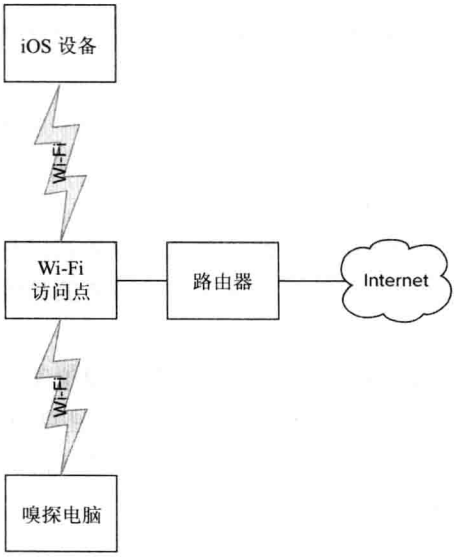


图 9-1

在常见的 Wi-Fi 配置中，iOS 设备和嗅探电脑与 Wi-Fi 访问点配对使用。嗅探电脑上的 Wi-Fi 适配器会接收 Wi-Fi 网络上传输的所有网络数据包。我们需要配置嗅探软件，使之运行在监控模式下，这样底层 Wi-Fi 驱动器就会将所有数据包传播到嗅探软件。在某些拥有高 Wi-Fi 覆盖率的企业网络中，移动设备与嗅探电脑可能会连接到不同信道上的不同访问点。甚至在设备与电脑离得很近时这种情况也有可能发生，因为两个系统有不同的规则控制着首选哪些访问点以及如何切换访问点。如果设备与电脑连接到了不同的网络，那么在电脑上可能就看不到设备发出的 Wi-Fi 流量。

图 9-2 展示了另一种配置——利用 OS X 的 Internet 共享特性将嗅探电脑作为 Wi-Fi 访问点。

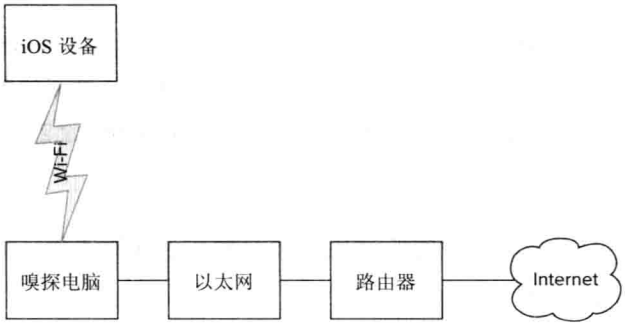


图 9-2

在这种配置下，需要将嗅探电脑连接到有线以太网，并在 OS X 中开启 Internet 共享来与 Wi-Fi 用户共享以太网。接下来需要配置 iOS 设备以加入嗅探电脑创建的 Wi-Fi 网络。从设备向 Internet 发出的每个网络数据包都会经过 Wi-Fi 与电脑的以太网接口。在这种配置下，可以观测到客户端到服务器的流量，还可以观测到点对点流量，比如由 Game Kit 产生的流量等。

9.1.2　嗅探软件

有很多网络嗅探器可用于 OS X；免费的有 Wireshark(稍后介绍)与 Packet Peeper(参见 http://sourceforge.net/projects/packetpeeper/)，还有一些共享软件，如 Cocoa Packet Analyzer (http://www.tastycocoabytes.com/cpa)等。

OS X 自带命令行数据包嗅探器 tcpdump，它是 OS X 中大多数其他嗅探器的根基。tcpdump 可以捕获来自于接口的网络流量，根据指定的标准进行过滤，显示流量，并将流量追踪信息保存到日志文件中。tcpdump 的流量显示是很难理解的，不过很多数据包分析器都使用了它的过滤语法，因此还是有必要学习一下。

1. 使用 tcpdump 进行捕获

网络工程师对嗅探的需求与应用开发者之间存在着显著的区别。网络工程师需要看到设备传输的每一位，这种细节层次模糊了对于应用开发者来说最为重要的 HTTP 请求数据。本节将会介绍应用开发者在分析应用产生的网络流量时最常使用的过滤器。

如果从命令行运行 tcpdump 而且不使用过滤器，那么你会看到大量的数据包。这会将你想要观测的流量淹没掉，即应用与服务器之间传输的流量。

为了避免这种情况发生，你应该使用过滤。最基本的过滤器是根据主机进行过滤，这样 tcpdump 就会忽略掉除了与指定主机传输的其他所有流量。如下代码片段展示了过滤器 tcpdump 命令，它会忽略掉除与主机地址 192.168.1.50 之间传输的其他所有流量。

```
sudo tcpdump host 192.168.1.50
```

说明：

sudo 命令会提示输入密码，你应该输入当前登录账号的密码。

host 过滤器可以使用 IP 地址、DNS 主机或域名。如果指定了 DNS 域，那么发往该域中主机的任何流量都会被捕获到。你还可以通过逻辑运算符包含其他的标准过滤来自于多个主机的流量。如下代码会捕获发送给主机 192.168.1.50 与 10.1.2.25 的流量：

```
sudo tcpdump host 192.168.1.50 or 10.1.2.25
```

取决于网络配置的不同，有时需要过滤发送给整个子网的流量。如下代码片段会过滤发送给 192.168.1 子网的流量：

```
sudotcpdump net 192.168.1.0/24
```

另一常见的做法是根据连接所用的 TCP 或 UDP 端口号来过滤数据包。tcpdump 使用 /etc/services 中定义的名字来将 TCP 服务端口名映射到端口号。如下代码展示了用于过滤 HTTP 流量的两个等价的命令：

```
sudo tcpdump port http
sudo tcpdump port 80
```

值 http 用于查找端口号，而不是检测连接所用的协议。因此，如果在 TCP 连接上所用的 HTTP 协议使用了非 80 的端口号，那么上述过滤器就无法捕获流量了。

可以通过逻辑运算符组合过滤条件来捕获特定主机与端口的流量。如下代码片段只会捕获主机 192.168.1.50 上端口 80 与 8080 的数据包：

```
sudotcpdump host 192.168.1.50 and \(port 80 or port 443\)
```

借助 and 与 or 运算符，再搭配上分开的圆括号，可以对捕获的数据进行过滤，去掉可能会影响结果的不相关的流量。过滤表达式中的反斜杠(\)可以防止命令 shell 解释圆括号。在图形程序中使用 tcpdump 过滤器语法时可以省略掉这些反斜杠字符。相反，请小心不要过度进行过滤而删除重要的数据。

tcpdump 可用于执行扩展的网络流量捕获，并将它们保存到捕获文件中以供事后分析。稍后的“2. 使用 Wireshark 进行捕获”将要介绍的图形化嗅探器 Wireshark 可以导入已有的捕获文件，这样就可以通过友好的界面深入数据中了。如下代码片段会捕获 80 端口的所

有流量并将其保存到名为 http-capture.trace 的文件中：

```
sudo tcpdump -s 1514 port 80 -w http-capture.trace
```

–s 1514 参数告诉 tcpdump 捕获数据包的前 1514 个字节。这会捕获大多数常见的以太网包大小；然而，如果网络使用了较大的数据包，那么你可能需要增大这个值。

2. 使用 Wireshark 进行捕获

Wireshark(http://www.wireshark.org)是个免费、跨平台的图形化网络嗅探器，可在大多数主流操作系统中使用。它可以捕获新的流量，也可以分析之前使用 tcpdump 捕获的流量。在 OS X 中，Wireshark 需要安装 X11。安装过程中会修改很多设备文件的权限，这样安装的用户就可以从那些设备上捕获数据了，无需超级用户权限。

启动 Wireshark 后，你会看到一个帮助起始页，如图 9-3 所示，可以在这里直接开始追踪某个接口、设置追踪选项，或是打开已有的追踪文件。

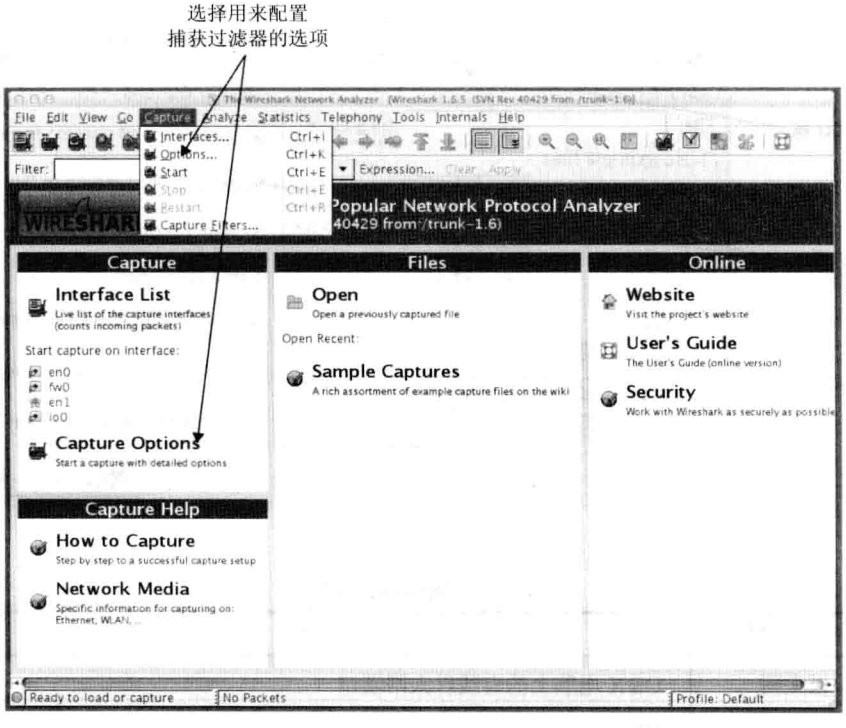


图 9-3

如果直接在某个接口上开始追踪，你会看到大量的数据包，这可能没什么用。如果通过 Capture->Options 菜单命令开始追踪，那就可以对追踪进行调整以满足需求。

说明：

由于 Wireshark 是个非常强大的工具，因此它提供了作为一名开发者可能永远也用不上的功能；请不要让这一点影响你尝试这个应用以探索有益的特性。如果因为配置问题导致无法使用，请关闭应用并重启；什么都不会被破坏。

使用如图 9-4 所示的 Capture Options 对话框，可以选择捕获的接口。表 9-1 列出了 3 个常见的网络接口。

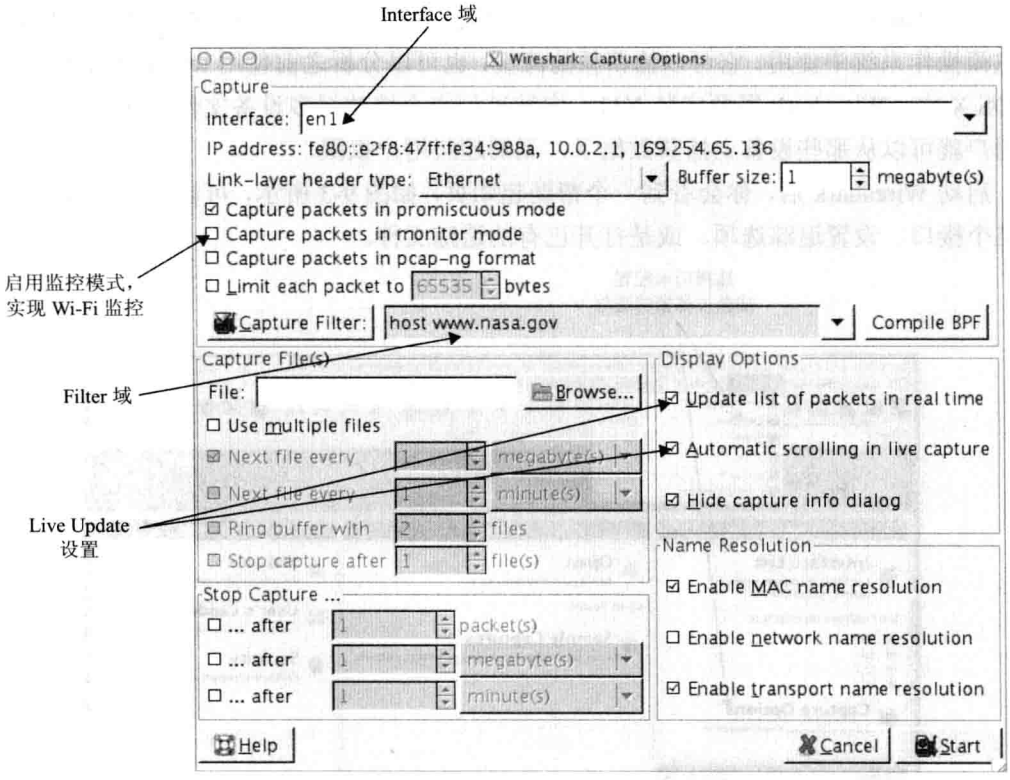


图 9-4

表 9-1 网络接口说明

接口名	接口类型	说明
en0	以太网	OS X 电脑上的有线以太网端口
en1	Wi-Fi	OS X 电脑上默认的 Airport 接口
lo0	回路	程序连接到 localhost 或 127.0.0.1 时所用的本地回路接口

通过 capture filter 域指定待捕获的数据包，使用与 tcpdump 命令相同的语法。探测器只会捕获进入到主机 www.nasa.gov 的数据包(如图 9-4 所示)。

如果通过常见的 Wi-Fi 拓扑(如图 9-1 所示)嗅探应用的数据包，那么需要启用监控模式。监控模式会配置 Wi-Fi 适配器，将 Wi-Fi 适配器接收到的所有网络数据包传递给软件驱动。如果不启用该选项，那么没有发送给嗅探电脑的数据包就会被 Wi-Fi 硬件忽略。这取决于无线网络配置、操作系统版本以及 Wireshark 的版本。Wireshark 可用于 OS X、Linux 以及 Windows 平台。每个平台提供的功能都有所不同，因此如果想得到网络的完美配置，可以使用不同的操作系统。

live update 设置告诉 Wireshark 当捕获到数据包时就持续更新显示内容。如果已经正确指定好过滤器并能轻松检测何时捕获到足够的数据，那么该特性将非常有用；不过，在速度较慢的机器上，这可能会影响应用的性能，导致丢包。单击对话框中的 Start 按钮会启动捕获会话。

启用 live updating 后，当捕获到数据包后，你会在 Captured Packet 列表中看到包的头部信息。图 9-5 展示了在 iPhone 上运行第 3 章“构建请求”中介绍的 VideoDownloader 应用时产生的网络数据包。

Packet Decomposition 面板会显示在 Captured Packets 列表中选择的数据包的每个协议层的值。该数据包会被分为 3 层：Ethernet II、Internet Protocol 及 Transmission Control Protocol(如图 9-5 所示)。

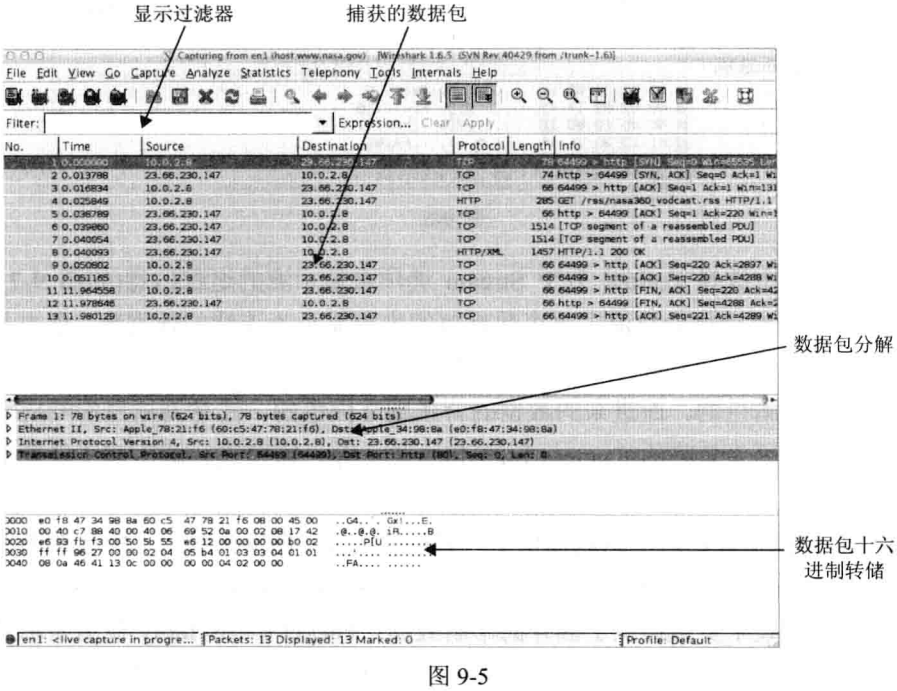


图 9-5

Packet Hex Dump 区域会显示所选数据包的每个字节的十六进制值以及相应的 ASCII 值。如果在 Packet Decomposition 面板中进入到协议层，那么数据包中所选的那部分就会在十六进制转储中高亮显示。

Wireshark 有个非常有用的特性，就是追踪 TCP 流的能力。在中等活跃程度的机器上的一个典型数据包中，你会看到多个 TCP 流与 HTTP 会话同时出现，它们的数据包也交织在一起。为了追踪 TCP 流，请按住 Ctrl 键并单击流中的一个数据包以激活数据包菜单，如图 9-6 所示。

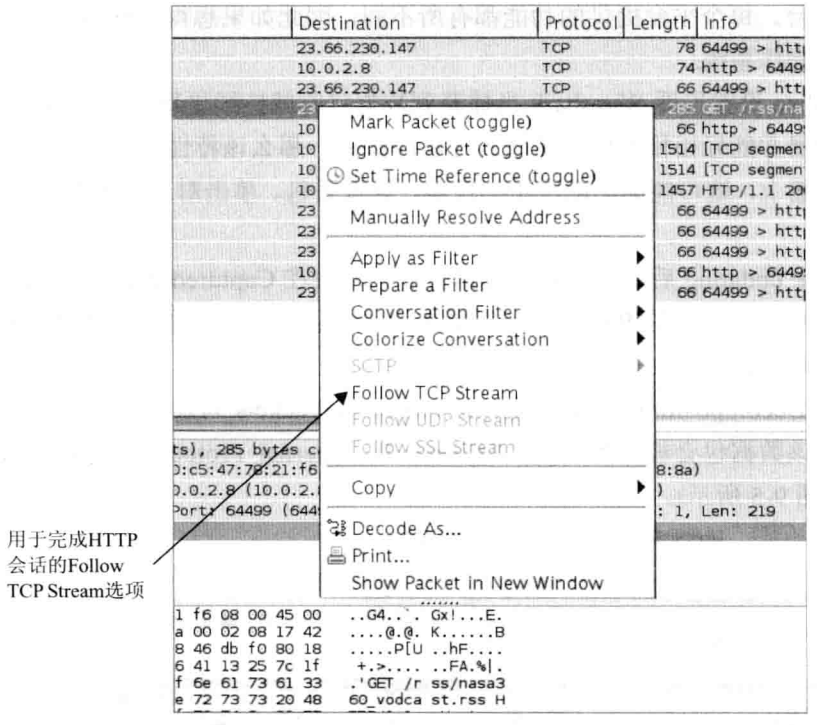


图 9-6

从数据包菜单中选择 Follow TCP Stream 选项来显示流的追踪信息(如图 9-7 所示)。Follow TCP Stream 对话框会将流中每个数据包的内容显示为连续的数据集。出去与进入的数据使用不同的阴影表示，这样就可以轻松分辨出数据包的流向。在图 9-7 中，从手机发出的数据的背景是暗色的，而进来的数据的背景则是白色的。注意到这里的响应数据看起来有些混乱，这是因为负载的 Content-Encoding 是 gzip，需要进行解压缩才能看到实际的内容。

Wireshark 为压缩数据这个问题提供了一种解决方案。图 9-8 展示了所选的第 8 个数据包的内容。Wireshark 已经推断出该数据包是对 HTTP 请求的响应，并且响应体中是 XML 数据。因此，它对整个负载提供了额外的诊断数据。Packet Reassembly 视图显示了包含该负载的后续网络数据包的所有响应负载。如果 HTTP 响应是压缩的，那么 Wireshark 就会对其进行解压缩。Uncompressed Entity Body 视图会显示出解压缩后响应的内容。


Follow TCP Stream
Stream Content
GET /rss/nasa360_vodcast.rss HTTP/1.1
Host: www.nasa.gov
User-Agent: VideoDownloader/1.0 CFNetwork/548.1.4 Darwin/11.0.0
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: keep-alive
HTTP/1.1 200 OK
Server: Apache
Last-Modified: Sat, 28 Apr 2012 04:55:57 GMT
ETag: "16840c0-656d-4beb609591140"
Accept-Ranges: bytes
Content-Encoding: gzip
Content-Length: 3983
Content-Type: application/rss+xml
Date: Sat, 28 Apr 2012 17:28:29 GMT
Connection: keep-alive
Vary: Accept-Encoding
Entire conversation (4506 bytes)
Find
Save As
Print
ASCII
EBCDIC
Hex Dump
C Arrays
Raw
Help
Filter Out This Stream
Close

图 9-7

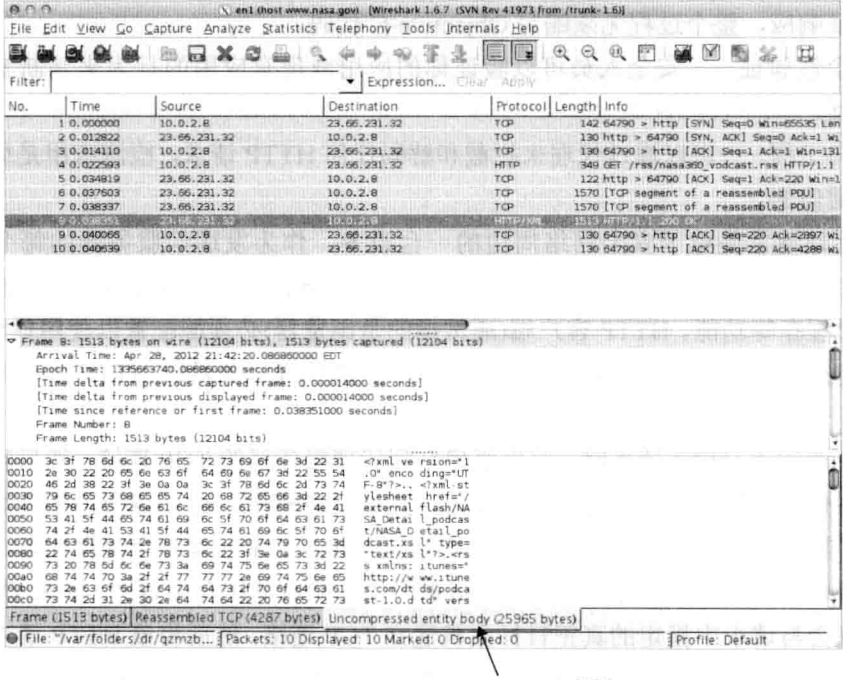

en1 (host www.nasa.gov) [Wireshark 1.6.7 (SVN Rev 41973 from /trunk-1.6)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply
No. Time Source Destination Protocol Length Info
1 0.000000 10.0.2.8 23.66.231.32 TCP 142 64790 > http [SYN] Seq=0 Win=65535 Len
2 0.012822 23.66.231.32 10.0.2.8 TCP 130 http > 64790 [SYN, ACK] Seq=0 Ack=1 Wi
3 0.014110 10.0.2.8 23.66.231.32 TCP 130 64790 > http [ACK] Seq=1 Ack=1 Win=131
4 0.022593 10.0.2.8 23.66.231.32 HTTP 349 GET /rss/nasa360_vodcast.rss HTTP/1.1
5 0.034819 23.66.231.32 10.0.2.8 TCP 122 http > 64790 [ACK] Seq=1 Ack=220 Win=1
6 0.037603 23.66.231.32 10.0.2.8 TCP 1570 [TCP segment of a reassembled PDU]
7 0.038337 23.66.231.32 10.0.2.8 TCP 1570 [TCP segment of a reassembled PDU]
8 0.038351 23.66.231.32 10.0.2.8 HTTP/XML 1513 HTTP/1.1 200 OK
9 0.040066 10.0.2.8 23.66.231.32 TCP 130 64790 > http [ACK] Seq=220 Ack=2897 Wi
10 0.040639 10.0.2.8 23.66.231.32 TCP 130 64790 > http [ACK] Seq=220 Ack=4288 Wi
Frame 8: 1513 bytes on wire (12104 bits), 1513 bytes captured (12104 bits)
Arrival Time: Apr 28, 2012 21:42:20.080860000 EDT
Epoch Time: 1335663740.080860000 seconds
[Time delta from previous captured frame: 0.000014000 seconds]
[Time delta from previous displayed frame: 0.000014000 seconds]
[Time since reference or first frame: 0.038351000 seconds]
Frame Number: 8
Frame Length: 1513 bytes (12104 bits)
0000 3c 3f 78 6d 6c 20 76 65 72 73 69 6f 6e 3d 22 31 <?xml ve rsion="1
0010 2e 30 22 20 65 6e 63 6f 64 69 6e 67 3d 22 55 54 .0" enco ding="UT
0020 46 2d 38 22 3f 3e 0a 0a 3c 3f 78 6d 6c 2d 73 74 F-8"?>.. <?xml-st
0030 79 6c 65 73 68 65 65 74 20 68 72 65 66 3d 22 2f ylesheet href="/
0040 65 78 74 65 72 6e 61 6c 66 6c 61 73 68 2f 4e 41 external flash/NA
0050 53 41 5f 44 65 74 61 69 6c 5f 70 6f 64 63 61 73 SA_Detai l_podcas
0060 74 2f 4e 41 53 41 5f 44 65 74 61 69 6c 5f 70 6f t/NASA_D etail_po
0070 64 63 61 73 74 2e 78 73 6c 22 20 74 79 70 65 3d dcast.xs l" type=
0080 22 74 65 78 74 2f 78 73 6c 22 3f 3e 0a 3c 72 73 "text/xs l"?>.<rs
0090 73 20 78 6d 6c 6e 73 3a 69 74 75 6e 65 73 3d 22 s xmlns: itunes="
00a0 68 74 74 70 3a 2f 2f 77 77 77 2e 69 74 75 6e 65 http://w ww.itune
00b0 73 2e 63 6f 6d 2f 64 74 64 73 2f 70 6f 64 63 61 s.com/dt ds/podca
00c0 73 74 2d 31 2e 30 2e 64 74 64 22 20 76 65 72 73 st-1.0.d td" vers
Frame (1513 bytes) Reassembled TCP (4287 bytes) Uncompressed entity body (25965 bytes)
File: "/var/folders/dr/qzmzb... Packets: 10 Displayed: 10 Marked: 0 Dropped: 0
Profile: Default
Packet Reassembly 视图

图 9-8

Wireshark 是个强大的工具，并且应该是每个网络开发者工具箱中的必备工具。我们可以通过模块(叫做解析器)对其进行扩展，这些解析器能够解析众多业界标准协议，不过也可以开发自己的解析器来解析自定义的协议。如果通过源代码进行构建，那么 Wireshark 可以在某些情况下解密 SSL 连接。借助于 Wireshark，可以查看任意网络协议下由设备发出的每一位和每一个数据包。如果使用的是自定义协议或是非 HTTP 通信，那么这将非常有用。下一节将会介绍如何在开发环境下使用一种更为简单，不过功能上没有那么强大的方式来捕获和解密网络流量。

9.2 操纵网络流量

网络数据包捕获工具提供了一种方式来监测网络流量，不过有时需要做的不仅仅是监测。幸好，有一个常用的网络组件(一个 HTTP 代理)，开发者可以利用它操纵 HTTP 与 HTTPS 请求。HTTP 操纵的一些使用场景如下所示：

- 错误模拟——可以拦截响应，将响应的状态码改为错误状态；还可以修改负载，让其表示为错误。这样就可以在不实际触发错误的情况下测试应用如何响应服务器的错误。
- 未来状态模拟——如果知道服务器的未来版本会提供与当前服务器不同的响应，那么可以针对改变的协议测试应用的老版本。
- 服务器验证——可以修改发送给服务器的请求，根据设备上可能难以复制的数据来验证响应，整个过程无须编写 Objective-C 代码。
- 安全性验证——安全人员可以验证你的应用或现有应用的行为来判断安全的网络交互。
- 逆向工程——可以通过代理来拦截和解码任何 HTTP 请求，检测应用是如何与服务器通信的。

网络代理通常是位于安全网络周边的一台设备，作为发送给服务器的每个请求的中介。很多大型企业都会部署网络代理为网络中的个人电脑提供内容过滤、访问控制、病毒防护与响应缓存等功能。HTTP 客户端(通常指的是浏览器)必须能配置以使用代理，否则将无法访问防护网络之外的服务器。图 9-9 展示了穿越代理服务器的 HTTP 请求与响应的活动序列图。

当客户端创建 HTTP 请求时，首先要建立到代理服务器的 TCP 连接，然后将 HTTP 请求发送给该代理。接下来，代理会对该请求应用其处理规则，这可能会拒绝该请求，因为请求访问了黑名单站点或是请求本身有未授权的内容。如果代理是个缓存代理，那么它可能会短路请求，将由之前访问同一目标的请求产生的响应从缓存中返回。如果请求可接受，那么代理就会与请求中指定的真正目标主机建立 TCP 连接，然后将该 HTTP 请求发送给目标主机。如果目标主机响应了，那么 HTTP 响应会被代理接收，然后再应用额外的处理规则。比如，可能会扫描响应以检查病毒或是将其缓存起来。如果响应可接受，那么代理就

会将其回传给请求客户端。

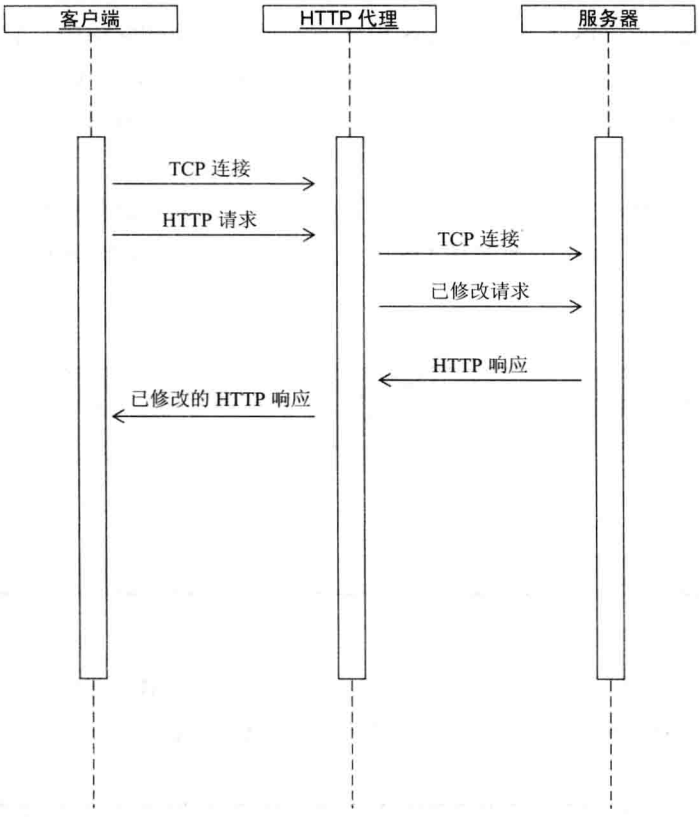


图 9-9

代理能以几种不同的方式来处理 HTTPS 请求。首先，代理可以拒绝所有的 HTTPS 请求，不过这种行为并不常见。此外，有些代理配置并不对 HTTPS 请求进行任何修改和过滤。第 3 种行为就是代理会与客户端建立 HTTPS 连接，并提供貌似来自于目标主机的 SSL 证书，不过却是用代理的认证授权(Certificate Authority，CA)证书进行的签名。代理本质上会对请求进行中间人攻击，但却不是出于不法目的，攻击的目的旨在保护企业。要想让这种方式能够运作，客户端需要在其钥匙串中安装好代理的 CA 证书。如果请求得到了许可，那么代理就会与目标主机建立 SSL 连接，然后安全地发送请求。

对于网络工程师来说，有很多代理软件和硬件包可供使用。这些包是针对网络工程师而设计的，对于应用开发者来说用处有限。Charles(http://www.charlesproxy.com)是一款针对开发者的价位适中的代理软件包。它是个桌面应用，可以针对任何 HTTP 客户端执行代理服务。开发者可以通过 Charles 手工拦截请求并修改请求和响应。我们还可以配置 Charles 来自动化地对请求进行相同类型的修改。

如果使用 HTTP 代理来操纵网络流量，那么只能操纵 HTTP 与 HTTPS 请求。如果应用使用了其他协议，那么代理就没什么用处了。

接下来将会介绍如何通过 Charles 来拦截与操纵 HTTP 请求和响应。

9.2.1 配置 Charles

要想使用 Charles，OS X 电脑与 iOS 设备必须处于相同的网络中。它们无需位于同一子网中，不过一台设备要能 ping 通另一台设备。iOS 设备必须处于 Wi-Fi 网络中，因为 iOS 并不会对 WWAN 连接使用代理设置。要想设置 Charles 以通过代理来捕获流量，请按照如下步骤进行：

(1) 在运行 Charles 时，默认情况下会在 OS X 机器上配置好代理设置。该行为会干扰从设备捕获到的数据，因此需要禁用 OS X 代理。为此，请选择 Proxy Settings 菜单，如图 9-10 所示。

