

- 世界知名iOS开发专家详细讲解创建优秀iOS移动应用程序的技术细节，系统阐释如何高效开发iOS程序
- 用直观的代码演示当前流行的编程技巧，为iOS开发者提供行之有效的解决方案

# iOS核心 开发手册

(原书第5版)

[美] Erica Sadun Rich Wardwell 著 爱飞翔 译



The  
Core iOS  
Developer's Cookbook  
Fifth Edition



机械工业出版社  
China Machine Press

# iOS核心 开发手册

The Core iOS  
Developer's Cookbook  
Fifth Edition

(原书第5版)

[美] Erica Sadun Rich Wardwell 著 爱飞翔 译



机械工业出版社  
China Machine Press



## 图书在版编目 (CIP) 数据

iOS 核心开发手册 (原书第 5 版)/(美) 萨顿 (Sadun, E.), (美) 沃德韦尔 (Wardwell, R.) 著; 爱飞翔译. —北京: 机械工业出版社, 2015.3

(iOS/ 苹果技术丛书)

书名原文: The Core iOS Developer's Cookbook, Fifth Edition

ISBN 978-7-111-49185-9

I. i… II. ①萨… ②沃… ③爱… III. 移动终端—应用程序—程序设计—手册 IV. TN929.53-62

中国版本图书馆 CIP 数据核字 (2015) 第 015570 号

本书版权登记号: 图字: 01-2014-4435

Authorized translation from the English language edition, entitled *The Core iOS Developer's Cookbook, Fifth Edition* 9780321948106 by Erica Sadun, Rich Wardwell, published by Pearson Education, Inc., Copyright © 2014.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2015.

本书中文简体字版由 Pearson Education (培生教育出版集团) 授权机械工业出版社在中华人民共和国境内 (不包括中国台湾地区和香港、澳门特别行政区) 独家出版发行。未经出版者书面许可, 不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education (培生教育出版集团) 激光防伪标签, 无标签者不得销售。

## iOS 核心开发手册 (原书第 5 版)

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号) 邮政编码: 100037

责任编辑: 关 敏

责任校对: 董纪丽

印 刷: 北京市荣盛彩色印刷有限公司

版 次: 2015 年 3 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 35.25

书 号: ISBN 978-7-111-49185-9

定 价: 119.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光/邹晓东

本书是一本很受欢迎的技术书，从第 1 版到第 5 版，虽然书名和编排形式略有调整，但是 Cookbook 这个特点（本书英文版书名为《The Core iOS Developer's Cookbook》）却一直在延续。

这种食谱式的秘诀手册之所以受欢迎，原因之一就在于开发者非常容易通过范例来学习一门语言或一套开发环境的用法。有了生动易懂的范例，我们很快就能明白如何制作出简单实用的程序。再经过反复练习，即可逐渐掌握开发流程。

另外，秘诀手册与普通的范例教程还有个非常重要的区别，那就是范例的选择。一本优秀的秘诀手册不仅要涵盖中级和高级范例，而且还要选择有代表性的范例，使读者在学会该范例之后，能够根据实际需要对其加以修改，以便将它快速整合到自己的程序中。

从上述两方面来看，本书都极好地体现了 Cookbook 的特点。作者根据长年积累的开发经验，将一百多条解决方案划分为 iOS 开发中的十几个专题，基本上满足了日常编程的需要。有些方案讲得尤为细致，使读者可以看到如何按照需求来深度定制各种组件，比如第 6 章的文本编辑器和第 7 章的自制容器等。除了冠以解决方案之名的范例项目外，还有一些以程序清单形式出现的代码片段也比较实用，每段代码都相当于一份小巧的模板，演示了某些对象或技术的配置方式。

综观这十几个专题，很多都与图形界面及手势有关，这对于提升程序流畅度和用户体验大有好处。作者还提供了范例项目的源代码。对于较为复杂的项目来说，我们可以先运行范例程序，看到实际效果，然后再与书中各步骤详细比对，以了解整个流程；也可以自己先尝试一种实现方式，然后与作者给出的方案或其他教程里给出的方案相比较，看看有何异同，并讨论各自的优缺点。

虽说本书所举的范例都较为浅显直观，但读者还是需要略微懂一些 Objective-C 语言和 iOS 开发的基础知识。如果你是新手，不妨先参考一下作者在前言里所推荐的几本入门书籍。此外，如果对后面的 Core Data、网络编程、设备适配及辅助功能等内容感兴趣，可以参考与这些主题有关的专著，以便深入研究。

在翻译过程中，得到机械工业出版社华章公司诸位编辑与工作人员的帮助，在此深表谢意，并感谢 goldlion 及 ChenGe 两位友人对术语翻译所提的建议。

由于译者水平有限，错误与疏漏之处在所难免，恳请各位读者批评指正。可发邮件至 [eastarstormlee@gmail.com](mailto:eastarstormlee@gmail.com) 与我联系，也可访问网页 <https://github.com/jeffreybaoshenlee/zh-translation-errata-cidc5/issues> 留言。

爱飞翔

欢迎阅读本书新版!

自苹果公司发行 iOS 移动操作系统以来, iOS 7 的变革是最为重大的。这本教程将会指导各位开发者针对这个新发布的优秀操作系统来制作应用程序。2013 年的全球软件开发年会 (Worldwide Developers Conference, WWDC) 公布了一些新的特性和视觉范式, 而本书这次修订已经将它们全都涵盖在内了, 笔者将会向你演示如何将其融入自己的应用程序里。

发行团队将这次修订过的 Cookbook 材料分成两本书来印刷, 以控制每本书的篇幅。本书的英文书名叫作《*The Core iOS Developer's Cookbook*》, 讲述了日常开发所需的关键知识, 介绍了使用标准 API 与界面元件来创建 iOS 应用程序时所需用到的类。同时, 本书以“解决方案”的形式讲解创建移动应用程序时所需的图像、触摸及视图等技术。

另外一本书的英文名叫作《*Learning iOS Development: A Hands-on Guide to the Fundamentals of iOS Programming*》, 其中包含了一些入门知识, 相当于老版本 Cookbook 的前面几章。该书适合想要从头学习 iOS 7 基础知识的开发者阅读。《*Learning iOS Development*》讲述了 Objective-C 编程语言、Xcode 开发环境以及与调试和部署有关的内容, 你可以通过它学会如何使用苹果公司的开发工具套件。

## 学习本书所需的材料和知识

想开发 iOS 应用程序, 肯定得有一台测试应用程序用的 iOS 设备, 而且最好是一台新款的 iPhone 或 iPad。下面列出阅读本书所需的材料和基础知识:

- 苹果公司的 iOS SDK——你可以从苹果公司的 iOS Dev Center (<http://developer.apple.com/ios>) 下载最新版的 iOS SDK。如果打算在 App Store 上发售应用程序, 那么还必须成为付费的 iOS 开发者。个人开发者每年需要付款 99 美元, 企业开发者每年需要付款 299 美元。注册成为付费开发者之后, 就会收到一份证书, 开发者可以用这份证书来签署应用程序, 并将其下载到 iPhone、iPod touch 或 iPad 中进行测试与调试, 此外, 付费开发者还可以提前获得预览版的 iOS 系统。未付费的开发者可以在 Mac 系统的模

拟器上测试软件，但是不能将其部署到设备中，也不能将其提交到 App Store。

- **运行 Mac OS X Mountain Lion (v 10.8) 系统的新款 Mac**，如果装的是 Mac OS X Mavericks (v 10.9) 系统就更好——你需要为软件开发留出足够的磁盘空间，而且应该尽可能把 Mac 的 RAM 装配得大一些。
- **iOS 设备**——尽管 iOS SDK 里的模拟器也能用来测试应用程序，但是开发者仍然需要一台 iOS 硬件设备，以便针对 iOS 平台做开发。你可以把 iOS 设备与电脑相连，并把自己构建的软件装上去。在开发真实的 App Store 程序时，应该准备数款硬件和固件各不相同的设备，以便在目标用户可能会用到的各种平台上进行测试。
- **因特网连接**——连上网之后，就可以测试应用程序在使用 WiFi 和使用移动数据网络时的效果了。
- **熟悉 Objective-C**——要想编写 iPhone 程序，需要了解 Objective-C 2.0。这是一门基于 ANSI C 的语言，并且带有面向对象扩展，也就是说，你需要了解一点 C 语言的知识。如果原来用 Java 或 C++ 写过程序，而且又熟悉 C 语言，那么应该能够迅速适应 Objective-C。

## 学习 Mac/iOS 开发的路线图

一本书不可能把各类读者所需的全部知识都囊括其中。假如两位作者把你所需的全部内容都写到这本书里，那它会重到根本拿不起来。实际上，想要开发 Mac 及 iOS 平台上的程序，需要学习很多内容。如果你刚处于起步阶段，而且没有写过程序，那么首先应该学习一门大学水平的 C 语言课程。大部分编程语言都以 C 语言为根基，对于想要成为开发者的人来说，自然也要从 C 语言学起。

学会 C 语言及编译器（基础的 C 语言课程会讲到它）的用法之后，剩下的事情就简单多了。此时可以直接跳到 Objective-C 语言，学习如何用它来编程，同时还可以学习 Cocoa 框架。图 1 是一张流程图，里面列出了培生教育出版集团所出版的一些关键书籍<sup>①</sup>，它们可以帮助你成为一名熟练的 iOS 开发者。

了解 C 语言之后，就可以通过多种方式来学习 Objective-C 编程了。如果想深入了解 Objective-C，那么可以阅读苹果公司自编的文档，也可以翻看下列 Objective-C 教材：

- 《*Objective-C Programming: The Big Nerd Ranch Guide*》，第 2 版，Aaron Hillegass 与 Mikey Ward 著（Big Nerd Ranch，2013 年）
- 《*Learning Objective-C 2.0: A Hands-on Guide to Objective-C for Mac and iOS Developers*》，第 2 版，Robert Clair 著（Addison-Wesley，2012 年）<sup>②</sup>

① 《The C Programming Language (Second Edition)》一书由 Brian W. Kernighan 及 Dennis M. Ritchie 著，中文版叫作《C 程序设计语言》，由机械工业出版社于 2004 年出版，影印版由机械工业出版社于 2006 年出版。《Xcode 5 Start To Finish》一书由 Fritz Anderson 所著，由 Addison-Wesley Professional 出版社于 2014 年出版。——译者注

② 该书中文版名为《Objective-C 2.0 Mac 和 iOS 开发实践指南》，李强等译，机械工业出版社 2012 年出版。——译者注

■ 《Programming in Objective-C 2.0》，第6版，Stephen Kochan 著（Addison-Wesley，2012年）<sup>①</sup>

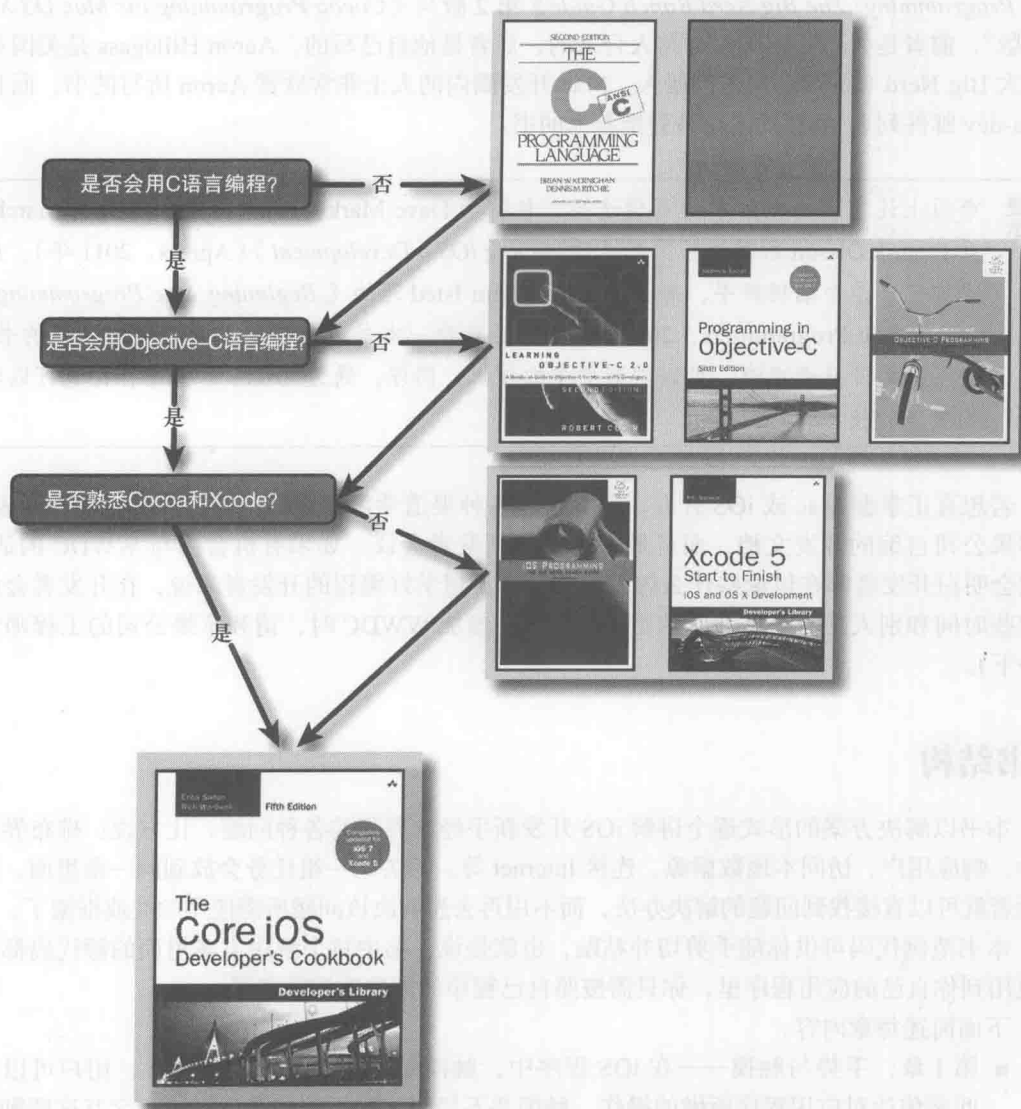


图1 成为 iOS 开发者的路线图

学会编程语言之后，接下来应该学习 Cocoa（适用于 Mac 开发）、Cocoa Touch（适用于 iOS 开发）以及开发工具，这个开发工具指的就是 Xcode。这一阶段也有几种不同的学习途

① 该书中文版名为《Objective-C 2.0 程序设计（原书第2版）》，张波，黄湘琴等译，机械工业出版社2009年出版。——译者注

径。你依然可以在苹果 Developer 查阅苹果公司自编的 Cocoa、Cocoa Touch 及 Xcode 文档 (网址是: [developer.apple.com](http://developer.apple.com)), 也可以通过阅读书籍来学习它们。笔者推荐两本经典的教材: 《iOS Programming: The Big Nerd Ranch Guide》第 2 版与《Cocoa Programming for Mac OS X》第 4 版<sup>①</sup>, 前者是 Aaron Hillegass 与人合著的, 后者是他自己写的, Aaron Hillegass 是美国亚特兰大 Big Nerd Ranch 公司的创始人。Mac 开发圈内的人士非常欣赏 Aaron 所写的书, 而且 cocoa-dev 邮件列表<sup>②</sup>里面的人也最爱推荐他的书。



**提示** 市面上还有非常多的书籍可供选择, 包括由 Dave Mark、Jack Nutting、Jeff LaMarche 及 Fredrik Olsson 所著的畅销书《Beginning iOS 6 Development》(Apress, 2011 年)。如果你完全是个编程新手, 那么可以看看 Tim Isted 写的《Beginning Mac Programming》(Pragmatic Programmers, 2011 年)。不要只看一本书, 也不要只看一家出版社的书。和不同的开发者交流, 可以学到更多的知识, 同理, 通过阅读多本书籍, 你也可以学到更多的技巧和窍门。

若想真正掌握 Mac 或 iOS 开发, 还需通过各种渠道学习: 看书、看博客、看邮件列表、看苹果公司自编的开发文档, 而且最好能参加开发者会议。如果有机会参与 WWDC 的话, 你就会明白开发者都在讨论些什么内容。对于真正想学好编程的开发者来说, 在开发者会议上花些时间和别人交流是件非常值得的事情 (在参加 WWDC 时, 请和苹果公司的工程师交流一下)。

## 本书结构

本书以解决方案的形式逐个讲解 iOS 开发新手经常遇到的各种问题, 比方说: 排布界面元件、响应用户、访问本地数据源、连接 Internet 等。相关的一组任务会放到同一章里面, 这样读者就可以直接找到问题的解决办法, 而不用再去想解决该问题所需使用的类或框架了。

本书范例代码可供你随手剪切并粘贴, 也就是说, 书中每个解决方案里面的源代码都可以复用到你自己的应用程序里, 你只需按照自己程序的需求来调整即可。

下面简述每章内容。

- 第 1 章, 手势与触摸——在 iOS 程序中, 触摸是一种非常重要的手段, 用户可以由此来传达对应用程序所做的操作。触摸并不局限于按下按钮及通过键盘交互这两种行为。本章将介绍直接操纵界面、多点触摸以及其他一些内容。你将会学到如何创建这样一种视图: 用户可以在屏幕上试验各种手势, 并看到不同手势之间的区别。另外, 本章还会告诉你如何创建自定义的手势识别器。

① 该书中文版名为《苹果开发之 Cocoa 编程 (原书第 4 版)》, 黄敏, 郝刚等译, 机械工业出版社 2012 年出版。——译者注

② 可访问 <https://lists.apple.com/archives/cocoa-dev/> 查阅其内容。——译者注

- 第2章，构建并使用控件——本章将深入讲解如何操控应用程序。你将会详细了解控件的运作机制，还能学会以多种方式来构建并自定义控件。这一章包含很多解决方案，有的比较简单，有的比较复杂，你可以把它们复用到自己的程序里。
- 第3章，提醒用户——iOS 提供了多种在屏幕上向用户显示信息的方式，比如弹出式对话框、进度条、本机通知（local notification）、popover 和 audio ping 等。本章将会讲解如何在应用程序中实现这些信息通知手段，以帮助读者用更多的方式向用户显示信息。本章将介绍这些类的基本使用方法，另外还会提供一些解决方案，使你可以通过基于块的 API（blocks-based API）来轻松地处理与警示信息有关的交互操作。
- 第4章，编排视图及其动画效果——UIView 类及其子类可用来填充 iOS 设备的屏幕。本章将会从头开始讲解视图。与视图有关的解决方案会分别演示如何获取视图对象、如何制作视图的动画效果以及如何操纵视图对象。你将会学到怎样构建、检视及分解视图层级，并了解多个视图是如何组织起来的。通过学习本章，你会发现在图形界面中创建并摆放视图的时候，视图位置的排布是非常重要的，另外，你还会学到如何制作视图在屏幕上移动和切换时所具备的动画效果。
- 第5章，视图的约束系统——Auto Layout 机制彻底改变了 iOS 程序里视图的排布方式。苹果公司的这种布局特性使开发者可以轻松地设计出更为协调一致的界面。此特性对于同一系列不同屏幕大小、不同界面、不同屏幕方向、不同语言的设备来说尤为重要。本章将会介绍如何用代码来做视图约束方面的开发。你会学到怎样在屏幕上的物件之间创建关系以及怎样指定布局规则，使 iOS 能够自动排布应用程序中的视图。看完本章后，你就能设定一套健全的屏幕布局规则了。
- 第6章，文本输入——本章的解决方案都与文本有关，这些解决方案能够解决许多问题。你会学到如何控制键盘、如何使屏幕上面的控件支持文本输入、如何扫描文本、如何格式化文本，等等。这一章会把与 iOS 程序文本处理有关的各项技术都涵盖在内，包括文本框、文本视图以及 iOS 内置的拼写检查器。
- 第7章，使用视图控制器——本章将会讲解各种视图控制器类的用法，这些类使得用户可以在更大的范围中与应用程序交互，而开发者也可以借此来排布视图。你将通过本章的各解决方案学到页面视图控制器、分栏视图控制器、导航控制器等视图控制器的用法。
- 第8章，常用的控制器——iOS SDK 里面有很多系统自带的控制器，开发者可以用它们来完成日常的开发任务。本章将介绍最为常用的控制器。你会学到如何从照片库中选取照片、如何拍照、如何录制并编辑视频。另外，你还会学到如何在程序中编写电子邮件及文本消息，以及如何在 Twitter 及 Facebook 等社交媒体上张贴信息。
- 第9章，创建并管理表格视图——表格（table）是一种可以滚动的交互类，它在屏幕较小的设备上面效果很好，在屏幕较大的平板电脑上面效果也很不错。由于表格可以把内容以一种简单而自然的方式组织起来，所以很多 iOS 应用程序都是以表格为中



心的。本章将介绍表格的用法，解释表格的工作原理，讲解可供开发者使用的各种表格，并且告诉你如何在程序中利用表格的各种特性。

- 第 10 章，集合视图——集合（collection）视图的许多概念都与表格相同，但是功能更加强大大，而且更加灵活。本章将会讲述使用集合视图所需的各种基础知识，包括如何创建可以横向滚动的列表、如何创建网格布局、如何创建圆形等特殊方式的布局，等等。你将学到怎样通过布局规格（layout specification）把视觉效果集成到集合视图里面，以及怎样使集合视图中的内容在滚动之后自动调整位置，另外，你还会学到如何利用内置的动画支持来创建出最有效的互动效果。
- 第 11 章，分享文档与数据——在 iOS 系统中，应用程序可以分享信息和数据，另外，开发者也可以使用系统所提供的许多特性，把控制权从一个程序转移到另一个程序。你可以用本章所介绍的几种方式在应用程序之间分享文档及数据。你将学到如何把这些特性添加到自己的应用程序之中，以及如何灵巧地使用分享功能，令自己的应用程序可以和 iOS 生态系统里的其他程序协同运作。
- 第 12 章，浅谈 Core Data——Core Data 提供了一套受托管的数据存储机制，使开发者可以在应用程序中查询并更新存储区里的数据。它提供了一套基于 Cocoa Touch 的对象接口，使得 iOS 开发者能够像使用 SQL 查询语句那样，通过 Objective-C 代码来管理关系型数据库中的数据。本章介绍 Core Data。通过其中的各解决方案，你可以初步了解这项技术，同时还能以本章内容为出发点，继续深入学习 Core Data。你将学到如何设计受托管的数据库存储机制、如何添加和删除数据、如何用代码查询数据，以及如何把这些操作同 UIKit 中的表格视图及集合视图相集成。
- 第 13 章，网络编程基础——在连接到 Internet 的设备上面，特别适合用 iOS 程序来订阅基于网络的服务。苹果公司为 iOS 平台提供了各种坚实的网络计算服务及支持技术。本章将介绍网络编程中的常用技术，同时也提供一些解决方案，用来简化日常的网络开发任务。本章介绍 iOS 7 新引入的 HTTP 系统，并且提供实现数据下载功能（包括后台下载）所用的一些范例代码。你还会学到如何判断网络是否可用，以及如何使用 Web 服务，这其中包含了一些范例代码，它们告诉你如何通过 XML 解析及 JSON 序列化来访问一些在线服务。
- 第 14 章，针对特定设备的开发——每台 iOS 设备都有许多属性，有些属性是该设备所独有的，而有些则是许多设备所共有的；有些属性是持续变化的，而有些则保持不变。这些属性包括设备当前的物理方向、型号名称、电池状态以及是否可以访问机体内的硬件等。本章将会讲解如何查看设备的硬件规格，以及如何查看设备中可供使用的感应器。这一章所提供的解决方案可用来查询当前设备的各项信息。
- 第 15 章，辅助功能——本章简单地介绍 VoiceOver 这项辅助功能，开发者可以通过该功能尽量扩大应用程序的受众。你将学到如何为应用程序添加与辅助功能有关的标签及提示，以及如何在模拟器和 iOS 设备中测试这些特性。
- 附录 A，Objective-C 字面量——本附录介绍了 Objective-C 语言里与数字、数组及字典有关的一些新特性。

## 对范例代码的说明

为了大家学起来方便，本书的范例代码只使用一个 `main.m` 文件。编写 iPhone 或 Cocoa 应用程序时，开发者一般都不会这么做，而且说实在的，也确实不应该这么做，但是，这种做法却非常适合展示一个大的概念。假如一份范例代码分成 5 个、7 个或 9 个文件，就不太容易讲述这个概念了。而将所有代码都写到一个文件里，则有助于专门把这个概念说清楚。

书中的范例代码不应当当成独立的应用程序来用。每份范例代码只对应于一个解决方案，而且只演示一个概念。每个 `main.m` 文件都是专门为了实现某个中心概念而编写的。读者在学会这些思路之后，可以按照平常开发时所用的文件组织结构及布局方式将其转换为普通的应用程序结构。本书所用的代码组织方式与日常开发中所应提倡的标准组织方式并不相符。笔者之所以采用这种方式，是为了提供精确的解决方案，而大家可以根据需求把它们集成到自己的工作项目中。

苹果公司的标准范例代码与本书不同，你必须查看很多源文件，才能在脑中构建出一套与待演示的概念有关的“思维模型”。那些范例代码都是完整的应用程序，里面通常会涉及一些与当前所要解决的问题没有关系的任务。我们必须花很大精力才能找到与当前问题有关的代码，这是得不偿失的。

本书还有些范例代码没有遵循“一个文件只说一件事”的规则：如果某个解决方案与类的实现有关，那么本书还会提供标准的类文件及头文件。有些解决方案并不是为了强调某项技术，而是为了提供某些类及 `category`（`category` 是一种针对现有类所做的扩展，它不产生新的类）的实现。对于这些解决方案来说，读者可以找到单独的 `.m` 与 `.h` 文件，而 `main.m` 文件里面则封装了一份框架代码，用来描述其余的事情。

本书大多数范例代码都只使用一个应用程序标识符，也就是 `com.sadun.helloworld`。只使用一个标识符的好处是：你的 iOS 设备里不会装很多范例程序。每安装一个范例程序，都会把前面那个替换掉，这样的话，设备的主屏幕就能干净一些。如果需要同时安装多个范例程序，那么只需给标识符加个后缀就可以了，例如 `com.sadun.helloworld.table-edits`。如果想令多个应用程序所显示出来的名称各不相同，那么可以编辑自定义的显示名称。你的 Team Provisioning Profile 能够匹配包括 `com.sadun.helloworld` 在内的每一个应用程序标识符。这样的话，无须修改标识符，就可以把编译后的代码安装到设备上面了，只是记得要在每个项目的 Build Settings 中更新 Code Signing Identity。

## 获取范例代码

你可以在开源项目托管网站 GitHub 中找到本书源代码：<http://github.com/erica/iOS-7-Cookbook>。每一章的源代码都放在一个文件夹内，其中包含书里的相关范例材料。解决方案的编号与其在书中的编号相同。比方说，第 5 章的第 6 个解决方案放在 C05 文件夹下面的 06 子文件夹中。

以 00 为编号的项目或是编号带有后缀（例如 05b、02c）的项目是为了便于搜索或创建

插图而使用的素材。比方说，第 9 章的 00 – Cell Types 项目是为了创建图 9-2 中的效果而编写的，那张图用来演示系统所提供的各种表格视图单元格样式。一般情况下，笔者会把这些多余的项目删掉。本书初稿的读者请求笔者把它们放在这个版本中。整个代码库里大约能找到六七个这样的范例项目。

## 为本书出力

范例代码绝不是一成不变的，它会随着苹果公司的 SDK 与 Cocoa Touch 库而不断进化。请各位读者一起参与这个过程。你可以提交 bug 修复和修改书中的错误，也可以扩充现有的代码。你可以对 GitHub 代码库做分支 (fork)，自己调整代码，实现一些功能，然后再分享回主代码库里。如果你有新的想法或思路，请告诉我们。我们很乐意将你的宝贵建议加到代码库中，并据此完善本书的下一个版本。

## 获取 git 工具

你可以使用 git 版本控制系统来下载本书源代码。Xcode 5 集成开发环境提供了非常健壮的 git 支持。Xcode 5 工具箱里面也包含了命令行式的 git 工具。此外，还有大量的第三方及商业版 git 工具可供选择。

## 使用 GitHub

GitHub (<http://github.com/>) 是最大的 git 托管网站，有超过 15 万个公开的代码库 (repository)<sup>①</sup>。它可以免费托管公开项目，也可以付费托管私有项目。该网站提供了一套可以自定义的 Web 界面，其中包含 Wiki 托管、问题追踪等功能，是项目开发者之间的一个优秀的社交网络，开发者可以在这里寻找新代码，也可以协同开发既有的程序库。你可以在 GitHub 网站注册免费账号，注册好之后，就可以复制并修改本书的范例代码库了，另外，也可以创建自己的开源 iOS 项目，并与他人分享。

---

① GitHub 的代码库里面还可以存放除代码之外的其他内容。——译者注

## Acknowledgments 致谢

### Erica Sadun

若没有 Chuck Toporek，此书不可能面世，多年以来，是他在为我的书做编辑，并且不停地敦促我写稿。他在多家图书发行商那里工作过，目前就职于 Omnigroup，我非常想念他。没有他的帮助，就不会有这本书。Chuck Toporek 可以玩转两套绝活：一是能够激发作者完成他们自认为无法做到的事情，二是能够挥舞着“务实的大棒”<sup>①</sup>，逼迫作者把书稿内容写得集中而又实用一些<sup>②</sup>。这世上最难的事情，莫过于有人不停地催着你必须赶在截止日期之前把书写完，而且还必须写得很好。

同样要感谢 Trina MacDonald（本书的优秀编辑）、Chris Zahn（卓有才华的策划编辑）以及 Olivia Basegio（兢兢业业的助理编辑，负责处理幕后事务）。此外，非常感谢整个 Addison-Wesley/Pearson 制作团队，尤其要谢谢 Kristy Hart、Betsy Gratner、Kitty Wilson、Anne Goebel、Lisa Stumpf、Gloria Schurick 与 Chuti Prasertsith。同时，感谢 Safari 团队的人员制作本书样稿，并及时修复技术问题。

感谢多年以来的经纪人——Studio B 的 Stacey Czarnowki，感谢最近退休的 Neil Salkind；感谢技术评审 Collin Ruffenach、Mike Shields 及 Ashley Ward，你们 3 位令本书内容更加贴近现实，从而免于空谈；感谢 TUAW、Ars Technica 及 Digital Media/Inside iPhone 博客的诸位同仁以及原来的各位同事。

在整个 iOS 开发者社区中，有很多人曾经帮助了我，在此深表感谢，他们是：Jon Bauer、Tim Burks、Matt Martel、Tim Isted、Joachim Bean、Aaron Basil、Roberto Gamboni、John Muchow、Scott Mikolaitis、Alex Schaefer、Nick Penree、James Cuff、Jay Freeman、

① 原文是 wielding the large “reality trout” of whacking，字面意思为：挥舞着超级大的“现实鲑鱼”。这是一种比喻的说法，大意是告诫某人应该怎样做。——译者注

② 在本书的编辑和制作环节中，我们没有伤害到任何一条所谓的“鲑鱼”，无论是真实的鲑鱼还是意念中的鲑鱼。但是我们却喝掉了无数罐饮料：Erica 喝了很多 Diet Coke（健怡可乐），Rich 喝了很多 Diet Mountain Dew（激浪轻怡），这些听装饮料无私地“奉献”了它们自己，为两位笔者写作本书效劳。

Mark Montecalvo、August Joki、Max Weisel、Optimo、Kevin Brosius、Planetbeing、Pytey、Michael Brennan、Daniel Gard、Michael Jones、Roxfan、MuscleNerd、np101137、UnterPerro、Jonathan Watmough、Youssef Francis、Bryan Henry、William DeMuro、Jeremy Sinclair、Arshad Tayyeb、Jonathan Thompson、Dustin Voss、Daniel Peebles、ChronicProductions、Greg Hartstein、Emanuele Vulcano、Sean Heber、Josh Bleacher Snyder、Eric Chamberlain、Steven Troughton-Smith、Dustin Howett、Dick Applebaum、Kevin Ballard、Hamish Allan、Oliver Drobnik、Rod Strougo、Kevin McAllister、Jay Abbott、Tim Grant Davies、Maurice Sharp、Chris Samuels、Chris Greening、Jonathan Willing、Landon Fuller、Jeremy Tregunna、Wil Macaulay、Stefan Hafenegger、Scott Yelich、chrallielinder、John Varghese、Andrea Fanfani、J. Roman、jtbandes、Artissimo、Aaron Alexander、Christopher Campbell Jensen、Nico Ameghino、Jon Moody、Julián Romero、Scott Lawrence、Evan K. Stone、Kenny Chan Ching-King、Matthias Ringwald、Jeff Tentschert、Marco Fanciulli、Neil Taylor、Sjoerd van Geffen、Absentia、Nownot、Emerson Malca、Matt Brown、Chris Foresman、Aron Trimble、Paul Griffin、Paul Robichaux、Nicolas Haunold、Anatol Ulrich (hypnocode GmbH)、Kristian Glass、Remy “psy” Demarest、Yanik Magnan、ashikase、Shane Zatezalo、Tito Ciuro、Mahipal Raythatha、Carbon Five 的 Jonah Williams、Joshua Weinberg、biappi、Eric Mock。同时感谢 irc.saurik.com 及 irc.freenode.net 的 iPhone 开发者频道 (iPhone developer channel) 里的每位网友。此外还有很多要感谢的人，由于数量众多，恕无法一一列名。有了他们所提供的技术、建议以及反馈，本书才能够得以完成。前述名单若有遗漏，还请帮助过我的诸君见谅。

特别感谢我的家人及朋友，连月以来，本书发行了很多新的 beta 版本，我有时无故缺席聚会，也经常发泄情绪，而他们则一直耐心以对。衷心感谢他们一路伴我写完这本书。尤其要谢谢孩子们对我的理解，我有时没能当个好妈妈，而是在弓着背敲键盘，但他们却依然支持我。过去几个月来，孩子们帮了我的大忙，他们帮我测试程序并提出建议，这真的很棒！每天我都提醒自己：能和他们生活在一起，实在是种福气。

## Rich Wardwell

临近交稿期限的时候，我有很多话要说，在此谨向 Erica 致以最深的敬意，感谢她邀请我参与本书新版的写作，我觉得非常荣幸。在她的督导之下，我学到了很多，至少可以说，我明白了她对书稿质量的严格要求。

感谢我们的编辑 Trina MacDonald，若没有她的坚持，我可能写完第 1 章之后就会抓狂无法坚持下去了。在写作本书的过程中，我时而沮丧、时而焦虑、时而逃避，但 Trina 一直在指导并鼓励我。也要感谢助理编辑 Olivia Basegio，她精心安排并管理了一个技术编辑团队。技术编辑们做了全方位的努力，大大提升了本书质量，如果没有他们，我自己是做不到这一点的，所以，我要对 Collin Ruffenach、Mike Shields 及 Ashley Ward 致以深深的谢

意。感谢本书制作团队及其成员 Betsy Gratner 和 Kitty Wilson，是他们提升了我的写作技能。Addison-Wesley/Pearson 团队中还有很多未曾说过话的朋友，也要感谢他们，感谢令本书得以面世的每个人。

特别感谢 Black Pixel 的 Bil Moorhead、George Dick 及 Daniel Pasco，在我专注写书而忽略了日常的工作时，你们给了我极大的理解。很荣幸能与 Black Pixel 的诸位同仁共事。

感谢我的父母 Rick 及 Janet，他们一直是我人生道路上最伟大的支持者，这次写书当然也是如此。感谢至亲 Steve 及 Cary 给我们提供了一个舒适的写作空间。

最后，感谢爱妻和两个孩子，他们是这个写作项目得以竣工的元勋。我没有陪着他们度过温馨甜蜜的家庭时光，而且也经常不和孩子们玩耍。有了他们的关爱，我才能把这本书写完。

# 目 录 *Preface*

译者序  
前 言  
致 谢

## 第 1 章 手势与触摸 ..... 1

### 1.1 触摸 ..... 1

#### 1.1.1 触摸操作所处的阶段 ..... 2

#### 1.1.2 UIResponder 类中的触摸 事件响应方法 ..... 3

#### 1.1.3 对视图的触摸 ..... 4

#### 1.1.4 多点触摸 ..... 4

#### 1.1.5 手势识别器 ..... 5

### 1.2 解决方案：添加简单的直接操纵 界面 ..... 5

### 1.3 解决方案：添加拖动手势识别器 ..... 7

### 1.4 解决方案：同时使用多个手势 识别器 ..... 9

### 1.5 解决方案：限制移动 ..... 14

### 1.6 解决方案：测试触摸 ..... 15

### 1.7 解决方案：针对位图的触摸测试 ..... 17

### 1.8 解决方案：根据触摸情况在屏幕 上绘制内容 ..... 19

### 1.9 解决方案：令绘制效果变得平滑 ..... 21

### 1.10 解决方案：启用多点触摸 ..... 24

### 1.11 解决方案：检测圆圈手势 ..... 27

### 1.12 解决方案：创建自定义手势 识别器 ..... 32

### 1.13 解决方案：把滚动视图中的内容 拖曳到外面 ..... 34

### 1.14 解决方案：实时的触摸反馈 ..... 37

#### 1.14.1 启用触摸反馈效果 ..... 38

#### 1.14.2 拦截并转发触摸事件 ..... 38

#### 1.14.3 实现 TOUCHkit 的 TOUCHkitView 类 ..... 40

### 1.15 解决方案：向视图添加菜单 ..... 42

### 1.16 小结 ..... 43

## 第 2 章 构建并使用控件 ..... 45

### 2.1 UIControl 类 ..... 45

#### 2.1.1 目标 - 动作模式 ..... 46

#### 2.1.2 控件的种类 ..... 46

#### 2.1.3 控件事件 ..... 46

### 2.2 按钮 ..... 48

### 2.3 Interface Builder 中的按钮 ..... 50

### 2.4 解决方案：构建按钮 ..... 51

#### 2.4.1 多行按钮文本 ..... 54

#### 2.4.2 为按钮添加动画元件 ..... 54

#### 2.4.3 为按钮添加额外状态 ..... 55

### 2.5 解决方案：使按钮以动画效果 来响应用户 ..... 55

2.6 解决方案: 为滑杆控件添加自定义的滑块 .....	57	3.3 解决方案: 将变长参数列表与 UIAlertView 结合起来使用 ..	98
2.6.1 定制 UISlider 控件 .....	57	3.4 展示选项列表 .....	99
2.6.2 添加优化代码 .....	58	3.4.1 滚动菜单 .....	101
2.7 解决方案: 创建可以连续点击两次的分段选择控件 .....	61	3.4.2 在动作表中显示文本 .....	101
2.7.1 实现第二次点击时的反馈效果 ..	62	3.5 将操作进度告知用户并提示其稍等片刻 .....	102
2.7.2 控件及带属性的字符串 .....	63	3.5.1 使用 UIActivity-IndicatorView .....	103
2.8 开关控件与步进控件 .....	64	3.5.2 使用 UIProgressView .....	103
2.9 解决方案: 编写 UIControl 的子类 .....	65	3.6 解决方案: 在屏幕上绘制模态的进度指示器 .....	104
2.9.1 创建控件 .....	68	3.7 解决方案: 自制的模态警告视图 ..	106
2.9.2 追踪触摸事件 .....	68	3.8 解决方案: 基本的 popover .....	110
2.9.3 派发控件事件 .....	69	3.9 解决方案: 本机通知 .....	111
2.10 解决方案: 构建评分所用的 Star Slider 控件 .....	69	3.10 用网络活动指示器提醒用户 .....	113
2.11 解决方案: 构建触摸转盘控件 ..	72	3.11 解决方案: 播放简单的提示音 ..	114
2.12 解决方案: 创建拉曳控件 .....	75	3.11.1 System Sound .....	114
2.12.1 为控件添加提示效果 .....	75	3.11.2 为使用系统框架而引入模块 ..	115
2.12.2 测试触摸 .....	77	3.11.3 震动 .....	115
2.13 解决方案: 构建自定义的锁定控件 .....	80	3.11.4 警示音 .....	116
2.14 解决方案: 图片库查看器 .....	83	3.11.5 延迟 .....	117
2.15 构建工具栏 .....	85	3.11.6 释放系统音 .....	117
2.16 小结 .....	88	3.12 小结 .....	117
第 3 章 提醒用户 .....	89	第 4 章 编排视图及其动画效果 .....	119
3.1 直接向用户弹出警告视图 .....	89	4.1 视图层级 .....	119
3.1.1 构建简单的警告视图 .....	89	4.2 解决方案: 用树状图来描述视图层级 .....	121
3.1.2 设置 UIAlertView 的委托 .....	91	4.3 解决方案: 查询子视图 .....	123
3.1.3 显示 UIAlertView .....	92	4.4 管理子视图 .....	125
3.1.4 各种 UIAlertView .....	92	4.4.1 添加子视图 .....	125
3.2 解决方案: 构建支持块的警告视图 .....	93	4.4.2 重排及删除子视图 .....	125
3.2.1 块简介 .....	93	4.4.3 UIView 的回调方法 .....	125
3.2.2 使用块时避免保留循环 .....	95	4.5 为视图设定标签并查找视图 .....	126
		4.6 解决方案: 通过对象关联机制为视图设定名称 .....	127



4.7 视图的几何特征 .....	129	5.5.1 基本约束规则声明 .....	167
4.7.1 框架 .....	130	5.5.2 用可视化格式字符串声明 约束规则 .....	168
4.7.2 与 CGRect 有关的工具函数 .....	130	5.5.3 变量绑定 .....	169
4.7.3 CGPoint 与 CGSize .....	131	5.6 格式字符串 .....	169
4.7.4 CGAffineTransform .....	132	5.6.1 方向 .....	169
4.7.5 坐标系统 .....	133	5.6.2 连接 .....	171
4.8 解决方案: 操控视图的框架 .....	133	5.7 谓词 .....	173
4.8.1 调整视图的尺寸 .....	134	5.7.1 指标 .....	173
4.8.2 CGRect 与中心点 .....	136	5.7.2 描述两个视图关系的谓词 .....	174
4.8.3 视图的其他几何特征 .....	137	5.7.3 优先级 .....	174
4.9 解决方案: 获取与坐标变换有关 的信息 .....	141	5.8 格式字符串总结 .....	174
4.9.1 获取与变换有关的属性 .....	141	5.9 用格式字符串将视图对齐并灵活 调整其尺寸 .....	176
4.9.2 判断两个视图是否相交 .....	142	5.10 处理约束规则的流程 .....	176
4.10 与显示和交互有关的特征 .....	147	5.11 管理约束规则 .....	177
4.11 UIView 的动画效果 .....	148	5.12 解决方案: 实现约束规则之间 的对比 .....	178
4.12 解决方案: 视图的淡入与淡出 .....	150	5.13 解决方案: 创建尺寸固定且受 规则约束的视图 .....	181
4.13 解决方案: 交换两个视图的 前后顺序 .....	151	5.13.1 禁用 translatesAutoresizingMaskIntoConstraints- MaskIntoConstraints .....	181
4.14 解决方案: 翻转视图 .....	151	5.13.2 令视图出现在上级视图 范围内 .....	182
4.15 解决方案: 采用 Core Animation API 来制作切换效果 .....	153	5.13.3 限定视图的尺寸 .....	183
4.16 解决方案: 使视图在出现之后 回弹 .....	155	5.13.4 把前面各节内容拼装起来 .....	183
4.17 解决方案: 关键帧动画 .....	156	5.14 解决方案: 将两个视图居中 对齐 .....	185
4.18 解决方案: UIImageView 的 动画效果 .....	157	5.15 解决方案: 设定宽高比 .....	186
4.19 小结 .....	158	5.16 解决方案: 响应屏幕方向的 变更 .....	188
<b>第 5 章 视图的约束系统 .....</b>	<b>160</b>	5.17 调试约束规则 .....	190
5.1 什么是约束 .....	161	5.18 解决方案: 描述约束规则 .....	191
5.2 约束系统所用的属性 .....	161	5.19 用宏来创建约束规则 .....	194
5.3 约束系统的运作规律 .....	163	5.20 小结 .....	197
5.4 约束规则与框架属性 .....	165		
5.4.1 固有内容的尺寸 .....	165		
5.4.2 对齐矩形 .....	166		
5.5 创建约束规则 .....	167		

## 第6章 文本输入 ..... 198

- 6.1 解决方案：隐藏 UITextField 的键盘 ..... 199
  - 6.1.1 阻止系统把键盘隐藏起来 ..... 200
  - 6.1.2 UITextInputTraits 协议中的属性 ..... 200
  - 6.1.3 文本框的其他属性 ..... 201
- 6.2 解决方案：把带有自定义辅助视图的键盘隐藏起来 ..... 203
- 6.3 解决方案：根据键盘来调整文本视图 ..... 205
- 6.4 解决方案：创建自定义的输入视图 ..... 209
- 6.5 解决方案：使视图具备文本输入功能 ..... 213
- 6.6 解决方案：为非文本视图添加自定义的输入视图 ..... 216
- 6.7 解决方案：创建更好的文本编辑器（第一部分） ..... 218
- 6.8 解决方案：创建更好的文本编辑器（第二部分） ..... 221
  - 6.8.1 启用 Attributed Text ..... 221
  - 6.8.2 控制文本的样式 ..... 221
  - 6.8.3 可供 UIResponder 使用的其他功能 ..... 223
- 6.9 解决方案：过滤用户所输入的文本 ..... 224
- 6.10 解决方案：检测文本模式 ..... 226
  - 6.10.1 构建自己的正则表达式 ..... 227
  - 6.10.2 枚举正则表达式 ..... 227
  - 6.10.3 数据探测器 ..... 228
  - 6.10.4 使用内置类型的探测器 ..... 229
  - 6.10.5 有用的网站 ..... 229
- 6.11 解决方案：检测 UITextView 中的拼写错误 ..... 231
- 6.12 搜寻文本中的字符串 ..... 232
- 6.13 小结 ..... 233

## 第7章 使用视图控制器 ..... 234

- 7.1 视图控制器 ..... 234
  - 7.1.1 UIViewController 类 ..... 235
  - 7.1.2 导航控制器 ..... 235
  - 7.1.3 标签栏控制器 ..... 235
  - 7.1.4 分栏视图控制器 ..... 236
  - 7.1.5 页面视图控制器 ..... 236
  - 7.1.6 popover 控制器 ..... 236
- 7.2 使用导航控制器与分栏视图控制器来开发程序 ..... 237
  - 7.2.1 使用导航控制器与导航栈 ..... 238
  - 7.2.2 推入与弹出视图控制器 ..... 239
  - 7.2.3 导航栏上的按钮 ..... 239
  - 7.2.4 延伸至屏幕边缘的布局形式 ..... 240
- 7.3 解决方案：UINavigationController 类 ..... 241
  - 7.3.1 标题与后退按钮 ..... 242
  - 7.3.2 宏 ..... 242
- 7.4 解决方案：模态界面 ..... 244
- 7.5 解决方案：构建分栏视图控制器 ..... 248
- 7.6 解决方案：用分栏视图及导航控制器创建通用的程序 ..... 253
- 7.7 解决方案：标签栏 ..... 255
- 7.8 记住标签的状态 ..... 259
- 7.9 解决方案：页面视图控制器 ..... 262
  - 7.9.1 与书籍展示风格有关的属性 ..... 262
  - 7.9.2 封装实现细节 ..... 263
  - 7.9.3 范例代码详解 ..... 269
  - 7.9.4 构建界面索引 ..... 270
- 7.10 解决方案：自定义的容器 ..... 271
  - 7.10.1 添加与移除子视图控制器 ..... 274
  - 7.10.2 视图控制器之间的切换效果 ..... 275
- 7.11 解决方案：segue ..... 276
- 7.12 小结 ..... 282

## 第8章 常用的控制器 ..... 284

- 8.1 图像选取器控制器 ..... 284
  - 8.1.1 图像来源 ..... 284
  - 8.1.2 在 iPhone 和 iPad 中显示选取器 ..... 285
- 8.2 解决方案：选取图像 ..... 286
  - 8.2.1 向模拟器中添加图片 ..... 286
  - 8.2.2 AssetsLibrary 模块 ..... 286
  - 8.2.3 展示选取器 ..... 287
  - 8.2.4 处理 delegate 的回调 ..... 288
- 8.3 解决方案：拍摄照片 ..... 293
  - 8.3.1 配置选取器 ..... 293
  - 8.3.2 显示图像 ..... 295
  - 8.3.3 把图像保存到相册 ..... 295
- 8.4 解决方案：录制视频 ..... 297
  - 8.4.1 创建录制视频用的选取器 ..... 298
  - 8.4.2 保存视频 ..... 299
- 8.5 解决方案：用媒体播放器播放视频 ..... 299
- 8.6 解决方案：编辑视频 ..... 302
- 8.7 解决方案：选取并编辑视频 ..... 304
- 8.8 解决方案：通过电子邮件发送图片 ..... 306
- 8.9 解决方案：发送文本消息 ..... 309
- 8.10 解决方案：在社交网站发布消息 ..... 311
- 8.11 小结 ..... 313

## 第9章 创建并管理表格视图 ..... 314

- 9.1 iOS 的表格 ..... 314
- 9.2 委托 ..... 315
- 9.3 创建表格 ..... 316
  - 9.3.1 表格的样式 ..... 316
  - 9.3.2 排布表格视图 ..... 316
  - 9.3.3 设置数据源 ..... 317
  - 9.3.4 提供单元格 ..... 317

- 9.3.5 注册单元格类 ..... 317
- 9.3.6 从队列中取出单元格 ..... 318
- 9.3.7 设置 delegate ..... 318
- 9.4 解决方案：实现简单的表格 ..... 319
  - 9.4.1 数据源方法 ..... 319
  - 9.4.2 响应用户的触摸 ..... 322
- 9.5 UITableViewCell 类 ..... 322
  - 9.5.1 单元格的 selectionStyle 属性 ..... 323
  - 9.5.2 添加自定义的单元格受选效果 ..... 323
- 9.6 解决方案：创建带有选取标记的单元格 ..... 323
- 9.7 给单元格添加详情展示控件 ..... 325
- 9.8 解决方案：编辑表格 ..... 327
  - 9.8.1 添加撤销功能 ..... 331
  - 9.8.2 实现撤销功能 ..... 332
  - 9.8.3 显示移除单元格所用的控件 ..... 332
  - 9.8.4 处理删除请求 ..... 332
  - 9.8.5 通过滑动手势删除单元格 ..... 333
  - 9.8.6 调整单元格的顺序 ..... 333
  - 9.8.7 添加单元格 ..... 333
- 9.9 解决方案：操控表格的区段 ..... 334
  - 9.9.1 构建区段 ..... 334
  - 9.9.2 区段数量与区段内的行数 ..... 335
  - 9.9.3 返回单元格 ..... 335
  - 9.9.4 创建每个区段的头部标题 ..... 337
  - 9.9.5 定制表格与区段的头部及尾部 ..... 338
  - 9.9.6 创建区段索引 ..... 338
  - 9.9.7 处理索引与区段不匹配的问题 ..... 339
  - 9.9.8 为分区表格实现委托方法 ..... 339
- 9.10 解决方案：在表格中搜索 ..... 339
  - 9.10.1 创建搜索显示控制器 ..... 341
  - 9.10.2 为搜索显示控制器注册单元格 ..... 341

9.10.3 构建支持搜索功能的数据源方法	342
9.10.4 委托方法	343
9.10.5 使用与搜索功能相配套的索引	344
9.11 解决方案: 给表格添加下拉刷新功能	345
9.12 解决方案: 添加指令行	348
9.13 制作自定义的分组表格	351
9.14 解决方案: 构建含有多个滚轮的表格	352
9.14.1 创建 UIPickerView	353
9.14.2 数据源方法与委托方法	353
9.14.3 使用带有选取器的视图	354
9.15 使用 UIDatePicker	356
9.16 小结	357

## 第 10 章 集合视图 358

10.1 集合视图与表格的异同	358
10.2 建立集合视图	360
10.2.1 通过控制器使用集合视图	361
10.2.2 直接使用集合视图	361
10.2.3 数据源与委托	362
10.3 流式布局	362
10.3.1 滚动方向	362
10.3.2 条目的尺寸以及行间距	362
10.3.3 头部与尾部的尺寸	364
10.3.4 内边距	365
10.4 解决方案: 采用流式布局的简单集合视图	366
10.5 解决方案: 自定义单元格	370
10.6 解决方案: 水平滚动的列表	372
10.7 解决方案: 创建交互式的布局效果	375
10.8 解决方案: 滚动之后自动调整位置	377

10.9 解决方案: 创建圆形布局	378
10.9.1 实现创建条目与删除条目时的动画效果	381
10.9.2 增强圆形布局的实用性	382
10.9.3 布局对象	383
10.10 解决方案: 用手势调整布局	383
10.11 解决方案: 创建真正的网格状布局	385
10.12 解决方案: 为集合视图中的条目添加自定义菜单	391
10.13 小结	393

## 第 11 章 分享文档与数据 394

11.1 解决方案: 使用统一类型标识符	394
11.1.1 根据文件扩展名来决定 UTI	395
11.1.2 把 UTI 转换成扩展名或 MIME 类型	396
11.1.3 判断两个 UTI 之间是否有依从关系	397
11.1.4 获取依从关系列表	398
11.2 解决方案: 访问系统剪贴板	400
11.2.1 存储数据	401
11.2.2 存储常见类型的数据	401
11.2.3 获取数据	402
11.2.4 自动更新剪贴板	402
11.3 解决方案: 监控 Documents 文件夹	403
11.3.1 启用文件分享功能	403
11.3.2 用户对 Documents 文件夹的控制能力	403
11.3.3 在 Xcode 里访问应用程序沙盒	405
11.3.4 扫描新的文档	405
11.4 解决方案: 活动视图控制器	408

11.4.1	展示活动视图控制器	409
11.4.2	UIActivityItemSource 协议	409
11.4.3	UIActivityItemProvider 类	411
11.4.4	实现 UIActivityItemSource 协议中的回调方法	411
11.4.5	添加分享服务	412
11.4.6	与各种数据类型相对应的 操作	416
11.4.7	排除某些操作	417
11.5	解决方案: Quick Look 预览 控制器	417
11.6	解决方案: 使用文档交互 控制器	420
11.6.1	创建 UIDocumentInteraction- Controller 实例	420
11.6.2	UIDocumentInteraction- Controller 的属性	424
11.6.3	提供快速查看文档的功能	424
11.6.4	判断是否应启用 “Open in...” 操作	425
11.7	解决方案: 声明程序所支持的 文档类型	426
11.7.1	创建自定义的文档类型	427
11.7.2	实现对文档的支持	428
11.8	解决方案: 创建基于 URL 的 服务	431
11.8.1	声明模式	432
11.8.2	测试 URL	433
11.8.3	添加处理程序方法	433
11.9	小结	434

## 第 12 章 浅谈 Core Data 435

12.1	Core Data 简介	435
12.2	实体与模型	436
12.2.1	构建模型文件	436

12.2.2	属性与关系	437
12.2.3	构建 NSManagedObject 的 子类	437
12.3	创建上下文	438
12.4	添加数据	439
12.5	查询数据库	441
12.5.1	配置 NSFetchedRequest	442
12.5.2	执行数据获取操作	443
12.6	移除对象	444
12.7	解决方案: 用 Core Data 来充当 表格的数据源	445
12.7.1	访问索引路径	445
12.7.2	sectionNameKeyPath 属性	445
12.7.3	获取每个区段内的对象	445
12.7.4	sectionIndexTitles 属性	446
12.7.5	Core Data 与表格之间的紧密 结合	446
12.8	解决方案: 用 Core Data 实现 表格的搜索功能	448
12.9	解决方案: 为 Core Data 表格 视图添加编辑功能	450
12.9.1	添加撤销 / 重做功能	451
12.9.2	创建撤销事务	452
12.9.3	重新思考编辑功能	452
12.10	解决方案: 由 Core Data 所驱 动的集合视图	456
12.11	小结	461

## 第 13 章 网络编程基础 462

13.1	解决方案: 判断网络状态	462
13.2	监测联网状况是否发生变化	465
13.3	URL 加载系统	467
13.3.1	配置	467
13.3.2	任务	468
13.3.3	NSURLSession	468

13.4 解决方案: 简单的下载 .....	469	14.7 使用基本的方向值 .....	504
13.5 解决方案: 在下载过程中提供 反馈 .....	473	14.7.1 根据加速计来判断方向 .....	505
13.6 解决方案: 后台传输 .....	482	14.7.2 计算相对角度 .....	506
13.6.1 测试后台传输 .....	484	14.8 解决方案: 使用加速计来移动 屏幕上的物体 .....	507
13.6.2 Web 服务 .....	484	14.9 解决方案: 基于加速计的滚动 视图 .....	511
13.7 解决方案: 使用 NSJSONSeriali- zation 类 .....	485	14.10 解决方案: 获取并使用设备的 姿态 .....	513
13.8 解决方案: 将 XML 转换为树状 结构 .....	487	14.11 用 Motion Event 来检测晃动 .....	514
13.8.1 树 .....	489	14.12 使用外接屏幕 .....	515
13.8.2 构建解析树 .....	489	14.12.1 检测屏幕 .....	516
13.9 小结 .....	492	14.12.2 获取屏幕分辨率 .....	517
<b>第 14 章 针对特定设备的开发</b> .....	493	14.12.3 配置视频输出 .....	517
14.1 访问基本的设备信息 .....	493	14.12.4 添加 CADisplayLink .....	518
14.2 添加设备能力限制 .....	494	14.12.5 对过扫描进行补偿 .....	518
14.2.1 提供描述信息以征求用户 同意 .....	496	14.12.6 VIDEOkit .....	518
14.2.2 Info.plist 文件中其他常用的 键 .....	496	14.13 追踪用户 .....	521
14.3 解决方案: 检查设备距离与 电池状态 .....	496	14.14 查询可用的磁盘空间 .....	522
14.3.1 启用与禁用距离感应器 .....	497	14.15 小结 .....	523
14.3.2 监控电池状态 .....	497	<b>第 15 章 辅助功能</b> .....	524
14.3.3 判断设备是否具有 Retina 显示屏 .....	499	15.1 辅助功能基础知识 .....	524
14.4 解决方案: 获取设备的其他 信息 .....	500	15.2 启用辅助功能 .....	526
14.5 Core Motion 基础知识 .....	502	15.3 特征 .....	526
14.5.1 判断设备是否支持某种 感应器 .....	502	15.4 标签 .....	528
14.5.2 获取感应器数据 .....	503	15.5 提示语 .....	528
14.6 解决方案: 通过加速度来判断 “上”方向 .....	503	15.6 用模拟器测试辅助功能 .....	529
		15.7 把变化情况传播出去 .....	531
		15.8 在 iOS 上面测试辅助功能 .....	531
		15.9 语音合成 .....	533
		15.10 动态字体 .....	534
		15.11 小结 .....	535
		<b>附录 A Objective-C 字面量</b> .....	536

## 手势与触摸

触摸是 iOS 交互的核心，它提供了一种非常关键的手段，使用户可以向应用程序表达自己的意图。触摸并不局限于“按下按钮”或“点击键盘”这两种动作。你可以设计并构建出一种应用程序，令其能够以有意义的方式来直接处理用户的手势。本章将要创建一套“直接操纵界面”（direct manipulation interface），这套界面远比系统内置的控制手段强大。你会在本章中学到如何创建一种可供用户在屏幕上拖曳的视图。此外，还会学到怎样用手势识别器类来分辨并解析手势，手势是对触摸的一种高级抽象方式，而手势识别器类则可以自动检测常见的点击（tap）、滑动或扫屏（swipe）及拖曳（drag）等手势。学完本章之后，你就可以在自己的应用程序里面以各种不同的方式来实现手势控制了。

### 1.1 触摸

Cocoa Touch 以最简单的方式实现了直接操纵，也就是向用户正在操作的视图发送触摸事件。iOS 开发者会告诉这个视图应该如何响应用户。在开始讲手势和手势识别器之前，首先应该牢固地掌握底层触摸技术。因为所有基于触摸的交互操作都需要使用由底层触摸技术所提供的关键组件。

每次触摸都包含下列信息：触摸操作发生在何处（当前触摸的位置和上一次触摸的位置）、触摸操作位于哪个阶段（直接操纵界面中的“按下”、“移动手指”及“抬起手指”分别相当于桌面应用程序里的“按下鼠标按钮”、“移动鼠标”和“松开鼠标按钮”）、点击的次数（是单击还是双击）以及触摸操作发生的时间（以时间戳表示）。

iOS 使用响应者链来决定应该由哪个对象处理触摸。顾名思义，响应者就是响应事件的对象，而响应者链则是一条由可以处理这种事件的对象所组成的链。用户触摸屏幕时，应用



程序会找寻相关对象来处理此操作。触摸事件将在视图之间传递，直到某个对象开始负责响应该事件为止。

从根本上说，每个触摸操作及其信息都保存在 `UITouch` 对象里面，而这样一组 `UITouch` 则放在 `UIEvent` 对象里面传递。每个 `UIEvent` 对象都表示一次触摸事件，其中包含一个或多个 `UITouch`。具体数量是多少取决于两个因素：一是开发者设定应用程序应该以哪种方式来响应触摸（也就是说，开发者是否开启了多点触摸），二是用户如何触摸屏幕（也就是实际触摸了几个点）。

应用程序会在视图类或视图控制器类里接收到触摸事件，而这两种类都继承了 `UIResponder` 类，于是也就自动实现了触摸处理程序（touch handler）。开发者可以自己决定应该在哪个类里面处理并响应触摸事件。许多 iOS 开发新手都想试着在非响应者类里实现底层的手势控制，但他们可能会遇到麻烦。

在视图中处理触摸事件似乎有违常理。你可能觉得应该把界面的样貌（也就是视图）与它响应触摸的方式（也就是控制器）分开。此外，直接用视图来处理触摸似乎也有违“模型 - 视图 - 控制器”（Model-View-Controller）设计规范，不过，有时候这么做还是必要的，而且可以提升封装程度。

我们有时要面对多个能够响应触摸事件的子视图，比方说在棋类游戏中，棋盘上的棋子就是如此。如果直接把交互行为构建在视图类里面，那么开发者就能够在隐藏实现细节的前提下，向应用程序的核心代码发送含义丰富的反馈信息。例如，开发者可以在玩家结束操作之后告知模型：“兵”（pawn）移动到了“后”（Queen）这一侧“象”（Bishop）列的第 5 格<sup>①</sup>，假如不采取这种响应方式，那么开发者就要传输一系列没有意义的坐标变化信息了。响应触摸事件时，把棋子的移动细节隐藏起来，就可以用棋类游戏所专有的术语来编写模型部分的代码，而不用再考虑棋子在视图中的位置变化。

选择在 `UIView` 类中响应触摸事件的另一个原因是便于绘制。如果想令应用程序在响应用户的触摸时处理绘制操作（drawing operation），就需要把触摸处理程序放在视图中实现。因为视图控制器与视图不同，它并没有实现 `drawRect:` 方法，而在执行自定义的绘制操作时，`drawRect:` 却是个至关重要的方法。

在 `UIViewController` 类里响应触摸同样有其优点。这样做的好处是，我们不用把主要的事件处理逻辑放在另一个类里实现，而是可以直接把管理触摸的代码放在视图控制器中，以便在需要用到“按住不放”（tap-and-hold）及“滑动”等标准手势的时候解析它们。这种做法可以令代码保持集中，而且可以使控制器能够直接和应用程序的模型交互。

在后续的章节及解决方案里面，你将学到触摸操作的工作原理、如何在应用程序里响应触摸以及如何把用户所看到的内容同其与屏幕的交互方式联系起来。

### 1.1.1 触摸操作所处的阶段

触摸有其生命周期。每个触摸操作都会处于下列五个阶段之一，而这五个阶段合起来就

① 国际象棋步法的描述方式请参见 [https://en.wikipedia.org/wiki/Descriptive\\_chess\\_notation](https://en.wikipedia.org/wiki/Descriptive_chess_notation)。——译者注



代表了界面中的触摸操作所历经的过程。这些阶段分别是：

- **UITouchPhaseBegan**——用户一旦触摸屏幕，即进入此阶段。
- **UITouchPhaseMoved**——用户的手指在屏幕上移动。
- **UITouchPhaseStationary**——自上一个事件发生之后，用户仍然在触摸着屏幕表面，但却没有移动。
- **UITouchPhaseEnded**——当用户把触摸屏幕的手指从屏幕上拿开之后，就进入了这个阶段。
- **UITouchPhaseCancelled**——如果 iOS 系统不再追踪某个触摸操作，那么就会进入该阶段。这通常是因为系统中断而导致的，比方说，应用程序不再处于活动状态，或相关的视图已经从视窗里移走。

总体来说，用这五个阶段就可以描述触摸事件了。它们描述了在处理界面中的触摸操作或无法处理某个触摸操作时所能遇到的全部状况，并且为相关界面提供了一套基本的控制手段。至于如何解析并响应这些情况，就是每一位开发者的事情。你可以实现一系列的响应者方法<sup>①</sup>来响应触摸事件。

### 1.1.2 · UIResponder 类中的触摸事件响应方法

包括 `UIView` 及 `UIViewController` 在内的所有 `UIResponder` 子类都可以响应触摸。每个类都可以决定自己要如何响应。决定好响应方式之后，就可以在类中实现自定义的行为了，而当用户拿一个或多个手指触摸视图（View）或视窗（Windows）时，该类能够以定义好的行为来响应触摸。

`UIResponder` 类中预先定义了一些回调方法，分别用来处理“用户开始触摸屏幕”、“在屏幕上移动手指”以及“从屏幕上拿开手指”等情况。相关的方法一共有四个，它们对应于上一节所讲的阶段：

- **`touchesBegan:withEvent:`**——当触摸事件处于“起步阶段”（starting phase），也就是用户刚开始触碰屏幕时，系统会调用这个方法。
- **`touchesMoved:withEvent:`**——当用户触摸屏幕并持续在屏幕上移动手指时，系统会调用这个方法。
- **`touchesEnded:withEvent:`**——当用户把触摸屏幕的一根手指或所有手指都从屏幕上拿开时，触摸过程就结束了，而系统此时会调用这个方法。如果在用户移动手指的过程中程序做了一些处理，那么此时应该执行相关的清理工作。
- **`touchesCancelled:WithEvent:`**——如果目前正在发生的触摸事件遭到系统阻断，致使 Cocoa Touch 必须对此做出响应，那么系统就会调用这个方法。

上面列出的每一个方法都是 `UIResponder` 方法，它们通常会由 `UIView` 或 `UIViewController` 子类来实现。所有的视图都会继承这几个方法，而继承下来的这些方法并没有实际的功效。如果想使自己的应用程序支持触摸，就应该覆写这些方法，并在其中

① 作者把 `UIResponder` 类里面与处理触摸事件有关的几个方法叫作“响应者方法”或“`UIResponder` 方法”，下同。——译者注

编写代码，以实现应用程序所需的响应功能。请注意，UITouchPhaseStationary 并不会触发回调。

你自己的类可以把四个方法全都实现了，也可以只实现其中的某些方法。在制作真实的应用程序时，开发者一般都会实现 `touchesCancelled:WithEvent:`，以便处理用户将手指拖动到屏幕边界外面或者有电话打进来的情况，在这两种情况下，当前的触摸序列（touch sequence）会取消。一般来说，当触摸事件取消时，我们可以在回调方法里面调用 `touchesEnded:withEvent:` 方法。这样做可以使代码“走”完整个触摸序列，即使用户还没有把手指从屏幕上拿起，程序也依然可以完成触摸序列。在处理触摸时，苹果公司建议开发者最好把四个方法全都覆写。



视图有个模式叫作 *exclusive touch*，此模式会阻止系统把触摸投递给同一个视窗里的其他视图。启用此模式之后，也就是把 `exclusiveTouch` 属性设为 YES 之后，这个视窗里的其他视图就收不到触摸事件了，而是由主视图来专门负责处理全部的触摸事件。

### 1.1.3 对视图的触摸

当屏幕上有很多视图的时候，iOS 会自动判断出用户触摸的是哪个视图，然后会把触摸事件传给适当的视图去处理。这样一来，开发者就可以编写具体的直接操纵界面，使用户可以触摸、拖曳并与屏幕上的物件交互。

某个视图上面发生了触摸操作并不意味着非得由这个视图来响应触摸。每个视图都可以经由触摸判定机制（hit test）来选择是处理这次触摸，还是把它留给下面的视图去处理。在后面的解决方案中，你会看到可以巧妙地运用一些响应策略来决定视图应该何时响应触摸，尤其当我们使用了半透明的特殊图形时，更可以灵活运用响应策略。

首个通过点击测试的视图可以决定自己是否处理这个触摸操作。如果此视图决定把这个触摸操作传递出去，那么它就会到达该视图的上级视图（superview），并且可以沿着响应者链逐级向上传递，直至有视图处理它，或是到达拥有这些视图的视窗为止。若视窗也不处理此操作，那么触摸事件就会移动到 UIApplication 实例，该实例要么处理它，要么丢弃它。

### 1.1.4 多点触摸

iOS 支持单点触摸（Single-Touch）和多点触摸（Multi-Touch）界面。单点触摸 GUI 每次只能处理一个触摸操作。在这种情况下，开发者无须判断当前正在追踪的是哪一个触摸事件。接收到的触摸事件也就是待处理的触摸事件。开发者只需要查看其数据、响应此事件，并等待下一个事件即可。

在处理多点触摸，也就是同时响应屏幕上的多个触摸时，开发者收到的是一组触摸。你需要决定其顺序并响应这一组触摸。不过，你也可以自己分别追踪每个触摸事件，观察它们各自是如何随着时间的推移而改变的，这样的话，你就可以实现出一套更为丰富的用户交互功能了。本章稍后的解决方案将会演示单点触摸和多点触摸。

### 1.1.5 手势识别器

手势识别器是苹果公司提供的一种强大的识别方式，用来检测界面中的特定手势。手势识别器简化了触摸程序的设计。这些识别器本身就封装了与触摸有关的方法，所以开发者无须自己来实现，它们提供了一套目标-动作（target-action）反馈机制，这套机制能把实现细节隐藏起来。这些识别器同时也形成了一套标准，可以把某些移动形式划分到不同的类别里面，例如拖曳（drag）、滑动（swipe）等。

使用了手势识别器类之后，当用户做出点击（tap）、双指聚拢（pinch）、旋转（rotate）、滑动（swipe）、拖动（pan）或长按（long press）手势时，系统就会自动触发相关的回调方法。有了这种检测能力，开发基于触摸的界面就变得更加简单了。为了提高程序的可靠程度，你可以自己编写相关的代码，但是大部分开发者都觉得系统自带的识别器已经足够健壮，并可以满足很多应用程序的需求。本章将会给出几个基于识别器的解决方案。由于各种识别器的基本工作原理都一样，所以你可以轻易地扩展这些解决方案，以满足自己的手势识别需求。

下面列出新版 iOS SDK 内置的几种手势：

- **点击**——用户拿一根或多根手指触碰屏幕，这就是点击。用户可以拿一根手指做出点击手势，也可以用多根手指做出该手势。开发者通过指定 `gestureRecognizers` 属性来设定自己想要检测的点击次数。你可以创建一种点击识别器，令其检测用一根手指所做的点击，也可以创建更为复杂的识别器，比方说，可以创建一种识别器，用来判断用户是否以两根手指做了三次点击（two-fingered triple-tap）。
- **滑动**——在水平或垂直方向（也就是上、下、左、右）上所做出的短距离单点触摸或多点触摸手势，就叫作滑动或扫屏。这种手势不能太偏离主方向（primary direction）。开发者可以在识别器上面设定想要侦测的方向，而识别器则会把侦测到的方向作为属性返回给开发者。
- **双指聚拢**——用户拿两根手指向内聚拢，这就是双指聚拢（或缩小、挤压）手势；两根手指向外移动，各自远离，这就是双指张开（或扩大、拉伸）手势。识别器会根据挤压或拉伸的程度把缩放因子返回给开发者。
- **旋转**——两根手指同时依顺时针或逆时针移动，这就是旋转手势，识别器会把旋转角度和旋转速度返回给开发者。
- **拖动**——拿手指在屏幕上做出拖曳动作，这就是拖动（或拖移）手势。识别器会侦测出拖曳操作所产生的坐标变化。
- **长按**——用户触摸屏幕，并在某段时间内按着手指不放，这就是长按手势。开发者可以指定用户必须按下多少根手指才能令识别器侦测到该手势。

## 1.2 解决方案：添加简单的直接操纵界面

在开始讲比较流行（也比较常用）的手势识别器之前，我们先花点时间看看如何用传统的方式响应用户触摸。在明白了 `UIResponder` 类里面简单的触摸事件处理机制如何运作之

后，你就能深入理解触摸界面了。

制作直接操纵界面时，开发者的设计重点是 `UIView`，而不是 `UIViewController`。研发直接操纵界面时的核心问题就是视图，或者说得更准确一些，是 `UIResponder`。我们自己编写从 `UIResponder` 类继承下来的相关方法，以创建基于触摸的界面。

解决方案 1-1 中的代码直接实现了触摸功能。这段范例程序创建了 `UIImageView` 类的子类 `DragView`，并为该类添加了响应触摸的能力。由于这是个 `ImageView`，所以要记得开启用户交互功能（也就是说，应该把 `setUserInteractionEnabled` 设为 `YES`）。此属性既会影响视图本身，也会影响它的所有子视图。大部分的视图在默认情况下都已经启用了用户交互功能，但 `UIImageView` 是个例外，许多新手都在这个问题上卡住了。显然，苹果公司觉得大部分人都不会在 `UIImageView` 上面使用触摸功能。

这个解决方案会更新 `DragView` 的中心坐标，使之与屏幕上的触摸移动情况相符。用户首次触碰屏幕上的某个 `DragView` 时，该 `DragView` 对象会把触摸点与 `DragView` 的原点之间的偏移量保存到 `startLocation` 对象里面。用户拖曳这个 `DragView` 的时候，它会随着用户的手指一起移动，也就是说，它目前的原点与用户目前的触摸点之间的偏移量会与 `startLocation` 里面所保存的那个量相同，这样的话，移动效果就会显得比较平滑。我们通过更新 `DragView` 对象的中心点（`center`）来移动 `DragView`。解决方案 1-1 中的代码会算出 `x` 轴和 `y` 轴方向上的偏移量，并在用户每次移动手指的时候据此调整 `DragView` 的中心点。

#### 解决方案 1-1 创建可以拖曳的视图

```
@implementation DragView
{
    CGPoint startLocation;
}

- (instancetype)initWithImage:(UIImage *)anImage
{
    self = [super initWithImage:anImage];
    if (self)
    {
        self.userInteractionEnabled = YES;
    }
    return self;
}

- (void)touchesBegan:(NSSet*)touches withEvent:(UIEvent*)event
{
    // Calculate and store offset, pop view into front if needed
    startLocation = [[touches anyObject] locationInView:self];
    [self.superview bringSubviewToFront:self];
}

- (void)touchesMoved:(NSSet*)touches withEvent:(UIEvent*)event
{
    // Calculate offset
```

```

CGPoint pt = [[touches anyObject] locationInView:self];
float dx = pt.x - startLocation.x;
float dy = pt.y - startLocation.y;
CGPoint newcenter = CGPointMake(
    self.center.x + dx,
    self.center.y + dy);

// Set new location
self.center = newcenter;
}
@end

```

用户所触摸的 DragView 会显示在屏幕最前方，这是因为我们在 `touchesBegan:withEvent:` 里调用了—一个方法，这个方法会告诉 DragView 的上级视图应该把当前 DragView 带到屏幕最前方。这样的话，当前处于活动状态的元件就总是会出现在界面的最上层。

这个解决方案并没有实现与触摸结束 (`touches-ended`) 或触摸取消 (`touches-cancelled`) 有关的方法<sup>①</sup>，它只关注用户在屏幕上移动手指的操作。当用户不再和屏幕交互时，DragView 类不需要做后续的处理。

---

#### 获取解决方案代码

访问 <https://github.com/erica/iOS-7-Cookbook> 网页，并打开“C01 Gestures”文件夹，即可找到与本章中的解决方案相对应的完整范例项目。

---

## 1.3 解决方案：添加拖动手势识别器

用手势识别器也可以实现与解决方案 1-1 相同的交互功能，而且还不用直接编写触摸处理程序<sup>②</sup>。拖动手势识别器可以侦测拖曳手势。只要 iOS 系统检测到拖动手势，它就会触发你所指定的回调方法。

解决方案 1-2 的代码与解决方案 1-1 的相似，在首次初始化时，这段代码会向视图中添加识别器。当 iOS 发现用户在 DragView 实例上面执行拖曳时，就会触发 `handlePan:` 回调方法，而该方法会更新 DragView 的中心点，使之与用户所拖曳的距离相符。

这段代码计算距离的方式看上去似乎有些奇怪。它把视图原来的位置保存在名为 `previousLocation` 的实例变量中，然后在系统每次检测到拖动手势并触发回调方法时，计算目前位置与 `previousLocation` 之间的偏移。我们本来可以执行仿射变换 (`affine transform`) 或运用 `setTranslation:inView:` 方法，但本例却采用了一种不常见的办法，就是移动视图的中心点。解决方案 1-2 根据  $x$  轴和  $y$  轴上面的偏移量创建了 `CGPoint`，并依

---

① 具体指的就是 `touchesEnded:withEvent:` 方法与 `touchesCancelled:withEvent:` 方法，下同。——译者注

② 作者把 `touchesMoved:withEvent:` 等四个用来处理触摸事件的方法称为触摸处理程序 (`touch handler`) 或触摸方法，下同。——译者注

照这个偏移量来设定视图的中心点，以此来改变视图的实际位置。

仿射变换与简单的偏移量不同，它可以同时达成旋转、缩放及平移操作。为了支持变换，手势识别器以绝对量的方式来描述坐标改变，而不是给出两次改变之间的相对差值。这样就不需要把多个偏移量累加起来了，因为 `UIPanGestureRecognizer` 只返回一个变化量，这个变化量以某个视图的坐标系来描述位置的变化，一般来说，参照的就是当前视图的上级视图的坐标系。有了这个平移变化量，我们就可以执行一些简单的仿射变换计算了，而且还可以通过数学运算的方式与其他变换结合起来，达到一次性执行多种变换操作的效果。

如果不保存状态，而是直接执行变换，那么 `handlePan:` 方法的代码就会变成下面这个样子：

```
-(void)handlePan:(UIPanGestureRecognizer *)uigr
{
    if (uigr.state == UIGestureRecognizerStateEnded)
    {
        CGPoint newCenter = CGPointMake(
            self.center.x + self.transform.tx,
            self.center.y + self.transform.ty);
        self.center = newCenter;

        CGAffineTransform theTransform = self.transform;
        theTransform.tx = 0.0f;
        theTransform.ty = 0.0f;
        self.transform = theTransform;

        return;
    }
}
```

```
CGPoint translation = [uigr translationInView:self.superview];
CGAffineTransform theTransform = self.transform;
theTransform.tx = translation.x;
theTransform.ty = translation.y;
self.transform = theTransform;
}
```

请注意，手势识别器会在交互操作结束的时候更新视图的位置，并重设 `transform` 属性的 `tx` 与 `ty`。经过上述改编之后，我们就无须在程序里记录上一次的位置了，于是也就用不着 `touchesBegan:withEvent:` 方法了。若是不按照上述代码来编写，那么解决方案 1-2 必须保存前一次的状态。

#### 解决方案 1-2 用拖动手势识别器实现可供拖曳的视图

```
@implementation DragView
{
    CGPoint previousLocation;
}

-(instancetype)initWithImage:(UIImage *)anImage
```

```

{
    self = [super initWithImage:anImage];
    if (self)
    {
        self.userInteractionEnabled = YES;
        UIPanGestureRecognizer *panRecognizer =
            [[UIPanGestureRecognizer alloc]
             initWithTarget:self action:@selector(handlePan:)];
        self.gestureRecognizers = @[panRecognizer];
    }
    return self;
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Promote the touched view
    [self.superview bringSubviewToFront:self];

    // Remember original location
    previousLocation = self.center;
}

- (void)handlePan:(UIPanGestureRecognizer *)uigr
{
    CGPoint translation = [uigr translationInView:self.superview];
    self.center = CGPointMake(previousLocation.x + translation.x,
                               previousLocation.y + translation.y);
}
@end

```

## 1.4 解决方案：同时使用多个手势识别器

解决方案 1-3 是基于解决方案 1-2 的思路而构建的，但是两者之间有一些区别。首先，它创建了多个平行运作的识别器。该解决方案的代码分别使用了旋转、双指聚拢及拖动这三种识别器，并把它们全都添加到了 DragView 的 `gestureRecognizers` 属性中，然后它把 DragView 设为每个识别器的委托<sup>①</sup>。这样的话，DragView 就可以实现名为 `gestureRecognizer:shouldRecognizeSimultaneouslyWithGestureRecognizer:` 的委托方法，从而令多个识别器能够同时运作。我们必须实现好这个方法，并将其返回值设为 YES，否则在同一时刻就只会一个识别器起作用。平行使用多个识别器可以实现出一些功能，例如：当用户做出双指聚拢手势时，我们可以同时用缩放及旋转效果来响应该手势。

① “把 A 设为 B 的委托”或“把 A 赋给 B 的委托属性”这种句式，其大概的意思是：B 需要通过某个实现了 C 协议的对象来处理发生在 B 身上的一些事件，而 A 对象所属的类实现了 C 协议，于是，我们就用 A 来处理发生在 B 上面的事件。这时，C 协议中所约定的一些方法就叫作“B 的委托方法”，而 A 或 A 所属的类则可以泛称为“B 的委托”。具体到解决方案 1-3 来说，这种句式对应于代码中的“`recognizer.delegate = self;`”，其中的 `self` 就是 A，`recognizer` 就是 B。——译者注



**提示** UITapGestureRecognizer 对象保存了含有手势识别器的数组。这个数组里的每个元素都是手势识别器，而每个识别器都用来接收相关的触摸对象。如果创建某个视图时没有指定手势识别器，那么在系统传给响应者方法的触摸对象里面，gestureRecognizers 数组就是空的。

解决方案 1-3 扩充了视图的状态，给对象添加了与缩放和旋转功能有关的实例变量。这些变量可以记录前一次的变换值，而程序代码则会根据它们来合成仿射变换。仿射变换的效果是在解决方案 1-3 的 updateTransformWithOffset: 方法里面合成出来的，该方法把平移、旋转及缩放这三种操作合并成一个效果。与前一个解决方案不同，这份解决方案统一经由 self.transform 来实现对视图的修改，而这也是使用手势识别器时的常见做法。

### 解决方案 1-3 同时识别多种手势

```
@interface DragView : UIImageView <UIGestureRecognizerDelegate>
@end

@implementation DragView
{
    CGFloat tx; // x translation
    CGFloat ty; // y translation
    CGFloat scale; // zoom scale
    CGFloat theta; // rotation angle
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Promote the touched view
    [self.superview bringSubviewToFront:self];

    // initialize translation offsets
    tx = self.transform.tx;
    ty = self.transform.ty;
    scale = self.scaleX;
    theta = self.rotation;
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    if (touch.tapCount == 3)
    {
        // Reset geometry upon triple-tap
        self.transform = CGAffineTransformIdentity;
        tx = 0.0f; ty = 0.0f; scale = 1.0f; theta = 0.0f;
    }
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
```



```

{
    [self touchesEnded:touches withEvent:event];
}

- (void)updateTransformWithOffset:(CGPoint)translation
{
    // Create a blended transform representing translation,
    // rotation, and scaling
    self.transform = CGAffineTransformMakeTranslation(
        translation.x + tx, translation.y + ty);
    self.transform = CGAffineTransformRotate(self.transform, theta);

    // Guard against scaling too low, by limiting the scale factor
    if (self.scale > 0.5f)
    {
        self.transform = CGAffineTransformScale(self.transform, scale, scale);
    }
    else
    {
        self.transform = CGAffineTransformScale(self.transform, 0.5f, 0.5f);
    }
}

- (void)handlePan:(UIPanGestureRecognizer *)uigr
{
    CGPoint translation = [uigr translationInView:self.superview];
    [self updateTransformWithOffset:translation];
}

- (void)handleRotation:(UIRotationGestureRecognizer *)uigr
{
    theta = uigr.rotation;
    [self updateTransformWithOffset:CGPointZero];
}

- (void)handlePinch:(UIPinchGestureRecognizer *)uigr
{
    scale = uigr.scale;
    [self updateTransformWithOffset:CGPointZero];
}

- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer
    shouldRecognizeSimultaneouslyWithGestureRecognizer:
        (UIGestureRecognizer *)otherGestureRecognizer
{
    return YES;
}

- (instancetype)initWithImage:(UIImage *)image
{
    // Initialize and set as touchable
    self = [super initWithImage:image];
}

```

```

if (self)
{
    self.userInteractionEnabled = YES;

    // Reset geometry to identities
    self.transform = CGAffineTransformIdentity;
    tx = 0.0f; ty = 0.0f; scale = 1.0f; theta = 0.0f;

    // Add gesture recognizer suite
    UIRotationGestureRecognizer *rot =
        [[UIRotationGestureRecognizer alloc]
         initWithTarget:self
         action:@selector(handleRotation:)];
    UIPinchGestureRecognizer *pinch =
        [[UIPinchGestureRecognizer alloc]
         initWithTarget:self
         action:@selector(handlePinch:)];
    UIPanGestureRecognizer *pan =
        [[UIPanGestureRecognizer alloc]
         initWithTarget:self
         action:@selector(handlePan:)];
    self.gestureRecognizers = @[rot, pinch, pan];
    for (UIGestureRecognizer *recognizer
         in self.gestureRecognizers)
        recognizer.delegate = self;
}
return self;
}
@end

```

最后请大家注意，这份解决方案同时实现了两套手势识别机制。一方面，我们给视图的 `gestureRecognizers` 数组里添加了手势识别器对象，而另一方面，我们像解决方案 1-1 那样，通过基本的触摸方法来捕捉“三连击”（triple-tap）<sup>①</sup>操作。在本例中，当用户执行三连击操作时，会把视图的 `transform` 重置为恒等变换<sup>②</sup>。这样的话，原来对该视图所做的操作就将全部还原，这个视图的位置、方向及尺寸都会回到原来的样子。通过这段代码，大家可以看到 `touchesBegan`、`touchesMoved`、`touchesEnded` 及 `touchesCancelled` 这四个方法与手势识别器的回调方法是并行不悖（work seamlessly）的，笔者之所以要在解决方案中同时实现两套手势识别机制，就是为了使大家体会到这一点。同样的功能其实也可以通过添加 `UITapGestureRecognizer` 来实现。

大家通过这个解决方案可以感觉到，用手势识别器来处理触摸操作是非常简明的。

## 解决手势冲突

如果需要同时辨认多种手势，那么可能会产生手势冲突（gesture conflict）。比方说，需

① 短时间内连续做三次点击。——译者注

② 也就是没有经过任何平移、旋转及缩放的变换效果，可以理解为“单位矩阵”。——译者注

要同时识别单击 (single-tap) 和双击 (double-tap) 的时候, 就会出现问题。如果用户想执行的是双击操作, 那么负责识别单击操作的识别器是应该在用户第一次点击的时候就触发单击操作, 还是应该等到用户彻底不打算点击第二下的时候再去响应呢? iOS SDK 提供了一些手段, 使开发者可以通过代码来解决这些冲突。

我们在编写类的代码时, 可以指明必须等某个手势无法触发时, 才去触发另一个手势。这种规则可以通过调用 `requireGestureRecognizerToFail:` 方法来确立。 `UIGestureRecognizer` 里的这个方法接收一个参数, 也就是另外一个手势识别器。调用该方法之后, 两个手势识别器之间就创建了依赖关系。系统必须先确定另外一个手势无法触发, 然后才能触发当前这个手势。假如系统可以辨识出另外一个手势, 那么就不触发这个手势了。

iOS 7 引入了一些新的 API, 经由 `UIGestureRecognizer` 的委托及子类, 我们可以在运行期 (runtime) 实现更为灵活的手势判定规则<sup>①</sup>。在 `UIGestureRecognizer` 的委托中, 我们可以实现 `gestureRecognizer:shouldRequireFailureOfGestureRecognizer:` 方法及 `gestureRecognizer:shouldBeRequiredToFailByGestureRecognizer:` 方法, 而在 `UIGestureRecognizer` 的子类中, 则可以覆写 `shouldRequireFailureOfGestureRecognizer:` 方法及 `shouldBeRequiredToFailByGestureRecognizer:` 方法。

对于上面所说的四个方法, 其返回值均为布尔类型。如果返回的是 YES, 就意味着必须等某个手势彻底无法触发时才能触发另一个手势。每当识别器想要辨识某个手势时, 系统就会调用 `UIGestureRecognizer` 的相关委托方法, 这样一来, 我们就可以在处于不同视图体系的识别器对象之间建立依赖关系了, 此外, 如果想定义针对某个 `UIGestureRecognizer` 子类的手势辨识规则, 那么可以在其中覆写并实现相关的方法。

在开发真实的应用程序时, 如果设定了手势间的依赖关系, 就意味着识别器的反应时间会有所延迟, 因为它得等另一个识别器彻底失败为止。也就是说, 它得等到另一个手势彻底无法触发, 只有到那时, 该识别器才能完成对本手势的辨识。如果我们要同时辨识单击和双击, 那么当用户初次点击屏幕之后, 应用程序必须多等一会儿, 才能确定是不是发生了单击。因为它必须在确定了不可能发生双击之后, 才能去触发单击。假如触发了双击, 那么就不再触发单击了, 这两种手势只能触发一种。

为了适应这种规则变化, GUI 的响应能力会下降, 它对单击操作的响应会稍显“迟钝”。这是因为程序必须等待一段时间, 才能判断出是否会发生双击。假如程序的高速响应能力对用户体验有着非常重要的影响, 那么就不要再同时使用这两种识别器了, 而是应该把单击行为解读为立刻执行某操作。在这种情况下, 不应该同时设计单击及双击这两种手势。

大家可别忘了, 开发者可以随时添加、移除或禁用手势识别器。比方说, 用户在执行单击操作之后, 程序界面会进入另一种模式, 而在那种模式下, 用户可以分别通过单击和双击来做不同的事情。离开了那种模式之后, 开发者可以将辨识双击手势所用的识别器移除或禁用, 这样能够重新提高程序对单击操作的辨识速度。在确有必要时, 这种优化技术可以缓解界面的延迟现象。

① 原文称为 failure condition (失败条件)。——译者注

## 1.5 解决方案：限制移动

本章前面几个解决方案所使用的办法都比较简单，但是有个问题，就是用户完全有可能把某个视图拖动到屏幕之外，导致其难以把自己看不到的视图重新拖回到屏幕内。那些解决方案没有对移动施加限制。它们并没有判断用户所要操作的对象是不是处在大视图的范围内，而且也没有判断那个对象是不是可供触摸。解决方案 1-4 对视图在其上级视图内的移动行为做出了限制，从而解决了此问题。具体的限制办法是：从  $x$  轴和  $y$  轴方向上分别检查视图的移动是否符合规则，以此来限制其在每个方向上的移动行为。分两个轴来检测有个好处，就是即便视图对象在某个轴上已经移动到了边界，它也依然可以在另一个轴上面继续移动。比方说，就算视图已经碰到了上级视图的右边界，它也照样可以在垂直方向上下移动。



**提示** iOS 7 引入了一套 UIKit Dynamics API，用于对真实的物理现象进行建模，其中包含物理模拟及响应式动画等功能。开发者可以使用这套声明式的 Dynamics API 来对重力、碰撞、力、附着、弹簧、伸缩等大量现象进行建模，并将其效果运用在 UIKit 物件上面。但是本条解决方案采用传统的做法，也就是通过手势识别器以及直接框架操作（direct frame manipulation）来实现 UI 物件的移动并对其施加限制，读者可以用 Dynamics 实现出更为精致的版本。

图 1-1 演示了一个样本界面。程序将各个子视图（也就是图中的花）的移动范围限制在界面中央的黑色矩形内，使用户无法将其拖到屏幕范围之外。解决方案 1-4 的代码写得比较通用，所以你可将其改编，使之适用于任意尺寸的外围边界（parent bound）及子视图。

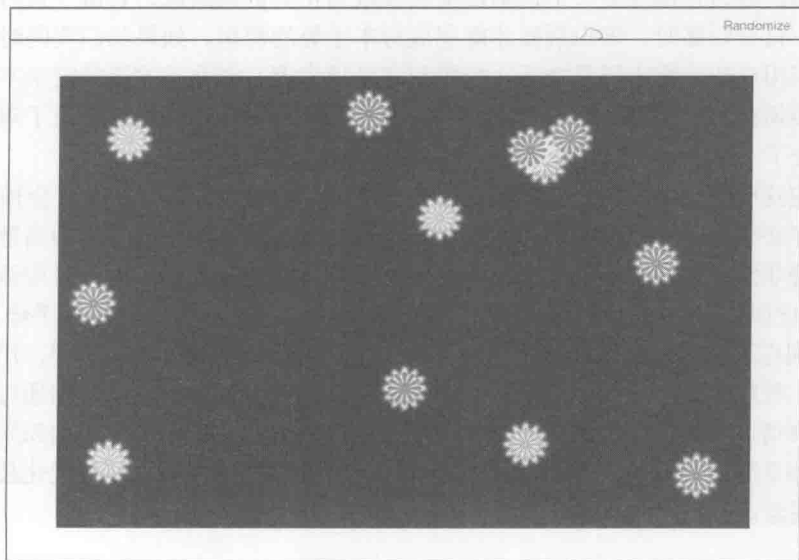


图 1-1 花的移动范围局限在黑色矩形内

## 解决方案 1-4 范围受限的移动

```

- (void)handlePan:(UIPanGestureRecognizer *)uigr
{
    CGPoint translation = [uigr translationInView:self.superview];
    CGPoint newcenter = CGPointMake(
        previousLocation.x + translation.x,
        previousLocation.y + translation.y);

    // Restrict movement within the parent bounds
    float halfx = CGRectGetMidX(self.bounds);
    newcenter.x = MAX(halfx, newcenter.x);
    newcenter.x = MIN(self.superview.bounds.size.width - halfx,
        newcenter.x);

    float halfy = CGRectGetMidY(self.bounds);
    newcenter.y = MAX(halfy, newcenter.y);
    newcenter.y = MIN(self.superview.bounds.size.height - halfy,
        newcenter.y);

    // Set new location
    self.center = newcenter;
}

```

## 1.6 解决方案：测试触摸

由于直接操纵界面中的大部分屏幕视图元件（onscreen view element）都不是矩形，所以触摸测试实现起来会比较困难，因为视图所在的矩形框与实际应该接受触摸的范围并不完全相同。图 1-2 演示了这个问题。右侧的屏幕截图展示了整套界面以及界面中可供触碰的各个子视图，而左侧的截图则描绘出每个子视图周围的矩形框<sup>①</sup>。其中灰色的部分虽然处在矩形框范围之内，但却在圆圈外面，因此，如果用户触碰了这个区域，那么应用程序不应该判定其点击（hit）了相关的子视图。

只要触碰点位于视图的范围内，iOS 就会侦测到这一动作。这个范围既包括视图的主体部分，也包括用户看不到的区域，也就是图 1-2 左侧截图中处于圆形之外但又位于矩形之内的灰色区域。在图 1-2 右侧截图中，每个 UIView 本体周围的透明部分（clear portion）的下方可能还有别的视图，所以我们必须添加某种触摸判定机制（hit test），才能使用户可以触碰到那些视图。

如果想把视图所对应的矩形框显示出来，那么可以用下面代码来设定其背景色：

```
dragger.backgroundColor = [UIColor lightGrayColor];
```

这样就可以在不影响屏幕实际图形的前提下，给它们后方画上一块灰色的板子，如图 1-2 左侧截图所示。在本例中，每个视图的主体部分都是圆形图样，如果不添加上面的代

① 也称为范围框、边界框、绑定框。——译者注

码, 那么其背景色就是透明的。由此可见, 我们必须添加某种触摸判定机制, 否则, 如果用户点击了透明部分, 系统就会认为用户点击的是相关的视图。手工指定背景色是一种便捷的调试手法, 它使开发者能够看到每个视图真正应该接受触摸的范围。在发布正式产品之前, 别忘了把对背景色的赋值语句注释掉。除了笔者所说的这种调试办法之外, 还有一种办法, 就是设定视图的层 (layer) 所具备的边框宽度及样式。

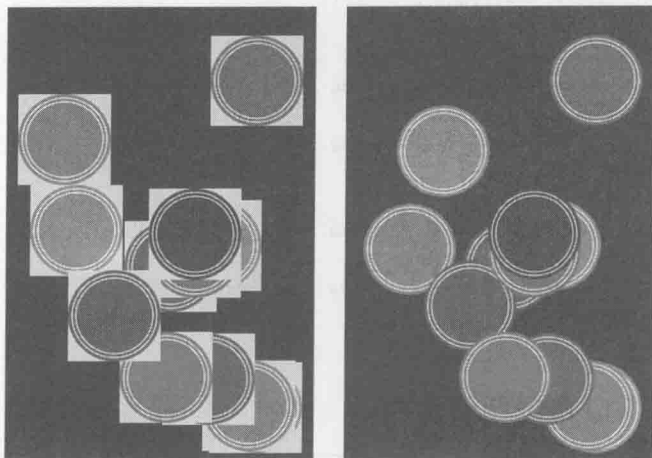


图 1-2 在左图中, 如果用户触摸了圆圈外围的灰色区域, 那么应用程序不应判定其点击了相关的子视图。右边这幅图中, 把左图中每个子视图周围的灰色部分用全透明色 (也就是 alpha 值为 0 的颜色) 来描绘, 这样就能显露出实际应该接受触碰的部分了

解决方案 1-5 给视图添加了简单的触摸判定机制, 以判断触摸点是否在圆圈范围内。我们是通过覆写 `UIView` 的 `pointInside:withEvent:` 方法来实现此机制的。该方法如果返回 YES, 就表明触摸点位于视图之内, 如果返回 NO, 则表明不在视图之内。本例所实现的判定机制采用的是基本的几何运算, 也就是检查触摸点是不是在圆圈的半径之内。你也可以采用与自己的应用程序相符的其他办法来判断用户是否触摸了屏幕上的视图。在下一节的解决方案 1-6 中, 笔者会对判定代码进行扩充, 以实现更加精细的控制。

#### 解决方案 1-5 判断某个点是否位于圆形的视图中

```
- (BOOL)pointInside:(CGPoint)point withEvent:(UIEvent *)event
{
    CGPoint pt;
    float halfSide = kSideLength / 2.0f;

    // normalize with centered origin
    pt.x = (point.x - halfSide) / halfSide;
    pt.y = (point.y - halfSide) / halfSide;


    // x^2 + y^2 = radius^2
    float xsquared = pt.x * pt.x;
    float ysquared = pt.y * pt.y;
```

```

// If the radius < 1, the point is within the clipped circle
if ((xsquared + ysquared) < 1.0) return YES;
return NO;
}

```

请注意，对于配备了 Retina 显示屏的设备，其触摸检测方式与老式的设备相同，在做数学运算时，使用的也是标准的点坐标系统，而不是实际的像素。设备上多出来的像素并不影响处理手势时所用的数学算式。视图的坐标系统依然采用亚像素精度（subpixel accuracy）的浮点数。设备绘制屏幕内容时所采用的具体像素数既不会影响 UIView 的范围框，也不会影响 UITouch 中的坐标。它只是一种能在坐标系统中提高绘制精度的手段。

 **提示** 我们这里所说的触摸判定机制（hit test）是指一种判断点是不是处于视图之中的机制，而不是指名字看上去与之相似的 hitTest:withEvent: 方法。这个方法用于返回视图体系中包含某个点的最顶层视图（topmost view，也就是距离用户及屏幕最近的视图）。它会在每个视图上面调用 pointInside:withEvent:。如果 pointInside 方法返回 YES，那么它就会沿着视图体系继续向下判断。

## 1.7 解决方案：针对位图的触摸测试

解决方案 1-5 所用的触摸判定方式非常直观，它只做了一些简单的几何运算，但不巧的是，大部分视图都不是解决方案 1-5 所演示的样子。比方说，对于图 1-1 中的花朵，其边界就是不规则的，而且其透明度也有变化。如果图形比较复杂，我们就必须针对位图来做测试，才能判断是否发生了触摸。对基于图像的视图（image-based view）来说，位图是一种按字节排列的信息，它描述了该视图的内容，从而使开发者可以判断出用户到底是触摸了图中不透明的部分（solid portion），还是触摸了透明的部分，如果是后者，就应该转而判断位图下方的视图是否受到触摸。

解决方案 1-6 从 UIImageView 里面提取了一幅图像的位图。它假定受测的视图是以像素来描述其中的图像的。假如你要扭曲视图的话（一般可以通过调整框架（frame）的大小或运用坐标变换来实现扭曲效果），那么需要修改判定触摸所用的算式。开发者可以通过 CGPointApplyAffineTransform() 来对 CGPoint 执行变换，以处理由于缩放及旋转而带来的坐标变更。为了简化判断过程，也为了避免繁杂的数学运算，我们把视图的图像与其实际像素之比定为 1 : 1。你可以把待测试的像素取出来，测试其透明程度，并据此判断用户是否点击了视图中不透明的部分。

### 解决方案 1-6 根据位图像素的不透明程度来测试触摸

```

// Return the offset for the alpha pixel at (x,y) for RGBA
// 4-bytes-per-pixel bitmap data
static NSUInteger alphaOffset(NSUInteger x, NSUInteger y, NSUInteger w)
{return y * w * 4 + x * 4;}

```

```

// Return the bitmap from a provided image
NSData *getBitmapFromImage(UIImage *image)
{
    if (!sourceImage) return nil;

    // Establish color space
    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    if (colorSpace == NULL) {
        NSLog(@"Error creating RGB color space");
        return nil;
    }

    // Establish context
    int width = sourceImage.size.width;
    int height = sourceImage.size.height;
    CGContextRef context =
        CGContextCreate(NULL, width, height, 8,
            width * 4, colorSpace,
            (CGBitmapInfo) kCGImageAlphaPremultipliedFirst);
    CGColorSpaceRelease(colorSpace);
    if (context == NULL) {
        NSLog(@"Error creating context");
        return nil;
    }

    // Draw source into context bytes
    CGRect rect = (CGRect){.size = sourceImage.size};
    CGContextDrawImage(context, rect, sourceImage.CGImage);

    // Create NSData from bytes
    NSData *data =
        [NSData dataWithBytes:CGBitmapContextGetData(context)
            length:(width * height * 4)];
    CGContextRelease(context);

    return data;
}

// Store the bitmap data into an NSData instance variable
- (instancetype)initWithImage:(UIImage *)anImage
{
    self = [super initWithImage:anImage];
    if (self)
    {
        self.userInteractionEnabled = YES;
        data = getBitmapFromImage(anImage);
    }
    return self;
}

```



```
// Does the point hit the view?
- (BOOL)pointInside:(CGPoint)point withEvent:(UIEvent *)event
{
    if (!CGRectContainsPoint(self.bounds, point)) return NO;
    Byte *bytes = (Byte *)data.bytes;
    uint offset = alphaOffset(point.x, point.y, self.image.size.width);
    return (bytes[offset] > 85);
}
```

本例所使用的分界值是 85。就是说，受测像素的不透明程度至少是 33%（也就是大约 85/255），我们才能认定用户点击了这个视图。我们所编写的 `pointInside:` 方法会将不透明度低于 33% 的像素视为透明。这个值是笔者随意选取的。你可以根据自己 GUI 的实际需求来调整该值（当然也可以换用另外的判定算法）。



**提示** 除非你要实现完美的像素级触摸检测，否则可以先把位图缩小，然后再用修改过的算式去判断，这样占用的内存会少一些。

## 1.8 解决方案：根据触摸情况在屏幕上绘制内容

开发者可以在 `UIView` 的范围内实现直接屏幕绘制（direct onscreen drawing）。它的 `drawRect:` 方法提供了一种低阶的方式，令我们能够使用 Quartz 2D API 来创建及显示任意元件，并直接向屏幕中绘制内容。将触摸与绘制结合起来，就能构建出既直观而又容易操控的界面。

解决方案 1-7 把手势与 `drawRect` 相结合，实现出了基于触摸的绘画功能。用户触摸屏幕时，`TouchTrackerView` 类会随着用户的手指而构建贝塞尔曲线。为了在用户触摸屏幕的过程中实现绘画功能，我们令程序的 `touchesMoved:withEvent:` 方法调用 `setNeedsDisplay`，而 `setNeedsDisplay` 又会触发 `drawRect:`，后者将根据收集到的贝塞尔曲线信息在视图中绘制相关图形。图 1-3 中的界面演示了用此方式创建出的一条路径。

### 解决方案 1-7 在 `UIView` 中实现基于触摸的绘画功能

```
@implementation TouchTrackerView
{
    UIBezierPath * path;
}

- (instancetype)initWithFrame:(CGRect) frame
{
    self = [super initWithFrame:frame];
    if (self)
    {
        self.multipleTouchEnabled = NO;
    }
}
```

```

    return self;
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Initialize a new path for the user gesture
    path = [UIBezierPath bezierPath];
    path.lineWidth = IS_IPAD ? 8.0f : 4.0f;

    UITouch *touch = [touches anyObject];
    [path moveToPoint:[touch locationInView:self]];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Add new points to the path
    UITouch *touch = [touches anyObject];
    [self.path addLineToPoint:[touch locationInView:self]];
    [self setNeedsDisplay];
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    [path addLineToPoint:[touch locationInView:self]];
    [self setNeedsDisplay];
}

- (void)touchesCancelled:(NSSet *)touches
    withEvent:(UIEvent *)event
{
    [self touchesEnded:touches withEvent:event];
}

- (void)drawRect:(CGRect)rect
{
    // Draw the path
    [path stroke];
}

@end

```

虽说我们可以用手势识别器来改写本例，但这样做没有实际意义。本程序中的触摸信息并没有太大的用途，我们只不过是用其来创建一条平滑的轨迹。基本的响应者方法（也就是 `touchesBegan`、`touchesMoved` 等方法）完全可以应付路径的创建及管理事宜。

本例创建出来的是连续轨迹。该程序并不响应静止的触摸事件。假如读者想扩充这个范例程序，使其具备画点或画符号的能力，需要自己去添加相关的行为代码。

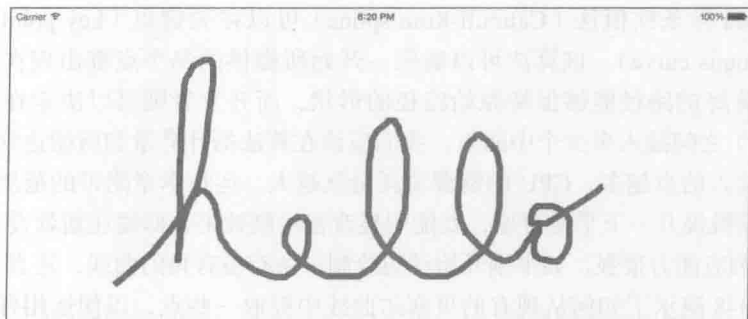


图 1-3 一套简单的 iOS 绘画工具，其所使用的技术只不过就是沿着路径收集触摸信息，并调用 UIKit/Quartz 2D API 将其绘制出来

## 1.9 解决方案：令绘制效果变得平滑

用户所使用的设备各有不同，其设备在同一时间所能处理的运算量也不一样，这就导致捕捉到的触摸事件可能要比预期的粗略一些。与生活中轮番握手时的情况类似，触摸事件的采样率通常受限于 CPU 的能力。我们可以在点之间进行插值（interpolating），用平滑算法（smoothing algorithm）来缓解这些限制。通过图 1-4 我们可以看到，用采集到的触摸点直接绘制出来的曲线与经过平滑算法处理过的曲线其圆润程度是不同的。

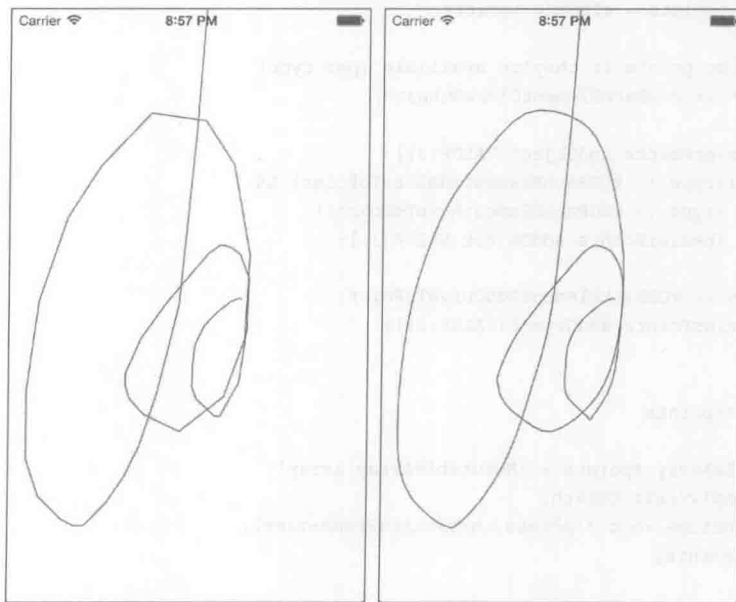


图 1-4 程序可以实时地运用 Catmull-Rom 平滑算法，使连接触摸事件发生点的弧可以变得柔和一些。两张截图都是根据同一套手势绘制的，右侧是运用了平滑算法之后所绘制出来的效果，而左侧则是没有运用平滑算法时所绘制出来的效果

Catmull-Rom 样条插值法 (Catmull-Rom spline) 可以在关键点 (key point) 之间创建连续曲线 (continuous curve)。该算法可以确保一开始所提供的每个点都出现在最后绘制好的曲线上。运算好的路径能够保持原始路径的形状, 而开发者则可以决定在每一对参考点 (reference point) 之间插入多少个中间点。我们应该在算法的计算量和所能达到的平滑程度之间进行权衡。加入的点越多, CPU 的资源消耗量就越大。运行本章附带的范例代码之后你就会发现, 只要稍微提升一下平滑程度, 就能明显改善绘制效果, 即便在新款设备上也是如此。最新版 iPad 的响应能力很强, 如果刚开始就想绘制一条有棱有角的曲线, 还真不是那么容易。

解决方案 1-8 演示了如何从现有的贝塞尔曲线中提取一些点, 以便运用样条插值法创建出平滑的效果。Catmull-Rom 算法每次使用四个点计算第二点和第三点之间的中间值<sup>①</sup>, 而开发者可以通过 granularity 参数来指定两个点之间到底应该插入多少个中间点。

### 解决方案 1-8 用 Catmull-Rom 样条插值法创建平滑的贝塞尔曲线

```
#define VALUE(_INDEX_) [NSValue valueWithCGPoint:points[_INDEX_]]

@implementation UIBezierPath (Points)
void getPointsFromBezier(void *info, const CGPathElement *element)
{
    NSMutableArray *bezierPoints = (__bridge NSMutableArray *)info;

    // Retrieve the path element type and its points
    CGPathElementType type = element->type;
    CGPoint *points = element->points;

    // Add the points if they're available (per type)
    if (type != kCGPathElementCloseSubpath)
    {
        [bezierPoints addObject:VALUE(0)];
        if ((type != kCGPathElementAddLineToPoint) &&
            (type != kCGPathElementMoveToPoint))
            [bezierPoints addObject:VALUE(1)];
    }
    if (type == kCGPathElementAddCurveToPoint)
        [bezierPoints addObject:VALUE(2)];
}

- (NSArray *)points
{
    NSMutableArray *points = [NSMutableArray array];
    CGPathApply(self.CGPath,
        (__bridge void *)points, getPointsFromBezier);
    return points;
}

@end
```

① 这里的第二点和第三点是 Catmull-Rom 样条插值法所使用的特殊称谓, 对应于代码中的 p1 和 p2。——译者注

```

#define POINT(_INDEX_) \
    [(NSValue *) [points objectAtIndex:_INDEX_] CGPointValue]

@implementation UIBezierPath (Smoothing)
- (UIBezierPath *)smoothedPath:(int)granularity
{
    NSMutableArray *points = [self.points mutableCopy];
    if (points.count < 4) return [self copy];

    // Add control points to make the math make sense
    // Via Josh Weinberg
    [points insertObject:[points objectAtIndex:0] atIndex:0];
    [points addObject:[points lastObject]];

    UIBezierPath *smoothedPath = [UIBezierPath bezierPath];

    // Copy traits
    smoothedPath.lineWidth = self.lineWidth;

    // Draw out the first 3 points (0..2)
    [smoothedPath moveToPoint:POINT(0)];

    for (int index = 1; index < 3; index++)
        [smoothedPath addLineToPoint:POINT(index)];

    for (int index = 4; index < points.count; index++)
    {
        CGPoint p0 = POINT(index - 3);
        CGPoint p1 = POINT(index - 2);
        CGPoint p2 = POINT(index - 1);
        CGPoint p3 = POINT(index);

        // now add n points starting at p1 + dx/dy up
        // until p2 using Catmull-Rom splines
        for (int i = 1; i < granularity; i++)
        {
            float t = (float) i * (1.0f / (float) granularity);
            float tt = t * t;
            float ttt = tt * t;

            CGPoint pi; // intermediate point
            pi.x = 0.5 * (2*p1.x+(p2.x-p0.x)*t +
                (2*p0.x-5*p1.x+4*p2.x-p3.x)*tt +
                (3*p1.x-p0.x-3*p2.x+p3.x)*ttt);
            pi.y = 0.5 * (2*p1.y+(p2.y-p0.y)*t +
                (2*p0.y-5*p1.y+4*p2.y-p3.y)*tt +
                (3*p1.y-p0.y-3*p2.y+p3.y)*ttt);
            [smoothedPath addLineToPoint:pi];
        }

        // Now add p2
    }
}

```

```

        [smoothedPath addLineToPoint:p2];
    }

    // finish by adding the last point
    [smoothedPath addLineToPoint:POINT(points.count - 1)];

    return smoothedPath;
}
@end

// Example usage:
// Replace the path with a smoothed version after drawing completes
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    [path addLineToPoint:[touch locationInView:self]];
    path = [path smoothedPath:4];
    [self setNeedsDisplay];
}

```

解决方案 1-8 只不过演示了一种实时几何处理算法，实际上，计算几何学里面还有很多算法都能以类似的方式运用到应用程序中。



**提示** Erica Sadun 所著的《*iOS Drawing: Practical UIKit Solutions*》(Addison-Wesley, 2013 年)一书里还讲了很多与 `getPointsFromBezier` 相似的 `UIBezierPath` 工具。若想了解更多优秀的图形算法，请访问 [www.graphicsgems.org](http://www.graphicsgems.org) 网站，查看由 Academic Press 所出版的《*Graphics Gems*》系列图书，其中也讲了很多先进的平滑算法。

## 1.10 解决方案：启用多点触摸

在 `UIView` 实例中启用了多点触摸模式的交互之后，当用户以几根手指同时触摸屏幕时，iOS 就可以侦测到这些触摸操作并对其进行响应。把 `UIView` 的 `multipleTouchEnabled` 属性设为 `YES`，或在自己的视图类中覆写 `isMultipleTouchEnabled` 方法，即可启用多点触摸。启用之后，系统在传给触摸回调方法的触摸事件里面，就会放入多个触摸对象。在触摸事件的设置中，如果有多于一个的元素，就说明需要处理多点触摸。

从理论上来说，iOS 支持任意多个点的同时触摸。你可以在 iPad 上面运行解决方案 1-9，尽可能用多根手指同时触摸屏幕，看看它的上限是多少。实际的上限以后可能会有变化，所以笔者不打算在这个解决方案里面给出具体的个数。

当年 iPhone 首次支持多点触摸的时候，开发者们还没有预见到把多点触摸同多用户操作结合起来其实可以实现出一些自由而灵活的功能。将多点触摸添加到游戏及其他应用程序里面，不仅可以扩大手势的种类，而且还能创造出新颖而优秀的多用户操作体验，在配有大屏幕的 iPad 上面更是如此。只要多点触摸符合应用程序的需求，并且对程序有帮助，就可以考

虑引入该特性。

多点触摸并不会按照用户的手来分组。比方说，当用户左右两手各用一根手指触摸屏幕时，系统无法判断出某个触摸点对应于哪只手。而且触摸对象的顺序也是随意排列的。就单个的触摸事件来说，在按下手指、移动直至松开的过程中，触摸对象与手指之间的对应次序保持不变（说得更具体些，同一个触摸对象在内存中的地址保持不变），虽说如此，但是用户下次触摸屏幕时，触摸对象与手指之间不一定还能保持这套对应关系。如果需要把这些触摸对象分辨清楚，那么可以像本条解决方案这样，构建一份以触摸对象为索引的字典。

你要是知道 iOS 还支持多于两个手指的多点触摸，是不是会觉得这个功能还不错呢？实际上，如果一次用三根或三根以上的手指来触摸屏幕，那么系统很有可能就会漏掉某些手指的触摸操作。假如用两根以上的手指来操作程序，那么难以以编程的方式追踪到平滑的手势。因此，在处理多点触摸这种操作方式时，不要执著于对手势的解析，而是应该把它理解成一系列在限定的时间内发生且各自独立的交互操作。你应该把每一个触摸对象都当成各自独立的東西来看待，并分别处理它们。

解决方案 1-9 通过设定 `multipleTouchEnabled` 属性来为 `UIView` 添加多点触摸功能，并记录下每根手指所画的线条。该程序遵照苹果公司的开发建议，记录下每个触摸对象在内存中的物理地址，并且不使用指向这些触摸对象的指针，也不对其做保留（retain）。

#### 解决方案 1-9 把用户所画的各条线收集起来，以实现叠加式绘图

```
@interface TouchTrackerView : UIView
- (void) clear;
@end

@implementation TouchTrackerView
{
    NSMutableArray *strokes;
    NSMutableDictionary *touchPaths;
}

// Establish new views with storage initialized for drawing
- (instancetype) initWithFrame: (CGRect) frame
{
    self = [super initWithFrame: frame];
    if (self)
    {
        self.multipleTouchEnabled = YES;
        strokes = [NSMutableArray array];
        touchPaths = [NSMutableDictionary dictionary];
    }
    return self;
}

// On clear, remove all existing strokes, but not in-progress drawing
- (void) clear
{
    [strokes removeAllObjects];
}
```

```

        [self setNeedsDisplay];
    }

    // Start touches by adding new paths to the touchPath dictionary
    - (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
    {
        for (UITouch *touch in touches)
        {
            NSString *key = [NSString stringWithFormat:@"%d", (int) touch];
            CGPoint pt = [touch locationInView:self];

            UIBezierPath *path = [UIBezierPath bezierPath];
            path.lineWidth = IS_IPAD ? 8 : 4;
            path.lineCapStyle = kCGLineCapRound;
            [path moveToPoint:pt];

            touchPaths[key] = path;
        }
    }

    // Trace touch movement by growing and stroking the path
    - (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
    {
        for (UITouch *touch in touches)
        {
            NSString *key =
                [NSString stringWithFormat:@"%d", (int) touch];
            UIBezierPath *path = [touchPaths objectForKey:key];
            if (!path) break;

            CGPoint pt = [touch locationInView:self];
            [path addLineToPoint:pt];
        }
        [self setNeedsDisplay];
    }

    // On ending a touch, move the path to the strokes array
    - (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
    {
        for (UITouch *touch in touches)
        {
            NSString *key = [NSString stringWithFormat:@"%d", (int) touch];
            UIBezierPath *path = [touchPaths objectForKey:key];
            if (path) [strokes addObject:path];
            [touchPaths removeObjectForKey:key];
        }
        [self setNeedsDisplay];
    }

    - (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
    {
        [self touchesEnded:touches withEvent:event];
    }

```



```
// Draw existing strokes in dark purple, in-progress ones in light
- (void)drawRect:(CGRect)rect
{
    [COOKBOOK_PURPLE_COLOR set];
    for (UIBezierPath *path in strokes)
    {
        [path stroke];
    }

    [[COOKBOOK_PURPLE_COLOR colorWithAlphaComponent:0.5f] set];
    for (UIBezierPath *path in [touchPaths allValues])
    {
        [path stroke];
    }
}
@end
```

本例所采用的做法显然是比较奇怪的，不过这套办法在各个版本的 SDK 中都能照常运作。这是因为在触摸 - 移动 - 释放这个生命期中，每个 UITouch 对象都驻留在其各自的内存地址处。苹果公司并不建议对 UITouch 实例做保留，所以，在本例中，我们把这些触摸对象转为整数值，并用这些整数值来作为字典的键。

请注意，每当发生新的触摸操作时，系统就会调用 touchesBegan:withEvent: 来开启一段新的生命期，这与目前仍在进行中的触摸操作是互不干扰的，这些操作依然可以进入其各自的移动、结束及取消等阶段。开发者在编写代码时应该考虑到这一点。

这条解决方案扩充了原有的解决方案 1-7。每次触摸操作都能独自产生一条贝塞尔路径，而视图的 drawRect 方法则会把这些路径绘制出来。解决方案 1-7 在每次触摸操作的生命期结束之后，如果发现有新的触摸操作，那么就会清除原有的屏幕内容，并开始绘制新的曲线。这对于一款仅用于演示的应用程序来说，是合适的，但若想创建一款标准的绘图程序，就不能这么做了，而是应该把新绘制的内容叠放到屏幕里原有的内容之上。

解决方案 1-9 不会擦除原有的内容，新绘制的线条会不断地放在现有的画面中。该程序会把触摸信息收集到一个可以持续增长的可变数组里面，并可以根据用户的需要来清空此数组。这个解决方案采用稍微淡一些的颜色来绘制用户正在画的线，以便与 strokes 数组中已经画好的路径相区别。



**提示** 在苹果公司的开发者网站上，可以找到很多 Core Graphics/Quartz 2D 资源。另外还有许多论坛、邮件列表及源代码范例，它们虽然不专门针对 iOS，但也都是非常有价值的资源，你可以由此来扩充自己的 iOS Core Graphics 知识。

## 1.11 解决方案：检测圆圈手势

对于 iOS 这种直接操纵界面来说，开发者可能会觉得大部分用户都只是简单地点击屏幕

上的物件。但是，有非常多的人希望程序能够支持圆圈手势，开发者实现了这种手势之后，用户就可以拿手指圈选屏幕上的物件了。有读者希望本书给出相关的解决方案，于是，笔者编写了解决方案 1-10，实现了一个相对简单的圆圈手势检测器，其效果如图 1-5 所示。

在实现该程序的时候，笔者采用了几个测试步骤。首先做时间测试（time test），以确保手势不能执行得太慢。圆圈手势应该是一种必须快速画出的手势。接下来做弯曲测试（inflection test），以确保方向变化不能太过频繁。一般来说，圆圈手势的方向改变次数是四次，本程序最多允许在画圆过程中变五次方向。第三种测试是覆盖度测试（convergence test），圆的起点与终点必须足够接近，这两个点必须要有某种程度的联系。由于程序并不会向用户提供直接的视觉反馈，所以用户可能会画得稍微偏一点，于是，我们应该在检测时留有一些余地。本程序所能容忍的像素偏差比较大，这个距离大约是视图尺寸的 1/3。

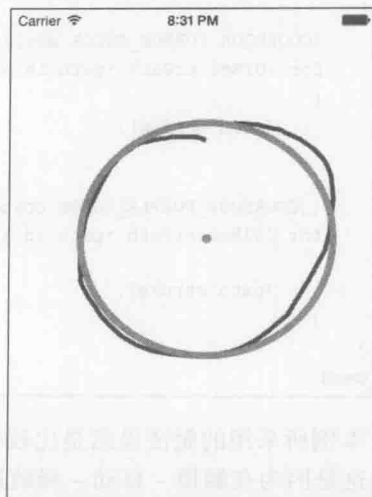


图 1-5 如果检测到了圆圈手势，那么程序会用圆点及红圈来描述该手势的关键特征

#### 解决方案 1-10 检测圆圈手势

```
// Retrieve center of rectangle
CGPoint GEORectGetCenter(CGRect rect)
{
    return CGPointMake(CGRectGetMidX(rect), CGRectGetMidY(rect));
}

// Build rectangle around a given center
CGRect GEORectAroundCenter(CGPoint center, float dx, float dy)
{
    return CGRectMake(center.x - dx, center.y - dy, dx * 2, dy * 2);
}

// Center one rect inside another
CGRect GEORectCenteredInRect(CGRect rect, CGRect mainRect)
{
    CGFloat dx = CGRectGetMidX(mainRect) - CGRectGetMidX(rect);
    CGFloat dy = CGRectGetMidY(mainRect) - CGRectGetMidY(rect);
    return CGRectOffset(rect, dx, dy);
}

// Return dot product of two vectors normalized
CGFloat dotproduct(CGPoint v1, CGPoint v2)
{
    CGFloat dot = (v1.x * v2.x) + (v1.y * v2.y);
}
```

```

CGFloat a = ABS(sqrt(v1.x * v1.x + v1.y * v1.y));
CGFloat b = ABS(sqrt(v2.x * v2.x + v2.y * v2.y));
dot /= (a * b);

return dot;
}

// Return distance between two points
CGFloat distance(CGPoint p1, CGPoint p2)
{
    CGFloat dx = p2.x - p1.x;
    CGFloat dy = p2.y - p1.y;

    return sqrt(dx*dx + dy*dy);
}

// Offset in X
CGFloat dx(CGPoint p1, CGPoint p2)
{
    return p2.x - p1.x;
}

// Offset in Y
CGFloat dy(CGPoint p1, CGPoint p2)
{
    return p2.y - p1.y;
}

// Sign of a number
NSInteger sign(CGFloat x)
{
    return (x < 0.0f) ? (-1) : 1;
}

// Return a point with respect to a given origin
CGPoint pointWithOrigin(CGPoint pt, CGPoint origin)
{
    return CGPointMake(pt.x - origin.x, pt.y - origin.y);
}

// Calculate and return least bounding rectangle
#define POINT(_INDEX_) [(NSValue *)[points \
    objectAtIndex:_INDEX_] CGPointValue]

CGRect boundingRect(NSArray *points)
{
    CGRect rect = CGRectZero;
    CGRect ptRect;

    for (NSUInteger i = 0; i < points.count; i++)
    {
        CGPoint pt = POINT(i);

```

```

        ptRect = CGRectMake(pt.x, pt.y, 0.0f, 0.0f);
        rect = (CGRectEqualToRect(rect, CGRectZero)) ?
            ptRect : CGRectUnion(rect, ptRect);
    }
    return rect;
}

CGRect testForCircle(NSArray *points, NSDate *firstTouchDate)
{
    if (points.count < 2)
    {
        NSLog(@"Too few points (2) for circle");
        return CGRectZero;
    }

    // Test 1: duration tolerance
    float duration = [[NSDate date]
        timeIntervalSinceDate:firstTouchDate];
    NSLog(@"Transit duration: %0.2f", duration);

    float maxDuration = 2.0f;
    if (duration > maxDuration)
    {
        NSLog(@"Excessive duration");
        return CGRectZero;
    }

    // Test 2: Direction changes should be limited to near 4
    int inflections = 0;
    for (int i = 2; i < (points.count - 1); i++)
    {
        float deltax = dx(POINT(i), POINT(i-1));
        float delty = dy(POINT(i), POINT(i-1));
        float px = dx(POINT(i-1), POINT(i-2));
        float py = dy(POINT(i-1), POINT(i-2));

        if ((sign(deltax) != sign(px)) ||
            (sign(delty) != sign(py)))
            inflections++;
    }

    if (inflections > 5)
    {
        NSLog(@"Excessive inflections");
        return CGRectZero;
    }

    // Test 3: Start and end points near each other
    float tolerance = [[[UIApplication sharedApplication]
        keyWindow] bounds].size.width / 3.0f;
    if (distance(POINT(0), POINT(points.count - 1)) > tolerance)

```

```

{
    NSLog(@"Start too far from end");
    return CGRectZero;
}

// Test 4: Count the distance traveled in degrees
CGRect circle = boundingRect(points);
CGPoint center = GEOFrectGetCenter(circle);
float distance = ABS(acos(dotproduct(
    pointWithOrigin(POINT(0), center),
    pointWithOrigin(POINT(1), center))));
for (int i = 1; i < (points.count - 1); i++)
    distance += ABS(acos(dotproduct(
        pointWithOrigin(POINT(i), center),
        pointWithOrigin(POINT(i+1), center))));

float transitTolerance = distance - 2 * M_PI;

if (transitTolerance < 0.0f) // fell short of 2 PI
{
    if (transitTolerance < - (M_PI / 4.0f)) // under 45
    {
        NSLog(@"Transit too short");
        return CGRectZero;
    }

    if (transitTolerance > M_PI) // additional 180 degrees
    {
        NSLog(@"Transit too long ");
        return CGRectZero;
    }

    return circle;
}
@end

```

最后还要执行一种测试，以判断用户的手指围绕手势的中心点旋转了多少度。程序会根据手指划过的弧来统计这个度数，标准的圆应该是 360 度。为了使用户在画圆的时候能把手势做得自然一些，我们扩大了对角度的容忍范围，用户最少可以少画 45 度，最多可以多画 180 度。

如果这些测试全都通过了，那么该算法就生成一个最小外接矩形 (least bounding rectangle)，并对原手势上的各点求几何平均值，然后把矩形中心点与均值点对齐。程序会把检测结果赋给 circle 变量。这套检测系统虽然不完美 (读者可以运行范例代码，并想办法骗过这套检测系统)，但它却能为许多 iOS 应用程序提供足够健壮而且相当好用的圆圈手势检测能力。

## 1.12 解决方案：创建自定义手势识别器

只需稍加修改，就能把解决方案 1-10 中的代码转换成一款可以辨识自定义手势的识别器，解决方案 1-11 实现了这种识别器。从 `UIGestureRecognizer` 中继承子类，即可构建出自己的圆圈手势识别器，然后，可以将其添加到自己的应用程序中。

你应该在自己的新类里面引入 `UIKit` 的 `UIGestureRecognizerSubclass.h`。实现 `UIGestureRecognizer` 的子类时，可能需要覆写或自定义一些方法，而这些方法都声明在 `UIGestureRecognizerSubclass.h` 头文件里。在覆写某个方法时，你应该先调用该方法的原型版本，然后再执行自己的新代码，也就是说，应该先调用超类的同名方法。

手势可以分为两大类：连续手势（continuous gesture）和不连续手势（discrete gesture）。圆圈手势识别器就是不连续的。它要么可以识别出圆圈手势，要么识别不出来。而双指聚拢及拖动则属于连续手势，手势识别器会在手势的整个生命期里面不断发送相关的更新。手势识别器通过设置 `state` 属性来产生更新。

手势识别器其实就是个描述指尖状态的状态机。每个识别器刚开始都处在 `possible` 状态（`UIGestureRecognizerStatePossible`），对于连续手势来说，识别器会历经一系列的 `changed` 状态（`UIGestureRecognizerStateChanged`），而对于不连续手势来说，如果能够识别出这样的手势，那么识别器就会进入 `UIGestureRecognizerStateRecognized` 状态，若识别不出来，则进入 `UIGestureRecognizerStateFailed` 状态，如解决方案 1-11 所示。除非我们将状态（`state`）设为 `possible` 或 `failed`，否则每次更新状态的时候，识别器都会向其目标发送动作消息。

### 解决方案 1-11 创建 `UIGestureRecognizer` 的子类

```
#import <UIKit/UIGestureRecognizerSubclass.h>
@implementation CircleRecognizer

// Called automatically by the runtime after the gesture state has
// been set to UIGestureRecognizerStateEnded. Any internal state
// should be reset to prepare for a new attempt to recognize the gesture.
// After this is received, all remaining active touches will be ignored
// (no further updates will be received for touches that had already
// begun but haven't ended).
- (void)reset
{
    [super reset];

    points = nil;
    firstTouchDate = nil;
    self.state = UIGestureRecognizerStatePossible;
}

// mirror of the touch-delivery methods on UIResponder
// UIGestureRecognizers aren't in the responder chain, but observe
// touches hit-tested to their view and their view's subviews.
```

```

// UIGestureRecognizer receives touches before the view to which
// the touch was hit-tested.
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    [super touchesBegan:touches withEvent:event];

    if (touches.count > 1)
    {
        self.state = UIGestureRecognizerStateFailed;
        return;
    }

    points = [NSMutableArray array];
    firstTouchDate = [NSDate date];
    UITouch *touch = [touches anyObject];
    [points addObject:[NSValue valueWithCGPoint:
        [touch locationInView:self.view]]];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    [super touchesMoved:touches withEvent:event];
    UITouch *touch = [touches anyObject];
    [points addObject:[NSValue valueWithCGPoint:
        [touch locationInView:self.view]]];
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    [super touchesEnded:touches withEvent:event];
    BOOL detectionSuccess = !CGRectEqualToRect(CGRectZero,
        testForCircle(points, firstTouchDate));
    if (detectionSuccess)
        self.state = UIGestureRecognizerStateRecognized;
    else
        self.state = UIGestureRecognizerStateFailed;
}
@end

```

解决方案 1-11 里面有几段相当长的注释，那是笔者从 `UIGestureRecognizerSubclass.h` 头文件里面拷贝过来的，感谢苹果公司撰写这些注释。它们分别解释了几个关键的方法所扮演的角色，我们在 `UIGestureRecognizer` 子类里覆写了这几个方法。`reset` 方法会令识别器回到沉寂（quiescent）状态，以便为识别下一轮手势做准备。

与 `UIResponder` 中的相关方法类似，识别器也会在相应的时机调用 `touchesBegan:withEvent:` 等方法，这使得开发者可以在识别器里面编写代码，在触摸操作生命期中的相关时间点上执行测试。本范例程序会一直等到系统调用 `touchesEnded:withEvent:` 回调方法时，再去判断对手势的识别是成功了还是失败了，它所用的 `testForCircle` 方法与解决方案 1-10 中定义的方法相同。



作为覆写超类方法时所应提倡的一条原则，一旦发现无法辨识出手势，我们就应该令手势识别器尽快“失败”。若是能够辨识出手势，就应当把与手势有关的信息存储在程序的属性里面。圆圈手势识别器应该把检测到的圆圈保存起来，使用户能够知道手势出现在何处。

## 1.13 解决方案：把滚动视图中的内容拖曳到外面

iOS 所提供的手势识别器的功能确实很丰富，但并不总是能够满足开发者的需要。比方说，有个可以水平滚动的视图，里面包含许多相邻的图像视图 `ImageView`，用户可以左右滚动这个大视图来查看其中的全部内容。现在，假设我们要实现一个功能，令用户可以把视图中的 `ImageView` 拖曳到滚动区域下方的空白区域里。要实现此功能，我们需要辨识发生在每个子视图身上的向下触摸操作（`downward touch`，其移动的方向与大视图的滚动方向相垂直）。

开发者 Alex Hosgrove 在构建一款应用程序时，就遇到了这个问题，那款程序里面的东西有点像吸附在冰箱门上的一套磁铁字母。用户可以把字母向下拖放到工作区中，然后可以操作并排列其所选的字母。实现这种功能的时候，要解决两个问题：一是触摸操作属于谁来管，二是识别出了向下触摸之后应该怎样处理。

滚动视图（`Scroll View`）中的子视图都需要关注触摸操作。如果检测到了向下的手势，那么程序就应该产生新对象，若是检测到横向的手势，则应该水平滚动滚动视图中的内容。为了使滚动视图及其子视图都能响应用户的操作，我们应该在程序内部共享触摸信息。经由 `UIGestureRecognizerDelegate` 即可实现这一点。

开发者通过 `UIGestureRecognizerDelegate` 可以实现同步识别，也就是说，两个手势识别器可以同时运作。要想实现这一点，只需令相关的类遵从 `UIGestureRecognizerDelegate` 协议，并向其中添加下面这个简单的委托方法：

```
- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer
    shouldRecognizeSimultaneouslyWithGestureRecognizer:
        (UIGestureRecognizer *)otherGestureRecognizer
{
    return YES;
}
```

由于我们不能重新设置滚动视图的委托，所以必须令滚动视图里面的子视图所对应的类遵循 `UIGestureRecognizerDelegate` 协议，并在其中实现上述方法。

要想解决上面所提的第二个问题，也就是如何令滑动这个动作产生出拖曳效果，需要从整个触摸生命期的角度来考虑。凡是能产生新对象的触摸操作，一开始都是一个垂直方向上的拖曳，但只要新对象创建出来了，它就变成了拖动手势。所以，此处使用拖动手势识别器更为合适，因为假如使用滑动手势识别器的话，那么等识别出来的时候，触摸操作的生命期已经结束了。



为了解决第二个问题，解决方案 1-12 在内置的手势探测代码中实现了对方位移动<sup>①</sup>的检测。从最后的结果来看，这种打破常规的做法收到了实效。这是因为程序检测到滑动之后，底层的拖动手势识别器依然会继续运作。如此一来，用户就可以继续移动刚才拖曳出来的物件，而无须先把手指抬起来，然后再重新触摸它。

解决方案 1-12 把滚动视图中的物件拖曳到外面

```
@implementation DragView

#define DX(p1, p2)    (p2.x - p1.x)
#define DY(p1, p2)    (p2.y - p1.y)

const NSInteger kSwipeDragMin = 16;
const NSInteger kDragLimitMax = 12;

// Categorize swipe types
typedef enum {
    TouchUnknown,
    TouchSwipeLeft,
    TouchSwipeRight,
    TouchSwipeUp,
    TouchSwipeDown,
} SwipeTypes;

@implementation PullView
// Create a new view with an embedded pan gesture recognizer
- (instancetype)initWithImage:(UIImage *)anImage
{
    self = [super initWithImage:anImage];
    if (self)
    {
        self.userInteractionEnabled = YES;
        UIPanGestureRecognizer *pan =
            [[UIPanGestureRecognizer alloc] initWithTarget:self
             action:@selector(handlePan:)];
        pan.delegate = self;
        self.gestureRecognizers = @[pan];
    }

    // Allow simultaneous recognition
    - (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer
    shouldRecognizeSimultaneouslyWithGestureRecognizer:
        (UIGestureRecognizer *)otherGestureRecognizer
    {
        return YES;
    }

    // Handle pans by detecting swipes
    - (void)handlePan:(UISwipeGestureRecognizer *)uigr
    {
        // Only deal with scroll view superviews
```

① directional-movement，也就是上、下、左、右四个方向上的移动。——译者注

```
if (![self.superview isKindOfClass:[UIScrollView class]]) return;
```

```
// Extract superviews
```

```
UIView *supersuper = self.superview.superview;
```

```
UIScrollView *scrollView = (UIScrollView *) self.superview;
```

```
// Calculate location of touch
```

```
CGPoint touchLocation = [uigr locationInView:supersuper];
```

```
// Handle touch based on recognizer state
```

```
if(uigr.state == UIGestureRecognizerStateBegan)
```

```
{
```

```
    // Initialize recognizer
```

```
    gestureWasHandled = NO;
```

```
    pointCount = 1;
```

```
    startPoint = touchLocation;
```

```
}
```

```
if(uigr.state == UIGestureRecognizerStateChanged)
```

```
{
```

```
    pointCount++;
```

```
    // Calculate whether a swipe has occurred
```

```
    float dx = DX(touchLocation, startPoint);
```

```
    float dy = DY(touchLocation, startPoint);
```

```
    BOOL finished = YES;
```

```
    if ((dx > kSwipeDragMin) && (ABS(dy) < kDragLimitMax))
```

```
        touchtype = TouchSwipeLeft;
```

```
    else if ((-dx > kSwipeDragMin) && (ABS(dy) < kDragLimitMax))
```

```
        touchtype = TouchSwipeRight;
```

```
    else if ((dy > kSwipeDragMin) && (ABS(dx) < kDragLimitMax))
```

```
        touchtype = TouchSwipeUp;
```

```
    else if ((-dy > kSwipeDragMin) && (ABS(dx) < kDragLimitMax))
```

```
        touchtype = TouchSwipeDown;
```

```
    else
```

```
        finished = NO;
```

```
    // If unhandled and a downward swipe, produce a new draggable view
```

```
    if (!gestureWasHandled && finished &&
```

```
        (touchtype == TouchSwipeDown))
```

```
{
```

```
    dragView = [[DragView alloc] initWithImage:self.image];
```

```
    dragView.center = touchLocation;
```

```
    [supersuper addSubview: dragView];
```

```
    scrollView.scrollEnabled = NO;
```

```
    gestureWasHandled = YES;
```

```
}
```

```
    else if (gestureWasHandled)
```

```

    {
        // allow continued dragging after detection
        dragView.center = touchLocation;
    }
}

if(uiagr.state == UIGestureRecognizerStateEnded)
{
    // ensure that the scroll view returns to scrollable
    if (gestureWasHandled)
        scrollView.scrollEnabled = YES;
}
}
@end

```

解决方案 1-12 中的实现代码在认定滑动操作时，所用的标准是垂直方向至少扫过 16 个像素，而且左右两侧的偏离不能超过 12 个像素。如果代码检测到了这种向下的滑动，那么它就把新建的 DragView 对象（本章前面曾经用过 DragView 类）添加到屏幕中，使该对象可以随着用户的触摸而完成接下来的拖动手势交互过程。

一旦识别出向下的滑动操作，PullView 类就把自己的 gestureWasHandled 标注成 TRUE，意思是说，自己已经把这个滑动处理过了，同时，它还会在继续处理本次拖动事件的过程中禁用 ScrollView。这样的话，用户所拖曳的子视图就可以完全掌控当前拖动手势的交互过程，而 Scroll View 也就不用再处理接下来的触摸移动了。

## 1.14 解决方案：实时的触摸反馈

你有没有给 iOS 应用程序录制过 demo 呢？如果要录制的话，总会遇到个两难的问题。一种办法是对着手机屏幕来录制，这样做的缺点是可能会把手机屏幕上反射的影像录进去，而且用户的手还可能会挡住屏幕。另一种做法是使用 Reflection (<http://reflectionapp.com>) 之类的工具，但是这些工具只能把直接发生在 iOS 设备屏幕上的内容录下来，没有办法录下用户触摸应用程序的情况，也没办法专门对某个部分进行特写。

解决方案 1-13 提供了一套简单的类（它们统称为 TOUCHkit），使程序可以具备实时的触摸反馈层，以供开发者向他人演示该程序的用法。有了这项功能之后，你既可以看到要录制的屏幕，也可以看到引发互动效果的触摸操作，而那些互动效果正是你想要展示给大家看的。TOUCHkit 提供了一种方式，使开发者可以编译出两个版本的程序，一种供普通场合使用，另一种供演示用。无论制作哪个版本，都不用改变核心的应用程序。你只需简单切换一下，就能构建出这两种不同用途的版本。

为了说明 TOUCHkit 的用法，笔者找了一款由苹果公司所制作的标准演示程序，并把它的范例代码连同本节的范例代码一并放在解决方案 1-13 之中。学会这套工具包的用法之后，你基本上就可以把它运用到各种标准的应用程序上面了。

### 1.14.1 启用触摸反馈效果

若想为现有程序添加触摸反馈效果，只需切换 TOUCHkit 中的相关特性，这不会影响到普通的应用程序代码。设定好相关标志之后，就可以编译并构建应用程序了，用户在触摸这种程序时，屏幕上会出现叠加效果，开发者可以拿这种程序做演示用。部署到 App Store 的时候，则应该将该标志禁用。禁用之后，应用程序的行为就恢复正常了，而且开发者也无需担心程序会执行 App Store 所认定的不安全调用：

```
#define USES_TOUCHkit 1
```

本条解决方案假定开发者所构建的程序是只有一个主窗口的标准应用程序。编译的时候，TOUCHkit 会用一个自定义的类来取代窗口类，而自定义的类可以捕获并复制所有的触摸操作，这就使得应用程序能够用气泡标志来表示用户的触摸点。

另外，开发者还需要做一次非常重要的代码修改操作，不过所涉及的代码量非常少。在应用程序委托类中，需要定义 WINDOW\_CLASS，构建 iOS 应用程序的窗口时会用到它。

```
#if USES_TOUCHkit
#import "TOUCHkitView.h"
#import "TOUCHOverlayWindow.h"
#define WINDOW_CLASS TOUCHOverlayWindow
#else
#define WINDOW_CLASS UIWindow
#endif
```

定义好 WINDOW\_CLASS 之后，我们不直接使用 UIWindow，而是根据 WINDOW\_CLASS 来决定具体使用哪个类：

```
WINDOW_CLASS *window;
window = [[WINDOW_CLASS alloc]
    initWithFrame:[UIScreen mainScreen] bounds]];
```

现在，你就可以像往常那样设置窗口的 rootViewController 属性了。

### 1.14.2 拦截并转发触摸事件

TOUCHkit 之所以能在屏幕上实现触摸反馈效果，是因为它拦截了触摸事件，并据此在应用程序通常的界面上方创建了叠加图样，然后又把事件转发给了应用程序。TOUCHkit 的视图位于程序原来的界面之上，而自定义的窗口类则可以抓取用户的触摸事件，并将其以圆圈的形式展示到 TOUCHkit 的视图上面。然后，它会把事件转发给程序，这样看上去就好像是用户直接在和普通的 UIWindow 交互一样。本条解决方案将使用事件转发来实现这一点。

事件转发是通过调用另外一个事件处理程序来完成的。TOUCHOverlayWindow 类覆写了 UIWindow 的 sendEvent: 方法，以便把触摸效果绘制到屏幕上面，然后，该方法会调用超类的同名方法，以便将控制权交还给普通的响应者链。

下面的实现代码是根据苹果公司的《Event Handling Guide for iOS》而编写的。它会把与当前事件有关的全部 UITouch 都收集起来，这样一来，无论是多点触摸还是单点触摸，

我们都能够应对。然后，该方法会将其派发给 TOUCHkit 的 TOUCHkitView，最后，它调用通常的 UIWindow sendEvent：实现代码，将事件转给视窗。

```
@implementation TOUCHOverlayWindow
- (void)sendEvent:(UIEvent *)event
{
    // Collect touches
    NSSet *touches = [event allTouches];
    NSMutableSet *began = nil;
    NSMutableSet *moved = nil;
    NSMutableSet *ended = nil;
    NSMutableSet *cancelled = nil;

    // Sort the touches by phase for event dispatch
    for(UITouch *touch in touches) {
        switch ([touch phase]) {
            case UITouchPhaseBegan:
                if (!began) began = [NSMutableSet set];
                [began addObject:touch];
                break;
            case UITouchPhaseMoved:
                if (!moved) moved = [NSMutableSet set];
                [moved addObject:touch];
                break;
            case UITouchPhaseEnded:
                if (!ended) ended = [NSMutableSet set];
                [ended addObject:touch];
                break;
            case UITouchPhaseCancelled:
                if (!cancelled) cancelled = [NSMutableSet set];
                [cancelled addObject:touch];
                break;
            default:
                break;
        }
    }

    // Create pseudo-event dispatch
    if (began)
        [[TOUCHkitView sharedInstance]
         touchesBegan:began withEvent:event];
    if (moved)
        [[TOUCHkitView sharedInstance]
         touchesMoved:moved withEvent:event];
    if (ended)
        [[TOUCHkitView sharedInstance]
         touchesEnded:ended withEvent:event];
    if (cancelled)
        [[TOUCHkitView sharedInstance]
         touchesCancelled:cancelled withEvent:event];
}
```

```

    // Call normal handler for default responder chain
    [super sendEvent: event];
}
@end

```

### 1.14.3 实现 TOUCHkit 的 TOUCHkitView 类

TOUCHkit 的 TOUCHkitView 是个简单而清晰的 UIView 单例。当应用程序首次请求访问该类的共享实例时，它才会创建 sharedInstance，创建的时候，该类会把 sharedInstance 添加到程序的 key window 之中。由于 TOUCHkitView 的 userInteractionEnabled 标志是 NO，所以即便我们通过标准的 touchesBegan、touchesMoved、touchesEnded 及 touchesCancelled 等事件回调方法处理了这些事件，它们也依然能够继续向后传播，并到达 TOUCHkitView 下方的界面里，使得系统可以将其纳入响应者链。

处理触摸事件的方法会在每个触摸点上绘制一个圆圈，并创建一个指向 theTouches 的强指针（strong pointer），待绘制完成之后，再清空该指针。解决方案 1-13 详细介绍了处理该功能的回调和绘制方法。

#### 解决方案 1-13 创建 TOUCHkitView 类，以绘制触摸反馈效果

---

```

@implementation TOUCHkitView
{
    NSMutableSet *touches;
    UIImage *fingers;
}

+ (instancetype)sharedInstance
{
    // Create shared instance if it does not yet exist
    if(!sharedInstance)
    {
        sharedInstance = [[self alloc] initWithFrame:CGRectZero];
    }

    // Parent it to the key window
    if (!sharedInstance.superview)
    {
        UIWindow *keyWindow = [UIApplication sharedApplication].keyWindow;
        sharedInstance.frame = keyWindow.bounds;
        [keyWindow addSubview:sharedInstance];
    }

    return sharedInstance;
}

// You can override the default touchColor if you want
- (instancetype)initWithFrame:(CGRect)frame

```

```

{
    self = [super initWithFrame:frame];
    if (self)
    {
        self.backgroundColor = [UIColor clearColor];
        self.userInteractionEnabled = NO;
        self.multipleTouchEnabled = YES;
        touchColor =
            [[UIColor whiteColor] colorWithAlphaComponent:0.5f];
        touches = nil;
    }
    return self;
}

// Basic touches processing
- (void)touchesBegan:(NSSet *)theTouches withEvent:(UIEvent *)event
{
    touches = theTouches;
    [self setNeedsDisplay];
}

- (void)touchesMoved:(NSSet *)theTouches withEvent:(UIEvent *)event
{
    touches = theTouches;
    [self setNeedsDisplay];
}

- (void)touchesEnded:(NSSet *)theTouches withEvent:(UIEvent *)event
{
    touches = nil;
    [self setNeedsDisplay];
}

// Draw touches interactively
- (void)drawRect:(CGRect)rect
{
    // Clear
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGContextClearRect(context, self.bounds);

    // Fill see-through
    [[UIColor clearColor] set];
    CGContextFillRect(context, self.bounds);

    float size = 25.0f; // based on 44.0f standard touch point
    for (UITouch *touch in touches)
    {
        // Create a backing frame
        [[[UIColor darkGrayColor] colorWithAlphaComponent:0.5f] set];
        CGPoint aPoint = [touch locationInView:self];
        CGContextAddEllipseInRect(context,

```

```

        CGRectMake(aPoint.x - size, aPoint.y - size, 2 * size, 2 * size));
CGContextFillPath(context);

// Draw the foreground touch
float dsize = 1.0f;
[touchColor set];
aPoint = [touch locationInView:self];
CGContextAddEllipseInRect(context,
    CGRectMake(aPoint.x - size - dsize, aPoint.y - size - dsize,
        2 * (size - dsize), 2 * (size - dsize)));
CGContextFillPath(context);
}

// Reset touches after use
touches = nil;
}

```

## 1.15 解决方案：向视图中添加菜单

UIMenuController 类使得开发者可以向身为第一响应者的任何物件之中添加弹出式菜单。一般来说，菜单是与文本视图（Text View）及文本框（Text Field）结合起来使用的，它使得用户能够执行选取、拷贝及粘贴等操作。另外，开发者也可通过菜单来提供与互动式屏幕元件有关的操作，例如，可以像本章这样，通过菜单来操作程序里的小 DragView。图 1-6 演示了一套自定义的菜单。在解决方案 1-14 中，如果用户长按某一朵花，程序就会展示出一组菜单。用户可以经由其中的菜单项对当前的 DragView 执行缩放、旋转或隐藏等操作。

本条解决方案将要演示如何获取共享的 UIMenuController，以及如何向其中添加菜单项。开发者需要设置菜单的目标矩形（一般来说，需要传入自己的 bounds 以及展示该菜单的视图），并调整菜单的箭头方向，然后调用菜单的 update 方法，使我们对菜单所做的修改能够生效。执行完这些操作之后，就可以把菜单设为可见状态了。

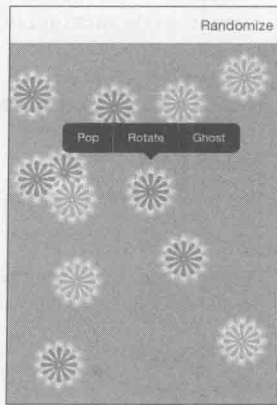


图 1-6 开发者可以通过弹出式关联菜单给充当第一响应者的视图添加交互操作

菜单项也有其标准的目标 - 动作回调机制，但我们并不需要直接设置目标。目标总是身为第一响应者的视图。本条解决方案没有在响应者上面执行 `canPerformAction:withSender:` 检测，不过，若是你的程序里面某些视图可以支持特定的动作，而另外一些视图不能，那么就需要添加这种检测了。是否能支持某项菜单所对应的操作一般和视图的状态有关。比方说，假如视图里面没有内容可供拷贝，那么我们就应该提供拷贝（copy）命令。



### 解决方案 1-14 向互动式的视图中添加菜单

```

- (BOOL)canBecomeFirstResponder
{
    // Menus only work with first responders
    return YES;
}

- (void)pressed:(UILongPressGestureRecognizer *)recognizer
{
    if (![self becomeFirstResponder])
    {
        NSLog(@"Could not become first responder");
        return;
    }

    UIMenuController *menu = [UIMenuController sharedMenuController];
    UIMenuItem *pop = [[UIMenuItem alloc]
        initWithTitle:@"Pop" action:@selector(popSelf)];
    UIMenuItem *rotate = [[UIMenuItem alloc]
        initWithTitle:@"Rotate" action:@selector(rotateSelf)];
    UIMenuItem *ghost = [[UIMenuItem alloc]
        initWithTitle:@"Ghost" action:@selector(ghostSelf)];
    [menu setMenuItems:@[pop, rotate, ghost]];

    [menu setTargetRect:self.bounds inView:self];
    menu.arrowDirection = UIMenuControllerArrowDown;
    [menu update];
    [menu setMenuVisible:YES];
}

- (instancetype)initWithImage:(UIImage *)anImage
{
    self = [super initWithImage:anImage];
    if (self)
    {
        self.userInteractionEnabled = YES;
        UILongPressGestureRecognizer *pressRecognizer =
            [[UILongPressGestureRecognizer alloc] initWithTarget:self
            action:@selector(pressed:)];
        [self addGestureRecognizer:pressRecognizer];
    }
    return self;
}

```

## 1.16 小结

UIView 及其底层的 CALayer 为用户能够在屏幕上看到的各种组件提供了支持。触摸功能使得用户可通过 UITouch 类及手势识别器来直接操作视图。正如本章范例程序所示，

即便在最为基本的形态之下，基于触摸的界面也提供了一套既容易实现又很灵活的接口。读者学会了怎样在屏幕中移动视图，也学会了如何对其移动行为施加限制。我们还讲了怎样通过对触摸操作的检测来判断视图是否应该响应这些操作。此外，大家还知道了如何在视图上面实现绘画功能，以及如何把手势识别器与视图相关联，以解读并响应手势。笔者把本章各个解决方案所体现出来的设计思路总结起来，读者在继续学习下一章之前，可以先思考一下这些问题。

- 程序要直观。iOS 设备的屏幕对触摸支持得非常好。所以，你应该考虑令用户可以来回拖曳屏幕上面的物件，并且使程序能够随着用户手指的移动而画线。这样的话，程序就显得更加真实了，而且也把 iOS 平台的互动性质体现出来了。
- 用户通常是可以十指并用的。iPad 提供了相当强大的屏幕操作能力。所以，不要总是设计只能用一个手指来操作的界面，而是应该在屏幕大小容许的前提下，为程序添加多点触摸式的互动操作。
- 要扎实地掌握 Quartz 图形绘制技术以及 Core Animation API，这对你很有帮助。开发者可以用 `drawRect:` 方法在 `UIView` 里面绘制出各式各样的内容，包括文本、贝塞尔曲线以及涂鸦效果等。
- 如果 Cocoa Touch 所提供的手势识别器不能满足你的特定需求，就自己编写一个。这不是非常困难的事情，你应该尽量把代码写得完备一些，以便将自定义识别器可能历经的各种状态都涵盖在内。
- 尽可能使程序支持多点触摸，如果你想令多位用户能够同时触摸程序的话，就更应该如此了。不要把程序局限在单人单指触摸这种交互方式上面，有时只需多用一点编程技巧，就可以使多位用户同时操作应用程序。
- 多多探索。在应用程序中实现直接操纵是有很多方式的，而本章只讲了有限的几种。请以本章为出发点，探索 `UITouch` 类的其他各种能力。

## 构建并使用控件

UIKit 类是很多 iOS 交互式元件的基础，比如按钮、文本框、滑杆和开关等。这些视图对象都是从同一个祖先类继承下来的，除此以外，它们之间还有许多共同点。所有控件都使用类似的布局范式（layout paradigm）及目标 - 动作触发器。只学会创建一种控件就够了：无论这种控件多么特殊，我们都能由此明白所有控件的工作原理。控件的外观及功能也许各有不同，但设计这些控件时所依循的模式却是相同的。本章要讲解各种控件及其用途。你将学会以多种方式来构建并定制控件。笔者会给出与控件有关的各种解决方案，供你在自己的程序里复用，这些解决方案有的比较简单，有的比较复杂。

### 2.1 UIControl 类

iOS 的控件是程序库中预先构建好的一批对象，它们是为了实现用户交互功能而设计的。这些控件包括按钮、文本框、滑杆、开关以及其他一些由苹果公司所提供的对象。控件所扮演的角色就是把用户的操作转化成回调。用户通过触摸及操纵控件来与应用程序交互。

在控件类的继承树里，UIKit 类是树根。控件类都是 UIView 的子类，它从 UIView 中继承了与显示及布局有关的全部属性。UIView 的子类添加了一套响应机制，该机制可以强化视图，使其能够具备交互性。

当用户操作控件的界面时，每个控件都有办法实现消息派发。控件使用目标 - 动作模式来发送消息。定义新控件的时候，开发者需要指明消息的接收方（也就是目标）以及所发送的消息（也就是动作），还要指定发送这些消息的时机（也就是触发条件，比方说，当用户在某个控件范围内完成触摸操作时，就触发某消息）。

### 2.1.1 目标 – 动作模式

目标 – 动作设计模式提供了一条可以响应用户交互的底层途经。我们基本上只会在 UIControl 类的子类中碰到它。经由目标 – 动作模式，开发者可以在发生了特定的用户事件时，命令控件给指定的对象发送消息。比方说，你可以指明当用户按下按钮或调整滑杆时，哪个对象应该接收相关的选择子 (selector)。

开发者可以随意提供选择子。由于系统并不在运行期检查它，所以编写代码时需要小心。假如指定的选择子还未声明，那么编译器会发出警告，有时这能够提醒开发者注意选择子中的拼写错误，从而防止程序在运行时崩溃。下面这段代码设置了一对目标 – 动作组合，当用户在按钮内松开手指时，程序就会调用 playSound: 这个选择子。假如目标 (也就是 self) 没有实现那个方法，程序在运行的时候就会因为未定义的方法调用错误而崩溃：

```
[button addTarget:self action:@selector(playSound:)
forControlEvents:UIControlEventTouchUpInside];
```

目标 – 动作模式并不像委托那样去遵循一套预先建立好的方法规范。与委托及其所需的协议不同，系统并不强迫目标对象一定要实现 playSound:。开发者应该自己来确保回调操作一定能够执行某个已经实现好了的方法。在设置目标 – 动作组合之前，谨慎的程序员会先检查目标对象到底有没有实现指定的选择子。例如：

```
if ([someObject respondsToSelector:@selector(playSound:)])
[button addTarget:someObject action:@selector(playSound:)
forControlEvents:UIControlEventTouchUpInside];
```

