

Amazon
计算机
榜首图书

- 创造销售奇迹的最新经典著作
- 全面深入探索iPhone开发的无限可能
- 从这里，抢先拥抱软件开发的未来



Beginning iPhone Development
Exploring the iPhone SDK

iPhone开发基础教程

[美] Dave Mark 著
Jeff LaMarche
漆振 解巧云 孙文磊 等译



人民邮电出版社
POSTS & TELECOM PRESS

“Dave Mark一直是Mac编程图书作者中的佼佼者，而他现在又无可争议地成为了iPhone开发图书的王牌作者！本书是iPhone开发的权威指南，任何有意开始iPhone开发的人都应该阅读这本宝贵的参考指南。”

——Brian Greenstone (Pangea软件公司的总裁兼CEO)

“Trism游戏让我在2个月内收入25万美元，然后有无数人问我怎么开发iPhone应用，现在答案出现了！Dave和Jeff的书深入浅出、循序渐进而且示例丰富，堪称完美。它已经成了我的必备参考书，需要不时查阅。强烈推荐！”

——Steve Demeter (《连线》杂志“2008最佳iPhone应用” Trism游戏开发者)

Beginning iPhone Development Exploring the iPhone SDK

iPhone开发基础教程

Apple公司的iPhone已经开创了移动平台新纪元！它与App Store的绝配也为全世界的程序员提供了一个施展才华的全新大舞台。只要有新奇的创意，你完全有可能像开发iShoot游戏的Ethan Nicholas (日收入2万多美元)和开发Trism游戏的Steve Demeter (月收入超过10万美元)那样，仅凭单枪匹马就赢得全球市场，成功创业，改变自己的人生。

本书由业界名家撰写，英文原版问世以后迅速登上Amazon计算机图书排行榜榜首并持续至今，总排名一度达到20名左右，创造了销售奇迹。而且，本书获得了读者的一致好评，已经被奉为经典。书中从到Apple网站注册账号，下载和安装免费iPhone SDK开始，清晰透彻地讲述了创建iPhone应用程序的全过程。在探讨基本概念和各个关键特性 (iPhone界面元素、数据保存、SQLite、Quartz和OpenGL ES、手势支持、本地化、Core Location等) 时，提供了丰富的实例。更难得的是，本书始终强调iPhone开发中的各种最佳实践，即使是有经验的开发人员，也会因此受益匪浅。

开卷阅读本书，进入iPhone开发的神奇世界吧，它将让你热血沸腾！

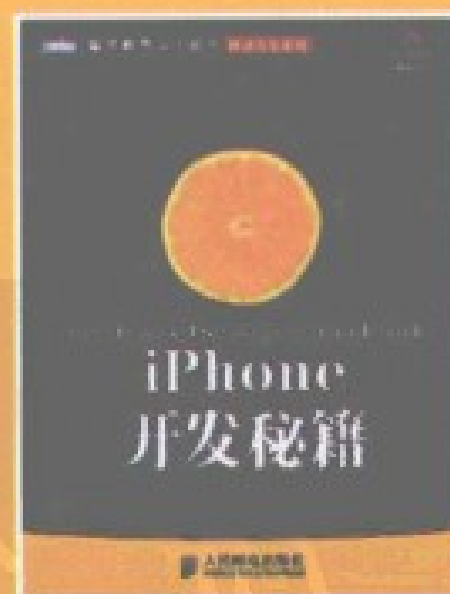
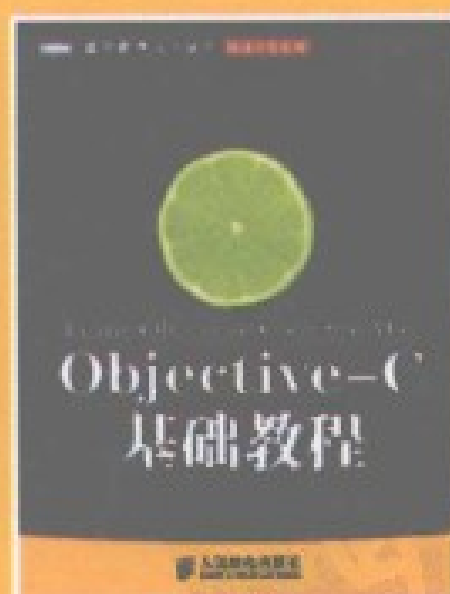


Dave Mark 深受爱戴的Apple技术开发专家，具有多年开发经验。他是许多Mac平台畅销书的作者，包括*Learn C on the Mac*、*Macintosh Programming Primer*系列以及*Ultimate Mac Programming*。可以通过www.davemark.com与他联系。



Jeff LaMarche 资深Apple平台专家，拥有多年企业级开发经验。他是*MacTech Magazine*和Apple公司开发人员网的专栏作家。

延伸阅读



Apress®

本书相关信息请访问：图灵网站 <http://www.turingbook.com>

读者/作者热线：(010)88593802

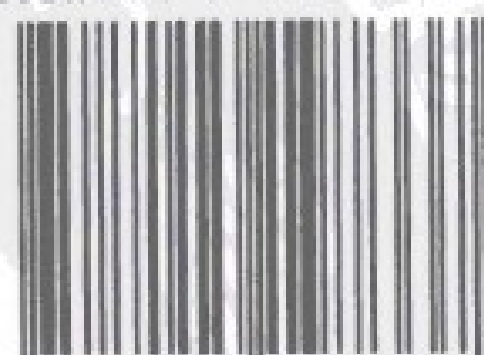
反馈/投稿/推荐信箱：contact@turingbook.com

分类建议 计算机/程序设计/移动开发

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-19733-7



9 787115 197337 >

ISBN 978-7-115-19733-7/TP

定价：65.00 元

版权声明

本书籍由pdf_world制作发布，针对目前图书数量繁多，质量参差不齐，读者面对买书易，买好书难的问题。pdf_world旨在为读者购书前提供另外一种快捷了解图书内容，及挑选好书的方法。因此，pdf_world每周都会更新一批当年度最新的畅销书供读者选择。如有需要，请关注我们的官方博客：

http://hi.baidu.com/pdf_world

声明：

- 1.本书仅供读者购买纸本书前预览内容使用
- 2.为尊重原作者的劳动成果，请于下载后48小时内删除。
- 3.如果您确实需要此书，请通过正规渠道购买纸本书。

图书在版编目 (CIP) 数据

iPhone 开发基础教程 / (美) 马克 (Mark, D.), (美) 拉马赫 (LaMarche, J.) 著; 漆振等译. —北京: 人民邮电出版社, 2009.4 (2009.4 重印)

(图灵程序设计丛书)

书名原文: Beginning iPhone Development: Exploring the iPhone SDK

ISBN 978-7-115-19733-7

I. i… II. ①马…②拉…③漆… III. 移动通信—携带电话机—应用程序—程序设计—教材 IV. TN929.53

中国版本图书馆CIP数据核字 (2009) 第014679号

内 容 提 要

iPhone 是一种全新的移动平台, 苹果公司为它推出了强大的软件开发工具包 iPhone SDK。本书是一部关于 iPhone SDK 和 iPhone 开发的基础教程, 内容翔实、语言生动。书中结合消费类设备上常见的实例, 循序渐进地讲解了 iPhone 开发的基本流程, 并介绍了最先进、时尚、受欢迎的 iPhone 特性。

本书内容完整丰富, 具有较强的通用性, 编程领域中各层次读者都能通过本书快速学习 iPhone 开发, 提高相关技能。

图灵程序设计丛书

iPhone开发基础教程

◆ 著 [美] Dave Mark Jeff LaMarche

译 漆振 解巧云 孙文磊等

责任编辑 傅志红

执行编辑 谢灵芝

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号

邮编 100061 电子函件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

北京隆昌伟业印刷有限公司印刷

◆ 开本: 800×1000 1/16

印张: 25.5

字数: 603千字

2009年4月第1版

印数: 3 001—5 000册

2009年4月北京第2次印刷

著作权合同登记号 图字: 01-2009-0532号

ISBN 978-7-115-19733-7/TP

定价: 65.00元

读者服务热线: (010)88593802 印装质量热线: (010)67129223

反盗版热线: (010)67171154

版 权 声 明

Original English language edition, entitled *Beginning iPhone Development: Exploring the iPhone SDK* by Dave Mark, Jeff LaMarche, published by Apress, 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705.

Copyright © 2009 by Dave Mark, Jeff LaMarche. Simplified Chinese-language edition copyright © 2009 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由Apress L. P.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。



送给Deneen J. Melander，你是我生命中的指明灯。

——Dave

送给我生命中最重要的人，我的妻子和孩子。

——Jeff



对本书的赞誉

“不断有人问我如何开始iPhone开发，但我始终没有找到很好的答案。现在有答案了，Dave和Jeff的这本书简明易懂，并通过许多插图确保你能理解基本概念。在此基础上，他们深入讲述了各种关键概念，如MVC模式和ImageBuilder基本知识。此外，我自己经常将它作为参考指南使用——丰富的代码示例使它成为案头必备。”

——Steve Demeter, Trism的创建者和Demiforce公司的所有者

“本书清晰地描述了从注册为iPhone开发人员到创建完整的应用程序的整个开发流程。它通过大量示例演示了iPhone的特性。作者在本书中出色地展示了‘最佳实践’编码方法。这是一本很难被超越的iPhone软件创建指南。”

——Aaron Basil, iDev2.com

“Dave Mark始终是Mac编程图书作者中的佼佼者，而他现在又无可争议地成为了iPhone开发图书的王牌作者！本书是iPhone开发的权威指南，任何有意开始iPhone开发的人都应该阅读这本宝贵的参考指南。”

——Brian Greenstone, Pangea软件公司的总裁兼CEO

“Jeff和Dave出色地完成了探索iPhone SDK的任务。本书是当之无愧的iPhone SDK开发最佳资源。开发人员将深深地被本书吸引，本书的实用性将在他们创建新的iPhone应用程序时体现出来。如果你是一名对此新兴平台感兴趣的开发人员，那这本书是必不可少的。”

——Chris Stewart, iPhoneDevSDK.com创始人

“如果你打算编写iPhone程序，请从这里开始。Dave和Jeff知道你需要什么，也知道如何解释你所学的知识。我对本书覆盖的内容之广感到非常惊讶，从Hello World到分析用户的手势。本书不仅涵盖了操作摄像机等有趣的内容，还包括本地化等现实世界中的开发问题。他们让我受益匪浅。”

——Mark Dalrymple, CocoaHeads创始人之一，*Advanced Mac OS X Programming*的主要作者

“从技术概述入手，到如何处理设备，作者带领我们直接进入了iPhone开发的核心领地。随着深入学习，你会了解关于各种布局引擎、视图管理器以及加速计和GPS API的更多知识。这本书是希望迅速且有效地开始iPhone开发的人士的必备之书！”

——Chris Pelsor, Tarantell:Hybrid公司经理

译者序

iPhone自从发布的那一天起就成为科技界的焦点。但一直为人诟病的是，其本身并不支持开源软件开发，也一直没有推出相应的第三方开发工具。而iPhone SDK的发布无疑解决了这一难题。开发人员可以使用iPhone SDK轻易地为iPhone和iPod touch创建应用程序。其简单易学的操作方式和强大的功能为开发人员带来了超强的能力。

本书提供了关于iPhone SDK和iPhone开发的全面信息，对Objective-C编程语言、Xcode和Interface Builder开发工具进行了深入浅出的介绍，同时对iPhone开发的基本流程、原理和原则进行了详细和通俗的讲解。本书采用理论与实践相结合的方式，指导读者创建一系列应用程序，让读者能在实践中理解iPhone应用程序的运行方式和构建方式，掌握具体的iPhone特性，学会如何控制这些特性或与之交互。

全书共18章，分为3个部分。前4章介绍iPhone开发中的相关基本概念和开发人员所需的必备知识，并通过示例演示了一些标准的iPhone用户界面控件。第5章至第16章深入介绍如何开发各种高级iPhone特性，其中包括自动旋转、工具栏控制器、表视图、分层列表、应用程序设置、数据管理、绘图、手势输入、Core Location、加速计以及照相机和相片库。最后两章介绍如何将iPhone应用程序翻译为其他语言，从而让更多的用户接受并使用它，以及看完本书之后应该努力的方向。

本书覆盖面广、结构清晰，是一本有关iPhone开发的全新入门指南。它面向具备基本Objective-C知识的iPhone初、高级开发用户，不论你是经验丰富的开发人员，还是初涉编程领域的新手，都可以从本书中得到有用的信息。书中的示例通用性高，特别适合读者参考使用，这使本书成为广大读者的首选。

iPhone SDK是一个新兴的软件开发平台，但目前国内关于iPhone开发的资料非常有限。相信本书的出版可以为iPhone开发的发展起到推波助澜的作用。

本书由漆振、解巧云、孙文磊等翻译，在翻译过程中得到了欧阳宇、盛海艳、杨越和张波的帮助，在此一并致谢。由于译者的知识水平有限，加之时间也比较仓促，文中难免会出现一些疏漏，恳请广大读者给予批评指正。

译者

2009年1月

前言

“从我开始使用Mac以来，我还没有看到过让我如此激动的编程平台。”最近我们经常听到这样的感言，坦白地说，我也有同感。iPhone是一种让人激动不已的出色技术，它将功能和乐趣完美地融合在一起。而程序员使用这种技术可以完成的工作也让人很激动！

这个世界的大门刚刚打开。花些时间浏览App Store，你会情不自禁地感动振奋。如果你并不负责设计自己的iPhone应用程序，那么为iPhone开发提供咨询也具有无限的商机。每个人好像都想把他们的产品导入该平台。我们的电话已经响个不停了。

如果你已经研究了几个月，偶尔访问一下我们的网站 (<http://iphonedevbook.com>)，并和我们打个招呼吧。请告诉我们有关你的项目的信息，我们很乐意倾听你的诉说。

Dave和Jeff



致 谢

没有我们善良、能干又聪明的家人、朋友和同伴的支持，本书是不可能完成的。首先，感谢Terry和Deneen对我们的宽容，他为我们专心写书提供了非常好的环境。这个项目耗费了相当长的时间，但是你们从未抱怨过。我们很幸运！

本书的完成还要归功于Apress工作人员的合作。他们不仅是本书的出版者，还是我们不可多得的朋友。Clay Andres策划了本书，并将我们带到了Apress。Dominic Shakeshaft始终带着微笑处理我们的抱怨，并总能找到合适的解决方法，使本书完成得更好。Laura Esterman是一位亲切的项目经理，她是我们实现每一个目标的动力。她让本书的编写能够有条不紊地进行，并为我们指明了正确的方向。Heather Lang是个极为出色的文字编辑，很荣幸能和你一起工作，请下一次依然做我们的文字编辑！Grace Wong和生产团队把零碎的稿件整合成书，Kari Brooks-Copony提供了出色的版式设计。Kelly Winkvist用我们的Word文档印刷出非常精美的页面。Pete Aylward征集宣传素材，策划推出一系列营销活动。我们对Apress的所有工作人员表示由衷的感谢！

特别感谢我们优秀的技术审稿人Mark Dalrymple。除了提供具有独到见解的反馈之外，Mark还测试了本书中的所有代码，帮助我们保证了本书的正确性。感谢您！

最后，感谢我们的孩子，他们在父亲努力工作时表现出非常好的耐心。本书是送给你们的：Maddie、Gwynnie、Ian、Kai、Daniel、Kelley和Ryan。



目 录

第 1 章 欢迎来到 iPhone 的世界.....	1
1.1 关于本书	1
1.2 必要条件	1
1.3 必备知识	3
1.4 编写 iPhone 应用程序有何不同.....	4
1.4.1 只有一个正在运行的应用程序	4
1.4.2 只有一个窗口	4
1.4.3 受限访问	4
1.4.4 有限的响应时间	4
1.4.5 有限的屏幕大小	5
1.4.6 有限的系统资源	5
1.4.7 缺少 Cocoa 工具	5
1.4.8 新属性	5
1.4.9 与众不同的方法	6
1.5 本书内容	6
1.6 准备开始吧	7
第 2 章 创建基本项目.....	8
2.1 在 Xcode 中设置项目	8
2.2 Interface Builder 简介	12
2.2.1 nib 文件的构成	14
2.2.2 在视图中添加标签	15
2.3 iPhone 美化	17
2.4 小结	20
第 3 章 处理基本交互.....	21
3.1 模型-视图-控制器范型	21
3.2 创建项目	22
3.3 创建视图控制器	22
3.3.1 输出口	23
3.3.2 操作	23
3.3.3 将操作和输出口添加到视图	
控制器	24
3.3.4 将操作和输出口添加到实现	
文件	26
3.4 使用应用程序委托	30
3.5 编辑 MainWindow.xib	32
3.6 编辑 Button_FunViewController.xib.....	33
3.6.1 在 Interface Builder 中创建视图	33
3.6.2 连接所有元素	35
3.6.3 测试	37
3.7 小结	38
第 4 章 更丰富的用户界面.....	39
4.1 满是控件的屏幕	39
4.2 活动、静态和被动控件	41
4.3 创建应用程序	41
4.3.1 导入图像	41
4.3.2 实现图像视图和文本字段	42
4.3.3 添加图像视图	43
4.3.4 添加文本字段	46
4.3.5 设置第二个文本字段的属性	49
4.3.6 连接输出口	49
4.4 构建和运行	49
4.4.1 完成输入后关闭键盘	50
4.4.2 通过触摸背景关闭键盘	51
4.5 实现滑块和标签	52
4.5.1 确定输出口	52
4.5.2 确定操作	52
4.5.3 添加输出口和操作	52
4.5.4 添加滑块和标签	53
4.5.5 连接操作和输出口	54
4.6 实现开关和分段控件	55
4.6.1 确定输出口	55

4.6.2 确定操作	55	6.3.2 修改应用程序委托	92
4.6.3 添加开关和分段控件	57	6.3.3 SwitchViewController.h	93
4.6.4 连接输出口	58	6.3.4 修改 MainWindow.xib	93
4.7 实现按钮、操作表和警报	59	6.3.5 编写 SwitchViewController.m	96
4.7.1 将输出口及操作添加到控制器 头文件	59	6.3.6 实现内容视图	99
4.7.2 在 Interface Builder 中添加按钮	60	6.4 制作转换动画	101
4.7.3 实现按钮的操作方法	60	6.5 重构	103
4.8 显示操作表	61	6.6 小结	105
4.9 美化按钮	63	第 7 章 标签栏与选取器	106
4.9.1 viewDidLoad 方法	64	7.1 Pickers 应用程序	106
4.9.2 控件状态	65	7.2 委托和数据源	108
4.9.3 可拉伸图像	65	7.3 建立工具栏框架	108
4.10 小结	65	7.3.1 创建文件	108
第 5 章 自动旋转和自动调整大小	67	7.3.2 设置内容视图 nib	109
5.1 使用自动调整属性处理旋转	68	7.3.3 添加根视图控制器	109
5.1.1 指定旋转支持	68	7.4 实现日期选取器	113
5.1.2 使用自动调整属性设计界面	70	7.5 实现单个组件选取器	116
5.1.3 自动调整属性	70	7.5.1 声明输出口和操作	116
5.1.4 设置按钮的自动调整属性	72	7.5.2 构建视图	116
5.2 在旋转时重构视图	73	7.5.3 将控制器实现为数据源和委托	117
5.2.1 声明和连接输出口	74	7.6 实现多组件选取器	121
5.2.2 在旋转时移动按钮	74	7.6.1 声明输出口和操作	121
5.3 切换视图	77	7.6.2 构建视图	122
5.3.1 确定输出口	78	7.6.3 实现控制器	122
5.3.2 确定动作	78	7.7 实现独立组件	125
5.3.3 声明动作和输出口	79	7.8 使用自定义选取器创建简单游戏	132
5.3.4 设计两个视图	79	7.8.1 编写控制器头文件	132
5.3.5 实现交换和动作	80	7.8.2 构建视图	133
5.3.6 链接 Core Graphics 框架	83	7.8.3 添加图像资源	133
5.4 小结	85	7.8.4 实现控制器	133
第 6 章 多视图应用程序	86	7.8.5 spin 方法	136
6.1 View Switcher 应用程序	88	7.8.6 viewDidLoad 方法	137
6.2 多视图应用程序的体系结构	88	7.8.7 最后的细节	139
6.2.1 多视图控制器也是视图控制器	89	7.8.8 链接 Audio Toolbox 框架	142
6.2.2 内容视图剖析	89	7.9 小结	143
6.3 构建 View Switcher	89	第 8 章 表视图简介	144
6.3.1 创建视图控制器和 nib 文件	90	8.1 表视图基础	144
		8.2 实现一个简单的表	147

8.2.1 设计视图	147	9.8 第 5 个子控制器: 可删除的行	213
8.2.2 编写控制器	148	9.9 第 6 个子控制器: 可编辑的详细窗格	218
8.3 添加一个图像	151	9.9.1 创建数据模型对象	219
8.4 附加配置	151	9.9.2 创建控制器	221
8.4.1 设置缩进级别	152	9.9.3 创建详细视图控制器	224
8.4.2 处理行的选择	152	9.10 更多内容	238
8.4.3 更改字体大小和行高	153	9.11 小结	240
8.4.4 委托还能做什么?	155	第 10 章 应用程序设置和用户默认设置	241
8.5 定制表视图单元	155	10.1 了解设置束	241
8.5.1 单元应用程序	155	10.2 AppSettings 应用程序	242
8.5.2 向表视图单元添加子视图	155	10.3 创建项目	243
8.5.3 使用 UITableViewCell 的自定义 子类	159	10.4 使用设置束	245
8.6 分组分区和索引分区	163	10.4.1 在项目中添加设置束	245
8.6.1 构建视图	163	10.4.2 设置属性列表	246
8.6.2 导入数据	163	10.4.3 添加文本字段设置	247
8.6.3 实现控制器	164	10.4.4 添加安全文本字段设置	249
8.6.4 添加索引	167	10.4.5 添加多值字段	249
8.7 实现搜索栏	168	10.4.6 添加拨动开关设置	250
8.7.1 重新考虑设计	168	10.4.7 添加滑块设置	251
8.7.2 深层可变副本	168	10.4.8 添加子设置视图	252
8.7.3 更新控制器头文件	170	10.5 读取应用程序中的设置	253
8.7.4 修改视图	171	10.6 更改应用程序中的默认设置	257
8.7.5 修改控制器实现	172	10.7 小结	259
8.8 小结	180	第 11 章 基本数据持久性	260
第 9 章 导航控制器和表视图	181	11.1 应用程序的沙盒	260
9.1 导航控制器	181	11.1.1 获取 Documents 目录	261
9.1.1 栈的性质	181	11.1.2 获取 tmp 目录	262
9.1.2 控制器栈	182	11.2 文件保存策略	262
9.2 由 6 个部分组成的分层应用程序: Nav	182	11.2.1 单个文件持久性	262
9.3 构建 Nav 应用程序的骨架	184	11.2.2 多个文件持久性	262
9.3.1 创建根视图控制器	185	11.3 持久保存应用程序数据	263
9.3.2 设置导航控制器	185	11.4 持久性应用程序	264
9.4 第 1 个子控制器: 展示按钮视图	191	11.4.1 创建持久性项目	264
9.5 第 2 个子控制器: 校验表	198	11.4.2 设计持久性应用程序视图	265
9.6 第 3 个子控制器: 表行上的控件	202	11.4.3 编辑持久性类	265
9.7 第 4 个子控制器: 可移动的行	207	11.4.4 对模型对象进行归档	269
9.7.1 编辑模式	208	11.4.5 实现 NSCopying	270
9.7.2 创建一个新的二级控制器	208	11.5 归档应用程序	272

11.5.1 实现 FourLines 类	272	13.8 检测捏合操作	333
11.5.2 实现 PersistenceViewController 类	273	13.9 自己定义手势	336
11.6 使用 iPhone 的嵌入式 SQLite3	276	13.10 小结	339
11.7 小结	284	第 14 章 我在哪里? 使用 Core Location 定位功能	340
第 12 章 使用 Quartz 和 OpenGL 绘图	285	14.1 位置管理器	340
12.1 图形世界的两个视图	285	14.1.1 设置所需的精度	341
12.2 本章的绘图应用程序	286	14.1.2 设置距离筛选器	341
12.3 Quartz 绘图方法	286	14.1.3 启动位置管理器	341
12.3.1 Quartz 2D 的图形上下文	286	14.1.4 更明智地使用位置管理器	341
12.3.2 坐标系	287	14.2 位置管理器委托	342
12.3.3 指定颜色	287	14.2.1 获取位置更新	342
12.3.4 在上下文中绘制图像	289	14.2.2 使用 CLLocation 获取纬度和 经度	342
12.3.5 绘制形状: 多边形、直线 和曲线	289	14.2.3 错误通知	343
12.3.6 Quartz 2D 工具示例: 模式、 梯度、虚线模式	289	14.3 尝试使用 Core Location	344
12.4 构建 QuartzFun 应用程序	290	14.3.1 更新位置管理器	347
12.4.1 创建随机颜色	291	14.3.2 确定移动距离	348
12.4.2 定义应用程序常量	291	14.4 小结	349
12.4.3 实现 QuartzFunView 框架	292	第 15 章 加速计	350
12.4.4 向视图控制器中添加输出口 和操作	294	15.1 加速计物理学	350
12.4.5 更新 QuartzFunViewController .xib	297	15.2 访问加速计	351
12.4.6 绘制直线	298	15.2.1 UIAcceleration	351
12.4.7 绘制矩形和椭圆形	299	15.2.2 实现 accelerometer: didAccelerate:方法	353
12.4.8 绘制图像	301	15.3 摇动与击碎	354
12.5 一些 OpenGL ES 基础知识	306	15.3.1 用于击碎的代码	355
12.6 小结	316	15.3.2 加载模拟文件	358
第 13 章 轻击、触摸和手势	317	15.3.3 完好如初——复原触摸	359
13.1 多触摸术语	317	15.4 滚弹珠程序	359
13.2 响应者链	318	15.4.1 实现 Ball View 控制器	360
13.3 多触摸体系结构	319	15.4.2 编写 Ball View	361
13.4 触摸浏览器应用程序	320	15.4.3 计算小球运动	364
13.5 Swipe 应用程序	324	15.5 小结	366
13.6 实现多个轻扫	327	第 16 章 iPhone 照相机和照片库	367
13.7 检测多次轻击	329	16.1 使用图像选取器和 UIImagePickerController	367
		16.2 实现图像选取器控制器委托	368

16.3 实际测试照相机和库.....	370	17.3.6 本地化应用程序图标.....	386
16.3.1 设计界面.....	370	17.3.7 生成和本地化字符串文件.....	386
16.3.2 实现照相机视图控制器.....	371	17.4 小结.....	388
16.4 小结.....	374	第 18 章 未来之路.....	390
第 17 章 应用程序本地化.....	375	18.1 答案揭晓.....	390
17.1 本地化体系结构.....	375	18.1.1 苹果公司的文档.....	390
17.2 使用字符串文件.....	376	18.1.2 邮件列表.....	391
17.3 现实中的 iPhone: 本地化应用程序.....	378	18.1.3 论坛.....	391
17.3.1 查看当前区域设置.....	381	18.1.4 网站.....	391
17.3.2 测试 LocalizeMe.....	381	18.1.5 博客.....	391
17.3.3 本地化 nib 文件.....	382	18.1.6 如果仍未解决问题.....	392
17.3.4 查看本地化的项目结构.....	383	18.2 再会.....	392
17.3.5 本地化图像.....	385		



第 1 章

欢迎来到 iPhone 的世界

你想编写iPhone应用程序？iPhone可能在今后很长一段时期内都是最有趣的新兴平台。毫无疑问，它是迄今为止最新颖的移动平台，特别是现在，苹果公司还提供了一组精美的、具有良好文档的工具来支持iPhone应用程序的开发。

1.1 关于本书

本书将带你走上创建iPhone应用程序的大道。我们的目标是让你通过初步学习，理解iPhone应用程序的运行方式和构建方式。在阅读过程中，你将创建一系列小型应用程序，每个应用程序都会突出特定的iPhone特性，展示如何控制这些特性或与其交互。如果将本书中的基本知识与你自己的创造力相结合，同时借助苹果公司大量翔实的文档，你将具备创建专业级iPhone应用程序所需的一切条件。

1.2 必要条件

在开始编写iPhone软件之前，需要做一些准备工作。对于初学者，需要一台运行Leopard（OS X 10.5.3或更高版本）的基于Intel的Macintosh计算机。2006年之后上市的任何Macintosh计算机（不管是笔记本还是台式机）应该都符合要求。

无需使用具备顶级配置的计算机，MacBook或Mac Mini就能够出色地完成任务。但是，对于较早且运行速度较慢的计算机型号，进行RAM升级能够获得较大的性能提升。

你还需要注册成为iPhone开发人员。只有完成了这一步，苹果公司才允许下载iPhone SDK（软件开发工具包）。

要进行注册，请访问<http://developer.apple.com/iphone/>（中文网站为<http://www.apple.com.cn/developer/iphone/>，如图1-1所示），该页面应该与图1-1中显示的页面类似。页面中提供了最新且功能最强大的iPhone SDK的下载链接。单击该链接将进入包含3个选项的注册页面。

最简单（而且免费）的选项是单击Download the Free SDK按钮。页面将提示输入Apple ID。使用你的Apple ID登录。如果还没有Apple ID，请单击Create Apple ID按钮，创建一个Apple ID，然后再登录。登录之后，将进入iPhone开发主页面。其中不仅有SDK的下载链接，还提供了各类文档、视频和示例代码等的链接，所有这些资源都能帮你进行iPhone应用程序开发。



图1-1 苹果公司的iPhone开发中心网站

iPhone SDK中包含的一个最重要的元素是Xcode，它是苹果公司的IDE（集成开发环境）。Xcode提供了各种实用工具，用于创建和调试源代码，编译应用程序以及调优应用程序性能。学习完本书，你将会迷恋上Xcode！

这个免费的SDK还包含一个仿真器，它支持在Mac上运行大多数iPhone程序。这对于学习如何编写iPhone程序极其有用。但是，免费选项不支持将应用程序下载到实际的iPhone（或iPod Touch）中。此外，它也不支持在苹果公司的iPhone App Store上分发应用程序。要实现这些功能，需要使用另外两个下载选项，它们不是免费的。

说明 仿真器不支持依赖于硬件的特性，比如iPhone的加速计或摄像功能。要支持这些特性，需要使用其他选项。

标准版程序的价格为99美元。它提供了全面的开发工具、资源和技术支持，支持通过苹果公司的App Store分发应用程序，并且最重要的是，支持在iPhone上（而不只是在仿真器上）测试和调试代码。

企业版程序的价格为299美元，可供企业开发专用的、内部的iPhone和iPod Touch应用程序。

有关这两种程序的详细信息，请访问<http://developer.apple.com/iphone/program/>。

由于iPhone是一种始终连网的移动设备，并且使用的是其他公司的无线基础设施，因此苹果公司对iPhone开发人员的限制比对Mac开发人员多得多，Mac开发人员无需经过苹果公司的审查或批准就能够编写和分发程序。

苹果公司添加这些限制，更多的是为了尽量避免分发恶意或效率低下的程序，因为这类程序可能降低共享网络的性能。开发iPhone应用程序似乎麻烦不少，但苹果公司在简化开发过程方面付出了巨大努力。还应该提及的是，99美元的价格比微软公司的软件开发IDE——Visual Studio的价格低得多。

另外，很明显，你还需要一部iPhone。虽然大部分代码都可以通过iPhone仿真器进行测试，但并非所有程序都是如此。一些应用程序需要在实际的iPhone上进行全面测试，然后才能分发给公众。

说明 如果要注册标准版或企业版程序，你应该立即注册。批准过程可能需要一些时间，并且通过批准之后才能在iPhone或iPod Touch上运行应用程序。但是不必担心，前几章中的所有项目以及本书中的大多数应用程序，都可以在iPhone仿真器上运行。

1.3 必备知识

学习本书应该具备一定的编程知识。你应该理解面向对象编程的基础知识，例如，了解对象、循环和变量的含义，还应该熟悉Objective-C编程语言。SDK中的Cocoa Touch是本书使用的主要工具，它使用的是Objective-C 2.0编程语言，但是如果不了解Objective-C语言的新增特性也没有关系。我们将重点介绍要使用的2.0语言特性，并解释其工作原理和使用它的原因。

你还应该熟悉iPhone本身。就像在任何其他平台中编写应用程序一样，你需要熟悉iPhone的各种特性，并了解iPhone界面以及iPhone程序的外观。

还不熟悉Objective-C?

如果你从未使用Objective-C编写过程序，那么以下资源有助于你了解该语言。

首先，阅读由Mac编程专家Mark Dalrymple和Scott Knaster撰写的*Learn Objective-C on the Mac*^①一书，该书浅显易懂，是学习Objective-C基础知识的优秀图书：

<http://www.apress.com/book/view/9781430218159>

^① 中文版即将由人民邮电出版社出版。——编者注

接下来,访问Apple iPhone开发中心网站并下载*The Objective-C 2.0 Programming Language*一书的电子版,该书深入浅出地介绍了Objective-C 2.0的方方面面,是一本优秀的参考指南:

<http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/ObjectiveC>

注意,需要登录才能访问此文档。

1.4 编写 iPhone 应用程序有何不同

如果从未使用过Cocoa或它的前期产品NextSTEP,那么你可能会发现Cocoa Touch(用于编写iPhone应用程序的应用程序框架)稍显另类。它与其他常用应用程序框架(如用于构建.NET或Java应用程序的框架)之间存在一些基本差异。你起初可能会有点不知所措,但不必担心,只要勤加练习,就可以掌握其中的规律。

如果你具备使用Cocoa或NextSTEP编程的经验,则会发现iPhone SDK中有许多熟悉的身影。其中的许多类都是从用于Mac OS X开发的程序版本中原样借鉴过来的,一些类即便存在不同,它们也遵循相同的基本原则,并使用类似的设计模式。但是,Cocoa和Cocoa Touch之间却存在一些差异。

无论你的知识背景如何,都需要谨记iPhone开发与桌面应用程序开发之间的重要差异。

1.4.1 只有一个正在运行的应用程序

除了操作系统之外,任何时候iPhone上都只能运行一个应用程序。随着iPhone内存的增大、处理器的增强,这一点在未来可能会发生变化。但是在目前,在执行代码时,你的应用程序将是唯一正在运行的程序。若你的应用程序不是用户正在交互中的,那么它不会起作用。

1.4.2 只有一个窗口

在桌面及笔记本操作系统中,多个程序可以同时运行,并且可以分别创建和控制多个窗口。而iPhone则有所不同,它只允许应用程序操作一个“窗口”。应用程序与用户的所有交互都在这个窗口中完成,而且这个窗口大小就是iPhone屏幕的大小,是固定的。

1.4.3 受限访问

计算机上的程序可以访问启动它们的用户能够访问的任何内容,而iPhone则严格限制了应用程序的权限。你只能在iPhone为应用程序创建的文件系统中读写文件。此区域称为应用程序的沙盒,应用程序在其中存储文档、首选项等需要存储的数据。

应用程序还存在其他方面的限制。举例来说,你不能访问iPhone上端口号较小的网络端口,或者执行台式计算机中需要根用户或管理员权限才能执行的操作。

1.4.4 有限的响应时间

由于其使用方式特殊,iPhone及其应用程序需要具备较快的响应时间。启动应用程序之后,

需要打开应用程序，载入首选项和数据，并尽快在屏幕上显示主视图，这一切要在几秒之内发生。只要应用程序在运行，就可以从其下方拖出一个菜单条。如果用户按主页（home）按钮，iPhone 就会返回主页，并且用户需要快速保存一切内容并退出。如果未在5秒之内保存并放弃控制，则应用程序进程将被终止，无论用户是否已经完成保存。

因此，你在设计iPhone应用程序时需要注意这一点，以确保用户退出时不会丢失数据。

1.4.5 有限的屏幕大小

iPhone的屏幕显示效果非常出色，从它推出直到现在，它一直是消费者设备上分辨率最高的屏幕。但是，iPhone的显示屏幕并不大，你施展的空间要比现代计算机小很多，仅有480×320像素。而在撰写本书时，苹果公司最便宜的iMac支持1680×1050像素，最便宜的笔记本电脑MacBook支持1280×800像素。而苹果公司最大的显示器，30英寸的Cinema Display，支持2560×1600像素。

1.4.6 有限的系统资源

阅读本书的任何资深程序员可能都会对128 MB内存、4 GB存储空间的机器嗤之以鼻，因为其资源实在是非常有限，但这种机器却是真实存在的。或许，开发iPhone应用程序与在内存为48 KB的机器上编写复杂的电子表格应用程序不属于同一级别，二者之间没有可比性，但由于iPhone的图形属性和它的功能，所以其内存不足是很容易出现的。目前上市的iPhone具备128 MB物理内存，当然这还会随时间不断增长。内存的一部分用于屏幕缓冲和其他一些系统进程。通常，大约一半内存将留给应用程序使用。

虽然64 MB对于这样的小型计算机可能已经足够了，但谈到iPhone的内存时还有另一个因素需要考虑：现代计算机操作系统，如Mac OS X，会将一部分未使用的内存块写到磁盘的交换文件中。这样，当应用程序请求的内存超过计算机实际可用的内存时，它仍然可以运行。但是，iPhone OS并不会将易失性内存（如应用程序数据）写到交换文件中。因此，应用程序可用的内存量将受到电话中未使用物理内存量的限制。

Cocoa Touch提供了一种内置机制，可以将内存不足的情况通知给应用程序。出现这种情况时，应用程序必须释放不需要的内存，甚至可能被强制退出。

1.4.7 缺少 Cocoa 工具

如果你在接触iPhone之前有过Cocoa方面的经验，那么你过去习惯使用的一些工具在iPhone中已经不可用了。iPhone SDK不支持Core Data或Cocoa Binding。我们之前已经说过，Cocoa Touch使用的是Objective-C 2.0，但该语言中的一个关键特性在iPhone中并不可用：Cocoa Touch不支持垃圾收集。

1.4.8 新属性

前面已经说过，Cocoa Touch缺少Cocoa的一些功能，但iPhone SDK中也有一些新功能是Cocoa所没有的，或者至少不是在任何Mac上都可用的。iPhone SDK为应用程序提供了一种定位方法，

即使用Core Location确定电话的当前地理坐标。iPhone还提供了一个内置的摄像和照片库，并且SDK允许应用程序访问这两者。iPhone还提供了一个内置的加速计，用于检测iPhone的持有和移动方式。

1.4.9 与众不同的方法

iPhone没有键盘和鼠标，这意味着它与用户的交互方式与通用的计算机截然不同。所幸的是，大多数交互都不需要你来处理。举例来说，如果在应用程序中添加一个文本字段，则iPhone知道在用户单击该字段时调用键盘，而不需要编写任何额外的代码。

1.5 本书内容

下面是本书其余章节的简要概述。

第2章：讲述如何使用Xcode和Interface Builder创建一个简单的界面，并在iPhone屏幕上添加一些文本。

第3章：开始实现与用户的交互，构建一个简单的应用程序，用于在运行时根据用户按下的按钮动态更新显示的文本。

第4章：以第3章为基础，介绍其他一些iPhone标准用户界面控件。我们还将介绍如何使用警告框和表提醒用户做出决策，或者通知用户发生了一些异常事件。

第5章：了解自动旋转机制，该机制允许在纵向或横向模式下使用iPhone应用程序。

第6章：介绍更多高级用户界面，并阐述如何创建多视图界面。我们将更改在运行时为用户显示的视图，以创建更加复杂的用户界面。

第7章：介绍如何实现工具栏控制器，它是一个标准的iPhone用户界面。

第8章：介绍表视图。表视图是向用户提供数据列表的主要方法，并且是基于分层导航的应用程序的基础。

第9章：介绍如何实现分层列表，它是最常用的iPhone应用程序界面之一，你可以通过它查看更多或更详细的数据。

第10章：介绍如何实现应用程序设置，iPhone中的这种机制允许用户设置他们的应用程序级首选项。

第11章：介绍iPhone上的数据管理。我们将讨论如何创建用于保存应用程序数据的对象，以及如何将这些数据持久存储到iPhone的文件系统和嵌入式数据库SQLite中。

第12章：绘图是人们的普遍爱好，这一章介绍如何实现一些自定义绘图，这需要使用Quartz和OpenGL ES中的基本绘图函数。

第13章：iPhone的多点触摸屏幕可以接受用户的各种手势输入。这一章讲述如何检测基本的手势，如双指捏合和单指滑动，还将介绍定义新手势的过程，并讨论新手势的适用情况。

第14章：iPhone可以通过Core Location确定其纬度和经度。这一章将编写利用Core Location计算iPhone的物理位置的代码，并在各种应用中使用该信息。