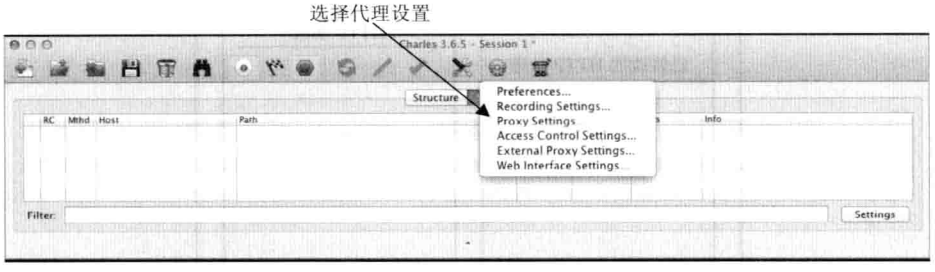


图 9-10

(2) 这时会弹出代理设置对话框，如图 9-11 所示。选择 Mac OS X 选项卡，不要选中该选项卡中的任何复选框；然后单击 OK 按钮。接下来，Charles 会从所有活动的网络接口中删除代理配置。

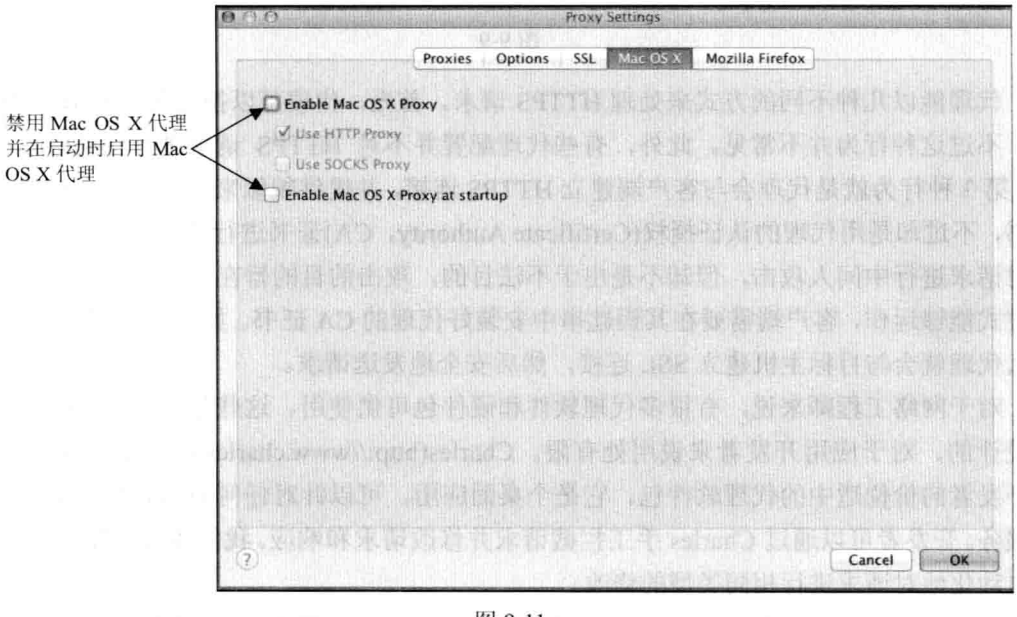


图 9-11

(3) 接下来需要配置 iOS 设备，使得所有的 HTTP 流量都通过代理。首先，确定好运行 Charles 的机器的 IP 地址。在该机器上，打开一个 Terminal 窗口，然后执行 ifconfig 命

令。如下代码片段展示了执行该命令后的部分输出：

```
$ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST>mtu 16384
...
en1: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST>mtu 1500
        ether e0:f8:47:34:98:8a
        inet6 fe80::e2f8:47ff:fe34:988a%en1 prefixlen 64 scopeid 0x5
        inet 192.168.1.34 netmask 0xffffff00 broadcast 192.168.1.255
        media: autoselect
        status: active
```

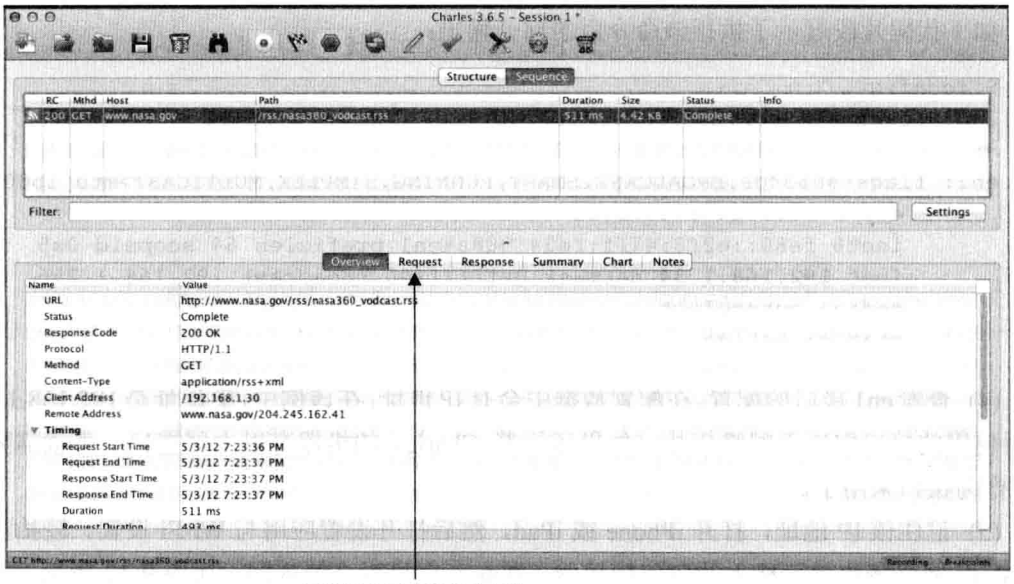
(4) 查看 en1 接口的配置。在配置数据中会有 IP 地址；在该例中，IP 地址是 192.168.1.34。你所使用的接口取决于网络拓扑。如果运行着 Charles 的机器有以太网接口，那就需要将 en0 作为接口来用了。

(5) 记住该 IP 地址，打开 iPhone 或 iPad，然后打开设置应用与 Wi-Fi 设置。轻拍活动的 Wi-Fi 网络表中单元格上的蓝色指示器。在底部会显示详细视图，如图 9-12 所示，视图中有个分隔控件，可以手工配置 HTTP 代理。

图 9-12

(6) 选择 Manual 选项，输入之前在服务器域中获得的 IP 地址。Charles 的默认端口是 8888。单击视图中的后退按钮，代理设置即可生效。如果在手机上应用这些设置，那么会阻塞所有的 HTTP 流量，除非 Charles 运行在指定的地址，因此在调试完成后请不要忘记将这些设置修改回去。

该例使用了第 3 章中的 VideoDownloader 应用，它会从 NASA 下载一个 RSS 种子。配置好代理，当运行并开始记录数据时，代理会捕获单个 HTTP 事务。Charles 会将这些事务分组到单独一行。如果选中某个事务，然后选择事务列表中的 Structure 选项卡，那么可以深挖请求并查看请求与响应的细节信息。图 9-13 展示了由代理捕获的一个应用请求的信息。



查看请求/响应的各个部分

图 9-13

9.2.2　HTTP 断点

可以通过 Charles 在 URL 上设置断点，这样就可以拦截并修改请求和响应了。可以在多个不同的 URL 上设置多个断点。要想设置断点，请按照如下步骤操作：

(1) 按住 Ctrl 键并从事务列表中单击某个事务，该事务要与你想要捕获的事务的 URL 相匹配。弹出的菜单(如图 9-14 所示)中包含了在该 URL 上设置断点的选项。

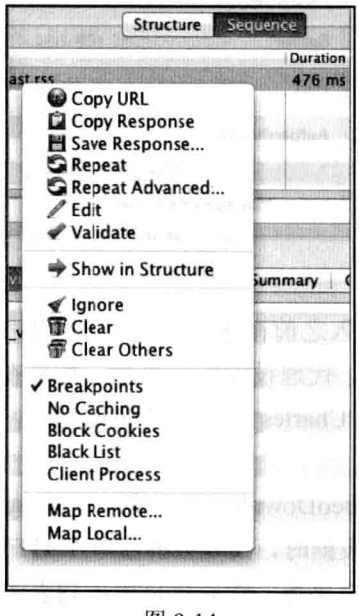


图 9-14

(2) 选择 Breakpoints 选项以在该 URL 上开启断点。在对 VideoDownloader 应用随后的调用中，Charles 会捕获到请求并显示出断点窗口，如图 9-15 所示。

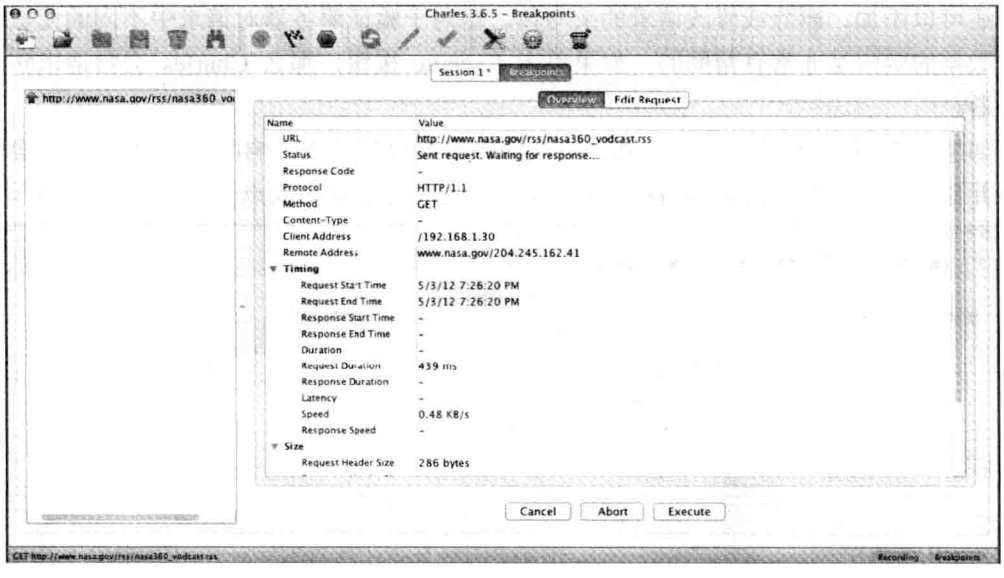


图 9-15

请求的 Overview 面板会显示出关于提取请求的概要信息。其中有些值是未定义的，这是因为现在还没有收到响应。如果选择 Edit Request 面板，那就会看到设备发出的请求的精确内容。在 Edit Request 面板中(如图 9-16 所示)，可以手工操纵请求的内容。

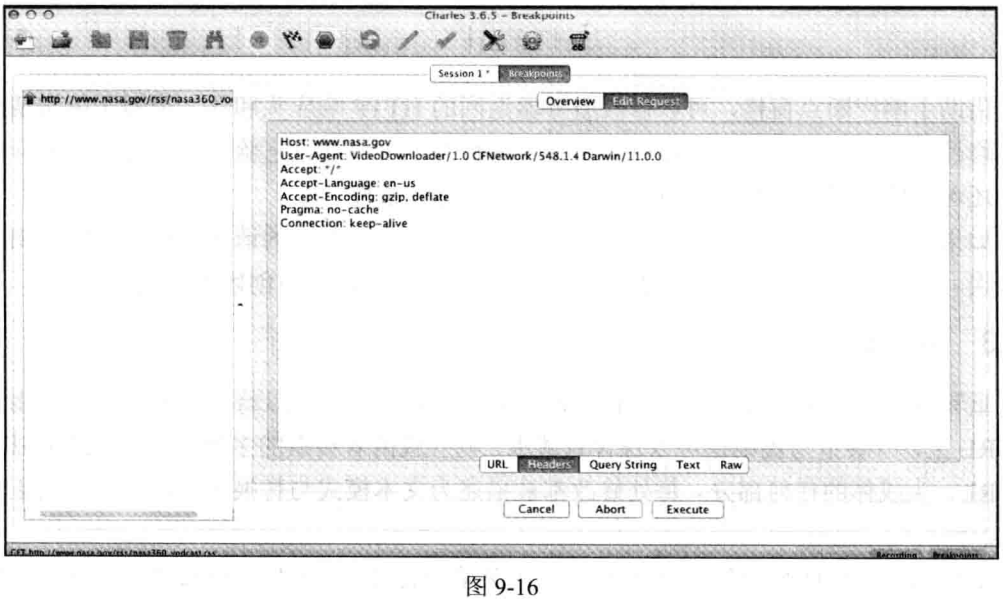


图 9-16

可以通过 Edit Request 面板修改 URL、头、查询字符串以及请求体的文本内容，还可以修改请求的原始字节信息。如果修改 URL，那么 Charles 会将请求发送给不同的 URL 而

不是由应用指定的 URL。如果使用该功能，那么可以将单个请求发送给测试服务器，从而在将新的服务器部署到生产前验证应用的老版本在新服务器上的行为。

还可以添加、删除或修改请求的头与体。这对于验证服务器对请求中不同的元数据或负载数据的响应是非常有帮助的。如果单击 Execute 按钮，那么 Charles 会将请求转发给服务器。

如果在 URL 上设置了断点，那么当从服务器接收到请求的响应后，Charles 还会中断事务并显示出用于编辑请求的窗格，如图 9-17 所示。对于每个断点来说，你会被中断两次。

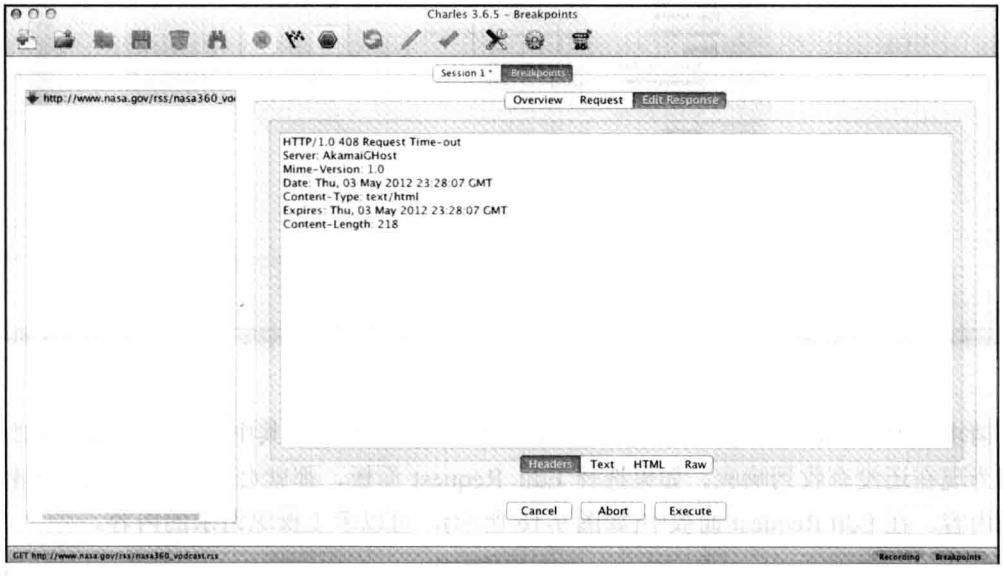


图 9-17

借助于响应断点窗格，可以修改服务器返回的 HTTP 响应头和体。类似于编辑请求，可以修改响应的所有内容。这是个非常强大的特性，可以通过它验证应用对错误响应的回应，还可以模拟通常情况下在服务层难以重现的边界条件。

如果要测试的应用的请求超时值很短，那么拦截带有断点的请求就会导致应用出现超时错误。需要快速地手工编辑请求或响应，也可以在 Charles 中创建重写规则。

9.2.3 重写规则

重写规则指定 Charles 自动对 HTTP 事务所做的修改。可以将一条重写规则应用到一组 URL 上，一条重写规则也可以包含对请求、响应或两者要做的多处修改。修改可以应用到 URL、头或体的任何部分。每处修改都被描述为文本模式与替换文本，可以使用正则表达式。

要想添加重写规则，请从 Charles 的应用菜单中选择 Tools➪Rewrite 菜单选项。图 9-18 展示了重写设置(Rewrite Settings)窗口。

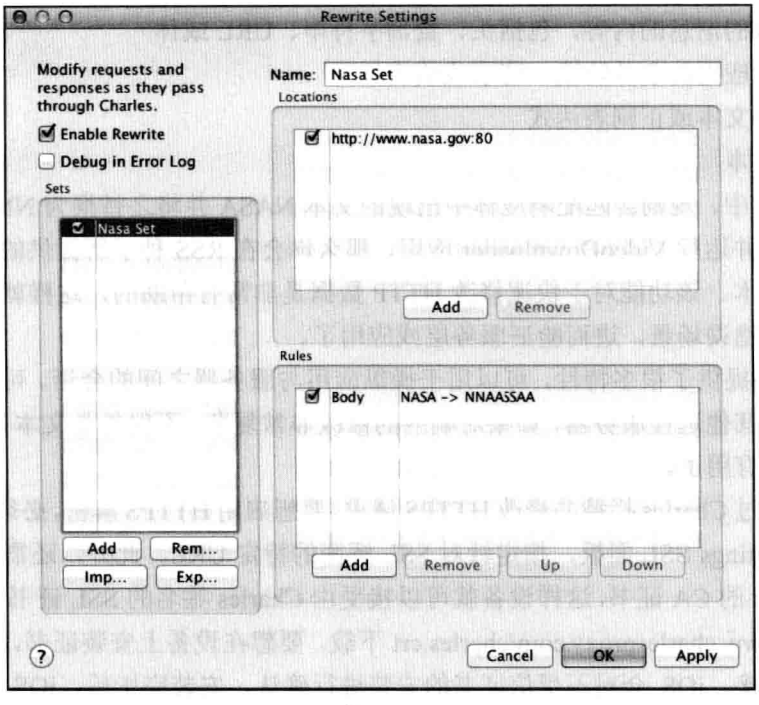


图 9-18

可以在该窗口中添加新的规则集、向规则集中添加目标 URL，还可以创建新的规则。每条规则都描述了对匹配 URL 的请求和响应要执行的文本处理。要想添加规则，请单击规则下的 Add 按钮。图 9-19 展示的重写规则定义对话框可以定义如下内容：

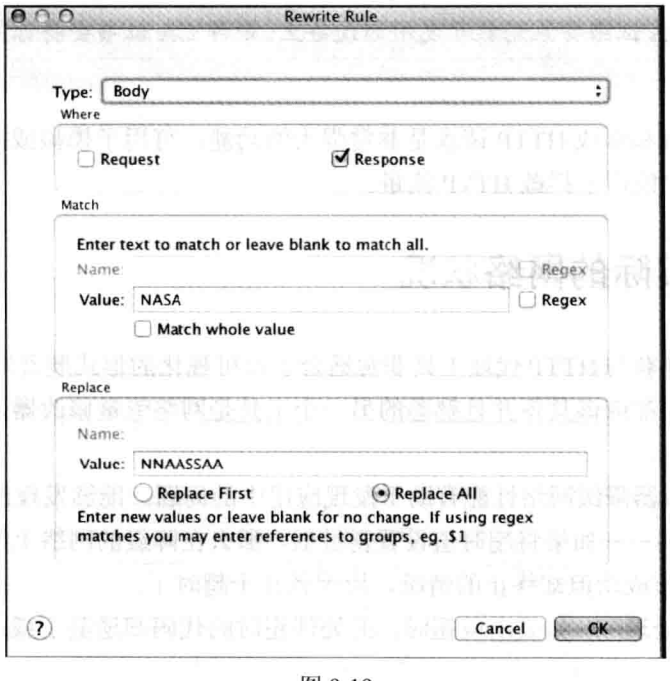


图 9-19

- 待修改的消息的内容，包括头、查询字符串、URL 或体
- 规则所应用的事务阶段：请求或响应
- 匹配的文本或正则表达式
- 替换文本

在图 9-19 中，规则会匹配响应体中出现的文本 NASA 并将之替换为 NNAASSAA。如果应用该规则并运行 VideoDownloader 应用，那么你会在 RSS 种子所提供的视频说明中看到修改过的文本。该功能对于快速修改 HTTP 数据是非常有帮助的，这样就可以模拟复杂的边界条件和错误场景，进而验证服务层或应用了。

Charles 还提供了很多特性，可以用于操纵应用与服务器之间的会话。可以将请求映射到本地文件或其他远程服务器，如果对响应的修改非常复杂，不仅仅是文本替换那么简单，该特性就非常有用了。

还可以通过 Charles 拦截并修改 HTTPS 请求。要想启用 HTTPS 解密，必须配置 Charles，使用 Proxy Settings SSL 面板，指定针对 SSL 解密的特定 URL。此外，还需要在 iOS 设备上安装 Charles 的 CA 证书，这样设备就可以接受由 Charles 签名的 SSL 证书了。Charles CA 证书可以从 www.charlesproxy.com/charles.crt 下载。要想在设备上安装证书，请在 iOS 设备上访问下载链接，iOS 会对不受信证书的安装进行确认。安装完毕后，iOS 设备就会接受由 Charles 签名的其他 SSL 证书。在配置 SSL 解密前，显示的事务内容是加密文本，这是不可读的。配置成功后，事务结果会显示为清晰、可读的文本。该特性对于观测来自于应用的 SSL 连接是非常有价值的。

警告：

不要将 Charles 证书安装到非开发用的设备上。欺诈 CA 证书会将你暴露给公共网络上的中间人攻击。

通过代理拦截和修改 HTTP 请求是非常强大的功能，可用于模拟或生成错误以验证应用的行为。代理只能用于拦截 HTTP 流量。

9.3 模拟实际的网络状况

网络数据包捕获与 HTTP 代理工具非常适合于以可视化的形式展现网络流量。所有 iOS 开发者的工具箱中都应该具备并且熟悉的另一个工具是网络流量修改器，它用于模拟低速或是不可靠的网络。

使用流量修改器降级网络性能有助于发现应用中的问题。能够发现的问题包括：

- 过短的超时——如果将超时值设置得过低，那么在降级的网络上测试应用会经常遇到本来应该成功但却终止的情况，原因就在于超时了。
- 遗漏错误处理——如果出现超时，但处理超时的代码却遗漏了或是有问题，那么在可能触发超时的网络上运行应用就会发现这些缺陷。

- 用户界面冻结——如果应用不小心在主线程上进行网络调用，那么在降级的网络上运行会有助于发现问题。
- 用户困惑——对于可用性测试来说，在低速的网络上运行应用能够发现应用没有冻结，但用户在等待响应时对应用正在做的事情感到困惑的情况。这些缺陷通常是通过用户界面的变化来解决的，通知用户应用正在执行的活动是什么。
- 缓存实现验证——在高速网络上运行应用难以量化数据缓存的改进到底有多少。借助于流量修改器，可以在实现缓存时量化用户期望的改进到底有多少。

OS X Lion 提供了一款名为 Network Link Conditioner(NLC)的优秀的流量修改器。NLC 会与 OS X 底层的网络接口驱动进行交互，并限制运行在机器上的每个接口的速度。它提供了自主修改带宽、延迟与接收和发送数据的丢包率的能力。NLC 还可以延迟 DNS 响应。

NLC 自带了几个与常见网络类型相匹配的预定义的网络配置文件，比如 Wi-Fi、3G 与 EDGE 网络。图 9-20 展示了良好的 3G 网络的配置文件。如果需要在网络上测试其他类型的行为，那么你还可以定义自己的配置文件。相较于实际的网络来说，通过预先配置的配置文件所得到的性能要好一些；因此，有必要使用比默认配置文件更慢一些的设置。

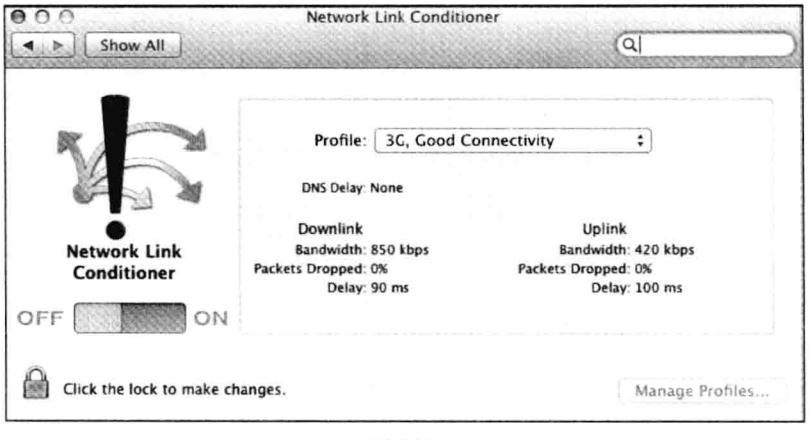


图 9-20

NLC 安装包是 Xcode 安装包的一部分，不过默认情况下是不会安装的。相反，目录/Applications/Utilities/Network Link Conditioner 中有首选项窗格。双击 Network Link Conditioner.prefPane 文件会在 OS X 系统首选项应用中安装首选项窗格。安装完毕后，可以通过系统首选项访问 NLC。

在使用 NLC 时，你首先要对其解锁，然后再打开。可以通过配置文件下拉列表选择打包好的或是自定义的网络配置文件来使用。可以通过 Manage Profiles 按钮来创建新的配置文件或是编辑打包好的配置文件参数。

要想让 NLC 能够反映出 iOS 设备的通信，需要在共享的网络拓扑中运行设备(如图 9-2 所示)；否则设备发出的网络数据包就会绕过 NLC。如果在 iOS 模拟器中调试应用，那么 NLC 会不断地改变网络流量。Xcode 使用本地回路网络接口来与运行在 iOS 模拟器中的应用进行通信，NLC 会修改通过每个网络接口的流量。因此，如果使用 NLC 中的低速配置

文件，那么模拟器中应用的启动时间就会变长。

虽然 NLC 对于模拟实际的网络状况非常有用，不过却不能替代实际情况下的测试。它无法复制实际的运营商网络的随机性，也无法复制非开发者用户的行为。可以使用 NLC 进行开发测试，但不要忘记在实际情况下测试你的应用。

9.4 小结

连接到企业的 iOS 应用涉及控制之外的很多网络组件的使用与协作。通过工具监测和操纵应用与企业基础设施之间的网络流量有助于发现并避免应用中出现的问题。网络嗅探器可以帮助查看设备与远程服务器之间传递的真实数据。可以通过 HTTP 代理操纵这些通信来模拟新的或是不完美的情况。网络流量修改器有助于观测到应用在不受控制或不完美的网络下的行为表现。

第10章

使用推送通知

本章内容

- 与本地通知交互
- 通过远程通知提供卓越的用户体验
- 在应用中使用通知最佳实践

本章代码下载

可以在www.wrox.com的Download Code选项卡中下载本章代码，地址是www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html。代码位于第10章的下载压缩包中，并且划分为如下几个主要示例：

- 一个Xcode项目，包含了本地与远程通知的代码
- 一个SQL脚本，用于构建远程通知所用的数据库
- PHP编写的服务器端脚本，用于管理远程通知

决定应用是否成功的一个关键指标是重复使用量。用户购买应用的花费可能很高，在吸引了用户后，如果有什么要引起用户注意的东西，就必须提供一种可预测、非侵入的方式来通知用户。

可以通过推送通知这种机制来告诉用户应用有新的信息。通知可以采取多种形式：企业应用可能会通知用户有一笔采购订单处于待审批状态，游戏可能会通知用户轮到他了。如果做法合理，那么推送通知将是一个非常棒的工具，可以推动企业效率的提升以及增强商业iOS应用。

Apple提供了两种通知方法：本地通知与远程通知。本地通知由系统根据应用的设置进行调度。本地通知是在设备上进行管理的，并且是免费的，这意味着没有服务器端的开销，也不需要用户的许可。从通知中心以及总的用户体验的角度来看，本地通知与远程通

知的处理方式是相同的，它们也都包含在通知中心中。

此外，远程通知要使用 Apple 推送通知服务(Apple Push Notification，APN)进行注册才能使用。APN 在 2009 年 6 月随 iOS 3.0 一同发布，并且使用持久的 IP 连接进行通知的发送。远程通知需要额外的服务器开销或是第三方供应商来简化与 APN 的通信。远程通知需要用户显式同意，并且要集成到业务流程中才能生效。

为了强化本地通知与远程通知背后的概念，本章的示例会创建一个类似于客户关系管理(Customer Relationship Manager，CRM)系统的轻量级的关系管理应用。该应用的首个迭代会使用本地通知调度设备提醒。接下来，我们创建一个自定义的远程通知服务，并将其集成到这个关系管理应用中。

10.1 调度本地通知

本地通知非常适合于自包含的应用，其中信息局限在单个设备中。诸如闹钟和任务管理器之类的应用就非常适合于使用本地通知。

与远程通知不同，本地通知无需显式的用户许可就可以使用。如前所述，本地通知是由操作系统传递的，因此开发者与用户可以免费使用，他们无需活动的网络连接即可使用本地通知。然而，本地通知依然会受用户在设置应用中对通知首选项所做的配置的约束。图 10-1 展示了用户可用的不同设置。

图 10-1

10.1.1 创建本地通知

本地通知都是 UILocalNotification 实例，需要使用 fireDate 来告知系统何时推送通知。不使用 fireDate 调度本地通知会立刻将通知发送出去。可能还需要指定 timeZone，这样通知的推送就会随着用户时区的变化而调整。提醒与任务管理器风格的应用可以使用 repeatInterval，这通过 NSCalendarUnit 类型指定。如下代码清单列出了重复间隔可能的 NSCalendarUnit 值。如果指定，那么在当前的通知推送完毕后，系统就可以使用该值调度下一次通知，默认值是不重复的一次性通知。

- NSEraCalendarUnit
- NSYearCalendarUnit
- NSMonthCalendarUnit
- NSDayCalendarUnit
- NSHourCalendarUnit
- NSMinuteCalendarUnit

- NSSecondCalendarUnit
- NSWeekdayCalendarUnit
- NSWeekdayOrdinalCalendarUnit
- NSQuarterCalendarUnit
- NSWeekOfMonthCalendarUnit
- NSWeekOfYearCalendarUnit
- NSYearForWeekOfYearCalendarUnit

本地通知的用户体验由创建时的配置选项设置驱动。可以通过这些选项调整通知的行为、系统如何推送通知以及应用如何响应通知。表 10-1 详细列出了本地通知的可用属性。

表 10-1　UILocalNotification 属性

属　性	类　型	说　明
fireDate	Date	操作系统推送消息的日期与时间
timeZone	TimeZone	如果不指定，那么 fireDate 会被当作 GMT，这可能会导致应用出现差劲的用户体验。指定该值会让 fireDate 根据当前时区进行调整
repeatInterval	CalendarUnit	指定通知重复的频率。通知会在每次推送完毕后重新调度，默认情况下通知是不重复推送的
repeatCalendar	Calendar	通知在创建时使用的日历，默认情况下使用用户当前的日历
alertBody	String	通知消息
hasAction	Boolean	告诉操作系统显示或不显示通知动作
alertAction	String	如果用户配置了通知风格或是配置了锁屏上 Slide To 旁边的值，那么它就是按钮的标题。如果指定了 alertBody 的值，那么该字段的默认值就是 View
alertLaunchImage	String	应用在启动时显示的应用包中的图片文件名，这是通过动作按钮或滑动锁屏上的条目触发的
soundName	String	在收到通知时播放的应用包中的声音文件名。指定 UILocalNotificationDefaultSoundName 会使用默认的系统声音。声音长度限定为 30 秒，如果超过这个时间，操作系统就会播放系统声音
applicationIconBadgeNumber	Integer	作为应用图标的标记号显示的值
userInfo	Dictionary	键值存储，可以通过通知传递自定义数据。userInfo 中的数据可用于增强用户体验，比如在特定的视图中或使用特定的值启动应用

可用于增强用户体验的属性是 alertLaunchImage，该属性指定通过通知动作按钮或锁屏上的启动滑块打开应用时显示的启动图片文件。如果搭配 userInfo 属性使用，那么 alertLaunchImage 可以显示出临时视图，临时视图类似于应用完成启动过程时用户最终看到的布局。这样在应用装配整个视图时，用户会即时获得反馈。

创建好通知后，可以通过操作系统调度通知。本地通知是通过两个 UIApplication 方法调度的，分别是 scheduleLocalNotification:与 presentLocalNotificationNow:。第 1 个方法会根据创建通知时指定的 fireDate 进行调度。第 2 个方法则会忽略掉指定的 fireDate，立刻将通知显示给用户。

代码清单 10-1 介绍了创建、配置与调度本地通知的方法。该方法可以作为应用中通知调度的基础。

代码清单 10-1 调度本地通知的方法(/Application/RelationshipManager/RelationshipManager/Model.m)

```
- (void)scheduleNotificationWithFireDate:(NSDate*)fireDate
                                timeZone:(NSTimeZone*)timeZone
                          repeatInterval:(NSCalendarUnit)repeatInterval
                              alertBody:(NSString*)alertBody
                             alertAction:(NSString*)alertAction
                             launchImage:(NSString*)launchImage
                               soundName:(NSString*)soundName
                             badgeNumber:(NSInteger)badgeNumber
                             andUserInfo:(NSDictionary*)userInfo {
    // create notification using parameter values
    UILocalNotification *notification = [[UILocalNotification alloc] init];
    notification.fireDate = fireDate;
    notification.timeZone = timeZone;
    notification.repeatInterval = repeatInterval;
    notification.alertBody = alertBody;
    notification.alertLaunchImage = launchImage;
    notification.soundName = soundName;
    notification.applicationIconBadgeNumber = badgeNumber;
    notification.userInfo = userInfo;
    // special handling for action
    // default hasAction is YES, so if we don't have one
    // set to no. this removes button / slider
    if(alertAction == nil) {
    notification.hasAction = NO;
    } else {
        notification.alertAction = alertAction;
    }

    // schedule notification asynchronously
    dispatch_async(dispatch_get_main_queue(), ^{
        [[UIApplication sharedApplication]
         scheduleLocalNotification:notification];
    });
}
```

