移动开发经典丛书

PEARSON

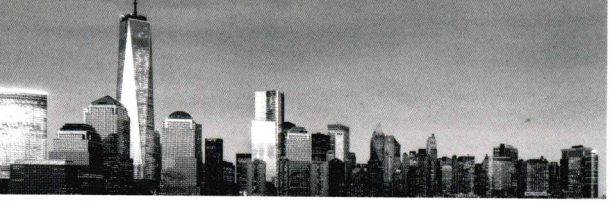
扫码关注"iOS猿吧"

开发技术随时关注

精通iOS框架(第2版)

[美] Kyle Richter　Joe Keeley　著
冯宗翰　江铭　朱倩　译

清华大学出版社



Mastering iOS Frameworks:

Beyond the Basics, Second Edition

苹果的iOS SDK提供了非常强大的框架集合，即使到现在为止，找到这些框架的详细介绍和使用方法都有一定困难。不过，借助本书的实用见解和经过验证的代码，你可以使用苹果公司提供的这些框架来创建更有创意、更实用的应用，同时也会让编写代码的过程更加快速、让代码更加可靠，也让应用更加成功、更有市场。

Kyle Richter和Joe Keeley关注专业开发者每天都会用到的一些中高级技术，涉及的技术范围从支持社交网络到安全保障，从Core Data到iCloud，甚至有关Apple Watch的内容都包含在内。

作为一本便捷的模块化参考书，几乎每一章都会包含一个完整的Objective-C示例程序。另有一个跨章节的Game Center案例，也正好演示了多个iOS功能是如何进行组合的。

主要内容：

- 添加类似物理效果的动画和动作到UIView视图
- 使用Core Location确定设备的位置，显示自定义地图并实现地理围栏
- 在游戏和其他应用中加入带有社交元素的排行榜功能
- 访问音乐和图片集
- 使用HealthKit实现带有健康和健身功能的应用
- 通过HomeKit实现家居智能化
- 使用JSON在平台间传输数据
- 设置本地和远程通知
- 使用CloudKit实现数据的远程存储和同步
- 实现访问应用的extension功能
- 轻松添加AirPrint功能
- 在iOS 8和Yosemite设备之间提供连续的Handoff功能
- 学习有效使用Core Data
- 通过社交框架在应用中整合Twitter和Facebook
- 使用Grand Central Dispatch机制处理多线程任务
- 使用Keychain和Touch ID保护用户数据
- 定制集合视图
- 掌握大部分的手势识别方法
- 创建和发送“通行证”到Passbook
- 调试工具的介绍及应用的优化

Kyle Richter是MartianCraft公司的CEO，该公司赢得了移动开发工作室大奖。他有着超过20年苹果生态系统的开发经验，他出版的有关iOS开发的著作包括*Beginning iOS Game Center Development*、*Beginning Social Game Development*和*iOS Components and Frameworks*。

Joe Keeley是MartianCraft公司的合伙人和首席工程师，为iOS客户项目提供技术指导并主导了许多成功的项目。他从Apple II开始就热衷于程序开发，并撰写了多本关于iOS和Mac技术方面的著作。

清华大学出版社数字出版网站
WQBook 书文局泉
www.wqbook.com

PEARSON
www.pearson.com

源代码下载

ISBN 978-7-302-43381-1
9 787302 433811 >
定价：79.80元

移动开发经典丛书

精通 iOS 框架

(第 2 版)

[美] Kyle Richter
Joe Keeley 著
冯宗翰 江铭 朱倩 译

清华大学出版社
北 京

北京市版权局著作权合同登记号 图字：01-2015-4610

图书在版编目(CIP)数据

精通 iOS 框架：第 2 版 / (美) 里克特 (Richter, K.)，(美) 姬莉 (Keeley, J.) 著；冯宗翰，江铭，朱倩 译. —北京：清华大学出版社，2016

(移动开发经典丛书)

书名原文：Mastering iOS Frameworks: Beyond the Basics, Second Edition

ISBN 978-7-302-43381-1

Ⅰ. ①精… Ⅱ. ①里… ②姬…③冯…④江… ⑤朱… Ⅲ. ①移动终端－应用程序－程序设计 Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字(2016)第 075620 号

责任编辑：王　军　李维杰
装帧设计：牛艳敏
责任校对：成凤进
责任印制：宋　林

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
装 订 者：三河市少明印务有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　印　　张：29　　字　　数：742 千字
版　　次：2016 年 5 月第 1 版　　印　　次：2016 年 5 月第 1 次印刷
印　　数：1~3000
定　　价：79.80 元

产品编号：065611-01

译 者 序

苹果公司的产品一直在引领着智能设备的发展，苹果公司的伟大就在于它的每一个产品都臻于完美，让使用者赞叹不已。我也是一个不折不扣的“果粉”，不过我不满足于使用它的产品，我也希望自己能够编写出改变世界的产品。虽然这个理想可能比较大，不过正是抱着这个梦想的开发者们创造出了一个又一个神话，从“疯狂的小鸟”到 Snapchat，从微信到 Uber，无一不让使用者疯狂。智能手机已经改变了人们的生活习惯，“互联网+”的时代也已经正式到来，而苹果公司和它的 iOS 系统无疑是这个领域的佼佼者。一方面 iOS 平台因为它的封闭而饱受指责，另一方面也正是得益于这样的生态系统，保护了开发者的利益，让他们有更多的动力专注于创造更优秀的产品，这也正是 App Store 中的应用无论在界面上还是用户体验上都优于其他平台的原因之一。

我从事 iOS 开发也有将近 10 年的时间了，记得最初手头甚至没有一本像样的参考资料。可想而知，在学习开发技术的最初一段时间有多么痛苦和无助，唯一能够帮助到我的就是苹果给出的官方文档，不过全英文的解释极大延缓了我对技术的探索。不过随着 iPhone 产品越来越出色，越来越多的人开始注意到 iPhone 的潜力，也有越来越多的开发者转到 iOS 平台，市面上有关 iOS 开发的入门教程也越来越多了。随着一步一步走进 iOS 的世界，你会觉得入门教程已经不能满足你对技术的渴求了，而市面上真正介绍高级开发技术的书又很少，大多数教程也非常晦涩难懂。本书的作者对 iOS 开发有着非常丰富的经验，同样有着很深的理解。他们很准确地抓住了从各种框架的介绍入手这一切入点，正是这些框架的熟练使用才能让一个菜鸟慢慢成长为一个资深开发者。这一点我不得不佩服作者的智慧，在我自己学习 iOS 开发遇到瓶颈的时候，多么希望有这样一本参考书，在我遇到问题的时候可以直接拿来解决问题，不过那时候还没有这本书，而现在的读者就幸运多了。

本书不仅有着非常明确的切入点，在成书的过程中也有过人之处。我们知道，学习编程技术最好的办法就是亲自动手实践，本书也正是采用了这种方法让读者能够将理论和实践相结合，最终通过自己动手完成对应功能的开发来掌握相应的技术。另外本书的结构也很有特色，除了几章关联度很强外，几乎每一章都是独立的，你完全可以在需要某一技术或知识的时候直接翻到相应的部分，同时本书又几乎覆盖了 iOS 中高级开发技术中所有的框架，所以只要把这本书放在手边，就可以应对许多开发中常见的问题了。

能够有幸拜读本书并完成本书的翻译工作我觉得很幸运，就像是同这样一位 iOS 开发界的大佬对话一般，怀着欣喜和紧张完成了本书的翻译工作，期间不敢有一点怠慢和疏忽，在

这里要感谢清华大学出版社的编辑对我的帮助，他们为本书的翻译投入了巨大的热情并付出了很多心血。没有他们的帮助和鼓励，本书不可能顺利付梓。还要感谢无锡职业技术学院领导和同事们的帮助，正是他们对我的支持让我可以专心于本书的翻译工作。也谢谢我的爱人和家人，是她们的支持让我能够专心完成本书的翻译工作。虽然在翻译过程中力求“信、达、雅”，但是鉴于译者水平有限，错误和失误在所难免，如有任何意见和建议，请不吝指正。感激不尽！本书所有章节由冯宗翰翻译，参与本次翻译的还有江铭、朱倩、孙婷婷、许亦男、孙伟、冯树彪、张怀洲、李志锋、李爽等，在此一并谢过。

最后，我相信这本书一定会成为开发者身边一本不可或缺的参考书，而这也正是本书作者最大的希望，相信大家一定能创造出改变世界的产品，向着梦想前进！

冯宗翰

序　言

从 2008 年 iPhone SDK(现在叫 iOS SDK)的第一个测试版发布之日起，我就一直从事有关 iOS 的开发工作。那时我主要关注有关 Mac 桌面程序的开发，没有过多考虑移动应用的开发。

如果你希望成为一个早期开发者，那你只能靠自己了。苹果公司一贯的做法就是文档非常少，并且由于访问 SDK 需要 NDA——秘密解码环，因此最初你只能靠自己。你还不能在 Google 上或打开 StackOverflow 寻求到帮助，并且那时也一定没有任何介绍 SDK 的书籍。

从苹果发布最初的 iPhone 到现在已经走过漫漫 8 年时光(是的，真的只有 8 年)。iPhone SDK 现在也被称为 iOS SDK。有关 iOS 开发的书籍和博客，以及播客和研讨会层出不穷。从 2009 年起，WWDC 大会变得更加难以参加，使得新老开发者在学习最新的平台技术时变得越发困难。尤其对于 iOS 开发者，要学的新东西真的太多了。

作为一名 iOS 开发者，我遇到的最大的难题就是设法驾驭苹果工具箱(kit)中所有的组件和框架。iOS HIG 本能帮助到我们，不过它对于组件和框架的介绍还不够详细深入。现在我们确实能够通过 Google 或者结合 StackOverflow 找到资料，不过这些资料一般都仅解释了如何去做，很少探究为什么要那样做，通常也无法做到很详细地分析。

所以 Kyle 和 Joe 决定这样做——给出所有这些框架的详细介绍，让读者可以全面了解组成 iOS SDK 的核心框架。

很荣幸与 Kyle 和 Joe 相识多年。他们是我所见过的最聪明的开发者。这些年来他们都各自编写了大量优秀的应用，并通过分享他们的经验为 iOS 开发社区不断贡献自己的力量，不断在研讨会上发表出色演讲，出版有关 iOS 开发的书籍。如果你有任何关于 iOS 的问题，有机会得到 Kyle 和 Joe 的解答将会是一件很美妙的事情。

不过让他们如此优秀的原因还不仅是他们如百科全书一般的 iOS 开发知识，更在于他们愿意同遇到的每个人分享这些宝贵的知识。Kyle 和 Joe 没有竞争对手，有的只是朋友。

Kyle 和 Joe 对于 iOS SDK 的深入理解贯穿本书，这也是我喜欢本书的原因之一。本书对每个组件都进行了详细介绍，有些内容甚至在网上都很难找到。

我还非常喜欢本书的结构。读者不需要从头到尾阅读。相反，你可能因为需要了解如何实现集合视图而翻开本书，或者在想学习如何在后台线程上运行任务时打开本书。在你需要本书时只需打开它，找到解决办法，将其在自己的代码中实现，然后再把它放回书架上直到下次再遇到困难。这就是该书能够成为 iOS 开发者最重要的一本参考书籍的原因，不管你是

初学者还是经验丰富的老手。你可能觉得自己精通 Core Location 和 MapKit，不过我觉得你在本书中一定能找到以前没有接触过的知识。

Kyle 和 Joe 为人非常谦虚，从不骄傲自大。他们从来不认为自己比其他开发者优秀。他们将这种精神一点点灌输到 Mac 和 iOS 开发者社区，使其成为一个开发者之间互助学习的大平台，对这个行业的发展帮助很大。这本著作也是他们无私分享多年经验和知识的另一个见证。

本书同 Mark 和 LaMarche 及 Sadun 的著作一样，将始终伴随在我手边。当我在 2008 年第一次开始做 iOS 开发时我多希望有这么一本书。现在终于如愿以偿，有了这本书就方便多了。

— Kirby Turner

White Peak Software 公司首席程序员，*Learning iPad Programming: A Hands-On Guide to Building iPad Apps, Second Edition*(Addison-Wesley Professional)一书的作者，Cocoa 开发者社区的管理者和研讨会的热衷者。

作者简介

Kyle Richter 是 MartianCraft 公司的 CEO，MartianCraft 公司曾赢得 Mobile Development Studio 称号。Kyle 在 20 世纪 90 年代初就开始从事软件开发工作，并始终专注于基于苹果公司平台的开发，他在 iOS 开发方面已经出版和共同出版了多种著作，包括 *Beginning iOS Game Center Development*、*Beginning Social Game Development* 和 *iOS Components and Frameworks*。利用管理 MartianCraft 公司每日运营的同时，Kyle 还要在全球出差，介绍有关开发和公司管理的经验。现在佛罗里达群岛是他的家，因为在那里他要花时间陪伴他可爱的边境牧羊犬。你也可以通过@kylerichter 在 Twitter 上找到他。

Joe Keeley 是 MartianCraft 公司的合伙人及首席工程师。Joe 为 iOS 客户项目提供技术指导并主导了许多成功的项目。他从 Apple II 开始就热衷于程序开发，在他的职业生涯中从事过许多不同技术和系统项目的开发。在美国，Joe 出版了多种有关 iOS 和 Mac 技术开发的参考书。Joe 和妻子及两个女儿居住在科罗拉多州丹佛市，在闲暇时间他喜欢从事击剑运动。他在 Twitter 上的名字是@jwkeeley。

前　言

欢迎阅读《精通 iOS 框架(第 2 版)》!

对于现在的读者来说，几乎有上百种“iOS 入门教程”可供选择，介绍特定专题的进阶教程也数不胜数，例如专门介绍 Core Data 知识或者专门介绍有关数据安全方面的教程。不过令人感到困惑的是还没有一种书能够作为初学者向更高级内容前进的桥梁。

撰写本书旨在向读者介绍中高级的开发知识，因为这些看起来零散的框架很多都无法单独出书。并不是这些框架不够吸引人，而是因为它们还不能算是一个很大的专题，不需要过多篇幅的介绍。从如何使用 JSON 到如何访问照片库，这些框架都是专业 iOS 开发者每天都会用到的，不过市面上少有书籍介绍它们。

此外，对很多进阶内容的讨论也要考虑到面对的读者可能是初学者。翻开一本 500 页的 Core Data 教程也是需要勇气的，不过还好本书第 15 章给出了一个简捷的 Core Data 入门教程。其他有关类似的入门教程还包括调试和工具、TextKit、HomeKit、HealthKit 和 CloudKit 等章节。

对于 Game Center 排行榜和成就榜、AirPrint、音乐库、地址簿和 Passbook 的内容则给出详细全面的讲解。无论你是刚刚完成第一个 iOS 项目开发的初学者还是一名经验丰富的资深开发人员，都可以从本书中找到对你有帮助的内容。

书中所有章节的内容都根据 iOS 8 版本进行了更新。如果你遇到有关兼容性的问题请联系我们，我们会发布更新并进行修改。

如果你对本书的内容有任何建议，或者发现并修改了本书的一些错误，它们都对本书后续版本有很大的帮助，可以通过 mastering.ios.frameworks@gmail.com 联系我们。我们非常愿意听取任何能使本书变得更加完善的建议，并会不断致力于让本书更加完美。

需要具备的知识

本书尽力让所有的示例程序和知识讲解更加简单易懂，不过它毕竟还是一本针对中高级开发者的参考书。所以为了更好地使用这本书，你需要具备基本的 iOS 开发知识，以及 Objective-C 和 C 语言的知识，熟悉 Xcode、Developer Portal、iTunes Connect 和 Instruments 工具的使用。学习 Objective-C 和 iOS 的内容时可以参阅 Stephen G. Kochan 撰写的 *Programming in Objective-C* 一书和 Maurice Sharp、Rod Strougo 及 Erica Sadun 共同撰写的 *Learning iOS Development* 一书。

准备工作

虽然可以借助 iOS 模拟器开发并测试 iOS 应用，不过我们还是建议你至少拥有一款 iOS 设备用于测试：

- Apple iOS Developer Account：iOS 开发工具 Xcode 和 iOS SDK 的最新版本可以从苹果公司网站上的 Developer Portal 进行下载(http://developer.apple.com/ios)。要在 App Store 上发布应用或者在一台个人设备上安装并测试应用，需要支付每年 99 美元的开发者账号费用。
- Macintosh Computer：要开发 iOS 程序并运行 Xcode，你需要一台能够运行最新发布的 OS X 系统的 Mac 机器。
- Internet Connection：iOS 开发中的很多功能都需要你的 Mac 机器和 iOS 设备保持网络连接。

本书结构

除了个别章节(Game Center 和 Core Data)之外，本书的每一章都是独立的。你可以按照顺序从头到尾阅读本书，也可以跳过一些你暂时不需要的专题而直接找到你需要的内容，我们撰写本书的目的就是使其成为一种能够解决大部分 iOS 开发任务的快捷参考书。

下面是各章简介：

第 1 章“UIKit Dynamics”：iOS 7 增加了 UI Kit Dynamics 元素，可以让 UIView 添加模拟物理运动的动画效果。你会学到如何给对象添加标准的动态动画效果、设置物理属性等功能。从重力效果到元素属性设置，按照从易到难的顺序介绍 7 种效果的实现方法。

第 2 章“Core Location、MapKit 和 Geofencing 框架”：iOS 6 带来了全新的苹果地图和相关数据。该章会介绍如何通过使用 Core Location 来确定设备的位置，如何在应用中显示地图，以及如何在地图上使用自定义标注、覆盖和弹出气泡。还会介绍如何设置地理监控区域(地理围栏)，当设备进入或离开一个区域时设备会有提醒。

第 3 章“排行榜”：Game Center 排行榜提供了一种非常简单的方式让你的 iOS 游戏或应用增加社交元素。该章会带来一个名为 Whack-a-Cac 的 iPad 游戏，我们为其添加一个排行榜。用户将会学到实现 Game Center 排行榜的所有步骤，同时也会对如何实现带有自定义界面的排行榜有所了解。

第 4 章“成就系统”：该章继续使用前几章介绍的 Whack-a-Cac 游戏。你将学习到如何在这个 iPad 游戏中实现成就系统。从如何使用 iTunes Connect 到显示成就进度，该章为你提供了所有快速创建成就系统所需的信息。

第 5 章“Address Book 框架初步”：很多项目现在都将地址簿整合到应用中，Address Book 框架是 iOS 系统最老的几个框架之一。该章你会学到如何使用这个框架，学习如何使用用户选取器、如何访问源地址数据以及如何修改并保存该数据。

第 6 章“Music Libraries 框架”：该章介绍如何在一个自定义应用中访问用户音乐库，包括如何查看具体音乐的信息数据以及如何从专辑中选择并播放一首乐曲。

第 7 章“实现 HealthKit 框架”：HealthKit 用于实现在应用间共享健康相关信息的功能。该章介绍如何开始使用 HealthKit，之后介绍如何访问 HealthKit 中的数据，以及如何读取和

写入不同类型的健康相关数据。

第 8 章“实现 HomeKit 框架”：该章介绍如何开始使用 HomeKit，它可以让 iOS 设备同智能家居设备进行通信。其中还对如何设置 HomeKit 进行了介绍，以及如何发现、开启设备和如何同这些设备进行互动，例如灯、锁或仓库大门的遥控器等。

第 9 章“JSON 的使用和解析”：JSON 的全称为 JavaScript Object Notation，是一种能够在不同平台和架构间传输数据的轻量级数据传输协议。所以它成为 iOS 客户端和服务器间传输复杂数据时最常用的协议。该章会介绍如何从一个现有的对象创建 JSON 以及如何将 JSON 解析为 iOS 对象。

第 10 章“通知机制”：iOS 支持两种类型的通知机制，分别是本地通知和远程通知。本地通知主要用于设备没有连接网络的情况；远程通知需要通过网络再经过苹果公司的 Push Notification Service 服务器推送通知到设备。该章会介绍两种通知机制的差异，并演示如何在一个应用中创建和使用这两种通知机制。

第 11 章“基于 CloudKit 的云存储”：CloudKit 提供公共的和私人的数据存储模式。该章会介绍基础的 CloudKit 概念，并通过创建一个应用来演示如何使用 CloudKit 实现远程存储和同步个人及公共数据的功能。

第 12 章“extension”：extension 机制提供了一种在应用沙盒之外访问应用功能的功能。该章介绍几种不同类型的 extension，并演示如何创建一个 Today extension 和一个 Apple Watch extension。

第 13 章“Handoff”：Handoff 是 iOS 8 和 Yosemite 最新引进的一种应用延续机制，它可以让用户在不同设备间进行切换，而应用无缝地在设备间进行延续。该章会介绍基础的 Handoff 机制，并演示根据开发者定义的 activity 和基于文档的 activity 实现 Handoff 功能。

第 14 章“AirPrint”：AirPrint 可以让用户通过无线方式在支持 AirPrint 的打印机上打印文档和图片，不过该框架属于 iOS 提供的众多框架中不太常用的。该章介绍如何在应用中快速有效地添加 AirPrint 打印功能。在该章的最后，你会学习到打印视图、图片、PDF 文档甚至渲染 HTML 文件。

第 15 章“开始使用 Core Data”：该章会介绍如何让应用使用 Core Data、如何设置 Core Data 模型，以及如何在应用中实现最常用的 Core Data 工具。如果你没有时间钻研那本 500 页的 Core Data 参考书，那么该章再适合你不过了。

第 16 章“使用社交框架整合 Twitter 和 Facebook”：整合社交元素是智能计算的未来，现在公认的做法是希望将所有应用都加入社交元素。该章会教你如何使用 Social 框架向应用中添加 Facebook 和 Twitter 功能。你会学到如何使用内置的整合功能创建新的 Twitter 和 Facebook 消息，还会学到如何从服务器获取反馈信息以及如何解析得到的数据。最后介绍如何使用该框架从自定义的用户界面上发送消息。完成该章的学习之后，你会对 Social Framework 有很深的理解，可以自如地在应用中添加 Twitter 和 Facebook 功能。

第 17 章“后台任务处理”：当应用离开前台运行时能够在后台继续完成某些任务是一个很重要的功能，这个功能在 iOS 4 版本中被加入进来，并随着时间的推移更加完善。该章会介绍当应用离开前台后是如何在后台运行的，以及如何执行 iOS 指定的一些特殊后台任务。

第 18 章“多线程开发的性能”：在主线程执行一些资源占用高的程序时可能会使系统变慢。该章会介绍一些由 Grand Central Dispatch 机制提供的方法，用于处理由于影响主线程性

能表现的复杂并发问题。

第 19 章“使用 Keychain 和 TouchID 保护并访问数据”：保护用户数据的安全性是一件非常重要的事，但有时却被粗心的开发者忽略了。几年前甚至有一家大公司因为使用明文对用户信用卡信息和密码进行存储而被大家批评。该章首选会介绍使用 Keychain 来保护用户数据，然后将开发涉及的安全性问题作为整体进行详细讲解。通过该章的学习你可以在用户设备上使用 Keychain 对任何小规模数据类型进行保护，这样用户就可以放心地使用这些数据了。

第 20 章“处理图片和过滤器”：该章首先介绍一些基础的图片处理技术，之后介绍一些有关如何应用过滤器的高级 Core Image 技术。示例程序对所有 Core Image 提供的方法给出了演示，并创建过滤链来实现实时交互功能。

第 21 章“集合视图”：集合视图是 iOS 6 中引入的一个功能强大的 API，可以让开发者更加灵活地处理可滑动界面的布局以及基于单元格的内容排版。除了新的布局功能之外，集合视图还提供了精彩的动画效果，在集合视图内容的淡入淡出及视图间的切换时都可以添加动画效果。

第 22 章“TextKit 介绍”：iOS 7 中引入了 TextKit 框架，它非常容易上手，是对 Core Text 非常重要的扩展。TextKit 可以让开发者在应用中提供非常丰富且互动性很强的字体格式。虽然 TextKit 是一个比较大的话题，不过该章会介绍一些基础的知识并实现一些常见的功能，例如从环绕图片添加文字到内嵌自定义字体属性。完成该章的学习之后，其实是为后面更深入地学习 TextKit 打下非常坚实的基础。

第 23 章“手势识别”：该章会介绍如何在应用中使用手势识别。同直接处理和解析触碰数据不同的是，手势识别对象可以提供一个简单且干净的方法来识别一些常见的手势动作并给予反馈。此外，还可以自定义手势并使用手势识别对象对其进行识别。

第 24 章“访问照片库”：现在 iPhone 实际上已经成为最流行的照相机，看看 Flickr 上用户上传的大量图片就可以证明这一点。该章会介绍如何访问用户照片库，并在自定义应用中处理这些照片和视频。示例程序演示了如何创建一个和 iOS 8 版本的 Photos.app 类似概念的应用。

第 25 章“Passbook 和 PassKit”：iOS 6 中苹果公司引入了 Passbook，它是一个独立的应用，用于保存用户的各种“卡”，或者诸如飞机票、优惠券、会员卡和演唱会门票等凭证。该章会介绍如何设置通行证，以及如何创建并发布它们，还有如何在应用中同它们进行交互。

第 26 章“调试和工具”：程序开发最重要的一部分工作就是能够调试和优化软件。很少有书籍会介绍这方面的内容，即使是简单的介绍也很少见。本章为你讲解了在 Xcode 中如何调试程序以及使用工具分析程序的性能。首先我们讲述了计算机错误的历史，之后逐步介绍了有关一些常见错误的提示和识别方法。然后简单介绍了断点的使用和调试器命令，并使用 Time Profiler 和 Leaks 工具分别对程序代码和程序内存的使用情况进行了分析。学习完该章后，你将对如何在模拟器和真机设备上定位程序错误及调试 iOS 应用打下坚实基础。

示例程序

本书的每个章节都是独立的(除了个别章节)，同时除了第 26 章“调试和工具”之外，所

有的章节都具有自己的示例程序。第 3 章“排行榜”和第 4 章“成就系统”使用同一个示例项目，基于这个项目演示各自的功能。每章都会对示例程序进行简单的介绍，并逐步让读者理解示例程序中包含的一些复杂内容，而不是直接介绍本章的理论内容。

我们致力于让示例程序简单易懂，所以有些代码可能没有进行很好的优化，可能也不是解决特定问题的最优方法。所以在每章内容中我们都会指出一些在实际创建应用时哪些步骤是不合适的。示例程序不是独立的应用，它主要用于演示相应章节所介绍的功能。本书的示例程序被特意设计为通用程序，所以读者应该关注章节中介绍的主要内容而不是那些同相应章节无关的代码。为将一些非必要的组件从示例程序中删除，我们进行了大量的工作，同时将代码行尽可能压缩。

很多读者看到示例程序中的代码可能会觉得意外，因为没有使用 Swift 语言，而是用的 Objective-C，本书就是这样设计的。因为所有的 API 都是用 Objective-C 写的，所以使用 Objective-C 可以更加容易地进行互动，而使用 Swift 还需要额外添加解释起来比较复杂的层。如果读者习惯使用 Swift 后，可以很容易地将概念移植到 Swift 中。示例程序前缀为“ICF”，大部分示例程序的名字都是章节标题名。

当学习 Game Center 章节时，bundle ID 关联了我们个人苹果账号的真实应用，以确保例子能够继续运行。这个例子还可以让读者学习到当开发者对同一个示例程序进行互动时会生成多个用户数据。对于 iCloud、推送通知和 Passbook 这 3 章，应用需要的设置都在相应的章节内容中进行了全面介绍，要在实际工作中使用这些程序，读者必须为自己的开发者账户创建一个新的 App ID 才可以。

获取示例程序

要随时获取最新版本的源代码，可以访问 https://github.com/dfsw/icf 网站下面的 Mastering 文件夹，也可以访问 http://www.tupwk.com.cn/downpage。这些代码都是面向公众的且是开源的。源代码按照章节分成各自的压缩文件，包含 Xcode 项目，每个章节都只包含一个项目。我们鼓励读者提供有关源代码的反馈信息并提出建议，这样我们就可以让工作更加严谨，并在本书出版之后不断对其进行改进。

安装并使用 GitHub

Git 即版本控制系统，多年来深受开发者的喜爱。要使用 GitHub 上的代码，你首先需要在 Mac 上安装 Git。在 Xcode 命令行工具安装包中包含一个命令行版本的 Git，或者你还可以通过 http://git-scm.com/downloads 地址找到 Git 的安装文件。此外，Git 还有许多前端 GUI 形式，甚至有一个就是用 GitHub 开发的，这样可以更好地帮助开发者了解 Git，避免命令行晦涩难懂。如果你不想安装 Git，GitHub 也允许你以压缩格式下载源文件。

用户可以在 https://github.com/signup/free 上免费注册 GitHub 账户。Git 安装好之后，在终端程序中输入命令行$git clone git@github.com: dfsw/icf.git 就可以将一份源代码的拷贝下载到当前工作目录中。这一版书中用到的示例程序都在 Mastering 文件夹下。欢迎大家对示例代码提出修改建议。

联系作者

如果关于本书有任何意见或问题，可以通过 mastering.ios.frameworks@gmail.com 邮箱与我们联系，或者在 Twitter 上@kylerichter 和@jwkeeley。

致谢

要感谢所有为了本书默默在幕后付出的人，如果没有他们就没有这本书，虽然封面上的作者只有两个人的名字，不过这本书是通过大家的帮助才顺利完成的。首先要感谢的是 Trina MacDonald，如果不是她的领导才能和对我们的不断督促，我们可能永远无法完成本书的撰写。Pearson 的编辑们也给了我们极大的帮助，正是他们的不断努力，从一点小的错误到技术问题都逐页进行勘察，使本书变得更加完美。NiklasSaers、Olivia Basegio、Justin Williams、Sheri Replin、Elaine Wiley、Cheri Clark、ChutiPrasertsith 和 Gloria Shurick 也为本书的完成做出了贡献。

还要感谢 Langille Design (http://jordanlangille.com)工作室的 Jordan Langille，他为我们提供了第 3 章和第 4 章 Whack-a-Cac 游戏的设计方案。正是他的帮助让 Game Center 示例项目变得更加引人入胜。

不止我们自己为了撰写本书付出了大量的时间，家人和同事同样为此付出了许多时间。我们要感谢身边所有人的付出，让我们能有时间专注于本书的编写。

最后，还要非常感谢开发社区的朋友们。我们经常在开发者论坛上和博客里咨询大家问题，谢谢大家的提问和无私的反馈。如果不是 iOS 开发社区中大量参与者的努力，本书很可能就无法完成了。

目　录

第1章

UIKit Dynamics

苹果公司在 iOS 7 版本中引入了 UIKit Dynamics 框架，使用该框架，开发者可以很容易地将真实的物理模拟动作应用在 UIView 上。在之前的版本中，开发者只能将这种真实的动作效果整合到部分程序中，比如可滑动的单元格和下拉刷新动画等。苹果公司在 iOS 7 和 iOS 8 版本中向前迈进了一大步，将这些动画加入到核心 OS 库中，同时也在极力鼓励开发者使用它们实现动画效果。

UIDynamicItem 协议和支持该协议的动态元素使用户体验得到了极大提升。在程序中添加重力、碰撞、弹跳、瞬间位移等效果变得异常简单。介绍这些动态元素的 API 很简单，且很容易实现，应用这些功能提升用户体验易如反掌。

1.1 示例程序

示例程序(如图 1-1 所示)是一个基本的表格元素，用来展示 UIKit Dynamics 各种不同的功能。在这个程序中一共展示了从重力感应到属性设置等 7 个效果，每个效果都会在后面小节中详细展开介绍。除了表视图和基本的导航视图，示例程序并不包含任何专门针对 UIKit Dynamics 的功能。

注意

在同一个视图中使用 UIKit Dynamics 和自动布局可能会导致一些布局问题。通常，这是由于自动布局与 UIKit Dynamics 争抢视图上的正确位置，导致视图出现无法预料的错位而致。如果视图没有像预想那样呈现，开发者可以检查自动布局的相关设置来查看是否出现了冲突。

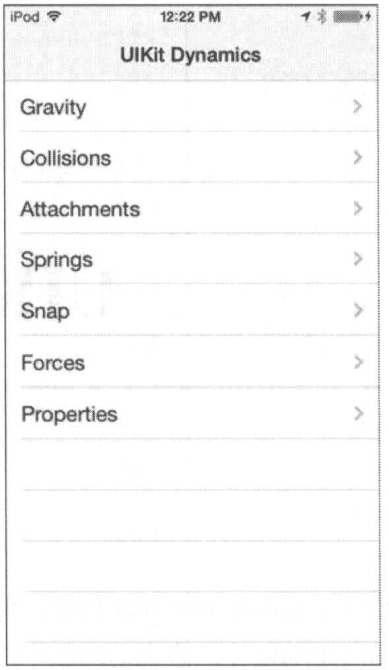


图 1-1 简单查看一下用于展示 UIKit Dynamics 各项功能的示例程序

1.2 UIKit Dynamics 介绍

UIKit Dynamics 是一组新的类和方法，最初是在 iOS 7 版本的 iDevices 中引入的。简单来说，UIKit Dynamics 通过在 UIView 视图中整合现实中的一些行为，提供了一种易于实现的方法来提升应用的用户体验。用最简短的术语来解释 UIKit Dynamics，其实它就是 UIKit 的基础物理引擎，不过它并不像传统的物理引擎一样是专为游戏开发而设计的。苹果公司提供了一些游戏框架，其中都包含了物理引擎，比如 SpriteKit。

当程序创建一个新的 UIDynamicAnimator 并将其添加到 UIView 中时，动态行为就会被激活。每个动画元素都可以对其属性和行为进行自定义，比如重力、碰撞检测、密度、摩擦力以及下面小节中将介绍的额外一些属性。

一共有 6 个附加类可以支持 UIDynamicAnimator 元素的自定义设置，分别是 UIAttachmentBehavior、UICollisionBehavior、UIDynamicItemBehavior、UIGravityBehavior、UIPushBehavior 和 UISnapBehavior。每个元素都允许指定自定义属性并且会在相应的视图中以真实的行为和动画反映出来。

1.3 UIKit Dynamics 具体实现

创建一个新的动画并将它添加到一个视图中，只需要两行代码就可以实现上述操作。示例中 self.view 对象即为将要使用 UIKit Dynamics 行为的对象。每一个特定的动态元素必须使用 addBehavior:方法添加到动画对象中。

```
UIDynamicAnimator *animator = [[UIDynamicAnimatoralloc]
initWithReferenceView:self.view];
```

```
[animatoraddBehavior:aDynamicBehavior];
```

每一个 UIDynamicAnimator 都是独立的，多个动画对象可以同时运行。对于一个持续运行的动画对象，对其的引用必须有效。当动画对象上的所有元素都处于静止状态时，动画对象此时不执行任何计算且处于暂停状态，不过实际操作中建议将不再使用的动画对象移除。

【游戏开发者的经验】

物理模拟对于游戏开发者而言已经使用了很多年了，很多难学的课程都已经学过了。现在物理层的处理技术已经蔓延到普通应用的开发中，下面介绍一些每位开发者都可以从中获益的基本原则。

当向游戏或应用添加物理特性时，请采取小步推进的方式。在多个互动代码段中试图找到出现的错误几乎是不可能的，采用越小步骤得到最终结果，程序也就越容易优化和调试。

在物理层进行处理时，有一些限制和边界在计算机模拟中无法体现。在 1997 年发布的经典游戏“死亡赛车”(Carmageddon)中，物理层处理是基于无上限帧率的。当计算机的处理速度变得越来越快后，帧率得到了大幅提升，在特定的公式中通过创建变量可以得到意想不到的结果。当把任何一种计算类型运用到物理引擎中时，需要确保其最大值和最小值都是符合要求且经过测试的。

预见下面这种意外情况：处理碰撞事件时，当 30 个对象发生重叠后，结果就会变得很扭曲。UIKit Dynamics 可以很好地确保开发者不会出现类似对象超过边界等情况，在处理上述碰撞场景时也能很完美地加以解决。不过当处理许多对象的复杂操作时也不能完全保证不出现边界情况和 bug。随着使用物理引擎的增加，越来越需要进行测试和调试，要能够预料到那些不期而遇和非常规情况下应该遵循的物理定律。

1.3.1　重力效果

重力效果被认为是最容易实现的 UIDynamicItem，同时也是实践中最常用的。苹果公司在 iOS 8 中重点强调了对重力相关元素的使用，用户操作重力相关的互动操作不需要再经过锁屏界面了。在 iOS 8 锁屏界面中，使用 UIGravityBehavior 函数可以实现向上拖动照相机图标并在到达中点之前松开手指，使屏幕再次返回锁屏状态的功能。在之前的 iOS 7 版本中引入 UIKitDynamics 时，这一功能还只能通过手工撰写计时器和传统动画相结合的方式才能实现。

下面的代码片段将在 frogImageView 视图中设置重力效果，该视图是 self.view 的子视图。首先为需要呈现动画的封闭视图创建一个新的 UIDynamicAnimator 对象，本例中呈现动画的视图即 self.view。创建一个新的 UIGravityBehavior 对象并初始化，初始化数组为使用重力效果的视图集合。设置本例中的重力行为参数：y 轴向下力度为 0.1。行为参数设置完成后，使用 addBehavior:方法将其添加到 UIDynamicAnimator 对象。

```
animator = [[UIDynamicAnimatoralloc] initWithReferenceView:self.view];

UIGravityBehavior* gravityBehavior = [[UIGravityBehavioralloc]
➥initWithItems:@[frogImageView]];
```

```
[gravityBeahviorsetXComponent:0.0f yComponent:0.1f];
[animatoraddBehavior:gravityBehavior];
```

注意

动态元素必须作为引用视图的子视图；如果动态元素不是子视图，动画对象就不会移动。

UIKit Dynamics 使用自身的物理系统，苹果工程师将其戏称为 UIKit Newtons。虽然它同标准的公式没有直接的关联，苹果公司还是做到了非常近似的效果。力度 1.0 大致等于 9.80655 m/s^2，即地球的重力。要使用地球重力十分之一的力，也就是 0.1 的力度。在 UIKit Dynamics 框架中使用重力效果不需要特别指定方向，默认就是向下的重力。如果 yComponent 参数为负值，重力方向才是向上的。同样，重力可以指定为沿 X 轴。元素还有密度属性，我们会在后面的 1.3.7 节中详细进行介绍。

运行重力效果示例代码，结果是 imageView 视图以大约十分之一地球重力的速度滑落(如图 1-2 所示)，并且完全滑出了屏幕。这是因为我们没有设置边界或碰撞事件，对象并不知道要碰到哪个边界才能停止运动，所以就一直滑落下去。

图 1-2 带有重力效果的图片视图在重力效果示例中向下滑到屏幕底端

1.3.2 碰撞效果

上一节中介绍了重力，不过应用了重力的对象会从屏幕底端滑出并一直无限地滑下去。这是因为我们没有在对象下降过程中设置碰撞点来阻止它。

我们对上一个示例程序进行修改，对封闭视图添加碰撞边界，同时添加第二个图片对象。碰撞示例的开始部分同重力示例一样，不过这里使用了两个图片视图。

创建 UICollisionBehavior 对象的过程同创建 UIGravityBehavior 对象的过程类似。对象通过将要应用效果的一些 UIView 视图进行初始化，本例中即两个 UIImageViews。除了视图之外，碰撞行为还需要在以下三个值中指定一个值作为参数：使用 UICollisionBehaviorModeItems 参数会使元素相互碰撞；使用 UICollisionBehaviorModeBoundaries 参数，元素虽然不会发生碰撞，但是会和边界发生碰撞；使用 UICollisionBehaviorModeEverything 参数会使元素在相互之间和同边界都发生碰撞。

要想程序中的对象和边界发生互动关系，需要先对边界进行定义。最简单的定义方法就是通过在 UICollisionBehavior 对象上设置一个称为 translatesReferenceBoundsIntoBoundary 的布尔值参数，在示例中我们将其用于 self.view 对象。边界也可以被设置为遵循某种路径，使用 addBoundaryWithIdentifier:forPath:方法使其遵循 NSBezierPath 表示的路径，或者使用 addBoundaryWithIdentifier:fromPoint:toPoint:方法使其遵循基于两个点表示的路径。

```
animator = [[UIDynamicAnimatoralloc] initWithReferenceView:self.view];

UIGravityBehavior* gravityBehavior = [[UIGravityBehavioralloc]
➥initWithItems:@[frogImageView, dragonImageView]];

[gravityBehaviorsetXComponent:0.0f yComponent:1.0f];


UICollisionBehavior* collisionBehavior = [[UICollisionBehavioralloc]
➥initWithItems:@[frogImageView, dragonImageView]];

[collisionBehaviorsetCollisionMode:UICollisionBehaviorModeBoundaries];

collisionBehavior.translatesReferenceBoundsIntoBoundary = YES;

[animatoraddBehavior:gravityBehavior];
[animatoraddBehavior:collisionBehavior];
```

UICollisionBehavior 还提供了一个代理回调用于遵照 UICollisionBehaviorDelegate 协议。

```
collisionBehavior.collisionDelegate = self;
```

UICollisionBehaviorDelegate 函数有 4 个回调方法，两个用于碰撞开始，另两个用于碰撞结束。每一个回调集都有一个方法用来定义边界是否出现碰撞。所有方法都提供了对导致回调方法触发的对象的引用。检测碰撞开始的方法还提供了一个 CGPoint 对象，用于记录碰撞发生的确切位置。示例代码会在对象检测到碰撞时更新标签的显示。

```
-(void)collisionBehavior:(UICollisionBehavior *)behavior
➥beganContactForItem:(id<UIDynamicItem>)item
➥withBoundaryIdentifier:(id<NSCopying>)identifier atPoint:(CGPoint)p
{
    if([item isEqual:frogImageView])
        collisionOneLabel.text = @"Frog Collided";
    if([item isEqual:dragonImageView])
```

```
        collisionTwoLabel.text = @"Dragon Collided";
}

-(void)collisionBehavior:(UICollisionBehavior *)behavior
➥endedContactForItem:(id<UIDynamicItem>)item
➥withBoundaryIdentifier:(id<NSCopying>)identifier
{

    NSLog(@"Collision did end");
}
```

1.3.3 附着效果

附着效果定义了两个对象间的动态连接，可实现两个移动对象间的绑定关系。默认情况下，UIAttachmentBehaviors 都是固定在对象中心的，附着点可以通过程序自行设置为对象上的任何一点。

本节示例程序的创建以上一节“碰撞效果”中的程序为基础。仍然使用两个图片视图，边界碰撞已经创建好，并且已应用到 UIDynamicAnimator 对象上。创建一个新的 CGPoint 并设置青蛙图片视图的中心点作为其关联点。创建一个新的 UIAttachmentBehavior 对象并使用 initWithItem:attachedToAnchor:对其进行初始化。这里仍然需要对 UICollisionBehavior 进行额外的初始化，从而指定具体的点和其他对象的规范。将碰撞效果和附着效果都添加到动画对象。

```
animator = [[UIDynamicAnimatoralloc] initWithReferenceView:self.view];

UICollisionBehavior* collisionBehavior = [[UICollisionBehavioralloc]
➥initWithItems:@[dragonImageView, frogImageView]];

[collisionBehaviorsetCollisionMode:UICollisionBehaviorModeBoundaries];

collisionBehavior.translatesReferenceBoundsIntoBoundary = YES;

CGPointfrogCenter = CGPointMake(frogImageView.center.x,
➥frogImageView.center.y);

self.attachmentBehavior = [[UIAttachmentBehavioralloc]
➥initWithItem:dragonImageViewattachedToAnchor:frogCenter];

[animatoraddBehavior:collisionBehavior];
[animatoraddBehavior:self.attachmentBehavior];
```

这些对象现在已经被一个长度为其初始距离的可见连接线围住，如果青蛙图片视图移动，那么龙图片视图将会保持中心点不变而随之移动。不过现在青蛙图片还不具有移动的能力，为解决这个问题，示例程序需要实现一个简单的平移手势。当青蛙图片视图在主视图中移动时，中心点位置更新的同时会设置更新的锚点。

```
-(IBAction)handleAttachmentGesture:(UIPanGestureRecognizer*)gesture
{
    CGPointgesturePoint = [gesture locationInView:self.view];

    frogImageView.center = gesturePoint;
    [self.attachmentBehaviorsetAnchorPoint:gesturePoint];
}
```

在移动过程中，碰撞边界始终有效并且覆盖了附着效果，这一点可以通过将龙图片移到视图边界上进行验证。

为了改变两个对象间的附着距离，还可以对附着视图的长度属性进行更新。附着点自身不需要是对象的中心点，可以通过调用 setAnchorPoint 方法设置任意偏移量作为附着点。

1.3.4　弹跳效果

弹跳效果(如图 1-3 所示)是上述附着效果的扩展。UIKit Dynamics 框架可以在 UIAttachmentBehavior 上额外设置一些属性，比如频率和阻尼等。

图 1-3　将弹跳效果应用在龙图片和青蛙图片上，展示了重力效果和设置 UIAttachmentBehavior 阻尼、频率的效果

下面小节的示例程序在创建 UIAttachmentBehavior 对象后又对其添加了三个新的 UIKit Dynamic 属性。第一个是 setFrequency，用于设置对象的振幅或摆动大小；第二个是 setDamping，用于校平动画峰值；第三个是 setLength，该属性也是根据其初始位置进行调整的。为更好地展示上述行为效果，我们仍然在例子中添加了重力效果。

```
animator = [[UIDynamicAnimatoralloc] initWithReferenceView:self.view];
```

```
UICollisionBehavior* collisionBehavior = [[UICollisionBehavioralloc]
➥initWithItems:@[dragonImageView, frogImageView]];

UIGravityBehavior* gravityBeahvior = [[UIGravityBehavioralloc]
➥initWithItems:@[dragonImageView]];

CGPointfrogCenter = CGPointMake(frogImageView.center.x,
➥frogImageView.center.y );

self.attachmentBehavior = [[UIAttachmentBehavioralloc]
➥initWithItem:dragonImageViewattachedToAnchor:frogCenter];

[self.attachmentBehaviorsetFrequency:1.0f];
[self.attachmentBehaviorsetDamping:0.1f];
[self.attachmentBehaviorsetLength:100.0f];

[collisionBehaviorsetCollisionMode: UICollisionBehaviorModeBoundaries];

collisionBehavior.translatesReferenceBoundsIntoBoundary = YES;

[animatoraddBehavior:gravityBeahvior];
[animatoraddBehavior:collisionBehavior];
[animatoraddBehavior:self.attachmentBehavior];
```

现在，在屏幕中移动青蛙图片会使龙图片由底部上行 100 像素并按照设置好的附着效果和重力效果进行摇摆。

1.3.5 瞬间位移

元素可以在视图中动态地瞬间位移至另一个点，该功能非常容易实现。在示例程序中，动作已经绑定了一个点击手势，点击屏幕中的任何位置都会使指定图片瞬间位移到指定的锚点。每个 UISnapBehavior 一次仅能关联一个单独的元素，并且在初始化过程中元素终点的位置已被设置好。另一个属性阻尼系数也可以被指定，用于影响动作的移动速度。

```
CGPoint point = [gesture locationInView:self.view];
animator = [[UIDynamicAnimatoralloc] initWithReferenceView:self.view];

UISnapBehavior* snapBehavior = [[UISnapBehavioralloc]
➥initWithItem:frogImageViewsnapToPoint:point];

snapBehavior.damping = 0.75f;
[animatoraddBehavior:snapBehavior];
```

1.3.6 推力效果

UIKit Dynamics 还支持对力度的运用，比如推力。UIPushBehavior 的使用比之前的几种动作效果稍微复杂一些，不过同其他物理引擎相比还是很容易使用的。示例仍然使用了一个前面测试程序中用到的 UICollisionBehavior 对象，这确保了在推力效果作用时图片视图始终

处于屏幕范围内。

创建一个新的 UIPushBehavior 并使用一个图片视图初始化它。目前方向和加速度两个属性的值为 0.0。

示例程序还在屏幕中心用一个小黑方块的形式描绘了一个引用。

```
animator = [[UIDynamicAnimatoralloc] initWithReferenceView:self.view];

UICollisionBehavior* collisionBehavior = [[UICollisionBehavioralloc]
➥initWithItems:@[dragonImageView]];

collisionBehavior.translatesReferenceBoundsIntoBoundary = YES;
[animatoraddBehavior:collisionBehavior];

UIPushBehavior *pushBehavior = [[UIPushBehavioralloc]
➥initWithItems:@[dragonImageView]
➥mode:UIPushBehaviorModeInstantaneous];

pushBehavior.angle = 0.0;
pushBehavior.magnitude = 0.0;

self.pushBehavior = pushBehavior;
[animatoraddBehavior:self.pushBehavior];
```

如果现在运行项目，图片视图将会始终固定在屏幕上，因为还没有对推力效果设置任何参数。创建一个新的平移手势，在其关联动作中会计算出 magnitude 与 angle 的新值并应用其中。在该例中，我们计算出一个角度，用于确定推力的来源。角度是基于中间参考点的，运动距离还要根据持续增加的力度进行计算。点击黑色方块的外面区域将会有一个力沿相应的方向作用于图片视图。图片离得越远，力度就越大。

```
CGPoint point = [gesture locationInView:self.view];

CGPoint origin = CGPointMake(CGRectGetMidX(self.view.bounds),
➥CGRectGetMidY(self.view.bounds));

CGFloat distance = sqrtf(powf(point.x-origin.x, 2.0)+powf(point.y-
➥origin.y, 2.0));

CGFloat angle = atan2(point.y-origin.y, point.x-origin.x);
distance = MIN(distance, 100.0f);

[self.pushBehaviorsetMagnitude:distance/100.0];
[self.pushBehaviorsetAngle:angle];

[self.pushBehaviorsetActive:YES];
```

除了可以手动设置角度和加速度的值之外，还可以对指定的目标点使用 setTargetPoint:forItem:方法自动进行计算。有必要对视图中的一部分施加力的效果，受力点并不一定是对象中心点，使用 setXComponent:yComponent:方法可以指定一个 CGPoint 类型的点作为受力点。

有两种类型的推力可以应用——UIPushBehaviorModeContinuous 和 UIPushBehaviorMode-Instantaneous，持续的力会推动对象不断加速，瞬间的力则会直接作用到对象上。

1.3.7 元素属性

动态元素有许多默认的预设属性，这些属性都可以自定义设置，用来表示对象针对物理引擎的不同响应。示例程序(如图 1-4 所示)展示了对一个图片视图修改默认属性和对另一个图片视图保留默认属性的对比效果。

图 1-4 修改动态元素的默认属性，展示在同一力的作用下得到的不同物理响应

要修改对象的属性，首先需要创建一个新的 UIDynamicItemBehavior 并使用相应的视图对象初始化它。结果就是其中一个对象动起来像一个橡皮球，对其应用了重力和碰撞效果后更容易弹跳。具体的属性和描述参见表 1-1。

```
animator = [[UIDynamicAnimatoralloc] initWithReferenceView:self.view];

UIGravityBehavior* gravityBeahvior = [[UIGravityBehavioralloc]
➥initWithItems:@[dragonImageView, frogImageView]];

UICollisionBehavior* collisionBehavior = [[UICollisionBehavioralloc]
➥initWithItems:@[dragonImageView, frogImageView]];

collisionBehavior.translatesReferenceBoundsIntoBoundary = YES;

UIDynamicItemBehavior* propertiesBehavior = [[UIDynamicItemBehavior
➥alloc] initWithItems:@[frogImageView]];

propertiesBehavior.elasticity = 1.0f;
```

```
propertiesBehavior.allowsRotation = NO;
propertiesBehavior.angularResistance = 0.0f;
propertiesBehavior.density = 3.0f;
propertiesBehavior.friction = 0.5f;
propertiesBehavior.resistance = 0.5f;

[animatoraddBehavior:propertiesBehavior];
[animatoraddBehavior:gravityBehavior];
[animatoraddBehavior:collisionBehavior];
```

表 1-1　UIDynamicItem 属性及其描述

属　性	描　述
allowsRotation	一个布尔值，用于指定元素是否根据力度进行旋转，默认值是 YES
angularResistance	一个位于 0.0 到 CGFLOAT_MAX 范围内的 CGFloat 值，用于指示带有角度的阻尼值，这个数值越大，旋转的减速就越快
density	用于表示密度。100×100 对象的默认密度值为 1.0，100×200 对象的默认密度值为 2.0。调整密度的大小会影响重力和碰撞效果的反映
elasticity	有效值介于 0.0 到 1.0，用于指示当对象间发生碰撞时弹力的大小。0.0 表示不发生弹跳，1.0 表示整个力度会应用到碰撞对象上
friction	当两个元素相互滑动时的线性阻力，0.0 表示无摩擦力，1.0 表示最大摩擦力。此外，也可以使用大于 1.0 的值表示额外的阻力
resistance	开放空间中遇到的线性阻力，取值范围为 0.0 到 CGFLOAT_MAX。0.0 表示无阻力，1.0 表示当没有力作用于元素时元素应该停止移动

1.4　深入了解 UIDynamicAnimator 和 UIDynamicAnimatorDelegate

本章前面部分介绍了 UIDynamicAnimator，并且示例程序中都用到了 addBehavior 方法，不过这个类的强大功能远不止这些。除了可以添加动态效果之外，还可以每次移除一个效果或者对一组对象使用 removeBehavior:和 removeAllBehaviors 方法。要得到当前 UIDynamicAnimator 对象所有的动作行为，可以通过行为属性返回的数组进行查看。

还可以通过运行属性来查看动画的运行状态，可以使用 elapsedTime 值确定动画时长。UIDynamicAnimator 还带有一个关联的委托函数，即 UIDynamicAnimatorDelegate。该委托函数给出了两个方法用于处理暂停和重启动作。开发者无法显式暂停 UIDynamicAnimator 对象的运行，当所有元素都处于静止且不再发生运动时动画效果会自动停止。当将任何新的动作效果应用在元素上时，都会使其开始运动并返回激活状态。

```
-(void)dynamicAnimatorDidPause:(UIDynamicAnimator *)animator
{
    NSLog(@"Animator did pause");
}
```

```
-(void)dynamicAnimatorWillResume:(UIDynamicAnimator *)animator
{
    NSLog(@"Animator will resume");
}
```

1.5 小结

无论是从开发者的立场还是从 iOS 系统本身未来的发展来看，UIKit Dynamics 都是一个令人感兴趣的话题。苹果公司在花大力气将软件推广到现实世界，希望能够做到让用户同应用的交互就像同真实世界交互的体验一样。用户期望应用对人类指令的反应同周围现实世界的事务一样。对于苹果公司来说这并不新鲜，传统 iPhone 最大的一个卖点就是动量卷动(momentum scrolling)，现在它们又为开发者提供了工具，以实现向程序中添加这些功能。

本章介绍了 UIKit Dynamics 的基本概念和基本组件，不过这些方法真正的强大之处还在于开发者的运用。这个框架有无限的潜能和各种组合，开发者利用这些功能做出的产品可能令苹果公司自己都感到惊讶。用户体验已经被重新定义，那么可以确定的是软件提供真实的物理响应已经不再是可有可无的，用户也非常期待这样的技术。

第2章

Core Location、MapKit 和 Geofencing 框架

地图和位置信息是 iOS 系统最有用的一些功能，通过这些功能可以让用户根据周围相关的地理环境找到前往目的地的路径。如今的应用在位置信息方面已经满足了用户的许多特定需求，比如查找道路和方向，使用指定的交通服务以及对同一个位置使用多种不同有趣的元素进行标注。随着苹果公司引入新的地图引擎，添加更多新的强大功能，开发者可以利用这些功能使他们的应用更上一个台阶。

iOS 用于支持位置和地图服务的框架有两个，其中一个是 Core Location，用于帮助设备确定当前的位置和运动朝向以及基于位置的其他信息。另一个是 MapKit，为用户提供位置感知相关的界面，它包含的 Apple Maps 以 2D 和 3D 方式提供地图视图、卫星视图和混合视图。MapKit 还支持以大头针图标显示的地图标注功能，以及支持对地图上的重点位置、路径或其他特征进行编辑。

2.1 示例程序

本章的示例程序 FavoritePlaces 支持用户使用地图，并在设备当前位置收藏他们喜欢的地点并查看。用户可以使用 Core Location 框架访问地理信息，查询经纬度和具体地址。此外，用户还可以在感兴趣的位置设置一个指定半径的圆形区域，当进入该区域时就会有提醒。应用还提供一个特殊的位置(由绿色箭头表示)，可以在地图上拖曳该箭头来准确定位到需要的下一个目的地。当箭头指向一个位置松开手指后，程序会自动解析地理编码将该位置的名称和地址显示出来。

2.2 获取用户位置

使用 Core Location 框架获取用户信息需要几个步骤。应用必须从用户那里获准访问当前

位置信息的权限。此外，在设备试图获取位置前应用需要确保位置服务是开启的。当上面两条都满足后，应用就可以发起位置请求，并对 Core Location 返回的结果进行解析和使用。本节详细介绍这些步骤的具体实现方法。

2.2.1 请求和允许

要在应用中使用 Core Location 框架，需要导入 CoreLocation，如下所示：

```
@import CoreLocation;
```

要在应用中使用 MapKit 框架，需要在每个类中都导入 MapKit，如下所示：

```
@import MapKit;
```

Core Location 涉及用户隐私，所以需要询问用户是否允许设备访问当前的位置信息。Location Services(位置服务)在手机的 Settings 程序的 Privacy 选项中，用户可以手动开启或关闭，也可以分别设置为 Always、While Using 或 Never，如图 2-1 所示。

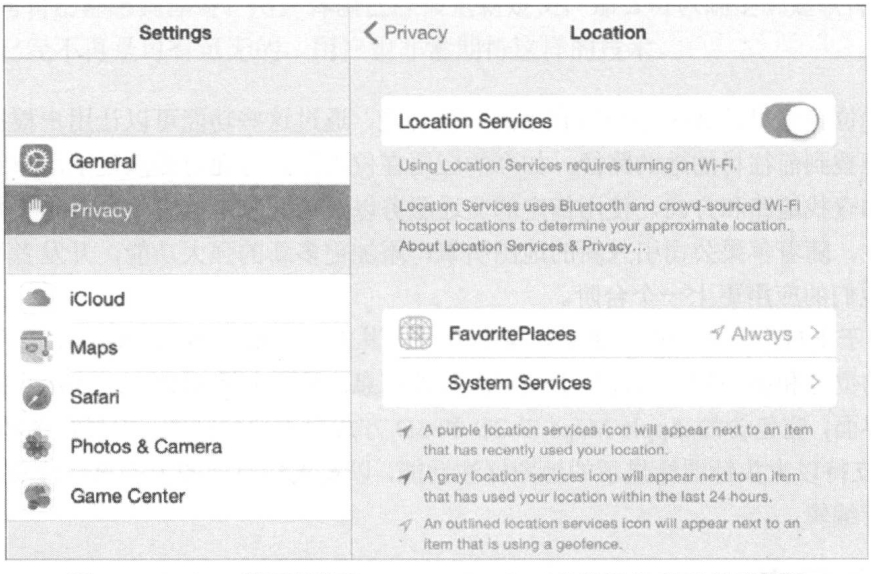


图 2-1 Settings 程序菜单和 Location Services(位置服务)隐私设置菜单

Always 表示即使在程序未激活时用户也允许相应的程序访问位置信息。While Using 表示只有当程序在前台处于激活状态时，才允许访问设备的位置信息。Never 表示不允许程序访问设备的位置信息。要获得使用位置信息的许可，应用需要向 CLLocationManager 请求正确的授权类型(While Using 或 Always)。对于示例程序，应用激活时和设置位置围栏时都要用到位置信息，所以请求的授权类型选择 Always：

```
[appLocationManager.locationManager requestAlwaysAuthorization];
```

如果 Location Services 处于关闭状态，Core Location 将弹出一个对话框，询问用户是否进行设置，如图 2-2 所示。

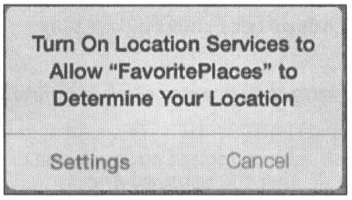


图 2-2　FavoritePlaces 示例程序的位置服务提醒框

如果位置管理器没有在使用设备位置前请求授权，程序将会弹出一个提醒框，询问用户是否允许访问位置信息，如图 2-3 所示。需要注意的是，开发者需要在.plist 文件中添加一个条目，用于说明程序将会使用用户位置信息，具体会用到关键字 NSLocationWhenInUseUsageDescription 或 NSLocationAlwaysUsageDescription。如果没有找到授权类型关键字，就不会调用该请求所允许的对话框。

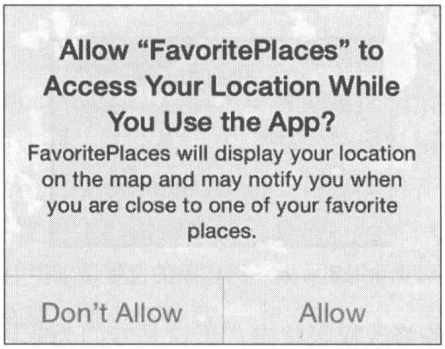


图 2-3　FavoritePlaces 示例程序的位置请求提醒框

如果用户点击 Allow，则授权成功，位置管理器将可以正常获取当前位置信息。如果用户点击 Don't Allow，访问当前位置的请求就会被拒绝，认证状态 CLLocationManager 的代理方法将会在 ICFLocationManager 中被调用。

```
-(void)locationManager:(CLLocationManager *)manager
➥didChangeAuthorizationStatus:(CLAuthorizationStatus)status
{
    if(status == kCLAuthorizationStatusDenied)
    {
        [self.locationManager stopUpdatingLocation];

        NSString *errorMessage =
        ➥@"Location Services Permission Denied for this app.";

        NSDictionary *errorInfo =
        ➥@{NSLocalizedDescriptionKey:errorMessage};

        NSError *deniedError =
        ➥[NSError errorWithDomain:@"ICFLocationErrorDomain"
                              code:1
                          userInfo:errorInfo];

        [self setLocationError:deniedError];
```

```
        [self getLocationWithCompletionBlock:nil];
    }
    if(status == kCLAuthorizationStatusAuthorizedWhenInUse)
    {
        [self.locationManager startUpdatingLocation];
        [self setLocationError:nil];
    }
}
```

示例程序的 ICFLocationManager 类在程序的其他位置使用 completion 代码块来请求对当前位置的访问，这样做可以简单地处理多条请求的情况。当位置服务可用或出现错误时，getLocationWithCompletionBlock:方法将会处理所有的 completion 代码块，这样源代码段就可以更加合理地调用位置服务或处理错误。在本例中，调用端将显示一个提醒框，显示位置拒绝权限的错误，如图 2-4 所示。

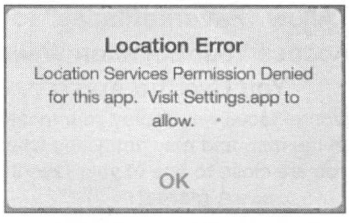


图 2-4 FavoritePlaces 示例程序的位置请求拒绝提醒框

如果用户日后为位置服务修改权限状态，无论是专门为应用设置还是对整个设备进行设置(参见图 2-1)，都会调用同一个委托方法并返回相应的值。在 ICFLocationManager 中，如果权限被拒绝，委托方法就会弹出一个错误提醒框；如果权限被允许，则会重新更新当前位置并清除上一个错误对话框。

2.2.2 检查服务

要直接确认设备当前是否开启了位置服务，可以使用 CLLocationManager 类中名为 locationServicesEnabled 的方法。

```
if([CLLocationManager locationServicesEnabled])
{
    ICFLocationManager *appLocationManager =
    ➥[ICFLocationManager sharedLocationManager];

    [appLocationManager.locationManager startUpdatingLocation];
}
else
{
    NSLog(@"Location Services disabled.");
}
```

上述代码可以根据是否成功获得当前位置为应用提供自定义的处理方法。当用户没有权限访问当前位置时，应用一定要处理好有关位置服务的请求，并给予用户清楚的引导，使其能够顺利开启当前位置的服务功能。

2.2.3　开始位置请求

位置服务授权完成后，就可以使用 CLLocationManager 实例获得当前位置。在示例程序中，ICFLocationManager 提供了一个核心类，用于管理位置相关功能，它为应用管理这个 CLLocationManager 实例。在 ICFLocationManager 的 init 方法中，为需要的位置搜索功能创建了一个 CLLocationManager 对象并进行了定制。

```
[self setLocationManager:[[CLLocationManager alloc] init]];

[self.locationManager
➥setDesiredAccuracy:kCLLocationAccuracyBest];

[self.locationManager setDistanceFilter:100.0f];
[self.locationManager setDelegate:self];
```

CLLocationManager 对象有多个参数，用于对当前位置进行管理。通过指定所需的精确参数，应用可以告诉 CLLocationManager 对象在目前电量使用情况下是否需要完成非常精确的定位，或者为了节省电量而选择较低精确度的定位，使用低精确度也会减少获取位置操作的时间。设置距离过滤器可以让 CLLocationManager 对象知道在新位置事件生成前需要遍历多远的距离，这适用于基于位置变化的微调功能。最后，为 CLLocationManager 设置委托，这样对于位置事件和权限情况的改变等，功能代码就可以写在该委托函数中。当应用准备获取位置时，它会请求位置管理器对象更新位置。

```
ICFLocationManager *appLocationManager =
➥[ICFLocationManager sharedLocationManager];

[appLocationManager.locationManager startUpdatingLocation];
```

根据参数设置的情况，CLLocationManager 将选择 GPS 或 Wi-Fi，或者同时使用这两者来确定位置。当位置管理器更新当前位置或无法更新当前位置时，需要实现两个委托方法。当成功获得位置信息时，会调用 locationManager:didUpdateLocations:方法。

```
-(void)locationManager:(CLLocationManager *)manager
    ➥didUpdateLocations:(NSArray *)locations
{
    //Filter out inaccurate points
    CLLocation *lastLocation = [locations lastObject];
    if(lastLocation.horizontalAccuracy < 0)
    {
        return;
    }

    [self setLocation:lastLocation];
    [self setHasLocation:YES];
    [self setLocationError:nil];
```

```
    [self getLocationWithCompletionBlock:nil];
}
```

位置管理器可以传递位置数组包含的多个位置信息，数组中的最后一个对象就是最近刷新的位置。位置管理器还可以在实际获取位置前快速返回最后一次 GPS 的位置信息，不过这样的话，如果 GPS 关闭且设备此时发生位移，就会出现位置偏差。locationManager:didUpdate-Locations:方法会检查位置的准确性并忽略无效的值。如果找到准确位置，就会将其保存并执行相应的 completion 代码块。值得注意的是，位置管理器会随着位置的刷新多次调用该方法，所以这里的任何逻辑处理都要注意到这一点。

```
-(void)locationManager:(CLLocationManager *)manager
        ➥didFailWithError:(NSError *)error
{
    [self.locationManager stopUpdatingLocation];
    [self setLocationError:error];
    [self getLocationWithCompletionBlock:nil];
}
```

如果位置管理器对象没有获得位置信息，将调用 locationManager:didFailWithError:方法。问题可能出在授权上，或者 GPS 和 Wi-Fi 不可用(比如飞行模式)。在示例程序的代码实现中如果出现错误，程序会通知位置管理器停止更新位置信息，并捕捉位置错误信息，执行 completion 代码块中的代码，这样请求当前位置的代码就会合理地处理错误事件。

位置管理器的委托方法可以监视过程变化，比如更新地图指示器来显示用户即将运动的方向。要接收过程或运动朝向信息，位置管理器需要开启监视功能。可以设置一个过滤器，使程序拒绝对微小的变化进行处理。

```
CLLocationDegrees degreesFilter = 2.0 ;
if([CLLocationManager headingAvailable])
{
    [self.locationManager setHeadingFilter:degreesFilter];
    [self.locationManager startUpdatingHeading];
}
```

运动朝向改变事件会传递到 locationManager:didUpdateHeading:委托方法。

```
-(void)locationManager:(CLLocationManager *)manager
      ➥didUpdateHeading:(CLHeading *)newHeading
{
    NSLog(@"New heading, magnetic:%f", newHeading.magneticHeading);

    NSLog(@"New heading, true:%f", newHeading.trueHeading);
    NSLog(@"Accuracy:%f", newHeading.headingAccuracy);
    NSLog(@"Timestamp:%@", newHeading.timestamp);
}
```

新的运动朝向提供了许多有用的信息，包括磁极和真实朝向，从正北方向按度的单位来表示。其提供一种准确的读法，即用数字表示磁场方向的度数。使用正值表示更精确的朝向，

使用负值表示朝向无效，或者存在磁场干扰无法读数。时间戳信息会记录读数发生的时间，以避免使用之前未更新的朝向信息。

2.2.4　解析和理解位置数据

当位置管理器返回位置信息时，其实就会生成一个 CLLocation 实例。CLLocation 实例包含许多位置相关的有用信息。首先就是经纬度信息，它由一个 CLLocationCoordinate2D 对象表示。

```
CLLocationCoordinate2D coord = lastLocation.coordinate;

NSLog(@"Location lat/long:%f,%f",coord.latitude, coord.longitude);
```

纬度用赤道向北或向南以数字表示的度数呈现，赤道的纬度为 0，北极的纬度为 90 度，南极为-90 度。经度则以本初子午线为中点向东或向西以数字表示的度数呈现，本初子午线是贯穿南北极的一条假想的线，穿过英国格林威治的皇家天文台。由本初子午线向西为负的经度值，最大到-180 度，向东为正值最大到 180 度。

关于坐标系还有一点需要补充的就是水平精度，精度由 CLLocationDistance 或单位米来表示。水平精度的意思就是实际位置距坐标指定的范围内多少米的距离。

```
CLLocationAccuracy horizontalAccuracy =
➥lastLocation.horizontalAccuracy ;

NSLog(@"Horizontal accuracy:%f meters",horizontalAccuracy);
```

如果设备具有 GPS 功能，那么位置信息还会包含当前位置的经度信息和垂直精度，它们都以米作为单位。如果设备不具有 GPS 功能，经度会返回 0，精度值返回-1。

```
CLLocationDistance altitude = lastLocation.altitude;
➥NSLog(@"Location altitude:%f meters",altitude);

CLLocationAccuracy verticalAccuracy = lastLocation.verticalAccuracy;

NSLog(@"Vertical accuracy:%f meters",verticalAccuracy);
```

位置信息还包含一个时间戳，用于指示位置管理器确定位置的时间，利用这个时间戳可以帮助程序知道目前系统内的位置是否是最新的，只需要比较两个位置信息的时间戳就可以得出结果。

```
NSDate *timestamp = lastLocation.timestamp;
NSLog(@"Timestamp:%@",timestamp);
```

最后，位置信息还会提供以米/秒表示的速度信息和以从正北计算表示的路径信息。

```
CLLocationSpeed speed = lastLocation.speed;
NSLog(@"Speed:%f meters per second",speed);

CLLocationDirection direction = lastLocation.course;
NSLog(@"Course:%f degrees from true north",direction);
```

2.2.5 重大变更通知

在应用获取位置信息之后，苹果公司强烈建议开发者停止位置更新请求以节省电池电量。如果应用不需要持续不断地为用户提供准确的位置信息，使用显著位置变化检测就是一种非常有效的办法，可以在设备发生较大位移时通知应用，这样做可以很好地维持 GPS 和 Wi-Fi 检测当前位置，又有效节省设备的用电量。

```
[self.locationManager startMonitoringSignificantLocationChanges];
```

通常情况下，当设备位移超过 500 米或者切换基站时会产生一条通知，上一条通知过去的 5 分钟内不会重复发送通知。位置更新事件会发送到 locationManager:didUpdateLocations:委托方法。

2.2.6 使用 GPX 文件测试指定位置

对基于位置的应用进行测试是一件很烦的事情，尤其是指定的位置非常不方便实地测试的情况。幸运的是，在 Xcode 中使用 GPX 文件所提供的工具可以很好地解决这一测试问题。一个 GPX 文件就是一个 GPS 交换格式文档，其使用 XML 语言在设备之间实现与 GPS 信息的交互。在调试模式下，Xcode 可使用定义在 GPX 文件中的“标记点”来为 iOS 模拟器或真实设备设置当前位置。

在示例程序中，当前位置由文件 DMNS.gpx 设置，设置为 Denver Museum of Nature and Science(丹佛自然科学博物馆)。

```
<?xml version="1.0"?>
<gpx version="1.1" creator="Xcode">

    <wpt lat="39.748039" lon="-104.94000">
       <name> Denver Museum of Nature and Science</name>
    </wpt>

</gpx>
```

要在 Xcode 中使用 GPX 文件进行调试，方法是首先找到项目窗口左上角的 Scheme 下拉菜单，选择其中的 Edit Scheme，在 Options 选项卡中选中 Allow Location Simulation 选项，如图 2-5 所示。选中此项后，就可以在 Default Location 旁边的下拉框中选中特定的位置了。下拉列表中包含了一些内置的位置和添加到项目中的每一个 GPX 文件。

应用在调试模式下运行时，Core Location 会返回 GPX 文件中指定的位置作为真机设备或模拟器当前的位置。调试过程中如果需要修改位置，可以从 Xcode 菜单中选择 Debug→Simulate Location，然后选择一个位置(如图 2-6 所示)。Core Location 会用新的选定位置替换之前的位置，同时调用 locationManager:didUpdateLocations:委托方法。

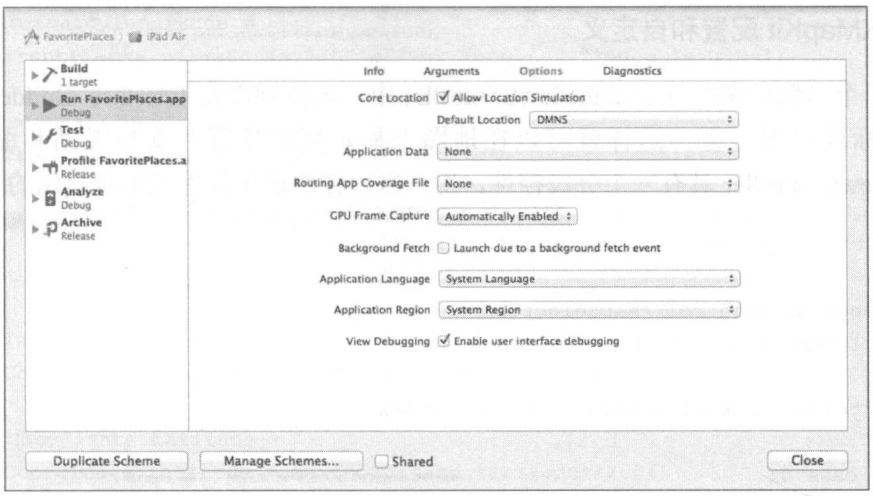


图 2-5　Xcode FavoritePlaces 配置信息

图 2-6　选择 Debug→Simulate Location

2.3　显示地图

MapKit 框架为 iOS 提供了用地图元素绘制用户界面的各种功能，其中最基础的类是 MKMapView，用于显示地图、处理用户和地图的交互操作并管理标注(annotation，类似大头针样的标注)和覆盖物(overlay，类似路线图或区域高亮显示等地图上的覆盖物)。为了更好地了解 iOS 系统中如何进行地图开发，很重要的一点是掌握坐标系的使用。

2.3.1　了解坐标系

MapKit 框架包含两类坐标系，分别是针对地图的坐标系和针对视图的坐标系。针对地图的坐标系使用的是 Mercator Projection，通过真实世界中的 3D 地图投身到 2D 坐标系，坐标用经度和维度表示。地图视图使用标准 UIKit 视图坐标来表示屏幕上所显示的部分地图，它是整个地图的一部分。地图视图然后根据地图坐标系来决定在该视图中显示哪些内容。

2.3.2 MKMapKit 配置和自定义

在同一个示例程序的 ICFMainViewController 中，地图视图在 Interface Builder 中被设置为默认的标准地图类型，这样就可以在地图上显示用户位置并支持滑动和缩放操作。ICFMainViewController 具有一个分段控件，用户可以使用这个控件调整所显示的地图类型。

```
-(IBAction)mapTypeSelectionChanged:(id)sender
{
    UISegmentedControl *mapSelection =
    ➥(UISegmentedControl *)sender;

    switch(mapSelection.selectedSegmentIndex)
    {
        case 0:
            [self.mapView setMapType:MKMapTypeStandard];
            break;
        case 1:
            [self.mapView setMapType:MKMapTypeSatellite];
            break;
        case 2:
            [self.mapView setMapType:MKMapTypeHybrid];
            break;

        default:
            break;
    }
}
```

除了设置地图类型，另一个常用自定义设置就是对地图显示区域的设置。在 ICFMainViewController 中，zoomMapToFitAnnotations 方法会检查当前聚焦的地点，并设置合适的尺寸和中心点。该方法首先会设置默认的最大值坐标和最小值坐标。

```
CLLocationCoordinate2D maxCoordinate =
➥CLLocationCoordinate2DMake(-90.0, -180.0);

CLLocationCoordinate2D minCoordinate =
➥CLLocationCoordinate2DMake(90.0, 180.0);
```

查看一下地图上已经存在的标注(在下一节“地图标注和覆盖物”中会有详细介绍)，zoomMapToFitAnnotations 方法为所有呈现的标注计算了最大和最小经纬度值。

```
NSArray *currentPlaces = [self.mapView annotations];

maxCoordinate.latitude =
➥[[currentPlaces valueForKeyPath:@"@max.latitude"] doubleValue];

minCoordinate.latitude =
➥[[currentPlaces valueForKeyPath:@"@min.latitude"] doubleValue];
```

```
maxCoordinate.longitude =
➥[[currentPlaces valueForKeyPath:@"@max.longitude"] doubleValue];

minCoordinate.longitude =
➥[[currentPlaces valueForKeyPath:@"@min.longitude"] doubleValue];
```

之后，zoomMapToFitAnnotations 方法根据最大和最小经纬度坐标值计算中心坐标。

```
CLLocationCoordinate2D centerCoordinate;

centerCoordinate.longitude =
➥(minCoordinate.longitude + maxCoordinate.longitude)/2.0;

centerCoordinate.latitude =
➥(minCoordinate.latitude + maxCoordinate.latitude)/2.0;
```

之后，该方法根据中心坐标计算显示所有坐标点所需的缩放范围。每个维度计算好的缩放范围值还需要乘以 1.2，用于在最远点和视图边缘之间创建一条边界。

```
MKCoordinateSpan span;

span.longitudeDelta =
➥ (maxCoordinate.longitude - minCoordinate.longitude) * 1.2;

span.latitudeDelta =
➥(maxCoordinate.latitude - minCoordinate.latitude) * 1.2;
```

在中心和缩放范围计算好之后，就可以创建地图区域并使用它对地图视图的显示区域进行设置。

```
MKCoordinateRegion newRegion =
➥MKCoordinateRegionMake(centerCoordinate, span);

[self.mapView setRegion:newRegion
               animated:YES];
```

设置 animated:的值为 YES，可以看到类似于用户在缩放地图的动画效果；如果设置该值为 NO，缩放的过程就不会有动画效果。

2.3.3　对用户操作的响应

可以通过指定 MKMapViewDelegate 对用户在地图上的操作进行响应。用户在地图视图上的常见操作包括平移和缩放、拖曳标注和点击 callout(标注内容)。

当地图被平移和缩放时，就会调用 mapView:regionWillChangeAnimated:和 mapView:regionDidChangeAnimated:委托方法。在示例程序中不必对地图尺寸进行修改和调整标注，不过在这个应用中有大量潜在的元素可以进行编辑，或者程序在不同的缩放级别可以显示多种不同的信息，这些委托方法可以实现移除地图中不再显示的标注，还可以添加新的标注。示例程序中的委托方法演示了如何得到一个新的显示地图区域，可以用来查询地图上显示的各

种控件。

```
-(void)mapView:(MKMapView *)mapView
➥regionDidChangeAnimated:(BOOL)animated
{
    MKCoordinateRegion newRegion = [mapView region];
    CLLocationCoordinate2D center = newRegion.center;
    MKCoordinateSpan span = newRegion.span;

    NSLog(@"New map region center: <%f/%f>, span: <%f/%f>",
    ➥center.latitude,center.longitude,span.latitudeDelta,
    ➥span.longitudeDelta);
}
```

处理拖曳标注和点击 callout 的方法将在下一节中介绍。

2.4 地图标注和覆盖物

地图视图(MKMapView)是一个很特殊的滚动视图，采用标准方法向其中添加的子视图是不能在地图视图中滚动的，不过该子视图还是可以保持对地图视图框架的静态关联。虽然这有些像悬浮按钮或文本标签等控件的特点，不过能够识别且做标记、加入详情信息到地图上才是最核心的功能。使用标注和覆盖物在地图视图上标注兴趣点是常用的方法。标注和覆盖物的位置不会随着地图的滚动和缩放而变化。地图标注通过一个单独的坐标点定义，地图覆盖物可以是直线、多边形或其他复杂的形状。MapKit 在逻辑标注或覆盖物和相关视图间的绘制有所区别，标注和覆盖物都是在地图上直接显示的数据。在需要显示这些标注时，地图视图随后会向视图请求标注或覆盖物数据，这类似于表视图在需要时根据索引值请求相应的单元格。

2.4.1 添加标注

在地图视图中任何对象都可以是标注，要成为标注，对象需要实现 MKAnnotation 协议。苹果公司建议尽量不要使用过多的标注对象，因为地图视图会保持对所有添加到视图中标注的引用，如果标注太多，地图在滚动和缩放时可能会影响程序的响应速度。如果对于标注的请求是简单请求，使用 MKPointAnnotation 对象即可。示例程序的 ICFFavoritePlace 类首先实现了 MKAnnotation 协议，其中有一个 NSManagedObject 子类，所以它可以一直使用 Core Data 框架。欲了解更多有关使用 Core Data 和 NSManagedObject 子类的内容，可以参见第 15 章“开始使用 Core Data”。

要实现 MKAnnotation 协议，类必须实现 coordinate 属性，这个属性供地图视图用来确定标注的添加位置。coordinate 是一个 CLLocationCoordinate2D 类型的值。

```
-(CLLocationCoordinate2D)coordinate
{
    CLLocationDegrees lat =
    ➥[[self valueForKeyPath:@"latitude"] doubleValue];
```

```
    CLLocationDegrees lon =
    ➥[[self valueForKeyPath:@"longitude"] doubleValue];

    CLLocationCoordinate2D coord =
    ➥CLLocationCoordinate2DMake(lat, lon);

    return coord;
}
```

由于 ICFFavoritePlace 类分别保存了各位置的经纬度信息，因此 coordinate 属性方法会通过 Core Location 框架提供的 CLLocationCoordinate2DMake 函数创建一个 CLLocationCoordinate2D 对象，它的值是根据其经纬度确定的。ICFFavoritePlace 会在 setter 方法中打散 CLLocationCoordinate2D 对象，将经度值和纬度值分别赋给 coordinate 属性。

```
-(void)setCoordinate:(CLLocationCoordinate2D)newCoordinate
{
    [self setValue:@(newCoordinate.latitude)
       forKeyPath:@"latitude"];

    [self setValue:@(newCoordinate.longitude)
       forKeyPath:@"longitude"];
}
```

MKAnnotation 协议中的另外两个属性 title 和 subtitle 可以有选择性地实现，它们的作用在于当用户点击标注时，地图视图会显示一个 callout，callout 的内容即 title 和 subtitle，如图 2-7 所示。

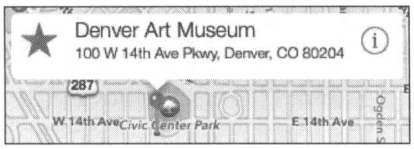


图 2-7　FavoritePlaces 示例程序：显示地图标注及其 callout

title 属性是 callout 最上面一行的内容，subtitle 属性则是 callout 下面一行的内容。

```
-(NSString *)title
{
    return [self valueForKeyPath:@"placeName"];
}

-(NSString *)subtitle
{
    NSString *subtitleString = @"";

    NSString *addressString =
    ➥[self valueForKeyPath:@"placeStreetAddress"];

    if([addressString length] > 0)
    {
        NSString *addr =
```

```
        ➥[self valueForKeyPath:@"placeStreetAddress"];

        NSString *city = [self valueForKeyPath:@"placeCity"];
        NSString *state = [self valueForKeyPath:@"placeState"];
        NSString *zip = [self valueForKeyPath:@"placePostal"];

        subtitleString =
        ➥[NSString stringWithFormat:@"%@, %@, %@ %@",
         ➥addr,city,state,zip];
    }
    return subtitleString;
}
```

在 ICFMainViewController 中，updateMapAnnotations 方法在 viewDidLoad:中被调用，用于填充标注，在兴趣点的从编辑视图(detail editing view)消失时会再次调用该方法。该方法首先会移除地图视图上的所有标注，虽然它在处理少量标注时很有效，不过对于有大量标注存在的情况，还是需要开发出更好的方法，将不用标注移除的同时添加新的标注。

```
[self.mapView removeAnnotations:self.mapView.annotations];
```

接下来，该方法会执行 Core Data 的获取请求，以得到保存了兴趣点信息的 NSArray 数组，并将这个数组添加到地图视图的标注中。

```
NSFetchRequest *placesRequest =
➥[[NSFetchRequest alloc] initWithEntityName:@"FavoritePlace"];

NSManagedObjectContext *moc = kAppDelegate.managedObjectContext;

NSError *error = nil;

NSArray *places = [moc executeFetchRequest:placesRequest
                                     error:&error];

if(error)
{
    NSLog(@"Core Data fetch error %@, %@", error,
    ➥[error userInfo]);
}
[self.mapView addAnnotations:places];
```

之后，地图视图将会在地图上显示所添加的标注。

2.4.2 显示标准和自定义的标注视图

标注视图表示地图上的一个标注。MapKit 框架提供两个标准的标注视图，分别是表示搜索位置结果的大头针标注和表示当前位置的蓝色跳动圆点。标注视图也可以使用静态图片进行自定义，或者完全使用 MKAnnotationView 子类进行定义。示例程序中对兴趣点位置使用标准的大头针标注，对当前位置使用标准的蓝色跳动圆点，对于可拖动的标注使用绿色箭头，如图 2-8 所示。

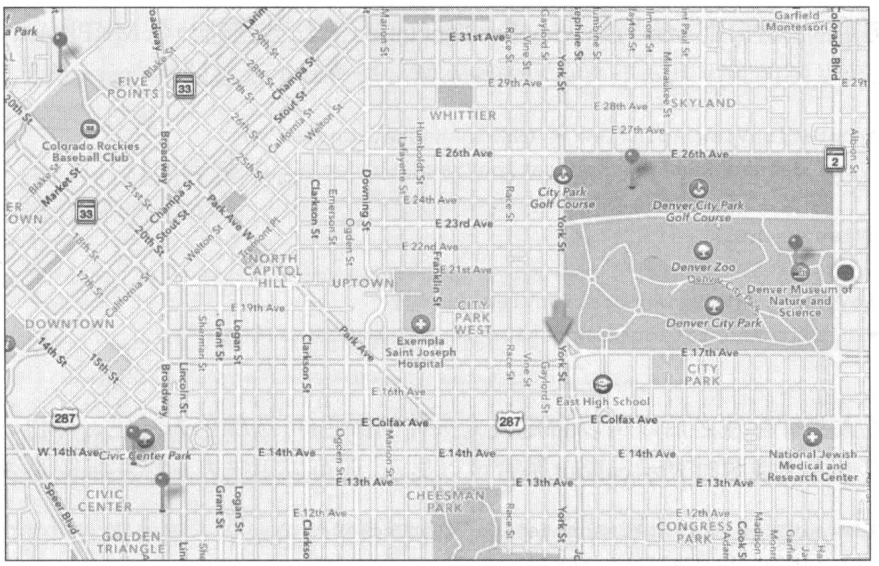


图 2-8　FavoritePlaces 示例程序：显示地图标注视图

要在地图视图中为标注显示相应的视图，地图视图委托函数需要实现 mapView:viewFor-Annotation 方法。在示例程序中，mapView:viewForAnnotation 方法在 ICFMainViewController 类中实现。该方法首先检查标注是否为当前位置。

```
if(annotation == mapView.userLocation)
{
    return nil ;
}
```

如果是当前位置，将 nil 返回给标注视图，程序就会告知地图视图在这个点使用标准蓝色圆点。之后，该方法会检查 ICFFavoritePlace 标注来确定程序所使用的标注类型。如果标注为“goingNext”，则会返回一个自定义的标注视图，否则会返回一个标准的大头针标注视图。

```
MKAnnotationView *view = nil;

ICFFavoritePlace *place = (ICFFavoritePlace *) annotation;

if([[place valueForKeyPath:@"goingNext"] boolValue])
{
    ...
}
else
{
    ...
}

return view;
```

要返回标准的大头针标注视图，首先需要尝试取消已经存在但不使用的标注视图队列。

如果没有可用的标注，则 mapView:viewForAnnotation 方法会创建一个 MKPinAnnotationView 实例。

```
MKPinAnnotationView *pinView = (MKPinAnnotationView *)
➥[mapView dequeueReusableAnnotationViewWithIdentifier:@"pin"];

if(pinView == nil)
{
    pinView = [[MKPinAnnotationView alloc]
    ➥initWithAnnotation:annotation reuseIdentifier:@"pin"];
}
```

创建了大头针标注之后，可以自定义大头针的颜色(可选颜色为红色、绿色、紫色)，用于在用户点击标注视图时表示 callout 是否可以显示，还可以用来表示标注视图是否可以拖动。

```
[pinView setPinColor:MKPinAnnotationColorRed];
[pinView setCanShowCallout:YES];
[pinView setDraggable:NO];
```

当点击标注视图时，在弹出的 callout 视图中还可以在其左侧和右侧添加自定义的 accessory 视图。左侧的 accessory 视图用于设置自定义图片，右侧的 accessory 视图用于设置标准的详情展开按钮。如果使用继承 UIControl 的对象自定义左右侧的 accessory 视图，则委托方法 mapView:annotationView:calloutAccessoryControlTapped:会在用户点击标注视图时被调用。否则对象就会按照开发者的需求进行配置。值得注意的是，苹果公司对于 accessory 视图给出的最大高度为 32 像素。

```
UIImageView *leftView = [[UIImageView alloc]
➥initWithImage:[UIImage imageNamed:@"annotation_view_star"]];

[pinView setLeftCalloutAccessoryView:leftView];

UIButton* rightButton = [UIButton buttonWithType:
                        ➥UIButtonTypeDetailDisclosure];

[pinView setRightCalloutAccessoryView:rightButton];
view = pinView;
```

要返回一个标准的大头针标注视图，委托方法需要尝试通过字符串标识符移除已经存在但不再使用的标注。如果没有可用的标注，该方法会创建一个 MKPinAnnotationView 实例。

```
view = (MKAnnotationView *)
➥[mapView dequeueReusableAnnotationViewWithIdentifier:@"arrow"];

if(view == nil)
{
    view = [[MKAnnotationView alloc]
    ➥initWithAnnotation:annotation reuseIdentifier:@"arrow"];
}
```

自定义的标注可以同标准的大头针标注类似，用于表示在用户点击标注视图时是否显示 callout，以及表示标注视图是否可以拖曳。最主要的区别在于标注用到的图片可以直接使用 setImage:方法进行设置。

```
[view setCanShowCallout:YES];
[view setDraggable:YES];

[view setImage:[UIImage imageNamed:@"next_arrow"]];

UIImageView *leftView = [[UIImageView alloc]
➥ initWithImage:[UIImage imageNamed:@"annotation_view_arrow"]];

[view setLeftCalloutAccessoryView:leftView];
[view setRightCalloutAccessoryView:nil];
```

上例中的标注将使用一个绿色的箭头替代标准的大头针进行显示，如之前的图 2-8 所示。

2.4.3　可拖曳的标注视图

可拖曳的标注视图对于希望在地图上标记位置的用户很有帮助。在示例程序中，有一个特殊的兴趣点是用来表示用户下一步去往的目的地，用绿色箭头表示。在配置过程中将 draggable 属性设置为 YES，可以将标注视图变为可拖曳的对象。

```
[view setDraggable:YES];
```

之后用户可以拖曳这个视图到地图中的任何位置，要了解更多关于标注视图拖曳操作的内容，可以在地图视图委托函数中通过实现 mapView:annotationView:didChangeDragState:fromOldState:方法来进一步了解相应的内容。在可拖曳标注视图的拖曳属性发生变化时就会激活这个方法，并显示当前的拖曳状态为以下状态之一：无拖曳、开始拖曳、拖曳中、取消、结束等。通过检查新的拖曳状态和之前的状态，逻辑程序段就可以根据拖曳的状态来处理多种不同的用例。

当用户在示例程序中停止拖曳箭头时，就会对箭头所指的新位置获取地理位置编码(在下面的 2.5 节中会详细介绍)，得到新位置的名称和地址。要实现这一功能，还需要检查拖曳动作是否完成。

```
if(newState == MKAnnotationViewDragStateEnding)
{
    ....
}
```

如果拖曳动作完成，该方法将会得到带有标注视图的标注点，通过反向地理编码就可以得到坐标信息。

```
ICFFavoritePlace *draggedPlace =
➥(ICFFavoritePlace *)[annotationViewannotation];
```

该方法还将一个标准的等待提示 Spinner 添加到了 callout 视图，这样用户就知道正在更

新位置，之后对新位置调用反向地理编码方法，后面将详细介绍有关地理编码的内容。

```
UIActivityIndicatorViewStyle whiteStyle =
➥ UIActivityIndicatorViewStyleWhite;

UIActivityIndicatorView *activityView =
➥[[UIActivityIndicatorView alloc]
➥initWithActivityIndicatorStyle:whiteStyle];

[activityView startAnimating];
[annotationView setLeftCalloutAccessoryView:activityView];

[self reverseGeocodeDraggedAnnotation:draggedPlace
                    forAnnotationView:annotationView];
```

2.4.4 使用地图覆盖物

地图覆盖物同标注相似，每个对象需要实现 MKOverlay 协议，并且地图视图委托函数也被要求为地图覆盖物提供相关的视图。地图覆盖物不同于标注最主要的方面是覆盖物可以表示更多的东西，而不只是表示地图上的一个点。它可以表示为线段和形状，所以非常适合用来表示路线或地图上感兴趣的一块区域。为了演示覆盖物的这些特征，示例程序加入了地理围栏功能(在下面的 2.0 节中会有详细的介绍)，即用户选定一处感兴趣的区域，设置半径将该区域包含进来。地理围栏功能添加成功后，用户选择半径大小，就会以位置坐标处为圆心生成一个圆形区域，如图 2-9 所示。

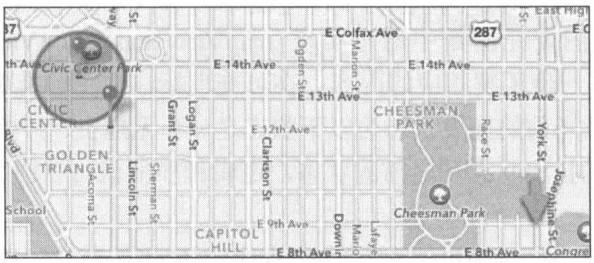


图 2-9 FavoritePlaces 示例程序：显示地图覆盖物视图

在之前的 2.4.1 节中曾提到，updateMapAnnotations 方法可以向地图添加标注。该方法同样也可以向地图添加覆盖物。这个方法首先会清理地图视图上所有已存在的覆盖物。

```
[self.mapView removeOverlays:self.mapView.overlays];
```

由于覆盖物只在地理围栏功能激活时才会显示，因此该方法会遍历这些位置，并只向这些需要添加覆盖物的地图区域添加设置好的覆盖物。

```
for(ICFFavoritePlace *favPlace in places)
{
    BOOL displayOverlay =
    ➥[[favPlace valueForKeyPath:@"displayProximity"] boolValue];

    if(displayOverlay)
```

```
    {
        [self.mapView addOverlay:favPlace];
        ...
    }
}
```

当地图需要显示一个地图覆盖物时，地图视图将会调用委托方法 mapView:viewForOverlay。该方法将会在地图上创建一个用于显示的覆盖物。MapKit 给出了 3 种选择：圆形、多边形和折线段。如果觉得这 3 种图案还不够，也可以自定义覆盖物的形状。示例程序根据兴趣点的半径和地图坐标，在指定位置的周围创建了一个圆形的区域。

```
ICFFavoritePlace *place = (ICFFavoritePlace *)overlay;

CLLocationDistance radius =
➥[[place valueForKeyPath:@"displayRadius"] floatValue];

MKCircle *circle =
➥[MKCircle circleWithCenterCoordinate:[overlay coordinate]
                                radius:radius];
```

Mapkit 圆形区域准备好之后，会创建 Mapkit 视图，并且自定义线段类型、填充颜色和线宽。之后返回该圆形视图，在地图上显示它。

```
MKCircleRenderer *circleView =
➥[[MKCircleRenderer alloc] initWithCircle:circle];

circleView.fillColor =
➥[[UIColor redColor] colorWithAlphaComponent:0.2];

circleView.strokeColor =
➥[[UIColor redColor] colorWithAlphaComponent:0.7];

circleView.lineWidth = 3;

return circleView;
```

2.5　地理编码和反向地理编码

地理编码(geocoding)是指从人们可以理解的地址信息，经过处理得到该地址信息由经纬度表示的坐标点；反向地理编码(reverse-geocoding)就是根据对坐标点的处理得出人们可以理解的地理位置信息。从 iOS 5 开始，Core Location 框架就支持这两种编码方式，并取消了一些特殊的条件或诸多限制(同 iOS 5.1 及之前版本中的 MapKit 一样)。

2.5.1　对地址进行地理编码

在示例程序中，用户可以在 ICFFavoritePlaceViewController 中通过输入地址来添加新的兴趣点。用户可以点击 Geocode Location Now 得到该位置的经纬度，如图 2-10 所示。

图 2-10 FavoritePlaces 示例程序：添加新的兴趣点

当用户点击 Geocode Location Now 按钮时，就会调用 geocodeLocationTouched:方法。该方法首先对用户输入的字符串进行连接，比如 2100 York St, Denver, CO 80205，再将其提供给地理编码器。

```
NSString *geocodeString = @"";
if([self.addressTextField.text length] > 0)
{
    geocodeString = self.addressTextField.text;
}
if([self.cityTextField.text length] > 0)
{
    if([geocodeString length ] > 0)
    {

        geocodeString =
        ➥[geocodeString stringByAppendingFormat:@", %@",
        ➥self.cityTextField.text];

    }
    else
    {
       geocodeString = self.cityTextField.text;
    }
}
if([self.stateTextField.text length] > 0)
    {
    if([geocodeString length] > 0)
    {

        geocodeString =
        ➥[geocodeString stringByAppendingFormat:@", %@",
        ➥self.stateTextField.text];
```

```
    }
    else
    {
        geocodeString = self.stateTextField.text;
    }
}
if([self.postalTextField.text length] > 0)
    {
    if([geocodeString length] > 0)
    {

        geocodeString =
        ➥[geocodeString stringByAppendingFormat:@" %@",
        ➥self.postalTextField.text];

    }
    else
    {
       geocodeString = self.postalTextField.text;
    }
}
```

为避免因多次点击带来的额外请求，该方法会暂时禁用 Geocode Location Now 按钮。苹果公司明确说明地理编码器同一时间只能处理一条请求，该方法会更新对话框和按钮的状态来提示地理编码正在进行中。

```
[self.latitudeTextField setText:@"Geocoding..."];
[self.longitudeTextField setText:@"Geocoding..."];

[self.geocodeNowButton setTitle:@"Geocoding now..."
                        forState:UIControlStateDisabled];

[self.geocodeNowButton setEnabled:NO];
```

该方法获取对 CLGeocoder 实例的引用：

```
CLGeocoder *geocoder =
➥[[ICFLocationManager sharedLocationManager] geocoder];
```

之后，geocoder 对象会对地址字符串进行地理编码，并在主队列中调用对应的 completion handler 代码块。completion handler 代码块首先重启按钮，使其处于可被点击的状态，之后会检查在编码过程中是否出现错误或者 geocoder 对象是否成功完成编码。

```
[geocoder geocodeAddressString:geocodeString
➥completionHandler:^(NSArray *placemarks, NSError *error) {

    [self.geocodeNowButton setEnabled:YES];
    if(error)
    {
       ...
```

```
        }
        else
        {
            ...
        }
    }];
```

如果地理编码器遇到错误，经纬度区域的值会被 Not found 填充并呈现一个提醒框视图，带有错误描述信息。如果没有网络连接，地理编码器就无法工作；同样，如果地址格式不正确或者无法找到地址，也都会返回错误信息。

```
[self.latitudeTextField setText:@"Not found"];
[self.longitudeTextField setText:@"Not found"];

UIAlertController *alertController =
➥[UIAlertController alertControllerWithTitle:@"Geocoding Error"
                                      message:error.localizedDescription
                               preferredStyle:UIAlertControllerStyleAlert];

[alertController addAction:
➥[UIAlertAction actionWithTitle:@"OK"
                           style:UIAlertActionStyleCancel
                         handler:nil]];

[self presentViewController:alertController
                   animated:YES
                 completion:nil];
```

如果地理编码成功，会向 completion handler 传递一个名为 placemarks 的数组。这个数组包含 CLPlacemark 实例，每个实例都包含潜在的匹配信息。placemark 带有经纬度坐标和地址信息。

```
if([placemarks count] > 0)
{
    CLPlacemark *placemark = [placemarks lastObject];

    NSString *latString =
    ➥[NSString stringWithFormat:@"%f",
    ➥placemark.location.coordinate.latitude];

    [self.latitudeTextField setText:latString];

    NSString *longString =
    ➥[NSString stringWithFormat:@"%f",
    ➥placemark.location.coordinate.longitude];

    [self.longitudeTextField setText:longString];
}
```

如果返回多个 placemark，用户界面允许用户选择一个与其需求最匹配的位置(Maps.app

使用的就是这个方法)。为了理解起来简单一点，示例程序选择数组中的最后一个 placemark 并使用其坐标信息来更新用户界面。

2.5.2　对位置进行反向地理编码

示例程序允许用户拖动绿色箭头，选择下一个目的地，如图 2-11 所示。

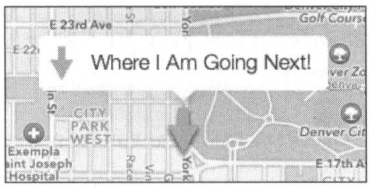


图 2-11　FavoritePlaces 示例程序：我的下一个目的地

当用户拖动绿色箭头时，会调用 ICFMainViewController 类中的地图视图委托方法 mapView:annotationview:didChangeDragState:fromOldState:。该方法会检查拖动状态，之前我们在“可拖曳的标注”一节中对各种状态进行了介绍，如果已停止拖动绿色箭头，则使用 spinner 更新 callout 视图并开始进行反向地理编码。

```
ICFFavoritePlace *draggedPlace =
➥(ICFFavoritePlace *)[annotationView annotation];

UIActivityIndicatorViewStyle whiteStyle =
➥UIActivityIndicatorViewStyleWhite;

UIActivityIndicatorView *activityView =
➥[[UIActivityIndicatorView alloc]
➥initWithActivityIndicatorStyle:whiteStyle];

[activityView startAnimating];
[annotationView setLeftCalloutAccessoryView:activityView];

[self reverseGeocodeDraggedAnnotation:draggedPlace
                 forAnnotationView:annotationView];
```

reverseGeocodeDraggedAnnotation:forAnnotationView:方法获取一个对 CLGeocoder 实例的引用。

```
CLGeocoder *geocoder =
➥[[ICFLocationManager sharedLocationManager] geocoder];
```

地理位置编码器会根据移动中箭头的坐标为其创建一个 CLLocation 实例。

```
CLLocationCoordinate2D draggedCoord = [place coordinate];

CLLocation *draggedLocation =
➥[[CLLocation alloc] initWithLatitude:draggedCoord.latitude
                        longitude:draggedCoord.longitude];
```

之后地理编码器会在 completion handler 代码块中对拖曳位置的标注进行反向地理编码，

这一调用在主队列中完成。completion handler 会使用绿色箭头代替 callout 视图中的 spinner，之后检查编码过程中是否出现错误或编码是否成功完成。

```
[geocoder reverseGeocodeLocation:draggedLocation
➥completionHandler:^(NSArray *placemarks, NSError *error) {

    UIImage *arrowImage =
    ➥[UIImage imageNamed:@"annotation_view_arrow"];

    UIImageView *leftView =
    ➥[[UIImageView alloc] initWithImage:arrowImage];

    [annotationView setLeftCalloutAccessoryView:leftView];

    if(error)
    {
        ...
    }
    else
    {
        ...
    }
}];
```

如果地理编码器遇到错误，就会弹出带有错误问题描述的提醒框。在没有网络连接的时候地理编码器无法工作。

```
UIAlertController *alertController =
➥[UIAlertController alertControllerWithTitle:@"Geocoding Error"
                                      message:error.localizedDescription
                               preferredStyle:UIAlertControllerStyleAlert];

[alertController addAction:
➥[UIAlertAction actionWithTitle:@"OK"
                           style:UIAlertActionStyleCancel
                         handler:nil]];

[self presentViewController:alertController
                   animated:YES
                 completion:nil];
```

反向地理编码成功完成时，将把一个由 CLPlacemark 实例组成的数组对象传给 completion handler。示例程序将会使用最后一个 placemark 来更新下一个目的地的名称和地址。

```
if([placemarks count] > 0)
{
    CLPlacemark *placemark = [placemarks lastObject];
    [self updateFavoritePlace:place withPlacemark:placemark];
}
```

placemark 包含带有遵循国际标准配置的详细位置信息，比如由数字表示的街道地址(或 subThoroughfare)和街道名(或 thoroughfare)，表示城市和州的参数分别为 subAdministrativeArea 和 administrativeArea。

```
[kAppDelegate.managedObjectContext performBlock:^{
    NSString *newName =
    ➥[NSString stringWithFormat:@"Next: %@",placemark.name];

    [place setValue:newName forKey:@"placeName"];

    NSString *newStreetAddress =
    ➥[NSString stringWithFormat:@"%@ %@",
    ➥placemark.subThoroughfare, placemark.thoroughfare];

    [place setValue:newStreetAddress
           forKey:@"placeStreetAddress"];

    [place setValue:placemark.subAdministrativeArea
           forKey:@"placeCity"];

    [place setValue:placemark.postalCode
           forKey:@"placePostal"];

    [place setValue:placemark.administrativeArea
           forKey:@"placeState"];

    NSError *saveError = nil;
    [kAppDelegate.managedObjectContext save:&saveError];
    if(saveError) {
        NSLog(@"Save Error: %@",saveError.localizedDescription);
    }
}];
```

提示

geocoder 对象的 CLPlacemark 实例包含一个 addressDictionary 属性，该属性可以对数据进行格式化，以便向 Address Book 中添加数据(欲了解更多信息，可以查看第 5 章“学习使用 Address Book”)。

之后，使用 Core Data 保存位置信息，这就保证了应用重启后数据仍然存在。现在当用户点击绿色箭头标注视图时，将显示拖曳位置的名字和地址，如图 2-12 所示。

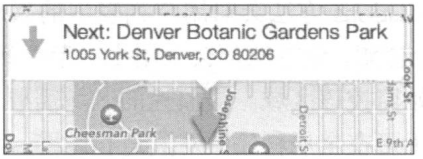


图 2-12　FavoritePlaces 示例程序：使用反向地理编码得到下个目的地的地址

2.6 地理围栏

地理围栏(geofencing)也称为区域监控，就是对设备进入或离开某个指定的地图区域进行跟踪。iOS 把这一功能很好地与 Siri 结合在一起，比如“当我离开办公室时提醒我带上面包”或者“当到家时提醒我将肉放入烤箱”。iOS 还在 Passbook 程序中使用了地理围栏，用户在主屏幕上可以看到与当前位置相关的信息(欲了解更多内容，可以参阅第 25 章“Passbook 和 PassKit”)。

2.6.1 判断区域监控是否可用

Core Location 框架的位置管理器对象具有一个用来判断设备是否具有区域监控能力的类方法，可以对这个方法进行自定义以检测应用是否支持区域监控。比如在示例程序中，将在 ICFFavoritePlaceViewController 中根据情况显示兴趣点的地理围栏。

```
BOOL hideGeofence =
➥![CLLocationManager isMonitoringAvailableForClass:[CLRegion class]];

[self.displayProximitySwitch setHidden:hideGeofence];

if(hideGeofence)
{
    [self.geofenceLabel setText:@"Geofence N/A"];
}
```

2.6.2 定义边界

Core Location 的位置管理器(CLLocationManager)为应用保存被监控的区域集合，在 ICFMainViewController 中，当位置发生变化时，updateMapAnnotations:方法会清理被监控的区域集合。

```
CLLocationManager *locManager =
➥[[ICFLocationManager sharedLocationManager] locationManager];

NSSet *monitoredRegions = [locManager monitoredRegions];

for(CLRegion *region in monitoredRegions)
{
    [locManager stopMonitoringForRegion:region];
}
```

接下来，updateMapAnnotations:方法会对用户列表中的兴趣点进行遍历，判断用户设置地理围栏的位置。对于每一个位置，该方法都会采用上一节介绍的方法添加一个覆盖物视图，并在之后通知位置管理器开始监控该区域。被监控的区域需要一个中心坐标、一个半径和一个标识符，标识符的作用在于可在全局程序中对其进行跟踪。示例程序使用 Core Data 全局资源 ID 作为标识符，这样当某一区域发生区域监控事件时就可以快速找到该地点。

```
NSString *placeObjectID =
```

```
➥[[[favPlace objectID] URIRepresentation] absoluteString];


CLLocationDistance monitorRadius =
➥[[favPlace valueForKeyPath:@"displayRadius"] floatValue];

CLCircularRegion *region =
➥[[CLCircularRegion alloc] initWithCenter:[favPlace coordinate]
                                    radius:monitorRadius
                                identifier:placeObjectID];

[locManager startMonitoringForRegion:region];
```

值得注意的是，目前区域监控只支持圆形区域。

2.6.3　监控变更

当设备进入或离开一个监控区域时，位置管理器通过调用 locationManager:didEnterRegion: 或 locationManager:didExitRegion:方法将事件通知给相应的委托函数。

locationManager:didEnterRegion:方法首先获取监控区域的标识符。当位置管理器被通知去对某一区域进行监控时，就会为其分配一个标识符，该标识符就是被保存兴趣点的 Core Data URI。这个 URI 之前用于管理对象 ID，即从被管理的对象文本中获取兴趣点信息。

```
NSString *placeIdentifier = [region identifier];
NSURL *placeIDURL = [NSURL URLWithString:placeIdentifier];

NSManagedObjectID *placeObjectID =
➥[kAppDelegate.persistentStoreCoordinator
➥managedObjectIDForURIRepresentation:placeIDURL];
```

locationManager:didEnterRegion:方法从兴趣点获取相应的详细信息，并将它们以提醒框的形式呈现给用户。

```
[kAppDelegate.managedObjectContext performBlock:^{

    ICFFavoritePlace *place =
    ➥(ICFFavoritePlace *)[kAppDelegate.managedObjectContext
    ➥objectWithID:placeObjectID];

    NSNumber *distance = [place valueForKey:@"displayRadius"];
    NSString *placeName = [place valueForKey:@"placeName"];

    NSString *baseMessage =
    ➥@"Favorite Place %@ nearby - within %@ meters.";

    NSString *proximityMessage =
    ➥[NSString stringWithFormat:baseMessage,placeName,distance];

    UIAlertController *nearbyAlertController =
```

```
    ➥[UIAlertController alertControllerWithTitle:@"Favorite Nearby!"
                                          message:proximityMessage
                                   preferredStyle:UIAlertControllerStyleAlert];

    [nearbyAlertController addAction:
     ➥[UIAlertAction actionWithTitle:@"OK"
                                style:UIAlertActionStyleCancel
                              handler:nil]];

    ICFAppDelegate *appDelegate =
    ➥(ICFAppDelegate *)[[UIApplication sharedApplication] delegate];

    [appDelegate.window.rootViewController presentViewController:nearbyAlertController
                                                         animated:YES
                                                       completion:nil];
}];
```

使用示例程序测试上述功能，选择带有 Denver Museum of Nature and Science (DMNS)信息的 GPX 文件，在调试模式下运行程序。确保 Denver Art Museum 被设置为地理围栏，如图 2-9 所示。在应用运行后，使用 Xcode 的调试位置菜单从 DMNS(如图 2-6 所示)移到 Denver Art Museum。这样就触发了地理围栏并显示一个提醒框，如图 2-13 所示。

图 2-13 FavoritePlaces 示例程序：兴趣点附近的提醒框

locationManager:didExitRegion:方法还会从区域中获得 Core Data 标识符，使用 Core Data 得到被管理的对象 ID，查找兴趣点并在用户离开该区域时弹出提醒框。要通过示例程序测试该功能，首先从图 2-13 所示的 Favorite Nearby 提醒框开始，点击 OK 按钮，之后从 iOS Simulator 菜单中选择 Debug,Location,Apple。几秒钟之后，模拟器将会变更位置并弹出一个提醒框，如图 2-14 所示。

位置管理器会故意延迟调用委托方法，为了避免由于设备离区域边界过近而出现错误消息，需要给出至少 20 秒的缓冲距离。

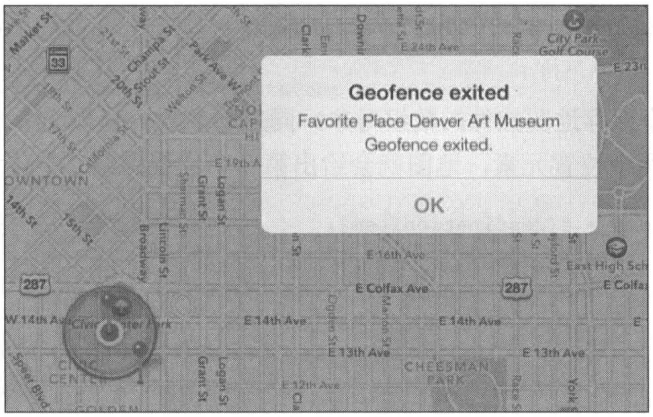


图 2-14　FavoritePlaces 示例程序：离开兴趣点区域的提醒框

2.7　获取路径

在 iOS 6 中，苹果公司内置的 Map.app 程序在功能上得到了加强，除了路径之外还加入了线路规划导航。Map.app 还可以通过特定的指令从其他程序激活地图显示。应用可以从 Map.app 中请求显示多组地图控件，比如提供两个位置间的路径，或者从当前位置到目的地的路径等。可以设置 Map.app 的中心和范围大小，以及地图显示的类型(标准、卫星或混合方式)。到了 iOS 7，MapKit 提供的路径请求可以直接在应用中使用。请求路径会返回一组用于表示路线选择的线段，同时还会以表视图的方式显示相关的路线段信息。每一种方法都会在示例程序中给出演示。

要在程序中使用 Map.app，可以使用 MKMapItem 类的 openMapsWithItems:launchOptions: 类方法或实例方法 openInMapsWithlaunchOptions:。在示例程序中，ICFFavoritePlaceViewController 中有一个按钮，用来获取兴趣点的路径。当点击按钮时，会调用 getDirectionsButtonTouched: 方法，在该方法中会为兴趣点创建一个 MKMapItem 实例。

```
CLLocationCoordinate2D destination =
➥[self.favoritePlace coordinate];

MKPlacemark *destinationPlacemark =
➥[[MKPlacemark alloc] initWithCoordinate:destination
                       addressDictionary:nil];

MKMapItem *destinationItem =
➥[[MKMapItem alloc] initWithPlacemark:destinationPlacemark];

destinationItem.name =
➥[self.favoritePlace valueForKey:@"placeName"];
```

Map.app 启动的相关设置需要在启动选项字典文件中进行设置。

```
NSDictionary *launchOptions = @{
    MKLaunchOptionsDirectionsModeKey:
    MKLaunchOptionsDirectionsModeDriving,
    MKLaunchOptionsMapTypeKey:
```

```
    [NSNumber numberWithInt:MKMapTypeStandard]
};
```

之后会创建一个保存地图控件的数组对象，同启动参数一起传递给 Map.app。如果传递的数组对象中包含两个位置元素，地图就会给出第一个位置到第二个位置的路径。

```
NSArray *mapItems = @[destinationItem];

BOOL success = [MKMapItem openMapsWithItems:mapItems
                              launchOptions:launchOptions];

if(!success)
{
    NSLog(@"Failed to open Maps.app.");
}
```

Map.app 此时已打开，同时会显示目的兴趣点的路径。如果遇到错误，openMapsWithItems:launchOptions:方法会返回 NO。

要在应用中显示路径，需要用 MKDirectionsRequest 实例来实例化一个 MKDirections 对象，并指定源(或起点)和目的地(表示为 MKMapItem 实例)。

```
CLLocationCoordinate2D destination =
➥[self.favoritePlace coordinate];

MKPlacemark *destinationPlacemark =
➥[[MKPlacemark alloc] initWithCoordinate:destination
                        addressDictionary:nil];

MKMapItem *destinationItem =
➥[[MKMapItem alloc] initWithPlacemark:destinationPlacemark];

MKMapItem *currentMapItem =
➥[self.delegate currentLocationMapItem];

MKDirectionsRequest *directionsRequest =
➥[[MKDirectionsRequest alloc] init];

[directionsRequest setDestination:destinationItem];
[directionsRequest setSource:currentMapItem];

MKDirections *directions =
➥[[MKDirections alloc] initWithRequest:directionsRequest];
```

接下来调用 calculateDirectionsWithCompletionHandler:方法并指定 completion 代码块，该代码块用于处理所有错误信息及查看 MKDirectionsResponse 返回的状态。在本例的方法实现中，我们确保至少返回一条路线(一个 MKRoute 实例)，并根据第一条路线执行相应的操作。该方法会迭代第一条路线对象的 steps 属性，该属性包含 MKRouteStep 实例和用于表示每段线路的距离和详情的日志字符串。之后，calculateDirectionsWithCompletionHandler:方法会调

用委托向地图上添加线路。

```
[directions calculateDirectionsWithCompletionHandler:
➥^(MKDirectionsResponse *response, NSError *error){
   if(error) {

      NSString *dirMessage =
      ➥[NSString stringWithFormat:@"Failed to get directions: %@",
      ➥error.localizedDescription];

      UIAlertController *dirAlertController =
     ➥[UIAlertController alertControllerWithTitle:@"Directions Error"
                                           message:dirMessage
                                    preferredStyle:UIAlertControllerStyleAlert];

      [dirAlertController addAction:
      ➥[UIAlertAction actionWithTitle:@"OK"
                                 style:UIAlertActionStyleCancel
                               handler:nil]];

      [self presentViewController:dirAlertController
                         animated:YES
                       completion:nil];
   }
   else
   {
      if([response.routes count] > 0) {
         MKRoute *firstRoute = response.routes[0];
         NSLog(@"Directions received.  Steps for route 1 are: ");
         NSInteger stepNumber = 1;
         for(MKRouteStep *step in firstRoute.steps) {

            NSLog(@"Step %d, %f meters: %@",stepNumber,
            ➥step.distance,step.instructions);

            stepNumber++;
         }
         [self.delegate displayDirectionsForRoute:firstRoute];
      }
      else
      {
         NSString *dirMessage = @"No directions available";

         UIAlertController *dirAlertController =
         ➥[UIAlertController alertControllerWithTitle:@"No Directions"
                                               message:dirMessage
                                        preferredStyle:UIAlertControllerStyleAlert];

         [dirAlertController addAction:
          ➥[UIAlertAction actionWithTitle:@"OK"
```

```
                                    style:UIAlertActionStyleCancel
                                  handler:nil]];

        [self presentViewController:dirAlertController
                           animated:YES
                         completion:nil];
    }
  }
}];
```

在委托方法中，组成路线的折线段被添加到地图的覆盖物上，不需要弹出对话框。

```
-(void)displayDirectionsForRoute:(MKRoute *)route
{
    [self.mapView addOverlay:route.polyline];

    if(self.favoritePlacePopoverController)
    {
        [self.favoritePlacePopoverController
        ➥dismissPopoverAnimated:YES];

        self.favoritePlacePopoverController = nil;
    } else
    {
        [self dismissViewControllerAnimated:YES
                                 completion:nil];
    }
}
```