对于标准的 UIControl 来说，在设定了目标 – 动作组合之后，系统一般会给选择子传递 0 个、1 个或 2 个参数。假如传递了参数的话，那么参数就是交互对象 (interaction object，也就是用户所操作的按钮、滑杆或开关等) 以及表示用户输入的 UIEvent 对象。系统可以单单把交互对象传递给选择子，也可以把交互对象和相关事件一并传过去。在前述范例代码中，选择子只接收一个参数，就是用户正在点击的 UIButton 实例。这种自我引用式的写法使得系统在回调选择子的时候，可以把用户所触发的对象以参数的形式传过去，从而令开发者能够编写出更为通用的动作代码，这种代码知道究竟是哪个控件引发了回调。

### 2.1.2 控件的种类

在 UIControl 家族中，系统所提供的成员有按钮、分段选择控件 (segmented control)、开关、滑杆、页面控制控件以及文本框。这些控件都可以在 Interface Builder (简称 IB) 的 Object Library 里面找到，如图 2-1 所示 (按下 Command-Control-Option-3 组合键，或者点击 View > Utilities > Show Object Library 菜单项)。

### 2.1.3 控件事件

控件主要响应三类事件：基于触摸的事件、基于值的事件，以及基于编辑的事件。表 2-1 列出了控件所能响应的每种事件。

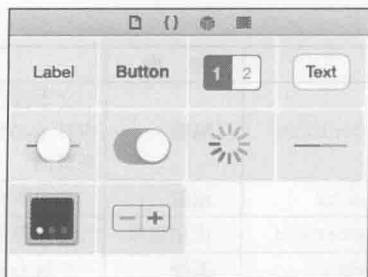


图 2-1 Interface Builder 提供了 Object Library 中可供使用的控件。从左上角开始，它们依次是：标签（label，UILabel）、按钮（button，UIButton）、分段选择控件（segmented control，UISegmentedControl）、文本框（text field，UITextField）、滑杆（slider，UISlider）、开关（switch，UISwitch）、活动指示器（activity indicator，UIActivityIndicatorView）、进度指示器（progress indicator，UIProgressView）、页面控制控件（page control，UIPageControl）以及步进控件（stepper，UIStepper），其中，活动指示器与进度指示器并不是严格意义上的控件

表 2-1 UIControl 所能响应的各种事件

事 件	类 型	含 义
UIControlEventTouchDown	触摸	在控件范围内所发生的 touch down 事件
UIControlEventTouchUpInside	触摸	在控件范围内所发生的 touch up 事件。这是按钮控件最常使用的事件类型
UIControlEventTouchUpOutside	触摸	完全发生在控件范围之外的 touch up 事件
UIControlEventTouchDragEnter UIControlEventTouchDragExit	触摸	这两个事件分别对应于由控件范围外进入控件范围内的拖曳操作以及从控件范围内离开控件的拖曳操作
UIControlEventTouchDragInside UIControlEventTouchDragOutside	触摸	在控件范围内发生的拖曳事件；刚刚离开控件范围的拖曳事件
UIControlEventTouchDownRepeat	触摸	tapCount 大于 1 的重复 touch down 事件（例如双击）
UIControlEventTouchCancel	触摸	能够取消当前触摸操作的一种系统事件。关于触摸操作的各个阶段及其生命期，请详见第 1 章
UIControlEventAllTouchEvents	触摸	对应于上述所有触摸事件的掩码，用于捕获任意触摸事件
UIControlEventValueChanged	值	由用户所触发且能够改变控件值的事件，例如移动滑杆控件上面的滑块，或切换开关状态
UIControlEventEditingDidBegin UIControlEventEditingDidEnd	编辑	分别表示在 UITextField 范围内和范围外所发生的触摸。在范围内触摸，会使文本框进入编辑状态。在范围外触摸，则会使其离开编辑状态
UIControlEventEditingChanged	编辑	修改 UITextField 内容的编辑操作

(续)

事 件	类 型	含 义
UIControlEventEditingDidEndOnExit	编辑	文本框在离开编辑状态时所发生的事件，该事件未必是由于用户在文本框范围之外点击而导致的
UIControlEventAllEditingEvents	编辑	表示所有 Editing 事件的掩码
UIControlEventApplicationReserved	应用程序	应用程序专用的事件范围（罕用）
UIControlEventSystemReserved	系统	系统专用的事件范围（罕用）
UIControlEventAllEvents	触摸、值、编辑、应用程序、系统	表示所有触摸、值、编辑、应用程序及系统事件的掩码

在大多数情况下，事件的用法都可以概括如下。按钮使用触摸事件，所有与按钮相关的操作几乎都可以通过 UIControlEventTouchUpInside 这种事件类型来处理，而这也正是 IB 创建连接 (connection) 时所采用的默认事件类型。如果用户调整了分段选择控件、开关、滑杆或页面控制控件，那么就会引发值事件（比方说 UIControlEventValueChanged）。刷新表格内容所用的刷新控件也能触发值事件。当用户切换、滑动或点击那些控件的时候，控件的值就会改变。UITextField 对象会引发编辑事件。当用户在文本框范围内或范围外点击以及修改文本框中的内容时，就会引发这类事件。

与 iOS 的所有 GUI 元件一样，你既可以通过 Xcode 的 IB 界面来排布控件，也可以用代码的方式来实例化它们。本章会讨论一些与 IB 有关的方案，不过我们主要还是关注基于代码的解决方案。只要学会了用 IB 来排布控件，以后无论遇到什么控件，都可以用相同的办法来操作。我们可以把控件拖放到界面里，通过检查器 (inspector) 及 Auto Layout(自动布局) 约束对其进行定制，并将它与其他 IB 对象相连。

## 2.2 按钮

UIButton 实例提供了简单的按钮。用户点击按钮之后，系统会经由目标 - 动作编程机制来触发回调。开发者可以指定按钮的外观、图样以及它所显示的文本。

iOS 提供了两种构建按钮的方式。你可以在预先设计好的风格中选择一种按钮，也可以从头开始构建自定义的按钮。目前 iOS SDK 所内置的按钮风格非常有限。iOS 7 系统以扁平化的极简风格重新设计了整套 UI。重新设计之后，最值得注意的变化之一就是它不再使用过时的圆角矩形 (rounded rectangle) 了。这就产生了一种非常新式的按钮，按钮中显示的是加粗的纯文本，这种按钮对应于 UIButtonTypeSystem 类型。此外，其余几种类型的按钮在 UI 方面的差别也变得不是那么明显了。

预设的几种按钮并不是特别通用。苹果公司之所以将其添加到 SDK 里，主要是为了它自己用着方便，而不是为了让开发者用起来方便。一般来说，假如苹果公司自己不常使用某种 UI 特性，那它就不会将其添加到 SDK 里。不过，只要你遵照苹果公司的《Human Interface Guidelines》(人机界面指南，简称 HIG) 来设计界面，就可以在自己的程序里使用那

些控件。图 2-2 演示了 5 种按钮。

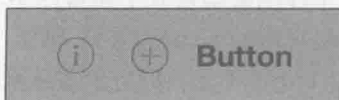


图 2-2 iOS SDK 提供了 5 种风格的按钮，它们的界面可以归结为 3 种不同的视觉效果。开发者可通过 IB 来获取按钮，也可在程序中直接用代码来添加。图中的第一个符号对应于三种按钮：Detail Disclosure、Info Light 与 Info Dark。另外两种按钮是 Contact Add 和 System 按钮

- **Detail Disclosure**（详情展示按钮）——这是个蓝色的圆圈，中间写有字母 i，向表格的单元格添加 Detail Disclosure Accessory 时就会看到这种按钮。点击表格中的 Detail Disclosure 按钮之后，程序应该会切换到另外一个屏幕，以显示用户所选单元格的详细信息。在 iOS 7 系统之前，该按钮的样子是个 encircled chevron<sup>①</sup>。
- **Info Light**（亮色信息按钮）及 **Info Dark**（暗色信息按钮）——这两种按钮的符号与 Detail Disclosure 类似，也都是蓝色圆圈里面带个字母 i。Macintosh 系统 Dashboard 界面的 Widget 里会出现这两种按钮，点击它们之后可以看到与该 Widget 有关的信息，或是会跳转到该 Widget 的设置界面。在许多应用程序中，这些按钮用来将视图从一面翻转到另一面。
- **Contact Add**（添加联系人按钮）——蓝色圆圈正中有个 + 号。在 Mail 程序中，用户可以通过该按钮为邮件信息添加新的收信人。
- **System**——该按钮的背景是透明的，按钮之中有个文本标签。System 按钮没有斜面效果 (baze)，也没有背景，不过开发者可以自定义它的图像和文本标题。

严格地说，UIButtonTypeCustom 也算是个预制的 (precooked) 的按钮，因为它毕竟包含了一个 UILabel。但由于它并没有再提供外观方面的其他支持，所以大部分开发者都将其看作一种完全需要由自己来定义的按钮。

若想在代码中使用某种内置的按钮，那就先为其分配内存，然后设定它的框架或 Auto Layout 约束，最后再调用 addTarget 方法来添加目标即可。开发者无须担心如何给按钮添加特定的图样，也不用考虑如何创建按钮的总体样貌，这些都由 SDK 来做。比方说，下面这段代码就能构建一个简单的 System 按钮：

```
UIButton *button = [UIButton buttonWithType:UIButtonTypeSystem];
[button setFrame: CGRectMake(0.0f, 0.0f, 80.0f, 30.0f)];
[button setCenter: self.view.center];
[button setTitle:@"Beep" forState:UIControlStateNormal];
[button addTarget:self action:@selector(playSound)
    forControlEvents:UIControlEventTouchUpInside];
[contentView addSubview:button];
```

如果要构建其他类型的标准按钮，那就把 setTitle 那一行代码删掉。在预设的各类按钮之中，只有 System 按钮才会用到 title。

① 可以理解为“带圆圈的大于号”或“带圆圈的 V 形图案”。——译者注



在 iOS 7 中, 包括 UIButton 在内的所有 UIView 都支持 `tintColor` 属性。该属性很特殊: 假如不做任何处理, 那么子视图就会从其上级视图中继承这个 `tint color`<sup>①</sup>。依照这种方式来看, 如果设置了应用程序根视图的 `tintColor` 属性, 那么它就会改变整个应用程序的 `tint color`。直接设置子视图的 `tintColor`, 即可覆盖继承下来的属性值。

内置的几种按钮默认的 `tint color` 都是蓝色, 而开发者可以通过 `tintColor` 属性把它改成自己所选用的颜色。对于 iOS 7 系统里面由苹果公司所提供的视图及控件来说, `tintColor` 所定义的颜色通常用于描绘用户正在操作的控件, 或是强调用户在视图里所选定的某一部分内容。

大部分按钮都通过 `UIControlEventTouchUpInside` 事件来处理用户的操作, 如果用户的触摸操作是在按钮自身范围内结束的, 那么就会触发该事件。iOS 的 UI 设计标准允许用户通过把手指移到按钮范围之外再松开手指的方式来取消对按钮的点击。选用 `UIControlEventTouchUpInside` 事件来处理用户对按钮的操作是符合这条设计标准的。

如果要使用系统所提供的按钮, 那么在设计按钮的用法时, 就必须遵循苹果公司的《HIG》。比方说, 如果添加了一个跳转到信息页面的 Detail Disclosure, 那么可能导致你的应用程序遭到 App Store 拒绝。你可能认为自己的这种用法只不过是适当地延伸了按钮的用途, 但是从《HIG》的具体规则来看, 这种用法与苹果公司对该按钮的要求并不相符, 所以, 程序可能通不过评审。(程序能否通过评审, 当然是由评审人员决定的, 但如果应用程序因为违背《HIG》而遭到拒绝, 那么程序提交者很难为此辩解。) 为了避开这些潜在的问题, 开发者应该尽量使用 System 按钮或自定义的按钮。

## 2.3 Interface Builder 中的按钮

IB 的 Object Library 中所列出的按钮的默认类型是 System (参见图 2-1)。要想使用这种按钮, 可以将其拖曳到程序界面中。然后, 你可以在 Attributes Inspector 里修改按钮的类型 (通过 View > Utility > Show Attributes Inspector 菜单项或 Command-Option-4 组合键即可调出 Attributes Inspector)。Attributes Inspector 顶部有个用于修改按钮类型的列表框。点击列表框之后, 会弹出一份菜单, 开发者可以从中选择自己想要的按钮类型。

如果按钮中需要显示文本, 那么可以在 Title 文本框里输入文本。开发者可通过 Image 及 Background 这两个下拉列表框来选择按钮的主图像及背景图像。每个按钮都提供了四套 State Config。这四套 Config 所对应的四种状态分别是: Default (表示按钮处在通常状态)、Highlighted (用户正在触摸该按钮)、Selected (对于能够切换状态的按钮来说, 这表示按钮处在 On 的状态) 以及 Disabled (表示用户无法操作该按钮)。

开发者在 Attributes Inspector 的 State Config 区域内对按钮所做的各项变更都只对应于当前所选的 State Config。比方说, 开发者可以为同一个按钮的 Default 状态及 Disable 状态分别设置文本颜色。

① 这是一种渲染 UI 所用的颜色, 用以表示当前拥有焦点的某个控件, 或强调屏幕中值得注意的某个部分。可以理解为主题色、提示色、高亮色、主题色, 等等。——译者注



要想预览一下按钮在各个状态下的效果，可以在 Attributes Inspector 的 Control 区域中调整那三个 Content 复选框。开发者可以经由 Highlighted、Selected 及 Enabled 这三个选项来设定按钮的状态。预览完毕之后，记得在编译前把按钮恢复到首次运行程序时所应具备的实际状态。

## 将按钮与动作相连

如果按住 Control 键不放，在 IB 编辑器界面里从按钮向某个 IB 对象（例如 File's Owner 所对应的视图控制器）拖曳，那么当松开鼠标按钮时，IB 就会把各种动作列在一份弹出式菜单里面，供开发者选择。（另一种操作方式是用鼠标右键拖曳。）这些动作是根据目标对象所能支持的 IBAction 而列出来的。把按钮与动作相连之后，我们就为按钮的 UIControlEventTouchUpInside 事件创建了一对目标 - 动作组合。你也可以按住 Control 键不放，把按钮拖曳到自己的代码上面，这样一来，Xcode 就会在实现文件里面添加一个空的函数定义。

另外，你还可以按住 Control 键并点击 Document Outline 中的按钮控件（也可以用鼠标右击 Document Outline 中的按钮控件），然后在弹出的菜单中找到 Touch Up Inside 这一项，把旁边的空心圆圈拖到你想要连接的东西上面（在笔者举的这个例子中，指的就是 File's Owner 对象）。此时也会出现刚才所说的弹出式菜单，菜单里面列出了各种可供使用的动作。



**提示** 在 IB 里面，你会遇到看着很像视图而行为也很像视图的按钮，但实际上，那些按钮并不是视图。工具栏或导航栏上会出现一些那样的按钮，UIBarButtonItem 用于保存那种按钮的属性，但 UIBarButtonItem 本身却不是按钮。工具栏与导航栏会在内部构建一种特殊的按钮来表示这种逻辑实体。

## 2.4 解决方案：构建按钮

如果要使用 UIButtonTypeCustom 风格的按钮，开发者应该自己来提供按钮所需的全部图片。具体应该提供几张图像要根据按钮的工作方式来定。对于那种简单的推压按钮（pushbutton）来说，只需要提供一张背景图即可，另外，还需要在用户按下按钮的时候改变其文本颜色，以实现高亮效果。

对于切换式按钮（toggle-style）来说，可能需要准备四张图：off 状态下的普通图样、off 状态下的高亮图样（所谓高亮，就是指用户按下按钮时的样子）、on 状态下的普通图样以及 on 状态下的高亮图样。与操作该按钮有关的细节需要由开发者选定并设计出来，我们还要记得给程序里面添加相关的状态（比如像解决方案 2-1 那样，用名为 isOn 的布尔型实例变量来表示按钮状态），这样就可以把普通的推压按钮扩展为切换式按钮了。假如你只提供了一张普通的图像，而没有指明按钮在高亮及禁用状态时所用的图像，那么 iOS 就会自动生成那些图像。

解决方案 2-1 构建了一种可在 on 和 off 之间来回切换的按钮，并演示了构建自定义按钮时所需处理的基本细节。用户点击按钮时，其颜色会从绿色变为红色，或从红色变为绿色（红色表示 off，绿色表示 on）。只要用户不是色盲，立刻就能够根据按钮颜色辨识出当前的状态。而按钮中间所显示的文本也与当前状态相对应，这就进一步强化了按钮状态的可辨识程度。图 2-3 左侧就是本条解决方案所要创建的那种按钮。

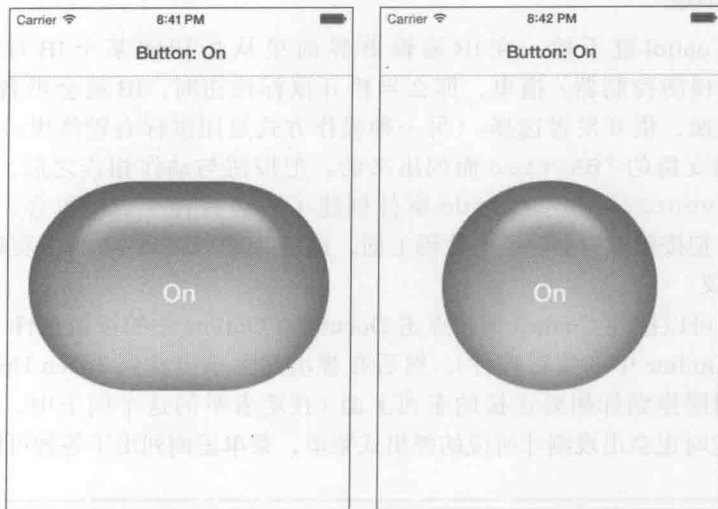


图 2-3 通过拉伸 UIImage 来调整按钮图样，使之能够适用于任意宽度（width）的按钮。开发者可以设置左右两侧的 cap width 值，以限定需要执行拉伸操作的范围

UIImage 类的 resizableImageWithCapInsets 方法可以缩放图像，而这正是本条解决方案在创建按钮时所执行的关键操作。借助图像缩放功能，我们把圆形图样拉伸为扁的菱形图样，从而创建出任意宽度的按钮来。开发者需要指定 cap 值（该值用于描述图像里面不应该拉伸的部分）。在本例中，左右两侧的 cap 都是 110 像素。假如我们把按钮宽度从本例中的 300 像素变为 220 像素，那么按钮中间拉伸的部分就不见了，它会变成图 2-3 右侧那个样子。

开发者可以根据状态来设定按钮的图像和背景图。图像表示按钮的实际内容，而背景图则好比一块幕布，图像及文本都会出现在它的前方。解决方案 2-1 设置了按钮的背景图，使得按钮内的标题（title）<sup>①</sup>可以浮动在开发者所提供的图样上方。

**提示** 通过调整 CALayer 对象的相关属性，我们可以修改视图及按钮的四个角所具备的圆滑程度。把 Quartz Core 框架添加到项目中，然后就可以用代码来访问视图的 CALayer 对象并设置其 cornerRadius 属性。设置好这个属性之后，我们把视图的 clipsToBounds 属性设为 YES，以便实现苹果公司风格的圆角效果。

① 具体是指 titleLabel 属性。title 或 text title 这种说法可以理解成按钮内的文本。——译者注

### 解决方案 2-1 构建一种可在 On 和 Off 状态之间切换的 UIButton

```
#define CAPWIDTH 110.0f
#define INSETS (UIEdgeInsets){0.0f, CAPWIDTH, 0.0f, CAPWIDTH}
#define BASEGREEN [[UIImage imageNamed:@"green-out.png"] \
    resizableImageWithCapInsets:INSETS]
#define PUSHGREEN [[UIImage imageNamed:@"green-in.png"] \
    resizableImageWithCapInsets:INSETS]
#define BASERED [[UIImage imageNamed:@"red-out.png"] \
    resizableImageWithCapInsets:INSETS]
#define PUSHRED [[UIImage imageNamed:@"red-in.png"] \
    resizableImageWithCapInsets:INSETS]
```

```
- (void)toggleButton:(UIButton *)aButton
```

```
{
```

```
    self.isOn = !self.isOn;
```

```
    if (self.isOn)
```

```
    {
```

```
        [self setBackgroundImage:BASEGREEN
            forState:UIControlStateNormal];
```

```
        [self setBackgroundImage:PUSHGREEN
            forState:UIControlStateHighlighted];
```

```
        [self setTitle:@"On" forState:UIControlStateNormal];
```

```
        [self setTitle:@"On" forState:UIControlStateHighlighted];
```

```
    }
```

```
    else
```

```
    {
```

```
        [self setBackgroundImage:BASERED
            forState:UIControlStateNormal];
```

```
        [self setBackgroundImage:PUSHRED
            forState:UIControlStateHighlighted];
```

```
        [self setTitle:@"Off" forState:UIControlStateNormal];
```

```
        [self setTitle:@"Off" forState:UIControlStateHighlighted];
```

```
    }
```

```
}
```

```
+ (instancetype)button
```

```
{
```

```
    PushButton *button =
```

```
        [PushButton buttonWithType:UIButtonTypeCustom];
```

```
    // Set up the button alignment properties
```

```
    button.contentVerticalAlignment =
        UIControlContentVerticalAlignmentCenter;
```

```
    button.contentHorizontalAlignment =
        UIControlContentHorizontalAlignmentCenter;
```

```
    // Set the font and color
```

```
    [button setTitleColor:
```

```
        [UIColor whiteColor] forState:UIControlStateNormal];
```

```
    [button setTitleColor:
```

```

        [UIColor lightGrayColor] forState:UIControlStateHighlighted];
        button.titleLabel.font = [UIFont boldSystemFontOfSize:24.0f];

        // Set up the art
        [button setBackgroundImage:BASEGREEN
            forState:UIControlStateNormal];
        [button setBackgroundImage:PUSHGREEN
            forState:UIControlStateHighlighted];
        [button setTitle:@"On" forState:UIControlStateNormal];
        [button setTitle:@"On" forState:UIControlStateHighlighted];
        button.isOn = YES;

        // Add action. Client can add one too.
        [button addTarget:button action:@selector(toggleButton:)
            forControlEvents: UIControlEventTouchUpInside];

        return button;
    }

```

### 2.4.1 多行按钮文本

通过按钮的 `titleLabel` 属性，可以修改其标题，例如，可以改变字体及换行模式。我们故意把字设置得非常大，使文本差不多达到足以换行的程度，然后把换行模式设为在词的边界处换行（word wrap），并把对齐方式设为居中：

```

button.titleLabel.font = [UIFont boldSystemFontOfSize:36.0f];
[button setTitle:@"Lorem Ipsum Dolor Sit" forState:
    UIControlStateNormal];
button.titleLabel.textAlignment = NSTextAlignmentCenter;
button.titleLabel.lineBreakMode = UILineBreakModeWordWrap;

```

在默认情况下，按钮内的标签会从按钮的一端延伸至另一端。这也就是说，文本可能会超出预期的宽度，甚至可能会越过按钮图片的边界。为解决此问题，可以在文本中插入换行符（也就是 `\n`），这样就能够在 word wrap 模式下实现强制回车了。由此，我们可以分别控制按钮标题每一行所包含的字符数。

### 2.4.2 为按钮添加动画元件

制作按钮的时候，可以把一些图形放在按钮的前面或后面。我们采用标准的 `UIView` 体系来做这件事，并在相关的视图中把 `setUserInteractionEnabled` 设为 `NO`，禁止用户去操作那些可能会干扰该按钮的视图。我们还要把 `Image` 视图里的内容露出来，使用户能够看到开发者为按钮所添加的动画元件。

解决方案 2-1 使用一张半透明的范例图来试验这种效果。这段范例代码会添加一组蝴蝶图案，开发者可以把它放在按钮后面，并播放其动画效果。

动画元件对于展示状态来说特别有用，尤其适合展示正在进行中的操作。它们可以告诉用户按钮为什么无法响应，也可以在用户按下按钮的时候产生另外一种反应。比方说，游戏里面可能会出现超强按钮（turbo-enhanced button），玩家按下它的时候会获得额外的力量。对

于这种按钮来说，开发者可以通过动画效果使玩家意识到按钮的功能已经发生了变化。

把图案及文本同按钮的实现代码分开，这种做法在开发程序的时候还有其他用途。我们可以先设计一个空白的按钮，然后把这些东西添加到该按钮的后面或前面，这样一来，开发者就能在无须重新设计整个按钮的前提下，分别控制其图案及文本了。

### 2.4.3 为按钮添加额外状态

解决方案 2-1 创建的按钮具备两种状态，分别能够呈现视觉上的开、关效果。有时候我们需要给按钮再添加一些容易辨识的状态，最常见的场合是游戏。许多游戏开发者需要实现具备四种状态的按钮，分别用来表示尚未开放的关卡（locked level）、已开放但还未玩过的关卡（unlocked-but-not-played）、已开放而且已玩过部分内容的关卡（unlocked-and-partially-played）以及已开放而且已完成的关卡（unlocked-and-mastered）。

解决方案 2-1 使用简单的布尔值（也就是 isOn 这个实例变量）来保存 on 和 off 这两种状态，然后在 toggleButton: 方法中根据当前状态来选择按钮图案。可以改编这段范例代码，用整数来保存状态，并通过 switch 语句选择相关状态下的图案，这样就能够令按钮支持更多的图案和状态了。

---

#### 获取解决方案代码

访问 <https://github.com/erica/iOS-7-Cookbook> 网页，并打开 C02 Controls 文件夹，即可找到与本章中的解决方案相对应的完整范例项目。

---

## 2.5 解决方案：使按钮以动画效果来响应用户

除了 frame 属性与目标 - 动作机制之外，UIControl 实例里面还有许多东西。所有的控件都继承自 UIView 类。这就意味着开发者在制作控件时，可以像对待标准的视图那样，在它上面调用 animateWithDuration，并把一段代码放在块里面，以实现动画效果。解决方案 2-2 所构建的开关式按钮能够在用户点击它的时候变大，并在用户松开手指的时候恢复原状。

---

#### 解决方案 2-2 把 UIView 动画代码块添加到控件中

---

```
- (void)zoomButton:(id)sender
{
    // Slightly enlarge the button
    [UIView animateWithDuration:0.2f animations:^(
        button.transform =
            CGAffineTransformMakeScale(1.1f, 1.1f);
    )];
}

- (void)relaxButton:(id)sender
{
    // Return the button to its normal size
```

```

[UIView animateWithDuration:0.2f animations:^(
    button.transform = CGAffineTransformIdentity;
)}
- (void)toggleButton:(UIButton *)button
{
    self.isOn = !self.isOn;
    if (self.isOn)
    {
        [button setTitle:@"On" forState:UIControlStateNormal];
        [button setTitle:@"On" forState:UIControlStateHighlighted];
        [button setBackgroundImage:BASEGREEN
            forState:UIControlStateNormal];
        [button setBackgroundImage:PUSHGREEN
            forState:UIControlStateHighlighted];
    }
    else
    {
        [button setTitle:@"Off" forState:UIControlStateNormal];
        [button setTitle:@"Off"
            forState:UIControlStateHighlighted];
        [button setBackgroundImage:BASERED
            forState:UIControlStateNormal];
        [button setBackgroundImage:PUSHRED
            forState:UIControlStateHighlighted];
    }
    [self relaxButton:button];
}

+ (instancetype)button
{
    PushButton *button =
        [PushButton buttonWithType:UIButtonTypeCustom];

    // Add actions
    [button addTarget:button action:@selector(toggleButton:)
        forControlEvents:UIControlEventTouchUpInside];
    [button addTarget:button action:@selector(zoomButton:)
        forControlEvents:UIControlEventTouchUpInside |
        UIControlEventTouchDragInside |
        UIControlEventTouchDragEnter];
    [button addTarget:button action:@selector(relaxButton:)
        forControlEvents:UIControlEventTouchUpInside |
        UIControlEventTouchCancel |
        UIControlEventTouchDragOutside];

    return button;
}

```

本条解决方案创建了一种活泼的交互效果，使用户能够更加关注其所操作的控件。



请注意，开发者可以在按钮上调用 `setAttributedTitleForState:` 方法来设定 `NSAttributedString` 值，通过这项巧妙的功能，我们可以给按钮再添加一些效果。本章稍后的解决方案 2-4 通过类似的办法，改变分段选择控件（`segmented control`）上面的文本颜色。

## 2.6 解决方案：为滑杆控件添加自定义的滑块

`UISlider` 实例对应于滑杆（`slider`）控件，用户可以通过滑块在该控件的左极值和右极值之间滑动，以选定某个数值。Music 程序里就有这个控件，用户可以用 `UISlider` 类控制音量大小。

在默认情况下，Slider 控件的最小值是 0.0，最大值是 1.0，开发者可以在 IB 的 `Attributes Inspector` 里面调整这两个值，也可以通过 `minimumValue` 及 `maximumValue` 属性来设置它们。为了美化该控件的两个端点，我们可通过 `minimumValueImage` 及 `maximumValueImage` 属性来设定一对图像，以便更直观地表示出控件的最小值和最大值。比方说，对于调整温度的 Slider 控件来说，你可以在一端放上雪人图案，另一端放上一杯热茶。

通过 `minimumTrackTintColor`、`maximumTrackTintColor` 及 `thumbTintColor` 这三个属性，开发者可以设置位于滑块之前和滑块之后的轨道颜色，也可以设置滑块本身的颜色。在 iOS 7 系统中，如果把 `minimumTrackTintColor` 设为 `nil`，那么位于滑块之前的轨道的色彩颜色（`tint color`）就和上级视图的相同。其他两个属性若是 `nil`，系统则将其视为相关的默认颜色。

用户拖曳滑块时，Slider 控件是否会持续不断地发送数值更新通知要通过 `continuous` 属性来控制。该属性的默认值是 `YES`，如果将其设为 `NO`，那么 Slider 控件只会在用户松开滑块时发送一次动作事件。

### 2.6.1 定制 `UISlider` 控件

除了可以设置表示最小值与最大值的图像之外，开发者也可以直接更新 `UISlider` 控件的滑块组件。`setThumbImageForState:` 方法可以把滑块设定成自己想要的图像。解决方案 2-3 采用这种方式动态地构建滑块图像，其效果如图 2-4 所示。用户在用手指调整滑块的时候，上方会出现气

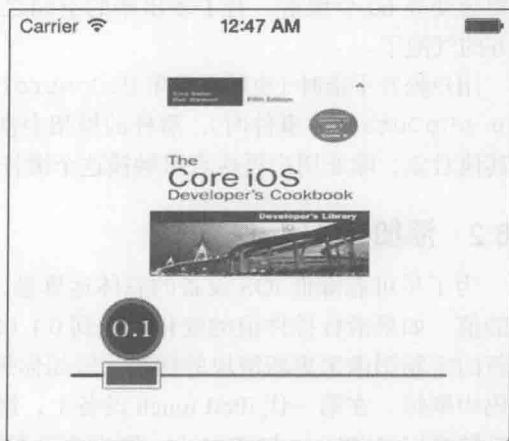


图 2-4 通过调用 Core Graphics/Quartz API，Slider 控件的滑块图像能够随着当前的控件值由暗变亮。写在滑块上方气泡里的文本就是当前的控件值（用户拖曳控件时，屏幕上方的图标会按比例缩放，笔者之所以又添加了这么一种视觉反馈方式，是想要在范例代码里演示目标 - 动作机制的用法）

泡状的提示信息，而这个气泡也是滑块的一部分。该气泡能够以文本和图像两种方式把控件数值实时反馈给用户：气泡内写有控件当前的值，而气泡颜色也会随着用户的拖曳逐渐由黑变白。



**提示** 修改 UIView 所显示的内容时，iOS 会为视图的 CALayer 创建一张位图。无论是在定制视图上面操作还是在生成的位图上面操作，其效率都差不多。

我们可以依照任意类型的数据来构建这种动态反馈效果。除了根据用户手指的移动来读取 Slider 控件值之外，还有很多办法，比方说，可以从设备的感应器里面抓取数据，也可以去获取 Internet 上面的数据。无论采用哪种办法，动态更新都必然要涉及大量的图形运算，不过，其运算量也没有大到令人担忧的程度。Core Graphics API 的执行速度很快，而且生成滑块那么大的图像所需的内存量也很小。

这份解决方案给 Slider 控件设置了两种滑块图像。只有当用户正在使用 Slider 的时候，也就是控件处于 UIControlStateHighlighted 状态时，其上方才会显示出气泡。在普通状态下，也就是 UIControlStateNormal 状态下，用户只会看到表示滑块的那个小矩形。点击滑块之后，就能知道 Slider 控件当前的值了。这种与上下文相关的反馈气泡与标准 iOS 键盘的字母高亮效果<sup>①</sup>相似。

为了适应绘制效果上的变化，我们需要在每个手势开始和结束的时候修改 Slider 控件的框架 (frame)。用户开始触摸控件时 (也就是发生 UIControlEventTouchDown 事件时)，框架会变高 60 个像素。有了多出来的空间之后，用户就可以在拖动控件的过程中看到滑块上方的气泡了。

用户松开手指时 (也就是发生 UIControlEventTouchUpInside 或 UIControlEventTouchUpOutside 事件时)，滑杆的框架会恢复到它原来的大小。这样就能腾出空间，以显示其他对象，除非用户再次直接触摸这个滑杆，否则它不会激活。

## 2.6.2 添加优化代码

为了尽可能降低 iOS 设备的总体运算量，我们在解决方案 2-3 里保存了 Slider 控件前一次的值。如果滑杆控件值的变化量达到 0.1 (也就是达到取值范围的 10%)，那么我们就用一张新的定制图像来更新滑块的样貌。假如你想实现完全实时的更新效果，可以把这项检测从代码中删掉。在第一代 iPod touch 设备上，滑块图像的更新操作执行得就已经相当快了，而在近期推出的 iPhone 与 iPad 上，则完全没有性能问题。

### 解决方案 2-3 为滑杆控件制作动态图样

```
/* Thumb.m */
// Create a thumb image using a grayscale/numeric level
UIImage *thumbWithLevel(float aLevel)
{
    float INSET_AMT = 1.5f;
```

① 用户点击键盘上的某个字母时，该字母会以放大的形态显示在按键上方。——译者注



```

CGRect baseRect = CGRectMake(0.0f, 0.0f, 40.0f, 100.0f);
CGRect thumbRect = CGRectMake(0.0f, 40.0f, 40.0f, 20.0f);

UIGraphicsBeginImageContext(baseRect.size);
CGContextRef context = UIGraphicsGetCurrentContext();

// Create a filled rect for the thumb
[[UIColor darkGrayColor] setFill];
CGContextAddRect(context,
    CGRectInset(thumbRect, INSET_AMT, INSET_AMT));
CGContextFillPath(context);

// Outline the thumb
[[UIColor whiteColor] setStroke];
CGContextSetLineWidth(context, 2.0f);
CGContextAddRect(context,
    CGRectInset(thumbRect, 2.0f * INSET_AMT, 2.0f * INSET_AMT));
CGContextStrokePath(context);
// Create a filled ellipse for the indicator
CGRect ellipseRect = CGRectMake(0.0f, 0.0f, 40.0f, 40.0f);
[[UIColor colorWithWhite:aLevel alpha:1.0f] setFill];
CGContextAddEllipseInRect(context, ellipseRect);
CGContextFillPath(context);

// Label with a number
NSString *numString =
    [NSString stringWithFormat:@"%0.1f", aLevel];
UIColor *textColor = (aLevel > 0.5f) ?
    [UIColor blackColor] : [UIColor whiteColor];
UIFont *font = [UIFont fontWithName:@"Georgia" size:20.0f];
NSMutableParagraphStyle *style =
    [[NSMutableParagraphStyle alloc] init];
style.lineBreakMode = NSLineBreakByCharWrapping;
style.alignment = NSTextAlignmentCenter;
NSDictionary *attr = @{NSFontAttributeName:font,
    NSParagraphStyleAttributeName:style,
    NSForegroundColorAttributeName:textColor};
[numString drawInRect:CGRectInset(ellipseRect, 0.0f, 6.0f)
    withAttributes:attr];

// Outline the indicator circle
[[UIColor grayColor] setStroke];
CGContextSetLineWidth(context, 3.0f);
CGContextAddEllipseInRect(context,
    CGRectInset(ellipseRect, 2.0f, 2.0f));
CGContextStrokePath(context);

// Build and return the image
UIImage *theImage = UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();

```

```

        return theImage;
    }

    // Return a base thumb image without the bubble
    UIImage *simpleThumb()
    {
        float INSET_AMT = 1.5f;
        CGRect baseRect = CGRectMake(0.0f, 0.0f, 40.0f, 100.0f);
        CGRect thumbRect = CGRectMake(0.0f, 40.0f, 40.0f, 20.0f);

        UIGraphicsBeginImageContext(baseRect.size);
        CGContextRef context = UIGraphicsGetCurrentContext();

        // Create a filled rect for the thumb
        [[UIColor darkGrayColor] setFill];
        CGContextAddRect(context,
            CGRectInset(thumbRect, INSET_AMT, INSET_AMT));
        CGContextFillPath(context);

        // Outline the thumb
        [[UIColor whiteColor] setStroke];
        CGContextSetLineWidth(context, 2.0f);
        CGContextAddRect(context,
            CGRectInset(thumbRect, 2.0f * INSET_AMT, 2.0f * INSET_AMT));
        CGContextStrokePath(context);

        // Retrieve the thumb
        UIImage *theImage = UIGraphicsGetImageFromCurrentImageContext();
        UIGraphicsEndImageContext();
        return theImage;
    }

    /* CustomSlider.m */
    // Update the thumb images as needed
    - (void)updateThumb
    {
        // Only update the thumb when registering significant changes
        if ((self.value < 0.98) &&
            (ABS(self.value - previousValue) < 0.1f)) return;

        // create a new custom thumb image for the highlighted state
        UIImage *customImg = thumbWithLevel(self.value);
        [self setThumbImage:customImg
            forState:UIControlStateNormal];
        previousValue = self.value;
    }

    // Expand the slider to accommodate the bigger thumb
    - (void)startDrag:(UISlider *)aSlider
    {
        self.frame = CGRectInset(self.frame, 0.0f, -30.0f);
    }

```

```

// At release, shrink the frame back to normal
- (void)endDrag:(UISlider *)aSlider
{
    self.frame = CGRectInset(self.frame, 0.0f, 30.0f);
}

- (instancetype)initWithFrame:(CGRect)aFrame
{
    self = [super initWithFrame:aFrame];
    if (self)
    {
        // Initialize slider settings
        previousValue = CGFLOAT_MIN;
        self.value = 0.0f;

        [self setThumbImage:simpleThumb()
            forState:UIControlStateNormal];

        // Create the callbacks for touch, move, and release
        [self addTarget:self action:@selector(startDrag:)
            forControlEvents:UIControlEventTouchDown];
        [self addTarget:self action:@selector(updateThumb)
            forControlEvents:UIControlEventValueChanged];
        [self addTarget:self action:@selector(endDrag:)
            forControlEvents:UIControlEventTouchUpInside |
            UIControlEventTouchUpOutside];
    }
    return self;
}

+ (instancetype)slider
{
    CustomSlider *slider = [[CustomSlider alloc]
        initWithFrame:(CGRect){.size=CGSizeMake(200.0f, 40.0f)}];

    return slider;
}

```

当滑块接近右端，也就是控件值接近 1.0 时，可能会出现前一次的值大于 0.9，而本次变更量又不足 0.1 的情况。为了应对这种情况，笔者把 0.98 这个“硬编码”放在解决方案里面，迫使滑杆在值大于 0.98 的时候无条件地更新滑块图像。

## 2.7 解决方案：创建可以连续点击两次的分段选择控件

UISegmentedControl 类提供了一套含有多个按钮的界面，用户可从这一组按钮里面选择一个。该控件有两种用法（这里指的不是用于改变控件 UI 样貌的 UISegmentedControlStyle，iOS 7 已经将 UISegmentedControlStyle 弃用了）。第

一种用法和一组普通的单选按钮类似，用户选定了一个按钮之后，它就一直处于受选状态。此时用户可以点击控件上的其他按钮，也可以点击目前已经选定的按钮，但是，重复点击已经选择的按钮并不产生新的事件。如果把名为 `momentary` 的布尔属性设为 `YES`，那么控件就会切换到另外一种用法，此时用户可以在每个按钮上面随意点击，但无论点击多少次，控件都不会用状态来记录当前受选的是哪个按钮。也就是说，控件不会用高亮效果来描绘用户最新选定的按钮。

解决方案 2-4 把这两种用法结合起来，使用户既能看到当前选中的这一项，又能继续点击已经选定的按钮。这种运作方式与分段选择控件的常见用法不同。不过有些情况下，我们需要在用户重复点击已经选中的按钮时产生一种新的效果（这类似于刚才提到的第二种用法），同时，又希望控件能够把最新选定的按钮标识出来（这类似于刚才提到的第一种用法）。

用常规方式是无法实现这个需求的。不能通过添加一对目标-动作组合来检测 `UIControlEventTouchUpInside` 事件。因为 `UISegmentedControl` 实例只会产生一种控件事件，即 `UIControlEventValueChanged` 事件。（读者可以自己试着添加针对触摸事件的目标-动作组合，看看是不是这样。）

该需求可以通过子类来实现。从 `UISegmentedControl` 继承新的类，并在该类上面响应用户对同一个按钮的第二次点击，这相对来说比较简单。解决方案 2-4 就定义了这样的子类。这个自定义的 `UISegmentedControl` 控件会从 `UIControl` 中继承一套内置的触摸事件处理器，此外，它自己还有一套触摸事件处理代码，当检测到触摸事件时，这两套处理机制将会各自独立运作。

`UISegmentedControl` 控件上的开关按钮都不会受影响，它们依然能够随着用户的点击而来回切换。但与其父类不同的是，如果用户在已经按下的按钮上面继续点击，那么子类会做一些事情。在本例中，子类所做的事情是在本对象的 `tapDelegate` 上触发 `performSegmentAction` 方法。

在使用 `UISegmentedControl` 子类时，我们不能像平常那样给它添加目标-动作组合。由于子类会捕获所有 `touch down` 事件，所以，如果开发者又针对 `UIControlEventValueChanged` 事件设定了目标-动作组合，那么实际上就等于把相关的回调方法重复添加了一遍，等到用户点击控件上的按钮时，系统就会回调两次。假如需要执行一些更新，那么就请在 `tapDelegate` 的回调方法里去实现代码，等系统在 `tapDelegate` 上触发回调时，自然就会执行那些代码了。

### 2.7.1 实现第二次点击时的反馈效果

有了检测第二次点击的能力之后，我们就可以在用户反复选取同一个按钮时实现一些反馈效果，比方说，可以更改导航栏的标题。同时，还可以考虑另外一种反馈效果，即修改控件文本的某些属性。从 iOS 5.x 开始，UIKit 就提供了一种 `text attribute`（文本属性）特

性，该特性很适合用来实现刚说的这种反馈效果。UISegmentedControl 控件可以设置一些基于状态的属性。我们可通过 setTitleTextAttributes:forState: 方法给用户所选定的按钮添加一些视觉美化效果。解决方案 2-4 使用该方法来修改按钮的文本颜色，当用户初次点击某按钮时，令其文本变为白色，重复点击同一按钮时，再将其改为红色。如果用户点击了其他按钮，那么就将其复原，效果如图 2-5 所示。

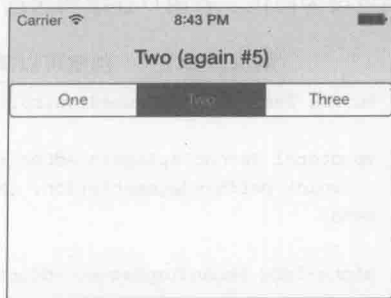


图 2-5 如果用户在 UISegmentedControl 控件上反反复复点击某个已经选定的按钮，那么就提供一种视觉反馈效果来表示这种不太常见的操作。我们通过 setTitleTextAttributes:forState: 方法修饰了相关文本，以便进一步强调用户当前所选的按钮

## 2.7.2 控件及带属性的字符串

从 iOS 6 开始，UIKit 中包括文本框、文本视图、标签、按钮及刷新控件等在内的很多类都开始支持带属性的字符串（Attributed String）<sup>①</sup>（也就是 Core Text 风格的字符串），开发者可以用这种字符串来为控件的文本及标题设定相关的属性：

```
[myButton setTitleAttributedString:attributedString forState:UIControlStateNormal]
```

与文本有关的属性包括字体（font）、颜色（color）、阴影（shadow）等，而 iOS 7 又扩充了开发者可以配置的属性范围。打开 NSAttributedString 类的参考文档<sup>②</sup>，在 Character Attributes 下能够看到一份完整的属性列表，开发者可通过 Attributed String 来修改控件文本的属性。

对于 UISegmentedControl 控件及 UIBarButtonItem 来说，我们可以调用 setTitleTextAttributes:forState: 方法来设置 Attributed String。开发者需要选用适当的 key（键）和 value（值），把待设置的属性放到 NSDictionary 里，然后传给该方法，下面列出了几种供大家参考：

- **NSFontAttributeName**——如果用它做键，那么值应该是个 UIFont 实例。
- **NSForegroundColorAttributeName**——这种键所对应的值应该是个 UIColor 实例。
- **NSShadowAttributeName**——如果把它当成键，那么值应该是 NSShadow 实例，而开发者可以在 NSShadow 实例上面指定阴影的偏移量（offset）、模糊半径（blur radius）及颜色（color）。
- **NSUnderlineStyleAttributeName**——如果选用这种 key，那么值应该是 NSNumber 实例，这个 NSNumber 实例里面封装的数字用来表示所需的下划线效果。

① 字面意思是“带属性的字符串”，实际可以理解为“带样式的字符串”。为了和编程中常见的“属性”一词相区隔，译文酌情保留该词英文原样。——译者注

② 实际指的是 [https://developer.apple.com/library/ios/documentation/uikit/reference/NSAttributedString\\_Uikit\\_Additions/Reference/Reference.html](https://developer.apple.com/library/ios/documentation/uikit/reference/NSAttributedString_Uikit_Additions/Reference/Reference.html)。——译者注

比方说, 解决方案 2-4 会把分段选择控件在 UIControlStateSelected 状态下的文本颜色设为白色。若用户连续两次选中某个按钮, 则将该按钮的文本颜色从白色改为红色。

#### 解决方案 2-4 创建可以响应第二次点击的 UISegmentedControl 子类

---

```

@class SecondTapSegmentedControl;

@protocol SecondTapSegmentedControlDelegate <NSObject>
- (void) performSegmentAction: (SecondTapSegmentedControl *) aDTSC;
@end

@interface SecondTapSegmentedControl : UISegmentedControl
@property (nonatomic, weak)
    id <SecondTapSegmentedControlDelegate> tapDelegate;
@end

@implementation SecondTapSegmentedControl
- (void) touchesEnded: (NSSet *) touches withEvent: (UIEvent *) event
{
    [super touchesEnded: touches withEvent: event];

    if (self.tapDelegate)
        [self.tapDelegate performSegmentAction: self];
}
@end

```

---

## 2.8 开关控件与步进控件

UISwitch 对象 (开关控件) 提供了一种简单的开 / 关切换能力, 用户可以在这两种布尔状态中选择一种。开关 (Switch) 控件包含一个可供设定的属性值, 它的名字叫作 on。该属性的值可能是 YES, 也可能是 NO, 具体的取值与控件的当前状态有关。你可以用程序代码直接修改这个属性值, 以更新控件状态, 也可以调用 setOn:animated: 方法来修改, 该方法会以动画效果展示状态改变的过程。

```

- (void) didSwitch: (UISwitch *) theSwitch
{
    self.title = [NSString stringWithFormat:@"%s",
        theSwitch.on ? @"On" : @"Off"];
}

- (void) viewWillAppear: (BOOL) animated
{
    [super viewWillAppear: animated];
    // Create the switch
    UISwitch *theSwitch = [[UISwitch alloc] init];

    // Trigger on value changes
    [theSwitch addTarget: self action: @selector(didSwitch:)

```

```

        forControlEvents:UIControlEventValueChanged];

        [self.view addSubview:theSwitch];

        // Initialize to "off"
        theSwitch.on = NO;
        self.title = @"Off";
    }

```

在上述代码中，如果开关控件的状态变了，我们就修改视图控制器的 title 属性。IB 为开关控件所提供的选项相对来说比较少。开发者可以启用该控件并设置其初始值，但除此之外，就没有太多内容可供定制了。用户调整控件的时候，它会产生 UIControlEventValueChanged 事件。



**提示** 不要把 UISwitch 实例起名叫作 switch。switch 是个 C 语言的保留字，用于实现条件控制语句。许多 iOS 开发者都容易忽视这个问题。

UIStepper 控件（步进控件）也具备与滑杆及 Switch 控件相似的功能。滑杆控件给出了一段连续的取值范围，Switch 控件提供了一种简单的开/关切换方式，而步进（Stepper）控件则介于两者之间。这种控件里面有两个按钮，一个是 -，一个是 +，它们分别用来递增及递减控件的 value 属性。

开发者一般会通过 minimumValue 及 maximumValue 给控件设定合适的取值范围，使其与应用程序的某些实际特征相符，例如音量、速度以及其他一些可以度量的值等。然而，在个别情况下，我们需要令用户在控件值达到上下限的时候，依然能够继续操作它。把控件的 wrap 属性设为 YES，即可令 Stepper 控件具备折回（wrap）能力。这样的话，用户在点击控件按钮时，如果控件值即将超过最大值，那么它就会折回到最小值；如果控件值即将低于最小值，那么它就会折回到最大值。

在默认情况下，Stepper 控件会自动重复（autorepeat），也就是说，如果用户按住其中某个按钮不放，那么控件值就会持续变化。把 autorepeat 属性设为 NO 即可禁用这一行为。而每次点击按钮时的变化量，则由 stepValue 属性来决定。不要把 stepValue 设为 0 或负数，否则会抛出运行时异常。

对于 Switch 与 Stepper 控件来说，其界面的 tint color 属性以及界面的图像都是可以定制的。开发者可以通过 onTintColor、tintColor 及 thumbTintColor 属性来定制 Switch 控件的颜色，并通过 onImage 及 OffImage 属性来定制控件的图像。UIStepper 控件的 tint color 属性可供定制，另外，其分隔条（divider）、递增按钮及递减按钮的图像也可以按状态来分别配置。

## 2.9 解决方案：编写 UIControl 的子类

UIKit 提供了许多内置的控件，开发者可以直接将其运用到应用程序里。这些控件包括

Button（按钮）、Switch 及 Slider 等，但我们不应就此止步。我们不要总是局限在苹果公司所提供的这些控件里，而是应该试着创建自己的控件。

解决方案 2-5 演示了如何通过继承 UIControl 的方式来从头开始构建新的控件。该范例创建了一种简单的颜色选取器（color picker）。用户可以触摸该控件，或在控件内拖曳，以选取颜色。当用户手指在屏幕上左右移动时，颜色的色相（hue）会改变；而上下移动时，其饱和度（saturation）则会变化。颜色的亮度与不透明程度固定为 100%。

#### 解决方案 2-5 构建自定义的颜色选取控件

```
@implementation ColorControl
- (instancetype)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self)
    {
        self.backgroundColor = [UIColor grayColor];
    }
    return self;
}

- (void)updateColorFromTouch:(UITouch *)touch
{
    // Calculate hue and saturation
    CGPoint touchPoint = [touch locationInView:self];
    float hue = touchPoint.x / self.frame.size.width;
    float saturation = touchPoint.y / self.frame.size.height;

    // Update the color value and change background color
    self.value = [UIColor colorWithHue:hue
                                saturation:saturation brightness:1.0f alpha:1.0f];
    self.backgroundColor = self.value;
    [self sendActionsForControlEvents:UIControlEventValueChanged];
}

// Continue tracking touch in control
- (BOOL)continueTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{
    // Test if drag is currently inside or outside
    CGPoint touchPoint = [touch locationInView:self];
    if (CGRectContainsPoint(self.frame, touchPoint))
    {
        // Update color value
        [self updateColorFromTouch:touch];

        [self sendActionsForControlEvents:
            UIControlEventTouchDragInside];
    }
    else
```



```

    {
        [self sendActionsForControlEvents:
         UIControlEventTouchDragOutside];
    }

    return YES;
}

// Start tracking touch in control
- (BOOL)beginTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{
    // Update color value
    [self updateColorFromTouch:touch];

    // Touch Down
    [self sendActionsForControlEvents:UIControlEventTouchDown];

    return YES;
}

// End tracking touch
- (void)endTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{
    // Test if touch ended inside or outside
    CGPoint touchPoint = [touch locationInView:self];
    if (CGRectContainsPoint(self.bounds, touchPoint))
    {
        // Update color value
        [self updateColorFromTouch:touch];

        [self sendActionsForControlEvents:
         UIControlEventTouchUpInside];
    }
    else
    {
        [self sendActionsForControlEvents:
         UIControlEventTouchUpOutside];
    }
}

// Handle touch cancel
- (void)cancelTrackingWithEvent:(UIEvent *)event
{
    [self sendActionsForControlEvents:UIControlEventTouchCancel];
}

@end

```

这确实是个非常简单的控件，因为除了获取触摸点的  $x$ 、 $y$  坐标之外，没有再涉及太多的交互操作。这个基本的范例可以演示出编写 `UIControl` 子类时可能遇到的问题。

我们为什么要自己来定制控件呢？首先，这样做可以打造自己的设计风格。对于界面中放置的元件，其风格应该与应用程序的整体风格相符。如果苹果公司内置的 Switch、Slider 及其他 GUI 元件不能与你自己的程序界面自然地融合起来，那么可以通过自定义的控件来满足应用程序的需求，而且这样做还不会破坏协调一致的设计风格。

其次，我们可以创建出苹果公司未提供的控件。比方说可以设计这样一种控件，它能够展示一排星形图案，使用户可以通过滑动来评分；还可以设计这样一种控件，它弹出许多颜色不同的蜡笔，令用户可以从中选择某种颜色。这些自定义的控件使用户可以通过更多的方式来操作应用程序，而不必局限于 SDK 提供的 Button 和 Switch 等内置控件。通过编写 UIControl 的子类，我们很容易就能构建出醒目的交互元件。

最后，通过定制控件这种方式，我们可以添加一些无法直接实现或无法通过继承固有控件类而实现出来的特性。开发者只需编写少量代码，就能够从头创建出自己的 Button 和 Stepper，也就是说，可以按自己的意愿来精确地调整它们的交互行为。

自己定制的这些控件的外观应该和系统提供的那些控件有所区别。请勿违背《HIG》中的规则。假如确实需要使用与系统控件样貌相似的自定义控件，那么在向 App Store 提交程序时，可以给苹果公司留言说明此问题。你需要明确表示自己是创建了新的类，而不是使用了私有的 API，同时自己也没有以 App Store 所认定的不安全方式来访问苹果公司的对象。即便如此，评审人员也依然有可能认为程序会对终端用户“造成困惑”，从而拒绝你的提交。

## 2.9.1 创建控件

构建 UIControl 的过程一般分为四步。正如解决方案 2-5 所示，首先我们要继承 UIControl，以创建新的自定义类。然后要在新类的初始化方法中安排好控件的样貌。接下来编写一些方法，以便追踪并拦截触摸事件，最后，产生相关事件及视觉反馈效果。

几乎所有控件都会提供某种“值”。比方说，Switch 控件有 isOn 值，Slider 控件有浮点型的 value 值，Text Field 控件有 text 值。自定义的控件应该提供哪种值由开发者来定，可以是整数、浮点数、字符串，甚至可以像解决方案 2-5 这样提供颜色值。

解决方案 2-5 所使用的控件布局基本上是个彩色矩形。有些复杂的控件需要更为复杂的布局，但即便像本例这样采用非常简单的布局，我们也依然能够向用户提供可触摸的空间，并能实现出适当的反馈效果，以保持清晰的用户体验。

## 2.9.2 追踪触摸事件

UIControl 实例自己有一套应对触摸事件的方法。这些方法可用来追踪用户操作控件对象的全过程：

- **beginTrackingWithTouch:withEvent:**——如果在控件范围内发生了触摸事件，那么系统就会调用这个方法。
- **continueTrackingWithTouch:withEvent:**——如果相关的触摸事件仍然在控件范围内持续，那么系统就反复调用这个方法。
- **endTrackingWithTouch:withEvent:**——该方法用于处理事件结束前的最后一

次触摸。

■ **cancelTrackingWithEvent:**——该方法用于处理触摸操作遭到取消的情况。

为了给自定义的控件添加逻辑代码，开发者可以在 `UIControl` 子类里实现上述某个方法，也可以同时实现那四个方法。解决方案 2-5 实现了前两个方法，以此来追踪用户对控件的触摸操作，直到用户松开手指或移出控件范围。

### 2.9.3 派发控件事件

控件使用目标 - 动作组合来传达由事件所引发的变化。构建新控件时，开发者必须想好自己的对象将会产生哪些类型的事件，并添加相关代码来触发那些事件。

调用 `sendActionsForControlEvents:` 方法即可为自定义控件添加事件派发功能。该方法可以把某个事件（比方说 `UIControlEventValueChanged` 事件）发送给控件的目标。控件是通过向 `UIApplication` 单例发消息来实现这一操作的。正如苹果公司的开发文档所说，`UIApplication` 对象是所有消息的集中派发点。

无论控件类多么简单，开发者都应该尽量多支持一些事件类型，因为你无法准确预料这个类在将来的用途。把事件类型提供得完备一些可以使控件以后用起来更加灵活。对于本例这种非常简单的控件来说，解决方案 2-5 所能够派发的事件类型已经算是很广泛了。

事件的派发时机很大程度上取决于你要构建的那个控件。比方说，`Switch` 控件只关注 `touch up` 事件，因为它的值只有在这个时候才会改变。而 `Slider` 控件则不同，它的值会随着滑块的移动而改变，所以它要持续关注用户手指的移动情况，并据此更新其值。所以，开发者应该根据自己控件的实际情况来编写代码，并且要注意在触摸操作的各个阶段适当地修改控件的外观。

## 2.10 解决方案：构建评分所用的 Star Slider 控件

用户可以用手指划过 `Rating Slider`（评分滑杆）控件上面的若干图像，以此来对电影、软件等项目做出评分。在基于触摸的界面中，这是个常见的功能，但是用简单的 `UISlider` 实例却不容易把它做好，因为 `UISlider` 控件的值是浮点数。笔者构建了解决方案 2-6 中的这个选取器（分数选取器），它把代表分数的那些图案逐次排开，将用户所能选取的评分限定为大于等于 0 且小于等于图案个数的整数。当用户触摸星形图案时，控件的值会变化，而且会产生相应的事件，这样一来，应用程序就可以像对待其他 `UIControl` 子类那样来处理用户对 `Star Slider`（含有星形图案的滑杆）控件的操作了。

开发者可以选用任意图案。本例采用图 2-6 中的星形图案，不过你完全可以改用其他图案。你可以随意选择自己喜欢的图案，只要能分别为 `on` 和 `off` 这两种状态提供对应的图像就行。“心”、“宝石”、“笑脸”等图案也都可以考虑。你还可以修改本条解决方案的代码，在显示控件之前，指定预设的评分<sup>①</sup>。

① 预设几分，就会亮起几颗星。——译者注



图 2-6 解决方案 2-6 创建了一种自定义的 star slider 控件，用户每给出一分，控件中就会有一颗星亮起。我们在块里面编写了一段简单的动画代码，当控件值发生改变时，相应的星形图案会呈现出放大并还原的效果

除了简单的滑动功能，解决方案 2-6 还添加了动画效果。当控件有了新值的时候，我们就把一段简单的动画代码放在块里面，并添加到当前最右侧的那颗星上面，使其放大，然后恢复原状，这样就可以在原有的视觉效果之上再向用户提供一种即时的反馈。图 2-6 里的图案是从模拟器的屏幕中抓下来的，而用户在真正使用应用程序时，触摸的却是设备的屏幕，所以，我们要刻意把动画做得夸张一些，使反馈效果的范围能够超过用户的手指大小。笔者选了个尺寸比较小的图案，并在动画效果中将它放大到原尺寸的 150%，你可以根据自己程序的实际需求来选取大小适当的图案，并指定合适的放大倍数。

#### 解决方案 2-6 构建以图案个数来表示评分的 Star Slider 控件

```
@implementation StarSlider
- (instancetype)initWithFrame:(CGRect)aFrame
{
    self = [super initWithFrame:aFrame];
    if (self)
    {
        // Lay out five stars, with spacing between and at the ends
        float offsetCenter = WIDTH;
        for (int i = 1; i <= 5; i++)
        {
            UIImageView *imageView = [[UIImageView alloc]
                                       initWithFrame:CGRectMake(0.0f, 0.0f, WIDTH, WIDTH)];
            imageView.image = OFF_ART;
            imageView.center = CGPointMake(offsetCenter,
                                           self.intrinsicContentSize.height / 2.0f);
            offsetCenter += WIDTH * 1.5f;
            [self addSubview:imageView];
        }
        // Place on a contrasting background
        self.backgroundColor =
            [[UIColor blackColor] colorWithAlphaComponent:0.25f];
        return self;
    }
}

- (CGSize)intrinsicContentSize
{
    return CGSizeMake(WIDTH * 8.0f, 34.0f);
}

// Handle the value update for the touch point
```

```

- (void)updateValueAtPoint:(CGPoint)point
{
    int newValue = 0;
    UIImageView *changedView = nil;

    // Iterate through stars to check against touch point
    for (UIImageView *eachItem in [self subviews])
    {
        if (point.x < eachItem.frame.origin.x)
        {
            eachItem.image = OFF_ART;
        }
        else
        {
            changedView = eachItem; // last item touched
            eachItem.image = ON_ART;
            newValue++;
        }
    }

    // Handle value change
    if (self.value != newValue)
    {
        self.value = newValue;
        [self sendActionsForControlEvents:
            UIControlEventValueChanged];

        // Animate the new value with a zoomed pulse
        [UIView animateWithDuration:0.15f
            animations:^(changedView.transform =
                CGAffineTransformMakeScale(1.5f, 1.5f));
            completion:^(BOOL done){ [UIView
                animateWithDuration:0.1f
                animations:^(changedView.transform =
                    CGAffineTransformIdentity;});}];
    }
}

- (BOOL)beginTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{
    // Establish touch down event
    CGPoint touchPoint = [touch locationInView:self];
    [self sendActionsForControlEvents:UIControlEventTouchDown];

    // Calculate value
    [self updateValueAtPoint:touchPoint];
    return YES;
}

- (BOOL)continueTrackingWithTouch:(UITouch *)touch

```

```

        withEvent:(UIEvent *)event
    {
        // Test if drag is currently inside or outside
        CGPoint touchPoint = [touch locationInView:self];
        if (CGRectContainsPoint(self.frame, touchPoint))
            [self sendActionsForControlEvents:
             UIControlEventTouchDragInside];
        else
            [self sendActionsForControlEvents:
             UIControlEventTouchDragOutside];

        // Calculate value
        [self updateValueAtPoint:[touch locationInView:self]];
        return YES;
    }

- (void)endTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{
    // Test if touch ended inside or outside
    CGPoint touchPoint = [touch locationInView:self];
    if (CGRectContainsPoint(self.bounds, touchPoint))
        [self sendActionsForControlEvents:
         UIControlEventTouchUpInside];
    else
        [self sendActionsForControlEvents:
         UIControlEventTouchUpOutside];
}

- (void)cancelTrackingWithEvent:(UIEvent *)event
{
    // Cancelled touch
    [self sendActionsForControlEvents:UIControlEventTouchCancel];
}

@end

```

解决方案 2-6 在定制 `UIControl` 时所采用的方式与解决方案 2-5 类似，也是追踪触摸操作的生命期，然后在适当的时机产生事件，只不过这次的布局及视觉反馈效果与解决方案 2-5 有所区别。本条解决方案只用了少量代码就为控件添加了星形图案，并实现了反馈效果，由此可见，通过继承 `UIControl` 类来定制自己的控件确实是件非常简单的事。

## 2.11 解决方案：构建触摸转盘控件

这条解决方案将会制作一种与老式 iPod 类似的触摸转盘（Touch Wheel）控件，它提供了一种可以无限滚动的输入方式。无论是顺时针旋转还是逆时针旋转，该控件的值都会随之增加或减少。每转动一整圈（也就是 360 度），控件的值就改变 1.0。顺时针旋转会令控件值增大，而逆时针旋转则会令其减小。每次触摸时所产生的控件值会累积起来，不过开发者也

可以重置该控件，只需把其 value 属性设为 0.0 即可。虽说该属性并不是 UIControl 实例中的标准属性，但很多控件都有个名叫 value 的属性。

本条解决方案会以控件中心点为原点计算当前触摸点的坐标，并算出用户的触摸操作所对应的变化量。在用户移动手指的过程中，这段代码会算出这一次与上一次之间的角度差，并据此更新控件当前的值。比方说，如果手指在控件内旋转了 3 圈，那么新的控件值就会比原来多 3 或少 3，是增加还是减少要看移动的方向是顺时针还是逆时针。

解决方案 2-7 定义了一个简单的触摸转盘控件，它能够记录手指的旋转，但除此之外就没有太多的功能了。最初版 iPod 的滚轮 (scroll wheel) 有 5 个地方可供点击，分别是圆圈中间以及滚轮上、下、左、右四个部位。读者可以做个练习，为解决方案 2-7 中的控件添加点击功能，并使其能够产生类似按钮那样的事件（比方说 UIControlEventTouchUpInside 事件）。

### 解决方案 2-7 构建触摸转盘控件

```
@implementation ScrollWheel

// Layout the wheel
- (instancetype)initWithFrame:(CGRect)aFrame
{
    self = [super initWithFrame:aFrame];
    if (self)
    {
        // This control uses a fixed 200x200 sized frame
        CGRect f;
        f.origin = aFrame.origin;
        f.size = self.intrinsicContentSize;
        self.frame = f;

        // Add the touch wheel art
        UIImageView *imageView = [[UIImageView alloc]
            initWithImage:[UIImage imageNamed:@"wheel.png"]];
        [self addSubview:imageView];
    }
    return self;
}

- (BOOL)beginTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{
    CGPoint point = [touch locationInView:self];

    // Center point of view in own coordinate system
    CGPoint centerPt = CGPointMake(self.bounds.size.width / 2.0f,
        self.bounds.size.height / 2.0f);
    // First touch must touch the gray part of the wheel
    if (!pointInsideRadius(point, centerPt.x, centerPt.y)) return NO;
    if (pointInsideRadius(point, 30.0f, centerPt.y)) return NO;
```

```

// Set the initial angle
self.theta = getAngle([touch locationInView:self], centerPt);

// Establish touch down
[self sendActionsForControlEvents:UIControlEventTouchUpInside];

return YES;
}

- (CGSize)intrinsicContentSize
{
    return CGSizeMake(200, 200);
}

- (BOOL)continueTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{
    CGPoint point = [touch locationInView:self];

    // Center point of view in own coordinate system
    CGPoint centerPt = CGPointMake(self.bounds.size.width / 2.0f,
        self.bounds.size.height / 2.0f);

    // Touch updates
    if (CGRectContainsPoint(self.frame, point))
        [self sendActionsForControlEvents:
            UIControlEventTouchDragInside];
    else
        [self sendActionsForControlEvents:
            UIControlEventTouchDragOutside];

    // Falls outside too far, with boundary of 50 pixels?
    if (!pointInsideRadius(point, centerPt.x + 50.0f, centerPt))
        return NO;

    float newtheta = getAngle([touch locationInView:self], centerPt);
    float dtheta = newtheta - self.theta;

    // correct for edge conditions
    int ntimes = 0;
    while ((ABS(dtheta) > 300.0f) && (ntimes++ < 4))
    {
        if (dtheta > 0.0f)
            dtheta -= 360.0f;
        else
            dtheta += 360.0f;
    }

    // Update current values
    self.value -= dtheta / 360.0f;
    self.theta = newtheta;
}

```



```

// Send value changed alert
[self sendActionsForControlEvents:UIControlEventValueChanged];
return YES;
}

- (void)endTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{
    // Test if touch ended inside or outside
    CGPoint touchPoint = [touch locationInView:self];
    if (CGRectContainsPoint(self.bounds, touchPoint))
        [self sendActionsForControlEvents:
            UIControlEventTouchUpInside];
    else
        [self sendActionsForControlEvents:
            UIControlEventTouchUpOutside];
}

- (void)cancelTrackingWithEvent:(UIEvent *)event
{
    // Cancel
    [self sendActionsForControlEvents:UIControlEventTouchCancel];
}

@end

```

## 2.12 解决方案：创建拉曳控件

我们可以假想屏幕上方有条绳索，当用户使劲向下拉拽它的时候，会响起铃声，或是导致系统通过控件的目标 - 动作机制触发某种事件。例如，该事件可能会展示出一个辅助的视图（secondary view），也可能启动下载操作，还可能会播放视频，等等。解决方案 2-8 所构建的这个控件的样子像是一条丝带。用户必须从其顶端开始操作它，而且还要向下拉拽足够大的距离，才能触发相关的事件。其后，丝带长度会恢复到原状，以等待下一次操作。

图 2-7 演示了本解决方案所构建的控件，它紧贴着辅助视图的下沿。用户向下拉拽，即可看到辅助视图，若是再次拉曳，则会使辅助视图重新回到屏幕之外。

### 2.12.1 为控件添加提示效果

本解决方案的一个难点是如何告诉用户丝带图案是可以拉曳的。用户看到这个红色图形之后，不

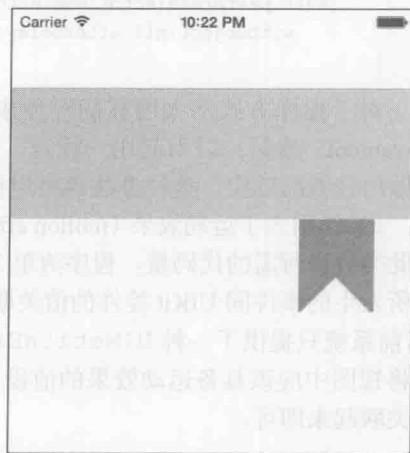


图 2-7 用户对这个丝带状控件的拉曳量必须达到某个最小距离才能起到效果，丝带的长度随后会复原。每次成功操作之后，程序都会把 `UIControlEventValueChanged` 事件发给目标 - 动作中的相关目标对象

一定能立刻想到这是个可供操作的控件。

开发者 Matthijs Hollemans 想出了一个简单的办法可以解决此问题。那就是，在用户操作控件之前，令丝带图案上下抖动 (wiggle) 几次，每次抖动之间相隔数秒。这样做可以使用户注意到这是个控件，等用户能够正确操作该控件之后，就尽快去掉这种效果。如果用户经常使用这个程序，那么可以考虑添加一项系统配置，令其可以关闭提示效果。

```
- (void)wiggle
{
    if (wiggleCount++ > 3) return;

    // Wiggle slightly
    [UIView animateWithDuration:0.25f animations:^(()) {
        pullImageView.center = CGPointMake(
            pullImageView.center.x,
            pullImageView.center.y + 10.0f);
    } completion:^(BOOL finished){
        [UIView animateWithDuration:0.25f animations:^(()) {
            pullImageView.center = CGPointMake(
                pullImageView.center.x,
                pullImageView.center.y - 10.0f);
        }];
    }];

    // Repeat until the count is overridden or it wiggles 3 times
    [self performSelector:@selector(wiggle)
        withObject:nil afterDelay:4.0f];
}
```

对于操作方式不太明显的控件来说，可以添加基于加速计的移动 (accelerometer-based movement) 效果，以引起用户注意。开发者 Charles Choi 建议，应该令丝带随着设备的移动而做出轻微的反应。这种办法也可以使用户明白控件的用法。

iOS 7 引入了运动效果 (motion effect, 动画效果)，它用一套新的声明式 API 大幅缩减了实现此类效果所需的代码量。程序清单 2-1 就演示了这么一种运动效果，它可以把设备加速计上面所发生的事件同 UIKit 控件的值关联起来。开发者只需创建 UIMotionEffect 子类的实例 (目前系统只提供了一种 UIMotionEffect 子类，即 UIInterpolatingMotionEffect)，并将视图中应该具备运动效果的值设为 keyPath，然后把 UIMotionEffect 实例和目标视图关联起来即可。

程序清单 2-1 添加运动效果

```
- (void)startMotionEffects
{
    UIInterpolatingMotionEffect *motionEffectX =
        [[UIInterpolatingMotionEffect alloc]
         initWithKeyPath:@"center.x"
         type:UIInterpolatingMotionEffectTypeTiltAlongHorizontalAxis];
    UIInterpolatingMotionEffect *motionEffectY =
        [[UIInterpolatingMotionEffect alloc]
```

```

initWithKeyPath:@"center.y"
type:UIInterpolatingMotionEffectTypeTiltAlongVerticalAxis];
motionEffectX.minimumRelativeValue = @-15.0;
motionEffectX.maximumRelativeValue = @15.0;
motionEffectY.minimumRelativeValue = @-15.0;
motionEffectY.maximumRelativeValue = @15.0;
motionEffectsGroup = [[UIMotionEffectGroup alloc] init];
motionEffectsGroup.motionEffects =
    @[motionEffectX, motionEffectY];
[pullImageView addMotionEffect:motionEffectsGroup];
}

```

你可以从苹果公司的程序中获取灵感。苹果公司在 iOS 系统中制作了许多这样的提示效果，例如对于滑动解锁操作来说，它会通过一段简单的动画效果，用提示性的文本写出应该滑动的部位以及所需的滑动方式。

### 2.12.2 测试触摸

解决方案 2-8 以两种方式来限定控件内可供操作的范围。首先，我们判断用户的触摸点是否在图案范围内。如果用户点击的地方不在丝带图案的范围内，那么就不处理这次触摸，而控件也不会响应它。即便触摸点确实在控件的框架之中，但只要不在图案范围内，我们就不做出响应。其次，根据丝带图案的位图信息来确认用户触摸的是不是图案中不透明的部分。从图 2-7 中可以看到，丝带图案的下方有个倒 V 字形的缺口。在缺口里发生的触摸操作不会使控件启动后续的追踪过程。我们要在解决方案的代码中比对触摸点和图案中的相关像素。假如不透明度小于等于 85，或者说透明度已达 67% 左右，那么程序就不认为用户触摸了控件。

解决方案 2-8 构建画有丝带图案的可拉拽式控件

```

- (BOOL)beginTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{
    // Establish touch down event
    CGPoint touchPoint = [touch locationInView:self];
    CGPoint ribbonPoint = [touch locationInView:pullImageView];

    // Find the data offset in the image
    Byte *bytes = (Byte *) ribbonData.bytes;
    uint offset = alphaOffset(ribbonPoint.x, ribbonPoint.y,
        pullImageView.bounds.size.width);

    // Test for containment and alpha value to disallow touches
    // outside the ribbon and inside the notched area

    if (CGRectContainsPoint(pullImageView.frame, touchPoint) &&
        (bytes[offset] > 85))

```

```

    {
        [self sendActionsForControlEvents:UIControlEventTouchUpInside];
        touchDownPoint = touchPoint;
        return YES;
    }

    return NO;
}

- (BOOL)continueTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{
    // Once the user has interacted, don't wiggle any more
    wiggleCount = CGFLOAT_MAX;

    // Test for inside/outside touches
    CGPoint touchPoint = [touch locationInView:self];
    if (CGRectContainsPoint(self.frame, touchPoint))
        [self sendActionsForControlEvents:
            UIControlEventTouchDragInside];
    else
        [self sendActionsForControlEvents:
            UIControlEventTouchDragOutside];

    // Adjust art based on the degree of drag
    CGFloat dy = MAX(touchPoint.y - touchDownPoint.y, 0.0f);
    dy = MIN(dy, self.bounds.size.height - 75.0f);
    pullImageView.frame = CGRectMake(10.0f,
        dy + 75.0f - ribbonImage.size.height,
        ribbonImage.size.width, ribbonImage.size.height);

    // Detect if travel has been sufficient to trigger everything
    if (dy > 75.0f)
    {
        // It has. Play a click, trigger the callback, and roll
        // the view back up.
        [self playClick];
        [UIView animateWithDuration:0.3f animations:^( ){
            pullImageView.frame = CGRectMake(10.0f,
                75.0f - ribbonImage.size.height,
                ribbonImage.size.width,
                ribbonImage.size.height);
        } completion:^(BOOL finished){
            [self sendActionsForControlEvents:
                UIControlEventValueChanged];
        }];

        // No more interaction needed or allowed
        return NO;
    }
}

```

```

// Continue interaction
return YES;
}

- (void)endTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{
    // Test if touch ended inside or outside
    CGPoint touchPoint = [touch locationInView:self];
    if (CGRectContainsPoint(self.bounds, touchPoint))
        [self sendActionsForControlEvents:
            UIControlEventTouchUpInside];
    else
        [self sendActionsForControlEvents:
            UIControlEventTouchUpOutside];

    // Roll back the ribbon, regardless of where the touch ended
    [UIView animateWithDuration:0.3f animations:^(
        pullImageView.frame = CGRectMake(10.0f,
            75.0f - ribbonImage.size.height,
            ribbonImage.size.width, ribbonImage.size.height);
    )];
}

// Handle cancelled tracking
- (void)cancelTrackingWithEvent:(UIEvent *)event
{
    [self sendActionsForControlEvents:UIControlEventTouchCancel];
}

```

解决方案中的范例代码并没有考虑拉伸之后的图案，它假定图案与屏幕上绘制出来的内容是 1 比 1 的关系。假如你要调整控件图案的尺寸，可以把透明度测试去掉，或是对代码进行改编。

启动追踪 (tracking) 过程之后，控件就会开始关注触摸点的移动以及用户手指的向上、向下拖曳操作了。在本解决方案中，如果触摸点的移动距离超过 75 个点 (point)，那么就触发控件事件，它会向其目标对象发送 `UIControlEventValueChanged` 事件。严格来说，该控件并没有 Value (值) 这个概念，但若改成触发 `UIControlEventTouchUpInside` 事件，则似乎又与控件的操作方式不太相符。

如果用户的拉拽操作已经到位，那么 `continueTrackingWithTouch` 方法就返回 NO，意思是说追踪流程已经结束，而控件也已经完成了它在这次操作中的任务。若是手指移动的距离没有达到触发控件事件的最小拉拽距离，或是用户在尚未拉拽到位时就提前停止了操作，那么程序会把控件恢复到开始时的样子。这会令丝带图案复原，使用户明白现在可以进行下一次操作了。

## 2.13 解决方案：构建自定义的锁定控件

写完本书上一版之后，笔者曾经为某次开发者聚会制备了图 2-8 中的锁定控件（Lock Control）。当时很多人都要求把它的制作过程放在本书这一版里面。以 UIControl 为基础来构建这个控件非常容易。它由四部分组成：背景图版（backdrop）、表示锁定和解锁状态的图像、滑块（thumb）以及拖曳滑块时所沿的轨道（drag track）。

解决方案 2-9 列出了实现控件行为所用的代码。这条解决方案对操作的判定非常宽松。只要触摸操作发生在轨道及滑块周围的 20 点以内就算有效。这是个非常粗放（Spartan）的控件，它留出了很大一片地方（相当于常人指尖的一半大小），使用户可以更加从容地执行解锁操作。

与判定触摸时所用的宽松程度类似，用户只需把滑块拖过大约 75% 的轨道长度，即可完成解锁操作。这里也留下了一定的余地，我们既能确保用户真的要完成解锁操作，同时又不至于令用户因为精确度要求太过严苛而感觉麻烦。滑块的回弹效果也是经历了好几次用户测试才调整好的，如果用户在尚未完成解锁之前就松开了滑块，那么它会向左恢复到原来的位置。该控件在用户解锁之后的消失过程需要持续半秒钟，笔者设定的这个时长比一般控件的变化速度要稍微长一点。比方说，键盘通常只需要过 1/3 秒就会显示出来。



图 2-8 这是个简单的锁定控件，如果用户能将滑块拖过至少 3/4 的轨道长度，那么该控件就会解锁并消失

### 解决方案 2-9 创建锁定控件

```
- (BOOL)beginTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{
    // Test touches for start conditions
    CGPoint touchPoint = [touch locationInView:self];
    CGRect largeTrack =
        CGRectInset(trackView.frame, -20.0f, -20.0f);
    if (!CGRectContainsPoint(largeTrack, touchPoint))
        return NO;
    touchPoint = [touch locationInView:trackView];
    CGRect largeThumb =
        CGRectInset(thumbView.frame, -20.0f, -20.0f);
    if (!CGRectContainsPoint(largeThumb, touchPoint))
        return NO;
    // Begin tracking
    [self sendActionsForControlEvents:UIControlEventTouchDown];
    return YES;
}
```

```

- (BOOL)continueTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{
    // Strayed too far out?
    CGPoint touchPoint = [touch locationInView:self];
    CGRect largeTrack =
        CGRectInset(trackView.frame, -20.0f, -20.0f);
    if (!CGRectContainsPoint(largeTrack, touchPoint))
    {
        // Reset on failed attempt
        [UIView animateWithDuration:0.2f animations:^(){
            NSLayoutConstraint *constraint =
                [trackView constraintNamed:THUMB_POSITION_TAG];
            constraint.constant = 0;
            [trackView layoutIfNeeded];
        }];
        return NO;
    }

    // Track the user movement by updating the thumb
    touchPoint = [touch locationInView:trackView];
    [UIView animateWithDuration:0.1f animations:^(){
        NSLayoutConstraint *constraint =
            [trackView constraintNamed:THUMB_POSITION_TAG];
        constraint.constant = touchPoint.x;
        [trackView layoutIfNeeded];
    }];
    return YES;
}

```

```

- (void)endTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{
    // Test if touch ended with unlock
    CGPoint touchPoint = [touch locationInView:trackView];
    if (touchPoint.x > trackView.frame.size.width * 0.75f)
    {
        // Complete by unlocking
        _value = 0;
        self.userInteractionEnabled = NO;
        [self sendActionsForControlEvents:
            UIControlEventValueChanged];

        // Fade away and remove
        [UIView animateWithDuration:0.5f animations:
            ^(){self.alpha = 0.0f;}
            completion:
            ^(BOOL finished){[self removeFromSuperview];
        }];
    }
}

```

```

else
{
    // Reset on failed attempt
    [UIView animateWithDuration:0.2f animations:^( ) {
        NSLayoutConstraint *constraint = [trackView constraintNamed:
            [trackView constraintNamed:
                THUMB_POSITION_TAG];
        constraint.constant = 0;
        [trackView layoutIfNeeded];
    }];
}

if (CGRectContainsPoint(trackView.bounds, touchPoint))
{
    [self sendActionsForControlEvents:
        UIControlEventTouchUpInside];
}
else
{
    [self sendActionsForControlEvents:
        UIControlEventTouchUpOutside];
}
}

- (void)cancelTrackingWithEvent:(UIEvent *)event
{
    [self sendActionsForControlEvents:
        UIControlEventTouchCancel];
}

```

## 添加页面指示器控件

UIPageControl 类会提供许多圆点，并将其中一个加亮，用来表示当前所显示的内容是多页视图中的哪一页。iOS 的 SpringBoard 主屏幕就用到了这种控件，大家可以看到屏幕底部显示的那几个小圆点。不过 UIPageControl 类有个缺点，就是操控起来非常困难，用户很难准确点击该控件，这通常会使他们觉得麻烦。所以，如果要使用这个控件，就应该另外提供一套切换页面的方式，把 UIPageControl 只当成个指示器来用就好。

图 2-9 演示了含有 3 个页面的 Page Control（页面控件）。在指示当前页面的那个亮色小圆点左方或右方点击，就会触发 UIControlEventValueChanged 事件，此时系统将会调用开发者所设定的动作。你可以通过 currentPage 属性查询控件的当前值，也可以通过设置 numberOfPages 属性来修改可用的页面数量。SpringBoard 把表示页面的圆点个数限制为 9 个，不过你自己的应用程序可以使用更多的小圆点，尤其是在横屏模式下。



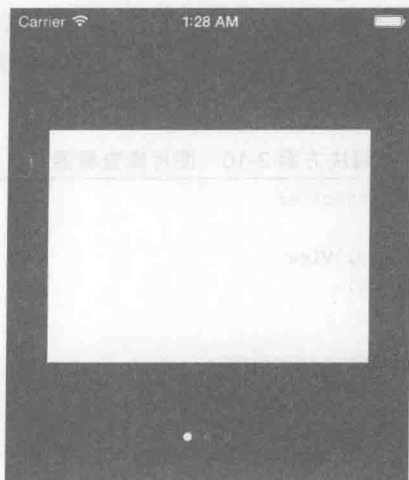


图 2-9 在需要显示多个页面的时候，我们可以通过 `UIPageControl` 类提供一种能够接受用户操作的指示器。用户在当前亮起的小圆点左方或右方点击，即可选定新的页面，至少在理论上是这样的。不过，`UIPageControl` 控件很难操作，它需要非常高的点击精确度才行，所以，其响应能力比较低

## 2.14 解决方案：图片库查看器

解决方案 2-10 用 `UIScrollView` 实例来显示多张图像，其效果如图 2-10 所示。用户可以通过滑动（swipe）来浏览图片，而页面指示器也会随之更新。另外，用户还可以直接在页面控件上点击，使 `UIScrollView` 把用户所选的页面展示出来。我们通过目标-动作机制给 `UIPageControl` 控件添加了一个回调方法，同时又在 `UIScrollView` 控件的委托里编写了另外一个回调方法，这样就实现出了这套两者联动的操作方案。由于每个回调方法都会更新另外一个对象，所以两个对象能够紧密结合起来。

使用 `UIPageControl` 控件时，开发者常犯的错误是没有把控件拓展到足够宽，也就是没有在小圆点的左右两侧留下可供用户点击的区域。该控件的固有大小与它的可见尺寸非常接近，这就严重限制了可供用



图 2-10 iOS 7 讲究内容至上。在某些应用程序的界面元件中，只有页面控件和 iOS 状态栏是可见的，二者都叠放在屏幕内容上方

户点击的区域。假如直接将没有拓宽的控件居中，那么控件就会很难操作。除非用户操作 `UIPageControl` 时可能与其他触摸元件相冲突，否则开发者应该设定 `Auto Layout` 约束或设定其框架，使之与上级视图同宽。

#### 解决方案 2-10 图片库查看器

```
@implementation TestBedViewController
{
    PagedImageScrollView *scrollView;
    UIPageControl *pageControl;
}

- (UIStatusBarStyle)preferredStatusBarStyle
{
    return UIStatusBarStyleLightContent;
}

- (void)loadView
{
    self.view = [[UIView alloc] init];
    self.view.backgroundColor = [UIColor blackColor];
    self.navigationController.navigationBarHidden = YES;

    scrollView = [[PagedImageScrollView alloc] init];
    scrollView.delegate = self;
    [self.view addSubview:scrollView];
    PREPCONSTRAINTS(scrollView);
    ALIGN_VIEW_LEFT(self.view, scrollView);
    ALIGN_VIEW_RIGHT(self.view, scrollView);
    ALIGN_VIEW_TOP(self.view, scrollView);
    ALIGN_VIEW_BOTTOM(self.view, scrollView);
    scrollView.images = @[UIImage imageNamed:@"bird",
        UIImage imageNamed:@"ladybug",
        UIImage imageNamed:@"flowers",
        UIImage imageNamed:@"sheep"]];

    pageControl = [[UIPageControl alloc] init];
    pageControl.numberOfPages = scrollView.images.count;
    pageControl.currentPage = 0;
    pageControl.pageIndicatorTintColor = [UIColor grayColor];
    pageControl.currentPageIndicatorTintColor =
        [UIColor redColor];
    [pageControl addTarget:self
        action:@selector(handlePageControlChange:)
        forControlEvents:UIControlEventValueChanged];
    [self.view addSubview:pageControl];
    PREPCONSTRAINTS(pageControl);
    ALIGN_VIEW_LEFT(self.view, pageControl);
    ALIGN_VIEW_RIGHT(self.view, pageControl);
    ALIGN_VIEW_BOTTOM_CONSTANT(self.view, pageControl, -20);
}
```

```

// Update the scrollView after page control interaction
- (void)handlePageControlChange:(UIPageControl *)sender
{
    CGFloat offset =
        scrollView.frame.size.width * pageControl.currentPage;
    [scrollView setContentOffset:CGPointMake(offset, 0)
        animated:YES];
}

// Update the page control after scrolling
- (void)scrollViewDidEndDecelerating:(id)sender
{
    CGFloat distance = scrollView.contentOffset.x /
        scrollView.contentSize.width;
    NSInteger page = distance * pageControl.numberOfPages;
    pageControl.currentPage = page;
}
@end

```

iOS 7 的设计风格主要围绕着内容而展开，所以对 UI 控件修饰得稍微少了一些。这个图片库查看器程序完全利用了屏幕上可以显示内容的区域，这其中也包括状态栏后面的那块地方。iOS 7 的状态栏已经不再支持半透明（translucent）和不透明（opaque）这两种风格了，现在只支持透明风格。iOS 提供了两种样式，供开发者在显示亮色及暗色内容时切换状态栏的样貌。令视图控制器类的 `preferredStatusBarStyle` 方法返回一种 `UIStatusBarStyle`，即可改变状态栏的样式。

开发者可以根据当前所显示的图片来修改状态栏的风格以及页面指示器的 tint color，使控件与图片能够更好地融合起来。修改时所参照的标准可以是图片的平均颜色，也可以是与之相似的其他量化指标。这项高级功能就留给读者当作练习吧。



**提示** 在实现 `UIScrollView` 时，想通过 Auto Layout（自动布局）来排布 UI 元件是相当困难的。苹果公司所提供的《Technical Note TN2154》开发文档（[http://developer.apple.com/library/ios/#technotes/tn2154/\\_index.html](http://developer.apple.com/library/ios/#technotes/tn2154/_index.html)）描述了两种方式。解决方案 2-10 采用的是混合方式。本书第 5 章将会深入讲解 Auto Layout。

## 2.15 构建工具栏

只要编写几个简单的宏（macro），我们就可以用代码非常方便地定义并排布工具栏了。下面这几个宏分别返回 4 种样式的按钮条目（button item）<sup>②</sup>，如果需要更多的样式，只需对这段代码稍加改编即可。这些宏都是在 ARC（automatic reference counting，自动引用计数）环

② 为了与 `UIButton` 这种普通的按钮控件相区别，工具栏上的按钮状图形称作 button item。在不引起混淆的情况下，译文酌情将其译为“按钮”，或保留 button item 及 item 英文原样。——译者注

境下使用的。假如采用手动执行 retain 及 release (manual retain-release, 简称 MRR) 的开发方式, 那么请记得加上适当的 autorelease 调用语句:

```
#define BARBUTTON(TITLE, SELECTOR) [[UIBarButtonItem alloc] \
    initWithTitle:TITLE style:UIBarButtonItemStylePlain\
    target:self action:SELECTOR]
#define IMGBARBUTTON(IMAGE, SELECTOR) [[UIBarButtonItem alloc] \
    initWithImage:IMAGE style:UIBarButtonItemStylePlain \
    target:self action:SELECTOR]
#define SYSBARBUTTON(ITEM, SELECTOR) [[UIBarButtonItem alloc] \
    initWithBarButtonSystemItem:ITEM \
    target:self action:SELECTOR]
#define CUSTOMBARBUTTON(VIEW) [[UIBarButtonItem alloc] \
    initWithCustomView:VIEW]
```

以上分别表示文本 (text)、图像 (image)、系统 (system) 及自定义视图 (custom view) 这 4 种风格的 button item。每个宏都提供了一种可以放入 UIToolbar 的 UIBarButtonItem。程序清单 2-2 演示了这些宏所制作的按钮效果, 大家可以看到每一种风格的样貌, 也可以看到按钮之间的间隔区域 (spacer)。开发者还可以像程序清单 2-2 这样, 在工具栏上添加自定义视图。在本例中, 我们把 UISwitch 实例当成 button item 添加到工具栏之中, 效果如图 2-11 所示。

程序清单 2-2 用代码创建工具栏

```
@implementation TestBedViewController
- (void)action
{
    // no action actually happens
}

- (void)loadView
{
    self.view = [[UIView alloc] init];
    self.view.backgroundColor = [UIColor whiteColor];
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    UIToolbar *tb = [UIToolbar alloc] init;
    [self.view addSubview:tb];
    PREPCONSTRAINTS(tb);
    ALIGN_VIEW_BOTTOM(self.view, tb);
    ALIGN_VIEW_LEFT(self.view, tb);
    ALIGN_VIEW_RIGHT(self.view, tb);
    NSMutableArray *tbItems = [NSMutableArray array];

    // Set up the items for the toolbar
    [tbItems addObject:
        BARBUTTON(@"Title", @selector(action))];
```

```

[tbItems addObject:
    SYSBARBUTTON(UIBarButtonSystemItemFlexibleSpace,
        nil)];

[tbItems addObject:
    SYSBARBUTTON(UIBarButtonSystemItemAdd,
        @selector(action))];

[tbItems addObject:
    SYSBARBUTTON(UIBarButtonSystemItemFlexibleSpace,
        nil)];

[tbItems addObject:
    IMGBARBUTTON([UIImage imageNamed:@"star.png"],
        @selector(action))];

[tbItems addObject:
    SYSBARBUTTON(UIBarButtonSystemItemFlexibleSpace,
        nil)];

[tbItems addObject:
    CUSTOMBARBUTTON([[UISwitch alloc] init])];

[tbItems addObject:
    SYSBARBUTTON(UIBarButtonSystemItemFlexibleSpace,
        nil)];

[tbItems addObject:
    IMGBARBUTTON([UIImage imageNamed:@"moon.png"],
        @selector(action))];

tb.items = tbItems;
}
@end

```



图 2-11 工具栏上的 button item 是可以定制的，比方说，我们可以把开关控件添加到其中

除了上述定义的这四个宏之外，只有一种 button item 是需要由开发者手动去添加的，那就是 fixed-space（宽度固定的间隔区域）。你需要设定这个 item 的 width 属性，以定义它所占据的宽度。笔者最后总结了设计工具栏时所应注意的几个问题：

- 要为 fixed space 设定宽度。在各种 UIBarButtonItem 中，只有 UIBarButtonSystemItemFixedSpace 这一种 item 可以指定宽度。所以在创建了 spacer 条目之后，应该先设定其宽度，然后再把它添加到保存 button item 的数组之中。
- 用一个宽度可变的间隔区域来实现左对齐或右对齐。如果在 item 列表的开头添加一个 UIBarButtonSystemItemFlexibleSpace，那么其余的 item 就会右对齐。若

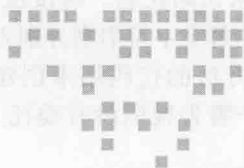
添加到众 item 尾部，则其余的 item 就会左对齐。要是头尾各添加一个，那么剩下的 item 就会居中。

- 要处理好消失的 button item。有时我们要根据程序的状况来隐藏某个 button item，假如没有使用布局限制，那么请勿通过 `UIBarButtonItemSystemItemFlexibleSpace` 来隐藏 item。此时我们应该把待隐藏的 item 替换成尺寸与之相同的固定宽度的间隔区域。这样做既可以保留原有布局，又能使其他图标的位置在 item 消失之前与消失之后保持不变。
- 导航栏上面也可以有多个 button item。开发者可以向导航栏及其 `UINavigationController` 对象添加 button item 数组。我们可以把 button item 数组添加到 `UINavigationController` 里（例如，`self.navigationController.rightBarButtonItemItems = anArray`），而不是像平常那样只添加单个 button item，这样一来，就可以实现出类似工具栏的效果了。在构建工具栏时，我们可以指定 `UIBarButtonItem` 的提示（hint），也可以通过宽度可变的间隔区域来调整各 button item 的布局，而在构建导航栏及其 `UINavigationController` 时，这些布局手段同样有效。

## 2.16 小结

本章介绍了多种交互方式，读者学到了如何利用控件来尽量提升应用程序的交互能力。在开始学习下一章之前，大家可以先回顾下列问题：

- 不要只把控件当成 `UIControl` 类来看待，而是要记得，它其实也是个 `UIView`。我们可以为其添加子视图、对其尺寸进行缩放、给它指定动画效果、使它在屏幕上移动或是为其贴上标签，以备后用。
- 开发者可以通过 Core Graphics 及 Quartz 2D 来构建视觉元件。只需在 SDK 所提供的类上稍微添加一些醒目的实时演算效果，即可令控件更加美观。
- 开发者可以通过 UIKit 中的相关方法，利用 `NSDictionary` 来配置带属性的字符串，并据此指定控件文本的特征。你可以选择符合自己设计风格的字体、线条样式、段落样式、颜色和阴影效果等。
- 如果 iOS SDK 没有提供你所需的控件，那么可以根据既有控件来改编，也可以从头构建一种新的控件。
- 苹果公司自己的 UI 设计风格是非常精良的。创建新的交互方式时，不妨参考一下苹果公司现有的样例，比方说，如果我们要在执行删除操作之前向用户提供一个确认按钮，可以参照苹果公司的做法来设计。
- 创建程序界面时，IB 并不一定是最好用的方式。比方说，我们可以直接在 Xcode 里编写代码，并以此构建工具栏，这么做要比在 IB 里面手工调整每个元件更省时间。



## 第3章 Chapter 3

# 提醒用户

有时，程序需要吸引用户注意。比方说，当程序要获取新数据或要改变状态时就是如此。有些情况下，我们要在事情发生之前先提醒用户等待一段时间；而有些情况下，则要在等待结束之后提醒用户，现在可以回来操作程序了。iOS 提供了许多种向用户展示信息的方式，例如警示窗、进度条、提示音等。本章要告诉大家如何来实现这些提示功能，令读者能够以更多的方式提醒用户。你会看到一些实用的范例，而通过这些与提示功能有关的类，我们可以明白怎样在适当的时机吸引用户的注意力。

## 3.1 直接向用户弹出警告视图

开发者可以通过警告视图（Alert View）来告诉用户一些事情。我们可以在程序中弹出 `UIAlertView`，也可以把 `UIActionSheet` 显示在其他视图上面，以此来告知用户相关的消息。这些轻量级的类可用来在程序中添加双向对话框。警告视图能够以视觉信息的形式对用户“说话”，并提醒用户答复它们。开发者向用户展示警告视图，在得到用户确认之后，将其隐藏起来，并继续处理其他任务。

你认为警告视图仅仅是一段信息加上一个 OK 按钮吗？请再好好想想。其实，`UIAlertView` 对象的用法十分丰富。比方说，我们可以用它来构建进度指示器，用它来输入文本，也可以通过它进行查询。本章的各条解决方案将会展示许多种实用的警告视图，有一些是系统提供的，还有一些是笔者自己定义的，它们都能够用在你自己的程序中。

### 3.1.1 构建简单的警告视图

想要构建警告视图，可以先分配 `UIAlertView` 对象，然后用一条标题以及一个包含按

钮标题的数组来初始化它。与按钮的标题一样，UIAlertView 的标题也是个 NSString。而按钮数组中的每个字符串则分别表示应该显示在 UIAlertView 之中的各个按钮。

下面这个方法的代码用来创建并显示最为简单的警告视图，它是一条标题加上一个 OK 按钮。这个警告视图没有委托及回调，所以，当用户点击按钮之后，它的生命期就终结了：

```
- (void) showAlert:(NSString *) theMessage
{
    UIAlertView *av = [[UIAlertView alloc] initWithTitle:@"Title"
        message:theMessage
        delegate:nil
        cancelButtonTitle:@"OK"
        otherButtonTitles:nil];
    [av show];
}
```

如果还想加入其他按钮，那么就通过 otherButtonTitles: 参数将其传入。请注意，在传给该参数的那一系列按钮的最后，必须有个 nil。这个 nil 会告诉 initWithTitle 方法按钮已经指定完了。接下来这个方法会创建一种包含三个按钮的警告视图，这三个按钮分别是：Cancel、Option 以及 OK。由于下面这段代码也没有指明委托，所以我们没有办法在稍后使用这个警告视图，也无法判断用户到底点击了哪个按钮。它只会把警告视图显示出来，等用户点击某个按钮之后，窗口就会消失，而且不会有其他效果：

```
- (void) showAlert:(NSString *) theMessage
{
    UIAlertView *av = [[UIAlertView alloc] initWithTitle:@"Title"
        message:theMessage
        delegate:nil
        cancelButtonTitle:@"Cancel"
        otherButtonTitles:@"Option", @"OK", nil];
    [av show];
}
```

在编写警告视图的时候，一定要利用好窗口空间。如果添加的按钮多于两个，那么系统就会以多行模式来显示。图 3-1 演示了两种警告视图，一种是两个按钮并排摆放，另一种是三个按钮从上到下摆放。无论何时，都不要给警告视图添加三四个以上的按钮。按钮越少，效果越好。一到两个是最理想的。如果想使用更多按钮，那么请考虑改用 UIActionSheet 对象而不是 UIAlertView 来实现，本章稍后将会讨论 UIActionSheet。

UIAlertView 对象提供了一套简单的高亮机制，可以突出显示窗口中的默认 (default) 按钮。由图 3-1 可以看出，哪个按钮呈现高亮与窗口中的按钮个数有关。如果只有两个按钮，那么右侧的按钮呈现高亮。也就是说，在初始化这种警告视图的时候，传给 otherButtonTitles 的那个按钮会呈现高亮。如果按钮超过两个，那么最底部的按钮呈现高亮。一般来说，此按钮就是开发者传给 cancelButtonTitle 参数的那个按钮。假如没有提供该参数，那么 otherButtonTitles 中的最后一个按钮就充当默认按钮。总之，传给 cancelButtonTitle 的那个按钮会出现在窗口底部或左侧。



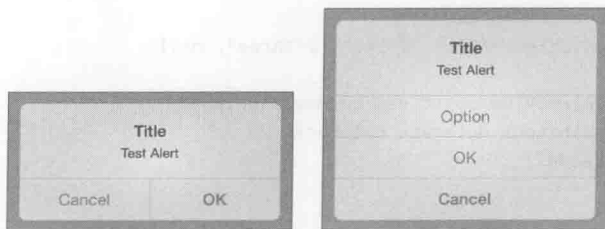


图 3-1 有一到两个按钮时，UIAlertView 的显示效果最好（如左图所示）。如果按钮多于两个，那么按钮就会像列表项一样堆叠起来，此时的显示效果就不那么好了（如右图所示）

### 3.1.2 设置 UIAlertView 的委托

开发者如何才能知道用户点了 OK 或 Cancel 按钮呢？通过 UIAlertView 的委托，我们可以在用户做出选择之后，于一个简单的回调方法中得知此信息。委托应该遵从 UIAlertViewDelegate 协议。一般情况下，我们会把主要的（也就是活动的）视图控制器对象设为警告视图的委托。

通过委托中的一些方法，我们可以响应用户的点击操作。正如前一节所示，假如只需显示一段带有 OK 按钮的信息，那么可以不设置委托。

用户操作完警告视图之后，委托就会收到名为 `alertView:clickedButtonAtIndex:` 的回调。系统传给这个方法的第二个参数就表示用户所点击的按钮。按钮编号从 0 开始，Cancel 按钮默认是 0 号按钮。在有的窗口中，Cancel 按钮位于左侧，而在有的窗口中，它则位于底部，虽说如此，但其编号却是一样的，除非开发者自己调整了 Cancel 按钮的索引（可以通过 `cancelButtonIndex` 属性来操作）。UIActionSheet 对象的情况与 UIAlertView 不同，本章稍后讨论这个问题。

下面这段范例代码会显示出一种带回调的警告视图，我们在回调方法中，把用户所点击的按钮编号打印到调试用的控制台里：

```
@interface TestBedViewController : UIViewController
<UIAlertViewDelegate>
@end

@implementation TestBedViewController
- (void)alertView:(UIAlertView *)alertView
  clickedButtonAtIndex:(int)index
{
    NSLog(@"User selected button %d\n", index);
}
- (void)showAlert
{
    UIAlertView *av = [[UIAlertView alloc]
        initWithTitle:@"Alert View Sample"
        message:@"Select a Button"
        delegate:self
```

```

        cancelButtonTitle:@"Cancel"
        otherButtonTitles:@"One", @"Two", @"Three", nil];

    // Tag your UIAlertView so it can be distinguished
    // from others in your delegate callbacks.
    av.tag = MAIN_ALERT;
    [av show];
}
@end

```

如果控制器要控制多个 UIAlertView, 那么在回调方法中, 我们可以通过标签来辨别不同的 UIAlertView。与控件所用的目标-动作机制不同, 所有的 UIAlertView 都会触发同一种回调方法。于是, 开发者可以根据标签, 在 switch 语句里面针对不同的 UIAlertView 分别进行响应。

### 3.1.3 显示 UIAlertView

show 这个实例方法可以把 UIAlertView 显示出来。而显示出来之后, 警告视图就会呈现模态 (modal)。也就是说, 它后面的屏幕内容会变暗, 同时, 用户无法操作程序里面除 UIAlertView 之外的部分。等用户通过点击某个按钮 (通常是 OK 或 Cancel) 对 UIAlertView 做出确认之后, 这个模态视图就消失了。此时, 系统会把控制权交给 UIAlertView 的委托, 使开发者可以在其中完成收尾工作, 并对用户所选的按钮做出响应。

UIAlertView 的各项属性在创建之后依然可以修改。你可通过修改 title 或 message 属性来定制它。message 是一段可选的文本, 它会出现出现在 UIAlertView 的 title 之下、按钮之上。经由 addButtonWithTitle: 方法, 还可以再添加一些按钮。

### 3.1.4 各种 UIAlertView

开发者可以通过 alertViewStyle 属性创建不同样式的 UIAlertView。按默认样式 (也就是 UIAlertViewStyleDefault) 创建出来的 UIAlertView 具备标题及信息文本, 另外还可以有一些按钮, 其效果如图 3-1 所示。这是 UIAlertView 系列中的基本款式, 开发者能够由此征询用户的意见, 而用户则可以通过点击 Yes/No、Cancel/OK 等按钮做出简单的选择。

iOS 还提供了三种样式, 它们专门针对需要输入文本的场合:

- UIAlertViewStylePlainTextInput——用户可以在这种样式的 UIAlertView 里输入文本。
- UIAlertViewStyleSecureTextInput——假如要考虑安全问题, 那么可以采用这种样式的 UIAlertView, 它会把用户键入的文本自动遮盖起来。文本将会以一系列大圆点的形式来显示, 而开发者则可以在委托的回调方法中编写代码, 读取用户输入的内容。
- UIAlertViewStyleLoginAndPasswordInput——该样式的 UIAlertView 提供了两个文本框供用户输入, 一个是 login (登录) 文本框, 用于输入明文的用户账号, 另一个是 password (密码) 文本框, 用于输入密码, 系统会把密码掩藏起来。

如果要给 UIAlertView 添加文本输入功能, 就把按钮设计得简单一些。最多用两个并

排摆放的按钮，一般是 OK 及 Cancel 按钮。要是再增加按钮的话，就不太美观了，会令文本框漂浮在 UIAlertView 上方，或出现在其两侧。

对于 UIAlertView 来说，开发者可以查看用户在每个文本框中输入的文本。textFieldAtIndex: 方法接受一个参数，此参数是个从 0 开始的索引，而返回值则是处在该索引位置上的文本框。在实际的开发中，除了 password 文本框的索引是 1 之外，其他文本框使用的索引都是 0。获取到文本框之后，就可以像下面这样通过 text 属性查询其内容了：

```
NSLog(@"%@", [myAlert textFieldAtIndex:0].text);
```

## 3.2 解决方案：构建支持块的警告视图

UIAlertView 的委托回调机制可能会导致开发者必须编写比较复杂的代码才行。所有的处理代码都遵循同一个套路，而我们在回调方法里必须通过标签来区分想要处理的各种 UIAlertView。此外，为了制作出图 3-2 这样的 UIAlertView，还必须将按钮及其对应功能小心地设置好。

有个办法要比使用委托回调简单得多，那就是在声明按钮的时候，直接把所要执行的实现代码传进去。这项任务正适合用块来完成。



图 3-2 这种警告视图没有使用 UIAlertView 类里传统的回调机制，而是通过开发者在创建按钮时所指定的块来响应用户的操作

### 3.2.1 块简介

块是对 C 语言的一种扩展，iOS 4 首次将其引入。它在概念上与方法或函数类似，而且可以保存到变量里面。C 语言中有一套类似的机制用来保存函数，那就是函数指针。然而块不仅仅是函数指针，它不但能保存其中的可执行代码，而且还能把外围作用域拷贝一份并保存起来。

定义块的时候，系统会创建一份局部栈的拷贝，并将其与块关联起来。等程序真正开始执行块时，它就能访问这份拷贝出来的栈了。这项功能非常强大，使得开发者可以把一段代码及其周边状态包裹到块里面，并传给某个方法。而那段代码则能在将来某个时间点或某种特定的环境下执行，比方说，可以在另外一个线程中执行，也可以按某种特定的次序来执行。

块的语法稍微有点复杂。它在某些方面与 C 语言的函数指针相似，所以，不熟悉函数指针的开发者也许会觉得块看上去有些奇怪。块的代码是以插入符 (^) 来表示的。一般来说，

块包含返回值类型、参数列表以及代码块本身：

```
^(return type)(argument list) { // code block }
```

返回值的类型若是 `void`，则可以省略。假如根本就没有参数，那么可以把整个参数列表都省略掉。我们可以像下面这样来定义最简单的块：

```
^{ // your code here }
```

如果想针对某个块编写 `typedef` 语句，那么代码的写法会与上面列出的块的通行写法稍有区别：

```
typedef (return type)(^typeName)(argument list);
```

定义好块之后，就可以像下面这样执行它了：

```
typedef void (^SomeBlock)(BOOL);
SomeBlock myBlock = ^(BOOL success)
{
    if (success)
        NSLog(@"Successful!");
    else
        NSLog(@"FAILED!");
};
myBlock(YES);
```

上面这段代码与普通的方法调用相比并没有什么特别之处。不过，当我们把块当成参数传给方法的时候，它的用法就变得丰富多了，比方说，在遍历数组时，可以针对其中每个元素来执行块。另外，块还有一项特性，就是能够访问外围作用域中的变量。下面这段范例代码可以访问捕获到的 `myBaseNumber`：

```
NSInteger myBaseNumber = 7;
NSArray *numbers = @[2, 3, 5, 8, 9, 11];

[numbers enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop)
{
    NSInteger current = [obj integerValue];
    NSLog(@"%d * %d = %d", myBaseNumber, current, myBaseNumber * current);
}];
```

即便离开了变量所处的作用域，块也依然能够访问捕获到的上下文<sup>①</sup>。

捕获外围作用域这样的特性有时可能会产生一些问题。由于块所存储的栈只是一份拷贝，所以，假如有人在定义了块之后，还未等它执行就先去修改捕获到的变量，那么到了真正运行块的时候，就无法看到修改后的值了。另外，开发者若是在块范围内对捕获到的变量进行修改，那么出了块范围之后，所做的变更就会丢失。

如果想在块范围内修改某个捕获到的变量，那么声明该变量时，必须加上 `__block` 这个存储类型修饰符。加上 `__block` 之后，变量就会放在一份共享的存储区里面，这使得原始代码的外围作用域以及块都可以去访问它。

① context 通译“上下文”，此处可以理解为“程序的执行环境中的内容”。——译者注

还有个问题也和块及捕获到的作用域相关，那就是保留循环（retain cycle）<sup>①</sup>。

### 3.2.2 使用块时避免保留循环

使用块时，要小心别出现了保留循环。开发者经常无意间在块里面对 `self` 进行了保留。这可能是直接由引用 `self` 造成的，也可能是由于使用 `ivar`<sup>②</sup>而导致的，因为 `ivar` 会间接地捕获 `self`。

为了避免保留循环，我们可以捕获弱引用版本的 `self`，并将这个弱引用赋给某个强引用，然后再于块里使用它。我们可以在块的开头声明这个强引用，这样的话，它就能够在整个块范围内保持有效了。别忘了检查这个指向 `self` 的强引用是不是 `nil`。下面这段代码既不会产生保留循环，又不会出现引用 `nil` 的问题：

```
_weak TestBedViewController *weakSelf = self;
[blockAlertView addButtonWithTitle:@"OK" actionBlock:^(
    TestBedViewController *strongSelf = weakSelf;
    if (strongSelf)
    {
        NSString *name = [strongSelf->blockAlertView
            textFieldAtIndex:0].text;
        NSLog(@"Tapped OK after entering: %@", name);
    }
}];
```

通过使用块，我们可以写出更为清晰的 API，而且，它还能极大地简化多线程环境下的编程。本节只打算稍微谈一谈块的功能及用法。苹果公司的开发文档里有非常详尽的信息，用来描述块及 Grand Central Dispatch（简称 GCD），那些内容值得一读。



在实现并行任务处理的时候，Grand Central Dispatch（GCD）是个非常强大的工具。iOS 从第 4 版开始引入了 GCD 这个工具，它提供了一套基于函数的 API，并利用块机制来构建并发代码（concurrent code），以发挥多核硬件的优势。

大部分苹果公司程序库都已经完全同块集成好了，不过还是有一些类没经过改装。比方说，`UIAlertView` 就亟需添加对块的支持。学会使用块之后，你会发现它们还有好多种用法。

解决方案 3-1 扩充了标准的 `UIAlertView` 类，令开发者可以在创建按钮时指定块，从而减少代码的复杂程度，并使得响应用户操作的代码能够离声明按钮的代码近一些。有了这个类之后，我们就不用实现委托或委托回调了。

虽说本条解决方案中并未出现 `UIAlertView` 类里原有的一些委托方法，但只需稍微编写

① 通译“循环引用”。它是由 `retain`（保留）操作而导致的引用循环，故而也可称为保留循环。为了保持循环引用这一中文术语同 `circular reference` 这一英文术语之间的对应关系，译文将 `retain cycle` 照原样写出。——译者注

② 是 `instance variable`（实例变量）的简称。——译者注

几行代码，就可以继续使用它们。setDelegate 方法会把委托保存到 externalDelegate 里面，以供开发者使用。我们可以给 BlockAlertView 添加代理 (proxy) 方法，把系统对 UIAlertView 的委托所做的调用转发给 externalDelegate:

```
- (void)didPresentAlertView:(UIAlertView *)alertView
{
    if ([externalDelegate
        respondsToSelector:@selector(didPresentAlertView:)])
    {
        [externalDelegate didPresentAlertView:alertView];
    }
}
```

当用户点击警告视图中的某个按钮时，BlockAlertView 会执行与该按钮相关的块。另外还可以再做一项改进，令开发者能够更精细地控制块的运行时机。这项改进留给读者作为练习。目前这个类会在 UIAlertView 中名为 alertView:clickedButtonAtIndex: 的委托回调方法里面执行块。这种做法也许没什么问题，不过有的开发者可能想等警告视图的消失动画彻底播放完之后再去执行块。你可以给 BlockAlertView 添加一个简单的布尔属性，令其根据该属性来决定应该在哪一个委托回调方法里面执行块。

### 解决方案 3-1 创建基于块的警告视图

```
@implementation BlockAlertView
{
    __weak id <UIAlertViewDelegate> externalDelegate;
    NSMutableDictionary *actionBlocks;
}

- (instancetype)init
{
    self = [super init];
    if (self)
    {
        self.delegate = self;
        actionBlocks = [[NSMutableDictionary alloc] init];
    }
    return self;
}

- (instancetype)initWithTitle:(NSString *)title
    message:(NSString *)message
{
    return [super initWithTitle:title
        message:message delegate:self cancelButtonTitle:nil
        otherButtonTitles:nil];
}

// Add cancel button to alert with title and block
```

```

- (NSInteger)setCancelButtonWithTitle:(NSString *)title
  actionBlock:(AlertViewBlock)block
{
    if (!title) return -1;
    NSInteger index = [self addButtonWithTitle:title
      actionBlock:block];
    self.cancelButtonIndex = index;
    return index;
}

// Add button to alert with title and block
- (NSInteger)addButtonWithTitle:(NSString *)title
  actionBlock:(AlertViewBlock)block
{
    if (!title) return -1;
    NSInteger index = [self addButtonWithTitle:title];
    if (block)
    {
        // Copy moves blocks from stack to heap
        actionBlocks[@(index)] = [block copy];
    }
    return index;
}

- (id<UIAlertViewDelegate>)delegate
{
    return externalDelegate;
}

// If the delegate is self, set on super, otherwise store
// for possible future use to proxy delegate methods.
- (void)setDelegate:(id)delegate
{
    if (delegate == nil)
    {
        [super setDelegate:nil];
        externalDelegate = nil;
    }
    else if (delegate == self)
    {
        [super setDelegate:self];
    }
    else
    {
        externalDelegate = delegate;
    }
}

#pragma mark - UIAlertViewDelegate

// Execute the appropriate actionBlock.

```

```
// View will be automatically dismissed after this call returns
- (void)alertView:(UIAlertView *)alertView
    clickedButtonAtIndex:(NSInteger)buttonIndex
{
    UIAlertViewBlock actionBlock = actionBlocks[@(buttonIndex)];
    if (actionBlock)
    {
        actionBlock();
    }
}
@end
```

---

### 获取解决方案代码

访问 <https://github.com/erica/iOS-7-Cookbook> 网页，并打开“C03 Alerts”文件夹，即可找到与本章中的解决方案相对应的完整范例项目。

---

## 3.3 解决方案：将变长参数列表与 UIAlertView 结合起来使用

有些方法的参数个数可以变化，这种方法叫作 variadic 方法。开发者可以在最后一个普通参数后面使用省略号 (...) 来声明变长参数列表 (Variadic Argument)。NSLog 与 printf 都是参数个数可变的方法。你可以向其传入格式化字符串，并同时传入任意数量的其他参数。

由于大部分警告视图都注重于显示文本，所以我们可以提供一些方法，使开发者能够用格式化字符串轻松地创建出 UIAlertView。解决方案 3-2 创建的 say: 方法可以把开发者传给它的参数收集起来，并据此构建字符串。然后，它会把字符串传给 UIAlertView，并将其显示出来。通过这个简单的方法，可以迅速把警告视图显示到屏幕上。

say: 方法既不解析变长参数列表，也不会去分析它们，而是直接把第一个参数当成格式化字符串，并与其他参数一起，传给 NSString 的 initWithFormat:arguments: 方法。这样就构建好了字符串，然后，say: 方法会把字符串作为标题 (title) 传给 initWithTitle 方法，以制作只含一个按钮的 UIAlertView。

本来我们要先用格式化字符串创建 NSString，然后再调用 initWithTitle 方法来创建 UIAlertView，但定义了参数个数可变的工具方法之后，就可以省略这些步骤了。现在只需调用 say: 方法，就可以完成上述步骤：

```
[NotificationAlert say:
    @"I am so happy to meet you, %@", yourName];
```

这条解决方案只用了一个非常简单的范例来演示变长参数列表的用法。实际上，除了简化字符串的初始化操作之外，它还有很多用途。



---

 解决方案 3-2 用参数个数可变的方法来简化 UIAlertView 的创建
 

---

```

+ (void)say:(id)formatstring,...
{
    if (!formatstring) return;

    va_list arglist;
    va_start(arglist, formatstring);
    id statement = [[NSString alloc]
        initWithFormat:formatstring arguments:arglist];
    va_end(arglist);

    UIAlertView *av = [[UIAlertView alloc]
        initWithTitle:statement message:nil
        delegate:nil cancelButtonTitle:@"Okay"
        otherButtonTitles:nil];
    [av show];
}

```

---

### 3.4 展示选项列表

UIActionSheet 实例可以创建简单的 iOS 菜单。在 iPhone 及 iPod touch 上面, 这种菜单会把各个选项展示到屏幕上, 并等待用户做出选择, 而那些选项基本上就是一系列表示相关操作的按钮。在 iPad 上面, 它们是以弹出框 (popover) 形式呈现的, 并且其中没有 Cancel 按钮。用户在 popover 范围外点击, 即可取消选择。

动作表 (Action Sheet) 与 UIAlertView 都是从同一个源头类继承下来的, 然而两者之间有所区别。从 iPhone 早期开始, 它们就成了各自不同的类。由于 UIAlertView 会从原有的界面中跳出来, 所以很适合用来吸引用户的注意力。而动作表菜单则会滑入视图之中, 所以更适合与程序里正在执行的任务相集成。Cocoa Touch 提供了五种表示菜单的方式:

- **showInView:**——在 iPhone 与 iPod touch 中, 该方法会将菜单从视图的底部向上滑入屏幕。在 iPad 里, 动作表会出现在屏幕正中。
- **showFromToolBar:** 及 **showFromTabBar:**——在 iPhone 与 iPod touch 中, 如果程序里使用了 toolbar (工具栏)、tab bar (标签栏) 或是其他种类的 bar, 那么这两个方法会先将菜单的顶部与 bar 的顶部对齐, 然后逐渐将其滑入屏幕, 并准确地滑动到合适的位置上。所谓 bar, 就是很多程序底部出现的那种条状物, 其中有许多组水平排列的按钮。在 iPad 上面, 动作表出现在屏幕正中。
- **showFromBarButtonItem:animated:**——在 iPad 中, 该方法会在特定的 UIBar-ButtonItem 上面, 将动作表以 popover 的形式展示出来。
- **showFromRect:inView:animated:**——开发者可以把视图里某块矩形区域的坐标指定给该方法, 而该方法则会将动作表从这块矩形区域中滑入屏幕。

**提示** 不要在使用选项卡的子视图控制器里使用 `showInView`。动作表虽然能够正常显示出来，但是 **Cancel** 按钮的底部却无法响应用户操作。

下面这段代码演示了如何初始化并展示简单的 `UIActionSheet` 实例。初始化方法中有一个概念是 `UIAlertView` 所没有的，那就是 **Destructive** 按钮。这种按钮是红色的，用以表示破坏性的操作，例如永久删除某个文件（如图 3-3 所示）。鲜红色可以警示用户该选项有风险。开发者应该谨慎使用这种选项。

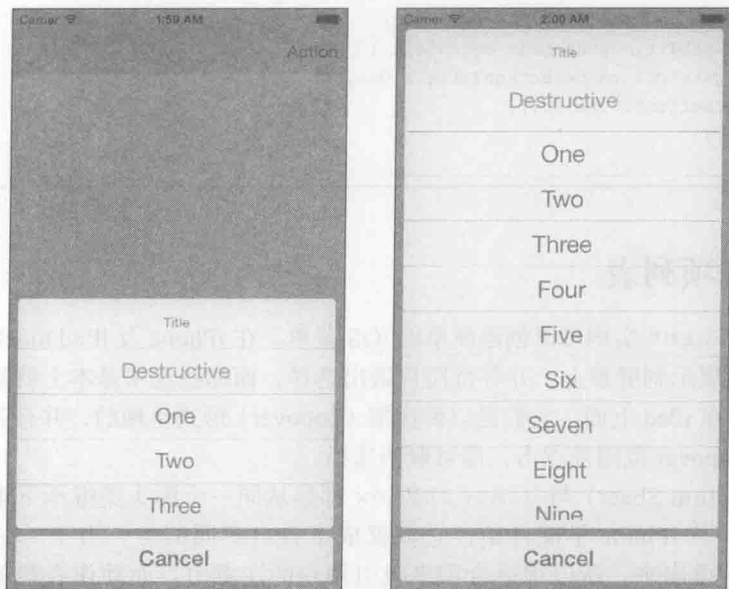


图 3-3 在 iPhone 和 iPod touch 中，动作表会从视图底部滑入屏幕中。菜单上的 **Destructive** 按钮呈现红色，以便向用户表示该操作可能产生持久的负面效果。如果菜单项比较多，那么就会显示成右侧那样的滚动列表

在动作表中，各按钮的索引值与该按钮的顺序有关。以图 3-3 左侧为例，**Destructive** 按钮是 0 号按钮，而 **Cancel** 按钮则是 4 号按钮。这和 `UIAlertView` 的默认编号方式不同，`UIAlertView` 会把 **Cancel** 当成 0 号按钮。在动作表里，**Cancel** 按钮的序号由其位置来决定，而它的位置又取决于开发者添加按钮的方式。对于某些没有 **Destructive** 按钮的动作表来说，**Cancel** 按钮可能会默认成为 0 号按钮，并成为菜单中的首个菜单项<sup>①</sup>。开发者可以经由动作表的 `cancelButtonIndex` 属性来查看 **Cancel** 按钮的索引值。下面这段代码会把用户所选按钮的索引值打印出来：

① 此问题的详情可参考：<https://stackoverflow.com/questions/5262428/uiactionsheet-buttonindex-values-faulty-when-using-more-than-6-custom-buttons>。——译者注

```

- (void)actionSheet:(UIActionSheet *)actionSheet
  didDismissWithButtonIndex:(NSInteger)buttonIndex
{
    self.title = [NSString stringWithFormat:@"Button %d", buttonIndex];
}

- (void)action:(UIBarButtonItem *)sender
{
    // Destructive = 0, One = 1, Two = 2, Three = 3, Cancel = 4
    UIActionSheet *actionSheet = [[UIActionSheet alloc]
        initWithTitle:@"Title"
        delegate:self
        cancelButtonTitle:@"Cancel"
        destructiveButtonTitle:@"Destructive"
        otherButtonTitles:@"One", @"Two", @"Three", nil];
    [actionSheet showFromBarButtonItem:sender animated:YES];
}

```

不要在 iPad 上面使用 Cancel 按钮。把动作表展示到 iPad 屏幕之后，用户可以在动作表的范围之外点击，以取消这次操作：

```

UIActionSheet *actionSheet = [[UIActionSheet alloc]
    initWithTitle:theTitle delegate:nil
    cancelButtonTitle:IS_IPAD ? nil : @"Cancel"
    destructiveButtonTitle:nil otherButtonTitles:nil];

```

如果在 iPad 上面取消了某个动作表，那么该表（Sheet）默认会返回 -1。开发者可以覆写这个值，但笔者不推荐这么做。



**提示** 你也可以参照解决方案 3-1 中的 BlockAlertView 类，采用基于块的方式来方便地创建动作表。

### 3.4.1 滚动菜单

有一条大致的规律：iPhone 及 iPod touch 在纵屏模式下最多可以显示大约 10 个按钮（包括 Cancel 在内），而在横屏模式下最多能显示大概 5 个按钮。（iPad 的显示空间会比它们大很多。）如果超过了这个数量，那么系统就会像图 3-3 右侧那样，以滚动列表的方式呈现菜单。请注意：即便在这种情况下，Cancel 按钮也依然出现在整份列表的下方，而不会出现在列表里。此时，Cancel 按钮的编号总是排在它前面的那些按钮之后。从图 3-3 中可以看出，这种滚动式的列表菜单并不美观，所以应该尽量避免使用。

### 3.4.2 在动作表中显示文本

动作表提供了与 UIAlertView 相同的文本展示功能，并且具备更大的绘制区域。下面这段代码演示了如何用 UIActionSheet 对象来显示消息。通过这种方式，我们可以方便地将多条文本同时展示出来：

```

- (void)show:(id)formatstring,...
{
    if (!formatstring) return;
    va_list arglist;
    va_start(arglist, formatstring);
    id statement = [[NSString alloc]
        initWithFormat:formatstring arguments:arglist];
    va_end(arglist);

    UIAlertController *actionSheet = [[UIAlertSheet alloc]
        initWithTitle:statement
        delegate:nil cancelButtonTitle:nil
        destructiveButtonTitle:nil
        otherButtonTitles:@"OK", nil];

    [actionSheet showInView:self.view];
}

```

### 3.5 将操作进度告知用户并提示其稍等片刻

操作计算机程序时，免不了要等待，在可以预见的将来，仍是如此。开发者的职责之一，就是把这种情况告诉用户。我们可以用 Cocoa Touch 所提供的一些类来提示用户，请其等待某个操作执行完毕。这些进度指示器有两种形式，一种是在执行任务期间持续显示的转轮，另一种是随着任务执行进度由左至右增长的进度条。提供这两种指示器的类分别是：

- **UIActivityIndicatorView**——这种进度指示器就是个旋转的圆形，用以提示用户需要等待某个操作执行完毕，但是，它并不能提供与任务完成进度有关的详细信息。iOS 的 `UIActivityIndicatorView` 很小，不过其动画效果可以吸引用户的注意力。如果应用程序里突然出现需要打断用户操作的任务，那么非常适合以这种方式来提示。
- **UIProgressView**——这种视图用来表示进度条。进度条是一种直观的反馈方式，它占用的空间相对较小，可以告诉用户已经完成和尚待完成的任务量。它是个沿水平方向延伸的扁平矩形，并会随着进度从左至右增长。这种经典的用户界面元件（user interface element，简称 UI 元件）适合表示那种延迟较长而用户又需要知道处理进度的任务。

请注意，不要阻塞主线程。与使用其他 GUI 对象时所应遵循的规则相同，这两个类也必须在主线程上使用。计算量比较大的代码会阻塞主线程，使得各视图无法实时更新其内容。假如你编写的代码阻塞了主线程，那么 `UIProgressView` 就不会随着进度实时更新了，而是会卡在初始值那里。

若想显示异步的反馈效果，请使用多线程。例如，可以把 `UIActivityIndicatorView` 放在主线程上，同时在另一个线程里执行计算。然后，我们可以把另一个线程里的计算情况更新到主线程的 `UIActivityIndicatorView` 上面，这样就能平稳地修改进度条的进度，使用户得知当前任务的处理情况了。

### 3.5.1 使用 UIActivityIndicatorView

UIActivityIndicatorView (活动指示器视图) 实例提供了一种轻量级视图, 以显示标准的转轮图案。使用这种视图的时候, 始终要注意, 它们是很小的。所有的 UIActivityIndicatorView 都非常袖珍, 如果将其放得太大, 那么看上去效果就不好了。

iOS 提供了几种不同风格的 UIActivityIndicatorView 类。UIActivityIndicatorViewStyleWhite 及 UIActivityIndicatorViewStyleGray 风格的长度和宽度均为 20 点。白色样式的指示器最好放在黑色背景上面, 而灰色样式的指示器则最好放在白色背景上面。这两种都算是比较小巧的风格。UIActivityIndicatorViewStyleWhiteLarge 风格用于暗色背景, 它是最大、最清晰的一种风格, 其长度和宽度都是 37 点:

```
UIActivityIndicatorView *aiv = [[UIActivityIndicatorView alloc]
    initWithActivityIndicatorStyle:
        UIActivityIndicatorViewStyleWhiteLarge];
```

通过 color 属性, 开发者可以指定 UIActivityIndicatorView 的 tint color。设置好的 color 属性将会覆盖风格中的相应颜色, 但是不会改变该风格所对应的尺寸 (也就是说, 普通尺寸的 UIActivityIndicatorView 依然是普通尺寸, 而大尺寸的 UIActivityIndicatorView 则依然是大尺寸):

```
aiv.color = [UIColor blueColor];
```

开发者并不一定要把 UIActivityIndicatorView 放在屏幕正中, 只要将其放在合适的位置上即可。由于它是个背景色透明的图案, 所以无论背后是什么视图, 它都会渲染在那个视图上面。我们应该根据背后那个视图的主要颜色来指定 UIActivityIndicatorView 的颜色。

一般情况下, 只需把 UIActivityIndicatorView 作为子视图添加到 Window (视窗)、View (视图)、Toolbar (工具栏) 或 Navigation Bar (导航栏) 之中, 即可令其出现在它们前方了。分配好指示器所用的内存之后, 可以用框架或 Auto Layout 约束来初始化它的尺寸, 然后还可以令其在上级视图里居中。向对象发送 startAnimating 消息, 即可启动指示器的动画效果。调用 stopAnimating, 则可令其停止, 系统会把停止后的 UIActivityIndicatorView 隐藏起来, 其余的事情则由 Cocoa Touch 来处理。

### 3.5.2 使用 UIProgressView

有了 UIProgressView 之后, 我们就可以把任务的处理进度告诉用户了, 而不是仅仅说句“请稍等”。进度条会随着时间的推移而填充自身内容, 它用填充程度来表示任务的完成进度。进度条很适合用在那种等待时间比较长的任务上面, 用户可以通过它所提供的状态反馈, 得知程序依然在正常运作。

要创建进度条, 首先得分配 UIProgressView 实例并设定其 frame。开始使用进度条时, 需要调用 setProgress: 方法。该方法接受一个浮点型的参数, 其值位于 0.0 至 1.0 之间。0.0 所对应的进度是 0%, 表示“没有任何进度”, 而 1.0 所对应的进度则是

100%，表示“任务已完成”。进度条有两种风格：一种偏白，另一种是浅灰色<sup>①</sup>。开发者使用 `setStyle:` 方法来指定风格，该方法的参数值可以取 `UIProgressViewStyleDefault` 或 `UIProgressViewStyleBar`。后者适用于工具栏中的进度条。

### 3.6 解决方案：在屏幕上绘制模态的进度指示器

虽说 `UIAlertView` 及 `UIActionSheet` 向用户提供了直观的通讯及互动方式，但是开发者却不能在其中添加自己的子视图。所以，为了实现模态的进度指示器，我们必须从头开始来实现自己的 `UIView`。解决方案 3-3 在设备屏幕上绘制了一个颜色较淡的 `UIView` 对象，并在其中添加了 `UIActivityIndicatorView`。

如图 3-4 所示，这个 `UIView` 覆盖在整个屏幕之上。占满全屏幕是为了能把导航栏也覆盖进去。这种 `UIView` 必须添加到应用程序的视图（window）中，而不是像有些读者想的那样，添加到主 `UIViewController` 的视图中。那个视图只能涵盖导航栏以下的那些空间（用 `UIScreen` 的术语来说，就是只能涵盖应用程序窗口的外框），所以没有办法阻止用户继续操作导航栏上的按钮及其他物件。填满整个视图（window）可以阻止用户去操作那些东西。



图 3-4 装配了 `UIActivityIndicatorView` 的模态视图，可以在程序执行同步操作（或称阻塞式操作）时向用户展示反馈效果。开发实际的应用程序时，要记得给用户提供一种操作方式，令其在无须强行退出程序的前提下，能够把比较耗时的任务取消掉

解决方案 3-3 展示并隐藏定制的模式 `UIView`

```
- (void)removeOverlay:(UIView *)overlayView
{
    [overlayView removeFromSuperview];
}

- (void)action
{
    UIWindow *window = self.view.window;

    // Create a tinted overlay, sized to the window
    UIView *overlayView =
        [[UIView alloc] initWithFrame:window.bounds];
    overlayView.backgroundColor =
```

① 实际效果请参见：<https://developer.apple.com/library/ios/documentation/userexperience/conceptual/mobilehig/Controls.html>——译者注

```
[[UIColor blackColor] colorWithAlphaComponent:0.5f];
overlayView.userInteractionEnabled = YES;
```

```
// Add an activity indicator
UIActivityIndicatorView *aiv =
    [[UIActivityIndicatorView alloc]
     initWithActivityIndicatorStyle:
         UIActivityIndicatorViewStyleWhiteLarge];
[aiv startAnimating];
[overlayView addSubview:aiv];
PREPCONSTRAINTS(aiv);
CENTER_VIEW.overlayView, aiv);
```

```
UILabel *label = [[UILabel alloc] init];
label.textColor = [UIColor whiteColor];
label.text = @"Please wait...";
[overlayView addSubview:label];
PREPCONSTRAINTS(label);
CENTER_VIEW_H.overlayView, label);
CENTER_VIEW_V_CONSTANT.overlayView, label, -44);
```

```
[window addSubview:overlayView];
// Use a time delay to simulate a task finishing
[self performSelector:@selector(removeOverlay:)
    withObject:overlayView afterDelay:5.0f];
```

```
}
```

为了阻止用户触摸其他物件，我们把这个 `UIView` 对象的 `userInteractionEnabled` 属性设为 `YES`。这样就可以令它把全部触摸事件都捕获起来，从而阻止这些事件到达下方的其他 GUI。由于用户在这个 `UIView` 消失之前无法操作其他控件，所以这种做法也就等于创建出了模态的显示效果。只需对本例稍加改编，即可实现出一种触摸之后就会消失的 `UIView`。但要注意：由于这种 `UIView` 并不属于视图控制器，所以在设备屏幕方向发生改变时，它并不会自我更新。假如想令程序自动适应设备的横屏 / 纵屏状态，那么可以先查询当前的屏幕方向，然后再根据方向来显示这个 `UIView`，同时，还应该订阅与屏幕方向变更有关的通知。

## 点击后即可消失的模态 `UIView`

通过自定义的 `UIView`，我们既可以展示消息，又能够限制用户的操作。只需对解决方案 3-3 中的 `UIView` 稍加扩充，就能实现出一种点击后即可消失的视图。用户点击这种视图之后，它会把自己从屏幕中移除。平常适合以 `UIAlertView` 来展示的消息现在也很适合改用这种视图来显示：

```
@interface TappableOverlay : UIView
@end

@implementation TappableOverlay
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Remove this view when it is touched
}
```

```

        [self removeFromSuperview];
    }
@end

```

### 3.7 解决方案：自制的模态警告视图

UIAlertView 的功能太少，而且不够灵活，所以其用途比较有限。解决方案 3-3 实现了一个简单的模态窗口，非常适合在程序执行比较耗时的任务时向用户显示消息。有时候，我们就是需要这样一种功能完备并且可以自行配置的警告视图，而不想使用由苹果公司施加了人为限制的 UIAlertView。

解决方案 3-4 提供的这种警告视图的各方面几乎都可以由开发者来定制。你可以根据需求向其中添加子视图或配置 UI 元件，可供配置的内容包括边框、背景以及子视图的位置等。使用我们自编的警告视图来取代系统内置的 UIAlertView 会有个好处，就是可以自己指定切换时的动画效果。解决方案 3-4 在展示和隐藏警告视图的时候，都使用了弹出效果。弹出效果是通过仿射缩放变换来实现的，第 5 章将会详述此问题。



**提示** 苹果公司在 iOS 7 里引入了一套基于物理的动画系统，叫作 Dynamics。为了在显示和隐藏警告视图的时候模拟弹出效果，本范例所采用的做法是对视图进行变换，但假如改用 Dynamics 来做的话，那么还能以声明式的方法添加更为复杂的视觉交互。

由图 3-5 可以看出，这种警告视图不会将开发者局限在一套特定的界面之中，而是提供了一张可以随时定制的白板（blank slate）。笔者在其中添加了标签和按钮，以演示它的用法。

#### 磨砂玻璃效果

在 iOS 7，尤其是 Control Center（控制中心）中，我们可以看到一种新的视觉效果，它叫作磨砂玻璃效果，笔者把这种效果运用到了自定义的警告视图中。苹果公司令许多 UIKit 元件都默认支持该效果，其中包括 UITabBar、UINavigationController 及 UIToolbar。

这些 Bar 类本身就实现了该特效，不过除此之外，我们没有其他办法能将其添加到自己的视图中。苹果公司提供了一段范例代码，并为 UIImage 类编写了 category，用以模拟此特效，但其效果却不如系统内置的实现效果好。UIImage 类的 category 里面所实现的特效与系

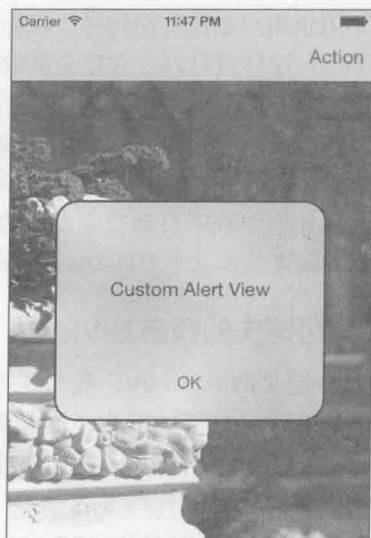


图 3-5 iOS 系统所提供的 UIAlertView 的功能非常有限，而笔者自己编写的这种警告视图则能使开发者对用户界面及交互方式做出全方位定制



统内置的特效还有一些差距，而且渲染速度也慢得多，所以，不适合当作实时特效来用。

为了解决这个问题，解决方案 3-4 从 UINavigationController 中继承了子类，以便能够使用基类所提供的特效。但愿苹果公司以后会把这项特性直接提供给开发者，而不是像现在这样，必须编写略显隐晦的代码才能使用它。

#### 解决方案 3-4 自制的模态警告视图

```
@implementation CustomAlert
{
    UIView *contentView;
}

#pragma mark - Utility
- (void)observeValueForKeyPath:(NSString *)keyPath
    ofObject:(id)object change:(NSDictionary *)change
    context:(void *)context
{
    if ([keyPath isEqualToString:@"bounds"])
        contentView.frame = self.bounds;
}

#pragma mark - Instance Creation and Initialization
- (void)internalCustomAlertInitializer
{
    // Add size observer
    [self addObserver:self forKeyPath:@"bounds"
        options:NSKeyValueObservingOptionNew context:NULL];

    // Constrain the size and width based on the initial frame
    self.translatesAutoresizingMaskIntoConstraints = NO;
    CGFloat width = self.bounds.size.width;
    CGFloat height = self.bounds.size.height;
    for (NSString *constraintString in
        @[@"V:[self(==height)]", @"H:[self(==width)]"])
    {
        NSArray *constraints = [NSLayoutConstraint
            constraintsWithVisualFormat:constraintString options:0
            metrics:@{@"width":@(width), @"height":@(height)}
            views:NSDictionaryOfVariableBindings(self)];
        [self addConstraints:constraints];
    }
    [self layoutIfNeeded];
    // Add a content view for auto layout
    contentView = [[UIView alloc] initWithFrame:self.bounds];
    [self addSubview:contentView];
    contentView.autoresizingMask =
        UIViewAutoresizingFlexibleHeight |
        UIViewAutoresizingFlexibleWidth;

    // Add layer styling
```

```

self.layer.borderColor = [UIColor blackColor].CGColor;
self.layer.borderWidth = 2;
self.layer.cornerRadius = 20;
self.clipsToBounds = YES;

// Create label
_label = [[UILabel alloc] init];
[contentView addSubview:_label];
_label.translatesAutoresizingMaskIntoConstraints = NO;
_label.numberOfLines = 0;
_label.textAlignment = NSTextAlignmentCenter;

// Create button
_button = [UIButton buttonWithType:UIButtonTypeSystem];
[contentView addSubview:_button];
_button.translatesAutoresizingMaskIntoConstraints = NO;

// Layout subviews on content view
for (NSString *constraintString in
    @[@"V:|-[ _label ]-[ _button ]-|",
      @"H:|-[ _label ]-|", @"H:|-[ _button ]-|"])
{
    NSArray *constraints = [NSLayoutConstraint
        constraintsWithVisualFormat:constraintString
        options:0 metrics:nil
        views:NSDictionaryOfVariableBindings(_button, _label)];
    [contentView addConstraints:constraints];
}
}

- (instancetype)initWithFrame:(CGRect)frame
{
    if (!(self = [super initWithFrame:frame])) return self;
    [self internalCustomAlertInitializer];
    return self;
}

- (instancetype)initWithCoder:(NSCoder *)aDecoder
{
    if (!(self = [super initWithCoder:aDecoder])) return self;
    [self internalCustomAlertInitializer];
    return self;
}

- (void)dealloc
{
    [self removeObserver:self forKeyPath:@"bounds"];
}

#pragma mark - Presentation and Dismissal
- (void)centerInSuperview

```

```

{
    if (!self.superview)
    {
        NSLog(@"Error: Attempting to present without superview");
        return;
    }

    NSArray *constraintArray =
        [self.superview.constraints copy];
    for (NSLayoutConstraint *constraint in constraintArray)
    {
        if ((constraint.firstItem == self) ||
            (constraint.secondItem == self))
            [self.superview removeConstraint:constraint];
    }
    [self.superview addConstraints:CONSTRAINTS_CENTERING(self)];
}

- (void)show
{
    self.transform =
        CGAffineTransformMakeScale(FLT_EPSILON, FLT_EPSILON);
    [self centerInSuperview];

    CustomAnimationBlock expandBlock = ^{self.transform =
        CGAffineTransformMakeScale(1.1f, 1.1f);};
    CustomAnimationBlock identityBlock = ^{self.transform =
        CGAffineTransformIdentity;};
    CustomCompletionAnimationBlock completionBlock =
        ^(BOOL done){[UIView animateWithDuration:0.3f
            animations:identityBlock];};

    [UIView animateWithDuration:0.5f animations:expandBlock
        completion:completionBlock];
}

- (void)dismiss
{
    CustomAnimationBlock expandBlock = ^{self.transform =
        CGAffineTransformMakeScale(1.1f, 1.1f);};
    CustomAnimationBlock shrinkBlock = ^{self.transform =
        CGAffineTransformMakeScale(FLT_EPSILON, FLT_EPSILON);};
    CustomCompletionAnimationBlock completionBlock =
        ^(BOOL done){[UIView animateWithDuration:0.3f
            animations:shrinkBlock];};

    [UIView animateWithDuration:0.5f animations:expandBlock
        completion:completionBlock];
}
@end

```

## 3.8 解决方案：基本的 popover

笔者编写本书时，只有 iPad 支持 popover 这项特性。不过将来苹果公司也可能会推出其他支持 popover 的新 iOS 设备，或是把这项特性移植到 iPhone 系列的手机上。有时我们想用模态的视图去展示信息，而有的时候则想改用 popover 来做。在日常开发中使用 popover 时，请参考下面几条经验法则：

- 应该把 popover 对象保留住。应该创建强指针形式的局部变量，使得系统能够保留住 popover，直到不再使用为止。在解决方案 3-5 中，程序会在 popover 消失时重置该变量。

解决方案 3-5 基本的 popover

---

```

- (void)popoverControllerDidDismissPopover:
    (UIPopoverController *)popoverController
{
    // Stop holding onto the popover
    popover = nil;
}

- (void)action:(id)sender
{
    // Always check for existing popover
    if (popover)
        [popover dismissPopoverAnimated:YES];

    // Retrieve the nav controller from the storyboard
    UIStoryboard *storyboard =
        [UIStoryboard storyboardWithName:@"Storyboard"
         bundle:[NSBundle mainBundle]];
    UINavigationController *controller =
        [storyboard instantiateInitialViewController];

    // Present either modally or as a popover
    if (IS_IPHONE)
    {
        [self.navigationController
         presentViewController:controller
         animated:YES completion:nil];
    }
    else
    {
        // No Done button on iPads
        UIViewController *vc = controller.topViewController;
        vc.navigationItem.rightBarButtonItem = nil;

        // Set the preferred content size to iPhone-sized
        vc.preferredContentSize =
            CGSizeMake(320.0f, 480.0f - 44.0f);

        // Create and deploy the popover
    }
}

```

---

```

popover = [[UIPopoverController alloc]
initWithContentViewController:controller];
[popover presentPopoverFromBarButtonItem:sender
permittedArrowDirections:UIPopoverArrowDirectionAny
animated:YES];
}
}

```

- 弹出新的 popover 之前，应该检查有没有仍在显示中的 popover，若有，则将其隐藏。如果要在程序里创建不同用途的多个 popover，那么尤其要注意这个问题。比方说，如果有好几个 UIBarButtonItem 都会弹出 popover，那么，在显示新的 popover 之前，应该先把现有的隐藏起来。
- 根据显示内容设置 popover 的尺寸。iPad 默认提供的 popover 是长而扁的，其风格可能与你自己的应用程序不符。设置视图控制器的 preferredContentSize 属性，即可指定 popover 所应具备的尺寸。
- 要考虑到 iPhone 上面的显示效果。程序运行在另一种设备上面时，其功能不应该缩减。对于 iPhone 系列的手机来说，我们应该通过控制器展示一种与 popover 类似的模态视图。
- 不要在 popover 上面添加 Done 按钮。一般的模态视图通常都会有个 Done 按钮，但是不要在 popover 上面放置这种按钮。用户只需在 popover 范围之外点击，即可令其消失，所以说，Done 按钮是多余的。



**提示** iOS 7 刚发行时，苹果公司在开发文档里宣称，不会再用箭头图案来表示 popover 是从什么地方弹出的了。开发文档把 UIPopoverController 的 popoverArrowDirection 属性标注成了 deprecated（弃用）。不过，现在这个箭头还在，而且 SDK 头文件里的相关代码也还没有变成 deprecated 状态。苹果公司可能曾对 popover 的设计做出了修改，但其后又决定放弃修改，然而却没有立即更新开发文档，从而导致了这种不一致的现象，后续版本的文档也许会修复这一问题<sup>①</sup>。

### 3.9 解决方案：本机通知

本机通知（local notification）是一种在程序未运行的时候通知用户的手段。用这种简单的方式，我们可以在特定的日期和时间点显示一条提醒信息。与推送通知不同，本机通知无须联网，也不用和远程服务器相通信。正如其名称所示，它是一种完全可以在设备本地处理的通知形式。

① 目前的开发文档已经不再将 popoverArrowDirection 标注为 deprecated 了。——译者注

本机通知是与日历及待办事项清单等日程安排工具结合起来使用的。某些多任务应用程序在没有运行于前台时，也可以经由本机通知来向用户提供更新。比方说，用户距离当地某家图书馆很近的时候，基于位置的应用程序也许就会弹出一条通知，告诉用户现在可以去查阅书籍了。

应用程序处在活动状态时，系统不会展示本机通知，只有当其进入暂停状态或切换到后台运行时，才会显示这种通知。解决方案 3-6 安排了一条通知，定于 5 秒钟之后弹出，为了到时候能把通知显示出来，需要强迫应用程序退出。假如你开发的程序想在 App Store 上架，那可别这么做；笔者此处只是为了演示。如果我们不把这个范例程序强行关闭的话，到时候可能会错过通知。

### 解决方案 3-6 安排一条本机通知

```
- (void)action:(id)sender
{
    UIApplication *app = [UIApplication sharedApplication];

    // Remove all prior notifications
    NSArray *scheduled = [app scheduledLocalNotifications];
    if (scheduled.count)
        [app cancelAllLocalNotifications];

    // Create a new notification
    UILocalNotification* alarm =
        [[UILocalNotification alloc] init];
    if (alarm)
    {
        alarm.fireDate =
            [NSDate dateWithTimeIntervalSinceNow:5.0f];
        alarm.timeZone = [NSTimeZone defaultTimeZone];
        alarm.repeatInterval = 0;
        alarm.alertBody = @"Five Seconds Have Passed";
        [app scheduleLocalNotification:alarm];
        // Force quit. Never do this in App Store code.
        exit(0);
    }
}
```

与推送通知一样，如果用户点击了显示本机通知的按钮，那么系统就会重新运行应用程序，并把控制权交给 `application:didFinishLaunchingWithOptions:` 方法。以 `UIApplicationLaunchOptionsLocalNotificationKey` 为键，在 `launchOptions` 参数所表示的字典中查询，即可找到与本机通知有关的那个对象。

某些开发者想利用这种重启应用程序的效果，经由通知中心来添加一些特色功能，这么做有时可行，有时不可行。该做法背后的思路是：添加本机通知之后，如果用户点击了这条通知，那么系统就会启动我们的应用程序，而此时可以执行一些任务。这样做实际上就等于

通过通知中心向用户提供了一些功能。对通知中心的非常规用法并未总是得到苹果公司的首肯。这种巧妙但是未受认可的使用方式究竟能不能成功，很大程度上要看苹果公司是否会调整其策略。

### 本机通知的使用原则

不要向用户滥发通知。本机通知确实是一种无须用户确认即可使用的通知手段，但不能因为如此就滥用它来做营销。总的原则为：如果某条信息不是用户专门指明要投递的，那就不要发送这种通知。（此原则也适用于推送通知。即使用户同意接收推送通知，我们也不应该滥发消息。）

不请自来的通知对程序的用户体验没什么好的效果。假如你在用户正吃饭的时候发送通知，或是在凌晨三点发送通知，那么你所制作的应用程序就不会讨人喜欢，于是就得不到好评，也吸引不到更多的用户。

无论开发者有没有给用户提供“请勿打扰”选项，只要滥用通知，就是种错误的做法。若是经由推送通知来发送广告，那么苹果公司通常会拒绝这种程序上架，使用本机通知的时候也同样要注意这个问题。最容易使应用程序评分和开发者声望受损的做法就是胡乱发送通知信息。

最后还要注意，应该仔细检查通知内容有没有拼写错误。

### 3.10 用网络活动指示器提醒用户

程序在后台联网时，开发者应将此情况告诉用户，这样显得礼貌一些。我们并不需要创建全屏的警告视图，因为 Cocoa Touch 已经提供了一个简单的 `UIApplication` 属性，该属性能够控制状态栏中旋转的网络活动指示器。图 3-6 演示了这种指示器，它位于 Wi-Fi 信号指示器的右侧，当前时间的左方。

下面这段代码演示了该属性的用法，此处我们只不过简单地切换了指示器的开/关状态：

```
-(void)action:(id)sender
{
    // Toggle the network activity indicator
    UIApplication *app = [UIApplication sharedApplication];
    app.networkActivityIndicatorVisible =
        !app.networkActivityIndicatorVisible;
}
```

开发实际应用程序时，通常需要在另一个线程里执行与网络有关的任务。与此同时，开发者必须在主线程上面执行与 UI 相关的更新操作。于是，我们可以像下面这段代码一样，在其他线程里通过 GCD 机制来请求系统在主线程上更新 GUI：

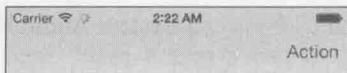


图 3-6 网络活动指示器是由 `UIApplication` 中的相关属性来控制的

```
dispatch_async(dispatch_get_main_queue(), ^{
    // set activity indicator here
});
```

你可以记录应用程序里正在执行的网络操作数目，当至少有一个网络操作处于活动状态时，才去启用指示器。

## 给应用程序加上徽章形提示标志

使用过 iOS 系统的读者会发现，主屏幕的应用程序右上方可能会出现红色徽章图样的小标志。这些标志里面的数字也许表示用户上次打开 Phone(电话) 程序之后未接听的来电数目，或是上次打开 Mail(邮件) 程序之后未读取的邮件数目。

用代码把 `applicationIconBadgeNumber` 属性设为正整数，即可给应用程序添加徽章图样的提示标志。将 `applicationIconBadgeNumber` 设为 0，就能隐藏徽章图样。

如果用户打开了应用程序，那么应该把这个徽章形的提示标志移除。因为用户一般都认为，只要启动了某个程序，就可以清除主画面程序图标右上角的提示了。

## 3.11 解决方案：播放简单的提示音

应用程序可以用提示音直接对用户“说话”。对于听觉没有障碍的用户来说，这是一种即时的反馈手段。所幸苹果公司在 Cocoa Touch SDK 里面通过 System Audio 服务提供了基本的声音播放功能。

除此之外，还可以通过 Audio Queue 来调用 `AVAudioPlayer`。在程序中使用 Audio Queue 来播放声音是相当耗时的，它比播放简单的提示音要复杂得多。反之，我们只需几行代码，就能加载并播放 System Audio 了。另外，`AVAudioPlayer` 也有其缺点，它会干扰 iPod 的声音，而 System Audio 所播放的声音则不会打断设备正在播放的音乐，只不过提示音的播放效果可能不太理想，有时会埋在音乐声中。

提示音越短越好，根据苹果公司的建议，应该保持在 30 秒以内。System Audio 只能播放 PCM 音频和 IMA 音频，也就是说，只支持 AIFF、WAV 及 CAF 声音格式。

### 3.11.1 System Sound

以指向声音文件的 URL 为参数来调用 `AudioServicesCreateSystemSoundID`，即可构建 System Sound。调用完成之后，开发者将获得一个初始化过的 System Sound 对象，我们可在这个对象上面播放声音。以该对象为参数，调用 `AudioServicesPlaySystemSound` 函数，即可播放声音。下面这行语句就是用来播放声音的：

```
AudioServicesPlaySystemSound(mySound);
```

如果 iPod 正在播放其他音乐，那么它一般会以相同的音量来播放 System Sound，而不会先把音乐声音调小。这样的话，用户有可能听不到提示音。下面这行代码用于判断设备当前的播放状态：



```
if ([MPMusicPlayerController iPodMusicPlayer].playbackState ==
    MPMusicPlaybackStatePlaying)
```

假如设备正在播放音乐，那么开发者可将当前音乐暂停，同时在程序中显示某个图案，也可以考虑用本章后面讲述的办法，给提示音添加震动效果。要想使用 `MPMusicPlayerController`，必须先引入 `MediaPlayer` 模块。下一节将会详细讲解模块。

我们也可以在程序里调用 `AudioServicesAddSystemSoundCompletion()` 来添加一段回调代码，以便在 `System Sound` 播放完毕之后执行。除非需要接连不断地播放许多时间较短的提示音，否则一般都不会用到这项功能。

以 `System Sound` 对象为参数来调用 `AudioServicesDisposeSystemSoundID`，即可清理该声音。这个函数会把声音对象及相关的资源一并释放。

### 3.11.2 为使用系统框架而引入模块

想使用 `System Sound` 服务，我们必须引入 `Audio Toolbox` 框架及头文件。苹果公司为 `iOS 7` 提供了模块机制，从而简化了开发者在 `Xcode` 中向代码里添加系统框架和头文件时的步骤。

以前我们必须把想要使用的系统框架添加到应用程序的目标中，并在源代码里用 `#include` 宏指令把相关的头文件包含进来。有了模块这一概念之后，只需在源文件顶部使用一条 `@import` 语句，即可将头文件包含进来，而且还会把相关的框架自动连接到当前的项目：

```
@import AudioToolbox;
```

通过模块机制，我们可以更方便地将苹果公司自带的框架添加到自己的项目中。不过，模块的主要用途是增加编译和编制代码索引时的性能。模块中有一份数据库包含了相关框架里的全部符号，以供快速查询之用。据说苹果公司曾经演示过：模块可以缩减程序的构建时间，并能够提升编制代码索引的速度，在不同的项目上面，效率可能会提升 200% 甚至更多。

如果用 `Xcode 5` 来创建新项目，那么默认会启用模块机制。而对于原有的项目来说，则可以在 `Build Settings` 里开启 `Enable Modules` 选项。

所有系统框架都能用作模块。不过，目前并没有办法将自制的框架转换成模块。

### 3.11.3 震动

与提示音一样，震动也能立刻吸引用户注意。同时它还有个优点，就是几乎适用于所有用户，包括听障用户和视障用户在内。目前，只有 `iPhone` 平台支持震动。此外，开发者只应该偶尔使用震动才对，因为它非常消耗设备的电量。

`System Audio` 服务不仅可以用来播放声音，而且也能令设备震动。我们只需像解决方案 3-7 那样，执行下面这行调用语句即可：

```
AudioServicesPlaySystemSound(kSystemSoundID_Vibrate);
```

开发者不能调整与震动相关的参数。每次调用之后，都会产生一秒钟或两秒钟的震动效果。在不支持震动的平台上（例如 `iPod touch` 和 `iPad`），调用该函数是没有效果的，但也不会

产生错误:

```
- (void)vibrate
{
    // Vibrate only works on iPhones
    AudioServicesPlaySystemSound (kSystemSoundID_Vibrate);
}
```

### 3.11.4 警示音

AudioServicesPlayAlertSound 函数可以播放一种既有声音又有震动的警示音 (Alert Sound):

```
AudioServicesPlayAlertSound(mySound);
```

正如解决方案 3-7 所示, 调用上述方法之后, 设备会播放开发者所指定的声音, 而且还有可能震动, 或是再播放一段旋律。在 iPhone 上面, 如果用户开启了 Settings > Sounds > Vibrate on Ring 选项, 那么该函数就会令手机震动。用户若是把手机音量调得很低或是将其设为静音, 则有可能听不到提示音, 然而有了震动效果之后, 用户就可以感觉到了。

解决方案 3-7 用 AudioServices 来播放系统音、警示音, 并产生震动效果

---

```
@implementation SoundPlayer
```

```
void _systemSoundDidComplete(SystemSoundID ssID,
    void *clientData)
```

```
{
    AudioServicesDisposeSystemSoundID(ssID);
}
```

```
+ (void)playAndDispose:(NSString *)sound
```

```
{
    NSString *sndpath = [[NSBundle mainBundle]
        pathForResource:sound ofType:@"wav"];
    if (!sndpath) ||
        (![[NSFileManager defaultManager]
            fileExistsAtPath:sndpath]))
    {
        NSLog(@"Error: %@.wav not found", sound);
        return;
    }
}
```

```
CFURLRef baseURL =
    (CFURLRef)CFBridgingRetain(
        [NSURL fileURLWithPath:sndpath]);
```

```
SystemSoundID sysSound;
AudioServicesCreateSystemSoundID(baseURL, &sysSound);
CFRelease(baseURL);
```

```
AudioServicesAddSystemSoundCompletion(sysSound, NULL,
    NULL, systemSoundDidComplete, NULL);
```

```

    if ([MPMusicPlayerController iPodMusicPlayer].playbackState
        == MPMusicPlaybackStatePlaying)
        AudioServicesPlayAlertSound(sysSound);
    else
        AudioServicesPlaySystemSound(sysSound);
}
@end