代码清单 10-1 介绍的方法使用 Grand Central Dispatch(Apple 提供的在多核硬件上管理并发的解决方案)来异步调用通知。随着调度通知数的增加，完成调度过程所需的时间也会相应增加。异步调用该过程可以使应用在保存通知时依然能够保持响应性。

说明：

虽然这对于大多数应用来说不算什么问题，不过在本书撰写之际，本地通知数有 64 个的最大限制。你依然可以调度通知，不过到达的通知数被限定为接近 64 个，并且按照 fireDate 的顺序排序，系统会忽略掉其余的通知。这意味着如果现在有 64 个调用的本地通知，那么再调用另一个通知会丢弃掉 fireDate 距当前日期最远的那个通知。循环通知会被当作单个通知，因为它们会自动被系统重新调度。如果超出这个限制，就应该检查一下该如何吸引用户，看看本地通知是否为正确的方式。

既然已经实现了通知调度程序的基础，代码清单 10-2 介绍了一种便捷的方法，可以调度“客户端的后续”通知。该方法联合使用动态与静态内容调用代码清单 10-1 中创建的方法。具体来说，这里指定了一张固定的启动图片和标记数字，并且没有使用声音。

代码清单 10-2　用于调度后续通知的便捷方法(/Application/RelationshipManager/RelationshipManager/Model.m)

```
- (void)scheduleContactFollowUpForContact:(Contact*)contact
                                   onDate:(NSDate*)date
                                 withBody:(NSString*)body
                                andAction:(NSString*)action {
    // add action to user info to help user experience on launch
    NSDictionary *userInfo = [NSDictionary dictionaryWithObjectsAndKeys:
                              contact.emailAddress, @"emailAddress",
                              contact.phoneNumber, @"phoneNumber",
                              @"contactProfile", @"type",
                              action, @"action", nil];

    [self scheduleNotificationWithFireDate:date
                                  timeZone:[NSTimeZone systemTimeZone]
                            repeatInterval:0     // don't repeat
                                 alertBody:body
                               alertAction:action
                               launchImage:@""   // contact default
                                 soundName:nil   // no sound
                               badgeNumber:1
                               andUserInfo:userInfo];
}
```

10.1.2　取消本地通知

现在已经完成通知的调用，如果通知在发送前不再需要了，那么需要一种方式来取消通知。iOS 在 UIApplication 中提供了两种方法来取消本地通知。第 1 个方法是

cancelLocalNotification:，可以用来取消单个通知；第2个方法是cancelAllLocalNotifications:，可以用来取消所有当前已调度的通知，包括那些具有重复间隔设置的通知。通常情况下，你会根据某种情况来取消通知，针对“重置”的情况使用 cancelAllLocalNotifications:，因为这会删除应用调度的所有通知。然而，有时还需要为 cancelLocalNotification:添加封装器，这样就可以删除某种类型的通知，或是根据某种契约对通知进行删除。

可以通过这个轻量级的 CRM 应用针对单个契约调度随后的提醒。当客户取消契约或是发生某些变化时，应用应该可以取消某个契约的通知。为了做到这一点，首先需要一个方法来获取给定契约当前已调度的所有通知，这是通过遍历所有本地通知并检查每个通知的 userInfo 属性做到的。代码清单 10-3 详细介绍了如何检索给定契约的所有通知。

代码清单 10-3 检索给定契约的所有通知(/Application/RelationshipManager/RelationshipManager/Model.m)

```
- (NSArray*)notificationsForContact:(Contact*)contact {
    NSMutableArray *contactNotifications = [[NSMutableArray alloc] init];

    // get ALL scheduled notifications and loop through them
    NSArray *scheduledNotifications = [[UIApplication sharedApplication]
                                       scheduledLocalNotifications];
    for(UILocalNotification *notification in scheduledNotifications) {

        // if the email address in the notification user info matches
        // the contacts email, add it to your output
        if([[notification.userInfo objectForKey:@"emailAddress"]
            isEqualToString:contact.emailAddress]) {

                [contactNotifications addObject:notification];
        }
    }

    return (NSArray*)contactNotifications;
}
```

既然已经获得给定契约的所有通知，现在就可以轻松取消它们了。由于某一时刻用户在多个契约上可能都有挂起的通知，因此在这种情况下就不应该使用 cancelAllLocalNotifications:方法。相反，你应该使用 cancelLocalNotification:方法，其中通知是 UILocalNotification 实例：

```
[[UIApplication sharedApplication]
      cancelLocalNotification:notification];
```

代码清单 10-4 展示了第2种方式，用于取消某个契约的所有本地通知。该方法使用了代码清单 10-3 中所定义方法的响应，并以此为基础来取消通知。

代码清单 10-4　取消某个契约所有本地通知的方法(/Application/RelationshipManager/RelationshipManager/Model.m)

```
- (void)cancelNotificationsForContact:(Contact*)contact {

    // retrieve all notifications for the specified
    // contact and loop through them
    NSArray *notifications = [self notificationsForContact:contact];
    for(UILocalNotification *notification in notifications) {

        // cancel the notification
        [[UIApplication sharedApplication]
        cancelLocalNotification:notification];
    }
}
```

10.1.3　处理本地通知的到达

处理本地通知对于总体的用户体验来说是非常重要的，因此在本地通知到达时必须管理好它们，确保必要的信息能够展现给用户。应用如何响应通知取决于通知创建时指定的配置，以及应用当前是否处于活动状态。如果应用当前没有处于活动状态，并且用户按下了动作按钮(或是滑动了锁屏滑块)，应用就会启动。就像正常的应用启动一样，application:didFinishLaunchingWithOptions:会在应用委托中得到调用，不过 Options 参数中包含了触发应用启动的通知。

如果应用处于活动状态并实现了 application:didReceiveLocalNotification:，那么该方法就会被调用并且带有触发的通知。你必须清楚，如果应用处于活动状态，那么很多配置好的通知行为都会被丢弃，比如显示包含 alertBody 的警告等。然而，如果用户没有禁用应用的通知中心，那么通知依然会显示在通知中心中。如果想在这两种情况下显示出警告，那么需要在 application:didReceiveLocalNotification:中手工创建警告视图。

为了提供最佳的用户体验，本节的示例实现了application:didFinishLaunchingWithOptions:与application:didReceiveLocalNotification:。二者的实现是类似的，它们都会检查通知并确定该采取何种动作。该例构建在代码清单 10-2 中介绍的客户端随后通知的基础之上，应用通过加载契约详细视图进行响应。在创建通知时使用指定的契约方法(call或email)，应用还会询问用户是否应该开始呼叫或发送邮件。当业务流程不断发展变化时，通过这种方式可以轻松为更多的通知类型添加支持。

代码清单 10-5 与 10-6 详细介绍了每一种实现。

代码清单 10-5　应用启动时本地通知的处理(/Application/RelationshipManager/RelationshipManager/AppDelegate.m)

```
- (BOOL)application:(UIApplication *)application
        didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
```

```
    // determine if app launched from a local notification
    UILocalNotification *localNotification =
        [launchOptions objectForKey:
        UIApplicationLaunchOptionsLocalNotificationKey];

    if(localNotification != nil) {
        NSDictionary *userInfo = localNotification.userInfo;

        NSString *action = [userInfo objectForKey:@"action"];
        Contact *contact = [[Model sharedModel]
                            contactWithEmailAddress:
                            [userInfo objectForKey:@"emailAddress"]];

        // initiate a phone call
        if([action isEqualToString:@"Call"]) {
            NSString *phone = [NSString stringWithFormat:@"tel:%@",
                               contact.phoneNumber];

            [[UIApplication sharedApplication]
            openURL:[NSURL URLWithString:phone]];

        // start an email
        } else if ([action isEqualToString:@"Email"]) {
            NSString *email = [NSString stringWithFormat:@"mailto:%@",
                               contact.emailAddress];

            [[UIApplication sharedApplication]
             openURL:[NSURL URLWithString:email]];
        }
    }
    ...
}
```

代码清单 10-6 应用处于活动状态时本地通知的处理(/Application/RelationshipManager/RelationshipManager/AppDelegate.m)

```
- (void)application:(UIApplication*)application
     didReceiveLocalNotification:(UILocalNotification *)notification {

    // alert the user that a notification was received
    // because the user was in the application, we present
    // them with some additional information
    dispatch_async(dispatch_get_main_queue(), ^{
        NSDictionary *userInfo = notification.userInfo;

        NSString *action = [userInfo objectForKey:@"action"];
        Contact *contact = [[Model sharedModel]
                            contactWithEmailAddress:
                            [userInfo objectForKey:@"emailAddress"]];
```

```
[UIAlertView alertViewWithTitle:@"Reminder"
                        message:notification.alertBody
              cancelButtonTitle:@"Cancel"
              otherButtonTitles:[NSArray
                                 arrayWithObjects:@"View Contact",
                                 action, nil]
       onDismiss:^(int buttonIndex)
          {
             // display the contact details
             if(buttonIndex == 0) {

                ContactDetailTableViewController *contactVC =
                   [[ContactDetailTableViewController alloc]
                   initWithStyle:UITableViewStyleGrouped];

                contactVC.contact = contact;
                contactVC.presentedModally = YES;

                UINavigationController *nc =
                   [[UINavigationController alloc]
                   initWithRootViewController:contactVC];

                [self.navigationController
                 presentModalViewController:nc animated:YES];

             // initiate the selected action
             } else if(buttonIndex == 1) {
                // initiate a phone call
                if([action isEqualToString:@"Call"]) {
                   NSString *phone =
                   [NSString stringWithFormat:@"tel:%@",
                    contact.phoneNumber];

                   [[UIApplication sharedApplication]
                    openURL:[NSURL URLWithString:phone]];

               // start an email
               } else if([action isEqualToString:@"Email"]) {
                  NSString *email =
                  [NSString stringWithFormat:@"mailto:%@",
                   contact.emailAddress];

                  [[UIApplication sharedApplication]
                   openURL:[NSURL URLWithString:email]];
               }
            }
         }
      onCancel:^()
         {
            // don't do anything for cancel
```

```
        });
}
```

代码清单 10-6 用到了UIAlertView上的一个类别，它是由Mugunth Kumar编写的，用于增强UIAlertView显示与委托方法。通过这种方式可以使用来自于触发通知中的数据处理UIAlertView的输入。这些类别位于https://github.com/MugunthKumar/UIKitCategoryAdditions。

警告：

在本书撰写之际，iOS 模拟器存在如下问题——application:didReceiveLocalNotification:会被触发两次，速度非常快。如果不做验证，位于 application:didReceiveLocalNotification:中的任何逻辑就会被执行两次。这个问题的解决办法就是在一台实际的设备上测试本地通知功能。

既然已经实现了本地通知，接下来就可以开始学习如何通过远程通知增强这个关系管理应用了。

10.2 注册并响应远程通知

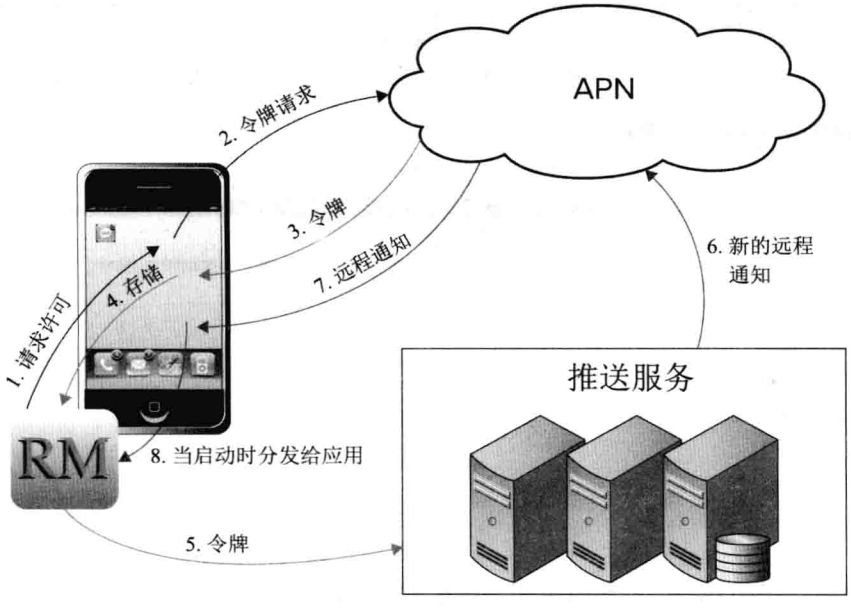
虽然本地通知在某些情况下是非常棒的解决方案，不过它们也是有局限性的，因为应用需要运行才能调度通知。本地通知还受限于调度它们的设备。远程通知可以解决这个问题，可以在外部服务器上发起设备通知，然后通过 APN 进行传输。

远程通知具有很高的灵活性，因为可以在应用外发起通知。假设如下场景：客户在某公司的收货部门，有这样一项新需求——当购买的东西到达仓库时需要向审批者发送一条通知。可以通过远程通知实现这个需求而无需应用处于运行状态。如果使用本地通知或自定义警告，那么需要修改应用，向所有用户发布新版本；最大的问题则是用户必须运行应用才能收到通知。

本节将会实现自定义的远程通知服务，同时还会介绍如何在应用中注册并响应远程通知。自定义的通知服务会连接到 APN 以注册待发送的通知。虽然可以遵照本节的示例，但要想测试自己的实现，则需要做两件事：

- 付费的 Apple 开发者身份：iOS 模拟器无法处理远程通知，因此需要将测试应用安装到设备上。此外，需要为通知服务器向 Apple 申请 SSL 证书，本章后面的 10.2.1 节“配置远程通知”将会对此进行介绍。
- 能够访问一台联网并且运行着 PHP 与 MySQL 的 Web 服务器：出于测试的目的，运行在 Mac 机上的服务器就足够了。

APN 是通信网关，控制着所有远程通知的发送。该网关对于消费型(通过 App Store 分发)与企业级应用来说是公用接口。图 10-2 概览了本章介绍的整个远程通信过程。如果将远程通知的发送外包出去，那么这张图就会有一些微小的差别。



改编自Apple开发者文档

图 10-2

应用请求通知发送许可意味着流程的开始。请求会被发送给操作系统，后者会提示用户进行许可。如果用户同意，那么操作系统会从 APN 处获取到设备令牌，然后将其发送给应用，随后存储到应用的远程推送服务中。当出现某个恰当的事件时，应用的远程推送服务就会使用之前接收到的设备令牌将某个远程通知注册到 APN。如果通知注册成功，APN 就会竭尽全力地发送通知。由于网络连接与设备状态存在着太多的变数，因此我们无法确定通知是否成功到达。如果一切正常，通知就会发送给设备。如果应用处于打开状态，那么操作系统就会将通知发送给活动的设备进行处理，如前所述。如果应用没有处于活动状态，那么操作系统就会根据用户的设置显示通知。

实际上，APN 会将远程通知发送给与某个设备关联的令牌。开发者负责获取该令牌，并且在将通知注册到 APN 时将其作为发送目的地。允许发送远程通知是由每个用户决定的；因此，开发者需要精心设计应用，使之能够优雅降级。远程通知应该增强用户体验，不过应用要能在用户未授权的情况下正常使用。

10.2.1　配置远程通知

在开始使用 APN 之前，需要在 iOS Provisioning Portal 中为每个应用配置远程通知。本节将会介绍如何为应用配置远程通知、获取必要的证书文件以与 APN 通信以及注册推送的通知。

(1) 首先需要使用电脑中的 Keychain Access 应用生成证书签名请求(Certificate Signing Request，CSR)。CSR 指的是从数字证书认证机构获取身份证书的请求，对于 APN 来说就是 Apple。图 10-3 展示了如何发起 CSR。

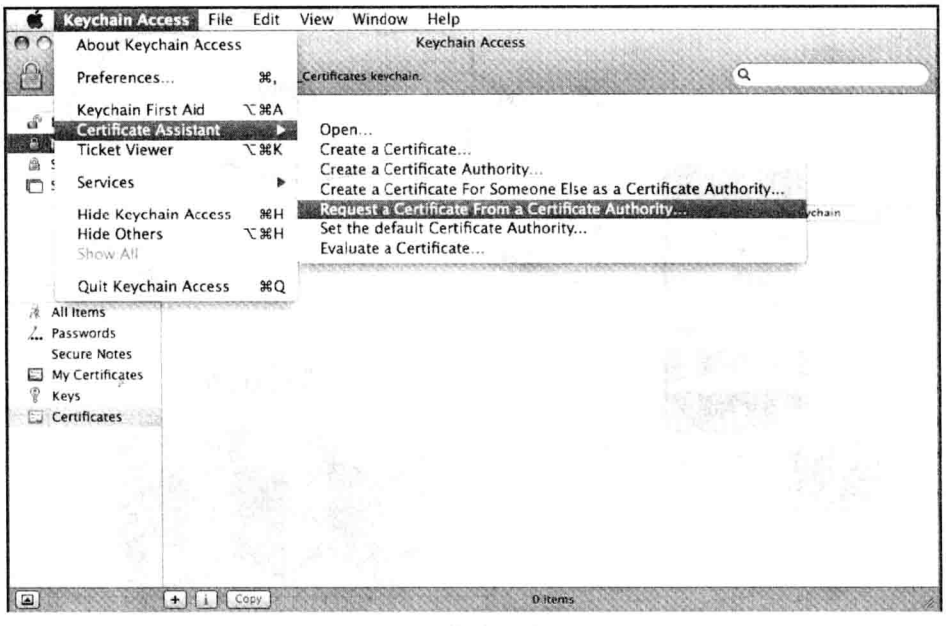


图 10-3

(2) 开启 CSR 向导后，你会看到类似于图 10-4 的界面。输入 Email 地址和请求的描述性名字。这个名字还会绑定到与 CSR 一同生成的私钥。选择描述性名字，使得未来定位私钥变得更加轻松。需要使用自己的私钥才能连接到 APN。

说明：

有必要将 CSR 保存起来。APN 证书、开发与产品都有一年的生命周期。保存 CSR 会简化续费过程，也有助于防止服务中断。

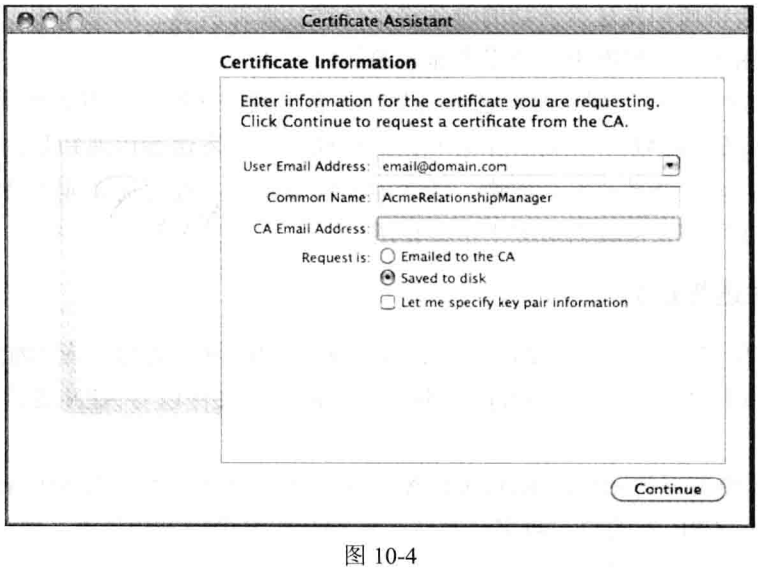


图 10-4

(3) 生成好 CSR 后，选择新的私钥，将其导出到某个安全且能记住的地方。可以按下 Ctrl 键并单击 Keychain Access 中的私钥条目，然后选择导出选项来导出私钥，不过请记住设置的导出密码。

(4) 现在，在浏览器中输入 https://developer.apple.com/ios/并回车，打开 iOS Provisioning Portal，然后转到应用列表。由于这是对已有的应用(即上一节创建的应用)进行增强，因此列表中应该有 Relationship Manager 这个条目。该条目应该类似于图 10-5。在列表中定位到应用后，单击 Action 列中的 Configure 动作链接。

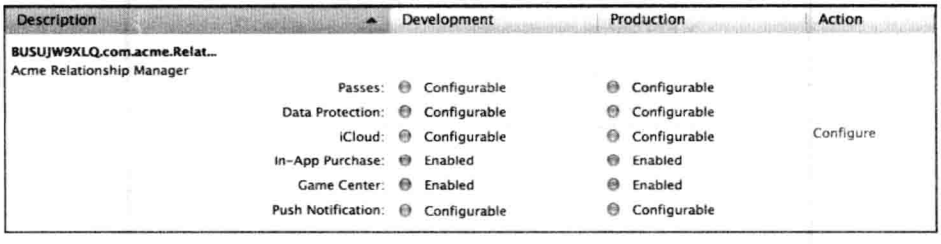


图 10-5

(5) 单击 Configure 动作链接会显示类似于图 10-6 的界面。可以在该界面上为开发与生产配置通知。选中 Enable for Apple Push Notification Service 复选框，然后在最右侧的列中选择配置项。

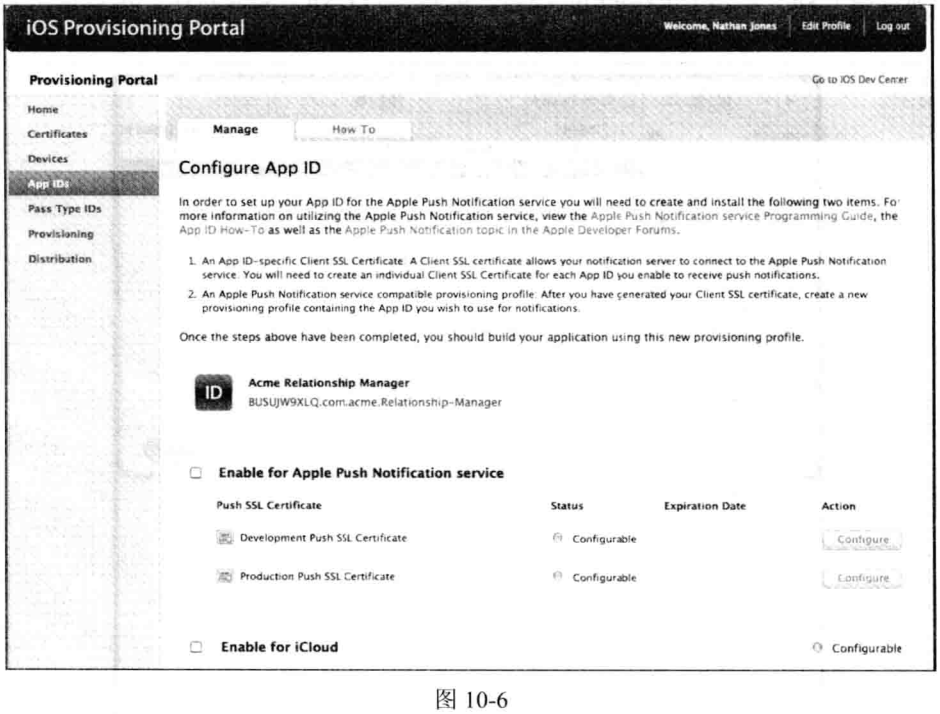


图 10-6

(6) 单击 Continue 后会出现类似于图 10-7 所示的界面。由于已经生成了 CSR(参见图 10-4)，因此可以单击 Continue 继续。

图 10-7

(7) 在图 10-7 所示的界面上单击 Continue 后会进入到另一个界面，类似于图 10-8，它会提示你选择 CSR。单击 Choose File 按钮，转到你的 CSR，将其选中，然后单击 Generate 按钮。

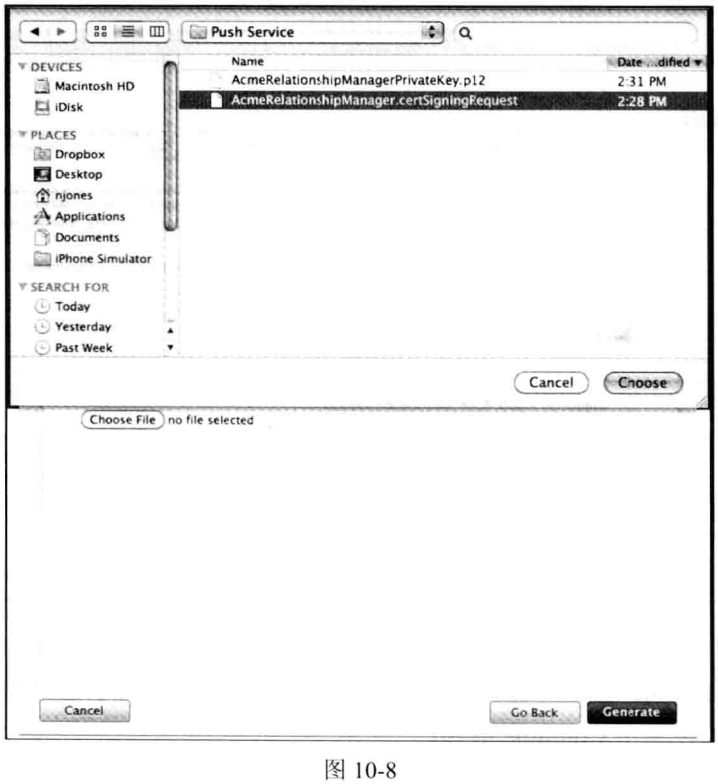


图 10-8

(8) 如果一切如预期一样，那么在 Apple 生成 APN 证书时，你会在界面上看到覆盖层，然后出现一条成功消息，类似于图 10-9。单击 Continue 按钮转到下一个界面，然后下载证书，如图 10-10 所示。

图 10-9

图 10-10

(9) 回到应用列表界面；你会看到已为开发成功配置好了推送通知，如图 10-11 所示。

说明：

使用远程通知的应用构建(开发与分发)必须使用针对远程通知的配置文件进行签名。如果之前已经为应用生成了配置文件，那么应该在配置完远程通知后重新生成。只有在应用正确签名后才能请求许可进行推送。

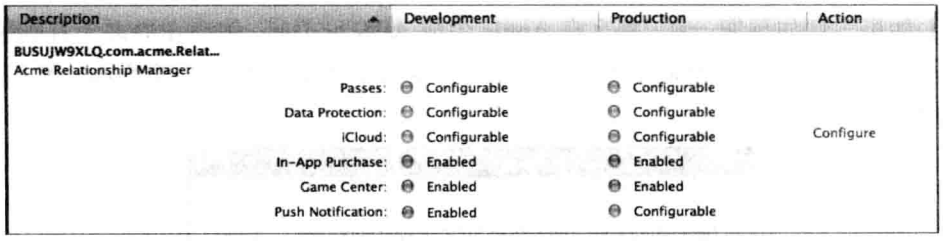


图 10-11

由于本章使用 PHP 实现服务层，因此需要将随 CSR 一同生成的私钥(参见图 10-4)及 Apple 提供的 SSL 证书(参见图 10-10)放到单个 PEM 格式的文件中。可以通过 PEM 文件格式指定证书、私钥或二者的组合。在其他的服务器端语言中，这些步骤可能并不需要，不过本章中用于发起 APN 连接的函数 stream_context_create()要求证书必须是 PEM 格式的。如下步骤介绍了如何转换与合并这两个文件：

(1) 打开电脑上的 Terminal 应用，转到这两个文件的存放目录。出于演示目的，假设这两个文件位于桌面的 Push Service 目录中，如下所示：

```
$ cd /Users/njones/Desktop/Push\ Service/
```

(2) 将从 Apple 下载的 SSL 证书转换为 PEM 格式，如下所示：

```
$ openssl x509 -inform der -in AcmeRelationshipManager.cer -out
AcmeRelationshipManagerCert.pem
```

(3) 将 PKCS12(.p12)格式的私钥转换为 PEM 格式，如下所示：

```
$ openssl pkcs12 -in AcmeRelationshipManagerPrivateKey.p12 -out
AcmeRelationshipManagerKey.pem -nocerts
```

(4) 你会被提示输入导入密码，这是在从 Keychain Access 中导出私钥时使用的密码。成功输入导入密码后，你会被提示输入 PEM 密码，然后再次输入以进行验证。应该确保密码的安全性并将其存储起来，在连接到 APN 时会用到。

(5) 最后，将证书与私钥 PEM 文件合并到单个文件中：

```
$ cat AcmeRelationshipManagerCert.pem AcmeRelationshipManagerKey.
pem> AcmeCertKey.pem
```

说明：

可以在 Terminal 中通过 openssl s_client 命令测试与 APN 的连接。可以通过 s_client 命令连接到 SSL 服务器。APN 有两台服务器：开发服务器与产品服务器，分别位于 gateway.sandbox.push.apple.com:2195 与 gateway.push. apple.com:2195。产品端点用于签名以进行分发的应用。可以访问 www.openssl.org/docs/apps/ s_client.html 以了解关于 s_client 命令的更多信息。

现在已经配置好了远程通知，并且也拥有服务器连接到APN所需的必要文件，还注册好了通知。现在需要请求许可以将通知发送给用户。

10.2.2　注册远程通知

在发送第 1 个通知前，需要配置远程服务器以注册每个用户的设备。用户可能有多台设备，存储方法必须考虑到这一点。代码清单 10-7 列出了一系列 SQL 语句，可以此为基础将数据存储到远程服务器。

代码清单 10-7　用于远程通知处理的数据库结构(/Push Server/pushdatastructure.sql)

```
CREATE TABLE IF NOT EXISTS 'users' (
  'userid' varchar(120)
     NOT NULL,
  'datecreated' timestamp
     NOT NULL DEFAULT '0000-00-00 00:00:00',
  PRIMARY KEY ('userid')
) ENGINE=MyISAM DEFAULT CHARSET=utf8;

CREATE TABLE IF NOT EXISTS 'user_tokens' (
  'userid' varchar(120) NOT NULL,
  'token' varchar(64) NOT NULL,
  'datecreated' timestamp
     NOT NULL DEFAULT '0000-00-00 00:00:00',
  'dateremoved' timestamp
     NOT NULL DEFAULT '0000-00-00 00:00:00'
     COMMENT 'date the feedback service was polled for token',
  PRIMARY KEY ('userid','token')
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

构建好数据库后，应该创建一个简单的 Web Service 脚本来注册用户与设备。该脚本会创建用户与设备的关系，用于确定通知应该发送给哪些设备。代码清单 10-8 展示了一段简单的 PHP 脚本，用于注册用户及设备。这段脚本首先会确定传进来的请求的用户是否已经存在。如果用户不存在，那么脚本会创建用户，然后注册设备。

代码清单 10-8　用于注册用户与设备的服务器端脚本(/Push Server/register.php)

```
<?php
    ...

