内容全面，系统讲解开发企业级iOS应用所需掌握的各项核心技术，以及各种工具和框架的用法，包含大量技巧和最佳实践

实战性强，不仅为各个知识点精心设计了能辅助读者理解的小案例，而且还有能指导读者完整实践的大案例，具备极强的可操作性

杨宏焱 著

Enterprise Application Development for iOS

企业级iOS应用开发实战

企业级 iOS 应用开发实战

杨宏焱　著

图书在版编目（CIP）数据

企业级 iOS 应用开发实战 / 杨宏焱著. —北京：机械工业出版社，2012.12

ISBN 978-7-111-40459-0

Ⅰ. 企…　Ⅱ. 杨…　Ⅲ. 移动电话机—应用程序—程序设计　Ⅳ. TN929.53

中国版本图书馆 CIP 数据核字（2012）第 274687 号

版权所有 · 侵权必究

封底无防伪标均为盗版

本书法律顾问　北京市展达律师事务所

本书内容全面，它不仅详细讲解了开发企业级 iOS 应用所需掌握的各项核心技术，以及各种工具和框架的用法，而且还系统讲解了企业级 iOS 应用开发的流程和方法；实战性强，不仅为各个知识点精心设计了能辅助读者理解的小案例，而且还有能指导读者进行完整实践的大案例，具备极强的可操作性。除此之外，本书还包含大量的开发技巧和最佳实践。

本书分为三部分：基础篇（1～6 章），首先介绍了传统企业级应用与 iOS 企业级应用的区别、iOS 企业级应用程序的架构以及发布方法，然后详细讲解了 iOS 的开发框架、Objective-C 语法的核心要素、Xcode 集成开发环境、Interface Builder 和高级图形界面；核心技术篇（7～17 章），系统深入地讲解了网络、XML 和 JSON、用户数据保存、安全、多媒体、绘图、动画、多点触摸和手势、GPS、重力感应、本地化、多线程、并行编程、通知、通讯簿等与企业级应用相关的核心技术特性，同时也讲解了开源框架 CorePlot；实战篇（18～19 章）以迭代的方式讲解了两个综合案例的完整实现过程，既融合了前面的理论知识，又展现了企业级 iOS 应用开发的流程和方法。

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）

责任编辑：吴　怡

印刷

2013 年 1 月第 1 版第 1 次印刷

186mm×240mm · 26 印张

标准书号：ISBN 978-7-111-40459-0

ISBN 978-7-89433-713-9（光盘）

定　价：69.00 元（附光盘）

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991　88361066　投稿热线：（010）88379604

购书热线：（010）68326294　88379649　68995259　读者信箱：hzjsj@hzbook.com

前　言

为什么写这本书

随着我国3G网络和移动互联网的兴起，许多传统的企业应用正在从桌面向移动终端扩展，移动办公、移动营销、移动作业等需求日渐强烈。

有迹象表明，传统的互联网正在向移动互联网发展。根据摩根士丹利发布的全球互联网发展趋势报告（Mary Meeker 2010）显示：全球互联网发展趋势正在由PC-Internet向Mobile-Internet转变，手机在某种意义上已经主导着互联网的发展，新兴的下一代互联网，即Mobile 2.0正在崛起，这完全得益于移动通信技术的迅猛发展。这不仅仅是一场由最新数码科技与网络技术导致的变革，还是传统企业应用从Internet向移动互联网转移的前提和诱因。

与PC相比，移动终端的这种全新用户体验是不可替代的，在一定程度上吸引了人们从桌面向移动终端的转移。而且，移动互联网也凭借着其出色的业务吸引力和资费吸引力，成为人们生活中不可或缺的一部分。

然而，机遇与挑战并存。对于企业而言，能否将自己的企业应用向移动互联网扩展，仍然存在着巨大的风险。诸如：用户体验改变、企业信息安全和企业机密泄露、移动应用开发中存在的技术风险等。

以苹果iOS为代表的移动应用开发正方兴未艾。iPhone和iPad正式进入中国的时间其实还不到3年（iPhone于2010年7月正式在中国香港上市），国内开发者在苹果商店上淘金的时间比这要早些，但绝对不会超过4年。实际上，App Store 拥有的历史还不到5年（App Store正式上线时间是2008年7月11日）。所以说，iPhone 应用开发仍然有着无限的潜力，称为“历

史”恐怕为时尚早。因此，作者选择了以 iOS 为目标平台的企业移动应用开发作为本书讲述的主题。

作者于 2009 年起开始接触 iOS 开发。对于一个多年奋战在企业应用开发第一线的开发人员来说，iOS 企业开发是一个全新的领域。完全陌生的 Mac OS X 操作系统，别扭的 Xcode IDE 和 Interface Builder，古怪的 iOS……由于不知道什么是开发证书和代码签名，甚至在第一次调试我的 3.5 英寸屏幕的 iTouch 时，都是充满了坎坷。

当我历尽千辛万苦，终于逐渐步入 iOS 开发这座大门。然而，却又面临新的问题：“企业应用是否能向 iOS 平台进行迁移？”

从我第一天接到任务起，这样的怀疑就在我心中存在。iOS 实在是太封闭了，相对于 Android 这样的开放平台，iOS 平台对企业开发人员的限制实在是太多。而且，App 商店始终只是游戏开发者的天堂，90%以上的个人开发者把自己的目光盯在了游戏、娱乐等个人应用领域，企业应用根本无法登上 App 商店的大雅之堂。原因也很好理解，企业应用不会为苹果公司带来可观的利润分成，App 商店的盈利模式是基于应用下载量的。一个企业应用会有多少用户？几百万？几千万？不，大多数企业应用的用户量不会超过 5 位数。没有庞大的用户群，就不会给 App 商店带来丰厚的利润分成。实际上，我第一次向 App 商店提交一个企业应用时，被无情地拒绝了。苹果公司给的理由很直白：“你的应用只针对有限的用户群”，换句话说“苹果将无利可图”。

幸好苹果公司提供了“企业开发程序”（企业版 IDP），虽然购买“企业开发程序”需要 299 美金一年，但对于一个真正决心将企业应用向 iOS 移动终端扩展的企业来说，还是负担得起的。“企业开发程序”不需要苹果公司审核，使用“企业开发程序”部署应用不需要经过 App 商店，企业可以任意分发给自己的用户，苹果公司也不会找你要一分钱。

可以预料，在将来一段时间内，国内的 iOS 个人应用开发者将不断向企业开发领域转移。实际上，苹果商店中个人应用的数量已经饱和，开发者的生存空间将逐渐变得狭小。大量的同质竞争直接导致了 App 商店的生态环境恶化，同类应用竞争激烈。为了保持 App 商店的竞争力和盈利模式不趋于低质化，苹果公司今后对商店应用的审核将越来越趋于严格，开发者想在商店中获利和生存的难度将越来越高。国内的公司及个人开发者会逐渐将目光转移到位于 App 商店生态圈之外的企业应用，iOS 企业应用将成为今后新的利润增长点。

最近，企业移动应用开发出现了一些新的趋势。最新的企业移动应用，有从 Native App（即本地代码）向云发展的趋势，比如 HTML5、虚拟桌面。HTML 5 充分利用移动终端的浏览器（如 Safari 或 IE）和网络连接能力，来访问企业服务，并实现“一次开发，跨平台共享”的目的。而虚拟桌面则利用“数据中心”进行桌面的扩展，将客户端的数据、资源和图像放到了“云”上，iOS 客户端则通过网络访问个人桌面。虚拟桌面一般作为企业的“云计算”解决方案进行实施，市场上比较成熟的产品主要来自 VMWare、Citrix、Microsoft 和 Oracle 等几大厂商，它跟开发人员没有太大的关系。还有一种趋势就是“服务器配置+中间代码+本地代码”。开发人员在服务器上以配置的方式产生出中间代码，然后服务器将中间代码编译为多个平台的本地代码（iOS、Android、Sybian、Windows），然后分别部署，以此实现跨平台的目的。

这些趋势的出现，从一定程度上试图解决当前本地代码开发（iOS 开发、Android 开发）的弊端，但效果还难以令人满意，比如都有性能下降、用户体验差、网络带宽占用大等缺点。因此，就目前来说，移动应用开发仍然是以本书介绍的本地代码开发为主流。

本书特色

本书是作者多年开发经验的总结，很多内容来自作者在 CSDN 上的博客，书中不少内容是经验之谈。本书根据 iOS 操作系统更新频繁的实际情况，针对新的 SDK 版本进行了内容上的调整（本书内容适用于 SDK 4.0~5.0，本书所有代码在 Xcode4.3 下编译通过）。在介绍每一种 SDK 框架的同时，注重扩展，在继承的基础上进行创新，而不是一味复制、粘贴代码。

本书具有如下特点：

- 主题明确，以“iOS”和“企业开发”为主题， 但并没有将二者割裂开来，而是将二者紧密联系、互相呼应。首先由浅入深介绍了整个 SDK 框架层次，包括 Objective-C 语言简介、SDK 的构成、Foundation 框架、UIKit 框架、QuartzCore、CoreAnimation 以及其他第三方扩展框架，然后对在企业应用中一些需要特别讨论的方面（如安全、网络、APN、多线程等内容）进行专门的论述。撇开企业开发的特色不谈，本书也完全可以作为一本 iOS 开发的经典教材。
- 理论和技术兼顾。许多 iOS 开发书籍，轻理论，重技术，往往只告诉你怎么做，而不告诉你为什么要这样做，难以让读者在理解的基础上加深记忆。而本书以理论为纲，以技术为体，从基本理论到具体使用的技术都一一道来，不仅告诉你怎么做，而且将每一种技术的来龙去脉阐述清楚。在讲解具体技术的同时，不时穿插着小的知识点，让读者进一步拓宽相关的背景知识。
- 详细分析代码，实用性强。作为编程类书籍，免不了有大量的代码。但本书对多数代码都进行了阐释，重点内容还会有专门的标注，如“提示”、“注意”等，以提醒读者注意，或者及时回顾前面的知识点。本书中的每一个示例程序，都收录到本书的随书光盘中。所有的程序都经过作者认真调试，可以直接运行。

合适阅读本书的人

本书适用于以下读者：

- 从未接触过 Objective-C、从其他语言转向 iOS 开发、有一定面向对象编程基础的程序员。
- 正准备转向企业移动应用开发的 iOS 应用程序开发人员 。

如何阅读本书

这是一本讲述 iOS 和企业应用开发相结合的书，介绍如何在 iOS 上进行企业应用的开发

及分发、部署。本书从一个企业应用开发者的角度出发，以实现企业移动办公和3G应用为宗旨，介绍如何充分发挥苹果新一代操作系统 iOS 的优势和 iPhone 手机的软、硬件特性将企业应用扩展到 iOS 平台。在最后一章以 step by step 的形式介绍了一个实战项目，以达到理论与实践结合的目的。本书也对苹果 Cocoa 框架和其他第三方开源框架进行了深入介绍。

针对本书面对的两种主要读者，我们建议如下：

对于本书第一类读者，即“从未接触过 Objective-C、从其他语言转向 iOS 开发、有一定面向对象编程基础的程序员”，本书提供了一个快捷的 Objective -C 语言入门，以及一个简单易读而又务实详尽的 iOS SDK 入门教程；本书的全部章节都将有助于读者尽快对 Objective -C 及 iOS SDK 有一个全面的了解，并迅速跨入 iOS 开发的大门。

对于本书第二类读者，即“正准备转向企业移动应用开发的 iOS 应用程序开发人员”，可以省略阅读本书部分章节，例如第2、3、4、5、6各章，但本书其他一些章节具备了良好的参考价值，例如：第10章以后的各章，在这些章节中，有部分内容是其他参考书中难以见到的，可以有选择地阅读相应章节。

本书共分19章，主要内容如下：

基础篇

第1章介绍了企业应用的概念，什么是 iOS 企业应用，iOS 企业应用的框架及构成，特别是对于苹果 iOS 企业证书申请和 iOS 企业应用程序的部署方式（In-House、Ad-Hoc、OTA)进行了详细的介绍。

第2章介绍 iOS SDK，包括其框架和构成。iOS SDK 是 iOS 开发中最为重要的工具和武器，每个 iOS 开发人员都必须熟悉并深刻理解它。

第3章介绍 iOS 开发语言 Objective-C。对于没有接触过这种语言的读者，将在本章对 Objective-C 有一个全面的理解。本章从两个方面对 Objective-C 进行了介绍，即 Objective-C 的 C 语言特性和面向对象特性。也对 Objective-C 的一些现代语言特性，如块编程（函数式编程中的主要内容）、反射（运行时支持）和可变参数也进行了介绍，这些内容在其他书籍中是比较罕见的。

第4章介绍 Xcode IDE。从 Xcode 4.0 开始，苹果对其功能和界面进行了全新的设计，把 Interface Builder 完全整合到 Xcode 中，使程序员的开发效率更高。

第5章单独对 Xcode 中的 Interface Builder 进行了进一步介绍，特别是 Assistant Editor 的出现，与之前的版本相比，大大简化了开发人员进行各种连接（IBOutlet 和 IBAction）的操作。

第6章介绍 UIKit 以及 UIKit 中包含的一系列最基本的 UI 组件，此外，介绍了如何在 UIKit 的基础上进行扩充，创建自己的自定义组件库。

企业应用篇

第7章到第10章，依次从网络、XML/Json、数据存储、安全这几个方面进行介绍。这些内容中，有相当一部分是企业开发人员早已熟知的领域（如网络、XML/Json、数据存储和安全）。这些章节结合 iOS 自身的特点进行详细的阐述，包含安全沙箱、嵌入式数据库以及 iOS

安全框架等内容。

第 11 章介绍 Cocoa 的多媒体、Quartz 2D 和 Core Animation 框架。

第 12 章介绍 Cocoa Touch 特有的多点触摸和手势识别。

第 13 章介绍如何利用 iPhone 的多语言支持实现应用程序的国际化。

第 14 章涉及两个方面：传统的线程编程和并行编程 GCD（Grand Central Dispatch）。在企业应用中，免不了要使用多线程。前者是传统的异步编程技术，直接与操作系统底层的线程打交道；后者是 iOS 4.0 以后新的异步编程技术，以一种函数式编程的方式，达到让系统自动进行线程管理的目的，从而避开了线程编程的复杂性。

第 15 章介绍通知、本地通知和远程通知。通知是多个对象间进行对话的机制，但耦合性低于直接的方法调用。本地通知和远程通知是两种不同的进程唤醒技术，前者由系统来唤醒，后者通过 RPC（Remote Process Calling）唤醒。

第 16 章介绍开源框架 Core Plot。Core Plot 是著名的 2D 图形框架，用于绘制散点图、柱状图和饼图等图表。

第 17 章针对 iOS 特有的硬件特性进行介绍，如通讯簿、相机、加速计和 GPS。

实战篇

第 18 章，介绍“企业 APN”在企业中的应用，以及使用“企业 APN”网络对 iOS 客户端的一些特殊要求。该章实际上包含了一个实战项目，即一个简单的 APN 切换工具（同时也提供了简单的网络状态检测）。在这个实战项目中，涉及了广泛的内容和前面诸多章节中介绍的知识，诸如后台任务、配置描述文件、BSD Socket 编程、网络检测、Safari 阻塞和并行编程 GCD。

第 19 章以案例导航的方式介绍了一个实战项目，指导读者从用户的实际需求出发，结合本书中讲述过的理论知识和技术，开发一个完整的 iOS 邮件客户端，使读者对企业应用的开发有直观的认识。

结语

本书从开始选题、写作至最终定稿，总共花费了作者 14 个月以来所有的业余时间。在本书写作的过程中，得到了许多人的无私帮助和大力支持。

首先要感谢的是吴怡编辑。她负责本书所有文字内容、图片及格式，她总是不厌其烦地向作者指出本书中的每一处错误。 直至最终定稿前，她仍然和作者对书中的内容进行讨论，甚至是逐字逐句地讨论，以使本书不至于出现太多谬误或引发读者产生歧义。

其次要感谢我的同事们。在他们身上，我学习到了许多。我在工作中取得的每一点进步，都离不开他们的支持和指导，他们敬业的态度和对我的无私帮助，是本书得以成书的动力。

还要感谢我的家人。本书是在他们的关怀和支持下才得以面世的，尤其是我的妻子，因为她的理解和默默奉献，使作者从家庭琐事中解脱出来，全身心投入本书的写作当中。

我要感谢始终关注作者博客、不断给作者以鼓励的网友们。虽然直至本书出书之时，我都向他们隐瞒了本书的消息，但他们留下的只言片语，仍然激励着我在知识的海洋中不断前行、探索和攀登。

最后，感谢所有在本书写作过程中，给本书以参考的文献作者和论坛作者。本书参考了大量文献，尤其是苹果文档参考库、苹果开发者论坛和 stackoverflow.com，它们对本书的写作起到了重要的作用。

由于作者的时间、精力及自身水平有限，书中错误在所难免，欢迎广大读者一一指正。作者的联系方式如下：

博客：blog.csdn.net/kmyhy

邮箱：kmyhy@126.com。

目　　录

前言

基础篇

第 1 章　企业应用的话题 / 2

1.1　什么是企业应用 / 2

1.1.1　传统意义的企业应用 / 2

1.1.2　iOS 企业应用 / 3

1.2　iOS 企业应用程序的架构 / 3

1.2.1　服务端 / 4

1.2.2　iOS 客户端 / 4

1.3　iOS 企业应用程序的发布 / 5

1.3.1　iOS 应用程序发布与 App Store / 5

1.3.2　Ad-Hoc 与 In-House 发布 / 6

1.3.3　OTA 无线部署 / 21

第 2 章　iOS 开发框架简介 / 24

2.1　苹果 iOS 简介 / 24

2.2　iOS 框架介绍 / 25

2.3　Cocoa Touch 框架简介 / 25

2.4 搭建 iOS 开发环境 / 27
2.4.1 安装 Mac OS X 操作系统 / 27
2.4.2 下载安装 SDK / 33
2.5 写一个 iPhone 程序 / 33
2.6 在模拟器上运行应用程序 / 39
2.7 在 iPhone 上运行应用程序 / 39
第 3 章 Objective-C 语法简介 / 42
3.1 Objective-C 的 C 语言特性 / 42
3.1.1 一个简单的 Hello World / 42
3.1.2 Objetive-C 是另一种 C / 43
3.1.3 数据类型 / 44
3.1.4 常量、变量和宏 / 50
3.1.5 #include 和#import / 51
3.1.6 函数 / 51
3.1.7 分支和循环 / 51
3.2 面向对象的 C / 51
3.2.1 类和对象 / 51
3.2.2 消息机制 / 54
3.2.3 Objective-C 的内存管理 / 55
3.2.4 类别和协议 / 57
3.2.5 反射机制 / 59
3.2.6 谓词 / 62
3.3 MVC 模式 / 65
3.4 KVO 模型 / 65
3.4.1 注册 KVO / 66
3.4.2 接收变更通知 / 67
3.4.3 发送变更通知 / 67
3.5 块编程 / 68
3.5.1 块的特点 / 68
3.5.2 Objective-C 中的块 / 69
3.6 可变参数 / 71
3.7 本章小结 / 73
第 4 章 Xcode 集成开发环境 / 74
4.1 创建第一个 Xcode 应用程序 / 74
4.2 构成应用程序的那些东西 / 76
4.2.1 Info.plist 和 pch 文件 / 76
4.2.2 Xib 文件 / 77

4.2.3　资源文件 / 77
4.2.4　源代码文件 / 77
4.2.5　项目和目标 / 77
4.2.6　Frameworks / 80
4.2.7　应用程序的文档目录和临时文件夹 / 81
4.3　了解 Xcode 为我们做了些什么 / 83
4.3.1　main.m / 83
4.3.2　应用程序委托 / 84
4.4　在 Xcode 中添加 View Controller / 84
4.5　在 Xcode 中添加框架 / 89
4.6　Xcode 使用技巧 / 90
4.6.1　自动完成 / 90
4.6.2　查找和替换 / 91
4.6.3　快速帮助 / 91
4.6.4　快照 / 91
4.6.5　书签 / 91
4.6.6　使用导航条 / 92
4.7　本章小结 / 92

第 5 章　Interface Builder / 93

5.1　IB 和 xib、nib 文件 / 93
5.2　初识 IB / 94
5.3　使用 IB 创建图形界面 / 95
5.3.1　控制器和视图 / 95
5.3.2　基本控件介绍 / 99
5.4　连接 / 100
5.4.1　IBOutlet 连接 / 100
5.4.2　IBAction 连接 / 102
5.4.3　委托连接 / 103
5.4.4　使用 Assistant Editor 创建连接 / 105
5.5　本章小结 / 106

第 6 章　高级图形界面 / 107

6.1　应用程序多视图的导航 / 107
6.1.1　UITabBarController / 107
6.1.2　UINavigationController / 110
6.1.3　窗体导航应用实例 / 114
6.2　表视图 UITableViewController 的应用及其扩展 / 116
6.2.1　简单的表视图控制器 / 116

6.2.2 UITableView 的数据源和委托 / 117
6.2.3 分组表视图 / 119
6.2.4 可折叠的分组表视图 / 121
6.3 扩展 UIKit / 131
6.3.1 扩展日期挑选控件 / 131
6.3.2 扩展单选按钮和复选按钮 / 133
6.3.3 扩展下拉列表框 / 135
6.3.4 封装自己的控件库 / 137
6.4 翻页控件和翻页控制器 / 142
6.4.1 UIPageControl / 143
6.4.2 UIPageViewController / 147
6.5 本章小结 / 152

企业应用篇

第 7 章 网络 / 154
7.1 使用 NSURLConnection 获得网络数据 / 154
7.2 使用 NSOperation 进行异步请求 / 158
7.3 与网络相关的示例 / 163
7.4 ASIHTTPRequest 框架介绍 / 166
7.4.1 发送同步请求 / 167
7.4.2 发送异步请求 / 168
7.4.3 文件上传 / 169
7.4.4 文件下载 / 172
7.4.5 Cookies 和 Sessions / 176
7.5 编写自己的网络模块类 / 179
7.5.1 PostRequest 类 / 179
7.5.2 NetworkModule 类 / 181
7.5.3 测试 NetworkModule / 185
7.6 本章小结 / 186
第 8 章 XML 和 Json / 188
8.1 Cocoa 与 XML 解析 / 188
8.1.1 NSXMLParser / 188
8.1.2 NSXMLParserDelegate / 189
8.2 TBXML / 190
8.3 libxml / 191

8.3.1　在项目中使用 libxml / 192
8.3.2　libxml 应用实例 / 192
8.4　GDataXML / 202
8.5　Json 和 SBJson / 218
8.5.1　在项目使用 SBJson / 218
8.5.2　SBJson 使用示例 / 218
8.6　本章小结 / 219
第 9 章　保存用户数据 / 220
9.1　文件的持久化 / 220
9.1.1　保存到 plist 文件 / 220
9.1.2　NSUserDefaults / 221
9.1.3　归档 / 224
9.2　数据库 / 226
9.2.1　嵌入式数据库 SQLite3 / 226
9.2.2　使用 Core Data / 228
9.2.3　使用 PLDatabase 访问数据库 / 232
9.3　本章小结 / 236
第 10 章　安全 / 237
10.1　iOS 安全框架简介 / 237
10.1.1　证书、密钥和信任服务 / 237
10.1.2　在 iPhone 中使用 X.509 证书 / 238
10.2　使用 SSL 和服务器通信 / 244
10.3　OpenSSL / 245
10.3.1　在 iOS 中使用 OpenSSL 库 / 245
10.3.2　OpenSSL 应用实例——使用 OpenSSL 进行 MD5 加密 / 248
10.4　CommonCrypto / 250
10.5　本章小结 / 252
第 11 章　多媒体、绘图及动画 / 253
11.1　播放视频 / 253
11.2　播放音频 / 254
11.3　Quartz 2D / 255
11.3.1　图形上下文 / 255
11.3.2　路径 / 256
11.3.3　变换 / 257
11.3.4　图案 / 261
11.3.5　阴影 / 262

11.3.6 渐变 / 263
11.3.7 透明图层 / 264
11.3.8 位图及遮罩 / 264
11.4 Core Animation / 267
11.4.1 隐式动画 / 267
11.4.2 显式动画 / 268
11.5 本章小结 / 269

第 12 章 多点触摸及手势 / 270

12.1 手势识别器：UIGestureRecognizer 类 / 270
12.2 创建手势识别器 / 272
12.3 实现图片的拖动及缩放 / 276
12.4 本章小结 / 279

第 13 章 本地化 / 280

13.1 iPhone 的本地化支持 / 280
13.1.1 国家代码和语言代码 / 280
13.1.2 本地化文件夹的匹配 / 281
13.2 本地化应用程序 / 281
13.2.1 使用 NSLocaledString 本地化字符串 / 281
13.2.2 本地化图像 / 285
13.2.3 本地化 xib 文件 / 285
13.2.4 本地化应用程序名称 / 285
13.3 示例 / 285
13.4 本章小结 / 289

第 14 章 iOS 多线程和并行编程 / 290

14.1 多线程 / 290
14.1.1 NSThread / 291
14.1.2 RunLoop / 293
14.2 并行编程 / 296
14.2.1 Dispatch Queue / 296
14.2.2 将任务加入 Dispatch Queue / 297
14.2.3 Dispatch 源 / 298
14.3 后台任务 / 301
14.4 本章小结 / 303

第 15 章 通知、本地通知和远程通知 / 304

15.1 通知 / 304
15.2 本地通知 / 307

15.3 远程通知 / 315
15.3.1 Apple Push 简介 / 316
15.3.2 准备使用 APNs / 316
15.3.3 准备接收推送通知 / 320
15.3.4 创建 Push Notification Provider / 322
15.4 本章小结 / 325
第 16 章 开源框架 Core Plot / 327
16.1 编译 Core Plot 框架 / 327
16.2 使用 Core Plot SDK / 327
16.3 安装 Core Plot 帮助文档 / 328
16.4 图表的构成 / 329
16.5 类图 / 330
16.6 使用 Core Plot 绘制折线图 / 331
16.7 使用 Core Plot 绘制柱状图 / 335
16.7.1 绘制基本的柱状图 / 335
16.7.2 固定坐标轴 / 336
16.7.3 显示数据点的值 / 338
16.7.4 显示网格线 / 339
16.8 使用 Core Plot 绘制饼图 / 339
16.8.1 饼图的绘制 / 340
16.8.2 显示每个扇形的比例 / 341
16.8.3 剥离扇形 / 341
16.8.4 显示图例 / 342
16.8.5 响应事件 / 343
16.9 自定义 Core Plot 主题 / 343
16.10 本章小结 / 346
第 17 章 通讯簿、GPS 和重力感应 / 347
17.1 通讯簿 / 347
17.1.1 Address Book UI / 347
17.1.2 Address Book / 348
17.1.3 联系人中文姓氏排序 / 350
17.2 GPS 和 CoreLocation / 351
17.3 重力感应 / 353
17.4 地理编码 / 355
17.5 本章小结 / 356

实战篇

第 18 章　企业 APN / 358

18.1　企业 APN 的建设 / 358
18.2　iPhone 与 APN / 359
18.3　配置描述文件 / 360
18.4　在 iPhone 上实现一个 HTTP 服务器 / 362
18.5　后台任务与无限后台任务 / 365
18.6　实现 APN 切换 / 368
18.7　检测网络状况 / 369
18.8　Safari 阻塞 / 373
18.9　本章小结 / 377

第 19 章　iOS 企业应用实战 / 378

19.1　应用场景与功能概述 / 378
19.2　应用程序架构 / 378
19.3　服务器端 / 378
19.3.1　环境搭建 / 378
19.3.2　实现登录接口 / 379
19.3.3　实现企业通讯簿接口 / 379
19.3.4　实现收件箱接口 / 380
19.3.5　实现附件上传接口 / 380
19.3.6　实现附件下载接口 / 380
19.4　iPhone 客户端 / 381
19.4.1　实现登录 / 381
19.4.2　查看收件箱 / 383
19.4.3　邮件浏览 / 387
19.4.4　新建邮件 / 389
19.4.5　正文输入界面 / 391
19.4.6　通讯簿 / 392
19.4.7　附件文件的上传 / 397
19.5　本章小结 / 399

第 1 章　企业应用的话题

本书是一本关于 iOS 企业应用开发的书。在本书开篇，首先讨论一下企业应用的话题。包括：什么是企业应用、iOS 企业应用、iOS 企业应用中所使用的应用程序发布方式 Ad-Hoc 和 In-House，以及 iOS 4.0 以后新增的无线部署功能。

1.1　什么是企业应用

iPhone 开发是一个新兴的话题，对于“企业应用”和“非企业应用”，它并没有很清晰的划分。这里借用了传统意义上的企业应用概念，试图阐述清楚如何区分 iOS 企业应用，以及 iOS 企业应用的定义。

1.1.1　传统意义的企业应用

据 IDC 统计，在过去的 10 年中，全球企业在信息系统上一共投资 18 万亿美元。巨大的投资为企业建立了众多信息系统，以帮助企业进行内外部业务的处理和管理工作。根据 METAGroup 的统计，一家典型的大型企业平均拥有 49 个应用系统。虽然迄今为止，“企业应用”都没有一个明确的定义，笔者认为企业应用是企业环境中的特定系统，例如：

ECS（电子商务系统），企业通过实施电子商务实现企业经营目标，电子商务系统提供了网上交易和管理等全过程的服务，如网上订购、网上支付、电子账户、服务传递、意见征询、业务管理等各项功能。

ERP（企业资源规划）系统，指建立在信息技术基础上，以系统化的管理思想，为企业决策层及员工提供决策运行手段的管理平台。企业通过企业资源规划系统能实现企业供应链管理、精益制造、敏捷制造以及整个生产过程的计划、控制、采购、销售、成本核算的管理目标。

CRM（客户关系管理）系统，企业利用信息技术（IT）和互联网技术实现对客户的整合营销，是以客户为核心的企业营销的技术实现和管理实现。客户关系管理注重的是与客户的交流，企业的经营是以客户为中心，而不是传统的以产品或以市场为中心。

OA（办公自动化）系统，它是利用计算机技术提高办公效率，实现办公自动化处理的系统。它采用 Internet/Intranet 技术，和工作流的概念，使企业内部人员能方便快捷地共享信息，协同工作，提升日常办公的工作效率，并为企业的管理和决策提供帮助。

DBS（数据库系统），是企业信息化的核心，负责整个企业在经营过程中的数据储存、共享和处理，为其他信息系统提供支撑。

DW（数据仓库）是在数据库已经大量存在的情况下，为了进一步挖掘数据资源、为了决

策需要而建设的数据仓库。数据仓库系统是一个信息处理平台，它从业务处理系统获得数据，并处理数据，从而获得战略信息。

由此可见，只要是在企业信息化环境中运行的应用软件，都可以称为企业应用。

1.1.2 iOS 企业应用

根据摩根士丹利发布的全球互联网发展趋势报告（Mary Meeker 2010）显示：全球互联网发展趋势正在由 PC-Internet 向 Mobile-Internet 转变，手机在某种意义上已经主导着互联网的发展，移动互联网将带来很多新的商业机会。新兴的下一代互联网，即 Mobile 2.0 正在崛起，同样带来令人刺激的软件行业商业机会。

此外，2008 年底中国 3G 牌照正式发放，标志着移动通信 3G 时代终于来临。移动通信网络由 2G/2.5G 向 3G 的过渡，为移动互联网绑上了高速发展的助推器。对国内软件开发商而言，这意味着新的机遇和挑战产生了。

根据工业和信息化部网站发布的数据（中国工业和信息化部 2010），随着中国电信 3G 用户数达到 1000 万、TD 用户数达到 1698 万、中国联通 3G 用户数达到 1166 万，目前我国三家电信企业的 3G 用户数均过千万。截至 10 月底，我国 3G 用户数累计达到 3864 万，环比增长 10.4%，同比增长 295.7%，比 2009 年年底增长 2538 万，10 月新增用户 364.6 万户。TD 用户在 3G 用户中的占比达到 43.9%。

与传统的 2G 和 2.5G 网络相比，3G 网络带宽已高达 300 ~ 600kb/s，比之 512kb/s 的 ADSL 也相差无几，因此诸多应用不再受到带宽限制，诸如：移动办公、个人应用、移动金融、GPS 导航、视频通话，甚至是传统的企业应用 CRM、ERP，也可能运行在手机上。

2007 年 1 月的 Macworld 年度大会上，苹果公司发布了令人期待已久的 iPhone 手机。iPhone 将创新的移动电话、可触摸宽屏 iPod 以及具有桌面级电子邮件、网页浏览、搜索和地图功能的突破性因特网通信设备这三种产品完美地融为一体，引入了基于大型多触点显示屏和领先性新软件的全新用户界面，让用户用手指即可控制 iPhone。iPhone 还开创了移动设备软件尖端功能的新纪元，重新定义了移动电话的功能。

全球互联网向移动互联网的迁移，3G 网络的兴起，新一代智能手机产品尤其是 iPhone 在全球市场中受到热烈追捧，导致企业应用正呈现由传统 internet/intranet 向移动网络/手机终端扩张的趋势。iOS 正是苹果公司为其创新性产品 iPhone 开发的新一代手机操作系统，iOS 企业应用的概念，也因此衍生而来。

1.2 iOS 企业应用程序的架构

本书把 iOS 企业应用定义为传统企业应用向 iOS 手机终端的顺延和扩张。在此定义下，iOS 企业应用由服务端和 iOS 客户端构成，二者间通过 3G 移动互联网（CDMA/TD/WCDMA）连接或通信。

1.2.1 服务端

服务端（企业网络或 Web 服务）实际上为 iOS 企业应用提供企业数据和服务。如果把 iOS 客户端看做是前端应用，则服务端就是后台服务。服务端向前端提供一系列访问传统企业应用的接口，也可以为前端提供企业数据库和业务系统的访问。因此，iOS 企业应用的服务端可能有两层或多层：接口、企业应用、企业数据库。

本书的核心内容是介绍 iOS 开发技术，不会对企业开发技术做过多的介绍。因此服务端代码（企业应用和企业数据库）的开发细节不会在本书中出现，但对于本书中涉及的接口，会提供必要的代码给读者学习。此外，本书中的接口代码是以 Java 编写的，需要读者对 Java 语言有一定的了解。

1.2.2 iOS 客户端

iOS 客户端是一个标准的 iOS 应用，当然它也具备一些企业应用所特有的特点。但无论如何，它不应当是在浏览器中运行的 Web 网页。如果你想找一本介绍如何开发在 iPhone 浏览器上运行的 Web 网页应用程序的书，那么不应该是本书。

本书大部分内容旨在教你开发标准的 iOS 应用程序，这与市面上大部分介绍 iPhone 开发的书籍是一致的，但有一些例外。

首先，作为运行在手机上的 iOS 企业应用来说，安全是尤其需要注意的问题。因为 iPhone 等手持式移动终端所特有的一些特点，比如随身携带、随处可用，不需要登录，容易丢失等等，稍有不慎，就有可能导致企业机密的泄漏。

其次，对于企业应用来说，访问网络的需要，尤其是访问企业网络内部资源，如服务器、数据库等，永远是必不可少的重要内容。无论在任何情况下，网络带宽永远是企业的稀缺资源，对于企业应用尤其如此，因此，必须在节省带宽和提高用户体验中进行平衡。本书使用了很大的篇幅来介绍网络访问技术，此外，企业网络的类型（例如 APN 网络）会给 iOS 访问企业数据带来麻烦。由于 iOS 本身的限制，iPhone 在切换 APN 网络时显得不太灵活——iPhone 只能通过.mobileconfig 描述文件切换 APN。你可以在 App Store 上找到一堆的应用，专门用于给 iPhone 提供 APN 切换的功能。因此，本书也会介绍如何在自己的项目中实现一个简单 APN 切换器。

另外，与 App Store 中占据主要份额的游戏应用不同，iOS 企业应用有使用数据库技术的迫切需要——作为企业开发人员，习惯于把业务数据保存在关系数据库中的这一顽疾早已根深蒂固——哪怕我们在客户端使用数据库的目的仅仅是出于把服务端数据缓存到本地的需要。

最后，还需要介绍一下文档和报表的显示。企业办公环境中离不开各种文档：文本、图片、视频和声音，尤其Microsoft 的 Office 文档俨然已成为了企业办公中公文流转的标准格式。如果在 iPhone 手机上竟然无法打开这些最为常见的企业办公文档，这绝对是一场悲剧。而报表和图表，是企业管理中最为常见的数据表现形式和数据分析手段，把企业运营数据以报表图表

的形式进行展示，显然是 iOS 企业应用中应该提供的基本功能。

综上所述，企业开发人员必须充分认识到 iOS 企业应用的特点，结合企业的实际需要，才能开发出一个优秀的 iOS 企业应用。

1.3　iOS 企业应用程序的发布

除了上述特点，iOS 企业应用还有一个显著的特点，就是应用程序的发布方式。iOS 企业应用具有两种发布方式：In-House 和 Ad-Hoc，它们并不经过苹果公司的 App Store 进行发布，而只是在企业内部进行发布。换句话说，不经过苹果商店的应用程序审核程序。

1.3.1　iOS 应用程序发布与 App Store

2008 年 3 月 6 日苹果公司推出了 iPhone 的应用程序开发包（iPhone SDK），吸引了全世界的开发者。2008 年 7 月 11 日，App Store 正式上线，从而开辟了一种前所未有的应用程序销售模式。起初，只有少数的开发者（主要是大型程序开发商）掌握 iPhone 的开发语言，应用程序商店出现的都是一些精品程序。例如，在 2008 年 TIOBE 发布的全球编程语言排名中，iPhone 等使用的 Objective C 一直都在 40 位左右徘徊，使用率为 0.134%；排名第一的为 Java，使用率约为 19%。《连线》杂志评出的 2008 年最佳 iPhone 程序中，排名第一的是来自谷歌的地图程序 Google Earth，Twitter 的 iPhone 手机客户端排名第七。

然而，随着更多开发者尤其是个人开发者的进入，一些开发者开始从 App Store 中获益，并被媒体大肆报道。例如， Freeverse 公司开发的《Skee-ball》游戏在一个月内就赚得 18.1 万美元，而 Freeverse 开发和部署该游戏仅花费了两个月的时间。国内开发者 139.ME 团队的水族箱程序第一天的下载就实现了 300 美元的收入。

此后，App Store 中应用程序的数量呈现出爆炸性增长。2010 年 8 月，Objective-C 已经进入了 TIOBE 的前十大编程语言排行榜，使用率达到 3.15%，并且是增长率最快的语言。App Store 中应用程序的数量，也从 2008 年底的 1 万个，增加到 2009 年底的 10 万个。截至 2010 年 10 月，App Store 已经增加到 30 万个应用程序，下载量更是突破 50 亿。

这些惊人的数字，只能用奇迹来形容，这也让人不难理解：为什么 iPhone 开发会这样火爆。

然而，如果你现在正准备成为或者已经是一个 iPhone 开发人员，你也许知道，要想下载 iPhone SDK，必须注册成为苹果公司的 iPhone Developer（iPhone 开发人员）——这并不是免费的，你需要购买苹果公司的 iPhone Development Program；否则你只能下载一个功能有限的 iPhone SDK。

除此之外，由于苹果公司一贯坚持的“精品应用”策略，苹果公司没有开放 iPhone 的操作系统。实际上，在 App Store 上线以前，人们甚至无法在 iPhone 上安装程序——苹果公司通过 App Store 控制着 iPhone 应用程序的发布。如果你想让自己开发的程序安装在 iPhone 上，你必须购买 iPhone Developer Program。

iPhone Development Program 有两个版本：标准版和企业版。

标准版程序价格为 99 美元/年，它提供大量开发工具、资源和技术支持，支持可以通过苹果公司 App Store 发布应用程序。同时也支持在 iPhone/iPod Touch/iPad（不仅仅是在模拟器）上调试代码。

企业版程序价格为 299 美元/年，支持开发企业专用的，在内部发布的企业应用程序。它不支持在苹果公司 App Store 销售和发布应用程序，但它支持不经苹果公司审核的应用程序发布方式。

标准版和企业版这两者的区别很像是 IT 界中“做产品”和“做项目”的提法。做产品依靠销售产品拷贝盈利，卖的拷贝数越多则赚的就越多，标准版程序也是如此，依靠下载数量盈利。而做项目并不需要销售产品拷贝，它为企业提供解决方案，做的企业越多则赚的越多，即企业版程序，不管有多少用户在下载（安装）和使用，每做一个项目（企业）收取的开发费用总是相对固定的。

网络资料中，介绍标准版 IDP 比较多，介绍企业版 IDP 相对较少，而 iOS 企业应用是本书的主题，与企业版程序有着直接关系，因此接下来详细介绍企业版程序（企业版 IDP）。

1.3.2 Ad-Hoc 与 In-House 发布

企业版 IDP 只支持两种应用程序发布方式，Ad-Hoc 和 In-House 发布。尤其是 In-House 发布，是企业版 IDP 所独有的。它使用一种叫做“In House Distribution Provisioning Profile”的文件进行发布，不能发布到 App 商店进行销售，也不需要 Apple 的评审。你可以把 In-House 应用通过任何方式发布给你的企业员工、用户及你认可的其他任何人，尤其适合于企业应用的开发。

1. 申请企业版 IDP

首先，你需要有一个 Apple ID，如果没有则需要事先申请一个。其次，你的企业需要拥有邓白氏编码。如果没有则需要进行注册。

邓白氏编码是美国联邦政府推荐使用的企业机构编码。可以看成是美国版的“组织机构代码”，只不过已经得到了联合国、澳大利亚政府、欧盟及美国政府的承认，成为了全球企业标准。

申请邓白氏编码在 D&B 公司的网站（英文）：http://www.dunsregistered.com/，或者“华夏邓白氏”网站（中文）：http://dnbregistered.com.cn/申请，在网站上提交注册申请后，等待 1 ~ 2 天，对方人员会跟你联系（Email）。如果英文沟通有问题，你可以在华夏邓白氏进行申请，他们会安排中籍文员跟你联系。

邓白氏注册服务有几个版本，收费情况也不一样。笔者一开始收到的邮件是“实地核实”的版本，报价 15200/2 年。后来经与北京苹果公司联系，只需要购买最基本的“标准版”即可，报价 8600 元/2 年，有网友说 2000 ~ 3000 元/年，现在看来是不可能的。联系时一定要强调是购买标准版服务（最便宜），否则你可能会花冤枉钱。

收到邮件后，把申请表、协议打印出来，填好并加盖公章，然后加上企业营业执照副本、扫描为电子的，发给对方邮箱。其实还有一个就是汇款水单（小票），需要发送给对方——这一步其实可以省略，笔者申请时并没有 Email 汇款水单 ，只要对方确认汇款到账即可。

大约 5 ~ 7 天后，对方发来第 2 封邮件，告诉你贵公司的编码。此外还可以在你的公司网站上安装一个邓白氏电子标识——在网页上嵌入指定脚本，则会在页面上显示一个 D&B 图标，点击图标自动链接到 D&B 的网站并呈现你们公司的电子注册信息。

现在，可以申请企业版 IDP 了。登录苹果公司开发者网站 http://developer.apple.com/iphone/，申请 Apple Developer Program，注意要选择 iOS Enterprise Program 链接（在页面底部）。

点击 Apply Now 按钮，进入下一页，点击 Continue 按钮，再进入下一页，选择“Use an existing Apple ID”，点击 Continue。进入下一页，输入你的 Apple ID、密码，登录。

后面就是确认注册协议和填写你的公司资料了（英文）。内容最好同邓白氏申请时一样，否则对方会打电话来确认，要你更改。填写完公司资料，还要填写委托人联系资料。注意委托人应该有权代表公司签字（需要贵公司认可，他们会在电话里确认）。

提交资料后，会在注册的联系邮箱里收到苹果公司的邮件，内容大概是感谢你提交了申请，申请的编号是多少，公司名称、邮箱地址等，如果你想看评审流程，可以登录 Member Center。接下来就是等待苹果公司的电话了。这个过程大概要 2 ~ 3 天，对方会安排懂中文的人员来电话，如果没什么问题，接下来（电话之后几分钟）会收到苹果公司的第 2 封邮件，大意是要你点击邮件中的链接，查看一个协议。

同意协议后，显示一个页面，大意是你所申请的国家不支持在线购买苹果公司产品（在线支付），需要你下载一个 PDF 格式的 Purchase Form（见图 1-1）。

将它打印出来，根据要求填好，然后传真给苹果公司。

注意，国内信用卡支持美元支付的一般是 Visa 卡（如招行）和 Master 卡（如交行），一定要找那种卡上印有“Visa”或“Master”标志的信用卡。

Cvc2 code 是指信用卡背面的那串数字（7 位）的末 3 位。

信用卡地址写申请信用卡时登记的地址。

如果传真机发送国际传真有麻烦，可将 Purchase Form 扫描后用 Email 发给亚洲苹果公司 chinadev@asia.apple.com，请其转交给 Billing 团队。亚洲苹果公司几分钟后自动回复了一封邮件，并在信中附了一个业务流水号：Follow-Up: 149653xxx。下次再给亚洲苹果公司联系时，可以附上这个业务流水号。

然后 3 ~ 5 个工作日后，如果信用卡办理了账户余额变动短信提醒功能，则会收到扣费成功短信（注意美国和中国有时差，很可能是在半夜发送的）。登录邮箱后，果然收到了苹果公司的 2 封 Email，1 封是发票，上面有你的发票号码，单位报账的时候把这封邮件内容打印出来就可以了。另 1 封是激活邮件，告诉你现在你的 IDP 账号已经生效了，你点击那个 login now 按钮可以登录到 member center，这时可以看到你的 developer program overview 的状态已经改变。同时，Peoples 中会包含一个成员，这个成员就是你注册 IDP 时所绑定的开发者账号（Apple ID），同时也是该 IDP 的 Agent（超级管理员，具有发布权限）。

Purchase Form

Apple Developer Programs

To complete the purchase process, fill in all the sections of this form clearly, sign, and fax to +1(408) 862-7602. You will receive an activation email once your order has been processed.

Fax Number: +1 (408) 862-7602

Attention: Apple Developer Programs Billing

1. Select the program you wish to purchase.

- ☐ iOS Developer Program Standard USD $99*
- ☐ iOS Developer Program Enterprise USD $299*
- ☐ Mac Developer Program USD $99*

* Your Order will be charged in US dollars

2. Enter your account information.

Full Name: ______

Company Name *(if applicable*)*: ______

*You must include your Company Name if this information needs to reflect on your purchase invoice

Apple Developer Program Enrollment ID: ______ Person ID: ______

3. Enter your billing information.

☐ Amex ☐ Visa ☐ MasterCard ☐ Discover

Credit card number: ______

Expiration date *(MM/YY)*: ______ CVV/CVC2 Code: ______

Name on card: ______

(Please ensure you provide your name exactly as it appears on your credit card)

Street/House number: ______

City: ______

State/Province: ______ Country: ______

Postal code : ______

4. Cardholder Signature: ______

(Your signature is required for us to process your purchase)

5. Email address to send activation code: ______

Once your order has been processed, your activation code will be sent to the email address provided above. Follow the instructions within the email to activate your Apple Developer Program.

图 1-1 Purchase Form

2. 制作开发者证书

（1）在本机生成证书请求 CSR

从 Dock 栏的“应用程序”→“实用工具”中，打开“钥匙串”应用程序，修改偏好设置如图 1-2 所示。

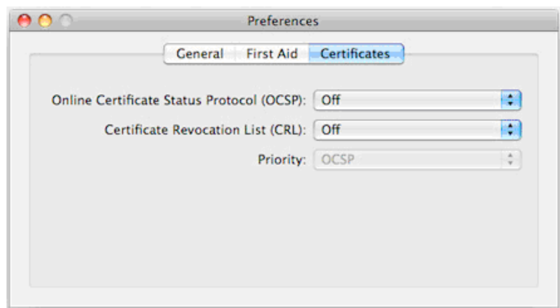


图 1-2 修改钥匙串偏好设置

选择菜单“钥匙串访问→证书助理→从证书颁发机构求证书”，如图 1-3 所示。

注意，如果此时密钥中的某个私钥处于选中状态，则菜单会变为“钥匙串访问→证书助理→用<私钥>从证书颁发机构求证书”，这样制作出来的 CSR 是无效的。

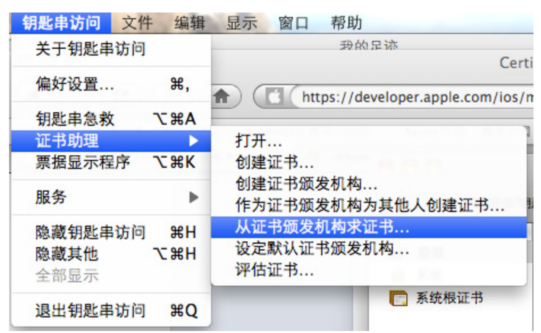


图 1-3　使用证书助理请求证书

输入你的 Email 地址和名字，确保 Email 地址和名字与你注册为 iOS 开发者时登记的相一致。

选中 Saved to Disk（保存到磁盘）单选按钮并勾选 Let me specify key pair information（指定密钥对信息）复选框，然后点击 Continue 按钮，如图 1-4 所示。

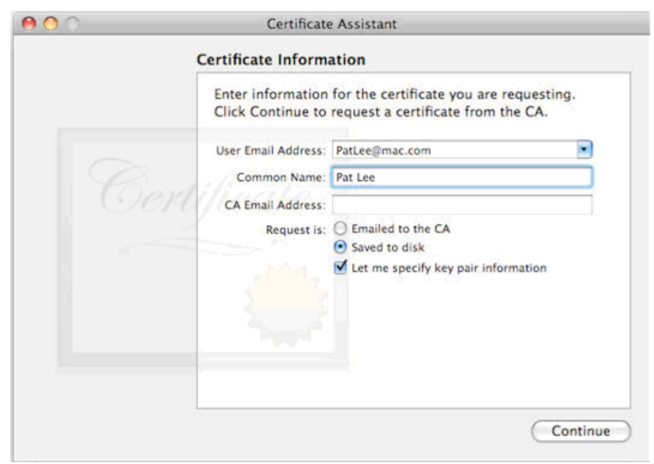


图 1-4　输入 CSR 证书信息

当选择了 Let me specify key pair 复选框之后，会要求你指定文件保存位置。接下来按图 1-5 所示指定密钥对信息。

点击 Continue 按钮，即可生成 CSR 文件。一旦生成 CSR，在“登录”钥匙串中会生成一对密钥对（一个私钥，一个公钥）。你可以在钥匙串的密钥栏中查看。

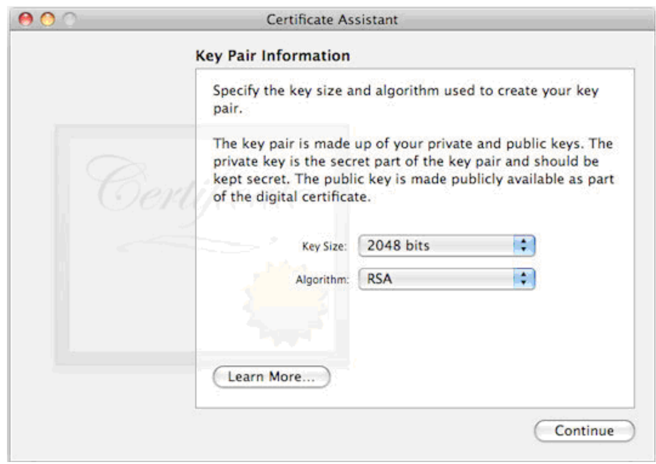


图 1-5　指定密钥对信息

（2）提交 CSR 文件

用企业版 IDP 绑定的 Apple ID（ 跟制作 CSR 时要求输的可能不一致，这里是注册企业版时绑定的 iOS 开发者账号，即 Agent）登录 iOS Provision Portal 。

在 Provision Portal 页面中，依次点击 “Certificates→Development 中的 Add Certificate”，进入图 1-6 所示的页面。

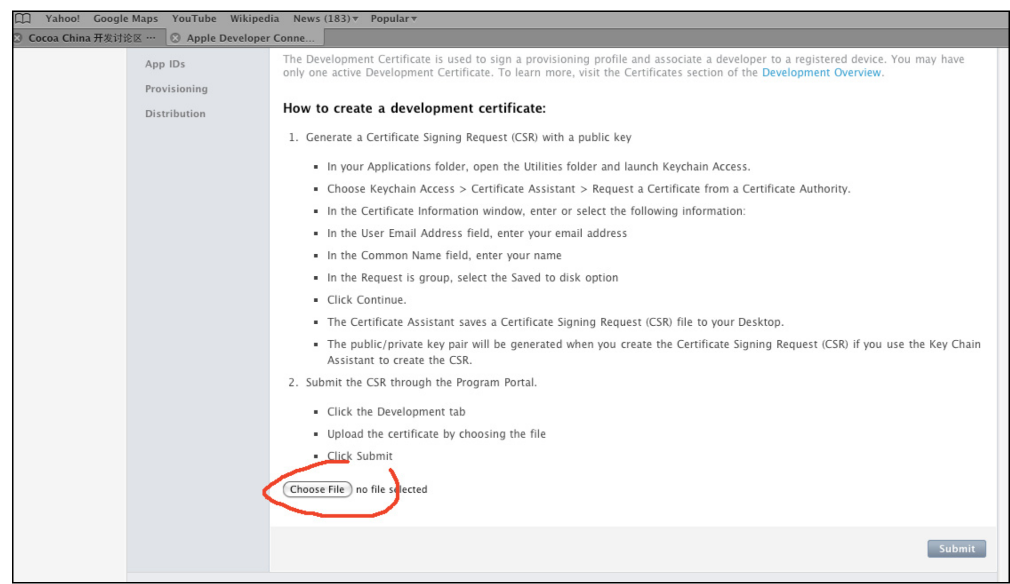


图 1-6　提交 CSR

接下来点击左下角的 Choose File 按钮，选择所生成的 CSR 文件，然后点击 Submit 按钮。如果密钥长度未设置为 2048，Portal 会拒绝 CSR。

提交 CSR 后，Team 管理员（Agent）会收到一封提醒邮件，主题为 Certificate Request Requires Your Approval，提示你需要去同意该 CSR。此时 Agent 需要登录 Portal 去同意该 CSR。但实际上，Agent 也可能根本不需要点击“同意”按钮，Portal 几秒钟后就自动同意了——笔者遇到的情况就是这样的（可能笔者是用 Agent 提交 CSR 的）。

（3）下载并安装开发者证书

用提交 CSR 的 Apple 账号登录 Portal。如果机器上未安装 WWDR 证书，请点击“Certificate→Distribution”中的链接“Saved Linked File to Downloads”，以下载 WWDR 证书，并通过双击 WWDR 证书文件进行安装。

在“Certificate→Development”中， 在 Your Certificate 下会列出当前有效的开发者证书。点击 Download 按钮，即可下载到本机。下载后双击，即可安装到本机。可以在钥匙串“证书”一栏中查看到导入的开发证书。

Team 成员只能下载自己的 iOS 开发证书。Team 管理员有权下载所有成员的公有证书。苹果公司不接受 CSR 中的私钥。私钥仅对创建者有效，并且必须存储在系统钥匙串里。

（4）保存私钥并迁移到其他系统

如果你在多台电脑上进行开发或者重装系统，那么把私钥存储在安全的地方是件很重要的事情。如果没有私钥，你无法在 Xcode 中签名代码并进行真机调试。

钥匙串在生成操作系统 CSR 时，就会在“登录”钥匙串中创建一个私钥。该私钥和你的用户账号绑定，如果重装操作系统导致该私钥遗失，则该私钥无法再次生成。如果你想在多台电脑上开发和调试，你必须将私钥导入到每一台机器上：

在钥匙串访问程序中，选择登录钥匙串的“密钥”。可以看到有许多密钥对，选择与你的开发者证书相对应的私钥（还记得创建 CSR 时要你输入的邮箱地址和名字吗？那个名字会显示在私钥的名字上）。然后选择菜单“文件→导出项目…”，将私钥保存为.p12 格式（Personal Information Exchange）。当提示输入密码时，设置一个密码并记住它，它会在导入.p12 文件时使用。现住，你可以把.p12 文件拷贝到其他机器上并双击它进行安装，这时会提示你输入导出私钥时设置的密码。

这个私钥是重要的。如果你机器重装系统了（或者你想把开发环境迁移到另一台机器上），那么很可能需要重新安装开发环境，包括导入开发者证书。每个开发者证书都是和申请证书（提交 CSR）时的私钥（私钥可以保存在.p12 文件中）是绑定的。如果仅仅是导入了开发者证书文件而机器上没有对应的私钥，则这个开发者证书对这台机器是无效的。

3. 设备 ID

所谓设备 ID（device ID 又称 UDID）是 Apple 设备上的 40 位十六进制码，每台 Apple 设备的设备 ID 都是唯一的，Apple 以此来识别每一台 iOS 设备。

我们通过在 Provision Portal 中录入设备的设备 ID，可以允许开发者在指定真实设备上进行调试。在 Provision Portal 中最多允许输入 100 个设备 ID。

因此，录入设备 ID 是后续制作 Provision Profile 的必需步骤（而 Provision Profile 又是真机调试的必需步骤）。

（1）获取设备 ID

有两种获取设备 ID 的方式：

把 Apple 设备（iPhone 或 iPod）连接电脑，打开 Xcode（以 4.2 版本为例）的 Organizer，如图 1-7 所示。

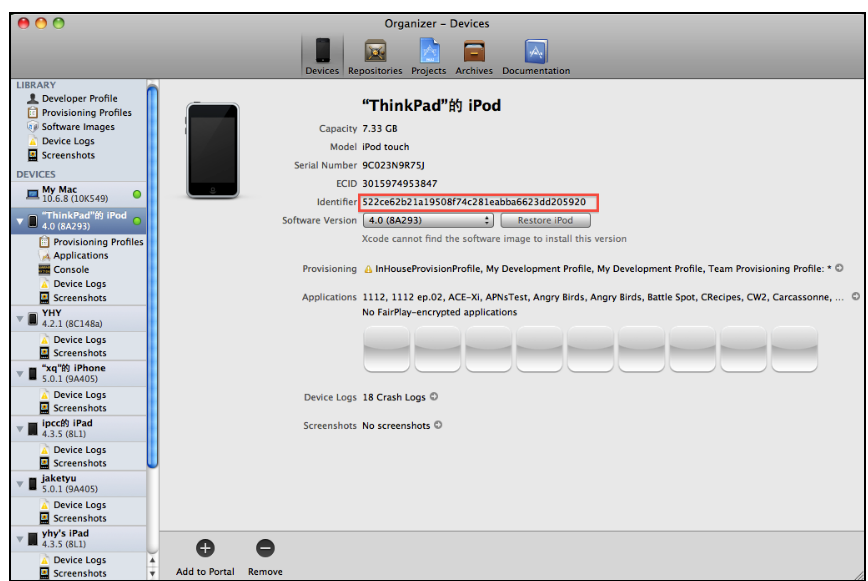


图 1-7　在 Organizer 中查看设备 ID

或者，把 Apple 设备（iPhone 或 iPod）连接电脑，打开 iTunes，如图 1-8 所示。

图 1-8　在 iTunes 中查看设备 ID

那个 40 位 16 进制的数字就是设备 ID。

（2）添加设备 ID

以 Team 管理员登录 Provision Portal，点击 Devices 页面中的“Add Device”按钮，如图 1-9 所示，在其中进行以下设置：

Device Name：设备名称，输入一个描述该设备的名字。

UD ID：即设备 ID。

点击 Submit 按钮即可。

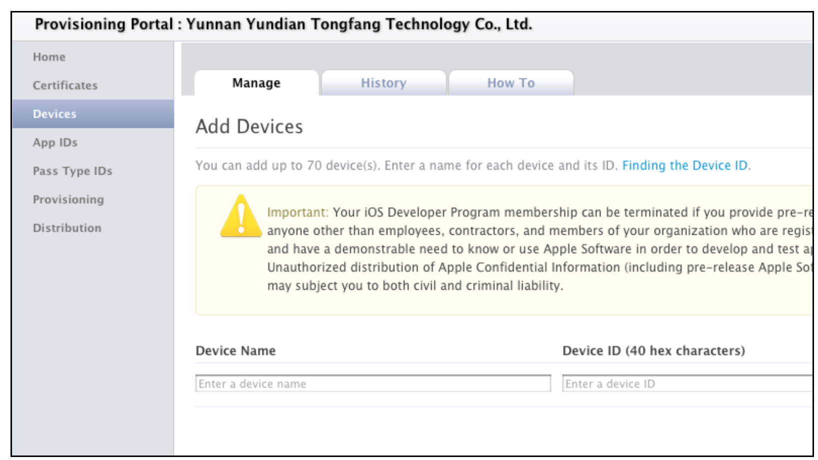


图 1-9　在 Provision Portal 中添加设备 ID

4. 创建 App ID

App ID 是识别不同应用程序的唯一编码。如果你的程序要连接 Apple Push Notification 服务（一种 push 通知），需要用到 App ID。如果应用程序之间要共享钥匙串数据，也会用到 App ID。总之，App ID 在 iOS 开发中很重要。在这里 App ID 的最大用处是制作真机调试用的 Provision Profile（对代码进行签名，它需要提供一个 App ID）。

一个 App ID 由两部分构成：一个 10 位字符的 Bundle Seed ID 前缀，这个 Bundle Seed ID 由 Apple 分配，全球唯一，保证不会重复；一个 Bundle Identifier 后缀，这个 Bundle Identifier 由 Team 管理员指派，Apple 建议用反域名规则命名这个 Bundle Identifier。例如：8E549T7128.com.apple.AddressBook。其中，8E549T7128 是 Bundle Seed ID，com.apple.AddressBook 是 Bundle Identifier。

如果你写了一系列应用程序，它们共用相同的钥匙串（如共用密码），或者根本就不使用钥匙串访问，你可以只创建一个 App ID，所有的应用程序都使用以星号结尾的 App ID。这个星号就是通配符，只能用于 App ID 最后一个字符。例如，这个 App ID 可以是：R2T24EVAEE.com.domainname.* 或者 R2T24EVAEE.*。

以 Agent 或 Team 管理员登录 Provision Portal，点击“App ID”页面中的“New App ID”按钮。如图 1-10 所示，在其中修改如下信息：

- ❑ Description：给这个 App ID 一个名字。如果存在多个 App ID，每个 App ID 需要一个易于识别的名称。
- ❑ Bundle Identifier：如前面所述，Bundle Seed ID 是 Apple 分配的，其实这里只需要你输入 Bundle Identifier。可以使用统配符*。

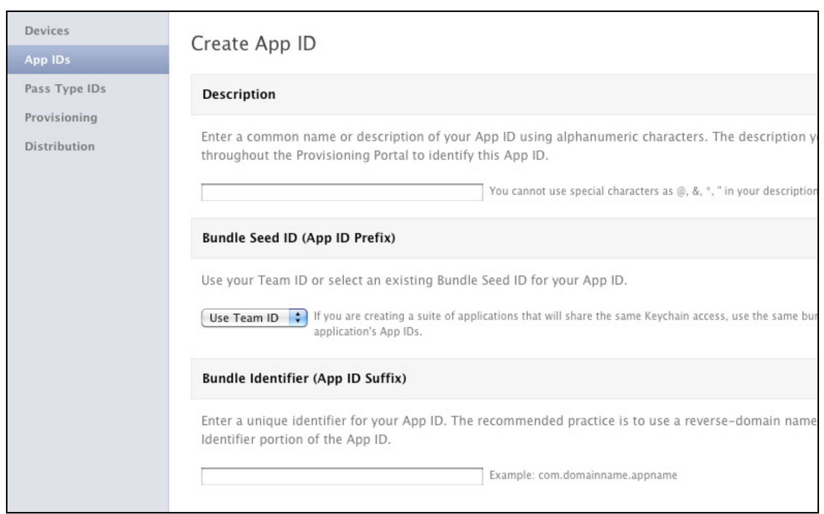


图 1-10 创建 App ID

5. 制作开发者 Provisioning Profile

拥有了开发者证书（Development Certificate），只是表明你有权利在电脑上进行开发，在模拟器上运行程序，但你还不能在 iPhone 上运行你开发的程序。其实如果你只是在模拟器上调试程序的话，要不要开发者证书都无所谓，因为证书只是用来代码签名（Code Sign）的，如果在模拟器上运行的话，你可以选择不签名（don't code sign）。

如果要在真机上调试就不一样了，没有这个 Provision Profile，苹果设备无法安装、运行你开发的程序（这个 Provision Profile 也将随程序一同安装到 iPhone 上）。这个 Provision Profile 中记录了一些信息：开发者证书、开发者 Apple ID、一系列设备 ID（开发者可以使用哪几部设备进行调试——这些设备的设备 ID 要登录到 Portal 上）。

（1）创建开发者 Provision Profile

登录 Provision Portal，点击“Provisioning→Development”，点击 New Profile 按钮，修改以下信息：

- ❑ Profile Name：输入 Profile 的名字，随意。
- ❑ Certificate：选择开发者证书。
- ❑ App ID：选择一个 App ID。
- ❑ Devices：设备 ID 列表。

点击 Submit 按钮，即会生成 Development Provisioning Profile，如图 1-11 所示。

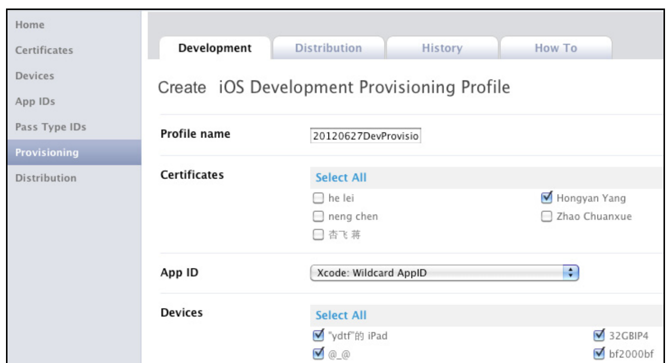


图 1-11　生成开发者 Provision Profile

（2）安装 Development Provision Profile

所有 Team 成员都可以下载 Development Provision Profile。但只有 Profile 中记录了设备 ID 的设备以及 iOS 开发者证书所指定的开发者能够使用这个 Profile。

在 Portal 的“Provisioning→Development”中，点击某个 profile 右边的“download”按钮。下载 profile 后，将下载到的文件拖曳到桌面 Dock 面板的 Xcode 图标上（或者直接拖到 Xcode 的 Organizer 中）。这会将 profile 文件拷贝到~/Library/MobileDevice/Provisioning Profiles 目录。

（3）签名并调试

这需要用到两个文件：证书用于给代码签名，Provisioning Profile 用于真机调试。

在 Xcode （以 4.2 版本为例）中打开项目，选中 Target，打开 info 窗口，在 Build Settings 面板中找到“Code Signing Identify”，打开并点击“Debug”下面的“Any iOS Device”将弹出一个签名文档（即 Provisioning Profile）选择列表，如图 1-12 所示。

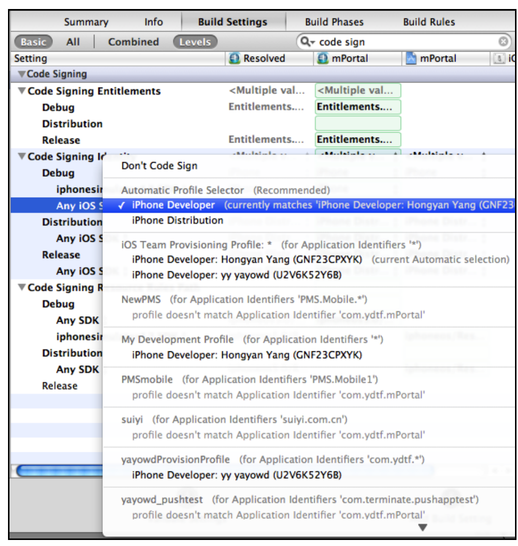


图 1-12　使用 Provision Profile 进行签名

提示：除了可以对“Debug”进行签名，我们还可以对“Release”、“Distribution”等进行签名。这里“Debug”、“Release”、“Distribution”指的是不同的编译版本，在 Xcode 里也叫做 Schema。一个 Schema 是一种编译方案，代表了一个项目在编译时所采用的编译选项，包括编译器选项参数、环境变量、签名文档和编译脚本等。默认情况下，一个 Xcode 项目只有“Debug”和“Release”两种 Schema，分别代表了调试时和发布时的不同编译选项。在 Xcode 里，这两种 Schema 是不一样的。因为程序员在开发调试过程中，对代码进行编译是非常频繁的，这种情况下，应该对编译速度进行一些优化，以节省编译时间，但同时运行效率就会较差一些。但对于发布版本就相反了，这时的编译器应当优先考虑在运行速度上进行优化，而编译速度就会有所下降。此外，你也可以自己加入一些定制的 Schema，比如“Distribution”，从而对编译选项进行一些调优。

在弹出菜单中选择你要用于签名的 Provision Profile（即先前在 Portal 中制作的 Provision Profile），该签名应当和一个开发者证书对应。这个 Profile 就是前面安装的 Development Provision Profile。

注意，有时候 Xcode 会自动根据当前连接的设备的设备 ID 选择一个有该设备调试权限的签名，比如一般是位于 Automatic Profile Selector（灰色）条目下面的 iPhone Developer 项。这在大部分时候是适用的，但有时候 Xcode 的选择并不符合你的意愿。此时就需要手动修改签名。

例如，在图 1-12 中，Xcode 自动选择了“Automatic Profile Selector”下面的“iPhone Developer（current matches 'iPhone Developer:Hongyan Yang...'）”进行签名。这表明将用 iPhone Developer “Hongyan Yang”的数字证书进行签名。一个证书可以绑定多个 Provision Profile。在图 1-12 的例子中，“Hongyan Yang”的证书就会存在于许多 Provision Profile 中：My Development Profile、iOS Team Provisioning Profile 。在图 1-12 中，有的证书是灰色的，表明不能用于当前签名（可能是 Provision Profile 的 IDs 列表中没有包含调试设备，或者 App ID 不匹配，或者证书和私钥不匹配等原因）。

在 Target 的 info 面板（其实就是 info.plist 中的内容）中，还需要设置的 Bundle Identifier。如果你的 App ID 是 A1B2C3D4E5.com.domainname. applicationname（我们在前面创建的 App ID），那么 Bundle Identifier 可以是 com.domainname.applicationname（不需要填写 Bundle Seed ID）。如果 App ID 使用了通配符，比如 A1B2C3D4E5.com.domainname.*，则 Bundle Identifier 可以是 com.domainname.<任意字符>。如图 1-13 所示（以 Xcode 4.2 为例）。

签名完成，你就可以在真机上运行程序了。点击 Xcode 工具栏左上角 Scheme 下拉按钮，从中选择 Device→Debug，然后点击 Build and Debug 按钮，编译并在真机上运行程序。

（4）发布应用程序

发布应用程序需要使用发布证书（Distribution Certificate）。发布证书的制作，跟制作开发者证书的步骤是一样的，只不过使用的是 Provision Portal 的“Certificates→Distribution”功能。

把制作好的发布证书下载、安装到本机。

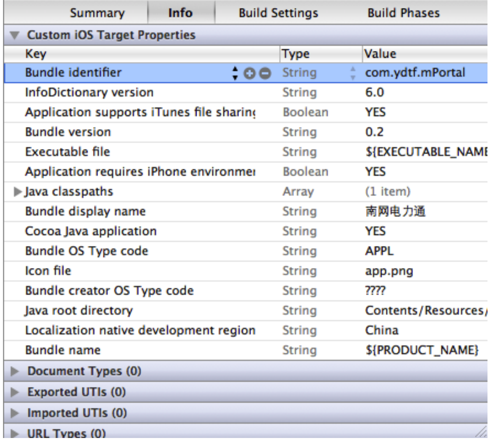


图 1-13　为程序分配 App ID

发布应用程序时使用的是“发布证书”，就如同开发时要使用“开发证书”一样。同理，发布时用的签名文档（即 Provision Profile）也与开发时使用的不太一样。

企业版 IDP 有两种发布方式：In-House 和 Ad-Hoc。两种 Profile 制作步骤稍有区别。而前者（In-House 方式发布）正是企业版 IDP 真正区别于其他版本的 IDP 所在。我们重点介绍 In-House 方式的发布。

6. 制作 In-House 发布的签名文档

以 Team Admin（Agent）登录 Provision Portal，打开“Provisioning Distribution”页面，如图 1-14 所示。

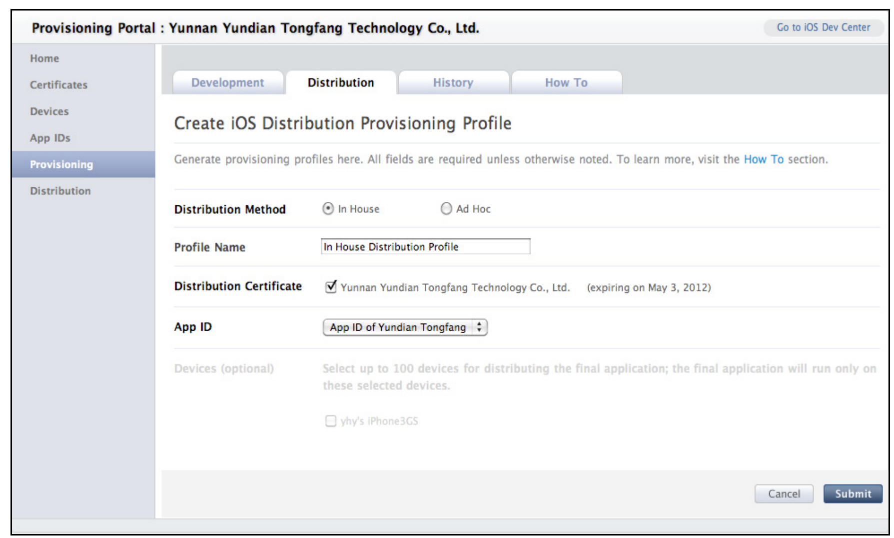


图 1-14　创建 In-House 发布证书

进行如下配置：

- ❑ Distribution Method：发布方式，选择 In House。
- ❑ Profile Name：Profile 名称，用于区别多个 Profile。
- ❑ Distribution Certificate：选择要在 Profile 中绑定的发布证书。
- ❑ App ID：指定一个已有的 AppID。
- ❑ Devices（optional）：要绑定的 device ID。因为 In-House 方式可以在任何 Apple 设备上发布，所以不需要设定 Devices，这一项不可用。

点击 Submit 按钮，生成 Profile。将 Profile 下载到本地进行安装。方法：把 Profile 文件拖曳到 Dock 上的 Xcode 图标。

7. 制作 Ad-Hoc 发布的签名文档

以 Admin 或 Agent 登录 Provision Portal。打开“Provisioning Distribution”页面，如图 1-15 所示。

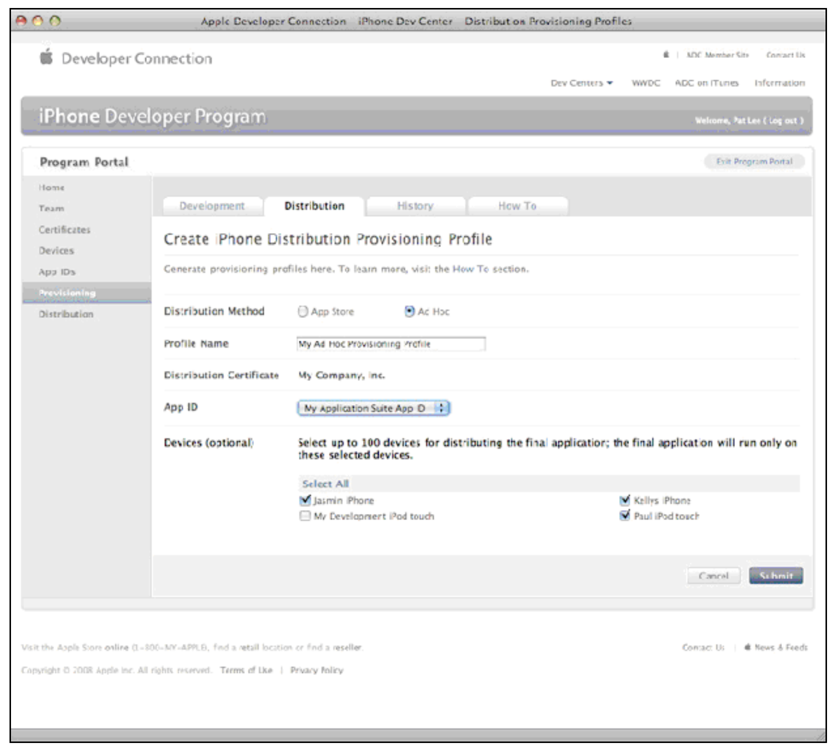


图 1-15 创建 Ad-Hoc 发布证书

与 In-House 方式大同小异，只不过发布方式选择 Ad-Hoc，同时在 Devices（optional）栏勾选要绑定的 device ID，最多可选择 100 个。

点击 Submit 按钮，生成 Profile，将 Profile 下载到本地进行安装。

8. 编译发布版本

打开你的项目。在 Target 的 Builder Settings 面板中，找到 Code Signing Identity 下面的 Release 项。将 Any iOS SDK 指定为你的发布证书（Distribution Certificate），如图 1-16 所示。

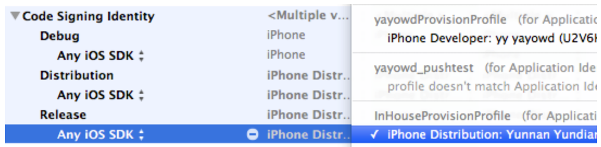


图 1-16　在 Build Settings 中为发布版本进行代码签名

切换到 Info 面板，在 Identifier 栏输入 Bundle Identifier。该 Bundle Identifier 应根据 App ID 填写。

选择菜单“Project→Edit Scheme”，在 Profile 的 Info 窗口中，将 Build Configuration 选择为 Release，如图 1-17 所示。

在菜单栏选择“Product→Archive”。

如果 Archive 是灰色的，请连接设备，在 Scheme 按钮中选择所连接的 iPhone，如图 1-18 所示。

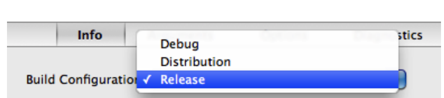


图 1-17　编辑 Scheme

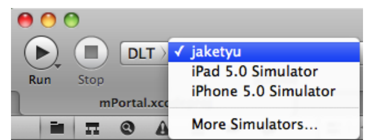


图 1-18　Scheme 请选择真机

这时再次选择“Product→Archive”，Archive 就变成可用的了。

点击“Product→Archive”，编译成功后，会弹出 Organizer 窗口，如图 1-19 所示。

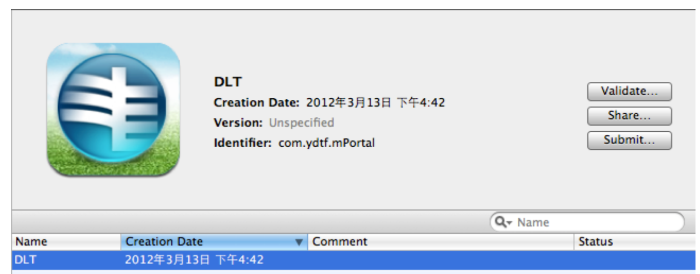


图 1-19　Archive 成功后的 Organizer 窗口

点击 Share 按钮，又会弹出另一个窗口，如图 1-20 所示。

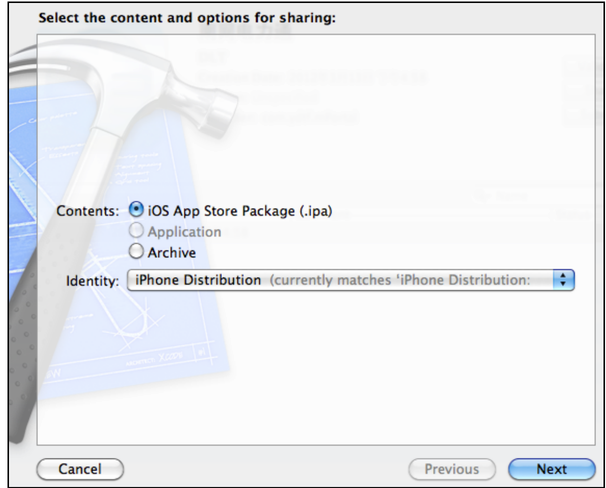


图 1-20 选择 Content 类型

然后，点击 Next 按钮，会弹出新窗口选择 Archive 保存路径，如图 1-21 所示。注意，不要勾选“Save for Enterprise Distribution”选项。

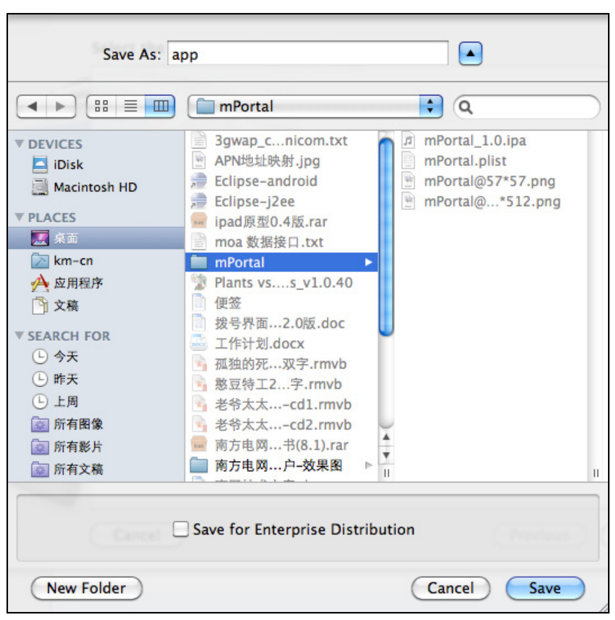


图 1-21 选择保存路径

填入文件名，选择保存路径，点击 Save 按钮，文件会保存在指定地方。打开 Finder，你可以在指定目录下看到生成的.ipa 文件。

9. 安装应用程序

以 Ad-Hoc 或 In-House 方式发布的应用程序，可以将.ipa 文件直接发送给用户。用户可以用两种方式安装：使用 iTunes，或者使用 iPhone 配置实用工具。

（1）使用 iTunes

用户将压缩包中的.ipa 文件拖到 iTunes 的“资料库→应用程序”下，然后和 iPhone/iPod 进行同步。

（2）使用 iPhone 配置实用工具

iPhone 配置工具是完全免费的，你可以从这里下载：

http://support.apple.com/kb/DL926?viewlocale=zh_CN

安装后会在“应用程序/实用工具”中生成一个快捷方式“iPhone 配置实用工具”。

同样，将 iPhone/iPod 连上电脑，打开“iPhone 配置实用工具”，将.ipa 文件拖放到“iPhone 配置实用工具”的“资料库→应用程序”下，然后选中你的 iPhone/iPod，在右边“安装或删除应用程序列表”中，点击某个应用程序右边的“安装”按钮进行安装。

10. 问题及错误

如果 Xcode 出现 Code sign 错误：

Code Sign Errors: profile doesn’t match any valid certificate/private key pair in the default keychain

同时在 Organizer 中出现下列提示：

A valid signing identity matching this profile could not be found in your keychain

则需要把钥匙串中的所有证书和密钥删除，然后重新请求证书、修复 provision profile、下载并安装，一般可以得到解决。

1.3.3　OTA 无线部署

所谓 OTA（Over The Air），是苹果公司在 iOS 4.0 中加入的一种新的企业部署方式，即 iOS 4 的无线部署。无线部署是完全脱离 iTunes 的发布程序的一种方式。 苹果公司扩展了 iOS 的 Safari 的功能，使得 iOS 企业应用可以通过 Safari 浏览器进行部署。Safari 对 URL 地址中的特殊协议 itms-services 进行识别并自动下载 ipa 安装包，并在下载完成后调用 iOS 操作系统的应用程序安装界面（iTunes）进行安装。

“无线部署”专用于企业部署，包括 Ad-Hoc 和 In-House 部署，所以这里你必须使用这两种 provision　profile 文件。下面介绍如何在 Xcode（以 4.2 版本为例）中进行 OTA 部署。

OTA 部署仍然使用“Product→Archive”菜单的功能（正如 1.3.2 节下面的“8. 编译发布版本”节所描述）。但在选择保存路径这一步（如图 1-21 所示步骤），注意勾选“Save for Enterprise Distribution”选项。这样将出现 OTA 部署选项，如图 1-22 所示。

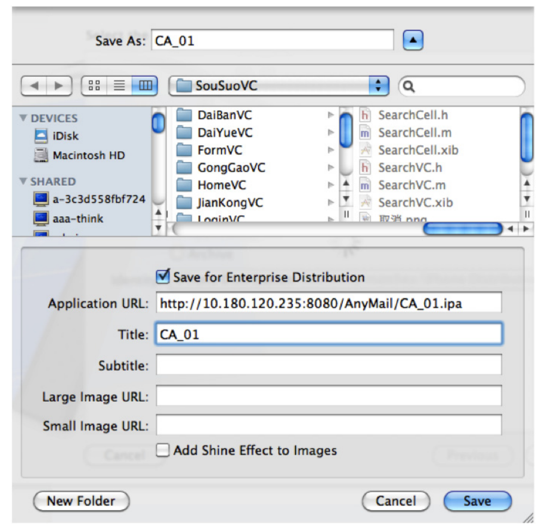


图 1-22　勾选 Save for Enterprise Distribution 将出现 OTA 部署选项

需要填写的各项设置意义如下：

- Application URL：你的.ipa 文件将部署到服务器的哪个位置。当用户通过 Safari 安装你的企业应用时，这个 URL 应指向.ipa 文件所在的地址，必填。
- Title：企业应用名称，必填。
- Subtitle：子标题，可选。
- Large Image URL：大图标（512×512）所在的 URL。大图标用于在 iPad 上安装时显示应用程序图标，可选。
- Small Image URL：小图标（57×57）所在的 URL。大图标用于在 iPhone 上安装时显示应用程序图标，可选。
- Add Shine Effect to Image：会在大/小图标上加上一个光照效果，可选。

填写完后，点击 Save 按钮。Xcode 会在硬盘上产生一个.ipa 文件和一个.plist 文件。这两个文件都 OTA 部署所必需的。其中.plist 文件是一个 XML 文件，保存了你刚才填入的那些配置选项。

将上述两个文件和图标文件（如果有的话），放到 Web 服务器上。注意，.ipa 文件和图标文件所放的位置应该和你在图 1-22 界面中配置的内容一致。至于.plist 文件，你可以让它和.ipa 文件放在一起，也可以单独放在另外一个地方，比如：http://10.180.120.235:8080/yourappname.plist。

另外制作一个 html 文件，如 install.html，内容如下：

```
<html>
<head><title>TextGlowDemo</title></head>
<body>
```

```
<ul>
    <li>
<a href="http://10.180.120.235:8080/AnyMail/InHouseProvisionProfile.
        mobileprovision"> Provisioning File</a>
</li>
    <li>
<a href="itms-services://?action=download-manifest&url=http://10.180.120.235:8080/
        AnyMail/GlowDemo.plist">
              install GlowDemo</a>
</li>
</ul>
</body>
</html>
```

主要是两个<a>标签。第一个<a>标签是 Provision Profile 文件的超级链接，因为在 iPhone 上安装应用程序之前，必须先安装 Provision Profile 文件，所以这个链接是有必要的。第二个<a>是.plist 的超级链接。

注意：在第 2 个<a>标签中，href 属性的“itms-services://?action=download-manifest&url=”是固定的（后面的跟着.plist 文件的真实 URL 地址）。“itms-services://?action=download-manifest&url=”不能改成其他内容，否则 Safari 不会调用安装程序。

把这个 HTML 文件放在 Web 服务器上，然后在 iPhone 上用 Safari 访问这个 HTML 文件地址（URL，比如：http://10.180.120.235:8080/install.html），你将看到如图 1-23 所示的页面。

图 1-23　在 iPhone 上访问 install.html

首先点击 Provisioning File 链接，iOS 提示你要安装该预置描述文件，根据提示安装就可。

然后点击 install GlowDemo 链接，Safari 会开始下载.ipa 包，当下载完成会立即启动系统安装程序进行安装。

注意：如果 iPhone 上已经安装有该应用程序，则会进行覆盖安装。

这样，通过 OTA 部署，不再需要用户将 iPhone 和 PC 同步的繁琐步骤（利用 iTunes）。OTA 部署直接利用 iPhone 的无线通信功能（WiFi 网络或移动网络）进行应用程序部署（或者在线更新），显得更加灵活和方便。

第 2 章　iOS 开发框架简介

本章首先介绍苹果操作系统 iOS 的起源、发展及构成，然后对 iOS 开发框架 Cocoa Touch 进行介绍。Cocoa Touch（或 Cocoa）是多个开发框架的集合，由多个层级的子框架构成。最后介绍苹果开发工具包 iOS SDK 及开发环境的搭建。

2.1　苹果 iOS 简介

苹果 iPhone 手机自发布之日以来就给人们带来了全新的感觉和操作体验。一是因为 iPhone 更为优秀的硬件性能，二是因为苹果手机跨时代的操作系统——苹果 iOS。

iOS 即 iPhone OS，是苹果公司针对其 iPhone、iPod Touch 和 iPad 产品开发的基于 UNIX 架构的苹果专属操作系统。原本这个系统名为 iPhone OS，直到 2010 年 6 月 7 日 WWDC 大会上宣布改名为 iOS。iOS 分为 iPhone、iPod Touch 和 iPad 三个版本，但三个版本往往同步更新。

iOS 的发展历史：

- 2007 年 6 月 29 日，苹果发布了 iPhone OS 1.0 固件。
- 2008 年 7 月 11 日苹果发布了 iOS 2.0。
- iOS 3.0 固件在 2009 年 6 月 17 日发布。
- 2010 年 6 月 21 日，苹果发布第四代 iPhone 操作系统——iOS 4.0，在这个版本中，苹果加入了人们期待已久的多任务处理功能。
- 目前最新的 iOS 版本为 2012 年 7 月发布的 iOS 6 版本，这只是 Beta 3 版，正式版预计稍后推出。

虽然 iOS 是一个相对封闭、苹果专属的操作系统，其他品牌厂商的产品无法使用，但其优秀的运行性能和杰出的操作体验，仍然是许多操作系统无法比拟的。作者在同时使用了一段时间的 iPhone 和 Android 手机后发现，无论是系统性能还是稳定性上而言，iOS 4.0 的表现都要远远优于 Android 2.x 系统。

其次，从应用程序的丰富程度上看，iOS 应用也远远超过了 Android 应用。很显然，App Store 能让应用开发商盈利这一点，是 iOS 开发平台比 Android 更胜一筹的主要原因。

可以预料到的是，在未来相当长一段时间内，三大智能手机操作系统中以 iOS 处于领先地位的事实不会改变。iOS 在 iPad 上获得的成功也从侧面印证了这一点。

搭载 iOS 系统的苹果 iPad 获得了巨大的成功，其良好的用户体验和完善丰富的应用软件是成功的重要因素，这一成功与 iOS 系统在 iPhone 智能手机上的积累和完善密不可分。在应用到 iPad 产品之前，iOS 经过三年多的改进，在 iPhone 上已经相当成熟，同时其积累的应用程序和开发经验也可以顺利转移到平板电脑上。另一方面，iPhone 形成的良好口碑和用户经验

也可以顺利转移到 iPad 平板电脑系统上。

RichRelevance 于 2011 年 12 月 25 日的调查数据显示，iOS 设备在移动商务市场上的份额由 4 月份的 88%上升到 12 月份的逾 92%，Chitika Insights 发布的数据显示，至 2012 年 2 月份，iOS 设备的网络流量市场份额甚至超过了苹果自己的 Mac OS。

2.2　iOS 框架介绍

iOS 衍生自 Mac OS X 的成熟内核，但 iOS 操作系统更紧凑和高效，支持 iPhone 和 iPod Touch 的硬件。iOS 继承了 Mac OS X 的风格，包括：统一的 OS X 内核，针对网络的 BSD 套接字，以及 Objective-C 和 C/C++编译器。

iOS 框架分为 Cocoa Touch、Media、Core Service、Core OS 四个层次，如图 2-1 所示。

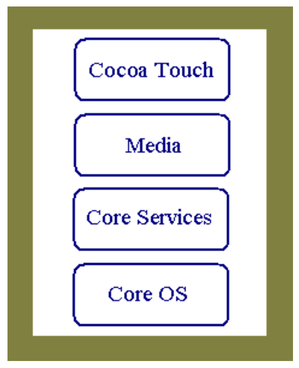


图 2-1　iOS 框架

这 4 个层次从上到下排列，位置越高说明层次越抽象，距离硬件底层越远，其特点如下：

- 层次最高的是 Cocoa Touch 框架，是我们使用得最多的框架，每个 iOS 应用都要使用，其中包括：UIKit 和 Foundation（NS），下一节会更详细地介绍这一层。
- Media 框架是对 iPhone 音频和视频协议的封装，例如，OpenGL ES、EAGL、Quartz、Core Animation、Core Audio、Open Audio Library 和 Media Player。
- Core Services 框架提供了一些核心框架，诸如 Address Book 和 Core Foundation，后者包含了基本的数据类型定义，如数组和集合。
- Core OS 框架包含系统内核级服务，如线程、文件、I/O、内存和网络。

2.3　Cocoa Touch 框架简介

Cocoa Touch 框架是进行 iPhone 应用程序开发工作的主要框架，主要包括 UIKit 和 Foundation（NS）框架，这些库统称为 Cocoa Touch 框架。该框架完全是面向对象的，它是 Cocoa 框架的子集。

注意：Cocoa 框架早先是用于 Mac OS X 上的一个面向对象的应用程序快速开发（Rapid Application Development，RAD）框架，包含了 Foundation 和 App Kit 框架，可用于开发 Mac OS X 系统的应用程序。而随后苹果又在 Cocoa 中加入了对 iOS 的支持，即 UI Kit 框架。习惯上，把 UI Kit 框架、Foundation 框架及一些附属框架合称为 Cocoa Touch 框架，如图 2-2 所示。

图 2-2　Cocoa 框架和 Cocoa Touch 框架

注意，App Kit 用于 Mac OS X。而 UIKit 用于 iOS（它参考了 App Kit 的实现）。Foundation 框架和附属框架则是二者所共有。

Cocoa Touch 是 iOS 上关于用户交互的可编程框架。采用源自 Cocoa 和强大的 Mac 桌面的技术，Cocoa Touch 和 iOS 针对多点触控进行了重新设计。由于其小巧的外形，iPhone 上的按钮、表格表单、页面过渡以及触摸手势都是独特的，而这些界面功能，都可以通过 Cocoa Touch 框架实现。

Cocoa 框架采用“模型-视图-控制器”（MVC）设计模式。“模型”封装应用程序的数据，“视图”显示和编辑数据，“控制器”处理前两者之间的逻辑关系。这种分工负责的方式使得程序易于设计，实现和维护，如图 2-3 所示。

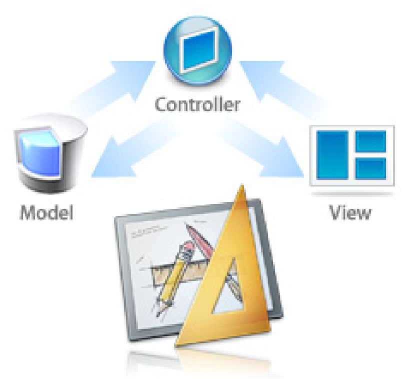


图 2-3　Cocoa 框架中的 MVC 设计模式

2.4 搭建 iOS 开发环境

迄今为止，iOS 只支持在苹果的 Mac OSX 操作系统下进行开发。因此，对于大部分开发者而言，一台基于 Intel 的苹果电脑仍然是必需的——无论是 Mac Book 还是 Mac Mini 都能满足开发的需要。当然，也可以在非苹果的电脑上安装 Mac OS X，正如下面介绍的，借助于硬件虚拟化技术的支持，可以在虚拟机中安装 Mac OS X。

此外，需要下载并安装苹果的 iOS 开发工具包（Software Development Kit，SDK）。这是一个应用程序集合，包括了用于创建 iOS 应用程序所必需的 IDE、API 库及实用工具。

最后，你可能需要在苹果官方网站进行注册。虽然这不是必需的，但如果不这样做的话，你可能无法将你的程序安装到设备上运行。

2.4.1 安装 Mac OS X 操作系统

自从 2007 年年底苹果公司正式发布代号为 Leopard 的 Mac OS X 10.5 开始，一种叫做“Hacked Apple”——把 Mac OS 安装到 PC 上的技术就成为了现实。仅仅在 Leopard 正式上市后的第二天就有高手将其成功破解，使用几个补丁文件便能让 Leopard 安装到普通的电脑上。

由于 Mac OS X 本身对 PC 硬件的支持十分有限，在普通 PC 和笔记本电脑上安装 Hacked Apple 极其不易。尽管网络上存在有各种破解补丁、硬件驱动，甚至破解好的镜像文件，要想在一台非苹果电脑上“啃”一嘴苹果仍然是被戏称为“拼人品”，网上有着无数失败的先例。

有鉴于此，笔者并不建议初学者在非苹果 PC 上安装 Mac OS X 操作系统，与浪费了的无数精力和时间相比，所获得的好处实在不足以称道。如果实在是无法接受苹果电脑的高端价格，那么你可以尝试另一种在 PC 上安装 Mac 系统的方式——在虚拟机中安装——幸好我们还有虚拟机，无论是 VMWare，还是 VirtualBox。

在虚拟机中安装 Mac 拥有以下好处：在 Windows 系统和 Mac 系统间切换不需要重启；在虚拟机中安装避免了硬件驱动不支持的问题，因为不需要安装硬件驱动程序；使用虚拟机安装有更高的成功率。

以下以笔者的华硕 X42J 笔记本为例，演示如何在 VirtualBox 中安装 Mac Snow Leopard OS X 10.6.5（支持 i3/i5/i7）。

1. 推荐硬件配置

原则上，CPU 必须支持 SSE2/SSE3 和硬件虚拟技术。如果不能确定 CPU 是否支持硬件虚拟，可以运行 SecurAble 进行测试，出现如图 2-4 所示的对话框即为支持。

以下列出笔者的笔记本硬件配置，以供参考：

- ❑ 电脑型号——华硕 K42JE 笔记本电脑
- ❑ 处理器——英特尔 Core i3 M350 @ 2.27GHz 笔记本处理器
- ❑ 主板——华硕 K42JE（英特尔 HM55 芯片组）
- ❑ 内存——2GB（海力士 DDR3 1333MHz）

图 2-4 用 SecurAble 进行测试

- 主硬盘——希捷 ST9320423AS（320 GB / 7200 转/分）
- 显卡——ATI Mobility Radeon HD 5470（512 MB）
- 光驱——日立-LG DVDRAM GT32N DVD 刻录机
- 声卡——瑞昱 ALC269 @ 英特尔 5 Series/3400 Series Chipset
- 网卡——智微 JMC25X PCI Express Gigabit Ethernet Adapter

2. 准备使用的工具

虚拟机 Virtual Box 的下载地址：http://u.115.com/file/t54cd05734。

破解版的 Mac OS X，iAntares OSx86 10.6.5 v3.2 繁简英整合版（2010 年 12 月 12 日更新），下载地址：http://www.ed2000.com/ShowFile.asp?FileID=255645。

3. 安装过程

打开 Virtual Box，点击工具栏上的“新建”按钮，弹出“新建虚拟电脑”向导，选择操作系统类型为 Mac OS X 及 Mac OS X Server，并为虚拟机设置一个名称（比如 Snow Leopard），如图 2-5 所示。

图 2-5 “新建虚拟电脑”向导

点击“下一步”按钮，设置虚拟机使用的物理内存，请至少选择 1GB（如图 2-6 所示）。

图 2-6　分配虚拟机使用的内存大小

在选择虚拟磁盘时，选择“创建新的虚拟硬盘”。为了取得更好的性能，虚拟硬盘类型选择“固定大小”（如图 2-7 所示）。

图 2-7　设置虚拟硬盘类型

虚拟硬盘容量至少设定为 30GB，并保证文件存放位置的可用空间是足够的（如图 2-8 所示）。

图 2-8　设定虚拟硬盘空间大小

点击“下一步”按钮，直至安装结束。

选择刚才创建的虚拟机 Snow Leopard，点击工具栏中的“设置”按钮，在弹出的虚拟机设置窗口左侧面板中选中“系统”，“启动顺序”选择“光驱、硬盘”，然后取消“启用 EFI”选项，如图 2-9 所示。

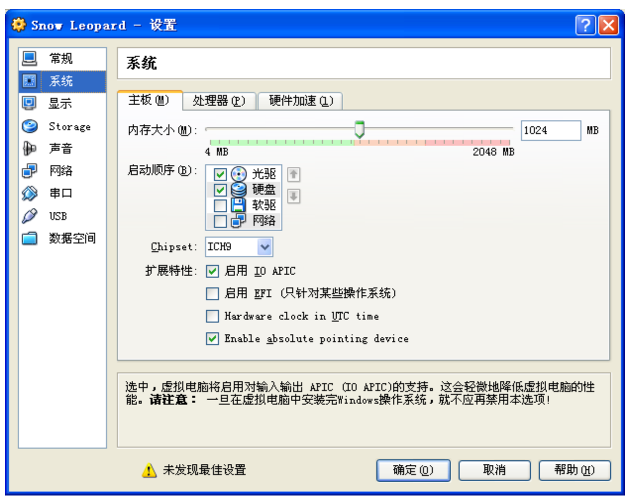


图 2-9　主板设置

选择左面板中“显示”项，将“显存大小”调为最大，然后选择“启动 3D 加速”选项（如图 2-10 所示）。

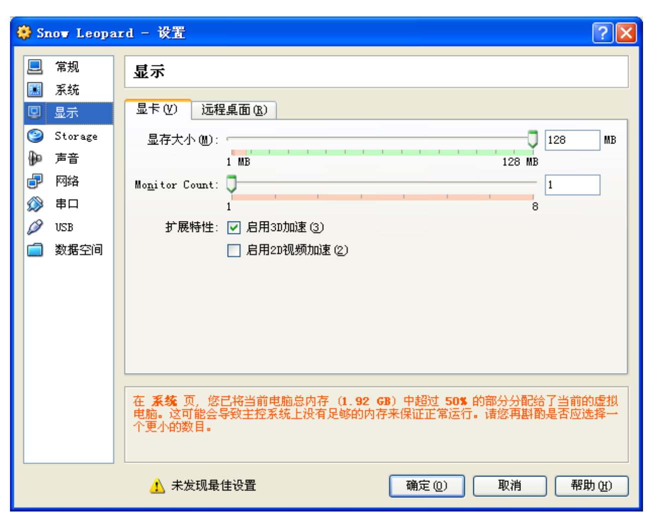


图 2-10　显卡设置

好的，虚拟机配置完成了，下面的步骤是安装 Snow Leopard。

在虚拟机设置窗口中，选择 Storage，在 IDE 控制器中添加一个虚拟光驱，然后为这个虚拟光驱添加一个盘片，把 iAntares OSx86 10.6.5 v3.2 的 iso 文件镜像加载进去（如图 2-11 所示）。

关闭设置窗口，双击虚拟机 Snow Leopard 启动虚拟机。如果顺利，虚拟机会用 iAntares_v3.iso 进行引导，并进入 Snow Leopard 的安装界面。选取中文作为安装语言，然后从菜单“实用工具”中打开“磁盘工具”。

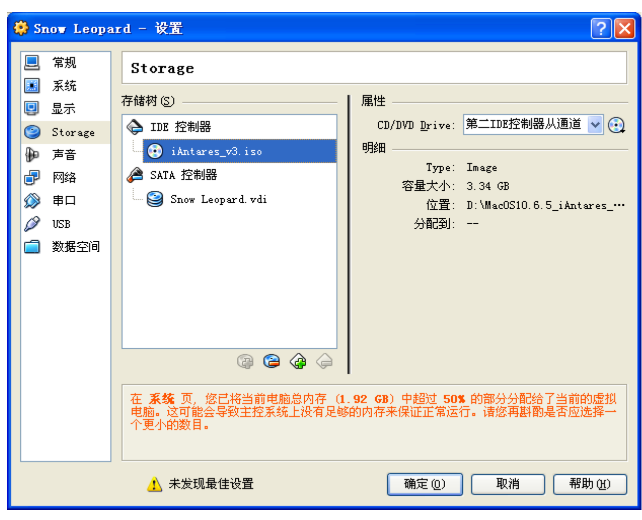


图 2-11　加载光盘镜像

在磁盘工具点击标签栏的“抹掉”，对磁盘进行格式化。文件系统格式为 Mac OS 扩展（日志式），然后点击按钮“抹掉”按钮（如图 2-12 所示）。

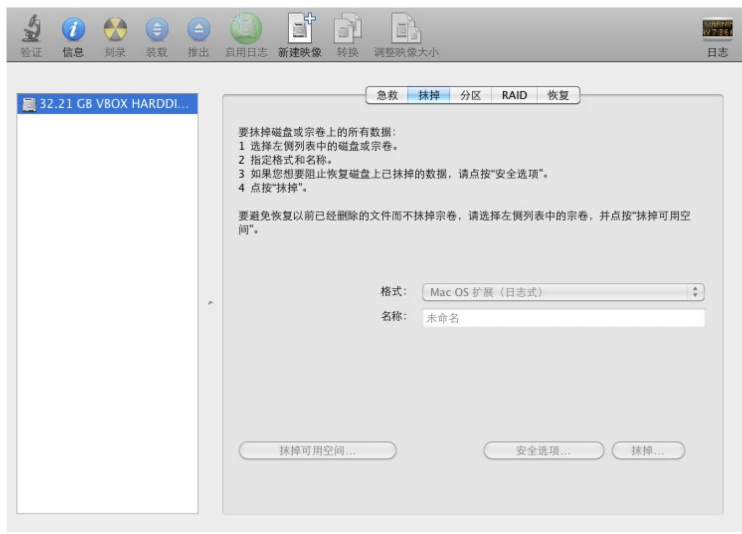


图 2-12　格式化磁盘

格式化完成后，选择格式化的磁盘作为安装目标，同时点击“自定”按钮。

在接下来的自定义安装界面中，“启动选项”除了后面 3 项以外全部选中，硬件驱动全部不需要选（虚拟机已经带硬件驱动），引导器选择变色龙 RC4 r684 而不是 RC5 r653，其余选项随意设置或保持默认值（如图 2-13 所示）。

这个步骤是整个安装中最重要的步骤，也许需要尝试很多次才知道最适合机器的设置。这个过程中需要不断地修改启动选项并重启，甚至可能会出现几次蓝屏。但在虚拟机中安装的好处就在于，除了出现蓝屏以外，都不需要按电源或 Reset 键，虚拟机重启的速度比硬启动要快许多。

这个步骤完成后就是缓慢的安装进度了，这需要一些时间，请耐心等候。

安装完成后，可能会出现“安装失败”的提示，不必惊慌，重启虚拟机后，会发现虚拟机引导菜单上多了一个 snow leopard 的引导选项，这个就是我们安装成功的 Mac OS 操作系统，另一个是安装光盘（如图 2-14 所示）。

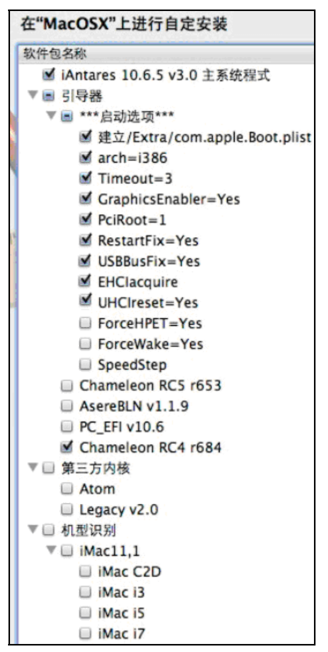


图 2-13 设置自定义选项

图 2-14 选择从新安装的 Mac OS 系统引导

用方向键把光标移动到 snow leopard 上，回车，变色龙开始从 Mac OS 进行引导。

启动后进入 Snow Leopard 桌面。由于某些 Bug，在这个桌面工具栏上会有 3 个图标显示为问号（如图 2-15 所示），当然如果为了美观，完全可以删除它们。

需要注意的是，如果 Mac 提示安装版本更新，请不要轻易更新系统，否则你可能进不了系统。因为破解的 Mac OS X 系统对系统内核进行了修改，如果升级的话有可能导致系统文件再次被覆盖，导致系统无法正常引导。

接下来需要下载 iOS SDK，并在 Mac 下进行安装。

图 2-15　Snow Leopard 桌面

2.4.2　下载安装 SDK

每一个在苹果网站上注册了 iPhone 开发人员计划的程序员，都可以登录到以下地址下载最新版本的 iOS SDK：

https://developer.apple.com/devcenter/ios/login.action

这是一个几个 GB（根据版本不同）的 Mac 安装镜像文件，里面包括如下内容：

- Xcode 集成在 SDK 中一起发布，它支持苹果的 Objective-C 语言，也支持 C 和 C++代码。我们将在第 4 章介绍它的使用。
- Interface Builder 用于创建程序的 GUI，它和 Xcode 集成在一起，也可以单独启动。在本书很多地方仍然使用了它，第 5 章将对 Interface Builder 进行介绍。
- iPhone 模拟器可以在 Mac 中调试 iOS 应用程序，它的外观和真实的 iPhone/iPad 设备一模一样。使用它调试程序，比在真实设备中更方便快捷。在后面的章节中，会大量使用这个工具调试程序。
- Dashcode 也是/Developer/Applications 中的一部分，它是用于创建 Web 应用的优秀、极为精巧的图形开发环境，本书中不会使用到它。

双击下载后的文件，把 SDK 安装到 Mac 上。

接下来，创建我们的第一个 iOS 应用程序，以此检验我们的开发环境已配置成功。

2.5　写一个 iPhone 程序

点击桌面上的 Xcode 图标，启动 Xcode。选择菜单“File→New Project”，显示新建项目模

板向导（如图 2-16 所示）。

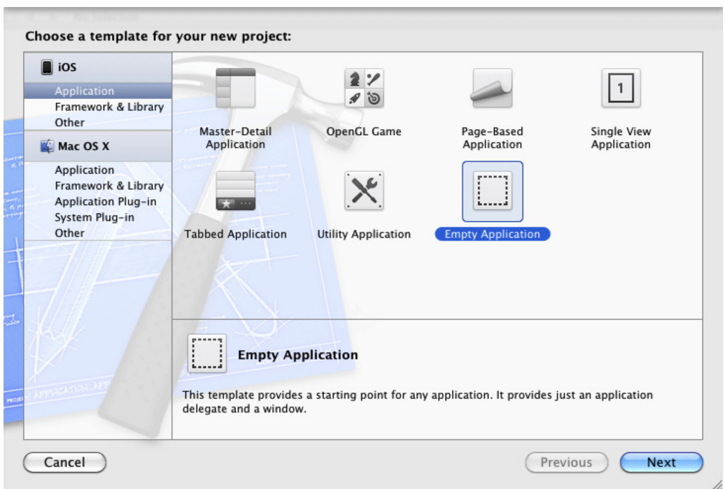


图 2-16　新建项目向导

在左边栏中列出了 Xcode 支持的两种项目类型：iOS 和 Mac OS X 项目，选择 iOS 下方的 Application，然后选择 Empty Application 类型的项目。点击 Next 按钮，进入新项目设置界面，如图 2-17 所示。

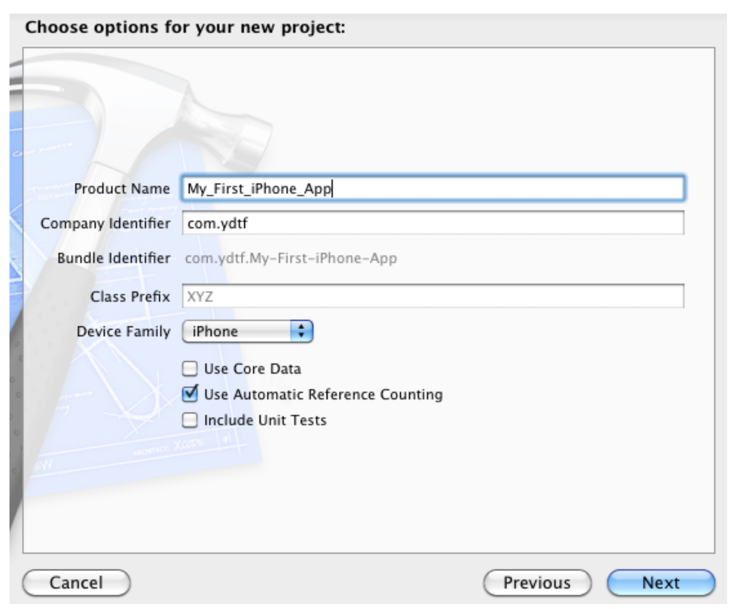


图 2-17　新项目设置界面

在新项目设置界面中，在 Product Name 栏填写项目名称，比如 My_First_iPhone_App。在

Company Identifier 栏，填写公司名前缀，比如 com.ydtf。在 Device Family 栏填写所开发目标平台，比如 iPhone（Universal 则表示 iPhone/iPad "二合一"版本）。然后点击 Next 按钮。

接下来是指定项目保存路径界面，如图 2-18 所示。

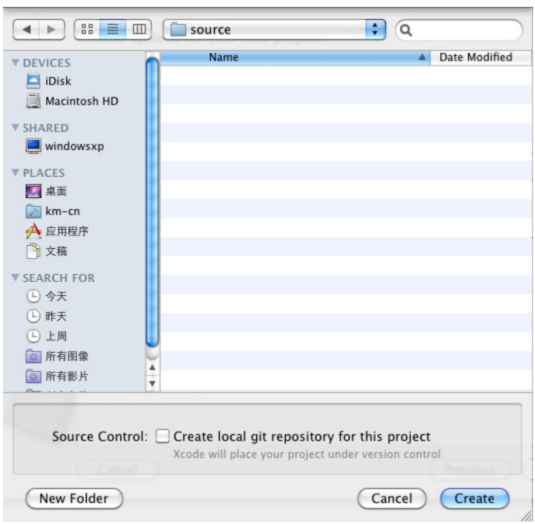


图 2-18　项目保存路径对话框

选择一个合适的项目保存路径，然后点击 Create 按钮。

这样，一个 iPhone 应用程序就创建好了。如图 2-19 所示是 My_First_iPhone_App 项目的项目编辑界面，由于图太大，这里只显示了窗口的一部分。

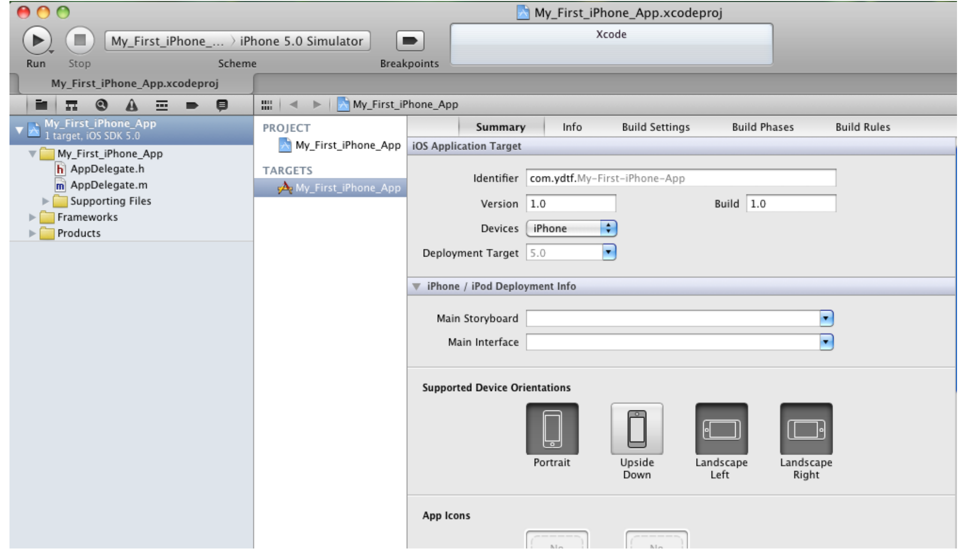


图 2-19　项目编辑窗口

界面的左侧是 Project Navigator 窗口，它列出了项目的所有资源，包括源文件、.xib、.plist、框架/库、二进制和图片等。右侧是指定资源（文件）的 Info 窗口或编辑窗口，我们主要的编辑工作都在这里完成。

提示：如果你看不到 Project Navigator 窗口，可以通过菜单“View→Navigators→Show Project Navigator”来重现它。

在 Project Navigator 中选择 My_First_iPhone_App 文件夹，单击右键，选择“New File”菜单，弹出新建文件模板向导，如图 2-20 所示。

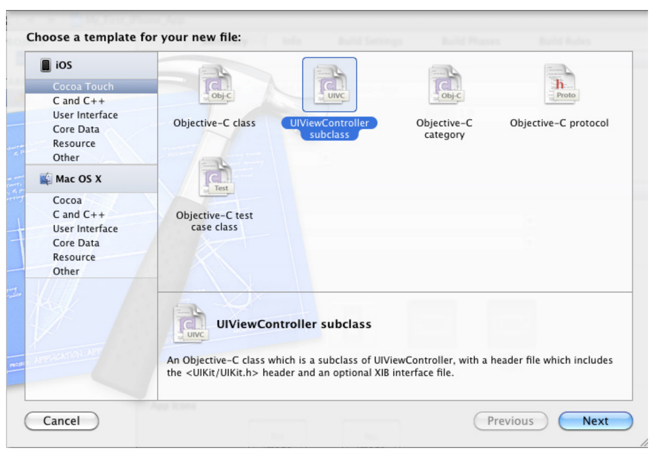


图 2-20　新建文件模板选择向导

Xcode 4.2 能创建各种各样的文件。我们选择 iOS/Cocoa Touch 下的“UIViewController subclass”，然后点击 Next 按钮，将弹出如图 2-21 所示的新文件设置向导窗口。

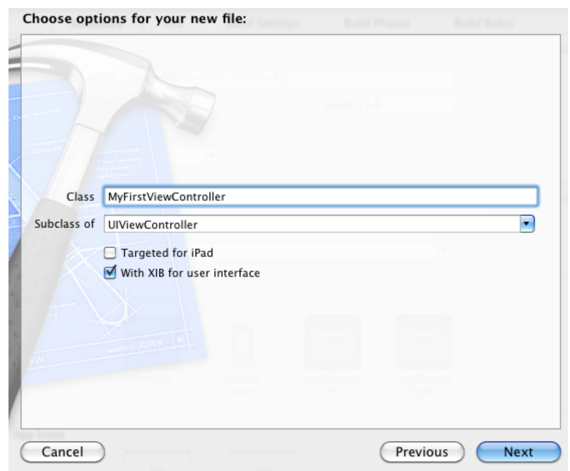


图 2-21　新文件设置向导

在该窗口中，输入类的名称，如 MyFirstViewController。勾选“With XIB for user interface”选项，点击 Next 按钮，进入文件保存路径窗口，如图 2-22 所示。

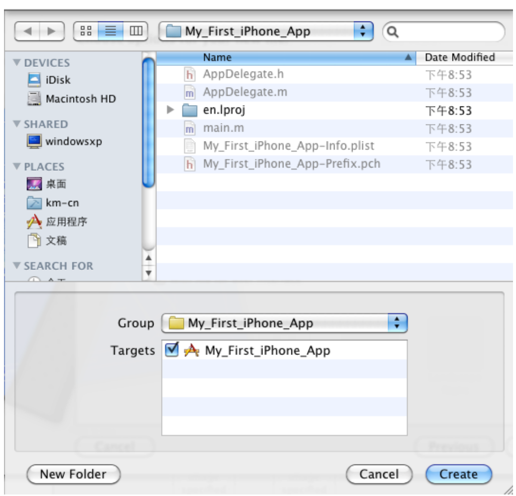


图 2-22　新文件保存路径界面

点击 Create 按钮。默认情况下，将转入 MyFirstViewController.xib 文件的编辑界面（即 Interface Builder 界面），如图 2-23 所示。

提示：与 Xcode 3.2 不同，在 Xcode 4.2 中，Interface Builder 是真正集成在 Xcode 的 IDE 中，而不再单独存在。

此时，在 Interface Builder 的右侧（用于全屏太大，图 2-23 不能显示出来），可以找到如图 2-24 所示的 Object Library 窗口。

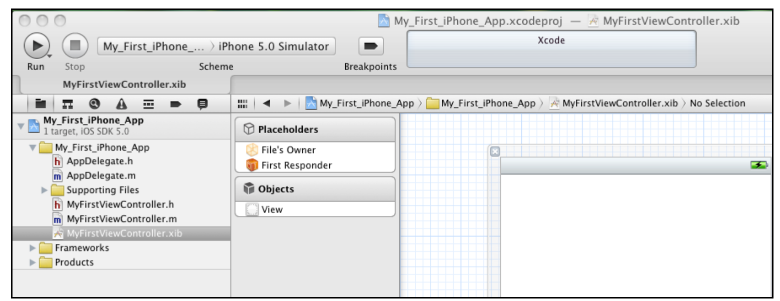


图 2-23　Xcode 中集成的 Interface Builder 界面　　图 2-24　Object Library 窗口

我们从 Object Library 窗口中找到一个 Label 对象（就在 Object Library 窗口的第 1 行），然后按住它不放，直接把它拖放到 MyFirstViewController 的编辑窗口中（Interface Builder 中），结果如图 2-25 所示。

图 2-25 在 MyFistViewController 中加入一个 Label

然后双击图 2-25 中的 Label 对象，将它的文本修改为“嗨，这是我的第 1 个 iPhone App!”，如图 2-26 所示。

图 2-26 修改 Label 的显示文本

你可以任意拖动标签控件改变它在窗口中的位置。保存在 Interface Builder 中所做的更改（快捷键 ⌘ + S）。

提示：对于 Windows 键盘，win 键对应苹果键盘中的苹果键⌘。

在 Project Navigator 窗口中找到源文件 AppDelegate.m，选中它，我们将对其进行一些编码工作。在 AppDelegate.m 的编辑窗口的顶部#import “AppDelegate.h” 一行后换行，增加以下代码：

```
#import "MyFirstViewController"
```

找到方法：

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
        (NSDictionary *)launchOptions
```

在其中 “self.window.backgroundColor = [UIColor whiteColor];” 一行后增加以下两行代码：