第15章：介绍如何与iPhone加速计交互，iPhone通过加速计确定其持有方式。我们将讨论应用程序如何通过该信息完成一些有趣的任务。

第16章：每个iPhone都有自己的摄像设备和图片库，这两者都可供应用程序使用。这一章介绍如何使用它们。

第17章：iPhone现已遍及70多个国家，这一章介绍以何种方式编写应用程序能方便地把所有部分翻译为其他语言，从而发掘应用程序的潜在用户。

第18章：至此，你已经掌握了iPhone应用程序的基本构建方法。但接下来再应向何处去呢？这一章将探索掌握iPhone SDK的后续步骤。

1.6 准备开始吧

iPhone是一款全新的、令人难以置信的计算平台，是轻松开发的利器。编写iPhone应用程序将成为一种全新的体验，这种体验与之前你使用过的任何平台都不同。所有看似熟悉的功能都具有其独特的一面，但随着深入体会本书中的代码，你将能把这些概念紧密联系起来并融会贯通。

应该谨记，本书中的练习并不是一份检验清单，似乎完成这些练习之后，你就自动具有了iPhone开发专家资格。在继续下一个项目之前，确保已经理解了这些概念和原理。不要害怕修改代码。多多尝试并观察结果是在Cocoa Touch环境中克服编码困难的最佳方法。

如果你已经安装了iPhone SDK，请继续阅读本书。如果还没有，请立即安装iPhone SDK。然后开始iPhone之旅！



任何编程书籍都习惯使用“Hello,World!”作为第一个项目，这已然成为了一种传统。我们考虑过打破这种常规，但又恐标新立异的做法引起人神共怒。因此，本书仍然采用常规做法。

本章将使用Xcode和Interface Builder创建一个小型iPhone应用程序，在屏幕上显示文本“Hello,World!”。我们将讨论使用Xcode创建iPhone应用程序的方方面面，深入使用Interface Builder设计应用程序用户界面的具体细节，最后在iPhone仿真器上运行应用程序。随后，我们将为应用程序指定一个图标和标识符，让它更具有真实iPhone应用程序的感觉。

要做的事情很多，我们出发吧。

2.1 在 Xcode 中设置项目

现在，你应该已经在机器上安装了Xcode和iPhone SDK。此外，还应该从Apress网站下载本书的项目归档文件。本书的页面链接如下：

<http://www.apress.com/book/view/9781430216261>

在页面左侧的Book Extras部分，找到Source Code链接。展开归档文件，将项目文件夹放置在合适的位置。

你已经拥有了完整的项目文件，但我们认为仅仅运行下载的项目是不够的，手动创建各个项目可以获得更大的收获。其中最大的一个原因就是，实践可以让你熟练掌握本书所介绍的各种工具和专业技能。跟随本书对程序的源代码进行修改，并亲身体验单击和拖动按钮及滑块，在此过程中你将得到无可比拟的收获。

我们的第一个项目位于02 Hello World文件夹中。如果你要创建自己的项目，请创建一个新的02 Hello World文件夹。

启动Xcode，它位于/Developer/Applications。如果这是你第一次使用Xcode，请不必担心，我们将指导你创建新项目。如果你已经具备经验，可以直接跳过这部分内容。

第一次启动Xcode时，会看到如图2-1所示的欢迎屏幕。欢迎屏幕包含一些有用的链接，包括iPhone和Mac OS X技术文档、教程视频、新闻、示例代码等。所有这些信息都包含在Apple的开发人员网站和Xcode的文档浏览器中。因此，如果今后不想看到该屏幕，只需在关闭Xcode之前

取消选中 Show at Launch 复选框。如果希望阅读其中的信息，当然没问题。完成阅读后，关闭该窗口，继续我们的项目。



图2-1 Xcode欢迎屏幕

说明 如果第一次启动Xcode时机器上连接有iPhone或iPod Touch，那么程序会询问你是否要使用该设备进行开发。此时，单击Ignore按钮。如果选择加入付费的iPhone Developer Program，那么将能够访问程序门户，了解如何使用iPhone或iPod Touch进行开发和测试。

创建一个新项目，方法是从File菜单中选择New Project...，或者按⌘N打开New Project帮助窗口（参见图2-2）。

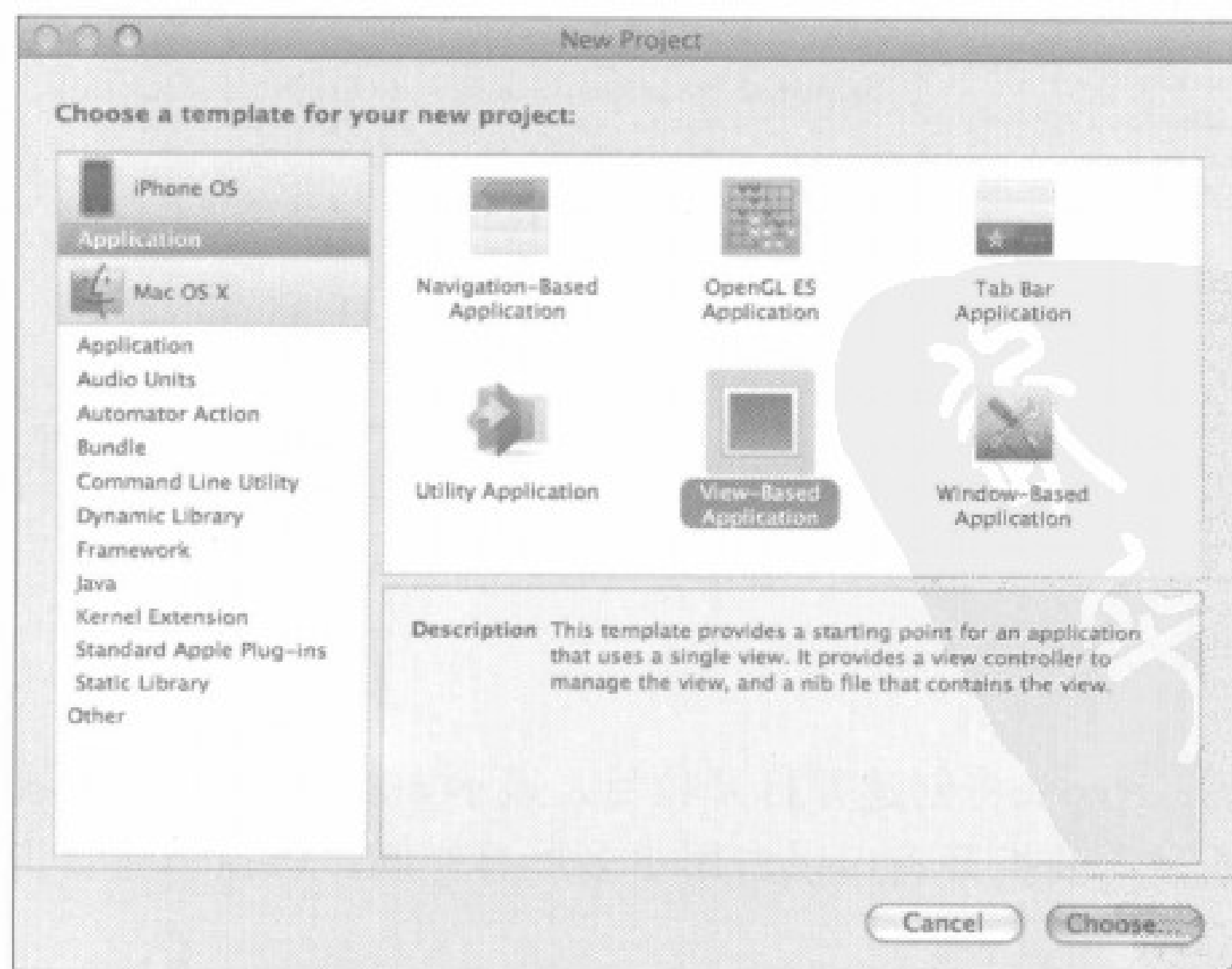


图2-2 New Project帮助窗口，你可以在创建新文件时选择各种文件模板

如图2-2所示，窗口左侧的窗格分为两个主要部分：iPhone和Mac OS X。可以看到，Mac OS X有许多可用的项目模板类别，但iPhone却只有一个类别（至少在撰写本书时是如此），即Application。

按图2-2所示操作，选择iPhone标题下方的Application，这将在右上方的窗格中显示一系列图标，每个图标表示可用作iPhone应用程序起始点的单独项目模板。标为View-Based Application的图标是最简单的模板，我们将在本书前几章中使用它。其他模板提供创建常见iPhone应用程序界面所需的其他代码和资源，并且包含一些深奥的高级功能。不必担心，我们稍后再对它们进行探讨。

对于第一个项目，单击View-Based Application图标（图2-2选择的的就是该图标），然后单击Choose按钮。

选择项目模板之后，需要使用标准保存表保存新项目（参见图2-3）。键入Hello World作为项目名称，并将它保存到适当的位置。Document文件夹是个不错的位置，不过你也可以为Xcode项目创建一个专用文件夹。

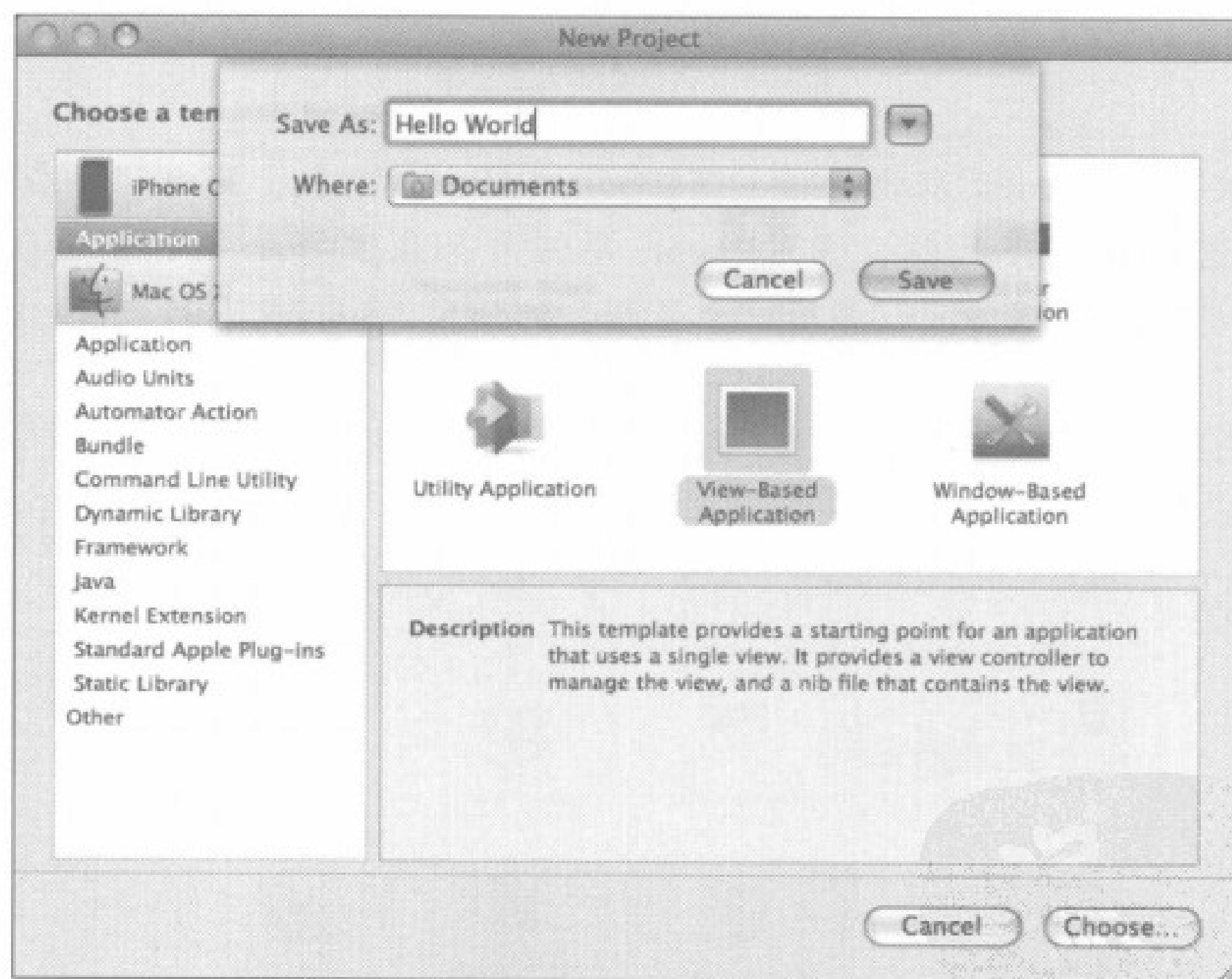


图2-3 选择项目的名称和位置

Xcode 项目窗口

离开保存面板之后，Xcode将创建并打开项目，显示如图2-4所示的新项目窗口。我们发现初始的项目窗口较小，通常需要扩开窗口以占用更多屏幕空间。该窗口包含许多信息，它是iPhone开发的主要窗口。

项目窗口顶部的工具栏提供了许多常用命令。工具栏下方的窗口分为3个主要部分（或窗格）。

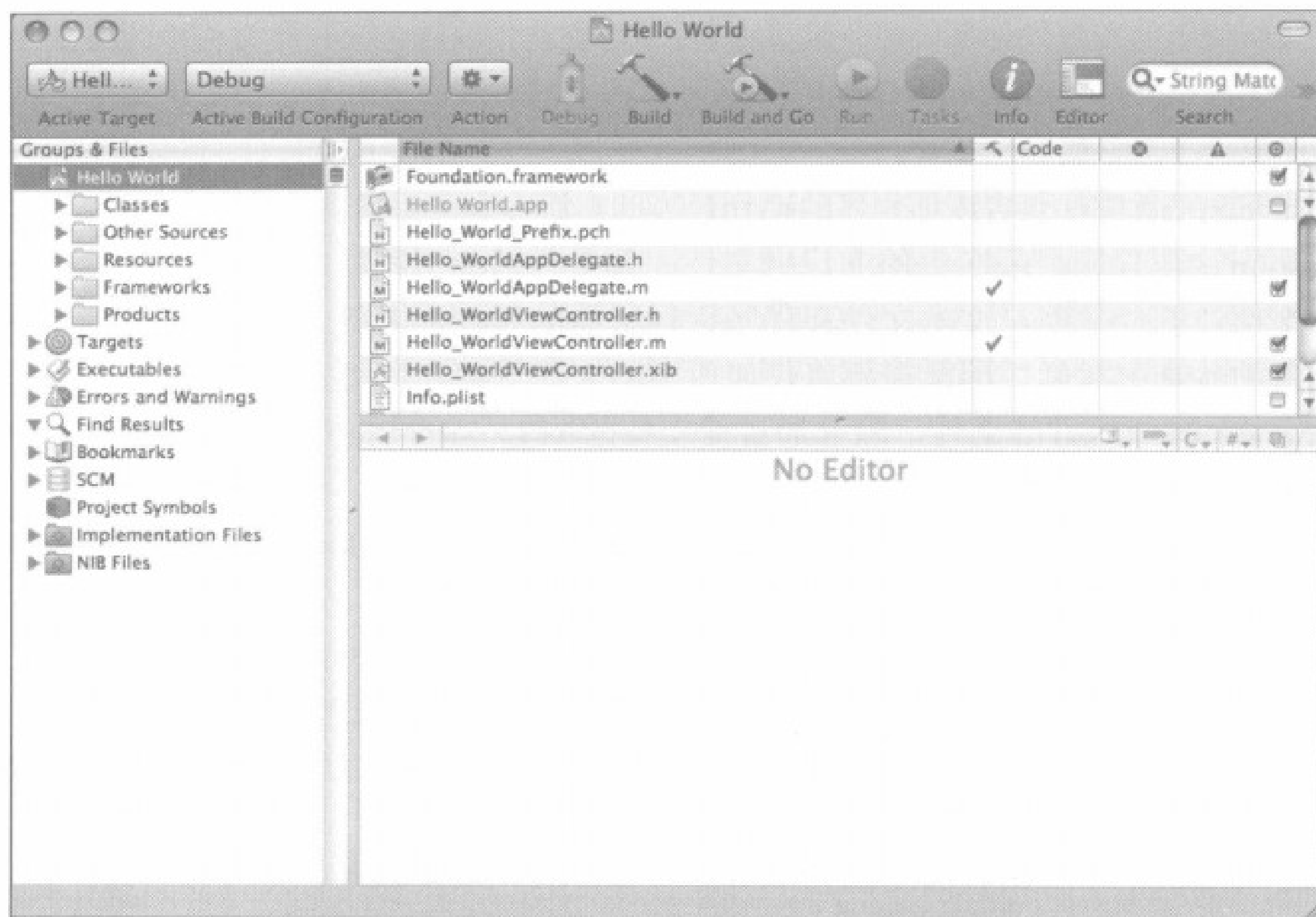


图2-4 Xcode中的Hello World项目

窗口左下方的窗格称为Groups & Files窗格。项目中的所有资源都在这里分类显示。其中还包括一系列相关的项目设置。与Finder类似，单击项目左侧的三角形图标可以展开该项目并显示可用的子项目。再次单击三角形图标可以隐藏子项目。

右上方的窗格称为Detail View，显示了在Groups & Files窗格中选择的项目的详细信息。右下方的窗格称为Editor窗格。如果选择Groups & Files或Detail中的某个文件，并且Xcode知道如何显示此类文件，则该文件的内容将在此处显示。可编辑的文件（如源代码）也可以在此处进行编辑。实际上，这就是编写和编辑应用程序源代码的地方。

了解了相关术语之后，我们先来看一下Groups & Files窗格。列表中的第一项应该与项目同名，在本例中为Hello World。此项汇聚了源代码和其他特定于项目的资源。目前只需关心Groups & Files窗格中Hello World下的项。

看一下图2-4。注意，Hello World左侧的三角形是打开的，其中包括5个子文件夹：Classes、Other Sources、Resources、Frameworks和Products。现在简要介绍各子文件夹的作用。

- ❑ Classes是最常用的一个文件夹。编写的大多数代码都保存在这里，其中包括所有的Objective-C类。可以在Classes文件夹下创建一些子文件夹来组织代码。我们将从下一章开始使用此文件夹。
- ❑ Other Sources包含除Objective-C类之外的源代码文件。通常，Other Sources文件夹不会占用你太多时间。创建新iPhone应用程序之后，此文件夹包含下面两个文件。
 - Hello_World_Prefix.pch：扩展名.pch表示“预编译的头文件（precompiled header）”。这是项

目所使用的来自外部框架的一组头文件。Xcode将预编译包含在此文件中的头文件，这会减少使用**Build**或**Build and Go**选项编译项目所需的时间。暂时不必担心该文件，因为其中已经包含了最常用的头文件。

- **main.m**: 此文件包含应用程序的**main()**方法。通常不需要编辑或修改此文件。
- **Resources**包含应用程序中的非代码文件。其中包括应用程序的图标图像和其他图像、声音文件、影片文件、文本文件和程序运行所需的属性列表等。请记住，由于应用程序在自己的沙盒中运行，因此需要将任何所需的文件保存在此处。这是因为你不能访问位于iPhone上其他地方的文件，除非通过受支持的API，如可以访问iPhone的相片库和地址簿的API。此文件夹中应包含下面3项。
 - **Hello_WorldViewController.xib**: 此文件包含程序Interface Builder所使用的信息，本章稍后将讨论它。
 - **Info.plist**: 包含应用程序相关信息的属性列表。本章稍后将讨论它。
 - **MainWindow.xib**: 应用程序的主Interface Builder（或nib）文件。在简单的应用程序中（如本章中构建的应用程序），通常不需要接触此文件。在后面的几章中，当设计比较复杂的界面时，我们将使用此文件并深入了解它。
- **Frameworks**是一种特殊的库，其中可以包含代码、图像和声音文件等资源。在此文件夹中添加的任何框架或库都将链接到应用程序中，并且代码将能够使用包含在该框架或库中的对象、函数和资源。项目中默认链接了最常用的框架和库，因此在大多数情况下，我们不需要操作此文件夹。但是，项目默认未包含较少使用的库和框架。本书后面的章节将介绍如何将它们链接到应用程序中。
- **Products**包含此项目在编译时生成的应用程序。展开**Products**，可以看到一个名称为**Hello World.app**的项。它是这个特定项目创建的应用程序。**Hello World.app**是项目的唯一产品。现在的**Hello World.app**显示为红色，这表示无法找到该文件，因为我们尚未编译项目！将文件的名称突出显示为红色，这是Xcode告诉我们它无法找到底层物理文件的方法。

说明 项目的**Groups & Files**窗格中的这些文件夹不一定与Mac文件系统上的文件夹相对应。Xcode已对它们进行了逻辑分组，以保持一切内容井然有序，便于用户在开发应用程序时更加快速地找到所需的内容。打开项目的文件夹，可以看到其中包含一个**Classes**文件夹，但是却没有**Other Sources**和**Resources**文件夹。通常，包含在这两个文件夹中的内容都保存在项目的根目录中，但是也可以将它们保存在任何位置，甚至位于项目文件夹的外部。Xcode内部的层次结构与文件系统的层次结构完全无关。举例来说，在Xcode中移出**Classes**文件夹中的文件不会更改该文件在硬盘上的位置。

2.2 Interface Builder 简介

现在，你已经熟悉了Xcode的基础知识。接下来，我们看一下iPhone软件开发中非常重要的一个工具：**Interface Builder**（通常简称为**IB**）。

在项目窗口的 Groups & Files 列表中，展开 Resources 组，然后双击 Hello_WorldViewController.xib 文件。这将在 Interface Builder 中打开该文件。在第一次使用 Interface Builder 时，窗口的分组方式应该与图 2-5 类似。如果之前使用过 Interface Builder，则各窗口的位置将与上次使用时相同。

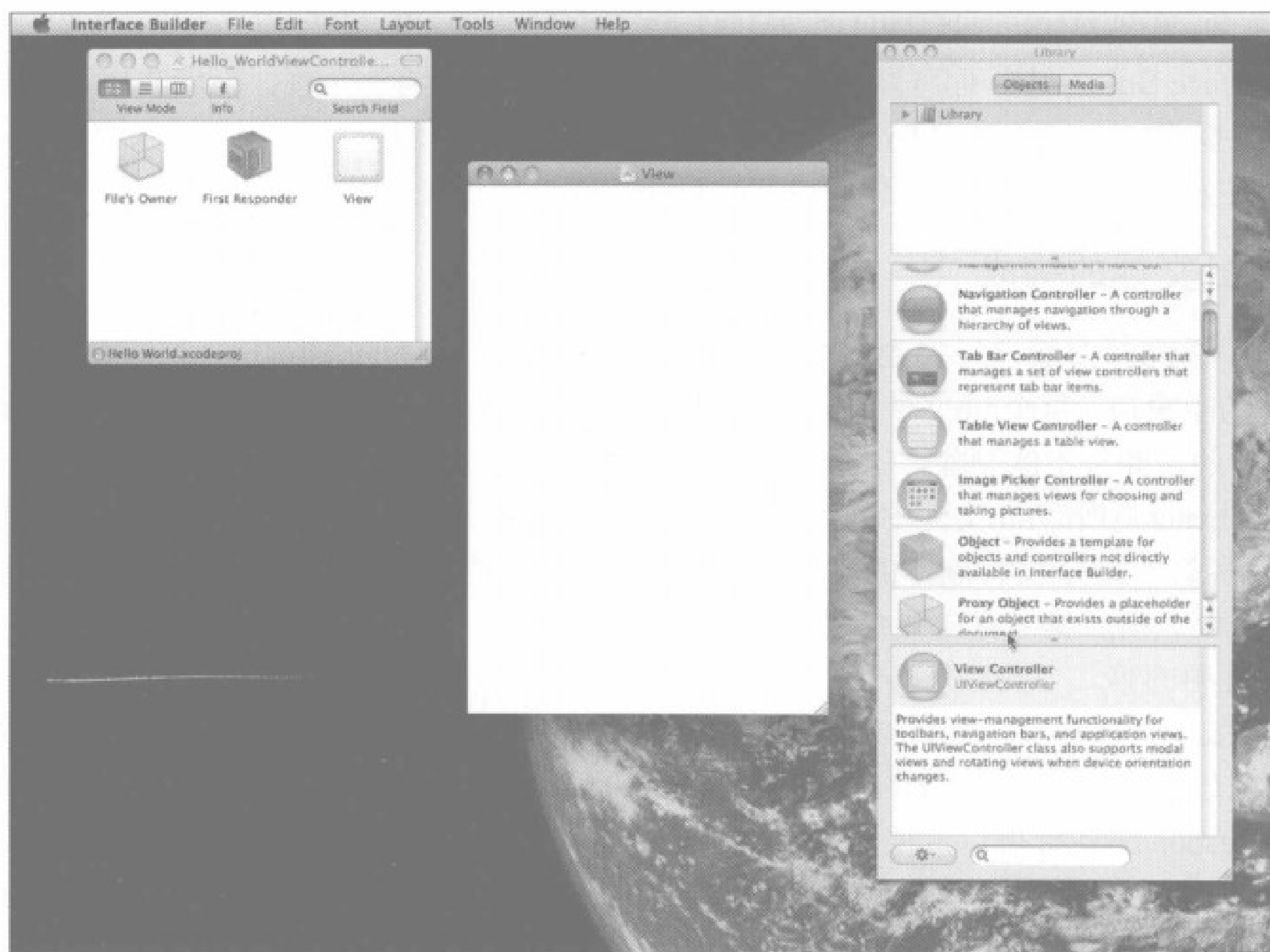


图 2-5 Interface Builder 中的 Hello_WorldViewController.xib

说明 Interface Builder 有一段较长的历史。它于 1988 年首次面世，并且曾用于为 NextSTEP、OpenSTEP、Mac OS X 和现在的 iPhone 开发应用程序。Interface Builder 支持两种文件类型：使用 .nib 扩展名的旧格式和使用 .xib 扩展名的新格式。iPhone 项目模板默认全部使用 .xib 文件，但直到不久前，所有 Interface Builder 文件还都使用 .nib 扩展名。结果，大多数开发人员将 Interface Builder 文件称为“nib 文件”。称它为 nib 文件，与该文件实际使用的是 .xib 扩展名还是 .nib 扩展名无关。实际上，苹果公司在其文档中统一使用术语“nib”和“nib 文件”。

标为 Hello_WorldViewController.xib 的窗口（图 2-5 左上方的窗口）是 nib 的主窗口。对于此特定的 nib 文件，该窗口是开发的主窗口和起始点。除了前两个图标（File's Owner 和 First Responder）之外，该窗口中的任何图标都表示 Objective-C 类的一个实例，它们将在加载 nib 文件时自动创建。

想创建一个按钮实例吗？当然，可以通过编写代码来创建按钮。但是，更加常用的方法是使用 Interface Builder 创建按钮并指定其属性（形状、大小、标签等）。

Hello_WorldViewController.xib文件将在应用程序启动时自动加载（目前不用关心其加载方式），因此它非常适合用于保存构成用户界面的对象。

例如，要在应用程序中添加一个按钮，需要实例化一个UIButton类型的对象。可以通过键入以下代码行来实现此目的：

```
UIButton *myButton = [[UIButton alloc] initWithFrame:aRect];
```

在Interface Builder中，你可以通过从界面对象组件面板中将一个按钮拖到应用程序的主窗口中来完成此操作。Interface Builder可以方便地设置按钮的属性，并且由于按钮将保存在nib文件中，因此按钮的实例化操作将在应用程序启动时自动完成。你很快就可以看到其运行原理。

2.2.1 nib 文件的构成

看一下图2-5。前面已经提到，标为Hello_WorldViewController.xib的窗口（左上角的窗口）是nib文件的主窗口。每个nib文件最初都有同样的两个图标：File's Owner和First Responder。它们是自动创建的，并且不能删除。鉴于此，你可能会猜测它们非常重要，事实也的确如此。

File's Owner是所有nib文件中的第一个图标，它表示从磁盘加载nib文件的对象。换言之，File's Owner是“拥有”此nib文件副本的对象。这可能有点令人费解，不过也不用担心，这个概念目前还不是很重要。稍后我们将详细介绍这方面的内容。

nib文件中的第二个图标是First Responder。本书将在后面的章节详细讨论Responder。简而言之，First Responder就是用户当前正在与之交互的对象。例如，如果用户当前正在文本字段中输入数据，则该字段就是当前的First Responder。First Responder将随着用户与界面的交互而变化，并且First Responder图标可以方便开发人员操作当前作为First Responder的控件或视图，而不需要编写代码来确定这些控件和视图。如有疑问，请不用担心，本书将在后面的章节详细介绍这方面内容。

除这两者之外，此窗口中的任何其他图标都表示将在nib文件加载时创建的对象实例。在本例中，如图2-5所示，第三个图标的名称为View。

View图标表示UIView类的一个实例。UIView对象是用户可以看到并能与之交互的区域。在此应用程序中，我们只拥有一个视图，因此该图标表示用户可以在应用程序中看到的所有内容。后面的章节将构建比较复杂的、拥有多个视图的应用程序。但现在，只需要考虑用户使用应用程序时可以看到的内容。

说明 从技术上说，我们的应用程序拥有的视图不止一个。可以在屏幕上显示的所有用户界面元素，包括按钮、文本字段和标签，都是UIView的子类。但是，当你在本书中看到术语“视图”时，我们通常指的是UIView的实际实例，而此应用程序仅有一个UIView实例。

从图2-5中可以看到，除主窗口之外，还打开了另外两个窗口。查看标题为View的窗口。该窗口是nib主窗口中的第三个图标的图形表示。如果关闭该窗口并双击nib文件主窗口中的View图标，则该窗口将再次打开。该窗口用于图形化设计用户界面。接下来将介绍如何完成此操作。

2.2.2 在视图中添加标签

图2-5中最右侧的窗口是Library窗口，图2-6显示了其详细信息。其中包含Interface Builder支持的所有常备的Cocoa Touch对象。从库中拖一个项放到nib文件窗口中，应用程序中将添加该类的一个实例。关闭库窗口之后，可以通过从Tools菜单中选择Library重新调出该窗口。此组件面板上的项主要来自iPhone UIKit，后者是用于创建应用程序用户界面的对象框架。

UIKit在Cocoa Touch中的作用与AppKit在Cocoa中的作用相同。这两种框架在概念上是类似的，但由于平台之间的差异，它们之间存在许多明显的不同。另一方面，一些Foundation框架类，如NSString和NSArray，是Cocoa和Cocoa Touch所共有的。

浏览Library组件面板中的对象列表，找到Label组件（参见图2-7）。

标签表示可以在iPhone上显示，但不允许用户直接编辑的文本。稍后，我们将在视图添加一个标签。

由于用户界面对象具有层次结构，因此我们将标签作为子视图添加到主视图（View视图）中。Interface Builder非常智能。如果某个对象不接受子视图，则不能将其他对象拖放到其上。

从库中将一个标签拖到View视图中，将会添加一个UILabel实例作为应用程序主视图的子视图。明白了吗？

现在执行此操作。从Library组件面板中将一个Label拖到View窗口中。完成后的视图应如图2-8所示。

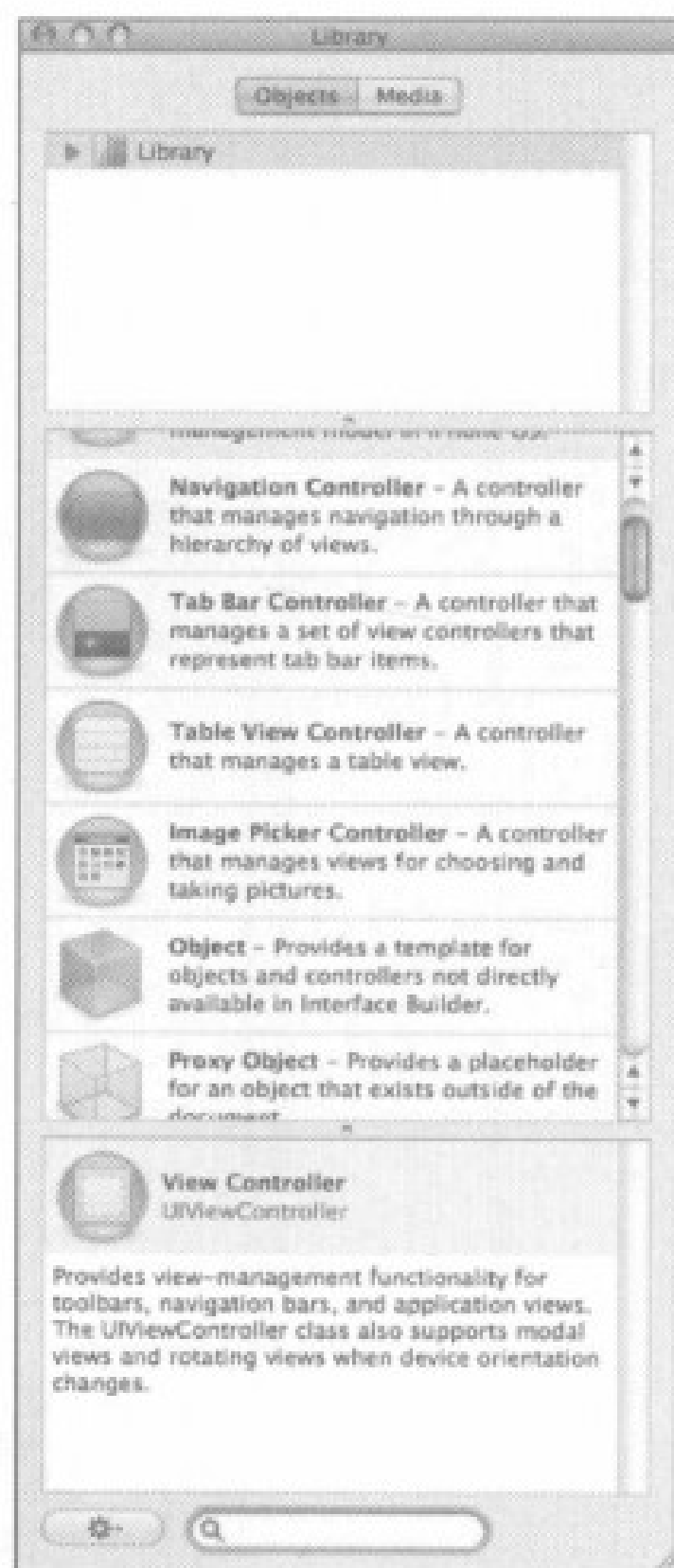


图2-6 Library组件面板，其中包含UIKit中的常备对象，可供Interface Builder使用

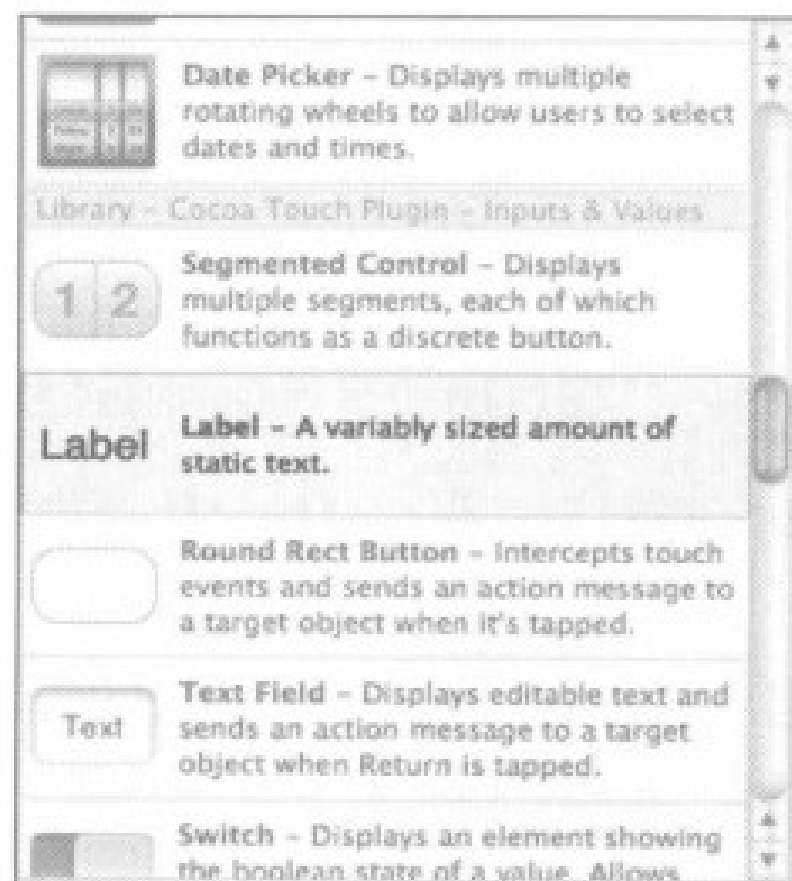


图2-7 Library组件面板中的Label对象



图2-8 在应用程序的View窗口中添加一个标签

接下来编辑标签的显示文本。双击刚才创建的标签并键入文本Hello, world!。然后, 将标签移动至屏幕上的适当位置。

大功告成。最后保存应用程序即可。从**File**菜单中选择**Save**, 然后返回Xcode, 以便构建和运行应用程序。

在Xcode中, 从**Build**菜单中选择**Build and Run** (或者按下⌘R)。Xcode将编译应用程序并在iPhone仿真器中启动它, 如图2-9所示。



图2-9 iPhone中的Hello World程序

欣赏完自己的作品之后, 确保退出仿真器。Xcode、Interface Builder和仿真器都是单独的应用程序。

请等一下! 难道就这么简单吗? 我们并没有编写任何代码。

没错。就是这样, 不敢相信吧?

但是, 如果我们希望修改标签的一些属性, 如文本大小或颜色, 应该如何实现呢? 我们需要编写代码, 对吗?