由于折线段是以覆盖物形式添加的，因此返回覆盖物视图的地图委托方法必须现在处理折线段，而不是处理自定义的地理围栏半径覆盖物。

```
-(MKOverlayRenderer *)mapView:(MKMapView *)mapView
            viewForOverlay:(id < MKOverlay >)overlay
{
    MKOverlayRenderer *returnView = nil;

    if([overlay isKindOfClass:[ICFFavoritePlace class]]) {
        ...
    }
    if([overlay isKindOfClass:[MKPolyline class]]) {
        MKPolyline *line = (MKPolyline *)overlay;

        MKPolylineRenderer *polylineRenderer =
        ➥[[MKPolylineRenderer alloc] initWithPolyline:line];

        [polylineRenderer setLineWidth:3.0];
        [polylineRenderer setFillColor:[UIColor blueColor]];
        [polylineRenderer setStrokeColor:[UIColor blueColor]];
        returnView = polylineRenderer;
    }
```

```
    return returnView;
}
```

mapView:viewForOverlay:方法将查看覆盖物属于哪个类，并为其创建正确的视图类型。对于折线段，该方法使用覆盖物上的折线段创建一个 MKPolylineRenderer 实例，然后选择线的宽度、蓝色填充和描边效果等对其进行自定义设置，用这条线在起点位置和目的地位置生成路径，如图 2-15 所示。

图 2-15　FavoritePlaces 示例程序：在地图上显示折线段路径

2.8　小结

本章介绍了 Core Location 和 MapKit 两个框架，学习了如何设置 Core Location、如何查看位置服务是否可用、如何处理用户请求以及如何获得设备的当前位置。

接下来介绍了如何使用 MapKit 框架在地图上显示带有标准标注和自定义标注的位置信息，还学习了在 callout 控件上如何显示更多标注的详细信息，如何对在 callout 控件上执行的点击或拖动标注等操作进行响应，以及如何向地图上添加覆盖物来强调地图上的关键位置。

之后学习了如何使用地理编码从地址信息解析得到该位置的经纬度，以及如何基于经纬度数据得知该位置的地址信息。

本章还介绍了地理围栏、区域监控两个功能，示例程序给出了当用户进入或离开地图指定区域时，程序应该如何处理这些事件的方法。

最后，本章演示了去往某一兴趣点的路径的两种方法，分别是使用 Map.app 来提供路径，以及直接在用户界面上通过路径请求的方法实现路径的获取和显示。

第3章

排 行 榜

排行榜(leaderboard)如今已经成为几乎所有游戏都非常重要的一个功能，很多非游戏应用也都开始使用排行榜。利用 Game Center 向应用添加排行榜比以前要容易得多。虽然排行榜几乎可以说是伴随着电子游戏的产生而产生(高分排行榜第一次出现是在 1976 年)，不过排行榜在日益火爆的社交类游戏中成为必要功能是近期才开始流行的。本章将介绍如何在真实的游戏中加入功能完善的排行榜，还将介绍如何在 iTunes Connect 中对信息进行设置以及使用 GKLeaderboardViewController 显示排行榜。

3.1 示例程序

在本章和第 4 章“成就系统”中我们都会使用同一个示例应用。了解游戏本身很重要，这样就不会在整合 Game Center 功能时感到困惑。例子 Whack-a-Cac 的设计宗旨是使代码量最小且最容易学习，这样该例就可以作为整合 Game Center 到任何应用中的泛型模板。如果已经有准备好的应用等着加入本章的内容，可以跳过本节的学习而使用自己的项目。

如图 3-1 所示，Whack-a-Cac 是一个简单的 Whack-a-Mole 类型的游戏。其中的仙人掌会随机蹦出，用户必须在它们缩回沙丘之前用很短的时间点击它们。随着游戏的进展会不断增加难度，当用户漏掉 5 个仙人掌之后游戏就会结束，并会给用户打分。游戏的实现通过 ICFGameViewController 类来控制，仙人掌随机出现在三行中的一行的某个位置。玩家在仙人掌弹出到缩回沙丘之间只有两秒的时间可以点击它们，每漏掉一个扣除一点生命值。获得 50 分之后，每击中 10 个仙人掌就会增加同一时间弹出的仙人掌个数，在游戏过程中可以暂停和恢复。

在深入研究游戏代码之前，首先应该注意的是游戏的主界面菜单。IFCViewController.m 并不只处理游戏的启动过程，还要提供用户访问排行榜和成就系统(第 4 章的内容)的功能。Game Center 相关的功能将在本章后面详细讲解，现在我们把注意力放在 play:方法上。当一个新游戏被创建时，IFCViewController 会在 IFCGameViewController 类中调用 alloc 和 init 方法，并将其推入导航栈，游戏的其余部分将会在这个类中处理。

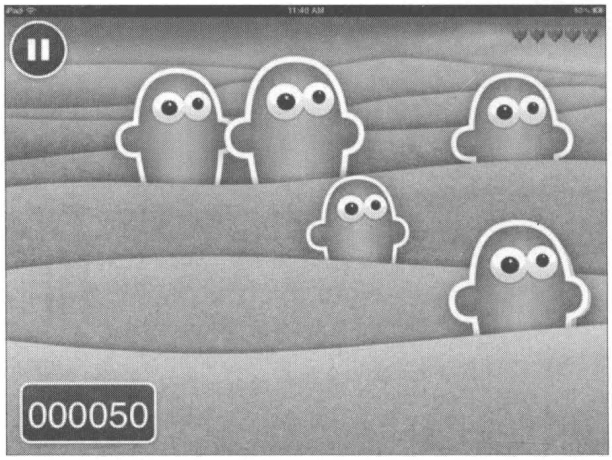


图 3-1 Whack-a-Cac 界面，Game Center 相关的两章都使用这个游戏作为例子

Whack-a-Cac 是 iOS 游戏中的一个简单例子。它所基于的状态引擎包含 3 种状态：进行中、暂停和游戏结束。当游戏处于进行中状态时，游戏引擎会持续生成仙人掌，直到玩家生命值耗尽或进入游戏结束状态。玩家可以随时点击屏幕左上角的暂停按钮暂停游戏。为了解该游戏引擎，下面直接介绍 viewDidLoad:方法。

```
-(void)viewDidLoad
{
    [[ICFGameCenterManager sharedManager] setDelegate:self];

    score = 0;
    life = 5;
    gameOver = NO ;
    paused = NO ;

    [super viewDidLoad];

    [self updateLife];

    [self spawnCactus];
    [self performSelector:@selector(spawnCactus) withObject:nil
    ➥ afterDelay:1.0];
}
```

代码第一行中的 ICFGameCenterManager 将其委托设置为 self，本节后面会深入讨论这一点。在 viewDidLoad 执行过程中，需要设置一些状态变量。首先得分应该被重新设置为 0，同时玩家生命值也应该被重新设置为 5。接下来需要设置两个布尔类型的值用来表示当前游戏的状态，游戏一旦开始，gameOver 和 paused 属性就应该被设置为 NO。之后调用一个名为 updateLife 的方法。虽然本节后面会对其进行讨论，不过现在也需要知道它是用来处理右上角玩家生命值的方法，如图 3-1 所示。初始化的最后一步就是在游戏开始时先要蹦出两个仙人掌，第一个在游戏启动瞬间出现，另一个延迟一秒之后出现。

3.1.1 弹出仙人掌

IFCGameViewController 类中一个重要的功能就是实现在屏幕上弹出仙人掌。在 viewDidLoad 方法中，spawnCactus 会被调用两次，现在我们来看下这个方法。在基于状态的游戏中，第一件事就要通过检查确保程序处于正确的状态。第一个测试是 gameOver 检查，如果游戏终止，仙人掌停止弹出。接下来是暂停测试，如果游戏被暂停，不能弹出新的仙人掌，当游戏恢复时继续弹出新的仙人掌。暂停状态的测试将每秒尝试弹出仙人掌，直到游戏恢复或退出才停止该尝试。

```
if(gameOver)
{
    return;
}

if(paused)
{
    [self performSelector:@selector(spawnCactus) withObject:nil
    ➥ afterDelay:1];

    return;
}
```

此方法完成上述检查后，就要开始弹出新的仙人掌了。首先游戏要确定弹出对象的随机位置，为此，需要生成两个随机数，一个表示仙人掌出现的行，另一个表示 x 轴上的一个随机位置。

```
NSInteger rowToSpawnIn = arc4random()%3;
NSInteger horizontalLocation = arc4random()%1024;
```

为创建更好的游戏体验，共有 3 张仙人掌图片。每当要弹出仙人掌时，运行下面的代码段随机选择一张图片：

```
NSInteger cactusSize = arc4random()%3;
UIImage *cactusImage = nil;

switch(cactusSize)
{
        case 0:
            cactusImage = [UIImage imageNamed:@"CactusLarge.png"];
            break;
        case 1:
            cactusImage = [UIImage imageNamed:@"CactusMedium.png"];
            break;
        case 2:
            cactusImage = [UIImage imageNamed:@"CactusSmall.png"];
            break;
        default:
```

```
            break;
    }
```

还有一个简单的检查就是要确保仙人掌不会出现在视图右边看不到的地方。因为 x 轴的位置是随机计算的，仙人掌的宽度是一个变量，所以需要一条简单的 if 语句来判断图片是否偏出屏幕，如果出现这种情况，就要将其移回边界之内。

```
if(horizontalLocation > 1024 - cactusImage.size.width)
      horizontalLocation = 1024 - cactusImage.size.width;
```

Whack-a-Cac 游戏具有深度和层的概念。游戏共有 3 个沙丘，仙人掌出现在本行沙丘的后面，即后一排沙丘的前面。实现这一效果的第一步是要确定新的仙人掌出现在哪个视图的后面，具体实现如下：

```
UIImageView *duneToSpawnBehind = nil;

switch(rowToSpawnIn)
{
    case 0:
        duneToSpawnBehind = duneOne;
        break;
    case 1:
        duneToSpawnBehind = duneTwo;
        break;
    case 2:
        duneToSpawnBehind = duneThree;
        break;
    default:
         break;
}
```

现在游戏知道在哪一层弹出新的仙人掌了，为了增强程序的可读性，在此设置了两个快捷变量。

```
CGFloat cactusHeight = cactusImage.size.height;
CGFloat cactusWidth = cactusImage.size.width;
```

所有关键的准备工作都已就绪，仙人掌终于可以出现在游戏视图中了。因为游戏中的仙人掌是可以被点击的对象，所以它应该是 UIButton 类型。插入一些 frame 变量，将可被插入的仙人掌置于沙丘后面且处于不可见状态。还要向仙人掌对象添加一个调用 cactusHit:方法的动作，我们会在后面讨论这个方法。

```
UIButton *cactus = [[UIButton alloc]
➥initWithFrame:CGRectMake(horizontalLocation,
➥(duneToSpawnBehind.frame.origin.y), cactusWidth, cactusHeight)];

[cactus setImage:cactusImage forState:UIControlStateNormal];

[cactus addTarget:self action:@selector(cactusHit:)
```

```
➥forControlEvents:UIControlEventTouchDown];

[self.view insertSubview:cactus belowSubview:duneToSpawnBehind];
```

现在仙人掌已经做好了，准备用动画让它从沙丘后面弹出来，并且激活计时器告知程序用户是否在两秒内点击了该仙人掌。仙人掌从沙丘后面弹出的动画大概用时四分之一秒，还要计算仙人掌的高度以确保露出的位置正确。一个两秒钟的计时器也会被激活，调用cactusMissed:方法，该方法将在“仙人掌间的相互影响”小节中讨论。

```
[UIView beginAnimations:@"slideInCactus" context:nil];
[UIView setAnimationCurve:UIViewAnimationCurveEaseInOut];
[UIView setAnimationDuration:0.25];

cactus.frame = CGRectMake(horizontalLocation,
➥(duneToSpawnBehind.frame.origin.y)-cactusHeight/2, cactusWidth,
➥cactusHeight);
[UIView commitAnimations];

[self performSelector:@selector(cactusMissed:) withObject:cactus
➥afterDelay:2.0];

[UIView animateWithDuration:0.25f
                      delay:0.f
                    options:UIViewAnimationOptionCurveEaseInOut
                 animations:^{
                     cactus.frame = CGRectMake(horizontalLocation,
                                    (duneToSpawnBehind.frame.origin.y)-
                                    ➥cactusHeight/2.f,
                                    ➥cactusWidth, cactusHeight);
                 } completion:nil];
```

3.1.2 仙人掌间的相互影响

仙人掌弹出后会出现两种可能的结果，分别是用户在两秒钟内点击了仙人掌，或是没有点到。在第 1 种场景下，cactusHit:方法会被调用，作为仙人掌按钮 UIControlEventTouchDown 动作的响应。点到后，仙人掌会快速从屏幕中消失并从 superView 视图中移除该对象。使用 UIViewAnimationOptionsBeginFromCurrentState 参数可以确保作用于该仙人掌之上的所有动画都被取消。这时玩家得分加 1，并调用 displayNewScore:方法更新屏幕上显示的得分，有关更新得分的实现会在本节后面介绍。仙人掌被点击后，关键一步就是弹出下一个仙人掌，实现的方法同 viewDidLoad 中的一样，只不过要加入随机时间延迟，目的是让玩家有时间高兴一下。

```
-(IBAction)cactusHit:(id)sender;
{
    [UIView animateWithDuration:0.1f
                          delay:0.0f
                        options:UIViewAnimationCurveLinear |
                        UIViewAnimationOptionBeginFromCurrentState
```

```
                    animations:^
        {
                [sender setAlpha:0];
        }
        completion:^(BOOL finished)
        {
                [sender removeFromSuperview];
    }];

    score ++;

    [self displayNewScore:score];
    [self performSelector:@selector(spawnCactus) withObject:nil
    ➥ afterDelay:(arc4random()%3) + .5f];
}
```

每个仙人掌弹出两秒钟之后都会调用 cactusMissed:方法，即使某个仙人掌被点击也是一样的。由于不管玩家是否点击了仙人掌都会调用该方法，因此非常有必要检查它的状态。被点击的仙人掌会从 superView 视图中移除。检查对象是否为 nil 并立即返回结果可以避免用户成功点击了仙人掌但却没有加分的情况出现。

同样，开发者不能因为用户暂停游戏而扣分，所以当游戏处于暂停状态时，玩家不能因此失去生命值。如果方法进行到这一步，但是没有任何返回值，就说明用户漏掉了一个仙人掌，需要进行惩罚。由于 cactusHit:方法的存在，游戏仍然需要将漏点击的仙人掌从 superView 中移除并重新启用一个新的计时器。此外，这时候不仅不能增加玩家的得分，还要扣除生命值，需要调用 updateLife 方法更新生命值的显示。

注意

这里介绍的获取暂停状态的方法存在一个有趣的漏洞，如果玩家暂停游戏，暂停之后本应消失的仙人掌会永远存在于屏幕上。虽然我们有办法解决这个漏洞，不过为了程序的简易性，这个较差的用户体验还是暂时留着吧。

```
-(void)cactusMissed:(UIButton *)sender;
{
    if([sender superview] == nil)
    {
        return;
    }

    if(paused)
    {
        return;
    }

    CGRect frame = sender.frame;
    frame.origin.y += sender.frame.size.height;
```

```
    [UIView animateWithDuration:0.1f
                          delay:0.0f
                        options:UIViewAnimationCurveLinear |
                       UIViewAnimationOptionBeginFromCurrentState
                     animations:^
    {
        sender.frame = frame;
    }
        completion:^(BOOL finished)
    {
            [sender removeFromSuperview];
            [self performSelector:@selector(spawnCactus)
            withObject:nil afterDelay:(arc4random()%3) + .5f];

            life--;
            [self updateLife];
    }];
}
```

3.1.3 显示生命值和得分

如果 Whack-a-Cac 游戏对漏掉的仙人掌没有进行惩罚，也就无从记录玩家取得了多少分数，游戏还有什么意思？所以在这个示例游戏中，显示得分和生命值是整个游戏非常重要的元素，这两个方法的调用都是在之前小节介绍的类中实现的。

现在我们把注意力放在 IFCGameViewController.m 文件中的 displayNewScore:方法上。每当得分更新时，都会调用 displayNewScore:方法在游戏中显示新的得分。除了显示得分之外，每当得分到达 10 的倍数但小于等于 50 时，会新弹出一个仙人掌。这个新仙人掌的出现是为了随着游戏的进行增加游戏的难度。

```
-(void)displayNewScore:(CGFloat)updatedScore;
{
    NSInteger scoreInt = score;

    if(scoreInt % 10 == 0 && score <= 50)
    {
        [self spawnCactus];
    }

    scoreLabel.text = [NSString stringWithFormat:@"%06.0f", updatedScore];
}
```

显示生命值同显示玩家得分类似，只不过稍微复杂一点。同显示得分用的文本框不同，玩家生命值是以图片呈现的。在创建 UIImage 表示每个生命值之后，首先需要做的是移除所有存在的生命值图标，只需要用一个简单的 tag 在子视图中进行搜索即可。接下来根据玩家所剩生命值的数量执行一个循环，在游戏视图的右上角显示相应数量的表示生命值的图标。最后游戏还要检查玩家是否还有生命值，如果玩家生命值为 0，会弹出一个 UIAlert 对象提醒玩家游戏结束并显示最终得分。

```
-(void)updateLife
{
    UIImage *lifeImage = [UIImage imageNamed:@"heart.png"];

    for(UIView *view in [self.view subviews])
    {
        if(view.tag == kLifeImageTag)
        {
            [view removeFromSuperview];
        }
    }


    for(int x = 0; x < life; x++)
    {
        UIImageView *lifeImageView = [[UIImageView alloc]
        initWithImage:lifeImage];

        lifeImageView.tag = kLifeImageTag;

        CGRect frame = lifeImageView.frame;
        frame.origin.x = 985 - (x * 30);
        frame.origin.y = 20;
        lifeImageView.frame = frame;

        [self.view addSubview:lifeImageView];
    }

    if(life == 0)
    {
        gameOver = YES;
        UIAlertView *alert = [[UIAlertView alloc]
                initWithTitle:@"Game Over!"
                      message:[NSString stringWithFormat:@"You
                      ➥ scored %0.0f points!", score]
                     delegate:self
            cancelButtonTitle:@"Dismiss"
            otherButtonTitles:nil];

        alert.tag = kGameOverAlert;
        [alert show];
    }
}
```

3.1.4 暂停和恢复

Whack-a-Cac 游戏在屏幕左上角有一个暂停按钮，可以支持玩家暂停和恢复游戏。点击该按钮会调用 pause:动作，这个方法很简单，状态变量 paused 被设置成 YES，同时弹出一个提醒框询问用户退出游戏还是恢复当前游戏。

```
-(IBAction)pause:(id)sender
{
    paused = YES;

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@""
        message:@"Game Paused!"
        delegate:self
        cancelButtonTitle:@"Exit"
        otherButtonTitles:@"Resume", nil];

    alert.tag = kPauseAlert;
    [alert show];
}
```

游戏的结束和暂停都使用 UIAlert 来处理响应。无论是游戏结束事件还是暂停界面上的退出选项，导航栈都返回到菜单界面。如果用户选择恢复游戏，唯一需要做的就是将暂停状态改回 NO。

```
-(void)alertView:(UIAlertView *)alertView
➥clickedButtonAtIndex:(NSInteger)buttonIndex
{
    if(alertView.tag == kGameOverAlert)
    {
        [self.navigationController popViewControllerAnimated:YES];
    }

    else if (alertView.tag == kPauseAlert)
    {
        if(buttonIndex == 0)
        {
            [self.navigationController popViewControllerAnimated:YES];
        }
        else
        {
            paused = NO;
        }
    }
}
```

3.1.5 有关 Whack-a-Cac 游戏的最后问题

现在你应该熟悉并对 Whack-a-Cac 游戏的功能有一定的信心了，不过该程序源代码中的一些其他的清除方法还是值得继续深入研究的。下面将介绍如何向 Whack-a-Cac 游戏中添加排行榜。

3.2 iTunes Connect

排行榜的数据保存在苹果公司的 Game Center 服务器上。要配置应用的排行榜，首先需要在 iTunes Connect 网站的 Game Center 部分对其进行配置。使用 iTunes Connect Portal((http://

itunesconnect.apple.com)创建一个新的准备发售的应用。如果你已经有一个应用，也可以用它替换。填完基本信息后，当前页面应该同图 3-2 类似。

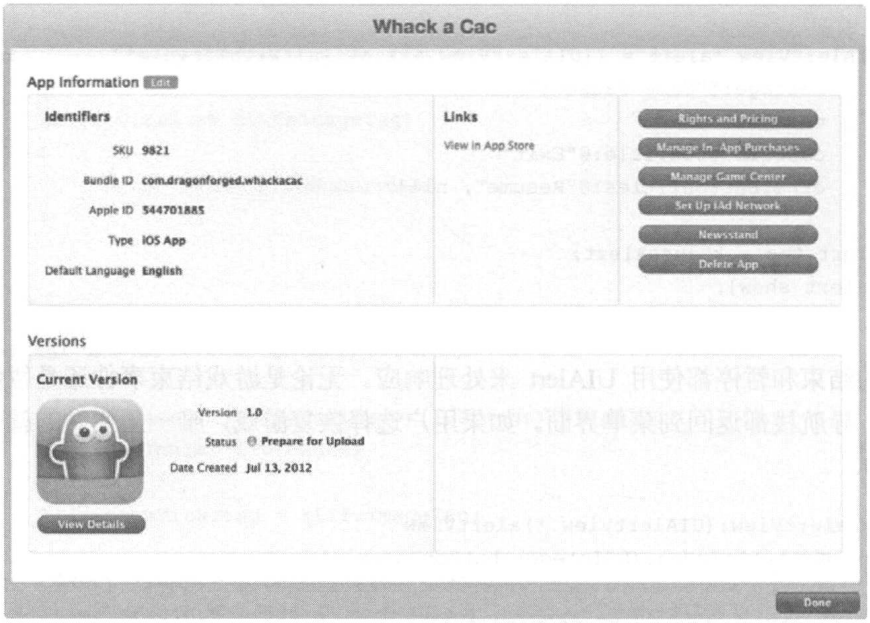


图 3-2 iTunes Connect 中的基本应用页面

提醒

从在 iTunes Connect 中创建一个新的应用开始，一共有 90 天的时间可以提交程序审核，这样做是为了避免应用名被占用。虽然可以创建一个假的测试程序关联 Game Center 进行测试，不过切记上面提到的这个时间限制很重要，一定要牢记在心。

现在查看页面的右上角，找到一个名为 Manage Game Center 的按钮，在此可以为应用配置 Game Center 行为。在 Game Center 配置页面，首先你会注意到一个由滑动条控制的 Game Center 开关，如图 3-3 所示。此外，还有一个选项用于在多个应用间共享排行榜，比如在免费和付费版本中共享。要分享排行榜，需要创建一个引用名，之后通过 iTunes Connect 账户在多个应用之间实现共享。点击 Move to Group 选项即完成排行榜的创建和配置。

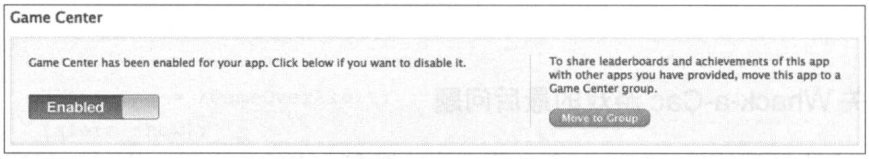


图 3-3 在应用中启用 Game Center 行为

为应用启用 Game Center 功能之后，需要设置第一个排行榜，如图 3-4 所示。很重要的一点是在应用通过审核并在 App Store 上架之后，就不能删除配置好的排行榜了。苹果公司近期提供了一个选项，可以让开发者删除排行榜上的测试数据，不过我们还是建议开发者在程序上架之前将测试过程中出现的测试数据删除。

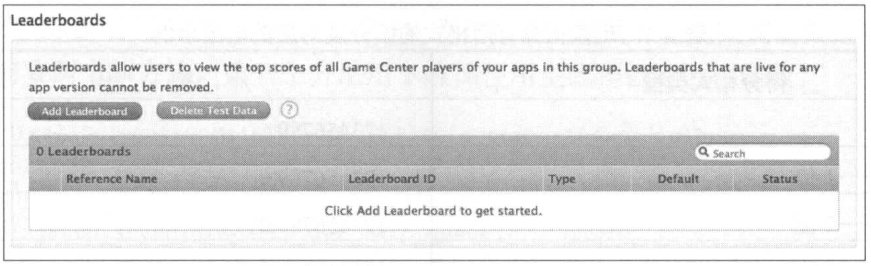


图 3-4 设置一个新的排行榜

从页面中选择 Add Leaderboard，将会出现两个选项。第一个选项是 Single Leaderboard，表示一个独立的排行榜，类似 Whack-a-Cac 游戏中使用的那样。Single Leaderboard 会为应用保存得分或游戏模式。第二个选项是 Combined Leaderboard，它可以让开发者将两个或更多个排行榜组合在一起，从而创建一个最终的得分排行榜。比如游戏有多个等级、每个等级都有自己的排行榜，这时就可以将所有排行榜组合起来，创建一个跨等级的总榜。根据本章之前设定的目标，在此只介绍 Single Leaderboard。

注意

在 iOS 7 版本中，苹果公司对每款应用中用到的排行榜的数量限制为不能超过 500 个，这比之前 25 个的限制提升不少。

当设置一个排行榜时，需要在配置页面上填写许多内容，如图 3-5 所示。首先需要输入的是 Leaderboard Reference Name，这个名字可以让程序在 iTunes Connect 中快速定位和识别排行榜。Leaderboard ID 属性用于在实际项目中查找排行榜，苹果公司建议使用反向 DNS 命名方案，Whack-a-Cac 使用的 Leaderboard ID 为 com.dragonforged.whackacac.leaderboard。如果用的是自己的应用，注意在下面的示例代码中用正确的内容进行替换。

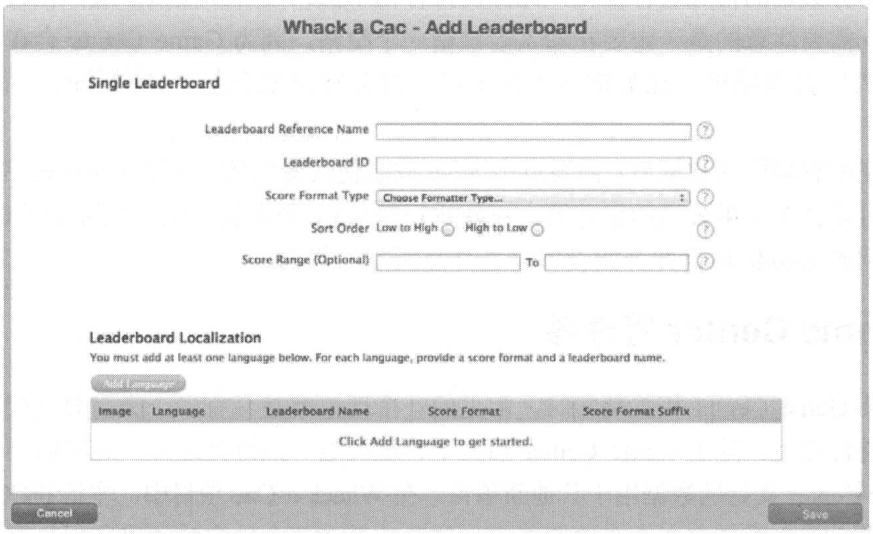


图 3-5 在 iTunes Connect 中为 Whack-a-Cac 配置标准的单独排行榜

苹果公司针对排行榜列表中得分数据的格式给出了一些默认值，表 3-1 给出了这些格式的示例。

表 3-1 玩家得分可用格式的细分及对应的输出示例

得分格式类型	输 出
整数	123456789
小数点——1 位	123456789.0
小数点——2 位	123456789.01
小数点——3 位	123456789.012
用时——分钟	3:20
用时——秒	3:20:59
用时——百分之一秒	3:20:58.99
金钱——整数	$497,776
金钱——两位小数	$497,766.98

注意

即使用到的得分表示方式同表 3-1 中的格式不一样也没有关系，可以使用自定义的排行榜呈现方式。如何获取原始的得分数值将在后面小节“深入讨论排行榜”中介绍。

排序选项用于控制在 Game Center 的得分列表中是显示最高分还是最低分。此外还可以指定一个分数区域，自动将由 Game Center 返回的数值中不满足的数据(高于上限或低于下限)丢弃。

创建新排行榜的最后一步是添加本地化信息，为应用使用的主语言提供最小化的本地数据支持显然是必要的，当然还可以为需要的其他语言提供支持。除了本地化名称之外，还可以选择对得分数据的格式进行优化，比如在排行榜中加入图片和输入后缀。在为得分数值添加后缀时，需要注意的是一定要在输入前预留一个空格，因为 Game Center 会在分数后面直接把后缀内容打印输出。比如键入“Points”，得分输出就会是“123Points”而不是“123 Points”。

输入完所有需要的数据后，需要点击 Save 按钮使改变生效。即使在保存之后，也还需要大约几小时才会在苹果公司的服务器上更新排行榜信息。现在我们已经配置好了 Game Center 排行榜，回到 Xcode 项目并开始设置与之交互所需的代码。

3.3 Game Center 管理器

当使用 Game Center 相关功能时，常见的一个现象就是会有多个类需要直接同一个共享的管理类进行交互。除了 Game Center 独立于管理类这一点好处之外，这样做还可以很容易地将 Game Center 支持的数据用于其他新项目。在 Whack-a-Cac 项目中，注意 IFCGameCenterManager 类，首先看到它由一个单例形成，这意味着在每个给定时间段应用只有一个实例。第一个需要完成的任务就是创建 Game Center 管理器的基础代码，Game Center 管理器用于直接处理所有和 Game Center 相关的操作，而不管这些调用是否能成功使用原型发送委托信息。

因为 Game Center 调用不是后台线程安全的，管理器需要将所有委托回调函数定向到主线程。为此，就需要两个新方法，第一个方法用于确保使用主线程创建回调函数。

```
-(void)callDelegateOnMainThread:(SEL)selector withArg:(id)arg
➥ error:(NSError *)error
{
    dispatch_async(dispatch_get_main_queue(), ^(void)
    {
        [self callDelegate:selector withArg:arg error:error];
    });
}
```

callDelegateOnMainThread:方法会向 callDelegate:方法传递所有参数和错误信息。callDelegate 方法需要完成的第一件事就是确保调用来自主线程，如果确实来自主线程，则不会直接调用。由于 callDelegate:方法不设置委托就无法正确实现相应的功能，因此接下来的检查就是要确保有一个委托函数。此时就可以确定我们在主线程上且具有一个委托函数，使用 respondsToSelector:方法可以测试委托函数是否已正确实现，如果没有，就会出现如下所示的帮助日志：

```
2012-07-28 17:12:41.816 WhackACac[10121:c07] Unable to find delegate
➥method 'gameCenterLoggedIn:' in class ICFViewController
```

所有的安全性和健全性测试都就绪后，就会根据传递的参数和错误信息调用委托函数。在此给出一个基本的委托回调系统，现在就开始实现实际的 Game Center 功能。

```
-(void)callDelegate:(SEL)selector withArg:(id)arg
➥error:(NSError*)error
{
    assert([NSThread isMainThread]);

    if(delegate == nil)
    {
        NSLog(@"Game Center Manager Delegate has not been set");
        return;
    }

    if([delegate respondsToSelector:selector])
    {
        if(arg != NULL)
        {
            [delegate performSelector:selector withObject:arg withObject:error];
        }

        else
        {
            [delegate performSelector:selector withObject:error];
        }
    }
```

```
    else
    {
        NSLog(@"Unable to find delegate method '%s' in class
        ➥%@", sel_getName(selector), [delegate class]);
    }
}
```

3.4 认证

Game Center 是一个需要认证的服务，也就是说，当用户没有登录认证时无法成功使用任何功能。所以基于上面的讨论，你一定要知道在处理排行榜相关代码前要进行用户身份认证。Game Center 的认证过程大部分都由 iOS 帮助实现，下面的代码将呈现一个 UIAlert 对话框，让用户登录 Game Center 平台或者创建一个新的 Game Center 账户。

注意

不要忘记在使用 Game Center 时添加 GameKit.framework 框架和导入 GameKit/GameKit.h 头文件。

```
-(void)authenticateLocalUser
{
    if([GKLocalPlayer localPlayer].authenticated == NO)
    {
        [[GKLocalPlayer localPlayer]
        ➥authenticateWithCompletionHandler:^(NSError *error)
        {
            if(error != nil)
            {
                NSLog(@"An error occurred: %@", [error
                ➥localizedDescription]);

                return;
            }

            [self callDelegateOnMainThread:
            ➥@selector(gameCenterLoggedIn:) withArg:NULL
            ➥error:error];
        }];
    }
}
```

若出现错误，则会在控制台输出日志。如果登录成功，会调用相应的委托函数。可以查看 ICFGameCenterManager.h 文件以了解这些委托方法的设置情况。

3.4.1 常见的认证错误

有些常见的错误案例可以帮助你解决一些有关认证的问题。下面是一个修改版的 authenticateLocalUser 函数，其中加入了错误处理机制。

注意

如果收到提醒说游戏无法被 Game Center 识别，请检查应用的 bundle ID 和之前在 iTunes Connect 中配置的名字是否一样。一个新的应用一般需要几个小时才能被 Game Center 完全识别。对于很多的 Game Center 问题，稍微等待一段时间再重新连接服务器，一般都可以得到解决。

```
-(void)authenticateLocalUser
{
    if([GKLocalPlayer localPlayer].authenticated == NO)
    {
        [[GKLocalPlayer localPlayer]
        ➥authenticateWithCompletionHandler:^(NSError *error)
        {
            if(error != nil)
            {
                if([error code] == GKErrorNotSupported)
                {
                    UIAlertView *alert = [[UIAlertView
                    ➥alloc] initWithTitle:@"Error"
                    ➥message:@"This device does not support
                    ➥Game Center" delegate:nil
                    ➥cancelButtonTitle:@"Dismiss"
                    ➥otherButtonTitles:nil];

                    [alert show];
                }

                else if([error code] == GKErrorCancelled)
                {
                    UIAlertView *alert = [[UIAlertView
                    ➥alloc] initWithTitle:@"Error"
                    ➥message:@"This device has failed login
                    ➥too many times from the app; you will
                    ➥need to log in from the Game
                    ➥Center.app" delegate:nil
                    ➥cancelButtonTitle:@"Dismiss"
                    ➥otherButtonTitles:nil];

                    [alert show];
                }

                else
                {
                    UIAlertView *alert = [[UIAlertView
                    ➥alloc] initWithTitle:@"Error" message:
                    ➥[error localizedDescription]
                    ➥delegate:nil
                    ➥cancelButtonTitle:@"Dismiss"
                    ➥otherButtonTitles:nil];
```

```
                [alert show];
            }

            return;
        }

        [self callDelegateOnMainThread:
        ➥@selector(gameCenterLoggedIn:) withArg:NULL
        ➥error:error];
        }];
    }
}
```

在上面的例子中，给出了 3 种错误的处理办法。第一个错误是设备不明原因而不支持 Game Center，这种情况会向用户弹出一个提醒框，通知现在无法访问 Game Center。第二个错误在实际上架的应用中很少见，不过在调试过程中却非常棘手，如果应用连续 3 次登录 Game Center 失败，苹果公司就会拒绝登入请求。这种情况下，必须从 Game Center 应用登录。第三个错误的处理方法是对于所有其他额外的错误都要明确通知用户。

成功登录之后，用户会收到一条从 Game Center 发来的登录消息。这条消息同时也会通知你当前是否处于沙盒环境中，如图 3-6 所示。

图 3-6 在 Whack-a-Cac 游戏中成功登录 Game Center

注意

所有未上架的应用都处于沙盒环境中。在应用从 App Store 下载之后，就是正常的生产环境了。在第一次上架 App Store 前是无法脱离沙盒环境进行测试的。

3.4.2 iOS 6 和新的认证系统

虽然之前的认证方法在 iOS 8 版本中仍然适用，但苹果公司还是为那些不支持 iOS 5 和之前版本的应用推出了一种简单的方法，用于处理认证相关的问题。

新方法中使用了 authenticateHandler block。错误捕捉的方式还和之前的示例一样，不过现在 Game Center 会将一个 viewController 传回程序。这种情况下，viewController 类的 authenticateHandler block 参数不能是 nil，因为需要向用户显示 viewController。

新的 authenticateHandler 首次设置成功后，应用会自动认证。此外，当应用从后台返回前台运行时还会进行自动重新认证。如果需要手动调用 authenticate，可以使用 authenticate 方法来实现。

```
-(void)authenticateLocalUseriOSSix
{
    if([GKLocalPlayer localPlayer].authenticateHandler == nil)
    {
        [[GKLocalPlayer localPlayer]
        ➥setAuthenticateHandler:^(UIViewController
        ➥*viewController, NSError *error)
       {
          if(error != nil)
          {
             if([error code] == GKErrorNotSupported)
             {
                UIAlertView *alert = [[UIAlertView alloc]
                ➥initWithTitle:@"Error" message:@"This
                ➥device does not support Game Center"
                ➥delegate:nil cancelButtonTitle:@"Dismiss"
                ➥otherButtonTitles:nil];

                [alert show];
             }

             else if([error code] == GKErrorCancelled)
             {
                UIAlertView *alert = [[UIAlertView alloc]
                ➥initWithTitle:@"Error" message:@"This
                ➥device has failed login too many times from
                ➥the app; you will need to log in from the
                ➥Game Center.app" delegate:nil
                ➥cancelButtonTitle:@"Dismiss"
                ➥otherButtonTitles:nil];


                [alert show];
             }

             else
             {
                UIAlertView *alert = [[UIAlertView alloc]
                ➥initWithTitle:@"Error" message:[error
                ➥localizedDescription] delegate:nil
                ➥cancelButtonTitle:@"Dismiss"
                ➥otherButtonTitles:nil];
```

```
                [alert show];
            }
        }

        else
        {
            if(viewController != nil)
            {
                [(UIViewController *)delegate
                ➥presentViewController:viewController
                ➥animated:YES completion:nil];
            }
        }
    }];
}

else
{
    [[GKLocalPlayer localPlayer] authenticate];
}
}
```

3.5 提交得分

通过 Game Center 认证之后，就需要准备提交得分了。IFCGameCenterManager 类中有方法 reportScore:forCategory，该方法会根据你在 iTunes Connect 上配置的 Leaderboard ID 上传最新的得分。需要为上传的得分数据创建一个新的 GKScore 对象，这个对象用于保存各种数值，比如得分、playerID、日期、排名、formattedValue、类别和上下文等。

当提交一个新的得分时，大部分数据都是自动生成的，只有分数和类别两项信息需要填写。内容信息是可选的，由一个任意 64 位的无符号整型数值(int64_t)表示。上下文用于保存有关得分的额外信息，比如游戏设置或某些重要的得分标志位，可以使用 context 属性获取这些信息。日期、playerID、formattedValue 和排名都是只读的，当创建或检索 GKScore 对象时由系统自动生成。

注意

在 iTunes Connect 中设置排行榜时支持默认的类别，如果将得分提交给一个默认的排行榜，类别参数可以不填，不过最好还是为程序设置一个类别参数，以免出现难以追踪的错误。

为排行榜指定类别并创建新的 GKScore 对象之后，就可以分配原始得分值。分值为整数或小数时，数字就是相应的得分。如果显示内容为消耗的时间，提交的数值应该以秒或者带有小数部分的更精确的时间来显示。

成功提交得分数据后，会在 GameCenterManager 委托中调用 gameCenterScoreReported: 方法。下一节“向 Whack-a-Cac 中添加得分”将详述该内容。

```
-(void)reportScore:(int64_t)score forCategory:(NSString*)category
```

```
{
    GKScore *scoreReporter = [[GKScore alloc]
    initWithCategory:category];

    scoreReporter.value = score;

    [scoreReporter reportScoreWithCompletionHandler:^(NSError *error)
    {
        if(error != nil)
        {
            NSData* savedScoreData = [NSKeyedArchiver
            ➥archivedDataWithRootObject:scoreReporter];

            [self storeScoreForLater:savedScoreData];
        }

        [self callDelegateOnMainThread:@selector
        ➥(gameCenterScoreReported:) withArg:NULL error:
        ➥error];
    }];
}
```

查看错误处理代码段 reportScoreWithCompletionHandler 很重要。如果得分没有成功提交到 Game Center，则需要尝试重新提交。由于程序问题或网络问题导致玩家获得的高分无法保存将非常令人气愤。在之前的示例代码中，获取得分失败时，NSKeyedArchiver 会创建一个 NSData 类型对象的副本并将其传给 storeScoreForLater:方法。关键点在于 GKScore 对象本身也要使用该值而不是只在得分参数中使用。如果得分是正确的，则 Game Center 按照日期对得分进行排序，由于日期是在 GKScore 对象创建时自动生成的，因此唯一确保用户信息不丢失的方法就是将整个 GKScore 对象归档。

保存得分数据时，示例程序使用了 NSUserDefaults，该数据也可以很容易地保存为 Core Data 类型或者其他存储系统要求的类型。得分数据保存好之后，重要的一点是当程序有需求时应该再次发送数据，一个较好的时间点就是在 GameCenter 成功认证之后。

```
-(void)storeScoreForLater:(NSData *)scoreData;
{
    NSMutableArray *savedScoresArray = [[NSMutableArray alloc]
    ➥initWithArray:[[NSUserDefaults standardUserDefaults]
    objectForKey:@"savedScores"]];

    [savedScoresArray addObject:scoreData];

    [[NSUserDefaults standardUserDefaults]
    ➥setObject:savedScoresArray forKey:@"savedScores"];
}
```

重新提交已经保存的得分数据并不比第一次提交得分数据难。首先需要从 NSUserDefaults 中获取得分，由于对象被保存为 NSData 类型，因此需要将数据转换为 GKScore

对象。同样，一定要注意捕捉提交失败事件并在之后重新进行提交。

```
-(void)submitAllSavedScores
{
    NSMutableArray *savedScoreArray = [[NSMutableArray alloc]
    ➥initWithArray:[[NSUserDefaults standardUserDefaults]
    ➥objectForKey:@"savedScores"]];

    [[NSUserDefaults standardUserDefaults] removeObjectForKey:
    ➥@"savedScores"];

    for(NSData *scoreData in savedScoreArray)
    {
        GKScore *scoreReporter = [NSKeyedUnarchiver
        ➥unarchiveObjectWithData:scoreData];

        [scoreReporter reportScoreWithCompletionHandler:
        ➥^(NSError *error)
        {
            if(error != nil)
            {
                NSData* savedScoreData = [NSKeyedArchiver
                ➥archivedDataWithRootObject:scoreReporter];

                [self storeScoreForLater:savedScoreData];
            }

              else
              {
                  NSLog(@"Successfully submitted scores that
                  ➥were pending submission");

                  [self callDelegateOnMainThread:
                  ➥@selector(gameCenterScroreReported:)
                  ➥withArg:NULL error:error];
              }
        }];
    }
}
```

提示

如果得分提交失败，最好提醒用户：应用会在稍后重新提交数据；否则用户可能会误以为程序没有将得分数据保存。

3.5.1 向 Whack-a-Cac 中添加得分

上一节介绍了 Game Center Manager 有关添加得分的功能，本节将在示例程序 Whack-a-Cac 中进行实际操作。在实现该功能之前，Game Center 必须对用户身份进行认证并指定一个委托函数，以下修改 IFCViewController.m 文件中的 viewDidLoad 方法：

```
-(void)viewDidLoad
{
    [super viewDidLoad];

    [[ICFGameCenterManager sharedManager] setDelegate:self];
    [[ICFGameCenterManager sharedManager] authenticateLocalUser];
}
```

IFCViewController 还需要响应 GameCenterManagerDelegate 委托函数，第一个需要处理的委托方法是 gameCenterLoggedIn:。因为 GameCenterManager 还要处理有关错误通知的所有 UIAlert，所以这里会把全部的错误信息打印输出以便后期进行调试。

```
-(void)gameCenterLoggedIn:(NSError*)error
{
    if(error != nil)
    {
        NSLog(@"An error occurred trying to log into Game
        ➥Center: %@", [error localizedDescription]);
    }

    else
    {
       NSLog(@"Successfully logged into Game Center!");
    }
}
```

在用户决定开始一局新的游戏后，更新 IFCGameViewController 类中的 GameCenterManager 委托是很重要的一步。在 Whack-a-Cac 中，委托函数一般都设置为最前端的视图，这样可以方便处理用户反馈和错误。只需要向 IFCGameViewController 的 viewDidLoad:方法中添加如下一行代码即可，不要忘记像 GameCenterManagerDelegate 一样对其进行声明。

```
[[ICFGameCenterManager sharedManager] setDelegate:self];
```

游戏将会在两个场景下提交分数，分别是当用户结束一局游戏时和从暂停菜单退出时。使用 IFCGameCenterManager 可以很容易地提交得分。将下面一行代码添加到两个方法中，其中一个是当生命值变为 0 时触发的 updateLife 方法，另一个是暂停弹出框 UIAlert 对象的退出按钮动作方法。

```
[[ICFGameCenterManager sharedManager]reportScore:
➥(int64_t)scoreforCategory:
➥ @"com.dragonforged.whackacac.leaderboard"];
```

注意

为排行榜设置的类别 ID 可能会和这些示例程序中使用的不同，要确保程序使用的 ID 与 iTunes Connect 中设置的一样。

虽然 GameCenterManager 的 reportScore:方法会处理提交得分和所有的错误修复问题，不过添加委托方法 gameCenterScoreReported:来观察潜在的错误和状态也非常重要。

```
-(void)gameCenterScoreReported:(NSError *)error;
{
    if(error != nil)
    {
        NSLog(@"An error occurred trying to report a score to
        ➥Game Center: %@", [error localizedDescription]);
    }

    else
    {
        NSLog(@"Successfully submitted score");
    }
}
```

注意

只有当结束时才提交得分，每当得分更新就将数据提交给 Game Center 会导致糟糕的用户体验。

当用户正在退出程序时，GameCenterManager 的委托函数会消失，不过提交得分的网络操作仍然在发生。将 IFCViewController 的 GameCenterManagerDelegate 重新设置为 SELF 非常重要，同时还要实现 gameCenterScoreReported:委托函数。

3.5.2 展示排行榜

如果用户无法查看他们在游戏中创新高的得分，那么高分的意义就不大了。本节将介绍如何使用苹果公司内置的视图控制器来展示排行榜。排行榜的视图管理器在 iOS 6 版本时有了长足进步，并沿用至 iOS 8，如图 3-7 所示。在前几个版本的 iOS 系统中，排行榜和成就榜是由两个独立的视图管理器处理的，现在将这两个功能整合了。此外，加入了 Game Center 挑战功能和类似 Facebook 展示的功能。

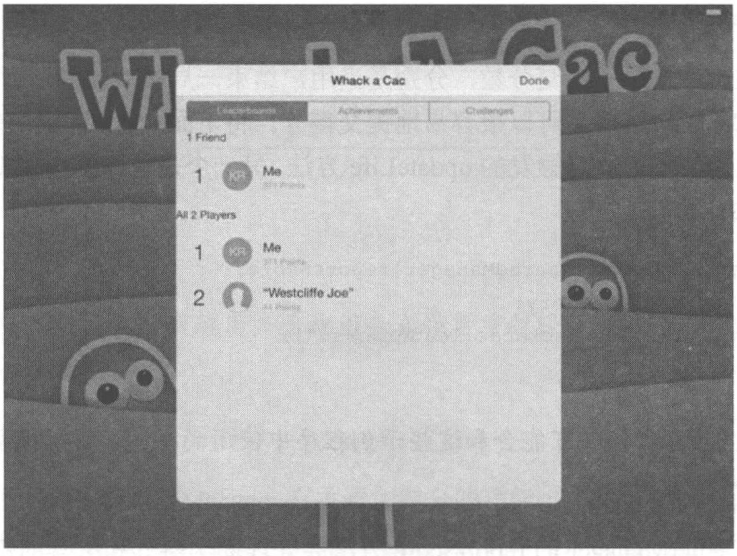


图 3-7 iOS 6 版本中 GKLeaderboardViewController 的主要改进和变化

ICFViewController 中的方法 leaderboards:用于处理排行榜视图控制器的展示。一个新的 GKLeaderboardViewController 创建好之后，有许多属性需要设置。首先设置类别属性，这里用到的类别和提交得分及展示排行榜视图的一样。还可以提供 timeScope 属性，以便让用户选择正确的默认页面，如图 3-7 所示。在下面的示例中，时间范围为所有时间，此外必须提供所需的 leaderboardDelegate 委托函数，用于删除排行榜模型。

```
-(IBAction)leaderboards:(id)sender
{
    GKLeaderboardViewController *leaderboardViewController =
    [[GKLeaderboardViewController alloc] init];

    if(leaderboardViewController == nil)
    {
        NSLog(@"Unable to create leaderboard view controller");
        return;
    }

     leaderboardViewController.category =
     ➥@"com.dragonforged.whackacac.leaderboard";

     leaderboardViewController.timeScope =
     ➥GKLeaderboardTimeScopeAllTime;

     leaderboardViewController.leaderboardDelegate = self;

    [self presentViewController:leaderboardViewController
    ➥animated:YES completion:nil];

    [leaderboardViewController release];
}
```

要让 GKLeaderboardViewController 完全实现我们期望的功能，这里还必须提供一个委托方法。调用该方法时，需要移除视图控制器，如下所示：

```
-(void)leaderboardViewControllerDidFinish:(GKLeaderboardViewController
➥*) viewController
{
    [self dismissModalViewControllerAnimated:YES completion:nil];
}
```

视图控制器还会用到一个新的委托方法，如下所示：

```
-(void)gameCenterViewControllerDidFinish:(GKGameCenterViewController
➥*)gameCenterViewController
{
    [self dismissModalViewControllerAnimated:YES completion:nil];
}
```

也可以使用排行榜的原始数据并创建自定义排行榜，在后面的小节“深入讨论排行榜”

中将详细介绍。

3.5.3 得分挑战

Game Center Challenges 机制可以让用户用自己游戏中的高得分或成就向 Game Center 好友发起挑战。这一功能为用户在社交网络中传播自己的游戏程序提供了一条非常好的途径。有关 Challenges 功能的实现由 Game Center 框架的 GameCenterViewController 类完成，如上面的示例和图 3-8 所示。不过仍可通过代码实现这一功能，对 GKScore 对象调用 issueChallengeToPlayers:withMessage:方法将会初始化挑战功能。当用户成功战胜一个挑战任务时，系统又会向该挑战发起者再次发出一条挑战信息。

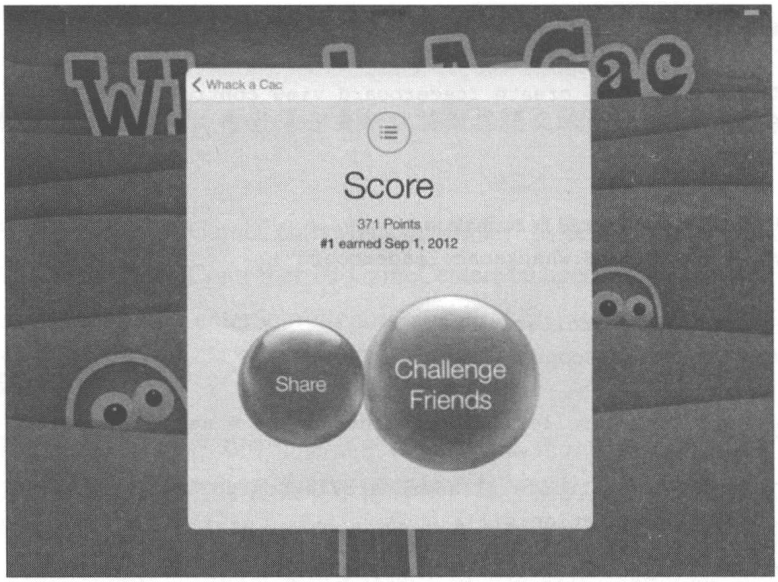


图 3-8 使用 Game Center 内置的挑战机制向朋友的得分发起挑战

```
[(GKScore *)score issueChallengeToPlayers:(NSArray *)players
➥message:@"Can you beat me?" ];
```

对于认证用户，可以获取一个包含所有待挑战的 GKChallenge 对象的数组，具体实现如下所示：

```
[GKChallenge loadReceivedChallengesWithCompletionHandler:^(NSArray
➥*challenges, NSError *error)
{
    if(error != nil)
    {
        NSLog(@"An error occurred:%@", [error localizedDescription]);
    }

     else
     {
         NSLog(@"Challenges:%@", challenges);
     }
}];
```

挑战所处的状态有以下4种：无效的、待挑战的、完成的和拒绝的。

```
if(challenge.state == GKChallengeStateCompleted)
   NSLog(@"Challenge Completed");
```

最后，对象调用decline就可以拒绝传入的挑战请求，如下所示：

```
[challenge decline];
```

挑战机制是用户帮助你开拓市场的好机会，如果用户向一名未安装应用的用户发起挑战，该用户很可能要去购买你的软件。通过Game Center默认的GUI就可以帮助你不费吹灰之力实现挑战功能，并且使用之前我们给出的例子可以很容易地将挑战功能添加到你的应用中。

注意

Whack-a-Cac没有实现或演示基于代码的挑战功能。

3.5.4 深入讨论排行榜

本章的重点是讲述使用苹果公司的标准Game Center GUI实现排行榜功能，不过完全可以在应用中实现自定义的排行榜系统。本节给出一个简单的使用原始GKScore数据实现排行榜的方法，同时还可以从Game Center服务器获取具体的数据。

将下面的方法添加到ICFGameCenterManager类，这个方法需要提供4个不同的参数。第一个是category——iTunes Connect中为有关排行榜设置的leaderboard ID。其次是withPlayerScore:——接受GKLeaderboardPlayerScopeGlobal或GKLeaderboardPlayerScope-FriendsOnly参数。TimeScope的值可以是今天、本周或所有时间段。最后一个参数是range，通过该参数可以为接收的得分设置一个特定的范围，比如NSMakeRange(1, 50)就是显示前50个得分。

```
-(void)retrieveScoresForCategory:(NSString *)category
➥withPlayerScope:(GKLeaderboardPlayerScope)playerScope
➥timeScope:(GKLeaderboardTimeScope)timeScope
➥withRange:(NSRange)range
{
   GKLeaderboard *leaderboardRequest = [[GKLeaderboard alloc] init];

   leaderboardRequest.playerScope = playerScope;
   leaderboardRequest.timeScope = timeScope;
   leaderboardRequest.range = range;
   leaderboardRequest.category = category;

   [leaderboardRequest loadScoresWithCompletionHandler:^(NSArray
   ➥*scores,NSError *error)
   {
      [self callDelegateOnMainThread:@selector
      (scoreDataUpdated:error:) withArg:scores error:error];
   }];
}
```

对于该请求还有一个新的相关委托回调，即 scoreDataUpdated:error:。

```
-(void)scoreDataUpdated:(NSArray *)scores error:(NSError *)error
{
    if(error != nil)
    {
        NSLog(@"An error occurred: %@", [error localizedDescription]);
    }
    else
    {
        NSLog(@"The following scores were retrieved: %@", scores);
    }
}
```

如果将这个示例用于 Whack-a-Cac，应该如下所示：

```
-(void)fetchScore
{
    [[ICFGameCenterManager sharedManager]
    retrieveScoresForCategory:
    ➥@"com.dragonforged.whackacac.leaderboard"
    ➥withPlayerScope:GKLeaderboardPlayerScopeGlobal
    ➥timeScope:GKLeaderboardTimeScopeAllTime
    ➥withRange:NSMakeRange(1, 50)];
}
```

委托方法的输出如下所示：

```
2012-07-29 14:38:03.874 WhackACac[14437:c07] The following scores were
➥retrieved: (

"<GKScore: 0x83c5010>player:G:94768768 rank:1 date:2012-07-28 23:54:19
➥+0000 value:201 formattedValue:201 Points context:0x0"
)
```

提示

要为带有 GKScore 参数的 GKPlayer 对象找到 displayName，可以使用[(GKPlayer *)player displayName]。不要忘记将该数据读入缓存，因为需要进行网络调用才可以获取它们。

3.6 小结

当用户向游戏或应用添加社交元素时，Game Center 排行榜是一种非常容易且有趣的途径。用户喜欢比赛，使用 Game Center Challenge 挑战机制可以很容易地在用户之间分享他们喜欢的应用。本章介绍了如何完全将 GameCenter 排行榜整合到游戏或应用中。现在你应该对 Game Center 的认证和得分提交系统、错误修复系统有了较深入的了解。第 4 章继续将展开讨论 Game Center 框架的功能，向 Whack-a-Cac 游戏中加入社交成就系统。

第4章

成就系统

同排行榜一样，成就系统很快就成为现今游戏里最重要的组成部分之一。虽然成就系统的历史不像排行榜那样在游戏界渊源已久，但现在普遍认为对于一款游戏社交化的成功而言，成就系统更加重要。

简言之，成就系统就是解锁需要完成的任务，虽然对于游戏的完整性不是必要的，但它可以增添游戏的竞技性。通常成就系统作为额外的任务或扩展游戏，比如完成游戏中的困难模式、探索更多区域或收集特定的物品等。Game Center 的一个核心功能就是成就系统让你在应用中加入自己的成就系统比之前更容易。

同大部分其他章节不同，本章同第 3 章“排行榜”共享同一个示例程序 Whack-a-Cac。虽然在学习本章之前不一定要完成上一章的内容，不过其中有很多实例是重复使用的。所以从程序环境和类的继承关系考虑，建议阅读第 3 章的以下小节：“示例程序”、“iTunes Connect”、“Game Center Manager”和“认证”。这些小节作为示例程序的背景知识对理解本章内容非常重要，同样有关 Game Center 的基本设置和本地用户认证也非常重要。本章的程序也是在第 3 章中 Game Center Manager 的代码基础上扩展而来。

4.1 iTunes Connect

在用 Xcode 编写成就系统的代码之前，先要访问 iTunes Connect 网站(http://itunesconnect.apple.com)，对成就系统相关内容进行设置。关于如何在 iTunes Connect 中使用 Game Center，可以参见第 3 章的“iTunes Connect”小节。

当进入 iTunes Connec 页面的 Manage Game Center 部分时，共有两个配置项，一个是对排行榜进行设置，另一个就是设置成就系统。在示例程序中，Whack-a-Cac 共有 6 个用于显示的成就。

要创建一个新的成就，点击 Add Achievement 按钮，如图 4-1 所示。

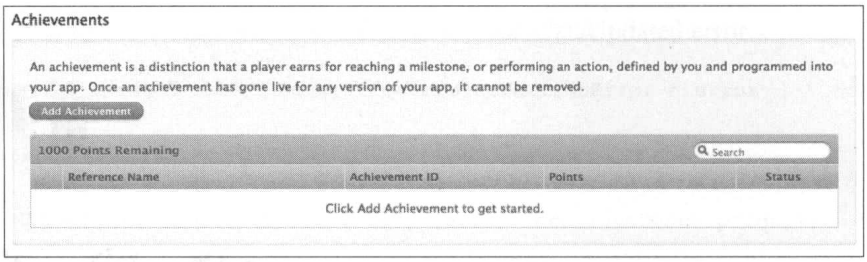


图 4-1 iTunes Connect 中还未添加任何成就的 Achievements 页面视图

同第 3 章“排行榜”类似，设置一个新的成就需要填写许多内容，如图 4-2 所示。Achievement Reference Name 是 iTunes Connect 中成就的引用名称，其在该页面之外都是不可见的。相反，Achievement ID 是作为程序代码和成就系统进行交互的桥梁。苹果公司建议使用反向 DNS 方式对 Achievement ID 进行命名，比如 com.dragonforged.whackacac.100whacks。

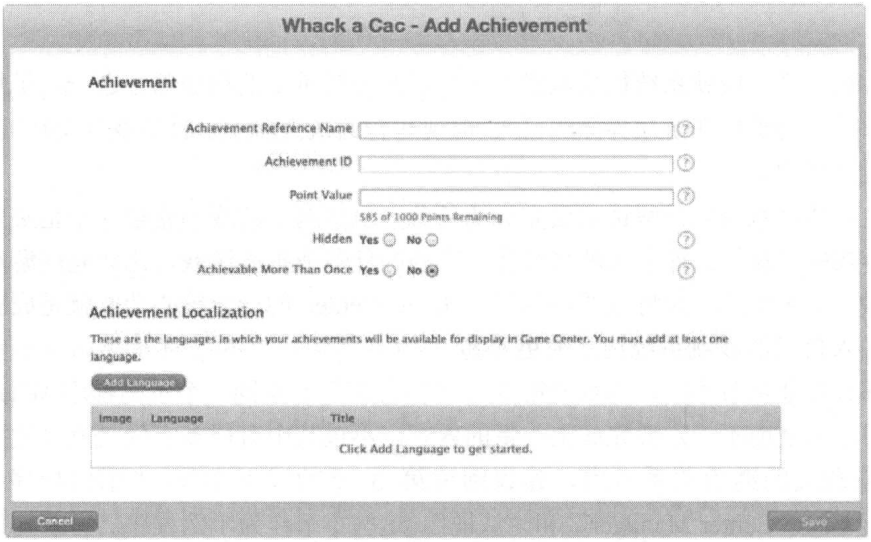


图 4-2 在 iTunes Connect 中添加新的成就

在 Game Center 中，每个成就的 Point Value 属性应该是唯一的，成就可以被指定为从 1 到 100 的值，每个应用最大的成就可以为 1000 points。points 可以用于表示成就获得的难度或成就值。成就的 points 并不是必需的，也不必一定用到 1000。

同样 Hidden 属性也必须唯一，表示在用户最终获得该成就前是否隐藏该成就，或者随着用户获得成就的进展而展示。Achievable More Than Once 设置选项支持用户对自己已获得的成就接受来自 Game Center 朋友的挑战。

同排行榜一样，成就系统也至少需要设置本地化描述。此信息包含 4 个属性，标题出现在成就描述之上，每个成就还包含两个描述，一个在锁定设备时显示，另一个在成就完成时显示。此外，还可以为每个成就提供一张图片，图片尺寸至少为 512×512。对于这些内容的布局可以查看图 4-3。

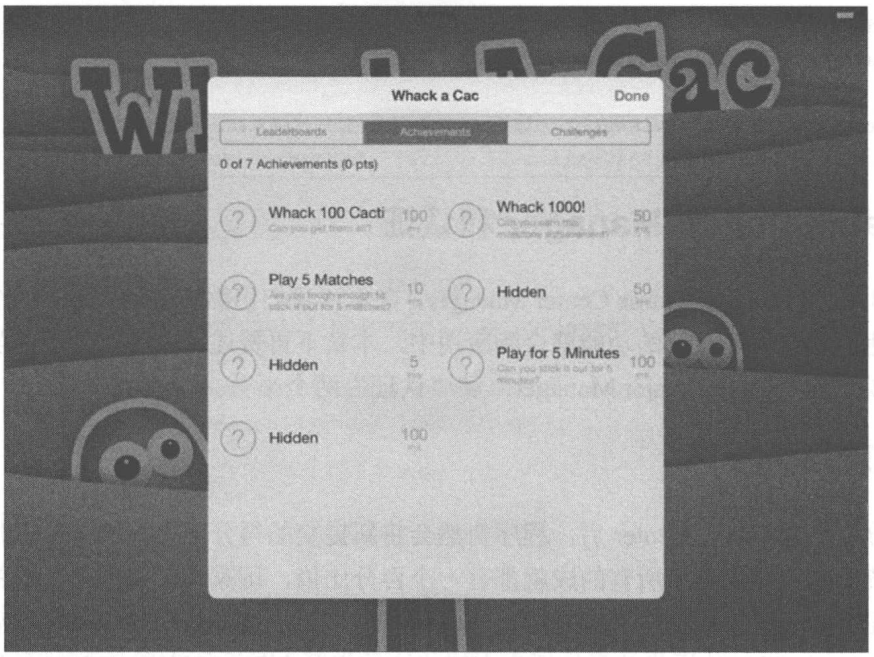


图 4-3 新的 Game Center View Controller，显示成就系统界面

注意

同排行榜类似，应用上架之后就不可以删除了。

在“向 Whack-a-Cac 添加成就系统”小节中，我们会在示例程序内添加多种成就类型来实际体验这些功能。

4.2 显示成就进度

如果不将当前成就的进度显示给用户，这个功能就没有任何意义。如果需要为成就系统显示一个自定义的界面，请查看“深入讨论成就系统”小节。下面的方法会启动一个组合的 Game Center View，如图 4-3 所示：

```
-(void)showAchievements
{
    [[GKGameCenterViewController sharedController] setDelegate:self];


[[GKGameCenterViewController sharedController]
➥setViewState:GKGameCenterViewControllerStateAchievements];

    [self presentViewController:[GKGameCenterViewController
    ➥sharedController] animated:YES completion:nil];
}
```

在上面的方法中，启动 Game Center View Controller 时设置了一个委托函数，这是该视图控制器需要的一个新的委托函数，它的实现如下所示：

```
-(void)gameCenterViewControllerDidFinish:(GKGameCenterViewController
➥*)gameCenterViewController
{
    [self dismissModalViewControllerAnimated:YES completion:nil];
}
```

4.3 Game Center Manager 和认证

在第 3 章介绍了新类 Game Center Manager，在本章我们仍然需要用到它，这个类可以让开发者快速将 Game Center 的功能整合到应用中，本章不再赘述。在继续下面的学习前好好阅读下第 3 章的“Game Center Manager”和“认证”两个小节。

4.4 成就系统缓存

把得分提交给 Game Center 后，程序自然会将新提交的得分同之前的得分进行比较。不过成就系统稍微有些复杂，所有的成就都有一个百分比值，玩家为了完成某个成就可能耗时几天甚至几个月。所以保存好玩家阶段性的成就非常重要，这样他们就可以在日后继续努力以达到最终目标。可以使用成就系统缓存来解决这一问题，将所有云端成就数据保存到本地，并使用新的会话刷新程序。

需要将一个新的便捷方法添加到 ICFGameCenterManager 类和新的类级标签 NSMutableDictionary，并调用 achievementDictionary。在这个方法中首先需要完成的是推送一条提醒信息，即在成就系统缓存建立好之后通知用户。虽然生成缓存可能不止一次，但是其不能影响程序的正常运行。如果 achievementDictionary 不存在，将会创建一个新的 NSMutableDictionary 对象并对其进行初始化。

注意

loadAchievementsWithCompletionHandler 方法不会向没有保存进度的成就对象返回 GKAchievement。

```
-(void)populateAchievementCache
{
    if(achievementDictionary != nil)
    {
        NSLog(@"Repopulating achievement cache:%@",
        achievementDictionary);
    }

    else
    {
        achievementDictionary = [[NSMutableDictionary alloc] init];
    }

    [GKAchievement loadAchievementsWithCompletionHandler:^(NSArray
    ➥*achievements, NSError *error)
    {
```

```
        if(error != nil)
        {
            NSLog(@"An error occurred while populating the
            ➥achievement cache:%@", [error localizedDescription]);
        }

        else
        {
            for(GKAchievement *achievement in achievements)
            {
                [achievementDictionary setObject:achievement
                ➥forKey:[achievement identifier]];
            }
        }
    }];
}
```

注意

本地用户只有成功认证之后才可以向 Game Center 发起调用。

4.5 上报成就系统

在实现了成就系统缓存之后，安全可靠的做法是向成就系统提交用户阶段性的成绩。在 ICFGameManager 类中添加新方法 reportAchievement:withPercentageComplete:以完成这个任务。当调用该方法时，将 achievementID 和成就完成百分比作为参数传入。要了解成就系统百分比完成度的有关内容，参见“添加成就关联”小节。

当提交阶段性成就时，首先要确定 achievementDictionary 已经生成。还要注意检查 Game Center 中当前成就的进度，以免由于用户切换设备或重新安装程序而出现丢失进度的问题。在本例中如果 achievementDictionary 为 nil，应用启动失败并打印输出一条日志；不过还可以实现一个更加复杂的初始化系统和成就缓存生成系统。

在确认 achievementDictionary 初始化成功之后，将创建一个 GKAchievement 对象。之后使用成就标识符将该对象保存在 achievementDictionary 中。如果该成就还没有任何进度，则不会出现在 achievementDictionary 中，且成就对象为 nil。此时将会分配一个新的 GKAchievement 对象并使用成就标识符对其进行初始化。

如果成就对象不为 nil，可以认为之前用户曾经取得过阶段性的成就。安全的做法是检查并确定即将上传的成就完成百分比不能低于在 Game Center 中发现的百分比。此外，还要检查成就是否全部完成。如果出现这两种情况，则需要向控制台输出一个 NSLog 对象并返回相应的方法。

注意

向成就系统提交一个低于已有进度的值是可能的，但这会降低用户现有的进度值。

至此，我们创建或重新得到了一个有效的 GKAchievement 对象，并且进度完成百分比大

于缓存中的值。成就对象使用 setPercentComplete:方法更新完成的百分比值。此时成就对象也保存到 achievementDictionary，即本地成就系统缓存的值为最新值。

要将实际的成就值发送给 Game Center，需要对成就对象调用 reportAchievementWithCompletionHandler:方法。当错误检查结束时，会用到一个新的委托回调函数来通知 GameCenterManager 委托提交操作是否成功。

```
-(void)reportAchievement:(NSString *)identifier
 withPercentageComplete:(double)percentComplete
{
    if(achievementDictionary == nil)
    {
          NSLog(@"An achievement cache must be populated before
          ➥submitting achievement progress");

          return;
    }

    GKAchievement *achievement = [achievementDictionary objectForKey:identifier];

    if(achievement == nil)
    {
       achievement = [[GKAchievement alloc]
       ➥initWithIdentifier:identifier];

       [achievement setPercentComplete:percentComplete];

       [achievementDictionary setObject:achievement forKey:identifier];
    }

    else
    {
       if([achievement percentComplete] >= 100.0 || [achievement
       ➥percentComplete] >= percentComplete)
       {
          NSLog(@"Attempting to update achievement %@ which is either
          ➥already completed or is decreasing percentage complete
          ➥(%f)", identifier, percentComplete);

         return;
       }

          [achievement setPercentComplete:percentComplete];

          [achievementDictionary setObject:achievement forKey:identifier];
    }

    [achievement reportAchievementWithCompletionHandler:^(NSError *error)
    {
          if(error != nil)
```

```
        {
            NSLog(@"There was an error submitting achievement
            ➥%@:%@", identifier, [error localizedDescription]);
        }

        [self callDelegateOnMainThread:
        ➥@selector(gameCenterAchievementReported:)
        ➥withArg:NULL error:error];
    }];
}
```

注意

成就系统同第3章的排行榜类似，如果其没有成功连接到Game Center服务器，程序不会尝试再次提交数据。开发者需要负责捕捉错误信息并在之后重新提交成就数据。不过同排行榜不同的是，成就系统没有日期属性，所有也就不需要存储实际的GKAchievement对象。

4.6 添加成就关联

在将成就系统整合到iOS游戏程序的开发过程中，公认最困难的工作就是将成就系统和工作流进行关联。比如，用户在一个角色扮演游戏中收集100个金币可获得成就。每收集一个金币，应用都需要更新成就值。

还有一个更加困难的例子，比如成就描述为在6个月之内每周完成一场比赛。这就需要许多节点和检查来确保用户是否满足了成就所需的条件。虽然试图记录每个单独的节点有点麻烦，不过“向Whack-a-Cac添加成就系统”小节中仍然对几种常见的关联节点进行了介绍。

在一个成就发生进展之前，首先需要检查其当前的进度。因为Game Center成就系统没有进度参数(比如向目前完成度增加1%)，这种工作还是留给开发者吧。下面的代码给出了根据标识符快速查找GKAchievement对象的一个简单方法，在GKAchievement对象返回之后，查询percentageComplete可以得到当前的进度。

```
-(GKAchievement *)achievementForIdentifier:(NSString *)identifier
{
    GKAchievement *achievement = nil;

    achievement = [achievementDictionary objectForKey:identifier];

    if(achievement == nil)
    {
        achievement = [[GKAchievement alloc]
        ➥initWithIdentifier:identifier];

        [achievementDictionary setObject:achievement forKey:identifier];
    }

    return achievement;
}
```

如果成就系统的进度的精度想要大于 1%，真实的完成度数值就不能从 Game Center 返回。Game Center 只能返回和接受整数值表示的百分比。解决这个问题有两种办法，简单的方法就是对进度四舍五入为整数；另一个相对复杂点的方法是将真实值保存在本地并使用该值计算百分比。时刻要记得用户会在不止一台设备上运行你的应用，所以在本地保存成就进度会存在很大的问题。第 11 章“使用 CloudKit 实现云存储”会给出一些其他的方法。

4.7 进度完成通知栏

Game Center 具有一种自动消息机制可以通知用户成就已达成。另外，还可以自定义一个系统用于显示非标准化的通知信息。虽然通知用户成就达成很重要，但是更加重要的是要充分考虑用户体验。

要自动显示用户成就已达成，需要在将成就数据提交到 Game Center 之前将 showsCompletionBanner 属性设置为 YES，如图 4-4 所示。添加该设置代码的一处比较好的地方是在 reportAchievement: withPercentageComplete:方法内。

```
achievement.showsCompletionBanner = YES;
```

图 4-4 Game Center 自动成就达成通知栏，在 iPad 上显示成就名称和具体描述信息

4.8 成就挑战系统

Game Center 对 Achievement Challenges(成就挑战)的支持类似于之前介绍的得分系统，用户可以向 Game Center 好友发起挑战，战胜他们的得分或达到他们取得的成就，如图 4-5 所示。

注意

只要成就可见，Game Center 就允许用户向未获得成就的用户发起挑战。

```
-(void)showChallenges
{
    [[GKGameCenterViewController sharedController]setDelegate:self];

     [[GKGameCenterViewController sharedController] setViewState:
     ➥GKGameCenterViewControllerStateAchievements];

    [self presentViewController:[GKGameCenterViewController
    ➥sharedController] animated:YES completion:nil];
}
```

如果本章之前的示例中没有实现 gameCenterViewControllerDidFinish 委托方法，在此就需要它。

```
-(void)gameCenterViewControllerDidFinish:(GKGameCenterViewController
➥*)gameCenterViewController
{
    [self dismissModalViewControllerAnimated:YES completion:nil];
}
```

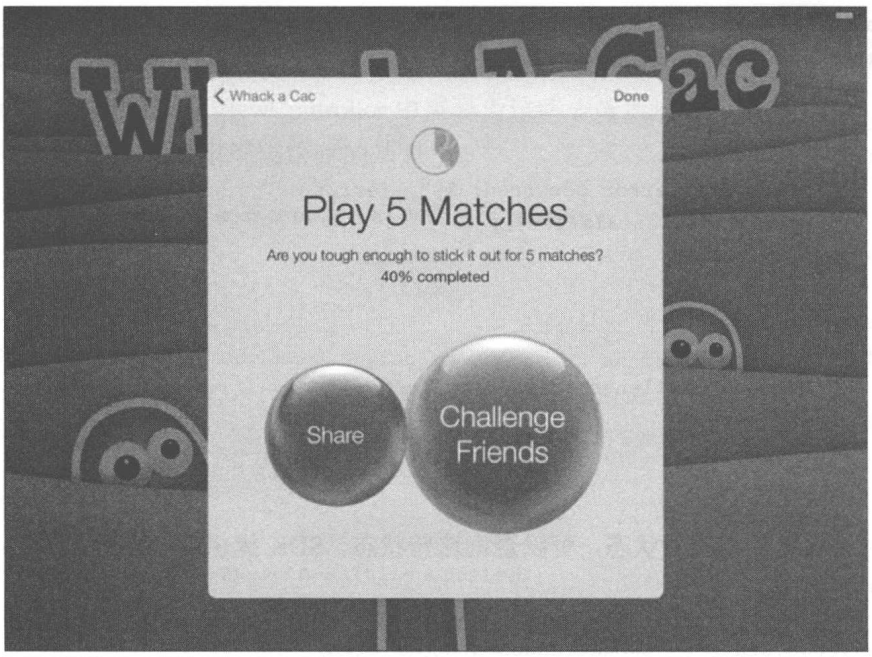


图 4-5 用一个正在努力完成的成就向朋友发起挑战

注意

如果用户接收的挑战是他们已经获得的成就，就需要在 iTunes Connect 中创建成就时选择 Achievable More Than Once 选项。

使用下面的方法可以通过程序编码的方式实现挑战功能：

```
[(GKAchievement *)achievementissueChallengeToPlayers:(NSArray
➥ *)players message: @"I earned this achievement, can you?"];
```

如果需要得到对某一成就发起挑战的用户列表(假定没有选择 Achievable More Than Once 选项)，可以通过下面的代码段实现用户列表：

```
[achievement selectChallengeablePlayerIDs:arrayOfPlayersToCheck
➥withCompletionHandler:^(NSArray *challengeablePlayerIDs, NSError
➥*error)
{
     if(error != nil)
     {
          NSLog(@"An error occurred while retrieving a list of
          ➥challengeable players: %@", [error localizedDescription]);
     }

     NSLog(@"The following players can be challenged: %@",
     challengeablePlayerIDs);
}];
```

还可以通过下面的代码段获取包含所有经过认证且待挑战的 GKChallenge 对象组成的数组：

```
[GKChallenge loadReceivedChallengesWithCompletionHandler:^(NSArray
➥*challenges, NSError *error)
{
     if(error != nil)
     {
          NSLog(@"An error occurred: %@", [error
          ➥localizedDescription]);
     }

     else
     {
          NSLog(@"Challenges: %@", challenges);
     }

}];
```

挑战对象都具有相应的状态，可以查询这些状态。SDK 提供的状态包括无效、进行中和拒绝这 3 种。

```
if(challenge.state == GKChallengeStateCompleted)
      NSLog(@"Challenge Completed");
```

最后，对象通过调用 decline 可以拒绝对方发起的挑战，如下所示：

```
[challenge decline];
```

通过利用挑战系统并鼓励用户使用挑战功能，可以很好地增加用户对产品的黏性和使用时间。如果对 Game Center 使用内置的 GUI，甚至不需要为程序支持挑战系统而编写额外的代码。

注意

Whack-a-Cac 示例中并未使用编程的方式实现成就挑战，也没有相关的代码。

4.9 向 Whack-a-Cac 添加成就系统

Whack-a-Cac 将使用不同的 hook 方法实现 6 种成就。表 4-1 给出了将会实现的成就描述。

表 4-1 Whack-a-Cac 中用到的成就及达成要求

成就 ID	详 情
com.dragonforged.whackacac.killone	击中第一个仙人掌之后获得。这是一个隐藏成就，直到用户完成该成就才会显示
com.dragonforged.whackacac.score100	在单次游戏中得分达到 100 之后获得。该成就为隐藏成就，直到用户开始进行任务时才会显示
com.dragonforged.whackacac.100whacks	完成击中 100 个仙人掌后获得，可以通过多局游戏完成
com.dragonforged.whackacac.1000whacks	完成击中 1000 个仙人掌后获得，可以通过多局游戏完成
com.dragonforged.whackacac.play5	完成 5 局比赛之后获得
com.dragonforged.whackacac.play5Mins	游戏时间超过 5 分钟后获得

假设本章前面有关 ICFGameCenterManager 的细节修改工作都已实现，现在可以向成就系统添加 hook。在 ICFGameViewController 中需要为这些成就添加委托回调方法，这样委托函数就可以接收成就达成或出现错误时的消息了。

```
-(void)gameCenterAchievementReported:(NSError *)error;
{
    if(error != nil)
    {
        NSLog(@"An error occurred trying to report an achievement to
        ➥Game Center: %@", [error localizedDescription]);
    }

    else
    {
        NSLog(@"Achievement successfully updated");
    }
}
```

注意

在 Whack-a-Cac 中，本地用户成功认证之后就会调用 populateAchievementCache 方法。

4.9.1 是否达成成就

表 4-1 中最容易实现的成就是 com.dragonforged.whackacac.killone。当为一个成就添加一

个锚点时，第一步是获取即将发生增量的 GKAchievement 对象的副本。使用“添加成就锚点”小节中讨论的方法获取最新的成就完成度。

```
GKAchievement *killOneAchievement = [[ICFGameCenterManager
➥sharedManager] achievementForIdentifier:
➥@"com.dragonforged.whackacac.killone"];
```

接下来，需要执行一条查询语句，检查成就是否已经完成。如果完成，则不需要对其进行更新。

```
if(![killOneAchievement isCompleted])
```

由于 Kill One(消灭一个仙人掌)这个任务不可能再分解了，因此该任务的增量就是 100%。向本章前面介绍的 ICFGameCenterManager 类中添加 reportAchievement:withPercentageComplete:方法以实现消灭一个仙人掌、任务增量完成 100%的功能。

```
[[ICFGameCenterManagersharedManager]
➥reportAchievement:@"com.dragonforged.whackacac.killone"
➥withPercentageComplete:100.00];
```

当仙人掌被击中时这个成就要经过测试并提交，所以将其写在 cactusHit:方法内部是比较合适的。我们对之前的 cactusHit:方法进行改进，如下所示：

```
-(IBAction)cactusHit:(id)sender;
{
    [UIView animateWithDuration:0.1
                          delay:0.0
                        options:UIViewAnimationCurveLinear |
                        ➥UIViewAnimationOptionBeginFromCurrentState
                     animations:^
                     {
                        [sender setAlpha:0];
                     }
                     completion:^(BOOL finished)
                     {
                         [sender removeFromSuperview];
                     }];

    score++;

    [self displayNewScore:score];

    GKAchievement *killOneAchievement = [[ICFGameCenterManager
    ➥sharedManager] achievementForIdentifier:
    ➥@"com.dragonforged.whackacac.killone"];

    if(![killOneAchievement isCompleted])
    {
        [[ICFGameCenterManager sharedManager]reportAchievement:
        ➥@"com.dragonforged.whackacac.killone"
```

```
        ➥withPercentageComplete:100.00];
    }

    [self performSelector:@selector(spawnCactus) withObject:nil
    ➥afterDelay:(arc4random()%3) + .5];
}
```

4.9.2 部分完成的成就

在上面的例子中，给出的成就类型要么完成要么未完成，只有这两种可能。我们在Whack-a-Cac中接下来要实现的成就是com.dragonforged.whackacac.score100。同Kill One成就不同，这个成就可以分步完成，不过不可以多局游戏累计完成，要求用户在一局游戏中得到100分。成就的开始同上面的例子一样，需要创建一个到GKAchievement对象的引用。

```
GKAchievement *score100Achievement = [[ICFGameCenterManager
sharedManager] achievementForIdentifier:
@"com.dragonforged.whackacac.score100"];
```

还需要对成就是否完成进行一次快速检查。

```
if(![score100Achievement isCompleted])
```

当成就有效且还未完成时，可以根据需求累加得分。由于成就达到100%即为完成，而得分正好是100分，因此每得一分即为完成进度1%。这样在完成成就的过程中得分可以用百分比数据替换。

```
[[ICFGameCenterManagersharedManager]
reportAchievement: @"com.dragonforged.whackacac.score100"
withPercentageComplete:score];
```

虽然这个锚点还可以放在cactusHit:方法内，不过将其放在displayNewScore:内可能更为合适，因为它主要处理与得分相关的内容。下面给出更新了新成就锚点的displayNewScore:方法的全部代码：

```
-(void)displayNewScore:(float)updatedScore;
{
    int scoreInt = score;

    if(scoreInt % 10 == 0 && score <= 50)
    {
        [self spawnCactus];
    }

    scoreLabel.text = [NSString stringWithFormat:@"%06.0f",
    ➥updatedScore];

    GKAchievement *score100Achievement = [[ICFGameCenterManager
    ➥sharedManager] achievementForIdentifier:
    ➥@"com.dragonforged.whackacac.score100"];
```

```
    if(![score100Achievement isCompleted])
    {
        [[ICFGameCenterManager sharedManager] reportAchievement:
        ➥@"com.dragonforged.whackacac.score100"
        ➥withPercentageComplete:score];
    }
}
```

由于得分 100 这个成就是隐藏的，因此直到用户开始做任务时才会显示给用户(至少得到 1 分开始)。在这一成就开始之后的任意时间点，用户都可以在 Game Center View Controllers 中看到进度，如图 4-6 所示。

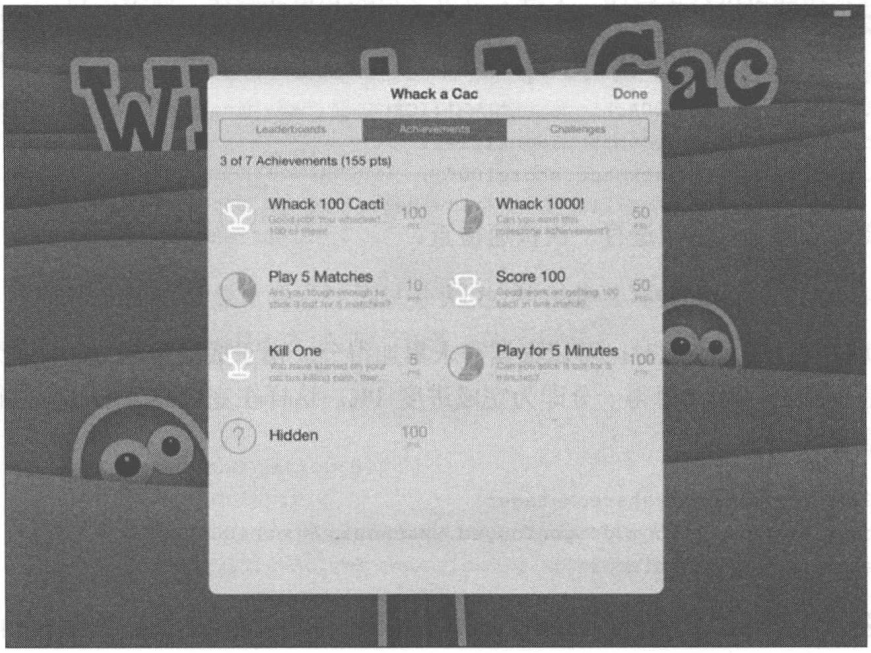


图 4-6 查看“得分达到 100”成就阶段性成果，这里玩家得分为 43 分，注意 Kill One 成就已经获得

4.9.3 多会话成就

在上一个例子中，获得 100 得分的成就要求用户必须在一局游戏中完成。很多时候，一些成就需要追踪用户多局游戏甚至多个应用才能实现。这里我们首先实现跨多局游戏的一个成就 com.dragonforged.whackacac.play5。

每当用户完成一局游戏，com.dragonforged.whackacac.play5 成就就会有所进展。由于成就描述为完成 5 局游戏，因此每完成一局游戏即为完成进度的 20%。将锚点添加到 ICFGameViewController 类的 viewWillDisappear 方法中。同上面的例子一样，首先要创建一个到 GKAchievement 对象的引用。如果成就还未完成，则增加完成进度。创建一个新变量，记录已经完成的游戏局数，它和完成百分比的关系是 20 倍率。matchesPlayed 增加 1，matchesPlayed 乘以 20 作为结果提交给 reportAchievement:withPercentageComplete:

```
-(void)viewWillDisappear:(BOOL)animated
{
```

```
    GKAchievement *play5MatchesAchievement = [[ICFGameCenterManager
    ➥sharedManager] achievementForIdentifier:
    ➥@"com.dragonforged.whackacac.play5"];

    if(![play5MatchesAchievement isCompleted])
    {
        double matchesPlayed = [play5MatchesAchievement
        ➥percentComplete]/20.0f;

        matchesPlayed++;

        [[ICFGameCenterManager sharedManager]
        ➥reportAchievement:@"com.dragonforged.whackacac.play5"
        ➥withPercentageComplete:matchesPlayed*20.0f];
    }

    [super viewWillDisappear:animated];
}
```

4.9.4 携带成就和保存成就精度

有时候两个成就之间会有重叠的部分，比如击中 100 次仙人掌和击中 1000 次仙人掌这两个成就会有重叠。由于这两个成就都跟踪同一对象类型(whacks)，因此可以使用一个更为简化的方法来实现。

同本章实现的其他成就锚点一样，首先需要创建一个关于 GKAchievement 的引用。在下面的例子里，对两个成就中的较大者创建了引用。由于最大的成就值要远高于百分比值，因此可能会四舍五入到相近的十位数数值。要解决这一问题，需要从 NSUserDefaults 生成一个 localKills 变量。当成就值用于两台不同的设备时系统就会崩溃，不过后面我们会学习使用 iCloud 来存储有关成就的数据(见第 11 章)。

在 Game Center 的报告中还会计算击杀数(降低精度)，如果 remoteKills 值大于 localKills，我们就知道用户可能重新安装了游戏或者在其他设备上运行该游戏取得了一些分数。这时系统会默认向 Game Center 请求进度值，避免缩减完成度，否则将会使用本地信息。

在本节前面介绍的 Kill One 成就代码之后，示例程序可以把下述代码添加到 cactusHit: 方法的实现中。每次点击成功后，localKills 值增加 1。还需要进行两次检查来确保成就尚未完成，因为关于两个 GKAchievement 的引用还未生效，所以检查击中个数可以用来替代标准的 isComplete 检查。在成就提交后，需要将新的本地数据值保存在 NSUserDefaults 中，以便将来引用。

```
GKAchievement *killOneThousandAchievement = [[ICFGameCenterManager
➥sharedManager] achievementForIdentifier:
➥@"com.dragonforged.whackacac.1000whacks"];

double localKills = [[[NSUserDefaults standardUserDefaults]
➥objectForKey:@"kills"] doubleValue];

double remoteKills = [killOneThousandAchievement percentComplete] *
```

```
➥10.0;

if(remoteKills > localKills)
{
    localKills = remoteKills;
}

localKills++;

if(localKills <= 1000)
{
     if(localKills <= 100)
     {
         [[ICFGameCenterManager sharedManager]
         ➥reportAchievement:@"com.dragonforged.whackacac.100whacks"
         ➥withPercentageComplete:localKills];
     }

     [[ICFGameCenterManager sharedManager]
     ➥reportAchievement:@"com.dragonforged.whackacac.1000whacks"
     ➥withPercentageComplete:(localKills/10.0)];
}

[[NSUserDefaults standardUserDefaults] setObject:[NSNumber
➥numberWithDouble:localKills] forKey:@"kills"];
```

4.9.5 基于时间的成就

iOS 游戏中一个最流行的成就任务就是累计游戏时间。在 Whack-a-Cac 中，com.dragonforged.whackacac.play5Mins 成就任务就需要用户累计游戏时间 5 分钟。这个特殊示例存在失真的问题，即上一小节“击中 1000 个仙人掌”成就任务中存在的问题。

要记录时间，需要创建一个 NSTimer 对象，为了确定计时器的计时间隔，就需要计算 5 分钟的 1%是多少。

```
play5MinTimer = [NSTimer scheduledTimerWithTimeInterval:3.0 target:self
➥selector:@selector(play5MinTick) userInfo:nil repeats:YES];
```

当计时器开启时，会调用新方法 play5MinTick。如果游戏暂停或处于 gameOver 状态，忽略成就的进度并返回。同其他例子一样，需要创建一个对 GKAchievement 对象的引用，并检查成就是否完成。如果成就已完成，计时器就处于无效状态以避免浪费 CPU 时间。如果没有完成，则计时器每 3 秒钟(5 分钟的 1%)启动一次，成就进度增加 1%。

```
-(void)play5MinTick;
{
    if(paused || gameOver)
    {
        return;
    }
```

```
    GKAchievement *play5MinAchievement = [[ICFGameCenterManager
    ➥sharedManager] achievementForIdentifier:
    ➥@"com.dragonforged.whackacac.play5Mins"];

    if([play5MinAchievement isCompleted])
    {
        [play5MinTimer invalidate];
        play5MinTimer = nil;
        return;
    }

    double percentageComplete = play5MinAchievement.percentComplete
    ➥+ 1.0;

    [[ICFGameCenterManager sharedManager] reportAchievement:
    ➥@"com.dragonforged.whackacac.play5Mins" withPercentageComplete:
    ➥percentageComplete];
}
```

4.10 重置成就系统

程序经常需要重置所有的成就，在开发阶段尤其如此。有时候这么做是让用户有机会重新玩一遍游戏或者在有些声望模式(prestige mode)中选择更高的游戏难度重新玩一遍。重置成就进度很简单，下面的代码段用于实现向程序中添加这个功能。如果准备向用户提供这个功能，一定要注意其中的一些步骤以避免出现重置事故。

```
-(void)resetAchievements
{
     [achievementDictionary removeAllObjects];

      [GKAchievement resetAchievementsWithCompletionHandler:
     ^(NSError *error)
      {
           if(error == nil)
           {
               NSLog(@"All achievements have been successfully
               ➥reset");
           }

           else
           {
              NSLog(@"Unable to reset achievements:%@",
              ➥[error localizedDescription]);
           }
      }];
}
```

提示

当应用还处在开发和调试阶段时，一个比较好的建议是在成功认证之后，在

ICFGameCenterManager 类的实现中保持一个对成就重置的调用，以方便测试和实现相应的功能，后面不用的时候只需将其注释掉即可。

4.11 深入讨论成就系统

苹果公司在有关显示和优化成就系统方面给予开发者很大的自由度，不过苹果公司不允许对成就视图界面进行自定义。因为担心开发者自定义的界面效果和体验同应用的整个设计不相符。在这种情况下，源成就信息可以通过自定义的界面进行访问。虽然完全自定义一个成就系统超出了本章的讨论范围，不过本节的内容还是对你大有裨益。

之前已经学习了有关创建 GKAchievement 本地缓存的内容，GKAchievement 对象缺少用户显示成就信息的关键数据，比如成就描述、标题、名称和得分等。此外，当使用缓存时，如果成就没有开始，则其不会出现在缓存中。要获得所有的成就信息并请求显示它们，需要使用一个新类。

使用 GKAchievementDescription 类和类方法 loadAchievementDescriptionsWithCompletion-Handler:可以访问 GKAchievementDescriptions 数据组成的数组。GKAchievementDescription 对象包含标题、描述、图片和其他关键信息所对应的属性值。不过 GKAchievementDescription 不包含本地用户当前成就进度的信息，所以为了确定成就进度，需要和本地成就缓存中的数据比较标识符。

```
[GKAchievementDescription
➥loadAchievementDescriptionsWithCompletionHandler:^(NSArray
➥*descriptions, NSError *error)
{
    if(error != nil)
    {
        NSLog(@"An error occurred loading achievement
        ➥descriptions:%@", [error localizedDescription]);
    }

    for(GKAchievementDescription *achievementDescription in
    ➥descriptions)
    {
        NSLog(@"%@\n", achievementDescription);
    }

}];
```

在 Whack-a-Cac 程序中运行以上代码，控制台将会输出如下信息：

```
WhackACac[48552:c07] <GKAchievementDescription:
0x1185f810>id:com.dragonforged.whackacac.100whacks Whack 100 Cacti
visible Good job! You whacked 100 of them!

WhackACac[48552:c07] <GKAchievementDescription:
0x1185f980>id:com.dragonforged.whackacac.1000whacks Whack 1000!
visible You are a master at killing those cacti.
```

```
WhackACac[48552:c07] <GKAchievementDescription:
0x1185f820>id:com.dragonforged.whackacac.play5 Play 5 Matches visible
Your dedication to cactus whacking is unmatched.

WhackACac[48552:c07] <GKAchievementDescription:
0x1185eae0>id:com.dragonforged.whackacac.score100 Score 100 hidden
Good work on getting 100 cacti in one match!

WhackACac[48552:c07] <GKAchievementDescription:
0x1185eaf0>id:com.dragonforged.whackacac.killone Kill One  hidden You
have started on your cactus killing path; there is no turning back now.

WhackACac[48552:c07] <GKAchievementDescription:
0x1185eb30>id:com.dragonforged.whackacac.play5Mins Play for 5 Minutes
visible Good work! Your dedication continues to impress your peers.

WhackACac[48552:c07] <GKAchievementDescription:
0x1185eb40>id:com.dragonforged.whackacac.hit5Fast Hit 5 Quick hidden
You are truly a quick gun.
```

所有成就项目组成的列表就显示出来了，现在可以使用这些数据同本地缓存中的数据进行比较，确定成就完成的百分比，这样创建一个自定义 GUI 所需的数据就已全部得到。

4.12 小结

本章介绍了如何将 Game Center 的成就系统整合到 iOS 项目中，我们继续使用第 3 章用到的示例程序 Whack-a-Cac，并继续对可重用的类 Game Center Manager 进行了扩展。

现在你应该熟练掌握了如何创建成就系统，设置应用同成就系统交互，发送和显示进度，重置成就进度等技术。此外，我们也对标准做法进行了一点扩展，通过本章学到的知识，你现在完全可以将成就系统整合到任何一款应用中。

第5章

Address Book 框架初步

目前 iOS 系统中用到的 Address Book 框架从 iOS 2.0 版本(即 iPhone OS 2.0)引入之后几乎没有多大的变化。该框架同 OS X 系统上的基本一致，从 OS X 10.2 就已经存在了，这也是 iOS 系统上最老的框架之一。当开发者开始学习有关 Address Book 技术时就可以印证这一点。该框架基于 Core Foundation 框架，iPhone SDK 的引入对于从 Objective-C 和 Cocoa 程序转过来的开发者可能有点陌生。

5.1 支持 Address Book 很重要

当开发 iOS 软件时，要时刻记住这个软件会伴随用户的移动生活而存在。用户会带着移动设备到各种地方，设备同用户的个人生活也紧密交织在一起，从日历到个人相册。用户移动生活的核心就是他们的通讯录，其中的数据信息都是用户长时间、从多个设备积累收集的，包含了用户的家庭、工作和社交生活的各种信息。

应用可以使用通讯录数据库，通过分析邮件地址或电话号码等列表来确定用户是否有朋友注册了某个服务，然后自动将其添加为好友。应用还可以使用通讯录列表实现自动生成邮件联系人、电话号码或用蓝牙共享联系人信息等功能。应用需要访问用户通讯录的理由数不胜数。

注意

只有当应用有正当的理由需要访问通讯录时这么做才有意义，因为让用户对你的应用感到反感的最快办法就是侵犯他的隐私。

5.2 Address Book 开发的限制

虽然 Address Book 框架已经相当开放了，不过还有一些重要的限制需要注意。最主要的就是对于 Mac 开发者来说，这里没有“我”的概念。事实上，没有办法在通讯录列表中识别用户。虽然有一些黑客试图这样做，不过至今还没有一个可靠的解决办法，也没有苹果公司授权的办法。

一个新的受欢迎的限制——主要是针对关系个人隐私的用户——是针对访问通讯录数据库的 Core Location 类型认证的扩展，意思是访问通讯录之前必须得到用户的允许。当编写 Address Book 软件时，要做到即使用户拒绝程序访问通讯录，也仍然可以继续运行其他功能。

从 iOS 6 开始，在 Settings 程序(设置程序)中加入了一个新的隐私页面。在这个页面上，用户可以启用或关闭程序访问通讯录、位置、日历、提醒、照片和蓝牙的权限。

5.3 示例程序

本章的示例程序是一个简单的通讯录视图和编辑器。当启动程序时，将会获取并展示设备上的联系人信息。导航栏的右上角有一个加号按钮“+”，用于通过内置界面添加新的联系人，还有一个切换键用来选择显示电话号码还是街道地址。此外还可以通过编程的方式添加新用户，并给出了一个使用内置用户选择器的例子。

由于示例程序(如图 5-1 所示)仅仅是一个基本的导航控制器项目并且只有关于 Address Book 的代码，因此在下一节使用这些代码时要慎重。

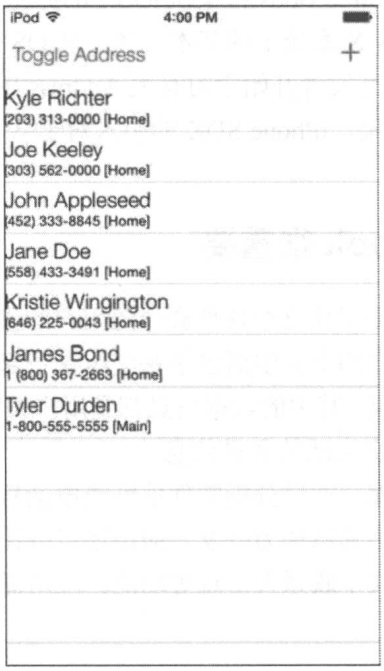


图 5-1 示例程序界面

5.4 开始实现 Address Book 并运行

在使用 Address Book 框架之前，首先需要将框架 AddressBookUI.framework 和 AddressBook.framework 关联到项目中。AddressBookUI.framework 框架主要处理用户界面，比如选择、编辑或展示联系人；AddressBook.framework 框架主要处理数据交互。需要添加两个头文件，如下所示：

```
#import <AddressBook/AddressBook.h>
#import <AddressBookUI/AddressBookUI.h>
```

在示例程序中，头文件在 RootViewController.h 文件中导入，因为类需要遵循一些委托协议，后面会有这些协议相关的介绍。还需要创建一个类级实例 ABAddressBookRef，在示例程序中叫作 addressBook。示例程序还有一个 NSArray 数组用来保存联系人的相关条目信息。将通讯录复制到内存的操作会消耗大量资源，所以应尽量减少该操作的次数。

```
ABAddressBookRefaddressBook;
NSArray *addressBookEntryArray;
```

要生成这个新的 NSArray 数组，需要调用 ABAddressBookCreate 方法，将会基于当前全局通讯录数据库创建新的通讯录数据。在示例程序中，这一部分的实现是在 viewDidLoad 方法中完成的。

```
if(addressBook == NULL)
{
   NSLog(@"Error loading address book: %@", CFErrorCopyDescription(creationError));
}

ABAddressBookRequestAccessWithCompletion(addressBook, ^(bool granted,
➥CFErrorRef error)
{
   if(!granted)
   {
      NSLog(@"No permission!");
   }
});
```

同时还要捕获通讯录中没有联系人的事件，这是 iOS 模拟器默认的行为。示例程序弹出一个 UIAlert 对话框提示用户程序没有出错，只不过目前没有可用的数据。还可以使用 ABAddressBookGetPersonCount 函数查询 ABAddressBookRef 的大小。

```
if(ABAddressBookGetPersonCount(addressBook) == 0)
{
     UIAlertView *alertView = [[UIAlertView alloc]initWithTitle:@""
     message:@"Address book is empty!"
     delegate:nil
     cancelButtonTitle:@"Dismiss"
     otherButtonTitles:nil];

     [alertView show];
}
```

现在有了一个关于通讯录的引用，不过需要将它转换为更易管理的数据集。示例程序将这些对象复制到 NSArray 数组中，因为需要用这些数据来填充表视图。

复制通讯录数据需要访问 3 个函数并返回一个 CFArrayRef 对象，这 3 个函数如下：

- ABAddressBookCopyArrayOfAllPeople：返回一个包含通讯录中所有人的数组(在下一个代码段中给出了该方法)。
- ABAddressBookCopyArrayOfAllPeopleInSource：返回在特定源上找到的通讯录组件。

- ABAddressBookCopyArrayOfAllPeopleInSourceWithSortOrdering：可以对获得的通讯录列表中的相关条目进行排序。

示例程序目前不需要考虑排序问题，所以只需要调用 ABAddressBookCopyArrayOfAllPeople 返回联系人信息即可。由于 CFArrayRef 是到 NSArray 数组的桥梁，因此可以进行类型转换得到 NSArray 数组。现在我们已经得到了一个包含所有通讯录条目的数组，将它们展示在表中就很容易了。

```
addressBookEntryArray = (NSArray *)ABAddressBookCopyArrayOfAllPeople(addressBook);
```

注意

本书中的源就是指获取联系人的地方，其可能值为 kABSourceTypeLocal、kABSourceTypeExchange、kABSourceTypeMobileMe 和 kABSourceTypeCardDAV。要得到相关通讯录中所有的源，可以使用 ABAddressBookCopyArrayOfAllSources(addressBook)。现在就可以在应用中根据需要查找相应的源了。

5.4.1 从 Address Book 读取数据

前面小节演示了如何使用用户通讯录中的条目来填充 NSArray 数组——每个对象就是一个 ABRecordRef。本节的主要任务是提取 ABRecordRef 中的信息。

示例程序通过一个 UITableView 视图显示用户数据，ABRecordRef 对象中包含两个类型的值，第一个类型是用于表示单一内容的值，比如姓和名。第二个类型是用于处理多个信息的多值类型，比如电话号码或街道地址。

下面的代码段从上一节创建的通讯录数组中提取一个 ABRecordRef 对象，并获取联系人的姓名，根据姓名设置 NSString 值。全部可用属性的列表如表 5-1 所示。

```
ABRecordRef record = [addressBookEntryArrayobjectAtIndex:indexPath.row];
NSString *firstName = (NSString *)ABRecordCopyValue(record,
➥kABPersonFirstNameProperty);

NSString *lastName = (NSString *)ABRecordCopyValue(record,
➥kABPersonLastNameProperty);

//...

    if(firstName)
        CFRelease(firstName);
    if(lastName)
        CFRelease(lastName);
```

表 5-1 ABRecordRef 中包含的所有可用单值常量

属 性 名	描 述
kABPersonFirstNameProperty	名字
kABPersonLastNameProperty	姓
kABPersonMiddleNameProperty	中间名或首字母

(续表)

属 性 名	描　述
kABPersonPrefixProperty	名字前缀(Mr.、Ms.、Dr.)
kABPersonSuffixProperty	名字后缀(MD、Jr.、Sr.)
kABPersonNicknameProperty	昵称
kABPersonFirstNamePhoneticProperty	发音拼写的名字
kABPersonLastNamePhoneticProperty	发音拼写的姓
kABPersonMiddleNamePhoneticProperty	发音拼写的中间名
kABPersonOrganizationProperty	公司或机构名
kABPersonJobTitleProperty	工作职位
kABPersonDepartmentProperty	部门职位
kABPersonBirthdayProperty	CFDate 格式的生日，它是到 NSDate 的桥梁
kABPersonNoteProperty	个人注释
kABPersonCreationDateProperty	CFDate 格式的创建时间
kABPersonModificationDateProperty	CFDate 格式的最后修改时间

注意

NARC(New,Alloc,Retain,Copy)机制是笔者在早期 Mac OS X 程序开发中学习的内存管理技术，目前对于手动处理内存管理问题仍然有效。诸如 ARC 等新机制的出现从根本上改变了手动内存管理方法。不过 Core Foundation 和 ARC 并不兼容，当我们在 Address Book 上对名字使用“复制”操作时，必须使用 CFRelease()方法释放内存。

5.4.2　从 Address Book 读取多值数据

我们经常会遇到存储多值数据的 Address Book 对象，比如电话号码、电子邮件地址或街道地址等，对这些内容的访问都使用 ABMultiValueRef。具体方法同获取单值数据类似，只不过稍微复杂一点而已。

在处理多值数据(比如电话号码)时，首先需要复制该多值属性的值。在下面的示例代码中，对上一小节中设置的记录使用 kABPersonPhoneProperty。这里为你提供了一个名为 phoneNumbers 的 ABMultiValueRef 对象。

使用 ABMultiValueGetCount 函数对数据进行检查，确定联系人至少有一个电话号码。这里可以遍历所有电话号码或者使用找到的第一个电话号码(例子中就是这么做的)。此外，还要处理“没有找到任何电话号码”这一情况。现在需要创建一个新的字符串并将电话号码保存到这个字符串中。调用 ABMultiValueCopyValueAtIndex 可以实现上述功能，索引号之后为 ABMultiValueRef 的第一个参数。

```
ABMultiValueRef phoneNumbers = ABRecordCopyValue(record,
➥kABPersonPhoneProperty);
```

```
if(ABMultiValueGetCount(phoneNumbers) > 0)
{
    CFStringRef phoneNumber =
    ➥ABMultiValueCopyValueAtIndex(phoneNumbers, 0);

    NSLog(@"Phone Number:%@", phoneNumber);

    CFRelease(phoneNumber);
}

CFRelease(phoneNumbers);
```

5.4.3 理解 Address Book 标签

在前一节中，我们从联系人数据库中获得了电话号码，不过我们只能根据它的索引号得到相应的信息。虽然这对于开发者来说很有帮助，不过对用户本身来说这并没有用。开发者需要获取用于联系人数据库的标签。在下面的代码段中，我们继续对上一节的程序进行扩展。

要获取多值引用对象的标签，首先需要调用 ABMultiValueCopyLabelAtIndex，该方法用到的参数和之前获取多值对象时所使用方法的参数相同。该函数会返回一个非本地化的字符串，比如"_$!<Mobile>!$_"。虽然这比原始索引号要好得多，不过仍然不能作为用户数据呈现。

需要通过一个本地化转换方法来获取用户可以读懂的字符串。为此，对刚刚得到的原始值 CFStringRef 使用 ABAddressBookCopyLocalizedLabel 方法。本例中将会返回 Mobile 或设备的选择语言所对应的值。

```
ABMultiValueRef phoneNumbers = ABRecordCopyValue(record,
➥kABPersonPhoneProperty);

if(ABMultiValueGetCount(phoneNumbers) > 0)
{
    CFStringRef phoneNumber =
    ➥ABMultiValueCopyValueAtIndex(phoneNumbers, 0);

    CFStringRef phoneTypeRawString =
    ➥ABMultiValueCopyLabelAtIndex(phoneNumbers, 0);

    NSString *localizedPhoneTypeString =  (NSString
    ➥*)ABAddressBookCopyLocalizedLabel(phoneTypeRawString);

    NSLog(@"Phone %@ [%@]", phoneNumber, localizedPhoneTypeString);

    CFRelease(phoneNumber);
    CFRelease(phoneTypeRawString);
    CFRelease(localizedPhoneTypeString);
}
```

回顾一下图 5-1 中的示例，现在可以完成对这些功能的设置了。

5.4.4　处理地址信息

在前面两节中，分别介绍了如何访问单值信息和多值数据。本节我们将结合之前介绍的知识学习如何处理联系人数据库中的街道地址信息。启动示例程序并点击导航栏上的切换按钮，将会看到地址取代了电话号码显示在表单元格中，如图 5-2 所示。

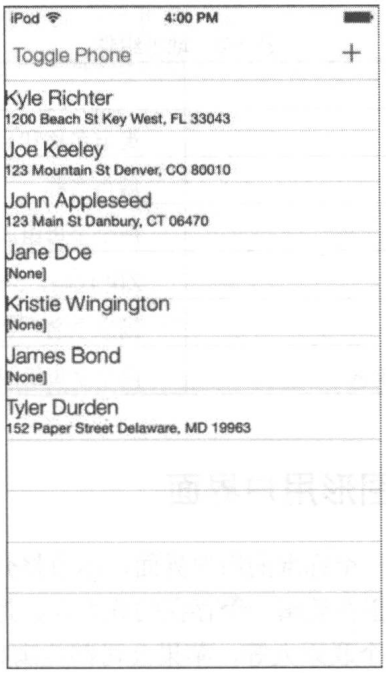


图 5-2　示例应用显示从联系人数据库中提取的地址信息

在开始处理地址时，所采用的方法同之前获取多值电话号码的类似。首先为 kABPersonAddressProperty 获得一个 ABMultiValueRef 对象。之后需要确定至少有可用地址数据，当使用索引值查询多值对象时，同返回单值 CFStringRef 不同的是，这里返回的是一个包含地址信息组件的字典文件。得到该字典文件之后，使用表 5-2 中的地址常量值提取具体的信息。

```
if(ABMultiValueGetCount(streetAddresses) > 0)
{
    NSDictionary *streetAddressDictionary = (NSDictionary
    ➥*)ABMultiValueCopyValueAtIndex(streetAddresses, 0);

    NSString *street = [streetAddressDictionary objectForKey:
    ➥(NSString *)kABPersonAddressStreetKey];

    NSString *city = [streetAddressDictionary objectForKey:
    ➥NSString *)kABPersonAddressCityKey];

    NSString *state = [streetAddressDictionary objectForKey:
    ➥(NSString *)kABPersonAddressStateKey];

    NSString *zip = [streetAddressDictionary objectForKey:
```

```
        ➥(NSString *)kABPersonAddressZIPKey];

        NSLog(@"Address: %@ %@, %@ %@", street, city, state, zip);

        CFRelease(streetAddressDictionary);
    }
```

表 5-2 地址组件

属　性	描　述
kABPersonAddressStreetKey	街道名称和号码，包括所有的公寓号码
kABPersonAddressCityKey	城市名称
kABPersonAddressStateKey	两个字母组成的州名或全称
kABPersonAddressZIPKey	ZIP 代码，5 位或 9 位
kABPersonAddressCountryKey	国家全名
kABPersonAddressCountryCodeKey	两个字母组成的国家代码

5.5 Address Book 图形用户界面

Address Book 框架提供了一个标准的用户界面，本节将介绍该界面并学习如何通过它节省大量的执行时间。比如是否正在编辑一个存在的联系人、是否正在创建一个新的联系人或是否允许用户从列表中选择一个联系人等，苹果公司都能满足你的需求。

人员选择器

开发者毫无疑问希望他的用户可以很容易地在列表中选择一个联系人。比如现在设计一款应用，主要功能是通过蓝牙向其他用户发送 vCard(电子名片)，需要让用户选择所希望发送的电子名片。使用 ABPeoplePickerNavigationController 类可以很容易实现上述功能。在 RootViewController.m 类中将第 63 行代码([self showPicker: nil];)的注释去掉就可以启用该功能。

ABPeoplePickerNavigationController 需要首先实现 ABPeoplePickerNavigationControllerDelegate 委托，之后使用下面的代码段创建一个新的选择器控制器(picker controller)：

```
ABPeoplePickerNavigationController *picker =
[[ABPeoplePickerNavigationControlleralloc] init];

picker.peoplePickerDelegate = self;
➥[self presentViewController:picker animated:YES completion:nil];
```

此时会向用户显示一个人员选择器，如图 5-3 所示。

还需要实现 3 个委托方法，处理用户与人员选择器交互操作的回调。首先需要处理的是用户点击的 Cancel 按钮，如果没有在这个方法中将模式框移除，用户就无法在视图中将其移除。

```
-(void)peoplePickerNavigationControllerDidCancel:(ABPeoplePickerNavigationController
➥*)peoplePicker
{
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

当选择人员时，可能需要涉及两个数据集。第一个是联系人本身及其所有的信息，第二个是一些特定的属性，比如联系人的特定电话号码或电子邮箱，首先需要选择有关个人联系信息的实体。

```
-(BOOL)peoplePickerNavigationController:
➥(ABPeoplePickerNavigationController *)peoplePicker
➥shouldContinueAfterSelectingPerson:(ABRecordRef)person
{
    NSLog(@"You have selected:%@", person);

    [self dismissViewControllerAnimated:YES completion:nil];

    return NO;
}
```

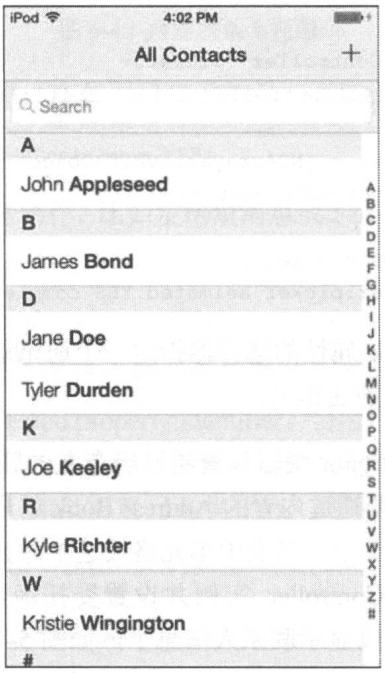


图 5-3　内置的人员选择器

在这个代码段中，peoplePickerNavigationController:shouldContinueAfterSelectingPerson:方法返回 NO。该方法会通知选择器——你不希望进一步获取联系人信息而只是选择一个 ABRecordRef 对象。同上一个示例类似，当完成选择时必须移除模式视图控制器。如果需要进一步访问详情，peoplePickerNavigationController:shouldContinueAfterSelectingPerson:方法需

要返回 YES 并实现相应的委托方法。如果需要更深入的细节，不要忘记移除前面对方法 dismissModalViewControllerAnimated:completion 的调用。

```
-(BOOL)peoplePickerNavigationController:
➥(ABPeoplePickerNavigationController *)peoplePicker
➥shouldContinueAfterSelectingPerson:(ABRecordRef)person
➥property:(ABPropertyID)property
➥identifier:(ABMultiValueIdentifier)identifier
{

   NSLog(@"Person: %@\nProperty:%i\nIdentifier:%i", person,property,
➥identifier);

   [self dismissViewControllerAnimated:YES completion:nil];

   return NO;
}
```

自定义人员选择器

你可能经常会希望选择器只从电话号码或街道地址进行选择而忽略其他信息，可以通过修改之前创建人员选择器的方法来满足这一需求，如下所示，这里只将电话号码作为选择依据：

```
ABPeoplePickerNavigationController *picker =
➥[[ABPeoplePickerNavigationController alloc] init];

picker.displayedProperties = [NSArrayarrayWithObject:[NSNumber
➥numberWithInt:kABPersonPhoneProperty]];

picker.peoplePickerDelegate = self;
[selfpresentViewController:picker animated:YES completion:nil];
```

还可以为使用 addressBook 属性的选择器指定一个通讯录，如果不设置该属性，当人员选择器出现时就会创建一个新的通讯录。

使用 ABPersonViewController 编辑和查看已经存在的联系人

大多数时候，你可能只需要通过内置的 Address Book 用户界面简单查看或编辑一个已经存在的联系人对象。在示例程序中，当表中单元格被选中时会默认进行上述动作。首先需要创建一个新的 ABPersonViewController 实例并设置委托函数和要显示的联系人，即一个 ABRecordRef 实例。这样就可以显示联系人信息了，如图 5-4 所示。

```
ABPersonViewController *personViewController = [[ABPersonViewController
➥alloc] init];

personViewController.personViewDelegate = self;

personViewController.displayedPerson = personToDisplay;
```

```
[self.navigationControllerpushViewController:personViewController
➥ animated:YES];
```

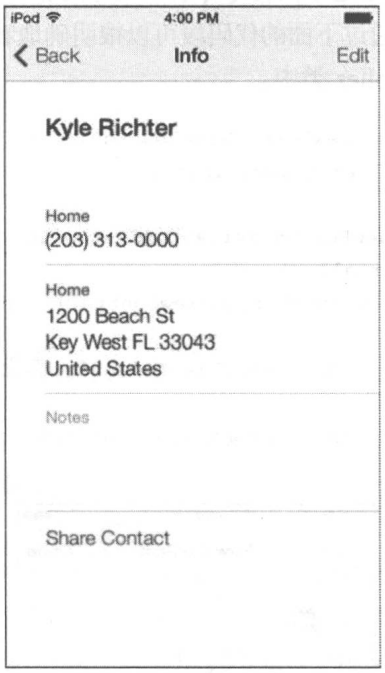


图 5-4　内置联系人视图

如果希望对联系人进行编辑，在代码段中添加另一个属性即可。

```
personViewController.allowsEditing = YES;
```

如果希望对联系人添加一些动作，比如发送短消息或 FaceTime 按钮等，可以添加一个额外的 allowsActions 属性。

```
personViewController.allowsActions = YES;
```

除了上面实现的步骤之外，还要注意一个需要的委托方法——personViewController:shouldPerformDefaultActionForPerson:property:identifier。当用户点击具体条目，比如街道地址或电话号码时就会调用该方法。如果希望应用执行默认的动作，比如调用或打开 Maps.app 程序，则在该方法中返回 YES；如果要重写这些行为，则返回 NO。

```
-(BOOL)personViewController:(ABPersonViewController
➥*)personViewController
➥shouldPerformDefaultActionForPerson:(ABRecordRef)person
➥property:(ABPropertyID)property
➥identifier:(ABMultiValueIdentifier)identifierForValue
{

    return YES;
}
```

使用 ABNewPersonViewController 创建新的联系人

当需要新建一个联系人时，通过示例程序导航栏上方的加号按钮可以使用苹果公司内置的用户界面来创建，如图 5-5 所示。通过下面的代码段可以很明确地看出 ABNewPersonViewController 必须封装在 UINavigationController 类中。

```
ABNewPersonViewController *newPersonViewController =
➥[[ABNewPersonViewController alloc] init];

UINavigationController *newPersonNavigationController =
➥[[UINavigationController alloc]
➥initWithRootViewController:newPersonViewController];

[newPersonViewController setNewPersonViewDelegate:self];

[self presentViewController:newPersonNavigationController animated:YES
➥completion:nil];
```

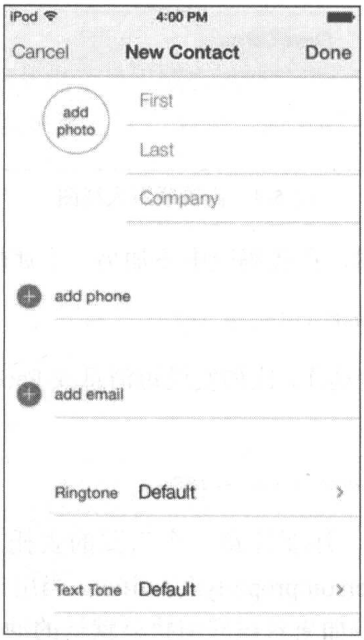


图 5-5 内置的新用户视图控制器

当用户保存联系人信息时会调用一个单独的委托。在验证所返回的联系人对象有效后，需要在想要添加联系人的通讯录中先后调用 ABAddressBookAddRecord 和 ABAddressBookSave 方法。如果类似示例程序那样，通过通讯录中的条目填充创建的数组对象，就可以重新生成该数组以观察其内容的变化。