```
MyFirstViewController* vc=[[MyFirstViewController alloc]init];
self.window.rootViewController=vc;
```

接下来要运行这个程序，看看它最终实现的效果。

2.6　在模拟器上运行应用程序

在 Xcode 4.2 的顶部工具栏中，找到 Scheme 按钮。从该按钮的下拉菜单中选择“iPhone 5.0 Simulator”，如图 2-27 所示。

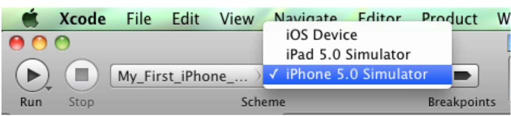


图 2-27　修改项目的 Scheme

提示：如果 Xcode 4.2 的工具栏未显示，请选择菜单“view→Show Toolbar”。

然后点击工具栏中的 Run 按钮（快捷键⌘+R），会启动 iPhone 模拟器界面，并通过 iPhone 模拟器来运行我们的 My_First_iPhone_App 应用程序（如图 2-28 所示）。

图 2-28　在模拟器中运行应用程序

2.7　在 iPhone 上运行应用程序

如果要在 iPhone 手机上运行程序则没有那么容易了。

正如第 1 章所述，在开始开发 iPhone 应用程序之前，你需要注册成为 iPhone 开发人员。只有这样，苹果公司才会允许你使用“完全的”的 SDK，否则你只能下载一个有功能限制的免费 SDK。

注册页面位于 http://developer.apple.com/iphone。苹果将该注册程序称为苹果开发者计划（Apple Developer Plan），其中针对 iPhone 开发人员的称作 iOS 开发者程序。在该页面的底部提供了苹果支持的所有注册程序（见图 2-29）。

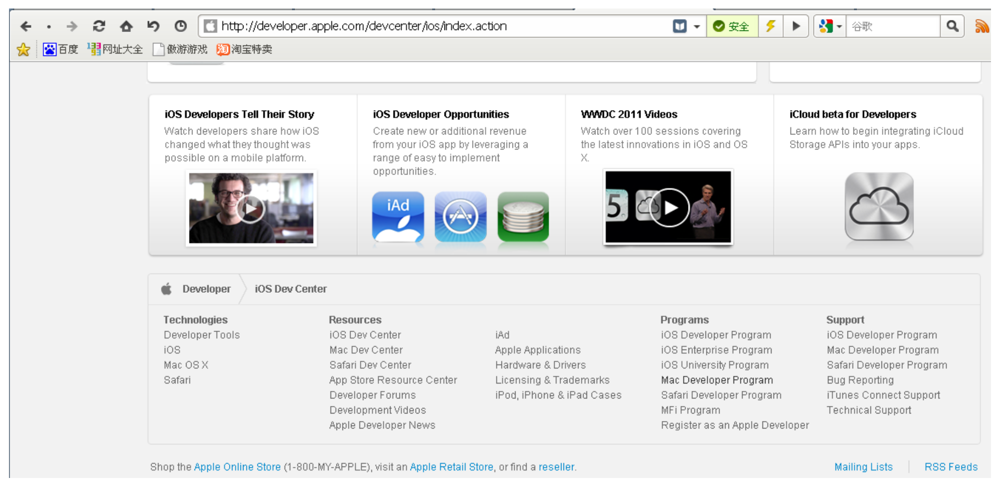


图 2-29　苹果开发者计划

在该页面底部的 Programs 列表中，列出了所有的 iOS 开发者程序类型（排在列表的头 3 项）：

- iOS Developer Program
- iOS Enterprise Program
- iOS University Program

iOS University Program 程序是免费的，面向科研和教学人员。它也提供了完整的 Xcode 和 iPhone 模拟器，但不支持将应用程序在真实的 iPhone（iPod Touch 或 iPad）中运行，而且也不支持通过 App Store 发布应用程序。

iOS Developer Program 程序是开发者们最常用的版本，即标准版 IDP，它的价格是 99 美元/年。它提供了一个 Xcode，一个 iPhone 模拟器——支持在 Mac 上运行绝大多数 iPhone 程序。标准版 IDP 支持通过苹果 App Store 分发应用程序，并可在 iPhone 上（不是模拟器上）调试应用程序。

iOS Enterprise Program 程序即企业版 IDP，在第 1 章中已经做了详细的介绍，它的价格是 299 美元/年。

当你拥有了标准版或企业版的 IDP 证书，还需要通过 Provision Portal 制作相应的 Provisioning Profile，并下载到你的电脑上。然后使用 IDP 对程序进行代码签名，才能在 iPhone 上运行你开发的程序，这个过程正如第 1 章所述。如果你已经仔细看完第 1 章，那么我们可以假设你已经完成了这些步骤。接下来就可以在真机上运行（调试）程序了。

将 iPhone 连接到电脑， Xcode 会自动识别出你的 iPhone。将项目的 Scheme 修改为你的 iPhone 名字，例如作者的 iPhone 名为“YHY's iPhone”，如图 2-30 所示。然后点击 Run 按钮，Xcode 将会在你 iPhone 上运行 My_First_iPhone_App 程序了。

图 2-30　在设备上运行程序

提示：如果是第一次调试这个设备，则 Xcode 会提示要在该设备上安装一个 Provisioning Profile，请选择同意安装。

第 3 章　Objective-C 语法简介

本书不是一本关于 Objective-C 编程语言的专著，但仍然会介绍一些 Objective-C 语言的语法基础和有趣特性。这对于刚刚接触到 iPhone 编程的人来说，会是一个很好的开始。

Objective-C 兼具 C 语言和面向对象语言的特性。因此本章会从两个方面来介绍 Objective-C（简称 O-C）。

首先，由于 Objective-C 具有 C 语言背景（它仍然是一种 C），它从 C 语言中继承了一些 C 语言特性，例如：Objective-C 全面支持 C 的数据类型（包括简单类型和复合类型）、常量/变量/宏、包含头文件（import/include 指令）、函数、条件和循环控制语句。

其次，Objective-C 在 C 语言的基础上进行了扩展，加入了面向对象的内容。例如：对象和类（方法及属性的集合）、Objective-C 的消息机制、内存管理（包括对象生命周期和对象的创建和释放）、类别和协议、反射、谓词，以及 Cocoa 对一些常见模式（Pattern）的 Objective-C 实现（如 MVC、KVO）。

在本书后续的一些章节中，使用了块语法。因此本章最后单独对 Objective-C 的块编程进行了介绍。 块是现代 C 语言标准的一部分，类似 C++中的 inline 函数。Objective-C 以面向对象的方式对 C 的块进行了封装。

你可以有针对性地阅读这些内容。比如对于有 C 基础的程序员，应该跳过基本语法部分（主要是 C 语言特性），重点阅读 Objective-C 在 C 语言基础上进行的扩展（如 NS 类、类别和协议），以及 Cocoa 提供的一些优秀特性（比如 MVC、KVO）。

3.1　Objective-C 的 C 语言特性

Objective-C 源自 C，它是 C 语言特性和 Smalltalk 语法的集合。从 20 世纪 80 年代开始，Objective-C 对 C 语言进行了大量的扩展，直至 30 年后的今天，Objective-C 已经发展成为当下最流行的编程语言之一。

Objective-C 全面支持 C99 标准。对于 C 这种程序员早已熟知熟悉的经典语言，作者在此并不准备多做介绍，你可以阅读大学计算机课程中使用的 C 语言教程，或者 Dave Mark 编写的经典专著《Learn C on the Mac》。本章只简单介绍 Objective-C 中一些主要的 C 语言特性，比如 Objective-C 支持的 C 语言数据类型、常量/变量和函数、条件和循环等，以及 Objective-C 的一些非 C 语言特性（面向对象特性）。

3.1.1　一个简单的 Hello World

了解一种新语言是从一个最简单的程序开始，Hello World 程序无疑是最简单的，让我们

从它开始。

打开 Xcode，使用菜单“New Project”打开新建项目向导，在项目模板中选择“Mac OS X → Application → Command Line Utility → Foundation Tool”，并可以将项目命名为 HelloWorld。为了能更清晰地了解 Objective-C 这门语言，我们特意选择了最简单的命令行项目，这种类型的项目并不能在 iPhone 上运行，它是在 Mac 命令终端环境中运行的。要知道，原本 Objective-C 就是用来设计 Mac 程序的，后来才扩展使用到 iPhone 程序。因此，我们避免了使用 Cocoa 框架带来的一些复杂性。

Xcode 为我们生成了一系列文件，打开其中 HelloWorld.m，查看 Xcode 自动产生的代码如下所示：

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[]) {
    pool = [[NSAutoreleasePool alloc] init];
    //插入代码
    NSLog(@"Hello, World!");
    [pool drain];
    return 0;
}
```

整个代码包括注释不超过 10 行。然而对于从未接触过这门语言的人来说，可以从中学习到许多东西。

3.1.2　Objetive-C 是另一种 C

C 语言作为传统的计算机教学语言，对许多接受过大学理工科教育的人来说，并不陌生。从上面的代码中，我们看到了许多 C 语言的特征，比如语句用分号结束，程序开头同样需要导入头文件，还有一模一样的 main 函数作为程序的入口，等等。实际上，Objective-C 完全兼容标准 C，它同样采用 gcc 编译器，可以在 Objective-C 程序中随意嵌入 C 代码。对于许多推崇标准 C 的 iPhone 程序员来说，他们更倾向于使用 C 代码而不是 Objective-C 代码来写 iOS 应用程序。

因此，虽然 Hello World 是一个 Objective-C 程序，但是我们却可以把它变成 C 程序，将代码修改为：

```
#include <stdio.h>
int main (int argc, const char * argv[]) {
   printf("Hello World");
   return 0;
}
```

编译运行程序，黑黑的控制台窗口输出了同样的文字内容。两段代码的区别仅在于：HelloWorld 的 C 语言版没有使用任何框架，而 Objective-C 版本则使用了 Foundation 框架。

同 C 一样，Objective-C 源代码也分成了两部分：接口和实现。Objective-C 的接口文件采

用.h 作为后缀名；实现文件的后缀则采用.m，而不是.c（C）或者.cc（C++）。

3.1.3 数据类型

在第 2 章的 Cocoa 框架介绍中，提到过 Foundation 框架是 Cocoa 框架的基本框架之一（见图 2-2 Cocoa 框架）。而 Foundation 框架中就定义了数量众多的数据类型和底层类，例如 NSString、NSArray、NSEnumerator 和 NSNumber。我们不可能全部介绍它们，整个 Foundation 框架定义了上百个类。但我们会讨论其中最有用的一些。

前面讨论的 HelloWorld 程序是很重要的，起码对于这一章来说。因为我们后面学习的代码都会基于这个 main 函数，只需在这个函数中加入少量的代码，程序就可以运行。

1. C 基本数据类型

由于 Objective-C 和 C 兼容，C 语言中的 5 种基本类型：char、int、float、double、void 在 Objective-C 中仍然可用，并且这些基本类型可以使用的修饰符 unsigned、signed、short、long 进行修饰。

2. C 复合类型

C 语言中的复合类型：数组、指针、结构、联合、枚举仍然得到了支持。此外，Objective-C 中特有的几种复合类型，就是 id、SEL、nil 和 NULL。

id 是指向任意对象的指针，类似 C 语言中的 void*。这里的任意对象是指 Objective-C 中任何继承自 NSObject 类的实例。在 Cocoa 中，NSObject 是所有类的根类。所以，可以用 id 指向任何类型的 Objective-C 对象。

选择器 SEL 实际上被定义为 const char*，在 Objective-C 中，用它来指向任何方法的定义，等同于 C 语言中的函数指针。要创建 SEL 类型有两种方式，一种是使用@selector()关键字，并在括号中传递一个方法签名作为参数，通过这种方式你可以调用一个 Objective-C 对象的指定方法：

```
[object performSelector:@selecotr(doSomething)];
```

另一种就是通过 NSSelectorFromString()函数，把方法签名以字符串的方式作为参数传递，这种方式类似于 Objective-C 的反射（关于选择器，后面还会介绍）。

nil 和 NULL 都代表了空指针。nil 用于 Objective-C 对象，而 NULL 则用于指针类型，并且二者不可互换。任何 Objective-C 对象都是 NSObject 类的子类，它们都可以用 id 来指向，因此可以使用 nil 指针来初始化为空。而 C 指针类型指向的数据是结构体，而不是对象，因此不能用 nil 指针初始化为空，但可以用 NULL 指针初始化为空。此外，NSNull 用于指向集合对象，表示集合为空，没有任何有效的对象。

3. 布尔类型

由于 C 没有布尔类型，你可以使用 C++中的 BOOL 类型。但 Objective-C 提供了自己的布尔类型 BOOL，它提供了两个表示真和假的名义值：YES 和 NO，类似于 True 和 False。让人

奇怪的是，它虽然属于 Cocoa，却没有使用 NS 作为前缀（如后所述）。

然而使用 BOOL 类型时要小心，否则可能会导致出人意料的事情发生。

因为 BOOL 的定义其实是带符号字符，占 1 个字节的存储空间。我们实际上可以把一个字节的数据放到 BOOL 中去，如果把一个超过 1 字节的整数（如 short 或 int）放进去，那么只有低位字节会被放进 BOOL 中去。在其他语言中，0 定义为假，非 0 定义为真。而在 Objective-C 中不同，1 定义为 YES，0 定义为 NO。0 和 1 以外的数据即不是 YES，也不是 NO。例如，我们习惯于这样的语句：

```
if(isNotEquals(a,b)==YES){// 如果 a,b 不相等
    ...
    }else{// 如果 a,b 相等
    ...
}
```

isNotEquals (a,b)是一个函数，无论它的返回结果是什么，在 if 的条件表达式中，它首先会被转换为 BOOL 类型。但是即使 isNotEquals (a,b)函数真的被定义为 BOOL 类型，它也不一定只返回 YES 和 NO（1 和 0）。原因如前面所述，如果在 isNotEquals 函数中返回其他类型，比如 int，仍然可以正常转换为 BOOL，Objective-C 并不会检查出错误。实际上，在 isNotEquals 函数中很可能是这样返回的：

```
return a-b;
```

如果 a 和 b 相等，那么返回的实际是 NO，即 0。如果 a 为 4，b 为 3，则返回的是 1，即 YES。这都没有什么问题。但如果 a 为 5，b 为 2，那么实际上返回的这个布尔值即不是 YES，也不是 NO，而是整型 3。那么接下来的 if 语句中，3 自然不等于 YES（YES=1），你会发现程序的结果是 a 与 b 相等。问题是，你觉得 5 和 2 应该相等吗？

解决问题的方式有两种，要么你要确保每一个 BOOL 值返回的除了 YES 和 NO 外不能有其他值，比如上面的返回语句应该修改为：

```
Return a-b!=0;
```

要么，在条件表达式中，永远只和 NO 进行比较，而不要和 YES 进行比较，比如：

```
If(isNotEquals(a,b)==NO){// 如果 a,b 相等
    ...
    }else{// 如果 a,b 不相等
    ...
}
```

喜欢使用哪种方式，完全由你个人的喜好来定。

4．NS 结构体

Cocoa 的 Foundation 框架中定义了大量的结构体，如 NSRange、NSPoint、NSSize 等，这些结构体统一用 NS 作为前缀，在后面的章节中你还会见到一些其他的 NS 结构体。

下面介绍一些有用的 NS 结构体。

（1）NSRange

NSRange 的定义是：

```
typedef struct _NSRange{
        unsigned int location;
        unsigned int length;
}NSRange;
```

它有两个有用的成员，location 表示范围从哪个数字开始，默认为 0，length 表示范围的大小，默认值是 NSNotFound，表示不存在的范围。比如：

```
NSRange range={3,6};
```

这个语句用 C 语言的风格创建了一个 NSRange 变量，这个 NSRange 表示了从 3 开始、总长度为 6 的数值范围。

或者用给结构成员赋值的方式初始化 NSRange：

```
NSRange range;
range.location=3;
range.length=6;
```

但是 Cocoa 还为所有的结构提供了便利函数 NSMakeXXX：

```
NSRange range=NSMakeRange(3,6);
```

可以很方便地创建一个 NSRange。

（2）NSPoint、NSSize 和 NSRect

结构体可能是最方便表述几何图形的数据类型了，比如 NSPoint、NSSize 和 NSRect。这 3 个结构体构成了一个二维平面中最基本的数据类型。

NSPoint 表示物体在二维坐标中的 x 位置和 y 位置：

```
typedef struct _NSPoint{
float x;
float y;
}NSPoint;
```

NSSize 用于表示一个形状的大小，任何占据一定二维空间的形状，总是离不开宽和高这两个属性：

```
typedef struct _NSSize{
       float width;
       float height;
}NSSize;
```

NSRect 用于表示一个矩形，它由矩形的坐标位置和大小共同构成，很显然位置可以用 NSPoint 表示，而大小就用 NSSize 表示：

```
typedef struct _NSRect{
        NSPoint origin;
        NSSize size;
}NSRect;
```

5. NS 类

既然 Objective-C 是“面向对象的 C”，那么除了对 C 的基本数据类型进行全面支持外，当然也对这些类型进行了面向对象的封装和扩展。Cocoa 框架把所有的类都加了“NS”前缀——来自于 Cocoa 前身 NextSTEP 的缩写，以显示和 Core Foundation 类的区别（以 CF 为前缀），比如 NSString、NSNumber、NSArray、NSDictionary。本书中会大量使用这些 NS 类。

与 C 语言寥寥数种的基本数据类型不同，NS 类家族可谓种类繁多，本书不可能逐一讨论。在此，我们将只介绍几种最常用的 NS 类。

（1）NSString 和 NSMutableString

在 HelloWorld 程序中有一个打印语句：

```
NSLog(@"Hello, World!");
```

其中@"Hello, World!"就是一个字符串，即 NSString。注意，使用@和双引号来括住字符串，而不是像 C++一样只使用双引号来括住字符串，这样就是一个 NSString 类型的字符串常量。

NSString 有许多封装的特性，比如统计字符串的长度，将自身与其他字符串进行比较，方便地将自身转换为整数或浮点数。

出于性能上的考虑，你不能改变 NSString 的值，比如在 NSString 对象中临时插入其他字符或者删除某些内容，这跟 Java 语言中的 String 类型是一样的。当然当你确实需要这样做的时候，可以使用 NSString 的可变版本，NSMutableString。后者和前者几乎一模一样，除了可以对其储存的字符串进行改变操作。

创建一个 NSMutableString 可以用以下代码：

```
NSMutableString *mutableStr=[NSMutableString stringWithString:@”Following me”];
```

现在，我们可以任意替换其中的字符内容：

```
NSRange range={10,2};
[mutableStr replaceCharactersInRange:range withString:@”you”];
```

于是，这个字符串的内容变成了“Following you”。如果想把这个字符串中的“me”删除，可以用 NSString 的 deleteCharatersInRange 方法：

```
[ mutableStr deleteCharactersInRange:range];
```

当然 NSMutableString 类最常用的方法还是 appendString 方法，在字符串的末尾加上另一个字符串：

```
-(void) appendString:(NSString*)nsstring;
```

（2）NSNumber

NSNumber 用于处理所有与数字相关的数据类型，从整型到浮点型的数据，也包括前面提到的 BOOL 类型。我们可以通过一系列的方法把基本数据类型包装为 NSNumber:

```
+(NSNumber*) numberWithChar:(char)value;
+(NSNumber*) numberWithInt:(int)value;
+(NSNumber*) numberWithFloat:(float)value;
+(NSNumber*) numberWithBool:(BOOL)value;
```

类似的方法还有很多，包括各种 unsigned 和 long 型数据的版本，我们不能一一列举。创建了 NSNumber 对象之后，我们就可以把它存储到 NSArray 和 NSDictionary 集合中去，而此前这两种集合对象是无法存放基本类型数据的。当我们从集合中把 NSNumber 取出来后，又可以通过以下方法将其还原为原来的基本类型：

```
-(char)charValue;
-(int)intValue;
-(float)floatValue;
-(BOOL)boolValue;
...
```

很显然，这些方法是和创建 NSNumber 的方法一一对应的。

（3）NSArray 和 NSMutableArray

NSArray 与 C 语言中的数组是类似的，但是它还封装了很多 C 数组不具备的特性。比如，它不需要像数组一样，随时随地都需要程序员对数组下标越界进行检查和判断。它其实更像 Java 语言中的 ArrayList。

但是正如前面提到过的，NSArray 只能存储 Objective-C 对象，而不能存放基本数据类型，如 int、float、char、struct、enum 等。

提示：NSDictionary 也有同样的限制。

此外，NSArray 也不能存放空值 nil，因为 nil 用作 NSArray 的结束符。NSArray 的最后一个元素永远是 nil 对象。

NSArray 有一个便利的创建方法，可以一次性创建一个包括了许多元素的 NSArray 对象，如下所示：

```
NSArray* array=[NSArray arrayWithObjects:@"one",@"two",@"three",nil];
```

这个方法提供了数目不定的参数作为其初始的创建时就包含的元素。牢记前面的两个限制：1）其元素只能是对象；2）最终要以一个 nil 结束。

要获得 NSArray 的元素数目很容易，访问其 count 属性即可。

使用以下方法访问指定索引的对象：

```
-(id)objectAtIndex:(unsigned int)index;
```

如果索引大于数组元素中的个数，Cocoa 会抛出 NSRangeException 异常。

然而，NSArray 是不可变的对象数组，一旦创建就不可能增加和删除其中包含的对象——当然对象自身还是可以改变的。但是数组的元素个数不会改变，其中的元素也不能替换。

如果你需要一个在创建后仍然可以任意增加和删除元素的数组，可以使用 NSArray 的可变版本 NSMutableArray。

通过类方法 arrayWithCapacity 和 arrayWithArray 来创建一个可变数组：

```
+(id)arrayWithCapacity:(unsigned)numItems;
+(id)arrayWithArray:(NSArray*)array;
...
```

创建 NSMutableArray 的类方法有很多，但最常见的是这两个。创建之后就可以用 addObject 方法从数组末尾添加对象：

```
-(void)addObject:(id)anObject;
```

同样，获取某个索引处的对象使用继承自 NSArray 的 objectAtIndex 方法。

NSMutableArray 还可以删除某个索引处的对象：

```
-(void)removeObjectAtIndex:(unsigned)index;
```

同 C 数组一样，NSArray 和 NSMutableArray 也是从 0 开始索引的。此外，可变数组可以在特定索引处插入/替换某个对象，进行数组排序等。

（4）NSDictionary 和 NSMutableDictionary

字典也称散列表（Java 语言中）或者关联数组（C 语言中），在 Objective-C 中用 NSDictionary 表示字典。创建一个 NSDictionary 通常是使用类方法：

```
+(id)dictionaryWithObjectsAndKeys:(id)firstObject,...;
```

...代表了可变参数数组，一系列个数不能确定的同类型参数的集合。跟 NSArray 一样，一旦创建并初始化了一个 NSDictionary，其包含的对象就不能改变：

```
NSDictionary* d=[NSDictionary dictionaryWithObjectsAndKeys:
        obj1,@"first object",obj2,@"second object",nil];
```

同样，在字典的最后以 nil 标志结束。在初始化方法的参数列表中，除了 nil 外，所有的参数都是成对的，比如：对象 obj1 和字符串“fist object”是一对，对象 obj2 和字符串“second object”是一对。每对都代表字典集合中所存储的一个对象和它对应的索引键，当然，索引键在字典中是唯一的。我们可以通过索引键来检索字典中存放的对象：

```
-(id)objectForKey:(id)aKey;
```

这样，通过键“first object”查找字典 d 中存储的对象 obj1：

```
id something=[d objectForKey:@"first object"];
```

NSMutableDictionary 是可变字典。假设需要在字典 d 中增加新的对象，我们可以创建

NSMutableDictionary：

```
NSMutableDictionary* mutableD=[NSMutableDictionary
        dictionaryWithDictionary:d];
[d setObject:obj3 forKey:@"third object"];
```

如果新加入的对象所采用的索引键与字典中已存在的键相同，则原来的对象会被覆盖。这就是为什么方法名采用了 setObject 而不是 addObject 的原因。

要删除字典中的对象，可采用如下方法：

```
-(void)removeObjectForKey:(id)aKey;
```

（5）NSDate 和 NSDateFormatter

日期的显示非常重要，我们最常用到日期的地方就是获取当前时间，这也是为什么可以使用 iPhone 手机用来取代手表，并用它来做烦人的闹钟。

NSDate 的 date 类方法创建了一个临时的 NSDate 对象，它代表了当前时间：

```
NSLog(@"当前时间：%@",[NSDate date]);
```

这会在控制台输出如下内容：

当前时间：2011-7-30 15:37:02 -0570

NSDateFormatter 类用于和 NSDate 配合，将各种格式的字符串转换为 NSDate 或从 NSDate 转换为指定格式的文本。

日期转换为文本：

```
NSDateFormatter* df=[[NSDateFormatter alloc]init];
[df setDateFormat:@"yyyy/MM/dd"];
NSLog(@"%d",[df stringFromDate:[NSDate date]];
```

从文本转换为日期使用：

```
NSDate *date=[df dateFromeString:@"2011-7-30"];
```

3.1.4 常量、变量和宏

Objective-C 的变量声明和 C 完全相同：类型<变量名>。如果是声明对象的话，需要使用指针符号“*”，例如前面曾经见到过的许多代码：

```
NSDate* date;
NSString *string;
NSArray* array;
...
```

常量可以使用 static 关键字声明：

```
static NSString *string=@"abc";
```

也可以通过预定义宏来声明：

```
#define PI 3.14
#define PATH @"images";
```

3.1.5　#include 和#import

C 和 Objective-C 都使用头文件来声明类型、结构体、符号常量和函数原型等，因此需要通知编译器去头文件中查找相应的定义。为了实现这个目的，你可以使用#include 和#import 语句，二者的区别仅在于#import 是 GCC 编译器提供的，它可保证同一个头文件无论被包含多少次始终只导入一次。由于这个特性，你完全可以在 Objective-C 程序中用#import 取代#include。

HelloWorld 程序中的#import <Foundation/Foundation.h>文件语句中的第一个 Foundation 并不代表目录，而是指我们在第 2 章中提到过的 Foundation 框架。

提示：包含头文件时，框架和库中的头文件用尖括号引住，项目中的头文件用双引号引住。如#import <Foundation/Foundation.h> 和 #import "ViewController.h"。

3.1.6　函数

Objective-C 支持函数的声明和定义。但函数不是类的一部分。函数可以在.h 头文件中进行声明，也可以直接在.m 文件中定义（调用前）。关于函数的声明和定义，请遵循标准 C 的语法规范。

3.1.7　分支和循环

Objective-C 支持 C 的所有分支语句和循环语句：switch 语句、判断、while 循环、do…while 循环和 for 循环。除此之外，从 Objective-C 2.0 开始，支持快速迭代，类似 VB 中的 For…each 语法。使用快速迭代，可以简单地遍历数组元素：

```
For(NSString *each in array){
        NSLog(@"%@",each);
}
```

其中，array 是一个字符串数组。显然，这样的语法显得更加简洁明快。快速迭代不仅仅可用于数组，也可以用于遍历任何集合对象。

3.2　面向对象的 C

从现在开始，我们开始介绍期待已久的 Objective-C 的面向对象特性。

3.2.1　类和对象

面向对象最重要的概念就是类。通过类，我们可以实现面向对象的两个主要特性：继承和聚合。

在 Cocoa 框架中，NSObject 是所有类的根类，其他所有类从此开始继承。

1. 类的定义

类的定义在接口.h 文件中进行，典型的类定义如下面的代码所示：

```
@interface MyClass:NSObject
{
    NSString *name;
    NSArray* array;
    ...
}
@property(nonatomic,retain)NSString *name;
@property(nonatomic,retain)NSArray* array;
-(id)initWithName:(NSString*)string;
...
@end
```

从上面的例子中，我们可以看出以下几点：

- 类的定义由@interface 开始，到@end 结束；
- 类名后面紧跟冒号和所继承的父类名；
- 花括号中定义类的成员变量；
- 属性由@property 关键字声明；
- 方法声明位于成员声明之后，@end 之前。

2. 方法

方法类似于函数，虽然形式上有区别，但仍然由返回值、方法名和参数列表构成。其中，返回值和参数的类型说明使用圆括号括住，方法名分为多个部分，有几个参数，方法名就被分成几个部分，每个部分都有一个冒号分隔，冒号后面才是参数类型和参数名。此外，方法一般由一个方法类型符（+号或者–号）修饰，+号表示方法为类方法，–号表示方法为实例方法。例如：

```
-(id)initWithName:(NSString*)string withArray:(NSArray*)arr;
```

方法开头的–号表示这是一个实例方法。实例方法和类方法不同，实例方法属于实例对象所拥有，必须通过类的实例来调用；而类方法属于类所有，它直接通过类来调用，甚至不需要创建实例对象（等同于 C++/Java 中的 static 方法）。

这个方法有两个参数（数一数参数列表中冒号的个数，一个冒号代表一个参数），因此方法名由两部分组成，两个带冒号的单词：initWithName:和 withArray:。

记住，方法名实际上由“方法名+参数名”两部分构成。而参数名是后面带冒号的单词，如果该方法没有参数，则方法名仅包含第 1 部分（即只有方法名，而没有参数名）。而且方法的第 1 个参数的参数名就是方法名。因此有多少个参数，方法名就会由多少个带冒号的单词组成。记住这一点，因为这跟现存的许多语言都不一样。在 Objective-C 中，总是用 initWithName: withArray: 这样的形式表示一个方法名，而不是像 Java 一样用 init 来表示方法名。

Objective-C 吸收了 C 和 C++的一些令人称道的优点，比如可变参数。如果你在使用方法中总是拿不定总共需要多少个参数，那么你可以使用可变参数，例如 NSString 的 stringWithFormat:方法：

```
+(id)stringWithFormat: (NSString*)format, ...;
```

这个方法的第一个参数是 NSString 类型的参数 format，但第二个参数就是一个可变参数。这样的参数不仅没有确切的参数数目，而且也无法得知其具体类型。对于 stringWithFormat:方法而言，这些不确定因素只能通过已确定的参数 format 来确定。因此这个方法需要通过 format 字符串中的%@、%d 等字符串来确定第 2 个可变参数的数目和类型。如果可变参数的类型和数目与 format 参数中的格式字符串不相匹配，stringWithFormat:方法将无法正常工作。这样的例子还有 Objective-C 程序中经常用到的 NSLog 函数。

3. 属性

属性用于封装对成员变量的访问，是面向对象语言中的一个重要特性，Objective-C 也不例外。也许你已经在 C++和 Java 中熟悉了属性的概念，那么我们不再对此进行过多强调。

在将实例变量声明为属性的过程中，首先需要声明实例变量，然后使用@property 关键字。例如前面的代码中，首先用：

```
NSString *name;
```

声明了实例变量 name，然后用@property 将 name 声明为属性：

```
@property(nonatomic,retain)NSString *name;
```

提示：实例变量的声明不是必须的。如果省略实例变量声明的话，编译器会自动提供一个与属性名同名的实例变量。

然后在.m 文件的 implementation 语句后使用@synthesize<属性名>;语句。

当你这样做完之后，编译器会自动声明变量 name 的 get、set 方法，get 方法就是变量名，而 set 方法是单词 set 加上首字母被大写的变量名，例如：

```
-(NSString*) name;
-(void)setName:(NSString)newValue;
```

虽然这些声明在源代码中不可见，但确实是存在的，由编译器在编译时产生。

在其中比较复杂的部分是 nonatomic 和 retain 关键字的真实含义。在声明属性时，我们可以在@property 后面的圆括号中使用多个关键字修饰属性，以指定属性的可访问性、线程管理、内存管理等。由编译器根据这些修饰词产生不同的 get、set 方法。

nonatomic 关键字的意思是非原子的，意指对属性进行存取操作时是线程不安全的，如果在多线程环境下，该属性很可能是不同步的，一个线程读取属性值时，另一个属性却修改了属性值，这样两个线程对同一个属性进行操作的情况下，属性的值是不一致的。我们在属性中使用 nonatomic 的原因是，该属性不会在多线程环境下使用，使用非原子特性能得到较好的性能。而在 iOS 编程中，性能始终是程序员首先要考虑的问题。

retain 关键字是我们用得最多的属性修饰符之一，它和属性的内存管理有关。这样在对这个属性进行赋值操作时，在编译器自动生成（如果你使用了@synthesize 关键字的话）的 Set 方法代码中，会对实例变量进行 retain 操作。对于 Objective-C 对象类型的实例变量而言，使用 retain 操作使得属性在赋值后一直到对象被销毁之前始终可用。如果实例变量或属性并不是 Objective-C 对象类型，而是一个简单类型，如 BOOL、int、id、float，则用 asign 关键字替换 retain 关键字。这样，属性在赋值时不会被持有，这样导致的直接后果是：刚对一个属性赋值后，再访问这个属性，这个属性就变成空了。关于 Objective-C 的内存管理，更多内容会在后面介绍。

属性还可以用 readonly、readwrite 进行修饰，分别用于创建只读的属性和可读可写属性。

属性声明之后，接下来就是在实现文件（.m 文件）中实现属性的 get、set 方法。这些方法你可以选择自己去实现，也可以让 Objective-C 替你实现，只需要使用@synthesize 或者@dynamic 编译器指令。

也许你已经习惯书写传统 C++和 Java 的 get、set 方法，虽然实际上那是一件相当枯燥的事情。在 Objective-C 中，你完全不需要这样做，除非你真的需要。在实现的.m 文件中，使用@synthesize 关键字，可以自动产生属性的 get、set 方法代码。而这一切不是在源代码中进行的，而是编译器在编译时产生的，因此你不会在源代码中看到自动产生的代码。@dynamic 指令则是在运行时才产生这些代码。不管使用@synthesize 还是@dynamic，这些自动产生的代码均不可见，但它们是存在的。

3.2.2 消息机制

消息是 Objective-C 区别于其他语言的最大的地方，它其实是 Objective-C 特有的方法调用语法。同 C++和 Java 的方法调用一样：

1）消息能够接受参数；

2）消息可以嵌套调用；

3）消息既可以发送 Objective-C 对象，也可以发送给类。

1. 消息的参数

一个没有参数的消息是这样的：

```
[object methodWithoutParameter];
```

如果要调用的方法带有参数，则这个消息是这样的：

```
[object methodWithOneParameter: value];
```

注意，当方法没有参数时，其方法名后没有冒号。当方法有多个参数时，其方法名如我们前面所述，会有多个带冒号的参数：

```
[object methodWithFirstParameter:value1 withSecondParameter:value2];
```

2. 消息的嵌套

当要进行一个嵌套的方法调用时，会使用嵌套消息：

```
[textView setTextColor:[UIColor whiteColor]];
```

内层的消息[UIColor whiteColor]首先调用并返回一个 UIColor，然后将这个临时的 UIColor 对象作为外层消息的参数传入。

即第 1 个消息的输出作为第 2 个消息的输入。

3. 接收者

如前面所举的例子中，消息的第一个元素 object 和 textView 都是消息的接收者，即我们要将消息发给的对象。

除了接收者可以是对象以外，接收者还可以是一个类。因为在方法的种类中，不仅仅有实例方法，还有类方法。使用类作为接收者，就能调用类方法：

```
NSMutableString *mstring=[NSMutableString stringWithString:@""];
```

3.2.3　Objective-C 的内存管理

内存管理是资源管理的重要部分，在 iPhone 中尤其如此（因为其内存有限）。相对于 Java 语言的垃圾回收机制而言，Cocoa 的内存管理相当粗糙。这意味着对 Objective-C 程序员而言，需要做更多的工作，即使是相当有经验的程序员，在这方面也很难不犯错，尤其是不使用 ARC 的情况下。

本书不讲述 ARC（即 iOS SDK 5.0 以后的自动引用计数），因为在 iOS 企业应用中会大量使用第三方框架，这些框架大部分不使用 ARC，这给我们在项目中应用 ARC 带来许多困难——要么你要单独设置每个.m 文件的 ARC 选项，要么你得放弃使用该框架，这在很多时候给我们带来困惑。因此，本书不推荐使用 ARC。

1. 对象的生命周期

每个对象都有生命周期，程序员要知道一个对象的生命周期是什么，就需要仔细回忆一下这个对象是什么时候创建的。每当对象创建出来，它的生命就已经开始了，一直到操作系统释放了该对象，对象的生命才结束。但操作系统怎么知道一个对象应该释放了呢？Objective-C 采用了两种内存管理模式：基于计数器的内存管理和基于自动释放池的内存管理。这两种内存管理模式会采用不同的方式来通知操作系统，何时应该释放一个内存对象。此外，Objective-C 2.0 以后也支持基于垃圾回收的内存管理，但垃圾回收只限于 Mac，而无法在 iOS 中使用。

基于计数器的内存管理和基于自动释放池的内存管理我们在下面介绍。而基于垃圾回收的内存管理则不属于本书讨论的范围。

2. 基于计数器的内存管理

在这种模式下，iOS 对内存对象的生命周期管理是通过引用计数来进行的。每个对象都有一个引用计数器，它记录了对象被使用的情况。如果计数器的值为 0，表示该对象不再被使用，对象的 dealloc 方法被调用， dealloc 方法是对象的销毁方法，于是对象被销毁。

要知道一个对象的计数器值是多少，可以发送 retainCount 方法：

```
NSLog(@"%d",[object retainCount]);
```

当使用 alloc、copy、new 三种方法之中的任一种方法创建对象时，对象计数器会被自动设置为 1。此外，如果向对象发送 retain 消息，对象计数器会自动加 1。而向对象发送 release 消息，对象计数器会自动减 1。

因此，在使用计数器情况下的内存管理应当遵循以下原则：

- alloc、copy 或者 new 总是和 release 配套使用。如果你通过 alloc、copy 或者 new 方法创建了对象，那么必然跟随着一个 release。如果这个对象不是使用这 3 种方法之一创建的，则不需要一个配套的 release。
- retain 总是和 release 配套使用。如果对象创建后发送了 1 个 retain 消息，则需要配套 1 个 release 消息。
- release 的使用时机。对于临时对象，如果使用过 alloc、copy、new 以及 retain 之后，当不再使用该临时对象时，就可以使用 release 消息。如果该对象不是临时对象，而是类的成员，则在类的 dealloc 方法中 release 对象。
- 对于不是使用 alloc、copy、new 方法创建的对象，如果该对象是一个临时对象，则不要对该对象进行 retain 操作，也不要对该对象进行 release；如果该对象是类的成员，则需要在获取该对象时 retain，在 dealloc 方法中 release。

3. 基于自动释放池的内存管理

如前面讲述，使用基于计数器的内存管理十分麻烦，程序员必须对自己创建的每一个对象是否使用了 alloc、copy、new、retain 方法了如指掌，同时依据一系列规则适时地并在恰当的地方使用 release 消息。而且有时候要明白一个对象什么时候不再使用并不是一件容易的事情。

如果你不想使用对象计数器进行内存管理，则自动释放池是你的另外一个选择。

Cocoa 引入了自动释放池（autorelease pool）的概念。自动释放池实际是一个对象集合或对象容器的概念。如果你把所有的对象在创建时，都放到这个池中，则自动释放池被销毁时，池中的对象都会接收到 release 消息。

要把对象加入到自动释放池中，只需在创建对象的同时调用 autorelease 方法。该方法由 NSObject 提供：

```
-(id) autorelease;
```

例如下面的代码，当对象 string 一创建就被放到了自动释放池里：

```
NSString *string=[[[NSString alloc]initWithString:@"I am a string"] autorelease];
```

那么，这个自动释放池是什么时候创建的？我们并没有创建它，它是 Foundation 框架自动为我们创建的。当我们使用 Xcode 新建任何一个 iPhone 应用程序时，Foundation 框架创建的模板代码中总是有这样的代码，你可以在应用程序委托的实现文件中找到它：

```
NSAutoreleasePool *pool;
pool=[[NSAutoreleasePool alloc]init];
...
[pool release];
```

其实我们的 HelloWorld 程序中也有类似的代码。原来，应用程序一开始就创建了这个自

动释放池，并立即成为当前活动的池。我们使用对象的 autorelease 消息就是把对象放入这个池中。当程序退出时，最后的[pool release]或[pool drain]代码会向池中对象发送 release 消息。

这样，对象会从创建开始，一直存在到应用程序结束。这样无疑会造成一定的内存浪费，整个应用程序运行期间，随着对象的不断创建，内存只见增加不见减少，容易造成内存资源的紧张。很显然，并不适合内存使用率较高的应用程序。

本书的建议，程序员主要使用基于计数器的内存管理模式，自动释放池模式的使用则适可而止。

提示：[pool release]方法适用于 Mac OS 的所有版本，而[pool drain]方法只适用于 Mac OS 10.4（Tiger）以上版本。

3.2.4　类别和协议

1. 类别

类别是另一种为现有类添加新行为的方法。不同于子类，类别实际上是 Objective-C 的一种动态行为，它利用了运行时分配机制。因此，类别甚至不需要拥有原类的源代码。此外，类别不能向现有的类中添加实例变量。

要声明一个类别，你需要使用类声明时使用的@interface 关键字，如：

```
@interface NSString(NumberConvenience)
-(NSNumber*)lengthAsNumber;
@end
```

注意，这很像是一个类的声明。首先它使用@interface 类名开始，以@end 结束。但实际上，NSString 的类名加上其后的圆括号，表明这是一个对现有类 NSString 的补充或扩充。圆括号说明扩充方式为通过类别扩充而非通过继承来扩充。圆括号中的单词是类别的名称，本例中的类别名称是 NumberConvenience。

另外，第 2 行的方法声明语句也很像是 NSString 的成员方法。但实际上 NSString 类中根本没有这个方法。这个方法是类别为 Cocoa 类 NSString 新增加的。

此外，由于类别不能扩充实例变量这一限制，类别的声明中不会出现用于声明实例变量的{}符号。

类别主要是定义对现有类的扩充方法，因此类别的实现中（.m 文件）就需要对这些新方法一一进行实现：

```
@implementation NSString(NumberConvenience)
-(NSNumber*)lengthAsNumber{
    ...
}
@end
```

这个实现文件跟类的实现文件没有多大区别，除了圆括号中的类别名外。接下来，你可以像这样调用这个新方法 lengthAsNumber（假设 NumberConvenience 类别声明是定义在

NSStringCategory.h 头文件里）：

```
#import "NSStringCategory.h"
...
NSNumber* number=[@"hello" lengthAsNumber];
NSLog(@"length by NSNumber: %d", [number intValue]);
```

就好像 lengthAsNumber 本来就是 NSString 中已经存在的方法一样。

类别有许多好处，比如把类的实现分散在多个 implementation 文件里（如果使用类，你做不到这点，这就好像 C#中的分部类），或者用于创建非正式协议（如下面有关协议的内容所述）。

但是，类别仍然有一些限制，在类别的使用中一定要注意以下两点：

- 类别仅仅是一堆方法的集合，你无法为类添加实例变量。
- 类别不能解决命名冲突，如果类别中的方法与类原有的方法重名，则类中的方法被覆盖。

2. 协议

Objective-C 的协议等同于 Java 中接口的概念。下面我们来讨论协议的声明:

```
@protocol NSCopying
-(id)copyWithZone: (NSZone*)zone;
...
@end
```

协议看起来就像类别的声明，还是一堆方法声明的集合。但@protocol 关键字的出现说明了这是一份正式协议。

正式协议的意思是，每个采用（Java 中用实现 implements 这个词）这份协议的类必须实现这份协议中的所有方法。这种说法一直持续到 Objective-C 2.0。

从 Objective-C 2.0 开始，协议中的方法可以有选择地由类实现。对于需要实现的方法，使用@required 关键字修饰，对于可选择性地实现的方法，使用@optional 关键字修饰，比如：

```
@protocol TheProtocol
@required
-(void)firstMethod;
-(void)secondMethod;
@optional
-(void)thirdMethod;
@end
```

这个协议规定，第 1、2 个方法是必须实现的，而第 3 个方法可以实现，但不要求一定实现。

如果一个类要采用（或实现）这个协议，则需要在@interface 中这样声明：

```
@interface MyClass:NSobject<TheProtocol>
```

而 MyClass 类的实现中，则应当实现该协议规定的 2 个必选方法及 1 个可选的方法。

除了正式协议外，我们也可以采用非正式协议。非正式协议不需要采用@protocol 关键字

声明，但需要创建一个类别，例如 MyCategory：

```
@interface NSObject(MyCategory)
-(void)doSomething;
...
@end
```

由于 NSObject 是所有 Cocoa 类的根类，这个类别实际上指明了任何类都可以实现这些方法。从而可以向任何对象发送这些消息，而不需要在类的声明中做任何特别的说明。

3.2.5　反射机制

同其他高级语言一样，Objective-C 也提供了运行时支持——即 Java 所谓的反射机制，这充分体现了 Objective-C 的动态性特征。尤其在 Mac OS X 10.5 以后，苹果对 Objective-C 运行时 API 进行了重要的升级，以提供对 64 位模式的支持。

Objective-C 运行时 API 包含了众多函数和结构体的定义，然而，我们不准备在此介绍庞大的 Objective-C 运行时库。如果要全面了解这些 API，请翻阅苹果运行时库参考。

通常，我们不需要直接调用 Objective-C 运行时 API，Cocoa 框架已经把它的许多有用函数进行了封装，从而使我们更容易调用。

1. 获取类信息

首先，我们知道 NSArray 等集合对象中不限制所存储的对象类型，只要它是一个 NSObject 就行。但有时候，我们想知道我们刚刚放进去的一个对象到底是什么类型（程序员总是那么健忘），是一个字符串？还是别的什么？

我们可以发送 class 消息：

```
id class=[[array objectAtIndex:0]class];
```

id 类型（即对象）其实是一个 objc_object 结构：

```
typedef struct objc_object {
    Class isa;
} *id;
```

NSObject 的 class 方法实际是上返回了这个结构中的 isa 成员。isa 指向了一个 Class 对象，因此除了发送 class 消息外，我们也可以使用 isa 成员来访问 Class 对象。Class 对象指向了该对象所属的类（即“类对象”，Objective-C 把类也看成是对象，以贯彻“万物皆对象”的原则）。

Class 对象实际上是一个 objc_class 结构：

```
typedef struct objc_class *Class;
struct objc_class {
    Class isa;
    Class super_class;
    /* followed by runtime specific details... */
};
```

注意：这个结构很象是 objc_object，但多了一个指向父类的 super_class 成员。也就是说，对象跟类的区别仅在于，对象中不保存继承关系，而类（或“类对象”）保存了继承关系。因此我们是通过类而不是对象来追溯类的“父类”、“祖父类”等。

我们有时需要比较一个对象是否属于某个类，可以使用类似的代码：

```
if (obj->isa ==[NSString class]){
    ...
}
```

而 super_class 成员指向了父类对象，通过它我们可以访问父类的信息。

在 Cocoa 中还有一个“元类”的概念，即“类对象的类”。因此完整的类信息称为“类对”，即由“类对象”和“元类”构成。

如果向一个对象发送 class 方法，我们可以得到“类对象”，而如果再向“类对象”发送 class 消息，则返回的就是“元类”（meta class）。每个类只能有一个元类，其包含了类方法列表。而类对象不同，它包含了对象的方法列表。

2. 选择器

选择器实际上是一个方法名称，用@selector 关键字来指定一个选择器，选择器用于查询对象的某个方法，例如：

```
@selector(setResponse:)
```

通过 NSObject 的 responseToSelector 方法，我们可以动态地查询某个对象是否存在指定的方法。该方法需要指定一个选择器参数：

```
if([object responseToSelector: @selector(setResponse:)]){
    ...
}
```

如果 object 对象能够响应 setResponse:方法，则返回 YES，否则返回 NO。

如果确定某对象能响应指定方法，则可以通过 performSelector:方法进行调用：

```
[anObject performSelector:@selector(method)];
```

如果该方法带有参数，则使用 performSelector: withObject:方法传递参数：

```
[anObject performSelector:@selecor(method) withObject:obj];
```

如果有 2 个参数，则使用 performSelector: withObject: withObject:。更多的参数或者有返回值，则使用 NSInvocation。

除了向对象发送 performSelector 消息之外，Objective-C 还提供了 objc_msgSend 函数，你可以用它向任何对象发送一条消息：

```
objc_msgSend( anObject,@selector(method),obj);
```

要使用 objc_msgSend 函数，需要导入<objc/message.h>头文件，其完整定义如下：

```
id objc_msgSend(id theReceiver, SEL theSelector, ...)
```

函数的第 1 个参数指向消息的接收者（即该方法的对象），第 2 个参数是一个选择器（即方法），第 3 个参数是一个可变参数，是该方法的 1 个或多个参数，如果该方法没有参数，用一个 nil 代替。

方法的返回值通过函数的返回值返回。

3. 类的动态创建

要在代码中创建类，而不是通过静态的.h 和.m 文件定义类，可以使用 Object C 运行时库 API（需要#include <objc/runtime.h>）：

```
Class newClz=objc_allocateClassPair([NSError class], "RuntimeErrorSubclass", 0);
class_addMethod(newClz, @selector(report), (IMP)ReportFunction, "v@:");
objc_registerClassPair(newClz);
```

首先，使用 objc_allocateClassPair 动态函数创建了一个类，并在参数中指明该类的父类和类名。用 class_addMethod 函数为该类增加了一个方法 report，这个方法是由函数 ReportFunction 实现的，由于该函数至少应包含两个参数 self 和_cmd，因此定义了该方法有 3 个参数，类型分别为 v、@、:（一个返回值，self，_cmd）。

v 代表 void，指定了方法的返回值；@代表了 id 类型（对象），指定了方法的固定参数 self；: 表示选择器类型（SEL），指定了固定参数_cmd。因此函数 ReportFunction 应当实现为：

```
void ReportFunction(id self, SEL _cmd)
{
    // 实现代码
}
```

最后，新类被注册为类对（Class Pair）。

类对注册后，即可在代码中这样使用类 newClz：

```
id obj =[[newClz alloc] init];
[obj performSelector:@selector(report)];
[obj release];
```

4. 类的动态加载

Cocoa 的 Foundation 框架提供的 NSClassFromString 函数类似于 Java 的 Class.forName()方法：

```
Class clz=NSClassFromString(@"MyClass");
```

返回对象为“类对象”Class。通过这种方法，我们从字符串构建类实例就不再是什么问题：

```
id obj=[[clz alloc]init]
```

或者

```
id obj=[[NSClassFromString(@"MyClass") alloc]init];
```

这使得我们可以在运行时而不是编译时加载类，同时，不需要#import “MyClass”。

5. 方法的动态调用

通过@selector 关键字我们已经可以在一定程度上实现方法的动态调用。然而更动态的方式是通过 Fundation 框架的 NSSelectorFromString 函数，它可以直接从字符串获得一个选择器：

```
SEL sel  =  NSSelectorFromString(@"doSomethingMethod:")
if([object  respondsToSelector:sel]) {
    [object  performSelector:sel  withObject:color];
}
```

3.2.6 谓词

Cocoa 提供 NSPredicate 类，用于描述条件和计算对象是否匹配指定条件。类似于 SQL 语句，通过 NSPredicate，可以将条件的计算从代码中分离出来，从而在比较对象时避免使用硬编码。

1. 创建谓词

首先需要创建一个 NSPredicate 对象：

```
NSPredicate *predicate=[NSPredicate redicateWithFormat:@"name=='Herbie'"];
```

类方法 predicateWithFormat 常用于创建一个 NSPredicate 对象。仅需要提供一个代表计算条件的字符串。字符串@"name=='Herbie'"代表了需要进行计算的 C 条件表达式。事实上，除了==运算符外，谓词的条件表达式支持括号和 C 语言的比较运算符和逻辑运算符。

下面我们来看看谓词是如何进行计算的。

2. 谓词的计算

要对谓词进行计算，需要将谓词应用在某个对象身上。例如：

```
BOOL match=[predicate evaluateWithObject: car];
```

谓词 predicate 的 evaluateWithObject 方法实际上对 car 对象进行了计算，并返回 BOOL 值 YES 或 NO。前面的条件表达式@"name=='Herbie'"，表明将用 car 的 name 属性与字符串 Herbie 进行比较，如果相等，则返回 YES，否则返回 NO。

谓词可以用任何对象作为参数进行计算，包括数组或集合。如果 persons 是一个包含了多个 person 实例的 NSArray 数组，那么我们可以用谓词计算或过滤出其中 gender（性别）属性为男性的对象：

```
NSPredicate* predicate=[NSPredicate predicateWithFormat: @"gender=='M'"];
NSArray *result=[persons filteredArrayUsingPredicate: predicate];
```

谓词将对 persons 中的所有对象进行条件计算，把 gender 等于 M 的对象加到结果集数组中返回。

3. 在条件中使用变量

有时候，在构建谓词时，使用变量组成条件表达式将更方便。这就类似于 JDBC 中的预编译语句，Where 条件子句中的表达式可以使用可变参数。只需在 SQL 语句中使用问号？来代表可变参数即可。这样的好处是显而易见的，我们就可以重用预编译语句，节省数据库编译上 SQL 语句的时间。我们只需在执行预编译语句时，提供不同的可变参数值，即可执行不同的 SQL 查询。

同样，我们也可以在条件表达式中使用格式化字符串或占位符，来作为条件表达式中的可变部分。例如：

```
NSPredicate* predicate=[NSPredicate predicateWithFormat:@”name==%@”,@”Herbie”];
```

这和原来没有任何区别。占位符%@表示这里将被一个对象（id 类型）所替换。如果想在表达式左边使用变量，可以使用%K 作为占位符，例如：

```
predicate=[NSPredicate predicateWithFormat:@”%K==%@”,@”name”,@”Herbie”];
```

这与前面一条语句是等效的。

提示：%K 中的字母 K 代表的是 keypath（键路径）。在后面介绍。

除此之外，我们还可以使用 NSLog 中使用的各种占位符，如%d 等。当然，不使用占位符，还可以使用命名的键-值对（NSDictionary）作为条件变量。例如：

```
predicate=[NSPredicate predicateWithFormat:@”name==$NAME”];
```

这里$NAME 就代表了一个变量（而不是对象），这个变量值未指定。如果要指定这个变量的值，需要在 NSDictionary 中加入一个 Key 为 Name 的对象，同时用这个 NSDictionary（Cocoa 将这个 NSDictionary 称为环境变量 SubstitutionVariances）创建新的 predicate：

```
NSPredicate* predicate=[NSPredicate predicateWithFormat: @"name=$NAME"];
predicate=[predicate predicateWithSubstitutionVariances: dic];
```

然后使用新的 predicate 去进行计算。也就是说，通过这种方式，我们提交一个 NSDictionary 作为谓词变量。

4. BETWEEN 和 IN

BETWEEN 运算符用于表示位于某个区间中的取值。例如：

```
predicate=[NSPredicate predicateWithFormat:@”age BETWEEN{16,50}”];
```

该条件表达式表示 age 属性值在 16 到 50 之间的对象。

IN 用于测试某个值处于一个集合范围内的情况。例如：

```
predicate=[NSPredicate predicateWithFormat:
    @"name IN {'Herbie','Elvis','Phoenix'}"];
```

该条件表达式表示 name 属性等于这三者之一的对象。

5. 字符串匹配

谓词常用于判断字符串的匹配，除使用比较运算符外，谓词表达式还支持多种对字符串进行计算的运算符：

- BEGINSWITH 用于判断一个字符串的开头是否包含另一个字符串。
- ENDSWITH 用于判断一个字符串的结尾是否包含另一个字符串。
- CONTAINS 用于判断一个字符串中是否包含另一个字符串。
- LIKE 运算符用于进行字符串的匹配。如同 SQL 语法一样，谓词表达式中的 LIKE 运算符支持两种通配符：*和?，*号匹配任意个数的任意字符，?号匹配 1 个任意字符。

此外，[c]选项用于表示忽略大小写，[d]选项用于表示忽略发音符号,[cd]选项用于表示同时忽略大小写和发音符号。例如：

```
predicate=[NSPredicate predicateWithFormat: @"name BEGINSWITH[cd] 'HER'"];
```

提示：如果你想在谓词表达式中使用正则式，则需要使用 MATCHES 运算符。

---

6. KeyPath 和 SELF

Cocoa 中经常引用 KeyPath 的概念。键路径 KeyPath 实际上和文件系统中路径概念无关，KeyPath 是面向对象语言如 Java 中“.”语法的概念，比如：

```
indexPath.row
```

这就是一个 KeyPath。

用前面的例子，我们想匹配 persons 数组中名称长度超过 10 的对象，那么在谓词表达式中不能使用 name 作为主语了（假设我们把谓词表达式分为“主语+运算符+宾语”结构，位于运算符左边的操作数是主语，而位于运算符右边的操作数就是宾语）。因为 name（具体地说，指数组中某个对象的 name 属性）只是一个字符串，它的长度应该用 name.length 表示。如前面所说，name.length 就是一个 KeyPath，用它作为表达式的主语是恰当的。那么，这个谓词应该这样构建：

```
predicate=[NSPredicate predicateWithFormat:@"name.length > 10"];
```

然后再对 persons 数组中的每个对象进行计算：

```
NSArray *result=[persons filteredArrayUsingPredicate: predicate];
```

其实 KeyPath“name.length”还可以是“SELF.name.length”，这二者是完全等效的。SELF 用来表示应用了谓词的对象。SELF 经常可以省略，但在一种情况下不能被省略。让我们来假设这样的情况：

```
predicate=[NSPredicate predicateWithFormat:@"SELF ==@'Herbie'];
BOOL match=[predicate evaluateWithObject:@"Rubby"];
```

先看第 2 句，我们把谓词应用到一个字符串对象@"Rubby"。那么第 1 句中的 SELF 就不能省略了。该谓词的 KeyPath 使用 SELF，表示我们要用它（而不是它的属性）直接和另一个字符串@"Herbie"进行比较，这是可以理解的。在这种情况下，你不能省略 SELF，否则谓词表达式就不完整了，因为它缺乏了“主语”这个关键的语法元素。

3.3　MVC 模式

MVC 模型是应用程序设计者们普遍采用的一种设计模式，在第 2 章介绍 Cocoa Touch 框架时曾简单介绍了 MVC。MVC 模式把应用程序 GUI 代码根据功能拆分为不同的类或组件:

- “模型”：用于封装应用程序的数据；
- “视图”：负责显示和编辑数据：
- “控制器”：负责处理前两者之间的逻辑关系。

它们之间的逻辑关系参考第 2 章的图 2-3。

Cocoa Touch 本身也遵循 MVC 模型原则。在 MVC 模型下，3 个层次都由截然不同的类来实现，编写任何类的代码都应当明显地归为其中一类，并且其大部分功能代码不应当属于另外两类。这种分工负责的方式使得程序易于设计、实现和维护。

一般情况下，我们会在 Interface Builder 中创建视图组件（关于 Interface Builder 的使用，我们会在第 5 章中进行介绍）。或者，使用 Xcode 通过代码的方式继承已有的视图和控件。

模型负责保存应用程序数据，通常我们使用 Objective-C 对象或者 Core Data 来构建模型组件。

控制器组件可以使用 UIKit 控件中的 ViewController 及其子类，或者是完全由程序员自已定制的类。

从第 4 章开始，我们将开始在 iOS 应用程序开发中逐步应用 MVC 模型的基本理论来构建应用程序框架。始终遵循 MVC 模型的基本理论，将有助于你创建出更加简洁、易于维护的代码。

3.4　KVO 模型

KVO（key-value observer，“键-值”观察）模型是 Cocoa 绑定技术中常用的一种编程模型，它可以使一个对象在属性值发生变化时主动通知另一个对象并触发相应的方法。与 NSNotification 不同，KVO 没有所谓的中心对象来为所有观察者提供变化通知。当“被观察者”对象状态发生变化时，通知被直接发送至“观察者”对象，如图 3-1 所示。

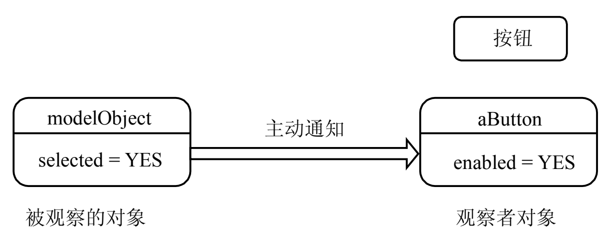


图 3-1　KVO 模型

在图 3-1 中，观察者是 aButton，被观察者是 modelObject。aButton 是一个按钮控件，它是一个 UI 对象，有一个 BOOL 型的 enabled 属性，表示按钮是否可被点击。modelObject 对象是一个模型对象，它没有可以呈现给用户的界面，同样，它有一个 BOOL 属性 selected。通过 KVO 模型，modelObject 的 selected 属性可以绑定到 UIButton 的 enabled 属性。即当 modelObject 的 selected 属性发生变化时（这是可以被编程的），KVO 会主动通知 aButton 这种改变，因此按钮的外观随之可发生相应的呈现。比如由不可点击的灰色改变为可点击的着色状态。

KVO 是一种很有用的绑定技术（Cocoa 还提供另外一种绑定技术：Dynamic bingding）。而且它是由被观察的对象主动通知观察者的，并不需要经过一个统一的通知中心（如后面章节介绍的通知技术所述），它的执行效率和适用场景要更佳。

为了实现 KVO，你需要进行如下操作：

- 注册观察者。所谓观察者即对象状态变化时需要通知的对象。
- 接收变更通知。接收变更通知主要是让观察者实现指定方法，在指定方法中，你可以接收到对象状态变更的消息，并在方法中进行处理。
- 取消所注册的观察者。观察者处理完状态变更消息之后，需要取消原先的注册状态。

3.4.1　注册 KVO

对象要将自己注册为观察者，必须发送一个 addObserver:forKeyPath:options:context:消息至被观察对象：

```
[account addObserver:inspector
           forKeyPath:@"name"
              options:(NSKeyValueObservingOptionNew |
                       NSKeyValueObservingOptionOld)
              context:NULL];
```

以上例子将 inspector 对象注册为 account 对象的观察者，并表明观察者将对名为“name”的属性变更感兴趣。

forKeyPath 参数“name”注明了需要观察的属性的关键路径 KeyPath。关键路径 KeyPath 实际是一个字符串，用于表示某个属性，你可以直接用属性名。但如果某属性是一个对象，则 KeyPath 可以用“.”语法的形式表示对象成员，如“account.name”。

options 参数注明了对该属性的何种状态感兴趣。NSKeyValueObservingOptionNew 表示属性在变更后的新值，NSKeyValueObservingOptionOld 表示属性未改变之前的值。以上例子中的 option 参数设置表明，当 name 属性变更时，会将这两个值以 NSDictionary 的方式（即 change 参数）提交给观察者，观察者可以从 NSDictionary 中以键－值对的方式检索到这两个值。

context 参数用于传递一个对象，该对象（或指针）会在属性变化时通过变更通知传递给观察者（通过 context 参数）。

移除观察者的注册，使用方法 removeObserver forKeyPath：

```
[subject removeObserver:observer forKeyPath:@"name"];
```

3.4.2　接收变更通知

观察者要想收到对象的属性变更通知，需要实现方法 observeValueForKeyPath:ofObject:change:context:，并在其中进行通知的处理。例如：

```
- (void)observeValueForKeyPath:(NSString *)keyPath
                  ofObject:(id)object
                  change:(NSDictionary *)change
                  context:(void *)context
{
  NSLog(@"%@",keyPath);
    if ([keyPath isEqual:@"name"]) {
         NSLog(@"name is changed:%@",
         [change objectForKey:NSKeyValueChangeNewKey]);
    }else
    [super observeValueForKeyPath:keyPath
                  ofObject:object
                  change:change
                  context:context];
}
```

3.4.3　发送变更通知

NSObject 支持两种属性变更通知，一种是自动变更通知，一种是手动变更通知。一般情况下，使用自动变更通知则更为简单，因此我们主要介绍自动变更通知。

1. 自动变更通知

要使用自动变更通知，需要实现被观察者的 automaticallyNotifiesObserversForKey 方法，在此方法中明确说明需要使用自动变更通知的属性。对于需要使用自动变更通知的属性，返回 YES，如下代码所示：

```
+ (BOOL) automaticallyNotifiesObserversForKey:(NSString*)key
{
// 对于属性 name ，使用自动通知
```

```
    if ([key isEqualToString:@"name"])
    {
        return YES;
    }
// 确保调用了父类的 automaticallyNotfiesObserversForKey 方法
    return [super automaticallyNotifiesObserversForKey:key];
}
```

然后，在 name 属性发生变化的时候通知观察者，比如调用以下语句之一：

```
subject.name=newName;
[subject setValue:newName forKey:@"name"];
[subject setValue:newName forKeyPath:@"name"];
```

如果属性是集合类型，则可以使用方法 mutableSetValueForKey 来支持以下集合方法导致的自动变更通知：

- 添加：insertObject:InKey:或者 insertObject:AtIndex:
- 替换：replaceObject:InKey:或者 replaceObject:AtIndex:
- 删除：removeObjectFromKey:或者 removeObjectAtIndex:

2. 手动变更通知

对于手动变更通知，除了需要在 automaticallyNotifiesObserversForKey：方法中将要使用的手动变更通知返回 NO 外，还需要 在改变值之前调用 willChangeValueForKey:并在更改它之后调用 didChangeValueForKey:。

为了便于你理解 KVO 模型，我做了一个示例程序，放在光盘“source/第 3 章/TestKVO”文件夹，它使用了本节所介绍的知识点，可供参考和学习。

3.5 块编程

C 语言的运行时特性中包括了块，标准 C 工作组的 N1370: Apple’s Extensions to C 中(其中也包括垃圾回收)对块进行了定义。作为 C 语言的扩展，Objective-C 在 OSX 10.6 及 iOS 4.0 以后支持块语法。块运行时也会被集成到 LLVM 的 compiler-rt 子项目存储库中。

3.5.1 块的特点

一些面向对象的动态语言如 ruby、groovy，都提供了对块的支持（在 groovy 中，块被称作为闭包“closure”)。块是用一对{}括号括起来的多个语句的集合。类似于函数，但不同于函数，可以把块作为表达式或变量的一部分，或者作为参数传递。在作为参数传递块时，代码被作为数据的一部分进行传递。

块具有以下特征：

- 同函数一样，有类型化参数列表。

- 有返回结果或者要申明返回类型。
- 能获取同一作用域（与块所在同一作用域）内的状态。
- 可以修改同一作用域的状态（变量）。
- 与同一范围内的其他块同享变量。
- 在作用域释放后能继续共享和改变同一范围内的变量。

除以上特点外，甚至可以复制块并传递到其他后续执行的线程，编译器和运行时负责把所有块引用的变量保护在所有块的拷贝的生命周期内。当然，这已经超出了本章的范围，可以参考苹果官方文档来了解这些内容。

3.5.2　Objective-C 中的块

对于 C 和 C++，块是变量，但对于 Objective-C，块仍然是对象。下面简单介绍 Objective-C 中的块。

1. 块变量声明

用^操作符声明一个块变量的开始，分号表示块结束，如下代码所示：

```
int multiplier = 7;
int (^myBlock)(int) = ^(int num) {
    return num * multiplier;
};
```

块语法比较奇怪，块变量声明的解释如图 3-2 所示。

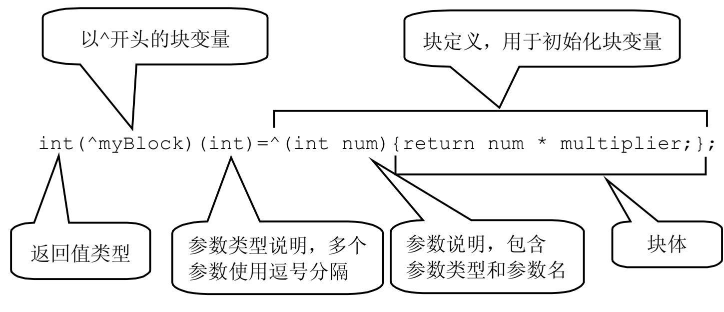


图 3-2　块变量声明的解释

块变量的声明语句从前至后分为了几部分：

- 返回值类型，如 int、double，如果未显式地声明块的返回值类型，可能会自动从块代码中推断返回类型（通过 return 语句）。
- 块变量名用括号括住，块变量名前加^符号。
- 参数类型用括号括住，多个参数以逗号分隔，如果参数列表为 void，而且返回类型依靠推断，可以省略参数列表的 void。
- 等号，将后面的块赋值给前面的块变量（即 myBlock）。

- 以^开头并以;结束的块定义。

块定义中又分为以下两个部分（除去开头的^和结尾的;外）：

- 参数列表，同函数的参数列表。
- 块体，同函数体。

值得注意的是，块可以使用同一作用域内定义的变量，而函数不行。

一旦声明了块，你可以像使用函数一样调用它：

```
int multiplier = 7;
int (^myBlock)(int) = ^(int num) {
    return num * multiplier;
};
printf ( "%d", myBlock(3));
```

2. 行内块

有时候，你不准备重复使用某个块，因此你不必为它想一个名称。那你可以使用行内块而不用声明为块变量。以下代码来自苹果文档：

```
// gsort_b 类似标准的 gsort_r 函数，但它最后一个参数是一个块。
char *myCharacters[3] = { "TomJohn", "George", "Charles Condomine" };
qsort_b(myCharacters, 3, sizeof(char *), ^(const void *l, const void *r) {
    char *left = * (char **)l;
    char *right = * (char **)r;
    return strncmp(left, right, 1);
});
// myCharacters 现在是 { "Charles Condomine", "George", TomJohn" }
```

在 qsort_b 方法调用中，第 4 个参数就是一个匿名的块（行内块）。匿名块跟块变量不同，它没有变量名，因此你无法重用匿名块。下次调用这个块时，必须把整个块定义的代码再复制一遍。

3. __block 关键字

块允许访问本地变量。这很重要。它使得我们在线程间共享变量变得简单，而且，你可以规定一个本地变量是否可以写，这可通过使用__block 关键字，这是一种类类似 register、auto 和 static 存储类型修饰符。

用__block 修饰的变量，可以在所有同一作用域内的块，以及块复制之间共享数据。在指定作用域内的多个块能同时使用共享变量。

如同块，__block 变量也使用栈存储。如果使用 block_copy 拷贝块（或者向块发送 copy 消息），变量被拷贝到堆里。而且，__block 变量的地址随后就会改变。

__block 变量有两个限制：不能是可变长度的数组，也不能是包含 C99 可变长度数组的结构体。

下面显示了__block 变量的使用：

```
__block int x = 123; //  x 是块可写的
void (^printXAndY)(int) = ^(int y) {
     x = x + y;
     printf("%d %d\n", x, y);
};
printXAndY(456); // 打印出: 579 456
// x 现在的值是: 579
```

下面显示了在块中使用多种类型的变量：

```
extern NSInteger CounterGlobal;
static NSInteger CounterStatic;
{
     NSInteger localCounter = 42;
     __block char localCharacter;
     void (^aBlock)(void) = ^(void) {
          ++CounterGlobal;
          ++CounterStatic;
          CounterGlobal = localCounter;
          localCharacter = 'a';
     };
     ++localCounter;
     localCharacter = 'b';
     aBlock();
}
```

3.6 可变参数

我们知道，C 和 C++语言支持可变参数的函数，例如我们常用的 NSLog 和 printf 函数。Objective-C 作为 C 语言的超集，当然毫无例外地也支持可变参数。迄今为止，我们至少用过了一种使用可变参数的方法，即 NSString 的 stringWithFormat:方法。

C 语言通过 stdarg.h 库支持可变参数，Objective-C 也不例外。在 C 语言中，如果你要使用可变参数，必须包含头文件 stdarg.h，但在 Cocoa 中却不必，因为苹果已经在 NSObjC Runtime.h 中包含了 stdarg.h。

stdarg.h 的定义如下：

```
typedef __void * va_list;
#define va_start(ap, param) __builtin_va_start(ap, param)
#define va_end(ap)           __builtin_va_end(ap)
#define va_arg(ap, type)     __builtin_va_arg(ap, type)
```

首先定义了一个 va_list 类型，其实就是一个 void*，即可以指向任何类型的指针。你可以把它看成是 char*，因为 char*实际上也可以指向任何内存单元的地址。

然后是 3 个预定义宏，va_start、va_end 和 va_arg。可以看出 stdarg.h 完全是以“预定义宏”

这种“古老”的方式来支持可变参数的。

接下来我们看一个例子，该方法使用了一个可变参数，并将这些可变参数进行了累加，然后返回一个 NSNumber：

```
- (NSNumber *) addValues:(int) count, ... {
    va_list args;
    va_start(args, count);
    NSNumber *value;
    double retval;
    for( int i = 0; i < count; i++ ) {
        value = va_arg(args, NSNumber *);
        retval += [value doubleValue];
    }
    va_end(args);
    return [NSNumber numberWithDouble:retval];
}
```

代码说明：

第 1 行是方法定义，该定义应当加到头文件中。省略号...表明方法接收一系列数目不定的参数，在...前面至少需要指定一个任意类型的参数。有时我们必须知道参数的个数以防止出现无效的引用，但在某种情况下，参数个数是可以通过其他参数推断出来的（例如 NSLog 或 printf 函数可以通过计算%号的个数推断可变参数的个数），或者对于 NSMutableArray 来说，它总是以 nil 终止。

如果是最后一种方法，我们可以把方法重新定义为：

```
- (NSNumber *) addValues:(NSNumber *) firstNumber, ...
```

这样，我们就可以用以下调用方式代替“addValues:3,num1,num2,num3”：

```
addValues:num1,num2,num3;
```

这样，我们就可以省略第 1 个表示可变参数个数的 int 参数。

第 2 行中的 va_list 是 void *类型，因此它实际上是一个可变的对象数组。

第 3 行用 args 来存放可变参数列表，而 count 则表示函数最后一个参数（即第一个“固定参数”）。这将使编译器把 args 指向第 1 个参数后的位置（通过 count 地址加上 count 变量的长度）。

很奇怪吗？可变参数中第 1 个参数的位址为什么是“count 地址+count 的长度”？因为对于大多数 C 编译器，函数栈中参数的存放顺序是从右到左的，也就是说先放入可变参数的最后一个参数，再放可变参数的倒数第 2 个参数……，然后放可变参数的第 1 个参数，最后是固定参数 count。而与此同时，栈的方向是向下的，即先入栈的数据位于高地址，后入栈的数据则位于栈的起始地址。这样，实际上最后放入的固定参数 count 的地址变成了栈的起始地址。而紧随 count 之后的地址则是可变参数的第 1 个参数地址，即“count 地址+count 的长度”，因

此编译器要能找到第 1 个可变参数的地址，只要知道 1 个参数：count 就够了，由 count 取得函数栈的起始地址，加上 sizeof(count)，得到第 1 个可变参数的地址。va_start 的第 1 个参数 args 是一个输出参数，经过 va_start 调用之后，args 将等于 arg_start 计算出来的第 1 个可变参数的地址。

第 6 行是一个 for 循环，因为我们无法通过 args 自身推断 args 的大小，因此必须显式地用 count 来指定 args 的大小。或者可以使用 nil 终止的列表来检索可变参数。

如果你使用 nil 终止的数组作为可变参数，则应该用下面一行来代替第 6 ~ 7 行：

```
while( value = va_arg( args, NSNumber * ) )
```

第 7 行将 args 中的下一个参数放入 value，并显式地转为 NSNumber*（如果不知道类型，可以用 id）。

第 10 行表明，一旦使用完 args 列表，就关闭它 。

提示：如果你使用 va_arg(args，double)（或者 float 等其他原始类型），那么当你试图传递一系列整数作为参数时（例如：addValues:4,4,3,2,1），可能会出现一些古怪的结果。而如果你显式地将这些参数说明为 double（例如，double num1,double num2,double num3,double num4）则不会有什么问题。

这是因为，如果编译器看到一个方法有一个 double 参数但你却传递了一个整数给这个方法时，它会进行类型转换。但如果方法使用了可变参数，编译器无法知道参数所使用的类型，因此编译器只会简单地把参数作为整型处理。

3.7　本章小结

Objective-C 是 C 语言的扩展和超集。本章重点从 C 语言特性和面向对象特性两个方面对 Objective-C 的语言特性进行论述，包括基本语法（数据类型、常量 / 变量、分支与循环）、运行时特性（反射支持）和一些特有属性（NS 类、类别和协议、消息、KVO 和块）。通过这些介绍，我们对 Objective-C 的一些重要特性有了最基本的了解，从而为后续的学习打下坚实的基础。接下来，本书将陆续介绍 iOS SDK 中的一些重要框架，如 UIKit、Core Animation、Quartz Core，以及其他一些第三方的开源框架。

如何结合iOS操作系统、iOS SDK，以及诸多相关开源框架的特点，开发出满足企业需求的iOS应用，这是每个企业级应用开发人员需要考虑的问题，本书给出了详细解决方案。本书不仅介绍了iOS开发中的一些基础的、具有共性的技术点，还重点讲解了企业级iOS开发中特有的难点，如Objective-C内存管理、iOS 的多任务、企业APN、多线程和GCD等。此外，还讲解了在云计算环境下，如何实现企业级iOS应用的安全和网络等方面的需求。

特别地，本书对iOS的多任务、多线程概念进行了详细的解读，给出了全新的编程模型范例。本书的最后是两个综合案例："企业APN"和"网络应用实战"，对书中的理论给出了实战解析。随书附赠光盘包含全部案例代码。

本书主要内容：

- 介绍iOS企业开发中的两个典型特征：企业IDP和无线部署。
- Objective-C语言中的新特性，如：块、可变参数、运行时、并行编程等。
- 全面介绍了Xcode IDE和Interface Builder的使用、快捷键及连接技巧。
- 如何使用ASIHTTPRequest简化网络编程。
- 如何使用PLDatabase访问数据库。
- 详细讲解对Cocoa的两种进程唤醒技术：本地通知和远程通知。
- 对UIKit进行扩展：自定义组件和静态库。
- 介绍了SDK的Security框架及两个安全算法库：OpenSSL和CommonCrypto。
- 如何使用图表框架CorePlot在应用程序中绘制图表。
- 介绍Cocoa touch新增的多点触摸和手势支持。
- 如何利用iPhone的多语言支持实现应用程序的国际化。
- 介绍iOS对iPhone特有硬件特性的支持，如通讯簿、相机、加速计和GPS。

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604

读者信箱：hzjsj@hzbook.com
华章网站：www.hzbook.com
网上购书：www.china-pub.com

定价：69.00元（附光盘）

第 4 章　Xcode 集成开发环境

Xcode 是苹果公司开发的集成开发环境（IDE），它不仅仅用于 Mac OS 下的编程，也可以用于 iOS 应用程序开发。Xcode 是完全免费的，可以从 Mac OS X 安装光盘中找到它。不过要使用最新的 Xcode，可从苹果公司网站下载最新 SDK，SDK 中已经集成了最新版本的 Xcode。

本章将介绍新版本的 Xcode 4.2（4.2 用于雪豹，4.3 用于 Lion）的诸多特性，例如：让你能快速创建各种程序的项目模板、源代码编辑、智能提示、代码自动完成、集成的上下文帮助、语法错误检测、代码跟踪和调试、性能调优工具，以及一个苹果公司定制的 GCC 编译器，而这些功能只需要点击几下鼠标就能轻易完成。如果没有 Xcode，就需要通过使用命令行来编译一个完整 Cocoa 程序。

4.1　创建第一个 Xcode 应用程序

实际上这并不是本书的第一个 Xcode 应用程序，但在前面第 3 章创建的 Hello World 项目只是一个在命令行下打印简单文本的小程序，总共也没有多少代码，哪怕仅仅使用命令行也能够完成这个工作。最重要是，HelloWorld 没有使用 Cocoa 框架！而 Cocoa 框架是在实际开发中不可缺少的基础。因此，我们决定在这里写一个“真正”的 Xcode 应用程序，以此来了解 Cocoa 框架是如何“组装”出一个应用程序的。

打开 Xcode，从 File 菜单中选择 New Project ...，出现 New Project 向导（参见图 4-1）。

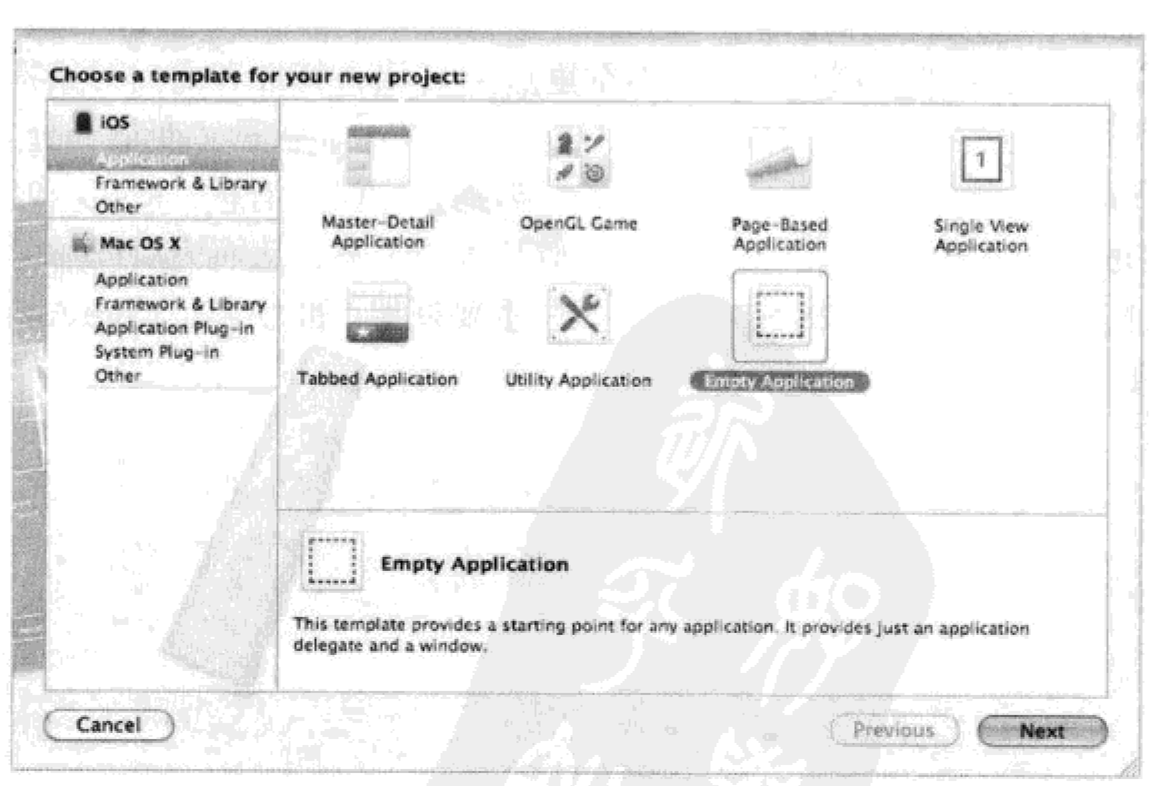


图 4-1　New Project 向导

在 New Project 窗口中的左栏，列出了 Xcode 所支持的所有项目类型，主要分为 Mac OS X 和 iOS 两类，在两个类别下又细分为几个小类，其中每个小类都包含了多个项目模板。我们选择 iOS 下的 Application 中的 Empty Applicate 模板，然后点击 Next 按钮。

此后，将进入项目设置窗口。输入 FirstProject 作为项目名称，其他选项保持反选或默认，点击 Next 按钮。

接下来，选择合适的磁盘位置，然后点击 Create 按钮，新项目将创建完毕，返回 Xcode，Xcode 将自动打开新项目，呈现项目的编辑界面如图 4-2 所示。

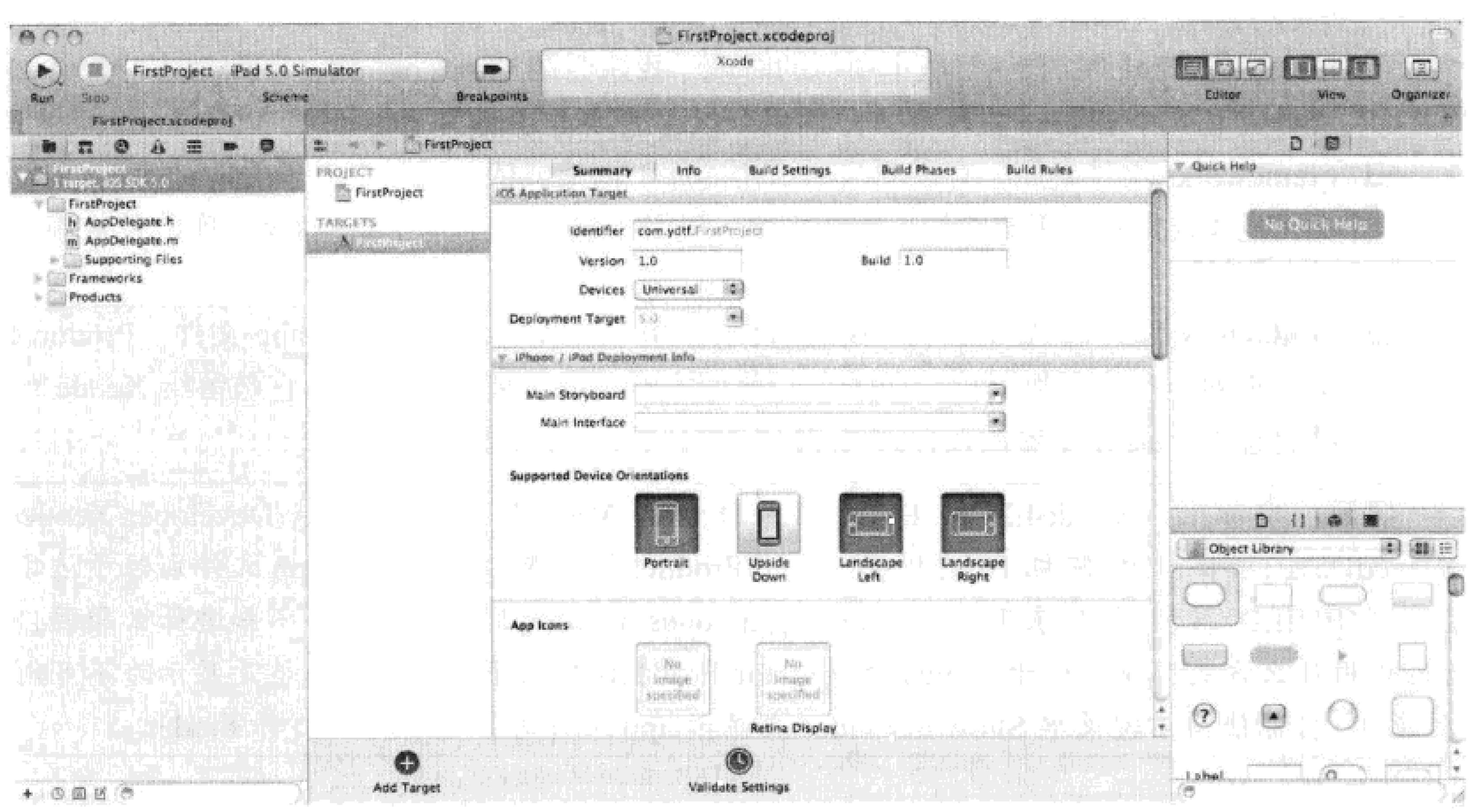


图 4-2　FirstProject 项目窗口

图 4-2 中，整个界面从左至右可分为 3 个部分（子窗口）。

在窗口的左侧是 Project Navigator 窗格，项目中所有源文件、资源都在此窗格中列出。

窗口中间占据大部分区域的是 Editor（编辑器）窗口。当你从 Project Navigator 中选择某个文件，即可在 Editor 窗口中显示对应的编辑界面。根据所选择的文件类型不同，Editor 窗口会自动呈现不同的编辑界面。我们编写源代码主要在这个窗口中进行。

窗口右侧是 Utilities（工具）窗口，根据编辑窗口中你正在编辑的对象的不同，显示不同的编辑工具，类似于 Photoshop 中的工具箱。比如，文件编辑窗口中当前编辑的文件是一个.xib 文件，并且当前编辑对象是一个按钮对象，那么 Utilities 窗口中会显示这个按钮相关属性的编辑界面。在 Utilities 窗口，你可以修改这个按钮的样式、文字和背景图片。如果你正在编辑的对象是一个 Image 对象，则 Utilities 窗口只会显示 Image 的相关属性。在 Utilities 中你只能修改 Image 的图像属性而无法修改文本属性，因为 Image 对象没有文本属性。

由此可见，Xcode4.2 是根据逐渐细化的原则将窗口从左至右划分的。

PDG

查看 Project Navigator 窗口，我们可以发现，项目的文件或资源是按照文件夹的形式列出的。以 FirstProject 项目为例，总共有 3 个文件夹：以项目名称命名的 FirstProject 文件夹、Frameworks 文件夹和 Products 文件夹，下面简单介绍：

- FirstProject 文件夹用于存放项目的源文件，包括：源代码(.h 文件和.m 文件)、图片、.plist 文件和 coredata 文件等。你可以任意组织自己的 FirstProject 文件夹。比如，通常我会在 FirstProject 下面新建一个 Resources 文件夹，然后把所有源代码以外的资源 放在 Resources 文件夹下（比如.png 图片和音频/视频文件）。此外，我也会建一个用于存放视图控制器的 ViewController 文件夹，和专门用于存放公共模块的 Shared 文件夹。在创建 iPhone 和 iPad“二合一”版本项目的时候，我还会在 ViewController 下面创建一个 iPad 文件夹和一个 iPhone 文件夹，分别用于保存适用于 iPad 和 iPhone 的 View Controller。
- Frameworks 文件夹用于存放项目所引用的库、框架或依赖项目。对于 Frameworks 文件夹，我建议你不要更改它们原本的用途，只用于包含项目所用到的框架、库以及依赖项目。
- Products 文件夹顾名思义，即 Build 目录，用于存放项目编译后的 app 文件。Products 文件夹对于程序员则是完全透明的，我们完全没有必要去关心它。它是留给 Xcode 自己管理的，最好不要做任何改变。

提示：实际上，Xcode4.2 的 Build 目录位于“{Mac 用户名}/Library/Developer/Xcode/DerivedData/{项目名称+随机 16 进制数}/Build/Products”目录。编译后的.app 文件则位于该目录的“{BUILD_类型}”目录下，比如 Debug-iphonesimulator 目录。而且，不知道什么原因，无论你用什么方式编译，位于 Project Navigator Products 目录下的.app 文件总是显示为不可用（红色）。当你使用右键菜单 Show in Finder 去显示.app 文件时，总是无法打开 Finder。

4.2 构成应用程序的那些东西

当你新建一个项目，你总是能在项目文件夹中找到某些类型的文件，比如：xxx-Info.plist 和 xxx-Prefix.pch 文件，它们是 Xcode 在创建项目时自动创建的。而另外有一些文件是程序员自己创建的，比如：.png 文件、.xib 文件、.m 文件和.h 文件。这些文件在一起，共同构成了你的应用程序。下面，我们对构成 iOS 应用程序的这些文件分别进行介绍。

4.2.1 Info.plist 和 pch 文件

plist 类型的文件是属性列表文件。Info.plist 文件的全称是“项目名称-Info.plist”，例如 FirstProject-Info.plist（我们可以在 `FirstProject` 文件夹下的 `Supporting Files` 文件夹下找到它），它包含了应用程序相关的信息。一般用于定义应用程序的常规信息，例如定义应用程序图标、程序包名称、语言类型和版本等。其中最重要的是 Bundle Identifier 信息，这在第 1 章关于代码签名的内容中已介绍。

pch 文件是“预编译头文件”，其完整的文件名称应该是“项目名称_Prefix.pch”，例如 FirstProject_Prefix.pch（它和 Info.plist 文件位于同一个地方）。默认，在其中包含了一些最为常用的头文件。Xcode 将对这些被包含的头文件进行预编译，这样当执行 Build 命令时不再需要重复编译这些头文件，从而节省整个项目编译的时间。

4.2.2 Xib 文件

Xib 文件或 nib 文件是 Interface Builder 创建的二进制文件，以.xib 或.nib 为后缀名。它包含使用 Interface Builder 创建的主要由 Cocoa 组件库中组件构成的 GUI 元素。在下一章，我们将介绍 Interface Builder。

注意：如果项目文件夹中存在同名的.xib 文件和.nib 文件，Interface Builder 优先加载.nib 文件。

4.2.3 资源文件

资源文件位于项目目录下，如果你跟我一样，那么我会将所有资源文件放在项目目录下单独的 Resources 目录里。之所以称之为“资源”，是便于与代码区别。资源文件通常是不可编译的，比如：图片、音频和视频、CoreData 数据文件。由于 iOS 安全机制的限制，应用程序只能在“沙盒”中运行，也不能访问“沙盒”之外的资源。因此，需要把应用程序用到的所有资源都保存在此处。

4.2.4 源代码文件

上一章提到，Objective C 源代码分为 interface 和 implementation 两部分。因此，源代码文件包含.h 文件和.m 文件。这些文件都保存在你的 FirstProject 文件夹下，例如 AppDelegate.h 和 AppDelegate.m。

4.2.5 项目和目标

1. 项目和目标的概念

在 Xcode 4.2 中，存在项目（Project）和目标（Target）的概念。在图 4-2 所示的文件编辑窗口（正中间的窗口）中，我们可以看到项目的 Project 文件夹和 Targets 文件夹。在 Project 文件夹下，只有一个对象，即当前项目 FirstProject。在 Targets 文件夹下，也有一个对象，默认也叫做 FirstProject（当然，你可以将它改成其他名字）。这两个 FirstProject 虽然拥有相同的名字，但不是同一个概念。其实仔细看，它们连图标都是不一样的。

最大的不同在于，Project 始终只会有一个，而 Target 却可能有多个。（注意到 Targets 使用了名词的复数形式了吗？）通过下面的 Add Target 按钮，我们可以在 Targets 目录下创建更多的 Target。但无论你添加多少个 Target，我们始终只有一个 Project。

可以认为，Target 代表了一次编译的结果。它把这次编译时所需要包含的资源和代码组织

PDG

在一起，并定义本次编译的编译指令，以便编译器按指定方式完成编译过程。而 Project 就是编译 Target 所需要的源文件及资源的集合。一个 Project 可能包含所有 Target（多个）编译时所需要的源文件及资源。

多个 Targets 有什么好处？其中一个好处就是创建 Universal 项目。Universal 项目的意思是这个项目中同时支持 iPad 和 iPhone 两种版本。我们知道，一个程序的 iPad 版和 iPhone 版的功能应该完全一样，除了显示的界面略有不同外。这样说来，iPad 版和 iPhone 版其实是有可能共用相同类文件（.m 和.h 文件）的，也许它们只会在.xib 上有区别而已——对于同一个 ViewController，可能有一个 iPad 版的.xib 文件，另外还会有一个 iPhone 版的.xib 文件。Universal 项目正是基于这个原理，在同一个项目中每个 ViewController 都会有两个类文件（一个.m 文件，一个.h 文件），同时有两个.xib 文件。我们可以在代码中根据设备的类型来判断该加载哪个版本的.xib 文件，于是 Universal 项目生成的可执行文件其实是可以同时运行于 iPhone 和 iPad 上的。

但这又导致了另外一个问题。编译后的可执行文件会比较大，因为它同时包含了 iPad 和 iPhone 使用的.xib。为了减小 ipa 文件的大小，我们又可以建立两个 Target，分别编译出 iPhone 版和 iPad 版的可执行文件。对于 iPad 版的 Target，我们可以只把类文件（.m 和.h 文件）和 iPad 专用的.xib 文件包含到编译结果中，而 iPhone 版的 Target 的编译结果中，也只包含类文件和 iPhone 专用的.xib 文件。这样，Universal 项目在分为多个 Target 之后，就起到了给可执行文件减肥的效果;同时也保证了 Universal 项目的完整性，是分是合，你可以灵活掌握。

在一个 Xcode 项目中，Project 只有一个，而 target 允许存在多个。它们分别被组织到如图 4-3 所示的 Project 和 Targets 文件夹下。

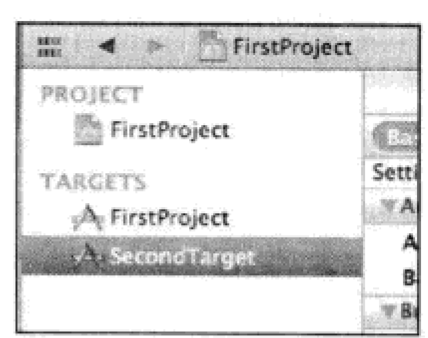


图 4-3　Project 和 Targets

选中 Targets 下的 FirstProject，点击右侧的 Build Phases，可以看到如图 4-4 所示的内容。

图 4-4　查看 Build Phases

- Target Dependencies 下面包含了 FirstProject 这个 Target 编译时所需的依赖项目。当 FirstProject 编译时需要连接其他项目编译的结果时，你可以把这个项目直接引用到项目中，从而形成项目依赖。
- Copy Bundle Resources 下面是该 target 编译时所包含的所有资源文件，后面的数字（1）表示总共有 1 个资源文件：MainWindow.xib。一般而言，这些文件应当是不可被编译器编译的。
- Compile Sources 下面是编译该 Target 时所包含的代码文件，后面的数字（2）表示总共有两个代码文件：main.m 和 FirstProjectAppDelegate.m。一般而言，这些文件应当是可被编译器编译的文件，例如：.m 文件、.c 文件和.cc 文件。
- Link Binary With Libraries 下面是该 target 编译时所包含的库文件，后面的数字（3）表示总共有 3 个框架（库）需要链接，它们是一个 Cocoa Touch 窗口程序最起码的 3 个框架（库）。
- Copy Headers 是编译时需要拷贝的头文件。头文件可以分为 3 种：Private、Public、Project 三种。一般，Copy Headers 只会在某些类型的项目中出现，比如静态库和框架。在通常项目中，不会有 Copy Headers 出现。
- Run Scipt 是编译或运行时额外附加的脚本（Shell）。同 Copy Header 一样，一般项目中不会用到 Run Script。

有时候我们需要在项目中增加新的 Target，例如调试时我们使用 Xcode 默认的 Target 进行编译，但在发布时，我们需要改变编译选项，采用不同的证书签名或者把程序从测试环境迁移到生产环境，但同时我们也需要保留调试时的 target，那么我们需要在项目中加入多个 Target。

2. 加入多个 Target

通过按钮 Add Target，可以让我们新增加一个 Target，这时会要我们选择 Target 的类型，这跟新项目向导中所见到项目模板选择窗口是一样（参考图 4-1）。

一般，我们应当和项目的模板保持一致，选择 Empty Application（除非你打算将项目编译为一种不同的类型，比如编译成静态库）。

接下来是命名 Target，随便取一个名字，比如 SecondTarget，然后点 Finished 完成，你就可以在 Targets 文件夹下看到新建的 Target 了（参考图 4-3 所示）。

随便作一些更改，比如把编译配置（Build Settings）从 Debug 修改为 Release，把 Base SDK 从 iPhone Device 3.1 修改为 iPhone Device 4.0 等。

查看 Project Navigator，你可以发现新增了一个“SecondTarget”文件夹。在 SecondTarget 文件夹下，包含了一份和原来项目一模一样的文件的所有拷贝，除了一份新的名为 SecondTarget-Info.plist 的应用程序配置文件，如图 4-5 所示。

这样新建的 Target，其中的内容几乎和原来的 Target 一模一样。如果你需要在 SecondTarget 中删除一些文件或添加新的文件，直接在 Project Navigator 窗口中操作或在 SecondTarget 的 Build Phases 窗口中操作即可。如果你需要修改 SecondTarget 的编译设置，通过它的 Build Setting

窗口进行。

3. 编译 Target

当项目中使用了多个 Target 的时候，我怎么知道点击 Run 按钮时，当前编译的是 FirstProject 还是 SecondTarget?

我们需要使用工具栏上的 Scheme 按钮。Scheme 按钮实际上被分成了左右两个部分。当你点击 Scheme 按钮的左半部分时，我们可以选择使用哪个 Target 编译，如图 4-6 所示。

选择 SecondTarget 这个 Target，然后点击 Run 按钮。 现在，你可以在 Build 目录下找到 SecondTarget.app 文件。

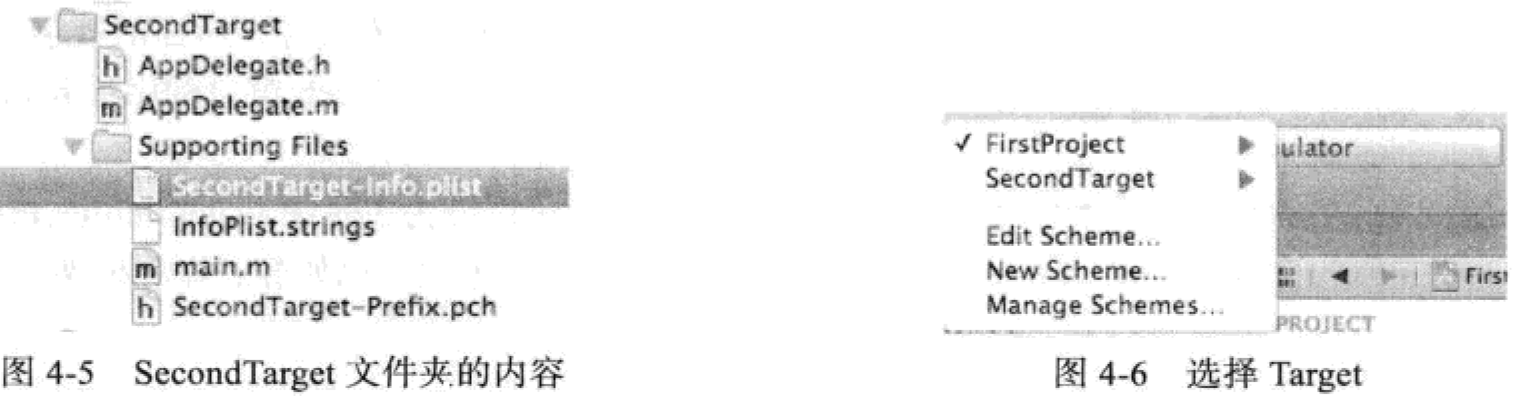


图 4-5 SecondTarget 文件夹的内容　　图 4-6 选择 Target

提示：请通过 Finder 来查看 SecondTarget.app 文件。 正如前面所述，Project Navigator 窗口的 Products 文件夹下的 SecondTarget.app 仍然是不可用的。

4.2.6 Frameworks

在 Groups & Files 的 Frameworks 文件夹中，包含了你通过“Build Phases → Link Binary with Libraries→ Add Items” 方式添加到项目的任何框架和库。但默认情况下，一个 Cocoa Touch 应用程序起码包含了以下 3 个框架（图 4-7）：

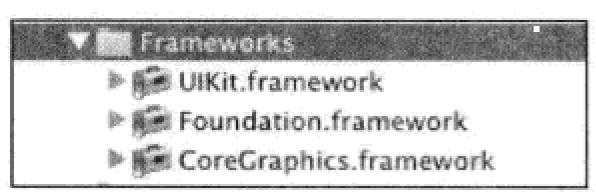


图 4-7 Frameworks 文件夹组

这里需要解释一下 framework（框架）的概念。以下内容来自于苹果文档：

框架是一个包含共享库及相关资源（比如 nib 文件、图像文件、和头文件）的程序包（即一个具有特定结构的目录）。开发应用程序的时候，项目通常会链接一个或多个框架。例如，iPhone 应用程序默认链接 Foundation、UIKit 以及 Core Graphics 三个框架。客户代码通过应用程序编程接口（API）访问框架的功能，而 API 则由框架发布于头文件。由于框架包含的库具有动态共享的性质，即多个应用程序可以同时访问框架代码及资源。因而系统会根据需要将框架代码和资源加载到内存，并让所有应用程序共享一份资源副本（如图 4-8 所示）。

图 4-8 框架的概念

综上所述，框架具备以下特点：

- 与 Cocoa 应用程序类似，框架也是一个具有固定结构的目录（程序包），其中包含代码文件（库文件）和非代码文件（资源）。
- 框架是共享的和动态链接的，内存中只存在一份拷贝，所有应用程序共享。
- Cocoa 程序通过框架的头文件（API)来调用框架。

实际上，我们可以把 framework 理解为类库的概念。所谓 iOS SDK 实际上就是由苹果提供的一系列类库的集合。

此外，在 Mac OS X 平台上，您可以创建自己的框架，但是 iPhone OS 平台则不允许创建第三方框架。如果想创建可以在 iOS 下使用的共享组件，我们必须采用其他方式，这会在第 6 章中介绍。

4.2.7 应用程序的文档目录和临时文件夹

1. iOS 沙盒机制

iOS 系统中“沙盒”的概念，将应用程序对文件系统的访问限制在一个安全的范围内，比如一组最基本的文件夹、网络资源和硬件。从某种意义上说，这就像是加进了一所严格的学校，要遵从极为苛刻的准则。

所有使用 iOS SDK 开发的应用程序，都受到“沙盒”的限制——应用程序只能访问其自身文件夹下的文件系统，而不能访问其他应用程序的文件夹。

“沙盒机制”具备如下规则[1]：

- 应用程序只在自己的沙盒中操作，不能访问其他沙盒。
- 除了通过用户控制的系统粘贴板外，不能共享数据。
- 文件必须位于沙盒提供的文件夹中，并且不能将文件复制到其他应用程序文件夹中，或从其他应用程序文件夹中复制文件。
- 不能读、写沙盒以外的文件，iOS 禁止应用程序将内容写到沙盒外的大多数文件夹中。
- 应用程序拥有自己的 Library、Documents 和/tmp 文件夹，这与其他平台上使用的标准文件夹类似，但对写入和访问数据有限制。

除了这些限制之外，应用程序必须具有数字签名，并且必须通过编码的应用程序标识符向

[1] 请参考《iPhone 开发秘籍》，作者：萨丹。

操作系统证明自己的身份，该标识符需在苹果公司的开发人员计划站点创建。

由于“沙盒规则”的存在，iOS 应用程序是“绿色”的，它不会像 Windows 程序一样产生很多垃圾文件，因为每个程序的所有数据都放在自己的文件夹中。而且，为了进一步防止恶意程序破坏其他程序，iOS 为每个应用程序文件夹起了一个随机名字。例如安装在模拟器中的应用程序都位于：/Users/< 用户名 >/Library/Application Support/iPhoneSimulator/User/Applications 目录下，这个文件夹下有一些采用 uuid 方式命名的子文件夹，每一个子文件夹就是一个已安装的 iOS 应用程序，比如：

“5E4A26E4-B4C4-40D1-B18F-49EF0A342D84”

这样就防止了通过文件夹名字猜测到应用程序所在的位置。而这些子文件夹，就是所谓的应用程序“沙盒”。

2. 访问沙盒数据

根据沙盒规则，每个应用程序都拥有独立的 Library、Documents 和 tmp 文件夹，如图 4-9 所示，各文件夹的简介如下：

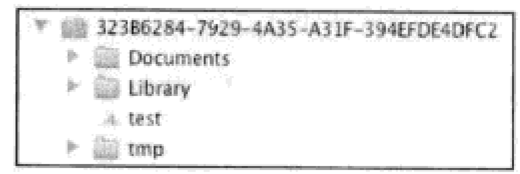


图 4-9 应用程序沙盒

- Library 文件夹用于存储程序的默认设置或其他状态信息，一般不建议将用户数据存储到此文件夹，因此在应用程序中产生的数据一般只使用另外两个文件夹。
- Documents 文件夹是苹果建议的用户文件夹，可以将程序运行中产生的和要访问的数据、文件保存在该目录下。
- tmp 文件夹用于存放临时文件。一些临时使用之后就不再使用的“一次性”数据和文件可以保存在此目录下。所谓“临时文件夹”是指当 iPhone 重启后，该文件夹会被清空。与此相反，Library 和 Documents 文件夹在 iPhone 与 iTunes 同步时会被备份，且不会在 iPhone 重启后被清除。

如果要访问“沙盒”中的文件，我们需要知道应用程序文件夹所在的位置。我们一般采用 NSHomeDirectory()函数：

```
// 访问 Documents 目录
NSString* documents=[NSHomeDirectory()
          stringByAppendingPathComponent:@"Documents"];
// 访问 Documents 目录下的某个文件
NSString  *imagePath = [NSHomeDirectory()
          stringByAppendingPathComponent:@"Documents/someImageName.png"];
```

如果要访问应用程序包（Bundle），则使用以下代码：

```
NSString *bundle=[[NSBundle mainBundle] bundlePath];
```

4.3 了解 Xcode 为我们做了些什么

前面我们说过，当 Xcode 4.2 在为我们创建项目时，总是会自动为我们创建一些文件。除了前面提到的外，还有 main.m、AppDelegate.h、AppDelegate.m。下面我们就来介绍它们的作用。

4.3.1 main.m

打开 FirstProject 的 main.m 源文件，我们可以看到如下代码：

```
#import <UIKit/UIKit.h>

#import "AppDelegate.h"

int main(int argc, char *argv[])
{
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate
                class]));
    }
}
```

首先导入了 Cocoa touch 框架中的 UIKit，这是一个 iPhone 窗口程序所必须具有的。

然后是程序入口 main 函数，跟 C 语言的写法完全一样。其中最重要的是这一句：

```
return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]);
```

UIApplicationMain 方法加载了应用程序主窗口——应用程序的第一个窗口，也就是我们运行 FirstProject 程序时模拟器中显示出来的那个空白窗口。主窗口的作用是让程序不会结束，并使主线程阻塞在主窗口线程中，一直停在主窗口的界面等待用户响应，直到用户关闭主窗口，应用程序才会退出。

你可以试试看，如果注释此句，并将此句改为“return 0;”，应用程序将会闪烁一下，然后直接退出，因为程序没有主窗口。

UIApplicationMain 函数有比较多的参数，前两个不用解释（它们来自 main 函数的参数），第 3 个参数比较重要，它是主类的名字，如果主类名为 nil，则使用 Info.plist 文件中的 NSPrincipalClass 作主类名，如果 NSPrincipalClass 也为 nil，则使用 UIApplication 作主类名。

大部分情况下，我们保持这个参数为 nil，而且也不在 Info.plist 文件中指定 NSPrincipalClass 值，因此都是使用 UIApplication 作主类。UIApplication 是 UIKit 中最为重要的一个类，代表了一个 iOS 程序的主线程（在其中运行了事件循环），一个 iOS 程序只能有一个主线程。

你可以不使用 UIApplication，但你需要继承 UIApplication 并自己实现其中的许多方法。

第 4 个参数指定了应用程序委托类。 应用程序委托是一个实现了 UIApplicationDelegate 协议的对象，这个协议也在 UIApplication 中定义。如果指定了应用程序委托类，主线程的许

多方法就会调用委托类来执行。在 Xcode3.2 版本中，通常是把程序委托类（第 4 个参数）指定为 nil,则 UIApplication 使用 xib 文件(这个 xib 文件的文件名在 Info.plist 的 Main nib file base name 中指定）中的 App Delegate 对象来作为应用程序委托。但在 Xcode4.2 中，则把应用程序委托类设置为 AppDelegate。

4.3.2 应用程序委托

应用程序委托是实现了协议 UIApplicationDelegate 的 NSObject 类。 Xcode 项目模板自动生成的 AppDelegate 就是一个应用程序委托。

一个应用程序委托需要实现一系列以 application 开头的方法（可选），但其中最重要的是以下方法：

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
        (NSDictionary *)launchOptions
```

这个方法在应用程序启动完成后调用。我们可以在这个时候显示程序的主窗口，例如：

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
        (NSDictionary *)launchOptions {
self.window = [[[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]]
        autorelease];

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

在这段代码中，我们实例化了一个 Window 对象并显示该 Window。Cocoa 通过 UIApplication 类，已经为一个 Cocoa 程序的启动做了大量工作，当主线程启动后，我们可以通过 UIApplicationDelegate 进行剩下的工作。因此，编写一个 Cocoa 应用程序，一般从修改应用程序委托 AppDelegate 开始。

4.4 在 Xcode 中添加 View Controller

大部分时候，我们不会直接在应用程序委托类中直接绘制窗口，而是通过继承 UIViewController 类来自定义窗口并显示，这需要我们创建新的类文件。通过点击右键菜单“New File…”，我们可以创建新类（如图 4-10 所示）。

要添加新的类，请选择 Cocoa Touch 中的模板，其中第一个类模板 Objective-C class 表示可以从根类 NSObject 开始创建类，而第二个类模板 UIViewController subclass 表示可以从 UIViewController 类继承。因为我们现在要创建自定义的窗口，所以应选择从 UIViewController 继承。然后点击 Next 按钮。接下来，要求我们为类文件输入一个名字，在此我们输入：MyViewController.m，如图 4-11 所示。

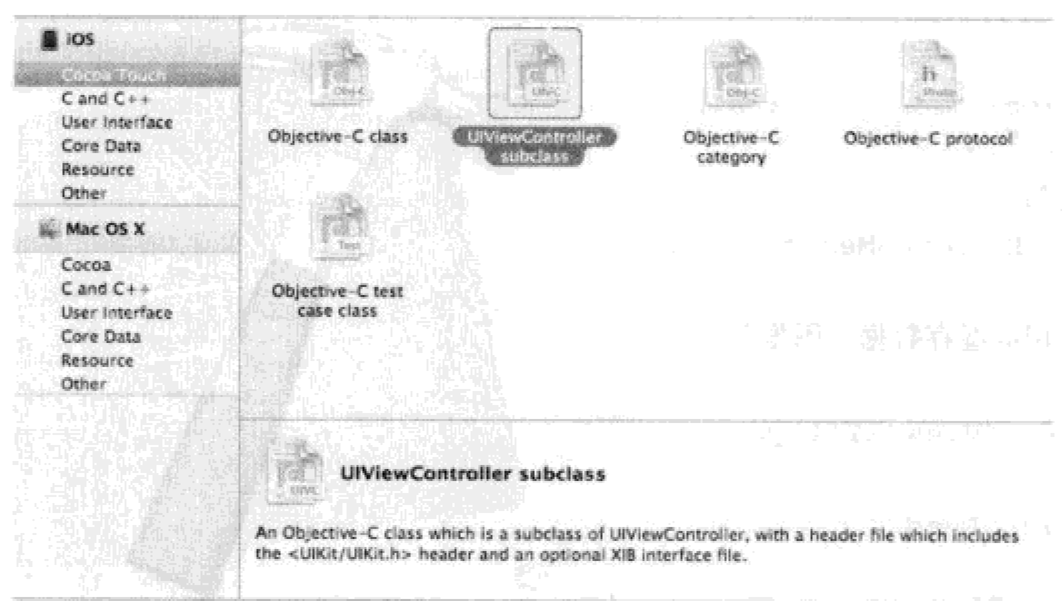


图 4-10　添加新类

图 4-11　命名类

注意勾选“With XIB for user interface”，点击 Next 按钮，然后就是选择文件保存位置。选择好保存路径后，点击 Create 按钮，一个空的 UIViewController 子类创建完成了。

在 Project Navigator 窗口中，我们可以看到新增加的 MyViewController.m、MyViewController.h 和 MyViewController.xib 文件。打开 MyViewController.h 文件，我们看到一个空的 UIViewController 子类定义：

```
#import <UIKit/UIKit.h>
@interface MyViewController : UIViewController
@end
```

切换到 MyViewController.m 文件。

提示：通过快捷键 Ctrl+⌘+↑，我们可以在类的.h 文件和.m 文件之间来回切换。

MyViewController.m 文件内容如下所示：

```
#import "MyViewController.h"
@implementation MyViewController
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        // 自定义初始化
    }
```

```
    return self;
}
- (void)didReceiveMemoryWarning
{
    // 释放视图
    [super didReceiveMemoryWarning];

    // 释放不用的缓存数据、图像等
}
#pragma mark - View lifecycle

- (void)viewDidLoad
{
    [super viewDidLoad];

}


- (void)viewDidUnload
{
    [super viewDidUnload];

}
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interface
  Orientation
{

    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
@end
}
```

可以看到许多方法还没有任何代码。我们不打算改变 MyViewController.m 的任何代码，先来看 MyViewController.xib 文件。MyViewController.xib 的 Editor 窗口界面如图 4-12 所示。

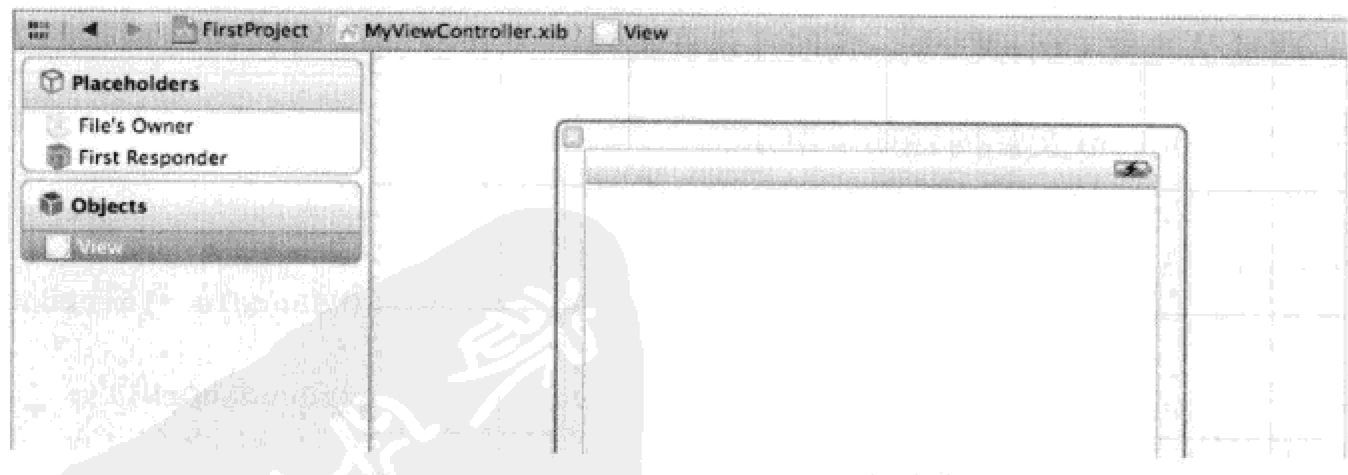


图 4-12 MyViewController.xib 编辑界面

xib 文件是苹果格式的图形界面设计文件（其实是一个 xml 文件，类似于 Android 开发中的 layout XML 文件），它可由 Interface Builder 生成和编辑。在 Xcode4.2 中，Interface Builder 已完全和 Xcode 集成开发环境集成在一起，所以当你在 Xcode4.2 环境中编辑 xib 文件时，Interface Builder 会自动打开该 xib 文件，并在 Xcode4.2 界面中显示 Interface Builder 编辑界面，而不是像早先版本的 Xcode 一样打开一个单独的 Interface Builder 程序。

好了，关于 Interface Builder 的话题就此止步，后面的章节我们会详细介绍 Interface Builder 的使用。现在，让我们先利用 Interface Builder 在 MyViewController.xib 中加入一点东西。

在 MyViewController.xib 编辑界面右下角，应该显示 Object Library 窗口（如果没有显示，你可以通过菜单“View→ Utilities→Show Object Library”来打开它），如图 4-13 所示。

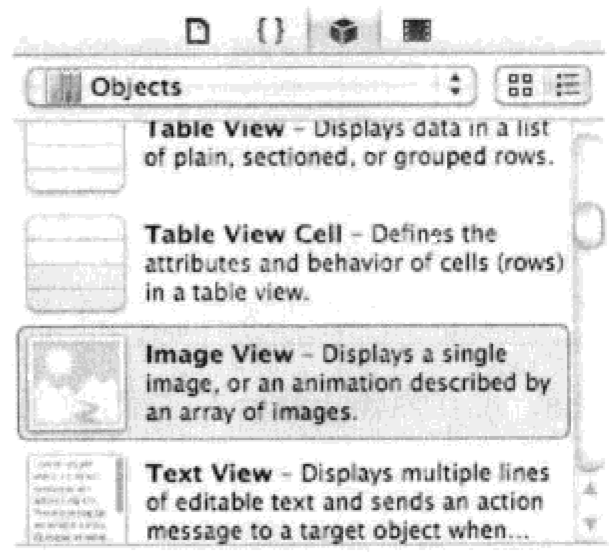


图 4-13 Object Library 窗口

从图 4-13 所示的 Object Library 中找到 Image View（即 UIImageView），将它拖到 MyViewController.xib 编辑界面的 View 视图上（即图 4-12 中由蓝色边框框住的区域），如图 4-14 所示。

然后在 Utilities 窗口的属性面板中（点击 Utilities 窗口的 按钮），可以看到 Image View 对象的属性设置，如图 4-15 所示。

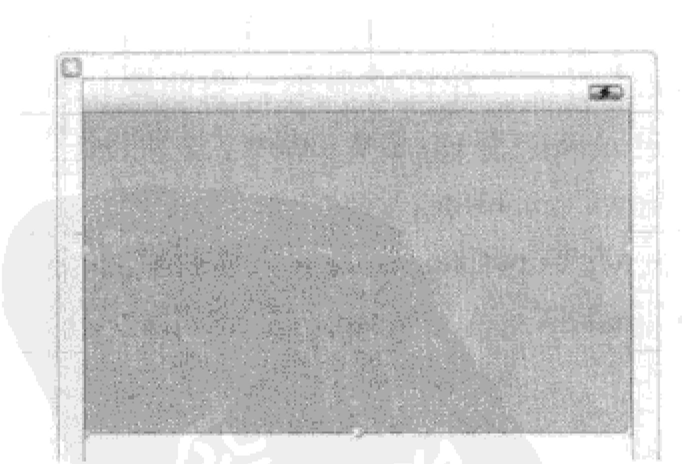


图 4-14 往 MyViewController.xib 的 View 视图中拖入一个 UIImageView

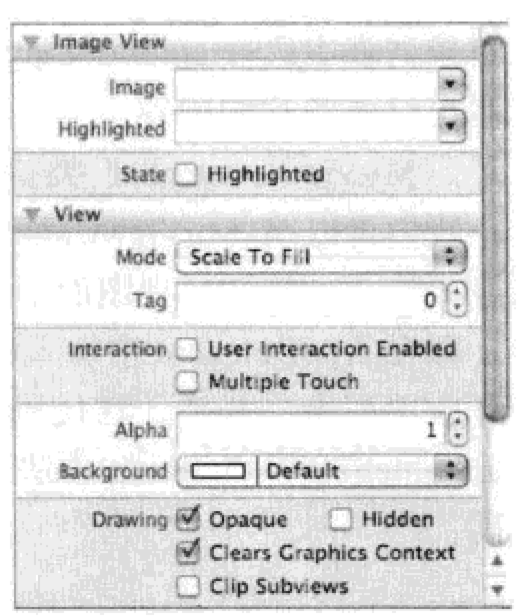


图 4-15 UIImageView 的属性面板

通过 Image View 的属性面板，我们准备修改 Image View 对象的 Image 属性（图 4-15 的第一个属性），以便在我们的 MyViewController 中显示一张照片。在此之前，我们还需要一些准备工作。

打开 Finder，从我们的电脑上找一张图片，然后把它拖到 Xcode 的 Project Navigator 窗口的 FirstProject 文件夹下。如图 4-16 所示，我们拖了一张名为“enter out.jpg”的照片到 FirstProject 中（你可以在 Xcode 中修改它的名字）。

图 4-16　以拖拽的方式向项目中添加文件或资源

提示：由于“安全沙盒”的限制，我们的程序无法访问位于沙盒之外的资源。因此要想在我们的程序中显示一张图片，必需先将它拷贝到项目文件夹中。

返回 Image View 的属性面板，现在点击 Image 栏右侧的下拉按钮，将显示出我们刚才添加到项目的图片文件，如图 4-17 所示。

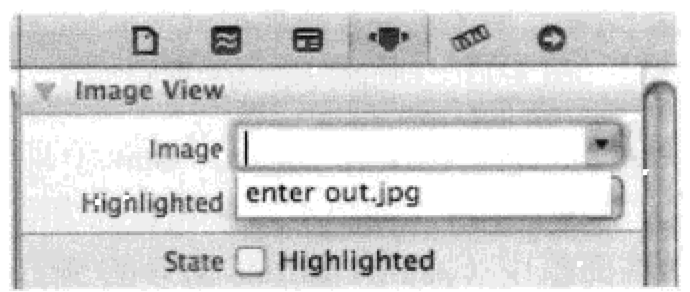


图 4-17　Image View 的属性面板

从下拉列表中选择 enter out.jpg 文件。这时你可以看到 MyViewController.xib 的 View 视图中也显示出这张图片的内容。

好了，MyViewController 类就编辑完了，虽然我们没有写一行代码，但通过 Interface Builder，我们可以在 MyViewController 的视图中加载图片，并自动显示。然而要让程序展示我们的 MyViewController 视图而不是 AppDelegate 中提供的空白的 Window，仍然还差最后一个步骤，即在 AppDelegate 中加入我们定制的 MyViewController。

打开应用程序委托类，在代码头部导入头文件 MyViewController.h，然后在 application:didFinishLaunchingWithOption:方法的“[self.window makeKeyAndVisible];”语句之前加入以下两句代码：