```

iPad 和第二代及以后的 iPod touch 都只会通过自带的扬声器来播放声音，但不会震动，因为它们并没有震动功能。第一代 iPod touch（不知现在还能不能找到这种型号！）会在扬声器里播放一段很短的警示旋律（alert melody），并通过耳机来播放开发者所指定的声音。

在播放警示音的时候，iOS 会自动调低当前的音乐音量。假如我们侦测到设备正在播放音乐，那么可以改为播放警示音，而不是系统音（System Sound）。

### 3.11.5 延迟

初次在 iOS 上面播放系统音时，可能会产生延迟。为了避免这一问题，我们可以在应用程序初始化的时候，先播放一段安静的声音，这样的话，稍后播放系统音时就不会有延迟了。

 **提示** 在 iPhone 上面测试的时候，记得不要把手机左侧的静音开关打开。如果打开了，那么系统就不能播放警示音了。很多 iPhone 开发者都忽视了这一问题。假如一定要确保提示音总能播放出来，那么请改用 AVAudioPlayer 类。

### 3.11.6 释放系统音

别忘了释放系统音。我们应该在 dealloc 方法里面处理这些事情，以便在对象的生命期结束时释放相关资源。编写代码时应该考虑到声音的生命期，并且要找到适当的方式将声音资源释放掉。

对于很多应用程序来说，即便在某个对象或整个程序的生命期里一直保留几个声音资源，也不会占用多少内存。但对于另外一些应用程序来说，则必须在播放完声音之后尽快将其清理干净。开发应用程序时，一定要注意声音资源的释放问题，在用完了声音资源之后，务必将其释放。

在新版的 Xcode 中，iOS 模拟器已经完全支持声音回放了。

## 3.12 小结

本章讲述了在应用程序里直接同用户交互的几种途经。大家学会了如何通过视觉、听觉及触觉等形式的提醒机制来吸引用户的注意力，并请求用户即刻做出回应。本章所提供的这些范例既增强了应用程序的互动效果，又利用了 iPhone 所特有的一些功能。在学习下一章之前，大家可以先回顾下面几个问题：

- 当需要用户立刻来操作应用程序时，可以弹出警告视图。它们既能传达信息，又可以提醒用户做出回应。系统自己提供的 `UIAlertView` 功能不多，但是实现起来很快，而且很容易。正如大家在本章中所见，我们可以自己编写一种非常强大的警告视图，以便实现一套非常灵活而且可供定制的操作界面。
- 假如有项任务比较耗时，那么开发者应该以某种反馈方式向用户显示它的执行进度。iOS 提供了许多反馈方式，包括 HUD（heads-up display，直接出现在程序画面上的显示信息）、状态栏上的指示器以及其他种种手段。我们应该把任务中的非 GUI 元素（non-GUI element）移到新的线程里，以避免阻塞主线程。如果有可能的话，还应向用户提供一种可以取消当前任务的手段。
- 偶尔可以用一下本机通知。只有当确实有必要向用户发送通知的时候，才应该显示它们。假如滥用本机通知机制来显示消息，那么用户很快就会讨厌你的程序，并将其从设备中删掉。
- 系统所提供的特性不会完全符合每个应用程序的设计需求，而且系统也不应该把它们全都包办了。你应该尽量用 `UIView` 实例及动画效果构建出符合自己程序的警告视图及菜单。
- 包括响铃及震动在内的各种音频反馈手段都可以提升并丰富应用程序的交互能力。播放系统音时，不会干扰到 iPod 自己的播放，所以它不会打断用户正在聆听的音乐。同时要注意，提示音别播放得太过频繁了。我们应该谨慎而明智地使用警示音，不要令用户感到厌烦。

## 编排视图及其动画效果

开发者可以通过 `UIView` 类及其子类在 iOS 设备的屏幕中显示内容。本章打算从头开始讲解视图。你将学会如何构建、检视并分解视图层级，也会了解到多个视图之间是怎样协同运作的。你会发现，在界面中创建并摆放视图位置的时候是需要一些空间排列知识的，此外，笔者还会讲解如何设定视图在移动和切换时的动画效果。

第 5 章将会介绍 `Auto Layout`（自动布局），它是一套视图布局系统，采用了声明式的、基于约束的模型。在声明式编程中，开发者会向 SDK 描述出应用程序或界面的行为方式，而系统则会在运行期来实施相关规则。我们通过约束规则（`Constraint rule`）来描述界面。约束会影响到视图的排布方式及平面结构。

无论是否使用 `Auto Layout`，都应该理解视图的基本平面特征<sup>①</sup>，因为这对于构建用户界面来说是至关重要的。

### 4.1 视图层级

我们可以用树状层级来分解 iOS 屏幕。从主视窗开始，各种视图都是依照特定的层次来排布的。所有视图都可以有下级，这些下级视图就叫作子视图。包括视窗在内的每个视图都有一份有序列表，用来保存它的下级视图。一个视图可以含有好多个子视图，也可以不包含任何子视图。视图之间的排布方式以及从属关系由应用程序来决定。

子视图是按照由后至前的顺序显示的，就像是叠起来的动画绘制板（`animation cel`，动画赛璐珞、动画胶片）一样。手绘动画时，我们会在很多张透明的板子里绘制卡通图样，然后

① 本章所说的“几何”（`geometry`）一词大多是与“位置”、“尺寸”这些平面特征有关的概念，与经典意义上的“几何学”并不完全相同。——译者注

将其上下叠放起来。每张板子上都只绘制一部分内容，以便使下方的图案能够透过来。我们可以透过没有上色的部分看到它后面那些板子上的图案。视图也是如此，开发者可以在每张视图上面都只绘制一部分内容，然后将其叠放起来，以产生复杂的视觉效果。

图 4-1 以分层的方式剖析了一个常见的应用程序视窗。此处的视窗拥有一套基于 UINavigationController 的体系，其中的各个视觉元件都是分层叠放起来的。视窗（也就是最右侧的那个空白元件）本身拥有导航栏及表格。导航栏拥有按钮及标题标签；而表格也有其自己的子视图。这些物件叠放起来，就构成了整个程序的 GUI。

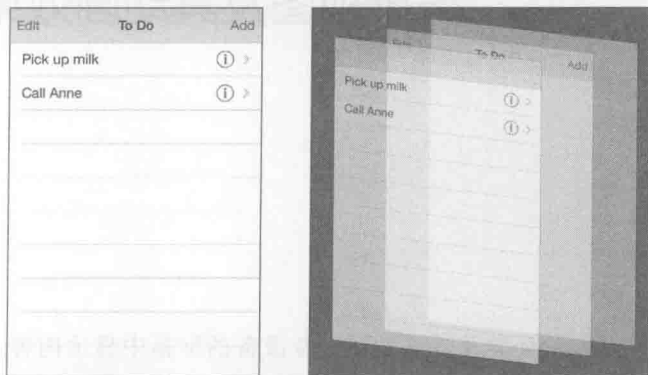


图 4-1 将各个子视图层级叠放起来，以构建复杂的 GUI

程序清单 4-1 列出了图 4-1 中的视窗所具备的视图层级。这个树状结构从顶端的 UIWindow 开始，列出每个子视图所对应的类名。沿着树状结构往下看，会发现导航栏位于第 2 级，导航栏上的按钮位于第 3 级，而表格视图则位于第 4 级，其中的两个单元格位于第 6 级。清单中的某些物件是 iOS 私用的类，它们是在系统排布视图的时候由 SDK 自动添加进去的。例如，UILayoutContainerView 就是个开发者不会直接去使用的类。在 SDK 里，它是 UIWindow 实现代码的一部分。

程序清单 4-1 以有待办清单的方式列出视图层级

```
--[ 1] UIWindow
----[ 2] UINavigationController
-----[ 3] UINavigationControllerWrapperView
-----[ 4] UITableView
-----[ 5] UITableViewWrapperView
-----[ 6] UITableViewCell
-----[ 7] UITableViewCellScrollView
-----[ 8] UITableViewCellContentView
-----[ 9] UILabel
-----[ 8] UITableViewCellDetailDisclosureView
-----[ 9] UIButton
-----[10] UIImageView
-----[ 9] UIImageView
```

```

-----[ 6] UITableViewCell
-----[ 7] UITableViewCellScrollView
-----[ 8] UITableViewCellContentView
-----[ 9] UILabel
-----[ 8] UITableViewCellDetailDisclosureView
-----[ 9] UIButton
-----[10] UIImageView
-----[ 9] UIImageView
-----[ 5] UIImageView
-----[ 5] UIImageView
----[ 2] UINavigationController
-----[ 3] _UINavigationControllerBackground
-----[ 4] _UIBackdropView
-----[ 5] _UIBackdropEffectView
-----[ 5] UIView
-----[ 4] UIImageView
-----[ 3] UINavigationControllerItemView
-----[ 4] UILabel
-----[ 3] UINavigationControllerButton
-----[ 4] UIButtonLabel
-----[ 3] UINavigationControllerButton
-----[ 4] UIButtonLabel
-----[ 3] _UINavigationControllerBackIndicatorView

```

清单里唯一没有列出来的东西是表格中的许多条行分隔线。为了节省篇幅，笔者把它们省略掉了。每条分隔线都是 `UITableViewSeparatorView` 实例。这些分隔线属于 `UITableView`，它们一般出现在第 5 级。

## 4.2 解决方案：用树状图来描述视图层级

每个视图都有其上级视图 (`aView.superview`) 和下级视图 (`aView.subviews`)。我们可以递归地遍历某个视图的所有下级视图，以构建一棵像程序清单 4-1 那样的视图树。解决方案 4-1 会在这张树状图中写出每个视图所对应的类，当它从某个上级视图进入其下级视图的时候，会增加缩进级别。范例代码会把遍历结果保存在可变的字符串 (`NSMutableString`) 中，并返回给调用它的那个方法。

解决方案 4-1 提取视图层级树

```

// Recursively travel down the view tree, increasing the
// indentation level for children
- (void)dumpView:(UIView *)aView atIndent:(int)indent
into:(NSMutableString *)outString
{
    // Add the indentation dashes
    for (int i = 0; i < indent; i++)
        [outString appendString:@"--"];
}

```

```

// Follow that with the class description
[outString appendFormat:@"%2d] %@\n", indent,
    [[aView class] description]];

// Recurse through each subview
for (UIView *view in aView.subviews)
    [self dumpView:view atIndent:indent + 1 into:outString];
}

// Start the tree recursion at level 0 with the root view
- (NSString *)displayViews:(UIView *)aView
{
    NSMutableString *outString = [NSMutableString string];
    [self dumpView:aView atIndent:0 into:outString];
    return outString;
}

```

程序清单 4-1 中的树就是由解决方案 4-1 中的代码所构建的。你可以用 `displayViews:` 方法来重现程序清单 4-1，也可以将其复制到别的应用程序里，以便打印出那些程序的视图层级。



有个针对 `UIView` 的 **Category**，名为 `UIDebugging`，它里面包含了一个“秘密”方法，叫作 `recursiveDescription`。苹果公司的《**Technical Note TN2239**》文档（[https://developer.apple.com/library/ios/technotes/tn2239/\\_index.html](https://developer.apple.com/library/ios/technotes/tn2239/_index.html)）中说：该方法会递归地遍历子视图，并把每个视图的 `description` 添加到遍历结果之中，这样所产生的效果与解决方案 4-1 相似，只是其可配置程度和易读程度稍差一些。由于这是个“私有方法”，所以只应该供调试器（**debugger**）来取用。如果在应用程序代码里通过某种技巧访问了这个方法，那么可能导致程序遭到 **App Store** 拒绝，所以笔者强烈不推荐使用它。解决方案 4-1 所提供的办法更清晰、更灵活，而且也没有使用方面的限制。

### 获取解决方案代码

访问 <https://github.com/erica/iOS-7-Cookbook> 网页，并打开“C04 Views”文件夹，即可找到与本章中的解决方案相对应的完整范例项目。

## 探查 XIB 及故事板中的视图

很多 Xcode 用户会在 **Interface Builder (IB)** 里面创建视图及视图控制器，并用 **XIB** 文件及故事板来构建界面，而不是直接用程序代码去编写。下面这段代码采用解决方案 4-1 所提供的方法剖析从这些资源加载进来的视图：



```

UIView *sampleView = [[[NSBundle mainBundle]
    loadNibNamed:@"Sample" owner:self options:NULL] objectAtIndex:0];
if (sampleView)
{
    NSMutableString *outstring = [NSMutableString string];
    [self dumpView:sampleView atIndent:0 into:outstring];
    NSLog(@"Dumping sample view: %@", outstring);
}

UIStoryboard *storyboard = [UIStoryboard
    storyboardWithName:@"Sample" bundle:[NSBundle mainBundle]];
UIViewController *vc = [storyboard instantiateInitialViewController];
if (vc.view)
{
    NSMutableString *outstring = [NSMutableString string];
    [self dumpView:vc.view atIndent:0 into:outstring];
    NSLog(@"Dumping sample storyboard: %@", outstring);
}

```

解决方案 4-1 的范例代码里面包含了用作样例的 XIB 及故事板文件。读者可自行编辑其内容，然后运行上面这段代码，看看你在 IB 中创建的界面是如何与底层结构相对应的。

### 4.3 解决方案：查询子视图

视图采用数组来保存其子视图，开发者可通过 `subviews` 属性获取这个数组。系统先绘制上级视图，然后再绘制其子视图，而子视图之间的绘制顺序则与它们在 `subviews` 数组中的位置相符。这些子视图按照从后至前的顺序来绘制，它们在数组里的位置也就反映了绘制的顺序。系统会先绘制数组里位置靠前的子视图，然后再绘制位置靠后的子视图。

`subviews` 属性只返回当前视图的直接下属视图。有时候，我们不仅想获取某视图的直接子视图，而且还想获取那些子视图的子视图。解决方案 4-2 所编写的 `allSubviews()` 是个简单的递归函数，可以完整地列出任意视图的全部下属视图。若以某视图所在的视窗（也就是 `view.window`）为参数来调用它，则会列出由该 `UIWindow` 所管理的全部下属视图。如果开发者想搜寻某个特定的视图，比方说某个滑杆（`Slider`）控件或按钮控件，那么这份视图列表会很有用处。

iOS 应用程序一般只有一个视窗，但有的程序可能会包含几个视窗，而每个视窗下面又包含许多视图，某些视图可能还会显示在设备之外的其他屏幕中。我们可以在每个视窗对象上面遍历，以详尽地列出程序里的所有视图。解决方案 4-2 里的 `allApplicationViews()` 函数就是用来做这件事的。调用 `[[UIApplication sharedApplication] windows]` 即可获得一份数组，其中含有应用程序的全部视窗。`allApplicationViews()` 函数会遍历此数组，并把其中的子视图添加到存放返回结果的那个数组里面。

## 解决方案 4-2 与子视图有关的工具函数

```
// Return an exhaustive descent of the view's subviews
NSArray *allSubviews(UIView *aView)
{
    NSArray *results = aView.subviews;
    for (UIView *eachView in aView.subviews)
    {
        NSArray *subviews = allSubviews(eachView);
        if (subviews)
            results = [results arrayByAddingObjectsFromArray:subviews];
    }
    return results;
}

// Return all views throughout the application
NSArray *allApplicationViews()
{
    NSArray *results = [[UIApplication sharedApplication] windows];
    for (UIWindow *window in
        [UIApplication sharedApplication].windows)
    {
        NSArray *subviews = allSubviews(window);
        if (subviews) results =
            [results arrayByAddingObjectsFromArray:subviews];
    }
    return results;
}

// Return an array of parent views from the window down to the view
NSArray *pathToView(UIView *aView)
{
    NSMutableArray *array = [NSMutableArray arrayWithObject:aView];
    UIView *view = aView;
    UIWindow *window = aView.window;
    while (view != window)
    {
        view = [view superview];
        [array insertObject:view atIndex:0];
    }
    return array;
}
```

除了可以查询子视图之外，我们还可以查询每个视图究竟属于那个视窗。UIView 对象的 window 属性指向了包含该视图的视窗。解决方案 4-2 里面也写了个简单的 pathToView() 函数，可以把从视窗到该视图所经的路径放在一份数组里，返回给调用者。为了确定这条路径，此函数会反复向上寻找 superview，直到发现该视图所属的 UIWindow 实例为止。

我们也可以通过另一种办法来查找某视图所属的上级视图。UIView 的 isDescendant-

OfView: 方法能够判断出当前视图是否位于另一个视图的体系之中, 即便那个视图并不是当前视图的直接上级, 该方法也依然能够做出判断。这个方法返回简单的布尔值。YES 表示该视图与开发者经由参数转进来的那个视图之间有上下级关系, 前者是后者的下属。

## 4.4 管理子视图

UIView 类里面有许多方法可供开发者构建并管理视图。我们可以通过这些方法来添加、排列、移除视图, 并查询视图层级。如图 4-1 所示, 视图的层级决定了视图的绘制顺序。只要在应用程序里修改视图之间的层级关系, 就能改变用户所看到的视图布局。下面我们来讲讲如何完成常见的视图管理任务。

### 4.4.1 添加子视图

在视图对象上面调用 addSubview: 方法即可添加子视图。该方法所添加的子视图会出现在视图的最前方, 也就是说, 这个子视图会出现在原有的其他子视图之上。假如想把子视图插入到其他位置, 那么可以使用 SDK 所提供的下列三个工具方法:

- insertSubview:atIndex:
- insertSubview:aboveSubview:
- insertSubview:belowSubview:

这些方法能够控制插入子视图的位置。开发者可以相对于另一个子视图来指定插入点, 也可以指明子视图应该放在 subviews 数组的哪个下标位置上。above 及 below 方法分别会将子视图插入到另一个子视图的前方或后方。对于插入点之后的那些子视图来说, 插入操作会使其下标递增, 但是该操作并不会把现有的子视图替换掉。

### 4.4.2 重排及删除子视图

用户操作应用程序的时候, 程序经常需要重新排列视图, 或者移除某些视图。iOS SDK 提供了许多种简单的方式来完成这些操作, 开发者可以用相关的方法更改视图的位置及内容:

- [parentView exchangeSubviewAtIndex:i withSubviewAtIndex:j] 方法可用来交换两个视图的位置。
- bringSubviewToFront: 及 sendSubviewToBack: 方法可以把子视图前移或后移。
- 调用 [childView removeFromSuperview] 方法, 可以将子视图从其上级视图中移走。假如子视图正显示在屏幕中, 那么执行完该操作后, 它就会从屏幕里消失。

重排、添加或移除视图之后, 系统会自动重绘屏幕上的内容, 以反映新的视图布局情况。

### 4.4.3 UIView 的回调方法

当视图层级有变化时, 系统可以向相关视图发送回调。iOS SDK 提供了六个回调方法, 应用程序可以通过这些方法来追踪视图的移动以及上级视图的变动:

- didAddSubview:——如果有人通过 addSubview: 方法或上一节提到的那几个插

人方法成功地向某个视图里添加了一个子视图，那么系统就会在上级视图上面调用这个方法。我们可以在 `UIView` 的子类里覆写该方法，以便在新的子视图添加进来的时候，执行一些额外的操作。

- **didMoveToSuperview:**——如果有人已经把某个子视图移动到另一个新的上级视图名下，那么系统就会在子视图上调用该方法。此时子视图可以用某种方式来响应新的上级视图。如果开发者把子视图从其上级视图中移除，那么系统也会调用该方法，只不过此时子视图的 `Superview` 是 `nil`。
- **willMoveToSuperview:**——在子视图即将变更其上级视图时，系统会调用该方法。
- **didMoveToWindow:**——它的回调时机和 `didMoveToSuperview` 相仿，但只有当视图移动到新的视窗层级（Window hierarchy）而不是仅仅改换其上级视图时，系统才会调用它。如果想通过 AirPlay 技术在设备之外的屏幕上显示内容，那么一般都需要用到这个方法。
- **willMoveToWindow:**——在子视图即将移动到别的视窗层级时，系统会调用该方法。
- **willRemoveSubview:**——如果某个子视图即将从其上级视图中移除，那么系统会在上级视图上调用该方法。

这些方法很少会用到，然而一旦需要用到，它们就总能帮上大忙，因为开发者无须预先知道子视图或上级视图所属的类，即可为视图添加新的行为。与 Window 有关的回调主要用于在另外一个 `UIWindow` 中显示某种视图，例如警告视图或是带键盘的输入界面等。

## 4.5 为视图设定标签并查找视图

iOS SDK 内置了一套搜寻机制，可通过标签（tag）来查找子视图。标签是个表示视图身份的数字，一般来说，是个正整数。我们可以通过视图的 `tag` 属性来设置这个标记，比方说，`myView.tag = 101`。在 IB 中，我们可通过 Attributes Inspector 来设置视图的标签。图 4-2 演示了如何在 Attributes Inspector 的视图区域中指定标签。

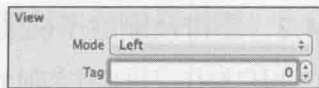


图 4-2 通过 IB 的 Attributes Inspector 为视图设定标签

标签的值可随意选取。系统只预留了 0 这个特殊值，对于新创建出来的视图来说，其标签值默认是 0。至于具体应该如何为视图选取标签以及用什么值来当标签，都由开发者自己决定。凡是 `UIView` 的子类的实例都能够打上标签标记，包括视窗和各种控件在内。假如程序里有很多按钮和开关控件，那么可以为其设定不同的标签，使我们能够区分出用户操作的到底是哪一个控件。开发者可在回调方法里加入一套简单的 `switch` 语句，以便根据标签值做出不同的反应。

苹果公司很少给予视图设置标签。笔者唯一碰到的例外出现在 `UIAlertView` 之中，该类会给按钮分别设置值为 1、2 等的标签，但那已经是很久以前的事了。（也许是苹果公司不小心把设置标签的代码遗留到了这个类里吧。）假如你担心和苹果公司的标签发生冲突，那么可以令自己的标签值从 10 或 100 开始，或是选择一个比苹果公司可能用到的标签更大的起始值。

## 使用标签来查找视图

有了标签之后，我们无须在程序里传递各种用户界面元件，即可直接通过上级视图来搜寻其下的子视图。`viewWithTag:` 方法可以在某视图的所有下属中根据标签值来查找相关的子视图。搜寻过程是递归式的，所以即便符合此标签的子视图不是当前视图的直接下属，该方法也依然能找得到。我们可以调用 `[window viewWithTag:101]` 语句，从 Window 开始，沿着整个树状层级及其中的各个分支来搜索标签值是 101 的那个视图。假如有多个视图都具备相同的标签值，那么该方法会返回首先找到的那个。

`viewWithTag:` 方法唯一的缺点在于它返回的是个 `UIView` 对象。也就是说，开发者通常需要先把它转换为适当的类型，然后才能使用。例如，我们需要用下面这两行代码来查找 `UILabel` 控件并设置其文本：

```
UILabel *label = (UILabel *)[self.view.window viewWithTag:101];
label.text = @"Hello World";
```

## 4.6 解决方案：通过对象关联机制为视图设定名称

虽说标签是一种识别视图对象的便捷手段，但某些开发者还是喜欢用名字而非数字来指代视图对象。使用名字的好处是可以令视图对象在标签之外多具备一种表达含义的方式，以便使开发者能够辨明其身份。这样一来，我们就不用再说“标签值是 101 的那个视图”了，而是可以把某个开关控件取名叫作 `Ignition Switch`（点火开关），这个名字本身就能描述其用途，而且与纯数字相比，它还能充当“自文档”：

```
// Toggle switch
UISwitch *s = (UISwitch *)[self.view viewNamed:@"Ignition Switch"];
[s setOn:!s.isOn];
```

我们很容易就能给 `UIView` 添加 `nametag` 属性，并使开发者可以按照名称来查找视图。其中用到的关键技术就是 Objective-C 运行期程序库里面的那几个关联对象（*associated object*）函数<sup>②</sup>。若是曾为某个类编写过 `category` 类，那么你可能会想到这样一个问题：假如现在要给类里再添加一个实例变量，那岂不是要创建新的子类，并把它放在子类里面才行吗？其实 `Associated Object` 机制并不需要向类里添加新的实例变量，它只是提供了一种在本对象的直接存储区之外使用键值对的手段，开发者可以把存储在别处的某份信息关联到当前这个对象上面。

解决方案 4-3 针对 `UIView` 类创建了 `category`，以实现 `nametag`。这个 `category` 为 `UIView` 添加了名为 `nametag` 的新属性，并通过相关的 `Associated Object` 函数来实现该属性，另外，它还提供了 `viewNamed:` 方法，该方法可根据名称来寻找子视图。这个方法会沿着视图层级进行深度优先式的递归搜索，并返回其名称与待查字符串相符的首个子视图。

② 具体指的是 `objc_getAssociatedObject` 及 `objc_setAssociatedObject` 函数。——译者注

## 在 Interface Builder 中设定视图名称

给视图起了名字之后，我们就能依照名称来获取子视图，而不用再声明 IBOutlet 实例变量了。（至于此做法能否令代码变得更易读懂且更易维护，则不在本节讨论范围内。）早前我们曾经写了一段范例代码，用以切换界面中的开关控件。现在可以在 IB 里面把那个开关控件的名称（也就是 Ignition Switch）设为一个运行期属性。

图 4-3 演示了这一操作。选中任意视图，然后打开 Identity Inspector（可通过 View > Utilities > Show Identity Inspector 菜单项打开它）。找到 User Defined Runtime Attributes 区域，然后点击“+”按钮，添加新的属性。把 Key Path 设为 nametag（解决方案 4-3 会针对 UIView 类编写 category，而此处的 Key Path 要和 category 中所用的属性名相符），将 Type 设为 String，并将 Value 设为视图的新名字。保存所做的修改。现在，我们就可以在代码中使用 category 里所定义的 viewNamed: 方法来获取开关控件并切换其状态了。

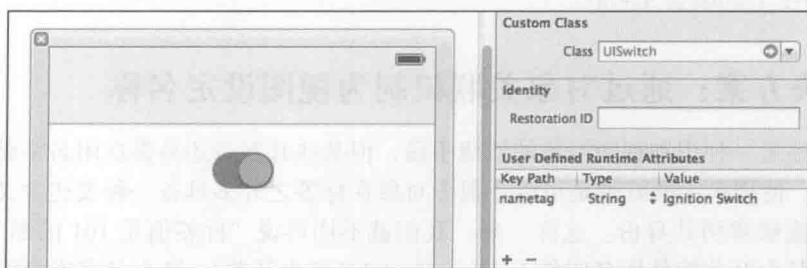


图 4-3 通过 IB 的 Attributes Inspector，我们可以为任意视图设定标签。此外，也可以在 Attributes Inspector 里面配置 User Defined Runtime Attributes，并在程序代码里通过 KVC 机制（Key-value coding，键值编码）来获取这些值。系统加载 XIB 文件的时候，会设定好这些值

### 解决方案 4-3 为 UIView 添加命名功能

```
#import <objc/runtime.h>
@implementation UIView (NameExtensions)

// Static variable's address acts as the key
// Thanks, Oliver Drobnik
static const char nametag_key;

- (id)nametag
{
    return objc_getAssociatedObject(self, (void *) &nametag_key);
}

- (void)setNametag:(NSString *)theNametag
{
    objc_setAssociatedObject(self, (void *) &nametag_key,
        theNametag, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}

- (UIView *)viewWithNametag:(NSString *)aName
```

```

{
    if (!aName) return nil;

    // Is this the right view?
    if ([self.nametag isEqualToString:aName])
        return self;

    // Recurse depth first on subviews
    for (UIView *subview in self.subviews)
    {
        UIView *resultView = [subview viewNamed:aName];
        if (resultView) return resultView;
    }

    // Not found
    return nil;
}

- (UIView *)viewNamed:(NSString *)aName
{
    if (!aName) return nil;
    return [self viewWithTag:aName];
}
@end

```



还有一种给视图取名的办法，就是直接设定 CALayer 的名称，而不使用 Associated Object 机制。CALayer 实例有个 name 属性，能够用来区分不同的 CALayer。使用它之前，需要先引入 Quartz Core 模块，在代码中，可通过 view.layer 访问这个 CALayer。

## 4.7 视图的几何特征

在苹果公司引入了 Auto Layout 之后，开发者就很少需要像原来那样直接调整视图的平面特征了，不过，某些情况下，我们还是需要直接去控制视图的这些特性。而且，苹果公司所引入的一些新 API（比如与视图的物理效果有关的那些 API）就与 Auto Layout 配合得不够好。因此，还是应该掌握一些操作并调整视图平面特征的基本方式。

几何属性定义了视图出现的位置、视图的尺寸以及方向。即便采用了 Auto Layout，这些属性也依然有效，在 Auto Layout 环境下，它们会由约束系统来管理。开发者仍然能够查看这些属性，并以此来了解视图的位置及系统对视图所做的几何变换。

操作动态视图时，由于视图的生命期比较短，所以在显示期间，其平面特征会持续变动，于是，我们需要抛开约束系统，直接去处理每个视图的基本布局。UIView 类提供了两个内置的属性，用于定义与布局有关的特征。



每个视图都使用框架来定义其边界。框架描述了视图的轮廓，也就是它在上级视图坐标系中的位置、宽度及高度，而相关的 `bounds` 及 `center` 属性则分别定义了本视图坐标系中的框架矩形以及框架在上级坐标系中的几何中心。这些属性紧密地集成在一起。

改变了视图的框架之后，视图会更新自己，以便同新的框架相符。假如 `frame` 的宽度变大了，那么视图就会拉伸，假如框架的位置变了，那么视图就会移动到新位置。视图的 `frame` 描绘了其轮廓。本视图的尺寸并不局限于其上级视图的尺寸，也不受屏幕大小的限制。视图可以比屏幕大，也可以比屏幕小。同理，它可以比上级视图大，也可以比上级视图小。子视图若是比上级视图还大，其可见区域则会越过上级视图的边界。开发者可以通过上级视图的 `clipsToBounds` 属性来限定子视图的可见内容，将其局限在上级视图的范围里。

视图还有个 `transform` 属性，使开发者可以经由仿射变换来修改其样貌。有一些数学公式可用来调整视图的二维几何特征。通过变换，可以拉伸或挤压视图，也可以对其进行转转，使之变得倾斜。将框架与 `transform` 属性相搭配，即可完全定义出视图的基本平面特征。

#### 4.7.1 框架

框架矩形就是本视图在其上级坐标系中的轮廓线。我们用 `CGRect` 结构来表示框架，由该结构的 CG 前缀可知，它是 Core Graphics 框架的一部分。`CGRect` 由原点 and 尺寸构成，前者是个 `CGPoint`，其中含有横坐标和纵坐标，后者是个 `CGSize`，其中含有宽度与高度。当在 Auto Layout 环境之外创建视图时，我们通常会在分配好 `UIView` 之后，以 `rect` 为参数来调用 `initWithFrame:` 方法。

```
CGRect rect = CGRectMake(0.0f, 0.0f, 320.0f, 416.0f);
myView = [[UIView alloc] initWithFrame:rect];
```

视图里面有两个联系紧密而且非常关键的 `CGRect` 型属性，即 `frame` 和 `bounds`。前者与后者之间的区别在于，它们所参照的坐标系不同。`frame` 是按照上级视图的坐标系来定义的，而 `bounds` 则按当前视图自己的坐标系来定义。鉴于此，视图的 `bounds` 的原点一般设为 0，而它的坐标系通常也从左上角开始。对于某些视图，比如 `UIScrollView` 来说，`bounds` 可能会超过其可视范围。

#### 4.7.2 与 CGRect 有关的工具函数

在前一节中我们看到，`CGRectMake()` 函数可以根据四个参数来新建矩形，这四个参数分别是原点的横坐标、纵坐标、矩形宽度及高度，它是个创建框架时所需的关键函数。除了 `CGRectMake()` 之外，还有一些比较方便的函数，也可以用来操作 `CGRect` 及 `frame`：

- `NSStringFromCGRect(aCGRect)` 可把 `CGRect` 结构转换为具有固定格式的字符串。在调试的时候，开发者可通过这个函数把视图的框架打印到控制台。
- `CGRectFromString(aString)` 函数可以根据字符串中的信息重建矩形。如果把视图的框架以字符串的形式放在 `NSUserDefaults` 里面，那么该方法可以将其转回 `CGRect`。
- 虽说 `[NSValue valueWithCGRect:rect]` 不算是函数，但它可以用来根据传进来



的矩形创建新的 Objective-C 对象，并把矩形的信息存储到那个 `NSValue` 里面。然后，开发者可以根据需要，把对象添加到字典或数组中。`CGRectValue` 方法可以从 `NSValue` 对象里面取出 `CGRect` 结构体。对于 Core Graphics 中的大多数类型来说，也都有相似的函数和方法，例如 `CGPoint`、`CGSize` 及 `CGAffineTransform` 等。

- `CGRectInset(aRect, xinset, yinset)` 函数可以创建出与源矩形中心点相同但尺寸较小或较大的矩形来。如果 `inset` 为正值，那么新矩形就比原来小，若为负，则比原来大。
- `CGRectOffset(aRect, xoffset, yoffset)` 可以创建出与源矩形大小相同但位置不同的矩形，`xoffset` 及 `yoffset` 分别表示横向和纵向偏移量。该函数很适合在移动框架的时候使用，也可以用来创建简单的阴影效果。
- `CGRectGetMidX(aRect)` 及 `CGRectGetMidY(aRect)` 函数分别获取矩形中心点的横坐标和纵坐标。这两个函数很适合用来查询 `bounds` 及 `frame` 的中心点。
- `CGRectIntersectsRect(rect1, rect2)` 可判断出两个 `CGRect` 结构体是否相交。此函数可用于检测两个长方形对象有没有重叠。调用 `CGRectIntersection(rect1, rect2)`，即可得知发生重叠的具体部位。若没有重叠，则返回空矩形。（用 `CGRectIsNull(rect)` 能够判断出矩形是不是空矩形。）还有个与 `CGRectIntersectsRect` 相关的函数叫作 `CGRectContainsPoint(rect, point)`，如果给定的点位于（非空的）矩形之内，那么此函数返回 `true`。
- `CGRectEqualToRect(rect1, rect2)` 用来比较两个矩形是否相同。该函数会判断两个矩形的尺寸及位置是否完全一样。相似的函数还有 `CGSizeEqualToSize(size1, size2)` 及 `CGPointEqualToPoint(point1, point2)`，它们分别用于比较 `CGSize` 及 `CGPoint` 实例。
- 还有几个便捷的工具函数：`CGRectDivide()` 能够把源矩形分成两部分，而 `CGRectApplyAffineTransform(rect, transform)` 则可以对矩形执行仿射变换，并把能够包含变换结果的最小矩形返回给调用者。
- `CGRectZero` 是个矩形常量，它位于 (0,0) 这个点，且宽、高均为 0。如果在创建框架的时候不能确定其大小和位置，那么可以使用此常量来表示。类似的常量还有 `CGPointZero` 及 `CGSizeZero`。

### 4.7.3 CGPoint 与 CGSize

`CGRect` 结构体由两个小的结构体组成，一个是 `CGPoint`，它定义了矩形的原点，另一个是 `CGSize`，它定义了矩形的边界。`CGPoint` 用 `x` 和 `y` 坐标来描述矩形的位置，而 `CGSize` 里面则有 `width` 及 `height`。`CGPointMake(x, y)` 用来创建 `CGPoint`，`CGSizeMake(width, height)` 用来创建 `CGSize`。尽管这两种结构体看上去似乎一样（都包含两个浮点值），但从语义角度讲，iOS SDK 却认为它们是不同的东西。`CGPoint` 用来表示位置，而 `CGSize` 用来表示尺寸。我们不能把某个 `CGSize` 设为 `myFrame.origin`。

由于 `CGRect`、`CGPoint` 及 `CGSize` 都是结构体，所以可以使用很多种灵活的写法来

初始化它们：

```
CGPoint origin = {0, 0};
CGSize size = {100, 200};
CGRect rect1 = CGRectMake(0, 0, 100, 200);
CGRect rect2 = {{0, 0}, {100, 200}};
CGRect rect3 = {origin, size};
CGRect rect4 = {origin, {100, 200}}
CGRect rect5 = {.size.width = 100, .size.height = 200, .origin = {0, 0}};
```

上述五个 CGRect 完全相同。

与 CGRect 一样，你也可以在其他几种结构体与字符串之间来回转换。NSStringFromCGPoint()、NSStringFromCGSize()、CGPointFromString() 及 CGSizeFromString() 函数可以执行相关的操作。另外，也可以把 CGPoint 及 CGSize 转换成字典，反之亦然。

#### 4.7.4 CGAffineTransform

iOS SDK 在 Core Graphics 里面实现了仿射变换功能。仿射变换可以将一套坐标系统变换成另一套坐标系统，这些功能广泛地运用于二维动画和三维动画之中。UIKit 版本的仿射变换采用  $3 \times 3$  的矩阵来定义 UIView 的变换效果，也就是说，它只支持二维的变换效果。三维变换需要使用  $4 \times 4$  的矩阵，Core Animation 中的 CALayer 使用的正是此种矩阵。通过仿射变换，我们可以实时地对视图执行缩放、平移及旋转操作。设置视图的 transform 属性，即可实现变换。例如：

```
float angle = theta * (PI / 100.0);
CGAffineTransform transform = CGAffineTransformMakeRotation(angle);
myView.transform = transform;
```

变换总是针对视图的中心点而实施的。所以，如果用上面这段代码对视图进行旋转，那么旋转时所围绕的点是视图的中心点。假如想绕着其他点旋转，那么必须先将视图平移到那个点，然后旋转，最后再把视图移回来。有很多种办法可以简化上述操作，其中包括直接操作视图的 layer 属性，不过，这些办法并不在本章的讨论范围内。

如果想撤销变换效果，那么可以把 transform 属性设为 CGAffineTransformIdentity。这样做会把视图的 frame 恢复到它目前的值：

```
myView.transform = CGAffineTransformIdentity;
```



在 iOS 系统中，y 坐标从顶部开始向下增长，这与 PostScript 所使用的坐标系统相似，但是却和 Quartz 的坐标系统相反，基于历史原因，Mac 使用的是 Quartz 坐标系统。在 iOS 系统中，原点位于左上角，而不是左下角。iOS 一直都在把 Quartz 及 Core Graphics 里的许多特性移植到 UIKit，使开发者在调整文本位置及处理图像时，不用总是像原来那样反转 y 轴。

### 4.7.5 坐标系统

早前说过，描述视图时，可以采用两套坐标系统。frame 和 center 是按照上级视图的坐标系来定义的，而 bounds 及子视图则参照当前视图的坐标系统来描述。只要本视图和其上级视图受同一个 UIWindow 管理，我们就能用 iOS SDK 所提供的许多工具方法在这两套坐标系统之间转换。convertPoint:fromView: 方法可把某点在另一坐标系中的坐标转换成它在本坐标系里的坐标。例如：

```
myPoint = [myView convertPoint:somePoint fromView:otherView];
```

假如这个点表示某对象的位置，那么转换之后的点坐标依然能够表示位置，只不过这次是在 myView 的角度来描述位置的。与该方法相反，convertPoint:toView: 方法则能够用另一个视图的坐标系来描述本坐标系中的点。convertRect:toView: 及 convertRect:fromView: 方法的功能与上述两方法相似，它们适用于 CGRect 型结构体，而非 CGPoint 型结构体。

请注意，iOS 设备的坐标系统与显示该系统的像素系统（pixel system）未必相符。比方说，iPhone 4S 采用 640 × 960 像素的 Retina 显示屏，其像素是离散的（discrete）<sup>①</sup>，而 SDK 却使用 320 × 480 的连续（continuous）坐标系统来表示那些像素，此系统的计量单位是点（point）。对于配有 Retina 显示屏的设备来说，尽管开发者可以用高质量的图片来填充那些像素，但代码中以点为单位所指定的坐标其坐标系依然参照的是像素密度较低的设备。无论显示屏的像素密度如何，对于显示屏为 3.5 英吋的 iPhone 及 iPod touch 来说，屏幕中心点的坐标大概都是 (160.0, 240.0)。而在配有 4 英吋 Retina 显示屏的 iPhone 及 iPod touch 上面<sup>②</sup>，中心点的坐标则是 (160.0, 284.0)。



**提示** UIScreen 类提供了名为 scale 的属性，用来表示显示屏的像素密度与点坐标系统之间的关系。通过该属性，我们可以把视图中逻辑坐标系里的点坐标（一个点大约等于 1/160 英吋）转换成设备的物理像素坐标。在配有 Retina 显示屏的设备中，scale 值是 2.0，而在非 Retina 显示屏的设备上，则是 1.0。

## 4.8 解决方案：操控视图的框架

如果手动修改了视图的框架，而不是令 Auto Layout 机制自动去调整的话，那么实际上就等于改动了它的尺寸（包含宽度和高度）以及位置。比方说，我们可以用下列代码来移动某个视图的框架：

```
CGRect initialRect = CGRectMake(0.0f, 0.0f, 100.0f, 100.0f);
myView = [[UIView alloc] initWithFrame:initialRect];
[topView addSubview:myView];
myView.frame = CGRectMake(0.0f, 30.0f, 100.0f, 100.0f);
```

① 大意是像素的坐标值必须是整数，而连续的意思则是坐标值可以取小数。——译者注

② 这种设备（比如 iPhone 5）的物理像素是 640 × 1136，而开发者使用的坐标系中则是 320 × 568。——译者注

上述代码会在 (0.0, 0.0) 点创建名为 myView 的子视图，然后将其移动到 (0.0, 30.0)。

这种移动视图的方式不太常用。iOS SDK 也不希望开发者通过修改 frame 的办法来移动视图。相反，它关注的是视图的位置。iOS 推荐的视图移动方式是：设置视图的 center 属性。这是个属于 UIView 的属性，我们可以直接修改它的值：

```
myView.center = CGPointMake(160.0f, 55.0f);
```

你可能认为，SDK 里面应该提供了一种通过移动原点来更新视图位置的方式，但实际上它没有提供。不过我们只需针对自己的视图类构建 category，即可实现此功能。把视图当前的 frame 取出来，将所需的点设置成原点 (origin)，再把修改后的 CGRect 设置回去。下面代码可以为我们自己的视图类创建新的 origin 属性，令开发者可以由此来获取并修改视图的原点：

```
-(void)setOrigin:(CGPoint)aPoint
{
    CGRect newFrame = self.frame;
    newFrame.origin = aPoint;
    self.frame = newFrame;
}
```

在上述扩展代码中，笔者使用了 origin (原点) 这个很常见的词来做属性名称，假如苹果公司将来也实现了“设定视图原点”这一功能，那么这段范例代码就会因为名称重复 (name overlap) 而失效。本书的范例代码经常使用这种常见词汇。这是为了令代码更容易读懂，同时也是为了使读者能够更快地理解我们想要演示的概念。编写真正的应用程序时，不要使用这种容易引发冲突的名称，而是应该将你的姓名或公司的首字母缩写当成前缀，这样有助于将内部代码同其他代码区分开。

移动视图对象时，不用担心曝露出来或隐藏起来的那些矩形区域，因为 iOS 会自动把视图重新绘制好。开发者只需把视图当成可以来回操作的实际物件就行，绘制问题由 Cocoa Touch 处理。

#### 4.8.1 调整视图的尺寸

最简单的一种用法就是通过 frame 和 bounds 来控制视图的尺寸。前面说过，frame 是以上级视图的坐标系来定义视图位置的。假如把 frame 的原点设为 (0.0, 30.0)，那么本视图的左边界就会与上级视图对齐，而上边界则会偏离上级视图顶端 30 个点。在非 Retina 显示屏上，这相当于比上级视图的顶端低了 30 个像素，而在 Retina 显示屏上，则低了 60 像素。

bounds 是以视图自己的坐标系来定义的。因此，视图的 bounds 属性 (即 myView.bounds) 其 origin 一般都是 (0.0, 0.0)。对于大多数视图来说，bounds 的宽和高都与视图的范围相同，也就是和 frame 的 size 属性相同。(对于某些类来说，则通常不是这样的，例如，UIScrollView 的范围就可能会超过当前可以显示出来的这一部分。)

调整 frame 或 bounds 属性即可修改视图的尺寸。实际上，我们修改的是这些结构体

里面的 `size` 字段。与实现原点移动功能时所用的办法类似，我们也可以创建一个工具方法，令开发者能够直接修改视图尺寸：

```
- (void)setSize:(CGSize)aSize
{
    CGRect newbounds = self.bounds;
    newbounds.size = aSize;
    self.bounds = newbounds;
}
```

如果某个视图正显示在屏幕上，而开发者又修改了它的尺寸，那么系统会自动更新该视图。系统还根据视图里所摆放的 UI 元件以及视图本身所属的类来决定是对其中的子视图进行缩放，还是移动它们的位置，使之与上级视图的新尺寸相符，此外，系统也有可能裁切子视图的可视范围。具体做法取决于一系列标志属性的取值以及视图是否处在 Auto Layout 系统中：

- `autoresizesSubviews` 属性决定了当视图的 `bounds` 发生变化时，其中的子视图是否会自动缩放。
- `autoresizingMask` 属性决定了当上级视图的 `bounds` 有变化时，本视图应该如何响应。假如视图处在约束系统之中，那么该属性表示会被忽略，iOS 的 Auto Layout 系统会自动调整其尺寸。
- `clipsToBounds` 标志决定了系统是否应该把子视图里面超出本视图 `bounds` 的部分显示出来。如果开启了此标志，那么系统只会将位于本视图 `bounds` 之内的部分子视图内容显示出来。开发者可以在本视图上面调用 `sizeToFit` 方法，令其重新调整自身大小，以便把所有子视图都囊括进来。
- `contentMode` 属性与其他几个涉及视图缩放的属性都有关联，不过它主要用于指明视图的 `CALayer`（也就是其内容位图（content bitmap）在 `bounds` 发生变化时应该如何调整。开发者可以在一系列包含“缩放”、“居中”以及“自适应”的选项之中选定该属性的取值，这个属性一般是在操作 `UIImageView` 时使用的。



提示

视图的 `bounds` 会受到 `transform` 属性影响，而 `transform` 是个用于改变视图显示方式的数学矩阵。假如要通过 `transform` 来调整视图的样貌，那么就on不要同时去操作视图的 `frame` 了，否则可能会产生不符合预期的结果。（本章稍后将会给出一些解决办法。）比方说，在执行了变换之后，`frame` 的原点在数学意义上可能就和 `bounds` 的原点不相符了。如果要修改视图的样貌，那么通常的操作顺序是：尽可能先修改它的 `frame` 或 `bounds`，然后设置其 `center`（中心点），最后再运用 `transform`（变换）。

有的时候，在把某个视图添加到新的上级视图之前，我们想先修改其大小。比方说，现在要把 `ImageView` 放到 `AlertView` 里面。为了能在不改变宽高比的情况下把前者放在后

者之中，我们可调用下列方法，适当地缩小视图的高度及宽度：

```
- (void)fitInSize:(CGSize)aSize
{
    CGFloat scale;
    CGRect newframe = self.frame;

    if (newframe.size.height > aSize.height)
    {
        scale = aSize.height / newframe.size.height;
        newframe.size.width *= scale;
        newframe.size.height *= scale;
    }

    if (newframe.size.width > aSize.width)
    {
        scale = aSize.width / newframe.size.width;
        newframe.size.width *= scale;
        newframe.size.height *= scale;
    }

    self.frame = newframe;
}
```

## 4.8.2 CGRect 与中心点

前面说过，UIView 实例使用了 CGRect 结构体，而该结构体中包含 origin 及 size 字段，这两个字段用来定义视图的 frame。我们无法通过 CGRect 结构体直接查询中心点。可是在另一方面，当开发者移动视图的时候，UIView 却要依靠其 center 属性来把自己放在新的中心点上。麻烦的地方在于，Core Graphics 并不认为中心点是用来描述矩形的主要概念，所以 Core Graphics 里内置的函数只能分别获取矩形中心点在 x 轴和 y 轴方向上的坐标，而不能直接获取这个中心点。

我们可以构建一种函数，在基于原点的 CGRect 结构体和基于中心点的 UIView 对象之间进行转换。用矩形中心点的 x 坐标值和 y 坐标值构建一个 CGPoint，即可实现此函数。该函数接受一个参数，也就是矩形，并返回其中心点：

```
CGPoint CGRectGetCenter(CGRect rect)
{
    CGPoint pt;
    pt.x = CGRectGetMidX(rect);
    pt.y = CGRectGetMidY(rect);
    return pt;
}
```

我们可以模仿 UIView 的工作方式再编写一个有用的函数，令开发者能够通过移动中心点来移动矩形。比方说，我们要把某视图移动到新的位置，但又要确保移动后的视图依然处在上级视图的框架之内。为了在移动之前先做测试，可用下面这个函数模拟一下当前的框架在视图移动到新的中心点之后会位于何处：

```
CGRect CGRectMoveToCenter(CGRect rect, CGPoint center)
{
    CGRect newrect = CGRectZero;
    newrect.origin.x = center.x-(rect.size.width/2.0);
    newrect.origin.y = center.y-(rect.size.height/2.0);
    newrect.size = rect.size;
    return newrect;
}
```

然后, 用 `CGRectContainsRect()` 函数判断移动后的 frame 与上级视图的 frame 之间是何关系, 以此确保移动后的视图不会超出其容器的范围。

我们通常需要把一个视图放在另一视图的中心位置上。为了实现居中, 可以把本视图所在的矩形 (subRect) 和外围视图所在的矩形 (mainRect) 传给下面这个函数, 以获取居中后的矩形, 并据此来设定本视图的位置。假如要把本视图添加成外围视图的子视图 (子视图的坐标系必须从 (0, 0) 开始), 那么 mainRect 参数就应该是外围视图的 bounds; 若要令本视图从属于外围视图的上级视图, 那么 mainRect 参数就应该是外围视图的 frame:

```
CGRect CGRectCenteredInRect(CGRect subRect, CGRect mainRect)
{
    CGFloat xOffset = CGRectGetMidX(mainRect)-CGRectGetMidX(subRect);
    CGFloat yOffset = CGRectGetMidY(mainRect)-CGRectGetMidY(subRect);
    return CGRectOffset(rect, xOffset, yOffset);
}
```

### 4.8.3 视图的其他几何特征

正如大家在前面看到的那样, 我们可以给视图添加 origin 及 size 属性, 使开发者能够像使用 center 属性那样来使用这些属性, 以便更好地同 Core Graphics 里的函数相集成。用相似的方式, 还可以把包括 width (宽度) 及 height (高度) 在内的其他属性曝露给开发者, 另外, 也可以提供以点为计量单位的 left (左边界)、right (右边界)、top (上边界)、bottom (下边界) 等基本几何特征。从某种程度上来看, 这与苹果公司的设计风格不符。苹果公司的风格是在不曝露底层结构体的实现细节这一前提下, 把其中某些特征提供给开发者使用。但从另一个角度来看, 我们也可以说, 上面提到的那些 width、height、left 等本身就应该设计成 UIView 的属性。因为它们确实反映了视图的一些基本特征, 所以理应作为属性提供给开发者来使用。

解决方案 4-4 针对 UIView 类编写了名为 ViewFrameGeometry 的 category, 并提供了一整套与 frame 有关的工具, 而读者可以自己来决定是否将这些属性公布出来。该 category 中的属性值都没有考虑 transform (变换)。



有了 Auto Layout 机制之后, 这个 category 中的许多工具方法都显得不那么重要了, 不过它们仍然很有价值, 因为有的时候, 你必须手工指定视图布局, 甚至在使用了 Auto Layout 的情况下, 偶尔也还是会用到它们的。



#### 解决方案 4-4 在 UIView 的 category 中添加与 frame 几何特征有关的属性

```

@interface UIView (ViewFrameGeometry)
@property CGPoint origin;
@property CGSize size;

@property (readonly) CGPoint midpoint;

// topLeft is synonymous with origin so not included here
@property (readonly) CGPoint bottomLeft;
@property (readonly) CGPoint bottomRight;
@property (readonly) CGPoint topRight;

@property CGFloat height;
@property CGFloat width;
@property CGFloat top;
@property CGFloat left;
@property CGFloat bottom;
@property CGFloat right;

- (void)moveBy:(CGPoint)delta;
- (void)scaleBy:(CGFloat)scaleFactor;
- (void)fitInSize:(CGSize)aSize;
@end

@implementation UIView (ViewGeometry)
// Retrieve and set the origin
- (CGPoint)origin
{
    return self.frame.origin;
}

- (void)setOrigin:(CGPoint)aPoint
{
    CGRect newFrame = self.frame;
    newFrame.origin = aPoint;
    self.frame = newFrame;
}

// Retrieve and set the size
- (CGSize)size
{
    return self.frame.size;
}

- (void)setSize:(CGSize)aSize
{
    CGRect newFrame = self.frame;
    newFrame.size = aSize;
    self.frame = newFrame;
}

```



```

// Query other frame locations

- (CGPoint)midpoint
{
    // midpoint is with respect to a view's own coordinate system
    // versus its center, which is with respect to its parent
    CGFloat x = CGRectGetMidX(self.bounds);
    CGFloat y = CGRectGetMidY(self.bounds);
    return CGPointMake(x, y);
}

- (CGPoint)bottomRight
{
    CGFloat x = self.frame.origin.x + self.frame.size.width;
    CGFloat y = self.frame.origin.y + self.frame.size.height;
    return CGPointMake(x, y);
}

- (CGPoint)bottomLeft
{
    CGFloat x = self.frame.origin.x;
    CGFloat y = self.frame.origin.y + self.frame.size.height;
    return CGPointMake(x, y);
}

- (CGPoint)topRight
{
    CGFloat x = self.frame.origin.x + self.frame.size.width;
    CGFloat y = self.frame.origin.y;
    return CGPointMake(x, y);
}

// Retrieve and set height, width, top, bottom, left, right
- (CGFloat)height
{
    return self.frame.size.height;
}

- (void)setHeight:(CGFloat)newHeight
{
    CGRect newFrame = self.frame;
    newFrame.size.height = newHeight;
    self.frame = newFrame;
}

- (CGFloat)width
{
    return self.frame.size.width;
}

```

```

- (void)setWidth:(CGFloat)newWidth
{
    CGRect newFrame = self.frame;
    newFrame.size.width = newWidth;
    self.frame = newFrame;
}

- (CGFloat)top
{
    return self.frame.origin.y;
}

- (void)setTop:(CGFloat)newTop
{
    CGRect newFrame = self.frame;
    newFrame.origin.y = newTop;
    self.frame = newFrame;
}

- (CGFloat)left
{
    return self.frame.origin.x;
}

- (void)setLeft:(CGFloat)newLeft
{
    CGRect newFrame = self.frame;
    newFrame.origin.x = newLeft;
    self.frame = newFrame;
}

- (CGFloat)bottom
{
    return self.frame.origin.y + self.frame.size.height;
}

- (void)setBottom:(CGFloat)newBottom
{
    CGFloat delta = newBottom -
        (self.frame.origin.y + self.frame.size.height);
    CGRect newFrame = self.frame;
    newFrame.origin.y += delta;
    self.frame = newFrame;
}

- (CGFloat)right
{
    return self.frame.origin.x + self.frame.size.width;
}

- (void)setRight:(CGFloat)newRight

```

```

{
    CGFloat delta = newRight-
        (self.frame.origin.x + self.frame.size.width);
    CGRect newFrame = self.frame;
    newFrame.origin.x += delta;
    self.frame = newFrame;
}
@end

```

## 4.9 解决方案：获取与坐标变换有关的信息

通过仿射变换，我们可以把视图对象从一个坐标系变换到另一个坐标系，从而改变其几何特征。iOS SDK 完全支持标准的二维仿射变换。开发者可以按照自己的想法和应用程序的需求，对视图进行缩放、平移、旋转、斜切（skew）等操作。

变换机制是定义在 Core Graphics 里面的，Core Graphics 提供了诸如 `CGAffineTransformMakeRotation()` 及 `CGAffineTransformScale()` 等函数。它们可以构建并修改  $3 \times 3$  的变换矩阵。构建好矩阵之后，利用 `UIView` 的 `transform` 属性，可以在 `UIView` 对象上面实施二维仿射变换。

例如，我们可以直接用 `transform` 属性来实现旋转。这样做会移除现有的变换矩阵，并代之以简单的旋转效果。名称中含有 `Make` 一词的函数会创建出新的变换矩阵：

```
theView.transform = CGAffineTransformMakeRotation(radians);
```

另外，也可以在现有的变换效果之上，对视图进行缩放。名字里不带 `Make` 一词的函数会以第一个参数为基础，根据其他参数来修改变换效果，并把修改后的矩阵返回给调用者：

```
CGAffineTransform scaled = CGAffineTransformScale(theView.transform,
    scaleX, scaleY);
theView.transform = scaled;
```

### 4.9.1 获取与变换有关的属性

在操作 `transform` 的时候，iOS 会以仿射矩阵来表示视图的变换效果。这种表示形式并不会直接告诉你视图要缩放多少倍或旋转多少度。于是，我们在解决方案 4-5 里针对 `UIView` 编写 `category`，并在这个简单的 `category` 里面算出具体的缩放倍数和旋转角度。

iOS 用六个字段来表示仿射矩阵，它们分别是 `a`、`b`、`c`、`d`、`tx` 及 `ty`。图 4-4 演示了这些值在标准仿射矩阵里的位置。解决方案 4-5 采用简单的数学算式，根据这些字段算出缩放倍数和旋转角度。从图 4-4 中可以看出，矩阵里的 `tx` 值和 `ty` 值直接对应于仿射变换的平移量。即便读者不擅长线性代数也没有关系，因为就算不理解变换的原理，我们也依然能用它实现出相关的效果来。

在回答“视图现在旋转了多少度？”以及“缩放倍数是多少？”等问题之前，我们一般要通过数学运算，把视图在变换之后的几何特征同其上级视图的坐标系统关联起来。要想实现

这一目标，得先想个办法来指定视图在屏幕上面的位置才行。

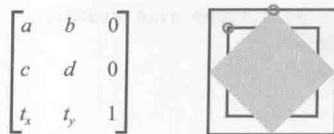


图 4-4 CGAffineTransform 结构体里面保存着执行仿射变换所用的矩阵，该结构体定义了六个字段，用来描述矩阵里的相关元素（如左侧小图所示）。运用了仿射变换之后，视图的 origin（原点）可能就和 frame 的 origin 不重合了（如右侧小图所示）

明确了视图的中心点，我们就可以放心地执行变换了。这个中心点的值可能会变，尤其是执行完缩放变换之后更是如此，然而，无论我们对视图执行了何种变换，该属性的值依然是比较有用的。这个 center 属性总是对应于 frame 的几何中心，它表示几何中心在上级视图坐标系里的位置。

而视图的 frame 则不那么有用。因为旋转之后，视图的 origin 可能就和 frame 的 origin 完全无关了。通过图 4-4 右侧的小图可以看出这一点。实心菱形就是旋转后的视图，它背后有两条外框线，其中范围较小的一条表示旋转之前的 frame，而范围较大的一条则表示旋转之后的 frame。两个红色小圆圈分别表示视图在旋转之前和旋转之后的 origin。

运用了变换效果之后，系统就会把 frame 修改成能够容纳该视图的最小边界框（minimum bounding box）。frame 的新 origin（也就是大外框线的左上角）与视图的新 origin（也就是正上方的那个小圆圈）实际上已经没什么关系了。iOS 没有向开发者提供一些手段，令其可以查询调整之后的点。

解决方案 4-5 实现了几个方法，可以代替开发者来完成这些数学运算。它会算出视图在执行完变换之后其四个角分别位于何处，并通过相关属性把这一结果提供给开发者。这些坐标都是以上级视图为参照物的。图 4-4 右侧的那幅小图其正上方有个红色圆圈，如果我们要把一个新的视图放在这个圆圈上面，那么可将其 center 设为 theView.transformedTopLeft。

这个解决方案里还提供了 originalFrame 方法，无论视图有没有运用变换效果，它都会返回本来的 frame，也就是图 4-4 中范围较小的外框线。笔者是用一种非常“笨”的办法把它算出来的，不过这个办法确实可行。

## 4.9.2 判断两个视图是否相交

应读者请求，笔者又在解决方案 4-5 里面写了一些代码，用以判断两个变换后的视图是否相交。对于没有施加变换效果的视图来说，这些代码依然适用，所以实际上它可以用于任何视图，只不过，在未施加变换效果的情况下，这样做意义不大，因为我们可以直接用 CGRectIntersectsRect() 函数来判断两者的 frame 是否相交。笔者自编的这段代码非常适合用来检测那种外观与底层几何结构不相符的视图，比如图 4-4 里的那种。

## 解决方案 4-5 获取与坐标变换有关的值

```

@implementation UIView (Transform)
- (CGFloat)xScale
{
    CGAffineTransform t = self.transform;
    return sqrt(t.a * t.a + t.c * t.c);
}

- (CGFloat)yScale
{
    CGAffineTransform t = self.transform;
    return sqrt(t.b * t.b + t.d * t.d);
}

- (CGFloat)rotation
{
    CGAffineTransform t = self.transform;
    return atan2f(t.b, t.a);
}

- (CGFloat)tx
{
    CGAffineTransform t = self.transform;
    return t.tx;
}

- (CGFloat)ty
{
    CGAffineTransform t = self.transform;
    return t.ty;
}

// The following three methods move points into and out of the
// transform coordinate system whose origin is at the view center

- (CGPoint)offsetPointToParentCoordinates:(CGPoint)aPoint
{
    return CGPointMake(aPoint.x + self.center.x,
        aPoint.y + self.center.y);
}

- (CGPoint)pointInViewCenterTerms:(CGPoint)aPoint
{
    return CGPointMake(aPoint.x - self.center.x, aPoint.y - self.center.y);
}

- (CGPoint)pointInTransformedView:(CGPoint)aPoint
{
    CGPoint offsetItem = [self pointInViewCenterTerms:aPoint];

```

```

    CGPoint updatedItem = CGPointApplyAffineTransform(
        offsetItem, self.transform);
    CGPoint finalItem =
        [self offsetPointToParentCoordinates:updatedItem];
    return finalItem;
}

// Return the original frame without transform
- (CGRect)originalFrame
{
    CGAffineTransform currentTransform = self.transform;
    self.transform = CGAffineTransformIdentity;
    CGRect originalFrame = self.frame;
    self.transform = currentTransform;

    return originalFrame;
}

// These four methods return the positions of view elements
// with respect to the current transform

- (CGPoint)transformedTopLeft
{
    CGRect frame = self.originalFrame;
    CGPoint point = frame.origin;
    return [self pointInTransformedView:point];
}

- (CGPoint)transformedTopRight
{
    CGRect frame = self.originalFrame;
    CGPoint point = frame.origin;
    point.x += frame.size.width;
    return [self pointInTransformedView:point];
}

- (CGPoint)transformedBottomRight
{
    CGRect frame = self.originalFrame;
    CGPoint point = frame.origin;
    point.x += frame.size.width;
    point.y += frame.size.height;
    return [self pointInTransformedView:point];
}

- (CGPoint)transformedBottomLeft
{
    CGRect frame = self.originalFrame;
    CGPoint point = frame.origin;
    point.y += frame.size.height;

```

```

        return [self pointInTransformedView:point];
    }

    // Determine if two views intersect, with respect to any
    // active transforms

    // After extending a line, determine which side of the half
    // plane defined by that line, a point will appear
    BOOL halfPlane(CGPoint p1, CGPoint p2, CGPoint testPoint)
    {
        CGPoint base = CGPointMake(p2.x - p1.x, p2.y - p1.y);
        CGPoint orthog = CGPointMake(-base.y, base.x);
        return (((orthog.x * (testPoint.x - p1.x)) +
            (orthog.y * (testPoint.y - p1.y))) >= 0);
    }

    // Utility test for testing view points against a proposed line
    BOOL intersectionTest(CGPoint p1, CGPoint p2, UIView *aView)
    {
        BOOL tlTest = halfPlane(p1, p2, aView.transformedTopLeft);
        BOOL trTest = halfPlane(p1, p2, aView.transformedTopRight);
        if (tlTest != trTest) return YES;

        BOOL brTest = halfPlane(p1, p2, aView.transformedBottomRight);
        if (tlTest != brTest) return YES;

        BOOL blTest = halfPlane(p1, p2, aView.transformedBottomLeft);
        if (tlTest != blTest) return YES;

        return NO;
    }

    // Determine whether the view intersects a second view
    // with respect to their transforms
    - (BOOL)intersectsView:(UIView *)aView
    {
        if (!CGRectIntersectsRect(self.frame, aView.frame)) return NO;

        CGPoint A = self.transformedTopLeft;
        CGPoint B = self.transformedTopRight;
        CGPoint C = self.transformedBottomRight;
        CGPoint D = self.transformedBottomLeft;

        if (!intersectionTest(A, B, aView))
        {
            BOOL test = halfPlane(A, B, aView.transformedTopLeft);
            BOOL t1 = halfPlane(A, B, C);
            BOOL t2 = halfPlane(A, B, D);
            if ((t1 != test) && (t2 != test)) return NO;
        }
    }

```

```

if (!intersectionTest(B, C, aView))
{
    BOOL test = halfPlane(B, C, aView.transformedTopLeft);
    BOOL t1 = halfPlane(B, C, A);
    BOOL t2 = halfPlane(B, C, D);
    if ((t1 != test) && (t2 != test)) return NO;
}
if (!intersectionTest(C, D, aView))
{
    BOOL test = halfPlane(C, D, aView.transformedTopLeft);
    BOOL t1 = halfPlane(C, D, A);
    BOOL t2 = halfPlane(C, D, B);
    if ((t1 != test) && (t2 != test)) return NO;
}
if (!intersectionTest(D, A, aView))
{
    BOOL test = halfPlane(D, A, aView.transformedTopLeft);
    BOOL t1 = halfPlane(D, A, B);
    BOOL t2 = halfPlane(D, A, C);
    if ((t1 != test) && (t2 != test)) return NO;
}

A = aView.transformedTopLeft;
B = aView.transformedTopRight;
C = aView.transformedBottomRight;
D = aView.transformedBottomLeft;

if (!intersectionTest(A, B, self))
{
    BOOL test = halfPlane(A, B, self.transformedTopLeft);
    BOOL t1 = halfPlane(A, B, C);
    BOOL t2 = halfPlane(A, B, D);
    if ((t1 != test) && (t2 != test)) return NO;
}
if (!intersectionTest(B, C, self))
{
    BOOL test = halfPlane(B, C, self.transformedTopLeft);
    BOOL t1 = halfPlane(B, C, A);
    BOOL t2 = halfPlane(B, C, D);
    if ((t1 != test) && (t2 != test)) return NO;
}
if (!intersectionTest(C, D, self))
{
    BOOL test = halfPlane(C, D, self.transformedTopLeft);
    BOOL t1 = halfPlane(C, D, A);
    BOOL t2 = halfPlane(C, D, B);
    if ((t1 != test) && (t2 != test)) return NO;
}
if (!intersectionTest(D, A, self))
{
    BOOL test = halfPlane(D, A, self.transformedTopLeft);

```



```

        BOOL t1 = halfPlane(D, A, B);
        BOOL t2 = halfPlane(D, A, C);
        if ((t1 != test) && (t2 != test)) return NO;
    }

    return YES;
}
@end

```

`intersectsView:` 方法采用凸多边形 (convex polygon) 的分离轴算法 (axis separation algorithm) 来判断两者是否相交。我们轮流测试每个视图的每条边, 如果发现其中一个视图的所有点都在该边的一侧, 而另一个视图的所有点都位于该边的另一侧, 那么两视图就不相交。这一测试是基于 `halfPlane` 方法而实现的, 该方法的返回值表示给定的点是在某条边的左侧还是在其右侧。

假如找到了满足此条件的边, 那么 `intersectsView:` 方法就返回 `NO`。如果有条线能够把一个视图里的所有点都划分到它的一侧, 而将另一个视图里的所有点都划分到它的另一侧, 那么这两个视图就不可能相交。

假如 8 次测试都失败了 (针对第一个视图的 4 条边做 4 次测试, 再针对第二个视图的 4 条边做 4 次测试), 那么该方法就认为两个视图确实相交, 它会返回 `YES`。

## 4.10 与显示和交互有关的特征

除了在屏幕上的物理布局之外, `UIView` 类还提供了一些属性, 用于控制视图的样貌以及用户是否可以操作此视图。视图的 `alpha` 值可以在完全不透明与完全透明之间变化。调用 `[myView setAlpha:value]` 语句或是设置 `myView.alpha` 属性, 都可以调整此值。`alpha` 的取值范围是 0.0 到 1.0, 0.0 表示完全透明, 1.0 表示完全不透明。用 `alpha` 值来实现视图的淡入淡出效果是个非常好的办法。如果想令视图不经动画效果就直接消失, 那么请使用 `hidden` 属性。

开发者可以指定任意视图的背景色。例如, 可以通过 `backgroundColor` 属性将视图的背景变红:

```
myView.backgroundColor = [UIColor redColor];
```

这个属性对不同类型的视图所造成的影响是不同的, 具体要看视图里面有没有挡住背景色的子视图。把视图的背景色设为 `[UIColor clearColor]`, 即可创建出透明背景:

```
myView.backgroundColor = [UIColor clearColor];
```

每个视图都提供了 `backgroundColor` 属性, 无论用户到底能不能看见这个视图, 开发者都可以操作该属性。使用鲜亮而且反差较大的背景色, 可以非常有力地呈现出视图的真正内容。对于初学 iOS 开发的读者来说, 可以通过调整视图颜色来判明哪些内容显示在屏幕上, 哪些内容没有显示出来, 也可以明确每个组件的位置。

并非每种颜色都是纯色。我们可以像使用纯色那样通过 `UIColor` 类来使用平铺的纹样 (tiled pattern)。`colorWithPatternImage:` 方法会根据开发者所提供的图像返回 `UIColor` 实例。该方法很适合用来构建渲染视图时所用的材质 (texture)。

`userInteractionEnabled` 属性用来控制用户能否触摸并操作某个特定的视图。对于大多数视图来说, 这个属性的默认值是 `YES`。而对于 `UIImageView` 来说, 它的默认值则是 `NO`, 这把很多初学者都给难住了。他们通常会用 `UIImageView` 来填满整个程序界面, 并在上面放置开关、文本框以及按钮等控件, 然后却发现用户无法操作这些控件。如果某个视图的子视图 (这些子视图包括按钮、开关、选取器以及其他控件) 需要接受触摸, 那么一定要先启用该属性才行。要是发现某个控件似乎无法响应触摸, 那么请检查该控件及其上级控件的 `userInteractionEnabled` 属性值。

如果要在可供用户操作的区域里放置一种仅用于显示的视图, 那么可以禁用这个属性。比方说, 我们要通过透明的全屏视图在屏幕上叠放一个不能接受用户操作的时钟, 那么, 就可以把 `userInteractionEnabled` 标志设为 `NO`, 使得用户无法去操作它。这样一来, 用户的触摸操作就会穿过这个透明的视图而到达它背后的那块区域, 那块区域正是应用程序里可以接受实际操作的区域。把 `userInteractionEnabled` 标志设为 `NO` 只能令该视图本身不接收触摸操作, 而这些触摸操作仍然可以穿过本视图, 到达它背后的其他视图。如果要在屏幕上创建一种提示用户等待的图案, 以阻止其操作应用程序, 那么请把该图案的 `userInteractionEnabled` 标志打开, 使它能够将用户的所有触摸操作都拦截下来, 从而令其无法达到图案背后的主界面。

切换视图的时候, 也可以禁用该标志, 以确保程序在播放切换动画的过程中, 不会因为用户的点击操作而触发相关的动作。对于游戏和解谜类的程序来说, 时机不当的触摸操作可能会令程序出问题。

## 4.11 UIView 的动画效果


在针对 iOS 平台做开发时, `UIView` 的动画效果是个有点俏皮的东西。更新视图的时候, 我们可以用一种运动的形式把视图内容变化的过程表现出来, 从而产生流畅的动画效果, 并提升程序的用户体验。动画效果最明显的优势在于, 无须编写多少代码, 就可将其实现出来。

`UIView` 的动画效果可以令用户清楚地看到视图是怎样从当前状态切换到另一种状态的。实现动画效果时, 我们关注的是视图的内容会有哪些变化, 以及如何把动画效果与这些变化联系起来。可以用动画效果来表现下列变化过程:

- 位置变化——通过移动其中心点而改变视图在屏幕上的位置。
- 尺寸变化——通过修改视图的 `frame` 和 `bounds` 而改变视图的尺寸。
- 可拉伸范围的变化——通过视图的 `contentStretch` 属性来改变视图内容里可供拉伸的区域。
- 透明度变化——改变视图的 `alpha` 值。
- 颜色变化——修改视图的背景色。

■ 旋转角度、缩放倍数、平移量的变化——基本上相当于对视图运用仿射变换。

在 SDK 从 3.x 版演化到 4.x 版的过程中,苹果公司彻底重制了动画效果。从 4.x 版本起,开发者就可以使用 Objective-C 的块来简化动画代码了,这是一种新的编程范式。虽说原来那套实现动画效果的技术依然可以使用,但是新的方式要简单许多,而且苹果公司的开发文档也专门说明:不推荐采用老式的做法来实现动画效果。

 **提示** 苹果公司自己制作的大多数动画的长度都在三分之一或二分之一秒左右。在设计辅助视图(helper view,也就是起支持作用的视图,它们与苹果公司的键盘或警告界面相似)的动画效果时,开发者可以令自己的动画长度与系统自带的动画长度相仿。调用 [UIApplication statusBarOrientationAnimationDuration],即可得知标准的动画时长。

## 用块来构建动画效果

用块来创建动画效果,可以减少所需编写的代码量。下面这段代码只用了一行语句就实现了视图的淡出效果,语句里面嵌套了一个块:

```
[UIView animationWithDuration: 1.0f
    animations:^(contentView.alpha = 0.0f;);];
```

我们可以通过 completion 参数传入一个块,以便在动画结束之后执行清理工作。下面这段代码会对 contentView 运用淡出效果,并且会在动画结束之后,将其从上级视图里移除:

```
[UIView animationWithDuration: 1.0f
    animations:^(contentView.alpha = 0.0f;
    completion:^(BOOL done){[contentView removeFromSuperview];});];
```

如果还想为动画效果指定一些选项,那么可以通过全能的 animateWithDuration:delay:options:animations:completion: 方法来做,该方法也是基于块的,开发者既可以传入掩码形式的动画选项,又可以指定动画的延迟(若想连续播放好几段动画效果,则可以考虑为它们指定不同的 delay 值,这是一种比较简单的实现方式)。

使用与动画有关的常量时,记得要选用名称里带有 Options 一词的新版 UIViewAnimationOptions 常量,而不要使用 UIViewAnimationCurveEaseInOut 等旧版常量,在 4.x 及以后的 iOS 系统里,这些常量无法使相关的方法正常运作。

在个别情况下,我们想把某属性的变化排除在动画效果之外,但有时候却无法确定视图的这个属性会在何处发生变化。比方说,开发者在自己创建的块里面编写了一段动画代码,而这段代码修改了我们不想对其运用动画效果的那个属性,或者 iOS 系统所提供的某些方法会修改这个属性,而该属性恰好又用在了在某个动画块<sup>①</sup>里面。在这些情况下,我们就急需一种手段,能够将该属性从动画效果中排除掉。在 iOS 7 中,苹果公司给 UIView 类提供了

① 也就是传给 animateWithDuration 方法 animations 参数的那个块,其中会包含一些修改视图属性的语句,用以实现动画效果,而传给 completion 参数的那个块则称为完成块,该块中的语句将会在动画效果结束之后执行。——译者注

`performWithoutAnimation:` 方法, 它和 `animationWithDuration` 类似, 也接受块作参数。凡是在这个块里发生变化的属性都会排除在动画效果之外, 即便我们在另一个封装好的动画块里修改了该属性, 它也不会具备动画效果。

## 4.12 解决方案：视图的淡入与淡出

有时候, 我们要在屏幕中现有的视图前方显示一些信息, 这些信息本身只起提示作用, 并没有别的用途。比方说, 我们要显示高分榜、用法说明, 或是提供与情境相关的提示语。解决方案 4-6 用块实现了两段 `UIView` 动画效果, 可以分别令视图逐渐显示出来, 或慢慢消失。该解决方案采用最基本的方式制作动画: 它只是创建了动画块, 并在其中设置了视图的 `alpha` 属性。

解决方案 4-6 通过修改视图的 `alpha` 属性来实现透明度渐变的动画效果

---

```

- (void)fadeOut:(id)sender
{
    self.navigationItem.rightBarButtonItem.enabled = NO;
    [UIView animateWithDuration:1.0f
        animations:^(
            // Here's where the actual fade out takes place
            imageView.alpha = 0.0f;
        )
        completion:^(BOOL done){
            self.navigationItem.rightBarButtonItem.enabled = YES;
            self.navigationItem.rightBarButtonItem =
                BARBUTTON(@"Fade In", @selector(fadeIn:));
        }];
}

- (void)fadeIn:(id)sender
{
    self.navigationItem.rightBarButtonItem.enabled = NO;
    [UIView animateWithDuration:1.0f
        animations:^(
            // Here's where the fade in occurs
            imageView.alpha = 1.0f;
        )
        completion:^(BOOL done){
            self.navigationItem.rightBarButtonItem.enabled = YES;
            self.navigationItem.rightBarButtonItem =
                BARBUTTON(@"Fade Out", @selector(fadeOut:));
        }];
}

```

---

请注意看这段代码是如何处理导航栏右边的 `UIBarButtonItem` 按钮的。当用户点击按钮时, 我们迅速将该按钮禁用, 直到动画结束之后, 再重新启用它。我们给 `animateWithDuration` 方法的 `completion` 参数传入了一个块, 这个块会重新启用按钮, 切换按钮的文本, 并通过 `@selector` 把效果相反的方法设为按钮的回调。这样的话,

用户每次点击该按钮时，imageView 所呈现的动画效果就会与上次相反，也就是说，会轮流播放淡入动画与淡出动画。

### 4.13 解决方案：交换两个视图的前后顺序

通过动画块，我们可以同时实现许多与 UIView 有关的动画效果，而不仅仅局限于一种。动画块里面可以放入很多待改变的属性。解决方案 4-7 把缩放倍数的变化与透明度的变化结合起来，创建出了更为丰富的动画效果。它向动画块里面添加了几条指令，使得程序同时呈现五个方面的动画效果。它会放大其中一个视图，令其淡入，同时又缩小另外一个视图，令其淡出，然后交换这两个视图在 subviews 数组里的位置。

解决方案 4-7 在块里面修改多项视图属性，以制作复杂的动画效果

---

```
@implementation TestBedViewController
- (void)swap:(id)sender
{
    self.navigationItem.rightBarButtonItem.enabled = NO;
    [UIView animateWithDuration:1.0f
    animations:^(
        frontObject.alpha = 0.0f;
        backObject.alpha = 1.0f;
        frontObject.transform = CGAffineTransformMakeScale(0.25f, 0.25f);
        backObject.transform = CGAffineTransformIdentity;
        [self.view exchangeSubviewAtIndex:0
        withSubviewAtIndex:1];
    )
    completion:^(BOOL done){
        self.navigationItem.rightBarButtonItem.enabled = YES;

        // Swap the view references
        UIImageView *tmp = frontObject;
        frontObject = backObject;
        backObject = tmp;
    }];
}
```

---

首次执行动画之前，我们要先做一些准备，也就是要将 backObject 缩小，并令其透明。这样的话，当程序初次运行 swap: 方法时，这个 backObject 视图就可以淡入到屏幕中，并逐渐变大。与解决方案 4-6 相似，我们也在传给 completion 参数的那个块里面重新启用了 UIBarButtonItem 按钮，使得用户可以在动画效果结束之后继续操作该按钮。

### 4.14 解决方案：翻转视图

UIView 类的 transitionFromView 方法也使用块，我们可以通过它实现出比前面更

为丰富的动画效果来。系统提供了许多种切换风格（transition style），其效果均与风格的名称相符。我们可以把视图翻转到背面，也可以像 Maps（地图）程序那样，将其向上或向下卷起来。解决方案 4-8 演示了如何把这些切换效果融入到程序界面中。

下面列出 iOS 7.0 的切换效果。其中有四种翻转效果、两种卷起效果、一种交叉融入（cross dissolve）效果以及一种什么效果都没有的空操作（no-op）：

- UIViewAnimationOptionTransitionNone
- UIViewAnimationOptionTransitionFlipFromLeft
- UIViewAnimationOptionTransitionFlipFromRight
- UIViewAnimationOptionTransitionFlipFromTop
- UIViewAnimationOptionTransitionFlipFromBottom
- UIViewAnimationOptionTransitionCurlUp
- UIViewAnimationOptionTransitionCurlDown
- UIViewAnimationOptionTransitionCrossDissolve

解决方案 4-8 使用 transitionFromView:toView:duration:options:completion: 这个基于块的 API。该方法会令程序从一个视图切换到另一个视图，并把前者从其上级视图里移除，同时将后者作为新视图添加到那个上级视图中。它会在给定的时长（duration）内，依照用户所指定的切换方式及选项标志来完成这段动画效果。解决方案 4-8 使用了从左往右的翻转效果，不过读者也可以根据自己的想法来指定其他切换方式。

#### 解决方案 4-8 通过 transitionFromView 来实现 UIView 之间的切换动画

---

```

- (void)flip:(id)sender
{
    self.navigationItem.rightBarButtonItem.enabled = NO;
    UIView *toView = fromPurple ? maroon : purple;
    UIView *fromView = fromPurple ? purple : maroon;
    [UIView transitionFromView: fromView
                    toView: toView
                duration: 1.0f
        options: UIViewAnimationOptionTransitionFlipFromLeft
    completion: ^(BOOL done){
        self.navigationItem.rightBarButtonItem.enabled = YES;
        fromPurple = !fromPurple;
        CENTER_VIEW(self.view, toView);
    }];
}

```

---


还有个与之相关但更为灵活的方法，叫作 transitionWithView:duration:options:animations:completion:。它的 animations 参数接受一个块，开发者可以由此来全方位地定制视图之间的切换效果。我们可以实现出缩放、翻转以及其他各种复杂的视图切换动画。

如果使用了约束（第5章会讲到），那么必须重新定义它们。把子视图从上级视图里移除之后，上级视图中的所有约束都将无效。

## 4.15 解决方案：采用 Core Animation API 来制作切换效果

除了 UIView 动画之外，iOS 也支持用 Core Animation 来实现动画效果，而这些功能是 Quartz Core 框架的一部分。Core Animation API 为应用程序提供了可以高度定制的动画解决方案。其优点在于：它不仅内置了与解决方案 4-8 相同的切换功能，用以实现视图之间的切换效果，而且还能实现一大批位于本章讨论范围之外的基本动画特效。

---

 **提示** 在发行新版 iOS 的过程中，苹果公司会持续将 Core Animation 里的功能直接移植到 UIKit。iOS 7 为 UIView 添加了“关键帧动画”这一功能，而在此之前，开发者必须通过 Core Animation 才能实现它。

---

有了 Core Animation 之后，我们在制作 UIView 之间的切换动画时就多了一种途径，它与原来那种方式在实现上面有几个小的区别。与 CATransition 协同运作的是 CALayer，而不是 UIView。CALayer 是 Core Animation 中的一种渲染表层（rendering surface），它与 UIView 相关联。用 Core Animation 来制作动画时，我们应该在视图默认的 CALayer（也就是 myView.layer）上面运用效果，而不应该对视图本身来运用。

通过 CATransition 制作切换动画时，不需要像制作 UIView 动画时那样在块里面设置 UIView 的相关属性，而是要创建 CATransition 对象，设置其属性，然后把设置好属性值的这个对象添加到 CALayer 之中：

```

CATransition *animation = [CATransition animation];
animation.delegate = self;
animation.duration = 1.0f;
animation.type = kCATransitionMoveIn;
animation.subtype = kCATransitionFromTop;

```

```
// Perform some kind of view exchange or removal here
```

```
[myView.layer addAnimation:animation forKey:@"move in"];
```

CATransition 同时采用 type 及 subtype 来定义动画效果。type（类型）表示如何切换，而 subtype（子类型）则表示从哪个方向开始切换。把两者合起来运用到 CALayer 上面，也就相当于指明了视图的切换效果。

Core Animation 里面的切换类型与上一个解决方案中的那几种 UIViewAnimation-Transition 选项是不同的。Cocoa Touch 提供了四种类型的 Core Animation 切换效果，解

决方案 4-9 演示了它们。这四种类型是：交叉淡入淡出（cross-fade）、推动（push，一个视图将另一个视图推离屏幕）、揭示（reveal，一个视图从另一个视图上滑走，露出后者）、覆盖（cover，一个视图滑入另一个视图，并将后者覆盖）。后面三种类型都需要通过 subtype 来指定动画的切换方向。

#### 解决方案 4-9 用 Core Animation 实现切换动画

```
- (void)animate:(id)sender
{
    // Set up the animation
    CATransition *animation = [CATransition animation];
    animation.delegate = self;
    animation.duration = 1.0f;

    switch ([[UISegmentedControl *)self.navigationItem.titleView
            selectedSegmentIndex])
    {
        case 0:
            animation.type = kCATransitionFade;
            break;
        case 1:
            animation.type = kCATransitionMoveIn;
            break;
        case 2:
            animation.type = kCATransitionPush;
            break;
        case 3:
            animation.type = kCATransitionReveal;
            break;
        default:
            break;
    }
    animation.subtype = kCATransitionFromLeft;

    // Perform the animation
    [self.view exchangeSubviewAtIndex:0 withSubviewAtIndex:1];
    [self.view.layer addAnimation:animation forKey:@"animation"];
}
```

由于 Core Animation 是 Quartz Core 框架的一部分，所以在使用这些特性之前，需要先在代码中使用 `@import QuartzCore` 指令。



**提示** 苹果公司的 Core Animation 是通过 Objective-C 的类来构建其二维和三维动画特性的。这些类为 iOS 和 Mac 应用程序提供了图形渲染及动画功能。使用 Core Animation 的好处是，我们不用去直接操作与 OpenGL 等技术相关的许多底层开发细节，就能够以非常简单的方式来操作分层排布的 CALayer。



## 4.16 解决方案：使视图在出现之后回弹

苹果公司经常会连续使用两个动画块，并在第一段动画结束之后播放第二段，以实现回弹（bounce）效果。比方说，在放大某个视图的时候，它会先把视图放大到略微超过期望大小的尺寸，然后再用第二段动画将这个稍微有点大的视图缩小到最终尺寸。添加了回弹效果之后，动画就显得更生动、更真实了。

如果要连续播放两段动画，那么需要保证两段动画的播放时段不会互相重叠。最简单的办法是在传给 completion 参数的那个块里面再嵌套一个动画块，把与第二段动画相关的代码放在这个块里。解决方案 4-10 采用这个办法先将视图放大到略微超过期望大小的尺寸，然后再将其缩小到最终尺寸。

解决方案 4-10 实现视图的回弹效果

---

```
typedef void (^AnimationBlock)(void);
typedef void (^CompletionBlock)(BOOL finished);

- (void)bounce
{
    // Prepare for animation
    self.navigationItem.rightBarButtonItem.enabled = NO;
    bounceView.transform = CGAffineTransformMakeScale(0.0001f, 0.0001f);
    bounceView.center = RECTCENTER(self.view.bounds);

    // Define the three stages of the animation in forward order
    AnimationBlock makeSmall = ^(void){
        bounceView.transform = CGAffineTransformMakeScale(0.01f, 0.01f);};
    AnimationBlock makeLarge = ^(void){
        bounceView.transform = CGAffineTransformMakeScale(1.15f, 1.15f);};
    AnimationBlock restoreToOriginal = ^(void){
        bounceView.transform = CGAffineTransformIdentity;};

    // Create the three completion links in reverse order
    CompletionBlock reenable = ^(BOOL finished){
        self.navigationItem.rightBarButtonItem.enabled = YES;};
    CompletionBlock shrinkBack = ^(BOOL finished){
        [UIView animateWithDuration:0.3f
            animations:restoreToOriginal completion:reenable];};
    CompletionBlock bounceLarge = ^(BOOL finished){
        [NSThread sleepForTimeInterval:0.5f]; // wee pause
        [UIView animateWithDuration:0.3f
            animations:makeLarge completion:shrinkBack];};

    // Start the animation
    [UIView animateWithDuration: 0.1f
        animations:makeSmall completion:bounceLarge];
}
```

---

这条解决方案定义了两个 typedef，使得我们在声明传给 animations 参数及 completion 参数的块时，能够把代码写得简单一些。请注意，笔者是依照动画播放顺序来定义表示各个阶段的那三个 AnimationBlock 的。第一个块会将视图缩到很小，第二个块会把视图放大到稍稍超过其正常大小的尺寸，而第三个块则会将视图尺寸复原。

但是在指定 CompletionBlock 时，顺序则相反。由于 CompletionBlock 所表示的动画必须在前一个动画结束之后才能播放，所以必须按照相反的顺序来创建它们。首先定义最后要执行的那个块，然后依逆序来定义其他块。在解决方案 4-10 之中，bounceLarge 要依赖于 shrinkBack，而 shrinkBack 又要依赖于 reenable。这种反向定义的代码虽然写起来有点麻烦，但它肯定要比把全部代码都逐层嵌套到块里面要好。

本解决方案的范例项目中提供了一个名为 AnimationHelper 的辅助类，它会把解决方案 4-10 所要实现的功能以稍微易用一些的形式封装起来。从解决方案 4-10 的代码中可以看到，必须按照逆序来定义 CompletionBlock 才能使先播放的那段动画能够引用后播放的那段动画，而这种写法很容易令代码变得杂乱。

辅助类会构建出完整的块序列，并返回嵌有 CompletionBlock 的块，开发者可以把返回的这个块经由 animations 参数直接传给 animateWithDuration 方法来执行，以实现类似解决方案 4-10 的效果。

## 4.17 解决方案：关键帧动画

通过嵌套 AnimationBlock 及 CompletionBlock，可以实现出相当丰富的动画效果，不过，代码的复杂程度也会迅速升高。在 iOS 7 里，苹果公司为 UIKit 引入了关键帧动画，这是一种强大的动画功能，可以满足高端需求。它能够极大地简化创建动画所需的代码，使我们只需编写少量代码即可实现出像解决方案 4-10 那样复杂的回弹效果。以前，我们必须深入 Core Animation 才能使用关键帧动画，而现在，这款强大的动画制作工具已经直接集成到 UIView 里面了。

制作传统的关键帧动画时，开发者提供动画序列中某些重要的帧，并在动画中设置相应的时间戳，而系统则会把相邻两个关键帧之间的其他帧补全，以便在各个关键帧之间渲染出平滑的动画效果。最简单的关键帧动画只提供起始帧和结束帧，并把其他帧交由系统来补充。

在 UIView 上面创建关键帧动画时，我们要给 animateKeyframesWithDuration:delay:options:animations:completion: 方法的 animations 参数传入一个块。在这个块中，要设定每一个重要的参考帧 (important reference frame，具体到本方法来说，就是设定想要以动画效果展示其变动过程的那些 UIView 属性)，同时还要用 addKeyframeWithRelativeStartTime:relativeDuration:animations: 来指定关键帧的起始时间及持续时长。系统会根据块中的代码，在适当的时间点播放给定时长的关键帧动画序列。

制作关键帧动画与用其他方式来制作动画之间有个重要的区别，就是关键帧动画的起始时间与时长都必须在 0.0 到 1.0 这个范围内取值，这些值对应于整个动画进度的百分比。0.0 表示整个动画流程的开端，而 1.0 则表示整个动画流程的末端。对于总长为两秒的动画

来说,如果某个关键帧的起始时间是 0.5,那么它会出现现在动画播放了一秒钟之后的那个位置上。

解决方案 4-11 采用关键帧动画实现出了与解决方案 4-10 相同的动画效果。这次不需要使用辅助类,而且代码非常好写,理解和管理起来也很容易。

解决方案 4-11 关键帧动画

---

```

- (void)bounce
{
    // Prepare for animation
    self.navigationItem.rightBarButtonItem.enabled = NO;
    bounceView.transform = CGAffineTransformMakeScale(0.0001f, 0.0001f);
    bounceView.center = RECTCENTER(self.view.bounds);

    // Begin the key frame animation
    [UIView animateKeyframesWithDuration:0.6
        delay:0.0
        options:UIViewKeyframeAnimationOptionCalculationModeCubic
        animations:^(
            // Implied first key frame - current view (tiny)
            // Second key frame - make view big
            [UIView addKeyframeWithRelativeStartTime:0.0
                relativeDuration:0.5
                animations:^(
                    bounceView.transform =
                        CGAffineTransformMakeScale(1.15f, 1.15f);
                )];

            // Third key frame - shrink to normal
            [UIView addKeyframeWithRelativeStartTime:0.5
                relativeDuration:0.5
                animations:^(
                    bounceView.transform =
                        CGAffineTransformIdentity;
                )];
        )
        completion:^(BOOL finished) {
            [self enable:YES];
        }
    ];
}

```

---

## 4.18 解决方案: UIImageView 的动画效果

UIImageView 类不仅可以显示静态的图片,而且也支持内置的动画序列。把每一格动画所用的图像放在数组里并加载到 UIImageView 之后,就可以令 UIImageView 实例展示动画效果了。解决方案 4-12 演示了具体做法。

### 解决方案 4-12 制作 UIImageView 自身的动画效果

```
NSMutableArray *butterflies = [NSMutableArray array];

// Load the butterfly images
for (int i = 1; i <= 17; i++)
    [butterflies addObject:[UIImage imageWithContentsOfFile:
        [[NSBundle mainBundle]
        pathForResource:[NSString stringWithFormat:@"bf_%d", i]
        ofType:@"png"]]];

// Create the view
UIImageView *butterflyView = [[UIImageView alloc]
    initWithFrame:CGRectMake(40.0f, 300.0f, 100.0f, 51.0f)];

// Set the animation cells and duration
butterflyView.animationImages = butterflies;
butterflyView.animationDuration = 0.75f;
[butterflyView startAnimating];

// Add the view to the parent
[self.view addSubview:butterflyView];
```

首先，创建数组，从文件里把每张图片加载到数组里，并把该数组赋给 UIImageView 实例的 animationImages 属性。然后把展示数组中全部图片所用的总时长设置成 animationDuration 属性的值。最后，发送 startAnimating 消息以启动动画效果。（还有个与之配套的 stopAnimating 方法可以停止动画效果。）

将本身正在播放动画的 UIImageView 添加到界面之后，我们可以把它放在那里不动，也可以像对待其他 UIView 实例那样，为其运用别的动画效果。

## 4.19 小结

UIView 是一种用户可以看到并操作的界面组件。通过阅读本章，大家可以发现，即使是最简单的 UIView，也都具备了相当灵活而强大的功能。我们学会了如何用 UIView 来构建屏幕上的元件，如何根据标签或 nametag（控件名称）来获取 UIView，以及如何制作醒目的动画效果。在继续学习下一章之前，读者可以先回顾下面几个问题：

- 处理多个视图的时候，一定要有层级这个概念。通过与视图层级相关的各种方式来管理程序中的视图，并根据适当的情境向用户展示出对应的视图布局。
- UIKit 是以 center 属性为中心的，而 Core Graphics 则不是。作为开发者，我们不应该令自己的代码受制于此，而是应该通过一些函数在这两套结构之间转换，尤其是在应对一些没有施加变换效果的简单视图时，更应如此。
- 多使用标签，无论是数值形式的标签，还是自定义的 nametag 都行。有了标签之后，就可以用一种类似于符号表（symbol table）的办法，在程序代码中直接访问相关的

UIView 了。这些标签没有什么不好的地方，在开发工作中，它们是一种有用的编程技巧。

- 灵活掌控坐标变换 (transform)。它们就是一些数学运算。实施了坐标变换之后，我们应该依然有办法获取视图的各项信息才对，比方说，应该能够查到当前的旋转角度、缩放倍数以及视图四个角的位置等。在许多 iOS 开发领域，坐标变换都能起到很大的作用。本章所提供的几条解决方案在该功能的基础之上添加了一些特性，使开发者能够掌控坐标变换功能，以便在需要用到某些信息时可以立刻查到它们。
- 块太有用了。它们可以简化开发工作、简化代码的编写过程以及动画的制作过程。
- 尽可能多用一些动画效果。动画效果并不一直都是杂乱而拙劣的。iOS SDK 支持非常强大的动画效果，令开发者能够在用户执行了某项操作之后，以平滑的动画表示出视图的切换过程。iOS 应用程序以精巧而流畅的切换动画著称。
- 本章绝大部分内容都是直接来调整视图的层级结构和摆放位置的，这样做的前提是开发者自己控制视图的布局。第 5 章将要讲述 Auto Layout，这是一套声明式的约束系统，用它来管理视图的布局要比手工管理更方便、更强大。

## Chapter 5 第5章

## 视图的约束系统

Auto Layout (自动布局) 彻底改变了开发者创建用户界面 (UI) 的方式。它提供了一套强大而灵活的系统, 用于描述视图与其内容之间的关系以及它们同上级视图之间的关系。与手动管理框架的几何属性及 Struts and Struts<sup>⊖</sup>相比, 这种新技术使开发者可以更好地控制视图布局, 并对其进行全方位定制。苹果公司发布了一些新设备, 其屏幕大小与原来的设备不同, 而且又引入了一批新的 API, 使得 UI 在程序运行的时候可以动态地变化, 它以这些方式来促使开发者采用这套新技术管理视图布局。在 Auto Layout 问世之前, 要想以手工方式处理刚说的这些情况是十分困难的, 有时甚至根本无法做到。

有了 Auto Layout 之后, 视图的排布就直观多了, 我们只需描述视图之间的关系即可, iOS 会负责把它们摆放到适当的位置上。此外, 如果定义了视图之间的关系以及与动态布局有关的属性, 而不是以硬代码来编写原点及尺寸的话, 那么视图对象就会根据屏幕方向、设备屏幕的宽高比等因素自动做出正确的反应, 运用了 Dynamic Type 技术之后, 甚至还能根据不同的界面语言以及用户所设定的文本大小来自动调整标签 (label) 的尺寸。

本章介绍如何用代码来控制 Auto Layout 约束。约束定义了视图之间的关系以及视图同其视窗、同上级视图的关系。iOS 系统会根据约束来定义每个视图实际的框架属性。



**提示** Auto Layout 是个深奥而广泛的话题, 用一整本书来写它都不为过。我们在这里只能扼要地谈一谈。如果想通过范例全面了解并深入分析 Auto Layout 技术, 包括如何在 Interface Builder (简称 IB) 里使用 Auto Layout, 那么请查阅 Erica Sadun 所著的最新版《iOS Auto Layout Demystified》, 该书也由 Addison-Wesley 出版。

⊖ IB 界面中的一套布局机制, 可用来指定视图的自动缩放能力。——译者注

## 5.1 什么是约束

约束 (constraint) 就是一系列描述 iOS 程序视图布局的规则。它们限定了视图之间的关系,也限定了视图的布局形式。使用约束时,我们可以说“这些视图在水平方向上必须对齐”,或是“此视图必须根据另一个视图来调整自身高度,以便与之相符”。约束向开发者提供了一套布局语言,使得可以向视图里添加约束,并以此来描述各视图的空间关系。

iOS 负责通过一套约束满足系统来实现这些布局需求。规则必须有意义。不能说某视图既位于另一个视图左侧,又位于它的右侧。使用约束时的一个难点就是如何保证规则之间总是协调一致的。假如规则之间有冲突,那么开发者将收到明确通知。Xcode 会提供详尽的记录信息来解释具体的错误情况。

另外一个难点在于规则要指定得足够具体才行。如果对界面所施加的约束过少,那么可能会产生不符合预期的布局,因为有很多种布局方案可供选用。我们可能会要求某视图处在另一个视图右侧,但是假如不指定垂直方向上的规则,那么系统就可能会把右侧这个视图排在屏幕顶端,而把左侧那个视图排在屏幕底端。

有了约束,就可以制作出不依赖于分辨率的应用程序了。对于一款针对 4 英寸屏幕的 iPhone 而制作的应用程序,只要是基于约束的,那么不用修改任何代码,就可以直接运行在将来发售的 5 英寸 iPhone 上。

对于需要进行本地化的程序来说,不要为每种界面语言都创建一个 XIB,而是应该采用约束。基于约束的 XIB 可以适应多种界面语言。

开发者既可以在 IB 中以可视化的形式来指定约束,也可以在程序源码中以编程的形式来制作约束。Xcode 5 的 IB 对 Auto Layout 功能做了很多改进。用 IB 来调整约束及视图布局简单易行,本章则专门讲解如何用代码来操作约束。我们提供以代码为中心的范例,通过这些范例,读者可以用 Objective-C 语言为视图创建出常见的约束。

## 5.2 约束系统所用的属性

约束所用的词汇非常有限,就是一些与几何特征有关的属性及关系。属性是约束系统中的“名词”,用来描述视图对齐矩形 (alignment rectangle) 里的位置。稍后我们会详细解释对齐矩形这一概念,而现在大家可以把它看作与视图的框架紧密相关的一个东西。关系 (relation) 是系统里的“动词”,用于在属性之间进行比较。

属性名词描述的是物理特征。约束系统提供了下面这几个“名词”,用来描述视图的相关属性:

- **left、right、top 及 bottom**——视图对齐矩形的左、右、上、下边界。它们分别对应于视图的最小 X 值、最大 X 值、最小 Y 值以及最大 Y 值。
- **leading 及 trailing**——视图对齐矩形的前边沿及后边沿。在从左至右的书写系统里 (比如英文),前边沿就是“左”,后边沿就是“右”。而在阿拉伯文、希伯来文等从右至左的语言环境下,则相反:前边沿是“右”,后边沿是“左”。

如果要令应用程序支持多种界面语言，那么应该使用 `leading` 和 `trailing` 来代替 `left` 和 `right`。这样的话，在阿拉伯文和希伯来文等语言环境下，应用程序的界面就能在左右方向上自动反转了。

- **width 与 height**——视图对齐矩形的宽度和高度。
- **centerX 与 centerY**——视图对齐矩形的中心点在  $x$  轴和  $y$  轴的坐标。
- **baseline**——对齐矩形的基线，通常比 `bottom` 属性小一些，而且两者之间的偏移量通常是某个定值。

关系动词用于比较属性值之间的关系。在约束系统的数学运算中只有三种关系：我们可以限定两个属性必须相等，也可以限定其下界或上界。可以使用下面三种布局关系：

- **Less-than inequality**（小于或等于）——`NSLayoutRelationLessThanOrEqualTo`
- **Equality**（等于）——`NSLayoutRelationEqual`
- **Greater-than inequality**（大于或等于）——`NSLayoutRelationGreaterThanEqual`

你可能觉得上面这三种关系不会产生太多的布局组合。但实际上，这三种关系可以把排布用户界面时所需的各种布局情况全都涵盖进来。通过这三种关系，我们可以给属性指定具体的值，也可以指定其上限或下限。

## 约束系统所用的数学算式

对于所有的约束规则，无论它是如何创建出来的，其本质都可归结为下列形式的等式或不等式：

$$y \text{ 关系 } m * x + b$$

如果读者掌握了一些数学知识，那么可能会熟悉另一个与上述算式很相似的表述形式，式子中的  $R$  就表示  $y$  与右侧运算值之间的关系：

$$y R m * x + b$$

$y$  与  $x$  都是上面介绍过的那些视图属性，比方说 `width`、`centerY` 或 `top`。而  $m$  则是个表示缩放比例的常数， $b$  是表示偏移量的常数。例如，我们可以规定，“B 视图的左界应该位于 A 视图右界的右方 15 点处”。那么关系式就可以写成：

$$B \text{ 视图的左界} = A \text{ 视图的右界} + 15$$

这是个“等于”关系，偏移量常数 ( $b$ ) 是 15，缩放倍数或放大倍数 (`multiplier`,  $m$ ) 是 1。笔者故意没有把上面这个式子写得和代码一样，因为我们并不需要以 Objective-C 代码的形式来直接声明约束规则。

约束规则未必都是严格的等式，也可以使用“不等”关系。比方说，我们可以规定，“B 视图的左界应该距离 A 视图右界的右方至少 15 点”。这可以写成：

$$B \text{ 视图的左界} \geq A \text{ 视图的右界} + 15$$

我们可以通过偏移量 ( $b$ ) 在控件之间安插固定的间隔，并通过放大倍数 ( $m$ ) 来进行缩放，这对于网格形式的布局来说尤为有用，因为开发者可以直接把某个视图的高度设为另一个视图的若干倍，而不用在两个视图之间添加固定的间隔距离。



### 5.3 约束系统的运作规律

你可能认为约束系统所用的数学算式非常严格，实际上它们只是个参考。iOS 会找到最符合约束的一种布局方案，有的时候，这种方案不止一套。下面给出约束系统的一些基本特征：

- 约束规则描述的是关系，而不一定是视图在某个方向上的属性。未必非要在知道右边界的情况下才能算出左边界。
- 每条约束规则都有其优先级。优先级的取值范围是从 0 到 1000。Auto Layout 系统用优先级来排列各条约束的顺序。它总是先设法满足优先级较高的约束规则，然后再去满足优先级较低的规则。优先级为 99 的规则总是排在优先级为 100 的规则后面。在排布视图位置的时候，系统会遍历开发者所添加的全部规则，并试着找出一种符合所有约束规则的布局方案。优先级用来判定各条规则的影响力。假如刚才说的那两条约束规则互相冲突，那么系统会考虑优先级为 100 的那条规则，而放弃优先级为 99 的那条。

优先级最高的约束规则是必须要满足的（required），它的类型是 `UILayoutPriorityRequired`，其值为 1000，而这种类型也是系统默认的约束类型。优先级为 1000 的约束必须完全满足才行，比方说，我们可以表达出“此按钮必须是这种尺寸”等意图。假如某条约束规则的优先级不是 1000，那么它在整个布局系统里的影响力就变弱了。

但即便是必须满足的约束，也有可能在发生冲突的情况下遭到覆盖。如果我们没有权衡好各条规则之间的关系，那么很有可能令本来应该是  $100 \times 100$  的视图变成  $102 \times 107$ 。表 5-1 详细列出了几种约束规则的优先级和对应的值。

表 5-1 表示优先级的各种常量

种 类	值
<code>UILayoutPriorityRequired</code> (默认)	1 000
<code>UILayoutPriorityDefaultHigh</code>	750
<code>UILayoutPriorityDefaultLow</code>	250
<code>UILayoutPriorityFittingSizeLevel</code>	50

- 除了优先级这一因素之外，各条约束规则之间并没有其他自然的顺序。系统会同时考虑优先级相同的一系列约束。如果想令某条约束占先，那么可以提升其优先级。
- 有些约束规则只需大致满足即可。我们可以用一些可选的（optional）约束规则来试着优化一下布局效果。比方说，“2 号视图的顶边应该和 1 号视图的底边处在同一位置”。约束系统会试着令两个视图靠近，并尽量缩小其距离。假如有其他约束规则使得这两个视图无法紧贴，那么系统会尽可能地令它们靠拢，以缩小这两个属性之间的差距。
- 约束规则之间可以有循环。所涉及的控件只要都满足规则就行了，并不需要明确每个视图具体对应于规则中的哪个物件。不要跟交叉引用较劲。这套声明式的系统可以接受循环引用，所以开发者无须担心无限循环问题。
- 可以用动画来表现约束规则的变化过程。从一套约束规则切换到另一套的时候，可以用 `UIView` 的动画功能来表现这个过程。在动画块中，我们可以于改变约束规则之后调用 `layoutIfNeeded`，这样的话，视图就会以动画效果来展示由上一套约束规则切换到新规则的过程。

- 在约束规则里可以引用同一体系里的其他视图。对于某个视图来说，可以令其中一个子视图的中心点与另外一个完全不同的视图的中心点相对齐，只要两者之间有公共的祖先视图即可。比方说，我们创建了一种复杂的文本输入视图，其中还嵌套有 UIImageView 控件，那么现在可以令它最右侧那个按钮的 right 属性与按钮下方 UIImageView 控件的 right 属性相对齐。但是下一段将会讲述一种例外情况。
- 约束规则不应该跨越边界系统 (bound system)。在指定对齐方式的时候，不要跨越 UIScrollView、UICollectionView 及 UITableView 的边界。如果某种视图有它自己的边界系统，那么就不要从一个边界系统跳到另一个完全不同的边界系统里面去。虽说这么做可能不会使程序崩溃，但毕竟不是个好办法，而且 Auto Layout 对这种做法的支持度也不好。
- Auto Layout 与 transform (坐标变换) 之间可能配合得不够好。同时使用 transform 及 Auto Layout 时要多加小心，尤其在涉及旋转的情况下更是如此。
- Auto Layout 不能和 iOS7 所提供的 UIKit Dynamics 协同运作。受动态行为所影响的那些视图依然可以施加 Auto Layout，但如果某个视图处在 UIDynamicAnimator 的管理之下，那么就不能同时用 Auto Layout 去调整它的排布方式了。
- Auto Layout 可以和 iOS 7 的运动效果 (motion effect) 一起工作。由 UIMotionEffect 实例所产生的视觉变化只会影响视图的 CALayer，而不会干扰底层的布局。
- 约束规则在运行期可能会出错。如果系统无法解析开发者所设定的约束规则 (参看本章末尾的范例)，或是规则之间有冲突，那么运行期系统就会丢弃一些规则，以便尽量把视图排布出来。这种做法很不优雅，而且经常会产生与程序需求不符的布局。Auto Layout 会把详尽的信息发送到 Xcode 控制台，告诉开发者哪里出了错。我们可以根据这些错误报告来修正约束规则，使各条规则之间和谐一致。
- 格式错误的约束规则可能会阻断应用程序的执行。规则之间若有了冲突，则只会产生错误消息，而应用程序还是可以继续运行的，但是，如果规则的格式写错了，那么执行了某些调用之后，程序就会因为未处理的异常而崩溃。比方说，我们可能会把类似于 @"V[view1]-|" 的格式字符串传给相关方法，以根据字符串所描述的规则来创建约束，但此时就会遭遇运行期错误 (因为字母 V 的后面漏了个冒号)：

```
Terminating app due to uncaught exception 'NSInvalidArgumentException', reason:
'Unable to parse constraint format'
```

这种错误在编译的时候无法检测到，所以我们必须仔细检查格式字符串。在 IB 里面设计约束规则时，就不用担心由于拼写错误而导致的程序崩溃问题了。

- 每条约束规则必须至少引用一个视图。如果创建了一条没有引用任何视图的约束规则，那么 Xcode 在编译代码时不会产生警告，但程序却会在运行的时候抛出异常。
- 避免无效的属性能搭配。把某个视图的左边界和另一个视图的高度相匹配是不合法的。无效的属性能组合会令程序在运行的时候抛出异常。尤其不应该把与尺寸有关的属性与与边界有关的属性混起来用。一般来说，我们都能够把有问题的属性能组合检查出来，因为那些属性能搭配在一起是没有意义的。



使用约束可能会导致应用程序在运行的时候因为一些稀奇古怪的原因而崩溃。在开发和部署基于约束的应用程序时一定要特别小心，我们应该构建一些测试来尽量确保相关操作在各种情况下都正常运作。但愿有人能够写出一款约束规则验证器（constraint validator），它至少要能够把 @"H:[myView" 等简单的拼写错误找出来，并告诉我们这条规则少了右方括号。

## 5.4 约束规则与框架属性

Auto Layout 约束系统所使用的底层几何特征与第4章中 frame 所使用的特征是相同的。Auto Layout 的强大之处基本上在于：它排布 UI 的同时，还可以处理内容多变的视图。UIView 有两个属性，分别是固有内容的尺寸（intrinsicContentSize）以及对齐矩形（alignmentRectForFrame:），该矩形的范围可以超越传统的框架，以便使 Auto Layout 系统能够适当地处理视图之间的关系。

### 5.4.1 固有内容的尺寸

在 Auto Layout 系统中，视图的内容与视图的约束规则一样，都对布局起着重要作用。内容是通过每个视图的 intrinsicContentSize 属性表达出来的。该方法描述了在既不挤压也不裁切的前提下完全容纳视图内容所需的最小空间，其含义可以根据每个视图所要表示的内容而推断出来。

例如，对于 UIImageView 来说，这就相当于其中的图像所具备的尺寸。图像如果比较大，那么 intrinsicContentSize 也就会大一些。而对于标签控件来说，则取决于字体和文本量（text amount）。标签控件的 intrinsicContentSize 会随着文本长短与所选字体而变化。

有了 intrinsicContentSize 之后，Auto Layout 系统就可以把视图的框架属性与其内容较好地匹配起来。我们通常需要在每个轴上设置两种属性才能避免约束规则过少或排版方式有歧义等情况。为视图指定了 intrinsicContentSize 之后，就相当于已经有了其中一种属性。现在我们可以把某个基于文本的控件或 UIImageView 控件放在上级视图的中心，这样的排版方案只有一种，所以不会产生歧义。把 intrinsicContentSize 与 location（位置）结合起来，就可以完整地确定视图的排布方式了。实际上，Auto Layout 系统在排版时会把 intrinsicContentSize 解析成一条约束规则。

改变视图的内容之后，我们可以调用 invalidateIntrinsicContentSize 方法，以此告知 Auto Layout 系统下次排版时应该重新计算 intrinsicContentSize。

#### Compression Resistance 与 Content Hugging

Auto Layout 系统在判定 intrinsicContentSize 的大小时会受到两个属性的影响。正如其名称所示，Compression Resistance（抗挤压性）表示视图保护其内容不受压缩的能力。抗挤压性越高的视图，越不容易收缩。它要竭力避免内容遭到裁切（clip）。而 content hugging（内容凝聚度）则表示一种优先级，它指明视图是否不愿意在其核

心内容之外添加边距, 或视图是否不愿意拉伸其核心内容 (比方说, `contentMode` 属性设为“缩放”<sup>①</sup>的 `UIImageView` 控件是否不愿意拉伸其图像)。开发者可以通过 `setContentCompressionResistancePriority:forAxis:` 及 `setContentHuggingPriority:forAxis:` 方法按坐标轴来设置视图的这两种优先级。

## 5.4.2 对齐矩形

约束系统所采用的布局办法与手动排版时所用的框架不同。框架描述的是视图摆放的位置以及视图的大小, 而约束系统在排布视图时, 则使用一个与之相关的几何概念, 即对齐矩形 (alignment rectangle)。

在创建复杂的视图时, 开发者可能会引入一些视觉上的装饰效果, 例如阴影、边缘高亮 (exterior highlight)、镜像 (reflection) 以及雕刻线条 (engraving line) 等。这些特性通常是以子视图或子层的形式添加到视图里的。在添加这些物件的过程中, 视图的框架及其完整范围也会随之不断变大。

与 `frame` 不同, 视图的对齐矩形仅局限于核心视觉元件 (core visual element)。把新的物件添加到主视图并不会影响它的大小。请看图 5-1 左侧的示意图。这个示意图带有阴影效果和徽章图样, 阴影效果位于主视图的后方, 而徽章则在视图右上角。排布这个视图的时候, `Auto Layout` 只会把核心元件同其他视图相对齐。

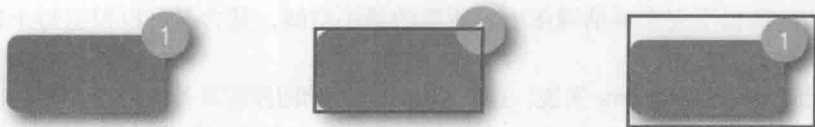


图 5-1 视图的对齐矩形 (中间示意图里的黑色框线) 只涵盖待对齐的核心视觉元件, 而不考虑附加的修饰效果

图 5-1 中间那张示意图描绘了视图的对齐矩形。该矩形把阴影及徽章等装饰物都排除在外。我们只希望 `Auto Layout` 系统参照对齐矩形里面的这部分内容来排布视图。

把这个对齐矩形和图 5-1 右侧示意图里的那个矩形比较一下。那个矩形把所有视觉装饰物都包含在内了, 于是, 它超出了 `Auto Layout` 系统在对齐这个视图时所应参考的边界。那个矩形把视图的所有视觉元件都涵盖进来了。假如按照此矩形来排版, 那么这些装饰物可能会干扰排列视图位置时所参照的一些特征 (例如中心点、底边、右边界等)。

`Auto Layout` 系统是根据对齐矩形来排布视图位置的, 而不是根据框架属性, 这样可以确保排版时所参照的关键信息 (例如视图的边界及中心点等) 准确无误。

### 声明对齐矩形

在构建带有装饰物的视图时 (例如内置有阴影效果的 `UIImageView`), 开发者应该把详

① 指代 `UIViewContentModeScaleToFill`、`UIViewContentModeScaleAspectFit` 及 `UIViewContentModeScaleAspectFill` 这三种内容显示模式。——译者注

细的几何特征告诉 Auto Layout 系统。如果自己的视图使用了阴影或镜像效果等装饰物，那么可以在视图类里实现 `alignmentRectForFrame:` 方法，并返回精确的对齐矩形。

该方法接受一个参数，也就是框架。该参数表示视图所居的目标框架（destination frame）。请看图 5-1 右侧示意图中的矩形。那个矩形所代表的框架就涵盖了整个视图，这其中也包括附着在视图上的装饰物。开发者应该根据那个目标框架以及视图中所嵌入的元件来计算出精确的对齐矩形。

此方法所返回的 `CGRect` 值指明了视图的核心视觉内容所处的那个矩形，如图 5-1 中间的矩形所示。它通常就是主视图对象本身的框架，而不包括作为子视图或 `CALayer` 的子层添加到视图中的那些装饰物。

如果要对视图执行坐标变换，那么请记得一并实现 `frameForAlignmentRect:` 方法。该方法描述的是反向关系，也就是根据传入的对齐矩形（比方说，图 5-1 中间那张示意图里的矩形）来算出可以完整包含所有装饰物的框架（比方说，图 5-1 右侧示意图里的那个矩形）。我们会以传入的那个对齐矩形为基础扩大其边界，使之能够把视图里的所有装饰物都涵盖进来。

## 5.5 创建约束规则

通过 `NSLayoutConstraint` 类，开发者可以用两种方式来创建约束规则。可以用一个相当长的方法调用语句来指明视图的某个属性与其他属性之间的关系，并描述这些属性之间的联系，也可以用一种写起来非常短小的格式化语言（formatting language）来指定视图在水平方向与垂直方向上的排布形式。

本节将会演示这两种方式，使读者明白它们的写法及用法。请记住：无论怎样构建约束规则，它们所产生的结果都是类似“ $y$  关系  $mx + b$ ”这样的关系式。不论创建了何种约束规则，它们都是 `NSLayoutConstraint` 类的成员。

### 5.5.1 基本约束规则声明

`NSLayoutConstraint` 类的 `constraintWithItem:attribute:relatedBy toItem:attribute:multiplier:constant:` 方法（这方法名够长吧！）一次能够创建一条约束规则。这些规则可以把一个视图同另一个视图关联起来。

这个方法会创建出严格的“ $view.attribute \ R \ view.attribute * multiplier + constant$ ”关系式。其中  $R$  可以是“等于”（`==`）、“大于等于”（`>=`）或“小于等于”（`<=`）。

请看下列范例代码：

```
[self.view addConstraint:
    [NSLayoutConstraint
        constraintWithItem:textfield
        attribute:NSLayoutAttributeCenterX
        relatedBy:NSLayoutRelationEqual
        toItem:self.view
```

```
attribute:NSLayoutAttributeCenterX
multiplier:1.0f
constant:0.0f]];
```

这段代码会向视图控制器的视图 (`self.view`) 里添加一条新的约束规则, 它会把文本框中心点的横坐标与本视图中心点的横坐标对齐。具体来说, 就是在两个视图中心点的横坐标 (以 `NSLayoutAttributeCenterX` 属性来表示) 之间设立等同 (`NSLayoutRelationEqual`) 关系。放大倍数是 1, 偏移量是 0。这样就产生了下面的关系式:

$$[\text{textfield}]'s\ centerX = ([\text{self.view}]'s\ centerX * 1) + 0$$

这个关系式的意思是: 请把本视图中心点的 X 坐标与文本框中心点的 X 坐标对齐。UIView 的 `addConstraint:` 方法会把这条约束规则添加到视图之中, 该规则与其他的约束规则都会存放在视图的 `constraints` 属性里。

## 5.5.2 用可视化格式字符串声明约束规则

上一节演示了如何创建单条约束关系。`NSLayoutConstraint` 类里还有个方法, 可以根据字符串来创建约束规则, 这种字符串使用基于文本的视觉格式语言来表示其内容。这就好比给 Objective-C 语言高手设计的一套 ASCII 符号图 (ASCII art)。请看下面这个例子:

```
[self.view addConstraints: [NSLayoutConstraint
constraintsWithVisualFormat:@"V:[leftLabel]-15-[rightLabel]"
options:0
metrics:nil
views:NSDictionaryOfVariableBindings(leftLabel, rightLabel)]];
```

上面这行代码会根据可视化格式字符串的内容创建出一套约束规则, 以满足其中的关系。在后面几节中, 我们将会多次看到这种字符串, 它们用来表述视图在横轴 (H) 与纵轴 (V) 方向上与其他视图的关系。本例所用的这个字符串的意思是说: 确保 `rightLabel` 出现在 `leftLabel` 下方 15 点处。

创建这种格式字符串的时候要注意下面这些事项:

- 先用 H: 或 V: 前缀来指明规则所针对的轴。
- 字符串里指代视图的变量名都用方括号括起来。
- 两控件之间的固定间距用左右带有“-”的常数来表示, 例如 -15-。
- 本例没有使用选项 (`options` 参数), 不过开发者可以借此来指定对齐的方向是从左至右、从右至左还是从头至尾, 本章前面讨论过这个问题。
- 本例也没有使用 `metrics` 参数, 开发者可以把一份字典 (`NSDictionary`) 传进去, 并在其中提供一些约束规则所用到的常量, 这样就不用再去专门创建相关的格式字符串了。比方说, 我们想令两个文本标签之间的间距可变, 那么请将本例中的 15 替换成某个指标的名字 (例如可以叫作 `labelOffset` 之类的), 然后把该指标的值放在 `metrics` 字典里面。字典的键是指标的名称, 而值的类型则是 `NSNumber`。传入这样一个字典 (例如 `@{@"labelOffset", @15}`) 比每次用到一种宽度时都去创建新的 `NSString` 实例要简单许多。



- **views**: 参数的实际意义与其名称不同, 它不是个包含视图的数组。我们要给该参数传入含有变量绑定 (variable binding) 的字典。这种字典会把作为变量名的字符串与变量所指代的视图关联起来。执行了这样一种操作之后, 开发者就可以在格式字符串里使用诸如 `leftLabel` 及 `rightLabel` 等有意义的符号了。

用格式字符串来构建约束规则时, 总会产生一个包含若干约束规则的数组。某些格式字符串相当复杂, 另外一些则比较简单。我们不太容易看出来每个字符串到底会产生多少条约束规则。请注意: `constraintsWithVisualFormat` 方法会制作出能够满足格式字符串的一系列约束规则, 而开发者要把这些规则全都添加到视图里面去。

### 5.5.3 变量绑定

Auto Layout 系统在处理可视化的约束字符串时, 需要把 `leftLabel` 及 `rightLabel` 等视图名称与它们所表示的实际视图对应起来。这时就需要用到变量绑定了, 我们通过 `NSLayoutConstraint.h` 文件里定义的宏来执行变量绑定, 该文件是 `UIKit` 中的一个头文件。

`NSDictionaryOfVariableBindings()` 宏以任意数量的局部变量作参数。正如大家在早前的范例中所见到的, 这些参数最后不需要添上 `nil`。该宏会根据传入的变量构建一份字典, 其中每个条目的键都是变量名, 而值则是变量本身。比方说, 如果执行下面这条语句:

```
NSDictionaryOfVariableBindings(leftLabel, rightLabel)
```

那么就会构建出如下字典:

```
@{"leftLabel":leftLabel, @"rightLabel":rightLabel}
```

如果你不想使用这个宏, 那么不妨手工制作一份字典, 并把它传给使用可视化格式字符串的约束规则构建器<sup>①</sup>。

## 5.6 格式字符串

创建约束规则时所用的格式字符串遵循下面这套基本语法:

```
(<orientation>:)? (<superview><connection>)? <view>(<connection><view>)*  
(<connection><superview>)?
```

问号表示可选的项目, 而星号则表示该项目可以出现 0 次或多次。虽说看上去比较复杂, 但实际上这些字符串很容易构建出来。下面几节将介绍格式字符串中的各种要素, 并提供丰富的范例来演示它们的用法。

### 5.6.1 方向

字符串开头的那个可选项目表示约束规则所针对的方向 (orientation), `H`: 表示水平方向, `V`: 表示垂直方向。意思是说, 这条规则所约束的是左右方向的布局还是上下方向的布局。

① 指的是 `constraintsWithVisualFormat` 方法。——译者注

假如省略该项目，那么默认就表示左右方向。比方说有这样一个约束字符串："H:[view1][view2]"，它的含义就是把 view2 直接放在 view1 右侧。H 表示这条约束规则所针对的方向。图 5-2 左侧那张示意图演示了运用这条约束规则之后的界面布局。

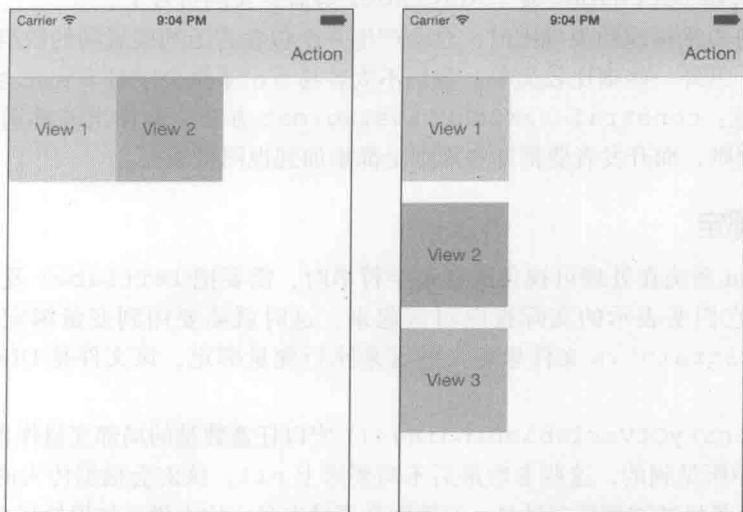


图 5-2 与 "H:[view1][view2]" (左侧截图) 及 "V:|[view1]-20-[view2]-20-[view3]" (右侧截图) 相对应的界面布局

接下来再看一个垂直布局的范例："V:|[view1]-20-[view2]-20-[view3]"。这条约束规则在 view1 与其下方的 view2 之间留出了 20 点空白，然后又在 view2 与其下方的 view3 之间留出了 20 点空白。图 5-2 右侧截图演示了运用这条约束规则之后的界面布局。

如果不继续施加约束，那么仅凭目前的约束规则是无法将界面准确排布好的。Auto Layout 会自行决定剩下的部分，不过它选出来的布局通常是错误的。在左侧截图中，还应该指定垂直方向的约束规则。如果不指定，那么两个视图就会像本例这样全都贴着上级视图的顶边。而在右侧截图中，还应指定水平方向的约束规则。如果不指定，那么三个视图就会像本例这样全都紧贴上级视图的左界。

要产生图 5-2 所示的布局效果，需要分别针对水平方向和垂直方向的约束规则执行：

```
[self.view addConstraints:[NSLayoutConstraint
    constraintsWithVisualFormat:@"H:[view1][view2]"
    options:0 metrics:nil
    views:NSDictionaryOfVariableBindings(view1, view2)]];
```

及

```
[self.view addConstraints:[NSLayoutConstraint
    constraintsWithVisualFormat:@"V:|[view1]-20-[view2]-20-[view3]"
    options:0 metrics:nil
    views:NSDictionaryOfVariableBindings(view1, view2, view3)]];
```



注意上面这个格式字符串开头的竖线(|)。竖线总是代表上级视图。我们只会在格式字符串的首尾两端看到它。如果出现在开头,那么它就紧跟着表示水平方向或垂直方向的那个标识符(比方说"V:|..."或"H:|...")。要是它出现在末尾,那么就放在字符串结束处的那个引号之前("...|")。在 iOS 7 系统中,如果把本例的竖线省略掉,那么视图就会出现在状态栏及导航栏的下方。



**提示** 调试程序时,可以使用 UIView 的 `constraintsAffectingLayoutForAxis:` 方法来获取影响水平布局或垂直布局的所有约束规则。但在发布软件产品时,请勿将该方法包含在内。它不是供正式软件使用的,而且苹果公司已经明确表示 App Store 里的软件不应该调用它。

### 5.6.2 连接

在视图名称之间放置连接(connection),就可以指定视图的排布流程了。空连接(也就是把连接省掉)表示紧跟其后。

图 5-2 中的第一条约束规则是 `"H:[view1][view2]"`,它里面就用到了空连接。在 `view1` 的右方括号与 `view2` 的左方括号之间没有其他内容,这就表示此约束规则要求 `view2` 直接出现在 `view1` 右侧。

连字符(hyphen, -)表示一小段固定空间。`"H:[view1]-[view2]"` 这条规则就用连字符来表示视图之间的连接,它会在 `view1` 与 `view2` 之间留出标准的空白(这个标准由苹果公司来定),如图 5-3 所示。

如果在两个连字符之间放上数值常数,那么就可以精确地指定间隔尺寸了。`"H:[view1]-30-[view2]"` 这条约束规则会在两个视图之间添加 30 点空白,如图 5-4 所示。这段空白看上去要比由单个连字符所产生的小缺口宽一些。

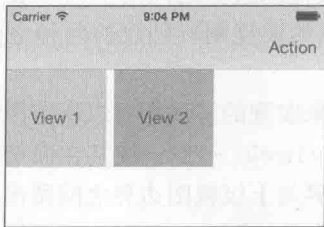


图 5-3 `"H:[view1]-[view2]"` 这条规则中的连接会在两视图之间添加空白

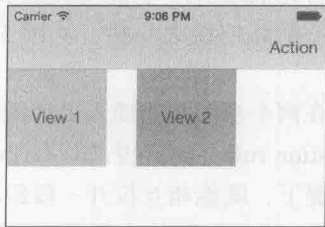


图 5-4 `"H:[view1]-30-[view2]"` 这条约束规则会在两个视图之间插入 30 点的固定空白,它所产生的间隔要比上一条约束规则宽

`"H:|[view1]-[view2]|"` 这个格式字符串会从上级视图开始,逐步描述水平方向上的排布方式。上级视图的边界紧贴着 `view1` 的边界, `view1` 后面是一段间隔,间隔后面再

跟着 view2, 而 view2 的后边沿也紧贴着上级视图, 整个效果如图 5-5 所示。

这条约束规则会将 view1 与上级视图左对齐, 同时将 view2 与上级视图右对齐。为了实现这种布局, 系统必须放弃一些原有的设定才行。要么调整左边那个视图的大小, 要么调整右边那个视图的大小, 只有这样, 才能满足此约束规则。笔者运行本书第 5 版的测试程序时发现, 系统调整的是 view1, 其效果如图 5-5 所示, 而在运行本书第 4 版的测试程序时, 发现系统调整的是 view2。

很多情况下, 我们不想令视图的边界紧贴着上级视图。于是, 可以指定 "H:|-[view1]-[view2]-|" 规则, 它与上面那条规则类似, 只是会在上级视图的左边界与 view1 的左边界之间留出空白, 同时也会在 view2 的右边界与上级视图的右边界之间留出空白, 如图 5-6 所示。

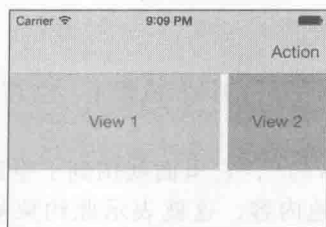


图 5-5 "H:|[view1]-[view2]|" 这条约束规则要求左右两个视图必须分别和上级视图的左右边界对齐。由于二者之间的空白是固定的, 所以至少要调整其中一个视图的宽度才能满足此规则

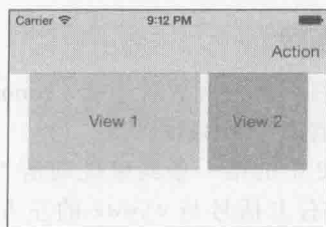


图 5-6 "H:|-[view1]-[view2]-|" 约束规则会在视图的边界与上级视图的边界之间留出空白

对于视图与上级视图之间的空白, 其尺寸遵循标准的 IB/Cocoa Touch 布局规则, 苹果公司没有在 iOS API 中说明其细节。视图边界与上级视图边界之间的空白一般要比同级视图之间的默认空白大一些。从图 5-6 中可以看到由该约束规则所创建的两块空白之间的区别。

要想在两个视图之间插入灵活的空白, 也是有办法来实现的。我们可以指定两者的关系规则 (relation rule, 比方说 "H:|-[view1]-(>=0)-[view2]-|"), 使其在保持各自尺寸不变的前提下, 既能相互拉开一段距离, 又能在自身边界与上级视图边界之间留出空白, 如图 5-7 所示。这条规则的意思是两视图之间至少有 0 个点的间隔。该规则使得系统在排布这两个视图的时候可以灵活地调整其间距。笔者觉得, 这条关系规则里面所使用的那个数字应该小一些才对, 不然的话, 可能会无意间干扰视图的其他几何规则。

字符串里面当然不局限于一到两个视图, 也可以放入三四个, 或者更多的视图。比方说可以写出这样一条约束规则: "H:|-[view1]-[view2]-(>=5)-[view3]-|". 第三个视图与另外两个视图之间留有灵活的空白。图 5-8 演示了满足该约束规则的布局方式。

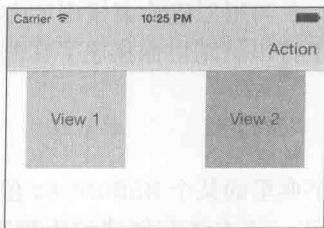


图 5-7 "H:|- [view1]-(&gt;=0)-[view2]-|"

规则会在两视图之间留出灵活的空白，使两者在保持各自尺寸不变的前提下，能够相互拉开一段距离

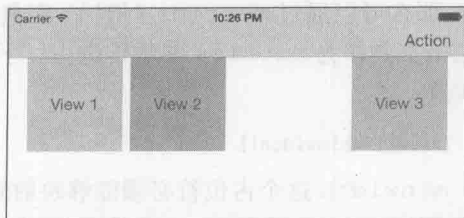


图 5-8 "H:|- [view1]-[view2]-(&gt;=5)-[view3]-|"

规则可以限定三个视图之间的关系

## 5.7 谓词

上一节最后两个范例都用到了带有比较运算符的关系规则。这些规则也叫谓词 (predicate)，它是对视图元件之间的关系所下的断言。谓词需要用圆括号括起来。比方说，我们可以用下面这个字符串来规定视图的尺寸至少是 50 点：

```
[view1(>=50)]
```

该谓词只和一个视图有关。请注意，这个谓词出现在描述视图所用的那一对方括号之内，而没有像上一节那样出现在两个视图之间的连接处。谓词并不局限于单个标准。例如，我们可以用类似的规则指明视图的尺寸必须在 50 点到 70 点之间。如果要添加复合谓词 (compound predicate)，那么请把规则的各个部分用逗号隔开：

```
[view1(>=50, <=70)]
```

相对关系谓词可以规定视图尺寸的增长方式。如果想令某视图尽量占满其上级视图，那么可以宣称其尺寸是个大于 0 的值。下面这条规则会在水平方向上拉伸 view1，使其在保留边界空白的前提下，尽量填满上级视图：

```
H:|- [view1(>=0)]-|
```

图 5-9 演示了这条规则的布局效果。

如果要指定的是等同关系 (==)，那么可以在格式字符串的谓词里把两个等号省掉。比方说，[view1(==120)] 等价于 [view1(120)]，而 [view1]-(==50)-[view2] 和 [view1]-50-[view2] 的含义是相同的。

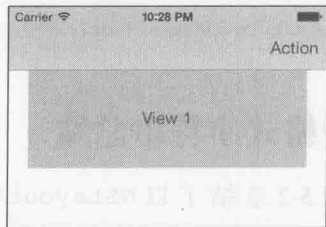


图 5-9 "H:|- [view1(>=0)]-|" 规则给视图添加了一个灵活的谓词，令其可以横跨上级视图。由于上级视图的边界和本视图的边界之间留有空白，所以这两条边线之间会有空隙

### 5.7.1 指标

如果事先不知道常量值是多少（也就是说，我们无法预先确定常量是 120 或 50 这样的数

值), 那么可以通过 `metrics` (指标) 字典来提供该值。在调用创建约束规则的方法时, 我们把该字典作为 `metrics` 参数传进去。下面这个格式字符串就用指标描述了其值尚不明确的常量:

```
[view1(>=minwidth)]
```

`minwidth` 这个占位符必须能够映射到 `metrics` 字典里的某个 `NSNumber` 值。解决方案 5-2 的 `constrainSize:` 方法详细演示了指标的用法。该方法在创建约束规则的时候, 把指标所对应的值放在了一份字典里面, 大家可以通过那段代码明白 `metrics` 的用法。

### 5.7.2 描述两个视图关系的谓词

谓词并不局限于数字常量。我们也可以在两个视图的尺寸之间建立关系, 以保证某视图不会比另外一个视图更大。下面这个例子限定了 `view2` 的尺寸, 使其在目前所涉及的这条轴上不会比 `view1` 更大:

```
[view2(<=view1)]
```

用格式字符串来描述视图之间的比较关系只适用于一些简单的场合。如果想建立更为复杂的关系, 比方说要描述中心点、顶边及高度之间的关系, 那就不要使用可视化的格式字符串了, 而是应该改用针对项目的约束构造器<sup>①</sup>。

### 5.7.3 优先级

每条约束规则都可以通过 `@` 符号 (优先级) 来指定优先级, 符号后面可以是数字, 也可以是某项指标。比方说, 我们想把 `view1` 的尺寸定为 500 点, 但又想把这条规则的优先级设置得低一些, 那么可以使用下列字符串:

```
[view1(500@10)]
```

优先级要放在谓词之后。例如, 下面就是一个在谓词中嵌有优先级的格式字符串:

```
[view1]-(>=50@30)-[view2]
```

## 5.8 格式字符串总结

表 5-2 总结了以 `NSLayoutConstraint` 类的 `constraintsWithVisualFormat:options:metrics:views:` 方法来创建约束规则时所用的格式字符串的各个部件。

表 5-2 可视化的格式字符串

类 型	格 式	举 例
在水平方向或垂直方向上排布	H: V:	V:[view1]-15-[view2] 把 view2 放在 view1 底边下方 15 点处

① 实际上指的就是 `constraintWithItem` 方法。——译者注

(续)

类 型	格 式	举 例
视图	[item]	[view1] 系统会在传给 views 参数的字典里面寻找与方括号中的名称相绑定的 UIView 实例
上级视图		H: [view1]  令 view1 与上级视图同宽
关系	== <= >=	H:[view1]-(>=20)-[view2] 令 view2 的前边沿距离 view1 的后边沿至少 20 点
指标	metric	H:[view1](<=someWidth) V:[view1]-mySpacing-[view2] 指标就是 metrics 字典的键。在开发者经由 metrics 参数传入的字典当中, 必须能够找到与 someWidth 及 mySpacing 相对应的 NSNumber 值
紧贴对齐	[item][item]	H:[view1][view2] 令 view1 的后边沿紧贴 view2 的前边沿
弹性空间	[item]-(>=0)-[item]	[view1]-(>=0)-[view2] 这两个视图都可以尽量拉开距离, “其间距至少是 0 点”
固定空间	[item]-[item]	[view1]-[view2] 在两个视图之间留下一小段固定的空白 (其大小是 8 点, 这是由系统定义的)
自定义的固定空间	[item]-gap-[item]	V:[view1]-20-[view2] 令 view2 的顶部距离 view1 底部 20 点
固定宽度或固定高度	[item(size)] [item(==size)]	[view1(50)] 将 view1 在当前坐标轴上的尺寸设为 50 点
最小或最大宽度 / 高度	[item(>=size)] [item(<=size)]	[view1(>=50)] [view1(<=50)] 限定 view1 在这条坐标轴上的最小尺寸或最大尺寸
令视图的宽度 / 高度与另外一个视图相匹配	[item(==item)] [item(<=item)] [item(>=item)]	[view1(==view2)] 令两个视图在当前坐标轴上的尺寸相同
紧贴上级视图的边界	[item] [item]	V: [view1] 令 view1 的顶边紧贴上级视图的顶边
在本视图的边界与上级视图的边界之间留出空白	-[item] [item]-	-[view1] 在当前坐标轴的方向上, 给 view1 的边界和上级视图的边界之间留出固定的空白 (20 点)
在本视图与上级视图的边界之间留出一定数量的空白	-gap-[item] [item]-gap-	H: -15-[view1] 在视图的前边沿与上级视图的边界之间留出 15 点空隙
优先级 (从 0 到 1000)	@value	[view1(<=50@20)] 令 view1 在当前坐标轴上的最大尺寸为 50 点, 同时把这条规则的优先级降到很低 (20)

## 5.9 用格式字符串将视图对齐并灵活调整其尺寸

通过约束规则，我们很容易就能指定视图的对齐方式：

- "H:[self]"、"H:[self]|"、"V:[self]" 及 "V:[self]|" 这 4 种格式字符串分别产生左对齐、右对齐、顶端对齐及底端对齐的效果。
- 向上述字符串中添加表示尺寸关系的谓词，即可实现拉伸至左边界、拉伸至右边界等效果："H:[self(>=0)]"、"H:[self(>=0)]|"、"V:[self(>=0)]"、"V:[self(>=0)]|"。
- 如果再添一条竖线，那么就表示在整条坐标轴方向上面拉伸，也就是令视图从左至右或从上至下跨越整个上级视图："H:[self(>=0)]|" 或 "V:[self(>=0)]|"。
- 可以添加连字符，以便在拉伸本视图的时候，给它的边界与上级视图的边界之间留出一些空隙："H:|-[self(>=0)]-|"、"V:|-[self(>=0)]-|"。

## 5.10 处理约束规则的流程

系统要经过多个阶段的处理，才能把视图内容显示出来。引入 Auto Layout 机制之前，这个过程分为两个阶段，即布局阶段（layout phase）与渲染阶段（rendering phase）。而 Auto Layout 机制则在这两个传统的阶段之前，又添加了约束阶段（constraint phase）。

开发者可以实现 `layoutSubviews` 方法，以便在布局阶段修改视图里各个子视图的几何特征。当 iOS 认定某视图的布局已经无效时，就会调用该方法，开发者可在其中手动更新子视图的排布方式。除了系统自行调用之外，开发者可以手工调用 `setNeedsLayout` 或 `layoutIfNeeded` 方法来请求重新排版。前者是个比较温和的请求，iOS 系统会把很多个这样的请求合并起来，然后在适当的时机调用 `layoutSubviews` 方法。而后者则更为迫切，调用了它之后，系统几乎立刻就会执行 `layoutSubviews`。

在渲染阶段，开发者可通过实现 `drawRect:` 方法来完全控制视图 UI 的绘制。当系统认定视图内容已经无效时，它会调用 `drawRect:` 方法，以便对视图进行底层绘制。需要改变渲染内容的时候，开发者可以通过 `setNeedsDisplay` 及 `setNeedsDisplayInRect:` 方法请求视图重新绘制它自己，这两者都会触发 `drawRect:`。

有了 Auto Layout 机制之后，上述两个阶段前面还会多出一个约束阶段（*constraints phase*）。通过实现 `updateConstraints` 方法，我们可在该阶段创建并更新 Auto Layout 系统所使用的约束规则。与布局阶段相似，iOS 系统可以自行判定某个视图的约束规则已经无效，从而开始执行约束阶段，而另一方面，开发者也可以手动触发此过程，相关的两个方法叫作 `setNeedsUpdateConstraints` 及 `updateConstraintsIfNeeded`，其名称与行为与刚才说的那两组方法类似。

如果覆写了视图的 `updateConstraints` 方法，那么务必要在该方法返回之前调用 `[super updateConstraints]`。待约束阶段结束之后，Auto Layout 会适当地计算出视图内所有子视图的几何特征。

每完成一个阶段，系统就会进入下一阶段。也就是说，执行完约束阶段之后，系统会执行布局阶段，而执行完布局阶段，又会执行渲染阶段。

这种阶段处理方式给了我们一些意想不到的机会。在执行布局阶段的时候，系统已经在约束阶段中把所有约束规则都计算好了，并且已经根据这些规则设定了视图的几何特征。而在布局阶段，则可以用一种与约束规则相违背的方式继续修改视图的几何特征。另外，也可以根据由约束规则所产生的几何特征来决定视图最终的样貌。由于系统在上个阶段已经把约束规则算好了，所以本阶段对视图样貌所做的修改会一直保留到下一轮约束阶段为止。请勿在布局阶段修改约束规则，否则系统又会触发相关方法来更新约束规则，从而使程序在约束阶段与布局阶段之间陷入无限循环。

大部分情况下都用不到这些“挂钩”方法。不过在偶尔用到它们时，这些方法会展现出强大的灵活度与控制力。

## 5.11 管理约束规则

无论约束规则是如何创建出来的，它们都属于 `NSLayoutConstraint` 类。开发者可通过 `addConstraint:` 方法给视图逐条添加约束规则，也可以把多条规则放在数组里，并通过 `addConstraints:` 实例方法（注意方法名最后的 `s`）将其添加到视图里面。在日常开发中，我们经常会把很多约束规则一并保存在某个数组里。

存放约束规则的视图自然应该是离规则所涉及的各视图最近的那个共同祖先。在其身上装配约束规则的视图必须是规则里各个视图的共同祖先。我们可以针对 `NSLayoutConstraint` 编写 `category`，并在其中实现一个方法，用程序代码把该规则自动装配到正确的视图上面。这个功能留给读者作为练习。

开发者可以随时添加并移除约束规则。`removeConstraint:` 及 `removeConstraints:` 方法分别可以从给定的视图中移除一条约束规则或一组约束规则。由于这两个方法是基于 `NSLayoutConstraint` 对象来运作的，所以它的执行效果可能与开发者所想的不符。

比方说，我们已经构建了一条居中约束规则，并将其添加到视图里面了。如果现在又构建了一条与之等效的约束规则，然后通过 `removeConstraint:` 来移除它，那么你会发现该规则并没有从视图中删掉。这两条约束规则虽然等效，但并不是完全相同的两个对象。下列代码演示了这个问题：

```
[self.view addConstraint:
    [NSLayoutConstraint constraintWithItem:textField
        attribute:NSLayoutAttributeCenterX
        relatedBy:NSLayoutRelationEqual
        toItem:self.view
        attribute:NSLayoutAttributeCenterX
        multiplier:1.0f constant:0.0f]];

[self.view removeConstraint:
    [NSLayoutConstraint constraintWithItem:textField
        attribute:NSLayoutAttributeCenterX
```



```
relatedBy:NSLayoutRelationEqual
 toItem:self.view
 attribute:NSLayoutAttributeCenterX
 multiplier:1.0f constant:0.0f]];
```

如果执行上面这两个方法调用,那么 `self.view` 实例中就会含有一开始所构建的那条约束规则,但是 `removeConstraint:` 方法却会遭到忽略。移除不归该视图所拥有的约束规则是没有效果的。

有两种解决办法。一种是把开头添加的那条约束规则保存到局部变量里。比如可以写成下面这样:

```
NSLayoutConstraint *myConstraint =
[NSLayoutConstraint constraintWithItem:textField
 attribute:NSLayoutAttributeCenterX
 relatedBy:NSLayoutRelationEqual
 toItem:self.view
 attribute:NSLayoutAttributeCenterX
 multiplier:1.0f constant:0.0f];
[self.view addConstraint:myConstraint];

// later
[self.view removeConstraint:myConstraint];
```

另一种是采用解决方案 5-1 中的方法来比较已有的约束规则与待移除的约束规则,并把数值相符者从视图里删掉。

具体采用哪种办法要看约束规则是静态的还是动态的(静态约束规则是在视图的整个生命期中使用的规则,动态约束规则是按需求随时指派的规则)。如果将来可能要移除某条约束规则,那么可将其保存到局部变量中,以便稍后移除,也可以采用解决方案 5-1 中详述的那种技巧来处理此问题。

管理约束规则时,应注意下面几点:

- 可以在 `UIView` 实例上面添加或移除约束规则。与之相关的几个核心方法是: `addConstraint:(addConstraints:)`、`removeConstraint:(removeConstraints:)` 及 `constraints`。`constraints` 方法会返回包含所有约束规则的数组。
- 约束规则的适用范围并不局限于容器视图,几乎所有视图都可以施加约束规则。(通过名为 `requiresConstraintBasedLayout` 的类方法,我们可以知道这种视图类是不是必须放在基于约束的布局系统里才能够正常运作。)
- 如果要以编程的方式为子视图施加约束规则,那么请把子视图的 `translatesAutoresizingMaskIntoConstraints` 属性关掉。读者稍后就能在本章的范例代码中看到这个属性,而且本章末尾的 5.17 节也会深入讨论它。

## 5.12 解决方案:实现约束规则之间的对比

所有的约束规则都遵循同一套固定的结构,而且都有相关的优先级:



$view1.attribute(relation) view2.attribute * multiplier + constant$

上述等式的每个部分都与 NSLayoutConstraint 对象的属性相对应，它们分别是 priority<sup>①</sup>、firstItem、firstAttribute、relation、secondItem、secondAttribute、multiplier 与 constant。通过这些属性，我们很容易比对两条约束规则。

UIView 会把约束规则当成 NSLayoutConstraint 对象来保存或移除。即便两条规则所描述的含义相同，只要其内存位置不同，它们就是不相等的。如果实现了规则之间的对比功能，那么就无须将规则专门存放到变量里面了，而是可以通过程序码随时添加并移除它们。

解决方案 5-1 编写了三个方法。constraint:matches: 方法会比较两条约束规则的相关属性，以此判断二者是否相同。请注意，在比较的时候，我们只考虑等式本身，而不考虑优先级（如果读者想添加这个因素，那只需稍微修改一下代码就能实现），因为两条约束规则只要描述的是同一套约束方式，那么无论开发者给其指定何种优先级，它们实际上都是等效的。

还有两个方法分别叫作 constraintMatchingConstraint: 及 removeMatchingConstraint:，前者在视图里查找与开发者所给规则相匹配的首条约束规则，而后者则把此约束规则从视图中移除。

解决方案 5-1 会把某条约束规则从视图里删除，并换上一条新约束规则，这样的话，此视图就会从上级视图的顶部跑到上级视图的底部了。对于本例这种情况来说，我们完全可以把约束规则保存到实例变量中，并在稍后将其从视图里移除。不过，在需要管理多条约束规则或动态移除某约束规则的时候，这种相似约束规则的查询与移除功能还是非常有用的。



**提示** 解决方案 5-1 针对 UIView 类编写了名为 ConstraintHelper 的 category。本章将会用到并逐步扩充这个 category，笔者会在里面编写一些工具方法，供大家在开发自己的应用程序时使用。

### 解决方案 5-1 实现约束规则之间的对比

```
@implementation UIView (ConstraintHelper)
// This ignores any priority, looking only at y (R) mx + b
- (BOOL)constraint:(NSLayoutConstraint *)constraint1
    matches:(NSLayoutConstraint *)constraint2
{
    if (constraint1.firstItem != constraint2.firstItem) return NO;
    if (constraint1.secondItem != constraint2.secondItem) return NO;
    if (constraint1.firstAttribute != constraint2.firstAttribute) return NO;
    if (constraint1.secondAttribute != constraint2.secondAttribute) return NO;
    if (constraint1.relation != constraint2.relation) return NO;
    if (constraint1.multiplier != constraint2.multiplier) return NO;
    if (constraint1.constant != constraint2.constant) return NO;

    return YES;
}
```

① 相当于规则的优先级。——译者注

```

// Find first matching constraint (priority ignored)
- (NSLayoutConstraint *)constraintMatchingConstraint:
    (NSLayoutConstraint *)aConstraint
{
    for (NSLayoutConstraint *constraint in self.constraints)
        if ([self constraint:constraint matches:aConstraint])
            return constraint;

    for (NSLayoutConstraint *constraint in self.superview.constraints)
        if ([self constraint:constraint matches:aConstraint])
            return constraint;

    return nil;
}

// Remove constraint
- (void)removeMatchingConstraint:(NSLayoutConstraint *)aConstraint
{
    NSLayoutConstraint *match =
        [self constraintMatchingConstraint:aConstraint];
    if (match)
    {
        [self removeConstraint:match];
        [self.superview removeConstraint:match];
    }
}

@end

```

---

### 获取解决方案代码

访问 <https://github.com/erica/iOS-7-Cookbook> 网页，并打开“C05 Constraints”文件夹，即可找到与本章中的解决方案相对应的完整范例项目。

---

## 用动画来表现约束规则的变更过程

解决方案 5-1 会把“本视图与上级视图顶端对齐（或底端对齐）”这条约束规则删掉，并添加一条反向约束规则，以此来移动当前视图。下面这段代码是从解决方案 5-1 里节选的，它可以令 view1 从上级视图的顶端移到底部：

```

NSLayoutConstraint * constraintToMatch =
    [NSLayoutConstraint constraintWithItem:view1
        attribute:NSLayoutAttributeTop
        relatedBy:NSLayoutRelationEqual toItem:self.view
        attribute:NSLayoutAttributeTop multiplier:1.0f constant:0];
[self.view removeMatchingConstraint: constraintToMatch];

NSLayoutConstraint * updatedConstraint =
    [NSLayoutConstraint constraintWithItem:view1
        attribute:NSLayoutAttributeBottom

```

```

relatedBy:NSLayoutRelationEqual toItem:self.view
attribute:NSLayoutAttributeBottom multiplier:1.0f constant:0];
[self.view addConstraint:updatedConstraint];

[self.view layoutIfNeeded];

```

最后一行的 `layoutIfNeeded` 语句会令 Auto Layout 系统重新处理约束规则，并把更新之后的视图渲染出来。不过，这样做出来的切换效果有些突兀。实际上，只需添加几行简单的代码，就能把这个过程用动画表现出来。我们需要做的就是调用 `layoutIfNeeded` 的那行语句放在块里，并传给 `animateWithDuration` 方法的 `animations` 参数：

```

[UIView animateWithDuration:0.3 animations:^(
    [self.view layoutIfNeeded];
)];

```

这样一来就可以看到视图从顶部移动到底部的动画了，它与直接在块里手工修改原点所做出的动画效果相同。

## 5.13 解决方案：创建尺寸固定且受规则约束的视图

使用约束规则的时候，要以一种新的思维方式来考虑视图的布局。现在我们不是像原来那样通过设置框架来固定其尺寸和位置，而是要采用一套全新的思路，通过约束规则来限定视图的布局。

在使用 Auto Layout 之前，我们会用下面这个工具方法创建标签控件：

```

- (UILabel *)createLabelTheOldWay:(NSString *)title
{
    UILabel *label = [[UILabel alloc]
        initWithFrame:CGRectMake(0.0f, 0.0f, 100.0f, 100.0f)];
    label.textAlignment = NSTextAlignmentCenter;
    label.text = title;

    return label;
}

```

但在使用了 Auto Layout 之后，要以另一种编程方式来创建视图。我们不再创建尺寸固定的框架并设置其中心点，而是用代码向视图里添加约束规则，以此调整其大小和位置。

### 5.13.1 禁用 `translatesAutoresizingMaskIntoConstraints`

`autoresizing`（自动调整尺寸）是一种由 IB 所使用的 struts-and-springs 式布局工具，它也可以指程序代码里用到的一些相关标志，例如 `UIViewAutoresizingFlexibleWidth` 等。如果要通过这些手段来指定视图的自动调整行为，那么在定义约束规则的时候，就不应该再引用该视图了。

使用约束规则之前，应该先禁用视图中的有关属性，该属性会把涉及自动调整功能的掩码自行转换为约束规则。如果启用该属性，那么视图会通过自动调整功能有关的掩码来参

与到 Auto Layout 系统里面，要是禁用它，开发者就需要自己添加约束规则。

这个与约束规则有关的属性叫作 `translatesAutoresizingMaskIntoConstraints`。如果将其设为 NO，我们就可以放心地向视图中添加自己的约束规则，而不必担心它会与系统自动生成的规则相冲突。这一点非常重要。若是尚未禁用该属性就直接开始添加约束，则可能会引发冲突。由自动调整功能所生成的那些规则无法与开发者自己编写的这些规则和平共处。如果程序在运行的时候出现了此问题，就会产生下面这样的错误消息：

```
2012-06-24 15:34:54.839 HelloWorld[64834:c07] Unable to simultaneously satisfy
constraints.
Probably at least one of the constraints in the following list is one you don't
want. Try this: (1) look at each constraint and try to figure out which you don't
expect; (2) find the code that added the unwanted constraint or constraints and
fix it. (Note: If you're seeing NSAutoresizingMaskMaskLayoutConstraints that you don't
understand, refer to the documentation for the UIView property
translatesAutoresizingMaskIntoConstraints)
(
    "<NSLayoutConstraint:0x6ec9430 H:[UILabel:0x6ec5210(100)]>",
    "<NSAutoresizingMaskMaskLayoutConstraint:0x6b8e2a0
        h---& v---& H:[UILabel:0x6ec5210(0)]>"
)
Will attempt to recover by breaking constraint
<NSLayoutConstraint:0x6ec9430 H:[UILabel:0x6ec5210(100)]>
Break on objc_exception_throw to catch this in the debugger.
```

在基于自动调整功能的布局和基于约束规则的布局之间进行抉择是编写代码时的一项重要任务。

### 5.13.2 令视图出现在上级视图范围内

解决方案 5-2 中的第一个方法叫作 `constrainWithinSuperviewBounds`，它会把某视图完全放在其上级视图的范围内。该方法创建了四条约束规则来保证这一点。其中一条规则要求本视图的左边界必须与上级视图的左边界对齐，或位于其右侧，另一条规则要求本视图的顶端必须与上级视图的顶端对齐，或位于其下方，其余两条也与之相似。

之所以要创建这个方法，是因为在约束规则比较宽松的系统中，视图的原点完全有可能是负值，从而导致自己出现在屏幕之外而不为用户所见。该方法的基本意思是：在排布这个子视图的位置时，请考虑到 (0, 0) 原点与上级视图的尺寸，而不要把它排布到上级视图外面去。

在日常开发中，我们不一定非要设置这套约束规则。但在刚开始接触约束并想通过代码来研究它的时候，这种约束规则就显得非常有用。它能够确保本视图不会越出上级视图的范围，从而令我们可以测试其他各项约束规则，并观察这些规则之间的关系。

此外，在熟悉了约束系统之后，我们可能还想给程序添加一些调试用的反馈代码，以便在主视图加载完毕并运用了约束规则之后，能够知道我们自己的视图位于何处。比方说，可以把下面这个 for 循环添加到 `viewDidAppear:` 方法中：

```

- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    for (UIView *subview in self.view.subviews)
        NSLog(@"View (%d) location: %@",
              [self.view.subviews indexOfObject:subview],
              NSStringFromCGRect(subview.frame));
}

```

### 5.13.3 限定视图的尺寸

解决方案 5-2 的第二个方法叫作 `constrainSize:`，它会把视图的尺寸固定为开发者所指定的 `CGSize`。在定义约束规则的时候，这是一种经常会用到的操作。我们不能再像原来那样设定视图的框架。另外也请记住：这些约束规则都只是一种请求，未必总是与最终的布局完全相符。假如没有合理地设计约束规则，那么宽度为 100 个点的文本框在最终的程序界面里其宽度可能会变成 107 个点，或是更糟。