```
-(void)newPersonViewController:(ABNewPersonViewController
➥*)newPersonViewController didCompleteWithNewPerson:(ABRecordRef)person
{
    if(person)
    {
        CFErrorRef error = NULL;
```

```
            ABAddressBookAddRecord(addressBook, person, &error);
            ABAddressBookSave(addressBook, &error);
            if(error != NULL)
            {
                NSLog(@"An error occurred");
            }
        }

        [self dismissViewControllerAnimated:YES completion:nil];

}
```

5.6　编写代码来创建联系人

如果希望用编写代码的方式创建一个新的联系人，而不是使用系统内置的界面来创建，需要怎么做呢？再想想那个共享联系人的应用。当编写代码将联系人信息输入通讯录时就不需要用到界面了。

在示例项目中，取消 RootViewController.m 文件中第 66 行的注释([self programmatically-CreatePerson];)并运行，你会注意到联系人列表中多了一个新人。创建一条新记录的第一步需要生成一个新的空 ABRecordRef 对象，这要用到方法 ABPersonCreate()。还要创建一个新的 NULL 指针 CFErrorRef。

```
ABRecordRef newPersonRecord = ABPersonCreate();

CFErrorRef error = NULL;
```

设置单值属性很简单，调用 ABRecordSetValue 方法且将第一个参数设置为 ABRecordRef，之后是属性常量、值还有 CFErrorRef 的地址。

```
ABRecordSetValue(newPersonRecord, kABPersonFirstNameProperty, @"Tyler",
➥&error);

ABRecordSetValue(newPersonRecord, kABPersonLastNameProperty, @"Durden",
➥&error);

ABRecordSetValue(newPersonRecord, kABPersonOrganizationProperty,
➥@"Paperstreet Soap Company", &error);

ABRecordSetValue(newPersonRecord, kABPersonJobTitleProperty,
➥@"Salesman", &error);
```

设置电话号码多值属性比设置单值对象稍微复杂一点，首先使用 ABMultiValueCreate-Mutable()方法创建一个新的 ABMutableMultiValueRef 对象，多值属性的具体值根据需要进行设置，在本例中为电话号码属性。

在示例程序中，创建了 3 个不同的电话号码，每一个都有不同的 label 属性。添加完所有的电话号码后，需要调用 ABRecordSetValue 并将新用户记录、设置好的多值常量和刚刚生成

的可变引用作为参数传递。完成这些操作之后不要忘记释放 Core Foundation 内存。

```
ABMutableMultiValueRef multiPhoneRef =
ABMultiValueCreateMutable(kABMultiStringPropertyType);

ABMultiValueAddValueAndLabel(multiPhoneRef, @"1-800-555-5555",
➥kABPersonPhoneMainLabel, NULL);

ABMultiValueAddValueAndLabel(multiPhoneRef, @"1-203-426-1234",
➥kABPersonPhoneMobileLabel, NULL);

ABMultiValueAddValueAndLabel(multiPhoneRef, @"1-555-555-0123",
➥kABPersonPhoneIPhoneLabel, NULL);

ABRecordSetValue(newPersonRecord, kABPersonPhoneProperty,
➥multiPhoneRef, nil);

CFRelease(multiPhoneRef);
```

邮箱地址的处理方法同电话号码类似。示例程序也给出了一个邮箱记录的例子。不过街道地址的处理稍微有些不同。

仍然需要创建一个新的可变多值引用，不过在这一步还需要创建一个新的可变 NSDictionary 对象。为每个需要设置的地址 key 分配一个对象(该值的完整列表参见表 5-2)。之后需要为这个街道地址添加一个标签，在下面的代码示例中使用的是 kABWorkLabel。完成之后，像之前保存电话号码和电子邮箱一样保存这些数据。

```
ABMutableMultiValueRef multiAddressRef =
➥ABMultiValueCreateMutable(kABMultiDictionaryPropertyType);

NSMutableDictionary *addressDictionary = [[NSMutableDictionary alloc]
➥init];

[addressDictionary setObject:@"152 Paper Street" forKey:(NSString *)
➥kABPersonAddressStreetKey];

[addressDictionary setObject:@"Delaware" forKey:(NSString
➥*)kABPersonAddressCityKey];

[addressDictionary setObject:@"MD" forKey:(NSString
➥*)kABPersonAddressStateKey];

[addressDictionary setObject:@"19963" forKey:(NSString
➥*)kABPersonAddressZIPKey];

ABMultiValueAddValueAndLabel(multiAddressRef, addressDictionary,
➥kABWorkLabel, NULL);

ABRecordSetValue(newPersonRecord, kABPersonAddressProperty,
➥multiAddressRef, &error);
```

```
CFRelease(multiAddressRef);
```

在把联系人所需的所有信息设置好之后，需要保存并检查是否出现错误。在示例程序中，会重新加载数组和表，以显示新的条目。

```
ABAddressBookAddRecord(addressBook, newPersonRecord, &error);
ABAddressBookSave(addressBook, &error);

if(error != NULL)
{
    NSLog(@"An error occurred");
}
```

5.7　小结

本章介绍了 Address Book 框架以及如何在 iOS 应用中使用该框架。我们学习了 Address Book 有关个人隐私和一些约束限制的知识，并强调一定要在应用真正需要使用通讯录时才使用该框架。

通过示例程序，我们掌握了如何快速添加并运行 Address Book 框架的有关功能。此外，还学习了如何从通讯录获取数据以及将新数据插入到通讯录的方法，前者采用的是苹果公司提供的图形用户界面，后者则使用编写编码的方式。现在你应该对 Address Book 框架的有关知识有了比较深入的了解并能够在 iOS 应用中运用自如了。

第6章

Music Libraries 框架

2007 年的 Macworld 大会上，Steve Jobs 第一次将 iPhone 推上舞台，成为一台全新的电话、iPod 和革命性的互联网连接设备。几年之后，经过第三方开发者的不懈努力，iPhone 的发展远远超过了这 3 个核心功能。尽管如此，传统的营销信息并没有改变，iPhone 最主要的功能仍然是电话、iPod 和互联网连接设备。用户并不是把 iPhone 加入到每天携带的众多设备之中，而是把手机和 iPod 合成了 iPhone 这一台设备。

当 2004 年苹果公司开始设计 iPhone 之初，就打算从音乐功能入手，iPhone 在 iPod 之上进行了革命性的进步。音乐给了 iPod 灵感，被认为是将苹果公司从低谷拉回来的功臣。没有人不爱音乐，它将人们聚在一起并充分表达自己的情感。尽管随着时间的流逝，用户已经不再把 iPhone 看成一款强大的音乐播放设备，但大部分人仍会在空闲时间信手拈来用 iPhone 听听自己喜欢的歌曲。

本章讨论如何在 iOS 应用中添加对用户音乐库的访问。不管是创建一个功能齐全的音乐播放器还是为游戏添加背景音乐，通过本章的学习你都能够掌握如何从用户自己的音乐库中播放音乐。

6.1 示例程序

本章的示例程序是一个简单的播放器(如图 6-1 所示)。该示例程序是一个功能齐全的音乐播放器，支持用户通过 Media Picker 选择需要播放的歌曲、播放随机歌曲、播放指定歌手的音乐。此外还具有暂停、恢复、前进、后退、音量调节、播放计数和播放前进或后退 30 秒等功能。如果有专辑图片的话，该应用还会为当前播放的音乐文件显示图片。

由于 Xcode 绑定的 iOS 模拟器中没有 Music.app，也没有转换音乐文件的方法，因此应用只能在真机上运行。当应用在模拟器上启动时就会出现很多错误。

```
player[80633:c07] MPMusicPlayer: Unable to
➥launch iPod music player server: application not found
player[80633:c07] MPMusicPlayer: Unable to
➥launch iPod music player server: application not found
```

```
player[80633:c07] MPMusicPlayer: Unable to
➥launch iPod music player server: application not found
```

所有试图访问媒体库的操作都会导致模拟器崩溃，出现如下错误提示：

```
*** Terminating app due to uncaught exception
➥'NSInternalInconsistencyException', reason: 'Unable to load
➥iPodUI.framework'
```

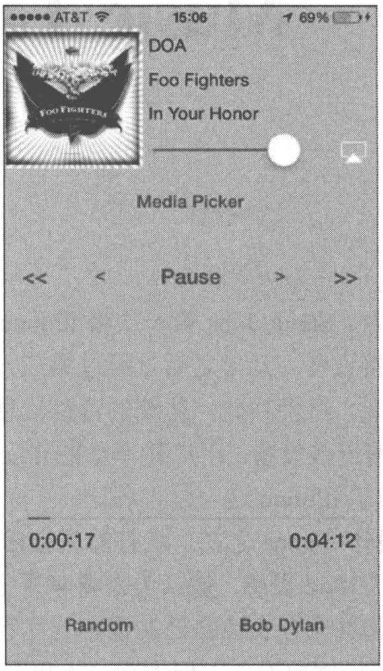


图 6-1 示例程序：iPod 上一个功能齐全的音乐播放器

6.2 创建播放引擎

在使用任何音频数据之前，需要详细了解一下有关播放控制的知识。从应用中播放音乐，首先需要创建一个新的 MPMusicPlayerController 实例。这一步在头文件 ICFViewController.h 中完成，新的对象叫作 player。MPMusicPlayerController 类会贯穿本章所有内容，用于实现控制播放和获取有关播放条目的信息等功能。

```
@interface ICFViewController:UIViewController
{
    MPMusicPlayerController *player;
}
```

在 viewDidLoad 方法中，MPMusicPlayerController 播放器可以使用 MPMusicPlayerController 类方法初始化。当创建 MPMusicPlayerController 对象时有两个选项。第一个选项 applicationMusicPlayer 播放应用中的音乐，不影响 iPod 状态并在应用退出时结束播放。第二个选项 iPodMusicPlayer 会控制 iPod 应用本身，用于确定用户指定的 iPod 播放指针和所选择的音乐，在应用进入后台时会继续播放。示例程序使用 applicationMusicPlayer，不需要更改

任何代码或行为就可以在这两种选项间切换。

```
-(void)viewDidLoad
{
    [super viewDidLoad];

    player = [MPMusicPlayerController applicationMusicPlayer];
}
```

6.2.1　注册播放通知

为了让音乐播放更有效率，时刻注意音乐播放器的状态很重要。当处理音乐播放器时，需要注意两个通知。“正在播放中”控件的变化和播放状态的变化。这些状态可以使用 NSNotificationCenter 进行监视并作用于之前的事件。示例程序使用一个新的便捷方法 registerMediaPlayerNotifications，目的是保持代码的整洁性和可读性。当把一个新的观察对象添加到 NSNotificationCenter 时，就需要在播放对象上调用 beginGeneratingPlaybackNotifications。

```
-(void)registerMediaPlayerNotifications
{
    NSNotificationCenter *notificationCenter =  [NSNotificationCenter
    ➥defaultCenter];

    [notificationCenter addObserver:self
                           selector:@selector
                              (nowPlayingItemChanged:)
                               name:
➥MPMusicPlayerControllerNowPlayingItemDidChangeNotification
                             object:player];

    [notificationCenter addObserver:self
                           selector:@selector
                              (playbackStateChanged:)
                               name:
➥MPMusicPlayerControllerPlaybackStateDidChangeNotification
                             object:player];

    [player beginGeneratingPlaybackNotifications];
}
```

当注册通知时，需要重点注意的是在对内存和视图进行清理时应该可以确保能够删除该通知，不这样做的话可能会导致程序崩溃和其他不可预知的错误。在 viewDidUnload 执行时还需要执行一个 endGeneratingPlaybackNotifications 调用。

```
-(void)viewWillDisappear:(BOOL)animated
{
    [[NSNotificationCenter defaultCenter] removeObserver:self
                                                    name:
➥MPMusicPlayerControllerNowPlayingItemDidChangeNotification
                                                  object:player];
```

```
    [[NSNotificationCenter defaultCenter] removeObserver:self
                                                    name:
➥MPMusicPlayerControllerPlaybackStateDidChangeNotification
                                                  object:player];

    [player endGeneratingPlaybackNotifications];

    [super viewWillDisappear:animated];
}
```

除了从音乐播放器注册回调之外，还需要创建一个新的 NSTimer 来处理更新播放进度和播放指针时间标签。在示例程序中，NSTimer 会调用 playbackTimer 方法。不过当下，通知回调 seletors 和 NSTimer 还没有完全实现。这一内容在后面的小节“处理状态改变”中会有介绍。

6.2.2 用户控制

示例程序为用户提供了很多按钮用于控制音乐，比如播放、暂停、跳过和切换到上一首，还有前进 30 秒和后退 30 秒等。第一个需要实现的方法是播放和暂停。按钮的功能就是一个简单的触发器：如果音乐正在播放，则暂停播放；如果音乐处于暂停状态，则恢复播放。代码会更新按钮对应的文本，从播放变为暂停或者从暂停变为播放，这其中也涉及状态变化通知回调的内容，在后面小节“处理状态改变”中会有相关内容的介绍。

```
-(IBAction)playButtonAction:(id)sender
{
    if([player playbackState] == MPMusicPlaybackStatePlaying)
    {
        [player pause];
    }

    else
    {
        [player play];
    }
}
```

当音乐播放时用户还可以直接跳到上一首歌或下一首歌，实现该功能需要对 player 对象额外调用两个方法。

```
-(IBAction)previousButtonAction:(id)sender
{
    [player skipToPreviousItem];
}

-(IBAction)nextButtonAction:(id)sender
{
```

```
    [player skipToNextItem];
}
```

用户还可以在一首歌中前进 30 秒或后退 30 秒，如果正好到音乐结尾，则跳到下一首歌。同样，如果向前超过了音乐的起始位置，则重新开始播放该音乐。这两个方法都使用了 player 对象的 currentPlaybackTime 属性。这个属性用于改变当前播放头指针，同时确定当前播放时间。

```
-(IBAction)skipBack30Seconds:(id)sender
{
    int newPlayHead = player.currentPlaybackTime - 30;

    if(newPlayHead < 0)
    {
        newPlayHead = 0;
    }

    player.currentPlaybackTime = newPlayHead;
}

-(IBAction)skipForward30Seconds:(id)sender
{
    int newPlayHead = player.currentPlaybackTime + 30;

    if(newPlayHead > currentSongDuration)
    {
        [player skipToNextItem];
    }

    else
    {
        player.currentPlaybackTime = newPlayHead;
    }
}
```

除了这些标准的控件用于实现播放之外，示例程序还可以调节音乐的音量。创建一个 MPVolumeView 对象用于实现音量调节功能，MPVolumeView 会提供一个滑动条并在合适时展示 Airplay 控件。

```
-(void)createAndDisplayMPVolumeViews
{

    UIView *volumeHolder = [[UIView alloc] initWithFrame:CGRectMake(125, 115, 185,
    ➥20)];
    [volumeHolder setBackgroundColor:[UIColor clearColor]];
    [self.view addSubview:volumeHolder];


    MPVolumeView *myVolumeView = [[MPVolumeView alloc] initWithFrame:volumeHolder.
```

```
        ➥bounds]
        [volumeHolder addSubview:myVolumeView];
    }
```

6.2.3 处理状态改变

本节之前注册了 3 个通知，用于接收回调方法。这些通知支持应用确定当前的状态和 MPMusicPlayerController 的行为。第一个被监视的方法在当前播放对象发生变化时被调用。这个方法包含两部分内容，一部分用于更新演唱者图片，另一部分用于更新表示作者、歌曲标题和图片的标签。

所有通过 MPMusicPlayerController 播放的音频或视频对象都由 MPMediaItem 对象表示。可以通过在 MPMusicPlayerController 实例上激活 nowPlayingItem 方法获取该对象。在 nowPlayingItemChanged 方法中可以看到该功能。

新建一个用于表示作品专辑图片的 UIImage 并将其初始化为一个占位符，在用户还没有为 MPMediaItem 设置专辑照片时暂时使用这个占位符。MPMediaItem 使用键值属性存储数据，完整的列表如表 6-1 所示。创建一个新的 MPMediaItemArtwork 对象并设置作品数据。虽然文档指出如果没有作者信息需要返回 nil，不过在 iOS 8 版本中实际不需要这么做。需要为作品载入一个 UIImage 并检查结果的值。如果是 nil，假设没有作品相册并载入占位符。示例程序中的代码即使在没有作品相册且 MPMediaItemArtwork 返回 nil 时也会继续执行。

表 6-1 MPMediaItem 常量

MPMediaItem 属性的键	可否用于谓词搜索
MPMediaItemPropertyPersistentID	YES
MPMediaItemPropertyAlbumPersistentID	YES
MPMediaItemPropertyArtistPersistentID	YES
MPMediaItemPropertyAlbumArtistPersistentID	YES
MPMediaItemPropertyGenrePersistentID	YES
MPMediaItemPropertyComposerPersistentID	YES
MPMediaItemPropertyPodcastPersistentID	YES
MPMediaItemPropertyMediaType	YES
MPMediaItemPropertyTitle	YES
MPMediaItemPropertyAlbumTitle	YES
MPMediaItemPropertyArtist	YES
MPMediaItemPropertyAlbumArtist	YES
MPMediaItemPropertyGenre	YES
MPMediaItemPropertyComposer	YES
MPMediaItemPropertyPlaybackDuration	NO
MPMediaItemPropertyAlbumTrackNumber	NO
MPMediaItemPropertyAlbumTrackCount	NO

(续表)

MPMediaItem 属性的键	可否用于谓词搜索
MPMediaItemPropertyDiscNumber	NO
MPMediaItemPropertyDiscCount	NO
MPMediaItemPropertyArtwork	NO
MPMediaItemPropertyLyrics	NO
MPMediaItemPropertyIsCompilation	YES
MPMediaItemPropertyReleaseDate	NO
MPMediaItemPropertyBeatsPerMinute	NO
MPMediaItemPropertyComments	NO
MPMediaItemPropertyAssetURL	NO
MPMediaItemPropertyIsCloudItem	YES

nowPlayingItemChanged:方法的第二部分处理歌曲标题、作者信息和相册名称的更新，如图 6-1 所示。如果这些属性都返回 nil，则设置占位字符串。MPMediaItem 可访问的属性列表参见表 6-1。查看该表时，注意如果媒体对象是 podcast，还可以使用其他键，这在苹果公司有关 MPMediaItem 的使用文档中会有介绍。该表还指示了当利用代码查找 MPMediaItem 时是否有可用的键。

```
-(void)nowPlayingItemChanged:(id)notification
{
    MPMediaItem *currentItem = [player nowPlayingItem];

    UIImage *artworkImage = [UIImage imageNamed:@"noArt.png"];

    MPMediaItemArtwork *artwork = [currentItem valueForProperty:
    ➥MPMediaItemPropertyArtwork];

    if(artwork)
    {
        artworkImage = [artwork imageWithSize:CGSizeMake(120,120)];

        if(artworkImage == nil)
        {
            artworkImage = [UIImage imageNamed:@"noArt.png"];
        }
    }

    [albumImageView setImage:artworkImage];

    NSString *titleString = [currentItem
    ➥valueForProperty:MPMediaItemPropertyTitle];
```

```
        if(titleString)
        {
            songLabel.text = titleString;
        }

        else
        {
            songLabel.text = @"Unknown Song";
        }

        NSString *artistString = [currentItem
        ➥valueForProperty:MPMediaItemPropertyArtist];

        if(artistString)
        {
                artistLabel.text = artistString;

        }

        else
        {
            artistLabel.text = @"Unknown artist";
        }

        NSString *albumString = [currentItem
        ➥valueForProperty:MPMediaItemPropertyAlbumTitle];

        if(albumString)
        {
             recordLabel.text = albumString;
        }

        else
        {
            recordLabel.text = @"Unknown Record";
        }
    }
```

监视音乐播放状态是非常关键的一步，因为这个值可能被应用控制之外的输入所影响。状态更新后，会激活 playbackStateChanged:方法。创建一个新的变量 playbackState 来保存播放器的当前状态。这个方法执行许多重要的任务，首先更新播放/暂停按钮来反映当前的播放状态。此外，“注册播放通知”一节中提到的 NSTimer 也会在这里创建和移除。当应用播放音频时，计时器被设置为每 0.3 秒激活一次，用于更新播放时长标签和 UIProgressIndicator 对象，通知用户播放指针的位置。计时器激活的方法 updateCurrentPlaybackTime 将在后面的小节中讨论。

除了示例程序中显示的状态之外，还有 3 个状态。第一个是 MPMusicPlaybackStateInterrupted，当音频被打断时使用，比如有电话拨入。另外两个状态分别是 MPMusicPlaybackStateSeeking-

Forward 和 MPMusicPlaybackStateSeekingBackward，用于指示音乐播放器正在向前或向后搜索。

```
-(void)playbackStateChanged:(id)notification
{
    MPMusicPlaybackState playbackState = [player playbackState];

    if(playbackState == MPMusicPlaybackStatePaused)
    {
          [playButton setTitle:@"Play"
          ➥forState:UIControlStateNormal];

      if([playbackTimer isValid])
      {
          [playbackTimer invalidate];
      }
    }

    else if (playbackState == MPMusicPlaybackStatePlaying)
    {
          [playButton setTitle:@"Pause" forState:
          ➥UIControlStateNormal];

          playbackTimer = [NSTimer
          scheduledTimerWithTimeInterval:0.3
          target:self
          selector:@selector(updateCurrentPlaybackTime)
          userInfo:nil
          repeats:YES];
    }

    else if (playbackState == MPMusicPlaybackStateStopped)
    {
          [playButton setTitle:@"Play"
          ➥forState:UIControlStateNormal];

          [player stop];

          if([playbackTimer isValid])
          {
              [playbackTimer invalidate];
          }
    }
}
```

对于音量变化这个事件，在应用中通过一个滑动条实时将其反映出来非常重要。这一功能是通过对 volumeChanged:通知进行监控实现的。可以从该方法得到播放音量并根据该数值对 volumeSlider 进行相应的设置。

```
-(void)volumeChanged:(id)notification
```

```
{
    [volumeSlider setValue:[player volume]];
}
```

6.2.4 时长和计时器

在大多数情况下，用户都希望得到有关歌曲的当前状态的信息，比如播放了多长时间和还剩多长时间结束，示例程序设计了两个方法用于生成这些数据。第一个方法是updateSongDuration，当切换歌曲或应用启动时调用。创建一个关于当前播放歌曲的引用，使用键值 playbackDuration 获取以秒为单位计算的歌曲播放时长。总的小时、分钟、秒通过该数据获得，歌曲持续时长用一个标签显示，位于 UIProgressIndicator 对象的旁边。

```
-(void)updateSongDuration;
{
    currentSongPlaybackTime = 0;

    currentSongDuration = [[[player nowPlayingItem] valueForProperty:
    ➥@"playbackDuration"] floatValue];

    NSInteger tHours = (currentSongDuration / 3600);
    NSInteger tMins = ((currentSongDuration / 60) - tHours*60);
    NSInteger tSecs = (currentSongDuration) - (tMins*60) - (tHours *3600);

    songDurationLabel.text = [NSString stringWithFormat:@"%zd:
    ➥%02d:%02d", tHours, tMins, tSecs ];

    currentTimeLabel.text = @"0:00:00";
}
```

第二个方法是 updateCurrentPlaybackTime，通过一个 NSTimer 对象每 0.3 秒调用一次，这由“处理状态改变”小节提到的方法 playbackStateChanged:控制。将同样的数学方法用于updateSongDuration 方法，以获取小时、分钟和秒。percentagePlayed 的计算也是基于之前确定的歌曲时长，它用于更新 playbackProgressIndicator 对象。由于 currentPlaybackTime 的精度只到 1 秒，因此不需要经常调用这个方法。不过调用越频繁，得到的结果就越精确。

```
-(void)updateCurrentPlaybackTime;
{
    currentSongPlaybackTime = player.currentPlaybackTime;

    int tHours = (currentSongPlaybackTime / 3600);
    int tMins = ((currentSongPlaybackTime / 60) - tHours*60);
    int tSecs = (currentSongPlaybackTime) - (tMins*60) - (tHours*3600);

    currentTimeLabel.text = [NSString stringWithFormat:@"%zd:
    ➥%02d:%02d", tHours, tMins, tSecs ];
```

```
    float percentagePlayed = currentSongPlaybackTime/
    currentSongDuration;

    [playbackProgressIndicator setProgress:percentagePlayed];
}
```

6.2.5　随机播放和循环播放

除了之前提到的一些属性和控件之外，MPMusicPlayerController 还具有循环播放和随机播放两个属性。虽然示例程序没有为这两个属性实现功能，不过它们很容易实现。

```
player.repeatMode = MPMusicRepeatModeAll;
player.shuffleMode = MPMusicShuffleModeSongs;
```

可用的循环模式有 MPMusicRepeatModeDefault，它是用户预设的参数。还有 MPMusicRepeatModeNone、MPMusicRepeatModeOne 和 MPMusicRepeatModeAll。随机播放可用的模式有 MPMusicShuffleModeDefault、MPMusicShuffleModeOff、MPMusicShuffleModeSongs 和 MPMusicShuffleModeAlbums，其中 MPMusicShuffleModeDefault 为默认使用的参数。

6.3　资源选择器

让用户选择喜欢听的歌曲的最简单办法就是让他们访问 MPMediaPickerController，如图 6-2 所示。MPMediaPickerController 允许用户浏览歌手、歌曲、播放列表和指定由一首或多首歌曲组成的唱片集。要使用 MPMediaPickerController 类，第一步需要指定其委托函数 MPMediaPickerControllerDelegate，这需要用到两个方法，其中的 mediaPicker:didPickMediaItems: 方法是在用户完成选择时调用。这些歌以 MPMediaItemCollection 对象的形式返回，MPMusicPlayerController 可以直接将这个对象作为 setQueueWithItemCollection:方法的参数。为 MPMusicPlayerController 设置好一个新的队列之后，就可以开始播放新的歌曲了。完成选择之后 MPMediaPickerController 不会自己消失，需要使用 dismissViewControllerAnimated:completion:方法将其删除。

```
-(void)mediaPicker:(MPMediaPickerController *) mediaPicker
➥didPickMediaItems:(MPMediaItemCollection *) mediaItemCollection
{
    if(mediaItemCollection)
    {
        [player setQueueWithItemCollection:mediaItemCollection];
        [player play];
    }

    [self dismissViewControllerAnimated:YES completion:nil];
}
```

图 6-2 在 iOS 8 中使用 MPMediaPickerController 选择歌曲

当用户在没有做出选择的情况下取消或删除 MPMediaPickerController 时，会调用 mediaPickerDidCancel:方法。作为方法完整性的一部分，开发者需要删除 MPMediaPickerController。

```
-(void)mediaPickerDidCancel:(MPMediaPickerController *) mediaPicker
{
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

在实现委托方法之后，创建一个 MPMediaPickerController 实例。在为 MPMediaPickerController 分配内存和进行初始化的过程中，需要一个描述所支持媒体类型的参数。表 6-2 中列出了所有可用的选项，注意，每个媒体对象都可以同多个媒体类型相关联。MPMediaPickerController 的其他一些参数还包括对多个对象指定一个选项，以及在选择期间显示一个提示框，如图 6-2 所示。还有一个布尔值属性用于设置是否显示 iCloud 对象，默认值为 YES。

```
-(IBAction)mediaPickerButtonAction:(id)sender
{
    MPMediaPickerController *mediaPicker =
    ➥[[MPMediaPickerController alloc] initWithMediaTypes:
    ➥MPMediaTypeAny];

    mediaPicker.delegate = self;
    mediaPicker.allowsPickingMultipleItems = YES;
    mediaPicker.prompt = @"Select songs to play";
```

```
    [self presentViewController:mediaPicker animated:YES completion:
    ➥nil];
}
```

表 6-2　创建 MPMediaPickerController 时用到的媒体类型参数

常　量	定　义
MPMediaTypeMusic	音乐对象的所有格式
MPMediaTypePodcast	音频广播对象
MPMediaTypeAudioBook	有声读物对象
MPMediaTypeAudioITunesU	iTunes U 音频格式
MPMediaTypeAnyAudio	音频格式
MPMediaTypeMovie	包含视频的媒体对象
MPMediaTypeTVShow	包含 TV show 的媒体对象
MPMediaTypeVideoPodcast	视频广播，不要同音频广播(MPMediaTypePodcast)相混淆
MPMediaTypeMusicVideo	音乐视频对象
MPMediaTypeVideoITunesU	iTunes U 视频对象，不要同 iTunes U 音频格式(MPMediaTypeAudio-ITunesU)混淆
MPMediaTypeAnyVideo	所有视频对象
MPMediaTypeAny	所有视频和音频对象

用户使用 MPMediaPickerController 选择歌曲时，以上步骤都是必需的，不过在很多情况下，为用户提供自定义的界面或无界面也是一种选择。下一节我们会介绍这些内容。

6.4　编程实现选择器

通常需要为用户提供更加个性化的音乐选择功能，包括创建自定义的音乐选择界面或者自动搜索歌手和唱片。本节我们讨论用编程的方式实现音乐的选择。

要实现不用 MPMediaPickerController 获取音乐的功能，需要创建一个新的 MPMediaQuery 对象并初始化。MPMediaQuery 函数存储着大量 MPMediaItem 对象，每一个都代表一首歌曲或一个音频对象。

示例程序给出了实现 MPMediaQuery 的两个方法。第一个方法 playRandomSongAction: 从用户音乐库中随机查找一首歌曲并使用已经存在的 MPMusicPlayerController 播放该音乐。使用代码查找音乐时，首先需要分配并初始化一个新的 MPMediaQuery 实例。

6.4.1　播放随机歌曲

如果不提供任何默认的参数，MPMediaQuery 将会包含所有从音乐库中找到的对象。创建一个新的 NSArray 对象用于保存这些对象，令 MPMediaQuery 使用对象方法获取相应的数据。每一个对象都由一个 MPMediaItem 表示。示例程序的随机歌曲功能同一时间只能播放一首歌曲。如果没有找到歌曲，会向用户显示一个 UIAlert 视图；如果找到多首歌曲，就随机

选择一首。

在找到一个(或多个)MPMediaItem 对象时，通过传入的 MPMediaItem 对象组成的数组创建一个新的 MPMediaItemCollection 对象。这个集合作为 MPMusicPlayerController 的播放列表，创建好这个集合之后，使用 setQueueWithItemCollection 将其传递给播放器对象。此时播放器就知道用户想听的歌曲是哪些了，之后调用播放方法播放 MPMediaItemCollection，播放顺序同创建 MPMediaItemCollection 数组的顺序相同。

```
-(IBAction)playRandomSongAction:(id)sender
{
    MPMediaItem *itemToPlay = nil;
    MPMediaQuery *allSongQuery = [MPMediaQuery songsQuery];
    NSArray *allTracks = [allSongQuery items];

    if([allTracks count] == 0)
    {
        UIAlertView *alert = [[UIAlertView alloc]
            initWithTitle:@"Error"
                  message:@"No music found!"
                 delegate:nil
        cancelButtonTitle:@"Dismiss"
        otherButtonTitles:nil];

        [alert show];
        return;
    }

    if([allTracks count] == 1)
    {
        itemToPlay = [allTracks lastObject];
    }

    int trackNumber = arc4random() % [allTracks count];
    itemToPlay = [allTracks objectAtIndex:trackNumber];

    MPMediaItemCollection * collection = [[MPMediaItemCollection
    alloc] initWithItems:[NSArray arrayWithObject:itemToPlay]];

    [player setQueueWithItemCollection:collection];
    [player play];

    [self updateSongDuration];
    [self updateCurrentPlaybackTime];
}
```

注意

arc4random()是标准 C 语言库中的一个方法，在 Objective-C 项目中用于生成一个随机数。同大多数随机数生成方法不同，arc4random()在第一次被调用时会自动生成种子数。

6.4.2　谓词匹配

通常应用不希望播放随机歌曲，而是提供更高级的搜索功能。使用关键词可以实现这个功能，下面的例子使用谓词查找音乐库中歌手属性等于“Bob Dylan”的对象，如图 6-3 所示。这个方法同之前的随机播放歌曲的方法很相似，只不过这里使用 addFilterPredicate 方法向 MPMediaQuery 添加筛选器。此外，结果并不会只筛选到一个单独的歌曲对象，传递给播放器对象的是一个包含所有匹配歌曲的一个数组。要查看所有关键字常量对应的完整列表，可以参见“处理状态变化”小节中表 6-1 的第 2 列。可以在 MPMediaQuery 中调用 addFilterPredicate 以添加多个谓词筛选。

```
-(IBAction)playDylan:(id)sender
{
    MPMediaPropertyPredicate *artistNamePredicate =
    [MPMediaPropertyPredicate predicateWithValue:@"Bob Dylan"
                                     forProperty:
                    MPMediaItemPropertyArtist];

    MPMediaQuery *artistQuery = [[MPMediaQuery alloc] init];

    [artistQuery addFilterPredicate:artistNamePredicate];

    NSArray *tracks = [artistQuery items];

    if([tracks count] == 0)
    {
        UIAlertView *alert = [[UIAlertView alloc]
        initWithTitle:@"Error"
        message:@"No music found!"
        delegate:nil
        cancelButtonTitle:@"Dismiss"
        otherButtonTitles:nil];

       [alert show];
       return;
    }

    MPMediaItemCollection * collection = [[MPMediaItemCollection
    ➥alloc] initWithItems:tracks];

    [player setQueueWithItemCollection:collection];

    [player play];

    [self updateSongDuration];
    [self updateCurrentPlaybackTime];
}
```

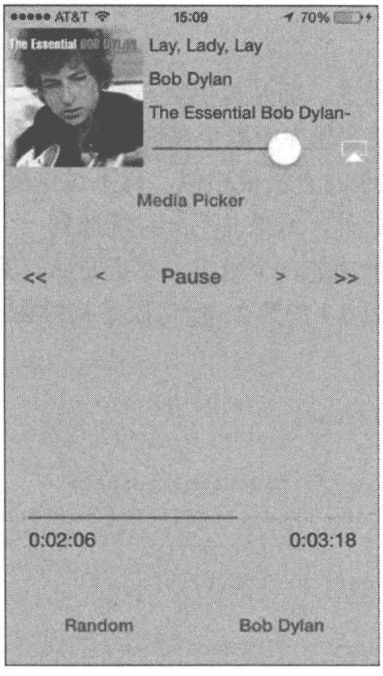


图 6-3　使用关键词查找 Bob Dylan 的歌曲

6.5　小结

本章学习了如何访问和使用用户的音乐库，首先创建一个播放引擎以实现用户和音乐的互动，比如暂停、恢复播放、控制音量和跳过等。接下来的两个小节介绍了访问和选择库中的音乐对象。媒体选择器介绍了使用内置 GUI 来实现用户选择一首歌曲或一组歌曲的功能，“程序实现的选择器”小节介绍了如何使用关键词查找和搜索歌曲。

示例程序演示了如何创建一个功能齐全但很简单的 iOS 音乐播放器。可以将本章介绍的知识应用于创建功能完整的音乐播放器或将用户的音乐库作为背景音频添加到任何应用中。

第 7 章

实现 HealthKit 框架

人们越来越关心个人健康问题，并希望使用技术获得更有价值的信息。智能手机的发展使其成为每个普通消费者的随身电脑，并且很快有关健身和健康的第三方应用如雨后春笋般出现了。

2013 年，与健康相关的移动应用的市场规模为 24 亿美元，并且预计到 2017 年会超过 200 亿美元。每个季度有超过 10 万个新的健康类应用在 iTunes App Store 上架，直到 HealthKit 出现之前这一应用领域还是很混乱的。由于 iOS 平台自然的沙盒机制，这些应用与其他的第三方应用之间无法共享数据，这样跟踪用户睡眠情况的应用就无法监测使用者的减重或运动情况，跑步应用也无法追踪用户营养的摄入。

HealthKit 通过为用户健康相关信息提供一个安全集中的环境来解决这一问题，这些信息从身体指数的测量到健身数据，甚至是食物摄入等应有尽有。不但允许第三方应用互相共享数据，更重要的是让用户在应用间切换变得更加容易。之前用户很容易被某一款应用绑定，因为该应用保存着他所有的数据，不过现在用户可以自由切换和选择新的解决方案，而不用担心历史数据丢失。对于开发者而言，这是再好不过的消息，因为新的健身和健康类应用在进入市场时原本面临着巨大障碍，如今终于能够蓬勃发展。

7.1 HealthKit 介绍

HealthKit 是在 2014 年苹果公司 WWDC 大会上同 iOS 8 一同发布的。最核心的是 HealthKit 是一个框架，主要支持应用共享和访问与健康和健身相关的数据。HealthKit 在所有新的 iOS 8 系统设备上还提供了一个系统应用，这个新的 Health 应用允许用户查看从所有第三方应用获取的整体数据集。

专门设计的 HealthKit 硬件设备还可以直接获取生命体征数据并直接同新的 iOS 设备交互，这是 M7 运动处理芯片的主要功能。有关硬件的知识不在本章的讨论范围之内，不过可以在 HealthKit 框架的介绍中找到更加详细的介绍，网址为 https://developer.apple.com/library/ios/documentation/HealthKit/Reference/HealthKit_Framework/index.html#//apple_ref/doc/uid/

TP40014707。

需要重点注意的是当使用 HealthKit 时，开发者会访问用户个人信息。HealthKit 是一个基于用户授权的服务，只有用户同意应用才能使用这些数据。应用成功访问这些数据之后，开发者需要对这些高度隐私化的数据负责。处理任何有关用户健康的数据都要像处理用户的信用卡或社交信息一样指定完善的安全方案。

7.2　Health.app 介绍

在编写带有 HealthKit 框架的应用之前，首先需要认识下 Health.app(如图 7-1 所示)，它是 iOS 8 自带的应用。Health.app 是用户健康数据的集中访问点。用户可以配置他们的 Dashboard 来查看感兴趣的测量值，如图 7-1 所示，用户查看的是步数、行走和跑步的距离以及睡眠分析这 3 方面的内容。Health Data 部分还可以让用户单独查看每一项的详细数据，也可以删除不需要的项。在 Sources 选项卡中用户可以配置哪些应用可以访问哪些数据，具体内容会在“请求授权 Health Data”一节中介绍。最后，在 Medical ID 选项卡中用户可以输入个人的一些重要信息，比如过敏情况、紧急联系人、血型和用药情况等。不输入用户密码也可以直接在锁屏界面上访问 Medical ID 信息。

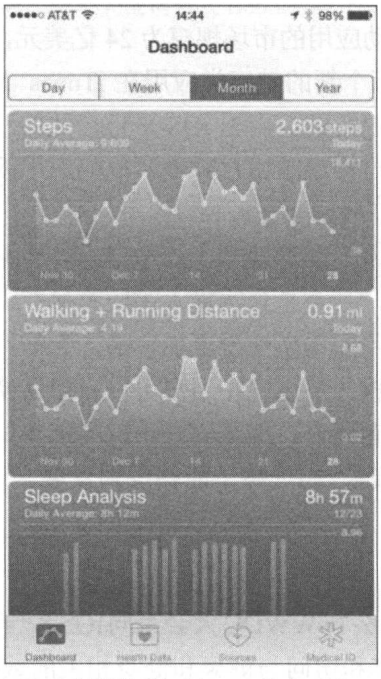


图 7-1　iOS 内置的 Health.app 的 Dashboard 部分

7.3　示例程序

本章的示例程序叫作 ICFFever，它是一个简单的基于 HealthKit 框架的应用，可以存储和获取基本的用户信息，比如年龄、身高和体重(如图 7-2 所示)。此外，该应用还展示了几种针对体温数据的操作，比如保存、获取、排序等(如图 7-3 所示)。为了避免过于混乱，示例程序的功能很简单，不过也能很好地展示与 HealthKit 数据进行交互的所有关键功能。

图 7-2　ICFFever 的个人信息界面

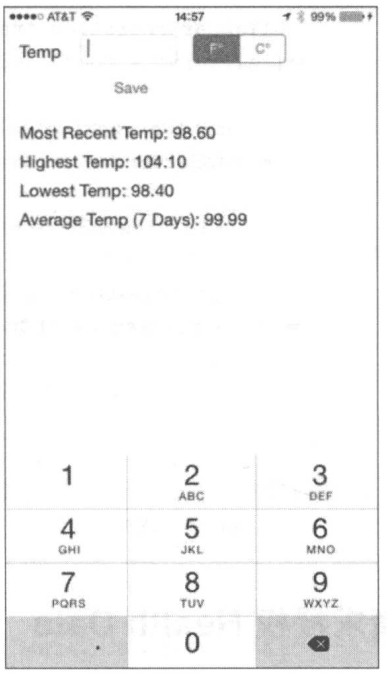


图 7-3　输入并查看体温信息

7.4　向项目添加 HealthKit

HealthKit 是一个独立的框架，所以对于一个新的项目而言首先需要进行配置，包括添加正确的授权、在应用的 infoplist 中添加 HealthKit 标签并关联 HealthKit 框架。苹果公司对在 Xcode 6 中正确设置这些内容进行了优化。当项目打开时，在项目导航栏中选择项目图标控件并找到 Capabilities 页面，开启 HealthKit 开关就会自动为项目添加上述配置了。

所有用到 HealthKit 的类，都需要导入该框架：

```
@import HealthKit;
```

HealthKit 是基于授权的服务，在应用访问或写入任何信息之前，需要得到用户的允许。为了实现这一过程，示例程序的委托函数中创建了一个单独的 HKHealthStore 实例。每个视图控制器都会逆向引用这个对象：

```
@property (nonatomic) HKHealthStore *healthStore;
```

ICFFever 是一个基于选项卡的应用，每个选项卡都需要访问 HKHealthStore 对象。在应用委托函数中需要提供一个在所有视图控制器中共享对象的方法。根据每个应用的侧重点不同，实现的方法可能不同。每个视图控制器都需要为 healthStore 创建自己的属性，如示例程序所示：

```
(BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
➥(NSDictionary *)launchOptions
{
        self.healthStore = [[HKHealthStore alloc] init];
```

```
    UITabBarController *tabBarController = (UITabBarController *)
➥[self.window rootViewController];

    for(UINavigationController *navigationController in
➥ tabBarController.viewControllers)
    {
        id viewController = navigationController;

        if([viewController respondsToSelector:@
➥ selector(setHealthStore:)])
        {
            [viewController setHealthStore:self.healthStore];
        }
    }

    return YES;
}
```

7.5 请求授权 Health Data

在完成使用 HealthKit 框架的基础工作之后，应用就可以请求访问特定健康数据的授权了。这一过程需要多步才能实现，首先需要进行检查以确保 HealthKit 在当前设备上是可用的。

```
if([HKHealthStoreisHealthDataAvailable])
```

下一步是生成一个包含所有需要读写的数据集列表。每一个信息点都必须专门请求，用户可以选择同意几个数据点，但是不能一次同意全部数据点。示例程序将列表的生成分为两个便捷的方法，每个方法返回一个 NSSet 对象。

```
// Returns the types of data that the app wants to write to HealthKit.
-(NSSet *)dataTypesToWrite
{
    HKQuantityType *heightType = [HKObjectType
➥ quantityTypeForIdentifier:HKQuantityTypeIdentifierHeight];

    HKQuantityType *weightType = [HKObjectType
➥ quantityTypeForIdentifier:HKQuantityTypeIdentifierBodyMass];

    HKQuantityType *tempType = [HKObjectType
➥ quantityTypeForIdentifier:HKQuantityTypeIdentifierBodyTemperature];

    return [NSSet setWithObjects:tempType, heightType, weightType, nil];
}

// Returns the types of data that the app wants wishes to read from HealthKit.
-(NSSet *)dataTypesToRead
{
   HKQuantityType *heightType = [HKObjectType
 ➥ quantityTypeForIdentifier:HKQuantityTypeIdentifierHeight];
```

```
    HKQuantityType *weightType = [HKObjectType
  ➥ quantityTypeForIdentifier:HKQuantityTypeIdentifierBodyMass];

    HKQuantityType *tempType = [HKObjectType
  ➥ quantityTypeForIdentifier:HKQuantityTypeIdentifierBodyTemperature];

    HKCharacteristicType *birthdayType = [HKObjectType
  ➥ characteristicTypeForIdentifier:HKCharacteristicTypeIdentifierDateOfBirth];

    return [NSSet setWithObjects:heightType, weightType,
  ➥ birthdayType, tempType, nil];
}
```

ICFFever 将会请求写入身高、体重和体温数据的授权，同样读取身高、体重、体温和生日信息也需要得到授权。值得注意的是应用可以只支持写入数据而禁止读取数据，或是只读取数据而禁止写入数据。

注意

HealthKit 包含数据点类型，通常称作特性(HKCharacteristicType)，这些特性包括性别、血型和生日。HealthKit 不允许第三方应用修改这些数据点，只能在 Health.app 程序中修改和初始化这些数据，不过第三方应用可以在用户允许的情况下读取这些数据。

HealthKit 目前支持超过 70 种数据类型，从最基本的体重数据到更加细致的数据，比如呼吸机的使用和氧气含量等。这些数据类型在 HealthKit 文档的汇总中都可以找到介绍，出于节省篇幅的目的，本章就不一一介绍它们了。

应用创建好两个 NSSet 后，其中一个用于读取数据，另一个用于写入数据，应用准备向用户发起授权请求。用户会看到一个请求使用 HealthKit 的界面(如图 7-4 所示)并且会选择希望授权的权限集。用户也可以随时在 Health.app 中对授权做进一步的修改。

```
if([HKHealthStore isHealthDataAvailable])
{
      NSSet *writeDataTypes = [self dataTypesToWrite];
      NSSet *readDataTypes = [self dataTypesToRead];

     [self.healthStore requestAuthorizationToShareTypes:writeDataTypes
    ➥readTypes:readDataTypes completion:^(BOOL success, NSError *error) {
        if(!success)
        {
            NSLog(@"HealthKit was not properly authorized to be added, check
          ➥ entitlements and permissions. Error: %@", error);

            return;
        }
    }];
}
```

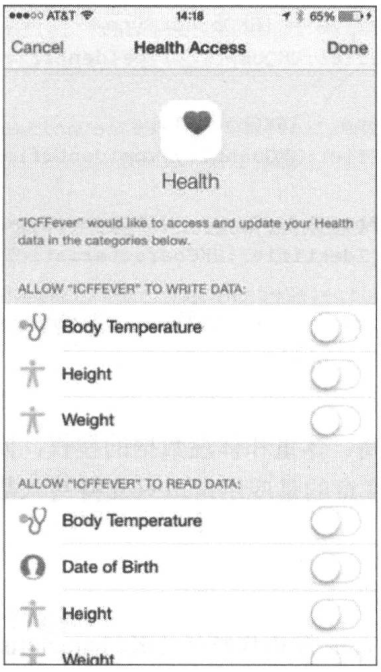


图 7-4 HealthKit 授权请求写入体温、身高、体重数据，以及读取体温、生日、身高和体重数据

7.6 读取 HealthKit 特征数据

假设用户已经授权从 Health.app 读取数据，应用现在可以开始使用已保存的数据了。示例程序的第一个标签页显示用户的基本个人信息。由于生日数据不可以由第三方应用修改，因此它是开始简单数据读取的逻辑位置。示例程序创建了一个 updateAge 方法，HealthKit 提供了一个快捷方法 dateOfBirthWithError，用于快速获取用户的生日数据。同样，对于其他特征数据，如性别和血型，也有类似的方法。该方法会进行基本的错误排查，如果数据没有找到，会提醒用户在 Health.app 中输入他们的生日。在取得 NSDate 数据之后，从中提取年份字段并显示给用户。

```
-(void)updateAge
{
    NSError *error = nil;
    NSDate *dateOfBirth = [self.healthStore dateOfBirthWithError:&error];

    if(!dateOfBirth)
    {
        NSLog(@"No age was found");
        dispatch_async(dispatch_get_main_queue(), ^{
                self.ageTextField.placeholder = @"Enter in HealthKit App";
        });
    }

    else
    {
        NSDate *now = [NSDate date];
```

```
        NSDateComponents *ageComponents = [[NSCalendar currentCalendar]
        components:NSCalendarUnitYear fromDate:dateOfBirth toDate:now
        options:NSCalendarWrapComponents];

        NSUInteger usersAge = [ageComponents year];

        self.ageTextField.text = [NSNumberFormatter localizedStringFromNumber:
      ➥ @(usersAge) numberStyle:NSNumberFormatterNoStyle];
    }
}
```

7.7　读写基本的 HealthKit 数据

示例程序除了读取身高和体重数据之外，还可以让用户在应用中更新这些数据。ICFFever 将上述功能分成两步实现，第一步先读取数据，第二步写入新的数据。要获取一个新的数据点，首先需要使用得到的数据点创建一个新的 HKQuanyityType 对象，即下例中的 HKQuantityTypeIdentifierBodyMass。

苹果公司很贴心地为获取最近一个条目类型提供了一个便捷方法，这个方法就包含在 HKHealthStore+AAPLExtensions.m 类中。假设 HealthKit 数据存储中已经具有体重数据，将会返回一个 HKQuantity 对象。示例程序会指定想要显示的数据。这里为 poundUnit 创建了一个 HKUnit 对象，其他数据可以返回的单位是克、盎司、英石或摩尔量。当明确需要以磅为单位时，就可以从 HKQuantity 对象获取一个双精度的值。随后，这个数据会在主线程中显示，因为 HealthKit 在后台线程上运行。

```
-(void)updateWeight
{
        HKQuantityType *weightType = [HKQuantityType
      ➥ quantityTypeForIdentifier:HKQuantityTypeIdentifierBodyMass];

        [self.healthStore aapl_mostRecentQuantitySampleOfType:weightType
      ➥predicate:nil
      ➥completion:^(HKQuantity *mostRecentQuantity, NSError *error)
        {
                if(!mostRecentQuantity)
                {
                        NSLog(@"No weight was found");

                        dispatch_async(dispatch_get_main_queue(), ^{
                                self.weightTextField.placeholder = @"Enter in lbs";
                        });
                }
        else
        {
                HKUnit *weightUnit = [HKUnit poundUnit];
                double usersWeight = [mostRecentQuantity
              ➥doubleValueForUnit:weightUnit];
```

```
            dispatch_async(dispatch_get_main_queue(), ^{
                self.weightTextField.text = [NSNumberFormatter
              ➥localizedStringFromNumber:@(usersWeight)
              ➥numberStyle:NSNumberFormatterNoStyle];
            });
        }
    }];
}
```

如果用户还没有保存他的体重信息，ICFFever 应用也可以支持用户保存新的体重数据。保存数据同获取数据类似，首先需要定义一个 HKUnit 来指定数据存储的单位。在示例程序中用户将会以磅为单位输入体重。

创建一个新的 HKQuantity 对象，该对象包含之前保存的单位和数值。之后创建一个 HKQuanityType，用于指定被保存的数据点类型。由于这里是体重数据，因此使用 HKQuantityTypeIdentifierBodyMass 常量。每个保存的数据点都带有时间戳信息，用于在之后进行比较，对于 ICFFever，假定输入的体重和身高数据都是当前的数据。使用类型、重量和日期范围等属性创建一个新的 HKQuantitySample 对象。保存对象调用 saveObject 方法并指定 completion block。可以使用上面介绍的方法将新输入的体重显示出来。

```
(void)saveWeightIntoHealthStore:(double)weight
{
    HKUnit *poundUnit = [HKUnit poundUnit];

    HKQuantity *weightQuantity = [HKQuantity quantityWithUnit:poundUnit
  ➥doubleValue:weight];

    HKQuantityType *weightType = [HKQuantityType
 ➥quantityTypeForIdentifier:HKQuantityTypeIdentifierBodyMass];

    NSDate *now = [NSDate date];

     HKQuantitySample *weightSample = [HKQuantitySample
    ➥quantitySampleWithType:weightType quantity:weightQuantity
    ➥startDate:now endDate:now];

     [self.healthStore saveObject:weightSample withCompletion:
    ➥^(BOOL success, NSError *error)
     {
         if(!success)
         {
             NSLog(@"An error occurred saving weight
           ➥(%@): %@.", weightSample, error);
         }

         [self updateWeight];
    }];
}
```

个人信息界面还允许用户保存和获取身高信息，这一过程同体重信息的处理类似，只是稍有不同。针对身高应用指定英寸作为单位，替换了之前的单位磅，HKQuantity 的类型为 HKQuantityTypeIdentifierHeight，替换了之前的 HKQuantityTypeIdentifierBodyMass。

7.8　读写复杂的 HealthKit 数据

前面小节讨论的是向 HealthKit 中写入以及从中读取非常基本的用户个人数据，比如身高和体重信息。本节将继续深入学习一些更加复杂的数据类型的处理方法，比如体温。ICFFever 应用的第二个标签页可以让用户保存当前的体温，可以使用摄氏温度或华氏温度作为单位。应用既会输出最近的体温数据，还会输出 7 天之内的最高、最低和平均体温。

HealthKit 提供了 3 种体温单位类型，分别是绝对温度、摄氏温度和华氏温度。由于应用可以在摄氏温度和华氏温度间切换单位，因此数据会使用绝对温度作为单位保存，并根据用户的选择进行转换。每次指定一种新的数据类型来保存或读取信息就可以解决这个问题。在学习新的办法前，使用一个统一的单位会减少开发者的困惑。下面的新方法将数据从绝对温度快速转换为相应的单位：

```
-(double)convertUnitsFromKelvin:(double)kelvinUnits
{
    double adjustedTemp = 0;

    //Kelvin to F
    if([self.unitSegmentedController selectedSegmentIndex] == 0)
    {
        adjustedTemp = ((kelvinUnits-273.15)*1.8)+32;
    }

    //Kelvin to C
    if([self.unitSegmentedController selectedSegmentIndex] == 1)
    {
        adjustedTemp = kelvinUnits-273.15;
    }

    return adjustedTemp;
}
```

第一个方法将会保存最近的温度数据，这一步同之前保存身高体重信息的方法非常类似。该方法首先将用户输入的任何单位的数据都转换为绝对温度数据。指定一个新的 HKUnit 并设置为 kelvinUnit。用绝对温度值创建 HKQuantity 对象，将类型设置为 HKQuantityTypeIdentifierBodyTemperature。数据录入还会再次用到当前的时间，之后将数据保存到 HealthKit 中。

```
-(void)updateMostRecentTemp:(double)temp
{
        double adjustedTemp = 0;

        //F to Kelvin
        if([self.unitSegmentedController selectedSegmentIndex] == 0)
```

```
        {
            adjustedTemp = ((temp-32)/1.8)+273.15;
        }

        //C to Kelvin
        if([self.unitSegmentedController selectedSegmentIndex] == 1)
        {
            adjustedTemp = temp + 273.15;
        }

        // Save the user's height into HealthKit.
        HKUnit *kelvinUnit = [HKUnit kelvinUnit];

        HKQuantity *tempQuainity = [HKQuantity quantityWithUnit:kelvinUnit
        doubleValue:adjustedTemp];

        HKQuantityType *tempType = [HKQuantityType
      ➥ quantityTypeForIdentifier:HKQuantityTypeIdentifierBodyTemperature];

        NSDate *now = [NSDate date];

        HKQuantitySample *tempSample = [HKQuantitySample
      ➥ quantitySampleWithType:tempType

        quantity:tempQuainity startDate:now endDate:now];

       [self.healthStore saveObject:tempSample withCompletion:
      ➥^(BOOL success, NSError *error)
        {
              if(!success)
              {
                    NSLog(@"An error occurred saving temp (%@): %@.",
                  ➥ tempSample, error);
               }

                [self updateTemp];
    }];

}
```

注意

创建好 HKQuantitySample 之后，调用 quantitySampleWithType:quantity:startDate:endDate:metadata:方法可以得到元数据。这个元数据遵循一些预定义的键，比如 HKMetadataKeyTimeZone、HKMetadataKeyWasTakenInLab、HKMetadataKeyBodyTemperatureSensorLocation 或是其他 20 种键。还可以创建自定义键，用于满足不同应用的需求。

虽然应用只可以保存当前体温值，不过用户还可以查看历史体温数据。这一过程同样需要 updateTemp 方法，和之前获取身高体重的方法一样。创建一个新的 HKQuantityType 并设

置为 HKQuantityTypeIdentifierBodyTemperature。再次用到苹果公司为返回最近条目数据提供的便捷方法。如果调用返回了数据，将其由绝对温度转换为用户在界面上选择的单位并显示给用户。

```
-(void)updateTemp
{
   HKQuantityType *recentTempType = [HKQuantityType
  ➥ quantityTypeForIdentifier:HKQuantityTypeIdentifierBodyTemperature];

   [self.healthStore aapl_mostRecentQuantitySampleOfType:recentTempType
  ➥predicate:nil completion:^(HKQuantity *mostRecentQuantity, NSError *error)
      {
            if(!mostRecentQuantity)
            {
                  NSLog(@"No temp was found");

                  dispatch_async(dispatch_get_main_queue(), ^{
                        self.recentTempLabel.text =
                       ➥@"Most Recent Temp: None Found";
                  });
            }

             else
            {
                  HKUnit *kelvinUnit = [HKUnit kelvinUnit];
                  double temp = [mostRecentQuantity
                 ➥ doubleValueForUnit:kelvinUnit];

                  dispatch_async(dispatch_get_main_queue(), ^{
                  self.recentTempLabel.text = [NSString stringWithFormat:
                 ➥@"Most Recent Temp: %0.2f",
                 ➥[self convertUnitsFromKelvin:temp]];
                  });
            }
      }];
```

不过这一结果值只为当前温度数据提供了一个单独的值，应用希望访问所有的温度数据以计算平均值、最低值和最高值。为了实现这个功能，创建一个新的方法并将其添加到 Apple 扩展文件中。新方法的核心是 HKSampleQuery。创建一个新的查询方法，接收的数据为 quantityType 对象。将 limit 设置为 0，表示返回所有符合指定类型的记录。结果还不能排序，因为没有传递有关排序的参数，此外，predicate 为 nil 表示没有任何过滤设置。结果将会返回一个包含 HKQuantitySample 对象的 NSArray 数组，每个元素表示一个体温值。

```
-(void)allQuantitySampleOfType:(HKQuantityType *)quantityType
➥ predicate:(NSPredicate *)predicate completion:
➥ (void(^)(NSArray *, NSError *))completion
{
   HKSampleQuery *query = [[HKSampleQuery alloc]
```

```
    ➥initWithSampleType:quantityType predicate:nil
    ➥limit:0 sortDescriptors:nil resultsHandler:^(HKSampleQuery *query,
    ➥NSArray *results, NSError *error) {

if(!results)
        {
            if(completion)
            {
                completion(nil, error);
            }

            return;
        }

        if(completion)
        {
            completion(results, error);
        }
    }];

      [self executeQuery:query];
}
```

可以对现存的 updateTemp 方法进行扩展，由于所有结果都是绝对温度，因此设置一个新的 HKUnit 来指定单位类型。还要创建一些变量来保存最大值、最小值、计数、相加和平均温度的值。该方法会遍历数组中的所有元素，每个 HKQuantitySample 对象都包含一个起始日期、一个最后日期和具体值。使用这些信息在对象之间进行比较得到最小值和最大值。此外，对过去的 604 800 秒(7 天)内所有的体温值进行相加并计算得到平均值。所有数据处理完之后，在主线程中更新标签。

```
[self.healthStore allQuantitySampleOfType:recentTempType predicate:nil
➥completion:^(NSArray *results, NSError *error)
{
        if(!results)
        {
                NSLog(@"No temp was found");
        }

        else
        {
                HKUnit *kelvinUnit = [HKUnit kelvinUnit];

                double max = 0;
                double min = 1000;

                double sum  = 0;
                int numberOfSamples = 0;
                double averageTemp = 0;
```

```
            for(int x = 0; x < [results count]; x++)
            {
                HKQuantitySample *sample = [results objectAtIndex:x];

                if([[sample quantity] doubleValueForUnit:kelvinUnit] > max)
                {
                    max = [[sample quantity]
                ➥doubleValueForUnit:kelvinUnit];
                }

                if([[sample quantity] doubleValueForUnit:kelvinUnit] < min)
                {
                    min = [[sample quantity]
                ➥ doubleValueForUnit:kelvinUnit];
                }

                //7 days' worth of seconds
                if([[sample startDate] timeIntervalSinceNow] < 604800.0)
                {
                    sum += [[sample quantity]
                ➥ doubleValueForUnit:kelvinUnit];
                    numberOfSamples++;
                }

            }

            averageTemp = sum/numberOfSamples;

            dispatch_async(dispatch_get_main_queue(), ^{
                self.highestTempLabel.text = [NSString stringWithFormat:
              ➥ @"Highest Temp: %0.2f", [self convertUnitsFromKelvin:max]];

                self.lowestTempLabel.text = [NSString stringWithFormat:
              ➥ @"Lowest Temp: %0.2f", [self convertUnitsFromKelvin:min]];

                self.avgTempLabel.text = [NSString stringWithFormat:
              ➥ @"Average Temp (7 Days): %0.2f", [self convertUnitsFromKelvin:
              ➥ averageTemp]];

        });
    }
}];
```

7.9　小结

HealthKit 是一个相当大的话题，包含非常多的数据，本章对诸如读取和写入等基本操作及原理进行了讲解，这些知识即使在复杂的案例中也非常适用。不管每天各种各样的应用通过 HealthKit 存取多少海量数据，现在我们都可以对这些数据进行解析和处理了。

目前 HealthKit 框架有超过 70 种不同的数据类型，并随着 iOS 版本的更新会扩展出更多

的数据类型。大家已经开始抱怨找不到标准的数据类型。本章和本书其余部分所提供的内容都只是为程序开发者提供了一个敲门砖。虽然对于 HealthKit 框架没有完整的说明，不过开发者使用该技术创建的各种应用已经远远超过苹果公司创建该框架之初的预想了。

第8章

实现 HomeKit 框架

iOS 8 版本中引入了 HomeKit 框架，为应用整合智能家居技术提供了一个统一的解决方案。应用不再需要根据不同的智能化硬件采用不同的处理方案，HomeKit 框架认证的设备(即使是不同生产厂家、使用不同的通信标准)都可以通过 HomeKit 应用进行管理。此外，HomeKit 信息只要在一台 iOS 设备上设置好之后，就可以在其他相同的设备上使用。用户只能为家居智能化设备设置一次信息，后面都会统一使用这些设置好的信息。

HomeKit 提供远程访问功能，用户可以从任何获得连接的位置使用智能家居技术同设备进行交互，并提供应用和智能家居设备之间的安全通信保障。

HomeKit 为组织和管理智能家居设备还提供了一些高级技术。比如，房间可以被设置为一些区域(例如楼上房间和楼下房间)，然后让指定区域的设备执行某动作(比如打开楼上房间的灯)。此外，HomeKit 还提供触发器，用于实现在指定的时间或循环执行某些动作。

HomeKit 交互只能发生在前端程序，不过由 iOS 维护的触发器是个例外。这么做可以让用户的安全感得到加强，不会担心应用在后台运行而导致的不可预期的结果。

8.1 示例程序

本章的示例程序名为 HomeNav，它支持将家庭添加到 HomeKit、向家庭添加房间、向房间中添加附件，并将附件和房间进行关联。可以查看这些附件，了解它们提供什么样的服务，每项服务的具体内容是什么，以及具体的数值信息。HomeNav 还可以更新电力设置和锁定状态，打开设备或关闭设备，以及打开或关闭大门。

8.2 HomeKit 介绍

HomeKit 为应用提供了一个统一的 API 接口，用于对智能设备进行设置和通信。值得反复强调的是，HomeKit 保存的关于家庭和相关设备的数据独立于每个应用，也就是说，所有的应用都可以使用 HomeKit。HomeKit API 只有在应用处于前端运行时才可以被访问，这可以避免多个应用同时在后台更新 HomeKit 数据而出现的潜在错误。

通过 HMHomeManager 类，HomeKit 可以访问家庭信息。使用这个类，应用可以获得有效的家庭数据(HMHome 实例)，并且可以在通过委托方法添加或删除家庭时得到通知信息。

HomeKit 中的家庭包含一些房间(HMRoom 实例)，可以被分为几个区域(HMZone)。房间可以属于多个区域，比如卫生间既属于楼上房间区域，也属于卫生间区域。

智能家居设备由一些附件(HMAccessory 实例)表示，必须通过 HMAccessoryBrowser 发现。这个浏览器会搜索本地支持 HomeKit 交互的 Wi-Fi 和蓝牙适配设备，并返回一个附件列表，用于显示和选择。附件被添加到家庭中之后，应用就可以找到它们并更新状态了。附件具有服务(HMService)特性，由各种特性(HMCharacteristic)组成。比如咖啡机附件可以提供制作咖啡的服务、指示灯服务(light service)和时钟服务(clock service)。咖啡机服务可能具有一个只读特性，包含当前是否正在酿制、加热器开启但没有酿制、加热器关闭没有酿制等。之后还会有一个可读特性和一个可写特性，它们包含咖啡机所需的状态信息，应用可以写入信息以更改咖啡机的状态。

修改特性可以按组分成动作集合(HMActionSet)，可以一次执行所有动作，或者设定一个时间表(HMTimerTrigger)。按时间表更新是之前规则的例外，不必一定要在应用前端执行，因为这个时间表是由 iOS 维护并执行的，在时间表更新时应用不需要处于前端。

当一个支持 HomeKit 的应用第一次访问 HomeKit 框架时(意味着当前没有在 HomeKit 中设置家庭信息)，应用会为用户提供一系列关于设置的步骤。也就是说，应用会提示用户设置一个家庭，添加房间和附件。示例程序会尽量简化这个过程，在没有任何设置的情况下提示用户添加家庭或房间信息。如果是在 App Store 上架的应用，就应该为用户提供简单清晰的步骤。

8.3 设置 HomeKit 组件

要让应用使用 HomeKit，首先在项目中启用 HomeKit 功能，开发者需要拥有一个有效的付费开发者账号，这样就可以为应用标识符添加授权。

8.3.1 设置开发者账号

Xcode 需要 iOS 开发者账号信息来连接到 Member Center，为开发者执行 HomeKit 所需的所有设置。选择 Xcode 菜单中的 Preferences，之后选择 Accounts 标签。添加一个新的账号，需要点击 Accounts 标签页左下角的加号并选择 Apple ID。在图 8-1 所示的对话框中输入账号认证信息并点击 Add 按钮。

Xcode 将会验证认证信息并收集有效的账户信息。一个有效的开发者账户配置完成后，Xcode 将执行有关 HomeKit 设置的步骤。

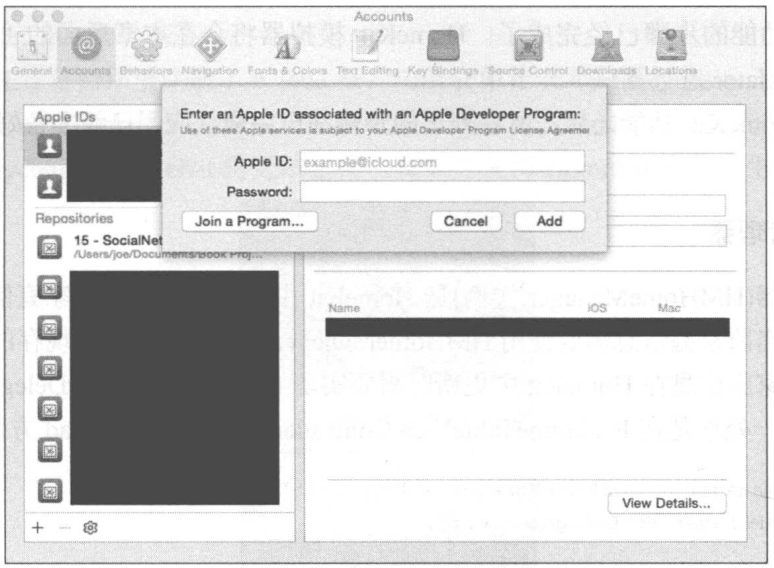


图 8-1　Xcode Accounts 标签页

8.3.2　启用 HomeKit 功能

要设置 HomeKit 的功能，需要查看 Xcode 中的 HomeNav Target，点击 Capabilities 标签，找到 HomeKit 部分。将 HomeKit 对应的选项变为 On，Xcode 将会自动为项目创建一个授权文件并使用 HomeKit 授权配置应用标识符，如图 8-2 所示(注意，在这个设置生效之前应用标识符将会由示例程序的应用标识符变更为其他的唯一标识符)。

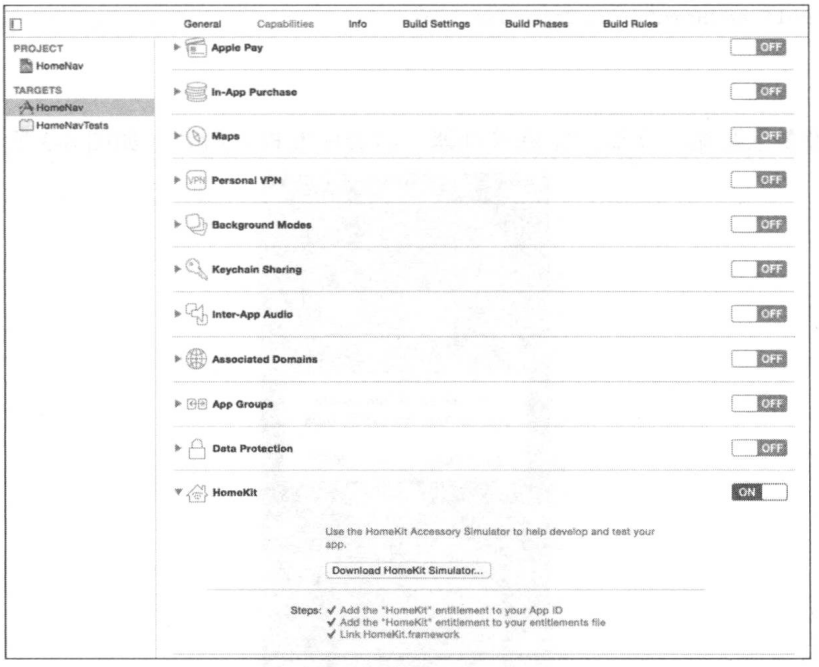


图 8-2　Xcode 目标功能——HomeKit

在启用 HomeKit 功能之后，Xcode 将会给出一个下载 HomeKit 模拟器的链接，并提示设

置 HomeKit 功能的步骤已经完成了。HomeKit 模拟器将会在本章后面的“使用 HomeKit Accessory Simulator 进行测试”一节中介绍。

在启用 HomeKit 功能之后，所有列出的步骤都已完成，应用已经准备好使用 HomeKit 的功能了。

8.3.3 家庭管理器

家庭管理器(HMHomeManager 实例)是 HomeKit 中唯一获取和更新家庭信息的途径。获取已经设置的当前家庭信息需要使用 HMHomeManager 实例，或者移除现存的家庭时也会用到。此外，当家庭信息在 HomeKit 中更新时需要实现 HMHomeManagerDelegate 协议。在示例程序中，这一处理是在 ICFHomeTableViewController 类的 viewDidLoad 方法中完成的。

```
self.homeManager = [[HMHomeManager alloc] init];
[self.homeManager setDelegate:self];
```

家庭管理器将会更新可用家庭的列表，当有可用的家庭列表时会调用委托方法 homeManagerDidUpdateHomes:。示例程序会重新载入表视图来显示最新的家庭信息。如果没有家庭被设置，示例程序将会提示用户创建一个新的家庭。

```
-(void)homeManagerDidUpdateHomes:(HMHomeManager *)manager {
    [self.tableView reloadData];

    if([manager.homes count] == 0)
    {
        [self addHomeButtonTapped:nil];
    }
}
```

如果在设备上是第一次调用家庭管理器，将会触发权限检查，如图 8-3 所示。

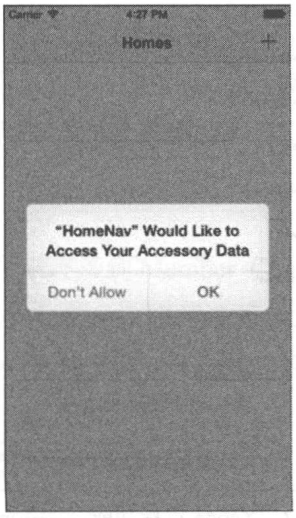


图 8-3 示例程序的 HomeKit 权限请求

选择 Don’t Allow 将会禁止 HomeKit 向应用提供任何信息，这一设置可以在 Settings.app

中的 Privacy,HomeKit 部分找到。

如果用户在设备商那里注册了 iCloud 账户但没有打开 iCloud Keychain，HomeKit 将会提示用户打开 iCloud Keychain，允许用户的所有 iOS 设备访问 HomeKit，如图 8-4 所示。如果没有 iCloud Keychain，HomeKit 将无法正常工作并对任何 HomeKit 操作产生错误提示。

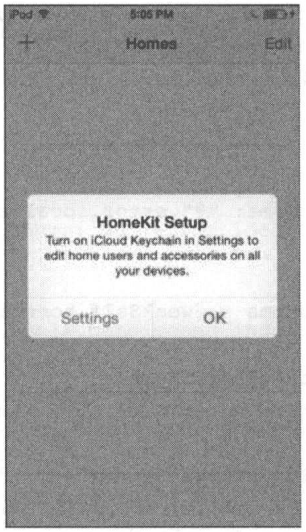


图 8-4　示例程序 HomeKit Setup 有关 iCloud Keychain 的提示框

提示

为了避免 HomeKit 中遇到的错误，Apple's Developer Web 网站的 HomeKit Constants Reference 中对可能出现的错误进行了详细描述。

8.3.4　家庭

如果在 HomeKit 中没有设置家庭，示例程序会提示用户添加一个家庭，如图 8-5 所示。用户还可以点击 Add(+)按钮随时添加一个新的家庭。

图 8-5　示例程序的 Add Home 对话框

点击 Add Home 按钮，程序将会获取用户在文本框中输入的家庭名称，并请求家庭管理器使用这个名称添加一个新的家庭。家庭名必须唯一。

```
UITextField *homeNameTextField = addHomeAlertController.textFields.firstObject;
NSString *newHomeName = homeNameTextField.text;
__weak ICFHomeTableViewController *weakSelf = self;
[self.homeManager addHomeWithName:newHomeName
            completionHandler:^(HMHome *home, NSError *error)
{
    if(error)
    {
        NSLog(@"Error adding home: %@",error.localizedDescription);
    } else
    {
        NSInteger rowForAddedHome = [weakSelf.homeManager.homes indexOfObject:home];

        NSIndexPath *indexPathForAddedHome =
        ➥[NSIndexPath indexPathForRow:rowForAddedHome inSection:0];

        [weakSelf.tableView insertRowsAtIndexPaths:@[indexPathForAddedHome]
                                withRowAnimation:UITableViewRowAnimationAutomatic];
    }
}];
```

家庭管理器将会调用带有错误或 HMHome 新实例的 completion handler。程序将会为新的 home 对象确定一个索引路径，并将其添加到视图中。home 对象现在可以用来设置房间和附件了。

如果用户错误设置了一个家庭或者不再想保存某个家庭，可以将其从表视图中删除，表视图被设置成可编辑的模式。当 tableView:commitEditingStyle:forRowAtIndexPath:方法接收到一个删除编辑动作时，homeManager 使用 removeHome:completionHandler:方法删除对应参数 row 的家庭。

```
HMHome *homeToRemove = [self.homeManager.homes objectAtIndex:indexPath.row];
__weak ICFHomeTableViewController *weakSelf = self;

[self.homeManager removeHome:homeToRemove completionHandler:^(NSError *error) {

    [weakSelf.tableView deleteRowsAtIndexPaths:@[indexPath]
                            withRowAnimation:UITableViewRowAnimationAutomatic];

}];
```

删除家庭之后，completion handler 块还会从表视图中删除相应的 row。注意，删除一个家庭会删除所有与之相关的 HomeKit 对象，比如和这个家庭绑定的房间和附件。

8.3.5 房间和区域

HomeKit 中的房间(HMRoom 实例)表示一个家庭中实际的物理房间，比如厨房、主卧或

客厅。房间还用来组织各种附件，比如“前门的锁”这个附件就可能出现在休息室。要编辑一个家庭中的房间，用户可以在示例程序中点击家庭中的行条目。如果用户访问房间视图时发现家庭中没有设置任何一个房间，程序会提示用户创建一个新的房间。用户需要提供一个唯一的房间名，并使用 addRoomWithName:completionHandler:方法向家庭添加房间。

```
__weak ICFRoomTableViewController *weakSelf = self;
[self.home addRoomWithName:newRoomName completionHandler:
➥ ^(HMRoom *room, NSError *error)
{
    if(error)
    {
        NSLog(@"Error adding home: %@",error.localizedDescription);
    } else
    {
        NSInteger row = [weakSelf.home.rooms indexOfObject:room];

        NSIndexPath *addedRoomIndexPath =
        ➥ [NSIndexPath indexPathForRow:(row + 1) inSection:0];

        [weakSelf.tableView insertRowsAtIndexPaths:@[addedRoomIndexPath]
        ➥ withRowAnimation:UITableViewRowAnimationAutomatic];
    }
}];
```

房间添加完毕后，completion handler 将会在表中插入一个新的行以显示该房间。表视图将会对 self.home 实例使用 rooms 属性来确定表中一共有多少行：

```
-(NSInteger)tableView:(UITableView *)tableView
  ➥ numberOfRowsInSection:(NSInteger)section {
   return [self.home.rooms count] + 1;
}
```

rooms 属性是一个由 HMRoom 实例组成的数组，表视图会为每一行获取相应的 HMRoom 实例，并在表视图单元格中显示它的 name 属性。

```
-(UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    UITableViewCell *cell =
    ➥ [tableView dequeueReusableCellWithIdentifier:@"roomNameCell"
                                       forIndexPath:indexPath];

    if(indexPath.row == 0)
    {
        [cell.textLabel setText:@"Tap to add new room"];
    } else
    {
        NSInteger row = indexPath.row - 1;
        HMRoom *room = [self.home.rooms objectAtIndex:row];
```

```
        [cell.textLabel setText:room.name];
    }
    return cell;
}
```

要移除一个房间，用户可以点击 Edit 按钮将表视图修改为编辑模式，在任何一个现有的房间上点击删除控件。在 tableView:commitEditingStyle:forRowAtIndexPath:方法中，选定的房间将会从家庭中移除。

```
HMRoom *roomToDelete = [self.home.rooms objectAtIndex:(indexPath.row - 1)];
[self.home removeRoom:roomToDelete completionHandler:^(NSError *error) {
    [tableView deleteRowsAtIndexPaths:@[indexPath]
                     withRowAnimation:UITableViewRowAnimationAutomatic];
}];
```

区域(HMZone 实例)为一些逻辑上相关房间的组合，区域之间的房间不必存在物理上的联系。区域主要是将房间进行组合以便进行一些方便的处理，比如设置区域为“楼上”或“楼下”，或者设置为“卧室”和“卫生间”。区域内所有的房间和附件都可以与之进行交互，比如“关闭楼下的灯”。

区域在家庭中的存储方法类似于房间，可以从家庭添加和移除，HMHome 实例具有 zones 属性。而 HMZone 实例具有 rooms 属性，它是包含 HMRooms 对象的数组，HMZone 还提供了添加和移除 HMRoom 的方法。

8.3.6 附件

附件是提供服务的具体物理设备，比如灯、开关、咖啡机、门锁和诸如监视感应器和门禁等安全系统组件。附件的提供者必须通过苹果公司的 HomeKit 功能为设备提供完整的认证过程。

注意

除了独立的设备之外，HomeKit 还具有一种特殊的设备类型，称作 bridge。bridge 是一个使其他未遵循 HomeKit 的设备以编程方式遵循 HomeKit 的控制器。HomeKit 可以同 bridge 进行通信，以查找 bridge 提供的设备，并在之后使用 bridge 同这些设备进行通信，在用户看来这一过程是无缝衔接的。

向家庭添加附件需要使用 HMAccessoryBrowser 实例扫描本地环境中遵循 HomeKit 协议的附件。在示例程序中，用户通过点击附件查看当前家庭中已经关联的附件。为了搜索新的附件，用户可以点击 Edit，然后选择 Search for New Accessories，将会得到一个 ICFAccessoryBrowserTableViewController 实例。这个视图控制器会创建一个空的数组，使用附件填充这个数组，并实例化一个 HMAccessoryBrowser。

```
self.accessoriesList = [[NSMutableArray alloc] init];
[self.tableView reloadData];

self.accessoryBrowser = [[HMAccessoryBrowser alloc] init];
```

```
[self.accessoryBrowser setDelegate:self];
[self.accessoryBrowser startSearchingForNewAccessories];
```

当附件浏览器找到新的附件时，需要设置 accessoryBrowser 的委托来进行处理。

```
-(void)accessoryBrowser:(HMAccessoryBrowser *)browser
    didFindNewAccessory:(HMAccessory *)accessory {

    [self.accessoriesList addObject:accessory];
    NSInteger rowAdded = [self.accessoriesList indexOfObject:accessory];
    NSIndexPath *addedIndexPath = [NSIndexPath indexPathForRow:rowAdded inSection:0];

    [self.tableView insertRowsAtIndexPaths:@[addedIndexPath]
     ➥withRowAnimation:UITableViewRowAnimationAutomatic];
}
```

浏览器每发现一个新的附件就将它添加到 accessoriesList 数组，之后将其插入到表视图中进行显示，如图 8-6 所示。

当用户点击其中一个附件时，提醒控制器会给出一个提示框，询问用户附件的名称。比如用户希望添加 Door Lock(门锁)附件，并为它指定一个专门的名称，比如 Front Door Lock(前门锁)，如图 8-7 所示。

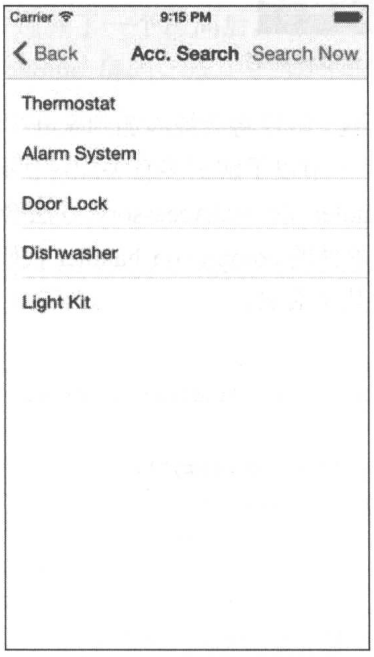


图 8-6　示例程序：附件浏览器找到的附件列表

图 8-7　示例程序：添加附件

当用户输入了名称后，提醒控制器的动作代码块会尝试向家庭添加这个附件。

```
UITextField *accNameTextField = addAccAlertController.textFields.firstObject;
NSString *newAccName = accNameTextField.text;
[self.home addAccessory:selectedAccessory completionHandler:^(NSError *error) {
    ...
}];
```

self.home 调用 addAccessory:方法时，HomeKit 会初始化一个配对过程。HomeKit 会呈现另一个提醒控制器，向用户请求配对代码，如图 8-8 所示。

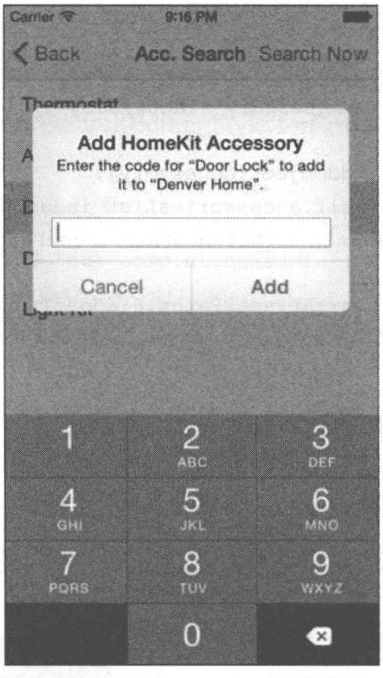


图 8-8 示例程序：HomeKit 附件配对

如果用户提供的配对代码同设备的配对代码相匹配，附件将会被添加到家庭中(查看本章后面的“使用 HomeKit Accessory Simulator 进行测试”一节以了解更多有关配对代码的内容)。如果不匹配或者用户取消配对请求，completion handler 的 addAccessory:方法会返回一个 NSError 实例。如果附件被成功添加到家庭中，添加附件的 completion handler 代码块中将会使用由用户为附件提供的名称(在图 8-7 中已设置)对其重命名。

```
if(!error) {
    [selectedAccessory updateName:newAccName completionHandler:^(NSError *error) {
        if(error) {
            NSLog(@"Error updating name for selected accessory");
        }
    }];
} else {
    NSLog(@"Error adding selected accessory");
}
```

附件被添加到家庭中后，还需要将它添加到家庭的房间内。默认情况下，会将附件添加到整个家庭的房间内。在示例程序中，用户可以在附件列表中点击刚刚添加的附件以查看详细信息，如图 8-9 所示。

之后，用户可以点击房间界面选择另一个房间，如图 8-10 所示。

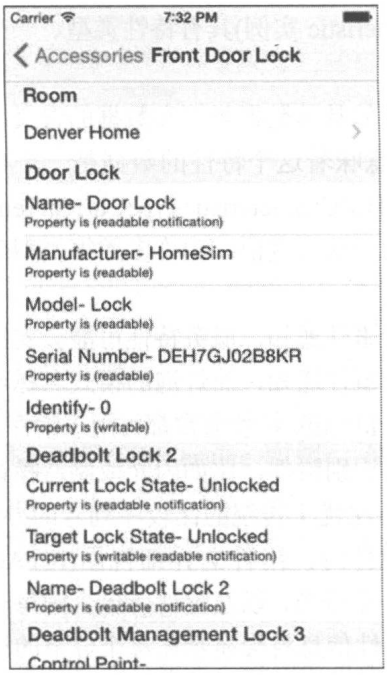


图 8-9　示例程序：附件详情

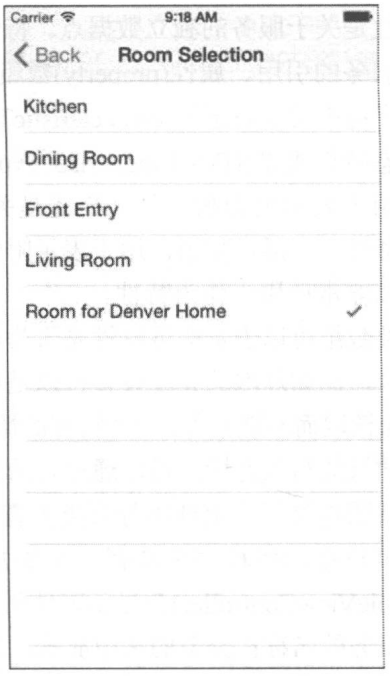


图 8-10　示例程序：为附件选择一个房间

用户选择了一个房间后，tableView:didSelectRowAtIndexPath:方法会将附件分配到选中的 self.home 房间。选中的房间可以是 self.home 的 rooms 属性中的房间，也可以是 roomForEntireHome。

```
HMRoom *selectedRoom = nil;
if(indexPath.row < [self.home.rooms count])
{
    selectedRoom = [self.home.rooms objectAtIndex:indexPath.row];
} else
{
    selectedRoom = [self.home roomForEntireHome];
}

[self.home assignAccessory:self.accessory
                    toRoom:selectedRoom
         completionHandler:^(NSError *error) {
    if(error)
    {
        NSLog(@"Error assigning accessory to room: %@", error.localizedDescription);
    }
}];
```

8.3.7　服务和服务组

在此，服务的含义是指附件所能提供的功能。服务(HMService 实例)具有名称、服务类型、其所包含附件的引用和一个服务所具有的特性(characteristic)列表。附件通常都会带有用于描述附件的信息服务，并且至少有汇报服务状态和允许交互的功能。

特性是关于服务的独立数据点。特性(HMCharacteristic 实例)具有特性类型、一个对于其所包含服务的引用、属性(property)数组和元数据。

由字符串常量表示的 characteristicType 指示特性的具体数据类型和数据的含义。比如，电力状态特性类型(HMCharacteristicTypePowerState)意味着这个特性的数据是一个布尔类型的值，用于表示电力状态是开启还是关闭。另外，HMCharacteristicTypeCurrentTemperature 特性类型是一个浮点型值，用于表示附件所测量出的当前温度值。由许多数据类型表示的特性类型都非常适用于作为特性。

属性数组可以表示服务特性是否为可读写或支持事件通知，服务特性可能会支持各种属性的组合，比如灯泡的亮度设置可以是可读写并支持事件通知，而灯泡的模式名称可以设置为只能够读取而不能写入。一种特殊的情况需要注意，Identify 特性通常都可用于附件。Identify 是典型的只能写入特性，可以通过合适的方法用它来识别附件，比如闪光或制造噪音。这样对用户试图将类似的附件进行分类大有裨益，这种方法优于传统的比较序列号的办法。

当用户在示例程序中选择一个添加到家庭的附件时，附件的详细视图(ICFAccessoryDetailTableViewController)会显示附件所对应房间的一个分节，以及附件所包含的每一个服务分节。服务的名称在分节的头中显示，服务包含的特性信息将会显示在每个分节的行内，如图 8-11 所示。

要修改特性的值，用户可以点击表视图的单元格，如图 8-12 所示，点击 Power State 单元格可关闭灯泡。

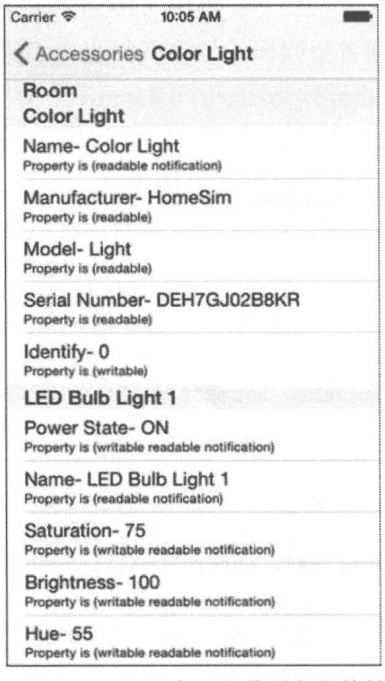


图 8-11 示例程序：附件服务和特性

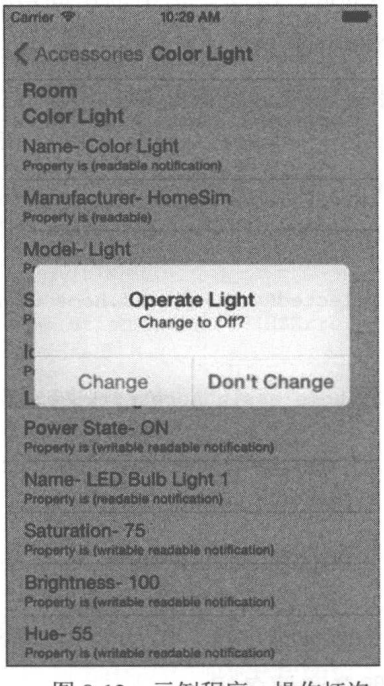


图 8-12 示例程序：操作灯泡

要打开灯泡，提醒控制器使用的动作代码块如下所示：

```
[characteristic writeValue:[NSNumber numberWithBool:targetState]
        completionHandler:^(NSError *error) {
```

```
            if(error) {
                NSLog(@"Error changing state: %@",error.localizedDescription);
            } else {
                [self.tableView reloadRowsAtIndexPaths:@[indexPath]
                ➥withRowAnimation:UITableViewRowAnimationAutomatic];
            }
    }];
```

writeValue:completionHandler:方法可以接收任何 id 值，不过只有与特性相符的传入值才会有效。示例程序会检查特性类型并使用不同的方法处理不同的特性类型，不过任何能够确保为特性提供正确值的方法都可以考虑。

服务组(HMServiceGroup 实例)提供了一种跨附件的服务组合方法，服务组包含一个名称和一个服务数组(HMService 实例)。使用 addService:completionHandler:和 removeService:completionHandler:方法分别实现向服务组中添加和移除某个服务的功能。可以将服务组添加到一个 HMHome 实例中并对它进行管理。

8.3.8　动作和动作集

动作和动作集提供了一种快速更新一组附件特性的方法，比如关闭厨房里所有的灯或者锁上家里的所有房门。

HMAction 是一个抽象类，只带有一个具体实现 HMCharacteristicWriteAction。可以使用 initWithCharacteristic:targetValue:方法创建 HMCharacteristicWriteAction 实例，之后将其添加到 HMActionSet 实例中。

一个 HMActionSet 实例包含名称、动作数组(HMAction 实例)和一个叫做 executing 的属性。使用 addAction:completionHandler 和 removeAction:completionHandler:方法可以分别实现添加动作和移除动作的功能。动作集通过将动作添加到 HMHome 来创建。

```
[self.home addActionSetWithName:@"Turn On Lights"
            completionHandler:^(HMActionSet *actionSet, NSError *error) {
    if(!error) {

        HMCharacteristicWriteAction *writeAction =
        ➥[[HMCharacteristicWriteAction alloc] initWithCharacteristic:characteristic
        ➥targetValue:[NSNumber numberWithBool:YES]];

        [actionSet addAction:writeAction completionHandler:^(NSError *error) {
            if(error) {
                NSLog(@"Error adding action to actionSet: %@",
                ➥error.localizedDescription);
            }
        }];
    }
}];
```

动作集可被添加到一个按预定计划要执行的触发器中，或者使用 HMHome 的 executeActionSet:方法立即执行该动作。

8.4 使用 HomeKit Accessory Simulator 进行测试

在编写本书时存在这样一个问题：如果在编码时没有 HomeKit 相关设备，开发者应该如何在没有设备的情况下创建和测试 HomeKit 应用呢？幸运的是，苹果公司也考虑到了这一问题并提供了解决方案：HomeKit Accessory Simulator(HomeKit 附件模拟器)。HomeKit Accessory Simulator 是一个基于 Mac OS X 的应用，支持开发者设置任何虚拟的 HomeKit 附件类型，并可以向同真实设备交互一样进行交互。模拟器通过网络将消息发送给附件，和真实场景一样，让 HomeKit 应用可以连接并同附件进行交互。

要获取 HomeKit Accessory Simulator，请访问 Xcode 中目标的 Capabilities 标签页(如本章之前的图 8-2 所示)。如果 HomeKit 功能是开启的，在 HomeKit 功能信息部分就会有一个名为 Download HomeKit Simulator 的按钮。点击该按钮将会引导开发者导航到 Apple developer Web 网站(需注册)的“Downloads for Apple Developers”部分，找到并下载“Hardware IO Toolsfor Xcode”项，其中包含 HomeKit Accessory Simulator。下载完毕后，解压模拟器并将其安装到 /Applications 目录下。

当 HomeKit Accessory Simulator 第一次启动时，里面没有任何数据。要使用模拟器，需要添加一个新的附件(或 bridge)。要想添加附件，可以点击应用左下角的加号(+)按钮，然后选择 New Accessory。HomeKit 会显示一个带有新附件信息的动作表，如图 8-13 所示。

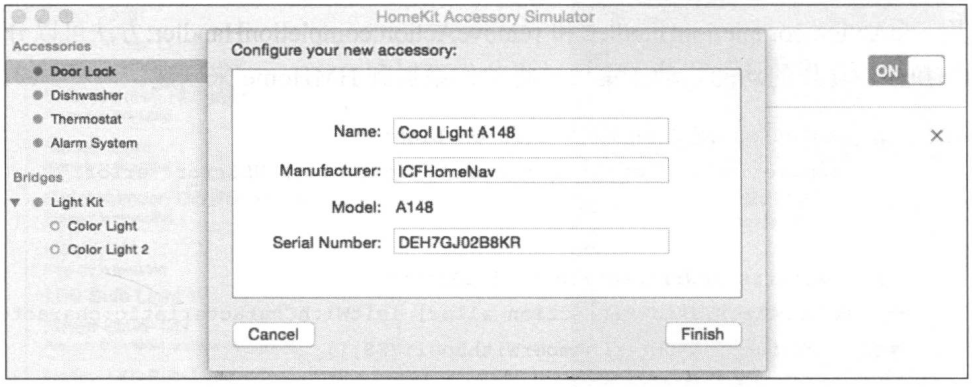


图 8-13 HomeKit Accessory Simulator：新附件

为附件提供名称和生产厂商名称，点击 Finish(注意，这时模拟器就会生成模型和序号)。模拟器会添加附件，在左边部分的列表中可以看到该附件。选择附件并注意它的服务信息，点击 Add Service 向附件添加服务，并在弹出框中选择需要添加的服务类型。模拟器会为附件添加服务和标准特性，如图 8-14 所示。

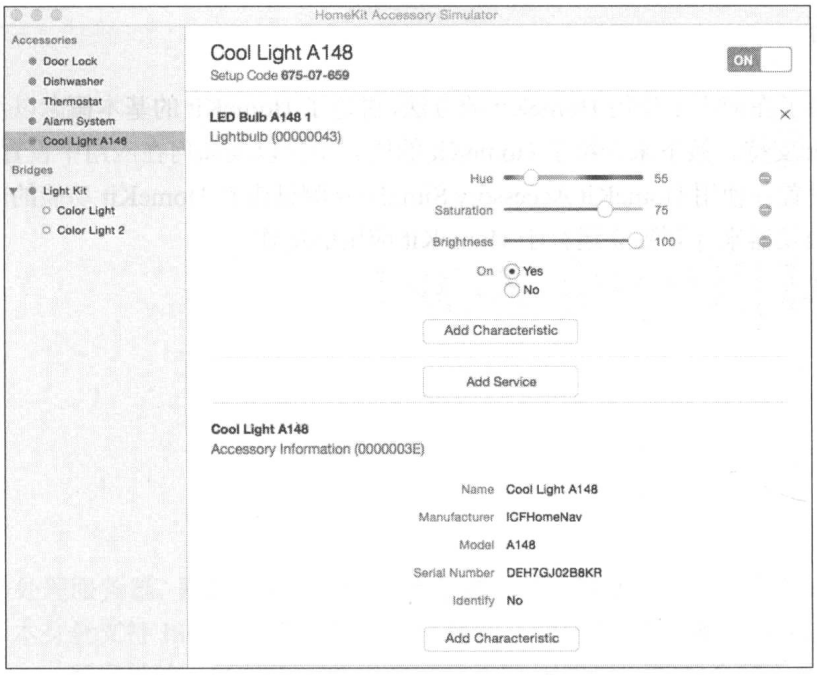


图 8-14　HomeKit Accessory Simulator：附件信息

特性信息可以直接编辑，并且在应用的 HMCharacteristic 信息中可以反映所做的编辑操作。可以通过右上角的开关来切换附件的开启和关闭状态，当附件开启时，可以在 HMAccessoryBrowser 中找到。附件名称之下的 Setup Code 选项用于为附件进行 HomeKit 配对操作(见图 8-8)。

如果 HomeKit Accessory Simulator 中一个已经配对的附件的特性值发生了变化，这些改变将会立即在模拟器中显示出来。

8.5　使用触发器计划动作

触发器提供了一种满足某一条件即开启 HomeKit 相关动作的方法。目前 HomeKit 支持计时器触发器(HMTimerTrigger)，可以根据设置的时间和日期或一定的时间间隔激活动作。使用这个类型的触发器是可以在后台运行并更新 HomeKit 对象状态的唯一方法，HomeKit 计时器触发器由 iOS 管理。

触发器基于动作集，如上节介绍的“动作和动作集”中所述。要实现一个触发器，首先需要创建触发器要执行的动作和动作集。之后，使用 initWithName:fireDate:timeZone:recurrence:recurrenceCalendar:方法初始化触发器。触发器准备好后，调用 enable:completionHandler:激活触发器。之后它就会在指定的日期触发相关的动作(或在每个循环间隔触发)。

计划好的触发器可以很好地解决按计划执行动作的问题，比如在晚上打开节日灯并在早上关闭，确保家里和车库的门在离开时关闭，甚至在出去度假时运行自动喷水系统或自动喂鱼系统。

由于触发器主要由 iOS 维护，因此应用不需要为了已触发的动作而特意运行它。

8.6 小结

本章介绍了在应用中使用 HomeKit 的方法，讲述了 HomeKit 的基本概念以及如何为项目设置 HomeKit 支持。接下来介绍了 HomeKit 的所有组件以及如何在应用中设置和维护它们，解释了如何设置并使用 HomeKit Accessory Simulator 测试带有 HomeKit 功能的应用，还介绍了如何使用触发器来计划独立运行于 HomeKit 应用的动作。

第9章

JSON的使用和解析

JSON是处理服务器、网站和iOS应用间数据通信的一种优秀技术。它比XML轻便和简单，并且iOS本身就支持JSON，很容易将其整合到iOS项目中。许多著名的网站，比如Flickr、Twitter和Google所提供的API的返回结果使用的都是JSON格式，很多语言都提供JSON支持。本章将讲解如何在应用中从一台样本消息服务器上解析和生成一条JSON语句，并在JSON中编码一条新的消息以发送给服务器。

9.1 JSON

JSON(JavaScript Object Notation)是一种轻量级数据共享技术。实际上它是JavaScript语言的一部分，是JavaScript对象序列化的一种方法，不过在实际使用中，JSON可以广泛支持多种程序开发语言，为跨平台间数据共享做出了非常大的贡献。JSON的可读性也是其优点之一。

JSON的语法很简单，作为最基础的语言，JSON文档包含的对象为核心键值字典，这一点和Objective-C程序语言类似。JSON包含对象数组和值数组，并可以构筑这些数组和对象的关系。保存在数组中或通过相应键值保存在JSON中的值可以是其他JSON对象，比如字符串、数值或数组，或是true、false和nil等对象。

9.1.1 使用JSON的好处

在iOS应用中之所以选择使用JSON，原因有很多：

- **服务器支持**：iOS应用经常会同远程服务器进行通信，由于许多服务器语言都自身支持JSON，因此很自然地选择JSON作为数据格式。
- **轻量化**：和XML相比，JSON几乎不需要在格式转换方面进行过多的开销，并且在服务器和设备之间进行数据传输时能够节省大量的带宽。
- **iOS支持**：随着NSJSONSerialization类的引入，JSON现在全面支持iOS 5及以上版本。这个类可以很方便地从JSON数据得到NSDictionary或NSArray(甚至是可能变化的其他类型)对象，也可以将NSDictionary或NSArray编码为JSON格式。

- **呈现和本地化处理**：从服务器获得数据到 iOS 设备最简单的方法是使用 UIWebView 并显示一个 Web 页面，不过这个方法在性能上并不理想。在很多情况下，直接从服务器上获取数据并使用诸如 UITableView 的本地工具将其呈现在设备上是更好的方法。这么做会有比较好的性能表现，并且呈现的方式可以根据 iOS 屏幕的尺寸进行优化，利用设备的高清显示屏进行显示。

9.1.2 JSON 资源

要想了解更多有关 JSON 的知识，可以访问 http://json.org。这个网站提供了 JSON 的官方定义以及一些有关格式和语法的详细解释。

9.2 示例程序

本章的示例程序名为 Message Board(消息板)，包括一台 Ruby on Rails 服务器和一个 iOS 应用。

Ruby on Rails 服务器只包含一个对象：消息。它被设置成使用 JSON 发送一个消息列表，并接收 JSON 格式的新消息。该服务器还支持基于 Web 的交互。

iOS 应用将从服务器获取消息并在标准的表视图中显示，它有能力将这些消息以 JSON 格式传送给服务器。

9.3 访问服务器

要查看 Message Board 的 Ruby on Rails 服务器，可以访问 http://freezing-cloud-6077.herokuapp.com/。消息的主界面如图 9-1 所示。

消息服务器已被设置为可以在 Web 上使用 JSON 创建并显示消息。

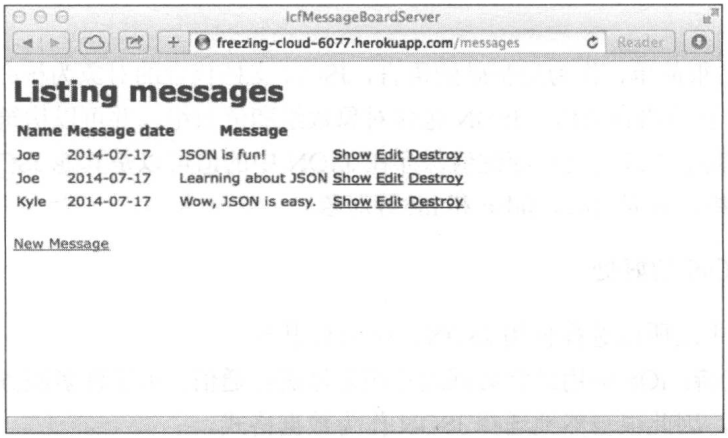


图 9-1 消息主界面

9.4 从服务器获取 JSON

要更新示例 iOS 应用来处理 JSON，首先需要从服务器获得消息列表并显示。

9.4.1　创建请求

首先设置 URL，这样应用可以向正确的位置发起调用：

```
NSString *const kMessageBoardURLString =
➥@"http://freezing-cloud-6077.herokuapp.com/messages.json";
```

在 ICFViewController.m 实现代码的 viewWillAppear:方法中，初始化一个对服务器的请求：

```
NSURL *msgURL = [NSURL URLWithString:kMessageBoardURLString];
NSURLSession *session = [NSURLSession sharedSession];

NSURLSessionTask *messageTask = [session dataTaskWithURL:msgURL
 ➥completionHandler:^(NSData *data, NSURLResponse *response, NSError *error) {
 ...
}];
[messageTask resume];
```

上面的程序根据服务器的 URL 地址创建并初始化一个到 messages.json 资源的网络请求。网络请求是异步运行的，当数据返回时将会调用 completion handler 块。需要重点注意的是，对于 JSON 不需要做任何专门的处理，它是标准的网络调用。唯一的不同是在 URL 中使用.json 扩展名会通知服务器返回结果应该是 JSON 格式。其他服务器可能使用 Content-Type 或 Accept 作为 HTTP 头，用来说明 application/json 作为文件类型时期望返回值为 JSON 格式。

注意

使用了.json 扩展名并不需要服务器返回 JSON 数据，这里只是在示例服务器上这样设置。虽然这是常见方法，但并不是一定要这样做。

9.4.2　检查反馈

当网络请求返回时，将会调用 completion handler。在示例程序中，数据被转换为 UTF-8 字符串以输出到控制台显示。在实际的应用中并不是每个请求都要进行这样的转换，在此这么做是为了当有 JSON 解析错误发生时可以更好地在调试过程中观察数据的反馈情况。

```
NSString *retString =
➥ [NSString stringWithUTF8String:[data bytes]];

NSLog(@"json returned: %@", retString);
```

数据接收到之后，日志消息将会显示在控制台上：

```
json returned: [{"message":{"created_at":"2012-04-29T21:59:28Z",
"id":3, "message":"JSON is fun!", "message_date":"2012-04-29",
"name":"Joe","updated_at":"2012-04-29T21:59:28Z"}},
{"message":{"created_at":"2012-04-29T21:58:50Z","id":2,
"message":"Learning about JSON", "message_date":"2012-04-
29","name":"Joe", "updated_at":"2012-04-29T21:59:38Z"}},
{"message":{"created_at":"2012-04-29T22:00:00Z","id":4,
"message":"Wow, JSON is easy.", "message_date":"2012-04-
```

```
29","name":"Kyle", "updated_at":"2012-04-29T22:00:00Z"}},
{"message":{"created_at":"2012-04-29T22:46:18Z","id":5,
"message":"Trying a new message.", "message_date":"2012-04-
29","name":"Joe", "updated_at":"2012-04-29T22:46:18Z"}}]
```

9.4.3 解析 JSON

现在从服务器接收到了 JSON 消息，使用简单的方法就可以对其进行解析。在示例程序中，我们希望得到一个消息数组，所以把 JSON 解析成一个数组对象 NSArray：

```
NSError *parseError = nil;
NSArray *jsonArray =
➥ [NSJSONSerialization JSONObjectWithData:data
                                   options:0
                                     error:&parseError];

if(!parseError) {
    [self setMessageArray:jsonArray];
    NSLog(@"json array is %@", jsonArray);
} else {
    NSString *err = [parseError localizedDescription];
    NSLog(@"Encountered error parsing: %@", err);
}
```

NSJSONSerialization 的方法 JSONObjectWithData:options:error:需要将序列化的数据和所需的 options 作为参数(比如返回可变数组而不是常规数组)，还需要一个对 NSError 的引用参数用于处理解析中可能出现的错误。

在示例中，将一个本地实例变量更新为刚刚解析好的数组，表视图被通知重新载入数据，现在已经有数据可以显示了，活动视图(activity view)已被隐藏。注意，completion handler 很有可能在后台队列中调用，所以如果用户界面需要更新，就需要切换到主队列。

```
dispatch_async(dispatch_get_main_queue(), ^{
    [self.messageTable reloadData];
    [self.activityView setHidden:YES];
    [self.activityIndicator stopAnimating];
});
```

9.4.4 显示数据

现在 JSON 已经被解析为一个 NSArray，可以在 UITableView 中显示。神奇之处就在于居然没有任何神奇的操作，从服务器接收的 JSON 现在是一个 NSDictionary 实例数组。每个 NSDictionary 都包含从服务器得到的消息，带有特性名和值。要在表中显示这些内容，只需直接访问数组和字典即可，就如同它们已在本地创建一样。

```
-(UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
    ➥ [tableView dequeueReusableCellWithIdentifier:@"MsgCell"];
```

```
    if(cell == nil) {
        cell = [[UITableViewCell alloc]
            ➥initWithStyle:UITableViewCellStyleSubtitle
            ➥reuseIdentifier:@"MsgCell"];

        cell.selectionStyle = UITableViewCellSelectionStyleNone;
    }
    NSDictionary *message =
    ➥ (NSDictionary *)[[self.messageArray
                ➥objectAtIndex:indexPath.row]
                ➥objectForKey:@"message"];

    NSString *byLabel =
    ➥ [NSString stringWithFormat:@"by %@ on %@",
    ➥ [message objectForKey:@"name"],
    ➥ [message objectForKey:@"message_date"]];

    cell.textLabel.text = [message objectForKey:@"message"];
    cell.detailTextLabel.text = byLabel;
    return cell;
}

-(NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return [[self messageArray] count];
}
```

已解析的 JSON 数据会在一个标准的表视图中显示，如图 9-2 所示。

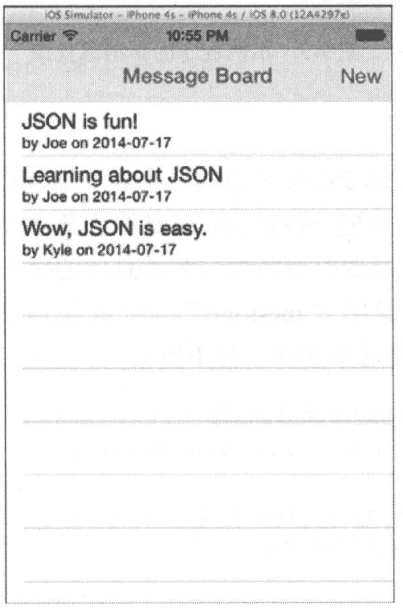


图 9-2　示例程序的消息表视图

提示

当 JSON 源数据中有 null 值的时候，将会被解析为 [NSNull null]。此时如果检查或比较语句，需要 nil 作为对象时就会出现问题，因为[NSNull null]会返回 YES 而 nil 会返回 NO。明智的做法是当转换为模型对象或显示 JSON 解析对象时对[NSNull null]进行特殊处理。

9.5 发送消息

示例程序包含一个 ICFNewMessageViewController 类，用于将新消息发送给服务器。控制器上有两个文本输入框：其中一个用来输入名称，另一个用来输入消息(如图 9-3 所示)。在用户输入信息并保存之后，将会编码为 JSON 数据并发送给服务器。

图 9-3 示例程序的新消息视图

9.5.1 JSON 数据编码

向 Ruby on Rails 服务器发送 JSON 消息的一个重要细节是对数据进行编码，这样它会镜像 Rails 服务器提供的内容。当把一条新消息发送给服务器时，其结构应该同消息列表中的一条单独的消息一样。为此，需要带有特性名和消息值的字典文件，之后字典文件被包装好并带有指向特性字典的关键字“message”。它是服务器发送消息的完全镜像，在 saveButtonTouched:方法中设置这个字典，如下所示：

```
NSMutableDictionary *messageDictionary =
➥ [NSMutableDictionary dictionaryWithCapacity:1];

[messageDictionary setObject:[nameTextField text]
                      forKey:@"name"];

[messageDictionary setObject:[messageTextView text]
                      forKey:@"message"];
```

```
NSDate *today = [NSDate date];

NSDateFormatter *dateFormatter =
➥ [[NSDateFormatter alloc] init];

NSString *dateFmt = @"yyyy'-'MM'-'dd'T'HH':'mm':'ss'Z'";
[dateFormatter setDateFormat:dateFmt];
[messageDictionary setObject:[dateFormatter stringFromDate:today]
                      forKey:@"message_date"];


NSDictionary *postDictionary = @{@"message" : messageDictionary};
```

注意，NSJSONSerialization 只接受 NSDictionary、NSArray、NSString、NSNumber 和 NSNull 5 种实例类型。对于日期和其他数据类型，NSJSONSerialization 并不直接支持，需要转换为支持的格式，比如例子中的日期就被转换成了服务器所期望的字符串格式。注意对字典对象进行 JSON 编码是很容易实现的。

```
NSError *jsonSerializationError = nil;
NSData *jsonData = [NSJSONSerialization
                ➥dataWithJSONObject:postDictionary
                ➥options:NSJSONWritingPrettyPrinted
                ➥error:&jsonSerializationError];

if(!jsonSerializationError)
{
   NSString *serJSON =
   [[NSString alloc] initWithData:jsonData
                     encoding:NSUTF8StringEncoding];

   NSLog(@"serialized json: %@", serJSON);
      ...
} else
{
   NSLog(@"JSON Encoding failed: %@",
        ➥ [jsonSerializationError localizedDescription]);
}
```

NSJSONSerialization 需要如下 3 个参数：

1) 含有要编码的数据的 NSDictionary 或 NSArray 对象。

2) 序列化选项(在此指定了 NSJSONWritingPrettyPrinted，旨在增强可读性，否则 JSON 的内容中就没有空格)。

3) 一个对 NSError 的引用。

如果 JSON 数据中没有编码错误，将会显示为下面这个样子：

```
serialized json: {
  "message" : {
```

```
    "message" : "Six Test Messages",
    "name" : "Joe",
    "message_date" : "2012-04-01T14:31:11Z"
  }
}
```

9.5.2 向服务器发送 JSON 数据

JSON 数据编码完成后，就准备将它发送给服务器。首先需要一个 NSMutableURLRequest 实例，创建带有目标服务器 URL 的请求，并自定义添加 HTTP 方法(“post”)和 HTTP 头，表示上传内容为 JSON 格式。

```
NSURL *messageBoardURL =
➥ [NSURL URLWithString:kMessageBoardURLString];

NSMutableURLRequest *request = [NSMutableURLRequest
                          requestWithURL:messageBoardURL
                      cachePolicy:NSURLRequestUseProtocolCachePolicy
                  timeoutInterval:30.0];

[request setHTTPMethod:@"POST"];

[request setValue:@"application/json"
forHTTPHeaderField:@"Accept"];

[request setValue:@"application/json"
forHTTPHeaderField:@"Content-Type"];
```

请求完成后，可以创建一个 NSURLSessionUploadTask 对象。任务需要请求、JSON 数据和一个 completion handler。completion handler 将在后台线程中调用，所以所有用户界面的更新都必须分派到主队列。

```
NSURLSession *session = [NSURLSession sharedSession];

NSURLSessionUploadTask *uploadTask =
[session uploadTaskWithRequest:uploadRequest fromData:jsonData
 ➥ completionHandler:^(NSData *data, NSURLResponse *response, NSError *error) {

    NSHTTPURLResponse *httpResponse = (NSHTTPURLResponse *)response;
    BOOL displayError = (error || httpResponse.statusCode != 200);

    dispatch_async(dispatch_get_main_queue(), ^{
        [self.activityView setHidden:YES];
        [self.activityIndicator stopAnimating];
        if(displayError) {
            NSString *errorMessage = error.localizedDescription;
            if(!errorMessage) {
                errorMessage =
                ➥ [NSString stringWithFormat:@"Error uploading - http status: %i",
                ➥ httpResponse.statusCode];
```

```
            }

            UIAlertController *postErrorAlertController =
            ➥ [UIAlertController alertControllerWithTitle:@"Post Error"
                                                   message:errorMessage
                                            preferredStyle:UIAlertControllerStyleAlert];

            [postErrorAlertController addAction:
             ➥ [UIAlertAction actionWithTitle:@"Cancel"
                                         style:UIAlertActionStyleCancel
                                       handler:nil]];

            [self presentViewController:postErrorAlertController
                               animated:YES
                             completion:nil];
        } else {
            [self.presentingViewController dismissViewControllerAnimated:YES
                                                               completion:nil];
        }
    });

}];
[uploadTask resume];
```

当 uploadTask 调用 resume 方法时，会发送一个到服务器的请求，完成之后调用 completion handler。需要对 completion handler 中返回的 error 和 response 都进行检查，如果在连接网络时遇到错误(比如设备处于飞行模式)，或者因为服务器的问题导致 HTTP 状态代码显示不同的问题(比如没有找到 URL 或者服务器无法处理传入的数据)，将会返回一个 error 对象。如果请求完成且没有错误发生，视图控制器将会被移除并且更新消息显示板上的内容。

9.6　小结

本章介绍了 JSON(JavaScript Object Notation)的有关内容，学习了如何在 iOS 应用中从服务器请求 JSON 数据，并解析这些数据和在表视图中显示它们。本章还介绍了如何将 NSDictionary 或 NSArray 对象编码成 JSON，并将其通过网络发送到服务器。

第10章 通知机制

用户在使用某个应用时，为在应用未激活状态下通知用户一些重要的 iOS 应用相关事件，苹果公司使用了一种称为通知机制的方法。由于 iOS 系统同一时间只能有一个应用处于前端激活状态，因此通知机制提供了一种方法，此方法能够让未激活的应用同样接收重要和时效敏感度高的事件。本章将介绍如何设置应用以实现接收本地和远程服务器的推送通知，以及当用户接收到带有应用 badge、声音和消息通知时应该如何进行自定义的处理。

10.1 本地通知和推送通知的区别

iOS 支持两种通知类型：一种是本地通知，另一种是远程通知，或者叫推送通知。本地通知不使用或者说不需要任何外部设施，完全在设备上发生。也就是说，显示本地通知设备不需要连接任何网络，也不需要“打开”任何设置。相反，推送通知则需要网络连接和可以通过 APNs(Apple PushNotification service)发送通知的服务器设备。需要重点注意的是推送通知的发送并不是有保障的，所以不能做出所有通知都能正确到达目的地的假设。不要让你的应用完全依赖于推送通知。

要理解为什么推送通知的发送不能得到保障，需要知道推送通知是如何到达一台设备上的。如果可以的话，APNs 首先会尝试使用蜂窝网络连接 iOS 设备，之后再尝试连接 Wi-Fi。一些设备需要处于激活状态(比如在第 4 代 iPod touch 上，屏幕必须处于点亮状态)才能接收 Wi-Fi 通信。其他设备(比如 iPad)可以在休眠状态也保持 Wi-Fi 网络正常连接。

这两类通知的处理流程也不一样，本地通知的步骤如下：

(1) 创建一个本地通知对象，并指定一些选项，比如计划时间表和日期、消息、声音和 badge 更新。

(2) 设置本地通知。

(3) iOS 显示通知、播放声音并更新 badge。

(4) 在应用委托中接收本地通知。

推送通知的步骤如下：

(1) 为应用注册推送通知和接收 token。
(2) 告知服务器(根据 token 识别)哪台设备将会接收推送通知。
(3) 在服务器上创建推送通知并与 APNs 通信。
(4) APNs 发送通知给设备。
(5) iOS 显示通知，播放声音并更新 badge。
(6) 在应用委托中接收通知。

通知机制处理流程的不同使得这两个方法分别适用于不同的场景，如果设备没有外部消息，则使用本地通知，反之则使用推送通知。

10.2 示例程序

本章的示例程序名为 ShoutOut，提供的功能是用户可以向设备推送消息并添加提醒器。该示例程序首先演示设置提醒器接收本地通知，之后再介绍如何让其能够接收推送通知，推送服务器通信的消息是什么以及如何让服务器通过 APNs 发送推送通知。