```
MyViewController* viewController=[[MyViewController alloc]init];
[self.window addSubview:viewController.view];
```

点击 Run 按钮运行程序（注意在 Scheme 下拉列表中选择正确的 Target），运行界面如图 4-18 所示。

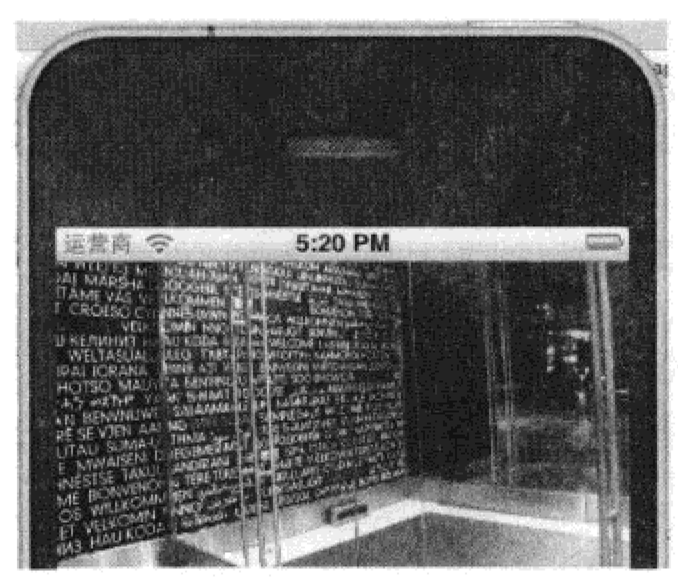


图 4-18　程序运行效果

4.5　在 Xcode 中添加框架

通过 Target 的“Build Phases→Link Binary With Libraries→Add Items”，可在项目中添加除了默认的基础框架以外的其他框架，这些框架分为 3 类：Frameworks、dylibs、Object Files，如图 4-19 所示。

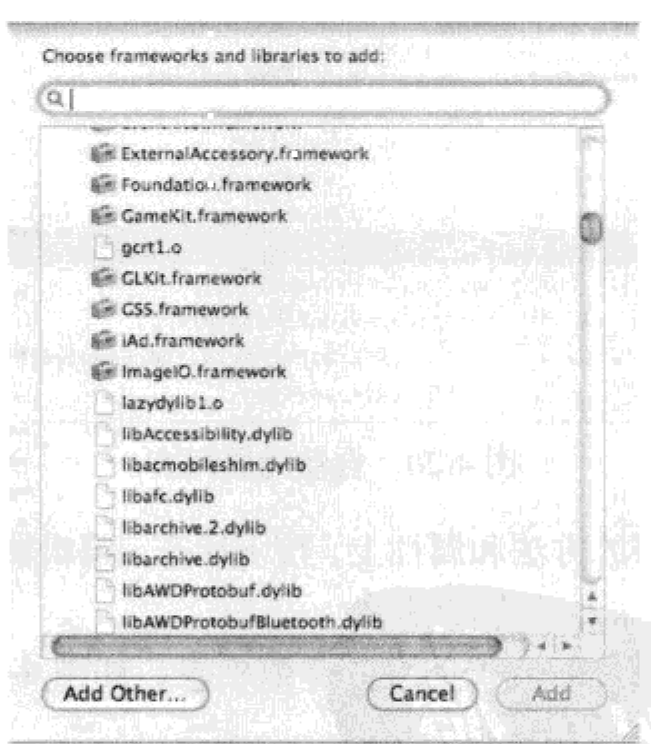


图 4-19　添加框架

Xcode 支持以下 3 种共享代码的方式：

- **框架**。Frameworks 的方式是我们最常用的方式，Cocoa 框架就是以这种方式链接到项目中的。比如 Foundation.framework、UIKit.framework 和 CoreGraphics.framework。

PDG

- 库。dylibs 库是 BSD 风格的动态链接库，类似于 Windows 的 dll 和 Linux 的 so。Mac 是基于 BSD 的，所以也支持 dylibs 库。不管是在 Mac OS X 还是 iOS 中，都有大量的 dylibs 动态库存在。
- **静态库**。静态库一般以 Object File 的形式存在。由于 iOS SDK 不支持第三方动态库，很多时候我们只能以静态库的形式链接这些 C/C++代码。

4.6 Xcode 使用技巧

Editor 窗口是源代码编辑窗口，我们绝大部分时间都是在这个窗口中工作。当我们处于编辑器窗口时，有一些技巧是我们必须注意的。

4.6.1 自动完成

当我们在源代码中输入字符时，Xcode 的自动完成功能随时为我们提供帮助，不管是变量名、方法名还是函数，自动完成列表总能帮助我们轻松解决输入错误。如果列表中列出的内容不是你所想要的，你继续输入即可；如果是的话，可以通过 Tab 键自动完成剩余的部分。

当你键入单词的一部分，然后按 Esc 键，Xcode 都会在光标处显示出自动完成列表。如果不想显示自动完成列表，按“ctrl+.”。如图 4-20 所示，当你在编辑器窗口输入一个不完整的语句，例如“window.adds”，再按下 Esc 键，编辑器将根据上下文自动提示完整的方法或属性列表。

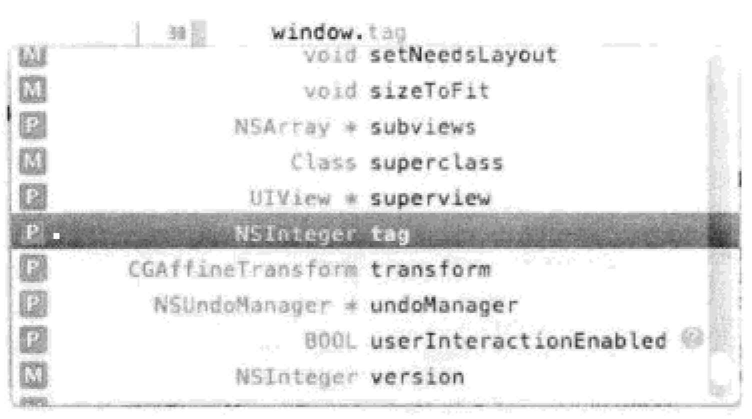


图 4-20 自动完成列表

你可以将光标移动到所需的方法和属性上，按下回车，编辑器将自动把这个方法或属性插入到当前编辑位置。

如果是方法列表，还会提示该方法的完整参数列表，如图 4-21 所示。

```
self.window addSubview:(UIView *)
M void addSubview:(UIView *)
```

图 4-21 方法提示中包含了完整的参数列表

当你回车确认后，该方法会自动补全。每个参数显示为突出显示的方块。当你按 Tab 或者 ctrl+/键，可以移动到下一个参数。

4.6.2　查找和替换

任何时候，你都可以用⌘+F 在当前编辑的源文件内进行查找。如果要进行替换，从下拉列表中选择 Find & Replace。

如果要在项目中查找，使用 Search Navigator 窗口顶部的搜索按钮或者 View→Navigators→Show Search Navigator 菜单，会弹出 Project Find 窗口，里面有许多有用的选项，让你对项目的所有文件进行查找和替换。

如果你仅仅是根据文件名搜索项目中的文件，使用 Project Navigator 底部的搜索栏（图 4-22）。

app

图 4-22　Project Navigator 的文件搜索栏

此外，选择一个变量或参数，然后使用右键菜单 Refractor → Rename ，能够对该变量进行重命名，并且自动更新所有对该变量的引用。如果在某个类上应用 Refractor→Rename 功能，可将该类的.m 文件和.h 文件进行重命名，如果勾选了 Rename Related Files 选项，连引用该类的所有源文件都会被更新。

4.6.3　快速帮助

在某个类名上按住⌘键双击，将快速打开这个类的头文件。如果按住 Option 双击，则可打开联机帮助文档。

4.6.4　快照

快照（Snapshot）将当前项目的状态记录下来，包括所有源代码。使用菜单 File→Create Snapshot（快捷键⌘+Ctrl+S），可以将项目的当前状态保存为一份快照，对于一些无法无天的人来说，这意味着从此可以对源代码进行随心所欲的破坏了。当你需要恢复这份快照时，使用 File→Restore Snapshot 菜单。

提示：快照实际上存储为一个叫做 SnapshotRepository.sparseimage 的磁盘镜像文件中。

4.6.5　书签

在 Xcode3. 2 中，书签可用于标记代码位置。但在 Xcode 4.2 中，这个很实用的功能被移除了。

我们可以用断点（Breakpoint）功能来替代书签。当然为了防止你把它当成真正的断点，应该把断点功能去掉（在断点上点击右键，然后选 Disable Breakpoint）。这样你可以通过断点导航器（Breakpoint Navigator）来浏览所有“书签”（实际上是 disabled 掉的断点）。

PDG

提示：断点导航器使用按钮或者菜单 View→Navigators→Show Breakpoint Navigator 来显示。

4.6.6 使用导航条

在代码 Editor 窗口的顶部有一个小工具栏，如图 4-23 所示。

图 4-23 Editor 窗口顶部的导航条

首先是一个按钮，这是 Related Files 按钮，提供了一些和当前编辑的文件相关的文件列表，如：最近文件（Recent Files）、未存文件（Unsaved Files）、Counterparts（.h 文件/.m 文件）、父类（Superclasses）、包含的文件（Includes）等。

接着是前进、后退按钮，可用于快速地在最近编辑过的文件中前进和倒退，等同于浏览器中的前进、后退按钮。

接下来是一个 4 级下拉列表，用于显示当前编辑位置。你可以把它看成是另一个 Project Navigator，用于在项目文件中导航。它的最后一级是方法属性列表，用于类结构中的导航，默认显示当前编辑在类结构中的位置。

如果在源代码中使用#pragma mark 标记，可以将一些可读标记放入到功能列表中，便于查看。此外，注释中以 MARK:、TODO:、FIXME:、!!!:和???:开头的文本也有同样的作用。

在源代码中插入断点，可以单击代码所在的行号。要想 disabled 或删除断点，可以在断点上点击右键，再使用相应的功能菜单。

4.7 本章小结

本章介绍了苹果公司的 Xcode 集成开发环境，包括一些最常用的操作或技巧：如何创建项目、如何快速地编辑代码和获取帮助等。

Xcode 是 iOS 程序员最主要的开发工具。当然，要想熟练地掌握这个有力的武器，技巧并不是唯一重要的东西，尽量多地使用它才是正途。在后续章节的大量练习中，我们将会逐步学习和熟悉这一工具。

下一章，将介绍 Xcode 集成的 Interface Builder 工具。

第 5 章 Interface Builder

在 Xcode 4.0 之前，Interface Builder 和 Xcode 一直是两个分开的应用程序。而在 iOS 开发中，往往需要同时使用 Interface Builder 和 Xcode 两种工具来配合。

Interface Builder 用于创建应用程序的图形界面，Xcode 用于创建代码，二者一直是相辅相成的。因此在介绍完 Xcode 4.2 集成开发环境之后，我们用单独一章来介绍 Interface Builder。

Interface Builder（以下简称 IB）与 Xcode 一样，也是苹果公司开发的运行在 Mac OS 下的 IDE，是专用于设计 Mac 和 iOS 应用程序的 GUI。虽然也可用源代码方式直接生成 GUI，但 IB 的出现极大地简化了这一工作。IB 随 SDK 一同发布，当安装完 Xcode 的时候，IB 已经随 Xcode 一起安装到你的 Mac 上了。

IB 所创建的 GUI 保存为 xib/nib 文件格式，当你在 Xcode 中编辑一个 xib/nib 文件时，IB 就会自动打开该文件。IB 为 Cocoa 开发者提供了包含一系列 GUI 元素的工具箱，称为 Cocoa Library，这些元素包括文本框、表格、滚动条、工具栏按钮等控件。并且，开发者可以通过 IB Kit 扩展 Cocoa Library（仅限于 Mac OS）。通过 IB，开发者只需要从工具箱中简单地拖出控件即可完成界面的设计。然后，通过一种所谓的“连接”方式，将控件或组件与 Xcode 代码相连。

本章将详细介绍如何使用 IB 创建图形用户界面，以及如何将这些界面元素和 Xcode 代码相连，并创建出能与用户交互的应用程序。

5.1 IB 和 xib、nib 文件

在第 4 章中，我们曾介绍过 MyViewController.xib 中所包含的对象 View 视图，这个对象由 IB 创建、实例化、加载并自动维护。如果我们要查看和编辑这些对象，只能通过 IB。IB 把 GUI 资源存储为 nib（或 xib）文件，在 Cocoa 框架中已为快速加载 nib 文件进行了优化。

IB 支持旧式的.nib 文件和较新的.xib 格式文件。在 IB 3 之后，开始使用.xib 文件。但对于 iPhone 项目而言，则只使用.xib 格式的文件。由于历史的原因，.nib 文件和.xib 文件都被称为“nib 文件”，但实际上该文件可能是以.xib 为扩展名的。在最新的 IB 程序中，默认的 nib 文件实际上是.xib 格式的，但仍然支持旧式的.nib 格式。

.xib 文件和.nib 文件本质上没有任何区别。只不过.xib 文件是 xml 格式，而.nib 文件是二进制格式；.xib 文件编译后仍然会产生一个二进制格式的.nib 文件。

由于 IB 只能用于创建图形界面，nib 文件中也不能包含 Objective C 代码，因此实际上 IB 并不能单独工作，程序的代码逻辑仍然需要在 Xcode 中编辑。因此 IB 是集成在 Xcode 中的。

PDG

5.2 初识 IB

打开 FirstProject 项目（见 4.1 节“创建第一个 Xcode 应用程序”）。在 Project Navigator 中找到 MyViewController.xib 并单击，将在编辑器窗口打开 IB 界面。由于在 Xcode 4.2 中，IB 已完全集成到 Xcode 的 IDE 中，所以整个 IB 将整合到 Xcode IDE 的 Editor 窗口和 Utilities 窗口来显示，如图 5-1 所示。

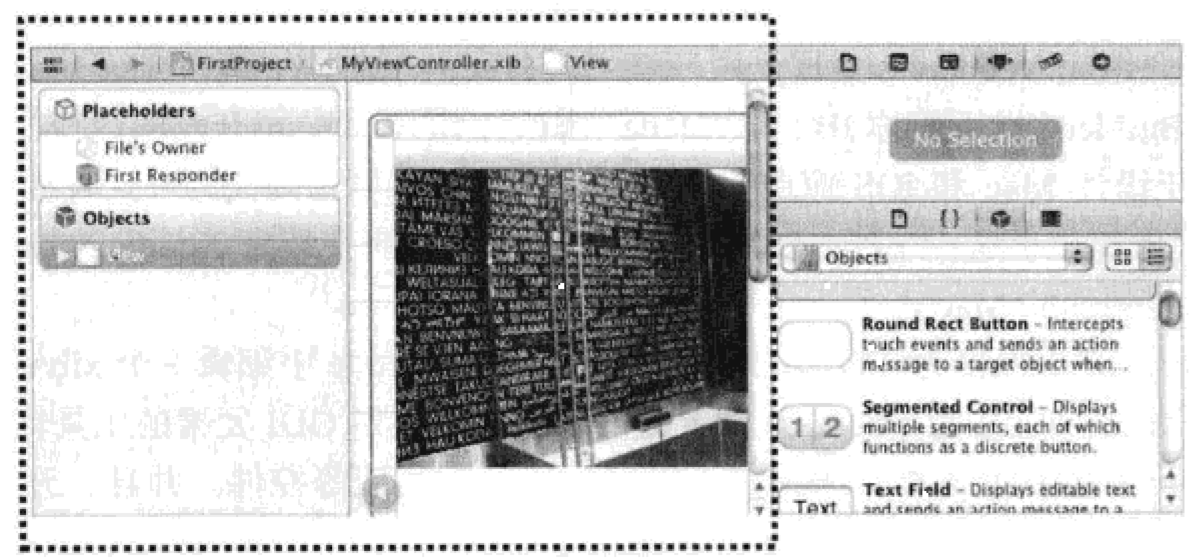


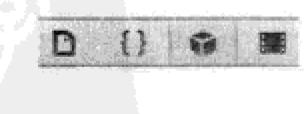
图 5-1 xib 文件的编辑界面

图 5-1 中虚线框区域为 Editor 窗口，Editor 窗口分为左右两个区域：IB 对象导航视图和 xib 预览窗口。左窗口为 IB 对象导航视图，显示 xib 文件中包含的所有 IB 对象，包括 File's Owner、First Responder 和 View 等。而 View 对象往往呈现了 View Controller 的视图层次（树）。通过 IB 对象导航视图，我们可以在 xib 文件中所包含的 IB 对象间导航，并选择所要编辑的对象。

xib 预览窗口以图形化的方式展现 xib 中的可视对象在 iPhone/iPad 中的样子。xib 预览窗口同时也是 xib 的设计窗口，我们可以直接在这个窗口中添加、删除 IB 对象，创建连接，修改对象的位置大小、外观样式和部分属性。当然，有的属性必须通过 Inspectors （面板区，后面会介绍）中才可编辑。

Editor 窗口右边是 Utilities 窗口，Utilities 窗口又分为上下两个区域：Inspectors（面板）和 Library（库）区域。在 Inspectors 区域顶部是一个工具条，依次摆放了 File Inspector、Quick Help Inspector、Identity Inspector、Attributes Inspector、Size Inspector、Connections Inspector 六个按钮，分别用于呈现不同的面板视图：

Library 区域顶部也有一个工具条，依次摆放了 File Template Library、Code Snippet Library、Object Library、Media Library 四个按钮，分别用于呈现不同的 Library 视图。在本书，我们只会使用到 Object Library，即第三个按钮：



Inspector 和 Library 都可以通过 View → Utilities 菜单打开。对于这些 Inspector 和 Library，我们并不需要进行过多的了解。在本书后面，将会不时地提到和使用到这些 Inspector 和 Library。在此，你只需要了解它们的名字，及其在工具条中所对应的按钮就可以了。

5.3 使用 IB 创建图形界面

IB 用于创建 iOS 应用程序的图形界面，准确地说，是用于创建二进制格式的 nib/xib 文件。而这些文件可以用于在实例化 UIViewController 和 UIView 对象时提供图形化界面。 接下来，我们看看如何使用 IB 来创建图形界面。

5.3.1 控制器和视图

1. UIViewController 和 UIView

UIViewController 对应了 MVC 框架中的控制器层。在第 4 章，我们曾经从 UIViewController 派生了一个子类 MyViewController。UIViewController 有一个默认的 view 成员， 这个 view 成员是一个 UIView 对象，我们可以通过这个 view 对象来显示（定制）自己的视图。视图控制器（UIViewController）正是通过这种方式来管理它的视图(UIView)的。

此外，视图控制器控制了视图的旋转。iPhone 的一个重要特性是加速计和重力感应。当 iPhone 感知到设备方向发生变化时，iPhone 会通知应用程序，因此应用程序可做出相应的反应，比如旋转视图的方向，使之和设备方向保持一致。视图本身不能旋转，要使视图旋转，必须利用视图控制器的能力。

因此，这里的视图控制器（UIViewController）和视图（UIView）就分别对应了 MVC 中的控制器（C）和视图（V）的角色。

在一个典型的 MVC 应用程序中，架构可能是这样的：

一个模型（M），代表来自服务器或数据库的数据，这些数据可能是 XML，也可能是关系型数据，这不重要，重要的是数据本身是为存储服务的，而不是面向用户的。这就导致原始的数据非常不容易被理解。当用户看到数据中有一个字段 gravity，他可能会理解为这是“权重”的意思，而实际上却可能指的是“重力加速度”。因此我们的程序永远不可能只是由模型（数据）构成，除非我们只打算自己使用。为了把数据展示为用户友好的视图，我们的应用程序起码还需要一个视图（V）。视图是必需的，因为这样我们才有可能将模型中的 gravity 展示为正确的意义。在程序界面很简单的情况下，比如程序只有一个视图（窗口），我们可以构建一种基于窗口的应用程序，因为程序不需要导航（只有一个视图）。但是 iPhone 的屏幕实在是太小了，一个 320×480 的屏幕一次性展现出来的数据又能有多少呢？大部分情况下，我们必须使用多视图。这时，因为视图本身无法导航，要在多视图间进行切换，则必须使用控制器，即 UIViewController。UIViewController 这个名字本身已经说明了控制器存在的意义，它是用于管理视图 UIView 的。而实际上，每个 UIViewController 及其子类，都包含了一个默认的成员 view。

你可以把任意类型的 UIView 实例赋值给该 view 成员，这样当 UIViewController 被弹出导航控制器时，将呈现这个 UIView。一个 UIViewController 只能有一个 view，但 UIView 是一个树型集合，view 又可以有任意多和任意层次的 subview，这些 subview 都可以是任何 UIView 和 UIView 的子类，比如各种可视的 GUI 元素。通过调用视图的 addSubview:方法，我们可以把一个控件或 UIView 添加为视图的 subview。

Cocoa 建议程序员使用 UIViewController 构建应用程序。iOS 企业应用也是基于视图控制器的，这已经成为本书的一个基本准则。一个 UIViewController 包含一个 UIView，而一个 UIView 呈现了一个窗口界面，这是通常的 Cocoa 式编程体验。但是在某种情况下，我们会在一个 UIViewController 中集成两个甚至多个窗口界面，而每个窗口又是用一个 UIView 组织起来的，这样就同时存在多个 UIView。当然最终这些 UIView 都必须用 addSubview 加到 view 成员中。控制器控制了什么时候呈现这些 UIView，以及视图切换时使用的动画效果。当然，如果使用 UIViewController 控制屏幕的旋转，一切都显得简单。

提示：如果你正在写一个支持旋屏的应用程序，那么最好不要使用 UIView 或 addSubview 方式的导航，而尽量使用 UIViewController 来进行多个界面间的导航。否则会有一些问题。

2. 视图控制器的种类

UIViewController 继承于根类 NSObject。这意味着，它不具备任何 UIResponder 和 UIView 的功能。因此，虽然 UIViewController 和 UIView 看起来很像，而且它确实也包含了一个 UIView，但它仍然不能代替 UIView，比如我们需要定制一个可重用的 UI 组件时，就应当使用 UIView 而不是 UIViewController。

UIViewController 是一种使用很广泛但同时也是最简单、最基本的视图控制器，在它之下，还有各种各样的用于不同目的的视图控制器。我们一般极少在应用程序中直接使用 UIViewController，而是使用它的子类（包括我们自己扩展的从 UIViewController 继承的子类）：

- UITableViewController
- UITabBarController
- UINavigationController
- ABPeoplePickerNavigationController
- ABNewPersonViewController
- ABPersonViewController
- ABUnknownPersonViewController
- UIImagePickerController

除了 AB 开头的几个与地址簿相关的不太常用的视图控制器之外，其他的视图控制器我们都会在后续章节（第 6 章）中陆续介绍。

实际上，去除几个 AB 开头的不常见的控制器后，iOS SDK 中提供的控制器种类并不能完全满足我们的需要。在本书中，更多时候，我们都是通过子类化 UIViewController 的方式，定

制所需要的视图控制器。注意，是 UIViewController 的子类，而不是 UIViewController 的子类的子类（孙子类）——因为导航控制器（UINavigationController）不能识别 UIViewController 的孙子类。

3. 使用 UIViewController

新建 Empty Application 项目：UsingViewController。新建 UIViewController subclass ，使用“With XIB...”选项，命名为 RootViewController。

提示：UsingViewController 项目位于光盘“source/第 5 章”目录下。

编辑 RootViewController.xib。从 Object Library 中找到 Round Rect Button（圆角按钮）。

技巧：可以使用 Object Library 底部的搜索栏，在搜索栏中输入 Button 这个单词的开头几个字母，Object Library 会很快定位到 Round Rect Button 上。

将 Round Rect Button 从 Object Library 拖到 xib 预览窗口的中央，如图 5-2 所示。

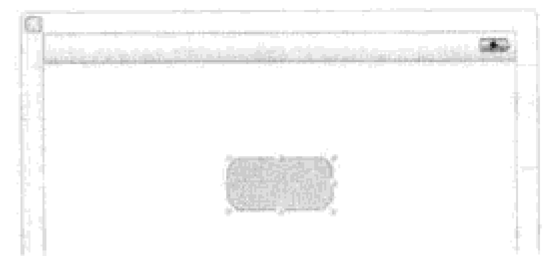


图 5-2 往 RootViewController 中添加按钮

保持 xib 预览窗口中的按钮处于选中状态（呈浅蓝色并在四周显示 6 个拉柄）。打开 Attributes Inspector 面板，并找到 Title 属性，将它改为“通讯录”。

提示：还记得 Attributes Inspector 按钮在 Inspectors 区工具条上的位置吗？从左至右第 4 个。你也可以使用 alt+⌘+4 快捷键来打开 Attributes Inspector。

现在，我们需要编辑 AppDelegate.m，在它的文件头部添加一行代码；

```
#import "RootViewController.h",
```

然后在代码 [self.window makeKeyAndVisible];之前加入一行代码：

```
self.window.rootViewController=[[RootViewController alloc]init];
```

再次在项目中添加 UIViewController subclass(勾选“With XIB...”选项），命名为 DetailViewController，同时 Subclass of 选择为 UITableViewController 而不是默认的 UIViewController。

打开 RootViewController.h，在代码顶部导入头文件 DetailViewController.h 和 AppDelegate.h。然后声明一个方法：

```
-(IBAction)goDetailAction;
```

提示：关键字 IBAction 表示一个 Action 出口，Action 出口可以连接到 IB 对象，后面将会具体介绍，在此我们可以将它视作等同于 void。

然后打开 RootViewController.m，实现这个方法：

```
-(IBAction)goDetailAction{
    AppDelegate * app=(AppDelegate*)[[UIApplication sharedApplication]delegate];
    app.window.rootViewController=[[DetailViewController alloc]init];
}
```

在代码第一句，获取了 AppDelegate 对象，即应用程序委托对象。每个应用程序在运行时都会有一个唯一的全局应用程序对象(UIApplication)，它又有一个 delegate 属性，指向了应用程序唯一的全局应用程序代理对象（UIApplicationDelegate，AppDelegate 继承了 UIApplication Delegate）。

第二句，我们将这个 AppDelegate 的 window 对象的 rootViewController 属性修改为一个 DetailViewController 实例。我们知道最初这个 rootViewController 被设置为 RootViewController 实例，在改为 DetailViewController 实例后，程序窗口中显示的视图自然就从 RootViewController 的视图切换到 DetailViewController 视图了。通过这两句最简单的代码，我们完成了从 RootViewController 到 DetailViewController 的导航。

接下来我们需要在 RootViewController.xib 中将按钮连接到这个 IBAction 方法。打开 RootViewController，从 xib 预览窗口，用右键从按钮上拖一条线到 File's Owner (位于 IB 对象导航视图) 上，并在弹出的列表中选择 goDetailAction，如图 5-3 所示。

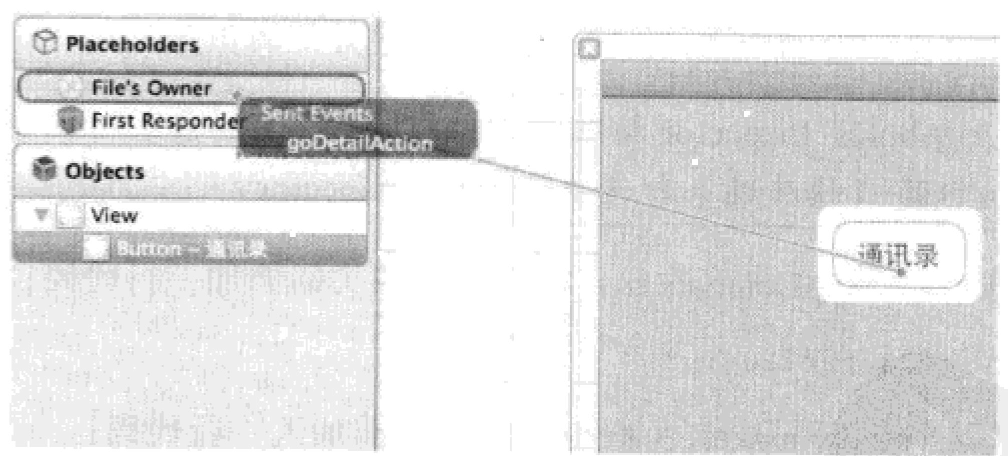


图 5-3　为按钮建立连接到方法 goDetailAction 的连接

这里，我们就建立了一个从按钮对象到 goDetailAction 的连接。当程序运行时，点击“通讯录”按钮，将自动触发 goDetailAction 方法。关于连接，后面我们还会详细介绍。

在 Scheme 中选择 Device 为 iPhone 5.0 Simulator，然后点击 Run 按钮，即可在 iPhone 模拟器中运行这个程序。当点击窗口中的“通讯录”按钮，窗口将导航到一个表格视图（UITableViewController 视图），如图 5-4 所示。

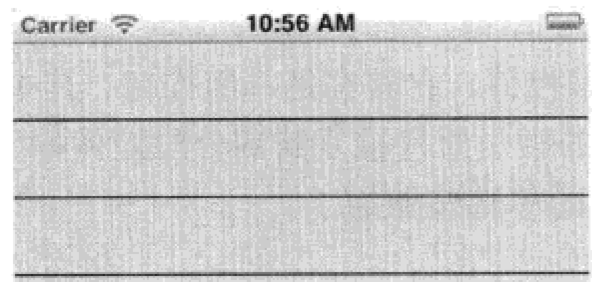


图 5-4 UITableViewController 视图

5.3.2 基本控件介绍

这些基本控件都继承自 UIControl→UIView→UIResponder→NSObject。它们的共同特征是在屏幕上可视，占据了屏幕的一个矩形区域，并接收用户事件。本章简单介绍一下这些控件，然后在后续实战部分再逐渐熟悉它们的使用。

1. UILabel

UILabel 是最简单的可视化控件，仅用于显示同一颜色、字体和大小的只读文本。在 UsingViewController 中你已经使用过 UILabel 了，可以修改 UILabel 的显示文本、字体、大小、颜色、位置、背景色等，使用 Attributes Inspector 很轻易就能做到。

2. UITextField

UITextField 的作用是提供一个文本编辑框。它比 UILabel 复杂一些，允许用户对文本内容进行编辑。除了和 UILabel 相同的一些属性外，它还有很多重要的属性，比如边框类型，默认的文本框是圆角矩形边框，可以设为另外 3 种：下凹型、上凸型和无边框。文本对齐属性允许文本左对齐、右对齐和居中。Keyboard 属性可为文本框指定一种适当的软键盘，因为 iPhone 屏幕大小限制的原因，无法为用户提供一个全按键的软键盘，你需要仔细考虑输入的用途——用于输入字母、数字还是 Email 地址，以选择适当的软键盘。

此外还有一个不容易发现的 Secure 属性，这是一个 BOOL 属性，如果选中它则将文本框转变为密码输入框。

3. UIImageView

UIImageView 用于显示位图，它最重要的属性是 image 属性，但我们却无法直接把一张图片文件名输入到这个属性，因为它的类型是 UIImage。我们唯一能做的是，把图片文件复制到项目的 Resources 文件组中，然后在 image 属性中选择图片文件的名字。

4. UISlider

UISlider 显示一个滑块控件，用户可通过拖动滑块的位置来改变控件的值。它有 3 个重要属性：value、minimumValue 和 maximumValue。value 属性表示滑块当前位置所代表的值，minimumValue 属性表示滑块能移动到的最小值，maximumValue 属性表示滑块能移动到的最大值。

PDG

5. UISegmentedControl

UISegmentedControl 是分段控件，它其实是一组按钮，其中每个按钮都有一个索引值，当某个按钮被按下时，这个按钮的索引值将被复制到分段控件的 selectedSegmentIndex 属性。它的作用类似于 windows 控件中的单选按钮组。

6. UIButton

UIButton 控件显示了一个普通按钮，专门用于响应用户的点击动作。它有一个 type 属性，可以指定按钮的不同风格：圆角矩形按钮、加亮的 info 按钮、加暗的 info 按钮、联系人添加按钮、自定义风格的按钮。按钮最主要的两个属性是 Image 和 Title ，可以为按钮指定一个背景图片和标题。

按钮最有用的是 Connections Inspector 中的 Sent Events 事件列表，可以利用这个事件列表创建一系列的 IBAction 连接。IBAction 连接在后面介绍。

7. UIToolbar 和 UIBarButtonItem

UIToolbar 和 UIBarButtonItem 分别是工具栏和工具栏按钮。工具栏是工具栏按钮的容器，必须先创建工具栏，才能在上面创建工具栏按钮。工具栏具有很少的属性可以设置，比如 style 属性，可以指定工具栏的风格：默认、黑色透明、黑色半透明。

工具栏按钮与按钮有着类似的属性：style、image、title。此外，它的使用也和 UIButton 差不多。

5.4 连接

前面我们已经使用过 IBAction 连接了（在 UsingViewController 项目中）。在 Xcode 中，除了 IBAction 连接，还有 IBOutlet 连接和代理连接。下面我们将对 Xcode 中种类丰富的连接进行介绍。

5.4.1 IBOutlet 连接

如果要在 Xcode 中（不是在 IB 中）访问、改变和操纵 xib 对象的各种属性，需要使用 IBOutlet 连接。要建立 IBOutlet 连接，需要在代码中使用 IBOutlet 关键字指定属性或成员变量。创建一个 IBOutlet 连接需要经过以下步骤。

1. 声明 IBOutlet 变量和 IBOutlet 属性

首先，声明一个和连接对象相同类型的成员变量或属性。例如，在 RootViewController 类的成员声明中，声明一个成员为 IBOutlet，可以使用如下语句：

```
IBOutlet UILabel* lbTitle;
```

如果想以属性的形式声明这个 IBOutlet，则使用以下语句：

```
@property(retain,nonatomic) IBOulet UILabel* lbTitle;
```

PDG

提示：IBOutlet 变量和 IBOutlet 属性实际上只有少许的不同，前者在类的内部可见，后者在类的外部可见。使用哪一种，全凭你的喜好。

如果声明 IBOutlet 属性，则别忘了 Synthesize 这个属性：

```
@synthesize lbTitle;
```

如果是 IBOutlet 变量，则不需要 Synthesize。

可以看到，除了使用了 IBOutlet 关键字以外，它们和一般的声明语句没有什么不同。此外，IBOutlet 变量或 IBOutlet 属性仅仅需要声明，它们不需要初始化，也不需要释放，只需要在 ViewDidUnload 的时候设置为 nil：

```
lbTitle=nil;
```

因为它们其实是连接到一个 xib 的内部对象，而这个 xib 对象是由 IB 负责管理（实例化并释放）。我们声明的 IBOutlet 变量或属性仅仅是一个符号而已，不需要负责它的内存释放。

2. 在 IB 中连接对象

在此之前，我们首先来了解一下 File's Owner 的概念。

在 IB 的"对象导航面板"中，你永远会在顶部找到一个 File's Owner 的对象。File's Owner 是指加载这个 xib 文件的对象。我无法找到任何简洁的词语来翻译这个字眼，如果可以的话，你可以把它叫做"加载该 xib 文件的 UIViewController 或 UIView 实例对象"。但这个名字太长了，我们还是仍然使用 File's Owner 好了。

一般情况下，UIViewController 在实例化时，IB 会自动加载 initWithNibName 方法中指定的 xib 文件。如果 UIViewController 是使用默认的 init 方法初始化，则会加载和类同名的 xib 文件。例如，当使用[[RootViewController alloc]init];方法初始化一个 RootViewController 实例时，IB 会同时加载 RootViewController.xib 文件。

IB 在加载 xib 文件时会进行连接，即将存在连接关系的 IB 对象和 File's Owner 的 IBOulet 或 IBAction 关联。同时，File's Owner 会指向此 UIViewController 实例。在 IB 的对象导航窗口中选择 File's Owner 对象，用 alt+⌘+3 打开它的 Identity Inspector 面板，我们可以查看 File's Owner 的 Class 属性，该属性标明了 File's Owner 对象的类别。

我们可以在 File's Owner 的 Connections Inspector 中看到我们声明的所有 IBOutlet。以 RootViewController.xib 为例，File's Owner 的 Connections Inspector 面板如图 5-5 所示。

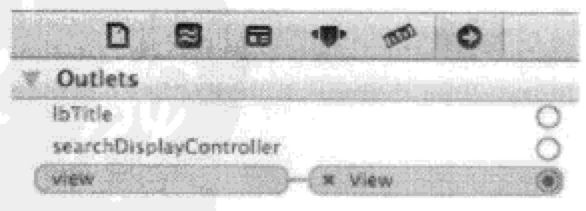


图 5-5　查看 File's Owner 的出口

在 Outlets 下，列出了 File's Owner 的所有出口（即 IBOutlet 变量和属性）。可以看出，这

些 Outlets 都是 RootViewController 类所定义的。其中 lbTitle 是我们刚才声明的。searchDisplayController 和 view 是 UIViewController 类自带的两个出口。在这个例子（RootViewController.xib）中，File's Owner 就是一个 RootViewController 类或者 RootViewController 实例。

接下来，我们在 RootViewController 中添加一个 Label 对象，然后从 File's Owner 用右键（或 ctrl+左键）拖一条线到 Label 对象，然后从弹出的列表中选择“lbTitle”。这样就完成了从 IB 对象 Label 到 IBOutlet 属性 lbTitle 的连接，如图 5-6 所示。

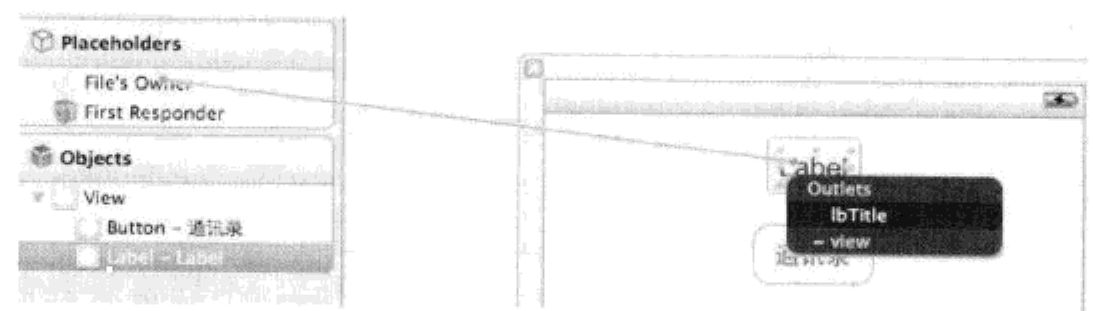


图 5-6　建立一个 IBOutlet 连接

建立 IBOutlet 连接之后，我们就可以在代码中使用和操作该 IB 对象了。可在 RootViewController 的 viewDidLoad 方法中加入这样一句代码：

```
lbTitle.text=@"进入通讯录";
```

在 viewDidLoad 方法中，我们使用了 lbTitle 对象并修改了它的 text 属性，如同在使用自己创建的对象。但实际上，这个对象是由 IB 创建和生成的，我们没有写任何一个初始化该对象的语句。使用 IBOutlet 连接对象的好处是在于，有的对象，比如一个复杂的 UI 界面，用代码创建十分繁琐；而采用 IB 创建则相对简单，不用你写一句代码。我们可以先用 IB 设计和创建复杂的 UI 元素，然后将其与 IBOutlet 进行连接，这样就可以直接在 Xcode 中使用该 IB 对象了。

运行程序后效果如图 5-7 所示。

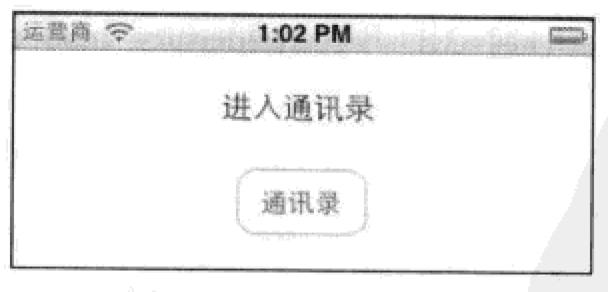


图 5-7　程序运行效果

5.4.2　IBAction 连接

在学习 IB 时我们发现，不可能让一切事情都在 IB 中完成。比如，我们无法在 IB 中编写代码。我们可以从 Library 中拖一个对象到.xib 文件中，但无法拖哪怕一句代码到.xib 文件，也无法在.xib 文件中编写方法。如果要编写方法代码，必须到 Xcode 中进行。那么在 Xcode

中编写的方法在.xib 中是如何调用的?

把 IB 对象和方法代码连接在一起的方法叫做 IBAction 连接。IB 把要连接的 Xcode 方法称为 Action(动作),并使用 IBAction 关键字修饰该方法。IB 加载.xib 文件时,会将 File's Owner 对象所指向的 Xcode 代码文件一同加载,并搜索其中的动作(用 IBAction 修饰),并在 File's Owner 的 Connections 窗口中显示。如果你将 IB 对象的 Event(事件)和这个动作相连接,则会在.xib 文件中增加这样一条记录:

```
<object class="IBConnectionRecord">
<object class="IBCocoaTouchEventConnection" key="connection">
    <string key="label">buttonClicked</string>
    <reference key="source" ref="62546756"/>
    <reference key="destination" ref="372490531"/>
    <int key="IBEventType">7</int>
</object>
<int key="connectionID">11</int>
</object>
```

这个动作是在程序员不知道的情况下进行的,我们并不需要手动编辑.xib 文件的内容。仅仅是了解一下 IB 究竟为我们做了些什么。

当 UIViewController 实例化时,IB 会加载 xib 文件。在此过程中,IB 将把 IB 对象(比如一个按钮)的事件代码(比如 TouchUpInside 事件)和方法(比如 buttonClicked 方法)关联起来。具体的过程有点复杂,但其结果和调用对象的 addTarget:action:forControlEvents:方法是完全一样的。

IBAction 连接的创建是非常简单的,它采用了图形化的方式。我们在 UsingViewController 项目中已经示范了如何创建 IBAction 连接,请参考图 5-5 的示例。

5.4.3 委托连接

与前面两种连接 IBOutlet 和 IBAction 不同,委托连接更加“间接”,它需要使用协议才能完成。

打开 DetailViewController.xib 文件,选中 Table View 对象,打开 Connections Inspector 面板,如图 5-8 所示。

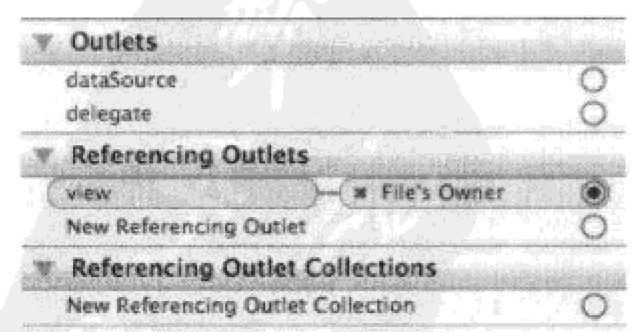


图 5-8 Table View 的 Connections Inspector 面板

我们看到 Table View 的 Outlets 中有一个 dataSource 对象和一个 delegate 对象，这就是两个委托连接。这两个对象右边的空心圆圈表示这两个对象当前都未连接。

首先，我们来连接 dataSource 对象。在 UITableView 中，dataSource 是一个属性：

```
@property(nonatomic, assign) id<UITableViewDataSource> dataSource
```

很显然，要将 dataSource 连接到某个对象（实际上是对它进行赋值），该对象应该是 id<UITableViewDataSource> 类型，即实现了 UITableViewDataSource 协议的 NSObject 对象。

实际上 DetailViewController 就是这样一个对象。你可以打开 DetailViewController.h，它继承了 UITableViewController，而 UITableViewController 中已经声明了对 UITableViewDataSource 协议的实现：

```
@interface UITableViewController : UIViewController <UITableViewDelegate, UITable-
          ViewDataSource>
```

而且在 DetailViewController.m 中也实现了 UITableViewDataSource 协议规定的一系列方法：

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView;
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)
            section;
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
            (NSIndexPath*)indexPath;
```

这些方法是用于为 Table View 提供展现数据的。显然，如果你不在这些方法中提供一些数据，表视图只会展现一些空白的行（如图 5-6 所示）。我们可以在这些方法中添加一些代码，为 Table View 中提供几行数据显示。

首先是-(NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger) section 方法，这个方法用于告诉 Table View 将有几行数据需要展示，在这个方法中，我们总共只需要一行代码就可以了：

```
return 2;
```

这表示，我们将为 Table View 提供 2 行数据显示。

然后是 - (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath 方法，这个方法用于设置 Table View 中每一行要显示的内容。默认情况下，这个方法中已经有一些代码，只需要添加以下内容：

```
switch (indexPath.row) {
        case 0:
            cell.textLabel.text=@"肖工 13304859095";
            break;
        case 1:
            cell.textLabel.text=@"小张 18900147830";
            break;
        default:
```

```
        break;
    }
```

这表示，表视图的第 1 行和第 2 行将分别显示两个人的名称和他们的电话号码。

好了，实现完 UITableViewDataSource 协议，我们就可以把 Table View 的 dataSource 属性连接到 DetailViewController 了。

由于 Table View 本身属于 DetailViewController.xib，而 DetailViewController.xib 的 File's Owner 也是一个 DetailViewController 对象，因此实际上把 dataSource 连接到 File's Owner 就可以了。

在 Table View 的 Connections Inspector 面板，拖动 dataSource 右边的圆圈按钮到 File's Owner 上，即可完成一个委托连接的建立，如图 5-9 所示。

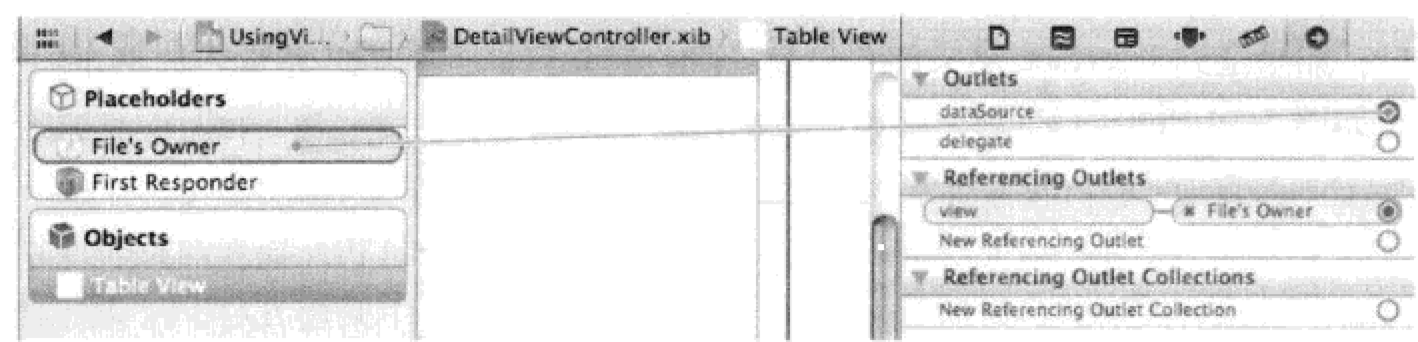


图 5-9 建立委托连接

如果你想取消连接，在 Connections Inspector 面板中，点击该连接中间的小叉，如图 5-10 所示。

运行程序，点击“通讯录”按钮，窗口将显示一个有 2 行数据显示的 Table View，如图 5-11 所示。

图 5-10 点击连接中间的小叉，取消该连接

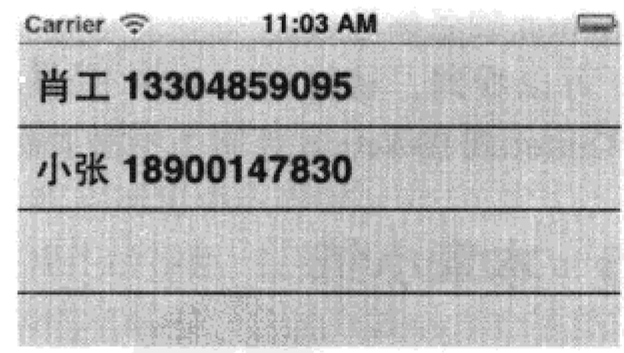


图 5-11 程序运行结果

5.4.4 使用 Assistant Editor 创建连接

Xcode 4 提供了简化的创建连接方式，那就是 Assistant Editor。在 xib 编辑界面时，你可以点击工具栏中的 Show Assistant Editor 按钮，即 Editor 中间的那个按钮（如图 5-12 所示），这样会显示 Assistant Editor 界面（如图 5-13 所示）。在这个窗口中，左边显示了 IB 界面，右边显示 Interface（接口定义）的代码编辑界面，因此非常便于在 IB 和代码之间进行拖拽和连接。

图 5-12 Show Assistant Editor 按钮

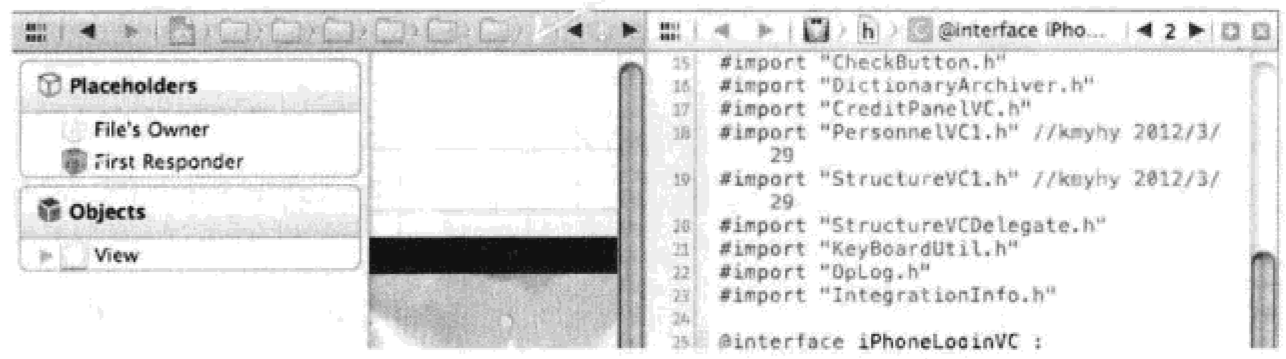


图 5-13 Assistant Editor 界面

要建立一个 IBOutlet 连接，从图 5-13 所示窗口的左边选择一个 IB 对象，然后用右键（或 Ctrl+左键）拖一条线到窗口右边的代码中（如果拖在 interface 定义的花括号内，是建立 IBOutlet 成员出口；如果拖到 interface 定义的花括号之外，则是建立 IBOutlet 属性出口或 IBAction）。这时弹出如图 5-14 所示的窗口：

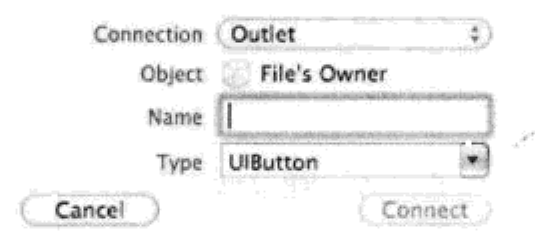


图 5-14 Assistant Editor 弹出的窗口

在 Name 中输入出口的名称，如 btSubmit，然后点击 Connect 按钮，便创建好了一个 IBOutlet 连接。如果要创建 IBAction 连接，在该窗口的 Connection 中选择 Action，然后在 Name 中输入 Action 的名称即可。

可以看到，通过 Xcode 4 提供的 Assistant Editor 工具自动为我们在接口中进行声明（IBOutlet 或 IBAction 声明），我们能更快捷地创建 IB 连接。减少编码工作。

5.5 本章小结

本章介绍了如何在 Xcode 集成环境中使用 Xcode 内置的 Interface Builder 创建图形化界面。

IB 将编辑成果储存为.xib 文件格式。xib 文件是程序中特殊的资源文件，在程序运行时，它会随同二进制代码一起加载到内存，我们要在程序中访问 xib 文件中产生的这些对象（我们称之为 IB 对象），或者让 IB 对象能够通过代码与用户交互，需要使用特殊的方式将 IB 对象和程序代码（类的方法、属性或成员变量）相连，这个过程就是所谓的“连接”。本章介绍了 IB 中的三种连接：IBOutlet 连接、IBAction 连接和委托连接。

IB 是 Mac 下优秀的图形界面设计工具。通过本章的学习，能够让我们用一种更高效、直观、图形化的方式为应用程序创建出优秀的图形界面，从而极大地提高开发效率。

第 6 章　高级图形界面

iOS 高级图形界面由 UIKit 提供，而 Quartz Core 和 Core Animation 则提供更底层的支持。

UIKit 位于 Cocoa Touch 中的最高层级（参见第 2 章中的 2.3 节“Cocoa Touch 框架简介”），是 iOS 开发人员打交道最多的框架，它包含了 Cocoa 中的高级图形界面 API。

上一章我们介绍了 IB，Object Library 中包含的所有内容都来自 UIKit，包括 UIViewController、UIView 及其子类。我们介绍了一些常见的 UIKit 组件，如 UILabel、UIButton 和 UITextField 等。在 IB 中使用它们是很容易的，你需要的仅仅是不断地去熟悉它们。使用这些高级图形界面 API，足以满足大部分 iOS 应用程序的需要。但在一些情况下，我们会在 UIKit 的基础上做一些扩展，即定制自己的 UIKit 组件，这就是本章要讲的内容。如果还不能满足需要，可去寻求更低级的底层 API 的帮助，如 Quartz 或 Core Animation——这也会在其他章节中介绍。

本章将介绍 UIKit 框架中的一些特别的主题，这些主题是关于 UIKit 高级图形 API 在开发中的一些典型问题，包括：

- 应用程序多视图的导航。
- 表视图 UITableView 的应用及其扩展。
- 自定义控件和静态库。
- 翻页控件和翻页控制器。

6.1　应用程序多视图的导航

很少有应用程序只包含一个视图。本书假设所有的 iOS 企业应用程序都是多视图的，这意味着每个应用程序都是基于导航的。所以接下来我们介绍应用程序的导航。

UIKit 框架提供了一系列特殊的控制器，专门用于多视图的导航，包括：UITabBarController 和 UINavigationController。其中，前者提供多个视图间的随机导航，后者则提供多个视图间的顺序导航。也就是说，使用 UITabBarController 进行导航，你可以在多个视图间任意切换，从一个视图直接跳到任意一个视图；而 UINavigationController 的导航是线性的，所有视图以双向链表的结构排列，你可以按从头到尾的顺序（或相反方向）来访问所有视图。

6.1.1　UITabBarController

UITabBarController 是 iPhone 上最常见的导航控件，它支持数量有限的几个视图之间的导航。我们在 iPhone 的拨号程序中可以看到 Tab Bar Controller（图 6-1）。

在图 6-1 所示的拨号程序中，位于窗口底部的一系列 Tab Bar 按钮列出了 UITabBarController 所管理的每个视图：个人收藏、通话记录、通讯簿等。可以看到，由于屏

幕空间的限制，可排放的按钮个数是有限的。

图 6-1 iPhone 的拨号程序底部显示了一个 UITabBarController

提示：如果 Tab Bar 上的按钮超过 5 个，第 5 个按钮将显示为 More 按钮。

在 UsingTabBarController01 项目中，我用纯代码方式创建了一个 UITabBarController，你可以在光盘的“source/第 6 章”目录下找到它。

与在 IB 中使用 Tab Bar Controller 不同，如果采用纯代码方式使用 Tab Bar Controller，必须子类化 UITabBarController。UITabBarController 有一个 NSArray 属性 viewControllerArray，我们只需把一个或多个 View Controller 加入到一个数组中，然后把这个数组赋值给 viewControllerArray 属性，即可让 Tab Bar Controller 在这些 View Controller 之间进行导航。同时，必须设置各个 View Controller 的 tabBarItem 属性。这个 tabBarItem 实际就是一个 UITabBarItem，它会在 Tab Bar Controller 底部的选项卡栏中显示（如图 6-2 所示）。这个 Tab Bar Item 需要你提供一张图片作为按钮的图标，以及一个字符串用于显示在按钮图标下方。

新建一个 Empty Application 项目。修改编辑应用程序委托类，在“[window makeKeyAnd Visible];”之前加入以下代码：

```
HomeViewController* home=[[HomeViewController alloc]init];
[home setTitle:@"TabBar Controller"];
[window addSubview:home.view];
```

别忘了，在文件开头要加上“#import “HomeViewController.h””。

然后新建 Objective－C Class HomeViewController：

```
#import <Foundation/Foundation.h>
#import <UIKit/UIKit.h>
@interface HomeViewController : UITabBarController {
}
@end
#import "HomeViewController.h"
```

```
@implementation HomeViewController
-(void)loadView{
    [super loadView];
    NSMutableArray *viewControllerArray = [NSMutableArray array];//❶
    UIViewController *firstVC=[[UIViewController alloc]init];// ❷
    firstVC.view.backgroundColor=[UIColor redColor];
    firstVC.tabBarItem = [[UITabBarItem alloc] initWithTitle:@"Red" image:[UIImage
            imageNamed:@"index.png"] tag:1000];// ❸
    [viewControllerArray addObject:firstVC];// ❹
    [firstVC release];// ❺
    UIViewController* secondVC=[[UIViewController alloc]init];// ❻
    secondVC.tabBarItem = [[UITabBarItem alloc] initWithTitle:@"blue" image:[UIImage
             imageNamed:@"settings.png"] tag:2000];// ❼
    secondVC.view.backgroundColor=[UIColor blueColor];
    [viewControllerArray addObject:secondVC];// ❽
    [secondVC release];
    self.viewControllers = viewControllerArray;// ❾
}
@end
```

代码说明：

❶ 初始化一个 NSMutableArray，名为 viewControllerArray。Tab Bar Controller 并不会在窗口中显示 Tab Bar 以外的东西，但它的 viewControllers 属性却可以包含多个用于导航和显示的 View Controller。因此我们用一个 NSMutableArray 用于向其中添加多个 View Controller。

❷ 构造第一个 View Controller 对象 firstVC。然后我们把它的背景色设置为红色，以便我们和第二个 View Controller 区分。

❸ 每个 View Controller 都有一个 tabBarItem 的属性。这个 tabBarItem 用于在 Tab Bar 中显示为 Tab Bar Item。我们需要为这个 Tab Bar Item 提供一个用作显示图标的图片文件和用于在图标下方显示的文本。

注意：你可能需要为 Tab Bar Item 准备一张图片。然而，最终 Tab Bar Item 上显示出来的可能不是你想要的结果。它会显示一个只有轮廓的图形而丢失所有的颜色信息。这是因为 UIKit 框架只使用了你的图片的 alpha 通道进行了遮罩处理。因此你提供一张有色的图片是没有任何意义的，最终用到的只是图片的 alpha 通道。关于这个问题，你可以参考作者的这篇博客：http://blog.csdn.net/kmyhy/article/details/7174162。

---

❹ 将 firstVC 加入到数组。

❺ 释放 firstVC，因为数组已经自动持有。

❻ 初始化第二个 View Controller。然后我们设置它的背景颜色为蓝色，以便我们和第一个 View Controller 区分。

❼ 设置 secondVC 的 tabBarItem。参考 3。

❽ 将 secondVC 加入到数组。

❾ 设置 Tab Bar Controller 的 viewController 为数组 viewControllerArray（我们已经向其中加入了两个 View Controller）。

程序运行的效果如图 6-2 所示，当你点击 UITabBarController 底部的选项栏按钮时，视图会在不同的 ViewController 之间切换。

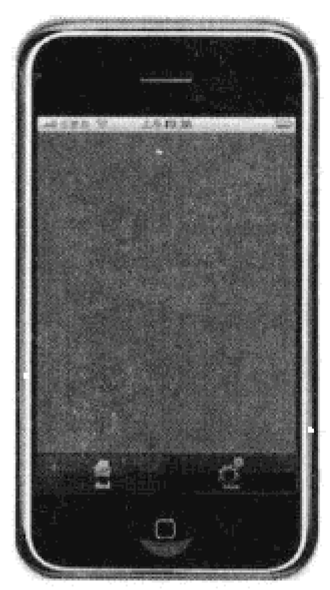


图 6-2 程序运行效果

提示：你可以在 IB 中直接使用 Tab Bar Controller 而不用自己编写代码。但通常，以源代码方式使用 Tab Bar Controller 要更加灵活一些。

6.1.2 UINavigationController

导航控制器在 iPhone 上非常常见，比如日历、照片等程序中。

在图 6-3 的照片程序中，导航控制器显示为窗口上方的一个导航条。如果你点击导航条左上角的导航按钮，可以切换到前一视图。

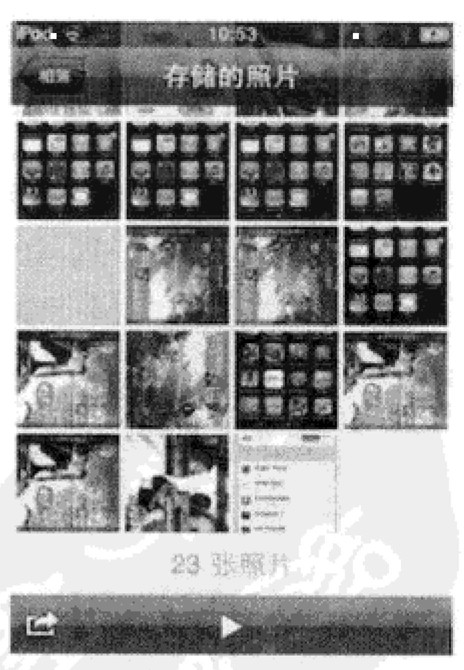


图 6-3 照片程序中的导航控制器

示例项目 UsingNavigationController 位于光盘“source/第 6 章/UsingNavigationController”目录下，它呈现了一个导航控制器使用的例子。按照老规矩，我们首先需要修改应用程序委托类，导入头文件 HomeViewController.h，在“[window makeKeyAndVisible];”之前加入以下代码：

```
HomeViewController* home=[[HomeViewController alloc]init];
    home.title=@"home";
UINavigationController* nc=[[UINavigationController alloc]
    initWithRootViewController:home];
[window addSubview:nc.view];
```

UINavigationController 需要一个 rootViewController 才能工作。在上面的代码中，我们在 Navigation Controller 的初始化方法中传递了一个 HomeViewController 作为它的 rootViewController。这个 HomeViewController 其实是一个普通的 UIViewController，但在其中我们加入了一个 UIButton，用以演示如何导航到其他页面：

```
#import <UIKit/UIKit.h>
#import "SecondViewController.h"
@interface HomeViewController : UIViewController {
    UIButton* button1;
}
@end
#import "HomeViewController.h"
@implementation HomeViewController
-(void)loadView{
    self.view=[[UIView alloc]initWithFrame:CGRectMake(0, 0, 320, 480)];
    button1=[UIButton buttonWithType:UIButtonTypeRoundedRect];
    button1.frame=CGRectMake(60, 60, 200, 45);
    [button1 setTitle:@"下一页" forState:UIControlStateNormal];
    [button1 addTarget:self action:@selector(openFirst) forControlEvents: UIControl-
            EventTouchUpInside];
    [self.view addSubview:button1];
}
-(void)openFirst{
    SecondViewController *next=[[SecondViewController alloc]init];
    next.title=@"第二页";
    [self.navigationController pushViewController:next animated:YES];
}
- (void)dealloc {
   [button1 release];
   [super dealloc];
}
@end
```

值得注意的是 openFirst 方法，我们在按钮被点击时，创建了一个视图控制器 SecondViewController ，然后使用 UINavigationController 的 pushViewController:animated:方法

将导航控制器的视图切换至新的视图。注意，UIViewController 的 navigationController 属性实际上指向了一个导航控制器，当一个 View Controller 被指定为该导航控制器的 rootViewController，或者使用 UINavigationController 的 pushViewController: animated:方法将这个 View Controller 加入 Navigation Controller 时，该 View Controller 的 navigationController 属性会保存一份 Navigation Controller 的引用。

SecondViewController 继承自 UIViewController，它的实现和 HomeViewController 很相像，它也提供了一个按钮在视图中。当这个按钮（即“返回”按钮）被点击，将触发 goHome 方法。该方法中仅有一句代码，调用了 UINavigationController 的 popViewControllerAnimated:方法，这个方法将当前 View Controller（即 SecondViewController）弹出栈顶，于是下面的 View Controller（即前一页视图 HomeViewController）成为栈顶并显示。

```
#import <Foundation/Foundation.h>
@interface SecondViewController : UIViewController {
    UIButton* button1;
}
@end
#import "SecondViewController.h"
        @implementation SecondViewController
-(void)loadView{
    self.view=[[UIView alloc]initWithFrame:CGRectMake(0, 0, 320, 480)];
    self.view.backgroundColor=[UIColor redColor];
    button1=[UIButton buttonWithType:UIButtonTypeRoundedRect];
    button1.frame=CGRectMake(60, 60, 200, 45);
    [button1 setTitle:@"返回" forState:UIControlStateNormal];
    [button1 addTarget:self action:@selector(goHome) forControlEvents:UIControl
            EventTouchUpInside];
    [self.view addSubview:button1];
}
-(void)goHome{
    [self.navigationController popViewControllerAnimated:YES];
}
- (void)dealloc {
    [button1 release];
   [super dealloc];
}
@end
```

导航控制器维护了一个由视图控制器组成的栈式结构。最初 rootViewController 位于栈底同时也是栈顶（因为此时栈中只有一个视图控制器）。当 pushViewController: animated:方法将一个视图控制器加入导航控制器时，这个视图控制器会被压入栈顶，同时该控制器的视图被显示。你可以不停地往栈中加入新的视图控制器，但导航控制器始终只显示位于栈顶的视图。与此相反，当导航控制器调用 popViewControllerAnimated:方法时，栈顶的视图控制器会被弹出，

于是位于下层的视图会显示出来，导航控制器返回上一层视图。不停地弹出视图控制器，直至返回到 rootViewController 为止。栈不会为空，因为 rootViewController 不能被弹出。

程序运行效果如图 6-4 和图 6-5 所示。

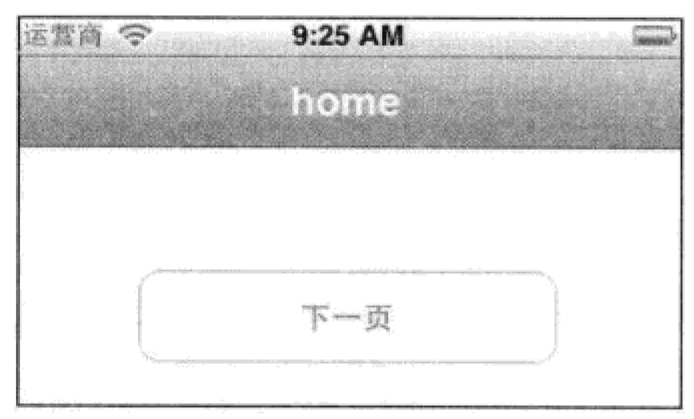


图 6-4　导航控制器的 rootViewController，点击按钮显示下层视图

图 6-5　导航控制器的第二层视图，点击按钮返回上层视图

Cocoa 提供的 UINavigationController 为多视图应用程序提供了简单的线性导航，你可以在一系列视图中顺序地前进或后退，但当你需要随机地从一个 View Controller 导航到堆栈的任意 View Controller 就不太方便了。

注意：很多时候，以代码方式使用 Navigation Controller 更容易。

---

navigationItem 是一个 UINavigationItem 对象，有 3 个主要的属性：leftBarButtonItem、title 和 rightBarButtonItem。分别代表导航栏的左、中、右三个部分，即左边的导航栏按钮、中间的导航栏标题（字符串）、右边的导航栏按钮。

此外，如果你想在左边或右边放多个导航栏按钮（不止一个），可以使用 navigationItem 的 leftBarButtonItems 属性和 rightBarButtonItems 属性。

接下来，我们分别介绍如何定制导航栏的左、右按钮。

首先是 leftBarButtonItem。通常，NavigationController 会在导航栏上提供一个默认的返回按钮作为导航栏的左按钮，用于返回到前一个视图控制器。但我们可以用自己的按钮替换掉默认的返回按钮。

我们复制了一份 UsingNavigationController 项目的副本 UsingNavigationController02，并对它做一些小改动（源文件位于光盘“source/第 6 章/UsingNavigationController02”目录下）。

在 HomeViewController 的 loadView 方法中加入：

```
UIBarButtonItem *item=[[UIBarButtonItem alloc]initWithTitle:@"下一页"
           style:UIBarButtonItemStyleBordered
          target:self
          action:@selector(openFirst)];
self.navigationItem.leftBarButtonItem = item;
```

这会在导航栏上左边加入一个按钮，我们通过初始化方法的 target 参数让它的功能与

button1 按钮一样（调用 openFirst）。

接下来是 rightBarButtonItem，在 SecondViewController 的 loadiew 方法中加入：

```
UIBarButtonItem *item=[[UIBarButtonItem alloc]initWithTitle:@"返回"
    style:UIBarButtonItemStyleBordered
    target:self
    action:@selector(goHome)];
self.navigationItem.rightBarButtonItem = item;
```

注意：这次，我们通过 rightBarButtonItem 属性把按钮放到导航条右边。

运行代码，除了按钮的位置略有不同（我们将按钮放到了导航栏里），跟上一个示例程序没有任何区别。

除此之外，你可以创建各式各样的按钮，UIBarButtonItem 支持以下初始化方法：

- initWithCustomView：支持创建完全自定义的按钮。
- initWithImage:style:target:action：支持图片按钮。
- initWithBarButtonSystemItem:target：action 可以创建系统指定的一系列标准按钮，如表 6-1 所列。

表 6-1 可用于工具栏和导航栏的标准按钮

按 钮	含 义	名 称
	打开操作列表，使用户能够执行特定应用程序相关的操作	Action
	打开操作列表，以显示拍照模式下的照片选择器	Camera
	在编辑模式下打开新消息视图	Compose
	显示与特定应用程序相关的书签	Bookmarks
	显示搜索框	Search
	创建新项	Add
	删除当前项	Trash
	将某项移动或传送至应用程序的目标位置，如某文件夹	Organize
	将某项发送或传送至其他位置	Repty
	停止当前操作或任务	Stop
	刷新内容（仅当必要时使用，否则自动刷新）	Refresh
	开始播放媒体或幻灯片	Play
	在播放媒体或幻灯片时快进	FastForward
	暂停播放媒体或幻灯片（这也暗示着要保留当前上下文位置）	Pause
	在播放媒体或幻灯片时快退	Rewind

6.1.3 窗体导航应用实例

现在，我们将结合已介绍过的 UITabBarController 和 UINavigationContoller，来创建一个多视图的导航应用程序。虽然它不会有太实用的功能，但却是搭建今后 iOS 企业应用框架的基础。

我们复制了一份 UsingTabBarController01 项目的副本，将项目重命名为 TabBarNavigator Controller，并在此基础上进行修改。

在项目 UsingNavigationController 中，打开 HomeViewController 类，在 Editor 窗口中选中符号 HomeViewController，点击右键并选择“Refractor → Rename…”，将 HomeViewController 重命名为 RootViewController，如图 6-6 所示。

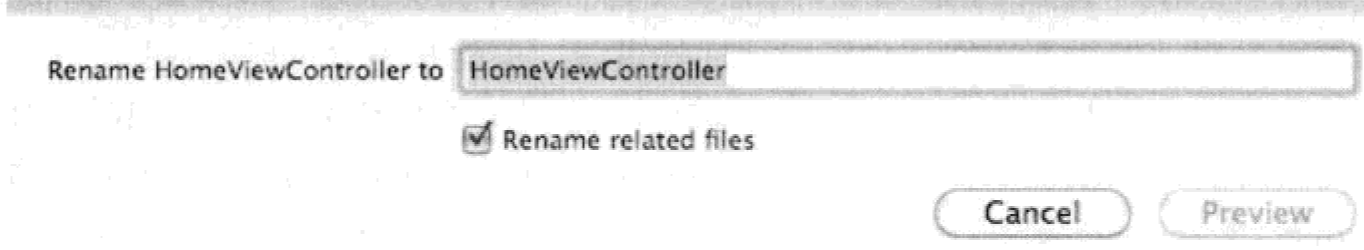


图 6-6 类的重命名

解决类名重复的问题后，将 RootViewController 类和 SecondViewController 类对应的 4 个文件拷贝到 TabBarNavigationController 项目中。然后修改 HomeViewController 的 loadView 方法：

```
-(void)loadView{
    [super loadView];
    NSMutableArray *viewContrcllerArray = [NSMutableArray array];
    RootViewController* root=[[RootViewController alloc]init];
    root.tabBarItem = [[UITabBarItem alloc] initWithTitle:@"red" image:
                      [UIImage imageNamed:@"index.png"] tag:2000];
    root.view.backgroundColor=[UIColor redColor];
    UINavigationController *nc = [[UINavigationController alloc]
                initWithRootViewController:root];// ❶
    nc.navigationBar.barStyle = UIBarStyleBlack;
    [viewControllerArray addObject: nc];// ❷
    [root release];
    [nc release];
    root=[[RootViewController alloc]init];
    root.tabBarItem = [[UITabBarItem alloc] initWithTitle:@"blue" image:
                      [UIImage imageNamed:@"settings.png"] tag:2000];
    root.view.backgroundColor=[UIColor blueColor];
    nc = [[UINavigationController alloc]
          initWithRootViewController:root];// ❸
    nc.navigationBar.barStyle = UIBarStyleBlack;
    [viewControllerArray addObject: nc];// ❹
    [root release];
    [nc release];
    self.viewControllers = viewControllerArray;
}
```

代码说明：

❶ 初始化一个 Navigation Controller，用一个 View Controller 作为 Navigation Controller 的 rootViewController。

❷ 把 Navigation Controller 加到数组，而不是直接把 View Controller 加到数组。

❸ 再次构建 Navigation Controller，用另一个 View Controller 作为 rootViewController。

❹ 把 Navigation Controller 加到数组。

如果我们忘记导入 RootViewController.h 头文件，编译时会出现一些错误。这个错误对初学者来说很容易犯。如果下次你还会出现这样的错误，可能我不会再提醒你！

所有的代码都是熟悉的，我们没有采用新技术，只是把两种技术结合在一起使用而已。我们在 HomeViewController（一个 UITabBarController）中用 UINavigationController 填充了其 viewControllers 数组。这样，程序的导航方式不仅仅有选项栏，而且可以在点击选项栏按钮后再使用导航条，从而多视图的组织形式更加丰富和立体。

6.2 表视图 UITableViewController 的应用及其扩展

UITableViewController 可能是 Cocoa 中最复杂的视图控制器，我们需要分成许多部分来进行介绍。

UITableViewController 和 UITableView 之间的关系类似于 UIViewController 和 UIView。UITableViewController 控制了 UITableView 的显示内容。默认情况下，UITableViewController 负责为 UITableView 提供表数据，并响应事件。

UITableView 其实类似于一个数组，它的“行”（或“单元格”）由 UITableViewCell 构成，多个行或单元格就组成了 UITableView。

6.2.1 简单的表视图控制器

创建一个简单的表视图很简单，只需要几行代码。新建 Empty Application 应用程序 UsingTableViewController01。打开应用程序代理类，在“[window makeKeyAndVisible];”一行前加入代码：

```
UITableViewController* tableVC=[[UITableViewController alloc]
   initWithStyle:UITableViewStylePlain];
[window addSubview:tableVC.view];
```

这样可以得到一个空的表视图。

注意 UITableViewController 的初始化方法使用了特别的参数 UITableViewStyle。因为表视图控制器提供了一个默认的 UITableView——即 UITableViewController 的 view 属性，所以这个参数会用来指定表视图的风格：UITableViewStylePlain 和 UITableViewStyleGrouped。前者是一个空白的普通的表视图，而后者提供了分组的表视图，后面将会介绍。

UITableViewController，之所以说它简单，是因为它其实和普通的 UIViewController 没有任何区别，除了提供一个免费的 UITableView 之外。我们其实完全可以只使用 UIViewController

而不用 UITableViewController，就能使用表视图。后面的例子会继续说明这一点。

6.2.2 UITableView 的数据源和委托

继续修改 UsingTableViewController01 项目中的代码。首先在应用程序委托中声明一个数组成员 model，用于作为表视图的数据：

```
NSArray* model;
```

然后在方法 application:didFinishLaunchingWithOptions:中：

```
model=[[NSArray alloc]initWithObjects:
      @"green",@"blue",@"black",nil];
UITableView* tableview=[[UITableView alloc]
      initWithFrame:CGRectMake(0, 20, 320, 460)
       style:UITableViewStylePlain];
tableview.dataSource=self;
 [window addSubview:tableview];
 [window makeKeyAndVisible];
return YES;
```

看看我们都做了些什么。首先，构建了一个 UITableView，然后把它的 dataSource（数据源）设置为 self（应用程序委托）。而 UITableView 的 dataSource 属性定义如下，它指定为一个实现了协议 UITableViewDataSource 的对象：

```
@property(nonatomic,assign)  id <UITableViewDataSource> dataSource;
```

因此我们应用程序委托需要实现 UITableViewDataSource 协议。查看 UITableViewDataSource 协议的定义，我们发现它规定了二个必须的方法和三个可选方法（在 UITableView.h 中）。为了简便，我们决定在应用程序委托类中只实现两个必须的方法：

```
#pragma mark -
#pragma mark tableview datasource method
-(NSInteger)tableView:(UITableView *)table numberOfRowsInSection:(NSInteger) section{
    return model.count;
}
-(UITableViewCell*)tableView:(UITableView *)tableView cellForRowAtIndexPath:
         (NSIndexPath *)indexPath{
    static NSString* cellIndentifier=@"myCellIndentifier";
    UITableViewCell* cell=[tableView dequeueReusableCellWithIdentifier:
           cellIndentifier];
    if (cell==nil) {
          cell=[[[UITableViewCell alloc]initWithStyle:UITableViewCellStyleDefault
               reuseIdentifier:cellIndentifier]autorelease];
    }
    cell.textLabel.text=[model objectAtIndex:indexPath.row];
    return cell;
}
```

第一个方法很简单，返回了表视图的行数。由于我们决定使用字符串数组 model 作为表视图的显示数据，因此我们简单地返回了 model 的元素个数。

第二个方法是 UITableViewDataSource 协议中最重要的方法，它需要我们返回表视图的每一行的单元格视图 UITableViewCell。UITableViewCell 中你可以加入任意多的控件，比如标签、文本、图片、按钮等等。但为图简便，我们没有加入任何东西，而只使用 UITableViewCell 默认自带的一个 textLabel（UILabel）控件，并将其 text 属性设置为对应的 model 数组中的字符串。在此之前的 3 行代码基本上是固定的，每次实现这个方法时，你都不得不拷贝一遍这些代码。为什么会这样？

为了节省性能，Cocoa 为表视图提供了一种单元格重用机制，当 Cocoa 需要渲染一个单元格视图时，它并不会每次都去创建一个，而会去检索“单元格缓存”中可重用的单元格。让我们举个实际点的例子。比如，当表视图第一次显示时，它显示了 20 行数据。这 20 行单元格肯定都是新建的（因为是第一次显示，缓存肯定是空的，没有任何数据），新创建的单元格会被放到指定了 Identifier 的“单元格缓存”中。然后用户通过滚动或者搜索工具栏检索了新的数据，有效的数据有 24 行，而这 24 行将刷新表视图。Cocoa 会从指定 Identifier 的缓存中取出 20 个已创建的单元格进行重用，而有 4 行单元格将不得不重新创建（因为缓存中只有 20 个可用）。然后会把这 24 个单元格缓存起来。记住每次刷新表视图（调用 reloadData），单元格缓存会被重建。所以如果你注意到单元格缓存的容量变化，会发现它数量总是在不停地变化着的，或者增加，或者减少。

基于以上机制，所以我们第二个方法的代码会写成那样。为什么方法第一行定义了一个 static NSString 变量？因为表视图需要一个 Identifier 作为要访问的单元格缓存的标识。对于某个指定的表视图，它总是需要一个自己专用的“单元格缓存”，我们用一个字符串指定这个单元格缓存的名字。至于要把它指定为 static 存储类型原因，可能你也想到了。方法会被调用多次，而 Identifier 的初始化却没有必要进行多次，因为单元格缓存的 Identifier 总是固定不变的。

方法第 2 行就是从指定 Identifier 的单元格缓存中检索一个可用的单元格。但缓存不一定总是可用的（因为有时候要显示的单元格可能比已缓存的单元格数目要多），所以第 3 行要判断到底缓存有没有读取成功。如果没有可用的缓存单元格，我们就不得不自己创建一个。

程序运行效果如图 6-7 所示。

图 6-7 表视图应用程序

注意，实际上任何对象（id 类型）都可以作为表视图的 dataSource，但该对象必需实现 UITableViewDelegate 协议。这里，我们用应用程序委托类来作为数据源（通过实现 UITableViewDataSource 协议）。然而更加常见的情况，是使用 UITableViewController 或者 UIViewController 来作为 Table View 的数据源。因为除了提供数据源之外，UITableViewController 或者 UIViewController 还能为表视图提供事件响应。

同 dataSource 一样，表视图的委托（delegate 属性）可以由任何对象(id 类型)承担，但必须实现 UITableViewDelegate 协议。查看 UITableViewDelegate 的定义，包括了 19 个可选方法（UITableView.h 中）。你可以实现其中任意方法。但通常我们最常用的是这个方法：

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
      *)indexPath;
```

该方法捕获了单元格的触摸事件，你可以在这里进行处理用户在单元格上的点击（或选定）。

我们决定用应用程序委托作为 Table View 的 delegate（委托对象）。找到代码 “tableView.dataSource=self;” 一行，在后面加入代码：

```
tableView.delegate=self;
```

于是接下来在应用程序委托类中实现 UITableViewDelegate 协议的 tableView:didSelectRow AtIndexPath:方法：

```
NSLog(@" 你选择了第%d 行：%@" ,indexPath.row,[model objectAtIndex:indexPath.row]);
```

运行程序，当你点击单元格时，注意控制台的输出：

```
2011-09-06 11:40:19.351 UsingTableViewController01[4451:207]你选择了第0行:green
2011-09-06 11:40:19.452 UsingTableViewController01[4451:207]你选择了第1行:blue
2011-09-06 11:40:19.535 UsingTableViewController01[4451:207]你选择了第2行:black
```

注意，你在编译程序时会发现编译器提示了两个警告：

```
Class 'UsingTableViewController01AppDelegate' does not implement the
'UITableViewDataSource'protocol
Class 'UsingTableViewController01AppDelegate' does not implement the
'UITableViewDelegate'protocol
```

这是因为尽管你确实实现了这两个协议，但编译器只检查了你的头文件。你需要在头文件中显式地声明你已经实现了这两个协议：

```
@interface UsingTableViewController01AppDelegate : NSObject
<UIApplicationDelegate,UITableViewDataSource,UITableViewDelegate>{
...
```

再次编译，警告消除了。

6.2.3 分组表视图

分组表视图可以把表视图按节显示（即 Style 指定为 UITableViewStyleGrouped），在节中

显示单元格。分组表视图需要实现额外的数据源方法。我们修改应用程序委托文件，把UITableViewDataSource协议的实现代码修改为：

```
#pragma mark -
#pragma mark tableview datasource method
-(NSInteger)tableView:(UITableView *)table numberOfRowsInSection:(NSInteger)
     section{ // ❶
     if (section==0) {
             return 1;
     }else {
             return 2;
     }

}
-(NSString*)tableView:(UITableView *)tableView titleForHeaderInSection:(NSInteger)
     section{ // ❷
     if (section==0) {
             return @"红色";
     }else {
             return @"其他颜色";
     }
}
-(NSInteger)numberOfSectionsInTableView:(UITableView *)tableView{ // ❸
     return 2;
}

-(UITableViewCell*)tableView:(UITableView *)tableView cellForRowAtIndexPath:
     (NSIndexPath *)indexPath{ // ❹
     static NSString* cellIndentifier=@"myCellIndentifier";
     UITableViewCell* cell=[tableView dequeueReusableCellWithIdentifier:cell
                          Indentifier];
     if (cell==nil) {
             cell=[[[UITableViewCell alloc]initWithStyle:UITableViewCellStyleDefault
                   reuseIdentifier:cellIndentifier]autorelease];
     }
     cell.textLabel.text=[model objectAtIndex:indexPath.row];
     return cell;
}
```

代码说明：

❶ 第1个方法在原来的基础上作了一些变化。我们把表视图分成2节。第1节1行，第2节2行。

❷ 第2个方法是新增的。我们为这两个节分别提供了两个标题字符串。

❸ 第3个方法也是新增的。我们返回了整个表视图分节的数目（即2节）。

❹ 第 4 个方法跟原来没有任何变化，因为不管是分成多少节，它们的单元格视图没有区别。如果我们要在这个方法中为不同的节提供不同的单元格视图，你可能得识别 indexPath 参数中的 section 属性分别进行处理，比如：

```
if(indexPath.section==0){
    cell.textLabel.textColor=[UIColor redColor];
}else{
...
}
```

现在，这个分组的表视图效果如图 6-8 所示，完整的代码见光盘“source/第 6 章/UsingTableViewController01”文件夹。

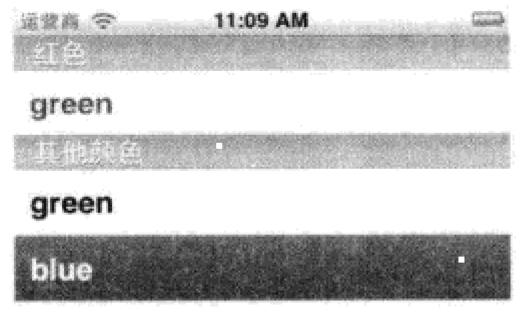


图 6-8　分组的表视图

6.2.4　可折叠的分组表视图

分组表视图的高级应用，就是可折叠分组表视图，如图 6-9 所示。示例代码位于光盘“source/第 6 章/ExTable”目录。

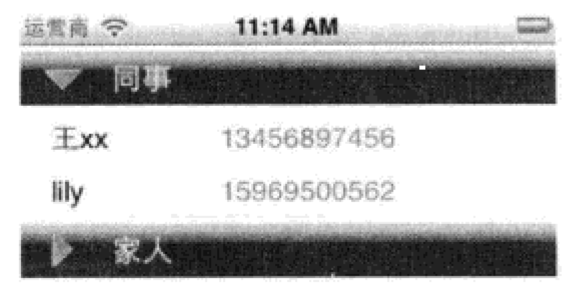


图 6-9　可折叠分组的表视图

打开 ExTable 项目，首先来查看类 ExpandableButton 的定义。

由于原来的分组表视图中，节或者节标题没有响应触摸的能力，它仅仅是一个简单的 Label，用于显示节标题。所以我们定制了 ExpandableButton 控件来代替 Label 控件作为节标题。在 ExpandableButton 中，我们设计了如下成员：

- ❑ Label 用于显示小节标题。
- ❑ ImageView 用于显示“折叠 / 展开”按钮。

- BOOL 属性用于存储按钮当前的“折叠 / 展开”状态。
- setTitle:方法用于设置节标题：

```
-(void)setTitle:(NSString*)title
{
    label.text=title;
}
```

用一对 expanded 的访问方法来读取和访问 BOOL 值 expanded：

```
-(BOOL)expanded
{
    return expanded;
}
-(void)setExpanded:(BOOL)b{
    if(b!=expanded){
        expanded=b;
        if(expanded){//如果是展开的，即“—”号状态，显示“—”号图片
                [icon setImage:[UIImage imageNamed:@"minus.png"]];
        }else {
                [icon setImage:[UIImage imageNamed:@"plus.png"]];
        }
    }
}
```

主要的代码在 initWithFrame:方法里，在该方法里初始化所有的控件属性，把它们布局到合适的位置：

```
-(id)initWithFrame:(CGRect)frame
{
    if (self=[super initWithFrame:frame]) {
        UIImage* img=[UIImage imageNamed:@"blank.png"];
        UIImageView *imageview=[[UIImageView alloc] initWithFrame:
                CGRectMake(0, 0, frame.size.width, frame.size.height)];
        [imageview setImage:img];
        [self addSubview:imageview];
        [imageview release];

        icon=[[UIImageView alloc]initWithFrame:
                CGRectMake(10, 0, frame.size.height, frame.size.height)];
        [icon setImage:[UIImage imageNamed:@"plus.png"]];
        [self addSubview:icon];

        label=[[UILabel alloc]initWithFrame:CGRectMake(icon.frame.size.width+24, 0,
```

```
                frame.size.width-icon.frame.size.width-24,
                frame.size.height)];
        label.font=[UIFont fontWithName:@"Arial" size:18];
        label.backgroundColor=[UIColor clearColor];
        label.shadowColor=[UIColor lightGrayColor];
        label.textAlignment=UITextAlignmentLeft;
        label.textColor = [UIColor whiteColor];
        label.highlightedTextColor =[UIColor
             colorWithRed:0x58/255.0
             green:0xA4/255.0
             blue:0xD6/255.0
             alpha:1]; //#58A4D6 十六进制转换
             label.shadowOffset = CGSizeMake(0.0, 1.0);
        [self addSubview:label];
    }
    return self;
}
```

接下来是 ExTableGroup 类，用它存储分组后的数据。

ExTableGroup 类有三个成员，BOOL 变量用于存储该小节的“折叠 / 展开”状态，NSString 用于存储该小节标题，可变数组用于存储该节所包含的行数据。这些属性都使用 @synthesize 合成了访问方法，初始化方法如下：

```
-(id)initWithTitle:(NSString *)string Expanded:(_Bool)expandedOrNo{
    if(self=[super init]){
            title=string;
            isExpanded=expandedOrNo;
            children=[[NSMutableArray alloc]init];
    }
    return self;
}
```

这个方法用于初始化带指定标题和 isExpanded 属性的 ExTableGroup 对象。然后是类 ExTable，继承 UIView。

ExTable 类中设计了一个 UITableView 对象，用于显示表视图；可变数组 groups 保存各个分组（小节），每个小节用 ExTableGroup 对象所代表；float rowHeight 指定了行的高度；id<ExTableDelegate> exTableDelegate 用于指向实现了 ExTableDelegate 协议的委托对象。

ExTableDelegate 协议定义了两个可选方法，一个用于提供表视图的单元格视图，一个用于响应单元格触摸事件：

```
@protocol ExTableDelegate<NSObject>
@optional
-(UITableViewCell*)tableView:(UITableView *)tableView cellForRowAtIndexPath:
       (NSIndexPath *)indexPath;
```

```
-(void)tableView:(UITableView *)table didSelectRowAtIndexPath:(NSIndexPath*)
      indexPath;
@end
```

在 ExTable 的实现部分，定义了一对 groups 方法和 setGroups 方法，用于作为属性 groups 的访问方法：

```
-(NSArray*)groups{
    return groups;
}
-(void)setGroups:(NSMutableArray*)arr{
    groups=[[NSMutableArray alloc]initWithArray:arr];
    [_tableView reloadData];
}
```

resetGroups 方法是一个公开的便利方法，使用两个嵌套的循环将所有节中所有行的行模型的 checked 属性设置为 NO。可以用它来重设表视图，清除所有已选择的行：

```
-(void)resetGroups{
    // 清空 groups 中所有选择
    assert(groups!=nil);
    for(id each in groups){
        assert(each->isa==[ExTableGroup class]);
        ExTableGroup* g=(ExTableGroup*)each;
        // 把所有已打开的 section 收起来
        if (g.isExpanded)
            g.isExpanded=NO;
        for(NSMutableDictionary* d in g.children){
            if([(NSNumber*)[d objectForKey:@"checked"] boolValue])
                [d setObject:[NSNumber numberWithBool:NO] forKey:@"checked"];
        }
    }
    [_tableView reloadData];
}
```

initWithFrame 方法初始化了表视图并用 addSubView:方法加入到 ExTable 的视图中：

```
- (id)initWithFrame:(CGRect)frame {
   if ((self = [super initWithFrame:frame])) {
        rowHeight=32;
        self.backgroundColor=[UIColor grayColor];
        _tableView = [[UITableView alloc] initWithFrame:frame style:UITableView
                     StylePlain];
        _tableView.delegate = self;
        _tableView.dataSource = self;
        //[self setModal];
        [self addSubview:_tableView];
```

```
    }
    return self;
}
```

expandButtonClicked:方法用于处理 ExpandableButton（即节标题）点击事件，调用了 collapseOrExpand:方法：

```
-(void)expandButtonClicked:(id)sender{
    ExpandableButton* btn=(ExpandableButton*)sender;
    int section=btn.tag; //取得节号
    [self collapseOrExpand:section];
    //刷新 tableview
    [_tableView reloadData];
}
```

collapseOrExpand:方法对指定索引所对应的节进行展开（或折叠）操作：

```
-(void)collapseOrExpand:(int)section{
    // 根据节索引取出组 model
    id obj=[groups objectAtIndex:(section)];
    // 确保 groups 数组中存放的是 ExTableGroup 对象
    assert(obj->isa==[ExTableGroup class]);
    ExTableGroup* group=(ExTableGroup*)obj;
    //若原来是折叠的则展开，若原来是展开的则折叠
    group.isExpanded=!group.isExpanded;
}
```

接下来就是实现 UITableViewDelegate 和 UITableViewDataSource 的七个协议方法。

numberOfSectionsInTableView:方法返回了 groups 数组中的小节数：

```
if(groups!=nil)
        return groups.count;
    else
        return 0;
```

tableView: heightForHeaderInSection:方法返回了节标题的高度 40：

```
return 40;
```

tableView: numberOfRowsInSection:方法根据 groups 数组中的 ExTableGroup 对象的 children 属性，返回了节的行数：

```
// 根据节索引取出组 model
    id obj=[groups objectAtIndex:(section)];
    // 确保 groups 数组中存放的是 ExTableGroup 对象
    assert(obj->isa==[ExTableGroup class]);
    ExTableGroup* group=(ExTableGroup*)obj;
    //对指定节进行“展开”判断
```

```
    if (group.isExpanded==NO) {//若本节不是“打开”的，其行数返回为 0
        return 0;
    }
    return [group.children count];
```

tableView: heightForRowAtIndexPath:方法返回了 rowHeight 属性作为行高：

```
return rowHeight;
```

tableView: viewForHeaderInSection:方法用于构造节标题视图，在该方法中，我们使用 ExTableGroup 控件构造了节标题视图，并把控件的触摸事件委托给 expandButtonClicked:方法来处理。下面代码中，我们利用了 ExTableGroup 控件的 tag 属性，把节号放到 tag 属性中，这样节号就会随同控件传递给 expandButtonClicked:方法：

```
NSString* title=@"";
    // 根据节索引取出组 model
    id obj=[groups objectAtIndex:(section)];
    // 确保 groups 数组中存放的是 ExTableGroup 对象
    assert(obj->isa==[ExTableGroup class]);
    ExTableGroup* group=(ExTableGroup*)obj;
    if(group!=nil)
        title=group.title;
    UIView* headView = [[UIView alloc] initWithFrame:CGRectMake(0, 0, self.frame.
                        size.width, 40.0)];
    headView.autoresizesSubviews = YES;
    headView.autoresizingMask = UIViewAutoresizingFlexibleWidth;
    headView.userInteractionEnabled = YES;
    headView.hidden = NO;
    headView.multipleTouchEnabled = NO;
    headView.opaque = NO;
    headView.contentMode = UIViewContentModeScaleToFill;
    ExpandableButton *btn=[[ExpandableButton alloc]initWithFrame:CGRectMake(0,4,
                          320,32)];
    btn.tag=section;// 把节号保存到按钮 tag，以便传递到 expandButtonClicked 方法
    [btn setTitle:title];
    [btn addTarget:self action:@selector(expandButtonClicked:) forControlEvents:
                   UIControlEventTouchUpInside];
    [headView addSubview:btn];
    [btn setExpanded:group.isExpanded];
    [btn release];
    return headView;
```

tableView: cellForRowAtIndexPath:方法用于构造单元格视图，在该方法中，我们首先判断 ExTable 是否存在委托对象（即 exTableDelegate 属性），如果 ExTable 的委托不为空，则调用委托对象的 tableView:cellForRowAtIndexPath:方法。这是实际上是对 ExTable 类的一种有弹性的扩展。把一部分代码放到 ExTable 的委托对象中，当需要扩展 ExTable 的行为时（具体说，

应该是 tableView:cellForRowAtIndexPath:方法），我们就会给 ExTable 设定一个委托对象，然后在委托对象中重新定义 tableView:cellForRowAtIndexPath:方法。其中，在调用委托对象的指定协议方法时，我们使用到了 O-C 提供的反射机制（见第 3 章）：

```
if (exTableDelegate!=nil) {
        SEL sel=NSSelectorFromString(@"tableView:cellForRowAtIndexPath:");
        if([exTableDelegate respondsToSelector:sel]){
            return [exTableDelegate performSelector:sel
                   withObject:tableView withObject:indexPath];
        }
    }
```

接下来根据 indexPath 参数中的节号和行号，在 groups 数组中检索单元格数据（这里我们可以看出，理想中的行数据模型应当是一个字典对象）：

```
NSDictionary* d;
NSString *name, *phone;
// 根据节索引取出组 model
id obj=[groups objectAtIndex:(indexPath.section)];
// 确保 groups 数组中存放的是 ExTableGroup 对象
assert(obj->isa==[ExTableGroup class]);
ExTableGroup* group=(ExTableGroup*)obj;
d=[group.children objectAtIndex:indexPath.row];
name=[d objectForKey:@"name"];
phone=[d objectForKey:@"phone"];
UILabel *lblName, *lblPhone;
```

然后在单元格中构建两个 UILabel 中以显示单元格模型的数据：

```
UILabel *lblName, *lblPhone;
    static NSString* cellid=@"exTable_cellid";
    UITableViewCell* cell=(UITableViewCell*)[tableView dequeueReusableCellWith
                        Identifier:cellid];
    if (cell==nil) {
        cell=[[[UITableViewCell alloc]initWithStyle:UITableViewCellStyleDefault
              reuseIdentifier:cellid]autorelease];
      //标签：姓名
      lblName=[[UILabel alloc]initWithFrame:
             CGRectMake(20, 6, 100, 20)];
      lblName.numberOfLines=0;//不显示边框
      lblName.backgroundColor=[UIColor clearColor];//无背景色
      //lblName.font=[UIFont fontWithName:@"Arial" size:18];
      lblName.autoresizingMask=
              UIViewAutoresizingFlexibleWidth;//宽度自动扩展
      lblName.tag=1022;
      [cell addSubview:lblName];
```

```
        [lblName release];
        //标签：号码
        lblPhone=[[UILabel alloc]initWithFrame:
                CGRectMake(120, 6, 120, 20)];
        lblPhone.backgroundColor=[UIColor clearColor];
        //lblPhone.font=[UIFont fontWithName:@"Arial" size:18];
        lblPhone.textColor=[UIColor brownColor];
          lblPhone.tag=1023;
        [cell addSubview:lblPhone];
        [lblPhone release];

    }
```

默认的单元格数据模型是一个字典，包含两个键分别为 name 和 phone 的数据。这两个数据将在我们构建的两个 UILabel 中显示：

```
lblName=(UILabel*)[cell viewWithTag:1022];
lblPhone=(UILabel*)[cell viewWithTag:1023];
lblName.text=name;
lblPhone.text=phone;
```

最后，根据单元格模型中的 checked 值，我们在单元格右端显示一个小勾图标。如果 checked 值为 yes（表示用户选中该行），显示小勾，如果为 no，则不显示：

```
if([(NSNumber*)[d objectForKey:@"checked"] boolValue] )
    cell.accessoryType=UITableViewCellAccessoryCheckmark;
else
    cell.accessoryType=UITableViewCellAccessoryNone;
return cell;
```

tableView: didSelectRowAtIndexPath:方法响应行触摸事件。在这个方法里，首先通过 indexPath 获取用户选定行的行数据模型，设置其 checked 字段值（如果原先为 YES，设置为 NO；如果原先为 NO，则设置为 YES）。从这里你也可以看出，理想中的行数据模型应当是可写的（这里使用了 NSMutableDictionary 对象），可随时会根据用户的选择改变其 checked 字段。然后调用 Table View 的 reloadData 刷新表视图 ：

```
id obj=[groups objectAtIndex:(indexPath.section)];
    // 确保 groups 数组中存放的是 ExTableGroup 对象
    assert(obj->isa==[ExTableGroup class]);
    ExTableGroup* group=(ExTableGroup*)obj;
    NSMutableDictionary* d=[group.children objectAtIndex:indexPath.row];
    BOOL b=[(NSNumber*)[d objectForKey:@"checked"]boolValue];
    [d setObject:[[NSNumber alloc]initWithBool:!b] forKey:@"checked"];
    [_tableView reloadData];
```

值得注意的是接下来的代码。要检查 ExTable 的委托对象即 exTableDelegate 属性是否存在。

PDG

如果 ExTable 存在委托，则调用委托的 tableView:didSelectRowAtIndexPath:方法：

```
if (exTableDelegate!=nil) {
    SEL sel=NSSelectorFromString(@"tableView:didSelectRowAtIndexPath:");
    if([exTableDelegate respondsToSelector:sel]){
        [exTableDelegate performSelector:sel
        withObject:table withObject:indexPath];
    }
}
```

同 tableView:cellForRowAtIndexPath:方法一样，这也是对 ExTable 进行的一种弹性扩展，把一部分代码分散到委托类中。不同的是，不管委托对象是否存在，我们都为 tableView:didSelectRowAtIndexPath:方法提供了默认的行为（前一段代码）。而只有委托对象实现了指定方法的情况下，才会调用委托对象的代码。

注意：在这里，委托对象是否实现了协议方法不是必然的（即强制性的），因为协议中定义的两个方法（在 ExTableDelegate 中定义）都被声明为可选的（@optional）。因此我们使用了 respondsToSelector:方法测试委托对象是否实现指定方法，用以保证委托对象未实现该方法时并不会调用该方法。

最后一个值得注意的地方是 ExTableViewController 类。这也是程序的唯一 View Controller，显示了程序的主界面。由于我们把可折叠分组表视图的大部分代码分散到了前面三个类中，ExTableViewController 的代码看起来相当精炼，因为它只需包含调用 ExTableView 和 ExTableViewGroup 的代码。

ExTableViewController 有两个成员，一个是 tableView ，它是一个 ExTableView 对象；另一个是 NSSet 对象，用于存储用户选择（即电话号码，允许多选）。同时，在类的声明部分，我们也声明了对自定义协议 ExTableDelegate 的实现，这表明 ExTableViewController 有可能扩展（或定制）ExTableView 的某些行为。

首先是 loadView 方法。我们初始化了 ExTableView 对象，设置了它的 exTableDelegate 属性为 self，同时调用 setGroups 方法设置了 ExTableView 的 groups 属性：

```
self.view=[[UIView alloc]initWithFrame:CGRectMake(0, 20, 320, 460)];
tableView=[[ExTableView alloc]initWithFrame:CGRectMake(0, 0, 320, 460)];
[self.view addSubview:tableView];
tableView.exTableDelegate=self;
[tableView setGroups:[self buildGroups]];
```

在 setGroups 方法中，我们刷新表视图。这意味着 Table View 的所有 UITableViewDataSource 协议方法会被调用，于是我们在 ExTableView 中实现的获取行高、获取节标题视图、获取行视图的代码都被调用，这导致我们定制的分组表视图被渲染（见 ExTable 的 setGroups 方法）。

buildGroups 方法构建了分组表视图的数据模型（作为示例，我们构建了一个 2 节、4 行的数据）。该数据模型是一个数组对象，数组中包含的是封装为 ExTableGroup 对象的数据。这里，

PDG

我们只展示了第 1 节数据的构建代码，第 2 节的省略了——因为它只是把第 1 节的代码进行了简单拷贝：

```
NSMutableArray* groups=[[NSMutableArray alloc]init];
// 第 1 组
ExTableGroup* group=[[[ExTableGroup alloc]initWithTitle:@"同事" Expanded:NO]
      autorelease];
// 第 1 行,注意 dic 必须是可变的
NSMutableDictionary* dic=[NSMutableDictionary dictionaryWithObjectsAndKeys: @"王
      xx",@"name",@"13456897456",@"phone",nil];
[group.children addObject:dic];
// 第 2 行
dic=[NSMutableDictionary dictionaryWithObjectsAndKeys:@"lily",@"name",@"15969500562",
      @"phone",nil];
[group.children addObject:dic];
[groups addObject:group];
// 第 2 组，代码省略
...
return groups;
```

ExTableGroup 类前面已经介绍过，它包含了节的标题和行数据（即 children 属性）。children 属性也是一个数组，它把每一行用一个 NSMutableDictionary 封装，这个可变字典有 3 个字段：name、phone、checked。checked 字段为 BOOL 值，默认值为 NO，但当用户选择了该行时，checked 字段被修改为 YES，如果用户再次点击该行，checked 字段又会被修改为 NO。因此我们使用 NSMutableDictionary 而不是 NSDictionary。

接下来是 tableView:didSelectRowAtIndexPath:该方法是一个 ExTableDelegate 协议方法。ExTableDelegate 协议定义了两个方法，但它们都是可选的，我们只实现其中一个。这个方法用于响应用户对行的触摸行为：

```
assert(tableView.groups!=nil);
ExTableGroup* g=[tableView.groups objectAtIndex:indexPath.section];
NSDictionary* data=[g.children objectAtIndex:indexPath.row];
NSString* phone=[NSString stringWithString:[data objectForKey:@"phone"]];
NSNumber* checked=[data objectForKey:@"checked"];
if(checked.intValue!=0){// checked
    [selphone addObject:phone];
}else{// unchecked
    [selphone removeObject:phone];
}
NSLog(@"selphone:%@",selphone);
```

在这个方法中，通过检索 ExTableView 对象的 groups 属性，检索到行的行数据模型，然后判断行的勾选情况（checked 字段），将该行的电话号码添加到 selphone 或从 selphone 中移除。 程序最终运行效果如图 6-10 所示。

```
2012-04-10 11:10:54.618 ExTable[2971:207] checked:1
2012-04-10 11:10:54.619 ExTable[2971:207] selphone:{(
    13684997456
)}
2012-04-10 11:10:56.010 ExTable[2971:207] checked:1
2012-04-10 11:10:56.011 ExTable[2971:207] selphone:{(
    13684997456,
    13769504523
)}
```

图 6-10　程序运行和控制台输出效果

6.3　扩展 UIKit

UIKit 为我们提供了许多有用的组件，但仍然不能完全满足我们的应用。比如，用户不想使用 Object Library 中的 UISegmentControl ，要求用 Window 的单选按钮组代替它们；我们想实现一个可以支持多选的表视图，但无法在 Object Library 中找到它；在过去的项目工作中，我们积累了大量的自定义组件，但我们却无法在 IB 中重用它们——由于 IB 本身的限制，我们无法把这些组件封装到 IB 的组件库中（因为苹果的 IB Plug-in 编程不支持 iPhone）。

所有这些问题，让我们不得不去扩展 UIKit，以定制自己的可重用组件。在本节中，我们将通过一些实际的例子，讨论如何扩展 UIKit 组件。由于篇幅的限制，有的例子我们只能作大概介绍，余下的时间可能需要你自己阅读示例源代码，这些代码在本书光盘中。

当然，除了从源代码级别重用这些组件外，还会有更好的选择。后面还会介绍如何从二进制的角度封装我们的组件库。

6.3.1　扩展日期挑选控件

Cocoa 提供了 UIDatePicker 控件，但是对于程序员而言并不友好。其糟糕之处，莫过于无法指定其 frame，它的大小固定为 320×216（占据了近整个 iPhone 屏幕的一半），导致在 UI 设计时很难安排下这个“巨大”的东西。

我们自定义的日期挑选控件默认情况下显示为一个文本输入框，以文本的方式显示日期，当你试图编辑它时，会弹出一个巨大的 UIDatePicker 控件，如图 6-11 所示。

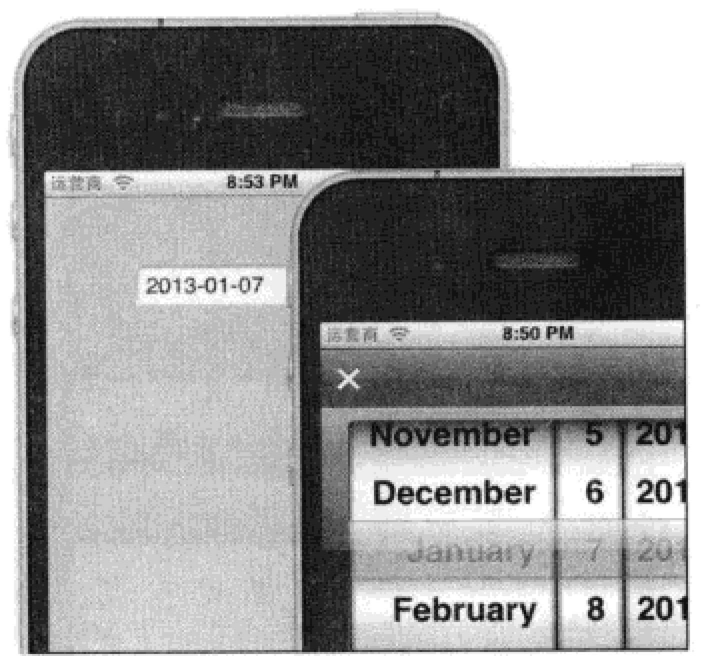


图 6-11　自定义的日期挑选控件

转动日期轮盘时，文本框中的值会作相应改变。选择好日期后，点击左上角的关闭按钮，即可关闭这个 UIDatePicker。

这个控件的使用非常简单。通常只需要通过下面的代码构造 DatePicker 对象，并用 addSubview 将其加到 UIView 中就可以了：

```
DatePicker* dp=[[DatePicker alloc]initWithFrame:CGRectMake (10,25,220,35)];
[self.view addSubview:dp];
```

如果想改变 DatePicker 的样式，比如显示时间而不是日期，可以修改其 datePickerMode 和 dateFormatter 属性：

```
dp.datePickerMode=UIDatePickerModeTime;
NSDateFormatter* df=[[NSDateFormatter alloc]init];
[df setDateFormat:@"HH:mm:ss"];
dp.dateFormatter=df;
```

如果要获取控件的日期时间值和字符串值，则可以使用控件的 date 属性和 textField.text 属性。

整个控件只由一个类 DatePicker 组成，源代码放在光盘“source/第 6 章”目录的 DatePicker 文件夹里。

DatePicker 有几个重要的成员属性：

```
UITextField* textField;
UIDatePicker* dp;
UIView* subview;
```

在 initWithFrame:方法里，控件实际上加入了两个 UIView。一个 UIView 就是 textField，textField 的大小刚好布满整个控件的 frame。一个 UIView 就是 subview，在 subview 我加入了

一个工具栏和一个 UIDatePicker 控件，subview 的大小为 320×480，占据了整个 iPhone 屏幕。这两个 UIView 的样子类似于图 6-11。由于在方法中 subview 并没有用 addSubview 加入，所以控件刚开始只显示 textField。

控件本身还实现了 UITextFieldDelegate 协议方法 textFieldShouldBeginEditing:，并且我们已经将 textField 的 delegate 属性设置为 self，因此当 textField 被点击（编辑事件发生）时，会触发 DatePicker 的 textFieldShouldBeginEditing:方法。在这个方法里，我们调用 addSubview 把 subview 对象加入到视图中。

同时在 initWithFrame:方法中，我们为日期挑选控件设置了一个事件代理，把值改变事件委托给 dateChanged:方法进行处理：

```
[dp addTarget:self action:@selector(dateChanged:) forControlEvents:UIControlEvent
   ValueChanged];
```

在 dateChanged 方法中，我们把改变后的日期值转变为字符串，然后填写到 textField 的 text 属性里。

当用户点击日期挑选控件上方工具栏的关闭按钮，我们用 removeFromSuperview 把 subview 从视图中移除：

```
[subview removeFromSuperview];
```

6.3.2 扩展单选按钮和复选按钮

我们先实现单选按钮，为了复用，不管单选还是复选按钮都是使用同一个类 实现（CheckButton），为了区别单选还是复选，我们用一个自定义枚举类型 CheckButtonStyle 属性 style 来区别，当其值设置为 CheckButtonStyleDefault 或 CheckButtonStyleBox 时，为复选按钮（见图 6-12）。

当其值设为 CheckButtonStyleRadio 时，为单选按钮（见图 6-13）。

图 6-12 复选按钮

图 6-13 单选按钮

整个控件由两个类构成，源代码放于光盘“source/第 6 章”目录的 CheckButton 目录中。按钮在 CheckButton 类中定义。CheckButton 首先定义了一个枚举：

```
typedef enum {
    CheckButtonStyleDefault = 0,
    CheckButtonStyleBox = 1,
    CheckButtonStyleRadio = 2
} CheckButtonStyle;
```

然后定义了三个重要的成员属性：

```
UILabel* label;
```

PDG

```
    UIImageView* icon;
    BOOL checked;
  id value;
```

UILabel 用于显示按钮的标题文本；UIImageView 用于显示按钮风格的方框或者园圈图片；BOOL 值用于代表按钮的选中状态；value 则代表按钮所包含的数据，value 的类型为 id 意味着 value 可以是任何 Cocoa 对象类型，如 NSString、NSNumber 或 NSDate，但不能是简单类型。

在 initWithFrame:方法里，首先按照默认样式（复选框）加入图片、标签到视图中。当 style 属性被改变，setStyle:方法被调用时，我们修改图片文件名 checkname 和 uncheckname。check.png 和 uncheck.png 使用于复选按钮样式，而 radio.png 和 unradio.png 使用于单选按钮样式。

CheckButton 继承自 UIControl 类，它可以捕获触摸事件。当控件被触摸，将调用控件的 clicked:方法处理。clicked:方法判断 delegate 属性是否为空，如果不为空，则委托 delegate 的 checkButtonClicked:方法处理。如果为空，则调用自己的 setChecked:进行处理。

setChecked:方法首先对控件的 checked 进行赋值，然后根据 checked 属性，改变图片的显示（checkname 是选中状态的图片文件，uncheckname 是未选中状态的图片文件）。同时检查 delegate 属性是否为空，如果不为空，还要委托 delegate 的 setText:方法和 setValue 方法进行处理。这是因为对于单选按钮组而言，还需要一些额外的处理，比如把选中的按钮与按钮组进行同步。

接下来就是单选按钮的实现。单选按钮比复选按钮要复杂一些。因为它有“组”的概念，在一个“组”中，只允许一个按钮被选中。我们用 RadioGroup 类实现了单选按钮“组”。RadioGroup 有如下成员属性：

```
@property (readonly)NSString* text;
@property (readonly)id value;
@property(retain,nonatomic)id delegate;
@property (retain,nonatomic)NSMutableArray* children;
```

- text 属性将会用“组”中已选的按钮的标签文本进行同步。
- value 属性将会用“组”中已选按钮的 value 属性进行同步。
- delegate 是一个委托，如果 RadioGroup 中有些事情不方便自身处理或者要从其他类增加一些“自选动作”，可以委托给 delegate 进行。说穿了，RadioGroup 的 delegate 就是一个预留给其他类扩展自身行为或能力的入口。通过这个入口，我们可以把位于类外部的代码连接到类的内部来执行。
- children 属性是一个数组，可以把任意多的 CheckButton 加入到这个数组中，方便单选按钮组进行管理。成员方法 add:允许你把一个 CheckButton 加入到 children 数组。

checkButtonClicked:方法实现了单选逻辑。首先它判断 id 参数 sender（应该是一个 CheckButton）的 checked 属性。如果这个 CheckButton 的 checked 属性为未选中，则接下来的事情就是要选中（用户触摸未选中的按钮表明用户就是想选中这个按钮）。先把按钮组中的按钮全部取消选择，再单独选中这个 CheckButton，就实现了所谓的“单选”功能。当然这一步是必须的，即把 CheckButton 的 text 和 value 复制到 RadioGroup 的 text 和 value 属性。

最后，如果委托对象 delegate 存在，则调用 delegate 的 selectChanged:方法。当然，使用了第 3 章讲过的反射机制，如果 delegate 未实现该方法，那么则方法也不会被调用。

好了，介绍了这么多，该来看看如何使用 CheckButton 类了。复选按钮的使用相对简单些：

```
CheckButton* cb=[[CheckButton alloc]initWithFrame:CGRectMake(20, 60, 260, 32)];
cb.label.text=@"★";
cb.value=[[NSNumber alloc]initWithInt:1];
[self.view addSubview:cb];
[cb release];
```

判断一个复选按钮是否被选中，使用 CheckButton 的 checked 属性，获取按钮的标题和值，使用 CheckButton 的 label.text 和 value 属性。

而复选按钮的使用就要复杂一些，需要 CheckButton 和 RadioGroup 一起使用：

```
rg=[[RadioGroup alloc]init];
CheckButton* cb=[[CheckButton alloc]initWithFrame:CGRectMake(20, 60, 260, 32)];
 [rg add:cb];
cb.label.text=@"★";
cb.value=[[NSNumber alloc]initWithInt:1];
cb.style=CheckButtonStyleRadio;
 [self.view addSubview:cb];
 [cb release];
//向 RadioGroup 中加入其他的单选按钮
...
```

要获取用户选择，使用 RadioGroup 的 text 和 value 属性即可。

6.3.3　扩展下拉列表框

UIKit 框架不提供下拉列表框控件，因为提供了 UIPickerView，为什么还要使用已经成为 Windows 标准控件之一的下拉列表框呢？苹果粉丝认为：这不是苹果的体验。但作为从 Windows 开发平台转移过来的程序员，只需要一个理由就足以反驳了：UIPickerView 太大了。真实情况就是这样的，放一个 UIPickerView，足够放三个下拉框都绰绰有余了。

跟前面自定义的日期挑选控件差不多，下拉列表框控件也是由多个控件组合起来的，包括一个文本框、一个下拉按钮和一个用于弹出列表的表视图，如图 6-14 所示。

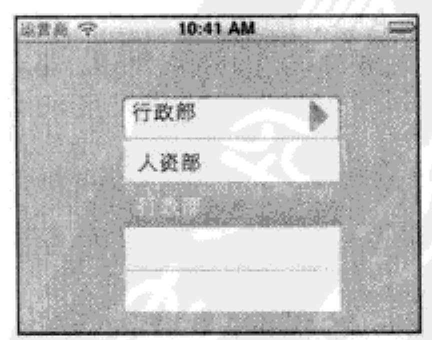


图 6-14　自定义的下拉列表框控件

源文件位于光盘“source/第 6 章”目录的 DropDownList 目录下，包括一个.h 和一个.m 文件（ DropDownList 类），两个.png 文件（分别用于显示下拉按钮的下拉和收起状态，使用时这两个文件应放置到项目的 Resources 文件组）。

DropDownList 的成员属性有许多，我们只关心重要的几个：

```
UITextField* textField;
UITableView* listView;
UIButton* btnDropdown;
BOOL editable;
BOOL showList;
NSDictionary* list;//下拉列表数据
NSArray* allKeys;//所有键
id<DropDownListDelegate> delegate;
```

前三个是用于组成 DropDownList 的 Cocoa 控件：一个可点击的 Text Field、一个显示列表的 Table View 和一个下拉按钮。后两个 BOOL 类型中，editable 表示文本框是否可接受输入，YES 为可以，NO 为不可以。showList 则表明下拉列表的拉出状态，如果列表是拉出的为 YES，否则为 NO。

list 成员用于保存下拉列表中要显示的数据。下拉列表通常表现出这样一个特征：显示内容和实际数据不一致。如果你在一个列表中选中了“市场部”，实际上它代表的数据却是 0104（该部门的 id）。因此列表项常用“键－值”对表示，键用于显示，值用于实际的数据或隐藏的内容。为了便于检索，我们用 allKeys 数组保存了下拉项数据的所有键。

最后一个属性是 delegate，用于实现 DropDowListDelegate。这是一个我们自定义的协议，要求必须实现方法-(void)selected: displayLabel:。这个方法会在 Table View 的某一行被选中时调用。如果我们想让下拉项被选中时执行一些自定义的行为，可以实现这个代理方法。

然后查看 DropDownList 的实现部分。打开 DropDownList.m。

首先是 initWithFrame:方法，它用于初始化一些成员变量，尤其是初始化用于下拉列表的数据。在 initWithFrame 方法中，我们任意准备了两个“键－值”对作为列表数据。最后一行代码调用了 drawView 方法。其实是把绘制控件的代码移到了 drawView 方法里。drawView 方法绘制了文本框、下拉按钮和表视图，当然一开始表视图被隐藏住了：

```
listView.hidden=YES;
```

setList:方法是 list 属性的 setter 方法。它把一个 NSDictionary 对象复制给 list 属性，把 NSDictionary 对象的 allKeys 赋值给 allKeys 属性。然后刷新表视图。这样表视图将根据新的字典数据显示列表内容。

dropdown 方法在下拉按钮被触摸时响应。这个方法有两个作用，当下拉列表是显示时，它隐藏下拉列表；当下拉列表隐藏时，它显示下拉列表（通过调用 setShowList:方法），同时切换按钮的状态图标。

setShowList:方法是 BOOL 值 showList 的 setter 方法。只要我们调用这个方法，下拉列表

会如实地显示或隐藏。它要做许多事情：改变 showList 变量值；根据当前的 showList 值重设控件的 frame 大小（下拉列表的拉出和收起状态下，控件所占据的空间大小是不一样的）；隐藏或显示表视图；调用 setNeedDisplay，以便重绘控件（因为表视图本身没有边框，我们要用 Quartz API 为表视图绘制一个矩形边框）。

当调用 setNeedDisplay 方法时，drawRect:方法会被调用。我们在方法中调用 Quartz API 在表视图周围绘制了一个矩形，这样当下拉列表被拉出时，不会显示一个缺少了边框的表视图。

setSelectedIndex:方法允许我们从下拉框中选择一个默认的项。比如当视图一开始显示时，我们可以用这个方法在文本框中显示一个默认的下拉项。

接下来是表视图的 UITableViewDataSource 协议方法。这些代码你应该很熟悉了，我们在前面 6.2.5 节"可折叠的分组表视图"中解释过这些代码。其中，tableView:cellForRowAtIndexPath:方法定义了表视图的单元格应当如何被渲染。

最后是表视图的 UITableViewDelegate 协议方法 tableView: didSelectRowAtIndexPath:方法。这个方法中，我们把用户的选择项分别复制到控件的 value 属性和文本框里，然后收起下拉列表。如果委托对象 delegate 不为空，还会调用 delegate 的 selected:displayLabel:方法。这个方法在协议 DropDownListDelegate 中定义。如果你还想自己捕获 DropDownList 控件的下拉列表的行选中事件，那么你应该使用这个协议。

DropDownList 的使用方法如下代码所示：

```
DropDownList *boxDept=[[DropDownList alloc]initWithFrame:
    CGRectMake(105, 124, 172, 35)];
boxDept.placeholder=@"请选择部门";
NSDictionary *d=[NSDictionary dictionaryWithObjectsAndKeys:
    @"行政部",@"0100",@"人资部",@"0200",nil];
boxDept.list=d;
[boxDept setSelectedIndex:1];
[self.view addSubview:boxDept];
```

你可以使用 DropDownList 的 data 属性获得选中项的 key 值（数据），DropDownList 的 text 属性获取选中项的文本值（显示）。

6.3.4　封装自己的控件库

经过前面的介绍，相信你已经积累了一些自定义控件方法了。正如前面所说，由于 IB 的限制，我们不能把这些控件加入到 IB 的控件库中。难道我们只能在"源代码"级别重用这些控件吗？不，我们可以把它们封装到静态库里。静态库是编译为二进制的可执行文件，从而脱离了只能从"源代码"上重用组件的尴尬。下面，我们以 CheckButton 为例，演示如何把该组件封装为 iOS 静态库。

1. 实现静态库

新建项目，项目模板选择"Framework & Library"下的"Cocoa Touch Static Library"，给

项目命名，例如：MyLibrary。

复制 CheckButton 组件的 4 个源文件：CheckButton.h、CheckButton.m、RadioGroup.h、RadioGroup.m 到项目的 Classes 目录下，同时把 CheckButton 的 4 个资源文件：check.png、uncheck.png、radio.png、unradio.png，复制到项目文件夹。

按下 Command+B 编译，在 Products 目录下即产生一个 .a 文件。

2. 添加资源束

静态库中并不能包含资源文件，虽然我们已经把 4 个资源文件（.png 文件）拷贝到静态库项目中，但实际上这些.png 是不会添加到 target 的，也就是说编译结果中并不包含这些资源，因此如果此时调用静态库，所有的资源（字符串、图片）都是缺失的。我们可以把资源建立成单独的束（Bundle）。

新建项目，选择模板“Mac OS X→Framework & Library→Bundle”，命名为：MyLibraryBundle。然后把上面 4 个.png 文件拷进 Resources 中去。编译，生成 MyLibraryBundle.bundle 文件。

返回静态库项目，新建一个类：Utils。编辑 Utils.h，如下代码所示：