我们可以在约束规则中针对某视图来指定其宽度或高度，然而这两条规则所要用的宽度值和高度值是不能预判的，因为 `constrainSize:` 方法要适用于各种视图才行。于是，我们把这两个值先放在 `metrics` (指标) 字典里，然后再传给 `constraintsWithVisualFormat`。所谓指标，其实就是约束规则里所用的一些数值变量。

这两条规则所用的两个指标分别叫作 `theHeight` 和 `theWidth`，这些名称完全可以随意选取。开发者应该把字符串作为键 (key)，放在传给 `metrics:` 参数的字典里。在调用创建约束规则的那个方法时，把这个字典传进去。指标里的每个键必须出现在这份字典中，而且它所对应的值必须是个 `NSNumber`。

方法里的两条约束规则设定了我们所期望的视图宽度及视图高度。两个格式字符串 (也就是 `"H:[self(theWidth)]"` 及 `"V:[self(theHeight)]"`) 分别告诉约束系统当前这个视图在横轴和纵轴上的尺寸应该是多少。

第三个方法叫作 `constrainPosition:`，它会构建两条约束规则，用于确定本视图的原点在上级视图里的位置。请注意看该方法在调用 `constraintsWithVisualFormat` 时是如何使用 `constant` 参数的。

### 5.13.4 把前面各节内容拼装起来

本书前面曾经列出了一个方法，它以传统方式创建标签控件，有了上述工具之后，我们便可采用 `Auto Layout` 来改写那个方法，新方法比原版本强大许多：

```

- (UILabel *)createLabelWithTitle:(NSString *)title
  onParent:(UIView *)parentView
{
    UILabel *label = [[UILabel alloc] init];
    label.textAlignment = NSTextAlignmentCenter;
    label.text = title;

    // Add label to parent view so constraints can be added
}

```

```

[parentView addSubview:label];

// Turn off automatic translation of autoresizing masks into constraints
label.translatesAutoresizingMaskIntoConstraints = NO;

// Add constraints
[label constrainWithinSuperviewBounds];

[label constrainSize:CGSizeMake(100, 100)];
[label constrainPosition:CGPointMake(50, 50)];

return label;
}

```

我们要用约束规则把上述标签和其上级视图联系起来，然而在添加规则之前，必须先把标签控件作为子视图放到上级视图里。在其上添加约束规则的那个视图和规则中所提到的视图必须位于同一个视图体系中，而且规则里的那些视图彼此之间也必须处在这个体系里，否则就会引发意想不到的行为，甚至会令程序在运行的时候崩溃。所以，我们有时可能要稍稍调整一下代码顺序，比方说，上面这个方法就会在添加约束规则之前，先把标签放在上级视图中。

无论按何种顺序来调整视图的生成代码，我们都要遵循下列步骤：首先创建视图，然后将其添加到上级视图里，接下来禁用 `translatesAutoresizingMaskIntoConstraints`，最后运用必要的约束规则。

#### 解决方案 5-2 用最基本的约束规则来限定视图的尺寸

```

@implementation UIView (ConstraintHelper)
- (void)constrainWithinSuperviewBounds
{
    if (!self.superview) return;

    // Constrain the top, bottom, left, and right to
    // within the superview's bounds
    [self.superview addConstraint:[NSLayoutConstraint
        constraintWithItem:self attribute:NSLayoutAttributeLeft
        relatedBy:NSLayoutRelationGreaterThanOrEqual
        toItem:self.superview attribute:NSLayoutAttributeLeft
        multiplier:1.0f constant:0.0f]];
    [self.superview addConstraint:[NSLayoutConstraint
        constraintWithItem:self attribute:NSLayoutAttributeTop
        relatedBy:NSLayoutRelationGreaterThanOrEqual
        toItem:self.superview attribute:NSLayoutAttributeTop
        multiplier:1.0f constant:0.0f]];
    [self.superview addConstraint:[NSLayoutConstraint
        constraintWithItem:self attribute:NSLayoutAttributeRight
        relatedBy:NSLayoutRelationLessThanOrEqual
        toItem:self.superview attribute:NSLayoutAttributeRight
        multiplier:1.0f constant:0.0f]];
    [self.superview addConstraint:[NSLayoutConstraint
        constraintWithItem:self attribute:NSLayoutAttributeBottom

```

```

        relatedBy:NSLayoutRelationLessThanOrEqual
        toItem:self.superview attribute:NSLayoutAttributeBottom
        multiplier:1.0f constant:0.0f]];
    }

- (void)constrainSize: (CGSize)aSize
{
    NSMutableArray *array = [NSMutableArray array];

    // Fix the width
    [array addObjectsFromArray: [NSLayoutConstraint
        constraintsWithVisualFormat:@"H:[self (theWidth@750)]"
        options:0 metrics:@{@"theWidth":@(aSize.width)}
        views:NSDictionaryOfVariableBindings(self)]];

    // Fix the height
    [array addObjectsFromArray: [NSLayoutConstraint
        constraintsWithVisualFormat:@"V:[self (theHeight@750)]"
        options:0 metrics:@{@"theHeight":@(aSize.height)}
        views:NSDictionaryOfVariableBindings(self)]];

    [self addConstraints:array];
}

- (void)constrainPosition: (CGPoint)aPoint
{
    if (!self.superview) return;

    NSMutableArray *array = [NSMutableArray array];

    // X position
    [array addObject: [NSLayoutConstraint constraintWithItem:self
        attribute:NSLayoutAttributeLeft relatedBy:NSLayoutRelationEqual
        toItem:self.superview attribute:NSLayoutAttributeLeft
        multiplier:1.0f constant:aPoint.x]];

    // Y position
    [array addObject: [NSLayoutConstraint constraintWithItem:self
        attribute:NSLayoutAttributeTop relatedBy:NSLayoutRelationEqual
        toItem:self.superview attribute:NSLayoutAttributeTop
        multiplier:1.0f constant:aPoint.y]];

    [self.superview addConstraints:array];
}

@end

```

## 5.14 解决方案：将两个视图居中对齐

如果想把两个视图居中对齐，可以将某视图的中心点属性（也就是 centerX 及

centerY) 同其容器的对应属性关联起来。解决方案 5-3 里面有两个方法可以获取某视图的上级视图, 并在这两个视图的中心点之间施加等同关系。

请注意, 这些约束规则是添加在上级视图而不是子视图身上的, 这是因为我们不能在约束规则里引用视图所在子树 (subtree) 之外的其他视图。若是那样做, 则会引发下列错误:

```
2012-06-24 16:09:14.736 HelloWorld[65437:c07] *** Terminating app due to uncaught
exception 'NSGenericException', reason: 'Unable to install constraint on view.
Does the constraint reference something from outside the subtree of the view?
That's illegal. constraint:<NSLayoutConstraint:0x6b6ebf0 UILabel:0x6b68e40.centerX
== UIView:0x6b64a00.centerX> view:<UILabel: 0x6b68e40; frame = (0 0; 0 0); text =
'View 1'; clipsToBounds = YES; userInteractionEnabled = NO; layer = <CALayer:
0x6b67220>>'
libc++abi.dylib: terminate called throwing an exception
```

下面是两条简单的原则:

- 创建约束规则的时候, 如果规则所涉及的某个视图是另一个视图的上级视图, 那么就把规则添加到这个上级视图里。
- 如果用格式化字符串来创建约束规则, 而字符串里面又包含表示上级视图的竖线, 那么就把规则添加到对应的上级视图里。

#### 解决方案 5-3 用约束规则来实现视图的居中对齐

```
@implementation UIView (ConstraintHelper)
- (void)centerHorizontallyInSuperview
{
    if (!self.superview) return;

    [self.superview addConstraint:[NSLayoutConstraint
        constraintWithItem:self attribute:NSLayoutAttributeCenterX
        relatedBy:NSLayoutRelationEqual
        toItem:self.superview attribute:NSLayoutAttributeCenterX
        multiplier:1.0f constant:0.0f]];
}

- (void)centerVerticallyInSuperview
{
    if (!self.superview) return;

    [self.superview addConstraint:[NSLayoutConstraint
        constraintWithItem:self attribute:NSLayoutAttributeCenterY
        relatedBy:NSLayoutRelationEqual
        toItem:self.superview attribute:NSLayoutAttributeCenterY
        multiplier:1.0f constant:0.0f]];
}
@end
```

## 5.15 解决方案: 设定宽高比

在约束规则中使用放大倍数, 也就是等式  $y = m * x + b$  中的  $m$ , 即可设置视图的宽高比。



解决方案 5-4 演示了如何通过  $m$  值来设定高宽比，并将视图高度 ( $y$ ) 与宽度 ( $x$ ) 联系起来。解决方案 5-4 在视图的宽度与高度之间构建了 `NSLayoutRelationEqual` 关系，并把宽高比用作放大倍数。

为了切换宽高比，这条解决方案会把设置好的约束规则保存到 `NSLayoutConstraint` 型的 `aspectConstraint` 变量里。用户每次把比例从 16:9 切换到 4:3 或是切换回去时，这个解决方案都会将前一条约束规则从视图里移除，然后创建新规则并将其保存起来。创建新规则的时候要设置适当的放大倍数，创建好之后，还要将其添加到视图里面。

我们既想令视图的宽度和高度可以灵活变化，又想使其保持一定的尺寸，所以，这条解决方案的 `createLabel` 方法需要做两件事。首先，要用谓词来限定宽度及高度，它请求系统把宽度和高度都至少设为 300 个点。其次，要给这一请求指定优先级。我们把优先级设定得比较高 (750)，但是并没有设定成非满足不可 (1000)，这样就给约束系统留下了继续调整布局的余地。做完这两件事之后，我们就实现了一套能够实时调整宽高比的程序界面，而且程序还可以在运行的时候动态改变其布局。

为了令代码读起来容易一些，我们把创建约束规则的语句直接写在了解决方案 5-4 里，实际上，只需稍加修改，就可以把限定宽高比的功能添加到名为 `ConstraintHelper` 的 `category` 中。

限定宽高比的约束规则也可以用来保持某张图像的横纵比例。视图的 `contentMode` 可能无法完全准确地设定图像的横纵比。我们可以通过 `UIImage` 的 `size` 属性得知图像尺寸，然后用其宽度除以高度，并据此创建出符合横纵比的约束规则来。

#### 解决方案 5-4 创建限定宽高比的约束规则

```
- (UILabel *)createLabelWithTitle:(NSString *)title onParent:(UIView *)parentView
{
    UILabel *label = [[UILabel alloc] init];
    label.textAlignment = NSTextAlignmentCenter;
    label.text = title;
    label.backgroundColor = [UIColor greenColor];
    [parentView addSubview:label];

    // Turn off automatic translation of autoresizing masks into constraints
    label.translatesAutoresizingMaskIntoConstraints = NO;

    // Add constraints
    [label constrainWithinSuperviewBounds];
    [label addConstraints:[NSLayoutConstraint
        constraintsWithVisualFormat:@"H:[label](>=theWidth@750)"
        options:0 metrics:@{@"theWidth":@300.0}
        views:NSDictionaryOfVariableBindings(label)]];
    [label addConstraints:[NSLayoutConstraint
        constraintsWithVisualFormat:@"V:[label](>=theHeight@750)"
        options:0 metrics:@{@"theHeight":@300.0}
        views:NSDictionaryOfVariableBindings(label)]];
    [label centerInSuperview];
}
```

```

        return label;
    }

    - (void)toggleAspectRatio
    {
        if (aspectConstraint)
            [self.view removeConstraint:aspectConstraint];

        if (useFourToThree)
            aspectConstraint = [NSLayoutConstraint
            constraintWithItem:view1
            attribute:NSLayoutAttributeWidth
            relatedBy:NSLayoutRelationEqual toItem:view1
            attribute:NSLayoutAttributeHeight
            multiplier:(4.0f / 3.0f) constant:0.0f];

        else
            aspectConstraint = [NSLayoutConstraint
            constraintWithItem:view1
            attribute:NSLayoutAttributeWidth
            relatedBy:NSLayoutRelationEqual toItem:view1
            attribute:NSLayoutAttributeHeight
            multiplier:(16.0f / 9.0f) constant:0.0f];

        [self.view addConstraint:aspectConstraint];
        useFourToThree = !useFourToThree;
    }
}

```

## 5.16 解决方案：响应屏幕方向的变更

设备屏幕的几何特征也会影响界面的布局方式。比方说，当设备处在横屏状态的时候，垂直方向上就没有太多的空间可以排布内容。请看图 5-10。竖屏状态下，iTunes 专辑封面会出现在屏幕上方，然后是专辑名称及歌手名称，最下面是写有专辑价格的 Buy（购买）按钮。而在横屏状态下，专辑封面则位于屏幕左方，专辑名称、歌手名称及 Buy 按钮排列于右下角。

为了适应这两种布局，我们必须根据屏幕方向的变更情况来修改约束规则，并重新排版。解决方案 5-5 中的 `updateViewControllerConstraints` 方法能够依照当前的屏幕方向来刷新约束规则，它会把已有的规则全部移除，并设置新的约束。我们应该在 `willAnimateRotationToInterfaceOrientation:duration:` 里调用此方法。这样所产生的效果就可以和界面里的其他动画效果平滑地融合起来。另外要注意，系统是在把视图控制器的 `interfaceOrientation` 属性设置好之后，再去调用 `willAnimateRotationToInterfaceOrientation:` 的，而不是像 `willRotateToInterfaceOrientation:duration:` 方法那样在该属性变更之前回调。这样的话，`updateViewControllerConstraints` 方法所查询到的界面方向本身就已经是正确的界面方向了，我们可以据此生成适当的约束规则。

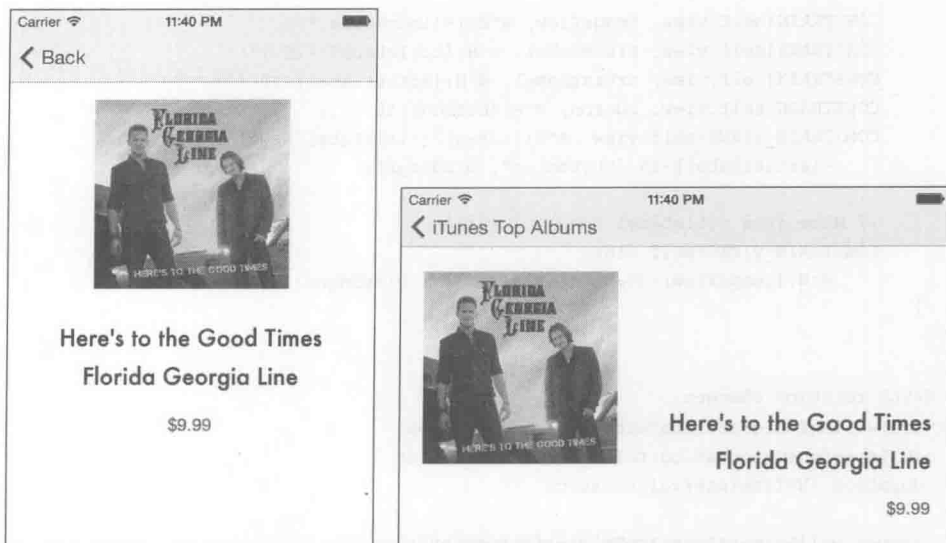


图 5-10 同一份内容的垂直排版和水平排版

解决方案 5-5 使用了几个与约束规则有关的宏，笔者将在本章末尾详细解释它们。

#### 解决方案 5-5 根据屏幕方向来更新视图的约束规则

```
(void)updateViewControllerConstraints
{
    [self.view removeConstraints:self.view.constraints];

    NSDictionary *bindings = NSDictionaryOfVariableBindings(
        imageView, titleLabel, artistLabel, button);

    if (IS_IPAD ||
        UIDeviceOrientationIsPortrait(self.interfaceOrientation) ||
        (self.interfaceOrientation == UIDeviceOrientationUnknown))
    {
        for (UIView *view in @[imageView,
                               titleLabel, artistLabel, button])
        {
            CENTER_VIEW_H(self.view, view);
        }
        CONSTRAIN_VIEWS(self.view, @"V:[-80-[imageView]-30-\
            [titleLabel(>=0)]-[artistLabel]-15-[button]-(>=0)]-",
            bindings);
    }
    else
    {
        // Center image view on left
        CENTER_VIEW_V(self.view, imageView);

        // Lay out remaining views
```

```

        CONSTRAINT(self.view, imageView, @"H:|--[imageView]");
        CONSTRAINT(self.view, titleLabel, @"H:[titleLabel]-15-|");
        CONSTRAINT(self.view, artistLabel, @"H:[artistLabel]-15-|");
        CONSTRAINT(self.view, button, @"H:[button]-15-|");
        CONSTRAINT_VIEWS(self.view, @"V:|-(>=0)-[titleLabel(>=0)]\
            -[artistLabel]-15-[button]-|", bindings);

        // Make sure titleLabel doesn't overlap
        CONSTRAINT_VIEWS(self.view,
            @"H:[imageView]-(>=0)-[titleLabel]", bindings);
    }
}

// Catch rotation changes
- (void)willAnimateRotationToInterfaceOrientation:
    (UIInterfaceOrientation)toInterfaceOrientation
    duration:(NSTimeInterval)duration
{
    [super willAnimateRotationToInterfaceOrientation:
        toInterfaceOrientation duration:duration];
    [self updateViewControllerConstraints];
}

```

## 5.17 调试约束规则

用代码向程序中添加约束规则时，最常出现的问题就是布局有歧义和布局无法满足。我们需要花大量时间来查看 Xcode 的调试控制台，而且会遇到很多以 Unable to simultaneously satisfy constraints (无法同时满足约束规则) 开头的信息。

iOS 在运行程序时会尽量告诉开发者哪些约束规则无法满足、哪些约束规则必须放弃。通常它会列出多条规则，而开发者可以检查这些规则，以确定问题所在。这种信息一般会像下面这样：

```

Probably at least one of the constraints in the following list is one you don't
want. Try this: (1) look at each constraint and try to figure out which you don't
expect; (2) find the code that added the unwanted constraint or constraints and
fix it. (Note: If you're seeing NSAutoresizingMaskConstraints that you don't
understand, refer to the documentation for the UIView property
translatesAutoresizingMaskIntoConstraints)
(
    "<NSAutoresizingMaskLayoutConstraint:0x6e5bc90 h=-&- v=-&-
    UILabelContainerView:0x6e540f0.height == UIWindow:0x6e528a0.height>",
    "<NSAutoresizingMaskLayoutConstraint:0x6e5a5e0 h=-&- v=-&-
    UINavigationControllerTransitionView:0x6e55650.height ==
    UILabelContainerView:0x6e540f0.height>",
    "<NSAutoresizingMaskLayoutConstraint:0x6e592f0 h=-&- v=-&-
    UINavigationControllerWrapperView:0x6bb90d0.height ==
    UINavigationControllerTransitionView:0x6e55650.height - 64>",

```

```

    "<NSAutoresizingMaskLayoutConstraint:0x6e57b90 h=-&- v=-&-
    UIView:0x6baef20.height == UIViewControllerWrapperView:0x6bb90d0.height>",
    "<NSAutoresizingMaskLayoutConstraint:0x6e5cd40 h=-&- v=-&-
    V:[UIWindow:0x6e528a0(480)]>",
    "<NSAutoresizingMaskLayoutConstraint:0x6bbe890 h=-&- v=-&-
    UILabel:0x6bb3730.midY ==>",
    "<NSLayoutConstraint:0x6bb8cc0 UILabel:0x6bb3730.centerY ==
    UIView:0x6baef20.centerY>"
)

```

假如碰到了这种消息，则可能说明某个视图的 translatesAutoresizingMaskIntoConstraints 没有关闭。若是看到其中有 NSLayoutConstraint，而且它还和待排版的某个控件（比方说本例中的 UILabel）有关，则问题很有可能出现在这个地方。在上面这段消息中，笔者把有问题的那两条规则加粗了。

还有些情况，可能是两条必须满足的规则之间发生了冲突，也就是说，两者所描述的布局方式互相矛盾了。从上面这段消息可以看出，同一个视图既要居中对齐，又要左对齐。为了解决这一矛盾，布局系统必须在二者之间做出选择。它决定把“在 Y 轴方向居中”这一需求舍弃掉，而将视图的顶端与上级视图的顶端相对齐：

```

Will attempt to recover by breaking constraint
<NSLayoutConstraint:0x6e94250 UILabel:0x6b90e00.centerY ==
UIView:0x6b8ca50.centerY>

```

分析这些与约束规则相关的调试信息是件非常麻烦的事，不过只要掌握了某些技巧，还是可以避开一些问题的。首先，在用代码编写约束规则的时候一次不要写太多，这样就可以及时发现有错误的地方。其次，可以考虑使用程序清单 5-1 中的宏，我们在下一节里就会给出这些宏。不要在代码里写入太多“与上级视图对齐”或“将尺寸设为  $n \times m$ ”之类的规则，这样会让源文件变得很乱。最后要注意，如果没有遇到必须用代码来设置约束规则的情况，那么可以考虑通过 IB 所提供的工具来设计程序的布局，这样会更容易些。

## 5.18 解决方案：描述约束规则

在编写与调试约束规则的时候，你也许会觉得，如果能把任意的 NSLayoutConstraint 对象转换成一段容易读懂的描述信息，那就方便多了。解决方案 5-6 可以构建出这样一种字符串，用来精确地描述出约束规则，其内容如下所示：

- (1000) [UILabel:6bb32a0].right <= [self].right
- (750) [self].width == ([self].height \* 1.778)
- (750) [UILabel:6bb32a0].leading == ([UILabel:6ed2e70].trailing + 60.000)

这条解决方案会把 NSLayoutConstraint 实例以文本形式表示出来。它会从该规则所涉视图的角度来完成这件事（因此其中也会包括对视图本身及其上级视图的引用，另外还包括以 [类名：内存地址] 这种形式所列出来的特定子视图）。

请注意，并非每条约束规则都包含两个视图。某些约束规则可能只会引用视图本身（比如上述第二个例子就是这样，它把自己的宽度设为其高度的倍数）。在这些情况下，item2 属性总是 nil。

#### 解决方案 5-6 描述约束规则

```
@implementation UIView (ConstraintHelper)
```

```
// Return a string that describes an attribute
```

```
- (NSString *)nameForLayoutAttribute:(NSLayoutAttribute)anAttribute
```

```
{
```

```
    switch (anAttribute)
```

```
    {
```

```
        case NSLayoutAttributeLeft: return @"left";
```

```
        case NSLayoutAttributeRight: return @"right";
```

```
        case NSLayoutAttributeTop: return @"top";
```

```
        case NSLayoutAttributeBottom: return @"bottom";
```

```
        case NSLayoutAttributeLeading: return @"leading";
```

```
        case NSLayoutAttributeTrailing: return @"trailing";
```

```
        case NSLayoutAttributeWidth: return @"width";
```

```
        case NSLayoutAttributeHeight: return @"height";
```

```
        case NSLayoutAttributeCenterX: return @"centerX";
```

```
        case NSLayoutAttributeCenterY: return @"centerY";
```

```
        case NSLayoutAttributeBaseline: return @"baseline";
```

```
        case NSLayoutAttributeNotAnAttribute: return @"not-an-attribute";
```

```
        default: return @"unknown-attribute";
```

```
    }
```

```
}
```

```
// Return a name that describes a layout relation
```

```
- (NSString *)nameForLayoutRelation:(NSLayoutRelation)aRelation
```

```
{
```

```
    switch (aRelation)
```

```
    {
```

```
        case NSLayoutRelationLessThanOrEqualTo: return @"<=";
```

```
        case NSLayoutRelationEqual: return @"=";
```

```
        case NSLayoutRelationGreaterThanEqualTo: return @">=";
```

```
        default: return @"unknown-relation";
```

```
    }
```

```
}
```

```
// Describe a view in its own context
```

```
- (NSString *)nameForItem:(id)anItem
```

```
{
```

```
    if (!anItem) return @"nil";
```

```
    if (anItem == self) return @"[self]";
```

```
    if (anItem == self.superview) return @"[superview]";
```

```
    return [NSString stringWithFormat:@"%[@:%d]", [anItem class], (int) anItem];
```

```
}
```

```
// Transform the constraint into a string representation
```

```

- (NSString *)constraintRepresentation:(NSLayoutConstraint *)aConstraint
{
    NSString *item1 = [self nameForItem:aConstraint.firstItem];
    NSString *item2 = [self nameForItem:aConstraint.secondItem];
    NSString *relation =
        [self nameForLayoutRelation:aConstraint.relation];
    NSString *attr1 =
        [self nameForLayoutAttribute:aConstraint.firstAttribute];
    NSString *attr2 =
        [self nameForLayoutAttribute:aConstraint.secondAttribute];

    NSString *result;

    if (!aConstraint.secondItem)
    {
        result = [NSString stringWithFormat:@"%4.0f) %@.%% %@ %0.3f",
            aConstraint.priority, item1, attr1,
            relation, aConstraint.constant];
    }
    else if (aConstraint.multiplier == 1.0f)
    {
        if (aConstraint.constant == 0.0f)
            result = [NSString stringWithFormat:@"%4.0f) %@.%% %@ %@.%%",
                aConstraint.priority, item1, attr1,
                relation, item2, attr2];
        else
            result = [NSString stringWithFormat:
                @"%4.0f) %@.%% %@ (%@.%% + %0.3f)",
                aConstraint.priority, item1, attr1, relation,
                item2, attr2, aConstraint.constant];
    }
    else
    {
        if (aConstraint.constant == 0.0f)
            result = [NSString stringWithFormat:
                @"%4.0f) %@.%% %@ (%@.%% * %0.3f)",
                aConstraint.priority, item1, attr1, relation,
                item2, attr2, aConstraint.multiplier];
        else
            result = [NSString stringWithFormat:
                @"%4.0f) %@.%% %@ ((%@.%% * %0.3f) + %0.3f)",
                aConstraint.priority, item1, attr1, relation,
                item2, attr2, aConstraint.multiplier,
                aConstraint.constant];
    }

    return result;
}

- (void)showConstraints

```

```

{
    NSString *viewName = [NSString stringWithFormat:
        @"[%@:%d]", [self class], (int) self];
    NSLog(@"View %@ has %d constraints",
        viewName, self.constraints.count);
    for (NSLayoutConstraint *constraint in self.constraints)
        NSLog(@"%@", [self constraintRepresentation:constraint]);
    printf("\n");
}

@end

```

## 5.19 用宏来创建约束规则

用约束规则来排布控件的位置是相当可靠的。不过，就其本身来说，它们非常繁琐而且特别冗长。开发者要一次又一次地去编写很难读懂的方法调用语句。

约束规则调试起来也特别麻烦。一个简单的拼写错误就会耗费很多时间，而且许多应用程序所使用的约束规则都是一样的。如果可以预先定义一些宏，那么就能把排布视图所用的代码写得更易读懂且更加可靠。假如要把某视图与另外一个视图居中对齐，那么直接使用名为 `CENTER_VIEW` 的宏就好，而不用每次都编写约束规则并调试它们。

创建好程序清单 5-1 中的这些宏之后，我们就可以把工作重心从编写精确的约束规则定义转移为令每个视图的约束规则都协调且完备。在排布视图的时候，开发者经常需要关注这两件事情。

### 宏

程序清单 5-1 列出了一套比较丰富的宏定义。笔者已经对其进行了测试，并在本书很多解决方案里面都使用了它们。这些宏以一种相当简单的方式来产生约束规则。请注意，它们并不把创建出来的 `NSLayoutConstraint` 对象返回给调用者，而是将这些约束规则添加到适当的视图里。如果需要获取这些新创建好的约束规则，以便稍后移除，那么可以自己添加一些能返回相关约束规则的宏，或采用解决方案 5-1 中的功能来搜寻并移除 `NSLayoutConstraint`。

由于篇幅所限，程序清单 5-1 中没有包含另外一套宏，也就是那种可以接受常量并创建出约束规则的宏，有时候它们用起来也很方便。比方说，在将某视图同上级视图对齐的时候，那种宏就显得特别有用。读者应该很容易就能写出这种宏，并将其添加到程序库里。在编写创建约束规则的宏时，我们需要在复杂度和灵活度之间做出权衡，并按照需求进行调整。

假如不喜欢宏这种形式，那么也可以改为编写函数库或是针对 `UIView` 类来创建与约束规则有关的 `category`。你应该选择一种最符合自己代码风格或布局需求的方式，然后逐步构建并完善这个工具库。



程序清单 5-1 用宏来创建约束规则

```

// Prepare Constraint Compliance
#define PREPCONSTRAINTS(VIEW) \
    [VIEW setTranslatesAutoresizingMaskIntoConstraints:NO]

// Add a visual format constraint
#define CONSTRAIN(PARENT, VIEW, FORMAT) \
    [PARENT addConstraints:[NSLayoutConstraint \
        constraintsWithVisualFormat:(FORMAT) options:0 metrics:nil \
        views:NSDictionaryOfVariableBindings(VIEW)]]
#define CONSTRAIN_VIEWS(PARENT, FORMAT, BINDINGS) \
    [PARENT addConstraints:[NSLayoutConstraint \
        constraintsWithVisualFormat:(FORMAT) options:0 metrics:nil \
        views:BINDINGS]]

// Stretch across axes
#define STRETCH_VIEW_H(PARENT, VIEW) \
    CONSTRAIN(PARENT, VIEW, @"H:[\"#VIEW\"(>=0)]|")
#define STRETCH_VIEW_V(PARENT, VIEW) \
    CONSTRAIN(PARENT, VIEW, @"V:[\"#VIEW\"(>=0)]|")
#define STRETCH_VIEW(PARENT, VIEW) \
    {STRETCH_VIEW_H(PARENT, VIEW); STRETCH_VIEW_V(PARENT, VIEW);}

// Center along axes
#define CENTER_VIEW_H(PARENT, VIEW) \
    [PARENT addConstraint:[NSLayoutConstraint \
        constraintWithItem:VIEW attribute:NSLayoutAttributeCenterX \
        relatedBy:NSLayoutRelationEqual \
        toItem:PARENT attribute:NSLayoutAttributeCenterX \
        multiplier:1.0f constant:0.0f]]
#define CENTER_VIEW_V(PARENT, VIEW) \
    [PARENT addConstraint:[NSLayoutConstraint \
        constraintWithItem:VIEW attribute:NSLayoutAttributeCenterY \
        relatedBy:NSLayoutRelationEqual \
        toItem:PARENT attribute:NSLayoutAttributeCenterY \
        multiplier:1.0f constant:0.0f]]
#define CENTER_VIEW(PARENT, VIEW) \
    {CENTER_VIEW_H(PARENT, VIEW); CENTER_VIEW_V(PARENT, VIEW);}

// Align to parent
#define ALIGN_VIEW_LEFT(PARENT, VIEW) \
    [PARENT addConstraint:[NSLayoutConstraint \
        constraintWithItem:VIEW attribute:NSLayoutAttributeLeft \
        relatedBy:NSLayoutRelationEqual \
        toItem:PARENT attribute:NSLayoutAttributeLeft \
        multiplier:1.0f constant:0.0f]]
#define ALIGN_VIEW_RIGHT(PARENT, VIEW) \
    [PARENT addConstraint:[NSLayoutConstraint \
        constraintWithItem:VIEW attribute:NSLayoutAttributeRight \
        relatedBy:NSLayoutRelationEqual \

```

```

        toItem:PARENT attribute:NSLayoutAttributeRight \
        multiplier:1.0f constant:0.0f]]
#define ALIGN_VIEW_TOP(PARENT, VIEW)
    [PARENT addConstraint:[NSLayoutConstraint \
        constraintWithItem:VIEW attribute:NSLayoutAttributeTop \
        relatedBy:NSLayoutRelationEqual \
        toItem:PARENT attribute:NSLayoutAttributeTop \
        multiplier:1.0f constant:0.0f]]
#define ALIGN_VIEW_BOTTOM(PARENT, VIEW) \
    [PARENT addConstraint:[NSLayoutConstraint \
        constraintWithItem:VIEW attribute:NSLayoutAttributeBottom \
        relatedBy:NSLayoutRelationEqual \
        toItem:PARENT attribute:NSLayoutAttributeBottom \
        multiplier:1.0f constant:0.0f]]

// Set Size
#define CONSTRAIN_WIDTH(VIEW, WIDTH) \
    [VIEW addConstraint:[NSLayoutConstraint constraintWithItem:VIEW \
        attribute:NSLayoutAttributeWidth \
        relatedBy:NSLayoutRelationEqual toItem:nil \
        attribute:NSLayoutAttributeNotAnAttribute \
        multiplier:1.0f constant:WIDTH]];
#define CONSTRAIN_HEIGHT(VIEW, HEIGHT) \
    [VIEW addConstraint:[NSLayoutConstraint constraintWithItem:VIEW \
        attribute:NSLayoutAttributeHeight \
        relatedBy:NSLayoutRelationEqual toItem:nil \
        attribute:NSLayoutAttributeNotAnAttribute \
        multiplier:1.0f constant:HEIGHT]];

#define CONSTRAIN_SIZE(VIEW, HEIGHT, WIDTH) \
    {CONSTRAIN_WIDTH(VIEW, WIDTH); CONSTRAIN_HEIGHT(VIEW, HEIGHT);}

// Set Aspect
#define CONSTRAIN_ASPECT(VIEW, ASPECT) \
    [VIEW addConstraint:[NSLayoutConstraint \
        constraintWithItem:VIEW attribute:NSLayoutAttributeWidth \
        relatedBy:NSLayoutRelationEqual \
        toItem:VIEW attribute:NSLayoutAttributeHeight \
        multiplier:(ASPECT) constant:0.0f]]

// Item ordering
#define CONSTRAIN_ORDER_H(PARENT, VIEW1, VIEW2) \
    [PARENT addConstraints:[NSLayoutConstraint \
        constraintsWithVisualFormat:@"H:[\"#VIEW1\"]->=0-[\"#VIEW2\"]" \
        options:0 metrics:nil \
        views:NSDictionaryOfVariableBindings(VIEW1, VIEW2)]]
#define CONSTRAIN_ORDER_V(PARENT, VIEW1, VIEW2) \
    [PARENT addConstraints:[NSLayoutConstraint \
        constraintsWithVisualFormat:@"V:[\"#VIEW1\"]->=0-[\"#VIEW2\"]" \
        options:0 metrics:nil \
        views:NSDictionaryOfVariableBindings(VIEW1, VIEW2)]]

```

## 5.20 小结

本章介绍了 iOS 的 Auto Layout 特性。在继续学习下一章之前，先回顾以下内容：

- 原来我们可能要从 struts and springs 以及尺寸的灵活变化等角度来考虑视图布局，但现在苹果公司推出了 Auto Layout 系统，它能够提供更好的控制手段、更强大的功能以及更灵活的工具。
- IB 本身就有一套出色的布局工具。但是，用代码来实现基于约束规则的界面也是相当可行并易于使用的。无论是用 IB 还是用程序码来指定约束规则，布局系统都能非常精准地控制视图。
- 采用约束规则来指定视图布局有个很大的好处，就是不用再针对每一种分辨率去设计一套界面了。虽说平板电脑的操作方式与 iPhone 系列的手机可能会大不相同，即使同样是手机，其视窗大小和屏幕大小也不一样，而有了这套新工具之后，我们就可以设计出一套能够自动适应这些手机的界面来了。这些约束规则虽然看上去很简单，但是其背后却有着非常灵活而且强大的地方。
- 在设计界面的过程中，可以把阴影等视觉装饰物考虑进来。无论给框架添加多少个辅助视图，我们总能通过对齐矩形来确保用户界面的布局是正确的。
- 可视化的格式字符串适合用来设计通用的视图布局，而视图之间的关系则适合用来指定具体的细节。这两种方式都非常重要，在设计界面的时候都不应忽视。

## 文本输入

有些人可能认为触摸设备上面的文本输入没什么大不了的，因为用户已经可以通过简单的手势来表达非常多的信息了。不过，文本输入仍然是个非常关键的功能，尤其是当用户从办公室或家中拿着手机出门操作时，就显得更为重要了。很多情况下，用户都需要输入并阅读屏幕上的文字。用户可以通过输入文本来登录账号、查看并回复电子邮件、输入 URL 并浏览网页等。苹果公司的键盘有非常智能的预测能力，这使得输入文本的过程简单而流畅。相关的类和框架也为开发者提供了强大的手段，用以展示并操作程序中的文本。

iOS 7 可以说成是一次与文本有关的更新。苹果公司去掉了很多复杂的图案、用户界面装饰物（UI chrome）以及阴影效果，而把注意力放在了内容上面。在很多情况下，这个内容指的就是文本。iOS 7 对文本布局和渲染引擎做了有史以来最为显著的一次更新，它与新系统的设计重点以及新的 Text Kit 技术一起，展示出文本在 iOS 生态圈中的重要地位。

Text Kit 为文本系统的展示与输入等方面带来了重大改变。以前若想调整字间距及行间距等特性，必须深入 Core Text 的底层细节才行，而现在，我们则可以用 Text Kit 来完全控制文本的渲染。UIKit 的文本及文本输入控件都构建在 Text Kit 上面，而 Text Kit 又构建在 Core Text 上面。

因受范围所限，本书不会完整地讲述由 Text Kit 所提供的功能，不过，这些功能很强大而且很灵活，并且非常受开发者欢迎，所以，还是值得去研究一下的。

本章要介绍的各条解决方案能够解决与文本有关的许多问题。你将学到怎样控制键盘、怎样使屏幕上的控件具备文本输入能力、如何扫描文本、格式化文本及编辑文本。开发者在实现文本输入功能时会经常遇到此类问题，而这一章所提供的解决方案可以非常方便地解决它们。

## 6.1 解决方案：隐藏 UITextField 的键盘

对于小设备及 UITextField 控件来说，有个常见的问题就是“如何在用户输入完之后把键盘关掉”。iOS 系统并没有内置这样一种方式来自动侦测用户是否已经输入完毕并做出响应。不过，当用户编辑完 UITextField 的内容之后，按理说确实应该把键盘关掉才对。iPad 提供了关闭键盘的按钮，但 iPhone 和 iPod touch 则没有。

幸好我们只需稍微编写一些代码，即可在用户编辑完文本框之后做出响应了，这种解决方案不受平台限制。具体做法是提供 Done 按钮供用户点击，然后令文本框放弃第一响应者的身份。正如解决方案 6-1 所示，只要放弃了第一响应者的身份，键盘就不会出现在屏幕里了。实现本方式时，有几个关键事项要注意：

- 把 Return 键的类型设为 `UIReturnKeyDone`，以便将文本由 *Return* 改为 *Done*。我们可以在 Interface Builder (IB) 的 Attributes Inspector 里面设置，也可以对文本框的 `returnKeyType` 属性赋值。把 Return 键的文本改为 Done，可以使用户明白：在编辑完之后，只需按下这个键，就能结束编辑。如果不这么做的话，那么除非用户曾在非移动平台的操作系统 (nonmobile system) 里以类似方式操作过，否则很难知道如何结束编辑状态。图 6-1 演示了带有 Done 键的键盘。

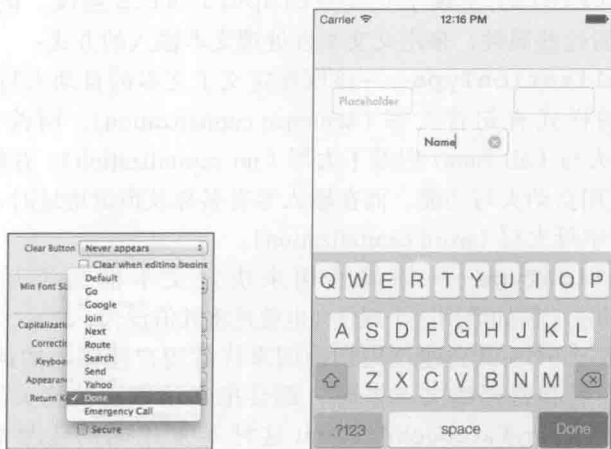



图 6-1 把 Return 键的名称设为 Done (如左侧截图所示)，可以令用户明白如何结束对文本框的编辑。我们可以用代码或 IB 的 Attributes Inspector 来调整与文本框有关的 Return 键，并定制文本框的样貌及行为

- 令视图控制器成为文本框的 **delegate**。你可以用代码把文本框的 `delegate` 属性设为视图控制器，也可以在 IB 里面右击文本框，并进行配置。请确认视图控制器已经遵循并实现了 `UITextFieldDelegate` 协议。
- 实现 `textFieldShouldReturn:` 方法。无论 Return 键叫什么名字，该方法都会把用户对这个按键的触摸捕获下来。我们在该方法里调用 `resignFirstResponder`,

以便把键盘隐藏起来，直到用户下次点击 `UITextField` 或 `UITextView` 的时候，键盘才会再度出现。

---

 **提示** 当用户按下 `Return` 键时，除了可以在 `textFieldShouldReturn:` 方法里隐藏键盘，还可以于该方法中执行其他操作。

---

开发者必须在代码中处理好上述三个问题，才能为 `UITextField` 实例创建出平滑的交互过程。

### 6.1.1 阻止系统把键盘隐藏起来

我们既可以把键盘隐藏起来，也可以通过代码阻止这种行为。如果当前的响应者并不支持文本，那么视图控制器可以强令键盘留在屏幕上。覆写 `disablesAutomaticKeyboardDismissal` 方法即可实现此功能。该方法所返回的布尔值表示程序是否允许系统将键盘隐藏起来。

### 6.1.2 `UITextInputTraits` 协议中的属性

文本框 (`UITextField`) 实现了 `UITextInputTraits` 协议。该协议提供了八项属性，我们可以通过设置这些属性，来定义文本框处理文本输入的方式：

- **`autocapitalizationType`**——该属性定义了文本的自动大写 (`autocapitalization`) 风格。可选的样式有句首大写 (`sentence capitalization`)、词首大写 (`word capitalization`)、全部大写 (`all caps`) 以及不大写 (`no capitalization`)。在输入用户名和密码的时候，不要使用自动大写功能。而在输入专有名称及街道地址时，则可将自动大写风格设为单词首字母大写 (`word capitalization`)。
- **`autocorrectionType`**——该属性用来决定文本框是否开启 iOS 的自动修正 (`autocorrect`) 功能。假如启用这个属性 (也就是将其值设为 `UITextAutocorrectionTypeYes`)，那么 iOS 就会建议用别的词来代替用户所输入的词。大多数开发者在设计输入用户名和密码的文本框时，都会把自动修正功能关闭，这样一来，iOS 就不会无意间把 `myFacebookAccount` 这种本来正确的账号名替换成 `myofacial count` 了。
- **`spellCheckingType`**——该属性决定了 iOS 系统是否会在用户输入文本时进行拼写检查。将其设为 `UITextSpellCheckingTypeYes`，即可启用它，而把它设为 `UITextSpellCheckingTypeNo`，则可以禁用它。拼写检查与自动修正不同，自动修正会在用户输入的过程中直接替换文本，而拼写检查则会在文本输入界面中检测用户可能拼错的词，并将其用下划线标出来，以此提示用户这些词可以换成正确的词。如果开启了自动修正功能，那么系统也会默认启用拼写检查。
- **`keyboardAppearance`**——我们可以通过该属性在两种键盘显示风格之间切换：一种是浅色风格 (默认)，另一种是深色风格。

- **keyboardType**——当用户操作文本框或文本视图时，这个属性决定了键盘的类型。

iOS 提供了十几个选项，这些类型包括：标准 ASCII、数字与标点、输入 PIN 的数字键盘（0 至 9）、输入电话号码的键盘（0 至 9、#、\*）、小数输入键盘（0 至 9、.）、为输入 URL 而优化的键盘（会凸显 .、/ 及 .com 键）、为电子邮件而优化的键盘（会凸显 @ 和 . 键）以及为 Twitter 而优化的键盘（会凸显 @ 及 # 键）。

每种键盘都排布了不同的按键，它们各有优缺点。比方说，电子邮件专用的键盘就很适合用来输入电子邮箱地址。它里面包含了 @ 符号，也能够输入其他文本。而 Twitter 专用的键盘则把 # 键（hashtag，话题标签）和 @ 键（用户 ID 标识符）摆在了显著位置。

- **enablesReturnKeyAutomatically**——当文本框或文本视图中没有文本时，该属性用来决定系统是否会禁用 Return 键。如果将该属性设为 YES，那么只有当用户输入了至少一个字符之后，系统才会自动启用 Return 键。
- **returnKeyType**——用来指定键盘上 Return 键里的文本。你可以选择默认值（Return），也可以选择 Go、Google、Join、Next、Route、Search、Send、Yahoo、Done 及 Emergency Call 等。你应该根据用户输入完之后所要执行的操作来设置该属性。
- **secureTextEntry**——该属性用来开启文本遮掩功能，以实现更安全的文本输入。启用了此属性之后，用户只能看到最后输入的那个字符，而其余字符都将显示为圆点。如果某个文本框是用来输入密码的，那么应该开启此功能。

### 6.1.3 文本框的其他属性

除了 `UITextInputTraits` 协议里面定义的那些标准属性之外，文本框还提供了一些属性用以控制其样貌。下面列出几个开发者应该知晓的属性：

- 占位文本——图 6-2 演示了带有占位文本（placeholder text）的文本框。当文本框里没有内容的时候，系统会把这些浅灰色文本显示出来，它能够提示用户这个文本框里应该输入什么内容。开发者可以通过这种占位文本来提示用户在文本框中输入用户名或电子邮箱地址等内容，如图 6-2 所示。



图 6-2 当文本框里没有内容的时候，就会以浅灰色把占位文本显示出来。用户在文本框里输入的内容会盖住这些文本。你可以通过 IB 的 Attributes Inspector 来配置文本框，也可以编辑 `UITextField` 对象的 `placeholder` 属性

- 边框样式——开发者可以通过 `borderStyle` 属性来控制文本周围的边线风格。可以选择简单线条、Bezel 以及圆角矩形（如图 6-2 所示）。最好是用 IB 来调整这个风格，因为我们可以通过 IB 的 Attributes Inspector 在不同的风格之间切换。
- 清除按钮——文本输入区域的右侧可以有个 X 字样的清除按钮。通过 `clearButtonMode` 属性，可以指定该按钮是否出现以及何时出现。`UITextFieldView-`

ModeAlways 表示总是出现、UITextFieldViewModeNever 表示决不出现、UITextFieldViewWhileEditing 表示只在编辑时出现、UITextFieldViewModeUnlessEditing 表示只在用户不编辑文本的时候出现。开发者应该尽量使用户能够最大限度地操控应用程序才对。

#### 解决方案 6-1 用 Done 键来关闭文本框的键盘

---

```
// Dismiss the keyboard when the user taps Done
- (BOOL)textFieldShouldReturn:(UITextField *)textField
{
    [textField resignFirstResponder];
    return YES;
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    // Update all text fields, including those defined in IB,
    // setting delegate, return key type, and other useful traits
    for (UIView *view in self.view.subviews)
    {
        if ([view isKindOfClass:[UITextField class]])
        {
            UITextField *aTextField = (UITextField *)view;
            aTextField.delegate = self;

            aTextField.returnKeyType = UIReturnKeyDone;
            aTextField.clearButtonMode =
                UITextFieldViewModeWhileEditing;

            aTextField.borderStyle = UITextBorderStyleRoundedRect;
            aTextField.contentVerticalAlignment =
                UIControlContentVerticalAlignmentCenter;
            aTextField.autocorrectionType =
                UITextAutocorrectionTypeNo;

            aTextField.font =
                [UIFont fontWithName:@"Futura" size:12.0f];
            aTextField.placeholder = @"Placeholder";
        }
    }
}
```

---

#### 获取解决方案代码

访问 <https://github.com/erica/iOS-7-Cookbook> 网页，并打开“C06 Text Views”文件夹，即可找到与本章中的解决方案相对应的完整范例项目。

---



## 6.2 解决方案：把带有自定义辅助视图的键盘隐藏起来

开发者可以通过自定义的辅助视图（custom accessory view）来为屏幕上的键盘添加一些额外的内容。常见的用法包括添加自定义的按钮、添加其他可用来选取文本字体及文本颜色的控件，另外也可以实现一些文本间的导航方式。解决方案 6-2 添加了两个按钮，一个用来清除已经输入的文本，另一个用来隐藏键盘。图 6-3 演示了带有这些附加组件的键盘。

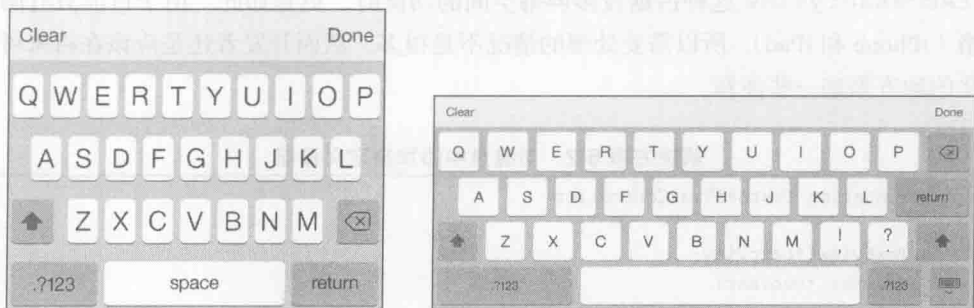


图 6-3 通过 `inputAccessoryView` 属性，可以在标准的 iOS 键盘上面添加自定义的视图元件。在本例中，我们给 iPhone 和 iPad 的键盘上面添加了两个按钮

每个附属视图都会与文本框或文本视图这样的 `Responder` 相关联（这个 `Responder` 继承自 `UIResponder` 类）。设置视图的 `inputAccessoryView` 属性即可为其添加辅助视图。解决方案 6-2 把一个简单的工具栏用作辅助视图，这样的话，只需编写极少的代码，就能实现一些附加功能。

在解决方案 6-1 中，用户可以通过 `Done` 键来隐藏文本框的键盘（所谓文本框，是指单行的文本输入控件），而本例也为用户提供了相同的功能，使其能够通过工具栏上的 `Done` 按钮来关闭文本视图的键盘（所谓文本视图，是指包含多行文字并且可以滚动的文本编辑视图）。两者的区别是：本例中，用户依然可以通过键盘上的 `Return` 键在文本中插入回车符，以便给文字分段。



本书的一位技术评审说他从来都记不住哪个是文本视图，哪个是文本框。对于这件事，笔者回复说：“A view is two; a field is sealed.”。意思就是说，文本视图可以包含多行（两行或两行以上）文本，而文本框只是个单行的文本输入控件，其文字周围画有某种风格的边框。

iOS 开发者 Phil Mills 说了个更有趣的口诀：“Take my text field...please.”。看到这句话，我们会联想到，文本框（Text Field）只能显示一行文字。

解决方案 6-2 的 `Done` 按钮会在其回调方法中放弃文本视图的第一响应者身份。由于 iPad 键盘会自动显示出一个用于隐藏键盘的按钮，所以 iPad 用户是用不到这个 `Done` 按钮的，不过放在这里也没什么坏处。如果既要编写通用的程序，又要在 iPad 上面隐藏 `Done` 按

钮，那么就请根据当前的用户界面风格（user interface idiom）来进行判断。下面这个宏提供了一种简便的判断方式，可以检测出程序是否运行在 iPad 上面：

```
#define IS_IPAD (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad)
```

请注意，苹果公司将来可能会推出一些规格不同的新 iOS 设备，有的也许屏幕会比较大，有的也许会比较小，所以开发者必须要针对那些设备编写相应的代码才行，在使用 `inputAccessoryView` 这种占据较多屏幕空间的功能时，更是如此。由于目前只有两套界面风格（iPhone 和 iPad），所以需要处理的情况不是很多，然而开发者还是应该在将来可能发生变化的地方添加一些注释。

#### 解决方案 6-2 向键盘中添加自定义按钮

---

```
@implementation TestBedViewController
{
    UITextView *textView;
    UIToolbar *toolBar;
}

// Remove text from text view
- (void)clearText
{
    [textView setText:@""];
}

// Dismiss keyboard by resigning first responder
- (void)leaveKeyboardMode
{
    [textView resignFirstResponder];
}

- (UIToolbar *)accessoryView
{
    // Create toolbar with Clear and Done
    toolBar = [[UIToolbar alloc] initWithFrame:
        CGRectMake(0.0f, 0.0f, self.view.frame.size.width, 44.0f)];
    toolBar.tintColor = [UIColor darkGrayColor];

    // Set up the items as Clear - flexspace - Done
    NSMutableArray *items = [NSMutableArray array];
    [items addObject:BARBUTTON(@"Clear", @selector(clearText))];
    [items addObject:SYSBARBUTTON(UIBarButtonSystemItemFlexibleSpace, nil)];
    [items addObject:BARBUTTON(@"Done", @selector(leaveKeyboardMode))];
    toolBar.items = items;

    return toolBar;
}

- (void)loadView
```

```

{
    self.view = [[UIView alloc] init];
    self.view.backgroundColor = [UIColor whiteColor];

    // Create text view and add the custom accessory view
    textView = [[UITextView alloc] initWithFrame:self.view.bounds];
    textView.font = [UIFont fontWithName:@"Georgia"
        size:(IS_IPAD) ? 24.0f : 14.0f];
    textView.inputAccessoryView = [self accessoryView];

    // Use constraints to fill application bounds
    [self.view addSubview:textView];
    PREPCONSTRAINTS(textView);
    STRETCH_VIEW(self.view, textView); }
@end

```

## 6.3 解决方案：根据键盘来调整文本视图

iOS 的键盘是相当大的，它占据了屏幕上面很大一部分空间。鉴于此，我们应该在屏幕上面有键盘的时候调整文本框及文本视图，不要令键盘把它们给遮住。图 6-4 演示了这个问题。

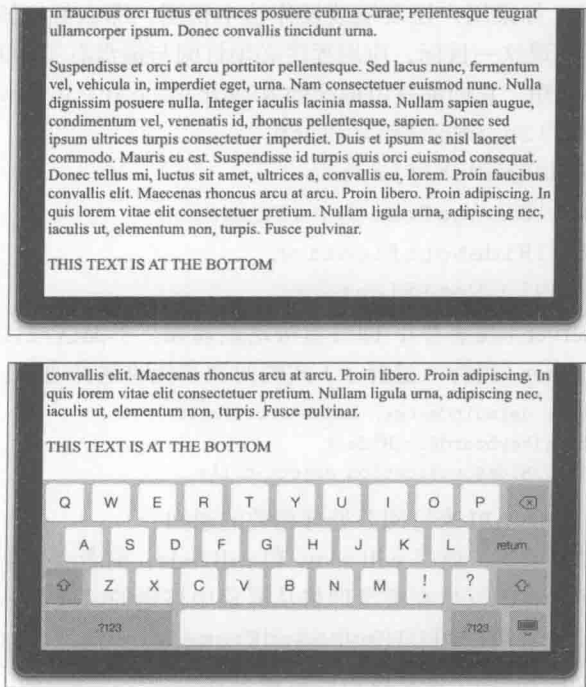


图 6-4 键盘占据了 iOS 设备很大一部分屏幕空间。当屏幕上有键盘的时候，如果我们既不调整视图的尺寸，也不将其上移，那么键盘就会把某些本来应该能看见的文本给遮住。在最后一张截图里，由于文本视图一直会延伸至屏幕底端，所以键盘会把屏幕下方的一部分文本挡住

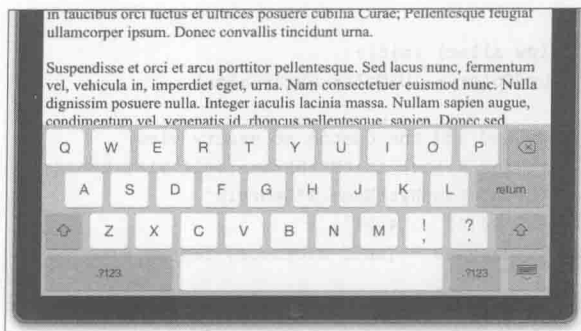


图 6-4 (续)

顶端截图演示了文本视图在具备第一响应者身份之前的样子。中间截图演示了用户所期望的效果，也就是说，即便屏幕上面有键盘，用户也应该能够看到并触摸整个文本视图。而底部截图则是既不调整视图尺寸，又不移动视图位置时的样子。这时，屏幕上约有三分之一的文本内容无法看到。用户看不到最后几行文本，而且也无法编辑它们。键盘把屏幕上最后几段文本盖住了，这使得用户不能触摸到那一个区域。

我们可以调整文本视图的大小，或移动其位置，使键盘不会把那么多文本挡住。当屏幕上显示出现键盘的时候，如果用户还要继续操作这个视图，那么就应该把视图与键盘分开，不要使两者重叠。为了实现这一目标，应用程序必须订阅与键盘有关的通知。

iOS 提供了许多种通知，它们都是由标准的 `NSNotificationCenter` 来散发的：

- `UIKeyboardWillShowNotification`
- `UIKeyboardDidShowNotification`
- `UIKeyboardWillChangeFrameNotification`
- `UIKeyboardWillHideNotification`
- `UIKeyboardDidHideNotification`

把对象注册为 `observer`（观察者），即可监听这些通知。下面这段代码能够监听 `will hide` 通知，并使用 `target-selector`（目标 - 选择子）式的回调来接收这种通知：

```
[[NSNotificationCenter defaultCenter] addObserver:self
 selector:@selector(keyboardWillHide:)
 name:UIKeyboardWillHideNotification object:nil];
```

你也可以通过基于块的 API 来订阅并处理相关的通知。

有两种通知经常需要监听，就是 `will show` 和 `will hide`，前者会发生在屏幕马上就要把键盘显示出来的时候，而后者则发生在键盘即将从屏幕中移走的时候。每个通知都提供了名为 `userInfo` 的字典，开发者可以用 `UIKeyboardFrameEndUserInfoKey` 作键来查询键盘最后的尺寸及位置（`frame`）。你不能直接访问键盘本身。

获取到键盘的 `frame` 之后，就可以根据屏幕上是否会显示键盘这一因素来调整文本视图。解决方案 6-3 向程序里添加了 `KeyboardSpacingView` 对象，它会适时地调整其约束规则，以适应键盘的高度。把这个 `KeyboardSpacingView` 添加到界面底部之后，它会监

听并处理与键盘有关的事件，并据此调整自身的尺寸。我们通过约束规则在文本视图与这个 KeyboardSpacingView 之间建立关系，这样的话，当 KeyboardSpacingView 的尺寸有变化时，文本视图也会调整其大小：

```
// Create a spacer
KeyboardSpacingView *spacer =
    [KeyboardSpacingView installToView:self.view];

// Place the spacer under the text view
CONSTRAIN(self.view, @"V:[textView][spacer]|",
    NSDictionaryOfVariableBindings(textView, spacer));
```

这样实现出来的文本视图能够完全适应不同的硬件，并且还能根据 inputAccessoryView 来调整自己的大小。

### 解决方案 6-3 创建专用的 KeyboardSpacingView 对象

```
@implementation KeyboardSpacingView
{
    NSLayoutConstraint *heightConstraint;
}

// Listen for keyboard
- (void)establishNotificationHandlers
{
    // Listen for keyboard appearance
    [[NSNotificationCenter defaultCenter]
        addObserverForName:UIKeyboardWillShowNotification
        object:nil queue:[NSOperationQueue mainQueue]
        usingBlock:^(NSNotification *note)
    {
        // Fetch keyboard frame
        NSDictionary *userInfo = note.userInfo;
        NSTimeInterval duration =
            [userInfo[UIKeyboardAnimationDurationUserInfoKey]
                doubleValue];
        CGRect keyboardEndFrame = [self.superview
            convertRect:[userInfo[UIKeyboardFrameEndUserInfoKey]
                CGRectValue]
            fromView: self.window];

        // Adjust to window
        CGRect windowFrame = [self.superview
            convertRect:self.window.frame fromView:self.window];
        CGFloat heightOffset = (windowFrame.size.height -
            keyboardEndFrame.origin.y) -
            self.superview.frame.origin.y;

        // Update and animate height constraint
        heightConstraint.constant = heightOffset;
        [UIView animateWithDuration:duration animations:^(
```

```

        [self.superview layoutIfNeeded];
    }];
}];

// Listen for keyboard exit
[[NSNotificationCenter defaultCenter]
    addObserverForName:UIKeyboardWillHideNotification object:nil
    queue:[NSOperationQueue mainQueue]
    usingBlock:^(NSNotification *note)
    {
        // Reset to zero
        NSDictionary *userInfo = note.userInfo;
        NSTimeInterval duration =
            [userInfo[UIKeyboardAnimationDurationUserInfoKey]
             doubleValue];
        heightConstraint.constant = 0;
        [UIView animateWithDuration:duration animations:^(
            [self.superview layoutIfNeeded];
        )];
    }];
}

// Stretch sides and bottom of spacer to superview
- (void)layoutView
{
    self.translatesAutoresizingMaskIntoConstraints = NO;
    if (!self.superview) return;
    for (NSString *constraintString in @[@"H:[view] | ", @"V:[view] |"])
    {
        NSArray *constraints = [NSLayoutConstraint
            constraintsWithVisualFormat:constraintString options:0
            metrics:nil views:@{@"view":self}];
        [self.superview addConstraints:constraints];
    }
    heightConstraint = [NSLayoutConstraint constraintWithItem:self
        attribute:NSLayoutAttributeHeight
        relatedBy:NSLayoutRelationEqual toItem:nil
        attribute:NSLayoutAttributeNotAnAttribute multiplier:1.0f
        constant:0.0f];
    [self addConstraint:heightConstraint];
}

+ (instancetype)installToView:(UIView *)parent
{
    if (!parent) return nil;
    KeyboardSpacingView *view = [[self alloc] init];
    [parent addSubview:view];

    [view layoutView];
    [view establishNotificationHandlers];
}

```

```

        return view;
    }

    @end

```

## 6.4 解决方案：创建自定义的输入视图

当文本视图或文本框成为第一响应者之后，我们可以用自定义的输入视图（custom input view）把系统所提供的键盘替换成自己所设计的视图。这种自定义的输入视图不仅可以添加到文本视图中，而且还可以前加到非文本视图（nontext view）<sup>①</sup>中。解决方案 6-4 关注前一种情况。

如果我们设置了响应者的 `inputView` 属性，那么赋给该属性的那个视图就会取代系统键盘。有种非常简单的办法可以演示这一特性，就是创建纯色视图，并将其赋给 `inputView` 属性。下面这段代码创建了两个文本框，然后又创建了背景为紫色的 `UIView` 实例，并把该实例赋给第二个文本框的 `inputView` 属性：

```

// Create two standard text fields
UITextField *textField1 = [[UITextField alloc] init];
textField1.borderStyle = UITextBorderStyleRoundedRect;
[self.view addSubview:textField1];
PREPCONSTRAINTS(textField1);
CONSTRAIN_SIZE(textField1, 30, 200);
CENTER_VIEW_H(self.view, textField1);
ALIGN_VIEW_TOP_CONSTANT(self.view, textField1, 40);

UITextField *textField2 = [[UITextField alloc] init];
textField2.borderStyle = UITextBorderStyleRoundedRect;
[self.view addSubview:textField2];
PREPCONSTRAINTS(textField2);
CONSTRAIN_SIZE(textField2, 30, 200);
CENTER_VIEW_H(self.view, textField2);
ALIGN_VIEW_TOP_CONSTANT(self.view, textField2, 80);

// Create a purple view to be used as the input view
UIView *purpleView = [[UIView alloc] initWithFrame:
    CGRectMake(0.0f, 0.0f, self.view.frame.size.width, 120.0f)];
purpleView.backgroundColor = COOKBOOK_PURPLE_COLOR;

// Assign the input view
textField2.inputView = purpleView;

```

图 6-5 演示了上述代码的运行效果。当第一个文本框成为第一响应者之后，系统所提供的键盘就会显示在屏幕上；而当用户选中第二个文本框时，屏幕上则会出现那个紫色的视图。

① 可以理解为本身不具备文本输入功能的视图。——译者注

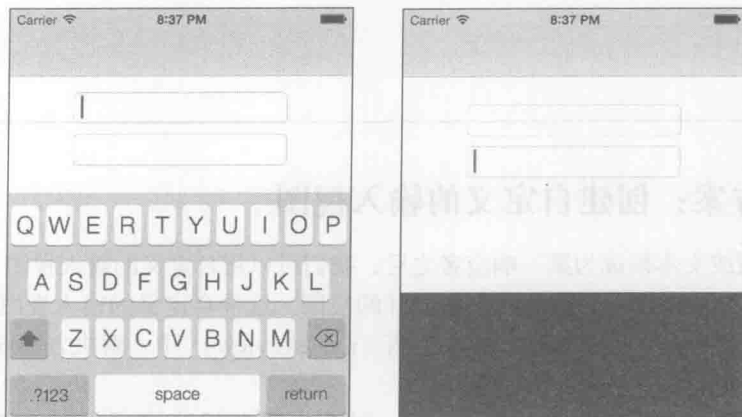


图 6-5 这两个文本框的其他方面都相同，只是在成为了第一响应者之后会产生不同的效果。顶部那个文本框会显示标准的键盘（如左侧截图所示），而底部的文本框则会显示出我们赋给 `inputView` 属性的那个纯色视图，而不会弹出系统键盘（如右侧截图所示）

由于紫色视图里面没有提供互动式控件，所以我们没办法再继续操作下去了。不能输入文本，也不能把这个“键盘”隐藏起来。我们只能欣赏一下这种可以显示自定义视图的功能。现在请重新选中顶部那个文本框，切换到标准键盘。

在绝大多数的日常编码工作中，我们都不会采用自定义的输入视图来实现文本输入。对于某些类型的程序来说，输入视图确实扮演着重要的角色，比方说，设计游戏时，就非常需要用到这种视图。尽管如此，在实现文本输入功能的时候，我们还是很少使用它。一方面是因为我们可以改用 `inputAccessoryView` 属性来做，那样既能保留系统内置的键盘，同时又能给键盘上面添加额外的按键。而另一方面则在于，系统键盘本身已经提供了输入数字和小数（iOS 4.1 系统加入了这一选项）专用的键位组合。在早期的 iOS 版本中，开发者之所以要实现自定义的输入视图，很大程度上是因为想设计这种专用的键盘，而现在则不需要这样做了。

那么，在处理文本的时候，应该于什么场合使用自定义的输入视图呢？它适用于那种开发者想要完全控制用户体验的场合。在这种情况下，我们要花时间和精力来设计自己的键盘，而且需要把各种平台及屏幕方向都考虑进来，另外还要注意 Shift 键的问题。我们可以创建一种能够完全自定义而且能够更换皮肤的输入控件来取代系统键盘，以便与自己独有的设计风格相搭配。这在多个层面上都需要很大的工作量才能做到。

解决方案 6-4 提供了一个极为简单的例子，用来演示自定义的文本输入视图。这个视图不是用来输入单个字符的，而是提供了两个按钮：一个可以输入 Hello，另一个可以输入 World（参见图 6-6）。

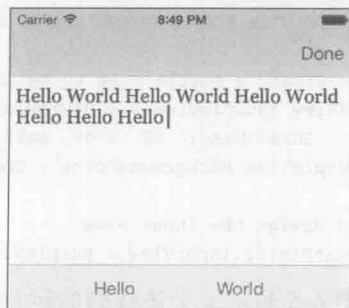


图 6-6 将这个文本视图的 `inputView` 设为自定义的键盘，使得用户可以输入 Hello 与 World 这两个词——这种键盘只有这个功能



用户按下某个按钮后，对应的单词就会添加到相关的文本视图里面。

创建这种自定义的文本输入视图时，其难点在于如何把文本的变更反馈给第一响应者。iOS 没有直接提供这样一种方式或属性，使得自定义的输入视图可以查出当前拥有焦点的那个视图，而且它也不能简单地通过访问上级视图的属性来确定此信息。为了应对这个问题，我们可以针对 `UIView` 类实现一个简单的扩展，用以获取当前的第一响应者：

```
@interface UIView (FirstResponderUtility)
+ (UIView *)currentResponder;
@end

@implementation UIView (FirstResponderUtility)
- (UIView *)findFirstResponder
{
    if ([self isFirstResponder]) return self;

    for (UIView *view in self.subviews)
    {
        UIView *responder = [view findFirstResponder];
        if (responder) return responder;
    }
    return nil;
}

+ (UIView *)currentResponder
{
    UIWindow *keyWindow =
        [[UIApplication sharedApplication] keyWindow];
    return [keyWindow findFirstResponder];
}
@end
```



**提示** 在上面这段代码中，笔者故意把那个类方法取名叫作 `currentResponder`，以免与私有的 API 相冲突。

没有对外公布的那个私有 API 方法名叫 `findFirstResponder`。在开发正式的软件产品（而不是本书这种范例程序）时，如果要针对苹果公司的类编写 `category`，并在其中添加方法，那么请遵循一条原则，就是给所有的方法名都加上前缀。这个前缀可以是开发者姓名的首字母缩写，也可以是公司名称的首字母缩写，还可以是某种独特的标识符。这样做可以确保 `category` 中的方法不会和苹果公司本身的方法重名。此外，还有个更为重要的原因，就是可以避免与苹果公司将来可能添加进来的方法相冲突。然而为了令范例代码读起来更容易也更好辨认，本书没有遵循这条建议。

解决方案 6-4 构建了自定义的 `UIToolbar`，并将其用作输入视图，这个 `UIToolbar` 上面有两个选项，分别是 `Hello` 和 `World`。用户点击某个按钮时，它会把对应的文本添加到

第一响应者现有的文本之中。如果发现 responderView 变量还没有设置好，就先查到第一响应者，并将其赋给该变量。然后，它会判断第一响应者所属的类是不是 UITextView。如果是的话，就把新的文本添加到里面。

对于输入视图（input view）来说，下列规律一般是成立的。首先，屏幕上展示出来的这个输入视图的拥有者（owner）总会具备第一响应者的身份。其次，该拥有者是应用程序的主窗口的子视图。我们可以利用这两条规律来改写代码，但这可能需要在解决方案 6-4 的基础上再添加一些错误检测代码，尤其要注意对 responderView 这个实例变量的复用。

#### 解决方案 6-4 创建自定义的输入视图

```
@interface InputToolbar : UIToolbar
@end

@implementation InputToolbar
{
    UIView *responderView;
}
- (void)insertString:(NSString *)string
{
    if (!responderView || ![responderView isKindOfClass:[UITextView class]])
    {
        responderView = [UIView currentResponder];
        if (!responderView) return;
    }

    if ([responderView isKindOfClass:[UITextView class]])
    {
        UITextView *textView = (UITextView *) responderView;
        NSMutableString *text =
            [NSMutableString stringWithString:textView.text];
        NSRange range = textView.selectedRange;
        [text replaceCharactersInRange:range withString:string];
        textView.text = text;
        textView.selectedRange =
            NSMakeRange(range.location + string.length, 0);
    }
    else
        NSLog(@"Cannot insert %@ in unknown class type (%@)",
            string, [responderView class]);
}

// Perform the two insertions
- (void)hello:(id)sender {[self insertString:@"Hello "];}
- (void)world:(id)sender {[self insertString:@"World "];}

// Initialize the bar buttons on the toolbar
- (instancetype)initWithFrame:(CGRect)aFrame
{
    self = [super initWithFrame: aFrame;
```

```

if (self)
{
    NSMutableArray *theItems = [NSMutableArray array];
    [theItems addObject:SYSBARBUTTON(
        UIBarButtonSystemItemFlexibleSpace, nil)];
    [theItems addObject:BARBUTTON(
        @"Hello", @selector(hello:))];
    [theItems addObject:SYSBARBUTTON(
        UIBarButtonSystemItemFlexibleSpace, nil)];
    [theItems addObject:BARBUTTON(
        @"World", @selector(world:))];
    [theItems addObject:SYSBARBUTTON(
        UIBarButtonSystemItemFlexibleSpace, nil)];
    self.items = theItems;
}
return self;
}
@end

```

## 6.5 解决方案：使视图具备文本输入功能

默认情况下，只有少数几个视图能够输入文本，不过，我们只需编写少量代码，就可以给几乎任意视图添加键盘支持了。其关键就在于实现简单的 `UIKeyInput` 协议。另外还需要稍微操作一下第一响应者，这样就可以随意为视图添加文本输入功能了。

解决方案 6-5 演示了如何把标准的 `UIToolbar` 变成一种可以接受键盘输入的视图，使得用户能够直接在工具栏里输入文本，其效果如图 6-7 所示。在用户打字的时候，工具栏里的文本也会随之更新，而且它还能正确地处理 `Delete`（删除）键。

这条解决方案需要用好几个特性才能实现出来。首先，工具栏必须声明它自己遵从 `UIKeyInput` 协议。遵从了该协议，就表明这个视图可以实现简单的文本输入功能，并且在成为第一响应者之后，能够把系统键盘（或自定义键盘）显示出来。

其次，工具栏必须保留自身状态，就是说，必须把用户输入的字符串保存起来。我们可以把字符串放在一个受保留而且可变的属性里，这样的话，工具栏就可以知道目前正在操作并且需要显示给用户看的那段文本了。

第三，工具栏必须成为第一响应者。可以用两种方式做到这一点：实现 `canBecomeFirstResponder` 方法并返回 `YES`；捕获触摸事件，并判断工具栏目前是否应该具备第一响应者这一角色。我们添加一个处理程序来处理用户对工具栏的触摸，并使其成为第一响应者。

最后，它必须实现 `UIKeyInput` 协议所规定的三个方法：`hasText`、`insertText:` 以及 `deleteBackward`。通过这些方法的名称，我们就能确切地知道其功能。如果视图里面有文本，那么 `hasText` 方法就应返回 `YES`。`insertText:` 方法会在当前的插入点添加

文本（在本例中，插入点总是位于现有文本的末端），而 `deleteBackward` 方法每次会从显示出来的这段文本末尾删除一个字符。

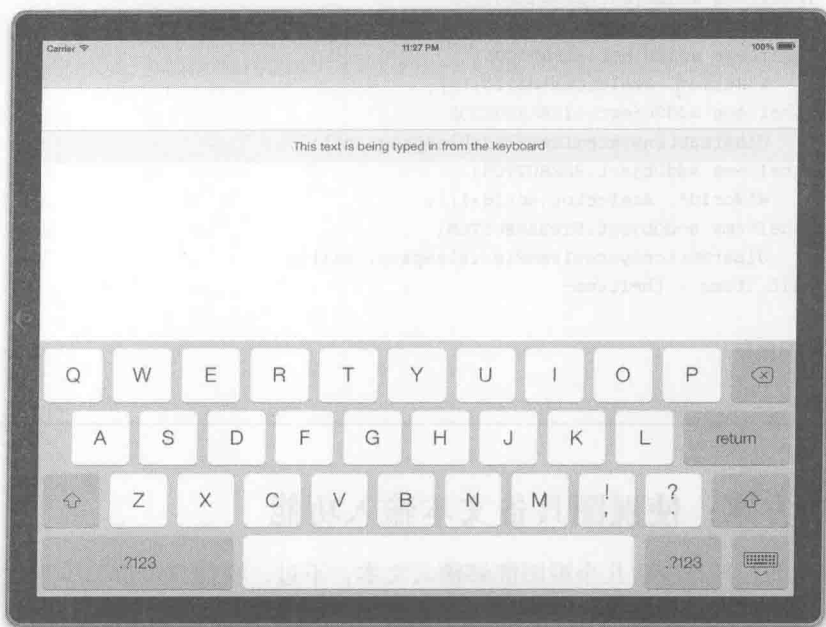


图 6-7 令 `UIToolbar` 遵从 `UIKeyInput` 协议，这样就可以把工具栏变成能够接受文本输入的视图了，这种视图会把输入用的键盘显示出来，而且能够正确处理“删除字符”这一操作

遵循 `UIKeyInput` 协议、成为第一响应者、处理与字符串有关的状态并处理与输入有关的回调方法——解决方案 6-5 正是通过这些手段，为基本的 `UIView` 对象添加了简单而稳定的文本输入功能。开发者可以根据应用程序的需求，把同样的文本输入功能添加到标签、导航栏、按钮等其他类里面。

#### 解决方案 6-5 给非文本视图添加键盘输入功能

```
@interface KeyInputToolbar: UIToolbar <UIKeyInput>
@end

@implementation KeyInputToolbar
{
    NSMutableString *string;
}

// Is there text available that can be deleted
- (BOOL)hasText
{
    if (!string || !string.length) return NO;
}
```

```

        return YES;
    }

    // Reload the toolbar with the string
    - (void)update
    {
        NSMutableArray *theItems = [NSMutableArray array];
        [theItems addObject:SYSBARBUTTON(UIBarButtonSystemItemFlexibleSpace, nil)];
        [theItems addObject:BARBUTTON(string, @selector(becomeFirstResponder))];
        [theItems addObject:SYSBARBUTTON(UIBarButtonSystemItemFlexibleSpace, nil)];

        self.items = theItems;
    }

    // Insert new text into the string
    - (void)insertText:(NSString *)text
    {
        if (!string) string = [NSMutableString string];
        [string appendString:text];
        [self update];
    }

    // Delete one character
    - (void)deleteBackward
    {
        // Super caution, even if hasText reports YES
        if (!string)
        {
            string = [NSMutableString string];
            return;
        }

        if (!string.length)
            return;

        // Remove a character
        [string deleteCharactersInRange:NSMakeRange(string.length - 1, 1)];
        [self update];
    }

    // When becoming first responder, send out a notification to that effect.
    // Can be used to add a Done button in the navigation bar
    - (BOOL)becomeFirstResponder
    {
        BOOL result = [super becomeFirstResponder];
        if (result)
        {
            [[NSNotificationCenter defaultCenter]
             postNotification:[NSNotification notificationWithName:
                             @"KeyInputToolbarDidBecomeFirstResponder" object:nil]];
        }
        return result;
    }

```

```

- (BOOL)canBecomeFirstResponder
{
    return YES;
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;
{
    [self becomeFirstResponder];
}
@end

```

---

## 6.6 解决方案：为非文本视图添加自定义的输入视图

我们可以给文本视图及文本框指定自定义的输入视图，不过，这种视图在其他情况下可能更为有用。输入的内容不一定必须是文本。实际上，只要抛开系统键盘，我们就可以根据自己的需要来设计自定义的输入视图了。

你可以把这种输入视图理解成与上下文相关的图形菜单，只有在特定的视图类成为第一响应者时，它才会弹出来。比方说，点击一个“战士”之后，屏幕上会出现几种武器，其中有弓、杖、剑。用户可以在里面选择战士的攻击方式。又比如有个图形排版程序。当用户点击其中的圆圈、正方形或线条时，屏幕上可能会出现调色板，用户可以在其中设置线条宽度、线条颜色以及填充方式。只要尽情地想象，就能用自定义的输入视图设计出很多东西来。

解决方案 6-6 演示了如何用自定义的输入视图来影响非文本视图中的内容。它把解决方案 6-4 与解决方案 6-5 的代码结合起来，创建了具备输入功能的视图（也就是 ColorView），该视图在接受用户触摸之后，会变为第一响应者，同时，它给这个视图配置了自定义的输入视图（也就是 InputToolbar），此视图能够影响主视图的显示效果。在本例中，ColorView 这个基本的视图仅仅负责演示颜色。而作为输入视图的 InputToolbar 则用来控制具体应该显示何种颜色。

解决方案 6-6 为非文本视图创建自定义的输入控制器

```

@interface ColorView : UIView
@property (strong) UIView *inputView;
@end

// Key Input Aware View
@implementation ColorView

// UITextInput protocol
- (BOOL)hasText {return NO;}
- (void)insertText:(NSString *)text {}
- (void)deleteBackward {}

```

```

// First responder support
- (BOOL)canBecomeFirstResponder {return YES;}
- (void)touchesBegan:(NSSet *)touches
    withEvent:(UIEvent *)event {[self becomeFirstResponder];}

// Initialize with user interaction allowed
- (instancetype)initWithFrame:(CGRect)aFrame
{
    self = [super initWithFrame:aFrame];
    if (self)
    {
        self.backgroundColor = COOKBOOK_PURPLE_COLOR;
        self.userInteractionEnabled = YES;
    }
    return self;
}
@end

// Color input toolbar
@interface InputToolbar : UIToolbar <UIInputViewAudioFeedback>
@end

@implementation InputToolbar
- (BOOL)enableInputClicksWhenVisible
{
    return YES;
}

- (void)updateColor:(UIColor *)aColor
{
    [UIView currentResponder].backgroundColor = aColor;
    [[UIDevice currentDevice] playInputClick];
}

// Color updates
- (void)light:(id) sender {
    [self updateColor:[COOKBOOK_PURPLE_COLOR
        colorWithAlphaComponent:0.33f]];
}
- (void)medium:(id) sender {
    [self updateColor:[COOKBOOK_PURPLE_COLOR
        colorWithAlphaComponent:0.66f]];
}
- (void)dark:(id) sender {
    [self updateColor:COOKBOOK_PURPLE_COLOR];
}

// Resign first responder on pressing Done
- (void)done:(id) sender
{
    [[UIView currentResponder] resignFirstResponder];
}

// Create a toolbar with each option available

```

```

- (instancetype)initWithFrame:(CGRect)aFrame
{
    self = [super initWithFrame:aFrame];
    if (self)
    {
        NSMutableArray *theItems = [NSMutableArray array];
        [theItems addObject:BARBUTTON(@"Light", @selector(light:))];
        [theItems addObject:SYSBARBUTTON(
            UIBarButtonSystemItemFlexibleSpace, nil)];
        [theItems addObject:BARBUTTON(@"Medium", @selector(medium:))];
        [theItems addObject:SYSBARBUTTON(
            UIBarButtonSystemItemFlexibleSpace, nil)];
        [theItems addObject:BARBUTTON(@"Dark", @selector(dark:))];
        [theItems addObject:SYSBARBUTTON(
            UIBarButtonSystemItemFlexibleSpace, nil)];
        [theItems addObject:BARBUTTON(@"Done", @selector(done:))];
        self.items = theItems;
    }
    return self;
}
@end

```

由于没有其他方式能够转移第一响应者身份，所以我们在自定义的输入视图上面，给用户提供了 Done 按钮，使其可以关闭键盘，从而解除大 ColorView 的第一响应者身份。

## 添加按键音效

我们可以通过 UIDevice 类来为自定义的输入视图添加点击按键的声音。playInputClick 方法能够播放标准的系统键盘敲击声，当用户点击我们自制的键盘时，可以调用该方法来播放按键音效。

我们令自定义的输入视图遵循 UIInputViewAudioFeedback 协议，并添加名为 enableInputClicksWhenVisible 的委托方法，令其返回 YES。程序是否真的会播放按键音效，还要看用户在 Settings > Sounds 里面所选则的音频播放设置。只有当用户开启了 Keyboard Clicks 时，才会在点击键盘按键的时候听到按键音。假如用户没开启该选项，那么系统将会忽略 playInputClick 语句。

## 6.7 解决方案：创建更好的文本编辑器（第一部分）

我们可以给应用程序里的文本编辑器添加撤销（Undo）功能，并令其随时显示出帮助信息，以便将它做得更好一些。这些特性能够确保用户在打错字之后可以撤销刚才所输入的内容，并使编辑器中的文本恢复到原来的样子。通过解决方案 6-7，我们会惊讶地发现这种功能只需编写一点点代码就能实现出来。

文本视图本身内置了 select（选择）、cut（剪切）、copy（复制）与 paste（粘贴）等操作。而 undo manager（撤销管理器）则能够理解这些操作的含义，而且还能够解读诸如 Undo



Paste (撤销粘贴操作)、Redo Cut (重做剪切操作) 等消息。若想支持撤销功能, 视图控制器只需实例化一个 undo manager 即可, 剩下的事都由系统内置的对象来完成。

解决方案 6-7 向键盘的辅助视图中添加了 Undo 和 Redo 按钮。我们必须根据文本视图的内容随时更新这些按钮的状态。为了实现此目标, 我们把视图控制器设为文本视图的 delegate (委托), 并实现名为 textViewDidChange: 的委托方法。这样就能根据文本内容来启用或禁用按钮了。

这条解决方案采用持久化存储形式来保存文本内容, 以便下次启动程序时可以继续使用。它会在 performArchive 方法里将文本存入文件中。AppDelegate 会在程序即将暂停之前调用这个方法, 而且还会在文本视图每次放弃第一响应者身份时调用它, 以确保下次打开应用程序后, 能够看到最新的文本:

```
- (void) applicationWillResignActive: (UIApplication *)application
{
    [tbvc archiveData];
}
```

程序启动的时候, 会读入该文件里的数据, 并在设置视图控制器的时候初始化 UITextView 实例。

#### 解决方案 6-7 为文本视图添加撤销功能及持久化功能

```
#define SYSBARBUTTON(ITEM, SELECTOR) [[UIBarButtonItem alloc] \
    initWithBarButtonSystemItem:ITEM target:self action:SELECTOR]
#define SYSBARBUTTON_TARGET(ITEM, TARGET, SELECTOR) \
    [[UIBarButtonItem alloc] initWithBarButtonSystemItem:ITEM \
    target:TARGET action:SELECTOR]

// Store data out to file
- (void)archiveData
{
    [textView.text writeToFile:DATAPATH atomically:YES
    encoding:NSUTF8StringEncoding error:nil];
}

// Update the undo and redo button states
- (void)textViewDidChange: (UITextView *)textView
{
    [self loadAccessoryView];
}

// Choose which items to enable and disable on the toolbar
- (void)loadAccessoryView
{
    NSMutableArray *items = [NSMutableArray array];
    UIBarButtonItem *spacer =
        SYSBARBUTTON(UIBarButtonSystemItemFixedSpace, nil);
    spacer.width = 40.0f;
```

```

        BOOL canUndo = [textView.undoManager canUndo];
        UIBarButtonItem *undoItem = SYSBARBUTTON_TARGET(
            UIBarButtonItemSystemItemUndo, self, @selector(undo));
        undoItem.enabled = canUndo;
        [items addObject:undoItem];
        [items addObject:spacer];

        BOOL canRedo = [textView.undoManager canRedo];
        UIBarButtonItem *redoItem = SYSBARBUTTON_TARGET(
            UIBarButtonItemSystemItemRedo, self, @selector(redo));
        redoItem.enabled = canRedo;
        [items addObject:redoItem];
        [items addObject:spacer];

        [items addObject:SYSBARBUTTON(UIBarButtonSystemItemFlexibleSpace, nil)];
        [items addObject:BARBUTTON(@"Done", @selector(leaveKeyboardMode))];

        toolbar.items = items;
    }

    // Call undo on the undoManager and update toolbar buttons
    - (void)undo
    {
        [textView.undoManager undo];
        [self loadAccessoryView];
    }

    // Call redo on the undoManager and update toolbar buttons
    - (void)redo
    {
        [textView.undoManager redo];
        [self loadAccessoryView];
    }

    // Return a plain accessory view
    - (UIToolbar *)accessoryView
    {
        toolbar = [[UIToolbar alloc]
            initWithFrame:CGRectMake(0.0f, 0.0f, 100.0f, 44.0f)];
        toolbar.tintColor = [UIColor darkGrayColor];
        return toolbar;
    }

    - (void)loadView
    {
        self.view = [[UIView alloc] init];

        // Load any existing string
        if ([[NSFileManager defaultManager] fileExistsAtPath:DATAPATH])

```

```

{
    NSString *string =
        [NSString stringWithContentsOfFile:DATAPATH
        encoding:NSUTF8StringEncoding error:nil];
    textView.text = string;
}

// Subscribe to keyboard frame changes and update layout
[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(updateTextViewBounds:)
    name:UIKeyboardDidChangeFrameNotification object:nil];
}

```

## 6.8 解决方案：创建更好的文本编辑器（第二部分）

从 iOS 6 开始，文本视图与文本框都支持用 Attributed Text String（带属性的文本字符串）来修饰文本了（也就是说，开始支持带有样式的文本了，而不是仅仅支持纯文本）。这样的话，我们就能创建字体、风格和颜色各异的文本视图和文本框了。

在 iOS 7 之前，要想实现一些稍微复杂的样式，就必须依赖 Core Text 才行。而现在有了 Text Kit，它简化并扩展了对文本样式的支持。对于本例这种简单的文本编辑器来说，只需编写少量代码，就能令其支持粗体、斜体及下划线等基本样式。

### 6.8.1 启用 Attributed Text

为了处理用户修改文本样式的请求，我们必须改变文本视图的一个标志，令其可以支持 Attributed Text（Attributed Text 就是 Styled Text 的意思，即有样式的文本）。把 `allowsEditingTextAttributes` 属性设为 YES 之后，会出现下面几种情况：

- 文本视图开始更新它的 `attributedText` 属性。开发者可以通过该属性，以 `AttributedString` 的形式获取文本视图中的内容。
- 文本视图开始响应一系列 `UIResponder` 方法，当用户要对所选文本运用粗体、斜体及下划线等效果时，我们就需要在对应的方法里做出响应。下一节将详细讲述这些方法。
- 视图的交互式用户界面菜单开始显示新的选项，用户可以通过这些选项，对当前所选的文本运用粗体、斜体及下划线等效果。

### 6.8.2 控制文本的样式

在 iOS 6 中，`NSObject` 提供了一些方法，可用来控制文本的许多特征。这些方法是非正式协议 `UIResponderStandardEditActions` 的一部分，而且是设计给 `UIResponder` 子类使用的。此协议声明了 iOS 用户界面中一些常用的编辑命令。

与本例有关的方法包括 `toggleBoldFace:`、`toggleItalics:` 及 `toggleUnderline:`。这三个方法用于给当前所选文本添加相关样式，如果这段文本已经运用了对应的样式，那么

就会将该样式移除。

我们只需命令 responder (在本例中, 指的是文本视图) 启用文本样式编辑功能, 就可以支持这些样式修改操作。相关的文本视图或文本框会处理所有细节问题。我们要做的, 就是把这些方法指定为工具栏上相关按钮的动作。

解决方案 6-8 演示了怎样将这些功能集成到 iOS 应用程序里。图 6-8 是由这条解决方案所构建出来的界面。

#### 解决方案 6-8 增强版文本编辑器

```
// Handy bar button macros
#define BARBUTTON(TITLE, SELECTOR) [[UIBarButtonItem alloc] \
initWithTitle:TITLE style:UIBarButtonItemStylePlain \
target:self action:SELECTOR]
#define BARBUTTON_TARGET(TARGET, TITLE, SELECTOR) \
[[UIBarButtonItem alloc] initWithTitle:TITLE \
style:UIBarButtonItemStylePlain target:TARGET action:SELECTOR]
#define SYSBARBUTTON(ITEM, SELECTOR) [[UIBarButtonItem alloc] \
initWithBarButtonSystemItem:ITEM target:self action:SELECTOR]
#define SYSBARBUTTON_TARGET(ITEM, TARGET, SELECTOR) \
[[UIBarButtonItem alloc] initWithBarButtonSystemItem:ITEM \
target:TARGET action:SELECTOR]

// Choose which items to enable and disable on the toolbar
- (void)loadAccessoryView
{
    NSMutableArray *items = [NSMutableArray array];

    BOOL canUndo = [textView.undoManager canUndo];
    UIBarButtonItem *undoItem = SYSBARBUTTON_TARGET(
        UIBarButtonSystemItemUndo, self, @selector(undo));
    undoItem.enabled = canUndo;
    [items addObject:undoItem];

    BOOL canRedo = [textView.undoManager canRedo];
    UIBarButtonItem *redoItem = SYSBARBUTTON_TARGET(
        UIBarButtonSystemItemRedo, self, @selector(redo));
    redoItem.enabled = canRedo;
    [items addObject:redoItem];

    // Add select all
    [items addObject:SYSBARBUTTON(UIBarButtonSystemItemFlexibleSpace, nil)];
    [items addObject:BARBUTTON_TARGET(textView, @"Sel", @selector(selectAll:))];

    // Add style buttons
    [items addObject:SYSBARBUTTON(UIBarButtonSystemItemFlexibleSpace, nil)];
    [items addObject:BARBUTTON_TARGET(textView,
        @"B", @selector(toggleBoldface:))];
    [items addObject:BARBUTTON_TARGET(textView,
        @"I", @selector(toggleItalics:))];
```

```

[items addObject:BARBUTTON_TARGET(textView,
    @"U", @selector(toggleUnderline:))];
[items addObject:SYSBARBUTTON(UIBarButtonSystemItemFlexibleSpace, nil)];
[items addObject:BARBUTTON(@"Done", @selector(leaveKeyboardMode))];

toolbar.items = items;
}

```

### 6.8.3 可供 UIResponder 使用的其他功能

请注意辅助工具栏上面的 Sel 按钮，该按钮位于 BIU 选项左侧（B、I、U 分别表示粗体（bold）、斜体（italic）、下划线（underline））。这个 Sel 按钮为编辑器添加了 Select All（选取全部文本）的功能，我们是经由 UIResponderStandardEditActions 协议来实现此功能的。这个协议包含了下列与编辑操作有关的方法：



图 6-8 UIResponderStandardEditActions 协议定义了常用的文本编辑命令，开发者可以在自己设计的用户界面里调用这些命令。系统菜单中会自动出现 BIU 选项，而键盘上方那个辅助视图（accessory view）也为这三个选项分别提供了三个按钮。用户可以通过辅助视图里的 Sel 按钮来选取全部文本，也可以通过 B、I、U 等按钮对文本运用（或者从文本中取消）加粗、变斜体以及加下划线等效果

- 基本的编辑操作：copy:、cut:、delete: 及 paste:
- 与选取有关的 select: 及 selectAll:
- 与修改样式有关的 toggleBoldFace:、toggleItalics: 及 toggleUnderline:

开发者可以通过该协议的 makeTextWritingDirectionLeftToRight: 及 makeTextWritingDirectionRightToLeft: 方法来控制书写方向。

## 6.9 解决方案：过滤用户所输入的文本

有时候我们想确保用户只能输入某个特定范围内的字符。比方说，开发者可能想创建一种只接受数字而不接受字母的文本框。虽说可以通过谓词将最终的文本同正则表达式 (regular expression) 相比较，并以此来过滤数据 (NSPredicate 类的 MATCH 操作符支持将正则表达式用作其值，解决方案 6-10 将会演示这一功能)，但还有一种更简单的办法，就是在用户打字时直接检查每个新字符是否处于可以接受的范围。

当用户输入字符的时候，我们可通过 UITextField 的委托来捕获这些字符，并决定是否应该将其添加到当前活动的文本框中。有个可选的委托方法叫作 textField:shouldChangeCharactersInRange:replacementString:, 如果它返回 YES, 就表示应该把新输入的字符添加到文本框中，若返回 NO, 则表示不允许把新字符添加进来。实际上，每当用户点击键盘时，系统就会调用这个方法，于是，我们就可以逐次检查每一个字符了。但如果用户通过 iOS 的剪贴板功能把文本粘贴到文本框里，那么粘贴进来的文本可就不一定只包含一个字符了。

解决方案 6-9 会在新字符串里寻找不允许输入的字符。如果找到了这种字符，就拒绝用户所输入的内容，并保持文本框里原有的文本不变。假如用户想要粘贴进来的文本里面既有我们允许的字符，又有不允许的字符，那么就把整个文本全部回绝。

解决方案 6-9 对用户所输入的文本进行过滤

```
#define ALPHA @"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz "
```

```
@implementation TestBedViewController
```

```
- (BOOL)textField:(UITextField *)aTextField
  shouldChangeCharactersInRange:(NSRange)range
  replacementString:(NSString *)string
```

```
{
```

```
    NSMutableCharacterSet *cs =
        [NSMutableCharacterSet
         characterSetWithCharactersInString:@""];

```

```
    switch (segmentedControl.selectedSegmentIndex)
    {
```

```
        case 0:
```

```
            [cs addCharactersInString:ALPHA];
```

```
            break;
```

```
        case 1:
```

```
            [cs formUnionWithCharacterSet:
             [NSCharacterSet decimalDigitCharacterSet]];
            break;
```

```
        case 2:
```

```
            [cs formUnionWithCharacterSet:
             [NSCharacterSet decimalDigitCharacterSet]];

```

```
        // permit one decimal only
```

```
        if ([textField.text rangeOfString:@"."].location
```

```

        == NSNotFound)
        [cs addCharactersInString:@"."];
        break;
    case 3:
        [cs addCharactersInString:ALPHA];
        [cs formUnionWithCharacterSet:
            [NSCharacterSet decimalDigitCharacterSet]];
        break;
    default:
        break;
}

NSString *filtered =
    [[string componentsSeparatedByCharactersInSet:[cs invertedSet]]
    componentsJoinedByString:@""];
BOOL basicTest = [string isEqualToString:filtered];
return basicTest;
}

- (void)segmentChanged:(UISegmentedControl *)seg
{
    // Reset text on segment change
    textField.text = @"";
}

- (void)loadView
{
    self.view = [[UIView alloc] init];
    self.view.backgroundColor = [UIColor whiteColor];

    // Create a testbed text field to work with
    textField = [[UITextField alloc] init];
    textField.placeholder = @"Enter Text";
    [self.view addSubview:textField];

    PREPCONSTRAINTS(textField);
    CONSTRAIN(self.view, textField, @"V:|-30-[textField]");
    CONSTRAIN(self.view, textField, @"H:|--[textField(>=0)]-|");

    textField.delegate = self;
    textField.returnKeyType = UIReturnKeyDone;
    textField.clearButtonMode = UITextFieldViewModeAlways;
    textField.borderStyle = UITextBorderStyleRoundedRect;
    textField.autocorrectionType = UITextAutocorrectionTypeNo;
    // Add segmented control with entry options
    segmentedControl = [[UISegmentedControl alloc] initWithItems:
        [@"ABC 123 2.3 A2C" componentsSeparatedByString:@" "]];
    segmentedControl.selectedSegmentIndex = 0;
    [segmentedControl addTarget:self action:@selector(segmentChanged:)
        forControlEvents:UIControlEventValueChanged];
}

```

```

        self.navigationItem.titleView = segmentedControl;
    }
@end

```

本条解决方案考虑了四种情况：只能输入字母、只能输入数字、只能输入数字和小数点、可混合输入字母及数字。你可以根据自己想要接受的其他字符集来改编范例代码。

对于第三种过滤类型，也就是只能输入数字和小数点的那种类型来说，需要使用一点小技巧才能保证文本框中只会会有一个小数点。如果在相关的文本框里已经发现了小数点，那么把可以接受的字符集从“数字和小数点”切换到“只包含数字”。用户可以通过粘贴操作来绕过这一限制。虽说这种情况不太可能发生，但开发者仍然应该考虑到才对。我们通过覆写文本框的 `canPerformAction:withSender:` 方法来把粘贴操作从用户可以使用的操作里排除掉。

下面这段代码使得用户无法向文本框中粘贴内容。当系统向该方法查询是否可以使用 `paste:` 操作时，该方法会返回 `NO`。它还通过类似的条件判断语句来保证 `Select`（选取）和 `Select All`（全选）操作必须在文本框里有内容的时候（也就是 `hasText` 为 `YES` 的时候）才能执行。在用户所选的内容不为空时，程序会开启 `Cut` 及 `Copy` 操作：

```

@interface LimitedTextField : UITextField
@end

@implementation LimitedTextField
- (BOOL)canPerformAction:(SEL)action withSender:(id)sender
{
    UITextRange *range = self.selectedTextRange;
    BOOL hasText = self.text.length > 0;

    if (action == @selector(cut:)) return !range.empty;
    if (action == @selector(copy:)) return !range.empty;
    if (action == @selector(select:)) return hasText;
    if (action == @selector(selectAll:)) return hasText;
    if (action == @selector(paste:)) return NO;

    return NO;
}
@end

```

我们要记住一条经验：如果程序的机制有漏洞，那么请不要低估用户利用这一漏洞的能力。

## 6.10 解决方案：检测文本模式

解决方案 6-9 介绍了如何把用户能够输入的字符限制在我们所允许的范围内。有了这个程序之后，只需再前进一步，就可以学会怎样检测出用户所输入的内容是否符合各种文本模式（`text pattern`）了。比方说要判断用户输入的是不是浮点数，我们可以把浮点数这种模式描述为：可选的正负号后面跟着整数部分，而整数部分后面可以有一个小数点，其后又有小数部分。还可以把整数部分规定为可选的，只要求有正负号、小数点及小数部分即可。



除了简单的数值之外，还有电话号码、电子邮箱地址、URL 等事物，而用来限定这些事物的标准会特别复杂。苹果公司已经用内置的 `NSDataDetector` 类把其中某些标准设计好了，但我们也应该学会如何构建自己的标准。

### 6.10.1 构建自己的正则表达式

某些标准化组织发布了对一些事物的精准描述，有些好心的开发者便把它们转化成了便于移植的正则表达式。比方说，下面这个正则表达式可以用来定义浮点数：

```
^[+-]?[0-9]+[\\.]?[0-9]*$
```

这个定义并不是十分完美，但在很多场合下还是相当好的，而且用起来也很灵活。它能够接受一大批浮点数，而且它规定开头的正负号是可选的。尽管这个表达式不能接受 `-75` 这样的数，但是它也有个好处，就是能把 `-.` 这样的输入拦住，笔者认为，这样的权衡方案是合理的，因为用户在输入了 `-75` 并遭到拒绝之后，很容易就能想到自己应该输入的是 `-0.75`。此外，也可以用另一个正则表达式把上述表达式无法接受的那些合法浮点数检测出来。比方说，我们想接受那种没有整数部分但是有小数点，而且小数点后面必须跟着一个或多个数位的浮点数：

```
^[+-]?\\.[0-9]+$
```

`NSPredicate` 实例可以把 `NSString` 文本与正则表达式相比较，并判断出用户是不是输入了有效的浮点数。比方说：

```
NSPredicate *fpPredicate = [NSPredicate predicateWithFormat:
    @"SELF MATCHES '^[-+]?[0-9]+[\\.]?[0-9]*$'"];
BOOL match = [fpPredicate evaluateWithObject:string];
```

刚才说过，要想用正则表达式来检测电话号码、电子邮件地址或其他更为复杂的输入类型是有些困难的。下面给出的这个正则表达式以最直接的方式描述了美国电话号码的格式：

```
^\\([\\]?([2-9][0-9]{2})[\\])?[-.\\.]?([2-9][0-9]{2})[-.\\.]?([0-9]{4})$
```

上述正则表达式规定括号是可选的，但我们没办法判断左右括号是否匹配，然而只需再编写一些简单的 Objective-C 代码，即可实现这一点。这条正则表达式能够确保区号（area code）和电话号码前缀（phone number prefix）都不以 0 或 1 开头，并且允许用户在电话号码的各个部分之间添加分隔符（分隔符可以是空格、连字符或句点）。换句话说，用上面这种单行的正则表达式来描述电话号码是相当合适的，但它并不是十分精准。

解决方案 6-10 采用这种正则表达式来判断用户输入的是不是电话号码。如果发现用户输入了格式正确的电话号码，那么程序就更新导航栏的标题，以此告知用户该输入是有效的。这条解决方案演示了怎样实时地过滤用户所输入的内容，以判断其是否与待匹配的模式相符，并在相符的时候做出响应。

### 6.10.2 枚举正则表达式

`NSRegularExpression` 类提供了一种基于块的枚举方式，可以用来寻找字符串中与

正则表达式相匹配的各个部分。我们可以用这种方式对特定范围内的文本做出修改。如果发现某段文本与正则表达式相匹配，那么可通过带属性的文本（attributed text）来设置其颜色或字体，以便向用户凸显这些内容。这种做法与文本视图的拼写检查器相似，后者会给拼错的单词添加下划线。

如果想自己来实现这种效果，那么就请创建一条正则表达式，然后将它与字符串（这个字符串通常是从文本视图之类的控件中获取的）相匹配，并在每一个能够匹配的范围（NSRange）内，为文本施加某种视觉效果。通过带属性的字符串（attributed string），我们可以用一种比早前 iOS 系统更为简单的方式来修改文本视图的内容，从而向用户提供视觉反馈效果：

```
// Check for matches
NSRegularExpression *regex = [NSRegularExpression
    regularExpressionWithPattern:@"REGEHERE"
    options:NSRegularExpressionCaseInsensitive error:nil];

// Enumerate over a string
[regex enumerateMatchesInString:text options:0 range:fullRange
    usingBlock:^(NSTextCheckingResult *match,
        NSMatchingFlags flags, BOOL *stop){
        NSRange range = match.range;
        // Perform some action on the range
    }];
```

## 6.10.3 数据探测器

NSDataDetector 类是 NSRegularExpression 的子类。数据探测器（data detector）可用来判断那些定义明确的数据类型，包括日期、地址、URL 链接、电话号码以及交通信息等，苹果公司已经彻底测试了相关算法，所以开发者不用自己再去创建正则表达式了。而且还有个好处，就是这些检测都已经本地化（localized）了。

我们采用与上一节相同的办法来遍历字符串中与正则表达式相匹配的各个部分。下面这段代码会搜寻字符串里的链接（也就是 URL）和电话号码：

```
NSError *error = NULL;
NSDataDetector *detector = [NSDataDetector dataDetectorWithTypes:
    NSTextCheckingTypeLink|NSTextCheckingTypePhoneNumber
    error:&error];

// Enumerate over a string
[detector enumerateMatchesInString:text options:0 range:fullRange
    usingBlock:^(NSTextCheckingResult *match,
        NSMatchingFlags flags, BOOL *stop){
        NSRange range = match.range;
        // Perform some action on the range
    }];
```

检查功能是围绕着 NSTextCheckingResult 类而构建的，该类描述了与数据探

测器所能发现的内容相匹配的信息。iOS 的数据探测器所支持的数据种类会不断地增加。目前支持的数据有日期 (NSTextCheckingTypeDate)、地址 (NSTextCheckingTypeAddress)、链接 (NSTextCheckingTypeLink)、电话号码 (NSTextCheckingTypePhoneNumber) 以及航班等交通信息 (NSTextCheckingTypeTransitInformation)。希望将来还能支持更多的类型, 比如常用的股票代码、UPS/FedEx 运单号码以及其他易于辨识的文本形式。

### 6.10.4 使用内置类型的探测器

UITextView 及 UIWebView 提供了内置的数据类型探测器, 能够辨识电话号码、HTTP 链接等内容。开发者设置了 dataDetectorTypes 属性之后, 视图就会自动把与文本模式相匹配的内容转换成可以点击的 URL, 并将其嵌入到视图的文本里。可以设置的数据类型包括地址、日历事件、链接以及电话号码。如果将属性设为 UIDataDetectorTypeAll, 那么就可以匹配所有支持的数据类型, 若设为 UIDataDetectorTypeNone, 则会禁用模式匹配功能。

### 6.10.5 有用的网站

使用正则表达式的时候, 下列网站可能会对编程工作有所帮助:

- Regular Expression Library 网站 (<http://regexlib.com/>) 列出了全球网友所贡献的上千个正则表达式。
- Regex Pal 网站 (<http://regexpal.com/>) 提供了互动式的 JavaScript 工具, 用于测试正则表达式。
- txt2re 生成器 (<http://txt2re.com/>) 可以根据访问者所提供的源字符串构建一段程序码, 这段程序码用正则表达式把字符串里与之相匹配的各部分提取出来。



**提示** 由于 iOS 7 支持 Text Kit, 所以除了使用 UITextField 的委托之外, 我们也可以在解决方案 6-9 和 6-10 中编写 NSTextStorage 的子类, 并覆写 processEditing 方法。

#### 解决方案 6-10 用谓词和正则表达式检测文本模式

```
@implementation TestBedViewController
{
    UITextField *textField;
    UISegmentedControl *segmentedControl;
}

- (void)updateStatus:(NSString *)string
{
    // This is a predicate matching U.S. telephone numbers
    NSPredicate *telePredicate = [NSPredicate predicateWithFormat:
```

```

        @"SELF MATCHES \
        '^[\(\)]{2}([0-9]{2})[\(\)]{2}[-.\. ]{2}([0-9]{2})\
        [-.\. ]{2}([0-9]{4})$'";
        BOOL match = [telePredicate evaluateWithObject:string];
        self.title = match ? @"Phone Number" : nil;
    }

- (BOOL)textField:(UITextField *)aTextField
    shouldChangeCharactersInRange:(NSRange)range
    replacementString:(NSString *)string
{
    NSString *newString = [textField.text
        stringByReplacingCharactersInRange:range withString:string];
    if (!string.length)
    {
        [self updateStatus:newString];
        return YES;
    }

    NSMutableCharacterSet *cs = [NSMutableCharacterSet
        characterSetWithCharactersInString:@""];
    [cs formUnionWithCharacterSet:
        [NSCharacterSet decimalDigitCharacterSet]];
    [cs addCharactersInString:@"()-. "];

    // Legal characters check
    NSString *filtered = [[string componentsSeparatedByCharactersInSet:
        [cs invertedSet]] componentsJoinedByString:@""];
    BOOL basicTest = [string isEqualToString:filtered];

    // Test for phone number
    [self updateStatus:basicTest ? newString : textField.text];

    return basicTest;
}

- (void)loadView
{
    self.view = [[UIView alloc] init];

    textField = [[UITextField alloc] initWithFrame:
        CGRectMake(0.0f, 0.0f, 300.0f, 30.0f)];
    textField.placeholder = @"Enter Phone Number";
    [self.view addSubview:textField];

    PREPCONSTRAINTS(textField);
    CONSTRAIN(self.view, textField, @"V:|-30-[textField]");
    CONSTRAIN(self.view, textField, @"H:|--[textField(>=0)]-|");

    textField.delegate = self;
}

```

```

textField.returnKeyType = UIReturnKeyDone;
textField.clearButtonMode = UITextFieldViewModeAlways;
textField.borderStyle = UITextBorderStyleRoundedRect;
textField.autocorrectionType = UITextAutocorrectionTypeNo;
}
@end

```

## 6.11 解决方案：检测 UITextView 中的拼写错误

UITextChecker 类可以自动扫描出文本里的拼写错误。使用这个类的时候，必须先设置目标语言，比方说，en 表示英语、en\_US 表示美国英语，fr\_CA 表示加拿大法语。语言码由 ISO 639-1 编码和可选的 ISO 3166-1 区域码组成。我们可以把目标语言设为 en，这样就会以通用的英语字典来检查拼写了，也可以将其设为 en\_US、en\_AU 或 en\_GB，以便用美国英语、澳大利亚英语或英国英语字典来检查拼写。开发者可以从 UITextChecker 中获取一份数组，它里面列出了可供选用的各种语言。

UITextChecker 类能够学习新词 (learnWord:) 也可以忘掉已经学会的词 (unlearnWord:)，这使得程序可以根据用户的需求来定制系统自带的字典。已经学会的那些词都是跨语言使用的，所以，如果向字典中添加了某个人名，那么该名字在每一种语言环境下均可使用。UITextChecker 对象也提供了一些实例方法，用来设置拼写检查时可以忽略的词。

解决方案 6-11 会逐次选中每一个拼错的词，以此演示如何将 UITextChecker 集成到应用程序中。为了实现此功能，需要控制每次在文本视图里所选定的文本范围。若想在 UITextView 中选取文本，它必须首先具备第一响应者的身份才行。我们用下列代码检查其是否具备响应者 (responder) 身份，若不具备该身份，则令其成为响应者：

```

if (![textView isFirstResponder])
    [textView becomeFirstResponder];

```

接下来计算要选中的文本范围 (计算时要考虑到文本长度)，并设置文本视图的 selectedRange 属性：

```

textView.selectedRange = NSRange(offset, length);

```

除了要具备第一响应者的身份之外，文本视图还必须能够编辑 (editable)，在这种情况下，只要选中了其中的内容，屏幕上就会出现键盘。由于用户可能会用键盘来编辑文本，所以程序代码必须考虑到编辑操作会干扰应用程序的情况。

### 解决方案 6-11 搜寻拼写错误

```

@implementation TestBedViewController

```

```

- (void)nextMisspelling:(id)sender
{

```

```

    if (![textView isFirstResponder])
        [textView becomeFirstResponder];

```

```

    NSRange entireRange = NSRange(0, textView.text.length);

```

```

// Scan for a new word from the current offset
NSRange range = [textChecker
    rangeOfMisspelledWordInString:textView.text
    range:entireRange
    startingAt:textOffset
    wrap:YES language:@"en"];

// Skip forward each time a new misspelling is found / select the word
if (range.location != NSNotFound)
{
    textOffset = range.location + range.length;
    textView.selectedRange = range;
}
else
    textOffset = 0;
}

@end

```

## 拼写检查器协议

我们可以给 NSString 添加一个方便的小协议 (little protocol), 通过 UITextChecker 来检查任意字符串的拼写是否正确 (如程序清单 6-1 所示)。

程序清单 6-1 拼写检查器协议

```

@implementation NSString (SpellCheck)
- (BOOL)isSpelledCorrectly
{
    UITextChecker *checker = [[UITextChecker alloc] init];
    NSRange checkRange = NSMakeRange(0, self.length);
    NSString *language = [[NSLocale currentLocale]
        objectForKey:NSLocaleLanguageCode];
    NSRange range = [checker rangeOfMisspelledWordInString:self
        range:checkRange startingAt:0 wrap:NO language:language];
    return (range.location == NSNotFound);
}

@end

```

## 6.12 搜寻文本中的字符串

对解决方案 6-11 做一些改编, 即可实现文本搜寻功能。为了实现此功能, 需要给导航栏添加一个文本框, 并把导航栏上面的按钮改为 Find, 然后用 NSString 的 rangeOfString:options:range: 方法来定位待搜寻的字符串。请注意, 要搜寻的字符串不能是 nil。在目标文本中找到与字符串相匹配的范围之后 (假设匹配位置不是 NSNotFound), 可调用文本视图的 scrollRangeToVisible: 方法, 将其滚动到这个位

置上。调用该方法时，要把刚才 `rangeOfString:options:range:` 方法所返回的范围传进去。



**提示** `NSNotFound` 是个常量，用来表示没能成功地定位到相关范围。搜索完之后请检查 `NSRange` 的 `location` 字段，以确保其值是有效的。

## 6.13 小结

本章介绍了在 iOS 应用程序里使用文本的许多种新方式。在这一章中，读者看到了如何控制键盘以及如何调整视图，使其与带键盘的文本输入界面相匹配。我们还讲解了怎样创建自定义的输入视图、怎样过滤文本以及怎样判断用户所输入的内容是否有效。在阅读下一章之前，请回顾下列问题：

- 不要预先假定用户会使用或不会使用蓝牙键盘。开发者应该用软件键盘及硬件键盘这两种输入方式来测试应用程序。
- 虽说辅助视图是一种向文本输入界面里添加额外功能的好办法，但也不要过度使用它。iPhone 和 iPod touch 的键盘本身已经占据了很大一部分屏幕。如果再添加辅助视图的话，就会进一步缩减用户可以看到的屏幕空间。若要使用辅助视图，则应尽量将其设计得简单一些。
- 别以为用户会使用晃动撤销（shake-to-undo）功能，这个功能的实用程度值得怀疑。我们应该直接在应用程序的 GUI 里面提供撤销/重做 Undo/Redo 功能，令用户立刻就能使用它们，而不是令其去回想苹果公司最近引入的那个比较隐晦的撤销方式。晃动撤销功能应该用来补充现有的撤销/重做功能，而不是替换它们。在附属视图上添加 Undo/Redo 按钮是非常合适的。
- 在检测用户所输入的内容时，可能找不到非常完美的正则表达式，但我们不应该因为不够完美就弃用那些能够覆盖大多数情况的式子。另外请不要忘记，可以依次使用多个正则表达式来测试同一问题的不同解法。
- 看看 Text Kit 所提供的新特性。Text Kit 比 Core Text 好用得多，而且提供了很多灵活的功能，用以实现文本渲染及文本输入。

## 使用视图控制器

视图控制器简化了许多 iOS 应用程序的视图管理。每个视图控制器都拥有一套视图层级，这套层级完整地体现了用户界面内的所有元件。在构建应用程序时，开发者可以把许多任务集中到视图控制器里面来做，比方说处理屏幕方向的变更以及应对用户的操作等。本章讲解如何使用基于视图控制器的类以及怎样用它们来设计实用的 iPhone/iPod 及 iPad 程序。

### 7.1 视图控制器

顾名思义，视图控制器就是 iOS Model-View-Controller（模型 - 视图 - 控制器）设计模式中的 *Controller* 部分。每个视图控制器都管理着一套视图，这些视图组成了程序用户界面里的一个组件。视图控制器负责协调视图的加载以及视图的样貌，同时还会响应用户的操作。

视图控制器也会与设备及底层操作系统相配合。比方说，用户旋转设备的时候，视图控制器会更新其中视图的布局。当操作系统的内存过低时，控制器也会对内存警告做出响应。

简言之，视图控制器提供了集中管理机制。它会处理一系列各自独立的开发需求，这些需求可能是由视图、模型、iOS 或设备本身所引发的。

视图控制器也把与显示方式有关的隐喻集中起来。我们可以在容器里面分层添加视图控制器，以实现出非常优秀的自定义界面。系统提供了一些很常用的父视图 / 子视图型视图控制器，比方说，导航控制器使得用户可以从一个视图切换到另一个视图，页面视图控制器可以模仿出电子书的效果，标签栏控制器（tab controller）提供了推压按钮（pushbutton），使得用户可以在多个子控制器之间切换，而分栏视图控制器（split view controller）则可以把主要内容以列表形式显示在一边，同时把细节信息展示在另一边。

视图控制器本身并不是视图，而是一种没有视觉样貌的类，它管理着视图。借助视图控



制器，开发者能够把视图放在比较大一些的应用程序里面使用。

iOS SDK 提供了许多视图控制器类。有些比较通用，有些比较具体。下面简单介绍其中几种视图控制器，大家在构建 iOS 应用程序的界面时会遇到它们。

### 7.1.1 UINavigationController 类

UINavigationController 是其他视图控制器类的父类，它用来管理主视图，视图控制器的很多工作都由这个类来完成。开发者需要花很大一部分时间来定制这个类的子类。基本的 UINavigationController 类能够管理主视图从启动到结束的整个生命期，并且会把此过程中某些必须响应的变化考虑进来。

UINavigationController 实例负责配置视图的样貌以及它所要显示的子视图。一般来说，它要从 XIB 或故事板文件中加载这些信息。此外，它也提供了一些实例方法，使开发者可以用代码来手工创建视图布局 (loadView)，或在视图加载完毕之后添加行为 (viewDidLoad)。

视图控制器还有另外一个职责，就是响应正在显示出来或正要隐藏起来的视图。对于大型应用程序中的视图来说，视图控制器确实要处理这些事。开发者可以在 viewWillAppear: 及 viewWillDisappear: 等方法中完成与视图管理有关的例行任务。我们可以在视图显示之前预先加载数据，也可以在视图即将消失时清理资源。

上面提到的每件任务都描述了视图与其外围应用程序之间的某种配合方式。UINavigationController 是用来协调视图和外部需求的，它会令视图自身发生改变，以满足那些需求。

### 7.1.2 导航控制器

从名称可知，导航控制器是一种可以在树状视图层级之间上下游走的控制器，对于比较小的 iOS 设备来说，这是一种重要而常见的界面设计方式，而对于平板电脑来说，它也可以用作辅助的设计方案。导航控制器会在屏幕上方创建半透明的导航栏，很多 iOS 应用程序里都能看见这种效果。

通过导航控制器，我们可以把新的视图叠放在已有的视图之上，此时会自动生成 Back 按钮，该按钮会显示出上一个视图控制器的标题。所有的导航控制器都使用根视图控制器来建立导航树，这个根控制器位于树状结构顶端，这样做使得用户可以通过 Back 按钮返回主视图。在平板电脑上，我们可以把基于导航控制器的界面与基于 UIBarButtonItem 的菜单项结合起来，以产生 popover 式的显示效果，也可以把它和 UISplitViewController 实例相集成，实现出主要内容与详细信息 (master/detail) 分栏展示的效果。

把界面导航这一功能交给导航控制器之后，开发者就可以专心地设计用户界面，并为每个视图控制器创建相应的屏幕。我们不用担心具体的导航细节，只需告诉导航控制器接下来要切换到哪个视图就行。系统会自动把原来的视图叠放起来，并处理好导航按钮。

### 7.1.3 标签栏控制器

通过 UITabBarController 类，我们可以在应用程序里控制许多平行的内容。这有

点像收音机的选台机制。不需要有特定的导航体系，用户就可以通过标签栏来“调”到自己喜欢的视图控制器上面。这些平行的内容都是各自独立的，而且可以具备自己的导航体系。开发者构建出与每个 Tab（标签）相对应的视图控制器或导航控制器，而 Cocoa Touch 则会把这些视图的相关细节处理好。

比方说，标签栏上面同时出现的视图控制器如果超过一定数量（在 iPhone 手机上面是 5 个，平板电脑上会更多），那么用户就可以通过 More > Edit 画面来定制它们。在这个画面中，用户可以把他们喜欢的控制器拖放到屏幕底部的按钮栏中。开发者不用编写额外的代码，就能为程序添加自由调整 Tab 的功能。我们要做的只是配置一下 `customizable-ViewControllers` 属性。

#### 7.1.4 分栏视图控制器

这种视图控制器适用于平板电脑上面的应用程序，`UISplitViewController` 类可以把一组固定的数据（通常是一张表格）封装起来，并将它们和细节展示界面相关联。iPad 的 Mail 程序里就会出现这种分栏视图（Split View）。在横屏模式下，消息列表出现在屏幕左侧，而每个消息的内容则显示在右侧。右侧的细节视图（也就是 Mail 程序里的消息内容）从属于左侧的主视图（也就是 Mail 程序的消息列表）。点击某条消息之后，右侧的视图就会随之更新，并将这条消息的内容显示出来。

在竖屏模式下，主视图通常是隐藏起来的。用户点击分栏视图左上角的按钮，程序就会把主视图以 popover 的形式展现出来，在 iOS 5.1 及其后的系统里，用户也可以通过滑动手势来访问主视图。

#### 7.1.5 页面视图控制器

与导航控制器、标签栏控制器、分栏视图控制器等相似，页面视图控制器也是存放其他视图控制器的一种容器。它以页面为单位来管理其内容，并且可以用书本那样的翻页形式或是滚动形式来展示这些内容。如果开发者要使用翻页模式，就需要设置“书脊”，一般来讲，书脊位于视图左侧或顶部。然后，我们把每个视图控制器当作书中的一页内容，添加到页面视图控制器里面，以便制作好这本“书”。用户可以通过翻页或拖动操作在页面之间切换。

#### 7.1.6 popover 控制器

popover 控制器专门用于平板电脑，这种控制器会在现有的界面内容上方弹出临时的视图。popover 控制器所展示出来的信息和一般的模态视图相似，都不会占据整个屏幕。用户通常是通过点击界面里的 `UIBarButtonItem` 来弹出这种 popover 的（开发者也可以用其他的交互技术来创建 popover），当操作完 popover 中的内容，或是在主视图范围以外点击屏幕时，popover 就会消失。

popover 的内容通常由另一个视图控制器来填充。我们可以先把那个视图控制器构建好，并将其设为 popover 的 `contentViewController` 属性，然后再把 popover 显示出来。通过这种极为灵活的编程技巧，我们可以把能够放在标准视图控制器中的任意内容都展示到

popover 里面去。



从 iOS 5 开始, 开发者可以在 UINavigationController 的子类里把自定义的内容集成到程序的导航栏界面中。请在 UINavigationController 的初始化方法 initWithNavigationBarClass:toolbarClass: 里编写相关代码。

## 7.2 使用导航控制器与分栏视图控制器来开发程序

对于屏幕空间有限的设备来说, UINavigationController 类是一种管理界面的重要方式。它构建出的虚拟界面比设备屏幕要大得多, 用户可以在这套界面的不同层级之间上下游走。导航控制器把它的 GUI 以简洁的树状结构组织起来, 用户可通过按钮及选项来浏览这个树状结构。Contacts 与 Settings 程序里都能见到导航控制器, 用户选取了某个内容之后, 程序就会切换到新的画面, 而当用户按下 Back 按钮时, 又会回到原来的画面。

很多标准的 GUI 元件都可以体现出导航控制器在应用程序中的用法, 如图 7-1 左侧所示。我们可以看到每个画面顶部导航栏里的大按钮, 当用户浏览下一级界面的时候, 导航栏左上角就会出现后退按钮, 另外, 导航栏右上角还会列出应用程序的其他功能, 例如 Edit (编辑) 等。很多带有导航控制器的应用程序都是以滚动列表为中心而构建出来的, 滚动列表里的元素会把用户领向新的画面, 在列表中, 每个单元格右侧会有扩展指示器 (是个灰色的横向 V 形图案 (chevron)) 或详情展示按钮 (是个带圆圈的 i 形图案)。

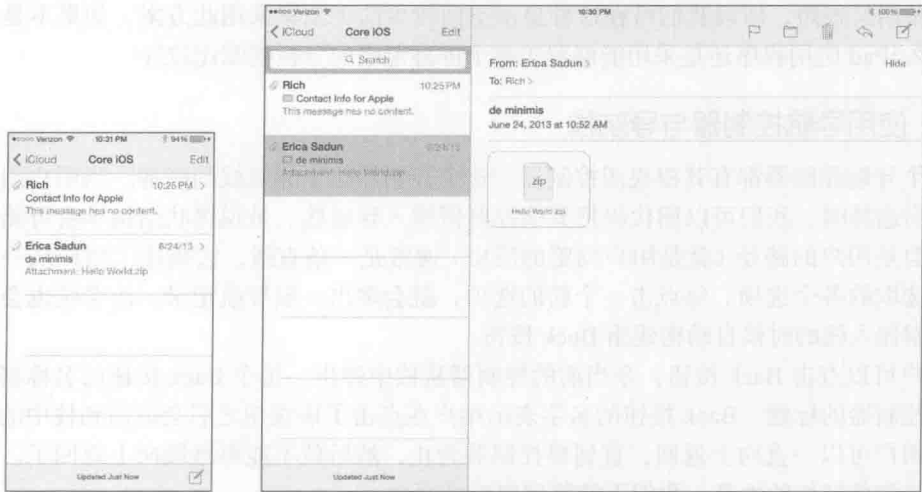


图 7-1 iPhone 的导航控制器 (如左侧截图所示) 用灰色的横向 V 形图案来提示用户: 选中某个内容之后, 程序会在屏幕上显示出与此内容有关的细节视图。而在 iPad 上面 (如右侧截图所示), 分栏视图控制器则会占据整个屏幕, 把左侧的导航元素和右侧的细节展示界面分隔开

由于 iPad 的屏幕比较大，所以不需要像 iPhone 手机那样采用导航控制器来节省屏幕空间。平板电脑上的程序也可以直接使用导航控制器，但是像图 7-1 右侧截图那样，采用 `UISplitViewController` 来设计界面，其效果会更加符合 iPad 这种大屏幕的设备。

请注意图 7-1 左侧的 iPhone 界面和右侧的 iPad 界面之间有何区别。iPad 的分栏视图控制器里没有横向的 V 形图案。用户点击某个元素之后，其细节数据会出现在同一个屏幕右侧那块较大的区域里面。而在 iPhone 上，由于屏幕空间较小，所以界面里会用横向的 V 形图案来提示用户：点击某个元素之后，屏幕上会弹出新的视图。这两种界面设计方式都把与设备屏幕有关的特征考虑进来了。

iPhone 和 iPad 上面的 Inbox 视图都采用了相似的导航控制器元件。它们包括 Back 按钮（写有“< iCloud”字样的那个按钮）、选项按钮（Edit 按钮）以及标题栏中的描述信息（也就是当前的文件夹 Core iOS）。每个元件都是通过导航控制器的 API 创建出来的，我们用这些元件来展示与电子邮件账户及其收件箱有关的层级结构。

这两套界面在导航树的最底层有所区别。树状数据结构中，这一层由很多“树叶”（leaf）组成，每个树叶表示一条电子邮件信息。iPhone 系列的界面用横向的 V 形图案来表示每个树叶节点。用户选中了某个树叶节点之后，程序会把相应的视图控制器叠放在导航栈（navigation stack）上面。而 iPad 程序则不会这样做。iPhone 程序是用横向 V 形图案来告诉用户当前已经浏览到整个树状结构最底层了，但 iPad 程序不使用这种图案，它直接把该节点的信息展示到右侧的视图中。

iPhone 风格的导航控制器也能够用在 iPad 上面。iPad 程序在显示临时的 popover 界面时，可以使用标准的（也就是 iPhone 风格的）导航控制器，由于此时的 popover 视图比较小，而且生命期又很短，所以我们可在这种显示空间较少的视图里采用此方案。如果不是这种情况，那么 iPad 应用程序还是采用能够覆盖整个屏幕范围的分栏视图比较好。

### 7.2.1 使用导航控制器与导航栈

每个导航控制器都有其根视图控制器。这个控制器位于导航栈的底部。当用户浏览模型树并做出选择时，我们可以用代码把其他控制器推入导航栈。虽说树状结构本身可能有许多分支，但是用户的路径（就是用户浏览的历史）通常是一条直线，它描述了用户从一开始到现在所选取的各个选项。每点击一个新的选项，就会多出一层导航记录，而系统也会在新的视图控制器入栈的时候自动构建出 Back 按钮。

用户可以点击 Back 按钮，令当前的控制器从栈中弹出。每个 Back 按钮的名称都是上一个视图控制器的标题。Back 按钮的名字表示用户在点击了该按钮之后会返回到栈中的哪个控制器。用户可以一直向上返回，直到根控制器为止，然后就不能再继续向上返回了。因为根控制器是树状结构的根基，我们不能跳到树根外面去。

即便界面里只有一个视图控制器，也依然可以采用这种基于栈的结构来设计。比方说，有时候我们只想使用 `UINavigationController` 内置的导航栏来构建一个简单的工具，而导航栏上面也仅仅包含两个表示菜单的按钮。虽说这种情况无须利用栈结构所具备的优势，但开发者仍然要通过 `initWithRootViewController:` 方法把这个控制器设为根控制器。

## 7.2.2 推入与弹出视图控制器

开发者通过 `pushViewController:animated:` 方法推入新的视图控制器，以便向导航栈中添加新元素。每个视图控制器都提供了 `navigationController` 属性，该属性指向与这个控制器相配合的导航控制器。如果未将控制器推入导航栈，那么该属性就是 `nil`。

我们可以通过 `navigationController` 属性查出导航控制器，并在其上调用 `pushViewController:animated:` 方法，把新的视图控制器推入导航栈。调用该方法之后，新的控制器会从右侧滑入屏幕（假设调用的时候把 `animated` 参数设为了 `YES`）。这时会出现一个带左箭头的 `Back` 按钮，点击该按钮，就可以回到栈里的上一个控制器了。设置 `backIndicatorImage` 属性，即可将这个横向的 `V` 形图案换成自定义的图像。在覆写苹果公司的标准元件时一定要多加小心，请务必和《Apple Human Interface Guidelines》(HIG) 的宗旨相符。

有很多情况都需要推入新的视图。一般来说，当程序需要显示诸如细节视图等专门的视图时，或是当用户需要访问下一级文件夹、下一级配置选项时，就需要推入新的视图了。我们可以在用户点击了按钮、表格项或 `disclosure` 按钮的时候，向导航控制器的栈里推入新的视图控制器。

我们几乎没有什么理由去编写 `UINavigationController` 子类。在 `UIViewController` 的子类中，可以向导航控制器里推入新的视图控制器，也能够定制导航栏（比方说，可以设置导航栏的标题或按钮）。开发者只需把定制好的子控制器放到导航控制器里面就行了。

在大多数情况下，都无须直接访问导航控制器。但在管理导航栏的按钮、修改导航栏的样貌或用自定义的导航栏类来初始化的时候，则需要访问它。比方说，我们可以用类似下面这样的代码来直接访问 `navigationBar` 属性，并修改导航栏的样式或 `tint color`。

```
self.navigationController.navigationBar.barStyle =  
    UIBarStyleBlack;
```

请注意，在 `iOS 7` 中，苹果公司添加了 `barTintColor` 属性来表示导航栏背景的 `tint color`，而原来的 `tintColor` 属性则用来表示 `UIBarButtonItem` 的 `tint color` 了。

## 7.2.3 导航栏上的按钮

如果想添加新按钮，可以修改 `navigationItem` 属性，该属性对应的 `UINavigationController` 类用来描述显示在导航栏中的内容，其中包括左侧和右侧的按钮项（`button item`），也包括标题视图（`title view`）。下列代码可用来设置导航栏中的按钮：

```
self.navigationItem.rightBarButtonItem = [[UIBarButtonItem alloc]  
    initWithTitle:@"Action" style:UIBarButtonItemStylePlain target:self  
    action:@selector(performAction:)];
```

若想移除按钮，则将按钮项设为 `nil`。导航栏上的按钮项（`UIBarButtonItem`）并不是视图，它们是一种包含标题、样式及回调信息的类，而 `UINavigationController` 和 `UIToolbar` 则会用这个类来构建实际的按钮，并将其放入界面之中。`iOS` 系统没有提供这

样一种手段，用以访问由 `UIBarButtonItem` 及其 `UINavigationController` 所创建出来的按钮视图 (button view)。

从 iOS 5 开始，我们可以给导航栏左右两侧添加多个 `UIBarButtonItem`。把数组赋给 `navigationItem` 的 `rightBarButtonItems` 或 `leftBarButtonItems` 属性 (注意最后的 “s”) 即可：

```
self.navigationItem.rightBarButtonItems = barButtonArray;
```

## 7.2.4 延伸至屏幕边缘的布局形式

iOS 7 的设计侧重于应用程序的内容，更具体地说，就是展示给用户的内容。它移除了导航栏和其他 UI 元件的边界及阴影效果，并添加了半透明效果。这一变化会极大地影响视图布局，尤其是使用了导航栏的视图布局。

从 iOS 7 开始，所有的视图控制器都采用全屏布局 (full-screen layout)。`UIViewController` 的 `wantsFullScreenLayout` 属性已经弃用了，如果将其设为 `NO`，那么可能会导致与需求非常不相符的布局。在全屏布局模式下，视图控制器会令它的视图填满整个屏幕，并出现在半透明的系统状态栏下方。而且在默认情况下，iOS 7 会把所有 bar (栏) 都绘制成半透明的，以便显示出下方的内容。

由于应用程序的内容现在会延伸到 bar 的下方，所以我们不能再按照以前那套方式来排布视图。排布视图的时候，必须把状态栏及程序的导航栏下方那些区域考虑进来。

`UIViewController` 提供了一些新的布局属性，它们可以实现更为精细的位置控制。在子类中实现 `prefersStatusBarHidden` 方法并返回适当的布尔值，即可在视图控制器这一级别设置状态栏是否可见。开发者还可以通过一些新属性，根据当前所显示出的 bar 来设置视图的位置及大小。

通过视图控制器的 `edgesForExtendedLayout` 属性，我们可以指定控制器是否应该把视图的内容延伸到半透明的 bar 下方，并达到屏幕边缘。该属性的默认值是 `UIRectEdgeAll`，表示视图在四个边界方向上都会延伸到半透明元件的下方，如图 7-2 左侧截图所示。如果把这个属性设为 `UIRectEdgeNone`，那么内容视图的边界在碰到 bar 之后就不继续延伸，如图 7-2 右侧截图所示。默认情况下，`edgesForExtendedLayout` 属性在延展视图的时候，也会把完全不透明的 bar 考虑在内。若想排除这种 bar，则可将 `extendedLayoutIncludesOpaqueBars` 设为 `NO`。

系统状态栏和开发者自己实现的 bar (比如 navigation bar (导航栏)、toolbar (工具栏) 及 tab bar (标签栏)) 还会影响到滚动视图 (scroll view) 的布局。在默认情况下，`UIScrollView` 会自动调整 inset<sup>①</sup>，以便应对这些 bar 元件。如果不想启用这种行为，而是想手动管理 `UIScrollView` 的 inset，那么就请将 `automaticallyAdjustsScrollViewInsets` 设为 `NO`。

最后要说的是，iOS 7 还提供了 `topLayoutGuide` 及 `bottomLayoutGuide` 属性，用

① 可以理解成视图内容与视图边界之间的距离。——译者注



来协助我们排布视图的内容。这些属性指定了屏幕顶端或底端那个 bar 的边界在视图控制器的视图中处于哪个位置。它们所指的那条边界会随着屏幕上是否显示各种 bar 而有所不同：

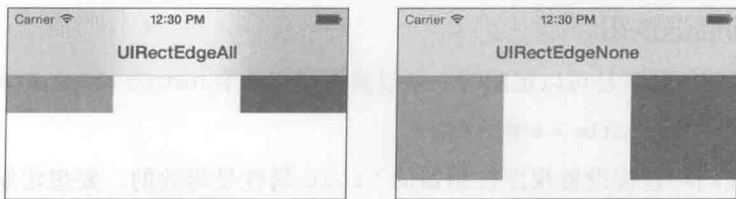


图 7-2 iOS 7 中，UIViewController 的 `edgesForExtendedLayout` 属性用来在排版时控制视图的边界位置。默认值是 `UIRectEdgeAll`，它表示视图的边界会越过半透明的 bar，如左侧截图所示。若设为 `UIRectEdgeNone`，则会令视图的边界在 bar 的边缘处停止延伸，如右侧截图所示

对于 `topLayoutGuide` 来说：

- 如果状态栏可见但没有可见的导航栏，那么就表示状态栏的底边。
- 如果有可见的导航栏，那么就表示导航栏的底边。
- 如果既没有可见的状态栏，也没有可见的导航栏，那么就表示屏幕上沿。

对于 `bottomLayoutGuide` 来说：

- 如果有可见的工具栏或标签栏，那么就表示工具栏或标签栏的顶边。
- 如果没有可见的工具栏或标签栏，那么就表示屏幕底边。

开发者可以用这些属性来创建相对的约束规则，这样的话，我们既不用考虑 `frame` 的位置，也不用预知 bar 是否可见，而是能够根据 bar 的边界来排布子视图的相对位置。在 Interface Builder (IB) 或布局代码中，可以将其同 Auto Layout 约束系统结合起来使用。在不使用 Auto Layout 的时候，可以用这些指南来进行基于 `frame` 的位置排布 (frame-based positioning)。我们可通过指南的 `length` 属性来获知相关的偏移量。

### 7.3 解决方案：UINavigationController 类

系统会用 `UINavigationController` 类的对象来生成导航栏的内容，该类存放了一些与这种对象有关的信息。`UINavigationController` 的属性包括导航栏左侧与右侧的 `UIBarButtonItem`、显示在导航栏中的标题、用来显示该标题的视图以及用于从当前视图返回到上一个视图的 `Back` 按钮。

通过这个类，可以把按钮、文本及其他 UI 对象添加到三个关键位置上，这三个位置分别是导航栏的左侧、中心及右侧。一般情况下，我们会在右侧放置普通按钮，在中间写上一些文本 (通常是 `UIViewController` 的标题)，并在左侧放置 `Back` 按钮。但开发者并不应该局限于这种布局。我们也可以在左、中 (标题区域)、右这三个位置上添加自定义的控件。比方说，可以把搜索用的文本框、分段选择控件 (segment control)、工具栏或图片等放在导

航栏中间。而且还可以把许多 `UIBarButtonItem` 放在数组里，并添加到导航栏左右两侧。`UINavigationController` 修改起来是很容易的。

### 7.3.1 标题与后退按钮

导航栏中间的标题区是可以定制的。可以像下面这样给 `navigationItem` 设置标题：

```
self.navigationItem.title = @"My Title"
```

上面这行代码与直接设置视图控制器的 `title` 属性是等效的。要想定制标题，最简单的办法应该是修改子视图控制器的 `title` 属性，而不是去修改 `navigationItem`：

```
self.title = @"Hello";
```

导航控制器会用这个 `title` 属性来构建下一个 Back 按钮的 go back (返回) 文本。如果当前控制器的标题叫作 "Hello"，而开发者又向导航栈里推入了一个新的控制器，那么，新控制器 Back 按钮上面的文本也会是 "Hello"。

你还可以把文本形式的标题改为控件之类的自定义视图。下面这段代码会添加自定义的分段选择控件，不过我们也可以添加图像视图、步进控件等其他东西：

```
self.navigationItem.titleView =
    [[UISegmentedControl alloc] initWithItems:items];
```

### 7.3.2 宏

由于创建 `UIBarButtonItem` 是一种重复性非常高的任务，所以我们可以用宏来简化它。下面这个宏能够创建基本的按钮项：

```
#define BARBUTTON(TITLE, SELECTOR) [[UIBarButtonItem alloc] \
    initWithTitle:TITLE style:UIBarButtonItemStylePlain \
    target:self action:SELECTOR]
```

若想调用上面这个宏，只需给出标题和选择子就可以了。由于每次调用宏的时候都只需指定标题和选择子这两条信息，所以这种写法要比直接创建 `UIBarButtonItem` 更容易读懂：

```
self.navigationItem.rightBarButtonItem =
    BARBUTTON(@"Push", @selector(push));
```

该版本的宏假设目标就是 "self" (自己)，这很符合常见的用法，不过，也可以对此稍加改编。下面这个宏就可以令开发者手工指定目标：

```
#define BARBUTTON_TARGET(TITLE, TARGET, SELECTOR) \
    [[UIBarButtonItem alloc] initWithTitle:TITLE \
    style:UIBarButtonItemStylePlain target:TARGET action:SELECTOR]
```

`UIBarButtonItem` 的用法会随着应用程序的特定需求而有所变化。我们很容易就能创建出一些宏，令其用苹果公司所提供的系统控件来制作 `UIBarButtonItem`，或是令其根据图片资源来制作带有图像的 `UIBarButtonItem`，还可以令其创建自定义的视图，并在其中嵌入别的控件以及非 `UIBarButtonItem` 式的按钮 (non-bar button)，然后用这种自定义的视图来制作 `UIBarButtonItem`。



解决方案 7-1 把这些技巧结合起来，实现了上下级界面之间的切换，并且示范了如何定制子控制器的标题栏以及导航控制器的 `navigationItem`。它构建了一套非常简单的界面：用户根据屏幕上列出的几个标题，自己来选择想要推入导航栈的下一个子控制器，然后按 `Push` 按钮将其推入。通过此程序，我们可以了解导航控制器的栈在一般情况下是如何增长的。

#### 解决方案 7-1 用导航控制器在上下级界面之间切换

```
// Array of strings
- (NSArray *)fooBarArray
{
    return [@"Foo*Bar*Baz*Qux" componentsSeparatedByString:@"*"];
}

// Push a new controller onto the stack
- (void)push:(id)sender
{
    NSString *newTitle =
        [self fooBarArray][seg.selectedSegmentIndex];

    UIViewController *newController =
        [[TestBedViewController alloc] init];
    newController.edgesForExtendedLayout = UIRectEdgeNone;
    newController.title = newTitle;

    [self.navigationController
     pushViewController:newController animated:YES];
}

- (void)loadView
{
    self.view = [[UIView alloc] init];
    self.view.backgroundColor = [UIColor whiteColor];

    // Establish a button to push new controllers
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Push", @selector(push:));

    // Create a segmented control to pick the next title
    seg = [[UISegmentedControl alloc] initWithItems:
        [self fooBarArray]];
    seg.selectedSegmentIndex = 0;
    [self.view addSubview:seg];
    PREPCONSTRAINTS(seg);

    UILabel *label =
        [self labelWithTitle:@"Select Title for Pushed Controller"];
    [self.view addSubview:label];
    PREPCONSTRAINTS(label);
}
```

```

id topLayoutGuide = self.topLayoutGuide;
CONSTRAIN(self.view, label, @"H:|- [label(>=0)]-|");
CONSTRAIN(self.view, seg, @"H:|- [seg(>=0)]-|");
CONSTRAIN_VIEWS(self.view,
    @"V:[topLayoutGuide]-[label]-[seg]",
    NSDictionaryOfVariableBindings(seg, label, topLayoutGuide));
}

```

### 获取解决方案代码

访问 <https://github.com/erica/iOS-7-Cookbook> 网页，并打开“C07 View Controllers”文件夹，即可找到与本章中的解决方案相对应的完整范例项目。

## 7.4 解决方案：模态界面

在使用一般的导航控制器时，用户会逐次深入每一个视图，偶尔还会停下来返回前一个视图看看。如果用户要浏览的数据与树状的视图结构相匹配，那么这种办法就比较合适。此外，我们还可以用模态的方式来显示视图控制器。

如果给视图控制器发送 `presentViewController:animated:completion:` 消息，那么该消息里所指定的那个视图控制器就会出现在屏幕上，并取得应用程序的控制权，直到我们用 `dismissViewControllerAnimated:completion:` 令其消失为止。通过这种做法，可以为应用程序创建一种不同于 alert view（警告视图，`UIAlertView`）的专用对话框。

一般情况下，我们会用模态控制器来提示用户选择某种数据，比如 `Contacts` 程序会提示用户选择联系人，`Photos` 程序会提示用户选择照片等，另外还会用模态控制器执行一些短时间的任务，例如发送电子邮件或调整程序的配置选项等。如果要执行一种与当前视图控制器的通常职责不同的短期任务，那么可以考虑使用模态控制器。

我们可以用下面几种方式切换到模态界面：

- **Slide（滑入）**——在原有视图上面滑入新的视图。
- **Flip（翻转）**——令原有视图翻转到背面，以显示新的视图。
- **Fade（淡入）**——令新的视图逐渐浮现在原有视图之上。
- **Curl（卷动）**——主视图向上卷起，露出下面的新视图。

在传给 `presentViewController:animated:completion:` 方法的那个视图控制器上面设置 `modalTransitionStyle` 属性，即可指定切换方式。标准的切换方式是 `UIModalTransitionStyleCoverVertical`，也就是把模态视图从屏幕底部向上滑入到现有的视图控制器上方。等它要消失的时候，会向下滑出屏幕。

`UIModalTransitionStyleFlipHorizontal` 会从右向左翻转当前的视图，使得用户可以看到它背后的模态视图。等模态视图要消失的时候，又会从左向右翻转回去。

`UIModalTransitionStyleCrossDissolve` 会令新视图以淡入的形式出现在原有视图上方。等到消失的时候，它会逐渐淡出，以便露出原来的视图。


`UIModalTransitionStylePartialCurl` 会把现有的内容向上卷起来（与 Maps 应用程序的做法相似），并露出主视图控制器下方的模态设置界面。

在 iPhone 和 iPod touch 上面，模态视图控制器总会占满整个屏幕。iPad 则提供了更为细致的展示方式。iPad 提供了五种展示风格，我们可以通过 `modalPresentationStyle` 属性来设置：

- **Full screen** (全屏)——这是 iPhone 默认的全屏展示风格 (`UIModalPresentationFullScreen`)，它会令新的模态视图占满整个屏幕，并覆盖在原有内容之上。如果 `modalTransitionStyle` 设为 `UIModalTransitionStylePartialCurl`，那么 `modalPresentationStyle` 只能选择 `UIModalPresentationFullScreen`，如果选了其他展示风格，那么程序会因为运行时异常 (runtime exception) 而崩溃。
- **Page sheet** (页面)——如果以页面风格 (`UIModalPresentationPageSheet`) 来展示，那么模态视图会按照竖屏状态下的宽高比来显示，所以，在设备处于竖屏状态时，模态视图控制器会填满整个屏幕，而当设备处于横屏状态时，则只会覆盖部分屏幕，就好比一张竖着的纸放到横着的屏幕里一样。
- **Form sheet** (表单)——如果以表单风格 (`UIModalPresentationFormSheet`) 来展示，那么模态视图只会覆盖屏幕中间的一小部分区域，这样的话，用户在操作模态视图中的控件时，还能尽量看到程序主视图里的内容。
- **Current context** (当前上下文)——`UIModalPresentationCurrentContext` 会把父视图控制器的 `modalPresentationStyle` 用作当前视图的展示风格。
- **Custom** (自定义)——iOS 7 添加了 Custom Transitions API，用于管理这种自定义的展示风格 (`UIModalPresentationCustom`)。

这些展示风格在横屏模式下的效果会好一些，因为此时能够体现出页面式的风格与全屏风格之间的区别。

---

 **提示** iOS 7 引入了一种模型，能够在视图控制器之间创建自定义的切换效果 (custom transition)，这些效果可以增强系统所提供的那些内置效果。自定义的切换效果提供了近乎无穷的灵活度，它可以在视图控制器之间发生切换的时候创建出新颖的动画效果。展示模态界面以及向导航控制器的栈里推入新视图时，也可以使用自定义的切换效果。

---

## 展示自定义的模态信息视图

把模态的视图控制器展示出来之后，程序的主导航路径上面就会出现分支，这个模态的新界面会获得应用程序的控制权，直到用户将其关闭为止。下面这行语句可用来展示模态控

制器：

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

展示出来的控制器可以是任意 `UIViewController` 子类的实例。假如它是个导航控制器，那么这个模态界面还可以随着用户的一系列操作而构建出自己的导航结构。开发者可以在完成块（completion block）里面编写一些执行任务的代码，这些代码会在视图控制器以动画方式展示出来之后运行。

我们应该给用户某种形式的 Done 按钮，使其可以关闭模态控制器。最简单的办法是展示一个导航控制器，并给它的 `navigationItem` 上面添加一个含有相关动作的 `UIBarButtonItem`：

```
- (IBAction)done:(id)sender
{
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

故事板简化了创建模态控制器元素的过程。我们可以向其中拖放一个导航控制器实例，这样会把配套的视图控制器也拖放进去，然后在提供好的导航栏上面添加 Done 按钮。把视图控制器的类（class）设为开发者自定义的模态控制器类型，并把 Done 按钮连接到 `done:` 方法上面。接着在 Attributes Inspector 中给导航控制器起个名字，以便在代码里使用这个标识符来加载它。

我们可以把模态组件添加到主故事板里，也可以将其创建到一份单独的文件中。解决方案 7-2 是从一份自定义的故事板文件（`Modal~DeviceType.storyboard`，其中 `DeviceType` 表示设备类型）中加载资源的，不过你也可以把那些元件放到 `MainStoryboard_DeviceType` 文件中。

解决方案 7-2 提供了创建模态元素的关键代码。我们在应用程序的主视图控制器层级中展示模态界面。这个范例程序使得用户能够从分段选择控件（segmented control）中选取切换方式及展示方式，不过在其他应用程序里，这两个选项一般都是由开发者预先选好并通过代码或 IB 来设置的。这条解决方案相当于一个工具箱，开发者可以在各个平台上面以不同的屏幕方向来测试每一种选项的效果。



**提示** 在 iOS 7 刚发布的时候，很多人都发现全屏状态下的翻转切换方式有问题，解决方案 7-2 可以重现这个问题。我们会发现，在播放动画的过程中，导航栏里的那些内容的位置并没有随着动画效果而改变，但是等动画播放完毕后，它们却会突然从上面掉下来。希望苹果公司在发行新版 iOS 时，能解决此问题。

#### 解决方案 7-2 模态控制器的展示与消失

```
// Presenting the controller
- (void)action:(id)sender
{
    // Load info controller from storyboard
```

```

UIStoryboard *storyBoard = [UIStoryboard
    storyboardWithName:
        (IS_IPAD ? @"Modal-iPad" : @"Modal-iPhone")
    bundle:[NSBundle mainBundle]];
UINavigationController *navController =
    [storyBoard instantiateViewControllerWithIdentifier:
        @"infoNavigationController"];

// Select the transition style
int styleSegment =
    [segmentedControl selectedSegmentIndex];
int transitionStyles[4] = {
    UIModalTransitionStyleCoverVertical,
    UIModalTransitionStyleCrossDissolve,
    UIModalTransitionStyleFlipHorizontal,
    UIModalTransitionStylePartialCurl};
navController.modalTransitionStyle =
    transitionStyles[styleSegment];

// Select the presentation style for iPad only
if (IS_IPAD)
{
    int presentationSegment =
        [iPadStyleControl selectedSegmentIndex];
    int presentationStyles[3] = {
        UIModalPresentationFullScreen,
        UIModalPresentationPageSheet,
        UIModalPresentationFormSheet};

    if (navController.modalTransitionStyle ==
        UIModalTransitionStylePartialCurl)
    {
        // Partial curl with any non-full-screen presentation
        // raises an exception
        navController.modalPresentationStyle =
            UIModalPresentationFullScreen;
        [iPadStyleControl setSelectedSegmentIndex:0];
    }
    else
    {
        navController.modalPresentationStyle =
            presentationStyles[presentationSegment];
    }

    [self.navigationController presentViewController:navController
        animated:YES completion:nil];
}

- (void)loadView
{
    self.view = [[UIView alloc] init];
}

```

```

self.view.backgroundColor = [UIColor whiteColor];
self.navigationItem.rightBarButtonItem =
    UIBarButtonItem(@"Action", @selector(action:));

segmentedControl =
    [[UISegmentedControl alloc] initWithItems:
        @"Slide Fade Flip Curl"
        componentsSeparatedByString:@" "];
[segmentedControl setSelectedSegmentIndex:0];
self.navigationItem.titleView = segmentedControl;

if (IS_IPAD)
{
    NSArray *presentationChoices =
        [NSArray arrayWithObjects:@"Full Screen",
            @"Page Sheet", @"Form Sheet", nil];

    iPadStyleControl =
        [[UISegmentedControl alloc] initWithItems:
            presentationChoices];

    [iPadStyleControl setSelectedSegmentIndex:0];
    [self.view addSubview:iPadStyleControl];
    PREPCONSTRAINTS(iPadStyleControl);
    CENTER_VIEW_H(self.view, iPadStyleControl);
    id topLayoutGuide = self.topLayoutGuide;
    CONSTRAIN_VIEWS(self.view,
        @"V:[topLayoutGuide]-[iPadStyleControl]",
        NSDictionaryOfVariableBindings(topLayoutGuide,
            iPadStyleControl));
}
}

```

## 7.5 解决方案：构建分栏视图控制器

分栏视图控制器 (split view controller) 很适合用来在 iPad 上面展示有层次的导航界面。这种控制器通常由左侧的目录和右侧的细节视图组成，但是该类并不局限于这种展示方式（而且苹果公司的设计指南也没有强令开发者必须采用这种展示方式）。这个类的核心概念就是它左侧的组织区域 (organizing section, 也叫 master, 主视图) 以及右侧的展示区域 (presentation section, 也叫 detail, 细节视图)，在横屏模式下，这两部分会同时出现在屏幕上，而在竖屏模式下，组织区域则可以通过 popover 的形式弹出来。（我们可以在 delegate 中实现 `splitViewController:shouldHideViewController:inOrientation:` 方法，以便修改这种默认的行为，使得分栏视图的左右两个部分都能在竖屏状态下显示出来。）

解决方案 7-3 创建了一种非常简单的分栏视图控制器，图 7-3 演示了此控制器在横屏（左侧截图）和竖屏（右侧截图）状态下的效果。用户需要在根视图的列表选取一种颜色，然后控制器会把细节视图渲染成该颜色。在横屏状态下，左右两个视图会同时显示出来，而在竖屏状态下，用户必须点击细节视图左上角的按钮，才能看到以 popover 形式弹出来的根视图。

另外，用户也可以通过滑动手势把这个视图显示出来，对于竖屏模式来说，由于 popover 显示在细节视图的前方，所以它可能会干扰到细节视图。开发者应该对此做出相应的设计。

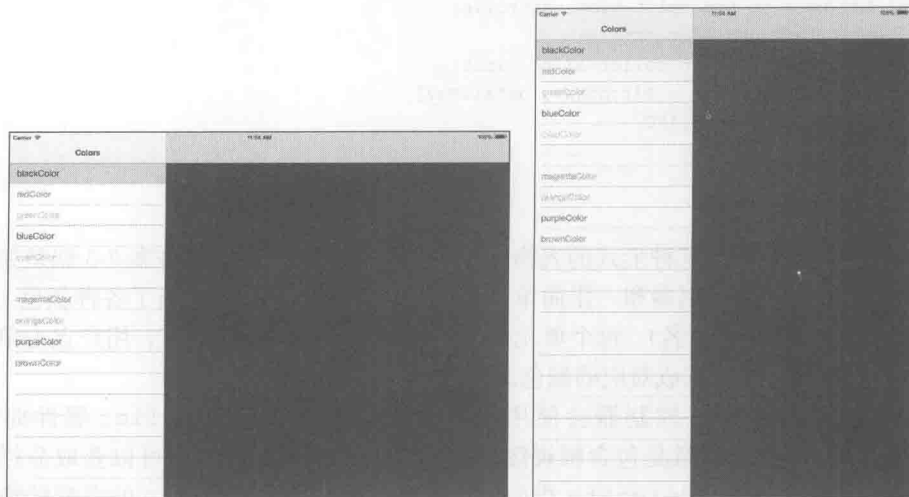


图 7-3 最简单的分栏视图控制器由左侧的组织面板和右侧的细节视图面板组成。图中看到的这个组织面板在竖屏模式下（右侧截图）通常是隐藏起来的。用户点击导航栏上的按钮之后，它会以 popover 的形式弹出来，此外，也可以用滑动手势把它显示出来

解决方案 7-3 的代码构建了三个不同的对象：主视图控制器、细节视图控制器以及分栏视图控制器，后者会拥有前两者。分栏视图控制器总是包含两个子控制器，也就是下标为 0 的主视图控制器和下标为 1 的细节视图控制器。

为了设计出风格一致的界面，我们应该把主视图控制器和细节视图控制器分别嵌套在各自的导航控制器里。对于细节控制器来说，这样做可以在竖屏模式下把按钮显示在导航栏中。下面这个方法创建了两个子视图控制器，并把它们分别嵌套在各自的导航控制器里，然后，用包含这两个导航控制器的数组来配置新创建的 UISplitViewController 对象，最后，把创建好的分栏视图控制器返回给调用者：

```
- (UISplitViewController *)splitViewController
{
    // Create the navigation-embedded root (master) view
    ColorTableViewController *rootVC =
        [[ColorTableViewController alloc] init];
    rootVC.title = @"Colors"; // make sure to set the title
    UINavigationController *rootNav =
        [[UINavigationController alloc]
         initWithRootViewController:rootVC];

    // Create the navigation-run detail view
    DetailViewController *detailVC =
        [[DetailViewController controller];
    UINavigationController *detailNav =
```

```

    [[UINavigationController alloc]
        initWithRootViewController:detailVC];

    // Add both to the split view controller
    UISplitViewController *svc =
        [[UISplitViewController alloc] init];
    svc.viewControllers = @[rootNav, detailNav];
    svc.delegate = detailVC;

    return svc;
}

```

主视图控制器一般是某种形式的表格视图控制器，比方说，解决方案 7-3 就是如此。这个解决方案里的主视图控制器和一张简单的表格差不多。表格里面列出了各种颜色（实际上就是 UIColor 的各种方法名），每个单元格的标题都是一种颜色的名字，用户点击单元格之后，右边的细节视图就会变成对应的颜色。

用户选中某颜色时，控制器会使用内置的 splitViewController 属性向细节视图发送请求。该属性指的就是包含根视图的分栏视图控制器。根视图可以获取分栏视图的 delegate（委托），这个 delegate 指向细节视图。我们把 delegate 的类型转换成细节视图控制器的类型，以便使根视图能够更为精细地控制细节视图。这个极为简单的范例程序会把用户所选单元格的文本设为细节视图的背景色。



**提示** 请务必设置好根视图控制器的 title 属性。因为在竖屏模式下，该属性的文本要出现在细节视图的按钮里。

解决方案 7-3 的 DetailViewController 类可以看作一份代码模板。它实现了一个具备基本功能的细节视图控制器，我们可以将这个控制器集成到分栏视图控制器之中。该类还实现了一对名叫 splitViewController:willHideViewController:withBarButtonItem:forPopoverController: 及 splitViewController:willShowViewController:invalidatingBarButtonItem: 的方法，并在其中分别为细节视图添加及移除了重要的 UIBarButtonItem。

### 解决方案 7-3 构建分栏视图控制器的细节视图及主视图

```

@interface DetailViewController : UIViewController
    <UIPopoverControllerDelegate, UISplitViewControllerDelegate>
@property (nonatomic, strong)
    UIPopoverController *popoverController;
@end
@implementation DetailViewController
+ (instancetype)controller
{
    DetailViewController *controller =
        [[DetailViewController alloc] init];
    controller.view.backgroundColor = [UIColor blackColor];
}

```