    // get the post body
    $userid = $_REQUEST['user'];
    $token = $_REQUEST['token'];

    if(empty($userid) || empty($token)) {
        sendAPIResponse(400);
        return;
    }
```

```
    // determine if user exists
    $sql = "SELECT userid
        FROM users
        WHERE userid='".$userid."' LIMIT 1;";
    $query = mysql_query($sql, $dbConnection);
    $userExists = mysql_fetch_row($query);

    // add a 'user' record
    if(!$userExists) {
        $sql = "INSERT INTO users (userid, datecreated)
                VALUES ('".$userid."', '".$now."');";
        if(!mysql_query($sql, $dbConnection)) {
            // return error
            sendAPIResponse(400);
            return;
        }
    }

    // determine if token already exists
    $sql = "SELECT token
            FROM user_tokens
            WHERE userid='".$userid."'
            AND token='".$token."' LIMIT 1;";
    $query = mysql_query($sql, $dbConnection);
    $tokenExists = mysql_fetch_row($query);

    // add a token for current user
    if(!$tokenExists) {

        $sql = "INSERT INTO user_tokens (userid, token, datecreated)
                VALUES ('".$userid."','".$token."','".$now."');";
        if(!mysql_query($sql, $dbConnection)) {
            // return error
            sendAPIResponse(400);
            return;
        }
    }

    // close the database connection
    mysql_close($dbConnection);

    // return success
    sendAPIResponse(200);
?>
```

准备好注册脚本后，现在是时候请求应用的用户许可来发送远程通知了。这需要为每台用户设备上的每个应用重复执行。代码清单 10-9 展示了如何发起远程通知的同意流程，它是应用启动过程的一部分。参见 10.3 节“理解通知最佳实践”以了解关于请求远程通知

许可的最佳实践。

代码清单 10-9　请求许可以发送远程通知(/Application/RelationshipManager/RelationshipManager/AppDelegate.m)

```
- (BOOL)application:(UIApplication *)application
     didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    // request permission to deliver remote notifications, if needed
    BOOL requested = [[NSUserDefaults standardUserDefaults]
                      boolForKey:kPushTokenTransmitted];
    if(requested != YES) {
        [[UIApplication sharedApplication]
        registerForRemoteNotificationTypes:
        (UIRemoteNotificationTypeAlert |
         UIRemoteNotificationTypeBadge |
         UIRemoteNotificationTypeSound)];
    }
    ...
}
```

应用启动后，你会看到类似于图 10-12 的警告，提示用户同意远程通知的发送。

图 10-12

表 10-2 列出了可以请求发送的 4 种类型的远程通知。如代码清单 10-9 所示，可以请求这 4 种通知类型的任意组合。用户可以通过 Settings 应用随时修改这些许可，限制远程通知可以触发或不能触发哪些内容。比如，如果用户禁用应用的声音，那么在通知发送时就不会播放声音，即便通知负载指定了声音文件也是如此。应用这样设计，便可以独立启用或禁用任何类型的通知。

表 10-2 远程通知的类型

类 型	说 明
UIRemoteNotificationTypeAlert	该许可可以根据用户的配置显示警告视图或横幅。如果用户已经配置，那么该警告还会显示在用户的通知中心中和锁屏上
UIRemoteNotificationTypeBadge	该许可可以设置应用的图标计数
UIRemoteNotificationTypeSound	当警告或图标计数送达时，该警告会播放一小段声音
UIRemoteNotificationTypeNewsstand-ContentAvailability	当通过 Newsstand 框架发现有新的内容可供下载时，该许可会通知应用

用户授权许可后，应用会调用应用委托的application:didRegisterForRemoteNotificationsWithDeviceToken:方法。相反，如果用户拒绝请求，那么 application:didFailToRegisterForRemoteNotificationsWithError:方法会被调用。在application:didRegisterForRemoteNotificationsWithDeviceToken:方法中，应用应该创建对代码清单 10-8 中远程服务器脚本的请求来注册用户及设备令牌。

代码清单 10-10 介绍了如何获取设备令牌，同时提供了一种方法来将其传递给在代码清单 10-8 中创建的脚本。如果选择实现服务器端的本地化(在 10.2.3 节“远程通知负载”中介绍)，那么可以首先在这里获取到用户的本地信息，然后将其与令牌一起传递。最佳实践是存储一个标识符，表示令牌已经成功传递给了通知提供者。这样做可以避免对服务器不必要的调用，因为许可之后对 registerForRemoteNotificationTypes:方法的调用已经可以调用 didRegisterForRemoteNotificationsWithDeviceToken:方法了。可以看到，如果对代码清单 10-8 的调用是成功的，那么代码清单 10-10 会设置 BOOL。

代码清单 10-10 处理远程通知的到达(/Application/RelationshipManager/RelationshipManager/AppDelegate.m)

```
- (void)application:(UIApplication *)application
    didRegisterForRemoteNotificationsWithDeviceToken:(NSData *)
        deviceToken {
    // hardcode the current user, this would typically
    // be a token or value retrieved after they logged
    // in to use the app
    NSString *userId = @"nate@emaildomain.com";
    NSString *token = [NSString stringWithFormat:@"%@", deviceToken];

    // clean the token
    token = [token stringByTrimmingCharactersInSet:
            [NSCharacterSet characterSetWithCharactersInString:@"<>"]];
    token = [token stringByReplacingOccurrencesOfString:@" "
                                                withString:@""];

    // handle the request off the main thread
    dispatch_async(dispatch_get_main_queue(), ^{
```

```
        // build the post body
        NSString *postBody = [NSString
                              stringWithFormat:@"user=%@&token=%@",
                              userId,
                              token];
        // build the request
        NSString *endpoint = @"http://yourdomain.com/push/register.php";
        NSMutableURLRequest *request =
            [[NSMutableURLRequest alloc]
             initWithURL:[NSURL URLWithString:endpoint]
             cachePolicy:NSURLRequestReloadIgnoringCacheData
             timeoutInterval:30.0];

        // configure the remaining request properties
        request.HTTPMethod = @"POST";
        request.HTTPBody = [postBody
                            dataUsingEncoding:NSUTF8StringEncoding];
        [request setValue:@"application/x-www-form-urlencoded"
       forHTTPHeaderField:@"Content-Type"];

        NSError *error = nil;
        NSHTTPURLResponse *response;

        // this method returns NSData, but in this case
        // we don't care about it
        [NSURLConnection sendSynchronousRequest:request
                              returningResponse:&response
                                          error:&error];

        // verify we got a success
        if(response.statusCode == 200) {

            // save our local flag so that we don't
            // hit this logic each time the app is opened
            [[NSUserDefaults standardUserDefaults]
             setBool:YES forKey:kPushTokenTransmitted];
            [[NSUserDefaults standardUserDefaults] synchronize];

        // alert the user if we didn't get a success
        } else {
            [[[UIAlertView alloc] initWithTitle:@"Error"
                                        message:@"Unable to "
                                       delegate:nil
                              cancelButtonTitle:@"OK"
                              otherButtonTitles:nil] show];
        }
    });
}
```

10.2.3 远程通知负载

APN 负载是 JSON 对象，并且被严格限制为 256 字节。需要确保负载不会超过这个限制，否则通知会被 APN 拒绝。负载采用键值对的形式，顶层命名空间是 aps。如下代码片段以较好的可读性展现了标准的 APN 负载的结构：

```
{
    "aps" : {
        "alert" : "New delivery received.",
        "badge" : 1,
        "sound" : "delivery.caf"
    }
}
```

考虑到每个字节计数有 256 字节的限制，因此最佳实践是删除空白与回车换行符。如下代码片段以压缩的形式展现了之前的负载：

```
{"aps":{"alert":"New delivery received.","badge":1,"sound":"delivery.caf"}}
```

aps 命名空间包含 3 个可用的属性：alert、badge 及 sound。表 10-3 详细介绍了每个属性的用法。注意传递给 alert 属性的值既可以是字符串，也可以是字典。如果传递字典，那么可以进一步配置通知的样子以及应用的功能。表 10-4 介绍了 alert 字典可用的各种属性。

表 10-3 可用的 aps 字典属性

属 性	类 型	说 明
alert	String 或 Dictionary	如果传递字符串，那就用作消息体，并且会使用两个按钮：关闭与查看。参见表 10-4 以了解在传递字典时可用的各种值
badge	Number	这是显示在应用图标上的值。该数字没有什么逻辑(比如增加或减少)：发送出去后就会显示出来。因此，要想显示未读的数量，就需要在服务器与标记上维护阅读状态。如果没有向该属性传递任何值，系统就会删除现有的标记值
sound	String	这是显示通知时播放的声音文件名。该文件必须位于应用包中，支持的格式有 aiff、wav 及 caf

表 10-4 alert 可用的子属性

子 属 性	类 型	说 明
body	String	通知消息
action-loc-key	String	如果指定，那么该值会修改警告的显示，方式是再添加一个按钮，并将该值作为右侧按钮上的文本。如果没有指定，那么只使用单个按钮

(续表)

子 属 性	类　型	说　明
loc-key	String	用作通知体的与本地化值对应的键。该键的值可以通过%@、%n 及%$格式化以接收来自于 loc-args 的参数值
loc-args	数组	一个字符串数组，用于替换由 loc-key 设置/获取的值中的格式化占位符
launch-image	String	如果用户通过动作按钮或滑动锁屏上的条目触发应用的启动，那它就是在应用启动时所显示的应用包中的图片文件名

有两种方法可以本地化远程通知：服务器端与应用内。服务器端本地化需要在发送前翻译消息体与动作按钮文本。该策略还需要维护对用户当前区域的引用，这样消息就能被正确地翻译了。

在应用内本地化远程通知需要预先规划，不过只需要为 loc-key 参数指定值即可，该参数是与用作消息体文本的本地化值相对应的键。还可以提供可选的参数数组，使用 loc-args 属性在本地化的消息体中显示。本地化消息依然受到 256 个字节负载的限制。该方法有如下额外好处：只需要消息而非整个消息体的键。如果某语言被本地化，那么考虑到负载大小的约束，这种方式具有极大的优势。

配置好 aps 命名空间后，还可以创建自定义命名空间以向应用发送特定的信息。自定义命名空间类似于前面介绍的 UILocalNotification 中的 userInfo 属性。自定义命名空间必须与负载结构中的 aps 命名空间位于同一层次。注意 256 字节的限制会施加于整个负载，而不仅仅是 aps 命名空间中的数据。在设计远程通知时，必须考虑这个限制才能避免因大小超出限制而被拒绝掉。如下代码介绍了如何在通知负载中加入自定义命名空间：

```
{
    "aps" : {
        "alert" : "New delivery received.",
        "badge" : 1,
        "sound" : "delivery.caf"
    },
    "emailAddress" : "nate@prospect.com",
    "action" : "Email"
}
```

10.2.4　发送远程通知

注册好用户的设备后，现在是时候发送通知了。要想注册通知以进行发送，需要通过本章之前在配置流程中创建的 SSL 证书连接到 APN。本节介绍的方法包含了两段重要的服务器端脚本：一段用于处理 APN 连接，另一段用于提供一个简单的接口来发起通知。代码清单 10-11 介绍了如何连接到 APN、使用简单的格式构建二进制负载以及如何注册通知。

代码清单 10-11　连接 APN 的方法(/Push Server/apns.php)

```
function sendPushNotification($token, $payload) {
```

```
        $certificate = "../Path/To/Certificate/AcmeCertKey.pem";
    $passphrase = "YourPassphrase";
    $endpoint = "ssl://gateway.sandbox.push.apple.com:2195"

    $context = stream_context_create();
    stream_context_set_option($context,
                              'ssl',
                              'local_cert',
                              $certificate);

    stream_context_set_option($context,
                              'ssl',
                              'passphrase',
                              $passphrase);

    // connect to APNs server
    $conn = stream_socket_client(
                $endpoint,
                $err,
                $errstr,
                60,
                STREAM_CLIENT_CONNECT | STREAM_CLIENT_PERSISTENT,
                $context
    );

    if(!$conn) {
        echo "Connection to APNs Failed...";
        return;
    }

    // build the binary
    $message = chr(0) .
               pack('n', 32) .
               pack('H*', $token) .
               pack('n', strlen($payload)) .
               $payload;

    // push the notification
    $result = fwrite($conn, $message, strlen($message));

    // close the connection to APNs
    fclose($conn);

    if($result) {
        echo "Notification sent...";
    } else {
        echo "Error sending notification...";
    }
}
```

为了解决简单格式的一些问题，Apple 引入了增强格式。开发者可以通过增强格式指定通知的过期日期。如果出现错误，还可以获得更多的细节信息。

APN 存储设备上每个应用注册的最后一条通知。这样一来，如果设备离线，那么在连接时 APN 会推送该条通知。问题在于通知中的内容会过期，这时你不希望再发送该条通知。过期日期是固定的时间，到了这个时间 APN 会丢弃掉消息，因为消息已经变得无效了。可以通过指定 0 值或更小的值，让 APN 不要存储通知。

对于简单格式来说，如果出现错误，那么 APN 不会有任何表示。开发者可以通过增强格式为每条待发送的通知分配一个标识符。如果出现错误，那么 APN 会返回错误响应以及分配的标识符以供进一步调查所用。表 10-5 详细列出了不同的 APN 响应代码及含义。

表 10-5　APN 增强格式的响应代码

响 应 代 码	说　　明
0	没有遇到错误
1	处理错误
2	缺少设备令牌
3	缺少主题
4	缺少负载
5	无效的令牌大小
6	无效的主题大小
7	无效的负载大小
8	无效令牌
255	无(未知)

代码清单 10-12 介绍了一段简单的脚本，可以在浏览器中调用来注册通知。该脚本会获取到与被通知的用户相关联的所有设备，然后调用代码清单 10-11 中创建的脚本来处理 APN 通信。实际上，这段脚本不应该让公众访问；而是应该通过某种类型的认证机制来访问。

代码清单 10-12　远程通知测试发送脚本(/Push Server/sendNotification.php)

```
<?php

    ...
    // get request parameter values
    $userId = $_REQUEST['userid'];
    $contactEmail = $_REQUEST['contact'];
    $message = $_REQUEST['message'];
    $badge = $_REQUEST['badge'];
    $sound = $_REQUEST['sound'];
```

```
    // clean up the action value
    $action = $_REQUEST['action'];
    $action = (!empty($action)) ? $action : "View";

    // get token(s) for user
    $sql = "SELECT token
            FROM user_tokens
            WHERE userid='".$userId."';";

    $query = mysql_query($sql, $dbConnection);

    // send push to ALL devices that belong to user
    while($row = mysql_fetch_array($query)) {
        // create the payload
        $alert['body'] = $message;
        $alert['action-loc-key'] = $action;

        $aps['alert'] = $alert;

        // add sound
        if(!empty($sound)) {
            $aps['sound'] = $sound;
        }

        // add badge
        if(!empty($badge)) {
            $aps['badge'] = intval($badge);
        }

        $payload['aps'] = $aps;

        // add custom namespace fields
        $payload['emailAddress'] = $contactEmail;

        // connect to apns and send push
        sendPushNotification($row['token'], json_encode($payload));
    }

    // close the database connection
    mysql_close($dbConnection);
?>
```

编写好服务器端组件并注册好设备后，现在可以进行测试了。调用代码清单 10-12 中创建的脚本并传递如下参数值。如果一切正常，那么很快就会接收到通知，类似于图 10-13。

- userid：nate@emaildomain.com
- contact：John@prospect.com
- message：Email John about proposal
- badge：1

- action：Email

上述参数列表中的 userid 值匹配代码清单 10-10 注册过程中的硬编码值。为契约指定的值应该匹配添加到 Relationship Manager 应用中的一个条目，这样可以确保你会触发下一节将会介绍的处理逻辑。

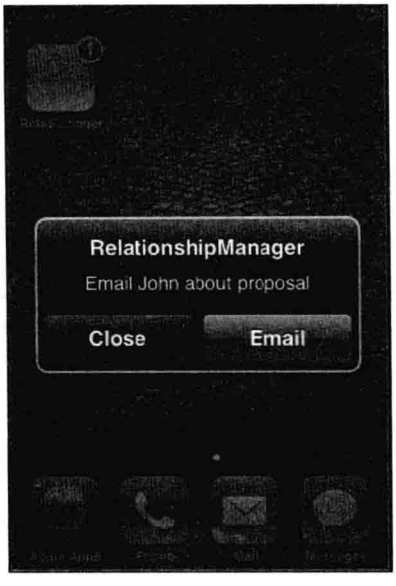


图 10-13

根据需求的不同，特别是如果希望推送服务有较高的吞吐量，你应该考虑使用队列消息方式来提升灵活性与效率。这需要使用服务器端脚本流将通知排队到新的数据库表中来发送，而不是立刻连接到 APN。接下来，后台进程会处理排队的通知，建立到 APN 的连接，注册并归档通知。

Apple 表示会不断监控 APN 连接，如果出现不断建立连接的情况，就会将此当作拒绝服务攻击。请参阅开发者门户上的 APNs Provider Communication 文档以了解详情，地址是 https://developer. apple.com/ios。上述增强有助于优化 APN 连接，并且提升了灵活性，能够实现本章后面将要介绍的众多最佳实践，比如“请勿打扰”和服务器端的本地化等。

10.2.5　响应远程通知

处理远程通知的过程与本地通知一样，除了调用的委托方法以及 UIAlertView 数据的获取方式有些差别。应用的响应方式依旧主要由通知提供者发送的负载以及应用是否处于活动状态驱动的。如果应用没有处于活动状态，那么当用户单击动作按钮(或是滑动锁屏的滑块)时，应用就会启动。就像标准启动一样，应用委托中的 application:didFinishLaunchingWithOptions:方法会被调用。可以通过关键的 UIApplicationLaunchOptionsRemoteNotificationKey 从 launchOptions 字典中获取远程通知负载。代码清单 10-13 介绍了如何在应用启动过程中添加自定义的通知处理。

代码清单 10-13　启动过程中的远程通知处理(/Application/RelationshipManager/RelationshipManager/AppDelegate.m)

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    ...

    // determine if app launched from a push notification
    NSDictionary *pushNotification =
        [launchOptions
        objectForKey:UIApplicationLaunchOptionsRemoteNotificationKey];
    if(pushNotification != nil) {
```

```
        NSString *action = [[[pushNotification objectForKey:@"aps"]
                                        objectForKey:@"alert"]
                                        objectForKey:@"action-loc-key"];

        Contact *contact = [[Model sharedModel]
                            contactWithEmailAddress:
                            [pushNotification
                             objectForKey:@"emailAddress"]];

        // initiate a phone call
        if([action isEqualToString:@"Call"]) {
            NSString *phone = [NSString stringWithFormat:@"tel:%@",
            contact.phoneNumber];
            [[UIApplication sharedApplication]
            openURL:[NSURL URLWithString:phone]];

            // start an email
        } else if([action isEqualToString:@"Email"]) {
            NSString *email = [NSString stringWithFormat:@"mailto:%@",
                               contact.emailAddress];

            [[UIApplication sharedApplication]
            openURL:[NSURL URLWithString:email]];

        }
    }
    ...
}
```

如果应用处于活动状态并实现了 application:didReceiveRemoteNotication:方法，那么该方法会被调用并且使用来自于通知的负载。因此，如果想要通知用户，就必须实现该功能。代码清单 10-14 介绍了在应用处于活动状态时如何拦截远程通知，并向用户展现待执行的动作列表。大部分逻辑都可以通过本地通知实现进行增强，除了动作文本的获取方式。对于本地通知与远程通知来说，它们在字典中的位置是不同的。

代码清单 10-14 应用处于活动状态下的远程通知处理(/Application/RelationshipManager/RelationshipManager/AppDelegate.m)

```
- (void)application:(UIApplication*)application
    didReceiveRemoteNotification:(NSDictionary *)userInfo {

    // alert the user that a notification was received
    // because the user was in the application, we present
    // them with some additional information / options
    dispatch_async(dispatch_get_main_queue(), ^{

        NSString *action = [[[userInfo objectForKey:@"aps"]
                                        objectForKey:@"alert"]
```

```
                                        objectForKey:@"action-loc-key"];
    Contact *contact = [[Model sharedModel]
                        contactWithEmailAddress:
                        [userInfo objectForKey:@"emailAddress"]];

// get the reminder message
NSString *message;
message = [[[userInfo objectForKey:@"aps"]
            objectForKey:@"alert"] objectForKey:@"body"];
if(message == nil) {
    // no message found at that path
    // that implies a simple notification structure
    message = [[userInfo objectForKey:@"aps"]
               objectForKey:@"alert"];
}

[UIAlertView alertViewWithTitle:@"Reminder"
                        message:message
               cancelButtonTitle:@"Cancel"
               otherButtonTitles:[NSArray
                                  arrayWithObjects:@"View Contact",
                                  action, nil]
                      onDismiss:^(int buttonIndex)
{
       // display the contact details
       if(buttonIndex == 0) {

          ContactDetailTableViewController *contactVC =
          [[ContactDetailTableViewController alloc]
          initWithStyle:UITableViewStyleGrouped];

          contactVC.contact = contact;
          contactVC.presentedModally = YES;

          UINavigationController *nc =
          [[UINavigationController alloc]
          initWithRootViewController:contactVC];

          [self.navigationController
          presentModalViewController:nc animated:YES];

       // initiate the selected action
       } else if(buttonIndex == 1) {
          // initiate a phone call
          if([action isEqualToString:@"Call"]) {

             NSString *phone = [NSString
                                stringWithFormat:@"tel:%@",
                                contact.phoneNumber];
```

```
                [[UIApplication sharedApplication]
                 openURL:[NSURL URLWithString:phone]];

            // start an email
            } else if([action isEqualToString:@"Email"]) {

                NSString *email = [NSString
                                   stringWithFormat:@"mailto:%@",
                                    contact.emailAddress];

                [[UIApplication sharedApplication]
                openURL:[NSURL URLWithString:email]];
            }
        }
    }
        onCancel:^()
    {
        // don't do anything for cancel
    }]; });
    // reset the application badge to 0
    [UIApplication sharedApplication].applicationIconBadgeNumber = 0;
}
```

应用现在包含对远程通知的自定义支持，如果用户在使用应用时接收到通知，就表明应用也提供了增强的用户体验。在测试设备上安装应用，发出几条测试通知，类似于之前确认通知成功注册到 APN 那样。如果应用没有处于活动状态，那么你应该看到标准的警告，类似于图 10-13。然而，当应用处于活动状态时，用户看到的警告应该类似于图 10-14。

图 10-14

10.3　理解通知最佳实践

下面是在实现通知时要谨记的一些最佳实践。它们会增强总体的用户体验，同时又不会丧失对用户的吸引力及效率。在某些情况下，它们可以帮助你更好地实现目标。

- 只在需要时请求许可：应用在首次启动时大有淹没用户的趋势，它们会提示用户需要访问诸多服务的许可，如位置(GPS)、远程通知和联系人等。这让人感到厌烦，会导致好评率降低。你应该只在需要发送远程通知时请求许可。比如，如果提供远程通知作为付费升级，那么应该在用户发起升级过程时才请求发送远程通知的许可。
- 确保远程通知值是清晰的：如果用户允许发送远程通知，那么他们必须能理解将会接收到的值。在理想情况下，这种价值定位要早于请求许可。
- 限制远程通知的频率：通知的目的在于告诉用户发生了某个事件，以及在应用中可以采取哪些动作，比如朋友请求等。为应用与内容设定有意义的发送节奏。用户希望每次接收到消息时消息应用都能够发送一条通知；然而，用户希望每日交易应用每天只发送一条通知。
- 允许用户配置通知体验：用户可以定义自己的体验。允许用户配置接收的通知类型(比如朋友请求和新消息)、通知中包含的内容(比如只显示新消息还是整条消息的内容)，以及设置请勿打扰时间窗，在这期间是不会接收通知的。在 iOS 6 中，Apple 提供了原生的请勿打扰特性。如果启用，那么该特性会应用系统范围的请勿打扰时间窗，在这期间所有通知都会静默。不过，用户可配置的设置不应该随意扩展。比如，特定的银行帐户信息和其他个人身份信息就不应该通过通知发送。
- 支持多台设备：一台设备上的通知产生的动作应该反映到该用户的其他所有设备。比如，如果用户注册了两台设备，你向每台设备推送了数量为 3 的未读消息数。如果用户在一台设备上阅读了一条消息，那么每台设备上的未读消息数都应该更新以反映出新的未读消息数 2。这需要在服务器端进行一些额外的处理，并且在应用中做规划，这有助于提升总体的用户体验。
- 开发应用内的通知中心：应用内的通知中心向用户提供中心化的位置来查看重要更新。远程通知是不可靠的。设备可能没有连接到 Wi-Fi 或蜂窝网络，当设备重新连接时，对于每个应用来说，APN 中只有一条待发送的通知。在启动时能够获取已发送通知的通知中心，这提供了一种简单的机制来确保用户会接收到更新。这对于企业应用来说颇具价值，用户可以通过远程通知获悉工作流程。
- 轮询反馈服务：当用户从设备上删除应用，远程通知尝试发送时，操作系统会向 APN 报告不可达的错误。为了限制不必要的失败，Apple 提供了反馈服务，这是个二进制接口，类似于注册远程通知的接口，位于 feedback.push.apple.com:2196 (feedback.sandbox.apple.com:2196 提供了沙箱访问)。反馈服务为送达失败的每个应

用设备令牌提供了一个列表。作为提供者，你应该周期性地查看反馈服务并据此更新数据库。

- 度量通知的成功：监控通知发送的频率以及通知驱动应用启动的次数。考虑指定通知的类型或使用自定义的通知负载值，从而更好地了解哪些消息最有效。

10.4 小结

本地通知与远程通知提供了一种特别的通道来吸引用户并提升业务效率。本地通知在驱动用户响应本地提醒方面非常有效，部署后几乎不需要什么额外的代价与管理成本。远程通知则提供了最棒的灵活性与外部集成，不过相对于本地通知来说，需要额外的配置和维护。根据需求的不同，可以同时使用本地通知与远程通知来优化成本和功能。

下一章将会介绍如何通过 URL 模式与 Keychain 等技术实现安装在设备上的应用之间的通信。

第Ⅳ部分

应用间网络通信

- 第 11 章　应用间通信
- 第 12 章　使用 Game Kit 实现设备间通信
- 第 13 章　使用 Bonjour 实现自组织网络

第11章

应用间通信

本章内容

- 注册自定义的 URL 方案
- 根据其他已安装的应用改变行为
- 利用企业的单点登录
- 检测与重用之前安装程序的数据

本章代码下载

可以在 www.wrox.com 的 Download Code 选项卡中下载本章代码，地址是 www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html。代码位于第 11 章的下载压缩包中，并且根据章节名字单独命名。

掌握传统的网络通信形式后，现在自然而然地会想到设备上的其他应用，想知道该如何与它们进行交互。iOS 应用的沙箱操作模型限制了本章将要介绍的应用间通信的能力，不过通过一些创造性的思维，可以实现比想象中更多的东西。最直接的方式就是在应用中实现一个或多个 URL 方案。应用可以通过 URL 方案感知到其他应用的存在，并且执行特定的动作。此外，还有一种间接的方式——使用共享的钥匙串作为一种通用的键值存储，针对同一开发者开发一组相关应用。本章为这两种方式提供了具体示例。大多数示例都针对 Facebook 和 Twitter，不过也提供了通用的实现，可用于其他应用的自定义方案。

11.1 URL 方案

URL 方案有 3 个主要用途：根据设备上其他应用的存在与否调整逻辑、切换到其他应用以及响应打开你的应用的其他应用。你还可以通过 URL 方案从某个站点或是在基于 Web

的认证流程结束时打开应用。本节将会通过一系列应用阐述自定义 URL 方案的基本用法。每个应用都实现了自己的方案，整个套件会通过配置将套件中安装的所有应用的功能包含进来。代码示例介绍了一些最佳实践，包括实现与响应自定义方案、感知安装在设备上的其他应用，以及在应用间发送序列化的数据。考虑到在大多数情况下，同一时刻只有一个应用会处于活动状态，因此数据主要是单向流动，接收数据的应用会变成活动状态以处理数据。

11.1.1　实现自定义的 URL 方案

实现自定义 URL 方案的第一步是确定应用的哪些特性对于其他应用来说是有用的，其他应用会调用该特性。比如，购物应用会根据给定的 UPC 代码显示出产品的信息，消息应用会预先装配好收件人与消息。让其他应用能够链接到你的应用中访问人数最多的视图也是很有用的，对于选项卡应用来说，这常常是每个选项卡的根视图。为此，在响应进来的 URL 请求时，你应该为每个视图或特性指定一个短的标识符，并创建一个短的名字作为 URL 标识符(比如 http 或 telnet)。确保这个短名是唯一的，因为操作系统如何处理实现了相同方案的多个应用是未定义的。

每个应用都在 Info.plist 中指定其 URL 方案，在安装时会将其注册到操作系统。比如，名为 Acme Employee Directory 的应用可能会在其 Info.plist 中使用方案 acme-directory。图 11-1 展示了 Xcode 的 plist 编辑器中所需的键与值。

Key	Type	Value
Localization native development region	String	en
Bundle display name	String	Directory
Executable file	String	${EXECUTABLE_NAME}
▶ Icon files	Array	(0 items)
Bundle identifier	String	com.acme.${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	String	6.0
Bundle name	String	${PRODUCT_NAME}
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
▼ URL types	Array	(1 item)
▼ Item 0	Diction...	(2 items)
URL identifier	String	com.acme.directory
▼ URL Schemes	Array	(1 item)
Item 0	String	acme-directory
Bundle version	String	1.0
Application Category	String	
Application requires iPhone environmer	Boolean	YES
▶ Required device capabilities	Array	(1 item)
▶ Supported interface orientations	Array	(1 item)
Icon already includes gloss effects	Boolean	YES

图 11-1

当操作系统遇到 acme-directory://URL 时，会调用应用委托中两个方法中的一个，具体调用哪一个取决于应用当前的状态。如果应用不在运行，那么它就会启动，application:didFinishLaunchingWithOptions:有一个 options 字典，里面包含了键 UIApplicationLaunchOptionsSourceApplicationKey 与 UIApplicationLaunchOptionsURLKey，它们分别用来标识调用应用与完整 URL。如果应用确定它可以处理给定的 URL，那么它就会从 application:didFinishLaunchingWithOptions:返回 YES。接下来，操作系统会调用应用

委托的 application:openURL:sourceApplication:annotation:以对 URL 进行实际处理。当应用进入前台时，操作系统会显示其默认图片(通常名为 Default.png)。此外，如果应用在后台运行或是被挂起，那么它就会继续，并且只会调用 application:openURL:sourceApplication:annotation:来处理 URL。图 11-2 展示了两条可能的执行路径。

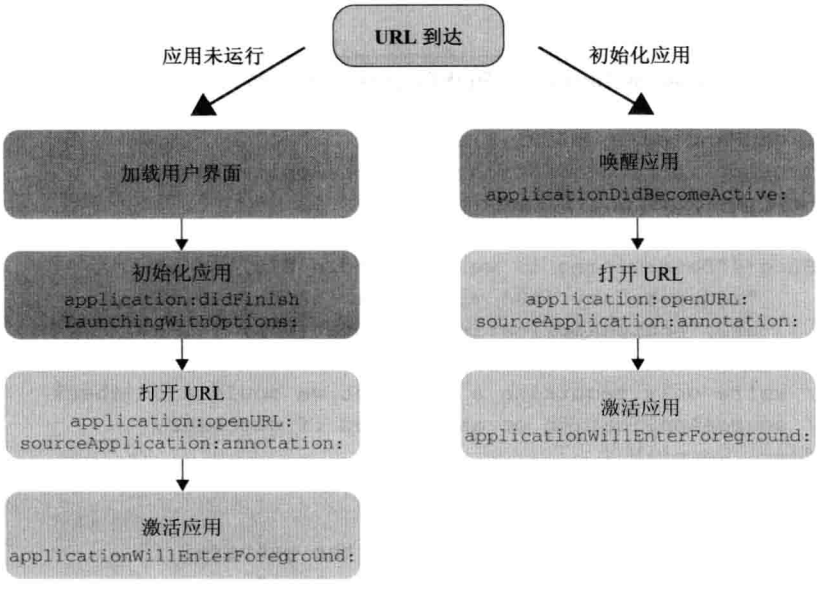


图 11-2

通常情况下，由于URL方案显示的视图与应用启动时显示的视图不同，因此可以为所要实现的每个URL模式定义各种默认图片。每一种都遵循着特定的格式，包含必填值与可选值。

- UILaunchImageFile 是必需的，如果不在 Info.plist 中指定，就要将其设为 Default。
- URL 方案字符串是之前在 Info.plist 中指定的 acme-directory 值，它也是必需的。
- orientation 值是可选的，不过可以通过 Portrait 或 Landscape 根据设备当前的方向显示启动图片。
- scale 是个可选值，大多数开发者对此都应该很熟悉了，可以通过@2x 为 Retina 设备指定图片。
- 最后一个可选值是 device，如果取值为 iphone，那么它会显示根据手机风格定制的图片；如果取值为 ipad，那么它会显示根据平板风格定制的图片。

就拿“员工名册”(Employee Directory)这个应用作为示例来说明一下 device 值。该应用包含了在非 Retina 与 Retina 显示的手机风格的设备上针对其 URL 方案的自定义图片，文件名分别为 Default-acme-directory-portrait@2x.png 与 Default-acme-directory-portrait.png。下述代码片段完整展示了针对某个具体方案的默认图片名。花括号中的值是必需的，方括号中的值是可选的：

```
{UILaunchImageFile value}-{URL scheme string}[-orientation]
                                                [scale][~device].png
```

决定是否能够处理某个 URL 在很大程度上取决于具体的应用；然而，要想快速检查，可以验证该 URL 是不是在请求之前选择的某个特性标识符。大多数特定的边界检测、模式匹配以及其他一些验证也是可行的。在代码清单 11-1 展示的 Acme Employee Directory 中，程序会检查 URL 以确保使用了自定义方案 acme-directory、请求标识符 employee，并确保提供了一个已经存在的员工标识符。

代码清单 11-1　验证 Acme Directory URL(EDAppDelegate.m)

```
- (BOOL)canHandleURL:(NSURL *)url
fromSourceApplication:(NSString *)sourceApplication
       withAnnotation:(id)annotation {

    NSLog(@"Determining if we can handle URL '%@' "
          "requested by '%@' with data '%@'",
          url, sourceApplication, annotation);

    // we're only requiring a URL, but we could also check
    // sourceApplication or annotation if necessary
    if(url == nil) {
        return NO;
    }
    // we'll only respond to URLs of the pattern:
    // "acme-directory://employee/{integer}"
    if([[url scheme] isEqualToString:@"acme-directory"]) {
        NSString *viewIdentifier = [url host];

        // there is only one view identifier we'll accept
        if([viewIdentifier isEqualToString:@"employee"]) {
            NSInteger employeeNumber = [self employeeNumberFromURL:url];
            EDEmployee *employee = [[EDEmployeeManager sharedManager]
                                    employeeForId:employeeNumber];

            // if we got an employee, then accept the URL
            if(employee != nil) {
                return YES;
            }
        }
    }

    return NO;
}
```

如果应用要响应多个方案、标识符或是需要更长的路径，那么可以很轻松地扩展示例的 canHandleURL:fromSourceApplication:withAnnotation:方法。要注意 URL 处理的一处不太直观的地方：应用即便从 application:didFinishLaunchingWithOptions:返回 NO，也仍会被启动并进入到前台，这是因为应用注册了该 URL 方案。

11.1.2　感知其他应用的存在

现在，各种各样的应用都会加入一些社交网络特性来吸引用户或是传播服务与产品。在很多情况下，开发者会将 Twitter 与 Facebook 加进来，对于所有用户来说，其社交特性的使用方式是一样的。不过，虽然用户可以通过一个移动应用来访问这两个网络，但他们却被迫使用不熟悉的用户界面，在使用所需的特性前还常常要对应用进行授权才行。这种额外的阻力会对社交特性造成不好的影响，还会影响应用的流行度。这个问题的解决之道是检测如果设备上安装了原生应用，那么调用应用的自定义 URL 方案。大多数社交或消息应用都为这种情况提供了 URL 方案。这样，用户就可以在客户端使用熟悉的用户界面，并且服务也已经授权了。