```
// Utils.h 文件
#define MYBUNDLE_NAME @"yhyLibraryBundle.bundle"
#define MYBUNDLE_PATH [[[NSBundle mainBundle] resourcePath] stringByAppending
       PathComponent: MYBUNDLE_NAME]
#define MYBUNDLE [NSBundle bundleWithPath: MYBUNDLE_PATH]
NSString* getMyBundlePath(NSString* filename);
// Utils.m 文件
#import "Utils.h"
NSString* getMyBundlePath(NSString* filename)
{

    NSBundle * libBundle = MYBUNDLE;
    if( libBundle && filename ){
        NSString* s=[[libBundle resourcePath] stringByAppendingPathComponent:
                    filename];
        NSLog(@"%@",s);
        return s;
    }
    return nil;
}
```

函数 getMyBundlePath 可以取得具体资源在 MyLibraryBundle 这个束中的绝对文件路径，如：

```
/Users/kmyhy/Library/Application Support/iPhone Simulator/4.2/Applications/
     8213652F-A47E-456A-A7BB-4CD40892B66D/yhyLibTest.app/yhyLibraryBundle.
     bundle/Contents/Resources/radio.png
```

同时，修改 CheckButton.m 中的代码，导入 Utils.h 头文件，把其中获取图片的代码由

imageNamed 修改为 imageWithContentsOfFile，如：

```
[icon setImage:[UIImage imageWithContentsOfFile:getMyBundlePath(checkname)]];
```

即通过绝对路径读取图片资源。

除了这种方法，我们还可以有一个简单办法，就是把 4 个资源文件直接拷贝到你调用静态库的应用项目中（不需要修改静态库代码）。

上述两个项目代码可以在光盘“source/第 6 章”目录中找到。

3. 静态库的使用

新建 Empty Application 项目，给项目命名，如 MyLibraryTest。把静态库项目的 MyLibrary.xcodeproj 文件拖拽到当前项目。打开 Target 的 Build Phases 窗口，点击 Target Dependencies 里面的“+”号按钮，在弹出窗口中选择将 MyLibrary 添加为本项目的依赖项目，然后点击 Add 按钮，如图 6-15 所示。

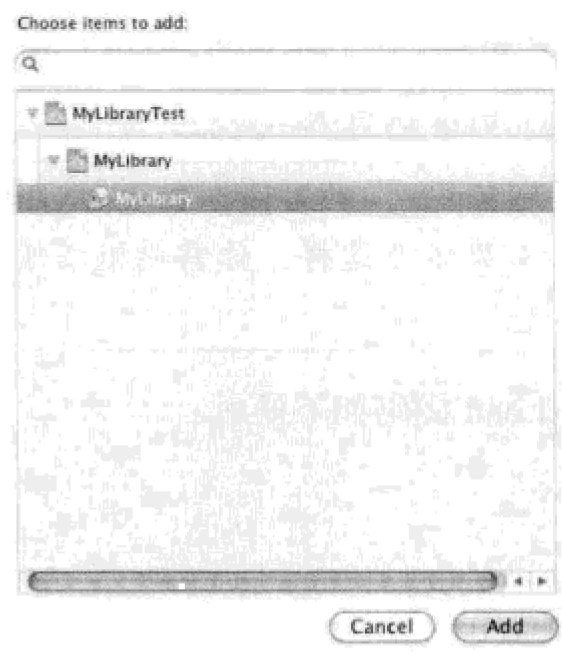


图 6-15　添加项目依赖

添加结束，Target 的 Target Dependencies 栏中将显示出最终结果，如图 6-16 所示。

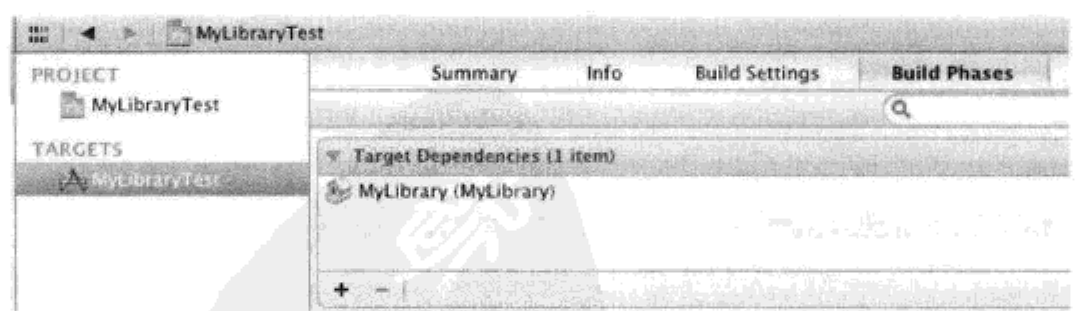


图 6-16　Target Dependencies 显示了我们添加的依赖

这样，我们就可以在项目中直接对依赖项目进行编辑了。同时，在编译 Target 时也会自动编译依赖项目，并将依赖项目的目标文件连接到 Target 中。这样做对于在调试中及时发现和修改 MyLibrary 中错误是很有必要的。

现在，我们新建一个 UIViewController subclass（勾选“With XIB …”选项），名为 RootViewController，用于作为程序运行的主窗体。

然后打开 AppDelegate.m，在 application: didFinishLaunchingWithOptions:方法中加入代码（别忘了导入 RootViewControler.h）：

```
self.window.rootViewController=[[RootViewController alloc]init];
```

打开 RootViewController.m，让我们来看看如何使用 MyLibrary 静态库。首先在文件头部导入相关的头文件：

```
#import "RadioGroup.h"
```

这里 Xcode 会提示一个错误：file not found。为什么会这样呢？因为 Xcode 找不到 MyLibrary 的头文件。MyLibrary 的头文件是位于 MyLibrary 项目目录下，而 Xcode 默认只会搜索当前项目的文件夹，当然找不到了。其实我们只需要告诉 Xcode，连接时不仅仅搜索自己的文件夹，还可以在额外的文件夹中搜索就可以了。

打开项目的 Build Settings（注意，不是 Target 的 Build Settings），找到 Header Search Paths，添加字符串“../MyLibrary”，这个错误就解决了。

注意：这里我们使用了相对路径“../”的写法，表示 MyLibrary 项目目录和 MyLibraryTest 项目目录应该位于同一目录下。如果不是这样，你需要根据情况自己修改 Header Search Paths 字符串。

接下来，我们修改 viewDidLoad 代码如下所示：

```
[super viewDidLoad];
    //单选按钮组
    RadioGroup* rg=[[RadioGroup alloc]init];
    //第 1 个单选按钮
    CheckButton* cb=[[CheckButton alloc]initWithFrame:CGRectMake(20, 60, 260, 32)];
    //把单选按钮加入按钮组
    [rg add:cb];
    cb.label.text=@"单选框 1";
    cb.value=[[NSNumber alloc]initWithInt:1];
    //把按钮设置为单选按钮样式
    cb.style=CheckButtonStyleRadio;
    //加入视图
    [self.view addSubview:cb];
    [cb release];//add 后，会自动持有，可以释放
    //第 2 个单选按钮
    cb=[[CheckButton alloc]initWithFrame:CGRectMake(20, 100, 260, 32)];
    [rg add:cb];
    cb.label.text=@"单选框 2";
    cb.value=[[NSNumber alloc]initWithInt:2];
    cb.style=CheckButtonStyleRadio;
```

```
    [self.view addSubview:cb];
    [cb release];
    //复选按钮
    cb=[[CheckButton alloc]initWithFrame:CGRectMake(20, 140, 260, 32)];
    cb.label.text=@"复选框 1";
    cb.value=[[NSNumber alloc]initWithInt:3];
    cb.style=CheckButtonStyleBox;
    [self.window addSubview:cb];
    [cb release];
```

在这段代码里，我们在视图中加入了一个单选按钮组（包含两个单选按钮）和一个复选按钮。如果你此时编译程序，会出现两个连接错误：

```
Undefined symbols for architecture i386:
  "_OBJC_CLASS_$_RadioGroup", referenced from:
     objc-class-ref in RootViewController.o
  "_OBJC_CLASS_$_CheckButton", referenced from:
     objc-class-ref in RootViewController.o
```

哦，我们忘记把静态库文件（.a 文件）加到项目中了。打开 Target 的 Build Phases 面板，在“Linke Binary With Libraries”区，点击“+”按钮，在弹出窗口中找到 Workspace 下面的 libMyLibrary.a 并选中，然后点击 Add 按钮，如图 6-17 所示。

⌘+R 编译并运行。运行结果如图 6-18 所示。

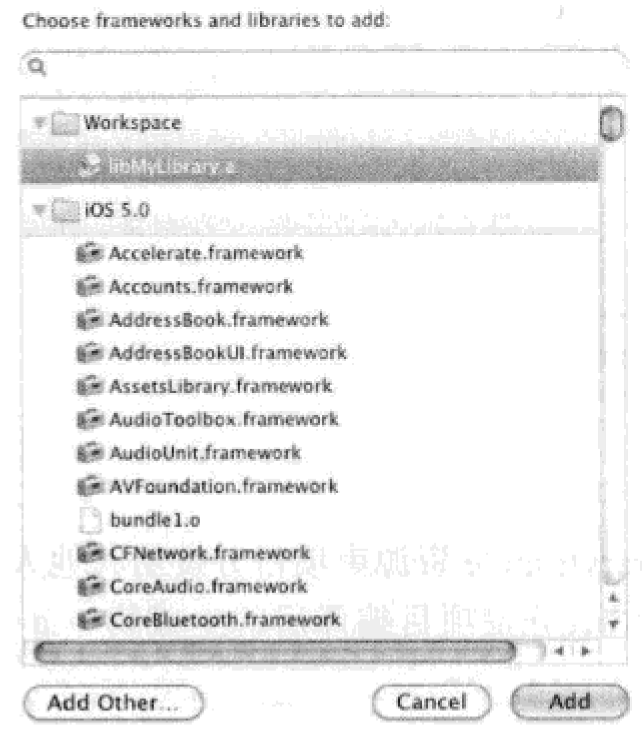


图 6-17　添加框架或库

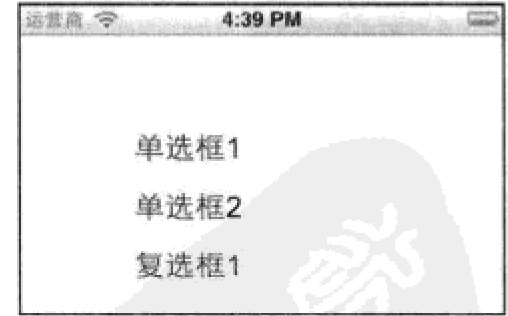


图 6-18　程序运行结果

为什么文字标签前空空如也？原来我们忘记把静态库的资源束添加到项目中了。

将 MyLibraryBundle.xcodeproj 从 Finder 拖到 Xcode 中。同样，为项目建立一个到 MyLibraryBundle 的项目引用（依赖）。这当然也便于在调试过程中修改 MyLibraryBundle 的资

源（它没有代码）。如图 6-19 所示，我们的 MyLibraryTest 项目现在有两个项目依赖了。

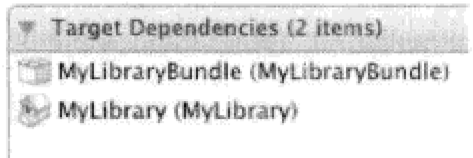


图 6-19　我们添加了一个对项目 MyLibraryBundle 的依赖

在 Project Navigator 窗口，打开 MyLibraryBundle.xcodeproj 下的 Products 目录，你会看到有一个 MyLibraryBundle.bundle 文件，如图 6-20 所示。

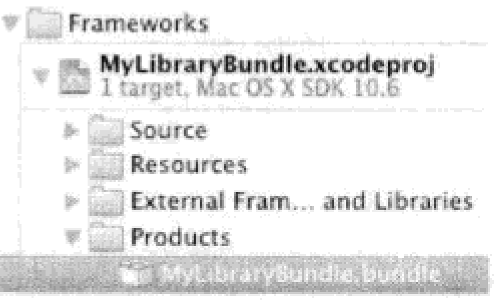


图 6-20　MyLibraryBundle.xcodeproj 目录下的.bundle 文件

选中它，点击右键，选择“Show in Finder…”，将会在 Finder 中显示它实际所在的位置。我们把它又从 Finder 中拖到 Xcode 中，在弹出的对话框中点击 Finish（不要选择“Copy items…”）。此时，再次编译并运行程序，运行结果如图 6-21 所示。

图 6-21　程序最终运行结果

4. 分发静态库

经过一番测试，终于大功告成。

现在，你可以把 MyLibrary 静态库项目和 MyLibraryBundle 资源束项目分发给其他人了。当然，如果你不愿意和别人共享源代码，那么也可以只分发项目编译后的生成物：.a 文件和.bundle 文件。

6.4　翻页控件和翻页控制器

大家应该熟悉 iPhone Home 屏上的那几个小点点吧？那就是 iPhone 用来控制翻页的翻页控件 UIPageControl，如图 6-22 所示。

图 6-22 iPhone Home 屏下方有一个翻页控件

这一节，我们就来了解一下翻页控件 UIPageControl，以及与它酷似的另一个 UI 组件翻页控制器 UIPageViewController。

UIPageControl 只是一个简单的翻页控件，它提供了简单的 API 让你实现多个视图间的切换。说它简单，是因为程序员在用它时无法偷更多的懒。他们必须自己实现用户触摸手势的识别，以及翻页动画的实现。优点是，你可以定制得更深，从而实现更大程度的自定义行为。比如你不想让“扫动”手势被识别为翻页，而想让两根手指被识别为向前翻页，五根手指被识别为后翻页，等等。

UIPageViewController 是一个特别的视图控制器（SDK 5.0 以后）。也许是苹果公司看到了程序员使用 UIPageControl 时的尴尬，在 Xcode4.2(SDK5.0)中，增加了 UIPageViewController。使用 UIPageViewController 可实现与 UIPageControl 相同的功能，但只需要程序员书写很少的代码。UIPageViewController 还允许你通过简单的参数设置来定制翻页时的动画效果，不需要你书写额外的 Core Animation 代码。此外，与 UIPageControl 不同，UIPageViewController 是通过视图控制器（UIViewController）而不是视图(UIView)来进行翻页导航的。一般而言，如果你不需要进行太深层的定制，使用 UIPageViewController 代替 UIPageControl，能极大地简化你的代码。

6.4.1 UIPageControl

新建项目 PageControlDemo。在 Resources 组中添加几张图片，在这里我随便找了几张动物的图片，你也可以另外找几张。这个项目放于光盘“source / 第 6 章/PageControlDemo”目录下，如果你实在找不到一些有趣的图片，可以直接使用示例程序中提供的图片。

打开应用程序代理类，在 application:didFinishLaunchingWithOptions:方法中加入以下代码：

```
self.window.rootViewController=[[RootViewController alloc]init];
```

加入框架 QuartzCore.framework（还记得如何加入框架吗？在第 4 章中讲过）。然后新建 UIView Controllersubclass 类 RootViewController。打开 RootViewController.xib，在视图上拖入一个 UIPageControl 和一个 UIView，如图 6-23 所示。

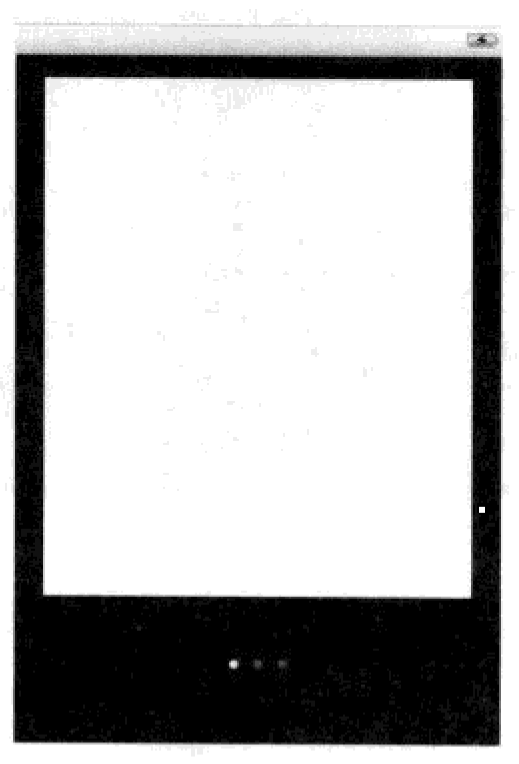


图 6-23 RootViewController 中放入了一个 UIView 和一个 PageControl

在接口中声明两个出口，将 UIPageControl 和 UIView 连接到出口上。

在 RootViewController.m 中，编辑 init 方法，把我们要用到的三个图片的文件名放入到 pages 数组中：

```
-(id)init{
    if (self = [super init]) {
        pages=[[NSMutableArray alloc]initWithObjects:@"Dolphin.png",
               @"Butterfly.png",
               @"Hippopotamus.png",
               nil];
    }
    return self;
}
```

编辑 viewDidLoad 方法，在 pageView 中用 addSubview 加入两个 UIImageView。为什么只用两个 UIImageView 而不是更多？我们的 pages 数组中不是明明有三张图片吗？

道理很简单，就像你在翻一本连环画册，不管有多少页图片，对于你来说，永远只能看到两页图片：最表面的一页，和它下面的一页。在还没有翻页之前，你只能看到最上面的一页，当最上面的那页开始翻动，第二页才开始逐渐可见，一直到完全翻开，第一页完全翻到后面去了（隐藏了），第二页成为了当前最上面的一页。现在，你又只能看到一页图片了。如果你不停地翻页，这个过程会不断重复，但最多，你只能同时看到两页图片。其他页是你无法看到的，

因为每次只能翻开一页而已。

当然，在还没有翻开任何图片之前，我们把最下面一页（在 subviews 中的索引为 0）默认加载为第一张图片：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    for (int i=0;i<2;i++) {
        [vwPage addSubview:[[UIImageView alloc] initWithFrame:vwPage.bounds]];
    }
    [[[vwPage subviews] lastObject] setImage:[UIImage imageNamed:
        [pages objectAtIndex:0]]];
}
```

由于 UIView 没有背景色，最上面一层又没有加载任何内容，于是默认情况下，我们会看到下面一页的图片（一只海豚图片）。

现在，我们声明一个 IBAction 方法 pageTurn，并且让它和.xib 文件中的 UIPageControl 对象连接（使用 UIPageControl 的 valueChanged 事件）：

```
-(IBAction)pageTurn:(id)sender{
    int fromPage=previousPage;
    int toPage=pageCtr.currentPage;
    CATransition *transition;// ❶
     if (fromPage!=toPage) {
         if(fromPage<toPage){// ❷
             transition=[self getAnimation:kCATransitionFromLeft];// ❸
         }else{
             transition=[self getAnimation:kCATransitionFromRight];// ❹
         }
         UIImageView *newImage=(UIImageView *)[[vwPage subviews] objectAtIndex:
                 0];// ❺
         [newImage setImage:[UIImage imageNamed:[pages objectAtIndex:toPage]]];// ❻
         [vwPage exchangeSubviewAtIndex:0 withSubviewAtIndex:1];// ❼
         [[vwPage.subviews objectAtIndex:0] setHidden:YES];// ❽
         [[vwPage.subviews objectAtIndex:1] setHidden:NO];// ❾
         [[vwPage layer] addAnimation:transition forKey:@"pageTurnAnimation"];//❿
     }
    previousPage=pageCtr.currentPage;
}
```

代码说明：

❶ 声明一个 CATransition 对象。要使用动画特效，最简单的方法是使用 Core Animation。我们决定显式地使用 CATransition 创建动画。并在后面调用 getAnimation 方法获得一

个 CATransition 动画特效（getAnimation 方法后面介绍，可参考第 11 章）。

❷ 判断动画的方向。因为用户可能有两种使用 Page Control 的方式：向前翻页、向后翻页。

❸ 如果是向后翻页，动画转换方式为 kCATransitionFromLeft。

❹ 如果是向前翻页，动画转换方式为 kCATransitionFromRight。

提示：关于使用 kCATransitionFromLeft 还是 kCATransitionFromRight 并不是绝对的，由你自己决定使用哪种方式更合适。

---

❺ 从 vwPage 中查找下面的那张图片（Image View）。

❻ 让这个 Image View 重新加载即将要显示的图片。

❼ pageView 的 subViews 中，让两个 View 的索引交换（原来索引为 0 的变成 1，原来索引为 1 的变成 0）。

❽ 索引为 0 的 Image View 隐藏。因为索引已经交换，所以其实是原来的旧图片（原来索引为 1）被隐藏。

❾ 索引为 1 的 Image View 设置为显示。因为交换过索引，所以其实是新加载的图片（原来索引为 0）被显示出来。

❿ 将前面配置好的 CATransition 用 addAnimation 方法添加到 pageView 的 layer 中去，这将播放这个动画。任何 UIView 的子类都有 layer 属性，所以我们可以利用 layer 的 addAnimation 方法为 UIView 附加任何动画。

接下来看看 getAnimation 方法。这个方法根据用户手指滑动的方向返回一个 CATransition 转换动画：

```
-(CATransition *) getAnimation:(NSString *) direction
{
    CATransition *animation = [CATransition animation];
    [animation setDelegate:self];
    [animation setType:kCATransitionPush];
    [animation setSubtype:direction];
    [animation setDuration:1.0f];
    [animation setTimingFunction:[CAMediaTimingFunction functionWithName:kCAMedia-
            TimingFunctionEaseInEaseOut]];
    return animation;
}
```

这是一个简单的翻页动画（其 type 被设定为 kCATransitionPush），而 subtype 属性则指定了翻页开始的方向：上、下、左、右。duration 属性是动画转换的时间；timingFunction 属性指定转换过程中要使用的特效，使用常量字符串指定 5 种不同效果。程序最终运行效果如图 6-24 所示。

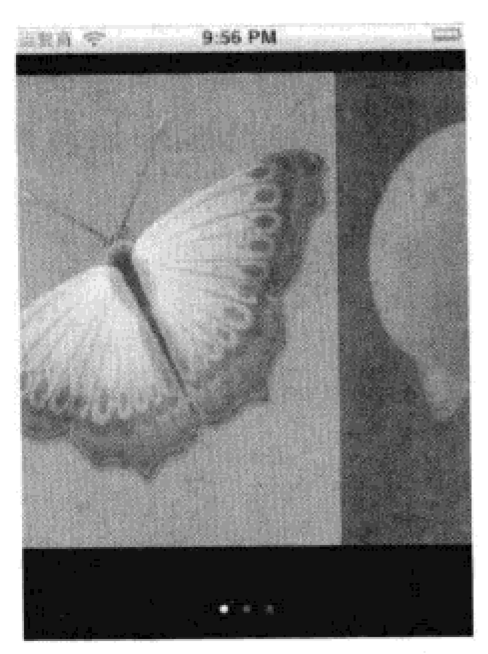


图 6-24 程序运行效果

6.4.2 UIPageViewController

要实现动画翻页效果，在 SDK 5.0 中还有很方便的办法，那就是使用 UIPageViewController。UIPageViewController 是 SDK 5.0（Xcode4.2）中新增的视图控制器，可以很方便地实现翻页动画效果并支持对扫动手势的识别。而使用 UIPageControl，要达到能够识别手势的目的，需要自己实现，还需要做许多额外的工作[1]。

现在，我们就来看看 UIPageViewController 的使用。示例项目位于光盘“source/第 6 章/pageViewControllerDemo”目录。

新建 Single View Application 项目，不要使用 Use Storyboard 选项。项目默认提供了一个 ViewController 类，继承自 UIViewController，默认 AppDelegate 在启动时会加载这个 View Controller 显示：

```
self.viewController = [[[ViewController alloc] initWithNibName:@"ViewController"
     bundle:nil] autorelease];
   self.window.rootViewController = self.viewController;
```

我们需要一个额外的 View Controller。新建 UIViewController subclass，命名为 contentViewController（使用“With XIB…”选项）。打开 contentViewController.xib，在其中放入一个 UIWebView。在 contentViewController.h 中增加两个属性：

```
@property (strong, nonatomic) IBOutlet UIWebView *webView;
@property (strong, nonatomic) id dataObject;
```

其中：

- webView 声明为 IBOutlet 出口，用于连接到 UIWebView 对象。

[1] 可以参考《iPhone 开发秘籍》中 Henry Yu 对 SwipeView 的实现。

PDG

- dataObject 用于引用将会在 UIWebView 中显示的 HTML 内容。

我们先到 contentViewController.xib 中创建一个 UIWebView 到 webView 出口的连接。然后打开 contentViewController.m 文件，加入@synthesize 语句并编辑 viewWillAppear:方法如下：

```
#import "ViewController.h"
@implementation ViewController
@synthesize webView,dataObject;
...
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    [webView loadHTMLString:dataObject
                baseURL:[NSURL URLWithString:@""]];
}
...
@end
```

contentViewController 用于负责显示每一页的 HTML 内容到一个 Web View 中（这里假设，不管我们有多少页，都用只用 contentViewController 来显示）。

接下来，我们需要编辑 ViewController 类，让它拥有一个 UIPageViewController 实例，并为该 UIPageViewController 对象提供必需的数据。

首先，打开 ViewController.xib 文件，并从 Object Library 拖入一个 UIPageViewController 到 xib 文件中。选中该 UIPageViewController 对象，打开 Attributes Inspector 面板，设置 UIPageViewController 的一些属性，如图 6-25 所示。

图 6-25 配置 UIPageViewController 属性

需要注意两个属性：Spine Location 和 Wants Full Screen。Spine Location 是指 Spine（书脊）所在的位置，它有 3 个可能取值：min、mid、max，分别对应左、中、右 3 个位置。Wants Full Screen 选项是指让 UIPageViewController 布满整个屏幕。IB 中提供的 UIPageViewController 有一个 Bug，如果你不勾选 Wants Full Screen 选项，则 UIPageViewController 最多只能占据 440

像素的高度，在顶部状态栏和 UIPageViewController 之间会出现一个 20 像素高的空间（如图 6-26 所示）。

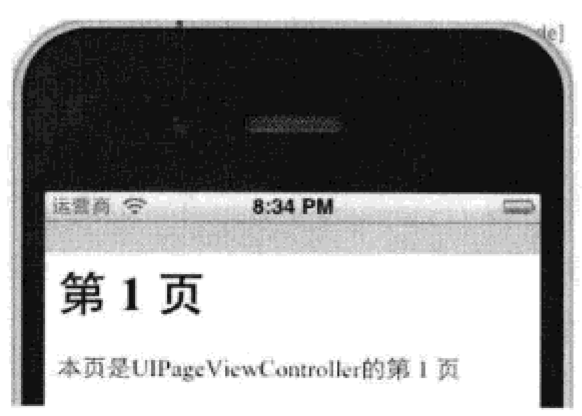


图 6-26　在 UIPageViewController 上方会空出 20 像素

打开 ViewController.h，导入 contentViewController.h，加入两个属性声明 pageController 和 pageModel：

```
#import "contentViewController.h"
@interface ViewController : UIViewController<UIPageViewControllerDataSource>
@property (strong, nonatomic)IBOutlet UIPageViewController *pageController;
@property (strong, nonatomic) NSArray *pageModel;
@end
```

接口声明了对 UIPageViewControllerDataSource 协议的实现。其中：

- pageController 即 UIPageViewController 对象的引用。
- pageModel 是一个数组用于为数个页面提供数据（HTML 字符串）。

将 xib 中的 UIPageViewController 连接到出口 pageController，请参考第 5 章 5.4.4 节，了解连接出口的方法。

打开 ViewController.m，实现一个 fillPages 方法：

```
- (void) fillPages {
    NSMutableArray *pageStrings = [[NSMutableArray alloc] init];
    for (int i = 1; i < 11; i++)
    {
        NSString *contentString = [[NSString alloc]
                initWithFormat:@"<html><head></head><body><h1> 第  %d 页 </h1><p>
                本页是 UIPageView Controller 的第 %d 页</p></body></html>", i, i];
        [pageStrings addObject:contentString];
    }
    pageModel = [[NSArray alloc] initWithArray:pageStrings];
}
```

fillPages 方法只干了一件事，即用一些 HTML 字符串初始化并填充了 pageModel 数组，用于作为 UIPageViewController 中各个页面的内容。

在 viewDidLoad 方法中加入调用 fillPages 方法的代码：

```
[self fillPages];
```

现在，需要把 UIPageViewController 对象的数据源委托给 ViewController，这需要一些步骤。首先，在 ViewController.xib 中选择 UIPageViewController 对象，打开 Connections Inspector，建立一条 dataSource 到 File's Owner 的连接。这个连接等同于语句：pageController.dataSource=self）。然后，需要在 ViewController 类中实现 UIPageViewController 的数据源方法。

UIPageViewController 的数据源方法有两个：一个 是返回当前显示的 view controller 之后的 view controller，而另一个是返回当前显示的 view controller 之前的 view controller。 因为我们使用 pageViewController 作为 UIPageViewController 的数据源，所以我们需要在 pageViewController 中实现这两个数据源方法。在此之前，我们先实现 2 个在数据源方法中会用到的便利方法：

```
- (contentViewController *)viewControllerAtIndex:(NSUInteger)index {
    if (([self.pageModel count] == 0) ||
        (index >= [self.pageModel count])) {
        return nil;
    }
    contentViewController *dataViewController =
          [[contentViewController alloc]
          initWithNibName:@"contentViewController"
          bundle:nil];
    dataViewController.dataObject =
           [self.pageModel objectAtIndex:index];
    return dataViewController;
}
- (NSUInteger)indexOfViewController:(contentViewController *)viewController {
    return [self.pageModel
           indexOfObject:viewController.dataObject];
}
```

说明如下：

- viewControllerAtIndex: 方法首先检查有效页数是否<0（用户不可能回到第 1 页以前）或者要检索的页数已经超出了 pageModel 数组的实际数目。如果 index 参数有效，就创建一个新的 ViewController 实例并将其 dataObject 属性设置为相应的 pageModel 数组元素（HTML 字串）。
- indexOfViewController 方法指定一个 viewController 作为参数，并返回这个 viewController 的索引。它使用 view controller 的 dataObject 属性在 pageModel 组元素中检索其所处的位置索引 。

现在来看两个数据源方法。我们使用这两个便利方法返回当前 view controller “之前”和“之后”的 view controller：

```
- (UIViewController *)pageViewController: (UIPageViewController *)pageViewController
   viewControllerBeforeViewController: (UIViewController *)viewController {
   NSUInteger index = [self indexOfViewController:
                     (contentViewController *)viewController];
   if ((index == 0) || (index == NSNotFound)) {
       return nil;
   }
   index--;
   return [self viewControllerAtIndex:index];
}
- (UIViewController *)pageViewController: (UIPageViewController *)pageViewController
   viewControllerAfterViewController:(UIViewController *)viewController {
   NSUInteger index = [self indexOfViewController:
                     (contentViewController *)viewController];
   if (index == NSNotFound) {
       return nil;
   }
   index++;
   if (index == [self.pageModel count]) {
       return nil;
   }
   return [self viewControllerAtIndex:index];
}
```

下一步，就是修改 viewDidLoad 方法代码，在 ViewController 加载时，将 UIPageView Controller 对象加到 self.view 中，以便显示 UIPageViewController：

```
- (void)viewDidLoad
{
    super viewDidLoad];
    [self fillPages];
    contentViewController *initialViewController =
            [self viewControllerAtIndex:0];              // ❶
    NSArray *viewControllers =
            [NSArray arrayWithObject:initialViewController];     // ❷
            [pageController setViewControllers:viewControllers
            direction:UIPageViewControllerNavigationDirectionForward
            animated:NO
            completion:nil];        // ❸
    [[self view] addSubview:[pageController view]]; // ❹
}
```

我们来分析一下这段代码：

❶ 这句代码调用 viewControllerAtIndex 便利方法，获得了一个 contentViewController（它的 dataObject 属性被配置为指定索引处的 pageModel 元素——HTML 字符串）。

❷ 把❶中获取的 contentViewController 对象放到一个数组。这个数组将用于 UIPageViewController 的 viewControllers 属性。注意，只需要一个 contentViewController。因为 page controller 被设定为一次只显示 1 页（单面）。如果将 page controller 配置为 2 页（spine 位于中央）或者双面，则需要创建 2 个 content view controller 并放入数组。

❸ 设置 UIPageViewController 的 viewControllers 属性，并将导航方向设置为向前模式。

❹ 将 UIPageViewController 加到当前视图（self.view）。

点击 Run 运行程序，第 1 页将显示出来。从右到左滑动屏幕，将翻到下一页，做相反方向滑动则翻到上一页，如图 6-27 所示。

图 6-27 程序运行效果

6.5 本章小结

本章介绍了 UIKit 框架中一些特别的主题，包括：UIKit 中关于多视图间的导航技术——UITabBarController、UINavigationController 和 addSubview 方式；表视图的使用，包括普通表视图、分组表视图及可折叠分组表视图。

当然，我们提到了 UIKit 框架的不足，介绍了如何扩展和自定义自己的控件库。如日期挑选控件 DatePicker、单选按钮和复选按钮、下拉文本框等。我们把多个 UIKit 框架中自带的控件使用简单“组合”的方式构建出新控件。这是一种高级的扩展组件方式。当然，如果你对 Quartz 2D 和 Core Animation 有足够深入的了解，就可以用更底层的方式自己绘制控件（这可能专门要用另外的一本书来介绍了）。

除了以源代码级别的方式共享这些自定义控件外，我们还介绍了使用静态库的方式来共享组件。

最后，我们还介绍了 UIKit 框架中一种颇具特色的翻页控件 UIPageControl，以及 SDK 5.0 中新增的 UIPageViewController 视图控制器。

PDG

企业应用篇

从第 7 章开始到第 17 章，是关于企业应用开发的主题。在这一篇，你将看到网络、数据库、安全等和企业应用开发相关的常见技术。

第 7 章　网络

第 8 章　XML 和 Json

第 9 章　保存用户数据

第 10 章　安全

第 11 章　多媒体、绘图及动画

第 12 章　多点触摸及手势

第 13 章　本地化

第 14 章　iOS 多线程和并行编程

第 15 章　通知、本地通知和远程通知

第 16 章　开源框架 Core Plot

第 17 章　通讯簿、GPS 和重力感应

PDG

第 7 章　网　　络

本章从较高层次介绍了 Cocoa 框架中对于网络服务访问技术的支持，例如如何使用 NSURLRequest 和 NSURLConnection 进行 HTTP 请求。此外，网络编程不是一门孤立的技术，有些技术虽然不能直接用于访问网络，但仍然与此密切相关。因此在本章还有一些相关技术的介绍，如多线程和异步技术（主要是 NSOperation 和 NSOperationQueue）。

然后，本章花了近一半的篇幅介绍了 ASIHTTPRequest 框架的使用。本书着重向读者推荐它的理由，不仅仅因为它是一个封装完美、功能强大的网络编程库，而且它是一个功能上没有任何限制的开源项目，不会向你收取任何费用。

在对 ASIHTTPRequest 框架进行全面介绍之后，我们又介绍了如何实现自己的网络访问模块，即使用具有特色的 PostRequest 类和 NetworkModule 类。PostRequest 类使用了 ASIHTTPRequest 进行异步 HTTP 访问，而 NetworkModule 使用了“池”的概念对 PostRequest 对象进行管理，并实现了单例模式。它们在性能、安全、代码共享和可伸缩性上得到了进一步的提升。

本章虽然没有涉及更底层的网络编程技术，如 CFNetWork、Socket 的内容，但对于本书主题——企业应用开发来说，这已经足够了。

7.1　使用 NSURLConnection 获得网络数据

在 iPhone 中，获取网络数据相当容易。大部分时候我们使用 NSURL 和 NSURLRequest 就可以进行基于网络的 HTTP 编程。NSURL 可以引用网络或本地资源，它是 URL 的一个包装。使用 NSURL，你需要创建一个 NSURL 对象，然后从 NSURL 获得数据：

```
NSURL *url = [NSURL URLWithString:urlString];
NSData* data=[NSData dataWithContentsOfURL:url];
```

NSURLRequest 通常结合 UIWebView 一起使用，首先创建一个 NSURL，然后转换为 NSURLRequest，最终在 UIWebView 中读取网页内容：

```
NSURL *url = [NSURL fileURLWithPath:path];
NSURLRequest *request = [NSURLRequest requestWithURL:url];
[webView loadRequest:request];
```

在最简单的情况下，我们使用上面两个类从网络获取数据。当需要一种更特定的方式处理 HTTP 数据时，比如异步请求或者要 POST 数据时，我们就要用到 NSURLMutableRequest（NSURLRequest 类的可变版本）和 NSURLConnection 类，下面我们用一个小程序来演示 NSURLConnection 的使用（源代码位于光盘“source\第 7 章”的 URLConnectionTest 目录下）。

打开 Xcode，新建一个 Empty Application，命名为 URLConnectionTest，同时反选“Use

Automatic Reference Counting"，在本书中我们不推荐使用ARC。

在项目中添加一个"UIViewController subclass"，命名为TestViewController，同时勾选"With XIB…"。打开TestViewController.xib，往其中拖入一个Button和一个Web View控件，如图7-1所示。

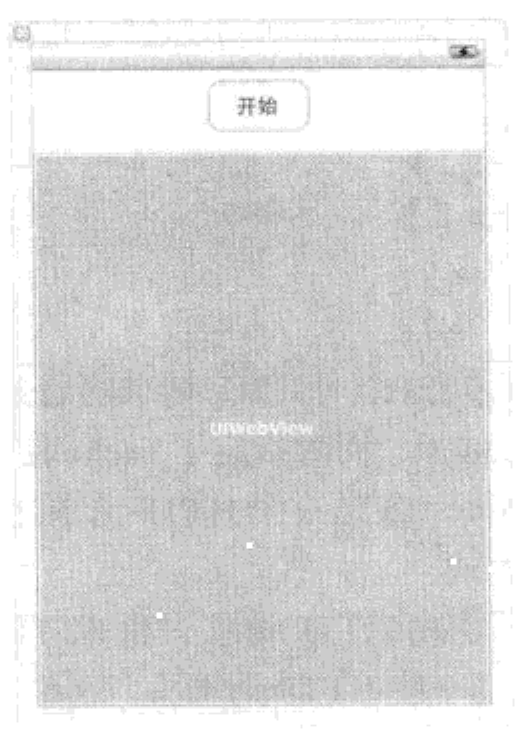


图7-1 在UIView中加入两个控件

编辑TestViewController.h文件，声明一个IBOutlet、一个成员变量和一个IBAction：

```
#import <UIKit/UIKit.h>
@interface TestViewController : UIViewController {
    NSMutableData* receivedData;
}
@property (nonatomic,retain)IBOutlet UIWebView *webView;
-(IBAction)goURL;
@end
```

头文件中，我们增加了一个UIWebView的出口webView，用于和IB中的Web View控件进行连接。goURL方法声明为IBAction出口。这样我们可以把"开始"按钮的事件连接到这个出口进行处理。由于我们使用了异步请求，会一点点地分批收到网络数据，我们的成员变量receivedData是一个可变的NSData，这样就可以把数据一点点地添加到receivedData中组装起来。

回到Interface Builder（TestViewController.xib的编辑界面），将UIWebView控件连接到IBOutlet变量webView，将UIButton控件的Touch Up Inside事件连接到-(IBAction)goURL。

在Xcode中编辑TestViewController.m，添加语句"@synthesize webView;"。实现方法-(IBAction)goURL：

```
-(IBAction)goURL{
    NSLog(@"gourl");
    NSURL* url=[NSURL URLWithString:@"http://www.apple.com"];//❶
```

```
    NSMutableURLRequest *request = [[NSMutableURLRequest alloc] initWithURL:url
         cachePolicy:NSURLRequestReloadIgnoringLocalCacheData
                            timeoutInterval:60];// ❷
    receivedData=[[NSMutableData alloc]init];
    [request setHTTPMethod:@"GET"];// ❸
    [request addValue:@"text/html" forHTTPHeaderField:@"Content-Type"];//❹
    NSURLConnection *conn = [[NSURLConnection alloc] initWithRequest:request
                            delegate:self];// ❺
    [request release];
    [conn release];
}
```

代码说明：

❶ 构建了一个 NSURL，然后用这个 NSURL 请求网络数据。

❷ 初始化 NSMutableURLRequest，同时指定了 cache 使用方式和超时时间。

❸ 在使用 NSMutableURLRequest 发送一个 HTTP 请求之前，我们需要指定发送请求的方式，这里我们使用 GET 方式。

❹ 根据 HTTP 协议，HTTP 请求由 HTTP 请求头和 HTTP 请求体构成。因此在发送 HTTP 请求正文前，往往还会发送一些 HTTP 请求头。这些 HTTP 请求头以“名-值”对的形式构成，服务器可以根据请求头的名检索出 HTTP 头的具体内容（值）。这里，我们在 HTTP 头中指定了 Content-Type 的值为 text/html，我们知道这是表明该请求所请求的资源类型为 HTML 文件。

❺ 然后使用一个 NSURLConnection 对象发送该请求，在初始化时指定一个 URL Request（前面所构造的 NSMutable URLRequest）和一个 delegate。delegate 是一个委托对象，它可以是任何类型，但必须实现 NSURLConnection 中定义的众多的委托方法（11 个），这些方法不是全部都要实现。由于我们把 NSURLConnection 对象的 delegate 设定为 self，因此接下来我们会在本类中实现其中的一些委托方法。

实现的这些委托方法（2 个）如下列代码所示：

```
//接收 NSData 数据
- (void)connection:(NSURLConnection *)aConn didReceiveData:(NSData *)data {// ❶
    [receivedData appendData:data];
    NSString *results = [[NSString alloc]
                    initWithBytes:[receivedData bytes]
                    length:[receivedData length]
                    encoding:NSUTF8StringEncoding];
    [webView loadHTMLString:results baseURL:url];
}
//接收完毕，显示结果
- (void)connectionDidFinishLoading:(NSURLConnection *)aConn {// ❷
    NSString *results = [[NSString alloc]
                    initWithBytes:[receivedData bytes]
```

```
                        length:[receivedData length]
                        encoding:NSUTF8StringEncoding];
    [webView loadHTMLString:results baseURL:nil];
}
```

代码说明：

❶ 这个方法在 NSURLConnection 收到 HTTP 数据时回调。我们这个方法中，把数据用 UTF-8 编码(网页中指定的编码)组装成 HTML 字符串，然后用 UIWebView 加载 HTML 字符串并显示。

❷ 这个方法在 NSURLConnection 接收完毕连接断开时回调。在这个方法中重复了第 1 个方法同样的动作。

程序最终运行效果如图 7-2 所示。

图 7-2　用 WebView 加载 HTML 页面

注意：在基于网络的企业应用中，尤其需要注意字符编码的问题。

对于大部分现代网站而言，使用 UTF-8 编码是可行的。上面的这个 URLConnectionTest 程序能在特定的 URL 下工作，比如用于访问苹果网站 http://www.apple.com。但对于某些网站（尤其是中文网站），比如 www.google.com.cn，即使页面本身采用的是 UTF-8 编码（我们可以用浏览器查看页面的编码），results 字符串也经常会获得 nil。实际上 Web 已经返回了所需的数据（打印 receivedData 成员可以看到，receivedData 并不为 nil）。出现这样情况的原因，是因为 Google 这样的网站会针对不同的客户端采用不同的编码。同样是访问 www.google.com.hk，如果是用 Safari、FireFox 或 IE 等浏览器，Google 会将页面采用 UTF-8 编码。而如果你采用一种 Google 无法识别的浏览器，那么 Google 实际上是采用 GB2312 编码。比如我们的 URLConnectionTest 程序，它实际上仅仅是使用了一个 NSURLConnection 而已，无法被 Google 识别为任何一种已有的浏览器。根据这样的情况，我们可以有两种解决的办法：

- 第 1 种方法，我们可以在 URLConnectionTest 中把 receivedData 按照 GB2312（或者其超集 18030）进行编码，那么可以得到正确的 HTML 字符串：

```
NSString *results = [[NSString alloc] initWithData:receivedData encoding:
  CFStringConvertEncodingToNSStringEncoding(kCFStringEncodingGB_18030_2000)];
```

- 第 2 种办法是欺骗服务器，让它把我们的 URLConnectioinTest 程序识别为某种有效的浏览器。你可能想到了，就是通过 UA（User-Agent）字串。UA 字串经常被放在 HTTP 头里用于告诉服务器客户端所使用的操作系统及浏览器版本。

比如：UA 字串“Mozilla/4.0（compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0）”。Mozilla/4.0 代表了 Netscape 公司的 Netscape Navigator 浏览器，由于该浏览器曾经流行一时，导致许多网页如果不使用该浏览器就根本无法打开网页。因此导致许多浏览器（包括 IE）在 UA 字串中保留了 Mozilla 名称以便兼容。Mozilla 的最终版本固定在了 4.0，因此 Mozilla/4.0 也由于历史的原因一直存在于 UA 字串中被各种浏览器用于伪装为 Netscape Navigator。MSIE 8.0 则表明浏览器真正的类型是 IE8。Windows NT 5.1 说明客户端操作系统为 Windows XP。Trident/4.0 是 IE8 的呈现引擎及版本。

如果我们要把客户端伪装成为 Safari，则 UA 字串可以用“AppleWebKit/533.18.1(KHTML，like Gecko）Version/5.0.2 Safari/533.18.5”。其中“AppleWebKit/533.18.1”表明了 WebKit 的版本号（Safari 的呈现引擎是 WebKit）。“(KHTML，like Gecko)”表明苹果实际上想让 Safari 同时兼容 KHTML 和 Gecko。KHTML 是另一个浏览器 Konqueror 的呈现引擎，Gecko 是火狐浏览器的呈现引擎（苹果是想让开发人员把 Safari 当成 Gecko）。最后是 Safari 的版本“Safrai/533.18.5”。533.18.5 是 Safari 的 Builder Version(构建版本)，它真正的版本是 Safari 5.02(即“Version/5.02”)。

讲了这么多，实际上只需要一句代码，即在 NSMutableURLRequest 中将 UA 字符串添加在 HTTP 头中：

```
[request setValue:@"AppleWebKit/533.18.1 (KHTML, like Gecko) Version/5.0.2 Safari/
      533.18.5" forHTTPHeaderField:@"User-Agent"];
```

这样，Google 会将 URLConnectionTest 程序识别为 Safari 浏览器，并将 HTML 以 UTF-8 编码的形式返回。将 URL 字符串修改为“http://www.google.com.hk”，再次运行程序，这次 Web View 能够正确地显示 Google 的首页。

7.2 使用 NSOperation 进行异步请求

NSOperation 是 Cocoa 的多线程处理类，一般和 NSOperationQueue 共同使用。在 iOS 网络应用中，NSURLConnection 通常需要和 NSOperation 一起使用，以实现杰出的用户体验——你永远不会希望，当用户从网络上下载数据时，因为糟糕的网络带宽导致界面被冻结整整 10 秒钟——在这种情况下，使用 NSOperation 建立多任务的 HTTP 异步请求是唯一的选择。所谓异

步请求，即请求线程是以非阻塞的方式运行的，程序在进行网络数据请求的同时，主线程不会被“冻结”，用户可以进行其他操作（比如点击按钮，滚动视图等）。与之相反，则为同步请求。使用多线程的好处并不仅仅是这些，多线程还可以让 CPU“同时” 执行多个任务（通过把 CPU 时间切分为细小的时间片），以充分利用 CPU 的计算能力。换句话说，我们可以用 NSOperation 同时进行多个 HTTP 请求。本节，我们将介绍使用 NSOperation 建立 HTTP 异步请求的技术。

新建 Objective-C 类，继承自 NSOperation：

```
@interface URLOperation : NSOperation
{
    NSURLRequest*  _request;
    NSURLConnection* _connection;
    NSMutableData* _data;
    //构建 encoding
    NSStringEncoding enc;
}
- (id)initWithURLString:(NSString *)url;
@property (readonly) NSData *data;
@end
```

在上面的头文件中，我们定义了一些有用的成员变量，如 URL Request、URL Connection、NSData、NSStringEncoding，你应该知道这些是用来干嘛的。在上一节中我们已经演示过这些对象的使用方法了。此外，我们还定义了一个特殊的初始化方法 initWithURLString:。实现代码如下：

```
@implementation URLOperation
@synthesize data=_data;
- (id)initWithURLString:(NSString *)url {
    if (self = [self init]) {
        _request = [[NSURLRequest alloc] initWithURL:[NSURL URLWithString:url]];
        //构建 utf-8 的 encoding
        enc = NSUTF8StringEncoding;
        _data = [[NSMutableData data] retain];
    }
    return self;
}
- (void)dealloc {
    [_request release],_request=nil;
    [_data release],_data=nil;
    [_connection release],_connection=nil;
    [super dealloc];
}
```

在初始化方法中，我们分别对头文件中声明的变量进行了初始化。

提示：enc 是 NSStringEncoding 类型，因为服务器返回的字符中使用了中文，所以我们通过它指定了一个 Utf-8 的字符编码。

整个类中最重要的方法是 start 方法，我们要单独介绍：

```
// 开始处理-本类的主方法
- (void)start {
    if (![self isCancelled]) {
        NSLog(@"start operation");
        _connection=[[NSURLConnection connectionWithRequest:_request delegate:
                    self]retain];// ❶
        while(_connection != nil) {// ❷
            [[NSRunLoop currentRunLoop] runMode:NSDefaultRunLoopMode beforeDate:
            [NSDate distantFuture]];
        }
    }
}
```

代码说明：

❶ 初始化 NSURLConnection 对象，以异步方式处理事件，并设置代理为 self。因此 self 应当实现 NSURL ConnectionDelegate 所定义的一系列方法。

❷ start 是 NSOperation 类的主方法，主方法的叫法充分说明了其重要性，因为这个方法执行完后，该 NSOperation 的执行线程就结束了（返回调用者），同时对象实例就会被释放，也就意味着你定义的其他代码（包括 delegate 方法）也不会被执行。所以在 start 方法代码中，还有一个 while 循环，这个 while 循环的退出条件是 HTTP 连接终止（即请求结束）。在循环中不停地运行当前循环，当循环结束，我们的工作也就完成了。

接下来，是 NSURLConnection 的 delegate 方法。这些方法中的一些已在上一节讨论 NSURLConnection 类的时候提到过了，不同的是在这里我们决定实现三个方法而不是两个。这些方法的代码是简单明了的，唯一需要注意的是方法参数 NSData* data。如你所见，你可以在其中添加自己想到的任何代码，包括接收数据，进行字符编码或者做 XML 解析：

```
#pragma mark NSURLConnection delegate Method
- (void)connection:(NSURLConnection*)connection
   didReceiveData:(NSData*)data {
    NSLog(@"connection:");
    NSLog(@"%@",[[NSString alloc] initWithData:data encoding:enc]);
    [_data appendData:data];
}
- (void)connectionDidFinishLoading:(NSURLConnection*)connection {
    [_connection release],_connection=nil;
}
-(void)connection: (NSURLConnection *) connection didFailWithError: (NSError *)
     error{
```

```
        NSLog(@"connection error");
    }
    @end
```

提示： 在第 1 个方法 connection:didReceiveData 中，data 参数是重要的。当你请求服务器资源时，NSURLConnection 并不是把服务器数据一次性地传递给你（委托对象），而是分许多次（如果数据不是少到只需要一次就结束的地步）。每一次，URL Connection 都会通过这个方法的 data 参数传递给 delegate。因此在这里我们会用 NSMutableData 的 append:方法把每次收到的 data 一点点添加进去。

到这里，虽然代码还没有完成，但我们已经可以运行它了。你可以看到 console 输出的内容，观察程序的运行状态。

我们的 NSOperation 类可以在 ViewController 中调用，也可以直接放在 AppDelegate 中进行。在这里，我是通过点击按钮来触发调用代码的：

```
-(void)buttonClicked{
    NSString* url=@" http://google.com" ;
    _queue = [[NSOperationQueue alloc] init];// ❶

    URLOperation* operation=[[URLOperation alloc ]initWithURLString:url];// ❷
    [_queue addOperation:operation];// ❸
}
```

代码说明：

❶ _queue 是一个 NSOperationQueue 对象，当往其中添加 NSOperation 对象后，NSOperation 线程会被自动执行（不是立即执行，根据调度情况）。

❷ 构造一个 NSOperation 对象。

❸ 将 NSOperation 对象加入到_queue。

我们需要一种机制，当 NSOperation 完成所有工作之后，通知调用线程处理数据。这里我们想到了 KVO 编程模型（这个主题在第 3 章中已讨论）。

首先，我们在 URLOperation 类中添加一个 BOOL 变量，当这个变量变为 YES 时，标志异步操作已经完成：

```
BOOL _isFinished;
```

在实现中加入这个变量的访问方法：

```
- (BOOL)isFinished
{
    return _isFinished;
}
```

Cocoa 的 KVO 模型中，有两种通知观察者的方式，自动通知和手动通知。顾名思义，自动通知由 Cocoa 在属性值变化时自动通知观察者，而手动通知需要在值变化时调用

willChangeValueForKey:和 didChangeValueForKey: 方法通知调用者。为求简便，我们一般使用自动通知。

要使用自动通知，需要在被观察对象的 automaticallyNotifiesObserversForKey 方法中明确告诉 Cocoa 哪些键路径要使用自动通知：

```
+ (BOOL) automaticallyNotifiesObserversForKey:(NSString*)key
{
    //当这个键路径改变时，使用自动通知
    if ([key isEqualToString:@"isFinished"])
    {
        return YES;
    }
    return [super automaticallyNotifiesObserversForKey:key];
}
```

注意：所谓"键路径"，是 Cocoa 中专有的术语（KeyPath），即可以通过"点语法"访问的对象的属性或方法。比如你通过 URLOperation 对象的键路径".isFinished"访问到的正是 URLOperation 的-(BOOL)isFinished 方法。而该方法实际上是成员"_isFinished"（注意前面的下划线）的访问方法。因此当一个观察者在注册 KVO 时，如果注册的是键路径 isFinished，其实也就是注册了对成员"_isFinished"的变更通知。从此可知，对象不应该直接注册对另一个对象的成员变量的变更通知，因为观察者不可能用"点语法"访问到它的实例成员，必须通过该实例成员的访问方法才可以。

然后，在需要改变_isFinished 变量的地方使用语句：

```
_isFinished=YES;
```

请在 - (void)connectionDidFinishLoading:(NSURLConnection*)connection 方法代码中使用上述语句。

最后，需要在观察者（AppDelegate）的代码中注册 KVO。在 AppDelegate 类的 buttonClicked 方法代码中，在 URLOperation 对象初始化之后加入：

```
//KVO 注册
[operation addObserver:self forKeyPath:@"isFinished"
               options:(NSKeyValueObservingOptionNew | NSKeyValueObserving Option
               Old) context:operation];
```

对于 KVO 模式，观察者（AppDelegate）的责任是实现方法：

```
observeValueForKeyPath:ofObject:change:context:
```

并在方法中接收所注册的键路径的变更通知：

```
- (void)observeValueForKeyPath:(NSString *)keyPath
    ofObject:(id)object change:(NSDictionary *)change
    context:(void *)context
```

```
{
    if ([keyPath isEqual:@"isFinished"]) {
      BOOL isFinished=[[change objectForKey:NSKeyValueChangeNewKey] intValue]; // ❶
      if (isFinished) {
           [indicatorView stopAnimating];
           URLOperation* ctx=(URLOperation*)context;// ❷
           NSStringEncoding enc=CFStringConvertEncodingToNSStringEncoding (kCFString
                   EncodingGB_18030_2000);// ❸
           NSLog(@"%@",[[NSString alloc] initWithData:[ctx data] encoding: enc]);// ❹
           [ctx removeObserver:self
                   forKeyPath:@"isFinished"];// ❺
      }
      }else{
           [super observeValueForKeyPath:keyPath
                ofObject:object change:change context:context];
      }
}
```

代码说明：

❶ 如果数据接收完成，isFinished 标志的值应当是 YES（int 值 1）。

❷ URLOperation 在最后一个参数 context 中被传递过来了。

❸ 指定编码为 GBK。

❹ 通过 URLOperation 的 data 属性获取 HTTP 请求所返回的数据，并以指定编码打印。

❺ 处理结束，移除 KVO 注册。

运行程序，点击“go”按钮，注意查看控制台的输出。整个项目的代码位于光盘“source\第 7 章\URLOperation”目录。

7.3　与网络相关的示例

在苹果开发者网站上，有许多苹果官方的示例代码，对于程序员来说这都是极其重要的学习资料。其中有关于 HTTP 网络编程的两个经典示例，分别是 SimpleURLConnection 和 AdvancedURLConnection。它们的下载地址分别为：

- http://developer.apple.com/iphone/library/samplecode/SimpleURLConnections/SimpleURLConnections.zip
- http://developer.apple.com/iphone/library/samplecode/AdvancedURLConnections/AdvancedURLConnections.zip

如果你不想去下载，则本书光盘中已经提供了这两个项目。

1. SimpleURLConnection 示例

SimpleURLConnection 中，包含了 3 个类，分别演示了 3 种不同的 HTTP 请求方式：GetController、PutController 和 PostController。

GetController类中，使用NSURLConnection进行异步Get请求，主要的代码是 _startReceive方法：

```
assert(self.filePath != nil);
self.fileStream = [NSOutputStream outputStreamToFileAtPath:self.filePath append:NO];
assert(self.fileStream != nil);
[self.fileStream open];
request = [NSURLRequest requestWithURL:url];
assert(request != nil);
self.connection = [NSURLConnection connectionWithRequest:request delegate:self];
```

首先，获得一个临时文件名（文件名以Get开头）：

```
self.filePath = [[AppDelegate sharedAppDelegate] pathForTemporaryFileWithPrefix:
    @"Get"];
```

提示：AppDelegate的pathForTemporaryFileWithPrefix:方法会在应用程序临时文件夹下生成一个临时文件名，文件名以指定前缀字符开头，以“-”号和随机产生的UUID字符串结尾。

然后构建NSURLRequest、NSURLConnection发送请求，并将NSURLConnection的委托设置为self。

在委托方法connection: didReceiveData:里，将异步接收到的NSData写入到临时文件夹的缓存文件中去。当数据接收完毕，在委托方法connectionDidFinishLoading:中调用_stopReceiveWithStatus:方法，关闭HTTP连接和文件流，并将UIImageView的image属性设置为缓存文件，从而显示出所下载的图片。

PutController类中，_startSend方法包含的主要代码如下所示：

```
self.fileStream = [NSInputStream inputStreamWithFileAtPath:filePath];
assert(self.fileStream != nil);
request = [NSMutableURLRequest requestWithURL:url];
assert(request != nil);
[request setHTTPMethod:@"PUT"];
[request setHTTPBodyStream:self.fileStream];
...
contentLength = (NSNumber *) [[[NSFileManager defaultManager] attributesOfItemAtPath:
    filePath error:NULL] objectForKey:NSFileSize];
assert( [contentLength isKindOfClass:[NSNumber class]] );
[request setValue:[contentLength description] forHTTPHeaderField:@"Content-Length"];
self.connection = [NSURLConnection connectionWithRequest:request delegate:self];
```

首先从资源束中获取文件流，然后将NSURLMutableRequest的提交方法改为PUT，并将文件流放到HTTPBodyStream属性进行提交，同时在HTTP头中加入Content-Length值。

同GetController一样，接下来的事情就是在PutController中实现NSURLConnection的委托方法。

PostController类的主要代码在 _startSend方法和NSStream委托方法stream:handleEvent:

里。对于 POST 方式，情况则比较复杂。因为根据 HTML 4.0 规范，POST 请求的 HTTP 头 Content-Type 为 multipart/form-data，表单数据被分成不同的 form-data，不同 form-data 之间以 Boundary 字符串分隔（该字符串以“Boundary-”开头，加上一个 UUID 字符串结束）。对于一个文件上传表单来说，包含了一个 FileUpload 表单控件和一个上传按钮控件，因此 POST 的数据由两个 form-data 组成。而文件数据就属于第 1 个 form-data 后半部分。构造完 multipart/form-data 之后，又构建了一对 NSStream 对象，用于读取缓存 buffer 中的数据。接下来的事情仍然是在 NSStream 的委托方法 stream:handleEvent:中进行处理。注意 buffer 中的数据首先是第 1 个 form-data 的头部，当头部读完后，buffer 的内容变为了文件内容，当缓存中的文件内容读完，继续将文件后面的内容载入 buffer，当整个文件内容都读完，才是第 2 个 form-data。由于代码太多，详细代码请参考项目源文件，在此不再详列。PostController 类中还掺杂了一些 C 代码，对有的读者来说理解起来比较困难。但不用担心，因为本书不会涉及过多 HTTP 或 Socket 底层的东西，对这些部分直接略过即可。

2. AdvancedURLConnection 示例

打开 AdvancedURLConnection 项目，直接编译运行。如果出现“no SDK with the name or path ‘iphoneos’”错误，则需修改项目的 Base SDK（在 info 窗口 Build 面板）。

在这个示例中，演示了 iPhone 如何访问一个 TLS/SSL HTTPS 服务器以及使用 Cocoa 的 security 框架访问 iPhone 钥匙串。在 Tab 页 Get 中，选择 URL：http://www.cacert.org/ images/ cacert4.png，我们可以获得并显示该图片。当我们点击 URL：https://www.cacert.org/images/ cacert4.png（同一图片，但使用 HTTPS 协议），却出现 Connection failed 错误。

切换到 Debug 页，将 TLS Server Validation 选项改为 Ask For Each Untrusted Site，重新访问 https://www.cacert.org/images/cacert4.png，会弹出服务器证书接受窗口，会问你是否接受服务器发来的证书，如果你想继续访问该地址，必须选 Accept（接受），如图 7-3 所示。

图 7-3　SSL 证书提示

点 Accept（接受）后，将在整个请求响应过程中使用服务器发来的证书进行 SSL 通信，同时图片也可以显示出来了。

AdvancedURLConnection 涉及 HTTP 编程中的高级话题，限于篇幅，我不准备在此详细解释它的源代码，读者可以自行参考光盘中的源文件，其中的实现细节读者也不必一一深究，因为这些东西已经有一些现成的、封装良好的框架可以调用。这里只是呈现一个简单概念，关于 SSL 的话题，会在第 9 章深入讨论。

7.4 ASIHTTPRequest 框架介绍

ASIHTTPRequest 项目地址：http://github.com/pokeb/asi-http-request/tree，关于 ASIHTTPRequest 到底是什么，你可以在项目首页看到。它提供如下功能：

- ❑ 提交数据到 Web 服务器或者从 Web 服务器获得数据；
- ❑ 下载数据到内存或磁盘；
- ❑ 采用 html input 相同的机制上传文件；
- ❑ 断点续传；
- ❑ 简单存取 HTTP 头；
- ❑ 上传/下载进度显示；
- ❑ 支持 Cookie；
- ❑ 后台运行（iOS4.0 以上支持）；
- ❑ 对于请求和响应的 GZIP 支持；
- ❑ 支持客户端证书；
- ❑ 支持同步/异步请求。

还有很多，关于它的介绍网上已经有很多了，该项目有很详细的指南文档：How to use ASIHTTPRequest（地址：http://allseeing-i.com/ASIHTTPRequest/How-to-use），也有网友翻译成中文了。本书并没有照搬官方文档的内容，而是着重介绍了几个常见的应用，并涵盖了一些自己的理解和实际应用经验，包括：简单异步/同步请求、队列请求、上传、下载及 Cookies。

注意，虽然这些技术在本文中是分开讲述的，但在实际工作中，往往是多种技术结合应用的。

此外，HTTP 请求往往伴随着 XML 技术的应用，实际上 ASIHTTPRequest 是可以和 SAX 异步解析结合应用的，后面会有详细介绍。本节首先介绍如何在项目中使用 ASIHTTPResquest。

首先需要下载 ASIHTTPRequest，下载地址：http://github.com/pokeb/asi-http-request/tarball/master。

下载后将文件解压缩到任意目录。打开该目录，其中包含了：

- ❑ 一个 iPhone Xcode 项目（源文件）
- ❑ 一个 Mac Xcode 项目（源文件）
- ❑ 一个 iPhone 下使用的 Sample Code（源文件）
- ❑ 一个 Mac 下使用的 Sample Code（源文件）

- 一个 Readme.texttile，关于该项目的介绍

其实所有的内容都在其中了，如果你是初学者，不知道怎么下手，可以看 http://allseeing-i.com/ASIHTTPRequest/How-to-use，这里有一份详细的入门指南。现在，我们要做的就是在自己的项目中使用它。

ASIHTTPRequest 是一个开源项目，要使用它，直接复制项目源文件到你的项目中，包括以下文件（即 Classes 下所有文件和 External/Reachability 下所有文件）：

- ASIHTTPRequestConfig.h
- ASIHTTPRequestDelegate.h
- ASIProgressDelegate.h
- ASICacheDelegate.h
- ASIHTTPRequest.h
- ASIHTTPRequest.m
- ASIDataCompressor.h
- ASIDataCompressor.m
- ASIDataDecompressor.h
- ASIDataDecompressor.m
- ASIFormDataRequest.h
- ASIInputStream.h
- ASIInputStream.m
- ASIFormDataRequest.m
- ASINetworkQueue.h
- ASINetworkQueue.m
- ASIDownloadCache.h
- ASIDownloadCache.m

对于 iPhone，还要拷贝以下文件：

- ASIAuthenticationDialog.h
- ASIAuthenticationDialog.m
- Reachability.h (External/Reachability 目录)
- Reachability.m （External/Reachability 目录）

此外，ASIHTTPRequest 依赖于以下 5 个框架或库，也需要在你的项目中添加它们：CFNetwork、SystemConfiguration、MobileCoreServices、CoreGraphics 和 libz1.2.5。

这样，在你的源代码中使用“#import "ASIHTTPRequest.h"”语句之后就可以使用 ASIHTTPRequest 了。

7.4.1 发送同步请求

同步请求以线程阻塞的方式接收服务器的响应消息。使用 ASIHTTPRequest 的

startSynchronous 方法可发送一个同步请求。以下代码示范了如何用 ASIHTTPRequest 发送同步请求。它主要分为 3 个步骤：构造一个 ASIHTTPRequest 对象，调用 startSynchronous 方法，通过 responseString 获得服务器返回的字符串。

```
NSURL *url = [NSURL URLWithString:@"http://localhost/interface/GetDept"];
// 构造 ASIHTTPRequest 对象
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
// 开始同步请求
 [request startSynchronous];
// 读取 Response
NSString *response = [request responseString];
NSLog(@"%@",response);
```

7.4.2 发送异步请求

发送异步请求有两个方法，一个是 delegate，一个是块。

1. 异步请求中的委托：delegate

异步请求以非阻塞的方式接收服务器响应消息。与同步请求不同，发送一个异步请求需要使用 startAsynchronous 方法。ASIHTTPRequest 负责拦截 HTTP 会话事件，并将事件委托给 delegate 对象处理。ASIHTTPRequest 的 delegate 可以由任意实现了一系列协议方法的 NSObject 担任：

```
NSURL *url = [NSURL URLWithString:@"http://localhost/interface/GetDept"];
ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
// 设定委托，委托自己实现异步请求方法
[request setDelegate:self];
// 开始异步请求
[request startAsynchronous];
```

可以看到，代码中使用了 startAsynchronous 方法来发送一个异步请求，同时还要用 setDelegate 方法设置 delegate 对象。delegate 对象（这里是 self）需要实现下列方法（可选）：

- (void)requestStarted; // 请求开始时调用
- (void)requestReceivedResponseHeaders:(NSDictionary *)newHeaders; // 收到 HTTP 头时调用
- (void)requestFinished; // 请求结束时调用
- (void)failWithError:(NSError *)theError; // 请求失败时调用
- (BOOL)retryUsingNewConnection; // 如果需要用新连接重发请求，返回 YES，否则
- (void)redirectToURL:(NSURL *)newURL; // 定义重定向 URL

上述方法 delegate 不需要全部实现，甚至一个都不用实现。以下代码演示两个最常见实现：

```
// 请求结束，获取 Response 数据
- (void)requestFinished:(ASIHTTPRequest *)request
{// ❶
    // NSData *responseData = [request responseData];
```

```
    NSString *responseString = [request responseString];// ❷
    NSLog(@"%@",responseString);// ❸
    button.enabled=YES;
}
// 请求失败，获取 error
- (void)requestFailed:(ASIHTTPRequest *)request
{// ❹
    NSError *error = [request error];// ❺
    NSLog(@"%@",error.userInfo);// ❻
    button.enabled=YES;
}
```

代码说明：

❶ 这个方法在请求结束时调用。

❷ 通过 request 对象的 responseString 就可以获取 Response 数据（以文本格式）。

❸ 打印 Response 数据。

❹ 这个方法在请求失败时调用。

❺ 通过 request 对象的 error 属性就可以获取错误对象。

❻ 打印错误信息。

2. ASIHTTPRequest 对块的支持

从 OS X 10.6 及 iOS 4.0 起，Objective C 也支持块(Blocks)语法。你可以用块语法调用 ASIHTTPRequest，在 setCompletionBlock:方法、setDataReceivedBlock:方法和 setFailedBlock;方法中，传递块参数：

```
-(IBAction)goURL{
    NSURL *url = [NSURL URLWithString:@"http://localhost/interface/GetDept"];
    __block ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
    [request setCompletionBlock:^{
        NSString *responseString = [request responseString];
    NSLog(@"%@",response String);
    }];//定义请求响应结束时调用的块。
    [request setFailedBlock:^{
        NSError *error = [request error];
        NSLog(@"error:%@",[error userInfo]);
    }];//定义请求响应失败时调用的块。
    [request startAsynchronous];// 发送异步请求。
}
```

如果你不熟悉块语法，请参考本书第 3 章关于块的介绍。

7.4.3 文件上传

文件上传需要服务端的配合。服务端代码用 Java 实现，我已经把完整的 Servlet 及 JSP 代

码放到光盘的“source/第 7 章/UploadTest”目录里——包括一个服务器上传组件 UploadServlet 和用于测试上传的 JSP 页面 main_upload.jsp。你可以直接使用它们。

注意：由于使用了 Apache 的 fileupload 组件，需要将 commons-io-1.4.jar、commons-fileupload-1.2.1.jar、commons-beanutils.jar 三个 jar 包复制到项目的 WEB-INF/lib 目录下，否则 Eclipse 提示编译错误。此外，在 web.xml 中，还需要增加一个上下文参数 file_upload 为服务器指定一个上传文件的存放路径，在<web-app>后加入：

```
<context-param>
<param-name>file-upload</param-name>
<param-value>/data/</param-value>
</context-param>
```

同时，在文件系统中要确保此目录的存在。即在 root 下新建文件夹 data，并在 data 目录下建立 temp 目录，作为 Apache fileupload 组件的临时文件目录。

将项目部署到 tomcat 中，启动 tomcat，访问 http://localhost:8080/test/upload.jsp，显示界面如图 7-4 所示。

选择一个文件进行上传，然后到 Web 目录的/data 目录下检查该文件是否上传成功。

下面来看 iPhone 客户端的实现。该示例项目已放到光盘“source/第 7 章/ASIUploadDemo”目录下。

新建 Empty Application 项目“ASIUploadDemo”。将 ASIHTTPRequest 的 Classes 目录下所有文件和 External/Reachability 下所有文件添加到项目中，并添加 ASIHTTPRequest 所必需的 5 个框架或库到项目中。

新建类，选择 UIViewController subclass，并勾上“With XIB for user interface”，命名为 UploadViewController。

用 IB 打开 Xib 文件，在其中拖入 1 个 UIToolBar、1 个 UIBarButtonItem 和 1 个 UIWebView（见图 7-5）。

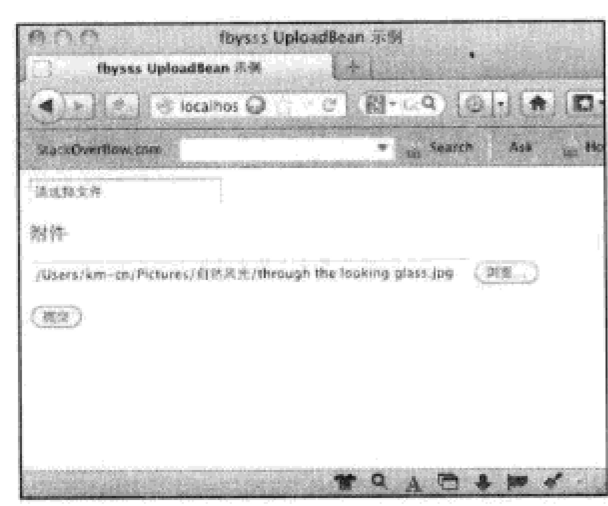


图 7-4　upoload.jsp

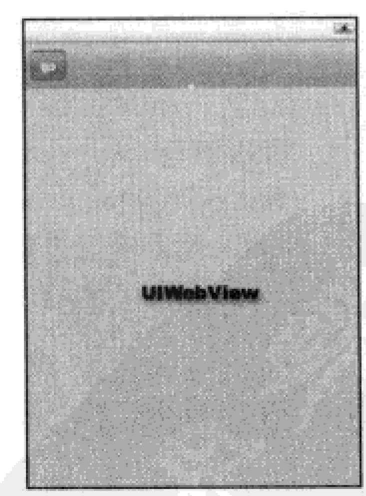


图 7-5　ASIUploadDemo 的 UI 设计

在 UploadViewController.h 中声明必要的变量和出口：

```
#import <UIKit/UIKit.h>
#import "ASIFormDataRequest.h"
#import "ASIHTTPRequest.h"
@interface UploadViewController : UIViewController {
    UIBarItem* button;// ❶UIBarItem，引用了工具栏上的 go 按钮。
    UIWebView* webView;// ❷一个 UIWebView，引用了 xib 中的 Web View。
    ASIFormDataRequest *request;// ❸声明一个 ASIFormDataRequest 对象，用于发送文件附件。
    NSURL *url;
}
@property(retain,nonatomic)IBOutlet UIBarItem* button;
@property(retain,nonatomic)IBOutlet UIWebView* webView;
-(IBAction)go; // ❹一个 Action 方法，由于响应 go 按钮的触摸事件。
-(void)printBytes:(NSString *)str encoding:(NSStringEncoding)enc;
@end
```

我们使用 ASIFormDataRequest 替换 ASIHTTPRequest 来发送 HTTP 请求，因为我们将以 Post 表单数据的形式而不是 Get 方式向服务器提交文件数据。

将所有出口正确地连接到 UploadViewController.xib 中，保存。

打开 UploadViewController.m，首先是工具栏 go 按钮的事件处理方法。在 go 方法中，我们演示了 ASIFormDataRequest 的使用：

```
@synthesize button,webView;
-(IBAction)go{// ❶
    NSString* s=@"哈哈哈";
    url=[NSURL URLWithString:@"http://localhost:8080/test/UploadServlet"];// ❷
    request = [ASIFormDataRequest requestWithURL:url];// ❸
    NSStringEncoding enc=CFStringConvertEncodingToNSStringEncoding(kCFString
                    EncodingMacChineseSimp);
    [request setStringEncoding:enc]; // ❹
    [request setPostValue:s forKey:@"title"];//❺
    [request setFile:@"/Users/km-cn/Documents/iphone/Iphone 开发介绍.doc" forKey:
          @"attach"];//❻
    [request setDelegate:self];// ❼
    [request setDidFinishSelector:@selector(responseComplete)];// ❽
    [request setDidFailSelector:@selector(responseFailed)];// ❾
    [button setEnabled:NO];
    [request startSynchronous];// ❿
```

代码说明：

❶ 当工具栏 go 按钮被点击时触发本方法。

❷ 构造 NSURL。

❸ 用 NSURL 构造 ASIFormDataRequest 对象。

❹ 设置请求时的编码为 GBK。

❺❻ 我们准备模拟 HTTP 表单提交。根据图 7-6 中显示的 HTML 表单，我们需要模拟一个显示描述性文本的 input text 和一个上传文件的 input file。

❼ 设置 delegate 为 self。

❽ 设置请求完成时调用的委托方法。

❾ 设置请求失败时调用的委托方法。

❿ 开始同步请求。

委托（也称为“代理”）方法的实现很简单，请求完成后，我们直接将服务器响应的 HTML 显示在一个 Web View 中，用户可以看到文件上传的结果：

```
-(void)responseComplete{//请求响应结束，将响应的 HTML 显示到 Web View。
    NSString *responseString = [request responseString];
    [webView loadHTMLString:responseString baseURL:url];
    [button setEnabled:YES];
}
-(void)respnoseFailed{//如果请求失败，显示错误信息。
    NSError *error = [request error];
    [webView loadHTMLString:[error description] baseURL:url];
    [button setEnabled:YES];
}
```

编译、运行。点击 go 按钮，程序运行效果如图 7-6 所示（请确认此时我们的 tomcat 服务器和 UploadTest 应用是启动的）：

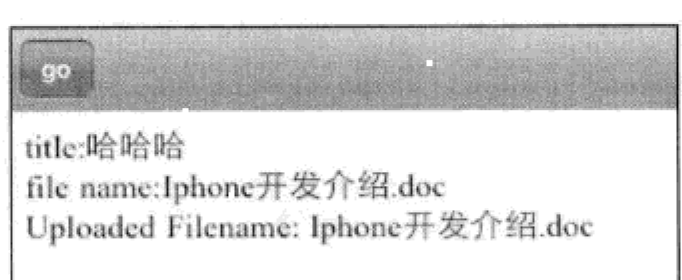


图 7-6 使用 ASIFormDataRequest 进行文件上传示例

注意，为了简便，代码从 Mac 文件系统中随便选取了一个文件进行上传。实际应用中，应该从 iPhone 文件系统中选择上传文件，因此你必须修改代码中的文件路径。

7.4.4 文件下载

ASIHTTPRequest 支持文件下载，同时在 UIProgressView 中显示下载进度。

在 Interface Builder 中打开.xib，拖入一个 Progress View，在源文件中声明一个 UIProgressView *progressView 变量，并声明为 IBOutlet。再在.xib 中拖入一个 Button 控件，在源文件中定义一个-(IBAction)goURL 方法出口，并将出口连接到 Button 的 touch up inside 事件。

首先用 NSURL 对象指定了一个服务器文件所在的 URL，然后用 requestWithURL:方法构

建 ASIHTTPRequest 对象。用 setDownloadDestinationPath:方法指定文件下载后保存的目标路径，用 setDownloadProgressDelegate 指定用于现实下载进度的窗体中的 Progress View 控件，然后调用 startSynchronous 方法开始同步下载：

```
-(IBAction)goURL{
    NSString* path=[NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
              NSUserDomainMask, YES) objectAtIndex:0];// ❶
    path=[path stringByAppendingPathComponent: @"plsqldev714.rar"];// ❷
    NSURL *url = [NSURL URLWithString:@"http:
               localhost/upload/plsqldev714.rar"];// ❸
    ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];// ❹
    [request setDownloadDestinationPath:path];// ❺
    [request setDownloadProgressDelegate:progressView];// ❻
    [request startSynchronous];// ❼
```

代码说明：

❶ 首先我们构造了文件能够保存的路径（应用程序 Documents 目录）。

❷ 在 Documents 目录后面加上文件名以构成完整的下载文件保存路径。

❸ 构造文件下载地址为 NSURL。

❹ 用 NSURL 初始化 ASIHTTPRequest。

❺ 设置 ASIHTTPRequest 的下载文件保存路径。

❻ 将下载进度显示到 progressView（一个 UIProgressView 控件）里。

❼ 开始请求（同步）。

运行程序，下载进度会在 progress view 中显示。下载进度显示当前完成的百分比。

ASIHTTPRequest 支持任务队列下载。队列是指 ASINetworkQueue 对象，其实是一种多线程操作，类似 NSOperationQueue，可以同时执行多个下载任务，甚至多线程下载同一文件。

提示： 多线程下载同一资源需要服务器支持，把同一个文件资源分成多个线程同时下载，最后再合并为一个文件。

下面的例子里我们使用了 ASINetworkQueue 同时进行多个下载任务，同时，在 Progress View 中显示精确进度。完整的 Xcode 项目位于光盘“source/第 7 章/ASIDownloadDemo”。

新建一个 Empty Application 项目。将 ASIHTTPRequest 的 Classes 目录下所有文件和 External/Reachability 下所有文件添加到项目中，并添加 ASIHTTPRequest 所必需的 5 个框架或库到项目中。

这个例子需要在 Interface Builder 中对界面进行一些设计。Add->New File，选择 UIViewController subclass，并勾上“With XIB for user interface”，命名为 QueueViewController。

用 Interface Builder 打开 QueueViewController.xib 文件，在其中拖入 6 个 UILable、1 个

UIButton 和 3 个 UIProgressView（见图 7-7）。

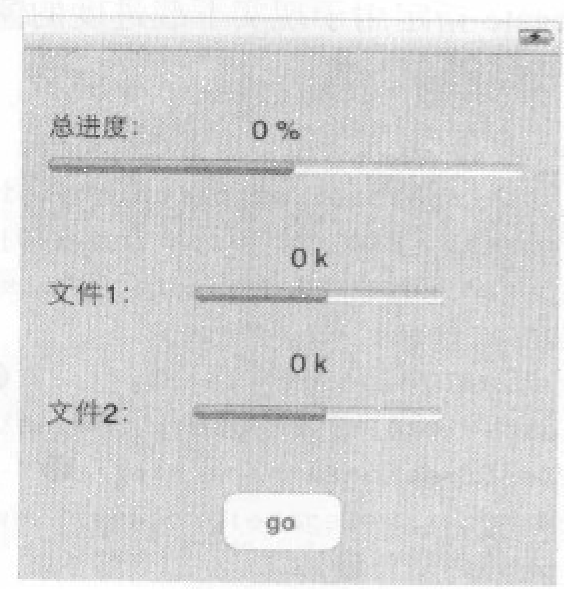


图 7-7 ASIDownloadDemo 的 UI 设计

在 Xcode 中声明必要的变量和出口：

```
#import <UIKit/UIKit.h>
#import "ASIHTTPRequest.h"
#import "ASINetworkQueue.h"
@interface QueueViewController : UIViewController {
    ASINetworkQueue *networkQueue;
    UILabel *status_total, *status_file1, *status_file2;
    UIButton *button;
    UIProgressView *progress_total, *progress_file1, *progress_file2;
    bool failed;
    NSFileManager* fm;
}
@property (nonatomic,retain)IBOutlet UILabel *status_file2, *status_file1,
            *status_ total;
@property (nonatomic,retain)IBOutlet UIButton *button;
@property (nonatomic,retain)IBOutlet UIProgressView *progress_file1, *progress_
            file2, *progress_total;
-(IBAction)go:(id)sender;
@end
```

将所有出口正确地连接到 QueueViewController.xib 中，保存。

编写 UIButton 的 Touch up inside 事件代码 go:方法。我们首先需要创建目标文件（下载文件将放在应用程序的 Documents 目录），使用 NSFileManager 的 createFileAtPath:contents:attributes:方法：

```
bool b=[fm createFileAtPath:path1 contents:nil attributes:nil]
```

如果创建成功，我们用 NSFileHandle 来打开它：

```
fh1=[NSFileHandle fileHandleForWritingAtPath:path1];
```

我们演示了 2 个文件的下载，对于第 2 个文件处理方式是一样的。

然后开始建立任务队列 ASINetworkQueue:

```
networkQueue = [[ASINetworkQueue alloc] init];
```

在使用 networkQueue 之前，我们还需要对它进行一些必要的设置。如队列清零、设置代理和设置 queue 的进度条。注意，queue 的进度条显示的是队列中所有请求的总体进度（两个文件），而每个请求的进度条则只是单个请求的（1 个文件）。

对于每个文件下载请求，我们使用单独的 url 和 ASIHTTPRequest。这里为了区别每个请求，我设置了请求对象的 userInfo 信息（每个请求的 userInfo 设置为它的文件名）：

```
[request setUserInfo:[NSDictionary dictionaryWithObject:file1 forKey:@"TargetPath"]];
```

然后就是设置请求的 complete 块、failed 块和 received 块。

首先看 received 块，这个块用于设置在请求成功后对所接收到的数据进行处理，这个块有一个 NSData 参数，保存了 ASIHTTPRequest 所接收的数据。我们把这些数据用前面构造的 File Handle 对象写到文件里，同时计算下载进度并显示在控件中：

```
[request setDataReceivedBlock:^(NSData* data){
    fSize1+=data.length;
    [status_file1 setText:[NSString stringWithFormat:@"%.1f K",fSize1/1000.0]];
    [status_total setText:[NSString stringWithFormat:@"%.0f %%",progress_total.
                progress*100]];
    if (fh1!=nil) {
        [fh1 seekToEndOfFile];
        [fh1 writeData:data];
    }
    NSLog(@"%@:%u",file1,data.length);
}];
```

接下来是 complete 块，在 ASIHTTPRequest 请求接收数据完成时进行一些释放资源的动作，比如释放 File Handle 对象，关闭所打开的文件：

```
[request setCompletionBlock:^(void){
        NSLog(@"%@ complete !",file1);
        assert(fh1);
        // 关闭 file1
        [fh1 closeFile];
}];
```

然后是 failed 块，如果发生任何错误，需要在这里处理。我们只进行了简单控制台打印，未做任何处理：

```
// 使用 failed 块，在下载失败时做一些事情
```

```
[request setFailedBlock:^(void){
    NSLog(@"%@ download failed !",file1);}
];
```

最后把 request 对象加入到任务队列：

```
[networkQueue addOperation:request];
```

对于第 2 个文件请求，所进行的动作跟第 1 个是完全一样的。当我们把所有请求都加到队列之后，使用代码：

```
[networkQueue go];
```

这些请求才会被执行。

完整的代码，请参考项目源文件 QueueViewController.m。

在代码中，我们从两个不同 URL 同时下载文件。注意，我不能保证这两个 URL 所指定的资源始终可用，如果资源不可用，需要读者重新指定可用的 URL。

7.4.5 Cookies 和 Sessions

Session 是重要的服务器状态保持策略，服务器为了支持 Session 机制，使用了多种实现机制，其中最常见的就是 Cookies。Web 服务器常使用 Cookies 技术来实现用户免登录功能和存储用户状态信息。ASIHTTPRequest 支持客户端 Cookies 的存取。

Session 是服务器端技术，虽然 Cookies 是保存在客户端的。因此我们需要一个服务器端环境。打开 Eclipse，新建 Web 项目 test，需要准备如下 JSP 页面：

- login.jsp——在 login.jsp，仅仅有一个用户登录表单，在 Session 机制中，用户登录是必需的。因为在 Web 中，往往只有经过合法登录的用户才需要保持状态。
- index.jsp——在 index.jsp 页面头部加入了一段 java 代码，要求用户必须登录，否则会自动转向登录页面。另外还把本页 URI 作为请求参数 lastUrl，这样在登录成功后，会自动跳转到用户登录前所请求的页面。
- second_page.jsp——second_page.jsp 页面跟 index.jsp 的差不多，只不过想演示一下 sessionid 在多个页面中的传递。

另外，我们还需要一个 Servlet，用于用户登录。这里我偷了个懒，只要用户名不为空，我们就通过验证，仅仅为了演示。需要注意以下两点：

- response.encodeURL()方法。这个方法服务器会检测客户端是否支持接收 Cookies，如果客户端（比如 IE）禁用了 Cookies，则服务器会通过重写 URL 的方式向客户端发送 sessionid。比如，客户端请求 index.jsp 页面，encodeURL()之后服务器会把 URL 地址改写为：

```
index.jsp;jsessionid=FC076B0E1F0763CFE7BFAFEBA2E0287C
```

- 页面的跳转。页面跳转有多种方式。比如 request.sendRedirect。在 servlet 中，千万不

要使用 sendRedirect ，这样会使用服务器跳转，servlet 会把原来的 request 参数和 attributes 全部抛弃（包括 Cookies）。所以 sendRedirect 之前和之后的请求使用的不是同一个 sessionid。因此在 servlet 中，我们采用的是 RequestDispatcher.forward()方法，它是客户端跳转，即服务器会让客户端（浏览器）重新请求另一个地址来转发，保证 session 状态不被丢失。

进行测试，打开浏览器，访问 http://localhost:8080/test/second_page.jsp，页面会自动跳到 login.jsp，要求你登录。因为 java 脚本检测不到你的 session 信息（未进行登录），所以认为你还未登录过。当你登录后（用户名密码未验证，随便输入），浏览器返回你原先请求的页面 second_page.jsp，在 seconde_page.jsp 页面，打印了登录用户的用户名，以及 session id。

完整的 Eclipse 项目放在光盘“source/第 7 章/UploadTest”目录下。

接下来是 iPhone 客户端的实现。我们需要演示 ASIHTTPRequest 对 Cookie 或 Session 的支持，项目代码放在光盘的“source/第 7 章/ASICookieDemo”目录下。

新建 Empty Application 项目。加入对 ASIHTTPRequest 的支持（参考前面的步骤）。新建 ViewController 子类 CookieViewController。CookieViewController 的界面很简单（如图 7-8 所示），1 个 UIWebView，1 个 UIToolBar，3 个 UIBarButtonItem。

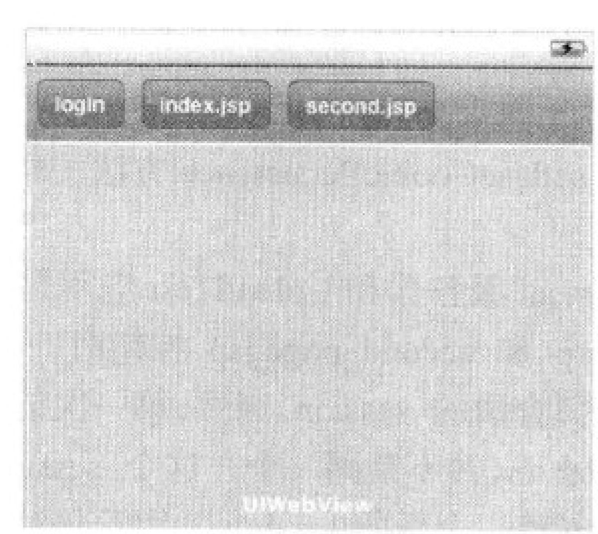


图 7-8 CookieViewController 的 UI 设计

在 CookieViewController 的 interface 声明必要的变量和出口。

将所有 IBOutlet 和 IBAction 对象在 Interface Builder 中建好连接。

打开 CookieViewController.m，我们在其中实现 login、gotoIndexPage、gotoSecondPage 三个方法。分别访问 3 个 JSP 页面。

在 login 方法中，我们将 URL 指定为 login.jsp：

```
url = [[[NSURL alloc] initWithString:@"http://localhost:8080/test/LoginServlet?
      username=dd&pass=1"] autorelease];
```

用 initWithURL:方法构建 ASIHTTPRequest，并指定 request 使用 Cookies 作为客户端 Sessions 保持策略：

PDG

```
request = [[[ASIHTTPRequest alloc] initWithURL:url] autorelease];
[request setUseCookiePersistence:YES];
```

并在返回的 HTML 字符串中查找“login failed!”字样，以此作为是否登录成功的判断：

```
NSString* html=[request responseString];
NSRange range=[html rangeOfString:@"login failed! " options:NSCaseInsensitive-
            Search];
```

然后根据用户登录成功与否在 Web View 中显示相应内容：

```
if (range.location==NSNotFound) {// 如果登录成功
        [webView loadHTMLString:@"login success" baseURL:url];
}else {// 如果登录失败
        [webView loadHTMLString:@"login failed" baseURL:url];
}
```

接下来是 gotoIndexPage 方法和 gotoSecondPage 方法。代码同 login 方法类似，同样指定 ASIHTTPRequest 的 Cookies 使用策略为 YES：

```
[request setUseCookiePersistence:YES];
```

然后用一个 Web View 分别显示 index.jsp 和 gotoSecondPage.jsp 的内容。

```
[webView loadHTMLString:[request responseString] baseURL:url];
```

可以看到，整个程序除了 setUseCookiePersistence:方法的调用以外，与一般的 ASIHTTP-Request 的使用没有任何不同。

编译运行，请确保此时 tomcat 服务器和 UploadTest 应用是运行的。

可以看到，当点击 index.jsp 和 second_page.jsp 按钮时，webView 中显示的始终是登录界面 login.jsp。因为我们没有获得合法的 session。当然，第一次请求始终可以获得一个 session，然而这个 session 和登录后的 session 并不是同一个，这个 session 中不会包含任何关于用户的信息。而登录成功后服务器会创建一个新的 session 并用于保存用户信息（比如用户名）。

点击 login 按钮，客户端以用户名 dd、密码 1 进行登录。由于服务端只检测用户名、密码是否不为空，只要不为空就可以登录成功。登录后再点击那两个按钮，则可以显示相应的页面。因为服务器已经在新的 session 中放入了用户登录时的用户名。其余两个页面都会检测 session 中是否保存有用户名，如果有则允许用户继续访问，否则会跳转到 login.jsp 要求用户登录以获得一个合法的 session。

ASIHTTPRequest 使用 Cookie 来支持与服务器的会话（Session）。当 setUseCookiePersistence: YES 时，表示在会话中使用 Cookie。这样，服务器在用户登录成功后，会发送一个 Cookie 给客户端。如果客户端的 Cookie 使用策略为 YES（useCookiePersistence= YES），则客户端在本地保存这个 Cookie（内存或者文件）。并且在会话过程中始终使用这个 Cookie 进行请求（其中就包括了 sessionid），直到会话过期或者浏览器关闭。

7.5 编写自己的网络模块类

经过对前面的学习，相信你对 ASIHTTPRequest 框架已经有一个全面的了解了。在企业应用开发中，网络的使用是必不可少的，通过使用 ASIHTTPRequest 访问网络，确实能极大地简化编码工作，但仍需我们编写少量的模式化代码。下面我们将对 ASIHTTPRequest 进行进一步的封装，构建我们自己的网络模块类，以便在今后项目中使用。

我将整个网络模块设计为两个类：NetworkModule 和 PostRequest。其中 PostRequest 是整个网络模块的基础，因为我们把每个网络请求封装成了一个 PostRequest，一个 PostRequest 对象就代表了一个 HTTP POST 请求。在这里，我们假定项目只使用 POST 的方式请求网络。PostRequest 类中包含了一个内置的 ASIHTTPRequest 对象，它实际上封装了 ASIHTTPRequest 的 HTTP 实现。

NetworkModule 是一个容器，用于容纳 PostRequest 对象。由于我们的网络模块支持异步操作,所以同一时间内可能有许多PostRequest在请求网络。NetworkModule 就类似于一个“池”，把这些 PostRequest 集中起来管理。同时，我们的 NetworkModule 实现了“单例”模式，即整个应用程序共用一个 NetworkModule 对象，而不会出现两个“池”（NetworkModule 实例）。

下面，我们就来介绍这两个类的实现。

7.5.1 PostRequest 类

首先有一个 ASIHTTPRequest 成员变量:

```
ASIHTTPRequest* _request;
```

我们将使用这个 ASIHTTPRequest 对象进行 HTTP 请求操作。

我们还声明了新枚举类型 kPostStatus，用于标志 PostRequest 的几个状态。在异步请求中这是很重要的，因为状态是我们进行各种操作的重要标志。

```
enum kPostStatus{
    kPostStatusNone=0,
    kPostStatusBeging=1,
    kPostStatusEnded=2,
    kPostStatusError=3,
    kPostStatusReceiving=4
};
typedef enum kPostStatus kPostStatus;
```

我们还需要一些属性声明，这些属性是进行 HTTP 请求时需要的重要参数，比如：

```
@property (nonatomic,retain)id<NetworkModuleDelegate> owner;
@property (nonatomic,retain)NSString* url;
@property (assign)kPostStatus postStatus;
@property (assign)kBusinessTag businessTag;
```

```
@property (nonatomic,readonly,getter = result)NSString* result;
```

其中，owner 是一个委托对象，这个委托实现了 NetworkModuleDelegate 协议（在后一个类介绍）。由于我们在 PostRequest 中采用的是异步请求，当请求发出后，PostRequest 不一定就一次性从服务器得到返回数据，必须在服务器数据返回时，通知该委托对象（一般会是一个 View Controller）取回结果或刷新界面。

url 是用于 HTTP 请求的 URL 地址。postStatus 用于跟踪异步操作的状态。

businessTag 属性很重要。在这里，我们使用“业务”的概念。每个 HTTP 请求都将其描述为一个业务，而 businessTag 就是这个业务的编号。因为在 NetworkModule 中，每次 HTTP 请求实质上总是要求服务器去完成某个业务，例如一次登录验证、一次数据库查询等。对于一个 PostRequest 对象来说，它的 businessTag 在 NetworkModule 的“池”中总是唯一的。我们把 businessTag 设计为 kBusinessTag 枚举类型，这样可以把每个业务编为不同的号。例如，用户登录编为 0，查询部门人员列表编为 1，等等。以下是 kBusinessTag 的定义：

```
enum kBusinessTag
{
    kBusinessTagUserLogin=0,
    kBusinessTagGetDeptPeople= 1
};
typedef enum kBusinessTag kBusinessTag;
```

这个枚举定义了两个枚举值。当然，随着业务功能的不断实现，我们很可能会再次扩充枚举的成员列表。

用 businessTag 作为 PostRequest 对象的 key，可以很方便地从“池”中检索出 PostRequest 对象。如果你要进行一次登录，从“池”中取出 businessTag 为 0 的 PostRequest 对象，就可以用这个 PostRequest 去向服务器进行登录。这样做的好处是可以共用“池”中的对象，减少内存占用。

最后一个有用的属性是 result 字符串，因为服务器返回的数据编码为文本后就放在这个属性里。

PostRequest 类有两个成员方法。其中最主要的方法是：

```
-(void)postXML:(NSString*)xml delegate:(id)delegate;
```

通过这个方法，你可以向服务器 POST 一个 XML 文件，然后由委托对象（由 delegate 参数指定）来接收和处理服务器返回的数据（也是 XML 文件）。这个方法是基于 ASIHTTPRequest 框架来实现的，实现代码如下：

```
-(void)postXML:(NSString*)xml delegate:(id)delegate{
    [self cancel];// ❶
    _request=[[ASIHTTPRequest requestWithURL:[NSURL URLWithString:url]]retain];// ❷
    [_request setShouldAttemptPersistentConnection:NO];// ❸
    [_request setResponseEncoding:self.enc];// ❹
```

```
    NSMutableDictionary *reqHeaders = [[NSMutableDictionary alloc] init];// ❺
    [reqHeaders setValue:@"text/xml; charset=UTF-8" forKey:@"Content-Type"];// ❻
    _request.requestHeaders = reqHeaders;// ❼
    [reqHeaders release];
    _request.tag=self.businessTag;// ❽
    [_request appendPostData:[xml dataUsingEncoding:self.enc]];// ❾
    [_request setDelegate:delegate];// ❿
    postStatus=kPostStatusBeging;// ⓫
    [_request startAsynchronous];// ⓬
}
```

代码说明：

❶ 在 POST 新的 XML 文件前，首先取消上次 HTTP 请求。

❷ 重新初始化 request 成员。

❸ 不要重用 HTTP 连接。

❹ 指明服务器响应时所使用的字符编码。

❺❻❼ 设置 HTTP 头。

❽ 把 businessTag 传入到 request 对象的 tag 属性中保存。这会在 NetworkModuleDelegate 协议的委托方法中用到。

❾ 设置要发送的 POST 正文。

❿ 设置 request 的委托对象，注意，这个委托对象是指实现了 ASIHTTPRequestDelegate 协议（而不是 NetworkModuleDelegate 协议）的委托对象。实际上，这就是 NetworkModule 对象。

⓫ 修改 PostRequest 的 postStatus 状态。

⓬ 开始异步请求。

result 属性的 getter 访问方法是特别设计的。我们检查了 PostRequest 的 postStatus 状态，当 postStatus 状态为服务器已响应完成时，取出 request 的 responseData 数据根据 enc 指定的编码进行字符编码，并将编码后的文本返回。

注意：后续的章节我们还会修改 result 访问方法，我们可能会在这里对服务器返回的数据进行 XML 解析。

接下来是 NetworkModule 类。

7.5.2　NetworkModule 类

NetworkModule 类中使用了自定义协议 NetworkModuleDelegate，该协议定义了三个方法：

```
@protocol NetworkModuleDelegate<NSObject>
@optional
-(void)beginPost:(kBusinessTag)tag;
-(void)endPost:(GDataXMLDocument*)result business:(NSNumber*)tag;
```

```
-(void)errorPost:(NSError*)err business:(NSNumber*)tag;
@end
```

NetworkModule 类将委托一个委托对象来实现这 4 个方法，并在不同的时机调用这些方法。当 NetworkModule 对象开始 POST 一个请求时，会调用委托对象的 beginPost 方法。当请求结束时，会调用委托对象的 endPost 方法。当请求失败，返回错误时，会调用委托对象的 errorPost 方法。

在 NetworkModule 类中，有一个 Mutable Dictionary 集合对象 queue，它充当了可以容纳多个 PostRequest 对象的“池”。

```
NSMutableDictionary* queue;
```

提示：这个“池”中，一个 PostRequest 对象代表了一种业务请求，它们不会重复。

NetworkModule 类只有两个成员方法：

```
-(void)postBusinessReq:(NSString*)xml
                   tag:(kBusinessTag)tag
                 owner:(id<NetworkModuleDelegate>)owner;
-(void)cancel:(kBusinessTag)tag;
```

第一个方法用于发送一个 POST 请求，第二个方法则取消一个 POST 请求。在介绍这两个方法之前，我们先来讨论如何在 NetworkModule 类中实现“单例”的问题。

有 Java 使用经验的程序员，应该熟悉“单例”的概念。在面向对象的概念中，“单例”即“单实例”，即一个类只有一个实例。我们需要类成为“单例”的，是因为很多时候这个类在应用程序中会被多个类所共用，比如公共的数据或模块。对于 NetworkModule 类而言，它的作用就是提供一个公共的“池”。这个“池”在应用程序中只需要一个就够了，我们可以把所有的 PostRequest 对象放在这个“池”里，没有必要分成几个“池”。因此 NetworkModule 被设计为“单例”的。

此外，有时候“单例”对象可以充任一种“全局对象”的角色。单例对象是类的唯一全局对象，任何对象都可以直接通过类使用这个“全局对象”而不必初始化它。

在 Cocoa 中，提供了一种实现单例的方式，你可以看苹果的文档“Creating a Singleton Instance”。在这里我不会详细介绍苹果提供的单例实现方式，我准备使用 Matt Gallaghter 提供的宏来实现单例。Matt Gallagher 曾经编写过一个宏，叫做 SYNTHESIZE_SINGLETON_FOR_CLASS，使用这个宏，你可以轻易地把任何类变成单例的。要使用这个宏，你必须使用 Matt Gallagher 编写的头文件 SynthesizeSingleton.h。你可以在本章的示例项目中找到它（后面我们提供了一个实现了网络模块的测试项目）。

使用 SynthesizeSingleton.h 实现的单例非常简单。首先，在类的头文件中导入 Synthesize-Singleton.h 头文件，并声明一个类方法 sharedMyClassName（MyClassName 为要实现单例的类的类名），比如：

```
+(NetworkModule*)sharedNetworkModule;
```

这个“sharedMyclass Name”方法不需要实现，在头文件中声明一下就可以了。

然后在类实现的@implementation 语句后使用宏：

```
SYNTHESIZE_SINGLETON_FOR_CLASS(MyClassName);
```

提示：MyClassName 为单例类的类名，比如 NetworkModule。

经过这样的处理，你可以把类 NetworkModule 变成单例。

一旦 NetworkModule 类成为单例，你可以使用下面的语句访问单例的 NetworkModule 对象：

```
[NetworkModule sharedNetworkModule];
```

注意：单例对象不需要显式地 alloc 和 init（在第一次访问时会自动调用 alloc 和 init），但如果你想实行初始化动作时仍然要实现默认的 init 方法。

接下来，我们看看 NetworkModule 方法的实现。在 NetworkModule 中，有两个成员方法，分别用于发送和取消一个 POST 请求。首先是发送方法：

```
-(void)postBusinessReq:(NSString*)xml
             tag:(kBusinessTag)tag
           owner:(id<NetworkModuleDelegate>)owner{// ❶
    PostRequest* req=(PostRequest*)[queue objectForKey:[NSNumber numberWithInt:
        tag]];// ❷
    if (req==nil) {
        req=[[PostRequest alloc]init];// ❸
    }
    req.businessTag=tag;// ❹
    req.postStatus=kPostStatusNone;// ❺
    [queue setObject:req forKey:[NSNumber numberWithInt:tag]];// ❻
    req.enc=NSUTF8StringEncoding;// ❼
    req.owner=owner;// ❽
    req.url=[NSString stringWithFormat:@" http://localhost:8080/RESTServer/
        Interface" ];// ❾
    [req postXML:xml delegate:self];// ❿
    if(owner!=nil) [owner beginPost:tag];// ⓫
}
```

代码说明：

❶ postBusinessReq 方法有 3 个参数。xml 参数用于指定要发送的 XML 内容。tag 参数指定要请求的业务编号，如 kBusinessTag 枚举中所定义的。owner 参数指定服务器返回的数据由哪个委托对象（一般是一个 View Controller）处理，owner 对象必须实现 NetworkModuleDelegate 协议。

❷❸ 根据业务编号（参数 tag 所指定），我们首先在 NetworkModule 的“池”里查找具有这个编号的 PostRequest 对象。如果找到，我们重用它；如果没有，则我们为这个业务编号创建一个新的 PostRequest 对象。

❹ 设置这个 PostRequest 的业务编号。

❺ 设置这个 PostRequest 的 postStatus 状态。因为还没开始请求，所以指定为 None 状态。

❻ 将 PostRequest 加入“池”中保存。

❼ 设置 PostRequest 的字符编码 enc 属性。

❽ 设置 PostRequest 的委托对象。这个委托对象应该是实现了 NetworkModuleDelegate 协议的对象（一般是一个 View Controller）。

❾ 设置 PostRequest 的 url 属性，即我们要请求的 URL 地址。这里我们指定为本机 8080（Tomcat 服务器）端口上的一个 URL 地址，我们的服务端代码将运行在这里（后面我们会提供服务端的实现代码）。

❿ 调用 PostRequest 对象的 postXML 方法，开始向服务器发送 XML 请求。

⓫ 如果委托对象存在，调用委托对象的 beginPost 方法。通知委托对象，我们已经开始请求了。

然后是取消请求方法，在这个方法中我们调用 tag 指定的 PostRequest 对象的 cancel 方法：

```
-(void)cancel:(kBusinessTag)tag{// cancel 方法有一个参数，用于标志要取消的 PostRequest。
    PostRequest* req=(PostRequest*)[queue objectForKey:[NSNumber numberWithInt:
                tag]];// 根据 tag 参数指定的业务编号，去“池”中查找指定的 PostRequest。
    if (req && [req isKindOfClass:[PostRequest class]]) {
        [req cancel];// 取消该 PostRequest 的请求。
    }
}
```

由于我们将 ASIHTTPRequest 的 delegate 委托对象设置为 self，所以 NetworkModule 还应当实现 ASIHTTPRequestDelegate 协议，如下代码所示：

```
- (void)requestFinished:(ASIHTTPRequest *)request
{//❶
    PostRequest* req=(PostRequest*)[queue objectForKey:
                                [NSNumber numberWithInt:request.tag]];
    req.postStatus=kPostStatusEnded;
    if (req.owner!=nil) {
        [req.owner endPost:req.result
                            business: req.businessTag]];
    }
}
- (void)requestFailed:(ASIHTTPRequest *)request
{//❷
    PostRequest* req=(PostRequest*)[queue objectForKey:
                                [NSNumber numberWithInt:request.tag]];
     NSError *error = [request error];
     if (req.owner!=nil) {
        [req.owner errorPost:error
                      business: req.businessTag]];
```

```
    }
}
```

代码说明：

❶ 这个方法在请求结束时调用。在这个方法中，我们根据业务编号从“池”中找出对应的 PostRequest 对象，然后调用委托对象 owner 的 endPost 协议方法。

❷ 这个方法在请求错误时调用。在这个方法中，我们根据业务编号从“池”中找出对应的 PostRequest 对象，然后调用委托对象 owner 的 errorPost 协议方法。

7.5.3 测试 NetworkModule

本章最后这个实例构建了一种纯 XML 的、无状态的、HTTP 网络应用模型。客户端向服务器发送一个 XML 文档，服务端代码根据该 XML 文档的内容，解析出客户端要进行的业务请求和对应的请求参数，进行处理运算，将结果以 XML 文档返回客户端。客户端解析 XML，然后呈现数据。不管有多少种业务，客户端和服务端的这种交互模式是不变的，甚至我们将所有业务都请求在一个 URL 地址上（参见 http://localhost:8080/RESTServer/Interface）。当然，不同的业务，在请求时应该有不同的 XML 文件请求格式。无论如何，我们最终都需要服务端代码的配合。

因此在测试 NetworkModule 之前，需要部署服务端代码。我用 Eclipse 编写了一个 Web 项目，你可以在光盘目录“source/第 7 章/RESTServer”中找到它。你可以直接用 Eclipse 打开这个 Web 项目，然后进行“Run AS→Run On Server”操作就可以在 Tomcat 中启动它了。

提示：在你运行测试项目前，请确保已启动了这个 Web 项目。

接下来我们打开测试项目。这个测试项目我们放在了光盘“source/第 7 章/NetworkModule”目录下。

用 Xcode 打开 NetworkModule 项目，可以看到我们的 PostRequest 和 NetworkModule 类已经在项目中了，如图 7-9 所示。ASIHTTPRequest 目录中是我们导入的 ASIHTTPRequest 框架。

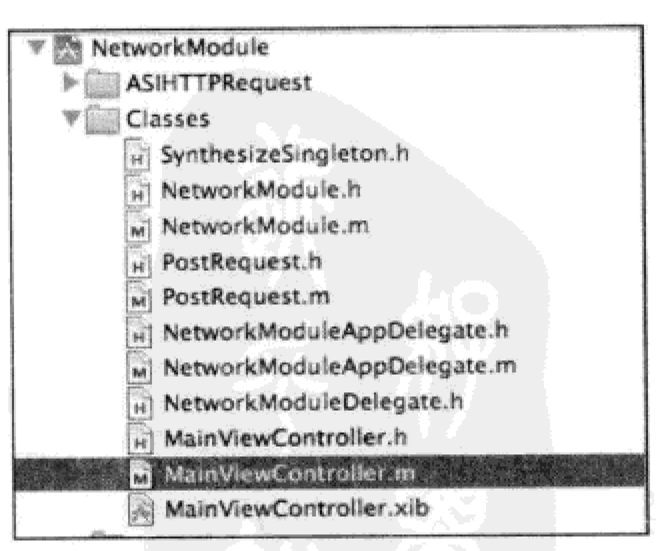


图 7-9 NetworkModule 项目的目录结构

另外，MainViewController 是我们的测试界面，我们在 MainViewController 中编写了代码，测试 NetworkModule 类是否能正常工作。MainViewController 实现了 NetworkModuleDelegate 协议。当点击界面上的按钮，会调用方法 buttonAction 进行 HTTP POST 请求：

```
-(IBAction)buttonAction{
    NSString* xml=@"<req><action>TestAction</action></req>";
    [[NetworkModule sharedNetworkModule]postBusinessReq:xml
        tag:kBusinessTagTest
        owner:self];
}
```

然后 MainViewController 实现了 NetworkModule Delegate 协议，并在 endPost 方法中把服务器返回的 XML 文件显示在 Text View 中：

```
-(void)endPost:(NSString *)result business:(kBusinessTag)tag{
    if (tag==kBusinessTagTest) {
        tv.text=result;
    }
}
```

在模拟器中运行效果如图 7-10 所示。

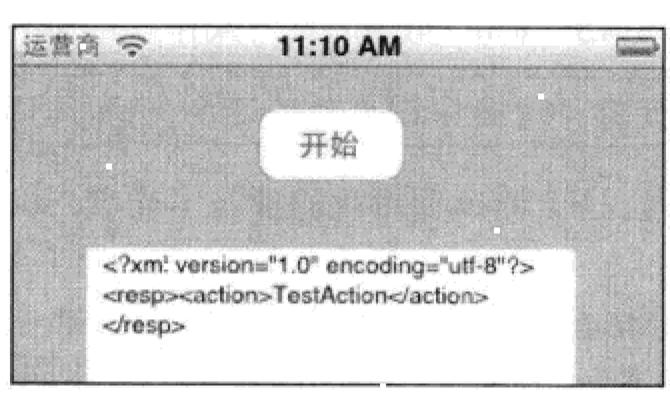


图 7-10 运行测试项目

提示： 在运行测试项目时，请确保服务端代码已在运行。

7.6 本章小结

iPhone SDK 对于网络的支持是充分的，提供了很多类，如 NSURL、NSURLRequest、NSURLConnection，甚至 NSString。除了 NSURLConnection 有稍微复杂的委托方法外，这些类的使用仍然是容易掌握的。

此外还有 NSOperation，这是 iOS SDK 的多线程类，它并不仅仅是在网络编程中使用。但网络应用中使用得最多的异步技术正是基于多线程技术实现的，因此 NSOperation 的使用在本章也有所提及。在多线程的操作中，线程通知技术是关键的，比如 KVO 和委托回调，本章都

有所提及，读者还可以回顾本书其他章节的介绍。

当然仅仅使用 SDK 内置网络访问技术仍然是低效的，我们仍然有一些功能完善、封装完好的第三方框架可以选择。本章剩下的篇幅介绍了一个优秀的第三方框架——ASIHttpRequest，从实用的角度对该框架进行了全面介绍。它对 HTTP 编程的支持是全面的，包括同步请求、异步请求、多线程和队列操作、Get 和 Post 请求、Cookies、TSL/SSL 各个方面。总之，使用 ASIHTTPRequest 框架，可以大大简化我们在 iOS 开发中对网络进行访问的代码，提高编码效率。

最后，我们介绍了如何在 ASIHTTPRequest 框架的基础上做进一步的封装，编写出自己的网络访问模块类 NetworkModule 和 PostRequest。读者可以修改这些类，并将它们应用到今后的项目中。

PDG

第 8 章　XML 和 Json

上一章我们介绍了网络，iPhone 和服务器进行网络数据交换的过程中，通常都要涉及数据格式的转换。本章就来介绍这方面的内容。

在诸多网络数据格式交换中，XML 和 Json 无疑是最典型的两种。前者使用广泛，已经成为了网络数据交换中的事实标准；后者在一些主流的网络开发语言中得到广泛支持。在这一章，将对这两种使用最为普遍的数据交换格式（尤其是 XML）在 iOS 网络编程中的应用进行介绍。

对于 XML，Cocoa 提供了专用于 XML 解析的类 NSXMLParser 及其委托 NSXMLParserDelegate。此外，第三方的支持也是非常丰富的。本章介绍了其中最优秀的几种，如 TBXML、libxml、GDataXML，尤其是对 libxml 和 GDataXML 进行了重点介绍。在 libxml 介绍部分，作者结合上一章介绍的 ASIHTTPRequest 框架，以及未完成的 RESTServer 项目，完成了一个 XML SAX 解析实现网络登录的示例程序。在 GDataXML 介绍部分，作者利用上一章介绍的自定义网络模块和 RESTServer 项目，结合 GDataXML 进行了树视图的展示及操作，包括：树的显示和遍历、节点的展开/折叠、节点的选择、节点状态的转换（未选/半选/全选）等。

对于 Json，Cocoa 没有提供内置的支持。但我们仍然可以找到第三方支持，如 SBJson 和 JSON Framework。本章介绍了 SBJson。

最后，我们结合上一章中介绍的自定义网络模块，和本章介绍的 GDataXML，完成了一个有趣小例子——在 iPhone 上完美地展现和操作（折叠/展开/选择）一棵 XML 树。

8.1　Cocoa 与 XML 解析

Cocoa 用 NSXML 来提供对 XML 解析的支持，包括 NSXMLParser 和 NSXMLParserDelegate。下面分别介绍它们。

8.1.1　NSXMLParser

首先我们初始化一个 NSXMLParser 对象：

```
NSXMLParser *parser = [[NSXMLParser alloc] initWithData:xmlData];
```

NSXMLParser 总共有 3 个初始化方法，它们分别使用不同的参数进行初始化：

- - (id)initWithContentsOfURL:(NSURL *)url；使用一个 NSURL 作为初始化参数，这个 URL 指向了网络中的 XML 资源。
- - (id)initWithData:(NSData *)data；使用 NSData 作为初始化参数，这个 NSData 包含了 XML 文件的内容。

PDG

- - (id)initWithStream:(NSInputStream *)stream；使用一个 NSInputStream 作为初始化参数，XML 文件的内容来自于输入流。

然后，设置 NSXMLParser 的代理，开始解析：

```
[parser setDelegate:self];
[parser parse];
```

所有的解析工作，通过这个代理对象进行。代理对象可以是任何对象（id 类型，比如 UIViewController 或者 NSObject），但必需实现 NSXMLParserDelegate 协议。

8.1.2 NSXMLParserDelegate

NSXMLParserDelegate 是一个 NSXMLParser 的委托者需要实现的协议，它总共定义了 20 个可选的委托方法，委托者可以根据需要实现。

例如，- parser:didStartElement:namespaceURI:qualifiedName:attributes:方法。该方法在解析到一个元素开始标签时调用，我们可以在这个委托方法中判断是否是我们需要的标签，如果是，则初始化对象或变量，获取标签属性值等。

```
static NSString *kName_Question = @"question";
static NSString *kName_Title    = @"title";
static NSString *kName_Reverse  = @"reverse";
- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName nam
        espaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qualifiedName
        attributes:(NSDictionary *)attributeDict {
   if ([elementName isEqualToString:kName_Question]) {
     QuestionItem *item = [[QuestionItem alloc] init];
     self.currentQuestion = item;
     [item release];
   } else if ([elementName isEqualToString:kName_Title] || [elementName isEqualT
          oString:kName_Reverse]){
     [currentString setString:@""];
     storingChar = YES;
   }
}
```

- parser:foundCharacters:方法在解析到标签体（开始标签和结束标签之间）文本时调用。我们可以在这个方法中将解析出来的文本（string 参数）添加到临时的字符串对象中：

```
- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string {
   [self.currentString appendString:string];
}
```

- parser:didEndElement:namespaceURI:qualifiedName:方法在解析到元素结束标签时调用。在这个方法中，我们应该保存解析出来的成果，重新初始化临时变量：

```
- (void)parser:(NSXMLParser *)parser didEndElement:(NSString *)elementName names
```

```
        paceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName {
    if ([elementName isEqualToString:kName_Question]) {
        [self finishedCurrentQuestion];
     } else if ([elementName isEqualToString:kName_Title]) {
        self.currentQuestion.title = currentString;
     } else if ([elementName isEqualToString:kName_Reverse]) {
        self.currentQuestion.reverse = [currentString boolValue];;
     }
      self.currentString = [[NSMutableString alloc]init];
}
```

NSXMLParser 的使用是简单的，而其代理 NSXMLParserDelegate 则要复杂得多。NSXMLParser 采用了 SAX 方式解析，其工作原理实际上是同 libxml 完全一致的，都是基于事件回调方式的解析，因此我没有在这里深入介绍，等后面介绍 libxml 部分读者还有机会详细了解。

对于 NSXMLParser 及其代理，我们只简单介绍到这里。考虑到它糟糕的性能（甚至比我们将要重点介绍的 GDataXML 还要不堪），我们不会把它作为本章的重点。你大概了解它就行。

8.2 TBXML

TBXML 是一个轻量级的 DOM 解析器，使用 Objective C 编写。说它轻量级，是因为它总共只有 4 个源文件：

- TBXML.h——tbxml 声明；
- TBXML.m——tbxml 实现；
- NSDataAdditions.h——NSData 类别的声明；
- NSDataAdditions.m——NSData 类别的实现，包括 base64，gzip 的实现。

TBXML 的接口定义得非常简洁，程序员使用它提供的数量非常有限的方法去完成所有的事情（总共 7 个静态方法）。你想不出任何理由来不使用它。

在所有知名的解析器当中，TBXML 以资源占用少，解析速度快闻名。但它的缺点和优点一样突出，它是只读的，不支持对 XML 文档的写入操作。如果你需要一种简单、高效的解析器，则 TBXML 是一种不错的选择。

TBXML 的项目地址：http://www.tbxml.co.uk/TBXML/TBXML_Free.html

下载 TBXML 后，将 4 个文件添加到项目中，并引入 libz.dylib（或 libz.1.2.3.dylib）框架，即可在代码中使用 TBXML。

我们以一小段代码来演示 TBXML 的使用，完整的项目代码位于光盘“source/第 8 章/testTBXML”目录中。

```
-(IBAction)parseXML{
    [tv resignFirstResponder];// 释放键盘
    tvResult.text=nil;
```

```
    NSMutableString* string=[[NSMutableString alloc]init];
    TBXML* xml=[TBXML tbxmlWithXMLString:tv.text];
     //取得根元素
     TBXMLElement* root=xml.rootXMLElement;
     //取得 root 第 1 个子节点 name
     TBXMLElement* node=[TBXML childElementNamed:@"书名" parentElement:root];
     //遍历 name 节点及其所有兄弟节点
     while (node) {
        [string appendString:[NSString stringWithFormat:@"%@=%@",[TBXML element
                Name:node],[TBXML textForElement:node]]];
        [string appendString:@"\n"];// 换行
        node=node->nextSibling;//[TBXML nextSiblingNamed:@"bill" searchFromElement:
              root];
     }
     tvResult.text=string;
}
```

程序运行效果如图 8-1 所示。在"TBXML 解析"按钮上方的文本框中有一小段 XML，当你点击按钮后，这段 XML 会被解析，并在按钮下方的文本框中显示出来：

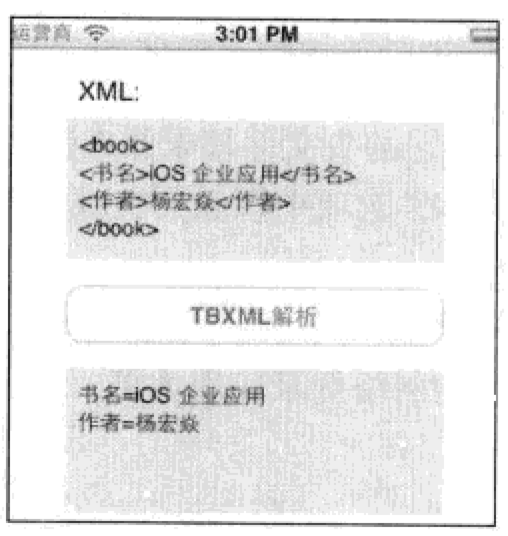


图 8-1 程序运行结果

TBXML 的使用非常简单，它提供了非常简洁的 API（总共只有 7 个静态方法）。它的优点是解析速度非常快——在本章介绍的 4 种 XML 解析框架中，TBXML 是速度最快的。缺点是功能有限，只读，不提供写 XML 接口。TBXML 适用于简单 XML 文件的解析。

8.3 libxml

有许多知名的 XML 解析器可供我们选择，比如：TBXML、TouchXML、KissXML、TinyXML 和 GDataXML。那么我们选择 libxml 的理由是什么？

XML 解析分为两类，一类是 DOM 解析，另一类为 SAX 解析。前者如 GDataXML 和 TBXML，解析过程中需要建立文档树，操作 XML 元素时通过 DOM 树的树形结构一级一级自