回答是: 完全不用。

返回Interface Builder并单击Hello World标签选中它。现在按下⌘I或从**Tools**菜单中选择**Inspector**。这将调出**Inspector**窗口, 在该窗口中, 你可以设置当前选中项的属性 (参见图2-10)。

在Inspector中, 你可以修改字体大小、颜色和阴影等各种属性。Inspector是对上下文敏感的。如果选择了某个文本字段, 它会显示该文本字段的编辑属性。如果选择了某个按钮, 它会显示该按钮的可编辑属性, 等等。

根据需要修改标签的外观并保存, 然后返回Xcode, 再次选择**Build and Run**。所做的更改应该

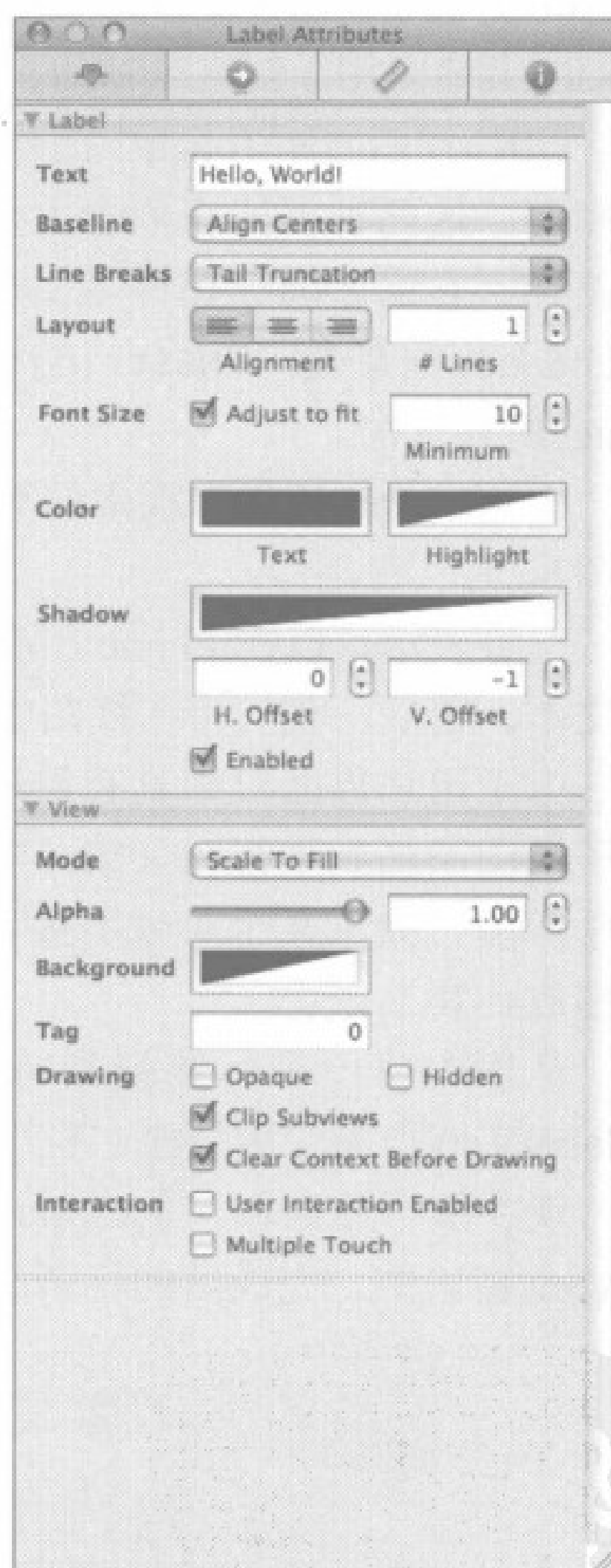


图2-10 显示标签属性的Inspector

显示在应用程序中，这一次同样没有编写任何代码。Interface Builder支持以图形化的方式设计界面，从而使开发人员能够专注于编写特定于应用程序的代码，而不必受累于编写单调的用户界面代码。

注意 如果在构建和运行应用程序时，iPhone已经连接到Mac，则可能会不太顺利。总地来说，要在iPhone上构建和运行应用程序，需要注册苹果公司的iPhone开发人员计划并支付一定的费用，然后才能适当配置Xcode。加入该计划之后，苹果公司将向你发送完成配置所需的信息。同时，本书中的大多数程序都可以通过iPhone仿真器正确运行。如果已经连接了iPhone，在选择**Build and Run**之前，需先从**Project**菜单中依次选择**Set Active SDK**和**Simulator—iPhone OS 2.0**。

说明 大多数现代应用程序开发环境都提供了一些工具用于以图形化的方式构建用户界面。Interface Builder与其他许多工具之间的一个显著区别就是，Interface Builder不会生成任何需要进行维护的代码。相反，Interface Builder将创建Objective-C对象，然后将这些对象序列化到nib文件中，以便在运行时将它们直接加载到内存中。这避免了与代码生成相关的许多问题，并且总地来说是一种比较强大的方法。

2.3 iPhone 美化

在本章结束之前，我们将对应用程序进行一些美化，让它更加接近真实的iPhone应用程序。首先，运行项目。当仿真器窗口出现之后，单击iPhone的主按钮，即窗口底部含白色方块的黑色按钮。这将返回iPhone的主屏幕（参见图2-11）。注意到什么问题了吗？

注意屏幕顶部的Hello World图标。该图标是空白的。要解决此问题，需要创建一个图标，并将它保存为一个可移植网络图形（.png）文件。它的大小必须为57×57像素。不要尝试与电话上已有按钮的样式相匹配，iPhone将自动圆角化边缘并让它具有玻璃质感。只需创建一个普通的方形图像。我们在项目归档文件（位于02 Hello World文件夹）中提供了一个图标文件，如果不愿意自己创建图标，也可以直接使用它。

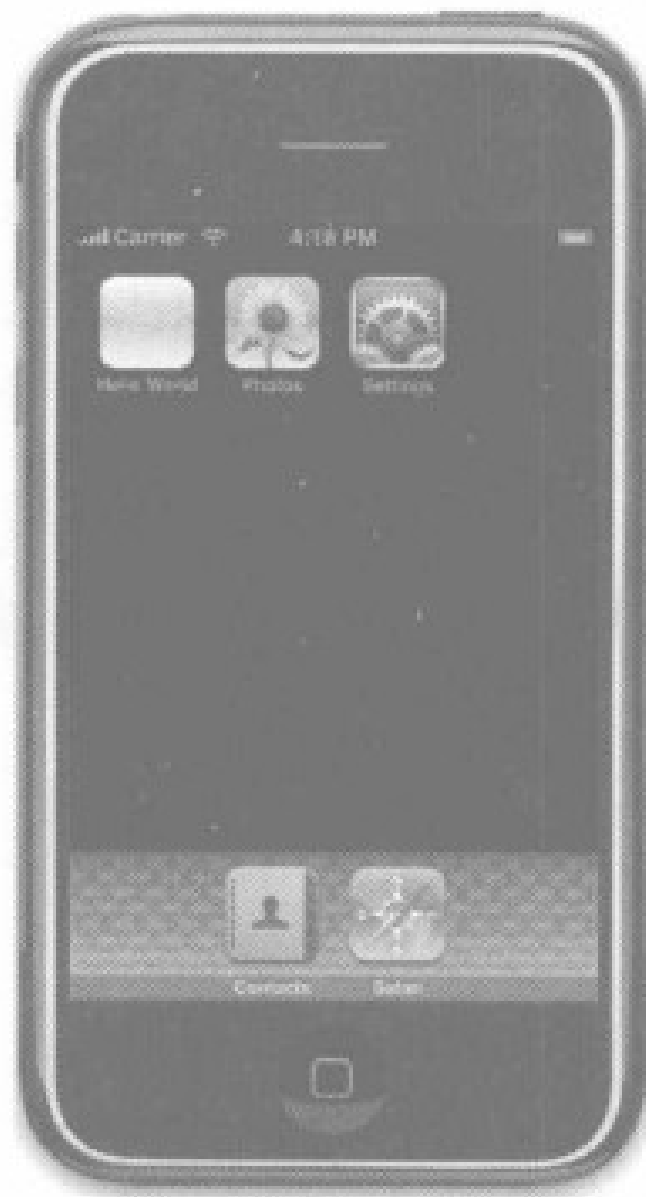


图2-11 最左侧的应用程序图标是空白的

说明 需要使用.png图像作为应用程序的图标，实际上，添加到iPhone项目中的图像都应该使用此格式。虽然大多数常见图像格式都可以正确显示，但是，除非有不可抗拒的原因，否则都应该使用.png文件。Xcode在构建应用程序时会自动优化.png图像，让它们成为iPhone应用程序中最快速和最有效的图像类型。

设计好应用程序图标之后，将.png文件从Finder中拖放到Xcode的Resources文件夹中，如图2-12所示。或者，选择Xcode中的Resources文件夹，从**Project**菜单中选择**Add to Project...**，然后导航到你的图标图像文件。

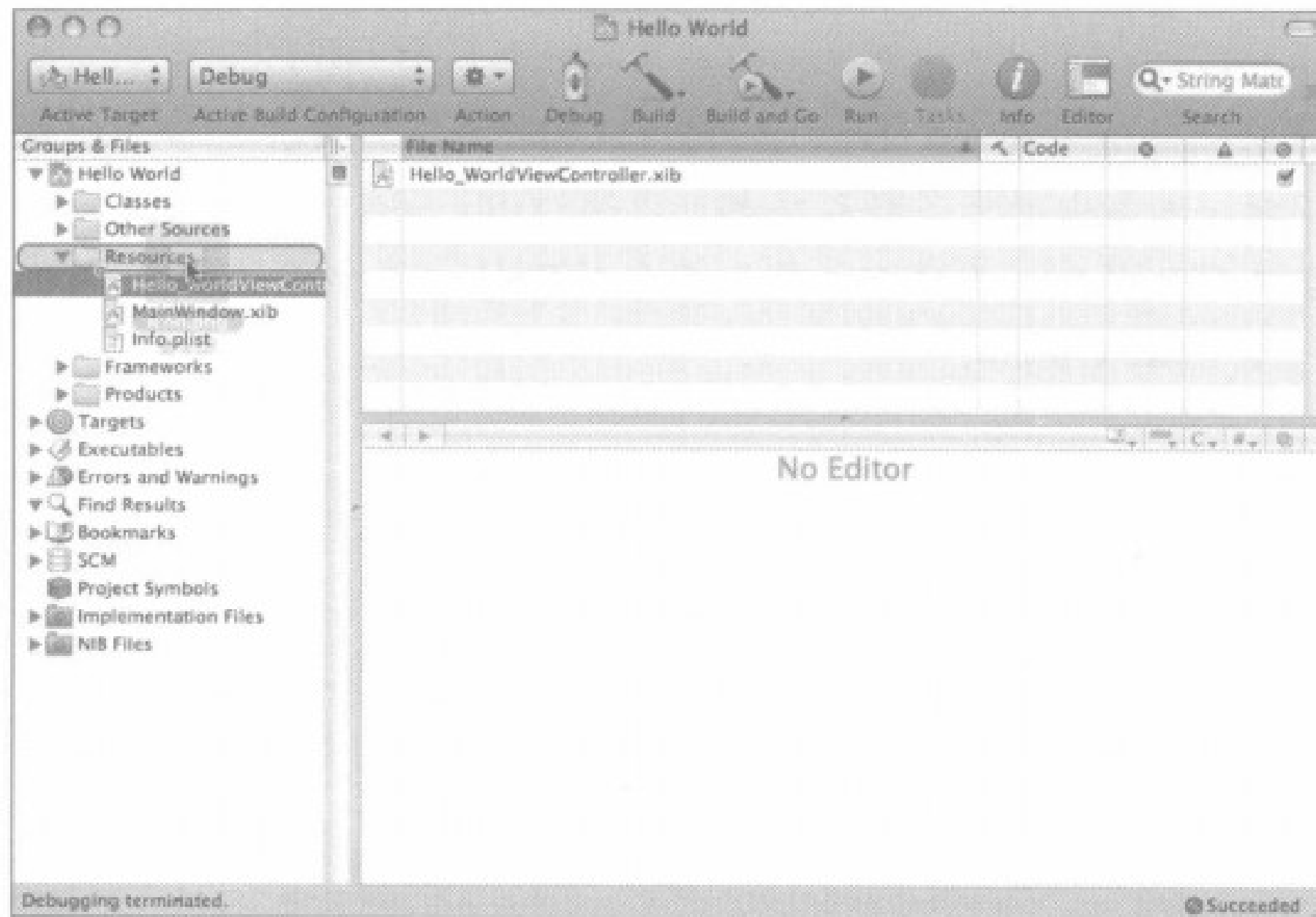


图2-12 将图标文件拖到Xcode项目的Resources文件夹中

完成上述操作之后，Xcode将提示你输入一些具体信息（参见图2-13）。可以选择让Xcode将该文件复制到项目目录中，也可以将它作为到原文件的引用添加到项目中。通常，较好的选择是将资源复制到Xcode项目中，除非该文件需要与其他项目共享。

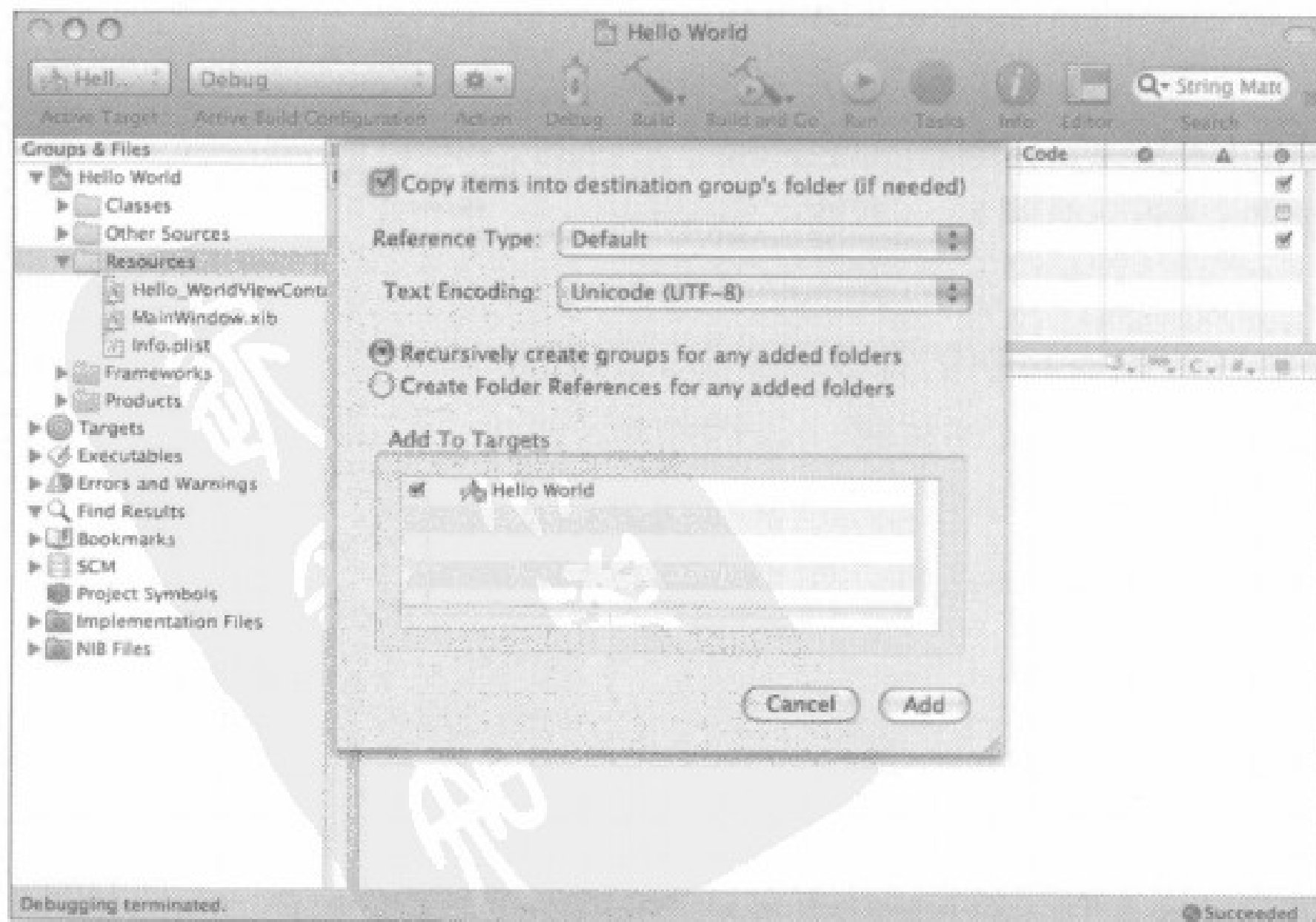


图2-13 选择如何将文件添加到项目中

在项目中添加某个常用文件之后，Xcode就会知道如何处理它们。结果，该图像文件将自动编译到应用程序中，而不需要任何其他操作。

现在，我们已经将icon.png图像并入到项目中，其结果是该图像将构建到应用程序的程序包中。接下来，我们需要指定使用此图像作为应用程序的图标。

在Xcode项目窗口的Groups & Files窗格中，展开Resources文件夹，然后单击Info.plist文件。它是一个属性列表文件，其中包含关于应用程序的一些常规信息，如图标文件的名称等。

选择Info.plist之后，属性列表将出现在正在编辑的窗格中（参见图2-14）。在属性列表中，从左列中找到标为Icon file的行。该行右边的列应该为空。双击该空白单元格，键入刚才添加到项目中的.png文件的名称。

Key	Value
▼ Information Property List	(12 items)
Localization native development re	en
Bundle display name	\$(PRODUCT_NAME)
Executable file	\$(EXECUTABLE_NAME)
Icon file	
Bundle identifier	com.yourcompany.\$(PRODUCT_NAME:identifier)
InfoDictionary version	6.0
Bundle name	\$(PRODUCT_NAME)
Bundle OS Type code	APPL
Bundle creator OS Type code	????
Bundle version	1.0
LSRequiresiPhoneOS	<input checked="" type="checkbox"/>
Main nib file base name	MainWindow

图2-14 指定图标文件

准备编译和运行

在编译和运行之前，先来看看Info.plist中的其他行。虽然大多数设置保留默认设置即可，但是需要注意一个特定的设置，即束标识符（bundle identifier）。这是应用程序的唯一标识符，并且始终需要设置。如果要在iPhone仿真器上运行应用程序，则束标识符的标准命名约定的格式为：顶级Internet域（如com或org）之后是点号，然后是公司或组织名称，接着是点号，最后是应用程序的名称。如果要在实际的iPhone上运行应用程序，则创建应用程序的束标识符是一个较为复杂的过程，后面的章节将详细介绍其方法。目前，我们将束标识符修改为com.apress.HelloWorld。

完成更改之后，编译并运行应用程序。启动仿真器之后，按白色方块按钮返回主屏幕，可以看到漂亮的新图标。如图2-15所示。

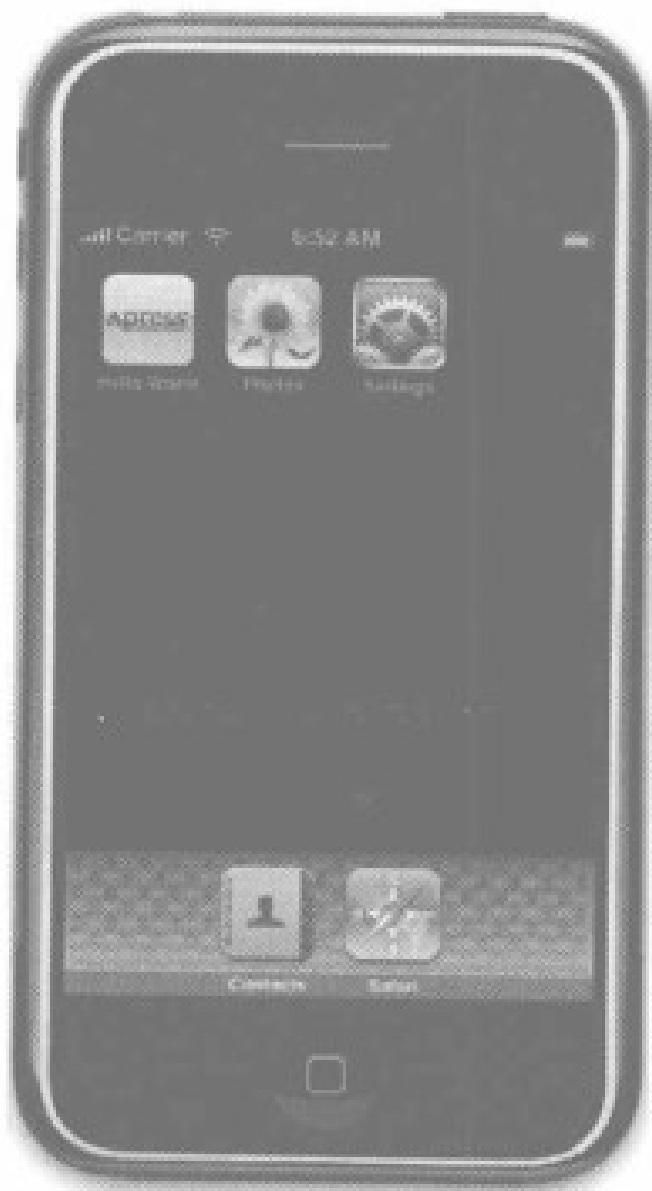


图2-15 应用程序现在拥有一个漂亮的图标

说明 如果要从iPhone仿真器的主屏幕中清除早期的应用程序，可以从主目录的Library文件夹的Application Support文件夹中删除iPhone Simulator文件夹。

2.4 小结

你可以暂时缓一口气了。虽然本章无法让你掌握iPhone开发的所有技巧，但我们确实涵盖了不少内容。你了解了iPhone项目模板，如何创建应用程序，如何使用Interface Builder，以及如何设置应用程序图标和束标识符。

但是，Hello World是一个严格的单向应用程序：我们向用户显示了一些信息，但并未让用户输入任何信息。下一章将介绍如何获取iPhone用户的输入，并根据该输入做出决策。来，请做个深呼吸，然后翻页吧。



上一章的Hello World应用程序很好地介绍了如何使用Cocoa Touch进行iPhone开发，但它缺少了一项至关重要的功能：与用户交互的能力。如果没有良好的交互性，应用程序的功能将受到极大限制。

本章将编写一个稍微复杂的应用程序，它具备两个按钮和一个标签（参见图3-1）。当用户按下任意一个按钮时，标签的文本将随之变化。这看上去是一个相当简单的示例，但它演示了iPhone控件开发所需的关键概念。

3.1 模型-视图-控制器范型

首先，让我们了解一些基本理论。Cocoa Touch设计者们采用模型-视图-控制器（MVC）泛型作为指导原则。MVC是用于拆分GUI应用程序代码的逻辑方法。目前，几乎所有面向对象框架都对MVC敬畏三分，但很少有像Cocoa Touch这样钟实于MVC的。

MVC模型将所有功能划分为3个不同类别。

- 模型：保存应用程序数据的类。
- 视图：窗口、控件和其他用户可以看到并能与之交互的元素的组成部分。
- 控制器：将模型和视图绑定在一起，确定如何处理用户输入的应用程序逻辑。

MVC的目标是实现3类尽可能截然不同的代码。编写的任何对象都应该能很明显地划分为其中一类，并且其功能大部分不属于或完全不属于另外两类。例如，实现某个按钮的对象不应包含处理按下按钮事件时的数据，而实现银行账户的代码不应包含绘制表格以显示其事务的代码。

MVC可以帮助确保实现最大可重用性。实现普通按钮的类可以在任何应用程序中使用。如果某个类实现的按钮将在被单击时执行一些特定的计算，那么此类仅能在其最初的应用程序中使用。

在编写Cocoa Touch应用程序时，我们将主要使用Interface Builder来创建视图组件，但有时仍



图3-1 本章将构建的简单应用程序，它具备两个按钮

然需要在代码中修改界面，或者需要继承已有的视图和控件。

创建模型的方法是设计一些Objective-C类来保存应用程序数据。在本章的应用程序中，我们不会创建任何模型对象，因为我们不需要存储或保留数据。但在后面的章节中，随着应用程序变得更加复杂，我们将引入模型对象。

控制器组件通常由开发人员创建的类和特定于应用程序的类组成。控制器可以是完全自定义的类（NSObject子类），但更多情况下，它们一般是UIKit框架中已有通用控制器类（如UIViewController，稍后介绍）的子类。通过继承其中的一个已有类，你可以免费获取大量功能，并且可以说不再需要重新设计类的结构。

随着对Cocoa Touch的深入了解，你会很快发现UIKit框架中的类是如何遵循MVC原则的。如果在开发的时候遵循这个概念，你将能够创建更加简洁、更易于维护的代码。

3.2 创建项目

现在可以开始创建Xcode项目了。我们将使用与上一章相同的模板：基于视图的应用程序。我们不用多久便会使用其他模板，但这次仍然从这个简单的模板入手。这样更便于读者了解视图和控制器对象在iPhone应用程序中的协作。继续创建项目，将其保存为Button Fun。如果在创建项目时遇到了任何问题，请参阅上一章了解详细步骤。

你可能还记得，项目模板将为我们创建一些类。可以在新项目中找到这些相同的类，但它们的名称可能稍有不同，因为类名是基于项目名的。

3.3 创建视图控制器

稍后，我们将与上一章一样，使用Interface Builder为应用程序设计一个视图（或用户界面）。在此之前，我们将对已经创建的源代码文件进行一些修改。没错，我们将在本章中实际编写一些代码。

在修改之前，我们先看看这些已经创建的文件。在项目窗口中，展开Class文件夹，可以看到其中包含4个文件（参见图3-2）。

这4个文件实现了两个类，分别包含一个.m文件和一个.h文件。本章将创建的应用程序仅包含一个视图，且控制器类Button_FunViewController将负责管理该视图。单击Groups & Files窗格中的Button_FunViewController.h，并查看该文件的内容：

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

```
@interface Button_FunViewController : UIViewController {
```

```
}
```

```
@end
```

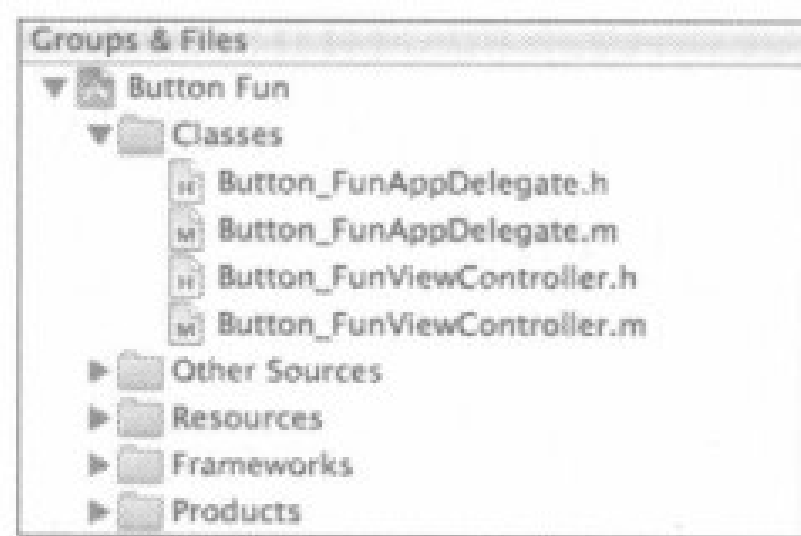


图3-2 项目模板创建的类文件

代码很短，对吗？这是UINavigationController的子类，UINavigationController也是我们之前提到的一个通用控制器。它是UIKit的一部分，并为我们免费提供了一些功能。Xcode不知道我们的应用程序具有哪些特定功能，但它知道一些基本功能，因为创建的这个类保存了这些功能。

回顾一下图3-1。我们的程序包含两个按钮和一个用于反映按钮状态的文本标签。我们将在Interface Builder中创建所有这3个元素。既然我们将编写代码，那么代码必须要通过某种方式与我们在Interface Builder中创建的元素进行交互。

非常正确。控制器类可以使用一种特殊的实例变量来引用nib中的对象，这个变量就是输出口（outlet）。可以把输出口看成是指向nib中的对象的指针。举例来说，假设你在Interface Builder中创建了一个文本标签，并且希望在代码中修改该标签的文本。通过声明一个输出口，并将其指向此标签对象，你可以在代码中使用该输出口来修改标签。稍后你将看到如何实现此操作。

另一方面，也可以设置nib文件中的界面对象触发控制器类中的特殊方法。这些特殊方法称为操作方法。举例来说，你可以告诉Interface Builder，当用户放开（手指离开屏幕）某个按钮时，应当调用代码中的特定操作方法。

我们的下一个程序将具备两个按钮和一个标签。

在代码中，我们将创建一个指向标签的输出口，并且此输出口允许我们更改该标签的文本。我们还将创建一个名为buttonPressed:的方法，该方法将在按下任意一个按钮时被触发。buttonPressed:将设置标签文本，让用户知道按下的是哪一个按钮。

我们将使用Interface Builder创建按钮和标签，然后通过单击和拖放操作将标签连接到标签输出口，并将按钮连接到buttonPressed:操作。

但在开始编写代码之前，下面给出了关于输出口和操作的详细信息。

3.3.1 输出口

输出口是使用关键字IBOutlet声明的实例变量。控制器头文件中的输出口声明应如下所示：

```
IBOutlet UIButton *myButton;
```

IBOutlet关键字的定义如下所示：

```
#ifndef IBOutlet
#define IBOutlet
#endif
```

感到困惑了吗？就编译器而言，IBOutlet并未执行任何操作。它的唯一作用是告诉Interface Builder，此实例变量将被连接到nib中的对象。你创建的任何需要连接到nib文件中的对象的实例变量都必须以IBOutlet关键字开头。打开Interface Builder时，它会在项目头文件中扫描此关键字，并将允许你根据这些（且只能根据这些）变量将代码连接到nib。稍后，你将了解如何通过Interface Builder在输出口和用户界面之间建立连接。

3.3.2 操作

操作是控制器类中的方法。它们也是通过特殊关键字IBAction声明的，该关键字告诉Interface

Builder，此方法是一个操作，且可以被某个控件触发。通常，操作方法的声明应如下所示：

```
- (IBAction)doSomething:(id)sender;
```

该方法的实际名称没有任何限制，但它的返回类型必须是**IBAction**，与声明**void**返回类型相同。这是声明操作方法不返回任何值的另一种方法。通常，操作方法接受一个参数，该参数通常被定义为**id**，名称被指定为**sender**。触发操作的控件将使用**sender**参数引用其自身。因此，举例来说，如果你的操作方法将在按下按钮时被调用，则**sender**参数将包含对该特定按钮（被按下的）的引用。

你将看到，我们的程序将使用**sender**参数将标签设置为文本“left”或“right”，这取决于按下的是哪一个按钮。如果不需要知道哪个控件调用了方法，那么也可以定义无**sender**参数的操作方法。如下所示：

```
- (IBAction)doSomething;
```

如果声明了一个带**sender**参数的操作方法，而随后又忽略了**sender**参数，这也不会造成任何不利影响。你可能会看到许多代码都采用了这种方法，因为在过去，Cocoa中的操作方法需要接受**sender**参数，而不管是否要使用它。

3.3.3 将操作和输出口添加到视图控制器

现在，你已经了解了输出口与操作的基本概念。接下来，让我们将它们添加到控制器类中。我们需要通过输出口来修改标签的文本。由于我们不会修改按钮，因此也不需要相应的输出口。

我们还将声明一个操作方法，将由两个按钮进行调用。虽然许多操作方法都是特定于某个控件的，但也可以使用一个操作来处理来自多个控件的输入，这正是本文将采用的方法。操作将通过**sender**参数获取按钮的名称，并使用标签输出口将该按钮名称嵌入到标签的文本中。你稍后将了解如何实现此操作。

说明 由于Xcode为我们创建的文件已经包含了一些需要的代码，因此我们通常只需在已有文件中插入代码。查看Button_FunViewController.h的代码清单，可以发现**normal typeface**类型的任何代码都是文件中的已有代码。粗体显示的代码为需要键入的新代码。

继续操作，将以下代码添加到Button_FunViewController.h：

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

```
@interface Button_FunViewController : UIViewController {
    IBOutlet UILabel *statusText;
}
```

```
@property (retain, nonatomic) UILabel *statusText;
```

```
- (IBAction)buttonPressed:(id)sender;
@end
```

使用过Objective-C 2.0的人对于@property声明可能比较熟悉。如果未使用过Objective-C 2.0, 则以上代码会让人感到恐惧。不必担心, Objective-C属性其实相当简单。因为它们相对较新且将在本书中广泛使用, 所以接下来简要介绍一下它们。即使你已经掌握了这些属性, 也请继续阅读, 因为一些特定于Cocoa Touch的信息是非常实用的。

Objective-C属性

在Objective-C中添加属性之前, 程序员通常会定义一些方法用于设置和检索类的实例变量的值。这些方法称为访问方法和修改方法(如果愿意, 也可以称之为获取方法和设置方法), 如下所示其形式为:

```
- (id) foo
{
    return foo;
}
- (void) setFoo: (id) aFoo
{
    if (aFoo != foo)
    {
        [aFoo retain];
        [foo release];
        foo = aFoo;
    }
}
```

虽然此方法仍然十分奏效, 但@property声明让你告别了枯燥乏味的访问方法和修改方法。我们刚才键入的@property声明与实现文件(@synthesize)中的另一个声明相结合, 可以通知编译器在编译时创建获取方法和设置方法。此处, 仍然需要声明底层实例变量, 但不需要定义访问方法或赋值方法。

在我们的声明中, @property关键字后面紧跟着一些可选属性(位于圆括号内)。它们进一步定义编译器将如何创建访问方法和修改方法。此处的两个属性通常在iPhone应用程序中定义属性时使用:

```
@property (retain, nonatomic) UILabel *statusText;
```

第一个属性retain通知编译器向分配给此属性的对象发送一个保留(retain)消息。这将确保属性底层的实例变量在使用过程中不会从内存中清除。这是必不可少的, 因为默认行为(assign)需要与垃圾收集一起使用。垃圾收集是iPhone当前尚未具备的一个Objective-C 2.0特性。因此, 如果定义的属性是一个对象(与int等原始数据类型相反), 则通常应该在可选属性中指定retain。为int、float或其他原始数据类型声明属性时, 不需要指定任何可选属性。

第二个可选属性nonatomic将更改访问方法和修改方法的生成方式。简单来说, 在默认情况下, 这些方法在创建时会具备另外一些代码, 用于帮助你编写多线程程序。这些额外开销虽然较小, 但对于声明指向用户界面对象的指针没有必要, 因此我们通过声明nonatomic来节省一些开销。在少数情况下, 你不希望为属性指定nonatomic(见本书稍后部分)。作为一般规则, 大多数

情况下，我们在编写iPhone应用程序时都将指定nonatomic属性。

Objective-C属性还具备另一个非常优秀的特性。它们在语言中引入了点表示法。通常，要使用访问方法，就需要向对象发送一个条消息，如下所示：

```
myVar = [someObject foo];
```

此方法仍然奏效。但是，如果已经定义了一个属性，也可以选择使用点表示法。这类似于Java、C++和C#，如下所示：

```
myVar = someObject.foo;
```

就编译器而言，这两条语句是完全相同的；你可以任意选择一种。点表示法同时适用于修改方法。下面这条语句：

```
someObject.foo = myVar;
```

在功能上等价于：

```
[someObject setFoo:myVar];
```

要了解新Objective-C属性的更多信息，请阅读Mark Dalrymple和Scott Knaster编写的*Objective-C on the Mac*第二版（Apress 2008），以及苹果公司开发人员网站上的“The Objective-C 2.0 Programming Language”，地址为：<http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf>。

3.3.4 将操作和输出口添加到实现文件

目前，我们的控制器类头文件已经告一段落，因此保存它并单击类的实现文件Button_FunViewController.m。如下所示：

```
#import "Button_FunViewController.h"

@implementation Button_FunViewController

/*
 Implement loadView if you want to create a view hierarchy programmatically
 - (void)loadView {
 }
 */

/*
 Implement viewDidLoad if you need to do additional setup after loading the
 view.
 - (void)viewDidLoad {
     [super viewDidLoad];
 }
 */

- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)➡
interfaceOrientation {
```

```

    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning]; // Releases the view if it doesn't
    have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [super dealloc];
}

@end

```

苹果公司已经预见到我们可能会覆盖的一些方法，并在实现文件中包含了方法桩。其中一些已经被注释掉，并且可以根据情况取消注释或删除。未注释掉的方法是由模板使用或者很常用的方法，因此被直接包含在代码中，以节省我们的时间。此应用程序不需要任何注释掉的方法，因此，请直接删除它们，这将缩短代码并且更便于理解我们插入的新代码。

删除注释掉的方法之后，添加以下代码。完成之后，让我们看看其作用：

```

#import "Button_FunViewController.h"

@implementation Button_FunViewController
@synthesize statusText;

- (IBAction)buttonPressed:(id)sender
{
    NSString *title = [sender titleForState:UIControlStateNormal];
    NSString *newText = [[NSString alloc] initWithFormat:
        @"%@ button pressed.", title];
    statusText.text = newText;
    [newText release];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning]; // Releases the view if it doesn't
    have a superview
}

```



```

    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [statusText release];
    [super dealloc];
}

@end

```

很好，我们看看新添加的代码。首先，我们添加了以下代码：

```
@synthesize statusText;
```

其作用是通知编译器自动为我们创建访问方法和修改方法。添加此行代码之后，我们的类中现已存在两个“不可见的”方法：`statusText`和`setStatusText:`。我们并没有编写这两个方法，它们是自动创建的，我们只需使用即可。

接下来的新增代码是在按下任意按钮时调用的操作方法的实现：

```

- (IBAction)buttonPressed: (id)sender
{
    NSString *title = [sender titleForState:UIControlStateNormal];
    NSString *newText = [[NSString alloc] initWithFormat:
        @"%@ button pressed.", title];
    statusText.text = newText;
    [newText release];
}

```

记住，传递到操作方法中的参数是调用它的控件或对象。因此，在我们的应用程序中，`sender`将始终指向被按下的按钮。这是一个非常便捷的机制，因为我们可以用一个操作方法处理多个控件的输入我们也正是这样做的：两个按钮调用此方法，并且我们通过查看`sender`参数来分辨它们。此方法的第一行代码通过`sender`获取被按下按钮的标题。

```
NSString *title = [sender titleForState:UIControlStateNormal];
```

说明 在请求按钮的标题时，我们需要提供控件状态。4种可能的状态分别为：正常（normal），表示控件处于活动状态，但当前并未使用；突出显示（highlighted），表示控件正被按住或正被使用；禁用（disabled），表示按钮未启用且无法使用；已选中（selected），仅特定控件具有该状态，表示控件当前已被选中。`UIControlStateNormal`表示控件的正常状态，并且是最常使用的一种状态。如果未指定其他状态的值，则这些状态将拥有与正常状态相同的值。

接下来，根据该标题创建一个新字符串：

```

NSString *newText = [[NSString alloc] initWithFormat:@"%@ button pressed.",
    title];

```

这个新字符串会将文本“button pressed.”附加在按钮名称后面。因此，如果我们按下标题为“Left”的按钮，则该字符串将等于“Left button pressed”。

最后，将标签的文本设置为此新字符串：

```
statusText.text = newText;
```

此处，我们使用点表示法来设置标签的文本，但我们也可以使用[statusText setText:newText];来代替。最后，释放字符串：

```
[newText release];
```

完成以后，释放对象的重要性就不再赘述了。iPhone是资源非常有限的设备，即使很小的内存泄漏也会造成程序崩溃。还应指出，我们并没有采用以下方式：

```
NSString *newText = [NSString stringWithFormat:@"%s@ button pressed.",  
title];
```

此代码与我们所使用的代码的作用完全相同。这种类方法称为简便（convenience）方法或工厂方法，并且它们将返回自动释放的对象。根据通用内存规则“如果未分配它或保留它，则不要释放它”，这些自动释放的对象不需要被释放，除非明确保留了它们，并且使用它们经常会让代码更加简短、更具可读性。

但是，这些简便方法也是有代价的，因为它们使用了自动释放池。分配给自动释放对象的内存，在对象使用完之后还会保留一段时间。在Mac OS X上，由于它有交换文件和相对较大的物理内存，因此使用自动释放对象的代价微乎其微。但是在iPhone上，这些对象会对应用程序的内存占用造成不利的影响。使用自动释放是可以的，但应该将它用到有用的地方，而不仅仅是为了少输入几行代码。

提示 如果你对objective-C内存管理尚存疑问，请查阅其内存管理“条约”：<http://developer.apple.com/documentation/Cocoa/Conceptual/MemoryMgmt/Tasks/MemoryManagementRules.html>。即使很小的内存泄漏也会对iPhone应用程序造成致命性的损害。

最后，释放dealloc方法中的输出口：

```
[statusText release];
```

释放此项目有点奇怪。你可能会想，既然我们没有实例化该对象，那么就不应该负责释放它。如果你使用过较早的Cocoa和Objective-C版本，则可能会认为这是错误的做法。但是，由于我们实现了各输出口的属性，并且为该属性的属性指定了retain参数，因为释放它是正确和必需的操作。Interface Builder在分配输出口时将使用生成的修改方法，并且该修改方法将保留分配给它的对象，因此在此处释放输出口以避免内存泄漏是非常重要的。

继续操作前，一定要保存文件，然后按⌘B编译该项目，确保没有输入错误。如果项目无法编译，请回到项目，并将其中的代码与本书中的代码进行对比。

消息嵌套

开发人员经常会嵌入一些Objective-C消息。你可以尝试编写以下代码：

```
statusText.text = [NSString stringWithFormat:@"%@" button pressed.",
    [sender titleForState:UIControlStateNormal]];
```

此行代码在功能上等价于buttonPressed:方法的4行代码。考虑到简洁性，我们通常不会在本书的代码示例中嵌入Objective-C消息，唯一的例外是alloc and init调用，它嵌入消息是形成已久的约定。

3.4 使用应用程序委托

Classes文件夹中的另外两个文件实现了应用程序委托。Cocoa Touch广泛使用了委托，它是负责为另一个对象处理特定事情的类。通过应用程序委托，我们可以在预先定义的时间为UIApplication处理事情。每个iPhone应用程序都有一个且仅有一个UIApplication实例，负责应用程序的运行循环以及处理各种应用程序级功能，如将输入发送给合适的控制器类。

UIApplication是UIKit的标准部分，它主要在后台处理任务，因此一般不需要担心它。但是，在应用程序执行过程中明确指定的时间，UIApplication将调用特定的委托方法（如果有委托且实现了该方法）。例如，如果需要在程序退出之前触发某段代码，可以在应用程序委托中实现applicationWillTerminate:方法，并将终止代码置于其中。这种委托可以让应用程序实现常用的应用程序级行为，而不需要继承UIApplication，或了解它的任何内部机制。

单击Groups & Files窗格中的Button_FunAppDelegate.h，查看应用程序委托的头文件。它应如下所示：

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

@class Button_FunViewController;

@interface Button_FunAppDelegate : NSObject <UIApplicationDelegate> {
    IBOutlet UIWindow *window;
    IBOutlet Button_FunViewController *viewController;
}

@property (nonatomic, retain) UIWindow *window;
@property (nonatomic, retain) Button_FunViewController *viewController;