10.3 应用设置

应用要想接收远程推送通知，需要进行一些设置。首先在 iOS Provisioning Portal 中设置一个 App ID，访问 iOSDev Center (https://developer.apple.com/devcenter/ios/index.action)，登录之后在屏幕右边的 iOS Developer Program 菜单(只有登录才能看到)中选择 Certificates, Identifiers & Profiles。从屏幕左侧的菜单中选择 Identifiers，之后点击右上角的加号按钮创建一个新的 App ID，如图 10-1 所示。

指定 App ID Description，此描述信息用于通过 iOS Provisioning Portal 显示应用的详情。选择一个 App ID Prefix(之前叫作 Bundle Seed ID)，将滑动条向下拖动找到设置 App ID Suffix 的地方，如图 10-2 所示。

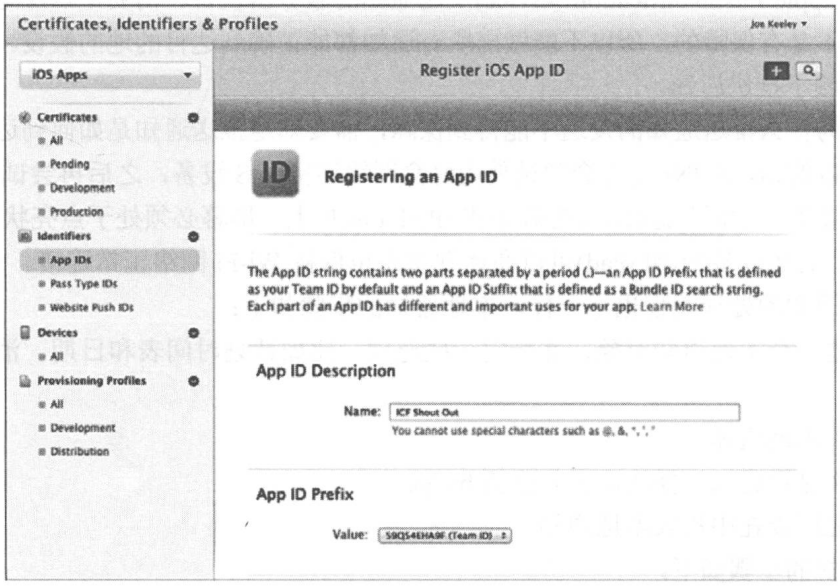


图 10-1 iOS Provisioning Portal：注册 App ID、App ID Description 和 App ID Prefix

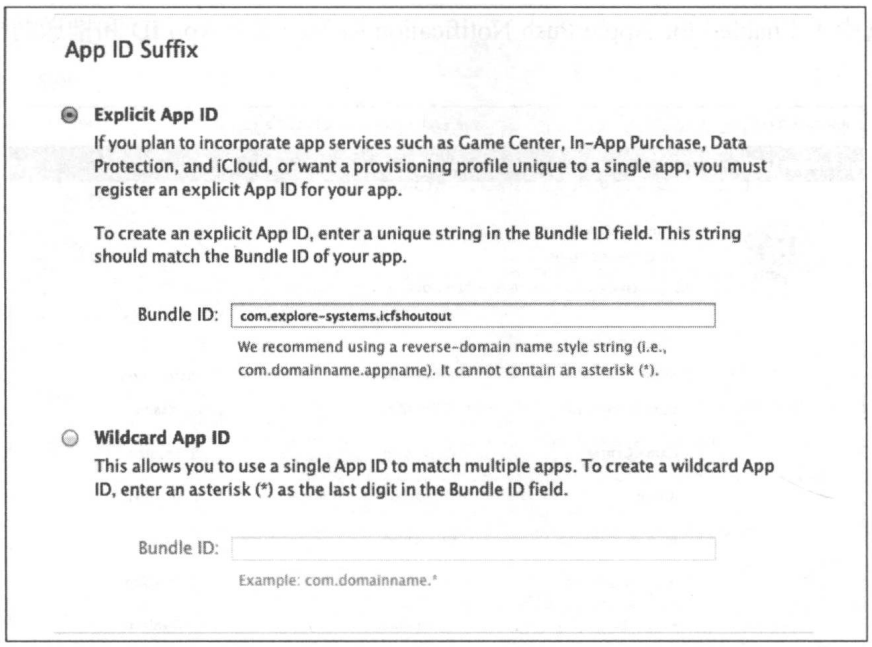


图 10-2　iOS Provisioning Portal：注册 App ID、App ID Suffix

推送通知需要一个明确的 App ID，所以为应用选择并指定同 Bundle ID 一样的字符串，向下滑动页面选择 App Services，如图 10-3 所示。

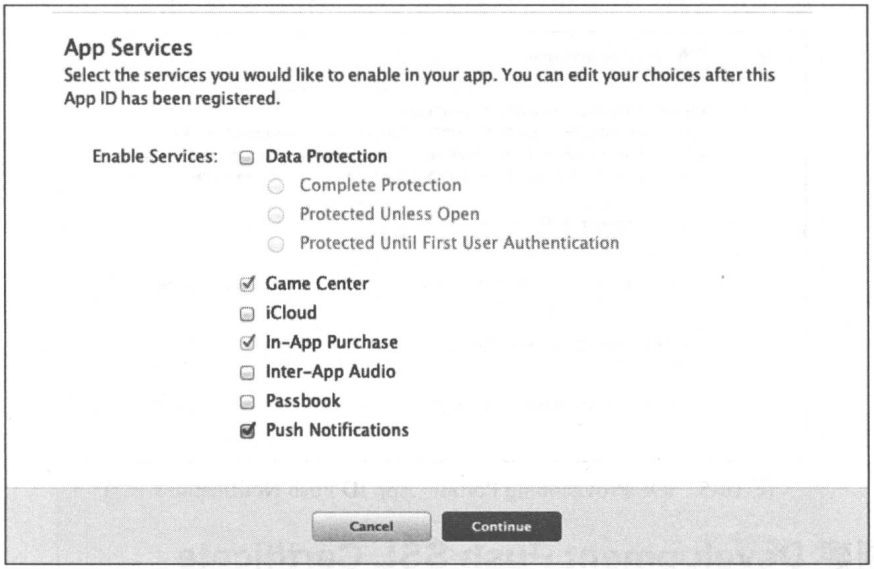


图 10-3　iOS Provisioning Portal：注册 App ID、App Services

在 App Services 列表中选中 Push Notifications，表示该 App ID 对应的程序可以使用推送通知。点击 Continue 保存新的 App ID，这样在 App ID 列表中就可以看到了。点击 App ID 展开页面并查看状态，如图 10-4 所示。

现在我们已经准备好 App ID 了，需要配置推送通知。点击 App ID 详情列表底部的 Settings 按钮，向下滑动到页面底部，查看推送通知，如图 10-5 所示。

确保选中了 Enabled for Apple Push Notification 服务，这样 App ID 和推送的授权就都创建就绪。

图 10-4 iOS Provisioning Portal：列表中的 App ID

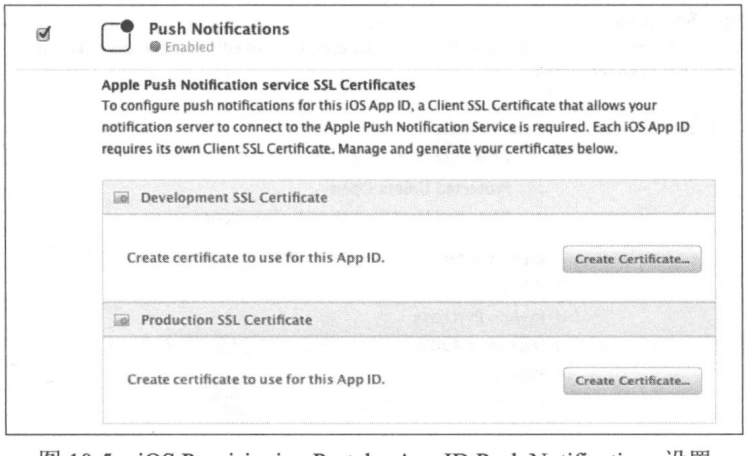


图 10-5 iOS Provisioning Portal：App ID Push Notifications 设置

10.4 创建 Development Push SSL Certificate

Development Push SSL Certificate(开发推送 SSL 证书)是当应用连接到 APNs 发送推送通知时推送服务器需要识别和认证账号的授权。要创建该授权，点击 Development 一行中的 Create Certificate 按钮(参见图 10-5)。网站会给出生成授权签名请求的介绍，如图 10-6 所示。

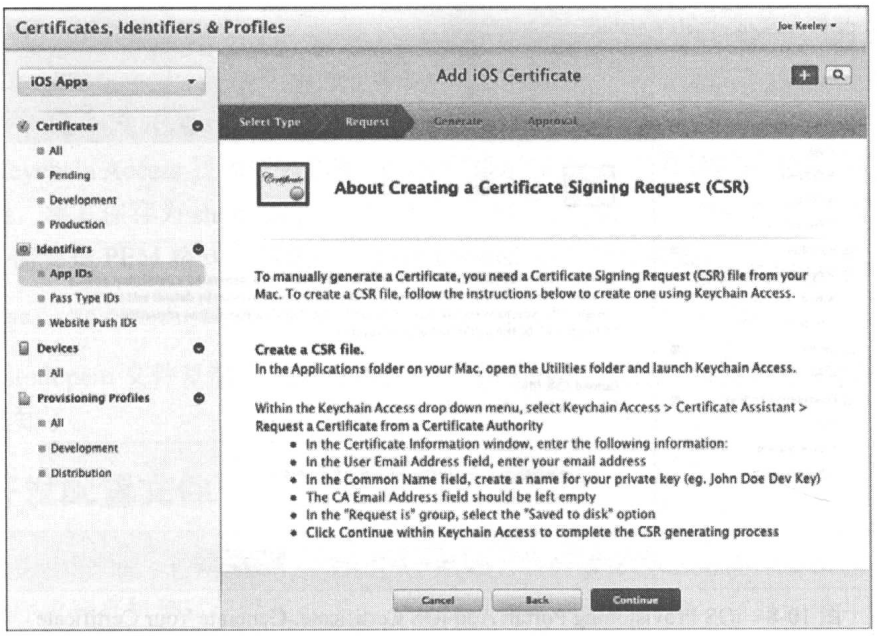


图 10-6　iOS Provisioning Portal：About Creating a Certificate Signing Request (CSR)

可以暂时离开浏览器的 Add iOS Certificate 页面，打开 Keychain Access(在 Applications，Utilities 中)。选择 Keychain Access，点击 Certificate Assistant，在应用菜单中找到 Request a Certificate from a Certificate Authority，将会呈现证书请求表单，如图 10-7 所示。

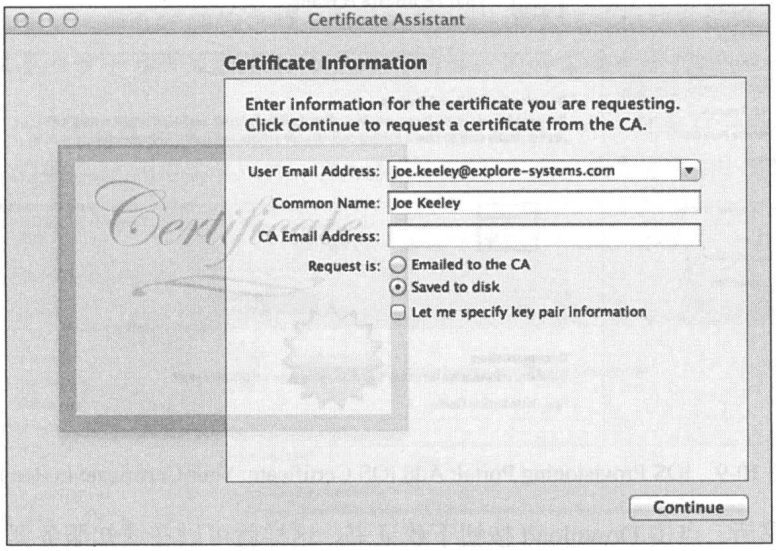


图 10-7　Keychain Access Certificate Assistant

输入电子邮箱地址和通用名(一般是公司名或个人名——使用 Apple Developer 账户的名称比较安全)，之后选择 Saved to Disk。点击 Continue 并指定保存请求的位置。这些步骤完成后，返回到 iOS Provisioning Portal 并点击 Continue，选择已保存的请求，如图 10-8 所示。

在选择已保存的请求后，点击 Generate，就会生成 Development SSL Certificate，如图 10-9 所示。

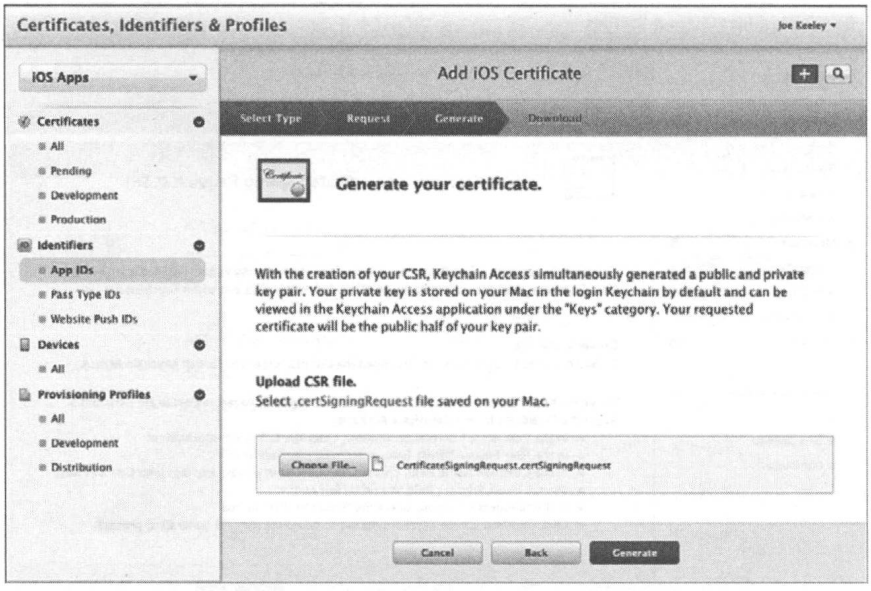


图 10-8 iOS Provisioning Portal: Add iOS Certificate: Generate Your Certificate

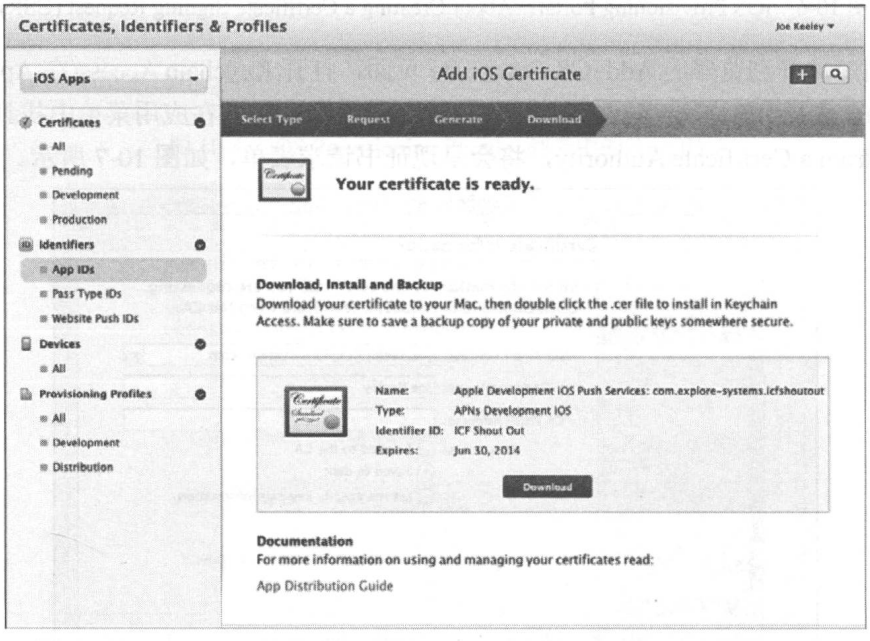


图 10-9 iOS Provisioning Portal: Add iOS Certificate: Your Certificate Is Ready

创建完证书后，点击 Download 按钮下载证书，这样就可以在通知服务器上安装证书了。

双击下载的证书文件会将其自动安装到 KeychainAccess，可以在证书列表中看到新安装的证书，如图 10-10 所示。点击前面小三角确认 private key 与证书是匹配的。

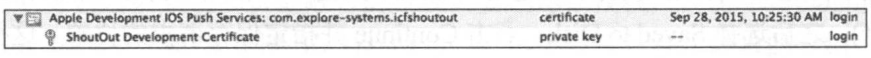


图 10-10 Keychain Access: Apple Development iOS Push Services SSL Certificate 和 private key

注意，当测试和将应用提交到 App Store 时，需要重复这个过程，以创建 Ad-Hoc 和 Production SSL Certificates。

示例程序包含一个简单的 PHP 文件，该文件可以从命令行执行同 APNs 进行通信并且可以发送测试的推送通知。为使这个文件能够执行，刚生成的证书需要转换为可以包含在 APNs 请求中的格式。同样的转换对于任何需要同 APNs 进行通信的服务器都是需要的。要转换证书，打开 Keychain Access 找到证书和图 10-10 中显示的 key。选择这两个对象，并点击 File | Export Items，将其保存为 shoutout.p12(可以使用任何带有 p12 的文件名)。为与 APNs 通信，证书和 key 需要是 PEM 格式，故采用下面的 openssl 命令进行转换：

```
$ openssl pkcs12 -in shoutout.p12 -out shoutout.pem -nodes -clcerts
```

将 shoutout.pem 文件复制到 Xcode 项目的 Push Server 组目录中，这样示例程序就准备好发送推送通知了。

10.5　开发配置文件

对于很多应用来说，使用 Xcode 自动生成的团队开发配置文件(team development provisioning profile)对程序进行真机测试已经足够了。不过要测试推送通知机制，需要创建并使用专门针对应用接收推送通知的开发配置文件。为完成该配置，点击 iOS Provisioning Portal 左侧菜单中的 Provisioning Profiles 选项。

注意

这里假设开发证书(在 Certificates,Development 页面下)已经创建好了，否则需要先创建它。网站上对于这一流程有详细的介绍，这与创建 SSL 证书的过程非常类似。这里还假设你已经在网站上设置至少一台开发设备，如果没有的话，同样需要先创建(在 Devices 页面下)。

此时会显示一个开发配置文件列表。要创建一个新的文件，点击列表右上角的加号按钮，选择创建配置文件的类型(本例中为 iOS App Development)并点击 Continue 按钮，如图 10-11 所示。

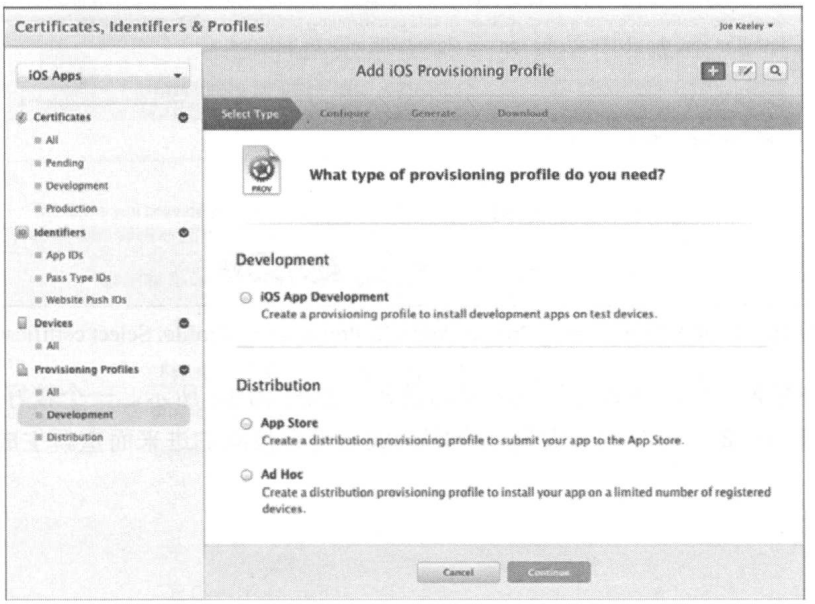


图 10-11　iOS Provisioning Profile: Add iOS Provisioning Profile

选择刚创建好的 App ID，如图 10-12 所示，点击 Continue 按钮。

接下来，为应用签署配置文件时选择需要的 Development Certificate，如图 10-13 所示，并点击 Continue 按钮。

图 10-12　iOS Provisioning Profile: Add iOS Provisioning Profile: Select App ID

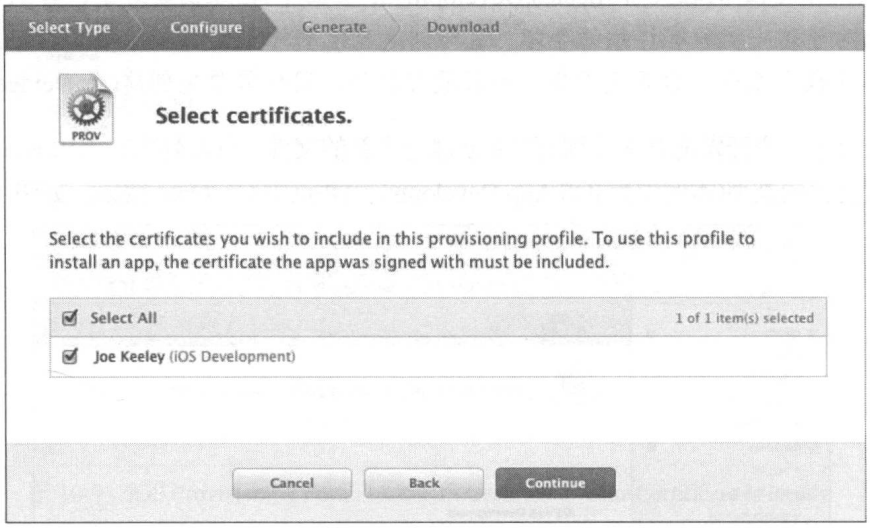


图 10-13　iOS Provisioning Profile: Add iOS Provisioning Profile: Select certificates

使用这个配置文件选择要运行该应用的设备，如图 10-14 所示。一个较好的建议是选择所有可以使用的设备，以免因为发现哪个团队成员没有被添加进来而重新生成配置文件。

图 10-14　iOS Provisioning Profile: Add iOS Provisioning Profile: Select devices

最后，为配置文件提供一个名称，并重新审查配置文件显示的概要信息，如图 10-15 所示。如果文件确认无误，点击 Generate 按钮创建即可。

注意

配置文件的名称应该具有一定的描述性，因为当配置文件越来越多时，在 Xcode 中就会很难识别。一个好的办法是使用应用名称和环境名称，比如“ICF Shout Out Development”。

图 10-15　iOS Provisioning Profile: Add iOS Provisioning Profile: Name this profile and generate

创建好配置文件后，会显示一个下载页面，如图 10-16 所示。

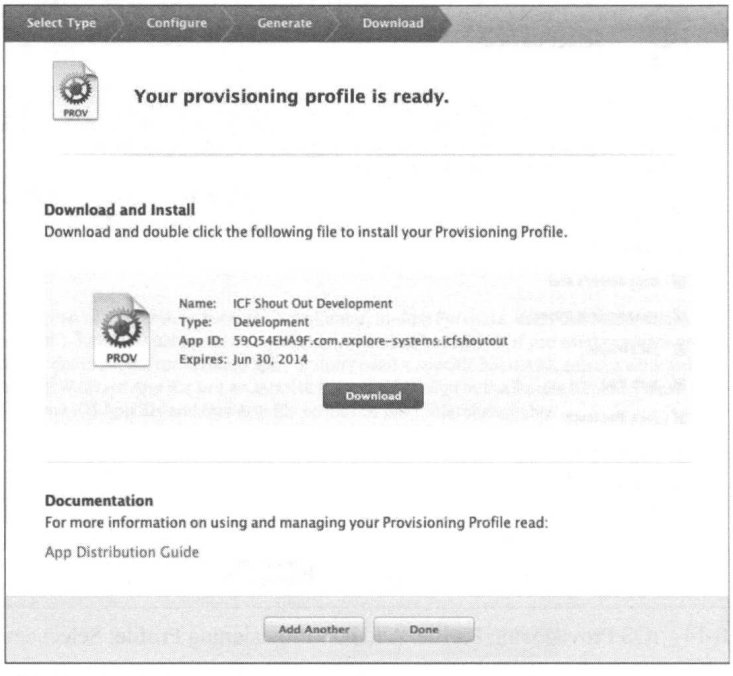


图 10-16 iOS Provisioning Profile: Your provisioning profile is ready

点击 Download 得到配置文件的一个副本，下载好之后双击文件会自动安装并在 Xcode 中生效。最后一步是确保应用使用的是最新的配置文件。在 Xcode 中，为项目(而非 target)编辑 Build Settings。找到名为 Code Signing 的选项，设置 Code Signing Identity。为了进行调试，选择 Automatic 选项下方的 iOS Developer，如图 10-17 所示。要想确认配置文件是否在系统中正确安装，检查 Code Signing Identity 项目下的 Provisioning Profile 选项，看列表中是否有该配置文件。还可以从 Xcode 菜单选择 Preferences，再找到 Accounts。如果账户信息还没有设置，需要对其进行设置。选择账户名并通过 View Details 查看配置文件是否已安装好。可以直接在此更新配置文件而不需要从网站重新下载，不过独立应用的配置文件必须从网站创建。

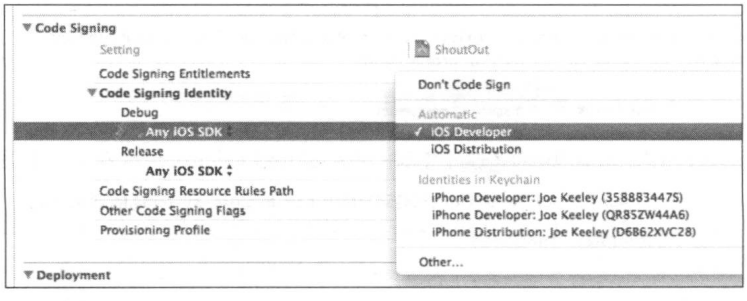


图 10-17 Xcode Project Build Settings: Code Signing Identity

这一步完成后，接收推送通知的所有配置过程就已全部完成，现在准备编写代码。

10.6　准备自定义声音

一个可以真正区分接收推送通知的细节就是自定义声音。如果应用 bundle 允许的话，iOS 可以播放任何短于 30 秒的声音。可以在 GarageBand(或者其他可以创建声音的应用)中创建一个自定义声音并导出。一个较好的做法是导出每一种压缩格式的声音，这样就可以适配各种需求，如图 10-18 所示。

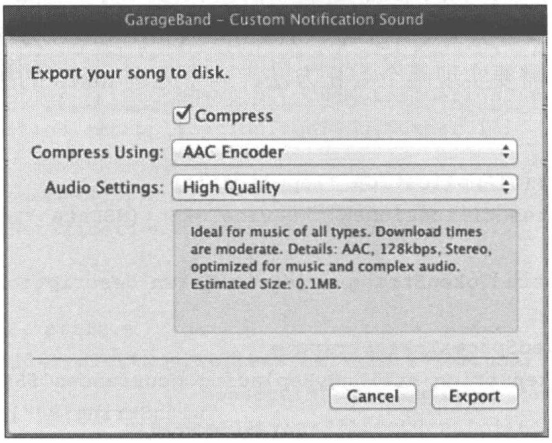


图 10-18　GarageBand：导出声音设置

现在我们有了声音文件，需要将它转换为 Core Audio 格式后应用才能使用。苹果公司提供了一个名为 afconvert 的命令行工具来实现这一转换。打开 Terminal 会话，找到声音文件目录，执行如下命令将声音文件转换为 Core Audio 格式：

```
$ afconvert -f -caff -d ima4 shout_out.m4a shout_out.caf
```

这个命令将 shout_out.m4a 文件转换为 ima4 格式(设备支持的压缩格式)并打包在一个 Core Audio 格式的声音文件中。这一过程完成后，将新的 Core Audio 格式的声音文件复制到 Xcode 项目，为一个通知指定好该文件后就可以播放了。

10.7　注册通知

为了让 ShoutOut 能够接收远程通知，应用需要在 APNs 上注册接收推送通知。此外，应用需要为用户通知进行设置以更新 badge，显示 banner 或提醒框，或者为本地和远程通知播放声音。应用可以选择任何合适的时间点自定义注册推送通知，比如示例程序在应用委托(即 application:didFinishLaunchingWithOptions:方法)中就会立即向 APNs 发起注册。

```
-(BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    [[UIApplication sharedApplication] registerForRemoteNotifications];

    UIUserNotificationSettings *notifSettings =
    ➥[UIUserNotificationSettings settingsForTypes:UIUserNotificationTypeAlert |
    ➥ UIUserNotificationTypeBadge | UIUserNotificationTypeSound categories:nil];
```

```
    [[UIApplication sharedApplication]
    ➥ registerUserNotificationSettings:notifSettings];

    return YES;
}
```

UIUserNotificationSettings 用于表示当通知被接收时向用户发送何种提醒，包括更新应用 badge、显示提醒框和播放声音等。registerForRemoteNotifications:方法将会调用 APNs 并得到一个表示设备的 token。需要实现两个委托方法，一个处理 token 的接收，另一个处理有关注册 APNs 的错误：

```
-(void)application:(UIApplication *)application
didRegisterForRemoteNotificationsWithDeviceToken:(NSData *)deviceToken {

    NSString *formattedTokenString = [deviceToken description];

    NSString *removedSpacesTokenString =
    ➥ [formattedTokenString stringByReplacingOccurrencesOfString:@" "
                                                    withString:@""];

    NSString *trimmedTokenString =
    ➥ [removedSpacesTokenString stringByTrimmingCharactersInSet:
    ➥ [NSCharacterSet characterSetWithCharactersInString:@"<>"]];

    [self setPushTokenString:trimmedTokenString];
}

-(void)application:(UIApplication *)application
➥didFailToRegisterForRemoteNotificationsWithError:(NSError *)error {
    NSLog(@"Error in push registration: %@", error.localizedDescription);
}
```

如果注册成功，token 将会以 NSData 格式返回到应用。对于 ShoutOut，该方法将会通过从 NSData 表示的字符串中移除空格和额外字符来对字符串进行格式化，并保存字符串 token，用于在发送测试推送时创建该命令。对于典型的应用，设备 token 可能需要被发送到推送服务器，以便用户账户发送通知。苹果公司建议开发者在应用启动时总是执行该注册，因为用户可能切换设备或更新 iOS 版本，这就需要一个新的 token 了。如果一些特定的错误动作需要处理，可以通过 didFailToRegiterForRemoteNotificationsWithError:方法来完成。示例程序中我们仅将错误信息进行输出。

现在 ShoutOut 已经准备接收远程推送通知并显示本地通知了。

10.8 设置本地通知

对于本地通知，不需要为应用进行额外的设置。在 ShoutOut 中，将会使用本地通知设计一个提醒器。在 ICFMainViewController 中，当用户点击 Set Reminder 按钮时会调用一个方法，

如下所示：

```
-(IBAction)setReminder:(id)sender
{
    NSDate *now = [NSDate date];
    UILocalNotification *reminderNotification = [[UILocalNotification alloc] init];
    [reminderNotification setFireDate:[now dateByAddingTimeInterval:15]];
    [reminderNotification setTimeZone:[NSTimeZone defaultTimeZone]];
    [reminderNotification setAlertBody:@"Don't forget to Shout Out!"];
    [reminderNotification setAlertAction:@"Shout Now"];
    [reminderNotification setSoundName:UILocalNotificationDefaultSoundName];
    [reminderNotification setApplicationIconBadgeNumber:1];

    [[UIApplication sharedApplication]
    ➥ scheduleLocalNotification:reminderNotification];


    UIAlertController *alert =
    ➥ [UIAlertController alertControllerWithTitle:@"Reminder"
                                    message:@"Your Reminder has been Scheduled"
                            preferredStyle:UIAlertControllerStyleAlert];

    UIAlertAction *dismissAction =
    ➥ [UIAlertAction actionWithTitle:@"OK Thanks!"
                            style:UIAlertActionStyleCancel
                          handler:^(UIAlertAction *action){
                              [self dismissViewControllerAnimated:YES
                                                       completion:nil];
                          }];

    [alert addAction:dismissAction];

    [self presentViewController:alert animated:YES completion:nil];
}
```

要创建本地通知，需要创建一个 UILocalNotification 实例，为通知指定激活日期。如果用户在旅行中，那么为其指定一个时区通常也是一个好办法，这样可以保证能在正确的时间接收到提醒。为了简单地看一下效果，设置激活日期为从现在开始后 15 秒。之后设置用户如何接收通知，包括为其指定一个提醒文本框、设置一个声音(或特定音效)并更新应用 badge，最后设置本地通知。运行该应用查看实现效果，点击 Set Reminder 按钮，之后关闭应用。15 秒之后将会出现一个自定义的文本提醒框并带有声音和提醒 badge。

除了通过日期和时间设置来设置本地通知外，本地通知还可以通过区域进行设置。当设备进入某一区域时激活相应的通知。请参考第 2 章的“地理围墙”一节来了解如何在本地通知中为 region 属性设置 CLRegion 值的相关信息。

注意

本地通知可以在模拟器中测试，远程推送通知则不可以。

10.9 接收通知

当设备接收通知时，无论是本地通知还是远程通知，设备都会检查应用的通知功能当前是否可用以及应用是否在前台运行。如果没有，则通过各种参数让设备播放声音、显示提醒框或更新应用图标的 badge。如果显示了提醒框，用户可以选择取消该提醒框或者通过这个提醒框进入到相应的应用。

如果用户选择进入应用，当应用处于终止状态和启动中状态时，将会调用应用委托函数的 appDidFinishLaunchingWithOptions:方法，或者当应用回到前台运行时将会调用一个委托方法。如果接收通知时应用在前台运行，则会调用相同的委托方法。

如果应用可以通过点击屏幕上的通知消息进行启动的话，通知负载将会出现在启动选项中并传递给 appDidFinishLaunchingWithOptions:方法，用来驱动任何需要的自定义功能，比如直接导航到通知特定的视图或显示一条消息。

```
NSDictionary *notif = [launchOptions
objectForKey:UIApplicationLaunchOptionsRemoteNotificationKey];

if(notif) {
    //custom logic here using notification info dictionary
}
```

如果应用是在处于激活状态或在后台运行时接收到通知，需要为接收该通知实现两个委托方法，一个用于本地通知，另一个用于远程通知。

```
-(void)application:(UIApplication *)application
didReceiveRemoteNotification:(NSDictionary *)userInfo {
    NSString *message =
    [[[userInfo objectForKey:@"aps"] objectForKey:@"alert"] objectForKey:@"body"];

    NSString *appState = ([application applicationState] ==
    ➥UIApplicationStateActive) ? @"app Active" : @"app in Background";

    [self presentAlertWithMessage:
     [NSString stringWithFormat:@"Received remote push for app state %@: %@",
      ➥ appState, message]];
}

-(void)application:(UIApplication *)application
➥ didReceiveLocalNotification:(UILocalNotification *)notification {
    NSString *message = [notification alertBody];

    NSString *appState = ([application applicationState] ==
    ➥ UIApplicationStateActive) ? @"app Active" : @"app in Background";

    [self presentAlertWithMessage:
     ➥ [NSString stringWithFormat:@"Received local notification for app state %@: %@",
     ➥ appState, message]];
}
```

本地通知委托方法接收本地通知，远程通知委托方法接收带有通知信息的字典对象。这个信息可以被审查和采纳，在示例程序中会向用户显示一个提醒框，其他应用可以使用这个信息在程序内直接导航到相关的视图页面。两个方法中，都可以通过查看 application 参数来确定通知接收时应用的状态。之后可以根据应用当前激活或从后台唤醒等不同情况，对通知的响应进行自定义处理。

10.10　推送通知服务器

应用准备接收通知后，需要对服务器进行设置来发送推送通知。推送通知可以通过 APNs 从任何支持安全 TCP 套接字连接(需要有 SSL 堆栈)类型的服务器发送。苹果公司要求当发送推送通知请求到 APNs 时，服务器应该持续保持连接以避免频繁建立连接所带来的系统开销。可以使用开源库让 APNs 支持各种不同的平台，以及通过第三方提供的 API 访问推送通知。对于示例程序，我们在沙盒 APNs 中使用一个简单的 PHP 命令行来发送测试推送消息。

服务器至少需要知道设备的 token 才能知道将推送发送给哪台设备。对于“data available”类型的通知，所需要做的就是通知应用下载新的数据。为了满足一些其他需求，服务器可能需要根据需要指定一条消息、播放的声音、设置应用的 badge 数值，以及能够将应用导航到相关信息页面而用到的自定义哈希数据。

推送服务器将会创建一个包含发送给设备的消息负载，这个负载是 JSON 格式的且必须包含 aps 键对应的哈希值。

```
{"aps":{"alert":"Hello Joe","sound":"shout_out.caf"}}
```

如果目标应用已经被本地化，服务器可以包含一个针对提醒控件的哈希值，而不只是提供一个提醒字符串：

```
{"aps":{"alert": {"loc-key": "push-msg-key",
➥"action-loc-key": "see-push-key"}, "sound":"shout_out.caf"}}
```

相反，如果目标应用没有进行本地化，当注册设备 token 时设备可以拒绝向服务器发送本地信息，服务器可以在 APNs 发送前对信息进行本地化。可以通过指定一个 action-loc-key 元素来自定义通知按钮的标题。

注意

欲了解更多有关 APNs 通信的内容，可以查看苹果公司的文档，网址为 https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Chapters/ApplePushService.html#//apple_ref/doc/uid/TP40008194-CH100-SW9。

10.11　发送推送通知

要使用示例程序发送一条测试推送，需要在设备上运行该应用。在呈现的文本框中输入文本并点击 Shout 按钮。示例程序将会使用该内容创建一个命令，可以在 PHP 终端会话中拷贝和执行这个命令。功能的实现还需要消息参数和设备参数。

```
php shout.php "testing1..2..3"
➥"f1313af4d5af93d53ba595fdd9a9dc8799bcf10c3e7b3e2cb53662816d5bcc89"
```

要执行该命令，在 Xcode 中，在示例程序的 Push Server 组中右击 shout.php 文件，选择 Show in Finder。打开 Terminal 会话，找到 shout.php 文件所在的目录。确保证书已经创建(如图 10-10 所示)，已导出为 pem 格式并且在 shout.php 相同目录中显示。代码现在假定证书没有设置任何密码，不过如果要设置密码，取消 shout.php 中有关密码设置的注释并添加密码即可。从 Xcode 控制台复制和粘贴该命令到 Terminal 窗口并执行。脚本将会使用提供的消息和设备 ID 创建 JSON 格式的推送信息，并将其发送到 APNs。注意，这个版本对于每一个连接只发送一个推送，并且在沙盒中仅适用于非常简单的测试，任何负载测试都应该使用健壮性更强的版本，才能保持与 APNs 的持久连接。

脚本执行后，通知将会立即出现在设备上(如图 10-19 所示)。访问 Settings.app(在 Notifications, ShoutOut 页面下)，修改通知的显示样式是显示为提醒框样式还是 banner 样式，并在设备上查看每一种样式的效果。

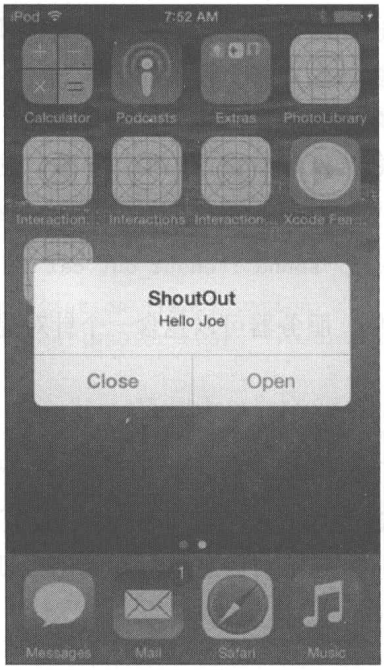


图 10-19 ShoutOut 接收到的通知并在主屏幕上显示

10.12 处理 APNs 反馈

APNs 能够对每个连接到它的服务器提供反馈并发送通知。如果在消息中指定的任何设备 token 出现错误(比如用户从设备上删除该应用)，服务器应该不再向该设备继续发送通知。有很多 APNs 库有助于实现同反馈端点进行通信，它们可以被设计成周期性地运行以实现相关功能。获得一个反馈后，应该更新或移除服务器上保存的设备 token 以防止向它们发送额外的通知。

10.13　小结

本章介绍了如何同非激活状态和在前端运行的应用进行通信，即苹果产品的通知机制。分别讲解了本地通知和远程推送通知的实现方法，学习了如何设置一个应用使其能够接收远程推送通知，以及如何对本地通知进行设置。本章还演示了如何使用推送脚本通过 Apple Push Notification 服务(APNs)将测试推送通知发送给应用。

第11章

基于 CloudKit 的云存储

iCloud 是由苹果公司提供的基于云端各种服务的集合，它在 iOS 5 中作为 MobileMe 的替代者被引入，提供云存储以及 iOS 设备、OS X 设备和 Web 间的自动同步等功能。iCloud 包括电子邮件、通讯录、备忘录、提醒事项和日历等应用的同步，自动完成 iOS 设备的备份和保存，Find My iPhone 功能还可以定位或锁死一台遗失的设备，Find My Friends 功能可以同家人和朋友共享位置，Photo Stream 功能可以在不同设备间自动同步照片，Back to My Mac 功能可以为用户配置基于网络访问 Mac 主机，iTunesMatch 可以不通过下载和同步就可以访问用户的音乐库，iCloud Keychain 用于同步密码以及使用 iCloud Drive 将文件和文档保存到云端。此外，iCloud 还可以为应用提供在云端保存应用专用数据并在设备间同步的功能。在撰写本书的时候，iCloud 已经可以提供 5GB 的免费存储空间，更多空间则要付费。

对于应用专用数据的存储和同步，iCloud 支持 4 种方案，分别是基于文档的存储和同步(基于 NSDocument 或 UIDocument)、基于键值的存储和同步(类似于 NSUserDefaults)、基于 CloudKit 的远程结构化数据存储以及 Core Data 同步。本章主要介绍如何使用 CloudKit 来设置应用，使它使用 CloudKit 设置远程结构化数据的存储。

11.1 CloudKit 基础

CloudKit 在 iOS 8 版本中引入，主要功能是提供远程数据存储、本地设备和远程存储之间的数据同步、基于推送通知的 iCloud 账户数据变更订阅。对于 CloudKit，苹果公司针对特定的存储和数据传输的限制是免费的，主要是基于应用的用户数。如果超过限制，苹果公司会收取费用，不过价格在本书撰写时还没有公布。

CloudKit 包含一个 Web 界面，用于设置和管理数据库架构和公共数据，CloudKit 还包含一组用于实现从 iOS(或 OS X)客户端访问、获取及保存数据的 API。注意，CloudKit 对于本地数据的保存没有给出任何机制，所以应用的开发者需要选择并实现本地数据存储方案，并将其整合到 CloudKit 中。如何选择本地存储机制，会在第 15 章“开始学习 Core Data”中进行详细讨论。此外，CloudKit 也不会在服务器端实现任何逻辑，所以任何同数据相关的应用

逻辑都要在客户端执行。

11.2　示例程序

本章的示例程序名为 CloudTracker，它是一个基础的数据追踪应用，使用 CloudKit 记录比赛和练习的成绩。CloudTracker 利用公共数据库保存比赛成绩，利用私有数据库保存练习的成绩。CloudTracker 中的 iCloud 用户可以获取并显示当前用户的信息。此外，CloudTracker 使用 CloudKit 订阅，当有新的比赛数据可用时会通知用户。

11.3　设置 CloudKit 项目

要使一个应用具有 iCloud 功能，需要分步骤对其进行设置。首先需要设置应用授权，应用需要在 iOS Provisioning Portal 的 iCloud 中进行配置。一些 iCloud 功能只能在真机上进行测试，所以为了能够让程序实际运行，需要完整的配置文件。从 Xcode 5 开始，这一过程都已经精简并且可以在 Xcode 中完成。

11.3.1　账户设置

Xcode 需要 iOS 开发者账户信息用于连接 Member Center 并根据用户信息为 iCloud 执行所有相关的设置。选择 Xcode，在 Xcode 菜单中找到 Preferences，选择 Accounts 页面，如图 11-1 所示。

要添加一个新的账户，点击 Accounts 页面左下角的加号图标，选择 Apple ID。输入账户信息并点击 Add 按钮。Xcode 会对账户进行验证，如果有效，则获取账户信息。点击 View Details 按钮查看当前账户信息和配置文件，如图 11-2 所示。

图 11-1　Xcode Accounts 页面

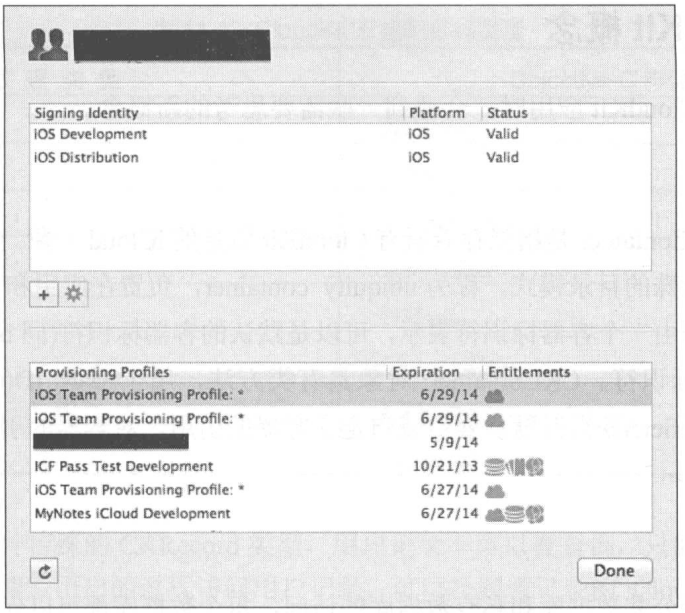


图 11-2　Xcode Accounts 详情页面

11.3.2　启用 iCloud 功能

在 Xcode 验证用户账户成功之后，就可以为该账户的应用配置功能了。可以根据需要设置 App ID、授权和配置文件。要设置 iCloud 功能，在 Xcode 中找到 CloudTracker Target，点击 Capabilities 页面并找到 iCloud 部分。将 iCloud 选项的开关改为 On，Xcode 会自动为项目创建一个授权文件。选中 CloudKit 复选框，确保为应用启用 CloudKit 功能。Xcode 将会在 Ubiquity Containers 表中自动生成一个带有项目 bundle ID 的条目。对于示例程序，上述准备已经足够了，不过对于更加复杂的应用，比如可能会同 Mac OS X 共享这类应用，还需要提供多个 ubiquity container(无处不在的容器)名，额外的 ubiquity container 名也是在这里添加的。Xcode 将会检查开发者门户来查看某个 App ID 是否正确配置了 iCloud 功能。如果没有，Xcode 将会显示错误信息，如图 11-3 所示。点击 Fix Issues 按钮，Xcode 将会向开发者门户发起通信并解决所有应用设置的问题。

图 11-3　Xcode Target Capabilities—iCloud

在配置好 iCloud 功能后，会有 CloudKit 标记和所有列出的步骤，现在应用已经准备好使用 CloudKit 了。

11.4 CloudKit 概念

当开发一款 CloudKit 应用时，会遇到一些需要思考的新概念。

11.4.1 容器

容器对象 CKContainer 是指保存着所有 CloudKit 信息的 iCloud 对象。在 iOS 文件系统中，iCloud 使用一种特殊的目录模式，称为 ubiquity container，负责在应用和 iCloud 间同步和管理数据。一个容器由一个容器标识符表示，可以是默认的容器标识符(同 bundle ID 一样)，也可以自定义容器标识符。CKContainer 对象具有类方法，用于根据 ID(defaultContainer 和 containerWithIdentifier:)获得对默认容器或自定义容器的引用。对容器的引用是为了能够访问 CloudKit 数据库。

11.4.2 数据库

CloudKit 支持公共数据库和私有数据库的访问，每个数据库都可以保存由记录类型定义好的数据。公共数据库可以被所有用户读取(甚至是没有 iCloud 账户的用户)，不过只有 iCloud 用户可以向其中写入内容。公共数据库虽然从理论上讲是所有用户都可以访问的，不过在实际中应用会对这种访问进行限制。保存在公共数据库中的数据占用了应用的 CloudKit 存储空间。要访问公共数据库，对容器对象使用 publicCloudDatabase 方法：

```
CKDatabase *publicDatabase = [[ CKContainerdefaultContainer] publicCloudDatabase];
```

私有数据库只针对应用中指定的 iCloud 用户开放。私有数据库中的数据只能被指定的用户读取和更新，甚至应用管理者都不能看到和更新私有数据库中的数据。保存在私有数据库中的数据会占用 iCloud 用户的存储空间。要访问私有数据库，对容器对象使用 privateCloudDatabase 方法：

```
CKDatabase *privateDatabase = [[ CKContainerdefaultContainer] privateCloudDatabase];
```

11.4.3 记录

CloudKit 以记录或 CKRecord 实例的方式保存结构化数据。记录由记录类型定义(同关系数据库术语中的“实体”和“表”类似)，可以在 CloudKitdashboard 中设置，也可以在开发环境中编写代码时保存一条新的记录。在生产环境下，当尝试保存未知记录类型或对记录类型使用未知属性时，服务器会返回错误信息。

CKRecord 实例具有的功能同 NSMutableDictionary 实例类似，特性和对象值可以设置为 key 值或特性名。记录支持很多特性类型，如表 11-1 所示。CloudKit 对于 CLLocation 查询提供特别支持，并且使用 CKReference 实现同其他记录的关联。此外，记录还可以使用 CKAsset 来保存数据(查看“Assets”一节)。

表 11-1　CloudKit 支持的数据类型

数 据 类 型	Objective-C 存储类型
String	NSString
Date/Time	NSDate
Int(64)	NSNumber
Double	NSNumber
Bytes	NSData
Location	CLLocation
对其他 CKRecord 的引用	CKReference
Asset	CKAsset

“用户”是一种特殊的 CKRecord 类型，用户记录不可以被查询，只能使用 record ID 直接获取，或者通过搜索用户的过程访问用户记录，可以找到通讯录的 iCloud 用户列表中存在的用户(通过电子邮件地址)。查看“搜索和管理用户”一节以了解更多内容。

11.4.4　记录区域

记录区域(CKRecordZone)是一种将相近记录类型归类的方法。记录区域的功能相当于一个将相近记录类型归类的容器。在应用的公共数据库中，只有一个记录区域，称为默认区域。对于私有数据库可以创建自定义记录区域。注意，在同一记录区域内对多条记录的修改可以一起执行或返回。记录之间的引用只存在于同一记录区域内的记录。

11.4.5　记录标识符

每一个 CKRecord 实例在被初始化时，都会由 CloudKit 默认分配一个唯一的标识符，即 CKRecordID 实例。这个记录标识符将会基于 UUID 并保证唯一，这样就可以安全地保存在本地并在需要时通过它获取 CKRecord 实例。

CloudKit 还支持自定义记录标识符，可以在记录创建时为其提供一个记录标识符，不过需要注意的是，该标识符必须在数据库中是唯一的。自定义的记录标识符必须用在当数据库在默认区域之外保存记录的情况下。

11.4.6　asset 对象

CKAsset 用于在文件中保存特性数据，而并非直接保存在记录的特性字段里。这个方法适用于图片、声音或任何其他相对大一点的数据。注意 CKRecord 对象的最大限制为 1MB，而 asset 对象可以保存 250MB 的数据。

当获取相关 CKRecord 对象时，也会下载 CKAsset。CKAsset 的数据可以通过 CKAsset 的 fileURL 方法载入。

如果需要在离线时使用 asset 的数据，应该从 CKAsset 中将数据复制到本地可以访问的文件 URL。

11.5 CloudKit 基础操作

CloudKit 给出两种处理数据的方法，完整的基于 NSOperation 的工具以及从 CKDatabase 对象而来的基于数据块的便捷方法。本章我们主要学习基于数据块的方法，以实现 CloudKit 基本操作，因为这种方法更加容易理解。不过要说明的是，基于 NSOperation 的方法会提供一些便捷方法所无法实现的额外功能，比如同时保存多条记录或在更复杂的操作上获取更新进度信息。

11.5.1 获取记录

要从 CloudKit 数据库中获取记录，需要进行查询操作。在示例程序中，比赛和练习视图都使用了查询功能来从 CloudKit 获得记录值，并将这些记录显示在表视图中。由于比赛视图中的信息数据来自于公共数据库，因此需要一个对公共数据库的引用：

```
CKDatabase *publicDatabase = [[ CKContainer defaultContainer] publicCloudDatabase];
```

CloudKit 中的查询操作会利用 NSPredicate 的功能。要从 CloudKit 中查询记录，需要创建一个 NSPredicate，用于表示目标状态，在选择标准处使用记录的特性名(可以查看第 15 章“开始学习 Core Data”的“使用谓词”一节以了解更详细的内容)。即使所有记录都是所期望的，查询也仍然需要谓词。本例中使用了一个特殊方法声明了一个谓词，用于表示全部选项：

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"TRUEPREDICATE"];
```

创建好谓词后，就可以创建 CKQuery 实例了，为其指定记录类型和谓词。

```
CKQuery *query = [[CKQueryalloc] initWithRecordType:@"Race" predicate:predicate];
```

之后，数据库上的便捷方法就可以使用它执行查询操作了。便捷方法会返回一组匹配记录的 NSArray 实例对象，或者返回 NSError 实例组成的错误集合。注意，如果区域 ID 的值为 nil，则意味着在默认区域进行查询，如下所示：

```
__weak ICFRaceListTableViewController *weakSelf = self;

[publicDatabase performQuery:query
             inZoneWithID:nil
        completionHandler:^(NSArray *results, NSError *error) {
            dispatch_async(dispatch_get_main_queue(), ^{
                weakSelf.raceList =
                ➥ [[NSMutableArray alloc] initWithArray:results];
                [weakSelf.tableView reloadData];
            });
        }];
```

这段代码中的 completion handler 可以在任意队列中执行。这样就很有可能不在主队列中执行，所以必须要注意，只要有任何用户界面更新，对其结果的处理就一定要分派到主队列来进行。异步分派可以降低死锁发生的概率。

在本例中，视图控制器在属性中保存结果数组，并通知表视图进行重载来显示这些保存的结果数据，所以需要将其分派到主队列。在公共数据库中的比赛成绩将会显示，如图 11-4 所示。

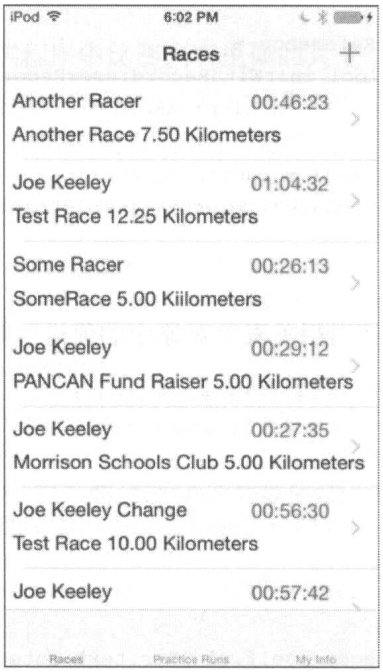


图 11-4　CloudTracker 示例程序：比赛成绩列表

11.5.2　创建并保存记录

要在示例程序中保存新的记录，点击比赛视图右上角的加号按钮(如图 11-4 所示)。应用将会显示一个表，用于填写新的比赛成绩信息。用户填完表并点击 Save 按钮后，视图控制器会显示一个活动指示器(activity indicator)，因为保存操作通常会花费一些时间。然后程序会创建一个到公共数据库的引用，之后初始化一个 CKRecord 实例，其中记录类型参数为“Race”。

```
if(!self.raceData)
{
    self.raceData = [[CKRecord alloc] initWithRecordType:@"Race"];
}
```

比赛记录的数据可以在用户界面进行更新。首先运动员或输入记录的用户会用到应用委托，并使用 CKReference 作为用户记录的引用。参考本章后面的“用户发现和管理”一节可以了解更多有关用户记录生成方面的内容。运动员的名字将会从用户记录中获取并为了方便而保存在比赛记录中。

```
if(![self.raceData objectForKey:@"racer"])
{

    AppDelegate *appDelegate =
    ➥(AppDelegate *)[[UIApplication sharedApplication] delegate];
```

```
    CKRecord *userRecord = appDelegate.myUserRecord;

    if(userRecord)
    {
        CKReference *racerReference =
        ➥[[CKReference alloc] initWithRecord:userRecord
                                    action:CKReferenceActionNone];

        self.raceData[@"racer"] = racerReference;
        self.raceData[@"racerName"] = userRecord[@"name"];
    }
}
```

接下来，每个特性都需要使用相关表单对象中的值进行赋值。

```
self.raceData[@"raceName"] = self.raceName.text;
self.raceData[@"location"] = self.location.text;
self.raceData[@"distance"] =
➥[NSNumber numberWithFloat:[self.distance.text floatValue]];
self.raceData[@"distanceUnits"] = [self selectedDistanceUnits];
self.raceData[@"hours"] =
➥[NSNumber numberWithInteger:[self.hours.text integerValue]];
self.raceData[@"minutes"] =
➥[NSNumber numberWithInteger:[self.minutes.text integerValue]];
self.raceData[@"seconds"] =
➥[NSNumber numberWithInteger:[self.seconds.text integerValue]];
self.raceData[@"raceDate"] = [self.datePicker date];
```

最后，在数据库上使用便捷方法保存比赛记录。保存方法需要一个 CKRecord 参数和 completion handler。

```
__weak ICFRaceDetailViewController *weakSelf = self;
[publicDatabase saveRecord:self.raceData
       completionHandler:^(CKRecord *record, NSError *error) {
           dispatch_async(dispatch_get_main_queue(), ^{
               [weakSelf.saveRaceActivityIndicator setHidden:YES];
               if(weakSelf.indexPathForRace) {
                   [weakSelf.raceDataDelegate raceUpdated:record
                   ➥forIndexPath:weakSelf.indexPathForRace];
               } else {
                   [weakSelf.raceDataDelegate raceAdded:record];
               }

               [weakSelf.navigationController popViewControllerAnimated:YES];
           });
       }];
```

再次注意，completion handler 可以在任意队列中调用，所以如果用户界面要更新，需要将其分派到主队列。completion handler 将会隐藏活动指示器，告知比赛列表视图控制器或者

更新保存的记录，或者处理添加新比赛记录的请求，之后离开从视图控制器(detail view controller)界面。

11.5.3　更新和保存记录

要更新比赛记录，点击示例程序中比赛列表里面的其中一行。在 prepareForSegue:sender: 方法中，和该视图相关的比赛对象 CKRecord 将会传递给从视图控制器。其对应的索引值也会传递到从视图控制器，这样当更新完成时只有对应的行会刷新。

```
if([segue.identifier isEqualToString:@"updateRaceDetail"])
{
    NSIndexPath *tappedRowIndexPath = [self.tableView indexPathForSelectedRow];
    CKRecord *raceData = [self.raceList objectAtIndex:tappedRowIndexPath.row];
    [detail setIndexPathForRace:tappedRowIndexPath];
    [detail setRaceData:raceData];
    [detail setConnectedToiCloud:self.connectedToiCloud];
}
```

从视图的用户界面将会根据传入的比赛记录对象生成。

```
if(!race)
{
    return;
}
[self.raceName setText:race[@"raceName"]];
[self.location setText:race[@"location"]];
[self.distance setText:[race[@"distance"] stringValue]];

[self.distanceUnit setSelectedSegmentIndex:
 [self segmentForDistanceUnitString:race[@"distanceUnits"]]];

[self.hours setText:[race[@"hours"] stringValue]];
[self.minutes setText:[race[@"minutes"] stringValue]];
[self.seconds setText:[race[@"seconds"] stringValue]];
[self.datePicker setDate:race[@"raceDate"]];
```

信息更新以及用户点击 Save 按钮之后，比赛记录将会以同样的逻辑保存为一条新的记录(同前面一节中演示的一样)。

11.6　订阅和推送

CloudKit 提供针对数据变化的订阅、监听等功能，当相关数据发生变化时可以接收到通知提醒。这样可以使应用在开销不大的情况下实现对数据变化的及时响应，甚至可以让一个不经常使用的应用保持数据更新。CloudKit 通过使用推送通知机制和谓词来实现这一强大的功能(请参考第 10 章“通知机制”以了解更多内容，并回想推送通知的发送是非保障的)。

11.6.1 推送设置

如果想让应用接收有关订阅的推送通知，需要进行一些设置。首先，应用需要注册远程通知并接收设备 token，这样 CloudKit 才知道将推送发送到哪台设备上。注意，不需要为 CloudKit 通知机制设置推送认证，CloudKit 直接在内部对其进行管理。

示例程序在应用委托的 application:didFinishLaunchingWithOptions:中注册推送通知。最好将这个经常使用的通知注册逻辑的代码保存在应用的另一个语境中，这样用户就很清楚地知道推送通知能够提供的功能。首先，示例程序注册远程通知：

```
[application registerForRemoteNotifications];
```

之后示例程序注册所需的用户通知设置，可以对这一步进行自定义。

```
UIUserNotificationSettings *notifSettings =
[UIUserNotificationSettings settingsForTypes:UIUserNotificationTypeAlert |
 UIUserNotificationTypeBadge | UIUserNotificationTypeSound categories:nil];

[application registerUserNotificationSettings:notifSettings];
```

两个注册完成后，应用就可以准备从 CloudKit 接收推送通知了。

11.6.2 数据变更的订阅

在注册好远程通知后(在应用委托的 application:didRegisterForRemoteNotificationsWithDeviceToken:方法中)，示例程序需要对比赛数据设置订阅。在实际的应用中，订阅通常在程序的另一个语境中进行设置。示例程序对于所有新增或更新的比赛记录都会接收通知。订阅(CKSubscription 实例)需要一个记录类型、一个谓词、一个唯一标识符和一些选项。其中对于唯一标识符的设置，一个好的建议是使用商家标识符(vendor identifier)，因为这样对于设备和应用的组合是唯一的。还可以通过其他的设置为记录类型或谓词进行更多的设置。唯一标识符也是移除订阅功能必需的参数，所以当创建时应该保存它，当使用一些其他方法复制时也应该保存它。订阅选项用于表示通知触发(创建、更新、删除)时的当前环境，可以指定为任何需要的组合。可以注册多个订阅用于处理不同的状况。

```
CKDatabase *publicDatabase = [[CKContainer defaultContainer] publicCloudDatabase];
NSPredicate *allRacesPredicate = [NSPredicate predicateWithFormat:@"TRUEPREDICATE"];
NSString *subscriptionIdentifier =
➥[[[UIDevice currentDevice] identifierForVendor] UUIDString];

CKSubscription *raceSubscription =
[[CKSubscription alloc] initWithRecordType:@"Race"
➥predicate:allRacesPredicate
➥subscriptionID:subscriptionIdentifier
➥options:CKSubscriptionOptionsFiresOnRecordCreation |
➥CKSubscriptionOptionsFiresOnRecordUpdate];
```

订阅实例化之后，需要为其设置通知样式。示例程序将使用一个提醒框，不过还可以添加 badge 和声音。注意，提醒消息可以通过替换变量进行自定义设置。

```
CKNotificationInfo *notificationInfo = [[CKNotificationInfo alloc] init];
[notificationInfo setAlertBody:@"New race info available!"];
[raceSubscription setNotificationInfo:notificationInfo];
```

之后，还需要将订阅保存到数据库以进行激活。

```
[publicDatabase saveSubscription:raceSubscription
            completionHandler:^(CKSubscription *subscription, NSError *error) {
    if(error)
    {
        NSLog(@"Could not subscribe for notifications: %@",
        ➥error.localizedDescription);
    } else
    {
        NSLog(@"Subscribed for notifications");
    }
}];
```

completion handler 可以在任何队列中被调用。如果保存成功，则会包含订阅对象；如果没有保存成功，则会返回一个错误。订阅保存好之后，应用就可以在任何约定好的时间接收到通知了，如图 11-5 所示。

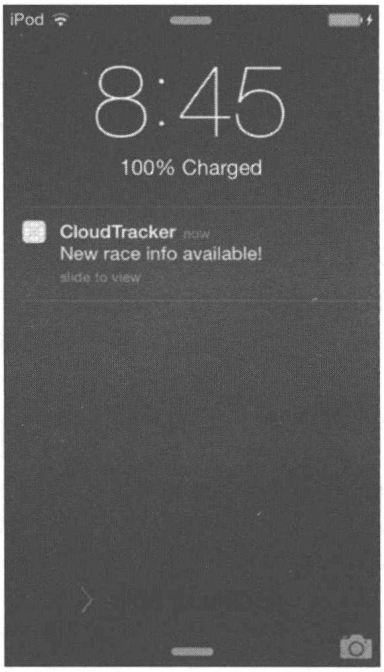


图 11-5　CloudTracker 示例程序——新数据通知

11.7　用户发现和管理

要在应用中使用 CloudKit 的所有功能，用户必须拥有一个 iCloud 账户。对于非 iCloud 用户来说，CloudKit 允许用户以只读的方式访问公共数据库；如果要向公共数据库中写入数据或访问私有数据库，就一定需要 iCloud 账户。同时，苹果公司也说明了使用 CloudKit 不需

要 iCloud 账户，所以 CloudKit 应用必须知道用户是否有 iCloud 账户，以调整用户界面的显示。

提示

虽然可以在模拟器中使用 iCloud 账户测试(同在真机上登录 iCloud 一样)，不过如果要在模拟器上测试未登录 iCloud 账户的情况，有些用例则无法实现。对于非 iCloud 账户的测试，最好使用真机进行以避免测试不完整。

在示例程序中，未登录 iCloud 账户情形下唯一能正常运行的界面就是比赛列表。未登录 iCloud 的用户可以查看比赛列表中的数据，但是不能添加或更新数据。比赛列表视图控制器将会在 CloudKit 容器上使用 accountStatusWithCompletionHandler:方法来确定当前设备的 iCloud 账户状态：

```
[[CKContainer defaultContainer] accountStatusWithCompletionHandler:
➥^(CKAccountStatus accountStatus, NSError *error) {
   dispatch_async(dispatch_get_main_queue(), ^{
      self.connectedToiCloud = (accountStatus == CKAccountStatusAvailable);
      [self.addRaceButton setEnabled:self.connectedToiCloud];
   });
}];
```

completion handler 将会保存诸如当前是否有 iCloud 账户这样的信息，这样当某一行被点击时，从界面就会根据此信息选择是否进行相应的更新，iCloud 账户验证可用之后，Add Race 按钮则会更新为可用状态。

在 CloudKit 中，用户记录是一个特殊的 CKRecord。用户记录总是存在于公共数据库中并不带有自定义特性，不过可以向其添加自定义特性。关于当前用户的信息不可以直接查询，应用只能通过用户发现过程查看 iCloud 账户可见的用户是哪些，这一过程使用用户通讯录中的电子邮件地址。该过程完成后，当前用户信息就可以访问了。此用户信息可以用来更新用户记录。

My Info 视图将会执行用户发现功能，首先需要进行访问授权检查。

```
[[CKContainer defaultContainer] requestApplicationPermission:
➥CKApplicationPermissionUserDiscoverability
➥completionHandler:^(CKApplicationPermissionStatus applicationPermissionStatus,
➥NSError *error) {
   if(error)
   {
      NSLog(@"Uh oh - error requesting discoverability permission:
      ➥%@",error.localizedDescription);
   } else
   {
      if(applicationPermissionStatus == CKApplicationPermissionStatusGranted)
      {
         [self lookUpUserInfo];
      }
```

```
    }
}];
```

应用将会显示一个提醒框，询问用户是否有权执行发现过程，该对话框只会出现一次。假设用户被授权，下一步就是获取当前用户的 recordID，它用来确定用户信息属于哪个当前用户：

```
[[CKContainer defaultContainer] fetchUserRecordIDWithCompletionHandler:
➥^(CKRecordID *recordID, NSError *error) {
    if(error)
    {
        NSLog(@"Error fetching user record ID: %@",error.localizedDescription);
    } else
    {
        [self discoverUserInfoForRecordID:recordID];
    }
}];
```

recordID 验证有效后，会在容器上调用 discoverAllContactUserInfosWithCompletionHandler: 方法以执行用户发现过程：

```
[[CKContainer defaultContainer] discoverAllContactUserInfosWithCompletionHandler:
➥^(NSArray *userInfos, NSError *error) {
    if(error)
    {
        NSLog(@"Error discovering contacts: %@",error.localizedDescription);
    } else
    {
        NSLog(@"Got info: %@", userInfos);

        for(CKDiscoveredUserInfo *info in userInfos) {
            if([info.userRecordID.recordName
            ➥isEqualToString:recordID.recordName]) {
                //this is the current user's record
                dispatch_async(dispatch_get_main_queue(), ^{
                    NSString *myName = [NSString stringWithFormat:
                                        @"%@ %@",info.firstName, info.lastName];

                    [self.name setText:myName];
                    self.myUserRecordName = info.userRecordID.recordName;

                    self.currentUserInfo = @{@"name":myName,
                                             @"location":self.location.text,
                                             @"recordName":
                                             ➥info.userRecordID.recordName};

                    NSUserDefaults *defaults =
                    ➥[NSUserDefaults standardUserDefaults];
```

```
                [defaults setObject:self.currentUserInfo
                            forKey:@"currentUserInfo"];

                [defaults synchronize];

                [self fetchMyUserRecord];
            });
        }
    }
}
}];
```

如果用户发现过程成功，将会返回一组 CKDiscoveredUserInfo 实例的数组。代码将会遍历整个数组，比较 CKDiscoveredUserInfo 的 userRecordID 和当前用户的 recordID。如果相同，从 CKDiscoveredUserInfo 中提取姓和名字段并将其写入 name 文本框。关于当前用户的信息字典，包括名字、位置和记录名称，都会创建并保存在 NSUserDefaults 中，这样在启动过程中就不必重复获取了。

当前用户的记录名称保存好之后，就可以获取当前用户记录了。

```
CKRecordID *myUserRecordID =
➥[[CKRecordID alloc] initWithRecordName:self.myUserRecordName];

CKDatabase *publicDatabase = [[CKContainer defaultContainer] publicCloudDatabase];
➥[publicDatabase fetchRecordWithID:myUserRecordID
            completionHandler:^(CKRecord *record, NSError *error) {
    if(error)
    {
        NSLog(@"Error fetching user record: %@", error.localizedDescription);
        [self setMyUserRecord:nil];
    } else
    {
        AppDelegate *appDelegate =
        ➥(AppDelegate *)[[UIApplication sharedApplication] delegate];

        [appDelegate setMyUserRecord:record];
        [self setMyUserRecord:record];
    }
}];
```

如果获取过程成功，表示当前用户的 CKRecord 对象将会保存在应用委托中，这样当保存新的比赛数据时，从视图控制器就可以很容易地访问它。如果用户更新名称和位置并点击 Save 按钮，变更将会保存到用户记录中。

```
if(self.myUserRecord) {
    self.myUserRecord[@"location"] = self.location.text;
    self.myUserRecord[@"name"] = self.name.text;
```

```
    CKDatabase *publicDatabase =
    ➥[[CKContainer defaultContainer] publicCloudDatabase];
    [publicDatabase saveRecord:self.myUserRecord
           completionHandler:^(CKRecord *record, NSError *error) {
       if(error) {
          NSLog(@"Error saving my user record: %@", error.localizedDescription);
       }
    }];
}
```

注意

对于所有指定了 iCloud 账户并在本地保存数据的 CloudKit 应用来说，都需要监听 NSUbiquityIdentityDidChangeNotification 通知，它用于识别 iCloud 账户状态的变化，并在用户退出时进行相应的处理，或者当有不同的 iCloud 账户登录时进行相应的处理。

11.8　在 dashboard 中管理数据

除了 iOS SDK 之外，CloudKit 还提供了一种基于 Web 的用 dashboard 管理 CloudKit 服务器的方法。要访问 dashboard，点击 iCloud Capabilities 页面的 CloudKit Dashboard 按钮，如图 11-6 所示。

图 11-6　Xcode Target Capabilities—iCloud

CloudKit dashboard 将会打开一个默认的 Web 浏览器，并请求认证。提供开发者账户证书来访问 dashboard。dashboard 将会在左侧面板中显示一些关于服务器的种类信息。选择其中一个类别会进一步展示更加详细的信息并在中间面板显示一些额外的选项。在中间面板中选择一个选项就会打开详情信息，可以在后边的面板中对信息进行编辑。

记录类型可以通过选择 Schema 分类下面的 Record Types 条目进行编辑，如图 11-7 所示。通过右上方面板的图标可以添加或删除记录类型，在右边面板的下方还可以对选中的记录类型设置特性和索引。

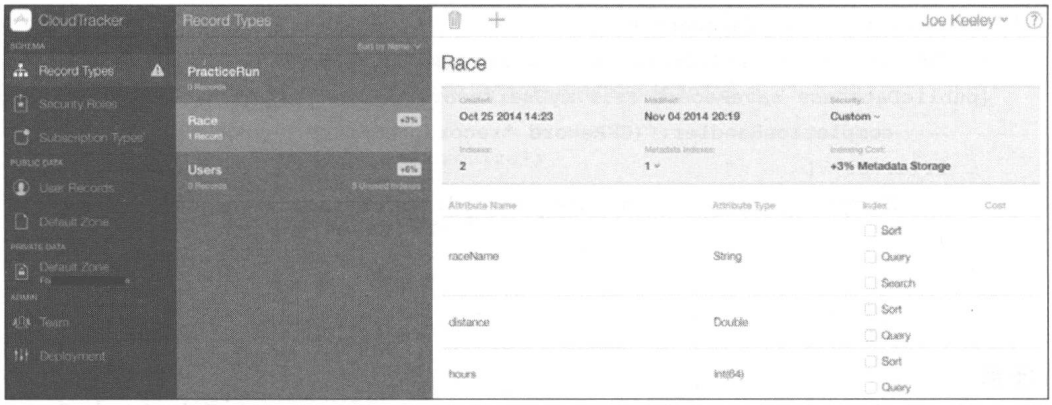


图 11-7 CloudKit dashboard—Record Types

公共数据库中的数据也可以通过 dashboard 查看和管理。用户信息可以通过选择 Public Data 分类下面的 User Records 条目进行查看。数据信息可以通过选择 Public Data 分类下面的 Default Zone 分类进行查看。在选择了 Default Zone 之后，在中间面板的上方选择一个记录类型，这样就可以查看和编辑它的数据了，如图 11-8 所示。也可以对排序进行调整，搜索记录。每一条记录都可以被查看、添加、编辑或删除。

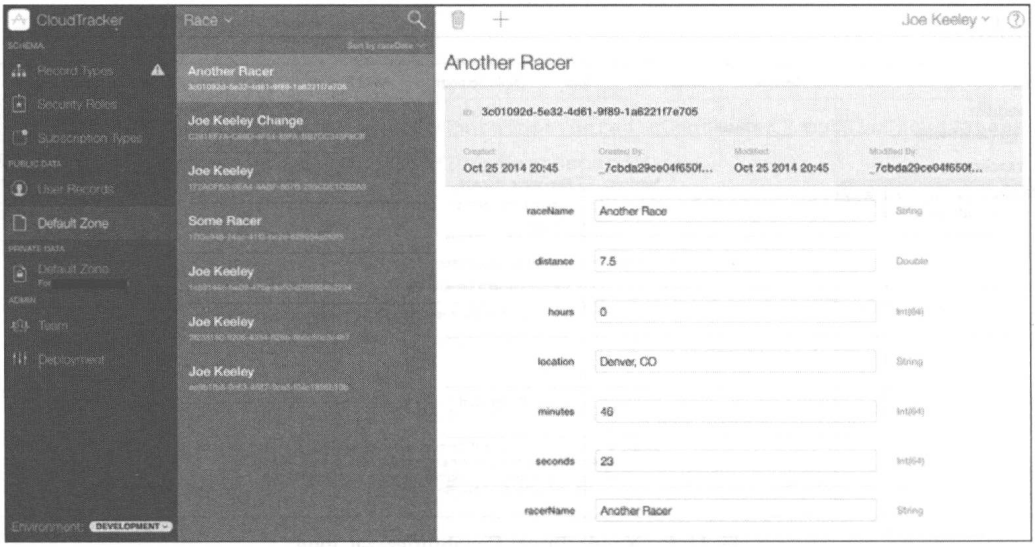


图 11-8 CloudKit dashboard—Public Data, Default Zone

私有数据只对单个 iCloud 用户可见。选择 Private Data 分类下的 Default Zone 将会在 dashboard 中为当前登录 iCloud 的用户显示数据。

在 Admin 分类下面的 Deployment 部分(如图 11-9 所示)，有一个重置开发环境或部署生产环境的选项。重置开发环境是一个“极端选项”——它会将所有记录类型都按照生产环境进行变化，同时删除开发环境中的所有数据。

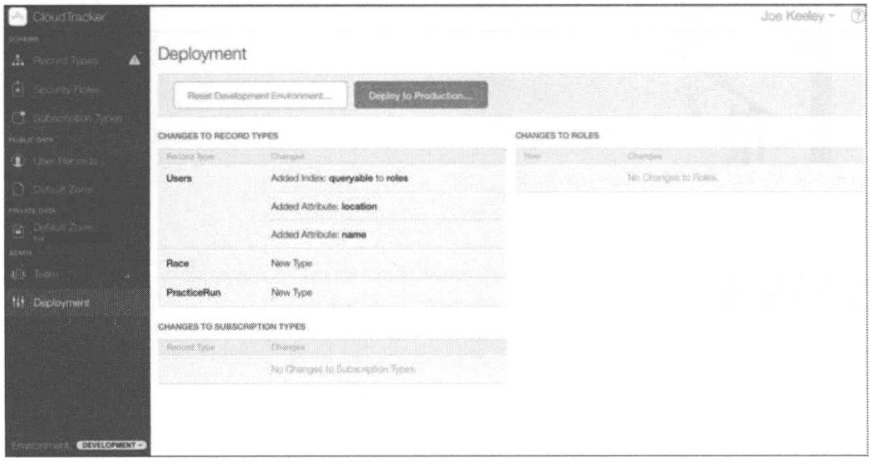


图 11-9　CloudKit dashboard: Deployment

11.9　小结

本章讨论了在应用中使用 CloudKit 的方法，介绍了如何对应用进行设置以使用 CloudKit，以及一些基础的 CloudKit 概念和如何执行基本的 CloudKit 操作，比如查询记录、从记录中显示数据、创建记录、更新和保存记录。本章之后还解释了如何设置订阅来通过推送接收数据更新通知，然后对如何自定义用户记录和如何执行用户发现过程进行了介绍。最后本章对使用 CloudKit dashboard 管理 CloudKit 进行了介绍。

第12章 extension

iOS 平台设计之初，应用就一直处于沙盒中运行。应用的运行不会受到第三方应用的影响，除非使用 URL 机制进行访问。从 iOS 8 开始，苹果公司第一次为开发者提供了在应用沙盒之外运行代码的功能。虽然 extension 的功能有限，不过它们为开发者的工具集增加了极大的灵活性。iOS 平台一共有 6 种 extension(Finder Sync 专用于 OS X)，每一种都有自己特有的功能。这里不包括苹果公司为了推广 WatchKit 而使用的 extension，我们会在本章后面的“Apple Watch extension”一节中进行讨论。

本章会介绍两个用得最广泛的 extension：第一个是 Today widget，它可以使应用在 Notification Center(通知中心)中快速推送一目了然的信息；第二个 extension 是 Apple Watch extension，它可以让 iOS 应用将信息发送到 Apple Watch 并接收来自 Apple Watch 的反馈。

12.1 extension 的类型

苹果公司将 extension 分为 7 种不同的类型(其中 6 种支持 iOS)。每种 extension 都有限制和约束，从而保护用户不被恶意行为影响，同时提供了特有的功能。为一个应用提供多个 extension 是可行的，本节将会对 extension 的这些类型进行介绍。

12.1.1 Today

Today extension 通常也称作 widget，一般显示在 iOS 设备的 Notification Center 中。之所以叫作 Today Extensions，是因为它出现的位置是在 Notification Center 的 Today 区域。这些 widget 被设计用来快速查看信息或者接收快速动作，比如按钮点击动作。

一些开发者已经突破了苹果公司对 Today extension 设置的限制。当广受欢迎的计算器应用 PCCalc 通过 Today extension 在 Notification Center 中添加一个完整的计算器时，苹果公司最初拒绝了 PCCalc 应用，不过后来在公众的呼吁下又改变了决定。

12.1.2 Share

Share extension 为用户提供了一种简便的方法，实现了在其他服务或 Web 网站之间共享内容。该 extension 将会向 iOS 程序内置的共享对话框中添加某些功能。比如，如果开发者想要同 Twitter 竞争创建一个新的社交服务，可以在原生 iOS 应用中写入一个 extension。这个 extension 可以在应用的共享页面直接将所需的内容同其他应用进行共享。

12.1.3 Action

Action extension 可以帮助用户从原始应用查看或转换内容。比如，如果有应用支持用户编辑选定的文本，可以创建一个新的 Action extension，从其他应用获取文本并放到指定位置。还比如，开发者可以撰写一个 extension，实现将西班牙语转换为英语的功能。Action extension 必须指定程序所需的数据类型，比如文本、视频或 PDF 等。

12.1.4 Photo Editing

Photo Editing extension 允许用户在 Photos.app 中编辑照片和视频。从某种意义上来说，这就创建了一个能够访问标准 Photo 应用的自定义修改插件。比如，开发者可以在照片库上创建一个使用自定义过滤器的 Photo Editing extension。在 Photos.app 应用中对照片进行任何改变都是可逆的，原始照片会被保存，这样就可以简单恢复原图了。

12.1.5 Document Provider

Document Provider extension 允许开发者在多个应用之间共享自定义类型的文件。Document Provider extension 会保存所有这些相关的文件。比如，Adobe 会发布一个 Photoshop Document extension，可以让应用处理和共享 PSD 文档。

12.1.6 Custom Keyboard

在 iOS 8 发布前，iOS 用户最大的诉求莫过于能够使用自定义键盘了。Custom Keyboard extension 可以完成这个任务，允许用户向系统键盘选择器添加各种第三方类型的键盘。Keyboard extension 仅能在用户向选中的文本输入框和其他文本输入区域输入数据时使用。

12.2 理解 extension

extension 的类型很多，不过它们的原理大致是相通的，即为用户提供了在其他应用中使用由开发者的应用所提供服务的一种方法。这对于 iOS 开发最初 6 年的发展来说是一次重大变革。

之前要实现类似 extension 的功能，用户需要对设备进行越狱。苹果公司做了极大的努力确保 extension 不成为用户眼中的恶意软件，所以在开发 extension 的过程中对其进行了各种各样的限制。要熟练掌握 extension 的开发，需要首先理解它们具有哪些功能。

extension 并不是独立应用，它们依附于其他的应用，称作 host app，用户将 extension 安装并激活在 host app 上。如果用户卸载了 host app，extension 也会一并移除。由于 extension 不是独立的应用，因此其不能在后台一直运行，直到用户从应用界面或者活动的视图控制器中选择了它，才会启动相应的 extension。host app 定义 extension 的参数并等待用户激活它。extension 激活后，执行完相应的任务后终止。extension 不会在后台持续运行，也不会长时间执行某一进程。

当 extension 运行时，在 extension 和包含它的应用间没有直接的通信。应用会传递所有 extension 需要的信息来执行具体的操作，之后等待 extension 执行完成。

注意

应用 extension 只能在用户有请求动作时运行，不能基于应用的某一状态使其自动启动或自动执行。

12.3　API 限制

在进行 iOS 开发时，应用 extension 有诸多特有的限制。处于安全和其他因素的考虑，很多 API 不能在 extension 中使用：

- extension 不能访问 sharedApplication 或其任何相关的方法，比如 openURL、delegate、applicationState，还不能访问推送或本地通知设置。
- 苹果公司使用 NS_EXTENSION_UNAVAILABLE 宏对特定的 API 进行了标记，带有标记的对象可以在 extension 中使用，这些 API 的例子包括日历事件和 HealthKit 交互等。
- extension 不能在 camera 或 microphone 中直接激活和使用。
- extension 不能长时间在后台执行任务，如果这样做，会被 App Store 拒绝上架或使程序在运行时终止。
- 从 extension 中不能访问 AirDrop，不过 extension 可以通过 UIActivityViewController 访问 AirDrop 发送的数据。

从代码的角度和用户的角度来看，extension 都是一种新技术。苹果公司知道要立即适应开发者的编码技术来确保最好的用户体验。

12.4　创建 extension

因为 extension 都是隶属于 host app 的独立对象，所以首先需要创建一个新的 host app。extension 可以被添加到任何常规 iOS 项目中。要在 Xcode 6 中创建一个新的 extension，依次选择 File|New|Target。这时会打开一个新的窗口(如图 12-1 所示)，在左边列表中选择 Application Extension，会出现适用于 iOS 的 6 种 extension 类型，选择一种希望添加到项目中的 extension 类型。

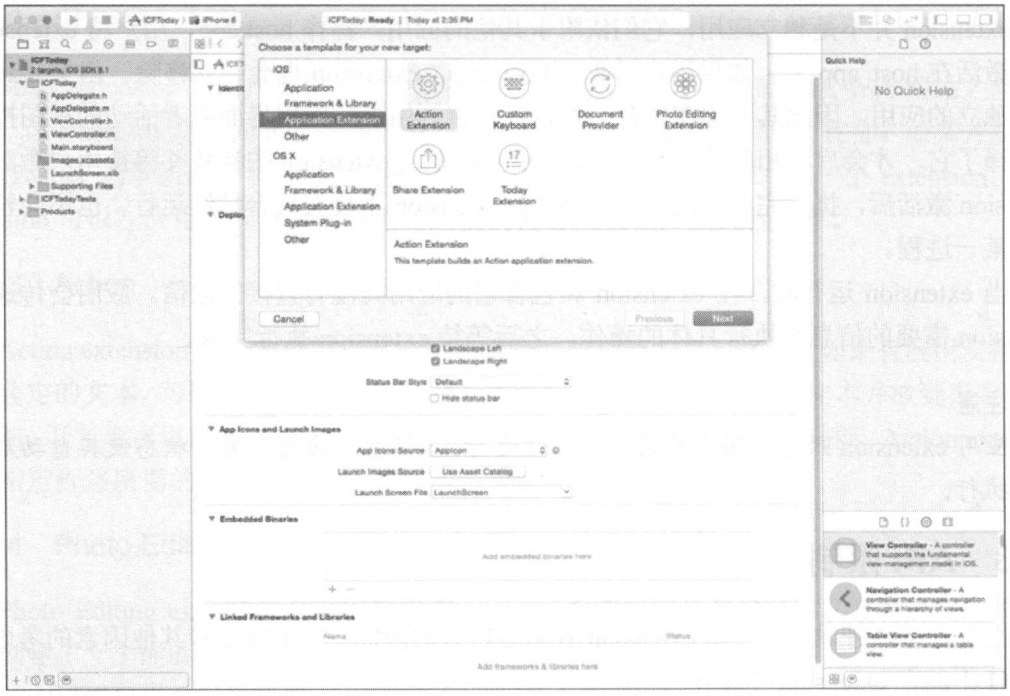


图 12-1　为项目添加一个新的 extension

创建好 extension 之后，项目目录中会生成一个以 extension 命名的文件组。这个文件组包含类文件和相关的故事板(storyboard)。如果选择的是 Today Extension，运行项目将会在 Notification Center 中显示一个 Hello World widget(如图 12-2 所示)。每个 extension 都有默认属性并且在测试时会有不同的显示。

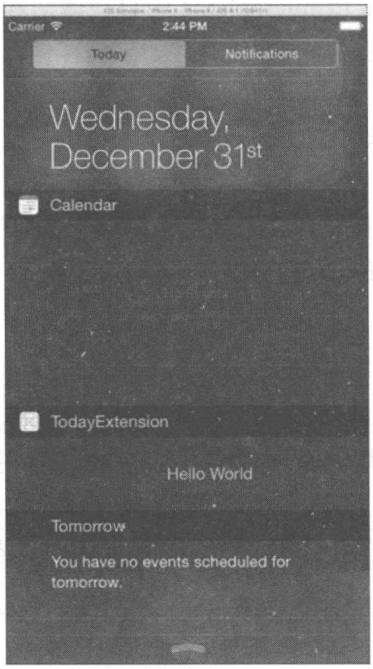


图 12-2　运行在 iOS 模拟器上的 Hello World Today extension

当 extension 运行时，需要特别注意的是，此时模拟器必须处于运行中，并且 host app 必须是安装好的。如果不满足上面的条件，程序会出错，如图 12-3 所示。如果 extension 没有成功载入或编译，尝试对项目进行清理并且重新编译 host app 之后再运行 extension。

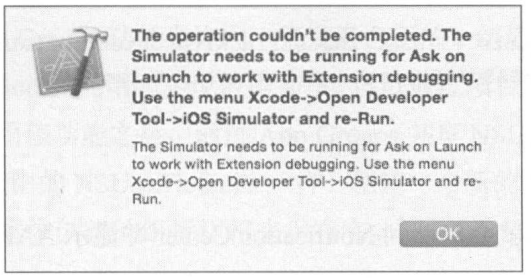


图 12-3　iOS 模拟器未启动而出现的 unable-to-launch-extension(无法启动 extension)错误

12.5　Today extension

本章前两个示例程序会创建一个 Today extension。ICFToday 是 host app，不过它没有任何功能，只是作为 extension 的 host app。对于 extension，使用 Yahoo 的 Finance API 获取最新的苹果公司股票价格并将其显示在 Notification Center 中。程序还给出了一个刷新按钮用于实现用户和 Today extension 之间的互动。

根据前面小节介绍的创建 extension 的步骤，需要创建一个新的项目并向其中添加一个新的 Today extension。extension 故事板被设置为一个单独的标签，用来显示股票价格，还带有一个按钮，用于刷新价格。

程序会默认创建一个新的方法 widgetPerformUpdateWithCompletionHandler。当 Today extension 或 widget 更新时会调用该方法，通常会在 Notification Center 显示时使用。可以在这个方法内设置一个断点，这样每次 Notification Center 刷新或重新载入时就会调用该方法。首先需要做的是为 extension 设置一个它会用到的显示尺寸，由于这里的 extension 非常小，因此只需要设置高度为 20 个像素即可。这个方法中的第二行代码用于调用刷新股票价格的方法。

```
-(void)widgetPerformUpdateWithCompletionHandler:
➥(void(^)(NCUpdateResult))completionHandler
{
        self.preferredContentSize = CGSizeMake(0, 20);
       [self refreshAction:nil];

       completionHandler(NCUpdateResultNewData);
}
```

Yahoo 提供了一个简单的 API 来获取任何一家上市公司的股票价格。下面方法中给出的 URL 就可以用来实现发起一个对当前 AAPL 股票价格的请求。结果会在一个 CSV 文件中返回，不过这个文件只有一行内容，可以将其看作明文。

```
-(NSString *)getStockPrice
{
```

```
    NSURL *url = [NSURL URLWithString:
    ➥@"http://finance.yahoo.com/d/quotes.csv?s=AAPL&f=a"];

    NSError *error = nil;

    NSString *quote = [NSString stringWithContentsOfURL:url
    ➥encoding:NSASCIIStringEncoding error:&error];

    return quote;
}
```

运行 extension，目标就会启动到 Notification Center 并显示 AAPL 的股票价格，如图 12-4 所示。

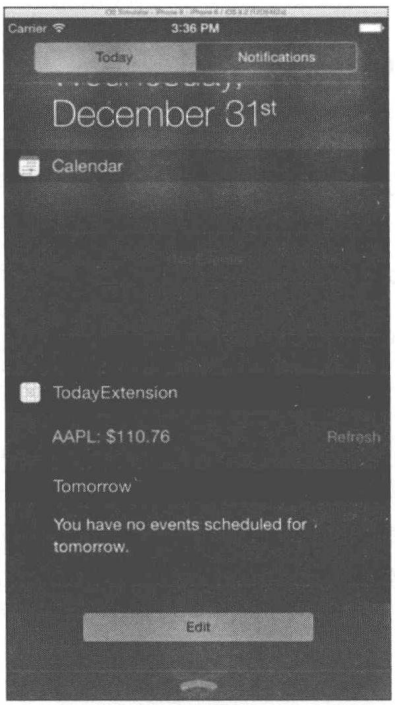


图 12-4 Today Extension 显示当前 AAPL 股票价格

12.6 在 host app 和 extension 间共享代码和信息

extension 不可以直接同它的 host app 进行通信，通常也不能共享代码。在实际使用时，这两个诉求往往是很常见的。要在 extension 和 host app 间共享代码，需要创建一个嵌入式框架。之后，可以从目标和代码中引用这个嵌入式框架，有效地实现 host app 和 extension 之间的共享。

要创建一个新的框架，在 Project Navigator 中选择项目并通过选择 Editor|Add Target 添加一个新的目标。选择 iOS,Framework & Library，从弹出的窗口中选择 Cocoa Touch Framework。现在可以将 Framework 添加到两个目标中，这样 extension 和 host app 间就可以共享类文件了。

通常，extension 需要在自身和 host app 间实现数据共享。由于两个目标处于不同的沙盒中运行，因此不能直接进行交互。不过通过 iOS 8 提供的一些新技术，简单的数据共享成了可能。

苹果公司支持的 App Groups 功能允许同一开发者名下的一组应用共享有限的数据。同样的技术也可以用在 extension 和应用间实现数据共享。可以通过项目编辑器页面中的设置来启用 App Groups 功能。启用该功能之后，通过 App Groups 共享 NSUserDefaults 就变得非常简单了，同老版本 iOS 系统中的 NSUserDefaults 一样。创建一个新的 NSUserDefaults 实例并使用在 App Groups 项目设置中创建的组标识符来保存它。

```
[[NSUserDefaults alloc] initWithSuiteName:@"<group identifier>"];
```

注意

当使用 App Groups 和 NSUserDefaults 时，特别需要注意的一点是两个目标的 NSUserDefaults 要始终存在，这一点和引用共享组 NSUserDefaults 是有区别的。每个目标默认保存的元素不会自动出现在共享的默认对象中。

12.7 Apple Watch extension

2014 年 9 月，在一次特殊的苹果发布会上，苹果公司宣布 Apple Watch 正式上市。在发布之前，有关苹果可能涉足手表产品的传言已经有好几年了。当 Tim Cook 在舞台上展示这款 Apple Watch 时，这一真相也终于尘埃落定。苹果公司的历史就是一部工业革命史，从最初的个人计算机到 MP3 播放器，从智能手机到笔记本电脑无不如此。Apple Watch 能否成为苹果公司下一个重大技术革命的代表产品，还需要拭目以待。

苹果公司已经为 Apple Watch 做出了两个有关软件开发的努力和承诺。第一个就是 Xcode 6.2 及以上版本全部支持 WatchKit，第二个就是 2015 年推出真正的原生态 SDK。苹果公司推广 Apple Watch 应用开发中心的第一个阶段是使用 extension。目前的 WatchKit 开发包同 Today extension 的使用原理类似，本章前面我们介绍过 Today extension 的用法。

注意

Apple WatchKit Development 需要 Xcode 6.2 Beta 3 或更新的 Xcode 版本。

要创建一个新的 WatchKit 项目，首先需要创建一个 host app。苹果公司目前还不允许将 Apple Watch 作为独立的应用。所有在 Apple Watch 上运行的第三方应用都必须隶属于另一个 iOS 设备。向一个应用中添加 WatchKit 组件的过程与之前添加 Today extension 类似，不同之处在于这时应该从目标菜单中选择 Apple Watch 对象，如图 12-5 所示。

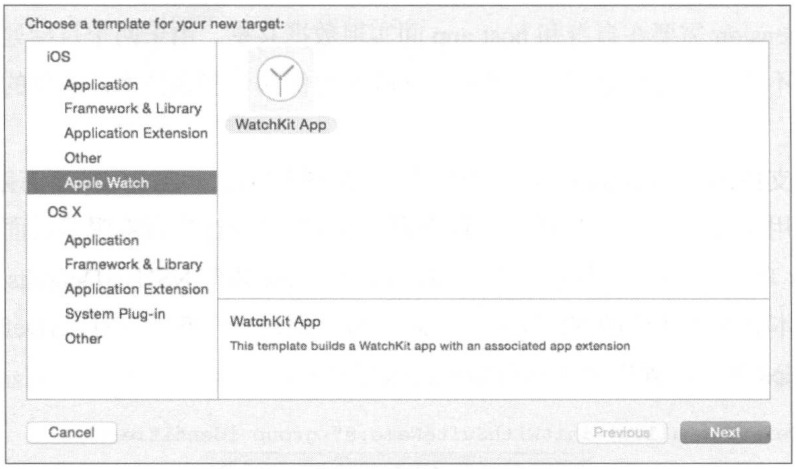


图 12-5 创建一个新的 WatchKit 应用

创建一个新的 WatchKit 应用对象时还需要生成一些新的文件，分别是 InterfaceController 和 NotificationController，这同 info.plist 的支持文件和 asset 目录一样。与 Today extension 不同的是，不需要创建 WatchKit 故事板。运行 WatchKit 应用将会出现一个空白的 Apple Watch 界面(如图 12-6 所示)。界面中只包含一个时钟和电量指示符。从 Add File 菜单中创建一个新的故事板，从可用的选项中选择 WatchKit 故事板。

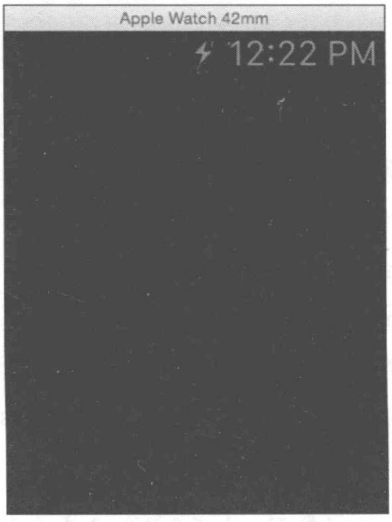


图 12-6 运行在 iOS 模拟器上的 WatchKit extension

WatchKit 故事板包含一个完全不同的控件集，和标准 iOS 开发控件不同。虽然有经验的 iOS 开发者还是能看出一些熟悉的元素，不过这些控件的外观和功能都有很大的区别，熟悉的一些控件(例如标签、表和标准控件)也都有，一些自定义控件(例如 steppers、date display 和 date pickers)也同样存在。

当使用故事板文件时，你很快就会发现元素同 iOS 的对齐位置不同，并且也没有标准的自动布局控件。不过随着 Xcode 6.2 版本的改进和 WatchKit 的优化，这些问题都在解决。相反，WatchKit 使用一些自己的位置控件，如图 12-7 所示。

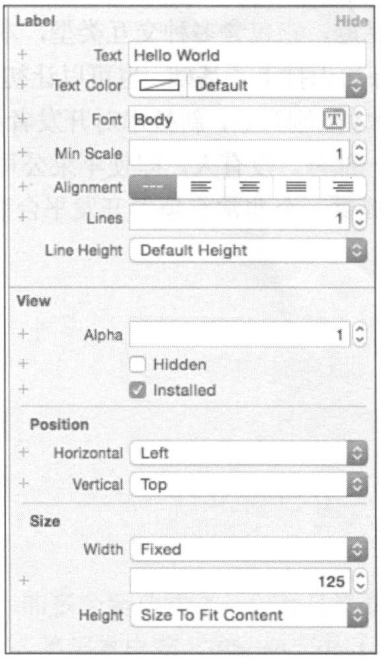


图 12-7　WatchKit 的位置控件同标准的 iOS 自动布局的区别

为控件建立关联动作的方法同标准的 iOS 开发模式一样。在示例程序中，创建一个新的标签和按钮。唯一的区别是在 Apple Watch 中 UILabel 称为 WKInterfaceLabel。WatchKit 模拟器在互动和响应上的延迟比较大。这一问题是否会遗留在真实的物理设备上或者只是模拟器的问题目前还不得而知。不过由于 WatchKit 的功能类似于运行在另一台设备上的 extension，因此有延迟也是可以预料到的。

这个简单的 WatchKit 应用还应该能够作为一个基础性的框架，读者可以根据自己掌握的 iOS 开发知识在此基础上进行扩展和添加功能。虽然这一平台还处于萌芽期，不过它已经可以提供很多功能了。

注意

Apple WatchKit Development 从发布之日起已经有了显著的变化，本章介绍的内容都基于 WatchKit 早期测试版本。每个版本的发布都会发现大量的错误，所以本章的资源尽量做到最简单，这样即使未来版本发生变化也能够很好地兼容。

12.8　小结

extension 是 iOS 系统提供的一个令人兴奋的新功能。extension 还需要继续优化以更好地适应未来的发展。extension 最基础的功能就是对之前 iOS 开发中的一些问题进行了更加灵活的处理。Apple Watch 中类似 extension 功能的效果能够看出苹果公司已经滞后于技术的发展，并慢慢在开放跨应用间的互动功能。保障、稳定和用户安全始终是苹果公司最首要考虑的问题，对于最初 iOS 发布时用户提出的种种诉求，苹果公司都在一步一步努力提供相应的功能，即使使用 extension，对于安全的关注也不能忽视。

extension 是一个相当大的话题，它包含多种交互类型，本章所介绍内容的难度应该比较适中，既为读者进一步挖掘更深知识打下了基础，也可以让初学者很好地进入 extension 开发的大门。iOS 开发者平台的革命才刚刚开始，新技术对开发者灵活性的改变已经非常显著了，extension 是这些重大跃进中的一部分。没有人，即使苹果公司自己也无法预测开发者使用这些技术能够做出什么，它只是提供一个非常有趣的开发平台而已。

第13章 Handoff

Handoff 是 iOS 8 版本中引入的多个新功能之一，而且最新的 Mac OS X Yosemite 系统也支持这个功能。它在台式计算机、笔记本电脑、iPhone、iPad 和 iPod touch 设备间提供了一种应用延续机制，这种延续性使这些设备在应用的使用上做到了无缝衔接。延续性会权衡设备的相近性和特殊性，例如从笔记本电脑发出拨号请求或发送短消息，或者让 iPad 使用附近的 iPhone，或者不断为另一个没有蜂窝网络连接的设备提供网络热点连接。Handoff 为移动应用或桌面应用提供了一种得到用户当前执行的活动(activity)通知的方法，这样就可以获取该活动并在其他设备上继续使用了。

13.1 示例程序

本章的示例程序是 HandoffNotes，它是一个基本的记事本应用，主要用于演示 Handoff 的一些基本功能。这个应用支持用户使用两种存储方法来保存记录列表，一种是 iCloud 键-值存储，另一种是 iCloud 文档存储。每条记录都包含标题、记录文本和创建日期。Handoff 功能就是指当用户使用任何一种保存方法编辑记录时都会得到通知。注意，示例程序需要设备成功登录有效的 iCloud 账户才能运行。

13.2 Handoff 基础

Handoff 需要两台运行 iOS 8 或更高版本 iOS 系统的设备，或是 OS X Yosemite 系统及其更高版本。每台设备都成功登录同一个 iCloud 账户，并且都要支持蓝牙 4.0(也称为 BTLE 或 Bluetooth Low Energy)。这些设置会让 iOS 和 OS X 系统自动执行设备间的匹配和 Handoff。

当用户正在设备上使用具有 Handoff 功能的活动时，Handoff 活动会被通知到其他附近的设备上。例如，如果用户正在 iOS 设备的 Safari 上查看一个 Web 页面，之后打开了她的笔记本电脑，OS X 将会在 Dock 中显示 Handoff 可用的活动，如图 13-1 所示。

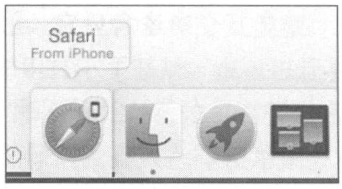


图 13-1　OS X Dock 中显示的 Handoff 通知

同样，如果用户在 HandoffNotes 中编辑一条记录并打开附近的其他 iOS 设备，锁屏界面左下方将会显示当前有一个支持 Handoff 功能的 HandoffNotes activity，在解锁滑动条旁边会显示 HandoffNotes 应用的图标，如图 13-2 所示。

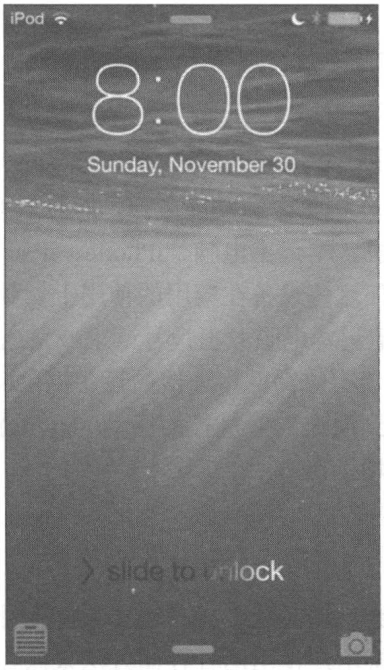


图 13-2　iOS 锁屏界面上显示的 Handoff 通知

在上面两个场景中，如果用户点击 OS X 中的 Dock 中的通知，以及在 iOS 锁屏界面上滑动应用图标，就会打开相应的目标应用。在 Safari 那个示例中，将会显示用户浏览的 Web 页面；在 HandoffNotes 示例中，应用将会导航到用户正在编辑的记录，并将所有正在进行的更新包含其中。

Handoff 使用一个简单的机制实现通知和这种延续性，即 NSUserActivity。在 iOS 中，很多 UIKit 类(包括 UIResponder)都具有 NSUserActivity 属性，所以可以通过响应链来设置和复制用户活动。让开发者可以选择最好的用户活动呈现方式，例如是否正在查看一些内容，编辑文本或执行其他的活动，以及直接将 Handoff 活动关联到用户界面。此外，UIDocument 也支持 NSUserActivity，这样基于文档的应用也可以支持 Handoff 功能。

NSUserActivity 实例包含两个关键的信息，一个是表示用户正在做什么的 activityType，另一个是包含关于用户活动指定信息的 userInfo 字典。activityType 应该是一个由反向 DNS 字符串组成的标识符，用来唯一标识一个活动。用户活动设置后，iOS 就可以开始通知了，

如图 13-1 和图 13-2 所示。当用户在另一台设备上继续使用该活动时，会将该活动传递给目标应用，目标设备会根据 activityType 和 userInfo 将其导航到正确的位置，并在其结束时回收该活动。注意，userInfo 字典对象并不包含数据传输机制。苹果公司建议 userInfo 字典应包含在其他设备上重启活动所需的最少信息，使用其他数据传输机制，例如 iCloud 来移动实际的数据。如果其他的数据传输机制无法满足要求，NSUserActivity 会在设备间建立一个数据流来实现数据的移动。

提示

在尝试为应用添加 Handoff 之前，应确保该 Handoff 已经使用支持 Handoff 的苹果应用(比如 Safari、Keynote 或 Pages)进行过测试。这样可以保证测试设备已经对 Handoff 所需的条件进行了正确配置，比如是否支持蓝牙 4.0 以及是否有匹配的 iCloud 账户。如果略过这一步骤，很可能要花费大量的时间在错误的位置查找 Handoff 问题。

13.3　实现 Handoff

对于非基于文档的应用，可以直接实现 Handoff。开发者必须确定应用使用哪个或哪些活动，以及 Handoff 应该如何显示。经常需要将 Handoff 导航到应用中的指定位置，以及传输一些进行中的更新数据。

在着手开发前，应用需要声明其支持的活动类型。在目标的 Info.plist 文件中，需要添加一个名为 NSUserActivityTypes 的条目。该条目应该包含由所支持的活动类型字符串组成的数组，其中每个元素都使用一个反向 DNS 标识符来表示应用支持的活动，如图 13-3 所示。

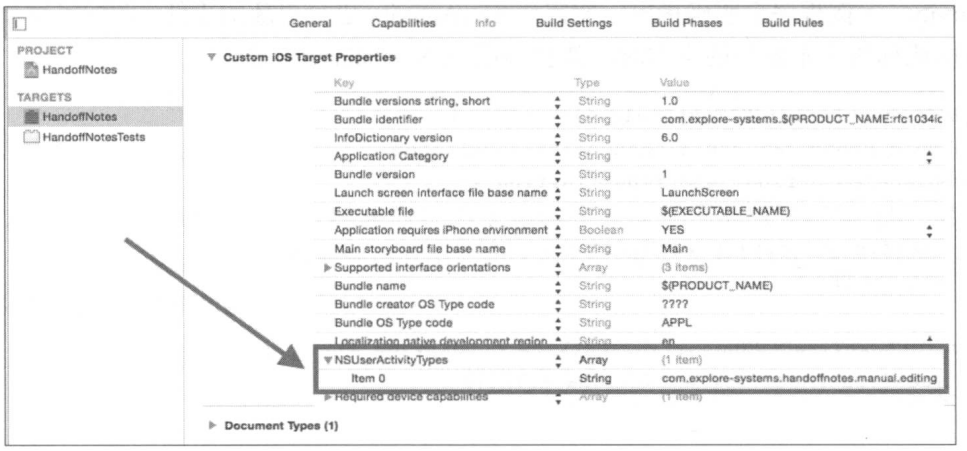


图 13-3　手动为 NSUserActivity 添加 NSUserActivityTypes 条目

在应用信息配置完成后，就可以实现自定义通知用户活动并在其他应用中继续该用户活动。

13.3.1　创建用户活动

示例程序使用 iCloud 键-值存储系统作为常规存储和传输机制，以演示手动实现 Handoff 的方法，实际发布的应用会采用更加复杂和兼容性更强的存储方法和数据传输机制。示例程

序以键-值存储的方式在数组中保存一个消息列表，并确保这个键-值存储跨设备兼容。要同正在编辑中的消息进行通信，示例程序只需要知道该消息在消息列表中的索引即可。此外，示例程序还将会得到有关正在通信的消息的更新进度。

当用户在 ICFManualNoteViewController 中编辑消息时，会创建一个 NSUserActivity 实例并为其配置好活动类型。该活动类型字符串必须同 Info.plist 文件中声明的字符串相匹配。示例程序在 viewWillAppear:方法中配置用户活动，不过其他应用应该根据用户实际的动作选择合适的时机对用户活动进行设置。

```
NSUserActivity *noteActivity = [[NSUserActivity alloc]
➥initWithActivityType:@"com.explore-systems.handoffnotes.manual.editing"];

noteActivity.userInfo = @{@"noteIndex":@(self.noteIndex),
                          @"noteTitle":self.note[@"title"],
                          @"noteText":self.note[@"note"]};

[noteActivity setDelegate:self];
 self.userActivity = noteActivity;
```

用户活动的配置会用到 userInfo 字典对象，它包含当前活动的信息，并为其分配了当前视图控制器对象作为委托，还为其分配了当前视图控制器的 userActivity 属性。完成这些操作之后，应用将会自动向其他设备发送用户活动通知。

由于用户正在执行一个活动，因此 userActivity 需要保持更新以反映活动的状态。在示例程序中，当用户更新消息文本和标题时，userActivity 就会更新。为了更有效率地更新 userActivity，示例程序会明确指出一些需要 userActivity 进行更新的事件。例如当用户改变消息文本框或标题文本框中的文本时，就会调用 setNeedsSave:方法。

```
-(BOOL)textField:(UITextField *)textField
➥shouldChangeCharactersInRange:(NSRange)range
➥replacementString:(NSString *)string {

    [self.userActivity setNeedsSave:YES];
    return YES;
}
-(void)textViewDidChange:(UITextView *)textView {
    [self.userActivity setNeedsSave:YES];
}
```

接下来，实现委托方法 updateUserActivityState:，更新用户活动。

```
-(void)updateUserActivityState:(NSUserActivity *)activity {
    activity.userInfo = @{@"noteIndex":@(self.noteIndex),
                          @"noteTitle":self.noteTitle.text,
                          @"noteText":self.noteDetail.text};
    NSLog(@"user info is: %@",activity.userInfo);
}
```

周期性地调用该委托方法，以保持用户活动的更新。当用户活动被通知时，会在其他设

备上看到消息，如图 13-2 所示。用户离开编辑视图后，用户活动也会自动无效并停止通知。如果用户活动的终止同视图控制器无关，则需要调用 invalidate 方法使其无效。

13.3.2 继续执行一个活动

当用户滑动图 13-2 中的图标准备继续执行一个活动时，应用委托得到通知，即用户将会继续执行一个活动。首先调用 application:willContinueUserActivityWithType:方法来确认应用是否可以继续执行活动。该方法会返回 YES 或 NO 来表示活动是否将会继续执行，以及通过任何逻辑代码判断继续执行的活动是否应该在该方法中实现。此外，这也是更新用户界面以向用户通知有活动需要继续的好时机，本例中这种继续并不是立即发生的。

接下来会调用 application:continueUserActivity:restorationHandler:方法来执行需要继续的活动。该方法会检查传入的 userActivity 对象的 activityType 类型，然后执行所有需要的设置或者将应用导航到正确的位置以继续执行活动。在设置完成后，可以调用 restorationHandler，使用一些额外的配置对活动进行恢复，其包含一个由多个视图控制器组成的数组。在示例程序中，该方法将会同步 iCloud 键-值存储来确保跨设备执行时的状态同步。之后会导航到手动记录列表，调用处理恢复任务的代码段，将手动记录列表的视图控制器对象传递给该方法，以执行剩下的导航和设置工作。

```
UIStoryboard *storyboard = [UIStoryboard storyboardWithName:@"Main"
                                                     bundle:[NSBundle mainBundle]];

ICFManualNoteTableViewController *manualListVC =
➥[storyboard instantiateViewControllerWithIdentifier:
➥@"ICFManualNoteTableViewController"];

[navController pushViewController:manualListVC animated:NO];

restorationHandler(@[manualListVC]);
```

当调用 restorationHandler 代码块时，对于数组中包含的所有视图控制器，都会调用 restoreUserActivityState:方法。在示例程序中，该方法会根据保存在用户活动的 userInfo 中的索引号导航到正确的消息。

```
if([activity.userInfo objectForKey:@"noteIndex"]) {
    NSNumber *resumeNoteIndex = [[activity userInfo] objectForKey:@"noteIndex"];

    NSIndexPath *resumeIndexPath =
    ➥[NSIndexPath indexPathForRow:[resumeNoteIndex integerValue]
                         inSection:0];
    [self.tableView selectRowAtIndexPath:resumeIndexPath
                                animated:NO
                          scrollPosition:UITableViewScrollPositionNone];

    [self performSegueWithIdentifier:@"showNoteDetail" sender:activity];
}
```

当程序使用 segue 切换到详情界面时，方法会将用户活动作为参数进行传递，这样在切换过程中就可以根据执行中的信息自定义消息了。

```
if([segue.identifier isEqualToString:@"showNoteDetail"]) {

    ICFManualNoteViewController *noteVC =
    ➥(ICFManualNoteViewController *)segue.destinationViewController;

    NSUInteger selectedIndex = [[self.tableView indexPathForSelectedRow] row];
    NSDictionary *note = [self.noteList objectAtIndex:selectedIndex];

    if([sender isKindOfClass:[NSUserActivity class]]) {
        NSUserActivity *activity = (NSUserActivity *)sender;
        note = @{@"title":activity.userInfo[@"noteTitle"],
                 @"note":activity.userInfo[@"noteText"],
                 @"date":note[@"date"]};
    }
    [noteVC setNote:note];
    [noteVC setNoteIndex:selectedIndex];
}
```

切换完成后，原设备上的消息详情界面将会反映当前消息的状态。由于原用户活动成功继续执行，因此就会终止向其他设备发送通知。要实现这一功能，消息详情视图控制器需要实现 userActivityWasContinued:方法，其中调用 userActivity 上的 invalidate 方法就可以从原设备停止发送通知。

```
-(void)userActivityWasContinued:(NSUserActivity *)userActivity {
    [self.userActivity invalidate];
}
```

注意，应用已经可以在设备间进行用户活动状态的通信了，用户可以毫不费力地在新的设备上继续完成原设备上未完成的任务。

13.4 在基于文档的应用中实现 Handoff

UIDocument 实例有一个自动设置的 NSUserActivity 属性，叫做 userActivity。如果文档保存在 iCloud 中，文档的 URL 将会包含在 userActivity 的 userInfo 中，这样接收设备就可以访问文档。在示例程序中，通过文档消息演示了这一方法。要查看具体的动作，单击上面菜单中的 UIDocument 按钮并选择一条存在的消息。当在 ICFDocumentNoteViewController 的 viewDidLoad 方法中已打开文档时，就创建了一个 NSUserActivity 并自动将其分配给文档，无需开发者自己编写代码。

```
self.noteDocument =
[[ICFNoteDocument alloc] initWithFileURL:[self noteURL]];

[self.noteDocument openWithCompletionHandler:^(BOOL success) {
```

```
    [self.noteTitle setText:[self.noteDocument noteTitle]];
    [self.noteDetail setText:[self.noteDocument noteText]];

    UIDocumentState state = self.noteDocument.documentState;

    if(state == UIDocumentStateNormal) {
        [self.noteTitle becomeFirstResponder];
        NSLog(@"opened and first responder.");
    }
}];
```

为了让 UIDocument 实例能够自动创建 NSUserActivity 实例，需要在项目中进行一些额外的设置。对于应用所支持的每一种文档类型，需要在 Info.plist 文件的 CFBundleDocumentTypes 中包含 NSUbiquitousDocumentUserActivityType 条目，如图 13-4 所示。

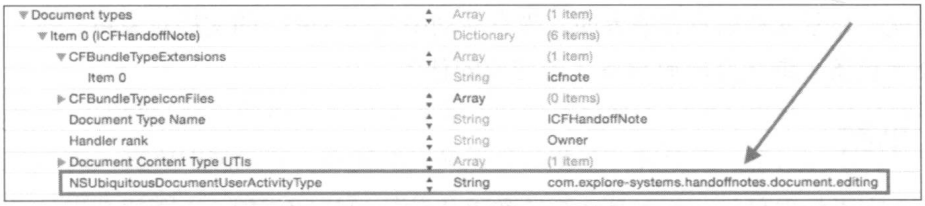


图 13-4　UIDocument 中为自动添加 NSUserActivity 设置 NSUbiquitousDocumentUserActivityType 条目

此外，还需要在 Info.plist 文件中为文档声明 Uniform Type Indicator (UTI)，如图 13-5 所示。

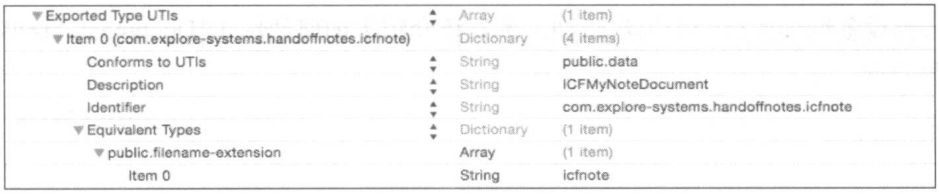


图 13-5　UIDocument 文件类型中的 Uniform Type Indicator (UTI)条目

当活动在另一台设备上继续运行时，接收设备应用的 application:continueUserActivity:restorationHandler:方法将会检查活动类型，之后将其导航到文档消息列表屏幕并调用 restorationHandler。

```
UINavigationController *navController =
➥(UINavigationController *)self.window.rootViewController;

[navController popToRootViewControllerAnimated:NO];

UIStoryboard *storyboard =
➥[UIStoryboard storyboardWithName:@"Main" bundle:[NSBundle mainBundle]];

ICFDocumentNoteTableViewController *documentListVC =
➥[storyboard instantiateViewControllerWithIdentifier:
➥@"ICFDocumentNoteTableViewController"];

[navController pushViewController:documentListVC animated:NO];
```

```
restorationHandler(@[documentListVC]);
```

在文档消息表视图控制器中，restorationHandler 将会调用 restoreUserActivityState:方法，得到传入的 userActivity 对应的 userInfo 对象中文档的 URL 地址，使用的键为 NSUserActivity-DocumentURLKey。

```
if([activity.userInfo objectForKey:NSUserActivityDocumentURLKey]) {

    self.navigateToURL =
    ➥[activity.userInfo objectForKey:NSUserActivityDocumentURLKey];