顶向下进行导航。DOM 解析的优点是便于程序员理解 XML 文档树的结构，API 的使用也较简单；缺点是速度较 SAX 解析慢，且内存开销较大。在某些情况下，比如 iPhone 开发，受制于有限的内存空间（一个应用最多可用 10 几兆的内存），DOM 解析性能较差（当然，在模拟器上是没有问题的）。NSXMLParser 和 libxml 属于 SAX 解析（其实 libxml 既支持 DOM 解析也支持 SAX 解析，不过我们使用 libxml 的原因，仅仅是为了使用它的 SAX 解析）。

libxml 是一个开放源码库，默认情况下 iPhone SDK 中已经包括在内。 它是一个基于 C 的 API，以 C 回调函数的方式使用，所以在使用上比 Cocoa 的 NSXMLParser 要麻烦一些。NSXMLParser 也实现了 SAX，但与 libxml 相比，NSXMLParser 提供的是一个面向对象的接口（Objective C 语言接口），而 libxml 提供的是 C 语言接口。对于 Objective C 程序员来说，使用 C 语言接口显然不如使用 O-C 接口便利。但是 libxml 同时支持 DOM 和 SAX 解析，其解析速度较快，而且占用内存小，是最适合使用在 iPhone 上的解析器。从性能上讲，所有知名的解析器中，TBXML 最快，libxml 次之。但在内存占用上，libxml 使用的内存开销是最小的。因此，使用 libxml 的 SAX 接口进行 XML 解析在很多情况下（需要兼顾速度和内存的时候），是个不错的选择。

8.3.1 在项目中使用 libxml

首先，我们需要在项目中导入 framework：libxml2.dylib。

虽然 libxml 是 SDK 中自带的，但它的头文件却未放在默认的地方，因此还需要我们在 Build Settings 的 HEADER_SEARCH_PATHS 选项中添加头文件搜索路径：/usr/include/libxml2，否则 libxml 库不可用。

然后，我们就可以在源代码中输入#import <libxml/tree.h>了。

接下来，我们将以一个最常见的网络登录应用为例，用 libxml 实现 iPhone 与服务器的基于 XML 的交互。

8.3.2 libxml 应用实例

还记得上一章“编写自己的网络模块类”吗？我们曾在其中用 Java 实现了一个 REST 服务器。现在，我们需要在这个 REST 服务器中增加一个登录验证功能。

1. 实现服务端

首先，在 Eclipse 中打开这个 Web 项目（项目位于光盘“source/第 7 章/RESTServer”目录下）。

我们需要在 Interface.java 的 doPost 方法中增加一个 Action。在这个 Action 中，我们验证用户名和密码是否为空。代码如下所示：

```
} else if("LoginAction".equals(action)){
    Node anode=doc.selectSingleNode("req/params/username");
    if(anode!=null && !"".equals(anode.getStringValue())&&
        anode.getStringValue()!=null){
```

```
            anode=doc.selectSingleNode("req/params/password");
            if(anode!=null && !"".equals(anode.getStringValue())&&
               anode.getStringValue()!=null){
            resptext="<resp><loginstatus>true</loginstatus></resp>";
        }else{
          resptext="<resp><loginstatus>密码错误</loginstatus></resp>";
        }
    }else{
        resptext="<resp><loginstatus>用户名错误</loginstatus></resp>";
    }
```

编译项目，并发布到 Tomcat 中进行测试。启动服务器后，打开 FireFox，在工具菜单中找到 Poster 这个插件。我们准备用 Poster 火狐插件来进行 REST 客户端的测试。如果你没有安装 Poster 插件，可以用“附加组件”管理来安装它。

在 Poster 窗口，在 URL 中输入 http://localhost:8080/RESTServer/Interface，在 Content 中（最下面的文本框）输入 XML 文本：

```
<req>
<action>LoginAction</action>
<params>
<username>1</username>
<password>1</password>
</params>
</req>
```

然后点击 Post 按钮，将会弹出一个窗口，显示服务器返回的 XML：

```
<?xml version="1.0" encoding="utf-8"?>
<resp><loginstatus>true</loginstatus></resp>
```

如果在 Content 中输入了空的用户名，那么你将得到服务器返回的 XML：

```
<?xml version="1.0" encoding="utf-8"?>
<resp><loginstatus>用户名错误</loginstatus></resp>
```

如果 Content 中密码为空，服务器返回的 XML 则是“密码错误”。

接下来是 iPhone 代码。

2. 界面设计（UI）

新建 Empty Application 项目 testLibxml(该项目位于光盘“source/第 8 章/testLibxml”目录)。

首先，我们需要在项目中加入 libxml 框架的支持。打开 Target 的 Build Phases 窗口，在 Link Binary With Libraries 中加入 libxml2.dylib 库。然后在 Poject 的 Build Settings 窗口找到 Header Search Paths，加入路径：/usr/include/libxml2。

新建视图控制器 testVC，勾选上“With XIB…”选项。拖 2 个 Text Field 和 1 个 UIButton 到画布中，如图 8-2 所示。

图 8-2 testVC.xib 的 UI 设计

声明出口：

```
@property (retain,nonatomic)IBOutlet UITextField* tfName;
@property (retain,nonatomic)IBOutlet UITextField* tfPass;
-(IBAction)loginAction;
```

加入@synthesize 语句：

```
@synthesize tfName,tfPass;
```

回到 IB，创建 IBOutlet 连接和 IBAction 连接。

我们将使用 ASIHTTPRequest 框架进行 HTTP 请求，请导入 ASIHTTPRequest 框架（拷贝框架源文件到项目中并加入相关框架）。

新建另一个视图控制器 loginStatusVC（使用“With XIB…”选项）。在画布中拖入一个 Label 并连接到出口 label，我们将用这个视图控制器作为登录是否成功的显示页面。

3. 实现网络接口（使用 ASIHTTPRequest）

回到 testVC 类。在头文件中分别导入 ASIHTTPRequest 框架和 libxml 库的头文件：

```
#import "ASIHTTPRequest.h"
#import <libxml/tree.h>
```

声明必要的成员：

```
ASIHTTPRequest* _request;
NSStringEncoding enc;
NSMutableData* mData;
NSString* login_status;
```

现在，来实现 loginAction 方法：

```
-(IBAction)loginAction{
    NSString* url=@"http://localhost:8080/RESTServer/Interface";
    NSString* xml=[NSString stringWithFormat:@"<req>\
            <action>LoginAction</action>\
            <params>\
            <username>%@</username>\
```

```
        <password>%@</password>\
        </params>\
        </req>",tfName.text,tfPass.text];
    NSLog(@"%@,%@",tfName.text,tfPass.text);
    [self postXML:xml url:url];
}
```

提示：反斜杠“\”是 C 语言的续行符。

在 loginAction 方法中，我们创建了两个 String，分别是请求的 URL 地址和发送给服务器的 XML 内容。然后调用 postXML:url:方法。

postXML:url:方法实现如下：

```
-(void) postXML:(NSString*) xml url:(NSString*)url{
    _request=[[ASIHTTPRequest requestWithURL:
            [NSURL URLWithString:url]]retain];
    [_request setShouldAttemptPersistentConnection:NO];
    [_request setResponseEncoding:enc];
    NSMutableDictionary *reqHeaders = [[NSMutableDictionary alloc] init];
    [reqHeaders setValue:@"text/xml; charset=UTF-8" forKey:@"Content-Type"];
    _request.requestHeaders = reqHeaders;
    [reqHeaders release];
    NSLog(@"post xml:%@",xml);
    // 重要
    [_request appendPostData:[xml dataUsingEncoding:enc]];
    [_request setDelegate:self];
    // 创建解析器上下文
    _parserContext = xmlCreatePushParserCtxt(&_saxHandlerStruct, self, NULL, 0,
                    NULL);
    [_request startAsynchronous];
}
```

这个方法的实现并不陌生，它使用 ASIHTTPRequest 框架来发送异步请求（用 self 作为 delegate 代理）——在上一章中我们已经介绍过 ASIHTTPRequest 的使用。

我们在 testVC 中实现了 ASIHTTPRequestDelegate 协议

首先是 request:didReceiveData:方法：

```
- (void)request:(ASIHTTPRequest *)request didReceiveData:(NSData *)data{
    [mData appendData:data];
}
```

然后是 requestFinished：方法：

```
- (void)requestFinished:(ASIHTTPRequest *)request
{
    NSString* string=[[NSString alloc] initWithData:mData encoding:enc];
    [self openStatusVC];
}
```

这两个方法充分体现了“异步”的概念。我们知道，在 ASIHTTPRequest 的同步执行中，可以使用 request.responseData 属性一次性获得服务器返回的数据。但在异步执行中你不能这样做。因为请求结束，ASIHTTPRequest 可能会清空缓存。这会导致你在 requestFinshed 方法中访问 request.responseData 属性时只会得到一个 nil 对象。因此，应当像上面的代码一样，先在 request:didReceiveData:方法中把接收到的 data 参数依次添加到 NSMutableData 对象，然后在 requestFinished:方法（请求结束）时通过这个 NSMutableData 得到完整的数据。

提示：实际上，只有 delegate 实现了 request:didReceiveData:方法时，ASIHTTPRequest 框架才会清空 responseData 属性。如果 delegate 不实现 request:didReceiveData:方法，则不会清空 responseData 属性。因此，我们也可以不实现 request:didReceiveData 方法，而直接在 requestFinished 方法中获取 responseData 属性。

---

openStatusVC 方法实现了页面的跳转，并显示登录成功/失败信息：

```
-(void)openStatusVC{
    if (login_status) {
        loginStatusVC* vc=[[[loginStatusVC alloc]init]autorelease];
        [self.navigationController pushViewController:vc
                                         animated:YES];
        if ([@"true" isEqualToString:login_status]) {
            vc.label.text=@"登录成功";
        }else{
            vc.label.text=login_status;
        }
    }
}
```

最后，修改 AppDelegate 的 application:didFinishLaunchingWithOptions:方法，在应用程序启动后加载我们的 testVC 视图控制器：

```
self.window.rootViewController=[[UINavigationController alloc]initWithRootView
     Controller:
     [[testVC alloc]init]];
```

程序已经可以运行了，然而结果并不正确。我们并没有对服务器的 XML 进行解析。接下来我们就是要利用 libxml 完成 XML 的解析。

4. XML 解析（使用 libxml）

在 testVC.h 中添加新的成员声明：

```
xmlParserCtxtPtr _parserContext;
NSMutableString* mString;
int tag;
```

我们声明了一个 xmlParserCtxtPtr 类型的变量_parserContext。xmlParserCtxtPtr 是 libxml

定义的一种指针类型，它指向了一个 xmlParserCtxt 结构。这个结构很复杂（超过 100 个成员），但最重要的是最开始的两个：

- struct _xmlSAXHandler * sax——xmlSAXHandler 结构体，你很快就会见到它了。后面我们会定义一个 xmlSAXHandler 结构体，定义了我们将要实现的 SAX 函数。
- void *userData——保留给用户使用。这个 userData 很重要。因为我们可以把一个 void* 类型（即 id 类型）传给它。在后面，我们会演示把一个 self（即 testVC 实例）传给它。这样我们就可以在 SAX 函数中取出 userData 并把它当作 testVC 使用——这给我们一个机会，在 SAX 函数中调用 testVC（或者任何你自己编写的 O-C 类）的方法。

要使用 libxml 需要大概经过 4 个步骤：

1）声明要实现的 SAX 事件处理函数。

2）创建 xmlSaxHandler 结构体。

3）创建 xmlParserCtxPtr。

4）调用 libxml 函数，实现 SAX 解析的异步处理。

下面，我们继续完成 testLibxml 项目，演示 libxml 的使用。

我们先声明 3 个静态函数的实现。因为 SAX 这 3 个静态函数将对应我们要实现的 3 个 handler 函数（SAX 事件处理函数）。在此，我们先声明 3 个空实现，具体的实现我们稍后介绍：

```
static void startElementHandler(
         void* ctx,
         const xmlChar* localname,
         const xmlChar* prefix,
         const xmlChar* URI,
         int nb_namespaces,
         const xmlChar** namespaces,
         int nb_attributes,
         int nb_defaulted,
         const xmlChar** attributes)
{
}
static void endElementHandler(
         void* ctx,
         const xmlChar* localname,
         const xmlChar* prefix,
         const xmlChar* URI)
{
}
static void charactersFoundHandler(
         void* ctx,
         const xmlChar* ch,
         int len)
{
}
```

然后是 xmlSAXHandler 结构体。这个结构体有大约 30 几个成员，定义了 Handler 可以实现（不必全部实现）的所有 SAX 事件处理函数。这些函数实际上是 libxml 回调函数。当 libxml 在读取 XML 文件的过程中，每当发生一个 SAX 事件，则会去调用该结构体中定义的某个 Handler 函数。如果对应的结构体成员为 NULL，则不调用。

由于我们前面只定义了 3 个函数，所以这个结构体中的 3 个成员以 3 个静态函数填充，其余成员都以 NULL 填充（除了 XML_SAX2_MAGIC 外）：

```
static xmlSAXHandler _saxHandlerStruct = {
    NULL, /* internalSubset */
    NULL, /* isStandalone   */
    NULL, /* hasInternalSubset */
    NULL, /* hasExternalSubset */
    NULL, /* resolveEntity */
    NULL, /* getEntity */
    NULL, /* entityDecl */
    NULL, /* notationDecl */
    NULL, /* attributeDecl */
    NULL, /* elementDecl */
    NULL, /* unparsedEntityDecl */
    NULL, /* setDocumentLocator */
    NULL, /* startDocument */
    NULL, /* endDocument */
    NULL, /* startElement*/
    NULL, /* endElement */
    NULL, /* reference */
    charactersFoundHandler, /* characters */
    NULL, /* ignorableWhitespace */
    NULL, /* processingInstruction */
    NULL, /* comment */
    NULL, /* warning */
    NULL, /* error */
    NULL, /* fatalError */
    NULL, /* getParameterEntity */
    NULL, /* cdataBlock */
    NULL, /* externalSubset */
    XML_SAX2_MAGIC,  /* initialized */
    NULL, /* private */
    startElementHandler,    /* startElementNs */
    endElementHandler,      /* endElementNs */
    NULL, /* serror */
};
```

注意： 函数或者结构体都是在 @implement 以外声明的。函数是 C 语言的概念，它不是类的一部分（C 没有类和对象的概念）。

接下来，我们讨论如何实现刚才声明的 3 个 SAX 函数。这 3 个函数你可以直接实现——这跟实现一般的 C 语言回调函数没有任何区别，但很显然，这对于 O-C 程序员来说，会很不习惯。因此，我们决定在 3 个函数中调用方法来实现，这样，真正的 C 语言实现就可以改为在类的方法中实现了。在 O-C 中，以函数的方式调用对象方法是容易的，我们可以用 objc_msgSend()_函数调用一个方法。

objc_msgSend 函数用于向对象发送消息。它的第一个参数指向消息的接收者，第二个参数是一个 selector，第三个参数是一个不定参数，指向参数列表。

注意：要使用 objc_msgSend 函数，需要导入头文件"<objc/message.h> "。

因此第一个 SAX 事件函数可以这样实现：

```
SEL sel=NSSelectorFromString(
@"startTag:prefix:URI:nb_namespaces:namespaces:nb_attributes:nb_defaulted:attri
                    butes:");
if ([(id)ctx respondsToSelector:sel]) {
        objc_msgSend(ctx, sel,localname,prefix,URI,nb_namespaces,namespaces,nb_
                    attributes,nb_defaulted,attributes);
}
```

首先，我们构建一个指向 startTag 方法的 selector。startTag 方法拥有 8 个参数（即 8 个冒号），如果你注意看，这 8 个参数跟 startElementHandler 函数的后 8 个参数完全一致的（见前面介绍的 3 个静态函数声明）。这是有意的，因为我们将委托对象来实现该函数的具体细节。

至于函数的第 1 个参数，则有点特殊，它不在方法的参数里。因为它实际上就是消息接收者 ctx。由于在调用 objc_msgSend 函数时，我们会把消息接收者 ctx 在第一个参数中单独传递，所以就没有必要再在方法参数中额外再传递 ctx 了。

然后，我们判断接收者是否能响应该 selector。如果是（当然应该是，否则我们前面的工作就白做了，startElementHandler 还是空实现），则使用 objc_msgSend 函数向接收者发送消息。

这里需要说明 startElementHandler 函数的第 1 个参数 ctx。我们说过 ctx 实际上是消息接收者，是因为后面在初始化 testVC 类的_parserContext 成员时，我们把 self（testVC 实例）传递给了 xmlParserCtxt 的 userData 成员。这个成员会在调用 SAX 事件函数时（包括 startElementHandler）作为第 1 个参数传递。这样在 startElementHandler 函数中我们就有了 ctx 可用，并且知道它其实就是一个 testVC 实例。但由于我们想让代码更通用一些，所以没有把 ctx 强制转换为 testVC*类型，而仅仅是转换成 id 类型。这样，我们就无法肯定 ctx 会不会有一个 startTag 的方法可用。所以我们用 respondsToSelector 方法去推断 ctx 到底有没有实现 startTag 方法，如果实现则调用。没有实现也没关系，大不了不调用罢了。

接着，endElementHandler 和 charactersFoundHandler 函数的情况也是类似的，我们也将函数分别委托给 ctx（实际上就是 testVC）的 endTag 和 textTag 方法处理。

startTage 方法：

```
mString=[[NSMutableString alloc]init];
    // loginstatus
    if (strncmp((char*)localname, "loginstatus", sizeof("loginstatus")) == 0) {
        tag=1;
    }else{
        tag=0;
    }
```

textTag 方法：

```
if (tag) {
        [mString appendString:[[NSString alloc] initWithBytes:ch length:len encoding:
                enc]];
    }
```

endTag 方法：

```
if (tag==1) {
        login_status=mString;
}
tag=0;
```

对于 libxml 来说，这 3 个方法的回调的时机截然不同的。libxml 拦截 SAX 解析过程中的各种事件，当事件发生时，把事件交由实现了 SAX 接口的对象去处理。我们的 testVC 实际上就是实现了 SAX 接口的对象——我们把 SAX 事件处理函数中的 3 个交给了 testVC 的 3 个方法实现，这样，这 3 个方法就分别对应了 SAX 解析过程中的 3 个重要事件的处理过程。这 3 个最主要的 SAX 事件分别是每个 XML 元素的：开始标签解析事件、标签体解析事件、结束标签解析事件。SAX 中有许多的事件，但绝大部分时间，我们只需要处理这 3 个事件。因为很多时候，我们只会对 XML 文件中的元素属性和内容感兴趣，而通过这 3 个事件已经足以使我们读取到 XML 节点的属性和内容。

以元素“<message>用户名或密码错</message>”的解析为例。整个元素可以分为 3 个部分：<message>是开始标记，</message>是结束标签，开始标签和结束标签之间的文本为元素体。

libxml 首先读取元素的开始标记“<message>”，这标志着一个元素的处理周期开始了，同时触发“开始标签解析事件”，并调用 Handler 的开始标签处理函数。程序员应该在此函数中识别是哪一个开始标签，同时，由于元素属性也是开始标记的一部分，也可以在此读取元素属性（如果元素有属性的话）；

回调结束，libxml 开始读取标签体“用户名或密码错”，并（多次）调用标签体处理函数，程序员应该在此函数中获取、保存元素体文本。注意这个函数可能会“多次”调用，特别是标签体中存在多种字符集比如“中英文”混合的情况，libxml 会把标签体分多次读取，并多次调用标签体处理函数，直到整个标签体都读取完；

标签体回调结束后，libxml 开始读取标签结束标记“</message>”，并调用 SAX 解析器的元素结束标签处理函数，程序员在此函数中应标志一个标签处理结束，并重置标志变量。这样一个 XML 元素就处理完成了。

好了，我们已经完成了一半的工作（4 个步骤中的 2 个）。现在看第 3 步，创建 SAX 解析上下文，这个工作当然应该是开始接收到数据时（即按钮被点击后）进行，我们把代码放在了 postXML 方法里：

```
_parserContext = xmlCreatePushParserCtxt(&_saxHandlerStruct, self, NULL, 0, NULL);
```

调用 xmlCreatePushParserCtxt 函数可以创建一个 SAX 解析上下文。这个方法的第 1 个参数和第 2 个参数有特别的意义。前面我们已经解释过了，第 1 个参数指定一个 xmlSAXHandler 结构（这个结构中的每个成员都代表不同的 SAX 事件处理函数，你把你要实现的函数填充到合适的位置）；第 2 个参数指定了实现 SAX 函数的基地址（如果是用对象来实现，即对象的地址）。

最后一步，调用 libxml 函数实现 XML 异步读取。很显然，这必需和 ASIHTTPRequest 的异步请求回调打交道。

首先，将 request:didReceiveData:方法代码改为：

```
xmlParseChunk(_parserContext, (const char*)[data bytes], [data length], 0);
```

xmlParseChunk 函数把网络中异步读到的数据（即 data 对象）传递给 libxml 进行解析。

然后将 requestFinished:方法代码改为：

```
xmlParseChunk(_parserContext, NULL, 0, 1);
    // 释放 XML 解析器
    if (_parserContext) {
        xmlFreeParserCtxt(_parserContext), _parserContext = NULL;
    }
    [self openStatusVC];
```

这样，当请求数据结束，我们调用 xmlParserChunck 函数把一个 NULL 传递给 libxml，以此来告诉 libxml 没有数据需要处理了，然后释放 SAX 解析上下文（即_parserContext）。释放 SAX 解析上下文使用 xmlFreeParserCtxt 函数。因为_parserContext 不是一个对象，要释放它，我们不能向它发送 release 消息而只能使用这种方法。

提示：当网络出错或者 testVC 被 unload 的时候，我们也应当释放 SAX 解析上下文。

程序运行结果如图 8-3 所示。

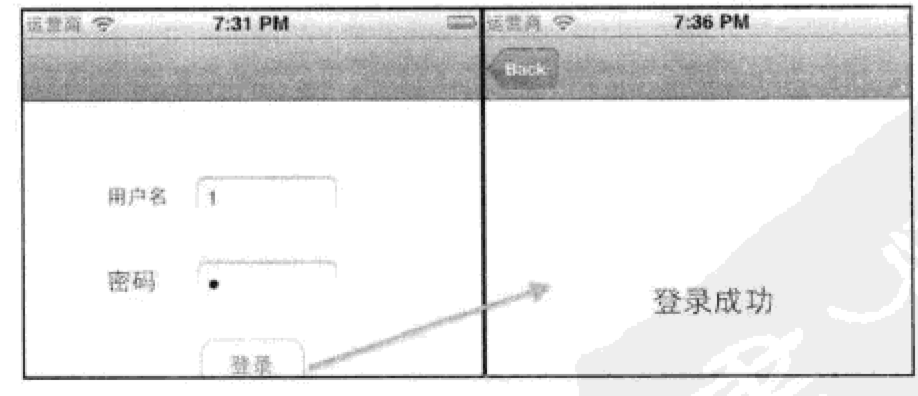


图 8-3　点击登录，转到登录成功/失败页面

提示：我们可以把 3 个 SAX 事件函数和 xmlSAXHandler 结构体声明拆分到一个头文件中。这样就可以在 testVC 类之外的其他类中共用这部分代码。我已经在示例项目中这样做了，你可以参考源文件的做法。

8.4 GDataXML

Google 的 GDataXML 可以说是所有 XML 解析框架中使用最为广泛的 DOM 解析 API，也是除了 TBXML 之外的最容易使用的。它同样只由两个文件组成：GDataxMLNode.h 和 GDataXMLNode.m。该项目地址为：http://code.google.com/p/gdata-objectivec-client/，你可以在这里下载到它。

与 SAX 解析不同，DOM 解析需要一次性读取完整的 XML 文件并形成 DOM 树。这显然要比 SAX 解析花费更多的 CPU 时间和内存。那么，既然我们有了像 libxml 这样的 SAX 解析框架，为什么还有必要介绍 GDataXML 这样 DOM 解析框架呢？

虽然 DOM 解析在性能和内存开销方面有着不如人意的表现，但在某些方面，也仍然有着 SAX 解析不具备的优点，如下所示：

- 有的 XML 文件中描述的数据（例如：描述组织结构的 XML），本身就是树型结构，将其解析为 DOM 树有着天然的优势，编码的效率更高、可读性更好；
- DOM 树是可读可写的，而 SAX 解析是只读的。在某些情况下，我们需要对数据进行某种修改，比如我们需要在某些节点上增加一些属性用于表示当前节点状态（比如：是否展开，是否被选中等等）。这在 DOM 操作中是很容易的、代价低廉的，因为 DOM 树一旦构建好，就已经成为一个内存对象，所有的数据修改就像修改内存对象一样简单，而且修改后的结果也很方便再次存储为 XML 文件。

下面我们将以一个示例项目，演示 GDataXML 使用。

这个项目位于光盘“source/第 8 章/testGData”目录。它的主要功能是从服务器上获取企业的组织结构数据（使用我们上一章介绍的自定义网络模块），然后进行展现。你可以在组织结构树上进行导航，展开或折叠某个节点，如图 8-4 所示。

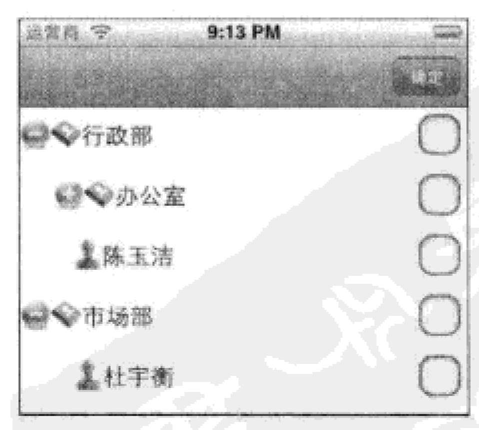


图 8-4 节点的展开/折叠

你可以在节点上进行操作，比如选择/反选某个机构下的所有人员，或单个人员（实际上，我们是对 DOM 树进行了写入操作），如图 8-5 所示。

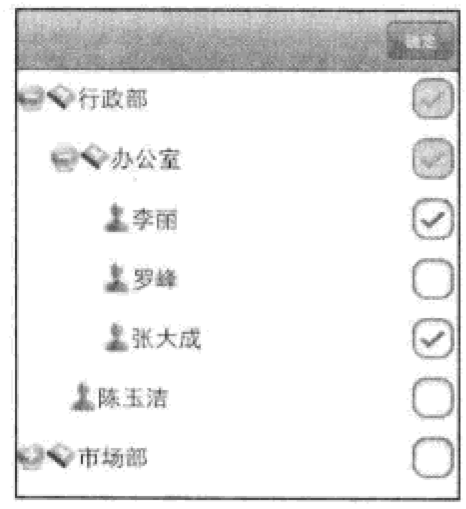


图 8-5 节点的选择/取消

下面介绍实现方法，首先是服务端代码。

1. 实现服务端

用 Eclipse 打开上一章的 RESTServer 项目，编辑 Interface.java，在 doPost 方法中加入以下代码：

```
else if("OrgTreeAction".equals(action)){
"<resp><Node type='1'>"+"\n"+
        "<Node type='1' name='行政部' id='U001' >"+"\n"+
        "<Node type='1' name='办公室' id='U004' >"+"\n"+
        "<Node type='0' name='李丽' id='P009'/>"+"\n"+
        "<Node type='0' name='罗峰' id='P101'/>"+"\n"+
        "<Node type='0' name='张大成' id='P652'/>"+"\n"+
        "</Node>"+"\n"+
        "<Node type='0' name='陈玉洁' id='P135'/>"+"\n"+
        "</Node>"+"\n"+
        "<Node type='1' name='市场部' id='U011' >"+"\n"+
        "<Node type='0' name='杜宇衡' id='P006'/>"+"\n"+
        "</Node>"+"\n"+
        "</Node></resp>";
}
```

打开 FireFox 的 Poster 工具进行测试，在 URL 栏输入 http://localhost:8080/RESTServer/Interface，Content 输入“<req><action>OrgTreeAction</action> </req>”，点击 POST 按钮。如果一切顺利，服务器将返回 XML：

```
<?xml version="1.0" encoding="utf-8"?>
<resp><Node type='1'>
<Node type='1' name='行政部' id='U001' >
<Node type='1' name='办公室' id='U004' >
```

```
<Node type='0' name='李丽' id='P009'/>
<Node type='0' name='罗峰' id='P101'/>
<Node type='0' name='张大成' id='P652'/>
</Node>
<Node type='0' name='陈玉洁' id='P135'/>
</Node>
<Node type='1' name='市场部' id='U011' >
<Node type='0' name='杜宇衡' id='P006'/>
</Node>
</Node></resp>
```

接下来是 iPhone 代码。

2. 将 GDataXML 和 NetworkModule 加入项目

新建 Empty Application。将 GDataxMLNode.h 和 GDataXMLNode.m 文件添加到项目目录。

GDataXML 使用了 libxml2 库，因此我们需要在项目中加入 libxml2。在 Link Binary With Libraries 中加入 libxml2.dylib。在 Header Search Paths 中添加路径：/usr/include/libxml2。

在项目中加入 ASIHTTPRequest 框架。把 ASIHTTPRequest 框架的源文件拷贝到项目目录，在 Link Binary With Libraries 中加入框架：libz.1.2.5、CFNetwork、MobileCoreServices、SystemConfiguration、Security。

将上一章 NetworkModule 项目中的下列源文件拷贝到项目中：

- NetworkModule.h 和 NetworkModule.m 文件
- PostRequest.h 和 PostRequest.m 文件
- SynthesizeSingleton.h 文件
- NetworkModuleDelegate.h 文件

3. 界面设计（UI）

新建视图控制器 TreeVC（使用“With XIB…”选项）。

修改 AppDelegate，在 application: didFinishLaunchingWithOptions:方法中加入 TreeVC 加载代码：

```
self.window.rootViewController=[[UINavigationController  alloc]initWithRootView-
Controller:[[TreeVC alloc]init]];
```

打开 TreeVC.xib，在画布中拖入一个 UITableView。在 TreeVC.h 中声明一个出口，将 UITableView 对象连接到出口。

在 TreeVC.h 中，导入 NetworkModule.h 并声明对 NetworkModuleDelegate 的实现：

```
#import <UIKit/UIKit.h>
#import "NetworkModule.h"
@interface TreeVC : UIViewController<NetworkModuleDelegate>{
}
@property(retain,nonatomic)IBOutlet UITableView* table;
```

```
@end
```

4. 实现网络接口（使用 NetworkModule）

编辑 TreeVC.m，在 viewDidLoad 方法中加入代码：

```
UIBarButtonItem* rightItem=[[UIBarButtonItem alloc]
            initWithTitle:@"确定"
            style:UIBarButtonItemStyleBordered
            target:self action:@selector(okAction)];
    self.navigationItem.rightBarButtonItem=rightItem;
    [rightItem release];
NSString* xml=@"<req><action>OrgTreeAction</action></req>";
    [[NetworkModule sharedNetworkModule]postBusinessReq:xml
        tag:kBusinessTagGetDeptPeople
        owner:self];
```

然后实现 NetworkModuleDelegate 的 3 个委托方法：

```
-(void)beginPost:(kBusinessTag)tag{
}
-(void)endPost:(NSString *)result business:(kBusinessTag)tag{
    if (tag==kBusinessTagGetDeptPeople) {
        NSLog(@"xml:%@",result);
    }
}
-(void)errorPost:(NSError *)err business:(kBusinessTag)tag{
}
```

在加入 GDataXML 进行 XML 的解析之前，我们需要先测试 NetworkModule 模块是否正常工作，运行程序，出现一个空白的表视图窗体。不要紧，现在我们还没有加入 XML 树，先查看控制台窗口，应当能够看到服务器返回的 XML 数据。

5. XML 解析（使用 GDataXML）

测试结果说明我们的自定义网络模块一切正常。现在我们需要对 NetworkModule 做一点点的改动，让它能直接返回 XML 解析的结果（即一个 GDataXMLELement 对象），而不是 NSString。

打开 PostRequest.h，将 result 属性由 NSString 类型改为 GDataXMLDocument 类型：

```
@property (nonatomic,readonly,getter = result)GDataXMLDocument* result;
```

然后修改 result 属性的 getter 方法（即-(GDataXMLDocument*)result 方法），将“return string;”一句修改为：

```
GDataXMLDocument* doc=[[GDataXMLDocument alloc] initWithXMLString:
                    string options:0 error:nil];
return doc;
```

打开 NetworkModuleDelegate.h，将协议方法 endPost:business 的定义改为：

```
-(void)endPost:(GDataXMLDocument*)result business:(kBusinessTag)tag;
```

提示：方法的第 1 个参数 result 修改为 GDataXMLDocument 类型,同时导入头文件 GDataXMLNode.h 。

同时，将 TreeVC.m 中 endPost:business:方法修改为：

```
-(void)endPost:(GDataXMLDocument *)result business:(kBusinessTag)tag{
    if (tag==kBusinessTagGetDeptPeople) {
        NSLog(@"xml:%@",result.rootElement.XMLString);
    }
}
```

提示：GDataXML 在解析之前会验证 XML 文件的有效性，请保证 XML 文件的合法性。如果 XML 文件格式不规范，比如属性值忘记用引号/双引号括住等，GDataXML 不会解析，并在控制台中给出明确的错误提示——这给我们及时发现问题提供了帮助。

---

6. 构建模型

接下来我们需要构建树的模型。新建两个 NSObject 类：XMLNode 和 OrgTreeNode，其中 OrgTreeNode 继承了 XMLNode。在 XMLNode 中，我们放一些公共的代码给子类共享，其头文件定义如下：

```
#import <Foundation/Foundation.h>
#import "GDataXMLNode.h"
@interface XMLNode : NSObject
@property (nonatomic, retain)GDataXMLElement *element;
@property (nonatomic, readonly, getter = hasChild)BOOL hasChild;
-(NSString*)stringValueFromPath:(NSString*)path;
-(NSString *) attributeForName : (NSString *)name;
-(void) setAttributeForName : (NSString *)name
                    value : (NSString *)value;
-(void)setStringValue:(NSString*)string ForPath:(NSString*)path;
@end
```

XMLNode 包含两个属性以及 4 个成员方法：

（1）element 属性

我们可以直接把一个 GDataXMLElement 对象赋值给它。它代表了一个树节点的 XML 数据。

（2）hasChild 属性

hasChild 属性用于表示该 XMLNode 对象是否包含子节点，BOOL 类型。hasChild 是只读的，它有一个自定的 getter 方法：

```
-(BOOL) hasChild
```

```
{
    if (_element != nil) {
        return ([_element childCount] > 0);
    } else {
        return NO;
    }
}
```

它通过计算节点的子节点数目是否为零来返回。

（3）stringValueFromPath 方法

该方法用于查找指定路径的子节点对象，并返回该子节点的文本内容。path 参数是一个 XPath 表达式字符串。关于 XPath，你应该有所了解，它是一种专用于查找、定位 XML 文档中某部分的查询语言。这是该方法的实现：

```
NSString* ret=nil;
    if (_element != nil) {
        NSArray* array=[_element nodesForXPath:path error:nil];
        GDataXMLElement* e=nil;
        if (array && array.count>0) {
            e=[array objectAtIndex:0];
        }else{
            return nil;
        }
        if (e!=nil) {
            ret=[e stringValue];
        }
    }
    return ret;
```

nodesForXPath:error:方法来自于 GDataXMLNode 类（GDataXMLElement 是它的子类，继承了这个方法）。该方法根据指定的 XPath 表达式在 GDataXMLNode 中查找指定节点并返回查找结果（一个数组，哪怕符合条件的节点只有一个）。很多时候，我们只需要用到数组索引 0 的节点对象，因为根据指定的 XPath 参数，nodesForXPath:error:方法只会返回单节点。

（4）attributeForName 方法

该方法返回 XMLNode 对象的指定属性值，用 name 参数指定属性名。该方法实现如下：

```
if (_element != nil && [_element attributeForName:name]!=nil) {
        return [[_element attributeForName:name]stringValue];
    } else {
        return nil;
    }
```

GDataXMLElement 的 attributeForName 返回指定属性值。

（5）setAttributeForName:value：方法

该方法通过指定属性名，向 GDAtaXMLElement 对象（即 XMLNode 的 element 属性）中

写入一个属性值；如果该属性不存在，则插入一个新属性：

```
if (_element != nil) {
    GDataXMLNode *attr  = [_element attributeForName: name];
    if (attr != nil) {
        [attr setStringValue: value];
    } else {
        attr = [GDataXMLNode attributeWithName: name stringValue: value];
        if (attr != nil) {
            [_element addAttribute: attr];
        }
    }
}
```

（6）setStringValue:ForPath:方法

该方法用于设置指定节点的文本值（通过 XPath 指定）：

```
GDataXMLElement* e=[[_element nodesForXPath:path error:nil]objectAtIndex:0];
if (e!=nil) {
     [e setStringValue:string];
}
```

接下来，我们实现 OrgTreeNode，OrgTreeNode 接口声明如下：

```
@interface OrgTreeNode : XMLNode
+(OrgTreeNode*)instanceWithXMLElement:(GDataXMLElement*)e;
-(id)initWithXMLElement : (GDataXMLElement *)e;
-(void)initNode : (GDataXMLElement *)node nodeLevel : (NSInteger)level;
@property (nonatomic, readonly, getter = name)NSString *name;
@property (nonatomic, readonly, getter = ID)NSString *ID;
@property (nonatomic, readonly, getter = type)int type;
@property (nonatomic, getter = level,   setter = setLevel:)NSInteger level;
@property (nonatomic, getter = check,  setter = setCheck:)int check;
@property (nonatomic, getter = expanded,  setter = setExpanded:)BOOL expanded;
@property (nonatomic,readonly,getter = showNodes)NSMutableArray* showNodes;
@property (nonatomic,readonly,getter = checkedPeople)NSMutableArray* checkedPeople;
-(void)refreshShowNodes;
-(void)getShowNodes:(GDataXMLElement*)node;
-(void)getSelectedPeople:(GDataXMLElement*)node;
-(void)checkParentAndChildren:(GDataXMLElement*)node checked:(BOOL)checked;
@end
```

首先是 OrgTreeNode 的 3 个实例化方法：

- instanceWithXMLElement:方法——这是 OrgTreeNode 的工厂化方法，我们的 OrgTreeNode 类有一些特殊，在实例化时一般不要直接调用 alloc 然后 init 的方法，而是用工厂化方法替代。在这个方法中又调用了以下两个方法来完成实例化。

- initWithXMLElement:方法——一个普通的实例化方法，但把传入参数 retain 住了。
- initNode:nodeLevel:方法——这是一个特殊的实例化方法。说它特殊，是因为它是一个递归方法。用递归的方式在整个树的每个节点上增加了一些必要的属性（用默认值设置属性）。

OrgTreeNode 有 3 个属性 name、ID、type 分别对应了 XML 文件中 Node 元素的 name、id、type 属性。XML 文件中 OrgTree 节点代表了企业的组织机构树，在 OrgTree 节点下有许多 Node 节点——不管是人员信息还是部门信息我们都用 Node 节点表示。Node 节点的 type 属性用于区分节点类型：为 1 代表示单位节点，为 0 表示人员节点。这是它们的 getter 方法（没有 setter 方法）：

```
-(NSString *) name
{
    return [self attributeForName:@"name"];
}
-(NSString*)ID{
    return [self attributeForName:@"id"];
}
-(int)type{// type=1,部门节点; type=0,员工节点
    NSString* string=[self attributeForName:@"type"];
    return [string intValue];
}
```

这 3 个属性定义为只读——它们来自于服务器返回的 XML，我们不应该修改它们。

为了方便对整个组织机构树的导航和操作，我们在 3 个只读属性之外额外定义了 3 个可读可写属性：level、check、expanded。分别用于表示节点在树中的深度（层级），节点是否被选中，以及是否展开了下级子节点。这 3 个属性在原始的 XML 文件中没有定义，是我们为了某种便利手动加上的。这是它们的 getter/setter 方法：

```
-(NSInteger) level
{
    return [[self attributeForName:@"level"]intValue];
}
-(void) setLevel : (NSInteger)value
{
    [self setAttributeForName: @"level"
       value: [NSString stringWithFormat:@"%d", value]];
}
-(int) check
{
    return [[self attributeForName : @"check"] integerValue];
}
-(void) setCheck: (int)value
{
```

```
    [self setAttributeForName: @"check"
      value: [NSString stringWithFormat:@"%d",value]];
}
-(BOOL) expanded
{
    return [[self attributeForName : @"expanded"] isEqualToString: @"1"];
}
-(void) setExpanded : (BOOL)value
{
    [self setAttributeForName:@"expanded"
         value: (value ? @"1" : @"0")];
}
```

可以看到，由于继承了父类（XMLNode）的方法，上面的代码实现起来是非常轻松的。

最后两个属性 showNodes 和 checkedPeople，一个用于返回树中所有处于显示状态的节点（数组），一个用于返回当前用户已选择的人员节点（不包括部门节点）：

```
-(NSMutableArray*)showNodes{
    return _showNodes;
}
-(NSMutableArray*)checkedPeople{
    [_checkedPeople release];
    _checkedPeople=[[NSMutableArray alloc]init];
    [self getSelectedPeople:self.element];
    return _checkedPeople;
}
```

showNodes 的 getter 方法很简单，就是返回了属性_showNodes。_showNodes 是属性 showNodes 的别名，当我们使用了 @synthesize showNodes=_showNodes;语句之后，当我们用 self.showNodes 引用 showNodes 属性的时候，可以用_showNodes 来代替（更简短）。

从这里看出，showNodes 属性不能单独使用，在你访问属性 showNodes(即调用 showNodes 方法）之前，我们起码要调用一下 getShowNodes:方法。这个方法会遍历树，把所有当前处于显示状态的节点罗列到数组中返回：

```
-(void)getShowNodes:(GDataXMLElement*)node{
    OrgTreeNode* dNode=[[[OrgTreeNode alloc]initWithXMLElement:node]autorelease];
    if (dNode.level>0) {
        [_showNodes addObject:node];
    }
    if (dNode.canExpand && dNode.expanded) {
        for (GDataXMLElement *e in node.children) {
            [self getShowNodes:e];
        }
    }
}
```

注意，getShowNodes:方法有一个 GDataXMLElement 参数。这个参数很有用，是遍历树的根本，如果没有它，我们就无法遍历树了。这个参数同时也指定了你要获取哪一个节点的可见节点。如果一个节点是展开的，那么它的可见节点应该包括两部分：

- 它自己。也就是[_showNodes addObject:node];这句。不过我们加了个条件，要 level 大于 0——意即根节点（根节点的 level 是 0）是不可见——一般来说显示一个组织机构的根节点是没有意义的（根节点很可能是抽象的，根本不存在这个部门）。
- 它的子节点。如果一个节点被展开，那么它的子节点也应该暴露给用户（可见）。当它的子节点也有子节点时，我们要递归，也就是这句：

```
for (GDataXMLElement *e in node.children) {
        [self getShowNodes:e];
}
```

checkedPeople 方法每次重新创建一个 NSMutableArray 用于保存所有已选人员节点。这会导致每次访问 checkedPeople（或调用 checkedPeople 方法）时，我们都会重新计算已选人员节点。同样，它调用了递归方法 getSelectedPeople:实现：

```
-(void)getSelectedPeople:(GDataXMLElement *)node{
    OrgTreeNode* dNode=[[[OrgTreeNode alloc]initWithXMLElement:node]autorelease];
    if (dNode.type==0 && dNode.check==2) {// 节点类型：人员节点；节点状态：全选
        [_checkedPeople addObject:dNode];
    }else{
        for (GDataXMLElement *element in node.children) {
            [self getSelectedPeople:element];
        }
    }
}
```

跟 getShowNodes:方法一样，一个节点所包含的已选人员节点也可以拆分为两部分：

- 节点本身。如果节点已经是一个已选择的人员节点。那么不需要多说，这个节点应当加入到_selectedPeople 数组中，也就是“[_checkedPeople addObject:dNode];”这句。同时，不需要考虑子节点了——因为人员节点不会再包含子节点；
- 如果节点是部门节点，那么要递归它的子节点，重来以上这两步过程，也就是“[self getSelectedPeople:element];”这句：

```
for (GDataXMLElement *element in node.children) {
        [self getSelectedPeople:element];
}
```

还有一个公开的成员方法 checkParentAndChildren:checked:方法：

```
-(void)checkParentAndChildren:(GDataXMLElement*)node checked:(int)checked{
  if (checked==2||checked==0) {
      OrgTreeNode* dNode=[[[OrgTreeNode alloc]initWithXMLElement:node]autorelease];
      dNode.check=checked;
```

```
        if (dNode.hasChild) {
            for(GDataXMLElement* e in node.children){
                [self checkParentAndChildren:e checked:checked];
            }
        }
        if(node.parent!=nil){
            OrgTreeNode* parent=[[[OrgTreeNode alloc]initWithXMLElement: (GDataXML-
                         Element*)node.parent]autorelease];
            [self setNodeCheckStatus:parent];
        }
    }
}
```

这个方法在一个节点的选中状态被用户改变时调用。这个方法需要使用两个递归：

- 向下级子节点递归，并改变下级子节点的选中状态；
- 向上级父节点递归，并改变上级父节点的选中状态。

当然，节点本身也需要设置自己的选中状态。

注意：这里使用了 GDataXMLNode 对象的 parent 方法。这个方法在最初的 GDataXMLNode 类中并不存在，是我们在后来增加到 GDataXMLNode 类中去的——读者可能需要更新自己的 GDataXML 源文件。

如果你奇怪没有在 checkParentAndChildren 方法中找到向上递归的代码，那么请注意“[self setNodeCheckStatus:parent];”这一句。setNodeCheckStatus:方法已经包含了一个向上递归：

```
-(void)setNodeCheckStatus:(OrgTreeNode*)node{
    if (node.level>0) {
        int status=[self getNodeCheckStatus:node];
        node.check=status;
          if (node.element.parent!=nil) {
            OrgTreeNode* dNode=[[[OrgTreeNode alloc]initWithXMLElement:
            GDataXMLElement*)node.element.parent]autorelease];
            [self setNodeCheckStatus:dNode];
        }
    }
}
```

在这个方法中，首先设置节点自身的 check 属性，然后向上递归父节点，并设置父节点的 check 属性。同样，我们过滤了根节点的处理（if(node.level>0)这句），因为根节点不用在视图中显示，也没有父节点。

现在，我们来看看如何在 View Controller 中展现这棵树。我们已经在 TreeVC 放了一个 UITableView，就是用于树的展现——用 Table View 的单元格来展现树的节点。这是一个挑战，因为这需要我们对单元格进行一些定制。

7. 定制单元格

首先，我们新建一个 NSObject 类 TreeNodeCell，继承 UITableViewCell。

新建一个 iOS→User Interface→View，名为 TreeNodeCell.xib。打开 Attributes Inspector，将 View 对象的 Size 属性改为 Freeform。打开 Identify Inspector，将 View 对象的 Class 修改为 TreeNodeCell。

提示：这会导致“Xcode4.2 以前版本不支持 Freedom size 属性”的警告，不用理它。你也可以消除它——将 File Inspector 中的“Document Versioning→Development”修改为“Xcode4.2”即可。

在 Size Inspector 中将 View 的 Height 属性改为 40。在画布上拖入 2 个 UIImageView，1 个 UILabel 和 1 个 UIButton，并将它们连接到 TreeNodeCell.h 的相应 IBOutlet 出口。设计后的 TreeNodeCell.xib 画布显示如图 8-6 所示。

图 8-6 TreeNodeCell 的 UI 设计

TreeNodeCell.h 定义如下：

```
@interface TreeNodeCell : UITableViewCell
+(TreeNodeCell *) getCell;
-(void)layoutSubviews;
@property (retain, nonatomic) IBOutlet UILabel *name;
@property (retain, nonatomic) IBOutlet UIButton *checkButton;
@property (retain, nonatomic) IBOutlet UIImageView *cellTypeImage;
@property (retain, nonatomic) IBOutlet UIImageView *expansionImage;
@property (nonatomic) int cellType;// 0: 人员节点, 1: 部门节点
@end
```

首先是getCell方法，这是一个工厂化方法，创建一个TreeNodeCell实例。首先用mainBundle的 loadNibNamed:方法加载 TreeNodeCell.xib 文件，这个方法将加载到内存中的所有 xib 文件中的对象放到一个 NSArray 中返回，这些对象就包括了我们在图 8-6 中看到的 View、ImageView、Label、Button。一般情况下，索引为 0 的对象就是根视图对象（即 View 对象）：

```
+(TreeNodeCell *) getCell
{
    NSArray *array = [[NSBundle mainBundle] loadNibNamed:@"TreeNodeCell" owner:self
                      options:nil];
    return  [array objectAtIndex:0];
}
```

然后是 layoutSubviews 方法，这个方法不用我们手动调用，它在刷新单元格时（比如 Table View 在 reloadData 的时候）自动调用：

```
-(void)layoutSubviews
{
    expansionImage.frame = CGRectMake(0 + self.indentationWidth * self. Indentation-
       Level - (cellType == 0 ? 9 : 0, expansionImage.frame.origin.y, expansionImage.
       frame.size.width, expansionImage.frame.size.height);
    if (cellType != 1) {
        expansionImage.hidden   = YES;
    }
    cellTypeImage.frame = CGRectMake(expansionImage.frame.origin.x+expansionImage.
         frame. size.width, cellTypeImage.frame.origin.y, cellTypeImage.frame.size.
         width, cellTypeImage. frame.size.height);
    name.frame = CGRectMake(cellTypeImage.frame.origin.x+cellTypeImage.frame.size.
        width, name.frame.origin.y,275-cellTypeImage.frame.origin.x, name.frame.
        size.height);
}
```

layoutSubviews 方法中，我们需要做几件事情：

- 根据单元格的 indentationLevel（缩进级别）计算第一张图片（"+/–"号图片）的位置。
- 如果单元格类型为"人员"（即 type=0），则不显示第一张图片（即"+/–"号图片）。
- 相对于第一张图片，计算第二张图片的位置（紧挨着第一张图片）。
- 相对于第二张图片，计算 UILabel 的位置（紧挨着第二张图片）。

TreeNodeCell 就介绍完了，接下来我们要在 TreeVC 中实现 TableView 的数据源方法以及展现树视图。

8. 实现 UITableViewDataSource 和 UITableViewDelegate

打开 TreeVC.h，将代码修改为：

```
#import <UIKit/UIKit.h>
#import "NetworkModule.h"
#import "OrgTreeNode.h"
@interface TreeVC : UIViewController<NetworkModuleDelegate,UITableViewDataSource,
            UITableViewDelegate>{
   OrgTreeNode* orgTree;
}
@property(retain,nonatomic)IBOutlet UITableView* table;
@end
```

可以看到我们增加了一个 OrgTreeNode 成员——这是树的模型，并声明实现了 UITableView DataSource 和 UITableViewDelegate 协议。然后打开 TreeVC.xib，在画布上选择 Table View 对象，打开 Connections Inspector，将 dataSource 属性和 delegate 属性连接到 File's Owner 上。

首先我们修改 NetworkModule 的 endPost:business 方法。在这个方法中，我们初始化树的

模型 orgTree 对象，然后刷新表视图：

```
orgTree=[OrgTreeNode instanceWithXMLElement:e];
[orgTree refreshShowNodes];
[table reloadData];
```

注意：orgTree 的初始化方法使用了工厂方法 instanceWithXMLElement:方法，原因在前面已说过。此外，在刷新表视图之前，需要计算一下树的可视节点：[orgTree refreshShowNodes];。否则，整棵树都不可见。

接下来在 TreeVC 中实现 UITableViewDataSource，有两种方法。

（1）numberOfRowsInSection:方法

我们的表视图不分组（只有一节），因此我们直接返回模型的可视节点数：

```
return orgTree.showNodes.count;
```

（2）cellForRowAtIndexPath:方法

这个方法中我们要做许多工作。在使用 TreeNodeCell 的工厂方法 getCell 获得一个 cell 之后，我们需要获取单元格的数据模型（即我们用 OrgTreeNode 节点作为单元格的数据模型）以便后续操作：

```
GDataXMLElement * element = [orgTree.showNodes objectAtIndex:indexPath.row];
OrgTreeNode *selSendNode = [[OrgTreeNode alloc]initWithXMLElement:element];
```

注意：orgTree 代表了整个组织机构树。然而 Table View 并不显示整棵树，它只展现树中的“可视”节点，因此它的数据模型实际上是 orgTree.showNodes 而不是 orgTree。

然后，我们用单元格模型（即 OrgTreeNode 节点）渲染单元格。首先，把节点的 element 属性传递给 cell 和 checkButton 的 tag 属性，因为在另外两个方法中我们还要利用到这个 element，放在 tag 中传递是一种很方便的做法。然后将节点的 name 在 UILabel 中显示出来，并根据节点的级别计算 cell 的缩进级别，把节点的 type 赋给 cell 的 cellType 属性：

```
cell.tag = (NSInteger)element;
cell.checkButton.tag = cell.tag;
cell.name.text = selSendNode.name;
cell.indentationLevel = selSendNode.level - 1;
cell.indentationWidth = 24;
cell.cellType = selSendNode.type;
```

根据节点的 type 属性和 expanded 属性，我们的 cell 要显示不同的图标。比如，对于人员节点要显示 person.png 图片，部门节点则显示 dept.png 图片，对于已经展开的节点要显示减号 minus.png 图片，折叠的节点则显示加号 plus.png 图片等：

```
if (cell.cellType==0) {
```

PDG

```
        cell.cellTypeImage.image = [UIImage imageNamed:@"person.png"];
    } else {
        cell.cellTypeImage.image = [UIImage imageNamed:@"orgnization.png"];
        if (selSendNode.hasChild ) {
            if (selSendNode.expanded) {
                cell.expansionImage.image = [UIImage imageNamed:@"minus.png"];
            } else {
                cell.expansionImage.image = [UIImage imageNamed:@"plus.png"];
            }
        } else {
            cell.expansionImage.image = nil;
        }
    }
```

同时，根据节点当前选择状态（check 属性），我们要在 checkButton 上显示不同图片。如未选中（check=0）显示 uncheck.png，半选（check=1）显示 halfcheck.png，全选显示 check.png 等：

```
    if (selSendNode.check == 0) {
            [cell.checkButton setImage:[UIImage imageNamed:@"unchecked.png"]
                             forState:UIControlStateNormal];
    } else if (selSendNode.check == 1) {
            [cell.checkButton setImage:[UIImage imageNamed:@"halfchecked.png"]
                             forState:UIControlStateNormal];
    }  else {
            [cell.checkButton setImage:[UIImage imageNamed:@"checked.png"]
                             forState:UIControlStateNormal];
    }
```

最后，我们让用户点击 checkButton 上时能产生一个动作：

```
    [cell.checkButton addTarget:self
                         action:@selector(checkAction:)
               forControlEvents:UIControlEventTouchUpInside];
```

checkAction:方法的目的很简单，改变当前节点的选中状态（未选、半选、全选），刷新整棵树，刷新表视图：

```
    - (IBAction)checkAction:(UIButton *)sender
    {
        OrgTreeNode *node = [[OrgTreeNode alloc]initWithXMLElement:
                             (GDataXMLElement *)sender.tag];
        int status = (node.check == 2 ? 0 : 2);
        [node checkParentAndChildren:node.element checked:status];
        [table reloadData];
    }
```

其中，checkParentAndChildren:checked:方法的调用通过向上和向下递归，刷新父节点和子节点（整棵树）的 check 状态。为什么一个节点的 check 属性变化要刷新整棵树的 check 属性？这是因为，一个节点的 check 状态改变，必然影响它的父节点和子节点的 check 状态。例如，一个节点由全选改变为未选，则它的所有父节点必然要改变为半选或未选状态，而它的所有子节点则要变为未选状态。这是一个相当麻烦的过程，除了向上、下级节点进行递归以外，我想不出什么好的办法。

好了，现在是 UITableViewDelegate 的实现了：

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
        *)indexPath
{
    [tableView deselectRowAtIndexPath:indexPath animated:NO];
    TreeNodeCell *cell = (TreeNodeCell *)[tableView cellForRowAtIndexPath:indexPath];
    OrgTreeNode *node = [[[OrgTreeNode alloc]initWithXMLElement:
                        (GDataXMLElement*)cell.tag]autorelease];
    if (node.type == 1 && node.hasChild) {// 如果是部门节点，且有子节点
        if (node.expanded) {
            node.expanded = NO;
        } else {
            node.expanded = YES;
        }
        [orgTree refreshShowNodes];
        [tableView reloadData];
    }
}
```

首先，deselectRoAtIndexPath:animated:方法的调用取消了当前单元格的选中状态。当用户点击某行单元格时，我们不需要显示一个加深的蓝色。

cellForRowAtIndexPath:方法通过 indexPath 参数获得当前单元格,然后再通过单元格的 tag 属性获得 GDataXMLElement 对象，并用这个对象实例化一个节点对象。接下来就是对节点的操作了——根据节点当前的 expanded 属性进行展开/折叠操作。当然我们排除了类型为 0（人员节点）以及 hasChild 为 NO 的节点，因为这些节点不能展开。refreshShowNodes 的调用有点奇怪，但很显然，如果你不这样做，可视节点的计算是不正确的，因为节点的展开和折叠直接影响了可视节点数目的变化——如果有节点被展开，必然有一些节点由不可视状态变为可视，如果节点被折叠，必然有一些节点由可视状态变为不可视。最后，刷新表视图。

9. 获取用户选择

我们的树是支持多选的，当用户勾选完毕，可以点击我们的导航栏“确定”按钮。这个时候由 okAction 方法负责获取用户选择：

```
-(void)okAction{
```

```
    if (orgTree) {
        for (OrgTreeNode* each in orgTree.checkedPeople) {
            NSLog(@"%@",each.name);
        }
    }
}
```

其实是调用了树的 checkedPeople 属性，在 checkedPeople 方法中我们过滤了部门节点，因为用户其实是想选择部门下面的人员。

好了，这个示例程序的代码就介绍完了。程序最终运行效果就如图 8-4、图 8-5 所示。

接下来，我们简单介绍一下 JSon 数据的解析。

8.5 Json 和 SBJson

Json(JavaScript Object Notation)是 HTTP 编程中另一种常用的轻量级数据交换格式。它是人类易于读写的纯文本，虽然最初是 JavaScript 编程语言规范中的一个子集，但其实它完全与语言无关，无论何种语言：C、C++、C#、Java、Perl、Python 还是其他，都能够轻易地使用它，并为众多的程序员们所熟悉。

SBJson 是由 Stig Brautaset 创建的面向 Objective C 的 Json 解析框架，它以极其简单的使用方式闻名。项目地址是：https://github.com/stig/json-framework/。下面介绍 SBJson 的使用。

8.5.1 在项目使用 SBJson

我们决定以源文件的方式使用，而不以框架的形式使用，这样更容易些。下载 SBJson 后，把 Classes 目录中所有.h 和.m 文件添加到项目中，然后在源文件中就可以使用#import "SBJson.h"了。

8.5.2 SBJson 使用示例

示例程序位于本书配套光盘“source/第 7 章/SBJsonDemo”目录。作为简单演示，我们直接在 main.m 中进行测试。我们在 testSBJson()中构建了一个 Json 格式字符串：

```
NSString *jsonString=@"[{\"id\":\"1\",\"name\":\"陈永和\",\"phone\":\" 13654789110
        \"}, {\"id\":\"2\",\"name\":\"王小乙\",\"phone\":\"13888789326\"}]";
```

然后调用 getArrayFromJSONString 函数进行解析：

```
NSLog(@"%@",getArrayFromJSONString(jsonString));
```

getArrayFromJSONString(NSString *string)函数只有一句代码：

```
return [string JSONValue];
```

NSString 并没有定义过 JSONValue 方法，这个方法是 SBJSon 通过类别（Catalogue）的方

式为 NSString 添加的。JSONValue 方法在 NSObject+SBJson.h 中定义。它是 NSString 的类别 NSString_SBJsonParsing 中的成员方法。关于类别的使用，在第 3 章中进行过讨论，它可以为已有的类（包括 Cocoa 框架中的类）添加新的行为（只是方法，不包括成员变量）。因此凡是 NSString 对象都可以调用 JSONValue 方法，以支持对 JSON 字符串的解析。

8.6 本章小结

本章介绍了 XML 和 Json 解析框架，尤其是前者，因为 XML 现在已成为最主流的网络数据交换格式。

XML 解析除了 Cocoa 的 NSXML 框架，还有 TBXML、libxml、GDataXML 等优秀的第三方框架和库，本章对这些框架进行了介绍。

在介绍 GDataXML 的过程中，作者演示了一个复杂树视图实现示例。详细介绍这个示例的具体实现过程，树的展现和操作是非常实用和常见的技术，读者在今后的企业应用开发项目必然会用到。

最后，作者简单介绍了 SBJson 对 Json 数据解析的支持。

第 9 章　保存用户数据

本章介绍如何保存用户（应用程序）数据，即 iOS 的持久化技术。持久化实际上就是把应用程序的数据以某种形式保存在可持续的介质上，比如文件。这样在应用程序退出后，数据不会被丢失，程序可以恢复原有的运行状态。

文件持久化的技术有许多种，由于“沙盒规则”的限制，文件持久化被限制在应用程序单独的一组文件夹中，比如 Documents 文件夹和 tmp 文件夹。在本章“文件的持久化”一节中，我们首先介绍了如何用 plist 文件保存简单数据，接着介绍了 NSUserDefaults 的使用。然后是 Cocoa 的基于 NSCoding 协议和 NSCoder 类的归档技术 。最后还介绍了一个自定义的归档/反归档类。读者可以自行进行扩展。

除了文件外，持久化的另一种常用技术就是数据库。在 iPhone 开发中，数据库技术不那么常用。然而作为企业应用中的一个部分，客户端访问数据库的技术仍然是必不可少的。因为对于某些数据（比如关系型数据），使用数据库仍然是比文件系统要好得多的解决方案。iPhone 支持对一种本地数据库的访问。这个数据库即轻量级数据库 SQLite。在本章“数据库”一节中，首先对 SQLite 数据进行了简单介绍。

Cocoa 框架对 SQLite 数据库的访问进行了优化，并提供了统一的、简化的编程接口 Core Data。Core Data 完全屏蔽了程序员对数据库的访问，使程序员以一种 OO（面向对象）的方式操作 SQLite 数据库。本章还介绍了 Core Data。

尽管如此，我们有时候还是要直接访问 SQLite，比如对某些复杂的数据模型，ORM（对象关系映射）的代价实在是过于昂贵，我们不得不放弃 Core Data。这时候，可以通过一些底层的编程接口，比如本地 C 语言接口直接访问 SQLite。当然，为了绕开 Cocoa 的 Core Data，同时避免底层接口的复杂性，可以使用封装良好的第 3 方框架。最后介绍了一个第 3 方开源框架 PLDatabase，它是一个面向 Objective-C 程序员的数据库编程接口，将 SQLite C 语言库进行了 Objective-C 封装，既保留了 Objective-C 语言的简单和高效，同时也保留了 SQLite C 语言原生库的强大功能。程序员可以直接在 PLDatabase 中利用 SQLite 对 SQL92 标准的支持，通过 SQL 语句更加灵活地操作底层数据。

9.1　文件的持久化

应用程序状态（数据）可以保存到沙盒中。Cocoa 框架提供了高级别的 API 来进行文件的持久化，而不需要程序员和底层文件 I/O 打交道。

9.1.1　保存到 plist 文件

在第 4 章，我们讨论了 iOS“安全沙盒”机制。根据沙盒规则，每个应用程序都拥有独

PDG

立的 Library、Documents 和 tmp 文件夹，应用程序对文件系统的访问被限制在这些最基本的文件夹内。随后，我们学习了如何访问这些文件夹。接下来，我们将学习如何将应用程序数据保存到这些文件夹中，包括从最简单的 plist 文件到嵌入式数据库 SQLite3。

plist 文件是一种常见的应用程序设置和首选项的保存策略。它的使用非常方便，在 Xcode 中可以直接打开和编辑 plist 文件。可以用 plist 文件存储任意文本、数值，甚至是序列化对象。序列化对象是能够转化为字节流的 Objective-C 对象，包括：

- NSArray 和 NSMutableArray
- NSDictionary 和 NSMutableDictionary
- NSData 和 NSMutableData
- NSString 和 NSMutableString
- NSNumber
- NSDate

在 plist 文件中保存序列化对象，需要先把上述 Objective-C 对象放到集合类（数组或字典）中，然后调用该集合类的 writeToFile:atomically:方法：

```
[arrayObj writeToFile:@"/xxx/xxx.plist" atomically:YES];
```

atomically 参数指定是否使用原子操作。当指定为 YES 时，数据会首先写入临时文件，当写入成功后才复制到第 1 个参数指定的文件。这样，当应用程序在保存数据时发生崩溃，保证原有 plist 文件的完整性。

当需要读取 plist 文件中的序列化对象时，可以使用集合对象的 initWithContentsOfFile:方法：

```
NSArray* array=[[NSArray alloc] initWithContentsOfFile:filepath];
```

提示：除了 NSArray/NSMutableArray，我们也可以在 NSDictionary /NSMutableDictionary 中使用类似的技术。

---

plist 文件的使用是如此简单，以至于我并没有为它编写示例程序，这个工作将留给读者自己去完成。

注意：在使用 plist 持久化应用程序数据时，有两个地方需要注意。1）要序列化的对象必须先放到集合中，才能序列化。2）序列化对象必须是上述列表中指定的 Objective-C 类，简单类型、结构、自定义类和其他不在列表中的 Objective-C 类不能序列化到 plist 文件中。

9.1.2 NSUserDefaults

NSUserDefaults 是 Cocoa 提供的默认应用程序状态（用户数据）保持接口。它提供了简化的 plist 文件持久化方法。通过 NSUserDefaults 类，你可以把用户首选项保存到 plist 文件中。当应用程序结束，这些数据仍然存在，并可在应用程序启动时，再次把上次运行的状态显示在应用程序中。

提示：NSUserDefaults 把 plist 文件存储在沙盒中的 Library/Preferences 目录下。这个 plist 文件一直存在，直到你删除应用程序。

---

1. 获取 User Defaults

要想获取一个 NSUerDefaults 实例，最简单的方法是获取一个应用程序共享实例（单例）：

```
+ (NSUserDefaults *)standardUserDefaults
```

如果不想使用共享的 User Defaults 对象，可以在 alloc 之后实例化自己的 NSUserDefaults：

```
- (id)init
-initWithUser: (NSString *)username
```

前者使用当前用户账号实例化 NSUserDefaults，后者可以用某个指定的用户账号实例化 NSUserDefaults（这个方法在 iOS 中无效，因为它需要用户以 superuser 来运行应用程序）。

2. 在 User Defaults 中保存值

在 Defaults 中保存值，需要调用 setX:forKey:方法。Defaults 实际上仍然是一个 plist 文件，每个值都有一个唯一的 key，对值的存取是根据 key 来进行的。setX 中的 X 是值的类型，如 BOOL、Float、Integer 等。总共有 6 个 setX:forKey 方法：

```
- setBool:forKey:
- setFloat:forKey:
- setInteger:forKey:
- setObject:forKey:
- setDouble:forKey:
- setURL:forKey:
```

3. 删除 User Defaults 值

删除值使用如下命令：

```
- removeObjectForKey:
```

4. 读取 User Defaults 值

读取存放在 User Defaults 对象中的值，使用 xForKey 方法，xForKey 方法中的 x 是值的类型：

```
- arrayForKey:
- boolForKey:
- dataForKey:
- dictionaryForKey:
- floatForKey:
- integerForKey:
- objectForKey:
- stringArrayForKey:
- stringForKey:
- doubleForKey:
- URLForKey:
```

5. User Defaults 使用示例

示例项目位于光盘“source/第 9 章/testDefaults”目录。在 Xcode 中打开示例项目，点击 Build and Run 按钮，程序将显示一个 TextField 和一个按钮，如图 9-1 所示。

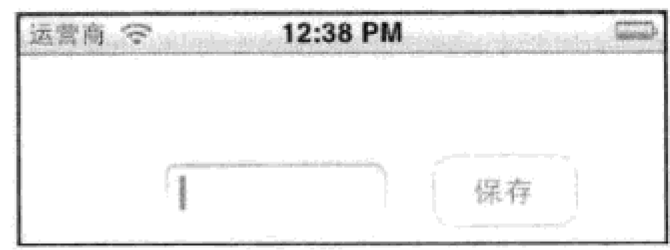


图 9-1 testDefaults 运行界面

当你在 TextField 中输入一些字符串，点击“保存”按钮，该文本字符串将被保存到 NSUserDefaults 中，然后程序会自动退出。

再次点击 Build and Run 按钮，程序会显示你上次运行程序时输入的文本内容，如图 9-2 所示。

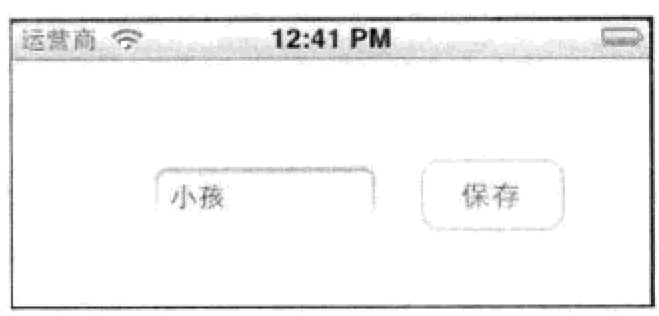


图 9-2 再次运行 testDefaults

程序是在 buttonClick 方法中保存 NSUserDefaults 的：

```
-(IBAction)buttonClick{
    [tf resignFirstResponder]; // ❶
    NSUserDefaults* defaults=[NSUserDefaults standardUserDefaults];// ❷
    [defaults setObject:tf.text forKey:@"words"]; // ❸
    [defaults synchronize]; // ❹
    exit(0); // ❺
}
```

代码说明：

❶ 释放键盘。

❷ 通过 standardUserDefaults 获取共享的 NSUserDefaults 对象。

❸ 将用户输入的文本保存到 NSUserDefault 对象。

❹ synchronize 方法手动将 NSUserDefaults 对象的改变保存到 defaults 数据库。

❺ 退出程序。

在 viewDidLoad 方法中，我们将 defaults 数据库中的数据取出：

```
- (void)viewDidLoad
```

```
{
    NSUserDefaults* defaults=[NSUserDefaults standardUserDefaults];  // 通过
                            standardUserDefaults 获取共享的 NSUserDefaults 对象。
    NSString *words=[defaults stringForKey:@"words"];// 从 NSUserDefaults 中读取 key
                            为 words 的对象，并赋给 NSString。
    tf.text=words;  // 用 NSString 设置 TextField 的 text 属性。
    [super viewDidLoad];
}
```

9.1.3 归档

归档是 Cocoa 中的另一种对象序列化（编码/反编码）技术。正如前面所述，plist 只能序列化种类有限的对象，简单类型和自定义对象无法序列化到 plist 文件中。而使用归档技术则可以将任何复杂对象写入文件，只需要归档对象实现了 NSCoding 协议。

让我们先来看看 NSCoding 协议。NSCoding 协议在 NSObject.h 头文件中说明，包括了两个方法：encodeWithCoder:方法（用于编码）和 initWithCoder:方法（用于反编码），二者都使用了 1 个 NSCoder 对象作为参数。这里 NSCoder 是关键。实际上，Cocoa 归档技术正是基于 NSCoder。

encodeWithCoder:方法用于把基本类型和对象进行编码。对于对象，我们使用 NSCoder 的 encodeObject:forKey 方法进行编码，而对于基本类型（如 int 和 float）使用 encodeInt:forKey 或 encodeFloat:forKey 方法。如果要使对象支持归档，必须在 encodeWithCoder:协议方法中把每个实例变量用合适的编码方法进行编码。例如：

```
-(void)encodeWithCoder:(NSCoder*)encoder{
[encoder encodeObject:aObj forKey:@"objectProperty"];
[encoder encodeInt:aInteger forKey:@"integerProperty"];
}
```

提示：如果父类也支持 NSCoding 协议，则须在方法最后一行加上：[super encodeWithCoder:encoder];代码。

---

initWithCoder:方法用于还原已编码的归档对象。对应地，它有一组用于还原实例变量的解码方法，decodeObjectForKey:、decodeIntForKey:和 decodeFloatForKey。在 initWithCoder 方法中，需要把每个实例变量用适当的解码方法进行解码：

```
-(id)initWithCoder:(NSCoder*)decoder{
    if(self=[super init]){
        self.aObj=[decoder decodeObjectForKey:@"objectProperty"];
        self.aInteger=[decoder decodeIntForkey:@"integerProperty"];
    }
    return self;
}
```

如果父类也支持 NSCoding 协议，确保调用的是父类的 initWithCoder:方法而不是 init:方法：

```
if(self=[super initWithCoder:decoder]){
...
```

NSCoder 是一个抽象类，定义了一堆的 encode/decode 方法让子类去实现。同时，在 Foundation 中也提供了几个具体的子类实现：NSArchiver/NSUnarchiver、NSKeyedArchiver/NSKeyedUnarchiver、以及 NSPortCoder。我们将介绍 NSKeyedArchiver 和 NSKeyedUnarchiver。

1. NSKeyedArchiver 和 NSKeyedUnarchiver

Cocoa 提供了 NSCoder 子类——NSKeyedArchiver 和 NSKeyedUnarchiver 类。这两个类分别实现了 NSCoder 中定义的部分方法。如果类或对象已经实现了 NSCoding 协议，我们可以使用 NSKeyedArchiver 和 NSKeyedUnarchiver 来进行对象的归档 / 反归档操作。

NSKeyedArchiver 实现了 NSCoding 的编码方法。归档操作可以通过 NSKeyedArchiver 把对象序列化到指定文件中：

```
[NSKeyedArchiver archiveRootObject: dictionary toFile:path];
```

dictionary 是一个 NSDictionary 对象，Cocoa 提供的集合类（数组和集合）都已经实现了 NSCoding 协议。path 是指定的文件保存路径。

NSKeyedUnarchiver 实现了 NSObject 的解码方法。反归档操作则通过 NSKeyedUnarchiver 把文件还原为一个实例对象：

```
NSDictionary *dictionary=[NSKeyedUnarchiver unarchiveObjectWithFile: path];
```

可以看到，使用 NSKeyedArchiver 和 NSKeyedUnarchiver 进行对象的归档和反归档非常方便，可以直接调用其静态方法而不需要经过实例化。

下面我将使用 NSKeyedArchiver 和 NSKeyedUnarchiver 类提供的功能定制自己的归档和反归档类。

2. 使用自己的归档类

我们编写了一个支持归档和反归档操作的 DictionaryArchiver 类，用于应用程序一般性数据的保存。该类的定义非常简单，只包含两个支持归档和反归档的静态方法：

```
#import <Foundation/Foundation.h>
@interface DictionaryArchiver : NSObject {
}
+(void)save:(NSDictionary*)dic filename:(NSString*)s;
+(NSDictionary*)load:(NSString*)filename;
@end
```

首先看 save 方法的实现。我们先获取了应用程序沙盒中的 Documents 目录，将参数指定的文件名附加到 Documents 目录后面组成目标文件。我们不得不这样做，因为“沙盒规则”限制我们的应用程序只能访问有限的文件目录。

```
NSString *prefix = [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUser-
```

PDG

```
        DomainMask, YES) objectAtIndex:0];
NSString* path = [prefix stringByAppendingPathComponent:s];
```

最后使用 NSKeyedArchiver 的 archiveRootObject:toFile:方法，把参数指定的 NSDictionary 保存到指定文件中。

```
[NSKeyedArchiver archiveRootObject: dic toFile:path];
```

NSDictionary 是一个实现了 NSCoding 的协议的对象，因此允许我们这样做。而在 NSDictionary 中，可以很方便地存放各种类型的数据——Foundation 框架的许多类都实现了 NSCoding 协议，但不包括你自定义的类。如果你要使用自定义的类，请确保自己已经实现了 NSCoding 协议。

Load 方法同样简单。首先我们也是把文件定位到沙盒的 Documents 目录，然后用 NSFileManager 检查文件是否存在，若存在则使用 NSKeyedUnarchiver 类的 unarchiveObjectWithFile 方法加载文件内容并反归档为一个 Dictionary 对象：

```
NSFileManager* fm=[NSFileManager defaultManager];
if ([fm fileExistsAtPath:path]) {//若文件存在
    dic=[[NSKeyedUnarchiver unarchiveObjectWithFile: path]retain];
}
return dic;
```

这样，我们在应用程序中只需将要保存的数据放入字典中，然后调用 DictionaryArchiver 的 save:filename:方法，即可将数据写入到指定文件里。要从文件恢复数据时，则调用该类的 load:方法。

提示：如果你需要自己子类化 NSCoder，可以参考苹果 Cocoa 参考“归档和序列化编程指南”中关于“子类化 NSCoder”的部分。

9.2 数据库

在文件中保持关系型数据是困难的。对于关系型数据，往往需要存储在关系型数据库中，比如用户关系数据、产品名录等。接下来我们就介绍另一种持久化方案—— iOS 对数据库的支持。

9.2.1 嵌入式数据库 SQLite3

iPhone 支持对数据库的访问。在你下载的 iPhone SDK 中，其实已经内置了一个数据库，即嵌入式数据库 SQLite。关于它的详细信息，可以访问它的主页：http://www.sqlite.com。

SQLite 是关系数据库管理系统，包含在一个轻量级的 C 语言库中，是一个开源项目。SQLite 支持大多数的 SQL-92 标准，包括事务，即原子性、一致性、隔离性和持久性的（ACID），还支持触发器和多数的复杂查询。这意味着你曾经非常熟悉的 SQL 语言又可以在 SQLite 中派上

用场了。本节将讨论如何在 iPhone 上访问 SQLite 数据库。

注意：有一点是很重要的，SQL 是无类型的，即不会进行类型检查，你可以把一个字符串插入到整型列中。此外，创建数据库在 SQLite 中不会是一件简单的事情，即不得不再次和命令行打交道了。最后一件不幸的事情是，SQLite 没有面向对象的接口，它只有基于 C 语言的 API，你不得不和讨厌的 C 函数库打交道，后面我们将介绍一个第三方的 API 框架 PLDatabase。

---

1. 创建 SQLite 数据库

创建数据库需要使用命令行。打开终端程序，键入如下命令，将在桌面上建立一个名为 animal 数据库：

```
cd ~/Desktop
sqlite3 animal
```

这时命令提示符变为了“sqlite>”，这表明你已经进入了 SQLite 命令模式，你可以在 SQLite 命令模式中输入 SQLite 命令或者 SQL 语句，如图 9-3 所示。

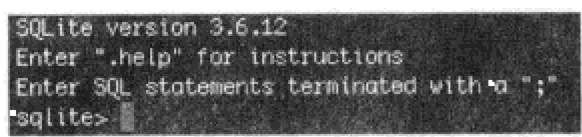


图 9-3 SQLite 命令提示模式

2. 创建表

清理你的桌面，让它们看起来整洁一些。你发现了什么？为什么没有在桌面上找到数据库文件？

别急，数据库文件需要创建一张表之后才会生成，它不会创建一个没有表的数据库文件，创建表的语句如下：

```
create table animal(id primary key,name);
```

注意句末的分号是必须的，因为这是一个 SQL 语句。而且，在这个表创建语句中，刻意省略了字段的类型定义，因为我们说过，SQLite 是无类型的，既然如此，为列指定类型是没有必要的。

3. SQLite Manager

也许你以为从此只能用命令行的方式建表、建库、插入记录、管理数据了。但是我们接下来要介绍一个好用的工具，使我们脱离讨厌的字符界面。这个工具就是 SQLite Manager，当前的最新版本是 0.7.6。它是一个 FireFox 插件，可以很方便地安装，前提是你已经装有 FireFox 浏览器。点击 FireFox 工具菜单中的 SQLite Manager 菜单，你可以看到如图 9-4 所示的界面。在这里你可以干一切在命令行中能干的事情，包括建库、建表、增删改查等。

点击工具栏上 Connect Database 按钮，可以打开刚才我们创建的数据库 animal（位于桌面的 animal 文件）。然后你可以为这个数据库添加表、记录，以及进行操作。具体 SQLite Manager

的使用就不用多说了，它的功能都显示在工具栏或窗口按钮里。

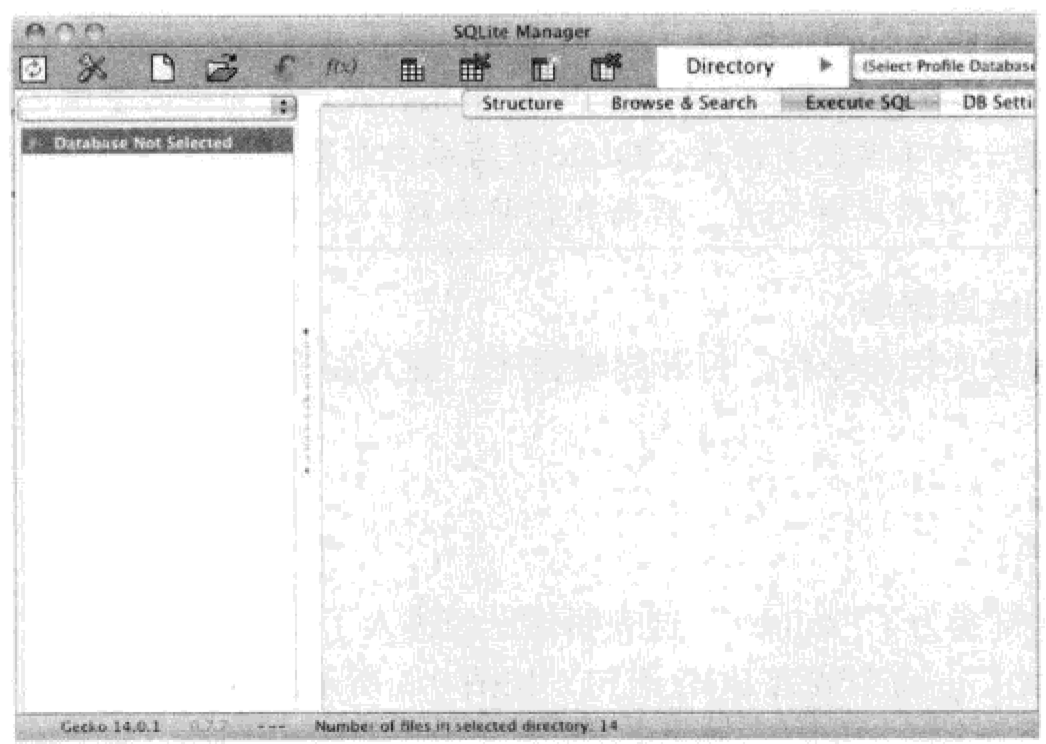


图 9-4 SQLite Manager

9.2.2 使用 Core Data

Core Data 是苹果提供的可视化模型数据设计工具，它屏蔽了数据库细节和难看的 SQL 语句，让你可以用操作 Objective-C 对象的方式操作数据库对象，但数据最终是被放到了一个 SQLite 数据库文件里。你可以在 Core Data 应用程序的 Document 目录中找到这个文件。

通过 CoreData.framework 提供的 API，极大地简化了编程中的数据库访操作。Core Data 使用一种类似 ORM（对象关系映射）的技术，把关系数据库中的表和数据封装为对象和属性，表与表之间的关系封装为对象间的包含和被包含关系。在某种程度上，可以把它视作 Cocoa 框架中的 Hibernate。

苹果公司为了在 Touch 应用程序提供对 Core Data 的支持做了大量的工作。要在项目中使用 Core Data，需要在选择项目模板时做一些特殊的事情。

新建 Empty Application 项目时，有个“Use Core Data”选项（如图 9-5 所示），选中该选项，便可在项目中使用 Core Data。

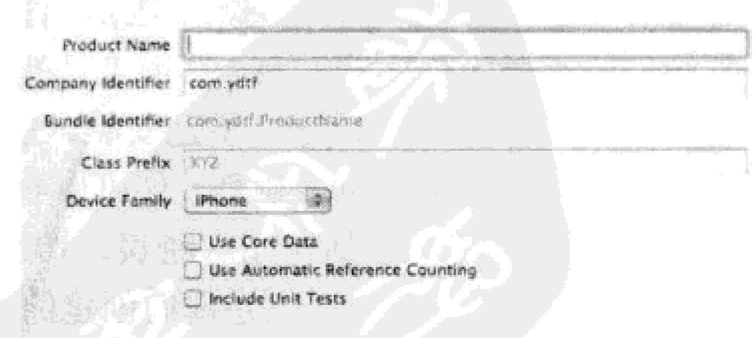


图 9-5 要使用 Core Data，创建项目时勾选 Use Core Data

打开 CoreDataDemo 项目（位于光盘“source/第 9 章/CoreDataDemo”目录），在 Project Navigator 窗口，Resources 目录下出现了一个.xcdatamodel 文件。这是我们的 Core Data 数据模型文件。这使我们以图形化的方式设计数据库（表和字段）。点击该文件，将显示数据模型编辑器（如图 9-6 所示）。

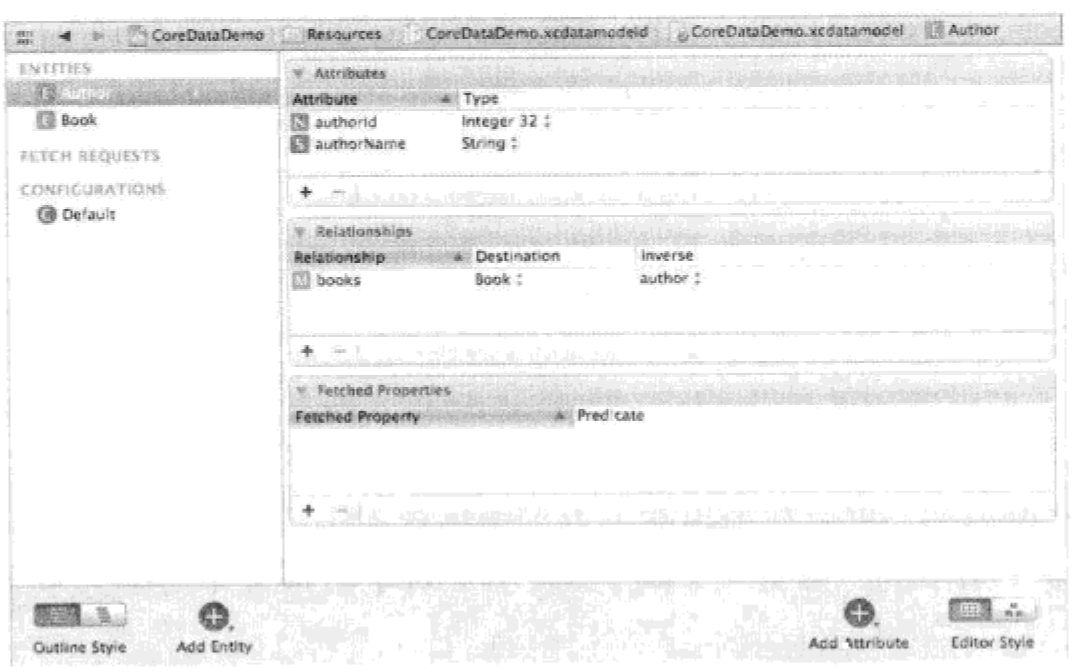


图 9-6　数据模型编辑器

在数据模型编辑器中，左栏是 Entity（实体）窗口，实体对应于数据库表的概念；右栏是 Attributes（属性）窗口和 Relationship（关系）窗口。属性对应的是数据库中字段的概念；关系则对应数据库中外键的概念。右下角有一个 Editor Style 的两个按钮，分别对应表视图和图形视图，当点击图形视图时，数据模型将以图形的形式显示。

点击 Entity 窗口下方的+号按钮，增加一个 Author 实体和一个 Book 实体。然后分别使用 Attributes 窗口设置它们的属性如下（见图 9-7 和图 9-8）。

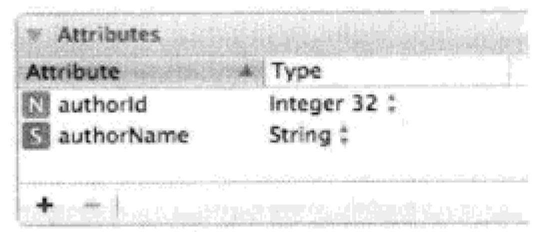


图 9-7　Author 实体的 2 个属性

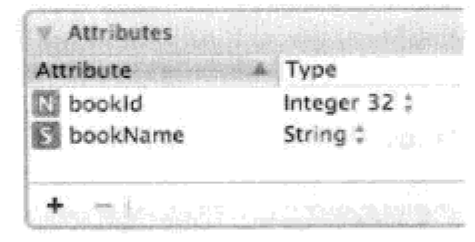


图 9-8　Book 实体的 2 个属性

Author 和 Book 之间是 1 对多的关系。一个 Author 可以拥有多个 Book 对象。因此我们需要为 Author 和 Book 建立一个 1 对多关系。

在 Entity 窗口选择 Book，点击 Relationships 窗口下的“+” 号按钮，创建一个 item，名称输入 author，Destination 选择 Author 表。同样，为 Author 表也创建一个 item，名为 books，Destination 选择 Book 表。

然后将 author 的 Inverse 设置为 books，books 的 Inverse 设置为 author。这样两个表就建立起关系来了。

选择 books，然后在 Data Modal Inspector 窗口中勾选 To-Many Relationship，如图 9-9 所示。

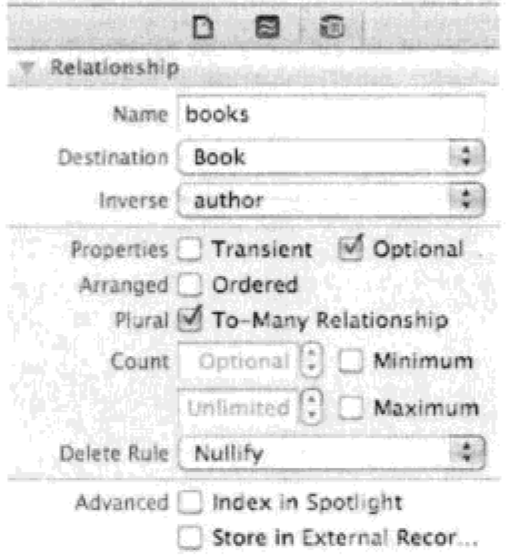


图 9-9 在 Data Modal Inspector 中修改关系属性

数据模型建立好以后，可以在编辑器下方的图形视图中进行查看，如图 9-10 所示。其中，双箭头表示 1 对多关系中的“多方”，单箭头表示 1 对 1 关系或 1 对多关系中的“1 方”。

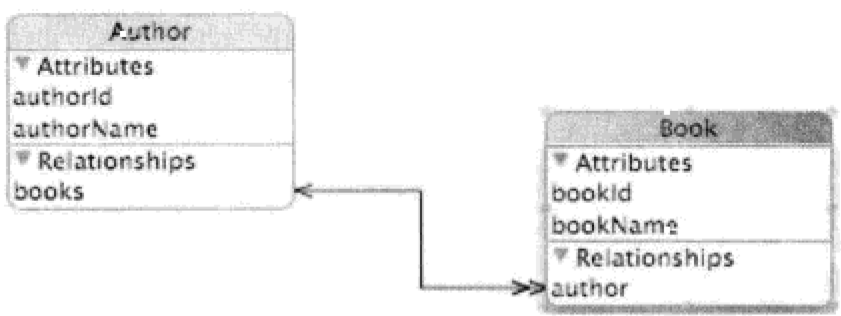


图 9-10 数据模型图形视图

接下来，需要根据设计好的数据模型生成 Managed Object Class。

选择 File→New File，选择其中的 Core Data 下的 NSManagedObject subClass 模板，如图 9-11 所示。

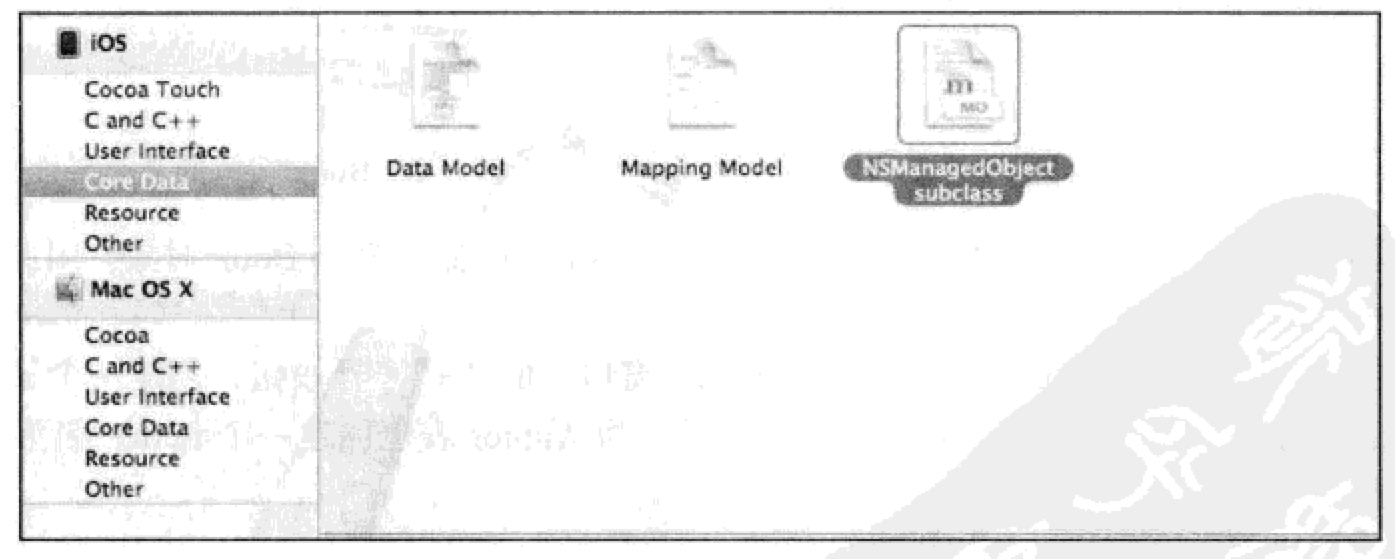


图 9-11 新建 Managed Object Class

点击 Next，再点击 Next，在选择要创建的实体类时，勾上 Book 和 Author，如图 9-12 所示。

点击 Finish，对应两个类的 4 个文件将产生，你可以查看源文件中的代码，了解 Cocoa 是如何进行 ORM（对象关系映射）的。接下来我们来了解如何使用这些 Managed Object 对象进行数据库操作。

图 9-12　选择产生类文件的实体

打开 CoreDataDemoAppDelegate.h 文件，我们发现勾选 Use Core Data for storage 选项之后，增加了如下变量、属性和方法声明：

```
NSManagedObjectContext *managedObjectContext_;
NSManagedObjectModel *managedObjectModel_;
NSPersistentStoreCoordinator *persistentStoreCoordinator_;
...
@property (nonatomic, retain, readonly) NSManagedObjectContext *managedObjectContext;
@property (nonatomic, retain, readonly) NSManagedObjectModel *managedObjectModel;
@property (nonatomic, retain, readonly) NSPersistentStoreCoordinator *persistent
        StoreCoordinator;
...
- (void)saveContext;
```

这些属性和方法在操作 Managed Object 对象时将会用到，而它们已经由 Xcode 自动为我们声明了。

为了简单演示 Core Data 的增、删、改、查操作，我们对 MainWindow.xib 进行了一点简单的修改，在窗体上加入了 4 个按钮控件，并将其连接到应用程序委托的 4 个 IBAction 方法上，用于执行增删改查 4 个操作。

其中，增加记录的方法实现如下：

```
-(IBAction)addObjects{
    Author *author=[NSEntityDescription insertNewObjectForEntityForName:@"Author"
        inManagedObjectContext:self.managedObjectContext];
    author.authorId=[NSNumber numberWithInt:1];
    author.authorName=@"张海藩";
    Book *book1=[NSEntityDescription insertNewObjectForEntityForName:@"Book"
        inManagedObjectContext:self.managedObjectContext];
    book1.bookId=[NSNumber numberWithInt:1];
    book1.bookName=@"软件项目导论";
    Book *book2=[NSEntityDescription insertNewObjectForEntityForName:@"Book"
        inManagedObjectContext:self.managedObjectContext];
    book2.bookId=[NSNumber numberWithInt:2];
```

```
        book2.bookName=@"面向对象程序设计实用教程";
        book1.author=author;
        book2.author=author;
        NSError* err=nil;
        if (![self.managedObjectContext save:&err]) {
            NSLog(@"Error: %@",[err localizedDescription]);
        }else {
            NSLog(@"add objects success!");
        }
    }
```

可以看到，在 Core Data 中进行插入记录操作的代码非常直观，你不会感到丝毫陌生，因为它完全屏蔽了 SQL 语句的书写，而使用一种面向对象的操作方法。其他的操作，如删除、修改和查询，你可以参考另外 3 个方法的代码。

9.2.3 使用 PLDatabase 访问数据库

PLDatabase 是一个开源项目，它对 SQLite C 语言库进行了 Objective-C 的封装，便于 Mac OS X 和 iPhone 开发人员使用。它比原生的 SQLite 库更加简单，使用者众多，因此我们直接略过 SQLite C 函数库的主题，直接介绍 PLDatabase 框架的使用。

PLDatabase 项目地址位于：http://pldatabase.googlecode.com/，你可以在这里下载它的最新版本（当前最新版本为 1.2.1）。它以两种方式提供：二进制（PlausibleDatabase.framework）和源代码。

为了避免编译过程中出现一些莫名其妙的错误，我们下面将使用源代码的 PLDatabase。

打开终端，输入命令：

```
svn checkout http://pldatabase.googlecode.com/svn/trunk/pldatabase
```

为了演示，我们创建了一个示例项目，位于光盘“source/第 8 章/ PLDatabaseDemo”目录。

checkout 后，最新版本的源代码就会放到/pldatabase 目录下，将/classes 目录下所有.h 和.m 文件(并包括以 tests.m 结尾的文件)拖到项目中（选择“Copy Item”选项）。

在 PLDatabaseDemo 项目的 Resources 组下，我们放了一个示例数据库 Animals.db，你可以用 SQLite Manager 打开它（如果不能打开，请将文件扩展名改为.sqlite），它包含了一张 animals 表，其中的数据如图 9-13 所示。

id	name	description	image
1	elephant	a elephant	http://dblog.com.
2	monkey	a monkey	http://dblog.com.

图 9-13 animals 表

PLDatabase 框架需要 SQLite 库的支持，在项目中加入 libsqlite3.0.dylib 库。

新建一个实体类 Animal，代码很简单，用于将表 animals 的数据模型映射为 Objective-C 对象的形式：

```
#import <Foundation/Foundation.h>
@interface Animal : NSObject {
    NSString *name;
    NSString *description;
    NSString *imageURL;
}
@property (nonatomic, retain) NSString *name;
@property (nonatomic, retain) NSString *description;
@property (nonatomic, retain) NSString *imageURL;
-(id)initWithName:(NSString *)n description:(NSString *)d url:(NSString *)u;
@end
#import "Animal.h"
@implementation Animal
@synthesize name, description, imageURL;
-(id)initWithName:(NSString *)n description:(NSString *)d url:(NSString *)u {
    self.name = n;
    self.description = d;
    self.imageURL = u;
    return self;
}
@end
```

注意：Animal 类的成员属性和 animals 表的字段是一一对应的。因此可以用一个 Animal 实例来代表一个 animals 记录（数据行），这样将极大地方便存取记录字段。同时，我们提供了一个便利的初始化方法 initWithName:description:url:，能够很方便地用该初始化方法构建出一个新 Animal 对象。

新建一个类 DB，用 PLDatabase API 提供的功能操作数据库，打开 SQLite 数据库、检索表中记录以及关闭数据库，DB 类的 interface 如下所示：

```
#import <Foundation/Foundation.h>
#import "PlausibleDatabase.h"
#define _DBFILENAME @"Animal.db" //常量:数据库文件名
static PLSqliteDatabase *db;
@interface DB : NSObject {
}
+(PLSqliteDatabase *)open;// 连接数据库，取得 sqlitedatabase 引用
+(void)close;//关闭数据库
+(NSArray *)getAnimals:(NSString *)sql ;//从 animals 表中获得结果集
@end
```

DB 类有 3 个类方法。open 方法用于打开数据库文件 Animal.db；close 方法则用于关闭数据库文件。getAnimals 用于执行参数指定的查询语句，并查询的结果集返回为一个数组对象。

首先看 open 方法实现。查看成员变量 db（一个 PLSqliteDatabase 对象），如果 db 已经被初始化，直接返回 db；否则要打开数据库文件并用 PLSQLiteDatabase 的 initWithPath 方法初始

化 db 对象。通过这种方式实现了 DB 类的单例模式，即 DB 类在应用程序生命周期中始终只有一个实例，数据库文件只会被打开一次：

```
if (db) {// 若 db 已经存在，直接返回 db
    return db;
}
```

接下来要获取应用程序 Documents 文件夹，将数据库文件夹定位到 Documents 目录下：

```
NSString *documentPath = [NSSearchPathForDirectoriesInDomains(NSDocument
        Directory, NSUserDomainMask, YES) objectAtIndex:0];
NSString *path = [documentPath stringByAppendingPathComponent:_DBFILENAME];
```

当然，在应用程序束（我们应该明白“应用程序束”的概念）中有一个空的数据库文件（表中没有数据），当在 Documents 文件夹中找不到数据库文件，我们会将“应用程序束”中的空数据库文件拷贝到 Documents 目录下：

```
// 应用程序中也有一个"备份"的数据库文件，取得它的路径
NSString *srcPath = [[NSBundle mainBundle] pathForResource:@"Animal" ofType:
                    @"db"];
/ 取得文件管理器
NSFileManager **fileManager = [NSFileManager defaultManager];
f (![fileManager fileExistsAtPath:path]) {//若用户文件系统中不存在数据库文件
    NSError *error;
    if (![fileManager copyItemAtPath:srcPath toPath:path error:&error]) {
        //把备份的数据库复制一份给用户
        NSLog(@"When setup db: %@", [error localizedDescription]);
    }
}
```

然后，我们使用 PLSqliteDatabase 的初始化方法 initWithPath：构建一个 PLSqliteDatabase 对象，并将这个临时对象赋给 db。然后用 open 方法打开数据库文件（我们已经保证了它的存在），并返回 db 对象：

```
// 以指定路径实例化 PLSqliteDatabase
db = [[PLSqliteDatabase alloc] initWithPath: path];
// 打开数据库
BOOL re = [db open];
NSLog(@"db open ok?: %d", re);
return db;
```

close 方法简单地调用 PLSqliteDatabase 对象的 close 方法来关闭数据库文件：

```
+ (void) close {
    if (db) {// 若 db 不为空，进行关闭
        [db close];
        db = NULL;
    }
}
```

getAnimals 方法用于执行任意的 SQL 查询语句。首先它调用 DB 类的 open 方法打开数据库：

```
PLSqliteDatabase* db=[DB open];// 打开数据库
```

然后声明一个 id<PLResultSet>对象，这是一个 PLDatabase 框架中定义的结果集对象，专门用于保存 SQL 查询语句返回的结果集。

```
id<PLResultSet> rs;// 声明 Resultset
```

然后用打开的 PLSqliteDatabase 对象执行 executeQuery 方法，以执行 SQL 查询语句：

```
rs=[db executeQuery:sql];// 执行 SQL 语句
```

在检索结果集之前，声明一个 NSMutableArray，用于返回结果：

```
NSMutableArray *array=[[NSMutableArray alloc]init];
```

开始遍历 id<PLResultSet>对象。代码与 Java 的 JDBC 代码有惊人的类似。首先，针对结果集的每一条记录，用 xxxForColumnIndex:或 xxxForColumn:方法读取每个字段并将字段值封装到一个 Animal 对象的属性中。xxxForColumnIndex:使用字段的索引来检索字段值；xxxForColumn:则使用字段名的索引检索字段值。其中，xxx 代表字段值将以何种数据类型的格式返回，比如 stringForColumn:方法会返回字段值的字符串形式，而 intForColumn 则以 NSInteger 或 int 的形式返回。

```
while ([rs next]) {//遍历结果集
    //int   ID=[[rs objectForColumnIndex:0]intValue];
    NSString *name=[rs stringForColumnIndex:1];
    NSString *description=[rs objectForColumn:@"description"];
    NSString *img=[rs stringForColumnIndex:3];
    //NSLog(@"%@,%@,%@",name,description,img);
    // 将结果集中的行封装为对象
    Animal* a=[[Animal alloc]initWithName:name description:description url:
            img];
    [array addObject:a];//将封装好的对象加到数组
    [a release];
}
[rs close];
return [array autorelease];//返回数组
```

关于数据库的操作都封装到 DB 类中了，接下来将在 ViewController 中调用。打开 RootViewController，你会在 init 方法代码找到代码：

```
animals=[DB getAnimals:sql];
```

这样，表 animals 中的记录就会被检索出来并放到一个 Animal 对象的数组中。其余的代码会将这些数据显示到 UITableView 中，程序运行的效果如图 9-14 所示。

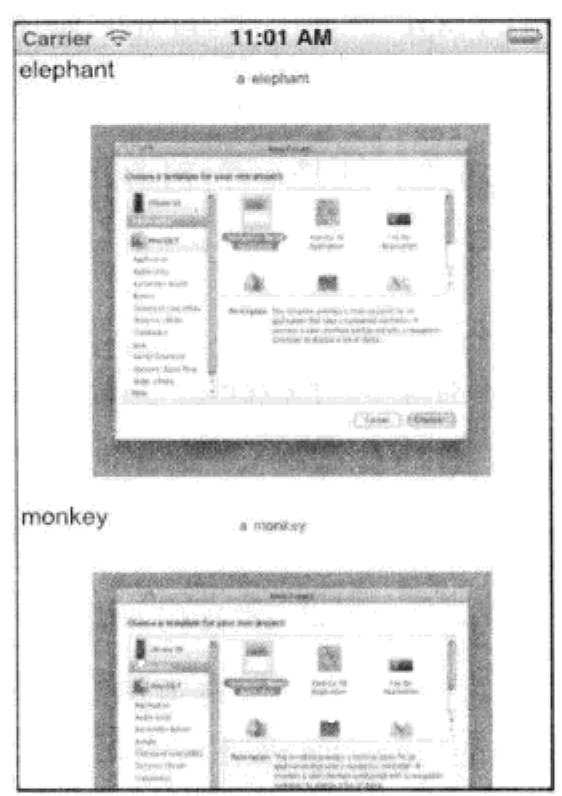


图 9-14 表 animals 中的数据显示到了 Table View 上

9.3 本章小结

持久化技术常用于保存程序运行的状态。本章介绍 iOS 开发中的几种持久化技术，如 plist 文件、NSUserDefaults、归档和数据库。

plist 文件实际上是以 XML 格式存储的，常用于保存用户选项、应用程序设置等简单数据，而且从一个 plist 文件中恢复数据也非常简单。这种技术的特点是使用简单，SDK 支持良好，不需要额外的编码。缺点是，只支持 Cocoa 指定的数据类型，不支持自定义类和简单类型。

NSUserDefaults 是 Cocoa 的应用程序设置储存方案。它提供了简单的跨应用程序会话的持久化。当应用程序重新启动，对 NSUserDefaults 所做的修改仍然存在。

归档技术是 Cocoa 提供的另一种持久化方案，Interface Builder 正是使用归档（nib 文件）来存储和加载用户界面。要支持 Cocoa 的归档，需要实现 NSCoding 协议，或者从 NSCoder 类继承。NSCoder 是一个抽象类，Cocoa 提供了一系列 NSCoder 的子类来支持归档和反归档操作，除了本章介绍的 NSKeyedArchiver\NSKeyedUnarchiver 类外，还包括：NSArchiver\NSUnarchiver 和 NSPortCoder。此外，我们也可以通过子类化的方式实现自己的归档/反归档类。

本书把数据库视作是对象持久化方案中的一种，并介绍了 iOS 内置的嵌入式数据库 SQLite 及其访问技术。SQLite 是一个小巧但功能完善的数据库，它支持大部分的 SQL92 标准。因此我们完全可以同访问其他 RDBS（关系型数据库）一样访问它。本章介绍了 Core Data 框架和 PLDatabase 框架两种访问技术。前者使用简单，完全面向对象；后者使用仍然简单，但可使用完整的 SQL 语句支持，使用上更加灵活。作者建议，在 iOS 企业应用中使用 PLDatabase 作为底层数据库访问技术。

第 10 章　安　　全

随着第 3 代数字通信技术的发展，手机网络的成熟度及信号质量获得了明显提升。这直接导致移动应用种类从单纯的个人应用、游戏娱乐逐渐向企业应用转移。这些应用涵盖了多媒体业务、交互式数据业务、电子商务、互联网服务等多种信息服务，手机信息安全的问题也就随之而来，比如个人密码和商业机密的泄漏。因此，在 iOS 企业应用开发中，为了保障系统的安全，常采用密钥机制对网络接入进行认证和鉴权，并使用各种加密算法对传输数据进行加密，以及确保数据完整性。本章主要讨论 iOS 客户端的安全机制。

首先介绍了 iOS 安全框架的相关概念，比如证书、密钥和信任服务。在苹果 iOS 和 Mac OS X 中，这些概念被集合到了钥匙串中。

在此基础上，我们介绍了企业应用中常见的 X.509 标准数字证书以及应用。通过 Cocoa 的 Security framework 框架，我们实现了对自签名 X.509 证书的验证，从而演示了如何应用 Security 框架来使用苹果的证书、密钥和信任服务。并总结了 Security 框架的不足之处。

Web 应用中常使用 SSL/TSL 协议保证网络数据的传输安全。我们在第 7 章中介绍了 ASIHTTPRequest 框架，该框架对底层网络协议进行了封装，包括 HTTP 和 HTTPs。

最后，我们介绍了两个比较常见的安全算法库 OpenSSL 和 CommonCrypto，并分别演示了如何利用这两个安全算法库实现了常见的 MD5 和 DES 加密。

10.1　iOS 安全框架简介

关于苹果的安全框架，在苹果的文档中有详细的描述，其中两篇是我们必须阅读的："Certificate，Key，and Trust Services Concepts" 和 "Certificate，Key，and Trust Services Programming Guide"。本节我们主要介绍第一篇文档的一些内容，关于苹果安全框架中的一些基本术语，如证书、密钥和信任的概念。

10.1.1　证书、密钥和信任服务

证书（或称为数字证书）是用于校验证书发行者或持有者身份的数据集合。在网络中，要想识别一个人的身份不是一件容易的事情，要想在证书中实现这点，证书中必须包含大量的信息：

- 证书是谁签发的？
- 证书是发给谁用的？
- 有效期（何时生效？何时过期？）
- 证书所用的公钥。
- 数字签名，用于表明证书签发者身份，并保证证书未被篡改。

PDG

- 扩展信息，包括额外的信息，比如也可以把私钥附在证书里面。

就像我们每个人的身份证，数字证书用于向它人证明我们的合法身份。签发者类似身份证上的发证机构（某某公安机关），持有人则是证件的使用者（身份证上的名字），有效期和身份证上的有效期意义相同，数字签名则类似发证机构的公章。公钥这东西身份证上是没有的，它是专门给其他人使用的，接收到证书的其他人，如果接收到证书持有人用私钥加密的数据，会使用这个公钥进行解密，这样就保证了证书持有人发出的数据不被泄密。

数字证书校验时，要使用另外一个证书来进行校验，这样证书与证书之间就形成了一个证书链，证书链的首端则是根证书。证书的签发者又称为证书机构（CA）。根证书是证书机构签发的，专用于标识证书机构的身份。

每一个公钥都有一个私钥与之配对，公钥能被其他人使用，私钥只能被所有者所使用。用私钥加密的数据，可以用配对的公钥进行解密。为了既能加密又能解密，用户必须既有公钥（证书中带有的）又有私钥。证书和与之关联的私钥合在一起，就代表了证书持有人的身份，称之为身份（identity）。

身份（即私钥+证书）和代码签名（Code Sign，见第1章）密切相关。Xcode代码签名使用iOS程序员的身份进行，仅仅有证书而没有与之关联的私钥是不能够形成有效身份的，并因此导致代码签名失败。

证书、密钥和信任服务就包含了这些功能：查找某个指定身份相关联的证书或密钥，以及按照给定条件查找某个身份。查找条件中包含了密钥授予的权限。

Mac OS X和iOS中，密钥和证书储存在钥匙串里。钥匙串是一个数据库，用于安全存储（加密的）私钥和非加密存储一些其他与安全相关的数据。证书、密钥和信任服务提供了在钥匙串中查找密钥、证书和身份的函数。在Mac OS X系统中，可以使用钥匙串访问工具去查看钥匙串中的内容，并读取证书内容。

提示：使用Mac OS X自带的钥匙串访问程序，可以查看和管理系统中所有的私钥、证书和身份。在证书一栏，我们可以很明显地看出有的证书下面会附带一个专有私钥，这就是有效的身份。

Mac OS X和iOS支持将密钥和证书存储在文件中。文件证书不仅仅用于存储数字证书，也用于存储密钥。文件证书可以用多种格式存储，比如X.509证书、PKCS7证书、PKCS12证书等。下面，我们将介绍X.509证书。

10.1.2 在iPhone中使用X.509证书

数字证书的格式一般采用X.509国际标准。目前，数字证书认证中心主要签发安全电子邮件证书、个人和企业身份证书、服务器证书以及代码签名证书等几种类型的证书。

数字证书由证书机构签发，证书机构通常需经权威认证机构注册认证。在企业应用中，也常用企业自身作为发证机构（未经过认证）签发数字证书，证书的使用范围也常是企业内部，这样的证书就是所谓的“自签名”的。

下面介绍我们在项目中常用的自签名证书在 iPhone 中的验证方法。作为简单演示，我们仅对证书的有效期进行验证。

假设证书文件为 dlt.cer，证书格式为 X.509 标准。可以用以下命令查看证书内容：

```
keytool -printcert -v -file ~/Desktop/dlt.cer
```

可以看到，证书的有效期是 2011 年 5 月 9 日到 2011 年 10 月 7 日。

以下我们演示如何验证证书的有效期。示例项目位于光盘“source/第 10 章/X509Demo”目录。打开 MyTrustService.m 文件，首先是它的初始化方法：

```
-(id)initWithFilename:(NSString*)filename EfficientDate:(NSDate*)date{
    if (self=[super init]) {
        file=filename;
        efficientDate=[date retain];
    }
    return self;
}
```

MyTrustService 类是我们自定义的数字证书操作类，使用了 iOS 的安全框架 API。它的构造参数中，除了第 1 个参数为证书文件路径外，还需要用一个有效的日期作为构造参数。这个有效日期是指证书有效期内的任何一个有效的日期都可以。因为苹果安全框架的限制，我们需要知道一个有效的日期在评估证书时作为参考。

接下来是它的证书校验方法：

```
-(ValuateResult)trustValuate:(NSDate*)date{
    ValuateResult ret;
    OSStatus err;
    NSData * certData;
    SecCertificateRef  cert;
    SecPolicyRef policy;
    SecTrustRef trust;

    assert(file != nil);
    assert(date != nil);
    certData = [NSData dataWithContentsOfFile:file];// ❶
    assert(certData != nil);
    cert = SecCertificateCreateWithData(NULL, (CFDataRef) certData);// ❷
    assert(cert != NULL);
    policy = SecPolicyCreateBasicX509();//❸
    assert(policy != NULL);
    err = SecTrustCreateWithCertificates(cert, policy, &trust);// ❹
    assert(err == noErr);
    err = SecTrustSetAnchorCertificates(trust, (CFArrayRef) [NSArray arrayWith
        Object:(id) cert]);// ❺
    assert(err == noErr);
```

```
    ret=[self valuate:cert Trust:trust Date:date];// ❻
    err=SecTrustSetAnchorCertificatesOnly(trust,NO);// ❼
        CFRelease(trust);
        CFRelease(policy);
        CFRelease(cert);
    return ret;
}
```

代码说明：

❶ 首先将一个 X.509 格式的文件证书（.cer 文件）以 NSData 加载。

❷ 然后再使用 SDK 安全框架的 SecCertificateCreateWithData 函数获得一个 SecCertificateRef 结构。

❸❹ 要对证书进行评估，除了要获得 SecCertificateRef 结构外，还需要获得一个 SecPolicyRef 和一个 SecTrustRef 结构，我们分别用 SecPolicyCreateBasicX509 函数和 SectTrustCreateWithCertificates 函数达到了这一点。

❺ 此外，我们还需要用 SecTrustSetAnchorCertificates 函数设置锚证书。锚证书是颁发这个证书的机构的证书，验证证书时用锚证书来验证证书是否由指定机构所颁发。由于我们的证书是“自签名”的，即自己颁发给自己的证书，所以锚证书仍然使用的是同一个证书。（锚证书使用证书链的概念，见上一节）。

❻ 调用 valuate 方法(后面介绍)进行证书验证。

注意：SecTrustSetAnchorCertifiactes 函数的影响是全局的。即设置锚证书之后，将影响后面所有的证书评估，一直到第 2 次调用 SecTrustSetAnchorCertificates 函数。

---

❼ 在评估结束后，应当调用 SecTrustSetAnchorCertificatesOnly 函数使锚证书失效。

接下来介绍 valuate 方法。在 valuate: Trust:Date:方法中，用指定的日期验证证书有效性，如代码清单 10-1 所示。

代码清单 10-1

```
-(ValuateResult)valuate:(SecCertificateRef)cert Trust:(SecTrustRef)trust Date:
 (NSDate*)date{
    ValuateResult ret;
    OSStatus err;
    SecTrustResultType  result;
     static const char * kTrustNames[8] = {
         "Invalid",
         "Proceed",
         "Confirm",
         "Deny",
         "Unspecified",
         "RecoverableTrustFailure",
         "FatalTrustFailure",
```

```
        "OtherError"
    };

    err = SecTrustSetVerifyDate(trust, (CFDateRef) date);// ❶
    assert(err == noErr);

    CFAbsoluteTime trustTime;
    trustTime = SecTrustGetVerifyTime(trust);// ❷

    err = SecTrustEvaluate(trust, &result);// ❸
    assert(err == noErr);

    if (result < (sizeof(kTrustNames) / sizeof(*kTrustNames))) {
        if (result==5) {// ❹
                // 设了个有效的日期，进行再次评估
                NSLog(@"%@",efficientDate);
                err=SecTrustSetVerifyDate(trust, (CFDateRef)efficientDate);
                assert(err==noErr);
                err=SecTrustEvaluate(trust, &result);// ❺
                assert(err == noErr);
                if (result==4) {// if result=Unspecified,
                    // 返回证书已过期，这里我们假设把证书尚未生效的情况也算作过期
                    ret= ValuateResultEXPIRED;
                }
        }else if (result==4) {// 如果第1次就通过评估，证书有效
                ret=ValuateResultOK;
        }else {
                ret=ValuateResultFAILED;//证书 无效
        }
    } else {

    }
     return ret;
}
```

代码说明：

❶ 评估某个日期时间（由参数 date 指定）是否在证书有效期内，使用 SecTrustEvaluate 函数。在此之前，我们可用 SecTrustSetVerifyTime/SecTrustSetVerifyDate 函数指定评估的时间/日期。这里，我们将证书评估时间指定为参数传入的日期。

❷ SecTrustGetVerifyTime 函数用于获取证书评估时间。

❸ 开始第 1 次评估。

❹ 因为苹果的安全框架的问题，证书、密钥和信任服务中没有单独的验证有效期的函数，API 也不能读取完整的证书信息，我们无法读出证书中的有效期字段。在评估一个证书时，安全框架不会告诉你证书评估失败的原因，到底是因为证书过期了，还是因为

别的原因无效。因此只能采用间接的方式，进行两次证书的评估，第 1 次用当前日期评估，如果评估通过，说明当前是处于有效期内，如果第 1 次评估未通过，需要进行第 2 次评估。这次评估采用一个明显处于有效期内的日期进行评估，这次评估如果通过则说明显然是由于评估时间无效导致的评估失败，否则是证书有其他方面的问题。

如果证书是过期了，SecTrustEvaluate 函数返回结果是可恢复的信任错误：RecoverableTrustFailure。如果这样，我们先用 SecTrustSetVerifiyDateh 函数设置一个明显有效的时间（efficientDate，我们在调用类的初始化方法实例化对象时，应该传入一个有效的证书日期,并用于初始化 efficentDate 成员）,然后再次用 efficentData 作为评估日期调用 SetTrustEvaluate 函数。如果函数返回结果为 Unspecified，则表明证书第 1 次评估的时间是过期的。

注意：SecTrustEvaluate 函数相当奇怪，如果评估有效，函数返回的结果是 Unspecified 而不是其他。Unspecified 以外的返回结果表明评估失败（对于自签名证书）。

---

❺ 第 2 次评估，此句之前，我们已经用 effeicentData 重新设置了评估时间。

接下来是在应用程序委托中调用 MyCertificate 类的代码，如代码清单 10-2 所示。

代码清单 10-2

```
-(IBAction)valuate{
    NSDateFormatter* fmt=[[NSDateFormatter alloc]init];// ❶
    [fmt setDateFormat:@"yyyy-MM-dd"];// ❷

    NSString* path=[[NSBundle mainBundle]pathForResource:@"dlt.cer" ofType:nil];// ❸
    MyTrustService* myTrust=[[MyTrustService alloc]
                             initWithFilename:path EfficientDate:[fmt dateFrom
                             String:@"2011-5-10"]];// ❹

    ValuateResult ret=[myTrust trustValuate:[fmt dateFromString:tfDate.text]];// ❺
    switch (ret) {//❻
        case ValuateResultOK:
            showMessage(@"",@"证书在有效期内");
            break;
        case ValuateResultEXPIRED:
            showMessage(@"",@"证书不在有效期内");
            break;
        default:
            showMessage(@"",@"证书校验失败");
            break;
    }
    [myTrust release];