```

        return controller;
    }

    // Called upon going into portrait mode, hiding the normal table view
    - (void)splitViewController:(UISplitViewController*)svc
        willHideViewController:(UIViewController *)aViewController
        withBarButtonItem:(UIBarButtonItem*)barButtonItem
        forPopoverController:(UIPopoverController*)aPopoverController
    {
        barButtonItem.title = aViewController.title;
        self.navigationItem.leftBarButtonItem = barButtonItem;
        self.popoverController = aPopoverController;
    }

    // Called upon going into landscape mode
    - (void)splitViewController:(UISplitViewController*)svc
        willShowViewController:(UIViewController *)aViewController
        invalidatingBarButtonItem:(UIBarButtonItem *)barButtonItem
    {
        self.navigationItem.leftBarButtonItem = nil;
        self.popoverController = nil;
    }

    // Use this to avoid the popover hiding in portrait.
    // When omitted, you get the default behavior.
    /* - (BOOL)splitViewController:(UISplitViewController *)svc
        shouldHideViewController:(UIViewController *)vc
        inOrientation:(UIInterfaceOrientation)orientation
    {
        return NO;
    } */
@end

@interface ColorTableViewController : UITableViewController
@end

@implementation ColorTableViewController
+ (instancetype)controller
{
    ColorTableViewController *controller =
        [[ColorTableViewController alloc] init];
    controller.title = @"Colors";
    return controller;
}

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

```

```

- (NSArray *)selectors
{
    return @[@"blackColor", @"redColor", @"greenColor", @"blueColor",
            @"cyanColor", @"yellowColor", @"magentaColor", @"orangeColor",
            @"purpleColor", @"brownColor"];
}

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    return [self selectors].count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:@"generic"];
    if (!cell) cell = [[UITableViewCell alloc]
        initWithStyle:UITableViewCellStyleDefault
        reuseIdentifier:@"generic"];

    // Set title and color
    NSString *item = [self selectors][indexPath.row];
    cell.textLabel.text = item;
    cell.textLabel.textColor =
        [UIColor performSelector:NSSelectorFromString(item)
        withObject:nil];

    return cell;
}

- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    // On selection, update the main view background color
    UINavigationController *nav =
        [self.splitViewController.viewControllers lastObject];
    UIViewController *controller = [nav topViewController];
    UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexPath];
    controller.view.backgroundColor = cell.textLabel.textColor;
}

@end

```

程序切换到竖屏状态时，分栏视图控制器会把主视图控制器转换成 UIPopoverController，并把这个新的 UIPopoverController 传给细节视图控制器。细节视图控制器则要负责保留并处理这个 popover，直到界面恢复横屏状态为止。在这份模板式的类代码中，我们定义了一个强（strong）属性，用来保留竖屏状态下的 popover，使得用户可以操作它。

## 7.6 解决方案：用分栏视图及导航控制器创建通用的程序

解决方案 7-4 修改了解决方案 7-3 的分栏视图控制器，使程序可以同时适用于 iPhone 及 iPad 平台，而且令用户能够在这两个版本中执行等效的操作。为了实现这一目标，我们需要以解决方案 7-3 的代码为基础再添加一些步骤。分栏视图控制器里已有的那些功能无须移除，但是必须在好几个地方采用两套做法才行。

解决方案 7-4 用宏来判断代码是运行在 iPad 还是 iPhone 系列的设备上面。它利用 UIKit 的 user interface idiom（用户界面风格）来实现这个判断：

```
#define IS_IPAD (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad)
```

如果设备特征与 iPad 系列相符但与 iPhone 系列（例如 iPhone 或 iPod touch）不同，那么上述宏就返回 YES。苹果公司在 iOS 3.2 里首次将 iPad 定为新的硬件平台，在代码里判断 user interface idiom，就能令程序在运行的时候可以根据部署到的硬件平台来提供适当的界面。

开发 iPhone 版本的程序时，解决方案 7-3 里细节视图控制器的代码依然保持不变，但在显示的时候，则需将其推入导航栈，而不是像 iPad 上面的分栏视图那样，把细节视图与主视图并排放置。所以，此时要把导航控制器设为应用程序视窗的主视图控制器，而不是分栏视图的子控制器。程序启动时，只需执行一段简单的检测代码，就能判断出具体应该使用哪种办法了：

```
- (UINavigationController *)navWithColorTableViewController
{
    ColorTableViewController *rootVC =
        [[ColorTableViewController alloc] init];
    rootVC.title = @"Colors";
    UINavigationController *nav = [[UINavigationController alloc]
        initWithRootViewController:rootVC];
    return nav;
}

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

    UIViewController * rootVC = nil;
    if (IS_IPAD)
        rootVC = [self splitViewController];
    else
        rootVC = [self navWithColorTableViewController];

    rootVC.edgesForExtendedLayout = UIRectEdgeNone;
    window.rootViewController = rootVC;
    [window makeKeyAndVisible];
    return YES;
}
```

解决方案 7-4 还需要在选取颜色所用的那个表格视图控制器里修改两个方法。我们要做两次关键的判断，以决定是否显示 Disclosure Indicator（扩展指示器）图案以及如何应对用户对表格的点击：

解决方案 7-4 用两种方式来实现分栏视图的效果，以便制作出通用的应用程序

---

```

- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:@"generic"];
    if (!cell) cell = [[UITableViewCell alloc]
        initWithStyle:UITableViewCellStyleDefault
        reuseIdentifier:@"generic"];
    NSString *item = [self selectors][indexPath.row];
    cell.textLabel.text = item;
    cell.textLabel.textColor =
        [UIColor performSelector:NSSelectorFromString(item)
        withObject:nil];

    cell.accessoryType = IS_IPAD ?
        UITableViewCellAccessoryNone :
        UITableViewCellAccessoryDisclosureIndicator;

    return cell;
}

- (void)tableView:(UITableView *)tableView
  didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
        [tableView cellForRowAtIndexPath:indexPath];

    if (IS_IPAD)
    {
        UINavigationController *nav =
            [self.splitViewController.viewControllers lastObject];
        UIViewController *controller = [nav topViewController];
        controller.view.backgroundColor = cell.textLabel.textColor;
    }
    else
    {
        DetailViewController *controller =
            [DetailViewController controller];
        controller.view.backgroundColor = cell.textLabel.textColor;
        controller.title = cell.textLabel.text;

        [self.navigationController
            pushViewController:controller animated:YES];
    }
}

```

---

- 在 iPad 上面，当程序到达最后一层细节界面的时候，不应该显示 Disclosure Indicator 图标。而在 iPhone 上面，则需要用这种图案提示用户选择该单元格之后，会有新的视图弹出来。我们在代码中根据程序所部署到的平台来决定是否为单元格添加此图案。
- 由于 iPhone 平台不能使用分栏视图控件，所以我们必须把新的细节视图推入导航控制器的栈中。而 iPad 平台则不然，它可以直接获取现有的细节视图，并更新其背景色。

在日常开发中，这两个地方需要检测的内容也许会更加复杂，而不会像本例这样只需处理一下细节视图就行了。开发者可能还得在模型里面做检测，以判断当前是不是真到了树状结构的最底层，并据此决定是否隐藏 Disclosure Indicator 图案。还有就是，我们可能需要更新或替换细节视图控制器中的界面。

## 7.7 解决方案：标签栏

iPhone 和 iPod touch 都可以使用 UITabBarController 类，令用户能够在多个视图控制器之间切换，并且能够定制屏幕底部的标签栏。这种用法在音乐类的程序里效果最好。用户只需点击一下屏幕，就能切换到其他视图，而且还能通过 More 按钮来选择并编辑需要显示在屏幕底部标签栏里的标签（Tab）。在 iPad 上面，不应该把标签栏用作主要的设计风格，不过苹果公司也认为，在确有必要时可以一用，尤其是在分栏视图及 popover 里面。

使用标签栏的时候，不需要像使用导航栏那样把视图推入栈中，而是需要设置 viewControllers 属性，以便将一系列控制器（每个控制器可以分别是 UIViewController、UINavigationController 或其他类型的控制器）添加到标签栏里。Cocoa Touch 会完成剩下的工作。如果把 allowsCustomizing 设为 YES，那么用户就能够重新调整各个标签的顺序了。

解决方案 7-5 创建了 11 个简单的视图控制器，它们都是 BrightnessController 类的实例。该类会把背景设为指定的灰度（gray level，灰阶），在本例中，我们采用从 0% 到 100% 的 11 种灰度，相邻两种灰度之间相差 10%。图 7-4 左侧截图演示了默认模式下的界面，它会把前四个标签显示出来，而且还会显示 More 按钮。

用户可以点击 More 选项，然后点击 Edit，这样就能重新排列各标签的次序了。此时屏幕上会出现如图 7-4 右侧截图所示的配置画面。里面有 11 个视图控制器，用户可以浏览并选取自己所需的那些。请注意，在笔者编写本书时所用的 iOS 7 系统里，图 7-4 右侧截图中的导航栏并没有变成 iOS 7 的扁平式（flat）标准 UI 风格。

在整个界面中，导航栏半透明背景的 tint color 都是黑色。苹果公司提供了 UIAppearance 协议，使开发者可以修改某个类所有实例的 UI 属性。解决方案 7-5 利用这一特性把导航栏背景的 tint color 设为黑色：

```
[[UINavigationController appearance] setBarTintColor:[UIColor blackColor]];
```



从 iOS 7 开始，栏背景（比如导航栏背景）的 tint color 不再用 tint color 属性来表示。如果想设定背景的 tint color，那么需要使用 barTintColor 属性。

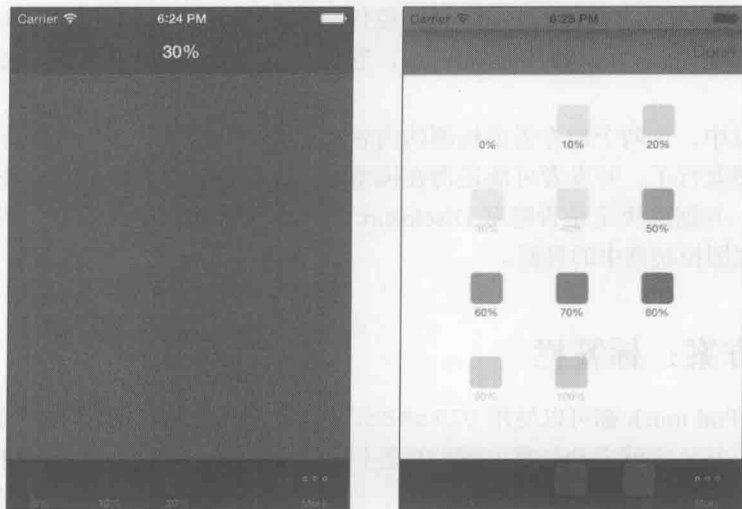


图 7-4 标签栏控制器使得用户可从屏幕底部的标签栏中选择所需的视图控制器（如左侧截图所示），而且还能在一系列可用的控制器中挑选自己喜欢的那些，用以定制标签栏（如右侧截图所示）

这条解决方案把 11 个控制器添加了两遍。第一遍是把它们赋给用户可以使用的视图控制器列表：

```
tabBarController.viewControllers = controllers;
```

第二遍是把它们赋给 customizableViewControllers，用户在定制屏幕底部的标签栏时，可以从这个列表所包含的控制器里选择自己喜欢的那些：

```
tabBarController.customizableViewControllers = controllers;
```

第二行代码是可选的，而第一行代码则必须要写。设置好这些视图控制器之后，可以把它们全都添加到可供配置的控制器列表里，也可以只添加其中一部分进去。如果不添加，那么用户点击 More 按钮之后，仍然能看到其他视图控制器，但他们无法将自己想要的控制器摆放到主标签栏中。

在 More 画面里，标签的图案会以反色效果显示。根据苹果公司的说法，这种效果是合理的。苹果公司也没打算修改这一行为。于是我们可以看到一种有趣的现象：100% 的纯黑色块在 More 画面中会变成纯白。此外，iOS 7 在渲染图标和文本的时候，会把从应用程序里继承下来的 tint color 属性当成这些物件的 tint color。

## 解决方案 7-5 创建标签栏视图控制器

```

#pragma mark - BrightnessController
@interface BrightnessController : UIViewController
@end

@implementation BrightnessController
{
    int brightness;
}

// Create a swatch for the tab icon using standard Quartz
// and UIKit image calls
- (UIImage*)buildSwatch:(int)aBrightness
{
    CGRect rect = CGRectMake(0.0f, 0.0f, 30.0f, 30.0f);
    UIGraphicsBeginImageContext(rect.size);
    UIBezierPath *path = [UIBezierPath
        bezierPathWithRoundedRect:rect cornerRadius:4.0f];
    [[[UIColor blackColor]
        colorWithAlphaComponent:(float) aBrightness / 10.0f] set];
    [path fill];

    UIImage *image = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();

    return image;
}

// The view controller consists of a background color
// and a tab bar item icon
-(BrightnessController *)initWithBrightness:(int)aBrightness
{
    self = [super init];
    brightness = aBrightness;
    self.title = [NSString stringWithFormat:@"%d%",
        brightness * 10];
    self.tabBarItem =
        [[UITabBarItem alloc] initWithTitle:self.title
            image:[self buildSwatch:brightness] tag:0];
    self.view.autoresizesSubviews = YES;
    self.view.autoresizingMask =
        UIViewAutoresizingFlexibleWidth |
        UIViewAutoresizingFlexibleHeight;
    return self;
}

// Tint the background
- (void)loadView
{
    self.view = [[UIView alloc] init];

```

```

        self.view.backgroundColor =
            [UIColor colorWithWhite:(brightness / 10.0f) alpha:1.0f];
    }

+ (id)controllerWithBrightness:(int)brightness
{
    BrightnessController *controller =
        [[BrightnessController alloc]
         initWithBrightness:brightness];
    return controller;
}

@end

#pragma mark - Application Setup
@interface TestBedAppDelegate : NSObject
    <UIApplicationDelegate, UITabBarControllerDelegate>
@property (nonatomic, strong) UIWindow *window;
@end

@implementation TestBedAppDelegate
{
    UITabBarController *tabBarController;

- (void)applicationDidFinishLaunching:(UIApplication *)application
{
    // Globally use a black tint for nav bars
    [[UINavigationController appearance]
     setBarTintColor:[UIColor blackColor]];

    // Build an array of controllers
    NSMutableArray *controllers = [NSMutableArray array];
    for (int i = 0; i <= 10; i++)
    {
        BrightnessController *controller =
            [BrightnessController controllerWithBrightness:i];
        UINavigationController *nav =
            [[UINavigationController alloc]
             initWithRootViewController:controller];
        nav.navigationBar.barStyle = UIBarStyleBlackTranslucent;
        [controllers addObject:nav];
    }

    tabBarController = [[UITabBarController alloc] init];
    tabBarController.tabBar.barTintColor = [UIColor blackColor];
    tabBarController.tabBar.translucent = NO;
    tabBarController.viewControllers = controllers;
    tabBarController.customizableViewControllers = controllers;
    tabBarController.delegate = self;

    window = [[UIWindow alloc]

```



```

        initWithFrame:[UIScreen mainScreen] bounds]];
    _window.tintColor = COOKBOOK_PURPLE_COLOR;
    tabBarController.edgesForExtendedLayout = UIRectEdgeNone;

    _window.rootViewController = tabBarController;
    [_window makeKeyAndVisible];
    return YES;
}
@end

```

## 7.8 记住标签的状态

对于 iOS 平台来说，持久是金（persistence is golden）。启动应用程序或者从暂停及中断状态继续执行程序的时候，我们应该把程序状态恢复到用户上一次离开时的样子。这样做使得用户能够继续操作上次正在操控的内容，并且能令用户界面与上次会话的界面相符。程序清单 7-1 中的范例代码可以实现这个功能。

程序清单 7-1 把标签状态保存到 `NSUserDefaults` 中

```

@implementation TestBedAppDelegate
{
    UITabBarController *tabBarController;
}

- (void)tabBarController:(UITabBarController *)tabBarController
  didEndCustomizingViewControllers:(NSArray *)viewControllers
  changed:(BOOL)changed
{
    // Collect and store the view controller order
    NSMutableArray *titles = [NSMutableArray array];
    for (UIViewController *vc in viewControllers)
        [titles addObject:vc.title];

    [[NSUserDefaults standardUserDefaults] setObject:titles
      forKey:@"tabOrder"];
    [[NSUserDefaults standardUserDefaults] synchronize];
}

- (void)tabBarController:(UITabBarController *)controller
  didSelectViewController:(UIViewController *)viewController
{
    // Store the selected tab
    NSNumber *tabNumber =
        [NSNumber numberWithInt:[controller selectedIndex]];
    [[NSUserDefaults standardUserDefaults]
      setObject:tabNumber forKey:@"selectedTab"];
    [[NSUserDefaults standardUserDefaults] synchronize];
}

```

```

- (BOOL)application: (UIApplication *)application
    didFinishLaunchingWithOptions: (NSDictionary *)launchOptions
{
    // Globally use a black tint for nav bars
    [[UINavigationController appearance]
        setBarTintColor: [UIColor blackColor]];

    NSMutableArray *controllers = [NSMutableArray array];
    NSArray *titles = [[NSUserDefaults standardUserDefaults]
        objectForKey:@"tabOrder"];

    if (titles)
    {
        // titles retrieved from user defaults
        for (NSString *theTitle in titles)
        {
            BrightnessController *controller =
                [BrightnessController controllerWithBrightness:
                    ([theTitle intValue] / 10)];
            UINavigationController *nav =
                [[UINavigationController alloc]
                    initWithRootViewController:controller];

            nav.navigationBar.barStyle = UIBarStyleBlackTranslucent;
            [controllers addObject:nav];
        }
    }
    else
    {
        // generate all new controllers
        for (int i = 0; i <= 10; i++)
        {
            BrightnessController *controller =
                [BrightnessController controllerWithBrightness:i];
            UINavigationController *nav =
                [[UINavigationController alloc]
                    initWithRootViewController:controller];

            nav.navigationBar.barStyle = UIBarStyleBlackTranslucent;
            [controllers addObject:nav];
        }
    }
    tabBarController = [[UITabBarController alloc] init];
    tabBarController.tabBar.barTintColor = [UIColor blackColor];
    tabBarController.tabBar.translucent = NO;
    tabBarController.viewControllers = controllers;
    tabBarController.customizableViewControllers = controllers;
    tabBarController.delegate = self;

    // Restore any previously selected tab
    NSNumber *tabNumber = [[NSUserDefaults standardUserDefaults]
        objectForKey:@"selectedTab"];

```

```

    if (tabNumber)
        tabBarController.selectedIndex = [tabNumber intValue];

    _window = [[UIWindow alloc]
        initWithFrame:[[UIScreen mainScreen] bounds]];
    _window.tintColor = COOKBOOK_PURPLE_COLOR;
    tabBarController.edgesForExtendedLayout = UIRectEdgeNone;

    _window.rootViewController = tabBarController;
    [_window makeKeyAndVisible];
    return YES;
}
@end

```

它修改了解决方案 7-5 中的代码，以便把当前的标签顺序以及用户目前所选的标签保存起来，当这些标签有变化时，它也会执行保存操作。用户启动应用程序之后，这段代码会搜寻上一次的配置，如果找到了，就据此配置标签栏。

为了在标签发生变化时做出响应，标签栏的 delegate 必须宣称自己遵循 UITabBarControllerDelegate 协议。范例代码用到了协议里的两个委托方法。第一个方法是 tabBarController:didEndCustomizingViewControllers:changed:，当用户通过 More > Edit 画面把标签定制好之后，可以经由这个方法所提供的数组来获知当前的各视图控制器。范例代码会获取控制器的标题（也就是 10%、20% 等文本），并且用这个信息来标识每个视图控制器的身份。

第二个委托方法叫作 tabBarController:didSelectViewController:。用户选择新的标签时，标签栏控制器会调用这个方法。范例代码可以获取到 selectedIndex 的值，以便将用户所选的控制器在当前数组内的序号保存起来。

本例通过 iOS 内置的用户默认值系统（user defaults system，NSUserDefaults）来存储这些值。这套选项系统的工作原理很像一个大型可变字典（mutable dictionary）。setObject:forKey: 方法可以给某个键（key）设置相关的值（value）：

```

[[NSUserDefaults standardUserDefaults] setObject:titles
    forKey:@"tabOrder"];

```

稍后可用 objectForKey: 方法来查询刚才设置的值：

```

NSArray *titles = [[NSUserDefaults standardUserDefaults]
    objectForKey:@"tabOrder"];

```

修改完之后，可调用 synchronize 方法，把数据同步到 NSUserDefaults 里面：

```

[[NSUserDefaults standardUserDefaults] synchronize];

```

应用程序启动时，会寻找上一次的配置信息，以获取用户所定制的标签顺序以及所选中的标签序号。如果找到了这种信息，就用它们来配置标签栏中的标签，并把上次选定的那个标签激活。由于标题里包含的信息是亮度值，所以代码要把上次保存的标题从文本变为数字，并将其除以 10，然后发送给 BrightnessController 的初始化方法。

大部分应用程序都不使用这种简单的数值系统。如果要用标题来保存标签栏上的标签顺序，那么请给各视图控制器都起个有意义的名称，并且要设计出一种方式，以便在排好顺序的一系列标签中找到某个视图控制器。



提示 也可以把各视图的标签保存到 `NSNumber` 数组中，或是采用 `NSKeyedArchiver` 类来实现，那样效果会更好。通过 `NSKeyedArchiver` 类，我们可以在程序终止的时候把视图的状态信息保存起来，以便下次启动时恢复。还有一种办法，就是使用由 iOS 6 所引入的状态保留系统。

## 7.9 解决方案：页面视图控制器

`UIPageViewController` 类可以构建出一种类似书籍的界面，并且分别用多个视图控制器来表示书中的每一页。用户可以通过滑动手势来翻页，也可以点击页面边缘，将书翻到下一页或上一页。我们可以用页面创建出类似书籍的版式（如图 7-5 左侧截图所示），也可以创建一种平坦的滚动界面（如图 7-5 右侧截图所示）。在滚动式的界面中，视图底部会出现页面指示器（page indicator）。

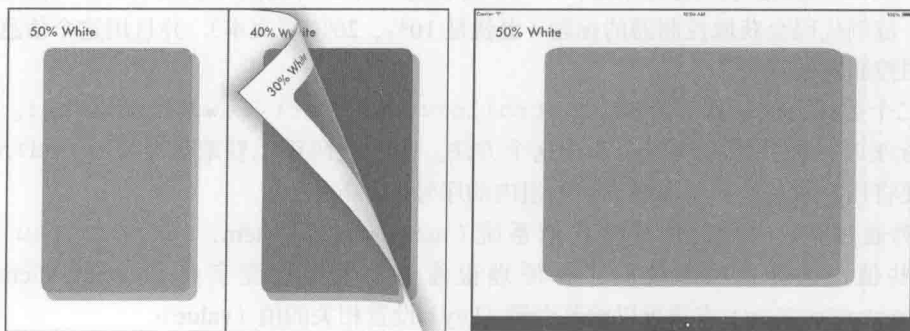


图 7-5 `UIPageViewController` 类用很多个视图控制器创建出了一本虚拟的“书”。用户可以通过翻页来查看这本书（左侧截图），也可以通过滚动来查看它（右侧截图）

控制器的页面既可以像图 7-5 这样都以一种方式来呈现，也可以拥有各自独特的用户操作体验。在 `UIPageViewController` 类中，苹果公司把播放动画效果及处理手势所需的代码全都预制好了。开发者负责提供内容，并实现委托方法以及与数据源有关的回调方法。

### 7.9.1 与书籍展示风格有关的属性

我们要用代码来定制页面视图控制器的样貌和行为。其中有一些关键属性可以指定多个页面同时显示时的方式以及每个页面背后的内容等。下面简单介绍一下由苹果公司所提

供的属性：

- `transitionStyle` 属性用来指定两个视图控制器之间的切换效果。笔者编写本书时，页面视图控制器只支持翻页（`UIPageViewControllerTransitionStylePageCurl`，如图 7-5 左侧截图所示）和滚动显示（`UIPageViewControllerTransitionStyleScroll`）两种风格。后一种风格是在 iOS 6 里引入的。
- 控制器的 `doubleSided` 属性决定了内容是出现在同一页的正反两面（如图 7-5 左侧截图所示），还是只出现在其中一面（如图 7-5 右侧截图所示）。假如要并排显示两页内容，那么就应该开启该属性。否则，有一半的页面都看不见，因为页面的“背面”无法出现在主视图空间内。书的排版由书脊位置来控制。
- 通过 `spineLocation` 属性，可以把书脊设置到屏幕左右两侧、上下两端或页面正中。三个相关的常数值分别是 `UIPageViewControllerSpineLocationMin`（对应于顶端或左侧）、`UIPageViewControllerSpineLocationMax`（对应于底端或右侧）及 `UIPageViewControllerSpineLocationMid`（对应于中心）。前两种常量会产生单页展示风格，后一种（也就是书脊在中间的那种）用来同时显示两页内容。设备方向改变时，系统会调用名为 `pageViewController:spineLocationForInterfaceOrientation:` 的委托方法，开发者应该在该方法中返回所选的 `spineLocation` 属性，令控制器可以根据当前设备方向来更新其视图。
- `navigationOrientation` 属性用来指定这本书是左右翻页还是上下翻页。`UIPageViewControllerNavigationOrientationHorizontal` 表示左右翻页 `UIPageViewControllerNavigationOrientationVertical` 表示上下翻页。对于上下翻页的书来说，页面应该是上下翻动的，而不是像平常那样左右翻动。

## 7.9.2 封装实现细节

页面视图控制器与表格视图相同的地方在于，二者都需要用委托和数据源来设定其行为以及所要展示的内容。但与表格视图不同的是，我们最好能把这些东西封装到自定义的类里，以便将细节从程序中隐藏起来。为了实现这样的页面视图，需要编写一些奇特的代码，这些代码写好之后，其可复用程度会非常高。这个封装器（`wrapper`）<sup>②</sup>可以令开发者把关注点从琐碎的代码细节转移到对具体内容的处理上面来。

按照标准的实现方式，数据源（`data source`）负责提供程序所需的页面控制器。它会根据当前给定的这个控制器返回下一个或上一个控制器。而委托则负责处理屏幕方向变更事件以及动画有关的回调，并负责设置页面视图控制器的控制器数组，该数组里含有一个或两个控制器，具体数量由视图的排版方式决定。从解决方案 7-6 可以看出，尽管实现代码有些乱，但是一旦写好之后，开发者就无须花太多时间来重复编写这部分内容了。

② 是指范例代码中的 `BookController`。——译者注

## 解决方案 7-6 创建页面视图控制器的封装类

```

// Define a custom delegate protocol for this wrapper class
@protocol BookControllerDelegate <NSObject>
- (id)viewControllerForPage:(NSInteger)pageNumber;
@optional
- (NSInteger)numberOfPages; // for scrolling layouts
- (void)bookControllerDidTurnToPage:(NSNumber *)pageNumber;
@end

// A book controller wraps the page view controller
@interface BookController : UIPageViewController
    <UIPageViewControllerDelegate, UIPageViewControllerDataSource>
+ (instancetype)bookWithDelegate:
    (id<BookControllerDelegate>)theDelegate
    style:(BookLayoutStyle)aStyle;
- (void)moveToPage:(NSInteger)requestedPage;
- (int)currentPage;

@property (nonatomic, weak)
    id <BookControllerDelegate> bookDelegate;
@property (nonatomic, assign) NSInteger pageNumber;
@property (nonatomic) BookLayoutStyle layoutStyle;
@end

#pragma mark - Book Controller
@implementation BookController

#pragma mark Utility
// Page controllers are numbered using tags
- (NSInteger)currentPage
{
    NSInteger pageCheck = ((UIViewController *)[self.viewControllers
        objectAtIndex:0]).view.tag;
    return pageCheck;
}

#pragma mark Presentation indices for page indicator (Data Source)
- (NSInteger)presentationIndexForPageViewController:
    (UIPageViewController *)pageViewController
{
    // Slightly borked in iOS 6 & 7
    // return [self currentPage];
    return 0;
}

- (NSInteger)presentationCountForPageViewController:
    (UIPageViewController *)pageViewController
{
    if (_bookDelegate && [_bookDelegate
        respondsToSelector:@selector(numberOfPages)])
        return [_bookDelegate numberOfPages];
}

```

```

    return 0;
}

#pragma mark Page Handling
// Update if you'd rather use some other decision strategy
- (BOOL)useSideBySide:(UIInterfaceOrientation)orientation
{
    BOOL isLandscape =
        UIInterfaceOrientationIsLandscape(orientation);

    // Each layout style determines whether side by side is used
    switch (_layoutStyle)
    {
        case BookLayoutStyleHorizontalScroll:
        case BookLayoutStyleVerticalScroll: return NO;
        case BookLayoutStyleFlipBook: return isLandscape;
        default: return isLandscape;
    }
}

// Update the current page, set defaults, call the delegate
- (void)updatePageTo:(NSUInteger)newPageNumber
{
    _pageNumber = newPageNumber;

    [[NSUserDefaults standardUserDefaults]
     setInteger:_pageNumber forKey:DEFAULTS_BOOKPAGE];
    [[NSUserDefaults standardUserDefaults] synchronize];

    SAFE_PERFORM_WITH_ARG(bookDelegate,
        @selector(bookControllerDidTurnToPage:),
        @(_pageNumber));
}

// Request view controller from delegate
- (UIViewController *)controllerAtPage:(NSInteger)aPageNumber
{
    if (_bookDelegate && [_bookDelegate respondsToSelector:
        @selector(viewControllerForPage:)])
    {
        UIViewController *controller =
            [_bookDelegate viewControllerForPage:aPageNumber];
        controller.view.tag = aPageNumber;
        return controller;
    }
    return nil;
}

// Update interface to the given page
- (void)fetchControllersForPage:(NSUInteger)requestedPage

```

```

orientation:(UIInterfaceOrientation)orientation
{
    BOOL sideBySide = [self useSideBySide:orientation];
    NSInteger numberOfPagesNeeded = sideBySide ? 2 : 1;
    NSInteger currentCount = self.viewControllers.count;

    NSUInteger leftPage = requestedPage;
    if (sideBySide && (leftPage % 2))
        leftPage = floor(leftPage / 2) * 2;

    // Only check against current page when count is appropriate
    if (currentCount && (currentCount == numberOfPagesNeeded))
    {
        if (_pageNumber == requestedPage) return;
        if (_pageNumber == leftPage) return;
    }

    // Decide the prevailing direction, check new page against the old
    UIPageViewControllerNavigationDirection direction =
        (requestedPage > _pageNumber) ?
        UIPageViewControllerNavigationDirectionForward :
        UIPageViewControllerNavigationDirectionReverse;

    // Update the controllers, never adding a nil result
    NSMutableArray *pageControllers = [NSMutableArray array];
    SAFE_ADD(pageControllers, [self controllerAtPage:leftPage]);
    if (sideBySide)
        SAFE_ADD(pageControllers,
            [self controllerAtPage:leftPage + 1]);
    [self setViewControllers:pageControllers
        direction:direction animated:YES completion:nil];
    [self updatePageTo:leftPage];
}

// Entry point for external move request
- (void)moveToPage:(NSUInteger)requestedPage
{
    // Thanks Dino Lupo
    [self fetchControllersForPage:requestedPage
        orientation:(UIInterfaceOrientation)
        self.interfaceOrientation];
}

#pragma mark Data Source
- (UIViewController *)pageViewController:
    (UIPageViewController *)pageViewController
    viewControllerAfterViewController:
    (UIViewController *)viewController
{
    [self updatePageTo:_pageNumber + 1];
}

```



```

        return [self controllerAtPage:(viewController.view.tag + 1)];
    }

    - (UIViewController *)pageViewController:
        (UIPageViewController *)pageViewController
        viewControllerBeforeViewController:
        (UIViewController *)viewController
    {
        [self updatePageTo:_pageNumber - 1];
        return [self controllerAtPage:(viewController.view.tag - 1)];
    }

#pragma mark Delegate Method
    - (UIPageViewControllerSpineLocation)pageViewController:
        (UIPageViewController *)pageViewController
        spineLocationForInterfaceOrientation:
        (UIInterfaceOrientation)orientation
    {
        // Always start with left or single page
        NSUInteger indexOfCurrentViewController = 0;
        if (self.viewControllers.count)
            indexOfCurrentViewController =
                ((UIViewController *) [self.viewControllers
                    objectAtIndex:0]).view.tag;
        [self fetchControllersForPage:indexOfCurrentViewController
            orientation:orientation];

        // Decide whether to present side by side
        BOOL sideBySide = [self useSideBySide:orientation];
        self.doubleSided = sideBySide;

        UIPageViewControllerSpineLocation spineLocation = sideBySide ?
            UIPageViewControllerSpineLocationMid :
            UIPageViewControllerSpineLocationMin;
        return spineLocation;
    }

    // Return a new book controller
    + (instancetype)bookWithDelegate:(id)theDelegate
    style:(BookLayoutStyle)aStyle
    {
        // Determine orientation
        UIPageViewControllerNavigationOrientation orientation =
            UIPageViewControllerNavigationOrientationHorizontal;
        if ((aStyle == BookLayoutStyleFlipBook) ||
            (aStyle == BookLayoutStyleVerticalScroll))
            orientation = UIPageViewControllerNavigationOrientationVertical;

        // Determine transitionStyle
        UIPageViewControllerTransitionStyle transitionStyle =

```

```

        UIPageViewControllerTransitionStylePageCurl;
    if ((aStyle == BookLayoutStyleHorizontalScroll) ||
        (aStyle == BookLayoutStyleVerticalScroll))
        transitionStyle = UIPageViewControllerTransitionStyleScroll;

    // Pass options as a dictionary. Keys are spine location (curl)
    // and spacing between vc's (scroll).
    BookController *bc = [[BookController alloc]
        initWithTransitionStyle:transitionStyle
        navigationOrientation:orientation
        options:nil];
    bc.layoutStyle = aStyle;
    bc.dataSource = bc;
    bc.delegate = bc;
    bc.bookDelegate = theDelegate;

    return bc;
}
@end

```

解决方案 7-6 创建了 `BookController` 类。该类会给每个页面编号，而且会把“下一页 / 上一页”这种实现细节以及处理屏幕方向变更事件所用的代码封装起来。我们自定义了名叫 `BookControllerDelegate` 的委托协议，系统向实现了该协议的对象发送 `viewControllerForPage:` 消息时，对象负责返回与给定页码相对应的控制器。这就简化了实现方式，使得调用 `BookController` 类的应用程序只需处理这一个方法就好，在该方法里，我们可以手工构建控制器，也可以从故事板中加载控制器。

在使用由解决方案 7-6 所定义的 `BookController` 类时，需要创建控制器<sup>①</sup>，并将 `BookController` 对象声明成它的子视图控制器，然后把 `BookController` 的 `view` 添加成它的子视图。把 `BookController` 添加为子控制器之后，它就能收到与屏幕方向及内存状况有关的事件了。下一个解决方案将会详细讨论视图控制器之间的这种关系。最后，我们把初始的页码设置好就行了。下面这段范例代码演示了 `BookController` 的用法：

```

- (void)viewDidLoad
{
    [super viewDidLoad];
    if (!bookController)
        bookController = [BookController bookWithDelegate:self
            style:BookLayoutStyleBook];
    bookController.view.frame = self.view.bounds;

    [self addChildViewController:bookController];
    [self.view addSubview:bookController.view];
    [bookController didMoveToParentViewController:self];

    [bookController moveToPage:0];
}

```

① 这个控制器相当于解决方案 7-6 中的 `TestBedViewController`。——译者注

创建 `BookController` 的那个便捷方法还可以接受第二个参数，即书的样式。解决方案 7-6 给开发者提供了四种构建电子书所用的样式：一种是左右翻页的传统样式，还有一种是上下翻页的书，另外两种是滚动展示风格：

```
typedef enum
{
    BookLayoutStyleBook, // side by side in landscape
    BookLayoutStyleFlipBook, // side by side in portrait
    BookLayoutStyleHorizontalScroll,
    BookLayoutStyleVerticalScroll,
} BookLayoutStyle;
```

在竖屏模式下，对于标准样式的书，其书脊是竖着摆放在屏幕左侧的，而在横屏模式下，书脊则会竖着摆在屏幕正中，这样就会并排显示出两页内容。这是西方书籍的标准翻页方式，用户可以从左向右或从右向左翻页。

而“翻转”形式的书，其书脊则是横着摆放的。在横屏模式下，书脊横放于屏幕顶端，每次只能显示一页内容。而在竖屏模式下，书脊则横着摆在屏幕中央，这样就能显示出上下两页内容了。

剩下的两种滚动展示风格可以令用户通过水平滚动或垂直滚动的方式来查看不同的页面。如果使用了滚动风格的版式，那么就不能同时显示两页内容了。

我们可以在 `viewWillDisappear` 方法中清理 `BookController`，将其从上级视图里移除：

```
-(void)viewWillDisappear:(BOOL)animated
{
    [super viewWillDisappear:animated];
    [bookController willMoveToParentViewController:nil];
    [bookController.view removeFromSuperview];
    [bookController removeFromParentViewController];
}
```

### 7.9.3 范例代码详解

为了处理与委托及数据源有关的任务，解决方案 7-6 给每个视图控制器都编了页码，并将其设为 `tag`。通过这个页码，我们可以知道当前显示出来的是哪一页，而且还能够命令 `BookControllerDelegate` 去制作接下来要显示的视图控制器。

页面控制器本身的 `viewControllers` 数组里总是存有 0 个、一个或两个页面。0 个页面表明控制器还没有配置好。一个页面适用于书脊在屏幕边缘时的情况，而两个页面则适用于书脊在屏幕正中时的情况。假如页面数量与书脊的位置不相匹配，那么程序在运行的时候就会出错。

展示在各个页面里的控制器是由两个数据源方法提供的，`BookController` 类分别实现了这两个方法表示上一页和下一页的回调方法。在页面控制器的传统实现方式中，我们用控制器之间的前后关系来描述它们，而不采用页码来描述。但这条解决方案所编写的 `BookController` 类会把这种关系替换成简单的数字，并且会向 `Book`

ControllerDelegate 查询与某个页码相对应的页面。

`useSideBySide`: 方法会根据屏幕方向来决定书脊的位置, 并决定屏幕上同时可以显示几个控制器。本条解决方案所实现的代码会在横屏模式下并排显示两页, 而在竖屏模式下只显示一页。你也可以根据自己的程序来修改这段代码。比方说, 你可能觉得在 iPhone 上面无论横屏还是竖屏, 都只应该显示一页才对, 这样可以令文本看起来更加清晰。

解决方案 7-6 提供了两种页面控制方式, 分别供用户与应用程序执行翻页操作。用户可以用滑动手势翻页, 也可以通过点击屏幕来翻页, 而应用程序则可以发送 `moveToPage`: 请求。这样一来, 我们就能在 `UIPageViewController` 的手势识别器之外, 多添加一种控制翻页的方式。

翻页方向是通过对比新旧两页的页码来确定的。本条解决方案按照西方书籍的翻页方式来计算, 也就是说, 右侧页面的页码较大, 向左翻可以看到后面的内容。对于中东及亚洲国家的书籍来说, 我们可以修改相关的代码。

解决方案 7-6 也采用 `NSUserDefaults` 来存放当前的页面, 以便在应用程序重新启动时恢复到该页。当用户翻到某个页面时, 它还会通知委托。

## 7.9.4 构建界面索引

在滚动排版模式下, 我们可以指定页面控制器的索引 `index` (以便令用户能够通过 `Page Control`(页面控制控件) 来查看各页面)。凡是采用滚动排版风格 (`UIPageViewControllerTransitionStyleScroll`) 的电子书都可以实现两个数据源方法。`iOS` 会用它们来构建屏幕底部的 `Page Control` 指示器, 如图 7-5 右侧截图所示。

从下面这段代码中可以发现, 这两个方法写得不太明晰:

```
- (NSInteger)presentationIndexForPageViewController:
    (UIPageViewController *)pageViewController
{
    // Slightly borked in iOS 6 & 7
    // return [self currentPage];
    return 0;
}


- (NSInteger)presentationCountForPageViewController:
    (UIPageViewController *)pageViewController
{
    if (bookDelegate &&
        [bookDelegate respondsToSelector:@selector(numberOfPages)])
        return [bookDelegate numberOfPages];

    return 0;
}
```

苹果公司的文档里说, `presentationIndexForPageViewController` 应该返回当前所选条目的索引, 但这样做会导致程序的行为发生混乱 (并使程序崩溃)。把 0 当作界面索

引 (presentation index), 把页面总数当作界面 (presentation count)<sup>①</sup>, 这样就可以制作出最为稳定的页面指示器了。我们把页面数量留给 BookControllerDelegate 来决定, 该协议里有个名为 numberOfPages 的可选方法。

请注意, 页面数量与 count 之间, 以及当前页数与 index 之间, 不一定非得是一比一的关系。如果一本书的页数较多, 那么可以把这个数与某数相除, 将其变小, 使得 Page Control 上面的每个圆点都能表示 5 页或 10 页书, 在翻书过程中, 页码和 Page Control 上面的圆点不一定要完全精确地对应起来。

 **提示** 苹果公司允许开发者访问 UIPageViewController 的手势识别器 (gesture recognizer), 以便根据触摸点在页面上的位置来实现触摸式翻页, 或是禁用这种翻页方式。但笔者并不推荐你去访问手势识别器。首先, 并不是所有基于触摸的控制器都支持这种做法。其次, 添加与手势识别器有关的委托方法可能会令应用程序变得不够稳定。

## 7.10 解决方案：自定义的容器

苹果公司的分栏视图控制器是个具有创新意义的控制器, 它引入了“同屏显示多个控制器”这一概念。在出现分栏视图之前, 我们只能把多个视图同时放在一个控制器中管理。而有了分栏视图之后, 多个控制器就能在屏幕上共存了, 它们可以各自独立地对屏幕方向变更事件及内存事件作出响应。

苹果公司在 iOS 5 SDK 中, 公布了这种多控制器范式 (multiple-controller paradigm), 它使得开发者可以设计上级控制器, 并向其中添加子控制器。系统会根据需要, 把事件从上级控制器传给子控制器。这样一来, 我们就能在标签栏和导航控制器等苹果公司内置的标准容器之外, 构建出一些自定义的容器。

解决方案 7-7 构建了一种可以复用的容器, 它能够包含一个或两个子控制器。如果开发者在构建时为其加载了两个子视图控制器, 那么用户就可以将视图从一面翻转到另一面了。它内置了很多条件判断语句。这是因为, 该类既可以用作独立的视图控制器, 也可以把自身当成子视图控制器, 还可以充当模态视图控制器。我们现在考虑下面几种情况。

我们可以像使用导航控制器那样, 直接创建这种 FlipViewController, 并把它设为主窗口的根视图控制器。在这种情况下, 它和别的视图层级之间没有任何关系, 它只需要管理好自己的子控制器就行了。另外, 开发者也可以把它用作其他容器的子控制器。比方说, 可以把它当作 UITabBarController 及 UISplitViewController 等容器的子控制器。在这种情况下, 它对于自身的那些子控制器来说是上级控制器; 而对于包含它的容器来说, 则又成了子控制器。最后, 开发者还可以直接把控制器展示出来。FlipViewController 类必须在这三种情况下都能稳定运作才行。因此, 这个控制器有两项任务要完成。第一, 它必须使用

① 是指页面指示器所表示的页面总数, 而界面索引则是指当前所选页面在页面指示器中的对应位置。——译者注

标准的 UIKit 调用来管理自己的子控制器。第二，它还要注意自己应该如何与视图层级相互协作。解决方案 7-7 会向该类里添加导航栏，以便在其上放置供终端用户使用的 Done 按钮。

#### 解决方案 7-7 创建视图控制器容器

```
- (void)viewDidDisappear:(BOOL)animated
{
    [super viewDidDisappear:animated];
    if (!controllers.count)
    {
        NSLog(@"Error: No root view controller");
        return;
    }

    // Clean up the child view controller
    UIViewController *currentController =
        (UIViewController *)controllers[0];
    [currentController willMoveToParentViewController:nil];
    [currentController.view removeFromSuperview];
    [currentController removeFromParentViewController];
}

- (void)flip:(id)sender
{
    // Please call only with two controllers
    if (controllers.count < 2) return;

    // Determine which item is front, which is back
    UIViewController *front = (UIViewController *)controllers[0];
    UIViewController *back = (UIViewController *)controllers[1];

    // Select the transition direction
    UIViewAnimationOptions transition = reversedOrder ?
        UIViewAnimationOptionTransitionFlipFromLeft :
        UIViewAnimationOptionTransitionFlipFromRight;

    // Hide the info button until after the flip
    infoButton.alpha = 0.0f;

    // Prepare the front for removal, the back for adding
    [front willMoveToParentViewController:nil];
    [self addChildViewController:back];

    // Perform the transition
    [self transitionFromViewController: front
     toViewController:back duration:0.5f options:transition
     animations:nil completion:^(BOOL done){

        // Bring the Info button back into view
        [self.view bringSubviewToFront:infoButton];
    }];
}
```

```

[UIView animateWithDuration:0.3f animations:^( ){
    infoButton.alpha = 1.0f;
}];

// Finish up transition
[front removeFromParentViewController];
[back didMoveToParentViewController:self];

reversedOrder = !reversedOrder;
controllers = @[back, front];
}];
}

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    if (!controllers.count)
    {
        NSLog(@"Error: No root view controller");
        return;
    }

    UIViewController *front = controllers[0];
    UIViewController *back = nil;
    if (controllers.count > 1) back = controllers[1];

    [self addChildViewController:front];
    [self.view addSubview:front.view];
    [front didMoveToParentViewController:self];

    // Check for presentation and for "flippability"
    BOOL isPresented = self.isBeingPresented;

    // Clean up instance if re-use
    if (navbar || infoButton)
    {
        [navbar removeFromSuperview];
        [infoButton removeFromSuperview];
        navbar = nil;
    }

    // When presented, add a custom navigation bar.
    // iPhone navbar height must consider status bar.
    CGFloat navbarHeight = IS_IPHONE ? 64.0 : 44.0;
    if (isPresented)
    {
        navbar = [[UINavigationController alloc] init];
        [self.view addSubview:navbar];
        PREPCONSTRAINTS(navbar);
        ALIGN_VIEW_TOP(self.view, navbar);
    }
}

```

```

        ALIGN_VIEW_LEFT(self.view, navbar);
        ALIGN_VIEW_RIGHT(self.view, navbar);
        CONSTRAIN_HEIGHT(navbar, navbarHeight);
    }

    // Right button is Done when VC is presented
    self.navigationItem.leftBarButtonItem = nil;
    self.navigationItem.rightBarButtonItem = isPresented ?
        UIBarButtonItem(UIBarButtonSystemItemDone,
            @selector(done:)) : nil;

    // Populate the navigation bar
    if (navbar)
        [navbar setItems:@[self.navigationItem] animated:NO];

    // Size the child VC view(s)
    CGFloat verticalOffset =
        (navbar != nil) ? navbarHeight : 0.0f;
    CGRect destFrame = CGRectMake(0.0f, verticalOffset,
        self.view.frame.size.width,
        self.view.frame.size.height - verticalOffset);
    front.view.frame = destFrame;
    back.view.frame = destFrame;

    // Set up info button
    if (controllers.count < 2) return; // our work is done here

    // Create the "i" button
    infoButton = [UIButton buttonWithType:UIButtonTypeInfoLight];
    infoButton.tintColor = [UIColor whiteColor];
    [infoButton addTarget:self action:@selector(flip:)
        forControlEvents:UIControlEventTouchUpInside];

    // Place "i" button at bottom right of view
    [self.view addSubview:infoButton];
    PREPCONSTRAINTS(infoButton);
    ALIGN_VIEW_RIGHT_CONSTANT(self.view, infoButton,
        -infoButton.frame.size.width);
    ALIGN_VIEW_BOTTOM_CONSTANT(self.view, infoButton,
        -infoButton.frame.size.height);
}
@end

```

### 7.10.1 添加与移除子视图控制器

在最简单的情况下，添加子视图控制器需要执行下列三步：

1. 调用上级控制器的 `addChildViewController:` 方法，并将子控制器作为参数传入（例如，`[self addChildViewController:childvc]`）。
2. 把子控制器的视图添加成上级视图的子视图（例如，`[self.view addSubview:`



childvc.view]】)。

3. 用适当的参数调用子控制器的 `didMoveToParentViewController:` 方法 (例如, `[childvc didMoveToParentViewController:self]`)。

若想移除子视图控制器, 则需执行下面三个步骤 (基本上就是把上述三个步骤反过来做一遍):

1. 以 `nil` 为参数, 调用子控制器的 `willMoveToParentViewController:` 方法 (例如, `[childvc willMoveToParentViewController:nil]`)。

2. 移除子控制器的视图 (例如, `[childvc.view removeFromSuperview]`)。

3. 调用子控制器的 `removeFromParentViewController` 方法 (例如, `[childvc removeFromParentViewController]`)。

### 7.10.2 视图控制器之间的切换效果

UIKit 提供了一种简单的方式, 可以在切换子视图控制器时, 用动画来表现视图样貌的变化过程。开发者需要提供源视图控制器、目标视图控制器以及动画效果的持续时间。此外也可以指定切换动画的类型。可供选用的类型包括翻页 (`page curl`)、融入 (`dissolve`) 及翻转 (`flip`)。下面这个方法会用简单的翻页动画来表现两个视图控制器之间的切换过程:

```
- (void)action:(id)sender
{
    [redController willMoveToParentViewController:nil];
    [self addChildViewController:blueController];

    [self transitionFromViewController:redController
        toViewController:blueController
        duration:1.0f
        options:UIViewAnimationOptionLayoutSubviews |
            UIViewAnimationOptionTransitionCurlUp
        animations:^(void){}
        completion:^(BOOL finished){
            [redController.view removeFromSuperview];
            [self.view addSubview:blueController.view];

            [redController removeFromParentViewController];
            [blueController didMoveToParentViewController:self];
        }
    ];
}
```

若想表现 `UIView` 的属性变化, 我们不一定要在 `transitionFromViewController` 方法中使用系统内置的切换效果。例如, 下面这个方法将会重新设置 `redController` 的中心点, 并令其从屏幕中淡出, 同时令 `blueController` 逐渐显示出来。对于我们在 `animations:` 块中所修改的那几个 `UIView` 属性来说, 其变更过程都可以用动画效果表示:

```

- (void)action:(id)sender
{
    [redController willMoveToParentViewController:nil];
    [self addChildViewController:blueController];

    blueController.view.alpha = 0.0f;
    [self transitionFromViewController:redController
        toViewController:blueController
        duration:2.0f
        options:UIViewAnimationOptionLayoutSubviews
        animations:^(void){
            redController.view.center = CGPointMake(0.0f, 0.0f);
            redController.view.alpha = 0.0f;
            blueController.view.alpha = 1.0f;
        }
        completion:^(BOOL finished){
            [redController.view removeFromSuperview];
            [self.view addSubview:blueController.view];

            [redController removeFromParentViewController];
            [blueController didMoveToParentViewController:self];
        }
    ];
}

```

在系统内置的切换效果和自定义的视图动画之间，我们只能选择其中一种。要么在 `options` 参数中设定内置的切换选项，要么在自编的 `animations` 块里面修改视图的特征。若两种方式并用，则会产生冲突，读者很容易就能验证这一点。在 `completion` 块中，我们移除旧的视图，并安排好新的视图。

虽说这里讲的效果实现起来很简单，但是这种切换方式却不应该和 `Core Animation` 相搭配。如果要在视图控制器的切换过程中添加 `Core Animation` 效果，那么可以考虑采用自定义的 `segue` 来实现。下一条解决方案将会讲解 `segue`。

除了上面提到的两种办法之外，在 `iOS 7` 中，还有一种办法也能实现 `UIViewController` 之间的切换动画，就是采用自定义的切换（`custom transition`）API。这套 API 令开发者可以创建出复杂的动画效果，而且还能与用户动态地交互。

## 7.11 解决方案：segue

使用故事板的时候，`IB` 提供了一系列标准的 `segue`，用以在视图控制器之间切换。前面我们制作了自定义容器，现在来制作与之相伴的自定义 `segue`。标签栏和导航控制器都能提供一种独特的方式，用来在各种子视图控制器之间切换，与之类似，我们也可以构建自定义的 `segue`，以便为自己的类制作特有的切换动画。

由于 `IB` 并未对自定义容器及其自定义 `segue` 提供太多的支持，所以我们最好是通过编程来研发自己的 `segue` 效果。下面这段代码可以实现两个视图控制器之间的切换：

```

// Informal custom delegate method
- (void)segueDidComplete
{
    // Retrieve the two vc's
    UIViewController *source =
        [childControllers objectAtIndex:vcIndex];
    UIViewController *destination =
        [childControllers objectAtIndex:nextIndex];

    // Reparent as needed
    [destination didMoveToParentViewController:self];
    [source removeFromParentViewController];

    // Update the bookkeeping
    vcIndex = nextIndex;
    pageControl.currentPage = vcIndex;
}

// Transition to new view using custom segue
- (void)switchToView:(int)newIndex
    goingForward:(BOOL)goesForward
{
    if (vcIndex == newIndex) return;
    nextIndex = newIndex;

    // Segue to the new controller
    UIViewController *source =
        [childControllers objectAtIndex:vcIndex];
    UIViewController *destination =
        [childControllers objectAtIndex:newIndex];
    // Start the reparenting process
    [source willMoveToParentViewController:nil];
    [self addChildViewController:destination];

    RotatingSegue *segue = [[RotatingSegue alloc]
        initWithIdentifier:@"segue"
        source:source destination:destination];
    segue.goesForward = goesForward;
    segue.delegate = self;
    [segue perform];
}

```

上述代码首先找出作为“源”和“目标”的那两个子控制器，然后构建 segue 并设置其参数，最后执行此 segue。解决方案 7-8 演示了该 segue 的构建方式。在本例中，我们用立方体的旋转来表示两个视图之间的切换。图 7-6 演示了由这个 segue 所制作出来的动画效果。

segue 的 goesForward 属性决定了虚拟的立方体是向右转还是向左转。从排布子视图控制器所用的那段代码可以看出，本例使用了四个视图控制器，这是由立方体这种隐喻的特性所决定的，而不是说代码里必须使用四个控制器，你也可以使用任意数量的子控制器。我

们很容易就能用这段代码来展示三个或七个子控制器，但是这种三个侧面的立方体或七个侧面的立方体会令用户觉得不真实。如果想增加（或减少）这个多面体的侧面数量，那么就要调整 segue 动画的几何特征，我们要用  $n$  个侧面的多面体来替换本例所使用的立方体。

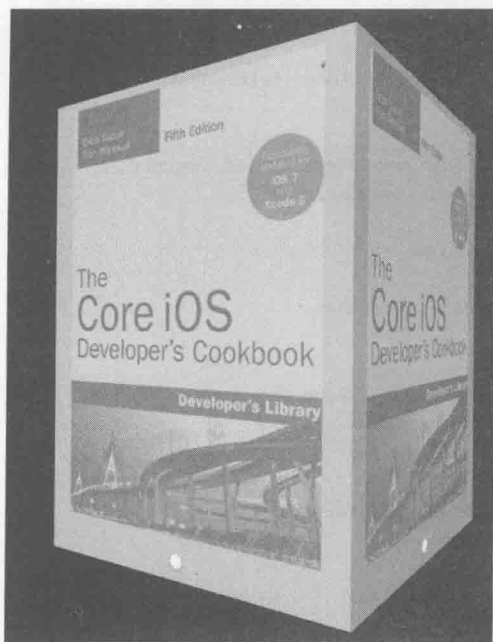


图 7-6 通过自定义的 segue，我们可以为自定义容器创建出视觉隐喻（visual metaphor）。解决方案 7-8 构建了由视图控制器所组成的“立方体”，它可以从一个视图控制器旋转到下一个控制器。每个控制器上面的 UISwitch 控件用来修改图案的 alpha 值，令其在完全不透明与半透明之间切换

#### 解决方案 7-8 创建自定义的 segue 动画，表现视图控制器之间的切换过程

```
@implementation RotatingSegue
{
    CALayer *transformationLayer;
    UIView __weak *hostView;
}

// Return a shot of the given view
- (UIImage *)screenShot:(UIView *)aView
{
    // Arbitrarily dims to 40%. Adjust as desired.
    UIGraphicsBeginImageContext(hostView.frame.size);
    [aView.layer renderInContext:UIGraphicsGetCurrentContext()];
    UIImage *image =
        UIGraphicsGetImageFromCurrentImageContext();
    CGContextSetRGBFillColor(UIGraphicsGetCurrentContext(),
```

```

    0, 0, 0, 0.4f);
    CGContextFillRect(UIGraphicsGetCurrentContext(),
        hostView.frame);
    UIGraphicsEndImageContext();
    return image;
}

// Return a layer with the view contents
- (CALayer *)createLayerFromView:(UIView *)aView
    transform:(CATransform3D)transform
{
    CALayer *imageLayer = [CALayer layer];
    imageLayer.anchorPoint = CGPointMake(1.0f, 1.0f);
    imageLayer.frame = (CGRect){.size = hostView.frame.size};
    imageLayer.transform = transform;
    UIImage *shot = [self screenshot:aView];
    imageLayer.contents = (__bridge id) shot.CGImage;

    return imageLayer;
}

// On starting the animation, remove the source view
- (void)animationDidStart:(CAAnimation *)animation
{
    UIViewController *source =
        (UIViewController *) super.sourceViewController;
    [source.view removeFromSuperview];
}

// On completing the animation, add the destination view,
// remove the animation, and ping the delegate
- (void)animationDidStop:(CAAnimation *)animation
    finished:(BOOL)finished
{
    UIViewController *dest =
        (UIViewController *) super.destinationViewController;
    [hostView addSubview:dest.view];
    [transformationLayer removeFromSuperlayer];
    if (_delegate &&
        [_delegate respondsToSelector:
            @selector(segueDidComplete)])
    {
        [_delegate segueDidComplete];
    }
}

// Perform the animation
- (void)animateWithDuration:(CGFloat)aDuration
{
    CAAnimationGroup *group = [CAAnimationGroup animation];
    group.delegate = self;

```

```

group.duration = aDuration;

CGFloat halfWidth = hostView.frame.size.width / 2.0f;
float multiplier = goesForward ? -1.0f : 1.0f;

// Set the x, y, and z animations
CABasicAnimation *translationX = [CABasicAnimation
    animationWithKeyPath:@"sublayerTransform.translation.x"];
translationX.toValue =
    [NSNumber numberWithFloat:multiplier * halfWidth];

CABasicAnimation *translationZ = [CABasicAnimation
    animationWithKeyPath:@"sublayerTransform.translation.z"];
translationZ.toValue = [NSNumber numberWithFloat:-halfWidth];

CABasicAnimation *rotationY = [CABasicAnimation
    animationWithKeyPath:@"sublayerTransform.rotation.y"];
rotationY.toValue =
    [NSNumber numberWithFloat: multiplier * M_PI_2];

// Set the animation group
group.animations = [NSArray arrayWithObjects:
    rotationY, translationX, translationZ, nil];
group.fillMode = kCAFillModeForwards;
group.removedOnCompletion = NO;

// Perform the animation
[CATransaction flush];
[transformationLayer addAnimation:group forKey:kAnimationKey];
}

- (void)constructRotationLayer
{
    UIViewController *source =
        (UIViewController *) super.sourceViewController;
    UIViewController *dest =
        (UIViewController *) super.destinationViewController;
    hostView = source.view.superview;

    // Build a new layer for the transformation
    transformationLayer = [CALayer layer];
    transformationLayer.frame = hostView.bounds;
    transformationLayer.anchorPoint = CGPointMake(0.5f, 0.5f);
    CATransform3D sublayerTransform = CATransform3DIdentity;
    sublayerTransform.m34 = 1.0 / -1000;
    [transformationLayer setSublayerTransform:sublayerTransform];
    [hostView.layer addSublayer:transformationLayer];

    // Add the source view, which is in front:
    CATransform3D transform = CATransform3DMakeIdentity;

```

```

[transformationLayer addSublayer:
 [self createLayerFromView:source.view
  transform:transform]];

// Prepare the destination view either to the right or left
// at a 90/270 degree angle off the main
transform = CATransform3DRotate(transform, M_PI_2, 0, 1, 0);
transform = CATransform3DTranslate(transform,
  hostView.frame.size.width, 0, 0);
if (!goesForward)
{
  transform =
    CATransform3DRotate(transform, M_PI_2, 0, 1, 0);
  transform =
    CATransform3DTranslate(transform,
    hostView.frame.size.width, 0, 0);
  transform =
    CATransform3DRotate(transform, M_PI_2, 0, 1, 0);
  transform =
    CATransform3DTranslate(transform,
    hostView.frame.size.width, 0, 0);
}
[transformationLayer addSublayer:
 [self createLayerFromView:dest.view
  transform:transform]];
}

// Standard UIStoryboardSegue perform
- (void)perform
{
  [self constructRotationLayer];
  [self animateWithDuration:0.5f];
}
@end

```

## segue 与 IB

从 iOS 6 SDK 开始, 开发者就可以在故事板中运用自定义的 segue 了。我们需要把这些 segue 绑定到某些动作项 (action item) 上面, 例如按钮或 UIBarButtonItem 上面, 或绑定到那种可以执行类似操作的元件上面。图 7-7 演示了 IB 所列出的自定义 segue。名为 rotating 的那个 segue 是我们在解决方案 7-8 中编写的。

此外, segue 还可以回退 (unwind)。系统可以用开发者所提供的自定义 segue, 由新的视图控制器退到它逻辑上的父控制器 (logical parent)。想实现此功能, 需要编写下面几个方法:

- 在 canPerformUnwindSegueAction:fromViewController:withSender: 方法里指定是否可以执行回退。
- 在 viewControllerForUnwindSegueAction:fromViewController:withSender: 方法中返回将要回退到的那个视图控制器。

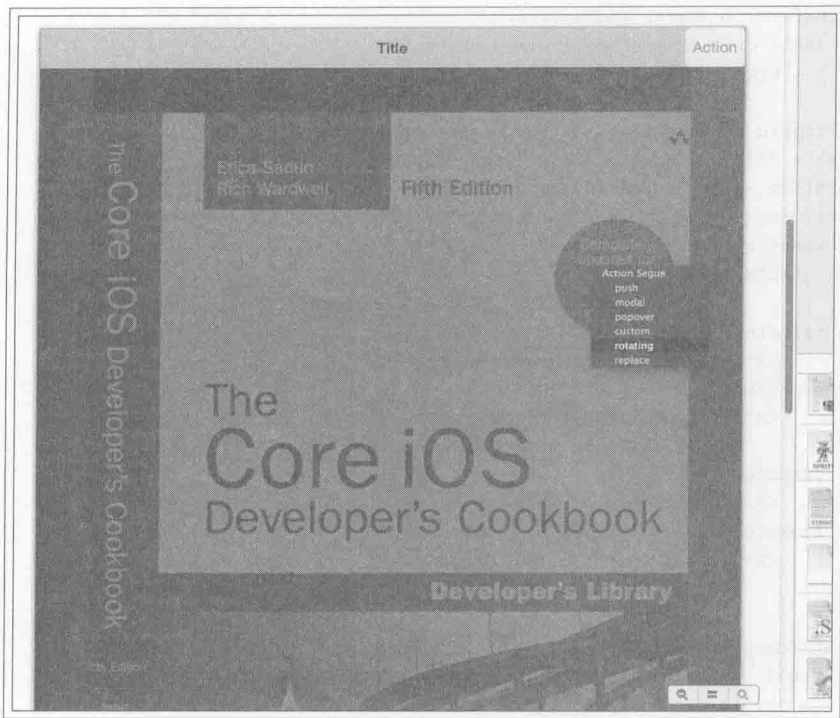


图 7-7 开发者可在故事板里运用自定义的 segue。IB 会扫描 UIStoryboardSegue 的子类。此处，IB 会列出我们自制的名为 rotating 的 segue 以及由系统提供的那些 segue

- 经由 segueForUnwindingToViewController:fromViewController:identifier: 方法提供回退时所需的 segue 实例。一般来说，此 segue 播放动画的方向应该和原来那个 segue 相反。

最后，我们可以通过实现 shouldPerformSegueWithIdentifier:sender: 方法来启用或禁用 segue。开发者可以返回 YES 或 NO，用以表明是否需要处理指定的 segue。

## 7.12 小结

本章演示了很多种视图控制器类。读者学到了如何用它们来展示视图，以及怎样为用户提供与硬件设备相符的导航方式。大家还看到了如何在既满足应用程序需求又遵循相关平台 HIG 规范的前提下，用这些类来拓展虚拟的操作空间并创建多页界面。开始阅读下一章之前，请先回顾下面几个问题：

- 用导航树构建分层的界面。在浏览文件结构或构建由设置选项所组成的树状结构时，导航树是很有用的。如果要显示细节视图或包含许多选项的视图，那么可以考虑把这个新视图控制器推入导航栈，或用分栏视图将其直接显示出来。



- 在完全符合苹果公司《HIG》(人机界面指南)的前提下,不妨试着以非常规的方式去使用常规的UI元件。我们可以彻底抛开 UINavigationController 类的导航功能,而以一种新颖的方式来使用它。开发者可以按照自己的需要去使用这些工具。
- 善用持久化机制。用户下次启动程序的时候,其 GUI 状态应该和上次离开时相同。NSUserDefaults 提供了一套内置的信息保存系统,使开发者可以在下次启动程序时恢复它们。我们可以用 NSUserDefaults 里的信息来重建界面,使其恢复到上次的状态。iOS 6 引入了 State Preservation and Restoration API,它提供了另一种方式,可用来保存大部分 UI 状态。
- 把程序做得通用一些。我们所写的程序要能自动适应各种硬件设备,而不是在所有设备上都只展示 iPhone 或 iPad 样式的界面。本章简述了一些在程序运行时检测设备特征所用的办法以及一些更新界面所用的技巧,读者可以由此为基础来扩充这些代码,并把它们运用到更为复杂的情境中去。要想把程序设计得通用一些,不能只考虑拉伸视图或加载不同的图案及 XIB 文件等手段,还应该考虑到用户会以什么样的方式来操作程序,以及程序在这种硬件平台上应该显示出什么风格的界面。
- 操作自定义的容器时,可以考虑直接使用故事板。这样的话,我们就不用再构建并保留一个数组,然后把所有控制器全都放到里面了。开发者可通过故事板直接访问这些元件。与本章新编的那个页面视图控制器类一样,我们应该等程序需要用到相关控制器的时候,再去加载它。

## 常用的控制器

iOS SDK 内置了很多种由系统所提供的控制器，开发者可在日常编码工作中使用它们。本章要介绍其中最常用的几种。读者将会学到如何从设备的媒体库里选择图片、如何拍摄照片以及如何记录并编辑视频。笔者还要讲解怎样令用户在程序中编写电子邮件和文本消息，以及怎样使用户能够在 Twitter 及 Facebook 等社交网站上发表文章。每个控制器都提供了一种方式，开发者可通过这种方式来使用 iOS 系统内置的功能。

### 8.1 图像选取器控制器

`UIImagePickerControllerController` 类使得用户能够从设备的媒体库里选择图像，并通过设备的摄像头来拍摄图片。它在某种程度上是个“活化石”，早在 iPhone OS 时代，系统就开始提供这个界面了。在苹果公司推出新设备的过程中，这个类也跟着不断进化，iOS 3.1 为其添加了视频录制功能，iOS 4 为其添加了前后摄像头功能。用户可以用它来编辑照片及视频，也可以自定义显示在拍照界面上的图案等。

#### 8.1.1 图像来源

图像选取器可以使用下面三种图像来源：

- `UIImagePickerControllerSourceTypePhotoLibrary`——这种类型的图像来源包含了所有同步到 iOS 上面的图像。其内容包括用户（通过 Camera Roll 程序）拍摄的图像、通过 Photo Stream（照片串流）获取到的图像、从电脑同步过来的相册以及经由 Camera Connection Kit 拷贝的图像等。
- `UIImagePickerControllerSourceTypeSavedPhotosAlbum`——这种类型的

图像来源仅局限于 Camera Roll，对于带有摄像头的设备来说，它包括用户所拍摄的照片及视频；而对于没有摄像头的设备来说，它表示 Saved Photos 相册中的照片。其他设备上的照片如果经由 Photo Stream 串流过来，那么也会同步到 Camera Roll 里。

- **UIImagePickerControllerSourceTypeCamera**——如果开发者将图像来源设为 UIImagePickerControllerSourceTypeCamera，那么用户就可以通过 iPhone 内置的摄像头来拍摄图片了。用户可以在前后两个摄像头之间切换，也可以选择是拍摄静态照片还是录制动态影像。

系统目前提供了上述三种数据来源，使得开发者既可以访问整个媒体库，也可以将范围限定在 Camera Roll 或 Camera 中。你也许还想更为精细地控制应用程序对 iCloud 的访问以及对共享 Photo Stream 及单个 Photo Stream 的访问。这些改进建议可以提交到 <http://bugreport.apple.com/>。

### 8.1.2 在 iPhone 和 iPad 中显示选取器

图 8-1 演示了由 iPhone 和 iPad 所展示的图像选取器界面，其图片来源是媒体库中的所有图片。在 iPhone 系列的设备上，我们把 UIImagePickerController 类以模态界面的形式显示出来（如左侧截图所示），而在平板电脑上，则将其显示成 popover 形式（如右侧截图所示）。

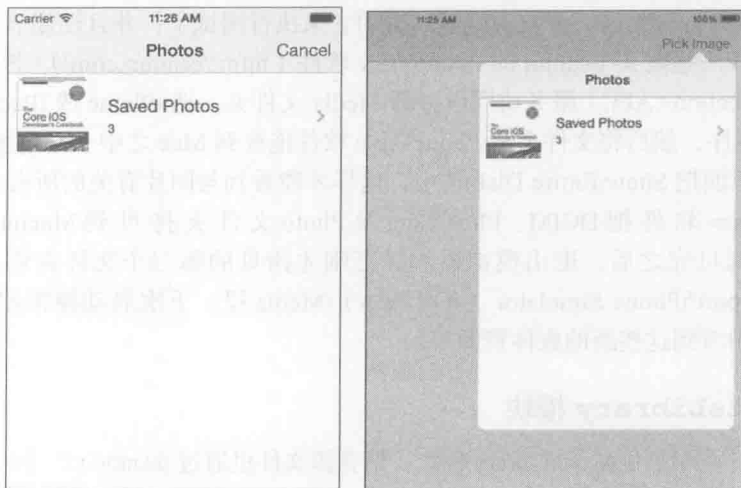


图 8-1 用户可以通过系统自带的图像选取器从媒体库里存储的图片中选择图像

在 iPhone 系列的设备上，应该以模态界面的形式来展示选取器。而在 iPad 上面，则应把它嵌套到 popover 里。开发者不应该把图像选取器推入现有的导航栈。如果在老版本的 iOS 系统上这样做，那么会使现有的主导航栏下方出现另一个导航栏。若是在新版 iOS 系统上面这样做，则会令程序抛出异常：“Pushing a navigation controller is not supported by the image picker”。

## 8.2 解决方案：选取图像

图像选取器最简单的用法是令用户浏览媒体库，并从中选取一张保存过的图片。解决方案 8-1 演示了怎样创建并展示选取器以及怎样获取用户所选的图像。在学习通常的使用方法之前，我们先来讲两个关键问题。

### 8.2.1 向模拟器中添加图片

在 Mac 上面运行本条解决方案之前，我们可能先得往模拟器的媒体库里添加一些图片才行。有两种办法可以添加图片。第一种办法是把图像从 Finder 拖放到模拟器里。模拟器会用 Mobile Safari 浏览器将图片打开，此时开发者可以按住图片不放，然后选择 Save Image，这样就能把图片拷贝到照片库了。

设置好测试用的图片之后，在 Mac 里打开 home 目录 Library 子目录下的 Application Support 文件夹。然后打开其下的 iPhone Simulator 文件夹，再打开与当前 iOS 版本相对应的文件夹（例如 7.0）。在这个文件中，就可以看到 Media 文件夹了。Media 文件夹的路径是这个样子的：`/Users/（你的用户名）/Library/Application Support/iPhone Simulator/（iOS 版本）/Media`。

我们可以把刚生成的 Media 文件夹备份到一个方便取用的位置。创建好备份之后，将来就不用再重新添加每张照片了。每次重置模拟器的内容与配置时，系统都会把这些图片删掉。而有了这样的备份之后，开发者就能随时用它来执行测试了，并且还能节省很多时间。

另外一种办法是购买 Ecamm 的 PhoneView 软件（<http://ecamm.com/>），该软件可以通过 Apple File Connection (AFC) 服务访问设备的 Media 文件夹。把 iPhone 或 iPad 设备与电脑相连，启动应用程序，然后将文件夹从 PhoneView 软件拖放到 Mac 之中。一定要在 PhoneView 的 preferences 里面把 Show Entire Disk 选中，这样才能看到与图片有关的所有文件夹。

用 PhoneView 软件把 DCIM、PhotoData 及 Photo 文件夹拷贝到 Macintosh 上面的某个文件夹中。拷贝完之后，退出模拟器，并把刚才拷贝的那三个文件夹添加到 `~/Library/Application Support/iPhone Simulator/（iOS 版本）/Media` 里。下次启动模拟器的时候，就能在 Photos 程序中看到这些新的媒体资源了。

### 8.2.2 AssetsLibrary 模块

这条解决方案要使用 AssetsLibrary 模块。请在源文件里通过 `@import AssetsLibrary` 指令将其引入。

采用 AssetsLibrary 模块这一做法听上去似乎有些复杂，但为了能够更好地使用图像选取器，我们确实有必要这么做。因为图像选取器返回的也许不是一张直接能够使用的图像，而是一个资源 URL（asset URL）。解决方案 8-1 考虑到了这种情况，所以，它提供了名为 `loadImageFromAssetURL:into:` 的方法，用以从资源库（assets library）中载入图像。资源 URL 通常是这个样子的：

```
assets-library://asset/asset.JPG?id=553F6592-43C9-45A0-B851-28A726727436&ext=JPG
```

上述 URL 可以直接访问对应的媒体资源。

原来访问资源库时有个非常恼人的问题，所幸苹果公司现在已经将它解决了。以前，iOS 会询问用户是否允许程序访问媒体资源中的位置（location）信息，而用户一般会拒绝。由于应用程序无法迫使系统再次询问用户，所以它会卡在这里。从 iOS 6 开始，这条询问消息不再说程序会访问位置了，而是改说应用程序想要访问用户的照片，这样一来，用户很可能允许该请求。我们可以通过查询 `ALAssetsLibrary` 的 `authorizationStatus` 方法来获知授权情况。用户若想重置已经授予的权限，可以打开 `Settings > Privacy`，然后在每个应用程序中调整那些基于服务的许可（例如对位置与照片的访问权限）。

不过，苹果公司在初版的 iOS 7 系统中，又引入了一个与 iPad 上面的资源库授权有关的 bug。如果在 popover 中显示选取器，那么当程序从征求用户许可的界面返回时，就会崩溃。虽说重启应用程序即可解决此问题，但在初次启动程序时，毕竟会造成糟糕的用户体验。有个解决办法是在显示 popover 之前，先请求用户允许应用程序访问资源库：

```
// Force authorization for asset library
[assetsLibrary enumerateGroupsWithTypes:ALAssetsGroupAll
 usingBlock:^(ALAssetsGroup *group, BOOL *stop) {
     // If authorized, catch the final iteration and display popover
     if (group == nil)
     {
         dispatch_async(dispatch_get_main_queue(), ^{
             popover = [[UIPopoverController alloc]
                 initWithContentViewController:
                     viewControllerToPresent];
             popover.delegate = self;
             [popover presentPopoverFromBarButtonItem:
                 self.navigationItem.rightBarButtonItem
                 permittedArrowDirections:
                     UIPopoverArrowDirectionAny
                 animated:YES];
         });
     }
     *stop = YES;
 } failureBlock:nil];
```

讲完相关的细节问题之后，下一节我们开始介绍图像选取器本身的用法。

### 8.2.3 展示选取器

首先创建并初始化图像选取器，然后把它的来源类型设置成 `UIImagePickerControllerSourceTypePhotoLibrary`（表示媒体库中的所有图片）或 `UIImagePickerControllerSourceTypeSavedPhotosAlbum`（表示用户拍摄的图片）。解决方案 8-1 把 `sourceType` 设置成 `UIImagePickerControllerSourceTypePhotoLibrary`，使得用户可以浏览媒体库中的所有图片。

```
UIImagePickerController *picker = [[UIImagePickerController alloc] init];
picker.sourceType = UIImagePickerControllerSourceTypePhotoLibrary;
```

还有个可选的属性叫作 `allowsEditing`，如果将其打开，那么用户在操作图像选取器的时候，就可以多执行一步。开启该属性后，用户可以在完成选取操作之前，先对所选图像执行缩放及调整。如果禁用该属性，那么只要用户选择了图像，系统就会立刻把控制权交给选取器程序，并使其进入生命期的下一个阶段。

一定要设置选取器的 `delegate` 属性。这个 `delegate` 要遵循 `UINavigationControllerDelegate` 及 `UIImagePickerControllerDelegate` 协议。当用户选择了某个图像或是取消了这次选择操作之后，`delegate` 会收到相应的回调消息。如果把 `UIImagePickerController` 和 `popover` 结合起来使用，那么还应该令 `delegate` 遵循 `UIPopoverControllerDelegate` 协议。

在 iPhone 系列的设备上面，总是应该以模态方式来展示选取器，我们需要在程序运行的时候判断设备类型。当程序运行在 iPhone 上时，下列测试语句会返回 `true`，如果运行在 iPad 上，则返回 `false`（适用于 iOS 3.2 及后续系统）：

```
#define IS_IPHONE (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPhone)
```

下面这段代码以常用的展示方式来显示图像选取器：

```
if (IS_IPHONE)
{
    [self presentViewController:picker animated:YES completion:nil];
}
else
{
    if (popover) [popover dismissPopoverAnimated:NO];
    popover = [[UIPopoverController alloc]
        initWithContentViewController:picker];
    popover.delegate = self;
    [popover presentPopoverFromBarButtonItem:
        self.navigationItem.rightBarButtonItem
        permittedArrowDirections:UIPopoverArrowDirectionAny
        animated:YES];
}
```

## 8.2.4 处理 `delegate` 的回调

解决方案 8-1 考虑了下面三种在使用图像选取器时可能发生的回调：

- 用户成功地选定了一张图像。
- 用户点击了 `Cancel`（只会发生在模态界面中）。
- 用户在 `popover` 的范围之外点击屏幕，从而隐藏了嵌有选取器的 `popover`。

后面两种情况比较简单。对于模态界面来说，把控制器隐藏起来就好。对于 `popover` 来说，不要令局部引用再持有 `popover` 实例即可。而要处理用户选好的图像，则需要多编写一些代码才行。

选取器会向其 `delegate` 返回一份含有自定义信息的字典（`dictionary`），然后它的生命期就结束了。这个字典里面包含与用户所选内容相关的键值对。字典里可能只有几个键

(key), 也可能会有很多键, 具体数量取决于开发者配置图像选取器的方式以及用户所选媒体资源的类型。

### 解决方案 8-1 选取图像

```
#define IS_IPHONE (UI_USER_INTERFACE_IDIOM() == \
    UIUserInterfaceIdiomPhone)

// Dismiss the picker
- (void)performDismiss
{
    if (IS_IPHONE)
        [self dismissViewControllerAnimated:YES completion:nil];
    else
    {
        [popover dismissPopoverAnimated:YES];
        popover = nil;
    }
}

// Present the picker
- (void)presentViewController:
    (UIViewController *)viewControllerToPresent
{
    if (IS_IPHONE)
    {
        [self presentViewController:viewControllerToPresent
            animated:YES completion:nil];
    }
    else
    {
        // Workaround to an Apple crasher when asking for asset
        // library authorization with a popover displayed
        ALAssetsLibrary * assetsLibrary =
            [[ALAssetsLibrary alloc] init];
        ALAuthorizationStatus authStatus;

        if (NSFoundationVersionNumber >
            NSFoundationVersionNumber_iOS_6_0)
            authStatus = [ALAssetsLibrary authorizationStatus];
        else
            authStatus = ALAuthorizationStatusAuthorized;

        if (authStatus == ALAuthorizationStatusAuthorized)
        {
            popover = [[UIPopoverController alloc]
                initWithContentViewController:viewControllerToPresent];
            popover.delegate = self;
            [popover presentPopoverFromBarButtonItem:
                self.navigationItem.rightBarButtonItem
                permittedArrowDirections:UIPopoverArrowDirectionAny
```

```

        animated:YES];
    }
    else if (authStatus == ALAuthorizationStatusNotDetermined)
    {
        // Force authorization
        [assetsLibrary enumerateGroupsWithType:ALAssetsGroupAll
         usingBlock:^(ALAssetsGroup *group, BOOL *stop){
            // If authorized, catch the final iteration
            // and display popover
            if (group == nil)
            {
                dispatch_async(dispatch_get_main_queue(), ^{
                    popover = [[UIPopoverController alloc]
                               initWithContentViewController:
                               viewControllerToPresent];
                    popover.delegate = self;
                    [popover presentPopoverFromBarButtonItem:
                     self.navigationItem.rightBarButtonItem
                     permittedArrowDirections:
                     UIPopoverArrowDirectionAny
                     animated:YES];
                });
            }
            *stop = YES;
        } failureBlock:nil];
    }
}

// Popover was dismissed
- (void)popoverControllerDidDismissPopover:
    (UIPopoverController *)aPopoverController
{
    popover = nil;
}

// Retrieve an image from an asset URL
- (void)loadImageFromAssetURL: (NSURL *)assetURL
    into: (UIImage **)image
{
    ALAssetsLibrary *library = [[ALAssetsLibrary alloc] init];
    ALAssetsLibraryAssetForURLResultBlock resultsBlock =
        ^(ALAsset *asset)
    {
        ALAssetRepresentation *assetRepresentation =
            [asset defaultRepresentation];
        CGImageRef cgImage =
            [assetRepresentation CGImageWithOptions:nil];
        CFRetain(cgImage); // Thanks, Oliver Drobnik
        if (image) *image = [UIImage imageWithCGImage:cgImage];
        CFRelease(cgImage);
    }
}

```



```

    };
    ALAssetsLibraryAccessFailureBlock failureBlock =
        ^(NSError *__strong error)
    {
        NSLog(@"Error retrieving asset from url: %@",
            error.localizedFailureReason);
    };

    [library assetForURL:assetURL
        resultBlock:resultsBlock failureBlock:failureBlock];
}

// Update image and for iPhone, dismiss the controller
- (void)imagePickerController:(UIImagePickerController *)picker
    didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    // Use the edited image if available
    UIImage __autoreleasing *image =
        info[UIImagePickerControllerEditedImage];

    // If not, grab the original image
    if (!image)
        image = info[UIImagePickerControllerOriginalImage];

    // If still no luck, check for an asset URL
    NSURL *assetURL = info[UIImagePickerControllerReferenceURL];
    if (!image && !assetURL)
    {
        NSLog(@"Cannot retrieve an image from the selected item. Giving up.");
    }
    else if (!image)
    {
        // Retrieve the image from the asset library
        [self loadImageFromAssetURL:assetURL into:&image];
    }

    // Display the image
    if (image)
        imageView.image = image;

    if (IS_IPHONE)
        [self performDismiss];
}

// iPhone-like devices only: dismiss the picker with cancel button
- (void)imagePickerControllerDidCancel:
    (UIImagePickerController *)picker
{
    [self performDismiss];
}

```

```

- (void)pickImage
{
    if (popover) return;

    // Create and initialize the picker
    UIImagePickerController *picker =
        [[UIImagePickerController alloc] init];
    picker.sourceType =
        UIImagePickerControllerSourceTypePhotoLibrary;
    picker.allowsEditing = editSwitch.isOn;
    picker.delegate = self;

    [self presentViewController:picker];
}

```

### 获取解决方案代码

访问 <https://github.com/erica/iOS-7-Cookbook> 网页，并打开“C08 Common Controllers”文件夹，即可找到与本章中的解决方案相对应的完整范例项目。

比方说，如果我们通过 Safari 把图像保存到模拟器，然后又选中了这些图像，那么字典里就只有“媒体类型”和“引用 URL”这两个键。如果图片是用户用设备拍摄的，而且还经过了编辑，那么图像选取器所返回的字典里就会包含下面六个键：

- UIImagePickerControllerMediaType——它表示用户所选的媒体资源是何类型。一般来说，public.image 表示图片，public.movie 表示影像。媒体类型定义于 Mobile Core Services 框架之中。在使用图像选取器的时候，媒体类型主要用来把用户所选内容添加到系统的剪贴板里。
- UIImagePickerControllerCropRect——它是 NSValue，其中存放 CGRect，用来描述用户所选的部分图像。
- UIImagePickerControllerOriginalImage——它对应于一个 UIImage 实例，该实例表示原始的（也就是没有经过编辑的）图像内容。
- UIImagePickerControllerEditedImage——它对应于编辑之后的图像，其中包含用户所选定的那一部分。这个 UIImage 比较小，它是按照设备屏幕尺寸调整过的。
- UIImagePickerControllerReferenceURL——它表示文件系统中的 URL，该 URL 指向用户所选的资源。无论用户是否裁切过图像或剪辑过影片，这个 URL 总是指向资源的原始版本。
- UIImagePickerControllerMediaMetadata——它表示用户在图像选取器内所拍照片的 metadata（元数据）。

为了根据字典中的内容获取到相关图像，解决方案 8-1 采取了多个步骤。首先，它检测字典中是否包含编辑过的版本。若是找不到，那么就访问原来的图像。如果这一操作也失败了，那么就从 reference URL 中获取图像，并试着用 AssetsLibrary 来加载它。一般来说，执

行完这些步骤之后，应用程序都会获取到可供操作的有效 UIImage 实例。若是没有这样的实例，那么就记录错误信息并返回。

最后，delegate 的回调方法在完成其任务之前，不要忘了把模态形式的控制器关掉。



**提示** 如果把各种用户操作都比作动物的话，那么 UIImagePickerControllerController 就是一头牛。它行动比较缓慢，非常容易消耗应用程序的内存，而且还要花些时间来“反刍”（chew its cud）。所以，在设计程序的时候要注意这些限制，不要因为使用图像选取器而影响整个应用程序的效果。

## 8.3 解决方案：拍摄照片

除了可以选取图片，用户还能通过图像选取器用设备内置的摄像头拍照。由于并非每种 iOS 设备都有摄像头（尤其是早期的 iPod touch 和 iPad 设备），所以在拍照之前，应该先检查运行应用程序的这台设备是否支持摄像头：

```
if ([UIImagePickerController isSourceTypeAvailable:
    UIImagePickerControllerSourceTypeCamera]) ...
```

有一条经验就是：决不要在没有摄像头的设备上面提供基于摄像头的功能。虽说 iOS 7 只会部署在带有摄像头的设备上，但除了苹果公司之外，谁也不知道将来还会不会发行没有摄像头的设备。虽然听起来不太可能，但苹果公司说不定还会发行没有摄像头的新设备。如果苹果公司没有否认这一点，那么即便在新版的 iOS 中，我们也依然得考虑到不带摄像头的设备才行。而且，我们还得假定：如果将来系统里引入了一些设定，能够在带有摄像头的设备上面禁用某些数据源，那么上述方法依然能够返回正确的结果。

### 8.3.1 配置选取器

拍照所用的图像选取器的实例化方式与选取普通图像时所用的选取器相似。只需把数据源的类型从 UIImagePickerControllerSourceTypePhotoLibrary 改成 UIImagePickerControllerSourceTypeCamera 即可：

```
picker.sourceType = UIImagePickerControllerSourceTypeCamera;
```

与其他模式下的图像选取器类似，我们也可以通过设置 allowsEditing 属性来允许或禁止用户在拍照过程中对图像进行编辑。

虽说用 UIImagePickerControllerSourceTypeCamera 来配置图像选取器与用 UIImagePickerControllerSourceTypePhotoLibrary 来配置是相同的，但这二者所产生的用户体验却略有差异（参见图 8-2）。用户点击照相机图标并拍好照片之后，图像选取器会展示出预览（preview）界面。在该界面中，用户可以重新拍照，也可以采用现在这张照片。如果点击 Use Photo，那么系统就会把控制权交给下一阶段的程序。假如开启了图像编辑功能，那么用户还可以编辑图像。如果没开启这个功能，那么控制权就会转移给 delegate 中那个

标准的 `UIImagePickerController:didFinishPickingMediaWithInfo:` 方法。

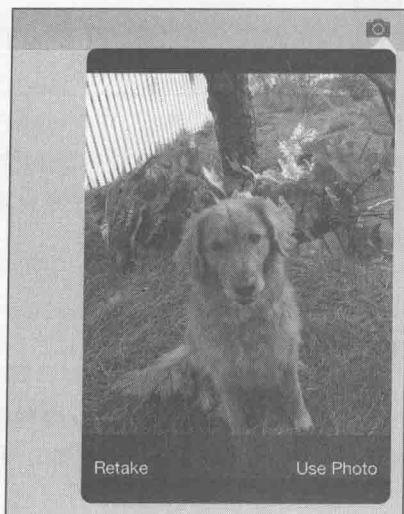


图 8-2 用于拍照的 `UIImagePickerController`，其用户体验与用于选取图像的 `UIImagePickerController` 有所不同，这使得用户可以通过它来拍摄照片

大部分新型设备都提供了不只一个摄像头。所有能运行 iOS 7 的设备都有前后两个摄像头。在支持 iOS 6 的设备之中，只有 iPhone 3GS 是单摄像头。开发者可通过 `cameraDevice` 属性来设定想要使用的摄像头。系统总是默认使用后置摄像头。

名为 `isCameraDeviceAvailable:` 的类方法可以查出设备的某个摄像头是否可用。下面这段代码能够检测前置摄像头是否可用，如果可以使用，那么就用它来拍照：

```
if ([UIImagePickerController isCameraDeviceAvailable:
    UIImagePickerControllerCameraDeviceFront])
    picker.cameraDevice = UIImagePickerControllerCameraDeviceFront;
```

通过 `UIImagePickerController` 类来使用摄像头时，还应注意下面几个问题：

- 开发者可通过名为 `isFlashAvailableForCameraDevice:` 的类方法查询设备是否能使用闪光灯。调用该方法时，应该传入与设备前置摄像头或后置摄像头相对应的常量。如果可以用闪光灯，该方法就返回 YES，否则返回 NO。
- 如果设备支持闪光灯，那么可以把 `cameraFlashMode` 属性设置成 `UIImagePickerControllerCameraFlashModeAuto`（自动，该值是默认值）、`UIImagePickerControllerCameraFlashModeOn`（总是打开）或 `UIImagePickerControllerCameraFlashModeOff`（总是关闭）。禁用闪光灯之后，无论拍照环境是暗还是亮，闪光灯都不会打开。
- 开发者通过 `cameraCaptureMode` 属性指定程序是用摄像头来拍照还是用它来录像。图像选取器默认的模式是拍照。`availableCaptureModesForCameraDevice:` 方法可以判断出设备所支持的模式。该方法返回的数组中包含 `NSNumber` 型的对象，

每个对象的值都表示设备所支持的一种捕获模式：UIImagePickerControllerCameraCaptureModePhoto 表示拍照，UIImagePickerControllerCameraCaptureModeVideo 表示录像。

### 8.3.2 显示图像

处理照片的时候，要考虑到其图像尺寸。用户所拍的照片（尤其是那种用高分辨率摄像头所拍的照片）会非常大，即便在 Retina 显示屏上也是如此。而前置摄像头所拍照片的分辨率则比较低，尺寸也小得多。

我们可以在程序里通过 contentMode 来解决显示大型图片这一问题。显示图片的视图能够将其中的图片缩放到与屏幕大小相符的尺寸。我们可以考虑使用下列模式：

- UIViewContentModeScaleAspectFit 模式会在保持宽高比不变的情况下，把整幅图像显示出来。为了保持宽高比，图像左右两侧或上下两端可能会出现空矩形。
- UIViewContentModeScaleAspectFill 模式尽可能用图像填满整个视图。为了填满视图，有些内容也许会裁切掉。

### 8.3.3 把图像保存到相册

调用 UIImageWriteToSavedPhotosAlbum() 函数，即可将拍好的图像（实际上可以是任意的 UIImage 实例）保存到相册。该函数接受四个参数。第一个参数表示待存的图像。第二与第三个参数分别是回调的目标以及回调的选择子，这两者通常是主视图控制器以及 image:didFinishSavingWithError:contextInfo:。第四个参数是可选的，它是个指向上下文的指针。无论采用哪种选择子，这个选择子都必须接受三个参数，它们分别表示图像、错误以及指向传入的上下文信息的指针。

解决方案 8-2 采用 UIImageWriteToSavedPhotosAlbum() 函数来演示如何拍摄新图像、如何允许用户编辑图像以及怎样把图像保存到相册。

解决方案 8-2 拍摄照片

```
// "Finished saving" callback method
- (void)image:(UIImage *)image
  didFinishSavingWithError:(NSError *)error
  contextInfo:(void *)contextInfo;
{
    // Handle the end of the image write process
    if (!error)
        NSLog(@"Image written to photo album");
    else
        NSLog(@"Error writing to photo album: %@",
              error.localizedDescription);
}

// Save the returned image
- (void)imagePickerController:(UIImagePickerController *)picker
  didFinishPickingMediaWithInfo:(NSDictionary *)info
```

```

{
    // Use the edited image if available
    UIImage __autoreleasing *image =
        info[UIImagePickerControllerEditedImage];

    // If not, grab the original image
    if (!image)
        image = info[UIImagePickerControllerOriginalImage];

    // If still no luck, check for an asset URL
    NSURL *assetURL = info[UIImagePickerControllerReferenceURL];
    if (!image && !assetURL)
    {
        NSLog(@"Cannot retrieve an image from selected item. Giving up.");
    }
    else if (!image)
    {
        NSLog(@"Retrieving from Assets Library");
        [self loadImageFromAssetURL:assetURL into:&image];
    }

    if (image)
    {
        // Save the image
        UIImageWriteToSavedPhotosAlbum(image, self,
            @selector(image:didFinishSavingWithError:contextInfo:),
            NULL);
        imageView.image = image;
    }

    [self performDismiss];
}

- (void)loadView
{
    self.view = [[UIView alloc] init];
    self.view.backgroundColor = [UIColor whiteColor];

    imageView = [[UIImageView alloc] init];
    imageView.contentMode = UIViewContentModeScaleAspectFit;
    [self.view addSubview:imageView];
    PREPCONSTRAINTS(imageView);
    STRETCH_VIEW(self.view, imageView);

    // Only present the "Snap" option for camera-ready devices
    if ([UIImagePickerController isSourceTypeAvailable:
        UIImagePickerControllerSourceTypeCamera])
        self.navigationItem.rightBarButtonItem =
            SYSBARBUTTON(UIBarButtonSystemItemCamera,
                @selector(snapImage));
}

```

```

// Set up title view with Edits: ON/OFF
editSwitch = [[UISwitch alloc] init];
UILabel * editLabel =
    [[UILabel alloc] initWithFrame:CGRectMake(0, 0, 40, 13)];
editLabel.text = @"Edits";
self.navigationItem.leftBarButtonItems =
    @[[[UIBarButtonItem alloc] initWithCustomView:editLabel],
      [[UIBarButtonItem alloc] initWithCustomView:editSwitch]];
}

```

## 8.4 解决方案：录制视频

虽说在 iOS 7 时代很多设备都带了摄像头，但我们还是得判断运行程序的这台设备到底有没有摄像头，此外，还要判断摄像头的类型。录制视频之前，应用程序应该先检查设备是否支持基于摄像头的视频录制。

这个过程要分两步执行。只检测设备有没有摄像头是不够的，因为第一代 iPhone 及 iPhone 3G 虽然有摄像头，但却没有录制视频的功能（早期 iPad 与 iPod touch 设备根本就没有摄像头），只有 3GS 及后续机型才可以录制视频。虽说不太可能，但苹果公司将来没准还会推出带有摄像头却只能拍摄静态照片的机型。

于是，我们必须执行两次检测：首先判断设备有没有摄像头，然后判断可以捕获的媒体类型里面有没有“视频”这一项。下列方法返回的布尔值用来表示运行程序的设备能不能拍摄视频：

```

- (BOOL)videoRecordingAvailable
{
    // The source type must be available
    if (![UIImagePickerController isSourceTypeAvailable:
        UIImagePickerControllerSourceTypeCamera])
        return NO;

    // And the media type must include the movie type
    NSArray *mediaTypes = [UIImagePickerController
        availableMediaTypesForSourceType:
        UIImagePickerControllerSourceTypeCamera]
    return [mediaTypes containsObject:(NSString *)kUTTypeMovie];
}

```

上述方法会查询设备所支持的媒体类型，然后在查询结果里搜寻表示视频的那种类型（kUTTypeMovie，也就是 public.movie）。统一类型标识符（Uniform Type Identifier，简称 UTI）是一种描述抽象类型的字符串，用来指代图像、影片及数据等常用的文件格式。第 11 章将会详述 UTI。由于这些类型定义于 Mobile Core Services 模块之中，所以在源文件里需要引入该模块：

```
@import MobileCoreServices;
```

### 8.4.1 创建录制视频用的选取器

用摄像头来录制视频与用它来拍摄静态照片是一样的。解决方案 8-3 新建并初始化了一个 UIImagePickerControllerController，然后设置它的 delegate，并将其展示出来：

```
UIImagePickerController *picker =
    [[UIImagePickerController alloc] init];
picker.sourceType = UIImagePickerControllerSourceTypeCamera;
picker.videoQuality = UIImagePickerControllerQualityTypeMedium;
picker.mediaTypes = @[NSString *kUTTypeMovie]; // public.movie
picker.delegate = self;
```

开发者可以指定视频的录制质量。视频质量越高，每秒钟所产生的数据量就越大。我们可以选择 UIImagePickerControllerQualityTypeHigh (高质量)、UIImagePickerControllerQualityTypeMedium (中等质量)、UIImagePickerControllerQualityTypeLow (低质量) 或 UIImagePickerControllerQualityType640x480 (VGA)。

与选取图片时一样，我们也可以在拍摄视频所用的选取器上面设置 allowsEditing 属性，解决方案 8-5 将会讲解该属性。

解决方案 8-3 录制视频

```
- (void)video:(NSString *)videoPath
    didFinishSavingWithError:(NSError *)error
    contextInfo:(void *)contextInfo
{
    if (!error)
        self.title = @"Saved!";
    else
        NSLog(@"Error saving video: %@",
            error.localizedDescription);
}

- (void)saveVideo:(NSURL *)mediaURL
{
    // check if video is compatible with album
    BOOL compatible =
        UIVideoAtPathIsCompatibleWithSavedPhotosAlbum(
            mediaURL.path);

    // save
    if (compatible)
        UISaveVideoAtPathToSavedPhotosAlbum(
            mediaURL.path, self,
            @selector(video:didFinishSavingWithError:contextInfo:),
            NULL);
}

- (void)imagePickerController:(UIImagePickerController *)picker
    didFinishPickingMediaWithInfo:(NSDictionary *)info
{

```



```

[self performDismiss];

// Save the video
NSURL *mediaURL =
    info[UIImagePickerControllerMediaURL];
[self saveVideo: mediaURL];
}

- (void)recordVideo
{
    if (popover) return;
    self.title = nil;

    // Create and initialize the picker
    UIImagePickerController *picker =
        [[UIImagePickerController alloc] init];
    picker.sourceType = UIImagePickerControllerSourceTypeCamera;
    picker.videoQuality = UIImagePickerControllerQualityTypeMedium;
    picker.mediaTypes = @[NSString *kUTTypeMovie];
    picker.delegate = self;

    [self presentViewController:picker];
}

```

### 8.4.2 保存视频

视频选取器所返回的信息字典 (info dictionary) 里面含有名为 UIImagePickerControllerMediaURL 的键, 该键所对应的媒体 URL 指向录制好的视频, 此视频存放于应用程序沙盒 (app sandbox) 里的临时文件夹中。我们可以用 `UISaveVideoAtPathToSavedPhotosAlbum()` 函数将视频保存到媒体库。

保存视频用的函数接受四个参数: 视频在媒体库中的保存路径、回调的目标、回调的选择子 (这个选择子接受三个参数, 与保存图片时所指定的那个回调选择子基本相同) 以及可选的上下文。此函数执行完任务之后, 会在目标上面调用选择子, 使得开发者可以检查任务有没有执行成功。

## 8.5 解决方案: 用媒体播放器播放视频

MPMoviePlayerViewController 类及 MPMoviePlayerController 类简化了应用程序中的视频播放。这些类是 Media Player 框架的一部分, 它们使得开发者可以把视频嵌入到视图之中, 或是在全屏模式下播放视频。图 8-3 演示了系统预置的视频播放器, 这个播放器的功能比较完备, 开发者只需提供指向视频内容的 URL 即可。播放器提供有 Done 按钮、时间滚动条、宽高比控制按钮、回放控件, 并且会把视频内容展示在这些控件后方。



图 8-3 MediaPlayer 框架简化了应用程序的视频回放。这个类既可以播放设备之外的串流视频，也可以播放存储在本机的固定视频资源。支持的视频格式包括 H.264 Baseline Profile Level 3.0 视频（每秒最多 30 帧，每帧最大  $640 \times 480$ ）以及 MPEG-4 Part 2 视频（Simple Profile）。以 .mov、.mp4、.mpv 及 .3gp 为扩展名的大部分文件都可以播放。支持的音频格式包括 AAC-LC 音频（采样率最大 48KHz）以及 MP3（MPEG-1 Audio Layer 3，采样率最大 48KHz）立体声

我们在解决方案 8-3 里构建了视频录制功能，解决方案 8-4 是基于解决方案 8-3 而编写的。每次录完视频之后，它会把导航栏上的 Camera 按钮切换成 Play 按钮，以便回放视频。这条解决方案没有把视频保存到媒体库中，用户可以反复地录制并播放视频。

#### 解决方案 8-4 视频回放

```
#define SYSBARBUTTON(ITEM, SELECTOR) [[UIBarButtonItem alloc] \
initWithBarButtonSystemItem:ITEM target:self action:SELECTOR]

- (void)playMovie
{
    // Prepare movie player and play
    MPMoviePlayerViewController *player =
        [[MPMoviePlayerViewController alloc]
        initWithContentURL:mediaURL];
    player.moviePlayer.allowsAirPlay = YES;
    player.moviePlayer.controlStyle = MPMovieControlStyleFullscreen;

    [self.navigationController
     presentMoviePlayerViewControllerAnimated:player];

    // Handle the end of movie playback
    [[NSNotificationCenter defaultCenter]
     addObserverForName:MPMoviePlayerPlaybackDidFinishNotification
     object:player.moviePlayer queue:[NSOperationQueue mainQueue]
     usingBlock:^(NSNotification *notification){
        // Return to recording mode
        self.navigationItem.rightBarButtonItem =
```

```

        SYSBARBUTTON(UIBarButtonSystemItemCamera,
        @selector(recordVideo));
        // Stop listening to movie notifications
        [[NSNotificationCenter defaultCenter]
        removeObserver:self];
    }];

    // Wait for the movie to load and become playable
    [[NSNotificationCenter defaultCenter]
    addObserverForName:MPMoviePlayerLoadStateDidChangeNotification
    object:player.moviePlayer queue:[NSOperationQueue mainQueue]
    usingBlock:^(NSNotification *notification) {

        // When the movie sets the playable flag, start playback
        if ((player.moviePlayer.loadState &
        MPMovieLoadStatePlayable) != 0)
            [player.moviePlayer performSelector:@selector(play)
            withObject:nil afterDelay:1.0f];
    }];
}

// After recording any content, allow the user to play it
- (void)imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    [self performDismiss];

    // recover video URL
    mediaURL = info[UIImagePickerControllerMediaURL];
    self.navigationItem.rightBarButtonItem =
        SYSBARBUTTON(UIBarButtonSystemItemPlay,
        @selector(playMovie));
}

```

图像选取器提供了视频媒体的 URL，我们只需通过这个 URL，就能建立播放器。解决方案 8-4 实例化了一个新的播放器，并设置了它的两个属性。首先激活 AirPlay，使手机可以把录好的视频串流到带 AirPlay 功能的接收方那里，比如苹果公司的 TV 或 Reflector (<http://reflectorapp.com/>) 等商业软件。然后设置回放风格，令视频以全屏状态播放。接下来把视频显示出来。

组成视频播放器的两个类中，有一个是可以展示出来的视图控制器 (MPMoviePlayerViewController)，另一个则是实际的播放控制器 (MPMoviePlayerController)，前者会把后者放在自己的一个属性里面。这就是解决方案 8-4 中多次提到 player.moviePlayer 的原因。视图控制器类非常小，而且易于启动。真正的工作是在播放控制器里完成的。

视频播放器采用通知机制与应用程序相通信，而不采用委托。开发者可以通过订阅通知来获知视频何时开始播放、何时播放完毕以及何时改变状态（比方说从暂停状态变为播放状态）。解决方案 8-4 订阅了两种通知，用来监测什么时候可以播放视频以及视频什么时候播放完毕。

等视频加载进来并且状态变为可以播放之后，解决方案 8-4 就开始回放该视频。程序会进入全屏状态，并持续播放视频，直到用户点击 Done 按钮或视频播放完毕为止。在这两种情况下，播放器都会产生播放完毕的通知。此时，应用程序会返回录制模式，并显示出 Camera 按钮，使得用户可以录制下一段视频。

这条解决方案演示了在 iOS 系统里播放视频所需的基本代码。程序不一定非要播放用户自己录制的视频，MPMoviePlayerController 没有限制视频的来源。你可以把 contentURL 设为沙箱中的某个文件，也可以令其指向互联网上某个能够播放出来的视频资源。



**提示** 如果视频播放器刚一打开就立刻关闭了，那么请检查 URL 是否正确。别忘了，指向本地文件的 URL 应该用 fileURLWithPath: 方法来创建，而指向远程资源的 URL 则应该以 URLWithString: 来制作。

## 8.6 解决方案：编辑视频

如果图像选取器的媒体来源是一段视频，那么在启用了 allowsEditing 属性之后，就会看到一条黄颜色的编辑栏，它与内置的 Photos 程序所显示的编辑栏是一样的。（拖动编辑栏左右两端的指示按钮，即可看到这种效果。）在进入编辑视频这一步骤之后，用户可以通过拖曳编辑栏左右两端，来选择想要保留的视频范围。

奇怪的是，UIImagePickerController 并不会直接剪辑当前的视频，而是返回包含下列四个项目的信息字典：

- UIImagePickerControllerMediaURL
- UIImagePickerControllerMediaType
- \_UIImagePickerControllerVideoEditingStart
- \_UIImagePickerControllerVideoEditingEnd

媒体 URL 指向尚未剪辑的视频，该视频保存在沙箱中的临时文件夹里。视频的起点与终点都用 NSNumber 值来表示，它们就是用户刚才在黄颜色的编辑栏中所指定的偏移量。媒体类型是 public.movie。

如果把视频保存到媒体库（像解决方案 8-3 那样），那么保存的将是没有剪辑过的版本，这并不是用户想要的效果。iOS SDK 提供了两种编辑视频的方式。解决方案 8-5 将会演示如何用 AV Foundation 框架来响应视频编辑请求，以便剪辑由 UIImagePickerController 所返回的视频。解决方案 8-6 将要演示怎样从媒体库中选取视频，并使用 UIVideoEditorController 来编辑这些视频。

### 解决方案 8-5 用 AV Foundation 框架来剪辑视频

```
(void)trimVideo:(NSDictionary *)info
{
    // recover video URL
    NSURL *mediaURL =
```

```

        info[UIImagePickerControllerMediaURL];
        AVURLAsset *asset =
            [AVURLAsset URLAssetWithURL:mediaURL options:nil];

        // Create the export range
        CGFloat editingStart =
            [info[@"UIImagePickerControllerVideoEditingStart"]
             floatValue];
        CGFloat editingEnd =
            [info[@"UIImagePickerControllerVideoEditingEnd"]
             floatValue];
        CMTime startTime = CMTimeMakeWithSeconds(editingStart, 1);
        CMTime endTime = CMTimeMakeWithSeconds(editingEnd, 1);
        CMTimeRange exportRange =
            CMTimeRangeFromTimeToTime(startTime, endTime);

        // Create a trimmed version URL: file:originalpath-trimmed.mov
        NSString *urlPath = mediaURL.path;
        NSString *extension = urlPath.pathExtension;
        NSString *base = [urlPath stringByDeletingPathExtension];
        NSString *newPath = [NSString stringWithFormat:
            @"%@-trimmed.%@", base, extension];
        NSURL *fileURL = [NSURL fileURLWithPath:newPath];

        // Establish an export session
        AVAssetExportSession *session = [AVAssetExportSession
            exportSessionWithAsset:asset
            presetName:AVAssetExportPresetMediumQuality];
        session.outputURL = fileURL;
        session.outputFileType = AVFileTypeQuickTimeMovie;
        session.timeRange = exportRange;

        // Perform the export
        [session exportAsynchronouslyWithCompletionHandler:^(){
            if (session.status ==
                AVAssetExportSessionStatusCompleted)
                [self saveVideo:fileURL];
            else if (session.status ==
                AVAssetExportSessionStatusFailed)
                NSLog(@"AV export session failed");
            else
                NSLog(@"Export session status: %d", session.status);
        }];
    }

```

## AV Foundation 及 Core Media 框架

这条解决方案需要访问两个功能非常专业的模块。AV Foundation 模块提供了一套 Objective-C 语言的接口，用来处理媒体资源。而 Core Media 则采用一套底层的 C 语言接口

来描述媒体的属性。把二者结合起来，就能在 iOS 上面实现出与 Mac 的 QuickTime 相仿的媒体播放效果了。我们在本条解决方案的源文件中引入这两个模块。

解决方案 8-5 首先从图像选取器所返回的信息字典里获取媒体的 URL。这个 URL 指向沙盒中的临时文件，该文件是由图像选取器所创建的。这条解决方案会根据媒体的 URL 来新建 AV 资源 URL (AV asset URL)。然后，它要创建导出范围，凡是位于此范围中的视频内容都应该保存到媒体库里。为了创建这个范围，我们需要用信息字典中的起始时间和终止时间来构建 Core Media 框架中的 CMTimeRange 结构体。CMTimeMakeWithSeconds() 函数接受两个参数：时间与缩放倍数。为了确保精确的剪辑时间，本条解决方案将缩放倍数设为 1。

应用程序可以通过 AVAssetExportSession 把数据存回文件系统。这个 session 并不会将视频存入媒体库，而是通过另外一个步骤来完成。这个 session 会把剪辑过的视频以本地文件的形式保存到沙箱中的临时文件夹里，并与用户所拍摄的原始视频文件放在一起。在创建 AVAssetExportSession 的时候，我们要设置资源和导出质。品质。

解决方案 8-5 会把剪辑过的视频保存到新的路径中。这条路径与读入资源时所用的路径相同，只不过文件名后面会多出来“-trimmed”字样。我们把新路径设为 AVAssetExportSession 的 outputURL，并采用 exportRange 里面所指定的起止时间来导出视频，同时，还把导出视频的类型设置成 AVFileTypeQuickTimeMovie。现在可以开始处理视频文件了。AVAssetExportSession 会以异步的方式执行文件导出操作，它所使用的媒体属性及媒体内容都是由我们传给它的资源所决定的。

剪辑完视频后，我们将其存入设备的中央媒体库。解决方案 8-5 把保存视频文件所用的代码放在块里，AVAssetExportSession 会在执行完导出操作之后，运行该块中的代码。

## 8.7 解决方案：选取并编辑视频

你可以像选取图像时那样，用 UIImagePickerController 来选取视频，解决方案 8-6 演示了这一功能。我们只需要稍微修改一下 mediaTypes 属性即可。sourceType 还是像原来那样设置，但这次要限定 mediaTypes 属性。下面这段代码演示了怎样设置 mediaTypes 才能令 UIImagePickerController 只显示视频资源：

```
picker.sourceType = UIImagePickerControllerSourceTypePhotoLibrary;
picker.mediaTypes = @[ (NSString *)kUTTypeMovie];
```

用户选定某个视频之后，解决方案 8-6 便进入编辑模式。要记得检查这个视频资源能不能修改。我们调用 UIVideoEditorController 的类方法 canEditVideoAtPath:。该方法返回的布尔值用来表示视频是否与 UIVideoEditorController 相兼容：

```
if (![UIVideoEditorController canEditVideoAtPath:vpath]) ...
```

如果待修改的视频能够与 `UIVideoEditorController` 相兼容,那么就新建视频编辑器。`UIVideoEditorController` 类中包含了一套由系统提供的界面,使得用户可以交互式地剪辑视频。我们设置好它的 `delegate` 及 `videoPath` 属性,并将其展示出来。(经由 `videoQuality` 属性,开发者也能通过这个类将视频重新编码成质量较低的格式。)

视频编辑器所用的 `delegate` 回调与 `UIImagePickerController` 类相似,但并不完全相同。这些回调方法分别用来处理“成功”、“失败”以及“取消”这三种情况:

- `videoEditorController:didSaveEditedVideoToPath:`
- `videoEditorController:didFailWithError:`
- `videoEditorControllerDidCancel:`

只有当用户点击视频编辑器中的 Cancel 按钮时,才会触发“取消”操作。要是在 `popover` 范围之外点击,则会直接把视频编辑器关掉,这时不会触发回调方法。如果用户取消了操作,或是 `UIVideoEditorController` 无法处理视频,那么解决方案 8-6 就会重置其界面,使得用户可以选取另一个视频。

#### 解决方案 8-6 用 `UIVideoEditorController` 编辑视频

```
// The edited video is now stored in the local tmp folder
- (void)videoEditorController:(UIVideoEditorController *)editor
  didSaveEditedVideoToPath:(NSString *)editedVideoPath
{
    [self performDismiss];

    // Update the working URL and present the Save button
    mediaURL = [NSURL URLWithString:editedVideoPath];
    self.navigationItem.leftBarButtonItem =
        BARBUTTON(@"Save", @selector(saveVideo));
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Pick", @selector(pickVideo));
}

// Handle failed edit
- (void)videoEditorController:(UIVideoEditorController *)editor
  didFailWithError:(NSError *)error
{
    [self performDismiss];
    mediaURL = nil;
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Pick", @selector(pickVideo));
    self.navigationItem.leftBarButtonItem = nil;
    NSLog(@"Video edit failed: %@", error.localizedDescription);
}

// Handle cancel by returning to Pick state
- (void)videoEditorControllerDidCancel:
    (UIVideoEditorController *)editor
{
    [self performDismiss];
```

```

mediaURL = nil;
self.navigationItem.rightBarButtonItem =
    UIBarButtonItem(@"Pick", @selector(pickVideo));
self.navigationItem.leftBarButtonItem = nil;
}

// Allow the user to edit the media with a video editor
- (void)editMedia
{
    if (![UIVideoEditorController canEditVideoAtPath:mediaURL.path])
    {
        self.title = @"Cannot Edit Video";
        self.navigationItem.rightBarButtonItem =
            UIBarButtonItem(@"Pick", @selector(pickVideo));
        return;
    }

    UIVideoEditorController *editor =
        [[UIVideoEditorController alloc] init];
    editor.videoPath = mediaURL.path;
    editor.delegate = self;
    [self presentViewController:editor];
}

// The user has selected a video. Offer an edit button.
- (void)imagePickerController:(UIImagePickerController *)picker
    didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    [self performDismiss];

    // Store the video URL and present an Edit button
    mediaURL = info[UIImagePickerControllerMediaURL];
    self.navigationItem.rightBarButtonItem =
        UIBarButtonItem(@"Edit", @selector(editMedia));
}

```

当用户编辑完毕并按下 Use 按钮时，系统会触发表示“成功”的那个回调方法。UIVideoEditorController 会把剪辑过的视频保存到临时路径，并调用 videoEditorController:didSaveEditedVideoToPath: 方法。但这并不等于说直接把视频保存到照片库，因为这条路径位于应用程序沙盒的临时文件夹中。如果我们不处理这份数据，那么设备下次重启的时候，iOS 就会将其删除。经过这一步之后，解决方案 8-6 会提供一个按钮，用来把剪辑过的视频保存到共享的 iOS 相册里，保存视频所用的技术与解决方案 8-3 相同。

## 8.8 解决方案：通过电子邮件发送图片

Message UI 框架使得用户可以在程序中编写电子邮件及文本消息。与通过 UIImage



PickerController 来使用摄像头的时候一样, 在使用这些功能之前, 也要先判断用户的设备是否支持这些服务。下面这条简单的测试语句能够判断出设备是否可以发送邮件:

```
[MFMailComposeViewController canSendMail]
```

如果设备启用了邮件功能, 那么用户就可以通过 MFMailComposeViewController 实例来发送照片了。文本消息则是通过 MFMessageComposeViewController 实例发送的。

解决方案 8-7 用 MFMailComposeViewController 类来创建新的邮件, 并把用户所拍照片放入其中。在 iPhone 及 iPad 设备上面, 最好能够以模态界面的形式来展示 MFMailComposeViewController。我们用主视图控制器将它展示出来之后, 可通过委托回调来接收用户的操作结果。

#### 解决方案 8-7 通过电子邮件发送图像

```
- (void)mailComposeController:
    (MFMailComposeViewController*)controller
    didFinishWithResult:(MFMailComposeResult)result
    error:(NSError*)error
{
    // Wrap up the composer details
    [self performDismiss];
    switch (result)
    {
        case MFMailComposeResultCancelled:
            NSLog(@"Mail was cancelled");
            break;
        case MFMailComposeResultFailed:
            NSLog(@"Mail failed");
            break;
        case MFMailComposeResultSaved:
            NSLog(@"Mail was saved");
            break;
        case MFMailComposeResultSent:
            NSLog(@"Mail was sent");
            break;
        default:
            break;
    }
}

- (void)sendImage
{
    UIImage *image = imageView.image;
    if (!image) return;

    // Customize the e-mail
    MFMailComposeViewController *mcvc =
        [[MFMailComposeViewController alloc] init];
    mcvc.mailComposeDelegate = self;
```

```

// Set the subject
[mcvc setSubject:@"Here's a great photo!"];

// Create a prefilled body
NSString *body = @"<h1>Check this out</h1>\n
<p>I snapped this image from the\
<code><b>UIImagePickerController</b></code>.</p>";
[mcvc setMessageBody:body isHTML:YES];

// Add the attachment
[mcvc addAttachmentData:UIImageJPEGRepresentation(image, 1.0f)
 mimeType:@"image/jpeg" fileName:@"pickerimage.jpg"];

// Present the e-mail composition controller
[self presentViewController:mcvc];
}

```

## 创建消息内容

通过 `MFMailComposeViewController` 的各项属性，我们可以用编程的方式来构建包含 `to/cc/bcc recipients`（直接收件人 / 抄送收件人 / 密件抄送收件人）与附件的邮件消息。解决方案 8-7 演示了如何创建带附件的简单 HTML 消息。这些属性基本上都是可选的。开发者可通过 `setSubject:` 及 `setMessageBody:` 来定义邮件主题及正文。这两个方法都接受一个字符串作为参数。

若是不指定收件人，用户则会看到一封不含收件地址的邮件消息。有时候可能需要预先填好收件地址，比方说，在实现 `Report a Bug`（报告程序错误）或 `Seed Feedback`（发送反馈）等需要联系软件开发者的功能时，就需要这样做，此外，如果程序允许用户从常用的收件人中提前选好邮件的接收方，那么开发者也需要这样做。

创建附件就要稍微复杂一点了。若想添加附件，我们必须提供邮件客户端所需的全部文件信息，包括数据（经由 `NSData` 对象提供）、`MIME` 类型（经由字符串来提供）以及文件名（经由另外一个字符串来提供）。`UIImageJPEGRepresentation()` 函数可用来获取图像数据。该函数可能要花些时间才能获取到数据，所以，在显示出邮件消息界面之前，可能会稍有延迟。

本条解决方案把 `MIME` 类型直接以硬代码的形式写成 `image/jpeg`。如果要发送其他类型的数据，那么请通过常用的文件扩展名向 iOS 查询与之对应的 `MIME` 类型。下面这个方法使用 `Mobile Core Services` 框架中的 `UTTypeCopyPreferredTagWithClass()` 函数来实现此功能：

```

#import <MobileCoreServices/UTType.h>
- (NSString *) mimeTypeForExtension: (NSString *) ext
{
    // Request the UTI for the file extension
    CFStringRef UTI = UTTypeCreatePreferredIdentifierForTag(
        kUTTagClassFilenameExtension,
        (__bridge CFStringRef) ext, NULL);
    if (!UTI) return nil;
}

```

```

// Request the MIME file type for the UTI,
// may return nil for unrecognized MIME types
NSString *mimeType = (__bridge_transfer NSString *)
    UTTypeCopyPreferredTagWithClass(UTI, kUTTagClassMIMEType);
CFRelease(UTI);
return mimeType;
}

```

上述方法会根据传入的文件扩展名返回标准的 MIME 类型，开发者可以传入 .jpg、.png、.txt、.html 等扩展名。iOS 内置了一份扩展名与 MIME 类型之间的对应关系库，但由于其中的数据比较有限，所以开发者要记得判断 mimeTypeForExtension: 方法的返回值是不是 nil。还有一种办法就是上网查找相应的 MIME 类型，然后手动将其添加到项目里。

通过电子邮件发送数据时，开发者要给这份数据起个文件名，选用任意名称都可以。范例程序使用的名称是 pickerimage.jpg。由于我们只是想演示一下数据发送功能，因此文件名并不需要和发送的内容有所关联：

```

[mcvc addAttachmentData:UIImageJPEGRepresentation(image, 1.0f)
    mimeType:@"image/jpeg" fileName:@"pickerimage.jpg"];

```



通过 iOS 的 MFMailComposeViewController 来撰写邮件时，附件会出现在所发邮件的末尾。由于苹果公司和 Microsoft 在邮件的表现形式上有区别，所以苹果公司并未提供把图像直接嵌入 HTML 文本的方式。

## 8.9 解决方案：发送文本消息

在应用程序里发短信要比发送电子邮件更简单。图 8-4 演示了相关的控制器。与发送邮件时的做法一样，我们也要先确保 iOS 设备具有发送文本消息的能力，此外，程序的控制器还要遵循 MFMessageComposeViewControllerDelegate 协议：

```
[MFMessageComposeViewController canSendText]
```

设备有时能够发送短信，有时则不能，我们可以监听 MFMessageComposeViewControllerTextMessageAvailabilityDidChangeNotification 通知来获知这一变化。

解决方案 8-8 新建了 MFMessageComposeViewController，并设置了它的 messageComposeDelegate 及 body。如果预先知道收信人，那么可以把表示电话号码的一些字符串放在数组里，传给该控制器。按照自己的方式把控制器展示出来之后，我们就等待委托回调，这个回调方法应该将控制器关闭。

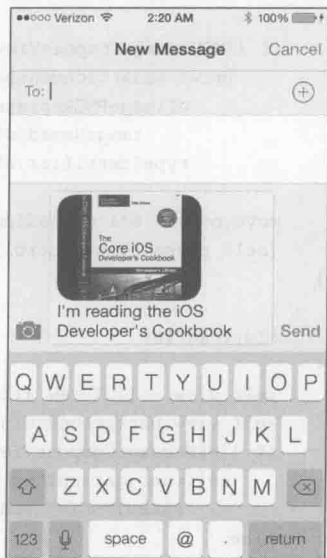


图 8-4 通过 MFMessageComposeViewController 来编写文本消息

## 解决方案 8-8 发送文本消息

```

- (void)messageComposeViewController:
    (MFMessageComposeViewController *)controller
    didFinishWithResult: (MessageComposeResult) result
{
    [self performDismiss];

    switch (result)
    {
        case MessageComposeResultCancelled:
            NSLog(@"Message was cancelled");
            break;
        case MessageComposeResultFailed:
            NSLog(@"Message failed");
            break;
        case MessageComposeResultSent:
            NSLog(@"Message was sent");
            break;
        default:
            break;
    }
}

- (void)sendMessage
{
    MFMessageComposeViewController *mvc =
        [[MFMessageComposeViewController alloc] init];
    mvc.messageComposeDelegate = self;

    if ([MFMessageComposeViewController canSendAttachments])
        [mvc addAttachmentData:
            UIImagePNGRepresentation([UIImage
                imageNamed:@"BookCover"])
            typeIdentifier:@"png" filename:@"BookCover.png"];

    mvc.body = @"I'm reading the iOS Developer's Cookbook";
    [self presentViewController:mvc];
}

- (void)loadView
{
    self.view = [[UIView alloc] init];
    self.view.backgroundColor = [UIColor whiteColor];
    if ([MFMessageComposeViewController canSendText])
        self.navigationItem.rightBarButtonItem =
            BARBUTTON(@"Send", @selector(sendMessage));
    else
        self.title = @"Cannot send texts";
}

```

iOS 7 的 `MFMessageComposeViewController` 已经有发送附件的功能了。在添加附件之前，请先用 `MFMessageComposeViewController` 的类方法 `canSendAttachments` 来判断设备是否能发送附件。如果可以，那么解决方案 8-8 就会给消息里面添加一张图片。

## 8.10 解决方案：在社交网站发布消息

Social 框架提供了一套统一的 API，能够把应用程序与社交网站结合起来。该框架目前支持 Facebook、Twitter 以及中国的新浪微博（Sina Weibo）和腾讯微博（Tencent Weibo）。与发送邮件及编写消息时类似，在使用某服务之前，先要判断系统是否支持该服务：

```
[SLComposeViewController isAvailableForServiceType:SLServiceTypeFacebook]
```

如果支持，那么就可以创建针对该服务的 `SLComposeViewController` 了。

```
SLComposeViewController *fbController = [SLComposeViewController  
composeViewControllerForServiceType:SLServiceTypeFacebook];
```

我们用图像、URL 以及一段文本来定制这个控制器。解决方案 8-9 按步骤演示了如何创建如图 8-5 中所示的界面。



图 8-5 编写 Twitter 消息



iOS 6 在引入 `SLComposeViewController` 之后，iOS 5 的 `TWTweetComposeViewController` 就废弃了。虽说这两套 API 基本相同，但是废弃的那个版本里面有一些容易出错的地方需要回避。所以我们应该使用 `SLComposeViewController` 版本来分享 Twitter 消息。

## 解决方案 8-9 在社交网站发布消息

```

- (void)postSocial:(NSString *)serviceType
{
    // Establish the controller
    SLComposeViewController *controller = [SLComposeViewController
        composeViewControllerForServiceType:serviceType];

    // Add text and an image
    [controller addImage:[UIImage imageNamed:@"BookCover"]];
    [controller setInitialText:
        @"I'm reading the iOS Developer's Cookbook"];

    // Define the completion handler
    controller.completionHandler =
        ^(SLComposeViewControllerResult result){
            switch (result)
            {
                case SLComposeViewControllerResultCancelled:
                    NSLog(@"Cancelled");
                    break;
                case SLComposeViewControllerResultDone:
                    NSLog(@"Posted");
                    break;
                default:
                    break;
            }
        };

    // Present the controller
    [self presentViewController:controller];
}

- (void)postToFacebook
{
    [self postSocial:SLServiceTypeFacebook];
}

- (void)postToTwitter
{
    [self postSocial:SLServiceTypeTwitter];
}

- (void)loadView
{
    self.view = [[UIView alloc] init];
    self.view.backgroundColor = [UIColor whiteColor];
    if ([SLComposeViewController
        isAvailableForServiceType:SLServiceTypeFacebook])
    {
        self.navigationItem.leftBarButtonItem =
            UIBarButtonItem(@"Facebook", @selector(postToFacebook));
    }
}

```

```

    if ([SLComposeViewController
        isAvailableForServiceType:SLServiceTypeTwitter])
    {
        self.navigationItem.rightBarButtonItem =
            UIBarButtonItem(barButtonSystemItem:UIBarButtonSystemItemPost,
                           target:self,
                           action:@selector(postToTwitter));
    }
}

```

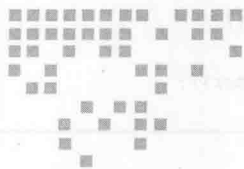
## 数据的分享与查看

上面几条解决方案分别演示了如何发送电子邮件、如何发送文本消息以及如何在社交网站发布消息。iOS 提供了一些控制器，可以简化数据的分享与查看操作，它们支持很多种数据类型，也支持很多种发布渠道。UIActivityViewController 可以把各种数据分享操作集中起来，它内置的功能包括发送电子邮件、在社交网站发布消息、打印等，此外，开发者还能够扩展这个控制器，用以支持自定义的操作。QLPreviewController 使得应用程序能够查看许多种自己不太能够处理的数据。第 11 章将会详述这两个类。

### 8.11 小结

本章介绍了一些现成的控制器，开发者可以用它们制备出各种优秀的功能。通过系统所提供的这些控制器，我们可以用代码轻松地实现出分享 Twitter 消息以及发送电子邮件等常见的任务。大家现在来回顾一下前面讲过的问题：

- 尽管我们自己也能实现出这些控制器，但是没有必要这么做。系统所提供的控制器可以在开发者自己设计的程序与其他程序之间保持协调一致的用户体验，我们很少能遇到这样好用的类了。无论采用哪个程序来编写邮件，用户看到的界面都是相同的。开发者在编写发送电子邮件、分享 Twitter 消息以及访问系统媒体库等功能的时候，应该通过苹果公司的系统服务来实现才对。
- UIImagePickerController 这个类现在变得有些笨拙，其实它早就应该重新设计了。我们可以调整媒体来源，以减少它的内存用量，但同时也希望苹果公司能把这个类改善一下。有很多功能强大的媒体处理类已经移植到 iOS 了，所以 UIImagePickerController 类不应该只是个可以显示出来的控制器，它还应该与 AV Foundation、Core Media 以及其他一些关键技术集成起来才好。尊重用户隐私固然非常重要，但如果系统能够多开放一些 API 的话，就更好了（这些 API 当然要在获得用户的许可之后方能执行）。
- 除了能在 Facebook 和 Twitter 上面发表消息，Social 框架还能执行很多事情。通过 SLComposeViewController 类，我们可以用适当的安全机制来提交已认证（authenticated）和未认证（unauthenticated）的各种服务请求。把 Accounts 框架同 Social 框架结合起来，就能获取受过验证的登录信息，并以该账户的身份发送相关请求。



## 创建并管理表格视图

表格是一种基于滚动列表的互动类，它尤其适合展示小型 GUI 元件。iPhone 与 iPod touch 自带许多程序，其导航方式都以表格为中心，包括 Contacts、Settings 及 iPod 播放界面等。由于这些小型 iOS 设备的屏幕空间有限，因此用可以滚动的表格来展示每个备选条目是一种比较理想的信息传递方式，它能把内容以简洁、易操作的形式展现出来。iPad 的屏幕则比较大，这使得表格能够与更大的细节展示视图集成起来，从而成为分栏视图控制器（split view controller）中的重要组件。本章要讲解 iOS 表格的运作方式、可供开发者使用的表格类型以及怎样在自己的程序中使用表格的各项特性。

### 9.1 iOS 的表格

标准的 iOS 表格是一张包含单元格的垂直滚动列表。用户可以向上、向下滚动或滑动这张表格，直到发现自己想要操作的内容。iOS 里面到处都能看见表格。许多内置的 iOS 程序完全以表格为基础来构建，而且大量的第三程序也把表格作为核心展示形式。

iOS 中的大部分表格都是用 UITableView 构建的，而且通过委托及数据源协议所提供的各种选项做了定制。通用的表格实现形式是把单元格放在垂直滚动的列表里，不过我们也可以创建带有自定义图案、背景色、标签等元素的专用表格。

这些特别的表格包括 Preferences 程序里面那种把白色单元格放在灰色背景上的表格，以及 Contacts 程序里面那种分成不同区域（section）并且带有索引（index）的表格，此外还有一些与滚动式表格有关的类，例如设置约会时间及闹钟所用的那种表格。如果想展示网格状的界面，而不仅仅是表格和滚动列表，那么可以使用与集合视图有关的一些类，第 10 章将会讲述这些类。



无论使用哪种表格，其运作方式通常都是相同的。表格是遵照 Model-View-Controller (模型-视图-控制器, MVC) 范式构建的。它们会把数据源以单元格的形式展示出来，并且通过定义明确的委托方法来响应用户的操作。

数据源是一个类，它可以根据需求来提供与表格内容有关的信息。它代表底层数据模型，而且可以协调模型与表格视图之间的关系。数据源会将其结构告知表格。比方说，它会告诉表格应该使用多少个区域以及每个区域包含多少个条目。数据源可以按照需要分别提供表格里的每个单元格，并根据单元格在表中的位置，以数据模型来填充这些单元格。

数据源相当于表格的模型，而委托则相当于控制器。委托用来管理用户的操作，当用户想要选择表格内容或想要编辑表格时，应用程序可以通过委托来响应这些变化。例如，用户可能想要点选新的单元格，或是想将某单元格放在另一个位置，也有可能是想添加或删除单元格等。委托可以监控用户的这些操作请求，允许或禁止它们，在操作成功之后，还能更新数据模型以反映这一变化。

视图、数据源与委托三者搭配起来，体现了一种 MVC 开发模式。这种模式并不局限于表格视图。很多关键的 iOS 类里都能看到这样的“视图/数据源/委托”组合。选取器视图、集合视图以及页面视图控制器都用到了数据源与委托。

## 9.2 委托

表格视图的数据源与委托是一种委托，也就是把特定的操作和信息交给另一个辅助对象来处理。UIKit 中的很多类都通过委托机制来响应用户操作并提供相关内容。比方说，设置了表格的委托之后，系统就会把与交互操作有关的消息传给它，并且令这个委托来处理这些消息。

表格视图很好地说明了委托的优点。用户点击表格的某一行时，UITableView 实例并没有提供内置的方式来响应这次触摸。它是个通用的类，所以本身并没有提供与点击操作相关的语义。用户点选某个单元格时，表格会询问其委托（这个委托通常是个视图控制器类），并把与这次操作有关的变更信息传过去。开发者可以在委托里面添加与处理点击操作所用的代码，这样就把添加此种代码的时间点与苹果公司创建表格类的时间点完全分隔开。通过委托机制，我们可以创建一种没有具体含义的类，而把与程序有关的具体处理代码留给开发者稍后去编写。

UITableView 里有个委托方法叫作 `tableView:didSelectRowAtIndexPath:`<sup>①</sup>，它很好地演示了委托这一概念。委托对象需要定义这个方法，并指明应用程序应该如何响应由用户所发起的选定操作。开发者可以显示一份菜单，或是跳转到子视图，也可以在用户点击的这一行里添上选取符号。具体的响应方式完全取决于开发者如何来实现这个处理选取操作的委托方法。在实现表格类本身的时候，系统完全不知道这些内容。

通过 `delegate` 及 `dataSource` 属性，我们可以设置表格视图的委托及数据源。应用程序会把与交互操作有关的回调传给开发者所赋的相关对象。为了使 Objective-C 知道这些对象实现了委托方法，我们必须在对象所属的类上面声明这些类遵从了相关的协议才

① 这是一种习惯称谓，严格来说，应该是 UITableViewDelegate 协议中的 `tableView:didSelectRowAtIndexPath:` 方法，下同。——译者注

行。在声明了某个类继承自何类之后，可以在右边放上一对尖括号，并把本类所遵循的协议置于其中（例如 `<UITableViewDelegate>` 或 `<UITableViewDataSource>`）。如果要宣称该类遵循多项协议，那么就用逗号将各协议隔开，并把它们一起放在尖括号内（例如 `<UITableViewDelegate, UITableViewDataSource>`）。遵循某项协议之后，该类就必须实现协议中规定的必备方法，同时也可以实现一些可选方法。

## 9.3 创建表格

iOS 里面主要有两个与表格有关的类，一个是预先构建好的控制器类（`UITableViewController`），另一个是可以直接显示出来的视图类（`UITableView`）。控制器是个专门为表格而定制的 `UIViewController` 子类，它所建立的表格视图会完全占据整个控制器的视图空间，而且它还给开发者省去了很多使用表格实例时所重复执行的编码工作。尤为重要，它已经把使用表格时所需遵从的协议全部声明好了，并将自身设为表格的 `delegate` 与数据源。如果要在该控制器类之外使用表格视图，那么开发者必须手工编写这些代码，而 `UITableViewController` 则会自动处理好这些事。

### 9.3.1 表格的样式

iPhone 上面的表格有两种形式：普通表格与分组表格。在默认情况下，普通表格的背景是白色的，上面会有透明的单元格。iOS 的 Settings 程序使用分组形式的表格，这种表格的背景是浅灰色的，其中每个分区的背景是白色的。

若想改变表格样式，只需在初始化 `UITableViewController` 的时候指定另外一种样式即可。在新建实例的时候，可以明确指定表格样式。一旦初始化完毕，就不能再更改了。下面给出范例代码：

```
myTableViewController = [[UITableViewController alloc]
    initWithStyle:UITableViewStyleGrouped];
```

如果使用的控制器是从 XIB 或故事板中加载的，那么可以在 Xcode 的 Attributes Inspector 里调整 Table View > Style 属性。

### 9.3.2 排布表格视图

正如其名称所示，`UITableView` 实例是一种在 iOS 屏幕中展示互动式表格的视图。因为 `UITableView` 类继承自 `UIScrollView` 类，所以表格具备上下滚动的功能。与其他视图一样，`UITableView` 实例也通过 `frame`（框架）来定义自身的边界，而且可以成为其他视图的子视图或上级视图。要创建表格视图，开发者需要分配内存，并且用 `frame` 或 `Auto Layout` 约束规则来初始化它，然后通过设置数据源与委托对象来添加所有的细节处理代码。

`UITableViewController` 会自己把视图排布好。该类创建标准的视图控制器，在其中放置 `UITableView`，并设置其 `frame`，以便给导航栏或工具栏等留出空间。我们可通过 `tableView` 实例变量来访问 `UITableView`（表格视图）。

### 9.3.3 设置数据源

UITableView 实例依赖于一种外部来源,该来源可以根据程序需要来提供新的单元格,或向现有的单元格内填充新数据。单元格 (Cell) 是一种构成表格的小视图,用以表示表格里每一行的内容。这种外部的数据来源就叫作数据源,它是指负责根据程序的需求来向表格提供单元格的对象。

设为表格 dataSource 属性的那个对象负责向表格提供单元格以及其他布局信息。该对象必须宣称自己遵循 UITableViewDataSource 协议,并实现协议中的相关方法。除了能返回单元格之外,表格的数据源还指定了表格中的分区数、每个分区的单元格数、分区的标题、单元格的高度等内容,此外,它也可以提供一份可选的目录。数据源定义了表格的样貌以及填充在表格中的内容。

一般来说,拥有表格视图的那个视图控制器就是表格视图的数据源。如果使用 UITableViewController 的子类来开发,那么就不需要再声明它遵循 UITableViewDataSource 协议了,因为父类本身就支持该协议,而且还会把控制器自动设为数据源。

### 9.3.4 提供单元格

表格的数据源需要实现 tableView:cellForRowAtIndexPath: 方法,以便向表格提供单元格。只要调用了表格的 reloadData 方法,表格就会向数据源索要数据,以便把屏幕上将要显示出来的单元格填充好。开发者可以随时用代码来调用 reloadData 方法,这样做会迫使表格重新加载其内容。

系统在调用 tableView:cellForRowAtIndexPath: 方法的时候,数据源应该根据 indexPath 参数中的索引路径来向表格提供单元格。索引路径是 NSIndexPath 类的对象,它描述了从数据树到某个节点所经的路径,这条路径实际上就是单元格所在的分区及行。用分区和行可以创建出 NSIndexPath:

```
NSIndexPath *myIndexPath = [NSIndexPath indexPathForRow:5 inSection:0];
```

我们可以用 section 把表格里数据按照逻辑分成组,然后在每个 section 中,用 row 来表示单元格的位置。数据源负责把 NSIndexPath 和具体的 UITableViewCell 实例对应起来,并根据需求向表格提供单元格。

### 9.3.5 注册单元格类

创建表格视图的时候,应该及早注册表格所用的单元格类型。注册了之后,UITableView 的 dequeueReusableCellWithIdentifierWithIdentifier 方法就能自动创建新的单元格了。一般来说,我们会在初始化方法、loadView 方法或 viewDidLoad 方法里面注册单元格。这个注册步骤必须在表格初次加载其数据之前完成。每个 UITableView 实例都需要注册它自己用到的单元格类型。注册的时候,开发者可将一个字符串用作标识符,稍后在索要新的单元格时,可以把这个字符串当成键。

我们可以按照类或 XIB 来注册单元格,前一种方式适用于 iOS 6 及后续系统,后一种方

式适用于 iOS 5 及后续系统。下面这段范例代码演示了这两种注册方式：

```
[self.tableView registerClass:[UITableViewCell class]
    forCellReuseIdentifier:@"table cell"];
[self.tableView registerNib:
    [UINib nibWithNibName:@"CustomCell" bundle:[NSBundle mainBundle]]
    forCellReuseIdentifier:@"custom cell"];
```

开发者可以注册很多种单元格，并不是说一张表格只能使用一种单元格。我们可以根据程序的需求，在同一张表格中混合使用多种类型的单元格。

### 9.3.6 从队列中取出单元格

当程序向数据源索要单元格的时候，它可以通过代码来构建 `UITableViewCell`，也可以从 Interface Builder (IB) 的资源中加载 `UITableViewCell`。下面是个极为简单的数据源方法，它会根据所请求的 `NSIndexPath` 返回单元格，并把数据模型中的相关文本用作这个单元格的标签：

```
- (UITableViewCell *)tableView:(UITableView *)aTableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [self.tableView
        dequeueReusableCellWithIdentifier:@"cell"
        forIndexPath:indexPath];
    cell.textLabel.text =
        [dataModel objectAtIndex:indexPath.index];
    return cell;
}
```

如果你是位 iOS 开发老手，那么应该会觉得这个方法挺好的，因为我们不需要再去判断队列中有没有所需类型的单元格了。无论有没有我们需要的单元格，队列都会在必要的时候创建并初始化新的 `UITableViewCell` 实例。

我们通过队列机制来索要单元格。单元格在离开表格的可视范围之后，就会缓存到队列里，以便稍后重新使用。队列机制会把保存在队列中的可用单元格返回给调用者，如果队列中没有这种单元格，那它就会自行创建并返回新的单元格实例。

把可以复用的单元格类型注册好之后，每个单元格实例就会有一个标识符标签 (identifier tag)。在需要用到单元格的时候，表格会按照类型在队列中搜寻，并把可以复用的单元格取出来。这样做可以节省内存，此外，如果用户快速地滚动一份很长的列表，那么这种做法还能够迅速而高效地提供表格所需的单元格。

### 9.3.7 设置 delegate

与 Cocoa Touch 中的很多交互式对象一样，`UITableView` 实例也通过 delegate 来响应用户的操作并做出适当的应答。表格的 delegate 可以响应诸如表格滚动、用户编辑或选中某行等事件。通过委托机制，表格把响应这些操作的责任交给开发者所指定的委托对象，而这个委托对象通常就是拥有表格视图的那个控制器对象。

如果想直接使用 `UITableView`，那么就把它 `delegate` 属性设置成可以响应相关事件的某个对象。这个 `delegate` 对象所属的类必须宣称自己实现了 `UITableViewDelegate` 协议。与 `dataSource` 属性一样，如果我们直接使用 `UITableViewController` 类或是从中继承子类，那么就无须设置 `delegate`，也不用再宣称该类遵循 `UITableViewDelegate` 协议了。

## 9.4 解决方案：实现简单的表格

要想实现一张简单的表格，我们只需提供一些数据，用以设置单元格的标签（label），然后再实现几个方法就行了。解决方案 9-1 就提供了这样一种极其简单的表格。这张表格是不分区的，其效果如图 9-1 所示。每个单元格都有文本标签及图像，这种图像是中间写有行号的方格。

用户可以点击这些单元格。点击之后，控制器的标题文本会变得和单元格中的文本相同。`Deselect` 按钮可以移除当前的选定效果，并重新设置控制器的标题。`Find` 按钮则会移动表格，使用户所选的那个单元格能够出现在屏幕中，如果用户把某个单元格滚动到了屏幕范围之外，那么点击这个按钮就可以令其重新回到屏幕范围内。

点击 `Find` 按钮之后，范例代码会试着滚动表格，如果空间允许的话，用户所选的单元格就会出现在视图的顶端（`UITableViewScrollPositionTop`）。表格中的最后一项（也就是 `Zulu`）是不能向上滚动的，它只能留在视图底部，因为后面没有其他单元格了。



图 9-1 由解决方案 9-1 所构建的简单表格视图

### 9.4.1 数据源方法

即便想显示解决方案 9-1 这样简单的表格，其数据源也必须实现三个核心的实例方法才行。这些方法定义了表格的组织方式，并且负责向表格提供内容：

- **`numberOfSectionsInTableView`**——表格可以把数据显示到不同的区域中，也可以把所有数据都放在一份清单里。如果想创建普通的表格，那么就令该方法返回 1。这表示整张表都会显示为一份清单。若想创建分区显示的表格，则返回 2 或 2 以上的值。
- **`tableView:numberOfRowsInSection:`**——该方法返回某个分区内的行数。对于解决方案 9-1 这种普通的表格来说，这个方法返回的就是整张表格的行数。如果表格更为复杂一些，那么就要设法查出每个分区所拥有的行数了。我们在 12 章将会看到，`Core Data` 非常适合与这种分成不同区域的表格相集成。与 `iOS` 中的所有计数机制一样，分区的序号也从 0 开始，首个分区的序号是 0。

- **tableView:cellForRowAtIndexPath:**——该方法应该返回调用表格所需的单元格。开发者需要根据索引路径 (NSIndexPath) 中的 row 和 section 属性来决定返回什么样的单元格，同时也应该利用可复用的单元格来尽量降低内存用量。

### 解决方案 9-1 构建简单的表格

```
@implementation TestBedViewController
{
    UIFont *imageFont;
    NSArray *items;
}

// Number of sections
- (NSInteger)numberOfSectionsInTableView:(UITableView *)aTableView
{
    return 1;
}

// Rows per section
- (NSInteger)tableView:(UITableView *)aTableView
    numberOfRowsInSection:(NSInteger)section
{
    return items.count;
}

// Return a cell for the index path
- (UITableViewCell *)tableView:(UITableView *)aTableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [self.tableView
        dequeueReusableCellWithIdentifier:@"cell"
        forIndexPath:indexPath];

    // Cell label
    cell.textLabel.text = items[indexPath.row];

    // Cell image
    NSString *indexString =
        [NSString stringWithFormat:@"%02d", indexPath.row];
    cell.imageView.image =
        stringImage(indexString, imageFont, 6.0f);

    return cell;
}

// On selection, update the title and enable find/deselect
- (void)tableView:(UITableView *)aTableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
```

```

        [self.tableView cellForRowAtIndexPath:indexPath];
        self.title = cell.textLabel.text;
        self.navigationItem.rightBarButtonItem.enabled = YES;
        self.navigationItem.leftBarButtonItem.enabled = YES;
    }

    // Deselect any current selection
    - (void)deselect
    {
        NSArray *paths = [self.tableView indexPathsForSelectedRows];
        if (!paths.count) return;

        NSIndexPath *path = paths[0];
        [self.tableView deselectRowAtIndexPath:path animated:YES];
        self.navigationItem.rightBarButtonItem.enabled = NO;
        self.navigationItem.leftBarButtonItem.enabled = NO;

        self.title = nil;
    }

    // Move to the selection
    - (void)find
    {
        [self.tableView scrollToNearestSelectedRowAtScrollPosition:
            UITableViewScrollPositionTop animated:YES];
    }

    // Set up table
    - (void)viewDidLoad
    {
        [super viewDidLoad];
        self.view.backgroundColor = [UIColor whiteColor];

        self.navigationItem.rightBarButtonItem =
            BARBUTTON(@"Deselect", @selector(deselect));
        self.navigationItem.leftBarButtonItem =
            BARBUTTON(@"Find", @selector(find));
        self.navigationItem.rightBarButtonItem.enabled = NO;
        self.navigationItem.leftBarButtonItem.enabled = NO;

        imageFont = [UIFont fontWithName:@"Futura" size:18.0f];

        [self.tableView registerClass:[UITableViewCell class]
            forCellReuseIdentifier:@"cell"];
        items = [@"Alpha Bravo Charlie Delta Echo Foxtrot Golf \
            Hotel India Juliet Kilo Lima Mike November Oscar Papa \
            Quebec Romeo Sierra Tango Uniform Victor Whiskey Xray \
            Yankee Zulu" componentsSeparatedByString:@" "];
    }
@end

```



---

### 获取解决方案代码

访问 <https://github.com/erica/iOS-7-Cookbook> 网页，并打开“C09 Tables”文件夹，即可找到与本章中的解决方案相对应的完整范例项目。

---

## 9.4.2 响应用户的触摸

解决方案 9-1 在名为 `tableView:didSelectRowAtIndexPath:` 的委托方法中响应用户的操作。如果用户点击某单元格，那么这条解决方案就会更新视图控制器的标题，并启用 Find 及 Deselect 按钮。只要用户所选内容有效，这两个按钮就会一直处于启用状态。用户点击 Deselect 按钮后，范例代码会调用 `deselectRowAtIndexPath:animated:` 方法，并禁用这两个按钮。



**提示** 如果想令表格的某个单元格忽略用户的触摸操作，那么可以将 `selectionStyle` 属性设为 `UITableViewCellStyleNone`。用户选中某个单元格之后，这个单元格本来应该变为灰色，但设置了该属性之后，它就不会变灰了。此时这个单元格仍然处在选中状态，但它并不会以视觉效果来强调这一状态。若是點選单元格这一操作除了展示信息之外还会产生其他效果，那么这个办法恐怕就不是最佳方案了。

---

## 9.5 UITableViewCell 类

`UITableViewCell` 类提供了四种实用的基本样式，如图 9-2 所示。该类有两个文本标签属性，一个是 `textLabel`，表示单元格的主标题，另一个是 `detailTextLabel`，用来创建子标题。这四种基本的样式分别是：

- **`UITableViewCellStyleDefault`**——这种单元格具备一个左对齐的文本标签，而且可以指定一幅图像。如果使用了图像，那么可以显示文本的空间就变小了，标签就会出现在图像右侧。开发者可以访问并修改 `detailTextLabel`，但它并不会出现在屏幕上。
- **`UITableViewCellStyleValue1`**——这种风格会把主标签以较大的黑字显示在单元格左侧，而把子标题以较小的灰字演示在单元格右侧。
- **`UITableViewCellStyleValue2`**——这种风格会以当前的 `tintColor` 为文本颜色，把主标签用小字显示在左侧，而子标题则会以黑色小字出现在其右方。由于能够显示主标签的那块地方很窄，所以在大部分情况下，文本都没办法完整显示出来，单元格会用省略号表示没显示出来的那些内容。这种单元格不支持图像。
- **`UITableViewCellStyleSubtitle`**——这种单元格会把标准的文本标签显示在稍微靠上一些的位置，从而给下方留出空间，用以显示细节标签。这两个标签文本都是黑色的。与默认风格的单元格一样，这种单元格也可以设置图像。



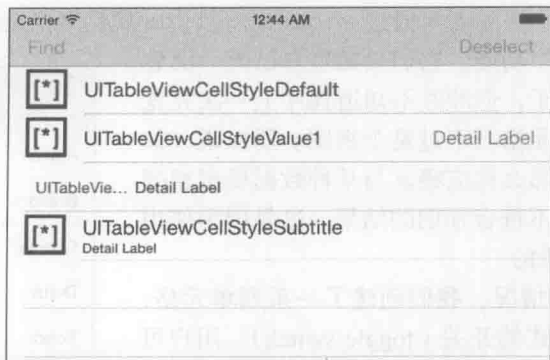


图 9-2 Cocoa Touch 提供了四种标准的单元格类型，其中有好几种都可以设置图像

### 9.5.1 单元格的 `selectionStyle` 属性

通过 `selectionStyle` 属性可以设置用户所选单元格的样貌。在 iOS 7 中，无论是把该属性设置成 `UITableViewCellStyleBlue`，还是把它设置成 `UITableViewCellStyleGray`，用户所选的单元格都会呈现浅灰色背景（虽说前者的字面意思是蓝色，但实际上还是浅灰色）。假如不想展示单元格受选时的视觉效果，那么就把该属性设为 `UITableViewCellStyleNone`。用户依然可以选择这种单元格，只是其背景不会呈现灰色。

### 9.5.2 添加自定义的单元格受选效果

当用户选中某个单元格时，可以通过 Cocoa Touch 所提供的功能来强化该单元格受选时的效果。开发者可以修改相关属性，以此来定制单元格受选时的行为，从而把它与别的单元格区分开。与之相关的属性有两个。

`selectedBackgroundView` 属性可以给用户所选的单元格上面添加控件或其他视图。这类似于出现在键盘上面的辅助视图。我们可以在用户所选单元格的背景视图上面添加预览按钮或购买选项。

此外，还可以修改单元格标签的 `highlightedTextColor` 属性，使得用户在选中单元格之后，其中的标签文本能够以另一种颜色显示出来。

## 9.6 解决方案：创建带有选取标记的单元格

开发者能够通过辅助视图来扩充普通 `UITableViewCell` 的功能。我们可以用选取标记（check mark）来创建图 9-3 这样的交互式单选（one-of-n selection）表或多选（n-of-n selection）表。用户可以用这样的表格来点餐或是选取想要更新的条目。

为了给受选条目添加选取标记，需要把单元格的 `accessoryType` 属性设为 `UITableViewCellAccessoryCheckmark`。若要取消选取标记，则应将其设为 `UITableViewCellAccessoryNone`。

CellAccessoryNone。开发者可通过 accessoryType 属性来指定辅助视图中的图案。

单元格没有“记忆”功能，它们只知道自己上一次显示出来的时候是什么样子，但却并不知道程序上一次究竟是怎样使用自己的。单元格只不过是个视图。所以说，如果要复用单元格的话，那么就应将其与某种数据模型相绑定，否则则会无意间产生不符合预期的结果，这是因为使用了 MVC 设计范式而导致的。

比方说有下面这种情况。我们创建了一系列单元格，每个单元格都带有切换式的开关（toggle switch）。用户可以操作开关来修改其值。如果某单元格离开了屏幕，那么就会出现于复用队列中。假如这个单元格在离开屏幕之前其开关处于打开状态，那么当表格稍后把该单元格复用到其他元素上面的时候，用户就会发现自己还没有点击那个元素，它的开关却已经打开了。

为了修复这个问题，我们应该把单元格的的状态与保存起来的模型相比对，并在 cellForRowAtIndexPath: 方法中完整地配置该单元格。这样做就使得视图的显示效果总能与应用程序的数据保持一致，从而避开了上次使用单元格时所残留下来的状态信息。单元格上面的开关状态只能影响这个单元格的样貌，并不能说明与其关联的那个逻辑条目也处于打开状态。由于可复用的单元格会保留上次使用时的受选或未受选状态，所以我们必须修改 accessoryType 属性，使其与模型的状态相符，而不能沿用单元格上次的样貌。

解决方案 9-2 构建了一份简单的状态字典，用以保存每条索引路径所对应的开 / 关状态。它的数据源方法会用字典中的相关状态来初始化单元格，并将其返回给调用者。我们只需稍稍扩充这条解决方案，就能把各元素的状态存入 NSUserDefaults 之中，以便在下次启动程序时恢复它们。这项改进添加起来很容易，所以就留给读者作为练习吧。

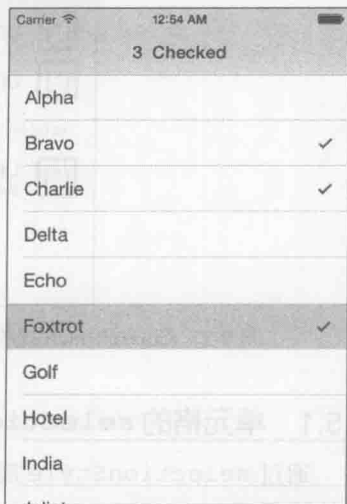


图 9-3 通过辅助视图来添加选取标记，我们就能非常方便地实现出单选表或多选表了

#### 解决方案 9-2 添加辅助视图并保存单元格状态

```
// Return a cell populated with data model state for the index path
- (UITableViewCell *)tableView:(UITableView *)aTableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [self.tableView
        dequeueReusableCellWithIdentifier:@"cell"
        forIndexPath:indexPath];

    // Cell label
    cell.textLabel.text = items[indexPath.row];
    BOOL isChecked =
        ((NSNumber *)stateDictionary[indexPath]).boolValue;
    cell.accessoryType = isChecked ?
```

```

        UITableViewCellAccessoryCheckmark :
        UITableViewCellAccessoryNone;

    return cell;
}

// On selection, update the title
- (void)tableView:(UITableView *)aTableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
        [self.tableView cellForRowAtIndexPath:indexPath];

    // Toggle the cell checked state
    BOOL isChecked =
        !((NSNumber *)stateDictionary[indexPath]).boolValue;
    stateDictionary[indexPath] = @(isChecked);
    cell.accessoryType = isChecked ?
        UITableViewCellAccessoryCheckmark :
        UITableViewCellAccessoryNone;

    // Count the checked items
    int numChecked = 0;
    for (NSUInteger row = 0; row < items.count; row++)
    {
        NSIndexPath *path =
            [NSIndexPath indexPathForRow:row inSection:0];
        isChecked =
            ((NSNumber *)stateDictionary[path]).boolValue;
        if (isChecked) numChecked++;
    }

    self.title = [@"@(numChecked).stringValue, @" Checked"]
        componentsJoinedByString:@" ";
}

```

## 9.7 给单元格添加详情展示控件

单元格右侧可以出现两种详情展示控件，一种是指向右方的灰色 V 形图案，另一种是以 tint color 来渲染的信息按钮 (info button)。通过详情展示控件，我们可以把单元格与支持该单元格的视图联系起来。在 iPhone 和 iPod touch 上面的 Contacts 程序中，用户可以在联系人列表里点击指向右方的灰色 V 形图案，以编辑某个联系人的信息，或是在 Calendar 程序里通过点击这个图案来安排某次约见。图 9-4 演示了两种详情展示控件，这张表格里每个单元格的右侧都有详情展示控件。

在 iPad 上面，我们应该考虑使用分栏视图控制器，而不要采用详情展示控件。因为 iPad 的屏幕空间较大，所以应该把各元素以列表形式展示在屏幕左边，同时把细节视图展示

在右边。这样做的效果与 iPhone 上面通过 V 形详情展示控件想要达到的效果是类似的。

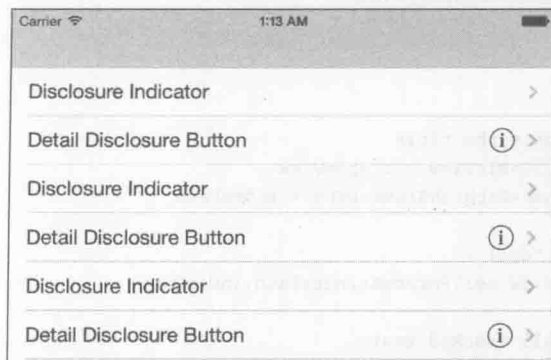


图 9-4 指向右方的灰色 V 形图案与带圆圈的 i 字按钮都是详情展示控件，用户可以通过点击它们来跳转到另一个视图

详情展示控件可以有下面两种样貌：

- `UITableViewCellAccessoryDetailDisclosureButton` 表示外面带有圆圈的 i 字按钮，按钮右侧还有个灰色的 V 形图案。该按钮能够追踪用户的触摸操作，这种按钮表示用户点击了单元格之后，程序会跳转到完全具备互动能力的细节视图。
- `UITableViewCellAccessoryDisclosureIndicator` 表示灰色的 V 形图案，该图案并不追踪触摸操作，它表示用户点击了该单元格之后，程序会跳转到选项视图，具体来说，这个视图里面会包含一些与用户所选单元格有关的选项。

iPhone 的 Settings 程序里就能看到这两种图案。点击带有扩展指示器 (disclosure indicator) 的 WiFi 单元格，即可切换到 WiFi 画面。在这个画面中，我们可以通过详情展示按钮 (detail disclosure) 来查看某个 WiFi 接入点的具体信息，包括它的 IP 地址、子网掩码、网关、DNS 信息等。

如果屏幕上面将要显示一份与当前单元格有关的子菜单，那么就应该使用扩展指示器。凡是要显示这种子菜单的场合都应该使用简单的灰色 V 形图案。请记住一条经验：灰色 V 形图案用于显示子菜单，而信息按钮则用于定制某个对象。如果使用扩展指示器，那么程序代码就应该响应用户对单元格的点击；若使用详情展示按钮，则应响应用户对该按钮的点击。

下面这段代码会把每个单元格的 `accessoryType` 属性都设为 `UITableViewCellAccessoryDetailDisclosureButton`。它还会把 `editingAccessoryType` 属性设为 `UITableViewCellAccessoryNone`：

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:@"CustomCell"];
```