为了感知其他应用的存在，首先需要知道目标应用提供的自定义 URL 方案。很多开发者都会在 API 文档中或是网站的开发者页面上提供关于其自定义方案的详细信息。知道方案名后，实现特性就是相对简单的事情了，只需使用 UIApplication 的 canOpenURL:方法即可。如果应用已安装并且在给定的 URL 中注册了方案，那么该方法会返回 YES，否则返回 NO。比如，要想测试 Facebook 或 Twitter 存在与否，那么只需看看 canOpenURL:针对下面这两个 URL 方案是否返回 YES 即可：

```
if([[UIApplication sharedApplication] canOpenURL:
    [NSURL URLWithString:@"twitter://"]]) {

    // this device has the Twitter for iPhone application
}

if([[UIApplication sharedApplication] canOpenURL:
    [NSURL URLWithString:@"fb://"]]) {

    // this device has the Facebook application
}
```

在检测其他应用存在与否后，就可以调整用户界面或功能以匹配其他应用了。比如，如果应用没有找到，那么你可能会禁用分享特性，或是采用别的方式来实现。有些应用会注册两个类似的 URL 方案，不过第 2 个 URL 方案会将当前版本附加到方案的末尾(比如 acme-directory-1-0://表示 1.0 版)。如果应用想要使用只在目标应用的某些版本中才有的 URL 方案特性，那么应用就不仅要检测目标应用存在与否，还需要检测特定的版本是否存在。

如果相同的团队或是彼此之间协作的团队开发出很多相互关联的应用，那么检测已安装的应用就变得非常有意思了。在企业环境中，很多公司都会在内部部署多个应用，每个应用都有特定的焦点或目标群体。如果知道存在着可以发送额外一些值的应用，那么这些应用就可以开启额外的特性或外部钩子(hook)了。

前面介绍的“员工名册”应用就可以进行扩展以对安装了另一示例应用 Employee Records 的用户加入一些钩子。在这种情况下，Employee Records 只会发送给 HR，因为其中包含敏感的个人信息。每个应用都可以打开其他应用的员工详细信息页面，只要这些应

用安装了就行。

在这个名录应用中，拥有Employee Records的用户会在Companion Apps表格下方看到相应的条目，如图 11-3 所示。这部分是由代码清单 11-2 中的dictionaryOfInstalledCompanionApps装配而成的，它只是检查了每个URL方案。Employee Directory与Employee Records也通过类似代码实现了相同的功能。

图 11-3

代码清单 11-2　检测伙伴应用(EDUtils.m)

```
+ (NSDictionary*)dictionaryOfInstalledCompanionApps {
    NSMutableDictionary *companionApps = [NSMutableDictionary dictionary];

    // employee records
    if([[UIApplication sharedApplication] canOpenURL:
        [NSURL URLWithString:@"acme-records://"]]) {

        [companionApps setValue:@"acme-records://employee/"
                         forKey:@"Employee Records"];
    }

    return companionApps;
}
```

11.1.3　高级通信

上述示例只使用纯文本将信息传递给了目标应用，不过可以在URL中使用几乎任何序列化对象。可以对Employee Records应用示例进一步扩展，从其他应用接收图片，然后将其添加到员工的文件中。可以序列化为NSData的任何对象都可以通过自定义URL方案发

送。对于大多数常见的iOS类型以及所有的原生类型来说，可以通过NSKeyedArchiver提供的方法来创建封装好的数据对象。对象是通过archivedDataWithRootObject:序列化的，原生类型是通过encode{type}:forKey:形式的方法来序列化的。NSDictionary对象与UIImage对象有自己专门的自定义序列化器。要想序列化自定义对象，该对象必须实现NSCoding与initWithCoder:和encodeWithCoder:这两个方法。在将对象传递给archivedDataWithRootObject:后，NSKeyedArchiver会自动使用这两个方法。表 11-1 展示了常见的iOS对象类型与各自的序列化器的对应关系。

表 11-1　常见 iOS 类型的序列化器

类　型	序 列 化 器
NSDictionary	NSPropertyListSerialization
UIImage	UIImageJPEGRepresentation()或 UIImagePNGRepresentation()
遵循 NSCoding 的对象	NSKeyedArchiver
原生类型	NSKeyedArchiver

在将对象表示为 NSData 对象后，需要将其转换为字符串以通过 URL 进行传递。我们这里当然会使用 NSString 的 initWithData:encoding:方法；不过，结果字符串放在 URL 中可能会不安全。RFC 3986(http://tools.ietf.org/html/rfc3986)定义了可以包含在 URL 中的有效字符，如表 11-2 所示。

表 11-2　有效的 URL 字符

字　符	描　述	字　符	描　述
A-Z	大写字母	@	at 符号
a-z	小写字母	!	感叹号
0-9	数字	$	美元符号
-	连字符	&	连接符
.	句点	'	单引号
_	下划线	)或(	左或右圆括号
~	波浪线	*	星号
:	冒号	+	加号
/	正斜杠	,	逗号
?	问号	;	分号
#	井号或哈希	=	等号
]或[	左或右方括号		

根据这套字符集，Internet 工程任务组在 RFC 4648(http://tools.ietf.org/html/rfc4648)中标准化了一种名为 base64 的编码，并专门用于将二进制数据(在该示例应用中就是图片)表示为 ASCII 文本字符串。可以使用一个标准化的转换表格将二进制数据的每 6 位编码为一个

字符，如表 11-3 所示。

表 11-3　base64 转换表格

6 位数值	编码字符	6 位数值	编码字符	6 位数值	编码字符	6 位数值	编码字符
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

虽然 base64 编码的字符串是完备的，不过它依然不太适合于包含在 URL 中。字符 62(加号)与 63(正斜杠)在 URL 中都有特殊的含义，可能会导致接收应用无法正确解码二进制数据。在查询字符串中，加号表示空白字符，如果数据被解释为空白，解析器就会过早终止或解释不正确的数据。正斜杠字符用于表示 URL 路径的部分，可能会导致解析器过早开始或结束。考虑这样一种情况，二进制数据的开头被编码为 employee/5/。当传递给 Employee Records 时，URL 可能会被解析为 acme-records://employee/5/...，这会打开第 5 个员工的详细信息视图而非根据期望解析完整的二进制数据。

对于 base64 编码字符串中的+与/字符，有两种解决方案可以防止出现意外后果。最直接的方式就是通过 URL 编码 base64 字符串。用到网络通信的很多应用都提供了辅助方法来将 URL 编码为字符串，这里也可以重用这个方案。如果应用没有提供，就要小心提防 NSString 的 stringByAddingPercentEscapesUsingEncoding:，因为 Apple 是这样描述这个方法的：

... 对接收者的一种表示，使用给定的编码确定百分比符号的转义，用于将接收者转换为合法的 URL 字符串。

回忆一下，加号与正斜杠都是合法的 URL 字符；因此，它们是不会被

stringByAddingPercentEscapesUsingEncoding:方法编码的。当给定一个 base64 编码的字符串时，stringByAddingPercentEscapesUsingEncoding:实际上会返回不经改变的相同字符串。

第 2 个解决方案使用名为 base64url 的 base64 变体，它就是用于解决这个问题的。它使用修改版的转换表格，将加号替换为连字符(-)，将正斜杠替换为下划线(_)。表 11-4 展示了修改版的转换表格。

表 11-4　base64url 转换表格

6 位数值	编码字符	6 位数值	编码字符	6 位数值	编码字符	6 位数值	编码字符
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	-
15	P	31	f	47	v	63	_

第 4 个示例应用 Employee Records Image Adder 通过 Employee Records URL 方案中的第 2 个标识符向员工添加了一张图片。比如，要向第 1 个员工添加一张图片，应用应该打开 acme-records://addimage/1/{image data string}。要想创建图像数据字符串，需要使用之前介绍的 3 步过程。序列化图片，使用 base64 对其进行编码，然后使用 URL 再次进行编码。代码清单 11-3 使用 NSData 的 base64 编码类别，NSData 提供了 base64EncodingWithLineLength:；不过，任何 base64 实现都会这么做。接收应用会按照相反的顺序执行同样的 3 步来恢复出 UIImage 对象。

代码清单 11-3　在自定义 URL 方案中传递一张图片(IAViewController.m)

```
// encode the image as data
NSData *imageData = UIImagePNGRepresentation(imageView.image);
```

```
// turn the data into a string by base64-encoding it
NSString *imageString = [imageData base64EncodingWithLineLength:0];

// url-encode the base64 string
NSString *encodedString = [IAUtilsencodeURL:imageString];

// create and open the URL
NSURL *url = [NSURL URLWithString:[NSString stringWithFormat:
                @"acme-records://addimage/%@/?%@",
                employeeNumberField.text,encodedString]];
[[UIApplication sharedApplication] openURL:url];
```

如果想要打开接收应用，那么 URL 方案是非常不错的选择，而且任意大小的数据都可以通过这种方式传递，不过用户体验可能会受到影响，因为验证的时间和打开长 URL 的时间会变得比较可观。除此之外，还有一些技术提供了共享数据的能力，同时又将焦点保持在你的应用之上。

11.2　共享钥匙串

共享钥匙串对于企业来说特别有用，因为它会创建一块公共区域，共享同一 Bundle Seed ID 的所有应用都可以访问这块区域。通过这块共享空间，我们可以非常轻松地为一组相关应用实现单点登录(Single Sign-On，SSO)认证系统。此外，在钥匙串中存储数据的应用会检测自身之前的安装，这样就可以通过重用之前提供的认证信息或是针对期望的用户调整用户界面来改善用户体验。

iOS 钥匙串针对受保护的操作系统数据(如 Wi-Fi 密码或账户信息等)提供了单独的区域来实现安全存储。第三方应用也可以通过该存储来保存类似的受保护数据。要想保证安全性，钥匙串条目应总是在加密之后保存在磁盘上和设备备份中。即便删除应用，受保护的数据也依然会留在钥匙串中，这样后面再次安装应用时就可以重用相同的信息了。同一开发者所开发的多个应用可以通过配置使用相同的加密密钥，这样每个应用就都可以访问共享的钥匙串条目了。本节的代码示例为高级特性提供了一个模板，比如为一系列应用实现 SSO，以及检测应用之前的安装。

11.2.1　企业 SSO

单点登录功能对于通过桌面或 Web 分发的大多数企业应用来说都是必不可少的，这是因为它增强了安全性并为用户提供了便利。随着内部部署的应用数量的不断增加，SSO 在移动环境中将会变得越来越重要。SSO 常常被实现为一个端到端的认证框架，可以让用户通过一套共享的认证信息对多个应用进行认证。如果登录成功，那么 SSO 提供者通常会生成一个认证令牌，令牌会存储起来并用于对所有后续请求进行签名。如果应用的安全需求想到要共享该令牌，那么可以将其安全地存储到共享钥匙串中，并对其他应用可用，而无须用户再次登录这些应用。如果无法共享令牌，那么用户的账号标识符或 Email 地址依然

可以保存起来，这样也可以在一定程度上提升效率。

项目要想能使用共享钥匙串以实现 SSO，就必须完成 3 个准备步骤：

(1) 每个应用都要共享一个公用的 Bundle Seed ID，这个值是在创建应用时在 iOS Provisioning Portal 中设置的。图 11-4 展示了 Provisioning Portal 中一个示例应用的条目。

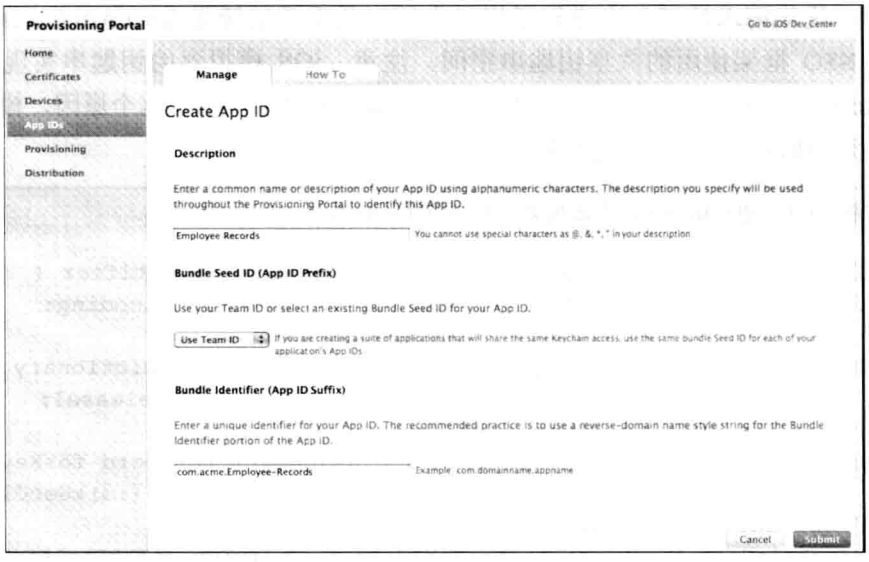


图 11-4

(2) 每个项目需要引入一个Entitlements.plist文件，该文件在一个或多个keychain-access-groups中指定了Bundle Seed ID。图 11-5 展示了在Xcode的plist编辑器中所需的键值。

Key	Type	Value
▼keychain-access-groups	Array	(2 items)
Item 0	String	$(AppIdentifierPrefix)com.acme.Employee-Records
Item 1	String	$(AppIdentifierPrefix)com.acme.sso

图 11-5

(3) 每个项目都需要引入 Security.framework 才能编译通过。为了将其引入进来，请执行如下步骤：

A. 在项目导航器中，选中项目。

B. 选中目标。

C. 选择 Build Phases 选项卡。

D. 展开 Link Binaries with Libraries。

E. 单击+按钮。

F. 搜索并添加 Security.framework。

既然配置好了项目，现在是时候开发一些底层的钥匙串辅助方法来驱动应用的上层功能了。由于这个辅助类将会包含在每个使用了 SSO 的应用中，因此其实现应该与任何一个应用的实现细节解耦。理想情况下，应该将其编译到一个静态库中，这样只需很少的配置或是根本无需配置就可以将其引入到应用中了。

大多数钥匙串的创建、读取、更新与删除(CRUD)操作都使用共同的配置参数，参数位于 NSDictionary 对象中，包含特定的键的集合。代码清单 11-4 包含名为 keychainSearch: 的创建方法，该方法使用共享的配置创建字典对象。每个操作都会在某个时刻调用 keychainSearch:来初始化钥匙串搜索上下文。辅助方法需要引入很多键，每个键的含义都在第 7 章“优化请求性能”中进行了详尽介绍。最重要的键是 kSecAttrAccessGroup，它定义了 SSO 框架使用的共享钥匙串空间。注意，iOS 模拟器的钥匙串实现并不支持 kSecAttrAccessGroup，需要将其放到针对 iOS 设备的编译中。由于这个原因，模拟器上的所有应用都能访问到所有的钥匙串条目。

代码清单 11-4　初始化一个钥匙串搜索字典(SSOUtils.m)

```
+ (NSMutableDictionary*)keychainSearch:(NSString*)identifier {
    NSData *encodedIdentifier = [identifier dataUsingEncoding:
                                 NSUTF8StringEncoding];
    NSMutableDictionary *keychainSearch = [[[NSMutableDictionary alloc]
                                           init] autorelease];
    // set the type to generic password
    [keychainSearch setObject:(id)kSecClassGenericPassword forKey:
                                                    (id)kSecClass];
    // set the item's identifier
    [keychainSearch setObject:encodedIdentifier forKey:(id)kSecAttrGeneric];
    [keychainSearch setObject:encodedIdentifier forKey:(id)kSecAttrAccount];

    // use the shared keychain
    // note: not supported in the simulator and will cause
    // all keychain calls to fail
    #if !(TARGET_IPHONE_SIMULATOR)
        [keychainSearch setObject:kKeychainSSOGroup
                          forKey:(id)kSecAttrAccessGroup];
    #endif

    return keychainSearch;
}
```

为了读取现有的钥匙串值，代码清单 11-5 实现了 getValueForIdentifier:，它返回一个钥匙串值，使用一个标识符作为值的键。它调用 keychainSearch:来初始化配置字典，然后设置两个额外的参数。

代码清单 11-5　获取一个钥匙串条目(SSOUtils.m)

```
+ (NSString*)getValueForIdentifier:(NSString*)identifier {
    NSMutableDictionary *search = [self keychainSearch:identifier];

    // limit to the first result
     [search setObject:(id)kSecMatchLimitOne forKey:(id)kSecMatchLimit];

    // return data vs a dictionary of attributes
```

```
    [search setObject:(id)kCFBooleanTrue forKey:(id)kSecReturnData];

    // perform the search
    NSData *value = nil;
      OSStatus status = SecItemCopyMatching((CFDictionaryRef)search,
                                            (CFTypeRef *)&value);

    if(status == noErr) {
        return [NSString stringWithUTF8String:[value bytes]];
    }

    return nil;
}
```

代码清单 11-5 为 kSecMatchLimit 指定值 kSecMatchLimitOne 来确保钥匙串返回找到的第一个结果。要想返回多个结果，可以传递一个 CFNumberRef，指定所要返回的最大结果数，或是传递 kSecMatchLimitAll 来返回所有可能的结果。该方法还将 kSecReturnData 设为 kCFBooleanTrue 来告诉钥匙串返回所需条目值的原始数据。还可以将其他键设为 kCFBooleanTrue 来返回其他类型的数据：

- kSecReturnAttributes 返回条目的一个特性
- kSecReturnRef 返回条目的一个引用
- kSecReturnPersistentRef 返回条目的一个持久化引用

如果多个类型都为真，那么钥匙串会返回请求信息的一个字典。查询本身是通过调用 SecItemCopyMatching()来处理的，它接收配置参数，然后执行搜索。搜索的结果会被复制到&value 提供的引用中，返回的 OSStatus 表示搜索的状态。表 11-5 列出了可能的状态码。辅助方法最后返回了返回值的字符串值，如果出现错误，则返回 nil。

表 11-5　钥匙串搜索状态码

返回码常量	说　明
errSecSuccess	没有错误
errSecUnimplemented	功能或操作没有实现
errSecParam	传递给函数的一个或多个参数不合法
errSecAllocate	分配内存失败
errSecNotAvailable	没有可信任的结果
errSecAuthFailed	授权或认证失败
errSecDuplicateItem	条目已经存在
errSecItemNotFound	无法找到条目
errSecInteractionNotAllowed	与安全服务器的交互不允许
errSecDecode	无法解码所提供的数据

代码清单 11-6 展示了另一个辅助方法 setValue:forIdentifier:，它实现了钥匙串条目的创

建与更新操作。这两个操作存在单独的钥匙串方法，你的实现要能确定哪一个是适合的。它会调用 getValueForIdentifier:以确定指定标识符的条目是否已经存在。如果存在，那么方法会进入更新逻辑分支。如果新的值与现有的值不同，那么它会被转换为 NSData 对象，然后插入到 update 参数 NSDictionary 中，键则为 kSecValueData。该字典还可以包含很多其他的属性，完整列表可在 Apple 的“钥匙串常量”文档中查阅。如果与给定标识符对应的条目不存在，那么方法就会执行 else 分支，这会创建该条目。类似于更新，它会将条目的 NSData 表示设为键 kSecValueData；然而，该键值对会被插入到搜索字典而非全新的字典中。该方法会将 NULL 传递给 SecItemUpdate()的 result 参数，这是因为它不需要保留对新插入条目的引用。如果操作成功，这两个分支代码都会返回 YES，否则返回 NO。

代码清单 11-6　创建或更新钥匙串条目(SSOUTILS.M)

```
+ (BOOL)setValue:(NSString*)value forIdentifier:(NSString*)identifier {
    NSString *existingValue = [self getValueForIdentifier:identifier];

    // check if value exists
    if(existingValue) {

        // update if the new value is different
        if(![existingValue isEqualToString:value]) {
            NSMutableDictionary *search = [self keychainSearch:identifier];

            NSData *valueData = [value dataUsingEncoding:NSUTF8StringEncoding];
            NSMutableDictionary *update = [NSMutableDictionary
                dictionaryWithObjectsAndKeys:valueData,
                (id)kSecValueData, nil];

            OSStatus status = SecItemUpdate((CFDictionaryRef)search,
                                            (CFDictionaryRef)update);

            if(status == errSecSuccess) {
                return YES;
            }

            return NO;
        } else {
            return YES;
        }

    // if no value exists, create a new entry
    } else {
        NSMutableDictionary *add = [self keychainSearch:identifier];

        NSData *valueData = [value dataUsingEncoding:NSUTF8StringEncoding];
        [add setObject:valueData forKey:(id)kSecValueData];

        OSStatus status = SecItemAdd((CFDictionaryRef)add,NULL);
```

```
        if(status == errSecSuccess) {
            return YES;
        }

        return NO;
    }
}
```

代码清单 11-7 展示了最后一项 CRUD 操作——删除，可以通过已有的 keychainSearch: 方法轻松实现。钥匙串方法 SecItemDelete()可以搜索匹配搜索字典的所有条目，并从钥匙串中将其删除。如果删除操作成功，那么辅助方法会返回 YES，否则返回 NO。

代码清单 11-7　删除钥匙串条目(SSOUtils.m)

```
+ (BOOL)deleteValueForIdentifier:(NSString*)identifier {
    NSMutableDictionary *search = [self keychainSearch:identifier];

    OSStatus status = SecItemDelete((CFDictionaryRef)search);

    if(status == errSecSuccess) {
        return YES;
    }

    return NO;
}
```

既然应用已经有了一些用于钥匙串交互的构建块，用于 SSO 实现的上层功能就可以通过它们来实现了。最重要的是 authenticateWithUsername:andPassword:，它会执行认证检测，并将结果令牌保存到钥匙串中。代码清单 11-8 展示的示例可以接收任意的用户名/密码组合；然而，真实的企业实现总是需要向 SSO 提供者发出网络调用。SSO 提供者应该返回一个加密的安全认证令牌，该令牌会在指定的时间间隔后过期。应用应该将该令牌存储起来供后续的网络请求使用，还可以存储过期时间(如果有的话)。

代码清单 11-8　使用用户名与密码进行认证(SSOUtils.m)

```
+ (BOOL)authenticateWithUsername:(NSString *)username
                     andPassword:(NSString *)password {

    // you should do a check of the given credentials here
    // this dummy application will always return a successful login
    BOOL loginResult = YES;

    if(loginResult == YES) {
        // set SSO username
          [self setValue:username forIdentifier:kCredentialUsernameKey];

        // set SSO token
```

```
        [self setValue:@"SSOValidToken" forIdentifier:kCredentialTokenKey];
    }

    return loginResult;
}
```

接下来，安全应用中的视图应该在请求或显示敏感数据前调用credentialsAreValid，它通常位于视图控制器的viewWillAppear:方法中或是位于应用委托的applicationWillEnterForeground:方法中。代码清单 11-9 验证保存的认证令牌并通知应用可以继续显示视图或是请求用户重新进行认证。该示例实现做了一次假的检查，严格的检查可能需要向SSO提供者发出网络调用或是需要客户端进行加密检测。

代码清单 11-9 验证存储的认证信息(SSOUtils.m)

```
+ (BOOL)credentialsAreValid {
    NSString *token = [self credentialToken];

    if(token == nil) {
        return NO;
    }

    // you should do a secure check of the token here
    // we'll do a dummy check to make sure it matches our
    // secret value 'SSOValidToken'
    return [token isEqualToString:@"SSOValidToken"];
}
```

最后一个上层操作通过删除存储的认证令牌注销当前用户。代码清单 11-10 展示了一个注销的示例实现。你的实现还可以删除 kCredentialUsernameKey；不过在大多数情况下，使用之前的值预先装配好用户名字段所带来的用户体验上的提升要比潜在的安全弱点更重要一些。

代码清单 11-10 删除存储的认证信息以实现注销(SSOUtils.m)

```
+ (void)logout {
    // destroy the saved token
    [self deleteValueForIdentifier:kCredentialTokenKey];
}
```

有了这个基础，企业应用就可以实现 SSO 系统来改进相关应用的用户体验及效率了。随着移动应用在企业中变得越来越流行，共享钥匙串编程的经验将成为每一个 iOS 开发者的必备技能。

11.2.2 检测应用之前的安装

通过将持久化与临时存储相组合，应用可以检测到自身之前的安装，并初始化用户界面来帮助那些回归的用户。该功能最常见的使用场景是记住之前的认证值，并使用现有的用

户名与密码来预先装配应用的一部分。对于使用之前介绍的 SSO 模式的应用来说，该功能几乎是必不可少的。安装过的应用还可以跳过介绍性的特性导览，让用户能够尽快上手使用。

本节所要介绍的示例应用会将用户的生日记录到钥匙串中，然后在后面应用启动时将其加载进来。如果删除并重新安装应用，那么会询问用户是使用之前保存的生日还是将其丢弃。代码清单 11-11 展示了视图控制器的 viewDidLoad 方法，如果应用不是首次启动，那么会加载之前保存的生日(如果有的话)，然后将其装配到文本域中。虽然该方法还会检查这是否是新安装应用的首次启动，不过这并不是必需的，只是为了防止在用户响应 UIAlertView 之前就填充生日域。

代码清单 11-11　根据应用之前的安装改变用户界面(IDViewController.m)

```
- (void)viewDidLoad {
    // check for a previous installation
    NSString *savedBirthday = [IDUtils savedBirthday];
    BOOL firstLaunch = [IDUtils isFirstLaunch];

    if(savedBirthday != nil &&firstLaunch) {
        [[[UIAlertView alloc] initWithTitle:NSLocalizedString(
             @"Previously Saved Birthday",@"Previously Saved Birthday")
          message:NSLocalizedString(@"It appears you saved a birthday
            in a previous installation of this application. Do you
             want to keep it?",@"It appears you saved a birthday
            in a previous installation of this application. Do you
             want to keep it?")
          delegate:self
          cancelButtonTitle:NSLocalizedString(@"Discard", @"Discard")
          otherButtonTitles:NSLocalizedString(@"Keep", @"Keep"),nil]
        show];
    }

    // pre-populate the birthday field if this isn't the first launch
    if(firstLaunch == NO) {
    self.birthdayField.text = savedBirthday;
    }

    // focus the birthday field
    [self.birthdayField becomeFirstResponder];
}
```

代码清单 11-12 中的 savedBirthday 方法只是根据预先确定的键 kKeychainBirthdayKey 来取回钥匙串中的值。如果找不到值，那么可以断定该应用之前从来没有安装过。要想可靠地测试该情况是很困难的，因为每台测试设备只有一次机会。然而，使用 deleteValueForIdentifier:删除键为 kKeychainBirthdayKey 的钥匙串条目会将设备置为预安装状态。要想实现反馈测试，在应用委托的 applicationDidEnterBackground:中调用删除辅助方法，并确保在 Xcode 中停止任务前应用进入了后台。

代码清单 11-12　获取之前保存的生日(IDUtils.m)

```
+ (NSString *)savedBirthday {
    return [self getValueForIdentifier:kKeychainBirthdayKey];
}
```

你应该能够认出 getValueForIdentifier:，我们在之前讨论 SSO 模式时介绍过它，这里重用了它的代码。在代码清单 11-13 中，NSUserDefaults 存储用于实现 isFirstLaunch:。

代码清单 11-13　检测应用的首次启动(IDUtils.m)

```
+ (BOOL)isFirstLaunch {
    BOOL hasBeenLaunched = [[NSUserDefaults standardUserDefaults]
                            boolForKey:kDefaultsHasBeenLaunchedKey];

    if(hasBeenLaunched == NO) {
        // this is the first launch, so set a defaults value
        // saying that we were launched at least once
        [[NSUserDefaults standardUserDefaults] setBool:YES
            forKey:kDefaultsHasBeenLaunchedKey];
        [[NSUserDefaults standardUserDefaults] synchronize];

        return YES;
    }

    return NO;
}
```

如果在当前的安装下应用首次启动，那么该方法会返回YES。与钥匙串不同，当卸载应用时，任何NSUserDefaults条目都会被清空；因此，如果没有找到kDefaultsHasBeenLaunchedKey(boolForKey:返回NO)，那么你就知道它已经被清空了。当首次调用它后，该方法会将YES写到kDefaultsHasBeenLaunchedKey中。

虽然简单，但这个示例应用有助于你了解针对应用之前的用户修改 UI 的细节信息。棘手之处在于确定如何在为老用户保留特性的同时又为新用户保持简洁的用户体验。当然，我们不希望对应用做过大调整，否则代码基将会变得无法管理。

11.3　小结

本章介绍了应用间通信的 4 种不同方式，用于增加功能或是增强 iOS 应用的用户体验。检测其他应用的自定义 URL 方案可以使得开发者在其他应用已安装或是链接到已安装应用的特定视图时能够增加额外的功能。读写共享钥匙串是实现同一开发者或公司开发的应用间通信的一种间接方式。共享钥匙串为那些应用提供公共的键值存储来存储 SSO 或其他持久化数据。应用间通信并非总是发送数据的最优雅或直接的方式，不过在 iOS 应用的隔离世界中，它能提供其他方式无法实现的功能。

第12章

使用 Game Kit 实现设备间通信

本章内容

- 使用 Game Kit 类及配置传输选项
- 理解传统的客户端-服务器通信
- 创建点对点连接

本章代码下载

可以在 www.wrox.com 的 Download Code 选项卡中下载本章代码，地址是 www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html。代码位于第 12 章的下载压缩包中，并且都在一个示例项目内：Game Kit Auctioneer.zip。

到目前为止，我们介绍的所有关于通信的主题都是基于如下假设：设备连接到了网络，同时网络又连接到了 Internet；然而，iOS 设备甚至可以不通过传统网络传输数据。Apple 的 Game Kit 框架可以实现没有网络状况下的设备与设备之间的通信，这包括没有蜂窝服务、无法访问 Wi-Fi 基础设施以及无法访问局域网或 Internet 等情况。比如在丛林深处、高速公路上或是建筑物的地下室等。

虽然名字基本上能够表达出其大多数使用场景，不过 Game Kit 却不仅仅只用于多人游戏。该框架对数据并没有什么要求，应用可以通过各种通信选项发送任意类型的数据。Game Kit 可以在没有任何网络基础设施的情况下用于短程的个人局域网(Personal Area Network，PAN)，也可以用于更加传统的 Wi-Fi 局域网，这一点使得 Game Kit 成为经验丰富的 iOS 开发者的必备工具。本章将会介绍如何初始化网络会话、探测其他设备，以及如何在使用了 Game Kit 的拍卖客户端发送数据包。

12.1　Game Kit 基础

除了底层的网络通信特性外，Game Kit 还包含了一些特定于游戏的技术，比如成就、排行榜和比赛等。图 12-1 概览了 Game Kit 在其他高层通信框架之上所处的位置。

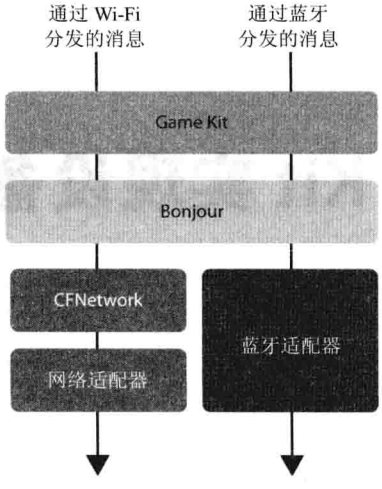


图 12-1

说明：

本章并不会介绍游戏特性，因为它们只能用在 Apple 的游戏中心服务中，并不适合于通用目的的通信。

Game Kit 网络提供了 3 种通信模式，可以用来控制数据消息在网络设备间的流动。这些网络在每台设备上都表示为一个 GKSession 实例，连接的每个设备或端点都通过其端点 ID 进行标识。

- 点对点(P2P)——该模式会平等地对待每个端点，并且将所有消息发送给连接到网络的每台设备。
- 客户端-服务器——在该通信模式下，一个端点会被指定为会话的主机，所有其他端点都是该主机的客户端。网络的数据传输属性保持不变；不过，在客户端-服务器网络中，完成同一任务的端点是看不见其他端点的。
- 基于回合的比赛(turn-based match)——该模式很少用于企业软件，因为每个参与者都要有一个游戏中心帐号才行。由于存在如此巨大的缺陷，本章并不会介绍基于回合的通信。

在这些高层的会话配置中，开发者也可以做一些配置来控制底层网络栈的可用性及行为。会话可用性的控制方式根据使用的用户界面的不同而不同。Apple提供的GKPeerPickerController实现可用于指定是否要使用connectionTypesMask以通过蓝牙和Wi-Fi来搜索端点。如下代码示例展示了这 3 种连接配置。除了可以搜索某一类型的端点外，Apple的实现在蓝牙没有启用时还会提示用户开启蓝牙。由于蓝牙设置并没有公开为API，

因此相同的功能是无法通过自定义的接口来实现的。如果应用要使用这两种连接类型，那么应该在应用的Info.plist的UIRequiredDeviceCapabilities字典中进行说明。要想进行Wi-Fi连接，请将键wifi的值设为YES。要想在iOS 3.1 及之后的版本中进行蓝牙连接，请将键peer-peer设为YES。如果应用使用了自定义UI而非GKPeerPickerController，那么GKSession总是会响应蓝牙与Wi-Fi端点。

```
GKPeerPickerController *picker = [[GKPeerPickerController alloc] init];

// search for only Bluetooth peers
picker.connectionTypesMask = GKPeerPickerConnectionTypeNearby;

// search for only Wi-Fi peers
picker.connectionTypesMask = GKPeerPickerConnectionTypeOnline;

// search for Bluetooth or Wi-Fi peers
picker.connectionTypesMask = (GKPeerPickerConnectionTypeNearby |
                              GKPeerPickerConnectionTypeOnline);
[picker show];
```