@end
```

我们不需要对此文件做任何修改，并且在实现控制器类之后，你应该已经对此处的大部分代码都非常熟悉。需要指出的一处代码是：

```
@interface Button_FunAppDelegate : NSObject <UIApplicationDelegate> {
```

注意到尖括号里面的值了吗？它表示此类符合UIApplicationDelegate协议。按下option键并将光标移动到UIApplicationDelegate上。光标应转变为十字线，然后，双击鼠标按钮。这将打开文

档浏览器并显示UIApplicationDelegate协议的文档（参见图3-3）。此技巧同样适用于类、协议和类别名称，以及编辑器窗格中显示的方法名称。按住option键并双击某个单词可以在文档浏览器中搜索该单词。

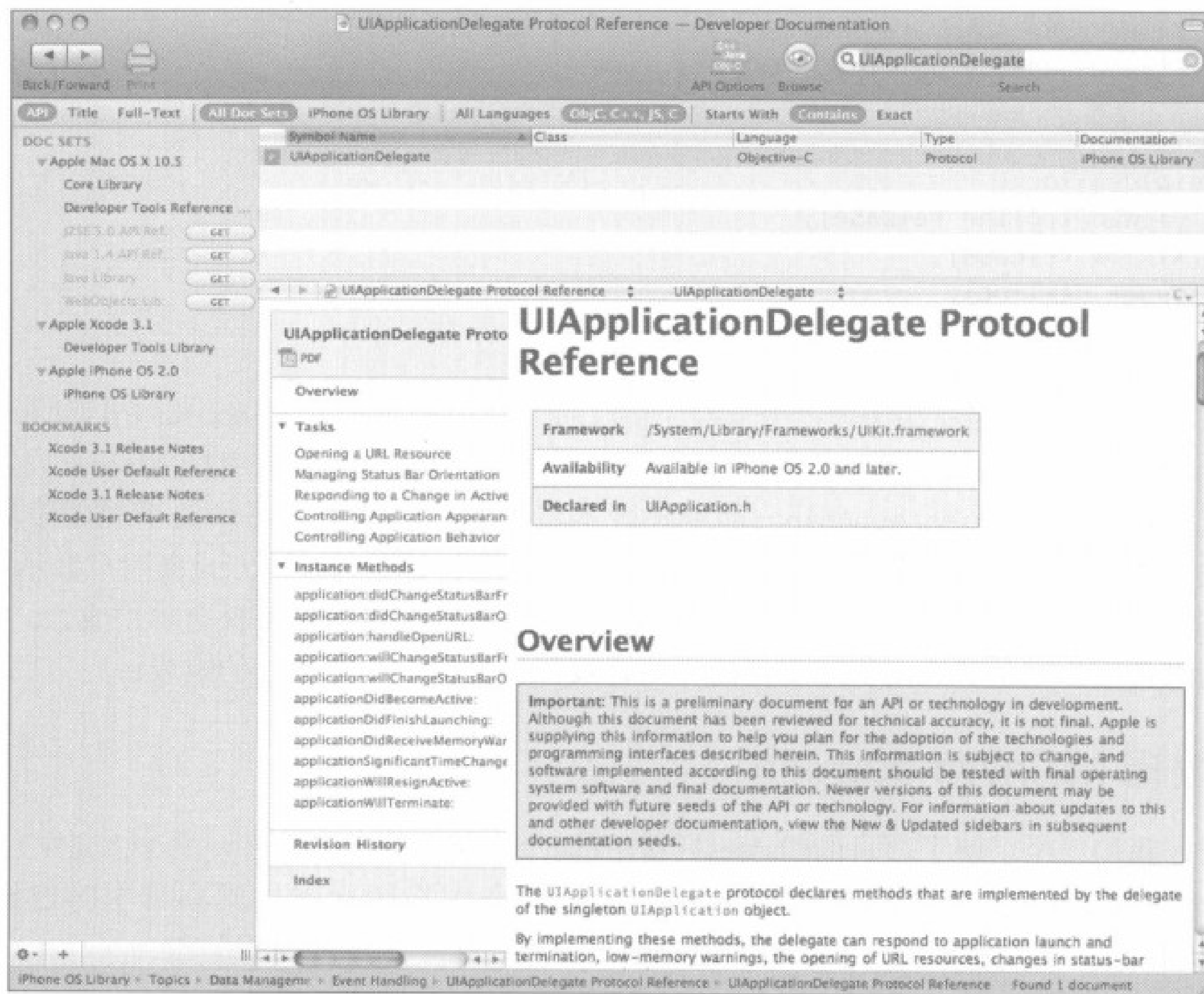


图3-3 使用文档浏览器查看UIApplicationDelegate文档

了解如何快速查找文档中的内容绝对是大有裨益的，但是查看此协议的定义可能更重要。你可以从中了解应用程序委托能实现哪些方法，以及这些方法将在何时被调用。花些时间阅读这些方法的说明是值得的。

说明 如果之前使用过Objective-C，但没用过Objective-C 2.0，则应该知道协议现在可以指定可选方法。UIApplicationDelegate可以包含许多可选方法，并且不需要在应用程序委托中实现任何可选方法，除非有特殊原因。

单击Button_FunAppDelegate.m，查看应用程序委托的实现。它应如下所示：

```
#import "Button_FunAppDelegate.h"
#import "Button_FunViewController.h"

@implementation Button_FunAppDelegate
```

```

@synthesize window;
@synthesize viewController;

- (void)applicationDidFinishLaunching:(UIApplication *)application {
    // Override point for customization after app launch
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];
}

- (void)dealloc {
    [viewController release];
    [window release];
    [super dealloc];
}

@end

```

在该文件中, 可以看到应用程序委托实现了协议的一个方法: `applicationDidFinishLaunching:`。你可能已经猜到, 该方法将在应用程序完成所有设置任务, 并准备好开始与用户交互时触发。

`applicationDidFinishLaunching:`的委托版本将视图控制器的视图作为子视图添加到应用程序的主窗口中, 并将窗口设置为可见, 我们设计的视图就是通过这种方式向用户显示的。你不需要通过任何操作来实现此目的; 所有代码都将由构建项目所使用的模板自动生成。

我们只希望让你了解应用程序委托的一些背景知识, 以及其各部分的组成。

3.5 编辑 MainWindow.xib

目前, 我们已经了解了项目的Classes标签中的4个文件(两个.m文件, 两个.h文件)。我们还将探索Resources标签中的3个文件之一。我们在第2章中将图标添加到项目中时已经了解了Info.plist。稍后, 我们将与上一章一样, 在Interface Builder中编辑视图控制器。

Resources 标签中还有另外一个文件需要讨论。`MainWindow.xib`文件的作用是让应用程序委托、主窗口和视图控制器实例在运行时创建。记住, 此文件是作为项目模板的一部分提供的。此处, 你不需要做任何修改, 或进行任何操作。你只需借此机会了解其内部机理, 从而对其有总体的把握。

展开Xcode的Groups & Files窗格中的Resources文件夹, 然后双击`MainWindow.xib`。打开Interface Builder之后, 注意nib的`MainWindow.xib`主窗口, 该窗口应如图3-4所示。

你应该记得此窗口中的前两个图标是在第2章中创建的。`nib`窗口中位于这两个图标后面的所有其他图标, 都表示将在`nib`文件加载时进行实例化的对象。我们来看第3、第4和第5个图标。



图3-4 应用程序的`MainWindow.xib`出现在Interface Builder中

说明 在默认视图中，nib文件主窗口中的长名称将被截短，如图3-4所示。如果将光标移动到这些图标上，并停留几秒，将会弹出一个工具提示，显示该项的全名。还需注意，主窗口中的名称不一定表示对象的底层类。新实例的默认名称通常会让你想到底层类，但这些名称可以（且经常）被修改。

第3个图标是Button_FunAppDelegate的一个实例。第4个图标是应用程序唯一的窗口（UIWindow的实例）。最后，第5个图标是Button_FunViewController的一个实例。这3个图标表示，当nib文件加载时，应用程序将拥有应用程序委托Button_FunAppDelegate的一个实例、UIWindow的一个实例（表示应用程序的唯一窗口的类），以及视图控制器Button_FunViewController的一个实例。不难看出，Interface Builder的功能远不止创建界面元素。你可以使用它创建任何其他类的实例。这是一个非常强大的特性。每少写一行代码，同时也就少了一行需要调试和维护的代码。现在，我们已经在启动时创建了3个对象实例，而没有编写一行代码。

很好，本节的任务已经完成；请继续学习。确保在退出时关闭此nib文件。如果弹出保存文件的提示，请选择“否”，因为你不应该修改任何内容。

3.6 编辑 Button_FunViewController.xib

现在，你已经了解了组成项目的文件，以及它们拼合在一起的原理。现在我们将注意力转向Interface Builder以及构建界面的过程。

3.6.1 在 Interface Builder 中创建视图

在Xcode中，双击Groups & Files窗格中的Button_FunViewController.xib。nib文件应在Interface Builder中打开。请确保库是可见的。如果不可见，可以通过从Tools菜单中选择Library来显示它。还需要确保nib的View窗口是打开的。如果未打开，就双击nib主窗口中的View图标（参见图3-5）。

现在，可以开始设计界面了。从库中拖动一个标签到视图窗口中，这与上一章相同。将标签放置在视图的底部，展开它，以占用视图的大部分宽度。使用弹出的蓝色引导线来帮助放置标签（参见图3-6）。

说明 蓝色引导线可以帮助你依照Apple Human Interface Guidelines（通常简称为HIG）来设计界面。与Mac OS X类似，苹果公司提供了iPhone Human Interface Guidelines来设计iPhone应用程序。HIG将指导你应（或不应）如何设计用户界面，你真的应该读读它，因为其中包含了iPhone开发人员都需要了解的宝贵信息。其地址为：<http://developer.apple.com/iphone/library/documentation/UserExperience/Conceptual/MobileHIG/>。

将标签放置在视图底部之后，单击以选中它，并按下⌘I调出检查器。使用检查器上的文本对齐按钮将文本更改为中间对齐（参见图3-7）。

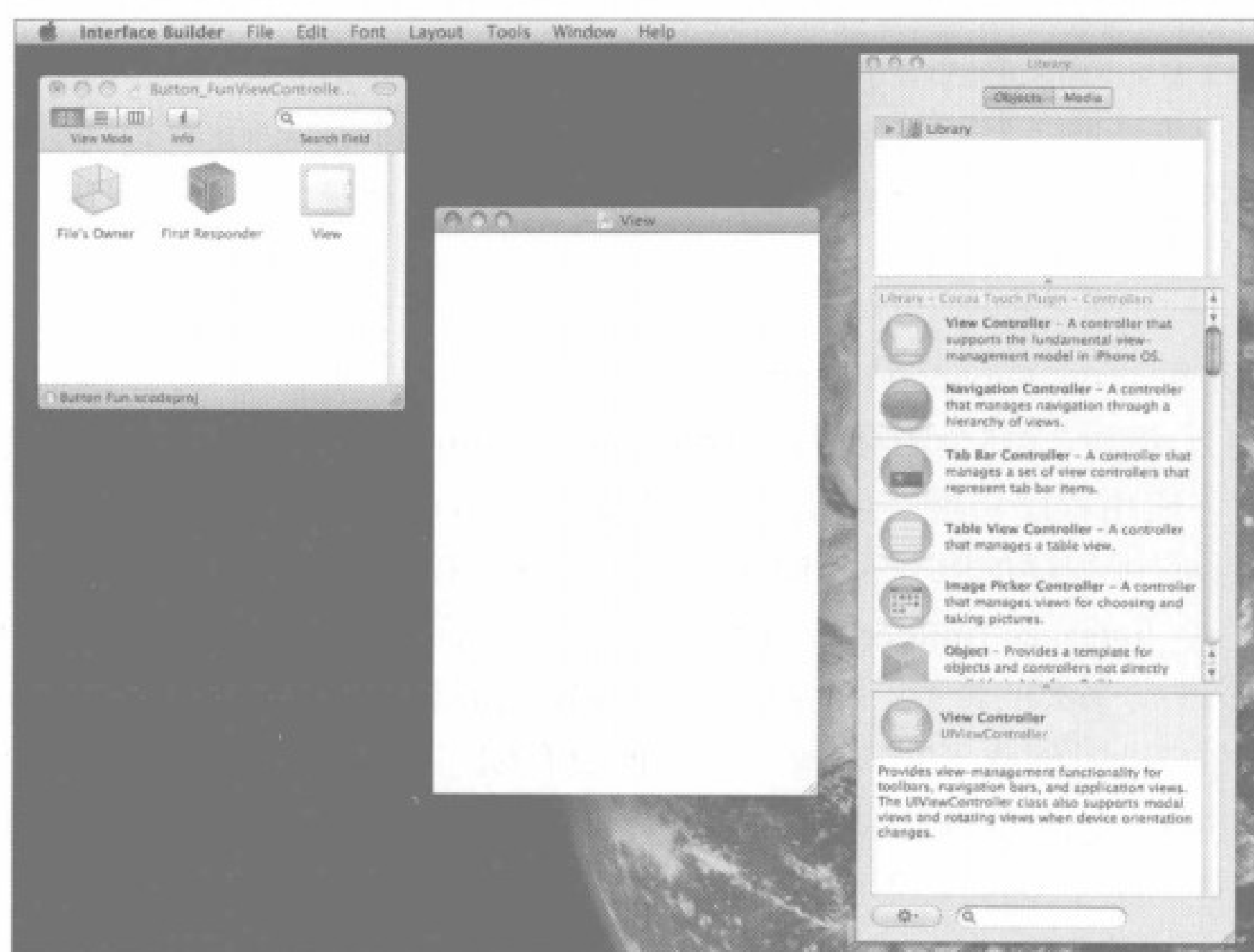


图3-5 Interface Builder已打开Button_FunViewController.xib

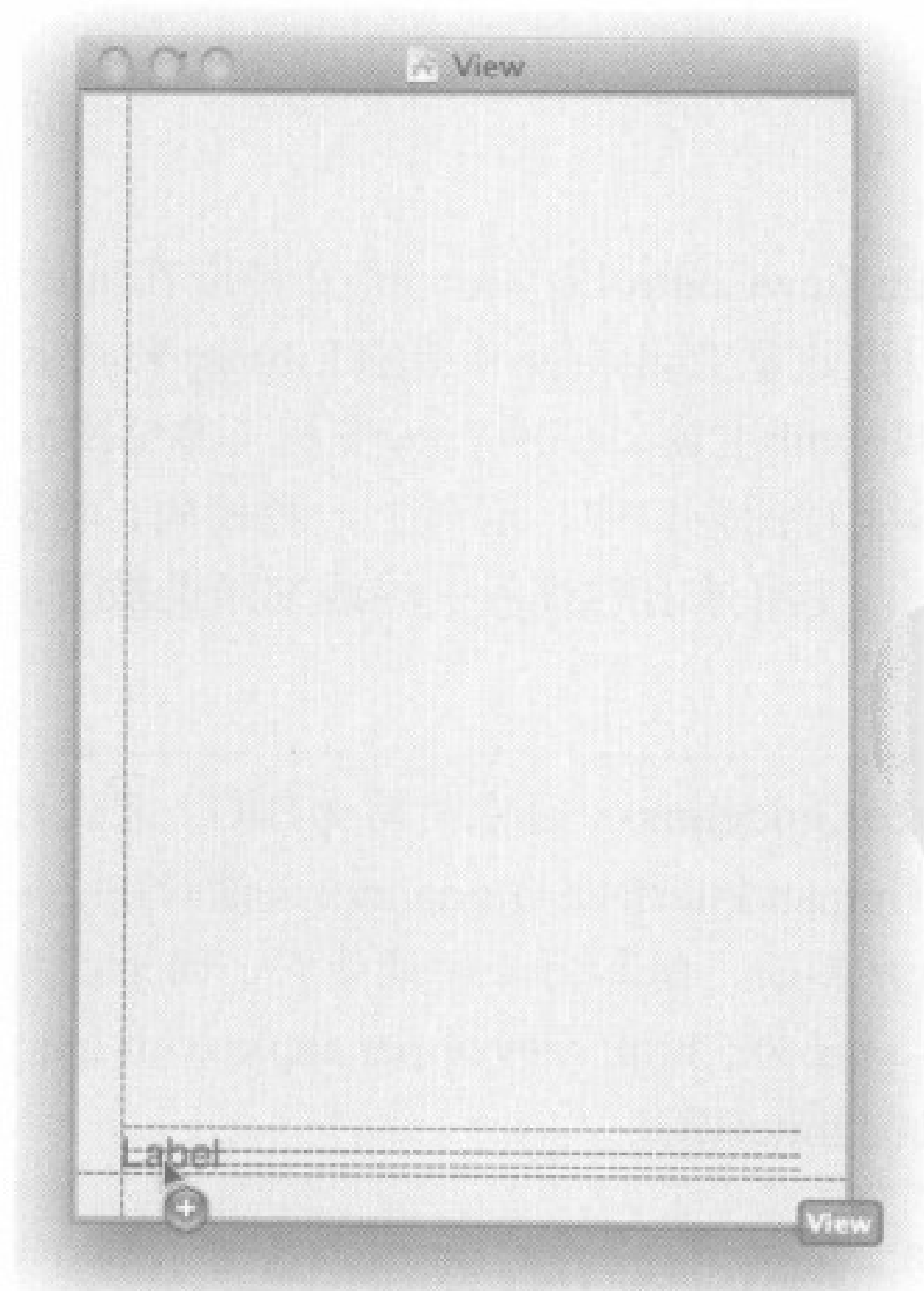


图3-6 使用蓝色引导线放置对象

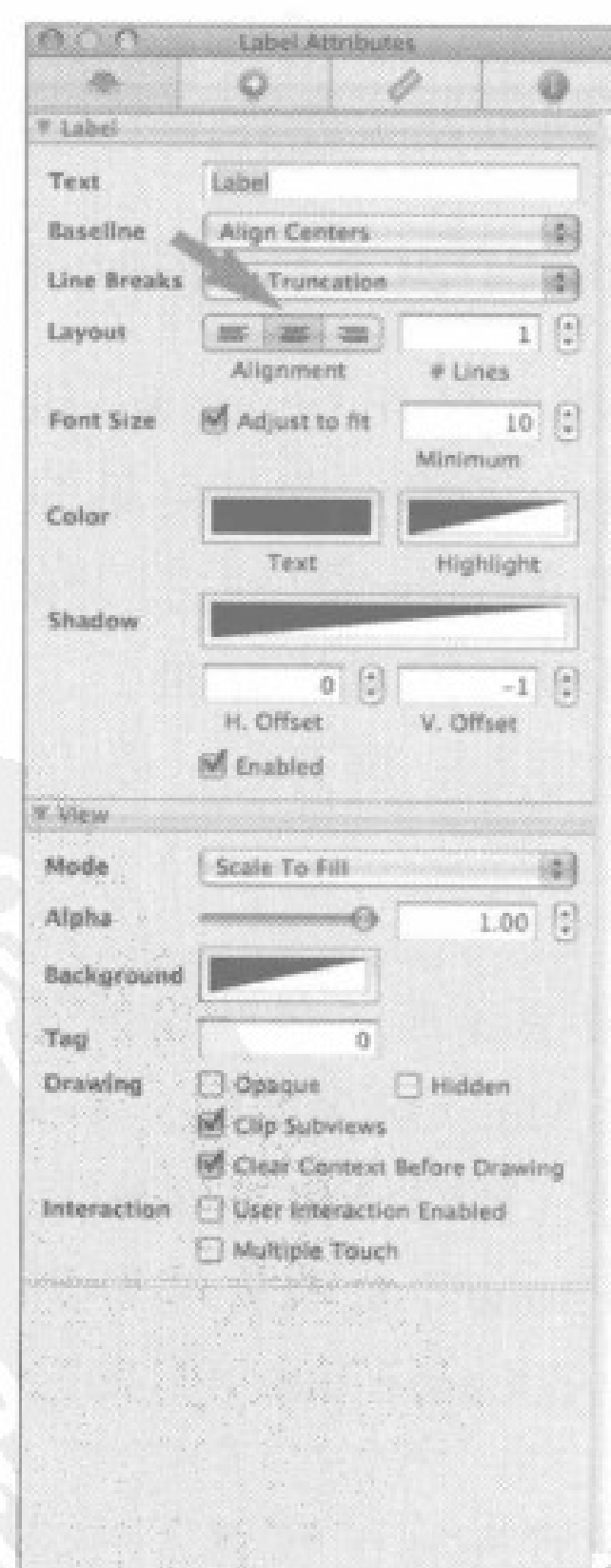


图3-7 检查器的文本对齐按钮

将标签放置到所需位置之后，双击标签并删除已有文本。我们不想在单击按钮之前显示任何文本。

接下来，我们将从库中拖动两个Round Rect Button到视图中（参见图3-8）。

将两个按钮并排排列，放置在视图的中间。其位置是否准确无关紧要。双击左侧的按钮。可以通过此操作编辑按钮的标题，将其文本修改为“Left”。接下来，双击右侧的按钮，将其文本修改为“Right”。完成之后，视图应如图3-9所示。

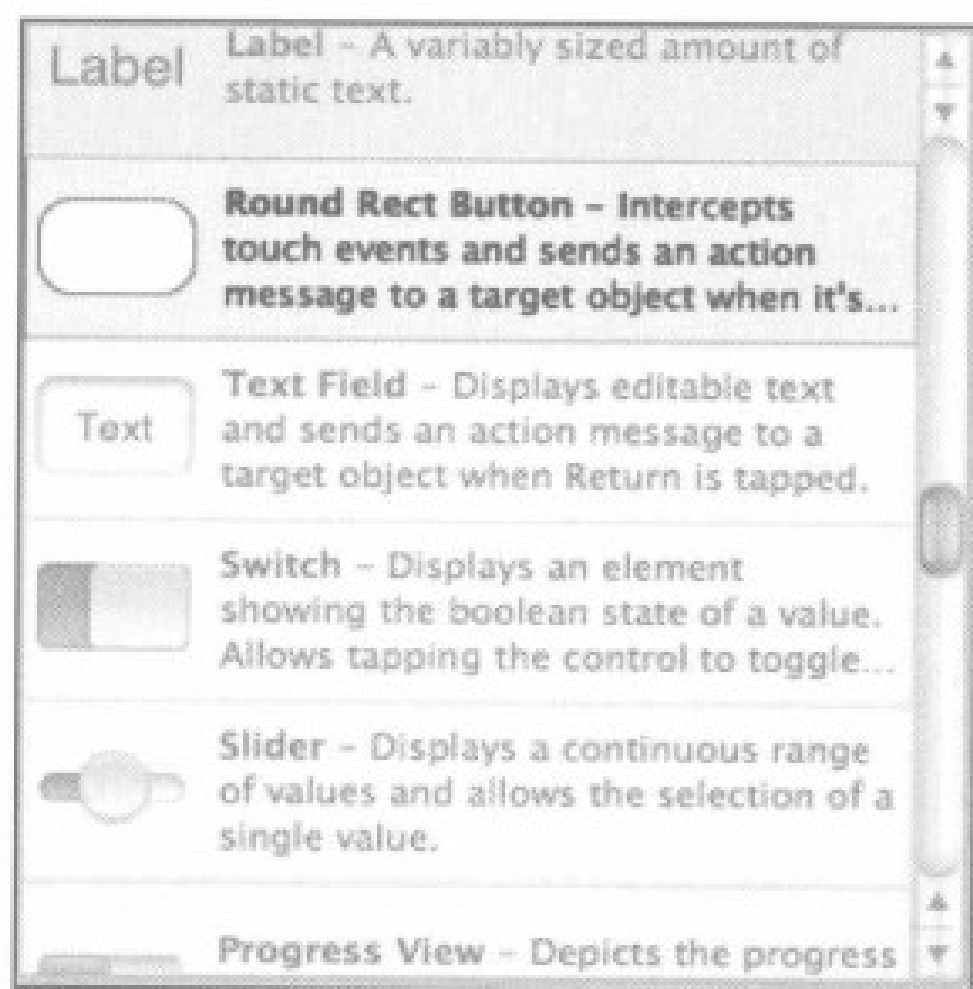


图3-8 库中的Round Rect Button



图3-9 完成后的视图

3.6.2 连接所有元素

现在，我们已经创建了界面的所有元素。接下来需要将这些元素连接起来，让它们形成一个整体。

第一步是从File's Owner连接到View窗口中的标签。为何是File's Owner呢？

当UIViewController的某个实例或某个子类被实例化时，它将查找相应的nib文件。在我们所使用的模板中，MyViewController将尝试加载名为MyViewController.xib的nib，后者将由我们选择的项目模板来创建。如果它找到此nib，则将其加载到内存中并成为该nib文件的所有者。由于MainWindow.xib文件包含一个表示Button_FunViewController的图标，因此应用程序在启动时将自动创建一个Button_FunViewController实例。此后，该实例会自动将Button_FunViewController.xib加载到内存中并成为该文件的所有者。

在本章前面部分，我们在Button_FunViewController（此nib文件的所有者）中添加了一个输出口。现在，我们可以使用File's Owner图标将该输出口与标签连接起来。接下来讨论连接的方法。

说明 如果还没有完全理解nib加载过程，也没有关系。这是一个复杂的过程，稍后几章将通过演示来深入讨论此过程。现在，只需记住控制器类是与之同名的nib文件的所有者。

1. 连接输出口

按下Control键，单击nib主窗口中的File's Owner图标，并按住鼠标按钮，从File's Owner图标拖向View窗口。此时应出现一条蓝色引导线。继续拖动鼠标，直到光标位于View窗口的标签之上。尽管看不到标签，但是当你将光标移到其上方时，它就会显示出来（参见图3-10）。

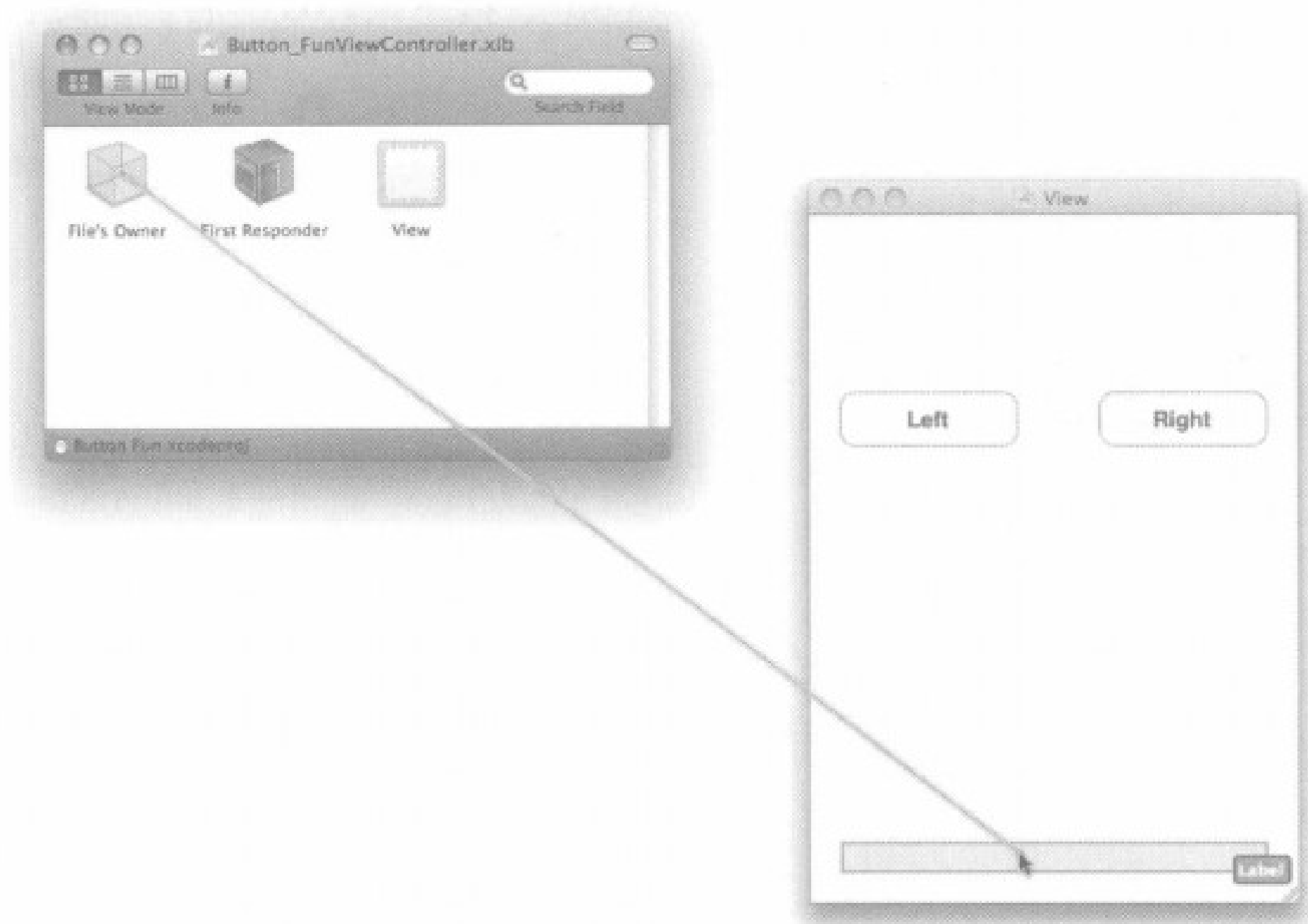


图3-10 通过拖动操作连接输出口

将光标放置在标签上不要动，放开鼠标按钮，此时应弹出如图3-11所示的灰色小菜单。

在灰色菜单中选中StatusText。

按住Control键并从File's Owner拖动到界面对象，这样，Interface Builder将知道我们希望在载入nib文件时将其中一个File's Owner输出口连接到此对象。在本例中，文件所有者是Button_FunViewController类，且我们感兴趣的Button_FunViewController是statusText。按住Control并从File's Owner拖动到标签对象，然后从弹出菜单中选择statusText，Interface Builder会将Button_FunViewContr的statusText输出口指向标签。因此在代码中引用statusText时，它表示的是此标签。

2. 指定操作

接下来唯一要做的就是确定这些按钮将触发哪些操作，以及是在何种情况下触发它们的。如果熟悉Mac OS X上的



图3-11 输出口选择菜单

Cocoa编程,那么可以采用按住Control键从按钮拖动到File's Owner图标的方式。说实话,这尽管是一种可行的方法,但它对于iPhone并不是最佳方法。

iPhone与Mac OS X之间的差异在此处比较明显。在Mac上,控件可以只与一个操作相关,并且此操作通常在使用该控件时触发。这也存在一些例外,但基本而言,控件将在释放鼠标按钮(如果光标仍然处于控件内部)时触发其相应的操作方法。

Cocoa Touch中的控件可以实现许多功能,因此使用连接检查器而不是点击控件后拖动是最好的方法,可以按下⌘2或从Tools菜单中选择Connection Inspector来打开连接检查器。单击Left按钮,然后调出连接检查器。它应如图3-12所示。

在Events栏下,可以看到能够潜在地触发操作的所有事件列表。如果愿意,你可以将不同的操作与不同的事件关联在一起。举例来说,可以使用Touch Up Inside来触发某操作,而使用Touch Drag Inside触发不同的操作。我们的情况相对比较简单和直接。当用户按下按钮时,我们希望它调用buttonPressed:方法。第一个问题是,应该使用图3-12中的哪个事件?

答案是Touch Up Inside。当用户的手指离开屏幕时,如果他最后触摸的位置在按钮内部,则用户将触发一个Touch Up Inside事件。试想,在大多数iPhone应用程序中,如果在触摸屏幕时改变了主意,你通常会把手指从按钮处移开,然后再离开屏幕,对吗?我们应该让用户具备相同的能力。如果用户的手指在离开屏幕之前仍然位于按钮上,那么可以放心地假设该按钮单击操作是故意的。

知道触发操作的事件之后,应该如何将事件与具体的操作方法相关联呢?

看到检查器中位于Touch Up Inside右侧的小圆圈了吗?单击该圆圈并拖动鼠标,这次不需要按下Control键。应该可以看到一条蓝色引导线,这与前面连接输出口时的情况一样。将此引导线拖动到File's Owner图标上,从弹出的小灰色菜单中选择buttonPressed:。记住,File's Owner图标代表的是我们正在编辑的nib文件的类。在本例中,File's Owner表示Button_FunViewController类。当我们从选定按钮的事件拖动到File's Owner图标之后,Interface Builder会知道在指定事件发生时调用选定方法。因此,当用户的手指离开按钮时,将调用Button_FunViewController类的buttonPressed:方法。

对其他按钮执行与此相同的步骤,然后保存。现在,当用户单击其中任意按钮时,都将调用buttonPressed:方法。

3.6.3 测试

保存nib文件之后,返回Xcode并运行应用程序。从Build菜单中选择Build and Run。Xcode将编译程序,并且应用程序将出现在iPhone仿真器中。单击左侧按钮时,应出现文本“Left button

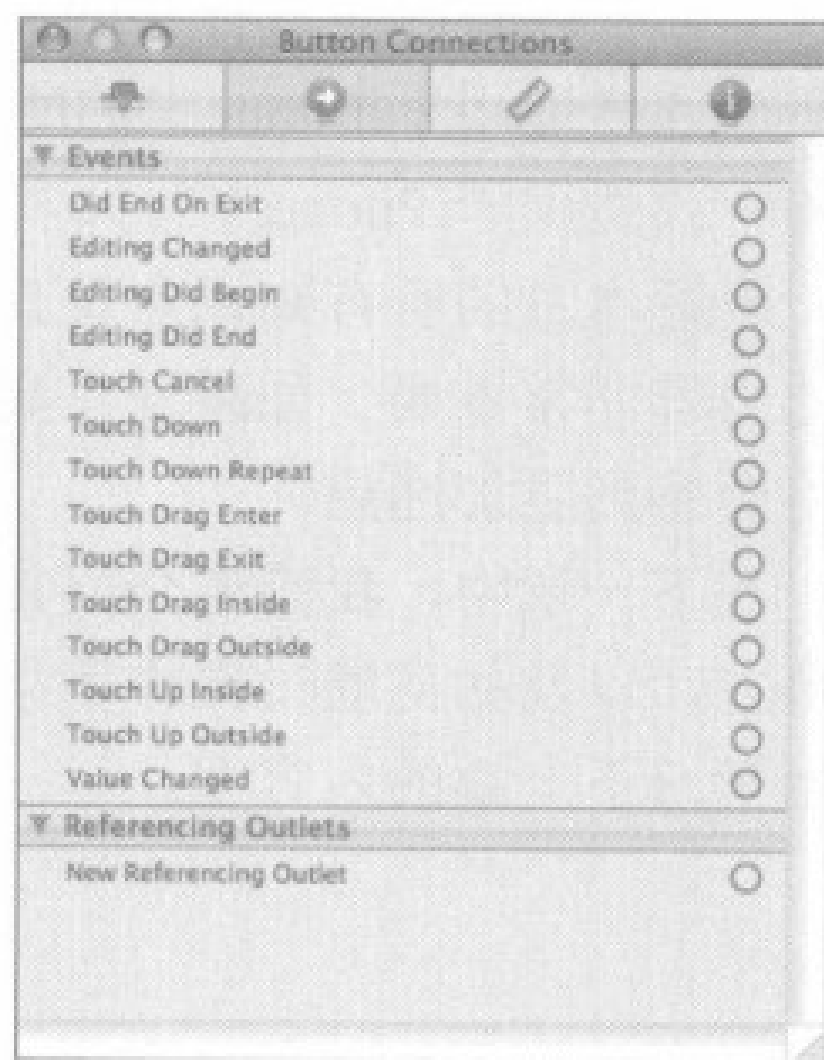


图3-12 显示按钮可用事件的连接检查器

pressed.”，如图3-1所示。随后，如果单击右键按钮，标签将变更为“Right button pressed.”。

3.7 小结

在本章中，我们通过一个简单的应用程序介绍了MVC，并学习了如何创建和连接输出口及操作，实现视图控制器，以及使用应用程序委托。我们还学习了如何在按下按钮时触发操作方法，以及如何在运行时更改标签的文本。虽然这是一个简单的应用程序，但我们在构建过程中所使用的概念同样适用于iPhone中的所有控件，而不仅仅是按钮。事实上，我们在本章中使用按钮和标签的方式适用于iPhone中的大多数标准控件。

理解本章的所有知识点及其原理非常重要。对于没有完全理解的部分，请回头重新阅读。本章的内容非常重要！如果现在尚未完全理解本章内容，那么在本书稍后创建比较复杂的界面时，你将会感到更加迷惑。

在下一章中，我们将介绍其他一些标准iPhone控件。你还将了解如何使用报警向用户通知重要事宜，以及如何通过操作表指示用户需要在继续之前做出选择。做好准备之后，深吸一口气，开始学习下一章的内容吧。



第3章讨论了模型-视图-控制器的概念，并依照它构建了一个实际的应用程序。我们学习了输出口和操作，并使用它们将按钮控件与文本标签绑定在一起。本章将构建一个更加复杂的应用程序，让你对控件的理解进入一个全新的层次。

我们将实现一个图像视图、一个滑块、两个不同的文本字段、一个分段控件、一些开关和一个更符合iPhone风格的按钮。你将学习如何使用视图层次结构对公共父视图下的多个项目进行分组，以及如何在运行时更加轻松地操作界面。你将了解如何设置和检索各种控件中的值，这可以使用输出口或使用操作方法的sender参数来完成。然后，我们将介绍如何使用操作表强制用户做出选择，并向用户显示重要反馈。我们还将介绍控件状态，以及如何使用可拉伸图像让按钮的显示效果更加美观。

由于本章的应用程序将使用大量不同的用户界面项，因此我们的操作方式将与前两章有所不同。我们会将应用程序分成若干个小块，每次实现其中的一块，你将往返于Xcode、Interface Builder和iPhone仿真器之间，每完成一块都要进行测试，然后再继续开发。将构建复杂界面的过程划分为小块，这样可以简化它，并且更加接近于实际的应用程序构建流程。这种“编码-编译-调试”周期是软件开发人员日常工作的主要组成部分。

4.1 满是控件的屏幕

前面已经提到，本章将构建的应用程序要比第3章稍微复杂一些。我们仍然使用一个视图和控制器，但这个视图中的元素将更加丰富，如图4-1所示。

iPhone屏幕顶部的徽标是一个**图像视图**。并且在此应用程序中，它的作用仅仅是显示一个静态图像。徽标下方是两个文本字段，一个允许输入字母和数字形式的文本，另一个只允许输入数字。位于文本字段下方的是一个**滑块**。当用户更改滑块时，其左侧标签的值将会随之更改，它反映了滑块的值。

滑块下方是一个分段控件和两个开关。分段控件将根据选中的是Show还是Hide来显示或隐藏两个开关。图4-2显示了用户单击Hide时发生的情况。更改任一开关的值都会导致另一个开关更改其值，以与之匹配。现在，这并不是你希望在真实应用程序中出现的，但它将演示如何通过代码更改控件的值，以及Cocoa Touch如何实现特定操作的动画，而你不需要做任何工作。

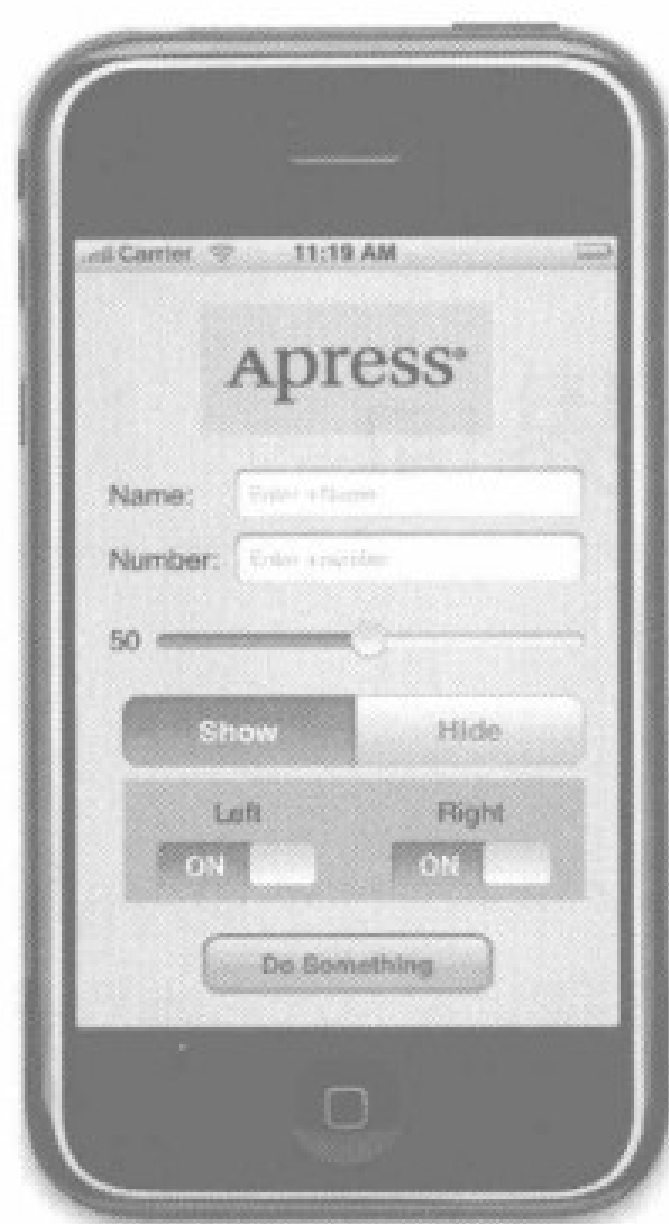


图4-1 Control Fun应用程序，包含文本字段、标签、滑块和若干其他常用的iPhone控件

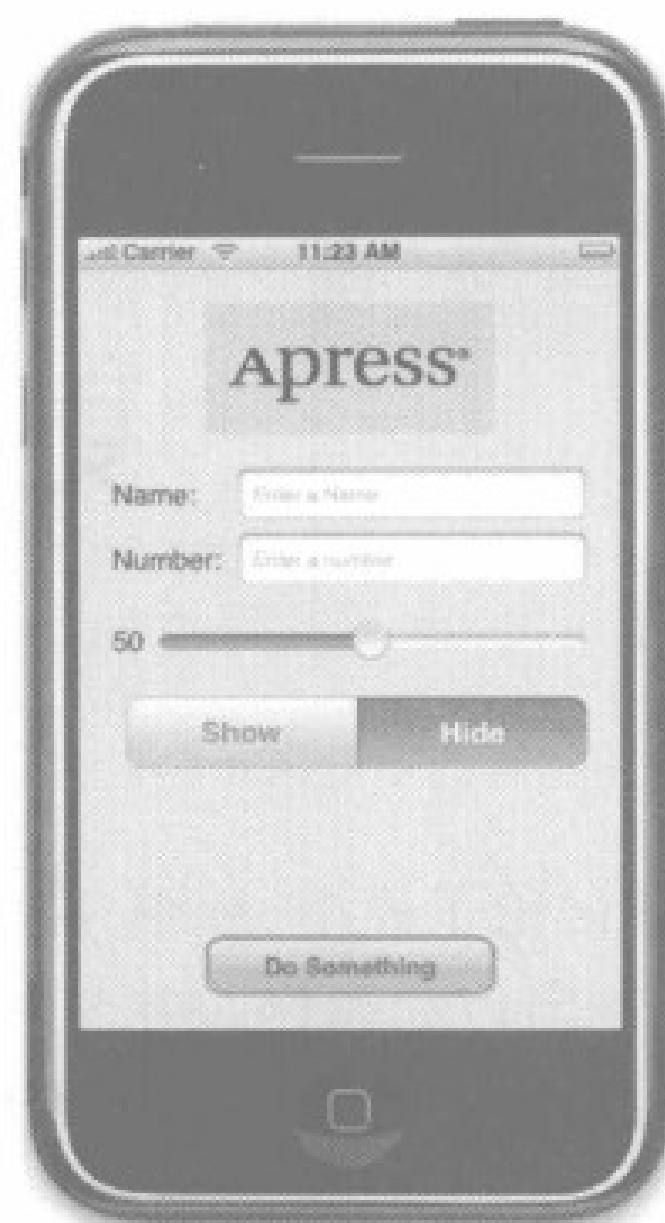


图4-2 使用分段控制器时，开关可以在隐藏和显示之间来回切换

开关下方是一个Do Something按钮，该按钮用于弹出一个操作表，询问用户是否确定要单击按钮（参见图4-3）。这是响应具有潜在危险或导致严重后果的输入的标准方式，可使用户将潜在危险拒之门外。

如果选择“*Yes, I'm Sure!*”，应用程序将使用警报通知用户是否一切正常（参见图4-4）。

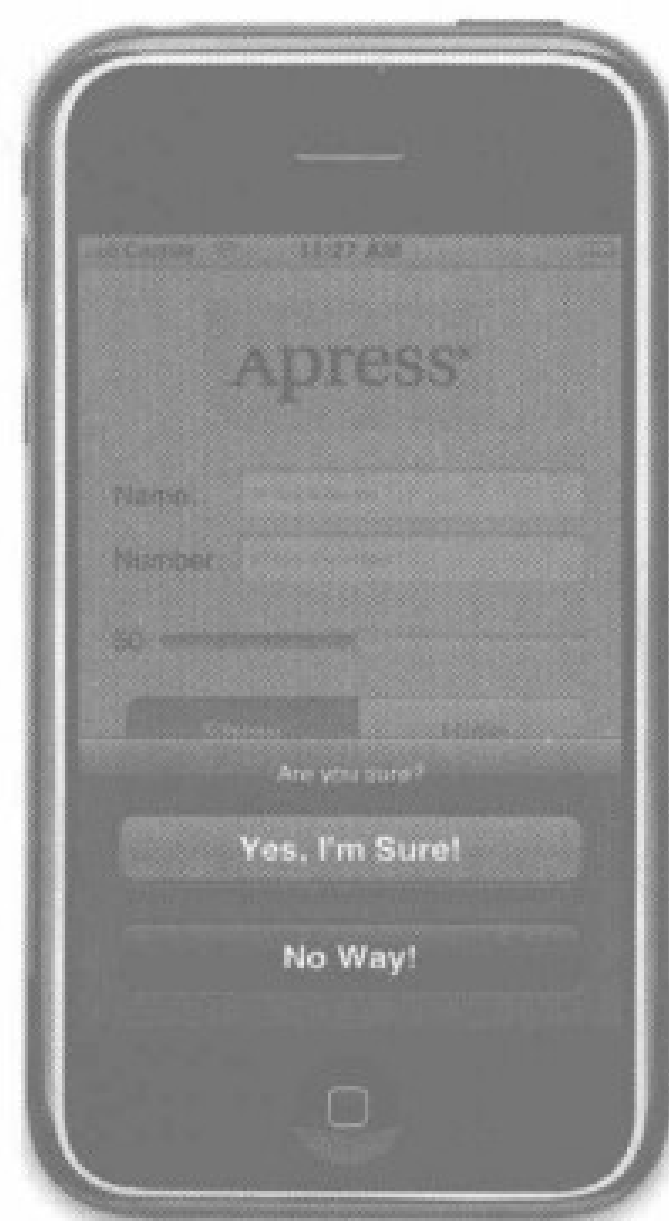


图4-3 应用程序使用操作表请求用户回应

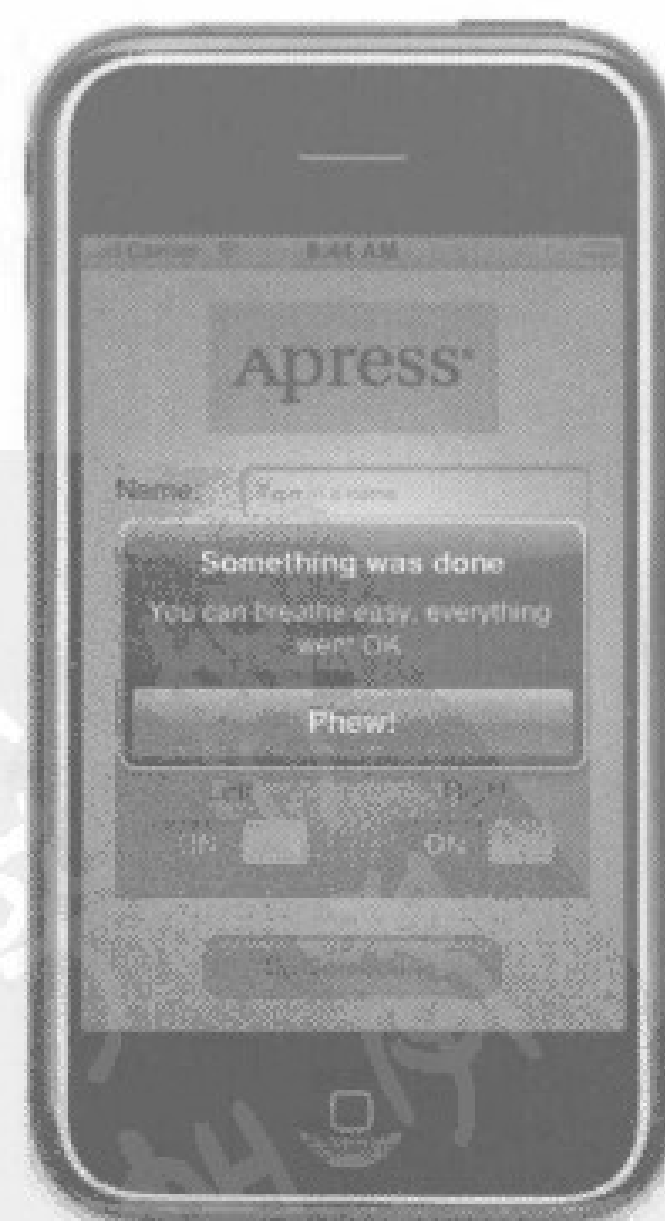


图4-4 使用警报向用户通知重要事件。此处使用的警报用于确认一切是否正常

4.2 活动、静态和被动控件

用户界面控件共有3种基本形式：活动、静态（非活动）和被动。上一章所使用的按钮都是典型的活动控件。单击按钮便会发生一些事情——通常是触发一段代码。虽然我们将使用的许多控件都将直接触发操作方法，但并非所有控件都是如此。

上一章所使用的标签是静态控件的一个很好的例子。我们将它添加到界面中，并通过代码修改它，但用户不能对其进行任何操作。标签和图像经常都采用这种方式，但这两者都是UIControl的子类，并且可以用它们触发特定的代码。

一些控件可以在被动方式下运行，仅用于存储用户输入的值，直到用户完成为止。这些控件不触发任何操作方法，但用户可以与之交互，并修改它们的值。

被动控件的典型例子是网页上的文本字段。虽然可以在移出字段时触发验证代码，但网页上的大多数文本字段都只是用作保存数据的容器，这些数据在用户单击提交按钮时被提交给服务器。文本字段自身不触发任何代码，但是单击提交按钮时，文本字段的数据将可以传递。

在iPhone上，许多可用控件都可以通过这3种方式加以使用，并且大多数控件的功能都不是唯一的，这根据开发人员的需要而定。所有iPhone控件都是UIControl的子类，因此它们能够触发操作方法。大多数控件还可以被动使用，并且它们都可以在创建时被设定为非活动，或者在运行时从活动更改为非活动（或反之，从非活动更改为活动）。例如，通过一个控件可以将另一个控件从非活动改为活动。但是，包括按钮在内的一些控件，除了在活动方式下用来触发代码以外，实际上并没有其他用途。

iPhone和Mac上的控件在行为上存在一些差异。下面给出了一些例子。由于多点触摸界面的引入，所有iPhone控件都可以根据其接触方式触发多种操作：用户可以通过触碰按钮来触发一个操作，而通过滑触动作来触发不同的操作。你还可以让用户按下按钮时触发一个操作，当用户手指离开按钮时触发另一个操作。相反，也可以让单个控件对单一事件调用多个操作方法。可以让Touch Up Inside事件触发两个不同的操作方法，这意味着，当用户的手指离开按钮时将调用两个方法。

iPhone与Mac之间的另一个主要区别是，iPhone没有物理键盘。iPhone键盘实际上是一个满是按钮控件的视图。代码可能永远都不会直接与iPhone键盘交互，但有时需要编写代码让键盘行为符合开发人员的需求，如本章稍后所示。

4.3 创建应用程序

如果未打开Xcode，请打开它，然后创建一个名称为Control Fun的新项目。我们将再次使用View-Based Application模板选项，仍按照前两章的方法创建项目。

4.3.1 导入图像

创建项目之后，找到将在图像视图中使用的图像。需要将图像导入到Xcode中，然后才能在

Interface Builder内部使用，因此先导入图像。你可以在04 Control Fun目录的项目归档中找到一个符合条件的.png图像，或者可以使用你自己的图像——确保所选图像为.png格式，且大小不超过可用的空间。图像的高度应小于100像素，宽度应小于300像素，以适应视图布局，而不需要重新调整大小。

将图像添加到项目的Resources文件夹，这与第2章的操作相同。可以将图像从Finder拖动到Resources文件夹，或者从Project菜单中选择Add to Project来完成这个操作。

4.3.2 实现图像视图和文本字段

将图像添加到项目之后，接下来需实现应用程序屏幕顶部的5个界面元素：图像视图、两个文本字段和两个标签（参见图4-5）。

1. 确定输出口

在使用Interface Builder之前，我们需要确定哪些对象需要输出口。记住，需要在控制器类的头文件中定义输出口，然后才能将它们连接到Interface Builder中的任何对象。

图像视图只是一个静态图像。我们将指定图像在Interface Builder中显示，并且图像不会随着应用程序的运行发生更改。因此，它不需要输出口。如果我们希望在运行时更改图像或更改其属性，则需要一个输出口。本例并不属于这种情况。

对于两个标签也是如此。它们仅用于显示文本，并不会在运行时发生更改。并且用户不会与它们进行交互，因此我们不需要为它们指定输出口。

另一方面，如果无法访问两个文本字段包含的数据，那么它们的实际作用也不大。访问被动控件保存的数据的方法是使用输出口。因此我们需要为这两个文本字段分别定义一个输出口。你现在已经熟悉了此操作，请自行添加两个输出口（名称分别为nameField和numberField）及其相应属性到Control_FunViewController.h类文件中。完成之后的代码应如下所示：

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