① 如果只想显示带圆圈的 i 字按钮，而不想显示其右侧的灰色 V 形图案，那么应该使用 `UITableViewCellAccessoryDetailButton`。——译者注

```

        cell.accessoryType =
            UITableViewCellAccessoryDetailDisclosureButton;
        cell.editingAccessoryType = UITableViewCellAccessoryNone;

        return cell;
    }

    // Respond to accessory button taps
    -(void)tableView:(UITableView *)tableView
        accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath
    {
        // Do something here
    }

```

为了处理用户对详情展示按钮的点击，我们需要在 `tableView:accessoryButtonTappedForRowWithIndexPath:` 方法里面根据用户所点的行 (row) 来实现一些适当的处理代码。在真实的应用程序中，点击了这种按钮后，程序就会跳转到另一个视图，那个视图会详细解释当前所选的这个条目，并且会为用户提供一些额外的选项。

灰色的扩展指示器则要采用另一套办法来处理。由于这种控件并不是按钮，所以我们需要响应的应该是用户对单元格的选择，而不是用户对按钮的点击。开发者需要在 `tableView:didSelectRowAtIndexPath:` 方法中添加逻辑代码，把相关的扩展视图推入导航栈，另外也可以考虑用模态视图控制器或警告视图来展示。

这两种详情展示控件都不会改变单元格的其他行为。就算给单元格添加了此类控件，用户也依然能够对单元格执行选择或编辑等操作。它们只是增加了一种交互手段，并不会取代现有的机制。

## 9.8 解决方案：编辑表格

添加编辑功能之后，表格就会变得丰富起来。编辑功能可以令表格里的静态信息变成能够滚动的交互式控件，从而使得用户可以添加或删除数据。虽说处理编辑功能所需的代码有些复杂，但同样的技术却可以反复运用在各种应用程序中。我们只要掌握了“进入编辑状态”、“离开编辑状态”以及“实现撤销功能”等基础知识，就可以把它们移用到其他程序上面。

解决方案 9-3 制作的这张表格可以有效地响应用户的编辑操作。本例创建了由数张随机图像所构成的一份滚动列表。用户可以点击 Add 按钮来添加新的单元格，也可以通过 Swipe (滑动) 手势来移除单元格，另外，点击 Edit 按钮之后，即可进入编辑模式，此时用户可使用红色的小圆点来移除单元格 (如图 9-5 所示)。

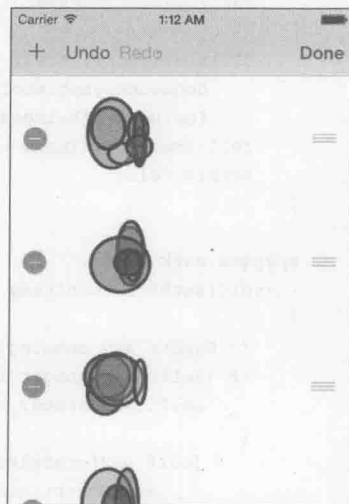


图 9-5 红色的圆形控件使得用户可以交互式地从表格中删除某些条目

在日常使用中，每位 iOS 用户都会迅速熟悉“通过小红点来移除表格单元格”这一操作方式。许多用户还会习惯“通过滑动手势来删除条目”这一功能。除此之外，本条解决方案又添加了一些控件，也就是每个单元格右侧那三条灰色的横线。用户可通过该控件把单元格拖曳到新的位置上。点击 Done 按钮，即可离开编辑模式。

### 解决方案 9-3 编辑表格

```
@implementation TestBedViewController
{
    NSMutableArray *items;
}

#pragma mark Data Source
// Number of sections
- (NSInteger)numberOfSectionsInTableView:(UITableView *)aTableView
{
    return 1;
}

// Rows per section
- (NSInteger)tableView:(UITableView *)aTableView
    numberOfRowsInSection:(NSInteger)section
{
    return items.count;
}

// Return a cell for the index path
- (UITableViewCell *)tableView:(UITableView *)aTableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [self.tableView
        dequeueReusableCellWithIdentifier:@"cell"
        forIndexPath:indexPath];
    cell.imageView.image = items[indexPath.row];
    return cell;
}

#pragma mark Edits
- (void)setBarButtonItems
{
    // Expire any ongoing operations
    if (self.undoManager.isUndoing ||
        self.undoManager.isRedoing)
    {
        [self performSelector:@selector(setBarButtonItems)
            withObject:nil afterDelay:0.1f];
        return;
    }
    UIBarButtonItem *undo = SYSBARBUTTON_TARGET(
```

```

        UIBarButtonItemSystemItemUndo, self.undoManager,
        @selector(undo));
undo.enabled = self.undoManager.canUndo;
UIBarButtonItem *redo = SYSBARBUTTON_TARGET(
    UIBarButtonItemSystemItemRedo, self.undoManager,
    @selector(redo));
redo.enabled = self.undoManager.canRedo;
UIBarButtonItem *add = SYSBARBUTTON(
    UIBarButtonItemSystemItemAdd, @selector(addItem:));

self.navigationItem.leftBarButtonItems = @[add, undo, redo];
}

- (void)setEditing:(BOOL)isEditing animated:(BOOL)animated
{
    [super setEditing:isEditing animated:animated];
    [self.tableView setEditing:isEditing animated:animated];

    NSIndexPath *path = [self.tableView indexPathForSelectedRow];
    if (path)
        [self.tableView deselectRowAtIndexPath:path animated:YES];

    [self setBarButtonItems];
}

- (void)updateItemAtIndexPath:(NSIndexPath *)indexPath
  withObject:(id)object
{
    // Prepare for undo
    id undoObject =
        object ? nil : items[indexPath.row];
    [[self.undoManager prepareWithInvocationTarget:self]
        updateItemAtIndexPath:indexPath withObject:undoObject];

    // You cannot insert a nil item. Passing nil is a delete request.
    [self.tableView beginUpdates];
    if (!object)
    {
        [items removeObjectAtIndex:indexPath.row];
        [self.tableView deleteRowsAtIndexPaths:@[indexPath]
            withRowAnimation:UITableViewRowAnimationTop];
    }
    else
    {
        [items insertObject:object atIndex:indexPath.row];
        [self.tableView insertRowsAtIndexPaths:@[indexPath]
            withRowAnimation:UITableViewRowAnimationTop];
    }
    [self.tableView endUpdates];
    [self performSelector:@selector(setBarButtonItems)
        withObject:nil afterDelay:0.1f];
}

```

```

    }

    - (void)addItem:(id)sender
    {
        // add a new item
        NSIndexPath *newPath =
            [NSIndexPath indexPathForRow:items.count inSection:0];
        UIImage *image = blockImage(IMAGE_SIZE);
        [self updateItemAtIndexPath:newPath withObject:image];
    }

    - (void)tableView:(UITableView *)tableView
        commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
        forRowAtIndexPath:(NSIndexPath *)indexPath
    {
        // delete item
        [self updateItemAtIndexPath:indexPath withObject:nil];
    }

    // Provide re-ordering support
    - (void)tableView:(UITableView *)tableView
        moveRowAtIndexPath:(NSIndexPath *)oldPath
        toIndexPath:(NSIndexPath *)newPath
    {
        if (oldPath.row == newPath.row) return;

        [[self.undoManager prepareWithInvocationTarget:self]
            tableView:self.tableView moveRowAtIndexPath:newPath
            toIndexPath:oldPath];

        id item = [items objectAtIndex:oldPath.row];
        [items removeObjectAtIndex:oldPath.row];
        [items insertObject:item atIndex:newPath.row];

        if (self.undoManager.isUndoing || self.undoManager.isRedoing)
        {
            [self.tableView beginUpdates];
            [self.tableView deleteRowsAtIndexPaths:@[oldPath]
                withRowAnimation:UITableViewRowAnimationLeft];
            [self.tableView insertRowsAtIndexPaths:@[newPath]
                withRowAnimation:UITableViewRowAnimationLeft];
            [self.tableView endUpdates];
        }
        [self performSelector:@selector(setBarButtonItems)
            withObject:nil afterDelay:0.1f];
    }

#pragma mark First Responder for undo support
    - (BOOL)canBecomeFirstResponder
    {
        return YES;
    }

```



```

    }

    - (void)viewDidAppear:(BOOL)animated
    {
        [super viewDidAppear:animated];
        [self becomeFirstResponder];
    }

    - (void)viewWillDisappear:(BOOL)animated
    {
        [super viewWillDisappear:animated];
        [self resignFirstResponder];
    }

#pragma mark View Setup
    - (void)viewDidLoad
    {
        [super viewDidLoad];
        self.view.backgroundColor = [UIColor whiteColor];

        [self.tableView registerClass:[UITableViewCell class]
         forCellReuseIdentifier:@"cell"];
        self.tableView.rowHeight = IMAGE_SIZE + 20.0f;
        self.tableView.separatorStyle =
            UITableViewCellStyleNone;
        self.navigationItem.rightBarButtonItem = self.editButtonItem;

        items = [NSMutableArray array];

        // Provide shake to undo support
        [UIApplication sharedApplication].applicationSupportsShakeToEdit = YES;
        [self setBarButtonItem];
    }
@end

```

### 9.8.1 添加撤销功能

Cocoa Touch 所提供的 `NSUndoManager` 类可用来反转用户的操作。在默认情况下，每个应用程序的视图都会提供一份共享的撤销管理器。开发者可以使用这份共享的管理器，也可以自己创建新的。

`UIResponder` 类的所有子类都能找到响应链中最近的那个撤销管理器。这就是说，如果在视图控制器中使用视图的 `NSUndoManager`，那么只需通过控制器的 `undoManager` 属性，就能找到那个 `NSUndoManager` 了。这是个很方便的特性，因为开发者只需给主视图控制器添加撤销功能，就能令其中的所有子视图自动具备该功能。

撤销管理器里面可以保存任意数量的撤销动作（undo action）。开发者可以指定栈的深度。栈越深，所用内存就越多。许多程序在内存比较紧张时都只支持 3、5 或 10 级撤销。栈中的每个动作既可以是由许多撤销操作所组成的复杂动作，又可以是像解决方案 9-3 所

演示的那种简单动作。

本条解决方案用撤销管理器来实现与“添加单元格”、“删除单元格”以及“移动单元格”这三种操作有关的撤销及重做功能。用户可通过 Undo 和 Redo 按钮在自己的编辑历程之中游走。本条解决方案会根据撤销管理器所能提供的动作来启用当前可供使用的按钮。

### 9.8.2 实现撤销功能

解决方案 9-3 采用同一个方法来添加及移除条目，这个方法就是 `updateItemAtIndexPath:withObject:`。此方法的运作方式为：如果传入的对象不是 `nil`，那么就把它插入到索引路径中；如果是 `nil`，则把位于该索引路径处的条目删除。

这样处理用户的请求似乎有点奇怪，因为我们需要编写一个方法，而且在方法里还要做一次判断，不过这样做其背后是有原因的。这种做法能够为撤销功能提供统一的基础代码，使得开发者可以更加方便地将其同撤销管理器相集成。

因此，该方法有两件事情要做。首先，它会准备好与撤销操作有关的行为。也就是说，它会告诉撤销管理器目前所运用的这项编辑操作将来应该如何还原。其次，它要执行实际的编辑操作，也就是修改 `items` 数组、更新表格，并更新导航栏中的按钮。

`setBarButtonItems` 方法要控制 Undo 与 Redo 按钮的状态。该方法会检查当前正在活动的撤销管理器，看看撤销栈中是否提供了能够撤销及能够重做的动作。如果有，就启用相应的按钮。

虽说笔者并不喜欢用晃动撤销（shake-to-undo）功能，但本条解决方案依然提供了这一功能。它在 `viewDidLoad` 方法里面把应用程序委托的 `applicationSupportsShakeToEdit` 属性设为 YES。另外也请大家注意，为了实现撤销功能，我们调用了与第一响应者有关的两个方法。当表格视图即将出现在屏幕上面时，令其变为第一响应者，而当它要从屏幕中消失时，则令其放弃第一响应者的身份。

### 9.8.3 显示移除单元格所用的控件

调用 `[self.tableView setEditing:YES animated:YES]` 方法，即可令表格把移除单元格所用的控件显示出来。这会更新表格的 `editing` 属性，并且会在每个单元格里显示出图 9-5 那样的红色移除控件。动画效果是可选的，不过笔者建议将它打开。有一条经验：如果要把程序从一个状态切换到另一个状态，那么应该在 iOS 界面中展示动画效果，使得用户意识到屏幕上的状态正在变化。

解决方案 9-3 采用了系统所提供的 Edit/Done 按钮（`self.editButtonItem`），并实现了 `setEditing:animated:` 方法，使得表格可以进入并离开编辑状态。用户点击 Edit 或 Done 按钮时（按钮文本会在这两者之间切换），该方法会更新表格的编辑状态以及导航栏上的按钮。

### 9.8.4 处理删除请求

删除表中的某一行时，表格会执行 `tableView:commitEditingStyle:forRowAtIndexPath:`

`IndexPath`: 回调, 以便将这一操作告知应用程序。从表格上移除某个条目只是令其不显示出来, 这并不会修改底层的数据。除非开发者把这个条目也从数据源里移除, 否则下次刷新表格的时候, 这个已删除的条目还是会显示出来。我们可以在该方法里协调表格与数据源之间的关系, 并对“用户想要删除这一行”的请求做出响应。

在本例中, 我们可以从给数据源方法提供内容的数据结构里 (也就是包含各图像的那个 `NSMutableArray`) 删除一项条目, 而在真实的应用程序中, 则可以执行诸如删除文件或移除联系人等操作, 以响应用户的编辑请求。

解决方案 9-3 会用动画效果来展示单元格的删除过程。我们可以把添加某行及删除某行等操作放在 `beginUpdates` 方法之后, 并放在配套的 `endUpdates` 方法之前, 从而令系统能够以动画效果来同时展示这些操作。

### 9.8.5 通过滑动手势删除单元格

要想从 `UITableView` 实例中移除条目, `Swipe` 手势是个很简洁的办法。只需实现 `tableView:commitEditingStyle:forRowAtIndexPath:` 方法, 即可启用 `Swipe` 功能。表格会处理好剩下的事情。

用户将某个单元格从右方迅速向左方拖曳, 这就是针对单元格的 `Swipe` 手势。单元格右侧会显示出长方形的 `Delete` 按钮, 用以确认此操作, 不过左侧并不会出现表示移除单元格功能的那个红色圆形控件。

当用户执行 `Swipe` 操作并确认之后, 我们就可以在 `tableView:commitEditingStyle:forRowAtIndexPath:` 方法里更新数据了, 这与在编辑模式下删除单元格是一样的。

### 9.8.6 调整单元格的顺序

如果能直接调整表格中各单元格的顺序, 用户就可以获得更大的自由度了。图 9-5 所演示的这张表格里有很多调整单元格顺序所用的控件, 这种控件是三条上下叠放的灰色线段。用户可以通过重排单元格顺序来调整各项待办事务的优先级, 或是在播放清单中选择首先要听的歌曲等。iOS 的表格内置了重排单元格顺序这一功能, 开发者很容易就能将其集成到应用程序中。

与通过滑动手势删除单元格一样, 单元格重排功能是否启用也取决于开发者是否实现了某个方法。这个方法就是 `tableView:moveRowAtIndexPath:toIndexPath`, 它可以将表格的数据源与屏幕上所发生的变化相同步, 这与删除单元格时回调的 `tableView:commitEditingStyle:forRowAtIndexPath:` 方法类似。添加这一方法即可调整单元格顺序。

### 9.8.7 添加单元格

解决方案 9-3 用 `Add` 按钮来为表格添加新内容。该按钮是 iOS 系统内置的 `UIBarButtonItem`, 它的样子是个加号。(参见图 9-5 左上角。) 解决方案 9-3 中的 `addItem:` 方法会向 `items` 数组末端追加一张新的随机图像。

## 9.9 解决方案：操控表格的区段

许多 iOS 应用程序使用区段 (section) 来划分各行。区段是在列表中划分小区域的方式，它把条目按照逻辑组织成单元。最常用的分区方式是首字母分区法，当然开发者并不一定非要这么组织数据。你可以选择与应用程序相匹配的任意分区方式。

图 9-6 演示的这张表格会把各种颜色名称分组显示在不同的区段里。每个区段都会显示各自的头部标题 (也就是 “Crayon names starting with ...” 形式的一段文本)，表格右侧的索引可以快速跳转到各个区段。请注意，索引里面没有列出与 K、Q、X 及 Z 相对应的区段，因为那些区段是空的。通常我们都想把索引中的空区段隐藏起来。



图 9-6 分区后的表格会显示出区段的头部标题，也会提供索引，使得用户可以迅速跳转到各个区段

### 9.9.1 构建区段

构建分组和分区时，要把它们理解成二维数组。开发者可以创建名叫 `sectionArray` 的二维数组，来存储并访问数据结构中各个分区里的数据。我们需要创建含有数组的数组。`sectionArray` 里可以存储与每个分区相对应的小数组，而那个小数组里面又会包含该分区每个单元格的标题。

通过谓词，我们可以用一系列字符串来构建区段。借助下面这个方法，开发者可以把普通数组中以某个字母开头的那些字符串都划分到一起。`beginswith` 这个谓词能够匹配以给定字母起头的字符串：

```
- (NSArray *)itemsInSection:(NSInteger)section
{
    NSPredicate *predicate = [NSPredicate predicateWithFormat:
        @"SELF beginswith[cd] %@", [self firstLetter:section]];
    return [crayonColors.allKeys
        filteredArrayUsingPredicate:predicate];
}
```

反复调用上述方法，并把每次返回的结果数组都添加到一个可变的数组里面，这样就能根据一份普通的列表创建出二维的 `sectionArray` 数组：

```
sectionArray = [NSMutableArray array];
for (int i = 0; i < 26; i++)
    [sectionArray addObject:[self itemsInSection:i]];
```

这种实现方式要想正确运作，必须满足两个条件。首先，各单词必须排好顺序（每个词在区段里出现的先后顺序与其在原数组里出现的先后顺序相同），其次，区段和该区段中的单词必须相符。上面这段循环代码无法处理以符号或数组开头的词。然而我们只需添加 `other`（其他）区段，即可将这些单词归为一组，不过由于本条解决方案比较简单，所以笔者就把对

这些单词的处理省略掉了。

刚才说过，按照首字母来分组是最常用的一种分组方式，不过，你也可以采用其他分组方式。比方说，可以按照部门来划分人员，按照等级来划分宝石或按照日期来划分约会。无论选用何种分组方式，以二维数组来充当这种分区表格的数据源都是非常合适的。

对于这个简单的范例来说，可以使用二维数组这种结构来添加或删除条目。不过读者很容易就能发现，用这种方式添加数据虽然简单，但维护起来却有点麻烦，而这正是 Core Data 能够派上用场的地方。使用了 Core Data 之后，我们就不用再处理多层数组了，而是可以在数据库中查询与任意的对象字段有关的信息，并且按照需要对其排序。第 12 章会介绍如何将 Core Data 与表格结合起来使用。等阅读了那一章你就会发现，Core Data 极大地简化了编程工作。目前这个范例依然使用简单的二维数组来介绍区段及其用法。

### 9.9.2 区段数量与区段内的行数

若想创建分区的表格，需要定制下面这两个关键的数据源方法：

- **numberOfSectionsInTableView:**——该方法返回表格中将要出现的区段数量，也就是表格要把其中的条目分成多少个组来显示。如果使用笔者所推荐的这种 `sectionArray`，那么就应该返回该数组里的元素个数，也就是 `sectionArray.count`。如果数组中的元素个数可以提前确定（例如在本例中，尽管某些区段里没有条目，但我们可以确定元素个数是 26），那么可将其用“硬数值”的形式写在代码里面，不过，代码还是尽量写得通用一些比较好。
- **tableView:numberOfRowsInSection:**——系统会以区段编号为参数来调用该方法。开发者需要指定出现在该区段中的行数。如果采用笔者推荐的 `sectionArray`，那么就应该返回第  $n$  个小数组里的元素个数：

```
sectionArray[sectionNumber].count
```

### 9.9.3 返回单元格

分区的表格需要通过行和分区这两种信息来确定单元格数据。本章早前那条解决方案采用普通的一维数组，以下标来表示行号。带分区的表格必须用完整的索引路径来确定区段及行的序号，以此找到填充单元格所用的数据。下面这个方法写在名为 `CrayonHandler` 的辅助类中，它首先根据区段取出对应的 `currentItems` 数组，然后再根据行来寻找具体的条目。如果用二维数组来表示分区表格的数据源，那么就可以使用解决方案 9-4 中的这个 `CrayonHandler` 辅助类了，下面详细列出该类 `colorNameAtIndexPath:` 方法的代码：

```
// Color name by index path
- (NSString *)colorNameAtIndexPath:(NSIndexPath *)path
{
    if (path.section >= sectionArray.count)
        return nil;
    NSArray *currentItems = sectionArray[path.section];

    if (path.row >= currentItems.count)
```

```

        return nil;
        NSString *crayon = currentItems[path.row];

        return crayon;
    }

```

还有个相似的方法用于获取单元格的颜色：

```

// Color by index path
- (UIColor *)colorAtIndexPath:(NSIndexPath *)path
{
    NSString *crayon = [self colorNameAtIndexPath:path];
    if (crayon)
        return crayonColors[crayon];
    return nil;
}

```

下面这个数据源方法通过调用上述两方法来返回具备适当颜色和适当名称的单元格：

```

// Return a cell for the index path
- (UITableViewCell *)tableView:(UITableView *)aTableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
        [self.tableView dequeueReusableCellWithIdentifier:@"cell"
         forIndexPath:indexPath];

    // Retrieve the crayon name
    NSString *crayonName = [crayons colorNameAtIndexPath:indexPath];

    // Update the cell
    cell.textLabel.text = crayonName;

    // Tint the title
    if ([crayonName hasPrefix:@"White"])
        cell.textLabel.textColor = [UIColor blackColor];
    else
        cell.textLabel.textColor = [crayons colorAtIndexPath:indexPath];

    return cell;
}

```

#### 解决方案 9-4 实现带有若干区段的表格

```

/* CrayonHandler.m */
// Return an array of items that appear in each section
- (NSArray *)itemsInSection:(NSInteger)section
{
    NSPredicate *predicate = [NSPredicate predicateWithFormat:
        @"SELF beginswith[cd] %@", [self firstLetter:section]];
    return [[crayonColors allKeys] filteredArrayUsingPredicate:predicate];
}

```

```

// Count of available sections
- (NSInteger)numberOfSections
{
    return sectionArray.count;
}

// Number of items within a section
- (NSInteger)countInSection:(NSInteger)section
{
    return [sectionArray[section] count];
}

// Return the letter that starts each section member's text
- (NSString *)firstLetter:(NSInteger)section
{
    return [[ALPHA substringFromIndex:section] substringToIndex:1];
}

// The one-letter section name
- (NSString *)nameForSection:(NSInteger)section
{
    if (![self countInSection:section])
        return nil;
    return [self firstLetter:section];
}

// Color name by index path
- (NSString *)colorNameAtIndexPath:(NSIndexPath *)path
{
    if (path.section >= sectionArray.count)
        return nil;
    NSArray *currentItems = sectionArray[path.section];

    if (path.row >= currentItems.count)
        return nil;
    NSString *crayon = currentItems[path.row];

    return crayon;
}

// Color by index path
- (UIColor *)colorAtIndexPath:(NSIndexPath *)path
{
    NSString *crayon = [self colorNameAtIndexPath:path];
    return crayonColors[crayon];
}

```

#### 9.9.4 创建每个区段的头部标题

若要给分区表格中的每个区段头部添加标题，则需稍微多写一些代码。开发者可以通过名为 `tableView:titleForHeaderInSection:` 的可选方法给每个区段提供标题。系统

会给该方法传入一个整数，而开发者则应该提供对应的标题。如果给定的区段里没有任何条目，或是整张表个只有一个区段，那么就返回 nil：

```
// Return the header title for a section
- (NSString *)tableView:(UITableView *)aTableView
  titleForHeaderInSection:(NSInteger)section
{
    NSString *sectionName = [crayons nameForSection:section];
    if (!sectionName) return nil;
    return [NSString stringWithFormat:
        @"Crayon names starting with '%@'", sectionName];
}
```

假如不想使用标题，那么可以返回自定义的头部视图。

## 9.9.5 定制表格与区段的头部及尾部

分区的表格视图特别容易定制其内容。开发者可以把任意类型的视图赋给 tableView:headerView 属性以及与其相关的 tableView:footerView 属性，每个视图都可以有自己的子视图。于是，我们可以添加标签、文本框、按钮以及其他控件，以此来丰富表格的特性。

每张表格并不是只能有一个头部（header）和尾部（footer）。每个区段都提供了可以定义的头部视图及尾部视图。你可以改变区段头部的高度，或将其中的元件改成自己定制的视图。可选的 tableView:heightForHeaderInSection: 方法（或 sectionHeaderHeight 属性）及 tableView:viewForHeaderInSection: 方法使得开发者能够单独指定每个分区的头部。对于分区的尾部来说，也有对应的一套方法。

## 9.9.6 创建区段索引

实现了 sectionIndexTitlesForTableView: 方法的表格可以展示出图 9-6 右侧那样的索引视图。系统创建表格视图的时候会调用这个方法，而开发者所返回的数组则决定了显示在屏幕上面的索引条目。返回 nil 就表示不需要显示索引。苹果公司建议只给普通样式的表格视图添加区段索引，也就是说，只应该给使用 UITableViewStylePlain 样式的表格添加索引，而不应该给 UITableViewStyleGrouped 样式的表格添加：

```
// Return an array of section titles for index
- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)aTableView
{
    NSMutableArray *indices = [NSMutableArray array];
    for (int i = 0; i < crayons.numberOfSections; i++)
    {
        NSString *name = [crayons nameForSection:i];
        if (name) [indices addObject:name];
    }
    return indices;
}
```

虽说本例采用单个字母作为分区索引的标题，但你并不一定非要受此限制。开发者也可



以把单词或是用 Unicode 编码来表示的符号当成标题，例如，可以在索引里使用 emoji 表情符号，它们也是 iOS 字符库的一部分。下面这行代码可以给索引里添加一张黄颜色的小笑脸符号：

```
[indices addObject:@"\ue057"];
```

### 9.9.7 处理索引与区段不匹配的问题

用户点击表格的索引时，表格会根据用户触摸点的偏移量来滚动。正如本节早前所述，这张表格的索引里并没有显示出 K、Q、X 及 Z 区段。由于缺了这几个字母，所以用户在索引中点选的字母会与表格实际显示出来的区段不相符。

为了解决此问题，我们需要实现可选的 `tableView:sectionForSectionIndexTitle:` 方法。该方法负责把区段索引的标题（也就是由 `sectionIndexTitlesForTableView:` 方法所返回的字符串）和区段编号对应起来。这样就能把原有的不匹配情况覆盖掉，并且会在用户所选的索引字母区段和表格要显示的区段之间建立起准确的一一对应关系：

```
#define ALPHA @"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
- (NSInteger)tableView:(UITableView *)tableView
    sectionForSectionIndexTitle:(NSString *)title
    atIndex:(NSInteger)index
{
    return [ALPHA rangeOfString:title].location;
}
```

### 9.9.8 为分区表格实现委托方法

与数据源方法一样，为分区表格实现 `UITableViewDelegate` 协议中的 `tableView:didSelectRowAtIndexPath:` 方法时，也需要使用索引路径中的 `section` 和 `row` 属性。本例中，我们可以通过这两个属性在 `sectionArray` 数组里找到与当前区段对应的小数组，并在小数组里找到用户所点选的条目：

```
// On selecting a row, update the navigation bar tint
- (void)tableView:(UITableView *)aTableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    UIColor *color = [crayons colorAtIndex:indexPath.index];
    self.navigationController.navigationBar.barTintColor = color;
}
```

## 9.10 解决方案：在表格中搜索

搜索显示控制器可以实现由用户所主导的搜索操作。这些控制器使得用户能够实时地过滤表格内容，而表格则会根据用户所输入的查询信息立即做出响应。这项功能很好，它使得用户可以交互式地找寻自己想要的内容，每次在搜索框中输入新字符时，搜索结果都会立刻更新。

创建这种控制器的时候，需要用搜索栏实例和内容控制器来初始化它，这个内容控制器

通常是表格视图，用户所要查询的数据就存放在表格的数据源之中。解决方案 9-5 演示了在应用程序中创建并使用搜索显示控制器时所需的步骤。

搜索功能最好通过谓词来构建，这样的话，只需调用一个简单的方法，即可在数组上面进行过滤，并获取到与待搜索内容相匹配的条目。下面我们来演示如何在普通的字符串数组里获取与搜索栏中的文本相匹配的字符串。contains 后面的 [cd] 表示匹配的时候既不区分大小写，也不区分附加符号（diacritic）。附加符号是与字母相伴的小标记，比方说表示变音的那两个点（umlaut，¨），或是西班牙语字母 n 上方的波浪线（tilde，~）。

```
NSPredicate *predicate =
    [NSPredicate predicateWithFormat:@"SELF contains[cd] %@",
        searchBar.text];
filteredArray = [[crayonColors allKeys]
    filteredArrayUsingPredicate:predicate];
```

本例中的搜索栏应该作为头部视图（header view）出现在表格顶端，如图 9-7 左侧截图所示。在下面这段代码中，我们还需要用这个搜索栏来配置搜索显示控制器：

```
self.tableView.tableHeaderView = searchBar;
searchController = [[UISearchDisplayController alloc]
    initWithSearchBar:searchBar contentsController:self];
```



图 9-7 开始搜索时，必须滚动到表格顶部。作为头部视图的搜索栏会出现在表格中的首个位置上（如左侧截图所示）。用户点击搜索栏之后，就会将其激活，此时搜索栏会跳到导航栏里，而表格则会根据用户输入的搜寻标准来展示与之匹配的条目（如右侧截图所示）

iOS 7 的搜索栏可以通过 searchBarStyle 属性来配置其样貌，该属性的取值可以是 UISearchBarStyleProminent、UISearchBarStyleMinimal 或 UISearchBarStyleDefault。UISearchBarStyleProminent 就是默认的样式（UISearchBarStyleDefault），这种样式的背景是半透明的，搜索框是完全不透明的，这与早前 iOS 版本中的风格相符。UISearchBarStyleMinimal 风格没有背景，它的搜索框是半透明的。当用户

点击搜索框时，画面会发生变化，搜索栏会上移至导航栏所在的区域，如图 9-7 右侧截图所示。此时将出现用于显示搜索结果的表格视图，它会临时替换掉原来的表格。搜索栏和显示搜索结果的表格视图会一直留在屏幕上，直到用户点击 Cancel 按钮为止，点击该按钮后，用户会重新看到未加过滤的表格。

### 9.10.1 创建搜索显示控制器

搜索显示控制器可以把另外一个视图控制器所拥有的数据显示出来（在本例中，该控制器是个标准的 `UITableViewController`）。搜索显示控制器通常会用谓词来过滤数据源，并把这份数据的某个子集显示到自己的表格视图中。开发者需要用搜索栏和内容控制器来初始化搜索显示控制器。

我们可以像平常那样设置搜索栏中的文本特性，但是不要设置委托。因为搜索栏会自行与搜索显示控制器相配合，而无须开发者手工指定其委托。

配置搜索显示控制器的时候，请像下面这段代码一样设置好 `searchResultsDataSource` 及 `searchResultsDelegate` 属性。这两者通常指向充当主表格视图控制器的那个 `UITableViewController` 子类，我们需要修改类中现有的数据源方法及委托方法，以便支持搜索功能：

```
// Create a search bar
searchBar = [[UISearchBar alloc]
    initWithFrame:CGRectMake(0.0f, 0.0f, width, 44.0f)];
searchBar.autocorrectionType = UITextAutocorrectionTypeNo;
searchBar.autocapitalizationType = UITextAutocapitalizationTypeNone;
searchBar.keyboardType = UIKeyboardTypeAlphabet;
self.tableView.tableHeaderView = searchBar;

// Create the search display controller
searchController = [[UISearchDisplayController alloc]
    initWithSearchBar:searchBar contentsController:self];
searchController.searchResultsDataSource = self;
searchController.searchResultsDelegate = self;
```

### 9.10.2 为搜索显示控制器注册单元格

程序里面每张表格视图所用的单元格类型都应该注册，这样系统才能把它们从队列中正确地取出来。搜索显示控制器里内置的表格所用的单元格自然也需注册。如果不执行这一步就直接从 `self.tableView` 的队列里获取单元格，那么程序就会崩溃。下面这段代码能够为这两张表格注册各自所用到的单元格类：

```
// Register cell classes
[self.tableView registerClass:[UITableViewCell class]
    forCellReuseIdentifier:@"cell"];
[searchController.searchResultsTableView registerClass:[UITableViewCell class]
    forCellReuseIdentifier:@"cell"];
```

iOS 7 中有个新问题导致我们必须改变注册单元格类型的时机。在 `tableView:`

`cellForRowAtIndexPath:` 这个数据源方法中获取单元格的时候, 系统所提供的搜索表格是新建出来的, 于是原来的注册信息就丢失了。要想解决这个问题, 最简单的办法是在该方法的开头注册单元格类。这虽然看上去不够高效, 但在苹果公司还没推出更为优雅的方法之前, 却能保证对单元格类型的注册操作总是有效的。

阅读本条解决方案的代码时, 你将看到在 `tableView:cellForRowAtIndexPath:` 方法里, 笔者会采用判断语句来帮助程序把普通的表格和显示搜索结果所用的表格区分开, 使该方法能够给系统提供正确的单元格。

### 9.10.3 构建支持搜索功能的数据源方法

在用户搜索表格的过程中, 表格里所显示的条目数量也会变化。如果输入的待搜索字符串比较短, 那么通常会匹配到许多条目, 若是比较长, 则匹配出来的条目就会少一些。我们需要向系统报告每张表格当前的行数。用户在搜索框中输入文本时, 显示出来的行数也要随之变化。我们需要检测传进来的 `aTableView` 参数, 以判断当前回调该方法的控制器到底是表格视图控制器还是搜索显示控制器, 然后根据情况调整相应的行数:

```
- (NSInteger)tableView:(UITableView *)aTableView
    numberOfRowsInSection:(NSInteger)section
{
    if (aTableView == searchController.searchResultsTableView)
        return [crayons filterWithString:searchBar.text];
    return [crayons countInSection:section];
}
```

我们通过谓词来报告表格中有多少条目能和搜索框里的文本相匹配。谓词是种极其简单的过滤方式, 能够找出数组中与某个待搜索字符串相匹配的那些元素。本例所用的谓词会以不区分大小写的方式来执行 `contains` 匹配。只要搜索框中的文本包含在某个字符串之中, 我们就把该字符串留在过滤后的数组里。此外, 也可以改用 `beginswith` 来进行匹配, 这样做将会把不以待搜索文本开头的字符串排除在搜索结果之外。下面这个方法会过滤数组, 保存搜索结果, 并返回搜索结果之中的条目数量:

```
- (NSInteger)filterWithString:(NSString *)filter
{
    NSPredicate *predicate = [NSPredicate predicateWithFormat:
        @"SELF contains[cd] %@", filter];
    filteredArray = [[crayonColors allKeys]
        filteredArrayUsingPredicate:predicate];
    return filteredArray.count;
}
```

提供单元格的时候, 一定要注意索取单元格的到底是哪张表格视图。向每张表格所注册的单元格类型必须正确无误; 而从队列里取出单元格并对其初始化时, 也必须注意该单元格究竟属于哪张表格。下面这个方法会根据情况, 用标准的数据集或过滤后的数据集来配置单元格, 并将其返回给调用者:

```

- (UITableViewCell *)tableView:(UITableView *)aTableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    [aTableView registerClass:[UITableViewCell class]
      forCellReuseIdentifier:@"cell"];
    UITableViewCell *cell =
        [aTableView dequeueReusableCellWithIdentifier:@"cell"
          forIndexPath:indexPath];

    NSString *crayonName;
    if (aTableView == self.tableView)
    {
        crayonName = [crayons colorNameAtIndexPath:indexPath];
    }
    else
    {
        if (indexPath.row < crayons.filteredArray.count)
            crayonName = crayons.filteredArray[indexPath.row];
    }

    cell.textLabel.text = crayonName;
    cell.textLabel.textColor = [crayons colorNamed:crayonName];
    if ([crayonName hasPrefix:@"White"])
        cell.textLabel.textColor = [UIColor blackColor];

    return cell;
}

```

#### 9.10.4 委托方法

并不是只有数据源方法才需要处理搜索问题。当用户点击表格的时候，委托方法也必须依照当前的情境做出适当的响应。与前面所讲的数据源方法一样，委托方法同样需要判断回调时所传入的 `aTableView` 参数，并以此来确定当前处于活动状态的是哪张表格视图。该方法会根据用户所点击的表格及索引路径选取对应的颜色，并以此来充当搜索栏和导航栏的 `barTintColor`：

```

// Respond to user selections by updating tint colors
- (void)tableView:(UITableView *)aTableView
  didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    UIColor *color = nil;
    if (aTableView == self.tableView)
        color = [crayons colorAtIndexPath:indexPath];
    else
    {
        if (indexPath.row < crayons.filteredArray.count)
        {
            NSString *colorName =
                crayons.filteredArray[indexPath.row];

```

```

        if (colorName)
            color = [crayons colorNamed:colorName];
    }
}

self.navigationController.navigationBar.barTintColor = color;
searchBar.barTintColor = color;
}

```

### 9.10.5 使用与搜索功能相配套的索引

解决方案 9-5 演示了如何用其他一些方式为分区的表格添加搜索功能。如果要支持搜索，那么添加到区段索引中的首个条目就应该是 `UITableViewIndexSearch` 常量。该常量只应该用在表格的索引中，而且只应该用作索引的首个条目。它会给索引里添加一个小小的放大镜图标，告诉用户可以搜索这张表格。我们需要编写代码，使得用户能够通过点击该图标来快速跳转到列表顶部。修改 `tableView:sectionForSectionIndexTitle:atIndex:` 方法，以捕获用户的请求。在用户点击放大镜图标的时候，我们会在该方法里手动调用 `scrollRectToVisible:animated:` 方法，以便把搜索栏显示到屏幕范围内。如果不这样做，那么用户就必须从 0 号区段向上滚动才能看到搜索栏，0 号区段里的单元格都以字母 A 开头。

解决方案 9-5 使用搜索功能

---

```

// Add Search to the index
- (NSArray *)sectionIndexTitlesForTableView:
    (UITableView *)aTableView
{
    if (aTableView == searchController.searchResultsTableView)
        return nil;

    // Initialize with the search magnifying glass
    NSMutableArray *indices = [NSMutableArray
        arrayWithObject:UITableViewIndexSearch];

    for (int i = 0; i < crayons.numberOfSections; i++)
    {
        NSString *name = [crayons nameForSection:i];
        if (name) [indices addObject:name];
    }

    return indices;
}

// Handle both the search index item and normal sections
- (NSInteger)tableView:(UITableView *)tableView
    sectionForSectionIndexTitle:(NSString *)title
    atIndex:(NSInteger)index
{

```

```

if (title == UITableViewIndexSearch)
{
    [self.tableView scrollRectToVisible:searchBar.frame
        animated:NO];
    return -1;
}
return [ALPHA rangeOfString:title].location;
}
// Handle the Cancel button by resetting the search text
- (void)searchBarCancelButtonClicked:(UISearchBar *)aSearchBar
{
    [searchBar setText:@""];
}

// Titles only for the main table
- (NSString *)tableView:(UITableView *)aTableView
    titleForHeaderInSection:(NSInteger)section
{
    if (aTableView == searchController.searchResultsTableView)
        return nil;
    return [crayons nameForSection:section];
}

// Upon appearing, scroll away the search bar
- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    NSIndexPath *path =
        [NSIndexPath indexPathForRow:0 inSection:0];
    [self.tableView scrollToRowAtIndexPath:path
        atScrollPosition:UITableViewScrollPositionTop
        animated:NO];
}

```

我们在 `viewWillAppear:` 方法里面调用 `scrollToRowAtIndexPath`，使得视图刚刚加载进来的时候搜索栏会位于屏幕范围之外。这样的话，一开始就不会看到表格的搜索栏了，用户可以向下滚动表格，把搜索栏显示出来，也可以跳转到自己想看的其他地方。

最后，如果用户要取消搜索操作，我们就提前把搜索栏里的文本清空。

## 9.11 解决方案：给表格添加下拉刷新功能

下拉刷新是个使用面很广的程序特性，在过去几年的 App Store 里非常流行。用户可以向下拉拽表格的顶端，从而请求程序刷新表格内容。由于这种操作方式相当直观，所以很多人都在考虑苹果公司为什么还不把它添加到 `UITableViewController` 类里。在 iOS 6 中，苹果公司创建了一种极具特色的刷新控件，它可以拉伸，而且具备动画效果。到了 iOS 7，这个可拉伸的控件换成了更为传统的活动指示器（activity indicator），而且指示器稍稍多了一

些动画效果，以便把控件的状态反馈给用户（参见图 9-8）。

新的 `UIRefreshControl` 类是一种非常方便的控件，可以用它来表示表格的刷新功能。解决方案 9-6 演示了怎样将其添加到应用程序里。开发者只需新建 `UIRefreshControl` 实例，并将其赋给表格视图控制器的 `refreshControl` 属性，即可使控件直接出现在表格视图中，而无须再执行其他操作。当用户向下滑拽表格视图的时候，就能显示并触发 `UIRefreshControl` 控件了。

若想以编程的方式激活刷新控件，则需要用 `beginRefreshing` 来触发刷新事件，此时控件会变成带有动画效果的进度转轮。准备好新数据之后，用 `endRefreshing` 结束刷新过程，并重新加载表格视图。

由于该控件是从 `UIControl` 类继承下来的，所以该控件的实例在激活的时候也采用目标-动作机制来向客户端发送定制的选择子。由于某些原因，在程序应该执行刷新操作时，此控件触发的是 `UIControlEventValueChanged` 事件。其实对于这种无状态（stateless）的控件来说，苹果公司早就应该引入一种 `UIControlEventTriggered` 事件了。

采用下拉刷新控件可以使程序暂缓执行某些开销比较大的任务。比方说，我们可以暂时不从互联网上获取新的信息，或是暂时不重新计算表格中的元素，直到用户明确触发这些请求的时候我们再去做。通过下拉刷新控件，用户可以控制刷新的时机，而程序也可以在按照客户的需求获取信息和减少计算开销之间取得平衡。

解决方案 9-6 中的 `DataManager` 类会通过操作队列（operation queue）以异步的方式加载数据：

```
- (void)loadData
{
    NSString *rss = @"http://itunes.apple.com/us/rss/topalbums/limit=30/xml";
    NSOperationQueue *queue = [[NSOperationQueue alloc] init];
    [queue addOperationWithBlock:
    ^{
        root = [[XMLParser sharedInstance] parseXMLFromURL:
        [NSURL URLWithString:rss]];
        [[NSOperationQueue currentQueue] addOperationWithBlock:^(
        [self handleData];
        )];
    }];
}
```

这样做可以避免数据加载操作阻塞到主线程。刷新控件的进度转轮依然会旋转，而用户也可以继续操作程序里的其他 UI 元件。执行完数据获取操作之后，我们把控制权还给主线程：



图 9-8 开发者很容易就能给表格添加下拉刷新功能。用户通过向下滑拽表格来请求程序更新其中的数据



```

if (delegate &&
    [delegate respondsToSelector:@selector(dataIsReady:)])
    [delegate performSelectorOnMainThread:@selector(dataIsReady:)
        withObject:self waitUntilDone:NO];

```

解决方案 9-6 除了有刷新控件之外，还提供了 Load 按钮。大部分程序都不会同时提供这两种刷新手段，不过解决方案 9-6 想通过这个按钮来演示它如何与刷新控件之间互动。当用户点击 Load 按钮之后，开发者仍然需要执行 UIRefreshControl 的 startRefreshing 及 endRefreshing 方法。这样做可以确保 UIRefreshControl 与通过点击 Load 按钮而触发的手工刷新操作之间能够同步。

#### 解决方案 9-6 给表格添加下拉刷新功能

```

- (void)dataIsReady:(id)sender
{
    // Update the title
    self.title = @"iTunes Top Albums";

    // Reenable the bar button item
    self.navigationItem.rightBarButtonItem.enabled = YES;

    // Stop refresh control animation and update the table
    [self.refreshControl endRefreshing];
    [self.tableView reloadData];
}

- (void)loadData
{
    // Provide user status update
    self.title = @"Loading...";

    // Disable the bar button item
    self.navigationItem.rightBarButtonItem.enabled = NO;

    // Start refreshing
    [self.refreshControl beginRefreshing];

    [manager loadData];
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.tableView.rowHeight = 72.0f;
    [self.tableView registerClass:[UITableViewCell class]
        forCellReuseIdentifier:@"generic"];

    // Offer a bar button item and...
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Load", @selector(loadData));
}

```

```

// Alternatively, use the refresh control
self.refreshControl = [[UIRefreshControl alloc] init];
[self.refreshControl addTarget:self action:@selector(loadData)
    forControlEvents:UIControlEventValueChanged];

// This custom data manager asynchronously (nonblocking) loads
// data in a secondary thread
manager = [[DataManager alloc] init];
manager.delegate = self;
}

```

## 9.12 解决方案：添加指令行

如果某个单元格与指令行（action row，也叫作 drawer cell（抽屉式单元格））相关联，那么当用户点击该单元格时，将会弹出与之有关的一些附加操作。在 Tweetbot（<http://tapbots.com>）等商业程序中可能会看到这种功能。解决方案 9-7 所构建的指令行里面有两个“抽屉”，每个“抽屉”里面各有一枚按钮（参见图 9-9）。用户点击 Title 按钮之后，程序会把单元格的文本设为导航栏的标题，如果用户点击 Alert 按钮，那么屏幕上就会弹出警告界面，界面里的字符串与单元格的文本相同。iOS 开发者 Bilal Sayed Ahmad（Twitter 号码是 @Demonic\_BLITZ）建议笔者把这个功能添加到本书之中，这条解决方案的代码借鉴了由他所创建的范例项目。

解决方案 9-7 会把一个隐藏的单元格添加到表格视图中，其他的单元格都会根据该单元格是否展示出来而做出调整。首先要调整的是汇报每个区段内单元格数量的方法，指令行的索引路径存放在 `actionRowPath` 中。如果我们已经把这个隐藏的单元格显示出来了，那么表格中的单元格数量就会比原来多 1。如果它是隐藏着的，那么数据源方法就返回平常的单元格数量。

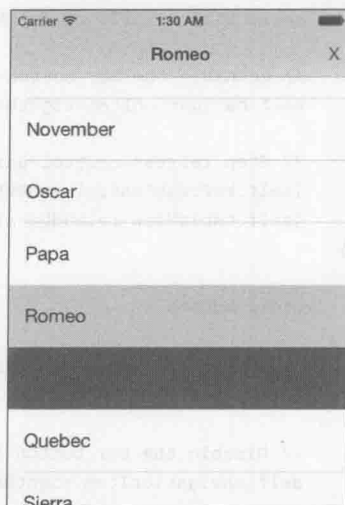


图 9-9 指令行里面提供了一些与单元格相关的指令，当用户点击单元格时，指令行就会弹出来。在本例中，用户点击了 Romeo 按钮，于是两个隐藏的抽屉式按钮就会弹出来，它们是 Title 按钮和 Alert 按钮

### 解决方案 9-7 向表格中添加指令行

```

// Rows per section
- (NSInteger)tableView:(UITableView *)aTableView
    numberOfRowsInSection:(NSInteger)section
{
    return items.count + (self.actionRowPath != nil);
}

```

```

// Return a cell for the index path
- (UITableViewCell *)tableView:(UITableView *)aTableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    if ([self.actionRowPath isEqual:indexPath])
    {
        // Action Row
        CustomCell *cell = (CustomCell *)[self.tableView
            dequeueReusableCellWithIdentifier:@"action"
            forIndexPath:indexPath];
        [cell setActionTarget:self];
        return cell;
    }
    else
    {
        // Normal cell
        UITableViewCell *cell = [self.tableView
            dequeueReusableCellWithIdentifier:@"cell"
            forIndexPath:indexPath];
        // Adjust item lookup around action row if needed
        NSInteger adjustedRow = indexPath.row;
        if (_actionRowPath &&
            (_actionRowPath.row < indexPath.row))
            adjustedRow--;
        cell.textLabel.text = items[adjustedRow];

        cell.textLabel.textColor = [UIColor blackColor];
        cell.selectionStyle = UITableViewCellSelectionStyleGray;
        return cell;
    }
}

- (NSIndexPath *)tableView:(UITableView *)tableView
  willSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Only select normal cells
    if([indexPath isEqual:self.actionRowPath]) return nil;
    return indexPath;
}

// Deselect any current selection
- (void)deselect
{
    NSArray *paths = [self.tableView indexPathsForSelectedRows];
    if (!paths.count) return;

    NSIndexPath *path = paths[0];
    [self.tableView deselectRowAtIndexPath:path animated:YES];
}

// On selection, update the title and enable find/deselect

```

```

- (void)tableView:(UITableView *)aTableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSArray *pathsToAdd;
    NSArray *pathsToDelete;

    if ([self.actionRowPath.previous isEqual:indexPath])
    {
        // Hide action cell
        pathsToDelete = @[self.actionRowPath];
        self.actionRowPath = nil;
        [self deselect];
    }
    else if (self.actionRowPath)
    {
        // Move action cell
        BOOL before = [indexPath before:self.actionRowPath];
        pathsToDelete = @[self.actionRowPath];
        self.actionRowPath = before ? indexPath.next : indexPath;
        pathsToAdd = @[self.actionRowPath];
    }
    else
    {
        // New action cell
        pathsToAdd = @[indexPath.next];
        self.actionRowPath = indexPath.next;
    }

    // Animate the deletions and insertions
    [self.tableView beginUpdates];
    if (pathsToDelete.count)
        [self.tableView deleteRowsAtIndexPaths:pathsToDelete
        withRowAnimation:UITableViewRowAnimationNone];
    if (pathsToAdd.count)
        [self.tableView insertRowsAtIndexPaths:pathsToAdd
        withRowAnimation:UITableViewRowAnimationNone];
    [self.tableView endUpdates];
}

// Set up table
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.tableView.rowHeight = 60.0f;
    self.tableView.backgroundColor =
        [UIColor colorWithWhite:0.75f alpha:1.0f];

    [self.tableView registerClass:[UITableViewCell class]
    forCellReuseIdentifier:@"cell"];
    [self.tableView registerClass:[CustomCell class]

```

```
forCellReuseIdentifier:@"action"];
items = [@"Alpha Bravo Charlie Delta Echo Foxtrot Golf \
Hotel India Juliet Kilo Lima Mike November Oscar Papa \
Quebec Romeo Sierra Tango Uniform Victor Whiskey Xray \
Yankee Zulu" componentsSeparatedByString:@" "];
}
```

viewDidLoad 方法注册了两种单元格类型，一种是表格中的普通行，另一种是指令行。如果系统向数据源方法索取的单元格是 CustomCell 类型的，那么该方法就会返回适当的指令行。

指令单元格 (action cell) 还有其他一些比较麻烦的地方。例如，用户是不能选中这种单元格的。为了确保这一点，解决方案 9-7 的 tableView: willSelectRowAtIndexPath: 方法若发现传入的索引路径表示指令行，则会返回 nil。

这条解决方案的重要实现代码基本上都写在 tableView:didSelectRowAtIndexPath: 方法里。该方法会修改指令行的路径，并更新表格中的单元格，以便把指令抽屉移动到适当的位置上。我们要考虑三种情况：用户在指令行没有显示出来的时候点击了新的单元格；用户在指令行显示出来之后又点击了原来那个单元格；用户在指令行显示出来之后点击了其他单元格。

当指令抽屉收起来的时候，指令行的索引路径总是 nil。如果用户点击了某个单元格，那么 tableView:didSelectRowAtIndexPath: 方法就把指令行的路径设置到单元格下方。如果用户在指令行显示出来的时候又点击了它上方的那个相关单元格，那么指令抽屉就会“关上”，而它的索引路径也会设为 nil。如果用户点击的不是原来那个单元格，那么该方法就会执行一些运算，根据新单元格是位于原有指令行的下方还是上方来做出相应的调整。

我们把与表格有关的操作都放在配套的 beginUpdates 方法与 endUpdates 方法之间，以便能够同时展示其动画效果。在移动、添加及移除指令行时，表格的内容要发生变化，而把这些变化放在 beginUpdates 与 endUpdates 之间执行，则可令这个过程更加平滑。

### 9.13 制作自定义的分组表格

在 iPhone 的表格中，如果把依照首字母来分区的表格看成 M. C. Escher<sup>①</sup> 的画作，那么自由形式的分组表格就相当于 Marc Chagall<sup>②</sup> 的作品。前者的每个区段都精确地排布在表格中，而后的每个部分都可以自由定制，它是一件手绘的艺术品。

对于本章前面讲过的那些表格来说，只要掌握了诀窍，就不难用代码把它们全都编写出来。但若想用代码编出一张精巧的分组表格，可就相当不容易了（因为系统的 Settings 程序里面用到了这种表格，所以分组表格的爱好者把它叫作（偏好设置表格）。

① 莫里茨·科内利斯·埃舍尔 (Maurits Cornelis Escher, 1898–1972)，荷兰版画家，因绘画中的数学性而闻名。——译者注

② 马克·夏卡尔 (1887–1985)，超现实主义画家。——译者注

用代码来构建分组表格相当于拼贴 (collage)，我们要一块一块地把这张表格拼出来。以编程的方式创建这样的表格需要处理很多细节问题。

## 创建分组式的偏好设置表格

在制作偏好设置表格的过程中，新建并排布 `UITableViewController` 的操作并没有特别之处，依然需要先分配内存，然后用表格样式来初始化它。基本上就是这些了。困难的地方在于如何实现数据源方法与委托方法。下面列出开发者需要定义的方法：

- **`numberOfSectionsInTableView:`**——每一张偏好设置表格都含有一定数量的分组。每个分组的背景都是白色的，这与整张表格的灰色背景形成了对比。该方法返回表格中的分组数量，这个值是个整数。
- **`tableView:titleForHeaderInSection:`**——这个可选方法应该返回每个区段的标题。当程序询问某个区段的名称时，返回相应的 `NSString`。
- **`tableView:numberOfRowsInSection:`**——该方法返回每个区段所包含的单元格数量。程序询问某个分组所包含的行数（也就是单元格数量）时，该方法应该返回整数值。
- **`tableView:heightForRowAtIndexPath:`**——如果表格的行高是可以变化的，那么程序所要执行的运算量就会更大一些。若想使用可变的行高，那么就实现这个可选的方法，并返回某行的高度。我们需要依照程序所询问的区段及行来返回适当的值。
- **`tableView:cellForRowAtIndexPath:`**——本章多次用到了这个标准的方法，它负责提供与某一行相对应的单元格。对于偏好设置表格来说，它的实现方式与普通的表格不同。普通的表格通常只使用一种单元格，但偏好设置表格可能会创建出很多种可供复用的单元格，每种单元格都对应于一种单元格类型（而且有它自己的复用标记 (reuse tag)）。开发者应该仔细管理好复用队列，并且尽量使用与 IB 相集成的 UI 元件。
- **`tableView:didSelectRowAtIndexPath:`**——开发者需要根据用户所选单元格的类型，在这个可选的委托方法中根据情况做出响应。



**提示** Google Code 上面有个开源项目叫作 `llamasettings` (<http://llamasettings.googlecode.com/>)，它能够根据 iPhone 程序 `settings bundle` 中的属性列表自动生成分组表格。这样就能把这些设置选项直接集成到应用程序里面，使得用户无须离开程序，即可调整其设置。开发者可把该项目添加到用 iOS SDK 所开发的商业程序中，而无须支付授权费。

## 9.14 解决方案：构建含有多个滚轮的表格

有时用户需要从一份很长的列表中选择某个条目，或是需要同时从多份列表中分别做出选择。`UIPickerView` 实例很适合用在这些情况下。`UIPickerView` 对象制作出的表格提

供了很多各自独立的滚轮，如图 9-10 所示。用户可以通过操作这些滚轮来做出选择。

这些表格从表面上看与标准的 UITableView 实例相似，但它们却使用独特的数据源协议和委托协议：

- 没有 UIPickerViewController 类。UIPickerView 实例扮演其他视图的子视图。在应用程序里，它们并不打算成为视图的中心焦点。开发者可以把 UIPickerView 实例放在另一个视图里面。
- UIPickerView 采用数字而非对象来表示其中的滚轮。UIPickerView 里的组件（也就是滚轮）是用数字来编号的，而不与 NSIndexPath 实例相对应。这个类不像 UITableView 那样严整。

在实现数据源协议时，开发者既可以提供字符串，也可以提供视图。这两种方式 UIPickerView 都能处理。



图 9-10 UIPickerView 实例使得用户能够操作多个独立的滚轮，以做出自己的选择

#### 9.14.1 创建 UIPickerView

创建选取器的时候，别忘了设置 delegate 与 dataSource。如果不这样做，那么就无法向视图里添加数据、无法定义其特性，也无法响应用户所做的选择。主视图控制器应该实现 UIPickerViewDelegate 与 UIPickerViewDataSource 协议。

#### 9.14.2 数据源方法与委托方法

为了使 UIPickerView 具备最基本的功能，我们需要实现下面这三个关键的数据源方法。它们分别是：

- **numberOfComponentsInPickerView:**——该方法返回视图里的列数，其返回值是个整数。
- **pickerView:numberOfRowsInComponent:**——该方法返回每个滚轮所具备的行数，其返回值是个整数。滚轮的行数不一定都要相同。其中一个滚轮可以有好多行，而另外一个滚轮则可以只有几行。
- **pickerView:titleForRow:forComponent:** 或 **pickerView:viewForRow:forComponent:reusingView:**——这两个方法可以为滚轮中的某一行指定文本或视图，以使用作该行的标签。

除了这些数据源方法之外，还可以再实现一个委托方法，该方法能够响应用户对某个滚轮的选取操作：

- **pickerView:didSelectRow:inComponent:**——可以在该方法内实现与应用程序有关的行为。如果有必要，可以在 pickerView 上面调用 **selectedRowInComponent:** 方法来查询用户在某个滚轮中选定了哪一行。

### 9.14.3 使用带有选取器的视图

选取器视图采用了一套简单的视图复用方式，它会把提供给自己的视图缓存起来，以便稍后重新使用。如果 `pickerView:viewForRow:forComponent:reusingView:` 方法的最后一个参数不是 `nil`，那么开发者可以重新使用经由该参数所传入的视图，并更新其设置或内容。我们应该检查这个参数，只在程序没有提供所需的视图时才去新建它。

滚轮每一行的高度不一定要与实际的视图相符。我们可以实现 `pickerView:rowHeightForComponent:` 方法，并在其中指定每个滚轮每一行的高度。解决方案 9-8 把行高设为 120 个点，这样就有足够的空间来显示图像了，而且还可以造成一种滚轮可以无限循环滚动的假象，令用户察觉不到它的起点和终点。

解决方案 9-8 模拟可以反复滚动的圆柱形滚轮

---

```

- (NSInteger)numberOfComponentsInPickerView:
    (UIPickerView *)pickerView
{
    return 3; // three columns
}

- (NSInteger)pickerView:(UIPickerView *)pickerView
    numberOfRowsInComponent:(NSInteger)component
{
    return 1000000; // arbitrary and large
}

- (CGFloat)pickerView:(UIPickerView *)pickerView
    rowHeightForComponent:(NSInteger)component
{
    return 120.0f;
}

- (UIView *)pickerView:(UIPickerView *)pickerView
    viewForRow:(NSInteger)row forComponent:(NSInteger)component
    reusingView:(UIView *)view
{
    // Load up the appropriate row image
    NSArray *names = @[@"club", @"diamond", @"heart", @"spade"];
    UIImage *image = [UIImage imageNamed:names[row%4]];

    // Create an image view if one was not supplied
    UIImageView *imageView = (UIImageView *) view;
    imageView.image = image;
    if (!imageView)
        imageView = [[UIImageView alloc] initWithImage:image];

    return imageView;
}

- (void)pickerView:(UIPickerView *)pickerView

```



```

didSelectRow:(NSInteger)row inComponent:(NSInteger)component
{
    // Respond to selection by setting the view controller's title
    NSArray *names = @[@"C", @"D", @"H", @"S"];
    self.title = [NSString stringWithFormat:@"%02d%02d%02d",
        names[[pickerView selectedRowInComponent:0] % 4],
        names[[pickerView selectedRowInComponent:1] % 4],
        names[[pickerView selectedRowInComponent:2] % 4]];
}

- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];

    // Set random selections as the view appears
    [picker selectRow:50000 + (rand() % 4) inComponent:0 animated:YES];
    [picker selectRow:50000 + (rand() % 4) inComponent:1 animated:YES];
    [picker selectRow:50000 + (rand() % 4) inComponent:2 animated:YES];
}

- (void)loadView
{
    self.view = [[UIView alloc] init];
    self.view.backgroundColor = [UIColor whiteColor];

    // Create the picker and center it
    pickerView = [[UIPickerView alloc] initWithFrame:CGRectMake(0, 0, 300, 100)];
    [self.view addSubview:pickerView];
    PREPCONSTRAINTS(pickerView);
    CENTER_VIEW_H(self.view, pickerView);
    CENTER_VIEW_V(self.view, pickerView);

    // Initialize the picker properties
    pickerView.delegate = self;
    pickerView.dataSource = self;
    pickerView.showsSelectionIndicator = YES;
}

```

请注意，我们把每个滚轮的总行数定为一百万。之所以要用这么大的值，就是为了模拟出真实的圆柱形滚轮（筒状滚轮）。一般来说，UIPickerView 里面的滚轮都有起始元素和终止元素，滚轮到了最后一个元素之后，就不能再继续滚动了。于是，我们会想：如果程序里的滚轮和真实的圆柱形滚轮一样，那么在到达了最后一项之后，接下来应该又出现第一项才对。为了模拟这种效果，本条解决方案把滚轮的行数设置成了一个非常大的值，使得用户不太可能卷动到滚轮的末端。它调用 `selectRow:inComponent:animated:` 方法，把滚轮的初始位置设为这个大值的一半。我们把用户当前滚动到的行数与滚轮中的图像种数相除，并根据余数来决定这一行应该显示哪幅图像（具体到本例来说，就是 `row % 4`）。虽说程序代码把每个滚轮的总行数设成了一百万，但用户却会以为这是个圆柱形滚轮，它里面只有四行。



在早前的 iOS 程序中, 如果用户要在某个视图中通过 UIPickerView 来选取数据, 那么这个 UIPickerView 通常会出现另一个视图中。比方说, 用户在点击了某个日期字段之后, 程序会展示一个新的视图, 并把选取日期所用的 UIPickerView 显示在那个视图里面。在 iOS 7 中, 苹果公司已经开始把 UIPickerView 直接嵌入到应用程序的内容中了。苹果公司的《Human Interface Guidelines》(人机界面指南, HIG) 里面说, 应该把 UIPickerView 嵌在内容之中, 无须令用户跳转到另外一个视图。我们在 iOS 7 的 Calendar 程序中就可以看到这种用法。

## 9.15 使用 UIDatePicker

有时可能需要请用户输入日期信息。苹果公司提供了非常好的 UIPickerView 子类, 用于处理几种日期与时间的输入。图 9-11 演示了四种内置的 UIDatePicker 样式, 它们分别用来选择时间、选择日期、选择日期与时间以及设定倒计时时。

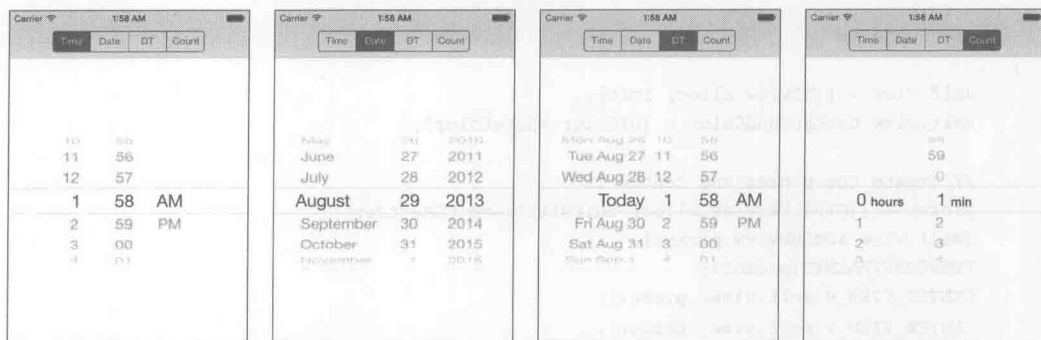


图 9-11 iPhone 提供了四种内置的数据选取器模型。开发者可以通过 datePickerMode 属性来设定程序所需的选取器

### 创建 UIDatePicker

UIDatePicker 的创建过程与 UIPickerView 的相同, 两者的布局方式也一样。创建好 UIDatePicker 对象之后, 就简单多了。我们不需要设置委托, 也不需要定义数据源方法, 而且还不用声明任何协议, 只需要为 UIDatePicker 指定一种模式就好。可供选择的模式有 UIDatePickerModeTime、UIDatePickerModeDate、UIDatePickerModeDateAndTime 及 UIDatePickerModeCountDownTimer;

```
[datePicker setDate:[NSDate date]]; // set date
datePicker.datePickerMode = UIDatePickerModeDateAndTime; // set style
```

开发者可以添加目标, 以便侦测用户通过 UIDatePicker 所做的修改 (此时会发生 UIControlEventValueChanged 事件), 同时需要创建目标 - 动作组合所使用的回调方法。

使用 UIDatePicker 类的时候, 可能需要操控下面几个属性:

- **date**——初始化 `UIDatePicker` 的时候，可以通过该属性来设定初始的日期；用户通过滚轮操作 `UIDatePicker` 之后，开发者可以通过该属性获取用户所选定的日期。
- **maximumDate** 和 **minimumDate**——这两个属性用来限定用户所能选取的日期及时间范围。我们应该给每个属性赋予标准的 `NSDate` 值。借助这一手段，开发者可以令用户只能选择一年之后的某个日期，而不是先等用户选择完了，然后再去判断所选日期是否处在可以接受的范围内。
- **minuteInterval**——有时我们想令用户在选择时间的时候，必须以 5 分钟、10 分钟、15 分钟或 30 分钟为间隔进行选择，比方说安排约会事项所用的应用程序可能就需要这样做。`minuteInterval` 属性用来指定这个间隔值。开发者所传入的值必须能为 60 所整除。
- **countDownDuration**——该属性用来设置用户能够在倒数计时器里选择的最大值。`countDownDuration` 最多可以达到 23 小时 59 分（也就是 86 399 秒）。

## 9.16 小结

本章介绍了 iOS 表格的用法，其中有些表格比较简单，有些则比较复杂。各种基本的 iOS 表格都讲到了，包括简单的表格、可供编辑的表格以及可以重排单元格顺序并支持撤销功能的表格。读者也学到了一些高级的 UI 元件，其中有按字母排序的索引列表、刷新控件以及选取器视图等。掌握了本章所讲的技巧之后，你就可以为 iPhone、iPad 及 iPod touch 构建出一大批基于表格的应用程序了。现在将本章的要点总结如下：

- 在学习表格的用法时，一定要明白数据源方法与委托方法之间的区别。数据源方法用来向表格里填充有意义的内容，而委托方法则用于响应用户的操作。
- 如果应用程序以 `UITableView` 为中心，那么就可以采用 `UITableViewController` 来简化程序的构建过程。不过，当程序需要直接使用 `UITableView` 的时候，则应该毫不犹豫地使用它，尤其是在 `popover` 里面，或是在与分栏视图控制器相搭配时，更应如此。在必要的时候，应该明确宣称自己所写的类支持 `UITableViewDelegate` 及 `UITableViewDataSource` 协议。
- 对于比较大的有序列表来说，索引是一种快速浏览表格的好办法。在编写表格的时候，应该善用索引，否则那些比较大的表格就不便于浏览了。从美观角度来讲，分组表格最好不要使用索引。
- 应该研究一下编辑功能。我们很容易就为用户提供编辑表格数据的功能，而且写好的代码还可以复用到很多项目之中。从设计程序之初就应该考虑撤销功能。即便一开始你觉得用不到这个功能，稍后也有可能改变主意。
- 把普通表格转变成分区表格是相当简单的。你应该用本章所介绍的办法，借助谓词，以简单的数组创建出表格里各个区段。分区表格能够更加有条理地展示数据，而且还支持索引，同时，开发者也可以把易于使用的搜索功能集成到里面。
- 数据选取器是一种非常专业的控件，很适合用来选取日期和时间。而选取器视图则提供了一种比较通用的解决方案，它需要开发者编写更多的代码才能运作。

## 集合视图

iOS 6 引入的集合视图 (collection view) 是一种网格状视图, 用来排布其中的各个单元格。这种集合视图的功能远远超过标准的表格, 它们不仅仅是能够垂直滚动的一系列单元格。集合视图的许多概念与表格相同, 但却更加强大, 也更为灵活。开发者可以用集合视图创建出横向滚动的列表、网格, 以及一些特殊的布局, 例如圆形布局等。此外, 该类还能通过布局规格提供内置的视觉效果, 并且支持很多有用的特性, 例如滚动之后自动就位。

与使用表格时一样, 你也可以向集合视图里添加大量的实现细节。本章将介绍集合视图的一些基本知识, 包括它的数据源、特殊用途的控制器以及单元格等。读者将会学到如何开发标准的集合视图与自定义的集合视图, 如何在视图添加特效, 以及如何利用内置的动画功能来创建出相当高效的交互方式。

大家要知道, 集合视图是非常强大的, 只用一章的篇幅不可能将它全部涵盖。本章只是讲解集合视图的一些基本概念。学会了这部分内容之后, 你应该自己去探索它的用法并积累使用经验。

### 10.1 集合视图与表格的异同

`UICollectionView` 实例会把各项数据展示成一份有序的集合。与表格视图一样, 集合视图也由单元格、头部及尾部构成, 而且由数据源方法及委托方法所驱动。但与表格不同的地方在于, 集合视图还引入了与布局有关的类, 这个类用来指定各条目应该如何摆放在屏幕上。该类负责管理每个单元格的位置, 使得对应的条目可以在必要时出现在适当的地方。

表 10-1 比较了集合视图与表格这两种布局。正如大家所见, 两者都提供了核心的视图类及预置的控制器类。这些类都依赖于数据源。数据源负责填充单元格, 并提供其他内容信

息。此外，两者都需要通过委托来响应用户的操作。

表 10-1 集合视图与表格之间的对比

对比项	集合视图	表 格
主类	UICollectionView	UITableView
控制器	UICollectionViewController	UITableViewController
内容	单元格、补充视图（例如头部和尾部）、装饰视图（背景图版以及视觉饰件）	单元格、头部、尾部
由用户所触发的重新载入	在用户浏览的时候会实时更新自己，以便与当前数据相符。特定的情况下会使用刷新控件	刷新控件 (UIRefreshControl)
以编程方式触发的重新载入	reloadData	reloadData
可复用的单元格	UICollectionViewCell (通过 dequeue-ReusableCellWithReuseIdentifier:forIndexPath: 方法获取)	UITableViewCell (通过 dequeue-ReusableCellWithReuseIdentifier:forIndexPath: 方法获取)
注册	用类或 XIB 来注册可复用的单元格、补充视图以及装饰视图	用类或 XIB 来注册可复用的单元格
头部及尾部	UICollectionViewReusableView	UITableViewHeaderFooterView
布局	UICollectionViewLayout 及 UICollectionViewFlowLayout	不适用
数据源	UICollectionViewDataSource	UITableViewDataSource
委托	UICollectionViewDelegate	UITableViewDelegate
用于布局的委托	UICollectionViewDelegateFlowLayout	不适用
索引机制	通过区段与条目来定位	通过区段与行来定位
滚动方向	水平或垂直	垂直
视觉效果	通过自定义的布局来设置	不适用

两者之间还是有许多根本区别的，我们先从不太起眼的索引路径说起。这两个类都把区段用作主要的分组方式，每个区段里面都含有一些单元格，这些单元格又都有各自的索引号。由于集合视图既可以垂直滚动，也可以水平滚动，所以它使用的术语就与表格不同了。表格采用区段和行来定位单元格，而集合视图则采用区段和条目来定位。iOS 6 更新了 NSIndexPath 类，以适应这一变化。

集合视图引入了一种新的内容——装饰视图，这种视图可以提供诸如背面图版 (backdrop) 等视觉增强效果。对于集合视图来说，单元格与滚动方向只是最基本的定制方式。开发者可以使用所能想到的任何隐喻来定制整套视图的样貌，使其与自己所要表达的概念相一致。集合视图的头部与尾部也与表格视图不同，它把这两者转化成了补充视图，并且提供了比表格稍微灵活一些的 API。

## 两者在实现层面的区别

编写实际的程序时，构建表格视图和构建集合视图所用的代码有几个地方是不同的。集合视图不太允许数据的延迟加载。有一条经验：创建集合视图的时候，为该视图提供内容的数

据源必须完全准备好，哪怕它现在只能为视图里的少数几个单元格提供数据，甚至暂时没有单元格数据都可以，但只要它自身准备好就行，稍后我们可以再从程序的其他地方加载数据。

我们不能等到程序执行初始化方法、loadView 方法或 viewDidLoad 方法的时候再准备数据源，而是必须首先把它准备好。可以在应用程序委托中准备，也可以在实例化集合视图并将其添加到其他视图之前，或是在把新的集合视图控制器设为其他对象的子控制器之前把数据源准备好。要是没准备好，程序就会崩溃，而这肯定不是我们想要的用户体验。

展示集合视图之前，一定要把集合视图的布局对象<sup>①</sup>完全建立好。大家在本章稍后的解决方案里面会看到，我们需要把布局对象的所有细节都设置好，包括滚动方向以及其他不依赖于委托回调的属性。只有在准备好这些之后，才能创建并初始化集合视图：

```
MyCollectionController *mcc = [[MyCollectionController alloc]
initWithCollectionViewLayout:layout];
```

如果传给 layout 参数的值是 nil，就会抛出异常。

在集合视图的生命期中，并不是只能使用一种布局。开发者可以通过 collectionViewLayout 属性来直接访问集合视图的布局。修改这个属性之后，视图就会以不带动画的方式立即更新其布局。iOS 7 提供了一个简单的方法，能够用动画效果来展示布局的变更过程：

```
- (void)setCollectionViewLayout:(UICollectionViewLayout *)layout
animated:(BOOL)animated completion:(void (^)(BOOL finished))completion
```

苹果公司在 iOS 7 中引入了一套机制，用于创建更为复杂且更具交互性的切换效果。这个问题已经超出了本书的范围，不过你可以在苹果公司的 iOS Developer Center 中查看《UICollectionView Class Reference》以获知更多信息，也可以在 Xcode 的 iOS 7 docset 中查询此信息。

## 10.2 建立集合视图

与表格一样，集合视图也有两种用法，一种是直接使用集合视图，另一种是使用系统预置的控制器。开发者可以构建一份单独的集合视图实例，并把它添加到界面中，也可以使用更为方便的 UICollectionViewController 对象，该对象是个预先制备好的视图控制器，其中带有一份集合视图。这个控制器会自动把视图的数据源及委托设置成该控制器本身，而且还会宣称自己遵从两个相关的协议。这样的集合视图控制器既可以用作其他容器（比如导航控制器、标签栏控制器、分栏视图控制器、页面视图控制器等）的子控制器，也可以独立展示出来。



**提示** 与表格视图一样，集合视图也有 delegate 及 dataSource 属性。UICollectionViewFlowLayout 类期望集合视图的 delegate 还能够遵从 UICollectionViewDelegateFlowLayout 协议。开发者可以在集合视图控制器里面实现这三个协议<sup>②</sup>中的相关方法。

① 也就是 UICollectionViewLayout 类型的对象。——译者注

② 另外两个协议是 UICollectionViewDelegate 和 UICollectionViewDataSource。——译者注

## 10.2.1 通过控制器使用集合视图

构建控制器的时候，首先要创建并设置好布局对象，然后分配新的实例，并用准备好的布局对象初始化它：

```
UICollectionViewFlowLayout *layout =
    [[UICollectionViewFlowLayout alloc] init];
layout.scrollDirection = UICollectionViewScrollDirectionHorizontal;

MyCollectionController *mcc = [[MyCollectionController alloc]
    initWithCollectionViewLayout:layout];
```

上面代码采用 `UICollectionViewFlowLayout` 作为布局对象，并且使用它的默认配置，只是修改了一下滚动方向。在本章后面的解决方案中，大家会看到，我们可以在布局对象上面配置更多的属性。通常我们都会再设置一些属性，或是从系统所提供的类中继承子类，并在子类里面添加自定义的行为代码。

一般来说，使用 `UICollectionViewFlowLayout` 类就行了。集合视图里的许多布局任务，它都能够完成。我们可以用它构建出基本的界面。在默认状态下，每个区段都会根据屏幕内容自行调整该区段中的条目，而开发者则可以指定区段之间、线条之间以及条目之间的空白尺寸等。下一节会详细讲述 `UICollectionViewFlowLayout` 里面许多可供调整的部分，大家将看到它的可定制程度相当高。

`UICollectionViewFlowLayout` 类的父类叫作 `UICollectionViewLayout`，它是个可供继承的抽象基类，不过，绝大部分情况下都不应该从它继承子类，而是应该从 `UICollectionViewFlowLayout` 继承。开发者并不需要直接使用这个父类。



**提示** 如果要继承布局类的子类，请参阅 `UICollectionViewLayout` 的文档。这个 `UICollectionViewFlowLayout` 父类的文档里面，规范地列出了各种可供定制的方法。

## 10.2.2 直接使用集合视图

如果想建立嵌入其他视图之中的集合视图（也就是不带 `UICollectionViewController` 的集合视图），就先创建好布局对象，然后用布局对象来创建集合视图，并设置数据源及委托。`UICollectionViewFlowLayout` 对象会使用开发者经由 `delegate` 属性赋给集合视图的委托：

```
UICollectionViewFlowLayout *layout =
    [[UICollectionViewFlowLayout alloc] init];
layout.scrollDirection = UICollectionViewScrollDirectionHorizontal;

collectionView = [[UICollectionView alloc] initWithFrame:CGRectZero
    collectionViewLayout:layout];
collectionView.dataSource = self;
collectionView.delegate = self;
```



### 10.2.3 数据源与委托

管理集合视图的视图控制器宣称自己实现了 `UICollectionViewDataSource` 及 `UICollectionViewDelegate` 协议。但与表格不同的是，如果使用流式布局 (flow layout)，那么控制器还会宣称自己实现了 `UICollectionViewDelegateFlowLayout` 协议。

`UICollectionViewDelegateFlowLayout` 协议通过一系列回调方法来给集合视图的布局对象提供布局信息。集合视图的 `delegate` 遵从了这一协议，也就是说，我们无须另外用一个名为 `delegateFlowLayout` 的属性来表示实现了该协议的对象。

与表格视图一样，数据源也负责提供每个区段及区段内每个条目的信息，并根据需求返回对应的单元格以及集合视图上面的其他部件。委托负责处理用户操作，并对用户的改动请求做出有效回应。而 `UICollectionViewDelegateFlowLayout` 则负责提供每个区段的详细布局信息，在大多数情况下，它所规定的方法都是可选的。下一节将会讲解流式布局以及与之相关的委托回调。

## 10.3 流式布局

由 `UICollectionViewFlowLayout` 类所提供的流式布局会在应用程序里创建出网格状的界面。它们有一些内置的属性，开发者既可以直接设置这些属性，也可以通过委托回调来提供属性值。这些属性用来指定布局对象应该如何配置自己，才能把各条目适当地显示在屏幕上面。从最简单的角度来说，这些布局属性可以看作一份与几何特征有关的字典，它们描述了行间距、缩进，以及条目之间的留白等特征。

### 10.3.1 滚动方向

`scrollDirection` 属性决定了视图中的区段是水平排列 (`UICollectionViewScrollDirectionHorizontal`) 还是垂直排列 (`UICollectionViewScrollDirectionVertical`)。图 10-1 中的左右两幅截图，分别演示了水平的流式布局以及垂直的流式布局，除了方向不同之外，这两种布局方式在其他方面都是相似的。每一个区段内的条目会根据可用的空间自动换行。iPhone 竖屏模式的垂直空间要多于水平空间，所以水平布局下的每个区段都会显得比垂直布局下的区段更窄。

### 10.3.2 条目的尺寸以及行间距

`itemSize` 属性可用来指定屏幕上每个条目的默认大小，就像图 10-1 中的小方块那样。`minimumLineSpacing` 及 `minimumInteritemSpacing` 属性规定了每个区段内的条目之间应该距离多远。行间距 (line spacing) 总是表示相邻两行之间的距离，在水平布局下行是垂直方向的，而在垂直布局下行则是水平方向的。比方说，在图 10-1 左侧截图中，行间距指的就是 `S0(0)` 和 `S0(6)` 之间的距离，而在右侧截图中，指的则是 `S0(0)` 和 `S0(4)` 之间的距离。条目间距 (item spacing) 与行间距的延伸方向相互垂直，它表示相邻两个条目之间的空隙，例如 `S0(0)` 和 `S0(1)` 之间的距离，以及 `S0(1)` 和 `S0(2)` 之间的距离等。



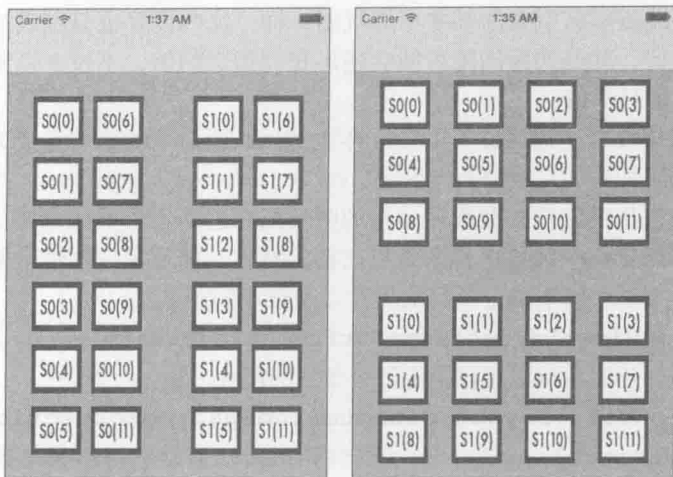


图 10-1 集合视图的总体滚动方向可以设为水平（如左侧截图所示）或垂直（如右侧截图所示）。左侧截图中的集合视图可以左右滚动，而右侧截图中的集合视图则是上下滚动的。采用流式布局排列各条目的时候，它会在到达每一行的末尾之后自动换行。在左侧截图中，每个纵向的行都有 6 个条目，而在右侧截图中，每个横向的行则有 4 个条目。每个区段内均包含 12 个条目

图 10-2 演示了这两个属性，它使用的是垂直流式布局。在左侧截图中，行间距与条目间距都是 10 个点。而在中间截图中，我们把行间距增大到 50 点。这个间距出现在相邻两行条目之间，布局对象在排布各条目的时候，如果到了某一行的尾部，会把下一个条目摆在下一行开头。右侧截图把条目间距扩大为 30 点。条目间距表现为水平方向上的间隔距离，也就是相邻两个条目之间的间隔距离。

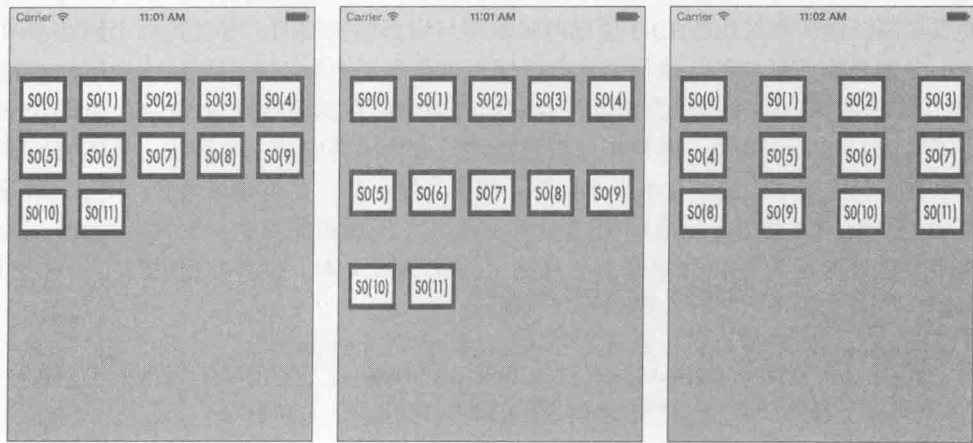


图 10-2 最小行距（`minimumLineSpacing`）与最小条目间距（`minimumInteritemSpacing`）决定了每个区段内的条目应该如何排列。而条目尺寸（`itemSize`）则决定了每个单元格的大小。左侧截图采用默认的间距。中间截图把行间距扩大到 50 点。右侧截图把条目间距扩大到 30 点

与 iOS 6 及后续版本新引入的某些布局机制一样，这些设置也只是表达对系统的一种请求。说得更明确一些，就是实际间距可能会超出我们指定的值，不过，系统在排版时会尽量满足开发者所指定的最小值。

开发者可以直接设置上述属性，如果这样做的话，所设定的值就会成为整个集合视图的默认值。另外，还可以在 `UICollectionViewDelegateFlowLayout` 协议的委托回调方法中以代码的形式来指定它们。在程序运行的时候指定这些值，要比直接设置默认值更为精细，因为可以具体指定每个区段内每个条目的相关特征，而不是一次性地影响集合视图内的所有条目。下面的方法可用来指定条目的尺寸以及最小间距：

- `collectionView:layout:sizeForItemAtIndexPath:`——该方法与 `itemSize` 属性相对应，不过它可以具体指定每个条目的尺寸。
- `collectionView:layout:minimumLineSpacingForSectionAtIndex:`——该方法与 `minimumLineSpacing` 属性相对应，但是它可以具体控制每个区段内的最小行间距。
- `collectionView:layout:minimumInteritemSpacingForSectionAtIndex:`——该方法与 `minimumInteritemSpacing` 属性相对应，然而它可以具体控制每个区段内的最小条目间距。

这三个方法之中，第一个方法在开发 iOS 程序的时候最容易用到。它使得开发者可以构建出条目大小各不相同的集合视图，而不是像图 10-2 那样，所有条目的尺寸都完全一样。在本章稍后的图 10-4 中，大家可以看到一种流式布局，它能够根据大小多变的单元格来调整自己。

### 10.3.3 头部与尾部的尺寸

`headerReferenceSize` 及 `footerReferenceSize` 属性定义了区段的头部和尾部应该多宽或多高。请注意对比图 10-3 顶部两张截图与底部两张截图，看看这两个属性在两种布局方向上分别是如何延伸的。顶部两张截图使用的是水平布局，我们把这两个属性的宽度设为 60 点；底部两张截图采用垂直布局，我们把这两个属性的高度设为 30 点。虽说开发者提供的是完整的 `CGSize` 结构，但是布局对象每次只会使用其中的一个字段，具体使用哪个字段，要根据布局方向来确定。在水平布局下，使用宽度字段；在垂直布局下，使用高度字段。

下面两个回调方法用于生成图 10-3 这样的版式。虽说布局对象一次只会用到一个字段，但这两个方法还是会返回包含宽度值与高度值的完整结构体。如果委托没有实现这些方法，那么布局对象就会采用本节开头提到的两个属性：

```
- (CGSize) collectionView:(UICollectionView *)collectionView
    layout:(UICollectionViewLayout *)collectionViewLayout
    referenceSizeForHeaderInSection:(NSInteger)section
{
    return CGSizeMake(60.0f, 30.0f);
}

- (CGSize) collectionView:(UICollectionView *)collectionView
    layout:(UICollectionViewLayout *)collectionViewLayout
```

```

referenceSizeForFooterInSection: (NSInteger)section
{
    return CGSizeMake(60.0f, 30.0f);
}

```

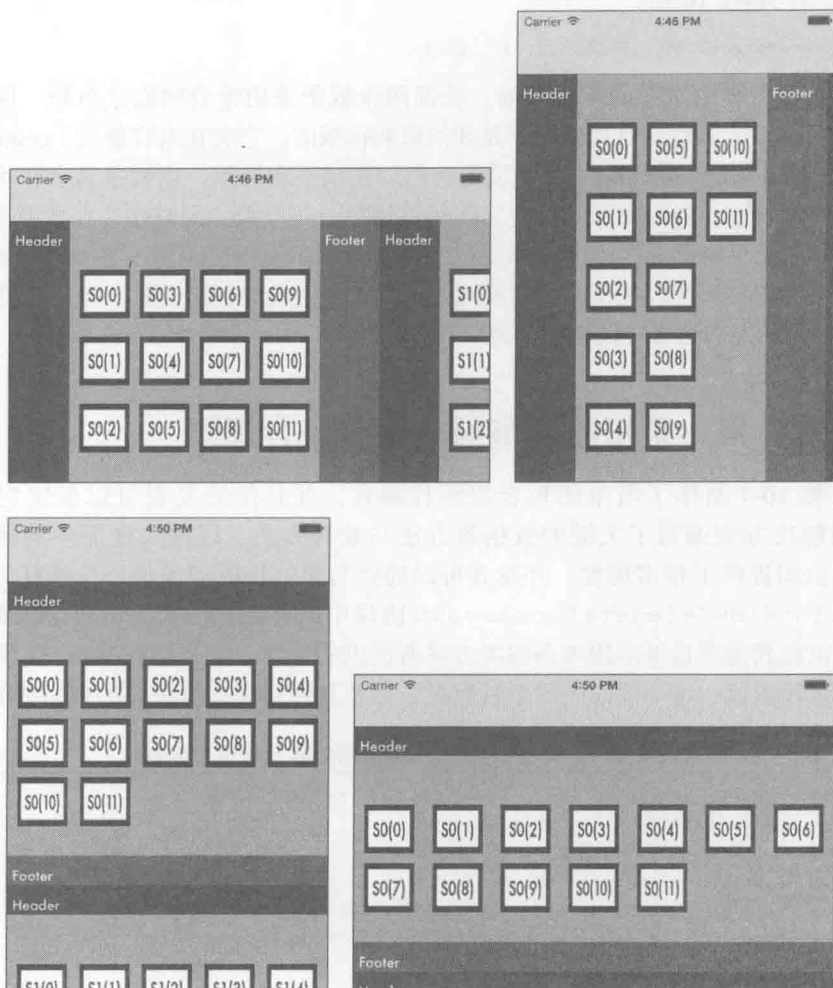


图 10-3 区段内边距决定了区段内的所有条目与外围视图边界之间的空隙。顶部两张截图采用水平布局，底部两张截图采用垂直布局。在这四张截图中，顶部内边距都是 50 点，底部内边距都是 30 点，左侧内边距与右侧内边距都是 10 点

### 10.3.4 内边距

最小行间距及最小条目间距两个属性定义了区段内的某个条目与其他条目之间的位置关系。与之相对，`sectionInset` 属性则描述了区段内所有条目的总体边界与外围的集合视图边界之间的距离。这段填充距离，既会影响区段与其头部和尾部之间的距离，又会影响两

个区段之间的距离。

每个内边距 (inset) 都是由一组 `{top, left, bottom, right}` 值构成的。图 10-3 演示了内边距对集合视图的影响。图 10-3 中的四张截图所采用的内边距完全相同，都是顶部 50 点、底部 30 点、左右两侧各 10 点：

```
UIEdgeInsetsMake(50.0f, 10.0f, 30.0f, 10.0f)
```

顶部两张截图采用水平的流式布局，底部两张截图采用垂直的流式布局。读者可以看到，`sectionInset` 属性在这些情况下是如何影响排版的。它会在内容条目 (content item)<sup>①</sup> 与外围容器 (enclosing container) 之间填入空白。在水平布局下，内容条目会在垂直方向上调整自己，使其顶边与外围视图的顶边之间能够留有一定空隙，同时还会在水平方向上调整自己，使其左右边界能够和区段的头部及尾部之间分别留出一定距离。在垂直布局下，内容条目的上下边界则要 and 区段的头部及尾部之间分别留出一定空隙。同时，其左右边界也必须与外围集合视图的左右边界之间留出距离。

## 10.4 解决方案：采用流式布局的简单集合视图

解决方案 10-1 制作了简单的集合视图控制器，并且使开发者可以指定它的头部及尾部。这条解决方案编写了关键的数据源方法与委托方法，以便实现简单的网格状流式布局。苹果公司提供了很多属性，开发者可以通过与集合视图有关的一些委托方法，以及 `UICollectionViewDelegateFlowLayout` 协议中的委托方法来提供与这些属性相对应的值。你可以在其他项目里沿用本条解决方案所提供的属性，并且稍微调整一下源代码，修改视图中的区段数量、每个区段内的条目数量，以及其他影响总体版式的布局细节。

解决方案 10-1 使用流式布局的简单集合视图控制器

```
@interface TestBedViewController : UICollectionViewController
// Layout and collection view configuration
@property (nonatomic, assign) BOOL useHeaders;
@property (nonatomic, assign) BOOL useFooters;
@property (nonatomic, assign) NSInteger numberOfSections;
@property (nonatomic, assign) NSInteger itemsInSection;
@end

@implementation TestBedViewController

#pragma mark Flow Layout
- (CGSize)collectionView:(UICollectionView *)collectionView
  layout:(UICollectionViewLayout *)collectionViewLayout
  referenceSizeForHeaderInSection:(NSInteger)section
{
    return self.useHeaders ? CGSizeMake(60.0f, 30.0f) : CGSizeZero;
}
```

① 可以理解为“由某区段内的全部条目所构成的整体”。——译者注

```

- (CGSize)collectionView:(UICollectionView *)collectionView
  layout:(UICollectionViewLayout *)collectionViewLayout
  referenceSizeForFooterInSection:(NSInteger)section
{
    return self.useFooters ? CGSizeMake(60.0f, 30.0f) : CGSizeZero;
}

#pragma mark Data Source
// Number of sections total
- (NSInteger)numberOfSectionsInCollectionView:
    (UICollectionView *)collectionView
{
    return self.numberOfSections;
}

// Number of items per section
- (NSInteger)collectionView:(UICollectionView *)collectionView
  numberOfItemsInSection:(NSInteger)section
{
    return self.itemsInSection;
}

// Dequeue and prepare a cell
- (UICollectionViewCell *)collectionView:
    (UICollectionView *)aCollectionView
  cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
    UICollectionViewCell *cell = [self.collectionView
        dequeueReusableCellWithReuseIdentifier:@"cell"
        forIndexPath:indexPath];

    cell.backgroundColor = [UIColor whiteColor];
    cell.selectedBackgroundView =
        [[UIView alloc] initWithFrame:CGRectZero];
    cell.selectedBackgroundView.backgroundColor =
        [[UIColor blackColor] colorWithAlphaComponent:0.5f];

    return cell;
}

// If using headers and footers, dequeue and prepare a view
- (UICollectionViewReusableView *)collectionView:
    (UICollectionView *)aCollectionView
  viewForSupplementaryElementOfKind:(NSString *)kind
  atIndexPath:(NSIndexPath *)indexPath
{
    if (kind == UICollectionViewElementKindSectionHeader)
    {
        UICollectionViewReusableView *header = [self.collectionView
            dequeueReusableSupplementaryViewOfKind:
                UICollectionViewElementKindSectionHeader

```

```

        withReuseIdentifier:@"header" forIndexPath:indexPath];
        header.backgroundColor = [UIColor blackColor];
        return header;
    }
    else if (kind == UICollectionElementKindSectionFooter)
    {
        UICollectionReusableView *footer = [self.collectionView
            dequeueReusableSupplementaryViewOfKind:
                UICollectionElementKindSectionFooter
                    withReuseIdentifier:@"footer" forIndexPath:indexPath];
        footer.backgroundColor = [UIColor darkGrayColor];
        return footer;
    }
    return nil;
}

#pragma mark Delegate methods
- (void)collectionView:(UICollectionView *)aCollectionView
    didSelectItemAtIndexPath:(NSIndexPath *)indexPath
{
    NSLog(@"Selected %@", indexPath);
}

- (void)collectionView:(UICollectionView *)aCollectionView
    didDeselectItemAtIndexPath:(NSIndexPath *)indexPath
{
    NSLog(@"Deselected %@", indexPath);
}

#pragma mark Setup
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Register any cell and header/footer classes for re-use queues
    [self.collectionView
        registerClass:[UICollectionViewCell class]
        forCellWithReuseIdentifier:@"cell"];
    [self.collectionView
        registerClass:[UICollectionReusableView class]
        forSupplementaryViewOfKind:UICollectionElementKindSectionHeader
        withReuseIdentifier:@"header"];
    [self.collectionView
        registerClass:[UICollectionReusableView class]
        forSupplementaryViewOfKind:UICollectionElementKindSectionFooter
        withReuseIdentifier:@"footer"];

    self.collectionView.backgroundColor = [UIColor lightGrayColor];

    // Allow users to select/deselect items by tapping
    self.collectionView.allowsMultipleSelection = YES;
}

```

```

- (instancetype)initWithCollectionViewLayout:(UICollectionViewLayout *)layout
{
    self = [super initWithCollectionViewLayout:layout];
    if (self)
    {
        // Set some reasonable defaults
        self.useFooters = NO;
        self.useHeaders = NO;
        self.numberOfSections = 1;
        self.itemsInSection = 1;
    }
    return self;
}
@end

// From the application delegate
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    _window = [[UIWindow alloc]
initWithFrame:[UIScreen mainScreen] bounds]];
    _window.tintColor = COOKBOOK_PURPLE_COLOR;

    // Create the layout and then pass to our collection VC
    UICollectionViewFlowLayout *layout =
        [[UICollectionViewFlowLayout alloc] init];
    TestBedViewController *tbvc = [[TestBedViewController alloc]
initWithCollectionViewLayout:layout];
    tbvc.edgesForExtendedLayout = UIRectEdgeNone;

    // Configure layout and collection view properties
    layout.itemSize = CGSizeMake(50.0f, 50.0f);
    layout.sectionInset =
        UIEdgeInsetsMake(10.0, 10.0f, 50.0f, 10.0f);
    layout.scrollDirection =
        UICollectionViewScrollDirectionVertical;
    layout.minimumLineSpacing = 10.0f;
    layout.minimumInteritemSpacing = 10.0f;
    tbvc.numberOfSections = 10;
    tbvc.itemsInSection = 12;
    tbvc.useHeaders = YES;
    tbvc.useFooters = YES;

    UINavigationController *nav = [[UINavigationController alloc]
initWithRootViewController:tbvc];
    _window.rootViewController = nav;
    [_window makeKeyAndVisible];
    return YES;
}

```

### 获取解决方案代码

访问 <https://github.com/erica/iOS-7-Cookbook> 网页，并打开“C10 Collections”文件夹，即可找到与本章中的解决方案相对应的完整范例项目。

代码中的两个布尔属性分别用来决定集合视图是否使用头部及尾部。如果使用的话，可以通过解决方案 10-1 中的前两个方法来提供与 `headerReferenceSize` 及 `footerReferenceSize` 相对应的值。这两个方法定义在 `#pragma mark Flow Layout` 指令下方。`UICollectionViewDelegateFlowLayout` 协议的委托方法如果把 0 作为头部或尾部的尺寸返回给调用者，就表明集合视图的相关区段不需要使用头部或尾部。若是返回其他尺寸，那么集合视图还会继续询问将要用作头部或尾部的补充视图是什么。

在数据源中使用单单元格或补充视图之前，一定要先把它们注册好。解决方案 10-1 在 `viewDidLoad` 方法里面注册了与它们相对应的类。注册好之后，就可以根据需要从队列里面取出这些实例了。开发者无须检查取出来的实例是否可用，因为负责从队列中取出实例的方法会在必要时自行创建并初始化相关实例。

笔者觉得你应该研究一下解决方案 10-1 中的范例代码，并且试着调整与布局相关的每个值和每个回调方法（前一节的那些图就是通过这种手段制作出来的），看看它们对集合视图的总体版式及样貌有什么影响。解决方案 10-1 是个很好的出发点，读者可以由此来测试集合视图，并观察每个属性怎样影响最终的版式。

## 10.5 解决方案：自定义单单元格

解决方案 10-1 创建出来的物件，其大小完全相同，不过，我们未必非要用尺寸完全一致的物件来填充集合视图。在流式布局之下，开发者可以创建出图 10-4 这样相当多变的界面。解决方案 10-2 改编了原有的集合视图，并创建了一些自定义的单元格，以便实现出丰富多彩的界面。这些单元格里面都含有 `UIImageView`（图像视图）。系统会向集合视图的数据源方法查询“位于某索引路径处的条目是何尺寸”，而该方法则会把图像的大小返回给系统。

```
- (CGSize) collectionView:(UICollectionView *)collectionView
    layout:(UICollectionViewLayout*)collectionViewLayout
    sizeForItemAtIndexPath:(NSIndexPath *)indexPath
{
    UIImage *image = artDictionary[indexPath];
    return image.size;
}
```

为了创建自定义的单元格，我们从 `UICollectionViewCell` 中继承了子类，并且把新的视图添加到单元格的 `contentView` 之中。这条解决方案给 `contentView` 里面添加了一个 `UIImageView` 类型的子视图，并且用 `imageView` 属性来表示这个子视图。当系统向集合视图索要单元格的时候，数据源方法会把自定义的图像添加到 `imageView` 里面，而负责布局的委托方法则会向系统提供单元格的尺寸。



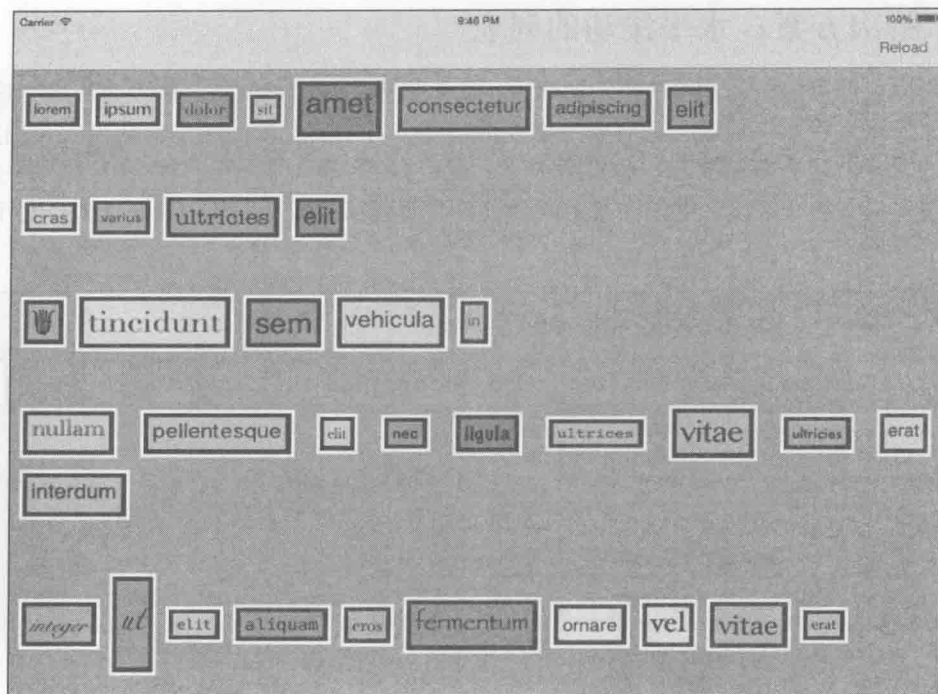


图 10-4 在流式布局下，集合视图中的各条目既可以呈现基本的网格状，也可以各自拥有不同的高度及宽度

#### 解决方案 10-2 对集合视图里的单元格进行定制

```
@interface ImageCell : UICollectionViewCell
@property (nonatomic) UIImageView *imageView;
@end

@implementation ImageCell
- (instancetype)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self)
    {
        _imageView = [[UIImageView alloc] initWithFrame:
            CGRectInset(self.bounds, 4.0f, 4.0f)];
        _imageView.autoresizingMask =
            UIViewAutoresizingFlexibleWidth |
            UIViewAutoresizingFlexibleHeight;
        [self.contentView addSubview:_imageView];
    }
    return self;
}
@end
```

## 10.6 解决方案：水平滚动的列表

用集合视图可以创建水平滚动的列表，而不像表格视图那样，只能创建出垂直滚动的列表。如果想实现水平滚动，就需要考虑几件事情，其中的主要问题是：在流式布局之下，区段内的条目默认会自动换行。请看图 10-5。图中有两个集合视图，它们都可以水平滚动。顶部截图里只有 1 个区段，区段内有 100 个条目，而底部截图里则有 100 个区段，每个区段内只有 1 个条目。

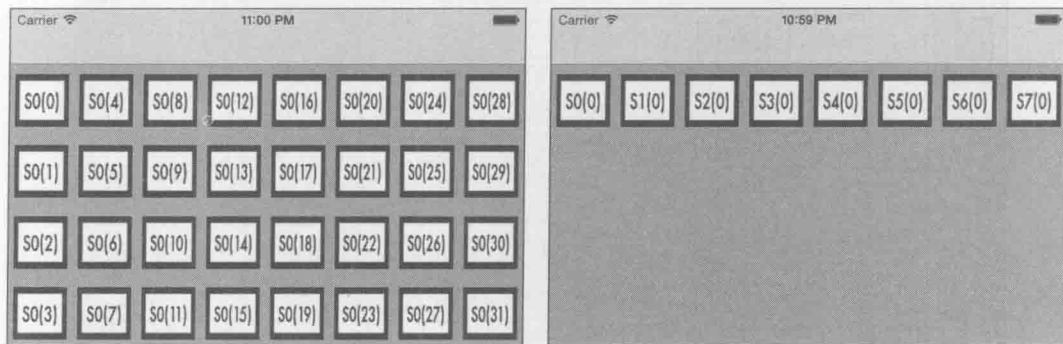


图 10-5 顶部截图：集合视图内只有 1 个区段，其中有 100 个条目。底部截图：集合视图内有 100 个区段，每个区段内只有 1 个条目

对于顶部截图中的情况来说，我们也可以给区段左右添加大量的留白，迫使区段内的条目不要折行，但这样很难实现出正确的效果，因为添加的间隔大小要取决于设备及屏幕方向。而给每个区段只安排一个条目，则要简单得多，因为无论条目的尺寸是多少，我们总能将其排成一行。

解决方案 10-3 把这个水平滚动列表创建成了独立的视图，而不是视图控制器。这样做开发者可以将其作为子视图嵌入其他视图之中，从而避免屏幕底部出现大块空白（如图 10-5 底部截图所示）。

### 解决方案 10-3 水平滚动的集合视图

```
@interface InsetCollectionView : UIView
<UICollectionViewDataSource>
@property (strong, readonly) UICollectionView *collectionView;
@end

@implementation InsetCollectionView

// 100 sections of 1 item each
- (NSInteger)numberOfSectionsInCollectionView:
    (UICollectionView *)collectionView
{
    return 100;
}
```

```

- (NSInteger)collectionView:(UICollectionView *)collectionView
  numberOfItemsInSection: (NSInteger)section
{
    return 1;
}

// This is a little utility that returns a view showing the
// section and item numbers for an index path
- (UIImageView *)viewForIndexPath:(NSIndexPath *)indexPath
{
    NSString *string = [NSString stringWithFormat:
        @"S%d(%d)", indexPath.section, indexPath.item];
    UIImage *image = blockStringImage(string, 16.0f);
    UIImageView *imageView =
        [[UIImageView alloc] initWithImage:image];
    return imageView;
}

// Return an initialized cell
- (UICollectionViewCell *)collectionView:
    (UICollectionView *)_collectionView
  cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
    UICollectionViewCell *cell = [self.collectionView
        dequeueReusableCellWithReuseIdentifier:@"cell"
        forIndexPath:indexPath];

    cell.backgroundColor = [UIColor whiteColor];
    cell.selectedBackgroundView =
        [[UIView alloc] initWithFrame:CGRectZero];
    cell.selectedBackgroundView.backgroundColor =
        [[UIColor blackColor] colorWithAlphaComponent:0.5f];

    // Show the section and item in a custom subview
    if ([cell viewWithTag:999])
        [[cell viewWithTag:999] removeFromSuperview];
    UIImageView *imageView = [self viewForIndexPath:indexPath];
    imageView.tag = 999;
    [cell.contentView addSubview:imageView];

    return cell;
}

#pragma mark Setup
- (instancetype)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self)
    {
        UICollectionViewFlowLayout *layout =

```

```

        [[UICollectionViewFlowLayout alloc] init];
        layout.scrollDirection =
            UICollectionViewScrollDirectionHorizontal;
        layout.sectionInset =
            UIEdgeInsetsMake(40.0f, 10.0f, 40.0f, 10.0f);
        layout.minimumLineSpacing = 10.0f;
        layout.minimumInteritemSpacing = 10.0f;
        layout.itemSize = CGSizeMake(100.0f, 100.0f);

        _collectionView = [[UICollectionView alloc]
            initWithFrame:CGRectZero collectionViewLayout:layout];
        _collectionView.backgroundColor = [UIColor darkGrayColor];
        _collectionView.allowsMultipleSelection = YES;
        _collectionView.dataSource = self;

        [_collectionView registerClass:[UICollectionViewCell
            class] forCellWithReuseIdentifier:@"cell"];
        [self addSubview:_collectionView];

        PREPCONSTRAINTS(_collectionView);
        CONSTRAINT(self, _collectionView,
            @"H:[_collectionView(>=0)]|");
        CONSTRAINT(self, _collectionView,
            @"V:-20-[_collectionView(>=0)]-20-|");
    }
    return self;}
@end

```

这条解决方案中的 `InsetCollectionView` 类提供了它自己的数据源，并把集合视图作为一项只读属性公布出来，令客户端能够提供相关的委托。图 10-6 演示了这条解决方案的效果，它能够制作出一张嵌入式的水平滚动集合视图。

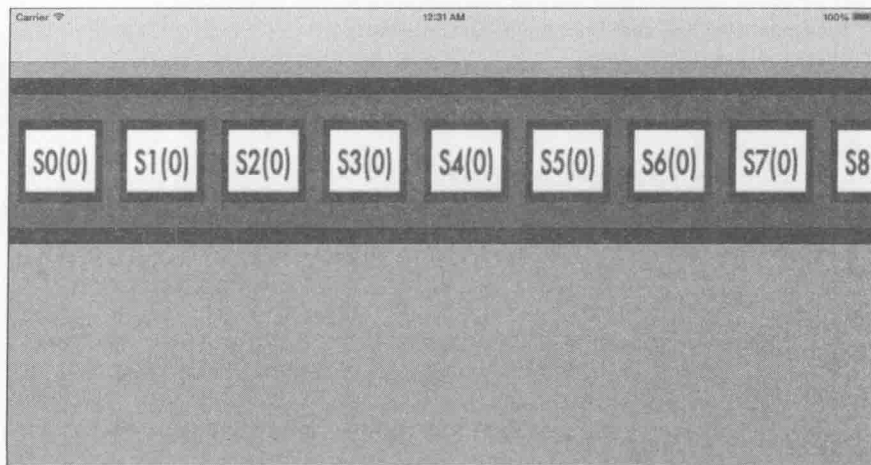


图 10-6 解决方案 10-3 创建了一种可嵌入其他视图中的水平滚动集合视图

本章稍后的解决方案 10-8 将会提供一种能够完全定制的 `UICollectionViewFlowLayout` 子类，它能够真正提供网格状布局。而解决方案 10-3 则提供了一种可以直接使用默认流式布局的便捷方案。此外，它还演示了在不使用内置控制器的环境下，应该如何创建集合视图。

## 10.7 解决方案：创建交互式的布局效果

流式布局是完全可控的。我们可以从 `UICollectionViewFlowLayout` 中继承子类，以便实时地控制每个条目的尺寸及其在屏幕上的摆放地点。而这对于开发者来说，是个相当强大的功能，它使得我们可以非常精准地指定每个条目的位置，从而模拟出三维布局效果，并且使我们能够突破线性模式，把行列形式转变成圆圈、堆叠、贝塞尔曲线等形式。

可以定制的布局属性包括标准的布局元素（`frame`、`center`、`size`）、透明度（`alpha` 和 `hidden`）、`z` 轴位置（`zIndex`），以及坐标变换方式（`transform3d`）。当布局对象查询这些属性的时候，我们可以像解决方案 10-4 那样给出调整后的值。

解决方案 10-4 交互式的布局效果

```
@interface PunchedLayout : UICollectionViewFlowLayout
@end

@implementation PunchedLayout
{
    CGSize boundsSize;
    CGFloat midX;
}

// Allow the presentation to resize as needed
- (BOOL)shouldInvalidateLayoutForBoundsChange:(CGRect)bounds
{
    return YES;
}

// Calculate the distance from the view center
- (void)prepareLayout
{
    [super prepareLayout];
    boundsSize = self.collectionView.bounds.size;
    midX = boundsSize.width / 2.0f;
}

// Lay out elements
- (NSArray *)layoutAttributesForElementsInRect:(CGRect)rect
{
    // Retrieve the default layout
    NSArray *array = [super layoutAttributesForElementsInRect:rect];
    for (UICollectionViewLayoutAttributes* attributes in array)
    {
```

```

attributes.transform3D = CATransform3DIdentity;
// Only handle layouts for visible items
if (!CGRectIntersectsRect(attributes.frame, rect)) continue;

CGPoint contentOffset = self.collectionView.contentOffset;
CGPoint itemCenter = CGPointMake(
    attributes.center.x - contentOffset.x,
    attributes.center.y - contentOffset.y);
CGFloat distance = ABS(midX - itemCenter.x);

// Normalize the distance and calculate the zoom factor
CGFloat normalized = distance / midX;
normalized = MIN(1.0f, normalized);
CGFloat zoom = cos(normalized * M_PI_4);
// Set the transform
attributes.transform3D =
    CATransform3DMakeScale(zoom, zoom, 1.0f);
}
return array;
}
@end

```

这条解决方案所创建的流式布局会根据条目距离屏幕中心的远近来缩放其尺寸，距离屏幕中心越近，尺寸越大，距离屏幕中心越远，尺寸越小。它会计算每个条目距离屏幕的水平中心有多远，然后根据余弦函数来决定缩放倍数（这样做的效果是：离屏幕中心越远，缩得就越小）。

图 10-7 演示了上述效果，不过读者最好能够自己运行解决方案 10-4，看一看各条目的尺寸实际改变了多少。

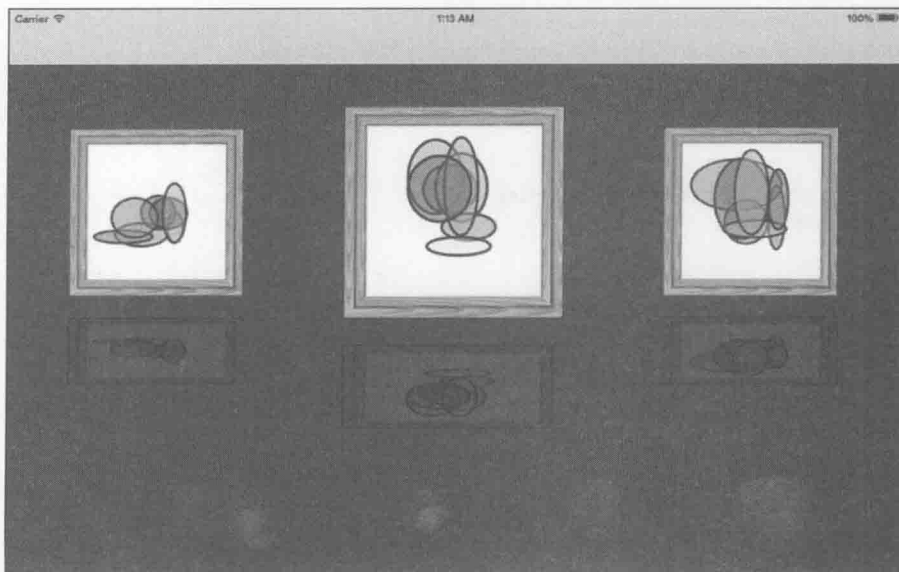


图 10-7 由解决方案 10-4 所定制的布局对象，会依照每个条目与屏幕水平中心之间的距离来缩放它们

## 10.8 解决方案：滚动之后自动调整位置

解决方案 10-4 使得用户能够更加关注位于屏幕中心的条目。那么，我们不妨把屏幕中心的物件自动调整到最合适的位置上。我们可以实现一个用于布局的方法，把集合视图的内容调整到特定的边界处。解决方案 10-5 演示了具体做法。

### 解决方案 10-5 定制 `targetContentOffsetForProposedContentOffset` 方法

```
- (CGPoint)targetContentOffsetForProposedContentOffset:
    (CGPoint)proposedContentOffset
    withScrollingVelocity:(CGPoint)velocity
{
    CGFloat offsetAdjustment = CGFLOAT_MAX;

    // Retrieve all onscreen items at the proposed starting point
    CGRect targetRect = CGRectMake(proposedContentOffset.x, 0.0,
        boundsSize.width, boundsSize.height);
    NSArray *array =
        [super layoutAttributesForElementsInRect:targetRect];

    // Determine the proposed center x-coordinate
    CGFloat proposedCenterX = proposedContentOffset.x + midX;

    // Search for the minimum offset adjustment
    for (UICollectionViewLayoutAttributes* layoutAttributes in array)
    {
        CGFloat distance =
            layoutAttributes.center.x - proposedCenterX;
        if (ABS(distance) < ABS(offsetAdjustment))
            offsetAdjustment = distance;
    }

    CGPoint desiredPoint =
        CGPointMake(proposedContentOffset.x + offsetAdjustment,
            proposedContentOffset.y);

    // Workaround for edge conditions. Hat tip, Nicolas Goles.
    if ((proposedContentOffset.x == 0) ||
        (proposedContentOffset.x >=
            (self.collectionViewContentSize.width -
                boundsSize.width)))
    {
        NSNotification *note = [NSNotification
            notificationWithName:@"PleaseRecenter" object:
                [NSValue valueWithCGPoint:desiredPoint]];
        // Notify view controller of modified desired point
        [[NSNotificationCenter defaultCenter]
            postNotification:note];
        return proposedContentOffset;
    }
}
```

```
// Offset the content by the minimal amount necessary to center  
return desiredPoint;  
}
```

用户在滚动集合视图时，系统会调用名为 `targetContentOffsetForProposedContentOffset:` 的方法，该方法描述了在不加人工干预的前提下集合视图会滚动到何处。覆写该方法的时候，我们遍历屏幕上的所有物件，找到距离视图水平中心最近的那个，然后调整该方法所要返回的偏移量，令该物件的中心与视图的中心相互重合。

## 10.9 解决方案：创建圆形布局

圆形布局是一种醒目的排版方式，它会将视图里的内容围绕着某个中心区域来排布，如图 10-8 所示。解决方案 10-6 在很大程度上参照了苹果公司的范例代码，那段范例代码是在 2012 年的 WWDC 上面首次公布的。这种布局方式很好地演示了如何在创建条目和删除条目时，把操作过程以动画形式展现出来。

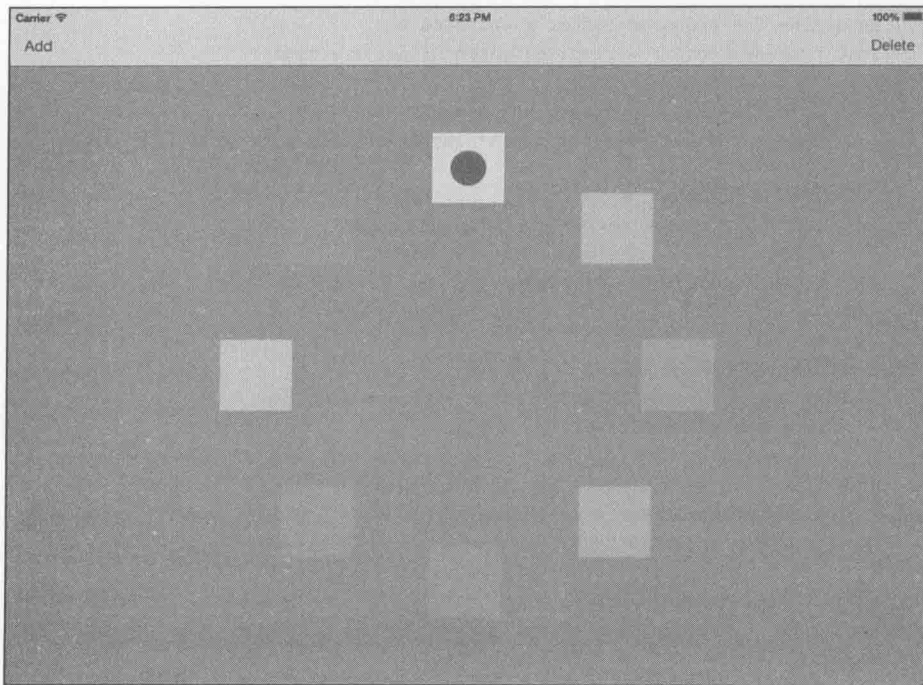


图 10-8 这种圆形的布局方式借鉴了苹果公司的范例代码，开发者 Greg Hartstein 对此亦有贡献

解决方案 10-6 所采用的布局对象，通过 `collectionViewContentSize` 方法把视图内容的尺寸设为固定值。由于它明确地创建了一块固定不变的排版区域，所以集合视图不会再滚动了。范例代码还会在 `prepareLayout` 方法中进行计算，以便进一步向内缩小排版区



域。屏幕高度与屏幕宽度之中较小的值，决定了圆的半径。无论屏幕方向如何改变，圆的半径总保持不变。

#### 解决方案 10-6 将集合视图中的各条目排列成圆形

```
@implementation CircleLayout
{
    NSInteger numberOfItems;
    CGPoint centerPoint;
    CGFloat radius;

    NSMutableArray *insertedIndexPaths;
    NSMutableArray *deletedIndexPaths;
}

// Calculate and save off the current state
- (void)prepareLayout
{
    [super prepareLayout];
    CGSize size = self.collectionView.frame.size;
    numberOfItems =
        [self.collectionView numberOfItemsInSection:0];
    centerPoint =
        CGPointMake(size.width / 2.0f, size.height / 2.0f);
    radius = MIN(size.width, size.height) / 3.0f;
    insertedIndexPaths = [NSMutableArray array];
    deletedIndexPaths = [NSMutableArray array];
}

// Fix the content size to the frame size
- (CGSize)collectionViewContentSize
{
    return self.collectionView.frame.size;
}

// Calculate position for each item
- (UICollectionViewLayoutAttributes *)
layoutAttributesForItemAtIndexPath:(NSIndexPath *)path
{
    UICollectionViewLayoutAttributes *attributes =
        [UICollectionViewLayoutAttributes
         layoutAttributesForCellWithIndexPath:path];
    CGFloat progress = (float) path.item / (float) numberOfItems;
    CGFloat theta = 2.0f * M_PI * progress;
    CGFloat xPosition = centerPoint.x + radius * cos(theta);
    CGFloat yPosition = centerPoint.y + radius * sin(theta);
    attributes.size = [self itemSize];
    attributes.center = CGPointMake(xPosition, yPosition);
    return attributes;
}
```

```

// Calculate layouts for all items
- (NSArray *)layoutAttributesForElementsInRect:(CGRect)rect
{
    NSMutableArray *attributes = [NSMutableArray array];
    for (NSInteger index = 0; index < numberOfItems; index++)
    {
        NSIndexPath *indexPath =
            [NSIndexPath indexPathForItem:index inSection:0];
        [attributes addObject:
            [self layoutAttributesForItemAtIndex:indexPath:indexPath]];
    }
    return attributes;
}

// Build insertion and deletion collections from updates
- (void)prepareForCollectionViewUpdates:(NSArray *)updates
{
    [super prepareForCollectionViewUpdates:updates];

    for (UICollectionViewUpdateItem* updateItem in updates)
    {
        if (updateItem.updateAction ==
            UICollectionViewUpdateActionInsert)
        {
            [insertedIndexPaths
                addObject:updateItem.indexPathAfterUpdate];
        }
        else if (updateItem.updateAction ==
            UICollectionViewUpdateActionDelete)
        {
            [deletedIndexPaths
                addObject:updateItem.indexPathBeforeUpdate];
        }
    }
}

// Establish starting attributes for added item
- (UICollectionViewLayoutAttributes *)
insertionAttributesForItemAtIndex:(NSIndexPath *)itemIndexPath
{
    UICollectionViewLayoutAttributes *attributes =
        [self layoutAttributesForItemAtIndex:itemIndexPath];
    attributes.alpha = 0.0;
    attributes.center = centerPoint;
    return attributes;
}

// Establish final attributes for deleted item
- (UICollectionViewLayoutAttributes *)
deletionAttributesForItemAtIndex:(NSIndexPath *)itemIndexPath
{
    UICollectionViewLayoutAttributes *attributes =
        [self layoutAttributesForItemAtIndex:itemIndexPath];
    attributes.alpha = 0.0;
}

```

```

        attributes.center = centerPoint;
        attributes.transform3D = CATransform3DMakeScale(0.1, 0.1, 1.0);
        return attributes;
    }

    // Handle insertion animation for all items
    - (UICollectionViewLayoutAttributes*)
        initialLayoutAttributesForAppearingItemAtIndexPath:
            (NSIndexPath*)indexPath
    {
        return [insertedIndexPaths containsObject:indexPath] ?
            [self insertionAttributesForItemAtIndexPath:indexPath] :
            [super initialLayoutAttributesForAppearingItemAtIndexPath:
                indexPath];
    }

    // Handle deletion animation for all items
    - (UICollectionViewLayoutAttributes*)
        finalLayoutAttributesForDisappearingItemAtIndexPath:
            (NSIndexPath*)indexPath
    {
        return [deletedIndexPaths containsObject:indexPath] ?
            [self deletionAttributesForItemAtIndexPath:indexPath] :
            [super finalLayoutAttributesForDisappearingItemAtIndexPath:
                indexPath];
    }
}
@end

```

布局对象会根据每个条目的索引路径来计算它的位置。本例所采用的布局方式只使用一个区段，每个条目在该区段内的顺序（比方说，它是第三个条目还是第五个条目）决定了它在圆周上的位置：

```

CGFloat progress = (float) path.item / (float) numberOfItems;
CGFloat theta = 2.0f * M_PI * progress;

```

上述计算方式也适用于其他图形或其他形式的索引路径，只要各条目在索引路径中的位置都能调整到  $[0.0, 1.0]$  这个范围内就行。对于圆形来说，此范围可以和从  $0$  至  $2\pi$  的弧度对应起来。而对于螺旋形的布局来说，此范围则可以和从  $0$  到  $3\pi$ 、 $4\pi$  乃至  $5\pi$  的弧度对应起来。如果想按照贝塞尔曲线来布局，那么开发者需要遍历能够决定曲线形状的各个控制点，并且要根据情况在其中插入其他的点。

### 10.9.1 实现创建条目与删除条目时的动画效果

在解决方案 10-6 里面有几个方法值得关注，它们分别指定了新插入的条目所应具备的初始属性，以及刚删除的条目所应具备的最后属性。这些属性使得集合视图在添加新条目及删除现有条目的时候，能够以动画效果来表示该操作执行前后的布局变化过程。

本条解决方案的动画效果与苹果公司的原始范例代码一样：新添加的条目一开始会以全透明的形态出现在圆圈正中，然后会移动到它应有的位置上，在移动过程中它将逐渐淡入。

而刚删除的条目则会从目前的位置移向圆心，并在此过程中逐渐缩小、淡出。运行范例代码的时候就能看到这些效果了。

开发文档中的 `initialLayoutAttributesForAppearingItemAtIndexPath` 及 `finalLayoutAttributesForDisappearingItemAtIndexPath` 方法，名字起得很令人困惑，从表面上看，这些方法似乎只会针对刚插入或删除的条目来调用。但实际上，系统会向每一个条目询问它的起始属性（starting attribute）和最终属性（ending attribute），而不仅仅向刚添加或删除的条目询问。所以，解决方案 10-6 在添加条目和删除条目的时候，会把所添条目及所删条目的索引路径分别记录到两个数组里面。这样的话，我们就可以只针对当前要添加或删除的条目来定制其属性了。

该机制所提供的这种方式，能够把视图中全部条目的布局属性都以动画形式表现出来，使得开发者可以按照需要添加额外的动画效果。比方说，如果有新的条目插入第 3 行，那么该行末尾的条目就应该移动到第 4 行开头。在默认的情况下，末尾的条目会按照斜线方向移动到下一行开头，但有了这套方式之后，我们就可以把第 3 行末尾的单元格先向右移出屏幕，然后再将其从第 4 行左侧移入屏幕。

## 10.9.2 增强圆形布局的实用性

本条解决方案对苹果公司原有的范例代码做了几处修改。其中之一就是：解决方案 10-6 使用了导航栏上的 Add 及 Delete 按钮，而没有使用手势。另外一处修改是：我们把每个小视图都设为不同颜色，从而令用户可以把它们区分开。解决方案 10-6 提供了选择功能。用户可以选定某个条目。然后可以把该条目删掉，也可以把新的条目添加到它后面。

下面这段删除代码会找到当前选定的条目将其删除，并选中下一个条目。然后，它会根据屏幕上现有条目的数量来启用或禁用 Add 及 Delete 按钮：

```
(void)delete
{
    if (!count) return;

    // Decrement the number of onscreen items
    count--;

    // Determine which item to delete
    NSArray *selectedItems =
        [self.collectionView indexPathsForSelectedItems];
    NSInteger itemNumber = selectedItems.count ?
        ((NSIndexPath *)selectedItems[0]).item : 0;

    NSIndexPath *itemPath =
        [NSIndexPath indexPathForItem:itemNumber inSection:0];

    // Perform deletion
    [self.collectionView performBatchUpdates:^(
        [self.collectionView deleteItemsAtIndexPaths:@[itemPath]];
    ) completion:^(BOOL done){
```

```

    if (count)
    {
        [self.collectionView selectItemAtIndexPath:
            [NSIndexPath indexPathForItem:
                MAX(0, itemNumber - 1) inSection:0]
            animated:NO
            scrollPosition:UICollectionViewScrollPositionNone];
        self.navigationItem.rightBarButtonItem.enabled =
            (count > 0);
        self.navigationItem.leftBarButtonItem.enabled =
            (count < (IS_IPAD ? 20 : 8));
    }
}

```

在真实的应用程序中，很少需要添加或删除一些彼此之间没有区别的视图，不过，有的时候却需要添加或删除一些含义各不相同的视图。因此，本例所做的改动使得这个范例变得更加实用了，读者可以以此为起点，根据应用程序的实际需要来扩充这条解决方案。

### 10.9.3 布局对象

图 10-8 演示了由解决方案 10-6 所构建的布局效果。当用户添加一些新条目进来之后，圆周上的条目变得拥挤了，在 iPad 上面最多可以有 20 个条目，而在 iPhone 上面最多则可以有 8 个条目。读者很容易在 add 及 delete 方法中修改这些上限值，使其与自己的应用程序相符。

## 10.10 解决方案：用手势调整布局

解决方案 10-7 是基于解决方案 10-6 而构建的，它添加了交互性的手势功能，使得用户可以通过手势来调整布局。该解决方案使用了两个手势识别器：一个识别双指缩放（pinch）手势，另一个识别旋转（rotate）手势。视图中的条目可以同时识别这两种手势，所以用户可以同时对其进行缩放与旋转。

解决方案 10-7 使用户可以通过手势来调整集合视图的布局

```

// Intermediate rotation
- (void)rotateBy:(CGFloat)theta
{
    currentRotation = theta;
}

// Final rotation
- (void)rotateTo:(CGFloat)theta
{
    rotation += theta;
    currentRotation = 0.0f;
}

// Scaling
- (void)scaleTo:(CGFloat)factor
{

```

```

        scale = factor;
    }

    // Calculate position for each item
    - (UICollectionViewLayoutAttributes *)
        layoutAttributesForItemAtIndexPath:(NSIndexPath *)path
    {
        UICollectionViewLayoutAttributes *attributes =
            [UICollectionViewLayoutAttributes
             layoutAttributesForCellWithIndexPath:path];
        CGFloat progress = (float) path.item / (float) numberOfItems;
        CGFloat theta = 2.0f * M_PI * progress;

        // Update the scaling and rotation to match the current gesture
        CGFloat scaledRadius = MIN(MAX(scale, 0.5f), 1.3f) * radius;
        CGFloat rotatedTheta = theta + rotation + currentRotation;
        // Calculate the new positions
        CGFloat xPosition =
            centerPoint.x + scaledRadius * cos(rotatedTheta);
        CGFloat yPosition =
            centerPoint.y + scaledRadius * sin(rotatedTheta);
        attributes.size = [self itemSize];
        attributes.center = CGPointMake(xPosition, yPosition);
        return attributes;
    }
    @end

```

处理旋转手势要比处理双指缩放手势稍微复杂一点。双指缩放手势会直接给出当前的缩放倍数，而旋转手势给出的则是当前角度与尚未开始旋转时的角度差。由于它给出的不是相邻两次旋转手势之间的差值，所以在持续执行旋转手势时，我们要根据当前角度与旋转之前的角度差来进行旋转。于是，解决方案 10-7 会在回调方法中分别处理两种状况。如果旋转操作正在持续进行，那就根据这次旋转的量来更新当前的旋转角度<sup>①</sup>；如果旋转操作即将结束，那就把当前的旋转角度置为 0，这样的话，下次旋转时就能从新的基准值<sup>②</sup>开始计算了：

```

- (void)pinch:(UIPinchGestureRecognizer *)pinchRecognizer
{
    CircleLayout *layout =
        (CircleLayout *)self.collectionView.collectionViewLayout;
    [layout scaleTo:pinchRecognizer.scale];
    [layout invalidateLayout];
}

- (void)rotate:(UIRotationGestureRecognizer *)rotationRecognizer
{
    CircleLayout *layout =
        (CircleLayout *)self.collectionView.collectionViewLayout;

```

① 相当于 CircleLayout 中的 currentRotation 变量。——译者注

② 相当于 CircleLayout 中的 rotation 变量，也就后文所说的“起始角”。——译者注

```

    if (rotationRecognizer.state == UIGestureRecognizerStateEnded)
    {
        [layout rotateTo:rotationRecognizer.rotation];
    }
    else
    {
        [layout rotateBy:rotationRecognizer.rotation];
        [layout invalidateLayout];
    }
}

```

请注意，上面两个方法都会调用 `invalidateLayout`，以迫使布局对象实时更新视图的布局。由于这条解决方案要进行大量的图形计算，所以最好是在真机上面进行测试。

——解决方案 10-7 会根据用户的手势来调整视图内容的半径（最小可以缩小到原来的一半，最大可以放大到原来的 1.3 倍）和起始角（start angle），以便修改布局效果。起始角的初始值是 0 度，但每次执行完旋转操作之后，它的值都会更新。程序会根据缩放后的半径以及调整后的角度来重新排布集合视图。

## 10.11 解决方案：创建真正的网格状布局

默认的流式布局会自动换行，以便使区段中的条目能够适应集合视图的长度或宽度，但这样做出来的视图只能在一个方向上滚动。如果愿意多做一些数学运算，就可以编写自定义的布局子类，从而实现不会换行的双向滚动视图。实现该功能所需的运算量比较大，而且不是特别容易。图 10-9 演示了这种布局。

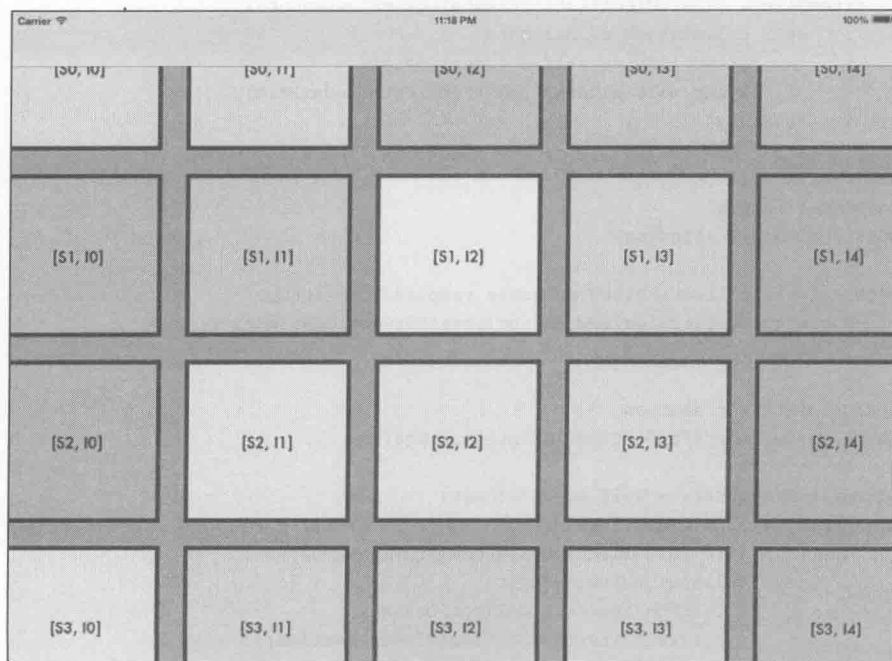


图 10-9 自定义的网格状布局，使得用户能够在水平方向和垂直方向上滚动集合视图

解决方案 10-8 完全定制了 UICollectionViewFlowLayout 的子类，覆写了 collectionViewContentSize 及 layoutAttributesForItemAtIndexPath: 方法，以便手工摆放每个条目。这种实现方式完全考虑到所有与间隔有关的请求以及回调方法。相比之下，普通的流式布局只会在满足多个最小值的前提下，试着把条目合适地摆放到屏幕中。而我们的布局子类，则会根据这些限定值，精确地调整视图底层的内容大小，令其与开发者所指定的尺寸完全匹配。

#### 解决方案 10-8 自己编写网格状的布局类

```
@implementation GridLayout

#pragma mark Items
// Does a delegate provide individual sizing?
- (BOOL)usesIndividualItemSizing
{
    return [self.collectionView.delegate respondsToSelector:
        @selector(collectionView:layout:sizeForItemAtIndexPath:)];
}

// Return cell size for an item
- (CGSize)sizeForItemAtIndexPath:(NSIndexPath *)indexPath
{
    CGSize itemSize = self.itemSize;
    if ([self usesIndividualItemSizing])
        itemSize = [(id <UICollectionViewDelegateFlowLayout>)
            self.collectionView.delegate
            collectionView:self.collectionView
            layout:self sizeForItemAtIndexPath:indexPath];
    return itemSize;
}

#pragma mark Insets
// Individual insets?
- (BOOL)usesIndividualInsets
{
    return [self.collectionView.delegate respondsToSelector:
        @selector(collectionView:layout:insetForSectionAtIndex:)];
}

// Return insets for section
- (UIEdgeInsets)insetsForSection:(NSInteger)section
{
    UIEdgeInsets insets = self.sectionInset;
    if ([self usesIndividualInsets])
        insets = [(id <UICollectionViewDelegateFlowLayout>)
            self.collectionView.delegate
            collectionView:self.collectionView
            layout:self insetForSectionAtIndex:section];
    return insets;
}
```



```

#pragma mark Item Spacing
// Individual item spacing?
- (BOOL)usesIndividualItemSpacing
{
    return [self.collectionView.delegate respondsToSelector:
        @selector(collectionView:layout:
            minimumInteritemSpacingForSectionAtIndex:)];
}

// Return spacing for section
- (CGFloat)itemSpacingForSection:(NSInteger)section
{
    CGFloat spacing = self.minimumInteritemSpacing;
    if ([self usesIndividualItemSpacing])
        spacing = [(id <UICollectionViewDelegateFlowLayout>)
            self.collectionView.delegate
            collectionView:self.collectionView
            layout:self
            minimumInteritemSpacingForSectionAtIndex:section];
    return spacing;
}

#pragma mark Layout Geometry
// Find the tallest subview
- (CGFloat)maxItemHeightForSection:(NSInteger)section
{
    CGFloat maxHeight = 0.0f;
    NSInteger numberOfItems =
        [self.collectionView numberOfItemsInSection:section];
    for (int i = 0; i < numberOfItems; i++)
    {
        NSIndexPath *indexPath = [NSIndexPath indexPathWithItemIndex:i];
        CGSize itemSize = [self sizeForItemAtIndexPath:indexPath];
        maxHeight = MAX(maxHeight, itemSize.height);
    }
    return maxHeight;
}

// "Horizontal" row-based extent from the start of the section to its end
- (CGFloat)fullWidthForSection:(NSInteger)section
{
    UIEdgeInsets insets = [self insetsForSection:section];
    CGFloat horizontalInsetExtent = insets.left + insets.right;
    CGFloat collectiveWidth = horizontalInsetExtent;

    NSInteger numberOfItems =
        [self.collectionView numberOfItemsInSection:section];
    for (int i = 0; i < numberOfItems; i++)
    {
        NSIndexPath *indexPath = [NSIndexPath indexPathWithItemIndex:i];
        CGSize itemSize = [self sizeForItemAtIndexPath:indexPath];
    }
}

```

```

        collectiveWidth += itemSize.width;
        collectiveWidth += [self itemSpacingForSection:section];
    }

    // Take back one spacer, n-1 fence post
    collectiveWidth -= [self itemSpacingForSection:section];

    return collectiveWidth;
}

// Bounding size for each section
- (CGSize)fullSizeForSection: (NSInteger)section
{
    CGFloat headerExtent = (self.scrollDirection ==
        UICollectionViewScrollDirectionHorizontal) ?
        self.headerReferenceSize.width :
        self.headerReferenceSize.height;
    CGFloat footerExtent = (self.scrollDirection ==
        UICollectionViewScrollDirectionHorizontal) ?
        self.footerReferenceSize.width :
        self.footerReferenceSize.height;

    UIEdgeInsets insets = [self insetsForSection:section];
    CGFloat verticalInsetExtent = insets.top + insets.bottom;
    CGFloat maxHeight = [self maxItemHeightForSection:section];

    CGFloat fullHeight = headerExtent + footerExtent +
        verticalInsetExtent + maxHeight;
    CGFloat fullWidth = [self fullWidthForSection:section];

    return CGSizeMake(fullWidth, fullHeight);
}

// How far is each item offset within the section
- (CGFloat)horizontalInsetForItemAtIndexPath: (NSIndexPath *)indexPath
{
    UIEdgeInsets insets = [self insetsForSection:indexPath.section];
    CGFloat horizontalOffset = insets.left;
    for (int i = 0; i < indexPath.item; i++)
    {
        CGSize itemSize = [self sizeForItemAtIndexPath:
            IndexPath(indexPath.section, i)];
        horizontalOffset += (itemSize.width +
            [self itemSpacingForSection:indexPath.section]);
    }
    return horizontalOffset;
}

// How far is each item down
- (CGFloat)verticalInsetForItemAtIndexPath: (NSIndexPath *)indexPath
{

```

```

CGSize thisItemSize = [self sizeForItemAtIndexPath:indexPath];
CGFloat verticalOffset = 0.0f;

// Previous sections
for (int i = 0; i < indexPath.section; i++)
    verticalOffset += [self fullSizeForSection:i].height;

// Header
CGFloat headerExtent = (self.scrollDirection ==
    UICollectionViewScrollDirectionHorizontal) ?
    self.headerReferenceSize.width :
    self.headerReferenceSize.height;
verticalOffset += headerExtent;

// Top inset
UIEdgeInsets insets = [self insetsForSection:indexPath.section];
verticalOffset += insets.top;

// Vertical centering
CGFloat maxHeight =
    [self maxItemHeightForSection:indexPath.section];
CGFloat fullHeight = (maxHeight - thisItemSize.height);
CGFloat midHeight = fullHeight / 2.0f;

switch (self.alignment)
{
    case GridRowAlignmentNone:
        break;
    case GridRowAlignmentTop:
        break;
    case GridRowAlignmentCenter:
        verticalOffset += midHeight;
        break;
    case GridRowAlignmentBottom:
        verticalOffset += fullHeight;
        break;
    default:
        break;
}

return verticalOffset;
}

#pragma mark Layout Attributes
// Provide per-item placement
- (UICollectionViewLayoutAttributes *)layoutAttributesForItemAtIndexPath:
    (NSIndexPath *)indexPath
{
    UICollectionViewLayoutAttributes *attributes =
        [UICollectionViewLayoutAttributes
            layoutAttributesForCellWithIndexPath:indexPath];
    CGSize thisItemSize = [self sizeForItemAtIndexPath:indexPath];

```

```

CGFloat verticalOffset =
    [self verticalInsetForItemAtIndexPath:indexPath];
CGFloat horizontalOffset =
    [self horizontalInsetForItemAtIndexPath:indexPath];

if (self.scrollDirection == UICollectionViewScrollDirectionVertical)
    attributes.frame = CGRectMake(horizontalOffset,
        verticalOffset, thisItemSize.width, thisItemSize.height);
else
    attributes.frame = CGRectMake(verticalOffset,
        horizontalOffset, thisItemSize.width,
        thisItemSize.height);

return attributes;
}

// Return full extent
- (CGSize)collectionViewContentSize
{
    NSInteger sections = self.collectionView.numberOfSections;

    CGFloat maxWidth = 0.0f;
    CGFloat collectiveHeight = 0.0f;

    for (int i = 0; i < sections; i++)
    {
        CGSize sectionSize = [self fullSizeForSection:i];
        collectiveHeight += sectionSize.height;
        maxWidth = MAX(maxWidth, sectionSize.width);
    }

    if (self.scrollDirection ==
        UICollectionViewScrollDirectionVertical)
        return CGSizeMake(maxWidth, collectiveHeight);
    else
        return CGSizeMake(collectiveHeight, maxWidth);
}

// Provide grid layout attributes
- (NSArray *)layoutAttributesForElementsInRect:(CGRect)rect
{
    NSMutableArray *attributes = [NSMutableArray array];
    for (NSInteger section = 0;
        section < self.collectionView.numberOfSections; section++)
        for (NSInteger item = 0;
            item < [self.collectionView
                numberOfItemsInSection:section];
            item++)
        {
            UICollectionViewLayoutAttributes *layout =
                [self layoutAttributesForItemAtIndexPath:
                    INDEXPATH(section, item)];

```

```

        [attributes addObject:layout];
    }
    return attributes;
}

- (BOOL) shouldInvalidateLayoutForBoundsChange:(CGRect) oldBounds
{
    return YES;
}

@end

```

解决方案 10-8 中的布局对象要把每个元素的排布位置都计算出来。但是它不会使用行间距属性，因为该属性只在自动换行的前提下才有意义。我们所制作的网格状布局，绝不会自动换行，所以本条解决方案会彻底忽略该属性。

本条解决方案的布局对象里，还有个名为 `alignment` 的属性。这个新属性决定了表格中的每一行是顶端对齐、底端对齐，还是居中对齐。它会查询一整行的总体高度，然后可能会把比这个高度矮的条目移动一段距离。

解决方案 10-8 中包含了这个子类的所有代码，读者可以看到：想实现这样一种完全定制的 `UICollectionViewFlowLayout` 子类需要编写多少代码。编程技巧当然还是隐藏在细节之中。我们应该尽量用各种物件来彻底地测试这个 `GridLayout`。

## 10.12 解决方案：为集合视图中的条目添加自定义菜单

当用户对某个条目执行标准的“长按”手势时，集合视图里面可以实现出图 10-10 这样的菜单。该菜单默认会提供剪切、复制及粘贴操作。不过开发者也可以把这些操作换成自己定义的其他操作，从而构建出下面的菜单。

菜单功能要通过 `UICollectionViewDelegate` 协议中的下面三个方法来实现：

- **`collectionView:shouldShowMenuForItemAtIndexPath:`**——该方法用来决定在给定的索引路径处是否应该显示菜单。
- **`collectionView:canPerformAction:forItemAtIndexPath:withSender:`**——该方法用来确认委托是否能在位于索引路径处的条目上面执行给定的操作。这个委托方法可以用来去掉我们不想要的默认操作，比如剪切、复制及粘贴。
- **`collectionView:performAction:forItemAtIndexPath:withSender:`**——该方法会命令委托在位于索引路径处的条目上面执行给定的操作。

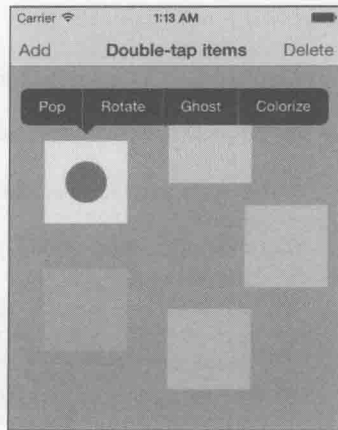


图 10-10 若想为每个条目实现自定义菜单，则需令单单元格成为第一响应者

除了要在前两个方法里返回 YES，并且在第三个方法里处理相关操作之外，我们还必须令集合视图能够成为第一响应者：

```
- (BOOL)canBecomeFirstResponder
{
    return YES;
}
```

满足上述所有要求之后，当用户按住某个条目不放时，集合视图里面就会出现菜单了。

## 使用户能够以双击的方式弹出菜单

由 UICollectionView 所提供的菜单，需要以长按不放的手势来激活，而解决方案 10-9 则创建了自定义的单元格类，并且添加了针对双击手势的识别器。如果侦测到双击手势，那么回调方法就会把相关单元格设为第一响应者，并在它上面展示出标准的菜单。

解决方案 10-9 在集合视图的单元格上实现自定义的菜单

```
- (BOOL)canBecomeFirstResponder
{
    return YES;
}

- (BOOL)canPerformAction:(SEL)action withSender:(id)sender
{
    if (action == @selector(ghostSelf)) return YES;
    if (action == @selector(popSelf)) return YES;
    if (action == @selector(rotateSelf)) return YES;
    if (action == @selector(colorize)) return YES;
    return NO;
}

- (void)tapped:(UIGestureRecognizer *)uigr
{
    if (uigr.state != UIGestureRecognizerStateRecognized) return;

    [[UIMenuController sharedMenuController] setMenuVisible:NO
    animated:YES];
    [self becomeFirstResponder];

    UIMenuController *menu = [UIMenuController sharedMenuController];
    UIMenuItem *pop = [[UIMenuItem alloc]
        initWithTitle:@"Pop" action:@selector(popSelf)];
    UIMenuItem *rotate = [[UIMenuItem alloc]
        initWithTitle:@"Rotate" action:@selector(rotateSelf)];
    UIMenuItem *ghost = [[UIMenuItem alloc]
        initWithTitle:@"Ghost" action:@selector(ghostSelf)];
    UIMenuItem *colorize = [[UIMenuItem alloc]
        initWithTitle:@"Colorize" action:@selector(colorize)];

    [menu setMenuItems:@[pop, rotate, ghost, colorize]];
```

```

[menu update];
[menu setTargetRect:self.bounds inView:self];
[menu setMenuVisible:YES animated:YES];
}

```

解决方案 10-9 列出了相关的细节代码。这个 `UICollectionViewCell` 的子类宣称自己可以成为第一响应者，这是要在它上面显示菜单的先决条件。该类设置了它想要显示的菜单项，并实现了 `canPerformAction:withSender:` 方法，使得这些菜单项能够显示出来。图 10-10 中的菜单就是用这条解决方案创建出来的。

## 10.13 小结

本章介绍了集合视图以及功能非常强大的流式布局 (`UICollectionViewFlowLayout`)。读者学到了如何创建简单的集合视图控制器，以及如何创建独立的视图。我们讨论了怎样设置布局对象的关键属性。此外，还讲解了怎样创建生动的反馈效果，以及怎样以动画形式来表现插入条目及删除条目的过程。阅读下一章之前，请回顾下面几个问题：

- 集合视图提供了许多功能，开发者无须编写大量代码，即可使用它们。集合视图所提供的 API 要比表格视图强大得多，这些功能如果改用表格来实现，需要编写很多代码，而且有些功能甚至不太可能用表格来实现。
- 本章只是粗略地提了一下头部视图和尾部视图，而且根本没有用到装饰视图。本章的范例代码详细体现了创建自定义的补充视图类时所应注意的事项。
- 集合视图在排布其中的条目时，可以带有丰富的变换效果。我们可以试着在界面中以动画来响应用户的操作。但是，不要盲目地添加动画效果，而是应该把握住分寸。
- 本章通过指定各条目刚添加进来时的初始属性，以及删除完毕之后的最终属性来实现添加及删除时的动画效果，而这对于补充视图来说也同样适用。利用这一特性，我们可以用动画效果来表示将要出现在集合视图里面的新区段，以及将要离开集合视图中的区段。
- 与动画效果类似，手势也应该设计得明智一些。如果用户不太能够发现自己可以通过长按或三连击手势来添加或删除条目，就不要设计这样的手势了。我们可以改用弹出式窗口、菜单、浮动的文字或是简单的按钮来告诉用户应该如何管理并改变视图中的条目。
- 研究 `UICollectionViewFlowLayout` 类的时候，不要单单依赖于该类的文档，而是应该看看它的抽象基类 `UICollectionViewLayout`。这个类描述了开发者所要定义的每一个关键方法。
- 最后要注意，我们应该在设备上面测试应用程序。当程序使用布局对象时，尤其是使用频繁更新的布局或带有变换效果的布局时，它的性能就无法准确地反映到模拟器之中了。此时我们应该在设备上面进行测试，并通过 Instruments 来分析程序的性能，看看程序的布局是不是设计得太过复杂。

## 分享文档与数据

在 iOS 系统中，应用程序之间可以共享信息及数据，也可以通过许多系统特性，把控制权转移给另一个程序。每个程序都能够访问系统共用的剪贴板，这使得不同的程序之间可以复制并粘贴数据。应用程序可以请求系统在某份文档上面执行由系统所提供的一些操作，比方说打印、发布 Twitter 消息，或是张贴到 Facebook 等。应用程序可以声明自定义的 URL 方案，这些 URL 方案能够嵌入文档及网页之中。本章将会介绍如何在应用程序之间分享文档及数据。读者将会学到怎样把这些特性添加到自己的应用程序中，以及怎样巧妙地利用这些特性，来与 iOS 系统中的其他程序相互配合。

### 11.1 解决方案：使用统一类型标识符

统一类型标识符（Uniform Type Identifier，简称 UTI）是 iOS 系统在分享信息时所使用的中心组件，可以将它们看成是新一代的 MIME 类型。UTI 是一种字符串，能够表示诸如图像及文本等资源类型。UTI 指明了程序之间将要共用的数据对象是何类型。它们并不依赖于原有的各种指示符，比如文件扩展名、MIME 类型或是 OSType 等与文件类型有关的元数据。UTI 用一种更新颖、更灵活的技术取代了原有的那些技术。

UTI 的命名遵循反向域名样式（reverse-domain-style）。由苹果公司所定义的那些常用标识符，遵循 public.html 及 public.jpeg 这样的格式。前者表示 HTML 源文本，后者表示 JPEG 图像，这两种 UTI 都对应于特定类型的信息。

继承在 UTI 中扮演了重要角色。UTI 使用了与面向对象类似的继承体系，其中的子 UTI 与上级 UTI 之间有 is-a（是一种）的关系。子 UTI 继承了上级 UTI 的全部属性，此外还添加了一些更为具体的属性，用以体现它所代表的特定信息类型。之所以要这样设计，是因



为对于每个 UTI 来说,既有比它更通用的 UTI,又有比它更具体的 UTI。我们以表示 JPEG 图像的 UTI 为例来说明。JPEG 图像 (public.jpeg) 是一种图像 (public.image), 而图像又是一种数据 (public.data)。数据是一种用户可以看到或听到的内容 (public.content), 而内容又表示了一个条目 (public.item)。public.item 是 UTI 体系中的通用基础类型。整套体系叫作遵循体系, 其中的子 UTI “遵循” 它的上级 UTI。比方说, 更为具体的 jpeg 型 UTI 就遵循更加通用的 image 型 UTI 或 data 型 UTI。

图 11-1 列出了苹果公司的基本遵循树。树中位置较低的 UTI, 必须遵循其上级 UTI 的全部数据属性。如果声明了某个上级 UTI, 那就表示要支持它的全部子 UTI。比方说, 如果某程序宣称自己能够打开 public.data 类型的数据, 那它就必须能处理文本、电影以及图像文件等内容才行。

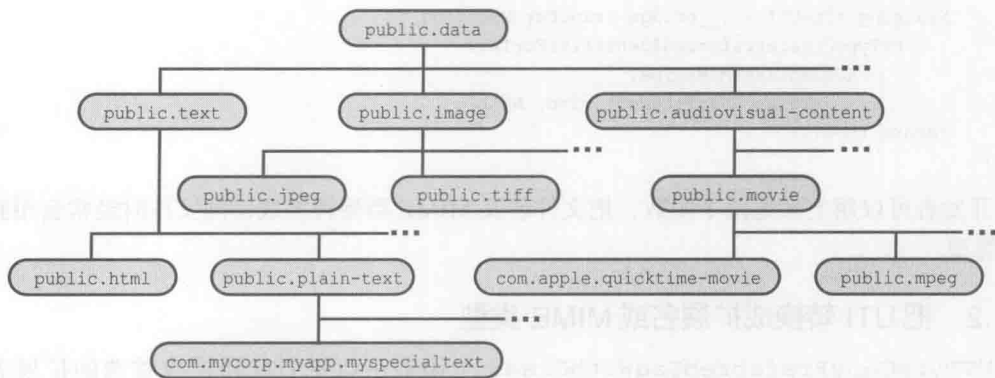


图 11-1 苹果公司的基本遵循树

UTI 可以多重继承。某个条目可以遵循不止一个上级 UTI。所以, 我们可以宣称它同时遵循 public.text 及 public.image, 这样就能指定一种既能容纳文本又能容纳图像的数据类型了。

虽说每个 UTI 都应该遵循命名规范, 但却没有统一的 UTI 注册机构。public 域用来表示 iOS 专用的类型, 大部分应用程序都可以共用这些类型。苹果公司生成了一份由 public UTI 所构成的完整体系图表。第三方厂商专用的 UTI, 应该按照标准的反向域名样式来命名 (例如 com.sadun.myCustomType 及 com.apple.quicktime-movie)。

### 11.1.1 根据文件扩展名来决定 UTI

Mobile Core Services 模块提供了一些工具, 可以根据文件扩展名来获取 UTI 信息。在使用这些基于 C 语言的函数之前, 必须先引入相关模块。下面这个函数会根据传入的 ext 字符串参数来返回首选的 UTI。返回的这个 UTI 是一个代表标识符的字符串:

```

#import MobileCoreServices;

NSString *preferredUTIForExtension(NSString *ext)
{
    // Request the UTI for the file extension