}
```

代码说明：

❶ 首先，我们初始化了一个 NSDateFormatter 对象，用于格式化日期字符串。

❷ 设置 NSDateFormatter 日期格式。NSDateFormatter 对象是专门用于将日期格式化为指定格式的日期字符文本，或将指定格式的日期字符文本顺利转换为 NSDate 的。具体说，当你用 NSDateFormatter 的 setDateFormat:方法指定一个日期格式串之后，NSDateFormatter 就可以将一个 NSString 以该格式转换为 NSDate，或者这一转换的可逆过程。

❸ 然后，我们获取应用程序束中证书文件所在的完整路径。

注意：文件证书被我们放在项目文件里，它将被编译到“应用程序束”（即编译后的可执行文件）中，而不是“应用程序文件夹”（沙盒）中。

---

❹ 然后调用 MyTrustService 的初始化方法，构造一个 MyTrustService 用于评估证书。MyTrustService 的初始化方法在前面已经做过介绍，它需要一个在证书有效期之内的日期作为参数，理由如前所述。注意这个日期应该是容易获得的，比如你创建证书的日期。这个日期不需要很精确，只需要在证书有效期之内的任意一个日期就行。

❺ 接下来，我们调用 MyTrustService 的 trustValuate 方法评估证书（trustValuate 方法我们已经介绍过了）。

❻ 根据返回结果判断证书是否有效。

运行程序，文本框中有一个有效的日期，不改变它，直接点验证按钮，程序弹出一个“证书在有效期内”的消息，如图 10-1 所示。

如果修改文本框内的日期为一个无效日期，比如 2011-5-1，点击验证按钮，则程序显示“证书不在有效期内”的消息，如图 10-2 所示。

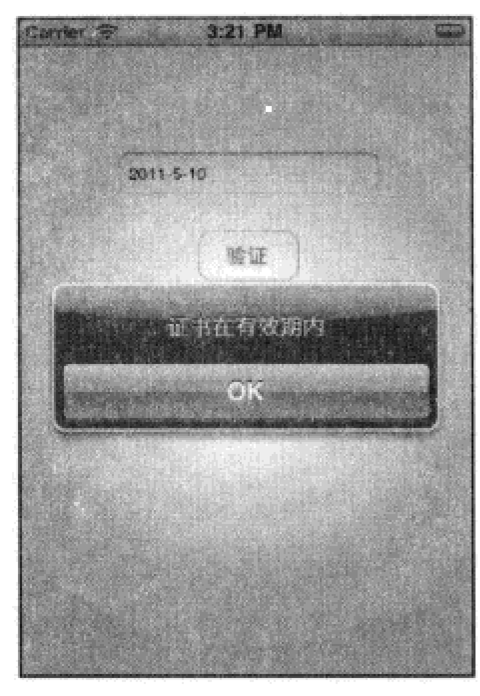


图 10-1　使用有效日期进行证书评估

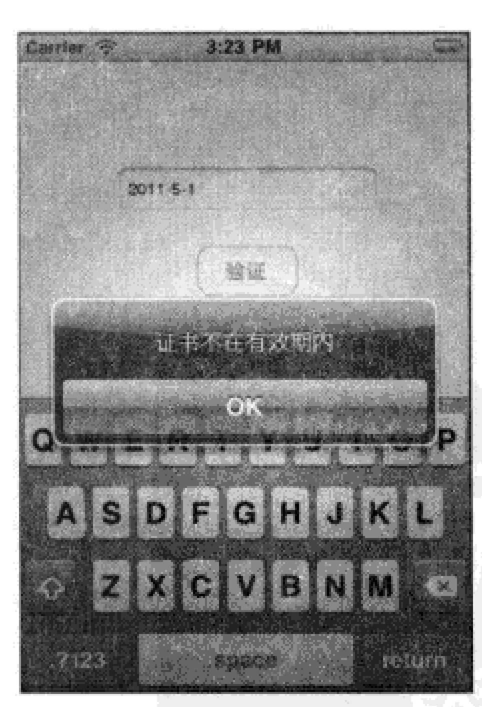


图 10-2　使用无效日期进行证书验证

10.2 使用 SSL 和服务器通信

SDK 支持 iPhone 客户端和服务器建立 SSL/TLS 连接进行通信，这需要使用到 CFNetwork 框架。

CFNetwork 框架是核心服务框架中的一个框架，它提供了一个抽象化的网络协议库。这种抽象使得进行各种网络任务都非常容易，比如： BSD sockets、SSL /TLS 加密连接、DNS 解析、 HTTP / HTTPS 协议、FTP 协议、Bonjour 服务等，它包括以下 API:

- CFFTP API
- CFHTTP API
- CFHTTP 认证 API
- CFHost API
- CFNetServices API
- CFNetDiagnostics API

虽然 CFNetwork 已经对各种网络协议进行了封装，使 iPhone 能够“更容易地”使用各种网络服务——但不幸的是，CFNetwork 基本上是一个 C 语言函数库。有时候，为了打开一个连接，你仍然不得不写大量模式化的、结构性的代码。

本书不准备花大量篇幅来介绍 CFNetwork 框架。由于第 7 章中已经介绍过 ASIHttpRequest 框架，而 ASIHttpRequest 是一个在 Objective C 下的极为优秀的网络 API 框架，它把对 HTTP 和 HTTPS 协议的支持都封装到 ASIHttpRequest 类中，在使用上显得非常简单。所以我们仍然决定使用 ASIHttpRequest 来介绍 iPhone 如何与服务器进行 HTTPS 通信。

提示：关于 CFNetwork，你可以参考苹果文档“CFNetwork Programming Guide”。

在 ASIHttpRequest 中进行 HTTPS 通信，其实跟 HTTP 通信只有少许差别。即在进行 HTTPS 请求时指定客户端要使用的 SSL 证书。

```
ClientSSLCert* csc=[[ClientSSLCert alloc]init];
[request setClientCertificateIdentity:[csc identity]];
// 对于自签名证书
[request setValidatesSecureCertificate:NO];
[request startSynchronous];
```

ClientSSLCert 类是关键的一环，我们在这个类中使用安全框架读取了客户端证书。ClientSSLCert 的 identity 方法从一个 PKCS12 格式（.p12）的客户端证书文件中把证书数据读取到一个 SecIdentityRef 结构中，然后返回这个 SecIdentityRef。这些技术和上一节“10.1.2 在 iPhone 中使用 X.509 证书”中讲述的技术是完全一致的。使用这个 SecIdentifyRef 结构，可以设置 ASIHTTPRequest 对象的 clientCertificateIdentity 属性，然后如同进行普通 HTTP 请求一样和服务器进行 HTTPs 通信。

注意：如果客户端证书是自签名的，请将 request 的 validatesSecureCertificate 选项设置为 NO。

我们在光盘“source/第 10 章/SSLCertificate”目录中，也放入了 ClientSSLCert 类的实现文件，读者可以自行参考。

10.3 OpenSSL

前面介绍了苹果的安全框架。苹果的安全框架提供了有限的功能，它能做到的永远要比你想象得少。但它毕竟属于苹果官方 SDK 中的内容。由于苹果商店不能使用第三方框架，很多时候它是我们唯一的选择。但对于企业开发（企业版 IDP 证书），却不在限制之列。接下来，就让我们看一看在 iOS 下的第三方安全算法库。对于 iOS 程序员来说，其实能有的选择并不多。我将向大家介绍 Unix 系统中大名鼎鼎的 OpenSSL 库。

OpenSSL 确实十分强大，然而其糟糕的文档却让人难以满意。大家可用的参考资源主要有两方面：一是 OpenSSL 库中的源代码，另一个是 openssl.cn 论坛。

10.3.1 在 iOS 中使用 OpenSSL 库

OpenSSL 库是一个 BSD C 函数库。要想在 iOS 应用中使用它，需要将它编译为静态库。在 iPhone 中，除了使用源代码，静态库是唯一共享代码的方式。

首先下载源代码库，下载地址：http://www.openssl.org/source/，当前最新版本是 1.0.0d。

然后，打开 crypto/ui/ui_openssl.c 进行编辑。将“static volatile sig_atomic_t intr_signal;”一行修改为“static volatile int intr_signal;”；否则会出现一个编译错误。

为了能在模拟器和真机上都能使用 OpenSSL 库，我们需要针对不同目标编译不同的静态库：i386 库（用于 iPhone 模拟器）、armv6 库（armv6 架构的 iOS 使用）、armv7 库（armv7 架构的 iOS 使用）。这 3 个库的编译过程大同小异，但使用的编译指令和 configure 不同。

1. 编译 i386 库（用于 iPhone 模拟器）。

打开终端，执行以下命令：

```
mkdir ssllibs
```

这将在用户主目录下建立 ssllibs 目录。切换到 openssl-1.0.0a(我们以 1.0.0a 这个版本为例)解压后的目录，在其下建立 3 个子目录：

```
cd openssl-1.0.0a
mkdir openssl_armv6 openssl_armv7 openssl_i386
```

然后执行目录下的 congfigure 脚本：

```
./configure BSD-generic32 --openssldir=/Users/<username>/openssl-1.0.0a/openssl_i386
```

这将把编译目标指定为 openssl_i386 目录，并设置配置环境。<username>为你的用户主目

录，请根据实际情况替换该字符串。

编辑 makefile 文件，找到：

```
CC= gcc
```

修改为：

```
CC= /Developer/Platforms/iPhoneSimulator.platform/Developer/usr/bin/gcc -arch i386
```

这将指定编译器路径及编译的目标架构。

下一行，在 CFLAG = 的后面增加以下代码：

```
-isysroot /Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhone
    Simulator4.0.sdk
```

这将指定编译时使用的 BaseSDK。

进行编译：

```
make
make install
```

注意，请检查 openssl_i386/lib 目录下 libcrypto.a 和 libssl.a 是否生成。

2. 编译 armv6 库（armv6 架构的 iOS 使用）

执行以下命令：

```
mv openssl_i386 ../ssllibs
```

这将用 move 命令将编译好的 i386 库移动到 ssllibs 目录中。

执行以下命令：

```
make clean
```

这将清除上次编译的配置。

执行 configure 命令：

```
./configure BSD-generic32 --openssldir=/Users/<username>/openssl-1.0.0a/openssl_ armv6
```

这将重新生成新的编译配置，使用新的目标目录。<username>为你的用户主目录，请根据实际情况替换该字符串。

修改 makefile 文件，将 CC=gcc 修改为：

```
CC= /Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/gcc -arch armv6
```

注意：这里是 iPhoneOS.platform 而不是先前的 iPhoneSimulator.platform 了。

同样，需要在 CFLAG=后面加上以下代码：

```
-isysroot /Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS4.0.sdk
```

注意，这里重新定义了 BaseSDK。

可以进行编译了：

```
make
make install
```

注意请检查 openssl_armv6/lib 目录下 libcrypto.a 和 libssl.a 是否生成。

3. 编译 armv7 库（armv7 架构的 iOS 使用）

执行以下命令：

```
mv openssl_armv6 ../ssllibs
```

这将先前编译好的 armv6 库移到 ssllibs 目录。

执行以下命令：

```
make clean
```

这将清除前面编译配置。

执行 configure 配置编译环境：

```
./configure BSD-generic32 --openssldir=/Users/<username>/openssl-1.0.0a/openssl_armv7
```

这将重新生成新的编译配置，使用新的目标目录。<username>为你的用户主目录，请根据实际情况替换该字符串。

修改 makefile 文件，将 CC=cc 修改为：

```
CC= /Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/gcc -arch armv7
```

注意 gcc 编译选项 arch 由 armv6 变为了 armv7。

同时，在 CFLAG=后面添加以下命令：

```
-isysroot /Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS4.0.sdk
```

注意这里重新定义了 BaseSDK。

进行编译：

```
make make
install
```

请检查 openssl_armv7/lib 目录下 libcrypto.a 和 libssl.a 是否生成。

把编译结果移到 ssllibs 目录：

```
mv openssl_armv7 ../ssllibs
```

这一步不要忘记，待会我们将利用这些编译结果制作“通用”静态库。

4. 制作“通用”静态库

通用静态库是一个“多架构”文件，它是多个单一架构静态库的融合。使用通用静态库的好处是，在链接时不需要针对每个架构修改编译指令和编译配置。

注意：制作“通用”静态库需要使用 Mac OS X 的 lipo 命令（具体请参考 Mac OS X 手册）。

合并 libcrypto.a 库的命令：

```
lipo -create ../ssllibs/openssl_i386/lib/libcrypto.a ../ssllibs/openssl_armv6/
    lib/libcrypto.a ../ssllibs/openssl_armv7/lib/libcrypto.a -output ../ssllibs/
    libcrypto.a
```

这将 3 种架构的 libcrypto.a 库合并为单个 libcrypto.a 文件并放在 ssllibs 目录中。

合并 libssl.a 库的命令：

```
lipo -create ../ssllibs/openssl_i386/lib/libssl.a ../ssllibs/openssl_armv6/lib/
    libssl.a ../ssllibs/openssl_armv7/lib/libssl.a -output ../ssllibs/libssl.a
```

这将 3 种架构的 libssl.a 库合并为单个 libssl.a 文件并放在 ssllibs 目录中。请检查 ssllibs 目录下 libcrypto.a 和 libssl.a 是否生成。

翻译之后，在 Xcode 项目中引入 OpenSSL 静态库要在 Xcode 项目中使用静态库，需要导入相应的.a 文件和头文件，并告诉 Xcode 该库的头文件在哪里可以找到。关于静态库的使用，请参考第 6 章 6.3.4 节“封装自己的静态库”。现在，我们来看看如何在项目中导入 OpenSSL 通用静态库。它包括以下具体步骤：

1）把 OpenSSL 的 include 目录（头文件）拷贝到项目文件夹。

2）把 libcrypto.a 和 libssl.a 文件拷贝到项目文件夹。

3）把 libcrypto.a 和 libssl.a 文件拖到项目的 Framework 组中。

4）打开 target 的 Build Settings，将 Library Search Path 设置为：$(inherited) “$(SRCROOT)”；将 User Header Search Paths 设为 include；选中 Always Search User Paths 选项。

经过以上步骤，你就可以在你的项目中使用 OpenSSL 了。

10.3.2 OpenSSL 应用实例——使用 OpenSSL 进行 MD5 加密

在企业应用中，经常需要对敏感数据进行加密，例如系统账号、密码、企业机密数据或用户私密信息等。下面我们应用 OpenSSL 安全加密算法库，实现对字符串的 MD5 加密。

MD5 的全称是 Message-Digest Algorithm 5（信息摘要算法），是一种非常优秀的安全加密算法，不收取任何版权费用，在实际应用中常用于数字证书私钥和密码的安全保护。它可将大容量信息在用数字签名软件签署私人密匙前被“压缩”成一种保密的格式（就是把一个任意长度的字节串变换成一定长度的大整数）。它采用典型的不对称加密算法，无法解密。OpenSSL 实现了 MD5 加密算法。

MD5 算法描述和 C 语言实现请参考 Internet RFCs 1321。

新建 Window-based application，命名为 OpenSSLTest。使用如下命令“Add → Existing Frameworks　Others…”，把 libssl.a 和 libcrypto.a 加进来（即我们前面制作的“通用”库）。

打开 Target 的 Build Settings 面板，在 Header Search Paths 中加入 OpenSSL 的头文件路

径，如：

```
/Users/<yourname>/Library/openssl-1.0.0a/include
```

<yourname>为你的用户主目录，请根据实际情况替换该字符串。

注意：勾选“Recursive”（搜索子目录）。

接下来写点简单的代码。为求简便，我们把所有代码写在 main.m 里。下面是 main 方法代码：

```
#import <UIKit/UIKit.h>
#include <Openssl/md5.h> //❶
void Md5(NSString*);
int main(int argc, char *argv[]) {

    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    Md5(@"12345");//❷
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

❶ 导入 OpenSSL 库中的 md5.h 头文件

❷ 调用自定义函数 Md5

MD5 函数的代码如下：

```
void Md5(NSString* string){
    unsigned char *inStrg = (unsigned char*)[[string dataUsingEncoding:NSASCIIString
                            Encoding] bytes];// ❶
    unsigned long lngth = [string length];// ❷
    unsigned char result[MD5_DIGEST_LENGTH];// ❸
    NSMutableString *outStrg = [NSMutableString string];
    MD5(inStrg, lngth, result);// ❹
    unsigned int i;
    for (i = 0; i < MD5_DIGEST_LENGTH; i++){// ❺
        [outStrg appendFormat:@"%02x", result[i]];
    }
    NSLog(@"input string:%@",string);
    NSLog(@"md5:%@",outStrg);
}
```

代码说明：

❶❷❸ 我们需要调用 OpenSSL 库的 MD5 函数实现指定功能。MD5 函数需要 3 个参数，第 1 个参数是 char*类型，即一个 C 字符串，需要加密的字符串；第 2 个参数需指定该字符串的长度；最后一个参数是无符号字符串数组，但它是一个输出参数，MD5 加密的结果通过它取得。

❹ 调用 MD5 函数。

❺ 由于 MD5 加密的结果(函数的输出参数)是一个 128 位的无符号字符串（实际上是一个无符号整型数），我们用循环将每个 uchar 转换为%02x 格式，即两位的 16 进制数，然后输出。

可以在控制台查看程序的输出如下：

```
input string:12345
md5:827ccb0eea8a706c4c34a16891f84e7b
```

10.4 CommonCrypto

iOS SDK 中自带了 CommonCrypto 安全算法库，它包含了一系列加密算法的实现，如最常见的 HASH 算法：AES、MD5、SHA、DES 等，这些算法都是以 C 语言实现的。它的使用非常方便，不需要导入任何第三方框架和库，直接在项目中引用它就可以了：

```
#import <CommonCrypto/CommonCryptor.h>
```

下面，我们将以一个示例程序来演示如何使用 CommonCrypto 对指定文本进行 DES 加密和解密。示例程序位于光盘“source/第 10 章/DesEncryptTest”目录。

DES 是 1972 年由 IBM 研制出来的一种对称加密算法。最早的 DES 算法采用 64 位（bit）长度的密钥（其有效位数为 55 位，每字节最后一位为奇偶校验位）对明文进行加密。此后，基于 DES 密钥长度相对较短的缺点，又提出了三 DES 加密算法——即采用两个密钥对明文进行三次加密。虽然 56 位 DES 加密有密钥长度不长、分组较短、运算速度慢等缺点，但在某些“弱加密”的场合，仍然是一种十分安全的算法。

本节将介绍如何使用 DES 算法对传输文本进行加密和解密。

运行示例程序，将出现如图 10-3 所示的界面。界面中有一个 TextView 和一个按钮：同时用于对文本框中的文字进行加密和解密操作。当点击加密按钮时，会将第一个 TextView 中的文字加密，加密后的数据以 16 进制的形式显示在文本框中，同时按钮文字转变为“解密”。当点击解密按钮则进行相反运算。

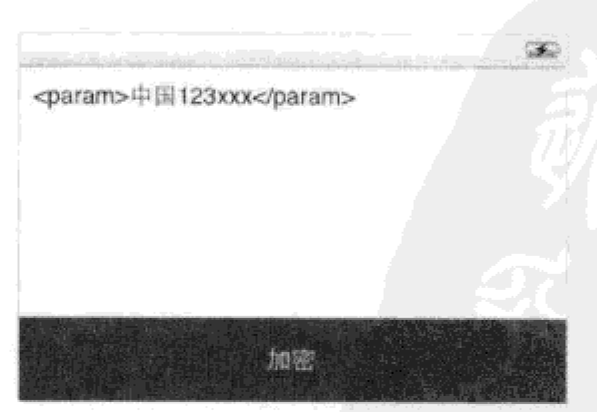


图 10-3 使用 DES 算法加密和解密

密码的加密和解密都使用类 DES 的 desData:key: CCOperation:实现。参数 CCOperation 指

定操作类型（加密/解密）。在 desData 方法中，我们调用了 CommonCrypto 库的 CCCrypt 函数：

```
// 初始化向量，CBC 加密时使用
static char cIv[8]={1,0,6,2,9,5,5,9};
//data 是需要加密或解密的数据，keyString 是密码（一般是 8 位），op 表示加密/解密
+ (NSData *)desData:(NSData *)data key:(NSString *)key CCOperation:(CCOperation)op
{
    // buffer 大小应该由下面公式计算。如果 bufferSzie 太小，无法正确解密
    NSUInteger bufferSize= ([data length] + kCCKeySizeDES) & ~(kCCKeySizeDES -1);
    char buffer[bufferSize] ;
    memset(buffer, 0, sizeof(buffer));
    size_t bufferNumBytes;
    CCCryptorStatus cryptStatus = CCCrypt(op,
                                          kCCAlgorithmDES,
                                          kCCOptionPKCS7Padding,
                                          [key UTF8String],
                                          kCCKeySizeDES,
                                          cIv,
                                          [data bytes],
                                          [data length],
                                          buffer,
                                          bufferSize,
                                          &bufferNumBytes);
    //   NSLog(@"crypt status:%d",cryptStatus);
    if(cryptStatus == kCCSuccess)
    {
        return [NSData dataWithBytes:buffer length:bufferNumBytes];
    }
    return nil;
}
```

CCCrypt 函数用于进行一次无状态的加密或解密操作，其各参数说明如下：

```
CCCryptorStatus CCCrypt(
    CCOperation op,              //说明操作类型，如加密或者解密
    CCAlgorithm alg,             //说明算法，如 AES、DES 等
    CCOptions options,           //选项，如对齐方式、ECB 模式等
    const void *key,             //加密所使用的密钥
    size_t keyLength,            //密钥长度
    const void *iv,              //初始向量，可选，用于 CBC 模式
    const void *dataIn,          //本次操作的输入，如要加密或解密的数据，
    size_t dataInLength,         //dataIn 的长度
    void *dataOut,               //加密或解密结果
    size_t dataOutAvailable,     //dataOut 缓存的大小
    size_t *dataOutMoved)        //操作完成后写到 dataOut 的字节数
```

注意 options 参数和 iv 参数。由于服务器端采用的是 DES/CBC 加密，我们在调用时使用

了 initial vector（iv 参数），并且 options 参数指定为 kCCOptionPKCS7Padding，而不要指定为 kCCOptionPKCS7Padding | kCCOptionECBMode。

在服务器端，我们以 Java 来实现 DES/CBC 加密和解密，具体请参考示例项目文件夹下的 DES.java 源文件。

10.5 本章小结

安全问题伴随网络而来。本章内容是第 7 章“网络”的延深。在这一章中涉及的所有概念和技术都与网络密切相关，比如证书、密钥、ASHTTPRequest 和 OpenSSL 网络编程库等。

本章从苹果最基本的“证书、密钥和信任服务”概念开始介绍，从而开始了一系列安全技术的介绍。这些技术是环环相扣的。

网络中常使用数字证书进行用户身份的识别与验证，因此了解如何利用 Cocoa Security framework 框架进行证书的有效性验证是必要的。数字证书技术也常在 TSL/SSL 协议中使用，因此本章还介绍了如何利用 ASIHTTPRequest 框架对 HTTPS 协议的支持进行 SSL 通信。此外，在网络中传输敏感数据时，我们常常需要对数据进行加密，要求每个开发者都了解安全算法的细节并进行实现是不现实的。于是本章又介绍了 Unix 系统下的 OpenSSL 库和 iOS SDK 自带的 CommonCrypto 库，以及如何在 iPhone 中使用这些安全算法库实现数据的加密和解密。

实际上，关于安全的主题包括了许多内容，并不仅仅限于证书、密钥和加密算法。在企业应用中，网络数据在传输中的安全也是非常重要的，比如使用企业 APN 和 VPN 来保证网络传输安全，这会涉及企业部署、配置描述文件的制作和安装等。此外，安全还应当包括：用户验证、物理设备绑定、客户端授权、应用程序自动保护（锁屏）等。由于篇幅所限，这些内容将不在本书的讨论范围。下一章，我们将讨论和多媒体相关的话题。

第 11 章　多媒体、绘图及动画

本章将介绍 iPhone 的多媒体和一般动画技术，包括视频和声频的播放、Quartz 2D 和 Core Animation。把所有这些内容放在一起，是因为这些技术和 SDK 的四层结构中的第二层 Media 有着密切关系。SDK 四层结构分别是指 iOS 框架从高到低的四个层次结构：Cocoa Touch、Media、Core Service、Core OS。iOS 程序员接触得最多的是第一层 Cocoa Touch，正如前面几章所介绍的内容正属于此。在本章中，我们将开始接触第二层，即 Media 层中的核心内容。

Media 层是对 iPhone 音频和视频协议的封装，例如 OpenGL ES、EAGL、Quartz、Core Animation、Core Audio、Open Audio Library 和 Media Player。

本章首先介绍的是音频和视频的播放，这分别由 Media Player、Core Audio 框架所提供。

接下来介绍的是 Quartz。Quartz 或 Quartz 2D，是 Cocoa 最主要的 2D 图形库，它位于 Media 层中的核心部分。其中介绍 Quartz 2D 的图形上下文、路径、变换、图案、阴影、渐变和遮罩。

最后介绍大家关心的 Core Animation 动画技术。包括 Core Animation 中的一般性技术：隐式动画和显式动画。Core Animation 也是 Media 层的核心内容。

11.1　播放视频

音频和视频实际上比图像更复杂，正因为如此，苹果将它们的处理完美地封装到了 Media Player 框架中。要使用 Media Player 框架，需要在项目中加入 MediaPlayer.framework 框架，并在源代码中导入头文件“<MediaPlayer/MediaPlayer.h>”，就能够很轻松地播放音频和视频了。

Media Player 框架包含 MPMoviePlayerController 类。而从 SDK3.2 开始，MPMoviePlayerController 被 MPMoviePlayerViewController 所替代，本书将只讨论 MPMoviePlayerViewController。如果你的发布平台为 SDK 3.1，请注意在代码中保持兼容。

将要播放的文件以 URL 的形式封装，然后传递到 MPMoviePlayerViewController 类的初始化方法中，构建 MPMoviePlayerViewController 对象。通过使用 UIViewController（注意，不是 MPMoviewPlayerViewController）的 presentMoviePlayerViewControllerAnimated:方法即可呈现播放界面。

```
- (void)presentMoviePlayerViewControllerAnimated:
(MPMoviePlayerViewController *)moviePlayerViewController;
```

提示：这个方法定义在 MPMoviePlayerViewController.h 里，它是在一个 UIViewController 的新类别中所定义的，它是类别中的方法，而不是类原有的方法。

PDG

MPMoviePlayerViewController 使用通知模式通知观察者某些事件正在发生。观察者需要注意 3 个通知：

- MPMoviePlayerContentPreloadDidFinishNotification：文件已加载
- MPMoviePlayerPlaybackDidFinishNotification：重放已完成
- MPMoviePlayerScalingModeDidChangeNotification：缩放模式改变

注意：第一个方法 MPMoviePlayerContentPreloadDidFinishNotification 在 SDK 3.2 中已弃用，可以用 MPMediaPlaybackIsPreparedToPlayDidChangeNotification 通知替代。

你需要在一个对象中注册该对象（例如一个 View Controller）为某个通知的观察者，然后在注册时指定的回调方法中对通知进行适当的处理：

```
[[NSNotificationCenter defaultCenter] addObserver:self selector:@selector
    (movieReplayed:)
    name:MPMoviePlayerPlaybackDidFinishNotification
    object:nil];
```

上述代码将 self 注册为 MPMoviePlayerPlaybackDidFinishNotification 通知的观察者，同时将 self 的 movieReplayed: 方法注册为通知处理方法。

示例项目位于光盘“source/第 11 章/MoviePlayerDemo”目录下。它演示了如何播放本地的视频文件。

11.2 播放音频

使用 Media Player 框架还可以播放音频。代码和播放视频没有任何区别，但很多时候，我们不需要在播放音频时出现一个笨拙的播放窗口。例如游戏背景音乐，只需要在后台播放出声音就好，并不需要任何显示界面。

你当然可以使用 Core Audio 框架播放声音。使用 AVAudioPlayer 类，可以播放苹果专有的 CAF 音频文件和 mp3 音乐。以下是 AVAudioPlayer 对象的初始化代码：

```
AVAudioPlayer  *player = [[AVAudioPlayer alloc] initWithContentsOfURL:[NSURL file
    URLWithPath:path] error:&error];
```

注意，使用 AVAudioPlayer 需要导入<AVFoundation/AVFoundation.h>。

如果要捕获 AVAudioPlayer 的回放事件，需要设置 AVAudioPlayer 对象的委托属性，并实现 AVAudioPlayerDelegate 协议。如果需要监控播放进度，可以使用 AVAudioPlayer 的 currentTime 和 duration 两个属性。

在光盘的“source/第 11 章/AudioPlayerDemo”中，有一个示例项目，演示了 AVAudioPlayer 的使用。注意，使用 AVAudioPlayer 需要在项目中引用 AVFoundation.framework 框架并在源文件中导入<AVFoundation/AVFoundation.h>。

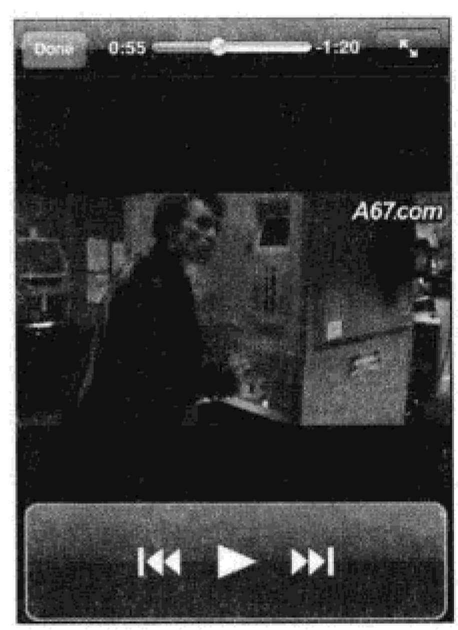


图 11-1　播放视频

图 11-2　音频播放示例：AudioPlayerDemo

11.3　Quartz 2D

Quartz 2D 是 Core Graphics 的框架的重要组成部分，它提供了一个用于绘制二维图形的 C 函数库。详细信息请参阅苹果的“Quartz 2D Programming Guide”文档。

11.3.1　图形上下文

在进行任何 Quartz 2D 绘图之前，需要获取一个上下文。图形上下文描述了一个绘图目标，包括了所有的绘图参数以及绘图操作时所需的设备相关信息。例如绘图时所用的颜色、裁切区域、线宽、线型、字体等。

创建图形上下文相当简单。很多时候，你可以直接从 UIKit 对象中检索出默认的上下文。这个上下文是和当前 UIKit 对象相关联的：

```
CGContextRef ctx=UIGraphicsGetCurrentContext();
```

有时，你需要自己创建一个上下文，然后检索出它们：

```
UIGraphicsBeginImageContext(CGSizeMake(20,20));
CGContextRef ctx=UIGraphicsGetCurrentContext();
```

UIGraphicsBeginImageContext 函数创建了一个位图上下文，同时由于上下文是在栈中创建的，因此该位图上下文现在位于栈顶。

UIGraphicsGetCurrentContext()函数则从栈顶返回了这个位图上下文。此后，你可以使用图形上下文进行 Quartz 2D 绘图了。绘图结束，你可能想将所绘制的图形保存到 UIImage 中。UIGraphicsGetImageFromCurrentImageContext()函数可以把当前上下文中绘制的图像拷贝到 UIImage 中并返回。

最后，可以调用 UIGraphicsEndImageContext()函数将该上下文从栈顶弹出（删除）。

11.3.2 路径

路径定义 1 个或多个形状或子路径。子路径由直线、曲线或二者共同构成，子路径可以开放，也可以闭合。它可以是简单形状，比如，点、线、圆、弧、曲线 、矩形、星形等，也可以是更复杂的形状。

1. 绘制路径

首先创建路径，然后渲染路径，请求 Quartz 进行绘制。创建路径使用 CGContextBeginPath 函数，然后使用绘图函数（表 11-1 列出了一些绘图函数）绘制路径。在绘制不连续的路径时，需要使用 CGContextMoveToPoint 函数把画笔移动到新的位置开始绘制。

绘制路径需要经过多个步骤才能完成。一个绘制路径的示例代码如下所示：

```
CGContextRef ctx=UIGraphicsGetCurrentContext();//❶
CGContextBeginPath(ctx); //❷
CGContextAddArc(ctx,10,10,10,0,2*M_P,1);//❸
CGContextSetRGBFillColor(ctx,1,0,0,1); //❹
CGContextFillPath(ctx); //❺
```

代码说明：

❶ 由于 Quartz 2D 绘图总是在图形上下文中进行的，因此我们需要获取一个有效的图形上下文。如果代码是在一个 UIKit 对象中（我们一般总是在 UIView 的上进行 Quartz 绘图），则可以使用默认的上下文（当前上下文）。

❷ 在开始创建路径之前，还需要调用 CGContextBeginPath 函数，标志接下来要进行路径的创建。

❸ 创建路径，一般使用各种矢量绘制函数（如后面所述），比如用 CGContextAddArc 函数绘制一段弧。

❹ 设置路径的填充色。

❺ 用填充色填充已绘制的路径。

注意：路径绘制结束需要关闭路径（CGContextClosePath）。CGContextFillPath 函数会用当前填充色或样式填充路径所包围的区域，并自动关闭路径。

除了使用填充色填充路径，还可以用图案、渐变色和阴影来处理路径，详细信息请参考苹果文档“Quartz 2D 编程指南”。另外，除了填充路径，还可以用 CGContextStrokePath 来描绘路径的线条和边缘。

2. 常见的绘图函数

这里列出了一些绘制各种简单矢量图形的函数（见表 11-1）。当然， UIKit 中也有一些简单的绘图函数，如 UIRectFrame、UIRectFill 以及 UIBezierPath。

从 iOS 3.2 开始，UIBezierPath 变为公有类。它的 roundedRectBezierPath 方法对于创建各种圆角矩形来说非常有用。

表 11-1　常用绘图函数

函　　数	参　　数	说　　明
CGContextBeginPath	context	创建新路径
CGContextAddArc	context,x,y,radius, startangle,endangle, clockwise	创建弧
CGContextAddEclipseInRect	context,CGRect	在矩形内部创建内切圆
CGContextAddLineToPoint	context,x,y	在当前点和指定点间绘制线段
CGContextAddRect	context,CGRect	创建矩形
CGContextMoveToPoint	context,x,y	将画笔移动到某个点而不绘制
CGContextClearRect	context,CGRect	擦除矩形
CGContextFillRect	context,CGRect	创建填充矩形
CGContextStrokeRect	context,CGRect	创建轮廓矩形
CGContextStrokenRectWithWidth	context,CGRect,width	用指定线宽绘制轮廓矩形

光盘“source/第 11 章/CGPathDemo”目录下的示例项目，演示了如何使用图形上下文和绘图函数绘制一个圆角矩形。

RoundRect 类继承了 UIView，这样就可以重载它的 drawRect:方法。

在使用 Quartz2D 进行图形绘制时，必须重载 UIView 子类的 drawRect:方法。只有在 drawRect:方法中使用 UIGraphicsGetCurrentContext()函数，才能得到正确有效的上下文，否则可能会得到“invalid context”错误。

在 drawRect:方法里，用 UIGraphicsGetCurrentContext()函数获得上下文，然后调用 ContextAddRoundedRect()函数绘制图形。ContextAddRoundedRect()函数使用 CGContext MoveToPoint()函数和 CGContextAddArcToPoint()函数创建一条封闭的路径，这条路径是一个圆角的矩形，然后用填充色填充它。

你可能会奇怪为什么只看到矩形的 4 个圆角的绘制代码，而没有看到将 4 个圆角连接在一起的线段的绘制代码。关键就在 CGContextFillPath(c):这一句上。CGContextFillPath()函数会在填充路径的同时，自动关闭路径，于是 4 个圆角之间会自动加入连接线。

最后，在你想绘制图形的时候，调用 UIView 子类的 setNeedsDisplay 方法或 setNeedsLayout 方法（不要直接调用 drawRect:方法）。

11.3.3　变换

变换是使图形移动、旋转或者缩放。变换主要有以下两种：

1. 当前变换矩阵

当前变换矩阵（Current Transformation Matrix，CTM）是一个计算机图形学上的概念，指一个 3×3 阶的齐次坐标矩阵，它是用户空间和设备空间中间存在的一个转换矩阵，运用于当前图形的所有顶点。CTM 运算包括旋转、缩放和平移。

旋转矩阵：

```
void CGContextRotateCTM(CGContextRef c,CGFloat angle);
```

缩放矩阵：

```
void CGContextScaleCTM(CGContextRef c,CGFloat sx,CGFloat sy);
```

平移矩阵：

```
void CGContextTranslateCTM(CGContextRef c,CGFloat tx, CGFloat ty);
```

以下用一小段代码演示如何应用 CTM 变换。首先我们创建图形上下文，我们可以直接从一个 UIImage 对象创建图形上下文：

```
CGImageRef imageRef = image.CGImage;
int m_width = image.size.width;
int m_height = image.size.height;
uint32_t * imageData = (uint32_t *) malloc(m_width * m_height * sizeof(uint32_t));
 CGColorSpaceRef colorSpace1= CGImageGetColorSpace(imageRef);
context1=CGBitmapContextCreate(m_imageData, m_width, m_height, 8, m_width* sizeof
    (uint32_t), colorSpace1, kCGBitmapByteOrder32Little|kCGImageAlphaNoneSkipLast);
```

CGBitmapContextCreate 函数用于根据指定位图创建一个位图上下文。image 是一个 UIImage 对象。

由于源图片大小和上下文的大小可能不一致，因此需要告诉 Quartz 绘制图形时使用何种插值算法：

```
CGContextSetInterpolationQuality(context1, kCGInterpolationHigh);
```

kCGInterpolation 有 3 种取值：

- Low——最近插值算法；
- Medium——线性或双线性插值；
- High——二次插值或双三次插值。

设置 Quartz 的抗锯齿选项：

```
CGContextSetShouldAntialias(context1, YES);
```

然后使用 CTM 函数对当前变换矩阵进行 CTM 变换，先进行平移、旋转（-90°）再平移：

```
CGContextTranslateCTM(context1, m_width/2, m_height/2);
CGContextRotateCTM(context1, -M_PI_2);
```

```
CGContextTranslateCTM(context1, -m_height/2, -m_width/2);
```

上面代码先平移 CTM 到矩阵中心，旋转-90 °，再把 CTM 平移回原处，这样坐标原点始终在原来的位置，不然旋转后的图片会偏移到原 CTM 一半的位置。

CTM 使用设备空间，即坐标原点位于左下角。而用户空间的坐标原点位于左上角。

接下来绘制图片：

```
CGContextDrawImage (context1, CGRectMake(0, 0, m_height, m_width), [image CGImage]);
```

这样，将原本的图片会由竖向转变为横向（旋转 90°）。

注意：CTM 变换是对当前变换矩阵进行操作。与 Mac 下不同，iPhone 下需要先进行变换，再进行图形的绘制。

---

2. 仿射变换

除了 CTM 变换，还可以用另一种方式对图形进行转换。

仿射变换是另一种二维坐标到二维坐标的转换。仿射变换经过对坐标轴的缩放、旋转，平移后取得原坐标在新坐标中的值。与 CTM 变换不同，仿射变换不会修改 CTM，而是先创建出一个仿射变换，然后将这个仿射变换应用到 CTM。这样做的好处是明显的，可以在代码中可以重用这个仿射变换。如果程序中有大量重复的转换动画，则仿射变换更利于转换的重用。

常用的仿射变换包括旋转、平移、缩放。分别使用 3 个函数创建这 3 种仿射变换对象：

- ❑ 创建旋转矩阵：CGAffineTransform CGAffineTransformMakeRotation（CGFloat angle）;
- ❑ 创建缩放矩阵：CGAffineTransform CGAffineTransformMakeScale（CGFloat sx,CGFloat sy）;
- ❑ 创建平移矩阵：CGAffineTransform CGAffineTransformMakeTranslation（CGFloat tx, CGFloat ty）;

可以在已有仿射变换的基础上再次进行变换，这需要使用到下列仿射变换函数：

- ❑ 旋转矩阵：CGAffineTransform CGAffineTransformRotate(CGAffineTransform t,CGFloat angle）;
- ❑ 缩放矩阵：CGAffineTransform CGAffineTransformScale（CGAffineTransform t,CGFloat sx, CGFloat sy）;
- ❑ 平移矩阵：CGAffineTransform CGAffineTransformTranslate（CGAffineTransformt, CGFloat tx, CGFloat ty）;
- ❑ 组合变换：CGAffineTransform CGAffineTransformConcat（CGAffineTransform t1, CGAffineTransform t2）;

下面我们来看看如何应用这些仿射变换函数。

首先，准备一个 UIImage 对象，并获得它的大小（size 和 bounds）：

```
CGImageRef img = [image CGImage];
CGFloat width = CGImageGetWidth(img);
```

```
CGFloat height = CGImageGetHeight(img);
CGSize size = CGSizeMake(width, height);
CGRect bounds = CGRectMake(0, 0, width, height);
```

在每次转换前，需要获取重置的仿射变换，否则多次仿射变换后的角度和位置偏移会越来越大：

```
CGAffineTransform transform = CGAffineTransformIdentity;
```

接下来进行仿射变换。首先创建一个平移转换，将坐标右移到图片高度的位置：

transform = CGAffineTransformMakeTranslation(height, 0.0f);

这样是为了留出足够的空间，这样当图片逆时针旋转 90°后，不会旋转到屏幕以外（同前面的例子一样）。注意，仿射变换使用设备空间，坐标原点位于左下角。

然后进行旋转：

```
transform = CGAffineTransformRotate(transform, M_PI / 2.0f);
```

因为 CGAffineTransformRotate 会在原有 Affine Transform 的基础上进行旋转，所以就没有必要使用组合变换函数 CGAffineTransformConcat 了。

由于需将图片进行横竖转换，转换后原图的宽和高应当被颠倒了，即宽变为高，高变为宽。下面代码进行了宽高的交换：

```
CGFloat origHeight = size.height;
size.height = size.width;
size.width = origHeight;
CGSize newSize = size;
```

在上下文中设置新图形绘制大小为新的尺寸（因为宽高交换了）：

```
UIGraphicsBeginImageContext(newSize);
```

获取上下文：

```
CGContextRef context = UIGraphicsGetCurrentContext();
```

在 CTM 中应用仿射变换：

```
CGContextConcatCTM(context, transform);
```

CGContextContcatCTM 函数把仿射变换应用于当前转换矩阵 CTM。

绘制图形：

```
CGContextDrawImage(context, bounds, img);
```

提示：可以用函数 UIGraphicsGetImageFromCurrentImageContext 把刚才在上下文中绘制的图形拷贝到一个 UIImage 对象中，以便在 UIImageView 中显示。

示例程序位于光盘“source/第 11 章/TransformDemo”目录下，它演示了仿射变换。

在项目目录中有一张图片“未命名.png”。这张图片并不是很符合我们的阅读习惯。你可以在预览程序中查看原图片的效果，这张图片并不是正面朝上的，它的方向旋转了 90°。当示例程序运行时，点击“旋转 90 度载入图片”按钮时，这张图片被加载到视图中，但它的方向现在已经是正向的了。通过以下代码，我们使用仿射变换将图片顺时针旋转了 90°：

```
transform = CGAffineTransformRotate(transform, MPI / 2.0f);
```

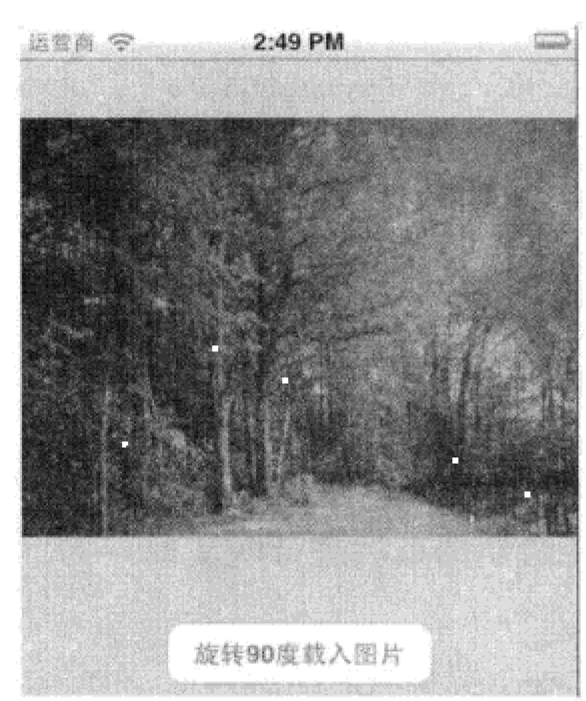


图 11-3　原来竖向的图片旋转为横向

11.3.4　图案

图案是一系列重复绘制在上下文中的操作。可以把图案当成颜色来使用，当使用图案进行绘图时，Quartz 会把画布分割成无数个小格子，每个小格子就是一个图案。

当使用颜色填充一个矩形时，使用下列函数：

```
CGContextSetRGBFillColor(ctx,1,0,0,1);//设置路径的填充色
CGContextFillRect(ctx,CGRectMake(0,0,100,60));//绘制填充矩形
```

用图案填充一个矩形时，也类似：

```
CGContextSetRGBFillParttern(ctx,pattern,alpha);//设置路径的填充图案
CGContextFillRect(ctx,CGRectMake(0,0,100,60));//绘制填充矩形
```

提示：图案和填充色是图形上下文状态的一部分，所以设置完图案或填充色之后，会自动应用于当前上下文的填充行为中。

其中 pattern 是一个 CGPatternRef 结构，使用函数 CGPatternCreate 创建。而 CGPatternRef 的创建及两种 Pattern（Colored 和 Stencil）的绘制过程比较复杂，限于篇幅，在此不深入讨论。具体信息请参考苹果“Quartz 2D Programming Guide”文档中“Patterns”主题。

11.3.5 阴影

阴影能产生一种类 3D 效果，使平面图形呈现立体效果。阴影受 3 个方面因素的影响：x 偏移量、y 偏移量和模糊系数。前二者不用多说，模糊系数则是决定了阴影的柔和程度。阴影分为两种：有颜色的、无颜色的。二者绘制的步骤稍有不同，前者比后者复杂一些。下列代码演示了颜色阴影：

```
CGContextSaveGState(ctx);
myColorSpace = CGColorSpaceCreateDeviceRGB();
CGColorRef myColor = CGColorCreate(myColorSpace,{1,0,0,.6});
CGContextSetShadowWithColor(ctx,CGSize(-15,20),5,myColor);
CGContextSetRGBFillColor(ctx,0,0,1,1);
CGContextFillRect(ctx,CGRectMake(0,0,60,40));
CGColorRelease (myColor);
CGColorSpaceRelease (ctx);
CGContextRestoreGState(ctx);//恢复图形上下文
```

由于阴影色属于当前上下文的一部分，因此为防止对后面的图形绘制产生影响，在绘制阴影前要先保存当前上下文（以便绘制结束后进行恢复）：

```
CGContextSaveGState(ctx);
```

并在绘制结束之后恢复所保存的上下文：

```
CGContextRestoreGState(ctx);
```

要设置颜色，首先创建一个 RGB 颜色空间：

```
myColorSpace = CGColorSpaceCreateDeviceRGB();
```

因为创建 RGB 颜色时需要一个 RGB 颜色空间作为参数。接下来用 CGColorCreate 函数创建一种 RGB 颜色（以作为阴影色）：

```
CGColorRef myColor = CGColorCreate(myColorSpace,{1,0,0,.6});
```

阴影也是图形上下文中的一种特殊状态，用 CGContextSetShadowWithColor 函数在上下文中设置一个有颜色阴影：

```
CGContextSetShadowWithColor(ctx,CGSize(-15,20),5,myColor);
```

以下代码绘制一个填充矩形，这个矩形将应用我们设置的颜色阴影：

```
CGContextSetRGBFillColor(ctx,0,0,1,1);
CGContextFillRect(ctx,CGRectMake(0,0,60,40));
```

最后是释放所声明的颜色空间和 RGB 色。

```
CGColorRelease (myColor);
CGColorSpaceRelease (ctx);
```

11.3.6　渐变

颜色渐变指从一种颜色向另一种颜色的过渡，颜色渐变分为两种：线性渐变和径向渐变。Quartz 提供了 CGGradient 类和 CGShading 类以支持颜色渐变。

以下代码是一个线性渐变的例子。

```
CGGradientRef myGradient;
CGColorSpaceRef myColorspace;
size_t num_locations = 2;
CGFloat locations[2] = { 0.0, 1.0 };
CGFloat components[8] = { 1.0, 0.5, 0.4, 1.0,  //开始颜色
                        0.8, 0.8, 0.3, 1.0 }; //终止颜色
myColorspace = CGColorSpaceCreateWithName(kCGColorSpaceGenericRGB);
              //创建 CGColorSpace 对象
myGradient = CGGradientCreateWithColorComponents (myColorspace, components, locations,
           num_locations);
CGPoint myStartPoint, myEndPoint;// 开始点，结束点
       myStartPoint.x = 0.0;
       myStartPoint.y = 0.0;
       myEndPoint.x = 1.0;
       myEndPoint.y = 1.0;
CGContextDrawLinearGradient (ctx, myGradient, myStartPoint, myEndPoint, 0);
```

首先是渐变对象 CGGradientRef 的创建。用 CGGradientCreateWithColorComponents 函数可以获得一个 CGGradientRef：

```
CGGradientRef myGradient;
myGradient = CGGradientCreateWithColorComponents (myColorspace, components,
      locations,num_locations);
```

CGGradientRef 函数的第一个参数是一个颜色空间。使用 CGColorSpaceCreateWithName 可以创建颜色空间：

```
CGColorSpaceRef myColorspace;
myColorspace = CGColorSpaceCreateWithName(kCGColorSpaceGenericRGB);
```

CGGradentRef 函数的第二个参数是一个 CGFloat 数组，指定了渐变色的开始颜色、终止颜色以及过渡色（如果有的话）：

```
CGFloat components[8] = { 1.0, 0.5, 0.4, 1.0, 0.8, 0.8, 0.3, 1.0 };
```

提示：数组中每 4 个浮点数构成了一个 RGBA 颜色值(A 为 Alpha 透明度），每个浮点数都是 0.0 ~ 1.0 之间的小数。因此 components[8]中只定义了两种颜色，开始色和终止色。

CGGradientCreateWithColorComponents 函数的第三个参数指定了每个颜色在渐变色中的位置，值介于 0.0 ~ 1.0 之间，0.0 表示最开始的位置，1.0 表示渐变结束的位置，例如，以下代

码定义了两种颜色的位置，一种位于渐变开始，一种位于渐变结束：

```
CGFloat locations[2] = { 0.0, 1.0 };
```

第四个参数指定了渐变中使用的颜色数。在示例代码中，只使用了两种颜色：

```
size_t num_locations = 2;
```

最后，用 CGContextDrawLinearGradient 函数绘制出颜色渐变：

```
CGContextDrawLinearGradient (ctx, myGradient, myStartPoint, myEndPoint, 0);
```

其中 myStartPoint 和 myEndPoint 是两个 CGPoint 结构，分别定义了矩形区域的左上角和右下角的点坐标。渐变色将在这个矩形区域内绘制。最后一个参数为选项，0 为从起点开始绘制，1 为从终点开始绘制。

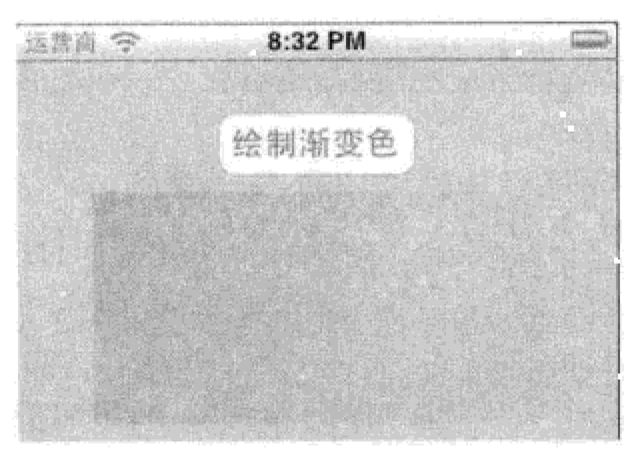


图 11-4　线性渐变示例

光盘“source/第 11 章/GradientDemo”目录中的示例项目演示了渐变色的绘制。当点击“绘制渐变色”按钮时，会在 ImageView 中显示一个由红色到黄色的渐变色填充的矩形。

11.3.7　透明图层

透明图层等同于 Photoshop 中“图层”的概念，图层是一个独立的部分，可以把图层中的对象视为同一个整体。可以把多个图形通过图层的概念把它们组合在一起，等同于一个图形。要使用图层绘制，只需如下步骤：

- 调用 CGContextBeginTransparencyLayer 函数开始图层绘图；
- 开始绘制任意的图形；
- 调用函数 CGContextEndTransparencyLayer 结束图层绘图。

在 CGContextBeginTransparencyLayer 和 CGContextEndTransparencyLayer 之间使用 Quartz 2D 绘制的图形将被视作一个图层中的对象。

11.3.8　位图及遮罩

位图和遮罩是类似的，在 Quartz 中都用 CGImageRef 结构表示。位图由像素点集合构成。每个像素都是 JPG、TIFF 以及 PNG 图片文件中的一个点。遮罩也是位图，但没有颜色。Quartz

使用当前填充色绘制遮罩。遮罩的色深为 1 ~ 8 位。以下代码演示如何绘制位图：

```
UIImage *image=[UIImage imageNamed:@" xxx.png" ];
UIGraphicsBeginImageContext(image.size);//创建位图上下文
CGContextRef ctx=UIGraphicsGetCurrentContext();//获得图像上下文
CGContextDrawImage(ctx,CGRectMake(0,0,image.size.width,image.size.height),[image
        CGImage]);//按原图绘制
CGContextSetLineWidth(ctx,20);//设置线宽
CGContextBeginPath(ctx);//创建路径
CGContextMoveToPoint(ctx,0,0);//移动画笔到左上角
CGContextAddLineToPoint(ctx,image.size.width,image.size.height);//从图形左上角到右
        下角画一条线
CGContextSetStrokeColorWithColor(ctx,[[UIColor redColor]CGColor]);//设置描边的颜色
CGContextStrokePath(ctx);//对当前路径进行描边,同时路径结束
UIImage *newImage=UIGraphicsGetImageFromCurrentImageContext();//保存当前绘制的内容
        到一个 UIImage 对象
UIGraphicsEndImageContext();//绘图结束，弹出当前位图上下文
```

这段代码演示了如何使用 CGContextDrawImage 函数绘制 png 位图，并在绘制的图形中加上了一条从左上到右下的对角线。

遮罩技术常用于产生一些特殊效果，如用一张图片遮挡另一张图片（部分），或者用指定颜色（颜色范围）遮罩图片的局部。

Quartz 2D 中提供 3 种遮罩技术：位图遮罩、颜色遮罩、上下文裁切遮罩。这里简单介绍一下位图遮罩。

位图遮罩最关键的一步是 CGImageMaskCreate 函数的使用。该函数根据提供的用作遮罩的位图信息创建图形遮罩，原型如下：

```
CGImageRef CGImageMaskCreate (
           size_t width,
           size_t height,
           size_t bitsPerComponent,
           size_t bitsPerPixel,
           size_t bytesPerRow,
           CGDataProviderRef provider,
           const float decode[],
           int shouldInterpolate);
```

以下代码实现了第一种遮罩效果：

```
CGImageRef imgRef = [image CGImage];//原图
CGImageRef maskRef = [mask CGImage];//遮罩图片
//调用 CGImageMaskCreate 函数，由图片 maskRef 生成图形遮罩
CGImageRef actualMask = CGImageMaskCreate(CGImageGetWidth(maskRef),CGImageGet
        Height(maskRef),CGImageGetBitsPerComponent(maskRef),CGImageGetBitsPerPixel
        (maskRef),CGImageGetBytesPerRow(maskRef),CGImageGetDataProvider(maskRef),
```

```
        NULL,false);
//调用 CGImageCreateWithMask 函数应用遮罩到 imgRef 上
CGImageRef masked = CGImageCreateWithMask(imgRef,actualMask);
```

imgRef 是原始图片的 CGImage 对象，maskRef 是用做遮罩的 CGImage 对象。CGImageMaskCreate 函数利用 maskRef 创建了遮罩 actualMask。然后将这个遮罩应用到 imgRef 图片上。

CGImageMaskCreate 函数需要提供许多信息，但基本上都是在 maskRef 的基础上获得的。具体信息请参考苹果文档“Quartz 2D Programming Guide”。

示例项目位于光盘“source/第 11 章/MaskDemo”目录。该示例项目演示了位图遮罩技术。在项目目录中，准备了 3 张图片，分别是：Icon.png、IconDeep.png 和 IconBase.png。如图 11-5 所示。

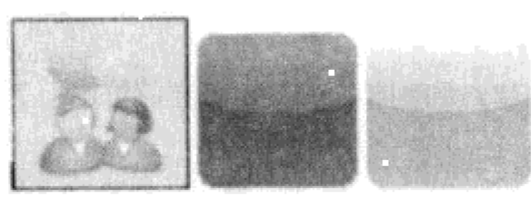


图 11-5　三张素材图片

用这三张图片进行了位图遮罩运算。第 1 张图片用做遮罩层，第 2 张和第 3 张图片用做背景层，两张背景图片分别用于在按钮的按起（未按下）和按下时和遮罩图片进行合成，以显示按钮按起和按下时的不同状态：

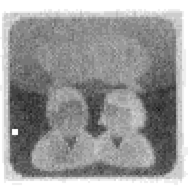
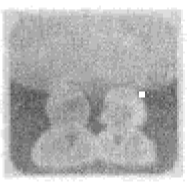



图 11-6　按钮在按起和按下状态下显示不同的位图

因为遮罩层只需要保留 Alpha 值，所以对 Icon.png 进行了一些特殊处理，只保留了它的 Alpha 通道，这个工作使用 Photoshop 或任何图形处理软件都可以完成。处理后的 Icon.png 效果如图 11-7 所示。由于除了 Alpha 通道外所有颜色通道都被舍弃，因此直接预览 Icon.png 时你只能看到一片空白，为了便于读者查看效果，图 11-7 实际上加了一点背景色。

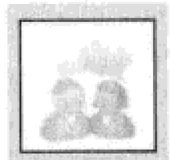


图 11-7　处理后的 Icon.png

注意：Icon.png 直接查看时是一片模糊的白色，蓝色实际上是在最下面加入了一个蓝色填充的图层，在保存图片时并不需要这个额外的图层（可将其隐藏或删除后保存）。

接下来在 viewDidLoad 方法中，调用 createButtonImage 方法加载 3 张位图分别进行遮罩处理，并将合成结果作为按钮的状态图片。

遮罩处理在 doImageMask 方法中完成。对于遮罩图片 Icon.png，还使用 fillImageWhite 方法进行了处理，主要目的是把 Icon.png 进一步转换为黑色和白色的位图。具体地说，是把透明的部分填充白色，有不透明的部分填充黑色。这样才能和背景图层（IconDeep.png 和 IconBase.png）作进一步的遮罩处理。

最后，调用 Quartz 函数 CGImageMaskCreate，将处理后遮罩图片和背景图片进行合成，并将合成后的图片返回。通过这种方式，模拟了类似 iPhone 于主屏上水晶按钮的半透明效果。

11.4 Core Animation

Core Animation 是 SDK 四层结构中的第二层 Media 层（上一层是 Cocoa Touch）中的重要组成部分。上一节已经讲述了 Media 层中的 Quartz，而 Core Animation 位于 Quartz 之上，与 Quartz 的 C 语言函数库不同，Core Animation 是基于 Quartz 的 Objective-C 封装。通过前面的介绍，我们了解到使用 Quartz 中进行图形的绘制是比较麻烦的，而 Core Animation 相比较而言更加容易。使得我们不必再使用 Quartz 创建动画。

iPhone 中几乎所有的可视化元素都继承自 UIView，UIView 有一个 layer 属性，它是一个 CALayer 对象。CALayer 是 Core Animation 的基础。所有的 Core Animation 动画都是基于 CALayer 的。通过修改 CALayer 的属性，Core Animation 让动画更加流畅。这些属性包括：anchorPoint、backgroundColor、opacity、position、transform。

使用 Core Animation，需要在项目中引入 Quartz Core 框架，同时导入头文件 <QuartzCore/QuartzCore.h>。Core Animation 有两种动画类型：隐式动画和显式动画。下面分别进行介绍。

11.4.1 隐式动画

隐式动画是最简单的 Core Animation 动画类型。隐式动画不需要显式地创建 CABasicAnimation 或 CAAnimation 对象，只需在一个“动画块”的代码块中修改目标对象的 CALayer 属性，示例代码如下所示：

```
[UIView beginAnimations:nil context:NULL]; //❶
CGAffineTransform moveTransform=CGAffineTransformMakeTranslation(200,200); //❷
[imageView.layer setAffineTransform:moveTransform]; //❸
imageView.layer.opacity=1; //❹
[UIView commitAnimations]; //❺
```

代码说明：

❶ 此句标志了一个“动画块”的开始。

❷ 声明一个仿射变换 moveTransform，CGAffineTransformMakeTranslation(200,200)函数返回了一个平移变换对象，其中的两个参数表明 x 坐标和 y 坐标偏移量。

❸ 修改 CALayer 的 transform 属性。

❹ 修改 CALayer 的 opacity 属性。

❺ 动画块结束。这将导致 imageView 从最初的位置向右 200 个像素、向下 200 个像素的位置移动，同时透明度从原来的值向 1 渐变。

11.4.2 显式动画

显式动画与隐式动画不同，需要显式地定义一个 CAAnimation 或 CABasicAnimation 对象来执行的。其中，CABasicAnimation 继承自 CAPropertyAnimation，CAPropertyAnimation 又继承自 CAAnimation。以下代码演示了使用 CAAnimation 创建显式动画的过程。CATransition 也继承自 CAAnimation。

```
CATransition *transition = [CATransition animation];// ❶
    transition.type = @"oglFlip";//❷
    transition.subtype = kCATransitionFromRight;// ❸
    transition.duration = 1.0f;// ❹
    transition.timingFunction = UIViewAnimationCurveEaseInOut;// ❺
        [self.view exchangeSubviewAtIndex:0 withSubviewAtIndex:1];// ❻
    // 设置转换动画
    [[self.view layer] addAnimation:transition forKey:@"oglFlipAnimation"];//❼
```

代码说明：

❶ 初始化一个 CATransition 对象。

❷ 设置 CATransition 的 type 属性为 oglFlip,注意 Core Animation 有很多内置的动画效果，可以通过设置字符串属性 type 使用它们。oglFlip 效果是一个前后翻转的效果，它属于苹果私有 API。私有 API 的意思是苹果修改它时不会通知你。而且使用私有 API 的应用可能不会被苹果商店审核通过。同样的动画效果还有：kCATransitionFade、kCATransitionMoveIn、kCATransitionPush、kCATransitionReveal、pageCurl、pageUnCurl、rippleEffect 等。

❸ 除了 type 属性外，CATransition 还有一个 subtype 属性，用于定义动画的附加属性，比如，动画变化的方向，可以有 kCATransitionFromRight、kCATransitionFromLeft、kCATransitionFromTop、kCATransitionFromBottom 四个方向。

❹ 设置动画持续时间。

❺ 设置 timingFunction 属性，timeingFunction 是时间线调度函数，负责将动画的播放和时间轴同步。 UIViewAnimationCurveEaseInOut 会产生一种开始时慢、中间稍快、结束时又慢下来的动画播放效果。

❻ 交换视图的 subView。这个动作将使用动画的方式进行。

❼ 使用 CALayer 的 addAnimation 方法，这将导致动画被执行。

示例项目位于光盘“source/第 11 章/CATransitionDemo”目录。通过示例项目，演示了一个使用显式动画进行翻页的效果，如图 11-8 所示。

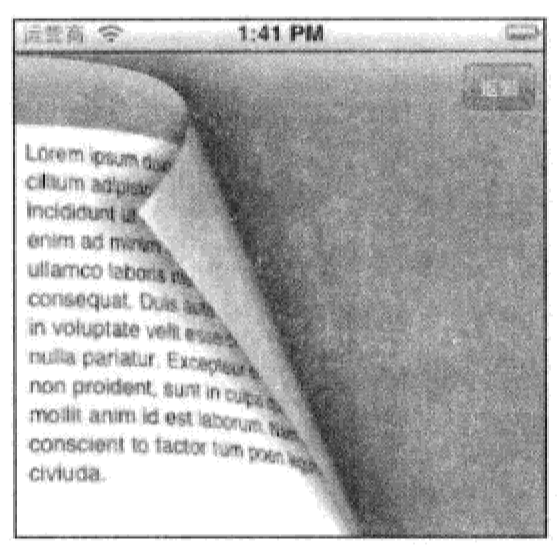


图 11-8　使用显式动画进行视图的切换

11.5　本章小结

本章介绍了 iPhone 的位图、多媒体、Quartz 2D 和 Core Animation。这些技术随着它们在 SDK 中所处的层次不同而有着不同的可替换性。一般而言，越处于底层的技术，其可替换性越差，重要性越高。因为上层技术所能提供的功能，我们都能在下层技术中找到替代方法，反之却不能。

其中 Quartz 是苹果在 Mac 和 iOS 中提供的一种较核心的、中间层图形处理技术，有一定的复杂性，同时又屏蔽了更低层的细节。在本章中，对整个 Quartz 2D 进行了较全面的介绍（除了部分内容，如文字）。这些内容是比较难掌握的，但在一些特殊场合（比如高级自定义和自绘制控件），我们仍然不得不使用它们。

对于动画，介绍了 Quartz Core（即 Core Animation）框架，包括隐式动画和显式动画。当然，对于企业应用来说，使用动画的地方总是有限的，一些简单的动画比如变换、翻页，使用 Core Animation 足矣。但游戏开发则例外。在游戏中，动画是不可缺少的重要内容，绘图和动画代码占据了游戏的很大比重。因此出于性能方面的原因（Quartz 的性能并不是很高），游戏开发中很少选择 Core Animation 来创建动画。而更多的选择是 EAGL、OpenGL 或 OpneGL ES。此外，一些第三方的游戏开发框架，已经把 OpenGL ES 和 EAGL 封装得很好了，例如时下流行的 Cocos2d 或 Cocos2d-x。《Learn iPhone and iPad Cocos2d Game Development》是值得推荐的，如果你对 Cocos2d 感兴趣，可以去看一看。

第 12 章　多点触摸及手势

多点触摸（Multi-Touch，又称多点触控、多点感应）是人机交互技术上的一个进步，它始于上世纪 80 年代。1982 年，多伦多大学发明了感应食指指压的多点触控屏幕。此后，多点触控技术逐渐发展，响应时间不断缩短，该技术也逐渐进入主流应用。2005 年，生产多点触控产品 iGesture 和多点触控键盘的“约翰埃利亚斯”和“鲁尼韦斯特曼”被苹果电脑收购。此后，苹果在其创新式设备中大量使用支持多点触摸的电容式触摸屏。

多点触摸技术是 iOS“手势”识别的基础。正是基于多点触摸技术，苹果公司提出了“手势”的概念。“手势”是 SDK 在操作系统的底层“多点触摸”技术上进行封装的结果。虽然“手势”识别的不仅仅是“多点触摸”，也包含“单点触摸”，但一般我们可以认为前者包含了后者。所以当我们说“多点触摸”的时候，实际上是在说“多点触摸+单点触摸”（iPhone 最多支持 5 点触摸，iPad 最多支持 11 点触摸）。

实际上，所有的应用程序都需要处理“手势”中的某些动作。例如手指“扫过”屏幕，可能会导致新视图的显示。同样，“捏合”手势通常用于图片或视图的缩放操作。

然而，“手势”不仅要识别手指放在屏幕上的根数，还要监控每根手指的移动轨迹。因此，手势是由“手指触摸的点数”加上“每根手指的动作”来构成的。在 iOS 4 以前，手势识别由开发人员负责，这往往需要进行复杂的数学运算。苹果公司意识到这种情况的复杂性和手势对 iPhone 用户界面的重要性，之后在 iOS4 中加入了 UIGestureRecognizer 类，使开发者更容易实现各种手势的识别。

本章将初步介绍 iOS 4 对手势的识别与支持。

12.1　手势识别器：UIGestureRecognizer 类

从 iOS 4 开始，新的 SDK 新增了一个类，叫做 UIGestureRecognizer，用于支持对手势的识别。UIGestureRecognizer 是一个抽象类，它拥有一系列子类，每个子类都用于识别某类特定的手势如：移动、点击等，它们是：

- UITapGestureRecognizer ——“轻击”手势识别。可以设定为“单击”的识别和“连击”的识别。
- UIPinchGestureRecognizer ——“捏合”手势识别。该手势通常用于缩放视图或改变可视组件的大小。
- UIRotationGestureRecognizer ——“转动”手势识别。用户两指在屏幕上做相对环形运动。
- UIPanGestureRecognizer ——“平移”手势识别。识别拖曳或移动动作。
- UISwipeGestureRecognizer ——“轻扫”手势识别。当用户从屏幕上划过时识别为该手

势。可以指定该动作的方向（上、下、左、右）。

- UILongPressGestureRecognizer ——“长按”手势识别。使用 1 指或多指触摸屏幕并保持一定时间。

这些手势识别器必需和视图（UIView 类）通过 addGestureRecognizer:方法联系在一起。同时，还需要为每个识别器指定一个响应方法，以便当指定手势发生时进行调用。removeGestureRecognizer:方法可以将识别器从视图中移出，方法的参数用于指定要移除的识别器。

手势“响应方法”或“处理方法”是，当手势识别器侦测到某个手势发生时，采用“目标-动作模式”来通知应用程序，应用程序回调用手势的“响应方法”或“处理方法“。这个方法在识别器创建之时就要指定。这样，只要有对应的手势发生，就会调用创建时指定的方法。每个手势响应方法都会被传入一个 UIGestureRecognizer* sender 对象，从中可以获取到和手势有关的信息。例如，对于“捏合”手势，可以通过这个参数获取缩放系数、捏合速度。对于“转动”手势，则可以获得转动的次数及速度。

手势分为“连续手势”和“不连续手势”两种。“不连续手势”只会导致调用响应方法一次。“轻击”（包括多击）仅仅会触发一次响应方法。而“轻扫”、“平移”、“旋转”、“捏合”则是连续手势，它们会连续不断地调用响应方法直到手势结束。

这 6 类手势识别器都支持多点触摸，但支持的程度不一。例如，有的识别器只能识别 2 点触摸，有的识别器可识别 1 ~ 5 点（iPhone）或 1 ~ 11 点（iPad）触摸。

- UITapGestureRecognizer（轻击）——识别 1 ~ 5 点（iPhone）或 1 ~ 11 点（iPad）的多点触摸。它有一个 numberOfTouchesRequired 属性，可以指定有效的点数，即需要几根手指同时轻击屏幕才被识别为“轻击”手势。
- UIPinchGestureRecognizer（捏合）——只识别 2 点触摸，所谓“捏合”手势是专指 2 指在屏幕上做径向运动（相向或反向）。单点或超过 2 点的多点触摸对“捏合”手势无效。因此“捏合”手势没有任何用于指定手势有效手指数目的属性。
- UIPanGestureRecognizer（平移）——识别 1 ~ 5 点（iPhone）或 1 ~ 11 点（iPad）的多点触摸。与“轻击”手势不同，“平移手势”有两个用于指定手指数目的属性：maximumNumberOfTouches 和 minimumNumberOfTouches。这两个属性用于指定一个有效的手指数目的范围。例如，minimumNumberOfTouches=1，maximumNumberOfTouches=3，则说明用户无论使用 1 根手指、2 根手指还是 3 根手指进行拖动，都是有效的平移手势。
- UISwipeGestureRecognizer（轻扫）——识别 1 ~ 5 点（iPhone）或 1 ~ 11 点（iPad）的多点触摸。与“轻击”手势一样，“轻扫”手势也有 numberOfTouchesRequired 属性，可以指定“轻扫”动作的有效手指数。
- UIRotationGestureRecognizer（转动）——同“捏合”手势一样，“转动”手势只识别 2 点触摸，所谓“转动”手势是专指 2 指在屏幕上做弧形运动。单点或超过 2 点的多点触摸对“转动”手势无效。因此“转动”手势没有任何用于指定手势有效手指数目的属性。

- ❑ UILongPressGestureRecognizer（长按）——识别 1 ~ 5 点（iPhone）或 1 ~ 11 点（iPad）的多点触摸。它有一个 numberOfTouchesRequired 属性，可以指定有效的点数，即需要几根手指持续地按住屏幕才被识别为“长按”手势。

12.2 创建手势识别器

iOS 4 SDK 内置了 6 种手势识别器（UIGestureRecognizer 子类），用于对常见手势的识别。通过使用 SDK 提供的这些手势识别器类，大大减少了程序员们的工作量。下面，我们将分别进行介绍。

1. 识别轻击手势

轻击手势使用 UITapGestureRecognizer 类进行识别。在 alloc 和 init 时必需指定一个手势触发时调用的方法引用（即 selector 选择器）。使用 numberOfTapsRequired 属性可以定义轻击必需被连续操作的次数。如以下代码所示，当识别器侦测到连续轻击 2 次时，将触发 tapDetected: 方法。

```
UITapGestureRecognizer *doubleTap=
          [[UITapGestureRecognizer alloc]
          initWithTarget:self
          action:@selector(tapDetected:)]; doubleTap.numberOfTapsRequired = 2;
[self.view addGestureRecognizer:doubleTap];
[doubleTap release];
```

手势的响应方法 tapDetected:可能定义为如下代码：

```
- (IBAction)tapDetected:(UIGestureRecognizer *)sender {
    // 响应手势的代码
}
```

2. 识别捏合手势

捏合手势使用 UIPinchGestureRecognizer 类识别，代码如下：

```
UIPinchGestureRecognizer *pinchRecognizer =
      [[UIPinchGestureRecognizer alloc]
      initWithTarget:self
      action:@selector(pinchDetected:)];
[self.view addGestureRecognizer:pinchRecognizer];
[pinchRecognizer release];
```

3. 识别转动手势

转动手势使用 UIRotationGestureRecognizer 类，代码如下：

```
UIRotationGestureRecognizer *rotationRecognizer =
     [[UIRotationGestureRecognizer alloc]
     initWithTarget:self
```

PDG

```
        action:@selector(rotationDetected:)];
[self.view addGestureRecognizer:rotationRecognizer];
[rotationRecognizer release];
```

4. 识别平移手势

平移手势使用 UIPanGestureRecognizer 类。平移手势是最基本的连续手势，例如手指在屏幕上随意乱划可以被识别为平移或拖拽操作：

```
UIRotationGestureRecognizer *panRecognizer =
      [[UIPanGestureRecognizer alloc]
      initWithTarget:self
      action:@selector(panDetected:)];
[self.view addGestureRecognizer:panRecognizer];
[panRecognizer release];
```

如果轻扫和平移都添加在同一个 view 中，那么很可能大部分轻扫手势都会被识别为平移。而且如果两种手势被添加到同一个 view 时，会产生一个警告。

5. 识别轻扫手势

轻扫手势使用 UISwipeGestureRecognizer 类，可以指定只侦测以下方向的轻扫：

- ❑ UISwipeGestureRecognizerDirectionRight（向右）
- ❑ UISwipeGestureRecognizerDirectionLeft（向左）
- ❑ UISwipeGestureRecognizerDirectionUp（向上）
- ❑ UswipeIGestureRecognizerDirectionDown（向下）

如果不指定 direction 属性，默认只侦测方向为右的轻扫。以下代码设置只侦测向上轻扫：

```
UISwipeGestureRecognizer *swipeRecognizer =
      [[UISwipeGestureRecognizer alloc]
      initWithTarget:self
      action:@selector(swipeDetected:)];
swipeRecognizer.direction = UISwipeGestureRecognizerDirectionUp;
[self.view addGestureRecognizer:swipeRecognizer];
[swipeRecognizer release];
```

6. 识别长按手势

长按手势使用 UILongPressGestureRecognizer 类。这个手势需要指定长按的时间、触摸的次数、点击的次数以及在触摸过程中是否允许移动。这些选项分别由 minimumPressDuration、numberOfTouchesRequired、numberOfTapsRequired 和 allowableMovement 属性指定。以下代码使识别器只侦测单指长按 3 秒以上的手势。allowableMovement 未指定，默认是允许 10 个像素的移动：

```
UILongPressGestureRecognizer *longPressRecognizer =
      [[UILongPressGestureRecognizer alloc]
      initWithTarget:self
```

```
     action:@selector(longPressDetected:)];
longPressRecognizer.minimumPressDuration=3;
longPressRecognizer.numberOfTouchesRequired = 1;
[self.view addGestureRecognizer:longPressRecognizer];
[longPressRecognizer release];
```

iOS 4 关于 UIGestureRecognizer 的基本概念我们已经介绍过了，接下来，我们将通过一个小示例程序演示各种 UIGestureRecognizer 子类的用法。代码位于光盘"source/第 12 章/Recognizers"目录下。在这个程序中，我们将创建和设置各种不同的手势识别器，然后用标签把每个侦测到的手势的具体信息显示出来。

打开 Xcode，创建一个 Single View Application 项目，命名为 Recognizers。打开 ViewController.xib，往 view 上拖一个 UILabel，如图 12-1 所示。

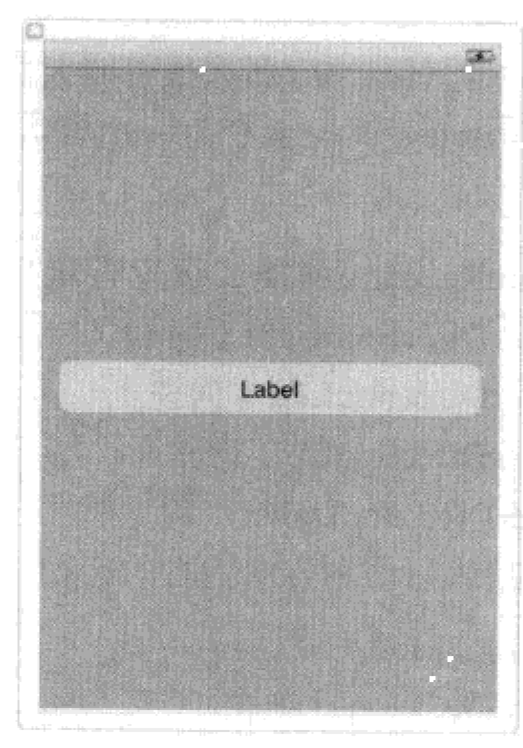


图 12-1　ViewController.xib

打开 Assistant Editor 视图，为 UILabel 创建一个出口 statusLabel。我们将在代码中使用这个 statusLabel 实时更新 UILabel 的文字显示。

需要在代码中使用 gesture recognizers 来识别轻击、轻扫、旋转和捏合。由于这些 recognizers 需要连接到 view 对象，因此创建它们的理想地方是 viewDidLoad 方法：

```
- (void)viewDidLoad {
    UITapGestureRecognizer *doubleTap =
         [[UITapGestureRecognizer alloc]
         initWithTarget:self
         action:@selector(tapDetected:)];
    doubleTap.numberOfTapsRequired = 2;
    [self.view addGestureRecognizer:doubleTap];
    [doubleTap release];
    UIPinchGestureRecognizer *pinchRecognizer = [[UIPinchGestureRecognizer alloc]
```

```
        initWithTarget:self
        action:@selector(pinchDetected:)];
    [self.view addGestureRecognizer:pinchRecognizer];
    [pinchRecognizer release];
    UIRotationGestureRecognizer *rotationRecognizer = [[UIRotationGesture Recognizer
        alloc]
        initWithTarget:self
        action:@selector(rotationDetected:)];
    [self.view addGestureRecognizer:rotationRecognizer];
    [rotationRecognizer release];
    UISwipeGestureRecognizer *swipeRecognizer = [[UISwipeGestureRecognizer alloc]
        initWithTarget:self
        action:@selector(swipeDetected:)];
    swipeRecognizer.direction = UISwipeGestureRecognizerDirectionRight;
    [self.view addGestureRecognizer:swipeRecognizer];
    [swipeRecognizer release];
    UILongPressGestureRecognizer *longPressRecognizer = [[UILongPressGesture Reco
            gnizer alloc]
            initWithTarget:self
            action:@selector(longPressDetected:)];
    longPressRecognizer.minimumPressDuration = 3;
    longPressRecognizer.numberOfTouchesRequired = 1;
    [self.view addGestureRecognizer:longPressRecognizer];
    [longPressRecognizer release];
    [super viewDidLoad];
}
```

设置完 gesture recognizer 之后，就应当编写处理方法，处理方法将在相应手势侦测到之后被 recognizer 所调用。这些方法也放在了 ViewController.m 文件里，同时这些方法会根据相应手势的具体信息来刷新 label 上显示的文字。

```
-(IBAction)longPressDetected:(UIGestureRecognizer *)sender{
    statusLabel.text = @"Long Press";
}
- (IBAction)swipeDetected:(UIGestureRecognizer *)sender {
    statusLabel.text = @"Right Swipe";
}
- (IBAction)tapDetected:(UIGestureRecognizer *)sender {
    statusLabel.text = @"Double Tap";
}
- (IBAction)pinchDetected:(UIGestureRecognizer *)sender {
    CGFloat scale =  [(UIPinchGestureRecognizer *)sender scale];
    CGFloat velocity = [(UIPinchGestureRecognizer *)sender velocity];
    NSString *resultString = [[NSString alloc] initWithFormat:
        @"Pinch - scale = %f, velocity = %f",
        scale, velocity];
```

```
        statusLabel.text = resultString;
        [resultString release];
    }
    - (IBAction)rotationDetected:(UIGestureRecognizer *)sender {
        CGFloat radians = [(UIRotationGestureRecognizer *)sender rotation];
        CGFloat velocity = [(UIRotationGestureRecognizer *)sender velocity];
        NSString *resultString = [[NSString alloc] initWithFormat:
                  @"Rotation - Radians = %f, velocity = %f",
                  radians, velocity];
        statusLabel.text = resultString;
        [resultString release];
    }
```

最后，编译和运行程序。为了能够充分测试捏合和旋转手势，最好是在物理设备中运行程序（因为模拟器上无法模拟多点触摸），如图 12-2 所示。

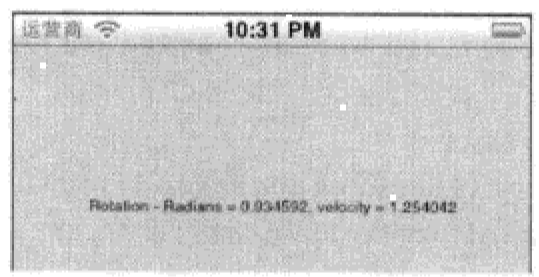


图 12-2　程序运行结果

提示：其实可以在模拟器中模拟 2 点触摸效果。按下 option 键，在模拟器上单击，可以模拟出 2 指触摸效果。通过这种方式，你可以用它模拟“捏合”手势。

12.3　实现图片的拖动及缩放

最常见的多点触摸应用是在对图片进行拖动和缩放。通常，我们用“平移”手势来进行拖动，而使用“捏合”手势来进行缩放。因为仅仅是把图片加载到 UIImageView 中是不够的，由于 iPhone 屏幕大小的限制，我们无法保证每张图片都刚好在屏幕上放下，如果放不下，就需要考虑图片移动的问题，这样用户可以通过“拖动”图片来查看图片的不同部分（用“平移”手势）。此外，还可以考虑用“捏合”手势来使图片放大或缩小。

接下来，我们将演示 SDK 4.0 中，如何利用 UIGestureRecognizer 来实现图片的拖动及缩放。实例代码位于光盘“source/第 12 章/GestureDemo”目录下。

1. 创建项目

新建 Xcode 项目，创建一个 Single View Application 项目。Xcode 将自动产生一系列文件。

2. 用 IB 创建界面和出口

打开 ViewController.xib，在其中拖入一个 UIImageView 对象，并加载一张图片（别忘记

将图片文件拷贝到项目文件夹中）。我们将用这张图片来演示平移和捏合手势的处理。

打开 Assistant Editor，为 UIImageView 创建一个出口 imageView。这样我们方便在代码中操纵它。

3. 识别平移和缩放

在 viewDidLoad 方法中，我们加入以下代码：

```
UIPinchGestureRecognizer *pinchRecognizer =[[UIPinchGestureRecognizer alloc]
      initWithTarget:self action: @selector(pinchDetected:)];
   [self.view addGestureRecognizer:pinchRecognizer];
   [pinchRecognizer release];

   UIPanGestureRecognizer* panRecognizer=[[UIPanGestureRecognizer alloc]
      initWithTarget:self action:@selector(panDetected:)];
   [self.view addGestureRecognizer:panRecognizer];
   [panRecognizer release];

   UITapGestureRecognizer* tapRecognizer=[[UITapGestureRecognizer alloc]
      initWithTarget:self action:@selector(tapDetected:)];
   tapRecognizer.numberOfTapsRequired=2;
   [self.view addGestureRecognizer:tapRecognizer];
   [tapRecognizer release];

   originFrame=imageView.frame;
```

注意，我们在 view 中用 addGestureRecognizer 加入了 3 个 Recognizer，一个 PinchRecognizer 用于识别捏合手势，一个 PanRecognizer 用于识别平移手势，还有一个 TapRecognizer 则用于识别双击——我们准备当用户连续触摸屏幕 2 下时，将图片还原为最初状态，取消所做的所有平移及缩放操作。

因此最后一句代码也记录了图片最初的大小：

```
originFrame=imageView.frame;
```

接下来是 3 个处理方法的实现。

首先是平移手势的处理：

```
-(void)panDetected:(UIPanGestureRecognizer*)sender{
   CGPoint center=imageView.center;
   CGPoint trans = [sender translationInView:imageView];
   [imageView setCenter:CGPointMake(center.x+trans.x,center.y+trans.y)];
   // 由于平移是一个连续手势，它会不断地累加 translation 的值，因此我们每次移动完 ImageView
      后把 translation 清零
   [sender setTranslation:CGPointZero inView:imageView];

}
```

平移手势的处理相当简单，UIPanGestureRecognizer 有一个 translationInView:方法，它返回一个 CGPoint，用于表示用户手指在平移过程中移动过的偏移量（x,y）。我们只需要参照这个偏移量将图片移动一定的距离就可以了。

但不幸的是，由于平移手势是一个连续手势，在用户移动手指的过程中实际上会不停地调用手势响应方法（处理方法），并且每一次这个偏移量会不断增加。因此如果你仅仅是简单地把图片位置移动相同的偏移量，你会发现图片移动的距离会比手指移动的距离要大得多，而且二者的误差会越来越大。

因此我们在方法的最后，在每次移动过图片位置后，使用 setTranslation:inView:方法将这个偏移量（translation）置零。

然后是捏合手势的处理：

```
-(void)pinchDetected:(UIPinchGestureRecognizer *)sender{
    scale =  [sender scale];
    CGSize size=imageView.frame.size;
    imageView.frame=CGRectMake(0, 0, size.width*scale, size.height*scale);
    [imageView setNeedsLayout];

}
```

UIPinchGestureRecognizer 拥有一个 scale 属性，这个属性表示了用户两指在捏合过程中距离发生的变化（用手指离开时两指间距除以两指最初触摸屏幕时的间距）。因此我们只要把图片的大小根据这个 scale 来进行缩放就可以了。

提示：这里的处理是直接改变 imageView 的 frame 大小。这样的处理未免过于简单。实际在很多场合下，我们未必希望得到一个 frame 会不断改变的 UIImageView（一般 UIImageView 的大小在一开始就会固定下来了）。这种情况下，你应该用 Quartz Core 绘图函数来进行缩放（比如 CGImageCreateWithImageInRect 函数）。

最后是连击的处理。连击的处理最为简单。由于在开始就设置了 TapGestureRecognizer 的 numberOfTapsRequired 属性：

```
tapRecognizer.numberOfTapsRequired=2;
```

所以在处理方法中实际上只针对了双击的情况：

```
-(void)tapDetected:(UITapGestureRecognizer*)sender{
    imageView.frame=originFrame;
}
```

我们把 imageView 的 frame 恢复到了原始的大小。

4. 运行程序

按⌘+R 运行程序，效果如图 12-3 所示。

现在你可以在 view 中通过平移来拖动图片，也可以通过捏合手势来缩放图片。如果你双

击屏幕，图片恢复为原始大小。

图 12-3　程序运行效果

12.4　本章小结

多点触摸和手势是新一代 iOS 设备中最重要的用户界面交互特性。在 iOS 4 以前，要实现手势的识别是比较困难的，程序员负责所有的事情：屏幕坐标和设备空间坐标的转换、实时检测用户手指移动的距离、速度、方向，以及一系列复杂的几何运算。到了 iOS4 以后，苹果终于在 SDK 中提供了对多点触摸和手势的支持，使得程序员从这一系列繁琐的事务中解脱出来。对程序员来说，手势的识别因此变得简单。

本章，我们介绍了 iOS 4 SDK 提供的对多点触摸和手势的支持，主要是 UIGestureRecognizer 类族：UITapGestureRecognizer、UIPinchGestureRecognizer、UIPanGestureRecognizer、UISwipeGestureRecognizer、UIRotationGestureRecognizer、UILongPressGestureRecognizer。

然后通过一个 Demo 程序，详细地演示了如何在一个图片预览程序中利用 UIGesture Recognizer 进行手势的识别，从而使用户通过手势来实现对图片的拖动和缩放。

第13章　本　地　化

在经济全球化背景下，软件开发所面向的客户越来越多，他们可能来自于世界各地。如果你所开发的应用程序面临到以下问题，那么你需要考虑本地化：

- ❑ 你在面向国际用户开发；
- ❑ 你在面对企业用户进行开发，该企业正在实施或已经实施了全球化战略；
- ❑ 企业对外宣传需要；
- ❑ 民族文化差异；
- ❑ 用户提出了本地化需求。

如果你已经计划将应用程序本地化，那么切记要在设计和开发阶段做许多事情。在项目早期做一些计划和准备是有必要的，这会使接下来的事情变得容易。当然，SDK 和 iOS 内置的本地化支持，会使开发适应于不同文化区域的应用程序变得简单。

本章将向你介绍 iPhone 应用程序中的国际化（本地化）支持。在 iPhone 应用程序中，可以本地化的资源包括：文字、图片、.xib 文件和应用程序名称等。本章对这些技术进行了一一介绍。

13.1　iPhone 的本地化支持

iPhone 支持本地化。所谓的本地化是指应用程序能够根据用户的语言或地区（用户在操作系统中选择的国家代码），自动载入相应语种的语言文字、图像及其他资源。简单地说，本地化就是使用一个应用程序束支持多种语言——即应用程序的多语言版本。这样做的好处是显而易见的，你不需要为每种语言都重新编写代码。

为了支持本地化，操作系统必须记载用户使用的语言。在 iOS 中，这个设置在 settings（设置程序）中进行。

13.1.1　国家代码和语言代码

iPhone 的本地化框架使用本地化文件夹来支持多语言的支持。在一个支持本地化的应用程序中，每一种语言都会有一个对应的以.lproj 为扩展名的子目录，在这个子目录中存放这种语言的应用程序资源。本地化文件夹的命名有一定的规则。一般是采用 ISO 语言代码+ISO 国家（地区）代码的方式进行命名。例如 fr_FR.lproj。fr 是语言代码，FR 是国家代码。

ISO 语言代码是 ISO639 标准的一部分，可以在网址 http://www.loc.gov/standards/iso639-2/iso639jac.html 找到它。在 ISO639 中，定义了两种不同的语言代码：alpha-2 和 alpha-3。分别

对应 2 位字母和 3 位字母的编码。以法语为例：

```
Alpha-3: fre/fra
Alpha-2: fr
全称（英语）: French
全称（法语）: français
```

ISO 国家（地区）代码是 ISO3166 标准的一部分，你可以到网址 http://www.iso.org/iso/iso-3166-1_decoding_table 找到它。在 ISO3166-1 中，定义了三种不同的编码：alpha-2、alpha-3 和 numeric-3。顾名思义，这三种编码的格式分别为：2 位字母、3 位字母、3 位数字。以南非共和国为例：

```
Alpha-2: ZA
Alpha-3: ZAF
Numeric-3: 710
简称（英语）: SOUTH AFRICA
全称（英语）: Republic of South Africa
```

13.1.2 本地化文件夹的匹配

iPhone 本地化搜索本地化文件夹时，采用一种复杂的机制。以法语为例，情况要分为几种。国家代码 FR 是唯一的 ，但语言代码却可能有几种情况：fre、fra 或者是 fr。fre 是法语 Alpha-3 码的英语拼写，fra 是 Alpha-3 码的本国语拼写，fr 是 Alpha-2 码。那么你的本地化文件夹可能会叫做 fre_FR.lproj、也会叫 fra_FR.lproj 或者叫做 fr_FR.lproj。应用程序会按照如下顺序查找匹配的本地化文件夹：

```
fr_FR.lproj -> fre_FR.lproj -> fra_FR.lproj
```

甚至你可以不使用国家代码，那么应用程序在按如上顺序搜索完之后，会接着进行如下顺序的搜索，直到找到一个匹配的本地化文件夹：

```
fr.lproj -> fre.lproj -> fra.lproj -> French.lproj
```

13.2 本地化应用程序

13.2.1 使用 NSLocaledString 本地化字符串

字符串总是首先要进行本地化的部分。对于需要进行本地化的字符串，我们首先是将它加入到字符串文件中。当然这个过程可以通过工具进行，没有必要手工创建字符串文件。我们只需要在程序中把每个需要本地化的字符串使用本地化函数 NSLocaledString 进行声明即可。

例如，如果我们有一个字符串变量要进行本地化。这个字符串变量原本的声明和初始化语句是：

```
NSString* string=@"My name is steven.";
```

如果我们想对这个字符串进行本地化，则这句代码应修改为：

```
NSString* string=NSLocaledString(@"My name is steven.",
@"Used to introduce myself");
```

第 1 个参数是程序默认情况下（没有本地化）程序将使用的字符。第 2 个参数将在字符串文件中作为注释。这样将在字符串文件（即 Localizable.strings，后面将介绍如何生成这个文件）的最后添加一条类似这样的记录：

```
/* Used to introduce myself */
"string" = "My name is Steven." ;
```

接下来我们用一个 Xcode 项目来演示本地化字符串的过程。新建 Single View Application 项目，命名为 LocalizableDemo。默认情况下，LocalizableDemo 只支持一种语言英文。在项目 info 视图的 Localizations 栏，可以看到当前只支持一种语言——英文，如图 13-1 所示。

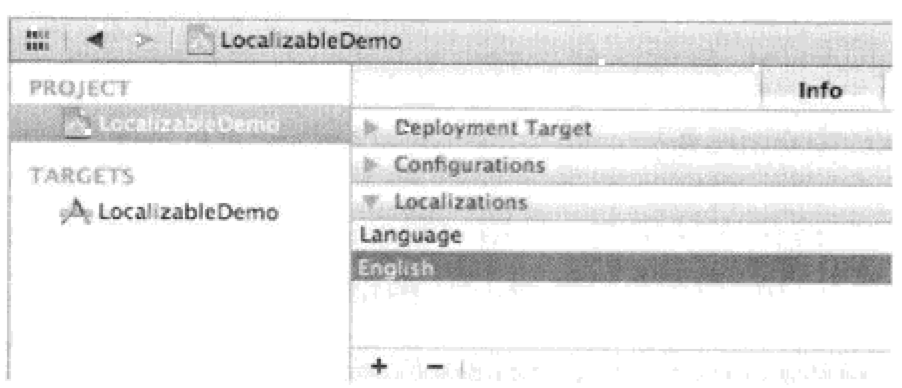


图 13-1　项目的本地化设置

同时，在 finder 中查看项目文件夹，会存在一个叫 en.lproj 的文件夹，如图 13-2 所示。

图 13-2　查看 en.lproj 中的内容

这是英文的本地化资源文件夹。文件夹中列出了两个文件，这两个文件会在本地化英文时使用。

接下来我们将增加中文的本地化。点击 Localizations 下面的“+”号按钮，选择“Chinese(zh_Hans)”，增加中文的本地化。同时，项目文件夹下会自动增加一个 zh_Hans.lproj 的文件夹，里面的内容同 en.lproj。

提示：“Chinese(zh_Hant)”是繁体中文。

本例中，zh_Hans 将作为中文的本地化文件夹名。从 Mac OSX 10.4 及 iOS 4 以后，可以不使用 ISO 639，而使用 rfc 的 BCP 47 标准作为语言代码。

BCP 47 规范：http://www.rfc-editor.org/rfc/bcp/bcp47.txt 。

与 ISO 639 不同，BCP 47 可以使用多个子标签（subtag）一起描述语言编码，其中每个子标签支持 3～8 个字符，子标签之间以“-”号连接。其中一种形式是“语言子标签-书写方式子标签”，共有 4 个：

- zh-Hant（中文-繁体）
- zh-Hans（中文-简体）
- sr-Cyrl（塞尔维亚-古斯拉夫文）
- sr-Latn（塞尔维亚-拉丁文）

其中，zh 表示中文，Hans 表示简体汉字。

接下来，在 ViewController.xib 中拖入一个 Label，为它新建出口 textLabel，连接上。

在 ViewController.m 的 ViewDidLoad 方法中，加入以下代码：

```
textLabel.text=NSLocalizedString(@"My name is steven.", @"Used to introduce
               myself");
```

在终端中运行命令：

```
genstrings -o ./ ./ *.m
```

genstrings 命令用于搜索指定源文件中（.c 文件和.m 文件）的 NSLocalizedString("key", comment) 或 CFCopyLocalizedString("key", comment)函数调用，并生成一个 strings，文件名为 Localizable.strings 。上述命令中的“-o”选项是固定的，表示输出（产生.strings 文件）；参数“./”表示所生成的.strings 文件将放在当前目录，参数“./*.m”则表示搜索当前文件夹下的所有.m 文件。

执行完上述命令，你会在 Finder 中发现项目目录下增加了一个 Localizable.strings 文件，文件内容如下：

```
/* Used to introduce myself */
"My name is steven." = "My name is steven.";
```

在 Xcode 中通过“Add Files to…”功能，将这个文件添加到项目中。

在 Project Navigator 窗口中选择 Localizable.strings，打开 File Inspector 面板，在 Localization 中点击“+”号按钮，添加两个语言 English 和 Chinese，如图 13-3 所示。

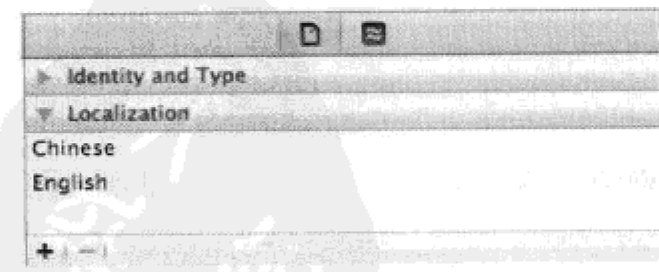


图 13-3　本地化 Localizable.strings

这会在 en.lproj 和 zh-Hans 两个目录下分别复制一份 Localizable.strings 文件的拷贝。你可以用 Finder 找到它们。

同时在 Project Navigator 窗口，你会发现 Localizable.strings 可以展开为两项，如图 13-4 所示。

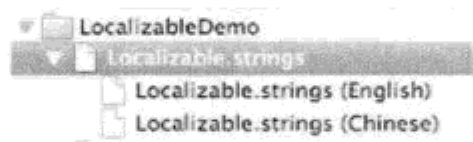


图 13-4　Localizable.strings 被区分为不同语言的版本

其中，Localizable.strings(English)为 en.lproj 文件夹内的拷贝，Localizable(Chinese)为 zh-Hans 文件夹内的拷贝。这样，我们就可以直接在 Xcode 中编辑它们。

我们可以将字符串文件 Localizable.strings(Chinese)中的内容编辑为对应的语言（简体中文），比如：

```
/* Used to introduce myself */
"My name is steven." = "我叫张三.";
```

注意："My name is steven." 不用翻译，它代表字符串在未本地化之前的样子。等号右边则为字符串经过本地化之后的内容。此处注意用于括住字符串的双引号和换行的分号不要写成中文的标点符号。

在模拟器中运行程序，Label 中显示的是英文。打开"设置"程序，在"通用 →多语言环境"中将语言设置为"简体中文"，如图 13-5 所示。

图 13-5　将语言设置为"简体中文"

然后再次运行程序，Label 中显示为中文，如图 13-6 所示。

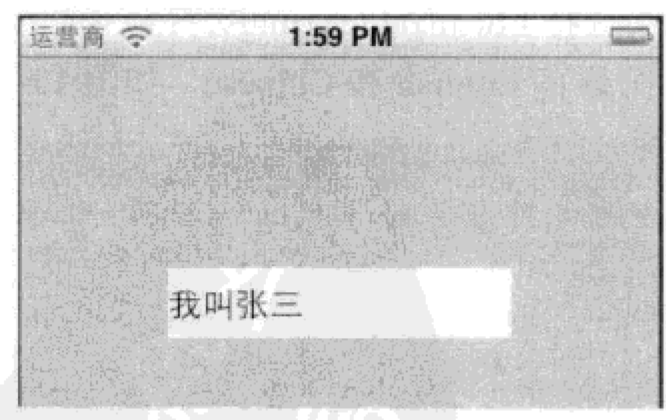


图 13-6　将语言设置为中文后，程序显示为中文内容

示例项目位于光盘的"source/第 12 章/LocalizableDemo"目录。

13.2.2 本地化图像

有时候需要本地化图片，因为图像中会有一些本地化的语言文字或图形，因此也需要进行相应的转化。在 Groups&Files 窗口中选择要本地化的图片，按 Command+ 1 打开 info 窗口，点击 Make File Localizable 按钮。这样，图片文件将被复制到 English.lprog 文件夹。返回 General 选项卡，点击 Add Localization 按钮，增加另一种语言的本地化图片。然后用图形编辑工具编辑或替换此图片文件，即可完成图片的本地化。

13.2.3 本地化 xib 文件

xib 文件中的过程和本地化图片一模一样。选择一个.xib 文件，然后打开其 info 窗口并点击 Make File Localizable 按钮。重新打开 info 窗口，并应用 Add Localization 按钮，即可生成针对某一语言的本地化 xib 文件。然后在 Interface Builder 中对该 xib 文件进行必要的编辑，即可完成 xib 文件的本地化。

13.2.4 本地化应用程序名称

作为一个完整应用程序的部分，应用程序名称也可以本地化。

本地化应用程序名称要稍微复杂一点，需要两个步骤才能完成：1）本地化 Info.plist 文件；2）本地化 InfoPlist.strings 文件。

首先，将 xxxx-Info.plist 文件名改为 Info.plist。打开这个 Info.plist 文件的 info 窗口，点击 Make File Localizable 和 Add Localization 按钮，将 Info.plist 也本地化。

在 Resources 目录下创建 InfoPlist.strings 文件，使用同样的方法也本地化。编辑各自语言版本的 InfoPlist.strings 文件，在其中加入：

```
CFBundleDisplayName = “本地化的应用程序名”；
```

13.3 示例

我们做一个小示例程序，来演示应用程序本地化的完整过程。这个示例程序项目被放在光盘“source/第 11 章/LocalizationDemo”。

新建 Window-based Application，将 flag.png（一幅英国国旗）拷贝到 Resources 文件组下。使用 Add→New File，在 Resources 下（注意，一定要放在 Resources 文件组下）新建一个 Localizable.strings 文件，文件类型选择 String File（图 13-7）。

注意：为了避免出现中文乱码问题，最好吧 Localizable.strings 文件修改为 UTF-16 编码。

打开 Localizable.strings 文件的 info 窗口，点击 Make File Localizable。切换回 General 栏，点击 Add Localization，输入 zh_CN，关闭 info 窗口。现在可以看到在 Localizable.strings 文件

下出现了两个条目 English 和 zh_CN（如图 13-8 所示）。

新建 View Controller，勾上“With XIB……”，输入名称 HelloViewController。

打开 HelloViewController.xib 文件，在 view 上添加两个控件：一个 Label，一个 UIImageView（如图 13-9 所示）。

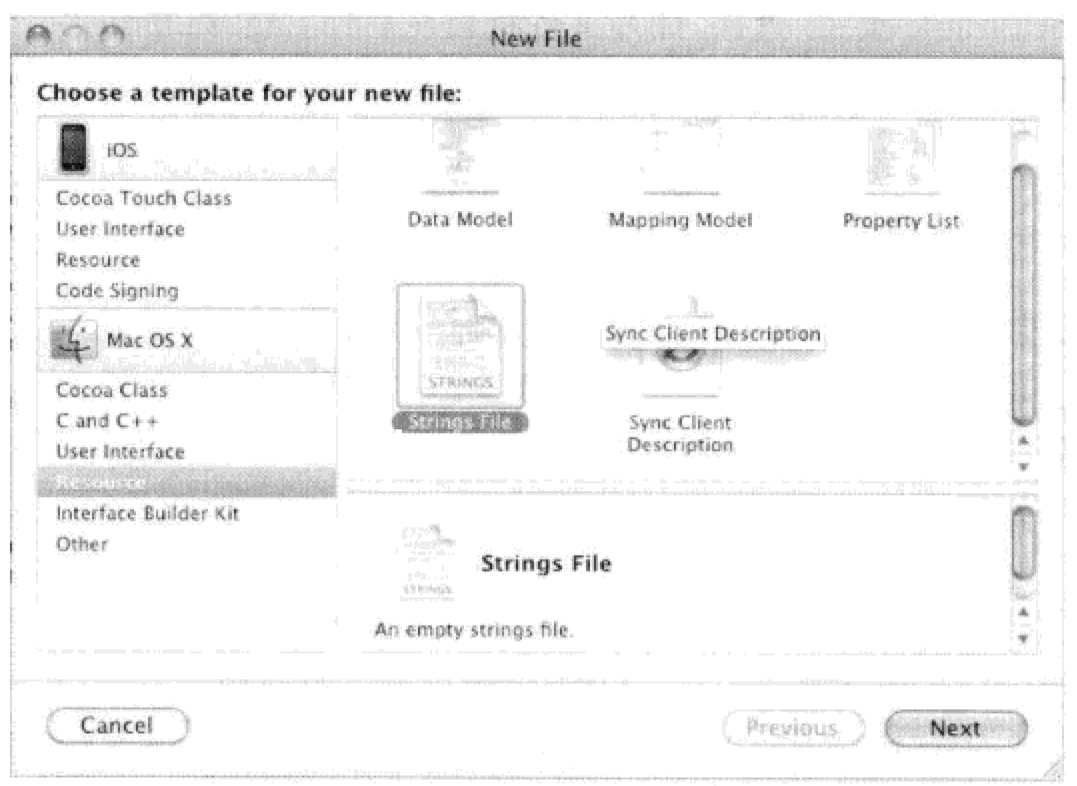


图 13-7　新建 Strings File 文件

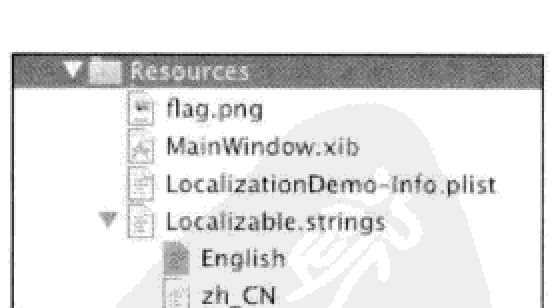


图 13-8　本地化字符串文件 Localizable.strings

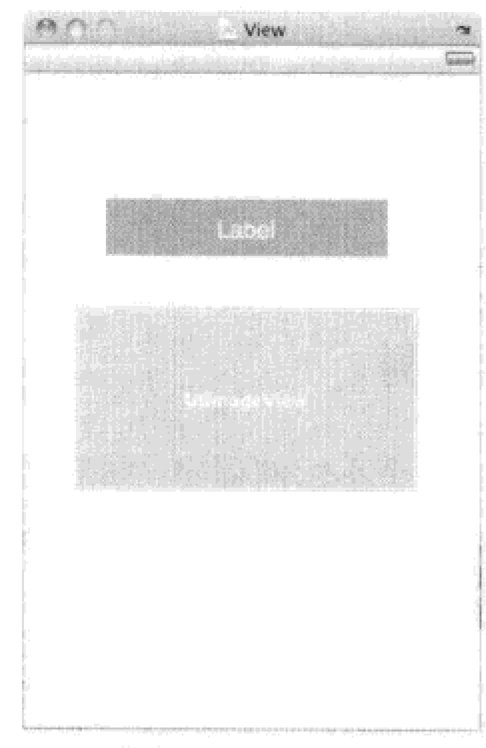


图 13-9　设计 HelloViewController.xib

Label 控件的 text 属性没有设置任何值，因为我们要对控件文本使用 NSLocaledfString()函数进行本地化，我们只能用代码进行赋值。

在 HelloViewController.h 中设计两个 IBOutlet，并在 Interface Builder 中将两个控件分别连

接到这两个 IBOutlet 中。

在 HelloViewController 的 viewDidLoad 方法中加入以下两句代码：

```
label.text=NSLocalizedString(@"I am from UK.",@"Used to introduce myself");
imageView.image=[UIImage imageNamed:@"flag.png"];
```

修改 LocalizationDemoAppDelegate 的 application: didFinishLaunchingWithOptions:方法代码如下：

```
HelloViewController *viewController=[[HelloViewController alloc]

initWithNibName:@"HelloViewController" bundle:nil];
    [self.window addSubview:viewController.view];
    [self.window makeKeyAndVisible];
    [viewController release];
    return YES;
```

运行程序，将出现如下界面（如图 13-10 所示）。

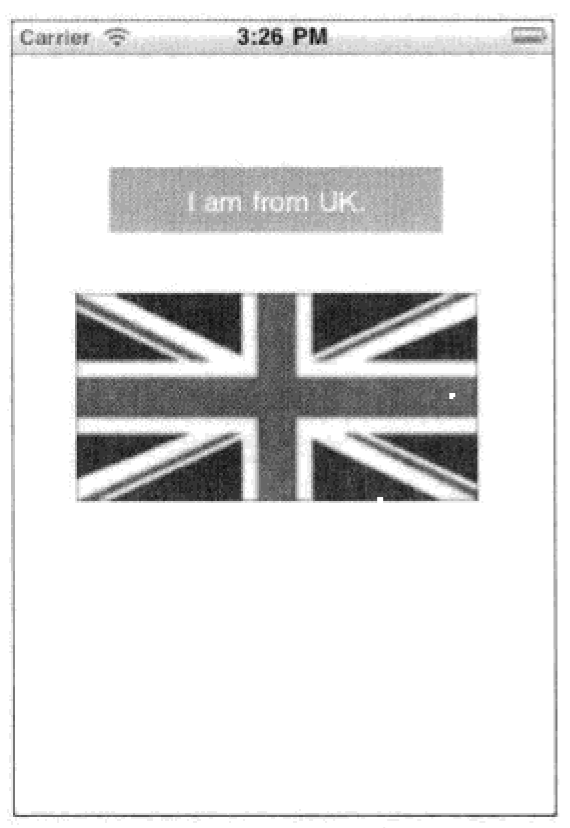


图 13-10 程序运行界面

接下来我们要对字符串“I am from UK.”进行本地化。打开终端程序，进入项目文件夹目录，运行命令：

```
genstrings -o English.lproj ./classes/*.m
genstrings -o zh_CN.lproj ./classes/*.m
```

点击 Groups & Files 窗格中的 Localizable.strings 文件下的 English 和 zh_CN，发现文件内容中增加了：

```
/* Used to introduce myself */
"I am from UK." = "I am from UK";
```

现在，是进行翻译的时候了。把 zh_CN.lproj 文件夹下字符串文件中的内容修改为：

```
/* Used to introduce myself */
"I am from UK." = "我来自中国。";
```

在设备中测试程序，因为模拟器中的地区和语言设置是无效的。如果设备中的地区设置为中国，语言设置为中文，则程序运行后的界面如图 13-11 所示。

图 13-11　文字已经本地化了

现在，我们来本地化图片 flag.png。让它在地区为中国时显示中国国旗。在 Xcode 中选择 flag.png 文件，然后打开 info 窗口，点击 Make File Localizable。返回 General 页面，添加语言 zh_CN，然后关闭 info 窗口。

现在，flag.png 也拥有两个语言的版本了（如图 13-12 所示）。

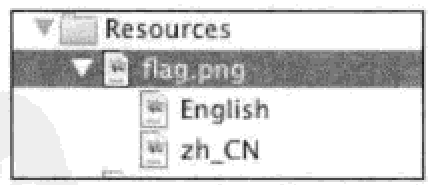


图 13-12　对 flag.png 进行本地化

但实际上两个版本都是来自同一个 flag.png 的拷贝，内容都是英国国旗。我们需要在 Finder 中把 zh_CN.lproj 目录中的 flag.png 替换成中国国旗（替换后，文件名还是 flag.png）。在 Xcode 中查看 zh_CN 条目，确保图片已被替换成中国国旗。

由于 iPhone 可能会缓存上一次运行的英国国旗图片，所以在再次运行程序之前，请删除

iPhone 上的应用程序。同时 Clean 项目（或者直接删除 Products 文件组下的.app 文件）。再次运行的结果可能是这样的（如图 13-13 所示）：

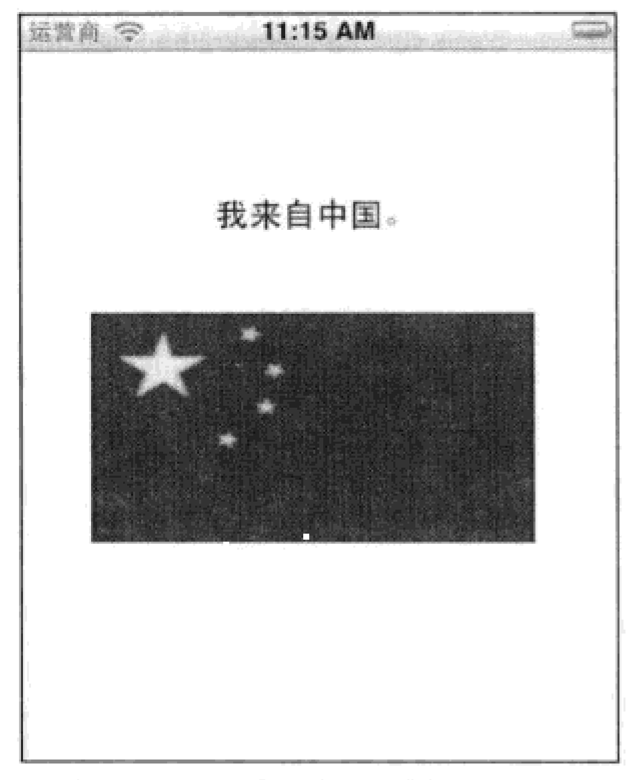


图 13-13　国旗图片也被本地化了

13.4　本章小结

本章介绍了如何利用 Xcode 内置的本地化支持对程序进行本地化。Xcode 的本地化支持会在应用程序束中为每种语言创建一个子目录，而每个目录实际上都是一个本地化项目，它们是采用特殊规则进行命名的——它们以 ISO 的语言代码和地区代码进行命名并以.lproj 为后缀。

每个本地化文件夹都包含一个翻译为此种语言的应用程序资源子集。这些资源包括：字符串、图片、xib 文件等。如本章中所介绍，这些文件夹中的资源，都可以使用 Xcode 提供的本地化支持进行本地化。

第 14 章　iOS 多线程和并行编程

随着 iPad 2 和 iPhone 4S 的推出，标志着以 A5 系列 CPU 为代表的多内核 CPU 开始成为主流。多线程和并行编程通过在程序中创建新的线程，使程序在较短时间内完成更多工作，以此来充分挖掘多核 CPU 的性能优势。

本章介绍 iOS 多线程和并行编程。前者是老的并行编程方案，后者是 iOS 4 以后增加的异步编程模型，苹果准备用新的并行编程取代传统的线程模型。本章将对二者进行介绍。

首先介绍多线程，这将包括以下主题：NSThread、RunLoop、锁。

尤其是 RunLoop，众多开发人员很少关注它。当然在多线程中，锁同步也是一个老生常谈的问题。

iOS 4 的并行编程，其实包括了：Operation Queue 和 Grand Central Dispatch（GCD，系统线程管理）。Operation Queue 此前章节中已经讨论过一些，本章简单回顾了这些内容。然后着重介绍 GCD，包括：Dispatch Queue 和 Dispatch Source。

最后，介绍了 iOS 4 的后台任务。

14.1　多线程

在讨论多线程之前，我们首先要明白线程的概念。一个线程就是一个数据结构，它负责管理一段可执行的代码片断。在像 O-C 这类面向对象的语言来说，数据结构就是对象，于是线程通常被封装为类。在 O-C 中它是 NSThread 类，在 Java 中它是 Thread 类。

对于单线程的应用来说，它执行代码的线路只有一条，就是 main()函数。从 main()函数第一个语句，执行到最后一句，这个应用程序就结束了。我们写的大部分应用程序都是单线程。

而多线程应用则不同，它可以在代码的某个地方开始分叉。这些分叉的代码跟分支语句的那种分叉不同。分支语句虽然从逻辑上把程序的执行线路分开来，但是每一次执行，CPU 只可能计算其中一条分支，也就是运行到最终只能得到一条分支的计算结果。多线程的这种分叉在开始分叉的地方，CPU 可以根据分支的数目进行“切分”（当然不是真的切分，只是便于理解这样说）。因为现在 CPU 被分成了几个，每个分支都能分到一个“单独”的 CPU（或处理单元），于是每个分支都可以得到执行，而且是互不干扰、同时执行。这样到程序结束，可得到每个分支的运行结果。这就是“并行编程”的由来。

提示：不是每台计算机都有多个 CPU，也不是所有 CPU 都是多核的。在许多情况下，每个线程分到的有可能只是 CPU 时间片而已，所有线程轮流使用 CPU 的一小段时间，看起来这些线程就像是“同步”执行。

多线程应用程序与单线程应用程序相比，具有以下明显的优势：

1. 提高程序响应速度，增强用户体验

对于单线程应用程序，主线程需要独立完成所有的事情。它必须对事件作出响应，更新UI，并执行所有的计算功能。如果程序需要进行一个长时间任务，那么当代码忙于进行这个繁重的计算时，你的程序就会停止响应用户事件和刷新窗口。如果长时间维持这样的情况，用户就会认为程序崩溃了，并试图强制退出。而如果使用多线程，就可以把计算任务放到一个单独的线程里，而你的主线程就可以及时响应用户行为和刷新 UI。

2. 提升计算性能

现在的电脑已经普及多核 CPU 了。然而你的应用程序是否充分利用了多核 CPU 的性能优势呢？如果是单线程的应用程序，那么程序运行在多核 CPU 上跟运行在单核 CPU 上是相同的，因为单线程没有采用并行编程。如果你打开性能监视器，你很容易就会发现，单线程应用程序运行时，只有一个 CPU 内核表现出较高的占用率，而其他的内核 CPU 占用率一直处于较低水平。而多线程应用程序不同，它一开始就是为多核 CPU 设计的。因此在多核 CPU 的平台中，多线程应用程序会有更好的表现。

当然，多线程带来的并不完全都是好处，同时也带来了许多潜在的问题，比如：增加了编程的复杂性，如线程间通信和同步。

接下来，我们讨论 iOS 中多线程的实现。

14.1.1 NSThread

NSThread 是 Cocoa 的线程类，负责线程的创建、执行和管理。

1. 创建线程

通过 NSThread 创建线程有两种方式：

（1）类方法 detachNewThreadSelector:toTarget:withObject:

这种方法创建一个匿名的 NSThread 对象，然后自动启动线程。例如：

```
[NSThread detachNewThreadSelector:@selector(myThreadMainMethod:) toTarget:self
        withObject:nil];
```

由于这种方法没有显式的 NSThread 对象可用，因此线程一旦启动，你无法对线程进行任何操作，比如设置线程优先级、取消线程等。

（2）用 init 或 initWithTarget:selector:object:方法显式地创建一个 NSThread 对象

这种方法创建一个 NSThread 实例，然后你再手动启动它（调用 start 方法）。这种方法创建线程需要显式地调用 start 方法，否则线程不会真正被创建。

```
NSThread* myThread = [[NSThread alloc] initWithTarget:self
selector:@selector(myThreadMainMethod:)
object:nil];
[myThread start];
```

无论哪种方式创建线程，都需要事先指定一个 selector。该 selector 参数指定了线程的主方法，需要将线程入口写在这个方法里。

2. 线程操作

线程启动后，你可以用以下方法停止线程：

```
+ sleepUntilDate:
+ sleepForTimeInterval:
+ exit
- cancel
```

前三个方法是类方法，默认操作的是当前线程。这就意味着这三个方法的调用代码只能是放在线程本身的代码里，在线程外部你无法调用它们。

最后一个 cancel 方法是实例方法，这意味着你可以在线程外部通过对线程对象的引用来终止线程。同时，它并不是真正地终止线程，而是修改线程的状态为 cancelled。同时，通过线程的 isCancelled 方法，可以知道线程状态是否为 cancelled。因此在线程代码中，你应该主动调用 isCancelled 方法来判断线程状态，并根据返回的 BOOL 值决定是否终止线程的执行。

在线程启动之前，你可以设置线程的优先级：

```
- (void)setThreadPriority:(double)priority
```

浮点参数 priority 指明要设置的优先级，取值范围从 0.0 到 1.0，1.0 为最高优先级。

3. 同步

同步是多线程编程中永恒不变的主题。每个线程都会和其他线程进行通信（起码要和主线程通信），这些通信的具体表现是，它们会修改其他线程中（例如主线程）的变量或状态。同时，其他线程也会修改这些状态。 如果两个线程试图同时修改同一个状态，一个线程有可能覆盖另一个线程的修改，导致该状态的结果是不可预料的。因此我们需要对线程间共享的数据结构进行保护，以确定在一个线程没有使用完该数据结构之前，其他线程不能操作该数据结构。这就是同步。

有不只一种线程同步技术。比如锁和 Conditions。下面，我们将介绍两种锁：NSLock 和 @synchronized 。

Cocoa 类 NSLock 实现了一种简单锁——“互斥盒”，它也实现了 NSLocking 协议，即 lock 和 unlock 方法。

以下语句创建一个 NSLock 对象：

```
NSLock *Lock = [[NSLock alloc] init];
```

然后你可以通过 NSLock 对象来使用锁。将需要加锁的代码用 lock 和 unlcok 方法调用包裹起来：

```
[Lock lock];
...// 某些多线程操作代码
```

```
[Lock unlock];
```

此外，可以用 tryLock 方法替代 lock 方法。tryLock 向操作系统请求一把锁，但会返回一个 BOOL 值，以表示是否请求到了可用的锁——如果锁不可用，tryLock 将返回 NO。我们可以通过判断 tryLock 方法返回值，以决定是否执行某些代码，比如刷新 UI 的显示：

```
if ([theLock tryLock]) {
    ... // 执行某些 UI 刷新动作
    [theLock unlock];
}
```

@synchronized 锁也是一种"互斥锁"，但使用它不需要事先声明和创建任何互斥体或锁对象，语法上更加简单：

```
@synchronized(anObj)
{
     // 任何需要保护的多线程代码
}
```

@synchronized 带一个参数（anObj），anObj 的类型是 id（任意指针类型），用于区别不同的锁。如果在两个线程中调用@synchronized 时，传递了同一个 anObj 参数，那么它们将使用同一把锁。这意味着，如果前面的线程没有执行完锁里的代码，另一个线程将被阻塞，不能再执行锁内的代码。一直到前面的线程退出锁，后面的线程才能进入锁。

14.1.2　RunLoop

RunLoop，从字面上说，可以翻译成"运行回路"或"运行循环"，我们可以把它看成是一种特殊的循环结构——就像我们知道的 for 或者 while 循环语句，其实 NSRunLoop 就是一种类似的循环，只不过它比 for/while 要复杂得多。

Foundation 框架提供了 NSRunLoop 类。在 CoreFoundation 中，也有对应的类 CFRunLoop。

说到这里，我们不得不先理解 RunLoop"源"的概念。

1. Input Source 和 Timer Source

RunLoop 是一种有源循环，它由各种"源"进行驱动。类似于 for / while 循环接受循环变量 / 条件表达式作为输入参数，RunLoop 接收"源"所发出的事件作为输入。RunLoop 接收两种源发生的事件："输入源"Input Source 和"定时器源"Timer Source。

"输入源"发送异步事件，比如来自于窗口系统的键盘鼠标事件，以及来自于 NSPort 和 NSConnection 对象的事件。

"定时器源"顾名思义，发送系统时钟事件（同步事件）。

2. 当前 RunLoop

每一个 NSThread 对象（包括主线程）都自带一个创建线程时自动创建的 RunLoop 对

象——你可以用 NSRunLoop 的类方法 currentRunLoop（或者 CFRunLoopGetCurrent 函数）获得这个“当前 RunLoop”。

而且，Cocoa 不允许显式地创建一个 RunLoop 对象。你唯一可以访问一个 RunLoop 实例的方式，就是调用 currentRunLoop 方法或 CFRunLoopGetCurrent 函数。

3. RunLoop 的使用

下面我们演示 RunLoop 的使用。在这个例子中，RunLoop 使用的源是“定时器源”（Timer Source）。

打开示例项目（位于关盘“source/第 14 章/TimerRun LoopTest”目录）。运行程序，点击 ViewController 上的 Run 按钮，按钮标题变为“Stop”，RunLoop 开始启动，并每隔 2 秒在下方的 UITextView 中刷新一次输出，一直到你再次按下标题为“Stop”的按钮，如图 14-1 所示。

图 14-1　示例程序 TimerRunLoopTest

按钮的 Action 方法代码如下：

```
- (IBAction)runAction:(id)sender {
    if ([@"Run" isEqualToString:btn.titleLabel.text]) {
        [btn setTitle:@"Stop" forState:UIControlStateNormal];
        [NSThread detachNewThreadSelector:@selector(myThreadMainMethod) toTarget:
                self withObject:nil];
    }else{
        textView.text=nil;
        [btn setTitle:@"Run" forState:UIControlStateNormal];
        if ( mRunLoopRef )
            CFRunLoopStop( mRunLoopRef );
    }
}
```

我们判断按钮的标题，如果标题为“Run”，则启动新线程（在线程的主方法中，我们会向 RunLoop 中添加定时器源）。如果按钮标题为“Stop”，则停止 RunLoop。