```
@interface Control_FunViewController : UIViewController {
    IBOutlet UITextField *nameField;
    IBOutlet UITextField *numberField;
}
@property (nonatomic, retain) UITextField *nameField;
@property (nonatomic, retain) UITextField *numberField;
@end
```

在转向使用Interface Builder之前，还需将@synthesize指令添加到Control_FunViewController.m中：

```
#import "Control_FunViewController.h"
```

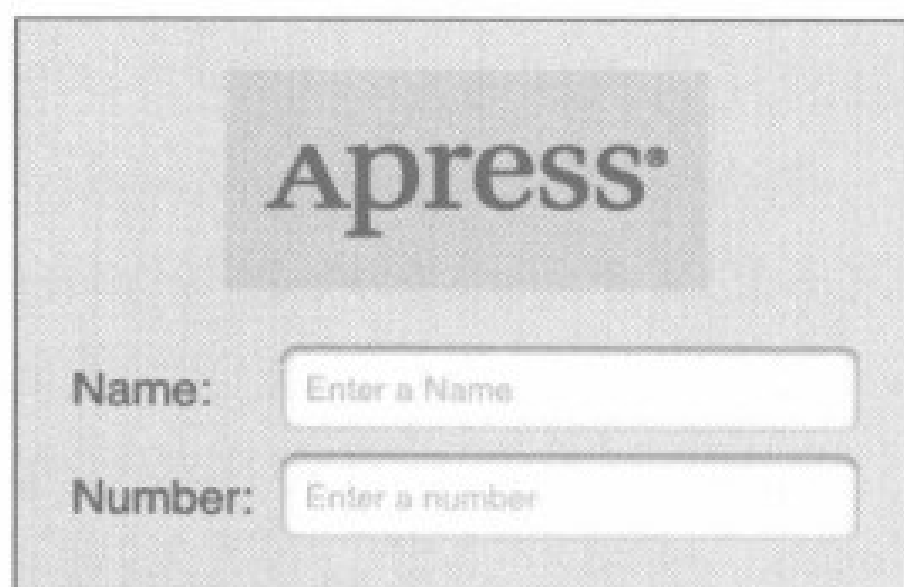


图4-5 首先将实现的图像视图、标签和文本字段

```
@implementation Control_FunViewController
@synthesize nameField;
@synthesize numberField;
...
```

说明 注意到代码清单尾部的省略号 (...) 了吗？我们将使用该符号指示没显示出来的其他不需要修改的代码。在本章中，我们会将所有代码添加到实现文件的顶部，因此使用省略号，而不是每次添加一行或两行代码都要显示整个文件。

2. 确定操作

回顾一下图4-5中的5个对象。是否发现需要声明某个操作呢？图像视图和标签不支持与用户交互，不能接受触摸，因此不需要为它们指定操作，对吗？没错，就是这样。

那么两个文本字段也是这样吗？文本字段是典型的被动控件。大多数情况下，它们只是为用户保存一些值。除了限制数字字段的输入（仅显示数字键，而不是完整的键盘，这完全可以通过Interface Builder来实现）以外，我们不会对这些字段执行任何验证，因此不需要为它们指定操作。现在，让我们来构建并测试用户界面的第一部分。

3. 构建界面

确保两个文件都已保存，展开Groups & Files窗格中的Resources文件夹，然后双击Control_FunViewController.xib来启动Interface Builder。如果View窗口未打开，双击nib文件主窗口中的View图标。

现在，将注意力转向库。如果它未打开，请从Tools菜单中选择Library。拖动滑块，在接近列表的四分之一处找到Image View（参见图4-6）。

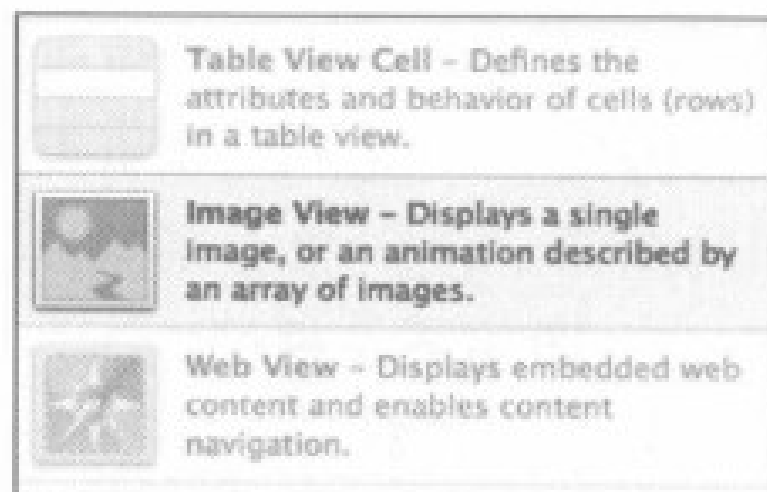


图4-6 Interface Builder的库中的Image View元素

4.3.3 添加图像视图

拖动一个图像视图到View窗口上。由于这是在视图上放置的第一个项，因此Interface Builder将自动重新调整图像视图的大小，使它与视图的大小保持一致。由于我们不希望图像视图占用整个空间，因此使用拖动手柄将图像视图的大小重新调整为与导入到Xcode中的图像相近的大小。不要担心很难做到精准无误，过一会儿你会发现，其实是很容易做到的。

顺便说一下，有时一些取消选中的对象很难再次选中，因为它们位于另一个对象后面，占用了整个视图，或者没有绘制边框。对于这些情况，完全不用担心！有一种方法可以重新选中这些对象。在nib的主窗口中，可以看到3个标签为View Mode的按钮。单击位于中间的按钮，可以得到nib的分层视图，你可以展开其中的子视图，如图4-7所示。双击此视图中的任何项目也可以在View窗口中选中同一项目。

选中图像视图之后，按⌘1调出检查器，应该能看到UIImageView类的可编辑选项，如图4-8所示。

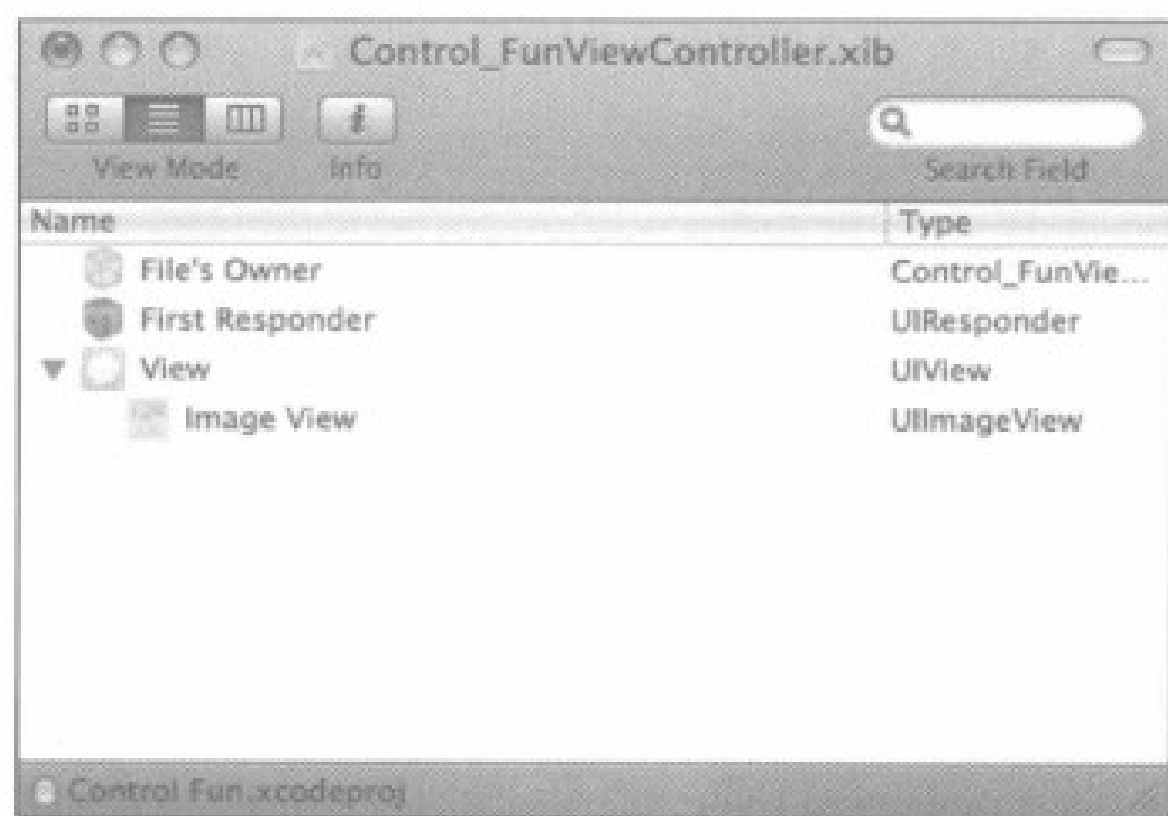


图4-7 打开nib的主窗口的分层视图并展开其子视图

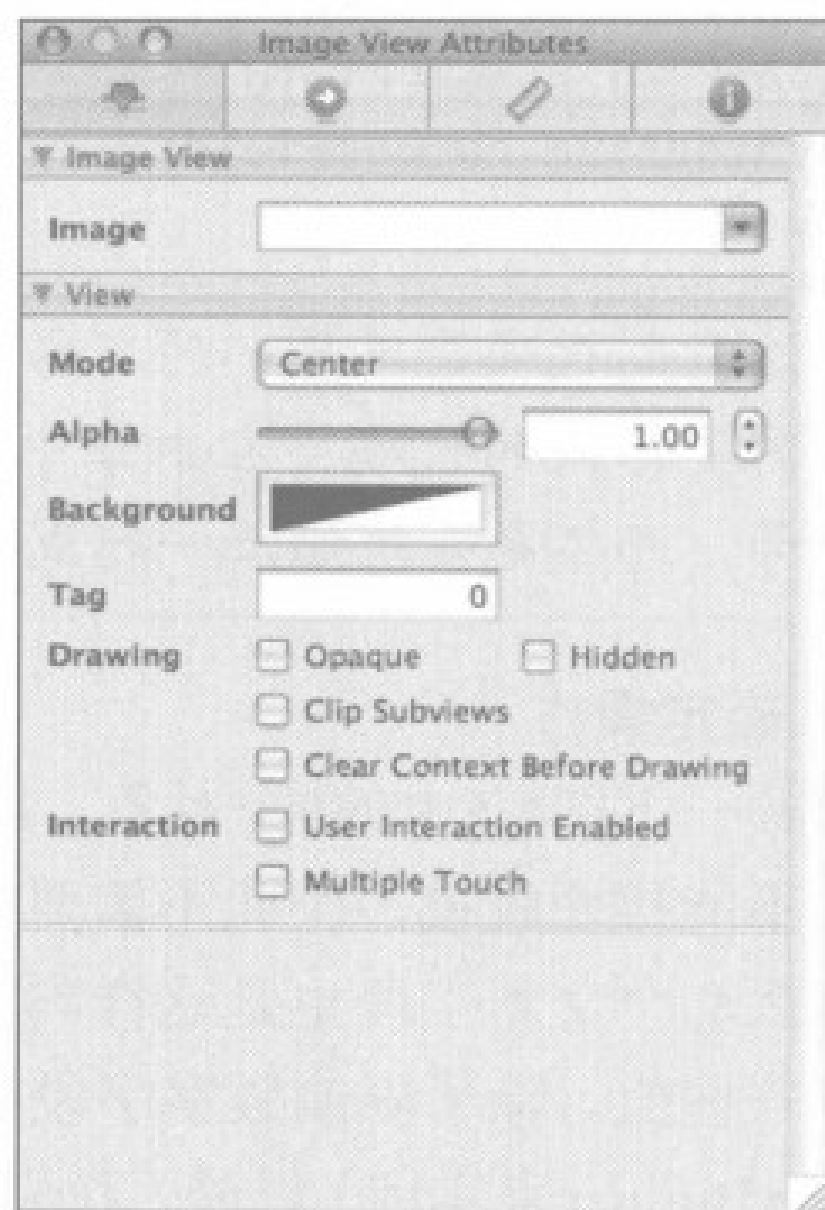


图4-8 图像视图检查器

图像视图中重要的设置就是位于检查器最顶部的Image属性。如果单击该字段右侧的小箭头，则会弹出一个菜单并显示可用的图像，其中应包括添加到Xcode项目中的任何图像。选择之前添加的图像。该图像现在应该出现在图像视图中。

1. 调整图像视图

现在，调整图像视图，使它与图像大小完全保持一致。我们将在稍后解释为何要这样做。实现此目的的一种简单方法是按⌘=或从Layout菜单中选择Size to fit。此选项将自动调整任何视图的大小，使它适应所包含的内容。我们还希望移动重新调整后的图像，使它位于视图中间，且顶部与蓝色引导线对齐。从Layout菜单的Alignment子菜单中选择Align Horizontal Center in Container，这样可以方便将视图中的项居中。

提示 在Interface Builder中拖动和重新调整视图需要一些技巧。不要遗忘了nib主窗口中的分层View Mode按钮。它将帮助你找到并选中（双击）图像视图。在重新调整视图大小时，可以按下option键。Interface Builder将在屏幕上绘制一些有用的红线，以便开发人员掌握图像视图的大小。此技巧不适用于拖动操作。在这种情况下，option键将通知Interface Builder你正在尝试创建已拖动对象的副本。

2. Mode属性

图像视图检查器中的下一个选项是Mode下拉菜单。Mode菜单用于定义图像在视图内部的对齐方式，以及是否缩放以适应视图。你可以随意尝试各种选项，但默认值Center可能最适合你的需要。记住，选择任何让图像缩放的选项都会潜在地增加处理开销，因此最好避免这些选项，并在导入图像之前调整好它们的大小。如果希望以多种大小显示同一图像，通常，更好的方式是在

项目中为该图像创建不同大小的多个副本，而不是强制iPhone在运行时对它们执行缩放。

3. Alpha滑块

检查器中的下一项是Alpha，此选项需要格外小心。Alpha定义图像的透明度，也就是图像背后的内容的可见度。如果使用任何小于1.0的值，则iPhone会将此视图绘制为透明的，这样其背后的任何对象都将可见。如果值小于1.0，即使图像背后没有任何内容，也会使应用程序花费处理器周期来计算透明度。因此，除非有足够的理由，否则一般要将该值设置为1.0。

4. 忽略Background

你可以忽略下一个项，即Background。该属性继承自UIView，但它不会影响图像视图的外观。

5. Tag属性

Tag选项值得注意，尽管本章并不会使用它。UIView的所有子类，包括所有视图和控件，都有一个标记属性，该属性只是与图像视图绑定在一起的一个数值。标记是供开发人员使用的，系统永远不会设置或修改它的值。如果为某控件或视图分配了一个标记值，则可以确定的是，该标记始终为这个值，除非你又修改了它。

标记（tag）是用于标识界面上的对象的一种与语言无关的方法。假设你有5个不同的按钮，每个按钮都有一个不同的标签（label），并且你希望使用一个操作方法来处理所有这5个按钮。在这种情况下，你可能需要通过某种方式在调用操作方法时区分这些按钮。当然，你可以查看按钮的标题，但是当应用程序转换为Swahili或Sanskrit之后，执行此操作的代码可能会失效。与标签不同，标记永远都不会更改，因此，如果在Interface Builder中设置了一个标记值，则随后可以使用它快速可靠地确定通过sender参数传递给操作方法的控件是哪一个。

6. Drawing复选框

Tag下方有一系列Drawing复选框。第一个复选框的标签为Opaque。选中它。这将通知iPhone OS，视图下的任何内容都不应绘制，并且允许iPhone的绘图方法通过一些优化来加速绘图。

你可能想知道为何需要选中Opaque复选框，因为已经将Alpha的值设定为了1.0（不透明）。其原因是，Alpha值适用于将被绘制的图像部分，但是，如果某个图像未完全填充图像视图，或者图像上存在一些洞（由Alpha通道或剪贴路径所致），则其下方的对象将仍然可见，而与Alpha的值无关。选中Opaque复选框之后，iPhone就会知道视图下方的任何内容都不需要绘制出来。我们可以放心地选中Opaque复选框，因为我们之前选中了Size to Fit，该选项使图像视图与它所包含的图像大小相匹配。

Hidden复选框的作用显而易见。选中它之后，用户将不能看到此控件。有时，隐藏控件是非常有用的，比如本章稍后需要隐藏开关。但在大多数情况下，开发人员都不会选中此选项。我们可以将其保留为默认值。

Clip Subviews是一个有趣的选项。如果你的视图有子视图，并且这些子视图并不是完全包含在其父视图中，则此复选框将确定子视图的绘制方式。如果选中了Clip Subviews，只有在父视图范围内的子视图部分将被绘制出来。如果未选中Clip Subviews，则全部子视图都将绘制出来，而不管它是否在父视图内部。图4-9展示了这一概念。

看上去，默认行为应与实际情况相反：默认启用Clip Subviews。由于iPhone上还有许多其他

元素，因此这与其性能密切相关。从数学上说，计算裁剪区域并仅显示子视图的部分是比较占用资源的操作，并且一般情况下，子视图不会位于父视图的外部。如果确实需要，你可以启用Clip Subviews，但出于性能的考虑，它在默认情况下是关闭的。

下一个复选框是很少需要选中的Clear Context Before Drawing。选中它之后，iPhone将使用透明黑色绘制控件覆盖的所有区域，然后才实际绘制控件。考虑到性能问题，并且适用情况很少，它默认为关闭状态。

7. Interaction复选框

最后两个复选框与用户交互有关。第一个复选框User Interaction Enabled指定用户能否对此对象进行操作。对于大多数控件，此复选框都是选中的，因为如果不这样的话，控件将永远不能触发操作方法。但是，标签和图像视图默认未选中此复选框，因为它们经常仅用于显示静态信息。

由于我们只是在屏幕上显示一张图像，因此不需要启用它。

最后一个复选框是Multiple Touch，用于确保此控件是否能够接收多点触摸事件。多点触摸事件支持各种复杂的手势，如用于缩放许多iPhone应用程序的双指捏合等操作。我们将在第13章中讨论有关手势和多点触摸事件的更多信息。由于此图像视图完全不接受用户交互，因此没有必要开启多点触摸事件，因此将它保留为默认值即可。

4.3.4 添加文本字段

完成图像视图之后，从库中拖出一个文本字段并将其移动到View窗口上。将它放置在图像视图下方，需使用蓝色引导线保持它与右边缘对齐（参见图4-10）。显示与图像视图的接近程度的水平蓝色引导线应该看作文本字段与图像视图之间的最短距离。你可以不改动它，但是为了获得更加平衡的外观，将它下移一点会比较好。记住，你随时可以回到Interface Builder中修改界面元素的位置和大小，而不需要修改代码或重新建立连接。

放置好文本字段之后，从库中拖出一个标签，移动它，使它与视图左侧对齐，并与之前放置的文本字段垂直对齐。注意，移动标签时会弹出多条蓝色引导线，这样便可以使用顶部、底部、中间或文本基准引导线来对齐文本字段与标签。我们将使用文本基准引导线来对齐标签和文本字段，该引导线将从标签文本的底部绘制一条到文本字段的线，如图4-11所示。如果蓝色引导线从标签文本中间穿过，则使用的是中间引导线，而不是文本基准引导线。使用文本基准引导线，可以保证标签的文本和用户将在文本字段中输入的文本在屏幕上处于相同的垂直位置。

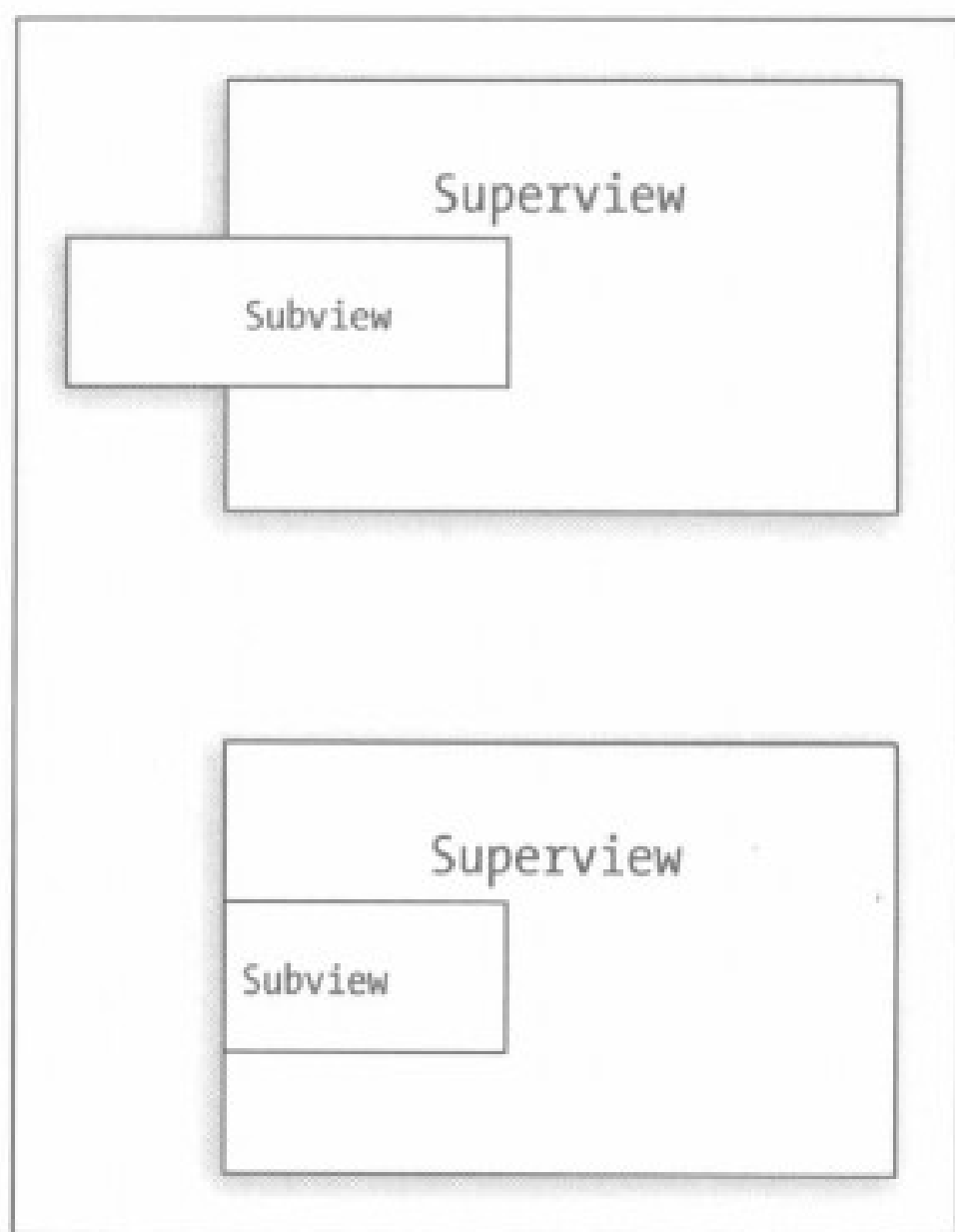


图4-9 Clip Subviews演示：上面的视图为默认设置，关闭了Clip Subviews。下面的视图是开启Clip Subviews后的情况

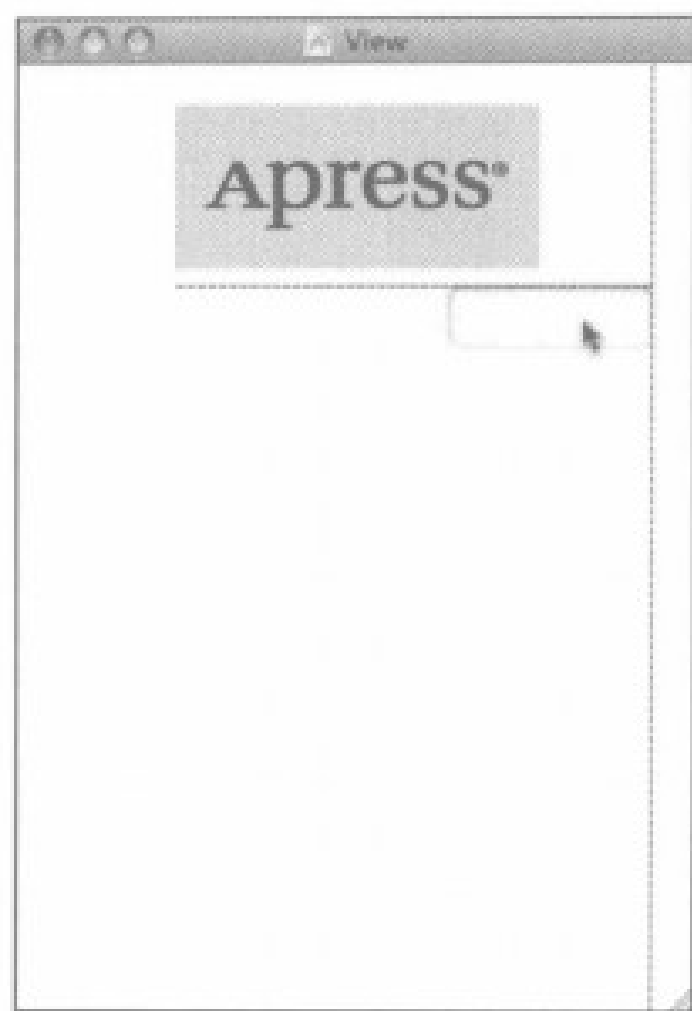
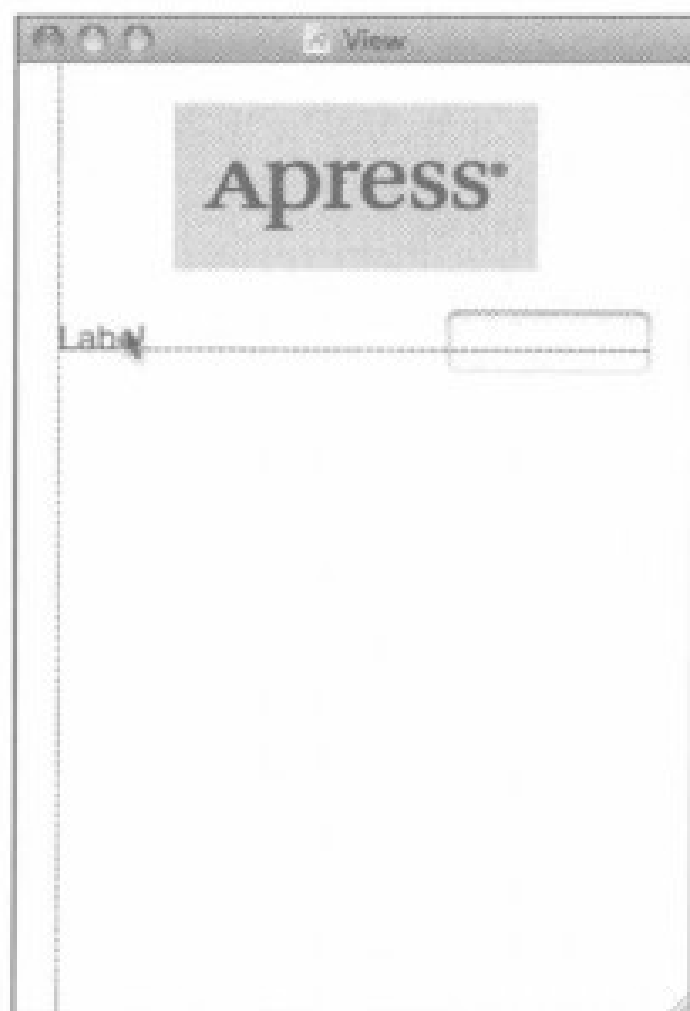


图4-10 放置文本字段

图4-11 使用文本基准引导线对齐
标签和文本字段

双击刚才放置的标签，进入编辑状态。键入Name:作为标签名，然后按回车键提交更改。接下来，从库中另外拖动一个文本字段到视图中，并使用引导线将它放置在第一个文本字段的下方（参见图4-12）。

放置第二个文本字段之后，从库中再拖出一个标签，将其置于已有标签的左下方。再次使用蓝色文本基准引导线将它与第二个文本字段对齐。双击新添加的标签，键入Number:。现在，单击左侧的调整点并向左拖动，扩展底部文本字段的大小。使用蓝色引导线确定文本字段的大小（参见图4-13）。

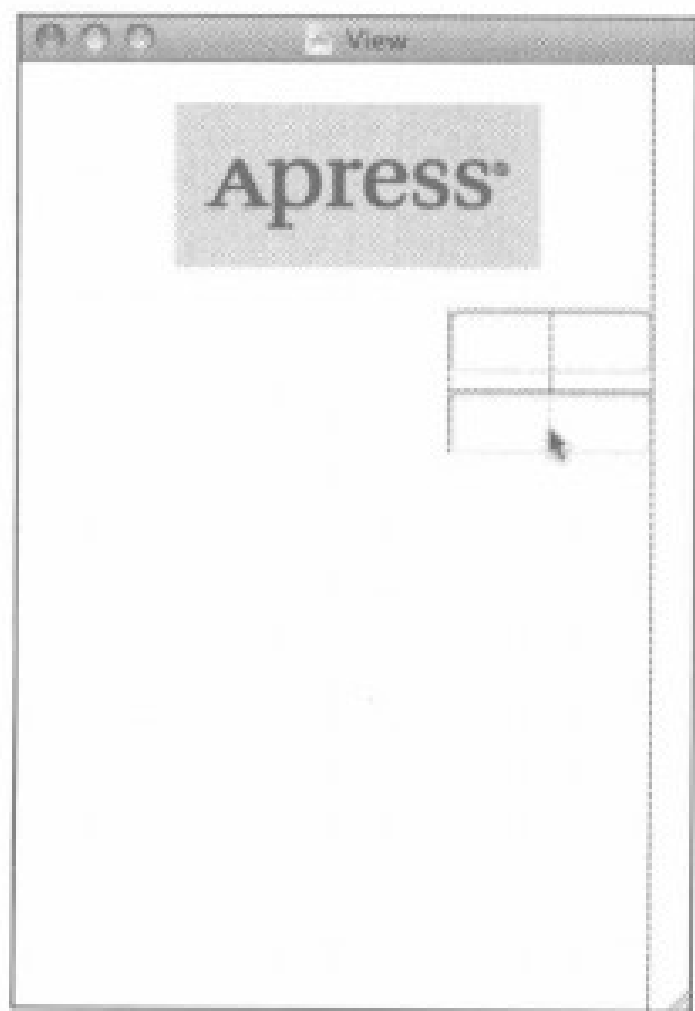


图4-12 添加第二个文本字段

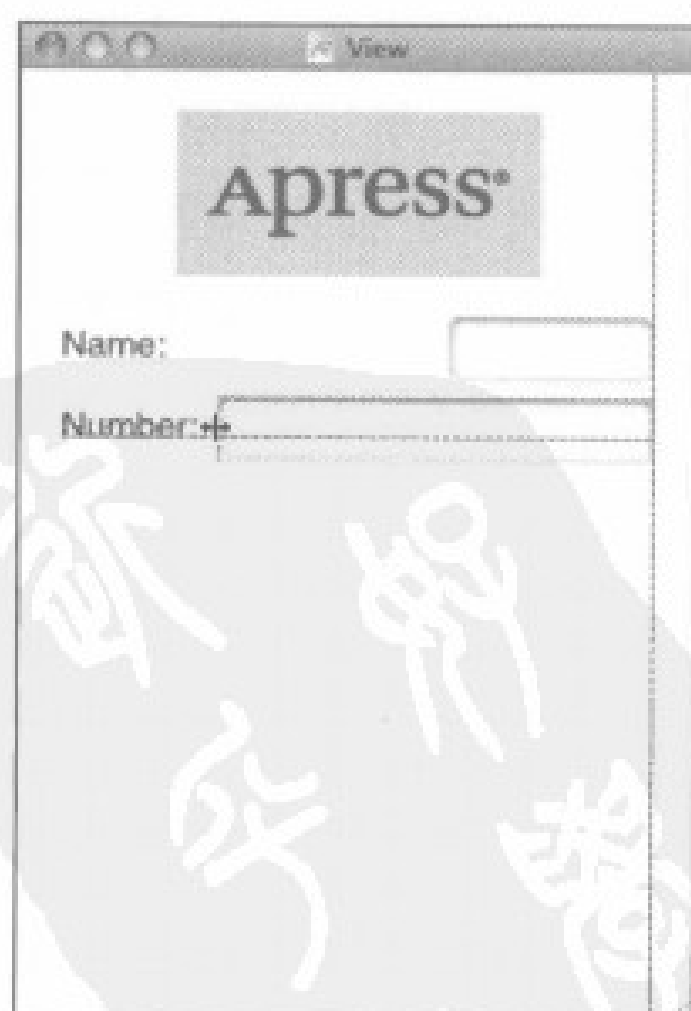


图4-13 扩展底部文本字段的大小

现在，采用相同的方法扩展顶部的文本字段，使它与底部文本字段的大小保持一致。完成之

后,界面应如图4-5所示。选中顶部的文本字段(如果未选中),然后按 $\mathbb{A}1$ 调出检查器(参见图4-14)。

1. 文本字段设置

文本字段是iPhone上最复杂和最常用的控件之一。首先看检查器最顶部的区域。在第一个字段Text中,可以将其设置为默认值。键入的任何内容都将在应用程序启动时在该字段中显示。

第二个字段是Placeholder,它用于指定将在文本字段中以灰色显示的文本,但前提是该字段没有值。如果空间不足的话,可以使用占位符来代替标签,或者使用它告诉用户应在此字段中键入的值。对于此字段,可以键入Type in a name作为占位符。

接下来的两个字段仅在需要定制文本字段的外观时使用,多数情况下,完全不必要也不建议使用它们。用户希望文本字段以预期的方式显示。因此,我们将略过Back-ground和Disabled字段,并将它们保留为空。

位于这些字段下方的是3个按钮,用于控制字段中文本的对齐方式。我将保留此字段为默认值,即左对齐(最左侧的按钮)。该按钮旁边的字段可用于指定文本字段文本的颜色。再一次,我们将它保留为默认的黑色。

接下来是4个Border按钮。它们用于更改文本字段边缘的绘制方式。可以任意尝试这4种不同的样式,但默认值是最左侧的按钮,它创建的文本字段样式是iPhone应用程序中最惯用的样式。因此,在尝试之后,请将其设置回默认值。

Clear When Editing Begins复选框指定用户触摸此字段时的操作。如果选中了该复选框,则之前该字段中的任何值都将被删除,并且用户将能够重新输入。如果未选中该复选框,则之前的值将仍然保留在此字段中,并且用户将能够编辑它。可以取消此字段(如果为选中状态),或者也可以保留为选中状态。

Adjust to Fit复选框指定文本的大小是否应随文本字段尺寸的减小而减小。此选项将确保整个文本在视图都可见,即使文本大于所分配的空间。此复选框的右侧是一个文本字段,用于指定最小文本大小。无论字段大小如何,文本都将根据此最小值进行调整。指定最小值可以确保文本不会因为过小而影响可读性。

2. 文本输入特征

接下来的部分定义在使用此文本字段时键盘的外观及行为。由于我们预期的文本是姓名,因此将Capitalize下拉列表更改为Words,此选项将所有单词自动转换为首字母大写,而这正符合姓名的要求。此外,将Return Key弹出项的值更改为Done,并将所有其他文本输入特征保留为默认

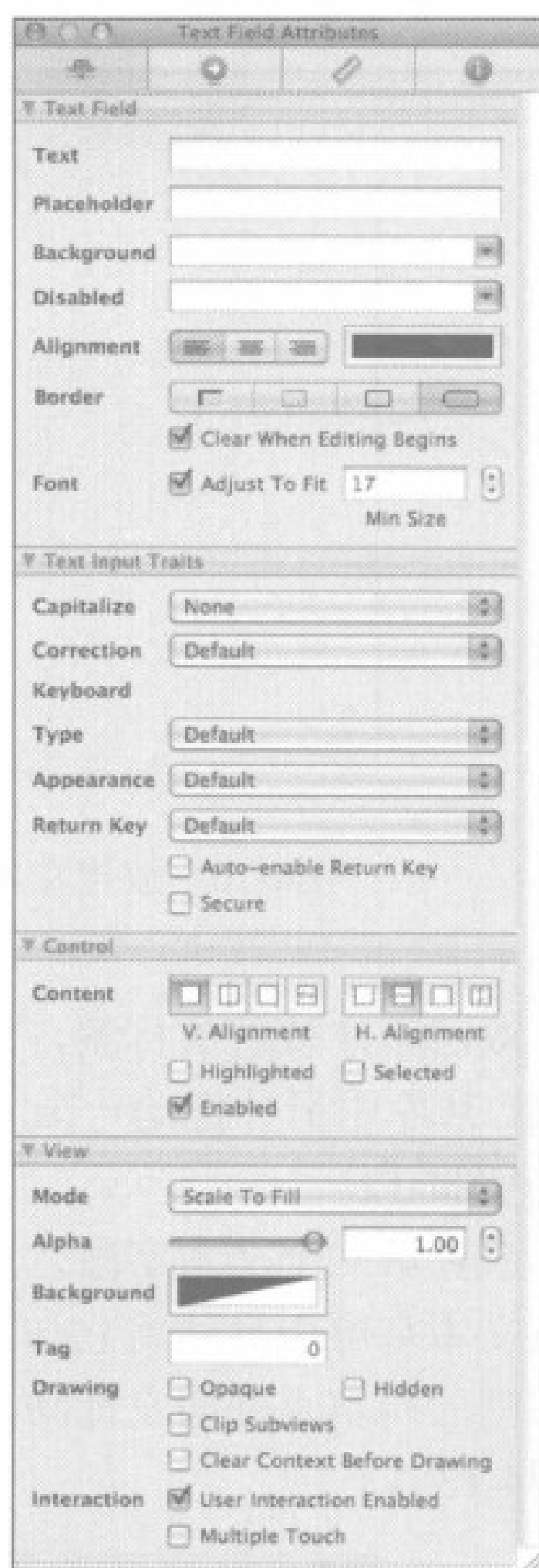


图4-14 显示默认值的文本字段检查器

值。Return Key是位于键盘右下方的键，并且其标签将根据用户的操作而变化。举例来说，如果在Safari的搜索字段中输入文本，那么它会显示Google。通常，对于普通应用程序的文本字段，Done是正确的选择。

3. 其他设置

接下来的部分用于设置继承自UIControl的一般控件属性，但它们通常不适用于文本字段，并且，除Enabled之外的所有复选框都不会影响字段的外观。我们希望启用这些文本字段，以便用户能够与它们交互，因此保留所有设置不变。

检查器上的最后一部分对你来说应该会比较熟悉。它与之前介绍的图像视图检查器上的同名部分是相同的。它们是继承自UIView类的属性，并且由于所有控件都是UIView的子类，因此它们都具有此部分中的属性。注意，对于文本字段，你不希望选中Opaque，因为这样会让输入的文本不可见。实际上，你可以将此部分中的所有值都保留为默认值。

4.3.5 设置第二个文本字段的属性

接下来，单击View窗口中的第二个文本字段，然后返回检查器。在Placeholder字段中，键入Type in a number，然后取消Clear When Editing Begins。在Text Input Traits部分中，单击Keyboard Type弹出菜单。由于我们只希望用户输入数字，而不包括字母，因此选中Number Pad。完成这些设置之后，用户所使用的键盘将只包含数字，这意味着不能输入字母字符、符号或数字之外的其他内容。我们不需要为数字键盘设置Return Key值。因为这种样式的键盘没有回车键，所以检查器上的任何选项都可保留默认值。

4.3.6 连接输出口

在设计界面的第一个阶段中，接下来的工作只需要连接输出口。按下Control键，并从File's Owner拖动到各文本字段，然后将它们连接到相应的输出口。将两个文本字段连接到相应的输出口之后，保存nib文件，然后返回Xcode。

4.4 构建和运行

接下来看看应用程序的运行情况。从Xcode的Build菜单中选择Build and Run。应用程序应出现在iPhone仿真器中。单击Name文本字段。此时会出现键盘（参见图4-15）。现在，单击Number字段，键盘将切换为数字面板。通过将文本字段添加到界面中，Cocoa Touch免费提供了所有这些功能。

非常顺利！但是，还有一个小问题。应该如何关闭键盘界面呢？且看下文分解。

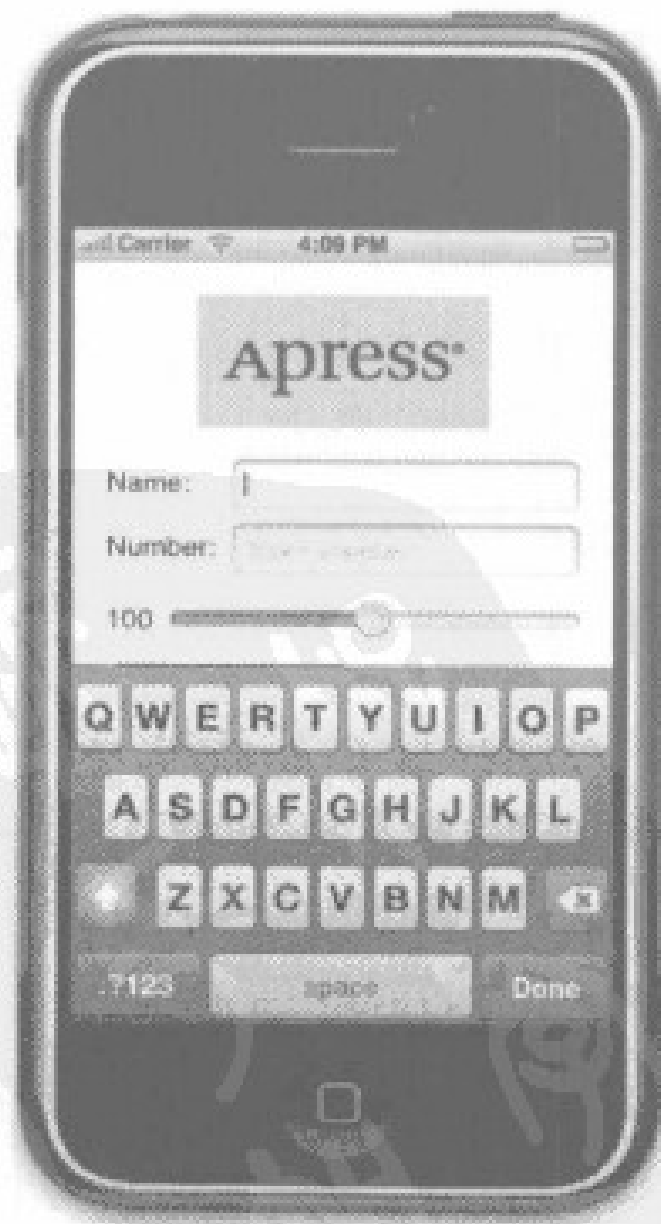


图4-15 触摸文本字段时键盘将自动显示

4.4.1 完成输入后关闭键盘

由于键盘是基于软件的，而不是物理键盘，因此我们需要一些额外的步骤来确保用户完成输入后可以关闭键盘。当用户按下Done按钮时，将生成一个“did end on exit”事件，此时，我们需要告诉文本字段取消控件，以关闭键盘。为此，我们需要在控制器类中添加一个操作方法，因此将以下代码行添加到Control_FunViewController.h中：

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

@interface Control_FunViewController : UIViewController {
    IBOutlet UITextField *nameField;
    IBOutlet UITextField *numberField;
}
@property (nonatomic, retain) UITextField *nameField;
@property (nonatomic, retain) UITextField *numberField;
- (IBAction)textFieldDoneEditing:(id)sender;
@end
```