蓝牙虽然具有不依赖于网络基础设施的独特优势，不过也有一些较大的缺陷。32 英尺的最大连接范围比 Wi-Fi 网络小太多了，而后者可以跨越多个访问点来覆盖更大的范围。在 Wi-Fi 网络中，每个端点的可用带宽是蓝牙的将近 10 倍。虽然蓝牙无线电要比 Wi-Fi 更省电，但如果所有端点都可以通过这两种类型的连接访问，那么 Game Kit 依然会首选 Wi-Fi 而非蓝牙。虽然只考虑一种连接看起来是错误的，不过实际上这会降低设备总体的电量消耗。连接到 Wi-Fi 网络的 iOS 设备会使用该连接进行后台的所有数据请求，比如周期性的邮件检查或推送通知等。由于将会使用该无线电，因此同时还使用蓝牙无线电实际上会增加电量消耗。

尽管 Apple 表示其“附近”功能实现是基于蓝牙的，不过应用在没有加入 Made for iPhone (MFi)计划前是无法直接控制蓝牙接口的。MFi 用于与外部附件进行连接，如扬声器底座、医用传感器和其他专用硬件等。与之类似，Apple 的 Wi-Fi 通信是通过 Bonjour 实现的，不过应用是无法直接与 Bonjour 服务交互的。要想了解关于 Bonjour 的更多信息，请参考第 13 章“使用 Bonjour 实现自组织网络”。

在连接 Game Kit 类与委托前，消息可靠性是最后一处需要做的重要配置。对于发送单个数据报来说，框架提供了两种可靠性设置，分别是 GKSendDataReliable 与 GKSendDataUnreliable。数据报指的是通过网络发送的消息，由一个或多个数据包组成。如果消息的大小超过 1000 字节，那么数据包中的数据量就会被划分为若干个块，然后单独发送给接收方，接下来再装配成原始消息。由于消息块是单独发送的，因此接收方在处理最终的消息前必须等待所有的数据块都到达才行，这会在很大程度上降低性能。Game Kit 强制限定单个数据报的大小最大为 87KB。使用 GKSendDataReliable 模式发送数据报可以确保出现网络错误时能够重新发送，能够保证消息按照发送的顺序到达。然而，对于 GKSendDataUnreliable 模式来说，数据报只会发送一次，如果没有到达指定的目的地，数

据报就永远丢失了。熟悉传输层协议的读者会注意到这两种可靠性模式几乎与传输控制协议(Transmission Control Protocol，TCP)和用户数据报协议(User Datagram Protocol，UDP)的运行特性如出一辙，Game Kit 底层使用的就是 TCP 和 UDP，通过它们来实现 GKSendDataReliable 与 GKSendDataUnreliable。如下代码展示了这两种可靠性模式的示例：

```
NSError *error;
GKSession *session;
NSMutableData *stateUpdatePacket;
NSMutableData *heartbeatPacket;

// initialization code omitted for brevity

// send this state update packet reliably
if(![session sendDataToAllPeers:stateUpdatePacket
                   withDataMode:GKSendDataReliable
                          error:&error]) {

    NSLog(@"Error sending packet: %@",[error localizedDescription]);
}

// send this heartbeat packet unreliably
if(![session sendDataToAllPeers:heartbeatPacket
                   withDataMode:GKSendDataUnreliable
                          error:&error]) {

    NSLog(@"Error sending packet: %@",[error localizedDescription]);
}
```

可靠性设置不是根据会话设置的，而是根据数据报设置的，这样应用就可以为待发送的数据类型选择最恰当的可靠性设置了。比如，初始化与状态更新数据报对于应用的操作来说是非常重要的，因此应该总是以可靠的形式发送。需要确保它们的顺序，因为如果没有按照顺序接收到消息，那么端点就会处在不同的操作状态下。简单的心跳数据报或频繁的 UI 更新(每个会话可能会发送成百上千次)通常以不可靠的形式发送，因为如果发送者发现数据报在传输过程中丢失了，那么很有可能还会再发一次。这样，使用 GKSendDataReliable 实际上会降低应用的性能，因为会消耗资源来发送过时的更新并排队最近的数据报，这会造成无状态的反馈循环。

通过 Game Kit 会话进行的所有通信都是未加密的。如果数据报的机密性对于应用来说是非常重要的，比如，移动付款设备会传输信用卡信息进行注册，那么开发者就要负责在将数据发送给 GKSession 前对流量进行加密。请参见第 6 章“保护网络传输”来了解关于 iOS 设备加密的更多信息。如果业务场景不需要加密的安全性，就需要检查通过 Game Kit 发送的所有值，从而确保不会被恶意用户篡改。比如，移动付款设备不应该允许负数的价钱或是对销售商数据库中不存在的条目进行交易。

在调试 Game Kit 应用时，在可能的情况下最好使用真实的设备。iOS 模拟器无法连接到蓝牙会话，通过 Wi-Fi 进行通信时其行为也与物理设备不同。虽然这要求同时有两台或多台设备，不过 Xcode 支持同时在多台设备上运行相同的应用，并且能够轻松在每台设备上切换日志控制台和调试器。这非常有助于调试，因为可以将状态变化和数据包打印到控制台，然后同时比较每台设备上的信息。

说明：

当两台设备通过 USB 连接时，它们都会出现在 Scheme/Device 下拉列表中，此外列表中还有之前安装好的模拟器。要想在这两台设备上调试应用，只需选择一台设备，然后运行应用；接下来，选择另一台设备并再次运行应用。要想在每台设备的控制台与调试器之间切换，请在调试区顶部工具栏的下拉列表中进行选择。

12.2　点对点网络

在点对点 Game Kit 连接下，网络中任何端点的行为同时既是服务器又是客户端。在很多情况下，同一台设备会在业务流程的处理过程中，在客户端与服务器角色间切换，不过可以使用单个 P2P 连接，而不必重新连接设备来转换为新的角色。P2P 还非常适合于在流程开始前并不知晓端点角色的流程。

既是客户端又是服务器这种能力非常强大，不过也有自己的缺陷。在网络中与其他端点交互时，如果要追踪状态，那么 P2P 会导致代码变得非常复杂。此外，虽然 Apple 并没有显式设置 Game Kit 所能支持的 P2P 连接的最大数量，不过随着每个端点不断加入到会话中，网络的可靠性会逐步下滑，当第 5 个端点加入时，可靠性会急剧下滑。可靠性下滑的症状表现为端点毫无征兆地从会话中消失，或是数据报没有成功到达期望的目的地。建议在一个会话中只连接 2 到 4 个端点。Game Kit 会在暗中指引你这么做，因为 GKPeerPickerController 只支持将一个端点连接到另一个，不支持连接到第 3 个或更多的端点。在探索 P2P 的功能时要谨记这些危险信号。

为了阐述 P2P 网络，本节将会介绍一个名为 Game Kit Auctioneer 的示例应用。它可以让端点为条目创建拍卖，接受来自其他端点的多个竞价，并且可以在任意时刻结束竞价。想要购买的端点只需等待拍卖出现、加入拍卖，然后觉得价格合适的话就竞价。在 eBay 时代，拍卖并不太适合用 Game Kit 实现。然而，考虑这样的情况：农村市场地处偏远，堆满了厚壁钢管的仓库。在这些地方可能根本就没有 Internet 连接，或是接待处实在太差。对于更加传统的基于 Web 服务的拍卖应用来说，如果没有 Internet，就完全没有任何用处。Game Kit 可以创建蓝牙 PAN 的能力意味着既可以用在偏远地区，也可以用在有 Wi-Fi 的地方。

12.2.1　连接到会话

要想创建或连接到已有的 Game Kit 会话，每个端点都需要创建一个 GKSession 对象，

它会代表端点与网络进行交互。它接收 3 个参数，分别是会话 ID、端点的显示名以及端点所需的网络模式。对于这个 P2P 应用来说，网络模式总是使用 GKSessionModePeer。代码清单 12-1 展示了完整的会话创建代码。

代码清单 12-1　创建 GKSession(GANetworkingManager.m)

```
#define kGameKitSessionID @"auctioneer1.0"

// create a new GameKit session
_session = [[GKSessionalloc] initWithSessionID:kGameKitSessionID
                                    displayName:nil
                                    sessionMode:GKSessionModePeer];
_session.delegate = self;
```

会话 ID 是第一个参数，在应用的所有实例中它是一个常量，用于确保只能连接到能够理解数据消息的端点。只有具有相同会话 ID 的端点才能被其他端点看到。值得注意的是，会话 ID 要包含应用当前的版本号，这样新版本才能修改或添加新的消息类型。要确保不匹配的版本之间不会进行通信；考虑一下，如果 Auctioneer 应用的下一版本改变了标识获胜的竞拍顺序，这会出现什么情况。版本 2.0 的用户会看到获胜者叫做$12.00 而不是 Johnny Appleseed。这很可能是因为数据格式不兼容，两个客户端会忽略掉彼此不合规的消息，甚至还有可能出现应用崩溃的情况。

第二个参数是显示名，用于标识端点，由 displayNameForPeer:返回。如果在创建会话时返回的显示名是 nil，那就会使用 Settings 应用中显示的设备名字。每个端点还有一个机器可读的 peerId，用于唯一标识会话中的每个端点。显示名并不确保唯一性，它只用于在 UI 中显示。

既然应用已经有了可用的会话，用户现在就应该看到 GALobbyViewController 了，它会在一个 UITableView 中显示出所有可用的端点，如图 12-2 所示。如果有用户想要组织竞拍，那么只需轻拍其他端点，等待他们的响应即可。要想加入其他人的拍卖，只需等待邀请即可，如图 12-3 所示。

当轻拍 UITableView 视图中的一行后，Auctioneer 会调用 GKSession 的 connectToPeer:withTimeout:，进而提示端点的 GKSessionDelegate 接收 session:didReceiveConnectionRequestFromPeer:，后者则会显示出邀请的 UIAlertView。如果远程端点接受，就会调用 GKSession 的 acceptConnectionFromPeer:error:，本地端点则会接收到拥有新状态 GKPeerStateConnected 的 session:peer:didChangeState:。表 12-1 展示了其他的端点状态。接下来，远程端点可以进入拍卖 UI，等待拍卖开始。如果远程端点拒绝邀请，那么会调用 denyConnectionFromPeer:，本地端点则会接收到 session:connectionWithPeerFailed:withError:。拒绝邀请的端点只是在拍卖大厅中等待而已。图 12-4 展示了在连接过程中，在本地端点与远程端点上调用的方法。

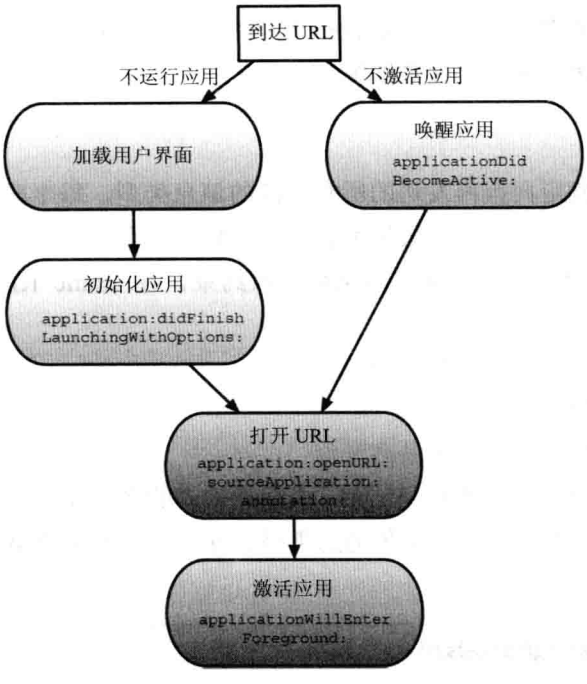


图 12-2

图 12-3

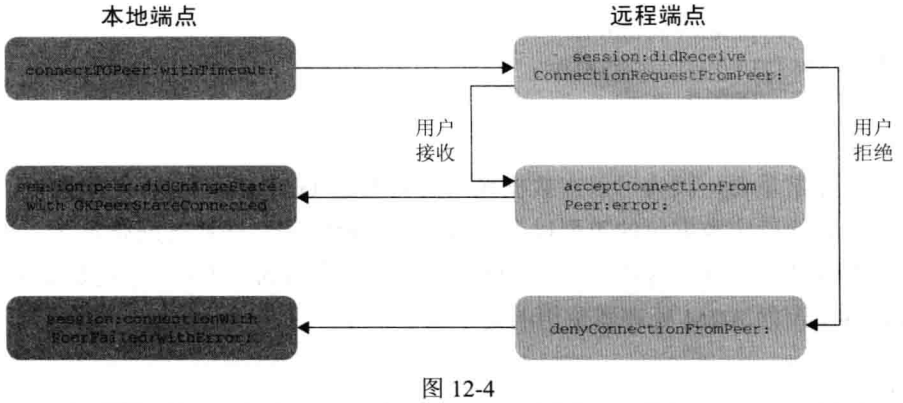


图 12-4

表 12-1　端点连接状态

状　态	说　明
GKPeerStateAvailable	端点可以连接，不过现在还没有连接
GKPeerStateUnavailable	端点无法连接
GKPeerStateConnected	端点已经连接到会话
GKPeerStateConnecting	端点开启了连接过程，不过尚未进入 GKPeerStateConnected 状态
GKPeerStateDisconnected	端点没有连接到会话

拍卖者会重复邀请过程，直到邀请到 6 个人为止，然后轻拍“开始”按钮开始拍卖。拍卖开始的消息是通过 GKSession 发送的第一个自定义数据报。

12.2.2　向端点发送数据

在与其他端点交换数据前，首先需要确定向组内发送的所有可能的消息类型。每个应用都定义了自己的数据报类型，对于复杂的应用来说，数量可能从一个到几十个不等。记住，最大可能的消息大小是 87KB，因此请在设计消息时确保符合该约束。在 Game Kit Auctioneer 中，一共有 4 种可能的数据报：

- GAPacketTypeAuctionStart：通知所有端点拍卖已经开始
- GAPacketTypeBid：向 auctioneer 提交竞价
- GAPacketTypeAuctionStatus：通知所有端点当前最高竞价
- GAPacketTypeAuctionEnd：通知所有端点拍卖已经结束、成功竞拍者的名字及竞价

代码清单 12-2 以枚举 GAPacketType 的形式定义了 4 种消息类型，并将它们映射为整型值。

代码清单 12-2　Packet Type 枚举(GANetworkingPackets.h)

```
typedefenum {
    GAPacketTypeAuctionStart = 0,
    GAPacketTypeAuctionEnd,
    GAPacketTypeAuctionStatus,
    GAPacketTypeBid
} GAPacketType;
```

整型映射非常重要，因为可以在数据包的前面附加 GAPacketType 的整数值，这样每个端点就可以轻松确定数据包的类型以及如何对其进行解码。在通过网络发送二进制数据时，数据要以网络字节或大端顺序发送。这个步骤总是会做的；然而，它也是非常重要的，因为 iOS 设备所用的 ARM 处理器使用的是小端字节序。

字节序

大端与小端字节序是表示二进制数据集的两种方式。大端字节序从最重要到最不重要的顺序来存储字节，而小端字节序则正好相反。大端字节序的一个例子就是电话号码，其中数字的分组是从最重要(国家代码)到最不重要(用户号码)排列的。字节序起源于在内存中存储值的硬件实现，标准化网络字节序的目的是防止小端机器在不知道需要转换的情况下不小心解释以大端字节序存储的二进制数据。大端与小端这两个名字起源于 Jonathon Swift 的小说《格利佛游记》，其中描述了打鸡蛋的两种不同方式。

当拍卖者准备好开始拍卖时，就会发送开始数据包，其中包含主机的 peer ID、待拍卖的商品以及其他端点可以竞拍的价格。代码清单 12-3 展示了如何装配数据包的数据字典、序列化以及如何将其发送给 GANetworkingManager 以进行传输。NSMutableDictionary 的键

必须匹配发送者的消息创建代码以及远程端点的接收代码。

代码清单 12-3　装配并序列化数据包(GALobbyViewController.m)

```
/*
 * tell all participants this auction is starting
 */
NSMutableDictionary *dataDict = [NSMutableDictionary dictionary];
GAPeer *devicePeer = [[GANetworkingManager sharedManager] devicePeer];

// auction owner
[dataDict setObject:devicePeer.peerID forKey:@"ownerPeerID"];

// item name
[dataDict setObject:itemName forKey:@"itemName"];

// number of participants
[dataDict setObject:[NSNumber numberWithInt:[confirmedPeers count]]
            forKey:@"numberOfParticipants"];

// participants
if([confirmedPeers count] > 0) {
    GAPeer *peer = [confirmedPeers objectAtIndex:0];
    [dataDict setObject:peer.peerID forKey:@"participant1PeerID"];
}

// other 5 participants removed for brevity

NSMutableData *data = [[NSMutableData alloc] init];
NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc]
                             initForWritingWithMutableData:data];
[archiver encodeObject:dataDict forKey:@"AuctionStarted"];
[archiver finishEncoding];

// send the message
[[GANetworkingManager sharedManager] sendPacket:data
                                         ofType:GAPacketTypeAuctionStart];
```

装配好数据包后，需要以广播的方式将其发送给所有感兴趣的端点。如下代码片段(GANetworkingManager.m)展示了如何向连接到 Game Kit 会话的所有端点发送数据包类型及其数据。CFSwapInt32HostToBig()用于将 32 位的整数 GAPacketType 转换为大端字节序。注意到数据包的数据并没有被转换为大端字节序，这是因为它是由 NSKeyedArchiver 编码的，后者会创建出独立于字节序的 NSData 对象。要想了解关于如何转换为其他数据类型的更多信息，请参考 https://developer.apple.com/library/mac/#documentation/CoreFoundation/Reference/CFByteOrderUtils/Reference/reference.html。

```
- (void)sendPacket:(NSData*)data ofType:(GAPacketType)type {
    NSMutableData *newPacket = [NSMutableData dataWithCapacity:(
```

```
                                    [data length]+sizeof(uint32_t))];

    // data is prefixed with GAPacketType so the peer knows how to handle it
    uint32_tswappedType = CFSwapInt32HostToBig((uint32_t)type);
    [newPacket appendBytes:&swappedType length:sizeof(uint32_t)];
    [newPacket appendData:data];

    // reliably send the packet
    NSError *error;
    if(![_session sendDataToAllPeers:newPacket
                        withDataMode:GKSendDataReliable
                               error:&error]) {
        NSLog(@"Error sending packet: %@",[error localizedDescription]);
    }
}
```

每个端点都会接收数据包并将其传递给 GANetworkingManagerAuctionDelegate 的 manager:didReceivePacket:ofType: 方法以进行稍后的动作，参考如下代码片段 (GANetworkingManager.m)。数据包类型的解码从数据包的开头进行，其余数据则被恢复到最初发送的 NSData 的副本中。

```
- (void)receiveData:(NSData*)data
           fromPeer:(NSString*)peerID
          inSession:(GKSession*)session
            context:(void*)context {

    GAPacketType header;
    uint32_t swappedHeader;

    if([data length] >= sizeof(uint32_t)) {
        // separate the bytes of the header into swappedHeader
        [data getBytes:&swappedHeader length:sizeof(uint32_t)];

        // convert to the host's endianness
        header = (GAPacketType)CFSwapInt32BigToHost(swappedHeader);

        // separate the remaining bytes of the message into payload
        NSRange payloadRange = {sizeof(uint32_t), [data
                                                   length]-sizeof(uint32_t)};
        NSData* payload = [data subdataWithRange:payloadRange];

        // tell the auction that we received a packet
        [auctionDelegatemanager:self
                didReceivePacket:payload
                          ofType:header];
    }
}
```

在网络管理器处理完数据包后，会将其交给 GAAuctionViewController 进行解释，拍卖

状态也会更新以反映其内容。代码清单 12-4 会解码 GAPacketTypeAuctionStart 数据包，其他 3 种数据包类型使用的也是类似的代码。开始数据包会装配参与者列表，然后更新 biddingHasStarted 标识，这会防止在拍卖开始前就有端点显示出拍卖 UI。图 12-5 与图 12-6 分别从拍卖者与参与者的视角展示出了拍卖视图控制器。

代码清单 12-4　解码数据包(GAAuctionViewController.m)

```
- (void)manager:(GANetworkingManager*)manager
      didReceivePacket:(NSData*)data
      ofType:(GAPacketType)packetType {

   switch(packetType) {
      case GAPacketTypeAuctionStart: {
         NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc]
            initForReadingWithData:data];
         NSDictionary *dataDict = [unarchiver decodeObjectForKey:
            @"AuctionStarted"];
         [unarchiverfinishDecoding];

         // item name
         self.itemName = [dataDict objectForKey:@"itemName"];

         // participants
         self.peerList = [[NSMutableArray alloc] init];

         int numberOfParticipants = [[dataDict
               objectForKey:@"numberOfParticipants"] intValue];

         //NSString *ownerPeerID = [dataDict objectForKey:@"ownerPeerID"];

        NSString *p1PeerID = [dataDict objectForKey:@"participant1PeerID"];
         if(numberOfParticipants> 0) {
            [self.peerListaddObject:[[GAPeer alloc]
                                        initWithPeerID:p1PeerID]];
         }

         // update UI with the info from this packet
         [self.tableViewreloadData];

         // allow participants to make bids
         biddingHasStarted = YES;

         break;
      }
   }
}
```

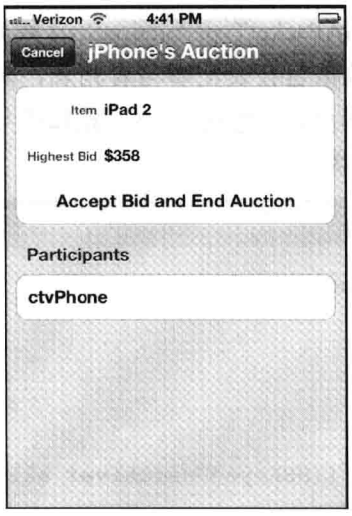


图 12-5

图 12-6

既然 Game Kit Auctioneer 能够发送与接收自定义数据报，因而示例代码的其余部分也只是实现了拍卖的业务规则。在这个特殊应用中，端点之间的连接在拍卖后并不会断开；然而，其他应用可能会要求断开主机或其他端点。比如，如果有规则要求某人一天内只能参加一项拍卖，那么他就应该与其他会话断开。为了做到这一点，端点可以调用 GKSession 的 disconnectFromAllPeers。如果端点决定另一个端点应该强制从会话中断开，那么可以调用 disconnectPeerFromAllPeers:并提供端点的 peer ID。

12.3　客户端-服务器通信

实现在 GKSessionModeServer 与 GKSessionModeClient 模式下带有端点的 GKSession 类似于 P2P 会话，只有一些差别。如果连接到会话的端点是服务器，那么只会将连接的其他端点看作客户端；客户端端点则只会将其他端点看作服务器。如下代码示例展示了如何将服务器端点与客户端端点连接到 Game Kit 会话：

```
#define kGameKitSessionID @"auctioneer1.0"

// create a new GameKit session as a server
_session = [[GKSessionalloc] initWithSessionID:kGameKitSessionID
                                    displayName:nil
                                    sessionMode:GKSessionModeServer];

// create a new GameKit session as a client
_session = [[GKSessionalloc] initWithSessionID:kGameKitSessionID
                                    displayName:nil
                                    sessionMode:GKSessionModeClient];
```

```
_session.delegate = self;
```

与 P2P 网络的第一个差别是客户端-服务器会话有 16 个连接端点的最大限制，这要比 P2P 大一些，因为网络拓扑得到了简化。框架中更为简化的逻辑还使得其代码比 P2P 的代码性能提升了一些。在很多情况下，使用 P2P 网络实现的相同业务规则也可以通过客户端-服务器网络实现，额外的开发时间也很少。

P2P 与客户端-服务器之间的另一差别在于对于后者来说，端点可视性的变化基于其会话模式。服务器端点只能看到客户端端点，客户端端点只能看到服务器端点。这种行为可以确保每个会话只有一个服务器端点以及一个或多个客户端端点。常见的误解是所有的网络流量都要通过服务器路由；然而，就像 P2P 配置一样，所有的数据报对于所有端点都是可见的。为了防止客户端端点解释本不应该由它们解释的数据报，应该在数据报中加入接收方的 peer ID，然后所有不匹配该 peer ID 的端点都会忽略掉该数据报。在 Game Kit Auctioneer 中也进行了类似检查以为成功的竞价选择提示风格。如果设备的 peer ID 是成功的 peer ID，那么提示就会显示“You Won the Auction …”；但如果 peer ID 是远程端点，那么提示就会显示“{peer name} won the auction …”。如下代码片段(GAAuctionViewController.m)展示了该检查，它作为拍卖结束数据报处理的一部分：

```
- (void)manager:(GANetworkingManager*)manager didReceivePacket:(NSData*)data
ofType:(GAPacketType)packetType {

    switch(packetType) {
        case GAPacketTypeAuctionEnd: {
            NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc]
                initForReadingWithData:data];
            NSDictionary *dataDict = [unarchiver
                decodeObjectForKey:@"AuctionFinish"];
            [unarchiver finishDecoding];

            // update data model
            NSInteger winningBid = [[dataDict objectForKey:@"winningBid"]
                 intValue];
            NSString *winnerPeerID = [dataDict objectForKey: @"winnerPeerID"];

            // tell the user who won
            NSString *message;

            if([winnerPeerID isEqualToString:
                [GANetworkingManager sharedManager].devicePeer.peerID]) {

                message = [NSString stringWithFormat:@"You won the auction with
                  a bid of $%i!",winningBid];
            } else {
                message = [NSString stringWithFormat:@"%@ won the auction with a
                    bid of $%i!",
                    [[GANetworkingManager sharedManager] displayNameForPeer:
```

```
                    [[GAPeer alloc] initWithPeerID:winnerPeerID]],
                winningBid];
            }

            UIAlertView *finishedAlert = [[UIAlertView alloc]
                initWithTitle:@"Auction Finished"
                message:message
                delegate:self
                cancelButtonTitle:nil
                otherButtonTitles:@"OK", nil];

            finishedAlert.tag = 700;
            [finishedAlert show];
        }
    }
}
```

12.4　小结

Game Kit 在 iOS 生态系统中占有独特角色，因为它可以集成蓝牙以创建附近设备的网络。其 Wi-Fi 功能是对 Bonjour 服务的简易封装；然而，其真正的优势在于可以使用相同的代码基来支持两种网络技术。其极具创新性的 P2P 模型为应用加入网络提供了空前的灵活性，客户端-服务器模型则为开发者提供了更为熟悉的环境，同时提升了稳定性。

第13章

使用 Bonjour 实现自组织网络

本章内容

- 使用零配置网络
- 解析并连接到 Bonjour 服务
- 实现 Bonjour 以提供优秀的用户体验

本章代码下载

可以在 www.wrox.com 的 Download Code 选项卡中下载本章代码，地址是 www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html。代码位于第 13 章的下载压缩包中，并且包含两个示例应用，这两个应用都包含一个共享的 Bonjour 库：

- Associate Help 应用发布了一个 Bonjour 服务，在与单个客户端通信时充当主机角色。
- Consumer Help 应用浏览可用的 Bonjour 服务，充当客户端角色，并且会请求与主机连接。
- 这个共享库包含 Bonjour(一个 Bonjour 服务)和 BonjourBrowser(一个 Bonjour Browser 类)，可以自定义该共享库并将其添加到项目中以从前端抽象出发布、探测、解析和通信等功能。

使用 iOS 设备的用户有大量的应用可供选择，应用要想在竞争激烈的市场中脱颖而出，就必须提供卓越的用户体验才行。网络设备与 Wi-Fi 网络的不断涌现为众多公司提供了很好的机会来联系用户、吸引用户，并为用户带来超出预期的体验。

Bonjour 就是这样一种技术；设备可以通过它轻松探测并连接到相同网络中的其他设备，整个过程只需要很少的用户参与或是根本就不需要用户参与。该框架提供了众多适合于移动的使用场景，如基于网络的游戏、设备间的文件共享，甚至是家庭自动化。虽然 Bonjour 的使用并不仅限于移动设备，不过实际使用情况还是普遍应用于移动设备，这包

括 Apple 的很多技术，如 Remote 应用、AirPrint、Game Kit；此外，TiVo 也是通过 Bonjour 实现了在一台 TiVo 盒子上录制，在另一台查看的功能。还有很多第三方应用通过 Bonjour 共享文件、联系人信息，以及在 iOS 与非 iOS 应用间共享日历信息等。

本章将会简要介绍 Bonjour 的历史并概览这项技术，接下来通过两个示例应用演示如何将 Bonjour 集成到应用中。这两个示例通过解决零售业的一个常见问题来强化本章介绍的主题：当顾客有问题时能够快速得到帮助，进而为用户提供卓越的购物体验。一共有两个应用，一个是内部的、只有员工才能使用的应用；另一个是顾客可以使用的应用，可以从 App Store 下载。

13.1　zeroconf 概览

Bonjour 技术使得设备可以轻松探测并连接到相同网络中的其他设备。它于 2002 年发布，代号为 Rendezvous，并在 2005 年最终更名为 Bonjour。虽然 Bonjour 能够极大增强用户体验，不过其真正的能力与影响还是取决于应用开发者。Bonjour 赋予开发者极大的灵活性与自由度来执行网络任务，如检测和连接受支持的设备而无须用户输入等。

更具体一些，Bonjour 是 Apple 零配置(zeroconf)网络的实现。zeroconf 是一组技术的集合，旨在简化网络，这是通过消除对动态主机配置协议(Dynamic Host Configuration，DHCP)和域名系统(Domain Name System，DNS)服务器的需要来实现的。zeroconf 的设计旨在能够容纳新的以及不断发展变化的网络产品。zeroconf 目前主要实现 3 个需求：寻址、解析以及探测。

13.1.1　寻址

设备需要在不借助 DHCP 服务器的情况下保护网络地址的能力。zeroconf 使用链路本地地址，这样位于相同链路或网络上的主机就能彼此通信了。iOS 原生支持 IPv4(定义在 RFC 3927 中，http://www.tools.ietf.org/html/rfc3927)与 IPv6(定义在 RFC 2462 中，http://www.tools.ietf.org/html/rfc2462)地址的自动配置。IPv4 链路本地地址的前缀位于 169.254/16 中，IPv6 链接地址的前缀位于 fe80::/64 中。每个前缀都被链路本地地址保留下来了。

13.1.2　解析

zeroconf 的想法之一是让设备可以通过名字而不仅仅是地址就能引用网络上的服务。每台设备都通过独一无二的名字来标识自身，并且这些名字都以.local 结尾。只要网络上的设备没有使用某个名字，客户端就可以使用该名字。主机名类似于上一节介绍的链路本地地址，因为它们只对自己所在的网络有意义。

zeroconf 通过多路域名服务(mDNS)实现了名字的解析，无须配置传统的 DNS 服务器。mDNS 类似于单路 DNS，不过是通过多路协议实现的，网络上的每台设备都会主动监听 DNS 查询。这些查询(以.local 结尾)会被发送到 mDNS 多路地址，具有相应名字的设备会通过其地址做出回应。链路本地的 mDNS 多路地址是 224.0.0.251(IPv4)及 ff02::fb(IPv6)。

有两种类型的 mDNS 查询：一次性的多路广播与持续性的多路广播。前者更加简单，接收它所收到的第 1 个响应，并且不再监听其他的响应。虽然简单，这种方式却能够满足最终用户输入全限定名的本地地址的需求，比如 http://njones.local 或 http://OfficeJet6300.local。

持续查询并不会假定单个响应就是全部。它是异步的，顾名思义，它会持续监听响应并在收到时做出回应。通常情况下，它会显示出所有可用服务的列表，比如网络上所有可用的打印机。如果在最快打印机响应后不再监听结果，那就会不断向用户展现出只有单个条目的列表。这未必是最合适的打印机，会导致非常差劲的用户体验。

要想支持持续查询，执行查询的主机应该保留下已知的回应，这会告诉响应者它们已经回应了特定的查询，这样后面就不必再这么做了。还有很多其他高效的手段已经内置在 mDNS 中，可以在 http://files.multicastdns.org/draft-cheshire-dnsext-multicastdns.txt 找到。

13.1.3　探测

设备必须浏览网络上的可用服务而不必维护服务目录。Apple 的 zeroconf 实现使用基于 DNS 的服务目录(DNS-SD)在网络上发布可用服务。每个可用服务都会发布其实例名、服务类型及域。DNS-SD 与 mDNS 构成了 Bonjour 的基础。DNS-SD 提供了根据类型查询服务的方法，并且通过持久化实例名将网络地址层从用户抽象出来。如果用户今天选择某服务(比如打印机)，那么该服务在明天依然可用，即便该服务的底层网络地址发生变化也如此。

为了提供该功能，Bonjour 使用实例名、服务类型及域的一个元组。该元组唯一标识一个服务，并且是存储指向服务(而不仅仅是地址)的引用的首选方法。服务类型遵循先来先使用的原则。自定义服务类型应该注册到互联网编号分配机构(Internet Assigned Numbers Authority，IANA)，这很简单并且免费，参见 RFC 6335(http://tools.ietf.org/html/rfc6335)。注册 IANA 的一个好处在于它会帮你管理服务类型的冲突。要想了解当前已注册的服务类型的列表，请参见 http://www.dns-sd.org/ServiceTypes.html。

服务由格式<实例名>.<服务类型>.<域>来表示。比如，一台在本地网络上广播 iTunes 家庭共享服务的笔记本电脑可能是 njones-mbp._home-sharing._tcp.local。下面详细介绍此格式的每一部分：