在子线程的主方法中，我们使用 CFRunLoopGetCurrent 函数去获得当前 RunLoop，然后在当前 RunLoop 中添加了一个定时器源。整个子线程主方法（入口方法）的代码如下所示：

```
-(void)myThreadMainMethod{
    CFRunLoopTimerRef timer;
    CFRunLoopTimerContext timer_context={0, self, NULL, NULL, NULL};
```

```
    timer = CFRunLoopTimerCreate(NULL, CFAbsoluteTimeGetCurrent(), 2, 0, 0, _timer,
            &timer_context);
    mRunLoopRef=CFRunLoopGetCurrent();
    CFRunLoopAddTimer( mRunLoopRef, timer, kCFRunLoopCommonModes);
CFRunLoopRun();
}
```

注意：我们不能把这些代码写在主线程里，这会导致主线程阻塞。主线程负责窗口及 UI 的绘制，不应该把大计算量的工作写在主线程里。这就是为什么我们创建了一个新的子线程的原因。

创建定时器源时，最主要的工作，是指定一个 RunLoopTimerContext 作为初始化时的上下文。这个上下文是个结构体（有 5 个成员），其中 info 成员是 void*类型（即 id 类型），对于我们来说可以把一些有用的对象传递进去，比如说 self。self 相当有用，因为 self 便于我们在 C 函数中调用成员方法：

```
CFRunLoopTimerContext timer_context;
bzero(&timer_context, sizeof(timer_context));
timer_context.info = self;
```

其他 4 个结构体成员对于我们来说都没有意义。因此上面 3 句其实也可以写成：

```
CFRunLoopTimerContext timer_context={0, self, NULL, NULL, NULL};
```

接下来就是用这个上下文构造 CFRunLoopTimerRef（定时器源）：

```
timer = CFRunLoopTimerCreate(NULL, CFAbsoluteTimeGetCurrent(), 2, 0, 0, _timer,
        &timer_context);
```

除了 NULL 和 0 之外的几个参数分别是：定时器启动时间、间隔秒数、定时执行函数、上下文指针。

其他参数我们都明白了，除了定时执行函数_timer（我们后面再讨论它）。接下来获取当前线程的 RunLoop：

```
mRunLoopRef=CFRunLoopGetCurrent();
```

然后将定时器源添加到 RunLoop 中：

```
CFRunLoopAddTimer( mRunLoopRef, timer, kCFRunLoopCommonModes);
```

这一句启动 RunLoop：

```
CFRunLoopRun();
```

接下来实现定时器执行函数_timer。_timer 是一个 CFRunLoopTimerCallBack 结构，它拥有两个参数，一个是 CFRunLoopTimerRef timer，另一个是 void *info。

info 参数实际上就是我们在 RunLoopTimerContext 中指定的 self，在这里派上了用场，我

们将会在这里调用 self 的 addLog:方法。因为 addLog:方法会对 UI 进行更新，根据 UKit 的规定，对 UI 进行刷新应当在主线程中进行，所以我们用 performSelectorOnMainThread 来强制在主线程执行 addLog:方法。因此，_timer 函数实现为：

```
void _timer(CFRunLoopTimerRef timer __unused, void *info){
    id obj=(id)info;
    [obj performSelectorOnMainThread:@selector(addLog:) withObject:[NSDate date]
        waitUntilDone:NO];
}
```

而 addLog 只是在 UITextView 中不断打印出当前日期：

```
-(void)addLog:(NSDate*)date{
    NSDateFormatter* df=[[[NSDateFormatter alloc]init]autorelease];
    [df setDateFormat:@"yyyy-MM-dd hh:mm:ss"];

        textView.text=[NSString stringWithFormat:@"%@\n%@",textView.text,[df
                stringFromDate:date]];

}
```

14.2 并行编程

所谓“并行”，是指在同一时间内同时执行多个任务。在以前，这往往意味着“多线程”。但随着多核 CPU 的盛行，以及在同一 CPU 内集成内核数的不断增加，新的“并行编程”机制需要为此做出优化。

使用传统的多线程编程，程序员很难根据硬件特性来动态指定应用程序所需的线程数，线程一旦生成，线程数就已固定，很难动态改变。

相对于直接创建线程，并行编程采用异步方式执行并行任务。这样，应用程序只需定义需要并行执行的任务，而将线程的管理完全交给了系统。程序员不需要操心线程，应用程序的伸缩性和编码效率都得到了提高。

iOS 并行编程技术分为两类：Operation Queue 和 GCD（Grand Central Disptach）。GCD 又包含 Dispatch Queue 和 Dispatch Source。

Operation Queue 是一个容器，用于放入多个 NSOperation。NSOperation 则定义了一个需要并行执行的任务（线程）。直接将 NSOperation 加入到 Queue 中，所有 NSOperationQueue 都会在单独的线程中运行。在第 7 章“网络”中我们已经做过一些介绍。这里就不再重复。

下面主要是介绍 GCD。

14.2.1 Dispatch Queue

同 Operation Queue 一样，Dispatch Queue 也是多个并行任务的容器。不同的是 Dispatch Queue 放入的不是 NSOperation，而是任务（块或者函数）。

Dispatch Queue 采用队列式的 FIFO（先进先出）结构。也就是说，任务启动的顺序和你加入它的顺序是一致的。

GCD 提供了三种 Dispatch Queue：串行队列、并行队列、主序列。

1. 串行队列

这种 Dispatch Queue 常用于某个资源的同步访问。串行队列每次只执行 1 个任务，你可以用串行队列保证你的任务按指定顺序执行。可以创建任意多的串行队列，每个队列都是和其他队列同步执行的。也就是说，如果你创建了 4 个串行队列 ，每个队列一次只执行一个任务，那么总共有 4 个任务会被同步执行（每个队列一个任务）。

创建串行队列，使用 dispatch_queue_create 函数，如下所示：

```
dispatch_queue_t queue = dispatch_queue_create("com.example.MyQueue", NULL);
```

第 1 个参数指定队列的名称，第 2 个参数保留，当前未使用。

对于任何自定义队列，系统都会自动创建一个串行队列并绑定到应用程序主线程。

2. 并行队列

并行队列（全局队列中的一种）同步执行 1 个或多个任务，只不过任务仍然是按 FIFO 顺序启动的。在任何时间点上，真正执行的任务数是动态的，取决于系统运行状态，比如 CPU 的可用内核数、已完成的任务数，以及其他串行队列的数目及优先级。并行队列不能手动创建，只能使用系统提供的 3 个并行队列。对于每个应用程序来说，这 3 个并行队列是全局的，它们只是在优先级上有所不同：优先级低、优先级高、默认。

由于是全局的，你根本不需要手动创建它们。要获得这三个并行队列，使用函数 dispatch_get_global_queue：

```
dispatch_queue_t aQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
```

第 1 个参数指定要获取哪种并行队列：DISPATCH_QUEUE_PRIORITY_DEFAULT、DISPATCH_QUEUE_PRIORITY_HIGH 和 DISPATCH_QUEUE_PRIORITY_LOW。第 2 个参数保留，目前未使用。

3. 主队列

另一种全局队列就是主队列，它是在主线程中执行的串行队列。这种队列通过在主线程 RunLoop 中插入 RunLoop 源来执行。

跟并行队列一样，主队列不需要你手动创建，使用函数 dispatch_get_main_queue 去获取主队列。

```
dispatch_get_main_queue();
```

14.2.2　将任务加入 Dispatch Queue

GDC 对块的支持极其丰富。可以用 dispatch_async 函数（异步执行）或 dispatch_sync 函

数（同步执行）将块加入到 Dispatch Queue。要将函数加入到 Dispatch Queue，则使用 dispatch_async_f 或 dispatch_sync_f。

```
void average_async(int *data, size_t len,
dispatch_queue_t queue, void (^block)(int))
{
    dispatch_retain(queue);
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    int avg = average(data, len);
    dispatch_async(queue, ^{ block(avg);});
    dispatch_release(queue);
    });
}
```

上述代码演示了如何用 dispatch_async 函数提交一个块。同时在块的最后将块计算结果提交到另一个队列的块中。以此来实现传统的异步编程中回调函数的效果。

向队列提交任务时，可以反复执行某个任务（指定执行次数）。例如我们可以用如下代码来代替 for 循环：

```
dispatch_apply(count, dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
        0), ^(size_t i) {
        printf("%u\n",i);
});
```

dispatch_apply 函数（或 dispatch_apply_f 函数）可以多次执行同一个任务，一直到指定的循环次数完成。它有 3 个参数：

- 第 1 个参数——指定循环次数。
- 第 2 个参数——指定块（或函数）将被提交到哪个目标队列执行。
- 第 3 个参数——指定任务块（或函数）。

14.2.3 Dispatch 源

Dispatch 源是一种数据结构，允许你通过它来处理低级别的系统事件。GCD 支持以下种类的 Dispatch Source：

- 定时器源（Timer Dispatch Source）——不解释，跟 RunLoop 中的定时器源类似。
- 信号源（Signal Dispatch Source）——Unix 信号产生时发出通知。Unix 信号是一种软件中断，类似于 Windows 中的事件。更详细的信息，请参考 Unix 信号编程。
- 描述符源（Descriptor Dispatch Source）——通知各种文件操作和 Socket 操作事件。如：有数据可读、是否可写入数据、元信息被改变、文件系统中的文件被删除、移动或重命名。
- 进程源（Process Dispatch Source）——用于通知各种进程相关事件，如：进程退出、进程调用了 fork（创建新进程）或 exec（在当前进程执行）、有信号（signal）送达该

PDG

进程。

- Mach 端口源（Mach Port Dispatch Source）——用于通知各种 Mach 相关事件。Mach 系统中，使用“端口”（port）来进行进程间通讯。“端口”（port）是任务（即进程）间进行通信的一组受保护的消息队列。任务间发送和接收数据都通过 port 进行。
- 自定义源（Custom Dispatch Source）——正如其名。自定义源允许你自己定义，并触发事件。

接下来我们介绍如何使用 Dispatch 源。Dispatch 源的使用一般包含以下步骤：

1）使用 dispatch_source_create 函数创建 dispatch 源。

2）设置 dispatch 源的事件处理代码（块或函数），如果是定时器源，用 dispatch_source_set_time 函数设置时钟信息。

3）设置 dispatch 源的取消代码（可选的）。

4）调用 dispatch_resume 函数开始接收事件。

位于光盘“source/dispatchSourceTest”目录的示例程序演示了一个描述符源的使用。打开 ViewController.m，其中 asyncLoadImage: 方法使用描述符源实现了一个大图片文件的异步加载。为求简便，本例中的大图片文件从应用程序束中加载。但是，你也可以将它修改为从网络加载。

描述符源使用 BSD Socket，因此我们首先需要获取一个文件描述符。NSFileHandle 是 Cocoa 对 file descriptor（文件描述符）的面向对象封装，可以利用 NSFileHandle 来获取文件描述符，这样大大简化了代码：

```
NSFileHandle *fh=[NSFileHandle fileHandleForReadingAtPath:filename];
int fd=fh.fileDescriptor;
```

当返回的文件描述符等于-1 时，表示发生 BSD Socket 错误，我们只能放弃处理：

```
if (fd == -1) return;
```

然后我们创建一个 Dispatch 源。dispatch_source_create 函数第一个参数指定要创建的源类型，在本例中我们使用的源类型为 DISPATCH_SOURCE_TYPE_READ，用于从文件描述符中读取数据；第二个参数则根据第一个参数而定，由于我们要创建的是描述符源，所以这里的参数就是指描述符；第三个参数也根据第一个参数而定，对于 DISPATCH_SOURCE_TYPE_READ 源，指定为 0；最后一个参数是 Dispatch Queue。

```
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,0);
dispatch_source_t readSource = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ,fd,
        0, queue);
```

然后我们用 dispatch_source_set_event_handler 函数设置源事件处理的块对象。该函数只有两个参数，第一个参数是要提交的 dispatch 源，第二个参数是一个块对象：

```
dispatch_source_set_event_handler(readSource, ^{
      size_t estimated = dispatch_source_get_data(readSource) ;
```

```
        char* buffer = (char*)malloc(estimated);
        if (buffer) {
            size_t actual = read(fd, buffer, sizeof(buffer));
            [data appendBytes:buffer length:actual];
            free(buffer);
            if (data.length%1024==0) {
                dispatch_async(dispatch_get_main_queue(), ^{
                    imageView.image=[UIImage imageWithData:data];
                });
            }
            if(abs(actual-estimated)==0){
                dispatch_async(dispatch_get_main_queue(), ^{
                    imageView.image=[UIImage imageWithData:data];
                });
            }
        }
```

在这个块的第一句，我们用 dispatch_source_get_data 函数计算描述符中还有多少字节未读取，将这个数字用于设置读取缓存的大小，然后试图从描述符中读取同样字节数的数据。我们把读到的数据保存到 NSMutableData 成员变量，然后每当收到 1024 个字节，就刷新 ImageView 的图片，这样会造成一种图片在逐渐加载的效果。刷新图片是 UI 操作，我们必需在主线程中进行，所以使用了主队列。

我们还需要判断什么时候到达文件末尾。因为描述符事件不会为这个特别通知你。 而在最后一次读取文件时，还必需把最后的这点数据加载到图片中（因为文件大小不可能刚好被 1024 整除，总是会有点余数，我们不能把这些数据丢掉）。

描述符源会对描述符的数据读取事件进行逐字节的监控，每当你从描述符中读取了一个字节，描述符中的数据就会被“消耗”掉一个字节，这样当文件最后一次读取时， estimated 应该为 1。但实际上，这个数字不一定是 1，它跟平台有关。描述符中每次读取多少个字节（不一定一次只读一个字节），依赖于“机器字长”，即一次整数运算所能处理的位数，比如对于主流的 32 位主机（例如 iPhone），就是 32 位字长（4 字节）。这样，文件最后一次读取完成时，应该是 4 个字节或小于 4 个字节（因为虽然 CPU 可以一次读取 4 字节，但文件大小本身不可能刚好能被 4 整除）。那么就很难通过 estimated 来判断文件是否结束。

那么，我们通过 actual 和 estimated 来共同判断文件是否结束。因为前者代表本次实际读取的字节，后者代表还剩多少字节未读。在读取过程中，剩余字节总是在不断减少，实际读取的字节数基本固定（一般都是 4，只有最后一次略有不同，可能是 4，3，2，1）。这样，二者的差异每经过一次总是在不断减少，一直到二者相等。比如最后一次只剩下 3 字节了，那么我们这次实际读取的也是 3 字节，那么说明文件已经结束了，下次不会再读。

接下来我们还为描述符源注册一个特殊事件——cancel 事件的处理块。这个事件在你调用 dispatch_source_cancel 函数时产生。这个函数可以异步取消一个 dispatch 源。你必须注册这个处理程序——哪怕你并没有在任何地方调用 dispatch_source_cancel 函数。

```
dispatch_source_set_cancel_handler(readSource, ^{
        [fh release];
});
```

最后一件事情就是调用 dispatch_resume 函数。这样导致 dispatch 源开始接收事件并调用处理块。

```
dispatch_resume(readSource);
```

运行程序，点击“载入”按钮，你会发现图片数据是从上到下逐渐加载完整的，如图 14-2 所示。

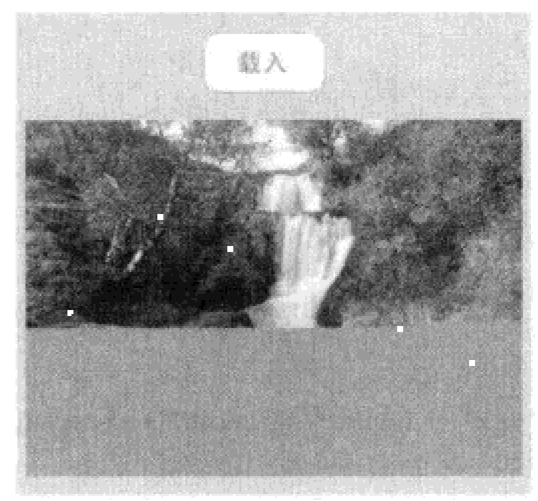


图 14-2　示例程序 dispatchSourceTest

14.3　后台任务

多任务从 iOS 4 起开始引入。由于 iOS 的限制，任何时候只允许一个应用程序在前台运行。当程序进入后台，程序很快就被操作系统挂起，同时程序中的代码不会再被执行。而通过多任务，允许你的应用在后台继续运行。多任务包括：后台执行、VoIP socket、后台任务。本节仅讨论“后台任务”，其他主题请参考苹果文档。

后台任务是 iOS 多任务中的重要内容，它和后台执行是截然不同的概念。后台执行在 iOS 多任务中专指一类能在后台执行代码的应用程序。如音乐播放器和 GPS 应用程序。 对于这些程序而言，退到后台并不意味着程序会就会被挂起。只有在首先停止了那些让它不被挂起的任务之后，它才会成为“可以被挂起”。继续以音乐播放器为例子，只有它停止播放后才会变得“可以被挂起”。而后台任务不同，任何应用程序都可以将一部分代码以“后台任务”的形式在后台执行。当然是限制有限的时间内。

光盘“source/第 13 章/BgTaskTest”目录下的示例项目演示了后台任务的使用。当应用程序执行后，按下 Home 键，程序进入后台。此时后台任务开始执行，并不断在控制台和文本框中输出新的日期时间，当你再次点击程序图标返回前台，文本框中的时间保持为最新时间，如图 14-3 所示。

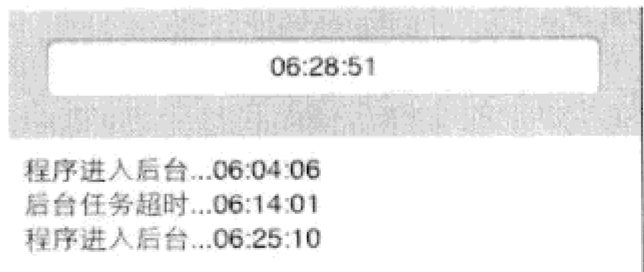


图 14-3 示例程序 BgTaskTest

所有的代码在 AppDelegate 的 applicationDidEnterBackground:方法中实现：

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    [self.viewController Log:@"程序进入后台..."];
    backgroundTask = [application beginBackgroundTaskWithExpirationHandler: ^{
      dispatch_async(dispatch_get_main_queue(), ^{
            if (backgroundTask != UIBackgroundTaskInvalid)
            {
                [self.viewController Log:@"后台任务超时..."];
                [application endBackgroundTask:backgroundTask];
                backgroundTask = UIBackgroundTaskInvalid;
            }
      });
    }];
    if (timer==NULL) {
        timer =dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER, 0, 0, dispatch_
                get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0));
        dispatch_source_set_timer(timer, DISPATCH_TIME_NOW, 1ull * NSEC_PER_SEC, 0);
        dispatch_source_set_event_handler(timer, ^{
            [self.viewController    performSelectorOnMainThread:@selector(setNow)
             withObject:nil waitUntilDone:NO];
        });
        dispatch_resume(timer);
    }
}
```

首先我们用 UIApplication 的 beginBackgroundTaskWithExpirationHandler: 方法（该方法是 iOS 4 以后新增的后台任务 API）声明了一个后台任务。该方法返回一个 UIBackgroundTaskIdentifier 作为后台任务的 ID（其实是一个无符号整型）。这个方法的第一个参数指定一个 Dispatch 队列，第二个参数则指定用于后台任务超时时回调的块对象。

后面的语句则定义了一个定时器源，设置了时钟事件发生时的回调块，然后提交到队列执行。

后台任务的执行时间是有限的。根据上面程序运行的结果，后台任务的最长超时时间约在 595 秒左右。这样就给了应用程序一个机会，在退出前台时能保存重要数据以保护程序的运行

状态。虽然这个时间很短，但在绝大多数情况下已经足够用了。

如果你需要“无限的”后台运行时间，请参考第 18 章 18.5 节“后台任务与无限后台任务”。

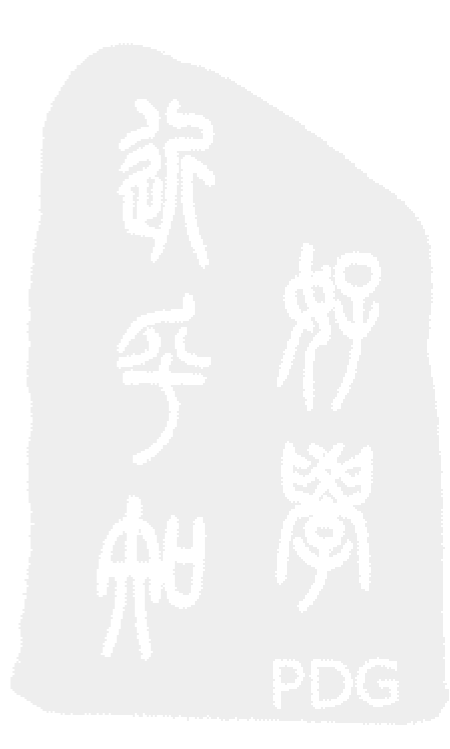
14.4　本章小结

本章介绍了 iOS 的多线程、并行编程和多任务。这些东西共同组成了 iOS 的异步编程模型。

多线程中，我们重点介绍了线程的概念，NSThread、RunLoop 和锁。通过传统的线程编程，我们可以将应用程序中的某些任务放在主线程以外执行，从而避免主线程由于这些任务而被阻塞。

对于并行编程，我们介绍了 GCD，包括 Dispatch 队列和 Dispatch 源。GCD 为程序员提供了系统级别的线程管理，使程序员更集中于任务本身的代码，也使得 OC 代码更加简洁和高效。我们可以直接向队列提交任务块，也可以使用 Dispatch 源——让任务在基于事件的基础上异步执行。

在多任务中，我们主要介绍了后台任务。后台任务使得程序在退出前台后仍然有小部分代码得到执行的机会，而不是简单地中断。



第 15 章　通知、本地通知和远程通知

本章介绍通知、本地通知和远程通知。

通知（Notification）是 Cocoa 的观察者模式的实现。每个应用程序都有一个唯一的共享 Notification Center 对象。通过这个 Notification Center，不同对象之间可以实现共享数据或进行线程间通信。在这个主题中讨论了 Observer 对象如何向 Notification Center 注册和注销，以及 Poster 对象如何发出通知。

本地通知（Local Notification）和通知（Notification）没有丝毫的关系，尽管都使用了“通知”一词。相反，本地通知和下面要介绍的远程通知一样，都是 iOS 后台进程的“唤醒”技术。随着 iOS 4.0 以后多任务技术的出现，苹果引入了“本地通知”的概念。我们可以使用本地通知在应用程序退出前台后仍然能响应一些事件。它可以用告警、徽章和声音 3 种方式通知用户去“唤醒”应用程序，从而使应用程序得以再次运行以处理某些特殊事件。在这个主题中，我们介绍了本地通知的创建和调度，以及通知到达后的处理。

远程通知是推送通知，即 APNs（Apple Push Notification services，苹果推送通知服务）。同本地通知类似，APNs 可用于当应用程序不在前台运行时，通知应用程序某些消息并唤醒应用程序。不同的是，本地通知由本地（同一个应用程序）发出，而远程通知由远程服务器（苹果 APNs 服务器）发出（类似手机短信）。通知、本地通知在同一进程内进行，而远程通知则是一种进程间通信的技术。

15.1　通知

通知（Notification）是 SDK 中一种特殊的事件驱动模型。当某些事件（一般是 UIView 标准事件之外的事件，比如网络连接事件或重力感应事件）发生时，操作系统以广播消息的方式通知事件的注册对象。这些注册对象就叫做观察者对象（Observer）。而负责广播消息的对象叫做通知中心 NSNotificationCenter，所有的通知都由这个中心发出，每个运行中的程序只有一个通知中心，所有对象共享。许多 Cocoa 类都会产生和发出通知。发送这些通知的对象有一个专门的名称，叫做“张贴者”（Poster）。比如 UITableView（张贴者）在用户选择了一行时会产生通知，这个通知会被发送到通知中心，再由通知中心转发给观察者。

实际上通知采用了观察者模式进行设计。在 Java 中也有同样的设计。NSNotification 对象本身非常简单，它只有 3 个成员方法：

```
- (NSString *)name;
- (id)object;
- (NSDictionary*) userInfo;
```

PDG

name 为返回通知的名称，object 为返回张贴者的引用，userInfo 是任何需要在通知中附加的自定义信息，一个对象可以把任何它想通知的信息放到 userInfo 里。

NSNotificationCenter 是整个通知过程的核心，它允许我们注册观察者，允许张贴者发送通知以及撤销观察者注册。它有一个类方法用于获取全局的 NSNotificationCenter 对象：

```
+ (id)defaultCenter;
```

此外，它允许在通知中心注册一个观察者：

```
- (void)addObserver:(id)anObserver
          selector:(SEL)aSelector
             name:(NSString *)notificationName
           object:(id)anObject
```

anObserver 是要注册的观察者对象；aSelector 为接收到通知后观察者将调用的处理方法；notificationName 为 Poster 对象发送的通知名称，如果 notificationName 为 nil，则代表将通知中心中或某个张贴者（如果指定 anObject 参数的话）的所有通知都转发给这个 observer 对象；anObject 为张贴者对象(Poster),如果 anObject 为 nil,则代表所有对象中的名为 notificationName 的通知（如果指定的话，未指定则指所有通知）都转发给 anObserver。

要取消 Observer 的注册，使用如下方法：

```
(void)removeObserver:(id)observer
```

参数 observer 指定要注销的观察者对象。

张贴者要发送通知时使用如下方法：

```
- (void)postNotificationName:(NSString *)aName
                    object:(id)anObject
```

或者

```
-(void)postNotificationName:(NSString *)notificationName
                  object:(id)notificationSender
                  userInfo:(NSDictionary *)userInfo
```

我们创建了一个 NotificationDemo 项目演示了如何使用 Notification。项目位于光盘“source/第 15 章/NotificationDemo”目录。

MyPoster 类就是一个最简单的“张贴者”，它唯一的功能就是张贴名为“noti_key”的通知。noti_key 是一个 NSString 常量：

```
#define noti_key @"MyPosterChangedNotification"
```

MyPoster 类只有一个 changeText 方法用于张贴 noti_key 通知：

```
-(void)changeText:(NSDictionary*)userInfo{
[[NSNotificationCenter defaultCenter] postNotificationName:noti_key
     object:self
```

```
        userInfo:userInfo];
}
```

任何对象都可以作为“张贴者”，只要它调用了 NSNotificationCenter 的 postNotificationName 方法，Notification 框架对此没有任何限制。我们其实可以在 ViewController 甚至是 App Delegate 中张贴通知，没有必要单独实现一个 MyPoster 类。我们这样做，仅仅是出于演示的目的。

MyObserver 类是一个简单的“观察者”，这个类没有任何特殊的地方，只有一个简单的用于接收（处理）通知的方法。在这个方法里，我们打印了通知内容：

```
-(void)handleNotification:(NSNotification*)notification{
    NSLog(@"%@:\n%@",notification.name,notification.userInfo);
}
```

任何对象都可以作为“观察者”，只需使用 NSNotificationCenter 的 addObserver 方法将它进行注册。这个工作我们留在 App delegate 中进行。

NotificationDemoAppDelegate 的 regOrUnregObserver 方法用于观察者对象的注册和取消注册。它作为 UIButton 的 Action 方法。当 UIButton 被点击时，它会在“已注册”或“已注销”状态之间切换。如果当前是“已注销”状态，regOrUnregObserver 方法会进行观察者的注册，并把状态切换到“已注册”状态；反之，regOrUnregObserver 方法会对已注册的观察者对象进行取消注册，并把状态切换到“已注销”状态：

```
-(IBAction)regOrUnregObserver{
    if ([@"注册观察者" isEqualToString:button.titleLabel.text]) {
        [button setTitle:@"注销观察者" forState:UIControlStateNormal];
        [center addObserver:observer
                   selector:@selector(handleNotification:)
                       name:nil
                     object:poster];
    }else{
        [button setTitle:@"注销观察者" forState:UIControlStateNormal];
        [center removeObserver:observer];
        [button setTitle:@"注册观察者" forState:UIControlStateNormal];
    }
}
```

注册观察者之后，观察者会等待 NSNotificationCenter 的通知。NSNotificationCenter 则负责监听张贴者的通知，并将符合观察者要求的通知转发给观察者进行处理。

那么接下来的事情就是张贴者张贴通知，如果张贴者没有张贴任何通知，观察者将不会有任何动作。我们将 MyPoster 的张贴动作放在了 TextView 的 delegate 方法里，这样当 TextView 中的文本有任何改动时，MyPoster 都会向 NSNotificationCenter 发送一条通知：

```
#pragma mark - TextField delegate
- (void)textViewDidChange:(UITextView *)_textView{
```

```
    NSString* text=textView.text?textView.text:@"";
    NSDictionary* userInfo=nil;
    userInfo=[NSDictionary dictionaryWithObjectsAndKeys:@"valueChanged",@"event_name",
             text,@"value", nil];
    [poster changeText:userInfo];
}
```

注意：我们使用 TextView 而不是 TextField，是因为 TextFieldDelegate 中没有一个 textFieldDidChange:方法（即文本值改变方法）。

运行程序，在文本框中输入任意字符，如图 15-1 所示。

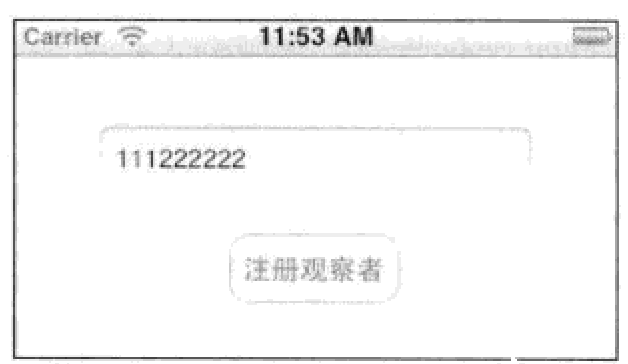


图 15-1　NotificationDemo 程序界面

此时，由于观察者处于“未注册状态”，控制台中将不会有任何输出。

我们点击“注册观察者”按钮，将会把观察者注册到 NSNotificationCenter。然后修改文本框中的文本，将在控制台中看到如下输出：

```
2012-07-19 11:51:23.568 NotificationDemo[56784:207] MyPosterChangedNotification:
{
    "event_name" = valueChanged;
    value = 111222222;
}
```

这些输出来自于观察者 MyObserver 对象，通知的内容来自于 MyPoster 对象，这两个类通过 NotificationCenter 实现了通信。这种方式的通信不同于方法调用，张贴者不需要知道观察者的实现细节，甚至可以不知道是否有观察者存在；同时观察者也不知道张贴者的实现细节，二者关心的只有数据（即 userInfo）。这也和方法委托不太一样，张贴者和观察者二者之间没有直接的调用关系（接口）存在，一切都是通过 NotificationCenter 来进行的。

15.2　本地通知

本地通知和远程通知是苹果两种通知应用程序的方法（通知时不需要程序在后台运行）。任何时候，iPhone、iPad 或者 iPod 中只能有一个程序在前台运行。许多实时性要求高，或者是交互式环境中运行的程序，要求当程序未在前台运行时能让用户知道发生了什么事情。本地通知和远程通知允许这些程序通知用户某些事情发生了。iPad 的 QQ 就是一种远程通知的典型

应用。只要你开启了 iPad QQ 的远程通知选项（设置程序中），则无论你是否在使用 iPad QQ，甚至你退出了 iPad QQ，重新启动了 iPad，只要你的 iPad 电源还在开启，网络还可以使用，则 QQ 消息总是能够发送到 iPad QQ 上，而无论 iPad QQ 是否在运行，你的 QQ 账号是否已登录。

远程通知在 iOS 3.0 中引入，本地通知在 iOS 4.0 中引入。本节介绍本地通知，远程通知在下一节介绍。

通知可能是一条消息、一个将要发生的日历事件，或者来自远程服务器的数据。当操作系统收到通知时，通知可能会显示为一个警告消息对话框，或者在应用程序的徽标中显示，同时发出声音提示。

要让操作系统发送本地通知，应用程序需要创建 UILocalNotification 对象，设置它的发送日期/时间，指定呈现的细节，然后把它放到任务列表中。

一个本地通知是一个 UILocalNotification 对象，它包含如下属性：

- fireDate 是 UILocalNotification 激发的确切时间。
- timeZone 是 UILocalNotification 激发时间是否根据时区改变而改变。
- repeatInterval 是 UILocalNotification 被重复激发之间的时间差。
- repeatCalendar 是 UILocalNotification 重复激发所使用的日历单位需要参考的日历，如果不设置的话，系统默认的日历将被作为参考日历。
- alertBody 是一串用于显示提醒内容的字符串（NSString），如果 alertBody 未设置的话，Notification 被激发时将不显示提醒。
- alertAction 也是一串字符 （NSString），alertAction 的内容将作为提醒中动作按钮上的文字，如果未设置的话，提醒信息中的动作按钮将显示为“View”相对应的文字形式。
- alertLaunchImage 是在用户点击提醒框中的动作按钮（“View”）时，等待应用加载时显示的图片，这个将替代应用原本设置的加载图片。
- hasAction 是一个控制是否在提醒框中显示动作按钮的布尔值，默认值为 YES。
- applicationIconBadgeNumber 是显示在应用图标右上角的数字，这样让用户直接了解到应用需要处理的 Notification。等于 0 则不显示徽标。
- soundName 是另一个 UILocalNotification 用来提醒用户的手段，在 Notification 被激发之后将播放这段声音来提醒用户有 Notification 需要处理，如果不设 soundName 的话，Notification 被激发时将不会有声音播放，除去应用特指的声音以外，也可以将 soundName 设为 UILocalNotificationDefaultSoundName 来使用系统默认提醒声音。
- userInfo 是 Notification 用来传递数据的 NSDictionary 对象。

我们有一个示例项目，放在光盘“source/第 15 章/LocalNotification”目录下。在 ViewController 类里，我们演示了本地通知的创建、属性设置、调度、取消调度。

我们在 ViewController 中放入了一个 UISwitcher，它会在 ON 或 OFF 状态中来回切换，并调用 Action 方法 onOrOff。在 onOrOff 方法中，我们判断 UISwitcher 按钮的状态，当按钮切换到 ON 状态时，循环调用[self scheduleNotification:index:]方法，一次创建了 4 条本地通知。当按钮切换到 OFF 状态，我们把所有未触发完的本地通知取消。以下是 onOrOff 方法代码：

```
-(IBAction)onOrOff:(id)sender{
    if(sw.on){
       [self restoreDefault];
       [[UIApplication sharedApplication]cancelAllLocalNotifications];
        for(NSDictionary* each in mArray){
            int index=((NSNumber*)[each objectForKey:@"number"]).intValue;
           [self scheduleNotification:index-1];
       }
       [table reloadData];

    }else {
         // 取消所有通知
       [mArray removeAllObjects];
       [table reloadData];
         [[UIApplication sharedApplication] cancelAllLocalNotifications];
     }
}
```

其中 restoreDefault 方法从应用程序的 NSUserDefaults 中读取 todo 列表到 mArray 数组。如果程序是第一次运行，NSUserDefaults 为空，则我们会创建一个默认的 todo 列表：

```
-(void)restoreDefault{
    mArray=[[NSMutableArray alloc]init];
    NSArray* arr=[Prefs array];
    if (arr==nil || arr.count==0) {
        for (int i=1; i<5; i++) {
            NSMutableDictionary* d=[NSMutableDictionary dictionaryWithObjectsAndKeys:
                                 [NSNumber numberWithInt:i],@"number",
                                 [NSNumber numberWithInt:10*i],@"time",
                                 @"NC",@"didRead",
                                 @"It's time to do something.",@"info",nil];
            [mArray addObject:d];

        }
    }else{
        for (NSDictionary* each in arr) {
            [mArray addObject:[each mutableCopy]];
        }
    }
}
```

Prefs 类负责从 UserDefaults 中读取 todo 列表以及将程序中的 mArray 数组保存到 UserDefaults 数据库。具体实现请参考 Prefs 代码。通过我们前面介绍过的 NSUserDefaults 类的使用方法，你可以很容易读懂它。

真正创建一条本地通知的代码在 scheduleNotification:index:方法里。我们在这里创建一条

本地通知，设置它的属性，然后调用 UIApplication 的 scheduleLocalNotification 方法将通知安排进计划里。为了便于查看本地通知的内容，我们把本地通知的 userInfo 复制到 MutableDictionary（加入了一个“didRead”的 key，用于标记该通知是否被用户查看过），又将 MutableDictionary 加入 mArray 数组中。UITableView 则根据 mArray 来进行刷新，如果用户阅读过某条通知，我们在 UITableView 的单元格中标记一个勾号。以下是 scheduleNotification:index: 方法：

```
-(void)scheduleNotification:(int)index{
    assert([mArray objectAtIndex:index]);
    NSMutableDictionary* dictionary=[mArray objectAtIndex:index];
    if ([@"YES" isEqualToString:[dictionary objectForKey:@"didRead"]]) {
        return;
    }
// 构造本地通知对象
    UILocalNotification *notification=[[UILocalNotification alloc] init];
    // 设置通知对象的属性
    notification.timeZone=[NSTimeZone defaultTimeZone];//时区
    notification.repeatInterval=0;//重复周期,0-不重复
    NSNumber* number=[dictionary objectForKey:@"number"];
    notification.applicationIconBadgeNumber = number.intValue;//徽章数
    notification.alertBody=@"收到一条提醒消息。";//告警消息文本
    [notification setSoundName:UILocalNotificationDefaultSoundName];//声音
    NSNumber* time=[dictionary objectForKey:@"time"];
    notification.fireDate=[NSDate dateWithTimeIntervalSinceNow:time.intValue];
    [dictionary setObject:notification.fireDate forKey:@"fire_date"];
    [notification setUserInfo:dictionary];
// 调度本地通知
    [[UIApplication sharedApplication]scheduleLocalNotification:notification];
    [notification release];
}
```

我们用 scheduleNotification:index:方法构建了 4 条本地通知，每条通知相隔 10 秒（见 onOrOff 方法）。为了具体识别每条通知，我们在 userInfo 中使用了 number 字段进行编号。

调度完本地通知，我们接下来要讨论的是本地通知的处理。苹果的本地通知处理 API 包括两个 UIApplication delegate 方法：

```
application:didFinishLaunchingWithOptions:
application:didReceiveLocalNotification:
```

这两个方法都在应用程序委托（即 AppDelegate）中实现。第一个方法调用的时机是应用程序启动时。第二个方法的调用则是在应用程序处于前台运行或后台运行时。

这两个方法都只能处理通知告警窗口的 Action 按钮，苹果并不允许你处理 Close 按钮。

第一个方法的调用时机很罕见。你可以先运行程序，将 UISwitcher 按钮拨至 ON 状态，然后迅速退出程序（双击 Home 键，在屏幕最下方的任务历史中叉掉它）。我不得不告诉你这个

动作必须在 10 秒内完成。如果你像我一样属于手脚比较慢的人，你可能要修改 restoreDefawlt 方法中 for 循环中的常量为 10（修改为更长的时间），然后再来测试。这样当第 1 条本地通知的告警窗口呈现时，你的应用程序应当是退出了。如果你按下 Action 按钮（即告警窗口中的 View 按钮），则此时会调用第一个委托方法。如果你要在这里处理本地通知，那么将代码写在这个方法里。比如你想保持会话状态，那么你必须在会话结束也就是应用程序退出时保存状态（正如我们在这个示例程序中所做的一样），然后这里重新恢复会话状态。注意，不要在这个方法中调用任何更新 UI 的代码，因为在 didFinishLaunchingWithOptions:方法返回之前，UI 是不可用的——你会发现 View Controller 的 viewDidLoad 方法要在此之后才调用。如果你有一些刷新 UI 的工作要做，那么建议你在这里修改一些 View Controller 的变量，然后在 View Controller 的 viewDidLoad 方法中根据这些变量更新 UI。例如，我们在 didFinishLaunchingWithOptions: 方法加入以下代码：

```
Class cls = NSClassFromString(@"UILocalNotification");
    if (cls) {
        UILocalNotification *notification = [launchOptions objectForKey:
                                    UIApplicationLaunchOptionsLocalNotificationKey];
        if (notification && notification.userInfo) {
                //--application.applicationIconBadgeNumber;
                self.viewController.switchON=YES;
        }
    }
```

如果程序退出以后，本地通知弹出，那么点击 Action 按钮会启动程序并触发 didFinishLaunchingWithOption:方法。为了识别它是不是被本地通知的 Action 按钮触发的，我们检查了方法的 lauchOptions 参数，看看它里面是不是有一个 key 叫做 UIApplicationLaunchOptionsLocalNotificationKey 的对象，这个对象保存了本地通知。如果存在本地通知，且 userInfo 有效，则我们设置 viewController 的 switchOn 属性为 YES，这样当 viewController 加载时（viewDidLoad 方法）会根据这个属性刷新 UI：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    if (switchON) {
        [sw setOn:YES];
        [self onOrOff:sw];
    }
}
```

其实就是将 UISwitch 设置为 ON 状态，然后加载 todo 列表。

第二个方法（didReceiveLocalNotification:方法）是通常的本地通知 Action 处理方法。当程序处于前台运行或后台运行状态时（程序运行后多半处于这两种状态），当本地通知告警窗口呈现时，用户点击 Action 按钮就会触发这个方法。记住，你不可能处理 Close 按钮事件。在

这个方法里，我们调用了 showNotification:方法：

```
- (void)application:(UIApplication *)application didReceiveLocalNotification:
      (UILocalNotification *)notification{
    if(notification.userInfo){
        int number=((NSNumber*)[notification.userInfo objectForKey:@"number"]).
              intValue;
        NSLog(@"notification number:%d", number);
        [self.viewController showNotification:number];
    }
}
```

viewController 的 showNotification:方法仅仅是使用 UIAlertView 显示 userInfo 中的 info 字段，然后将 didRead 字段标志为已读：

```
-(void)showNotification:(int)number{
    NSMutableDictionary* dic=(NSMutableDictionary*)[mArray objectAtIndex:number-1];
    NSString* info=(NSString*)[dic objectForKey:@"info"];
    UIAlertView *alertView=[[UIAlertView alloc]initWithTitle:@""
                        message:info delegate:nil cancelButtonTitle:@"好"
                        otherButtonTitles:nil];
    [alertView show];
    if ([@"NO" isEqualToString:[dic objectForKey:@"didRead"]]) {
        [dic setObject:@"YES" forKey:@"didRead"];
        [table reloadData];
    }
}
```

TableView 的 delegate 方法也进行了同上的处理，但我们还要考虑到一个问题：当用户点击了 TableViewCell，则表明它已经看过消息了，那么这条消息所对应的本地通知是否还有必要触发（如果还未触发的话）？答案是否定的，因此在这里我们直接把这个本地通知进行了取消：

```
-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
      *)indexPath{
    [self showNotification:indexPath.row+1];

    for (UILocalNotification* each in
         [UIApplication sharedApplication].scheduledLocalNotifications) {
        int number=((NSNumber*)[each.userInfo objectForKey:@"number"]).intValue;
        if (number-1==indexPath.row) {
            [[UIApplication sharedApplication]cancelLocalNotification:each];
            break;
        }
    }
}
```

在调试程序的过程中，我们发现本地通知存在着一个最大的挑战：正确显示应用程序的

徽标数字。说实话，这个问题真的很难。我们可以通过两种方式修改应用程序图标右上角徽标数字：

- 修改 UIApplication 的 applicationIconBadgeNumber。
- 设置 UILocalNotification 的 applicationIconBadgeNumber。

但不管是哪一种，你都必须先解决这个难题：什么时候更新这个数字？因为应用程序的状态是不确定的，它可能未启动，这时仅仅是放在桌面上的一个图标；也可能正在后台或前台运行。如果在前台，徽标数字发生变化，比如收到新的通知或者我们处理掉一个通知时，我们要更新这个应用程序图标显示。不管采用两种方法中的哪一种，我们都很容易做到。

问题是当它处于后台时怎么办？当一个应用程序处于后台时，它的代码是被挂起的。“挂起”这个词充分说明了代码的执行状态（就像中断一样）。那么你不能执行任何代码，包括去修改应用程序的 applicationIconBadgeNumber。程序处于关闭状态也面临同样的问题。虽然程序已经退出，但它所调度的本地通知仍然会按时触发，“调度”一词充分说明，我们可以预先安排一个计划，然后 iOS 会自动按照这个计划去发送本地通知，而不论这个应用程序是否仍然在运行。如果程序退出后，本地通知呈现，徽标的数字是否需要改变？但我们的代码没有任何机会执行。唯一能做的就是等待用户点击 Action 按钮。此时你可以更新 applicationIconBadgeNumber。但如果用户点击的是 Close 按钮，我们没有任何机会。于是我们放弃了一个本地通知，但应用程序图标上没有任何变化，右上角的数字仍然还是老样子。

另外还有一个有歧义的地方：这个数字到底表明什么？它的定义依赖于你自己的理解。在这里我们假设这个数字代表了当前已到期的通知数。UIApplication 对象有一个 scheduledLocalNotifications 数组属性，代表已调度的本地通知，你可以通过它的 count 属性获得当前尚未发出的通知数。这个数字恰巧就是未到期的通知数。那么很显然，所有通知减去未到期的，就应当是已到期的。未到期的通知数我们通过 scheduledLocalNotifications.count 获得。那么所有通知数是什么？你会说 4 条，即 mArray.count。并不是这样，mArray 中的已读消息是不用发通知的。所以我们应当统计 mArray 中的未读消息数，然后减去 scheduledLocal Notifications 的 count（未到期的），即可得到当前已发过通知数，用这个数字更新应用程序徽标。这个计算逻辑在 showedNotifications 方法（即 showed_noti 属性的 getter 方法）中表现：

```
-(int)showedNotifications{
    int result=0;
    for (NSDictionary* each in mArray) {
        if ([@"NO" isEqualToString:[each objectForKey:@"didRead"]]) {
            result++;
        }
    }
    result=result-[[UIApplication sharedApplication]scheduledLocalNotifications].
                count;
    return result;
}
```

最后，我们再来解决刷新徽标的问题。

首先，当程序在前台运行时，我们不需要考虑徽标的更新问题。这是因为：1）程序运行在前台时，本地通知不会呈现告警窗口；2）程序在前台时，用户看不见应用程序图标，看不见的东西自然不用考虑刷新了。

其次，当程序在后台运行时，我们可以考虑用 iOS4 的后台任务来执行刷新。由于我们不知道通知什么时候触发（为什么苹果不提供一个 willReceiveLocalNotification:的委托方法呢？），所以我们只能用一个定时的 RunLoop（关于 RunLoop，请参考前面的章节）不停地刷新当前已到期的通知数：

```
-(void)refreshBadges:(UIApplication*)application{
    application.applicationIconBadgeNumber=self.viewController.showed_noti;
}
- (void)applicationDidEnterBackground:(UIApplication *)application
{
// 设置超时 handler，超时时系统会调用这个块
    backgroundTask = [application beginBackgroundTaskWithExpirationHandler: ^{
        // 提交一个块到目标队列，以便异步执行
        dispatch_async(dispatch_get_main_queue(), ^{
            if (backgroundTask != UIBackgroundTaskInvalid)
            {
                [application endBackgroundTask:backgroundTask];
                backgroundTask = UIBackgroundTaskInvalid;
            }
        });
    }];
// 开始运行后台任务 dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_
   PRIORITY_DEFAULT, 0), ^{
        // [[HTTPServer sharedHTTPServer]start];
        dispatch_source_t timer;
        timer = dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER, 0, 0, dispatch_
                get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0));
        dispatch_source_set_timer(timer, DISPATCH_TIME_NOW, 3ull * NSEC_PER_SEC, 0);
        dispatch_source_set_event_handler(timer, ^{
            [self refreshBadges:application];
        });
        dispatch_resume(timer);
    });
}
```

最后，当程序处于退出状态时，我们想不出任何办法来更新徽标数字，因为即使是后台任务，也不会在这种状态下得到执行。

程序运行时，显示界面如图 15-2 所示，此时尚未生成本地通知。

当你将 switch 控件拉到开状态时，退出程序。稍微等待数秒钟，本地通知将弹出告警消

息和徽章图标，如图 15-3 所示，并播放声音。

图 15-2　程序运行界面

图 15-3　本地通知告警窗口

如果你点击 action 按钮 “View”，将重新打开应用程序，并弹出 AlertView 窗口显示消息内容，如图 15-4 所示。

图 15-4　点击 Action 按钮之后呈现的界面

15.3　远程通知

远程通知即推送通知（Apple Push Notification service，苹果推送通知服务，APNs）。同前面介绍的本地通知一样，APNs 可用于在应用程序不在前台运行时，通知应用程序某些消息并唤醒应用程序。

接下来介绍苹果推送通知服务，包括：如何在应用程序中接收 APNs 以及如何发送一条推送通知。

真正推送通知的发送是由提供者（Push Notification Provider）来完成的。因此创建 Push Notification Provider 是我们在使用苹果推送通知服务中必不可少的过程。一个推送通知提供者需要实现以下内容：

- 负责与苹果 APNs 服务器进行 SSL 通信（使用我们在前面所申请和下载到的 SSL 证书）；
- 负责构造通知内容——即消息载体 Payload；
- 发送载体到 APNs 服务器；

在接下来的内容里，我们将介绍如何创建一个 APNs 应用程序。

15.3.1 Apple Push 简介

iPhone 对于应用程序在后台运行有诸多限制（除非你越狱）。因此，当用户切换到其他程序后，原先的程序无法保持运行状态。对于那些需要保持持续连接状态的应用程序（比如社区网络应用），将不能收到实时的信息。

为解决这一限制，苹果推出了 APNs（苹果推送通知服务）。APNs 允许设备与苹果的推送通知服务器保持常连接状态。通过使用 APNs ，我们能够将消息推送到目标设备上的某个已安装的应用程序。

当一个推送通知发送到 iPhone 时，可能会弹出一个警告框，如图 15-5 所示。

图 15-5　在 iPhone 上收到的远程通知

当你点击 Action 按钮（显示按钮）后，推送通知被送达目标应用程序，目标应用程序应该被启动（当它未在前台运行时）并对通知进行处理。

同本地通知一样，推送通知有多种呈现形式。除了告警对话框，它还会在应用程序图标的右上角显示数字徽标，或者发出提示性的声音。

注意，早先苹果公司的相关文档叫做苹果推送通知服务编程指南（Apple Push Notification Service Programming Guide），而后来的文档则更新为本地通知和推送通知编程指南（ Local and Push Notification Programming Guide）。实际上，本地通知和推送通知是两种不同的技术，苹果可能因为它们在客户端的呈现方式相同（都以警告、徽章和声音的形式出现）而将它们放在了一起。

15.3.2 准备使用 APNs

要使应用程序能够接收到推送通知，需要经过一系列的准备工作。如：创建和配置 App ID、申请 SSL 证书、创建 Provisioning Profile 并在设备上安装。然后用已配置的 App ID 和

Provisioning Profile 进行代码签名。最后一个环节才是编写 APNs 代码。

1. 生成证书请求

首先，我们需要生成一个 CSR 证书请求，证书请求在你申请开发者证书时就已经生成过（在第 1 章制作开发者证书的过程中我们曾经介绍过证书请求的生成）。如果你不是第一次开发 iOS 应用程序，那么你可能已经有了这些东西。 那么你可以继续以下的步骤。

2. 生成 App ID

登录你的 provisioning portal，点击左侧边栏的 App IDs 菜单，在这里你可以查看已有的 App ID。

你可以从 App ID 列表中选择一个 App ID 进行配置（点击右边的 Configure 按钮）。也可以重新创建一个新 App ID（页面右上角的 New App ID 按钮）。我们假设是后者。点击 New App ID，按照如图 15-6 所示进行设置。

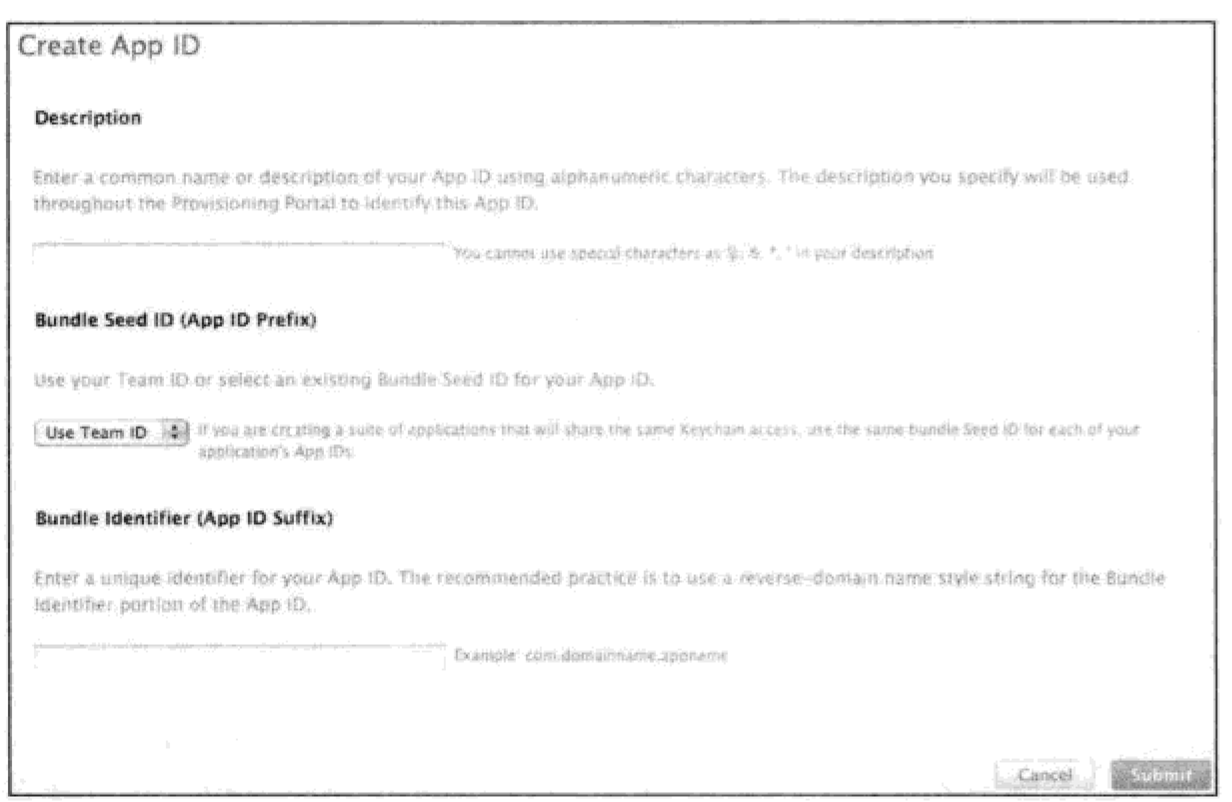


图 15-6　新建 App ID

在 Description 一栏为 App ID 输入一个描述性的名称，比如"My APNs' App ID"。

在 Bundle Seed ID 一栏，苹果提供了 Team ID 的概念（即 App ID 前缀），如果你要和其他应用程序共享钥匙串，可以选择一个已经存在的 Bundle Seed ID。否则保持默认无需改变。

在 Bundle Identifier(App ID Suffix)一栏，填入 App ID 后缀。苹果推荐使用反域名作为 App ID 后缀。

点击 Submit 按钮，马上可以在 App IDs 列表中看到你新建的 App ID。最终生成 App ID 类似于如下形式：

```
7AP5A3T0EK.com.ydtf.AppID
PushAppID
```

其中，PushAppID 是新建 App ID 时我们为它输入的描述性名称。7AP5A3T0EK.com.ydtf.AppID 则是 App ID，它由“App ID 前缀”和“App ID 后缀”两部分构成。“App ID 前缀”（即“7AP5A3T0EK”）不可以由我们自己设定，它是由苹果分配的 GUID 串，由苹果维护，不可能重复。“App ID 后缀”（即“com.ydtf.AppID”）是我们新建 App ID 时输入的 Bundle Identifier，是根据反域名规则来命名的。

点击右边的 Configure 按钮，即可配置 App ID 的 APNs 选项。

注意：在 App IDs 列表中，有的 App ID 的 Apple Push Notification Service 列是灰色的，并且不允许使用 Configure 按钮。这是因为 APNs 不支持带通配符的 App ID。

---

3. 配置 App ID 生成 SSL 证书

App ID 配置页面如图 15-7 所示。

Enable for Apple Push Notification service

Push SSL Certificate	Status	Expiration Date	Action
Development Push SSL Certificate	Configurable		Configure
Production Push SSL Certificate	Configurable		Configure

图 15-7　配置 App ID

勾选“Enable for Apple Push Notification service”，点击“Development Push SSL Certificate”右边的“Configure”按钮。

注意，“Development Push SSL Certificate”用于开发调试，“Production Push SSL Certificate”用于发布。

在接下来的“Apple Push Notification service SSL Certificate Assistant”页面中，点击 Continue 按钮。

然后选择你硬盘上保存的 CSR 证书请求文件，点击 Generate 按钮，以生成一个 SSL 证书。点击 Download 按钮把 ssl 证书下载到本地。文件名为 aps_developer_identity.cer，双击，将证书安装到钥匙串中。这个证书会在你的程序中用到，它允许程序接收 APNs 发送来的推送通知。

4. 创建 Provisioning Profile

Provisioning Profile（即.mobileprovision 后缀名的文件）用于把 ssl 证书安装到设备上。点击 Provisioning Portal 中左侧边栏的 Provisioning 菜单进入 Provisioning Profile 页面。

用 New Profile 按钮新建一个 Provisioning Profile，如图 15-8 所示。

填写 ProfileName，如 PushDeviceProfile，在 Certificate 栏中勾选这个 Profile 所要绑定的开发证书——如果一个开发证书表示一个开发者，那么这里就是指定哪些开发者可以使用这个 Profile。如果不太清楚，把所有证书勾上即可。在 App ID 栏选择我们前面创建的 App ID（这里是指定哪些应用程序可以使用这个 Profile 签名）。在 Devices 栏勾上用于接收推送通知的设备，也可以把所有设备勾上。

Profile name　PushDeviceProfile

Certificates　Select All

he lei　Hongyan Yang

yang steven　yy yayowd

Zhao Chuanxue

App ID　PushAppID

Devices　Select All

32GBIP4

bf2000bf

ch999的 iPad　Duanhg

图 15-8　新建 Provisioning Profile

点击 Submit，将创建 Profile。等待几秒刷新页面，Download 按钮将出现，下载该证书，文件名为 PushDeviceProfile.mobileprovision。把设备（iPhone/iPad/iPod）连接上 Mac，将该 .mobileprovision 文件拖到 Dock 栏的 Xcode 图标，即可在设备上安装该 Profile。

5. 代码签名

用 Xcode 新建 Application，命名为 APNsTest。准备一个.wav 文件，比如 machinegun.wav，拖到项目文件夹中（勾选“Copy Items...”)。

选择 Targets 下的 APNsTest，打开 Summary 窗口，把 Indentifier 修改为我们前面创建的 App Id 。

切换到 Build Settings 页，找到 Code Signing Identity。在 “Debug”下的“Any iPhone OS Device”选项中选择正确的 profile(即前面创建的 Provisioning Profile) ，注意证书和 profile 是成对使用的，如图 15-9 所示。

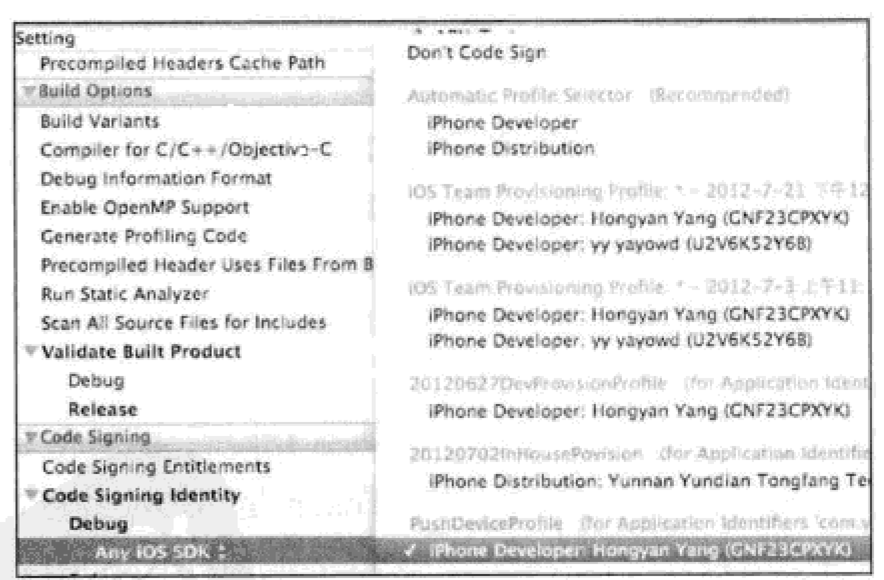


图 15-9　代码签名

在本例中，我们用于代码签名的 profile 是前面创建的 PushDeviceProfile，使用的证书是 Hongyan Yang（正是我们在创建 profile 时，在 Certificates 栏中选择的证书，见图 15-8）。

15.3.3 准备接收推送通知

打开 APNsTestAppDelegate.m，找到代码“[window makeKeyAndVisible];”，在后面加入一句：

```
// 注册 APNs 类型：警告+徽章+声音
[[UIApplication sharedApplication] registerForRemoteNotificationTypes: (UIRemote
    NotificationTypeAlert|UIRemoteNotificationTypeBadge|UIRemoteNotification-
    TypeSound)];
```

registerForRemoteNotificationType 方法用于向 iOS 注册应用程序愿意接收的远程通知类型（警告、徽标和声音）。未注册的远程通知类型将被 iOS 忽略。

然后实现 3 个 UIApplicationDelegate 委托方法：

```
- (void)application:(UIApplication *)app didRegisterForRemoteNotificationsWith
        DeviceToken:(NSData *)deviceToken { // ❶
    NSString *str = [NSString  stringWithFormat:@"Device Token=%@",deviceToken];
    NSLog(@"%@",str);
}
// 注册 APNs 错误
- (void)application:(UIApplication *)app didFailToRegisterForRemoteNotifications
        WithError:(NSFrror *)err { // ❷
    NSString *str = [NSString stringWithFormat: @"Error: %@", err];
    NSLog(@"%@",str);
}
// 接收推送通知
- (void)application:(UIApplication *)application didReceiveRemoteNotification:
        (NSDictionary *)userInfo {//❸
    NSString *msg=[NSString stringWithFormat:@"%@",userInfo];
    UIAlertView* alert=[[UIAlertView alloc]initWithTitle:@"通知" message:msg
        delegate:nil cancelButtonTitle:@"OK" otherButtonTitles:nil];
    [alert show];
    alert=nil;
}
```

代码说明：

❶ 通过这个方法的第 2 个参数可以获取到 Device Token。Device Token 是一个 64 位十六进制数。在苹果推送通知服务中，Device Token 是 APNs 服务器为每个已注册设备生成的 UUID 码。当应用程序使用 registerForRemoteNotificationTypes:方法向 iOS 注册远程通知类型时，iOS 操作系统会接收到 APNs 服务器返回的 Device Token，并通过这个方法将 Device Token 传给应用程序。

❷ 如果设备注册 APNs 服务不成功，则 iOS 操作系统通过这个方法 application:didFailToRegisterForRemoteNotifications WithError:将错误返回给应用程序。

❸ 这个方法只有在 APNs 注册成功、已获得一个 Device Token 并收到一个有效的通知时才有可能被调用（这个有可能是指用户点击了 Action 按钮，而不是 Close 按钮）。如果 iOS 收到 APNs 发来的推送通知，将会以警告、声音或徽章的形式提醒用户。如果用户点击了警告对话框中的 Action 按钮或者应用程序图标右上角的徽章图标，iOS 将调用这个方法，把通知发送给应用程序。

注意：通知能否通知应用程序，将由用户的动作决定。苹果不允许 iOS 直接将一个通知（不管是本地通知还是远程通知）发送给应用程序，而总是要由用户来触发是否让某个应用程序进行处理。

虽然到现在为止，我们还没有讨论如何真正发起一个远程通知，但已经可以从这个程序中获得一些重要信息（比如 Device Token）并对设备进行一些设置（比如程序所接受的通知种类）。在真实设备上运行程序（不要在模拟器上运行！），iOS 会在屏幕出现一个提示框，问你是否同意应用程序 APNsTest 接收推送通知，如图 15-10 所示。

当你点击“好”按钮之后，系统“设置”程序会配置好 APNsTest 的推送通知选项。同时，可以在控制台中看到如下输出：

```
Device Token=<804d95ab 708a0aad 33a08c1f 00341ae2 3774bcb0 02362e33 9a853575
        8f36dc90>
```

你可以把这个 Device token 复制下来，等一下要用。

打开设备的“设置→通用”，可以看到所有使用 APNs 的应用程序列表，如图 15-11 所示。

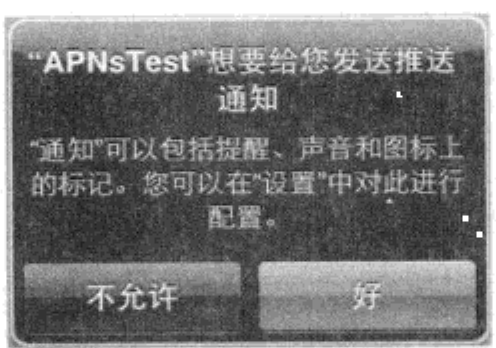


图 15-10　iOS 询问是否运行程序接收远程通知

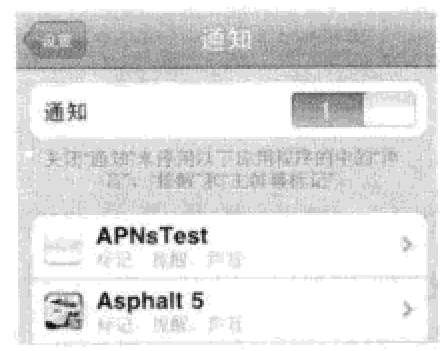


图 15-11　iOS 的“通知”设置

点击应用程序名称“APNsTest”，进入 APNsTest 的通知类型设置，可以看到当前应用程序已注册的通知类型包含了我们在 registerForRemoteNotificationTypes 方法调用中所注册的三种通知类型。你可以在这里进行修改，关掉或打开其中的任何一种，如图 15-12 所示。

图 15-12　应用程序的通知类型设置

15.3.4 创建 Push Notification Provider

Push Notification Provider（推送通知提供者）是一个程序，用于负责和苹果 APNs 服务器进行 SSL 通信，它是实际上的通知发送者（将远程通知发送到苹果 APNs 服务器）。如果你想向用户设备上的应用程序发送消息，实际上是由推送通知提供者进行的。整个 APNs 服务由 3 层应用程序构成：提供者、APNs 服务器、iOS 客户端。

苹果 APNs 服务器位于两者中间，起到一个中间人的角色。提供者向客户端发送消息，必须由 APNs 服务器来转发。开发人员除了需要实现客户端外，还需要实现提供者的代码。我们前面创建的 SSL 证书 aps_developer_identity.cer 需要在提供者程序中使用，因为苹果规定，要与 APNs 服务器通信需要一个合法的 SSL 数字证书。

提供者应该是一个基于流的 TCP socket，它以 SSL 协议与 APNs 服务器进行通信。推送通知（包括载体）是以二进制流的方式发送的。提供者和 APNs 建立连接后，可以维持该连接并在连接中断之前发送多个通知。

提示：应避免每发送一次推送通知就建立、关闭一次连接。频繁地建立、关闭连接可能会被 APNs 认为是 DOS 攻击，从而拒绝发送提供者的请求。

一个推送通知消息的格式如图 15-13 所示。

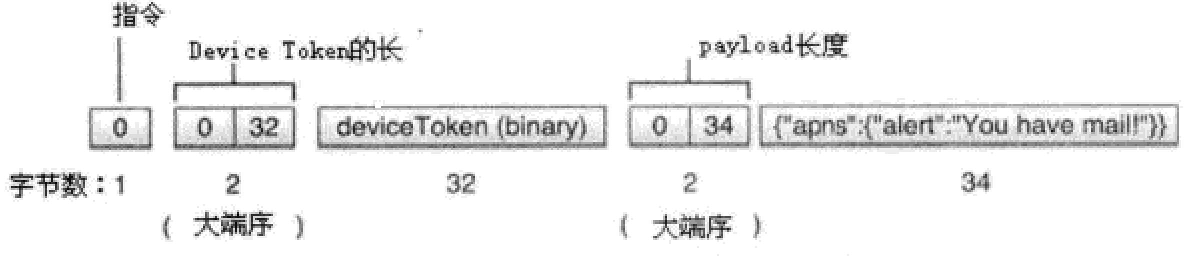


图 15-13 推送通知的消息格式

更多细节，请参考苹果公司参考库“Apple Push Notification Service Programming Guide”。

推送通知的消息格式中，payload 是通知消息的重要组成部分。

一个载体（payload）是一个 JSON 字符串（最长 256 字节），封装了你发送给 iOS 应用的信息。下面是一个 payload 的例子：

```
{
    "apns": {
     "alert" : "You got a new message!" ,
     "badge" : 5,
     "sound" : "beep.wav"},
     "acme1" : "bar",
     "acme2" : 42
}
```

其中，acme1 字段和 acme2 字段不是必须提供的，它们是用户自定义的数据。

注意： payload 最大不能超过 256 字节。也就是说，除了简单消息，你无法用推送通知来发送复杂数据。

如果仅仅是测试，我们可以使用一个其他人已经实现的提供者 PushMeBaby，下载地址为：

```
http://stefan.hafeneger.name/download/PushMeBabySource.zip
```

下载后解压缩，实际上得到的是一个 Xcode 项目。打开 PushMeBaby 项目，将 aps_developer_identity.cer 文件导入到 Resources 文件夹。

打开 ApplicationDelegate.m 文件，在 init 方法找到“self.deviceToken =”一行，将刚才复制的 Device token 粘贴到这里：

```
self.deviceToken = @"804d95ab 708a0aad 33a08c1f 00341ae2 3774bcb0 02362e33 9a853575
                    8f36dc90";
```

把“self.payload =”一行中的 You got a new message!替换成我们自己的提示信息“收到一条推送通知。”。

把“self.certificate =”一行中的 apns 替换成我们自己的 SSL 文件 aps_developer_identity（不需要扩展名）。

运行 PushMeBaby，程序显示的界面如图 15-14 所示。

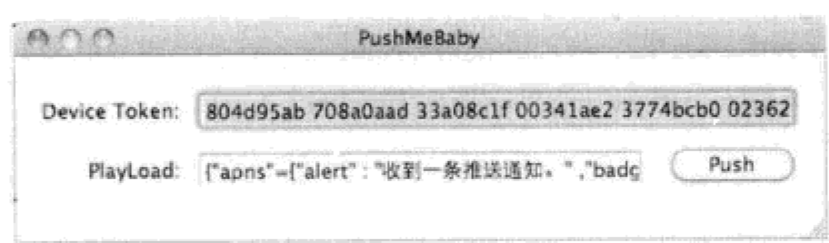


图 15-14　PushMeBaby 的发送界面

当 Mac 提示是否允许使用证书时，选择“总是允许”。然后点击 Push 按钮。

稍后，你的 iPhone/iPod 会收到一条推送通知，在图 15-5 中已经展示了通知到达时告警对话框的样子。

注意： 这个时候应当将 APNsTest 程序退到后台运行（按 Home 键），否则远程通知以另外的方式呈现。

如果你点击“显示”按钮（苹果称之为“Show Detail”按钮或“Action”按钮），则会启动 APNsTest 应用程序，在这个程序的 application: didReceiveRemoteNotification: 方法中，我们仅仅是用一个 UIAlertView 以字符串形式显示了这个通知的内容（payload）。

在载体中我们也可以使用其他自定义的字段（注意载体中的 myData 字段是我们自定义的），唯一的限制是整个 payload 不能超过 256 字节。

还有一个有趣的地方，如果发送通知时，用户的 APNsTest 已经在当前任务中，那么当用户收到通知时，则忽略通知，并直接在 APNsTest 程序中呈现如图 15-15 所示的界面。

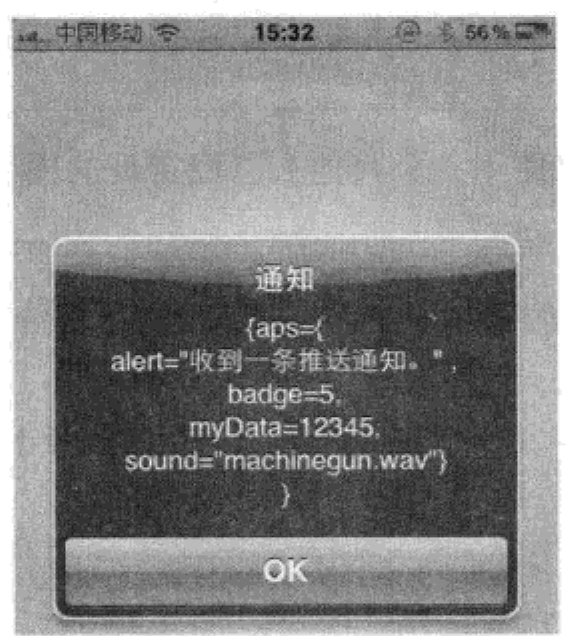


图 15-15　点击 Action 按钮后，应用程序被唤醒

如果想自己实现一个 Push Notification Provider，那么使用 C++、C#或者 Java 会更容易一些，虽然苹果官方的例子是使用 Objective-C 实现的。下面是一个 Java 实现的例子：

```
int port = 2195;
    String hostname = "gateway.sandbox.push.apple.com";
    char []passwKey = "<keystorePassword>".toCharArray();
    KeyStore ts = KeyStore.getInstance("PKCS12");
    ts.load(new FileInputStream("/path/to/apn_keystore/cert.p12"), passwKey);

    KeyManagerFactory tmf = KeyManagerFactory.getInstance("SunX509");
    tmf.init(ts,passwKey);
    SSLContext sslContext = SSLContext.getInstance("TLS");
    sslContext.init(tmf.getKeyManagers(), null, null);
    SSLSocketFactory factory =sslContext.getSocketFactory();
    SSLSocket socket = (SSLSocket) factory.createSocket(hostname,port);
    // 创建 ServerSocket
    String[] suites = socket.getSupportedCipherSuites();
    socket.setEnabledCipherSuites(suites);
    // 开始联络
    socket.startHandshake();
    // 为服务器安全收发数据创建流
    InputStream in = socket.getInputStream();
    OutputStream out = socket.getOutputStream();
    // 读入和输出
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    baos.write(0); // 命令
    System.out.println("First byte Current size: " + baos.size());
    baos.write(0); // deviceId 长度的第一个字节
    baos.write(32); // deviceId 长度
    System.out.println("Second byte Current size: " + baos.size());
```

```
String deviceId = "<heaxdecimal representation of deviceId";
baos.write(hexStringToByteArray(deviceId.toUpperCase()));
System.out.println("Device ID: Current size: " + baos.size());
String payload = "{\"aps\":{\"alert\":\"I like spoons also\",\"badge\":14}}";
System.out.println("Sending payload: " + payload);
baos.write(0); //First byte of payload length;
baos.write(payload.length());
baos.write(payload.getBytes());
out.write(baos.toByteArray());
out.flush();
System.out.println("Closing socket..");
// 关闭 socket
in.close();
out.close();
```

Java 代码不是本书重点，以上代码在此不做过多说明。

此外，你可以在 github 社区中找到一些 APNs Provider 的实现，比如 APNS-Sharp，这是一个 C#项目，作者是 Redth。该项目地址是：

```
https://github.com/Redth/APNS-Sharp
```

你可以自由地下载和使用这些代码。

15.4　本章小结

本章介绍了通知、本地通知和远程通知，这是三种完全不同的技术。

通知是 Cocoa 的观察者模式的实现。通过 NSNotificationCenter，观察者和张贴者能够跨过线程通信。这其中 Notification Center 扮演着“发行者”的角色，张贴者向 Notification Center 发布各种各样的消息，如果其他对象对其中的某些消息感兴趣，可以向“发行者”注册为观察者（或“订阅者”），于是 Notification Center 会在收到消息后将之发送给注册过的观察者。对象可以注册为多个对象（或通知）的观察者，也可以向不止一个对象张贴消息。

本地通知是应用程序用于“唤醒”自己而设置的“闹钟”。由于 iOS 限制了同一时间只能有一个进程在前台运行，所以 iOS 上用户的应用程序状态是不确定的。它们随时会被 iOS 中断执行而“挂起”。应用程序可以被本地通知所唤醒，前提是应用程序要预先为自己设置“闹钟”。但是，应用程序不能被其他应用程序调度的本地通知唤醒。

远程通知和本地通知很类似，但远程通知可以跨过进程传递。也就是说，其他应用程序可以用“远程通知”来唤醒 iOS 本地应用程序。这里的其他应用程序被称作“提供者”，并不仅限于是 iOS 应用程序——由于远程通知使用苹果 APNs 服务器来推送消息，因此 iOS 不关心提供者的具体实现（它可以用任何操作系统和语言来实现）。

本地通知和远程通知的设计初衷是为了弥补 iOS 的先天缺陷。iOS 4 以前，iPhone 并不支持多任务——即便是在 iOS 4 以后支持的多任务，也只能有一个程序在前台运行。iPhone 用户

出于各种各样的原因，随时会中止应用程序的运行——可能是用户想切换到 Mail 程序查看新邮件，或者接听来电。很不幸的是，当程序退出前台运行后，基本上就没有任何恢复执行的机会了（iOS 4 以前）。

这种情况变得糟糕，对于网络应用尤其如此——因为程序一旦进入后台，socket 将被 iOS 回收，网络连接会丢失。在 iOS 4 出现以前，用户的唯一选择就是重新启动应用程序，但这同时意味着此前的工作都会丢失，包括你已经下载的数据，或未提交的订单等。本地通知和远程通知的出现，在一定程度上解决了这方面的问题。通过本地通知和远程通知，你可以使应用程序在进入后台后被唤醒（虽然需要用户点击一下 Action 按钮），从而使未完成的工作得以继续。

第 16 章　开源框架 Core Plot

在企业应用中，企业数据经常要以图形、图表的形式展现。在 iOS 中内置了如 Quartz 2D 和 OpenGL ES 这样的图形框架（Quartz 2D 在本书第 11 章有介绍），然而要求企业开发人员用这些底层框架去绘制出令人满意的图表是不现实的。我们迫切需要找出一个第三方图形框架为我们完成这个任务。

iPhone 下的 2D 图形绘制框架并不是很多。大名鼎鼎的 Core Plot 正是其中之一。

Core Plot 是一个成熟的开源图表框架，经过多次版本的更新，目前已基本稳定。当前最新版本为 1.0。项目地址位于：http://code.google.com/p/core-plot/。它具有功能强大、支持 Mac OSX 和 iOS、拥戴者众多、文档资源丰富等特点。而且它是纯 Objective C 的——比起 C 函数库，显然更适合于我们的 ObjectiveC 程序员。目前 Core Plot 支持绘制：柱状图、折线图、散点图、饼状图以及函数图。

本章将初步介绍 Core Plot 框架。

16.1　编译 Core Plot 框架

首先需要下载 Core Plot 框架。可以在这里找到 1.0 版本 Core Plot 的下载链接：

```
http://code.google.com/p/core-plot/downloads
```

下载后文件名为：CorePlot_1.0.zip。解压缩，即可得到 Core Plot 框架的 SDK 及示例项目。

Core Plot SDK 以静态库的方式提供，我们需要编译成.a 静态库文件。打开解压缩目录的 Source/framework/ 目录，有两个 Xcode 项目：CorePlot-CocoaTouch.xcodeproj 和 CorePlot.xcodeproj。前者是 iOS SDK，后者是 OSX SDK。双击 CorePlot-CocoaTouch.xcodeproj，用 Xcode 打开。点击 Run，即可在 Source/build 目录下编译出 CorePlot-CocoaTouch.a 文件。

提示：如果你不想自己编译 Core Plot SDK 静态库，在解压缩目录 “Binaries/iOS/CorePlotHeaders” 目录中已经提供了一个现成的 CorePlot-CocoaTouch.a 文件。

16.2　使用 Core Plot SDK

将编译好的静态库（.a 文件）以静态库的方式链接到项目中。执行以下步骤来新建项目：

1）将解压缩目录 “Binaries/iOS/CorePlotHeaders” 下的所有头文件拖到你的 Xcode 项目中。

2）将.a 文件拖到你的项目中。

3）在 Target 的 Build Settings 设置中，找到 Other Linker Flags，加入 “ObjC -all_load”。

4）在项目中加入 QuartzCore 框架。

注意： 如果你在编译项目时遇到错误“Command /Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/clang failed with exit code 1”，请将 Scheme 从 iOS Device 改为 iPhone 5.0 Simulator。或者将 Compiler for C/C++改为 LLVM GCC 4.2。同时，Core Plot 1.0 不再支持老的 armv6 CPU。

接下来我们运行 Core Plot SDK 中的一个散点图的例子，演示如何在项目中使用 Core Plot SDK。

将解压缩目录的 Source/examples/CPTTestApp-iPhone/Classes 目录下的 CPTTestAppScatterPlotController.m 和 CPTTestAppScatterPlotController.h 文件拖到 Xcode 项目中。

在你的 view controller 的.xib 中，拖入一个 Object 对象，将 Identifier 改成 CPTTestAppScatterPlotController。这样是为了从.xib 中创建一个 CPTTestAppScatterPlotController 实例。将.xib 中的 View 对象的 Identifier 改成 CPTGraphHostingView。这是因为 CPTGraph 必须放在 CPTGraphHostingView 对象中，如图 16-1 所示。

连接 Graph Hosting View 对象到 Test App Scatter Plot Controller 的 view 出口上。运行程序，如图 16-2 所示。

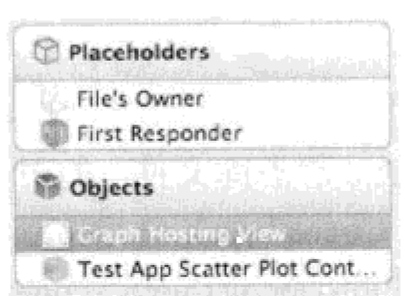


图 16-1 将 View 对象的类型改为 CPTGraphHostingView

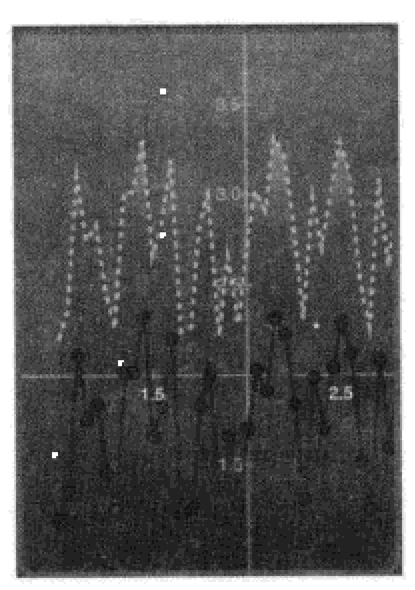


图 16-2 运行程序

但程序还有一个 Bug 需要解决。此时如果你拖动图表，程序将出现 EXEC_BAD 错误并崩溃。我们需要自己持有 Test App Scatter Plot Controller 对象。最简单的办法，是在 view controller 中创建一个出口，并连接到 Test App Scatter Plot Controller 对象。再次运行程序，你就可以拖动或缩放图表了。

16.3 安装 Core Plot 帮助文档

退出 Xcode。将解压缩目录“Documentation”目录下的 com.CorePlot.Framework.docset 文件和 com.CorePlotTouch.Framework.docset 文件拷贝到 Xcode 的文档目录：

```
~/Library/Developer/Shared/Documentation/DocSets/
```

注意，这个目录并不固定，它跟 Xcode 安装路径有关，如果你在机器上找不到这个目录，就得参考 Xcode 的安装路径。

重启 Xcode。现在你可以打开 Xcode 的帮助菜单，即点击“Help→Documentation and API Reference”。在 Orgnizer 的 Documentation 窗口，点击工具栏左边第 4 个按钮，会多出两项“Core Plot(iOS)”和“Core Plot(Mac OS)”，如图 16-3 所示。这样，你就可以在 Xcode 中查看 Core Plot 框架的帮助文档了。

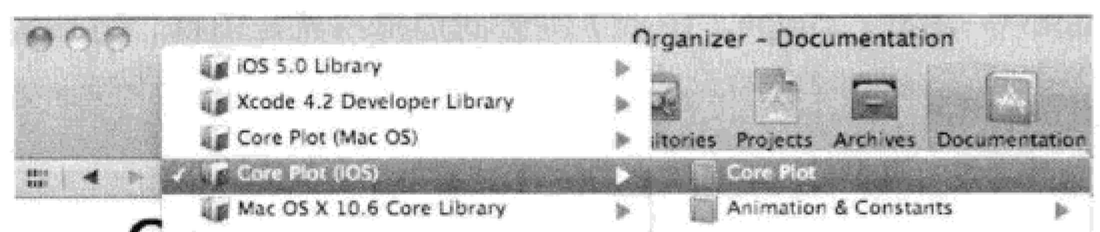


图 16-3 Core Plot 的文档

16.4 图表的构成

图 16-4 解释了 Core Plot 中图表的构成，及各组件的名称。

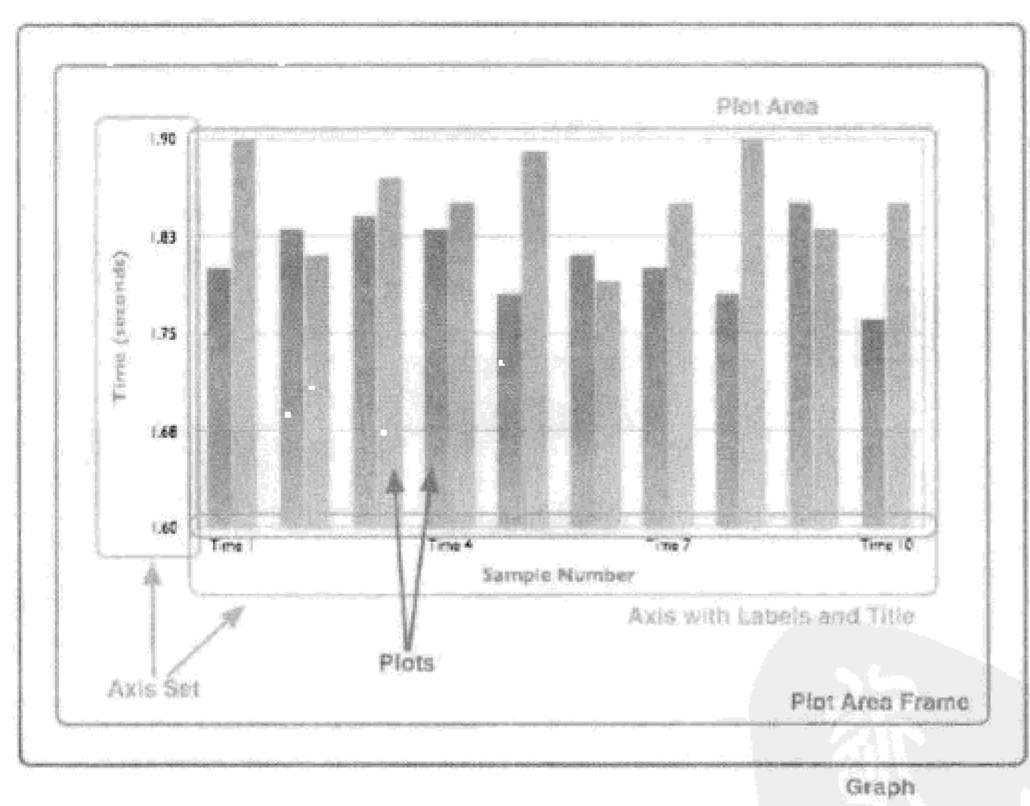


图 16-4 图表的构成

名词解释如下：

Graph——图纸，你可以把它想象成一张画布，上面承载了所有的东西。

Plot Area——绘图区，绘制各种数据的图形在这里显示，如折线、柱体和饼图等。注意，它可以是空白的，如果没有绘制任何图形的话。

Axis Set——坐标系，包括横、纵两个坐标。在横、纵坐标上往往还包括 Label（刻数）和 Title（标题）。标题（Title）是指坐标的名义表示，一个坐标只有一个标题，比如纵坐标为“销量”，横坐标为“销售季度”。Label 则是指坐标轴上刻度上的数字。当然 Label 也不一定是数字，也可以是文本：1 季度、2 季度、3 季度...等等。

Plot——图形，一个图形的数据应该是同一系列的（描述同一类型的数据）。比如说：一条连续曲线、一个饼图、一系列柱状图（虽然不是连续的），都可以称作一个图形。在一个绘图区上可以放多个图形（Plots），甚至是不同类型的图形（混合图表）。

Plot Area Frame——图框，Graph（图纸）除去四边留白就是图框，包括坐标轴和绘图区。

16.5 类图

Core Plot 的类图及其相互之间的关系如图 16-5 所示。

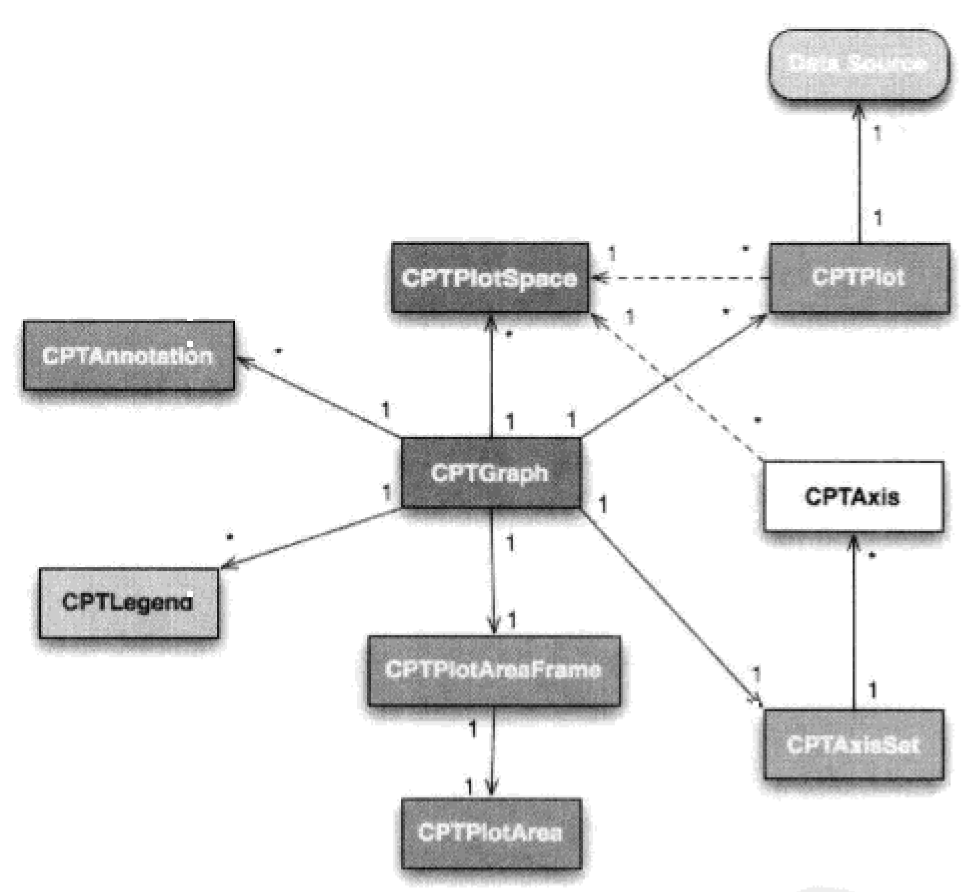


图 16-5 Core Plot 类图

箭头上面的数字和星号，代表了各实体之间的关系：“1 对 1 关系”、“1 对多关系”。

各类的简要说明如下：

CPTGraph 类——即代表了图 16-4 中的 Graph（图表、画布）。

CPTAxisSet 类和 CPTAxis 类——分别对应坐标系和坐标，即图 16-4 中的 Axis Set。

CPTPlotArea 类——对应绘图区，即图 16-4 中的 Plot Area。

CPTPlotAreaFrame 类——对应画框，即图 16-4 中的 Plot Area Frame。

PDG

CPTPlot 类——对应图形，即图 16-4 中的 Plot。

CPTPlotSpace 类——图形空间。这个比较特殊，定义图形的坐标系统。定义了从设备坐标到数据点坐标之间的映射。

CPTLegend 类——图例，每个图纸上都可能会有一个图例，用以说明图形中所用线型、颜色或图案所代表的意义。

CPTAnnotation 类——标注，在图纸上可以用标注对图形进行额外的说明。

16.6　使用 Core Plot 绘制折线图

我们先来介绍 Core Plot 中最简单的图形：折线图。折线图也叫散点图，图形由一个个的数据点和点之间的连接线段构成。

新建 Single View Application。使用前面介绍的 4 个步骤将 Core Plot 框架加到项目中。

打开 ViewController.xib，将 View 对象的 Identifier 改为 CPTGraphHostingView。

要使用 CorePlot，我们还必须在头文件中导入“CorePlot- CocoaTouch.h”。然后在 ViewController.h 文件中声明一个 CPTXYGraph 类型的成员：

```
CPTXYGraph *graph;
```

图 16-6　空白图表

我们将使用这个 CPTXYGraph 对象来显示一张空白的图纸，如图 16-6 所示。这很容易做到，在 viewDidLoad 方法中使用如下代码：

```
graph = [[CPTXYGraph alloc] initWithFrame:CGRectZero];
CPTTheme *theme = [CPTTheme themeNamed:kCPTDarkGradientTheme];
[graph applyTheme:theme];
CPTGraphHostingView *hostingView = (CPTGraphHostingView *)self.view;
   hostingView.hostedGraph = graph;
```

注意，我们代码的第 2 句，我们使用了一个名为 kCPTDarkGradientTheme 的 CPTTheme 来作为图表的主题。主题定义了图表的背景色和坐标轴样式。框架默认提供了 5 种样式，分别用字符串常量 kCPTDarkGradientTheme、kCPTPlainBlackTheme、kCPTPlainWhiteTheme、kCPTSlateTheme 和 kCPTStocksTheme 来指定。如果你觉得这些样式不够，可以定义自己的样式（后面介绍）。

第 3 句代码，则是将主题应用于图表。当然，你也可以不在图表上使用任何主题，这样的话你可能无法看到坐标系。因为默认的背景色和坐标轴同为黑色。

第 4 句代码，是将 self.view 转换为一个 CPTGraphHostingView 对象（本来就是，因为我们在 ViewController.xib 中设置过了）。

最后一句代码，将应用了主题的 CPTXYGraph 对象放到 CPTGraphHostingView 上，以便

图表能够绘制在窗体中。

我们先来构造一些数据，以便显示为散点图的形式。先在 ViewController 的头文件中声明一个可变数组：

```
NSMutableArray* points;
```

然后在 ViewController 的 initWithNibName 方法中，采用随机数初始化数组：

```
points=[[NSMutableArray alloc]init];
NSUInteger i;
for ( i = 0; i < 60; i++ ) {
     id x = [NSNumber numberWithFloat:1 + i * 0.05];
     id y = [NSNumber numberWithFloat:1.2 * rand() / (float)RAND_MAX + 1.2];
     [points addObject:[NSMutableDictionary dictionaryWithObjectsAndKeys:x, @"x",
            y, @"y", nil]];
 }
```

在 viewDidLoad 方法中，我们绘制散点图。在开始绘图之前，必须先定义绘图空间（CPTPlotSpace）。图形空间其实是图形在图纸上的二维坐标空间，对于一个图形都有一个自己的坐标空间，比如（1,1）在某个位置，（2,–1）在某个位置。每个图纸（CPTGraph）起码有一个图形空间（defaultPlotSpace）。如果图纸上有多个图形，这些图形则可能采用不同的图形空间，那么图纸就可能有不只存在一个图形空间。图形空间的坐标和设备空间的坐标不是一回事。CPTPlotSpace 类自身有一系列的方法用于在二者之间转换。

对于图形空间来说，最重要的事情莫过于设置 x、y 坐标的可见范围：从一个点开始，显示坐标轴上一段固定的长度。当然这只是说坐标轴一开始的样子。因为坐标轴是无限数轴，通过在屏幕上拖动，你可以滑动坐标轴。但不管怎么说，你能同时看到的坐标轴长度总是固定（不缩放的情况下）：

```
CPTXYPlotSpace *plotSpace = (CPTXYPlotSpace *)graph.defaultPlotSpace;
    plotSpace.allowsUserInteraction = YES;
    // 设置 x,y 坐标范围
    plotSpace.xRange = [CPTPlotRange plotRangeWithLocation:CPTDecimalFromFloat (1.0)
                       length:CPTDecimalFromFloat(2.0)];
    plotSpace.yRange = [CPTPlotRange plotRangeWithLocation:CPTDecimalFromFloat(1.0)
                       length:CPTDecimalFromFloat(3.0)];
```

然后才是散点图的绘制。在 Core Plot 中，散点图使用 CPTScatterPlot 类。散点图主要是由线段组成，因此线段的 Style（线型）就非常重要。CPTLineStyle 是用于描述线型的数据结构，但它是只读的，要想修改线段的 Style，我们必须使用它的可变形式 CPTMutableLineStyle。通过 CPTMutableLineStyle，我们可以修改线段的粗细、颜色、尖角限制。

```
CPTScatterPlot *boundLinePlot = [[[CPTScatterPlot alloc] init] autorelease];
CPTMutableLineStyle *lineStyle= [CPTMutableLineStyle lineStyle];
lineStyle.miterLimit = 1.0f;
```

```
lineStyle.lineWidth = 3.0f;
lineStyle.lineColor = [CPTColor blueColor];
```

所谓尖角限制就是两个线段连接处的肘部。尖角限制的算法在不同的系统有不同的解释，总的来说是把尖角从底到顶尖处横向等分为 4 份，尖角限制为几，我们就保留几份，剩下的部分截去。但是在有的系统里，尖角的计算方式是从线段下端算起（比如 Illustrate），有的系统则是把尖角下端从线段的中心线算起。假设 Core Plot 是使用第二种，则尖角限制可以用图 16-7 所示。

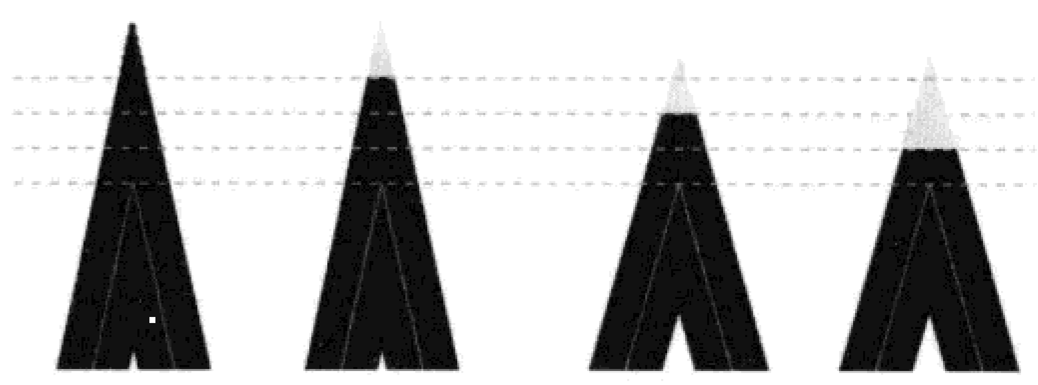


图 16-7　尖角限制从左至右依次为：4、3、2、1

然后将散点图的线型设置为 CPTMutableLineStyle 对象。设置散点图的数据源属性（dataSource），最后才将散点图对象加入到图纸中，就像我们使用 UIView addSubview 一样：

```
boundLinePlot.dataLineStyle = lineStyle;
boundLinePlot.identifier  = @"Blue Plot";
boundLinePlot.dataSource  = self;
[graph addPlot:boundLinePlot];
```

仅仅是将散点图的 dataSource 指定为 self 并不会绘制图形，还需要 self 去实现 CPTPlotDataSource 委托。跟 UITableView 需要 ViewController 实现 UITableViewDataSource 委托一样，CPTPlot 需要数据源实现 CPTPlotDataSource 委托才能呈现数据。首先在头文件中声明对 CPTPlotDataSource 委托的实现，然后在 implementation 中实现两个委托方法。

首先，要想绘制散点图，起码要知道需要画几个点吧？Core Plot 通过调用数据源对象（这里是 self）的 numberOfRecordsForPlot:方法来获取散点图的散点数，如以下代码所示：

```
-(NSUInteger)numberOfRecordsForPlot:(CPTPlot *)plot
{
    return [points count];
}
```

Core Plot 还需要知道每个数据点的具体值（x，y）。数据源对象可以用多个委托方法（self）来通知数据点的值，但 numberForPlot:field:recordIndex:方法是最常见的。Core Plot 通过该方法的三个参数来查询数值。第 1 个参数指定要绘制的图形对象（CPTPlot），第 2 个参数指定当前正在绘制的点的字段名（代表 x 坐标或 y 坐标），第 3 个参数表示正在绘制第几个点。

有了这些参数，我们自然可以返回数据源中对应的点的值。在我们的数据源 points 数组中，

每个点由两个字段“x”和“y”构成。我们根据方法的第 2 个参数返回 x 和 y 坐标。

```
-(NSNumber *)numberForPlot:(CPTPlot *)plot field:(NSUInteger)fieldEnum recordIndex:
            (NSUInteger)index
{
    NSString *key = (fieldEnum == CPTScatterPlotFieldX ? @"x" : @"y");
    NSNumber *num = [[points objectAtIndex:index] valueForKey:key];
    return num;
}
```

运行程序。现在，我们散点图终于可以显示了。但很遗憾，我们仍然看不见坐标轴。这并不是我们想要的样子，我们还需要对坐标轴进行一些设置，让它显示出来：

```
CPTXYAxisSet *axisSet = (CPTXYAxisSet *)graph.axisSet; //❶
CPTXYAxis *x = axisSet.xAxis; //❷
x.majorIntervalLength= CPTDecimalFromString(@"0.5"); //❸
x.orthogonalCoordinateDecimal = CPTDecimalFromString(@"2"); //❹
x.minorTicksPerInterval = 2; //❺
NSArray *exclusionRanges = [NSArray arrayWithObjects:
        [CPTPlotRange plotRangeWithLocation:CPTDecimalFromFloat(1.99)
                      length:CPTDecimal FromFloat(0.02)],
        [CPTPlotRange plotRangeWithLocation:CPTDecimalFromFloat(0.99)
                      length:CPTDecimal FromFloat(0.02)],
        [CPTPlotRange plotRangeWithLocation:CPTDecimalFromFloat(2.99)
                      length:CPTDecimal FromFloat(0.02)], nil]; //❻
x.labelExclusionRanges = exclusionRanges; //❼
```

代码说明：

❶ 获取图纸对象的坐标系；

❷ 获取坐标系的 X 轴坐标；

❸ 设置大刻度线的间隔为 0.5 个单位；

❹ 设置 X 坐标的原点（Y 轴将在此与 X 轴相交）；

❺ 设置小刻度线的间隔为每两个大刻度线之间分布有 2 个小刻度线；

❻❼ 这两句将 X 轴上的某些点排除（既不显示数字也不显示刻度线）。

接下来 Y 轴也做同样的事情（把上面的代码复制一遍再略做修改），只不过这次我们设置的是坐标系的 Y 坐标轴。

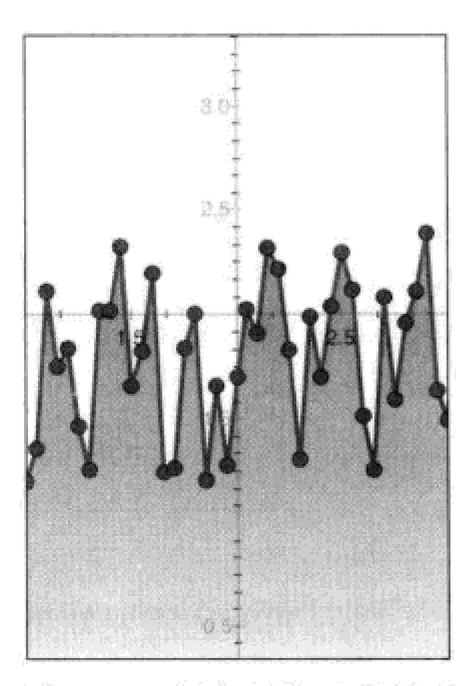


图 16-8 程序最终运行效果

运行程序，我们可以看到程序运行的效果，如图 16-8 所示。当然，我们还做了一些额外的工作，比如在数据点上绘制了一个蓝色圆圈，在折线下方显示了一个渐变色层等。限于篇幅，这些代码就不一一介绍了，你可以自行参考位于光盘“source/第 16 章/CorePlotScatterTest”的示例程序代码。

PDG

16.7　使用 Core Plot 绘制柱状图

Core Plot 提供了 CPTBarPlot 类用于柱状图的绘制。在 1.0 版本中，Core Plot 终于提供了水平柱状图（如果你要绘制水平柱状图，[CPTBarPlot tubularBarPlotWithColor: horizontalBars:] 方法的第 2 个参数设置为 YES）。

在光盘"source/第 16 章/CorePlotBarTest"目录中，有一个示例项目，在这个程序中我们演示了柱状图的绘制，如图 16-9 所示。

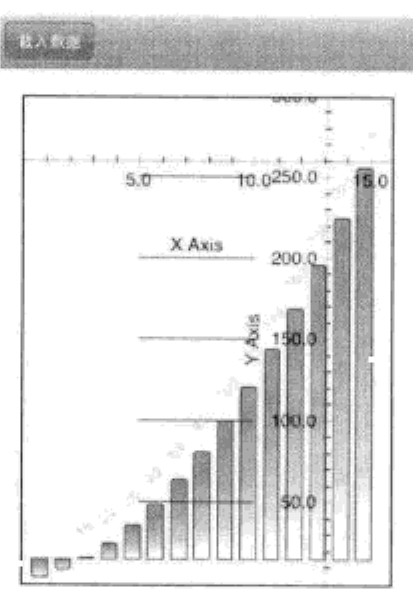


图 16-9　示例程序 CorePlotBarTest

16.7.1　绘制基本的柱状图

在这个程序中，数据点是动态加载的，当你点击工具栏按钮"载入数据"时，柱状图会逐渐显现，每隔一秒添加一点数据。这里我们用到了定时器源：

```
- (IBAction)loadData:(id)sender {
    points=[[NSMutableArray alloc]init];
if(timer) dispatch_source_cancel(timer);
    timer = dispatch_source_create(DISPATCH_ SOURCE_TYPE_TIMER, 0, 0, dispatch_get_
          global_queue (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0));
    __block int i=0;
    dispatch_source_set_timer(timer, DISPATCH_TIME_NOW, 1ull * NSEC_PER_SEC, 0);
    dispatch_source_set_event_handler(timer, ^{
        i++;
        if (i<max_bar_num) {
            id x = (NSDecimalNumber *)[NSDecimalNumber numberWithUnsignedInteger:i];
            id y = (NSDecimalNumber *)[NSDecimalNumber numberWithUnsignedInteger:(i
                  + 1) * (i + 1)];
            NSDictionary* point=[NSDictionary dictionaryWithObjectsAndKeys:x, @"x",
                  y, @"y", nil];
            [self addPoint:point];
        }else{
```

```
            NSLog(@"timer stop!");
        dispatch_source_cancel(timer);
        }
    });
    dispatch_resume(timer);
}
```

同散点图的例子一样，在绘制图形之前，我们需要配置图纸、图形空间、坐标系和 CPTPlot 的属性，我们把代码放到了 viewDidLoad 方法里。这些代码有一点模式化了，你可以参考散点图的例子。

其中，CPTBarPlot 实例由唯一工厂方法获得：tubularBarPlotWithColor:horizontalBars:

第一个参数指定柱子的颜色，第二个参数指定是否绘制水平柱状图，YES 为绘制。默认为 NO。

实例化 CPTBarPlot 之后，需要设置 CPTBarPlot 的属性，常见属性如下：

- baseValue 属性——柱子的基线值。大于这个值以上的点，柱子只从这个点开始画。小于此值的点，则是反向绘制的，即从基线值向下画，一直画到到数据点。
- barWidth 属性——柱子宽度。
- barWidthScale 属性——柱宽的缩放系数。
- barOffset 属性——柱子开始绘制的偏移位置。这个值非常有用。由于在横坐标上占据一定宽度，所以如果柱子从 x 坐标某个值开始画的话，柱子的中线就不在数据点上了。为了保持柱子中心线和数据点对齐，我们一般会把 barOffset 设置为-barWidth/2。
- dataSource 属性——这个当然是用来设置数据源的。数据源必须实现 CPTPlotDataSource 委托。
- fill 属性——这个属性用来设置柱子的填充风格（CPTFill）。如果设置为 nil，则柱子不会填充。当然我们也可以用渐变色填充。
- lineStyle 属性——用于设置柱子外框的线型。

设置完 CPTBarPlot 的属性，我们需要在数据源实现 CPTPlotDataSource 委托方法。这些方法的实现同前面我们讲过的散点图的例子是一模一样的。

16.7.2 固定坐标轴

从 0.9 版本开始，Core Plot 改变了 Constraints 机制。结构体 CPConstraints 被新的 CPTConstraints 类所取代，同时也取消了 CPTMakeConstraints()函数的使用。

通过 CPTConstraints 类，我们可以指定坐标轴始终固定在屏幕的某个地方。将 CPTXYAxis 的 axisConstraints 属性用一个 CPTConstraints 对象指定，可以在拖动发生时使 X 或 Y 轴不跟随图形移动。

CPTConstraints 有三个重要的工厂方法，用于创建 CPTConstraints 实例：

- constraintWithLowerOffset:
- constraintWithRelativeOffset:
- constraintWithUpperOffset:

三个方法用于计算偏移量的方式略有不同，具体区别如下：

1．constraintWithLowerOffset:

根据指定的偏移量（float），可以将坐标轴固定于图框(PlotFrame)近端的偏移量处。lower 一词实际上是以 X 轴坐标代入的，意指屏幕下端。如果是 Y 轴，则是指屏幕左端。偏移量以像素点为单位。偏移量为 50 的效果如图 16-10 所示。

2．constraintWithUpperOffset:

与上面相反，将坐标轴固定于图框远端的偏移量处。upper 一词对于 X 轴是指屏幕上端，对于 Y 轴是指屏幕右端。偏移量为 50 的效果如图 16-11 所示。

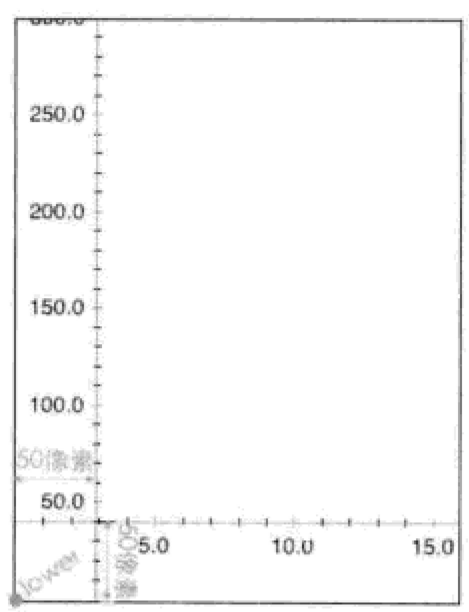


图 16-10　X、Y 轴同时用 constraintWithLowerOffset:50 的效果

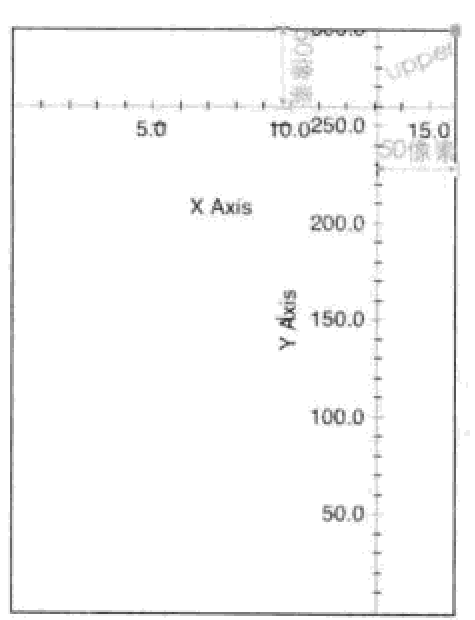


图 16-11　X、Y 轴同时用 constraintWithUpperOffset:50 的效果

3．constraintWithRelativeOffset:

与上面两者不同，此偏移量以 0～1 之间的相对比例计算（从近端到远端）。0 代表最左/下边，1 代表最右/上边。0.5 代表中间。偏移量为 0.3 的效果如图 16-12 所示。

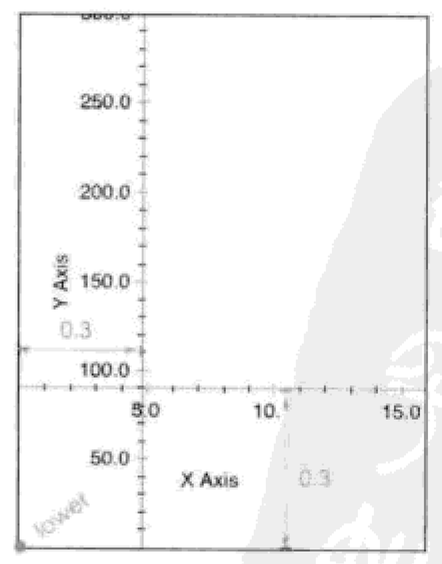


图 16-12　X,Y 轴同时用 constraintWithRelativeOffset:0.3 的效果

16.7.3 显示数据点的值

有时候需要在曲线或柱子的顶点上显示数字（即 Data Label），如图 16-13 所示。

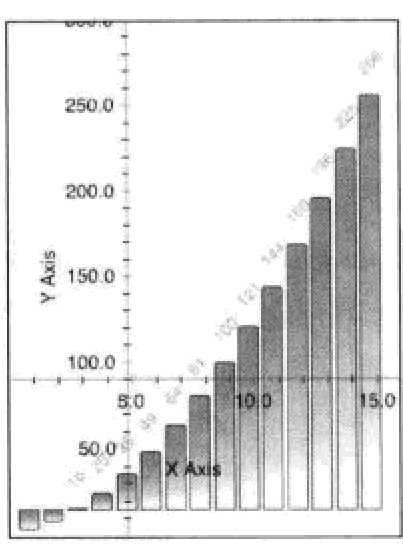


图 16-13　在图形中显示 Data Label

CPTPlot 提供默认的 Data Label，通过对一些属性的改变，你可以控制 Data Label 是否显示及显示的样式，其属性如下所示：

- labelOffset：Data Label 显示的位置到顶点之间的距离。
- labelRotation：Data Label 的倾斜角。
- labelTextStyle：Data Label 的文字样式，包括字体名、颜色、字体大小。
- labelFormatter：可以用于设置 Data Label 的数字格式，提供一个 NSNumberFormatter 对象给这个属性。
- labelShadow: Data Label 的阴影属性。

要想让默认的 Data Label 能够显示，起码要配置 labelTextStyle 属性：

```
CPTMutableTextStyle* textStyle=[CPTMutableTextStyle textStyle];
textStyle.color=[CPTColor greenColor];
textStyle.fontSize=10;
barPlot.labelTextStyle=textStyle;
```

如果想提供自定义的 Data Label，则需要实现 CPTPlotDataSource 中的 dataLabelForPlot:recordIndex:方法。以下代码演示在 dataLabelForPlot:recordIndex:方法中，我们对 Data Label 中的数字格式和文本颜色进行了定制：

```
- (CPTLayer *) dataLabelForPlot:(CPTPlot *) plot recordIndex: (NSUInteger) index{
    NSNumber* num=[[points objectAtIndex:index] valueForKey:@"y"];
    CPTTextLayer *label = [[CPTTextLayer alloc] initWithText:[NSString stringWith-
        Format:@"%d", [num intValue]]];
     CPTMutableTextStyle *textStyle = [label.textStyle mutableCopy];

     textStyle.color = [CPTColor lightGrayColor];
     label.textStyle = textStyle;
```

```
    [textStyle release];
    return [label autorelease];
}
```

16.7.4　显示网格线

网格线是大小刻度线的延伸。通过设置坐标轴的下列属性，我们可以在图形坐标轴上显示网格线：

- majorGridLineStyle：指定大刻度线上的网格线样式。为空则不显示网格线。
- minorGridLineStyle：指定小刻度线上的网格线样式。为空则不显示网格线。
- gridLinesRange：指定网格线的显示范围，包括网格线的起始及长度。

以下代码将在 Y 坐标的大刻度线上显示网格线，并指定网格线线型和线段的范围：

```
CPTXYAxis *y = axisSet.yAxis;
CPTMutableLineStyle *lineStyle1 = [CPTMutableLineStyle lineStyle];
lineStyle1.lineWidth = 1.0f;
lineStyle1.lineColor = [CPTColor blueColor];
barPlot.lineStyle=lineStyle1;
y.majorGridLineStyle = lineStyle1;
CPTPlotRange* range=[CPTPlotRange plotRangeWithLocation:CPTDecimalFromString(@"5")
                                         length:CPTDecimalFromString(@"5")];
y.gridLinesRange=range;
```

以上代码实际效果如图 16-14 所示。

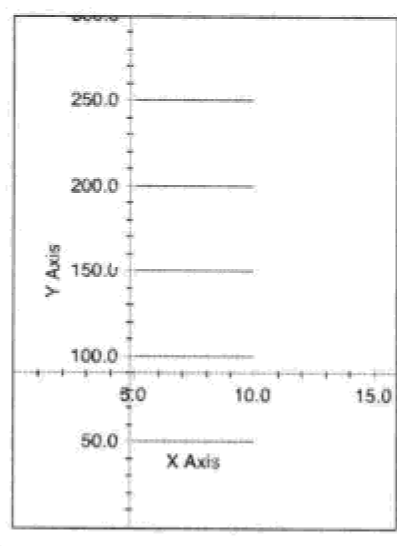


图 16-14　在图形中绘制网格线

16.8　使用 Core Plot 绘制饼图

在 Core Plot 中，饼图是一类特殊的图形，因为它不需要显示坐标轴。数据不以坐标象限内的点表示，而以椭圆中的扇形面积表示。

示例程序 CorePlotPieTest 演示了饼图的绘制，位于光盘“source/第 16 章/CorePlotPieTest”目录下。

16.8.1 饼图的绘制

Core Plot 用 CPTPieChart 对象代表一个饼图。在 CPTGraph 中添加一个饼图很简单，以下代码向 pieChart 中绘制了一个饼图：

```
CPTPieChart *piePlot       = [[CPTPieChart alloc] init];
piePlot.dataSource         = self;
piePlot.pieRadius          = 131.0;
piePlot.identifier         = @"Pie Chart 1";
piePlot.startAngle         = M_PI_4;
piePlot.sliceDirection     = CPTPieDirectionCounterClockwise;
piePlot.centerAnchor       = CGPointMake(0.5, 0.38);
piePlot.borderLineStyle    = [CPTLineStyle lineStyle];
piePlot.delegate           = self;
[pieChart addPlot:piePlot];
[piePlot release];
```

从代码中可见，alloc、init 一个 CPTPieChart 对象后，需要设置它的一些属性并在 CPTGraph 中添加图表。CPTPieChart 的属性很多，上述代码使用到了如下属性：

- dataSource 属性——指定饼图的数据源。数据源必须实现 CPTPieDataSource 委托。
- pieRadius 属性——饼图的半径。
- startAngle 属性——第 1 片扇形的起始角度，默认是 PI/2。
- sliceDirection 属性——扇形绘制的方向：正时针、反时针。
- centerAnchor 属性——饼图的重心（旋转时以此为中心）坐标（x,y），以相对于饼图直径的比例表示（0～1）之间。默认和圆心重叠（0.5,0.5）。
- borderLineStyle 属性——饼图边线的样式。
- delegate 属性——指定饼图的事件委托。委托必须实现 CPTPieChartDelegate 中定义的方法。

最重要的是数据源方法。我们假设用数组 model 定义了扇形的分布比例，则 CPTPieChart-DataSource 委托可实现如下：

```
-(NSUInteger)numberOfRecordsForPlot:(CPTPlot *)plot
{
    return [model count];
}
-(NSNumber *)numberForPlot:(CPTPlot *)plot field:(NSUInteger)fieldEnum recordIndex:
          (NSUInteger)index
{
    return [model objectAtIndex:index];
 }
```

numberOfRecordsForPlot：方法是重要的，它告诉 Core Plot 需要绘制几个数据点。当然对于每种图形来说，绘制数据点的方式不一样。对于散点图，每个数据点会被绘制为节点，对于柱状图，每个数据点会被绘制为柱子，而对于饼图，每个数据点用不同弧度的扇形表示。

在 Core Plot 具体绘制图形时，它根据数据源方法返回的数据点数来绘制每一个点。每绘制一个点都会来调用 numberForPlot:field:recordIndex:方法，但调用的次数不一。因为每种图形在描述每个点时需要的数据不一样。比如柱状图和散点图，每个点以二维表示（x,y），则 Core Plot 对每个点都会调用两次 xxxForPlot:field:recordIndex: 方法。每次调用时，依靠 field 参数来区分是请求 X 坐标还是 Y 坐标（recordIndex 参数则指定是第几个数据点）。

而对于饼图，每个点只需要一个数字描述（百分比），因此只会调用一次 xxForPlot:field:recordIndex 方法。我们直接根据索引（index）返回 model 数组中的数值就可以了。

16.8.2　显示每个扇形的比例

默认情况下，饼图不显示 Data Label。如果要显示 Data Label，需要实现数据源的 dataLabelForPlot: recordIndex: 方法。以下代码在每个扇形的 Data Label 上显示百分比数字：

```
-(CPTLayer *)dataLabelForPlot:(CPTPlot *)plot recordIndex:(NSUInteger)index
{
    float f=((NSNumber*)[model objectAtIndex:index]).floatValue;
    CPTTextLayer *label  = [[CPTTextLayer alloc] initWithText:[NSString stringWithFormat:
                @"%.1f%%",f]];
    CPTMutableTextStyle *textStyle = [label.textStyle mutableCopy];
    textStyle.color = [CPTColor lightGrayColor];
    label.textStyle = textStyle;
    [textStyle release];
    return [label autorelease];
}
```

16.8.3　剥离扇形

对于饼图，我们可以把某块扇形“切除”下来，以此突出该扇形区域。这需要实现数据源方法 radialOffsetForPieChart:recordIndex:方法。以下代码将饼图中第 2 块扇形“剥离”10 个像素点：

```
-(CGFloat)radialOffsetForPieChart:(CPTPieChart *)piePlot recordIndex:(NSUInteger)
         index{
return (indexix==1?10:0);
}
```

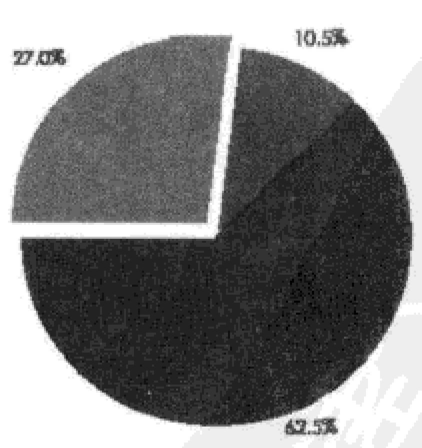


图 16-15　扇形的“剥离”

16.8.4 显示图例

图例可用于描述图形（尤其是饼图）的构成部分，以下代码显示了饼图的图例：

```
CPTLegend *theLegend = [CPTLegend legendWithGraph: pieChart];
    theLegend.numberOfColumns = 1;
    theLegend.fill= [CPTFill fillWithColor:[CPTColor lightGrayColor]];
    theLegend.borderLineStyle = [CPTLineStyle lineStyle];
    theLegend.cornerRadius= 5.0;
    pieChart.legend = theLegend;
    pieChart.legendAnchor= CPTRectAnchorTopRight;
    pieChart.legendDisplacement = CGPointMake(-30.0, -30.0);
```