    [self performSegueWithIdentifier:@"showDocNoteDetail"
                              sender:activity];
}
```

此时就会显示文档消息详情界面，带有活动中文件 URL 的文档对象将会载入，并且用户可以在新设备上继续完成原设备上未完成的任务。

13.5 小结

本章主要介绍通过 Handoff 使应用在设备之间可以持续运行，介绍了使用 Handoff 的基本要求，例如需要具备蓝牙 4.0 并支持在同一 iCloud 账户下的 OS X Yosemite 或 iOS 8.0 以及更高版本的系统。之后还讲述了如何设置用户活动，使其可以向其他设备发送通知，以及如何从另一台设备继续运行一个用户活动。本章还介绍了如何使用 UIDocument 自动完成活动的创建和处理。

第14章

AirPrint

毫无疑问，印刷行业正在走一条和渡渡鸟、塔斯马尼亚虎相同的道路，只是现在还不会在一夜之间消失，物理印刷还有一段安详的晚年可以度过。和传真一样，打印可能还将会伴随我们的生活很长时间。从 iOS 4 开始，苹果公司提供了一个 SDK，并恰当地取名为 AirPrint，它的主要功能是实现物理印刷和无线打印机间的兼容。

通过过去几年的努力，在移动软件向传统行业进行扩展并建立信任的基础上，AirPrint 越来越受欢迎。销售终端应用和医疗处理应用都对打印物理文档有比较强烈的需求。

14.1 AirPrint 打印机

可选择的带有 AirPrint 功能的打印机非常有限，即使在 AirPrint 技术发布 4 年后也同样如此。不过大部分生产打印机的厂家都至少有几款产品是支持 AirPrint 的。苹果公司也通过技术报告发布了一个和 AirPrint 兼容的打印机名录(http://support.apple.com/kb/HT4356)。此外，一些第三方 Mac 应用可以让当前已有的打印机支持 AirPrint 功能，例如 Printopia 软件(在 www.ecamm.com/mac/printopia 上售价 19.95 美元)。

从 AirPrint 功能发布开始，就有大量的博客、文献和新手入门教程介绍有关测试 AirPrint 功能方面的知识。苹果公司似乎注意到这一需求并开始在开发者工具中绑定了一个名为 Printer Simulator(打印机模拟器)的应用(Developer/Platforms/iPhoneOS.platform/Developer/Applications)，如图 14-1 所示。Printer Simulator 可以让你的 Mac 为 iOS 模拟器设置一些打印机相关配置，使用这个工具，可以测试大部分兼容性问题。

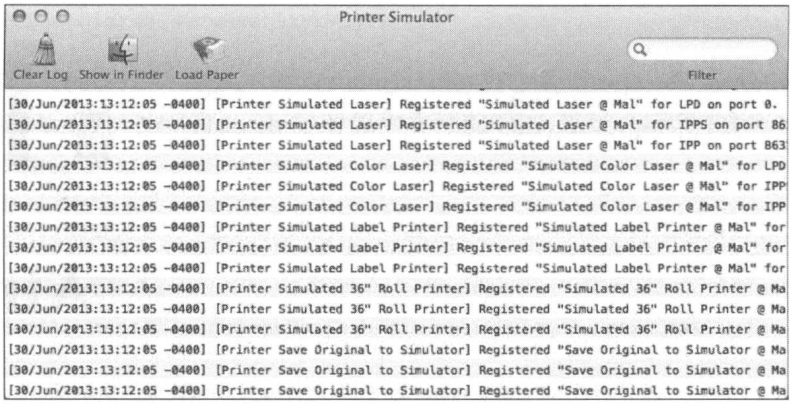


图 14-1　Printer Simulator 工具

本章的示例程序就是一个简单的 iPhone 应用，为用户提供两个视图，如图 14-2 所示。第一个视图是一个简单的文本编辑器，用来演示对页面和文本的编辑。第二个视图是一个 Web 浏览器，用来演示如何对已呈现的 HTML 内容和截屏得到的 PDF 内容进行打印。

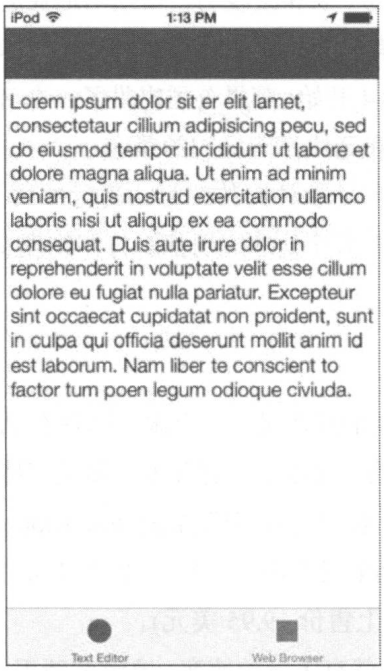


图 14-2　AirPrint 示例程序的打印功能

Print 应用很简单，它不包含大量的复杂代码。它基于苹果 TabBarController 默认项目，包含两个标签页。文本编辑器包含一个带有两个按钮的 UITextView，其中一个按钮用于隐藏键盘，另一个按钮用于开始打印过程。Web 浏览器视图包含一个 Print 按钮，不过它还需要前面文本框中的 URL。

14.2　测试 AirPrint

在将 AirPrint 功能开放给用户使用前，很重要的一点是在软件中对 AirPrint 进行测试，

因为可能有些用户使用的 iOS 版本并不支持 AirPrint 功能。在示例程序中，会在 application:didFinishLaunchingWithOptions:方法中进行测试，如下所示：

```
if(![UIPrintInteractionController isPrintingAvailable])
{
    UIAlertView *alert = [[UIAlertView alloc]
    ➥initWithTitle:@"Error"
    ➥message:@"This device does not support printing!"
    ➥delegate:nil
    ➥cancelButtonTitle:@"Dismiss"
    ➥otherButtonTitles:nil];

    [alert show];
}
```

注意

AirPrint 被定义为 UIKit 的一部分，所以不需要额外导入头文件或框架。

14.3 打印文本

无论是对于 AirPrint 还是所有的打印任务来说，文本打印都是最常见的应用场景。如果你对打印相关的术语不是很熟悉的话，可能会认为使用 AirPrint 打印文本有些复杂。下面的代码来自示例程序的 ICFFirstViewController.m 类。首先我们把它作为一个整体来看，在后面的小节中我们会逐行进行讨论。

```
-(IBAction)print:(id)sender
{
    UIPrintInteractionController *print =
    ➥[UIPrintInteractionController sharedPrintController];

    print.delegate = self;

    UIPrintInfo *printInfo = [UIPrintInfo printInfo];
    printInfo.outputType = UIPrintInfoOutputGeneral;
    printInfo.jobName = @"Print for iOS";
    printInfo.duplex = UIPrintInfoDuplexLongEdge;
    print.printInfo = printInfo;

    print.showsPageRange = YES;

    UISimpleTextPrintFormatter *textFormatter =
    ➥[[UISimpleTextPrintFormatter alloc] initWithText:[theTextView
    ➥ text]];

    textFormatter.startPage = 0;

    textFormatter.contentInsets = UIEdgeInsetsMake(36.0, 36.0,36.0,
    ➥36.0);
```

```
    textFormatter.maximumContentWidth = 540;

    print.printFormatter = textFormatter
}
```

前面的代码用来获取文本视图的内容，并以 8 1/2 格式对其进行打印，即纸张大小为 11 英寸且四周边距为 0.5 英寸。在每次准备打印操作之前，首先需要做的是创建一个到 sharedPrintController 的引用，如以下代码所示：

```
UIPrintInteractionController *print = [UIPrintInteractionController
➥sharedPrintController];
```

在示例代码的下一行中，设置了一个委托。为打印操作使用的委托方法将在下面的“UIPrintInteractionControllerDelegate”一节中进行定义。

14.3.1 打印信息

下一步就是配置打印信息。这里为如何设置打印任务指定了许多控件和修改器。可以使用程序提供的单例 UIPrintInfo 来获取一个新的默认对象 UIPrintInfo。

示例程序中为打印信息设置的第一个属性是 outputType。在例子中，输出设置为 UIPrintInfoOutputGeneral。这样设置意味着打印任务的输出可以是文本、图像和图片的混合，同时指定使用默认纸张尺寸。outputType 的其他选项还包括 UIPrintInfoOutputPhoto 和 UIPrintInfoOutputGrayscale。

下一个设置的属性为 jobName。它是一个可选字段，用于在打印中心应用(print center app)中识别具体的打印任务。如果没有设置 jobName，则默认为应用的名称。

打印机行业中有一个术语叫“Duplexing”，意思是打印机如何处理双面打印的问题。如果打印机不支持双面打印，则忽略这些属性。如果希望禁止双面打印，则使用 UIPrintInfoDuplexNone 属性即可。如果想要使用双面打印，有两个选项需要设置：第一个为 UIPrintInfoDuplexLongEdge，沿着纸张的长边翻转；第二个为 UIPrintInfoDuplexShortEdge，顾名思义，也就是沿着纸张的短边进行翻转。

示例程序中除了使用 printInfo 属性之外，还用到了两个属性。第一个是 orientation，用于为打印指定水平方向还是垂直方向。第二个是 printerID，用于表示当前使用的打印机是哪一个。printerID 通常会自动指定为执行上一个打印任务的打印机，可以使用 UIPrintInteractionControllerDelegate 进行设置。

在为打印任务配置完 printInfo 之后，需要在 UIPrintInteractionController 上对相关的属性进行设置，如下所示：

```
print.printInfo = printInfo;
```

14.3.2 设置页面范围

很多时候，打印任务会包含许多页，有时候用户希望选择需要打印的页码。可以通过 showsPageRange 属性进行设置。当设置该属性为 YES 时，则允许用户在打印选择界面选择需要打印的页码。

```
print.showsPageRange = YES;
```

14.3.3 UISimpleTextPrintFormatter

在配置完 printInfo 之后，UIPrintInteractionController 对于即将开始的打印任务类型可以提前知道，不过对于打印什么则一无所知。所打印的数据的类型则使用打印格式器进行设置，本节主要讨论针对文本的打印格式器。在后面的小节中我们会深入讨论其他打印格式器。

```
UISimpleTextPrintFormatter *textFormatter =
➥[[UISimpleTextPrintFormatter alloc] initWithText:[theTextView
➥ text]];

textFormatter.startPage = 0;

textFormatter.contentInsets = UIEdgeInsetsMake(36.0, 36.0, 36.0, 36.0);

textFormatter.maximumContentWidth = 540;

print.printFormatter = textFormatter;
```

当创建一个新的 UISimpleTextPrintFormatter 实例时，会使用将要打印的文本内容为对象分配内存空间并初始化。在示例程序中，会打印 UITextView 中出现的所有文本内容。

示例程序中设置的第一个属性为 startPage，该属性以 0 作为首页的索引值。示例程序将从第一页开始打印(索引值为 0)。

下面对 contentInsert 进行设置。在打印纸张中，1 英寸的值为 72 单位。示例程序在上下左右 4 个方向都给出半英寸的边界。此外，最大宽度设置为 504，即指定 7 1/2 英寸打印宽度(72.0×7.0)。

还有两个属性是示例程序中没有用到的。font 属性可以让你为打印的文本内容指定一个 UIFont 值。font 是可选属性，如果不指定任何值，则使用系统默认的 12 号字。还可以用 color 属性来指定文本的颜色。如果不指定 color 属性，则默认使用[UIColorblackColor]。

在配置完 textFormatter 后，需要将其设置为 UIPrintInteractionController 对象的 printFormatter 属性。

14.3.4 错误处理

总是能够妥善处理程序中的错误是很重要的，尤其是对于打印正在进行时所出现的错误更要注意。在打印过程中，有很多开发者无法控制的状况，诸如没有打印纸或者打印机没有打开等问题。

示例程序定义了一个新的 block，叫作 completionHandler，用于处理所有从打印任务返回的错误。在下一小节中，在开始一个打印任务时，completionHandler 会作为一个参数传入。

```
void(^completionHandler)(UIPrintInteractionController *,BOOL, NSError
➥*) = ^(UIPrintInteractionController *print,BOOL completed, NSError
➥*error)
{
   if(!completed && error)
```

```
        {
            NSLog(@"Error!");
        }
    };
```

14.3.5　开始一个打印任务

在创建好新的 UIPrintInteractionController 后，为其指定 printInfo 和 printFormatter，以及创建一个用于处理返回错误的 block。使用上一小节中创建的 completion block 对 UIPrintInteractionController 对象调用 presentAnimated:completionHandler:方法。此时会向用户呈现 Printer Options 视图，如图 14-3 所示。

```
[printpresentAnimated:YES completionHandler:completionHandler];
```

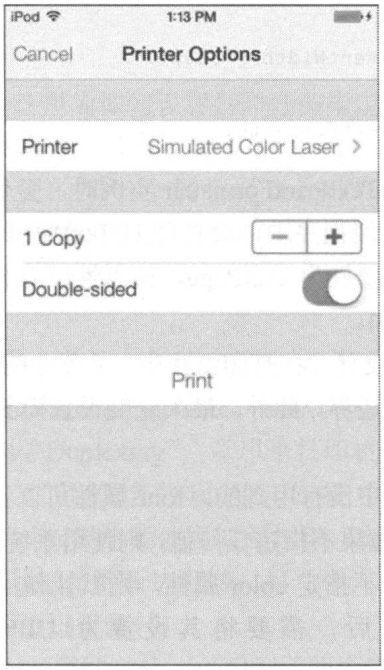


图 14-3　打印机的打印选项：多页打印、支持双面打印

根据所选打印机和文本大小的情况，程序给出的选项是不同的。例如，如果打印任务只有一页纸，则用户不会看到设置打印范围的选项。同样，如果打印机不支持双面打印，则对应的选项是被禁用的。

14.3.6　打印机模拟器反馈

如果用到“AirPrint 打印机”一节中讨论的打印机模拟器应用进行打印，在打印任务完成后，预览界面会显示一个新的页面(或者默认的 PDF 查看程序)，展示最终打印的效果。示例程序中用到的打印详情和打印格式如图 14-4 所示。

图 14-4　使用打印机模拟器时显示的预览界面，注意高亮显示的页边距

14.4　打印中心

同台式计算机一样，iOS 平台也为开发者提供了一种能够同当前打印机进行交互的方法。虽然所有的打印任务都是在 iOS 设备上激活的，不过会有一个新的应用出现在激活打印应用的附近，即如图 14-5 所示的 Print Center 应用(打印中心应用)。打印中心应用在 iOS 7 中被删除了，现在有关打印反馈的操作直接在打印过程中进行处理。

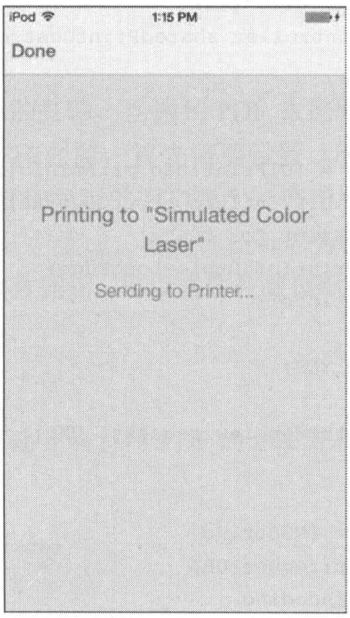


图 14-5　打印中心应用提供的关于当前打印任务的信息

UIPrintInteractionControllerDelegate

在之前的“打印文本”一节中我们提到过，可以为 UIPrintInteractionController 对象提供一个委托。在示例程序中我们使用了两种可能的委托回调，表 14-1 给出了有关这些委托方法的介绍。

表 14-1 可用的 UIPrintInteractionControllerDelegate 方法

方 法 名	描 述
printInteractionControllerWillPresentPrinterOptions:	对于用户配置打印任务将提供选项的 API
printInteractionControllerDidPresentPrinterOptions:	对于用户配置打印任务已经提供选项的 API
printInteractionControllerWillDismissPrinterOptions:	将要删除的打印选项视图
printInteractionControllerDidDismissPrinterOptions:	已经删除的打印选项视图
printInteractionControllerWillStartJob:	打印任务开始时即调用该方法
printInteractionControllerDidFinishJob:	打印任务已完成或失败时调用该方法

14.5 打印呈现的 HTML

打印呈现的 HTML 会通过打印格式器自动进行处理，基本同打印普通文本内容的方法一样。下面的方法用于处理 HTML 字符串的打印，该内容从示例程序的 UIWebView 获得。你可能会注意到这其实和之前例子中打印文本的方式类似。先将这个方法当作整体来看，在后面的小节中将详细对其进行讨论。

```
-(IBAction)print:(id)sender
{
    UIPrintInteractionController *print =
    ➥[UIPrintInteractionController sharedPrintController];

    print.delegate = self;

    UIPrintInfo *printInfo = [UIPrintInfo printInfo];
    printInfo.outputType = UIPrintInfoOutputGeneral;
    printInfo.jobName = @"Print for iOS";
    printInfo.duplex = UIPrintInfoDuplexLongEdge;
    print.printInfo = printInfo;

    print.showsPageRange = YES;

    NSURL *requestURL = [[theWebView request] URL];
    NSError *error;

    NSString *contentHTML = [NSString
    stringWithContentsOfURL:requestURL
    encoding:NSASCIIStringEncoding
    error:&error];
```

```
    UIMarkupTextPrintFormatter *textFormatter =
    ➥[[UIMarkupTextPrintFormatter alloc]
    ➥initWithMarkupText:contentHTML];

    textFormatter.startPage = 0;

    textFormatter.contentInsets = UIEdgeInsetsMake(36.0, 36.0, 36.0,
    ➥36.0);

    textFormatter.maximumContentWidth = 540;
    print.printFormatter = textFormatter;


    void(^completionHandler)(UIPrintInteractionController *,BOOL,
    ➥NSError *) = ^(UIPrintInteractionController *print,BOOL
    ➥ completed, NSError *error)
    {
        if(!completed && error)
        {
            NSLog(@"Error!");
        }
    };

    [print presentAnimated:YES completionHandler:completionHandler];
}
```

实现文本打印功能时，第一步需要完成的就是创建一个对 UIPrintInteractionController 对象的新引用。下一步就是为即将开始的打印任务设置 printInfo。本段代码中对打印的处理同之前的例子没有任何区别，欲了解这些属性的详细信息可以参考上面的小节。

14.6 打印 PDF

AirPrint 本身支持 PDF 文件的打印，当遇到 PDF 数据时使用 PDF 文件类型被认为是最简单的。在 PDF 文件可以被打印之前，首先需要设置 UIPrintInteractionController 和相关 UIPrintInfo。这些设置同之前“打印呈现的 HTML”一节中用到的方法一样。在示例程序中，PDF 从上一节创建的 UIWebView 中生成，不过你还可以为程序指定任何 PDF 数据的路径。在使用 renderInContext 方法创建数据后，可以将图片数据赋给 printingItem。这个方法还可以用于打印任何 UIImage 数据。

注意

示例程序现在还不能将任何动作和打印 PDF 的方法进行关联。可以为这个方法设置一个按钮，用于激活它。

```
-(IBAction)printPDF:(id)sender
{
    UIPrintInteractionController *print =
```

```
    [UIPrintInteractionController sharedPrintController];

    print.delegate = self;
    UIPrintInfo *printInfo = [UIPrintInfo printInfo];
    printInfo.outputType = UIPrintInfoOutputGeneral;
    printInfo.jobName = @"Print for iOS";
    printInfo.duplex = UIPrintInfoDuplexLongEdge;
    print.printInfo = printInfo;

    print.showsPageRange = YES;

    UIGraphicsBeginImageContext(theWebView.bounds.size);

    [theWebView.layer
    renderInContext:UIGraphicsGetCurrentContext()];

    UIImage *image = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();

    print.printingItem = image;

    void(^completionHandler)(UIPrintInteractionController *,BOOL,
    ➥NSError *) = ^(UIPrintInteractionController *print,BOOL
    ➥completed,NSError *error)
    {
        if(!completed && error)
        {
            NSLog(@"Error!");
        }
    };

    [print presentAnimated:YES completionHandler:completionHandler];
}
```

14.7 小结

本章介绍了如何使用 AirPrint 功能从 iOS 设备上打印文档、图片和 HTML 文本。你已经牢固地掌握了有关打印任务的所有知识，包括创建新的打印任务、提供打印所需的元素、设置输出格式、处理错误以及如何与不同的打印机进行交互。本章给出的示例程序会引导你完成从打印普通文本到 HTML 文本及 PDF 数据的学习。现在你可以很自信地在未来的 iOS 项目中添加 AirPrint 支持了。

第15章

开始使用 Core Data

对于许多应用来说，需要将本地保存和获取的数据持久用在其他会话中。从 iOS 3.0 开始，Core Data 就负责完成这项工作。Core Data 是一个非常强大的对象数据库，提供了非常健壮的数据存储和管理功能。

Core Data 来源于 NeXT 的 Enterprise Object Framework (EOF)，EOF 的功能是将对象映射到关系数据库。它为对象编写业务逻辑提供了极大便利，因为它不必创建数据库或撰写专门的持久性逻辑。主要是因为这样就不需要编写太多的代码，而这些代码恰恰又都是关注于应用的执行而非数据库的需要。EOF 可以支持很多关系数据库。由于 Core Data 最初是为了支持 Mac OS X 中的单用户应用而创建的，因此 Core Data 将数据保存在一个内嵌的关系数据库 SQLite 中。SQLite 具有 SQL 数据库的各种优势，同时又是 SQL 的轻量化版本，在维持数据库服务器连接时的开销也很小。

Core Data 包括以下几点特性：

- 使用可视化模型编辑器模块化数据对象
- 对象架构变化时自动处理和手动处理间转换的工具
- 在对象间建立关系(一对一、一对多、多对多)
- 在不同的文件和文件格式中保存数据
- 对象特性的有效性验证
- 查询和排序数据
- 数据延迟加载
- 同 iOS 表视图和集合视图紧密交互
- 通过提交和撤消功能管理相关对象的变化

最初接触 Core Data 可能会感到困难和无所适从，甚至有专门的整本书用来介绍 Core Data 的内容，苹果公司的官方文档对 Core Data 的全部扩展内容都进行了深入详细的介绍，所以阅读起来也非常困难。不过大部分的应用并不需要用到 Core Data 提供的所有功能。本章旨在介绍应用在开发时所需的大多数常见的 Core Data 功能。

本章将介绍如何为项目添加 Core Data 功能，并通过示例程序展示如何实现一些常见的

功能，包括如何建立数据模型、如何填充起始数据、如何使用取回结果控制器(fetched results controller)在表视图中显示数据。本章还将演示如何添加、编辑和删除数据，如何取回数据以及如何使用谓词获取特定的数据。掌握了这些知识，你就为快速在应用中添加 Core Data 功能打下了良好的基础。

15.1 Core Data 的选择

在开始讨论 Core Data 之前，了解一下应用的持久性需求并将其同可用的持久性方法进行比较非常有必要。如果应用不需要实现 Core Data 就能满足需求，就可以省掉一些开发任务，同时减少应用的复杂性，数据的长期保存也会变得简单。对于想要使用持久性数据的 iOS 开发者而言，有如下一些选择：

- NSUserDefaults：这个方法通常用来保存应用参数。NSUserDefaults 的功能同带有键-值存储的 NSDictionary 对象非常类似，保存的值可以是 NSNumber、NSString、NSDate、NSData、NSDictionary、NSArray 或其他遵循 NSCoding 协议的任何对象。如果一个应用的持久性需求可以使用键-值配对、字典对象和数组等来满足，则 NSUserDefaults 是一个很好的选择。
- iCloud 键-值存储：这个方法的用法与 NSUserDefaults 类似，只是在需要支持 iCloud 并且涉及设备间数据同步时才有所不同。其对于存储和同步多大的数据有着严格的限制。如果应用的持久性需求可以使用键-值配对、字典对象和数组来满足且需要进行设备间的数据同步，则 iCloud 键-值存储是一个很好的选择。
- 属性列表文件(plist)：NSDictionary 和 NSArray 分别支持从用户指定的属性列表文件中读取和向该列表文件中写入的功能，属性列表是一个支持 NSNumber、NSString、NSDate、NSData、NSDictionary 和 NSArray 的 XML 文件格式。如果一个应用的持久性需求可以使用字典或数组满足，则属性列表文件是一个很好的选择。
- 编码/归档：NSCoder 和 NSKeyedArchiver 支持将任意对象图表保存为二进制文件。这些选项需要在每一个要保存的自定义对象上实现 NSCoder 方法，需要开发者对保存和加载操作进行管理。如果一个应用的数据持久性需求可以使用大量的自定义对象满足，则可以选择 coder/archiver(编码/归档)方法。
- 结构化文本文件(JSON、CSV 等)：例如 CSV 或 JSON 可以用于存储数据，尤其是 JSON 可以利用内置的序列化和反序列化机制(参加第 9 章“JSON 的使用和解析”以了解更多内容)；不过任何结构化文本方法都需要额外添加支持自定义模型对象和所有搜索、筛选功能。如果应用的数据持久性需求可以使用大量自定义对象或字典和数组满足，则可以使用结构化文本文件这个方法。
- Direct SQLite：使用 C 语言库 libsqlite，应用可以直接同 SQLite 数据库进行交互。SQLite 是一个不需要服务器的内置关系数据库，支持大部分使用 SQL92 描述的 SQL 语言。所有能够使用 SQL 创建的数据持久性逻辑都可以通过 SQLite 在 iOS 应用中使用，包括定义数据库表和关系、插入数据、查询数据、更新和删除数据。这个方法的缺点在

于应用需要在应用对象和 SQL 文件中进行数据映射，需要编写 SQL 查询语言来获取和保存数据。如果需要对保存的对象进行追踪，还需要额外编写代码。

- Core Data：它提供一种直接使用 SQLite 最灵活的方法，使应用从数据库运行机制中脱离出来。如果应用需要大量数据，需要在不同对象间保持关系，或者需要快速简单地访问特定的对象和对象组，Core Data 是一个非常好的选择。

让 Core Data 成为优秀的数据持久性机制的一个最主要的特性就是 NSFetchedResultsController。通过 NSFetchedResultsController，表视图或集合视图可以简单地绑定到数据，并且在数据发生变化时及时得到通知。表视图和集合视图都内置了单元格插入、删除和移动的动画方法，当 NSFetchedResultsController 检测到相关数据变化时就会调用这些方法。当应用从数据库获取数据并将其保存在本地时这一特性非常有效。本章将介绍如何在应用中实现 NSFetchedResultsController 功能。

15.2　示例程序

本章的示例程序为 MyMovies，它是一个基于 Core Data 的应用，帮助你记录所有的物理介质的影片。如果你将影片借给别人，会记录借给了谁和借出时间，如图 15-1 所示。

该示例程序有 3 个标签页，分别是 Movies、Friends 和 Shared Movies。Movies 标签页在一个表视图中显示用户已添加和跟踪的影片的列表。表视图被分成两个部分，用于演示数据是如何通过取回结果控制器来进行区分的。用户可以在这个页面添加新的影片，也可以编辑已有的影片。Friends 标签页列出了可以共享影片的好友和借影片的好友，在该标签页上可以添加和编辑好友。Shared Movie 标签页显示当前哪部影片正在与好友共享。

图 15-1　示例程序：Movies 标签页

15.3 开始一个 Core Data 项目

要开始一个新的 Core Data 项目，打开 Xcode，从菜单中依次选择 File | New |Project。Xcode 将会显示一些项目模板，如图 15-2 所示。

创建一个 Core Data 项目最快的方法是选择 Master-Detail Application 模板。单击 Next 为新项目设置一些选项，确保选中了 Use Core Data(如图 15-3 所示)，这样就可以确保项目具备 Core Data 功能。

单击 Next 之后，Xcode 会创建项目模板。项目模板包括一个“master”视图控制器，它包含一个由 NSFetchedResultsController 生成的表视图和一个指定的用于将 Core Data 和表视图进行配对的控制器。项目模板包括一个“detail”视图用于显示单独的数据记录。在示例程序中，master 和 detail 视图根据项目的描述已被重命名。

注意

要向一个已有的项目快速添加 Core Data，需要创建一个支持 Core Data 的空白模板，并将下面一节“Core Data 环境”中介绍的元素复制到已有的这个项目中。将新的托管对象模型文件添加到项目，并确保在已有的项目中添加了 Core Data 框架。

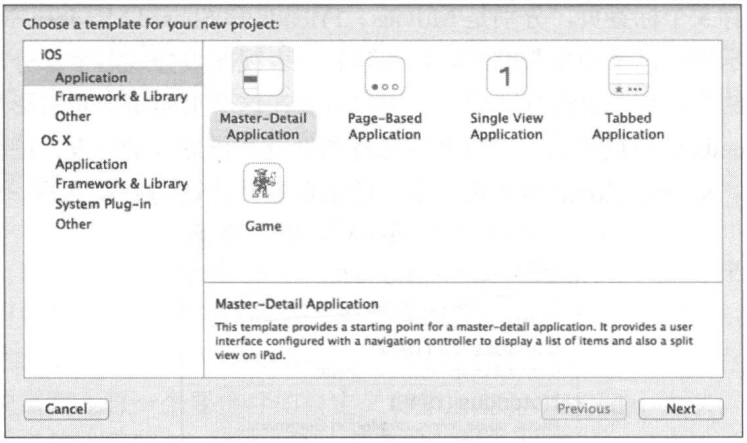


图 15-2 Xcode 新项目模板选择

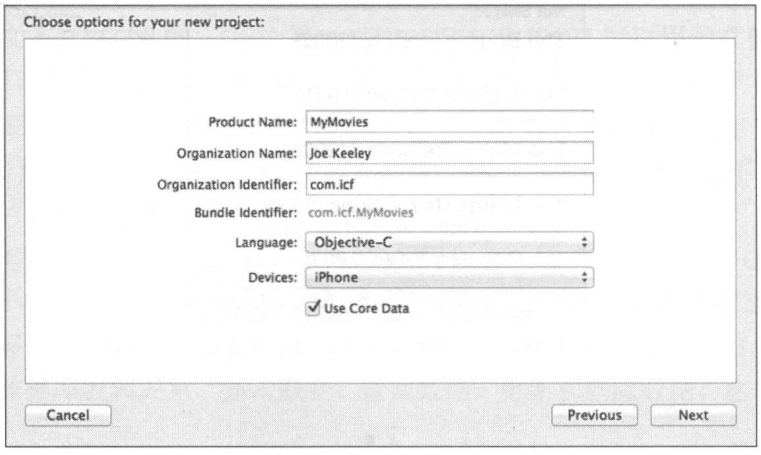


图 15-3 Xcode 新项目选项

Core Data 环境

项目模板会在 UIApplicationDelegate 协议的实现类中为项目设置 Core Data 环境；在示例程序中这个类称为 ICFAppDelegate。项目模板对于 Core Data 环境用到的每个属性使用延迟加载机制，所以属性只有在需要时才会被加载。

加载 Core Data 环境的过程是在应用第一次引用托管对象上下文时开始的。托管对象上下文(NSManagedObjectContext)是托管对象的工作区域。要创建新的对象、删除对象或查询已有对象，应用都要同托管对象上下文进行交互。此外，托管对象上下文还可以对有关的变化进行管理。例如应用可能插入一些对象、更新一些对象、删除一些对象以及保存所有的变化，甚至还可以将不需要的变化回滚。

同一时间可以使用不止一个托管对象上下文独立或一起进行工作。试想应用从 Web 服务器获取一些新的数据，需要对其进行展示。这时一个托管对象上下文可能需要在主线程进行查询和显示已有数据，另一个托管对象上下文可能作为主上下文的子对象创建。应用完成数据的导入后，可以快速自动地将两个托管对象上下文进行合并，并置于后台上下文进行处理。Core Data 有足够的能力处理同一对象在两个上下文中更新的情况，并可以将这些更新进行合并。

注意，需要对模板设置进行修改：传统的 alloc/init 方式已不再使用，取而代之的是 initWithConcurrencyType:方法。在 iOS 8 版本中，Core Data 的线程约束方法(开发者需要对 Core Data 操作发生的线程进行管理和负责)已不再使用。相反，数据量比较小的基础应用使用主线程并发类型进行处理，这需要在一个代码块中执行 Core Data 操作。这个方法确保所有 Core Data 访问都在正确的线程中执行，避免了潜在的线程和调试错误。对于更复杂的情况，例如数据更新在后台线程执行等问题，可以创建更高级的带有父子关系概念的 Core Data 栈机制来处理后台操作，并实现与主队列托管对象上下文的有效合并。

托管对象上下文的访问器方法将会检查托管对象上下文实例变量是否有引用。如果没有，将会获取一个持久化存储协调器并用它实例化一个新的托管对象上下文，将新的实例赋给实例变量，并返回该实例变量。

```
-(NSManagedObjectContext *)managedObjectContext
{
  if(__managedObjectContext != nil)
  {
    return __managedObjectContext;
  }

  NSPersistentStoreCoordinator *coordinator =
  ➥[self persistentStoreCoordinator];

  if(coordinator != nil)
  {
    __managedObjectContext = [[NSManagedObjectContext alloc]
    ➥initWithConcurrencyType:NSMainQueueConcurrencyType];

    [__managedObjectContext
```

```
    ➥setPersistentStoreCoordinator:coordinator];
  }
  return __managedObjectContext;
}
```

持久性存储协调器是 Core Data 用来管理持久性存储(或文件)的类。要初始化它，需要一个 NSManagedObjectModel 实例，这样持久性存储协调器就知道需要处理的对象模型。持久性存储协调器对于每一个添加的对象都需要设置 URL，如果文件不存在，Core Data 会创建它。如果持久性存储与托管对象模型不匹配(Core Data 使用托管对象模型的散列唯一进行标识，目的是使持久性存储中的数据能够进行比较)，之后模板逻辑将会输出错误和异常信息。对于跨版本的应用而言，需要添加逻辑来正确地处理从旧数据模型迁移到新数据模型时出现的错误。在开发过程中，让应用终止是一个有效的方法，可以提醒开发者模型变化导致错误的出现，需要重新安装应用再次进行测试。

```
-(NSPersistentStoreCoordinator *)persistentStoreCoordinator
{
  if(__persistentStoreCoordinator != nil)
  {
    return __persistentStoreCoordinator;
  }

  NSURL *storeURL =
  ➥[[self applicationDocumentsDirectory]
  ➥URLByAppendingPathComponent:@"MyMovies.sqlite"];

  NSError *error = nil;
  __persistentStoreCoordinator =
  ➥[[NSPersistentStoreCoordinator alloc]
  ➥initWithManagedObjectModel:[self managedObjectModel]];

  if(![__persistentStoreCoordinator
     ➥addPersistentStoreWithType:NSSQLiteStoreType
     ➥configuration:nil URL:storeURL options:nil
     ➥error:&error])
  {
    NSLog(@"Unresolved error %@, %@", error,
       [error userInfo]);

    abort();
  }

  return __persistentStoreCoordinator;
}
```

托管对象模型从应用的主 bundle 中加载。Xcode 将会为托管对象模型指定同项目一样的名称。

```
-(NSManagedObjectModel *)managedObjectModel
```

```
{
  if(__managedObjectModel != nil)
  {
    return __managedObjectModel;
  }

  NSURL *modelURL =
  ➥[[NSBundle mainBundle] URLForResource:@"MyMovies"
                        withExtension:@"momd"];

  __managedObjectModel =
  ➥[[NSManagedObjectModel alloc]
   initWithContentsOfURL:modelURL];

  return __managedObjectModel;
}
```

15.4　创建托管对象模型

从 Xcode 项目模板创建一个同名的数据模型文件。示例项目的文件为 Managed Object Model。要编辑数据模型，单击数据模型文件，Xcode 将会呈现数据模型编辑器，如图 15-4 所示。

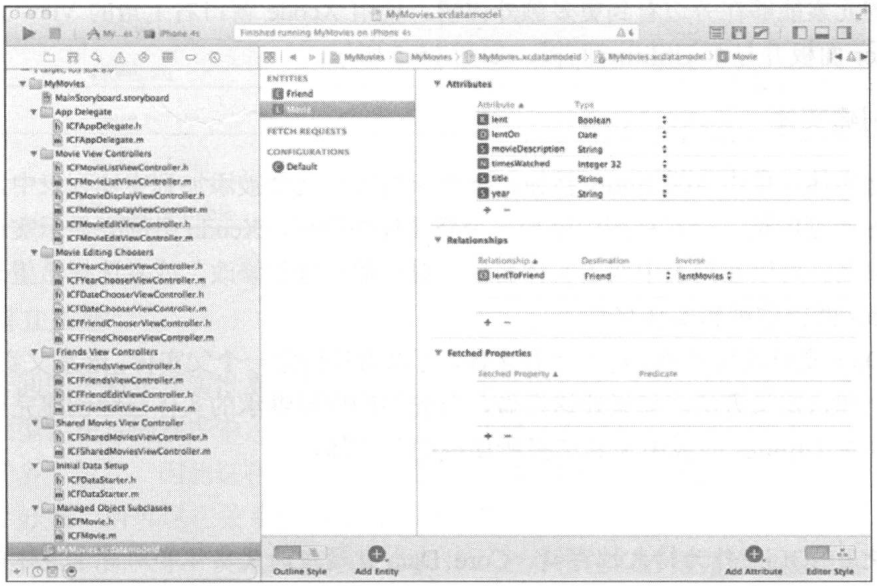


图 15-4　Xcode 数据模型编辑器，Table 类型

Xcode 有两种类型的数据模型编辑器，分别是 Table 和 Graph。Table 类型在左边以列表样式显示数据模型中的实体。选择一个实体就会显示其内容，并可以对该实体的特性、关系和取回的属性进行编辑。

要想切换为 Graph 类型，单击数据模型编辑器右下角的 Editor Style Graph 按钮，如图 15-5 所示。数据模型编辑器左边仍然会列出实体的列表，不过编辑器的主要部分显示的内容是具

体实体数据模型的关系图表。每个矩形图表示一个实体，上面会标注实体的名称，它包含的属性在中间显示，它的关系在下面显示。图形带有实体关系连接的箭头，这些箭头表示关系的基数。

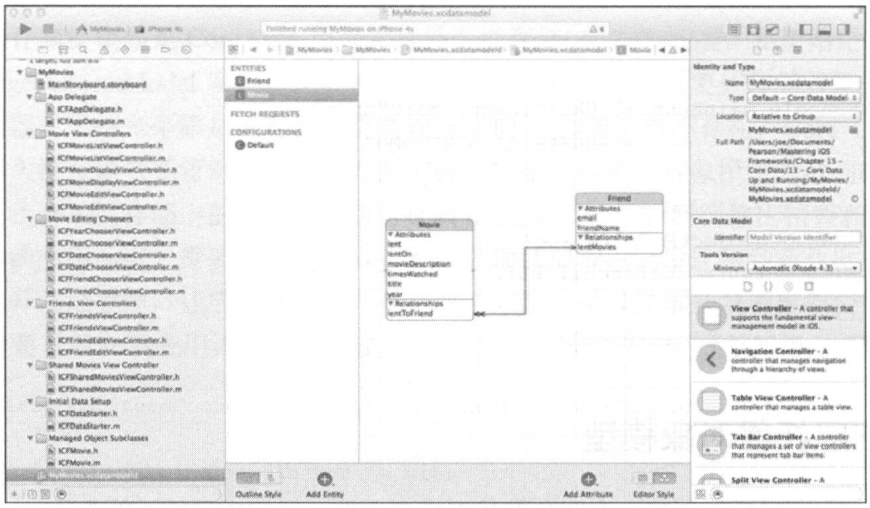


图 15-5 Xcode 数据模型编辑器，Graph 类型

当对数据模型进行处理时，为了方便起见，通常会希望有更大的编辑空间，同时也希望对于选中的元素能够在旁边看到更多额外详情。使用 Xcode 窗口右上角的 View 选项可以隐藏 Navigator 面板并显示 Utilities 面板，如图 15-5 所示。

15.4.1 创建实体

要创建实体，单击 Add Entity 按钮，一个新的实体就会被添加到实体列表中。如果编辑器是 Graph 类型的话，会在视图中添加一个新实体的矩形。Xcode 会高亮显示实体名称并让你为实体取名。可以在列表中双击实体名称，对它随时进行修改，或者选择希望改名的实体并在 Utilities 面板中编辑实体名称。

Core Data 支持实体继承。对于所有实体，可以为其指定一个父实体，继承父实体的特性、关系、验证和自定义方法。如果要这样做，首先要确保要继承的实体已经创建完毕，之后选择子实体并在 Utilities 面板中为其选择所期望的父实体。

注意

如果使用 SQLite 作为持久性存储，Core Data 就通过为父实体和所有子实体创建一个表来实现实体继承关系，并带有一个包含所有特性的超集。如果实体中有大量数据，这样做显然不会影响性能，所以可以在程序开发中广泛使用这个方法。

15.4.2 添加特性

要为实体添加特性，首先在图形模式或实体列表下选择实体。之后单击编辑器下面的 Add Attribute 按钮，该按钮在 EditorStyle 按钮的左边。Xcode 会向实体添加一个名为 attribute 的特性。为特性选择一个 Type(查看表 15-1 给出的数据类型)。注意 Core Data 将所有特性都视

为 Objective-C 对象，所以如果选择的数据类型为 Interger 32，Core Data 会认为该特性类型为 NSNumber。

需要注意的是，Core Data 会自动为每一个实例分配一个唯一的对象 ID，即 objectID，用于内部管理存储和关系。还可以为实体添加一个唯一 ID 或其他合适的键，以及添加索引值用于快速访问，但注意 Core Data 将使用所生成的对象 ID 来管理关系。

NSManagedObject 实例还包含一个名为 description 的方法，如果希望得到 description 特性，只需要稍微修改一下它的名称就可以避免冲突。在示例程序中，Movie 实体就具有 movieDescription 特性。

表 15-1　Core Data 支持的数据类型

数 据 类 型	Objective-C 的存储类型
Integer 16	NSNumber
Integer 32	NSNumber
Integer 64	NSNumber
Decimal	NSNumber
Double	NSNumber
Float	NSNumber
String	NSString
Boolean	NSNumber
Date	NSDate
Binary Data	NSDate
Transformable	使用值变换

15.4.3　建立关系

用对象间的关系对真实世界进行建模是一项重要的技术。Core Data 支持一对一和一对多两种关系模型。在示例程序中，在好友和影片之间建立的就是一对多的关系。由于一个好友可能同时借多个影片，因此这在关系模型中就是“多”的一方，不过一个影片在某个时间只能借给一个好友，所以这在关系模型中就是“一”的一方。

要为实体间添加关系，先选择其中一个实体，之后按住 Ctrl 键并拖曳到目标实体。另一种方法是单击并按住 Add Attribute 按钮，然后从出现的菜单中选择 Add Relationship。Xcode 将会创建一个到目标实体的关联，称为“关系”。在 Utilities 面板上，选择 Data Model 检查器，修改关系的名称。在 Data Model 检查器中，可以进行如下操作：

- 指明关系是否为暂时的。
- 通过 Optional 复选框来确定关系是可选的还是必需的。
- 指定关系是否有序。

- 创建一个相反的关系。要实现这个操作，首先创建并命名一个反向的关系，之后从下拉框中选择它。除了原始的关系之外，反向关系允许之前的目的对象现在变为起点对象。
- 通过选中 Plural 复选框和取消选中 Plural 复选框来指定关系的基数。如果选中此复选框，则表示为一对多关系。
- 指定关系计数的最小值和最大值。
- 当对象被删除时为关系设置 Core Data 遵循的规则。可选项为 No Action(对于删除不进行任何其他动作)、Nullify(设置关系为 nil)、Cascade(关系另一侧的对象也同时删除)和 Deny(如果存在关系，则报错)。

15.4.4 自定义托管对象子类

如果希望对模型对象使用自定义逻辑，那么使用自定义的 NSManagedObject 子类会很有帮助；或者如果希望为模型对象属性使用点语法和使用编译器验证，也可以使用这个子类。

Xcode 有一个菜单选项可以自动创建子类。要使用它，确保在数据模型编辑器中已完成实体的设置。在数据模型编辑器中选择一个(或多个)实体，从 Xcode 菜单中选择 Editor，之后选择 Create NSManagedObject Subclass 选项。Xcode 会问你希望将生成的类文件保存在什么位置。指定一个位置并单击 Create，Xcode 就会为每个指定的实体生成头文件和实现代码。Xcode 还会根据项目的名称为每个类命名一个带有指定类前缀的名称。

在生成的头文件中，Xcode 将会为每个实体中的特性创建属性参数。注意，Xcode 还会为指定实体的每个关系创建属性参数。如果关系为一对一，Xcode 将会创建一个 NSManagedObject 属性(或者如果目标实体是一个自定义子类，则使用 NSManagedObject 子类)。如果关系为一对多，Xcode 会创建一个 NSSet 属性。

在生成的实现文件中，Xcode 会为每个实体创建@dynamic 而不是@synthesize 声明。这是因为对于 Core Data 托管特性，Core Data 会动态地处理访问器，不需要编译器创建访问器方法。

注意

mogenerator 项目将会为每个实体生成两个类：一个是针对特性访问器的类，另一个是针对自定义逻辑的类。这个方法使你在修改数据模型时很容易重新生成类，而不需要重写自定义逻辑。可以从 http://rentzsch.github.com/mogenerator/下载 mogenerator 项目。

15.5 设置默认数据

当 Core Data 项目第一次设置时，其中还没有数据。虽然在有些场景下这样也没有问题，但通常应用在第一次运行时还是需要生成一些数据的。在示例程序中有一个自定义数据设置类 ICFDataStarter，用于为 Core Data 生成一些初始数据。MyMovies-Prefix.pch 中用#define 定义了一个变量 FIRSTRUN，可以取消注释让应用在 ICFDataStarter 中运行相应的逻辑代码。

15.5.1　插入新的托管对象

要为还未在模型中的数据创建一个新的托管对象实例，需要具有对托管对象上下文的引用。对于 ICFDataStarter，示例程序从 ICFAppDelegate 将托管对象上下文属性传递到 setupStarterDataWithMOC:方法：

```
[ICFDataStarter setupStarterDataWithMOC:[self managedObjectContext]];
```

为了避免线程错误，所有非 Core Data 对象都要在托管对象上下文指定的代码块内使用。有两个方法可以选择，分别是 performBlock:和 performBlockAndWait:。第一个方法会将代码块异步提交给托管对象上下文的队列，并在当前作用域内继续执行代码。第二个方法会将代码块异步提交给托管对象上下文的队列，并直到所有操作都完成后才可以在当前作用域内继续执行代码。如果没有和当前作用域的依赖关系，使用 performBlock:方法可以避免死锁和等待；如果有依赖关系，则使用 performBlockAndWait:。这种情况下，因为托管对象上下文是在主队列中的，并且代码也在主队列执行，所以使用 performBlockAndWait:可以确保所有任务按序执行，从而避免由于潜在的执行顺序问题而导致的错误。

```
[moc performBlockAndWait:^{
```

要插入数据，Core Data 需要知道新数据是用于哪个实体的。Core Data 具有一个名为 NSEntityDescription 的类，用于提供有关实体的信息。使用 NSEntityDescription 的类方法创建一个新实例：

```
NSManagedObject *newMovie1 =
➥[NSEntityDescription insertNewObjectForEntityForName:@"Movie"
                        inManagedObjectContext:moc];
```

实例创建完毕后，使用相应的数据填充特性：

```
[newMovie1 setValue:@"The Matrix" forKey:@"title"];
[newMovie1 setValue:@"1999" forKey:@"year"];

[newMovie1 setValue:@"Take the blue pill."
           forKey:@"movieDescription"];

[newMovie1 setValue:@NO forKey:@"lent"];
[newMovie1 setValue:nil forKey:@"lentOn"];
[newMovie1 setValue:@20 forKey:@"timesWatched"];
```

Core Data 使用键-值编码处理设置特性。如果特性名不正确，在运行时就会出错。要对特性的赋值情况进行检查，需要创建一个 NSManagedObject 子类，并对每个特性直接使用特性访问器。

托管对象上下文相当于修改特性的工作区域，所以示例程序设置了更多的初始数据：

```
NSManagedObject *newFriend1 =
➥[NSEntityDescription insertNewObjectForEntityForName:@"Friend"
                        inManagedObjectContext:moc];
```

```
[newFriend1 setValue:@"Joe" forKey:@"friendName"];
[newFriend1 setValue:@"joe@dragonforged.com" forKey:@"email"];
```

设置好所有的初始数据后，最后一步就是保存托管对象上下文，同时关闭代码块。

```
    NSError *mocSaveError = nil;

    if([moc save:&mocSaveError])
    {
        NSLog(@"Save completed successfully.");
    } else
    {
        NSLog(@"Save did not complete successfully.Error: %@",
        ➥[mocSaveError localizedDescription]);
    }
}];
```

保存好托管对象上下文后，Core Data 将会在数据存储中保留数据。对于应用实例来说，即使程序关闭再重启数据也同样存在。如果应用从模拟器或设备上已被卸载，数据就不存在了。对于应用在首次运行时填充数据，还有另一个方法就是从应用存储目录中将数据存储复制到应用 bundle。这样可以保证首次运行时默认数据集被复制到应用目录中，应用可以使用这些数据。

15.5.2 其他默认的数据设置方法

还有两个常用的默认数据设置方法，分别是数据模型版本迁移和从 Web 服务或 API 加载数据。

Core Data 托管对象模型都是版本控制的。Core Data 知道托管对象模型和当前数据存储之间的关系。如果托管对象模型发生变化并且不再同数据存储兼容(例如一个实体添加了一个特性)，Core Data 就不能使用已经存在的数据存储和新的托管对象模型对持久存储对象进行初始化。这种情况下，需要使用迁移来更新现存的数据存储以匹配更新后的托管数据模型。大多数情况下，Core Data 可以通过在持久化存储对象初始化时传递一个选项字典来自动执行迁移；有些情况下，执行迁移还需要额外的一些步骤。迁移不在本章的讨论范围内，不过要记住，迁移是苹果公司推荐的用于初始数据设置的方法。

另一个方法是从 Web 服务或 API 获取数据。当应用需要将本地数据子集复制到 Web 服务器上时这个方法很适用，在正常的操作中需要为应用编写 Web 调用方法，实现从 API 获取数据。要设置应用的初始数据，可以一种特殊的状态来运行 Web 调用，以获取所有需要的初始数据并将其保存在 Core Data 中。

15.6 显示托管对象

要显示或使用应用中存在的实体数据，需要从托管对象上下文中取回托管对象。取回的动作类似于关系数据库中的查询操作，可以指定想要取回哪个实体，设置匹配规则以及如何保存结果。

15.6.1　创建取回请求

Core Data 中用于取回托管对象的是一个名为 NSFetchRequest 的对象。查看示例程序的 ICFFriendChooserViewController 类。这个视图控制器显示 Core Data 中设置的好友信息，用户可以选择好友，借给他们影片，如图 15-6 所示。

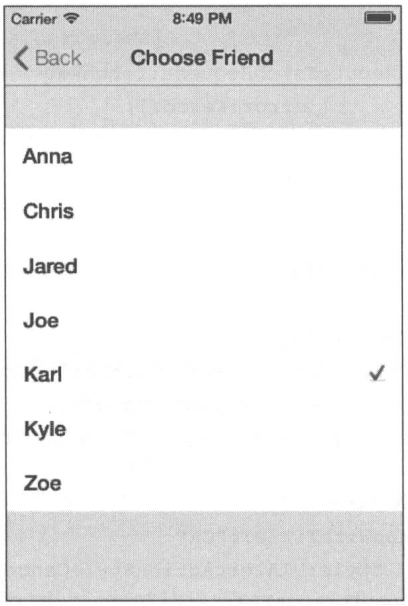


图 15-6　示例程序：好友选择界面

要得到这个好友列表并显示它，当视图控制器加载时要执行一个标准的取回请求。第一步是创建一个 NSFetchRequest 实例并将实体和取回请求建立关联：

```
NSManagedObjectContext *moc = kAppDelegate.managedObjectContext;

[moc performBlockAndWait:^{
    NSFetchRequest *fetchReq = [[NSFetchRequest alloc] init];

    NSEntityDescription *entity =
    ➥[NSEntityDescription entityForName:@"Friend"
                inManagedObjectContext:moc];

    [fetchReq setEntity:entity];
```

接下来要告诉取回请求如何对得到的托管对象进行排序。为此，需要为取回请求关联一个排序描述器，指定按特性名排序：

```
NSSortDescriptor *sortDescriptor =
➥[[NSSortDescriptor alloc] initWithKey:@"friendName"
                              ascending:YES];

NSArray *sortDescriptors = @[sortDescriptor];
```

```
[fetchReq setSortDescriptors:sortDescriptors];
```

由于好友选择器需要显示所有可用的好友，因此不需要特别指定匹配规则。剩下的事情就是执行取回操作了：

```
NSError *error = nil;

self.friendList = [moc executeFetchRequest:fetchReq
                                      error:&error];

 if(error)
{
  NSString *errorDesc =
  ➥[error localizedDescription];

    UIAlertController *alertController =
    ➥[UIAlertController alertControllerWithTitle:@"Error fetching friends"
                                          message:errorDesc
                                   preferredStyle:UIAlertControllerStyleAlert];

    [alertController addAction:
    ➥[UIAlertAction actionWithTitle:@"OK"
                               style:UIAlertActionStyleCancel
                             handler:nil]];

    [self presentViewController:alertController
                       animated:YES
                     completion:nil];
}
```

要执行一条取回命令，需要创建一个 NSError 实例并设置它为 nil。之后让托管对象上下文执行刚刚创建好的取回请求。如果遇到错误，托管对象上下文就会将这个错误返回给刚刚创建的实例。示例程序会用 UIAlertController 实例显示错误。如果没有错误，将会返回一个由 NSManagedObjects 组成的 NSArray 数组作为结果。视图控制器将会在一个实例变量中保存表视图中显示的这些结果。

15.6.2 根据对象 ID 取回托管对象

当只有一个特定的托管对象需要取回时，Core Data 给出了一种方法，不需要创建取回请求就可以快速获取托管对象。要使用这个方法，托管对象必须有 NSManagedObjectID。

要得到托管对象的 NSManagedObjectID，必须已经得到取回或创建好的托管对象。参考示例程序中的 ICFMovieListViewController 类的 prepareForSegue:sender:方法。这种情况下，用户从列表中选择一个影片，视图控制器将程序从列表界面切换到选择影片的 detail 视图。要通知 detail 视图控制器用户点击的是哪个影片，在 ICFMovieDisplayViewController 中需要将选中影片的 objectID 作为属性。

```
if([[segue identifier] isEqualToString:@"showDetail"])
{

  NSIndexPath *indexPath =
  ➥[self.tableView indexPathForSelectedRow];

  ICFMovie *movie =
  ➥[[self fetchedResultsController]
  ➥objectAtIndexPath:indexPath];

  ICFMovieDisplayViewController *movieDispVC =
  ➥(ICFMovieDisplayViewController *)
  ➥[segue destinationViewController];

  [movieDispVC setMovieDetailID:[movie objectID]];
}
```

当加载 ICFMovieDisplayViewController 时，它会根据 objectID 对托管对象上下文使用一个加载托管对象的方法：

```
[kAppDelegate.managedObjectContext performBlockAndWait:^{
    ICFMovie *movie = (ICFMovie *)[kAppDelegate.managedObjectContext
    ➥objectWithID:self.movieDetailID];

    [self configureViewForMovie:movie];
}];
```

当加载时，影片视图控制器可以使用影片数据来配置视图，如图 15-7 所示。

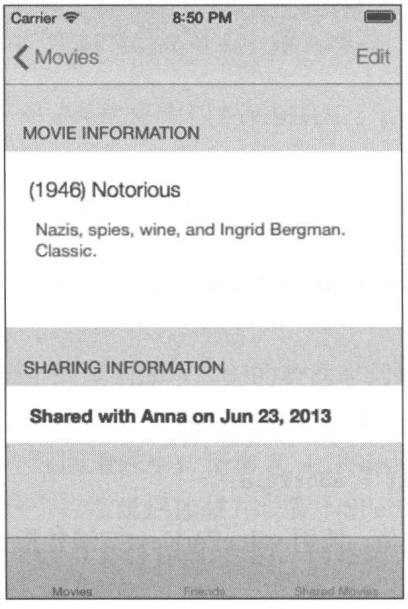


图 15-7　示例程序：影片显示视图

将一个托管对象从一个视图控制器传递到下一个视图控制器是完全有可能的，这样就不

需要传递 objectID，也不需要在目标视图控制器中加载托管对象了。不过有些时候使用 objectID 要比使用托管对象更好，例如下面的情况：

- 如果已在不同的线程上取回或创建托管对象，而不是使用目标视图控制器来处理并显示托管对象，就必须使用 objectID，因为托管对象不是线程安全的。
- 如果后台线程可能在其他托管对象上下文中更新托管对象的取回和显示状态，在显示最新变化时就可以避免可能的错误。

15.6.3 显示对象数据

取回托管对象后，数据的访问和显示都很简单。对于每个托管对象，使用键-值方法来获取特性值。示例程序的 ICFFriendsViewController 类有一个 configureCell:atIndexPath 方法，可以将它作为参考示例。该例中的代码会生成表单元格的标签和详情文本标签。

```
NSManagedObject *object =
➥[self.fetchedResultsController objectAtIndexPath:indexPath];

cell.textLabel.text = [object valueForKey:@"friendName"];

NSInteger numShares = [[object valueForKey:@"lentMovies"] count];

NSString *subtitle = @"";

switch(numShares)
{
  case 0:
    subtitle = @"Not borrowing any movies.";
    break;

  case 1:
    subtitle = @"Borrowing 1 movie.";
    break;

  default:
    subtitle =
    ➥[NSString stringWithFormat:@"Borrowing %d movies.",
    ➥numShares];

    break;
}

cell.detailTextLabel.text = subtitle;
```

要从托管对象中得到特性值，调用 valueForKey:方法并指定特性名。如果特性名不正确，应用会在运行时出现错误。

对于托管对象子类，还可以通过在托管对象子类中调用 property 参数，并指定特性名来访问特性值。请参考示例程序中 ICFMovieDisplayViewController 类的 configureViewForMovie:方法。

```
-(void)configureViewForMovie:(ICFMovie *)movie

{
  NSString *movieTitleYear = [movie yearAndTitle];

  [self.movieTitleAndYearLabel
  ➥setText:movieTitleYear];

  [self.movieDescription setText:[movie movieDescription]];

  BOOL movieLent = [[movie lent] boolValue];

  NSString *movieShared = @"Not Shared";
  if(movieLent)
  {
    NSManagedObject *friend =
    ➥[movie valueForKey:@"lentToFriend"];

    NSDateFormatter *dateFormatter =
    ➥[[NSDateFormatter alloc] init];

    [dateFormatter setDateStyle:NSDateFormatterMediumStyle];

    NSString *sharedDateTxt =
    ➥[dateFormatter stringFromDate:[movie lentOn]];

    movieShared =
    ➥[NSString stringWithFormat:@"Shared with %@ on %@",
    ➥[friend valueForKey:@"friendName"],sharedDateTxt];
  }

  [self.movieSharedInfoLabel setText:movieShared];
}
```

如果使用属性的参数方法从使用的托管对象子类中获取特性值，在编译时可能会出现错误。

15.6.4　使用谓词

使用谓词可以让程序根据特定的规则缩小匹配的数据取回结果。它会模拟 SQL 语言的句法，谓词还可以用于从集合(例如 NSArray)中提取元素，就像对 Core Data 使用一条取回请求一样。要了解如何将谓词应用在取回请求中，请参考 ICFSharedMoviesViewController 类的 fetchedResultsController 方法。这个方法延迟加载并设置 NSFetchedResultsController，可以帮助表视图与取回结果进行交互(下一节会有详细描述)。构建谓词很容易，例如：

```
NSPredicate *predicate =
➥[NSPredicate predicateWithFormat:@"lent == %@",@YES];
```

在格式字符串中，谓词的构建可以使用特性名、比较运算符、布尔运算符、聚合运算符

和置换表达式。点分隔法表示的表达式列表将会被使用置换表达式的格式字符串所取代。点表示法可以在谓词格式字符串中用于指定关系。谓词支持多种运算符和参数，如表 15-2 所示。

表 15-2 Core Data 谓词支持的运算符和参数

类　型	运算符和参数
基础比较	=、==、>=、=>、<=、=<、>、<、!=、<>、BETWEEN {low,high}
布尔型	AND、&& OR、|| NOT、!
字符串	BEGINSWITH、CONTAINS、ENDSWITH、LIKE、MATCHES
聚合类型	ANY、SOME、ALL、NONE、IN
字面量	FALSE、NO、TRUE、YES、NULL、NIL、SELF，Core Data 还支持字符串和数值字面量

通知取回请求使用谓词的方法如下：

```
[fetchRequest setPredicate:predicate];
```

现在取回请求会根据谓词中指定的标准缩小返回结果，只返回符合匹配要求的托管对象集，如图 15-8 所示。

图 15-8 示例程序：Shared Movies 标签页

15.7 取回结果控制器介绍

取回结果控制器(NSFetchedResultsController)是 Core Data 和 UITableView 或 UICollectionView 之间一个非常有效的连接者。取回结果控制器提供了一种设置取回请求的方法，可以让结果在视图分类界面和单元行中返回，并使用索引路径对其加以访问。此外，取回结果控制器可以监听 Core Data 的变化并根据委托方法中的逻辑更新表视图。

在示例程序中，请参考 ICFMovieListViewController 类，了解在实际运行中取回结果控制器的详细使用示例，如图 15-9 所示。

图 15-9　示例程序：影片列表的视图控制器

15.7.1　准备取回结果控制器

当使用 Xcode 的 Master Detail 模板设置“master”视图控制器时，Xcode 会为取回结果控制器创建一个属性，并覆盖访问器方法(fetchedResultsController)使其延时加载，或者在取回结果控制器第一次被请求时初始化它。首先该方法会检查取回结果控制器是否已被初始化：

```
if(__fetchedResultsController != nil)
{
  return __fetchedResultsController;
}
```

如果取回结果控制器已经设置好，则直接返回结果。否则需要设置一个新的取回结果控制器，先从一个取回请求开始：

```
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
```

取回请求需要同托管对象模型中的实体进行关联，托管对象上下文如下：

```
NSManagedObjectContext *moc = kAppDelegate.managedObjectContext;

NSEntityDescription *entity =
➥[NSEntityDescription entityForName:@"Movie"
           inManagedObjectContext:moc];

[fetchRequest setEntity:entity];
```

可以设置批量处理的值以避免取回操作一次得到太多的记录：

```
[fetchRequest setFetchBatchSize:20];
```

接下来，使用 NSSortDescriptor 实例为取回请求创建排列顺序。需要重点注意的一点是，用于表示分节的特性应该是序列中的第一个，这样就可以保证记录会正确地分到各自的部分。排序是由取回请求中的一个保存排序描述的数组确定的。

```
NSSortDescriptor *sortDescriptor =
➥[[NSSortDescriptor alloc] initWithKey:@"title" ascending:YES];

NSSortDescriptor *sharedSortDescriptor =
➥[[NSSortDescriptor alloc] initWithKey:@"lent" ascending:NO];

NSArray *sortDescriptors = @[sharedSortDescriptor,sortDescriptor];

[fetchRequest setSortDescriptors:sortDescriptors];
```

取回请求准备完毕后，就可以初始化取回结果控制器了。初始化的过程需要一个取回请求、一个托管对象上下文、一个用于表视图分节的键路径或特性名以及一个缓存名称(如果传递 nil，则不使用数据缓存)。取回结果控制器可以指定一个委托来对所有 Core Data 变化进行响应。完成初始化之后，取回结果控制器会被赋给视图控制器的属性：

```
NSFetchedResultsController *aFetchedResultsController =
➥[[NSFetchedResultsController alloc]
➥initWithFetchRequest:fetchRequest
➥managedObjectContext:moc
➥sectionNameKeyPath:@"lent"
➥cacheName:nil];

aFetchedResultsController.delegate = self;
self.fetchedResultsController = aFetchedResultsController;
```

现在取回结果控制器已经就绪，它可以执行取回操作，获得表视图中显示的结果集了，取回结果控制器可以返回到调用函数：

```
NSError *error = nil;
if(![self.fetchedResultsController performFetch:&error])
{
  NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
  abort();
}

return __fetchedResultsController;
```

15.7.2 整合表视图和取回结果控制器

整合表视图和取回结果控制器只是一种更新表视图数据源的方法，让委托方法使用取回结果控制器中的信息。在 ICFMovieListViewController 类中，取回结果控制器会告诉表视图需要分成几个分节：

```
-(NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
  return [[self.fetchedResultsController sections] count];
}
```

取回结果控制器告诉表视图每个分节包含多少行，这要用到 NSFetchedResultsSectionInfo 协议：

```
-(NSInteger)tableView:(UITableView *)tableView
➥numberOfRowsInSection:(NSInteger)section
{
  id <NSFetchedResultsSectionInfo> sectionInfo =
  ➥[[self.fetchedResultsController sections]
  ➥objectAtIndex:section];

  return [sectionInfo numberOfObjects];
}
```

取回结果控制器提供分节的标题，即指定了分节名称的特性值。由于示例程序中为分节使用的是布尔类型特性，即 0 和 1，因此这种类型的值不便于用户理解。示例程序从取回结果控制器查看标题并返回一个更有助于理解的标题：用 Shared 代替 1，用 Not Shared 代替 0。

```
-(NSString *)tableView:(UITableView *)tableView
➥titleForHeaderInSection:(NSInteger)section
{
  id <NSFetchedResultsSectionInfo> sectionInfo =
   ➥[[self.fetchedResultsController sections]
   ➥objectAtIndex:section];

  if([[sectionInfo indexTitle] isEqualToString:@"1"])
  {
    return @"Shared";
  }
  else
  {
    return @"Not Shared";
  }
}
```

要向表单元格中填充内容，示例程序将一个可重用的单元格移出队列，之后调用 configureView:方法，传递单元格的 indexPath 参数：

```
-(UITableViewCell *)tableView:(UITableView *)tableView
➥cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell =
  ➥[tableView dequeueReusableCellWithIdentifier:@"Cell"];

  [self configureCell:cell atIndexPath:indexPath];
```

```
    return cell;
}
```

取回结果控制器知道对应每个索引路径的影片，所以示例程序可以在取回结果控制器上调用 objectAtIndexPath:方法来获得正确的影片信息进行显示。之后使用影片实例中的数据就很容易更新单元的内容了。

```
-(void)configureCell:(UITableViewCell *)cell
➥atIndexPath:(NSIndexPath *)indexPath
{
  ICFMovie *movie =
  ➥[self.fetchedResultsController objectAtIndexPath:indexPath];

  cell.textLabel.text = [movie cellTitle];

  cell.detailTextLabel.text = [movie movieDescription];
}
```

最后需要整合到表视图中的详细信息是在 tableView:didSelectRowAtIndexPath:方法中对单元格进行的选中操作。这时方法内就不再有整合操作了，因为此时选中单元格开始由 storyboard 的切换机制来处理了。在 prepareForSegue:sender:方法中，使用 showDetail 标识符来处理表格单元格的选中：

```
if([[segue identifier] isEqualToString:@"showDetail"])
{
NSIndexPath *indexPath =
 ➥[self.tableView indexPathForSelectedRow];

    ICFMovie *movie =
    ➥[[self fetchedResultsController]
    ➥objectAtIndexPath:indexPath];

    ICFMovieDisplayViewController *movieDisplayVC =
    ➥(ICFMovieDisplayViewController *)
    ➥[segue destinationViewController];

    [movieDisplayVC setMovieDetailID:[movie objectID]];
}
```

这个方法从表视图获取选中行的索引路径，之后使用这个索引值从取回结果控制器获取影片实例。该方法之后会使用影片实例的 objectID 来设置 ICFMovieDisplayViewController 实例的 movieDetailID 参数。

15.7.3 对 Core Data 变化的响应

为了实现取回结果控制器对 Core Data 变化的响应和更新表视图，需要实现 NSFetchedResultsControllerDelegate 协议的一些方法。首先，视图控制器需要声明它将要实现委托方法：

```
@interface ICFMovieListViewController:UITableViewController
➥<NSFetchedResultsControllerDelegate>
```

当有内容发生变化时取回结果控制器委托将会得到通知，委托函数可以在表视图中更新这些变化。在表视图上调用 beginUpdates 方法通知所有的更新同步执行，直到 endUpdates 被调用时才会停止。

```
-(void)controllerWillChangeContent:
➥(NSFetchedResultsController *)controller
{
  [self.tableView beginUpdates];
}
```

基于数据改变可能会调用两个委托方法。第一个方法将会通知委托数据变化的发生对表视图的分节产生了影响；第二个方法将会通知委托数据的变化影响了与指定索引路径对应的对象，表视图需要更新相应的行。因为数据变化由变化类型来表示，所以如果变化是插入、删除、移动或更新等操作，则通知委托函数。典型的方法是创建一条 switch 语句，根据变化类型执行正确的动作。对于视图分节，示例程序只有在插入或删除分节时才会发生变化(如果分节名发生变化，可能导致分节移动和更新)。

```
-(void)controller:(NSFetchedResultsController *)controller
➥didChangeSection:(id <NSFetchedResultsSectionInfo>)sectionInfo
➥atIndex:(NSUInteger)sectionIndex
➥forChangeType:(NSFetchedResultsChangeType)type
{
  switch(type)
  {
    case NSFetchedResultsChangeInsert:
      ...
      break;

    case NSFetchedResultsChangeDelete:
      ...
      break;
  }
}
```

表视图有一个便捷的方法用于插入新的分节，委托方法会收到插入新分节所用到的全部信息：

```
[self.tableView
➥insertSections:[NSIndexSet indexSetWithIndex:sectionIndex]
➥withRowAnimation:UITableViewRowAnimationFade];
```

删除分节的实现也很简单：

```
[self.tableView
➥deleteSections:[NSIndexSet indexSetWithIndex:sectionIndex]
➥withRowAnimation:UITableViewRowAnimationFade];
```

当对象发生变化时，变化类型、发生变化的对象、对象当前的索引路径都会通知给委托。如果对象是插入或移动的情况，则会将一个“新”的索引路径传递给委托。使用 switch 逻辑根据变化类型对变化进行响应。

```
-(void)controller:(NSFetchedResultsController *)controller
➥didChangeObject:(id)anObject
➥atIndexPath:(NSIndexPath *)indexPath
➥forChangeType:(NSFetchedResultsChangeType)type
➥newIndexPath:(NSIndexPath *)newIndexPath
{
  UITableView *tableView = self.tableView;

  switch(type)
  {
    case NSFetchedResultsChangeInsert:
      ...
      break;

    case NSFetchedResultsChangeDelete:
      ...
      break;

    case NSFetchedResultsChangeUpdate:
      ...
      break;

    case NSFetchedResultsChangeMove:
      ...
      break;
  }
}
```

表视图使用便捷方法根据索引路径插入具体的行。注意，当对一个插入对象实际插入一行数据时，所使用的正确路径是 newIndexPath 的值。

```
[tableView
➥insertRowsAtIndexPaths:[NSArray arrayWithObject:newIndexPath]
➥withRowAnimation:UITableViewRowAnimationFade];
```

要删除行，使用传递给委托方法的 indexPath 参数。

```
[tableView
➥deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
➥withRowAnimation:UITableViewRowAnimationFade];
```

要更新行，为当前 indexPath 调用 configureCell:atIndexPath:方法。这一配置方法的调用同表视图委托的 tableView:cellForRowAtIndexPath:方法一样。

```
[self configureCell:[tableView cellForRowAtIndexPath:indexPath]
➥atIndexPath:indexPath];
```

要移动行，删除当前 indexPath 对应的行，用 newIndexPath 插入一个新行。

```
[tableView
➥deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
➥withRowAnimation:UITableViewRowAnimationFade];

[tableView
➥insertRowsAtIndexPaths:[NSArray arrayWithObject:newIndexPath]
➥withRowAnimation:UITableViewRowAnimationFade];
```

当内容的修改完成时，会通知取回结果控制器委托，所以委托可以通过调用 endUpdates 方法通知表视图所有的修改都已完成。调用 endUpdates 方法后，表视图会在用户界面中更新所有这些修改内容。

```
-(void)controllerDidChangeContent:
➥ (NSFetchedResultsController *)controller
{
  [self.tableView endUpdates];
}
```

15.8　添加、编辑和删除托管对象

虽然能够取回并显示数据是很实用的功能，不过用户还经常需要添加新数据、编辑已有数据和删除不需要的数据。

15.8.1　插入新的托管对象

在示例程序中查看 Movies 标签页。要插入一个新的影片，用户可以单击导航栏中的 Add 按钮。为 Add 按钮设置关联，切换到 ICFMovieEditViewController 类。在切换逻辑中，一个新的影片托管对象被插入到 Core Data 中，并且这个新的影片对象 ID 也会被传递给编辑影片的视图控制器。示例程序在编辑视图控制器中使用的方法不希望同时处理创建新托管对象和编辑已有托管对象；不过如果有些应用想这么做，在编辑视图控制器中创建新的影片托管对象也是可行的。

要创建一个新的影片托管对象实例，需要一个对托管对象上下文的引用。

```
NSManagedObjectContext *moc =
➥[kAppDelegate managedObjectContext];
```

设置一个变量，用于获取在存储空间中新影片的托管对象 ID，这样该参数就可以用在切换逻辑中了。

```
_block NSManagedObjectID *newMovieID = nil ;
```

再次使用 performBlockAndWait:方法将核心数据的变化从托管对象上下文线程中隔离出来。

```
[moc performBlockAndWait:^{
```

要插入数据，Core Data 需要知道新数据对应的实体。Core Data 有一个名为 NSEntityDescription 的类，用于提供关于实体的信息。使用 NSEntityDescription 的类方法创建一个新的实例：

```
ICFMovie *newMovie = [NSEntityDescription
➥insertNewObjectForEntityForName:@"Movie"
➥inManagedObjectContext:moc];
```

用得到的数据填充新影片托管对象的特性：

```
[newMovie setTitle:@"New Movie"];
[newMovie setYear:@"2014"];
[newMovie setMovieDescription:@"New movie description."];
[newMovie setLent:@NO];
[newMovie setLentOn:nil];
[newMovie setTimesWatched:@0];
```

准备一个 NSError 变量，用于捕获所有潜在的错误，保存托管对象上下文并关闭执行代码块。

```
NSError *mocSaveError = nil;

if(![moc save:&mocSaveError])
{
  NSLog(@"Save did not complete successfully. Error: %@",
  ➥[mocSaveError localizedDescription]);
}
```

成功保存托管对象上下文后，如果保存影响到控制器的取回结果，就会通知取回结果控制器，同时会调用本章前面介绍的委托方法。

15.8.2 删除托管对象

在示例程序的 Movies 标签页上，用户可以在单元格的右边滑动，或者单击 Edit 按钮来显示删除的控件。当在单元格内单击 Delete 按钮时，会调用表视图委托方法 tableView:commitEditingStyle:forRowAtIndexPath:。这个方法会检查 editingStyle 是否等于 UITableViewCellEditingStyleDelete。如果是，则表示用户对单元格单击了 Delete 按钮，所以该方法将准备删除相应的影片，从取回结果控制器中获取关于托管对象上下文的引用。取回结果控制器要始终保存对最初托管对象上下文的引用，因为在删除对象时会用到该引用。

```
NSManagedObjectContext *context =
➥[self.fetchedResultsController managedObjectContext];
```

取回结果控制器根据指定的索引路径找到需要删除的托管对象。

```
[context performBlockAndWait:^{
➥NSManagedObject *objectToBeDeleted =
➥[self.fetchedResultsController objectAtIndexPath:indexPath];
```

要删除托管对象，方法会通知托管对象上下文进行删除。

```
[context deleteObject:objectToBeDeleted];
```

删除动作在托管对象上下文被保存时执行。保存之后，将会调用本章前面介绍的委托方法并更新表。

```
NSError *error = nil;
if(![context save:&error])
{
  NSLog(@"Error deleting movie, %@", [error userInfo]);
}
```

15.8.3　编辑现有的托管对象

在示例程序的 Movies 标签页上，用户可以点击影片查看它的详细信息。要修改任何关于影片的信息，单击导航栏上的 Edit 按钮，将会显示一个 ICFMovieEditViewController 实例。当视图加载时，将会使用从显示视图或列表视图传入的 objectID 加载一个 ICFMovie 实例，然后将该实例保存到 editMovie 属性中，并使用从影片托管对象得到的信息对视图进行配置。

例如，如果用户决定编辑影片的年份，首先会为用户显示一个 UIPickerView 视图控制器让用户选择年份。设置 ICFMovieEditViewController 作为年份选择的委托方法，当用户选择好年份并单击“保存”时，调用委托方法 chooserSelectedYear:。在这个方法中，使用新选择的日期更新 editMovie 并对显示的内容进行更新。

```
-(void)chooserSelectedYear:(NSString *)year
{
    [self.editMovie setYear:year];
    [self.movieYearLabel setText:year];
}
```

注意，托管对象上下文在 editMovie 更新后没有保存。在用户确定修改完成前，可以随时更新托管对象 editMovie，确定完成修改动作的标志就是单击 Save 按钮或 Cancel 按钮。

15.8.4　保存和回滚修改

如果用户单击 Save 按钮，他的意思就是将修改好的内容保存到 editMovie 中。在 saveButtonTouched:方法中，没有通过委托方法更新的字段会被保存到 editMovie 属性中：

```
[kAppDelegate.managedObjectContext performBlockAndWait:^{
    NSString *movieTitle = [self.movieTitle text];
    [self.editMovie setTitle:movieTitle];

    NSString *movieDesc = [self.movieDescription text];
    [self.editMovie setMovieDescription:movieDesc];

    BOOL sharedBool = [self.sharedSwitch isOn];
    NSNumber *shared = [NSNumber numberWithBool:sharedBool];
    [self.editMovie setLent:shared];
```

之后保存托管对象上下文，修改的内容已生效。

```
NSError *saveError = nil;
[kAppDelegate.managedObjectContext save:&saveError];
if(saveError)
{
    UIAlertController *alertController =
    ➥[UIAlertController alertControllerWithTitle:@"Error saving movie"
    ➥message:[saveError localizedDescription]
    ➥preferredStyle:UIAlertControllerStyleAlert];

    [alertController addAction:
    ➥[UIAlertAction actionWithTitle:@"OK"
                              style:UIAlertActionStyleCancel
                            handler:nil]];

    [self presentViewController:alertController
                       animated:YES
                     completion:nil];
}
else
{
    NSLog(@"Changes to movie saved.");
}
```

如果用户决定将之前的修改取消，可以单击 Cancel 按钮，程序会调用 cancelButtonTouched:。这个方法首先检查托管对象上下文是否有未保存的修改。如果有，该方法将会命令托管对象上下文回滚或取消未保存的修改。回滚完成后，托管对象上下文将会回到最初的状态，即没有进行任何修改的状态。与其因为放弃修改而导致用户界面更新，不如直接删除视图。

```
if([kAppDelegate.managedObjectContext hasChanges])
{
  [kAppDelegate.managedObjectContext rollback];
  NSLog(@"Rolled back changes.");
}

[self.navigationController.presentingViewController
➥dismissModalViewControllerAnimated:YES];
```

在视图更新的过程中可以随时保存托管对象上下文，不过需要提醒的是，在主队列托管对象上下文中保存大量的修改可能会导致明显的延迟和潜在的对用户界面的影响。所以一般建议保持相对小的修改；如果保存的变化一定要很大(例如从 Web API 接收大量的信息)，可以考虑使用复杂的带有父/子关系的多上下文 Core Data 栈，以避免向主队列写入过多持久化存储数据。

15.9　小结

本章介绍了如何创建一个使用 Core Data 的新项目以及如何设置所有的 Core Data 环境，详细介绍了如何创建一个托管对象模型，包括如何添加新的实体、为实体添加特性、设置实体间的关系，还介绍了 NSManagedObject 子类的重要性以及如何创建它。

本章讲述了如何为项目设置初始数据，并演示了如何插入一个新的托管对象。另外，对其他一些有关初始数据设置的方法也进行了讨论。

接着，本章详细讲解了如何创建一个取回请求来获取保存的托管对象，以及如何使用 objectID 取回单个托管对象，还介绍了如何在应用的用户界面上显示从托管对象获得的数据，同时讲解了如何使用谓词取回与特定标准相匹配的托管对象。

本章介绍了取回结果控制器，它是一个在 UITableView 中整合 Core Data 的强大工具，介绍了如何使用取回结果控制器设置 UITableView，还介绍了如何设置取回结果控制器委托，根据 Core Data 的变化自动更新表视图。

最后本章介绍了如何添加、编辑和删除托管对象，以及如何保存这些修改或回滚不想要的修改。

掌握了上述这些工具后，你现在应该对如何在应用中有效使用 Core Data 打下了良好的基础。

第16章

使用社交框架整合 Twitter 和 Facebook

社交网络已无处不在，用户希望能访问他们的社交媒体账户，无论是在最新的 iOS 游戏中还是在智能冰箱上(三星的一款 RF4289HARS 型号的冰箱就具有这个功能)。在 iOS 5 之前，向应用中添加 Twitter 和 Facebook 功能非常复杂，是一项挑战开发者能力的任务，由其他用户撰写的第三方库没有考虑到平台的不断扩展，经常出现无法编译的错误。从 iOS 5 开始，苹果公司引入了 Social Framework(社交框架)，只需少量的代码就可以为开发者实现直接在应用中整合 Twitter 服务的功能。随着 iOS 6 的推出，苹果公司进一步扩展了社交框架的功能，加入了 Facebook 和新浪微博(中国最火的社交网络之一)的功能。

这个框架的出现不仅让用户所期待的在所有设备上整合社交网络成为可能，这一社交整合还可以给应用开发者带来很多好处。当用户从一个游戏中获得高分时，他会使用 Facebook 分享这个游戏应用，这样就为开发者开拓了应用推广的市场。这个应用不仅被推荐给了一个新客户，还可能会继续推广到潜在的新客户的朋友群中。几乎所有应用都受益于社交媒体的整合，随着 iOS 8 的推出，这一功能的实现变得更加简单。

16.1 示例程序

本章的示例程序叫作 SocialNetworking，如图 16-1 所示。该应用带有一个文本视图，其中包含一个字符计数标签对象和一个用于添加图片的按钮。标题栏上有两个按钮，分别用于访问 Twitter 和 Facebook。示例程序要求在 Facebook 和 Twitter 中发布的内容不超过 140 个字符，实际上 Facebook 支持更多的字符。

单击标题栏上面的任意按钮都会出现 3 个选项：composer、auto-post 和 timeline。选择 composer 选项会转到自带的 SLComposeViewController 类，这是将消息发送到社交网络服务器最简单和最快速的方法。选择 auto-post 将会自动从主屏幕发送文本和选中的图片，不需要用户进行额外的操作，这一步也可以认为是编程发送。timeline 选项将会弹出用户 Twitter 的

timeline 或 Facebook 订阅。示例程序不包含新浪微博功能，虽然相对而言这也很容易实现。

图 16-1 示例程序 SocialNetworking

16.2 用户登录

社交框架为 Facebook 和 Twitter 使用了一个集中登录系统，可以在 Settings.app 中找到相关内容，如图 16-2 所示。

图 16-2 在 iOS 设备上登录社交服务需要用户离开应用并使用 Settings 应用

如果用户当前没有登录 Twitter 或 Facebook 并尝试访问 Twitter 和 Facebook 功能，就会被要求设置一个新的账户，如图 16-3 所示。这个系统只有在使用 SLComposeViewController 时才会正常工作，否则，没有配置账户就会提示拒绝访问的消息。除了会看到无账户消息之外，如果在 Settings.app 中没有正确进行配置，还可能偶尔会看到“Error 6”错误信息，这个错误通常是由于为账户设置了不正确的证书而引起的。

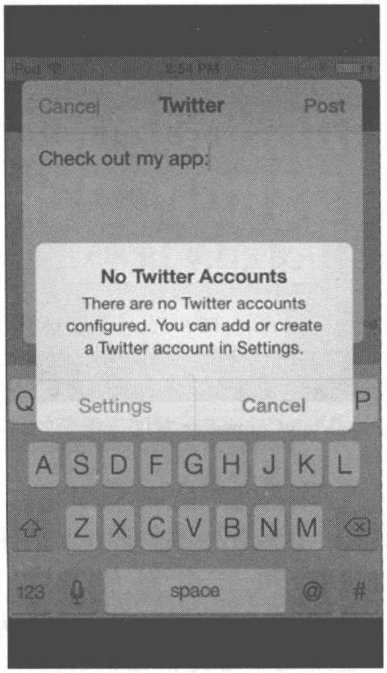


图 16-3　用户被提示需要为设备配置一个 Twitter 账户

注意

目前还没有苹果正式授权的方法用于实现不经过 SLComposeViewController 消息而直接打开 Twitter 和 Facebook 账户配置界面的功能。

16.3　使用 SLComposeViewController

要将新的消息发送到 Twitter 或 Facebook，最简单的方法就是使用 SLComposeViewController。它不需要复杂的认证过程，如果用户没有设置账户，就会提示他进行配置。SLComposeViewController 的缺陷在于无法自定义视图呈现的方式，如图 16-4 所示。

在应用可以与 SLComposeViewController 交互前，必须首先将社交框架导入到项目中。此外，头文件“Social”也需要导入，注意头文件的大小写。

图 16-4 使用 SLComposeViewController 发送一条新的带有图片的 Twitter 消息

下面的代码是为 Twitter 呈现 SLComposeViewController 最简易的方法。第一步是调用 isAvailableForServiceType，如果设备不支持发送给 Twitter 的能力，程序会正常退出。创建一个新的 SLComposeViewController 和一个新的 block 来处理动作的结果。SLComposeViewController 的 completion handler 被设置为新创建的 block，并使用 presentViewController 进行呈现。要实现从 iOS 应用将消息发送到 Twitter 的功能，以上这些步骤是最低要求。示例程序在 Twitter 菜单下的 Composer 选项中对上述内容进行了演示。

```
if([SLComposeViewController
➥isAvailableForServiceType:SLServiceTypeTwitter])
{

    SLComposeViewController *controller =
    ➥[SLComposeViewController
    ➥composeViewControllerForServiceType:SLServiceTypeTwitter];

    SLComposeViewControllerCompletionHandler myBlock =
    ^(SLComposeViewControllerResult result){
        if(result == SLComposeViewControllerResultCancelled)
        {
            NSLog(@"Cancelled");
        }

        else
        {
            NSLog(@"Done");
        }
```

```
            [controller dismissViewControllerAnimated:YES
            ➥completion:nil];
        };

        controller.completionHandler = myBlock;

        [self presentViewController:controller animated:YES
        ➥completion:nil];
    }

    else
    {
        NSLog(@"Twitter Composer is not available.");
    }
```

还可以通过设置初始文本、图片和 URL 来自定义一个 SLComposeViewController：

```
[controller setInitialText:@"Check out my app:"];
[controller addImage:[UIImage imageNamed:@"Kitten.jpg"]];
[controller addURL:[NSURL URLWithString:@"http://amzn.to/Um85L0"]];
```

可以调用 addImage 和 addURL 来添加多个附件：

```
[controller addImage:[UIImage imageNamed:@"Kitten1.jpg"]];
[controller addImage:[UIImage imageNamed:@"Kitten2.jpg"]];
```

如果在添加附件之后又想从 SLComposeViewController 中移除 URL 或图片，可以调用一些简单的方法，如下所示：

```
[controller removeAllImages];
[controller removeAllURLs];
```

SLComposeViewController 关于 Facebook 的实现方法与 Twitter 类似，只有一点区别，就是需要将 SLServiceTypeTwitter 方法替换成 SLServiceTypeFacebook。

16.4　使用自定义界面发送消息

很多时候需要在上面介绍使用 SLComposeViewController 的基础上更进一步，以实现一个整体的解决方法。幸运的是，社交框架完全支持这种自定义操作。在前面使用 SLComposeViewController 的示例中，向 Facebook 发送消息和向 Twitter 发送消息的区别很小，不过当准备处理自定义界面时情况就不再是这样了，在底层对 Facebook 和 Twitter 的实现几乎是完全不同的。本节分为两个子小节，一个小节针对 Twitter，另一个小节针对 Facebook。Twitter 的处理相对简单，所以我们先介绍有关 Twitter 的实现。

16.4.1　向 Twitter 发送消息

除了导入 Social.framework 框架和从 SLComposeViewController 导入"Social/Social.h"头文件外，还需要导入"Accounts/Accounts.h"头文件。在开始直接访问 Twitter 的 API 之前，需要

先创建两个新对象。

```
ACAccountStore *account = [[ACAccountStore alloc] init];
ACAccountType *accountType = [account
➥accountTypeWithAccountTypeIdentifier:ACAccountTypeIdentifierTwitter];
```

ACAccountStore 允许代码访问在 Settings.app 中配置好的 Twitter 账户，ACAccountType 包含特定账户类型所需的信息。可以查询 accountType 对象来判断访问是否被授权给用户。

```
if(accountType.accessGranted)
{
    NSLog(@"User has already granted access to this service");
}
```

要弹出用户对 Twitter 账户进行授权访问的界面，需要在 ACAccountStore 上调用 requestAccessToAccountsWithType:options:completion:。如果账户已经被授权，completion block 会将 YES 返回给 granted 参数并不再提示用户。

```
[account requestAccessToAccountsWithType:accountType options:nil
➥completion:^(BOOL granted, NSError *error)
```

如果用户授权访问或访问已经得到授权，就需要获取用户 Twitter 账户的列表。一个用户可以向设备添加多个 Twitter 账户，程序此时还不知道用户希望使用哪个账户发送消息。如果发现有多个账户的情况，应该提示用户指定一个本次使用的账户。

```
if(granted == YES)
{
    NSArray *arrayOfAccounts = [account accountsWithAccountType:
    ➥accountType];
}
```

示例程序中，为了简化处理，如果出现多个账户，会自动选择最后一个账户。在 App Store 中，如果有不止一个账户，一定要让用户选择所使用的账户。

```
if([arrayOfAccounts count] > 0)
{
    ACAccount *twitterAccount = [arrayOfAccounts lastObject];
}
```

在对创建的账户创建引用并保存在 ACAccount 对象中之后，可以开始配置发送数据了。根据发送内容是否包含图片或其他媒体资源，需要使用不同的 URL。

```
NSURL *requestURL = nil;

if(hasAttachmentedImage)
{
    requestURL = [NSURL URLWithString:
    ➥@"https://upload.twitter.com/1.1/statuses/
    ➥update_with_media.json"];
}
```

```
else
{
    requestURL = [NSURL URLWithString:
    ➥@"http://api.twitter.com/1.1/statuses/update.json"];
}
```

警告

将一条 tweet 发送到不正确的 URL 将会导致无法处理的错误。不可以将带有图片的 tweet 发送到 update.json 端点，也不能将非图片 tweet 发送到 update_with_media.json 端点。

在端点 URL 确定之后，创建一个新的 SLRequest 对象。SLRequest 对象包含将全部 tweet 详情发送到 TwitterAPI 所需的所有信息。

```
SLRequest *postRequest = [SLRequest
➥requestForServiceType:SLServiceTypeTwitter
➥requestMethod:SLRequestMethodPOST
➥URL:requestURL parameters:nil];
```

创建 SLRequest 后，必须为其定义一个账户。这里使用前面确定好的账户，对账户属性进行设置。

```
postRequest.account = twitterAccount;
```

要为这条 tweet 添加文本，对 postRequest 调用 addMultipartData:withName:type:filename: 方法。文本是一个简单的 NSUTF8StringEncoding 字符串。这里用到的名称遵循 Twitter API 文档的要求，对于文本使用 status 值，对于 type 参数，根据 Twitter API 的用法使用 multipart/form-data 值。这里文本不需要文件名。

```
[postRequest addMultipartData:[socialTextView.text dataUsingEncoding:
➥NSUTF8StringEncoding] withName:@"status"
➥type:@"multipart/form-data"filename:nil];
```

注意

欲了解更多关于 Twitter API 的信息及其常量的含义，可以访问 https://dev.twitter.com/docs 网站进行查看。

如果一条 tweet 消息想要带上图片，需要进行添加。首先需要使用 UIImageJPEGRepresentation 将 UIImage 对象转换为 NSData。这里的例子与之前基于文本的例子基本一样，只是文件名不同而已。

```
NSData *imageData = UIImageJPEGRepresentation(self.attachmentImage, 1.0);

[postRequest addMultipartData:imageData withName:@"media"
type:@"image/jpeg" filename:@"Image.jpg"];
```

注意

重复调用 addMultipartData:withName:type:filename:方法可以添加多张图片。

在用 tweet 所需的所有信息完全填充 postRequest 之后，就可以将它发送到 Twitter 服务器。为此，可以调用 performRequestWithHandler:。URLResponse 返回代码为 200 意味着发送成功，任何其他返回代码都表示相应的错误类型。示例程序中消息发送成功的界面如图 16-5 所示。

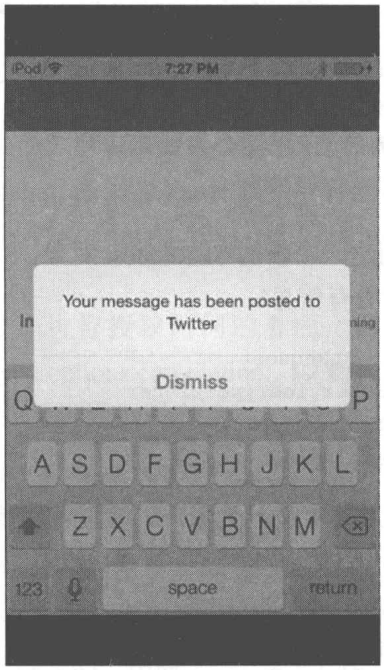


图 16-5 在示例程序中使用自定义界面成功发送一条 tweet 到 Twitter

注意

切记，UIAlertViews 不可以从内部 completion block 直接显示，因为 completion block 不需要一定在主线程上执行。示例程序中，要将错误信息传递到主线程方法来显示提醒。

```
[postRequest performRequestWithHandler:^(NSData *responseData,
➥NSHTTPURLResponse *urlResponse, NSError *error)
{
    if(error != nil)
    {
        [self performSelectorOnMainThread:
        ➥@selector(reportSuccessOrError:) withObject:[error
        ➥localizedDescription] waitUntilDone:NO];
    }

    if([urlResponse statusCode] == 200)
    {
        [self performSelectorOnMainThread:
        ➥@selector(reportSuccessOrError:) withObject:@"Your
        ➥message has been posted to Twitter" waitUntilDone:NO];
    }

}];
```

至此，就完成了使用自定义界面发送字符串和图片到 Twitter 的所有步骤。在下面的子小节中，将仔细探讨 Facebook 发送消息的方法。

提示

在示例程序中，向 Twitter 发送消息的所有过程都在 twitterPost 方法中进行。

16.4.2　向 Facebook 发送消息

处理 Facebook 消息发送时的基本原则与 Twitter 的情况一样，不过需要额外进行诸如认证、授权请求等步骤。与 Twitter 不同，Facebook 具有不同级别的认证。如果用户授权一个应用访问他的 feed，他们可能不希望应用能够发布他们的 feed。为了使事情更加复杂，认证必须按照一定的顺序进行请求，并且对于读和写的权限请求还不能同时发生。

16.4.3　创建 Facebook 应用

要从移动应用向 Facebook 发送消息或者同 Facebook 进行交互，首先需要创建一个和这个移动应用相关联的 Facebook 应用。

使用希望应用拥有的 Facebook 账户登录 https://developers.facebook.com/apps 网站。

单击+ Create New App 按钮，如图 16-6 所示。在 App Name 和 App Namespace 栏输入相应的值。单击问号图标可以显示额外的说明。

创建好一个新的 Facebook 应用后(如图 16-7 所示)，将 App ID 号复制下来。浏览所有新应用的页面并确保其按照 iOS 应用的需要进行配置。默认情况下，本小节的介绍中不需要修改任何选项。

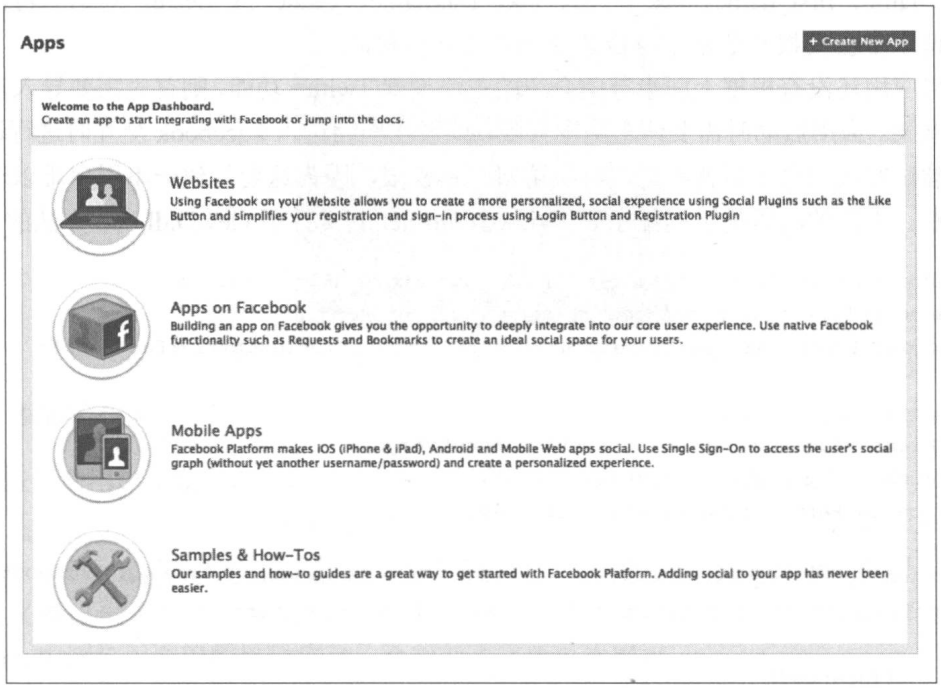


图 16-6　在 Facebook 网站的 Developers Portal 中创建一个新的 Facebook App ID

图 16-7 新创建的 Facebook 应用，iOS 应用内部调用 Facebook 时需要使用 App ID

1. 基本的 Facebook 权限

每个支持 Facebook 消息的应用首先需要请求的一组权限(除非仅使用 SLComposeViewController 类)就是基本的身份认证访问。可以通过请求下面特性中的任何一个来实现身份认证：id、name、first_name、last_name、ink、username、gender 或 locale。对这些特性中任何一个的请求都会授权其他基本身份认证元素的访问权限。

如果应用还没有根据上面小节中介绍的方法设置 Twitter 功能，则首先需要导入正确的头文件和框架。应用启动时需要请求基本权限，或者在应用进入 Facebook 交互时进行请求。用户已经尝试发送消息之后就不建议再次访问基本权限，因为这会创建一串用户难以理解的弹出提醒框。下面的代码是示例程序 ICFViewController.m 文件中 viewDidLoad:方法的一部分：

```
ACAccountStore *accountStore = [[ACAccountStore alloc] init];
ACAccountType *facebookAccountType = [accountStore
➥accountTypeWithAccountTypeIdentifier:ACAccountTypeIdentifierFacebook];

NSDictionary *options = @{
ACFacebookAudienceKey:ACFacebookAudienceEveryone,
ACFacebookAppIdKey:@"363120920441086",
ACFacebookPermissionsKey:@[@"email"]};

[accountStore requestAccessToAccountsWithType:facebookAccountType
➥options:options completion:^(BOOL granted, NSError *error)
{
    if(granted)
    {
        NSLog(@"Basic access granted");
```

```
        }

        else
        {
            NSLog(@"Basic access denied");
        }
    }];
```

在前面一节中介绍 Twitter 的功能时提到，ACAccountStore 和 ACAccountType 的配置方法类似。新建一个名为 options 的字典对象，为所有相关的调用方法提供 API 参数。对于基本权限，为 ACFacebookAudienceKey 传递参数 ACFacebookAudienceEveryone。ACFacebookAppIdKey 是“创建 Facebook 应用”一节中创建的 App ID。由于任何基本权限都可以作为所有基本权限的请求访问，因此 email 特性可以用于 ACFacebookPermissionsKey。在 accountStore 对象上调用 requestAccessToAccountWithType:options:completion:方法，这会向用户呈现一个对话框，如图 16-8 所示。用户授权访问或拒绝访问的结果都将被记录下来。

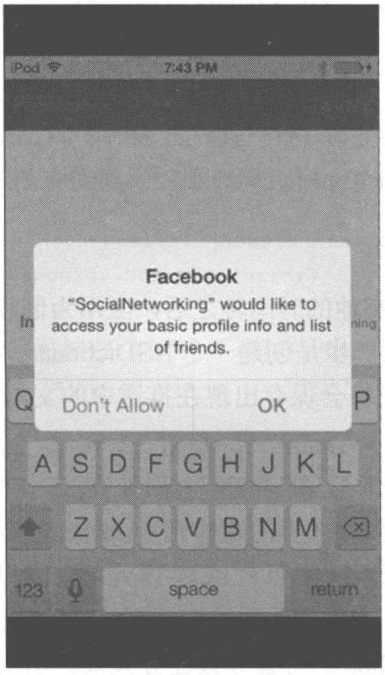


图 16-8　示例程序 SocialNetworking 询问用户是否可以访问基本身份认证信息

2. 发布流权限

在应用向用户流发送消息之前，首先需要请求写入权限。这一步骤必须在基本权限完成授权之后才能进行。请求发布权限几乎等于请求基本身份认证信息的授权。这里不再对 ACFacebookPermissionsKey 请求访问 email 特性了，而是请求 publish_stream 权限。将会以用户的名义弹出一个授权访问对话框，以发布一条新的消息。在用户得到授权后，除非在 Facebook 中删除了应用的权限信息，否则就不需要再次授权了。

```
ACAccountStore *accountStore = [[ACAccountStore alloc] init];
ACAccountType *facebookAccountType = [accountStore
```

```
➥accountTypeWithAccountTypeIdentifier:ACAccountTypeIdentifierFacebook];

NSDictionarvy *options = @{
ACFacebookAudienceKey:ACFacebookAudienceEveryone,
ACFacebookAppIdKey:@"363120920441086",
ACFacebookPermissionsKey:@[@"publish_stream"]};

[accountStore requestAccessToAccountsWithType:facebookAccountType
➥options:options completion:^(BOOL granted, NSError *error)
{
     if(granted)
     {
          NSLog(@"Publish permission granted");
     }

     else
     {
          NSLog(@"Publish permission denied");
     }
}];
```

注意

不要忘记修改 ACFacebookAppIdKey 以匹配你要发布的 Facebook 应用的 ID。

3. 发送 Facebook 流

用户以自己的名义授权发布他的时间轴之后，应用为创建新的消息推送做好了准备。创建一个新的 Facebook 推送的第一步是创建一个 NSDictionary，其保存一个单独的对象，对应的键为@"message"。该键/值对将会保存出现在推送中的文本。

```
NSDictionary *parameters = [NSDictionary
➥dictionaryWithObject:socialTextView.text forKey:@"message"];
```

如果推送不包含任何类似图片之类的媒体元素，消息就会被发送到 https://graph.facebook.com/me/feed；但如果新的推送包含照片或媒体元素，消息将会被发送到 https://graph.facebook.com/me/photos。这些 URL 是不能混淆的，例如发送一个不包含图片的 feed 到 https://graph.facebook.com/me/photos 将会导致任务失败。示例程序执行一个简单的检查来确定要使用的端点。

```
if(self.attachmentImage)
{
    feedURL = [NSURL URLWithString:
    ➥@"https://graph.facebook.com/me/photos"];
}

else
{
    feedURL = [NSURL URLWithString:
    ➥@"https://graph.facebook.com/me/feed"];
}
```

在确认 URL 正确之后，创建一个新的 SLRequest 对象用于指定 URL 和参数。

```
SLRequest *feedRequest = [SLRequest

requestForServiceType:SLServiceTypeFacebook
                        requestMethod:SLRequestMethodPOST
                        URL:feedURL
                        parameters:parameters];
```

如果推送包含图片，需要将图片数据添加到 feedRequest 中。可以使用 addMultipartData:withName:type:filename:方法实现这一步骤。

```
if(self.attachmentImage)
{
    NSData *imageData =
    ➥UIImagePNGRepresentation(self.attachmentImage);
    ➥[feedRequest addMultipartData:imageData withName:@"source"
    ➥type:@"multipart/form-data" filename:@"Image"];
}
```

在添加完(可选的)图片数据后，同 Twitter 的处理一样，需要调用 performRequestWithHandler:方法。Facebook 会返回一个 urlResponse，如果代码等于 200，则说明推送已经成功。

```
[feedRequest performRequestWithHandler:^(NSData *responseData,
➥NSHTTPURLResponse *urlResponse, NSError *error)
{
    NSLog(@"Facebook post statusCode: %d", [urlResponse
    ➥statusCode]);

    if([urlResponse statusCode] == 200)
    {
       [self performSelectorOnMainThread:@selector
        ➥(reportSuccessOrError:) withObject:@"Your message has
        ➥been posted to Facebook" waitUntilDone:NO];
    }

    else if(error != nil)
    {
        [self performSelectorOnMainThread:
         ➥@selector(faceBookError:) withObject:error
         ➥waitUntilDone:NO];
    }
}];
```

关于格式化推送消息从及在 Facebook 中嵌入媒体元素的其他内容可以通过 http://developers.facebook.com 网站上的文档进行查看。

16.5 访问用户时间轴

仅仅推送状态更新信息往往无法满足社交类应用对交互的需求。在 Twitter 和 Facebook 上访问时间轴数据有点复杂，因为涉及大量复杂的边界情况和数据类型兼容性问题，比如从 Twitter 的 retweets 到 Facebook 的内置数据类型等。本节将简述如何从时间轴访问原始数据并将其显示在 tableView 上。有关时间轴的问题一点也不简单，甚至可以用一整本书的篇幅来讲解它。

16.5.1 Twitter

在前面一节中介绍过，Twitter 的交互与更加复杂的 Facebook API 相比会简单一点，Facebook 主要是因为需要实现多个层级的权限请求。访问用户的 Twitter 时间轴在开始时同推送一条新的 tweet 的方法一样，需要创建对 ACAccountStore 和 ACAccountType 的引用。

```
ACAccountStore *account = [[ACAccountStore alloc] init];

ACAccountType *accountType = [account
➥accountTypeWithAccountTypeIdentifier:ACAccountTypeIdentifierTwitter];
```

接下来仍然同推送一条新的 tweet 的方法一样，对账户对象调用 requestAccessToAccountWithType:方法。基础的错误处理也在这个方法中进行。

```
[account requestAccessToAccountsWithType:accountType options:nil
➥completion:^(BOOL granted, NSError *error)
{
    if(error != nil)
    {
        [self
    performSelectorOnMainThread:@selector(reportSuccessOrError:)
    ➥withObject:[error localizedDescription]
    ➥waitUntilDone:NO];
    }

}];
```

如果没有返回错误且访问得到授权，用户会得到一个 ACAccount 对象的副本。这里对于示例程序，再次选择账户数组中的最后一个对象，不过一定要时刻注意有些用户可能同时登录多个 Twitter 账户，所以要给用户提供选择账户的功能。用户获取用户时间轴副本的请求 URL 是 http://api.twitter.com/1.1/statuses/home_timeline.json。还需要为其指定大量的选项。首先需要指定 count，用于表示在每次调用时获取的 tweet 个数。第二个选项是一个布尔类型值，用于表示 tweet 条目是否应该被包含进来。一个 tweet 条目可以包含一些额外的详情信息，比如用户提醒、标签、URL 和媒体资源。

上述对象创建完成后，需要提交 SLRequest，提交方法同推送新的状态更新一样。performRequestWithHandler 成功 block 将会包含之后可以显示的 responseData 对象。下面的代码是 ICFViewController.m 文件中 twitterTimeline 方法的一部分：

```
if(granted == YES)
{
    NSArray *arrayOfAccounts = [account
    ➥accountsWithAccountType:accountType];

    if([arrayOfAccounts count] > 0)
    {
        ACAccount *twitterAccount = [arrayOfAccounts
        ➥lastObject];

        NSURL *requestURL = [NSURL URLWithString:
        @"http://api.twitter.com/1.1/statuses/home_timeline.json"];

        NSDictionary *options = @{
        @"count" : @"20",
        @"include_entities" : @"1"};

        SLRequest *postRequest = [SLRequest
        ➥requestForServiceType:SLServiceTypeTwitter
        ➥requestMethod:SLRequestMethodGET
        ➥URL:requestURL parameters:options];

        postRequest.account = twitterAccount;

        [postRequest performRequestWithHandler:^(NSData
        ➥*responseData, NSHTTPURLResponse *urlResponse, NSError
        ➥*error)
        {
            if(error != nil)
            {
                [self performSelectorOnMainThread:@selector
                ➥(reportSuccessOrError:) withObject:[error
                ➥localizedDescription] waitUntilDone:NO];
            }

             [self performSelectorOnMainThread:
             ➥@selector(presentTimeline:) withObject:
             ➥[NSJSONSerialization JSONObjectWithData:responseData
             ➥options:NSJSONReadingMutableLeaves error:&error]
             ➥waitUntilDone:NO];
        }];
    }
}
```

这里给出的是一个从典型 Twitter 时间轴获取的带有 tweet 条目的 responseData 示例，此外，该 tweet 如何在 Twitter 网站进行显示的例子如图 16-9 所示。根据下面控制台上的输出内容，Twitter 提供了大量有关哪个开发者进行了访问的相关信息。欲了解更多有关使用 JSON 数据的信息，可以参阅第 9 章“JSON 的使用和解析”。

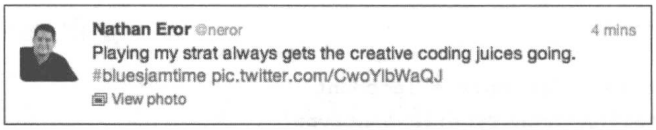


图 16-9 Twitter 网站上的一个完整 tweet，构成这个 tweet 的数据可以在本节前面的内容中找到

```
2013-02-27 21:50:54.562 SocialNetworking[28672:4207] (
      {
      contributors = "<null>";
      coordinates = "<null>";
      "created_at" = "Thu Feb 28 02:50:41 +0000 2013";
      entities =       {
         hashtags =          (
                        {
               indices =                (
                  63,
                  76
               );
               text = bluesjamtime;
            }
         );
         media =          (
                        {
               "display_url" = "pic.twitter.com/CwoYlbWaQJ";
               "expanded_url" =
➥"http://twitter.com/neror/status/306959580582248448/photo/1";
               id = 306959580586442753;
               "id_str" = 306959580586442753;
               indices =                (
                  77,
                  99
               );
               "media_url" =
➥"http://pbs.twimg.com/media/BEKKHLlCAAEUQ6x.jpg";
               "media_url_https" =
➥"https://pbs.twimg.com/media/BEKKHLlCAAEUQ6x.jpg";
               sizes =                {
                  large =                   {
                     h = 768;
                     resize = fit;
                     w = 1024;
                  };
                  medium =                   {
                     h = 450;
                     resize = fit;
                     w = 600;
                  };
                  small =                   {
                     h = 255;
                     resize = fit;
                     w = 340;
```

```
            };
            thumb =                    {
                h = 150;
                resize = crop;
                w = 150;
            };
        };
        type = photo;
        url = "http://t.co/CwoYlbWaQJ";
    }
);
urls =          (
);
"user_mentions" =          (
);
};
favorited = 0;
geo = "<null>";
id = 306959580582248448;
"id_str" = 306959580582248448;
"in_reply_to_screen_name" = "<null>";
"in_reply_to_status_id" = "<null>";
"in_reply_to_status_id_str" = "<null>";
"in_reply_to_user_id" = "<null>";
"in_reply_to_user_id_str" = "<null>";
place =       {
    attributes =          {
    };
    "bounding_box" =          {
        coordinates =              (
                        (
                            (
                "-95.90998500000001",
                "29.537034"
            ),
                            (
                "-95.01449599999999",
                "29.537034"
            ),
                            (
                "-95.01449599999999",
                "30.110792"
            ),
                            (
                "-95.90998500000001",
                "30.110732"
            )
        )
        );
        type = Polygon;
```

```
            };
            country = "United States";
            "country_code" = US;
            "full_name" = "Houston, TX";
            id = 1c69a67ad480e1b1;
            name = Houston;
            "place_type" = city;
            url =
➥"http://api.twitter.com/1/geo/id/1c69a67ad480e1b1.json";
        };
        "possibly_sensitive" = 0;
        "retweet_count" = 0;
        retweeted = 0;
        source = "<a href=\http://tapbots.com/software/tweetbot/mac\
➥rel=\"nofollow\">Tweetbot for Mac</a>";
        text = "Playing my strat always gets the creative coding juices
➥going. #bluesjamtime http://t.co/CwoYlbWaQJ";
        truncated = 0;
        user =        {
            "contributors_enabled" = 0;
            "created_at" = "Mon Sep 04 02:05:35 +0000 2006";
            "default_profile" = 0;
            "default_profile_image" = 0;
            description = "Dad, iOS & Mac game and app developer,
➥Founder of Free Time Studios, Texan";
            "favourites_count" = 391;
            "follow_request_sent" = "<null>";
            "followers_count" = 2254;
            following = 1;
            "friends_count" = 865;
            "geo_enabled" = 1;
            id = 5250;
            "id_str" = 5250;
            "is_translator" = 0;
            lang = en;
            "listed_count" = 182;
            location = "Houston, Texas";
            name = "Nathan Eror";
            notifications = "<null>";
            "profile_background_color" = 1A1B1F;
            "profile_background_image_url" =
➥"http://a0.twimg.com/images/themes/theme9/bg.gif";
            "profile_background_image_url_https" =
➥"https://si0.twimg.com/images/themes/theme9/bg.gif";
            "profile_background_tile" = 0;
            "profile_image_url" =
➥"http://a0.twimg.com/profile_images/1902659692/36A2FDF8-72F4-
➥485E-B574-892C1FF16534_normal";
            "profile_image_url_https" =
➥"https://si0.twimg.com/profile_images/1902659692/36A2FDF8-72F4-
```

```
➥485E-B574-892C1FF16534_normal";
            "profile_link_color" = 2FC2EF;
            "profile_sidebar_border_color" = 181A1E;
            "profile_sidebar_fill_color" = 252429;
            "profile_text_color" = 666666;
            "profile_use_background_image" = 1;
            protected = 0;
            "screen_name" = neror;
            "statuses_count" = 5091;
            "time_zone" = "Central Time (US & Canada)";
            url = "http://www.freetimestudios.com";
            "utc_offset" = "-21600";
            verified = 0;
        };
    }
)
```

16.5.2　Facebook

获取 Facebook 时间轴可以通过访问端点 https://graph.facebook.com/me/feed 来实现。首先创建一个新的 NSURL 并使用它生成一个新的 SLRequest 对象。下面的例子假设应用在前面已经被授予用户的权限。可以参阅前面的“基本的 Facebook 权限”一节以了解相关的内容。

```
NSURL *feedURL = [NSURL URLWithString:
➥@"https://graph.facebook.com/me/feed"];

SLRequest *feedRequest = [SLRequest

requestForServiceType:SLServiceTypeFacebook
                        requestMethod:SLRequestMethodGET
                        URL:feedURL
                        parameters:nil];

feedRequest.account = self.facebookAccount;
```

在设置好 SLRequest 后，就会调用 feedRequest 对象上的 performRequestWithHandler:方法。如果推送成功，Facebook 会返回 urlResponse 状态码 200；若返回任何其他状态码，则意味着推送失败。

```
[feedRequest performRequestWithHandler:^(NSData *responseData,
➥NSHTTPURLResponse *urlResponse, NSError *error)
{
    NSLog(@"Facebook post statusCode:%d", [urlResponse
    ➥statusCode]);

    if([urlResponse statusCode] == 200)
    {
        NSLog(@"%@", [[NSJSONSerialization
        ➥JSONObjectWithData:responseData
        ➥options:NSJSONReadingMutableLeaves error:&error]
```

```
            ➥objectForKey:@"data"]);

            [self performSelectorOnMainThread:
            ➥@selector(presentTimeline:) withObject:
            ➥[[NSJSONSerialization JSONObjectWithData:responseData
            ➥options:NSJSONReadingMutableLeaves error:&error]
            ➥objectForKey:@"data"] waitUntilDone:NO];
        }

        else if(error != nil)
        {
            [self performSelectorOnMainThread:
            ➥@selector(faceBookError:) withObject:error
            ➥waitUntilDone:NO];
        }
    }];
```

Facebook 支持许多推送更新类型，从喜欢、评论、新朋友到更新留言板。这些字典对象使用不同的键对信息进行设置。示例程序将会处理最常见的 Facebook 推送类型。下面介绍 3 种标准的推送类型及其所有相关的数据。第一个是用于表示当有一个新 Facebook 好友登录时的信息。数组中的第二个元素是用于表示用户喜欢的推送链接。最后的例子演示了用户向其他好友的推送添加一条评论。很重要的一点是，对于 Facebook 时间轴用法的解析一定要进行充分的测试以确保良好的兼容性。有关格式化和 Facebook 推送行为的更多内容，可以在 http://developers.facebook.com 网站上找到相关介绍。欲了解更多有关使用 JSON 数据的内容，可以参阅第 9 章。

```
(
    {
    actions =        (
                {
            link =
➥"http://www.facebook.com/1674990377/posts/4011976152528";
            name = Comment;
        },
                {
            link =
➥"http://www.facebook.com/1674990377/posts/4011976152528";
            name = Like;
        }
    );
    comments =        {
        count = 0;
    };
    "created_time" = "2013-02-10T18:26:44+0000";
    from =        {
        id = 1674990377;
        name = "Kyle Richter";
```

```
    };
    id = "1674990377_4011976152528";
    privacy =        {
        value = "";
    };
    "status_type" = "approved_friend";
    story = "Kyle Richter and Kirby Turner are now friends.";
    "story_tags" =        {
        0 =            (
                        {
                id = 1674990377;
                length = 12;
                name = "Kyle Richter";
                offset = 0;
                type = user;
            }
        );
        17 =            (
                        {
                id = 827919293;
                length = 12;
                name = "Kirby Turner";
                offset = 17;
                type = user;
            }
        );
    };
    type = status;
    "updated_time" = "2013-02-10T18:26:44+0000";
},
    {
    comments =        {
        count = 0;
    };
    "created_time" = "2013-01-03T00:58:41+0000";
    from =        {
        id = 1674990377;
        name = "Kyle Richter";
    };
    id = "1674990377_3785554092118";
    privacy =        {
        value = "";
    };
    story = "Kyle Richter likes a link.";
    "story_tags" =        {
        0 =            (
```