```

```

NSString *theUTI = (__bridge_transfer NSString *)
    UTTypeCreatePreferredIdentifierForTag(
        kUTTagClassFilenameExtension,
        (__bridge CFStringRef) ext, NULL);
return theUTI;
}

```

开发者也可以不传入文件扩展名,而是传入 MIME 类型,此时 `UTTypeCreatePreferredIdentifierForTag()` 的第一个参数应该改为 `kUTTagClassMIMETYPE`。下面这个函数可以根据给定的 MIME 类型返回首选的 UTI:

```

NSString *preferredUTIForMIMETYPE(NSString *mime)
{
    // Request the UTI for the MIME type
    NSString *theUTI = (__bridge_transfer NSString *)
        UTTypeCreatePreferredIdentifierForTag(
            kUTTagClassMIMETYPE,
            (__bridge CFStringRef) mime, NULL);
    return theUTI;
}

```

开发者可以用上面这两个函数,把文件名及 MIME 类型转换成访问文件时经常会用到的 UTI 类型。

### 11.1.2 把 UTI 转换成扩展名或 MIME 类型

`UTTypeCopyPreferredTagWithClass()` 函数可以把 UTI 转换成首选的扩展名或 MIME 类型。如果给下面这两个函数传入 `public.jpeg`,那么它们就会分别返回 `jpeg` 和 `image/jpeg`:

```

NSString *extensionForUTI(NSString *aUTI)
{
    CFStringRef theUTI = (__bridge CFStringRef) aUTI;
    CFStringRef results =
        UTTypeCopyPreferredTagWithClass(
            theUTI, kUTTagClassFilenameExtension);
    return (__bridge_transfer NSString *)results;
}

NSString *mimeTypeForUTI(NSString *aUTI)
{
    CFStringRef theUTI = (__bridge CFStringRef) aUTI;
    CFStringRef results =
        UTTypeCopyPreferredTagWithClass(
            theUTI, kUTTagClassMIMETYPE);
    return (__bridge_transfer NSString *)results;
}

```

使用这两个函数的时候,必须传入树状结构最底端的 UTI,也就是与具体的扩展名直接对应的 UTI。开发者不能只给出上级 UTI 的类型。扩展名声明在属性列表中,像文

件扩展名及默认图标等特征信息，都是在属性列表里描述的。举例来说，开发者如果给 `extensionForUTI` 函数传入 `public.text` 或 `public.movie`，那么该函数会返回 `nil`，但如果传入的是 `public.plain-text` 和 `public.mpeg`，则会分别返回 `txt` 和 `mpg`。

前面给出的 `public.text` 及 `public.movie` 这两个 UTI，在整个树状结构中所处的地位太高了，`extensionForUTI` 函数需要开发者提供更加具体的类型，而不是这种比较抽象的类型。目前并没有 API 函数能够在抽象的 UTI 上面找到应用程序中继承自该 UTI 的所有子条目。读者可以访问 [bugreport.apple.com](http://bugreport.apple.com)，向苹果公司提交这一改进建议。肯定有某个地方注册了所有的扩展名和 MIME 类型，不然的话，`UTTypeCopyPreferredTagWithClass()` 函数怎么能够执行转换呢？所以，笔者认为，在扩展名与更加通用的 UTI 之间应该能够建立起对应关系。

### MIMEHelper 类

把扩展名转换成 UTI 的操作，基本上都可以得到结果，开发者只要把扩展名传给 `extensionForUTI`，差不多都可以返回 UTI。但是，把 UTI 转换成 MIME 的操作，却经常会落空。如果传入的 UTI 比较常见，那么通常可以得到适当的 MIME 类型，但如果传入的 UTI 比较少见，那就很难得到有效的答案了。

下面列出了扩展名、UTI（通过 `preferredUTIForExtension()` 函数获得），以及根据 UTI 所转换出来的 MIME 类型（通过 `contentTypeForUTI()` 函数获得）：

```
xlw: dyn.age81u5d0 / (null)
xlw: com.microsoft.excel.xlw / application/vnd.ms-excel
xm: dyn.age81u5k / (null)
xml: public.xml / application/xml
z: public.z-archive / application/x-compress
zip: public.zip-archive / application/zip
zoo: dyn.age81y55t / (null)
zsh: public.zsh-script / (null)
```

读者可以看到，上述有好几处都是 `null`。如果 `contentTypeForUTI()` 函数找不到与 UTI 相匹配的 MIME 类型，那就会返回 `nil`。为了解决此问题，本解决方案的范例代码提供了名为 `MIMEHelper` 的辅助类。它定义了下面用来把给定的扩展名转换成 MIME 类型的函数：

```
NSString *mimeForExtension(NSString *extension);
```

该函数所依据的扩展名及 MIME 类型，来自 Apache Software Foundation（Apache 软件基金会）对外公布的一份列表。本解决方案的范例代码一共支持 450 种扩展名，iOS 系统能够查出与这 450 种扩展名相对应的每一种 UTI，但却只能把其中的 89 种转换为 MIME 类型。使用了 Apache 所提供的数据之后，我们可以识别多达 230 种 MIME 类型。

### 11.1.3 判断两个 UTI 之间是否有依从关系

`UTTypeConformsTo()` 函数可以判断 UTI 之间的依从关系。此函数接收两个参数，一个表示待比较的源 UTI，另一个表示用于参照的 UTI。如果前者遵循后者，那么函数就返回 `true`。通过这个函数，我们可以判断出某个具体的 UTI 是否遵循另一个宽泛的 UTI。

UTTypeEqual() 函数则可以判断两个 UTI 是否相等。下面这段范例代码演示了如何判断某个文件路径是否指向图像资源：

```

BOOL pathPointsToLikelyUTIMatch(NSString *path, CFStringRef theUTI)
{
    NSString *extension = [path pathExtension];
    NSString *preferredUTI = preferredUTIForExtension(extension);
    return (UTTypeConformsTo(
        (__bridge CFStringRef) preferredUTI, theUTI));
}

BOOL pathPointsToLikelyImage(NSString *path)
{
    return pathPointsToLikelyUTIMatch(path, CFSTR("public.image"));
}

BOOL pathPointsToLikelyAudio(NSString *path)
{
    return pathPointsToLikelyUTIMatch(path, CFSTR("public.audio"));
}

```

### 11.1.4 获取依从关系列表

在 iOS 的 API 中，UTTypeCopyDeclaration() 可以算是最为通用也最为有用的 UTI 函数了。它所返回的字典 (dictionary) 中包含下列键 (key)：

- **kUTTypeIdentifierKey**——表示调用者传给函数的 UTI 名称 (例如, public.mpeg)。
- **kUTTypeConformsToKey**——表示该类型所遵循的上级类型 (例如, public.mpeg 遵循 public.movie)。
- **kUTTypeDescriptionKey**——如果 UTI 名称还有一种易于理解的描述方式, 那么该键所对应的值就是那种描述方式 (例如, “MPEG movie”)。
- **kUTTypeTagSpecificationKey**——该键对应于一份字典, 字典里含有与调用者所传入的 UTI 等效的 OSType (例如 MPG 与 MPEG)、文件扩展名 (例如 mpg、mpeg、mpe、m75 与 m15) 及 MIME 类型 (例如 video/mpeg、video/mpg、video/x-mpeg 与 video/x-mpg)。

除了上述常用的键之外, 还有一些键用于指明导入和导出的 UTI 声明 (kUTImportedTypeDeclarationsKey 与 kUTExportedTypeDeclarationsKey)、同 UTI 相关联的图标资源 (kUTTypeIconFileKey)、指向该类型描述页面的 URL (kUTTypeReferenceURLKey), 以及该 UTI 的版本字符串 (kUTTypeVersionKey)。

开发者可以在 UTTypeCopyDeclaration() 所返回的字典上面, 沿着依从关系树进行遍历, 从而构建出一份数组, 并且令这份数组能够包含该 UTI 所遵循的全部上级 UTI。比方说, public.mpeg 类型的 UTI 遵循 public.movie、public.audiovisual-content、public.data、public.item 及 public.content。解决方案 11-1 中的 conformanceArray 函数会把这些条目放在数组里面返回给调用者。

## 解决方案 11-1 测试 UTI 之间的依从关系

```

// Build a declaration dictionary for the given type
NSDictionary *utiDictionary(NSString *aUTI)
{
    NSDictionary *dictionary =
        (__bridge_transfer NSDictionary *)
        UTTypeCopyDeclaration((__bridge CFStringRef) aUTI);
    return dictionary;
}

// Return an array where each member is guaranteed unique
// but that preserves the original ordering wherever possible
NSArray *uniqueArray(NSArray *anArray)
{
    NSMutableArray *copiedArray =
        [NSMutableArray arrayWithArray:anArray];

    for (id object in anArray)
    {
        [copiedArray removeObjectIdenticalTo:object];
        [copiedArray addObject:object];
    }

    return copiedArray;
}

// Return an array representing all UTIs that a given UTI conforms to
NSArray *conformanceArray(NSString *aUTI)
{
    NSMutableArray *results =
        [NSMutableArray arrayWithObject:aUTI];
    NSDictionary *dictionary = utiDictionary(aUTI);
    id conforms = [dictionary objectForKey:
        (__bridge NSString *)kUTTypeConformsToKey];

    // No conformance
    if (!conforms) return results;

    // Single conformance
    if ([conforms isKindOfClass:[NSString class]])
    {
        [results addObjectsFromArray:conformanceArray(conforms)];
        return uniqueArray(results);
    }

    // Iterate through multiple conformance
    if ([conforms isKindOfClass:[NSArray class]])
    {
        for (NSString *eachUTI in (NSArray *) conforms)
            [results addObjectsFromArray:conformanceArray(eachUTI)];
    }
}

```

```

        return uniqueArray(results);
    }

    // Just return the one-item array
    return results;
}

```

### 获取解决方案代码

访问 <https://github.com/erica/iOS-7-Cookbook> 网页，并打开“C11 Documents”文件夹，即可找到与本章中的解决方案相对应的完整范例项目。

## 11.2 解决方案：访问系统剪贴板

剪贴板（pasteboard，有些系统将其称为 clipboard）为操作系统提供了一块集中存放数据的区域，使得应用程序之间可以共享数据。用户可以在某个程序里复制一份数据，然后切换到其他程序，将那份数据粘贴到那个程序里。大部分操作系统里面都有与剪切 / 复制 / 粘贴操作类似的功能。此外，用户也可以在同一个应用程序内部的文本框与视图之间复制并粘贴数据，而开发者则可以创建应用程序专用的剪贴板，它里面存放的数据，其他程序无法解读。

UIPasteboard 类可以访问设备共用的剪贴板及其中的内容。下面这行代码可以返回通用的系统剪贴板，绝大部分复制 / 粘贴操作都可以在它上面执行：

```
UIPasteboard *pb = [UIPasteboard generalPasteboard];
```

由系统所提供的通用剪贴板和搜索剪贴板是设备上所有程序共用的。除了这些共享的系统剪贴板之外，iOS 还提供了应用程序专用的剪贴板，以及带有自定义名称的剪贴板，同一个组织里共用同一个团队 ID 的应用程序开发者，可以在不同的应用程序之间使用这些剪贴板。pasteboardWithUniqueName 方法可以创建应用程序专用的剪贴板，该方法所返回的剪贴板对象，会一直延续到应用程序退出时为止。

pasteboardWithName:create: 方法用来创建共享的剪贴板，该方法会返回具备指定名称的剪贴板。create 参数的意思是：如果系统里没有这一剪贴板，那么是否应该新建它。创建好剪贴板后，如果把 persistent 属性设为 YES，那它就可以在程序运行完毕后继续保留其数据了。



**提示** 在 iOS 7 之前的系统里，开发者可以根据剪贴板的名称，在所有应用程序之间共享带有自定义名称的剪贴板，而不是只能在同一个组织的同一个开发团队里面共享。此特性在 iOS 7 系统中有所变化，《iOS 7 Release Notes》里面说明了这一点。原来有许多程序会共用这种自定义的剪贴板，而现在，这些程序无法按照以前的方式继续运作下去了。我们需要采用新的办法在程序间分享数据。比方说，可以考虑解决方案 11-8 中的 openURL，或是采用外部共享的存储区。

### 11.2.1 存储数据

剪贴板中可以存放一项或多项数据。剪贴板中的每项数据，都可以表示成一份包含若干键值对的字典，键值对里存放有数据及其类型。剪贴板里的某项数据，可以含有多份与之相关的条目，以便使其他程序能够找到它们所兼容的数据类型。我们常用 UTI 来表示数据类型。比方说，可以用 `public.text` 类型（具体来说就是 `public.utf8-plain-text` 类型）来存放文本数据，可以用 `public.url` 类型来保存 URL 地址，或是用 `public.jpeg` 类型来存放图像数据。这都是 iOS 程序常用的数据类型。

UIPasteboard 提供了一些方法，有的可以处理一个剪贴板条目，有的可以处理多个剪贴板条目，另外还有一些方法用于获取和设置剪贴板数据，以及查询剪贴板中的数据类型。处理单个剪贴板数据所用的那些方法，其中有很多处理的都是剪贴板里的首个条目。开发者可以经由 `items` 属性获取包含全部条目的数组。

开发者可以给剪贴板中的首个条目赋予一个 NSData 对象以及一个描述数据类型的 UTI，以便把该条目的数据同其类型关联起来：

```
[[UIPasteboard generalPasteboard]
 setData:theData forPasteboardType:theUTI];
```

另外，对于属性列表式的对象<sup>①</sup>（property list object，也就是字符串、日期、数组、字典、数值或 URL）来说，可以通过 `setValue:forPasteboardType:` 方法把它们放到剪贴板中（译者批注：该方法的第一个参数是 id 型，所以，为了谨慎起见，译文没有翻译句中的 `NSValue`。）。这些属性列表式的对象的内部存储方式与等效的原始数据（raw-data）不同，这体现出了 `setValue:forPasteboardType:` 方法与 `setData:forPasteboardType:` 方法之间的区别。

### 11.2.2 存储常见类型的数据

剪贴板还为某些数据类型提供了专门的方法，这些数据类型表示几种最常使用的剪贴板条目。它们是：颜色（这不是一种属性列表式的对象）、图像（这也不属于属性列表式的对象）、字符串及 URL。UIPasteboard 类提供了专用的设置器（getter）与获取器（setter），使得开发者可以更加方便地处理这几种数据类型。我们可以将其当成剪贴板的属性，直接以 “.” 写法来设置并获取它们。另外，每个属性都有一种对应的复数形式，这使得开发者可以经由对象数组来操作它们。

上面提到的这些剪贴板属性，极大地简化了向系统剪贴板里放置常用数据时所需编写的代码。可供使用的属性有下面这些：

- **string**——把剪贴板中的首个条目当成字符串来设置或获取。
- **strings**——把剪贴板中的所有条目当成字符串数组来设置或获取。
- **image**——把剪贴板中的首个条目当成图像来设置或获取。
- **images**——把剪贴板中的所有条目当成图像数组来设置或获取。

① 此概念请参阅 iOS 开发文档中的《About Property Lists》。——译者注

- **URL**——把剪贴板中的首个条目当成 URL 来设置或获取。
- **URLs**——把剪贴板中的所有条目当成 URL 数组来设置或获取。
- **color**——把剪贴板中的首个条目当成颜色对象来设置或获取。
- **colors**——把剪贴板中的所有条目当成颜色对象数组来设置或获取。

### 11.2.3 获取数据

如果要获取的数据属于前一节所提到的四种类型，那么只需使用相关的属性即可将其从剪贴板里取出来。否则，需要用 `dataForPasteboardType:` 方法来获取。该方法只会返回剪贴板第一个条目里的数据，而忽略剪贴板中的其他条目。

要想获取与某些类型相符的所有数据，需要调用 `itemSetWithPasteboardTypes:` 方法，并在返回的 `NSIndexSet` 上面遍历，以访问其中的每个字典。然后通过字典里的键和值来查明该条目所含的数据类型及数据内容。

当剪贴板中的数据有变化时，它会发出 `UIPasteboardChangedNotification` 通知，开发者可以通过 `NSNotificationCenter` 中默认的那些方法来添加监听器，以便监听此通知。另外，也可以监听 `UIPasteboardRemovedNotification`，以便在某条目移出剪贴板时得到通知。



**提示** 如果想把文本数据顺利粘贴到 Notes 或 Mail 程序，那么在向剪贴板中存储文本信息时，应该使用 `public.utf8-plain-text` 类型的 UTI。若通过 `string` 或 `strings` 属性来保存文本，则系统会默认使用该类型。

### 11.2.4 自动更新剪贴板

坦率地说，iOS 的选取与复制界面，并不是整个操作系统里最简洁的 UI 元件。有的时候，为了简化用户的操作流程，我们会把可能要与其他应用程序分享的那些内容提前准备好。

请看解决方案 11-2。用户在文本视图 (`UITextView`) 里面输入并编辑文本的时候，程序会自动把文本更新到剪贴板。如果 `enableWatcher` 变量处于启用状态（用户可以通过点击按钮来启用它），那么每次编辑文本时，程序都会把文本更新到剪贴板。为了实现此功能，我们编写了 `UITextViewDelegate` 协议中的 `textViewDidChange:` 方法，以便响应用户的编辑操作。在该方法中，我们通过 `updatePasteboard` 方法把改变之后的文本内容自动粘贴到剪贴板里面。

解决方案 11-2 自动把文本复制到剪贴板里

```
- (void)updatePasteboard
{
    // Copy the text to the pasteboard when the watcher is enabled
    if (enableWatcher)
        [UIPasteboard generalPasteboard].string = textView.text;
}
```



```

- (void)textViewDidChange:(UITextView *)textView
{
    // Delegate method calls for an update
    [self updatePasteboard];
}

- (void)toggle:(UIBarButtonItem *)bbi
{
    // switch between standard and auto-copy modes
    enableWatcher = !enableWatcher;
    bbi.title = enableWatcher ? @"Stop Watching" : @"Watch";
}

```

通过本条解决方案可以看出，访问及更新剪贴板是一件相对简单的事。

## 11.3 解决方案：监控 Documents 文件夹

iOS 的文档并不总是局限在沙盒之中。开发者可以而且也应该为用户提供文档分享功能。我们应该令用户能够直接控制他们的文档，并且能够访问到他们在设备上创建的任何内容。开发者只需在 Info.plist 中启用一个简单的设置，就可以令 iTunes 显示出用户 Documents 文件夹下的内容，并且可以令用户能够按照自己的需要来添加并删除这些内容。

过一阵子，也许开发者就可以通过 NSMetadataQuery 来监控 Documents 文档并观察其中的更新了。但在笔者编写本书时，它所能监控的元数据 (metadata) 仅仅局限在 iCloud 自身的目录之中。从 OS X 移植过来的代码，无法顺利运行在 iOS 平台上面。准确地说，目前只支持两种搜索范围，分别是 NSMetadataQueryUbiquitousDataScope 和 NSMetadataQueryUbiquitousDocumentsScope，而这两种搜索范围又都局限在 iCloud 之中。在 iOS 尚未支持更加通用的监控功能时，我们应该使用 kqueue 来编程。这项比较旧的技术提供了灵活的事件通知功能。开发者可以通过 kqueue 来监控、添加并清除事件。这样做大致相当于检测文件的添加及删除情况，而这两种情况正是应用程序要响应的主要状况。后面的解决方案 11-3 演示了如何用 kqueue 来监控 Documents 文件夹。

### 11.3.1 启用文件分享功能

若要启用文件分享功能，请向应用程序的 Info.plist 里面添加名为 Application supports iTunes file sharing 的键 (key)，并把其值设为 YES。开发者可以直接编辑 plist，也可以用 Xcode 里面的编辑器来修改它。如果想用编辑器修改，那就切换到 Project > Target > Info 画面，并在 Custom iOS Target Properties 区域中操作相关属性，如图 11-2 所示。若要直接操作它的值，则需使用名为 UIFileSharingEnabled 的键。iTunes 会把支持文件分享功能的所有应用程序都列在每台设备的 Apps 清单中，如图 11-3 所示。

### 11.3.2 用户对 Documents 文件夹的控制能力

开发者不能限定 Documents 文件夹里的文件类型。用户可以把任意类型的文件添加到里

面，也可以移除不想要的东西。但是，他们却不能在 iTunes 的界面里浏览 Documents 文件夹中的子文件夹。请看图 11-3 中的 Inbox 文件夹。该文件夹是开发者在应用程序之间分享文档后所留下的痕迹，它本来是不应该出现在 Documents 里面的。用户不能直接管理这些数据，而开发者也不应该把这样的子目录留在这里，否则，用户会感到困惑。

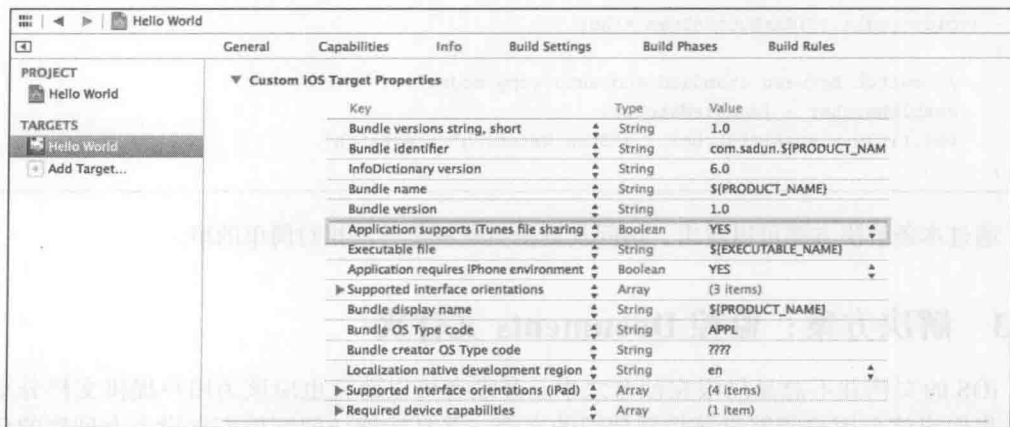


图 11-2 启用“Application supports iTunes file sharing”功能，使得用户可以通过 iTunes 来访问 Documents 文件夹

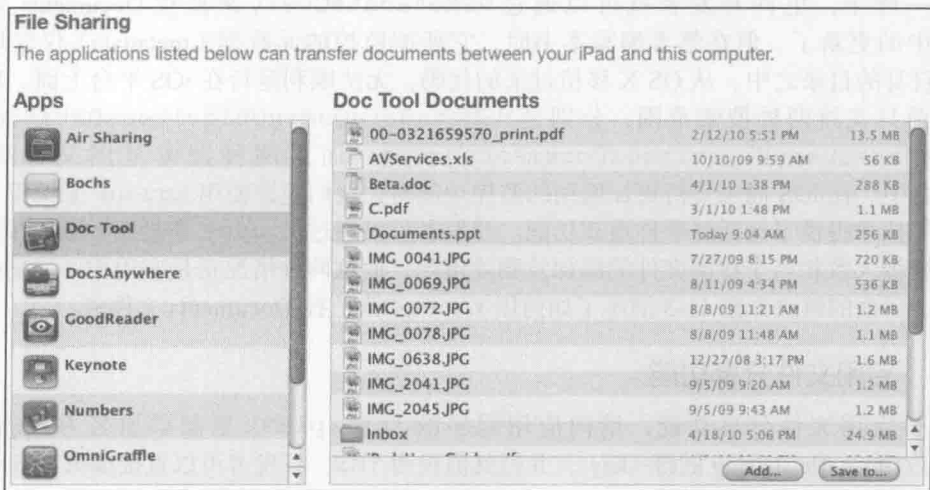


图 11-3 凡是宣称自己支持 UIFileSharingEnabled 功能的应用程序，都会出现在 iTunes 的 Apps 列表中

在 iTunes 中，用户不能像删除其他内容那样删除 Inbox 文件夹。而我们的应用程序，也不应该直接把文件写到 Inbox 里。此文件夹专门用来捕获由其他程序分享过来的数据，开发者应该明确它的这一用途。如果要实现文件分享功能，那么在重新启动应用程序时，应该检查 Inbox 文件夹，并恢复到上次运行时的状态，同时，应该把 Inbox 里的数据处理干净，并

将其删除。本章稍后将会讨论：怎样恰当地处理由其他程序分享过来的文档。

### 11.3.3 在 Xcode 里访问应用程序沙盒

开发者不仅能够访问 Documents 文件夹，而且还能访问整个应用程序沙盒。在 Xcode 中，通过 Organizer（可按“Command-Shift-2”组合键调出该界面）> Devices > Device（具体的设备）> Applications > Application Name（应用程序名称），即可浏览沙盒、向沙盒里上传文件，并从沙盒中下载文件。

若要测试文件分享功能，我们可以启用应用程序的 `UIFileSharingEnabled` 属性，并向 Documents 文件夹里上传一些数据。等创建好这些文件后，用 Xcode 及 iTunes 来查看、下载并删除它们。

### 11.3.4 扫描新的文档

解决方案 11-3 在 `beginGeneratingDocumentNotificationsInPath:` 方法中请求系统用 `kqueue` 来投递通知。该方法根据开发者所提供的路径（在本例中，该路径就是程序的 Documents 文件夹）来获取文件描述符，然后据此请求系统在适当的时机向队列中添加事件或清除事件的状态。它把这个功能添加到当前的“运行循环”（run loop）里面，这样的话，只要待监控的文件夹发生变化，就可以触发通知了。

解决方案 11-3 通过 `kqueue` 来监控文件变化

---

```
#import <fcntl.h>
#import <sys/event.h>

#define kDocumentChanged \
    @"DocumentsFolderContentsDidChangeNotification"

@interface DocWatchHelper : NSObject
@property (strong) NSString *path;
+ (id)watcherForPath:(NSString *)aPath;
@end

@implementation DocWatchHelper
{
    CFFiledescriptorRef kqref;
    CFRunLoopSourceRef rls;
}

- (void)kqueueFired
{
    int          kq;
    struct kevent event;
    struct timespec timeout = { 0, 0 };
    int          eventCount;

    kq = CFFiledescriptorGetNativeDescriptor(self->kqref);
```

```

assert(kq >= 0);

eventCount = kevent(kq, NULL, 0, &event, 1, &timeout);
assert( (eventCount >= 0) && (eventCount < 2) );

if (eventCount == 1)
    [[NSNotificationCenter defaultCenter]
        postNotificationName:kDocumentChanged
        object:self];

CFFileDescriptorEnableCallbacks(self->kqref,
    kCFFileDescriptorReadCallBack);
}

static void KQCallback(CFFileDescriptorRef kqRef,
    CFOptionFlags callBackTypes, void *info)
{
    DocWatchHelper *helper =
        (DocWatchHelper *)(__bridge id)(CTypeRef) info;
    [helper kqueueFired];
}

- (void)beginGeneratingDocumentNotificationsInPath:
    (NSString *)docPath
{
    int                dirFD;
    int                kq;
    int                retVal;
    struct kevent       eventToAdd;
    CFFileDescriptorContext context =
        { 0, (void *)(__bridge CTypeRef) self,
          NULL, NULL, NULL };

    dirFD = open([docPath fileSystemRepresentation], O_EVTONLY);
    assert(dirFD >= 0);

    kq = kqueue();
    assert(kq >= 0);

    eventToAdd.ident  = dirFD;
    eventToAdd.filter  = EVFILT_VNODE;
    eventToAdd.flags   = EV_ADD | EV_CLEAR;
    eventToAdd.fflags  = NOTE_WRITE;
    eventToAdd.data    = 0;
    eventToAdd.udata   = NULL;

    retVal = kevent(kq, &eventToAdd, 1, NULL, 0, NULL);
    assert(retVal == 0);

    self->kqref = CFFileDescriptorCreate(NULL, kq,
        true, KQCallback, &context);
    rls = CFFileDescriptorCreateRunLoopSource(

```

```

        NULL, self->kqref, 0);
    assert(rls != NULL);

    CFRunLoopAddSource(CFRunLoopGetCurrent(), rls,
        kCFRunLoopDefaultMode);
    CFRelease(rls);

    CFFileDescriptorEnableCallBacks(self->kqref,
        kCFFileDescriptorReadCallBack);
}

- (void)dealloc
{
    self.path = nil;
    CFRunLoopRemoveSource(CFRunLoopGetCurrent(), rls,
        kCFRunLoopDefaultMode);
    CFFileDescriptorDisableCallBacks(self->kqref,
        kCFFileDescriptorReadCallBack);
}

+ (id)watcherForPath:(NSString *)aPath
{
    DocWatchHelper *watcher = [[self alloc] init];
    watcher.path = aPath;
    [watcher beginGeneratingDocumentNotificationsInPath:aPath];
    return watcher;
}

@end

```

在收到回调的时候，它会投递通知（比方说，本例会在 `kqueueFired` 方法中投递自定义的 `kDocumentChanged` 通知）并继续监控新的事件。由于这些操作都运行在主线程的主运行循环里面，所以 GUI 依然能够响应用户的操作，并且能在收到通知的时候更新自己。

下面这段范例代码演示了如何用解决方案 11-3 的 `DocWatchHelper` 来更新 GUI 中的文件列表。当文件夹的内容有变化时，系统会给程序发送通知，而程序则会据此刷新列表，以反映文件夹中的新内容：

```

- (void)scanDocuments
{
    NSString *path = [NSHomeDirectory()
        stringByAppendingPathComponent:@"Documents"];
    items = [[NSFileManager defaultManager]
        contentsOfDirectoryAtPath:path error:nil];
    [self.tableView reloadData];
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    [self.tableView registerClass:[UITableViewCell class]

```

```

        forCellReuseIdentifier:@"cell"];
[self scanDocuments];

// React to content changes
[[NSNotificationCenter defaultCenter]
 addObserverForName:kDocumentChanged
 object:nil queue:[NSOperationQueue mainQueue]
 usingBlock:^(NSNotification *notification){
 [self scanDocuments];
 }];

// Start the watcher
NSString *path = [NSHomeDirectory()
 stringByAppendingPathComponent:@"Documents"];
helper = [DocWatchHelper watcherForPath:path];
}

```

测试这条解决方案的时候，请把设备与 iTunes 相连。然后用 iTunes 的 App 界面来添加并删除文件。此时，设备上面的文件列表就会随着用户的操作而持续更新了。

在使用解决方案 11-3 的时候，有几个问题要注意。首先，如果系统在创建某个文档的时候通知了应用程序，而这个文档又比较大，那就不要在接到通知之后立即读取其内容，而是应该持续查询文件的大小，以判断数据是否已经完全写入文件之中了。其次，iTunes 的文件传输功能可能偶尔会卡住，开发者需要根据情况做出适当处理。

## 11.4 解决方案：活动视图控制器

iOS 6 引入了活动视图控制器<sup>①</sup>，它可以把与数据相关的操作集成到图 11-4 这样的界面中。开发者只需编写很少一点代码，就可以使用这个控制器了。它使得用户可以把数据复制到剪贴板、发布到社交网站，或通过电子邮件及短信分享数据等。内置的操作包括：发布到 Facebook、发布到 Twitter、发布到 Weibo、发送短信、发送电子邮件、打印、复制到剪贴板、添加到联系人信息中，以及保存到 Camera Roll。iOS 7 又添加了一些新的操作，包括：添加到阅读列表、发布到 Flickr、发布到 Vimeo、发布到 Tencent Weibo 以及分享到 AirDrop。程序也可以定义自己特有的操作，本节稍后会讲到这个功能。下面列出目前支持的所有操作类型：

- UIActivityTypePostToFacebook



图 11-4 UIActivityViewController 类提供了一些系统服务及自定义服务

① 这里的活动 (activity) 一词，可以理解成“操作”或“动作”。——译者注

- UIActivityTypePostToTwitter
- UIActivityTypePostToWeibo
- UIActivityTypeMessage
- UIActivityTypeMail
- UIActivityTypePrint
- UIActivityTypeCopyToPasteboard
- UIActivityTypeAssignToContact
- UIActivityTypeSaveToCameraRoll
- UIActivityTypeAddToReadingList
- UIActivityTypePostToFlickr
- UIActivityTypePostToVimeo
- UIActivityTypePostToTencentWeibo
- UIActivityTypeAirDrop

上面显然缺少两项重要的操作：一个是“在……中打开”(Open in …)，该操作可以在程序之间共享文档；另一个是“Quick Look”(快速查看)，该操作可以预览文件内容。本章稍后会谈到这两个功能，也会用解决方案来演示如何将其独立地实现出来，另外，还会讲解怎样把 Quick Look 功能集成到活动视图控制器里面。

#### 11.4.1 展示活动视图控制器

这种控制器的展示方式，应该由具体设备决定。如果是在 iPhone 上面，那就应该以模态界面的形式来展示，但若在平板电脑上，则应将其展示为 popover。UIBarButtonItem-SystemItemAction 图标最适合用来把 UIBarButtonItem 与该控制器联系起来。

在比较理想的状况下，开发者几乎无须编写其他代码，即可实现数据分享。当用户选定某个操作之后，控制器会自动处理接下来的事情，例如：展示电子邮件撰写界面、展示 Twitter 消息撰写界面、将图片添加到设备媒体库，或是把图片设置成联系人的头像。

#### 11.4.2 UIActivityItemSource 协议

解决方案 11-4 创建并展示了活动视图控制器。它的主类实现了 UIActivityItemSource 协议，并且把 self (自己) 放在数组中，通过 activityItems 参数传给控制器：

```
UIActivityViewController *activity =
    [[UIActivityViewController alloc] initWithActivityItems:@[self]
    applicationActivities:nil];
[self presentViewController:activity];
```

实现了 UIActivityItemSource 协议的那个对象，负责提供将要操作的数据，而在本例中，该对象就是 self。

协议中有两个必须要实现的方法，分别用来提供待处理的数据（用户所选的操作就是针对这个数据执行的），以及与该数据相对应的占位符。用来表示该数据的那个对象，应该与给

定的操作类型相符。我们可以根据系统传给回调方法的操作类型来提供不同的数据对象。比方说，操作类型如果是“发布 Twitter 消息”，那么该方法返回的可能就是含有“I created a great song in App Name”（我在某程序里创建了一首动听的歌曲）字样的字符串，但如果是“发送电子邮件”，那么该方法可能会把包含声音信息的实际文件发过去。

#### 解决方案 11-4 活动视图控制器

```
- (void)presentViewController:
    (UIViewController *)viewControllerToPresent
{
    if (popover) [popover dismissPopoverAnimated:NO];
    if (IS_IPHONE)
    {
        [self presentViewController:viewControllerToPresent
            animated:YES completion:nil];
    }
    else
    {
        popover = [[UIPopoverController alloc]
            initWithContentViewController:viewControllerToPresent];
        popover.delegate = self;
        [popover presentPopoverFromBarButtonItem:
            self.navigationItem.leftBarButtonItem
            permittedArrowDirections:UIPopoverArrowDirectionAny
            animated:YES];
    }
}

// Popover was dismissed
- (void)popoverControllerDidDismissPopover:
    (UIPopoverController *)aPopoverController
{
    popover = nil;
}

// Return the item to process
- (id)activityViewController:
    (UIActivityViewController *)activityViewController
    itemForActivityType: (NSString *)activityType
{
    return imageView.image;
}

// Return a thumbnail version of that item
- (id)activityViewControllerPlaceholderItem:
    (UIActivityViewController *)activityViewController
{
    return imageView.image;
}
```



```
// Create and present the view controller
- (void)action
{
    UIActivityViewController *activity =
        [[UIActivityViewController alloc]
         initWithActivityItems:@[self]
         applicationActivities:nil];
    [self presentViewController:activity];
}
```

占位符一般来说都与返回的数据对象相同，除非我们需要在对象上面另做处理，或是要创建不同的对象。此时，开发者可以创建不含真实数据的占位符对象。

由于这两个回调方法都运行在主线程上面，所以我们应该尽量把数据设计得小一些。如果要处理数据，那么可以考虑使用下一节所讲的 `UIActivityItemProvider`。

iOS 7 还引入了几个可选的方法，开发者可以经由这几个委托方法来进一步配置待操作的数据。这些委托方法可以用来提供缩略图、主题文本，以及与特定操作类型相对应的 UTI。如果用户所要执行的操作支持这些信息，那么相关的服务就可以使用它们。

### 11.4.3 UIActivityItemProvider 类

`UIActivityItemProvider` 类可以在上一节所说的内容上面进行扩展，该类遵循 `UIActivityItemSource` 协议，它使得开发者可以稍后再向系统传递待操作的数据。该类继承了 `NSOperation` 类，这使得我们可以在操作数据之前对其进行一些灵活的处理。比方说，在向社交网站上传一份大的视频文件之前，我们想先处理这份文件，或是在使用一份较长的音频文件之前，我们想先截取其中的一段内容。

开发者应该从 `UIActivityItemProvider` 中继承子类，并实现 `item` 方法。在使用一般的 `NSOperation` 时，我们会实现 `main` 方法，而在使用时需要实现的则是 `item` 方法。开发者可以在该方法中生成处理过的数据，这样做不会妨碍用户对程序的操作，因为它是异步执行的。

### 11.4.4 实现 UIActivityItemSource 协议中的回调方法

解决方案 11-4 在初始化控制器的时候，把自己 (`self`) 放在了数组中，并传给了 `activityItems` 参数。由于自身实现了 `UIActivityItemSource` 协议，所以控制器在收到用户的请求时，就会通过回调方法来索取待操作的数据。在回调方法中，我们可以根据数据的用途来决定需要返回什么样的数据。开发者可以根据操作类型（比如，是分享到 Facebook 还是添加到联系人信息里，本节前面列出了各种操作类型）来提供准确的数据。如果要为不同的用途准备不同质量的资源，那么操作类型就显得尤为重要了。比方说，如果用户要打印某份数据，那么我们应该提供高质量的资源，但如果要把数据分享到 Twitter，那么只需提供一张质量较低的图像就行了。

如果程序针对各种操作所返回的都是同一份数据（比方说，无论用户是要在 Facebook

上面发布信息，还是要通过电子邮件来传送信息，我们都返回相同的数据)，那么，直接把数据（一般来说是字符串、图像及 URL）放在数组里传给控制器就好了，而无须再使用 `UIActivityItemSource` 对象。例如，用下面这行代码创建出来的控制器，无论针对何种操作，都只会返回一张图像：

```
UIActivityViewController *activity = [[UIActivityViewController alloc]
initWithActivityItems:@[imageView.image]
applicationActivities:nil];
```

这种直接使用 `UIActivityViewController` 的方式非常简单。程序的主类无须遵循 `UIActivityItemSource` 协议，也不用再去实现其他方法。如果待操作的数据比较简单，那就可以用这种便捷的方式来响应各种操作了。

待操作的数据未必只能有一份。在 `activityItems` 数组里，我们还可以添加更多元素。下面这个控制器会根据用户所选的操作类型，同时把两张图像通过电子邮件发出，或是将两张图像保存到系统的 `Camera Roll` 里面：

```
UIImage *secondImage = [UIImage imageNamed:@"Default.png"];
UIActivityViewController *activity = [[UIActivityViewController alloc]
initWithActivityItems:@[imageView.image, secondImage]
applicationActivities:nil];
```

一次操作多份数据，可以令用户更有效率地使用程序。

### 11.4.5 添加分享服务

每个程序都可以继承 `UIActivity` 的子类，并在初始化 `UIActivityController` 的时候，将该类的对象传过去，以便提供应用程序特有的操作。这种自定义的操作，也会和系统提供的操作一起出现在活动视图控制器里面。用户选择某个自定义的操作之后，它就会展示出视图控制器，使得用户可以与传过去的数据进行交互，或以某种方式使用相关的服务，比如输入密码或操控数据等。

程序清单 11-1 实现了一份模板式的 `UIActivity` 子类，它能够展示一套简单的文本视图界面。图 11-5 演示了该程序所特有的操作，它在图中以“List Items (Cookbook)”图标来表示。当用户点击此图标时，程序就会展示出自定义的视图控制器，这个控制器的界面上会把 `UIActivityController` 传给它的各条数据都列出来。它会列出每项数据的类及其描述信息。这个视图控制器里含有一段处理代码，当用户按下 Done 按钮时，它会调用 `activityDidFinish:` 方法，以便更新 `UIActivity` 实例。



图 11-5 把应用程序自定义的操作添加到 `UIActivityController` 里

程序清单 11-1 实现应用程序自定义的操作

```

// All activities present a view controller. This custom controller
// provides a full-sized text view.
@interface TextViewController : UIViewController
    @property (nonatomic, readonly) UITextView *textView;
    @property (nonatomic, weak) UIActivity *activity;
@end
@implementation TextViewController

// Make sure you provide a done handler of some kind, such as this
// or an integrated button that finishes and wraps up
- (void)done
{
    [_activity activityDidFinish:YES];
}

// Just a super-basic text view controller
- (instancetype)init
{
    self = [super init];
    if (self)
    {
        _textView = [[UITextView alloc] init];
        _textView.font =
            [UIFont fontWithName:@"Futura" size:16.0f];
        _textView.editable = NO;

        [self.view addSubview:_textView];
        PREPCONSTRAINTS(_textView);
        STRETCH_VIEW(self.view, _textView);

        // Prepare a Done button
        self.navigationItem.rightBarButtonItem =
            BARBUTTON(@"Done", @selector(done));
    }
    return self;
}
@end

// A custom activity subclass to display a list of source items
@interface MyActivity : UIActivity
@end

@implementation MyActivity
{
    NSArray *items;
}

// A unique type name
- (NSString *)activityType

```

```

{
    return @"CustomActivityTypeListItemsAndTypes";
}

// The title listed on the controller
- (NSString *)activityTitle
{
    return @"List Items (Cookbook)";
}

// A custom image that says "iOS" with a rounded rect edge
- (UIImage *)activityImage
{
    CGRect rect = CGRectMake(0.0f, 0.0f, 75.0f, 75.0f);
    UIGraphicsBeginImageContext(rect.size);
    rect = CGRectInset(rect, 15.0f, 15.0f);
    UIBezierPath *path = [UIBezierPath
        bezierPathWithRoundedRect:rect cornerRadius:4.0f];
    [path stroke];
    rect = CGRectInset(rect, 0.0f, 10.0f);
    NSMutableParagraphStyle * paragraphStyle =
        [[NSMutableParagraphStyle alloc] init];
    paragraphStyle.lineBreakMode = NSLineBreakByWordWrapping;
    paragraphStyle.alignment = NSTextAlignmentCenter;
    NSDictionary * attributes =
        @{NSParagraphStyleAttributeName : paragraphStyle,
         NSFontAttributeName : [UIFont fontWithName:@"Futura"
            size:18.0f]};
    [@"iOS" drawInRect:rect withAttributes:attributes];
    UIImage *image = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();

    return image;
}

// Specify if you can respond to these items
- (BOOL)canPerformWithActivityItems:(NSArray *)activityItems
{
    return YES;
}

// Store the items locally for later use
- (void)prepareWithActivityItems:(NSArray *)activityItems
{
    items = activityItems;
}

// Return a view controller, in this case one that lists
// its items and their classes
- (UIViewController *)activityViewController
{

```

```

    TextViewController *tvc = [[TextViewController alloc] init];
    tvc.activity = self;
    UITextView *textView = tvc.textView;

    NSMutableString *string = [NSMutableString string];
    for (id item in items)
        [string appendFormat:
         @"%@: %@\n", [item class], [item description]];
    textView.text = string;

    // Make sure to provide some kind of done: handler in
    // your main controller.
    UINavigationController *nav = [[UINavigationController alloc]
        initWithRootViewController:tvc];
    return nav;
}
@end

```

一定要添加一种令分享操作得以结束的方式，当控制器不会自行终止的时候，更要注意这个问题。把数据上传到 FTP 服务器时，开发者会知道上传操作何时结束。把信息发布到 Twitter 时，开发者也会知道信息到底贴上去了没有。然而本例却需要由用户来决定该操作应该在什么时候结束。视图控制器里面要设计一个属性，它应该是指向 `UIActivity` 的弱引用，等到分享操作结束时，我们可以通过该属性向 `UIActivity` 发送 `activityDidFinish:` 消息。

`UIActivity` 类里面有一些方法是必须实现的，另外一些方法则是可选的。开发者应该实现下面列出的所有方法，这些方法使得我们可以制作出应用程序特有的操作：

- **activityType**——该方法返回一个独特的字符串，用于描述这个操作的类型。此字符串应该与系统所提供的 `UIActivityTypePostToFacebook` 相仿。也就是说，两者的命名方式应该类似。字符串指明了操作的类型以及它要做的事情。程序清单 11-1 所返回的字符串是 `@“CustomActivityTypeListItemsAndTypes”`，它描述了这项操作的功能。
- **activityTitle**——该方法负责提供需要显示在活动视图控制器里的文本。图 11-5 中自定义的那段文本，就是由该方法所返回的。在描述应用程序自定义的操作时，应该采用主动句式。我们可以模仿苹果公司的做法，比方说，“Save to Camera Roll”（保存到 Camera Roll）、“Print”（打印）、“Copy”（复制）等。开发者所选的标题，应该是“I Want to...”（我想要……）句式的后面那一部分。比方说，可以是“I Want to Print”（我要打印）、“I Want to Copy”（我要复制），或是本例所使用的“I Want to List Items”（我要把这些条目列出来）。除了 to 或 and 等不太重要的单词之外，字符串里的其他单词均应首字母大写。
- **activityImage**——该方法返回控制器所使用的图像。控制器会把开发者所提供的图像转换成单色的位图。制作这种图标的时候，我们应该在透明的背景上面绘制简单的图案。

- **canPerformWithActivityItems:**——系统会用该方法来扫描传给 `UIActivity` 的各条数据。如果程序的控制器可以处理某条数据，那么该方法就应该针对这条数据返回 `YES`。
- **prepareWithActivityItems:**——开发者可以在该方法中保存系统传入的数据，以便稍后使用（在本例中，我们把传过来的数据保存到实例变量里），也可以预先对数据进行必要的处理。
- **activityViewController**——该方法应该用早前传入的那些数据返回一份完全初始化好了的视图控制器。该控制器将会自动展示给用户，这使得用户在执行相应的操作之前，能够先在界面中调整某些选项。

添加应用程序自定义的操作，可以扩大程序所能处理的数据类型，同时还能把这些功能集成到系统所提供的界面中。这是 iOS 系统里一个非常强大的特性。它使得开发者能够把功能非常强大的操作同系统服务（例如“复制到剪贴板”或“保存到相册”等）整合起来，或是将其与本机之外的 API 相连通，例如发布到 Facebook、发布到 Twitter、上传到 Dropbox、上传到 FTP 等。

本例中实现的操作没有那么强大，它只是把各项数据简单地列出来。该功能其实完全可以用程序内的普通界面来实现。但在思考“操作”这一概念的时候，应该把思维跳到应用程序之外。我们应该把用户的数据同一些分享操作或处理操作联系起来，而那些操作未必会以普通的 GUI 形式来呈现。

#### 11.4.6 与各种数据类型相对应的操作

用户能够在每条数据上面执行什么操作，取决于该数据的类型。表 11-1 列出了美国版 iPhone 手机中的各种源数据类型以及与之对应的操作。

表 11-1 各种数据类型所对应的操作

源数据	系统所能提供的操作
NSString 单个字符串或多个字符串	Message（通过短信发送）、Mail（通过邮件发送）、Twitter（发布至 Twitter）、Facebook（发布至 Facebook）、Copy（复制）
NSAttributedString 带属性的字符串	Message、Mail、Twitter、Facebook、Copy
UIImage 单张图像	Message、Mail、Twitter、Facebook、Save Image（保存图像）、Assign to Contact（指定给联系人）、Copy、Print（打印）
UIImage 多张图像	Message、Mail、Facebook、Save Image、Copy、Print
UIColor 颜色	Copy
NSURL URL	Message、Mail、Twitter、Facebook、Add to Reading List（添加到阅读列表）、Copy。采用 <code>assets-library:</code> 格式（ <code>scheme</code> ）的 URL 可以分享到 Facebook，采用 <code>mailto:</code> 格式的 URL 可以分享到 Mail 程序，采用 <code>sms:</code> 格式的 URL 可以分享到 Message 程序

(续)

源数据	系统所能提供的操作
UIPrintPageRenderer、UIPrintFormatter 及 UIPrintInfo	Print
NSDictionary 字典	如果这种对象可以分享，那么系统就会显示出分享对象所用的操作。但是系统并不支持数组
不支持的数据类型	例如 AVAsset、NSData、NSArray、NSDate 及 NSNumber，如果试图分享这些数据，那么系统展示出来的视图控制器将会是空白的
各种类型的多项数据	系统会把每项数据所支持的操作类型汇总起来（比方说，如果要同时操作字符串和图片，那么系统就会展示出 Message、Mail、Twitter、Facebook、Save Image、Assign to Contact、Copy 及 Print）

上述这些操作会随着语言设定而改变。我们在接下来的解决方案中会看到，除了这些基本类型之外，开发者还能通过预览控制器（preview controller）来支持其他类型：

- iOS 的 Quick Look 框架把活动控制器与文件预览相结合。由 Quick Look 所提供的活动控制器可以打印并发送很多类型的文档。用户也可以在某些类型的文档上面执行其他操作。
- 文档交互控制器（UIDocumentInteractionController，document interaction controller）提供了“Open in…”功能，使得用户能够在应用程序之间分享文件。这个控制器在展示文档的时候，既可以把各种操作都集成到一份“选项菜单”里面，也可以只列出与“Open in…”有关的操作。

#### 11.4.7 排除某些操作

如果想排除某些操作，那么可以将这些操作放在一份列表里，并设置 `excludedActivityTypes` 属性：

```
UIActivityViewController *activity =
    [[UIActivityViewController alloc]
     initWithActivityItems:items
     applicationActivities:@[appActivity]];
activity.excludedActivityTypes = @[UIActivityTypeMail];
```

### 11.5 解决方案：Quick Look 预览控制器

QLPreviewController 类使得用户可以预览很多类型的文档。该控制器支持文本、图像、PDF、RTF、iWork 文件、Microsoft Office 文档（Office 97 及之后的版本，包括 DOC、PPT、XLS 等）以及 CSV 文件。开发者提供一份控制器所支持的文件，Quick Look 控制器就会把它展示给用户。我们可以把系统提供的活动视图控制器集成进来，令用户可以像图 11-6 这样把正在预览的文档分享出去。

开发者既可以将这种控制器推入导航栈，也可以把它以模态形式直接显示出来。QLPreview-

Controller 能够适应这两种用法。解决方案 11-5 演示了这两种使用方式。

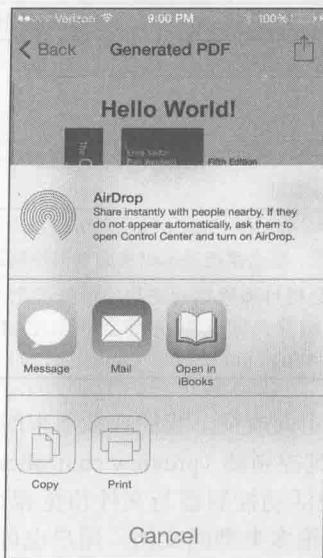


图 11-6 Quick Look 控制器以模式形式呈现，如果用户点击了动作按钮，那它就会显示出图中这幅画面。Quick Look 能够处理许多类型的文档，它使得用户能够先看到文档内容，然后再决定执行何种操作

## 实现 Quick Look 功能

若想实现 Quick Look，只需完成下列几步：

1. 令主控制器类遵从 QLPreviewControllerDataSource 协议。
2. 实现 numberOfPreviewItemsInPreviewController: 及 previewController:previewItemAtIndex: 两个数据源方法。前者返回待预览的条目数量。后者返回给定索引处的预览条目。

3. 预览条目要遵循 QLPreviewItem 协议。协议中包含两个必须实现的属性，一个是预览条目的标题，另一个是该条目的 URL。解决方案 11-5 所创建的 QuickItem 类遵从了 QLPreviewItem 协议。该类以极其简单的方式提供了数据源方法所需的数据。

满足了上述要求之后，我们可以在代码中新建预览控制器，设置其数据源，然后把它展示出来或将其推入导航栈。

### 解决方案 11-5 实现 Quick Look 功能

```
@interface QuickItem : NSObject <QLPreviewItem>
@property (nonatomic, strong) NSString *path;
@property (readonly) NSString *previewItemTitle;
@property (readonly) NSURL *previewItemURL;
@end
```

```
@implementation QuickItem
```



```

// Title for preview item
- (NSString *)previewItemTitle
{
    return [_path lastPathComponent];
}

// URL for preview item
- (NSURL *)previewItemURL
{
    return [NSURL fileURLWithPath:_path];
}
@end

#define FILE_PATH [NSHomeDirectory() \
    stringByAppendingPathComponent:@"Documents/PDFSample.pdf"]

@interface TestBedViewController : UIViewController
<QLPreviewControllerDataSource>
@end

@implementation TestBedViewController
- (NSInteger)numberOfPreviewItemsInPreviewController:
    (QLPreviewController *)controller
{
    return 1;
}

- (id <QLPreviewItem>)previewController:
    (QLPreviewController *)controller
    previewItemAtIndex:(NSInteger)index;
{
    QuickItem *item = [[QuickItem alloc] init];
    item.path = FILE_PATH;
    return item;
}

// Push onto navigation stack
- (void)push
{
    QLPreviewController *controller =
        [[QLPreviewController alloc] init];
    controller.dataSource = self;
    [self.navigationController
        pushViewController:controller animated:YES];
}

// Use modal presentation
- (void)present
{
    QLPreviewController *controller =

```

```

        [[QLPreviewController alloc] init];
        controller.dataSource = self;
        [self presentViewController:controller
            animated:YES completion:nil];
    }

- (void)loadView
{
    self.view = [[UIView alloc] init];
    self.view.backgroundColor = [UIColor whiteColor];

    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Push", @selector(push));
    self.navigationItem.leftBarButtonItem =
        BARBUTTON(@"Present", @selector(present));
}
@end

```

## 11.6 解决方案：使用文档交互控制器

UIDocumentInteractionController 类使得应用程序可以向用户展示一些交互选项，令其能够以多种方式来使用文档。借助这个类，用户可以享受如下好处：

- 由 iOS 系统所提供的应用程序间文档分享功能（也就是“Open this document in...*some app*”）。
- 由 Quick Look 框架所提供的文档预览能力。
- 由 UIActivityViewController 所提供的“打印”、“分享”以及“发布到社交网站”等操作。

本章前面在讲解活动视图控制器的时候，已经演示过后两项特性了。无论在样貌还是在行为上面，UIDocumentInteractionController 都与 UIActivityViewController 非常接近。UIDocumentInteractionController 增加了强大的程序间分享功能。

UIDocumentInteractionController 有两种样式，如图 11-7 所示。“Open in...”样式只提供“Open in”选项。而“options”样式则提供一份包含各种交互选项的列表，里面有“Open in...”、Quick Look 以及系统所支持的其他操作。这基本上相当于在标准的动作菜单所提供的功能之外，又加进了“Open in...”功能。开发者需要手工添加与 Quick Look 功能有关的回调，但这只需很少的代码即可实现。

### 11.6.1 创建 UIDocumentInteractionController 实例

每个 UIDocumentInteractionController 都针对一份特定的文件。该文件一般位于用户的 Documents 文件夹中，下面代码中的 fileURL 就代表这份文件：

```

dic = [UIDocumentInteractionController
    interactionControllerWithURL:fileURL];

```

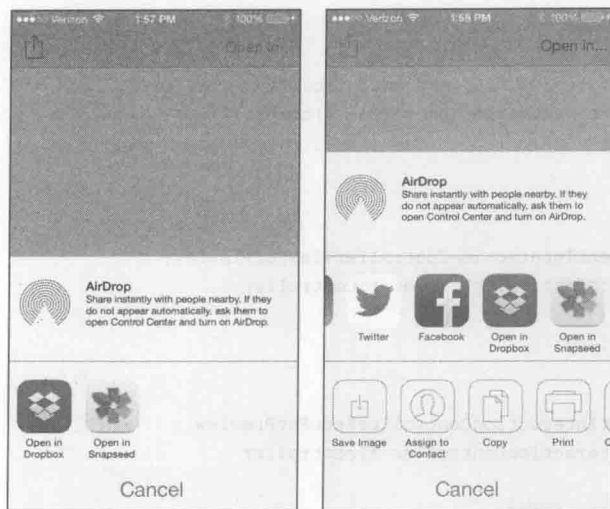


图 11-7 左右两张截图分别是“Open in...”风格与“options”风格的  
UIDocumentInteractionController

开发者提供指向本地文件的 URL，然后用“options”风格（其实就是动作菜单）或“Open in...”风格将控制器显示出来。我们可以用两种方式来显示“选项菜单”，一种是将其与 UIBarButtonItem 连接起来，另一种是将其显示在屏幕上的某块矩形区域内：

- presentOptionsMenuFromRect:inView:animated:
- presentOptionsMenuFromBarButtonItem:animated:
- presentOpenInMenuFromRect:inView:animated:
- presentOpenInMenuFromBarButtonItem:animated:

在 iPad 上面，系统会把 UIBarButtonItem 或开发者所传入的 rect 与将要显示的 popover 连接起来。而在 iPhone 上，系统则会显示一份模态的控制器视图。正如大家所料，iPad 上要进行更多的处理工作，因为用户可能会点击导航栏上面的其他按钮，也可能会关掉 popover。

等 iPad 显示出相关的控制器之后，我们就应该把每个 UIBarButtonItem 都禁用，然后等控制器消失之后，再启用它们。这是相当重要的一件事，因为我们肯定不希望用户重新点击正在使用中的 UIBarButtonItem，否则开发者又要处理另外一个 popover 了。如果没有仔细监控好按钮的状态以及正在显示的 popover，那就会遭遇很多尴尬的情况。解决方案 11-6 防范了这些情况。

#### 解决方案 11-6 使用 UIDocumentInteractionController

```
@implementation TestBedViewController
{
    NSURL *fileURL;
    UIDocumentInteractionController *dic;
    BOOL canOpen;
}
```

```

#pragma mark QuickLook
- (UIViewController *)
    documentInteractionControllerViewControllerForPreview:
        (UIDocumentInteractionController *)controller
{
    return self;
}

- (UIView *)documentInteractionControllerViewForPreview:
    (UIDocumentInteractionController *)controller
{
    return self.view;
}

- (CGRect)documentInteractionControllerRectForPreview:
    (UIDocumentInteractionController *)controller
{
    return self.view.frame;
}

#pragma mark Options / Open in Menu

// Clean up after dismissing options menu
- (void)documentInteractionControllerDidDismissOptionsMenu:
    (UIDocumentInteractionController *)controller
{
    self.navigationItem.leftBarButtonItem.enabled = YES;
    dic = nil;
}

// Clean up after dismissing open menu
- (void)documentInteractionControllerDidDismissOpenInMenu:
    (UIDocumentInteractionController *)controller
{
    self.navigationItem.rightBarButtonItem.enabled = canOpen;
    dic = nil;
}

// Before presenting a controller, check to see if there's an
// existing one that needs dismissing
- (void)dismissIfNeeded
{
    if (dic)
    {
        [dic dismissMenuAnimated:YES];
        self.navigationItem.rightBarButtonItem.enabled = canOpen;
        self.navigationItem.leftBarButtonItem.enabled = YES;
    }
}

// Present the options menu

```

```

- (void)action:(UIBarButtonItem *)bbi
{
    [self dismissIfNeeded];
    dic = [UIDocumentInteractionController
        interactionControllerWithURL:fileURL];
    dic.delegate = self;
    self.navigationItem.leftBarButtonItem.enabled = NO;
    [dic presentOptionsMenuFromBarButtonItem:bbi animated:YES];
}

// Present the open-in menu
- (void)open:(UIBarButtonItem *)bbi
{
    [self dismissIfNeeded];
    dic = [UIDocumentInteractionController
        interactionControllerWithURL:fileURL];
    dic.delegate = self;
    self.navigationItem.rightBarButtonItem.enabled = NO;
    [dic presentOpenInMenuFromBarButtonItem:bbi animated:YES];
}

#pragma mark Test for Open-ability
- (BOOL)canOpen:(NSURL *)aFileURL
{
    UIDocumentInteractionController *tmp =
        [UIDocumentInteractionController
            interactionControllerWithURL:aFileURL];
    BOOL success =
        [tmp presentOpenInMenuFromRect:CGRectMake(0,0,1,1)
            inView:self.view animated:NO];
    [tmp dismissMenuAnimated:NO];
    return success;
}

- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    // Only enable right button if the file can be opened
    canOpen = [self canOpen:fileURL];
    self.navigationItem.rightBarButtonItem.enabled = canOpen;
}

#pragma mark View management
- (void)loadView
{
    self.view = [[UIView alloc] init];
    self.view.backgroundColor = [UIColor whiteColor];
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Open in...", @selector(open:));
    self.navigationItem.leftBarButtonItem =
        SYSBARBUTTON(UIBarButtonSystemItemAction,

```

```

        @selector(action:));

    NSString *filePath = [NSHomeDirectory()
        stringByAppendingPathComponent:@"Documents/DICImage.jpg"];
    fileURL = [NSURL URLWithString:filePath];
}
@end

```

## 11.6.2 UIDocumentInteractionController 的属性

每个 `UIDocumentInteractionController` 都提供了若干属性，这些属性可以在控制器的委托回调方法里使用：

- **URL**——开发者可通过该属性查出控制器正在服务于哪份文件。这个 URL 与创建控制器时所传入的 URL 相同。
- **UTI**——该属性用来决定哪些应用程序可以打开此文档。它会根据文件名及元数据，用本章早前提到的系统函数来找出与之最匹配的 UTI。开发者可在代码中手工设置该属性，以覆盖由系统所提供的值。
- **name**——该属性对应于 URL 中的最后一个“路径组件”，这使得开发者无须手工截取 URL，即可获得到用户可以看懂的名称。
- **icons**——该属性可用来获取与当前文件类型相对应的图标。应用程序在声明自己所能支持的文件类型时，可以提供相关的图像链接（11.7 节会讲到这个问题）。这些图像对应于由 `kUTTypeIconFileKey` 键所指的值，本章前面曾经提到过 `kUTTypeIconFileKey`。
- **annotation**——开发者可以通过该属性给将要打开此文件的那个应用程序传递一些自定义的数据。这个属性没有标准的用法，不过它必须是属性列表式的顶级对象，也就是说，它必须是 `NSDictionary`、`NSArray`、`NSData`、`NSString`、`NSNumber` 或 `NSDate`。由于不同的开发者之间没有形成一套标准，所以使用该属性的场合不多，除非是一群开发者在他们自己设计的一组应用程序中借此分享信息。

## 11.6.3 提供快速查看文档的功能

在委托中实现下列三个回调方法，即可为控制器添加 Quick Look 功能：

```

#pragma mark QuickLook
- (UIViewController *)
    documentInteractionControllerViewControllerForPreview:
    (UIDocumentInteractionController *)controller
{
    return self;
}

- (UIView *)documentInteractionControllerViewControllerForPreview:
    (UIDocumentInteractionController *)controller
{

```

```

        return self.view;
    }

    - (CGRect)documentInteractionControllerRectForPreview:
        (UIDocumentInteractionController *)controller
    {
        return self.view.frame;
    }
}

```

上面这些方法分别用来指定展示预览画面的视图控制器、预览画面所在的视图，以及预览画面的 frame 尺寸。对于平板电脑来说，在个别情况下，我们可能需要用子视图控制器把预览画面显示到屏幕中的某一部分里（比方说，在使用分栏视图的时候，我们就希望预览画面只占据一部分屏幕空间），但对于 iPhone 来说，基本上都应该令预览画面占据整个屏幕。

#### 11.6.4 判断是否应启用“Open in…”操作

使用 `UIDocumentInteractionController` 的时候，Options 菜单里面几乎总是会列出有效的选项。在实现了 Quick Look 功能所需的回调方法之后，更是如此。但是，我们不一定能找到“Open in…”选项。该选项取决于开发者提供给控制器的文件数据，以及用户在设备中所安装的应用程序。

如果用户安装在设备上的程序无法打开控制器中的这种文档，那么就会碰到没有合适的“Open in…”操作可供执行的情况。这可能是由于文件类型比较奇怪，但一般来说，还是由于用户没有购买并安装相应的程序。在使用 iOS 模拟器运行程序时，经常会遇到这种情况。

开发者总是应该检查一下是否需要提供“Open in…”菜单项。解决方案 11-6 用了一个非常笨的办法来判断是否有其他应用程序能够显示并编辑 URL 中的文件。具体做法是：创建临时控制器，并试着用它来展示该文件。如果成功，就表示有程序能够打开此文件，并且该程序已经安装在设备上了。若是失败，则说明并没有这样的程序，此时我们应该禁用“Open in…”按钮。

对于 iPad 来说，开发者必须在 `viewDidAppear:` 中或在它之后执行此检测，也就是说，必须等视窗建起来之后才能进行检测。把临时控制器展示出来之后，我们立刻就将其关闭了，而且也没有使用动画效果，所以终端用户是不会发现它的。

这种实现方式显然不够优雅，但它却能在展示程序界面或使用一种新文件之前，首先判断出设备里有没有别的程序能打开它。读者可以通过 [bugreporter.apple.com](http://bugreporter.apple.com) 向苹果公司提交改进建议。

还有一点要注意：这种测试技巧在解决方案 11-6 这样的主视图上面是可行的，但对于 iPad 中某些非标准的 popover 界面来说，它可能还是有些问题。



提示 在同一个应用程序里，我们很少会同时给用户提供“选项菜单”及“Open in…”按钮。解决方案 11-6 采用系统自带的动作图标来表示选项菜单。如果程序只给用户提供各种“Open in…”操作，而不提供选项菜单，那么就可以用该图标来表示那些操作，而无须再使用解决方案 11-6 中的“Open in…”文字按钮了。

## 11.7 解决方案：声明程序所支持的文档类型

应用程序里的文档并不局限于程序本身所创建的那些文件，也不局限于它从网上下载的那些文件。在解决方案 11-6 中我们已经看到：程序可以处理某些特定类型的文件。它们可以打开由其他程序所发过来的文件。在本章前面的内容里，我们站在发送者的角度讲解了文档分享功能，并告诉大家怎样用控制器的“Open in…”选项把文件导出给其他应用程序。而现在，我们则要站在接收者的角度来看待文档分享功能。

应用程序可在其 Info.plist 属性列表中声明它所支持的文件类型。Launch Services 系统会读取这些数据，并创建出文件类型与程序之间的对应关系，以供 UIDocument-InteractionController 使用。

开发者可以直接编辑属性列表，不过 Xcode 的 Project > Target > Info 界面提供了一种更为简单的方式。打开 Custom iOS Target Properties 下方的 Document Types 区域。点击“+”按钮，即可添加应用程序所支持的文档类型。图 11-8 演示了如何用该界面来声明本程序支持 JPEG 图像文档。

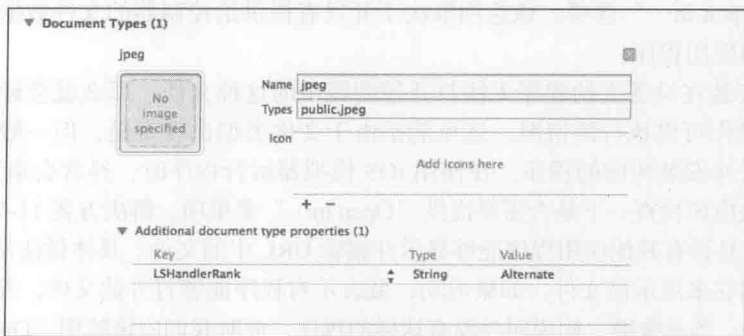


图 11-8 在 Xcode 的 Target > Info 界面中声明程序所支持的文档类型

声明文档类型的时候，应该提供下面三种信息：

- **名称**——开发者必须指定名称，但其内容可以是任意值。它应该描述当前文档类型，然而目前 iOS 并未使用该字段，它可能会用在以后的 iOS 系统里。这个字段在 Macintosh 上面或许更有用（Mac 系统的 Finder 程序会将其视为“kind”字符串），不过，开发者还是必须指定它。
- **一个或多个 UTI**——为当前类型指定一个或多个 UTI。本例只指定了 public.jpeg。如果有多个 UTI，就用逗号将其隔开。比方说，我们可以声明一种 image 文档类型，它对应于 public.jpeg、public.tiff 及 public.png 三种 UTI。若想把程序能够支持的文件类型限定得严格一些，那就使用具体的类型。虽说 public.image 可以涵盖上述三种类型，但这样做可能会使程序支持它本来不该支持的图像类型。
- **处理程序级别**——Launch Services 的处理程序级别，描述了当前程序在能够打开该类型文档的所有程序之中处于何种地位。如果是 Owner，就表明本程序是创建这种



文件的原生程序。若像图 11-8 一样把值设为 Alternate，则表明该程序只是充当这种文档的辅助查看器。开发者需要在 Additional document type properties 区域手工添加 LSHandlerRank 键。

开发者也可以指定图标文件。这些文件在 OS X 系统中会用作文档的图标，但是在 iOS 系统中的用途却不大。唯一会出现图标的情况，就是在 iTunes 里面使用 File Sharing 来添加及删除文件时，iTunes 会把程序图标列在 Apps 分页中。图标一般是 320×320 (UTTypeSize320IconFile) 及 64×64 (UTTypeSize64IconFile) 大小，而且通常仅限于程序能够创建的文件或是程序为其定义了自定义类型的文件。

Xcode 会根据开发者在上述交互式界面里设置的数据，在程序的 Info.plist 之中构建一个 CFBundleDocumentTypes 数组。图 11-8 中的配置会在 Info.plist 里产生下面这段代码：

```
<key>CFBundleDocumentTypes</key>
<array>
  <dict>
    <key>CFBundleTypeIconFiles</key>
    <array/>
    <key>CFBundleTypeName</key>
    <string>jpg</string>
    <key>LSHandlerRank</key>
    <string>Alternate</string>
    <key>LSItemContentTypes</key>
    <array>
      <string>public.jpeg</string>
    </array>
  </dict>
</array>
```

### 11.7.1 创建自定义的文档类型

如果程序构建了新类型的文档，那么开发者应该在 Target > Info 编辑器界面的 Exported UTIs 区域中声明这些类型，如图 11-9 所示。这样做相当于向系统注册了这种文件类型，并使得系统可以把本程序视为该类型的拥有者。

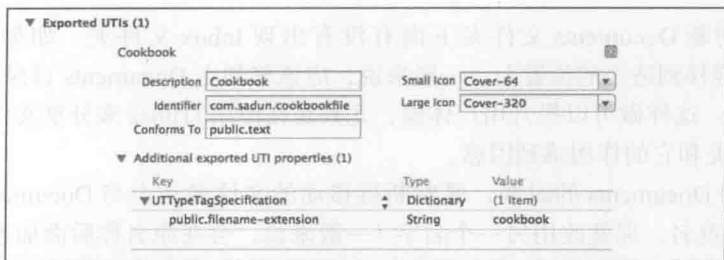


图 11-9 在“Target > Info”编辑器界面的“Exported UTIs”区域中声明程序自定义的文件类型

定义新类型的时候，应该提供自定义的 UTI（本例中是 com.sadun.cookbookfile）、

文档图标（64×64 与 320×320 大小），以及能够表示文件类型的文件扩展名。与声明程序所支持的文档类型一样，Xcode 也会向项目的 Info.plist 文件中导出一份数组。图 11-9 中的配置将会产生下面的代码：

```
<key>UTExportedTypeDeclarations</key>
<array>
  <dict>
    <key>UTTypeConformsTo</key>
    <array>
      <string>public.text</string>
    </array>
    <key>UTTypeDescription</key>
    <string>Cookbook</string>
    <key>UTTypeIdentifier</key>
    <string>com.sadun.cookbookfile</string>
    <key>UTTypeSize320IconFile</key>
    <string>Cover-320</string>
    <key>UTTypeSize64IconFile</key>
    <string>Cover-64</string>
    <key>UTTypeTagSpecification</key>
    <dict>
      <key>public.filename-extension</key>
      <string>cookbook</string>
    </dict>
  </dict>
</array>
```

如果把上述配置添加到自己的项目中，那么你的程序就可以用 com.sadun.cookbookfile 类型的 UTI 打开扩展名为 cookbook 的文件了。

### 11.7.2 实现对文档的支持

应用程序对文档提供支持的时候，应该在每次处于活动状态时检查 Inbox 文件夹：

```
- (void)applicationDidBecomeActive:(UIApplication *)application
{
    // perform inbox test here
}
```

我们应该判断 Documents 文件夹下面有没有出现 Inbox 文件夹。如果有，就应该把 Inbox 里面的内容移到适当的位置上，一般来说，应该移到主 Documents 目录下。清空 Inbox 之后，就删掉它。这样做可以提升用户体验，尤其是在使用 iTunes 来分享文件时，不会令用户对 Inbox 文件夹和它的作用感到困惑。

把文件移到 Documents 的时候，要判断待移动的文件名是否与 Documents 中已有的文件相冲突，如果重名，那就改用另一个名字（一般来说，会在原名称后面加连字符，然后再跟个数字），以免覆写了现有的文件。解决方案 11-7 会根据目标路径寻找可供使用的文件名。如果尝试了一千次之后还找不到合适的名称，那就放弃此文件。正常情况下用户不会产生那么多份重名的文档。若真是那样，则说明应用程序的设计有严重缺陷。

## 解决方案 11-7 处理传入的文档

```

#define DOCUMENTS_PATH [NSHomeDirectory() \
    stringByAppendingPathComponent:@"Documents"]
#define INBOX_PATH [DOCUMENTS_PATH \
    stringByAppendingPathComponent:@"Inbox"]

@implementation InboxHelper
+ (NSString *)findAlternativeNameForPath: (NSString *)path
{
    NSString *ext = path.pathExtension;
    NSString *base = [path stringByDeletingPathExtension];
    for (int i = 1; i < 999; i++)
    {
        NSString *dest =
            [NSString stringWithFormat:@"%d.%@", base, i, ext];

        // if the file does not yet exist, use this destination path
        if (![NSFileManager defaultManager]
            fileExistsAtPath:dest)
            return dest;
    }

    NSLog(@"Exhausted possible names for file %@. Bailing.",
        path.lastPathComponent);
    return nil;
}

- (void)checkAndProcessInbox
{
    // Does the Inbox exist? If not, we're done
    BOOL isDir;
    if (![NSFileManager defaultManager]
        fileExistsAtPath:INBOX_PATH isDirectory:&isDir)
        return;

    NSError *error;
    BOOL success;

    // If the Inbox is not a folder, remove it
    if (!isDir)
    {
        success = [[NSFileManager defaultManager]
            removeItemAtPath:INBOX_PATH error:&error];
        if (!success)
        {
            NSLog(@"Error deleting Inbox file (not directory): %@",
                error.localizedDescription);
            return;
        }
    }
}

```

```

// Retrieve a list of files in the Inbox
NSArray *fileArray = [[NSFileManager defaultManager]
    contentsOfDirectoryAtPath:INBOX_PATH error:&error];
if (!fileArray)
{
    NSLog(@"Error reading contents of Inbox: %@",
        error.localizedDescription);
    return;
}

// Remember the number of items
NSUInteger initialCount = fileArray.count;

// Iterate through each file, moving it to Documents
for (NSString *filename in fileArray)
{
    NSString *source = [INBOX_PATH
        stringByAppendingPathComponent:filename];
    NSString *dest = [DOCUMENTS_PATH
        stringByAppendingPathComponent:filename];

    // Is the file already there?
    BOOL exists =
        [[NSFileManager defaultManager] fileExistsAtPath:dest];
    if (exists) dest = [self findAlternativeNameForPath:dest];
    if (!dest)
    {
        NSLog(@"Error. File name conflict not resolved");
        continue;
    }

    // Move file into place
    success = [[NSFileManager defaultManager]
        moveItemAtPath:source toPath:dest error:&error];
    if (!success)
    {
        NSLog(@"Error moving file from Inbox: %@",
            error.localizedDescription);
        continue;
    }
}

// Inbox should now be empty
fileArray = [[NSFileManager defaultManager]
    contentsOfDirectoryAtPath:INBOX_PATH error:&error];
if (!fileArray)
{
    NSLog(@"Error reading contents of Inbox: %@",
        error.localizedDescription);
    return;
}

```

```

    if (fileArray.count)
    {
        NSLog(@"Error clearing Inbox. %d items remain",
              fileArray.count);
        return;
    }

    // Remove the inbox
    success = [[NSFileManager defaultManager]
               removeItemAtPath:INBOX_PATH error:&error];
    if (!success)
    {
        NSLog(@"Error removing inbox: %@",
              error.localizedDescription);
        return;
    }

    NSLog(@"Moved %d items from the Inbox", initialCount);
}
@end

```

解决方案 11-7 会逐个扫描 Inbox 目录里的文件，并将其移走。它会在清空 Inbox 目录之后把该目录删除。正如大家所见，这些方法会频繁使用与 File Manager (NSFileManager, 文件管理器) 有关的功能。这么做主要是为了把任务执行过程中可能出错的各种情况都处理好，以免干扰程序运行。如果 Inbox 里面都是小文件，那么对这些文件的处理应该很快就能完成。但如果要处理视频或音频等大文件，那就应该在自己的操作队列上完成此任务了。

程序如果要支持 public.data 类型的文件（也就是说，程序想要打开任意类型的文件），那么可能需要使用 UIWebView 实例来显示这些文件。《Technical Q&A QA1630》(<http://developer.apple.com/library/ios/#qa/qa1630>) 详细列出了 iOS 系统在这种视图里面可以显示以及不能显示的文件类型。除了可以显示简单的 HTML 以外，UIWebView 还能展示大多数音频及视频文件，以及 Excel、Keynote、Numbers、Pages、PDF、PowerPoint、Word 等文件。

## 11.8 解决方案：创建基于 URL 的服务

苹果公司内置的应用程序提供了很多可以通过 URL 来调用的服务。开发者可以用 Safari 来开启网页、用 Maps 来显示地图，或通过 mailto: 格式的 URL 启动 Mail 程序，以展示编写邮件的界面。URL 模式就是 URL 开头的那一部分，也就是出现在冒号前的字符，比方说 http 或 ftp。

这些服务之所以能够运作，是因为 iOS 系统知道如何把 URL 模式与应用程序对应起来。以 http: 开头的 URL 会用 Mobile Safari 来打开。而以 mailto: 开头的 URL 则会转向 Mail 程序。有些读者也许不知道：我们可以定义自己的 URL 模式，并在应用程序里面实现它们。并非所有的标准模式都受 iOS 系统支持，例如 FTP 模式就无法使用。

自定义的模式使得应用程序能够以 Mobile Safari 或另一个程序来打开那种类型的 URL。比方说，如果程序注册了 xyz，那么以 xyz: 开头的链接就会交由该程序来处理，系统会把链接

传给应用程序委托的 `application:openURL:sourceApplication:annotation:` 方法。不过,我们并不需要在那个方法里面专门编写代码。如果开发者只是想启动应用程序,那么把模式添加到程序里面就可以了,系统会在打开相关类型的 URL 时,自动实现程序间的跳转。

通过处理程序,我们可以在系统跳转到本程序的时候对传入的 URL 做一些处理。比方说,可以打开某个特定的数据文件、获取某个特定的名称、显示某张图像,或是处理包含在调用中的某些信息。

### 11.8.1 声明模式

Target > Info 编辑界面的 URL Types 区域列出了程序自定义的 URL 模式,开发者可以在这里声明 URL 模式,如图 11-10 所示。图中所声明的模式将会在 `Info.plist` 里面产生下列代码:

```
<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleURLName</key>
    <string>com.sadun.urlSchemeDemonstration</string>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>xyz</string>
    </array>
  </dict>
</array>
```

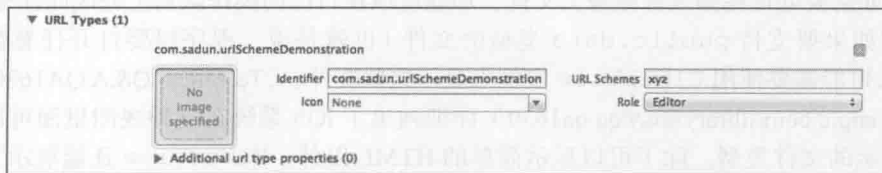


图 11-10 在 Target > Info 编辑界面的 URL Types 区域中添加自定义的 URL 模式

名为 `CFBundleURLTypes` 的条目中包含了一份由字典所构成的数组,用来表述程序所能打开并处理的 URL 类型。每份字典都相当简单,里面包含了两个键:一个是 `CFBundleURLName`,用来表示开发者所指定的标识符;另一个是 `CFBundleURLSchemes`,用来表示 URL 模式数组。

模式数组列出了属于这个抽象标识符的一系列 URL 前缀。开发者可以添加一个或多个模式。本例只声明了一种模式。你可以在想要使用的名称前面再加一个 `x` (例如 `x-sadun-services`)。虽说制定 URL 模式标准的组织没有规定 iOS 程序所应使用的模式,但我们可以通过开头的这个 `x` 来表示这是个未注册的名称。<http://x-callback-url.com> 网站正在研讨 `x-callback-url` 规范草案。

有很多非正式的模式注册网站,iOS 开发者可以把自己的模式分享到这些网站中。我们可以找到自己想调用的程序遵循何种模式,也可以把自己所使用的模式告诉别人。这种网

站会把各项服务及其 URL 模式列出来，并告诉其他开发者应该如何使用这些服务。<http://handleopenurl.com>、[http://wiki.akosma.com/iphone\\_url\\_schemes](http://wiki.akosma.com/iphone_url_schemes) 及 <http://appllookup.com/Home> 都是这样的网站。

## 11.8.2 测试 URL

开发者可以测试某个 URL 服务是否可用。UIApplication 的 `canOpenURL:` 方法如果返回 YES，就表明我们可以用 `openURL:` 来启动另一个程序，并开启那个 URL：

```
if ([[UIApplication sharedApplication] canOpenURL:aURL])
    [[UIApplication sharedApplication] openURL:aURL];
```

这并不表示 URL 本身是有效的，它只能说明系统里面已经有程序正确地注册了这种模式。

## 11.8.3 添加处理程序方法

为了处理由其他程序传过来的 URL 请求，我们可以像解决方案 11-8 一样，实现 UIApplicationDelegate 协议中的 `application:openURL:sourceApplication:annotation:` 方法。此方法只能在应用程序已经运行的时候触发。如果程序尚未运行，而系统又接到了需要由该程序来处理的 URL 请求，那么系统会先执行与程序启动有关的那两个回调方法（也就是 `application:didFinishLaunchingWithOptions:` 和 `application:willFinishLaunchingWithOptions:`）。

### 解决方案 11-8 提供对 URL 模式的支持

---

```
// Called if the app is open or if didFinishLaunchingWithOptions returns YES
- (BOOL)application:(UIApplication *)application
    openURL:(NSURL *)url
    sourceApplication:(NSString *)sourceApplication
    annotation:(id)annotation
{
    NSString *logString = [NSString stringWithFormat:
        @"DID OPEN: URL[%@] App[%@] Annotation[%@]\n",
        url, sourceApplication, annotation];
    tbvc.textView.text =
        [logString stringByAppendingString:tbvc.textView.text];
    return YES;
}

// Make sure to return YES
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    _window = [[UIWindow alloc]
        initWithFrame:[UIScreen mainScreen] bounds];
    tbvc = [[TestBedViewController alloc] init];

    UINavigationController *nav = [[UINavigationController alloc]
        initWithRootViewController:tbvc];
```

```
window.rootViewController = nav;  
[window makeKeyAndVisible];  
return YES;  
}
```

我们应该确保 `application:didFinishLaunchingWithOptions:` 方法能够照常返回 YES。这样的话,系统才能把控制权交给 `application:openURL:sourceApplication:annotation:`,使得本程序可以处理传入的 URL。

## 11.9 小结

开发者可能需要在应用程序之间共享数据,并且需要使用由系统所提供的某些操作。本章讲述了这些功能的实现方式。读者学到了 UTI 这个概念,还学到了在应用程序之间传递数据的时候,应该如何用 UTI 来表示数据用途。大家看到了剪贴板的工作原理,也看到了怎样用 iTunes 分享文件。此外,我们还讲了如何监控文件夹以及如何实现自定义的 URL 模式。读者深入了解到活动视图控制器 (`UIActivityViewController`) 及文档交互控制器 (`UIDocumentInteractionController`) 的用法,并且看到了如何令应用程序支持“打印”及“预览”等操作。学习下一章之前,先回顾下面几个知识点:

- 开发者未必非要使用苹果公司内置的 UTI,不过,在定义自己的 UTI 时,应该遵循苹果公司的惯例。务必按照反向域名格式来定义 UTI,并且在导出的定义信息里面尽量多提供一些细节(诸如公开的 URL 定义页面、常用的图标以及文件扩展名等)。UTI 定义得越准确越好。
- 通过解决方案 11-1 提供的依从关系数组,我们可以判断出数据的类型。知道了这一信息之后,开发者就可以更好地处理文件中的数据了,比方说,我们可以判断出待处理的数据是图像,而不是文本文件或视频文件。
- Documents 目录属于用户,而不属于开发者。我们应该记住这一点,并把这个目录管理好。
- 如果在编写数据分享功能时需要实现“一站式体验”,那么可能需要寻找比活动视图控制器 (`UIActivityViewController`) 更好的解决办法。不过,它依然是个易于使用且易于展示的控制器的,其中包含了一大批功能,使得我们可以把 iOS 系统所提供的服务集成到自己的程序里。
- 由于多种原因,很多程序员都用过自定义的 URL 模式,然而文档交互控制器 (`UIDocumentInteractionController`) 提供了一种更好的方案。这个控制器可以根据用户的需要,实现应用程序之间的交互,开发者不妨指定一些注释 (annotation) 信息,使得接收数据的那个应用程序能够更加方便地处理数据。
- 如果系统里面没有其他程序可以打开当前文档,那就不要提供“Open in…”按钮。本章给出了一种比较原始的解决办法,它虽然不够优雅,但是若不解决该问题,会令用户感到愤怒、沮丧或者困惑。在使用这套办法的基础上,我们可以提供一个警示界面,告诉用户没有别的程序能打开此文档。



## 浅谈 Core Data

iOS 的 Core Data 框架提供了保存持久化数据的解决方案。应用程序可以查询并更新 Core Data 的托管数据存储器区。Core Data 提供了一套基于 Cocoa Touch 的对象接口，使 iOS 开发者可以用熟悉的 Objective-C 来管理关系型数据库，而不必再使用 SQL 查询语句。Core Data 技术能够同表格视图与集合视图完美结合起来。

本章介绍 Core Data。笔者会给出足够的入门知识，使你初步了解这门技术，并为今后深入学习 Core Data 打下基础。读完这一章之后，大家就会明白应该如何在 iOS 程序里面使用 Core Data，并且了解该技术。

### 12.1 Core Data 简介

Core Data 简化了应用程序创建及使用持久化对象的方式，这种新方式就是托管对象。在 3.x 版本之前的 SDK 中，用来管理数据及访问 SQL 的所有操作，都位于一套相当底层的程序库里。那个程序库不够优雅，而且也不易使用。从 3.x 开始，Core Data 就加入了 Cocoa Touch 框架系列中，它为 iOS 开发者带来了一套强大的数据管理机制。Core Data 有着相当灵活的底层架构，而且提供了操作持久化数据存储区的工具，同时还可以生成管理对象整个生命期的解决方案。

在模型 - 视图 - 控制器 (Model-View-Controller, MVC) 范式中，Core Data 位于模型部分。之所以要这样设计，是因为程序专用的那部分数据应该定义在程序的 GUI 以外，而且应该从 GUI 外面操作，即便这些数据是用来驱动应用程序界面的，我们也依然应该这样想。Core Data 能够同表格视图与集合视图很好地集成在一起。Cocoa Touch 的 `NSFetchedResultsController` 类就是这样设计并构建出来的，它能够与上面说的那些

视图类协同运作。该类提供了一些有用的属性及方法，使得开发者可以提供数据源，并把相关的委托集成进来。

## 12.2 实体与模型

实体位于 Core Data 体系最顶层，描述了数据库里面存储的对象。实体就好比饼干切割机，用来指明每个数据对象应该如何创建出来。创建新对象的时候，实体会详细描述出构成每个对象的属性及关系。每个实体都有名称，程序在运行的时候，Core Data 会根据这个名称来获取实体的描述信息。

开发者需要在模型文件里构建实体。每个同 Core Data 框架相链接的项目，都包含一个或多个模型文件。这些 .xcdatamodeld 文件定义了实体、实体的属性以及实体间的关系。

### 12.2.1 构建模型文件

要想创立 Core Data 模型，我们需要在 Xcode 中新建数据模型文件。某些 iOS 模板已经把 Core Data 作为项目的一部分包含进来了。如果不是这样，那么开发者需要在 Xcode 中手工创建它。选择 Xcode 的 File > New > File 菜单项，然后选择 iOS、Core Data、Data Model，并单击 Next 按钮。输入新文件的名称（本例使用的是 Person），勾选与项目相对应的目标，然后单击 Save 按钮。Xcode 会新建模型文件，并将其加入项目中（本例的模型文件叫作 Person.xcdatamodeld）。在 File Navigator 中点击 xcdatamodeld 文件，就会打开如图 12-1 所示的编辑器窗口。

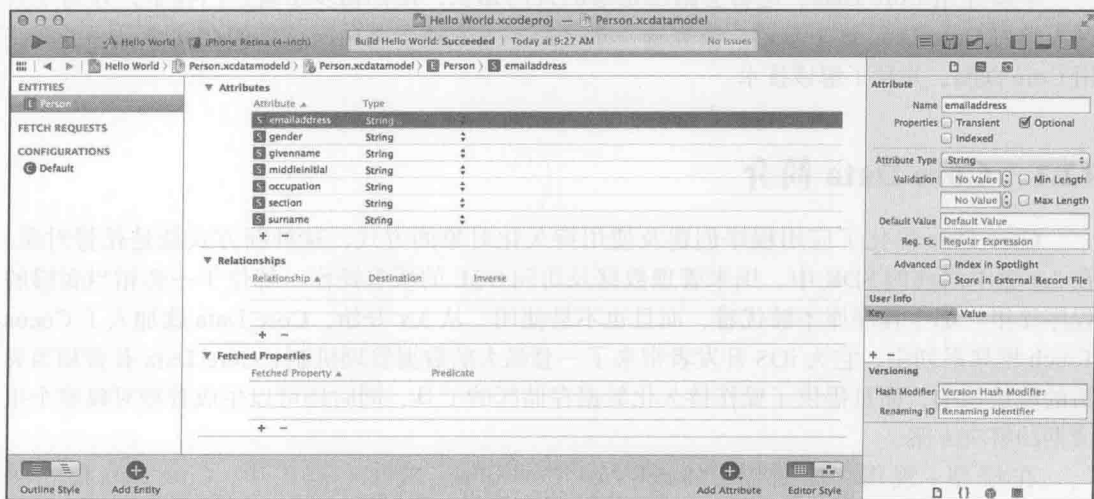


图 12-1 开发者可以在 Xcode 的编辑器中为自己的 Core Data 应用程序定义托管对象

在编辑器窗口中，我们可以点击左下角的 Add Entity 按钮来向左侧列表里添加新的实体（实体基本上相当于对象所属的类），还可以点击右下角的 Add Attribute 按钮来添加属性（实

体的属性相当于实例变量)。双击实体名称或属性名称,即可为其改名,另外可以用 Type 列表框来修改属性的类型。

开发者可以在编辑器中间的区域里定制实体的属性及关系。关系是一种联系方式,能够把本实体与数据库中的其他实体关联起来。右侧的 inspector 面板提供了与当前情境相关的设定,在图 12-1 中,列出了 Person 实体 emailaddress 属性的详细信息。

Entity 编辑器提供了两种界面风格。你可以点击编辑器面板右下角的 Editor Style 按钮,在表格视图与对象图之间切换。

图 12-1 中的详细列表风格,会列出模型中定义每个实体、实体的每个属性以及与其他实体的关系。而对象图风格的界面则会以网格状来显示模型中定义的实体,并以直观的方式描述实体间的关系,使得开发者可以编辑这些关系。例如,父亲或母亲 (parent) 可以有多个孩子 (child) 及一名配偶 (spouse)。部门 (department) 可以包含多位成员 (member),经理 (manager) 可以服务于多个委员会 (committee)。

### 12.2.2 属性与关系

每个实体都可以包含属性,用来存储名字、生日、称谓等信息。与实体相对应的 Objective-C 对象,则会用性质 (property) 来描述这些属性。

每个实体也可以定义关系,用来把一个对象与另一个对象联系起来。关系可以是单一的 (single),即一对一关系 (one-to-one relationship, 比如某人和其配偶的关系、某人和其雇主的关系),也可以是多重的 (multiple),即一对多关系 (one-to-many relationship, 比如某人与其孩子之间的关系、某人与其信用卡账户之间的关系)。此外,还可以有相互 (reciprocal) 关系,即反向关系 (inverse relationship, 比如甲是乙的孩子,乙就是甲的父亲或母亲)。

點選某个实体之后,就可以为其添加属性了。选中实体后,点击编辑器面板右下方的 Add Attribute 按钮。(或者按住这个按钮不放,然后从 Add Attribute、Add Relationship 与 Add Fetched Property 中进行选择。)每个属性都有名称及数据类型,这和定义实例变量时是相似的。

关系提供了指向其他对象的指针。当编辑器处于对象图风格时,开发者可以按住 Ctrl 键,以拖曳鼠标的方式在实体之间建立关系。通过箭头,我们可以看出项目中各实体之间的关系。

虽说 Core Data 对关系型数据库提供了非常强大的支持,但是在最简单的情况下,我们可以只创建一个实体,并且不创建任何关系。大多数 iOS 应用程序都不需要太过复杂的实体结构。对于表格视图与集合视图来说,使用包含 section 属性的平面数据库就足够了。

若想构建图 12-1 中的模型,我们需要创建 Person 实体,并添加 7 个属性: emailaddress、gender、givenname、middleinitial、occupation、surname 及 section。每个属性的类型都要设为 String。

### 12.2.3 构建 NSManagedObject 的子类

创建好实体的定义之后,把修改过的数据保存到数据模型文件中。在屏幕左侧的边栏里选定某个实体,然后点击 Xcode 的 Editor > Create NSManagedObject Subclass 菜单项。选中数据模型以及想要托管的实体(可以选定多个实体)。把待生成的文件保存到项目的文件夹

中，同时还要指定这个类将要添加到项目的哪一个组里，然后点击 Create 按钮。Xcode 会根据开发者所提供的实体描述信息来生成类文件。下面就是由 Xcode 自动生成的 Person 类：

```
@interface Person : NSManagedObject

@property (nonatomic, strong) NSString *section;
@property (nonatomic, strong) NSString *emailaddress;
@property (nonatomic, strong) NSString *gender;
@property (nonatomic, strong) NSString *givenname;
@property (nonatomic, strong) NSString *middleinitial;
@property (nonatomic, strong) NSString *occupation;
@property (nonatomic, strong) NSString *surname;

@end

@implementation Person

@dynamic section;
@dynamic emailaddress;
@dynamic gender;
@dynamic givenname;
@dynamic middleinitial;
@dynamic occupation;
@dynamic surname;

@end
```

每个属性 (attribute) 都对应于一个字符串型的特性 (property)。如果你使用了其他的属性类型，那么生成的特性类型也会随之改变 (比如，可能会生成 NSDate、NSNumber 或 NSData 型的特性)。假如添加了一对多关系，那么就会生成 NSSet 型的特性。@dynamic 指令可以在程序运行的时候创建与特性相关的访问器。

## 12.3 创建上下文

在 Core Data 中，实体提供了描述信息。而托管对象则是根据实体规范创建出来的类实例。这些实例都创建自 NSManagedObject 类，表示数据库中的相关记录。

Core Data 对象位于托管对象上下文之中。这些上下文都是 NSManagedObjectContext 的实例，每个上下文都表示应用程序中的某一块对象空间。本章只采用一个 NSManagedObjectContext，如果你的程序比较复杂，那么可能会用到多个上下文，这通常是为了使多个线程能够同时访问 Core Data 而设计的。

在单一上下文的范例中，开发者启动程序时就会创建 NSManagedObjectContext，并用这个上下文来执行数据获取请求，以便从数据库中获取数据。使用上下文的时候，首先要把应用程序仓 (application bundle) 中创建的模型加载进来。这里不需要指定名称：

```
// Init the model
NSManagedObjectModel *managedObjectModel =
    [NSManagedObjectModel mergedModelFromBundles:nil];
```

接下来, 创建存储区协调器并将其和应用程序沙盒内的某个文件 (也就是某个数据存储区) 相连。在程序中, 协调器负责处理托管对象模型与本地文件之间的关系。开发者提供的 URL 应该指明保存数据所用的文件。下面这段代码使用 `NSSQLiteStoreType` 类型的数据库, 也就是说, 它会创建一份使用标准 SQLite 二进制格式的文件:

```
// Create the store coordinator
NSPersistentStoreCoordinator *persistentStoreCoordinator =
    [[NSPersistentStoreCoordinator alloc]
     initWithManagedObjectModel:managedObjectModel];

// Connect to the data store (on disk)
NSURL *url = [NSURL fileURLWithPath:dataPath];
if (![persistentStoreCoordinator
    addPersistentStoreWithType:NSSQLiteStoreType
    configuration:nil URL:url options:nil error:&error])
{
    NSLog(@"Error creating persistent store coordinator: %@",
          error.localizedDescription);
    return;
}
```

最后, 创建上下文, 并把它的 `persistentStoreCoordinator` 属性设置成刚才创建的那个协调器:

```
// Create a context and assign to the context property
_context = [[NSManagedObjectContext alloc] init];
_context.persistentStoreCoordinator = persistentStoreCoordinator;
```

## 12.4 添加数据

开发者可以用 `NSEntityDescription` 类向上下文中插入新对象。这样的话, 我们就可以用新的数据记录来填充数据存储文件了。插入对象的时候, 要提供实体名称, 以及当前正在操作的上下文:

```
// Create new object
- (NSManagedObject *)newObject
{
    NSManagedObject *object = [NSEntityDescription
        insertNewObjectForEntityForName:_entityName
        inManagedObjectContext:_context];

    return object;
}
```

上述方法会返回新建的 `NSManagedObject`, 以供开发者操作。获得这个新的托管对象之后, 我们就可以根据自己的需要来修改它了, 修改完之后, 应该将其存入上下文:

```
// Save
- (BOOL)save
```

```

{
    NSError __autoreleasing *error;
    BOOL success;
    if (!(success = [_context save:&error]))
        NSLog(@"Error saving context: %@", error.localizedDescription);
    return success;
}

```

常见的用法是这样的：首先创建一个或多个新对象，然后设置其属性，最后保存。我们可以用下面这段代码来调用上述方法，以便在数据库中插入新的 `Person` 实体：

```

Person *person = (Person *)[dataHelper newObject];
person.givenname = @"Chris";
person.surname = @"Zahn";
person.section = [[person.surname substringFromIndex:0] substringToIndex:1];
person.occupation = @"Editor";
[dataHelper save]

```

请注意，`section` 属性是根据 `surname` 属性计算出来的。在比较简单的 iOS 程序里，我们基本上都会添加 `section` 这样的属性，以便使 Core Data 能够依照数据之间的共性对其分组。属性的名称并不重要，到时候只是把它作为参数传给相关方法。笔者之所以选了 `section` 这个名字，是因为它好认而且好记。有高端需求的开发者可能会编写一个方法，用来实现自己的分组标准，而不是像本例这样直接使用硬代码。

上面这段代码会按照姓氏的首字母来分组。如果想根据其他属性分组，可以逐个修改每个对象的 `section`，依照那个属性来决定 `section` 的值，也可以在执行数据获取请求时不传入 `section`，而是传入另外一个属性。通过这些灵活的做法，我们可以把按姓氏首字母分组改成按职业分组。本章稍后会讨论如何获取及查询 Core Data 存储区中的数据。

不要把 iOS 的 `section`（也就是表格视图和集合视图所用的区段）和 `sorting` 相混淆，后者是 Core Data 中的另外一个概念。`section` 是把一批对象分成若干组，而 `sorting` 则决定了每一组里的对象之间应该怎样排序。

## 查看数据文件

如果在模拟器里运行上述代码，很容易就能看到由 Core Data 所创建的 SQLite 文件。打开模拟器文件夹（`~/Library/Application Support/iPhone Simulator/Firmware/Applications`，其中 *Firmware* 是指当前的固件版本，比如 7.0），然后打开与本应用程序相对应的子文件夹。

`Documents` 目录中的 SQLite 文件（具体位置取决于创建持久化存储区时所用的 URL）里面包含了一份数据库，用以代表我们刚才创建的那些数据。执行 `sqlite3` 这个命令行工具，然后用 `.dump` 操作来查看其内容：

```

% sqlite3 Person.sqlite
SQLite version 3.7.13 2012-07-17 17:46:21
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .dump
PRAGMA foreign_keys=OFF;

```

```

BEGIN TRANSACTION;
CREATE TABLE ZPERSON ( Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER, Z_OPT INTEGER,
ZEMAILADDRESS VARCHAR, ZGENDER VARCHAR, ZGIVENNAME VARCHAR, ZMIDDLEINITIAL VARCHAR,
ZOCCUPATION VARCHAR, ZSECTION VARCHAR, ZSURNAME VARCHAR );
INSERT INTO "ZPERSON" VALUES(1,1,1,'ChristopherLRobinson@foomail.com','male','Christop
her','L','Home care aide','C','Robinson');
INSERT INTO "ZPERSON" VALUES(2,1,1,'NicholasJGrant@spambob.com','male','Nicholas','J',
'Steadicam operator','N','Grant');
INSERT INTO "ZPERSON" VALUES(3,1,1,'JosephJTreece@spambob.
com','male','Joseph','J','Shoe machine operator','J','Treece');
INSERT INTO "ZPERSON" VALUES(4,1,1,'HelenEShaffer@dodgit.
com','female','Helen','E','Coin vending and amusement machine servicer
repairer','H','Shaffer');
CREATE TABLE Z_PRIMARYKEY (Z_ENT INTEGER PRIMARY KEY, Z_NAME VARCHAR, Z_SUPER INTEGER,
Z_MAX INTEGER);
INSERT INTO "Z_PRIMARYKEY" VALUES(1,'Person',0,3000);
CREATE TABLE Z_METADATA (Z_VERSION INTEGER PRIMARY KEY, Z_UUID VARCHAR(255), Z_PLIST
BLOB);
INSERT INTO "Z_METADATA" VALUES(1,'85E928DB-1464-4C3B-BCEA-
9277B8817A04',X'62706C6973743030D601020304050607090A0D0E0F5F101E4E5353746F72654D6F646
56C56657273696F6E4964656E7469666696572735F101D4E5350657273697374656E63654672616D65776F
726B56657273696F6E5F10194E5353746F72654D6F64656C56657273696F6E4861736865735B4E5353746
F7265547970655F10125F4E534175746F56616375756D4C6576656C5F10204E5353746F72654D6F64656C
56657273696F6E48617368657356657273696F6EA1085011019AD10B0C56506572736F6E4F1020D261E38
54795D61A5D69048846ECC3DCFEAC4861D9FCD1540A071C875FE89EA95653514C69746551321003081536
56727E93B6B8B9BCBFC6E9F0F200000000000010100000000000010000000000000000000000000000
000F4');
COMMIT;
sqlite> .quit
%

```

上面有很多 SQL 表格定义语句，它们用来保存每个对象的信息，另外还有一些 insert 命令，用来存放程序代码所构建的实例。虽说我们不应该直接用 sqlite3 来操作 Core Data 存储区，但它能使我们看到 Core Data 在底层所执行的操作。

## 12.5 查询数据库

我们可以通过执行 `NSFetchRequest`（获取请求）来从数据库中获取对象。`NSFetchRequest` 描述了选取对象时所依从的标准。开发者把它传给 Core Data，并用它来初始化 `NSFetchedResultsController`，以便存放获取结果，执行完 `NSFetchRequest` 之后，系统将把符合标准的对象放到 `NSFetchedResultsController` 里面的某个数组中。本例的 `fetchItemsMatching` 方法会把获取到的结果存放在名为 `_fetchedResultsController` 的实例变量中，该变量与 `CoreDataHelper` 类中的属性相关联：

```

- (void)fetchItemsMatching:(NSString *)searchString
    forAttribute:(NSString *)attribute
    sortingBy:(NSString *)sortAttribute
{
    // Build an entity description

```

```

NSEntityDescription *entity = [NSEntityDescription
    entityForName:_entityName inManagedObjectContext:_context];

// Init a fetch request
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];

fetchRequest.entity = entity;
[fetchRequest setFetchBatchSize:0];

// Apply an ascending sort for the items
NSString *sortKey = sortAttribute ? : _defaultSortAttribute;
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
    initWithKey:sortKey ascending:YES selector:nil];
NSArray *descriptors = @[sortDescriptor];
fetchRequest.sortDescriptors = descriptors;

// Optional setup predicate
if (searchString && attribute) fetchRequest.predicate =
    [NSPredicate predicateWithFormat:@"%K contains[cd] %@",
        attribute, searchString];

// Perform the fetch
NSError __autoreleasing *error;
_fetchedResultsController = [[NSFetchedResultsController alloc]
    initWithFetchRequest:fetchRequest managedObjectContext:_context
    sectionNameKeyPath:@"section" cacheName:nil];
if (![ _fetchedResultsController performFetch:&error])
    NSLog(@"Error fetching data: %@", error.localizedDescription);
}

```

## 12.5.1 配置 NSFetchedRequest

NSFetchRequest 描述了开发者想要如何搜索数据。首先要根据给定的实体名称取得该实体的描述信息。对于 Person 实体来说，这个实体名称就是 @"Person"。描述信息指明了待搜索的数据类型。

新建 NSFetchedRequest 之后，我们应该把刚才获取到的实体描述信息 (NSEntityDescription) 设置给它，并设置 fetchBatchSize。如果 fetchBatchSize 的值是 0，那么表示一次获取完，不分批处理。若想分批次获取，则将 fetchBatchSize 设为正值。

每个 NSFetchedRequest 必须包含至少一个排序描述符。上一节里的那个方法会依照 sortKey，将获取到的对象按升序 (ascending:YES) 排列。与实体名称一样，sortKey 也是个字符串 (比如，那个例子所用的字符串是 @"surname")。开发者需要把包含描述符的数组设置给 NSFetchedRequest 的 sortDescriptors 属性。

NSFetchRequest 可以用谓词来过滤搜索结果，使其中只包含符合某些规则的内容。如果调用者提供了 searchString 及 attribute 参数，那么 fetchItemsMatching 方法就会创建出 *attribute contains[cd] searchString* 形式的谓词。



这种谓词在匹配文本的时候，不区分大小写。contains 后面的 [cd] 表示匹配的时候既不区分大小写，也不区分附加符号（diacritic）。附加符号是与字母相伴的小标记，比方说表示变音的两个点（umlaut，¨），或是西班牙语字母 n 上方的浪纹线（tilde，~）。

在谓词中，%@ 格式表示字符串值，比方说，我们可以用它来表示 searchString 参数中的字符串。而 %K 则表示实体中某个属性的值。假如在本该使用 %K 的地方使用了 %@，那么 'surname' contains[cd] 'u' 这个谓词就总是 true，因为 surname 的第二个字母是 u。%K 的意思是匹配属性的值，而不是匹配这个属性的名字。

如果要执行更为复杂的查询，那么可以配置复合谓词（compound predicate）。复合谓词能够以 AND、OR 及 NOT 等标准的逻辑操作符来组合简单的谓词。NSCompoundPredicate 类会从单个谓词中构建出复合谓词。此外，也可以不使用 NSCompoundPredicate 类，而是直接把 AND、OR 及 NOT 等记法嵌入简单的 NSPredicate 文本里。



**提示** 谓词是一套能够过滤数据并搜索数据的强大机制。苹果公司的《Predicate Programming Guide》详细地解释了如何创建及使用谓词。请参阅：<https://developer.apple.com/library/ios/DOCUMENTATION/Cocoa/Conceptual/Predicates/predicates.html>。

### 12.5.2 执行数据获取操作

我们要针对每一种查询操作新建与之对应的 NSFetchedResultsController，然后用 NSFetchedRequest、NSManagedObjectContext 及 sectionNameKeyPath 来初始化它。sectionNameKeyPath 参数的值可以设为 @"section"，只要对象里定义了名为 section 的属性就行。通常来说，上面这些需求都不难办到。

控制器的初始化方法里面，有个名为 name 的参数，用来表示缓存。获取数据的时候，系统要把数据安排到各个区段的不同位置上，而缓存则可以减少这一过程的开销。如果数据没有变化，那么下次获取的时候，直接从缓存里取就可以了，这样能够降低程序运行过程中的数据获取开销。缓存名称可以任意选取。如果不想使用缓存功能，那么将其设为 nil，如果想使用，那就提供一个字符串。为了避免与修改 NSFetchedRequest 有关的一些错误，fetchItemsMatching 方法把这个参数设成了 nil。

最后，调用 performFetch 方法，执行数据获取操作。如果成功，那么此方法就返回 true。若是失败，则会修改由调用者以引用形式传入的 error 参数，使其获知该操作为何出错。

获取操作是同步执行的。当该方法返回之后，开发者就可以直接通过 NSFetchedResultsController 的 fetchedObjects 属性获取到对象数组了。下面这段代码演示了如何用 fetchItemsMatching 来获取数据。它会根据文本框中的字符串，把姓氏之中含有该字符串的 Person 对象都找出来，并将其列在文本视图里面。

```
- (void)list
{
    if (!textField.text.length) return;
```

```

[dataHelper fetchItemsMatching:textField.text
    forAttribute:@"surname" sortBy:@"surname"];
NSMutableString *string = [NSMutableString string];
for (Person *person in dataHelper.fetchedResultsController.fetchedObjects)
{
    NSString *entry = [NSString stringWithFormat:@"%s, %s %s: %s\n",
        person.surname, person.givenname,
        person.middleinitial, person.occupation];
    [string appendString:entry];
}
textView.text = string;
}

```

## 12.6 移除对象

从平面数据库中移除对象是相当简单的。只需令上下文把对象删掉，然后保存上下文即可。下面两个方法分别用来删除数据库中的单个对象及全部对象：

```

// Delete one object
- (BOOL)deleteObject:(NSManagedObject *)object
{
    [self fetchData];
    if (!_fetchedResultsController.fetchedObjects.count) return NO;
    [_context deleteObject:object];
    return [self save];
}

// Delete all objects
- (BOOL)clearData
{
    [self fetchData];
    if (!_fetchedResultsController.fetchedObjects.count) return YES;
    for (NSManagedObject *entry in
        _fetchedResultsController.fetchedObjects)
        [_context deleteObject:entry];
    return [self save];
}

```

如果对象之间还有关系，那么移除起来可能会稍微复杂一点。Core Data 在写入数据之前，必须确保内部一致性，若无法确保一致，则会抛出错误。包含交叉引用的数据模型，删除起来比较复杂。在某些数据模型中，开发者必须先把即将失效的那些引用移走，然后才能把对象从持久化存储区里正确地删掉。若是不清理那些引用，则把对象删除之后，其他一些对象会指向已删除的数据，从而产生无法预料的错误。

为了避免这一问题，我们可以在 Data Model Inspector 界面中设置删除规则。删除规则用来决定在相关的对象即将移除时，程序应该对这个操作做出何种反应。如果将规则设为 Deny，那么只要还有其他对象与本对象相连，开发者就不能删除该对象。如果把规则设为 Nullify，那么在删除本对象之前，系统会先把反向的关系清空。Cascade 规则会把本对象以

及该对象通过这条关系所指向的目标对象全部删掉。比方说,通过 Cascade 规则,我们可以把整个部门连同其中的所有成员一并删除。No Action 规则会确保关系所指向的目标对象不受影响,即便那些对象还有指向本对象的反向关系,开发者也依然能够把本对象删掉。

如果 Xcode 发现了非相互关系 (nonreciprocal relationship, 单向关系), 就会给出警告。我们应该避免不平衡的单向关系, 以便简化编码工作, 同时也能更好地维护内部一致性。若是无法避免单向关系, 则应在编写删除方法的时候把它们考虑进来。

## 12.7 解决方案: 用 Core Data 来充当表格的数据源

Core Data 与 iOS 的表格视图结合得相当紧密。NSFetchedResultsController 类包含了一些特性, 可以简化 Core Data 对象与表格数据源的集成过程。大家在下面几小节里会看到: NSFetchedResultsController 类中的许多属性和方法, 从一开始就是为支持表格而设计的。

### 12.7.1 访问索引路径

NSFetchedResultsController 类提供了对象与索引路径之间的双向查询。开发者可以调用 `objectAtIndexPath:` 从对象数组中根据索引路径来获取对象, 也可以调用 `indexPathForObject:` 根据对象来查询索引路径。这两个方法既适用于分区的表格 (sectioned table), 也适用于不分区的普通表格 (flat table) —— 只用一个区段来显示所有数据的表格。

### 12.7.2 `sectionNameKeyPath` 属性

`sectionNameKeyPath` 属性可以把托管对象的属性与区段的名称相关联。该属性用来决定每个托管对象应该属于哪个区段。开发者在任何时间都可以直接设置此属性, 另外也可以在初始化 NSFetchedResultsController 的时候配置它。

解决方案 12-1 采用名为 `section` 的属性来分区, 不过开发者也可以把 `sectionNameKeyPath` 设置成其他属性的名称。本例会把每个对象的 `section` 属性设置成 `surname` 的首个字符。如果要使用不分区的普通表格, 那么把 `sectionNameKeyPath` 设为 `nil`。

### 12.7.3 获取每个区段内的对象

通过控制器的 `sections` 属性, 我们可以获取与每个区段相对应的小组。该属性返回一系列区段, 每个区段里面都保存了若干托管对象, 这些托管对象的 `section` 属性都包含同一个字母。

由 `sections` 属性所返回的每个区段, 都实现了 NSFetchedResultsControllerSectionInfo 协议。此协议可以提供该区段的 `objects` (对象)、`numberOfObjects` (对象个数)、`name` (名称) 及 `indexTitle`。所谓 `indexTitle`, 指的是在表格右侧的快速索引列表里面, 与该区

段相对应的那个标题。

## 12.7.4 sectionIndexTitles 属性

NSFetchedResultsController 的 sectionIndexTitles 属性, 会根据获取到的数据所在的区段, 生成一份包含各区段标题的列表。对于解决方案 12-1 来说, 这份数组里面的每个标题都是一个字母。默认情况下, 系统会根据现有各区段的名称来生成这份列表。

还有两个实例方法, 分别是 sectionIndexTitleForSectionName: 和 sectionIndexTitleAtIndex:, 开发者可以通过它们来查询区段标题。第一个方法会根据区段的名称返回区段的标题, 第二个方法会根据区段的标题来查询其编号。如果你想使用的区段标题与系统根据 sectionNameKeyPath 所拟定的标题不一样, 那么可以覆写这些方法。

## 12.7.5 Core Data 与表格之间的紧密结合

从上面讲到的属性及方法, 我们可以看出: NSFetchedResultsController 实例非常适合与表格相搭配。解决方案 12-1 列出了表格的所有标准方法, 这些方法都是针对 Core Data 而编写的。大家可以发现, 创建及管理区段所用的每个方法都很精练。由于我们借助了系统内置的 Core Data 特性, 所以每个方法只需一两行代码就能写好。Core Data 会直接处理区段的创建及访问事宜。fetchItemsMatching 方法在初始化 NSFetchedResultsController 的时候, 只需向它提供划分区段所参照的属性就可以了。Core Data 会完成其余的工作。

图 12-2 列出了由解决方案 12-1 所构建的界面。这是个功能完备的表格, 每个区段都设置了头部, 而且表格右侧还有浮动的索引。

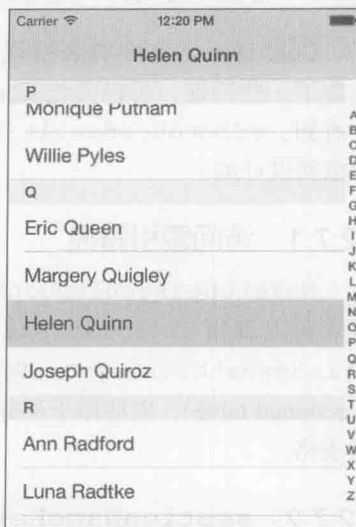


图 12-2 解决方案 12-1 用极少量的代码创建出了功能完备的表格。Core Data 会为表格的所有特性提供支持, 包括单元格的内容、每个区段的头部以及索引



**提示** 如果运行完本章的某条解决方案之后又要运行本章的另外一个解决方案, 需要先重置模拟器, 或是把 Hello Word 程序从设备中删掉, 因为本章所有的解决方案都使用同一份数据库文件 (Person.sqlite), 该文件会留在 Documents 文件夹中。

### 解决方案 12-1 用 Core Data 构建分区的表格

```
#pragma mark Data Source
// Number of sections
- (NSInteger)numberOfSectionsInTableView:
    (UITableView *)tableView
{
```

```

        return dataHelper.fetchedResultsController.sections.count;
    }

    // Rows per section
    - (NSInteger)tableView:(UITableView *)tableView
        numberOfRowsInSection:(NSInteger)section
    {
        id <NSFetchedResultsController> sectionInfo =
            dataHelper.fetchedResultsController.sections[section];
        return sectionInfo.numberOfObjects;
    }

    // Return the title for a given section
    - (NSString *)tableView:(UITableView *)aTableView
        titleForHeaderInSection:(NSInteger)section
    {
        NSArray *titles = [dataHelper.fetchedResultsController
            sectionIndexTitles];
        if (titles.count <= section)
            return @"Error";
        return titles[section];
    }

    // Section index titles
    - (NSArray *)sectionIndexTitlesForTableView:
        (UITableView *)aTableView
    {
        return [dataHelper.fetchedResultsController
            sectionIndexTitles];
    }

    // Populate a cell for the index path
    - (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
    {
        UITableViewCell *cell =
            [tableView dequeueReusableCellWithIdentifier:
                @"cell" forIndexPath:indexPath];
        Person *person =
            (Person *) [dataHelper.fetchedResultsController
                objectAtIndex:indexPath:indexPath];
        cell.textLabel.text = person.fullname;

        return cell;
    }

#pragma mark Delegate
    - (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
    {
        // When a row is selected, update title accordingly
    }

```

```

    Person *person =
        (Person *)[dataHelper.fetchedResultsController
            objectAtIndex:indexPath:indexPath];
    self.title = person.fullname;
}

```

### 获取解决方案代码

访问 <https://github.com/erica/iOS-7-Cookbook> 网页，并打开“C12 Core Data”文件夹，即可找到与本章中的解决方案相对应的完整范例项目。

## 12.8 解决方案：用 Core Data 实现表格的搜索功能

Core Data 的存储区能够同 NSPredicate 高效地结合起来。在创建 NSFetchedRequest 的时候，可以通过谓词实现过滤，使得系统只把符合谓词规则的对象选出来。向 NSFetchedRequest 中添加谓词，即可限定获取到的结果，使其中只包含与谓词相匹配的对象。解决方案 12-2 利用本章早前提到过的谓词，给表格视图添加了搜索功能。

用户可以在表格中输入字符串，把姓氏中包含该字符串的人找出来。表格顶部是搜索栏，如果其中的文本有变化，那么它的委托就会收到 searchBar:textDidChange: 回调。而这个回调方法又会以用户所输的字符串为标准，重新获取数据。

只需要对解决方案 12-1 做下列几处修改，就可以使表格支持搜索功能了：

- 在 loadView 方法中添加 UISearchDisplayController，并且令 viewDidLoad: 方法把搜索栏滚动到可视范围之外。
- 修改 sectionIndexTitlesForTableView: 方法，把表示搜索功能的图标也添加到索引里面，同时修改 tableView:sectionForSectionIndexTitle:atIndex: 方法，使得表格可以把 searchController 滚动到可视范围之内。
- 只要搜索栏的内容有变化，它的委托方法就会获取新的结果。这个方法会提交新的 Core Data 请求，以便重新获取数据，并用新的数据填充表格视图。

经过这些修改之后，我们就可以创建出带有搜索框的表格了，这种表格可以响应用户所输入的查询内容。从解决方案 12-1 及解决方案 12-2 我们可以看出，只需编写极少的代码，就可以把表格视图与 Core Data 集成起来。

### 解决方案 12-2 使用谓词过滤获取到的数据

```

// Section index titles plus search
- (NSArray *)sectionIndexTitlesForTableView:
    (UITableView *)aTableView
{
    if (aTableView == searchController.searchResultsTableView)
        return nil;
    return [[NSArray arrayWithObject:UITableViewIndexSearch]
        arrayByAddingObjectsFromArray:

```

```

        [dataHelper.fetchedResultsController sectionIndexTitles]];
    }

    // Allow scrolling to search bar
    - (NSInteger)tableView:(UITableView *)tableView
        sectionForSectionIndexTitle:(NSString *)title
        atIndex:(NSInteger)index
    {
        if (title == UITableViewIndexSearch)
        {
            [self.tableView scrollRectToVisible:
                searchController.searchBar.frame animated:NO];
            return -1;
        }
        return [dataHelper.fetchedResultsController.sectionIndexTitles
            indexOfObject:title];
    }

    // Return a cell specific to the table being shown
    - (UITableViewCell *)tableView:(UITableView *)aTableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
    {
        [aTableView registerClass:[UITableViewCell class]
            forCellReuseIdentifier:@"cell"];
        UITableViewCell *cell =
            [aTableView dequeueReusableCellWithIdentifier:@"cell"
                forIndexPath:indexPath];
        Person *person = [dataHelper.fetchedResultsController
            objectAtIndex:indexPath];
        cell.textLabel.text = person.fullname;
        return cell;
    }

    // Handle cancel by fetching all data
    - (void)searchBarCancelButtonClicked:(UISearchBar *)aSearchBar
    {
        aSearchBar.text = @"";
        [dataHelper fetchData];
    }

    // Handle search field update by fetching matching entries
    - (void)searchBar:(UISearchBar *)aSearchBar
        textDidChange:(NSString *)searchText
    {
        [dataHelper fetchItemsMatching:aSearchBar.text
            forAttribute:@"surname" sortingBy:nil];
    }

    // Set up search and Core Data
    - (void)loadView
    {

```

```

self.view = [[UIView alloc] init];
self.tableView = [[UITableView alloc] init];
self.view.backgroundColor = [UIColor whiteColor];

// Create a search bar
UISearchBar *searchBar =
    [[UISearchBar alloc] initWithFrame:
        CGRectMake(0.0f, 0.0f, 0.0f, 44.0f)];
searchBar.autocorrectionType = UITextAutocorrectionTypeNo;
searchBar.autocapitalizationType = UITextAutocapitalizationTypeNone;
searchBar.keyboardType = UIKeyboardTypeAlphabet;
searchBar.delegate = self;
self.tableView.tableHeaderView = searchBar;

// Create the search display controller
searchController = [[UISearchDisplayController alloc]
    initWithSearchBar:searchBar contentsController:self];
searchController.searchResultsDataSource = self;
searchController.searchResultsDelegate = self;

// Establish Core Data
dataHelper = [[CoreDataHelper alloc] init];
dataHelper.entityName = @"Person";
dataHelper.defaultSortAttribute = @"surname";

// Check for existing data
BOOL firstRun = !dataHelper.hasStore;

// Set up Core Data
[dataHelper setupCoreData];
if (firstRun)
    [self initializeData];

[dataHelper fetchData];
[self.tableView reloadData];
}

// Hide the search bar
- (void)viewDidAppear: (BOOL)animated
{
    [super viewDidAppear:animated];
    NSIndexPath *path = [NSIndexPath indexPathForRow:0 inSection:0];
    [self.tableView scrollToRowAtIndexPath:path
        atScrollPosition:UITableViewScrollPositionTop animated:NO];
}

```

## 12.9 解决方案：为 Core Data 表格视图添加编辑功能

读者已经看到了如何将表格视图与静态数据相结合。现在我们来增强这项技术。解决方



案 12-3 演示了怎样向表格界面里添加编辑功能，并把编辑后的数据存入相应的 Core Data 存储区中。

本条解决方案里的大部分代码，读者都应该比较熟悉了，因为第 9 章在实现基本的表格编辑功能时所用的代码与此相似。用户可以点击“+”按钮来添加新的行，也可以通过横向扫屏（Swiping）手势或进入编辑模式来删除现有的行。剩下的功能，比如表格搜索与区段索引等，都保持不变。

本条解决方案所用的新数据，都是从一系列虚构的联系人信息中加载进来的，感谢 fakenamgenerator.com 网站提供此功能。用户点击“+”按钮之后，程序会从这些信息中随机选取一个名称，并将其添加到数据库中。

如果要把表格编辑功能添加到实际的 Core Data 程序里，那么还应该对范例代码做出一定的修改。比方说，我们可以考虑为自己的表格添加撤销/重做功能，也可以给用户提供控制数据个数的功能，还可以考虑通过控制器的委托方法来更新数据。

### 12.9.1 添加撤销/重做功能

有了 Core Data 之后，想为表格添加撤销和重做功能就简单得多了。由于 Core Data 会自动支持这些操作，所以开发者只需编写非常少的代码。创建 Core Data 环境的时候，可以为其配置撤销管理器：

```
_context = [[NSManagedObjectContext alloc] init];
_context.persistentStoreCoordinator = persistentStoreCoordinator;
_context.undoManager = [[NSUndoManager alloc] init];
_context.undoManager.levelsOfUndo = 999;
```

与实现其他的撤销/重做功能时一样，如果主控制器出现在屏幕中，那么它也必须成为第一响应者才行。为此，我们需要实现下面三个标准的方法：canBecomeFirstResponder、viewDidAppear: 及 viewWillDisappear:。第一个方法返回 YES，以响应系统的查询；第二个方法会在控制器视图刚出现的时候立刻令其成为第一响应者；第三个方法会在控制器视图离开屏幕的时候令其放弃第一响应者的身份。

```
- (BOOL)canBecomeFirstResponder
{
    return YES;
}

- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    [self becomeFirstResponder];

    if ([dataHelper numberOfEntities] == 0) return;

    // Hide the search bar
    NSIndexPath *path = [NSIndexPath indexPathForRow:0 inSection:0];
    [self.tableView scrollToRowAtIndexPath:path
                      atScrollPosition:UITableViewScrollPositionTop animated:NO];
}
```

```

    }

    - (void)viewWillDisappear:(BOOL)animated
    {
        [super viewWillDisappear:animated];
        [self resignFirstResponder];
    }

```

请注意，只有当表格中包含至少一个数据项时，程序才会把搜索栏滚动到屏幕范围之外。解决方案 12-2 是不需要这个措施的。由于用户可以通过添加及删除功能来直接控制表格内容，所以表格完全有可能以不包含任何数据的形式出现在屏幕上。

### 12.9.2 创建撤销事务

开发者需要把修改对象所用的 Core Data 语句放在撤销群组 (undo grouping) 之中，以便构成一项撤销事务。修改上下文所用的那些语句，应该出现在 beginUndoGrouping 下方，并放在与之配套的 endUndoGrouping 上方。我们需要用动作名称 (action name) 来描述即将执行的操作。这个动作名称，主要是为晃动撤销功能而设计的 (比方说，程序里可能会出现 “Undo delete?” 字样)。另外，它也有助于阐明代码的意图。

在下面代码中，[dataHelper.context deleteObject:object]; 语句前后的一对花括号纯粹是为了排版美观而添上去的，它们能够凸显出其中的语句是一项事务。

```

// Delete request
if (editingStyle == UITableViewCellEditingStyleDelete)
{
    NSManagedObject *object = [dataHelper.fetchedResultsController
                                objectAtIndex:indexPath];
    NSUndoManager *manager = dataHelper.context.undoManager;
    [manager beginUndoGrouping];
    [manager setActionName:@"Delete"];
    {
        [dataHelper.context deleteObject:object];
    }
    [manager endUndoGrouping];
    [dataHelper save];
}

```

上述代码片段中的 beginUndoGrouping、setActionName: 及 endUndoGrouping 方法调用，可以确保 Core Data 能够反转其中的操作。由于有了 Core Data，所以我们只需编写这几行代码，就可以令应用程序具备一套非常实用的撤销管理系统。请注意，应用程序退出之后，与撤销和重做功能有关的记录就丢失了。因为每次启动程序时，它都会重置这个栈。

### 12.9.3 重新思考编辑功能

在操作由 Core Data 所驱动的表格时，解决方案 12-3 不允许用户重新排列各行的顺序。这是因为程序在执行数据获取请求的时候会把数据排列好，无须用户参与。解决方案 12-3 的

tableView:canMoveRowAtIndexPath: 方法会把返回值写成硬代码 NO。我们当然可以再添加一个表示行位置的属性,但是大部分情况下都不需要此功能。所以,解决方案 12-3 会展示一种比较常见的用法。

### 解决方案 12-3 修改表格的编辑功能,使之与 Core Data 相搭配

```
// Update items in the navigation bar
- (void)setBarButtonItems
{
    // Expire any ongoing operations
    if (dataHelper.context.undoManager.isUndoing ||
        dataHelper.context.undoManager.isRedoing)
    {
        [self performSelector:@selector(setBarButtonItems)
            withObject:nil afterDelay:0.1f];
        return;
    }

    UIBarButtonItem *undo = SYSBARBUTTON_TARGET(
        UIBarButtonItemUndo,
        dataHelper.context.undoManager, @selector(undo));
    undo.enabled = dataHelper.context.undoManager.canUndo;
    UIBarButtonItem *redo = SYSBARBUTTON_TARGET(
        UIBarButtonItemRedo,
        dataHelper.context.undoManager, @selector(redo));
    redo.enabled = dataHelper.context.undoManager.canRedo;
    UIBarButtonItem *add = SYSBARBUTTON(
        UIBarButtonItemAdd, @selector(addItem));

    self.navigationItem.leftBarButtonItems = @[add, undo, redo];
}

// Refetch data
- (void)refresh
{
    // If searching, fetch search results, otherwise all data
    if (searchController.searchBar.text)
        [dataHelper fetchItemsMatching:
            searchController.searchBar.text
            forAttribute:@"surname" sortBy:nil];
    else
        [dataHelper fetchData];
    dataHelper.fetchedResultsController.delegate = self;

    // Reload tables
    [self.tableView reloadData];
    [searchController.searchResultsTableView reloadData];

    // Update bar button items
    [self setBarButtonItems];
}
```

```

    }

    // Respond to section changes
    - (void)controller:(NSFetchedResultsController *)controller
        didChangeSection:(id <NSFetchedResultsSectionInfo>)sectionInfo
        atIndex:(NSUInteger)sectionIndex
        forChangeType:(NSFetchedResultsChangeType)type
    {
        if (type == NSFetchedResultsChangeDelete)
        {
            [self.tableView deleteSections:
                [NSIndexSet indexSetWithIndex:sectionIndex]
                withRowAnimation:UITableViewRowAnimationAutomatic];
        }

        if (type == NSFetchedResultsChangeInsert)
        {
            [self.tableView insertSections:
                [NSIndexSet indexSetWithIndex:sectionIndex]
                withRowAnimation:UITableViewRowAnimationAutomatic];
        }
        sectionHeadersAffected = YES;
    }

    // Respond to item changes
    - (void)controller:(NSFetchedResultsController *)controller
        didChangeObject:(id)anObject
        atIndexPath:(NSIndexPath *)indexPath
        forChangeType:(NSFetchedResultsChangeType)type
        newIndexPath:(NSIndexPath *)newIndexPath
    {
        UITableView *tableView = self.tableView;

        if (type == NSFetchedResultsChangeInsert)
        {
            [tableView insertRowsAtIndexPaths:@[newIndexPath]
                withRowAnimation:UITableViewRowAnimationAutomatic];
        }

        if (type == NSFetchedResultsChangeDelete)
        {
            [tableView deleteRowsAtIndexPaths:@[indexPath]
                withRowAnimation:UITableViewRowAnimationAutomatic];
        }
    }

    // Prepare for updates
    - (void)controllerWillChangeContent:
        (NSFetchedResultsController *)controller
    {
        sectionHeadersAffected = NO;
        [self.tableView beginUpdates];
    }

    // Apply updates
    - (void)controllerDidChangeContent:
        (NSFetchedResultsController *)controller
    {
        [self.tableView endUpdates];
    }

```

```

// Update section headers if needed
if (sectionHeadersAffected)
    [self.tableView reloadData:
        [NSIndexSet indexSetWithIndexesInRange:
            NSRange(0, self.tableView.numberOfSections)]
        withRowAnimation:UITableViewRowAnimationNone];

[self setBarButtonItems];
}

// Only allow editing on the main table
- (BOOL)tableView:(UITableView *)aTableView
    canEditRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (aTableView == searchController.searchResultsTableView) return NO;
    return YES;
}

// No reordering allowed
- (BOOL)tableView:(UITableView *)tableView
    canMoveRowAtIndexPath:(NSIndexPath *)indexPath
{
    return NO;
}

- (void)addItem
{
    // Surround the "add" functionality with undo grouping
    NSUndoManager *manager = dataHelper.context.undoManager;
    [manager beginUndoGrouping];
    {
        Person *person = (Person *)[dataHelper newObject];
        [self setupNewPerson:person];
    }
    [manager endUndoGrouping];
    [manager setActionName:@"Add"];
    [dataHelper save];
}

// Handle deletions
- (void)tableView:(UITableView *)tableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
    forRowAtIndexPath:(NSIndexPath *)indexPath
{
    // delete request
    if (editingStyle == UITableViewCellEditingStyleDelete)
    {
        NSManagedObject *object = [dataHelper.fetchedResultsController
            objectAtIndex:indexPath];
        NSUndoManager *manager = dataHelper.context.undoManager;
        [manager beginUndoGrouping];
        {

```

```

        [dataHelper.context deleteObject:object];
    }
    [manager endUndoGrouping];
    [manager setActionName:@"Delete"];
    [dataHelper save];
}

// Toggle editing mode
- (void)setEditing:(BOOL)isEditing animated:(BOOL)animated
{
    [super setEditing:isEditing animated:animated];
    [self.tableView setEditing:isEditing animated:animated];

    NSIndexPath *path = [self.tableView
        indexPathForSelectedRow];
    if (path)
        [self.tableView deselectRowAtIndexPath:path
            animated:YES];

    [self setBarButtonItems];
}

```

另外，我们也要把数据库中的改动与数据源同步。对于由 Core Data 所驱动的表格来说，这些改动有可能是由用户发起的（比方说通过横向扫屏来删除单元格，通过 + 按钮来添加单元格等），但也有可能是由撤销管理器执行的。我们可以通过委托来订阅 NSFetchedResultsController，以获知是否有数据因为撤销操作而发生变化。当数据有变化时，我们会在 NSFetchedResultsControllerDelegate 协议的委托回调中重新加载数据。

## 12.10 解决方案：由 Core Data 所驱动的集合视图

想要把解决方案 12-3 从表格移植到集合视图上面，是需要花一些工夫的，但是工作量不太大。我们需要做的是：删掉搜索视图控制器、舍弃索引视图、略微更新一下编辑功能，并且把控制器类从表格视图改成集合视图。图 12-3 演示了修改后的效果。集合控制器会显示出与表格相同的数据，而且使得用户能够选定并编辑单元格，同时还具备撤销和重做功能。

首先要重构数据模型。解决方案 12-4 添加了一个新属性，它是一个二进制数据项，名为 imageData。这幅图像是根据每个人的名和姓构建出来的，并以二进制格式存储。有了这个属性，集合视图就可以把每条数据都展示成一幅可供重用的图像了，图像的尺寸会与姓名相符。

所有的数据源方法都需要从表格视图迁移到集合视图。某些方法所需的改动非常少。用于返回区段数量的方法以及用于返回各区段所含条目数量的方法，都会根据集合视图做出相应调整，但是它们的内部实现代码保持不变。

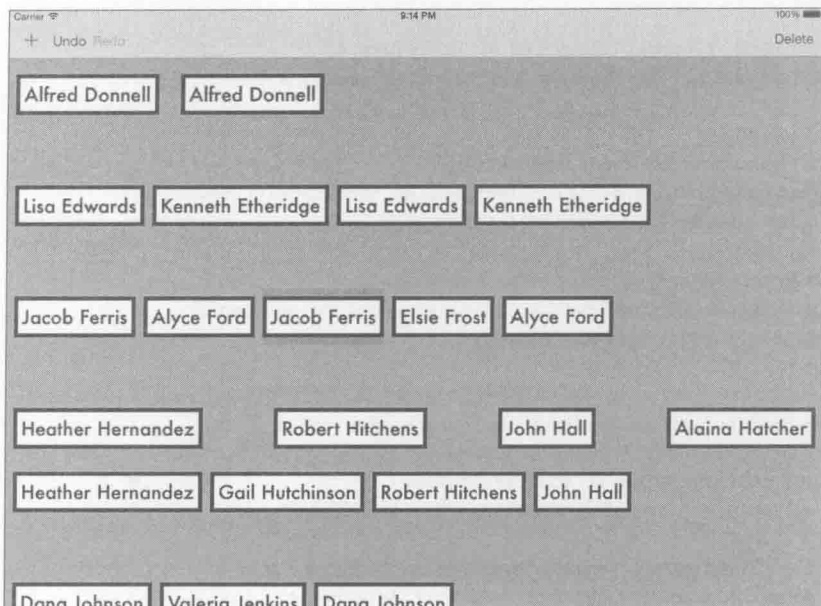


图 12-3 解决方案 12-4 构建了由 Core Data 所驱动的集合视图

其他方法则需要做较大改动。根据路径来提供单元格的方法要彻底更新，因为集合视图是通过单元格来展示图像，而不是像表格视图那样只显示标题及文本。解决方案 12-4 没有包含搜索视图控制器，也没有包含与索引视图及区段头部有关的回调。最后，解决方案 12-4 还添加了一个能够定制单元格尺寸的布局方法，以便使每个单元格视图的大小与其中的图像相符。这个布局方法对于集合视图来说比较重要，但在表格视图里面却用不到。

编辑功能也需要修改，它不再以单元格的动画效果为中心了。所以解决方案 12-4 不需要像表格那样，通过相关的编辑方法来提交用户的删除操作，而是添加独立的 `deleteItem` 方法，该方法与解决方案 12-3 中的 `addItem` 方法是相对应的。

在表格视图中，导航栏右侧的按钮用于切换编辑模式，而在集合视图中，则需改为 `Delete`，以使用户在选中某单元格之后，通过该按钮来删除它。在由表格视图迁移到集合视图的过程中，导航栏里原有的 `Undo` 与 `Redo` 按钮保持不变，支持撤销功能与重做功能的方法也保持不变。

与采用 MVC 范式开发的其他项目一样，本例所需的修改也就这么多了。Core Data 的模型方法和解决方案 12-3 所用的相同。视图部分使用 UIKit 内置的视图。只有控制器部分需要调整，或需要接收其他更新，所以总体来看，MVC 范式使得重构的过程变得比较容易。

#### 解决方案 12-4 由 Core Data 所驱动的集合视图

```
#pragma mark Data Source
// Return the number of sections
- (NSInteger)numberOfSectionsInCollectionView:
    (UICollectionView *)collectionView
```

```

{
    if (dataHelper.numberOfEntities == 0) return 0;
    return dataHelper.fetchedResultsController.sections.count;
}

// Return the number of items per section
- (NSInteger)collectionView:(UICollectionView *)collectionView
    numberOfItemsInSection:(NSInteger)section
{
    id <NSFetchedResultsControllerSectionInfo> sectionInfo =
        dataHelper.fetchedResultsController.sections[section];
    return sectionInfo.numberOfObjects;
}

// This method builds images into collection view cells
- (UICollectionViewCell *)collectionView:
    (UICollectionView *) aCollectionView
    cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
    UICollectionViewCell *cell = [self.collectionView
        dequeueReusableCellWithReuseIdentifier:@"cell"
        forIndexPath:indexPath];
    Person *person = [dataHelper.fetchedResultsController
        objectAtIndex:indexPath:indexPath];
    UIImage *image = [UIImage imageWithData:person.imageData];

    cell.backgroundColor = [UIColor clearColor];
    if (![cell.contentView viewWithTag:IMAGEVIEWTAG])
    {
        UIImageView *imageView =
            [[UIImageView alloc] initWithImage:image];
        imageView.tag = IMAGEVIEWTAG;
        [cell.contentView addSubview:imageView];
    }

    UIImageView *imageView =
        (UIImageView *) (cell.contentView viewWithTag:IMAGEVIEWTAG);
    imageView.frame = CGRectMake(0.0f, 10.0f, image.size.width, image.size.height);
    imageView.image = image;

    cell.selectedBackgroundView = [[UIView alloc] init];
    cell.selectedBackgroundView.backgroundColor =
        [UIColor redColor];

    return cell;
}

// Return the size for layout
- (CGSize)collectionView:(UICollectionView *)collectionView
    layout:(UICollectionViewLayout*)collectionViewLayout
    sizeForItemAtIndexPath:(NSIndexPath *)indexPath

```



```

{
    Person *person = [dataHelper.fetchedResultsController
        objectAtIndex:indexPath:indexPath];
    UIImage *image = [UIImage imageWithData:person.imageData];
    return CGSizeMake(image.size.width, image.size.height + 20.0f);
}

#pragma mark Delegate methods
- (void)collectionView:(UICollectionView *)aCollectionView
    didSelectItemAtIndexPath:(NSIndexPath *)indexPath
{
    [self setBarButtonItems];
}

#pragma mark Editing and Undo
- (void)setBarButtonItems
{
    // Delete requires a selected item
    self.navigationItem.rightBarButtonItem.enabled =
        (self.collectionView.indexPathsForSelectedItems.count != 0);

    // Set up undo/redo items
    UIBarButtonItem *undo =
        SYSBARBUTTON_TARGET(UIBarButtonSystemItemUndo,
            self.dataHelper.context.undoManager, @selector(undo));
    undo.enabled = self.dataHelper.context.undoManager.canUndo;
    UIBarButtonItem *redo =
        SYSBARBUTTON_TARGET(UIBarButtonSystemItemRedo,
            self.dataHelper.context.undoManager, @selector(redo));
    redo.enabled = self.dataHelper.context.undoManager.canRedo;
    UIBarButtonItem *add =
        SYSBARBUTTON(UIBarButtonSystemItemAdd, @selector(addItem));
    self.navigationItem.leftBarButtonItems = @[add, undo, redo];
}

// Refresh the data, update the view
- (void)refresh
{
    [dataHelper fetchData];
    dataHelper.fetchedResultsController.delegate = self;
    [self.collectionView reloadData];
    [self performSelector:@selector(setBarButtonItems)
        withObject:nil afterDelay:0.1f];
}

- (void)controllerDidChangeContent:
    (NSFetchedResultsController *)controller
{
    // Respond to data change from undo controller
    [self refresh];
}

```

```

    }

    // Add a new item
    - (void)addItem
    {
        NSUndoManager *manager = dataHelper.context.undoManager;
        [manager beginUndoGrouping];
        {
            Person *person = (Person *)[dataHelper newObject];
            [self setupNewPerson:person];
        }
        [manager endUndoGrouping];
        [manager setActionName:@"Add"];
        [dataHelper save];
        [self refresh];
    }

    // Delete the selected item
    - (void)deleteItem
    {
        if (!self.collectionView.indexPathsForSelectedItems.count)
            return;

        NSIndexPath *indexPath =
            self.collectionView.indexPathsForSelectedItems[0];
        NSManagedObject *object =
            [dataHelper.fetchedResultsController
             objectAtIndex:indexPath.index];
        NSUndoManager *manager = dataHelper.context.undoManager;
        [manager beginUndoGrouping];
        {
            [dataHelper.context deleteObject:object];
        }
        [manager endUndoGrouping];
        [manager setActionName:@"Delete"];
        [dataHelper save];
        [self refresh];
    }

#pragma mark Setup
- (void)viewDidLoad
{
    [super viewDidLoad];
    [self.collectionView registerClass:
     [UICollectionViewCell class]
     forCellWithReuseIdentifier:@"cell"];

    self.collectionView.backgroundColor =
        [UIColor lightGrayColor];
    self.collectionView.allowsMultipleSelection = NO;
}

```

```

self.collectionView.allowsSelection = YES;

self.navigationItem.leftBarButtonItem =
    UIBarButtonItem(UIBarButtonSystemItemAdd,
        @selector(addItem));
self.navigationItem.rightBarButtonItem =
    UIBarButtonItem(@"Delete", @selector(deleteItem));
self.navigationItem.rightBarButtonItem.enabled = NO;
}

```

## 12.11 小结

操作表格视图与集合视图的时候，Core Data 提供了相当好的支撑技术。它提供了简单易用的模型，这些模型很容易与 UIKit 中的数据源相结合。本章只是初步介绍了 Core Data 的一些能力。其中的解决方案演示了如何设计并实现基本的 Core Data 功能，以使用它来支持托管对象模型。我们看到了怎样定义模型，以及怎样实现数据获取请求。读者也学会了如何添加对象、修改对象、删除对象及保存对象。此外，还学到了怎样使用谓词，以及怎样执行撤销操作。在阅读下一章之前，请回顾以下几个知识点：

- 如果不把表格视图与集合视图同 Core Data 结合起来，那就会错过一种非常优雅的数据生成与控制方式。
- Core Data 的使用范围并不局限于可以滚动的视图，凡是表格状的信息，都可以用 Core Data 来保存。它所提供的关系型数据库解决方案，其能力远远超出了大部分应用程序的需求。
- 开发者总是应该提供撤销及重做功能。就算刚开始觉得用不到，也可以提前把功能做好，以便将来用到的时候直接添加进去。笔者并不是特别欣赏晃动撤销（shake-to-undo）这一特性，但如果界面已经很繁杂了，那么可以考虑用它来向程序中添加一种不使用按钮的撤销功能。
- 谓词是 SDK 中一个很受欢迎的特性。你应该花些时间研究一下怎样构建谓词，以及如何用 NSArray 及 NSSet 等各种对象来构建谓词，而不是仅仅将它与 Core Data 相搭配。
- iCloud 提供了非常优秀的特性，能够把 Core Data 和随处可用的数据结合起来，使得 iOS 的数据能够全面延伸至用户的桌面、设备及云端。以前，Core Data 与 iCloud 结合得不够稳定，而且容易出一些问题，但到了 iOS 7，苹果公司宣称它已经对此做了必要的改进。请读者查阅 UIManagedDocument 类的开发文档，以便深入了解 iCloud 与 Core Data 的集成。
- Core Data 的能力远远不止本章各条解决方案所讲的基本功能。请参阅 Tim Isted 与 Tom Harrington 所著的《Core Data for iOS: Developing Data-Driven Applications for the iPad, iPhone, and iPod touch》（Addison-Wesley 出版），该书详细地探究了 Core Data 及其特性。

## 网络编程基础

iPhone 及 iOS 系列的其他设备都可以联网，它们非常适合通过基于 Web 的服务来获取远程数据。苹果公司为 iOS 平台提供了坚实的基础架构，用以实现各种网络操作以及相关的支持技术。本章讲解与网络编程有关的一些基本技巧，包括：如何测试联网状态、如何从网站下载数据，以及如何处理由 Web 服务所提供的传统形式数据等。

### 13.1 解决方案：判断网络状态

联网的应用程序在与因特网或附近的设备通信时，必须有一条活动的连接。应用程序在发送或接收数据之前，应该知晓是否有活动的连接可供使用。所以，开发者需要判断网络的状态，使得程序可以与用户交流，并告诉他们某些功能为何无法使用。

如果程序在向用户提供下载选项之前不检测网络状态，那么苹果公司拒绝这样的应用程序在 App Store 上架，该策略以后也会延续下去。苹果公司的评审员是经过培训的，他们能够判定程序有没有正确地提示用户，尤其是当网络出错时有没有给出适当的提示。开发者总是应该验证网络的状态，并向用户展示相应的警示信息。

此外，苹果公司还会拒绝那种太耗费数据流量的程序。如果在程序里有大量的数据要流动，比方说语音或视频，那么应该判断当前的网络类型。如果使用的是移动网络（cell network，蜂窝网络），那么应该提供低品质的数据流，若使用 WiFi 网络，则可以提供高品质的数据流。苹果公司不太能够容忍那种在移动网络之下耗费较多流量的程序。大家要知道，在美国，按流量计费的收费方式已经取代了无限流量的收费方式。如果程序消耗的流量太多，那么用户和苹果公司都会对此不满。

iOS 可以测试出是否有某种网络连接可供使用（无论是何种类型的连接）、是否有

WiFi 可供使用，以及是否有手机服务可供使用。目前还没有一种为 App Store 所容许的 API (Application Programming Interface, 应用编程接口) 能够判断出 iPhone 是否可以使用蓝牙连接，不过，开发者可以限定程序只能运行在开启了蓝牙的设备上。此外，开发者在访问数据之前，也无法判断出用户是否处在漫游状态，以及是否使用了可能比较昂贵的移动网络。

System Configuration 框架提供了网络检测函数。在这些函数中，SCNetworkReachabilityCreateWithAddress 可以判断设备是否能够连通某个 IP 地址。解决方案 13-1 用一个简单的范例演示了这项测试。

networkAvailable 方法会判断出设备是否能对外联网，也就是说，设备中是否有能够访问而且处于活动状态的连接。该方法借鉴了苹果公司的范例代码，如果网络可以使用，就返回 YES，否则返回 NO。方法里使用了两个标志，以判断是否有网络可达 (kSCNetworkFlagsReachable)，以及是否不再需要建立网络连接 (kSCNetworkFlagsConnectionRequired)。可能会用到的其他标志还有：

- **kSCNetworkReachabilityFlagsIsWWAN**——判断用户是在使用运营商的无线广域网络 (Wireless Wide area Network, WWAN)，还是在使用本地 WiFi。如果可以使用 WWAN，那么设备就能通过 EDGE、GPRS、LTE 或任意类型的移动网络来联网了。在通过 WWAN 联网时，由于带宽或资费所限，开发者可能需要使用轻量版的资源 (比方说，尺寸较小的图像，或是占用带宽较少的视频)。
- **kSCNetworkReachabilityFlagsConnectionOnTraffic**——该标志表示设备在当前的网络配置下可以连接到给定的地址，但是必须先建立连接。如果要与给定的地址之间进行网络通信，那么设备就会发起连接。
- **kSCNetworkReachabilityFlagsIsDirect**——该标志表示设备与给定地址之间的网络通信是直达还是经由网关送达。

判断网络是否连通的代码，最好能在多台设备上面测试。iPhone 与能够移动上网的 iPad 提供了多种上网途径。这些设备既可以用移动网络来上网，也可以通过 WiFi 上网，所以，当用户使用 WWAN 连接的时候，我们能够确定网络是可用的。

在 iPhone 的“设置”里面切换 WiFi 与移动网络，然后分别测试解决方案 13-1 的代码。测试网络连通状况时，程序可能稍有延迟，所以程序设计者应该考虑到这一点。我们应当把需要执行的检测告诉用户。

SCNetworkReachabilityGetFlags 是一个同步调用，有可能会长时间阻塞线程，尤其是在执行到 DNS (Domain Name System, 域名系统) 查询这一步的时候，如果没有网络连接，更是会阻塞很长时间。在实际的应用程序里，绝不应该在主线程中调用这个方法。如果主线程延时了很长一段时间，那么 iOS 的 watchdog 可能会令程序终止。

我们可以通过 NSOperationQueue 把这个阻塞式的调用从主线程里移走。但是要注意，与 UI 有关的交互操作还是要回到主线程上面执行：

```
[[[NSOperationQueue alloc] init] addOperationWithBlock:
^{
```

```

// blocking call
BOOL networkAvailable = [device networkAvailable];

// UI interaction
[[NSOperationQueue mainQueue] addOperationWithBlock:^(
    textView.text = networkAvailable ? @"Yes" : @"No";
)];
}];

```

开发者可以在共享的应用程序实例上面设置 `networkActivityIndicatorVisible` 属性,用以表明本程序是否在使用网络。如果正在联网,那么状态栏上会有个旋转的指示器。

### 解决方案 13-1 测试网络是否连通

```

SCNetworkReachabilityRef reachability;
SCNetworkConnectionFlags connectionFlags;

- (void)pingReachability
{
    if (!reachability)
    {
        BOOL ignoresAdHocWiFi = NO;
        struct sockaddr_in ipAddress;
        bzero(&ipAddress, sizeof(ipAddress));
        ipAddress.sin_len = sizeof(ipAddress);
        ipAddress.sin_family = AF_INET;
        ipAddress.sin_addr.s_addr =
            htonl(ignoresAdHocWiFi ? INADDR_ANY : IN_LINKLOCALNETNUM);

        reachability = SCNetworkReachabilityCreateWithAddress(
            kCFAllocatorDefault, (struct sockaddr *)&ipAddress);
        CFRetain(reachability);
    }

    // Recover reachability flags
    BOOL didRetrieveFlags = SCNetworkReachabilityGetFlags(
        reachability, &connectionFlags);
    if (!didRetrieveFlags)
        NSLog(@"Error. Could not recover network reachability flags");
}

- (BOOL)networkAvailable
{
    [[UIApplication sharedApplication]
        setNetworkActivityIndicatorVisible:YES];
    [self pingReachability];
    BOOL isReachable =
        (connectionFlags & kSCNetworkFlagsReachable) != 0;
    BOOL needsConnection =
        (connectionFlags & kSCNetworkFlagsConnectionRequired) != 0;
    [[UIApplication sharedApplication]

```

```

        setNetworkActivityIndicatorVisible:NO];
    return (isReachable && !needsConnection) ? YES : NO;
}

```

### 获取解决方案代码

访问 <https://github.com/erica/iOS-7-Cookbook> 网页，并打开“C13 Networking”文件夹，即可找到与本章中的解决方案相对应的完整范例项目。

## 13.2 监测联网状况是否发生变化

应用程序运行的时候，其联网状态可能会改变。如果程序在整个生命期都需要使用数据连接，那么只在刚启动时判断是否联网是不够的。在网络连接断开或建立好的时候，UI 应该做出相应调整，比方说可以禁用或启用某个按钮，也可以向用户展示警告信息。

程序清单 13-1 扩充了 `UIDevice` 类，在名为 `Reachability` 的 `category` 里面，提供了监控联网状况是否发生变化的功能。其中的两个方法，分别用来排定并解除网络状况监视器（`reachability watcher`）。这些监视器会在联网状态发生变化的时候得到通知。这段代码构建了名为 `ReachabilityCallback` 的回调，如果网络状态有变化，那么它就会向监视器对象发送消息。监视器安排当前的运行循环（`run loop`）上面，而且以异步方式执行。检测到变化之后，系统会触发回调函数。

程序清单 13-1 的回调函数又会调用 `reachabilityChanged` 方法，它是我们自定义的委托方法，监视器必须实现该方法。监视器对象收到通知之后，可以查询当前的网络状态。

排定监视器的方法会把委托传给 `scheduleReachabilityWatcher:` 的 `watcher` 参数。解决方案 13-1 所实现的 `reachabilityChanged` 方法非常简单，它只是依照程序清单 13-1 的框架编写出来的。在开发真实的程序时，我们需要修改 GUI 里面与网络有关的功能，以便与当前的联网状况相匹配。如果联网状态变了，应该告知用户，并根据当前状态来更新程序界面。如果程序无法联网，那么可能要将依赖网络的按钮和菜单项禁用。此外，也应该给用户展示某种形式的警示信息，告诉他们 GUI 为什么变了。

程序清单 13-1 监控联网状态的变化

```

@protocol ReachabilityWatcher <NSObject>
- (void)reachabilityChanged;
@end

// For each callback, ping the watcher
static void ReachabilityCallback(
    SCNetworkReachabilityRef target,
    SCNetworkConnectionFlags flags, void* info)
{
    @autoreleasepool {
        id watcher = (__bridge id) info;
        if ([watcher respondsToSelector:@selector(reachabilityChanged)])
    
```

```

        [watcher performSelector: @selector(reachabilityChanged)];
    }
}

// Schedule watcher into the run loop
- (BOOL)scheduleReachabilityWatcher:(id <ReachabilityWatcher>)watcher
{
    [self pingReachability];

    SCNetworkReachabilityContext context =
        {0, (__bridge void *)watcher, NULL, NULL, NULL};
    if (SCNetworkReachabilitySetCallback(reachability,
        ReachabilityCallback, &context))
    {
        if (!SCNetworkReachabilityScheduleWithRunLoop(
            reachability, CFRunLoopGetCurrent(),
            kCFRunLoopCommonModes))
        {
            NSLog(@"Error: Could not schedule reachability");
            SCNetworkReachabilitySetCallback(reachability, NULL, NULL);
            return NO;
        }
    }
    else
    {
        NSLog(@"Error: Could not set reachability callback");
        return NO;
    }
    return YES;
}

// Remove the watcher
- (void)unscheduleReachabilityWatcher
{
    SCNetworkReachabilitySetCallback(reachability, NULL, NULL);
    if (SCNetworkReachabilityUnscheduleFromRunLoop(
        reachability, CFRunLoopGetCurrent(),
        kCFRunLoopCommonModes))
    {
        NSLog(@"Success. Unscheduled reachability");
    }
    else
    {
        NSLog(@"Error: Could not unschedule reachability");
    }

    CFRelease(reachability);
    reachability = nil;
}

```

要做好接收多次回调的准备。对于每一种状态变化（也就是指移动网络的建立与断开）或 WiFi 的创建与断开来说，程序通常只会同一时间收到一次回调。然而用户的网络设置（尤其是记住并自动登录已知的 WiFi 网络）可能会影响回调的种类与次数，所以开发者可能需要处理多次回调。



## 13.3 URL 加载系统

苹果公司提供了一套健壮的 API 栈，用于执行网络通信。这个栈自下至上的每一层，包括 BSD Sockets 层、基于 C 语言的 CoreFoundation 层，以及基于 Objective-C 的 Foundation 层，都可以供开发者调用。对于客户端程序来说，很少需要调用比 Foundation 更低的层。

大多数应用程序都可以使用 Foundation 层里提供的 URL 加载系统进行网络通信。只要某服务可以通过 URL 访问，开发者就能够用这个系统来连接、下载及上传数据。此功能并不局限于 HTTP 服务（包括 http 和 https），它也支持文件传输协议（file transfer protocol, ftp）、本地文件 URL 以及数据 URL。

URL 加载系统从一开始就包含在 iOS SDK 之中。NSURLConnection（URL 连接）这个字眼，无论是作为技术名还是作为类名，都令人困惑，不过，很多应用程序都要依靠它来执行各种繁杂的网络操作。这个框架本来是给 Safari 浏览器构建的，后来迁移到了 Foundation。

为了使 NSURLConnection 变得更加灵活、更易配置且更为健壮，苹果公司在 iOS 7 中彻底修订了这个类。新技术叫作 NSURLSession，该类完全能够取代原有的类，而且对其做了大幅度的改进，使开发者可以更为精准地配置它、更好地处理验证事宜、更加方便地设置丰富的委托模型，并且能够轻松地访问由 iOS 7 引入的后台下载功能。虽说原有的 NSURLConnection 系统仍然可以使用，但我们应该改用新的 NSURLSession 技术。

NSURLConnection 所使用的很多类依然会用在 NSURLSession 里面，比方说 NSURL、NSURLRequest 及 NSURLResponse 等，另外，NSURLSession 还增加了一些新的类，用于实现配置，并执行下载数据和上传数据等任务。

### 13.3.1 配置

我们要使用 NSURLSessionConfiguration 对象来配置会话。每次会话（session）都使用各自的配置对象，它们会由原来 NSURLConnection 所提供的全局配置进行加强。开发者可以通过各种属性来设置连接策略、连接数、数据网络的使用、缓存、安全凭据以及 cookie 存储等。我们可以使用 NSURLSessionConfiguration 类的某个工厂方法来创建配置对象，然后再对其做出相应的修改。

有一个很好的属性，它能够在网络超时之外指定整份资源的超时。网络超时（timeoutIntervalForResource）是个“请求级”的配置参数，对于由若干字节所构成的最小传输单位来说，它指明了每一小块数据的超时。而新的资源超时属性（timeoutIntervalForResource），则指明了整份数据的传输超时。