现在切换到Control_FunViewController.m，我们将实现此方法。要让这个新操作方法正常运行，我们只需要添加一行代码。将以下方法添加到Control_FunViewController.m中：

```
- (IBAction)textFieldDoneEditing:(id)sender
{
    [sender resignFirstResponder];
}
```

我们之前已经提到第一响应者（first responder）的概念，并且说过它是用户当前正在与之进行交互的控件。此处，我们告诉触发此操作的任何控件取消第一响应者状态。当文本字段生成第一响应者状态之后，与之相关的键盘将消失。

保存刚才编辑的两个文件。返回Interface Builder，并通过这两个文本字段触发此操作。

返回Interface Builder之后，单击Name文本字段，然后按⌘2调出连接检查器。这一次，我们不需要设置第3章所使用的Touch Up Inside事件。而应采用Did End on Exit事件，该事件将在用户单击iPhone键盘上的Done按钮时被触发。从Did End on Exit旁边的圆圈拖动到File's Owner图标，并将它连接到textFieldDoneEditing:操作。对其他文本字段重复上述步骤，然后保存。最后，返回Xcode，再次构建和运行应用程序。

当仿真器出现之后，单击Name字段，键入一些内容，然后按下Done按钮。不出所料，键盘将随之消失。那么Number字段也是这样吗？可是该字段的Done按钮在哪里呢（参见图4-16）？

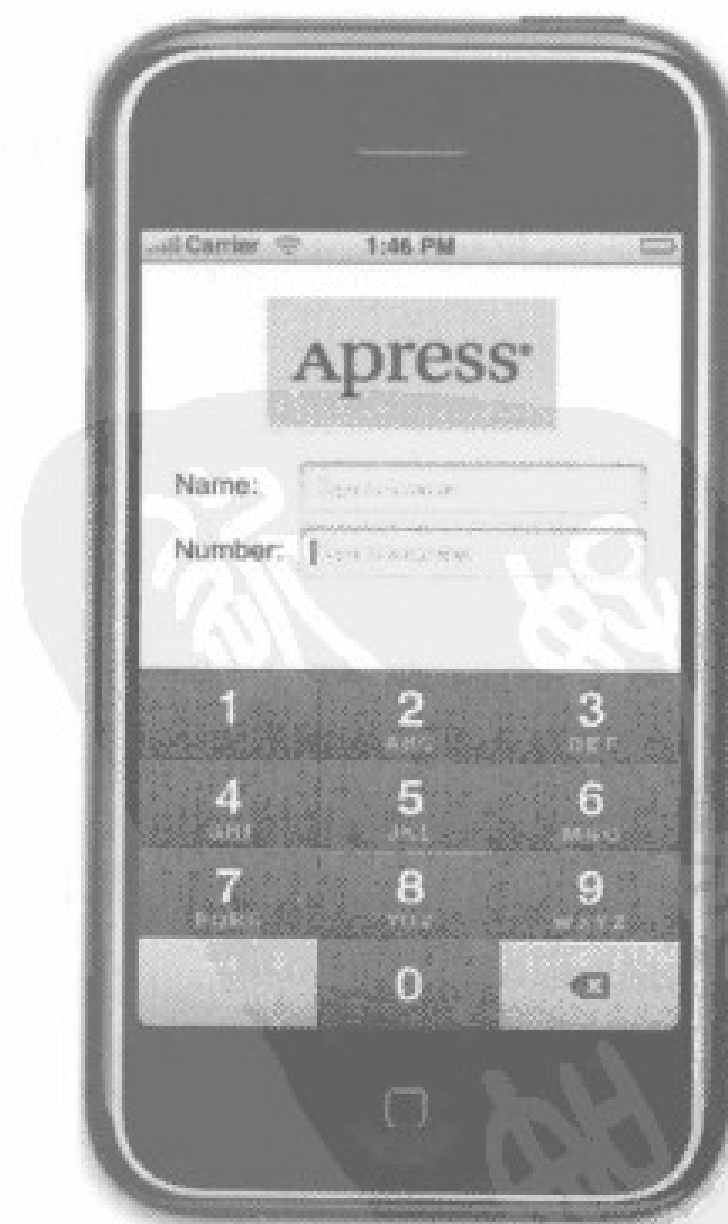


图4-16 数字键盘没有Done按钮

并非所有键盘布局都有Done按钮。我们可以强制用户按下Name字段，然后再按Done按钮，但这缺乏较佳的用户体验。我们显然希望应用程序具有很好的用户体验。

还记得苹果公司的iPhone应用程序是如何应对此问题的吗？在大多数有文本字段的情况下，在视图中任何无活动控件的位置按下手指都可以让键盘消失。应如何实现此功能呢？

答案可能会让你大吃一惊，因为它实在太简单了。我们需要创建一个不可见的按钮，将其置于其他所有元素后面，用于通知文本字段在检测到触摸操作时生成第一响应者状态。

4.4.2 通过触摸背景关闭键盘

返回Xcode。我们需要在控制器类中再添加一个操作。将以下行添加到Control_FunView-Controller.h文件中：

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

@interface Control_FunViewController : UIViewController {
    IBOutlet    UITextField    *nameField;
    IBOutlet    UITextField    *numberField;
}
@property (nonatomic, retain) UITextField *nameField;
@property (nonatomic, retain) UITextField *numberField;
- (IBAction)textFieldDoneEditing:(id)sender;
- (IBAction)backgroundClick:(id)sender;
@end
```

保存头文件并切换到实现文件。在其中添加此代码，其作用是通知所有文本字段在必要时生成第一响应者状态。在非第一响应者控件上调用resignFirstResponder是绝对安全的，因此我们可以放心地对两个文本字段调用它，而不需要检查哪一个才是第一响应者（如果有的话）。

```
- (IBAction)backgroundClick:(id)sender
{
    [nameField resignFirstResponder];
    [numberField resignFirstResponder];
}
```

提示 在编写代码时，你将在头文件与实现文件之间来回切换。幸运的是，Xcode提供了一个组合键，用于在这些文件之间快速来回切换。默认的组合键为⌘⇧（option+command+向上箭头），但你可以在Xcode的首选项中任意更改它们。

保存此文件，然后返回Interface Builder。我们现在需要创建那个非常神秘的不可见按钮，该按钮将调用backgroundClick:操作。请确保打开了View窗口和库。从库中拖一个Round Rect Button到视图窗口中。使用按钮边缘的调整点让它填满整个屏幕。这次不要在蓝色边缘线的位置停止，而是继续拖动到边缘外部，不要担心它会覆盖视图中的其他项目。如果要将此按钮置于其他所有元素后面，就从Layout菜单中选择Send to Back。

现在，仍然选中这个新按钮，按⌘1调出检查器，并将按钮类型从Rounded Rect更改为Custom，

如图4-17所示。

按钮应基本处于不可见状态，除了边缘附近的调整点。Custom选项经常用于覆盖按钮的默认外观。但是，它也适用于没有外观的按钮，本例正是这种情况。

现在，按⌘2切换到连接检查器，然后按住鼠标从该按钮的Touch Up Inside事件拖到File's Owner图标，并选择backgroundClick:操作。现在，触摸视图中没有活动控件的区域都将触发这个新操作方法，从而关闭键盘。

保存nib文件，返回Xcode并尝试运行应用程序。再次编译和运行应用程序。这一次，可以通过两种方式来关闭键盘：按Done按钮或单击无活动控件的任何区域。而这正是用户所期望的行为。

非常好！解决了键盘的问题之后，我们将继续创建下一组控件。

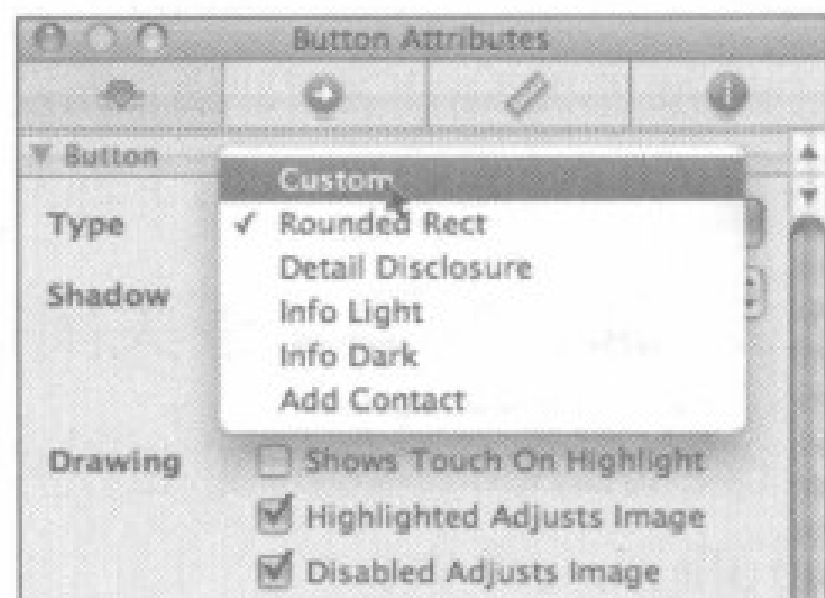


图4-17 将按钮类型从Rounded Rect改为Custom

4.5 实现滑块和标签

完成文本字段之后，我们现在开始实现滑块控件。记住，当用户移动滑块时，标签将随之更改以反映滑块的值。

4.5.1 确定输出口

我们将在界面中添加另外两个项。想知道我们究竟需要多少输出口吗？我们需要编写代码让标签随滑块的变化而变化，因此标签只需要一个输出口。那么滑块呢？

滑块将触发一个操作，随后，该操作方法将通过sender参数接受一个指向滑块的指针。我们能够从sender中检索滑块的值，因此我们不需要通过输出口来获取滑块的值。那么，滑块是否完全不需要输出口呢？换句话说，我们需要在滑块将调用的操作方法外部访问滑块的值吗？

在实际的应用程序中，经常需要这样做。此处，由于另一个控件将与滑块拥有相同的值，并且已经有了一个输出口，因此没有必要为滑块本身再定义一个输出口。编写iPhone应用程序时需要有考虑内存的属性。虽然指针占用的内存是最小的，但如果没有必要，也不应使用它。况且这样还会让代码更乱。

4.5.2 确定操作

这两个控件的操作非常简单。我们需要在滑块发生更改时调用一个操作。标签是静态的，并且用户不能直接对它执行任何操作，因此它不需要触发任何操作。

4.5.3 添加输出口和操作

在Control_FunViewController.h文件中再声明一个输出口和一个操作，如下所示：

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

@interface Control_FunViewController : UIViewController {
    IBOutlet    UITextField    *nameField;
    IBOutlet    UITextField    *numberField;
    IBOutlet    UILabel        *sliderLabel;
}
@property (nonatomic, retain) UITextField *nameField;
@property (nonatomic, retain) UITextField *numberField;
@property (nonatomic, retain) UILabel *sliderLabel;
- (IBAction)textFieldDoneEditing:(id)sender;
- (IBAction)backgroundClick:(id)sender;
- (IBAction)sliderChanged:(id)sender;
@end
```

由于知道该方法的准确作用，因此切换到Control_FunViewController.m，添加属性同步程序并编写sliderChanged:方法：

```
#import "Control_FunViewController.h"

@implementation Control_FunViewController
@synthesize nameField;
@synthesize numberField;
@synthesize sliderLabel;
- (IBAction)sliderChanged:(id)sender
{
    UISlider *slider = (UISlider *)sender;
    int progressAsInt = (int)(slider.value + 0.5f);
    NSString *newText = [[NSString alloc] initWithFormat:@"%d",
        progressAsInt];
    sliderLabel.text = newText;
    [newText release];
}
- (IBAction)backgroundClick:(id)sender
{
    ...
}
```

下面介绍一下sliderChanged:方法。该方法首先将sender转换为UISlider *。其作用只是让我们的代码更具有可读性，并避免对sender进行类型转换。此后，我们接受滑块的值（int类型），将其加0.5，以便四舍五入为整型值。然后使用该整型值创建一个新字符串，用于设置标签的文本。由于分配了newText，因此我们需要负责释放它。此操作由该方法的最后一行完成。很简单，不是吗？接下来将这两个对象添加到界面中。

保存更改，继续下一节内容。

4.5.4 添加滑块和标签

你现在已经了解了例程。双击Control_FunViewController.xib，将其打开。或者，如果已经打开该文件，直接返回Interface Builder。从库中拖出一个滑块，并将其放置在Number文本字段的下

方，让它占用大部分（不是全部）水平空间。在左侧留一点空间给标签。可以参照图4-1进行操作。单击新添加的滑块以选中它，如果它还不可见，就按 $\text{⌘}1$ 返回检查器。检查器应如图4-18所示。

用户可以通过滑块选择特定范围内的数值。此处，我们在Interface Builder中设置滑块范围和初始值。输入1作为最小值，100作为最大值，50作为初始值。目前只需要了解这些设置。

在滑块旁边放置一个标签，需使用蓝色引导线保持它与滑块垂直对齐，并保持其左侧边缘与视图的左侧边缘对齐（参见图4-19）。

双击新添加的标签，将其文本从Label更改为100。这是滑块可以支持的最大值，并且我们可以使用它确定滑块的正确宽度。由于“100”比“Label”短，因此应该调整标签的大小，方法是将正中间的调整点向左拖动。确保不要让文本变小。如果它开始变小，应将调整点往回拖一点，回到原来的位置。还可以使用之前讨论的Size to Fit选项，方法是按 $\text{⌘}=\text{}$ 或从Layout菜单中选择Size to Fit。接下来，调整滑块的大小，单击滑块以选中它并将其左侧调整点向左拖动，直到与蓝色引导线靠齐。

现在，再次双击标签，将其值改回滑块的初始值50。这是因为我们需要确保界面在启动时能正确显示；使用滑块之后，刚才编写的代码将确保标签继续显示正确的值。

4.5.5 连接操作和输出口

接下来只需要连接这两个控件的输出口和操作。你还等待什么呢？你已经知道此操作的步骤了。按下Control键并从File's Owner图标拖动到刚才添加的标签，然后选择sliderLabel。接下来，单击滑块，按 $\text{⌘}2$ 调出连接检查器。这次不再需要Touch Up Inside事件，对吗？那么可以使用Value Changed吗？这听起来是一个不错的主意。很好，从Value Changed拖到File's Owner，并选择sliderChanged。

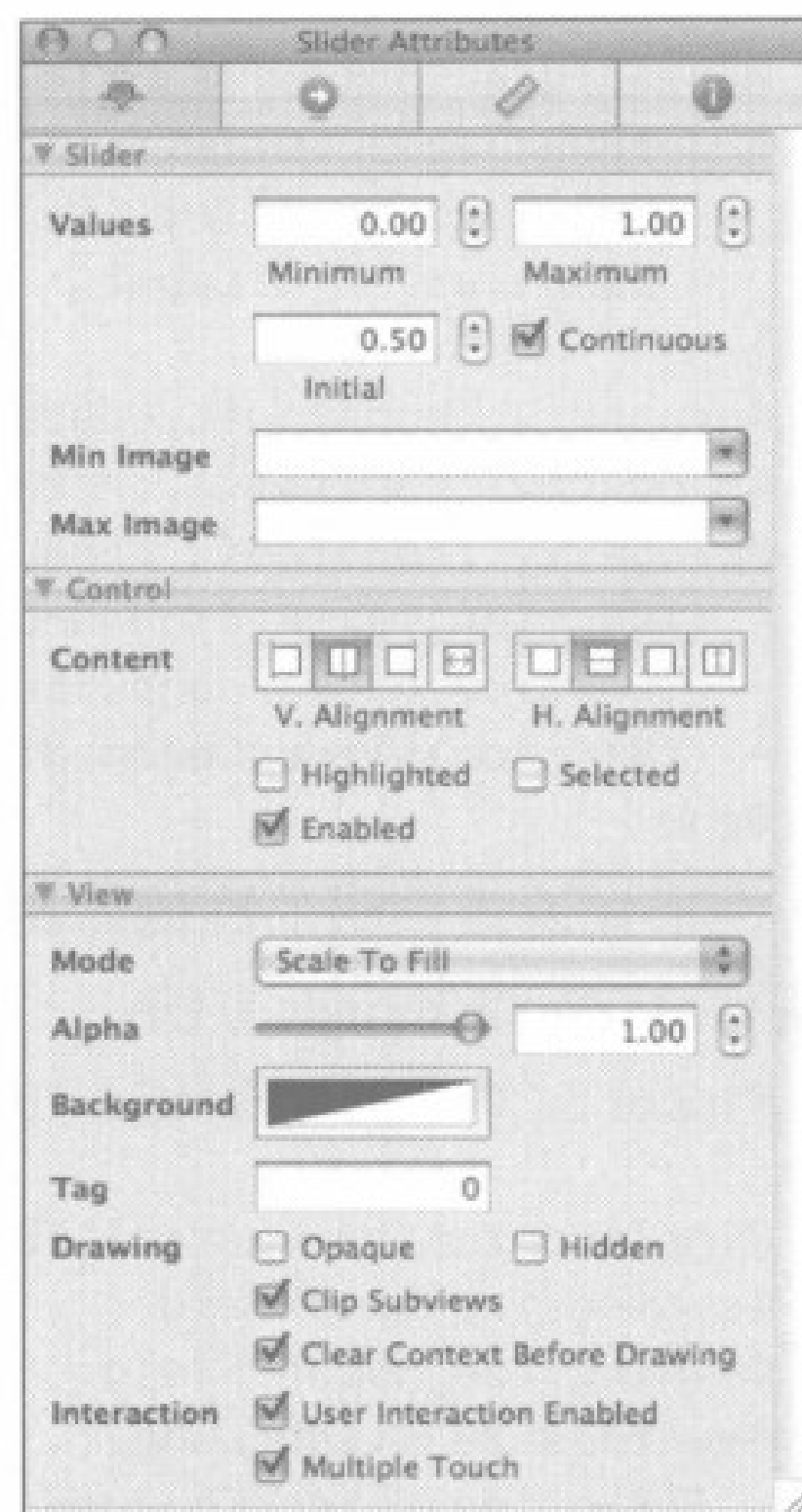


图4-18 显示滑块默认属性的检查器

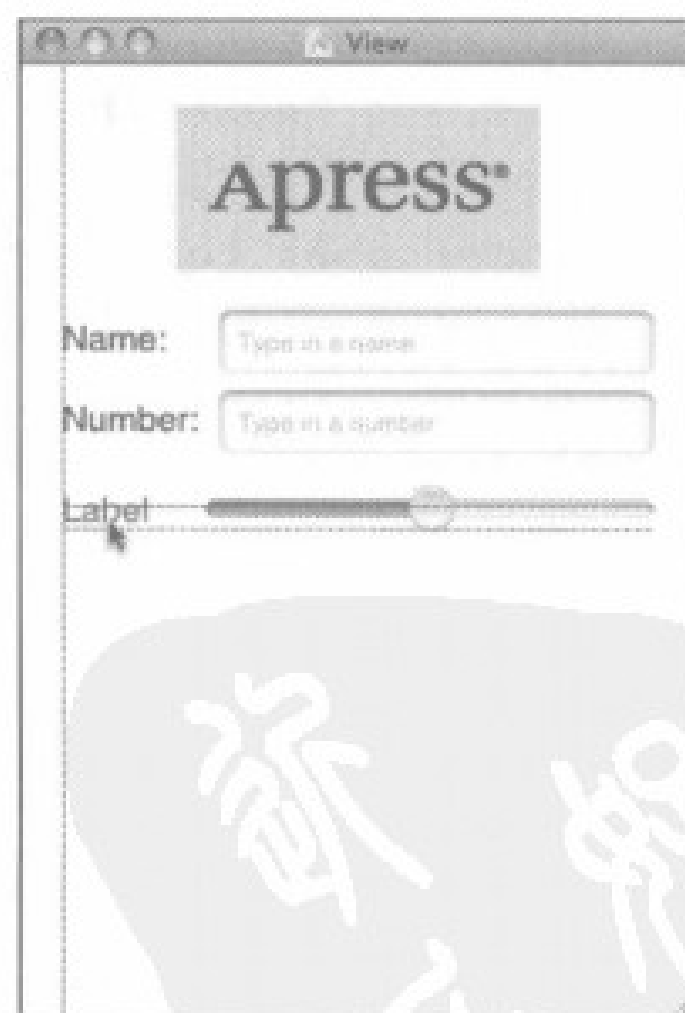


图4-19 放置滑块的标签

注意 在某些版本的Interface Builder中，当你尝试按下Control键并从File's Owner拖动到界面上的项时，只能选择较大的背景按钮，而不能选择sliderLabel等其他项。如果遇到过此问题，可以按 $\text{⌘}2$ 调出连接检查器，并从输出口旁边的小圆圈拖动到要连接的对象。

保存nib文件并返回Xcode。试用一下滑块。当你移动它时，应该可以看到标签的文本会实时变化。是时候讨论另一个问题了。接下来，我们介绍如何实现开关。

4.6 实现开关和分段控件

再次返回Xcode。有点头晕了吧？这种来回切换看上去有点奇怪，但在开发时经常需要在Interface Builder、Xcode和iPhone仿真器之间来回切换。

我们的应用程序将包括两个开关。这种小控件仅有两种状态：开和关。我们还将添加一个分段控件，用于隐藏和显示开关。接下来，我们将实现它们。

4.6.1 确定输出口

我们不希望为分段控件指定输出口，因为我们不会修改其属性或在其调用的操作方法外部引用它。但是，我们需要为开关指定一些输出口。由于更改任一开关的值都会使另一个开关的值随之变化，因此我们更改未触发操作方法的开关的值，从而不会依赖sender。我们还需要另外一个输出口，用于将添加的另一个视图。记住，我们将在触摸分段控件时隐藏或显示这些开关以及它们的标签。

我们可以单独隐藏各项，但隐藏和显示多个控件的最简单的方法是，使用UIView作为需要隐藏或显示的项的公共父项。稍后，你将看到如何在Interface Builder中完成此任务。不过首先，除了两个开关的输出口之外，我们还需要为父视图创建一个输出口。

4.6.2 确定操作

分段控件需要触发一个操作方法，用于隐藏或显示包含开关及其标签的视图。我们还需要在触摸开关时触发一个操作。这两个开关将调用相同的操作方法，这与第3章中的两个按钮一样。在Control_FunViewController.h中，添加3个输出口和两个操作，如下所示：

```
#import <UIKit/UIKit.h>
#define kShowSegmentIndex 0
@interface Control_FunViewController : UIViewController {
    IBOutlet UITextField *nameField;
    IBOutlet UITextField *numberField;
    IBOutlet UILabel *sliderLabel;
    IBOutlet UISwitch *leftSwitch;
    IBOutlet UISwitch *rightSwitch;
    IBOutlet UIView *switchView;
}
@property (nonatomic, retain) UITextField *nameField;
@property (nonatomic, retain) UITextField *numberField;
@property (nonatomic, retain) UILabel *sliderLabel;
@property (nonatomic, retain) UISwitch *leftSwitch;
@property (nonatomic, retain) UISwitch *rightSwitch;
@property (nonatomic, retain) UIView *switchView;
```



```

- (IBAction)textFieldDoneEditing:(id)sender;
- (IBAction)backgroundClick:(id)sender;
- (IBAction)sliderChanged:(id)sender;
- (IBAction)switchChanged:(id)sender;
- (IBAction)toggleShowHide:(id)sender;
@end

```

注意 在键入代码时，Xcode会猜测你的意图并帮你完成单词。此特性称作代码感知（Code Sense），能够节省大量时间。有时，它可能会犯错。当你开始键入switchView时，它会尝试给出建议的switch语句。只需继续键入并忽略建议，建议词条便会自动消失。

在即将编写的代码中，我们将引用一个UISegmentedControl属性，即selectedSegmentIndex。该属性告诉我们当前选中的是哪一个分段。它是一个整数值。Show分段的索引将为0。我们在代码中将其定义为kShowSegmentIndex常量，而不是直接使用0值，从而让代码更具有可读性。

切换到Control_FunViewController.m，然后添加以下代码：

```

#import "Control_FunViewController.h"

@implementation Control_FunViewController
@synthesize nameField;
@synthesize numberField;
@synthesize sliderLabel;
@synthesize leftSwitch;
@synthesize rightSwitch;
@synthesize switchView;
- (IBAction)switchChanged:(id)sender
{
    UISwitch *whichSwitch = (UISwitch *)sender;
    BOOL setting = whichSwitch.isOn;
    [leftSwitch setOn:setting animated:YES];
    [rightSwitch setOn:setting animated:YES];
}
- (IBAction)toggleShowHide:(id)sender
{
    UISegmentedControl *segmentedControl = (UISegmentedControl *)sender;
    NSInteger segment = segmentedControl.selectedSegmentIndex;

    if (segment == kShowSegmentIndex) [switchView setHidden:NO];
    else [switchView setHidden:YES];
}
- (IBAction)sliderChanged:(id)sender
{
    ...
}

```

我们添加的第一个方法相当简单。其作用只是获取sender的值，并使用该值来设置两个开关。现在，sender的值始终只能等于leftSwitch和rightSwitch之一。你可能想知道为何要同时设定两个开关的值。与确定当前调用的开关并仅设置另一个开关的方法相比，每次都设置两个开关的

值会比较省事一些。无论哪个开关调用此方法，它们都将被设置为正确的值，并且将其重新设置为该值不会有任何影响。

注意，当我们修改开关的值时，需要使用`animated`参数。该参数用于指定按钮的移动方式：缓慢移动或瞬间跳转。我们将其指定为`YES`，因为缓慢移动的效果比较好，并且iPhone用户比较喜欢这种视觉效果。你可以指定`NO`来看看它们之间的差异，但是，除非有足够的理由，否则应采用动画效果，这样用户才能知道发生了什么。