```
                    {
                id = 1674990377;
                length = 12;
                name = "Kyle Richter";
                offset = 0;
                type = user;
            }
        );
    };
    type = status;
    "updated_time" = "2013-01-03T00:58:41+0000";
},
    {
    application =        {
        id = 6628568379;
        name = "Facebook for iPhone";
        namespace = fbiphone;
    };
    comments =        {
        count = 0;
    };
    "created_time" = "2013-01-02T19:20:59+0000";
    from =        {
        id = 1674990377;
        name = "Kyle Richter";
    };
    id = "1674990377_3784462784836";
    privacy =        {
        value = "";
    };
    story = "\"Congrats!\" on Dan Burcaw's link.";
    "story_tags" =        {
        15 =            (
                    {
                id = 10220084;
                length = 10;
                name = "Dan Burcaw";
                offset = 15;
                type = user;
            }
        );
    };
    type = status;
    "updated_time" = "2013-01-02T19:20:59+0000";
})
```

16.6　小结

本章介绍了将 Twitter 和 Facebook 功能整合到 iOS 应用中的基础知识，从内置的整合工具到如何编写高度自定义的推送机制，都进行了详细介绍。此外，读者还学习了如何获取时间轴和 feed 数据并进行显示。

社交媒体的整合从来都不是一个简单的话题，不过随着社交框架的增强和苹果公司承诺继续对第三方应用开放整合，这些功能会变得越来越简单。创建一个优秀的社交应用(即包含 Twitter 和 Facebook 交互功能的应用)所需的技术现在变得更加清晰了。

第17章

后台任务处理

当 2008 年 iOS 首次推出时，同一时间只能有一个第三方应用被激活——前端应用。这就意味着所有任务都要在应用处于前台被激活时才能完成，否则就必须暂停，等下一次应用重新在前台激活时再恢复执行。iOS 4 的推出，为第三方应用加入了后台运行的功能。由于 iOS 设备对于系统资源的使用有限制，并且电池电量也是优先要考虑的元素，因此后台处理就自然存在许多限制，以避免对前端运行的程序产生干扰，同时也要避免消耗过多的电量。应用可以通过正确运用后台功能来完成许多任务。本章介绍可以使用的这些功能以及如何实现这些功能。

iOS 支持两种方法来实现后台任务的处理，分别如下：

- 第一种方法是在后台完成一个长线任务。这种方法对于完成诸如下载大数据或数据更新类的任务比较合适，因为通常下载时用户也无法同应用进行交互。
- 第二种方法是支持 iOS 允许的指定类型的后台活动，比如播放音乐、同蓝牙设备的交互、获取最新的应用内容、监测 GPS 的显著变化以及维护持久的网络连接以支持 VoIP 类应用的正常运行。

注意

“后台任务”这个术语经常在两个不同的含义间互用，一个是指当应用不在前端运行时(本章会有描述)执行的任务，另一个含义是指同主线程异步运行的任务(在第 18 章“多线程编程的性能”中会有描述)。此外，NSURLSession 提供终止应用时上传和下载文件的功能，当上传或下载完成时还可以重启相应的应用。

17.1 示例程序

本章的示例程序为 BackgroundTasks。这个应用对后台运行的两种方法都给出了演示：一个是当应用进入后台时完成一个长线任务，另一个是在应用处于后台时持续播放音频文件。用户界面很简单——有一个按钮用于开始和停止后台音乐的播放，还有一个按钮用于开始后台任务的执行，如图 17-1 所示。

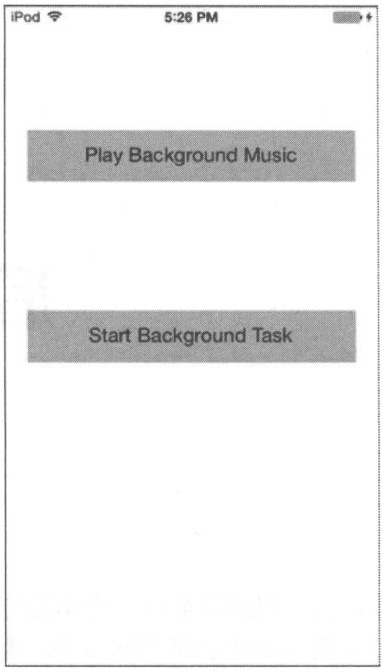


图 17-1 示例程序 BackgroundTasks

17.2 检查后台运行的可行性

所有能够运行 iOS 5 版本或更高版本系统的设备都支持第三方程序在后台运行，在苹果公司的文档中也将其称为多任务处理。如果目标应用仅支持 iOS 4(或者未来可能发布的新设备)，要注意可能有些设备无法支持多任务处理。所有使用多任务处理功能的代码在编写时都要注意对设备是否支持该功能进行检查。当用户在示例程序中点击 Start Background Task 按钮时，将会调用 ICFViewController 类的 startBackgroundTaskTouched:方法检查设备是否支持多任务处理。

```
-(IBAction)startBackgroundTaskTouched:(id)sender
{
    UIDevice* device = [UIDevice currentDevice];

    if(![device isMultitaskingSupported])
    {
        NSLog(@"Multitasking not supported on this device.");
        return;
    }

    [self.backgroundButton setEnabled:NO];
    NSString *buttonTitle =@"Background Task Running";

    [self.backgroundButton setTitle:buttonTitle
                           forState:UIControlStateNormal];

    dispatch_queue_t background =
```

```
    ➥dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

    dispatch_async(background, ^{
        [self performBackgroundTask];
    });

}
```

要检查多任务处理的可行性，可以对 UIDevice 对象使用类方法 currentDevice 来获得关于当前设备的信息。之后调用 isMultitaskingSupported 方法确定其是否支持多任务处理。如果支持多任务处理，则更新用户界面并异步调用 performBackgroundTask 方法，开始后台任务的处理(查看第 18 章可以了解更多有关异步执行任务的内容)。

17.3　在后台完成任务

要在后台执行一个长线任务，应用需要得到该任务能够在后台运行的通知。任务对于内存的消耗也要给予考虑，还要考虑运行完任务所需的总时长。如果任务持续 10 到 15 分钟，应用可能在任务完成前就终止了。任务逻辑应该能够处理快速的终止活动，并能够在应用重新启动时也重启任务的执行。在应用中可以设置用于完成后台任务的总时间，如果操作系统监测到任务所需的资源都已获得，可以在总时间之前终止程序。

要实际观察后台运行的功能，从 Xcode 打开示例程序，如图 17-1 所示。点击 Start Background Task 按钮，之后点击 Home 按钮退出应用。在 Xcode 控制台观察应用所生成的日志描述，通过日志描述可以确定程序正在后台运行。

```
Background task starting, task ID is 1.
Background Processed 0. Still in foreground.
Background Processed 1. Still in foreground.
Background Processed 2. Still in foreground.
Background Processed 3. Still in foreground.
Background Processed 4. Still in foreground.
Background Processed 5. Still in foreground.
Background Processed 6. Time remaining is: 599.579052
Background Processed 7. Time remaining is: 598.423525
Background Processed 8. Time remaining is: 597.374849
Background Processed 9. Time remaining is: 596.326780
Background Processed 10. Time remaining is: 595.308253
```

执行后台任务的一般过程如下：

(1) 从应用程序请求一个后台任务标识符，指定一个代码块作为超时处理程序。该超时处理程序仅在应用后台运行时间用完或系统确认资源使用率过高时才会调用，此时应用应该被终止。

(2) 执行后台任务逻辑，其中包含介于请求后台任务标识符和后台任务完成之间的所有代码。

(3) 告诉应用结束后台任务，验证后台任务标识符。

17.3.1 后台任务标识符

要开始一个完全在后台运行的任务，需要从应用获得后台任务的标识符。后台任务标识符帮助应用对任务的执行进行跟踪及判断任务何时完成。需要使用后台任务标识符来告诉应用，任务已经完成不再需要后台处理了。ICFViewController 类的 performBackgroundTask 方法可以在后台工作开始之前获取后台任务标识符。

```
__block UIBackgroundTaskIdentifier bTask =
➥[[UIApplication sharedApplication]
➥beginBackgroundTaskWithExpirationHandler:
➥^{
    ...
}];
```

当得到一个后台任务时，需要为其指定一个超时处理程序。后台任务标识符的声明完成后，之所以会使用_block 修饰符的原因是超时处理程序需要后台任务标识符，并且在超时处理程序中需要对该标识符进行修改。

17.3.2 超时处理程序

如果操作系统确定应用执行所需的时间和/或资源已经用完且需要关闭，会调用后台任务的超时处理程序。在应用程序将要关闭前会在主线程调用超时处理程序。这一过程所需的时间很短(最多只有几秒)，所以处理程序执行的工作要尽可能少。

```
__block UIBackgroundTaskIdentifier bTask =
➥[[UIApplication sharedApplication]
➥beginBackgroundTaskWithExpirationHandler:
➥^{
    NSLog(@"Background Expiration Handler called.");
    NSLog(@"Counter is: %d, task ID is %u.",counter,bTask);

    [[UIApplication sharedApplication]
    ➥endBackgroundTask:bTask];

    bTask = UIBackgroundTaskInvalid;
}];
```

超时处理程序所做的最基本工作就是通知应用，通过对共享的应用实例调用 endBackgroundTask:方法来结束后台任务，并且通过将 bTask 变量设置为 UIBackgroundTaskInvalid 使后台任务 ID 无效，这样可避免不慎再次用到它。

```
Background Processed 570. Time remaining is: 11.482063
Background Processed 571. Time remaining is: 10.436456
Background Processed 572. Time remaining is: 9.394706
Background Processed 573. Time remaining is: 8.346616
Background Processed 574. Time remaining is: 7.308527
Background Processed 575. Time remaining is: 6.260324
Background Processed 576. Time remaining is: 5.212251
```

```
Background Expiration Handler called.
Counter is: 577, task ID is 1.
```

17.3.3　完成后台任务

在获得一个后台任务 ID 之后，就可以着手开始后台运行的实际工作了。对 performBackgroundTask 方法中的一些变量进行设置，用于让开发者知道实际工作从何时开始、持续迭代的次数以及停止时间。建立一个到 NSUserDefaultsstandardUserDefaults 的引用，用于获取用到的最后一个计数值，并在每次迭代时都存储最后一个计数值。

```
NSInteger counter = 0;

NSUserDefaults *userDefaults =
➥[NSUserDefaults standardUserDefaults];

NSInteger startCounter =
➥[userDefaults integerForKey:kLastCounterKey];

NSInteger twentyMins = 20 * 60;
```

示例程序用到的后台任务比较简单——在循环中会将线程置于几秒钟的休眠状态，以模拟大量的长线迭代任务。在 NSUserDefaults 中为迭代保存当前的计数值，这样就可以在后台任务超时后简单地跟踪应用的执行情况。还可以将这个逻辑修改为跟踪每一个重复的后台任务。

```
NSLog(@"Background task starting, task ID is %u.",bTask);
for(counter = startCounter; counter<=twentyMins; counter++)
{
    [NSThread sleepForTimeInterval:1];
    [userDefaults setInteger:counter
                      forKey:kLastCounterKey];

    [userDefaults synchronize];

    NSTimeInterval remainingTime =
    ➥[[UIApplication sharedApplication] backgroundTimeRemaining];

    NSLog(@"Background Processed %d. Time remaining is: %f",
    ➥counter,remainingTime);
}
```

当所有迭代完成时，可以从应用程序中获取后台任务剩余的时间，它可以用于确定后台任务是否还有其他的迭代需要执行。

注意

后台任务一般会在仅剩几秒钟时结束，会在这几秒钟内对对象进行封装，所以任何一个新迭代的开始都需要考虑到这一点。

在后台任务完成后，在 NSUserDefaults 中更新最后一个计数值，这样就可以正确地重新开始计数了，也需要更新 UI，这样才能使用户可以再次开始一个后台任务。

```
NSLog(@"Background Completed tasks");

[userDefaults setInteger:0
                  forKey:kLastCounterKey];

[userDefaults synchronize];

dispatch_sync(dispatch_get_main_queue(), ^{
    [self.backgroundButton setEnabled:YES];
    [self.backgroundButton setTitle:@"Start Background Task"
                           forState:UIControlStateNormal];
});
```

最后，完成后台任务还需要做两件关键事情：一是告诉应用结束后台任务，二是使后台任务标识符无效化。在获得后台任务 ID 和结束后台任务执行之间的所有代码都在后台执行。

```
[[UIApplication sharedApplication] endBackgroundTask:self.backgroundTask];

self.backgroundTask = UIBackgroundTaskInvalid;
```

17.4 实现后台活动

iOS 支持一组特殊的后台活动，可以不受后台任务标识符的限制而持续运行。这些活动可以持续运行且不受时间的限制，并且可以避免因使用过多系统资源而导致应用终止。

17.4.1 后台活动的类型

一共有下面几种后台活动：

- 播放背景音乐
- 跟踪设备位置
- 支持 VoIP 应用
- 下载新的 Newsstand 应用内容
- 同外部或蓝牙附件通信
- 获取后台数据内容
- 通过推送通知初始化一个后台下载任务

要支持所有这些后台活动，应用需要在 Info.plist 文件中对要使用的活动进行声明。为此，在 Xcode 中选择目标应用，打开 Capabilities 标签页。将 Background Modes 设置成 ON，之后选中所要支持的模式。Xcode 将会把该选中的条目添加到 Info.plist 文件中。或者，还可以在 Xcode 中选择目标应用直接对 Info.plist 文件进行修改，之后选择 Info 标签页。查看列表的 Required Background Modes 条目，如果没有显示，用鼠标悬停在一个已经存在的条目上，点击加号按钮添加一个新的条目，之后再选择 Required Background Modes。一个带有空 NSString 元素的条目数组就被添加进来，这时再选择所需的后台模式，如图 17-2 所示。

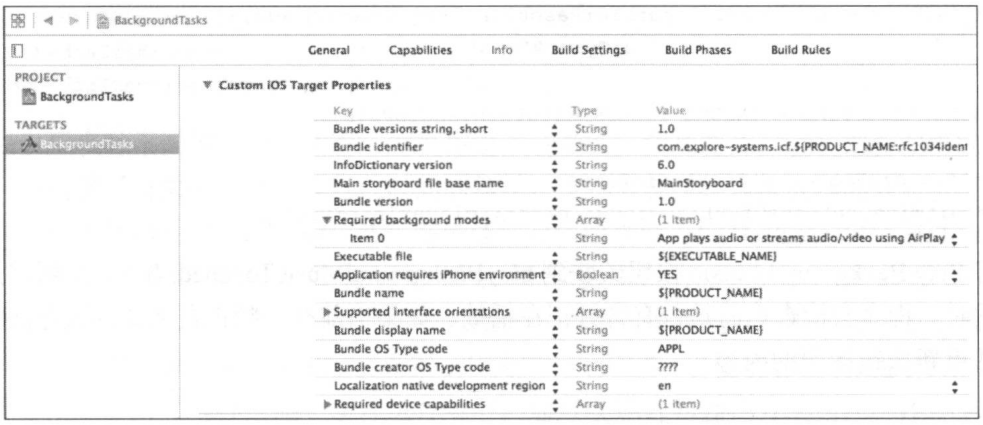


图 17-2　Xcode 的 Info 编辑器显示所需的后台模式

在创建好 Required Background Modes 条目之后，活动特有的逻辑代码可以直接在应用中创建，当应用进入后台运行时就会执行相应的功能。

17.4.2　在后台播放音乐

要在后台播放音乐，第一步就是为应用调整音频会话的设置。默认情况下，应用使用 AVAudioSessionCategorySoloAmbient 音频会话类型。使用这种类型的音频会话可以确保当应用开始时关闭其他的音频，并且当屏幕锁定或设备切换为静音模式时应用能够自动保持静音。当屏幕锁定或其他应用置于前台时音频将会停止，所以上面设置的音频会话也停止工作。在 ICFViewController 类的 viewDidLoad 方法中调整音频会话的类型为 AVAudioSessionCategory-Playback，这样就可以实现当应用置于后台或用户切换设备为静音模式时还可以继续播放音频。

```
AVAudioSession *session = [AVAudioSession sharedInstance];

NSError *activeError = nil;
if(![session setActive:YES error:&activeError])
{
    NSLog(@"Failed to set active audio session!");
}

NSError *categoryError = nil;
if(![session setCategory:AVAudioSessionCategoryPlayback
                 error:&categoryError])
{
    NSLog(@"Failed to set audio category!");
}
```

播放音频功能的下一步是初始化音频播放器，该任务仍然是在 viewDidLoad 方法中实现，这样音频播放器就可以根据用户的指示随时播放。

```
NSError *playerInitError = nil;

NSString *audioPath =
```

```
➥[[NSBundle mainBundle] pathForResource:@"background_audio"
                                  ofType:@"mp3"];

NSURL *audioURL = [NSURL fileURLWithPath:audioPath];

self.audioPlayer = [[AVAudioPlayer alloc]
➥initWithContentsOfURL:audioURL error:&playerInitError];
```

将 Play Background Music 按钮关联到 playBackgroundMusicTouched:方法。当用户点击这个按钮时，程序会检查当前是否有音频正在播放。如果当前有音频正在播放，该方法会停止音频并更新按钮标题的内容。

```
if([self.audioPlayer isPlaying])
{
   [self.audioPlayer stop];

   [self.audioButton setTitle:@"Play Background Music"
                     forState:UIControlStateNormal];

}
else
{ ...
}
```

如果当前没有音频播放，playBackgroundMusicTouched:方法会开始播放一个音频并修改按钮标题的内容。

```
[self.audioPlayer play];

[self.audioButton setTitle:@"Stop Background Music"
                  forState:UIControlStateNormal];
```

当音频播放时，用户可以按下设备的锁屏键或 home 键来让应用进入后台，此时音频会继续播放。一个更好的处理方法是当音频置于后台播放时在锁屏界面上显示当前播放的音频信息。要实现这个功能，首先设置一个带有播放中媒体信息的字典对象。

```
UIImage *lockImage = [UIImage imageNamed:@"book_cover"];

MPMediaItemArtwork *artwork =
➥[[MPMediaItemArtwork alloc] initWithImage:lockImage];

NSDictionary *mediaDict =
➥@{
   MPMediaItemPropertyTitle:@"BackgroundTask Audio",
   MPMediaItemPropertyMediaType:@(MPMediaTypeAnyAudio),
   MPMediaItemPropertyPlaybackDuration:
   @(self.audioPlayer.duration),
   MPNowPlayingInfoPropertyPlaybackRate:@1.0,
   MPNowPlayingInfoPropertyElapsedPlaybackTime:
   @(self.audioPlayer.currentTime),
```

```
MPMediaItemPropertyAlbumArtist:@"Some User",
MPMediaItemPropertyArtist:@"Some User",
MPMediaItemPropertyArtwork:artwork };
```

可以设置多种不同的选项。注意，这里指定了音频的标题和图片，这些都会在锁屏界面上显示。还提供了有关音乐时长和当前时间的信息，这些内容在媒体播放器部分显示，根据设备的状态和内容，显示的效果也不一样。在创建好媒体信息后，playBackgroundMusicTouched: 方法会开启音频播放器，之后将有关播放媒体元素的信息告知该媒体播放器的 MPNowPlayingInfoCenter。设置 self 为第一响应者，由于媒体播放器对象的信息中心需要视图或视图控制器来播放音频，因此才会设置 self 为第一响应者，以便能够正确地执行该功能。它会告诉应用开始对“远程控制”事件进行响应，通过实现好的委托方法让锁屏控件来控制应用的音频播放。

```
[[MPNowPlayingInfoCenter defaultCenter]
➥setNowPlayingInfo:mediaDict];

[self becomeFirstResponder];

[[UIApplication sharedApplication]
➥beginReceivingRemoteControlEvents];
```

现在，当音频在后台播放时，锁屏界面就会显示相关的信息，如图 17-3 所示。

此外，Control Center 将会显示有关音频的信息，如图 17-4 所示。

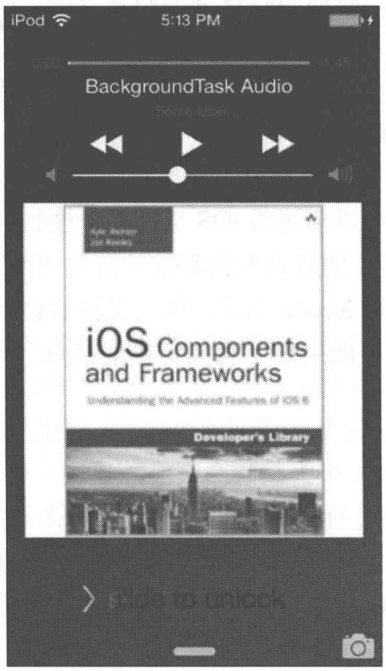


图 17-3　锁屏界面显示示例程序正在后台播放音频

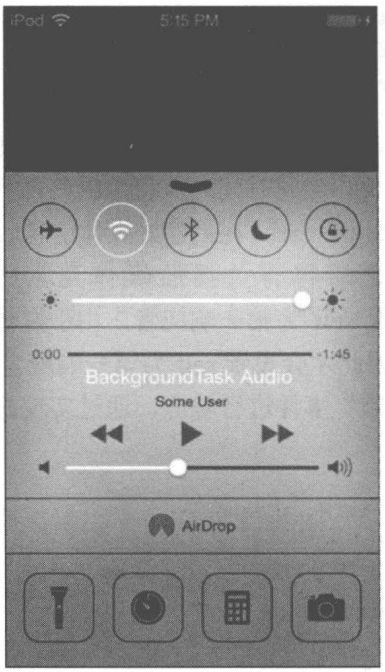


图 17-4　锁屏 Control Center 显示示例程序正在后台播放音频

17.5　小结

本章给出了当应用在后台运行时执行任务的两种方法，所谓后台运行，即应用被切换到后台或者说我们的应用不是用户当前操作的应用。

第一种后台任务处理方法可以描述为在后台执行长线任务。示例程序演示了如何检查设备是否有支持后台任务处理的能力，以及如何设置并执行后台任务，还解释了当 iOS 终止应用时如何处理后台任务的问题以及通知后台任务将要停止(或超时)。

第二种后台任务处理方法可以描述为 iOS 支持的一些特殊类型的后台活动，比如播放音乐、同蓝牙设备交互、监视 GPS 的显著变化和维护持久的网络连接以确保 VoIP 类应用的正常运行。本章还解释了如何配置 Xcode 项目，使其支持后台活动，之后示例程序演示了如何在后台播放音频，同时在锁屏界面上显示有关音频的信息。

第18章

多线程开发的性能

很多应用都对性能有很高的需求，包括需要同时执行的多处理器密集型任务和高延时的任务。本章首先展示因为主队列堵塞而导致的副作用，即用户界面的运行变慢或完全卡住，这是非常糟糕的用户体验。之后研究由 iOS 提供的能够让程序员在后台执行任务的工具，也就是说，任务照常运行但不直接对用户界面产生延迟。苹果公司对于后台任务如何完成提供了多种不同控制级别的工具。

程序的并发运行通常都会用到线程的概念。要想运行于多核设备上的应用直接通过线程的管理来提升性能是非常具有挑战性的工作，因为有效的线程管理需要实时的监测以及对系统资源的管理和使用。为了解决这个问题，苹果公司引入了 Grand Central Dispatch(GCD，多线程处理)机制。GCD 对由线程抽象出来的队列进行管理。队列可以并发操作或串行操作，并且可以在系统级别自动处理线程管理和优化的问题。

本章介绍几种对长线任务进行后台处理的方法，并分析它们的优缺点。

18.1 示例程序

本章的示例程序 LongRunningTasks 将演示一个运行在主线程上的普通长线任务，之后使用一些技术在主线程之外处理同样的长线任务。这个普通的长线任务包含 5 个循环，每次以一定的时延添加 10 个元素到数组中，之后在表视图中显示它们。示例程序中有一个表视图，用于显示可用方法的列表。选择一个方法会为之打开一个表视图。表视图具有 5 个初始元素，当试图滑动界面时可以很清楚地看出程序是否因为长线任务的运行影响到主线程的处理。长线任务之后会按照 10 个一组的方法创建 50 个元素，并显示在表视图中。任务完成的消息会通知给表视图，以更新主线程上的 UI，然后新的元素就出现了。

示例程序(如图 18-1 所示)用到的技术如下：

- performSelectorInBackground:withObject：这是让代码在主线程之外运行的最简单方法，当任务很简单且需求明确时这个方法的效果很好。使用这个方法，系统不会承担有关任务任何额外的管理工作，所以它最适用于非循环的任务。

- NSOperationQueue：这个方法稍微有点复杂，它提供了一些额外的控制功能，比如串行运行、并发运行或者按照任务间的依赖关系运行。队列操作最适用于重复任务和定义明确的异步任务，比如网络调用和解析。
- GCD 队列：这是让代码在主线程外运行所使用的最底层的方法，其灵活性也是最好的。示例程序将会演示使用 GCD 队列以串行和并发的方式运行任务。GCD 的使用范围很广，从实现后台程序和主队列间的通信到为列表中的每一个元素快速执行代码段都可以使用，还可以处理大型、重复的异步任务。

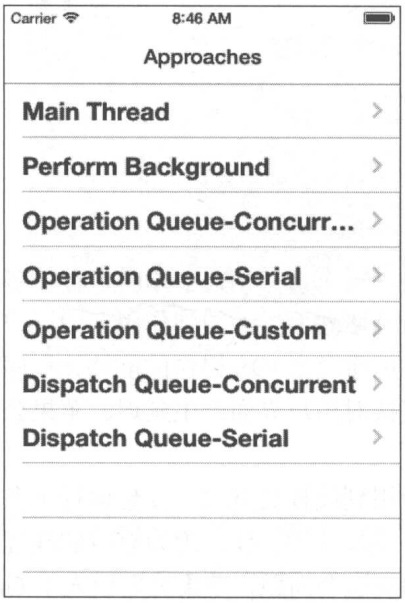


图 18-1　示例程序：长线任务方法列表

18.2　队列介绍

有关并发处理的各种术语会让人觉得很困惑。线程就是其中一个常用术语；在 iOS 应用的上下文中，一个线程就是一个标准的 POSIX 线程。从技术上讲，一个线程就是在进程(一个应用可以看作一个进程)中的一组可以独立处理的指令，一个进程可以包含多个线程，它们共享内存和资源。因为线程的功能是独立的，所以可以通过将线程分开工作来获得更快的运行速度。当多个线程需要访问同样的资源或数据时，有可能出现问题。所有 iOS 应用都具有一个主线程用于处理运行环路和更新 UI 界面。应用要保持对用户交互的响应，主线程的任务必须是可以在六十分之一秒内完成的任务。

队列是苹果公司在 Grand Central Dispatch 中提出的用于描述一种上下文的术语。队列是由 GCD 管理的一组需要执行的任务。根据所处当前系统的情况，GCD 会动态确定队列中用来执行的线程个数。主队列是一个由 GCD 管理的特殊队列，它和主线程相关联。所以当你在主队列运行一个任务时，GCD 也会在主线程上执行该任务。

人们会经常认为线程和队列这两个概念是可以互换的，一定要记住队列就是一组被管理的线程，而“主”的概念只是对于处理主运行环路和 UI 的线程而言的。

18.3　在主线程上运行

运行示例程序并选择 Main Thread 方法。注意，表视图中的 5 个初始元素都是可见的，不过它们不能滑动，当额外的元素被添加进来时 UI 完全没有响应。当额外的元素添加时 UI 是冻结的，通过在运行应用时查看输出日志和调试控制台就可以看出。冻结的 UI 显然是糟糕的用户体验，并且遗憾的是，在应用中很容易找到类似的情况。想要知道为什么会出现这种情况，请查看 ICFMainThreadLongRunningTaskViewController 类。首先，设置保存显示数据的数组并完成其初始化。

```
-(void)viewDidLoad
{
    [super viewDidLoad];

    self.displayItems =
    ➥[[NSMutableArray alloc] initWithCapacity:45];

    [self.displayItems addObject:@[@"Item Initial-1",
    ➥@"Item Initial-2",@"Item Initial-3",
    ➥@"Item Initial-4",@"Item Initial-5"]];
}
```

设置好初始数据且视图可见之后，长线任务就开始了：

```
-(void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];

    for(int i=1; i<=5; i++)
    {
        NSNumber *iteration = [NSNumber numberWithInt:i];
        [self performLongRunningTaskForIteration:iteration];
    }
}
```

应用会调用 performLongRunningTaskForIteration:方法 5 次来设置额外的表数据。还没有出现任何减慢主线程的动作。查看 performLongRunningTaskForIteration:方法以了解主线程此时挂起了哪些任务。长线任务的目的是向数组中添加 10 个元素，即添加到 displayItems 数组中，该数组作为表视图的数据源。

```
-(void)performLongRunningTaskForIteration:(NSNumber *)iterationNumber
{
    NSMutableArray *newArray =
    ➥[[NSMutableArray alloc] initWithCapacity:10];

    for(int i=1; i<=10; i++)
    {

        [newArray addObject:
```

```
            [NSString stringWithFormat:@"Item %@-%d",
            ➥iterationNumber,i]];

            [NSThread sleepForTimeInterval:.1];

            NSLog(@"Main Added %@-%d",iterationNumber,i);
        }

        [self.displayItems addObject:newArray];
        [self.tableView reloadData];
    }
```

由于主线程负责保持用户界面的更新，只有超过 1/60 秒的任务才会导致可以感觉到的延迟。这种情况下，注意，每次循环都对 NSThread 调用一个名为 sleepForTimeInterval:的方法。显然，对于一般的应用这么做似乎并没有什么意义，不过它可以很清楚地显示出哪个方法执行的时间过长了，阻塞主线程可能会导致严重的性能问题。

注意

找到耗时比较久的方法在本例中还是比较容易的。查看第 26 章“调试和工具”中介绍的技术可以发现性能问题出现的地方。

在这种情况下，调用 sleepForTimeInterval:方法会很快且经常地阻塞主线程，即使 for 循环完成后，在所有 performLongRunningTaskForIteration:方法没有执行完之前，主线程仍然没有足够的时间更新 UI。

18.4 在后台运行

运行示例程序，选择名为 Perform Background 的行。注意，在最初可见的表视图中一共有 5 个元素，在长线任务执行时也可以进行滑动(在滑动表视图时可以查看调试控制台来确认长线任务是否正在执行)。任务完成后，额外的行就可见了。

这个方法设置初始数据的方式完全和 Main Thread 方法一样。查看示例程序源代码的 ICFPerformBackgroundViewController 类以了解具体的初始化方法。在初始数据设置好且视图可见之后，长线任务就开始了；同时也是在这里设置任务在后台运行。

```
-(void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];

    SEL taskSelector =
    ➥@selector(performLongRunningTaskForIteration:);

    for(int i=1; i<=5; i++)
    {

        NSNumber *iteration = @(i);
```

```
        [self performSelectorInBackground:taskSelector
                               withObject:iteration];
    }
}
```

从代码中可以看到，使用后台执行的方法对 selector 进行设置。NSObject 定义的方法 performSelectorInBackground:withObject:，需要传递一个 Objective-C 对象作为 withObject:的参数。这个方法将会生成一个新的线程，根据传入的参数在新线程上执行该方法，并立即返回调用线程。这个新的线程由开发者负责管理，所以频繁调用这个方法完全有可能创建许多新的线程而使系统压力增大。如果测试发现有错误出现，可以使用操作队列或调度队列(本章后面会有介绍)在任务执行过程中进行更为准确的控制，以便更好地管理系统资源。

方法 performLongRunningTaskForIteration:执行任务的方法同 Main Thread 方法中的一样，不过这里不再是直接向 displayItems 中添加 newArray，而是使用 NSObject 的方法 performSelectorOnMainThread:withObject:waitUntilDone:调用 updateTableData:方法。使用这个方法的原因有两个：首先，包含在表视图中的 UIKit 对象只有在主线程上才能更新 UI 界面；其次，displayItems 属性被声明为 nonatomic 类型，这意味着生成的 getter 和 setter 方法并不是线程安全的。要“修复”这个问题，可以将 displayItems 声明为 atomic 类型，这样就需要在数组更新前将其锁定，这会增加性能上的开销。如果属性在主线程上更新，就不需要锁定了。

```
-(void)performLongRunningTaskForIteration:(NSNumber *)iterationNumber
{
    NSMutableArray *newArray =
    [[NSMutableArray alloc] initWithCapacity:10];

    for(int i=1; i<=10; i++)
    {

        [newArray addObject:
        ➥[NSString stringWithFormat:@"Item %@-%d",
        ➥iterationNumber,i]];

        [NSThread sleepForTimeInterval:.1];

        NSLog(@"Background Added %@-%d",iterationNumber,i);
    }

    [self performSelectorOnMainThread:@selector(updateTableData:)
                           withObject:newArray
                        waitUntilDone:NO];
}
```

updateTableData:方法用于将新创建的元素添加到 displayItems 数组中并通知表视图重载和更新 UI 界面。

一个有意思的副作用是这些后添加的行的顺序是不确定的，即应用每次启动时都有可能不同。

```
10:51:09.324 LongRunningTasks[29382:15903] Background Added 3-1
10:51:09.324 LongRunningTasks[29382:16303] Background Added 5-1
10:51:09.324 LongRunningTasks[29382:15207] Background Added 1-1
10:51:09.324 LongRunningTasks[29382:15e03] Background Added 4-1
10:51:09.324 LongRunningTasks[29382:15107] Background Added 2-1
10:51:09.430 LongRunningTasks[29382:15207] Background Added 1-2
10:51:09.430 LongRunningTasks[29382:16303] Background Added 5-2
10:51:09.430 LongRunningTasks[29382:15e03] Background Added 4-2
10:51:09.430 LongRunningTasks[29382:15107] Background Added 2-2
10:51:09.430 LongRunningTasks[29382:15903] Background Added 3-2
...
```

这是一个事实，所以使用这种技术对于任务何时完成或者说任务执行的顺序都是无法确定的，因为任务都是在不同线程上执行的。如果操作的顺序不重要，就适合使用该技术；如果顺序重要，操作队列或调度队列就需要以串行的方式执行任务(后面我们会对这两种队列进行描述，分别在“串行操作”和“串行调度队列”两个小节中)。

18.5 在操作队列中运行

操作队列(NSOperationQueue)可以对一组任务进行管理或操作(NSOperation)。一个操作队列可以指定多个操作并发运行、可以挂起任务和重启、可以取消所有暂停的操作。操作可以是简单的方法调用、代码块或自定义的操作类。操作可以具有使其能够串行运行的依赖关系。操作和操作队列实际上由 Grand Central Dispatch 进行管理，并在调度队列中实现。

示例程序演示了使用操作队列的 3 种方法：并发操作、带有依赖关系的串行操作和支持取消的自定义操作。

18.5.1 并发操作

运行示例程序，选择 Operation Queue-Concurrent。表视图中显示了 5 个初始元素，在长线任务执行时它们仍然可以滑动(在滑动表视图时可以查看调试控制台来确认长线任务是否正在执行)。任务完成后，额外的行就可见了。

查看示例程序源代码的 ICFOperationQueueConcurrentViewController 类以了解这个方法是如何设置的。在添加操作前，需要在 viewDidLoad:方法中设置操作队列：

```
self.processingQueue = [[NSOperationQueuealloc] init];
```

这个方法设置初始数据的方式同 Main Thread 方法一样。在初始数据设置完且视图可见之后，长线任务作为一个 NSInvocationOperation 实例被添加到操作队列中：

```
-(void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];

    SEL taskSelector =
    ➥@selector(performLongRunningTaskForIteration:);
```

```
    for(int i=1; i<=5; i++)
    {

        NSNumber *iteration = @(i);

        NSInvocationOperation *operation =
        ➥[[NSInvocationOperation alloc] initWithTarget:self
        ➥selector:taskSelector object:iteration];

        [operation setCompletionBlock:^{
            NSLog(@"Operation #%d completed.",i);
        }];

        [self.processingQueue addOperation:operation];
    }
}
```

每个操作都被分配了一个 completion block，操作处理结束时会运行这个代码块。performLongRunningTaskForIteration:方法所执行的任务完全同 Perform Background 方式的一样，实际上，并发操作上用到的这个方法并没有变化。updateTableData:方法也没有变化。结果也和 Perform Background 方式类似，元素的添加也没有确定的顺序。

```
21:00:16.165 LongRunningTasks[31009] OpQ Concurrent Added 1-1
21:00:16.165 LongRunningTasks[31009] OpQ Concurrent Added 3-1
21:00:16.165 LongRunningTasks[31009] OpQ Concurrent Added 4-1
21:00:16.165 LongRunningTasks[31009] OpQ Concurrent Added 2-1
21:00:16.165 LongRunningTasks[31009] OpQ Concurrent Added 5-1
...
21:00:17.107 LongRunningTasks[31009] Operation #4 completed.
21:00:17.108 LongRunningTasks[31009] Operation #2 completed.
21:00:17.107 LongRunningTasks[31009] Operation #5 completed.
21:00:17.108 LongRunningTasks[31009] Operation #3 completed.
21:00:17.109 LongRunningTasks[31009] Operation #1 completed.
```

这里主要的区别是 NSOperationQueue 现在负责管理线程，并且只有当队列中的线程到达默认最大并发操作时才进行处理。当应用中有很多不同的竞争任务同时出现时这一点很重要，并且需要对其进行管理以避免系统超负荷。

注意

一个操作队列的默认最大并发量是根据现实中的系统动态确定的，会基于当前系统的负载而变化。也可以为操作队列指定最大并发数量，这种情况下只有当操作并发达到指定的数量时队列才会处理。

18.5.2　串行操作

打开示例程序并选择名为 Operation Queue-Serial 的行。表视图中显示了 5 个初始元素，在长线任务执行时它们仍然可以滑动(在滑动表视图时可以查看调试控制台来确认长线任务

是否正在执行)。任务完成后，额外的行就可见了。

初始数据和操作队列的设置同 Operation Queue-Concurrent 方式一样。要让操作进程按照正确的顺序串行执行，需要对其设置依赖关系。要完成这个任务，viewDidAppear:方法会添加一个数组，用于保存新创建的操作，并使用 NSInvocationOperation(prevOperation)跟踪之前创建的操作。

```
NSMutableArray *operationsToAdd = [[NSMutableArrayalloc ]init ];

NSInvocationOperation *prevOperation = nil ;
```

当创建操作时，该方法会跟踪之前创建的操作。新创建的操作会将一个依赖关系添加到之前的操作，直到前面的操作完成时才运行新的操作。新操作被添加到操作对象数组，然后被添加到队列中。

```
for(int i=1; i<=5; i++)
{

    NSNumber *iteration = @(i);

    NSInvocationOperation *operation =
    ➥[[NSInvocationOperation alloc] initWithTarget:self
    ➥selector:taskSelector object:iteration];

    if(prevOperation)
    {
        [operation addDependency:prevOperation];
    }

    [operationsToAdd addObject:operation];

    prevOperation = operation;
}
```

当所有的操作都创建完成并将它们添加到数组后，就会将它们添加到队列中。由于操作一旦被加到操作队列中就会立即开始执行，因此应该一次添加所有操作，这样队列就可以明确操作间的依赖关系。

```
for(NSInvocationOperation *operation in operationsToAdd)
{
    [self.processingQueue addOperation:operation];
}
```

操作队列会对添加的操作和依赖关系进行分析，并确定执行它们的最优顺序。观察调试控制台可以看到，操作按照正确的串行顺序执行。

```
16:51:45.216 LongRunningTasks[29554:15507] OpQ Serial Added 1-1
16:51:45.318 LongRunningTasks[29554:15507] OpQ Serial Added 1-2
16:51:45.420 LongRunningTasks[29554:15507] OpQ Serial Added 1-3
16:51:45.522 LongRunningTasks[29554:15507] OpQ Serial Added 1-4
```

```
16:51:45.625 LongRunningTasks[29554:15507] OpQ Serial Added 1-5
16:51:45.728 LongRunningTasks[29554:15507] OpQ Serial Added 1-6
16:51:45.830 LongRunningTasks[29554:15507] OpQ Serial Added 1-7
16:51:45.931 LongRunningTasks[29554:15507] OpQ Serial Added 1-8
16:51:46.034 LongRunningTasks[29554:15507] OpQ Serial Added 1-9
16:51:46.137 LongRunningTasks[29554:15507] OpQ Serial Added 1-10
16:51:46.246 LongRunningTasks[29554:14e0b] OpQ Serial Added 2-1
16:51:46.349 LongRunningTasks[29554:14e0b] OpQ Serial Added 2-2
16:51:46.452 LongRunningTasks[29554:14e0b] OpQ Serial Added 2-3
16:51:46.554 LongRunningTasks[29554:14e0b] OpQ Serial Added 2-4
16:51:46.657 LongRunningTasks[29554:14e0b] OpQ Serial Added 2-5
16:51:46.765 LongRunningTasks[29554:14e0b] OpQ Serial Added 2-6
...
```

串行方式增加了完成所有任务所消耗的时间，不过它能够成功确保任务按照正确的顺序执行。

18.5.3　取消操作

回到示例程序中，选择 Operation Queue-Concurrent 方式。当操作运行时快速点击表视图上方的 Cancel 按钮。注意，点击 Cancel 按钮后并没有显示任何内容，所有操作将会正常结束。当点击 Cancel 按钮时，操作队列得到命令，取消所有尚未完成的操作。

```
-(IBAction)cancelButtonTouched:(id)sender
{
    [self.processingQueue cancelAllOperations];
}
```

这并不是我们所希望的结果，因为此时 Cancelled 还只是操作对象上的一个标记——当取消命令发起时一定要设置好表示操作如何进行的逻辑代码。调用 dutifully 负责对所有尚未完成的操作设置标记，由于它们不会在运行时检查自己的取消状态，因此它们会继续执行，直到完成任务。

要正确处理取消动作，必须创建一个 NSOperation 子类，或者必须检查 NSBlockOperation 实例的弱引用，如下所示：

```
NSBlockOperation *blockOperation =
➥[[NSBlockOperation alloc] init];

__weak NSBlockOperation *blockOperationRef = blockOperation;
[blockOperation addExecutionBlock:^{
    if(![blockOperationRef isCancelled])
    {
        NSLog(@"...not canceled, execute logic here");
    }
}];
```

在下一节中，将创建一个带有取消处理功能的自定义 NSOperation 子类。

18.5.4 自定义操作

回到示例程序，选择 Operation Queue-Custom 方式。表视图中显示了 5 个初始元素，在长线任务执行时它们仍然可以滑动(在滑动表视图时可以查看调试控制台来确认长线任务是否正在执行)。在操作完成前快速点击表视图上方的 Cancel 按钮。注意，这次任务立即停止了。

初始数据和操作队列的设置几乎同 Operation QueueSerial 方式一样。唯一的区别是这里用到的自定义 NSOperation 子类称作 ICFCustomOperation：

```
-(void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];

    NSMutableArray *operationsToAdd =
    ➥[[NSMutableArray alloc] init];

    ICFCustomOperation *prevOperation = nil;
    for(int i=1; i<=5; i++)
    {

        NSNumber *iteration = [NSNumber numberWithInt:i];

        ICFCustomOperation *operation =
        ➥[[ICFCustomOperation alloc] initWithIteration:iteration
                                        andDelegate:self];

        if(prevOperation)
        {
            [operation addDependency:prevOperation];
        }

        [operationsToAdd addObject:operation];

        prevOperation = operation;
    }

    for(ICFCustomOperation *operation in operationsToAdd)
    {
        [self.processingQueue addOperation:operation];
    }
}
```

ICFCustomOperation 已经被声明为 NSOperation 子类，这里还需要同时声明一个协议，这样就可以通知委托函数，任务处理已完成并返回响应结果。

```
@protocol ICFCustomOperationDelegate <NSObject>

-(void)updateTableWithData:(NSArray *)moreData;
```

```
@end
```

NSOperation 子类需要实现 main 方法，操作的处理逻辑代码应该写在该方法中，如下所示：

```
-(void)main
{
    NSMutableArray *newArray =
    ➥[[NSMutableArray alloc] initWithCapacity:10];

    for(int i=1; i<=10; i++)
    {

        if([self isCancelled])
        {
            break;
        }

        [newArray addObject:
        ➥[NSString stringWithFormat:@"Item %@-%d",
        ➥self.iteration,i]];

        [NSThread sleepForTimeInterval:.1];
        NSLog(@"OpQ Custom Added %@-%d",self.iteration,i);
    }

    [self.delegate updateTableWithData:newArray];
}
```

在 for 循环开始时，检查取消状态：

```
if([self isCancelled])
{
    break;
}
```

这个检查可以让操作对取消请求做到立即响应。当设计一个自定义操作时，要谨慎考虑取消操作应该如何执行，以及是否需要任何回滚逻辑。

正确处理取消动作不仅对创建一个自定义操作子类有帮助，同时也是一个将复杂逻辑进行封装的有效方法，这样它就可以很好地在操作队列中运行了。

18.6　在调度队列中运行

调度队列由 Grand Central Dispatch 提供，用于在受控环境内执行一段代码。GCD 被设计为最大化实现并发处理，同时基于系统的状态充分利用多核处理器的性能对大量动态分布的队列进行管理。

GCD 提供了 3 种类型的队列，分别是主队列、并发队列和串行队列。主队列是由系统创建的特殊队列，同应用的主线程绑定。在 iOS 中，可以使用一些全局并发的队列，分为高、

普通、低和后台运行这 4 个优先级队列。私有的并发和串行队列可以由应用创建并一定要像其他应用资源一样被系统管理。示例程序给出了使用并发和串行 GCD 队列的例子，以及如何在这些队列中和主队列进行交互。

注意

从 iOS 6 开始，创建的调度队列都由 ARC 管理，不需要对它们进行 retain 和 release 操作。

18.6.1 并发调度队列

打开示例程序，选择 Dispatch Queue-Concurrent 方式。表视图中显示了 5 个初始元素，在长线任务执行时它们仍然可以滑动(在滑动表视图时可以查看调试控制台来确认长线任务是否正在执行)。任务完成后，额外的行就可见了。

注意，这种方式的完成速度明显比前面任何一种方式都要快。

要在 viewDidAppear:方法中开始一个长线任务，应用需要获得一个对高优先级并发调度队列的引用：

```
dispatch_queue_t workQueue =
➥dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0);
```

使用 dispatch_get_global_queue 可以访问 3 个全局且由系统维护的并发调度队列。对这些队列的引用不需要进行 retain 和 release 操作。当队列引用完成后，可以在代码段内编写任务的处理逻辑。

```
for(int i=1; i<=5; i++)
{

    NSNumber *iteration = @(i);

    dispatch_async(workQueue, ^{
        [self performLongRunningTaskForIteration:iteration];
    });
}
```

使用 dispatch_async 表示逻辑是异步执行的。如果这样，这段代码内的工作将会提交给队列，并且对其调用会立即返回，不会阻塞主线程。该代码块还可以使用 dispatch_sync 同步提交给队列，这样在代码段内的逻辑处理完成前调用线程只能在那里等待。

在 performLongRunningTaskForIteration:方法中，需要强调与前面方式相比不同的几点。用于跟踪所创建元素的 newArray 对象需要一个 _block 修饰符，这样代码块才能更新它。

```
_block NSMutableArray *newArray =
➥[[NSMutableArrayalloc] initWithCapacity:10];
```

之后，该方法会得到一个对低优先级并发调度队列的引用。

```
dispatch_queue_tdetailQueue =
➥dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_LOW,0);
```

之后，低优先级调度队列会用在强大的 GCD 技术上，即同步处理整个枚举。

```
dispatch_apply(10, detailQueue, ^(size_t i)
{
    [NSThread sleepForTimeInterval:.1];

    [newArray addObject:[NSString stringWithFormat:
    ➥@"Item %@-%zu",iterationNumber,i+1]];

    NSLog(@"Dispatch Q Added %@-%zu",iterationNumber,i+1);
});
```

使用 dispatch_apply，所需的参数包括迭代次数、关于调度队列的引用、用于表示迭代正在进行的变量(一定要为 size_t 类型)和每次迭代需要用到的逻辑代码块。GCD 会使用可能的迭代将队列填满，在系统约束的范围内它们的执行会尽可能同步。这一技术使得该方式在任务执行方面比其他方式都快，如果任务的顺序不重要，这种方式就非常高效。

注意

方法通过高级别的抽象后，用在集合类中也可以起到同样的效果。比如，NSArray 具有一个名为 enumerateObjectsWithOptions:usingBlock:的方法。这个方法可以按数组序列遍历对象，也可以反向遍历或并发遍历。

迭代完成并且由新元素组成的数组创建好之后，dispatch_async 方法需要通知 UI 更新表视图。

```
dispatch_async(dispatch_get_main_queue(), ^{
    [self updateTableData:newArray];
});
```

dispatch_async 调用使用函数 dispatch_get_main_queue 来访问主队列。注意，该技术可以在任何地方访问主队列，并且可以很容易地更新 UI，报告长线任务当前的状态。

18.6.2　串行调度队列

运行示例程序，选择 Dispatch Queue-Serial 方式。表视图中显示了 5 个初始元素，在长线任务执行时它们仍然可以滑动(在滑动表视图时可以查看调试控制台来确认长线任务是否正在执行)。任务完成后，额外的行就可见了。这种方式没有并发调度队列方式那么快，不过可以按照任务添加到队列中的顺序来执行它们。

要在 viewDidAppear:中开始一个长线任务，应用需要创建一个串行调度队列：

```
dispatch_queue_t workQueue =
➥dispatch_queue_create("com.icf.serialqueue", NULL);
```

可以用异步的方法向串行队列中添加实际工作的代码段：

```
for(int i=1; i<=5; i++)
{
```

```
    NSNumber *iteration = @(i);

    dispatch_async(workQueue, ^{
        [self performLongRunningTaskForIteration:iteration];
    });
}
```

performLongRunningTaskForIteration:方法执行的任务同主线程非常类似，可以后台运行和使用并发操作队列方法；不过这个方法使用 dispatch_async 对主队列调用 updateTableData:方法。

```
-(void)performLongRunningTaskForIteration:(id)iteration
{
    NSNumber *iterationNumber = (NSNumber *)iteration;

    NSMutableArray *newArray =
    ➥[[NSMutableArray alloc] initWithCapacity:10];

    for(int i=1; i<=10; i++)
    {

        [newArray addObject:[NSString stringWithFormat:
        ➥@"Item %@-%d",iterationNumber,i]];

        [NSThread sleepForTimeInterval:.1];
        NSLog(@"DispQ Serial Added %@-%d",iterationNumber,i);
    }

    dispatch_async(dispatch_get_main_queue(), ^{
        [self updateTableData:newArray];
    });
}
```

串行调度队列将会按照任务添加到队列中的顺序来执行长线任务：先进先出。调试控制台将会显示按正确顺序执行的这些操作信息。

```
20:41:00.340 LongRunningTasks[30650:] DispQ Serial Added 1-1
20:41:00.444 LongRunningTasks[30650:] DispQ Serial Added 1-2
20:41:00.546 LongRunningTasks[30650:] DispQ Serial Added 1-3
20:41:00.648 LongRunningTasks[30650:] DispQ Serial Added 1-4
20:41:00.751 LongRunningTasks[30650:] DispQ Serial Added 1-5
20:41:00.852 LongRunningTasks[30650:] DispQ Serial Added 1-6
20:41:00.955 LongRunningTasks[30650:] DispQ Serial Added 1-7
20:41:01.056 LongRunningTasks[30650:] DispQ Serial Added 1-8
20:41:01.158 LongRunningTasks[30650:] DispQ Serial Added 1-9
20:41:01.261 LongRunningTasks[30650:] DispQ Serial Added 1-10
20:41:01.363 LongRunningTasks[30650:] DispQ Serial Added 2-1
20:41:01.465 LongRunningTasks[30650:] DispQ Serial Added 2-2
20:41:01.568 LongRunningTasks[30650:] DispQ Serial Added 2-3
20:41:01.671 LongRunningTasks[30650:] DispQ Serial Added 2-4
```

```
20:41:01.772 LongRunningTasks[30650:] DispQ Serial Added 2-5
...
```

同样，串行方法增加了完成所有任务所需的时间开销，不过可以很好地确保任务按正确的顺序执行。使用调度队列串行执行任务要比管理有依赖关系的操作队列容易，不过没有提供同样高级别的管理选项。

18.7　小结

本章介绍了几种在不影响 UI 界面的前提下处理长线任务的技术，包括 performSelectorInBackground:withObject:、操作队列和 GCD 队列。

对 NSObject 使用 performSelectorInBackground:withObject:函数让任务在后台执行是最简单的方法，不过有关这个方法的支持和管理则很有限。

操作队列可以并发或串行地处理任务，这通过使用方法调用、代码块或自定义操作类来实现。操作队列可以指定最大并发操作数，操作还可以挂起和重启，所有尚未完成的操作都可以被取消。操作队列可以处理自定义操作类。操作队列通过 GCD 来实现。

调度队列也可以并发或串行地处理任务。一共有 3 个全局并发队列，应用还可以创建自己的串行队列。调度队列能够接受要执行的代码块，可以同步或异步执行代码块。

没有一种技术是“最好”的，因为每种技术都有自身的优势和劣势，所以要通过充分的测试才知道哪一种技术是最合适的。

第19章

使用 Keychain 和 TouchID 保护并访问数据

对高敏感度用户数据的保护是移动开发中非常关键且经常被忽视的一个步骤。在技术发布会上经常会爆出大公司使用明文对密码或信用信息进行保存的恼人事件。用户应该相信开发者对待他们的敏感信息会非常小心并会得到应有的尊重，这包括对远程和本地的信息副本进行加密以阻止未授权的访问。开发者对待用户数据应该像对待自己的机密信息一样，这是每个开发者的责任。

苹果公司一直都在提供一种称作 Keychain 的安全框架，用于在 iOS 上保存加密的信息。Keychain 相对标准应用的数据安全机制还有一些其他的优势。保存在 Keychain 中的信息即使当应用从设备上卸载后也不会消失，Keychain 信息甚至可以在同一开发者的多个应用间共享。

本章会介绍如何使用苹果官方的 KeychainItemWrapper 类(1.2 版本)来保存和获取敏感信息。虽然从头开始编写一个 Keychain 封装器也是可以接受的并且偶尔需要这样做，不过一般情况下还是使用苹果提供的库，这样可以节省大量时间并且这个库几乎可以提供所有需要的功能。所以本章不介绍创建一个自定义 Keychain 封装器的方法，而是讲解如何使用苹果提供的代码将 Keychains 快速添加到一个 iOS 应用中。Keychains 的交互有些复杂，一点小的错误都可能导致数据不安全，不过使用苹果提供的类可以最小化这一风险。

提示

苹果的 KeychainItemWrapper 类的最新版本可以在如下网站找到：http://developer.apple.com/library/ios/#samplecode/GenericKeychain/Listings/ClassesKeychainItemWrapper_m.html。

非常重要的一点是，把信息安全地保存在磁盘上仅仅是全部安全范畴中的一小部分。诸如网络传输数据、远程存储和强制密码机制这些其他因素，对于提供一个全方位安全的应用起着至关重要的作用。

19.1 示例程序

Keychain 示例程序只有一个视图，主要功能就是保护用户信用卡账号和相关的用户信息，比如名字和有效期。为访问这些信息，用户在第一次启动时就设置了一个 PIN。PIN 和信用卡信息都由 Keychain 保护。

注意

从 iOS 8 版本开始，Keychain 无法在 iOS 模拟器中运行。本章所使用的苹果公司封装类为正确模拟 Keychain 功能做出了非常大的努力。此外，由于在模拟器上执行的代码没有签名，因此很重要的一点是，要记得此时对于应用访问 Keychain 元素是没有做任何限制的。强烈建议有关 Keychain 的开发在使用模拟器进行测试之后，还要彻底地在真机设备上进行调试。

示例程序本身很简单，它包含 4 个文本框和 1 个按钮。示例程序的主要代码并不直接和 Keychain 布局信息相关。

注意

从设备上卸载程序并不能移除应用的 Keychain，只会让调试变得更加困难。模拟器具有一个 Reset Contents and Settings 选项，用于彻底删除 Keychain。所以建议大家如果应用受困于无法返回一个干净的状态，就对它进行优化，先不要在真机上进行调试。

19.2 创建和使用 Keychain

Keychain 是 Security.framework 的一部分，在 iOS 最初发布 SDK 时就存在。Keychain 源自于 Mac OS X，最初在 OS X 10.2 中引入。不过 Keychain 的历史早在 OS 8.6 中就开始了。Keychain 最初是为苹果公司的邮件系统 PowerTalk 开发的，这就使其成为 iOS 中最老的框架之一。

Keychain 可以用于保护少量数据的安全，比如密码、键、证书和记录等。不过如果应用试图保存大量信息，比如编码图片或视频，通常使用第三方加密库要比 Keychain 更合适。Core Data 也可以提供加密功能，如果应用是基于 Core Data 的，则可以考虑使用 Core Data 自己的方法。

在正式使用 Keychain 之前，需要向项目中添加 Security.framework 框架，还要在所有包含直接访问 Keychain 的方法和函数的文件前导入<Security/Security.h>。

19.2.1 创建新的 KeychainItemWrapper

iOS Keychain 会根据请求基于应用代码签名进行解锁。因为这里没有类似 OS X 系统上的系统级别的密码，所以保护数据就需要一个额外的步骤。因为带有 Keychain 数据的应用控件只能访问真正安全的信息，所以应用本身需要设置密码。示例程序在进入系统时使用 PIN 来实现这一步骤。

当应用第一次启动时，用户需要输入一个新的 PIN 码，以后会重复使用这个 PIN 码。要安全保存这个 PIN，需要创建一个新的 KeychainItemWrapper 对象。

```
pinWrapper = [[KeychainItemWrapper
➥alloc]initWithIdentifier:@"com.ICF.Keychain.pin" accessGroup:nil];
```

创建新的 KeychainItemWrapper 需要为其设置两个特性。第一个特性是 Keychain 元素的标识符。这里建议使用反向 DNS 的命名方法，比如 com.company.app.id。例子中的 accessGroup 被设置为 nil，accessGroup 参数用于在多个应用间共享 Keychains。请参阅“在应用间共享 Keychains”一节以了解更多关于 accessGroup 的信息。

创建新的 KeychainItemWrapper 所需的第二个特性是 kSecAttrAccessible，用于控制数据解锁的时间。在示例程序中当设备解锁时数据就处于可用状态，以便为锁定设备保护数据。对于这个参数有许多可能的选项进行设置，如表 19-1 中所示。

```
[pinWrapper setObject:kSecAttrAccessibleWhenUnlocked forKey:
➥ (id)kSecAttrAccessible];
```

表 19-1　kSecAttrAccessible 可用的所有常量和相关描述

常　量　名	描　　述
kSecAttrAccessibleAfterFirstUnlock	在重启后设备第一次解锁时 Keychain 会解锁数据，直到设备重启后元素才会解锁。建议为后台任务使用这个设置。当用户升级时这些元素将会移到新设备
kSecAttrAccessibleAfterFirst UnlockThisDeviceOnly	这个设置将会复制 kSecAttrAccessibleAfterFirstUnlock 的功能描述，不过如果用户升级设备或从备份中恢复，元素不会转移到新的设备
kSecAttrAccessibleAlways	无论设备是否处于解锁状态，元素都是解锁的。出于对信息的保护，不建议使用这个设置。当用户升级设备时元素会转移
kSecAttrAccessibleAlwaysThis DeviceOnly	这个设置会复制 kSecAttrAccessibleAlways 的功能描述，不过如果用户升级设备或从备份中恢复，元素不会转移到新的设备
kSecAttrAccessibleWhenUnlocked	只有在用户解锁设备时才解锁数据。应用处于前台运行且需要对数据进行保护时，建议使用这个设置。这也是 kSecAttr-Accessible 最常用的一个设置。当用户升级设备时元素也会转移
kSecAttrAccessibleWhenUnlocked ThisDeviceOnly	这个设置会复制 kSecAttrAccessibleWhenUnlocked 的功能描述，不过如果用户升级设备或从备份中恢复，元素不会转移到新的设备

应用现在知道 Keychain 的标识符的含义及对应的安全等级。不过在数据开始保存前还需要设置一个额外的参数 kSecAttrService，该参数用于为 PIN 密码对保存用户名。PIN 码没有相关联的密码，示例程序使用的是 pinIdentifer。虽然经常用到 Keychain 而忽略 kSecAttrService，不过对 kSecAttrService 进行设置还有助于避免许多难以处理的错误，同时我们也建议这样做。

```
[pinWrapper setObject:@"pinIdentifer" forKey:(id)kSecAttrAccount];
```

19.2.2 保存和获取 PIN

使用前面小节介绍的方法配置新的 KeychainItemWrapper 后，就可以开始用它保存数据了。在 Keychain 中保存信息同在字典里保存数据类似。示例程序首先通过检查确保两个 PIN 文本框的内容相符，之后令前面小节创建的 pinWrapper 对象调用 setObject:方法。使用 kSecValueData 作为标识符参数。此参数的信息会在“Keychain 特性键”一节中深入介绍，不过现在使用这个参数很重要。

```
if([pinField.text isEqualToString:pinFieldRepeat.text])
{
    [pinWrapper setObject:[pinField text] forKey:kSecValueData];
}
```

当一个新的值被保存到 Keychain 后，可以通过同样的方法获取它。在示例程序中要测试用户是否输入了正确的 PIN，使用下面的代码：

```
if([pinField.text isEqualToString:[pinWrapper
➥objectForKey:kSecValueData]])
```

当确认输入的 PIN 值同 Keychain 中保存的 PIN 值匹配后，用户就可以访问应用的下一个会话的功能了，“保护字典对象”一节中会有描述。

19.2.3 Keychain 特性键

Keychain 元素的保存同 NSDictionary 对数据的保存类似，不过 Keychain 会用到和其相关联的键。与 NSDictionary 不同的是，一个 Keychain 不能使用任何随机字符串作为键值。每个 Keychain 都和一个 Keychain 类相关联；如果使用苹果的 KeychainItemWrapper，则默认使用 CFTypeRefkSecClassGenericPassword。还可以为其选择其他的参数：kSecClassInternetPassword、kSecClassCertificate、kSecClassKey 和 kSecClassIdentity。每个类都有不同的关联值。根据本章的计划我们使用 KeychainItemWrapper，所以把重点放在 kSecClassGenericPassword 的使用上。

kSecClassGenericPassword 在保存和访问数据方面包含 14 个可能的键，如表 19-2 所示。需要时刻记住的是，对于实现正确的功能，这些键都是可选的，不是必需的。

表 19-2 kSecClassGenericPassword 用到的 Keychain 特性键

特　性	描　述
kSecAttrAccessible	锁定行为的键，在表 19-1 中介绍过
kSecAttrAccessGroup	在新的 Keychain 初始化过程中看到的访问组，“在应用间共享 Keychain”一节中会有深入讨论
kSecAttrCreationDate	一个 CFDateRef，表示 Keychain 元素创建的数据
kSecAttrModificationDate	一个 CFDateRef，表示 Keychain 元素最后修改的数据
kSecAttrDescription	一个字符串，为 Keychain 提供用户可见的描述内容，比如 Twitter Password

(续表)

特　性	描　述
kSecAttrComment	一个字符串，表示关于 Keychain 元素的用户可编辑的注释
kSecAttrCreator	一个 4 位的 CFNumberRef，表示反映 Keychain 生成器代码的字符代码，比如 aCrt。如果自定义实现代码中需要使用，则其可以用于识别 Keychain 的原始生成器
kSecAttrType	一个 4 位的 CFNumberRef，表示反映 Keychain 元素类型的字符代码，比如 aTyp
kSecAttrLabel	一个字符串，表示 Keychain 元素的可见标签
kSecAttrIsInvisible	一个 CFBooleanRef，表示元素是否可见，kCFBooleanTrue 表示可见。
kSecAttrIsNegative	一个 CFBooleanRef，用于表示这个元素是否带有一个有效的密码。如果不希望保存密码而让用户每次登录时必须输入密码，这个参数就非常有用
kSecAttrAccount	“创建新的 KeychainItemWrapper”一节中讨论的账户名特性
kSecAttrService	一个 CFStringRef，用于和元素相关联的服务
kSecAttrGeneric	一个泛型特性键，用于保存用户定义的特性

19.2.4　保护字典对象

保护一个更加复杂的数据类型同前面小节介绍的保护 PIN 的方法一样，比如字典对象就属于相对复杂的数据类型。Keychain 封装器方法只允许存储字符串类型的数据。要保护字典对象，首先将其转换为字符串类型。示例代码选择的方法是，首先使用 NSJSONSerialization 类将字典对象保存为 JSON 字符串(参见第 9 章“JSON 的使用和解析”以了解更多相关内容)。

```
NSMutableDictionary *secureDataDict = [[[NSMutableDictionary alloc]
➥init] autorelease];

NSError *error = nil;

if(numberTextField.text)
    [secureDataDict setObject:numberTextField.text
    ➥forKey:@"numberTextField"];

if(expDateTextField.text)
    [secureDataDict setObject:expDateTextField.text
    ➥forKey:@"expDateTextField"];

if(CV2CodeTextField.text)
    [secureDataDict setObject:CV2CodeTextField.text
    ➥forKey:@"CV2CodeTextField"];

if(nameTextField.text)
```

```
    [secureDataDict setObject:nameTextField.text
    ➥forKey:@"nameTextField"];

NSData *rawData = [NSJSONSerialization
➥dataWithJSONObject:secureDataDict
          options:0
            error:&error];

if(error != nil)
{
    NSLog(@"An error occurred: %@", [error localizedDescription]);
}

NSString *dataString = [[NSString alloc] initWithData:rawData
➥encoding:NSUTF8StringEncoding];
```

当字典对象的值被转换为表示字典数据的字符串之后，可以使用前面小节中讨论的方法将其添加到 Keychain 中。

```
KeychainItemWrapper *secureDataKeychain = [[KeychainItemWrapper alloc]
➥initWithIdentifier:@"com.ICF.keychain.securedData" accessGroup:nil];

[secureDataKeychain setObject:@"secureDataIdentifer" forKey:
➥(id)kSecAttrAccount];

[secureDataKeychain setObject:kSecAttrAccessibleWhenUnlocked forKey:
➥(id)kSecAttrAccessible];

[secureDataKeychain setObject:dataString forKey:kSecValueData];
```

要取回字典格式的数据，反向执行所有的步骤。首先从 Keychain 中得到 NSString 对象，将其转换为 NSData 值。NSJSONSerialization 使用 NSData 来获取原始字典值。当字典对象重新创建好之后，则会生成显示用户信用卡的新文本框。

```
KeychainItemWrapper *secureDataKeychain = [[KeychainItemWrapper alloc]
➥initWithIdentifier:@"com.ICF.keychain.securedData" accessGroup:nil];

NSString *secureDataString = [secureDataKeychain
➥objectForKey:kSecValueData];

if([secureDataString length] != 0)
{
    NSData* data = [secureDataString
    ➥dataUsingEncoding:NSUTF8StringEncoding];

    NSError *error = nil;

    NSDictionary *secureDataDictionary = [NSJSONSerialization
    ➥JSONObjectWithData:data
```

```
    ➥options:NSJSONReadingMutableContainers
    ➥error:&error];

    if(error != nil)
    {
        NSLog(@"An error occurred: %@", [error localizedDescription]);
    }

    numberTextField.text = [secureDataDictionary
    ➥objectForKey:@"numberTextField"];

    expDateTextField.text = [secureDataDictionary
    ➥objectForKey:@"expDateTextField"];

    CV2CodeTextField.text = [secureDataDictionary
    ➥objectForKey:@"CV2CodeTextField"];

    nameTextField.text = [secureDataDictionary
    ➥objectForKey:@"nameTextField"];
}

else
{
    NSLog(@"No Keychain data stored yet");
}
```

19.2.5　重置 Keychain 元素

有时可能需要删除 Keychain 中的数据而不使用其他用户数据集进行替换。对需要重置的 Keychain 封装器，使用苹果提供的库中的 resetKeyChainItem 方法完成上述功能。

```
[pinWrapper resetKeychainItem];
```

19.2.6　在应用间共享 Keychain

同一个开发者发布的多个应用间可以共享 Keychain，或者在一些特殊的情况下也可以。对于在两个应用间共享 Keychain 数据所需的最重要的一点是这两个应用必须具有相同的 bundle seed。比如，两个应用的 bundle 标识符分别是 659823F3DC53.com.ICF.firstapp 和 659823F3DC53.com.ICF.secondapp。这些应用能够访问并修改彼此的 Keychain 数据。使用通配符 ID 的 Keychain 共享无法正常工作，虽然官方文档目前对这一方面还没有明确的说明。当新的应用被创建时，可以在开发者门户网站上配置 bundle seed。

当有两个应用共享同一个 bundle seed 时，每个应用都需要为 Keychain 访问组配置各自的 entitlement。Keychain 组在目标的 summary 页面上配置，如图 19-1 所示。

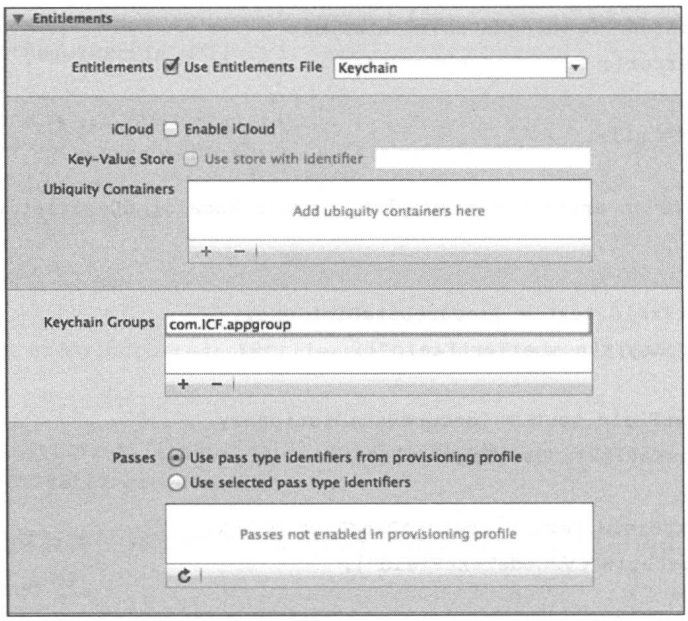


图 19-1 使用 Xcode 6 创建一个新的 Keychain

要访问共享的 Keychain，首先需要设置 Keychain 组。修改本章之前的 PIN 例子，如下所示：

```
[pinWrappersetObject:@"659823F3DC53.com.ICF.appgroup"
➥forKey:(id)kSecAttrAccessGroup];
```

注意

记住，在 Xcode 中设置访问组时，不需要指定 bundle seed。不过当设置 kSecAttrAccessGroup 属性时，需要指定 bundle seed 并且两个应用的 bundle seed 必须匹配。

在 KeychainItemWrapper 上设置好访问组后，就可以使用本章之前介绍的一般方法对其进行创建、修改和删除。

19.2.7 Keychain 错误代码

Keychain 会根据运行时遇到的具体错误返回一些专门的错误代码，这些可能遇到的错误及其代码如表 19-3 所示。

表 19-3 Keychain 错误代码

代　码	值	描　述
errSecSuccess	0	没有遇到错误
errSecUnimplemented	-4	功能或操作没有实现
errSecParam	-50	方法中传递的一个或多个参数无效

(续表)

代　码	值	描　述
errSecAllocate	-108	分配内存失败
errSecNotAvailable	-25291	没有可信的结果
errSecAuthFailed	-25293	授权/认证失败
errSecDuplicateItem	-25299	元素已经存在
errSecItemNotFound	-25300	元素未找到
errSecInteractionNotAllowed	-25308	无法同 Security Server 交互
errSecDecode	-26275	无法解码提供的数据

19.3 实现 Touch ID

iPhone 5S 是第一个提供硬件级别指纹识别功能的 iOS 设备。这一技术统称为 Touch ID，允许用户仅使用指纹就可以进行认证和购买。第三方开发者可以实现 Touch ID 来认证用户。一定要知道指纹数据本身保存在 A7 芯片上，不能在 Touch ID 之外访问。任何时候使用 Touch ID 时，一定要认识到并非所有用户都有一部支持指纹认证的设备，也并非所有用户都选择使用这种技术。

Touch ID 允许用户使用指纹进行认证、输入密码或取消认证动作。如果一个用户选择不使用指纹认证，根据苹果公司的 Review Guidelines 要求，应用必须提供一个回调方法。要开始在应用中使用 Touch ID，首先添加 Local Authentication Framework 框架，同时还必须导入<LocalAuthentication/LocalAuthentication.h>头。

创建一个新的本地 Authentication Context：

```
LAContext *myContext = [[LAContextalloc] init];
```

应用需要检查确保 Touch ID 存在并且设备是支持的，使用 canEvaluatePolicy:方法进行验证。如果应用支持 Touch ID 认证，则可以调用 evaluatePolicy:；否则，应用应该选择其他的认证方法。

```
NSError *authError = nil;
NSString *myReasonString = @"Human readable string for reason
➥access is being requested";

if([myContext canEvaluatePolicy:LAPolicyDeviceOwnerAuthenticationWithBiometrics
➥error:&authError])
{
   [myContext evaluatePolicy:LAPolicyDeviceOwnerAuthenticationWithBiometrics
➥localizedReason:myReasonString reply:^(BOOL succes, NSError *error)
{
      if(success)
      {
            // Authenticated successfully
```

```
        }
        else
        {
            // Authenticate failed
        }
    }];
}
else
{
    // Could not evaluate policy; check authError
}
```

如果认证失败，LAContext 可能返回许多可能的错误类型，如表 19-4 所示。

表 19-4 Touch ID 错误代码

代　码	描　述
LAErrorAuthenticationFailed	由于用户没有提供有效的证书导致的认证失败
LAErrorUserCancel	用户取消认证(比如点击 Cancel 按钮)
LAErrorUserFallback	用户点击了退格按钮来取消认证(输入密码)
LAErrorSystemCancel	系统取消认证(比如另一个应用被切换到前台)
LAErrorPasscodeNotSet	因为设备没有设置密码锁而无法启动认证
LAErrorTouchIDNotAvailable	设备不支持 Touch ID 而导致认证无法启动
LAErrorTouchIDNotEnrolled	因为 Touch ID 没有登记任何指纹而导致认证无法启动

19.4 小结

本章介绍了如何使用 Keychain 为应用保护少量的数据。示例程序演示了在启动时添加 PIN 访问密码的功能，还实现了存储和取回多条信用卡数据的功能。本章还引入了 Touch ID 的概念，iPhone 5S 及更新的版本都支持指纹识别功能。

Keychain 和数据安全是一个很大的专题，本章仅仅介绍了它的冰山一角。开发者社区也在寻找安全领域的专家，尤其是针对移动市场方面的专家。Keychain 是一个令人激动且很大的话题，现在它已经不再那么令人生畏了。多么希望本章中对数据安全的介绍能够让你意识到计算机数据安全的重要性，避免出现新闻中经常报道的因粗心的开发者丢失用户机密信息而导致的错误。

第20章

处理图片和过滤器

图片和图片的处理对于许多 iOS 应用来说都是非常核心的功能。从最基本的情况来看，图片是自定义用户界面的必要组成部分，从自定义视图背景到自定义按钮和视图元素都会用到。不仅如此，iOS 6 还加入了 Core Image 库函数，为用户提供的图片增添了更加丰富的定制功能。在 iOS 6 之前，图片操作需要使用 Quartz 或自定义 C 语言库函数提供的自定义图片处理代码。随着 Core Image 的推出，许多复杂的图片编辑功能一一在那些成功的应用上实现，不费吹灰之力就可以对图片进行复杂的操作，比如 Instagram 应用就是一个很好的例子。

本章介绍基本的图片处理方法，包括如何载入和显示一张图片，对于性能不同的设备如何对图片进行处理，以及一些基本的图片显示技术。还会介绍如何从设备的图片库或相册中获取图片。此外，本章还会演示并介绍如何使用 Core Image 库对用户提供的图片添加一些效果。

20.1 示例程序

本章的示例程序为 ImagePlayground，它可以从设备的照片库选择一张图片或者用摄像头拍摄的照片作为源图而获得图片，之后再将其压缩成小尺寸的图片。用户可以选择对源图片使用过滤器来修改图片的效果，通过在旁边表中给出的带有名称的过滤器对图片应用相应的效果。通过对过滤器的选择可以看出，Core Image 的过滤器被分成不同的类型，并且让用户可以自定义选中的过滤器并预览其效果。

20.2 基本图片数据和显示

要显示图片，示例程序还需要对图片进行一些基本的处理。本节会介绍一些不同的显示图片的技术，以及针对按钮的不同尺寸来处理可拉伸图片的技术，还会介绍从用户照片库或相机中获取图片的基本方法。

20.2.1 实例化图片

要在应用中使用图片，iOS 提供了一个名为 UIImage 的类。这个类支持许多图片格式：

- 可移植网络图形格式(PNG)：.png
- 标签图像文件格式(TIFF)：.tiff、.tif
- 联合图像专家小组(JPEG)：.jpeg、.jpg
- 图像互换格式(GIF)：.gif
- Windows 的位图文件(DIB)：.bmp
- Windows 的图标文件格式：.ico
- Windows 的光标文件格式：.cur
- X BitMap：.xbm

当将图片用于背景、按钮或其他用户界面中的元素时，苹果公司建议使用 PNG 格式。UIImage 具有一个名为 imageNamed:的方法，用于实例化图片对象。这个方法具有如下一些优势：

- 可以从应用的主 bundle 中查找和载入图片，不需要指定主 bundle 的路径。
- 自动载入 PNG 图片时不需要文件扩展名。即如果在应用的主 bundle 中存在资源，myImage 就会载入 myImage.png。
- 当载入图片时会考虑屏幕的尺寸，如果屏幕尺寸大于 1.0，它会将@2x 或@3x 自动添加到图片文件名后进行适配。此外，它还会检查图片的~ipad 和~iphone 版本，如果可用，则会使用它们。
- 支持内存缓冲。如果同一张图片已经载入又再次请求，将会返回已经载入的图片而不会重新载入图片。当用户界面多次使用同一张图片时这很有帮助。

对于不在应用的主 bundle 中出现的图片，不适合用 imageNamed:方法，还有其他一些方法可以对图片进行实例化。如下面的代码一样，可以从应用的 Documents 目录中载入 myImage.png 文件来实例化 UIImage：

```
NSArray *pathForDocuments =
➥NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                      NSUserDomainMask, YES);

NSString *imagePath =
➥[[pathForDocuments lastObject]
➥stringByAppendingPathComponent:@"myImage.png"];

UIImage *myImage =
➥[UIImage imageWithContentsOfFile:imagePath];
```

还可以从 NSData 来实例化 UIImage，如接下来的代码所示。NSData 可以来自任何源，通常来自于文件或者来自于从网络连接获取的数据。

```
NSData *imageData = [NSData dataWithContentsOfFile:imagePath];

UIImage *myImage2 = [UIImage imageWithData:imageData];
```

UIImage 可以通过 Core Graphics 图片来创建，如下面的示例代码所示，这里使用 Core Graphics 从一张已经存在的图片得到一个样本矩形。一张 Core Graphics 图片(CGImage 或

CGImageRef)代表一组 Core Graphics 图片数据。

```
CGImageRef myImage2CGImage = [myImage2 CGImage];
CGRect subRect = CGRectMake(20, 20, 120, 120);

CGImageRef cgCrop =
➥CGImageCreateWithImageInRect(myImage2CGImage, subRect);

UIImage *imageCrop = [UIImage imageWithCGImage:cgCrop];
```

可以从 Core Image 图片创建 UIImage，在后面的 20.3.4 节“渲染过滤后的图片”中会有详细介绍。

20.2.2　显示图片

在 UIImage 实例创建完毕后，有许多种方法可以将它显示在用户界面中。第一种方法就是使用 UIImageView。当 UIImageView 实例可用时，它有一个名为 image 的属性：

```
[self.sourceImageView setImage:scaleImage];
```

还可以使用一张图片初始化 UIImageView，它将会根据图片的尺寸设置边界：

```
UIImageView *newImageView =
➥[[UIImageView alloc] initWithImage:myImage];
```

对于在 UIImageView 中显示的图片，可以通过设置 contentMode 让图片的显示变得更有趣一些，同时还可以为图片添加一些实用的效果。

内容模式(content mode)的设置可以通知视图如何显示具体内容。aspect fit 和 fill 模式将会保存图片的长宽比，scale to fill 模式将会拉伸图片来适应视图的尺寸。这些模式，比如居中、顶部、底部、靠左和靠右，当图片在视图中定位时会保持图片的尺寸。

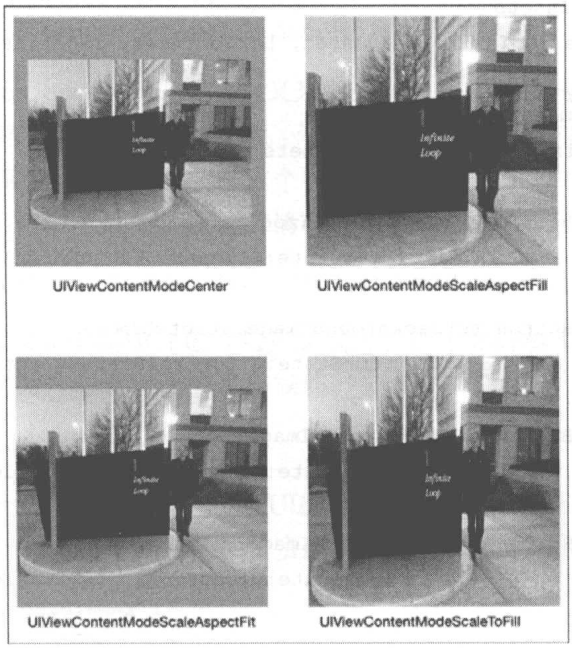


图 20-1　示例程序：内容模式的效果

许多其他 UIKit 控件支持或使用图片，比如 UIButton。对于按钮或其他一些特殊的可变尺寸的用户界面元素都有一项适用的技术，即使用 UIImage 类提供的可变尺寸图片功能。这样在按钮尺寸不同的情况下，可以让其保持同样的风格，而不需要对按钮图片的边缘进行扭曲或变形，如图 20-2 所示。重新设置图片尺寸的这个方法在 UIKit 中已经进行了优化并且执行速度和效率都得到了提升，可以大幅减少应用中需要的图片数量和资源的大小。

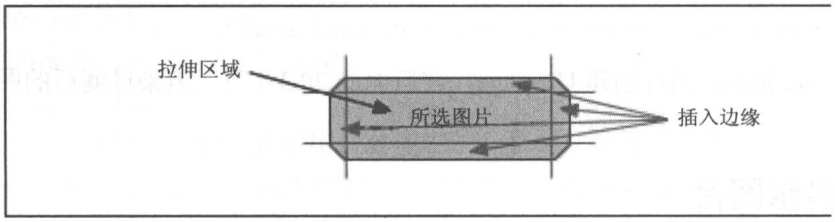


图 20-2 示例程序：使用可变尺寸背景图片的自定义按钮

按钮的源图片只有 28×28 像素。背景中用到的可变尺寸图片由原始图片创建，指定好的插入边缘用于表示图片的哪一部分应该保持不动，相应的其他部分就可以应用拉伸或平铺效果。

```
UIImage *startImage =
➥[UIImage imageNamed:@"ch_20_stretch_button"];

CGFloat topInset = 10.0f;
CGFloat bottomInset = 10.0f;
CGFloat leftInset = 10.0f;
CGFloat rightInset = 10.0f;

UIEdgeInsets edgeInsets =
➥UIEdgeInsetsMake(topInset, leftInset, bottomInset, rightInset);

UIImage *stretchImage =
➥[startImage resizableImageWithCapInsets:edgeInsets];

[self.selectImageButton setBackgroundImage:stretchImage
                                  forState:UIControlStateNormal];

[self.selectImageButton setBackgroundImage:stretchImage
                                  forState:UIControlStateSelected];

[self.selectImageButton setBackgroundImage:stretchImage
                                  forState:UIControlStateHighlighted];

[self.selectImageButton setBackgroundImage:stretchImage
                                  forState:UIControlStateDisabled];
```

提示

在 Xcode 5 或后续版本中升级之前的项目，使它们使用 Asset Catalog，可以在不添加代码的前提下使程序自动支持可变尺寸的图片。Asset Catalog 包括一个可视化编辑器，用于创建图层，对图层资源名调用 imageNamed:方法可以返回一张可变尺寸的图片。

20.2.3　使用图片选择器

在应用中使用用户提供的图片是很常见的情况。要让应用能够访问用户相册和相簿中的照片，iOS 提供了两种方法，分别是使用 UIImagePickerController 和使用资源库。UIImagePickerController 提供了一个模态的用户界面，用于导航到用户相簿和照片，所以当应用中用到苹果提供的设计样式并且对于照片的浏览和选择没有其他特殊要求时，选择这种方法很合适。使用资源库的方法提供对于照片和相簿的完全访问，所以当对于界面跳转和选择图片有特殊的用户界面和样式需求时，比较适合采用这种方法。有关照片库的内容会在第 24 章“访问照片库”中进行介绍。

要查看如何使用 UIImagePickerController，请参考示例程序中 ICFViewController 类的 selectImageTouched:方法。该方法首先分配内存空间并初始化一个 UIImagePickerController 实例。

```
UIImagePickerController *imagePicker =
➥[[UIImagePickerController alloc] init];
```

之后该方法会定制选择器。UIImagePickerController 可以被定制为从摄像头获取图片或视频，从照片库获取图片或视频，以及从保存好的相簿中获取图片或视频。这里我们指定从照片库获取资源。

```
[imagePicker setSourceType:
➥UIImagePickerControllerSourceTypePhotoLibrary];
```

可以对 UIImagePickerController 进行自定义来选择图片、视频或两者都选择，通过指定一个保存了媒体类型的数组进行设置。注意，这里用于指定媒体类型的参数都是定义在 MobileCoreServices 中的常量，所以有必要将这个框架添加到项目中，并在视图控制器类的实现过程中导入它。对于示例程序只需要照片的情况，使用的参数为 kUTTypeImage。

```
[imagePicker setMediaTypes:@[(NSString *)kUTTypeImage]];
```

选择器可以对允许或禁止编辑图片进行自定义设置。如果允许编辑，则用户可以在界面提供的一个矩形区域内对图片进行挤压或平移操作。

```
[imagePicker setAllowsEditing:YES];
```

最后，选择器接收一个委托。当用户根据 UIImagePickerControllerDelegate 协议选中一张图片或取消选中时，就会通知委托。委托还负责删除图片选择器的视图控制器。

```
[imagePicker setDelegate:self];
```

在示例程序中，图片选择器在一个弹出视图控制器中呈现，使用源图片容器视图作为其

锚视图。

```
self.imagePopoverController = [[UIPopoverController alloc]
➥initWithContentViewController:imagePicker];

[self.imagePopoverController
➥presentPopoverFromRect:self.sourceImageContainer.frame
➥inView:self.view
➥permittedArrowDirections:UIPopoverArrowDirectionAny
➥animated:YES];
```

当用户选择了一张图片并对其裁剪时，就会调用委托方法。

```
-(void)imagePickerController:(UIImagePickerController *)picker
➥didFinishPickingMediaWithInfo:(NSDictionary *)info
{
...
}
```

该委托方法会收到一个带有选中的媒体信息的字典对象。如果在选择过程中编辑，有关用户操作的编辑信息都会包含在该字典对象中。键和字典对象所包含的详细信息如表 20-1 所示。

表 20-1 UIImagePickerControllerDelegate 媒体信息字典

键	值
UIImagePickerControllerMediaType	一个 NSString 常量，表示用户选择的是图片(kUTTypeImage)还是视频(kUTTypeMovie)
UIImagePickerControllerOriginalImage	一个 UIImage，它包含用户选择的尚未裁剪或编辑的原始图片
UIImagePickerControllerEditedImage	一个 UIImage，它包含用户选择的已经裁剪过或编辑过的图片。如果图片没有编辑，其在字典中不可用
UIImagePickerControllerCropRect	一个 NSValue，它包含对于原始图片所应用的裁剪角度。如果图片没有编辑过，其在字典中不可用
UIImagePickerControllerMediaURL	一个 NSURL，指向设备文件系统中影片数据的位置。这个 URL 可以在应用中访问并且可以用于上传影片数据或在影片播放器中使用。如果媒体类型为 image，则不可使用该参数
UIImagePickerControllerReferenceURL	一个 NSURL，表示相对应的 ALAsset。利用第 24 章介绍的资源库，该参数可以直接用在与资源的交互上
UIImagePickerControllerMediaMetadata	一个 NSDictionary，它包含从摄像头获取的照片的元数据。如果是从照片库得到的图片或影片，则该参数不可用

在示例程序中，选中的图片都会被重新设置为 200×200 像素，这样方便程序处理并且显示效果也最好。首先，获取一个对正在编辑的图片的引用。

```
UIImage *selectedImage =
➥[info objectForKey:UIImagePickerControllerEditedImage];
```

之后，创建一个 CGSize 对象表示目标尺寸，为 UIImage 调用缩放方法对图片尺寸进行调整。将调整尺寸后的图片设置为将要在用户界面中显示的图片。

```
CGSizescaleSize = CGSizeMake(200.0f, 200.0f);

UIImage *scaleImage =
➥[selectedImage scaleImageToSize:scaleSize];

[self.sourceImageView setImage:scaleImage];
```

在设置好图片之后，带有图片选择器的弹出框就被移除了。

20.2.4　调整图片尺寸

当显示图片时，建议在保持美观的用户界面的同时，使用尽可能小的图片。虽然示例程序完全可以使用完整大小的图片，不过这样如果有大图片需要显示，程序性能就会受到很大的影响，因为需要更多的内存来处理这张大图片。因此，示例程序将会调整选择图片的尺寸，使其变成用户界面能够处理的大小正好的图片。要在 iOS 中对图片进行缩放，需要用到 Core Graphics。示例程序在 UIImage 上包含一个名为 Scaling 的分类方法(在 UIImage+Scaling 里)，以及一个名为 scaleImageToSize:的方法。该方法带有一个 CGSize 参数，即图片的长和宽。

调整图片尺寸的第一步是创建 Core Graphics 环境，作为图片的处理区域：

```
UIGraphicsBeginImageContextWithOptions(newSize,NO,0.0f);
```

UIKit 提供了一个便捷函数 UIGraphicsBeginImageContextWithOption，用于根据给定的选项创建 Core Graphics 环境。第一个参数是 CGSize，用于指定环境的尺寸，此时该方法使用从调用函数得到的尺寸。第二个参数是 BOOL，它会通知 Core Graphics 环境和结果图片是否作为不透明(YES)或包含 alpha 通道的透明(NO)对象处理。最后一个参数就是图片的缩放参数，1.0f 为非高清效果，2.0f 为高清效果。如果传递的参数为 0.0f，则意味着使用当前设备屏幕的尺寸。环境设置完毕后，使用 drawInRect:方法将图片绘制到环境中。

```
CGFloat originX = 0.0f;
CGFloat originY = 0.0f;

CGRectdestinationRect =
➥CGRectMake(originX, originY, newSize.width, newSize.height);

[self drawInRect:destinationRect];
```

drawInRect:方法使用目标矩形指定好的位置和大小，将图片内容绘制到 Core Graphics 环

境中。如果源图片的长和宽与新的目标尺寸不同，drawInRect:方法则会调整图片的大小以适配新的尺寸。

在整个处理过程中很重要的一点是，要考虑源图片的长宽比和目标区域的要求。长宽比即图片长和宽的比例。如果长宽比不同，drawInRect:方法将会拉伸或压缩图片以满足目标环境的要求，最终的图片显示可能会是扭曲的。

接下来，drawInRect:方法从环境中创建一个新的 UIImage，之后就可以结束该环境并返回调整后的新图片了。

```
UIImage *newImage = UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();
Return newImage;
```

20.3 Core Image 过滤器

用户可以在示例程序中选择一张初始图片，之后为该图片应用一系列的过滤器来达到最终的图片效果。一般情况下，Core Image 需要一张输入图片(除非过滤器自己生成一张图片)和过滤器用到的一些参数。当请求发生时，Core Image 将会对输入图片使用过滤器来得到输出图片。Core Image 可以有效地用于应用中的过滤器。只有当最终的输出图片被请求时才会应用过滤器列表，而不是在指定每个过滤器时应用。此外，Core Image 可以在任何合适的时候将这些过滤器进行数学逻辑上的组合，这样就会做到计算量最小。

注意

要在项目中使用 Core Image，需要在项目中添加 Core Image 框架和@import CoreImage，每个类都需要导入 CoreImage。

20.3.1 过滤器类别和过滤器

Core Image 过滤器在内部被分成几种类型。一个过滤器可以属于多个类型，例如，Sepia Tone 颜色效果过滤器就属于 6 个分类，包括 CICategoryColorEffect、CICategoryVideo、CICategoryInterlaced、CICategoryNonSquarePixels、CICategoryStillImage 和 CICategoryBuiltIn。这些分类可以帮助开发者确定任务所需的 Core Image 过滤器，还可以让用户浏览和选择可用的过滤器，如示例程序中所做的那样。

在示例程序中，当用户点击 Add Filter 按钮时，会启动一个弹出 segue。这个 segue 将会初始化一个 UIPopoverController、一个 UINavigationController 和一个 ICFFilterCategoriesViewController 实例(该实例作为导航控制器最顶部的根视图)。ICFViewController 实例会实现 ICFFilterProcessingDelegate 并使其作为委托，当选好过滤器或选择过程取消时程序就会得到通知。在 viewDidLoad 方法中，ICFFilterCategoriesViewController 实例会创建一个由用户可读的分类名组成的字典对象，对于要在表视图中呈现的每个元素都有相应的 Core Image 分类键。

```
self.categoryList = @{
    @"Blur" : kCICategoryBlur,
    @"Color Adjustment" : kCICategoryColorAdjustment,
```

```
    @"Color Effect" : kCICategoryColorEffect,
    @"Composite" : kCICategoryCompositeOperation,
    @"Distortion" : kCICategoryDistortionEffect,
    @"Generator" : kCICategoryGenerator,
    @"Geometry Adjustment" : kCICategoryGeometryAdjustment,
    @"Gradient" : kCICategoryGradient,
    @"Halftone Effect" : kCICategoryHalftoneEffect,
    @"Sharpen" : kCICategorySharpen,
    @"Stylize" : kCICategoryStylize,
    @"Tile" : kCICategoryTileEffect,
    @"Transition" : kCICategoryTransition
};
self.categoryKeys = [self.categoryList allKeys];
```

表视图看起来如图 20-3 所示。

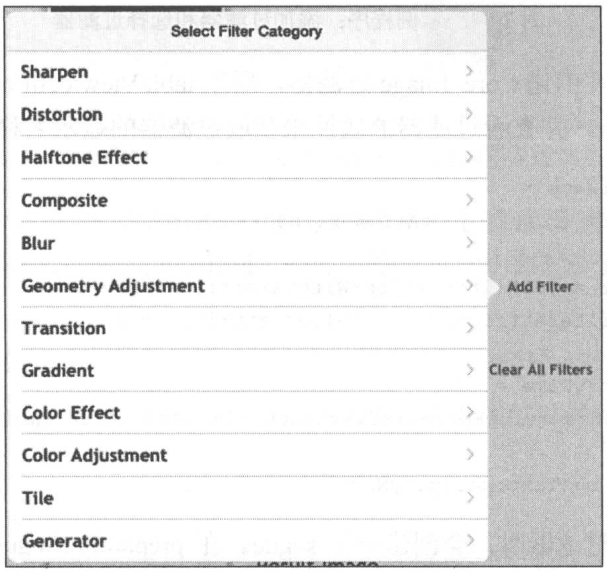


图 20-3　示例程序：添加过滤器和选择分类

当用户选择一个分类时，会创建一个 segue。在 prepareForSegue:sender:方法中，目标 ICFFiltersViewController 实例通过所选的分类的 Core Image 常量进行更新，并为其分配一个 ICFFilterProcessingDelegate 委托。在 viewDidLoad 方法中，ICFFiltersViewController 将会得到一个可用过滤器的列表。

```
self.filterNameArray =
➥[CIFilter filterNamesInCategory:self.selectedCategory];
```

表视图中显示的过滤器如图 20-4 所示。

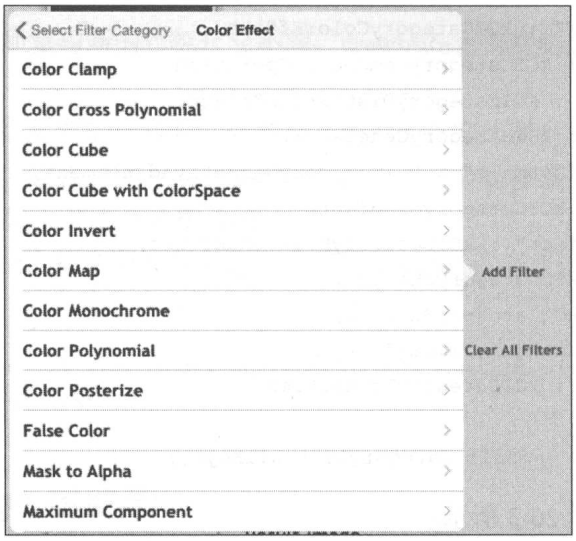


图 20-4 示例程序：添加过滤器和选择过滤器

可以根据名称来实例化 Core Image 过滤器，所以 tableView:cellForRowAtIndexPath:方法会实例化一个 CIFilter，并查看过滤器的属性来获取过滤器的显示名称。

```
NSString *filterName =
➥[self.filterNameArray objectAtIndex:indexPath.row];

CIFilter *filter = [CIFilter filterWithName:filterName];
NSDictionary *filterAttributes = [filter attributes];

NSString *categoryName =
➥[filterAttributes valueForKey:kCIAttributeFilterDisplayName];

[cell.textLabel setText:categoryName];
```

当用户选择一个过滤器时，会创建一个 segue。在 prepareForSegue:sender:方法中，目标 ICFFilterViewController 实例会根据所选的过滤器的 CIFilter 实例来更新，并为其分配 ICFFilterProcessingDelegate 委托。当 ICFFilterViewController 实例出现时，将会为选好的过滤器显示可以自定义的特性。

20.3.2 过滤器特性

Core Image 过滤器具有非常灵活的自定义方法。所有 CIFilter 实例都有一个名为 attributes 的字典特性，它包含有关过滤器的信息和所有对于过滤器的定制信息。CIFilter 实例包含一个名为 inputKeys 的属性，它列出了每个可自定义输入元素的键；还包含一个名为 outputKeys 的属性，它列出了从过滤器得到的输出元素。

示例程序包含一个专门的表单元格，用于显示哪些特性是所选过滤器可用的，调整这些特性并基于当前的特性参数预览图片，如图 20-5 所示。

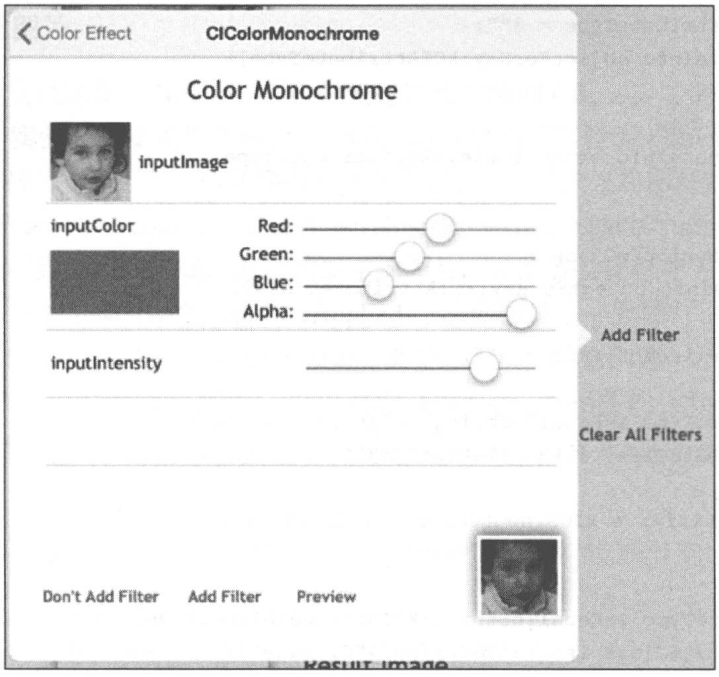


图 20-5　示例程序：添加 Color Monochrome 过滤器及过滤器特性

每个特性都包含用来确定其在用户界面中如何显示和编辑的信息。ICFFilterViewController 中的 tableView:cellForRowAtIndexPath:方法根据索引路径来查找具体的特性。

```
NSString *attributeName =
➥[[self.selectedFilter inputKeys] objectAtIndex:indexPath.row];
```

使用从过滤器的 inputKeys 属性得到的特性名称，该方法可以得到一个有关特性信息的 NSDictionary 对象。

```
NSDictionary *attributeInfo =
➥[[self.selectedFilter attributes] valueForKey:attributeName];
```

如果特性信息可用，该方法可以检查特性的类型和过滤器所希望的用于表达特性信息的类。使用这些信息方法可以令正确的自定义单元格出列(dequeue)以编辑特性。ICFInputInfoCell 是一个超类，它包含很多处理图片、颜色、数字、矢量和图片转换的子类。

```
NSString *cellIdentifier =
➥[self getCellIdentifierForAttributeType:attributeInfo];

ICFInputInfoCell *cell = (ICFInputInfoCell *)
➥[tableView dequeueReusableCellWithIdentifier:cellIdentifier
                                  forIndexPath:indexPath];
```

自定义 getCellIdentifierForAttributeType:方法使用特性类型和类来确定返回什么类型的单元格。注意，过滤器并不是完全同特性类型和类保持一致，一个过滤器可能指定一个 kCIAttributeTypeColor 特性类型，同时其他过滤器可能指定一个 CIColor 特性类。对下面的方法进行修改，只选择其中一个过滤器。

```
NSString *attributeType = @"";
if([attributeInfo objectForKey:kCIAttributeType])
{
    attributeType =
    ➥[attributeInfo objectForKey:kCIAttributeType];
}

NSString *attributeClass =
➥[attributeInfo objectForKey:kCIAttributeClass];

NSString *cellIdentifier = @"";

if([attributeType isEqualToString:kCIAttributeTypeColor] ||
    ➥[attributeClass isEqualToString:@"CIColor"])
{
    cellIdentifier = kICSInputColorCellIdentifier;
}

if([attributeType isEqualToString:kCIAttributeTypeImage] ||
    ➥[attributeClass isEqualToString:@"CIImage"])
{
    cellIdentifier = kICSInputImageCellIdentifier;
}

if([attributeType isEqualToString:kCIAttributeTypeScalar] ||
    ➥[attributeType isEqualToString:kCIAttributeTypeDistance] ||
    ➥[attributeType isEqualToString:kCIAttributeTypeAngle] ||
    ➥[attributeType isEqualToString:kCIAttributeTypeTime])
{
    cellIdentifier = kICSInputNumberCellIdentifier;
}

if([attributeType isEqualToString:kCIAttributeTypePosition] ||
    ➥[attributeType isEqualToString:kCIAttributeTypeOffset] ||
    ➥[attributeType isEqualToString:kCIAttributeTypeRectangle])
{
    cellIdentifier = kICSInputVectorCellIdentifier;
}

if([attributeClass isEqualToString:@"NSValue"])
{
    cellIdentifier = kICSInputTransformCellIdentifier;
}

return cellIdentifier;
```

每个单元格子类都可以接收特性字典，根据给定或默认的值配置显示效果，管理编辑中的参数值，当请求发生时返回一个预期类的实例。

20.3.3　初始化图片

要对图片使用过滤器，Core Image 所需的是图片对应的 CIImage 实例。要从 UIImage 得到一个 CIImage 实例，需要先将其转换为 CGImage，之后再转换为 CIImage。如果 UIImage 在内存中已经有图片数据了，这些转换很快就会完成；否则必须先将图片数据载入到内存中。

在 ICFFilterViewController 中，tableView:cellForRowAtIndexPath:对 inputImage 进行处理，通过过滤器委托得到初始图片或之前过滤器中的图片。过滤器委托保存了一个 UIImage 对象组成的数组，用于从 imageWithLastFilterApplied 方法返回最后一张图片。

如果存在初始图片，将其转换为 CGImage 类型，之后使用它创建一个 CIImage 对象。如果输入图片是从其他过滤器得到的，可以假定 UIImage 有相关的 CIImage 对象(只有当 UIImage 是从 CIImage 中创建的时候，才可以向 UIImage 请求其 CIImage 格式，否则会返回 nil)。

```
if([attributeName isEqualToString:@"inputImage"])
{
    UIImage *sourceImage =
    ➥[self.filterDelegate imageWithLastFilterApplied];

    [[(ICFInputImageTableCell *)cell inputImageView]
    ➥setImage:sourceImage];

    CIImage *inputImage = nil;
    if([sourceImage CIImage])
    {
        inputImage = [sourceImage CIImage];
    }
    else
    {
        CGImageRef inputImageRef = [sourceImage CGImage];
        inputImage = [CIImage imageWithCGImage:inputImageRef];
    }

    [self.selectedFilter setValue:inputImage
                           forKey:attributeName];
}
```

20.3.4　渲染过滤后的图片

要渲染一张过滤后的图片，需要从过滤器的特性中请求 outputImage，并调用一个可用的方法将其渲染为环境、图片、位图文件或像素缓存，所以过滤器操作可以作用于 inputImage 并产生 outputImage。在示例程序中，上述过程发生在两种实际情况下：用户在过滤器视图控制器中点击 Preview(如图 20-5 所示)，或者用户点击 Add Filter。当用户在 ICFFilterViewController 中点击 Preview 按钮时，会调用 previewButtonTouched:方法。这个方法首先会初始化一个 Core Image 环境(注意这个环境可以作为个属性被创建一次，不过在此创建它是为了能够在同一个界面中演示这个功能)。

```
CIContext *context = [CIContext contextWithOptions:nil];
```

上面的方法之后会从过滤器得到一个对 outputImage 的引用，并创建一个矩形区域告诉环境渲染图片的哪个部分。此时矩形区域的大小同图片的尺寸一样并且可能看起来略大一些。不过注意，有一些过滤器可以生成无限大尺寸的输出图片(参考过滤器分类 Generator)，所以在指定所需的大小时一定要谨慎。

```
CIFilter *filter = self.selectedFilter;
CIImage *resultImage = [filter valueForKey :kCIOutputImageKey];
CGRectimageRect = CGRectMake(0.0f,0.0f,200.0f,200.0f);
```

之后，该方法会使用 outputImage 和矩形区域请求环境创建一张 Core Graphics 图片。

```
CGImageRef resultCGImage =
➥[context createCGImage:resultImage fromRect:imageRect];
```

Core Graphics 的结果图片可以用于创建在屏幕上显示的 UIImage 图片。

```
UIImage *resultUIImage =
[UIImage imageWithCGImage:resultCGImage];

[self.previewImageView setImage:resultUIImage];
```

预览图片在过滤器视图的右下角显示，如图 20-5 所示。

20.3.5 链式过滤

链式过滤是指对一张源图片应用多于一个过滤器的过程。将一个组合过滤器应用于源图片，可以产生有趣的效果，如图 20-6 所示。

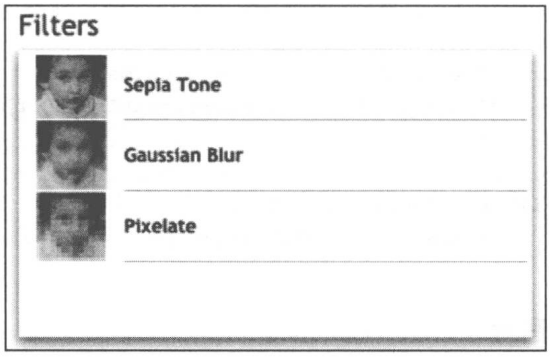


图 20-6 示例程序：过滤器列表

当用户点击 Add Filter(见图 20-5)时，在 ICFViewController 中会调用 ICFFilterProcessing 协议的 addFilter:方法。该方法会查看即将添加的过滤器是否为第一个过滤器，如果是，将源图片作为 inputImage 添加给过滤器。如果不是第一个过滤器，该方法将会使用最后一个过滤器的 outputImage 作为将要添加过滤器的 inputImage。

```
CIFilter *lastFilter = [self.filterArray lastObject];
```

```
if(lastFilter)
{
    if([[filter inputKeys] containsObject:@"inputImage"] )
    {
        [filter setValue:[lastFilter outputImage]
                  forKey:@"inputImage"];
    }
}

[self.filterArray addObject:filter];
```

使用这种技术，任意数量的过滤器都可进行链式组合。经过最后一个过滤器之后，该方法就会渲染最终图片。

```
CIContext *context = [CIContext contextWithOptions:nil];
CIImage *resultImage = [filter valueForKey:kCIOutputImageKey];

CGImageRef resultCGImage =
➥[context createCGImage:resultImage
             fromRect:CGRectMake(0.0f, 0.0f, 200.0f, 200.0f)];

UIImage *resultUIImage =
➥[UIImage imageWithCGImage:resultCGImage];
```

最终图片会被添加到图片列表中，并且过滤器列表也会重新载入以显示对图片使用的每一步过滤器的信息。

```
[self.resultImageView setImage:resultUIImage];

[self.filteredImageArray addObject:self.resultImageView.image];
[self.filterList reloadData];

[self.filterPopoverController dismissPopoverAnimated:YES];
```

Core Image 将会自动优化过滤器链来最小化计算量，换句话说，Core Image 不会独立处理过滤的每一步，而是根据过滤器的描述组合一些数据运算，最后一步执行过滤操作即可。

20.4　特征检测

Core Image 提供了针对图片和视频的特征检测功能，包括人脸识别和从 iOS 7 开始的面部特征识别，以及从 iOS 8 开始的二维码和矩形马赛克识别等功能。特征检测可以用在很多场合下，比如标准的照相机应用可以在取景框中高亮显示识别出的人脸。识别了人脸的位置和大小，可以用于过滤器来做鬼脸或者用于各种有趣的应用。矩形马赛克的尺寸可以类似地用于过滤器来指定一张图片的一部分，这为各种创意应用打开了广阔的想象空间。示例程序包含在源图片上检测和标注人脸及人脸特征等功能，同样的技术也可以用于二维码和矩形马赛克的识别。

20.4.1 创建人脸检测器

要使用人脸检测，Core Image 需要一个 CIImage 实例作为源图片进行分析。在 ICFViewController 的 detectFacesTouched:方法中，根据显示的源图片创建一个 CIImage 实例。

```
UIImage *detectUIImage = [self.sourceImageView image];
CGImageRef detectCGImageRef = [detectUIImage CGImage];

CIImage *detectImage =
➥[CIImage imageWithCGImage:detectCGImageRef];
```

Core Image 中的人脸检测功能由 CIDetector 类实现。这个类方法创建一个检测器，并接受以字典对象表示的多种选择设置。在这种情况下，一个人脸检测器可以设置为高精度的检测或低精度的检测。高精度的检测有更高的识别率，不过完成识别消耗的时间更长，所以高精度的识别适合于那些不太注重性能的情况。低精度的检测适合于类似实时视频反馈类场景，即对于性能的要求高于识别率。可以提供 Core Image 环境，不过它不是必需的。

```
NSDictionary *options =
➥@{CIDetectorAccuracy:CIDetectorAccuracyHigh};

CIDetector *faceDetector =
➥[CIDetector detectorOfType:CIDetectorTypeFace
                    context:nil
                    options:options];
```

创建完检测器后，调用 featuresInImage:方法将会得到检测到的源图片特征。

```
NSArray *features = [faceDetector featuresInImage:detectImage];
```

所发现的特征将会以 CIFaceFeature 实例的形式返回，它是 CIFeature 的子类。CIFeature 实例具有一个 bounds 属性，它是一个矩形区域，用来表示源图片中特征的相对位置。

20.4.2 处理人脸特征

ICFViewController 的 detectFacesTouched:方法将会遍历所找到的人脸，并在源图片中高亮显示它们，如图 20-7 所示。该方法还会将每个人脸的特征信息在一个文本视图中详细显示出来。

对于每个人脸，该方法首先会在人脸周围生成一个红色边框的矩形区域。

```
CGRect faceRect =
➥[self adjustCoordinateSpaceForMarker:face.bounds
                              andHeight:detectImage.extent.size.height];
```

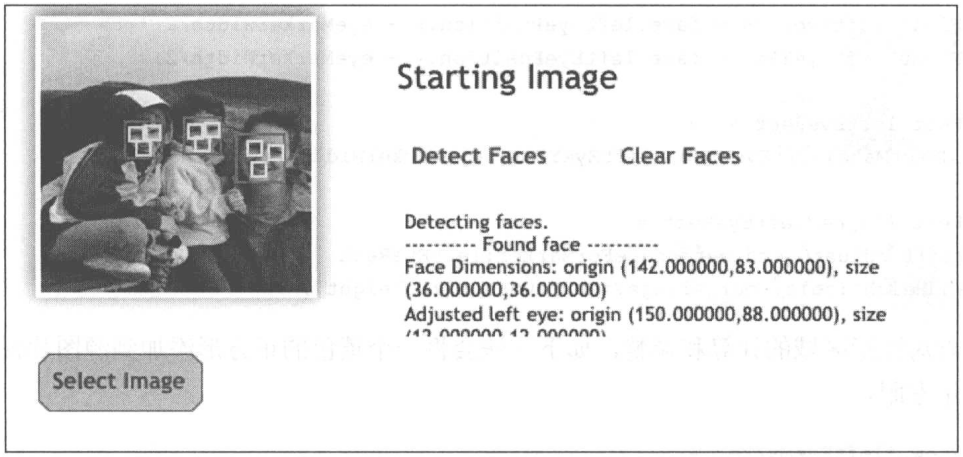


图 20-7　示例程序：人脸检测

要确定如何对找到的人脸绘制矩形框，该方法首先会根据 Core Image 调整位置，使用与 UIKit 不同的坐标系。对 Core Image 坐标系的 Y 轴进行翻转，即垂直方向的位置 0 位于图片的底部，而不是原来的顶部。adjustCoordinateSpaceForMarker:andHeight:方法将会根据图片的高度对 Y 轴翻转来调整矩形区域，这样新坐标的表达就同 UIKit 使用的坐标系一样了。

```
CGAffineTransform scale = CGAffineTransformMakeScale(1, -1);

CGAffineTransform flip =
➥CGAffineTransformTranslate(scale, 0, -height);

CGRect flippedRect = CGRectApplyAffineTransform(marker, flip);
return flippedRect;
```

坐标正确之后，如下方法会将一个带有红色边框的基本视图添加到源图片视图，以突出显示人脸。

```
UIView *faceMarker = [[UIView alloc] initWithFrame:faceRect];
faceMarker.layer.borderWidth = 2;
faceMarker.layer.borderColor = [[UIColor redColor] CGColor];
[self.sourceImageView addSubview:faceMarker];
```

CIFaceFeature 类有一个方法可以表示是否检测出人脸中的左眼、右眼和嘴。从 iOS 7 开始，还有方法可用于检测面部是否处于微笑的表情(hasSmile)，或者是否眨眼(leftEyeClosed 和 rightEyeClosed)。

```
if(face.hasLeftEyePosition)
{
...
}
```

如果检测到眼睛或嘴，将会为面部特征给出一个 CGPoint 表示的位置。由于对于每个面部特征只有唯一的位置(没有大小)，因此会使用默认的长和宽组成的矩形区域来表示这个面部特征的位置，如下所示：

```
CGFloat leftEyeXPos = face.leftEyePosition.x - eyeMarkerWidth/2;
CGFloat leftEyeYPos = face.leftEyePosition.y - eyeMarkerWidth/2;

CGRect leftEyeRect =
➥CGRectMake(leftEyeXPos, leftEyeYPos, eyeMarkerWidth, eyeMarkerWidth);

CGRect flippedLeftEyeRect =
➥[self adjustCoordinateSpaceForMarker:leftEyeRect
➥andHeight:self.sourceImageView.bounds.size.height];
```

随着对矩形区域的计算和调整，如下方法会将一个黄色的正方形添加到源图片视图，以突出显示左眼。

```
UIView *leftEyeMarker =
➥[[UIView alloc] initWithFrame:flippedLeftEyeRect];

leftEyeMarker.layer.borderWidth = 2;

leftEyeMarker.layer.borderColor =
➥[[UIColor yellowColor] CGColor];

[self.sourceImageView addSubview:leftEyeMarker];
```

同样的方法也可用于检测右眼和嘴。

20.5 小结

本章介绍了基本的图片处理方法，包括如何载入和显示图片，如何指定内容模式来调整图片的显示，以及如何创建一张可拉伸的图片在不同尺寸的情况下重用。还介绍了如何从用户照片库或摄像头获取图片，如何通过各种选项自定义图片选择器，比如选择相簿和是否允许对选中的图片进行修剪，同时还介绍了如何调整图片的尺寸。

之后，本章介绍了如何使用 Core Image 过滤器，包括如何得到有关可用过滤器的信息和过滤器类别，如何在一张图片上应用过滤器，以及如何使用链式过滤器实现一些有趣的效果。最后，本章介绍了如何应用 Core Image 的人脸检测功能。

第21章 集合视图

集合视图(collection view)是在 iOS 6 中新添加的一种数据显示方法，这种方法能够将可滑动的基于单元格的信息在视图中任意布局。想象下 Photos.app 的 iOS 6+版本，它可以在一个可滑动的网格中呈现图片的单元格。在 iOS 6 之前，实现一个网格视图可能需要对表视图设置一些复杂的逻辑，即计算单元格应该处于表中行的位置信息，或者可能需要定制逻辑，将单元格置于滑动视图中并对这些单元格进行管理。这两个方法都既复杂又耗时长，并且在实现过程中还极有可能出现错误。集合视图通过提供一个单元格管理架构来处理这种情况，同表视图管理行的方法类似，将单元格的布局从整体中抽离出来。

集合视图有一个默认的布局，称作流布局(flow layout)，用于在水平和垂直滑动时快速简单地实现多个网格类型的布局。还可以创建定制布局，实现特殊的网格类型，或者实现在运行时可见和可计算的无网格布局。

集合视图可以分成多个分节(section)，并根据分节数据显示分节头(section header)和分节尾(section footer)视图。此外，还可以用一些和内容数据不相关的修饰视图(decoration view)来增强集合视图的显示效果。

同样重要的一点是，集合视图支持多种类型的动画，包括单元格滑动时的定制状态，插入或移除单元格时的动画，以及在布局间进行切换时的动画效果。

21.1 示例程序

本章的示例程序 PhotoGallery 会通过几种不同的方法实现集合视图，在集合视图中显示用户的照片库。

- 第一个实现方法是一个基本的单元格集合视图，按相簿分组，可以垂直滑动。分节头显示相簿名称，可以使用最少量的定制代码来创建它。
- 第二个实现方法是使用一个定制的流布局子类，这样就可以使用修饰视图了。
- 第三个实现方法在无网格布局中使用定制布局呈现控件，并且包含通过捏压手势(pinch gesture)切换布局的功能。

21.2 集合视图介绍

为了完成相应的功能，集合视图需要用到一些不同的类。基类叫作 UICollectionView，它是 UIScrollView 的子类，负责管理 datasource 对象提供的单元格的显示(可以是任何实现了 UICollectionViewDataSource 协议的对象)。根据集合视图引用的布局，将会得到一个 UICollectionViewLayout 实例。还需要指定一个遵循 UICollectionViewDelegate 协议的 delegate，用于管理单元格的选中和高亮事件。

遵循 UICollectionViewDataSource 协议的类将会在集合视图中返回配置好的单元格，即 UICollectionViewCell 实例。如果集合视图被配置为使用分节头和(或)分节尾，数据源将会返回配置好的 UICollectionReusableView 实例。

在示例程序中，参看 Basic Flow Layout 以了解所有这些类的情况，如图 21-1 所示。

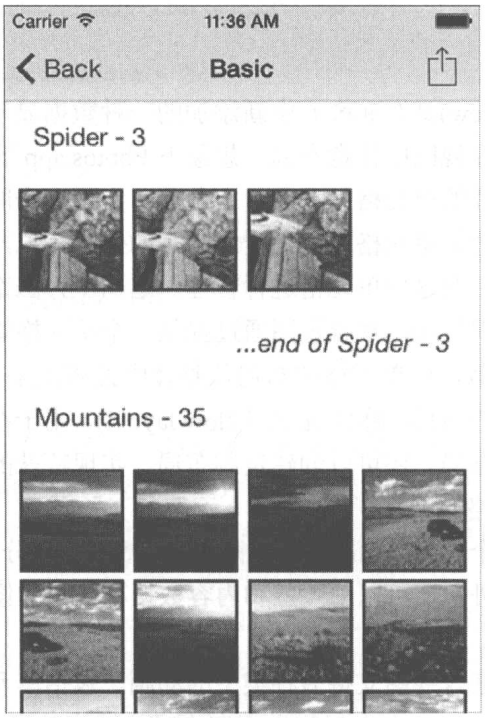


图 21-1 示例程序：Basic Flow Layout

21.2.1 创建一个集合视图

示例程序中的 Basic Flow Layout 使用最少量的定制代码创建了一个集合视图，该例演示了如何用快速简单的方法创建一个集合视图。同之前使用基本的 UIViewController 子类不同，Basic Flow Layout 使用一个名为 PHGBasicFlowViewController 的 UICollectionViewController 子类，其遵循 UICollectionViewDataSource 和 UICollectionViewDelegate 协议。这个方法不是必需的，当视图控制器中只显示集合视图时使用这个方法会比较方便。集合视图可以毫无问题地使用标准视图控制器。

(1) 在 MainStoryboard 中，检查 Basic Flow View Controller–Basic Scene。

(2) 展开场景查看集合视图控制器，如图 21-2 所示。

(3) 对于选中的集合视图控制器，注意 identity 查看器中指定的定制类。

这会确保集合视图控制器使用定制子类 PHGBasicFlowViewController。

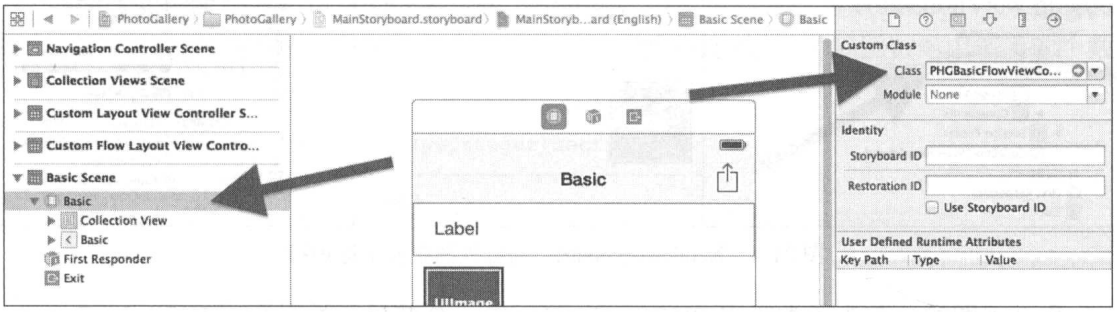


图 21-2 Xcodestoryboard：为集合视图控制器指定定制类

UICollectionViewController 实例包含一个名为 collectionView 的属性，在 Interface Builder 中表示为集合视图对象。对于选中的集合视图对象，注意可以为其配置一些设置：布局的类型、滑动方向以及是否使用分节头和(或)分节尾，如图 21-3 所示。

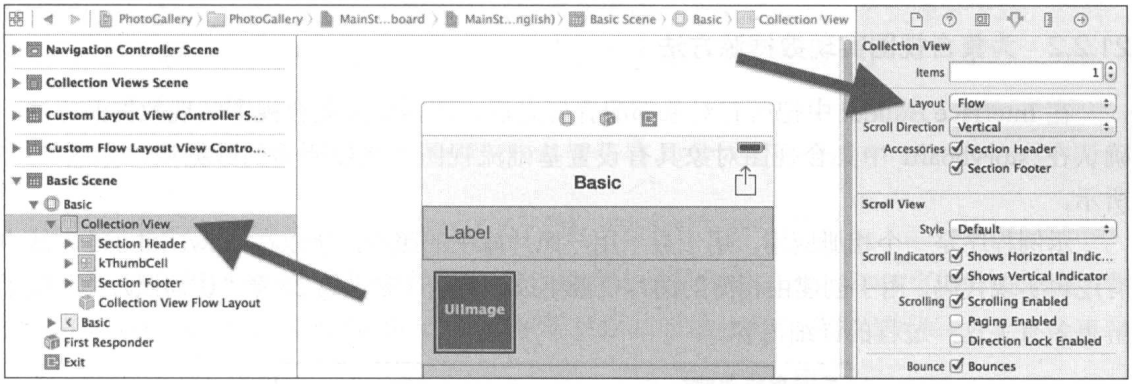


图 21-3 Xcodestoryboard：定制集合视图设置

Interface Builder 将会显示需要定制的分节头、集合视图单元格和分节尾的对象。对于每个对象，都已经设置好了一个定制子类，这些子类用于简化对于运行时所需子视图的管理。但这不是必需的，如果愿意，可以直接使用 UICollectionViewCell 和 UICollectionReusableView 这两个类。

集合视图的单元格(例子中的每个单元格即一个缩略图)子类叫做 PHGThumbCell，还有一个属性叫作 thumbImageView，它用于显示单元格。Interface Builder 中的集合视图对象在 identity 查看器中配置为使用定制子类，并建立 UIImageView 和 thumbImageView 属性间的关联。对于集合视图单元格需要的所有设置里面，最关键的是标识符，如图 21-4 所示；这是数据源方法识别单元格类型和显示它们的依据。

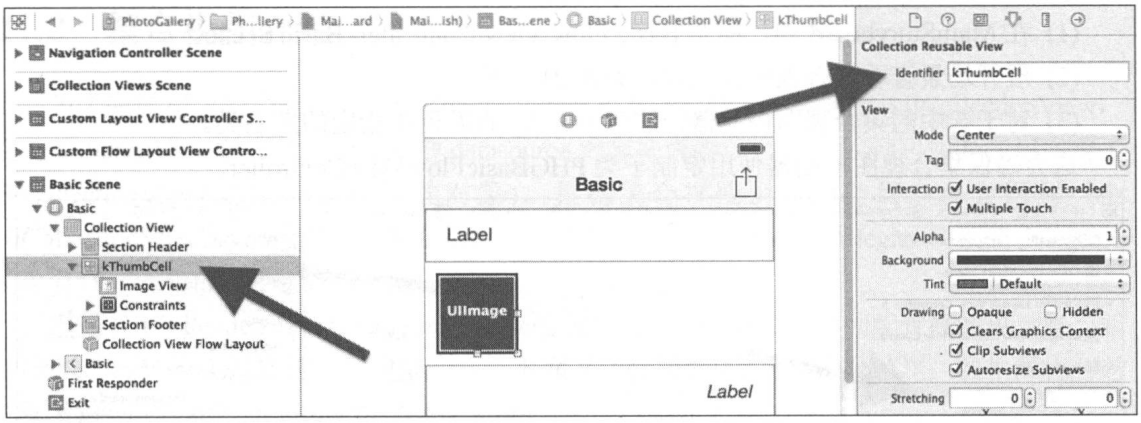


图 21-4 Xcodestoryboard：集合视图单元格标识符

集合视图的分节头子类叫作 PHGSectionHeader，分节尾子类叫做 PHGSectionFooter。每个子类都有一个 label 属性，用于显示分节头和分节尾的标题。这两个对象在 Interface Builder 中使用它们各自的定制子类，以及为标题属性关联各自的 UILabel 对象。如同集合视图单元格的标识符是为集合视图单元格指定的一样，分节头和分节尾也都有各自指定好的标识符。

21.2.2 为集合视图实现数据源方法

在 Interface Builder 中把所有对象都配置好之后，就需要为集合视图实现数据源方法。确认在 storyboard 中集合视图对象具有设置基础流视图控制器所需的数据源，如图 21-5 所示。

示例程序是一个相册应用，用于显示用户照片库中的照片，所以在 viewDidLoad 方法中有逻辑实现代码，用于创建由相簿的图片资源组成的数组。参见第 24 章“访问照片库”以了解更多有关这一过程的详细内容。

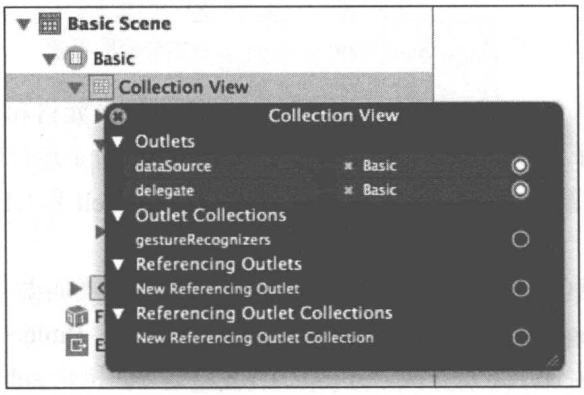


图 21-5 Xcodestoryboard：集合视图数据源设置

集合视图需要知道所要显示的分节的个数，这由 numberOfSectionsInCollectionView:方法返回的结果可知。该方法会根据 viewDidLoad 中创建的资源组数来返回相簿(或照片组)的个数。