配置好 NSURLSessionConfiguration 对象之后，把它传给 NSURLSession 的构造器。会话中存储了一份 NSURLSessionConfiguration 的拷贝。把配置对象传给会话之后，如果开发者又修改了配置，那么所做的修改是不会生效的。所以，应该在一开始就把它设定好。

### 13.3.2 任务

会话中的每个工作单元都定义成 `NSURLSessionTask` 对象。而“任务”则是从原有的 `NSURLConnection` 类衍生出来的一个概念，每项任务都表示一次网络请求。开发者能够通过任务来获知该网络请求的当前状态。激活任务之后，我们可以用它来取消、暂停或继续某个网络操作。`NSURLConnection` 里的许多连接状态都需要实现相关的委托才能查询，而现在，我们可以通过 `NSURLSessionTask` 的属性直接查询状态。

`NSURLSessionTask` 是个抽象类，它有三个具体类，名为 `NSURLSessionDataTask`、`NSURLSessionDownloadTask` 及 `NSURLSessionUploadTask`，分别提供一般的数据传输、下载及上传功能，如图 13-1 左图所示。这三种任务，既支持基于块的处理程序，又可以通过更为灵活的委托机制来执行并处理网络请求。

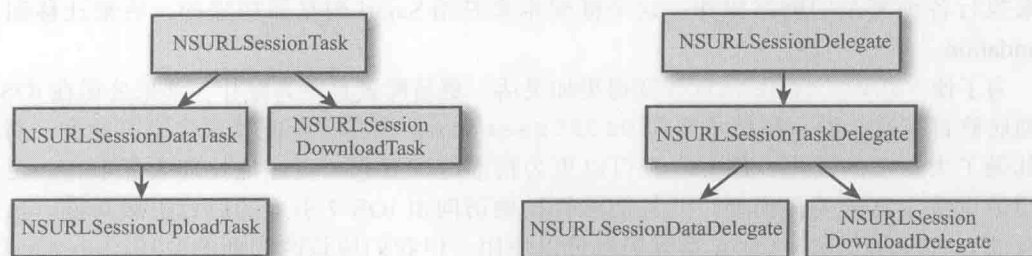


图 13-1 与会话相关的一些任务，提供了下载及上传数据的功能（如左图所示）。而 `NSURLSession` 也提供了与之配套的一些委托协议，用于获取及响应状态变更（如右图所示）

`NSURLSessionTask` 的各子类所提供的功能基本相似，然而有几个重要区别。`NSURLSessionDataTask` 是个通用的基类，能够从内存发送数据，也能接收数据。`NSURLSessionUploadTask` 与之相似（它实际上就是 `NSURLSessionDataTask` 的子类），但是提供了一些与状态有关的委托调用，而且可以使用后台传输功能（参见解决方案 13-5）。`NSURLSessionDownloadTask` 会把收到的数据直接存入文件，并且允许程序从上次取消或失败的地方继续下载。

### 13.3.3 NSURLSession

在新的 URL 加载系统中，会话保存了当前的配置，并且充当创建任务的工厂。这些持续时间比较长的对象，应该在处理多项网络任务的过程中始终保持活动状态。创建会话的时候有两种办法，一种是使用类的构造器来获取默认的会话，另一种是创建自定义的会话。这两种办法的主要区别在于，默认的会话使用共享的配置（该配置与旧的 `NSURLConnection` 所使用的共享栈相同），而自定义的会话则可以使用开发者自己定制的私有配置。

`NSURLSession` 使用同一个委托来处理整个栈中的各种回调，如图 13-1 右图所示。这些委托协议包括 `NSURLSessionDelegate`、`NSURLSessionTaskDelegate`、`NSURLSessionDataTaskDelegate` 及 `NSURLSessionDownloadDelegate`。解决方案 13-3 会进一步讲解这些委托的用法。

用完会话之后，要调用 `invalidateAndCancel` 方法令该会话失效，并且立刻取消尚未完成的任务。也可以调用 `finishTasksAndInvalidate`，该方法会立刻返回，而没有执行完的任务则会继续执行下去。当任务取消或完成之后，系统会把指向委托对象和回调的引用清理并释放掉。会话对象一旦失效，就不能再使用了。

## 13.4 解决方案：简单的下载

许多类都提供了一些便捷的方法，使得开发者可以从因特网中下载数据，并等待数据接收完毕，然后再执行接下来的操作。下面这段代码是同步执行的，而且会阻塞当前的线程：

```
- (UIImage *)imageFromURLString:(NSString *)urlstring
{
    // This is a blocking call
    return [UIImage imageWithData:[NSData
        dataWithContentsOfURL:[NSURL URLWithString:urlstring]]];
}
```

在接收完所有数据之前，上述方法不会返回。如果网络连接卡住了，那么应用程序也会失去响应。若程序阻塞主线程的时间过久，则 iOS 系统的 watchdog 会立刻终止该程序，而不会任其一直卡在那里。

不要在主线程上面使用这种便捷方法，而是应该将其移到后台线程去执行（解决方案 13-1 就是这样做的）。

这些辅助方法写起来很简单，用起来也很容易，然而它们却不够灵活，控制能力也不强，开发者无法追踪下载进度、无法暂停数据传输，也不能设置安全凭据。如果想完全控制下载的过程，并且设定与之有关的配置，那么应该使用 `NSURLSession`。我们可以使用一系列委托回调，也可以使用基于块的处理程序。

解决方案 13-2 采用基于块的办法实现下载，这样简单一些。我们使用会话对象的工厂方法来创建 `NSURLSessionDownloadTask`，并于创建任务时传入处理程序，以便在任务完成后执行<sup>①</sup>：

```
NSURLSessionDownloadTask *task =
    [session downloadTaskWithRequest:request
        completionHandler:^(NSURL *location,
            NSURLResponse *response,
            NSError *error) { // do something }];
```

下载完成之后，系统会向完成处理程序传入下载好的文件位置、应答对象以及错误对象。然后，开发者可以对下载好的文件进行处理，或将其移动到更合适的地方。该文件一开始会存放在临时的文件夹中，处理程序之外的代码不应该访问这个临时文件。

即便开发者提供的 URL 完全无效，某些网络服务提供商也还是会返回一个有效的网页。我们可以通过 `response` 参数来判断是否出现了这样的情况。该参数会

① 这种处理程序叫作完成处理程序，下同。——译者注

指向 `NSURLResponse` 对象。此对象中存放的信息，与 URL 连接所返回的数据有关，包括预期的内容长度 (`expectedContentLength`)，以及系统所建议的文件名 (`suggestedFilename`)。如果 `expectedContentLength` 小于 0，那很可能是网络服务提供商返回的数据与程序所需的数据不符：

```
NSLog(@"Response expects %d bytes",
      response.expectedContentLength);
```

解决方案 13-2 测试了三个预先写好的 URL。其中一个可以下载到比较小的影片 (3MB)，另一个可以下载到比较大的影片 (35MB)，第三个 URL 是伪造的，用于测试程序在 URL 出错时的反应。两个电影文件位于 Internet Archive (<http://archive.org>) 网站，该网站存放了大量公共领域的的数据。

开发者可能想规定只有在使用移动网络以外的方式上网时，才下载比较大的影片。这一策略可以通过 `NSURLSessionConfiguration` 对象来配置：

```
NSURLSessionConfiguration *configuration =
    [NSURLSessionConfiguration defaultSessionConfiguration];
configuration.allowsCellularAccess = NO;
```

创建 `NSURLSession` 的时候，把以下配置对象传进去：

```
NSURLSession *session =
    [NSURLSession sessionWithConfiguration:configuration];
```

解决方案 13-2 把完成处理程序与 `NSURLSessionDownloadTask` 结合起来使用，这种方式无法在下载过程中向用户提供反馈信息。为了解决此问题，解决方案 13-3 使用了更为复杂但功能更加完备的委托机制。

### 解决方案 13-2 简单的下载

```
// Large Movie (35 MB)
#define LARGE_MOVIE @"http://www.archive.org/download/\
    BettyBoopCartoons/Betty_Boop_More_Pep_1936_512kb.mp4"

// Short movie (3 MB)
#define SMALL_MOVIE @"http://www.archive.org/download/\
    Drive-inSaveFreeTv/Drive-in--SaveFreeTv_512kb.mp4"

// Fake address
#define FAKE_MOVIE \
    @"http://www.idontbelievethisisavalidurlforthisexample.com"

// Current URL to test
#define MOVIE_URL [NSURL URLWithString:LARGE_MOVIE]

// Location to copy the downloaded item
#define FILE_LOCATION [NSHomeDirectory() \
    stringByAppendingString:@"/Documents/Movie.mp4"]

@interface TestBedViewController : UIViewController
@end
```

```

@implementation TestBedViewController
{
    BOOL success;
    MPMoviePlayerViewController *player;
}

- (void)playMovie
{
    // Instantiate movie player with location of downloaded file
    player = [[MPMoviePlayerViewController alloc]
        initWithContentURL:[NSURL fileURLWithPath:FILE_LOCATION]];
    [player.moviePlayer setControlStyle: MPMovieControlStyleFullscreen];
    player.moviePlayer.movieSourceType = MPMovieSourceTypeFile;
    player.moviePlayer.allowsAirPlay = YES;
    [player.moviePlayer prepareToPlay];

    // Listen for finish state
    [[NSNotificationCenter defaultCenter] addObserverForName:
        MPMoviePlayerPlaybackDidFinishNotification
        object:player.moviePlayer queue:[NSOperationQueue mainQueue]
        usingBlock:^(NSNotification *notification)
        {
            [[NSNotificationCenter defaultCenter] removeObserver:self
                name:MPMoviePlayerPlaybackDidFinishNotification
                object:nil];
            self.navigationItem.rightBarButtonItem.enabled = YES;
        }];

    [self presentMoviePlayerViewControllerAnimated:player];
}

// Perform an asynchronous download
- (void)downloadMovie:(NSURL *)url
{
    // Turn on network activity indicator
    [UIApplication sharedApplication].networkActivityIndicatorVisible
        = YES;

    NSDate *startDate = [NSDate date];

    // Create a URL request with the URL to the movie
    NSURLRequest *request = [NSURLRequest requestWithURL:url];

    // Create a session configuration
    NSURLSessionConfiguration *configuration =
        [NSURLSessionConfiguration defaultSessionConfiguration];

    // Turn off cellular access for this session
    configuration.allowsCellularAccess = NO;

    // Create a session with the custom configuration

```

```

NSURLSession *session =
    [NSURLSession sessionWithConfiguration:configuration];

// Create a download task with the block-based convenience
// handler to fetch the data
NSURLSessionDownloadTask *task =
    [session downloadTaskWithRequest:request
     completionHandler:^(NSURL *location,
                          NSURLResponse *response, NSError *error) {
        // Turn off the network activity indicator
        [UIApplication sharedApplication]
            .networkActivityIndicatorVisible = NO;

        // Upon an error, reset the UI and abort.
        if (error)
        {
            self.navigationItem.rightBarButtonItem.enabled = YES;
            NSLog(@"Failed download.");
            return;
        }

        // Copy temporary file
        [[NSFileManager defaultManager] copyItemAtURL:location
         toURL:[NSURL fileURLWithPath:FILE_LOCATION]
         error:&error];

        NSLog(@"Elapsed time: %0.2f seconds.",
              [[NSDate date] timeIntervalSinceDate:startDate]);

        // Play the movie
        [self playMovie];
    }];

// Begin the download task
[task resume];
}

// Respond to the user's request to play movie
- (void)action
{
    self.navigationItem.rightBarButtonItem.enabled = NO;

    // Stop any existing movie playback
    [player.moviePlayer stop];
    player = nil;

    // Remove any existing data
    if ([[NSFileManager defaultManager] fileExistsAtPath:FILE_LOCATION])
    {
        NSError *error;
    }
}

```

```

        if (![NSFileManager defaultManager]
            removeItemAtPath:FILE_LOCATION error:&error])
        {
            NSLog(@"Error removing existing data: %@",
                error.localizedDescription);
        }

        // Fetch the data
        [self downloadMovie:MOVIE_URL];
    }

    - (void)loadView
    {
        self.view = [[UIView alloc] init];
        self.view.backgroundColor = [UIColor whiteColor];
        self.navigationItem.rightBarButtonItem =
            UIBarButtonItem(@"Play Movie", @selector(action));
    }

@end

```

## 13.5 解决方案：在下载过程中提供反馈

NSURLSession 提供了基于块的工厂方法，它可以用比较方便的完成处理程序来创建任务。任务完成后，我们可以继续处理数据，或是执行接下来的操作。这些方法用起来比较简单，不过，有时候程序需要在下载或上传的过程中提供更多的交互能力。开发者可以通过 NSURLSessionDelegate 协议及其子协议来访问并配置与任务有关的更多内容。这些子协议不仅支持上级协议中与 NSURLSession 有关的委托方法，而且还提供了一般任务 (NSURLSessionTask)、数据任务 (NSURLSessionDataTask) 或下载任务 (NSURLSessionDownloadTask) 所特有的回调方法。

这些委托提供了与会话及任务的进度或状态相对应的数据。NSURLSession 对象有个共用的委托对象，它既能响应与会话有关的回调，又能响应与任务有关的回调。这种做法初看上去有些奇怪，但是大家要明白，无论给 NSURLSession 对象设置何种委托，它都要同时通过委托方法对会话及对应的任务做出响应。

如果想监控某项下载任务的状态，那么需要实现名为 URLSession:downloadTask:didWriteData:totalBytesWritten:totalBytesExpectedToWrite: 委托方法。totalBytesWritten 表示已传输的字节数，totalBytesExpectedToWrite 表示总共应该传输多少字节，通过这两条信息，我们可以在界面中构建出进度指示器。你也可以在下载任务上面直接查询 countOfBytesReceived 及 countOfBytesExpectedToReceive 属性。请注意，方法签名中的 “written” (已写入的) 及 “received” (已接受的) 可能会令人困惑，其实它们的意思相同，指的都是已传输的字节数。

用已传输的字节数除以需要传输的字节总数，就可以构建出很有用的状态字符串，而且还能够算出进度百分比，以便传给 UIProgressView:

```

int64_t kilobytesReceived =
    downloadTask.countOfBytesReceived / 1024;
int64_t kilobytesExpected =
    downloadTask.countOfBytesExpectedToReceive / 1024;
NSString * statusString = [NSString stringWithFormat:@"%lldk of %lldk",
    kilobytesReceived, kilobytesExpected];
double progress = (double)kilobytesReceived / (double)kilobytesExpected;
progressLabel.text = statusString;
[progressBarView setProgress:progress animated:YES];

```

解决方案 13-3 的表格里面列出了一些待下载的电影，用户可以点击表格视图中的单元格，以下载对应的电影。导航栏显示出了下载进度，而且会实时更新，如图 13-2 所示。下载完成之后，用户可以观看下载的电影。如果想在一份教学范例中实现多个文件同时下载，并显示出它们的进度，需要编写相当复杂的代码。所以，解决方案 13-3 限制用户每次只能下载一个文件。



图 13-2 表格视图中列出了许多下载任务，并显示出每个任务的状态



**提示** 苹果公司在 iOS 7 中引入了 `NSProgress`，用以提供通用的进度管理及进度汇报功能。由于解决方案 13-3 每次只能下载一个文件，而且它所需的状态汇报功能也比较简单，所以我们只需即时更新相关 UI。通过与下载任务相关的委托回调，很容易实现这一点，因此不需借助更加健壮且更加复杂的 `NSProgress`。`NSProgress` 擅长追踪一组任务的总体状态，包括那些不太容易了解其进度的任务，另外，使用了 `NSProgress` 之后，开发者还可以方便地通过 KVO (Key-Value Observing, 键值观测) 机制来更新 UI 元件，以反映进度。

### 解决方案 13-3 在下载过程中提供反馈

```

// Helper class to hold information about a movie and its corresponding download
@interface MovieDownload : NSObject
@property (nonatomic, strong) NSURL *movieURL;
@property (nonatomic, strong) NSURLSessionDownloadTask *downloadTask;
@property (nonatomic, readonly) NSString *localPath;

```



```

@property (nonatomic, readonly) NSString *movieName;
@property (nonatomic, readonly) NSString *statusString;
@property (nonatomic, readonly) double progress;
- (instancetype)initWithURL:(NSURL *)movieURL
    downloadTask:(NSURLSessionDownloadTask *)downloadTask;
@end

@implementation MovieDownload

- (instancetype)initWithURL:(NSURL *)movieURL
    downloadTask:(NSURLSessionDownloadTask *)downloadTask
{
    self = [super init];
    if (self)
    {
        _movieURL = movieURL;
        _downloadTask = downloadTask;
    }
    return self;
}

// A local file path for copying our temporary file
- (NSString *)localPath
{
    NSString *localPath =
        [NSString stringWithFormat:@"%s/Documents/%s",
            NSHomeDirectory(), [self.movieURL lastPathComponent]];
    return localPath;
}

// Display name in UI
- (NSString *)movieName
{
    return [self.movieURL lastPathComponent];
}

// Status string based on progress from download task
- (NSString *)statusString
{
    int64_t kReceived =
        self.downloadTask.countOfBytesReceived / 1024;
    int64_t kExpected =
        self.downloadTask.countOfBytesExpectedToReceive / 1024;
    NSString *statusString =
        [NSString stringWithFormat:@"%lldk of %lldk",
            kReceived, kExpected];
    return statusString;
}

// Progress percentage from download task
- (double)progress

```

```

    {
        double progress = (double)self.downloadTask.countOfBytesReceived /
            (double)self.downloadTask.countOfBytesExpectedToReceive;
        return progress;
    }
@end

// Large Movie (35 MB)
#define LARGE_MOVIE @"http://www.archive.org/download/\
    BettyBoopCartoons/Betty_Boop_More_Pep_1936_512kb.mp4"

// Medium movie (8 MB)
#define MEDIUM_MOVIE @"http://www.archive.org/download/\
    mother_goose_little_miss_muffet/\
    mother_goose_little_miss_muffet_512kb.mp4"

// Short movie (3 MB)
#define SMALL_MOVIE @"http://www.archive.org/download/\
    Drive-inSaveFreeTv/Drive-in--SaveFreeTv_512kb.mp4"

@interface TestBedViewController : UITableViewController
    <NSURLSessionDownloadDelegate>
@end

@implementation TestBedViewController
{
    NSMutableArray *movieDownloads;
    NSURLSession *session;
    UIProgressView *progressBarView;
    MPMoviePlayerViewController *player;
    BOOL downloading;
}

#pragma mark - NSURLSessionDownloadDelegate

// Handle download completion from the task
- (void)URLSession:(NSURLSession *)session
    downloadTask:(NSURLSessionDownloadTask *)downloadTask
    didFinishDownloadingToURL:(NSURL *)location
{
    NSInteger index =
        [self movieDownloadIndexForDownloadTask:downloadTask];
    if (index < 0) return;
    MovieDownload *movieDownload = movieDownloads[index];

    // Copy temporary file
    NSError *error;
    [[NSFileManager defaultManager] copyItemAtURL:location
        toURL:[NSURL fileURLWithPath:[movieDownload localPath]]
        error:&error];
}

```

```

// Handle task completion
- (void)URLSession:(NSURLSession *)session
    task:(NSURLSessionTask *)task
    didCompleteWithError:(NSError *)error
{
    if (error)
        NSLog(@"Task %@ failed: %@", task, error);

    // Update UI
    [progressBarView setProgress:0 animated:NO];
    self.navigationItem.title = @"";
    downloading = NO;

    // This method is called after didFinishDownloadingToURL
    // Task state is up-to-date and reflects completion.
    [self.tableView reloadData];
}

- (void)URLSession:(NSURLSession *)session
    downloadTask:(NSURLSessionDownloadTask *)downloadTask
    didResumeAtOffset:(int64_t)fileOffset
    expectedTotalBytes:(int64_t)expectedTotalBytes
{
    // Required delegate method
}

// Handle progress update from the task
- (void)URLSession:(NSURLSession *)session
    downloadTask:(NSURLSessionDownloadTask *)downloadTask
    didWriteData:(int64_t)bytesWritten
    totalBytesWritten:(int64_t)totalBytesWritten
    totalBytesExpectedToWrite:(int64_t)totalBytesExpectedToWrite
{
    NSInteger index =
        [self movieDownloadIndexForDownloadTask:downloadTask];
    if (index < 0) return;
    MovieDownload *movieDownload = movieDownloads[index];

    // Update UI
    [progressBarView setProgress:movieDownload.progress animated:YES];
    self.navigationItem.title = movieDownload.statusString;
}

#pragma mark - UITableViewDataSource

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    NSInteger sectionCount = 1;
    return sectionCount;
}

```

```

- (NSInteger)tableView:(UITableView *)tableView
  numberOfRowsInSection:(NSInteger)section
{
    NSInteger rowCount = movieDownloads.count;
    return rowCount;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *cellIdentifier = @"CellIdentifier";
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:cellIdentifier];
    if (cell == nil)
    {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleSubtitle
            reuseIdentifier:cellIdentifier];
    }

    // Reset the cell UI
    cell.selectionStyle = UITableViewCellSelectionStyleDefault;
    cell.textLabel.enabled = YES;
    cell.detailTextLabel.enabled = YES;

    // Set our text label to the file name
    cell.textLabel.text = [movieDownloads[indexPath.row] movieName];
    // Acquire the appropriate download task and check its state
    NSURLSessionDownloadTask *downloadTask =
        [movieDownloads[indexPath.row] downloadTask];
    if (downloadTask.state == NSURLSessionTaskStateCompleted)
    {
        cell.detailTextLabel.text = @"Ready to Play";
    }
    else if (downloadTask.state == NSURLSessionTaskStateRunning)
    {
        cell.detailTextLabel.text = @"Downloading...";
    }
    else if (downloadTask.state == NSURLSessionTaskStateSuspended)
    {
        // If download already in progress, disable suspended cells.
        if (downloading)
        {
            cell.selectionStyle = UITableViewCellSelectionStyleNone;
            [cell.textLabel setEnabled:NO];
            [cell.detailTextLabel setEnabled:NO];
        }

        if (downloadTask.countOfBytesReceived > 0)
        {
            cell.detailTextLabel.text = @"Download Paused";
        }
    }
}

```

```

    }
    else
    {
        cell.detailTextLabel.text = @"Not Started";
    }
}

return cell;
}

#pragma mark - UITableViewDelegate

- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Acquire downloadTask and respond to user's selection
    NSURLSessionDownloadTask *downloadTask =
        [movieDownloads[indexPath.row] downloadTask];
    if (downloadTask.state == NSURLSessionTaskStateCompleted)
    {
        // Download is complete. Play movie.
        NSURL *movieURL =
            [NSURL URLWithString:[movieDownloads[indexPath.row]
                localPath]];
        [self playMovieAtURL:movieURL];
    }
    else if (downloadTask.state == NSURLSessionTaskStateSuspended)
    {
        // If suspended and not already downloading, resume transfer.
        if (!downloading)
        {
            [downloadTask resume];
            downloading = YES;
        }
    }
    else if (downloadTask.state == NSURLSessionTaskStateRunning)
    {
        // If already downloading, pause the transfer.
        [downloadTask suspend];
        downloading = NO;
    }
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
    [tableView reloadData];
}

#pragma mark - Movie Download Handling & UI

// Helper method to get index of a movieDownload object from array
- (NSInteger)movieDownloadIndexForDownloadTask:
    (NSURLSessionDownloadTask *)downloadTask
{

```

```

    NSInteger foundIndex = -1;
    NSInteger index = 0;
    for (MovieDownload *movieDownload in movieDownloads)
    {
        if (movieDownload.downloadTask == downloadTask)
        {
            foundIndex = index;
            break;
        }
        index++;
    }
    return foundIndex;
}

// Play movie at the provided URL
- (void)playMovieAtURL:(NSURL *)url
{
    // Instantiate movie player with location of downloaded file
    player =
        [[MPMoviePlayerViewController alloc] initWithContentURL:url];
    player.moviePlayer.controlStyle = MPMovieControlStyleFullscreen;
    player.moviePlayer.movieSourceType = MPMovieSourceTypeFile;
    player.moviePlayer.allowsAirPlay = YES;
    [player.moviePlayer prepareToPlay];

    [self presentMoviePlayerViewControllerAnimated:player];
}

// Convenience method to add movieDownload objects to our array
- (void)addMovieDownload:(NSString *)urlString
{
    NSURL * url = [NSURL URLWithString:urlString];
    NSURLRequest *request = [NSURLRequest requestWithURL:url];
    NSURLSessionDownloadTask *downloadTask =
        [session downloadTaskWithRequest:request];

    MovieDownload *movieDownload = [[MovieDownload alloc]
        initWithURL:url downloadTask:downloadTask];
    [movieDownloads addObject:movieDownload];
}

// Reset the UI, session, and tasks
- (void)reset
{
    for (MovieDownload *movieDownload in movieDownloads)
    {
        // Cancel each task
        NSURLSessionDownloadTask *downloadTask =
            movieDownload.downloadTask;
        [downloadTask cancel];
    }
}

```

```

// Remove any existing data
if ([[NSFileManager defaultManager]
    fileExistsAtPath:movieDownload.localPath])
{
    NSError *error;
    if (![NSFileManager defaultManager]
        removeItemAtPath:movieDownload.localPath
        error:&error])
        NSLog(@"Error removing existing data: %@",
            error.localizedFailureReason);
}

// Cancel all tasks and invalidate session (releases delegate)
[session invalidateAndCancel];
session = nil;

// Create new configuration / session and set delegate
NSURLSessionConfiguration *sessionConfiguration =
    [NSURLSessionConfiguration defaultSessionConfiguration];
session = [NSURLSession
    sessionWithConfiguration:sessionConfiguration
    delegate:self delegateQueue:[NSOperationQueue mainQueue]];

// Create the MovieDownload objects
movieDownloads = [[NSMutableArray alloc] init];
[self addMovieDownload:SMALL_MOVIE];
[self addMovieDownload:MEDIUM_MOVIE];
[self addMovieDownload:LARGE_MOVIE];

// Reset the UI
[progressBarView setProgress:0 animated:NO];
self.navigationItem.title = @" ";
downloading = NO;
[self.tableView reloadData];
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.view.backgroundColor = [UIColor whiteColor];
    self.navigationItem.rightBarButtonItem =
        UIBarButtonItem(@"Reset", @selector(reset));

    // Set up the progress bar in the navigation bar
    progressBarView = [[UIProgressView alloc]
        initWithProgressViewStyle:UIProgressViewStyleBar];
    progressBarView.frame = CGRectMake(0, 0,
        self.navigationController.navigationBar.frame.size.width, 4);
    [self.navigationController.navigationBar

```

```

        addSubview:progressBarView];

        [self reset];
    }

@end

```

## 继续下载

NSURLSessionDownloadTask 有个很棒的功能，就是可以从上次中断的地方继续下载。如果下载失败，那么可以向下载任务获取一块续传数据。当连接出错时，系统会执行相关的回调，而且会通过 [error userInfo] 字典的 NSURLSessionDownloadTaskResumeData 键来提供这份数据。开发者也可以取消下载任务，并通过块处理程序<sup>①</sup>获取最新的续传数据：

```

[downloadTask cancelByProducingResumeData:^(NSData *resumeData) {
    // save resume data
}];

```

如果想继续下载数据，就调用会话对象的 downloadTaskWithResumeData: 或 downloadTaskWithResumeData:completionHandler: 方法，并把原来保存的续传数据块作为参数传进去，这样就可以创建新的下载任务了，该任务会从上次中断的地方继续下载。

## 13.6 解决方案：后台传输

自从有 SDK 以来，许多开发者都希望当程序处在后台时，系统能够继续处理下载任务或上传任务。iOS 7 终于引入了这项强大的功能。后台传输（background transfer）需要用到委托，以便投递事件。开发者可以使用 NSURLSessionUploadTask 及其对应的委托，也可以使用 NSURLSessionDownloadTask 及其对应的委托。后台传输功能仅限于 HTTP 与 HTTPS 协议。无论程序是在前台还是已经切入后台，都可以进行数据传输。

进程内的数据传输与进程外的数据传输，在创建与处理方面，没有多少区别。开发者都需要设定会话对象并设定下载任务，然后开始网络传输。

启动后台传输功能的关键在于会话对象是如何配置的。我们使用 NSURLSessionConfiguration 中名为 backgroundSessionConfiguration: 的类构造器，并传入独特的标识符字符串，以便将来能够重新和这个会话对象相连：

```

NSURLSessionConfiguration *configuration =
    [NSURLSessionConfiguration
        backgroundSessionConfiguration:@"CoreiOSBackgroundID"];
configuration.discretionary = YES;

session = [NSURLSession sessionWithConfiguration:configuration
    delegate:self delegateQueue:nil];

```

① 以“块”的形式编写的处理代码。——译者注




将配置对象的 `discretionary` 属性设为 YES，即可使系统优化后台传输任务，令其在设备接入充电器及连入 WiFi 网络的时候执行。

若想使用后台传输功能，则应在创建应用程序时执行上面的代码，以配置会话对象。如果程序在崩溃之后重新运行，或是在有后台传输的情况下退出并重新运行，那么系统将用后台 ID 来重新创立会话，以便重建上次正在执行的后台会话。系统还会立刻触发该会话中相关任务的委托方法。你也可以调用 `getTasksWithCompletionHandler:`，以便直接获取现存的全部任务。

如果应用程序在前台，那么与传输进度及传输完成有关的委托方法仍然会照常调用，此时的传输就相当于一次普通的进程内下载。程序离开前台时，下载任务依然会继续。等到完成之后，系统会在后台重新启动应用程序。

---

 **提示** 即便应用程序暂停、退出，甚至崩溃，后台传输也依然会继续执行。后台传输发生在独立的守护进程中。程序下次启动的时候，就可以使用传输好的数据了。

---

在 `UIApplicationDelegate` 中，我们需要实现 `application:handleEventsForBackgroundURLSession:completionHandler:` 方法。当程序未处在运行状态时，如果传输操作需要执行验证，或是传输操作已经完成，那么系统将在后台启动应用程序，而且会调用相关的委托方法。

虽说程序运行在后台，但它的 UI 实际上是完整的，只不过隐藏起来了。开发者可以根据刚下载好的数据来更新 UI。更新完 UI 之后，可以抓取快照，以供任务切换器 (task switcher) 使用。若想抓取快照，我们可在委托方法中先把后台任务处理完，并将 UI 更新好，然后执行系统经由 `completionHandler` 参数所传入的块。

解决方案 13-4 使用了解决方案 13-2 所编写的基本功能，同时还支持以后台传输的方式下载数据。开始下载影片之后，即便程序退出或暂停，数据传输也依然不会停止。当传输完成时，我们把电影保存起来，并且适当地更新 UI，以便程序下次来到前台的时候，用户可以看到更新后的样子。虽说后台传输 API 并不允许开发者把程序自动带到前台，但正如解决方案 13-4 所示，我们可以触发一条本机通知 (local notification)。

#### 解决方案 13-4 后台传输

```
// Notify the user of a background transfer completion
- (void)presentNotification
{
    UILocalNotification *localNotification =
        [[UILocalNotification alloc] init];
    localNotification.alertBody = @"Download Complete!";
    localNotification.alertAction = @"Background Transfer";
    localNotification.soundName = UILocalNotificationDefaultSoundName;
    localNotification.applicationIconBadgeNumber = 1;
    [[UIApplication sharedApplication]
        presentLocalNotificationNow:localNotification];
}
```

```

    }

    // Reset the application icon badge on activation
    - (void)applicationDidBecomeActive:(UIApplication *)application
    {
        application.applicationIconBadgeNumber = 0;
    }

    // Handle the background transfer completion event
    - (void)application:(UIApplication *)application
        handleEventsForBackgroundURLSession:(NSString *)identifier
        completionHandler:(void (^)(void))completionHandler
    {
        // Update the UI to make it apparent in the task switcher
        tbvc.view.backgroundColor = [UIColor greenColor];
        tbvc.statusLabel.text = @"BACKGROUND DOWNLOAD COMPLETED!";

        // Present the local notification to the user
        [self presentNotification];

        // Update the task switcher snapshot
        completionHandler();
    }
}

```

### 13.6.1 测试后台传输

为了测试后台传输功能，我们应该在开始下载之后按设备上的 Home 键，以便暂停程序，也可以点击导航栏中的 Exit App 按钮。此按钮将调用 `abort()` 函数，该函数会立刻终止应用程序。（不要在实际程序里使用这个函数，否则会在评审过程中遭到拒绝。）

等到程序切入后台或是终止之后，立刻重新运行该程序，此时程序应该会回到前台，而且其中的下载也会继续进行。我们可以再次将其切入后台，但是不重新运行程序，一直等数据在后台传输完毕。传输完成之后，系统会在后台启动应用程序，并且显示出一条本机通知，此外，在任务切换器中，我们也可以看到更新之后的 UI。

### 13.6.2 Web 服务

目前，如果不借助 Web 服务（网络服务）的话，很难构建出有影响力的 iOS 应用程序。互联网上面的所有数据，无论是公开的还是私有的，几乎都通过 Web 服务来提供。

实用的网络终端会通过包含消息的 HTTP 来运作，而这些消息则是以 JSON 或 XML 来编码的。许多网站都公开发布了与这些服务有关的 API。有些 API 是需要注册方可使用的，不过另外一些则可以随意使用。企业私用的 Web 服务及其文档，只对经过授权的人员开放。

苹果公司提供了一些易于使用的工具，可以下载并处理大部分 Web 服务。URL Loading System 可以获取（get）并投递（post）数据，而且提供了解析器，以便解释并生成需要来回传递的消息。

## 13.7 解决方案：使用 NSJSONSerialization 类

使用基于 JSON 的网络服务时，NSJSONSerialization 类是非常方便的。开发者只需提供有效的 JSON 容器，并在其中放入有效的 JSON 对象即可。JSON 容器就是数组或字典，而其中的 JSON 对象则可以是字符串、数组、字典或 NSNull。isValidJSONObject 方法可以判断某对象是不是有效的 JSON 对象，如果它返回 YES，就说明该对象可以正确地转换成 JSON 格式：

```
// Build a basic JSON object
NSArray *array = @[@"Val1", @"Val2", @"Val3"];
NSDictionary *dict = @{@"Key 1":array,
    @"Key 2":array, @"Key 3":array};

// Convert it to JSON
if ([NSJSONSerialization isValidJSONObject:dict])
{
    NSData *data = [NSJSONSerialization
        dataWithJSONObject:dict options:0 error:nil];
    NSString *result = [[NSString alloc]
        initWithData:data encoding:NSUTF8StringEncoding];
    NSLog(@"Result: %@", result);
}
```

上面代码将会产生下列 JSON。请注意，显示字典内容时，系统不一定会按照每个键的字母顺序来打印：

```
Result: {"Key 2":["Val1","Val2","Val3"],"Key 3":
    ["Val1","Val2","Val3"],"Key 1":["Val1","Val2","Val3"]}
```

把 JSON 转为 Objective-C 对象同样容易。解决方案 13-5 使用 JSONObjectWithData:options:error: 方法，将表示 JSON 对象的 NSData 转换成 Objective-C 对象。这条解决方案会从 <http://openweathermap.org> 网站下载当前的天气预报，并从返回的字典中取得一份数组，数组里含有今后七天的预报。程序会用这些数据来填充标准的表格视图。

### 解决方案 13-5 JSON 数据

```
#define WXFORECAST @"http://api.openweathermap.org/data/2.5/\
    forecast/daily?q=%&units=Imperial&cnt=7&mode=json"
#define LOCATION @"Fairbanks"

// Return a cell for the index path
- (UITableViewCell *)tableView:(UITableView *)aTableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Top level dictionary for the forecast data
    NSDictionary *top = [items objectAtIndex:indexPath.row];

    // The date of this forecast under top
    NSString *unixtime = top[@"dt"];
```

```

// The weather dictionary that includes the sky description
NSDictionary * weather = top[@"weather"][0];

// The sky description string under weather
NSString *wxDescription = weather[@"description"];

// Convert the unixtime to something we can use
NSDate *wxDate = [NSDate dateWithTimeIntervalSince1970:
    [unixtime doubleValue]];

UITableViewCell *cell = [self.tableView
    dequeueReusableCellWithIdentifier:@"cell"];
if (cell == nil)
{
    cell = [[UITableViewCell alloc]
        initWithStyle:UITableViewCellStyleSubtitle
        reuseIdentifier:@"cell"];
}
cell.textLabel.text = wxDescription;
cell.detailTextLabel.text =
    [dateFormatter stringFromDate:wxDate];
return cell;
}

#pragma mark - Web Service Download

- (void)loadWebService
{
    self.title = LOCATION;

    // Start the refresh control
    [self.refreshControl beginRefreshing];
    // Create the URL string based on location
    NSString *urlString =
        [NSString stringWithFormat:WXFORECAST, LOCATION];

    // Set up the session
    NSURLSessionConfiguration * configuration =
        [NSURLSessionConfiguration defaultSessionConfiguration];
    NSURLSession *session =
        [NSURLSession sessionWithConfiguration:configuration];
    NSURLRequest *request =
        [NSURLRequest requestWithURL:[NSURL
            URLWithString:urlString]];

    // Create a data task to transfer the web service endpoint contents
    NSURLSessionDataTask *dataTask =
        [session dataTaskWithRequest:request
            completionHandler:^(NSData *data,
                NSURLResponse *response, NSError *error) {

```

```

// Stop the refresh control
[self.refreshControl endRefreshing];
if (error)
{
    self.title = error.localizedDescription;
    return;
}

// Parse the JSON from the data object
NSDictionary *json = [NSJSONSerialization
    JSONObjectWithData:data options:0 error:nil];

// Store off the top level array of forecasts
items = json[@"list"];

[self.tableView reloadData];
}];

// Start the data task
[dataTask resume];
}

```

## 13.8 解决方案：将 XML 转换为树状结构

虽说许多 Web 服务都开始使用较为简单的 JSON 格式了，但是 XML 作为一种文档编码格式，依然很流行。iOS 的 `NSXMLParser` 类可以扫描 XML，并且会在开始处理某个新元素以及处理完该元素时创建回调（也就是说，它按照 SAX 解析器的通常逻辑来运作）。如果要从某些简单的数据源里面下载一两份相关的信息，那么这个类用起来很合适。但如果要执行的操作依赖于错误检查、状态信息以及反复握手等机制，那它也许就不太能胜任了。

解决方案 13-6 与解决方案 13-5 类似，也要从 <http://openweathermap.org> 网站获取同样的天气预报数据，只不过这次我们使用 XML 格式。它需要把请求的模式由 json 改为 xml，并使用 XML 解析器来填充其表格视图：

```

#define WXFORECAST \
    @"http://api.openweathermap.org/data/2.5/\
    forecast/daily?q=%@&units=Imperial&cnt=7&mode=xml"
#define LOCATION @"Fairbanks"

- (void)loadWebService
{
    self.title = LOCATION;

    // Start the refresh control
    [self.refreshControl beginRefreshing];

    // Create the URL string based on location
    NSString *urlString =

```

```

        [NSString stringWithFormat:WXFORECAST, LOCATION];

// Set up the session
NSURLSessionConfiguration * configuration =
    [NSURLSessionConfiguration defaultSessionConfiguration];
NSURLSession *session =
    [NSURLSession sessionWithConfiguration:configuration];
NSURLRequest *request =
    [NSURLRequest requestWithURL:[NSURL URLWithString:urlString]];

// Create a data task to transfer the web service endpoint contents
NSURLSessionDataTask *dataTask =
    [session dataTaskWithRequest:request
    completionHandler:^(NSData *data, NSURLResponse *response,
        NSError *error) {

// Stop the refresh control
[self.refreshControl endRefreshing];
if (error)
{
    self.title = error.localizedDescription;
    return;
}

// Create the XML parser
XMLParser *parser = [[XMLParser alloc] init];

// Parse the XML from the data object
root = [parser parseXMLFromData:data];

// Store off the top level parent of forecasts
forecastsRoot = [root nodesForKey:@"forecast"][0];

[self.tableView reloadData];
}];

// Start the data task
[dataTask resume];
}

// Return a cell for the index path
- (UITableViewCell *)tableView:(UITableView *)aTableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    TreeNode *forecastRoot = forecastsRoot.children[indexPath.row];
    NSString *day = [forecastRoot attributes][@"day"];
    TreeNode *cloudsNode = [forecastRoot nodeForKey:@"clouds"];
    NSString *wxDescription = [cloudsNode attributes][@"value"];

    UITableViewCell *cell =
        [self.tableView dequeueReusableCellWithIdentifier:@"cell"];

```

```

    if (cell == nil)
    {
        cell = [[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleSubtitle
                reuseIdentifier:@"cell"];
    }
    cell.textLabel.text = wxDescription;
    cell.detailTextLabel.text = day;
    return cell;
}

```

### 13.8.1 树

树这种数据结构很适合用来表示 XML 数据。开发者可以用 XML 数据创建出一棵树，只要我们能够把数据正确地放在内存中，就可以通过树中的搜索路径找到自己想要的数据。我们可以获取所有元素，也可以搜寻某个表示操作成功的值，等等。树可以把基于文本的 XML 转化成一种多维度的结构。

为了把 NSXMLParser 的解析结果转化成树状结构，我们需要使用一种基于 NSXMLParser 的辅助类来返回标准的树状数据。而这个辅助类又需要用到下面这种简单的树节点：

```

@interface TreeNode : NSObject
@property (nonatomic, weak)      TreeNode      *parent;
@property (nonatomic, strong)    NSMutableArray *children;
@property (nonatomic, strong)    NSString      *key;
@property (nonatomic, strong)    NSDictionary *attributes;
@property (nonatomic, strong)    NSString      *leafValue;
@end

```

这种节点里面有两个链接，分别用来表示本节点的上级节点和下级节点，以便使开发者能够在树里进行双向遍历。在这两个链接中，只有从上级节点指向下级节点的引用是强引用，这使得系统可以回收树中的内存，而无须开发者手工执行清理。

比方说，我们从 <http://weathermap.org> 网站的 Web 服务中获取了下面这段 XML 代码（笔者对其做了简化）：

```

<time day="2013-12-01">
    <clouds value="sky is clear"/>
</time>

```

这段有效的 XML 将会解析成两个 TreeNode 对象，这两个对象的 key 属性分别是 time 和 clouds。time 节点的 children 数组将会包含 clouds 节点。而 clouds 节点的 parent 属性又会指向 time 节点。time 节点的 attributes 属性是一份字典，其中含有名为 day 的键，该键所对应的值是 2013-12-01。与之相似，clouds 节点里也会有 attributes 字典，在该字典中，value 键所对应的值是 sky is clear。

### 13.8.2 构建解析树

解决方案 13-6 设计了 XMLParser 类。该类会在 NSXMLParser 类遍历 XML 源数据的

过程中构建出解析树。在实现 NSXMLParserDelegate 协议中三个标准的解析方法（系统分别会在找到新元素、解析完现有元素以及找到字符的时候回调这三个方法）时，该类会读取 XML 流，并按照深度优先方式递归地遍历这棵树。

### 解决方案 13-6 XMLParser 辅助类

```
@implementation XMLParser
// Parser returns the tree root. Go down
// one node to the real results
- (TreeNode *)parse:(NSXMLParser *)parser
{
    stack = [NSMutableArray array];
    TreeNode *root = [TreeNode treeNode];
    [stack addObject:root];

    [parser setDelegate:self];
    [parser parse];

    // Pop down to real root
    TreeNode *realRoot = [[root children] lastObject];

    // Remove any connections
    root.children = nil;
    root.leafValue = nil;
    root.key = nil;
    realRoot.parent = nil;

    // Return the true root
    return realRoot;
}

- (TreeNode *)parseXMLFromURL:(NSURL *)url
{
    TreeNode *results = nil;
    @autoreleasepool {
        NSXMLParser *parser =
            [[NSXMLParser alloc] initWithContentsOfURL:url];
        results = [self parse:parser];
    }
    return results;
}

- (TreeNode *)parseXMLFromData:(NSData *)data
{
    TreeNode *results = nil;
    @autoreleasepool {
        NSXMLParser *parser =
            [[NSXMLParser alloc] initWithData:data];
        results = [self parse:parser];
    }
    return results;
}
```



```

    }

    // Descend to a new element
    - (void)parser:(NSXMLParser *)parser
      didStartElement:(NSString *)elementName
      namespaceURI:(NSString *)namespaceURI
      qualifiedName:(NSString *)qName
      attributes:(NSDictionary *)attributeDict
    {
        if (qName) elementName = qName;

        TreeNode *leaf = [TreeNode treeNode];
        leaf.parent = [stack lastObject];
        [(NSMutableArray *)[[stack lastObject] children] addObject:leaf];
        leaf.attributes = attributeDict;
        leaf.key = [NSString stringWithString:elementName];
        leaf.leafValue = nil;
        leaf.children = [NSMutableArray array];

        [stack addObject:leaf];
    }

    // Pop after finishing element
    - (void)parser:(NSXMLParser *)parser
      didEndElement:(NSString *)elementName
      namespaceURI:(NSString *)namespaceURI
      qualifiedName:(NSString *)qName
    {
        [stack removeLastObject];
    }

    // Reached a leaf
    - (void)parser:(NSXMLParser *)parser
      foundCharacters:(NSString *)string
    {
        if (![stack lastObject] leafValue)
        {
            [[stack lastObject]
             setLeafValue:[NSString stringWithString:string]];
            return;
        }
        [[stack lastObject] setLeafValue:
         [NSString stringWithFormat:@"%s%@",
          [[stack lastObject] leafValue], string]];
    }
@end

```

该类会在发现新元素的时候(也就是系统回调 `parser:didStartElement:qualifiedName:attributes:` 的时候)添加新节点,并且会在找到文本的时候(也就是系统回调 `parser:foundCharacters:` 的时候)添加叶值(leaf value)。在 XML 中,同一深

度上可能会有平级的节点，因此，该类的代码要用栈来记录当前节点与根节点之间的路径。这样的话，我们就可以在 `parser:didEndElement:` 方法中把具有相同上级元素的平级节点从栈中弹出，使得这些节点能够处在正确的级别上面。

扫描完 XML 之后，`parseXMLFromData:` 方法返回根节点。

## 13.9 小结

本章介绍了与网络有关的基础知识。读者学到了如何检测网络连接、如何下载数据，以及如何在 Objective-C 对象与 JSON 之间相互转换。学习下一章之前，请先回顾以下几个知识点：

- 在苹果公司所提供的各种网络技术里面，有一些是通过底层的 C 语言例程来实现的。如果能够找到与之对应的 Objective-C 封装器，就应该用封装过的技术来编程，以简化编码工作。这么做只有一个缺点，就是我们无法从最基础的层面上紧密控制程序中的网络操作，但需要这样做的场合非常少。有许多优秀的网络编程资源，在网上搜一下就能找到。
- iOS 7 提供了基于 `NSURLSession` 的 URL 加载系统，这个新的抽象系统的功能非常强大，它可以从互联网下载数据，也可以向互联网上传数据。开发者应该尽量使用这套新的 API 来编程。
- iOS 7 的后台传输功能很有用，它可以令程序及时响应用户的操作，并向用户展示出最新的界面。后台传输技术可以与 iOS 7 新引入的无声推技术（`silent push notification`，可以发送通知以触发下载操作，但不会出现警示界面）及后台获取技术（`background fetch`，一种新的后台模式，为频繁的后台下载而设计）结合起来，使得用户每次启动应用程序时，总能看到最新的内容。
- 设备联网时，最重要的事情就是要考虑到网络错误。开发者需要设计出适当的应对措施。我们要检测网络是否连通、下载操作是否中断，以及收到的数据是否已损坏。这些情况都基于一项基本的事实，那就是：程序所需的数据并不总是能够获取到，而且获取到的数据未必就是正确的。
- 编写联网代码的时候，要有“线程”这个概念。如果毫无顾忌地在主线程上执行网络操作，那么系统很可能会直接把你的程序关掉。块和队列是两项非常有用的新技术，它们能够使网络应用程序更加及时地响应用户。

## 针对特定设备的开发

每台 iOS 设备都有其特殊属性，同时，这些设备之间也有共同属性。有些属性随时都会改变，有些属性则一直保持不变。上述属性包括设备当前的物理方向、型号、电池状态，以及能够操作的硬件。本章讲解设备的硬件规格以及可供使用的感应器，其中的各条解决方案将会给出很多与设备有关的信息。读者会学到如何在程序运行的时候判断硬件是否符合需求，以及怎样在应用程序的 Info.plist 文件里指定这些硬件需求。你会看到怎样通过 Core Motion 来激发感应器的反馈，以及怎样订阅通知，以便在感应器状态有变时触发回调。我们还会学习如何使用屏幕镜像及第二屏幕输出等功能，以及如何获取可供记录的设备。本章涵盖 iPhone、iPad 与 iPod touch 的硬件、文件系统及感应器，旨在告诉大家如何以编程的方式利用这些特性。

### 14.1 访问基本的设备信息

UIDevice 类提供了与设备有关的一些重要属性，包括 iPhone、iPad 或 iPod touch 的型号、设备名称、OS 名称以及 OS 版本。通过这种一站式解决方案，我们可以获知系统的一些详细信息。每项属性都要通过实例方法来获取，我们需要用 [UIDevice currentDevice] 获得 UIDevice 单例，然后在单例上面调用相应的实例方法。

可以从 UIDevice 中获取的设备信息有：

- **systemName**——该属性返回当前操作系统的名称。对于目前的 iOS 设备来说，只有一种系统运行在 iOS 平台上，那就是 iPhone OS。虽然苹果公司已经将 iPhone OS 改名为 iOS，但这个属性并未随之更新。
- **systemVersion**——该属性表示设备上面安装的固件版本，比方说 4.3、5.1.1、6.0、7.0.2 等。

- **model**——该属性是个字符串，用来描述设备的型号，也就是 iPhone、iPad 或 iPod touch。如果 iOS 系统将来能够运行在新型的设备上面，那么还会有其他字符串用来描述那些型号。localizedModel 提供了该属性的本地化版本。
- **userInterfaceIdiom**——该属性表示当前设备的界面样式，它要么是 iPhone 式（iPhone 与 iPod touch），要么是 iPad 式。如果苹果公司将来发布了具有新式界面的设备，那么可能还会出现其他属性值。
- **name**——该属性表示用户在 iTunes 中给 iPhone 所起的名字，比方说“Joe’s iPhone”或“Binky”。这个名称也用来创建设备的本地主机名。

下面举例说明以上几个属性的用法：

```
UIDevice *device = [UIDevice currentDevice];  
NSLog(@"System name: %@", device.systemName);  
NSLog(@"Model: %@", device.model);  
NSLog(@"Name: %@", device.name);
```

在目前的 iOS 系统中，我们可以用一条简单的布尔测试来判断设备的界面样式。下面这段代码可以判断界面是不是 iPad 式的：

```
#define IS_IPAD (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad)
```

请注意，上面代码使用了 UIKit 中一个很方便的宏——UI\_USER\_INTERFACE\_IDIOM()。它会判断 UIDevice 能否响应特定的选择子，如果可以，就返回 [[UIDevice currentDevice] userInterfaceIdiom]，若不能，则返回 UIUserInterfaceIdiomPhone。这项测试如果失败，就可以假定程序正运行在 iPhone 或 iPod touch 上面。苹果公司将来若是推出一系列新设备，我们就要修改上述代码，以进行更详细的测试。

## 14.2 添加设备能力限制

把程序提交到 App Store 的时候，可以在 Info.plist 属性列表里面指定程序的需求。这些需求将会告诉 iTunes 以及移动版的 App Store 设备必须具备哪些能力，才能运行本程序。

每台 iOS 设备的硬件能力各不相同。有些带有摄像头和 GPS 功能，有些则没有。还有一些带有陀螺仪（gyro）、自动对焦以及其他强大的特性。开发者可以指明设备必须具备何种能力，方能运行本程序。

如果 Info.plist 文件里面有 UIRequiredDeviceCapabilities 键，那么 iTunes 及移动版的 App Store 就会对应用程序的安装施加限制，令其只能安装在具备这些能力的设备上面。开发者可以通过字符串数组或字典来提供这份列表。

如果采用数组来指明设备应该具有的能力，那么数组中的每个元素就表示设备必须要有的一项功能。若是用字典，则可以明确规定设备必须具有或不能具有的特性。字典里的每个键都表示一种特性。键所对应的值如果是真，就表示设备必须具备此特性，如果是假，则表示设备绝不能具备此特性。

表 14-1 详细列出了描述各种设备特征的键。只有当程序必须运行在具备某特征的设备上面，或是绝不能运行在具备某特征的设备上面时，我们才应该对其做出限定。程序如果能够在不具备某特征的设备上面设法运行，那我们就不应该增加这条限制。表 14-1 以肯定句式描述了每项特征。如果我们要规定设备绝不能具备某特征，那就把对应的意思反过来解读，比方说，可以规定程序绝不能运行在具备自动对焦摄像头或陀螺仪的设备上面，也可以规定程序绝不能运行在支持 Game Center 的设备上面。

表 14-1 对应用程序施加限制时可以提及的设备能力

键	含 义
telephony	应用程序需要使用 Phone 程序，或是需要使用 tel:// 格式的 URL
wifi	应用程序需要访问本地的 802.11 网络。如果开发者要求 iOS 在运行程序的时候必须保持 WiFi 连接，需要把 UIRequiresPersistentWiFi 作为顶级的键添加到属性列表里
sms	应用程序需要使用 Messages 程序，或是需要使用 sms:// 格式的 URL
still-camera	应用程序需要使用能够拍摄静态照片的摄像头，而且可以通过图像选取器界面用这种摄像头来拍摄照片
auto-focus-camera	应用程序需要使用带有自动对焦功能的摄像头，以便进行微距摄影（macro photography），或是需要用它来拍摄特别清晰的图像，以便检测其中的某些数据
front-facing-camera	应用程序需要使用设备的前置摄像头
camera-flash	应用程序需要使用摄像头的闪光灯
video-camera	应用程序需要使用能够录影的摄像头
accelerometer	除了使用简单的 UIViewController 方向事件之外，应用程序还要使用与加速计有关的反馈功能
gyroscope	应用程序需要使用设备中的陀螺仪
location-services	应用程序需要使用 Core Location 框架中的功能
gps	应用程序需要使用 Core Location 框架，而且要使用更为精确的 GPS 定位功能
magnetometer	应用程序要使用 Core Location 框架，而且要使用与设备朝向有关的事件，也就是说，程序想知道设备的移动方向（磁力仪是设备内置的指南针）
gamekit	应用程序需要访问 Game Center（适用于 iOS 4.1 及后续系统）
microphone	应用程序需要使用内置的麦克风，或是使用（苹果公司所认可的）带有麦克风的配件
opengles-1	应用程序需要使用 OpenGL ES 1.1
opengles-2	应用程序需要使用 OpenGL ES 2.0
armv6	应用程序只针对 armv6 指令集而编译（适用于 iOS 3.1 及后续系统）
armv7	应用程序只针对 armv7 指令集而编译（适用于 iOS 3.1 及后续系统）
peer-peer	应用程序需要通过蓝牙使用 GameKit 的点对点连接（适用于 iOS 3.1 及后续系统）
bluetooth-le	应用程序必须运行在具有低功耗蓝牙装置的设备上面（适用于 iOS 5.0 及后续系统）

比方说，某应用程序可以在具备摄像头的设备上面提供拍照功能。如果此程序也能在不带摄像头的早期 iPod touch 上面运作，那么就不要再施加 still-camera 限制，而是应该在程序里面判断设备有没有摄像头，如果有，就展示拍照选项。若施加了 still-camera 限制，会失去很多使用早期 iPod touch（第一代至第三代）及 iPad（第一代）的潜在用户。

### 14.2.1 提供描述信息以征求用户同意

为了保护隐私，应用程序在使用日历数据、摄像头、联系人、照片、位置等功能时，必须获得终端用户同意。而为了使终端用户能够同意，开发者应该解释一下程序是如何使用相关数据的，并且应该描述出这样做的原因。在 Info.plist 文件中，我们可以为下面几个顶级的键指定相关的字符串值：

- NSBluetoothPeripheralUsageDescription
- NSCalendarsUsageDescription
- NSCameraUsageDescription
- NSContactsUsageDescription
- NSLocationUsageDescription
- NSMicrophoneUsageDescription
- NSMotionUsageDescription
- NSPhotoLibraryUsageDescription
- NSRemindersUsageDescription

当 iOS 向用户征询特定资源的访问权时，它会把相关字符串显示在标准的对话框里。

### 14.2.2 Info.plist 文件中其他常用的键

下面列出其他常用的键及其含义，这些键可以用在属性列表之中：

- **UIFileSharingEnabled** (布尔值，默认关闭) ——该属性使得用户可以从 iTunes 中访问应用程序 Documents 文件夹下的内容。该文件夹出现在应用程序沙盒的最顶层。
- **UIAppFonts** (数组，其中每个字符串都是字体文件的名称，文件名里也包含扩展名) ——该属性用来指定 app bundle 中所包含的自定义 TTF 字体。如果添加了这种字体，那么可以使用标准的 UIFont 调用来访问它们。
- **UIApplicationExitsOnSuspend** (布尔值，默认关闭) ——当用户按下 Home 键时，系统可以把应用程序直接终止，而不是将其切入后台。如果启用了该属性，那么 iOS 就会终止程序，并将其从内存中移除。
- **UIRequiresPersistentWifi** (布尔值，默认关闭) ——该属性可以告诉 iOS 系统：在程序处于活动状态时，应该保持 WiFi 连接。
- **UIStatusBarHidden** (布尔值，默认关闭) ——如果启用该属性，当程序启动时状态栏会隐藏起来。
- **UIStatusBarStyle** (字符串，默认是 **UIStatusBarStyleDefault**) ——设定状态栏在程序启动时的样式。

## 14.3 解决方案：检查设备距离与电池状态

通过 UIDevice 类所提供的 API，开发者可以追踪设备的某些特征，比方说电池的状态

以及距离感应器 (proximity sensor)<sup>①</sup> 的状态等。解决方案 14-1 演示了如何监控并查询这两项特征。它们都以通知的形式来表达状态的变化, 我们可以在程序中订阅这种通知, 以便在特征发生变化时得到通报。

### 14.3.1 启用与禁用距离感应器

目前来说, 距离感应器是 iPhone 特有的功能。iPod touch 和 iPad 并没有提供距离感应器。除非我们有强烈的理由要判断 iPhone 与用户是否贴得很紧 (或离得很远), 否则距离感应器没有多大用处。

启用距离感应器之后, 它的主要任务就是判断设备正前方是否有比较大的物体。如果有, 就关掉屏幕, 并发送一条通用的通知。物体若是远离设备, 屏幕则会重新亮起。用户在打电话时, 耳朵可能会碰到屏幕, 而有了距离感应器之后, 我们就能防止用户无意间碰到某个按钮或拨出某个号码。有些手机保护壳设计得不够好, 可能会使 iPhone 的距离感应器无法正常运作。

Siri 也会使用距离感应器。如果把电话放在耳朵旁边, 它就会记录下用户所说的话, 并给出答复。Siri 的语音界面并不依赖于可视的 GUI。

解决方案 14-1 演示了如何在 iPhone 上使用距离感应器。我们用 `UIDevice` 类来切换距离感应器, 并订阅 `UIDeviceProximityStateDidChangeNotification` 通知, 以捕获状态变化。两种状态分别是 on (开) 和 off (关)。如果 `UIDevice` 的 `proximityState` 属性返回 YES, 就说明距离感应器已经激活了。

### 14.3.2 监控电池状态

我们可以用编程的方式来追踪电池及充电状态。通过相关的 API, 开发者可以知道电池的电量, 以及设备是否接入了电源。电量是个浮点值, 1.0 表示完全充满, 0.0 表示完全耗尽。在执行非常耗电的操作之前, 我们可以通过这个值了解设备大概还剩下多少电。

比方说, 在执行一大批数学计算之前, 我们想提醒用户将设备接入电源。此时可通过 `UIDevice` 来查询电量, 查到的电量值是以 5% 为间隔递进的:

```
NSLog(@"Battery level: %0.2f%%",
      [UIDevice currentDevice].batteryLevel * 100);
```

充电状态有四种取值。设备可能处于正在充电的状态 (charging, 也就是说, 正在同电源相连)、充满电的状态 (full)、未接入充电器的状态 (unplugged), 以及不同于上述三种的未知状态 (unknown)。通过 `UIDevice` 的 `batteryState` 属性, 可以查到充电状态:

```
NSArray *stateArray = @[
    @"Battery state is unknown",
    @"Battery is not plugged into a charging source",
    @"Battery is charging",
    @"Battery state is full"];

NSLog(@"Battery state: %@",
      stateArray[[UIDevice currentDevice].batteryState]);
```

① 也称“接近度感应器”。——译者注

这些值并不表示持久的状态，应该把它们看成是对设备实际状态的一种即时反映。这些东西不是标志（flag），所以不能通过 OR（或）运算组合成对电池状态的通用描述信息。它们只是反映了最近的状态变化。

我们可以在电池状态发生改变时响应通知，以便监控这些状态。如此一来，就可以捕获随时发生的事件了。比方说，电池何时彻底充满电，用户何时将设备接入电源重新充电，以及用户何时把设备与电源断开等。

把 `batteryMonitoringEnabled` 属性设为 YES，即可开始监控。在监控期间，`UIDevice` 类会在电池状态或电量发生变化时产生通知。解决方案 14-1 订阅了这两种通知。请注意，我们也可以不等接到通知就直接去检查这些值。虽然苹果公司并未承诺将以何种频率来投递关于电量变化的通知，但是大家只要测试一下这条解决方案就会发现，通知的投递频率还是相当规律的。

#### 解决方案 14-1 监控距离感应器及电池

---

```
// View the current battery level and state
- (void)peekAtBatteryState
{
    NSArray *stateArray = @[@"Battery state is unknown",
                             @"Battery is not plugged into a charging source",
                             @"Battery is charging",
                             @"Battery state is full"];

    NSString *status = [NSString stringWithFormat:
        @"Battery state: %@, Battery level: %0.2f%%",
        stateArray[[UIDevice currentDevice].batteryState],
        [UIDevice currentDevice].batteryLevel * 100];

    NSLog(@"%@", status);
}

// Show whether proximity is being monitored
- (void)updateTitle
{
    self.title = [NSString stringWithFormat:@"Proximity %@",
        [UIDevice currentDevice].proximityMonitoringEnabled ? @"On" : @"Off"];
}

// Toggle proximity monitoring off and on
- (void)toggle:(id)sender
{
    // Determine the current proximity monitoring and toggle it
    BOOL isEnabled = [UIDevice currentDevice].proximityMonitoringEnabled;
    [UIDevice currentDevice].proximityMonitoringEnabled = !isEnabled;
    [self updateTitle];
}

- (void)loadView
{

```



```

self.view = [[UIView alloc] init];

// Enable toggling and initialize title
self.navigationItem.rightBarButtonItem =
    UIBarButtonItem(@"Toggle", @selector(toggle:));
[self updateTitle];

// Add proximity state checker
[[NSNotificationCenter defaultCenter]
    addObserverForName:UIDeviceProximityStateDidChangeNotification
    object:nil queue:[NSOperationQueue mainQueue]
    usingBlock:^(NSNotification *notification) {
        // Sensor has triggered either on or off
        NSLog(@"The proximity sensor %@",
            [UIDevice currentDevice].proximityState ?
            @"will now blank the screen" : @"will now restore the screen");
    }];

// Enable battery monitoring
[[UIDevice currentDevice] setBatteryMonitoringEnabled:YES];

// Add observers for battery state and level changes
[[NSNotificationCenter defaultCenter]
    addObserverForName:UIDeviceBatteryStateDidChangeNotification
    object:nil queue:[NSOperationQueue mainQueue]
    usingBlock:^(NSNotification *notification) {
        // State has changed
        NSLog(@"Battery State Change");
        [self peekAtBatteryState];
    }];

[[NSNotificationCenter defaultCenter]
    addObserverForName:UIDeviceBatteryLevelDidChangeNotification
    object:nil queue:[NSOperationQueue mainQueue]
    usingBlock:^(NSNotification *notification) {
        // Level has changed
        NSLog(@"Battery Level Change");
        [self peekAtBatteryState];
    }];
}

```

### 获取解决方案代码

访问 <https://github.com/erica/iOS-7-Cookbook> 网页，并打开“C14 Device”文件夹，即可找到与本章中的解决方案相对应的完整范例项目。

## 14.3.3 判断设备是否具有 Retina 显示屏

近年来，苹果公司为很多设备都装配了 Retina 显示屏，只有一些成本较低的设备没有升

级。根据苹果公司的说法，Retina 显示屏的像素密度很高，以致人眼无法分辨出单个的像素。应用程序可以利用清晰度较好的显示屏来展示分辨率较高的图像。

UIScreen 类提供了一种简单的办法，可以判断当前设备有没有内置 Retina 显示屏。这种办法就是检测 UIScreen 的 scale 属性。在把逻辑坐标空间（该空间的坐标以点为单位，每个点大约是 1/160 英寸）转换到设备坐标空间（该空间的坐标以像素为单位）时，该属性用来表示换算倍数。对于标准的显示屏来说，这个倍数是 1.0，也就是说，1 个点对应 1 个像素。而对于 Retina 显示屏来说，这个倍数则是 2.0，也就是说，1 个点会用 4 个像素来显示：

```
- (BOOL) hasRetinaDisplay
{
    return ([UIScreen mainScreen].scale == 2.0f);
}
```

UIScreen 类还提供了两个有用的属性，用来表示显示区域的尺寸。bounds 属性代表屏幕的外接矩形（bounding rectangle），它所用的计量单位是点。该属性表示屏幕的总大小，这与屏幕上是否有状态栏、导航栏或标签栏等元件无关。applicationFrame 属性也是以点来计量的，它表示应用程序初始的视窗尺寸。

## 14.4 解决方案：获取设备的其他信息

开发者可以通过 sysctl() 及 sysctlbyname() 来获取系统信息。这些标准的 UNIX 函数能够向操作系统查询硬件及 OS 的详情。读者可以看看 Macintosh 上面的 /usr/include/sys/sysctl.h 头文件，以便了解能够查询的范围。该头文件详细列出各种常量，这些常量可以用作这两个函数的参数。

我们可以通过常量查出一些关键信息，例如系统的 CPU 个数、可供使用的内存数量等。解决方案 14-2 演示了这一功能。它针对 UIDevice 类编写 category（扩展），以收集系统信息，开发者可以调用 category 中的一系列方法来查询这些信息。

解决方案 14-2 收集更多的设备信息

```
@implementation UIDevice (Hardware)
+ (NSString *)getSysInfoByName:(char *)typeSpecifier
{
    // Recover sysctl information by name
    size_t size;
    sysctlbyname(typeSpecifier, NULL, &size, NULL, 0);

    char *answer = malloc(size);
    sysctlbyname(typeSpecifier, answer, &size, NULL, 0);

    NSString *results = [NSString stringWithCString:answer
                                                encoding: NSUTF8StringEncoding];
    free(answer);
}
```

```

        return results;
    }

- (NSString *)platform
{
    return [UIDevice getSysInfoByName:"hw.machine"];
}

- (NSUInteger)getSysInfo:(uint)typeSpecifier
{
    size_t size = sizeof(int);
    int results;
    int mib[2] = {CTL_HW, typeSpecifier};
    sysctl(mib, 2, &results, &size, NULL, 0);
    return (NSUInteger) results;
}

- (NSUInteger)busFrequency
{
    return [UIDevice getSysInfo:HW_BUS_FREQ];
}

- (NSUInteger)totalMemory
{
    return [UIDevice getSysInfo:HW_PHYSMEM];
}

- (NSUInteger)userMemory
{
    return [UIDevice getSysInfo:HW_USERMEM];
}

- (NSUInteger)maxSocketBufferSize
{
    return [UIDevice getSysInfo:KIPC_MAXSOCKBUF];
}

@end

```

读者可能会问：既然 `UIDevice` 类已经提供了返回设备型号的 `model` 属性，为什么还要在这个 `category` 里编写 `platform` 方法呢？这是因为，我们可以更加具体地判断出同一型号下的各类设备。

如果用 `UIDevice` 类的 `model` 来查询，那么 iPhone 3GS 和 iPhone 4S 这两种设备所返回的型号都是 `iPhone`。但如果改用解决方案中的 `platform` 方法来查询，那么 iPhone 3GS 会返回 `iPhone2,1`，iPhone 4S 会返回 `iPhone4,1`，iPhone 5 会返回 `iPhone5,1`。这就使我们能够以编程的方式把 3GS 同第一代 iPhone (`iPhone1,1`) 或 iPhone 3G 区分开 (`iPhone1,2`)。

每种设备都内置了一些独有的特征。开发者获知具体的 iPhone 类型之后，就可以判断出该设备是否具备辅助功能、GPS 以及磁力仪等特征。

## 14.5 Core Motion 基础知识

Core Motion 框架可以集中访问由 iOS 硬件所生成的运动数据 (motion data)。它监控着三个重要的感应器：一个是陀螺仪，用以度量设备的旋转角度；一个是磁力计，用以反映指南针的方位读数；还有一个是加速计，用以探测设备在三条坐标轴上的加速度。此外，该框架还提供了名为设备动作的入口，可以把上述三种感应器集成到一套监控系统之中。

Core Motion 根据这些感应器的原始值来创建可供开发者使用的读数，这些读数主要以力向量 (force vector) 的形式来体现。可以测量的条目包括下面这些属性：

- **attitude**——设备的姿态是指设备相对于某个参照系的方向。attitude 是以 roll、pitch、yaw<sup>①</sup>这三种角来衡量的，角的单位是弧度。
- **rotationRate**——旋转率是设备绕着每条坐标轴旋转时的速率。它包括设备围绕 *x* 轴、*y* 轴及 *z* 轴旋转时的角速度，该速度以每秒所转过的弧度来计量。
- **gravity**——重力是当前设备受到普通的重力场吸引而产生的加速度向量。设备的 gravity 要分别沿着 *x*、*y*、*z* 这三条坐标轴来描述，每条坐标轴所用的计量单位都是 *g*。*g* 就是地球上标准的重力加速度 (也就是 32 英尺 1 秒<sup>2</sup>，或 9.8 米 1 秒<sup>2</sup>)。
- **userAcceleration**——用户加速度是设备因为用户的动作而产生的加速度向量。与 gravity 一样，userAcceleration 也要分别沿着 *x*、*y*、*z* 这三条坐标轴来计量，其计量单位也是 *g*。把用户向量与设备向量叠加在一起，就是设备总体的加速度向量。
- **magneticField**——磁场是表示设备附近总体磁场值的向量。它是按照 *x* 轴、*y* 轴及 *z* 轴上面的微特斯拉 (microtesla) 来度量的。此外，系统还会给出校准精确度 (calibration accuracy)，使得应用程序可以了解该读数的准确程度。

### 14.5.1 判断设备是否支持某种感应器

本章前面说过，我们可以通过应用程序的 Info.plist 文件来限定设备必须具备或绝不能具备某种感应器。另外，开发者也可以在程序里面查询 Core Motion 的 CMMotionManager 对象，以了解设备是否支持某种感应器：

```
if (motionManager.gyroAvailable)
    [motionManager startGyroUpdates];

if (motionManager.magnetometerAvailable)
    [motionManager startMagnetometerUpdates];

if (motionManager.accelerometerAvailable)
    [motionManager startAccelerometerUpdates];

if (motionManager.deviceMotionAvailable)
    [motionManager startDeviceMotionUpdates];
```

① 它们分别表示设备绕着 *x* 轴、*y* 轴及 *z* 轴旋转的角度。——译者注

### 14.5.2 获取感应器数据

Core Motion 提供了两套访问感应器数据的机制。其中一套机制能够定期地、被动地访问运动数据，也就是先激活感应器（比方说，调用 `startAccelerometerUpdates`），然后在 `CMMotionManager` 对象上面通过对应的属性（例如 `accelerometerData`）来获取数据。

如果轮询（polling）方式不能满足需要的话，可以使用基于块的更新机制，也就是提供一个块，每当感应器有变化时，系统就会执行这个块（比方说，调用 `startAccelerometerUpdatesToQueue:withHandler:`）。在使用这种基于处理程序的方法时，一定要给感应器设置更新间隔（例如 `accelerometerUpdateInterval`）。这个间隔是有最大值和最小值限制的，如果实际频率对应用程序来说非常重要，需要在系统传给块的数据对象里面检查时间戳。

## 14.6 解决方案：通过加速度来判断“上”方向

iPhone 和 iPad 提供了三个感应器，可以分别度量设备在三条坐标轴上的加速度。这三条轴是相互垂直的，X 轴表示左右方向、Y 轴表示上下方向、Z 轴表示前后方向。三个感应器的值分别表示设备在三条坐标轴上由于重力和用户的动作而受到的影响。我们可以把 iPhone 放在头上转圈（向心力），或是将其从高楼掷下（自由落体），这样做虽然能够得到相当强烈的力反馈，但是却会把 iPhone 摔成碎片，从而无法查到感应器的数据。

若想监控加速计的变化，就要创建 Core Motion 的管理器对象，然后设置更新间隔，启动管理器，并传入作为处理程序的块，以便在感应器发生变化时得到调用：

```
motionManager = [[CMMotionManager alloc] init];
motionManager.accelerometerUpdateInterval = 0.005;
if (motionManager.isAccelerometerAvailable)
{
    [motionManager startAccelerometerUpdatesToQueue:
        [NSOperationQueue mainQueue]
        withHandler:^(CMAccelerometerData *accelerometerData,
            NSError *error) {
            // handle the accelerometer update
        }];
}
```

使用 Core Motion 框架的时候，总是应该先检查相关感应器是否可用。开始监控之后，处理程序块会收到 `CMAccelerometerData` 对象，开发者可以追踪并响应它们。`CMAccelerometerData` 对象包含 `CMAcceleration` 结构体，该结构体由 x 轴、y 轴及 z 轴上的浮点值构成，每个浮点值都处在 -1.0 到 1.0 的范围内。

解决方案 14-3 使用这些值来判断真实的“上”方向。它会根据 X 轴和 Y 轴上面的加速度来计算反正切值，以求出箭头图案当前的“上”方向与真实的“上”方向之间的夹角。收到新的加速度消息之后，这条解决方案会旋转 `UIImageView` 实例，使得 `UIImageView` 之中的箭头图案能够指向真实的“上方”，如图 14-1 所示。由于程序可以实时响应用户操作，所以无论用户如何调整设备的方向，箭头总是会对准真实的“上方”。



图 14-1 只需通过反正切函数对  $x$  轴和  $y$  轴的力向量稍作数学运算，就可以找到真实的“上”方向。在本例中，无论用户如何调整设备方向，程序的箭头总是指向真正的上方

### 解决方案 14-3 处理加速度事件

```

- (void)loadView
{
    self.view = [[UIView alloc] init];
    self.view.backgroundColor = [UIColor whiteColor];
    arrow = [[UIImageView alloc]
        initWithImage:[UIImage imageNamed:@"arrow"]];
    [self.view addSubview:arrow];
    PREPCONSTRAINTS(arrow);
    CENTER_VIEW(self.view, arrow);

    motionManager = [[CMMotionManager alloc] init];
    motionManager.accelerometerUpdateInterval = 0.005;
    if (motionManager.isAccelerometerAvailable)
    {
        [motionManager
            startAccelerometerUpdatesToQueue:
                [NSOperationQueue mainQueue]
            withHandler:
                ^(CMAccelerometerData *accelerometerData,
                    NSError *error) {
                    CMAcceleration acceleration =
                        accelerometerData.acceleration;

                    // Determine up from the x and y acceleration components
                    float xx = -acceleration.x;
                    float yy = acceleration.y;
                    float angle = atan2(yy, xx);
                    [arrow setTransform:CGAffineTransformMakeRotation(angle)];
                }
        ];
    }
}

```

## 14.7 使用基本的方向值

UIDevice 类内置的 orientation 属性可以提供设备的物理方向。对于 iOS 设备来

说，该属性有下面七种取值：

- `UIDeviceOrientationUnknown`——当前方向未知。
- `UIDeviceOrientationPortrait`——home 按钮在下方。
- `UIDeviceOrientationPortraitUpsideDown`——home 按钮在上方。
- `UIDeviceOrientationLandscapeLeft`——home 按钮在右侧。
- `UIDeviceOrientationLandscapeRight`——home 按钮在左侧。
- `UIDeviceOrientationFaceUp`——屏幕面朝上。
- `UIDeviceOrientationFaceDown`——屏幕面朝下。

设备在运行应用程序的时候，其方向可能会历经上述七种取值中的几种，也可能会历经全部七种取值。虽说方向值是与加速计相配合的，但它们并不对应于某种特定的角度值。

iOS 提供了 `UIDeviceOrientationIsPortrait()` 及 `UIDeviceOrientationIsLandscape()` 这两个内置的宏，可以根据枚举值来判断设备是否处于竖屏或横屏状态。我们可以参照下面这段代码对 `UIDevice` 做扩展，以便通过两个属性来给出设备的横竖屏状态：

```
@property (nonatomic, readonly) BOOL isLandscape;
@property (nonatomic, readonly) BOOL isPortrait;

- (BOOL) isLandscape
{
    return UIDeviceOrientationIsLandscape(self.orientation);
}

- (BOOL) isPortrait
{
    return UIDeviceOrientationIsPortrait(self.orientation);
}
```

开发者需要用 `beginGeneratingDeviceOrientationNotifications` 来启动方向变更通知，否则 `orientation` 属性就会返回 0。把设备的方向变更通知激活之后，就可以用代码来订阅通知了。我们可以添加监听器（observer），以捕获后续的 `UIDeviceOrientationDidChangeNotification` 事件。调用 `endGeneratingDeviceOrientationNotification` 方法则可以取消监控。

### 14.7.1 根据加速计来判断方向

程序刚启动的时候，`UIDevice` 类并不会报告正确的方向。只有当设备移动到新的位置，或是 `UIViewController` 方法生效之后，它才会更新设备的方向。

必须等用户把设备从标准方向移开，并将其重新恢复到标准方向之后，程序才能意识到设备正处在竖屏状态。模拟器和 iPhone 上面都会出现这种情况，大家很容易就能测试出来。（据说有人在苹果公司的事务管理系统里面报告过这个问题，但是苹果公司将该问题关闭了，因为它认为这个特性本来就应该设计成这个样子。）

为了解决此问题，我们可以考虑通过 Core Motion 框架来获取加速计的信息，以判断角度。下面这段代码能够计算出设备的角度：

```
float xx = acceleration.x;
float yy = -acceleration.y;
device_angle = M_PI / 2.0f - atan2(yy, xx);

if (device_angle > M_PI)
    device_angle -= 2 * M_PI;
```

计算好上面的数值之后，我们把基于加速计的角度换算成设备的方向。下面是换算所用的代码：

```
// Limited to the four portrait/landscape options
- (UIDeviceOrientation)acceleratorBasedOrientation
{
    CGFloat baseAngle = self.orientationAngle;
    if ((baseAngle > -M_PI_4) && (baseAngle < M_PI_4))
        return UIDeviceOrientationPortrait;
    if ((baseAngle < -M_PI_4) && (baseAngle > -3 * M_PI_4))
        return UIDeviceOrientationLandscapeLeft;
    if ((baseAngle > M_PI_4) && (baseAngle < 3 * M_PI_4))
        return UIDeviceOrientationLandscapeRight;
    return UIDeviceOrientationPortraitUpsideDown;
}
```

请注意，这段范例代码只考虑  $x$ - $y$  平面，因为与用户界面有关的大部分测试都是依照这个平面来判定的。这段代码完全忽视了  $z$  轴，也就是说，对于设备面朝上和设备面朝下的情况来说，这段代码会产生随机的结果。若是真需要处理这两种情况，需要修改范例代码，以进行更为详细的判断。

`UIViewController` 类的 `interfaceOrientation` 实例方法用于报告视图控制器的界面方向。虽说这并不能取代加速计的值，但是很多界面布局操作其实都依赖于底层的视图方向，而不是设备加速计的读数。

请注意，子视图控制器的布局方向有可能和设备的方向不同，在 iPad 上面尤其如此。比方说，在横屏的分栏视图控制器里，可能嵌有采用竖屏布局的子控制器。开发者需要根据底层的界面方向来考虑自己所写的方向检测代码是否合用。界面方向也许比设备方向更为可靠，尤其是在应用程序刚启动的时候。我们需要根据情况做出相应处理。

## 14.7.2 计算相对角度

由于用户可以调整屏幕方向，所以在设备屏幕面朝上的时候，我们必须分四种情况来给出界面方向与设备方向之间的夹角。当 `UIViewController` 自动旋转其视图的时候，开发者需要根据旋转后的界面方向，通过数学运算求出这个夹角。

下面的方法是为 `UIDevice` 的 `category`（扩展）而编写的，它会依照设备方向算出相对角度。这段范例代码根据 GUI 目前的显示方式，计算设备在垂直方向上的偏移角：

```
- (float)orientationAngleRelativeToOrientation:
    (UIDeviceOrientation)someOrientation
{
    float dOrientation = 0.0f;
```



```

switch (someOrientation)
{
    case UIDeviceOrientationPortraitUpsideDown:
        {dOrientation = M_PI; break;}
    case UIDeviceOrientationLandscapeLeft:
        {dOrientation = -(M_PI/2.0f); break;}
    case UIDeviceOrientationLandscapeRight:
        {dOrientation = (M_PI/2.0f); break;}
    default: break;
}

float adjustedAngle =
    fmod(self.orientationAngle - dOrientation, 2.0f * M_PI);
if (adjustedAngle > (M_PI + 0.01f))
    adjustedAngle = (adjustedAngle - 2.0f * M_PI);
return adjustedAngle;
}

```

上述方法用浮点数模除（floating-point modulo）来计算设备的实际角度与界面方向角之间的偏移，从而返回非常重要的垂直方向偏移角。



从 iOS 6 开始，我们就应该在根视图控制器和 Info.plist 文件中使用 supported-InterfaceOrientations 来规定界面方向，而不应该再使用 shouldAutorotateToInterfaceOrientation: 了。iOS 会采用这两处 supported-InterfaceOrientations 值的交集来决定应用程序中可以出现的界面方向。

## 14.8 解决方案：使用加速计来移动屏幕上的物体

只需编写少许代码，即可利用 iPhone 的加速计来“移动”屏幕上的物体，以便在用户倾斜手机的时候做出实时响应。解决方案 14-4 构建了一只带有动画效果的蝴蝶，它可以随着用户的操作在屏幕上飞来飞去。

实现该功能的关键在于添加“物理计时器”（physics timer）。我们不像解决方案 14-3 那样直接响应加速度的变化，而是令加速度处理程序去衡量当前的力度。计时器例程会修改蝴蝶的 frame，以便把力的效果运用在它身上。下面列出实现该功能时的几个要点：

- 只要力的方向不变，蝴蝶就会持续加速。它的速度会一直增大，并且会根据加速力（acceleration force）在  $x$  轴或  $y$  轴的程度来相应地缩放。
- 计时器会调用 tick 例程，该例程会把速度向量叠加到蝴蝶的原点上。
- 蝴蝶的移动范围是有限制的。碰到边缘之后，就不能继续朝着那个方向移动了。这样可以令蝴蝶始终出现在屏幕上。tick 方法会检测蝴蝶是否碰到了某边界。比方说，蝴蝶碰到了垂直边界之后，依然可以在上下方向上移动。

- 蝴蝶会调整自己的方向，使它看起来总是朝“下”。我们在 tick 方法中对其进行简单的旋转变换。在对 frame 或中心点做偏移之后，如果还要执行变换，就要小心了。此时总是应该先重置变换矩阵，然后运用偏移量，最后再执行旋转变换。如果不这么做，就会导致 frame 出现不符合预期的缩放或斜切效果。

## 动作管理器的配置与清理

解决方案 14-4 的 `establishMotionManager` 及 `shutDownMotionManager` 方法可以按照需要来启用或关闭程序中的动作管理器 (motion manager)。应用程序激活或暂停的时候，系统会从应用程序委托里面调用这两个方法：

```
- (void)applicationWillResignActive:(UIApplication *)application
{
    [tbvc shutDownMotionManager];
}

- (void)applicationDidBecomeActive:(UIApplication *)application
{
    [tbvc establishMotionManager];
}
```

这两个方法提供了一种优雅的方式，使得开发者可以根据应用程序的当前状态来关闭或开启动作服务。



**提示** 计时器本身并不使用基于块的 API。如果你喜欢把计时器同块结合起来，而不喜欢用回调来处理事件，需要在 GitHub 上面搜索一套可以实现此功能的代码。

### 解决方案 14-4 根据加速计所反馈的数据来滑动屏幕上的物体

@implementation TestBedViewController

```
- (void)tick
{
    butterfly.transform = CGAffineTransformIdentity;

    // Move the butterfly according to the current velocity vector
    CGRect rect = CGRectOffset(butterfly.frame, xVelocity, 0.0f);
    if (CGRectContainsRect(self.view.bounds, rect))
        butterfly.frame = rect;

    rect = CGRectOffset(butterfly.frame, 0.0f, yVelocity);
    if (CGRectContainsRect(self.view.bounds, rect))
        butterfly.frame = rect;

    butterfly.transform =
        CGAffineTransformMakeRotation(mostRecentAngle - M_PI_2);
}
```



```

        userInfo:nil repeats:YES];
    }

- (void)initButterfly
{
    CGSize size;

    // Load the animation cells
    NSMutableArray *butterflies = [NSMutableArray array];
    for (int i = 1; i <= 17; i++)
    {
        NSString *fileName =
            [NSString stringWithFormat:@"bf_%d.png", i];
        UIImage *image = [UIImage imageNamed:fileName];
        size = image.size;
        [butterflies addObject:image];
    }

    // Begin the animation
    butterfly = [[UIImageView alloc]
        initWithFrame:CGRectMake(.size=size)];
    [butterfly setAnimationImages:butterflies];
    butterfly.animationDuration = 0.75f;
    [butterfly startAnimating];

    // Set the butterfly's initial speed and acceleration
    xAccel = 2.0f;
    yAccel = 2.0f;
    xVelocity = 0.0f;
    yVelocity = 0.0f;

    // Add the butterfly
    [self.view addSubview:butterfly];
}

- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];

    // Get our butterfly centered
    butterfly.center = RECTCENTER(self.view.bounds);
}

- (void)loadView
{
    self.view = [[UIView alloc] init];
    self.view.backgroundColor = [UIColor whiteColor];
    [self initButterfly];
}

```

## 14.9 解决方案：基于加速计的滚动视图

很多读者都希望本书能在这一版中添加有关“倾斜滚动功能”（tilt scroller）的解决方案。此功能会根据设备内置的加速计来移动 UIScrollView 中的内容。用户调整设备的时候，其中的内容会相应地向下滚动。我们并不是把视图直接放在屏幕上，而是要把 UIScrollView 中的内容滚动到新的位置。

创建这种界面的难点在于：如何规定坐标轴的基准角度，使得设备处在该角度的时候，其中的内容不会滚动。很多人都觉得应该把设备平躺于桌面时的角度设为基准角度，此时设备的  $z$  轴是垂直向上的。其实这是个很糟糕的设计。如果使用这个角度作为基准角度，那么用户在浏览内容的时候，必须把设备向下倾斜或向上倾斜才行。但是设备倾斜之后，用户就无法完全看到屏幕里的内容了，当用户坐下来使用手机，或是手机位于用户头顶的时候，这个问题尤为突出。

于是，解决方案 14-5 规定：当  $z$  轴指向大约 45 度角的方向时，设备中的内容不会滚动，这个角度正是用户手持 iPhone 或 iPad 时的角度。它处于“设备面朝上”和“设备面朝用户”这两个位置之间。解决方案 14-5 根据这一角度来执行相应的运算。用户从这个 45 度角的位置向下或向上倾斜手机时，都能最大限度地看到屏幕中的内容。

解决方案 14-5 为 UIScrollView 添加倾斜滚动功能

```
- (void)tick
{
    xOff += xVelocity;
    xOff = MIN(xOff, 1.0f);
    xOff = MAX(xOff, 0.0f);

    yOff += yVelocity;
    yOff = MIN(yOff, 1.0f);
    yOff = MAX(yOff, 0.0f);

    // update the content offset based on the current velocities
    UIScrollView *sv = (UIScrollView *) self.view;
    CGFloat xSize = sv.contentSize.width - sv.frame.size.width;
    CGFloat ySize = sv.contentSize.height - sv.frame.size.height;
    sv.contentOffset = CGPointMake(xOff * xSize, yOff * ySize);
}

- (void) viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    NSString *map = @"http://maps.weather.com/images/\\
        maps/current/curwx_720x486.jpg";
    NSOperationQueue *queue = [[NSOperationQueue alloc] init];
    [queue addOperationWithBlock:
        ^{
            // Load the weather data
            NSURL *weatherURL = [NSURL URLWithString:map];
```

```

NSData *imageData = [NSData dataWithContentsOfURL:weatherURL];

// Update the image on the main thread using the main queue
[[NSOperationQueue mainQueue] addOperationWithBlock:^(
    UIImage *weatherImage = [UIImage imageWithData:imageData];
    UIImageView *imageView =
        [[UIImageView alloc] initWithImage:weatherImage];
    CGSize initSize = weatherImage.size;
    CGSize destSize = weatherImage.size;

    // Ensure that the content size is significantly bigger
    // than the screen can show at once
    while ((destSize.width < (self.view.frame.size.width * 4)) ||
        (destSize.height < (self.view.frame.size.height * 4)))
    {
        destSize.width += initSize.width;
        destSize.height += initSize.height;
    }

    imageView.frame = (CGRect){.size = destSize};
    UIScrollView *sv = (UIScrollView *) self.view;
    sv.contentSize = destSize;
    [sv addSubview:imageView];

    // only allowing accelerometer-based scrolling
    scrollView.userInteractionEnabled = NO;

    // Activate the accelerometer
    [motionManager startAccelerometerUpdatesToQueue:
        [NSOperationQueue mainQueue] withHandler:
        ^(CMAccelerometerData *accelerometerData,
            NSError *error) {

            // extract the acceleration components
            CMAcceleration acceleration =
                accelerometerData.acceleration;
            float xx = -acceleration.x;
            // between face-up and face-forward
            float yy = (acceleration.z + 0.5f) * 2.0f;

            // Has the direction changed?
            float accelDirX = SIGN(xVelocity) * -1.0f;
            float newDirX = SIGN(xx);
            float accelDirY = SIGN(yVelocity) * -1.0f;
            float newDirY = SIGN(yy);

            // Accelerate. To increase viscosity lower the additive value
            if (accelDirX == newDirX)
                xAccel = (abs(xAccel) + 0.005f) * SIGN(xAccel);

```

```

if (accelDirY == newDirY)
{
    yAccel = (abs(yAccel) + 0.005f) * SIGN(yAccel);
    // Apply acceleration changes to the current velocity
    xVelocity = -xAccel * xx;
    yVelocity = -yAccel * yy;
}

// Start the physics timer
[NSTimer scheduledTimerWithTimeInterval:0.03f
 target:self selector:@selector(tick)
 userInfo:nil repeats:YES];
}
}
}

```

与解决方案 14-4 相比，本例还有个变化，那就是把加速度常量设置得更慢了。由于屏幕上的内容滚动得比较慢，所以用户更加容易控制滚动速度。

## 14.10 解决方案：获取并使用设备的姿态

假设桌子上有一台 iPad。iPad 里面显示了一幅图像，用户可以俯身查看这幅图。如果 iPad 在不离开桌子的前提下原地旋转，那么当 iPad 转动时，该图片始终会处于固定的位置，继续与它周边的空间相对齐，而不会受到 iPad 的影响。无论如何旋转 iPad，这幅图都不会随设备“移动”，因为它会根据设备的物理运动来平衡自身的位置。这就是解决方案 14-6 的工作原理，它利用设备中的陀螺仪来实现此功能。这条解决方案必须使用陀螺仪才能运作。

图像会根据用户握持设备的方式而做出调整。除了把 iPad 放在桌面上旋转之外，我们也可以把设备拿起来，令其朝着空间中的某个方位。如果将设备面朝下并置于头顶，那么用户就能抬头看到图像的“反面”了。在操控设备的过程中，图像会做出相应的调整，以便在 iPad 里面创建一个虚拟的静态空间。

解决方案 14-6 演示了如何通过几次简单的几何变换来实现此功能。它会建立动作管理器、订阅与设备动作有关的通知，然后根据动作管理器所返回的 roll、pitch、yaw 角度来变换图像。

### 解决方案 14-6 通过设备的动作信息把图像固定在空间中的某个位置上

```

- (void)shutDownMotionManager
{
    NSLog(@"Shutting down motion manager");
    [motionManager stopDeviceMotionUpdates];
    motionManager = nil;
}

- (void)establishMotionManager
{
    if (motionManager)

```

```

[self shutDownMotionManager];

NSLog(@"Establishing motion manager");

// Establish the motion manager
motionManager = [[CMMotionManager alloc] init];
if (motionManager.deviceMotionAvailable)
{
    [motionManager
     startDeviceMotionUpdatesToQueue:
     [NSOperationQueue currentQueue]
     withHandler: ^(CMDeviceMotion *motion, NSError *error) {
        CATransform3D transform;
        transform = CATransform3DMakeRotation(
            motion.attitude.pitch, 1, 0, 0);
        transform = CATransform3DRotate(transform,
            motion.attitude.roll, 0, 1, 0);
        transform = CATransform3DRotate(transform,
            motion.attitude.yaw, 0, 0, 1);
        imageView.layer.transform = transform;
    }];
}

```

---

## 14.11 用 Motion Event 来检测晃动

iPhone 如果侦测到晃动等动作事件，就会把该事件传给目前的第一响应者，也就是传给响应者链中的主对象。所谓响应者（responder），就是一种能够处理事件的对象。所有的视图与视窗都是响应者，而应用程序对象也是一种响应者。

响应者链是一套对象体系，其中的所有对象都可以对事件做出响应。位于响应者链开头的对象如果能够处理某事件，那么该事件就不再继续向下传递了。若是不能处理，那么该事件将会移动到下一个响应者。

任何对象都可以通过 `becomeFirstResponder` 来宣称自己是第一响应者，并获得这种身份。在下面这段代码中，当 `UIViewController` 把视图显示到屏幕上的时候，它会变成第一响应者，而当视图消失时，它又会放弃第一响应者的身份：

```

- (BOOL)canBecomeFirstResponder {
    return YES;
}

// Become first responder whenever the view appears
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    [self becomeFirstResponder];
}

// Resign first responder whenever the view disappears
- (void)viewWillDisappear:(BOOL)animated {

```



```

    [super viewWillDisappear:animated];
    [self resignFirstResponder];
}

```

第一响应者会收到所有的触摸事件及动作事件。与动作有关的回调，分别对应于 `UIView` 里面的各种触摸回调。这些回调方法是：

- **`motionBegan:withEvent:`**——这个回调方法会在动作事件刚开始的时候执行。笔者编写本书时，系统只能识别一种动作事件，即晃动（shake）。以后也许还能识别其他类型的动作，所以我们应该用代码来判断动作类型。
- **`motionEnded:withEvent:`**——动作事件结束时，第一响应者会收到这个回调。
- **`motionCancelled:withEvent:`**——与触摸事件一样，动作事件也会由于打进来的电话或其他系统事件而取消。苹果公司建议开发者在编写实际的代码时，应该把与动作事件有关的三个回调方法全都实现好（同理，也应该把与触摸事件有关的四个回调方法全都实现好）。

下面这段范例代码实现了与动作事件有关的两个回调方法：

```

- (void)motionBegan:(UIEventSubtype)motion
  withEvent:(UIEvent *)event {

    // Play a sound whenever a shake motion starts
    if (motion != UIEventSubtypeMotionShake) return;
    [self playSound:startSound];
}

- (void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event
{
    // Play a sound whenever a shake motion ends
    if (motion != UIEventSubtypeMotionShake) return;
    [self playSound:endSound];
}

```

如果要在设备上测试这段代码，请注意几件事。首先，从用户的角度来看，事件的开始和结束几乎在同一时间发生。所以，给这两种情况都播放一遍声音，显得有点重复。第二，设备更容易检测出左右方向的晃动。iPhone 检测左右方向的晃动，要比检测前后方向及上下方向的晃动更为灵敏。第三，苹果公司在实现动作事件时采用了轻微的间隔机制。也就是说，在前一个事件处理完之后的一两秒内，不会再产生新的动作事件。晃动切换和晃动撤销事件都有这种延迟。

## 14.12 使用外接屏幕

有很多种使用外接屏幕的办法。以最新的 iPad 为例。第 2 代、第 3 代、第 4 代 iPad 都提供了内置的屏幕镜像功能。通过 VGA 或 HDMI 线，我们可以把内容同时显示在外接屏幕与内置屏幕上。某些设备还可以用 AirPlay 把屏幕内容以无线方式分享到苹果公司 TV 上面，AirPlay 是苹果公司的一项专利，它能够无线地传播视频。这些镜像分享功能非常方便，

而且我们并不局限于把一个 iOS 屏幕里的内容简单地复制到另一个屏幕中。

UIDevice 类可以检测外接屏幕，并向其中分别写入内容。开发者可以把接入的任何显示器都当成新的窗口，并在其中创建与设备主显示器上的视图不同的内容。凡是通过线缆接入的屏幕，都能够这样做，在 iPad 2 及后续机型、iPhone 4S 及后续机型、iPod touch 第五代及后续机型上面，还可以通过 AirPlay 技术以无线方式把内容传播到 Apple TV 2 及其后续机型的屏幕中。有个第三方案程序叫作 Reflector，它可以通过 AirPlay 把内容显示到 Mac 或 Windows 电脑上。

屏幕的几何特征<sup>①</sup>是非常重要的。iOS 设备目前的屏幕分辨率有如下几种：老式 iPhone 是 320 像素 × 480 像素、带有 Retina 屏幕的 iPhone 是 640 像素 × 960 像素、iPad 是 1024 像素 × 768 像素、带有 Retina 屏幕的 iPad 是 2048 像素 × 1536 像素。而一般的复合输出 / 分量输出所采用的分辨率是 720 像素 × 480 像素（480i 及 480p），VGA 输出则是 1024 像素 × 768 像素及 1280 像素 × 720 像素（720p），另外，也有质量更高的 HDMI 输出。除了这些问题之外，还有“过扫描”（overscan）<sup>②</sup>以及“目标设备的限制”等因素，这使得 Video Out（视频输出）的分辨率变得复杂起来。

所幸苹果公司提供了一些方便而实用的适配技术，可以化解这一难题。我们不需要在外接屏幕与设备内置的屏幕之间创建一一对应关系，而是可以直接根据外接设备的属性来构建内容。我们只需创建视窗，向其中填充内容，然后把视窗显示出来即可。

如果想开发 Video Out 程序，就不要假定用户一定会使用 AirPlay。很多用户依然会通过老式的线缆接入方式来连接显示器或投影仪。所以，每种类型的线缆至少要准备一条（复合输出线、分量输出线、VGA 线、HDMI 线），而且还要分别准备一台支持 AirPlay 的 iPhone 及 iPad，只有这样，才能完整地测试每一种输出方式。第三方线缆（也就是没有打上 Made for iPhone/iPad 标贴的线缆）是无法支持 Video Out 的，一定要购买苹果公司认可的线缆。

## 14.12.1 检测屏幕

UIScreen 类可以查出设备所连接的屏幕数量：

```
#define SCREEN_CONNECTED ([UIScreen screens].count > 1)
```

如果 [UIScreen screens] 的数量大于 1，就表明设备接入了外部的屏幕。screens 数组中的首个元素，总是表示设备主屏幕。

每个屏幕都有自己的 bounds 及 scale，前者表示以点为单位的物理尺寸，后者表示点与像素之间的换算关系。通过下面两个标准的通知，开发者可以获知设备何时连上了外接屏幕，以及何时与外接屏幕断开：

```
// Register for connect/disconnect notifications
[[NSNotificationCenter defaultCenter]
 addObserver:self selector:@selector(screenDidConnect:)
```

① 这里是按照原文 geometry 对译，实际上可以根据语境理解为“分辨率”等含义。——译者注


② 这个术语的大意是：显示器会把将要显示的图像略微放大，从而导致用户能够看到的图像范围略小于图像的真实尺寸。——译者注

```
name:UIScreenDidConnectNotification object:nil];
[[NSNotificationCenter defaultCenter]
 addObserver:self selector:@selector(screenDidDisconnect:)
 name:UIScreenDidDisconnectNotification object:nil];
```

与设备相连的屏幕，有可能是通过线缆连接的，也有可能是通过 AirPlay 无线连接的。接到上述通知之后，我们应该查询屏幕数量，并根据新的屏幕个数来调整用户界面。

开发者应该在设备接入新屏幕的时候设置好视窗，并且在设备与屏幕断开时释放视窗。每个屏幕都应该有自己的视窗，以便管理该屏幕中的内容。设备与屏幕断开连接之后，就不要再持有那个视窗了，而是应该将其释放，等下次接入屏幕的时候再去重建。

---

 **提示** 通过镜像方式接入的屏幕是不计入 screens 数组的。镜像保存在主屏幕的 mirroredScreen 属性中。如果设备禁用镜像功能、尚未连接镜像屏幕，或是不支持镜像，那么该属性就是 nil。

如果创建了新的屏幕对象，并用它来向外接屏幕里写入内容，就会覆盖现有的镜像功能。只要程序开始向外接屏幕里写入内容，它就会优先于镜像，即使用户启用了镜像也是如此。

---

### 14.12.2 获取屏幕分辨率

每个屏幕都提供了 availableModes 属性。该属性是个由 UIScreenMode 对象所构成的数组，其中的对象按照分辨率从低到高排列。每个 UIScreenMode 都有 size 属性，用来表示目标屏幕所支持的某种分辨率。很多屏幕都支持多种分辨率模式。比方说，VGA 显示器可能会提供多达六种或六种以上的分辨率模式。支持的模式数量由硬件决定。显示器至少能够支持一种分辨率模式，如果能够支持多种模式，那么应该向用户提供选项。

### 14.12.3 配置视频输出

从 [UIScreens screens] 数组获取到表示外接屏幕的对象之后，应该查询可供使用的分辨率模式，并从中选择一种。一般来说，列表中的最后一种模式总是分辨率最高的模式，而列表中的首个模式，则是分辨率最低的模式。

若想启动 Video Out 流，则需新建 UIWindow，并按照选定的分辨率模式来设置其大小。然后给视窗添加视图，用以绘制内容。接下来，把外部屏幕同视窗相关联，并调用视窗的 makeKeyAndVisible 方法。该方法令视窗能够显示在外接屏幕上面，并且使开发者可以使用该视窗。最后，重新在原来的视窗上面调用 makeKeyAndVisible 方法。这使得用户可以继续操作主屏幕。不要忽略最后这一步。否则，用户会发现设备无法响应触摸。下面这段代码实现了必要的步骤，从而把外接屏幕设置好：

```
self.outputWindow = [[UIWindow alloc] initWithFrame:theFrame];
outputWindow.screen = secondaryScreen;
[outputWindow makeKeyAndVisible];
[delegate.view.window makeKeyAndVisible];
```

### 14.12.4 添加 CADisplayLink

CADisplayLink 是一种计时器，可以将绘制操作与显示器的刷新率相同步。开发者可以修改 CADisplayLink 的 `frameInterval` 属性，以调整每次刷新时必须经过的帧数。它的默认值是 1。值越大，刷新率越低。将它设置成 2，可以令刷新率减半。开发者应该在显示器与设备相连的时候创建 CADisplayLink。UIScreen 类中有个方法，可以返回与该屏幕相对应的 CADisplayLink 对象。该方法需要指定待调用的目标和选择子。

CADisplayLink 会定期触发其目标，使得开发者知道何时应该更新 Video Out 屏幕。如果想把 CPU 负载降下来，就可以调高间隔值，但这样做会令帧速率下降。在刷新率与 CPU 负载之间权衡，是一件很重要的事，直接操纵界面对设备 CPU 的响应能力要求很高，设计这种程序时，更应注意此问题。

解决方案 14-7 中的代码，依照常规方式来使用运行循环，这样做延迟最小。每次用完 CADisplayLink 之后，就立刻令其失效，然后把它从运行循环里面移除。

### 14.12.5 对过扫描进行补偿

开发者可以为 UIScreen 类的 `overscanCompensation` 属性设置某种值，以指定显示器应该如何补偿因处于屏幕边缘而损失掉的像素。该属性的取值请参阅苹果公司的文档，它的大概意思是说：显示器是应该裁切程序的内容，还是应该缩小程序的内容并给屏幕边缘填上黑边。

### 14.12.6 VIDEOkit

解决方案 14-7 创建了名为 VIDEOkit 的客户端，用来演示外接屏幕的基本用法。它会演示使用有线或无线外接屏幕时所需的每个步骤。我们调用 `startupWithDelegate:` 来对外接屏幕进行监控，并把负责为外接屏幕创建内容的主视图控制器传给该方法。

VIDEOkit 内部的 `init` 方法会监听屏幕的连接和断开事件，并按照需要来构建或释放视窗。当 CADisplayLink 对象触发回调的时候，它会执行名为 `updateExternalView:` 的非正式委托方法。而在执行该方法时，它会传入一个视图，此视图位于外部显示器的视窗之中，委托方法可以在这个视图上面绘制内容。

本条解决方案的范例代码会把某种颜色值保存到视图控制器的实例变量中，然后在控制器所实现的委托方法里面以这种颜色来渲染外部显示器：

```
- (void)updateExternalView:(UIImageView *)aView
{
    aView.backgroundColor = color;
}

- (void)action:(id)sender
{
    color = [UIColor randomColor];
}
```

用户点击按钮时，视图控制器会生成一种新颜色。而当 VIDEOkit 请求视图控制器去更

新外部视图的时候，控制器会把这种颜色设置成视图的背景色。于是，外部显示器的屏幕立刻就会变成这种新的随机色。



**提示** 在调试 AirPlay 的时候，Reflector 是个非常方便的辅助程序（单机授权的价格是 12.99 美元，5 台电脑授权的价格是 54.99 美元，网站是 <http://reflectorapp.com>），可以在既不使用线缆又不使用 Apple TV 的前提下，把设备屏幕中的内容传播到 Mac 及 Windows 电脑上。该软件会模拟 Apple TV 的 AirPlay 接收器，从而使开发者可以把 iOS 设备屏幕里的内容直接传播到电脑桌面，并记录设备的输出。

#### 解决方案 14-7 VIDEOkit

```
@protocol VIDEOkitDelegate <NSObject>
- (void)updateExternalView:(UIView *)view;
@end

@interface VIDEOkit : NSObject
@property (nonatomic, weak) UIViewController<VIDEOkitDelegate> *delegate;
@property (nonatomic, strong) UIWindow *outputWindow;
@property (nonatomic, strong) CADisplayLink *displayLink;
+ (void)startupWithDelegate:
    (UIViewController<VIDEOkitDelegate> *)aDelegate;
@end

@implementation VIDEOkit
{
    UIImageView *baseView;
}

- (void)setupExternalScreen
{
    // Check for missing screen
    if (!SCREEN_CONNECTED) return;

    // Set up external screen
    UIScreen *secondaryScreen = [UIScreen screens][1];
    UIScreenMode *screenMode =
        [[secondaryScreen availableModes] lastObject];
    CGRect rect = (CGRect){.size = screenMode.size};
    NSLog(@"Extscreen size: %@", NSStringFromCGSize(rect.size));

    // Create new outputWindow
    self.outputWindow = [[UIWindow alloc] initWithFrame:CGRectZero];
    _outputWindow.screen = secondaryScreen;
    _outputWindow.screen.currentMode = screenMode;
    [_outputWindow makeKeyAndVisible];
    _outputWindow.frame = rect;

    // Add base video view to outputWindow
```

```

        baseView = [[UIImageView alloc] initWithFrame:rect];
        baseView.backgroundColor = [UIColor darkGrayColor];
        [_outputWindow addSubview:baseView];

        // Restore primacy of main window
        [_delegate.view.window makeKeyAndVisible];
    }

    - (void)updateScreen
    {
        // Abort if the screen has been disconnected
        if (!SCREEN_CONNECTED && !_outputWindow)
            self.outputWindow = nil;

        // (Re)initialize if there's no output window
        if (SCREEN_CONNECTED && !_outputWindow)
            [self setupExternalScreen];

        // Abort if encounter some weird error
        if (!self.outputWindow) return;
        // Go ahead and update
        SAFE_PERFORM_WITH_ARG(_delegate,
            @selector(updateExternalView:), baseView);
    }

    - (void)screenDidConnect:(NSNotification *)notification
    {
        NSLog(@"Screen connected");
        UIScreen *screen = [[UIScreen screens] lastObject];

        if (_displayLink)
        {
            [_displayLink removeFromRunLoop:[NSRunLoop currentRunLoop]
                forMode:NSRunLoopCommonModes];
            [_displayLink invalidate];
            _displayLink = nil;
        }

        self.displayLink = [screen displayLinkWithTarget:self
            selector:@selector(updateScreen)];
        [_displayLink addToRunLoop:[NSRunLoop currentRunLoop]
            forMode:NSRunLoopCommonModes];
    }

    - (void)screenDidDisconnect:(NSNotification *)notification
    {
        NSLog(@"Screen disconnected.");
        if (_displayLink)
        {
            [_displayLink removeFromRunLoop:[NSRunLoop currentRunLoop]
                forMode:NSRunLoopCommonModes];
            [_displayLink invalidate];
        }
    }

```

```

        self.displayLink = nil;
    }
}

- (instancetype)init
{
    self = [super init];
    if (self)
    {
        // Handle output window creation
        if (SCREEN_CONNECTED)
            [self screenDidConnect:nil];

        // Register for connect/disconnect notifications
        [[NSNotificationCenter defaultCenter]
         addObserver:self selector:@selector(screenDidConnect:)
         name:UIScreenDidConnectNotification object:nil];
        [[NSNotificationCenter defaultCenter] addObserver:self
         selector:@selector(screenDidDisconnect:)
         name:UIScreenDidDisconnectNotification object:nil];
    }
    return self;
}

- (void)dealloc
{
    [self screenDidDisconnect:nil];
}

+ (VIDEOkit *)sharedInstance
{
    static dispatch_once_t predicate;
    static VIDEOkit *sharedInstance = nil;
    dispatch_once(&predicate, ^{
        sharedInstance = [[VIDEOkit alloc] init];
    });
    return sharedInstance;
}

+ (void)startUpWithDelegate:
    (UIViewController <VIDEOkitDelegate> *)aDelegate
{
    [[self sharedInstance] setDelegate:aDelegate];
}

@end

```

### 14.13 追踪用户

对于开发者来说，追踪（tracking）是一件无法避免的事情。UIDevice 类里面有个

属性，能够提供与设备硬件相关联的唯一标识符，但是苹果公司已经弃用该属性，并用另外两个标识符属性来取代它。ASIdentifierManager 类的 advertisingIdentifier 属性会返回针对当前设备的唯一标识符，以供发布广告之用。而 UIDevice 类的 identifierForVendor 属性则会提供与每个程序开发商相关的标识符。对于同一台设备上由同一家厂商所开发的各种程序来说，这个唯一标识符是相同的。但它并不是客户 ID (customer ID)。不同的开发商所制作的程序，返回的标识符是不同的，而同一个开发商所制作的程序，在不同的设备上也会返回不同的标识符。

这些标识符是用新的 NSUUID 类来构建的。在追踪之外的场合，也可以使用该类来创建 UUID 字符串，它能保证这些字符串的全局唯一性。苹果公司的文档中说：“UUID (Universally Unique Identifier, 全局唯一标识符) 也叫作 GUID (Globally Unique Identifier, 全局唯一标识符) 或 IID (Interface Identifier, 接口标识符、界面标识符)，它是个 128 位的二进制值。UUID 能够在空间和时间上具备唯一性。它是由两个值组合起来的，第一个值是针对生成 UUID 的这台电脑所选取的特定值，第二个值以 100 纳秒为单位来描述当前时刻与 1582 年 10 月 15 日 0 时 0 分 0 秒的间隔。”

UUID 类可以根据需要生成符合 RFC 4122v4 标准的新版 UUID。[NSUUID UUID] 方法会返回新的 UUID 实例 (其中的全部字母都是大写)。有了这个实例之后，我们可以通过 UUIDString 属性来获取对应的字符串，也可以用 getUUIDBytes: 方法来查询它的字节形式。

## 14.14 查询可用的磁盘空间

NSFileManager 类可以给出 iPhone 上面的可用空间以及设备的总空间。程序清单 14-1 示范了如何检测并显示这两个值，它会在数值中插入逗号，以美化字符串的格式。这两个值都以字节为单位，分别表示设备的总空间及可用空间。

程序清单 14-1 获取文件系统的大小及可用空间

```
- (void)logFileSystemAttributes
{
    NSFileManager *fm = [NSFileManager defaultManager];
    NSDictionary *fsAttr =
        [fm attributesOfFileSystemForPath:NSHomeDirectory()
         error:nil];

    NSNumberFormatter *numberFormatter =
        [[NSNumberFormatter alloc] init];
    numberFormatter.numberStyle = NSNumberFormatterDecimalStyle;

    NSNumber *fileSystemSize =
        [fsAttr objectForKey:NSFileSystemSize];
    NSLog(@"System space: %@ bytes",
          [numberFormatter stringFromNumber:fileSystemSize]);
```



```

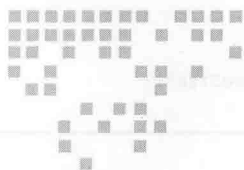
NSNumber *fileSystemFreeSize =
    [fsAttr objectForKey:NSFileSystemFreeSize];
NSLog(@"System free space: %@ bytes",
    [numberFormatter stringFromNumber:fileSystemFreeSize]);
}

```

## 14.15 小结

本章讲解了与 iOS 设备互动的一些关键方式。读者学到了如何获取设备信息、如何检查电池状态以及如何订阅与距离感应器有关的事件。我们还学习了怎样区分 iPod touch、iPhone 和 iPad，以及怎样判断程序所在的设备是什么型号。笔者介绍了加速计，并通过几个例子演示了其用法，告诉大家如何找出真正的“上”方向、如何移动屏幕上的物体，以及如何检测晃动操作。此外，还介绍了 Core Motion 框架，并告诉大家怎样用块来实时地响应设备事件。读者也学到了如何令应用程序把内容输出到外接屏幕中。下面列出本章各条解决方案所涉及的要害：

- App Store 允许开发者执行一些底层调用。苹果公司的 API 会随着当前固件版本而变化，但这些底层调用却不依赖于 API。虽说 UNIX 系统调用看上去比较麻烦，但是很多系统调用都能够在 iOS 设备上面执行。
- 向 iTunes 提交程序时，可在 Info.plist 文件里面规定设备必须具备何种能力，方能运行本程序。iTunes 将会根据文件中所列出的要求来判断程序是否能够下载并正确运行在某台设备之中。
- 开发程序时，应该考虑到设备的各种限制。在执行大批量的文件操作之前，应该先检查可用的磁盘空间，而在运行 CPU 负载很高的操作之前，则应判断电池的电量。
- 研究一下 Core Motion 框架。该框架可以实时提供设备的反馈信息，从而令我们能够以此为基础，将 iOS 设备同真实的运动效果结合起来。
- iPhone 与 iPad 的加速计提供了一种新颖的手段，用以补充基于触摸的界面。除了给应用程序设计触摸界面之外，还可以利用加速计的数据，令用户能够以倾斜设备的方式来操作程序。
- AirPlay 是一种能够把程序内容分享到外接屏幕的新技术，我们可以用 Video Out（视频输出）创建出许多原来无法想象的优秀软件项目。将 AirPlay 和外部显示器结合起来之后，iOS 设备就成了一台遥控器，用户可以把它当作游戏手柄，也可以用它的小屏幕来操作一些工具软件，以便把内容显示在大屏幕上。



## Chapter 15

## 第 15 章

## 辅助功能

通过辅助功能，开发者可以把 iOS 应用程序提供给身体有障碍的人士使用。系统的 General Settings 里面包含一些辅助功能，可以把显示的内容放大，并反转界面颜色等。而对于开发者来说，辅助功能主要是围绕着 VoiceOver 来实现的，它可以令视障用户“听”到程序的 GUI。VoiceOver 能够以声音来描述程序的图形界面。

不要把 VoiceOver 与 Voice Control 或 Siri 语音助手相混淆。VoiceOver 能够以声音来描述 UI，并且与手势结合得非常紧密，而后两者则是苹果公司的语音识别专利技术，能够在不用手操作的前提下控制手机（hands-free interaction，免提交互）。

本章简述 VoiceOver 辅助功能。读者将会学到怎样给应用程序添加辅助标签和提示，以及如何在模拟器与 iOS 设备上面测试这些功能。辅助功能可以在第三代及后续设备上面使用并测试，包括所有型号的 iPad、3GS 及后续型号的 iPhone，以及第三代及后续机型的 iPod touch。

## 15.1 辅助功能基础知识

为 UI 元件添加描述性的属性，即可创建辅助功能。这套编程接口由名为 UIAccessibility 的非正式协议来定义，该协议包含一系列属性，其中包括标签（label）、提示（hint）以及值（value）等。这些内容合在一起，向 VoiceOver 提供信息，使它可以把程序界面用语音描述出来。

开发者可以在代码中设置这些属性，也可以通过 Interface Builder（IB）来添加它们。程序清单 15-1 演示了怎样设置按钮的 accessibilityHint 属性。该属性描述了这个按钮控件如何响应用户的操作。本例中，如果用户在相关的文本框里输入用户名，那么按钮的 accessibilityHint 就会随之更新。更新后的 accessibilityHint 将与当前的 UI 情

境相符。程序不再宽泛地提示此按钮可以打电话，而是会具体地说出它能打给谁。

程序清单 15-1 以编程的方式更新辅助功能信息

```
- (BOOL)textField:(UITextField *)textField
shouldChangeCharactersInRange:(NSRange)range
replacementString:(NSString *)string
{
    // Catch the change to the username field and update
    // the accessibility hint to mirror that
    NSString *username = textField.text;
    if (username && username.length > 1)
        callbutton.accessibilityHint = [NSString
            stringWithFormat:@"Places a call to %@", username];
    else
        callbutton.accessibilityHint =
            @"Places a call to the person named in the text field.";
    return YES;
}
```

UIAccessibility 协议包含如下属性：

- **accessibilityTraits**——用于表述 UI 元件的一系列标志。这些标志指明了控件的行为，并告诉解释系统应该如何对待此控件。比方说，有一些标志与状态相关，描述了控件是否处于受选或启用状态，还有一些与行为相关，描述了控件的行为是否与按钮类似。
- **accessibilityLabel**——该属性是描述视图角色或控件动作的短语（例如 Pause 或 Delete）。标签可以本地化。
- **accessibilityHint**——该属性是个短语，用来描述用户可以通过此控件执行何种操作（例如转到主页）。该属性也可以本地化。
- **accessibilityFrame**——该矩形描述了非视图的元件应该如何显示在屏幕上面。对于普通的 UIView 视图来说，该属性就是其 frame 属性。
- **accessibilityPath**——对于非矩形的元件来说，可以用 UIBezierPath 来描述它的形状，而不使用 accessibilityFrame 来描述。
- **accessibilityValue**——它是与控件相关联的值，比如滑杆的当前值（例如 75%）或开关的状态（例如 ON）。

## 用 IB 来设置辅助功能

开发者可以通过 IB 的 Identity Inspector > Accessibility 面板（如图 15-1 所示）给界面中的 UIKit 元件添加辅助功能。这些文本框与其中的文本在辅助功能中扮演着不同的角色。IB 的面板里所展示的选项，与 UIAccessibility 协议中的属性是相互对应的。Label 用于指明视图的角色，Hint 用来描述具体的操作，这与通过代码设置 UIAccessibility 属性是一样的。除了上述文本框之外，还有个通用的 Accessibility Enabled 复选框，以及一些与 accessibilityTraits 有关的复选框。

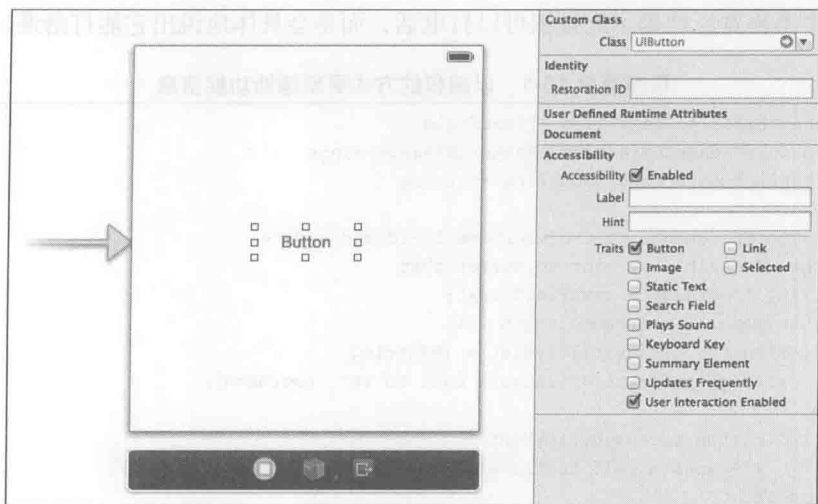


图 15-1 开发者可以在 IB 的 Identity Inspector 里面指定控件的辅助功能信息

## 15.2 启用辅助功能

Enabled 复选框用来决定 UIKit 视图是否与 VoiceOver 协同运作。如果想宣称某元件支持辅助功能，需要以代码将 `isAccessibilityElement` 属性设为 YES，或在 IB 里面选中 Accessibility Enabled 复选框（如图 15-1 所示）。这个布尔属性用来表示 GUI 元件是否参与到辅助系统之中。默认情况下，所有 UIControl 实例的 `isAccessibilityElement` 属性都是 YES。

一般来说，可以无障碍访问的子视图所在的容器不开启辅助功能，其他视图都应该开启辅助功能。也就是说，只应该在用户能够直接操作或直接看到的控件上面启用辅助功能。用来容纳其他视图的容器在语音展示系统里面并没有实质作用，所以应将其排除在外。

表格视图的单元格能够很好地说明什么是可以无障碍访问的控件所在的容器（容器就是包含其他对象的对象）。下面两条规则适用于单元格：

- 如果单元格里没有嵌入控件，那么该单元格应该开启辅助功能。
- 如果单元格里嵌入了控件，那么单元格本身不应开启辅助功能，但它里面的子控件应开启辅助功能。

不提供辅助功能的容器，用来报告其中包含多少带有辅助功能的子视图，以及这些子视图分别是什么。请参阅苹果公司的《Accessibility Programming Guide for iOS》，以便深入了解如何为与辅助功能有关的容器编写代码。对于自制的容器视图来说，开发者需要为其声明并实现 `UIAccessibilityContainer` 协议。

## 15.3 特征

特征（trait）描述了 UIKit 控件的行为。VoiceOver 采用特征来描述程序的界面。如

图 15-1 所示，开发者可以给视图设置许多特征。我们可以在选定的视图上面设置它应有的特征，也可以通过编程的方式来修改特征。

特征可用来向 VoiceOver 系统描述界面中的元件。它们指明了控件的行为，以及 VoiceOver 应该如何对待此控件。开发者通过 `accessibilityTraits` 属性来设置特征，我们可以只设置一个标志，也可以用 OR 把多个标志组合起来，还可以像图 15-1 那样在 IB 里面设置标志。这些特征标志的运作方式不同，VoiceOver 使用它们的方式也不同。

最基本的标志即默认的标志是“没有特征”：

- **UIAccessibilityTraitNone**——该元件没有与辅助功能有关的特征。

除此之外，还有一些标志用来描述这个元件是什么，包括：

- **UIAccessibilityTraitButton**——该元件是按钮。

- **UIAccessibilityTraitLink**——该元件是超链接。

- **UIAccessibilityTraitStaticText**——该元件是一段不变的文本。

- **UIAccessibilityTraitSearchField**——该元件是搜索框。

- **UIAccessibilityTraitImage**——该元件是图像。

- **UIAccessibilityTraitKeyboardKey**——该元件是键盘上的按键。

- **UIAccessibilityTraitHeader**——该元件是某段内容的标题。

苹果公司的辅助功能开发文档里面说，Button、Link、Static Text 及 Search Field 这四个标志是互斥的，开发者只应选择其中之一。如果某个按钮本身也是链接，则要么选择 Button，要么选择 Link，不能两个都选。我们应该选择最能描述按钮特征的标志。如果按钮可以显示图像，并且能在点击的时候发出声音，那么可以随意指定与图像及声音有关的特征，而不必考虑互斥问题。

下面几个状态标志可用来描述控件是否受选、是否可以调整以及是否允许用户直接操作它：

- **UIAccessibilityTraitSelected**——表示该控件当前处于受选状态，可用来描述分段选择控件中的分段或表格中的行。

- **UIAccessibilityTraitNotEnabled**——表示该控件已经禁用，用户无法操作它。

- **UIAccessibilityTraitAdjustable**——表示该控件可以有多个取值，比如滑杆或选取器控件就是这样。开发者可以实现 `accessibilityIncrement` 及 `accessibilityDecrement` 方法，以规定用户每次能在当前值的基础上调整多少。

- **UIAccessibilityTraitAllowsDirectInteraction**——表示用户是否能够通过触摸来直接操作该控件。

如果控件在用户操作它的时候会发出声音，那么也可以指定下面这个标志：

- **UIAccessibilityTraitPlaysSound**——表示该控件会在激活时发出声音。

最后，还有一些状态标志用来描述控件的行为，并告诉用户该控件如何与外部环境互动：

- **UIAccessibilityTraitUpdatesFrequently**——表示该控件变化得很频繁，所以用户不需要经常了解它的状态变化，比方说秒表的读数就是如此。

- **UIAccessibilityTraitStartsMediaSession**——表示该控件会启动一段媒体会话。播放或录制音频、视频时，可以使用该标志，以防 VoiceOver 打断。

- **UIAccessibilityTraitSummaryElement**——表示该控件会在应用程序中提供摘要信息，例如当前的设置或状态等。
- **UIAccessibilityTraitCausesPageTurn**——表示 VoiceOver 读完控件中的文本之后，该控件会自动翻页。

如图 15-1 所示，大部分（但不是所有）特征标志都可以通过 IB 的 Identity Inspector 面板来切换。如果需要更为细致地控制这些标志，需要使用代码来调整它们。

## 15.4 标签

`accessibilityLabel` 属性用来设置控件的辅助功能标签。好的标签通常会用一个词向用户描述出控件是什么。我们应该像设置按钮文本那样设置 GUI 的辅助功能标签。Edit、Delete 和 Add 等词都可以描述出控件的用途。这些词既适合用作按钮的文本，也适合用作辅助功能标签的文本。

`accessibilityLabel` 属性并不局限于按钮。对于文本视图、图像视图以及文本标签来说，也可以分别用 Feedback、User Photo 及 User Name 来描述其内容及功能。凡是在视觉界面中有意义的控件，都应该能在 VoiceOver 里面读出来。下面是几条 `accessibilityLabel` 的设计技巧：

- 不要把视图的类型写在标签里。例如，不要把标签写成“Delete button”（删除按钮）、“Feedback text view”（反馈文本视图）或“User Name text field”（用户名文本框）。由于 VoiceOver 会自动添加视图类型信息，所以如果在 identity 面板里把标签写成“Delete button”（删除按钮），那么 VoiceOver 会把它读成“Delete button button”（删除按钮按钮）。
- 将标签首字母大写，但不要在末尾加句点。VoiceOver 会根据大小写情况来决定发音时的语调。如果给标签末尾加了句点，那么 VoiceOver 一般会以降调来发音，这样读出来的语气与后面的控件类型名称听起来不搭调。“Delete. button”的发音听上去很怪，而“Delete button”的发音则是正确的。
- 把多条描述信息写在一个标签里。如果某些复杂的视图在功能上是一个整体，那么可以把这些视图的描述信息合起来写在一个标签里，并将该标签设置给上级视图。比方说，如果表格的某个单元格里有许多子视图，但这些子视图都不是独立的控件，那么可以考虑在单元格的标签里用一段文本把这些子视图全部描述出来。
- 只在用户直接操作的视图上面设置标签。如果用户需要直接操作子视图，那就在子视图级别上设置标签，而不要给包含子视图的上级视图设置标签。
- 本地化。将辅助功能字符串本地化，以尽量扩大程序的目标用户。

## 15.5 提示语

`accessibilityHint` 属性用来设置控件的提示信息。提示信息可以告诉用户操作该

控件的效果。当这种效果不太明显的时候,更应该指定提示信息。比方说,界面里有个人名是 John Smith,用户点击了名字之后,程序会给这个人拨电话。由于名字本身并没有描述出用户操作该控件后的效果,所以我们应该添加一条提示语,告诉用户操作该控件之后会发生什么事情。比方说,可以把 accessibilityHint 设置成“Places a phone call to this person”(给这个人打电话),或是设置成更为贴切的“Places a phone call to John Smith”(给 John Smith 打电话)。下面给出一些设置 accessibilityHint 的技巧:

- **采用句子的形式给出提示语。**提示语应该以大写字母开头,并以句点结尾。即便把隐含的主语省略掉,也依然要采用句子的形式。比方说,在省略主语“This button”(该按钮)的情况下,应该把提示语写成“Clears text in the form.”(清除表单中的文本)。采用这种形式可以保证 VoiceOver 能够以正确的音调来发声。
- **动词用来描述控件所做的事情,而不是用户所做的事情<sup>①</sup>。**我们应该告诉用户“[This text label] Places a phone call to this person.”([该文本标签用来]给这个人打电话),而不是“[You will] Place a phone call to this person.”([你会]给这个人打电话)。
- **不要写上 GUI 元件的名称或类型。**不要在提示语里面提到待操作的 UI 控件。GUI 的名字(也就是它的标签,比如“Delete”)及类型(也就是它所属的类,比如“button”)都不应该出现。VoiceOver 会根据需要自动添加这些信息,我们不要叫它读出多余的词。假如把这两个信息都写入提示语,那么说出来的话就会变成“Delete button [这个 button 是标签里的] button [这个 button 是 VoiceOver 自动补充的] button [这个 button 是提示语里的] removes item from screen.”。使用简洁的“Removes item from screen.”就好。
- **不要提到操作方式。**不要在提示语里给出用户操作该控件的方式。不要把提示语写成“Swiping places a phone call to this person”(通过滑动来给这个人打电话)或“Tapping places a phone call to this person”(点击控件来给这个人打电话)。由于 VoiceOver 使用它自己的一套手势来激活 GUI 元件,所以不要在提示语里直接描述手势。
- **把效果描述得详细一些。**描述控件的效果时,“Places call”(打电话)这个说法不如“Places a call to this person”好(给这个人打电话),而它又不如“Places a call to John Smith”(给 John Smith 打电话)好。提示语要能够简短而详尽地告诉用户该操作的效果,但又不能太过简单,以致用户必须猜测才能知道结果。不要使用那种用户必须听第二遍才能明白的提示语。
- **本地化。**与标签一样,提示语也应该本地化,以便尽量扩大目标用户。

## 15.6 用模拟器测试辅助功能

部署到 iOS 设备之前,可以用模拟器的 Accessibility Inspector 来测试带有辅助功能的程序。模拟器的 Accessibility Inspector 能够在不直接使用 VoiceOver 手势的前提下,模仿 VoiceOver 的操作效果,并通过浮动面板来提供即时的视觉反馈(但不会发出声音)。由于很

① 动词应该酌情使用第三人称单数形式。——译者注



多 VoiceOver 手势都无法用模拟器来重现（比方说 triple-swipe 手势（三指滑动、三指扫屏）及连续的 hold-then-tap 手势（按住然后点击）），所以 Accessibility Inspector 着重于表述界面元素，而不是响应 VoiceOver 手势。

打开 Settings > General > Accessibility，即可启用此特性。把 Accessibility Inspector 开关打开之后，Inspector 就会立刻出现在屏幕上，如图 15-2 所示。它会显示出当前选定的无障碍访问元件的设置。

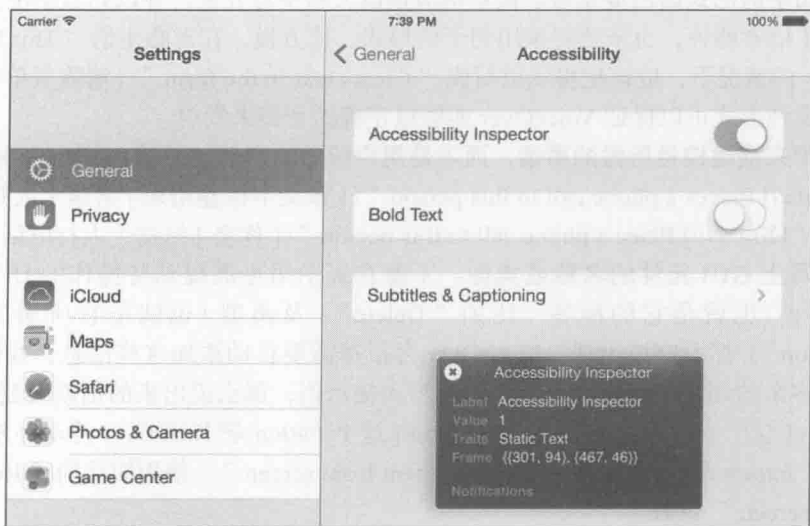


图 15-2 iPad 模拟器的 Accessibility Inspector 面板会显示出当前选定的 GUI 元件所具备的辅助功能属性，例如标签 (label)、提示语 (hint) 等



**提示** 苹果公司目前的 Xcode 5 开发工具，并不能正确地模拟出 Accessibility Inspector。苹果公司应该会在将来的更新中修复这一问题。此处给出的截图，是根据 Xcode 5 中的效果与 Xcode 4.6 里正确的 Inspector 效果合成出来的。Xcode 5 里正确的 Accessibility Inspector，应该与此处的截图相似。

启用与禁用 Inspector 的方法是：点击 Inspector 左上角的圆形 × 按钮。点击一次之后，就会禁用 Inspector，并将其变为一条线。再次点击，即可恢复并启用 Inspector。在大多数情况下，都应该禁用 Inspector，直到我们真的想检视某个 GUI 控件为止。图 15-3 演示了 Accessibility Inspector 中的按钮界面。

与 VoiceOver 一样，Accessibility Inspector 也会干扰正常的应用程序手势。它会减缓程序测试工作，所以只应该偶尔用一下（一般来说，应该在测试辅助功能的时候使用）。我们可以在禁用 Accessibility Inspector 但将其保留的前提下启动应用程序，等切换到需要调试辅助功能的画面时再启用它。



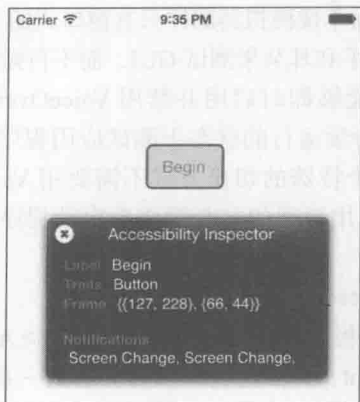


图 15-3 在 Accessibility Inspector 中可以看到开发者通过 IB 或代码为当前所选控件设定的值

如果用编程的方式来修改辅助功能的提示语，那么 Accessibility Inspector 会实时更新，以反映这些变化。把 Accessibility Inspector 激活之后，就可以观察到当前的提示语何时发生了变化了，它会确保屏幕上显示的提示语和标签与界面中的相符。

## 15.7 把变化情况传播出去

—— 屏幕上的元素如果不是因为用户的直接操作而发生变化，应用程序就应该向 VoiceOver 辅助功能系统投递通知，令其知晓这一情况：

- 添加或移除 GUI 元件时，UIAccessibilityLayoutChangedNotification 通知可以令 VoiceOver 辅助功能系统立刻获知相关变化。
- 完成滚动操作之后，程序应该投递 UIAccessibilityPageScrolledNotification 通知。通知对象应该包含一条描述新位置的信息（例如，“Page 5 of 17”（17 页中的第 5 页）或“Tab 2 of 4”（4 个分页中的第 2 个分页））。
- 如果放大后的控件有了变化，那么应该发送 UIAccessibilityZoomFocusChanged 通知。发送通知时还应该指定 type、frame 及 view 这三个参数。第一个参数表示放大类型，第二个参数表示当前放大的 frame（以屏幕坐标来衡量），第三个参数表示包含放大的 frame 的视图。

除了上面这些通知之外，还可以通过 VoiceOver 辅助系统来广播通用的布告。UIAccessibilityAnnouncementNotification 接收一个参数，表示包含布告内容的字符串。如果 GUI 发生了微小的变化，屏幕上发生了短暂的变化，或是发生了不会直接影响 UI 的变化，那么可以用它来通知用户。

## 15.8 在 iOS 上面测试辅助功能

对于辅助功能的开发来说，在 iPhone 或 iPad 上面进行测试是很重要的一部分。设备上面可

以看到 VoiceOver 的实际效果，而不像模拟器那样只有窗口式的 Inspector。开发者可以听到用户应该会听到的声音，并且可以用手和耳朵来测试 GUI，而不再需要去观察 Inspector 中的内容了。

与模拟器一样，iPhone 也能够即时启用并禁用 VoiceOver。我们可以在 Settings 中启用 VoiceOver，然后在 VoiceOver 持续运行的状态下测试应用程序。不过，还有个特殊的切换方式，要比这个办法更简单。这个特殊的切换方式不需要用 VoiceOver 手势从 Settings 切换到程序，可以把 VoiceOver 关掉，用普通的 iOS 操作来启动程序，然后等到需要测试的时候再启用 VoiceOver。

按照下列步骤即可切换 VoiceOver：

1. 打开 Accessibility 设置面板。通过 Settings > General > Accessibility 即可找到这个面板。
2. 找到 Accessibility Shortcut，并点击它，然后会看到一系列辅助功能操作，它们可以用作三击 Home 按钮时的目标动作。
3. 把 VoiceOver 选为三击 Home 按钮的目标动作。选好之后（其右边会出现对勾），我们就可以通过连续点击三次 iOS 设备的 Home 键来启用或禁用 VoiceOver 了。用户可以听到语音提示，以确认当前的 VoiceOver 设置。

这种 VoiceOver 切换方式，使得开发者在操作应用程序时可以跳过很多繁琐的三指拖放和多级按钮点击操作。不过，我们也应该熟悉 VoiceOver 的手势和操作方法。表 15-1 列出了测试程序时可以用到的 VoiceOver 手势。

表 15-1 操作程序时常用的 VoiceOver 手势

任 务	VoiceOver 手势
切换 VoiceOver	三击设备的 Home 按钮
切换 ScreenCurtain	三指三击屏幕（也就是用三根手指三次点击屏幕）
切换 VoiceOver 语音	用三根手指双击屏幕，即可彻底切换 VoiceOver 语音功能（而不是只针对某个条目）
停止说出当前条目	用两根手指双击屏幕。重复这一手势，即可继续语音播报。如果在未启用 VoiceOver 的主屏幕做这个手势，那么它的功能是停止播放或继续播放音乐
激活某条目（例如激活某个按钮）	方式 1：用一根手指点击并按下某个条目不放，然后用另一根手指点击屏幕 方式 2：点击某个条目以选中它。然后双击屏幕，激活此条目
调整文本插入点	选中可以编辑的文本视图或文本框之后，用一根手指向上或向下拨动屏幕。根据设备的配置，插入点可能会移动一个字符，也可能会移动一个词
访问 VoiceOver 菜单，以调整语音设置	选中可以编辑的文本视图或文本框之后，把两根手指放在屏幕上，然后顺时针或逆时针旋转。这个手势叫作 rotor
选中文本或取消选择文本	设定插入点并进入文本编辑模式。双指张开可以选中文本，双指聚拢可以取消选择文本
输入文本	选中文本框或文本视图，然后双击屏幕，进入文本编辑模式。此时键盘会显示在屏幕上 方式 1：用左手食指点击并按住键盘上的某个键。用右手食指点击屏幕上的其他地方。如果想反复使用 Delete 键，这就是最好的办法 方式 2：点击某个键以选中它。双击屏幕以敲击该键

(续)

任 务	VoiceOver 手势
移动滑杆	选中滑杆, 然后用一根手指向上或向下拨动屏幕, 以调整它的值
向上或向下翻页	用三根手指向上或向下拨动屏幕
向左或向右翻页	用三根手指向左或向右拨动屏幕
选中并说出某条目	点击某条目
逐字符或逐单词说出选中的条目	用一根手指向上或向下拨动屏幕。VoiceOver 会根据 rotor 菜单中的设置来发音
移到下一个或上一个条目	用一根手指向左或向右拨动屏幕
读出整个屏幕中的内容	用两根手指向上拨动屏幕。这种操作方式未必能够见效。也可以改用另一种方式: 反复向左拨动屏幕, 移到首个条目。然后用双指向下拨动的手势从当前选中的条目向后读
解锁	选定解锁用的滑杆, 然后单指双击屏幕

特别注意 Screen Curtain 功能, 它可以令设备屏幕变为空白, 从而使开发者可以真正测试基于语音的界面。启用 Screen Curtain, 然后操作 iPhone 的计算器程序, 试试如何在看不到屏幕内容的情况下使用该程序。

## 15.9 语音合成

苹果公司在 iOS 7 中添加了文字转语音的功能, 这对于辅助功能和其他任务来说都是非常有用的工具, 可以帮助用户浏览内容或增加程序的趣味。可以用 AVSpeechSynthesizer 和 AVSpeechUtterance 类来说出任意字符串。对于长篇文本来说, 这项功能非常方便, 它令开发者可以获得比使用 VoiceOver 时更为精细的控制权, 从而能够以编程的方式控制语音, 包括选定发音内容和时机, 以及调整音调和语速等。此外, 即使用户不使用辅助功能, 语音合成也依然有效。

程序清单 15-2 演示了如何从可供使用的英语发音中随机选择一种, 并以它来说出简单的字符串。只需稍微修改一下代码, 就可以换用其他语言和地域口音来发声了。使用 AVSpeechSynthesisVoice 可以选定某一种语言, 也可以遍历可供使用的各种语言。

程序清单 15-2 使用 iOS 7 的文字转语音功能

```

- (void)action
{
    // Establish a new utterance
    AVSpeechUtterance *utterance =
        [AVSpeechUtterance speechUtteranceWithString:
         @"Hello there you beautiful world!"];
    // Slow down the rate
    utterance.rate = AVSpeechUtteranceMinimumSpeechRate +
        (AVSpeechUtteranceMaximumSpeechRate -
         AVSpeechUtteranceMinimumSpeechRate) * 0.2f;
}

```

```

    // Set the language
    utterance.voice = [self anotherVoiceForLanguage:@"en"];

    // Speak
    AVSpeechSynthesizer *synthesizer =
        [[AVSpeechSynthesizer alloc] init];
    [synthesizer speakUtterance:utterance];
}

- (AVSpeechSynthesisVoice *)anotherVoiceForLanguage:
    (NSString *)lang
{
    srand(time(NULL));
    NSArray *voices = [AVSpeechSynthesisVoice speechVoices];
    NSMutableArray *voicesForLanguage =
        [[NSMutableArray alloc] init];
    for (AVSpeechSynthesisVoice * voice in voices)
    {
        if ([voice.language hasPrefix:lang])
            [voicesForLanguage addObject:voice];
    }
    NSUInteger voiceIndex =
        rand() % voicesForLanguage.count;
    return voicesForLanguage[voiceIndex];
}

```

## 15.10 动态字体

一直以来，iOS 的辅助功能都可以令应用程序与某些能力相配合，并适应很多限制。iOS 7 已经把这套设计理念融入许多日常的应用程序之中。用户可以调整显示设置，以影响设备上所安装的全部程序。

General > Text Size 中的一项设置可以调整所有程序的阅读字体大小，包括字体高度、行高以及行间距（如图 15-4 所示）。视力较好的年轻用户可以把字体调小，以便在屏幕中显示更多内容。而视力较差的老年用户则可以通过拖曳滑块把字体调大。

要想在程序中使用动态字体（Dynamic Type），必须通过 preferredFont-ForTextStyle 方法来选择字体，并传入下列几种样式之一：UIFontTextStyleHeadline、UIFontTextStyleSubheadline、UIFontTextStyleBody、UIFontTextStyleFootnote、UIFontTextStyleCaption1 或 UIFontTextStyleCaption2。如果我们不直接指定字体名称和大小，而是采用这些预先配置好的文本样式，那么 iOS 会根据通用设定来选取样式及大小合适的字体。

为了响应用户对文本大小所做的修改，我们需要监听 UIContentSizeCategoryDidChangeNotification，并适当地更新 UI：

```

UIViewController __weak *weakSelf = self;

```

```

[[NSNotificationCenter defaultCenter]
 addObserverForName:UIContentSizeCategoryDidChangeNotification
 object:nil
 queue:[NSOperationQueue mainQueue]
 usingBlock:^(NSNotification *note) {
     UIViewController *strongSelf = weakSelf;
     [strongSelf performSelector:@selector(updateLayout)];
 }];

```

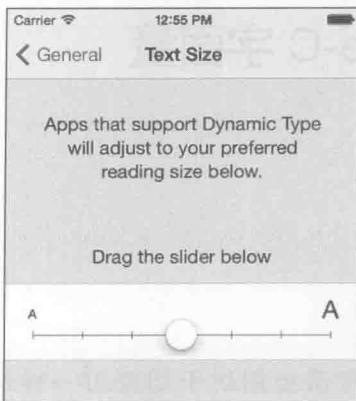


图 15-4 在 iOS 7 中，只需拖动系统设置中的一个滑杆，即可调整所有程序的文本大小

如果使用 Auto Layout 及标准的 UIKit 文本元件，那么大部分工作都会由系统自行处理。重置 UILabel 或 UITextView 的字体，一般都会使得控件的固有内容尺寸 (intrinsic ContentSize) 失效，并迫使它重新布局。但如果开发者是根据 frame 来手工布局，就需要调用 setNeedsLayout 来重新布局了。

## 15.11 小结

iOS 应用程序支持辅助功能之后，可以更加积极地参与到更为广泛的软件市场之中，并吸引更多的用户。下面列出本章要点：

- 和程序界面的本地化一样，如果给程序添加与辅助功能有关的标签和提示语，就可以吸引新的用户。添加这些信息所需的工作量很少，但却能给用户带来很大好处。
- 用 IB 给程序设计声音界面时，要分清标签和提示语的用途。
- 尝试用编程的方式来修改提示语。我们可以根据界面当前的情况来更新提示语，以便给视障人士提供最佳的用户体验。
- iOS 的辅助功能系统是不不断演化的。随时留意苹果公司的开发文档，以便了解最近的更新和变动。
- 在开启 Screen Curtain 的情况下测试程序。由于屏幕上不会显示内容，所以我们能够更好地模拟出用户通过 VoiceOver 来操作程序时的情景。

## Objective-C 字面量

写代码的时候，我们经常需要用饼干切割刀一样的 `[NSNumber numberWithInt:5]` 模板来制作 `NSNumber` 对象。有的开发者也许定义了一些宏，以简化编码工作。从 Xcode 4.4 (及 LLVM 4.0) 起，开发者可以通过精简且易读的表达式来使用 Objective-C 字面量，而不用再像原来那样，用复杂的代码去创建 `NSNumber` 及 `NSArray` 等实例。

笔者以前也定义并使用了一些可以简化 `NSNumber` 声明的宏，但是现在感觉使用字面量可以写出更易看懂且更为精简的代码。这些字面量令我们能够少输入一些代码，而且提供了一种自然而一致的代码风格。

现在不用再写各种各样的声明了，而是可以使用像 `@5` 这样简单的字面量。这种数字字面量与大家一直在使用的字符串字面量非常相似。字符串字面量就是在 `@` 符号后面跟上字符串常量 (比方说 `@"hello"`)，而数字字面量则是在 `@` 符号后面跟上数值。还有一些类似的字面量，可以简化 `NSDictionary` 及 `NSArray` 的创建及查询。

这项新技术令原来冗长的写法变得简洁了许多。

### A.1 数字

通过 LLVM Clang 编译器所提供的机制，开发者可以在 Xcode 中把整数及浮点数等标量值以数字字面量的形式封装到对象容器里面。只需给标量前面加上 `@`。比方说，下面代码可以把 2.7182818 转为对应的 `NSNumber` 对象：

```
NSNumber *eDouble = @2.7182818;
```

上面的数字字面量在功能上与下面写法等效：

```
NSNumber *eDouble = [NSNumber numberWithDouble: 2.7182818];
```

这种方式与原来的区别在于，编译器会自动把复杂的事情处理好。开发者不需要调用类的方法，不需要完整地写出方法，也不需要使用一对方括号。我们只需在数值前面加上 @ 就可以了，剩下的事情 Clang 会做。

通过标准的后缀可以定义浮点数 (F)、长整数 (L)、超长整数 (LL) 和无符号整数 (U)。下面举例说明这些后缀的用法。每条声明语句都很简单，我们不再需要调用与数值类型相对应的专门方法了：

```
NSNumber *two = @2; // [NSNumber numberWithInt:2];
NSNumber *twoUnsigned = @2U; // [NSNumber numberWithUnsignedInt:2U];
NSNumber *twoLong = @2L; // [NSNumber numberWithLong:2L];
NSNumber *twoLongLong = @2LL; // [NSNumber numberWithLongLong:2LL];
NSNumber *eDouble = @2.7182818; // [NSNumber numberWithDouble: 2.7182818];
NSNumber *eFloat = @2.7182818F; // [NSNumber numberWithFloat: 2.7182818F];
```

但是，Clang 规范并不支持采用这种形式来声明长浮点数，而且苹果公司的运行期系统也不支持长浮点数。所以，下面语句无法编译：

```
NSNumber *eLongDouble = @2.7182818L; // Will not compile
```

布尔型的常量 @YES 与 @NO 会产生与 [NSNumber numberWithBool:YES] 及 [NSNumber numberWithBool:NO] 相等价的对象。

最后要注意，在 Xcode 4.5 及后续版本中，开发者可以使用 @-5 这种写法。我们不再需要把值放到括号中。

## A.2 装箱

使用 Xcode 4.4 时，我们只能在 @ 符号后面写上字面标量常数。如果想先用算式表达某个值，然后将该值转为数值对象，就要使用传统形式的方法调用语句：

```
NSNumber *two = [NSNumber numberWithInt:(1+1)];
```

Xcode 4.5 支持装箱表达式，从而免去了上面这种麻烦的写法。所谓装箱表达式，就是一种可以解释并转换成 NSNumber 对象的值。装箱表达式两端要加括号，编译器会求出表达式的值，然后将其转为对象。例如：

```
NSNumber *two = @(1+1);
```

以及

```
int foo = ...; // some value
NSNumber *another = @(foo);
```

装箱表达式并不局限于数值，它们也适用于字符串。下面赋值语句可以求出 strstr() 的结果，并把结果转为 NSString 形式（也就是 @"World!"）：

```
NSString *results = @(strstr("Hello World!", "W"));
```

### 枚举

装箱表达式还有其他一些用途，比方说枚举。你也许认为定义好枚举之后，应该能够直

接将其名称放在 @ 符号后面使用，但这样做会有问题。例如，下面这个枚举的名字选得不好：

```
enum {interface, implementation, protocol};
```

你可能觉得下面语句能够创建值为 2 的 NSNumber：

```
NSNumber *which = @protocol;
```

假如允许上面写法的话，显然会相当糟糕。而通过装箱表达式来使用枚举，则不会和当前及将来以 @ 开头的关键字产生冲突：

```
NSNumber *which = @(protocol); // [NSNumber numberWithInt:2];
```

## A.3 容器字面量

容器字面量也是 LLVM Clang 编译器中一个非常有用的功能。在支持容器字面量之前，我们必须采用下面写法来创建字典及数组，这段代码创建了包含三个元素的数组，以及包含三个键的字典：

```
NSArray *array = [NSArray arrayWithObjects: @"one", @"two", @"three", nil];
NSDictionary *dict = [NSDictionary dictionaryWithObjectsAndKeys:
    @"value 1", @"key 1",
    @"value 2", @"key 2",
    @"value 3", @"key 3",
    nil];
```

上面写法比较冗长，而且需要以 nil 结尾。虽说这不一定是坏事，但很多人容易忘记写 nil，而且几乎每个开发者都曾遇到过这个问题。另外，声明字典的时候，必须先写 value（值），再写 key（键）。这与大部分人的认知相反，虽然 dictionaryWithObjectsAndKeys 这个方法名已经表达了正确的顺序，但很多人还是想先写 key，再写 value。

容器字面量引入了一种简单的新写法，从而解决了上面的两个问题。下面代码所声明的数组和字典，其内容与前面代码相同：

```
NSArray *array = @[@"one", @"two", @"three"];
NSDictionary *dict = @{
    @"key 1": @"value 1",
    @"key 2": @"value 2",
    @"key 3": @"value 3"
};
```

数组字面量的左右两侧带有方括号，括号里面是一串由逗号所分隔的元素。字典是由花括号括起来的一份列表，其中的键值对以逗号隔开，键和值之间用冒号关联。定义数组和字典时，不需要在结尾加上 nil。键值对的书写顺序也更为合理了，现在是先写键后写值，而不是像原来那样先写值后写键。

这两个表达式求值之后的结果，与通过传统方式所声明的两条语句相同。采用这种写法时，仍然需要遵守标准的容器规则：

- 不要添加值为 nil 的键或值。



- 每个元素都必须是对象指针。
- 键必须遵循 <NSCopying>。

## A.4 通过下标来访问元素

Clang 采用标准的取下标操作，以方括号来访问容器中的元素。换句话说，开发者可以像使用 C 语言数组那样来使用 NSArray。NSDictionary 也可以按照类似方式访问，只不过下标是键而不是数字。下面两行代码用来访问前一节所定义的数组和字典：

```
NSLog(@"%@", array[1]); // @"two"
NSLog(@"%@", dictionary[@"key 2"]); // @"value 2"
```

这种写法并不是只能用来查询值。我们也可以在这种新的写法给可变的实例赋值。下面是两条采用新式下标写法的简单赋值语句：

```
mutableArray[0] = @"first!";
mutableDictionary[@"some key"] = @"new value";
```

开发者仍然需要注意下标是否越界。如果在读取或写入数组元素时，采用越界的下标来操作，那么程序会抛出异常。

另外还有个好处是，我们可以在自定义的类上面实现几个关键方法，从而令其支持下标操作。如果想通过下标来访问自定义类中的元素，那么可以考虑实现下列方法中的一个或几个：

- -(id) objectAtIndexedSubscript: anIndex
- -(void) setObject: newValue atIndexedSubscript: anIndex
- -(id) objectForKeyedSubscript: aKey
- -(void) setObject: newValue forKeyedSubscript: aKey

你可以决定是像数组那样以数字作为下标按照顺序来访问，还是像字典那样以键作为下标按照关键字来访问，另外，还可以决定是否能够通过下标来修改元素的值。我们只需实现想要支持的方法就可以了，剩下的事情由编译器完成。

## A.5 功能测试

最后要说的是，开发者可以编写依赖于某种特性的代码。使用 Clang 的 `_has_feature` 来测试当前编译器是否支持某种字面量写法。这些功能测试可以判断出编译器是否支持数组字面量 (`objc_array_literals`)、字典字面量 (`objc_dictionary_literals`)、对象的取下标操作 (`objc_subscripting`)、数值字面量 (`objc_bool`) 及装箱表达式 (`objc_boxed_expressions`)：

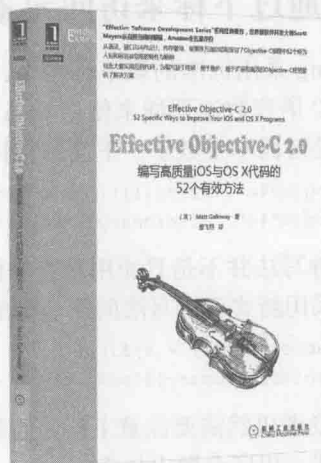
```
#if __has_feature(objc_array_literals)
    // ...
#else
    // ...
#endif
```

## 推荐阅读



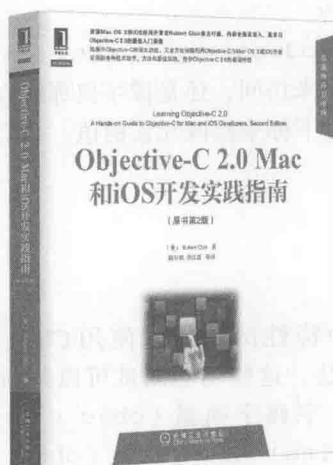
### iOS 应用软件设计之道

作者: William Van Hecke ISBN: 978-7-111-47833-1 定价: 69.00元



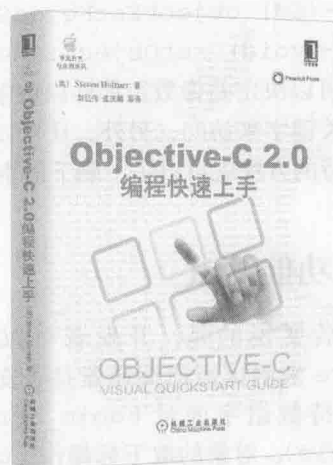
### Effective Objective-C 2.0: 编写高质量iOS与OS X代码的52个有效方法

作者: Matt Galloway ISBN: 978-7-111-45129-7 定价: 69.00元



### Objective-C 2.0 Mac和iOS开发实践指南 (原书第2版)

作者: Robert Clair ISBN: 978-7-111-48456-1 定价: 79.00元



### Objective-C 2.0编程快速上手

作者: Steven Holzner ISBN: 978-7-111-30874-4 定价: 35.00元

本书为iOS 7日常开发者提供了可靠而行之有效的解决方案。著名iOS编程专家Erica Sadun和顶级iOS开发者Rich Wardwell向大家展示了创建优秀iOS 7移动应用程序所需的全部技巧，告诉读者如何使用标准的API，以及如何充分利用iOS 7的图形、触摸与视图。

和Sadun所写的其他iOS畅销书一样，本书也用直观的代码来演示当前流行的编程技巧，并把关键概念浓缩为简洁的解决方案（recipe），使读者能够轻松掌握并将其运用到自己的项目中。本书并不是简单地把代码拼凑起来，而是会向你详细解释如何高效地开发iOS 7程序，并告诉你这样做的原因。

本书所有代码都经过全面修订与测试，而且利用了iOS 7的新功能及设备的新特点。

### 通过阅读本书，你将学到：

- 用多点触控、手势及自定义的手势识别器来创建先进的触摸式界面。
- 通过功能强大的新方式来构建并自定义控件。
- 创建与iOS 7的设计风格相符的新式界面。
- 实现iOS 7中新引入的动画效果。
- 通过弹出式窗口、进度条、本机通知、popover、提示音等手段来提醒用户。
- 用Xcode模块轻松地整合系统框架及头文件。
- 编排视图及其动画效果，规划视图的体系结构并理解视图之间协同运作的方式。
- 用iOS 7中强大的Auto Layout约束系统来适配各种尺寸的显示屏。
- 控制键盘，令屏幕上的控件具备文本输入功能，高效地扫描并调整文本格式。
- 用视图控制器来规划用户的工作区。
- 管理照片、视频、电子邮件及文本信息。
- 用iOS 7的先进特性来操作Flickr及Vimeo等社交媒体。
- 实现VoiceOver辅助功能，其中也包括iOS 7新引入的文本转语音功能。
- 初步了解如何使用Core Data来管理数据库。
- 使用iOS 7中强大的网络与Web服务功能。
- 用iOS 7新提供的API及灵活功能来增强程序的效果，其中包括提升可靠性及美化文本。
- 解决iOS 7中的一些问题和程序缺陷。

PEARSON

www.pearson.com

投稿热线：(010) 88379604

客服热线：(010) 88378991 88361066

购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com

网上购书：www.china-pub.com

数字阅读：www.hzmedia.com.cn



上架指导：计算机/移动开发

ISBN 978-7-111-49185-9



9 787111 491859 >

定价：119.00元