在第二个方法`toggleShowHide:`中，我们查看当前选中的分段，并显示或隐藏`UIView`（已为它创建了输出口），`UIView`是需要隐藏的所有对象的父视图。一些特定的属性是从父视图继承而来的，因此如果某个视图为隐藏状态或被禁用，则该视图的所有子视图也将隐藏或被禁用。

保存两个文件，返回Interface Builder，以便于编辑`Control_FunViewController.xib`和创建开关。

4.6.3 添加开关和分段控件

从库中拖出一个分段控件（参见图4-20），并将它放置在View窗口的滑块的下方。

扩展分段控件的宽度，使它从视图左侧拉伸到右侧，如图4-21所示。将光标移动到分段控件的`First`单词上，然后双击它。分段的标题将变为可编辑状态，将它从`First`更改为`Show`，如图4-21所示。完成此操作之后，对`Second`分段重复此步骤，并将其重命名为`Hide`。

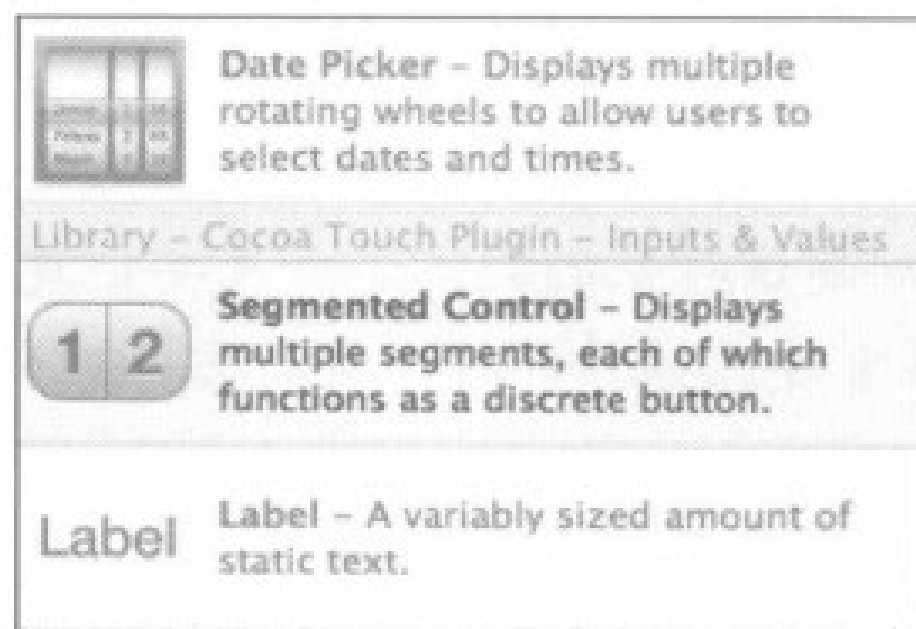


图4-20 库中的分段控件选项

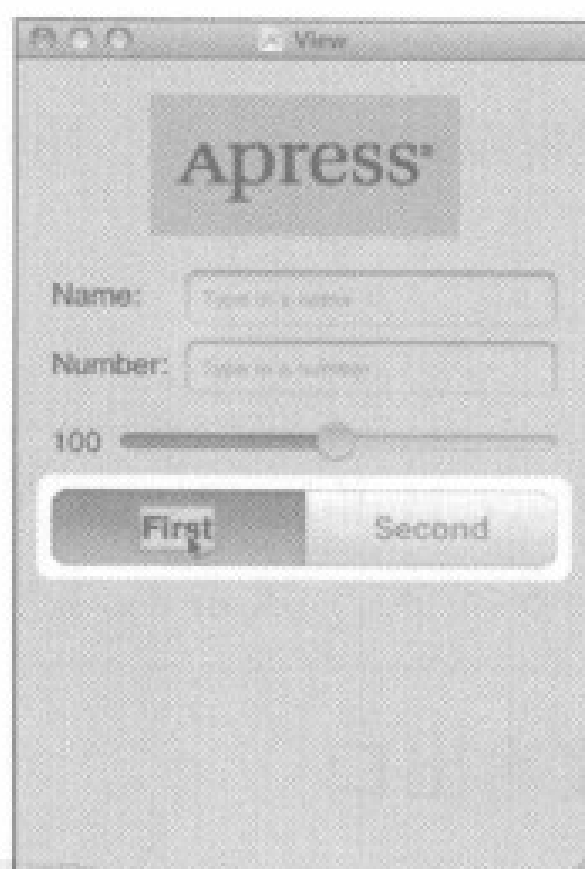


图4-21 为分段重新命名

1. 添加另一个视图

从库中再拖出一个视图，并将它放置在View窗口中的分段控件的正下方。默认情况下，`UIView`元素未被绘制出，但这不会影响界面的外观。它只是一个用于保存其他视图的容器。但是，我们可以更改视图的背景颜色，使它可见并且更易于操作。你可以在检查器中更改新视图的背景色，我们选择的是浅灰色（如图4-22所示），但你也可以选择其他颜色。

如果在设置新背景色之前未选中视图，那么可以通过一个小技巧来找到它并帮助你设计视图。按下`option`键，将光标移动到View窗口中。当光标移动到某个对象上时，它将突出显示为红色，你还可以看出项与其父视图之间距离多少像素，如图4-23所示。

2. 添加两个带标签的开关

从库中拖出一个开关，将其放置在刚才添加到窗口中的视图中。记住，你需要确保将开关添加为新视图的子视图，且在释放鼠标之前能够看到一个灰色的View框和一个绿色的加号图标，如图4-24所示。对第二个开关重复此步骤，确保也将其添加为子视图。现在，同样以子视图方式添加两个标签。在各开关上方分别放置一个标签，并将左侧标签的值更改为Left，将右侧标签的值更改为Right。

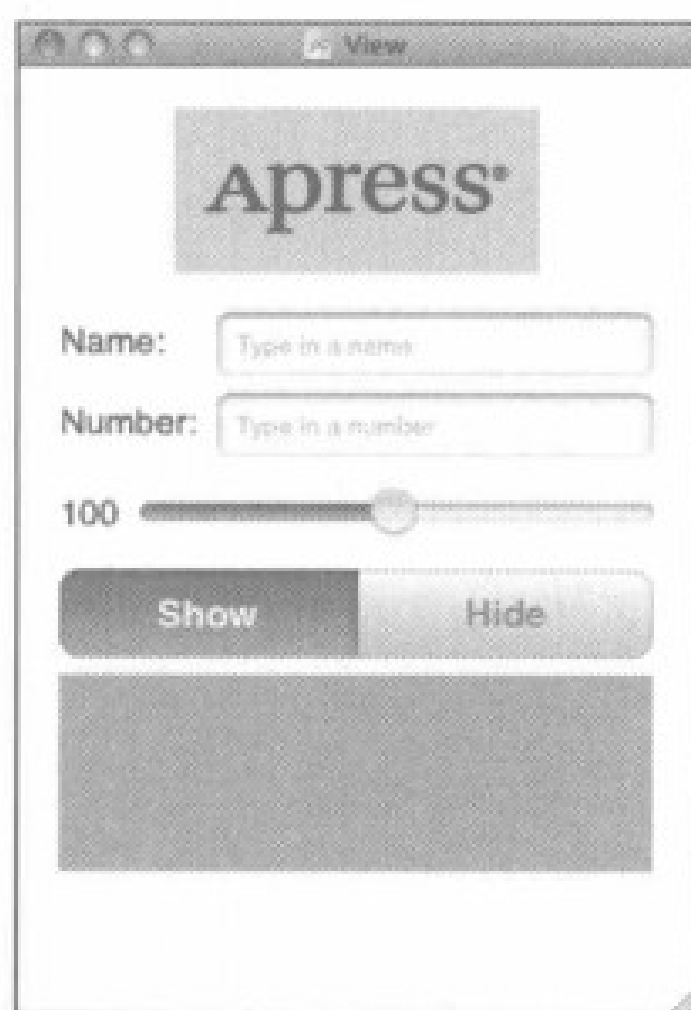


图4-22 放置视图对象

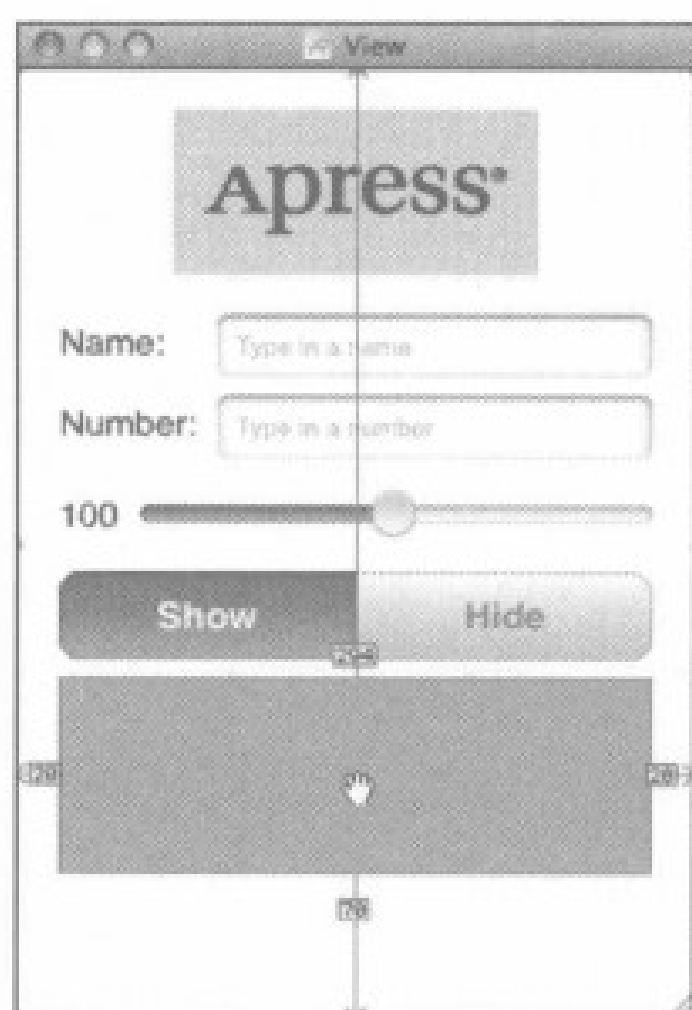


图4-23 按下option键，并将光标移动到界面上的项目上方，可以获取更多信息

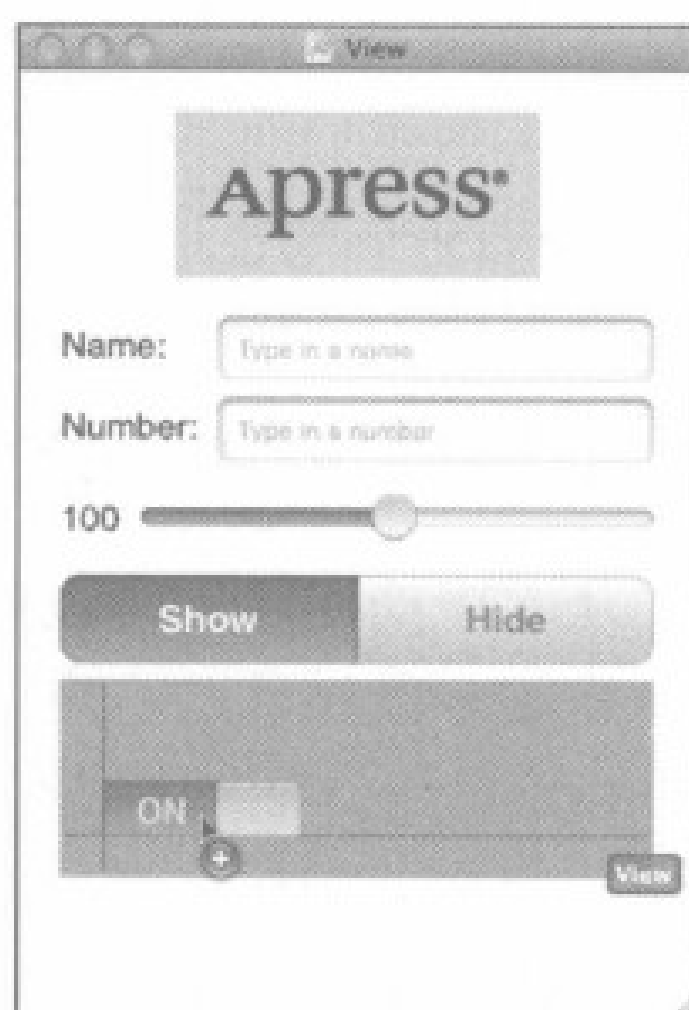


图4-24 将开关添加为已有视图的子视图

提示 在Interface Builder中，按下option键并拖动对象会创建该对象的副本。如果要创建同一对象的许多实例，只需从库中拖出一个对象，然后再重复此操作即可。这种方式比较快捷。

4.6.4 连接输出口

按下Control键并从File's Owner拖到每个开关，然后将它们连接到合适的leftSwitch或rightSwitch输出口。完成此操作后，按下Control键并从File's Owner图标拖到开关的父视图，再将它连接到switchView输出口。

现在，再次单击左侧开关将其选中，并按⌘2调出连接检查器。从Value Changed事件拖动到File's Owner图标，然后选择switchChanged:操作。对另一个开关重复此步骤。

单击分段控件，在连接检查器中找到Value Changed事件。从该事件旁边的小圆圈拖动到File's Owner图标，并选择toggleShowHide:操作方法。

保存工作。

返回Xcode，并测试应用程序。单击一个开关，观察另一个开关的值的变化的情况。单击分段控件，观察开关和标签的显示及隐藏行为。

完成测试之后，我们继续最后一部分。

4.7 实现按钮、操作表和警报

本章的内容非常有趣，而这一节将会更加有趣，因为最精彩的内容总是会放在最后。最后一个尚未实现的控件是Do Something按钮。上一章已经介绍了该按钮的作用，所以此处将讨论一些新内容。

4.7.1 将输出口及操作添加到控制器头文件

我们需要为按钮创建一个新的操作方法和输出口。该操作方法将在用户按下按钮时被调用。稍后在编写代码定制按钮外观时需要使用该输出口。

除了输出口和操作之外，头文件中还有一个要求：我们的类必须符合UIActionSheetDelegate协议。当用户按下按钮时，程序将向他们显示一个操作表，询问是否确定要继续。当用户按下一个操作表按钮时，表会消失，并调用一个方法告诉我们按钮已被按下。为了接收操作表中的消息，我们需要符合UIActionSheetDelegate协议，并实现该协议中的一个回调方法。操作表是模式化的，这意味着当程序显示它们时，用户不能与应用程序的任何其他部分交互。因此，我们基本上是在强制用户做出决定，然后才能执行其他操作。

添加到Control_FunViewController.h中的代码如下所示：

```
#import <UIKit/UIKit.h>
#define kShowSomeSegmentID 0
@interface Control_FunViewController : UIViewController
    <UIActionSheetDelegate> {
        IBOutlet UITextField *nameField;
        IBOutlet UITextField *numberField;
        IBOutlet UILabel *sliderLabel;
        IBOutlet UIProgressView *progressView;
        IBOutlet UISwitch *leftSwitch;
        IBOutlet UISwitch *rightSwitch;
        IBOutlet UIView *switchView;
        IBOutlet UIButton *doSomethingButton;
    }
@property (nonatomic, retain) UITextField *nameField;
@property (nonatomic, retain) UITextField *numberField;
@property (nonatomic, retain) UILabel *sliderLabel;
@property (nonatomic, retain) UIProgressView *progressView;
@property (nonatomic, retain) UISwitch *leftSwitch;
@property (nonatomic, retain) UISwitch *rightSwitch;
@property (nonatomic, retain) UIView *switchView;
@property (nonatomic, retain) UIButton *doSomethingButton;
- (IBAction)textFieldDoneEditing:(id)sender;
- (IBAction)backgroundClick:(id)sender;
- (IBAction)sliderChanged:(id)sender;
- (IBAction)switchChanged:(id)sender;
```

```

- (IBAction)toggleShowHide:(id)sender;
- (IBAction)doSomething:(id)sender;
@end

```

4.7.2 在 Interface Builder 中添加按钮

保存工作，然后切换回Interface Builder。从库中拖出一个Round Rect Button，并将其放置在视图中。双击该按钮，将其标题指定为Do Something。接下来，按下Ctrl键并从File's Owner图标拖到该按钮，并将其连接到doSomethingButton输出口。然后，将Touch Up Inside事件连接到doSomething:操作，方法是从连接检查器上的该事件拖回到File's Owner图标。

保存nib文件，然后返回Xcode，我们将进入最后一步。

4.7.3 实现按钮的操作方法

切换回Control_FunViewController.m并实现我们的操作方法。我们实际上需要实现多个方法，因为如前所述，除了操作方法之外我们还需要实现一个UIActionSheetDelegate方法，以便在用户按下操作表上的按钮时获取通知。下面是需要添加到Control_FunViewController.m中的代码。键入代码，稍后我们讨论其运行原理：

```

#import "Control_FunViewController.h"

@implementation Control_FunViewController
@synthesize nameField;
@synthesize numberField;
@synthesize sliderLabel;
@synthesize leftSwitch;
@synthesize rightSwitch;
@synthesize switchView;
@synthesize doSomethingButton;

- (IBAction)doSomething:(id)sender
{
    UIActionSheet *actionSheet = [[UIActionSheet alloc]
        initWithTitle:@"Are you sure?"
        delegate:self
        cancelButtonTitle:@"No Way!"
        destructiveButtonTitle:@"Yes, I'm Sure!"
        otherButtonTitles:nil];
    [actionSheet showInView:self.view];
    [actionSheet release];
}

- (void)actionSheet:(UIActionSheet *)actionSheet
didDismissWithButtonIndex:(NSInteger)buttonIndex
{
    if (!buttonIndex == [actionSheet cancelButtonTitle])
    {
        NSString *msg = nil;
    }
}

```

```

    if (nameField.text.length > 0)
        msg = [[NSString alloc] initWithFormat:
            @"You can breathe easy, %@, everything went OK."
            , nameField.text];
    else
        msg = @"You can breathe easy, everything went OK.";

    UIAlertView *alert = [[UIAlertView alloc]
        initWithTitle:@"Something was done"
        message:msg
        delegate:self
        cancelButtonTitle:@"Phew!"
        otherButtonTitles:nil];
    [alert show];
    [alert release];
    [msg release];
}

- (IBAction)switchChanged:(id)sender
{
    ...
}

```

4.8 显示操作表

首先看一下doSomething:方法。该方法的确切作用是什么？我们首先分配并初始化一个UIActionSheet对象，该对象表示一个操作表：

```

UIActionSheet *actionSheet = [[UIActionSheet alloc]
    initWithTitle:@"Are you sure?"
    delegate:self
    cancelButtonTitle:@"No Way!"
    destructiveButtonTitle:@"Yes, I'm Sure!"
    otherButtonTitles:nil];

```

初始化方法接受多个参数。让我们依次介绍这些参数。第一个参数是显示的标题。如图4-3所示，我们提供的标题将显示在操作表的顶部。

第二个参数是操作表的委托。操作表的委托将在该表上的按钮被按下时收到通知。更确切地说，委托的actionSheet:didDismissWithButtonIndex:actionSheet:didDismissWithButtonIndex:方法将被调用。通过将self作为委托参数传递给该方法，我们可以确保本程序的actionSheet:didDismissWithButtonIndex:方法将被调用。

接下来的参数是取消按钮的标题，用户可以点击此按钮以表明他们不希望继续操作。所有操作表都应有一个取消按钮，但你可以根据需要为它指定合适的标题。如果没有选择，则不必使用操作表。如果只希望通知用户，而不让用户做出选择，则警报表会比较合适。我们将简要讲解一下如何使用警报表。

下一个参数是destructive按钮，你可以将它理解为“确定继续”按钮。你仍然可以根据需要为它指定合适的标题。

最后一个参数用于指定希望在表单上显示的其他按钮的数量。该参数可以使用各种值，这是Objective-C语言中的一个非常好的特性。如果我们希望操作表上还有另外两个按钮，可以编写以下代码：

```
UIAlertSheet *actionSheet = [[UIAlertSheet alloc]
    initWithTitle:@"Are you sure?"
    delegate:self
    cancelButtonTitle:@"No Way!"
    destructiveButtonTitle:@"Yes, I'm Sure!"
    otherButtonTitles:@"Foo", @"Bar", nil];
```

这样，操作表将提供4个按钮供用户选择。你可以在otherButtonTitles参数中传递任意数量的变量，只要nil作为最后一个变量传递即可。但根据可用屏幕空间的大小，按钮的数量将受到实际限制。

创建操作表之后，我们将让它显示自己。在iPhone上，操作表始终有一个父视图，即当前对用户可见的视图。在本例中，我们希望使用在Interface Builder中设计的视图作为父视图，因此使用self.view。view是父类UIViewController的一个属性，它指向该类的控制器所对应的视图。

最后，完成所有上述操作之后，我们将释放操作表。不必担心，此操作在用户按下按钮之后才会执行。

操作表委托和创建警报

这并不难，对吗？在寥寥数行代码中，我们显示了一个操作表，并要求用户做出决定。iPhone甚至为表的显示创建了动画，而这不需要我们做任何其他工作。现在，我们只需要找出用户按下的按钮即可。刚才实现的actionSheet:didDismissWithButtonIndex方法是一个UIAlertSheet-Delegate方法，并且由于我们指定self作为操作表的委托，因此该方法将在用户按下按钮时自动被警报表调用。

参数buttonIndex可以告诉我们实际按下的是哪个按钮。但是，我们如何才能知道哪个按钮索引指向取消按钮，哪个按钮索引指向其他按钮呢？幸运的是，委托方法的参数中包含一个到UIAlertSheet对象（表示该操作表）的指针，并且该操作表对象知道哪个按钮是取消按钮。我们只需要查看它的cancelButtonIndex属性：

```
if (!buttonIndex == [actionSheet cancelButtonTitle])
```

此行代码将确保用户不会按下取消按钮。由于我们只为用户提供了两个选项，因此我们知道，如果用户未按下取消按钮，则他们必须按下另一个按钮，即选择OK按钮以继续操作。知道用户未取消操作之后，首先需要创建一个向用户显示的新字符串。在实际的应用程序中，此处将实现用户所请求的任务。我们假设执行了一些任务，并使用警报通知用户。

如果用户在顶部的文本字段中输入了姓名，我们将获取该值并在警报消息中使用它。否则，仅显示一条普通消息：

```
NSString *msg = nil;

if (nameField.text.length > 0)
    msg = [[NSString alloc] initWithFormat:
        @"You can breathe easy, %@, everything went OK.",
        nameField.text];
else
    msg = @"You can breathe easy, everything went OK.";
```

接下来的代码看上去应非常熟悉。警报的创建和使用方式与前面操作表的非常相似：

```
UIAlertView *alert = [[UIAlertView alloc]
    initWithTitle:@"Something was done"
    message:msg
    delegate:nil
    cancelButtonTitle:@"Phew!"
    otherButtonTitles:nil];
```

我们又传递了一个要显示的标题，这次附带了一条比较详细的消息，也就是刚才创建的字符串。警报同样有自己的委托，并且，如果我们需要知道用户何时关闭了警报，或按下的是哪个按钮，则可以在此处指定self作为委托，这与操作表相同。如果完成了此操作，那么现在还需要让类符合UIAlertViewDelegate协议，并实现该协议中的一个或多个方法。在本例中，我们只是向用户通知了一些消息，并且仅为用户提供了一个按钮。实际上，我们对按钮是何时按下的并不关心，并且我们已经知道哪个按钮将被按下，因此此处仅指定了nil，表示我们不需要在用户按下按钮时执行任何操作。

与操作表不同，警报并没有与特定视图绑定在一起。因此，我们仅需显示警报，而没有指定父视图。随后，仅需要进行一些内存清理操作，这样，我们的任务就完成了。保存工作，然后构建并运行完成后的应用程序。

4.9 美化按钮

比较运行中的应用程序与图4-1，你可能会注意到一个有趣的差异。Do Something按钮的外观不一样，它与操作表或其他iPhone应用程序中的按钮也不一样。这是因为默认Round Rect Button按钮的外观本来就不太好，因此我们最后再稍微处理一下该按钮。

iPhone中的大多数按钮都是使用图像绘制的。不必担心，你不需要在图像编辑器中为每个按钮都创建一个图像。只需要指定iPhone在绘制按钮时所使用的模板图像类型。

需要记住，你的应用程序是沙盒化的。你不能访问iPhone上的其他应用程序所使用的模板图像，或iPhone OS所使用的图像，因此，你需要确保所有需要的图像都位于应用程序的束中。那么，可以从何处获取这些图像模板呢？

好在苹果公司提供了一个束。你可以从iPhone示例应用程序UICatalog中获取它们，地址为：

<http://developer.apple.com/iphone/library/samplecode/UICatalog/index.html>, 或者也可以从本书项目归档的04 Control Fun文件夹将它们复制出来。没错, 你可以在自己的应用程序中使用这些图像, 苹果公司的示例代码许可证特别允许你使用和分发它们。

因此, 从UICatalog项目的04 Control Fun文件夹或Images子文件夹中将两个图像(blueButton.png和whiteButton.png)添加到你的Xcode项目中。

如果在Preview.app或图像编辑程序中打开其中一个图像, 你将看到它们并没有什么特别之处, 但在按钮中使用它们需要一点技巧。

返回Interface Builder, 单击Do Something按钮, 然后将其类型从Round Rect修改为Custom(与之前的不可见背景按钮相同)。在检查器中, 你可以为按钮指定图像, 但我们并不会这样做, 因为这些图像模板需要采用稍微不同的处理方式。保存nib文件, 然后返回Xcode。

4.9.1 viewDidLoad 方法

如果需要修改在nib中创建的任何对象, 我们可以重写控制器父类UIViewController中的viewDidLoad方法。由于在Interface Builder中并不能完成一切任务, 因此我们将利用viewDidLoad方法。将以下代码添加到Control_FunViewController.m文件中。完成之后, 我们将解释其原理。

```
#import "Control_FunViewController.h"

@implementation Control_FunViewController
@synthesize nameField;
@synthesize numberField;
@synthesize switchView;
@synthesize leftSwitch;
@synthesize rightSwitch;
@synthesize sliderLabel;
@synthesize doSomethingButton;

- (void)viewDidLoad
{
    UIImage *buttonImageNormal = [UIImage imageNamed:@"whiteButton.png"];
    UIImage *stretchableButtonImageNormal = [buttonImageNormal
        stretchableImageWithLeftCapWidth:12 topCapHeight:0];
    [doSomethingButton setBackgroundImage:stretchableButtonImageNormal
        forState:UIControlStateNormal];

    UIImage *buttonImagePressed = [UIImage imageNamed:@"blueButton.png"];
    UIImage *stretchableButtonImagePressed = [buttonImagePressed
        stretchableImageWithLeftCapWidth:12 topCapHeight:0];
    [doSomethingButton setBackgroundImage:stretchableButtonImagePressed
        forState:UIControlStateHighlighted];
}

- (IBAction)sliderChanged:(id)sender
{
    ...
}
```

此代码根据添加到项目中的模板图像为按钮设置背景图像。它指定按钮被触摸时应从白色图像转换为蓝色图像。这个简短的方法引入了两个概念：**控件状态**和**可拉伸图像**。接下来我们将依次介绍这两个概念。

4.9.2 控件状态

每个iPhone控件都有4种不同的控件状态，并且它任何时候都处于并仅能处于其中的一种状态。最常见的状态是**普通控件状态**，这是默认状态。控件在未处于其他状态时都为这种状态。**突出显示状态**是控件在使用时的状态。对于按钮来说，这表示用户将手指放在其上。**禁用状态**是控件被关闭时的状态。要禁用控件，可以在Interface Builder中取消Enabled复选框，或将控件的enabled属性设置为NO。最后一种状态是**选中**，仅有一部分控件支持该状态，并且通常用于指示该控件已启用或被选中。选中状态与突出显示状态类似，但控件可以在用户不再直接使用它时继续保持选中状态。

某些iPhone控件的属性可以根据其状态接受不同的值。举例来说，通过为UIControlStateNormal指定一个图像，并为UIControlStateHighlighted指定另一个图像，我们告诉iPhone在加亮状态（用户将手指放在按钮上时）和其他状态下分别使用这两个不同的图像。

4.9.3 可拉伸图像

可拉伸图像是一个有趣的概念。可拉伸图像是可调整大小的图像，它知道如何智能地重新调整其大小，以维持正确的外观。对于这些按钮模板，我们不希望边缘也被均匀拉伸。**端帽**（end cap）是一个图像的一部分，以像素为单位进行度量，它就不应该被调整。我们希望边缘保存原样，而与按钮的大小无关，因此我们将左侧端帽的大小设置为12。

由于我们为按钮指定了新的可拉伸图像，而未使用图像模板，因此iPhone知道如何正确绘制任意大小的按钮。现在，我们可以在Interface Builder中更改按钮的大小，iPhone仍然会正确绘制它。如果我们在Interface Builder中直接指定了按钮图像，则它将均匀地调整整个图像，这样按钮在大部分大小下看起来会比较奇怪。

提示 我们应该如何确定端帽的值呢？非常简单：我们复制了苹果公司的示例代码。

为何不保存并尝试运行应用程序呢？一切都应该与之前相同，只是这个按钮现在更加符合iPhone的风格了。

4.10 小结

本章内容比较多。在概念上，我们并没有讲述太多新内容，而是着重介绍了许多控件的用法，以及各种实现的细节。你应该熟悉了输出口和操作的实际用法，并了解了如何利用视图的分层属性。你学习了控件状态和可拉伸图像，以及如何使用操作表和警报表。

这个小应用程序包含众多元素，因此可以尝试使用更改各属性的值，或者尝试添加和修改代

码，看Interface Builder中的不同设置会有哪些不同效果。我们无法逐一介绍iPhone中可用的每一个控件，但本章中的应用程序是了解每个控件的一个很好的起始点，涵盖了许多基础知识。

下一章将介绍用户在纵向模式和横向模式之间来回旋转iPhone时会发生什么情况。你可能已经知道许多iPhone应用程序会根据用户握持iPhone的方式来更改其显示风格，而我们将介绍如何在你的自己的应用程序中实现此功能。



iPhone是工程设计领域中一项卓越的成就。苹果公司的工程师竭尽所能在口袋大小的设备中实现了最丰富的功能。比如，支持在纵向模式（长而窄）或横向模式（短而宽）下使用应用程序，支持在旋转电话时更改应用程序的方向。这种行为称为自动旋转，iPhone的Web浏览器Mobile Safari就是一个典型的例子（参见图5-1）。



图5-1 与许多iPhone应用程序一样，Mobile Safari能够根据握持电话的方式来更改显示效果，以充分利用可用的屏幕空间

并非所有应用程序都支持自动旋转。苹果公司的一些iPhone应用程序仅支持单方向模式。例如，影片只能在横向模式下观看，联系人信息只能在纵向模式下编辑。基本原则是，仅当自动旋转能够增强用户体验时，才将它添加到应用程序中。

幸运的是，苹果公司在iPhone OS和UIKit中出色地隐藏了自动旋转的复杂性，因此在你自己的iPhone应用程序中实现这种行为实际上非常容易。

可以在视图控制器中指定自动旋转，如果用户旋转电话，视图控制器将判断是否应该旋转到新的方向（本章稍后将介绍如何操作）。如果视图控制器认为应该旋转，则会旋转应用程序的窗

口和视图，调整窗口和视图的大小以适应新的方向。

对于在纵向模式下启动的视图，其宽度和高度分别为320像素和460像素，如果没有状态栏，则高度为480像素。状态栏位于屏幕顶部，其宽度为20像素（参见图5-1），用于显示信号强度、时间以及电池电量等信息。将电话切换到纵向模式时，视图和应用程序的窗口也会随之旋转，它们将调整大小以适应新的方向。对于调整之后的视图，其宽度和高度分别为480像素和300像素（如果没有状态栏，则高度为320像素）。

在屏幕中调整像素的大部分实际工作是由iPhone OS来完成的。在此过程中，应用程序的主要任务是确保每个对象都能够与屏幕吻合，能够在调整后的窗口中正确显示。

应用程序可以采用3种常用方法来管理旋转。具体使用哪种方法依界面的复杂度而定，本章稍后将介绍这3种方法。对于较简单的界面，可以为界面中的所有对象指定正确的自动调整属性。在调整视图时，自动调整属性将通知iPhone视图中的控件应该采用哪种行为方式。如果你曾经在Mac OS X上使用过Cocoa，那么应该熟悉这个基本过程，因为这也是用户在调整窗口时，指定其中包含的Cocoa控件的行为方式的过程。本章稍后将结合实际例子来介绍这个概念。

自动调整既快速又简单，但并非所有应用程序都适用。较复杂的界面必须采用不同的方式来处理自动旋转。对于较复杂的视图，可以采用两种基本方法来处理自动旋转。一种方法是在看到视图旋转提示时，手动调整视图中的对象位置。第二种方法是在Interface Builder中为视图设计两种不同版本，一种适用于纵向模式，而另一种适用于横向模式。无论使用哪种方法，都需要覆盖视图控制器类的UIViewController中的方法。

让我们开始吧，准备好了吗？首先看一下自动调整。

5.1 使用自动调整属性处理旋转

在Xcode中创建一个新项目，将其命名为Autosize。此应用程序将始终采用同一个基于视图的应用程序模板。在Interface Builder中设计视图之前，需要告诉iPhone该视图支持自动旋转。可以通过修改视图控制器类来实现这一点。

5.1.1 指定旋转支持

在Xcode中打开项目之后，展开Classes文件夹，然后单击AutoSizeViewController.m。如果查看该文件中已有的代码，你会看到模板已经提供了一个名为shouldAutorotateToInterfaceOrientation:的方法。现在该方法应该和如下所示的一样：

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
```

系统通过调用此方法询问视图控制器是否应该旋转到指定方向。系统共定义了4种方向，分别对应握持iPhone的4种常见方式：

- ❑ `UIInterfaceOrientationPortrait`
- ❑ `UIInterfaceOrientationPortraitUpsideDown`
- ❑ `UIInterfaceOrientationLandscapeLeft`
- ❑ `UIInterfaceOrientationLandscapeRight`

当电话的方向发生更改时，系统将调用此方法来操作视图控制器。`interfaceOrientation`参数将包含上述4个值之一，并且此方法需要返回YES或NO，以指示是否应该旋转应用程序的窗口以匹配新的方向。由于每个视图控制器子类实现此方法的方式各不相同，因此一个应用程序可能仅支持旋转部分视图，而不支持旋转其他视图。

提示 注意，iPhone中定义的系统常量采用以下命名方式：彼此相关的值都使用相同的字母开头。`UIInterfaceOrientationPortrait`、`UIInterfaceOrientationPortraitUpsideDown`、`UIInterfaceOrientationLandscapeLeft`和`UIInterfaceOrientationLandscapeRight`都以`UIInterfaceOrientation`开头的一个原因，是为了充分利用Xcode的代码感知（Code Sense）特性。你可能注意到，在Xcode中输入代码时，Xcode经常会尝试自动完成要输入的字符串。这就是代码感知特性的一种实际应用。开发人员不可能记住系统中定义的所有常量，但可以记住常用变量集的开头形式。当需要指定方向时，只需输入`UIInterfaceOrientation`（或者只是`UIInterf`），然后按下`escape`键便可显示所有匹配元素的列表（可以在Xcode的首选项中更改键设置）。与输入的字符串匹配的所有合法常量、变量、方法和函数都会显示出来，只需按下`tab`或`return`键就可以选择它们。这比在文档或头文件中查找值要快得多。

此方法的默认实现会检查`interfaceOrientation`，仅当它与`UIInterfaceOrientationPortrait`相等时才返回YES，这样会将应用程序限制为一种方向，从而能够有效地禁用自动旋转。

如果要启动自动旋转，只需将方法更改为对传入的任何值都返回YES，例如：

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    return YES;
}
```

如果只想支持其中的一部分方向，则必须检查`interfaceOrientation`的值，对想要支持的值返回YES，对不想支持的值返回NO。例如，要支持两个方向中的纵向模式和横向模式，但是不支持旋转到倒置的纵向模式，可以编写如下代码：