```
-(NSInteger)numberOfSectionsInCollectionView:(UICollectionView *)collectionView
{
    return [self.albumsResult count];
}
```

接下来集合视图要知道每个分节包含多少元素，所使用的方法为 collectionView:numberOfItemsInSection:。该方法会根据分节索引号查找符合的资源数组，并返回数组中资源的个数。

```
-(NSInteger)collectionView:(UICollectionView *)view
     numberOfItemsInSection:(NSInteger)section;
{
    PHFetchResult *albumAssets = self.albumAssetsResults[section];
    return [albumAssets count];
}
```

集合视图知道了分节数和其中元素的个数之后，就知道如何对视图进行布局了。根据当前视图在可滑动边界的位置，集合视图会请求相应的分节头、分节尾以及视图可见区域中显示的单元格。分节头和分节尾通过 collectionView:viewForSupplementaryElementOfKind:atIndexPath: 方法获取。分节头和分节尾都需要是 UICollectionReusableView 实例(或子类)。该方法声明 UICollectionReusableView 的 nil 实例，这样将会生成一个配置好的分节头或分节尾。此外，该方法还会获取相簿和其中资源的信息，并在分节头或分节尾中显示。

```
UICollectionReusableView *supplementaryView = nil;
PHAssetCollection *album = [self.albumsResult objectAtIndex:indexPath.section];
PHFetchResult *albumAssets =
➥[self.albumAssetsResults objectAtIndex:indexPath.section];
```

方法中的逻辑必须检查 kind 参数的值(该值应该是 UICollectionElementKindSectionHeader 或 UICollectionElementKindSectionFooter)来确定返回的是分节头还是分节尾。

```
if([kind isEqualToString:UICollectionElementKindSectionHeader]) {

    PHGSectionHeader *sectionHeader =
    ➥[collectionView dequeueReusableSupplementaryViewOfKind:kind
    ➥withReuseIdentifier:kSectionHeader forIndexPath:indexPath];

    [sectionHeader.headerLabel
    ➥setText:[NSString stringWithFormat:@"%@ - %lu",album.localizedTitle,
    ➥(unsigned long)albumAssets.count]];

    supplementaryView = sectionHeader;
}
```

要得到一个定制的 PHGSectionHeader 实例，要求集合视图根据指定的索引路径提供一个附加视图，通过重用标识符使相应的元素出列。注意，重用标识符必须同之前在 Interface Builder 中为分节头指定的相同。该方法会初始化一个新的视图或重用一个不再使用的已有视

图。之后在组数组(group array)中查找分节标题，并将其赋给分节头的标题标签。

对单元格调用 collectionView:cellForItematIndexPath:方法。在这个方法中，根据重用标识符指定的单元格出列(必须同 Interface Builder 中单元格的重用标识符匹配)。

```
PHGThumbCell *cell =
[cv dequeueReusableCellWithReuseIdentifier:kThumbCell
                                forIndexPath:indexPath];
```

之后配置单元格，显示 indexPath 对应资源的单元格。

```
PHFetchResult *albumAssets = self.albumAssetsResults[indexPath.section];
PHAsset *asset = albumAssets[indexPath.row];

PHImageManager *imageManager = [PHImageManager defaultManager];
[imageManager requestImageForAsset:asset
                        targetSize:CGSizeMake(50, 50)
                       contentMode:PHImageContentModeAspectFill
                           options:nil
                     resultHandler:^(UIImage *result, NSDictionary *info){
                         [cell.thumbImageView setImage:result];
                         [cell setNeedsLayout];
                     }];

return cell;
```

如果在 storyboard 中创建单元格和分节头/分节尾对象不是你所喜欢的方法，还可以在 nibs 或代码中创建它们。这样就需要为单元格的重用标识符注册类或 nib，分别使用方法 registerClass:forCellWithReuseIdentifier:和 registerNib:forCellWithReuseIdentifier:。对应分节头和分节尾使用的方法分别是 registerClass:forSupplementaryViewOfKind:withReuseIdentifier:和 registerNib:forSupplementaryViewOfKind:withReuseIdentifier:。

21.2.3 实现集合视图委托方法

集合视图委托可以对单元格的选中和突出显示进行管理，可以跟踪单元格的移除或分节，可以用于为元素显示 Edit 菜单并执行相应的编辑操作。示例程序中的基础流可以演示单元格的选中和突出显示效果。要确保集合视图对象的委托设置到基础流视图控制器，如图 21-5 所示。

当集合视图单元格被选中或突出显示时，显示效果被设计成会发生变化。集合视图单元格具有一个名为 contentView 的子视图，所有需要显示的内容都在这个子视图中。它还有一个 backgroundView 视图，可以定制 backgroundView 视图，同时它始终处于 contentView 视图的后面。此外，它具有一个 selectedBackgroundView 视图，被置于 contentView 视图的后面，当单元格突出显示或被选中时置于 backgroundView 视图之前。

对于定制的 PHGThumbCell 类，初始化一个 selectedBackgroundView 并在单元格的 initWithCoder:方法中对其进行定制设置，这是因为单元格是从 storyboard 生成的。根据单元

格期望初始化的情况，确保使用合适的 init 方法来定制 backgroundView 和 selectedBackground-View。

```
-(instancetype)initWithCoder:(NSCoder *)aDecoder
{
    self = [super initWithCoder:aDecoder];
    if(self) {

        self.selectedBackgroundView =
        ➥[[UIView alloc] initWithFrame:CGRectZero];

        [self.selectedBackgroundView
        ➥setBackgroundColor:[UIColor redColor]];
    }
    return self;
}
```

默认情况下集合视图支持单选。要想集合视图支持多选，使用下面的方法：

```
[self.collectionView setAllowsMultipleSelection:YES];
```

对于单元格的选择，一共有 4 个委托方法可以使用。其中两个方法用于表示单元格是否应该被选中或取消选中，另外两个方法用于表示单元格是否已经被选中或取消选中。在这个示例中，程序只实现后两种方法。

```
-(void)collectionView:(UICollectionView *)collectionView
➥didSelectItemAtIndexPath:(NSIndexPath *)indexPath
{
    NSLog(@"Item selected at indexPath: %@",indexPath);
}

-(void)collectionView:(UICollectionView *)collectionView
➥didDeselectItemAtIndexPath:(NSIndexPath *)indexPath
{
    NSLog(@"Item deselected at indexPath: %@",indexPath);
}
```

注意，这两个方法中还没有逻辑代码来实际管理选中元素的列表——由集合视图处理。当单元格被选中时就需要有关选中操作的委托方法来对一些定制情况进行处理，或者对选中或取消选中进行响应。集合视图会维护一个保存有选中单元格索引路径信息的数组，这个数组可以用在任何定制的逻辑代码中。示例中演示了在导航栏点击一个动作按钮以显示所选单元格的数量，如图 21-6 所示。很容易将其优化，显示所选单元格的一个活动视图。

图 21-6 示例程序：演示单元格选择的基本流程(basic flow)

21.3 定制集合视图和流布局

对于流布局的集合视图有多种不同的定制方法。每个单元格的尺寸都可以独立地进行定制，同样每个分节头和分节尾的尺寸也都可以定制。单元格间都有一条参照线用于确保相互预留了最小空间，在单元格和分节头、分节尾和分节边界之间同样有参照线。此外，用于提升集合视图美观的修饰视图可以放在集合视图的任何位置，它和集合视图的数据没有直接的关系。

21.3.1 基础定制

流布局 SDK 提供了多种不同的基于网格的布局模式。流布局对所显示图片的尺寸、大小和分节情况，以及在每行显示多少图片和滑动视图的大小都能定制，并内置了基于滑动方向和所有参数设置的计算逻辑。当使用这些参数进行操作时，可以创建一个集合视图，令每行只显示一张图片(甚至每个屏幕显示一张图片)，或者多张图片紧密排列在一行(如 iOS 7 的 Photo.app 一样)，或是这两种情况中的任何一种。这些参数在图 21-7 中有详细的介绍。

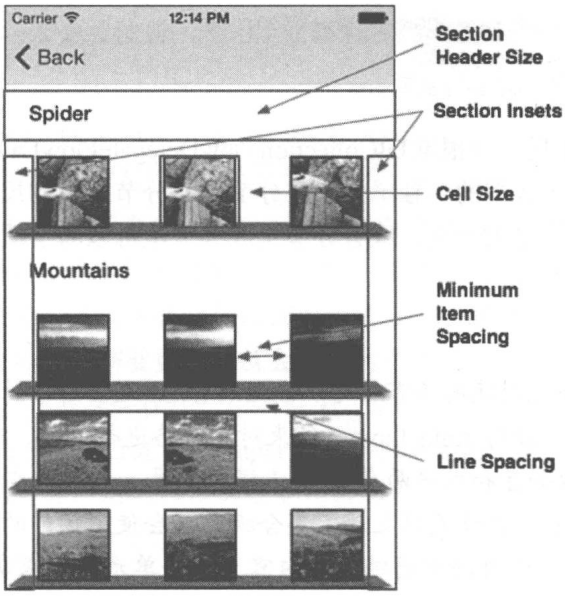


图 21-7 集合视图的可定制参数

有多种方法可以对流布局集合视图进行定制。最简单的方法就是在 Interface Builder 中为集合视图设置默认状态，通过选择集合视图对象(或集合视图流布局对象)并使用 SizeInspector，如图 21-8 所示。注意调整这些参数会影响和集合视图相关联的流布局对象。

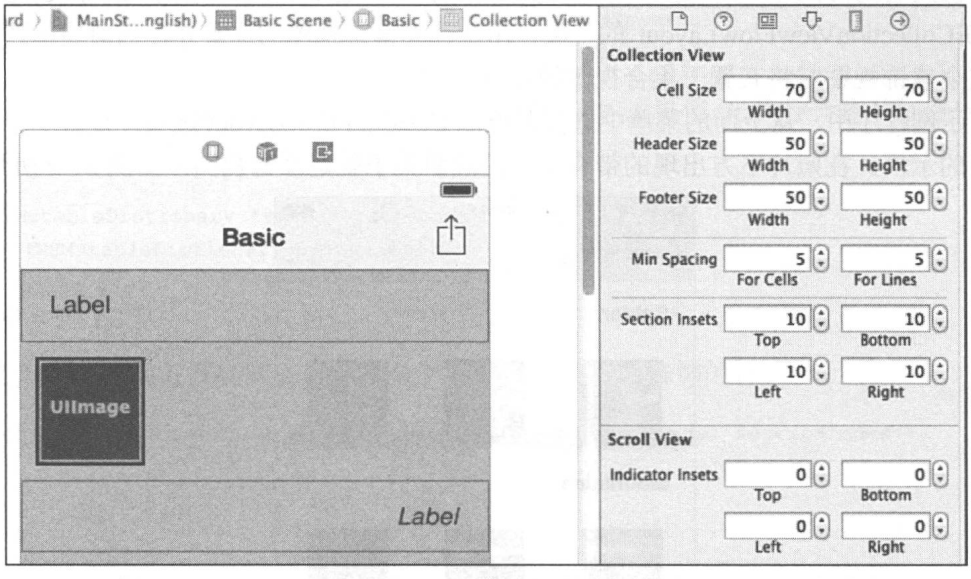


图 21-8 Xcode Interface Builder：集合视图对象的 Size Inspector 界面

另一个方法是在代码、实例或 UICollectionViewFlowLayout 子类中更新元素。在 PHGCustomFlowLayout 中，定制流布局的特性是在 init 方法中设置的。

```
self.scrollDirection = UICollectionViewScrollDirectionVertical;

self.itemSize = CGSizeMake(60, 60);
self.sectionInset = UIEdgeInsetsMake(10, 26, 10, 26);
```

```
self.headerReferenceSize = CGSizeMake(300, 50);
self.minimumLineSpacing = 20;
self.minimumInteritemSpacing = 40;
```

最后，集合视图的委托可以根据 UICollectionViewDelegateFlowLayout 协议实现一些方法。这些方法可以用于为基于数据的单独单元格或分节头、分节尾定制尺寸。比如，有着更高用户评级的照片可以显示的更大一点，或者分节头或分节尾可以为了适应长标题而扩展显示额外的行。

注意

从 iOS 8 开始，集合视图使用流布局可以根据所含 UI 元素内容的大小自动调整单元格的尺寸。要实现这个功能，使用 Auto Layout 约束对单元格进行设置，可以允许一个或多个 UI 元素在布局的滑动方向(垂直和水平两个方向)上进行扩展。之后，在代码中设置集合视图的流布局中预计元素的尺寸。在设置好之后，集合视图将会使用预计的元素尺寸对集合视图的布局进行初始化，之后基于自动布局约束和内容为每个单元格请求实际的尺寸。

21.3.2 修饰视图

使用修饰视图可以增强集合视图的显示效果，它独立于单元格和分节头/分节尾。修饰视图是独立于集合视图数据的，同集合视图的数据源和委托也没有关系。由于修饰视图可以置于集合视图的任何位置，因此在编程时一定要指明修饰视图的位置。基于这一点创建了一个名为 UICollectionViewFlowLayout 的子类，用于计算修饰视图的位置并在当前显示区域需要显示这个修饰视图时将其置于集合视图的合适位置。

在示例程序中，从上面的菜单中选择并点击 Custom Flow Layout 选项，查看一个有关修饰视图的示例。在照片下方出现的带有阴影的倾斜架子就是修饰视图，如图 21-9 所示。

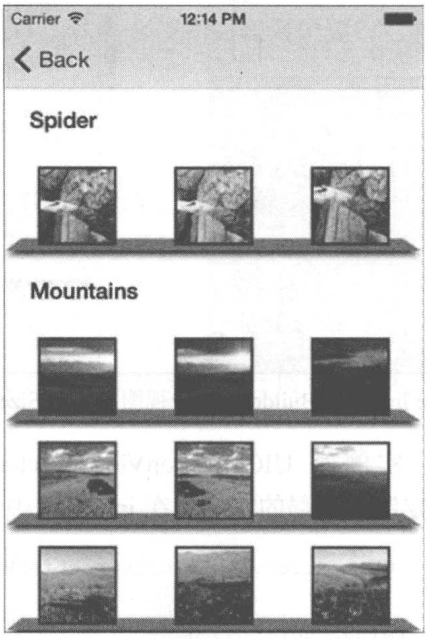


图 21-9 示例程序：定制流布局示例

使用修饰视图的第一步是要注册一个用于修饰视图的类或nib。在PHGCCustomFlowLayout中，在init方法中注册了一个名为PHGRowDecorationView的UICollectionReusableView子类。

```
[self registerClass:[PHGRowDecorationView class]
➥forDecorationViewOfKind:[PHGRowDecorationView kind]];
```

PHGCCustomFlowLayout具有定制绘制逻辑，可以绘制图21-9的架子和阴影。注意，如果需要，可以为集合视图注册多种修饰视图的类型，它们之间使用kind参数区分。在修饰视图类或nib完成注册之后，布局需要计算修饰视图应该出现的位置。为此，定制布局会覆盖prepareLayout方法，每当布局需要更新时就会调用这个方法。该方法会为每个需要的修饰视图计算边框的rects值，并将其保存在一个属性中，这样在需要时就可以取回这个值。

在prepareLayout方法中，首先调用[super prepareLayout]获取基本布局。之后再执行有关每行放置多少单元格的计算，这里假设这些单元格都是同一尺寸。

```
[super prepareLayout];

NSInteger sections = [self.collectionView numberOfSections];

CGFloat availableWidth = self.collectionViewContentSize.width -
➥(self.sectionInset.left + self.sectionInset.right);

NSInteger cellsPerRow =
➥floorf((availableWidth + self.minimumInteritemSpacing) /
➥(self.itemSize.width + self.minimumInteritemSpacing));
```

创建一个可变字典，保存为每个修饰视图计算得到的边框值，当执行计算时创建一个float对象来跟踪修饰视图在布局中的当前位置。

```
NSMutableDictionary *rowDecorationWork =
➥[[NSMutableDictionary alloc] init];

CGFloat yPosition = 0;
```

之后，下面的方法会遍历所有的分节，找到需要进行修饰的行。

```
for(NSInteger sectionIndex = 0; sectionIndex < sections; sectionIndex++)
{
...
}
```

在每个分节中，下面的方法会计算分节头需要占多少空间，以及在分节头和第一行的第一个单元格之间应该有多少空间。之后会根据单元格的数量计算该分节中需要设置多少行。

```
yPosition += self.headerReferenceSize.height;
yPosition += self.sectionInset.top;

NSInteger cellCount =
➥[self.collectionView numberOfItemsInSection:sectionIndex];
```

```
NSInteger rows = ceilf(cellCount/(CGFloat)cellsPerRow);
```

之后会遍历每一行，计算每一行修饰视图的大小，为行和分节创建索引路径。使用索引路径作为键来保存边框矩形到工作字典中，调整当前 y 轴位置，使其占尽量小的空间，除非该行为分节的最后行。

```
for(int row = 0; row < rows; row++)
{
    yPosition += self.itemSize.height;

    CGRect decorationFrame = CGRectMake(0,
    ➥yPosition-kDecorationYAdjustment,
    ➥self.collectionViewContentSize.width,
    ➥kDecorationHeight);

    NSIndexPath *decIndexPath = [NSIndexPath
    ➥indexPathForRow:row inSection:sectionIndex];

    rowDecorationWork[decIndexPath] =
    ➥[NSValue valueWithCGRect:decorationFrame];

    if(row < rows - 1)
        yPosition += self.minimumLineSpacing;
}
```

注意

修饰元素的索引路径不需要严格正确，因为布局只把它作为修饰视图中元素的唯一标识符。开发者可以使用任何能够描述修饰视图索引路径的方法，它对于集合视图中同样类型的修饰视图是唯一的。如果索引路径不唯一，会出现断言错误。

之后，该方法将对分节末尾所需的所有空间进行调整，包括分节插图和分节尾。

```
yPosition += self.sectionInset.bottom;
yPosition += self.footerReferenceSize.height;
```

在所有分节都设置完成后，带有修饰视图边框的字典对象会被保存在布局的属性中，在后面布局中会用到该属性。

```
self.rowDecorationRects =
➥[NSDictionary dictionaryWithDictionary:rowDecorationWork];
```

现在修饰视图的框架已经计算完毕，当集合视图在重写的 AttributesForElementsInRect: 方法中为可视区域请求布局特性时就可以使用该框架。首先该方法会从父类获取每一个单元格和分节头的特性，之后更新这些特性确保单元格在修饰视图的前面显示。

```
NSArray *layoutAttributes =
➥[super layoutAttributesForElementsInRect:rect];
```

```
for(UICollectionViewLayoutAttributes *attributes
    ➥in layoutAttributes)
{
   attributes.zIndex = 1;
}
```

该方法将会设置一个特性的可变拷贝，可以向它添加修饰视图所需的特性。之后迭代计算好修饰视图框架的字典对象，并检查哪个框架是在集合视图的可见区域。为这些修饰视图创建布局特性，并进行调整确保它们处于单元格视图的后面。最后返回经过更新的特性数组。

```
NSMutableArray *newLayoutAttributes =
➥[layoutAttributes mutableCopy];

[self.rowDecorationRects enumerateKeysAndObjectsUsingBlock:
 ^(NSIndexPath *indexPath, NSValue *rowRectValue, BOOL *stop) {

   if(CGRectIntersectsRect([rowRectValue CGRectValue], rect))
   {
      UICollectionViewLayoutAttributes *attributes =
      ➥[UICollectionViewLayoutAttributes
      ➥layoutAttributesForDecorationViewOfKind:
      ➥[PHGRowDecorationView kind] withIndexPath:indexPath];

      attributes.frame = [rowRectValue CGRectValue];
      attributes.zIndex = 0;
      [newLayoutAttributes addObject:attributes];
   }
}];

layoutAttributes = [NSArray arrayWithArray:newLayoutAttributes];

return layoutAttributes;
```

随着修饰视图的特性都被包含进整个布局特性集中，集合视图就可以显示这些修饰视图了，如图 21-9 所示。

21.4 创建定制布局

对于一些不适合网格模式的集合视图可以为它们创建定制布局。在示例程序的主菜单中点击 Custom Layout 就可以看到一个布局示例，它比之前的网格模式的布局要更复杂一些。这个布局会准备按一条不间断的正弦曲线路径来显示照片库中的图片，即使分节也不会中断该路径，如图 21-10 所示。

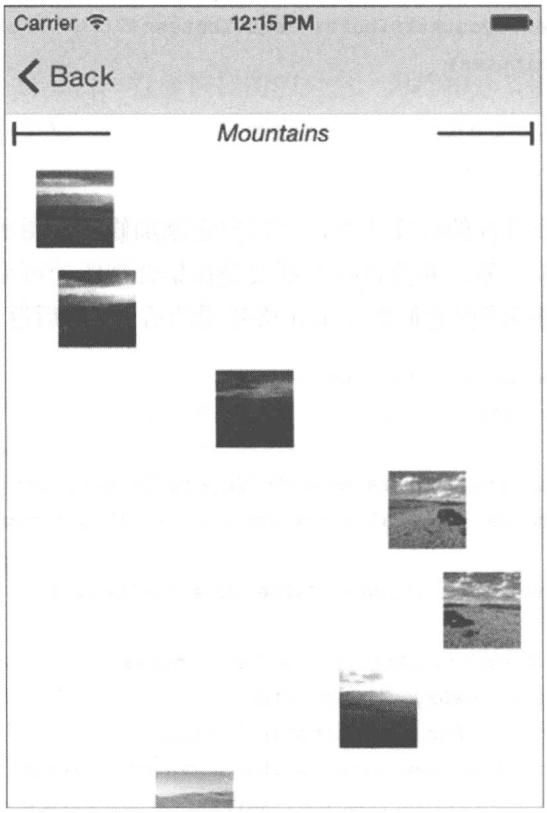


图 21-10 示例程序：带有正弦曲线布局的定制布局示例

要创建 UICollectionViewLayout 的一个子类，需要实现如下一些方法：

- collectionViewContentSize 方法会通知集合视图如何设置滑动视图的尺寸。
- layoutAttributesForElementsInRect:方法会通知集合视图在指定矩形区域内每个单元格对象、分节头、分节尾和修饰视图所需的所有布局特性。
- layoutAttributesForItemAtIndexPath:方法根据索引路径为单元格返回布局特性。
- layoutAttributesForSupplementaryViewOfKind:atIndexPath:方法根据索引路径为分节头和分节尾返回布局特性。如果集合视图没有用到分节头或分节尾，就不需要实现了。
- layoutAttributesForDecorationViewOfKind:atIndexPath:方法根据索引路径为修饰视图返回布局特性。如果集合视图没有用到修饰视图，就不需要实现了。
- shouldInvalidateLayoutForBoundsChange:方法为布局中的元素添加动画。如果方法返回 yes，集合视图将为可见区域重新计算所有的布局特性。这其实就是允许在屏幕上基于位置修改布局特性。
- prepareLayout 方法虽然是可选的，但它是计算布局的一个很好的地方，因为每当布局需要更新时就会调用这个方法。

在 PHGCustomLayout 中，prepareLayout 方法首先从确定显示多少个分节开始。创建一个浮点类型的变量用来在计算时追踪当前 y 轴位置，创建一个字典对象用于保存单元格的中点，并创建一个数组用于保存分节头的框架。

```
NSInteger numSections = [self.collectionView numberOfSections];
```

```
CGFloat currentYPosition = 0.0;
self.centerPointsForCells = [[NSMutableDictionary alloc] init];
self.rectsForSectionHeaders = [[NSMutableArray alloc] init];
```

该方法之后会对每个分节进行迭代处理，都会计算并保存分节头的框架，并在之后更新当前 y 轴位置，从上面一个计算得到的分节头到第一个需要显示的单元格之间的垂直中位线。之后确定分节中需要显示的单元格数量。

```
for(NSInteger sectionIndex = 0; sectionIndex < numSections;
    sectionIndex++)
{
   CGRect rectForNextSection = CGRectMake(0, currentYPosition,
   ➥self.collectionView.bounds.size.width, kSectionHeight);

   self.rectsForSectionHeaders[sectionIndex] =
   ➥[NSValue valueWithCGRect:rectForNextSection];

   currentYPosition +=
   ➥kSectionHeight + kVerticalSpace + kCellSize / 2;

   NSInteger numCellsForSection =
   ➥[self.collectionView numberOfItemsInSection:sectionIndex];
   ...
}
```

下一个方法将会对单元格进行迭代。它会使用正弦函数计算单元格的水平中位线，并在字典对象中保存中位线数据，使用索引路径作为每个单元格的键。该方法将会更新当前垂直位置并继续执行迭代过程。

```
for(NSInteger cellIndex = 0; cellIndex < numCellsForSection;
    cellIndex++)
{
   CGFloat xPosition =
   ➥[self calculateSineXPositionForY:currentYPosition];

   CGPoint cellCenterPoint =
   ➥CGPointMake(xPosition, currentYPosition);

   NSIndexPath *cellIndexPath = [NSIndexPath
   ➥indexPathForItem:cellIndex inSection:sectionIndex];

   self.centerPointsForCells[cellIndexPath] =
   ➥[NSValue valueWithCGPoint:cellCenterPoint];

   currentYPosition += kCellSize + kVerticalSpace;
}
```

在所有分节头框架和单元格中点位置都计算并保存好之后，如下方法将会计算并将集合

视图的内容尺寸保存在一个属性中，这样就可以从 collectionViewContentSize 方法返回相应的结果了。

```
self.contentSize =
➥CGSizeMake(self.collectionView.bounds.size.width,
➥currentYPosition + kVerticalSpace);
```

当显示集合视图时，会为集合视图的可视区域调用 layoutAttributesForElementsInRect:方法。这个方法会创建一个可变数组用于保存返回的特性，并迭代保存有分节框架的数组来确定应该显示哪个分节头。对每个要显示的分节头调用 layoutAttributesForSupplementaryView-OfKind:atIndexPath:方法来获取分节头的特性，并在工作数组(work array)中保存这些特性。

```
NSMutableArray *attributes = [NSMutableArray array];
for(NSValue *sectionRect in self.rectsForSectionHeaders)
{
    if(CGRectIntersectsRect(rect, sectionRect.CGRectValue))
    {
        NSInteger sectionIndex =
        ➥[self.rectsForSectionHeaders indexOfObject:sectionRect];

        NSIndexPath *secIndexPath =
        ➥[NSIndexPath indexPathForItem:0 inSection:sectionIndex];

        [attributes addObject:
        ➥[self layoutAttributesForSupplementaryViewOfKind:
        ➥UICollectionElementKindSectionHeader
        ➥atIndexPath:secIndexPath]];
    }
}
```

之后，该方法会迭代包含索引路径和单元格中点信息的字典对象，目的是确定显示哪些单元格，从 layoutAttributesForItemAtIndexPath:方法取回所需的单元格特性，并将这些特性保存到工作数组中。

```
[self.centerPointsForCells enumerateKeysAndObjectsUsingBlock:
 ➥^(NSIndexPath *indexPath, NSValue *centerPoint, BOOL *stop) {

    CGPoint center = [centerPoint CGPointValue];

    CGRect cellRect = CGRectMake(center.x - kCellSize/2,
    ➥center.y - kCellSize/2, kCellSize, kCellSize);

    if(CGRectIntersectsRect(rect, cellRect)) {
        [attributes addObject:
        ➥[self layoutAttributesForItemAtIndexPath:indexPath]];
    }
}];
```

要确定每个分节头的布局特性，layoutAttributesForSupplementaryViewOfKind:atIndexPath:

方法首先会得到一个关于分节头的默认特性集，通过调用 UICollectionViewLayoutAttributes 类方法 layoutAttributesForSupplementaryViewOfKind:withIndexPath:实现。之后该方法根据前面 prepareLayout 方法中计算好的框架信息更新分节头的框架尺寸和中点，并返回相应的特性值。

```
UICollectionViewLayoutAttributes *attributes =
➥[UICollectionViewLayoutAttributes
➥layoutAttributesForSupplementaryViewOfKind:
➥UICollectionElementKindSectionHeader withIndexPath:indexPath];

CGRect sectionRect =
➥[self.rectsForSectionHeaders[indexPath.section] CGRectValue];

attributes.size =
➥CGSizeMake(sectionRect.size.width, sectionRect.size.height);

attributes.center =
➥CGPointMake(CGRectGetMidX(sectionRect),
➥CGRectGetMidY(sectionRect));

return attributes;
```

要确定每个单元格的布局特性，layoutAttributesForItemWithIndexPath:方法会得到一个有关单元格的默认特性集，通过调用 UICollectionViewLayoutAttributes 类方法 layoutAttributes-ForCellWithIndexPath:实现。之后，该方法根据前面 prepareLayout 方法中计算好的中点值更新单元格的尺寸和中点，并返回相应的特性值。

```
UICollectionViewLayoutAttributes *attributes =
➥[UICollectionViewLayoutAttributes
➥layoutAttributesForCellWithIndexPath:path];

attributes.size = CGSizeMake(kCellSize, kCellSize);

NSValue *centerPointValue = self.centerPointsForCells[path];

attributes.center = [centerPointValue CGPointValue];
return attributes;
```

所有这些方法都实现完之后，集合视图就可以为视图中的所有元素计算位置了，并可以获取显示定制布局所需的所有位置信息，如图 21-10 所示。

21.5 集合视图动画

集合视图本身支持大量的动画效果。集合视图可以变换布局，将所有单元格从第一个布局的位置以动画效果移到另一个新的布局中。在布局中，当滑动视图时集合视图可以通过调整布局特性单独对每个单元格执行动画效果。在布局中对单元格的修改主要包括插入和删除两个操作，都可以使用动画。

21.5.1 集合视图布局切换

在示例程序中，在菜单中点击 Custom Flow 选项。对视图中的任何一张图片执行捏压手势操作，观察图片由一个布局切换另一个新布局的动画效果。屏幕上的所有单元格从原始位置移到新的位置，集合视图也发生了滑动并在视图中间显示被捏压操作的图片。这一效果的逻辑代码在 PHGCustomLayoutViewController 中。当视图控制器载入时，会创建两个捏压手势识别对象并保存在属性中。捏压放大操作的手势识别对象被添加到集合视图。要了解更多有关手势识别的内容，可以参见第 23 章“手势识别”。

```
self.pinchIn = [[UIPinchGestureRecognizer alloc]
                initWithTarget:self
                action:@selector(pinchInReceived:)];

self.pinchOut = [[UIPinchGestureRecognizer alloc]
                 initWithTarget:self
                 action:@selector(pinchOutReceived:)];

[self.collectionView addGestureRecognizer:self.pinchOut];
```

当收到捏压放大请求时，会调用 pinchOutReceived:方法。这个方法会检查手势的状态来判断正确的处理方法。如果状态是 UIGestureRecognizerStateBegan，该方法要判断哪个单元格被操作了，同时还要将其保存起来，这样当布局切换发生时就可以找到该元素了。

```
if(pinchRecognizer.state == UIGestureRecognizerStateBegan)
{
    CGPoint pinchPoint =
    ➥[pinchRecognizer locationInView:self.collectionView];

    self.pinchedIndexPath =
    ➥[self.collectionView indexPathForItemAtPoint:pinchPoint];
}
```

捏压手势完成后，会再次调用该方法，该方法会查看此时手势状态是否已经结束。如果已经结束，就会从视图中移除手势识别对象，避免在布局切换过程中不小心出现捏压动作，之后会创建新布局并初始化动画切换效果。当切换到新布局的动作完成时，该方法为其定义了一个 completion 代码块。这个 completion 代码块将会添加捏压缩小手势，这样用户就可以通过这个方法返回到之前的视图，并导航到之前用户实施捏压动作的单元格。

```
[self.collectionView removeGestureRecognizer:self.pinchOut];

UICollectionViewFlowLayout *individualLayout =
➥[[PHGAnimatingFlowLayout alloc] init];

__weak UICollectionView *weakCollectionView = self.collectionView;
__weak UIPinchGestureRecognizer *weakPinchIn = self.pinchIn;
__weak NSIndexPath *weakPinchedIndexPath = self.pinchedIndexPath;
void(^finishedBlock)(BOOL) = ^(BOOL finished) {
```

```
    [weakCollectionView scrollToItemAtIndexPath:weakPinchedIndexPath
    ➥atScrollPosition:UICollectionViewScrollPositionCenteredVertically
    ➥animated:YES];

    [weakCollectionView addGestureRecognizer:weakPinchIn];
};
[self.collectionView setCollectionViewLayout:individualLayout
                                    animated:YES
                                  completion:finishedBlock];
```

注意

所有动画都由集合视图处理，对于动画不可以使用定制的逻辑执行任何相关的计算。

21.5.2 集合视图布局动画

在对一个定制布局进行完捏压放大操作后，新出现的布局具有一个特征。每行中的单元格都变大了，并且沿 y 轴向越靠近视图中间，放大效果越明显，如图 21-11 所示。

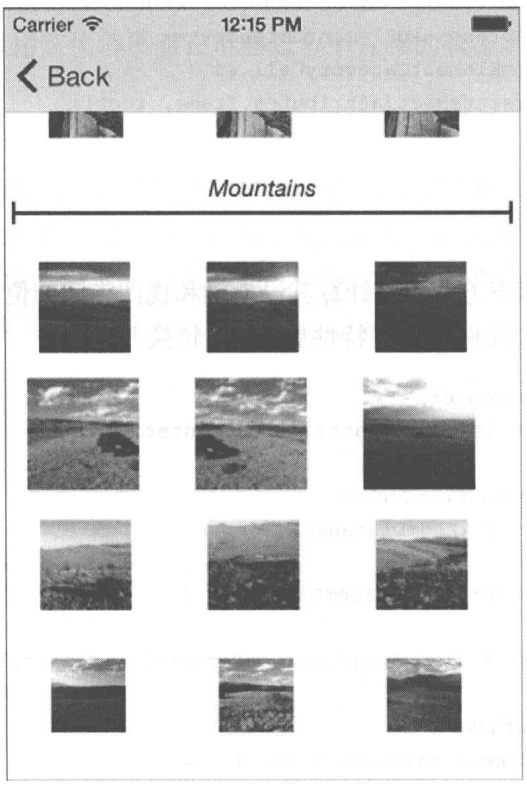


图 21-11 示例程序：定制布局动画示例

当用户滑动视图时，单元格尺寸会根据它们和视图中央的位置动态发生变化。要实现这样的效果，需要在 PHGAnimatingFlowLayout 类中实现一些定制逻辑。首先需要告诉布局当发生视图滑动时应该重新计算布局特性，这通过在 shouldInvalidateLayoutForBoundsChange: 方法中返回 YES 来实现。

```
-(BOOL)shouldInvalidateLayoutForBoundsChange:(CGRect)oldBounds
{
    return YES;
}
```

在滑动过程中当流布局失效时，会调用 layoutAttributesForElementsInRect:方法为每个可视单元格得到一个新的布局特性。这个方法将会为可视 rect 对象中的单元格确定对应的布局特性，这样就可以对其进行修改了。

```
NSArray *layoutAttributes =
➥[super layoutAttributesForElementsInRect:rect];

CGRect visibleRect;
visibleRect.origin = self.collectionView.contentOffset;
visibleRect.size = self.collectionView.bounds.size;

for(UICollectionViewLayoutAttributes *attributes
    ➥in layoutAttributes)
{
    if(attributes.representedElementCategory ==
       ➥UICollectionElementCategoryCell &&
       ➥CGRectIntersectsRect(attributes.frame, rect))
    {
        ...
    }
}
```

对于每个单元格，以下方法将会计算其沿 y 轴和视图中间的位置有多远。之后基于这个距离计算对单元格的放大比例。布局特性更新 3D 转换并返回。

```
CGFloat distanceFromCenter =
➥CGRectGetMidY(visibleRect) - attributes.center.y;

CGFloat distancePercentFromCenter =
➥distanceFromCenter / kZoomDistance;

if(ABS(distanceFromCenter) < kZoomDistance) {
    CGFloat zoom =
    ➥1 + kZoomAmount * (1 - ABS(distancePercentFromCenter));

    attributes.transform3D =
    ➥CATransform3DMakeScale(zoom, zoom, 1.0);
}
else
{
    attributes.transform3D = CATransform3DIdentity;
}
```

21.5.3 集合视图变化动画

当向视图中插入或删除元素时，集合视图支持使用动画效果来实现。示例程序没有演示这一动画，在这里对其进行一些讨论。要创建插入和删除操作的动画效果，需要在集合视图布局子类中实现一些方法。首先就是 prepareForCollectionViewUpdates:方法，用于在动画出现前做好相应的准备工作。这个方法接收由可以检查的更新内容组成的数组对象，这样就会根据不同的元素和更新类型进行定制准备了。

对于插入操作，可以实现 initialLayoutAttributesForAppearingItemAtIndexPath:方法。这个方法用于告诉布局在使用动画计算元素放置位置前应该在什么位置显示该元素。此外，所有其他分配给元素的初始化特性都会以动画方式变为最终的布局特性，这意味着元素可以被拉伸、旋转或是文件中包含的任何其他动作。

对于删除操作，可以实现 finalLayoutAttributesForDisappearingItemAtIndexPath:方法。这个方法用于告诉布局当元素以动画效果从布局中删除时最终的位置应该在哪里。同样，任何其他最终特性都可以被分配给元素，为其添加动画效果。

最后，可以实现 finalizeCollectionViewUpdates 方法。当所有的插入和删除操作完成时会执行这个方法，所以可以用它清理所有在准备过程中保存的状态。

21.6 小结

本章介绍了集合视图的概念，介绍了如何通过最少的编码量实现一个基本的集合视图，之后又介绍了一些更加高级的定制方法，包括定制流布局、修饰视图和完全定制布局。本章讨论了集合视图支持哪些动画选项，以及在布局切换过程中、滑动集合视图时和插入或删除元素时如何实现这些动画效果。

第22章

TextKit 介绍

iPhone 和 iPad 从一开始就支持许多用于文本呈现的元素。从 OS 发布之日起就有了文本框、标签、文本视图和 Web 视图。随着时间的推移，开发者对文本的处理有了更灵活、更丰富的功能需求，这些类也就不断在扩展和优化。

在 iOS 出现早期(即 iPhone OS)，显示特性文本唯一可行的方法就是使用 UIWebView 和利用 HTML 来处理定制特性。不过这种方法实现起来很困难，且性能很糟糕。iOS 3.2 中加入了 Core Text，它将 Mac 平台的 NSAttributedString 的全部功能带到了移动平台。不过 Core Text 有些复杂不太实用，并且开发者如果不是从 Mac 平台过渡来的话学习起来有很大难度，仅仅为了一个应用的文本处理功能而花大量的时间来研究这个方法并不明智。

TextKit 的第一次发布是作为 iOS 7 的一部分，TextKit 不是一个传统意义上的框架。相反，TextKit 是对于更好地处理文本展示对象及其特性的一组增强功能在术语上的表示。虽然 TextKit 加入了一些 Core Text 所没有的新特性和新功能，但 TextKit 的很多功能都是重新创建的，它们使用起来很简单。现存的 Core Text 代码很容易移植到 TextKit，通常都不需要做任何修改或者通过一些免费的工具仅做很小的修改即可。

本章介绍 TextKit，将会演示一些 iOS 7 版本下文本处理的基本方法，不过对于当前设备来说文本处理是一个庞大的话题，可以独立出版一本书介绍了。苹果公司花费了大量的时间和努力使一些高级的文本布局和处理变得更加简单。这里介绍的技术和工具可以作为开发者学习复杂文本呈现技术的一个很好的跳板。

22.1 示例程序

本章的示例程序(如图 22-1 所示)是一个简单的基于表视图的应用，用户可以在其中使用 4 种最常见的 TextKit 功能。示例程序没有直接使用 TextKit 的新功能，所以理解起来很容易。它包含一个 UINavigationController 上的主视图和一个用于选择 4 种 TextKit 功能的表视图。

示例程序提供的演示包括 Dynamic Link Detection，它将自动检测并突出显示不同的数据类型。另外，Hit Detection 可以让用户从 UITextView 中选择一个单词。Content Specific Highlighting 可以演示 TextKit 有关特性字符串处理的功能。最后，示例程序还会显示 Exclusion Paths，它提供文本环绕对象的效果或 Bézier 路径的功能。

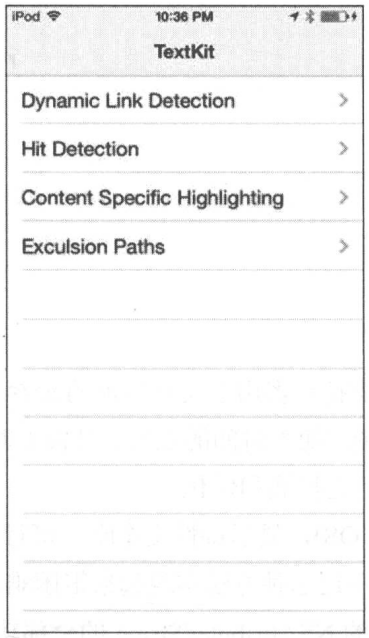


图 22-1 示例程序中带有 4 个不同 TextKit 功能选项的表视图

22.2 NSLayoutManager 介绍

NSLayoutManager 第一次出现是在 iOS 7 版本中，作为 TextKit 补充的一部分。它可以用于调节 NSTextStore 中布局和字符显示的问题，在下一小节会针对这个问题做详细讨论。NSLayoutManager 可以用于同时渲染多个 NSTextViews 视图以创建复杂的文本布局。NSLayoutManager 包含许多有关添加、移除、对齐或 NSTextContainer 相关的类，在后面的小节中会有更深入的介绍。

NSTextStore

每个 NSLayoutManager 都有一个相关联的 NSTextStorage 来扮演 NSMutableAttributedString 的子类。比较熟悉 Core Text 或 Mac OS X 文本渲染技术的读者可能会对特性字符串很了解，它用于保存具有一定风格的文本。NSTextStorage 为从文本添加或移除特性提供了一种非常容易的封装方法。

NSTextStorage 可用于 setAttributes:range:方法中，向字符串添加一个新特性。特性列表可以参见表 22-1。要查询当前可用特性，可以使用 attributesAtIndex:effectiveRange:方法。

表 22-1　可用的文本特性

特　性	描　述
NSFontAttributeName	文本的字体名，默认为 Helvetica(Neue) 12
NSParagraphStyleAttributeName	文本的段落风格，可以使用 NSParagraphStyle 常量。默认为 defaultParagraphStyle
NSForegroundColorAttributeName	文本字母的颜色，可以使用 UIColor，默认为 blackColor
NSBackgroundColorAttributeName	文本背景的颜色，可以使用 UIColor，默认为 nil，也就是没有任何颜色
NSLigatureAttributeName	NSNumber 对象表示文本是否可以连写，可以为 1，不可以为 0，默认为不可以
NSKernAttributeName	NSNumber 对象控制字间距。虽然这个特性在 iOS 中可以使用，不过除了 0 之外不支持其他值
NSStrikethroughStyleAttributeName	NSNumber 对象表示文本是否有删除线，有的话为 1，没有为 0，默认为 0
NSUnderlineStyleAttributeName	NSNumber 对象表示文本是否有下划线，若有为 1，没有为 0，默认为 0
NSStrokeColorAttributeName	UIColor 对象表示文本笔画的颜色；默认为 nil，也就是说，使用和 NSForegroundColorAttributeName 一样的颜色
NSStrokeWidthAttributeName	浮点型 NSNumber 对象以百分比的形式表示字体尺寸的宽度。常用于创建轮廓线效果。默认为 0，即没有任何字体加粗。负值表示实心的加粗效果，正值表示空心加粗效果
NSShadowAttributeName	用于文本的阴影量；支持 NSShadow 使用的常量，默认为无阴影
NSTextEffectAttributeName	文本效果；从 iOS 7 开始除了 nil 之外只有一个值可以用，即 NSTextEffectLetterpressStyle
NSAttachmentAttributeName	NSTextAttachment 对象，它是一个由 UIImage 表示的 NSData 值，默认为 nil
NSLinkAttributeName	一个用于表示链接的 NSURL 或 NSString
NSBaselineOffsetAttributeName	NSNumber 对象，包含针对基线偏移量的浮点型值，默认为 0
NSUnderlineColorAttributeName	UIColor 对象表示下划线的颜色，默认为 nil，即同前景色一样
NSStrikethroughColorAttributeName	UIColor 对象表示删除线的颜色，默认为 nil，即同前景色一样
NSObliquenessAttributeName	NSNumber 对象，它是一个浮点型数值，用于控制字符的倾斜效果。默认为 0，即无倾斜

(续表)

特　性	描　述
NSExpansionAttributeName	NSNumber 对象，它是一个浮点型数值，用于控制字符的粗度，默认为 0，即不加粗
NSWritingDirectionAttributeName	一个表示覆盖效果的值，可以使用 NSWritingDirection 和 NSTextWritingDirection
NSVerticalGlyphFormAttributeName	NSNumber 对象表示当前为水平文本(用 0 表示)还是垂直文本(用 1 表示)

NSLayoutManagerDelegate

NSLayoutManager 还有一个相关的委托用于处理文本的渲染。处理分隔线最有用的一组方法集可以用来解决行和段落的分隔问题。此外，文本完成渲染后也可以使用这些方法。

NSTextContainer

NSTextContainer 是另一个在 iOS 7 的 TextKit 中新增的重要类。NSTextContainer 定义了一块文本放置区域；前面小节中讨论过的 NSLayoutManager 可以控制多个 NSTextContainer。NSTextContainer 在文本视图中支持多行显示、文本封装和重置尺寸。对于路径排除的支持我们会在后面的 22.5 节中讨论。

22.3 动态链接检测

Dynamic Link Detection(动态链接检测)实现起来非常简单，如果用户在文本视图中处理地址、URL、电话号码或日期，使用动态链接检测能够提供非常好的用户体验。激活这些属性最简单的方法就是在 Interface Builder 中对其进行设置，如图 22-2 所示。

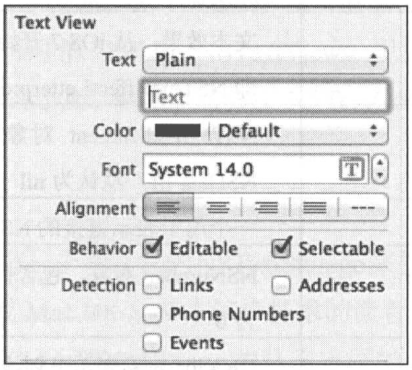


图 22-2　Xcode 6 中的 Dynamic Link Detection 控件

这些属性还可以使用代码进行开启或关闭。

```
[textView setDataDetectorTypes: UIDataDetectorTypePhoneNumber |
➥UIDataDetectorTypeLink | UIDataDetectorTypeAddress |
➥UIDataDetectorTypeCalendarEvent];
```

TextKit 添加了一个新的委托方法作为 UITextViewDelegate 的一部分，用来截获有关启动的事件。下面的例子会对一个 URL 检测启动 URL 事件并为用户弹出一个提醒框。

```
-(BOOL)textView:(UITextView *)textView shouldInteractWithURL:(NSURL
➥*)URL inRange:(NSRange)characterRange
{
    toBeLaunchedURL = URL;

    if([[URL absoluteString] hasPrefix:@"http://"])
    {
        UIAlertView *alert = [[UIAlertView alloc]
        ➥initWithTitle:@"URL Launching" message:[NSString
        ➥stringWithFormat:@"About to launch %@", [URL
        ➥absoluteString]] delegate:self
        ➥cancelButtonTitle:@"Cancel"
        ➥otherButtonTitles:@"Launch", nil];

        [alert show];
        return NO;
    }

    return YES;
}
```

22.4　检测点击

点击的检测用传统的方法实现起来很复杂，同时这个功能又是文本类应用最常用的功能之一。TextKit 添加了对每个字符进行点击检测的功能。要支持这个功能，需要创建一个 UITextView 子类，在示例程序中这个子类叫做 ICFCustomTextView。之后 UITextView 实现了 touchesBegan:事件方法。

当触碰开始时，捕获视图中的触碰位置，并沿 y 轴向下调整 10 个像素来选中文本元素。此时对文本视图属性 layoutManager 调用方法 characterIndexForPoint:inTextContainer: fractionOfDistanceBetweenInsertionPoints:。该方法返回所选字符的索引。

确定了字符索引之后，对于单词起止的确定是通过向前和向后查询最近一个空格来实现的。之后就可以在 UIAlertView 中为用户显示完整的单词了。

```
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    CGPoint touchPoint = [touch locationInView:self];

    touchPoint.y -= 10;
```

```
    NSInteger characterIndex = [self.layoutManager
    ➥characterIndexForPoint:touchPoint
    ➥inTextContainer:self.textContainer
    ➥fractionOfDistanceBetweenInsertionPoints:0];

    if(characterIndex != 0)
    {
        NSRange start = [self.text
        ➥rangeOfCharacterFromSet:[NSCharacterSet
        ➥whitespaceAndNewlineCharacterSet]
        ➥options:NSBackwardsSearch range:NSMakeRange(0,characterIndex)];

        NSRange stop = [self.text rangeOfCharacterFromSet:
        ➥[NSCharacterSet whitespaceAndNewlineCharacterSet]
        ➥options:NSCaseInsensitiveSearch
        ➥range:NSMakeRange(characterIndex,self.text.length-
        ➥characterIndex)];

        int length =  stop.location - start.location;

        NSString *fullWord = [self.text
        ➥substringWithRange:NSMakeRange(start.location, length)];


        UIAlertView *alert = [[UIAlertView alloc]
        ➥initWithTitle:@"Selected Word"
        ➥message:fullWord
        ➥delegate:nil
        ➥cancelButtonTitle:@"Dismiss"
        ➥otherButtonTitles:nil];

        [alert show];
    }

    [super touchesBegan:touches withEvent:event];
}
```

22.5 路径排除

路径排除(如图 22-3 所示)支持文本环绕图片或其他对象排列。TextKit 添加了一个简单的属性，从而为任意文本容器添加路径排除功能。

图 22-3　使用 iOS 7 的路径排除功能实现文本绕图片排列的效果

要指定一条排除路径，首先需要创建一个 UIBezierPath 对象，用来表示排除区域。要设置一条排除路径，需要将一个由排除区域对象组成的数组传递到 textContainer 的 exclusionPaths 属性。可以找到作为 UITextView 的属性出现的这个文本容器。

```
-(void)viewDidLoad
{
    [super viewDidLoad];

    UIBezierPath *circle = [UIBezierPath
    ➥bezierPathWithOvalInRect:CGRectMake(110, 100, 100, 102)];

    UIImageView *imageView = [[UIImageView alloc]
    ➥initWithFrame:CGRectMake(110, 110, 100, 102)];

    [imageView setImage: [UIImage imageNamed: @"DF.png"]];
    [imageView setContentMode:UIViewContentModeScaleToFill];
    [self.myTextView addSubview:imageView];

    self.myTextView.textContainer.exclusionPaths = @[circle];
}
```

22.6　Content Specific Highlighting 特性

TextKit 最有趣的特性之一就是 Content Specific Highlighting(指定内容突出显示)。在 iOS 7 之前要实现这个功能，需要使用 CoreText 修改文本视图中指定字符串的外观，这样做非常复杂且麻烦。TextKit 为本文的渲染和定义提供了许多改进方法。

要处理定制特性文本，需要创建一个 NSTextStorage 子类，该子类在示例程序中叫作

ICFDynamicTextStorage。这种方法可以让开发者为不同的需要渲染的特性字符串设置 token 标记。创建一个 NSMutableAttributedString 类级标签，它保存有关显示文本的所有特性。

```
-(id)init
{
    self = [super init];

    if(self)
    {
        backingStore = [[NSMutableAttributedString alloc] init];
    }

    return self;
}
```

还要创建一个用于返回字符串结果的便捷方法，以及一个用于返回带有索引值的特性结果的方法。

```
-(NSString *)string
{
    return [backingStore string];
}

-(NSDictionary *)attributesAtIndex:(NSUInteger)location
➥effectiveRange:(NSRangePointer)range
{
    return [backingStore attributesAtIndex:location
    ➥effectiveRange:range];
}
```

下面的 4 个方法用于实际处理特性的输入和设置，从替换字符到确保文本已经正确更新。

```
-(void)replaceCharactersInRange:(NSRange)range withString:(NSString
➥*)str
{
    [self beginEditing];
    [backingStore replaceCharactersInRange:range withString:str];

    [self edited:NSTextStorageEditedCharacters|
    ➥NSTextStorageEditedAttributes range:range
    ➥changeInLength:str.length - range.length];

    textNeedsUpdate = YES;
    [self endEditing];
}

-(void)setAttributes:(NSDictionary *)attrs range:(NSRange)range
{
    [self beginEditing];
    [backingStore setAttributes:attrs range:range];
```

```
    [self edited:NSTextStorageEditedAttributes range:range
    ➥changeInLength:0];

    [self endEditing];
}

-(void)performReplacementsForCharacterChangeInRange:
➥(NSRange)changedRange
{
    NSRange extendedRange = NSUnionRange(changedRange, [[self
    ➥string] lineRangeForRange:NSMakeRange(changedRange.location,
    ➥0)]);

    extendedRange = NSUnionRange(changedRange, [[self string]
    ➥lineRangeForRange:NSMakeRange(NSMaxRange(changedRange), 0)]);

    [self applyTokenAttributesToRange:extendedRange];
}

-(void)processEditing
{
    if(textNeedsUpdate)
    {
        textNeedsUpdate = NO;
        [self performReplacementsForCharacterChangeInRange:[self
        ➥editedRange]];
    }

    [super processEditing];
}
```

NSTextStore 子类中的最后一个方法将 NSTextStore 上使用属性设置的实际标签应用于字符串。标签作为一个 NSDictionary 对象传递，定义了应用操作的子串。当使用 enumerateSubstringsInRange:方法检测到子串时，使用前面介绍的 addAttribute:range:方法将特性应用到子串上。这个系统同时还允许在没有设置指定的特性时使用默认的 token 标签。

```
-(void)applyTokenAttributesToRange:(NSRange)searchRange
{
    NSDictionary *defaultAttributes = [self.tokens
    ➥objectForKey:defaultTokenName];

    [[self string] enumerateSubstringsInRange:searchRange
    ➥options:NSStringEnumerationByWords usingBlock:^(NSString
    ➥*substring, NSRange substringRange, NSRange enclosingRange,
    ➥BOOL *stop)
    {
        NSDictionary *attributesForToken = [self.tokens
        ➥objectForKey:substring];
```

```
        if(!attributesForToken)
        {
            attributesForToken = defaultAttributes;
        }

        [self addAttributes:attributesForToken
        ➥range:substringRange];

    }];
}
```

NSTextStore 子类创建完毕后，修改文本自身就变得简单了，修改结果见图 22-4。在创建一个新的 NSLayoutManager 实例之后，需要分配并初始化一个新的定制文本存储(text store)实例，最后创建一个 NSTextContainer 对象。文本容器被设置为可以共享文本视图的框架和边界，并在之后被添加到 layoutManager。之后，文本存储添加布局管理器。

新建一个 NSTextView 并将其设置为视图的框架，它的文本容器由之前创建的 NSTextView 设置。接下来，将文本视图的自动调整覆盖层配置为可以根据屏幕尺寸进行调整。最后，为文本视图配置可滑动的功能和键盘功能，并将文本视图添加为主视图的子视图。

定制文本区域的 tokens 属性用于为每个子串所分配的特性设置字典对象。第一个例子就是 Mary，用于设置 NSForegroundColorAttributeName 特性为红色。前面的表 22-1 已经给出一个完整的特性列表。例子演示了基于不同键值的多个特性类型。通过单词 was 的例子可以看出如何使用定制的字体、颜色和下划线同时向文本添加多个特性。同时还设置了一个默认的 token 标签，表示没有特殊处理的文本应该如何进行显示。

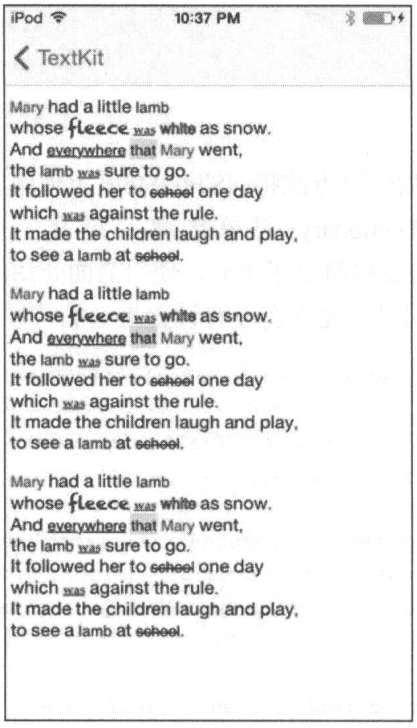


图 22-4 Content Specific Highlighting 特性：一些关键词已经更新了相应的特性

在所需的特性设置完成后，一些静态文本以“Mary Had a Little Lamb”诗歌的形式被添加到文本视图中；最终增加了特性效果的文本如图 22-4 所示。向文本视图输入内容会实时更新特性，并且对于已经配置好特性的内容来说，输入一部分内容就可以看到相应的效果。

```
-(void)viewDidLoad
{
    [super viewDidLoad];

    ICFDynamicTextStorage *textStorage = [[ICFDynamicTextStorage
    ➥alloc] init];

    NSLayoutManager *layoutManager = [[NSLayoutManager alloc] init];

    NSTextContainer *container = [[NSTextContainer alloc]
    ➥initWithSize:CGSizeMake(myTextView.frame.size.width,
    ➥CGFLOAT_MAX)];

    container.widthTracksTextView = YES;
    [layoutManager addTextContainer:container];
    [textStorage addLayoutManager:layoutManager];

    myTextView = [[UITextView alloc] initWithFrame:self.view.frame
    ➥textContainer:container];

    myTextView.autoresizingMask = UIViewAutoresizingFlexibleHeight |
    ➥UIViewAutoresizingFlexibleWidth;

    myTextView.scrollEnabled = YES;

    myTextView.keyboardDismissMode =
    UIScrollViewKeyboardDismissModeOnDrag;

    [self.view addSubview:myTextView];

    textStorage.tokens = @{ @"Mary":@{ NSForegroundColorAttributeName:
    ➥[UIColor redColor]},
    ➥@"lamb":@{ NSForegroundColorAttributeName:[UIColor blueColor]},
    ➥@"everywhere":@{ NSUnderlineStyleAttributeName:@1},
    ➥@"that":@{NSBackgroundColorAttributeName:[UIColor yellowColor]},
    ➥@"fleece":@{NSFontAttributeName:[UIFont
    ➥fontWithName:@"Chalkduster" size:14.0f]},
    ➥@"school":@{NSStrikethroughStyleAttributeName:@1},
    ➥@"white":@{NSStrokeWidthAttributeName:@5},
    ➥@"was":@{NSFontAttributeName:[UIFont fontWithName:@"Palatino-Bold"
    ➥size:10.0f], NSForegroundColorAttributeName:[UIColor purpleColor],
    ➥NSUnderlineStyleAttributeName:@1}, defaultTokenName:@{
    ➥NSForegroundColorAttributeName:[UIColor blackColor],
    ➥NSFontAttributeName:[UIFont systemFontOfSize:14.0f],
    ➥NSUnderlineStyleAttributeName:@0,
    ➥NSBackgroundColorAttributeName:[UIColor whiteColor],
```

```
    ➥NSStrikethroughStyleAttributeName:@0,
    ➥NSStrokeWidthAttributeName:@0}};

    NSString *maryText = @"Mary had a little lamb\nwhose fleece was
    ➥white as snow.\nAnd everywhere that Mary went,\nthe lamb was sure
    ➥to go.\nIt followed her to school one day\nwhich was against the
    ➥rule.\nIt made the children laugh and play,\nto see a lamb at
    ➥school.";

    [myTextView setText:[NSString stringWithFormat:@"%@\n\n%@\n\n%@",
    ➥maryText, maryText, maryText]];
}
```

22.7 使用 Dynamic Type 更改字体设置

TextKit 支持 Dynamic Type(动态类型)，可以让用户在操作系统级别指定字体大小。用户可以在 iOS 8 的 Settings 应用的 General 下访问 Dynamic Type 控件，如图 22-5 所示。用户修改了字体大小首选项之后，该应用会收到一个名为 UIContentSizeCategoryDidChangeNotification 的通知。可以监控这个通知来处理字体大小的更新事件。

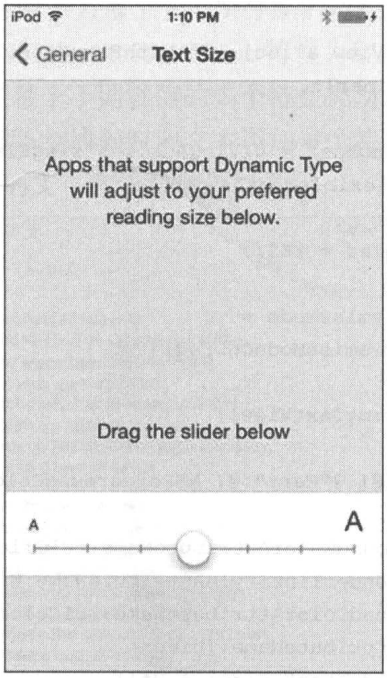


图 22-5 使用 iOS 7 的 Settings 应用中的 Dynamic Type 设置修改字体大小

```
[[NSNotificationCenter defaultCenter] addObserver:self
➥selector:@selector(preferredSizeDidChange:)
➥name:UIContentSizeCategoryDidChangeNotification object:nil];
```

要以用户首选项设置好的字体显示文本，需要使用 Font Text Styles(见表 22-2)中的一个特性对字体进行设置。

```
self.textLabel.font = [UIFont
➥preferredFontForTextStyle:UIFontTextStyleBody];
```

这样就会基于用户的设置返回一个大小合适的字体。

表 22-2　iOS 7 中定义的 Font Text Styles

特　性	描　述
UIFontTextStyleHeadline1	1 号标题
UIFontTextStyleHeadline2	2 号标题
UIFontTextStyleBody	正文文本
UIFontTextStyleSubheadline1	1 号副标题
UIFontTextStyleSubheadline2	2 号副标题
UIFontTextStyleFootnote	脚注
UIFontTextStyleCaption1	标准标题
UIFontTextStyleCaption2	替换标题

22.8　小结

在 TextKit 推出之前，iOS 的文本渲染是个有一定难度且复杂的话题。本章仅仅涉及 TextKit 强大功能的冰山一角，但愿本章的讨论不会像之前文本渲染那样吓退大家。

本章中用到了许多例子，从点击检测到处理特性字符串。此外，你应该掌握了创建文本渲染对象的基础知识。虽然文本渲染问题是一个很大的话题，甚至可以用一本书来讲解，不过本章介绍的内容为你后面的深入学习提供了坚实的基础。

第23章

手势识别

如果一个应用需要快速简单地处理点击、滑动、捏压和旋转等事件，那它会怎么做呢？回到曾经的年代(iPad 发布之前)，开发者必须有一个 UIView 子类，并实现 touchesBegan:/touchesMoved:/touchesEnded 方法，然后通过编写定制逻辑来判断这些动作发生的时间。实现所有这些步骤可能要花上一整天的时间。

iPad 发布时，苹果公司在 iOS 3.2 版本中推出了手势识别功能来满足上述需求。UIGestureRecognizer 是一个有关处理手势识别的通用架构抽象类。其中包括识别各种日常手势的具体实现方法，甚至还包括一些子类化方法，让你可以用同样的架构实现定制的手势。通过这些新类，实现一些复杂的手势操作比以前要容易很多。

23.1 手势识别的类型

苹果将手势识别定义为以下两大类：

- 离散型：离散型手势识别主要是指对快速简单的交互操作的处理，比如点击。这样，应用实际上需要知道只有该点击动作发生之后，才可以完成相应的动作。
- 连续型：连续型手势识别主要是指对手势动作持续过程中始终需要获取信息的交互操作的处理，比如捏压或旋转操作。在这些情况下，应用通常需要交互过程中的信息来处理 UI 的变化。例如，需要知道用户捏压操作的大小来调整视图，或者可能需要知道用户手指是如何旋转的，以此来匹配视图的旋转。

预定义的手势识别一共有 6 个，如表 23-1 中所示。它们的功能各异，可以处理我们熟知的大部分 iOS 标准手势交互动作。

表 23-1 内置的 UIGestureRecognizer 子类

类　名	类　型
UITapGestureRecognizer	离散型
UIPinchGestureRecognizer	连续型
UIPanGestureRecognizer	连续型

(续表)

类　名	类　型
UISwipeGestureRecognizer	离散型
UIRotationGestureRecognizer	连续型
UILongPressGestureRecognizer	连续型

23.2　基础手势识别的用法

创建一个基础的手势识别对象很简单。通常手势识别对象是在视图控制器中创建的，并在控制器类中实现具体的方法来完成相应的操作。唯一需要确认的是点击识别对象所作用的视图，以及该对象所调用的方法。

```
UITapGestureRecognizer *tapRecognizer =
➥[[UITapGestureRecognizer alloc] initWithTarget:self
➥action:@selector(myGestureViewTapped:)];

[myGestureView addGestureRecognizer:tapRecognizer];
```

一些手势识别对象可以接收更多的参数来重新定义动作，不过对于大多数情况，手势一般只作用于一个视图和方法。当手势在一个特定的视图中被识别时，会调用带有手势识别对象的方法。

手势识别对象可以很容易地在 storyboard 中创建。要添加一个手势识别对象，首先在观察 storyboard 时在 Object Library 中找到所需的识别类型。将一个识别对象拖到接收视图中，该视图就是接收并处理该手势识别对象的视图。当向视图中添加识别对象时，它会出现在视图控制器场景的提示栏中。之后，手势识别对象可以在 storyboard 中和其他视图一样进行配置，如图 23-1 所示。可以为其分配属性，可以为方法调用分配动作选择器，或者可以触发场景切换。

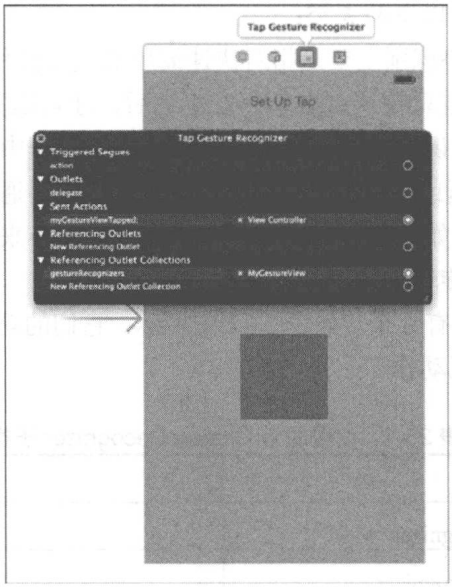


图 23-1　在 storyboard 中配置 Tap Gesture Recognizer 对象的示例

23.3　示例程序介绍

本章的示例程序 Gesture Playground 中只包含一个名为 myGestureView 的视图(如图 23-2 所示)，可以通过手势来操作它。点击相关的按钮为每个示例配置手势识别。注意，示例项目使用的 storyboard 是禁用 Auto Layout 的，这会让示例变得简单且易于理解，同时避免了布局因为手势变化而导致的潜在错误。首先在 Xcode 中打开这个项目。

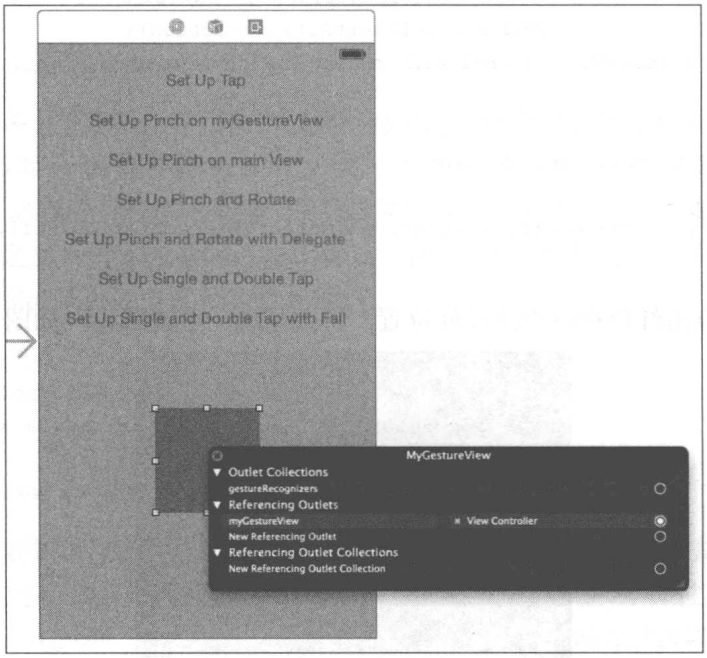


图 23-2　在 storyboard 中查看 Gesture Playground 的视图控制器

23.3.1　点击识别动作

设置点击手势的代码在示例程序的视图控制器的 setUpTapGestureRecognizer:方法中实现。要执行该代码，运行应用并点击标题为 Set Up Tap 的按钮。该方法首先会从主视图和 myGestureView 中清理所有的手势识别对象。之后在点击动作被识别后为 myGestureTapped:方法设置一个点击识别对象。

```
[self clearAllGestureRecognizers];

UITapGestureRecognizer *tapRecognizer =
➥[[UITapGestureRecognizer alloc] initWithTarget:self
                                  action:@selector(myGestureViewTapped:)];

[self.myGestureView addGestureRecognizer:tapRecognizer];
```

因为点击手势是一个离散型手势，所以只有在手势被识别后才会调用 myGestureViewTapped:方法，之后显示一个提醒框：

```
-(void)myGestureViewTapped:(UIGestureRecognizer *)tapGestureRecognizer {
```

```
    UIAlertController *alert =
    ➥[UIAlertController alertControllerWithTitle:@"Tap Received"
                                          message:@"Received tap in myGestureView"
                                   preferredStyle:UIAlertControllerStyleAlert];

    UIAlertAction *dismissAction =
    ➥[UIAlertAction actionWithTitle:@"OK, Thanks"
                               style:UIAlertActionStyleCancel
                             handler:^(UIAlertAction *action){
        [self dismissViewControllerAnimated:YES completion:nil];
    }];

    [alert addAction:dismissAction];

    [self presentViewController:alert animated:YES completion:nil];
}
```

运行项目并点击红色视图中的任何位置，将会呈现一个提醒框，如图 23-3 所示。

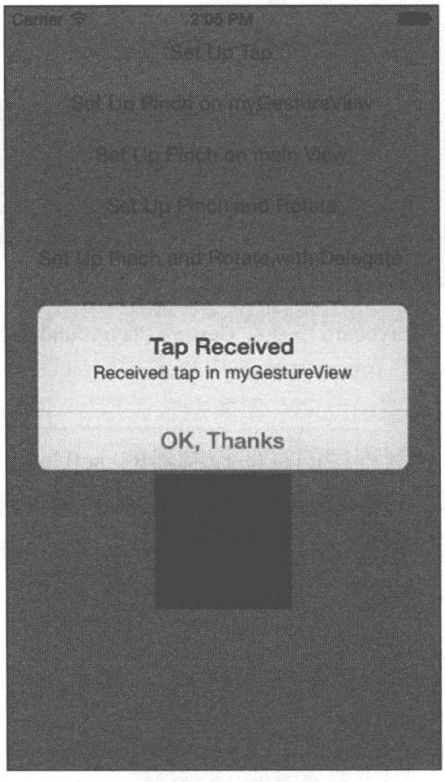


图 23-3 识别单击动作

尝试点击视图之外的区域，注意，此时不会显示提醒框，除非点击动作确实出现在视图中。

点击识别和按钮

那么为什么不能用按钮呢？它不是更加快速和简单，并且不需要代码就可以在 Interface Builder 中创建吗？的确，在大部分情况下，使用 UIButton 是最佳选择。不过还有许多情况下使用点击识别更理想。例如，当有许多文本输入区域需要在键盘中上下滑动时，你希望点击随意位置就可以取消键盘。如果将所有文本输入框都置于一个 UIView 中，可以向该视图添加点击识别对象，这样就可以很容易地将键盘取消掉。

23.3.2 捏压识别动作

当用户在屏幕上使用两个手指向内或向外移动时，就会用到捏压识别动作。两个手指间位置的变化可以用于调整视图或图片的大小。在示例程序中，点击 Set Up Pinch on myGestureView，向视图中添加一个捏压手势识别对象。所调用的方法会清除所有已经存在的手势识别对象，之后再向 myGestureView 添加一个捏压手势识别对象，并为该手势关联一个 myGestureViewSoloPinched:方法作为目标动作。

```
UIPinchGestureRecognizer *soloPinchRecognizer =
➥[[UIPinchGestureRecognizer alloc] initWithTarget:self
➥action:@selector(myGestureViewSoloPinched:)];

[myGestureView addGestureRecognizer:soloPinchRecognizer];
```

该方法一定要知道捏压手势的缩放程度。好消息是 iOS 将会在此方法中将一个关于该数据的引用传递到手势识别对象，所以需要一个实例变量或属性来保存此数据。UIPinchGestureRecognizer 实例还包含一个名为 scale 的方法，该方法可以在视图上设置一个映射变换，这被认为是一种非常好的方法。

```
-(void)myGestureViewSoloPinched:(UIPinchGestureRecognizer *)
➥pinchGesture {
    CGFloat pinchScale = [pinchGesture scale];

    CGAffineTransform scaleTransform =
    ➥CGAffineTransformMakeScale(pinchScale, pinchScale);

    [myGestureView setTransform:scaleTransform];
}
```

运行项目，在视图中向内和向外捏压手指，注意，视图随着捏压动作不断地调整。

注意

要在 iOS 模拟器中执行两个手指的捏压动作，按住 Option 键并注意显示的两个圆圈，它们代表两个手指。随着鼠标指针的移动，手指间的距离将变大或变小。两个手指的中心点就是应用视图的中心点。不过在 Gesture Playground 中这么做有些不方便，因为 myGestureView 靠近屏幕的底部。要重新调整中心点的位置，按住 Option 键的同时按住 Shift 键并移动鼠标指针。

如果视图很小不好操作怎么办？将手势识别添加到父视图，它可以在该视图的任何位置获取捏压动作。要查看这个方法的具体效果，可以点击 SetUp Pinch on main View 按钮。首先会清除所有已经存在的手势识别对象，并向视图控制器的视图添加捏压手势识别对象，这样在视图中的任何位置都可以获得该捏压手势：

```
UIPinchGestureRecognizer *soloPinchRecognizer =
➥[[UIPinchGestureRecognizer alloc] initWithTarget:self
➥action:@selector(myGestureViewSoloPinched:)];

[[self view] addGestureRecognizer:soloPinchRecognizer];
```

运行项目，注意，应用中任何位置出现的捏压操作都会对 myGestureView 产生影响。这其实是手势识别被低估的一个功能：因为这个功能可以从受影响的视图中很容易地复制点击动作。一个点击动作可以在应用中的任何位置被检测到，可以使用或转换此点击数据来影响其他视图。如果要更加精确的话，可以使用 locationInView:方法检查点击手势发生的位置，以此来确定它是否和要处理的视图足够近。

23.4 在一个视图中识别多个手势

很多时候需要在一个视图中识别多个手势，例如，用户希望能够同时对 myGestureView 进行拉伸和旋转操作。为此，示例程序添加了一个旋转手势识别对象，观察它是如何同捏压手势进行互动的。点击 Set Up Pinch and Rotate 按钮，首先会清除其他手势识别对象，并添加捏压识别对象和旋转识别对象：

```
UIPinchGestureRecognizer *pinchRecognizer =
➥[[UIPinchGestureRecognizer alloc] initWithTarget:self
➥action:@selector(myGestureViewPinched:)];

[myGestureView addGestureRecognizer:pinchRecognizer];

UIRotationGestureRecognizer *rotateRecognizer =
➥[[UIRotationGestureRecognizer alloc] initWithTarget:self
➥action:@selector(myGestureViewRotated:)];

[myGestureView addGestureRecognizer:rotateRecognizer];
```

现在要同时处理旋转和拉伸，需要对 myGestureView 应用一个新的方法来建立一个关联映射变换。为了实现该方法，需要保存最终的和当前的拉伸情况以及旋转因子，这样在手势变化的过程中就不会漏掉数据。注意，这些属性都已经在视图控制器中创建完毕。

```
@property (nonatomic, assign) CGFloat scaleFactor;
@property (nonatomic, assign) CGFloat rotationFactor;
@property (nonatomic, assign) CGFloat currentScaleDelta;
@property (nonatomic, assign) CGFloat currentRotationDelta;
```

setUpPinchAndRotationGestureRecognizers:方法会初始化 scaleFactor 和 rotationFactor，以

避免在视图最初调整大小时出现闪烁的情况。

```
[self setScaleFactor:1.0];
[self setRotationFactor:0.0];
```

myGestureViewRotated:方法会处理旋转手势识别对象，它包含一个名为 rotation 的属性，可以让你知道用户所旋转的角度：

```
-(void)myGestureViewRotated:(UIRotationGestureRecognizer *)
➥rotateGesture {
    CGFloat newRotateRadians = [rotateGesture rotation];

    [self updateViewTransformWithScaleDelta:0.0
    ➥andRotationDelta:newRotateRadians];
    if([rotateGesture state] == UIGestureRecognizerStateEnded) {
        CGFloat saveRotation = [selfrotationFactor] +
        ➥newRotateRadians;
        [self setRotationFactor:saveRotation];
        [self setCurrentRotationDelta:0.0];
    }
}
```

这个方法会从手势识别对象获取旋转的角度，以弧度来表示。之后调用一个定制方法为视图创建拉伸和旋转映射变换。如果触碰事件结束，该方法会基于当前状态和新的旋转角度计算最后的旋转角度，并将数据保存到旋转因子属性中。之后，该方法会清除所计算的旋转增量，以避免两个触碰之间旋转变换不正常的情况发生。创建拉伸和旋转映射的方法如下：

```
-(void)updateViewTransformWithScaleDelta:(CGFloat)scaleDelta
➥andRotationDelta:(CGFloat)rotationDelta;
{
    if(rotationDelta != 0) {
        [self setCurrentRotationDelta:rotationDelta];
    }
    if(scaleDelta != 0) {
        [self setCurrentScaleDelta:scaleDelta];
    }
    CGFloat scaleAmount = [self scaleFactor]+[self currentScaleDelta];

    CGAffineTransform scaleTransform =
    ➥CGAffineTransformMakeScale(scaleAmount, scaleAmount);

    CGFloat rotationAmount =
    ➥[self rotationFactor]+[self currentRotationDelta];

    CGAffineTransform rotateTransform =
    ➥CGAffineTransformMakeRotation(rotationAmount);

    CGAffineTransform newTransform =
    ➥CGAffineTransformConcat(scaleTransform, rotateTransform);
```

```
    [myGestureView setTransform:newTransform];
}
```

这个方法会从触碰开始正确计算拉伸的变化和旋转的变化，会检查拉伸和旋转的大小是否为 0，手势识别对象会从触碰开始的地方计算并返回拉伸和旋转的大小。这个所谓的大小就是增量。因为当触碰开始时视图需要保持当前的状态，所以不能立即应用增量，而是将增量添加到当前状态，从而避免视图乱跳。

运行 Gesture Playground，用两个手指触碰界面并旋转，观察视图的变化。同时注意，仍然可以使用捏压动作，不过两者不能同时进行，本章后面会解释为什么会这样。首先关注手势识别是如何处理触碰事件的。

23.4.1 手势识别的工作原理

现在已经演示了基础的手势识别在实际中的应用，可以继续向前探索更为详细的内容，即手势识别对象是如何工作的。

第一个需要理解的概念是手势识别操作是在常规视图响应链之外的。首先 UIWindow 会将触碰事件发送到手势识别对象，它们必须指明无法处理该事件，之后该事件才会默认向前传到视图响应链。

接下来，当应用试图确定是否有手势动作被识别时，理解事件发生的顺序非常重要。

(1) 窗口会将触碰事件发送到手势识别对象。

(2) 手势识别对象将进入 UIGestureRecognizerStatePossible 状态。

(3) 对于离散型手势，手势识别对象会判断它是 UIGestureRecognizerStateRecognized 还是 UIGestureRecognizerStateFailed 类型。

(4) 如果是 UIGestureRecognizerStateRecognized，手势识别接收触碰时间并调用指定的委托方法。

(5) 如果是 UIGestureRecognizerStateFailed，手势识别对象将触碰事件返回到响应链。

(6) 对于连续型手势，手势识别对象会判断它是 UIGestureRecognizerStateBegan 还是 UIGestureRecognizerStateFailed。

(7) 如果是 UIGestureRecognizerStateBegan，手势识别对象会接收触控事件并调用指定的委托方法。之后每当手势发生变化时就将状态更新为 UIGestureRecognizerStateChanged，并始终调用委托方法，直到最后的触碰事件结束，此时状态为 UIGestureRecognizerStateEnded。如果触碰模式不再和期望的手势相匹配，可以将状态改为 UIGestureRecognizerStateCancelled。

(8) 如果是 UIGestureRecognizerStateFailed，手势识别对象将触碰事件返回到响应链。

注意，UIGestureRecognizerStatePossible 和 UIGestureRecognizerStateFailed 这两个状态之间消耗的时间很久。如果用户界面中的手势识别对象由于触碰动作导致运行缓慢而无法解释，问题可能就出在这里。最好的办法是在具体的处理方法中添加记录——每次方法被调用时就记录状态。之后就可以根据记录的时间戳信息很清楚地了解状态变化的过程，也就可以很好地判断延迟问题出在哪里。

23.4.2 在一个视图中识别多个手势：Redux

现在，本章已经介绍了手势识别对象如何接收和处理触碰事件，显然之前的介绍是在同一时刻只接收和处理一个手势识别对象。要同步处理两个手势识别对象，需要实现UIGestureRecognizerDelegate 协议，对触碰动作如何发送到手势识别对象进行更加复杂的控制。这个协议指定了如下 3 个方法：

- (BOOL)gestureRecognizerShouldBegin:(UIGestureRecognizer *)gestureRecognizer: 使用这个方法来表示手势识别对象根据应用的状态，是否应该从 UIGestureRecognizer-StatePossible 转换为 UIGestureRecognizerStateBegan 状态。如果返回 YES，进行手势识别对象的处理；否则状态变为 UIGestureRecognizerStateFailed。
- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizershouldReceiveTouch:(UITouch *)touch: 使用这个方法来表示手势识别对象是否应该接收一个触碰动作。这个方法可以让手势识别根据用户设置的一些约束条件拒绝识别手势动作。
- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizershouldRecognize-SimultaneouslyWithGestureRecognizer:(UIGestureRecognizer*)otherGestureRecognizer: 当有超过一个手势识别对象同步接收触碰事件时使用这个方法。返回 YES 则同步处理所有操作，或者对传入的手势识别对象进行测试，判断这些动作是否满足同步处理的要求。

示例程序实现 shouldRecognizeSimultaneouslyWithGestureRecognizer:方法来完成同步处理捏压和旋转手势的功能：

```
-(BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer
➥shouldRecognizeSimultaneouslyWithGestureRecognizer:
(UIGestureRecognizer *)otherGestureRecognizer
{
    return YES;
}
```

要查看具体效果，点击示例程序中的 Set Up Pinch and Rotate with Delegate 按钮。这个方法会像之前一样先创建捏压和旋转手势识别对象，不过还会为它们设置委托，这样才会调用shouldRecognizeSimultaneouslyWithGestureRecognizer:方法。

```
UIPinchGestureRecognizer *pinchRecognizer =
➥[[UIPinchGestureRecognizer alloc] initWithTarget:self
➥action:@selector(myGestureViewPinched:)];

[pinchRecognizer setDelegate:self];
[[self view] addGestureRecognizer:pinchRecognizer];

UIRotationGestureRecognizer *rotateRecognizer =
➥[[UIRotationGestureRecognizer alloc] initWithTarget:self
➥action:@selector(myGestureViewRotated:)];

[rotateRecognizer setDelegate:self];
```

```
[[self view] addGestureRecognizer:rotateRecognizer];
```

运行 Gesture Playground 并用两个手指进行捏压和旋转操作。视图现在会很平滑地进行拉伸和旋转了，如图 23-4 所示。

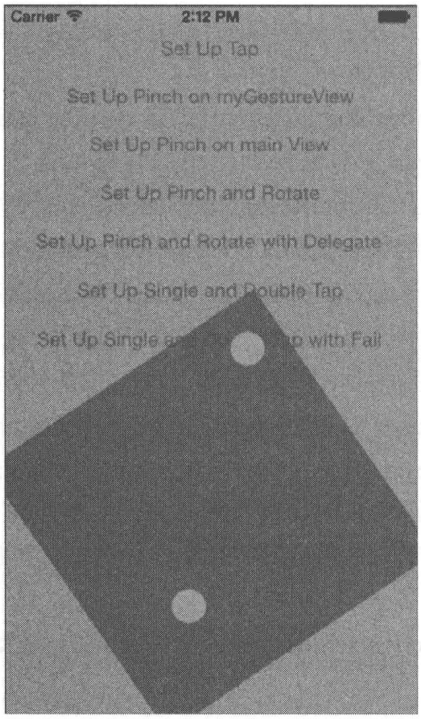


图 23-4 同步旋转和拉伸

23.4.3 请求手势识别失败

在一些情况下，为了满足应用的一些需求，也许会令手势识别失败。一个比较容易理解的示例就是当在同一视图中同时需要单击和双击操作时。默认情况下，如果在同一个视图中带有单击手势识别对象和双击手势识别对象，即使用户手势为双击操作也会激活单击识别对象。也就是说，单击和双击操作对应的方法都会被调用。要在示例程序中查看这个现象，点击 SetUp Single and Double Tap 按钮。所调用的方法会清除所有已经存在的手势识别对象，并创建和配置关联 myGestureView 的双击手势识别对象。

```
UITapGestureRecognizer *doubleTapRecognizer =
➥[[UITapGestureRecognizer alloc] initWithTarget:self
➥action:@selector(myGestureViewDoubleTapped:)];

[doubleTapRecognizer setNumberOfTapsRequired:2];
[myGestureView addGestureRecognizer:doubleTapRecognizer];

UITapGestureRecognizer *singleTapRecognizer =
➥[[UITapGestureRecognizer alloc] initWithTarget:self
➥action:@selector(myGestureViewTapped:)];

[myGestureView addGestureRecognizer:singleTapRecognizer];
```

注意，项目中调用的处理方法会使用 NSLog 语句而不是之前的 UIAlertView。如果使用 UIAlertView，可能会因为锁住用户界面而无法实现双击。

```
-(void)myGestureViewSingleTapped:(UIGestureRecognizer *)
➥tapGestureRecognizer {
    NSLog(@"Single Tap Received");
}

-(void)myGestureViewDoubleTapped:(UIGestureRecognizer *)
➥doubleTapGestureRecognizer {
    NSLog(@"Double Tap Received");
}
```

当双击操作发生时会调用单击和双击两个方法：

2014-08-04 14:00:45.299 GesturePlayground[38536:2398989] Single Tap Received

2014-08-04 14:00:45.476 GesturePlayground[38536:2398989] Double Tap Received

如果不希望这样，应该让双击识别对象在单击方法之前就失效，对 UIGestureRecognizer 使用一个名为 requireGestureRecognizerToFail 的方法。要解决这个问题，可以使用下面的步骤：

(1) 创建双击识别对象。

(2) 创建单击识别对象。

(3) 从单击识别对象调用 requireGestureRecognizerToFail:，将双击识别对象作为参数传递给该方法。

要查看这一效果，在示例程序中点击 Set Up Single and Double Tap with Fail 按钮。它会像之前那样创建一个单击手势识别对象和一个双击手势识别对象，不过对于单击手势识别对象还会包含 requireGestureRecognizerToFail:调用。

```
UITapGestureRecognizer *doubleTapRecognizer =
➥[[UITapGestureRecognizer alloc] initWithTarget:self
➥action:@selector(myGestureViewDoubleTapped:)];

[doubleTapRecognizer setNumberOfTapsRequired:2];
[myGestureView addGestureRecognizer:doubleTapRecognizer];

UITapGestureRecognizer *singleTapRecognizer =
➥[[UITapGestureRecognizer alloc] initWithTarget:self
➥action:@selector(myGestureViewTapped:)];

[singleTapRecognizer requireGestureRecognizerToFail:doubleTapRecognizer];

[myGestureView addGestureRecognizer:singleTapRecognizer];
```

首先尝试双击操作，此时在双击过程中不会激活单击处理方法。

2014-08-04 14:03:39.137 GesturePlayground[38536:2398989] Double Tap Received

23.5 定制 UIGestureRecognizer 子类

当应用需要识别一个非标准手势时，需要用到 UIGestureRecognizer 子类。首先需要确定的是该定制手势是属于离散型还是连续型。考虑到这一点之后，子类需要实现如下方法：

```
-(void)reset;
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;
-(void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;
-(void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event;
-(void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event;
```

在子类处理手势识别的 touchesBegan:/touchesMoved:/touchesEnded:方法中创建逻辑代码，之后在触碰动作的过程中更新子类使其处于正确的状态。记得为了避免 UI 延迟尽量设置状态为 UIGestureRecognizerStateFailed，并检查这些方法中对象的状态以避免执行不需要的逻辑代码。比如，如果手势需要双击，当有多于或少于双击触碰动作时应该立即使 touchesBegan:失效。如果状态已经是 UIGestureRecognizerStateFailed，则立即从 touchesMoved:和 touchesEnded:返回。

在重置方法中，将所有用于跟踪手势的实例变量更新为初始状态，这样识别对象就可以为下次触碰动作做好准备了。

注意

要了解更多有关创建 UIGestureRecognizer 子类的信息，可以查看 Apple 的 EventHandling Guide for iOS: Gesture Recognizers，网址为 https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/GestureRecognizer_basics/GestureRecognizer_basics.html#//apple_ref/doc/uid/TP40009541-CH2-SW2。所有需要的详细信息都可以在这里找到，并可以链接到相关类的参考资料。

23.6 小结

本章介绍了有关手势识别的内容，分别介绍了离散型和连续型两种手势识别，以及 6 种 iOS 标准的手势识别动作。本章首先介绍了手势识别的基本用法，之后根据多个手势识别动作组合的复杂情况又对其进行了分类介绍，最后介绍了定制手势识别的概念。

此时，读者对于创建并使用内置的标准手势识别应该很熟悉了，并且可以根据这些知识继续探索更加复杂的手势识别事件(例如，如何处理 3 个手指的滑动操作)。另外，读者还应该理解了手势识别的工作原理，并准备好使用定制子类了。

第24章 访问照片库

目前所有的 iOS 设备都至少带有一个摄像头，用于拍摄照片和视频。此外，所有 iOS 设备都可以从 iTunes 上将手机 Photos 应用中的照片同步到电脑上，并将这些照片按相簿进行整理。在 iOS 4 之前，开发者访问用户照片库的唯一方法是使用 UIImagePickerController。这个方法有一些缺点，即同时只能选择一张照片，开发者对于 UI 的显示效果也不能进行控制。随着 iOS 4 版本引入 AssetsLibrary 类函数，苹果公司提供了更多功能完善的访问用户照片、视频、相簿和事件的方法。虽然 AssetsLibrary 对 UIImagePickerController 有大幅改进，但它仍然无法浏览大量的资源，也无法以较好的性能得到任意尺寸的图片。此外，对于用户相簿的操作也比较困难(甚至无法访问相簿)，在相簿中维护图片也很难。

为了解决上述问题，苹果公司在 iOS 8 中引入了 Photos 框架。Photos 框架提供了一种健全、线程安全的方法访问和管理用户照片库。Photos 框架的方法可以从照片库获取任意尺寸的图片，还提供了一种回调机制能够在照片库发生变化时通知委托。Photos 框架还可以让用户无缝访问 iCloud 中的照片；无论照片是在本地设备还是在远程 iCloud 空间都可以进行访问和更新。

24.1 示例程序

本章的示例程序 PhotoLibrary 是 iOS Photos 应用的简单复现，只实现了一些核心功能。它具有两个标签页用于显示照片和相簿。Photos 标签将会以集合视图(collection view)的方式显示设备上的所有照片，照片根据时刻进行组合。点击缩略图会显示整张照片，并可以将图片从设备上删除。

Albums 标签将会按照用户创建的相簿以表视图的方式呈现，包括相簿名、包含照片数和封面图片。用户可以在这个视图中添加或删除相簿。点击相簿会显示其所包含的全部图片的缩略图。点击缩略图会显示相应的照片。

在运行示例程序之前，准备一台测试设备并同步一些照片，也可以拍摄几张照片。这样相簿和相册中就有一些照片了。如果使用 iCloud，也请一并打开 My Photo Stream。

注意

要在应用中使用 Photos 框架，在需要访问 Photos 框架的类中添加@import Photos，这样就可以在项目中自动添加 Photos 框架并导入所需的类了。

24.2 Photos 框架

Photos 框架包含一组用于浏览和管理设备中收藏、照片和视频的类。下面是一些重要的类：

- PHPhotoLibrary：该类用于表示设备和 iCloud 上所有的收藏和资源。可以使用一个共享实例以一种线程安全的方法对照片库的变化进行管理，比如添加新的资源或相簿，或者编辑和删除已有的资源或相簿。此外，共享实例还可以注册一个关于照片库发生变化的监听对象，以实现用户界面同步的功能。
- PHAssetCollection：该类用于表示一组照片和视频。可以在设备上本地创建，可以从 iPhoto 中同步照片，可以从相册或保存照片的相簿中获取图片，或者从根据一定的约束条件得到的智能相簿(比如全景照片)中同步图片。该类还提供以组的方式访问资源。资源集合可以集合列表的方式进行组合(PHCollectionList)。
- PHAsset：该类用于表示照片或视频的元数据。它提供了一些类方法，返回使用了相同约束条件的资源结果，提供带有资源信息的实例方法，比如日期、位置、类型和方向之类的信息。
- PHFetchResult：这是一个轻量级的对象，用于表示一组资源或资源集合。当按照约束条件请求资源或资源集合时，类方法将会返回一个抓取结果(fetch result)。抓取结果会按照需求载入资源，而不是一次将所有资源都载入内存，这样即使面对大量的资源也能够很好地加以处理。抓取结果同时还是线程安全的，这意味着如果底层数据发生变化，对象的个数不会变。可以为照片库的变化注册一个通知类，这些变化通过 PHFetchResultChangeDetails 实例发送，可以用于更新抓取结果和所有相应的用户界面。
- PHImageManager：图片管理器以异步的方法处理图片数据的抓取和保存，尤其适用于获取指定尺寸的图片，或者管理从 iCloud 获得的图片数据。此外，当需要在表视图或集合视图中显示大量资源时，照片库还提供 PHCachingImageManager 类来提升视图滑动的流畅性。

24.3 使用资源集合和资源

Photos.app 在设备上显示照片是按照“时刻”分组的——相同日期的照片放在一起。照片库中的时间是通过 PHAssetCollection 表示的。每个带有时刻的显示的图片都由一个 PHAsset 实例表示。Photos.app 中的用户相簿也是一个 PHAssetCollection 实例，其中的图片同样由 PHAsset 表示。在访问资源集合和资源时，应用需要获得用户对于访问照片库的权限。

24.3.1　权限

应用第一次访问照片库时，设备会询问用户是否允许进行访问，如图 24-1 所示。

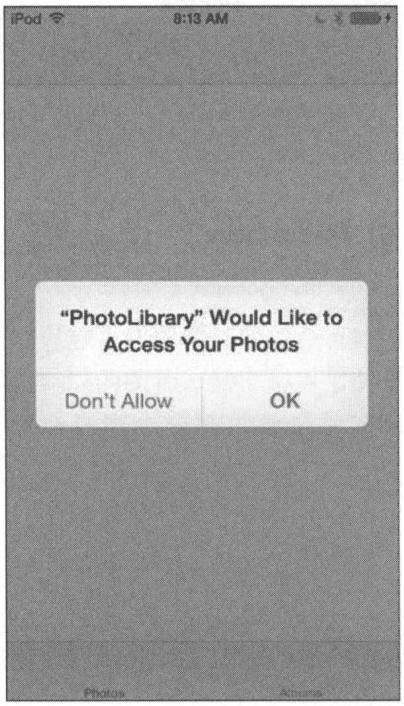


图 24-1　示例程序 PhotoLibrary 的访问权限提醒框

要请求权限，该应用需要对 PHPhotoLibrary 使用 requestAuthorization:类方法：

```
[PHPhotoLibrary requestAuthorization:^(PHAuthorizationStatus status) {
    if(status == PHAuthorizationStatusAuthorized) {
        [self loadAssetCollectionsForDisplay];
    } else {
        ...
    }
}];
```

这个方法会检查应用是否已经得到了权限。如果没有，则显示一个提醒框询问用户是否有权限访问照片库。用户做出选择之后，在 24 小时内只会显示一次该提醒框，即使应用卸载或重新安装也是如此。要在一天内多次测试该提醒框的响应情况，可以访问 Settings.app 中 General 下面的 Reset 菜单，并选择 Reset Location & Privacy(重置位置和隐私)。这样就可以抹去系统内存中有关位置和个人隐私的设置，此时设备上的所有应用都会再次向用户弹出提醒框。

如果用户已经得到授权或拒绝权限，提醒框就不再显示，同时要返回用户选择的结果。知道了授权状态之后，方法会调用处理状态的代码段，并传递当前的状态。注意，处理状态的代码可以在后台线程被调用，所以要确保在更新用户界面之前将应用切换回主队列。

如果用户拒绝访问照片库的权限，提醒视图会向用户解释如果后面需要权限时如何恢复

权限。要恢复权限，用户需要在 Settings.app 中找到相应的设置位置，如图 24-2 所示。

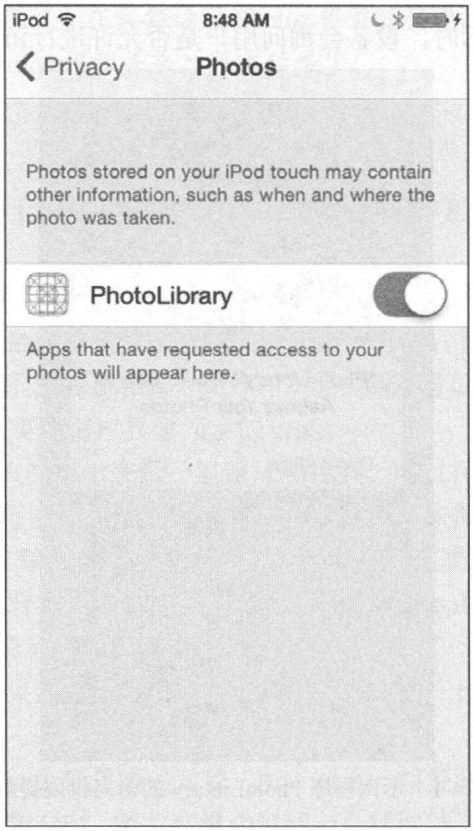


图 24-2 Settings.app:照片隐私设置

用户修改好设置后，iOS 会关闭示例程序，这样就可以重启应用而不是将程序从后台切换回来，只有这样才能应用修改好的新的隐私设置。

24.3.2 资源集合

获得权限之后，应用现在就可以访问照片库来显示时刻、相簿和图片。在示例程序的 Photos 页面，ICFPhotosCollectionViewController 会得到一个由 PHFetchResult 实例组成的时刻列表，这些实例都是在 loadAssetCollectionsForDisplay 方法中获得的。

```
PHFetchOptions *options = [[PHFetchOptions alloc] init];
options.sortDescriptors = @[[NSSortDescriptor sortDescriptorWithKey:@"startDate"
                                                          ascending:YES]];

self.collectionResult =
➥[PHAssetCollection fetchAssetCollectionsWithType:PHAssetCollectionTypeMoment
                                           subtype:PHAssetCollectionSubtypeAny
                                           options:options];
```

PHFetchResult 可以用于资源集合和具体资源。它的作用与 NSArray 类似，用到的函数也类似，如 objectAtIndex:和 indexOfObject:，不过这个方法很聪明，只有在确定需要时才会获

取相关的信息。之后，该方法从请求中迭代获取时刻信息，并把抓取结果实例保存到 collectionAssetResults 属性中，这样就可以方便地访问每个时刻对应的资源。

```
self.collectionAssetResults =
[[NSMutableArray alloc] initWithCapacity:self.collectionResult.count];

for(PHAssetCollection *collection in self.collectionResult) {
    PHFetchResult *result = [PHAsset fetchAssetsInAssetCollection:collection
                                                          options:nil];
    [self.collectionAssetResults insertObject:result
    ➥atIndex:[self.collectionResult indexOfObject:collection]];
}
```

在得到关于时刻和资源的信息后，视图控制器现在可以填充集合视图，为每个时刻显示一个分节头，在每个分节中显示相应的图片。欲了解更多关于集合视图的信息，可以参考第 21 章“集合视图”。要确定集合视图中分节的数量很简单，只需直接计算表示时刻的 PHFetchResult 实例个数即可。

```
-(NSInteger)numberOfSectionsInCollectionView:(UICollectionView *)collectionView {
    return self.collectionResult.count;
}
```

确定分节中元素的个数需要查看该分节中资源的抓取请求，得到的结果数量就是分节中元素的个数。

```
-(NSInteger)collectionView:(UICollectionView *)collectionView
     numberOfItemsInSection:(NSInteger)section {

    PHFetchResult *result =
    ➥(PHFetchResult *)[self.collectionAssetResults objectAtIndex:section];

    return result.count;
}
```

要在分节头中显示时刻的日期和位置，可以使用 collectionView:viewForSupplementary-ElementOfKind:atIndexPath:方法来获得 PHAssetCollection 实例，该实例根据分节的索引值来表示时刻信息：

```
PHAssetCollection *moment = [self.collectionResult objectAtIndex:indexPath.section];

[headerView.titleLabel setText:
➥[NSString stringWithFormat:@"%@ - %@",
➥[self.momentDateFormatter stringFromDate:moment.startDate],
➥[self.momentDateFormatter stringFromDate:moment.endDate]]];

[headerView.subTitleLabel setText:moment.localizedTitle];
```

之后，该方法会根据 PHAssetCollection 实例中得到的 startDate、endDate 和 localizedTitle 属性更新分节头，以显示正确的时刻日期范围和时刻对应的位置，如图 24-3 所示。

示例程序的 Albums 标签页显示所有用户在表视图中添加到照片库中的定制相簿，如图 24-4 所示。通过一个抓取结果获得相簿列表：

```
self.albumsFetchResult =
➥[PHAssetCollection fetchAssetCollectionsWithType:PHAssetCollectionTypeAlbum
                                    subtype:PHAssetCollectionSubtypeAny
                                    options:nil];
```

图 24-3 PhotoLibrary 示例程序中的时刻资源集合

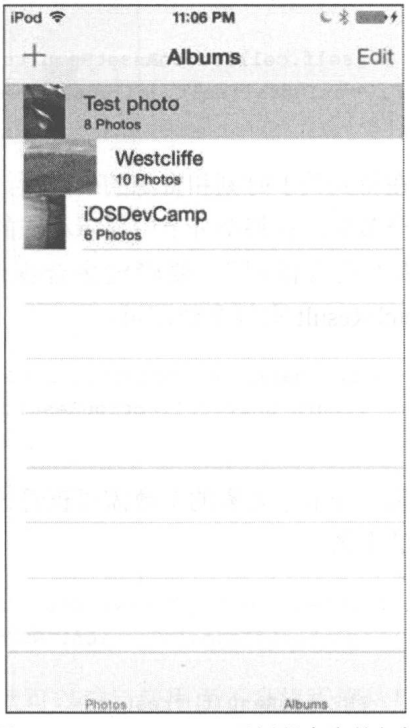


图 24-4 PhotoLibrary 示例程序中的相簿

对于表视图中的每个相簿，使用 localizedTitle 显示相簿的名称，使用 estimatedAssetCount 从资源集合确定相簿中资源的数量。

```
PHAssetCollection *album =
➥[self.albumsFetchResult objectAtIndex:indexPath.row];

[cell.textLabel setText:album.localizedTitle];

if(album.estimatedAssetCount != NSNotFound)
{
   NSString *albumPlural = album.estimatedAssetCount > 1 ? @"s" : @"";

   NSString *subTitle =
   ➥[NSString stringWithFormat:@"%lu Photo%@",
   ➥(unsigned long)album.estimatedAssetCount, albumPlural];

   [cell.detailTextLabel setText:subTitle];
```

```
} else
{
    [cell.detailTextLabel setText:@"-- empty --"];
}
```

当用户点击一个相簿时，示例程序会显示该相簿中的所有资源。因为示例项目使用 storyboard 来实现导航操作，所以从表视图单元格到 ICFAlbumCollectionViewController 创建了一个切换动作。该切换的名称为 showAlbum。在 ICFAlbumCollectionViewController 中，prepareForSegue:sender:方法用于创建目标视图控制器。

```
-(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if([segue.identifier isEqualToString:@"showAlbum"])
    {

        ICFAlbumCollectionViewController *controller =
        ➥(ICFAlbumCollectionViewController *)segue.destinationViewController;

        NSIndexPath *tappedPath = [self.tableView indexPathForSelectedRow];

        PHAssetCollection *tappedCollection =
        ➥[self.albumsFetchResult objectAtIndex:tappedPath.row];

        [controller setSelectedCollection:tappedCollection];
    }
}
```

prepareForSegue:sender:方法首先会检查切换对象的名称是否为 showAlbum，这是因为该方法在任何 ICFAlbumCollectionViewController 中的切换发生时都会被调用。之后，确定点击对象的索引路径，使用得到的行从抓取结果中获取相应的资源集合。在目标视图控制器中设置得到的资源集合，以显示该集合中的资源。

24.3.3　资源

在 ICFPhotosCollectionViewController 中，为了显示资源的图片，需要访问集合视图中每个单元格的资源。在 collectionView:cellForItemAtIndexPath:方法中，使用 indexPath 的 section 来确定在哪个抓取结果中查找资源，之后使用 indexPath 的 row 参数来获得具体的资源。

```
PHFetchResult *result = self.collectionAssetResults[indexPath.section];
PHAsset *asset = result[indexPath.row];
```

资源是描述图片的元数据。可以从照片库中请求一张大小合适且表示资源的图片。由于显示的图片尺寸可能不合适，因此需要调整本地或 iCloud 中下载的图片的尺寸。使用 PHImageManager 可以满足这个需求，以异步的方式提供显示所需尺寸的图片。要请求一张图片，需要提供目标尺寸和内容模式(参见第 20 章“使用图片和过滤器”以了解更多信息)，还要提供图片抓取的选项(PHImageRequestOptions)和一个结果处理程序块。该方法会返回一个 PHImageRequestID，用于取消该请求。

```
__weak ICFPhotosCollectionViewCell *weakCell = cell;
PHImageManager *imageManager = [PHImageManager defaultManager];

PHImageRequestID requestID =
[imageManager requestImageForAsset:asset
                        targetSize:CGSizeMake(50, 50)
                       contentMode:PHImageContentModeAspectFill
                           options:nil
                     resultHandler:^(UIImage *result, NSDictionary *info){
                        [weakCell.assetImageView setImage:result];
                        [weakCell setNeedsLayout];
                     }];

cell.requestID = requestID;
```

如果本地的图片版本暂时还不能在结果代码中显示，图片管理器将会立即返回一个低质量的近似图片到 resultHandler 代码块(注意，低质量的图片是通过在 info 字典中包含 PHImageResultIsDegradedKey 实现的)，并且当实际图片可以显示时再次调用结果处理程序，这次显示的就是高质量的图片。结果处理程序会在调用它的同一个队列中执行程序，所以如果请求是主队列发起的，即使不切换回主队列，更新用户界面也是安全的。如果请求来自后台队列，将 PHImageRequestOptions 的 synchronous 属性设置为 YES，阻塞后台队列直到请求返回。

如果在图片请求填满屏幕之前手势动作就滑出了视图，可以在 prepareForReuse 方法中为单元格执行取消请求。

```
-(void)prepareForReuse {
    self.assetImageView.image = nil;

    PHImageManager *imageManager = [PHImageManager defaultManager];
    [imageManager cancelImageRequest:self.requestID];
}
```

PHAsset 实例是线程安全的，可以按照需要传递。如果用户点击一个单元格，将会激活一个切换并呈现该资源的全屏视图。

```
-(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if([segue.identifier isEqualToString:@"showImage"]) {

        ICFAssetViewController *controller =
        ➥(ICFAssetViewController *)segue.destinationViewController;

        NSIndexPath *indexPath = [self.collectionView indexPathsForSelectedItems][0];
        PHFetchResult *result = self.collectionAssetResults[indexPath.section];
        controller.asset = result[indexPath.row];
    }
}
```

之后，详细视图(ICFAssetViewController)可以从 PHImageManager 中请求一张全屏图片进

行显示。

```
PHImageManager *imageManager = [PHImageManager defaultManager];
[imageManager requestImageForAsset:self.asset
                        targetSize:self.assetImageView.bounds.size
                       contentMode:PHImageContentModeAspectFit
                           options:nil
                     resultHandler:^(UIImage *result, NSDictionary *info){
                         [self.assetImageView setImage:result];
                         [self.assetImageView setNeedsLayout];
                     }];
```

24.4　照片库中的编辑操作

照片库支持以稳健且线程安全的方式对资源、资源集合和集合列表进行编辑，主要包括从照片库中添加、修改或删除对象。对于资源，不仅包括添加新资源和删除已存在的资源这些方法，还可以为这些动作应用特殊的编辑和过滤效果。示例程序演示了添加和删除资源集合和资源的功能，通用方法对所有照片库都是可用的。要编辑照片库，应用首先需要得到一个照片库对象来执行编辑请求，之后提供一个监听器在编辑完成后进行处理。

24.4.1　编辑资源集合

在相簿页面的导航上有一个添加(+)按钮。点击这个按钮将会弹出一个框，向用户询问新相簿的名称，如图 24-5 所示。

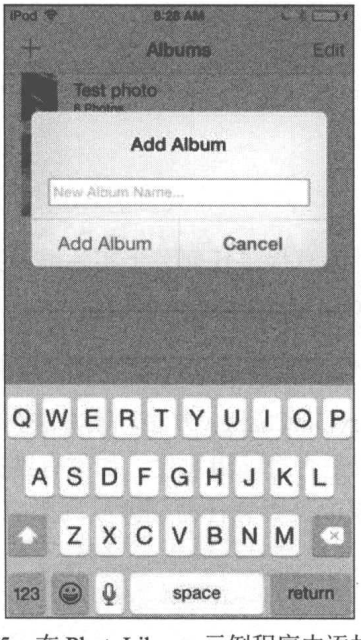


图 24-5　在 PhotoLibrary 示例程序中添加相簿

如果用户选择 Add Album，动作处理程序会尝试使用在提醒框中输入的名称创建一个相簿。

```
UITextField *albumNameTextField = addAlbumAlertController.textFields.firstObject;
NSString *newAlbumName = albumNameTextField.text;
[[PHPhotoLibrary sharedPhotoLibrary] performChanges:^{
    [PHAssetCollectionChangeRequest
    ➥creationRequestForAssetCollectionWithTitle:newAlbumName];
} completionHandler:^(BOOL success, NSError *error) {
    if(!success) {
        NSLog(@"Error encountered adding album: %@",error.localizedDescription);
    }
}];
```

要添加一个新的相簿，需要一个编辑请求。编辑请求包括创建一个新相簿、修改一个已存在的相簿和删除一个相簿。编辑请求被传递到照片库的 performChanges 中，它会根据 completionHandler 进行响应，completionHandler 用于表示照片库中的编辑是否成功完成。同样，Edit 按钮可以让用户从表视图显示的相簿列表中删除一个相簿。在 tableView:commitEditingStyle:forRowAtIndexPath:方法中，所选中行的相簿将会被删除。

```
if(editingStyle == UITableViewCellEditingStyleDelete) {

    PHAssetCollection *albumToBeDeleted =
    ➥[self.albumsFetchResult objectAtIndex:indexPath.row];

    [[PHPhotoLibrary sharedPhotoLibrary] performChanges:^{
        [PHAssetCollectionChangeRequest deleteAssetCollections:@[albumToBeDeleted]];
    } completionHandler:^(BOOL success, NSError *error) {
        if(!success) {
            NSLog(@"Error encountered adding album: %@",error.localizedDescription);
        }
    }];
}
```

注意，completion handler 并没有提供任何关于实际编辑完成的信息，要得到这些信息，视图控制器需要注册为 PHPhotoLibraryChangeObserver 协议的观察者。

```
[[PHPhotoLibrary sharedPhotoLibrary] registerChangeObserver:self];
```

之后，需要实现 photoLibraryDidChange 方法。该方法会接收一个 PHChange 实例，它可以告诉程序对于给定的抓取结果是否有编辑发生，如果有则提供所有这些编辑动作。对于相簿视图，该方法会检查抓取结果中是否有任何相簿受到影响。

```
PHFetchResultChangeDetails *changesToFetchResult =
➥[changeInstance changeDetailsForFetchResult:self.albumsFetchResult];
```

如果抓取结果受到影响，则需要对其进行更新，以反映照片库新的状态。PHFetchResultChangeDetails 有一个简单的方法来完成这一任务：提供抓取结果之前和之后的视图。

```
self.albumsFetchResult = [changesToFetchResult fetchResultAfterChanges];
```

现在抓取结果已是最新结果，可以使用这些编辑的细节对表视图进行更新，这样就和抓取结果相匹配了。现在，更新表视图或集合视图所需的全部细节都已具备。

```
if([changesToFetchResult hasIncrementalChanges])
{
    [self.tableView beginUpdates];

    [[changesToFetchResult removedIndexes]
     ➥enumerateIndexesUsingBlock:^(NSUInteger idx, BOOL *stop) {

        NSIndexPath *indexPathToRemove =
        ➥[NSIndexPath indexPathForRow:idx inSection:0];

        [self.tableView deleteRowsAtIndexPaths:@[indexPathToRemove]
                              withRowAnimation:UITableViewRowAnimationAutomatic];
    }];

    [[changesToFetchResult insertedIndexes]
     ➥enumerateIndexesUsingBlock:^(NSUInteger idx, BOOL *stop) {

        NSIndexPath *indexPathToInsert =
        ➥[NSIndexPath indexPathForRow:idx inSection:0];

        [self.tableView insertRowsAtIndexPaths:@[indexPathToInsert]
                              withRowAnimation:UITableViewRowAnimationAutomatic];
    }];