- 实例名：展现给用户的可配置、用户友好的服务名。根据 zeroconf 文档描述，实现不应该要求指定该值。
- 服务类型：一对 DNS 标签。第 1 个标签是个下划线，后跟服务类型或应用协议名，比如_home-sharing。从技术上来说，这个名字可以是任何值，只要你想与之通信的服务也知道它即可；不过，最好将其注册到之前介绍的 IANA。第 2 个标签要么是_tcp(通过 TCP 运行的服务)，要么是_udp(其他服务)。请访问 http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt 以了解更多信息。

- 域：表示服务名在其中注册的 DNS 子域。该值可以是 local。子域是受支持的，它提供了一种简单的方式来组织服务。比如，可以在 printers.apple.com 上广播打印机，从而让 apple.com 变得更加整洁。

13.2　Bonjour 概览

考虑到 Bonjour 是个 zeroconf 实现，因此自然而然地我们会认为部署并连接到 Bonjour 服务也应该遵循着与 zeroconf 相似的一系列步骤。然而，iOS 抽象出了大多数底层网络，留给开发人员一套 API 和一个简单的含有 4 个步骤的流程。该流程包括发布、浏览、解析以及最后连接到服务。当在网络上广播了服务后，浏览相同类型服务的设备就能探测到它。当设备找到所需的服务后，它就会尝试解析服务的地址。确定服务的地址后，主机与客户端就会建立起一条双向的通信通道，并在其上共享数据。该通信通道独立于 Bonjour，由每个应用直接管理。图 13-1 概览了 Bonjour 探测过程，后续章节将会详细介绍其中的每个阶段。

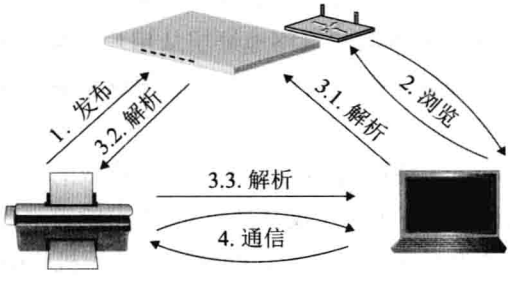


图 13-1

13.2.1　发布服务

NSNetService 类表示 Bonjour 服务。如果想要注册 Bonjour 服务，那么需要创建一个可以发布的 NSNetService 对象。用于创建可发布的 Bonjour 服务的方法是下面介绍的 initWithDomain:type:name:port:，它假定存在一个名为 service 的 NSNetService 实例变量。

```
service = [[NSNetService alloc] initWithDomain:@""
                                           type:@"_serviceType._tcp."
                                           name:name
                                           port:port];
service.delegate = self;
```

前 3 个参数(domain、type 与 name)看起来似曾相识。在 13.1.3 节“探测”中曾介绍过，这 3 个参数与服务名的 3 个组成部分一一对应。从技术上来说，要想发布服务，只需要 type 与 port 参数即可。通常情况下，domain 会被置空，这样服务就会在网络上所有可能的域中可用。name 参数是可选的，如果没有指定值，那么默认情况下就会使用设备名。

还有一个类似的方法 initWithDomain:type:name:，它用在 domain、type 与 name 均已知，而应用想要封装浏览过程的情况下。它通常用在客户端之前已经连接过某服务的情况下。13.2.3 节“解析服务”将会对其进行介绍。

type 参数是必填的，它标识服务如何实现以及服务是什么，还包含了服务类型(或应用协议名)以及传输协议。传输协议的两个可能值分别是._tcp(用于运行在 TCP 之上的服务)与._udp(用于其他服务)。大多数 iOS 应用都会使用._tcp。在之前的 13.1.3 节“探测”中曾介绍过，应用协议名可以是你指定的任何值，不过建议将该名字注册到 http://www.iana.org。应用协议名有 15 个字符的最大限制。Apple 为域的命名约定提供了一些指导性的信息，可以在 https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/NetServices/Articles/domainnames.html 找到。

最后一个参数是 port，它指定应用的端口，这样在与连接的端点通信时数据就会被正确路由。在创建服务时指定的端口是连接过程中最终解析的地址的一部分。一个端口在某一时刻只能由一个应用使用，因此需要选择尚未使用的端口。最佳实践是允许内核从当前可用的端口中进行分配，方式是首次调用 CFSocketSetAddress()函数时传递 0 值。

虽然对于 Bonjour 实现服务探测并非必需，但打算接受连接请求的每个应用都必须配置监听 Socket 来处理这些请求。但遗憾的是，在 iOS 中，这要求开发者从 Cocoa 层转向下面的 CFNetwork，这已经在第 8 章“底层网络”中详细介绍过了。代码清单 13-1 展示了如何让系统为服务分配可用端口，并注册 Socket 回调来监听连接请求。

代码清单 13-1　配置 Socket 并获得端口(/Apps/Associate/associate-help/associate-help/Bonjour.m)

```
CFSocketContext socketCtxt = {0, (__bridge void*)self, NULL, NULL, NULL};
ipv4socket = CFSocketCreate(kCFAllocatorDefault,
                            PF_INET,
                            SOCK_STREAM,
                            IPPROTO_TCP,
                            kCFSocketAcceptCallBack,
                            (CFSocketCallBack)&BonjourServerAcceptCallBack,
                            &socketCtxt);

ipv6socket = CFSocketCreate(kCFAllocatorDefault,
                            PF_INET6,
                            SOCK_STREAM,
                            IPPROTO_TCP,
                            kCFSocketAcceptCallBack,
                            (CFSocketCallBack)&BonjourServerAcceptCallBack,
                            &socketCtxt);

if(ipv4socket == NULL || ipv6socket == NULL) {
    if(ipv4socket) CFRelease(ipv4socket);
    if(ipv6socket) CFRelease(ipv6socket);
```

```
    ipv4socket = NULL;
    ipv6socket = NULL;
    return;
}

int yes = 1;
setsockopt(CFSocketGetNative(ipv4socket),
           SOL_SOCKET,
           SO_REUSEADDR,
           (void *)&yes,
           sizeof(yes));

setsockopt(CFSocketGetNative(ipv6socket),
           SOL_SOCKET,
           SO_REUSEADDR,
           (void *)&yes,
           sizeof(yes));

// set up the IPv4 endpoint
// if port is 0, causes the kernel to choose a port
struct sockaddr_in addr4;
memset(&addr4, 0, sizeof(addr4));
addr4.sin_len = sizeof(addr4);
addr4.sin_family = AF_INET;
addr4.sin_port = htons(port);
addr4.sin_addr.s_addr = htonl(INADDR_ANY);
NSData *address4 = [NSData dataWithBytes:&addr4 length:sizeof(addr4)];

if(kCFSocketSuccess != CFSocketSetAddress(ipv4socket,
                                          (__bridge CFDataRef)address4)) {
    NSLog(@"Error setting ipv4 socket address");
    if(ipv4socket) CFRelease(ipv4socket);
    if(ipv6socket) CFRelease(ipv6socket);
    ipv4socket = NULL;
    ipv6socket = NULL;
    return;
}

if(port == 0) {
    // get the port number, port will be used for IPv6 address and service
    NSData *addr = (__bridge NSData *)CFSocketCopyAddress(ipv4socket);
    memcpy(&addr4, [addr bytes], [addr length]);
    port = ntohs(addr4.sin_port);
}

// set up the IPv6 address
struct sockaddr_in6 addr6;
memset(&addr6, 0, sizeof(addr6));
addr6.sin6_len = sizeof(addr6);
addr6.sin6_family = AF_INET6;
```

```
addr6.sin6_port = htons(port);
memcpy(&(addr6.sin6_addr), &in6addr_any, sizeof(addr6.sin6_addr));
NSData *address6 = [NSData dataWithBytes:&addr6 length:sizeof(addr6)];

if(kCFSocketSuccess != CFSocketSetAddress(ipv6socket,
                                          (__bridge CFDataRef)address6)) {

    NSLog(@"Error setting ipv6 socket address");
    if(ipv4socket) CFRelease(ipv4socket);
    if(ipv6socket) CFRelease(ipv6socket);
    ipv4socket = NULL;
    ipv6socket = NULL;
    return;
}

// set up sources and add sockets to run loop
CFRunLoopRef cfrl = CFRunLoopGetCurrent();
CFRunLoopSourceRef src4 = CFSocketCreateRunLoopSource(kCFAllocatorDefault,
                                                      ipv4socket,
                                                      0);

CFRunLoopAddSource(cfrl, src4, kCFRunLoopCommonModes);
CFRelease(src4);

CFRunLoopSourceRef src6 = CFSocketCreateRunLoopSource(kCFAllocatorDefault,
                                                      ipv6socket,
                                                      0);

CFRunLoopAddSource(cfrl, src6, kCFRunLoopCommonModes);
CFRelease(src6);
```

在上述代码清单中，应用首先创建了一个 Socket 上下文和两个 Socket，PF_INET 用于 IPv4，PF_INET6 用于 IPv6。kCFSocketAcceptCallback 告诉 Socket 接收连接请求并调用回调函数，(CFCallBack)&BonjourServerAcceptCallback，这是以指向第 6 个参数的指针形式传递的，代码清单 13-6 将会对其进行说明。最后一个参数是 Socket 上下文，传递的是指向自身的引用，最终会被传递给回调函数。之所以这么做，是因为 C 函数无法访问 Objective-C 结构。在使用了自动引用计数(Automatic Reference Counting，ARC)的应用中，指针转换要通过__bridge 桥接才行。创建好 Socket 后，应用告诉内核重用 TIME_WAIT 状态的端口，这是通过调用 setsockopt 函数并将 SO_REUSEADDR 作为第 3 个参数传递进来而实现的。

接下来，应用创建 IPv4地址结构 sockaddr_in，并向 sin_port 传递实例变量 port，它已经被初始化为0。传递值为0的端口告诉内核选择一个可用的端口。然后应用通过 CFSocketSetAddress()函数将地址赋给 IPv4 Socket。在应用设置好 IPv4 Socket 地址后，它会通过 CFSocketCopyAddress()获取内核分配的实际端口，并使用该端口创建 IPv6地址结构 sockaddr_in6，然后设置 IPv6 Socket 地址，这类似于设置 IPv4 Socket 地址一样。在指定这

两个 Socket 地址后，将每个 Socket 注册到运行循环中。

端口分配

如果手工分配端口号，需要清楚哪些端口是可用的。通常情况下，你不应该使用介于 0~1023 之间的端口。这些端口会被分配给诸如 HTTP 之类的协议。介于 1024~49151 之间的端口是可以使用的，不过需要注册到 IANA。如果想要使用这个范围内的端口，端口应该注册到 IANA。介于 49152~65535 之间的端口无须注册即可使用。由内核分配的端口就在该范围内。

NSNetService 已经集成到了应用的运行循环中，并通过 NSNetServiceDelegate 协议的一系列委托调用与各种状态变化通信。每个应用都有主运行循环，对象可以在其中进行注册。在创建时，NSNetService 对象可以在当前的线程运行循环中进行调度。

注册运行循环可以让对象执行任务并在每次通过运行循环时发起必要的委托调用。iOS 中大多数的高层网络 API 都已经被注册到了运行循环中，这样它们就可以监听网络活动了。NSNetService 对象可以通过 scheduleInRunLoop:forMode:方法在运行循环中进行调度。然而，除非服务在次级线程中运作或是需要指定不同的模式，否则这么做是没有必要的。如果应用不需要在不同的运行循环中调度服务，那么应用就应该通过方法 removeFromRunLoop:forMode:将服务从当前的运行循环中移除。在将服务从当前运行循环中移除时，可以再次安全地对其进行调度。

当应用准备好向网络发布服务时，它会调用服务的 publish 方法。在默认情况下，如果为新服务指定的名字已经在网络中存在，那么在调用 publish 时会对其重命名。如果应用需要对发布过程进行额外控制，那么它应该使用 publishWithOptions:方法。目前只有一个选项，即 NSNetServiceNoAutoRename，它会阻止重命名。

发布过程的成败可分别通过 netServiceDidPublish:与 netService:didNotPublish:委托方法获悉。netService:didNotPublish:的第 2 个参数是一个 NSDictionary 对象，它包含两个键值对。一个是错误域，可以通过键 NSNetServicesErrorDomain 得到；另一个是错误代码，可以通过键 NSNetServicesErrorCode 得到。错误代码是通过 NSNetServicesError 枚举表示的。如果出现服务名冲突，那么 netService:didNotPublish:委托方法就会被调用，并传递 NSNetServicesCollisionError 错误代码。错误还有可能出现在服务成功发布之后。如果由于错误的原因导致服务没有发布，那么 netServiceWillPublish:委托方法会在服务发布但调用前得到调用。

Bonjour 服务还包含称为 TXT 记录的内容。TXT 记录是一种存储关于服务的额外自定义、键值对信息的机制。其意图在于 TXT 记录可以在与服务建立连接前传输一些不重要的少量信息。比如，预期的 100 字节及以下的信息量使得它非常适合于向客户端传输服务版本号等信息。TXT 记录可以由发布者设置并由客户端读取，但却无法修改。本章后面的 13.2.3 节“解析服务”将会介绍如何读取 TXT 记录。NSNetService 提供了类方法 dataFromTXTRecordDictionary:，可以通过 NSDictionary 创建恰当的记录数据。字典的键必

须为 NSString 对象，值必须为 NSData 对象。如果在生成恰当的格式时出现错误，那么该方法将返回 nil。

```
- (void)netServiceDidPublish:(NSNetService *)sender {
    ...
    // Advertise the service version
    NSData *versionData = [@"1.0" dataUsingEncoding:NSUTF8StringEncoding];
    NSDictionary *txtRecord = [NSDictionary
                               dictionaryWithObject:versionData
                               forKey:@"version"];

    NSData *txtRecordData = [NSNetService
                             dataFromTXTRecordDictionary:txtRecord];
    sender.TXTRecordData = txtRecordData;
}
```

当应用准备停止网络上的服务时，它会调用 stop 方法。stop 方法告诉服务停止广播，这会在下一次运行循环中执行，应用会通过 netServiceDidStop:委托方法得到通知。当服务停止后，它就无法再次被探测到了，也不会再接受新的连接。然而，服务依旧存在，因此应用可以通过再次调用 publish 方法来继续广播。如下代码展示了如何停止网络上的服务广播，这里假定 service 是 NSNetService 的一个实例：

```
[service stop];
```

多任务环境下的 Bonjour 服务

Apple 在 iOS 4 中引入了多任务，虽然这个特性颇受欢迎，不过却极大增加了网络处理的复杂度，换言之，当应用进入到后台，然后又回到前台时该如何处理网络。发送到后台的应用会准备挂起。Bonjour 通过 Socket 进行通信，如果应用挂起，那么它将无法对其进行处理。不过操作系统依旧认为 Socket 是活动的，还可以接受连接，但挂起的应用是无法通过其进行通信的。

当应用准备进入后台时，它应该停止监听，并不再广播任何 Boujour 服务。当回到前台时，应用可以再次打开这些连接并重新发布服务。要想了解这方面的更多信息，请参见 Technical Note TN2277，地址是 http://developer.apple.com/library/ios/#technotes/tn2277/_index.html。

13.2.2　浏览服务

要想让用户能够选择他们想要连接的已发布的服务，应用需要提供一种机制来浏览网络上可用的服务。在 iOS 中，浏览网络上的服务可以通过 NSNetServiceBrowser 类轻松实现。实现 NSNetServiceBrowser 类似于实现上一节介绍的 NSNetService。应用创建一个 NSNetServiceBrowser 实例，然后在运行循环中对其进行调度，执行搜索(而非发布)，然后等待委托方法的调用。与 NSNetService 一样，NSNetServiceBrowser 对象也在创建时被放在当前线程的运行循环中进行调度，因此通常情况下应用无须手工对其进行调度。如果应

用不需要在不同的运行循环中调度浏览器，那么首先应该将其从当前的运行循环中移除，然后再次调度。如下示例演示了如何创建服务浏览器并执行搜索，不要忘记设置委托，这样当出现变化时应用就会收到通知：

```
if(browser == nil) {
    browser = [[NSNetServiceBrowser alloc] init];
}
browser.delegate = self;
[browser searchForServicesOfType:@"_serviceType._tcp."
                        inDomain:@""];
```

可以看到，NSNetServiceBrowser 的 searchForServicesOfType:inDomain:方法执行网络上的搜索。指定的搜索类型应该与主机执行的服务发布过程中指定的类型相匹配。当配置好服务浏览器并准备开始执行搜索时，netServiceBrowserWillSearch:方法会通知委托。无须实现该方法，不过可以在这里更新用户界面，表示搜索正在执行当中。

当探测到服务时，netServiceBrowser:didFindService:moreComing:方法会通知委托。该方法会针对探测到的每个服务调用一次，这样如果想向用户展示服务，应用就必须维护所有服务的集合。moreComing 参数表示该方法是否会针对额外的服务再次被调用。要想提供最佳的用户体验，你应该在 moreComing 参数为 NO 时再次更新用户界面。如果知道有很多服务，并且想要提供渐进的反馈，应用可以对结果进行“分组”，并在每检测到 n 个服务时更新用户界面。获取到 NO 值并不意味着未来不会再探测到更多服务，比如未来会有新的服务发布。下面展示了 netServiceBrowser:didFindService:moreComing:方法的一种可能的实现，它将每个 NSNetService 对象添加到 services 集合 NSMutableArray 中，添加时可以触发用户界面的更新：

```
- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser
           didFindService:(NSNetService *)aNetService
               moreComing:(BOOL)moreComing {

    if(![services containsObject:aNetService]) {
        [services addObject:aNetService];
    }

    if(moreComing == NO) {
        // Update UI
    }

}
```

类似地，当之前探测到的服务不再可用时，netServiceBrowser:didRemoveService:moreComing:方法会通知委托。每当移除服务时，该方法都会被调用一次，它也包含了 moreComing 参数，其行为与 netServiceBrowser:didFindService:moreComing:方法一致。应用可以使用该委托方法更新可用服务集合与用户界面，如下所示：

```
- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser
```

```
        didRemoveService:(NSNetService *)aNetService
           moreComing:(BOOL)moreComing {

    [services removeObject:aNetService];
    if(moreComing == NO) {
        // Update UI
    }
}
```

如果出于某些原因导致搜索失败，netServiceBrowser:didNotSearch:方法就会通知委托。类似于在 13.2.1 节“发布服务”中介绍的 netService:didNotPublish:委托方法，第 2 个参数是一个 NSDictionary 对象，包含一段错误代码和一个域。错误代码与域可以分别通过 NSNetServicesErrorCode 与 NSNetServicesErrorDomain 键得到。错误代码的值是个 NSNetServicesError 枚举。如果搜索失败，应用就应该检查错误并停止搜索，如下所示：

```
- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser
        didNotSearch:(NSDictionary *)errorDict {

    NSString *errorCode = [errorDict
                           objectForKey:NSNetServicesErrorCode];
    NSString *errorDomain = [errorDict
                           objectForKey:NSNetServicesErrorDomain];

    // alert user of the error

    [browser stop];
}
```

当搜索停止后，netServiceBrowserDidStopSearch:委托方法会被调用。这样应用就可以执行任何必要的清理工作，并更新用户界面来表示搜索已经不再执行了。根据应用结构的不同，也可以在这里重置服务浏览器实例与委托，如下所示：

```
- (void)netServiceBrowserDidStopSearch:
      (NSNetServiceBrowser *)aNetServiceBrowser {

    // clears browser and delegate
    // a new browser will be created
    // if search is initiated again
    browser = nil;

    // Update UI
}
```

如果应用需要浏览本地网络之外的服务，那么它应该通过调用 NSNetServiceBrowser 的 searchForBrowsableDomains 方法实现域搜索。当检测到新的服务或是从网络中删除服务时，netServiceBrowser:didFindDomain:moreComing:与 netServiceBrowser:didRemoveDomain:moreComing:方法会被通知。服务检测完成后，应用应该维护域的集合，并展示给用户以

供选择：

```
- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser
           didFindDomain:(NSString *)domainString
              moreComing:(BOOL)moreComing {

    if(![domains containsObject:domainString]) {
        [domains addObject:domainString];
    }
}

- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser
         didRemoveDomain:(NSString *)domainString
              moreComing:(BOOL)moreComing {

    [domains removeObject:domainString];
}
```

13.2.3　解析服务

在与服务通信前，应用首先需要确定服务的网络地址。要想做到这一点，需要调用应用想要连接的服务的 resolveWithTimeout:方法。有两个方法可以获取到 NSNetService 对象。首先，用户可以从之前介绍的 NSNetServiceBrowser 返回的列表中选择一个。其次是直接使用 initWithDomain:type:name:方法创建 NSNetService 实例。后者用在设备之前已经连接到服务并且应用已经保存连接信息的情况下，打印机就是这种方式的一个很好的示例。

```
NSNetService *savedService = [[NSNetService alloc]
                                  initWithDomain:@""
                                            type:@"_serviceType._tcp."
                                            name:@"Kids Shoes Department"];
savedService.delegate = self;
[savedService resolveWithTimeout:5.0];
```

保存服务连接信息

每次使用服务时都会执行解析，这是因为底层服务可能会发生变化。如果想保存服务以供将来使用，就必须保存域、类型及名字的三元组而非服务当前的网络地址。Bonjour 的意图在于抽象出网络地址信息；因此，当网络中的服务来来去去时，它们依旧可以使用。

应用可以使用保存的域、类型与名字来解析服务，这是通过直接使用 initWithDomain:type:name:方法创建 NSNetService 对象并向服务发送 resolveWithTimeout:消息实现的。

在调用 NSNetService 实例的 resolveWithTimeout:方法前，应用需要分配委托。该委托分别通过 netServiceDidResolveAddress:与 netService:didNotResolve:委托方法来通知成功与失败。此外，netServiceWillResolve:方法会在解析服务前得到调用，并提供更新用户界面的好时机。客户端发起服务解析；因此，客户端需要表明除了 NSNetServiceBrowserDelegate

协议外，还遵循 NSNetServiceDelegate 协议。假定用户选择了一个想要连接的服务，代码如下所示：

```
- (void)connectToService:(NSNetService*)service {
    // set the services delegate so the
    // app gets the resolve callbacks
    service.delegate = self;
    [service resolveWithTimeout:5.0];

    // halt browsing since the app
    // is connecting to a service
    [browser stop];
}
```

在解析服务的每个地址后，netServiceDidResolveAddress:委托方法会被调用一次。netServiceDidResolveAddress:方法可能会被调用多次，特别是在同时支持 IPv4 与 IPv6 的设备上。可以部分解析地址信息，因此开发者应该确保在初始化连接前所有必要的地址信息都已经被设置好。地址信息可以通过 NSNetService 对象的 addresses 属性获得，它会被传递给委托。addresses 属性是个由 NSData 对象构成的 NSArray，其中每个对象都包含一个 sockaddr 结构。如果地址信息缺失，那么可以调用 netServiceDidResolveAddress:方法。可以按照下列方式提取出地址信息：

```
- (void)netServiceDidResolveAddress:(NSNetService *)sender {

    for(NSData *addressData in [sender addresses]) {
        struct sockaddr *address;
        address = (struct sockaddr*)[addressData bytes];

        switch(address->sa_family) {
            // IPv6
            case AF_INET6: {
                struct sockaddr_in6 *addr6 =
                    (struct sockaddr_in6*)address;
                //addr6->sin6_port; // Port
                //addr6->sin6_addr; // IP Address
                break;
            }

            // IPv4
            case AF_INET:
            default: {
                struct sockaddr_in *addr4 =
                    (struct sockaddr_in*)address;
                //addr4->sin_port; // Port
                //addr4->sin_addr; // IP Address
                break;
            }
        }
```

```
    }
}
```

类似地，如果在解析地址时出现问题，netService:didNotResolve:委托方法就会被通知。类似于之前介绍的 netService:didNotPublish:与 netServiceBrowser:didNotSearch:委托方法，netService:didNotResolve:的第 2 个参数是个 NSDictionary，包含错误代码与域。错误代码可以通过键 NSNetServicesErrorCode 得到，域可以通过键 NSNetServicesErrorDomain 得到。应用可以从传递给委托方法的服务中读取到任何可用的地址信息，从而确定哪些地址信息是没有解析的。

根据解析服务的不同，连接应用可能需要从主机服务的 TXT 记录数据中获取到额外的自定义信息，比如版本号。TXT 记录存储在 NSNetService 对象的 TXTRecordData 属性中。类似于之前 13.2.1 节“发布服务”中介绍的如何初始化 TXT 记录数据设置，NSNetService 的 dictionaryFromTXTRecordData:类方法会将 TXT 记录数据转换为 NSDictionary，如下所示：

```
- (void)netServiceDidResolveAddress:(NSNetService *)sender {

    NSDictionary *txtDictionary = [NSNetService
                                   dictionaryFromTXTRecordData:
                                   [sender TXTRecordData]];

    NSData *versionData = [txtDictionary objectForKey:@"version"];
    NSString *version = [[NSString alloc]
                         initWithData:versionData
                         encoding:NSUTF8StringEncoding];
    ...
}
```

这里提醒一下，每个键存储的对象都是 NSData 实例，通常情况下这需要一些额外的语句来将它们转换为有意义的形式。然而，由于对象是 NSData 实例，因此可以很灵活地选择传递什么内容给连接服务，只要应用遵循着期望的用法与大小限制即可。

此外，连接应用可以通过 netService:didUpdateTXTRecordData:委托方法收到 TXT 记录数据的更新信息。应用需要调用 NSNetService 对象(通过该对象收到变化通知)的 startMonitoring 方法。类似地，应用应该在不需要收到变化通知时调用 stopMonitoring 方法。

当客户端应用解析完选择的服务后，就可以连接并开始与主机进行通信了。

13.2.4　与服务进行通信

当 Bonjour 完成地址解析后，其使命就宣告结束了，两台设备之间就可以准备连接并通信了。通信是通过流实现的，这是数据交换的一种直接且独立于设备的方法。流是在连接的两个端点间传输的字节序列。在 iOS 中，具体类 NSInputStream 与 NSOutputStream 表示流。

有两种方法可以连接到主机：使用所选的 NSNetService 对象的 hostName 与 port 属性手工进行连接，或是使用服务提供的一对预先配置好的 NSStream 对象。本章的示例使用

预先配置好的流进行通信。第 8 章则更为深入地介绍了底层网络与 Socket。

可以通过调用 NSNetService 对象的 getInputStream:outputStream:方法获取预先配置好的流，并将 NSInputStream 与 NSOutputStream 指针传递给恰当的参数。如果应用只需要一个流，那么应该传递 NULL 而非 NSStream 对象。假定应用只需要读取数据，如下代码展示了如何获取预先配置好的流：

```
- (void)netServiceDidResolveAddress:(NSNetService *)sender {
    ...
    NSInputStream *is;
    if(![sender getInputStream:&is outputStream:NULL]) {
        NSLog(@"Error getting stream");
    }

    // Input Stream
    if(is != NULL ) {
        inputStream = is;
        inputStream.delegate = self;
        [inputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
                               forMode:NSDefaultRunLoopMode];
        if(inputStream.streamStatus == NSStreamStatusNotOpen) {
            [inputStream open];
        }

    }
    ...
}
```

该例获取输入流，然后进行测试，确保返回的是有效的流。如果返回的是有效的流，那么会将其赋给流委托方法中所用的实例变量来测试处理的流。要想通过 NSStreamDelegate 获取事件通知，需要设置委托。返回的流不会在任何运行循环中进行调度，因此也应该在这个时刻完成设置。这样流就可以实现自己的目的，比如每次经过运行循环时确定主机是否还有可供读取的数据。返回的流对象不应该打开，不过你应该通过防御式编程避免任何可能会出现的问题。streamStatus 属性包含一个状态值，它是个 NSStreamStatus 枚举。

stream:handleEvent:委托方法会收到重要的流事件通知，这些通知定义在 NSStreamEvent 枚举中，如下所示：

- NSStreamEventNone
- NSStreamEventOpenCompleted
- NSStreamEventHasBytesAvailable
- NSStreamEventHasSpaceAvailable
- NSStreamEventErrorOccurred
- NSStreamEventEndEncountered

根据应用读写需求的不同，大多数应用所关心的事件是 NSStreamEventOpenCompleted、

NSStreamEventHasBytesAvailable 与 NSStreamEventHasSpaceAvailable。然而，出于完整性以及提供最佳体验的目的，应用应该实现 NSStreamEventErrorOccurred(这样用户就可以收到相应的问题)及 NSStreamEventEndEncountered(这样应用就可以关闭流并将其从运行循环中移除)。如果遇到问题，应用应该检查流的 streamError 属性(一个 NSError 对象)以寻找更多信息。

继续这个通过输入流接收数据的应用示例，如下是 stream:handleEvent:委托方法的一种可能的实现，假定接收到的数据是一个 NSString 对象。

```
- (void)stream:(NSStream *)aStream handleEvent:(NSStreamEvent)eventCode {
    switch(eventCode) {
        case NSStreamEventHasBytesAvailable:
            if(aStream == inputStream) {
                if(receiveData == nil) {
                    receiveData = [[NSMutableData alloc] init];
                }
                uint8_t buffer[1024];
                unsigned int len = 0;
                len = [(NSInputStream *)aStream read:buffer
                                            maxLength:1024];

                if(len) {
                    [receiveData appendBytes:(const void *)buffer
                                      length:len];

                    bytesRead = [NSNumber
                                 numberWithInt:([bytesRead intValue]+len)];

                    if(![inputStream hasBytesAvailable]) {

                        NSString *result = [[NSString alloc]
                                            initWithData:receiveData
                                            encoding:NSUTF8StringEncoding];
                        NSLog(@"** Result: %@", result);

                        // clean up
                        receiveData = nil;
                        bytesRead = nil;
                    }
                } else {
                    NSLog(@"No data found in buffer.");
                }
            }
            break;
        case NSStreamEventOpenCompleted:
        if(aStream == inputStream) {
            NSLog(@"Input Stream Opened");
        }
        break;
```

```
        case NSStreamEventEndEncountered: {
            [aStream close];
            [aStream removeFromRunLoop:[NSRunLoop currentRunLoop]
                               forMode:NSDefaultRunLoopMode];
            break;
        }

        case NSStreamEventErrorOccurred:
            if(aStream == inputStream) {
                NSLog(@"Input stream error: %@", [aStream streamError]);
            }
            break;

        default:
            break;
    }
}
```

该例看起来有点复杂，但实际上却非常直接。首先，委托方法被分解为各个逻辑部分以响应各种可能的 NSStreamEvent。当接收到 NSStreamEventOpenCompleted 事件时，应用只是将其打印到控制台。然而，如果应用想要在 streamStatus 属性外维持特定的流状态，那就非常适合在这里实现。NSStreamEventEndEncountered 也是非常直接的，因为它关闭了已经结束的流，并将其从运行循环中移除，这样在每次经过运行循环时就不必再监控了。如果接收到 NSStreamEventErrorOccurred 事件，那么应用会在控制台打印出语句，包括额外的 streamError 信息。

相对于本书介绍的其他代码来说，NSStreamEventHasBytesAvailable 事件中的代码看起来有些另类，不过应该类似于第 8 章中的代码。当发送系统接收到 NSStreamEventHasSpaceAvailable 事件并且有传输中的数据时，NSStreamEventHasBytesAvailable 事件就会触发。Apple 建议应用以适当大小来发送和接收流数据，一般为 512 或 1024 字节。根据这个建议，单个读事件可能不会接收到传输的整个数据集。当数据集超出接收者的缓存限制时，委托方法会继续因为 NSStreamEventHasBytesAvailable 事件而被调用，直到整个数据集都被从流中读取完为止。在上述示例中，读取到缓存中的数据被附加到 NSMutableData 实例变量，直到数据流传输完所有数据为止。当接收到所有数据后，代码会将这个简单的数据集转换为 NSString 对象并打印到控制台。下一节的示例将会介绍如何传输更为复杂的数据结构。

NSNetService 对象的 getInputStream:outputStream:方法极大简化了设备间的连接与通信过程。虽然本节介绍的内容不属于 Bonjour 的范围，不过在 Bonjour 的探测过程完成后，理解客户端与服务器(或是端点)之间的通信机制是很重要的。下一节将会通过一个更为深入的示例介绍 Bonjour 以及设备与设备之间的通信。

13.3　实现基于 Bonjour 的应用

在本章一开始时曾提到过，本章包含两个示例应用；一个部署到服装零售商的员工设备上；另一个面向顾客，作为已有应用的一部分进行部署。员工会通过商店的客户 Wi-Fi 网络向客户广播来为其提供帮助。使用零售商应用的客户可以浏览到员工，并在当前特定的位置寻求帮助。

从技术上来说，员工应用的实例总是交互模式的主机或服务端，顾客则是相应的客户端，总是发出连接请求。这两个示例应用都遵循着类似的模式，其中模型抽取出了 Bonjour 活动，并通过 NSNotificationCenter 与前端进行通信。

在应用开始通信前，它们需要说相同的语言。如何传输结构化信息取决于两个应用的需求；然而，本章介绍的应用使用了两个类——HelpRequest 与 HelpResponse，它们由员工与顾客应用共享。两个应用的交互模式是这样的：顾客请求员工的帮助，员工则会进行回复。顾客求助请求由 HelpRequest 类表示，如代码清单 13-2 所示。

代码清单 13-2　HelpRequest 类定义(/Apps/Associate/associate-help/associate-help/HelpRequest.h)

```
@interface HelpRequest : NSObject <NSCoding>

@property(nonatomic,strong) NSString *question;
@property(nonatomic,strong) NSString *location;

@end
```

如代码清单 13-2 所示，HelpRequest 类遵循 NSCoding 协议，这是序列化与反序列化结构化对象的一种简单方式。当与 NSKeyedArchiver 和 NSKeyedUnarchiver 一起使用时，应用就可以将结构化对象序列化为可通过网络传输的字节。由于 HelpRequest 类在两个应用间共享，因此传输的数据可以在接收端轻松转换为结构化对象。代码清单 13-3 展示了如何为 HelpRequest 类实现 NSCoding。