图例由 CPTLengend 对象表示。上述代码中，分别设置了 CPTLengend 的如下属性：

- numberOfColumns 属性——图例的列数。有时图例太多，单列显示太长，可分为多列显示。
- fill 属性——图例的填充属性，CPTFill 类型。
- borderLineStyle 属性——图例外框的线条样式。
- cornerRadiuss 属性——图例外框的圆角半径。

把一个图例对象赋值给图形的 legend 属性，即可在绘制图形时加上图例。此外图形对象还有两个和图例相关的重要属性：

- legendAnchor 属性——图例对齐于图框的位置，可以用 CPTRectAnchor 枚举类型，指定图例向图框的 4 角、4 边（中点）对齐。
- legendDisplacement 属性——图例对齐时的偏移距离。注意，对齐时是使用图例的相同锚点和图框的相同锚点对齐。比如，图形的 legendAnchor 属性是右上角（CPTRectAnchorTopRight），则对齐时用图例的右上角和图框的右上角进行对齐。这个偏移坐标（x,y）就是这两个锚点之间的距离（用图例的锚点坐标减图框的锚点坐标）。默认 legendDisplacement 为（0,0）。注意这个值的正负关系，比如（–30, –30）和（30,30）是截然相反的，你多试几次就明白了。

图例上的描述文字，从数据本身是看不到的，默认情况下 CorePlot 会以“Pie Chart 1”、“Pie Char 2”来命名。

我们需要实现数据源方法 legendTitleForPieChart: recordIndex:来覆盖默认的图例名称：

```
-(NSString *)legendTitleForPieChart:(CPTPieChart *)pieChart recordIndex: (NSUInteger)
        index{
    switch (index) {
      case 0:
          return @"其他";
      case 1:
          return @"Google Android";
      case 2:
```

图 16-16 显示图例

```
            return @"Apple iOS";
        default:
            return nil;
    }
}
```

16.8.5 响应事件

通过 CPTPieChartDelegate 协议，我们可以响应饼图的事件。CPTPieChartDelegate 协议只定义了一个唯一的委托方法：

```
pieChart:sliceWasSelectedAtRecordIndex:
```

你可以使用这个方法来响应扇形切片的触摸事件：

```
-(void)pieChart:(CPTPieChart *)plot sliceWasSelectedAtRecordIndex:(NSUInteger)index
{
    [self.view bringSubviewToFront:label];
    float f=((NSNumber*)[model objectAtIndex:index]).floatValue;
    NSString *text = [self legendTitleForPieChart:plot recordIndex:index];
    text=[NSString stringWithFormat:@"%@:%.1f%%", text,f];
    label.text=text;
    label.alpha=0;
    [CATransaction begin];
    CGAffineTransform textTransform = CGAffineTransformMake(1.0, 0.0, 0.0, -1.0,0.0,
                                                 0.0);
    [label.layer setAffineTransform:textTransform];
    CABasicAnimation *animation = [CABasicAnimation animationWithKeyPath:@"opacity"];
    animation.duration = 1.0f;
    animation.removedOnCompletion = YES;
    animation.fillMode = kCAFillModeForwards;
    animation.toValue = [NSNumber numberWithFloat:1.0];
    [label.layer addAnimation:animation forKey:@"OutEase"];
    [CATransaction commit];
}
```

上述代码会在你点击扇形切片时会显示一个动画效果。

16.9 自定义 Core Plot 主题

一个主题，描述了一个图表预定义样式。在 Core Plot 中用 CPTTheme 类来表示。Core Plot 提供了 5 种内置的主题，这些主题可以直接通过 CPTGraph 的 applyTheme 方法应用到图表中。

除此之外，我们可以定制自己的 Core Plot 主题。自定义 Core Plot 主题需要继承 CPTTheme 类，并覆盖它的 4 个方法：

- -(void)applyThemeToBackground:(CPTGraph *)graph;

- -(void)applyThemeToPlotArea:(CPTPlotAreaFrame *)plotAreaFrame;
- -(void)applyThemeToAxisSet:(CPTAxisSet *)axisSet;
- - (void) applyThemeToGraph: (CPTGraph *) graph;

这些 applyThemeToXXX 方法都是类似的，只不过提供了不同类型的参数。你可以利用这些参数来配置不同图表对象：图纸、绘图框、坐标系。

提示：applyThemeToGraph:方法其实是用于代替过去的 applyThemeToBackground:方法，它们都使用相同的参数。为了和之前的版本保持兼容，SDK 中保留了老的 applyThemeToBackground:方法。

下面就来演示如何扩展 CPTTheme 类以自定义主题。我们首先新建一个类 MyTheme，继承 CPTTheme。然后在类中分别实现这 3 个方法。

首先是 applyThemeToBackground 方法，通过这个方法，我们为图纸背景上设置了一个自定义颜色渐变效果：

```
-(void)applyThemeToBackground:(CPTXYGraph *)graph
{
    CPTColor *endColor = [CPTColor colorWithGenericGray:0.2f];// ❶
    CPTGradient *graphGradient = [CPTGradient gradientWithBeginningColor:endColor
                              endingColor:endColor];// ❷
    graphGradient = [graphGradient addColorStop:[CPTColor colorWithGenericGray:
                              0.3f] atPosition:0.3f];// ❸
    graphGradient = [graphGradient addColorStop:[CPTColor colorWithGenericGray:
                              0.5f] atPosition:0.5f];// ❹
    graphGradient = [graphGradient addColorStop:[CPTColor colorWithGenericGray:
                              0.3f] atPosition:0.6f];// ❺
    graphGradient.angle = 90.0f;// ❻
    graph.fill = [CPTFill fillWithGradient:graphGradient];// ❼
}
```

代码说明：

❶ 颜色渐变是多个颜色进行平滑过渡的结果。因此需要设置多个颜色。这句代码设置了一个 20% 灰度的颜色，该颜色将作为颜色渐变的开始色和终点色。

❷ Core Plot 提供了一个 CPTGradient 对象，它可以很方便地创建颜色渐变，因此不需要我们去使用比较麻烦的 Quartz API。这句代码创建了一个 CPTGradient 对象，并使用前面的 CPTColor 颜色作为开始色和终点色。

❸❹❺ CTGradient 有一个 addColorStop:方法，可以很方便地为颜色渐变效果中添加任意的过渡色。这 3 句代码一口气设置了 3 个颜色渐变的过渡色。

❻ 还可以设置渐变效果的方向。这句代码将渐变方向设置为垂直（逆时针）90 度向上。

❼ 最后用渐变色填充 CPTGraph。

接下来是 applyThemeToAsisSet 方法。通过这个方法，可实现在 Y 轴上添加 X 轴平行线的效果。

```
-(void)applyThemeToAxisSet:(CPTXYAxisSet *)axisSet {
    CPTMutableLineStyle *majorGridLineStyle = [CPTMutableLineStyle lineStyle];// ❶
    majorGridLineStyle.lineWidth = 1.0f;// ❷
    majorGridLineStyle.lineColor = [CPTColor lightGrayColor];// ❸

    CPTXYAxis *axis=axisSet.yAxis;// ❹
    axis.tickDirection = CPTSignNegative;// ❺
    axis.majorGridLineStyle = majorGridLineStyle ;// ❻
    //axis.labelingPolicy = CPAxisLabelingPolicyNone ;
}
```

代码说明：

❶❷❸ 首先创建了一个线型 CPTLineStyle 对象，并定义其线宽为 1，颜色为亮灰色。

❹ 获得 Y 轴对象。

❺ 设置 Y 轴轴标签（tick）显示方向。这里我们将 Y 轴的轴标签显示在 X 轴的负轴方向（即左边）。

注意： 如果 tickDirection 设置为 CPTSignNegative，表示标签位于另外一根轴的负轴方向；CPTSignPositive 则位于另一根轴的正轴方向。以 Y 轴的轴标签为例，设置为 CPTSignPosition，表示 Y 轴标签将位于 X 轴的正轴方向，即 Y 轴右边；为 CTSignNegative 则表示 Y 轴标签位于 X 轴的负轴方向，即 Y 轴的左边；默认值是 CPTSignNone，等同于 CPTSignNegative。

---

❻ CPTXYAxis 对象有一个 majorGridLineStyle 属性，该属性是 CPTLineStyle 类型。如果为 majorGridLineStyle 指定一个线型对象，则 Core Plot 自动为坐标轴添加平行线。这里我们在 Y 轴上设置了一系列与 X 轴平行的平行线。

注意： X 轴平行线将添加在 Y 轴的每个大刻度线的位置。另外，如果将坐标轴的 labelingPolicy 属性设置为 CTPAxisLabelingPolicyNone，则 majorGridLineStyle 属性将不起作用。

最后是 applyThemeToPlotArea 方法，下面的代码在绘图区（PlotArea）填充一个灰色渐变：

```
-(void)applyThemeToPlotArea:(CPTPlotAreaFrame *)plotAreaFrame
{
    CPTGradient *gradient = [CPTGradient gradientWithBeginningColor:[CPTColor
    colorWithGenericGray:0.2f] endingColor:[CPTColor colorWithGenericGray:0.7f]];
    // 我们用 CPTGradient 创建了一个渐变颜色效果，从 20%的灰度过渡到 70%灰度。
    gradient.angle = 45.0f;// 设置渐变色过渡方向为逆时针 45 度。
    plotAreaFrame.fill = [CPTFill fillWithGradient:gradient]; //然后用此渐变色填充
        绘图区（CPTPlotAreaFrame）。
}
```

注意： 由于绘图区（PlotArea）位于背景图（Graph）的上层，因此对于绘图区所做的设置会覆盖对 Graph 所做的设置，除非你故意在 Graph 的 4 边留白，否则看不到背景图的设置。

下面来看看如何使用自定义主题 MyTheme。

其实很简单，用 MyTheme 初始化一个 CPTheme 对象，并应用到 CPGraph：

```
#import "MyCPTheme.h"
...
CPTheme *theme=[[MyCPTheme alloc]init];
[graph applyTheme:theme];
```

完整的代码请参考光盘中的"source/第 16 章/CorePlotCustomeThemeTest"示例项目。

16.10 本章小结

本章介绍了 iOS 下一个杰出的 2D 图形框架 Core Plot，包括它的安装、设置和使用。

本章从浅入深逐步介绍 Core Plot 1.0 版本的最新特性，介绍开发者如何使用 Core Plot SDK 开发图表应用程序，包括如何设置和自定义图表样式，如何设置坐标轴，如何显示渐变区、数据标签和图例，以及绘制最常见的图形：折线图、柱状图和饼图。

Core Plot 在基于 Quartz 2D 框架之上搭建了一个高级的、对程序员更加友好的图形 API。尤为难得的是，它还是一个开源框架。实践的结果证明，Core Plot 比 Quartz 框架更加适合于企业图形图表应用程序的开发。

第 17 章　通讯簿、GPS 和重力感应

本章介绍通讯簿、GPS 和重力感应，包括：

- 通讯簿框架：这部分内容介绍 Cocoa 的通讯簿框架，包括 AddressBook 和 AddressBookUI 两个子框架。
- Core Location 框架：这部分内容介绍 Core Location 框架和地理编码服务。通过 Core Location 框架提供的功能，iPhone 程序员能很容易地获取 iPhone 的 GPS 定位信息。
- 重力感应：这部分内容介绍 UIAccelerometer。通过 UIAccelerometer 的唯一委托方法，我们可以精确地获得 iPhone 在三维空间中的移动数据和重力加速度，并通过一个小示例程序，演示了如何将加速计数据成功地识别为摇动手势。

17.1　通讯簿

SDK 提供了对 iPhone 通讯簿的支持，这样使得应用程序也具备了访问 iPhone 联系人和电话号码的能力。与通讯簿相关的框架有两个：AddressBook.framework 和 AddressBookUI.framework，下面分别介绍。

17.1.1　Address Book UI

AddressBookUI 中提供了用于显示联系人信息的相关控制器，共有以下四个：

- ABPeoplePickerNavigationController：显示整个通讯簿并可以选择一个联系人。
- ABPersonViewController：显示一个具体联系人的信息。
- ABNewPersonViewController：增加一个新的联系人。
- ABUnknownPersonViewController：完善一个联系人的信息。

其中最主要的是 ABPeoplePickerNavigationController，下面就具体介绍一下如何浏览整个通讯簿，并让用户查看其中某个联系人的步骤：

```
ABPeoplePickerNavigationController *picker =
    [[ABPeoplePickerNavigationController alloc] init];  // ❶
    picker.peoplePickerDelegate = self;  //❷
    [self presentModalViewController:picker animated:YES];  //❸
    [picker release];
```

代码说明：

❶ 初始化 ABPeoplePickerNavigationController。

❷ 然后设置它的委托对象（ABPeoplePickerNavigationControllerDelegate 协议的实现者）为 self。

PDG

❸ 呈现 ABPeoplePickerNavigationController。

协议 ABPeoplePickerNavigationControllerDelegate 里主要包含了以下 3 个方法：

- peoplePickerNavigationControllerDidCancel:方法：该方法在用户取消通讯簿页面时调用。
- peoplePickerNavigationController:shouldContinueAfterSelectingPerson:方法：该方法在用户选择了通讯簿中某个联系人时调用，返回值为 BOOL 类型，返回 NO 表示取消选择，返回 YES 则允许用户查看该联系人的具体细节。
- peoplePickerNavigationController:shouldContinueAfterSelectingPerson:property:identifier:方法：这个方法是在"联系人细节"页面中，当用户点击了联系人的某一项信息（比如手机号码、Email 地址）时调用。该方法返回值为 BOOL 类型，返回 YES 表示将对该项信息（比如手机号码、Email 地址）进行具体操作，如拨打号码或发送邮件。返回 NO 表示不进行任何操作。

17.1.2 Address Book

AddressBook 提供了通讯簿框架中除了 UI 之外的其他功能，包括所有访问通讯簿数据库的函数和方法，下面介绍几个重要方法。

```
CFTypeRef ABRecordCopyValue (
   ABRecordRef record,
   ABPropertyID property
);
```

该函数从联系人记录中读取某个字段（属性）。record 参数为 ABRecordRef 类型，代表了一条通讯簿数据库中的联系人记录。注意，在 ABPeoplePickerNavigationControllerDelegate 协议的 peoplePickerNavigationController:shouldContinueAfterSelectingPerson:方法中，其第 2 个参数会返回一个 ABRecordRef 参数，这样你可以通过这个方法得知用户所选择的联系人记录。然后使用 ABRecordCopyValue 函数即可读取到联系人记录的某个具体字段。property 参数为 ABPropertyID 类型，是以 kABPerson 开头的枚举常量，即要访问的字段名，包括名称、地址、图片等，具体信息请参考 ABPerson.h 中的常量声明。

```
CFStringRef ABRecordCopyCompositeName (
   ABRecordRef record
);
```

该函数返回联系人的完整姓名。

```
CFTypeRef ABMultiValueCopyValueAtIndex (
   ABMultiValueRef multiValue,
   CFIndex index
);
```

ABMultiValueCopyValueAtIndex 函数通过索引获取联系人记录中某个集合属性中的成员。

ABRecordRef 类型中的属性（字段）可能是单个值，也可能是多个值集合。对于单个值的属性（多数情况），我们使用 ABRecordCopyValue 函数访问；而对于集合属性，就要用到 ABMultiValueCopyValueAtIndex 函数了。例如联系人电话，一个联系人可能有多个联系电话（手机、座机、办公室、家等），这样要访问该联系人的某个电话而不是所有电话，就需要使用这个函数。

```
CFStringRef ABMultiValueCopyLabelAtIndex (
  ABMultiValueRef multiValue,
  CFIndex index
);
```

对于 ABRecordRef 的集合属性，它其实是“键－值”对集合。除了用索引访问集合属性中每个成员的“值”以外，我们还可以使用 ABMultiValueCopyLabelAtIndex()函数访问成员的“键名”属性。

```
CFIndex ABMultiValueGetCount (
  ABMultiValueRef multiValue
);
```

该函数获取集合属性中的成员个数。

以下代码演示了 AddressBook 框架中各函数的使用：

```
name.text = (NSString*)ABRecordCopyCompositeName(person);  // ❶
ABMutableMultiValueRef phoneMulti = ABRecordCopyValue(person, kABPersonPhone
        Property);  //❷
NSMutableArray *phones = [[NSMutableArray alloc] init];
int i;
for (i = 0; i < ABMultiValueGetCount(phoneMulti); i++) {  // ❸
    NSString *aPhone = [(NSString*)ABMultiValueCopyValueAtIndex(phoneMulti, i)
    autorelease];
    NSString *aLabel = [(NSString*)ABMultiValueCopyLabelAtIndex(phoneMulti, i)
    autorelease];
    NSLog(@"PhoneLabel:%@ Phone#:%@",aLabel,aPhone);
    if([aLabel isEqualToString:@"_$!<Mobile>!$_"])
    {
        [phones addObject:aPhone];
    }
}
```

代码说明：

❶ 我们用 ABRecordCopyCompositeName 函数获得联系人的全称（first name 和 last name）；

❷ 然后通过 ABRecordCopyValue 访问这个联系人的电话属性（即指明为 kABPersonPhoneProperty 的字段），由于这是一个集合属性，我们得到的会是一个 ABMutableMultiValueRef（多值对象）。

❸ 接下来我们对这个集合属性进行遍历，并将集合中的每个元素的名称和值分别用 ABMultiValueCopyValueAtIndex 和 ABMultiValueCopyLabelAtIndex 函数读取出来，分别打印。同时将电话号码加到了可变数组 phones 里。

关于地址簿框架和地址簿 UI 框架的使用，我们就简单介绍到这里。完整示例程序，请读者参考苹果官方的 demo，这是一个极好的教程，下载地址为：

http://developer.apple.com/iphone/library/samplecode/QuickContacts/QuickContacts.zip。

17.1.3 联系人中文姓氏排序

通过通讯簿框架可以很方便地读取联系人信息，但这些数据是未排序的，我们需要自己实现排序功能。对于中文姓名，这是个很大的问题。因为中文字符本身并不是有序的，我们对于中文的排序通常是用它的其他信息来进行的，比如拼音或笔画。拼音是字母，笔画是数字——虽然后者比较罕见，但它们的排序却是比较容易的。

问题的产生并不是排序本身。以拼音排序为例，我们面临的第一个挑战是，如何将每个中文字符映射为拼音字母？当然对于仅仅是用于排序而言，我们并不需要转换整个汉字为汉语拼音，而只需要转换拼音的第一个字母就可以了。

幸好有人为我们完成了这个工作，它基于以下原理：

在 unicode 字符集中，汉字的编码范围为 4E00 到 9FA5 之间（即从第 19968 开始的 20902 个字符是中文简体字符）。我们把这些字符的拼音首字母按照顺序都存放在一个 char 数组中。当需要查找一个汉字的拼音首字母时，只需把这个汉字的 unicode 码（即 char 强制转换为 int ）减去 19968，然后用这个数字作为索引去找 char 数组中存放的字母即可。

因此，这就有了一个 firstLetterArray 数组和一个 pinyinFirstLetter() 函数。firstLetterArray 数组囊括了所有简体中文字符（20902 个）的拼音首字母，而 pinyinFirstLetter 函数则将一个汉字字符所对应的首字母返回。在此基础上，我又增加了一个遍历函数 pinyinAbbreviation()。这个函数可以查找出一个中文字符串中每个中文的首字母，然后拼接在一起以新的字符串返回。

接下来是对通讯簿进行排序的时候了，这依靠以下代码来完成：

```
-(void)sortAddressArray{
    ABAddressBookRef addressBook = ABAddressBookCreate();
    NSArray* arr = (NSArray *)ABAddressBookCopyArrayOfAllPeople(addressBook);

    NSArray *sortedArray=[arr sortedArrayUsingComparator:^(id a, id b) {
        NSString* fullName1=(NSString*)ABRecordCopyCompositeName(a);
        NSString* fullName2=(NSString*)ABRecordCopyCompositeName(b);
        NSString* convertString1=pinyinAbbreviation(fullName1);
        NSString* convertString2=pinyinAbbreviation(fullName2);
        return [convertString1 compare:convertString2];
    }];
    addressArray=[sortedArray mutableCopy];
}
```

排序过程在 NSArray 的 sortdArrayUsingComparator 中进行。该方法参数是一个块，包含你想如何进行排序的代码。我们对联系人的全名调用 pinyinAbbreviation 方法进行转换，然后排序。

示例程序位于光盘“source/第 17 章/AddressPinyinSortTest”目录下，程序最终运行效果如图 17-1 所示。注意，请在真实设备上调试。

图 17-1　示例程序 AddressPinyinSortTest

17.2　GPS 和 CoreLocation

在 SDK 中，访问 GPS 功能的唯一方式是使用 Core Location 框架。它包含两个类：CLLocation Manager 和 CLLocation 及一个委托 CLLocation ManagerDelegate。

Core Location 框架提供的地理服务包括位置（经纬度）和海拔。由于后者（海拔）的使用在企业应用中是少见的，下面的示例程序只演示了如何使用 Core location 获得经纬度。完整的项目位于光盘“source/第 17 章/CoreLocationDemo”目录下。

主要的代码在 CoreLocationController 类里。在 init 方法中，我们先获取了一个 CLLocationManager 对象，然后将该对象的 delegate 设置为 self：

```
self.locMgr = [[[CLLocationManager alloc] init] autorelease];
self.locMgr.delegate = self;
```

这就要求 CoreLocationController 类实现 CLLocationManagerDelegate 协议。因此在头文件中，我们声明了对 CLLocationManagerDelegate 进行实现：

```
@interface CoreLocationController : NSObject <CLLocationManagerDelegate>
```

该协议有两个方法需要实现：

- locationManager:didUpdateToLocation: fromLocation: 方法——这个方法当 Location Manager 感知到用户位置发生变化时调用。
- locationManager:didFailWithError: 方法——这个方法当 Location Manager 无法获取用户位置或异常时调用。

我们在 CoreLocationController 中的这两个方法中，又将调用 delegate 对象（CoreLoation Controller 的）的相应方法，代码如下：

```
- (void)locationManager:(CLLocationManager *)manager didUpdateToLocation:(CLLocation *)
        newLocation fromLocation:(CLLocation *)oldLocation {
    if([self.delegate conformsToProtocol:@protocol(CoreLocationControllerDelegate)]) {
        [self.delegate locationUpdate:newLocation];
    }
}
- (void)locationManager:(CLLocationManager *)manager didFailWithError:(NSError
        *)error {
    if([self.delegate conformsToProtocol:@protocol(CoreLocationControllerDelegate)]) {
        [self.delegate locationError:error];
    }
}
```

在上面的代码中，我们使用[NSObject conformsToProtocol:] 方法确认委托是否实现了 CoreLocationControllerDelegate 协议。如果委托对象实现了 CoreLocationControllerDelegate 协议（这个协议是我们在 CoreLocationController 中定义的），则调用对应的协议方法。这样，通过自定义协议，我们封装了原来的 CLLocationManagerDelegate 的接口。

在 CoreLocationController.h 中，我们定义了这个协议，它包含了以下 2 个方法：

```
@protocol CoreLocationControllerDelegate
@required
- (void)locationUpdate:(CLLocation *)location;
- (void)locationError:(NSError *)error;
@end
```

接下来我们要在 View Controller 中使用 CoreLocationController 获取 iPhone 的 GPS 位置信息并显示。这个 View Controller 就是 CoreLocationDemoViewController。

在 CoreLocationDemoViewController.h 中，我们声明了一个 CoreLocationController 成员：

```
CoreLocationController *CLController;
```

在 viewDidLoad 方法中，初始化 CoreLocationController 对象并设置它的委托属性：

```
CLController = [[CoreLocationController alloc] init];
CLController.delegate = self;
```

也许你想到了，我们接下来的工作就是实现委托方法，即 CoreLocationControllerDelegate 协议方法 locationUpdate: 和 locationError: 。在这两个方法中，我们将地理信息或者错误信息显示在 UILabel 中：

```
- (void)locationUpdate:(CLLocation *)location {
    locLabel.text = [location description];
}
```

```
- (void)locationError:(NSError *)error {
    locLabel.text = [error description];
}
```

注意，CoreLocation 不能在模拟器上使用。如果你在模拟器中运行程序，将得到一个错误信息，如图 17-2 所示。

要得到正确的信息，请在 iPhone 上运行程序，iPhone 会询问是否允许使用你的位置信息，请选择允许，运行结果如图 17-3 所示。

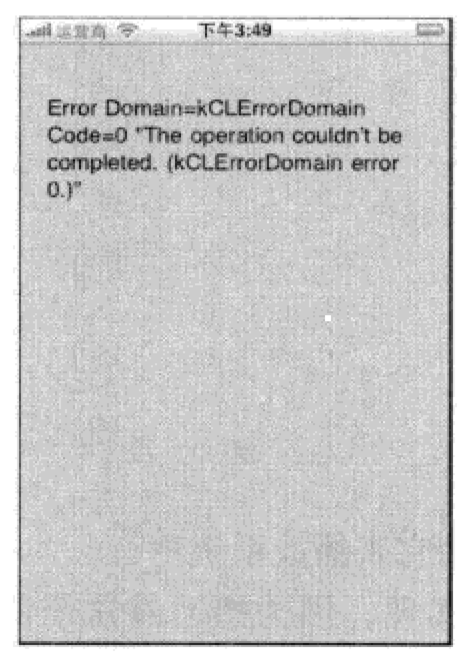


图 17-2　在模拟器上运行 CoreLocationDemo

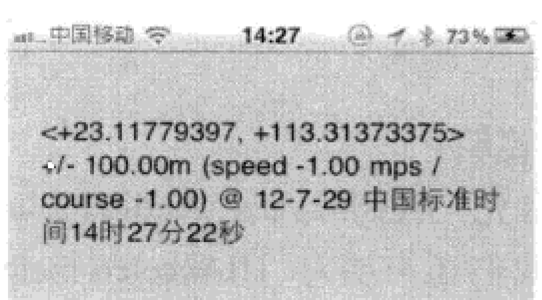


图 17-3　在 iPhone 上运行 CoreLocationDemo

17.3　重力感应

从 iPhone3GS 开始，iPhone 内置了一个三轴加速计。三轴加速计（准确地说是一个三轴陀螺仪和一个加速传感计）则根据角动量守恒原理来计算和检测三维空间中的完整运动。如果在 iPhone 开发过程中使用这些新特性，无疑能为我们的 iPhone 程序加分不少。

iPhone 可通过硬件感知 iPhone 在三维空间中的运动或重力。SDK 中实际上有两个地方使用了重力感应：UIKit 和 UIAccelerate 框架。

其实对于第一种使用重力感应的方式，我们曾经使用过很多次，即旋屏感知。要确定 iPhone 当前方向很简单，访问 UIDevice 的 orientation 属性即可：

```
UIInterfaceOrientation orientation=[[UIDevice currentDevice] orientation];
```

这种方式获得的数据很粗糙。我们可以用它来简单判断 iPhone 屏幕在二维空间中的大致方向，却不能得到任何有用的三维向量和夹角。

现在我们来看第二种使用重力感应的方式：UIAccelerate 框架。UIAccelerate 框架包含两个类：UIAccelerometer 和 UIAcceleration。

UIAccelerometer 类用于访问加速计，并注册委托对象。UIAcceleration 类用于封装加速

计返回的数据，诸如 x、y、z 三个方向的移动和重力。

加速计的常见应用是识别摇动。 摇动手势不同于其他手势，它不通过屏幕触摸来进行识别，只需轻轻晃动 iPhone 就可触发摇动。在某种情形下，摇动手势是一种很酷的特性。比如在某些触摸不可识别情况下，摇动就变得非常有用。

光盘“source/第 17 章/ShakeDemo”中的示例程序是一个使用加速计识别摇动手势的例子。

这是一个简单的 iPhone 应用程序。运行这个程序，如果你用力摇动你的 iPhone，iPhone 会发出一个剧烈的爆炸声音——不管往哪个方向摇动（我们对每个方向的晃动都进行了识别）。

打开 ShakeDemoViewController.h，interface 声明如下。

```
#import <UIKit/UIKit.h>
#import <AudioToolbox/AudioToolbox.h>  //❶
@interface ShakeDemoViewController : UIViewController
<UIAccelerometerDelegate>{ // ❷
    SystemSoundID soundID; //❸
}
@end
```

代码说明：

❶ 首先，我们导入了 AudioToolbox 框架，因为需要用它来播放系统声音。

❷ 我们还声明对 UIAccelerometerDelegate 协议进行实现，因为想通过这个协议使用 UIAccelerometer。

❸ 声明了一个 SystemSoundID 变量，因为 AudioToolbox 框架需要使用一个 SystemSoundID 才能播放声音。

UIAccelerometerDelegate 协议其实只有一个方法：accelerometer: didAccelerate: 。在下面类的实现中将会看到这个方法。

打开类的实现 ShakeDemoViewController.m，实现部分如下所示：

```
#import "ShakeDemoViewController.h"
@implementation ShakeDemoViewController
- (void)viewDidLoad {
    UIAccelerometer *acc=[UIAccelerometer sharedAccelerometer]; //❶
     acc.delegate=self; // ❷
     acc.updateInterval=0.1f; // ❸
     NSString* sound=[[NSBundle mainBundle]pathForResource:@"explodelow.wav"
                ofType:nil]; // ❹
     AudioServicesCreateSystemSoundID((CFURLRef)[NSURL fileURLWithPath:sound],
                &soundID); //❺
}
-(void)accelerometer:(UIAccelerometer *)accelerometer didAccelerate:(UIAcceleration*)
                acceleration{ // ❻
     if(acceleration.x>2.2 || acceleration.y>2.2 || acceleration.z>2.2){ //❼
```

```
            AudioServicesPlaySystemSound(soundID);
        }
    }
- (void)dealloc {
    [super dealloc];
}
@end
```

代码说明：

❶ 获取一个共享的 UIAccelerometer 对象。

❷ 将 UIAccelerometer 对象的 delegate 设置为 self，这样后面实现的 accelerometer:didAccelerate:方法就派上了用场。

❸ updateInteval 属性用于设置加速计的更新频率（浮点值 0.1 表示每秒更新 10 次）。

❹ 我们在项目的 Resources 下面放了一个 explodelow.wav 文件（这个文件在 Mac 系统中搜索得到），可以用 NSBundle 的 pathForResource:ofType: 方法获得这个文件的路径。

❺ 使用 AudioToolBox API 函数 AudioServicesCreateSystemSoundID 创建出一个有效的 SystemSoundID。

❻ 这里是委托方法 accelerometer:didAccelerate:。

❼ 判断加速计在 x、y、z 三个方向上感应到的重力，如果任何一个方向的重力感应超过 2.2g（g 是重力加速度，一个 g 约等于 9.8 米 / 秒的平方），则判定为一个摇动手势，于是用 AudioToolbox API 函数 AudioServicesPlaySystemSound 播放指定声音（一个爆炸的声音）。

17.4 地理编码

地理编码是一种将地理坐标（例如经纬度）和街道地址及其他地理信息关联在一起的服务。实际上地理编码服务包含了两种：地理编码和逆地理编码。前者把地理位置编码为地理坐标（经纬度），后者根据地理坐标查找地理位置（如街道名和门牌号）。

GeoNames(www.geonames.org)就是一个提供地理编码服务的知名网站。你可以在 http://www.geonames.org/export/ws-overview.html 找到它所提供服务的完整列表。这些服务大部分都是免费的。其中，大部分服务都提供 XML 和 JSON 两种数据格式的返回，因此很容易在 iOS 中进行调用。我们使用 CCLocation 的 coordinate 获得地理坐标（经纬度），然后使用 GeoNames 提供的地理编码服务，就可以从 GeoNames 的地理数据库中获得地理编码信息，而这些信息（如海拔、地名、街道）都是 XML 和 JSon 格式的。我们很容易获取 GeoNames 这类网站提供的地理服务，因为不管是请求网络服务，还是进行 XML 和 JSon 解析，都是简单的（可参见第 8 章“XML 和 JSon”）。

17.5 本章小结

本章介绍了通讯簿、GPS 和重力感应在 iPhone 开发中的使用。

SDK 对这些特性提供了支持，包括：通讯簿框架、Core Location 框架和 Accelerate 框架。你不能在模拟器上使用这些框架，它们必须由 iPhone 硬件特性来支持。

虽然 SDK 已经为我们使用这些特性提供了便利，但仅仅是框架本身所提供的功能是不够的。苹果 SDK 所提供的功能是简陋的和基础性的，设计者永远不可能考虑到实际开发过程中的方方面面，我们需要根据实际情况进行各种创新或扩展。比如中文通讯簿的排序，还比如通过 GPS 使用地理编码服务。

实 战 篇

第 18 章　企业 APN

第 19 章　iOS 企业应用实战

第 18 章　企业 APN

APN（Access Point Name，接入点名称）是手机通过运营商接入互联网时必须设置的一个名称。这个名称根据运营商和网络类型的不同而不同，比如中国移动的 cmnet/cmwap，中国联通的 uninet/uniwap/3gnet/3gwap，中国电信的 ctnet/ctwap。在日常生活中，我们通过在手机上设置这个接入点名称，就能使手机浏览网页并访问互联网。

在企业移动应用中，手机客户端当然也可以通过互联网来使用移动应用。但是，通过互联网来访问位于企业内部的数据和服务，相当于把整个企业内部网络暴露在公网中，这种做法并不安全。

那么，我们可以选择“企业 APN”。企业 APN 也叫“专线 APN”，是 APN 中的一种，不同的是，通过企业 APN，我们的手机接入的是企业内网，而不是互联网。这样，通过使用企业 APN，把企业私网与互联网隔离开来，满足了企业在网络使用上的安全要求。

本章将介绍企业 APN 在企业应用中的应用概况，以及使用企业 APN 网络后对 iOS 客户端的一些特殊要求。

18.1　企业 APN 的建设

企业 APN 的建设，需要建设方（企业）和运营商之间的密切合作。大体来讲，需要经过以下几个步骤：

1. 商务谈判阶段

首先企业向营运商提出申请，双方业务部门进行初步的磋商，达成一致意见后以签订正式合同的方式确定合作关系。商务谈判阶段需要确定的内容包括：建设费用、带宽要求、企业专线的资费标准、APN 用户资费标准、工期等。

2. 方案准备阶段

此阶段需要和运营商共同确定具体施工方案，以及双方权责的划分。包括光缆走线布线、设备上架和调试、光缆通过路径、最近接入的光交箱或基站位置、室内光纤布线等。

3. 施工阶段

此阶段按照施工方案进行施工，由双方各自负责相应的工作量，分头进行。

一般而言，企业负责光纤进入企业后的项目，包括：机房装修，室内光纤布线，防火墙 IP 地址和端口过滤，以及配套设备采购及安装（GRE 隧道路由器、Radius 服务器、DHCP 服务器、光电路由器及配套的机箱和电源）。而运营商则负责室外光缆的布放、尾纤的制作及跳线。

4. 调试阶段

施工完成，由移动运营商负责建立业务数据、分配网关和企业的私有 IP 地址。双方通过私有 IP 进行联网，双方互 Ping 连通。

5. 完工阶段

调试成功后，企业（或运营商）需要设置 Radius 服务器和 DHCP 服务器。Radius 服务器负责对企业 APN 用户的主叫号码及账号、密码进行认证，而 DHCP 服务器负责为认证通过的用户分配 IP 地址。因此需要将准备使用企业 APN 的用户的主叫号码向 Radius 服务器进行注册。这样就可以限制了只有某些主叫号码能够访问企业 APN，其他号码则不能访问。也就是说，企业 APN 和用户的 SIM 卡号码是绑定的，进一步保证了企业网络的安全运行。

18.2 iPhone 与 APN

iOS 升级至 4.1 之后，市面上许多版本的 iPhone 已经无法设置 APN 选项。比如使用联通卡的用户，比如美版 iPhone，有的港版 iPhone 和移动卡用户也无法设置 APN。

当然，解决方法也不是没有。网上流传的方法很多，比如安装 APN Editing，或者 TetherMe-个人热点，前提当然是要越狱。对企业 APN 而言，这绝对算不上是什么好消息。想解决这个问题，不一定非要越狱。苹果 App Store 中，也有不少的 APN 切换工具可以选择。

但是对于用户来说，每次打开企业应用前都需要先打开 APN 切换工具，肯定会有一些不好的体验。我对此的建议是，在企业应用中嵌入一个 APN 切换工具的所有功能。

首先需要声明的是，iOS SDK 中绝对没有关于 APN 设置的 API，你可以查找所有的苹果官方文档。不要说 APN 设置，关于系统"设置"程序中的所有功能选项，苹果都没有提供 API。

提示：在 iOS 5.0 时，苹果曾开放了一种新的 URL Scheme 方案，用于支持对设置程序的访问。你可以看这里 http://blog.csdn.net/kmyhy/article/details/7940660。但不幸的是，在 iOS 5.1 之后，苹果又迅速将这个权限收回了。

那么，苹果应用商店中的 APN 切换工具又是如何做的呢？通过苹果的文档企业部署指南（Enterprise Deployment Guide），我们可以知道，要修改 iPhone 的 APN 设置，可以通过配置描述文件（.mobileconfig 后缀文件）进行。企业通过 iTunes、iPhone 配置工具或 OTA（通过 Email 或者 HTTP）部署配置描述文件，iPhone 手机就可以访问配置描述文件中指定的 APN。而运营商们（比如联通），正是通过这种方式，将每一台 iPhone 签约机设置为自己的 APN。

同时，在企业部署指南中提到，通过"iPhone 配置工具"，我们可以编辑、创建自己想要的配置描述文件。配置描述文件的安装则是通过 Safari 浏览器自动进行的。Safari 下载到配置描述文件之后，会自动调用系统的"设置"程序进行安装。

当然，还有一个问题。对于一个 APN 切换工具来说，它很难以 OTA 的方式部署配置描述文件。因为 OTA 部署需要通过网络进行，而用户本来就无法访问网络——APN 都无法设置，

手机当然无法上网。没有网络，APN 切换工具无法使用 OTA 部署配置描述文件，只能另辟蹊径。

经过多次观察市面上的 APN 切换工具，你会发现多数 APN 切换工具都会提供自己的 HTTP 服务。也就是说，APN 切换工具本身就带有一个 HTTP 服务器。不要奇怪，iPhone 是可以作为一台功能正常的服务器使用的。很早以前就有人这样干过。

如果把配置描述文件放到了本地服务器（即 iPhone）上，那么 Safari 可以通过本机的 HTTP 服务来访问配置描述文件，从而避开网络的访问。

本章接下来的内容，就将围绕如何实现一个 APN 切换工具进行。相信经过本章的介绍，你不难在自己的企业应用中集成一个 APN 切换器的功能。

18.3 配置描述文件

iOS 支持两种描述文件，预置描述文件和配置描述文件。预置描述文件以.mobileprovision 为后缀名，配置描述文件以.mobileconfig 为后缀名。

提示：预置描述文件（.mobileprovision）即本书早些时候介绍过的用于在真机上调试和部署应用程序的“设备激活文档”。

安装到 iPhone 上的配置描述文件，可以在 iPhone 的系统设置程序中进行查看。打开系统设置程序，转到“通用→描述文件”页面，在“配置描述文件”文件栏，你可以看到所有已安装的配置描述文件，如图 18-1 所示。

图 18-1　在设置程序中查看配置描述文件

配置描述文件其实是一个 plist 文件，其形式类似于如下代码所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple/DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
          PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>PayloadContent</key>
<array>
<dict>
<key>PayloadDisplayName</key>
<string>LDAP Settings</string>
<key>PayloadType</key>
```

```
<string>com.apple.ldap.account</string>
<key>PayloadVersion</key>
<integer>1</integer>
<key>PayloadUUID</key>
<string>6df7a612-ce0a-4b4b-bce2-7b844e3c9df0</string>
<key>PayloadIdentifier</key>
<string>com.example.iPhone.settings.ldap</string>
<key>LDAPAccountDescription</key>
<string>Company Contacts</string>
<key>LDAPAccountHostName</key>
<string>ldap.example.com</string>
<key>LDAPAccountUseSSL</key>
<false />
</dict>
</plist>
```

.mobileconfig 文件是一个标准的 XML 文件，包含对 iPhone 进行的所有设置（包括 APN 设置）。你可能不明白.mobileconfig 文件语法，不用担心，在苹果公司官方文档“企业部署指南”的附录 B 中，列出了.mobileconfig 文件格式规范。

但是，你没有必要手工编写.mobileconfig 文件。苹果提供了专门的“iPhone 配置工具”用于创建配置描述文件，不需要你记忆繁琐的.mobileconfig 文件语法。

首先，你需要下载“iPhone 配置工具”：

http://support.apple.com/kb/DL1465?viewlocale=zh_CN

并将它安装到你的 Mac 上。

打开“应用程序→实用工具→iPhone 配置实用工具”。在“资料库”一栏中选择“配置描述文件”，然后点击工具栏第一个按钮“新建”，创建一个新的配置描述文件，如图 18-2 所示。

图 18-2　创建一个配置描述文件

接下来，我们尝试配置一个中国移动 cmnet 的 APN 网络设置。其实，对于 APN 配置来说，不管是中国移动的 cmnet 还是中国连通的 3gnet，都是非常简单的：只需要在配置文件中指定 APN 名称一项即可。

在图 18-2 中的“通用”栏，我们按照如下设置：

- “名称”一项为文档指定一个描述性的名称，比如设置为“移动 cmnet”，这个名称会作为配置文件的文件名。
- “标识符”一项为文档指定一个唯一标识，比如“com.company.××××”。com.company 使用反域名规则，××××标识文档的内容。iOS 通过标志符来识别不同的配置描述文件，安装描述文件时，iOS 会对已安装的标志符相同的描述文件进行覆盖。
- “机构”一项可填写你公司的名字。
- “描述”一项随便填写，用于描述该配置文件的内容。

注意，“通用”项是必须填写的，不管你实际要配置什么内容。

接下来，我们切换到“APN”一项，将“访问点名称”指定为中国移动的上网配置：cmnet，其他不用填写。

点击工具栏中的“导出”按钮，弹出“导出配置文件向导”，如图 18-3 所示。

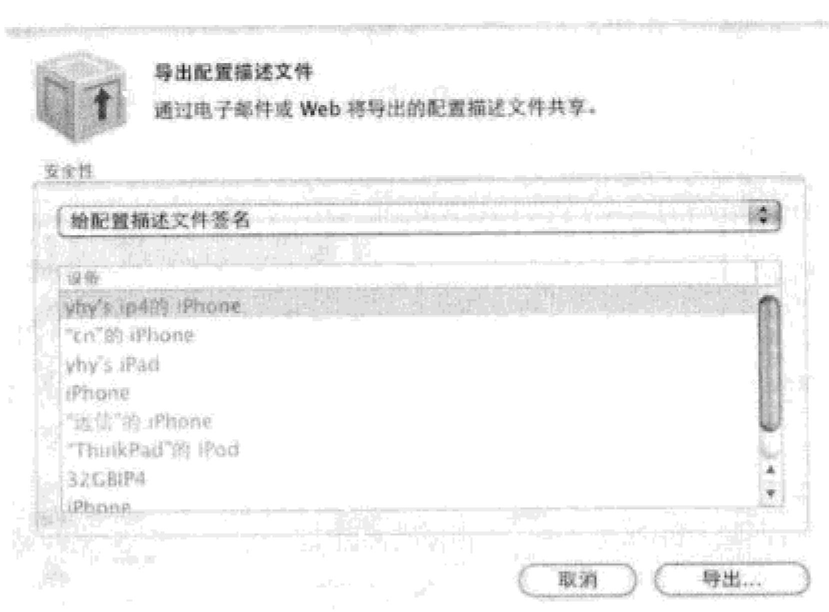


图 18-3 导出配置文件向导

在“安全性”中选择“给配置描述文件签名”，然后点击“导出...”按钮。在接下来的窗口中选择保存地址及文件名称，然后点击“存储”。

18.4 在 iPhone 上实现一个 HTTP 服务器

为什么要在 iPhone app 中自己实现一个 HTTP 服务器？因为 APN 切换需要在 iPhone 上安装一个配置描述文件。但当我们获得了配置描述文件之后，问题就来了。我们把这个配置描述文件放在什么地方才是用户可以获得的？

根据苹果文档“OTA 文档的下发及配置”，可以有 4 种方法部署配置文档：

PDG

- 通过与设备的物理连接。
- 通过 Email。
- 通过 Web 网页。
- 通过 Over-the-air 设置。

首先第一项和第四项被排除。因为通过物理连接方式，几台或少量的设备还好办，如果大范围部署（成百上千台）是不可想象的。如果采用 OTA 部署，则需要使用苹果的 SCEP 协议实现设备的注册和认证，而一个完整的 SCEP 验证过程相当复杂，对于我们这样的简单需求完全没有必要。

使用 Email 或 Web 页的方式是不错的选择。相比较而言，后者（使用 Web 页）更加方便，Safari 通过 URL 来访问配置描述文件并进行安装。问题是当用户的 APN 设置不正确的时候，用户无法上网，又如何访问该 Web 页？所以我们要自己实现一个 HTTP 服务器，并将配置描述文件放到服务器可以访问的地方，比如应用程序自身的资源束中。这样，无论网络是否可用，本地回环地址（127.0.0.1）总是可以访问的。

要实现一个 HTTP 服务器，这涉及 Unix Socket 编程，不管你是用 BSD Sockets 还是 Cocoa CFSocket API。无论如何，你不得不跟这些"丑陋"的 C 语言的 API 打交道——这不是一件容易的事情。幸运的是，有人实现了一个简单的，可扩展的 HTTP 服务器：

```
http://cocoawithlove.com/2009/07/simple-extensible-http-server-in-cocoa.html
```

在这个简单的 HTTP 服务器的实现中，只包含了两个类：HTTPServer 和 HTTPResponseHandler。在后面的 Demo 里，我引入了这两个类（有一些细微的修改）。

提示：HTTPServer 使用了 CFNetwork 框架，别忘了在项目中引入 CFNetwork.framework。

接下来是一个 Demo，我们演示了如何用 HTTPServer 和 HTTPResponseHandler 实现定制的 HTTP 服务器。示例程序位于光盘"source/第 18 章/ApnDemo"目录下。

新建 Single View Application，将 HTTPServer 和 HTTPResponseHandler 拷贝到项目文件夹。由于 HTTPServer 使用了 SynthesizeSingleton.h，别忘了把 SynthesizeSingleton.h 也拷贝到项目文件夹。

注意：在 HTTPServer.h 中，LocalPort 常量定义了本地监听端口为 7531，如果你想监听不同的端口，请修改为其他值。

新建类 MyResponseHandler，继承自 HTTPResponseHandler。我们首先需要重载 NSObject 的类方法 load，在这个方法中，我们必须向父类进行注册：

```
+ (void)load
{
    [HTTPResponseHandler registerHandler:self];
}
```

通过这种方法，HTTPResponseHandler 能准确地知道自己所拥有的子类。这是为了便于 HTTPRequestHandler 能根据不通请求在已注册的 HTTPResponseHandler 子类中查找适当的处理程序。

接下来实现 canHandleRequest:method:url:headerFields:方法。在这个方法中，我们应当表明 MyResponseHandler 类所能处理的请求。这里，返回 YES 即用 MyResponseHandler 来处理所有的请求：

```
+ (BOOL)canHandleRequest:(CFHTTPMessageRef)aRequest
    method:(NSString *)requestMethod
    url:(NSURL *)requestURL
    headerFields:(NSDictionary *)requestHeaderFields
{
    return YES;
}
```

然后实现 startResponse 方法，在这个方法中，我们从请求中获取 URL 路径的最后一部分，然后在资源束中查找相应的文件，如果找到，返回文件内容：

```
- (void)startResponse
{
 NSString* filename=url.lastPathComponent;
    NSString *path =[[NSBundle mainBundle]pathForResource:filename ofType:nil];
    BOOL exists = [[NSFileManager defaultManager] fileExistsAtPath:path];
    if (!exists)
    {
        return;
    }
    NSData *fileData =
        [NSData dataWithContentsOfFile:path];
        CFHTTPMessageRef response =
        CFHTTPMessageCreateResponse(
        kCFAllocatorDefault, 200, NULL, kCFHTTPVersion1_1);
    CFHTTPMessageSetHeaderFieldValue(
        response, (CFStringRef)@"Content-Type",
        (CFStringRef)@"text/plain");
   CFDataRef headerData = CFHTTPMessageCopySerializedMessage(response);
    @try
    {
      [fileHandle writeData:(NSData *)headerData];
      [fileHandle writeData:fileData];
    }
    @catch (NSException *exception)
    {
    }
    @finally
```

```
    {
        CFRelease(headerData);
          [server closeHandler:self];
    }
}
```

现在我们可以在应用程序一启动的时候启动服务器了。编辑 AppDelegate.m，引入 HTTPServer.h 头文件，在 application:didFinishLaunchingWithOptions:方法中加入此句：

```
[[HTTPServer sharedHTTPServer]start];
```

同时在 applicationWillTerminate:方法中记得关闭 HTTPServer：

```
[[HTTPServer sharedHTTPServer]stop];
```

随便将任意文件（文本文件、图片文件）复制到项目文件夹中，例如我们把一个名为"textfile"的文本文件（文件内容是：Just so so.）复制到示例项目中，运行程序，然后打开 Safari 浏览器，在地址栏输入：http://127.0.0.1:7531/textfile，浏览器将输出文本文件 textfile 的内容"Just so so."。如图 18-4 所示。

提示： 如果是图片文件，则浏览器会输出图片的内容。当然，因为我们在 startResponse 方法中，没有正确地设置响应头（如 Content-Type），浏览器中输出的只能是乱码，而不是正确的图形。

18.5 后台任务与无限后台任务

刚才我们是在 Mac 的 Safari 浏览器中进行测试的。ApnDemo 程序在 iOS Safari（不管是模拟器还是 iPhone）上测试无法得到正确的结果。为什么呢？因为 iOS 本质上仍然是一个单任务系统，当你在 iPhone 上，将前台程序由 ApnDemo 切换到 Safari 时，ApnDemo 将退出前台。而当程序一旦转入后台，它的代码就不会再被执行，HTTP 服务器也就无法正常工作。

要让应用程序能够在后台执行，必须使用第 14 章 14.3 节讲到的 iOS 4 "后台任务"。我们需要在 applicationDidEnterBackground:方法中声明一个后台任务：

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    backgroundTask = [application beginBackgroundTaskWithExpirationHandler: ^{
        dispatch_async(dispatch_get_main_queue(), ^{
            if (backgroundTask != UIBackgroundTaskInvalid)
            {
                if([[HTTPServer sharedHTTPServer] state]!=0){
                    [[HTTPServer sharedHTTPServer]stop];
                }
                [application endBackgroundTask:backgroundTask];
                backgroundTask = UIBackgroundTaskInvalid;
```

```
            }
        });
    }];
    // 开始运行后台任务
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        [NSThread sleepForTimeInterval:15];
        NSLog(@"Time remaining: %f",[application backgroundTimeRemaining]);
        dispatch_async(dispatch_get_main_queue(), ^{
            if (backgroundTask != UIBackgroundTaskInvalid)
            {
                if([[HTTPServer sharedHTTPServer] state]!=0){
                    [[HTTPServer sharedHTTPServer]stop];
                }
                [application endBackgroundTask:backgroundTask];
                backgroundTask = UIBackgroundTaskInvalid;
            }
        });
    });
}
```

注意，[NSThread sleepForTimeInterval:15] ;一句向 iOS 申请了 15 秒钟的后台运行时间。iOS 不会管你把这 15 秒用来干什么，甚至是什么也不干。它只是把应用程序“挂起”的期限往后延了 15 秒而已。因此在这 15 秒内虽然我们什么也没做，但 HTTPServer 服务器仍然是工作的。你可以再次运行 ApnDemo 程序，然后按“Home”键把它退到后台，打开 Safari，在地址栏输入 http://127.0.0.1:7531/textfile，就可以查看到 textfile 文件的内容，如图 18-4 所示。当然，这一切必须在 15 秒之内完成。

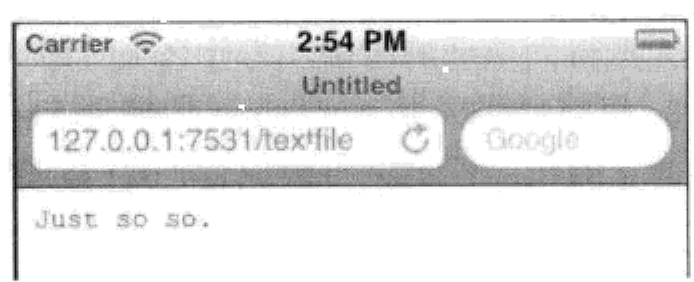


图 18-4　在 iOS Safari 中进行测试

如果你觉得 15 秒钟的后台执行时间不够，你可以增加这个数字。但这个数字不是无限大的。

根据苹果文档中关于后台执行的描述，任何 App 都有 10 分钟左右的后台任务执行时间。10 分钟后，App 会被 iOS 强行挂起。

但是，有 5 类 App 允许有“无限的”后台运行时间：

- Audio
- Location/GPS
- VoIP
- Newsstand

- Exernal Accessory

你可以将任何应用程序声明为上述 5 种类型以获得无限的后台运行时间，但当你提交 App 到 App Store 时，苹果会审查你的 App，一旦发现你“滥用”了后台 API，你的 App 将被拒绝。

当然，对于企业开发而言，不存在“滥用”的问题——企业 App 可以通过 OTA 部署，不经过苹果商店审查。

你可以将 ApnDemo 声明为 VoIP，虽然 ApnDemo 和 VoIP 没有丝毫关系，我们的目的只是为了让 iOS 给我们无限后台执行的权限。声明过程是在 App 的 ApnDemo-Info.plist 文件中加入一个名为 UIBackgroundModes 的 key：

```
<key>UIBackgroundModes</key>
<array>
    <string>voip</string>
</array>
```

然后在 AppDelegate.m 中定义一个 backgroundHandler 方法：

```
- (void)backgroundHandler {
    NSLog(@"### -->backgrounding handler");
    UIApplication* app = [UIApplication sharedApplication];
    backgroundTask = [app beginBackgroundTaskWithExpirationHandler:^{
        [app endBackgroundTask:backgroundTask];
        backgroundTask = UIBackgroundTaskInvalid;
    }];
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    });
}
```

在 backgroundHandler 中，我们启动了一个后台任务，当然在后台任务块中没有代码——我们没有额外的工作要做，我们要求的只是应用程序不被挂起而已。

然后修改 applicationDidEnterBackground:方法中的代码如下：

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    BOOL backgroundAccepted = [[UIApplication sharedApplication] setKeepAlive
        Timeout:600 handler:^{ [self backgroundHandler]; }];
    if (backgroundAccepted)
    {
        NSLog(@"backgrounding accepted");
    }

    [self backgroundHandler];
}
```

这里，我们使用了新的 setKeepAliveTimeout:handler: API 方法。注意，魔法来自于这里。我们在 handler 块以及 iOS 接受后台任务之后都调用了 backgroundHandler 方法！也就是说，当

第一次申请的 600 秒后台执行时间用完之后，我们会再次申请新的后台执行时间。

现在，我们的 ApnDemo 拥有了无限的后台执行时间，你可以在 ApnDemo 程序转入后台后的任何时间请求 HTTPServer 服务。

18.6 实现 APN 切换

现在，我们来实现不同 APN 之间如何切换。比如，我们中国移动的两个 APN，其接入点名称分别为 cmnet 和 cmwap。

首先，参照前面介绍的步骤，我们用“iPhone 配置使用工具”分别制作两个 APN 配置文件：cment.mobileconfig 和 cmwap.mobileconfig。

提示：在创建这两个描述文件时，两者的标识符最好相同，比如都叫做“com.company.apnconfig”。

将这两个.mobileconfig 文件加入（复制）到 ApnDemo 项目中。

打开 MyResponseHandler.m：

```
CFHTTPMessageSetHeaderFieldValue(
response, (CFStringRef)@"Content-Type",
(CFStringRef)@"text/plain");
```

将“text/plain”修改为“application/x-apple-aspen-config”。

打开 ViewController.xib，在界面上拖入两个按钮，如图 18-5 所示。

图 18-5 ViewController.xib 的界面设计

打开 Assistant Editor，将两个按钮连接到 Action 方法 switchAPN：

```
- (IBAction)switchAPN:(id)sender {
    UIButton* button=(UIButton*)sender;
    NSString* filename=[button.titleLabel.text stringByAppendingString:@".
            mobileconfig"];
    NSURL* url=[NSURL URLWithString:[@"http://127.0.0.1:7531/" stringByAppending
            String:filename]];
    [[UIApplication sharedApplication]openURL:url];
}
```

在设备上运行程序，当你点击 cmnet 按钮，Safari 会弹出，紧接着 iOS 会调用系统设置程

序的描述文件安装界面如图 18-6 所示。

图 18-6　设置程序的描述文件安装界面

点击“安装”按钮，将开始安装描述文件。

提示：如果你的 iPhone 上已经安装了其他安装描述文件，则 iOS 弹出“一次只能安装一个 APN 配置”的提示。这时你只有先将已安装的描述文件删除才能继续安装。

安装后的描述文件，可以通过“设置→通用→描述文件”来查看。

如果你想将 APN 网络切换到 cmwap，可以点击 cmwap 按钮。设置程序将自动安装 cmwap.mobileconfig 文件。

注意：虽然你的 iPhone 已安装过一个 cmnet.mobileconfig 描述文件，但此时并不会提示“一次只能安装一个 APN 配置”的提示，而是提示“安装此描述文件将改变 iPhone 上的设置”。这是因为 cmwap 和 cmnet 的描述文件使用的是同一个“标识符”，iOS 将 cmwap 的安装行为视同为对同一个描述文件的“更改”而不是新安装。

18.7　检测网络状况

APN 切换经常会出现意外。比如，用户重复安装描述文件——由于不能很直观地看到 iPhone 当前的网络状况，用户往往会在网络已经可用的情况下，仍然会执行多余的安装步骤。此外，iPhone 的网络状况也会因移动信号的强弱而发生变化。

为此，我们需要为应用程序提供网络检测的功能，让用户能够对 iPhone 的网络通断情况有一个更直观的了解。比如获知 iPhone 的某个网卡（WiFi 网卡或蜂窝数据网卡）是否已经打开。比如读取用户当前的 IP 地址。甚至于需要提供一个 ping 工具，以便用户可以在 iPhone 上对某个服务器进行 ping 测试。

首先，我们来看如何获取 iPhone 的 IP 地址。我们在 ApnDemo 的 ViewController.xib 中增加两个控件：一个按钮和一个 TextView，如图 18-7 所示。

图 18-7 ViewController.xib 的界面设计

用 Assistant Editor 创建两个连接，一个 IBAction 连接由“检测网络”按钮连接至方法 detectNetwork，一个 IBOutlet 由 TextView 连接至属性 textView。

在方法 detectNetwork 中，我们可以使用 getifaddrs()函数（在头文件 ifaddrs.h 中声明）获得网卡地址：

```
- (IBAction)detectNetwork:(id)sender {
    NSMutableString* tvString=[[NSMutableString alloc]init];
    NSString* wIFIStatus=nil, *cellularStatus=nil;
    UInt32 address = 0;
    struct ifaddrs *interfaces;
    if( getifaddrs(&interfaces) == 0 ) {
        struct ifaddrs *interface;
        for( interface=interfaces; interface; interface=interface->ifa_next ) {
            if( (interface->ifa_flags & IFF_UP) && ! (interface->ifa_flags & IFF_
                    LOOPBACK)
              &&(strcmp(interface->ifa_name, "en0")==0 || strcmp(interface->ifa_
                    name, "pdp_ip0")==0)
              ) {
               const struct sockaddr_in *addr = (const struct sockaddr_in*) interface
                    ->ifa_addr;
               if( addr && addr->sin_family==AF_INET ) {
                    address = addr->sin_addr.s_addr;
                    if (strcmp(interface->ifa_name, "en0")==0) {
                        wIFIStatus=[self ipv4Address:address];
                    }
                    if(strcmp(interface->ifa_name, "pdp_ip0")==0){
                        cellularStatus=[self ipv4Address:address];
                    }
               }
            }
```

```
        }
        freeifaddrs(interfaces);
    }
    if (wIFIStatus==nil) {
        wIFIStatus=@"关闭";
    }
    if (cellularStatus==nil) {
        cellularStatus=@"关闭";
    }
    [tvString appendFormat:@"WiFi 网络:%@\n 蜂窝数据网络:%@",wIFIStatus,cellular
            Status];
    textView.text=tvString;
}
```

getifaddrs()函数可以获取设备（iPhone）的网络接口信息，注意这些信息保存为一个 ifaddrs 指针。这是一个链表式的结构，每个 ifaddrs 都有一个 ifa_next 成员，指向下一个 ifaddrs 结构体（如果为 NULL，表明链表结束）。

需要注意的是 ifaddrs 的 ifa_name 成员，它指向了接口名，比如我们关心的两个网卡——WiFi 网卡和蜂窝数据网卡，它们的接口名分别是“en0”和“pdp_ip0”。

ifaddrs 的 sin_addr 存放的是接口的地址信息，包括 IP 地址。这个 IP 地址是一个 UInt32 值，要转换成我们习惯的 IPv4 格式的地址，需要进行一些格式转换，这个过程由 ipv4Address: 方法完成：

```
-(NSString*)ipv4Address:(UInt32)ipv4{
    NSString* IPString;
    if( ipv4 != 0 ) {
        const UInt8* b = (const UInt8*)&ipv4;
        IPString= [NSString stringWithFormat: @"%u.%u.%u.%u",
                (unsigned)b[0],(unsigned)b[1],(unsigned)b[2],(unsigned)b[3]];
    }
    return IPString;
}
```

现在，运行程序。点击“检测网络”按钮，文本框中将列出 iPhone 的 WiFi 网卡和蜂窝数据网卡的 IP 地址。如果有一个网络不可用，则对应网卡将显示“关闭”字样，如图 18-8 所示。

有时候在网络可用的情况下，我们访问服务器仍然会出现异常，比如服务器不可用。因此我们也需要对服务器连接性进行测试。测试的方法有许多种。

其中一种是 HTTP 连接测试，我们可以尝试连接目标服务器，等待 HTTP 响应，查看 HTTP 是否会抛出异常，以此来检查服务器是否工作正常。但是这样需要付出一些额外的代价——这个测试的时间可能很短，也可能很长，因为如果服务器工作正常，HTTP 会很快返回服务器的响应，但网络经常会有延迟，如果服务器不能正常响应，则从一个请求开始到 HTTP 抛出异常的时间可能会很长，如果服务器迟迟不返回响应，客户端必须等待，一直到 HTTP 响应超时，这个时间是你在请求时设置的超时时间。

另一种方法是采用 Ping 测试。因为 Ping 一个服务器很快，而且 Ping 采用的是 ICMP 协议，可能几个毫秒就能得到结果，相比较 HTTP 连接测试而言，这个消耗可以忽略不计。

图 18-8 检测网络

因此，我们决定用 Ping 测试来检测服务器的连接状态。首先将 SimplePing 和 SimplePingHelper 类引入到 ApnDemo 项目。SimplePing 是苹果提供的用于进行 Ping 测试的一个实用工具类，SimplePingeHelper 类是它的助手类。当然，为了更符合我们的目的，我对这两个类进行了一些细微的修改。主要是在 Ping 测试过程中提供了一个 Timeout 超时选项。

SimplePingHelper 类的使用非常简单，在它的接口文件中，只提供了一个类方法：+ ping: target: sel:timeout:，用于对某个地址进行 Ping 测试。注意 Ping 是异步进行的，所以我们需要提供一个 target 和 selecotr 参数，以便在 Ping 返回结果时指定接收的方法。

```
+(void)ping:(NSString*)address target:(id)target sel:(SEL)sel timeout:(float)
  second;
```

其中，方法参数的说明如下：

- address：Ping 测试的目标地址。address 可以是 IP 地址，也可以是域名。
- target 和 sel：指定 Ping 返回时，要返回的目标对象和方法。
- second：指定 Ping 测试的超时时间，超过这个时间仍未返回则认为是 Ping 不通。提供这个选项的原因是，由于服务器性能和网络的原因，Ping 所需要的时间往往是不一定的。有的服务器 Ping 很短的时间（几个毫秒）就能得到结果，有的服务器却需要更长的时间。有时候 second 参数直接影响了 Ping 测试的结果，有时候为了得到准确的结果，我们需要把 second 的值设置得更大。

首先我们在 detectNetwork 方法中加入 Ping 测试的代码：

```
[SimplePingHelper ping:@"www.baidu.com"
                target:self sel:@selector(pingReturn:) timeout:1];
```

然后实现 Ping 测试的回调方法 pingReturn:方法：

```
- (void)pingReturn:(NSNumber*)success {
    NSString* ret=success.boolValue?@"成功":@"失败";
   textView.text=[NSString stringWithFormat:@"%@\n ====ping www.baidu.com:\n\t%@!",
                  textView.text,ret];
}
```

注意，pingReturn:方法应该带有一个 NSNumber 类型的参数，这个参数用数值 1 或 0 表示 Ping 测试的结果：成功或失败。

这样，当我们点击“检测网络”按钮后约 1 秒后，文本框中将输出我们对 www.baidu.com 进行 Ping 测试后的结果如图 18-9 所示。

图 18-9 Ping 测试结果

注意：Ping 不是测试服务器连通性的唯一手段。由于防火墙的存在，某些服务器往往是禁止被 Ping 的。这时，你可以考虑使用 HTTP 连接测试。

18.8 Safari 阻塞

在 iPhone 上进行网络检测，哪怕是网络正常的情况下，也会出现 Ping 失败的现象。这其中的一个原因就是“Safari 阻塞”。

要想明白什么是“Safari 阻塞”，请进行下面这个测试。测试只能在真机上进行，你的手机必须能够打开蜂窝数据网络，同时还需要能够打开 WiFi 网络。如果不具备这个前提条件，你就无法进行下面的测试。

首先，在 iPhone 上运行 ApnDemo，点击“检测网络”。如果你的 WiFi 和蜂窝数据网络都处于打开状态，那么你将在文本框中得到输出文字，如图 18-10 所示。

注意我们的 Ping 测试结果，Ping www.baidu.com 是成功的。

然后切换到 Safari，用 Safari 访问任何一个你可以访问的网址，比如位于办公网络上的某个服务器页面，或者苹果的首页。无论是什么，只要这个页面是 WiFi 网络可以访问的。如图 18-11 所示。

图 18-10 第一次网络测试结果

图 18-11 通过 Safari 使用 WiFi 网络

注意：如果这个地址能够同时通过 WiFi 网络和蜂窝数据网络访问到，则 iOS 优先使用 WiFi 网络，以节省你的蜂窝数据流量费用。因此，在 WiFi 打开的情况下，Safari 肯定是通过 WiFi 网络对 www.apple.com.cn 进行访问的。

打开设置程序，关闭 WiFi 网络。然后切换至 ApnDemo 程序，点击"检测网络"。出人意料的结果发生了，此时我们的 Ping 测试失败了！如图 18-12 所示。

```
WIFI 网络:关闭
蜂窝数据网络:9.179.102.98
====ping www.baidu.com:
    失败!
```

图 18-12 再次进行 Ping 测试，结果失败了

我不知道苹果怎么解释这件事情，我把这个诡异的现象称为"Safari 阻塞"，因为我还没有发现其他人报告过这个 Bug。

如果你还不明白什么是"Safari 阻塞"，那么你可以再次按照以下步骤进行测试。

- ❑ 打开 WiFi 网络，用 Safari 访问 WiFi 网络。
- ❑ 关闭/不关闭 WiFi 网络。
- ❑ 再次 Ping 测试。

注意，第 2 步"关闭/不关闭 WiFi 网络"直接影响了 Ping 测试的结果。如果在第 2 步中"关闭"WiFi 网络进行 Ping 测试，结果是失败；而"不关闭"WiFi 进行 Ping 测试，结果是成功。

此外，如果在 Ping 测试之前，将 Safari 进程退出，测试总是成功的。

我个人的解释，是 Safari 的存在会干扰 iOS 对于"WiFi 优先"策略的执行。我不知道如何关闭 Safari，iOS 不会允许你直接关闭一个后台挂起的进程。但是在测试中我发现有一点，当第 1 次 Ping 失败后，后续几次 Ping 又是成功的，可能是第 2 次，也可能是第 3 次。我们可以从这一点入手，来解决这个问题。比如，当用户点击"监测网络"时，我们可以进行多次 Ping 测试，只要有一次 Ping 成功，我们就返回。

由于为了防止在 Ping 测试过程中阻塞主线程，我们采用 Dispatch 方式进行 Ping 测试。

还是用 detectNetwork 方法。我们将 detecNetwork 方法中的代码修改为：

```
pingSuccess=false;
__block int timesOfTry=0; // ❶
source = dispatch_source_create(DISPATCH_SOURCE_TYPE_DATA_ADD, 0, 0, dispatch_
     get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)); // ❷
dispatch_source_set_event_handler(source, ^{        //❸
    if (pingSuccess==YES || timesOfTry==5) {    //❹
        dispatch_source_cancel(source);
        dispatch_release(source);
        dispatch_source_cancel(timer);
        dispatch_release(timer);
```

```
        printf("Dispatch stopped.\n");

    }else{                              // ❺
        timesOfTry++;
        NSLog(@"%d times for try.\n",timesOfTry);
        [self performSelectorOnMainThread:@selector(ping:) withObject:@"www.
              baidu.com"
                         waitUntilDone:NO];
    }
});
dispatch_resume(source);        // ❻
timer = dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER, 0, 0, dispatch_get_
        global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0));  // ❼
dispatch_source_set_timer(timer, DISPATCH_TIME_NOW, 1ull * NSEC_PER_SEC, 0);
                              // ❽
dispatch_source_set_event_handler(timer, ^{
    dispatch_source_merge_data(source, 1);
});            // ❾
dispatch_resume(timer); // ❿
```

提示：在方法中使用了 3 个静态变量，请在适当的地方加入变量的声明语句：

```
static dispatch_source_t source,timer;
static bool pingSuccess;
```

代码说明：

❶ 声明了一个 int 变量，用于统计 Ping 测试的次数。由于在块中会修改这个变量，我们使用了__block 关键字修饰（请参考第 3 章 3.5 节“块编程”部分）。

❷ 用 dispatch_source_create 创建一个 dispatch 源（请参考第 14 章 14.2 节“并行编程”部分）。

❸ 设置 dispatch 源的 handler。

❹ 在 handler 块中，我们判断 Ping 测试是否成功，以及 Ping 的次数达到了最大限制（5 次）。如果是，我们将取消 dispatch 源，于是 Ping 测试全部结束。

❺ 如果不是，则继续调用 ping:方法进行 Ping 测试。

❻ 在设置好 dispatch 源 source 的 handler 之后，我们用 dispatch_resume 提交 dispatch 源。这不会导致 source 的 handler 块被执行。因为 source 的类型是 DISPATCH_SOURCE_TYPE_ DATA_ADD 类型，这种类型的 dispatch 源的 handler 块只能通过 dispatch_source_merge_data 函数来调用。

❼ 我们再次用 dispatch_source_create 创建第 2 个 dispatch 源。与第 1 个源不同，这次创建了一个定时器源 timer。

❽ 定时器源比较特殊的是必须设置它的定时器，我们用 dispatch_source_set_timer 将 timer 的定时器运行周期设置为 1 秒钟。

❾ 接下来设置定时器源的 handler。handler 中仅有代码“dispatch_source_merge_data(source,

1);”一句，它会根据 source 的类型提交 handler 块。由于 source 是 DISPATCH_SOURCE_TYPE_DATA_ADD 类型，所以这句代码将以 ADD 的方式合并提交 source 的 handler 块。

⑩ 提交 dispatch 源 timer。这会导致定时器立即运行。于是每间隔 1 秒，source 源的 handler 块被调用一次（进行一次 Ping 测试），直到 Ping 返回成功或达到 5 次。

ping:方法用于进行 Ping 测试，方法定义如下：

```
-(void)ping:(NSString*)address{
    [SimplePingHelper ping:address
                    target:self sel:@selector(pingReturn:) timeout:1];
}
```

当 Ping 有结果（成功或失败）返回时，pingReturn:方法被调用。pingReturn 方法简单判断 Ping 成功与否，然后调用 tvLog:方法进行输出：

```
- (void)pingReturn:(NSNumber*)success {
    pingSuccess=success.boolValue;
    NSString* ret=pingSuccess?@"成功":@"失败";
    [self tvLog:@"====ping www.baidu.com:%@!",ret];
}
```

tvLog:方法仅仅是输出文本到 UITextView 控件，但它的参数有一点点特殊，它有一个可变参数，可变参数是个数不定的参数列表（请参考第 3 章 3.6 节“可变参数”部分）。

```
-(void)tvLog:(NSString*)fmt,...{
    va_list args;
    va_start(args,fmt);
    NSString* string=[[[NSString alloc] initWithFormat:fmt arguments:args] autorelease];
    va_end(args);

    if (textView.text && textView.text.length>0) {
        string=[NSString stringWithFormat:@"%@\n%@",textView.text,string];
    }
    objc_msgSend(textView, @selector(setText:),string);
    NSRange range=NSMakeRange(textView.text.length,0);
    objc_msgSend(textView, @selector(scrollRangeToVisible:),range);
}
```

tvLog 方法很像 NSLog 方法。它也具有两个参数，第 1 个是 NSString，是一个带百分号的格式字符串；第 2 个是一个省略号...，代表一个可变参数。可变参数代表了一个个数未知的参数列表，此外我们也无法获知它的参数类型。因此使用 tvLog 方法时，我们完全可以用使用 NSLog 一样的方法，例如：

```
[self tvLog:@"%d%@",123,@" copies"];
```

从"va_list args:"一行到"va_end(args);"一行，我们将可变参数按照格式字符串 fmt 指定的格式组装出一个完整的文本。注意，NSString 有一个初始化方法 initWithFormat:arguments: 方法，它的第 2 个参数允许指定一个 va_list 类型（可变参数列表），这样它会自动计算第 1 个格式化字符串参数中%号的个数来作为可变参数的个数，同时配合第 2 个参数（可变参数）来初始化一个指定格式的字符串。类似的还有 NSLog（NSString *format，va_list args）函数，也使用了同样的技术。

然后将文本追加到 UITextView 文本的最后。然后滚动 UITextView 的文本区域。注意，我们使用 ojbc_msgSend 函数来调用 UITextView 的相应方法，是因为无论修改 UITextView 的 text 属性还是滚动文本视图，都需要刷新 UI。刷新 UI 应该在主线程中进行，如果你不使用 objc_msgSend 方法，那么就得用 performSelectorOnMainThread 方法。但是 performSelectorOnMainThread 方法无法直接传递 NSRange 这样的简单参数（必须是 NSObject 类型的参数）。

运行 ApnDemo 程序，点击"检测网络"，如果 Ping 测试不成功，则会进行多次 Ping 测试，直到 Ping 成功，如图 18-13 所示。

```
WIFI 网络:10.81.117.193
蜂窝数据网络:9.227.156.238
====ping www.baidu.com:失败!
====ping www.baidu.com:失败!
====ping www.baidu.com:成功!
```

图 18-13　程序最终运行结果

18.9　本章小结

为了满足企业对网络安全和信息安全的实际要求，使用企业 APN（APN 专线）是一种很常见的做法。对于企业移动应用来说，业务系统位于企业内网，使用 APN 将它和互联网进行隔离是非常必要的，很难有其他可以替代的手段。

但是，由于 iOS 的种种限制，对于某些不能通过设置程序修改 APN 配置的用户来说，只能通过安装配置描述文件的方式来进行 APN 的切换。本章介绍了一个以编程方式实现的修改 iPhone APN 设置的解决方案，即一个简单的 APN 切换工具（同时它也提供了简单的网络状态检测）。你可以把它集成在自己的应用程序中。虽然它很简单，但涉及了广泛的内容，诸如后台任务、配置描述文件、BSD Socket 编程、网络检测、Safari 阻塞和并行编程 GCD（Grand Central Disptach）。

第 19 章　iOS 企业应用实战

本章中，我们讨论一个综合网络应用案例——Any Mail。这是一个 iPhone 上的简单邮件客户端，它可以让你通过 iPhone 接收服务器上存放的邮件，也可以让你用 iPhone 向其他人发送邮件。这只是一个虚拟的项目，其服务器端代码（是用 Java 实现的）被大大简化了，这是有意的，因为我们不应该冲淡本书主题——iOS 应用开发。

19.1　应用场景与功能概述

Any Mail 完全依靠于 iPhone 的网络通信能力。如果你想在 iPhone 上测试这个程序，那么需要打开 iPhone 的蜂窝数据网络或 WiFi。因为 Any Mail 随时可能请求服务器的数据。

Any Mail 需要实现如下功能：邮箱登录、查看邮件、查看附件、发送邮件、写邮件、从服务器中获取联系人列表。当然服务器端也必须实现相应的接口。

注意：在这个案例中，服务器端的代码被尽可能地简化了，我们甚至没有使用到数据库，而实际开发中这几乎是不可能的。在实际项目中，服务端的代码要复杂得多，但本书只能把重心放到 iPhone 端。

19.2　应用程序架构

应用程序架构为 C/S 架构。客户端和服务器通过 HTTP 协议进行通信。服务器端采用 Tomcat 和 Java 实现，客户端为 iOS。

服务器端代码位于光盘“source/第 19 章/test”目录下，客户端代码位于光盘“source/第 19 章/AnyMail”目录下。

19.3　服务器端

服务器端代码为 Java 实现。下面我们简单介绍服务器端代码，熟悉 Java 开发的读者可以跳过这部分内容不读，直接使用作者已经简单实现的服务器端代码就可以了。服务器端的代码是一个完整的 Eclipse J2EE 项目，你可以直接在 Eclipse 中打开它，或者直接在 Tocmat 服务器上部署它。

19.3.1　环境搭建

在本章中，服务器环境是 Tomcat。因此你需要在机器上安装 JDK 和 Tomcat。同时还需要

安装一个 Eclipse 作为开发环境。Mac OS X 默认已经安装了 JDK 1.6。因此你只需要安装一个 Tomcat 和一个 Eclipse IDE 就行，以下是下载地址。

```
Tomcat: http://tomcat.apache.org/download-70.cgi
Eclipse: http://www.eclipse.org/downloads/
```

19.3.2 实现登录接口

为求简便，登录接口用一个 JSP 页面实现简单验证。只要用户名和密码不为空，我们就验证为登录成功。打开 Eclipse，新建 Web 项目。新建 JSP 页：Login.jsp

```
<%
String user=request.getParameter("user");
String pass=request.getParameter("pass");
out.println("<?xml version=\"1.0\"?>");
if(user!=null || pass!=null){
    out.println("<login><status>true</status><user_id>007</user_id></login>");
}else{
    out.println("<login><status>false</status></login>");
}
%>
```

运行 Web 程序，在浏览器中输入：`http://localhost:8080/AnyMail/login.jsp?user=&pass=` 回车，服务器将返回 XML：

```
<?xml version= "1.0"?> <login><status>true</status><user_id>007</user_id></login>
```

提示：请求参数 “?user=&pass=” 并不会使 user 和 pass 为空，而是表示 user 和 pass 为空字符串 “”。通过 request.getParameter 获取到这两个参数时，会得到空字符串，但字符串本身不会为空。

19.3.3 实现企业通讯簿接口

企业通讯簿接口用 directory.jsp 实现如下：

```
<%
String user=request.getParameter("user");
String pass=request.getParameter("pass");
out.println("<?xml version=\"1.0\" encoding=\"utf-8\" ?>");
if(user!=null || pass!=null){
    out.println("<list><dept name=\"行政部\" id=\"01\"><linkman id=\"001\" name=\"
        郭书全\"></linkman></dept>");
    out.println("<dept name=\"人力部\" id=\"02\"><linkman id=\"002\" name=\"王有福
        \"></linkman></dept></list>");
}else{
    out.println("<login><status>false</status></login>");
}
%>
```

作为一个 Demo，企业通讯簿接口直接返回了静态 HTML 数据。实际上，你应该从企业数据库中获得企业通讯簿数据。

19.3.4 实现收件箱接口

企业通讯簿接口用 inbox.jsp 实现，代码如下：

```
<%
String user=request.getParameter("user");
String pass=request.getParameter("pass");
out.println("<?xml version=\"1.0\" encoding=\"utf-8\" ?>");
if(user!=null || pass!=null){
    out.println("<list><mail><title>测试邮件 1</title>");
    out.println("<id>01</id><content>test</content>");
    out.println("<sender>王有福</sender><attachid>0001</attachid>");
    out.println("<time>2011-11-11</time></mail></list>");
}else{
    out.println("<login><status>false</status></login>");
}
%>
```

作为一个 Demo，收件箱接口直接返回了静态 HTML 数据。实际上，你应该从邮件服务器或数据库中获得收件箱邮件数据。

19.3.5 实现附件上传接口

附件上传接口使用第 7 章 7.4.3 节“文件上传”中介绍的 UploadServlet 类实现。

注意： 在文件系统中要确定文件上传目录的存在。即在 root 下新建文件夹 data，并在 data 目录下建立 temp 目录（Apache fileupload 组件的临时文件目录）。

19.3.6 实现附件下载接口

附件下载由 download.jsp 页面实现，代码如下：

```
<%
String user=request.getParameter("user");
String pass=request.getParameter("pass");
String attachid=request.getParameter("attachid");
out.println("<?xml version=\"1.0\"?>");
if(user!=null || pass!=null){
    out.print ("<attach><attachid>0001</attachid><file_size>219169</file_size>");
    out.print ("<file_name>0064.jpg</file_name>");
    out.print("<url>http://localhost:8080/AnyMail/0064.jpg</url>");
```

```
        out.print("</attach>");
    }else{
        out.print("<login><status>false</status></login>");
    }
%>
```

注意：在 Web 目录中放一个用于示范的下载文件 0064.jpg。

作为一个 Demo，附件下载接口直接返回一个图片文件的 URL 地址。实际上，你应该从邮件服务器或数据库中获得这个 URL，或者直接返回文件流。

19.4 iPhone 客户端

接下来介绍 iPhone 客户端的实现，这是本章重点关注的内容。iPhone 客户端代码是一个 Xcode 项目，你可以直接在 Xcode 中打开它。

19.4.1 实现登录

本书的第 7 章“网络”和第 8 章“XML 和 JSON”与本章有密切联系。本例需要使用第 7 章中介绍的 ASIHttpRequest 框架。

新建 Empty Application，并在项目中引入 ASIHttpRequest 框架（需要导入相关依赖库，请参考第 7 章相关内容）。

本例需要使用第 8 章中介绍的 GDataXML 框架解析 XML，请将 GDataxMLNode.h 和 GDataXMLNode.m 文件添加到项目目录（需要 libxml 库）。

此外，我们使用了第 7 章中自定义网络模块 NetworkModule 和 PostRequest。打开光盘“source/第 8 章/testGData”目录，将“NetworkModule”目录下的所有文件加入项目中，如图 19-1 所示。

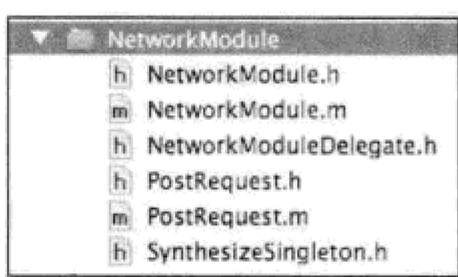


图 19-1 NetworkModule 目录中包含的文件

注意：根据本章实战项目的实际情况，我们对 PostRequest 类和 NetworkModule 类做了一些改动，主要是 PostRequest 类的 postURLWithDelegate 方法和 NetworkModule 类的 postURL 方法。请自行参考源代码。

首先我们来创建解析器。登录接口中只有一个 XML 文件，使用第 8 章中介绍的 GDataXML

和 XMLNode 类，我们很容易就创建这个 XML 文件的解析器 loginXML，你可以在 xmlObjects 文件夹下看到它。它有一个实例化方法：initWithXMLDocument:以及两个 String 属性 status 和 user_id。

```
@property(retain,nonatomic)NSString* status;
@property (nonatomic,retain)NSString* user_id;
-(id)initWithXMLDocument:(GDataXMLDocument *)doc;
```

由于使用了 SynthesizeSingleton 的单例模式（在第 7 章专门介绍过），我们也声明了一个单例方法：

```
+ (loginXML *)sharedloginXML;
```

提示：这个单例方法不需要实现。声明就行。

在实现中，只有一个 initWithXMLDocument:方法的实现，如下所示：

```
-(id)initWithXMLDocument:(GDataXMLDocument *)doc{
    self=[super init];
    if (self) {
        if (doc && doc.rootElement!=nil) {
            XMLNode* xmlNode=[[[XMLNode alloc]init]autorelease];
                    xmlNode.element=doc.rootElement;
            for (GDataXMLNode* each in doc.rootElement.children) {
               NSString* label=[GDataXMLNode localNameForName:each.name];
                       objc_property_t property=class_getProperty([self class],
                           [label cStringUsingEncoding:NSUTF8StringEncoding]);
                if (property) {
                       NSString* text=each.stringValue;
                       [self setValue:text forKey:label];
                }
            }
        }
    }
    return self;
}
```

由于使用了 Objective-C 的运行时库，上面的代码显得很简洁，而且复用性很高，此后你会发现，我们会在后续实现中大量重复使用这段代码。它会自动把 XML 元素的标签名和类声明中的属性名进行匹配，以后对于不同的 XML 文件解析类，我们只要根据 XML 文件中的标签名来声明属性就可以了。

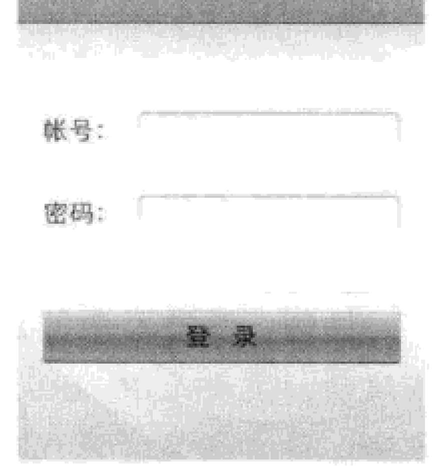


图 19-2 登录界面设计

接下来我们对登录界面 loginVC.xib 做简单设计（如图 19-2 所示）。

进行必要的连接。即将两个 Text Field 连接到两个 IBOutlet，将 Button 的 touchUpInside

事件连接到动作-(IBAction)loginAction。

在下面的代码中，我们使用了网络模块 NetworkModule 类向服务器进行登录：

```
- (IBAction)loginAction:(id)sender {
    NSString* url=@"http://localhost:8080/AnyMail/login.jsp?user=%@&pass=%@";
    url=[NSString stringWithFormat:url,tfName.text,tfPass.text];
     [[NetworkModule sharedNetworkModule]postURL:urltag:kBusinessTagUserLogin
         owner:self];


}
```

总共 3 行代码，复杂逻辑都被封装到了 NetworkModule 类里。接下来实现 NetworkModule Delegate 协议：

```
-(void)endPost:(GDataXMLDocument *)result business:(kBusinessTag)tag{
    if (tag==kBusinessTagUserLogin) {
       loginXML* login=[loginXML sharedloginXML];
       [login initWithXMLDocument:result];
       if([@"true" isEqualToString:login.status]){
           [login setPass:tfPass.text];
           [login setUsername:tfName.text];
           InboxVC* vc=[[[InboxVC alloc]init]autorelease];
           [self.navigationController pushViewController:vc animated:YES];
       }else{
           showMessage(@"",@"登录失败！");
       }
    }
}
```

登录成功，我们跳转到收件箱页面。

19.4.2 查看收件箱

收件箱页面由 InboxVC 实现。它显示了收件箱的邮件列表，由一个 Table View 构成。Table View 上的单元格是我们自定义的 InboxCell 类（继承自 UITableViewCell）。我们先讲 InboxCell 类的实现。

选择 New File，新建类 InboxCell ，继承自 UITableViewCell。

选择 New File，“新建 iOS→User Interface→View”，命名为 InboxCell。这将生成一个 InboxCell.xib 文件。编辑 InboxCell.xib ，将 View 对象的 Idenifier 修改为 InboxCell，Size 修改为 freedom。

提示：Xcode 会提示一个警告，大意为只有 Xcode 4.2 以上版本支持 freedom，默认的 xib 文件是 Xcode4.1 的。选择 InboxCell.xib 文件，打开它的 File Inspector 面板，找到 Development 一行，将其修改为 Xcode4.2，警告消除。

在 InboxCell.xib 中设计单元格的布局，放入几个 UILabel。最终结果如图 19-3 所示。创建必要的连接。

编辑 InboxCell.h 声明两个方法：

图 19-3 InboxCell 的 UI 设计

```
+(InboxCell*)getCell;
-(void)setCell:(MailXML*)mailInfo;
```

setCell: 方法中使用到 MailXML 类，这是我们编写的 XML 解析类，将在后面介绍。

编辑 InboxCell.m，实现这两个方法：

```
+(InboxCell *) getCell
{
    NSArray *array = [[NSBundle mainBundle] loadNibNamed:@"InboxCell" owner:self
            options:nil];
    InboxCell* cell=[array objectAtIndex:0];
    return cell;
}
-(void)setCell:(MailXML*)mailInfo{
    self.lbTitle.text=mailInfo.title;
    self.lbSender.text=mailInfo.sender;
    self.lbTime.text=mailInfo.time;
}
```

接下来编辑 InboxVC.h，将接口的声明修改为：

```
#import "NetworkModuleDelegate.h"
#import "InboxXML.h"
@interface InboxVC : UIViewController
<UITableViewDelegate,UITableViewDataSource,NetworkModuleDelegate>{
    InboxXML *inboxInfo;
}
```

我们声明了一个 InboxXML 对象。这是我们的 XML 解析类，同时也为 table view 提供用于显示的数据。将在后面介绍。

编辑 InboxVC.m。在 viewDidLoad 方法中加入代码，用我们的自定义网络模块 NetworkModule 来请求 XML 数据：

```
NSString* url=@"http://localhost:8080/AnyMail/inbox.jsp?user=%@&pass=%@";
   url=[NSString stringWithFormat:url,[loginXML sharedloginXML].username, [loginXML
```

```
        sharedloginXML].pass];
    [[NetworkModule sharedNetworkModule]postURL:url tag:kBusinessTagInbox owner:
        self];
```

实现 NetworkModuleDelegate 委托。在 endPost 委托方法中，我们接收 XML 文件并用 InboxXML 类进行解析，封装为更容易使用的对象，然后刷新表格显示：

```
-(void)beginPost:(kBusinessTag)tag{
}
-(void)endPost:(GDataXMLDocument *)result business:(kBusinessTag)tag{
    if (tag==kBusinessTagInbox) {
       inboxInfo=[[InboxXML alloc]initWithXMLDocument:result];
       [table reloadData];
    }
}
-(void)errorPost:(NSError *)err business:(kBusinessTag)tag{
}
```

Table View 的 reloadData 方法调用导致数据源方法被触发，从而刷新表格单元格。Table View 的数据源方法实现如下：

```
-(NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)
            section{
   return inboxInfo.list.count;
}
-(float)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath
            *)indexPath{
   return 70;
}
-(UITableViewCell*)tableView:(UITableView *)tableView cellForRowAtIndexPath:
            (NSIndexPath*)indexPath{
   static NSString* s=@"erbd3909";
   InboxCell* cell=(InboxCell*)[tableView dequeueReusableCellWithIdentifier:s];
   if (cell==nil) {
      cell=[InboxCell getCell];
   }
   MailXML* mailInfo=(MailXML*)[inboxInfo.list objectAtIndex:indexPath.row];
   [cell setCell:mailInfo];
   cell.accessoryType=UITableViewCellAccessoryDisclosureIndicator;
   return cell;
}
```

在数据源方法 cellForRowAtIndexPath 中，我们使用定制的 UITableViewCell 子类 InboxCell 作为表单元格。调用 InboxCell 的 getCell 方法将获得一个新的 InboxCell 实例。然后从 inboxInfo 对象（我们姑且把它看成是一个 XML 文件封装成的对象）的 list 属性中检索出一个 MailXML 对象，用于渲染表单元格的显示。

XML 解析类 lnboxXML 和 MailXML 是“集合”关系。一个 lnboxXML 中包含了一个由 MailXML 对象构成的数组（即 list 属性）。在 lnboxXML 的初始化方法 initXMLDocument:中，我们遍历了 XML 文档中的 mail 元素，然后把每个 mail 元素封装为 MailXML 对象并放入 list 数组：

```
-(id)initWithXMLDocument:(GDataXMLDocument *)doc{
    self=[super init];
    NSArray* a=[doc.rootElement nodesForXPath:@"//list/mail" error:nil];
    list=[[NSMutableArray alloc]init];
    for (GDataXMLElement* each in a ) {
        MailXML* mail=[[[MailXML alloc]initWithXMLElement:each]autorelease];
        [list addObject:mail];
    }
    return self;
}
```

MailXML 类的初始化方法 initWithXMLElement 检索 mail 元素的每个子元素，将它们一一放入 MailXML 对象的属性值中。MailXML 初始化方法中的这段代码，其实在前面我们就见过。它跟 loginXML 的 initWithXMLDocument 方法并没有太多区别。

提示：唯一需要注意的是，XML 文件中的 mail 元素使用了“id”作为元素名。“id”是 Objective-C 的保留字（表示 NSObject 类型），我们不能使用“id”作为成员属性。因此我们进行了一小点变通，把它转换为大写以用做属性名。如果在你的 XML 文件中有任何使用 Objective-C 保留字的情况，你应该警惕并做出类似的处理。

运行程序，如果一切正常，收件箱列表将显示如图 19-4 所示的界面。

图 19-4　收件箱界面

我们还要实现 Table View 的委托方法。这样，当你点击表单元格的时候，会导航到邮件浏览界面（即 InboxDetailVC 类，用于显示邮件的内容和邮件附件。邮件浏览将在下一主题介绍）。

```
-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
        *)indexPath{
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
    MailXML* mailInfo=(MailXML*)[inboxInfo.list objectAtIndex:indexPath.row];
```

```
    InboxDetailVC* vc=[[InboxDetailVC alloc]init];
    [self.navigationController pushViewController:vc animated:YES];
    [vc setMailInfo:mailInfo];
}
```

19.4.3　邮件浏览

阅读邮件页面由 InboxDetailVC 视图控制器实现。在 InboxDetailVC.xib 中，我们加入了一个 UIWebView 用于显示邮件内容，一个 UILabel 用于显示附件文件名称，一个按钮用于下载附件，如图 19-5 所示。

建立连接的过程不表，请参考.xib 文件的连接面板。

打开 InboxDetailVC.h，声明一个属性，用于保存收件箱页面传过来的 MailXML 对象：

```
@property (retain,nonatomic)MailXML* _mailInfo;
```

声明方法 setMailXML:。在该方法中利用 NetworkModule 网络模块，调用服务器的“获取附件”接口（即 download.jsp），以获得附件 url。方法实现如下：

```
-(void)setMailInfo:(MailXML*)mailInfo{
    NSString* url=@"http://localhost:8080/AnyMail/download.jsp?user=%@&pass=
            %@&attachid=%@";
    url=[NSString stringWithFormat:url,[loginXML sharedloginXML].username,
        [loginXML sharedloginXML].pass,mailInfo.attachid];
    [[NetworkModule sharedNetworkModule]postURL:url tag:kBusinessTagAttachInfo owner:
        self];
    self._mailInfo=mailInfo;
}
```

然后实现 NetworkModuleDelegate 委托方法如下：

```
-(void)beginPost:(kBusinessTag)tag{
    [btDown setEnabled:NO];
}
-(void)endPost:(GDataXMLDocument *)result business:(kBusinessTag)tag{
    [btDown setEnabled:YES];
     if (tag==kBusinessTagAttachInfo) {
        AttachInfoXML* info=[[AttachInfoXML alloc]initWithXMLDocument:result];
        fileUrl=info.url;
        lbAttachID.text=info.file_name;
        [webView loadHTMLString:[HTMLMaker buildDetailInfoHtml:_mailInfo]
                     baseURL:nil];
     }
}
-(void)errorPost:(NSError *)err business:(kBusinessTag)tag{
    [btDown setEnabled:YES];
}
```

在 endPost 委托方法中，我们读取 XML 文件，并用 AttachInfoXML 进行解析，获得附件文件名和 URL。然后利用 UIWebView 的 loadHTMLString: 方法加载一个 HTML 内容。这里，我们把 _mailInfo 的内容以 HTML 表格的方式显示到 Web View。这是一种很讨巧的办法，对于仅仅用来浏览的信息，用一个 Web View 控件就足够了，不必再浪费时间设计一个纯粹由 UILabel 构成的 UI。

HTML 表格是由 HTMLMaker 这个类动态构建的，它的内容来自于两方面：一是 detail_template.html 文件，这文件的内容主要包括一个没有单元格的 table 和一些 CSS 样式；二是 MailXML 对象，我们把 MailXML 对象的属性添加为 table 的单元格。HTMLMaker 的 buildDetailInfoHtml 方法返回最终生成的 HTML 字符串。HTMLMaker 的具体实现细节请参考源文件。HTML 表格在 UIWebView 中加载后效果如图 19-6 所示。

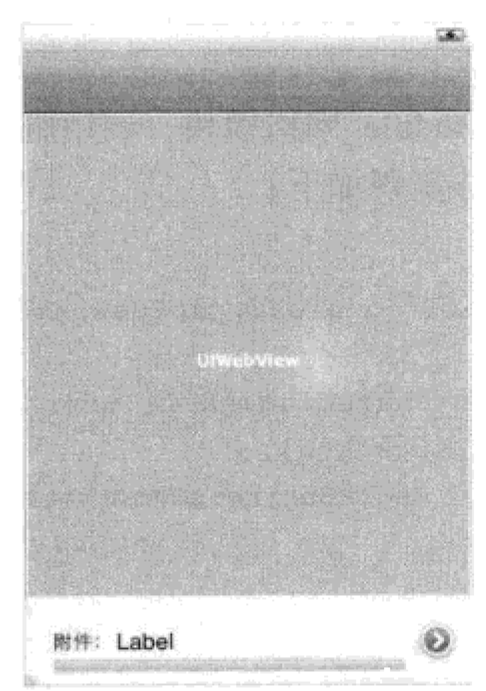


图 19-5 InboxDetailVC 的界面设计

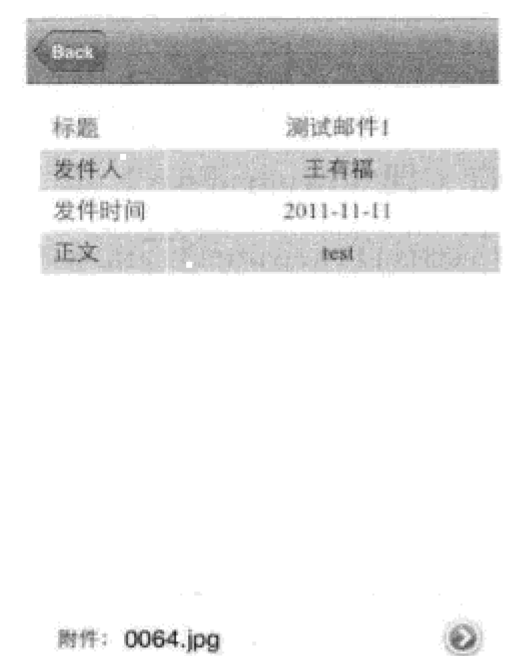


图 19-6 用 Web View 显示邮件内容

AttachInfoXML 代表从服务器获得的附件详细信息，包括附件文件名、附件下载地址、文件大小和附件 ID。AttachInfoXML 的实现和前面讲过的其他 XML 解析器类似，我们就不再复述。

获得附件 URL 之后，接下来就是下载附件。附件下载由 downAction 方法负责完成。在该方法中，我们先到 Document 目录中查看文件附件是否已存在，如存在则调用 openAttachFile 方法打开文件。否则，用 ASIHTTPRequest 进行下载，然后再用 openAttachFile 打开。这些代码在第 7 章中已进行过介绍，在此不再逐句解释。

```
- (IBAction)downAction:(id)sender {
   NSString* path=[NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
           NSUserDomainMask, YES) objectAtIndex:0];
   path=[path stringByAppendingPathComponent: lbAttachID.text];
   if ([[NSFileManager defaultManager] fileExistsAtPath:path]) {
      [self openAttachFile:path];
```

```
    }else if (fileUrl) {
        progressView.hidden=NO;
        NSURL *url = [NSURL URLWithString:fileUrl];
        ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
        [request setDownloadDestinationPath:path];
        [request setDownloadProgressDelegate:progressView];
        request.showAccurateProgress=YES;
        [request startSynchronous];
        [self openAttachFile:path];
        progressView.hidden=YES;
    }
}
```

openAttachFile：方法调用视图控制器 InboxAttachOpenVC 打开指定的文件附件：

```
-(void)openAttachFile:(NSString*)filePath{
    NSLog(@"file:%@",filePath);
    InboxAttachOpenVC* vc=[[[InboxAttachOpenVC alloc]init]autorelease];
    [self.navigationController pushViewController:vc animated:YES];
    [vc openFile:filePath];
    vc.title=lbAttachID.text;
}
```

InboxAttachOpenVC 的实现非常简单，它包含了一个 UIWebView 对象。通过 UIWebView 我们能浏览大部分的图片和文档格式，如 .jpg、.png、.doc、.text、.xls 和 .pdf。

19.4.4 新建邮件

新建邮件由 NewMailViewController 视图控制器实现，图 19-7 是新建邮件 NewMailViewController.xib 的界面设计。

其中标题输入框用于输入邮件标题。收件人和正文看似普通的 Text Field ，其实并不接受用户的输入。我们把它们的 delegate 设置为视图控制器 NewMailVC，同时实现了 textFieldShould BeginEditing 方法，目的是弹出新的输入界面。这样用户操作的空间会显得更加充裕一些：

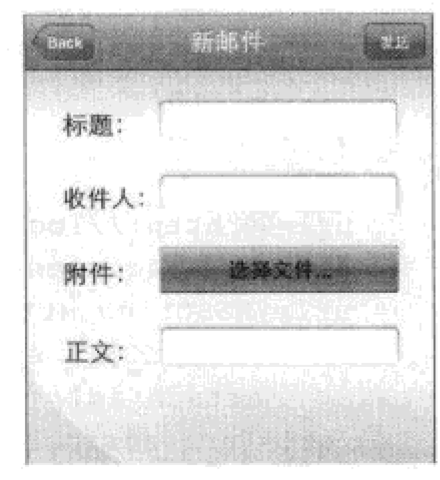


图 19-7 新建邮件的界面设计

```
-(BOOL)textFieldShouldBeginEditing:(UITextField *)textField{
    if (textField==tfSend) {
        if (_directoryVC==nil) {
            _directoryVC=[[directoryVC alloc]init];
        }
        _directoryVC.delegate=self;
        [self.navigationController pushViewController:_directoryVC animated:YES];
        return NO;
    }else if(textField==tfContent){
        if (editVC==nil) {
```

```
            editVC=[[EditVC alloc]init];
        }
        editVC.delegate=self;
        [self.navigationController pushViewController:editVC animated:YES];
        [editVC.textView becomeFirstResponder];
        [editVC setTextViewText:tfContent.text];
        return NO;
    }else
        return YES;
}
```

对于收件人输入框，我们提供的编辑界面是 directoryVC ，对于正文输入框，提供的是 EditVC。前者显示一个收件人选择列表，用户可以选择多个收件人，后者显示一个 Text View，用户可以输入大文本。这两个类将在后面介绍。

还有一个按钮是附件上传按钮，点击后将显示一个文件选择列表，用户在这个界面选择 Document 目录下的文件进行上传，上传成功后服务器会返回一个附件 ID。新建邮件时，只需带上这个附件 ID 就可在发送邮件时附带上附件。附件上传的视图控制器也将在后面介绍。

接下来是邮件的发送。点击发送按钮，将进行邮件的发送：

```
-(void)sendMailAction{
    if (tfTitle.text==nil || tfTitle.text.length==0) {
        showMessage(@"", @"请输入邮件标题");
        [tfTitle becomeFirstResponder];
        return;
    }
    if (tfSend.text==nil || tfSend.text.length==0) {
        showMessage(@"", @"收件人不能为空");
        return;
    }
    NSString*
    url=@"http://localhost:8080/AnyMail/sendmail.jsp?user=%@&pass=%@&title=%@&
    sender=%@&content=%@&attachid=%@";
    url=[NSString stringWithFormat:url,[loginXML sharedloginXML].username,
         [loginXML  sharedloginXML].pass,tfTitle.text,tfSend.text,tfContent.text,
         attach_id];
    url=[url stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
    [[NetworkModule sharedNetworkModule]postURL:url
                                        tag:kBusinessTagSendMail
                                      owner:self];
}
```

注意： 在提交 URL 之前，需要对 URL 字符串进行仔细检查。如果 URL 中包含有 unicode 字符，应当调用 stringByAddingPercentEscapesUsingEncoding 将 URL 字符串进行编码。否则 HTTP 请求不会成功。

发送邮件的实现与前面的其他 HTTP 请求并无不同，仍然分为两个步骤：使用 NetworkModule 发送异步 HTTP 请求，然后实现 NetworkModuleDelegate 委托。由于这次返

回的 XML 文件很简单，就是返回成功失败状态，我们就没有必要专门实现一个 XML 解析器了，直接用 XMLNode 去获取 XML 中的成功失败状态即可：

```
-(void)endPost:(GDataXMLDocument *)result business:(kBusinessTag)tag{
    if (tag==kBusinessTagSendMail) {
        XMLNode* node=[[[XMLNode alloc]init]autorelease];
        node.element=result.rootElement;
        NSString* s=[node stringValueFromPath:@"//response/code"];
        if ([@"success" isEqualToString:s]) {
            showMessage(@"", @"邮件发送成功");
        }else{
            showMessage(@"", @"邮件发送失败");
        }
    }
}
```

19.4.5 正文输入界面

接下来先介绍相对简单的 EditVC 视图控制器，它提供了 “新邮件”视图中“正文”输入框的编辑界面。

它的界面相对简单，只提供了一个 UITextView 控件。你可能会觉得奇怪：这样做有必要吗？为什么不在“新邮件”视图中直接提供一个 UITextView？

我们有这样做的理由。因为 iPhone 软键盘经常会遮挡住输入控件。软键盘从视图控制器的下方弹出，它会占据屏幕下端 216 像素的空间。

解决这个问题的方式有许多，有的复杂，有的简单。我们采用最简单的一种： 如果输入控件的位置位于视图控制器的偏下方，我们就单独提供新的输入界面。 这对我们来说不会有任何额外的开销，仅仅是一个简单的 UIViewController，也不会有任何复杂的代码。在新的界面中，我们可以把输入控件尽量靠屏幕偏上方安排，下方则由弹出后的软键盘占据。如图 19-8 所示。这样，无论如何我们的视线都不会被软键盘挡住了。

图 19-8 EditVC 视图控制器

PDG

为了能让 EditVC 将用户编辑的内容返回，我们定义了 EditVCDelegate 协议。协议只有一个方法：

```
-(void)editVC:(EditVC*)vc withText:(NSString*)text;
```

如果任何视图控制器需要弹出 EditVC 并获得 EditVC 编辑后的结果，那么它需要把自己设置为 EditVC 的 delegate。并实现 EditVCDelegate 协议。

而 EditVC 会在用户点击“确定”按钮时，回调 delegate 的委托方法：

```
-(void)performAction{
    [textView resignFirstResponder];
    if (delegate) {
        [delegate editVC:self withText:textView.text];
    }
}
```

然后，在 NewMailVC 中，我们实现 EditVCDelegate 委托协议，通过委托方法参数接收到用户在 EditVC 中编辑的文本，并关闭 EditVC：

```
-(void)editVC:(EditVC *)vc withText:(NSString *)text{
    tfContent.text=text;
    [self.navigationController popViewControllerAnimated:YES];
}
```

19.4.6 通讯簿

通讯簿接口有一个 XML 需要解析，我们先实现一个解析器 DirectoryXML。其实我们的 DirectoryXML 类并不仅仅是做 XML 解析这么简单，我们还想让它构建通讯簿的树状结构。

DirectoryXML 的源文件中其实定义了两个类：DirectoryXML 类和 Node 类。DirectoryXML 类负责 XML 文档的解析，Node 类负责描述树中的节点。

DirectoryXML 类是一个单例。因为我们想要一个全局的通讯簿，只要我们创建了一个通讯簿（DirectoryXML 对象），我们任何时候都可以引用它，而不用在视图中传来传去。

DirectoryXML 有一个 initWithXMLDocument 方法，用于加载一个 XML 文档，然后从文档的根元素开始，构建整棵树：

```
-(id)initWithXMLDocument:(GDataXMLDocument *)doc{
    self=[super init];
    if (self) {
        NSArray* a=[doc.rootElement children];
        root=[[Node alloc]init];
        root.name=@"root";
        root.hasChildren=a.count>0;
        root.isPeople=NO;
        root.nodeLevel=0;
        root.children=[[NSMutableArray alloc]init];
```

PDG

```
        root.isExpanded=YES;
        for (GDataXMLNode* each in a ) {
            Node* node=[[[Node alloc]initWithXMLNode:each level:root.nodeLevel+1]
                    autorelease];
            [root.children addObject:node];
        }
    }
    return self;
}
```

DirectoryXML 有一个虚拟的根元素对象：root。这个 root 是一个 Node 对象，它有一个 children 属性（NSArray），用于存放 XML 文档的子节点。注意，for 循环构成了递归，它遍历了 XML 文档的所有层级的子节点，将它们封装为 Node 对象。

Node 对象定义了如下属性：

```
@property(retain,nonatomic)NSString* ID, *name;
@property(nonatomic)BOOL isChecked;
@property(nonatomic)int nodeLevel;
@property(nonatomic)BOOL hasChildren;
@property(nonatomic)BOOL isPeople;
@property(nonatomic)BOOL isExpanded;
@property(retain,nonatomic)NSMutableArray* children;
```

可以看到，除了 ID 属性和 name 属性是来自于 XML 外，其他属性都是为了便于树的操作而额外定义的。比如，isChecked 属性用于标记该节点是否被用户选中；nodeLevel 属性用于标记该节点位于树的第几级（树的根节点是第 0 级，以此递增）；hasChildren 属性用于判断是否有子节点，很多时候我们要通过它来决定是否要进入下一级查找；isPeople 属性用于判断该节点是否是 XML 文档中的“linkman”元素；isExpanded 属性则决定是否展开该节点的子节点。

Node 定义了一个便利的初始化方法：

```
-(id)initWithXMLNode:(GDataXMLNode *)node level:(int)level{
    self=[super init];
    if (self) {
        NSArray* attrs=[(GDataXMLElement*)node attributes];
        [attrs enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {
            NSString* key=[(GDataXMLNode*)obj name];
            NSString* value=[(GDataXMLNode*)obj stringValue];
            objc_property_t property=class_getProperty([self class], [key cString-
                UsingEncoding:NSUTF8StringEncoding]);
            if (property){
                [self setValue:value forKey:key];
            }
        }];
        nodeLevel=level;
        hasChildren=node.children.count>0;
```

```
        children=[[NSMutableArray alloc]init];
        isPeople=[@"linkman" isEqualToString:node.localName];
        if (hasChildren) {
            [node.children enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL
                    *stop) {
                GDataXMLNode* _node=(GDataXMLNode*)obj;
                Node* aNode=[[[Node alloc]initWithXMLNode:_node level:nodeLevel+
                            1]autorelease];
                [children addObject:aNode];
            }];
        }
    }
    return self;
}
```

在方法中，我们根据参数传递的 XML 元素初始化 Node 的属性。注意，有的属性是通过推断而得出的。例如，hasChildren 属性根据 XML 元素的 children 数组元素数目是否>0来推断，如果大小为 0，说明节点没有子节点，hasChildren 为 NO，否则反之；isPeople 属性则从元素的名称进行推断，为“linkman”的节点 isPeople 为 YES；nodeLevel 则来自于 level 参数。

提示：为了加快遍历操作的速度，我们使用了 NSArray 的枚举方法 enumerateObjectsUsingBlock。

然后遍历 XML 元素的子节点。依次调用 initWithXMLNode 方法初始化这些子节点并添加到 children 数组。当然，它们的 nodeLevel 应该在父节点的基础上加一。

关于 Node（或树）的操作，我们定义了两个：

```
-(void)assessShowNodes;
-(void)assessCheckedNodes;
```

与之相关联的属性如下：

```
@property(strong,nonatomic,readonly)NSArray* checkedNodes;
@property(strong,nonatomic,readonly)NSArray* showNodes;
```

assessShowNodes 方法负责遍历整棵树，查找其中需要显示的节点。对于有的节点，如果它的 isExpanded 属性是 YES，则我们需要显示的不仅仅是节点本身，还要加上它的子节点，否则我们只显示节点本身即可。

这个方法执行后的结果会保存在 showNodes 属性中。当节点的 isExpanded 状态无改变时，我们要获取显示节点，只需访问 showNodes 属性而不必调用 assessShowNodes 方法，这样可减少遍历树的操作。遍历树是采用递归方法（countShowNodes: 方法）进行的，属于耗时操作，能减少一次则减少一次。

assessCheckedNodes 方法负责遍历树中的已选节点（除要求 isChecked 属性为 YES 外，

我们还要求 isPeople 属性也为 YES,因为我们规定部门节点是不可选的)。assessCheckedNodes 方法调用了递归方法 countCheckedNodes: 方法,执行后的结果保存在 checkedNodes 属性中,当 isChecked 状态无改变时,我们直接访问 checkedNodes 属性即可获得已选节点。

接下来我们来看通讯簿的视图控制器 directoryVC,它是一个普通的表视图控制器,拥有一个 TableView。当加载数据之后,显示如图 19-9 所示的界面。

directoryVC 的表视图使用了我们定制的单元格 directoryCell。directoryCell 是一个 UITableViewCell 子类,我们在上面使用了一个 UILabel 和一个 UIButton,如图 19-10 所示。

directoryCell 是实现是简单的,除了我们所熟悉的 getCell 工厂方法,还有实现了 layoutSubviews 方法。这是因为我们使用了 TableViewCell 的 indentationLevel 属性来计算节点缩进,因此必须实现此方法。

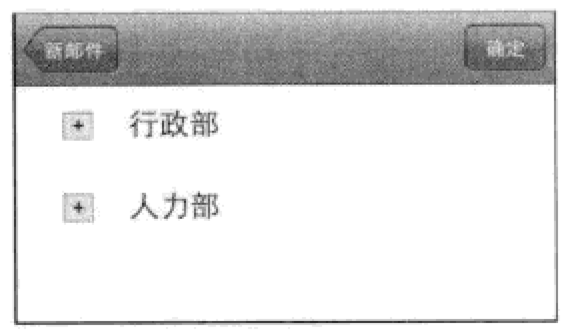


图 19-9 通讯簿显示界面

图 19-10 定制单元格 directoryCell

提示: 如果你不实现 layoutSubviews 方法,单元格的 intentationLevel 属性不会生效。

在 directoryVC 的 viewDidLoad 方法中用 NetworkModule 加载了网络数据。因此我们也在 NetworkModuleDelegate 的协议方法 endPost 中把 XML 数据封装到了 DirectoryXML 对象 directory。这个步骤你应该熟悉了,因此不再详细介绍。

我们来看看主要的 table view 数据源方法 cellForRowAtIndexPath,代码如下所示:

```
-(UITableViewCell*)tableView:(UITableView *)tableView cellForRowAtIndexPath:
        (NSIndexPath *)indexPath{
    static NSString* s=@"erb234dd3909";
    directoryCell* cell=(directoryCell*)[tableView dequeueReusableCellWithIdentifier:s];
    if (cell==nil) {
        cell=[directoryCell getCell];
    }
    NSArray* array=[directory.root showNodes];
    Node* info=(Node*)[array objectAtIndex:indexPath.row];
    cell.textLabel.text=info.name;
    if (info.isExpanded || !info.hasChildren) {
        cell.symbolLabel.text=@"-";
    }else
        cell.symbolLabel.text=@"+";
    if (info.isChecked) {
        cell.accessoryType=UITableViewCellAccessoryCheckmark;
```

```
    }else
        cell.accessoryType=UITableViewCellAccessoryNone;
        cell.indentationLevel = info.nodeLevel - 1;
    cell.indentationWidth = 25;
    [cell.expandButton setTag:(NSInteger)info];
    [cell setTag:(NSInteger)info];
    [cell.expandButton addTarget:self action:@selector(expandAction:) forControl
        Events:UIControlEventTouchUpInside];
return cell;
}
```

点击单元格中的按钮（+号/-号按钮），将展开和折叠下级子节点。这是由 expandAction 方法负责完成的：

```
-(void)expandAction:(UIButton*)sender{
    Node* node=(Node*)sender.tag;
    if (node.hasChildren) {
        node.isExpanded=!node.isExpanded;
        [directory.root assessShowNodes];
        [table reloadData];
    }
}
```

这个方法很简单，修改节点的 isExpanded 属性，然后刷新表视图。

选择收件人由单元格的选择事件完成，即 table view 的 didSelectRowAtIndexPath 委托方法：

```
-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
        *)indexPath{
    [tableView deselectRowAtIndexPath:indexPath animated:YES];// deselect cell
    UITableViewCell* cell=[tableView cellForRowAtIndexPath:indexPath];
    Node *node=(Node*)cell.tag;
    if (node.isPeople) {
        node.isChecked=!node.isChecked;
        [tableView reloadData];
    }
}
```

这个方法同样简单，修改节点的 isChecked 属性，然后刷新表视图。

当用户点击“确定”按钮，directoryVC 将以回调的方式通知委托对象，用户的操作已经完成：

```
-(void)performAction{
    if (delegate) {
        [directory.root assessCheckedNodes];
        [delegate directoryVC:self selectedNode:[directory.root checkedNodes]];
    }
}
```

委托对象 delegate 是一个实现了 directoryVCDelegate 协议的对象：

```
@property (assign,nonatomic)id<directoryVCDelegate> delegate;
```

directoryVCDelegate 是 directoryVC 定义的一个委托协议，它只定义了一个必须实现的方法：

```
-(void)directoryVC:(directoryVC*)vc selectedNode:(NSArray*)nodeArray;
```

其中第二个参数中包括用户所选的节点数据（Node 对象）。

在 NewMailVC 中，我们实现了这个协议，并将用户选择的收件人显示在 TextField 中，如图 19-11 所示。

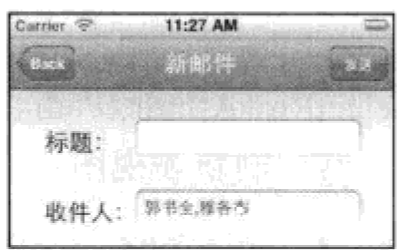


图 19-11 显示用户在通讯簿中的选择

提示：你也可以从节点中获取到收件人的 ID。

19.4.7 附件文件的上传

附件文件选择需要单独的视图控制器 uploadVC。uploadVC 显示了一个文件列表供用户选择。用户选择之后，点击“上传”即可，如图 19-12 所示。

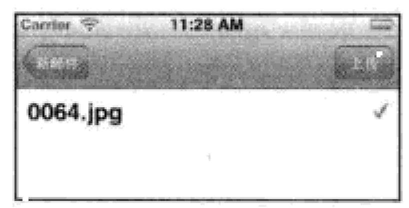


图 19-12 文件上传选择界面

在 viewDidLoad 方法中，我们调用了 listFiles 方法获取 Documents 目录的文件列表并保存到 allFiles 数组中。

```
allFiles=[[NSArray alloc]initWithArray:listFiles(DOCUMENT_PATH)];
```

在 viewDidLoad 方法之后，Table View 的数据源方法将自动被调用。在 Table View 的 cellForRowAtIndexPath 数据源方法中，我们读取了 allFiles 中的文件名，显示到单元格中。并与 selectedFile 字符串进行比较，判断该单元格的选中状态。如果是已选中，则在单元格的 accessory view 中显示一个对勾，代码如下所示。

```
-(UITableViewCell*)tableView:(UITableView *)tableView cellForRowAtIndexPath:
    (NSIndexPath *)indexPath{
   static NSString* s=@"erbd393a1";
   UITableViewCell* cell=[tableView dequeueReusableCellWithIdentifier:s];
```

```
    if (cell==nil) {
        cell=[[[UITableViewCell alloc]initWithStyle:UITableViewCellStyleDefault
                                reuseIdentifier:s]autorelease];
    }
    NSString* filename=[allFiles objectAtIndex:indexPath.row];
    cell.textLabel.text=filename;
    if ([filename isEqualToString:selectedFile]) {
        cell.accessoryType=UITableViewCellAccessoryCheckmark;
    }else
        cell.accessoryType=UITableViewCellAccessoryNone;
    return cell;
}
```

然后是表视图的委托方法 didSelectRowAtIndexPath。在这个方法中我们将单元格所对应的文件名拷贝到 selectedFile 对象中，然后刷新表视图：

```
-(void)tableView:(UITableView  *)tableView  didSelectRowAtIndexPath:(NSIndexPath
      *)indexPath{
    [tableView deselectRowAtIndexPath:indexPath animated:YES];// deselect cell
    selectedFile=[allFiles objectAtIndex:indexPath.row];
    [tableView reloadData];
}
```

注意：这样的方法只实现了单选，即一封邮件只有一个附件。

当用户点击“上传”按钮后，触发 performAction 方法。在 performAction 方法中，我们判断 selectedFile 字符串是否为空。如果不为空，则调用 upload 方法：

```
-(void)performAction{
    if (selectedFile && selectedFile.length>0) {
        [self upload];
    }else{
        showMessage(@"", @"请选择上传的文件");
    }
}
```

在 upload 方法中，我们用 ASIHFormDataRequest 将文件以模拟表单的方式提交到服务器的 UploadServlet：

```
-(void)upload{
     NSURL* url=[NSURL URLWithString:@"http://localhost:8080/AnyMail/UploadServlet"];
     request = [ASIFormDataRequest requestWithURL:url];
     NSStringEncoding enc=NSUTF8StringEncoding;
     [request setStringEncoding:enc];
     NSString* fullname=[DOCUMENT_PATH stringByAppendingPathComponent:selectedFile];
     [request setFile:fullname forKey:@"attach"];
     [request setDelegate:self];
```

```
    [request setDidFinishSelector:@selector(responseComplete)];
    [request setDidFailSelector:@selector(responseFailed)];
    [request startAsynchronous];
}
```

关于使用 ASIFormDataRequest 进行文件上传，我们在第 7 章“网络”关于 ASIHTTPRequest 框架的介绍中有过介绍。

最终，在文件传送完毕后，我们取得服务器响应的内容，用于设置 attach_id 的值，并将 attach_id 传递给委托对象。委托对象是实现了 uploadVCDelegate 协议的对象，该协议只定义了一个方法 uploadVC:fileName:attachId:。通过回调委托对象的 uploadVC:fileName:attachId:协议方法，我们把 attach_id 和 selectedFile 传递给了委托对象：

```
-(void)responseComplete{
    // 请求响应结束，返回 responseString
    attach_id = [request responseString];
   NSLog(@"attach_id:%@",attach_id);
   if (delegate) {
      [delegate uploadVC:self fileName:selectedFile attachId:attach_id];
   }
}
```

提示：在服务器上应存在“/data”目录和“/data/temp”目录。如果不存在，请新建。上传文件将放在服务器“/data”目录下。

19.5 本章小结

iOS 企业应用最重要的特征不是别的，正是企业网络向移动终端的扩展。iPhone 的成功，就在于它把移动互联的概念带到了我们的生活当中。因此本章以一个典型的 iOS 网络应用作为实战项目，来顺应企业应用正在向移动网络扩展的最新趋势。

当你阅读完本书，可对 iOS 企业应用和苹果的 iOS SDK 框架有一个更加深入和全面的理解，并以此为基础继续钻研移动开发技术，搭建自己的企业应用。