    [self.tableView endUpdates];
}
```

该方法会检查编辑详情是否还包括其他增加的编辑动作。如果包含，就迭代已经移除的索引号，为移除的每个索引号创建一条索引路径，并从表视图中以动画效果将其删除。接下来会迭代已经插入的索引号，为每个增加的行创建一条索引路径，并以动画效果将它们插入到表视图中。在编辑详情中要提供索引号以支持这个模式，注意，必须先移除行，否则插入操作的索引号和编辑操作的索引号都将不正确。编辑完成后，表视图会以动画效果的方式应用所有的编辑并给出照片库最新的状态。

24.4.2　编辑资源

在示例程序的 Albums 标签页，选择一个相簿查看其中的资源。导航栏上面的添加按钮可以将一张图片添加到照片库，并将这张新的图片添加到选中的相簿。点击添加按钮会打开摄像头类 UIImagePickerController，所以示例程序必须运行在具有该硬件的设备上。注意来自任何外部源的图片都可以被添加到照片库；我们使用图片选取器只是因为它是演示这个功能的一个比较方便的办法而已。

当用户从摄像头得到一张图片时，会将这张图片发送到选取器的委托方法。

```
UIImage *selectedImage = [info objectForKey:UIImagePickerControllerOriginalImage];
```

要将图片添加到照片库，需要在 performChanges 代码段中向照片库传递一个编辑请求。

```
[[PHPhotoLibrary sharedPhotoLibrary] performChanges:^{

    PHAssetChangeRequest *addImageRequest =
    ➥[PHAssetChangeRequest creationRequestForAssetFromImage:selectedImage];

    ...
} completionHandler:^(BOOL success, NSError *error) {
    if(!success) {
        NSLog(@"Error creating new asset: %@", error.localizedDescription);
    }
}];
```

直到调用 photoLibraryDidChange:时，用于表示最新添加的图片的 PHAsset 对象才生效，那么新的图片是如何添加到相簿中的呢？答案就是 placeholder(占位符)对象。照片库会为编辑请求中新创建的对象提供一个占位符对象，这样在相同的编辑请求中就可以使用那些新对象了。

```
PHObjectPlaceholder *addedImagePlaceholder =
➥[addImageRequest placeholderForCreatedAsset];
```

占位符对象可以用于那些即将在相簿中创建的资源：

```
PHAssetCollectionChangeRequest *addImageToAlbum =
➥[PHAssetCollectionChangeRequest
➥changeRequestForAssetCollection:self.selectedCollection];

[addImageToAlbum addAssets:@[addedImagePlaceholder]];
```

完成照片库的编辑后，会在任意队列上调用 photoLibraryDidChange:委托方法。切换回主队列后，该方法会检查相簿中对于资源的抓取结果是否有变化。

```
PHFetchResultChangeDetails *changesToFetchResult =
➥[changeInstance changeDetailsForFetchResult:self.assetResult];
```

如有变化，就更新抓取结果和表视图：

```
if(changesToFetchResult)
{
    self.assetResult = [changesToFetchResult fetchResultAfterChanges];

    if([changesToFetchResult hasIncrementalChanges]) {
        NSMutableArray *indexPathsToInsert = [[NSMutableArray alloc] init];

        [[changesToFetchResult insertedIndexes]
        ➥enumerateIndexesUsingBlock:^(NSUInteger idx, BOOL *stop) {

            NSIndexPath *indexPathToInsert =
```

```
            ➥[NSIndexPath indexPathForRow:idx inSection:0];

            [indexPathsToInsert addObject:indexPathToInsert];

        }];

        [self.collectionView insertItemsAtIndexPaths:indexPathsToInsert];
    }
}
```

在集合视图中以动画的方式添加新的单元格，用于显示新的图片。

当用户在相簿视图或时刻视图中点击图片时，图片会以 ICFAssetViewController 格式全屏显示。该视图带有一个删除按钮，允许用户从照片库中删除该资源。

```
[[PHPhotoLibrary sharedPhotoLibrary] performChanges:^{
    [PHAssetChangeRequest deleteAssets:@[self.asset]];
} completionHandler:^(BOOL success, NSError *error) {
    if(success) {
        [self.navigationController popViewControllerAnimated:YES];
    } else {
        NSLog(@"Error deleting asset: %@",error.localizedDescription);
    }
}];
```

当用户点击 Delete 按钮并执行删除资源的编辑请求时，照片库会弹出一个确认提醒框，如图 24-6 所示。如果用户选择 Don't Allow，则不会出现任何动作。如果用户选择 Delete，则会从照片库中删除该资源，并通知所有已注册的监听器。

图 24-6　在 PhotoLibrary 示例程序中删除一张照片

24.5 处理照片流

Photo Stream(照片流)是指一种照片同步的方法，是苹果公司 iCloud 服务的一部分。当一个 iCloud 用户向支持照片流的设备添加一张照片时，照片会立即同步到用户所有其他支持照片流的设备上。比如，如果用户拥有 iPhone、iPad 和一台 Mac，在 iPhone 上拍摄一张照片，不需要进行任何请求就会立即显示在 iPad(在 Photos 应用中)和 Mac(在 iPhone 或 Aperture 程序里)上。

要使用 Photo Stream，用户需要拥有一个 iCloud 账户。可以在 iOS 设备上免费创建 iCloud 账户。在 Settings 应用中找到 iCloud，然后创建一个新的账户或输入 iCloud 账户信息即可。在设备上登入 iCloud 账户后，就可以使用 Photo Stream 了，如图 24-7 所示。

启用 Photo Stream 之后，会在设备间自动同步时刻信息。示例程序不需要额外编码就可以显示和处理从其他设备得到的时刻。当从 PHImageManager 请求显示图片时，可以根据需要明确地指定允许或禁止网络访问。

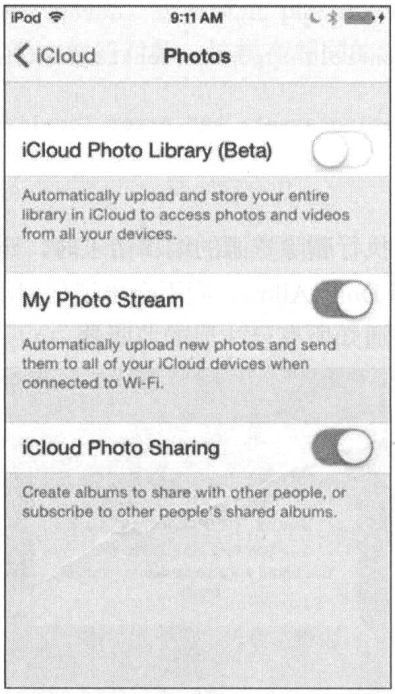


图 24-7 Settings 应用：iCloud Photos

24.6 小结

本章介绍了如何使用 iOS 8 版本中的 Photos 框架来访问照片库，详细介绍了用于访问照片库的类，并描述了如何处理有关照片库访问权限的问题。本章还介绍了如何使用 Photos 框架的类实现与 Photos.app 同样的组织结构来显示图片。本章接下来讨论了以异步的方法从网络端获取尺寸合适的图片，然后探讨了有关编辑照片库的功能，包括添加和删除资源集合，以及添加和删除具体的资源。最后，本章介绍了如何通过 iCloud 的 Photo Stream 在照片库中同步远程照片。

第25章

Passbook 和 PassKit

随着 iOS 6 的推出，苹果公司引入了一个新的标准应用，叫作 Passbook。在这个应用中，用户可以很容易地访问他们之前购买的活动门票、旅行车票、优惠券和商铺发放的各种卡(类似礼物卡、储值卡或返券)。Passbook 目前只支持在 iPhone 和 iPod touch 设备上使用，不支持 iPad。

在锁屏界面中有一种特殊的方法可以访问 Passbook。如果用户在一个由位置或通行证规定的一组位置集所在的地理围栏之内，以及在一个通行证所指定的相关日期之内，在锁屏界面上就会显示这个通行证，这样用户就可以通过滑动锁定条直接打开 Passbook 应用。

Passbook 应用显示通行证的方式类似于堆栈，用户可以看到每类通行证最上面的内容。其中包括一个图标和带有颜色的背景，还包含一些定制信息。当用户点击某类通行证的最上部分时，就会展开该分类并显示其中包含的所有通行证，这其中可以包含许多定制区域、背景图片和条形码。用户可以删除不需要的通行证。Passbook 应用本身支持通过 Web 服务和推送通知的方法更新已包含在 Passbook 中的通行证信息。

使用 Passbook 需要以规定的格式创建一个“通行证”，并且将该通行证发送给用户。通行证的创建不是在用户设备上实现的，通常是在一台服务器上，因为它需要对图标和用于显示的图片进行签名打包，包括一个带有通行证信息的文件、一个签名文件和一个清单。

在将通行证发送给用户时有一些选项可以设置。Mail 应用和 Safari 应用可以识别通行证并将其导入到 Passbook 中，或者一个定制应用可以使用 PassKit 添加一个新的通行证或对一个已存在的通行证进行更新。要测试有关通行证的功能，当有一个通行证被拖到 Passbook 中时 iOS 模拟器会立即显示它。

PassKit 是 iOS SDK 的一部分，它用于实现在定制应用上将通行证导入到 Passbook 的功能，并检查该通行证是新的还是需要对其进行更新，同时显示关于已存在的通行证的一些信息。

本章会介绍关于不同类型通行证的设计模式，以及如何创建并测试这些通行证，将演示如何使用 PassKit 实现从应用中获取通行证的功能，最后还将介绍 Passbook 如何处理来自 Web 服务的更新。

25.1 示例程序

本章的示例程序名为 Pass Test。对于每个类型的通行证它都包含了预设好的示例通行证，如图 25-1 所示。用户可以使用 PassKit 向应用中添加新的通行证到 Passbook，也可以使用新的信息模拟更新一个已存在的通行证，还可以直接在 Passbook 中查看通行证，同时还可以从 Passbook 中删除来自于应用的通行证。有关示例程序更详细的介绍将在 25.4.6 节“具体应用中的通行证交互”中讨论。

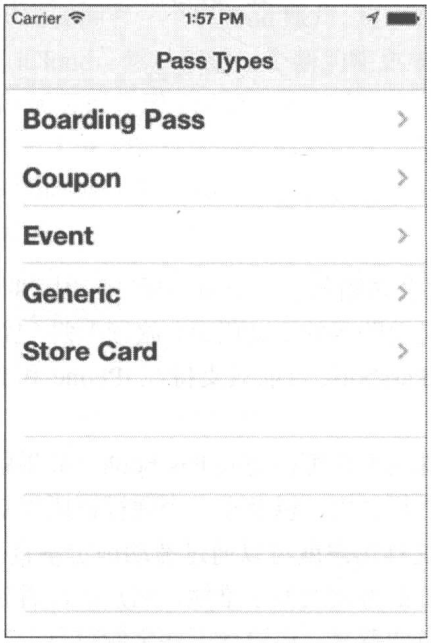


图 25-1 示例程序 Pass Test

25.2 设计通行证

在将通行证发给用户之前，需要对其进行设计和配置。通行证的提供者需要确定通行证的类型及外观。这其中包括需要在通行证上指定显示哪些信息，以及在什么位置显示这些信息，是否使用条形码，以及在锁屏界面显示通行证时应该使用什么内容。一个单独的通行证实际上是一个签好名的文件组，包括一些图片、一个带有通行证及显示信息的名为 pass.json 的 JSON 文件，以及一个签名和清单。

25.2.1 通行证的类型

苹果给出了一些标准的通行证类型，对于每种类型还可以进行定制：

- Boarding Pass(登机牌)：登机牌类型一般包括有关旅行的票据，例如飞机票、火车票、汽车票、巴士票或任何带有出发地和目的地交通工具所需的票据。
- Coupon(优惠券)：优惠券一般是有关商店打折的通行证。优惠券的设计非常灵活，可以用于处理多种情况，例如打折百分之多少、指定商品打折、指定一组商品打折、指定特殊区域、指定一片特殊区域等。

- Event(入场券)：入场券即参加任何活动或事件的门票，例如剧院的演出票、电影票、体育赛事门票或者参观博物馆的门票等——任何需要门票才能访问的场所和活动。
- Store Card(购物卡)：购物卡通行证同礼品卡类似，可以在该通行证上预充值，并向商家出示该凭证，可以直接从中扣除相应的费用。当购物卡余额不足时商家可以允许用户向其中充值或者购买新的购物卡。购物卡还可以作为奖励或会员卡使用，每次购物可以积分直到达到相应的奖励标准。
- Generic(通用卡)：通用卡通行证可以用于除了预生成通行证类型之外的情况，或者有些预生成通行证无法满足需求的情况。通用卡包含一个缩略图，所以它可以用于带有指定组织 ID 的卡，例如健身会员卡。

在通行证的设计过程中，每种类型的卡都具有特定的布局。根据数据区域的不同分成一些部分：内容头、主区域、次区域、辅助区域和背景。在一些实例下通行证还可以使用定制图片。下面的一节会介绍每种通行证的布局规范。

25.2.2　通行证布局——登机牌

登机牌通行证的布局示例如图 25-2 所示。

对于登机牌类通行证，出发地和目的地通常都在主区域内显示(一个登机牌可以有两个主区域)。次区域和辅助区域都在主区域下面。页脚图片是可选的。

图 25-2　登机牌布局

25.2.3　通行证布局——优惠券

优惠券通行证的布局示例如图 25-3 所示。

图 25-3　优惠券布局

优惠券布局只有一个主区域，可以在主区域的后面选择显示一张条状图片。优惠券一共可以有 4 个次区域和辅助区域。

25.2.4　通行证布局——入场券

入场券通行证的布局示例如图 25-4 所示。

图 25-4　入场券布局

入场券布局只有一个主区域，并且可以在所有区域和条码区域的后面选择显示一张背景图片。如果提供背景图片，会自动修剪和模糊。入场券还会在主区域和次区域的右侧显示一个缩略图。

25.2.5　通行证布局——通用卡

通用卡通行证的布局示例如图 25-5 所示。

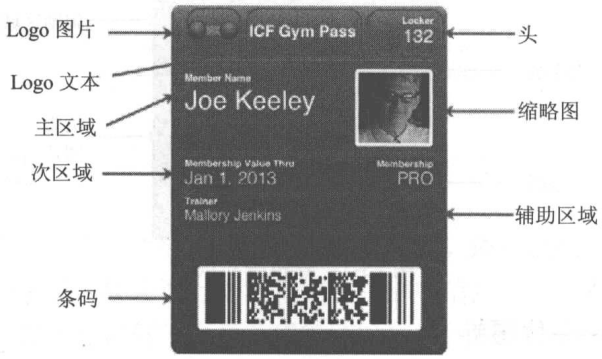


图 25-5　通用卡布局

通用卡布局只有一个主区域，可在主区域的右侧选择显示一个缩略图。次区域和辅助区域在主区域的下面显示。

25.2.6　通行证布局——购物卡

购物卡通行证的布局示例如图 25-6 所示。

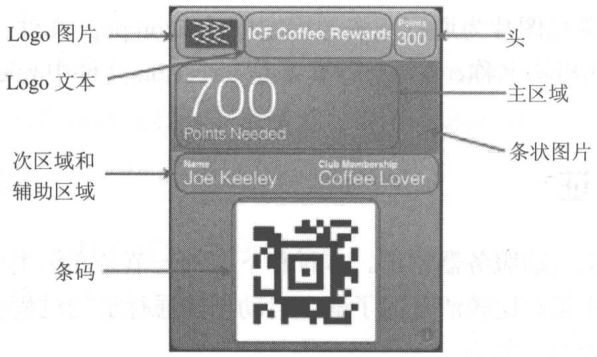


图 25-6　购物卡布局

购物卡布局只有一个主区域，可以在主区域的后面选择显示一张条状图片。购物卡一共可以有 4 个次区域和辅助区域，在主区域的下面显示。

25.2.7　通行证的显示

在 Passbook 之外，在很多情况下需要为用户显示通行证，所以理解通行证的哪些部分可以定制处理来进行呈现就变得非常重要。当一个通行证通过 email 发送给用户时，显示的效果如图 25-7 所示。

从通行证绑定资源中得到的显示图片为 icon.png 文件。最上面的蓝色文本是根据 pass.json 文件保存的通行证类型自动获得的，文本下面的内容是 pass.json 文件中保存的机构名。

当设备处于通行证指定的机构附近时，或者时间到了通行证指定的日期附近，通行证就会在设备锁屏界面上显示，同推送通知类似，如图 25-8 所示。要了解更多关于这方面的内容，可以参阅 25.3.2 节“通行证相关信息”。

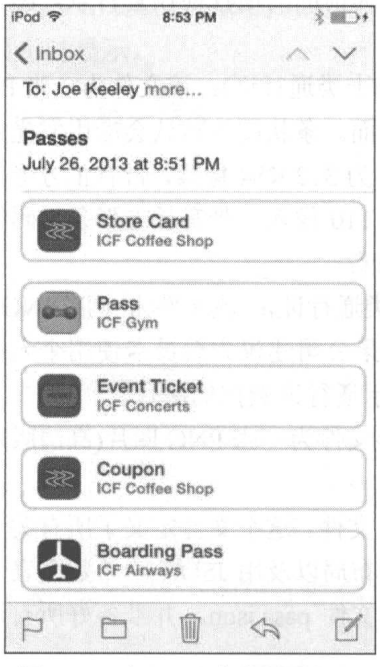


图 25-7　在 email 中发送通行证

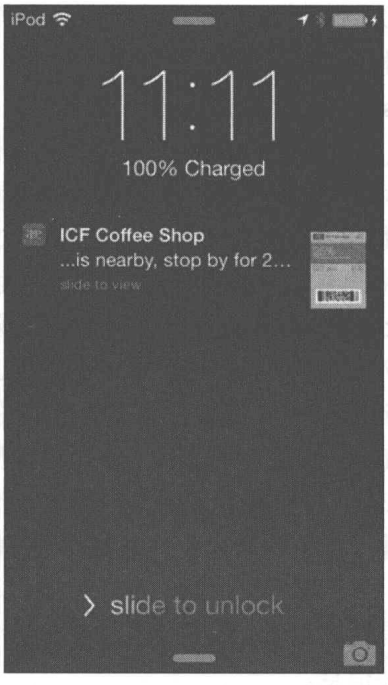


图 25-8　锁屏界面中显示的通行证

通知左侧显示的图标图片为通行证资源绑定中的 icon.png 文件。最上面一行文本是在 pass.json 文件中指定的机构名称，下面一行文本是 pass.json 文件中 relevantText 的值，并带有相关位置信息。

25.3 创建通行证

通行证通常可以由自动服务器创建。本章和下面的一节会介绍手动创建通行证的方法，这样会让读者对所有步骤有比较清楚的了解，自动创建通行证的过程要根据选择服务器的环境进行设置，留给读者自己练习。

创建一个通行证需要很多步骤。苹果公司建议开发者创建一个文件夹保存通行证所需的用到的所有文件，例如文件夹名为 Boarding Pass.raw。

将用到的所有图片都放在这个文件夹中。通行证支持高清和非高清两个版本，分别使用标准的“@2x”和“@3x”进行命名。需要提供的图片如下所示：

- icon.png(必需)：当检查到通行证并需要使用它时，在所有支持 PassKit 的应用(例如 Mail 应用或 Safari 应用)中都会显示一个 29×29 像素的 PNG 图片(高清版本同样如此)。锁屏界面用到的图片也是这张图片。同应用图标的显示一样，该图标会自动应用圆角边框和发光效果。
- logo.png(必需)：标志 PNG 图片，最大尺寸为 160×50 像素(高清版本同样如此)，位于 Passbook 通行证头的位置。苹果公司建议只使用一张图片作为标志，并建议避免在标志中出现任何格式文本。当标志显示时，会带有一个标准的头(显示定制文本)，在 Passbook 中看起来非常一致。
- background.png(可选，只用于入场券类通行证)：该文件为一张 PNG 图片，最大尺寸为 180×220 像素(高清版本同样如此)，用于为前面的通行证设置背景。图片会自动修剪和模糊化。
- strip.png(可选，只用于优惠券、入场券和购物卡类通行证)：该文件为一张 PNG 图片(高清版本同样如此)，位于通行证主区域的后面。条状图片默认会应用发光效果，也可以关闭这个效果。入场券类通行证最大尺寸为 312×84 像素，带有正方形条形码的优惠券和购物卡类通行证的最大尺寸为 312×110 像素，带有长方形条形码的优惠券和购物卡类通行证的最大尺寸为 312×123 像素。
- thumbnail.png(可选，只用于入场券和通用卡类通行证)：该文件为一张 PNG 图片(高清版本同样如此)，显示在通行证的前面。苹果公司建议为会员卡使用个人图片，不过也可以使用表示会员等级的图片或者用表示通行证数据的图片。
- footer.png(可选，只用于登机牌类通行证)：该文件为一张 PNG 图片(高清版本同样如此)，位于通行证的条形码的上面。

创建一个单独通行证的主要工作是创建 pass.json 文件。这个文件定义了所有关于通行证的信息，包括通行证的唯一标识、类型、相关信息和布局以及用 JSON 哈希数列表示的可视化定制信息。在通行证文件夹中创建一个简单的文本文件 pass.json，并准备好所有通行证所需的相关信息。

注意

参阅“JSON 的使用和解析”以了解更多有关创建 JSON 哈希数列的信息。注意，刚开始时你可以解压缩任何已存在的通行证，并查看它的 pass.json 文件。

25.3.1　基础通行证标识

通行证具有很多可以识别的字段，如下所示：

```
"description" : "Event Ticket",
"formatVersion" : 1,
"passTypeIdentifier" : "pass.explore-systems.icfpasstest.event",
"serialNumber" : "12345",
"teamIdentifier" : "59Q54EHA9F",
"organizationName" : "ICF Concerts",
```

下面这些字段是一个通行证必须包含的：

- description(必需)：一个本地 iOS 可访问的字符串，用于描述通行证。
- formatVersion(必需)：Passbook 格式版本，当前必须为 1。
- passTypeIdentifier(必需)：由苹果公司提供的通行证类型标识符。查看 25.4.1 节以了解更多有关如何获取通行证类型 ID 的信息。
- serialNumber(必需)：通行证类型 ID 所需的唯一标识符。通行证类型 ID 和序列号组合在一起作为一个通行证的唯一标识。
- teamIdentifier(必需)：苹果公司为组织机构提供的一个团队标识符。可以在 Developer Member Center 中找到，在 Organization Profile 下面的 Company/Organization ID 部分。
- organizationName(必需)：通行证提供的表示组织机构名称的本地化字符串。当通行证在 iOS 6 及以上版本的系统的 Mail 应用中呈现时显示，以及当通行证在锁屏界面上显示时显示。

25.3.2　通行证相关信息

一个通行证可以有选择地提供一些相关信息，包括位置和日期，如下所示：

```
"locations" : [
  {
    "latitude" : 39.749484,
    "longitude" : -104.917513,
    "relevantText" : "...is nearby, stop by for 20% off a coffee!"
  }
],
"relevantDate" : "2014h-10-20T19:30:00-08:00"
```

相关信息字段如下所示：

- locations(可选)：一组保存相关位置信息的数组。位置信息可以包括 latitude、longitude、altitude 和 relevantText 值。当设备接近相关位置时会在锁屏界面上显示 relevantText。如何检查近似位置的半径是根据通行证类型设置的。

- relevantDate(可选)：一个用字符串表示的 ISO 8601 日期表达式。

根据通行证的不同，应用的规则也不同：

- 登机牌类通行证：对于位置的检测半径很大。如果位置或日期匹配，即会激活关联。
- 优惠券：对于位置的检测半径比较小，不使用日期关联检测。
- 入场券：对于位置的检测半径很大。如果位置或日期匹配，即会激活关联。
- 通用卡：对于位置的检测半径比较小。如果位置或日期匹配，即会激活关联，或者仅有位置匹配而无时间的情况也会激活关联。
- 购物卡：对于位置的检测半径比较小，不使用日期关联检测。

25.3.3 条形码识别

要在通行证上显示一个条形码，需要提供消息、条形码格式和消息编码参数。对于消息信息，还可以选择提供可读性比较强的文本参数来显示。

```
"barcode" : {
  "message" : "123456789",
  "format" : "PKBarcodeFormatQR",
  "messageEncoding" : "iso-8859-1"
  "altText" : "123456789",
},
```

显示条形码会用到如下字段：

- format(必需)：表示 PassKit 常量的字符串，用来指定显示条形码的格式。Passbook 当前支持 QR 码(PKBarcodeFormatQR)、PDF417 码(PKBarcodeFormatPDF417)和 Aztec 码(PKBarcodeFormatAztec)。PDF417 是长方形条码格式，QR 和 Aztec 是正方形条码。
- message(必需)：条码所表示的字符串消息。
- messageEncoding(必需)：表示 IANA 字符集的字符串，用于将消息由字符串转换为数据，通常为 iso-8859-1。
- altText(可选)：可读文本，用于表示编码好的消息，显示在条码附近。

25.3.4 通行证视觉外观信息

通行证可以定制背景色、区域值和区域标签，同标志中显示的文本一样。

```
"logoText" : "ICF Concerts",
"foregroundColor" : "rgb(79, 16, 1)",
"backgroundColor" : "rgb(199, 80, 18)",
"labelColor" : "rgb(0,0,0)",
```

如下字段可以用于定制通行证的外观显示：

- logoText(可选)：一个本地字符串，在标志右边的头位置显示。
- foregroundColor(可选)：表示 CSS 类型 RGB 颜色的字符串，用于通行证的区域值。
- backgroundColor(可选)：表示 CSS 类型 RGB 颜色的字符串，用于通行证的背景。入场券通行证不使用该字段，而是使用背景图片。

- labelColor(可选)：表示 CSS 类型 RGB 颜色的字符串，用于通行证的区域标签。苹果公司建议使用白色，做到一定程度的统一。
- suppressStripShine(可选)：用于表示是否禁用条状图片的发光效果(只对优惠券、入场券或购物卡类通行证有效)的布尔值(true 或 false)。默认值为 false，即使用发光效果。

25.3.5　通行证区域

用一个带有键值的元素表示通行证区域的类型或样式，可以选择 boardingPass、coupon、eventTicket、generic 和 storeCard。元素中还有额外的元素用于组成通行证上的具体区域。

```
"boardingPass" : {
  "transitType" : "PKTransitTypeAir",
  "headerFields" : [
     ...
  ],
  "primaryFields" : [
     ...
  ],
  "secondaryFields" : [
     ...
  ],
  "auxiliaryFields" : [
     ...
  ],
  "backFields" : [
     ...
  ]
}
```

用于表达通行证指定信息的字段如下所示：

- transitType(登机牌类通行证必须包含该字段，其他通行证不能使用)：表示登机牌类通行证的交通工具类型，使用一个表示交通工具的字符串常量来表示，包括 PKTransitTypeAir、PKTransitTypeTrain、PKTransitTypeBus、PKTransitTypeBoat 和 PKTransitTypeGeneric。通行证还会为交通工具显示一个图标。
- headerFields(可选)：头区域显示在每个通行证的最上面。当通行证叠加显示时这个部分的内容是可见的，所以要特别认真对待此处应该显示的内容。
- primaryFields(可选)：主区域位于头区域下面，一般使用最大、最突出的字体显示。
- secondaryFields(可选)：次区域显示在主区域下面，通常使用常规字体大小。
- auxiliaryFields(可选)：辅助区域显示在次区域下面，通常使用比较小、不突出的字体。
- backFields(可选)：背景区域显示在通行证的后面。

在每个区域包含的元素都是一组数组。描述一个区域最起码需要一对键值和一个标签。

```
{
  "key" : "seat",
  "label" : "Seat",
  "value" : "23B",
  "textAlignment" : "PKTextAlignmentRight"
}
```

对于每个区域，可以提供如下信息：

- key(必需)：这里的键必须是一个字符串，用于在通行证中唯一表示一个区域，例如“seat”。
- value(必需)：区域的值，例如“23B”。该值可以是一个本地字符串、数字或 ISO 8601 格式的日期。
- label(可选)：区域的字符串标签。
- textAlignment(可选)：表示文本对齐方式的字符串常量，包括 PKTextAlignmentLeft、PKTextAlignmentCenter、PKTextAlignmentRight、PKTextAlignmentJustified 和 PKTextAlignmentNatural。
- changeMessage(可选)：描述区域修改的消息，例如“Changed to %@”，此处的%@可以用新的值替换。在本章后面会详细讨论这部分的内容，可以参考 25.4.6 节的“模拟更新通行证”部分。

对于日期和时间区域，可以为它们指定日期类型和时间类型。要显示日期或时间的话，就必须指定日期和时间的类型。

```
{
  "key" : "departuretime",
  "label" : "Depart",
  "value" : "2012-10-7T13:42:00-07:00",
  "dateStyle" : "PKDateStyleShort",
  "timeStyle" : "PKDateStyleShort",
  "isRelative" : false
},
```

需要指定日期和时间的字段如下所示：

- dateStyle(可选)：可选项包括 PKDateStyleNone(相当于 NSDateFormatterNoStyle)、PKDateStyleShort(相当于 NSDateFormatterShortStyle)、PKDateStyleMedium(相当于 NSDateFormatterMediumStyle)、PKDateStyleLong(相当于 NSDateFormatterLongStyle)和 PKDateStyleFull(相当于 NSDateFormatterFullStyle)。
- timeStyle(可选)：可选项包括 PKDateStyleNone(相当于 NSDateFormatterNoStyle)、PKDateStyleShort(相当于 NSDateFormatterShortStyle)、PKDateStyleMedium(相当于 NSDateFormatterMediumStyle)、PKDateStyleLong(相当于 NSDateFormatterLongStyle)和 PKDateStyleFull(相当于 NSDateFormatterFullStyle)。

- isRelative(可选)：true 显示相对日期，false 显示绝对日期。

对于数字，可以为其指定货币代码或数值类型。

```
{
  "key" : "maxValue",
  "label" : "Max Value",
  "value" : 1.50,
  "currencyCode" : "USD"
}
```

为数字或货币类型的值可以指定如下字段：

- currencyCode(可选)：ISO 4217 货币代码，显示由代码表示的货币数值。
- numberStyle(可选)：可选项包括 PKNumberStyleDecimal、PKNumberStylePercent、PKNumberStyleScientific 和 PKNumberStyleSpellOut。

提示

当在开发中创建 pass.json 文件时，要对 JSON 进行测试，验证其有效性。这样可以避免很多为了发现错误而进行的测试以及由此带来的烦恼。访问 www.jslint.com 并将 JSON 解析到原始区域。单击 JSLint 按钮，网站会验证解析好的 JSON 并突出显示出现的错误。如果显示“JSON:good”，则证明 JSON 是有效的；否则将跳出错误信息。

pass.json 文件创建好并且所有其他图片都准备好后，就要为通行证的发布进行签名和封装了。

25.4　通行证的签名和封装

Passbook 需要通行证以加密的方式进行签名，以此来确保来自提供方并且没有经过任何方式的修改。要对一个通行证签名，需要在 iOSProvisioning Portal 中创建一个 Pass Type ID，并且需要生成一个针对 Pass Type ID 的签名证书。Pass Type ID 和证书都准备好后，就可以对通行证进行签名了。对于每个独立的通行证实例，需要创建一个带有每个文件校验值的清单文件，这样 Passbook 就可以对其中的每个文件进行验证了。

25.4.1　创建 Pass Type ID

Pass Type ID 定义了提供者为通行证设置的类型或分类。例如，如果提供者希望发布一个优惠券和一个回馈卡，提供者可能要创建两个 Pass Type ID，一个用于优惠券，另一个用于回馈卡。要创建 Pass Type ID，访问 iOSDev Center (https://developer.apple.com/devcenter/ios/index.action)，在屏幕右边的 iOS Developer Program 菜单中选择 Certificates, Identifiers & Profiles。单击 Identifiers 后就可以看到左边菜单中显示的 Pass Type ID 元素了，如图 25-9 所示。

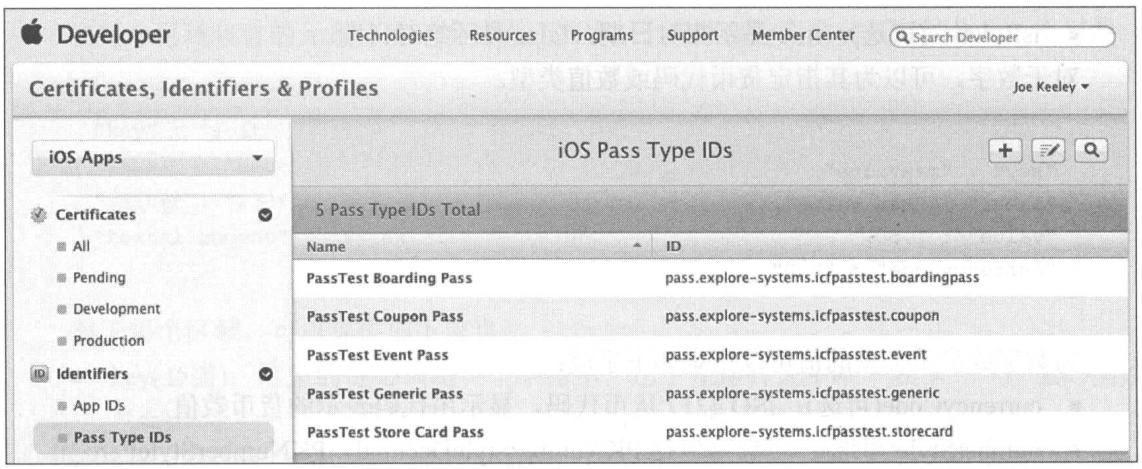


图 25-9 iOS Provisioning Portal: Pass Type IDs

要注册一个新的 Pass Type ID，单击右上角的加号按钮，会显示一个注册新 Pass Type ID 的表格，如图 25-10 所示。

为 Pass Type ID 指定描述信息和标识符。苹果公司建议为 Pass Type ID 使用反向 DNS 命名风格，并要求 Pass Type ID 以字符串“pass”开始。单击 Continue 按钮，会打开确认窗口，如图 25-11 所示。

图 25-10 iOS Provisioning Portal：注册一个新的 Pass Type ID

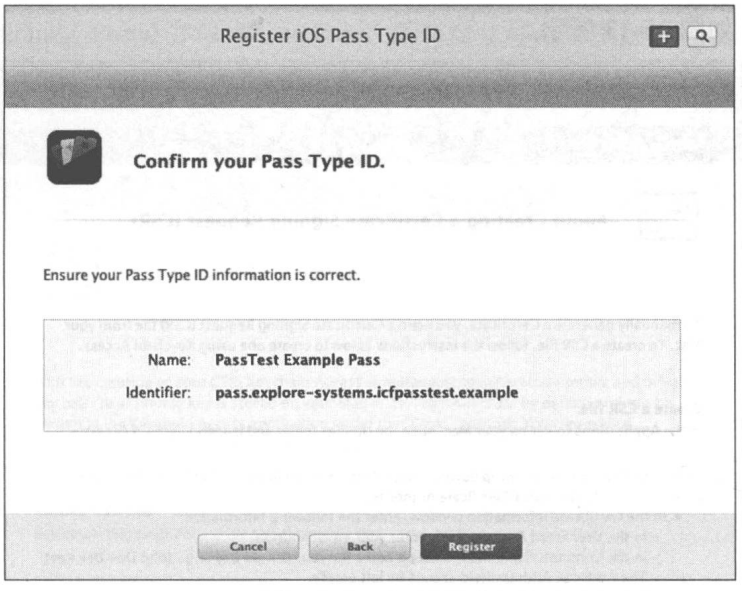


图 25-11　iOS Provisioning Portal：确认 Pass Type ID

单击 Register 按钮，确认通行证类型设置并注册 Pass Type ID。Pass Type ID 注册好后，为了使用新的 ID 对通行证进行签名，还必须为其生成一个证书。

25.4.2　创建通行证签名证书

要查看一个 Pass Type ID 是否已经配置了证书，单击列表中的具体 Pass Type ID，再单击 Edit 按钮。如果该 Pass Type ID 已经创建好了证书，将在 Production Certificates 中显示。还有一个选项可以为 Pass Type ID 创建一个新的证书，如图 25-12 所示。

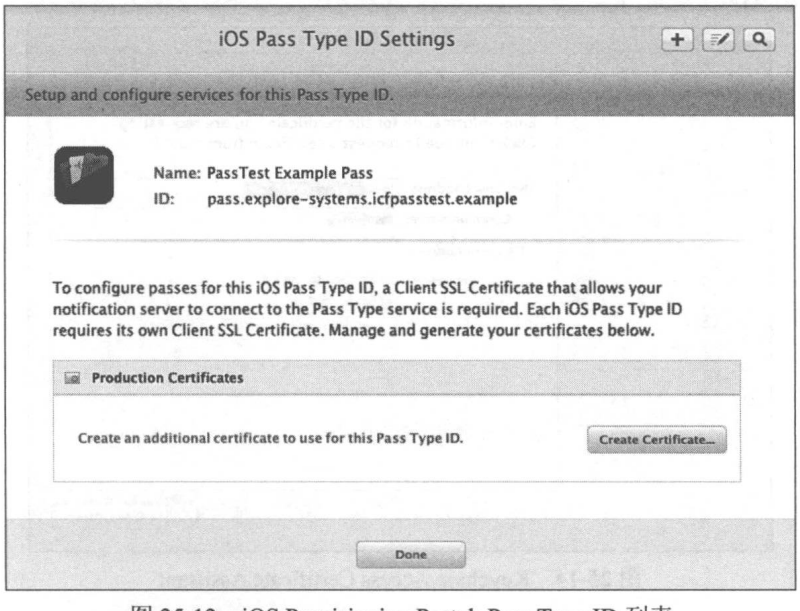


图 25-12　iOS Provisioning Portal: Pass Type ID 列表

单击 Create Certificate 按钮开始证书生成过程。iOSProvisioning Portal 会为证书生成请求

给出引导说明，如图 25-13 所示。

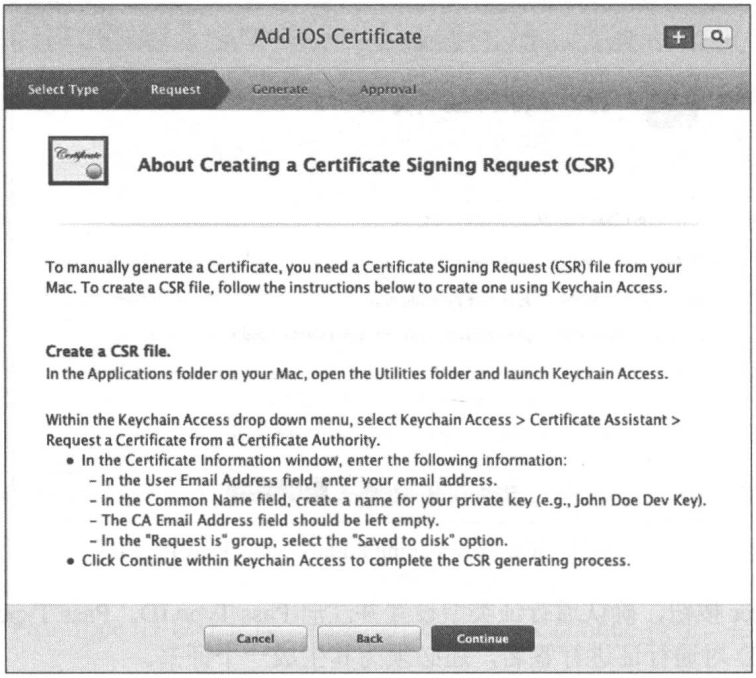


图 25-13 iOS Provisioning Portal: Pass Certificate Assistant，生成一个新的证书签名请求

要生成一个证书请求，在浏览器中保留 iOS Provisioning Portal: Pass Certificate Assistant 页面，同时打开 Keychain Access(在 Applications 的 Utilities 中)。从应用菜单的 Certificate Authority 中依次选择 Keychain Access、Certificate Assistant 和 Request a Certificate。打开的表格如图 25-14 所示。

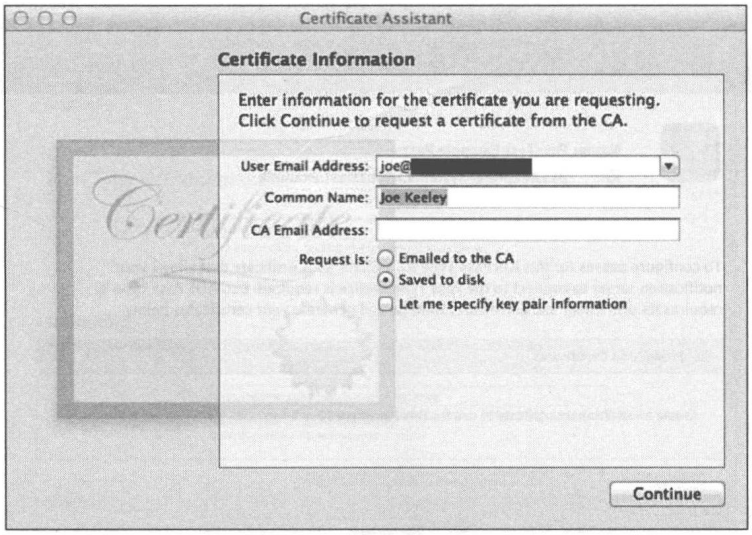


图 25-14 Keychain Access Certificate Assistant

输入邮箱地址和通用的名字(通常是公司名或个人名——使用你认为安全的 Apple Developer 账户)，之后选择 Saved to Disk。单击 Continue，指定保存位置。这一步完成后，返

回到 iOS Provisioning Portal 并单击 Continue。此时会让你选择刚刚保存的请求，如图 25-15 所示。

图 25-15　iOS Provisioning Portal: Pass Certificate Assistant，提交证书签名请求

选择了保存好的请求后，单击 Generate 就会生成 SSL 证书，如图 25-16 所示。

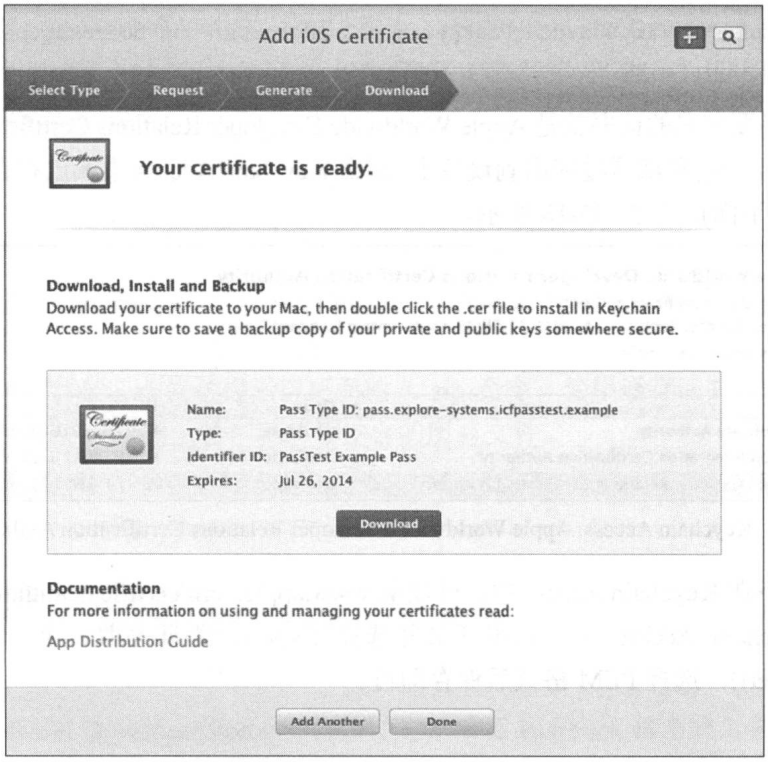


图 25-16　iOS Provisioning Portal: Pass Certificate Assistant，生成通行证证书

证书生成好之后，需要将其下载下来才能使用它对通行证进行签名。单击 Download 按钮下载证书。成功下载后，可以单击 Done 按钮离开 Certificate Assistant。双击下载好的证书文件，将自动安装到 Keychain Access。此时在 Keychain Access 的证书列表中就可以看到该证书了，如图 25-17 所示。单击前面的小三角按钮可以确认证书已经成功创建好了私有键。

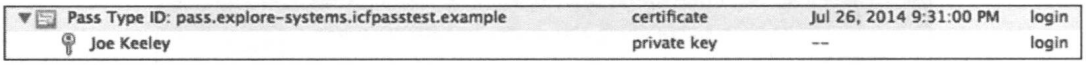


图 25-17 Keychain Access：通行证证书和私有键

如果要在命令行中使用证书进行签名，必须导入证书并将其转换为 PEM 格式。注意，在 Keychain Access 中显示的证书实际上包含一个私有键和一个公共键。私有键用于对通行证进行签名，必须保密以阻止恶意签名。公共键用于签名的外部认证。要导入证书，选中它并右击，之后选择 Export；再为文件选择一个目标位置。Keychain Access 会要求用户提供密码来保护文件——如果只在本地使用并且使用后将其删除，则可以跳过密码。如果文件完全被发送，强烈建议使用比较复杂的密码对其加密。Keychain Access 之后会导出私有键和公共键到一个扩展名为.p12 的文件中。执行下面的命令可以将公共键解析出来并将其保存为 PEM 格式：

```
$ openssl pkcs12 -in BoardingPassCerts.p12 -clcerts -nokeys -out
➥boardcert.pem -passin pass:
```

执行下面的命令，将私有键解析出来并将其保存为 PEM 格式。选择一个密码来替换 mykeypassword。

```
$ openssl pkcs12 -in BoardingPassCerts.p12 -nocerts -out boardkey.pem
➥-passin pass: -passout pass:mykeypassword
```

最后一个需要签名的证书就是 Apple Worldwide Developer Relations CertificationAuthority。如果 Xcode 已经创建和部署过应用到设备上，那么 Keychain Access 中就已存在有效的证书了(在 Certificates 下面)，如图 25-18 所示。

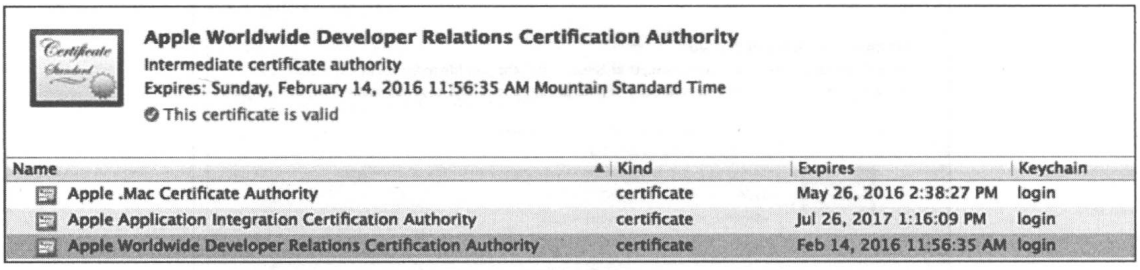


图 25-18 Keychain Access: Apple Worldwide Developer Relations Certification Authority 证书

如果证书不在 Keychain Access 中，可以从 www.apple.com/certificateauthority/下载，并将其安装在 Keychain Access 中。右击证书并选择 Export，为证书取一个简短的名称(例如 AppleWWDRCert)，选择 PEM 格式后保存即可。

25.4.3　创建清单

必须为每个单独的通行证创建清单文件，它是一个名为 manifest.json 的 JSON 文件，包含组成通行证的带有相应 SHA1 校验码的每个文件条目。要创建清单文件，首先在文本编辑器中创建一个新文件。由于文件会生成一个 JSON 数组，因此该部分从一个起始括号开始，最后以括号结尾。对于每个文件，先打上引号，再加上一个冒号，最后添加带有引号的 SHA1 校验码，元素之间用逗号分开。要得到每个文件的 SHA1 校验码，在通行证文件目录下的终端窗口中执行如下命令：

```
$ openssl sha1 pass.json
SHA1(pass.json)= b636f7d021372a87ff2c130be752da49402d0d7f
```

任务完成时清单文件应该如下所示：

```
{
  "pass.json" : "09040451676851048cf65bcf2e299505f9eef89d",
  "icon.png" : "153cb22e12ac4b2b7e40d52a0665c7f6cda75bed",
  "icon@2x.png" : "7288a510b5b8354cff36752c0a8db6289aa7cbb3",
  "logo.png" : "8b1f3334c0afb2e973e815895033b266ab521af9",
  "logo@2x.png" : "dbbdb5dca9bc6f997e010ab5b73c63e485f22dae"
}
```

25.4.4　通行证的签名和封装

清单文件必须签名才能使通行证的内容在 Passbook 中生效。要对清单文件签名，在终端提示符后面使用 openssl 命令。指定 certfile 作为 AppleWorldwide Developer Relations 证书，之前开发者已经创建了 PEM 版本的证书，也创建了 PEM 版本的键作为证书的键，用私有键的密码代替 mykeypassword。

```
$ openssl smime -binary -sign -certfile ../AppleWWDRCert.pem
➥-signer ../boardcert.pem -inkey ../boardkey.pem -in manifest.json
➥-out signature -outform DER -passin pass:mykeypassword
```

创建一个名为 signature 的文件(-out signature)。清单文件中列出的对每个文件的修改都需要更新 SSA 签名，并重新对清单文件签名。

要封装一个通行证，在终端提示符后面输入 zip 命令，使用通行证文件的原始目录。指定通行证的目标文件，并列出通行证所要包含的所有文件。

```
$ zip -r ../boarding_pass.pkpass manifest.json pass.json
➥signature icon.png icon@2x.png logo.png logo@2x.png footer.png
➥footer@2x.png
```

这样就可以在根目录中对所有 boarding_pass.pkpass 归档中列出的文件进行压缩。

注意

苹果公司在 iOS Developer Portal 中提供了一个带有 Passbook 信息的工具叫做 signpass。首先对于 Xcode 项目——在项目编译时会将编译结果输出到一条可以找到的最终路径。之后

执行 signpass，提供通行证目录，将自动创建并对清单文件进行签名，并且对通行证进行封装，只需要一步操作即可。他会使用所有需要证书的 Keychain，所以在开发过程中就不需要导入这些证书了，例如$ ./signpass -p Event.raw 命令将生成 Event.pkpass

25.4.5 测试通行证

要测试通行证，将 boarding_pass.pkpass 文件拖到运行中的模拟器并松开。模拟器会尝试在 Safari 中载入通行证。如果有问题，Safari 会提示错误信息，如图 25-19 所示。

Safari 将所有通行证错误输出到控制台。要找到错误，依次打开 Applications、Utilities、Console，查看错误消息，如图 25-20 所示。

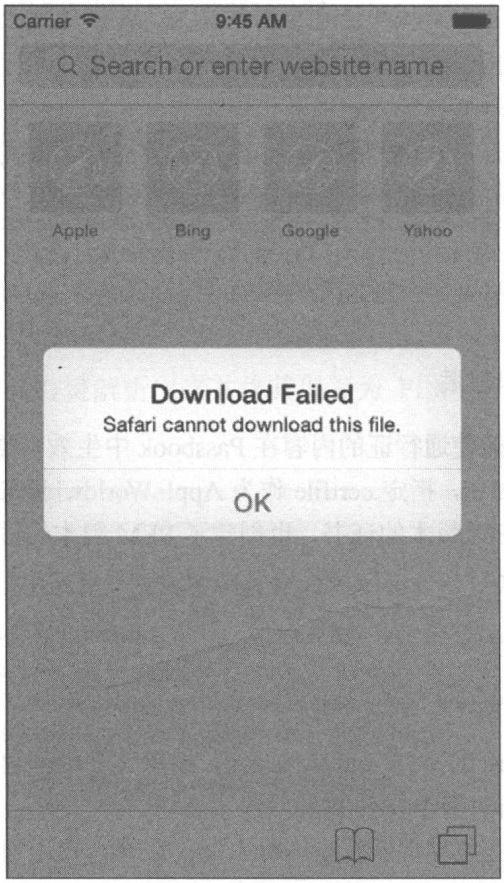


图 25-19 iOS 模拟器中的 Safari：通行证载入错误

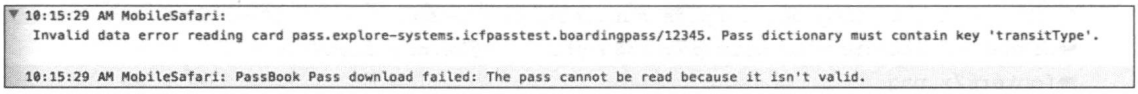


图 25-20 控制台：显示通行证错误

在这个例子里，错误信息显示通行证必须包含一个名为 transitType 的键。这个键是登机牌类通行证必需的，其他类型的通行证不能用。所以要确保 pass.json 文件的登机牌通行证部分一定包含 transitType 键，重新对通行证签名，并将其再次拖到模拟器中查看错误是否被修复。

一定要单击 Add 按钮在模拟器中向 Passbook 添加一个通行证，这是因为并不是所有错误都是在显示通行证时发现的。当 Add 按钮被单击时将以动画效果向 Passbook 中添加通行证。如果没有出现动画而通行证逐渐消失了，即通行证出现了错误，无法将其添加到 Passbook。查看控制台找到所有错误。

25.4.6　具体应用中的通行证交互

通行证可以完全置于应用之外——实际上，不需要为通行证产生的全过程而特意定制应用。不过在定制应用中会有许多这样的案例，例如创建新的通行证、处理已存在的通行证的更新以及删除通行证。示例程序会演示如何执行这些任务。

准备应用

应用要和通行证进行交互需要完成几个准备步骤。首先确保 PassKit.framework 已被添加到项目中，并且在每个使用 PassKit 的类中都加上了@import PassKit。

接下来，回到 iOS Provisioning Portal 并选择左边菜单的 App IDs。单击加号按钮创建一个新的 App ID，如图 25-21 所示。

图 25-21　iOS Provisioning Portal：创建一个新的 App ID

在创建新的应用时可以使其支持 Passbook，或者在创建好后也可以进行设置。要为一个 App ID 设置支持 Passbook，单击列表中的具体 App ID，单击 Edit 按钮。选中 Enable Passes

选项，iOS Provisioning Portal 会生成一个提醒对话框，告诉开发者如何支持通行证功能，该 App ID 对应的所有配置文件都必须重新生成，如图 25-22 所示。

图 25-22 iOS Provisioning Portal：为 App ID 设置通行证功能

在 Xcode 中，选择项目，之后选择目标，选中 Capabilities 页面，如图 25-23 所示。设置 Passbook 选项为 On 可以让 Xcode 检查 PassKit 框架是否已经关联到项目中、需要的授权项目是否已配置正确以及配置文件是否已经设置正确。

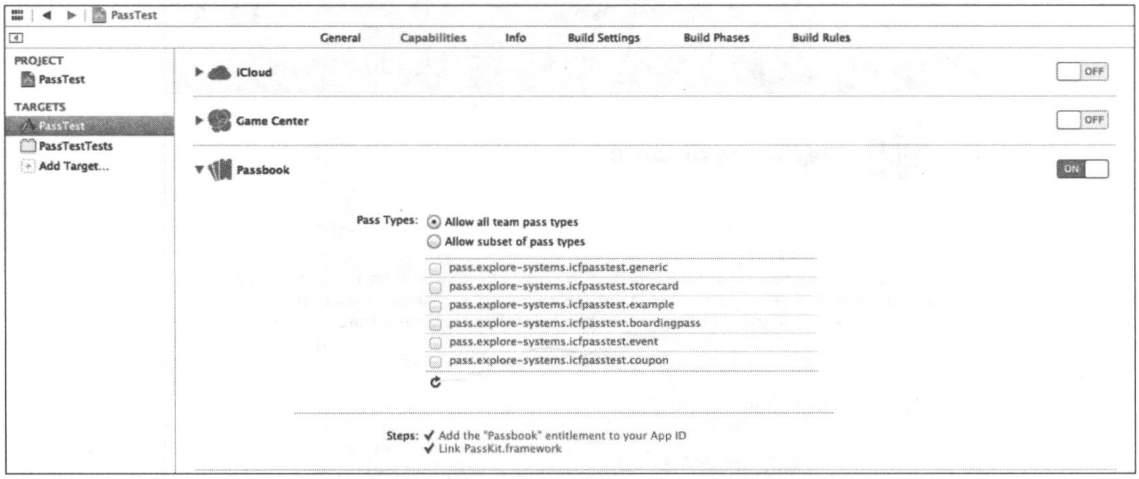


图 25-23 Xcode 的 Capabilities 页面显示 Passbook 部分

注意

这些准备步骤只针对应用在真机上运行的情况。如果是在模拟器中同 Passbook 进行交互，则不需要这些设置。

现在应用已经设置好在 Passbook 中访问通行证的功能了。示例程序在主 bundle 中包含了每一种通行证的演示。不过通行证一般并不是这样发送的，而是在用户提供了一些关于通行证的信息后从服务器下载的。要查看如何使用程序同 Passbook 进行交互，打开示例程序并单击任一种通行证(本例中演示的为登机牌类通行证)。应用会检查一共有多少通行证在 Passbook 中，然后确定选中的通行证是否在 Passbook 中，如图 25-24 所示。

要得到此信息，视图控制器需要同通行证库进行通信。为了方便，创建一个属性来保存

PKPassLibrary 实例，在 viewDidLoad 方法中对其初始化。

```
-(void)viewDidLoad
{
    [super viewDidLoad];

    self.passLibrary = [[PKPassLibrary alloc] init];
    [self refreshPassStatusView];
}
```

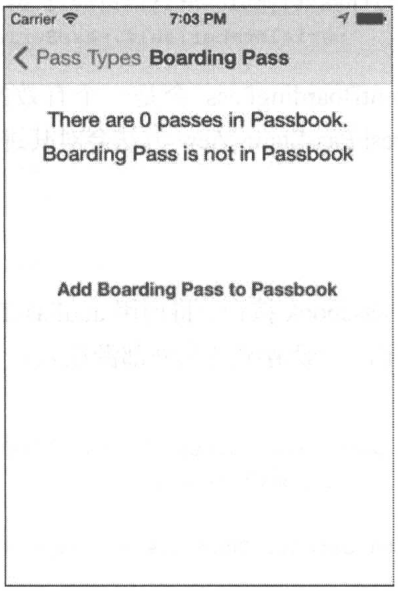


图 25-24　Pass Test 示例程序：登机牌类通行证

在 refreshPassStatusView 方法中，视图控制器首先会检查通行证库是否可用。

```
if(![PKPassLibrary isPassLibraryAvailable])
{
    [self.passInLabel setText:@"Pass Library not available."];

    [self.numPassesLabel setText:@""];
    [self.addButton setHidden:YES];
    [self.updateButton setHidden:YES];
    [self.showButton setHidden:YES];
    [self.deleteButton setHidden:YES];
    return;
}
```

如果通行证库不可用，就不会有后续的操作，所以该方法会更新 UI 并隐藏所有的按钮。如果通行证库可用，该方法会从 passLibrary 对象获取信息来更新 UI。要确定其中有多少通行证，访问 passLibrary 的 passes 属性即可。

```
NSArray *passes = [self.passLibrary passes];

NSString *numPassesString =
```

```
➥[NSString stringWithFormat:
➥@"There are %d passes in Passbook.",[passes count]];

[self.numPassesLabel setText:numPassesString];
```

passLibrary 会提供一个方法来通过通行证类型标识符和通行证序列号访问指定的通行证。还可以用这个方法来确定指定的通行证是否在通行证库中。

```
PKPass *currentBoardingPass =
➥[self.passLibrary passWithPassTypeIdentifier:self.passIdentifier
                                  serialNumber:self.passSerialNum];
```

如果通行证在库中，currentBoardingPass 会是一个有效的 PKPass 实例；如果不在，currentBoardingPass 为 nil。refreshPassStatusView 方法会对其进行检查并根据结果相应地更新 UI 界面。

添加通行证

点击 Add Boarding Pass to Passbook 按钮，将调用 addPassTouched:方法。这个方法首先会从主 bundle 中载入通行证(同样，一般情况是从外部源载入)。

```
NSString *passPath =
➥[[NSBundle mainBundle] pathForResource:self.passFileName
                                  ofType:@"pkpass"];

NSData *passData = [NSData dataWithContentsOfFile:passPath];

NSError *passError = nil;
PKPass *newPass = [[PKPass alloc]
➥initWithData:passData error:&passError];
```

如有任何错误，PassKit 会验证通行证数据并在 passError 中返回一个错误。如果通行证有效但不在库中，则会给出一个 PKAddPassesViewController，显示这个将要呈现在 Passbook 中的通行证，并根据用户是选择 Add 还是 Cancel 来对其进行管理。否则，该方法会显示一个带有错误描述消息的提醒框。

```
if(!passError && ![self.passLibrary containsPass:newPass])
{
    PKAddPassesViewController *newPassVC =
    ➥[[PKAddPassesViewController alloc] initWithPass:newPass];

    [newPassVC setDelegate:self];

    [self presentViewController:newPassVC
                       animated:YES
                     completion:^(){}];

}
else
```

```
{
    NSString *passUpdateMessage = @"";

    if(passError)
    {

        passUpdateMessage =
        ➥[NSString stringWithFormat:@"Pass Error: %@",
        ➥[passError localizedDescription]];

    }
    else
    {
        passUpdateMessage = [NSString stringWithFormat:
        ➥@"Your %@ has already been added.", self.passTypeName];
    }

    UIAlertController *alertController =
    ➥[UIAlertController alertControllerWithTitle:@"Pass Not Added"
                                          message:passUpdateMessage
                                   preferredStyle:UIAlertControllerStyleAlert];

    [alertController addAction:
    ➥[UIAlertAction actionWithTitle:@"Dismiss"
                               style:UIAlertActionStyleCancel
                             handler:nil]];

    [self presentViewController:alertController
                       animated:YES
                     completion:nil];
}
```

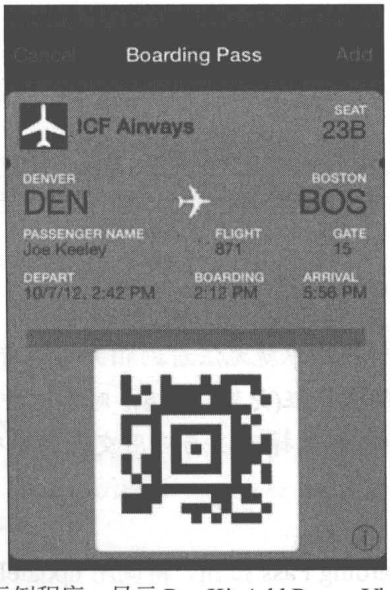


图 25-25　示例程序：显示 PassKit Add Passes View Controller

用户选择添加通行证后，如果设置了委托方法，PKAddPassesViewController 会调用该委托方法。

```
-(void)addPassesViewControllerDidFinish:
➥(PKAddPassesViewController *)controller
{
    [self dismissViewControllerAnimated:YES completion:^{
        [self refreshPassStatusView];
    }];
}
```

委托方法负责移除 PKAddPassesViewController。移除 PKAddPassesViewController 后，更新 UI 显示新加入的通行证，如图 25-26 所示。

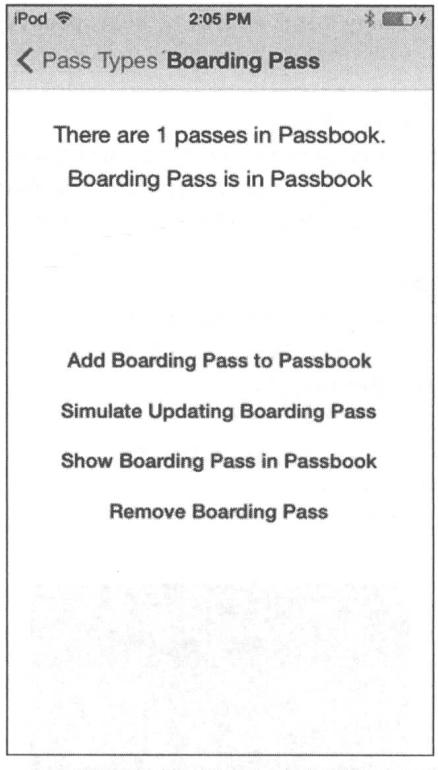


图 25-26 Pass Test 示例程序：通行证库中新加入的登机牌通行证

注意

如果应用的 bundle ID 和通行证的 ID 不匹配(例如 com.myorg.mybundleid 和 pass.myorg.someotherid)，在通行证库中就无法看到相关的通行证，这样在添加通行证后，通行证库就会报告库中没有可用的通行证(见图 25-24)。对于这种情况，在 Capabilities 标签页中选择 Allow Subset of Pass Types，并选择可以与应用交互的通行证。

模拟更新通行证

点击 Simulate Updating Boarding Pass 按钮，将调用 updatePassTouched 方法。这个方法首先从主 bundle 中载入更新好的通行证数据(这里仅模拟这一过程，通常更新好的通行证都是

根据修改情况从服务器下载得到的)，之后初始化一个 PKPass 对象。

```
NSString *passName =
➥[NSString stringWithFormat:@"%@-Update",self.passFileName];

NSString *passPath =
➥[[NSBundle mainBundle] pathForResource:passName ofType:@"pkpass"];

NSData *passData = [NSData dataWithContentsOfFile:passPath];

NSError *passError = nil;

PKPass *updatedPass = [[PKPass alloc] initWithData:passData
                                             error:&passError];
```

该方法会检查在初始化通行证的过程中是否有错误，以及通行证库中是否已包含通行证。如果没有错误且有已存在的通行证，则使用更新后的通行证替换已存在的通行证。

```
if(!passError && [self.passLibrary containsPass:updatedPass])
{

    BOOL updated = [self.passLibrary
    ➥replacePassWithPass:updatedPass];

    if(updated)
    {
        passUpdateMessage = [NSString stringWithFormat:
        ➥@"Your %@ has been updated.",self.passTypeName];

        passAlertTitle = @"Pass Updated";
    }
    else
    {
        passUpdateMessage = [NSString stringWithFormat:
        ➥@"Your %@ could not be updated.",self.passTypeName];

        passAlertTitle = @"Pass Not Updated";
    }

    UIAlertController *alertController =
    ➥[UIAlertController alertControllerWithTitle:passAlertTitle
                                          message:passUpdateMessage
                                   preferredStyle:UIAlertControllerStyleAlert];

    [alertController addAction:
    ➥[UIAlertAction actionWithTitle:@"Dismiss"
                               style:UIAlertActionStyleCancel
                             handler:nil]];

    [self presentViewController:alertController
```

```
                    animated:YES
                  completion:nil];
}
```

replacePassWithPass:方法将指明通行证是否成功完成更新，并显示一个提醒框给用户。

如果更新对时效性要求很高，且用户需要立即得知更新结果所包含的关键信息，可以为被更新通行证的 pass.json 文件指定 changeMessage 参数。

```
"headerFields" : [
  {
    "key" : "seat",
    "label" : "Seat",
    "value" : "14C",
    "textAlignment" : "PKTextAlignmentRight",
    "changeMessage" : "New Seat: %@"
  }
],
```

指定修改消息后，当通行证更新时 Passbook 将显示一个通知给用户，如图 25-27 所示。通知将显示通行证包含的图标、组织机构名称和相应的消息。如果在 changeMessage 中使用了%@，那么在为用户显示的 changeMessage 中，%@的位置将被一个新的值代替。如果 changeMessage 中没有使用%@，将显示一条通用消息，例如“Boarding Pass changed”。

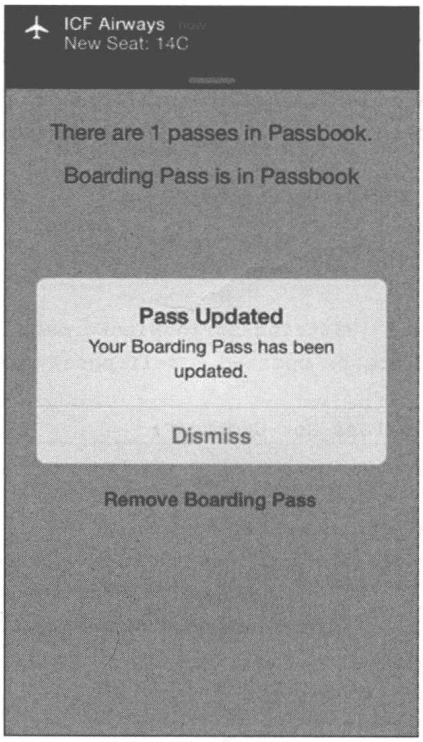


图 25-27　通行证修改提醒

该消息会一直保存在 Notification Center 中，直到用户删除它。

显示通行证

要显示已存在的通行证，点击 Show Boarding Pass in Passbook 按钮，此时会调用 showPassTouched 方法。由于 PassKit 不支持在应用中显示通行证，因此该方法需要得到通行证的公共 URL，再令应用打开这个链接。这样就可以直接在 Passbook 中打开所需的通行证了。

```
PKPass *currentBoardingPass =
➥[self.passLibrary passWithPassTypeIdentifier:self.passIdentifier
                                  serialNumber:self.passSerialNum];

if(currentBoardingPass)
{
    [[UIApplication sharedApplication]
    ➥openURL:[currentBoardingPass passURL]];
}
```

删除通行证

要从应用中直接删除通行证，点击 Remove Boarding Pass 按钮，此时会调用 deletePassTouched 方法。该方法会使用通行证的标识符和序列号定位到具体的通行证，并将其从 Passbook 中删除。

```
PKPass *currentBoardingPass =
➥[self.passLibrary passWithPassTypeIdentifier:self.passIdentifier
                                  serialNumber:self.passSerialNum];

if(currentBoardingPass)
{
    [self.passLibrary removePass:currentBoardingPass];

    [self refreshPassStatusView];

    NSString *passUpdateMessage =
    ➥[NSString stringWithFormat:@"Your %@ has been removed.",
    ➥self.passTypeName];

    ➥UIAlertController *alertController =
    [UIAlertController alertControllerWithTitle:@"Pass Removed"
                                        message:passUpdateMessage
                                 preferredStyle:UIAlertControllerStyleAlert];

    [alertController addAction:
    ➥[UIAlertAction actionWithTitle:@"Dismiss"
                               style:UIAlertActionStyleCancel
                             handler:nil]];

    [self presentViewController:alertController
                       animated:YES
```

```
                completion:nil];
}
```

25.5 自动更新通行证

Passbook 的一个核心功能就是可以自动更新通行证，而不需要用到具体的应用。这是一个整体功能，因为实现这个功能需要一台能够创建并更新通行证的服务器，完全实现这台服务器超出了本章的讨论范围。

如果要支持通行证更新，pass.json 文件需要指定一个 webServiceURL 和一个 authenticationToken。当通行证首次被添加时，Passbook 将调用 webServiceURL，在服务器上注册设备和通行证，并在下一步中为用户提供一个推送令牌(token)。

当有关该通行证的更新发生在服务器上时，服务器需要通知带有该通行证的设备此时更新是有效的。服务器会发送一个带有推送令牌的推送通知，该推送令牌是在之前注册时生成的，并在推送中包含一个类型 ID。

注意

欲了解更多有关发送推送通知的信息，可以参考第 10 章“推送通知”。

设备接收到推送通知后，Passbook 会根据指定的类型 ID 和最后一个更新标签从服务器发起一个请求，即获取发生修改的通行证列表。服务器将返回一个序列号列表和一个最近更新的标签。

设备之后会遍历这些序列号，并为每个序列号从服务器请求更新的通行证版本。如果更新的通行证包含 changeMessage(在之前的“模拟更新通行证”一节中有介绍)，Passbook 将显示一个通知给用户。

使用这个机制，用户的通行证可以保持实时更新，并且当关键和时效性高的信息发生变化时也可以及时得到通知。

25.6 小结

本章深入介绍了有关 Passbook 的使用，介绍了 Passbook 是什么以及 Passbook 所支持的通行证类型，还讲解了如何设计和创建通行证，以及如何对通行证进行签名和封装。本章演示了如何在应用中使用 PassKit 实现通行证和 Passbook 的交互，并讨论了如何使用 Web 服务器保持通行证的实时更新。

第26章 调试和工具

同本书大部分其他章节不同，本章并没有示例代码和具体的项目。通过前面的学习，读者应该已经掌握了一些 iOS SDK 高级特性和功能的实现方法。本章主要关注如何处理程序中出现的一些错误。调试和性能优化对于任何一款软件来说都至关重要，有时在开发过程中会忽略程序调试。用户希望一款应用运行迅速、平滑、连贯且没有错误，更不会无缘无故地崩溃。不管多么出色的开发人员编写的程序都有可能出现错误，都有可能出现程序崩溃的情况，此时再讨论性能就没有什么意义了。本章使用系统之外的工具辅助软件的开发，将性能提升到最优标准。

26.1 调试

“如果调试是删除错误，那么程序开发就是不断制造错误的过程。”

——Edsger W. Dijkstra

计算机很复杂——复杂到几乎很少有人能够对它工作原理的所有方面都了如指掌。只有很少数的开发者能够使用二进制或汇编语言进行程序开发，尽管它们对于计算机本身来说是易于解析的。复杂就意味着有时候看起来正确的代码也会出现错误。对于有些错误，比如争用条件(race conditions)和线程安全，都是很难计划且不好排除的。

当能够很好地利用调试器提供的技术时，软件调试就变得非常简单了。从使用定制断点到 NSZombies 参数，大部分调试难题都迎刃而解。同学习其他所有的规则一样，调试也需要时间和练习，在寻求帮助前通常需要自己尝试找到解决办法。理解解决问题和找到解决方法之间的不同对一名程序员的发展至关重要。

26.1.1 第一个计算机错误

1947 年，第一台计算机在大型公司、高校和政府研究机构的努力下终于能够正式向媒体曝光了。Grace Murray Hopper 曾是哈佛大学的早期 Mark II Aiken Relay Calculator 系统开发人员之一。1947 年 9 月，该机器开始出现问题，工程师也随即展开调查。他们的发现令人惊讶，

不过对于有一间屋子那么大的计算机来说出现问题也并不是完全出乎意料。一只普通的飞蛾被困在了 MarkII Aiken Relay Calculator 的 F 窗格 70 号继电器之间了。飞蛾阻止了本该工作的继电器的运行，机器在不断报错。知道这段历史的工程师将这个飞蛾用胶带和手写记录保存下来了，“第一个真实案例的错误被找到了”(如图 26-1 所示)。今天，第一个计算机错误被保存在维吉尼亚州达尔格伦的海军水面作战中心计算机博物馆中。

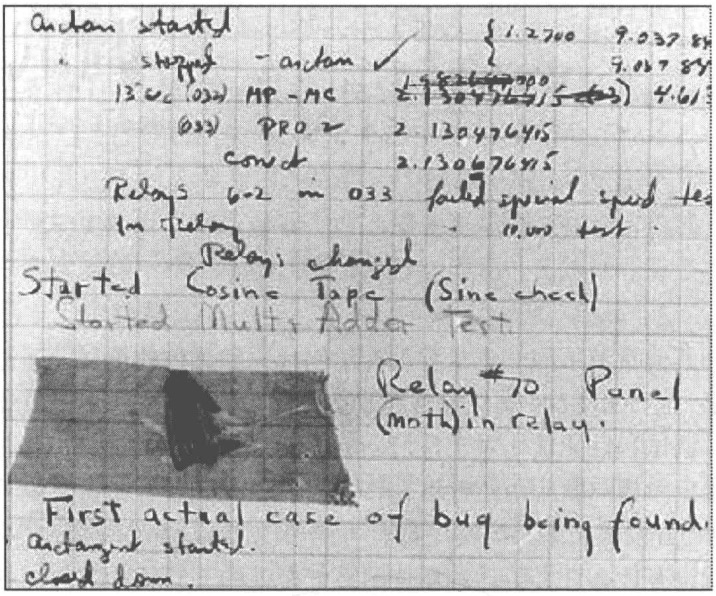


图 26-1 来自 1947 年 Mark II Aiken Relay Calculator 的第一个计算机错误

26.1.2 Xcode 基础调试

同目前大部分 IDE 类似，Xcode 也带有内置的调试器，称作 LLDB。计算机执行代码的速度非常快，人们无法看到程序具体的执行步骤。这就是调试器的作用所在，它可以为开发者放慢程序的执行并检查其中元素的状态。调试视图最初是隐藏的，使用中间的视图按钮可以访问它，如图 26-2 所示。只有程序在 Xcode 中运行时才可以使用调试器。

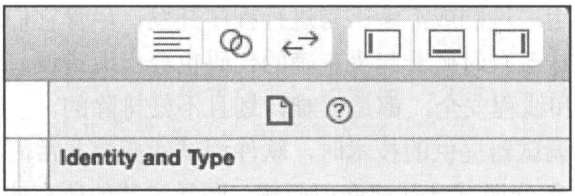


图 26-2 通过下面视图区域的设置在 Xcode 中访问调试器

调试区域(如图 26-3 所示)包括 3 个主要部分：左边是变量视图，用于检查内存中当前对象的详细信息；右边是控制台组合视图，它还包括调试提示符；视图的上面是调试命令栏，用于与调试器交互。

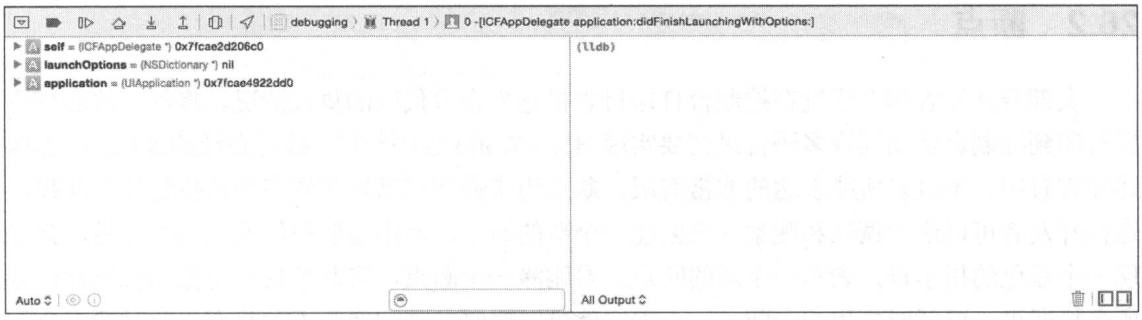


图 26-3　调试区域

默认状态下，调试器会自动显示遇到的异常。开发者可以通过调试工具栏上的 Pause 按钮暂停当前执行的程序并跳出调试器。

如果调试器在一个异常点停住了，通常可以跳过这个异常继续运行程序，为此，使用工具栏上的 Resume 按钮即可。工具栏还提供了一些其他有用的命令。在工具栏上，暂停时点击 Step Over 命令即执行下一条处理指令。Step Into 命令让程序执行到当前停止的一个新的方法或函数中。同样，Step Out Of 按钮让程序离开当前方法或函数。

此外，在调试器的工具栏上可以查看每个线程的执行情况，显示栈跟踪(stack trace)。栈跟踪会提供到目前位置的时间序列。可以使用 Debug Navigator 访问和上面同样的信息，在 Xcode 窗口的最左边窗格中可以找到 Debug Navigator，如图 26-4 所示。

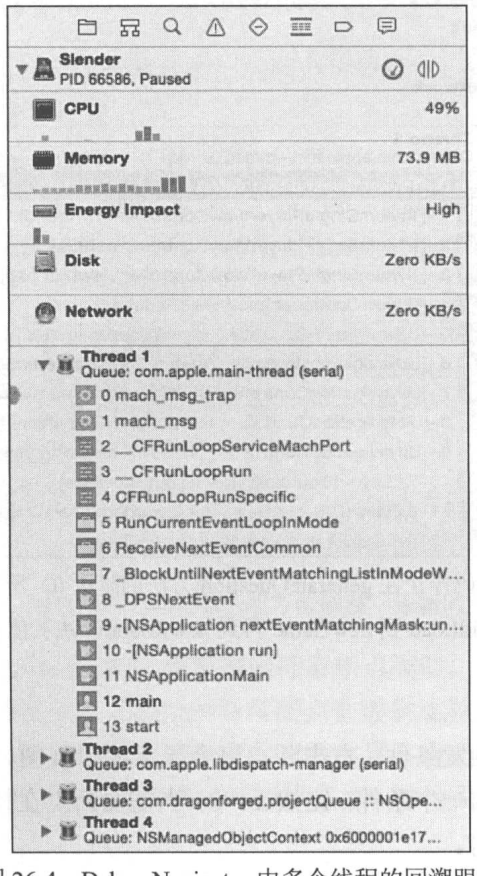


图 26-4　Debug Navigator 中多个线程的回溯跟踪

26.2 断点

大部分开发者都会通过在控制台打印日志描述来查看代码的执行情况。将程序的运行情况打印到控制台就如同许多语言最初要实现 Hello World 程序一样，都是必经的过程。日志描述非常有用，不过它所能表达的非常有限。断点用来通知调试器代码在当前执行位置挂起，这样开发者可以进行调试和观察。要创建一个新的断点，点击需要暂停的代码的行号，会出现一个蓝色的指示符，表示一个新的断点；要删除一个断点，将蓝色指示符拖出代码行标号所在栏即可；要暂时禁用一个断点，点击它就可以暂时将其切换为无效状态，此时断点会变成稍微有点透明的蓝色。

遇到一个断点后，代码的执行就会暂停。变量视图将会显示当前域的所有变量，栈跟踪界面会显示断点之前方法的路径和调用情况。由开发者撰写的调用函数将会以黑色显示，系统调用函数将会以浅灰色显示。开发者可以在栈跟踪中找到调用某一函数的代码行，如图 26-5 所示。

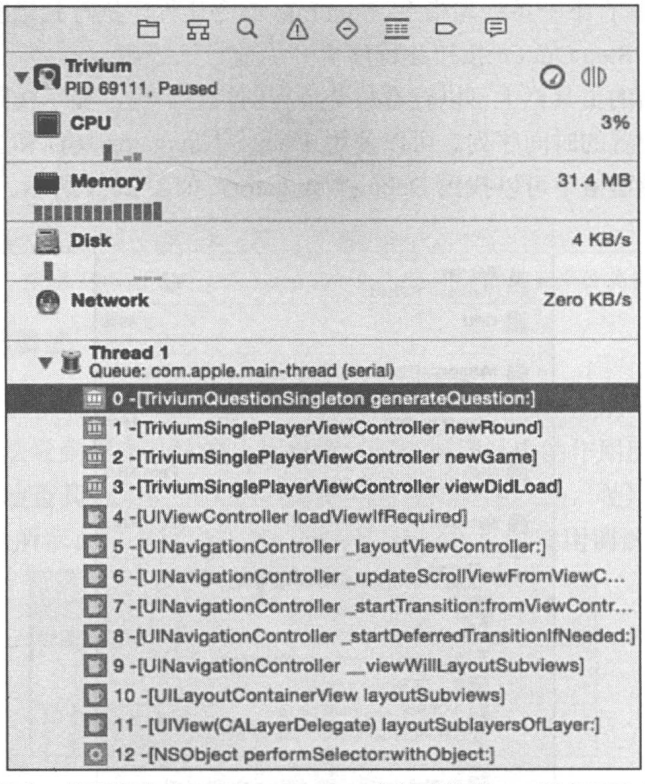


图 26-5 常见的栈跟踪。代码停止在 generateQuestion:方法的阶段 0。所有和该方法相关的事件都可以看到，从 viewDidLoad 到 newGame 再到 newRound。浅灰色表示的方法为系统调用

26.2.1 定制断点

可以定制断点来修改触发条件。右击断点将会显示编辑视图，如图 26-6 所示。首先可以被定制的属性是向断点添加一个条件，比如 x==0。当需要断点在特定的条件下激活时可以这样使用，比如当 x 等于 0 时。

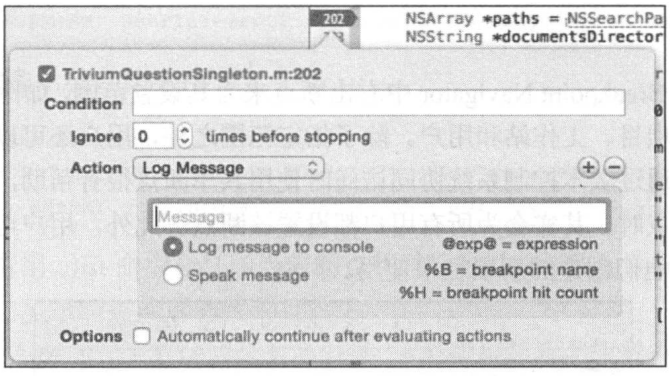


图 26-6　定制一个断点

开发者可能需要根据实际情况在程序最初几次运行时忽略一些断点。比如，一个错误可能在一行代码执行多次之后才会出现，所有设置条件都可以避免不断地单击 Continue 按钮。

断点还可以关联动作，比如运行一个 AppleScript、执行一个调试器命令、运行一个 Shell 命令、输出信息或播放声音。可以让播放声音这个动作作为事件发生的音频提醒，这很有帮助，比如在网络连接或 Core Data 数据合并时播放。在特定的条件下，比如播放音频，开发者可能不希望代码因为断点而暂停执行。如果首选的动作是输出信息或播放声音而不暂停程序运行，可以设置 Evaluating Actions 选项后面的 AutomaticallyContinue 来实现这个功能。

26.2.2　标志断点和异常断点

除了用户设置的断点之外，还有两种类型的断点可以使用。可以通过 Xcode 窗口左侧窗格的 Breakpoint Navigator 进行设置。

标志断点(symbolic breakpoint)可以用于捕捉所有正在运行的方法或函数的实例。比如，输出每个被调用的 imageNamed: 实例，可以创建标志断点，设置标志为 +[UIImageimageNamed:]。图 26-7 显示了一个标志断点，它会以播放声音的形式输出每个 imageNamed:。

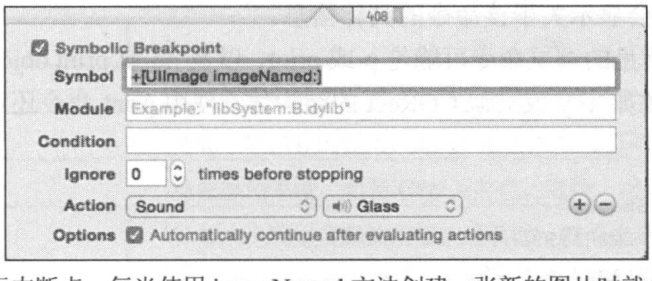


图 26-7　标志断点，每当使用 imageNamed:方法创建一张新的图片时就会播放声音

异常断点(exception breakpoint)的工作原理同标志断点类似，只不过是在任何时候出现异常都会执行该断点。通常情况下，设置全局异常断点会提供比程序崩溃事件更好的栈跟踪效果。这是因为栈跟踪是基于断点的，而崩溃事件可能是异常情况的结果，而不必找到最初的起始原因。大部分的开发者认为最好在调试时始终保持一个全局异常断点。

26.2.3 断点范围

还可以通过在 Breakpoint Navigator 中右击断点来为其设置范围，如图 26-8 所示。断点有效范围的选项包括项目、工作站和用户。除了指定范围之外，用户还可以创建共享断点。当程序由多个开发者通过版本控制系统协同访问时使用共享断点很有帮助，很重要的一个原因是，当设置一个断点时，其实会为所有用户都设置该断点。此外，用户可以让断点作为用户断点，可以在所有他们创建的新项目中使用。

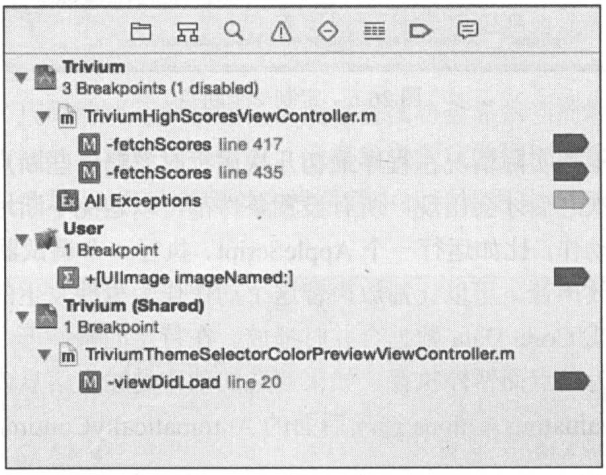


图 26-8 在 Xcode 中创建不同的断点范围，包括用户断点和共享断点

26.3 使用调试器

Xcode 有一个非常先进且健全的调试器：LLDB。可以在代码运行停止的任何时刻访问该调试器。一个 lldb 提示符会出现在控制台窗口的下面。虽然调试器是一个非常大且复杂的系统，不过只有一部分命令是和 iOS 开发者相关的。

解决不明问题的第一个命令是 help 或 h。help 命令会打印一个根级别的帮助菜单，任何命令的 help 选项都会显示关于该命令的具体信息。

iOS 开发者最常见的调试命令可能是 p 或 print，以及 po 或 print object。print 命令会打印标量表达式的值，比如 x+y 或类似 CGRect 的结构体。使用 print 命令还可以在调试器中修改变量的值。

```
(lldb) p scaleStage2
(float) $0 = 0.600019991
(lldb) p scaleStage2 = 5.25
(float) $1 = 5.25
(lldb) p scaleStage2
(float) $2 = 5.25
```

print object (po)命令将会令一个 Objective-C 对象打印它的描述信息。比如，要查看一个内存地址的内容，可以在 lldb 提示符的后面输入如下命令：

```
(lldb) po 0x7c025990
<UIImageView: 0x7c025990; frame = (0 0; 320 480); opaque = NO;
```

```
➥autoresize = RM+BM; userInteractionEnabled = NO; layer = <CALayer: 0x7c025ad0>>
```

也可以使用对象名称，比如：

```
(lldb) po backgroundView
<UIImageView: 0x7c025990; frame = (0 0; 320 480); opaque = NO;
➥autoresize = RM+BM; userInteractionEnabled = NO; layer = <CALayer: 0x7c025ad0>>
```

list 命令也很有用。list 命令会打印当前断点代码行附近的代码。此外，list 命令还可以使用+/- X 参数来指定显示断点前后的代码行。

有时候开发者在调试一个方法或函数的时候，需要覆盖其返回值或提前给出一个返回值。可以使用 return x 命令实现这个功能。比如在 continuing 的后面输入 return 0 会模拟在断点位置的函数成功返回。

backtrace 命令或 bt 可以用于打印当前的回溯情况到控制台。虽然这会对调试有所帮助，不过此信息通常只会以一种用户友好的格式在 Debug Navigator 出现。

除了这些命令之外，基础工具命令也可以在调试提示符的后面执行，这样一般会比在工具栏上查找那些很小的按钮要容易。step 或 s 命令可以在执行过程中向下移动一行。continue 或 c 命令可以越过当前断点让程序继续执行。fin 命令会持续执行程序直到方法结束，这个方法没有对应的工具栏选项。最后，kill 命令可以用于终止程序。

LLDB 是一个功能强大的工具，它提供了大量实用且灵活的调试方法。要了解 LLDB 更详细的内容，可以参考官方文档，网址为 http://llvm.org/docs/。

26.4　工具

“Instruments(工具)”是指 Xcode 绑定的多个分析工具的统称。表 26-1 显示了一个绑定工具的列表。虽然这些工具详细的内容和行为都可以写一本书了，不过关于如何阅读和同这些工具互动的基础知识对于普通 iOS 开发者来说已经足够满足大部分需求了。

表 26-1　Xcode 提供的工具及其功能

工　具	描　述
Allocations	通过跟踪内存分配情况计算堆内存的使用
Leaks	计算一般内存的使用，检查内存泄漏并提供对象使用内存的统计数据
Activity Monitor	动态检测系统，包括 CPU、内存、磁盘、网络和统计情况
Zombies	计算当出现过度释放的僵尸对象时一般内存的使用情况。还按照类提供对象分配内存的统计，记忆之前所有已经分配过的内存地址
Time Profiler	执行一个运行在系统 CPU 之上的低消耗的基于时间的进程样本
System Trace	通过显示安排好的线程提供关于系统行为的信息，以及显示所有从用户到系统代码的转换
Automation	执行一个脚本，模拟从工具启动 iOS 应用的 UI 交互
File Activity	动态监视文件和目录，包括文件打开或关闭的调用、文件任务修改、创建目录、文件移动等

(续表)

工　具	描　述
Core Data	动态跟踪 Core Data 文件系统，包括获取、快取失效和保存
Energy Diagnostics	提供关于电量使用的诊断，以及主要设备组件的开关状态
Network	分析 TCP/IP 和 UDP/IP 连接情况
System Usage	动态记录从工具启动的一个进程有关文件、socket 和共享内存的 I/O 系统使用情况
Core Animation	监视图形性能和使用 Core Animation 处理时 CPU 的使用情况
OpenGL ES Driver	计算 OpenGL ES 图形性能，以及处理过程中 CPU 的使用情况
OpenGL ES Analysis	动态计算和分析 OpenGL ES 来检测 OpenGL ES 的准确性和性能问题，还提供对于解决这些问题的建议
Cocoa Layout	观察并调试 NSLayoutConstraint 对象来帮助布局约束修改错误
Counters	使用基于时间和事件的样本方法采集遇到事件的性能结果
Dispatch	动态监视和计算分发队列和期间唤起的代码块
Multicore	计算多核的性能，包括线程状态、分发队列和代码段的使用情况
UI Recorder	允许开发者捕捉用户交互并在之后执行
Sudden Termination	分析目标进程突然终止的情况，为访问的文件系统报告回溯的情况

注意

并不是所有的工具在特定的环境下都可以使用，这一点一定要明确，比如 Core Data 工具只有在模拟器运行时才有效，Network 工具只有程序在物理设备上运行时才有效。

26.4.1 工具界面

要访问 Xcode 中的工具界面，首先要选择编译目标，选择模拟器或真机，之后在 Product 菜单中选择 Profile。将会出现一个新的窗口(如图 26-9 所示)，在这里用户可以选择他们希望使用的工具类型。之后，可以从库中将额外的元素添加进来(如图 26-10 所示)。

工具界面本身会根据你明确选中的工具而包含很多部分。

在 Instruments 窗口工具栏的上面显示了多种控件，比如暂停、记录和重启当前目标的执行。此外，还可以从所有运行的进程中选择新的目标。用户还可以设置所有工具的观察范围，过滤掉左右标识之间的部分。

工具应用还会保存每个应用的运行情况，这样可以方便比较程序修改对性能的改变。用户还可以从库中添加新的工具以实现组合测试的目的。左边的视图，可以切换为视图菜单，其中包含对于选中工具的一些设置。下面的视图包含关于测试的一些详细信息，比如 Call Tree 或 Statistics；这些内容也会根据选择的工具不同而不同。最后一个视图是右边的扩展信息视图，它通常包含当前中央视图选择元素的一些回溯信息。

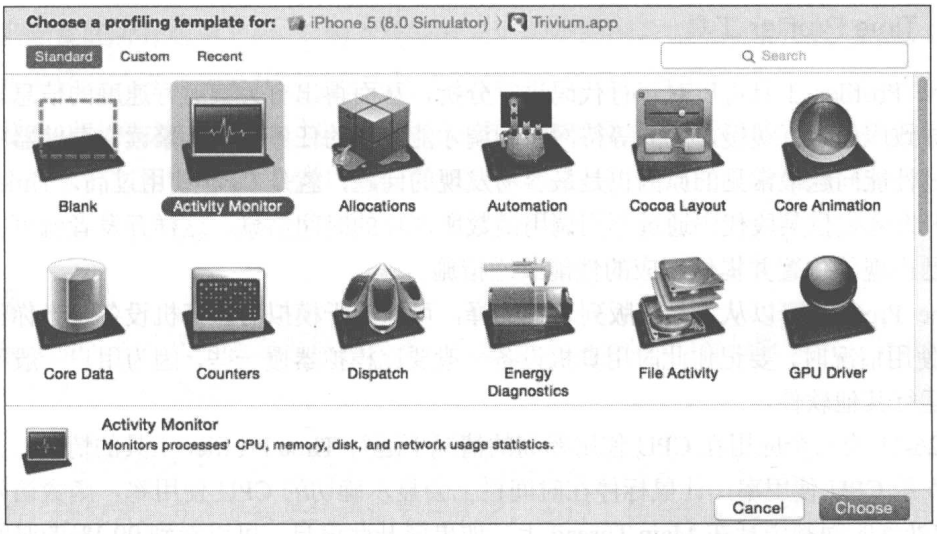


图 26-9　在配置模式下运行应用之后选择调试工具

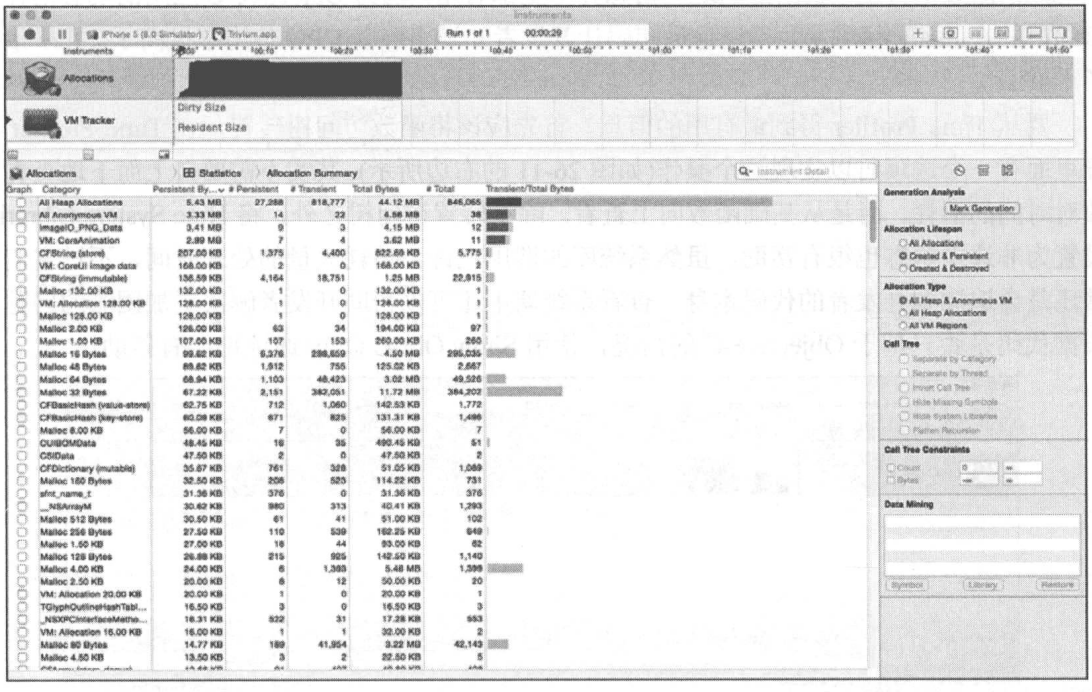


图 26-10　基础工具界面，显示正在运行 Allocations 工具

中央视图和右侧视图中的大部分基础对象都可以通过双击显示附加信息，比如代码的引用分节信息。

在下面的小节中，会介绍两个最常用的工具。第一个是 TimeProfiler，它帮助开发者确定应用中哪些代码的执行耗时最久。通过对每行代码执行时间的分析，开发者可以对代码进行优化和改进，增加整个应用的运行速度。第二个主要的工具集包含 Leaks and Allocation 工具。这些工具可以让开发者分析应用中内存的使用情况，尽早发现内存泄漏和释放错误。

26.4.2 Time Profiler 工具

Time Profiler 工具可以对每行代码进行分析，从而得出有关其运行速度的信息。有很多瓶颈会导致应用运行缓慢，比如等待网络回调才能停止的任务或者频繁读取存储器中的内容等。不过性能问题最常见的原因也是最容易发现的问题，就是 CPU 使用过高。Time Profiler 可以为开发者提供某段代码通过不同调用函数所消耗的时间信息，这样开发者就可以准确定位到问题出现的位置并提供相应的性能改进措施。

Time Profiler 可以从工具模版列表中选择，可以用于模拟器和真机设备。当你正在分析 CPU 的使用情况时，要记住此时用真机设备一般要比模拟器慢一些，因为用户一般不会在模拟器上运行其他软件。

图 26-11 为一个应用在 CPU 使用率高的情况下运行 Time Profiler 工具的情况。上面的紫色部分表示 CPU 使用率，让鼠标停在时间栏上会显示确切的 CPU 使用率。函数访问树显示 99.6%的处理时间都消耗在 Main Thread 上，如果展开此信息，可以看到 99.1%的时间都消耗在 main()函数本身上。表面上的这些信息并不全部有用，因为 Objective-C 应用的确会消耗大量的时间在 main()函数上，不过这最起码让开发者知道了当前 CPU 使用率过高了，几乎接近了 100%。

要从 Time Profiler 得到最有用的信息，首先应该将函数访问树转置，在 Time Profiler 设置里面有一个选项可以实现这个操作(如图 26-11 的右边所示)。其实不需要自上而下地查看函数对时间的消耗，而是从基础函数向上查看。除了转置访问树之外，将 Hide System Libraries 设置为非选中状态也很有帮助。虽然系统库的调用也许会消耗大量的处理时间，不过它们通常还是会追溯到开发者的代码本身。查看系统调用还可以帮助开发者解决更加困难的问题。根据代码是否只基于 Objective-C 的情况，使用 Show Obj-C Only 选项也很有帮助。

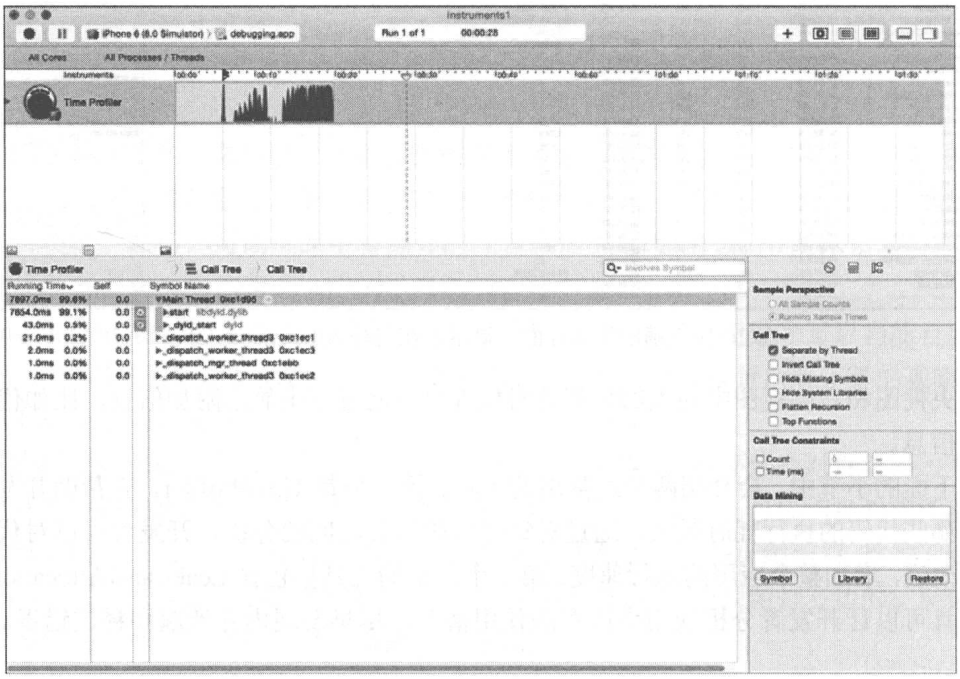


图 26-11 在 CPU 使用率高的情况下对应用运行 Time Profiler 工具

正确配置好所有选项之后，就剩下获取开发者指定的函数调用列表及其消耗的 CPU 时间了。实践中最好的方法就是从耗时最大的函数到最小的函数逐步进行优化，这是因为通常解决大问题的同时可以顺带解决一些小的问题。要获取有关问题代码的更多信息，双击具体的元素将会显示对应的调用树。可以看到代码查看器按行将代码分开，并带有表示相关方法处理时间的标注，如图 26-12 所示。

注意

不能使用工具的代码查看器编辑代码，不过单击 Xcode 图标(如图 26-12 所示)，可以在 Xcode 中打开相应的代码。

虽然 Time Profiler 不能对导致运行缓慢的代码给出优化建议，但它可以为开发者指出问题出现的正确位置。并不是每处代码都要被优化，不过工具会给出问题位置的行号和运行所需的具体时间开销，这样优化的难度也会大大降低。

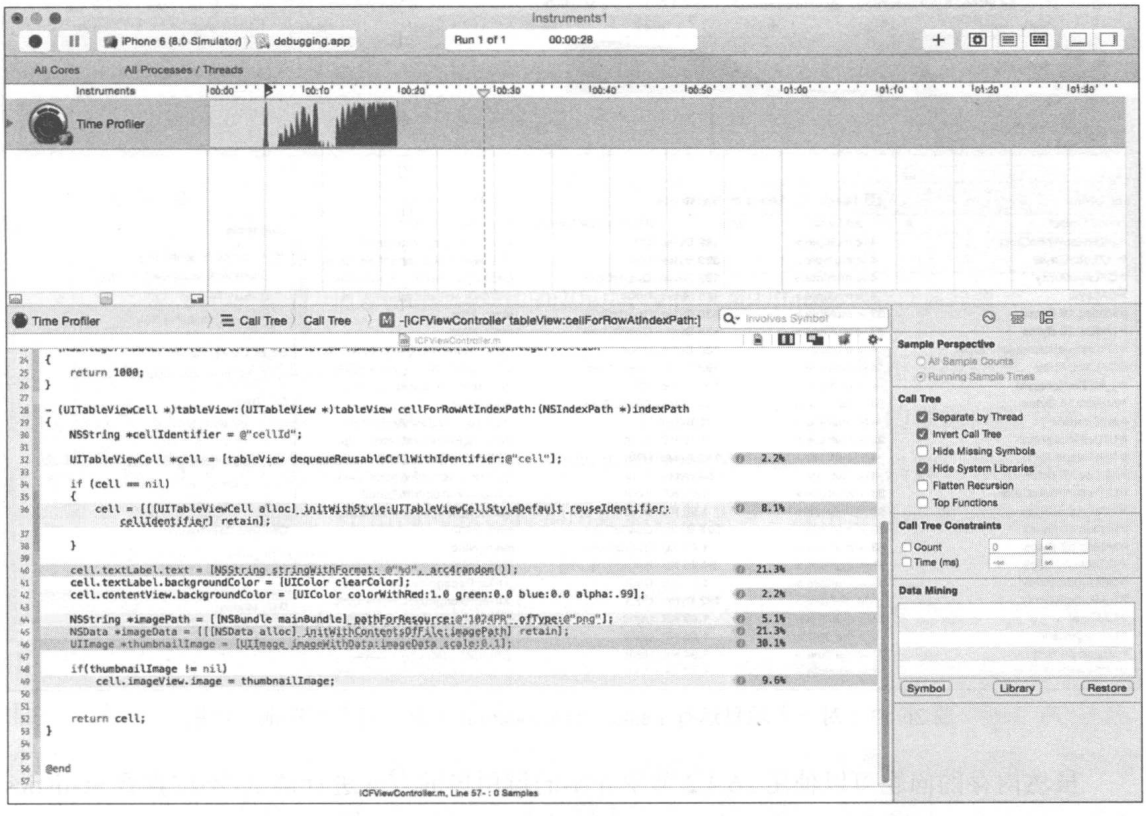


图 26-12　逐行查看 Time Profiler 信息

提示

使用工具中的查看器范围设置对于精确定位或查看具体时间片段有很大的帮助。控件用于标志时间轴的起始位置和查看范围的终点。

26.4.3 Leaks 工具

Leaks 工具和与它类似的 Allocations 工具可以让开发者很好地解决有关内存的问题。可以帮助开发者找到内存重用、泄漏、计数周期和其他内存相关的问题。随着 Automatic Reference Counting (ARC)机制越来越受欢迎，Leaks 和 Allocation 工具渐渐褪去了原有的光环，不过对于开发者来说它们还是能提供非常大的帮助。此外，当 ARC 机制不适合处理内存问题时，一般首先选择这两个工具查找问题。

可以从 Instrument Selector 窗口启动 Leaks 工具，同启动 Time Profiler 的方式一样。Leaks 的启动会自动包含 Allocations 工具，这两个工具都可以用于模拟器和真机设备。在图 26-13 中可以看到一个开发者正在对一个性能拙劣的应用进行分析，程序内存不断被占用，由 Allocations 部分持续增加的图形表示。此外，还检查到一些内存泄漏问题，由 Leaks 部分的红色竖条表示。程序长时间运行之后，这些问题就会导致应用内存耗尽而崩溃。

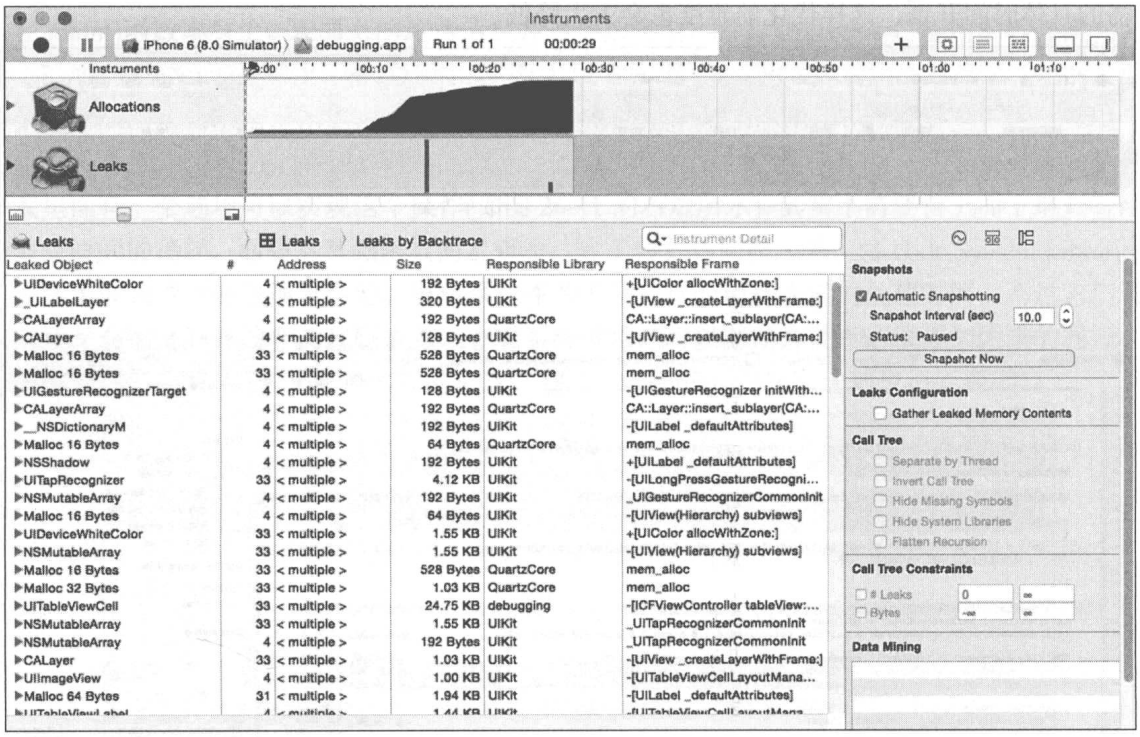


图 26-13 对一个项目运行 Leaks 和 Allocation 工具，查看内存泄漏问题

虽然内存的问题可以使用 26.4.2 节中介绍的调用树的方法进行调试，不过查看 Statistics 或 Leaks 显示的信息有时更加有效。要查看内存泄漏问题，通常是由于内存使用增加，从左上角选择 Leaks 工具。在图 26-14 所示的项目中，UIImage 对象发生了大量内存泄漏问题。

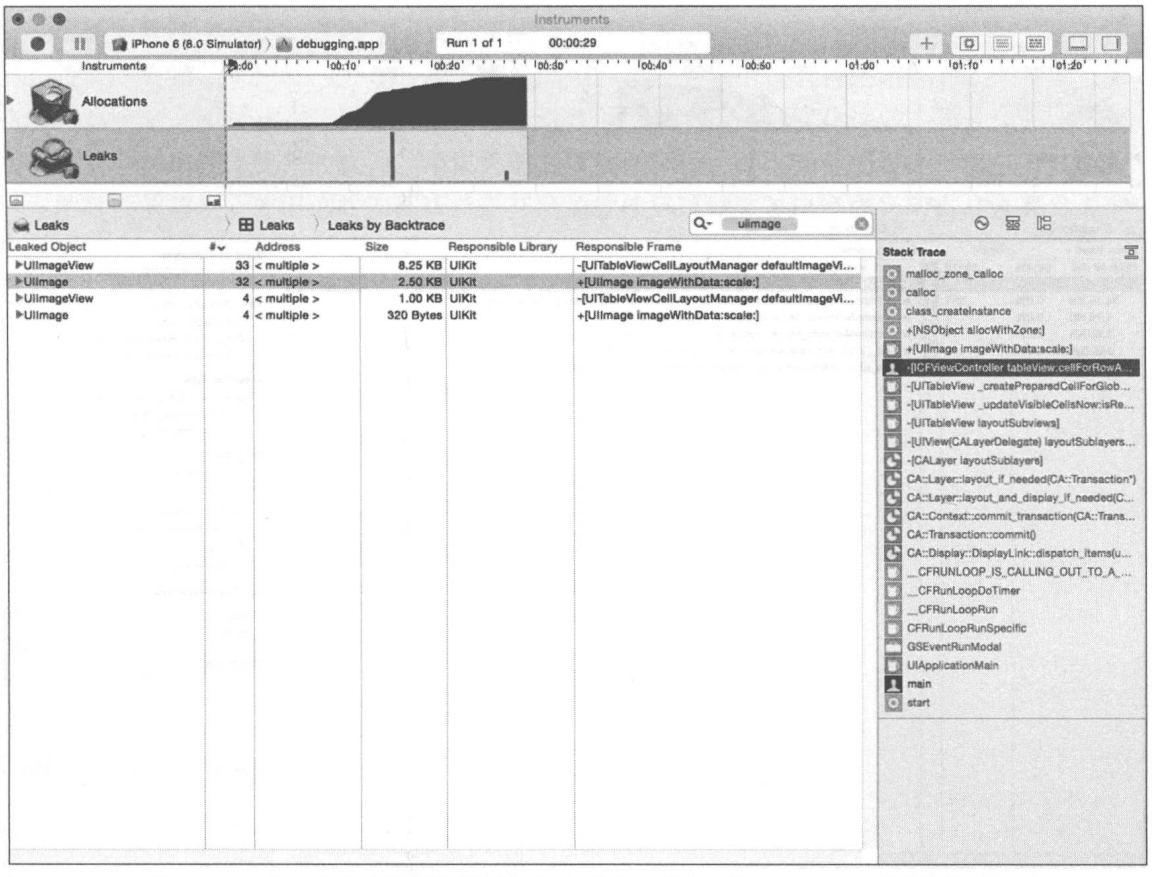


图 26-14　在一个内存使用持续增加的应用中，研究大量 UIImage 内存泄漏问题

工具会尝试将泄漏问题组成可以识别的栈跟踪，不过这个系统可能不是很完美，因为同样的内存泄漏问题在列表中可能不只出现一次。首先解决出现次数最多的泄漏问题并再次运行分析工具。要找到内存泄漏发生的代码，需要展开左边的视图。在工具窗口的标题栏上选择视图控制器即可。选择一个泄漏问题会转到出现问题的事件。双击非系统调用的函数(通常用黑色文本显示)将会转到内存泄漏出现的代码位置。

有时候应用内存的使用持续增加到无法接受的水平，不过还没有出现泄漏。这是因为应用使用了超出自己需要的内存。要定位到此信息，选择 Allocations 工具并查看调用树。用上面同样的方法转置调用树，隐藏系统库函数，在 Time Profiler 部分显示只针对 Obj-C 的内容。在图 26-15 中，cellForRowAtIndexPath:占用了 19.30MB 的内存，导致程序运行缓慢。双击该对象会打开代码查看器，精确定位到消耗内存最多的那一行代码，该区域就是导致内存泄漏的主要程序段。

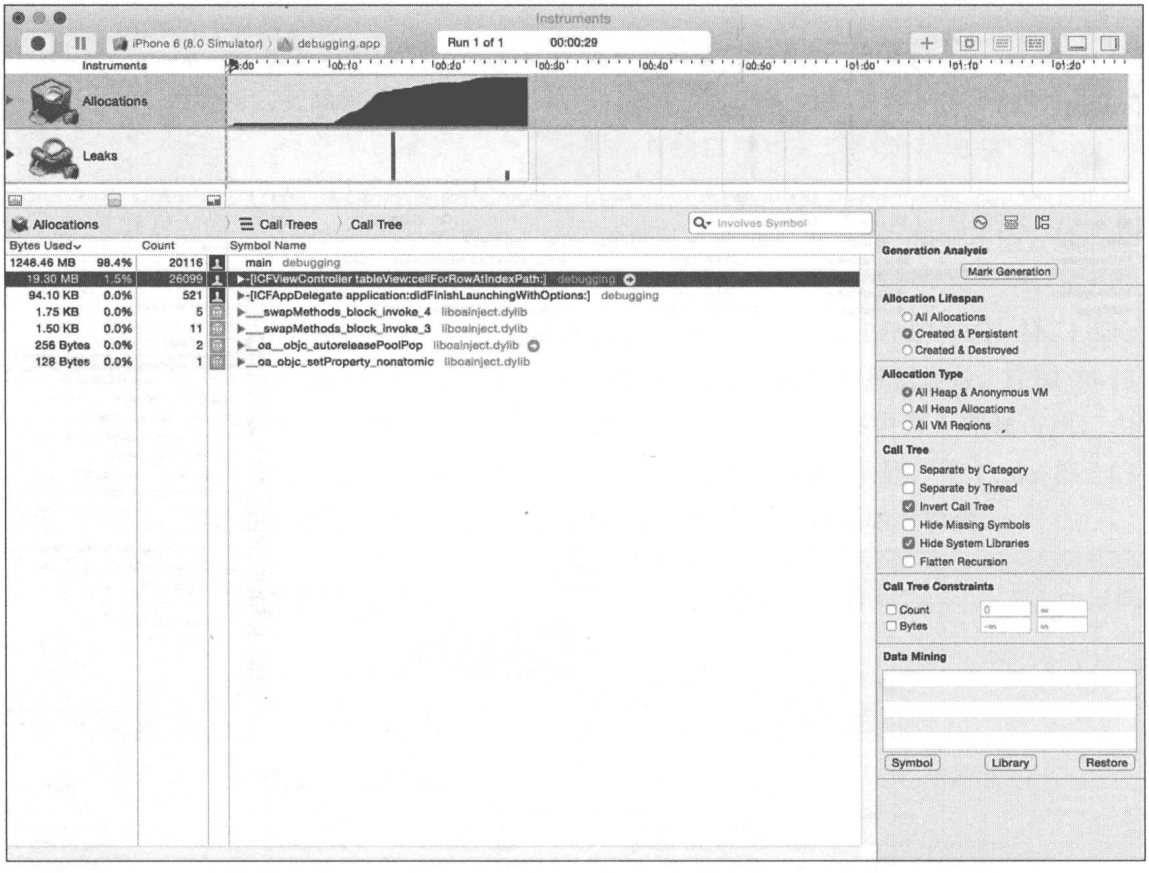


图 26-15 在 Allocations 的调用树中研究 cellForRowAtIndexPath:使用大量内存的情况

26.4.4 进一步了解调试工具

Xcode 的调试工具非常值得研究。开发者了解了基本功能和控件之后，可以推导出大量的工具。苹果公司在不断地为开发者提升和改进调试工具。基于这个出发点，对于应用能够做的任何事都有工具进行调试，从 Core Data 到电池计划，甚至有工具可以帮助你优化 Core Animation 和 OpenGL ES 之间的动画。要学习某一特定工具的详细内容，可以访问苹果公司的在线文档，网址为 http://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Introduction/Introduction.html。

26.5 小结

本章和本书中的其他章节不太一样，并没有介绍示例程序，也没有演示新框架的使用，而是介绍了一些更有价值的知识，即有关程序调试和代码优化的内容。代码调试是一个很大的话题，可能需要几本书才能介绍完。我们希望本章介绍的内容作为你不断在此领域里探索并击败一个又一个程序错误的起点。一个能够快速高效地解决问题、优化程序、调试错误的开发者从来不愁好的工作，也一定是团队中最有价值的一员。

调试工具和 Xcode IDE 都是苹果公司给开发者最好的礼物。也就是前几年，购买 IDE 还要花费几千美元，而且那时的 IDE 也不是很好用，类似的调试工具也还不存在。当苹果公司将 Xcode 免费之后，带来了一场新的革命。这几年，他们又不断优化这些工具，让开发者使用他们的平台来创建优秀的软件。苹果公司之所以这样做，是因为他们很在乎第三方开发者开发的软件的质量。使用这些工具让自己开发的软件尽可能完美已经成为所有 iOS 开发者的义务。