代码清单 13-3　HelpRequest 类实现(/Apps/Associate/associate-help/associate-help/HelpRequest.m)

```
#import "HelpRequest.h"

@implementation HelpRequest
@synthesize question, location;

#pragma mark - NSCoding
- (void)encodeWithCoder:(NSCoder*)aCoder {
    [aCoder encodeObject:self.question forKey:@"question"];
    [aCoder encodeObject:self.location forKey:@"location"];
}

- (id)initWithCoder:(NSCoder*)aDecoder {
```

```
    self = [super init];
    self.question = [aDecoder decodeObjectForKey:@"question"];
    self.location = [aDecoder decodeObjectForKey:@"location"];
    return self;
}

@end
```

帮助请求的响应由 HelpResponse 类表示。HelpResponse 是个简单的类，也遵循 NSCoding 协议。相比 HelpRequest 来说，它还展示了如何编解码不同的数据类型，如代码清单 13-4 所示。

代码清单 13-4　HelpResponse 接口与实现(/Apps/Associate/associate-help/associate-help/HelpResponse.h/m)

```
@interface HelpResponse : NSObject <NSCoding>
@property(nonatomic,assign) BOOL response;
@end
@implementation HelpResponse
@synthesize response;

#pragma mark - NSCoding
- (void)encodeWithCoder:(NSCoder*)aCoder {
    [aCoder encodeBool:self.response forKey:@"response"];
}

- (id)initWithCoder:(NSCoder*)aDecoder {
    self = [super init];
    self.response = [aDecoder decodeBoolForKey:@"response"];
    return self;
}
```

如前所述，这些基础通信类的代码会在两个应用间共享。

图 13-2

13.3.1　员工应用

既然已经实现了通信基础，现在是时候构建员工应用了。运行该应用的员工可以帮助顾客。该应用包含一个视图，可以让员工设置自己的部门并切换自己是否可以提供帮助。启动时，员工应用的界面如图 13-2 所示。

员工应用的核心主要聚焦在将自身发布为服务，并处理与所连接设备的通信。该功能位于 Bonjour 类中，是个单例，用于处理所有与 NSNetService 及 NSStream 相关的细节信息。

代码清单 13-5 展示了 Bonjour 类的定义；注意接口上面声明的通知常量。它们是使用 NSNotificationCenter 广播的通知名。员工应用的所有代码位于从本书站点下载的压缩包中。

代码清单 13-5　Bonjour 接口定义(/Apps/Associate/associate-help/associate-help/Bonjour.h)

```
#import "HelpRequest.h"
#import "HelpResponse.h"

#define kNotificationResultSet              @"NotificationObject"
#define kPublishBonjourStartNotification    @"PublishStartNotification"
#define kPublishBonjourErrorNotification    @"PublishErrorNotification"
#define kPublishBonjourSuccessNotification  @"PublishSuccessNotification"
#define kStopBonjourSuccessNotification     @"StopSuccessNotification"
#define kHelpRequestedNotification          @"HelpRequestedNotification"

@interface Bonjour : NSObject <NSNetServiceDelegate, NSStreamDelegate>

+ (Bonjour*)sharedPublisher;

- (BOOL)publishServiceWithName:(NSString*)name;

- (void)stopService;

- (void)sendHelpResponse:(HelpResponse*)response;

@end
```

虽然代码清单 13-5 中声明的大多数方法类似于核心的 NSNetService 方法，不过封装 NSNetService 方法的做法可以让模型尽可能将发布细节从前端抽取出来。该例中的 Bonjour 类旨在用于单个的服务类型和空的服务域。虽然这并不适用于所有情况，不过这样可以使模型以 publishServiceWithName:方法的形式公开流式的发布过程。需要实现代码清单 13-5 中声明的方法，如代码清单 13-6 所示。

代码清单 13-6　Bonjour 实现(/Apps/Associate/associate-help/associate-help/Bonjour.m)

```
- (BOOL)publishServiceWithName:(NSString*)name {

    // setup the listening socket for connection attempts
    // and determine a port on which to advertise the service
    if(![self setupListeningSocket]) {
        return NO;
    }

    // create the service for publishing
    // this type should be registered - iana.org
    service = [[NSNetService alloc]
                    initWithDomain:@""
```

```
                                type:@"_associateHelp._tcp."
                                name:name
                                port:port];

    if(service == nil) {
        return NO;
    }

    service.delegate = self;

    // Publish service
    [Utils postNotification:kPublishBonjourStartNotification];
    [service publish];

    return YES;
}

- (void)stopService {
    [service stop];
}
```

publishServiceWithName:方法在实例化 NSNetService 对象之前创建了一个监听 Socket，并让内核分配一个端口，如之前的代码清单 13-1 所示。如果服务创建成功，就会设置委托，UI 也会在将服务广播到网络前进行调整。如果服务成功发布，netServiceDidPublish:委托方法就会得到通知，它会以此通知前端，这样就可以相应地更新界面，如代码清单 13-7 所示。如果在发布服务时遇到错误，netService:didNotPublish:委托方法就会得到调用，它还会向前端发出警告，如代码清单 13-7 所示。

代码清单 13-7　NetService 发布委托方法(/Apps/Associate/associate-help/associate-help/Bonjour.m)

```
- (void)netServiceDidPublish:(NSNetService *)sender {
    [Utils postNotification:kPublishBonjourSuccessNotification];
}

- (void)netService:(NSNetService *)sender
    didNotPublish:(NSDictionary *)errorDict {
    // typically you would pass along the errorDict
    // object or some form of error messaging
    [Utils postNotification:kPublishBonjourErrorNotification];
}

- (void)netServiceDidStop:(NSNetService *)sender {
    // reset port so a new one is assigned
    port = 0;
    CFRelease(ipv4socket);
    CFRelease(ipv6socket);

    [Utils postNotification:kStopBonjourSuccessNotification];
}
```

服务成功停止时，netServiceDidStop:方法会被调用，应用应该将端口重置为 0，这样后续的发布请求就会在 Socket 创建时分配新的端口。在发布与其他设备通信的服务的过程中，最为单调的步骤之一就是要配置 Socket 与连接处理器回调。由于所有连接请求都源自该例中的顾客应用，因此只有员工应用需要考虑这一点。

13.2.1 节“发布服务”中的代码清单 13-1 介绍了如何配置 Socket 以及如何让内核分配端口。虽然可以为待使用的服务硬编码端口，不过你可能会遇到冲突，造成应用无法使用。因此，我们建议让系统帮你选择端口。如代码清单 13-1 所示，应用需要实现一个回调函数，当 Socket 接收到连接请求时该回调函数会被调用。代码清单 13-8 展示了如何实现代码清单 13-1 中设置的回调函数。

代码清单 13-8　连接请求回调函数(/Apps/Associate/associate-help/associate-help/Bonjour.m)

```
static void BonjourServerAcceptCallBack(CFSocketRef socket,
                                        CFSocketCallBackType type,
                                        CFDataRef address,
                                        const void *data,
                                        void *info) {

    Bonjour *server = (__bridge Bonjour*)info;
    if(type == kCFSocketAcceptCallBack) {
        // AcceptCallBack: data is pointer to a CFSocketNativeHandle
        CFSocketNativeHandle socketHandle
            = *(CFSocketNativeHandle *)data;

        CFReadStreamRef readStream = NULL;
        CFWriteStreamRef writeStream = NULL;
        CFStreamCreatePairWithSocket(kCFAllocatorDefault,
                                     socketHandle,
                                     &readStream,
                                     &writeStream);
        if(readStream && writeStream) {
            CFReadStreamSetProperty
                (readStream,
                kCFStreamPropertyShouldCloseNativeSocket,
                kCFBooleanTrue);

            CFWriteStreamSetProperty
                 (writeStream,
                 kCFStreamPropertyShouldCloseNativeSocket,
                 kCFBooleanTrue);

            NSInputStream *is = (__bridge NSInputStream*)readStream;
            NSOutputStream *os = (__bridge NSOutputStream*)writeStream;
            [server handleNewConnectionWithInputStream:is
                                          outputStream:os];
        } else {
            // encountered failure
```

```
            // no need for socket anymore
            close(socketHandle);
        }

        // clean up
        if(readStream) {
            CFRelease(readStream);
        }

        if(writeStream) {
            CFRelease(writeStream);
        }
    }
}
```

类似于代码清单 13-1，代码清单 13-8 看起来也有些复杂，不过实际上却非常直接。首先，当 Socket 接收到来自顾客应用的连接请求时，该函数会被调用。可以将其看作 Objective-C 中的委托方法。当该函数被调用时，首先要获取到 self 引用。由于 C 函数无法访问 self，因此这是通过将 info 参数转换为 Bonjour 类实现的。在创建用于配置监听 Socket 的 Socket 上下文时会通过(__bridge void*)self 设置 info 参数，如代码清单 13-1 所示。

如果回调类型是 kCFSocketAcceptCallBack，这就意味着连接请求已被接受，子流将会传递给回调函数，应用则会为 Socket 创建读写流。如果两个流都创建了，那么当流释放时，应用就会告诉每个流关闭并释放底层的本地 Socket。在设置好流属性后，它们会被转换为相应的 Objective-C 类型，接下来 handleNewConnectionWithInputStream:outputStream:会被调用，如代码清单 13-9 中的实现所示。

代码清单 13-9　handleNewConnectionWithInputStream:outputStream:方法(/Apps/Associate/associate-help/associate-help/Bonjour.m)

```
- (void)handleNewConnectionWithInputStream:(NSInputStream*)istr
                                outputStream:(NSOutputStream*)ostr {
    inputStream = istr;
    outputStream = ostr;

    inputStream.delegate = self;
    outputStream.delegate = self;

    [inputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
                           forMode:NSDefaultRunLoopMode];
    // output stream is scheduled in the runloop when it is needed


    if(inputStream.streamStatus == NSStreamStatusNotOpen) {
        [inputStream open];
    }

    if(outputStream.streamStatus == NSStreamStatusNotOpen) {
```

```
        [outputStream open];
    }
}
```

handleNewConnectionWithInputStream:outputStream:方法设置了两个实例变量，用于后续处理中流的比较，然后又设置了委托。流并没有预先在运行循环中进行调度，因此需要对其进行调度，这样它们就可以监控各种 NSStreamEvent 了。接下来，该方法会检查每个流的状态，如有必要就打开流。当流打开后，委托就会收到 NSStreamEventOpenCompleted 事件通知，如代码清单 13-10 所示，此处只是将事件打印到控制台。

代码清单 13-10　stream:handleEvent:打开与完成事件(/Apps/Associate/associate-help/associate-help/Bonjour.m)

```
- (void)stream:(NSStream *)aStream
    handleEvent:(NSStreamEvent)eventCode {

    switch(eventCode) {
        case NSStreamEventHasBytesAvailable:
            if(aStream == inputStream) {
                if(receiveData == nil) {
                    receiveData = [[NSMutableData alloc] init];
                }
                uint8_t buf[1024];
                unsigned int len = 0;
                len = [(NSInputStream *)aStream read:buffer
                                            maxLength:1024];
                if(len) {
                [receiveData appendBytes:(const void *)buffer
                                  length:len];

                bytesRead = [NSNumber
                             numberWithInt:([bytesRead intValue]+len)];

                if(![inputStream hasBytesAvailable]) {

                    // you could optionally keep the 'transaction'
                    // state stored so that you could determine
                    // which object you are expecting.
                    HelpRequest *request;
                    @try {
                        request =
                        [NSKeyedUnarchiver
                        unarchiveObjectWithData:receiveData];
                        NSDictionary *info =
                        [NSDictionary
                         dictionaryWithObject:request
                         forKey:kNotificationResultSet];

                        [[NSNotificationCenter defaultCenter]
```

```
                        postNotificationName:
                        kHelpRequestedNotification
                        object:nil
                        userInfo:info];
                    }
                    @catch(NSException *exception) {
                        NSString *msg =
                        @"Exception while unarchiving request data.";
                        NSLog(@"%@", msg);
                    }
                    @finally {
                        // clean up
                        receiveData = nil;
                        bytesRead = nil;
                    }
                }
            } else {
                NSLog(@"No data found in buffer.");
            }
        }
        break;

    ...

    case NSStreamEventOpenCompleted:
        if(aStream == inputStream) {
            NSLog(@"Input Stream Opened");
        } else {
            NSLog(@"Output Stream Opened");
            }
            break;
        ...
     }
}
```

代码清单 13-10 还包含从输入流中读取数据的逻辑。当连接系统接收到数据时，流委托会收到 NSStreamEventHasBytesAvailable 事件通知。处理该事件的逻辑会持续追加来自输入流中的数据，直到没有可读取的字节时为止。由于对员工应用来说进来的唯一数据就是 HelpRequest，因此当读取完所有数据后，应用会尝试通过 NSKeyedUnarchiver 与 NSCoding 来创建 HelpRequest 对象，如代码清单 13-3 所示。如果对象创建成功，应用就会通知前端，这会向员工展现警告，如图 13-3 所示。

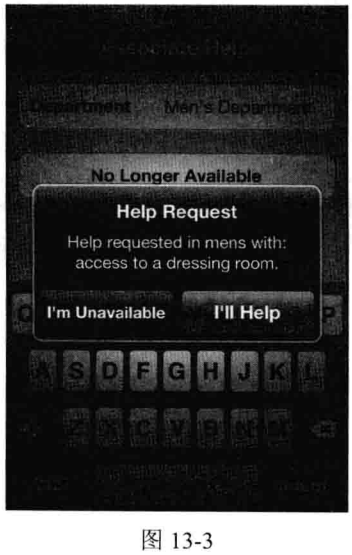


图 13-3

当员工对 HelpRequest 做出回应后(如图 13-3 所示)，Bonjour 的 sendHelpResponse:方法就会得到调用。代码清单 13-11 详细列出了 sendHelpResponse:方法的实现。

代码清单 13-11　sendHelpResponse:实现(/Apps/Associate/associate-help/associate-help/Bonjour.m)

```
- (void)sendHelpResponse:(HelpResponse*)response {
    if(sendData == nil) {
        sendData = [[NSMutableData alloc] init];
    }
    NSData *responseData =
        [NSKeyedArchiver archivedDataWithRootObject:response];

    [sendData appendData:responseData];
    [outputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
                            forMode:NSDefaultRunLoopMode];

    // associate is going to help customer
    // stop the service so they aren't discoverable
    // while with the customer
    if(response.response == YES) {
        [self stopService];
    }
}
```

sendHelpResponse:方法通过 NSKeyedArchiver 创建响应的数据表示，然后将其添加到数据传输的队列中。对于更加复杂的应用来说，诸如 NSData 构成的 NSMutableArray 队列更具保证。在将 HelpResponse 数据添加到队列中后，应用会在当前运行循环中调度 outputStream，这样就可以开始监控 NSStreamEventHasSpaceAvailable 事件了，它会通知 stream:handleEvent:委托方法。当在运行循环中调度完 outputStream 后，服务就停止了，这样在员工繁忙时，其他顾客就不会再尝试获取帮助了。当员工帮助完当前顾客后，他可以重新发布，表明自己可以提供帮助。

代码清单 13-12 表明如果有待发送的数据，委托每次就会向流缓存中写入 1KB 数据，顾客应用的流可以从中读取数据。当所有数据都传输完毕后，应用会清空队列并从运行循环中移除 outputStream。当有新的数据要传输时，它会被重新调度。

代码清单 13-12　NSStreamEventHasSpaceAvailable 逻辑(/Apps/Associate/associate-help/associate-help/Bonjour.m)

```
- (void)stream:(NSStream *)aStream
  handleEvent:(NSStreamEvent)eventCode {

    switch(eventCode) {
        ...
        case NSStreamEventHasSpaceAvailable: {
            if(aStream == outputStream) {
                // send data if there is some pending
```

```
                if([sendData length] > 0) {
                    uint8_t *readBytes =
                        (uint8_t *)[sendData mutableBytes];

                    // keep track of pointer position
                    readBytes += [bytesWritten intValue];
                    int data_len = [sendData length];

                    unsigned int len =
                        ((data_len - [bytesWritten intValue] >= 1024) ?
                        1024 : (data_len-[bytesWritten intValue]));

                    uint8_t buffer [len];
                    (void)memcpy(buffer, readBytes, len);
                    len = [(NSOutputStream*)aStream
                              write:(const uint8_t *)buffer
                          maxLength:len];

                    bytesWritten =
                        [NSNumber
                        numberWithInt:([bytesWritten intValue]+len)];

                    if([sendData length] == [bytesWritten intValue]) {
                        sendData = nil;
                        [outputStream
                            removeFromRunLoop:[NSRunLoop currentRunLoop]
                                      forMode:NSDefaultRunLoopMode];
                    }

                    if([bytesWritten intValue] == -1) {
                        NSLog(@"Error writing data.");
                    }
                }
            }
            break;
        }
        ...
    }
}
```

13.3.2　顾客应用

现在已经完成了主机各个部分的工作，员工也可以发出广播以提供帮助，接下来顾客应用就需要发起请求了。顾客应用包含一个带有两个视图的选项卡控制器。第 1 个视图向顾客提供从其移动设备购买和浏览的功能。对于该例来说，购买与浏览功能并未实现。第 2 个视图向顾客呈现出一个可用的员工列表。该例假定顾客总是与员工连接到相同的网络。在实际情况中，你应该为帮助请求提供一种后备方式，比如简单的形式。如果顾客没有连接到网络，那么禁用所有特性会提供更好的体验。

图 13-4 展示了购物视图，图 13-5 展示了帮助视图(图中显示出一名员工)。正如在员工应用中提到的，所有的表示层代码都位于下载源代码的压缩包中。然而，与员工应用一样，所有的 Bonjour 浏览、连接与通信功能都已经从表示层抽象出来，这使得服务的显示与交互变得非常直接。

图 13-4

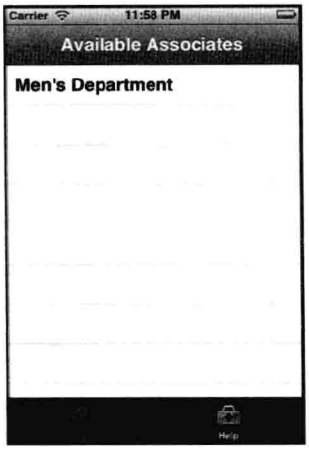


图 13-5

模型中的主类是 BonjourBrowser，定义在代码清单 13-13 中。BonjourBrowser 也是个单例。类似于员工应用，模型与前端之间的交互是通过 NSNotificationCenter 完成的。通知常量则定义在@interface 声明前。

代码清单 13-13　BonjourBrowser 接口定义(/Apps/Consumer/consumer-help/consumer-help/BonjourBrowser.h)

```
#import "HelpRequest.h"
#import "HelpResponse.h"

#define kNotificationResultSet       @"NotificationObject"

#define kBrowseStartNotification     @"BonjourBrowseStartNotification"
#define kBrowseErrorNotification     @"BonjourBrowseErrorNotification"
#define kBrowseSuccessNotification   @"BonjourBrowseSuccessNotification"

#define kConnectStartNotification    @"BonjourConnectStartNotification"
#define kConnectErrorNotification    @"BonjourConnectErrorNotification"
#define kConnectSuccessNotification  @"BonjourConnectSuccessNotification"

#define kServiceRemovedNotification  @"BonjourServiceRemovedNotification"
#define kSearchStoppedNotification   @"BonjourSearchStoppedNotification"

#define kHelpRequestedNotification   @"HelpRequestedNotification"
#define kHelpResponseNotification    @"HelpResponseNotification"

@interface BonjourBrowser : NSObject <NSNetServiceDelegate,
```

```
                                        NSNetServiceBrowserDelegate,
                                        NSStreamDelegate>

+ (BonjourBrowser*)sharedBrowser;

- (void)browseForHelp;

- (NSArray*)availableServices;

- (void)connectToService:(NSNetService*)service;

- (void)sendHelpRequest:(HelpRequest*)request;

@end
```

如果顾客选择Help选项卡，应用就会调用browseForHelp，它会实例化一个NSNetServiceBrowser对象，并开始搜索类型_associateHelp._tcp.，这是在员工应用发布服务时指定的类型。当执行搜索时，方法会通知监听器相应的事件，如代码清单 13-14 所示。

代码清单 13-14　browseForHelp 实现(/Apps/Consumer/consumer-help/consumer-help/BonjourBrowser.m)

```
- (void)browseForHelp {
    if(browser == nil) {
        browser = [[NSNetServiceBrowser alloc] init];
    }

    browser.delegate = self;
    [browser searchForServicesOfType:@"_associateHelp._tcp."
                            inDomain:@""];

    [Utils postNotification:kBrowseStartNotification];
}
```

如果在执行搜索时遇到问题，netServiceBrowser:didNotSearch:委托方法就会收到通知，它会向前端广播一条消息并停止搜索，如代码清单13-15所示。当搜索停止后，netServiceBrowserDidStopSearch:委托方法会被调用。如代码清单13-15所示，应用可以利用这个机会清空 browser 实例变量并通知前端，这样相关的状态指示器就可以得到更新了。

代码清单 13-15　netServiceBrowser:didNotSearch:与 netServiceBrowserDidStopSearch:实现(/Apps/Consumer/consumer-help/consumer-help/BonjourBrowser.m)

```
- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser
             didNotSearch:(NSDictionary *)errorDict {
    // alert the user and stop the browser
    [Utils postNotification:kBrowseErrorNotification];
    [browser stop];
}
```

```
- (void)netServiceBrowserDidStopSearch:
      (NSNetServiceBrowser *)aNetServiceBrowser {

    // clears browser and delegate
    // a new browser will be created
    // if search is initiated again
    browser = nil;
    [Utils postNotification:kSearchStoppedNotification];
}
```

如果探测到服务，netServiceBrowser:didFindService:moreComing:委托方法就会得到调用，它会将服务添加到 NSMutableArray 实例变量中，该实例变量维护着可用的服务，并且会通知前端，如代码清单 13-16 所示。请在通知监听者进行更新前特别注意 moreComing 指示器，因为针对每个被探测到的服务，它都会被调用一次。在该例中，服务的数量不会很多，假设每个部门有两三个服务，不过在大范围的公司网络中搜索打印机也是很耗时间的。虽然该例并不会对服务集合进行排序，不过可以在广播通知前做到这一点。

代码清单 13-16　netServiceBrowser:didFindService:moreComing:实现(/Apps/Consumer/consumer-help/consumer-help/BonjourBrowser.m)

```
- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser
        didFindService:(NSNetService *)aNetService
           moreComing:(BOOL)moreComing {

    if(![services containsObject:aNetService]) {
        [services addObject:aNetService];
    }

    if(moreComing == NO) {

        [Utils postNotification:
            kBrowseSuccessNotification];
    }
}
```

前端可以通过调用 availableServices 从模型获取到当前可用的服务。这样模型就可以添加和移除服务，前端可以根据需要从中获取所需的服务而无须维护自己的副本。availableServices 返回一个包含着 NSNetService 对象的 NSArray，可以显示在 UITableView 中，如图 13-5 所示。当顾客准备求助时，他们可以轻拍恰当的服务。选择某服务会开始解析过程，将所选择的服务传递给 connectToService:，然后向顾客展示出一个提示了更多信息的视图，如图 13-6 所示。

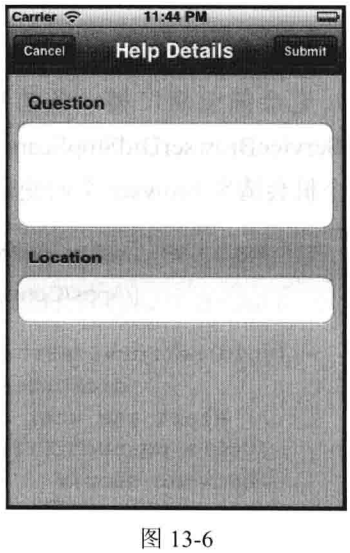


图 13-6

如果在解析服务时遇到问题，netService:didNotResolve:委托方法就会得到调用，它会向前端发送通知，这样顾客就会收到通知。当服务解析完毕后，netServiceDidResolveAddress:委托方法会得到调用。代码清单 13-17 展示了 netService:didNotResolve: 与 netServiceDidResolveAddress:委托方法的实现。正如在13.2.3节“解析服务”中介绍的那样，开发者应该通过防御式编程，确保在获取预定义流或在手工连接前必要的地址信息已经得到解析。由于这些内容之前已经介绍过，因此出于简洁的目的，下面的代码清单13.7将它们省略掉了。

代码清单 13-17　netService:didNotResolve:与 netServiceDidResolveAddress:实现(/Apps/Consumer/consumer-help/consumer-help/BonjourBrowser.m)

```
- (void)netService:(NSNetService *)sender
     didNotResolve:(NSDictionary *)errorDict {
    [Utils postNotification:kConnectErrorNotification];
}

- (void)netServiceDidResolveAddress:(NSNetService *)sender {
    NSInputStream *tmpIS;
    NSOutputStream *tmpOS;
    BOOL error = NO;

    // this application requires both streams
    // if you don't get them both, that poses
    // a problem
    if(![sender getInputStream:&tmpIS outputStream:&tmpOS]) {
        error = YES;
    }

    // Input Stream
    if(tmpIS != NULL ) {
        inputStream = tmpIS;
        inputStream.delegate = self;
        [inputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
                               forMode:NSDefaultRunLoopMode];
        if(inputStream.streamStatus == NSStreamStatusNotOpen) {
            [inputStream open];
        }

    } else {
        error = YES;
    }

    // Output Stream
    if(tmpOS != NULL ) {
        outputStream = tmpOS;
        outputStream.delegate = self;
        //output stream is scheduled in runloop when it is needed
        if(outputStream.streamStatus == NSStreamStatusNotOpen) {
```

```
            [outputStream open];
        }

    } else {
        error = YES;
    }

    if(error == NO) {
        [Utils postNotification:kConnectSuccessNotification];
    } else {
        [Utils postNotification:kConnectErrorNotification];
    }
}
```

netServiceDidResolveAddress:方法通过 NSNetService 的 getInputStream:outputStream:方法获取到预定义流。对于返回的每个流来说，它会将其分配给对应的实例变量 inputStream 或 outputStream；然后设置委托；接下来检查 streamStatus 来确定流是否已经打开。inputStream 会在运行循环中进行调度，但 outputStream 不会。outputStream 会在需要时在运行循环中进行调度，正如在员工应用中所做的那样。

当顾客在请求视图中输入必要的信息并轻拍 Submit 时，应用会调用 sendHelpRequest:，它会通过 NSKeyedArchiver 将请求参数转换为 NSData，并将数据追加到应用发出队列中，然后在当前运行循环中调度 outputStream，如代码清单 13-18 所示。

代码清单 13-18　sendHelpRequest:实现(/Apps/Consumer/consumer-help/consumer-help/BonjourBrowser.m)

```
- (void)sendHelpRequest:(HelpRequest*)request {

    if(sendData == nil) {
        sendData = [[NSMutableData alloc] init];
    }

    // convert the request to NSData (using NSKeyedArchiver/NSCoding)
    NSData *requestData = [NSKeyedArchiver
                           archivedDataWithRootObject:request];

    [sendData appendData:requestData];
    [outputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
                           forMode:NSDefaultRunLoopMode];
}
```

当 outputStream 在运行循环中得到调度后，应用就开始监听来自连接端点的 NSStreamEventHasSpaceAvailable 事件。如果设备接收到事件，它会通知 stream:handleEvent:委托，正如之前在员工应用中介绍的那样。代码清单 13-19 演示了如何向员工应用发送数据。该过程与代码清单 13-12 介绍的一样。

代码清单 13-19　stream:handleEvent: NSStreamEventHasSpaceAvailable 实现(/Apps/Consumer/consumer-help/consumer-help/BonjourBrowser.m)

```
- (void)stream:(NSStream *)aStream
   handleEvent:(NSStreamEvent)eventCode {

    switch(eventCode) {
        case NSStreamEventHasSpaceAvailable: {
            if(aStream == outputStream) {
                if([sendData length] > 0) {
                    uint8_t *readBytes =
                        (uint8_t *)[sendData mutableBytes];

                    // keep track of pointer position
                    readBytes += [bytesWritten intValue];
                    int data_len = [sendData length];

                    unsigned int len =
                        ((data_len - [bytesWritten intValue] >= 1024) ?
                        1024 : (data_len-[bytesWritten intValue]));

                    uint8_t buffer[len];
                    (void)memcpy(buffer, readBytes, len);
                    len = [(NSOutputStream*)aStream
                              write:(const uint8_t *)buffer
                          maxLength:len];

                    bytesWritten =
                        [NSNumber
                        numberWithInt:([bytesWritten intValue]+len)];

                    if([sendData length] == [bytesWritten intValue]) {
                        sendData = nil;
                        [outputStream
                            removeFromRunLoop:[NSRunLoop currentRunLoop]
                                      forMode:NSDefaultRunLoopMode];
                    }

                    if([bytesWritten intValue] == -1) {
                        NSLog(@"Error writing data.");
                    }
                }
            }
            break;
        }
        ...
}
```

当数据传输完毕后，发出队列会被清空，outputStream 也会从运行循环中移除。当下

一次顾客向员工发送请求时，它会被再次调度。

当请求发出后，顾客应用会等待员工的响应。该响应通过 inputStream 传输，如果有响应，stream:handleEvent:委托方法会接收到 NSStreamEventHasBytesAvailable 事件。代码清单 13-20 介绍了如何处理进来的响应数据。

代码清单 13-20　stream:handleEvent: NSStreamEventHasBytesAvailable 实现(/Apps/Consumer/consumer-help/consumer-help/BonjourBrowser.m)

```
- (void)stream:(NSStream *)aStream
   handleEvent:(NSStreamEvent)eventCode {

    switch(eventCode) {
        ...
        case NSStreamEventHasBytesAvailable:
            if(aStream == inputStream) {
                if(receiveData == nil) {
            receiveData = [[NSMutableData alloc] init];
            }
            uint8_t buffer[1024];
            unsigned int len = 0;
            len = [(NSInputStream *)aStream read:buffer
                                    maxLength:1024];
            if(len) {
                [receiveData appendBytes:(const void *)buffer
                                    length:len];

                bytesRead = [NSNumber
                             numberWithInt:([bytesRead intValue]+len)];

                if(![inputStream hasBytesAvailable]) {

                    // you could optionally keep the 'transaction'
                    // state stored so that you could determine
                    // which object you are expecting.
                    HelpResponse *response;

                        @try {
                            response =
                                [NSKeyedUnarchiver
                                unarchiveObjectWithData:receiveData];

                            NSDictionary *info =
                                [NSDictionary
                                dictionaryWithObject:response
                                forKey:kNotificationResultSet];

                            [[NSNotificationCenter defaultCenter]
                            postNotificationName:kHelpResponseNotification
                            object:nil
```

```
                    userInfo:info];
                }
                @catch(NSException *exception) {
                    NSLog(@"Exception unarchiving data.");
                    NSLog(@"Possible missing / corrupt data.");
                }
                @finally {
                    // clean up
                    receiveData = nil;
                    bytesRead = nil;
                }

            }
        } else {
            NSLog(@"No data found in buffer.");
        }
    }
    break;
...
}
}
```

在确保被处理的流是 inputStream 后，应用会初始化一个 NSMutableData 对象来存储所有接收到的数据。接下来，应用从进入的流中取出 1KB 的数据。在数据读取后，它会被追加到之前创建的 NSMutableData 变量，同时会更新字节数量以反映实际读取的字节数。如果 inputStream 没有可供读取的字节，应用就会使用 NSKeyedUnarchiver 创建一个 HelpResponse 对象，然后通知前端并传递给响应。如果还有可读取的字节，那么委托会继续接收到 NSStreamEventHasBytesAvailable 事件，直到缓冲区满为止。

在收到员工响应后，消息会呈现给客户，如图 13-7 所示，请求视图也会消失。然而，在求助请求视图中，在员工提交请求后，应用会禁用导航栏上的所有按钮，直到出现错误或员工做出回应为止。更复杂的应用可能还需要使用其他一些选项，从而可以使得员工在响应请求时顾客依然可以继续使用应用。一种可能的选择是向 AppDelegate 添加监听器。如果提供的服务允许顾客对请求进行排队，这就变得更为重要了，这类似于票务系统，可能会有几个顾客排在前面。

图 13-7

现在已经实现了两个功能完善的应用：一个会将自身发布为主机，可以通过 Bonjour 探测；另一个使用 Bonjour 浏览并连接到可用的服务。连接后，这两个应用就可以无缝地进行通信了。

13.4　小结

Bonjour 是一项非常棒的技术，可以实现自组织网络，在相同网络的设备间进行数据的共享。本章介绍了 Bonjour 的背景知识、在网络上成功发布服务的步骤，以及浏览并最终连接主机与端点的流程。Bonjour 的使用并不局限于 iPhone 与 iPad；你能通过它探测可以发布 Bonjour 服务的任何类型的设备。设备涵盖的范围从打印机、DVR 到家庭自动化工具。

员工应用与顾客应用提供了良好的框架来将 Bonjour 集成到应用中。然而，你还可以看看其他一些网络工具，从而确保 Bonjour 最适合你的需求。一个是 Game Kit，第 12 章对其进行了介绍。第 8 章介绍了底层网络编程功能，对于某些需求来说可以考虑使用它们。

这是本书的最后一章。我们三位作者非常感谢你购买本书，让我们有机会能与大家分享我们的经验，并且能够坚持到底。我们衷心希望这本书能对你有所帮助。再一次表示深深的感谢。

——Jack Cox，John Szumski，Nathan Jones