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    return (interfaceOrientation == UIInterfaceOrientationPortrait ||
            interfaceOrientation == UIInterfaceOrientationLandscapeLeft ||
            interfaceOrientation == UIInterfaceOrientationLandscapeRight);
}
```

然后，更改`shouldAutorotateToInterfaceOrientation:`方法以匹配之前的版本。一般而言，

苹果公司不建议使用`UIInterfaceOrientationPortraitUpsideDown`，因为如果在电话倒置的情况下接到来电，在接听电话时，电话可能仍然处于倒置状态。

保存nib文件，接下来我们将探讨如何在Interface Builder中设置自动调整属性。

5.1.2 使用自动调整属性设计界面

在Xcode中，展开Resource文件夹，双击`AutoSizeView-Controller.xib`，在Interface Builder中打开该文件。使用自动调整属性的一个好处在于，它们需要的代码极少。我们必须指定要支持的方向，就像在视图控制器中的操作一样，但是实现此技术所需的其他所有工作将在Interface Builder中完成。

要了解如何指定方向，从库中拖出6个Round Rect Button，并将它们放置到视图上，摆放位置如图5-2所示。双击各按钮并分别为它们指定一个标题，以便于区分。我们将按钮从1到6进行编号。

保存工作，并返回到Xcode。现在让我们看一下发生了什么，我们指定了支持的方向，但是还没有设置任何自动调整属性。构建并运行应用程序。当iPhone仿真器出现之后，从**Hardware**菜单中选择**Rotate Left**，这将模拟将iPhone切换到横向模式的情况。参见图5-3。



图5-2 将6个带有编号的按钮添加到界面

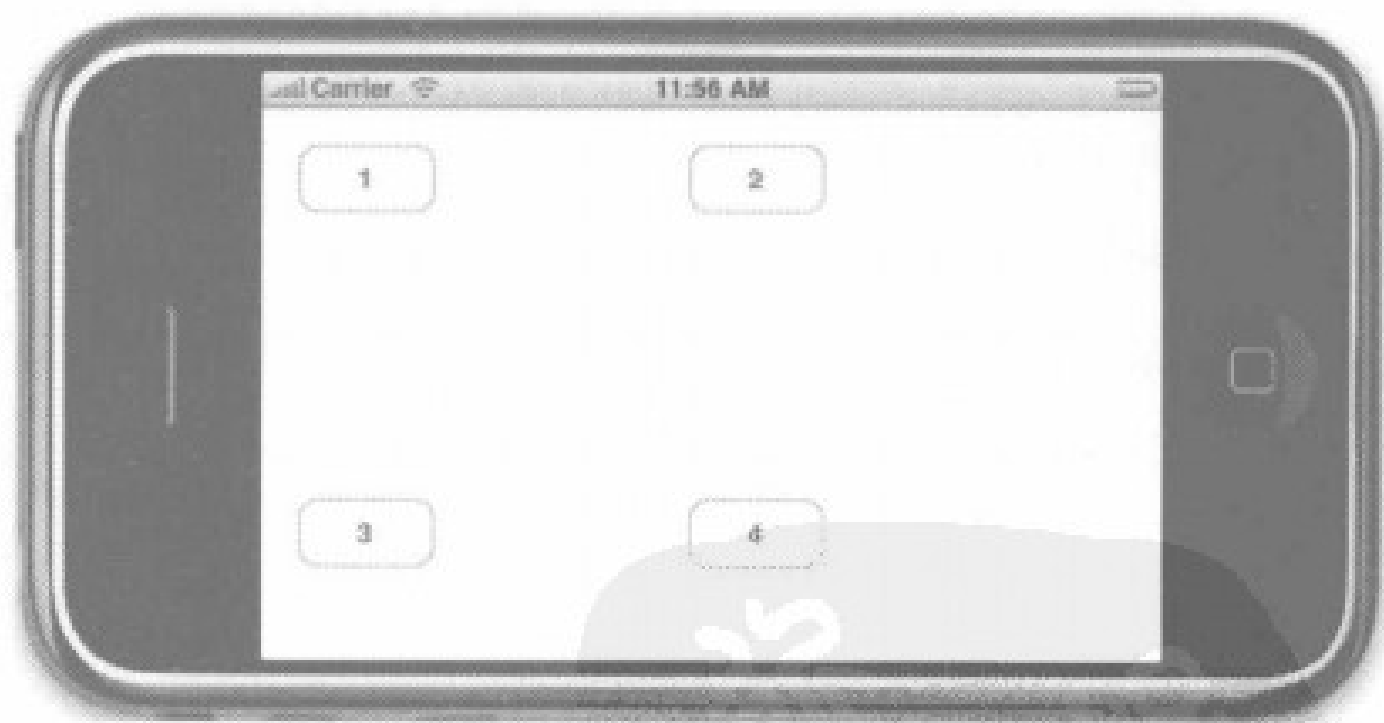


图5-3 显示效果不太理想

在默认情况下，大部分控件都会设置为保持其与屏幕左侧和上侧的相对位置。这种设置适合于某些控件。例如，左上角的按钮（编号为1）可能就处在我们想要的位置，但是其他按钮却不是这样。

退出仿真器，并返回到Interface Builder。

5.1.3 自动调整属性

单击视图左上方的按钮，然后按`⌘3`调出大小检查器，其外观与图5-4类似。

大小检查器用于设置对象的自动调整属性。图5-5显示了大小检查器中控制对象的自动调整属性的部分。

图5-5中左侧的框就是实际设置属性的地方，右侧的框是一个小动画，显示对象在调整大小时的行为方式。在左侧的框中，内部的正方形表示当前的对象。如果选定了某个按钮，则内部的正方形代表该按钮。

内部正方形中的红色箭头表示选定对象内部的水平和垂直空间。单击任意箭头都可将其由实线变为虚线或由虚线变为实线。如果水平箭头是实线，则可在调整窗口大小时自由更改对象的宽度；如果水平箭头是虚线，则iPhone会尽可能将对象的宽度保持为原始值。对象的高度和垂直箭头也是如此。

内部正方形四周的4个红色“I”形表示选定对象的边与包含它的视图的同侧边之间的距离。如果“I”是虚线，那么距离就是灵活可变的；如果“I”是红色实线，则间距的值应尽可能保持不变。

如果实际操作一下，你应该更容易理解。参见图5-5，其中显示了默认的自动调整设置。这些默认设置指定当调整包含对象的视图大小时，对象的大小将保持不变，对象的左边和顶边与视图的对应边之间的距离也保持不变。如果看一下自动调整控件旁边的动画，就会看到对象在调整大小期间的行为方式。注意，当父视图大小发生更改时，内部框相对于父视图左边和顶边的位置保持不变。

现在，单击两个红色的实线“I”形（内部框的上方和左侧），使其变为虚线，如图5-6所示。

将所有线都设置为虚线之后，对象的大小将保持不变，当调整父视图的大小时，对象将移动到父视图的中央。

现在，单击方框内部的垂直箭头以及方框上侧和下侧的“I”形，这样可以得到如图5-7所示的自动调整属性。

使用此配置，可以更改对象的垂直高度，而对象顶部到窗口顶部以及对象底部到窗口底部之间的距离应该保持不变。使用此配置，对象的宽度将不会改变，但是其高度将会改变。多次更改自动调整属性并查看动画，这样可以理解不同设置如何影响在旋转视图和更改其大小时对象的行为方式。

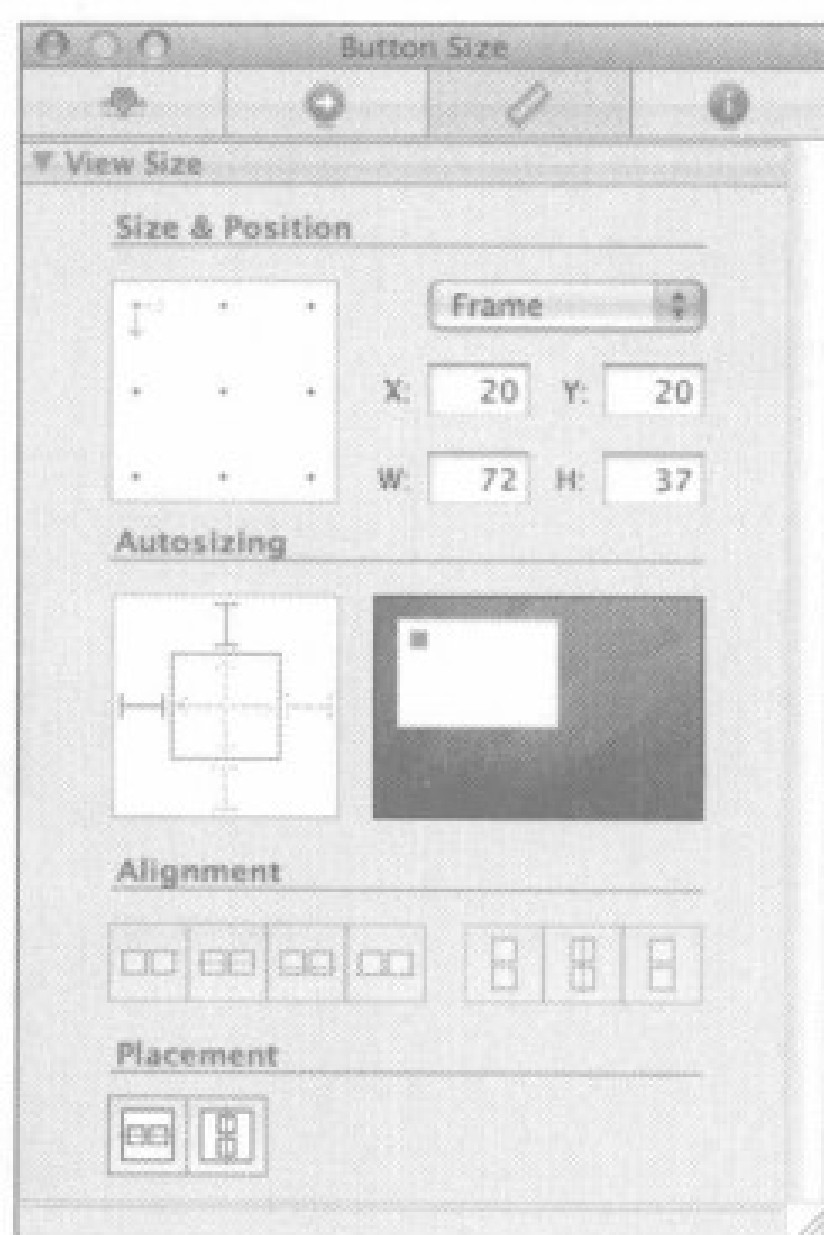


图5-4 大小检查器用于设置对象的自动调整属性

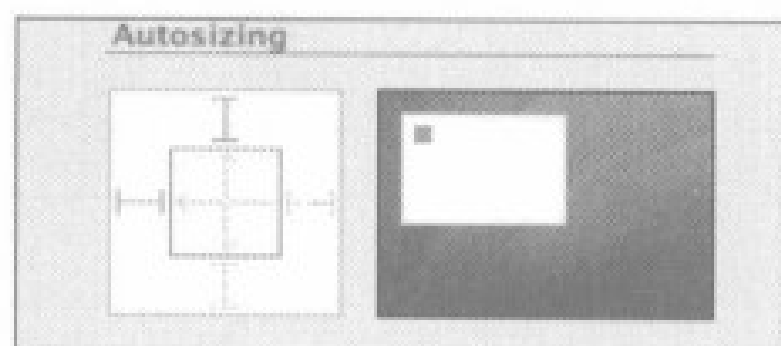


图5-5 大小检查器的Autosizing部分

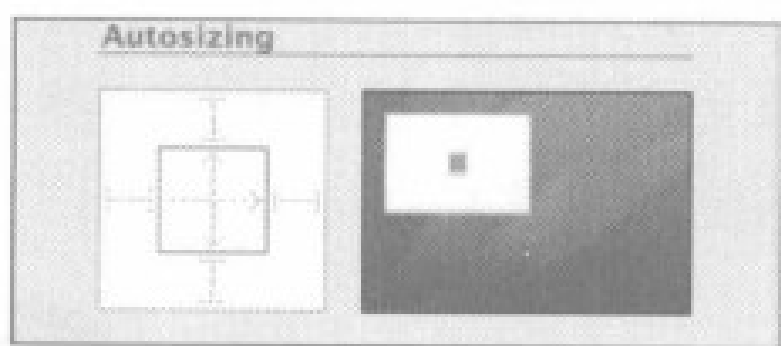


图5-6 将所有线设置为虚线之后，控件将移动到父视图的中央，并且其大小保持不变

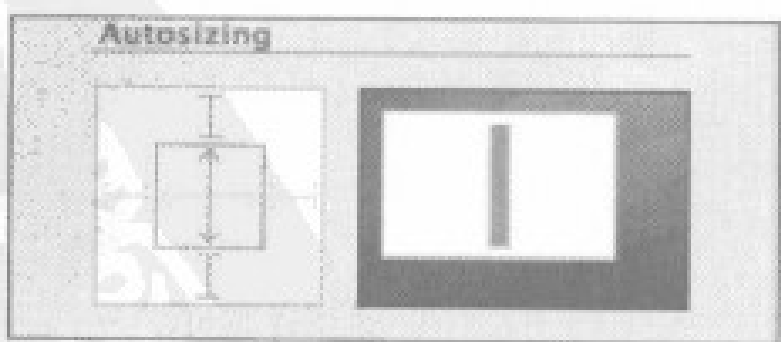


图5-7 此配置支持更改对象的垂直高度

5.1.4 设置按钮的自动调整属性

现在，设置6个按钮的自动调整属性。继续前进，看看能否正确设置这些属性。如果不能进行正确设置，可以看一下图5-8，其中显示了在旋转电话时使每个按钮都出现在屏幕上所需的自动调整属性。

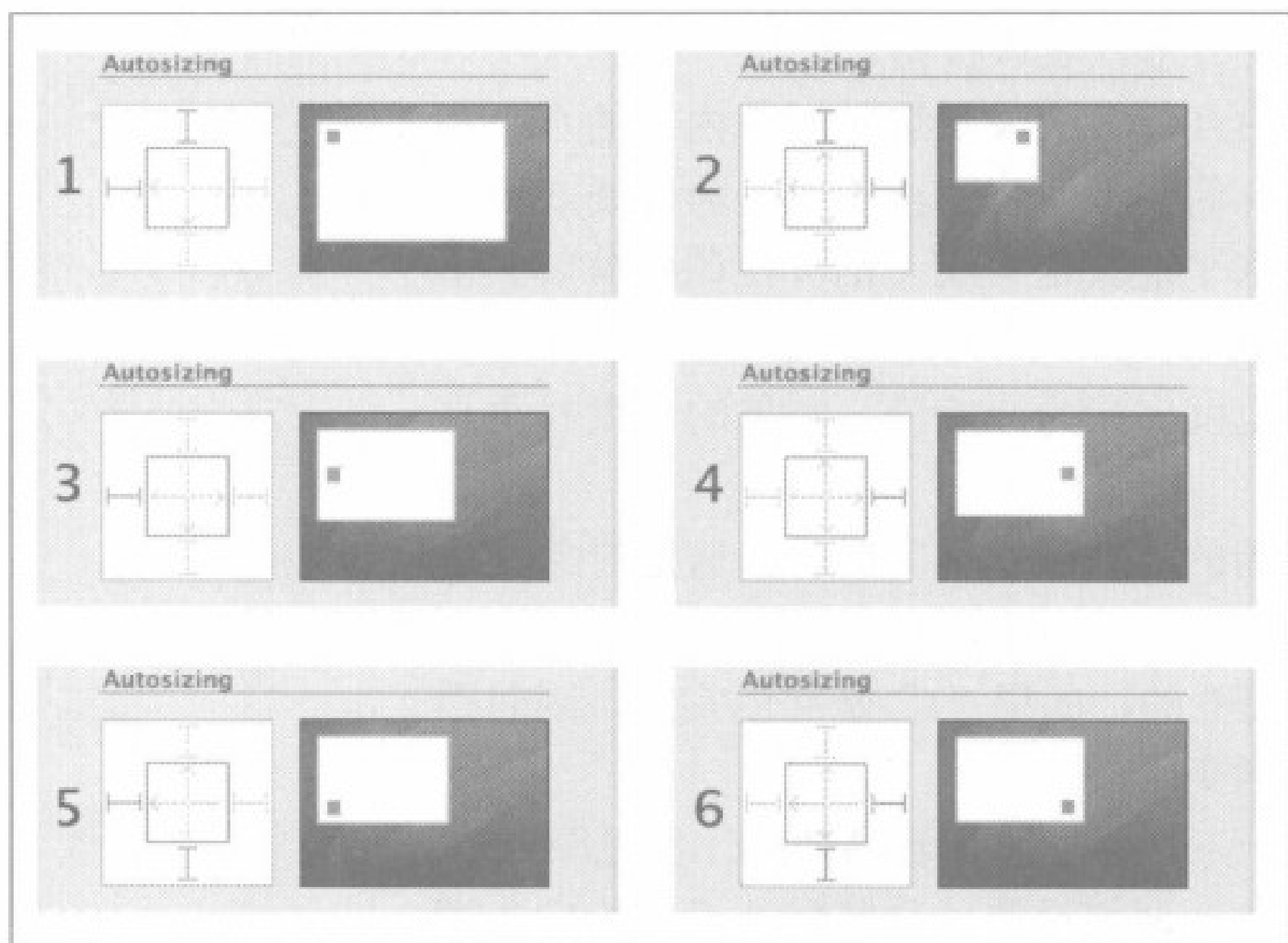


图5-8 所有6个按钮的自动调整属性

按照图5-8设置属性之后，保存nib文件并返回到Xcode，然后构建和运行应用程序。现在，当iPhone仿真器出现时，应该能够从**Hardware**菜单选择**Rotate Left**或**Rotate Right**，而且所有按钮都会在屏幕上显示出来（参见图5-9）。如果将电话旋转回原来的方向，这些按钮应该返回到原来的位置。这种技术适用于许多应用程序。



图5-9 按钮在旋转屏幕之后的新位置

在此示例中，所有按钮的大小都是相同的，因此它们都可见且可以使用，但是屏幕上还存在大量未使用的空白空间。如果支持更改按钮的宽度或高度会更好一些，这样可以减少界面上的空白空间。可以自由调整这6个按钮的自动调整属性，并根据需要添加其他按钮。多次实践之后，

你就会适应自动调整属性的工作方式。

在实践中，你一定会注意到，有时候没有哪种自动调整属性组合能够准确满足需要。有时候，可能需要彻底改变界面的布局，而且此技术可能无法完成任务。对于这些情形，需要编写更多代码。让我们看一下这类情形。

5.2 在旋转时重构视图

在Interface Builder中，单击各按钮，并使用大小检查器将w和h字段更改为125，将按钮的宽度和高度设置为125像素。完成之后，使用蓝色的引导线重新排列按钮，使视图与图5-10类似。

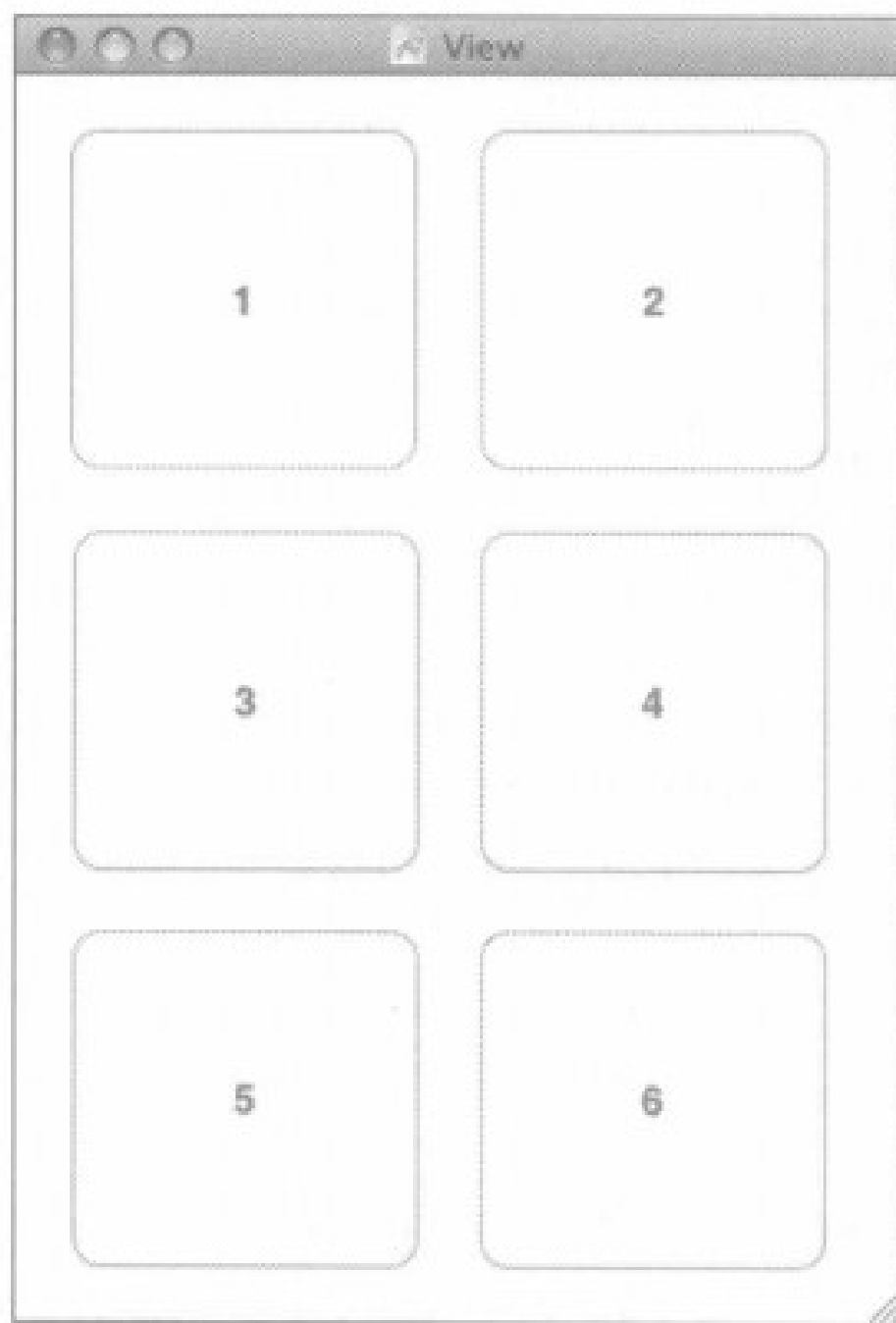


图5-10 调整所有按钮之后的视图

现在，让我们看一下旋转屏幕时会发生什么。如果将按钮的自动调整属性设置回图5-8中所示的设置，那么将产生的结果很可能不是我们想要的。按钮将会彼此重叠，如图5-11所示，因为在横向模式下，屏幕上没有足够的高度来容纳3个高度为125像素的按钮。

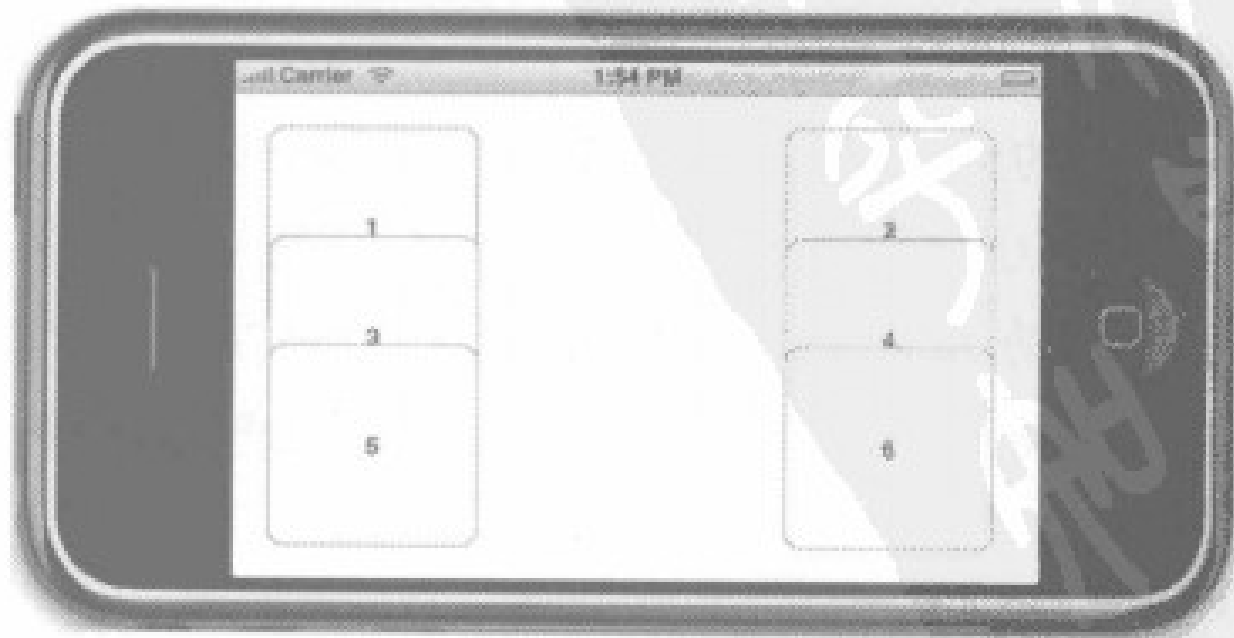


图5-11 得到的结果不理想

我们可以使用自动调整属性来解决此问题，即允许更改按钮的高度。但是这样不能充分利用屏幕空间，因为屏幕中间存在很大的空白空间。如果当界面处于纵向模式时，屏幕能容纳6个正方形按钮，那么在横向模式下也应该能容纳6个正方形按钮，我们只需将这些按钮稍稍移动一下。一种方法是指定每个按钮在旋转视图之后的新位置。

5.2.1 声明和连接输出口

要更改控件的属性，我们需要通过输出口指向要更改的对象。因此，我们需要为6个按钮分别声明一个输出口，以便对它们进行重新排列。将以下代码添加到AutoSizeViewController.h:

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

@interface AutosizeViewController : UIViewController {

    IBOutlet UIButton *button1;
    IBOutlet UIButton *button2;
    IBOutlet UIButton *button3;
    IBOutlet UIButton *button4;
    IBOutlet UIButton *button5;
    IBOutlet UIButton *button6;
}

@property (nonatomic, retain) UIView *button1;
@property (nonatomic, retain) UIView *button2;
@property (nonatomic, retain) UIView *button3;
@property (nonatomic, retain) UIView *button4;
@property (nonatomic, retain) UIView *button5;
@property (nonatomic, retain) UIView *button6;
@end
```

保存此文件，然后返回到Interface Builder。按住Control键并将File's Owner图标拖到每个按钮，将这些按钮与相应的输出口连接。连接了所有按钮之后，保存nib文件，并返回到Xcode。

5.2.2 在旋转时移动按钮

要移动按钮以便充分利用空间，需要覆盖AutoSizeViewController.m中的willAnimateSecondHalfOfRotationFromInterfaceOrientation:duration:。此方法将在旋转开始之后，最后的旋转动画发生之前自动调用。

说明 还可以使用另一个名为willAnimateFirstHalfOfRotationToInterfaceOrientation:duration:的方法，本章稍后将使用它。在该方法中的更改将在旋转动画发生之前完成，该方法是专为应该在旋转动画完全完成之前发生的更改而设计的。在本例中，我们需要按钮在旋转完成的同时移动到它们的新位置，这就是我们选择second-half方法的原因。

添加以下代码，然后看一下这段代码的运行原理：

```
#import "AutosizeViewController.h"

@implementation AutosizeViewController
@synthesize button1;
@synthesize button2;
@synthesize button3;
@synthesize button4;
@synthesize button5;
@synthesize button6;

- (void)willAnimateSecondHalfOfRotationFromInterfaceOrientation:
    (UIInterfaceOrientation)fromInterfaceOrientation
    duration:(NSTimeInterval)duration {

    UIInterfaceOrientation toOrientation= self.interfaceOrientation;

    if (toOrientation== UIInterfaceOrientationPortrait
        || toOrientation== UIInterfaceOrientationPortraitUpsideDown)
    {
        button1.frame = CGRectMake(20, 20, 125, 125);
        button2.frame = CGRectMake(175, 20, 125, 125);
        button3.frame = CGRectMake(20, 168, 125, 125);
        button4.frame = CGRectMake(175, 168, 125, 125);
        button5.frame = CGRectMake(20, 315, 125, 125);
        button6.frame = CGRectMake(175, 315, 125, 125);
    }
    else
    {
        button1.frame = CGRectMake(20, 20, 125, 125);
        button2.frame = CGRectMake(20, 155, 125, 125);
        button3.frame = CGRectMake(177, 20, 125, 125);
        button4.frame = CGRectMake(177, 155, 125, 125);
        button5.frame = CGRectMake(328, 20, 125, 125);
        button6.frame = CGRectMake(328, 155, 125, 125);
    }
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    return (interfaceOrientation == UIInterfaceOrientationPortrait ||
        interfaceOrientation == UIInterfaceOrientationLandscapeLeft ||
        interfaceOrientation == UIInterfaceOrientationLandscapeRight);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [button1 release];
    [button2 release];
}
```



```

[button3 release];
[button4 release];
[button5 release];
[button6 release];
[super dealloc];
}

```

所有视图（包括按钮等控件）的大小和位置都在frame属性中指定，该属性是一个类型为CGRect的结构。CGRectMake是苹果公司提供的一个函数，支持通过指定x和y位置以及width和height来轻松创建CGRect。此代码中唯一可能引起混淆的地方在于，我们实际上获知的不是要旋转到方向，而是旋转之前的原方向。因为我们需要知道新方向，因此忽略了该参数，并使用了继承自UIViewController的interfaceOrientation属性。

保存代码。现在构建并运行代码，以查看其实际效果。尝试旋转电话，观察按钮如何移动到它们的新位置。效果非常不错，但是存在一个稍显怪异的地方。仔细观察将会发现，按钮直接跳转到了新的位置。是否可以通过某种方式让按钮的移动方式更加生动，而不是这么怪异？但是制作动画非常复杂，是这样吗？

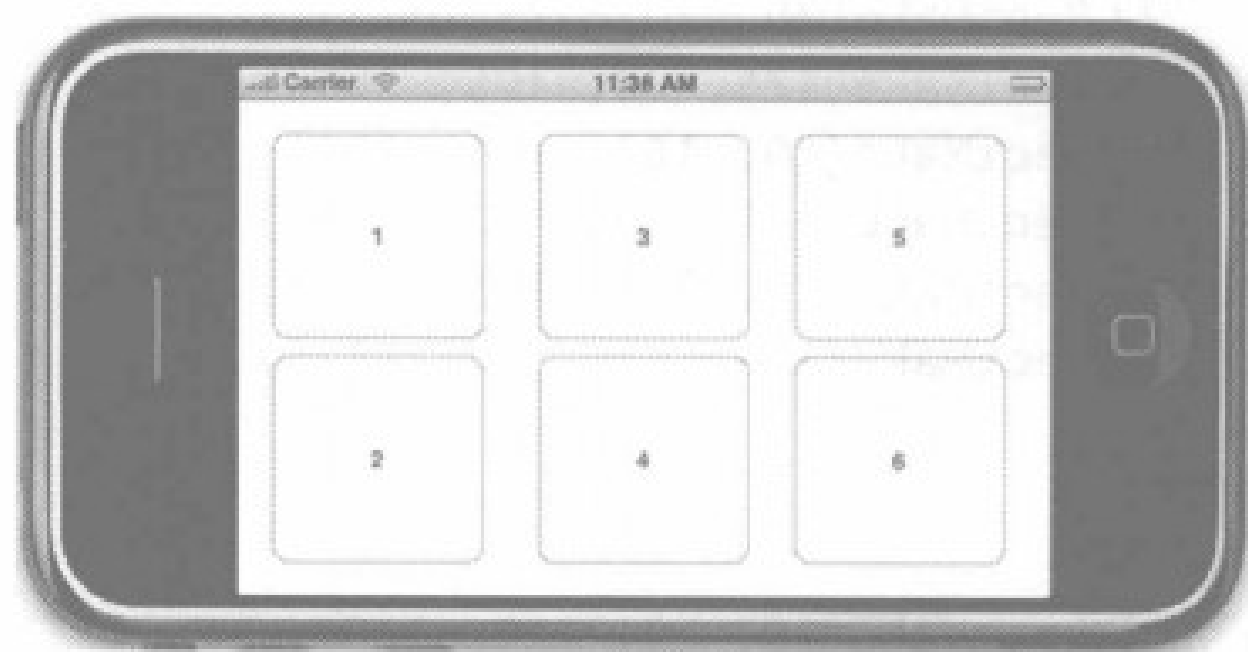


图5-12 更好地利用横向模式下的屏幕

在过去，处理用户界面中的动画非常困难。但是借助Core Animation技术，可以在iPhone上轻松实现各种所需的动画类型。本书不打算直接使用Core Animation，但是Cocoa Touch在后台大量使用了这种技术。在上一章中，将按钮切换到新设置时，就是Core Animation在发挥作用。在本章中，我们可以将这些按钮移动到新位置的过程制作成动画。我们需要将变化过程制作成动画，实现方式是将想要制作成动画的所有更改一起放置在动画块内部。为此，我们在方法中添加了以下两行代码，以指示动画块的开始和结束位置：

```

...
- (void)willAnimateSecondHalfOfRotationFromInterfaceOrientation:
    (UIInterfaceOrientation)fromInterfaceOrientation
    duration:(NSTimeInterval)duration {

    UIInterfaceOrientation toOrientation= self.interfaceOrientation;
    [UIView beginAnimations:@"move buttons" context:nil];

    if (toOrientation== UIInterfaceOrientationPortrait
        || toOrientation== UIInterfaceOrientationPortraitUpsideDown)

```

```

{
    button1.frame = CGRectMake(20, 20, 125, 125);
    button2.frame = CGRectMake(175, 20, 125, 125);
    button3.frame = CGRectMake(20, 168, 125, 125);
    button4.frame = CGRectMake(175, 168, 125, 125);
    button5.frame = CGRectMake(20, 315, 125, 125);
    button6.frame = CGRectMake(175, 315, 125, 125);
}
else
{
    button1.frame = CGRectMake(20, 20, 125, 125);
    button2.frame = CGRectMake(20, 155, 125, 125);
    button3.frame = CGRectMake(177, 20, 125, 125);
    button4.frame = CGRectMake(177, 155, 125, 125);
    button5.frame = CGRectMake(328, 20, 125, 125);
    button6.frame = CGRectMake(328, 155, 125, 125);
}

[UIView commitAnimations];
}
...

```

添加的第一行代码是对一个类方法的调用，用于指示动画块的开始位置。第一个参数是我们分配给动画块的名称。第二个参数供Core Animation的各种委托方法使用。因为我们不会使用这些方法，因此只需传递参数nil。当代码重新放置按钮之后，我们调用另一个类方法commitAnimations，指示动画块的结束位置。这就是整个过程。现在，当运行此新版本，并从Hardware菜单选择Rotate Left时，将会看到在自动旋转结束时，按钮会迅速移动到它们的新位置。

5.3 切换视图

还可以通过另一种方式来处理自动旋转，这种方式可能仅适用于非常复杂的界面。将控件移动到不同位置（就像上一节所做的）是一个非常乏味的过程，特别是对于复杂的界面。那么能否分别设计横向模式和纵向模式视图，然后在旋转电话时在它们之间进行切换呢？

当然，我们可以这么做。但是这种方式比较复杂。当两个视图中的控件可能触发相同的操作时，我们必须提供两个完全不同的输出口集，分别对应两个视图，而且还会增加代码的复杂度。这种复杂度并不是无法克服的，并且在有些情况中，这种方法是最佳选择。让我们尝试一下这种方法。

在Xcode中，使用基于视图的应用程序模板创建一个新项目，下一章将使用其他模板。将此项目命名为Swap。实际上，将在此应用程序中生成的界面还不够复杂，不足以充分施展我



图5-13 启动时的Swap应用程序

们所使用的技术。但是，我们希望确保流程清晰，因此，我们将使用一个非常简单的界面。当编写的这个应用程序启动时，它将处于纵向模式。应用程序中包含两个垂直排列的按钮（参见图5-13）。

当旋转电话时，我们将切换到在横向模式下显示的完全不同的视图。该视图同样包含两个按钮，它们的标签与纵向模式下完全相同（参见图5-14），因此用户不会发现他们看到的是两个不同的视图。

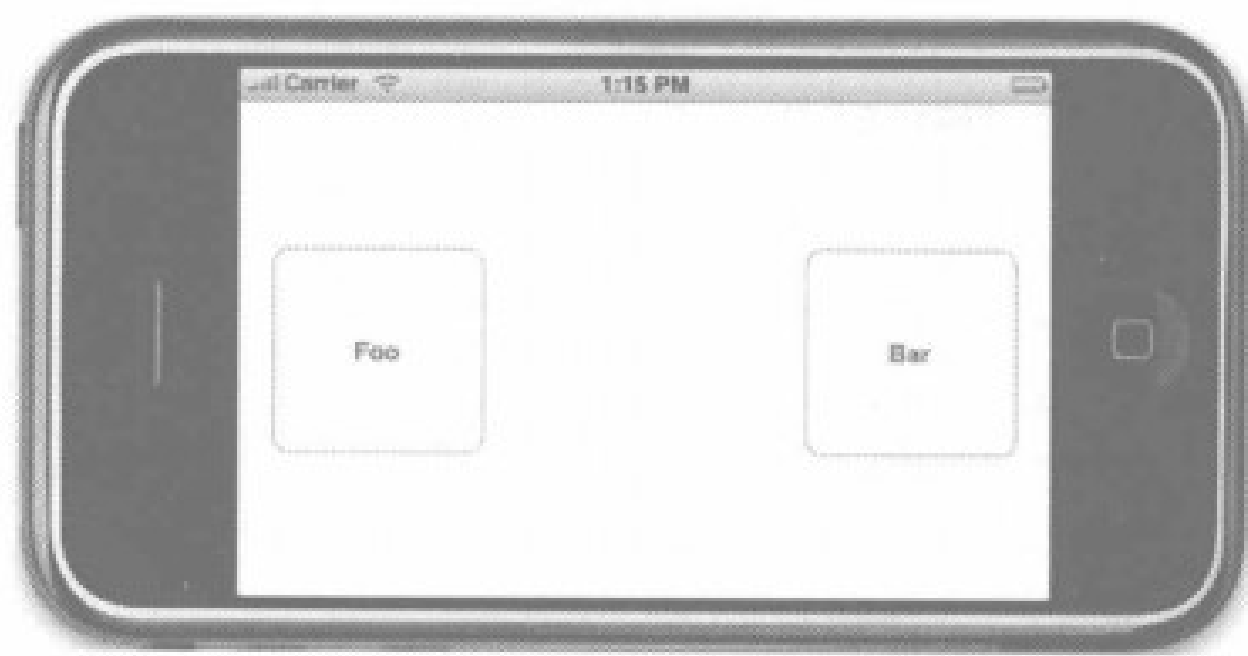


图5-14 两种视图相似，但不同

单击按钮时，它们将被隐藏。这使我们有机会看到处理两个不同的输出口集合的细微差别。在实际应用中，可能有很多时候需要隐藏或禁用某个按钮。例如，你创建的按钮可能会开始一个很长的操作过程，并且你不想让用户在该过程完成之前单击同一个按钮。

5.3.1 确定输出口

因为我们将在每个视图上构建两个按钮，而且一个输出口不能指向多个对象，所以我们需要声明4个输出口，其中两个用于横向视图按钮，其余两个用于纵向视图按钮。在使用此技巧时，需要为输出口指定合适的名称，使其便于阅读，这一点非常重要。

但是，有人可能会问：“我们真的需要为所有这些按钮都提供输出口吗？当我们禁用了被单击的按钮时，能否使用`sender`来代替？”在单视图场景中，使用`sender`就是正确的处理方式。

考虑这样一种情形，如果用户单击Foo按钮之后再旋转电话，会发生什么情况？另一个视图上的Foo按钮是一个完全不同的按钮，并且它仍然处于活动状态，这不是我们想要的行为方式。我们不想告知用户，他们现在处理的对象不是刚才处理的对象。

除了按钮的输出口之外，我们还需要两个输出口来指向两个不同的视图版本。当只有一个视图时，我们只需使用父类的`view`属性。但是，因为我们需要在运行时更改`view`的值，所以需要确保能够获得两种视图，因此需要两个`UIView`输出口。

5.3.2 确定动作

按钮需要触发相应的动作，因此我们至少需要一个动作方法。我们将设计一个动作方法来处理按下任何按钮的操作，因此，我们仅在视图控制器类中声明了一个`buttonPressed:`动作。

5.3.3 声明动作和输出口

将以下代码添加到SwapViewController.h，创建使用Interface Builder时所需的输出口。

```
#import <UIKit/UIKit.h>
#define degreesToRadian(x) (M_PI * (x) / 180.0)

@interface SwapViewController : UIViewController {
    IBOutlet    UIView    *landscape;
    IBOutlet    UIView    *portrait;

    // Foo
    IBOutlet    UIButton *landscapeFooButton;
    IBOutlet    UIButton *portraitFooButton;

    // Bar
    IBOutlet    UIButton *landscapeBarButton;
    IBOutlet    UIButton *portraitBarButton;
}
@property (nonatomic, retain) UIView *landscape;
@property (nonatomic, retain) UIView *portrait;
@property (nonatomic, retain) UIButton *landscapeFooButton;
@property (nonatomic, retain) UIButton *portraitFooButton;
@property (nonatomic, retain) UIButton *landscapeBarButton;
@property (nonatomic, retain) UIButton *portraitBarButton;

-(IBAction)buttonPressed:(id)sender;
@end
```

下面这行代码：

```
#define degreesToRadian(x) (M_PI * x / 180.0)
```

是一个宏，用于在度数和弧度之间进行转换。稍后在调用需要输入弧度值的函数时将使用这个宏。大多数人（包括我们）都不习惯使用弧度，因此，这个宏支持以度数（而不是弧度）的方式来指定角度，从而使代码更具可读性。

你应该熟悉这个头文件中的所有内容。现在我们实现了所需的输出口，接下来转到Interface Builder，并构建所需的两个视图。在Groups & Files窗格的Resources文件夹中双击SwapViewController.xib，在Interface Builder中打开该文件。

5.3.4 设计两个视图

希望你对刚才在Interface Builder中看到的内容很熟悉。在进行任何操作之前，都要找到View图标并生成一个副本，这样，我们的nib文件中就有两个视图。对于本示例，我们不需要作为模板一部分提供的View图标，因为它的大小无法更改，于是单击View图标，并按下delete按钮。接下来，从库中拖出两个UIView。完成之后，你将拥有两个名为View的图标。这可能容易使人混淆，因此我们将对它们进行重命名，以便于区分。

要重命名nib的主窗口中的图标，需要单击视图将其选中，等待几秒钟，然后单击图标的名称。等待图标的名称变为可编辑状态，然后可以输入新名称。注意，此技巧仅适用于图标视图模式。将两个视图分别命名为Portrait和Landscape。

现在，按住Control键并将File's Owner图标拖到Portrait图标，在弹出灰色菜单时选择portrait输出口。然后，再次按住Control键，并将File's Owner图标拖到Landscape图标，选择landscape输出口。现在，再次按住Control键，并将File's Owner拖到Portrait，选择View输出口，指定应该在启动时显示的视图。

双击Landscape图标，按下⌘3调出大小检查器。现在，此视图的高和宽应该分别为460像素和320像素。将高和宽分别更改为300像素和480像素。现在，从库中拖出两个Round Rect Button到Landscape视图中。按钮的准确大小和位置无关紧要，我们将其设置为125像素宽和125像素高。双击左侧的按钮，然后将其命名为Foo，再双击右侧的按钮，并将其命名为Bar。

按住Control键并将File's Owner图标拖到Foo按钮，将Foo按钮分配给landscapeFoo输出口，然后对Bar按钮执行相同操作，将其分配给landscapeBar输出口。现在，单击Foo按钮，按下⌘2切换到连接检查器。将事件中表示触摸屏幕操作的圆拖到File's Owner图标，然后选择buttonPressed:动作。对Bar按钮重复此过程，这样，两个按钮都能够触发buttonPressed:动作方法。现在可以关闭Landscape窗口了。

双击Portrait图标，打开该视图进行编辑。从库中拖出两个Round Rect Button，这次将它们垂直排列。双击顶部的按钮，将其命名为Foo。然后，双击底部的按钮，将其命名为Bar。按住Control键并将File's Owner图标拖到Foo按钮，将其分配给portraitFooButton输出口。再次按住Control键，并将File's Owner图标拖到Bar按钮，将其分配给portraitBarButton输出口。单击Foo按钮，将连接检查器上的Touch Up Inside事件拖到File's Owner图标，并选择buttonPressed:动作。对Bar按钮重复此连接操作。

保存nib文件，并返回到Xcode。

5.3.5 实现交换和动作

现在已经完成了大部分工作，我们只需编写代码来处理交换过程和点击按钮操作。将以下代码添加到SwapViewController.m文件。

说明 在此代码清单中，我们删除了已有的代码。要删除的代码将带有一条删除线。

```
#import "SwapViewController.h"

@implementation SwapViewController
@synthesize landscape;
@synthesize portrait;
@synthesize landscapeFooButton;
@synthesize portraitFooButton;
@synthesize landscapeBarButton;
@synthesize portraitBarButton;
```

```
- (void)willAnimateFirstHalfOfRotationToInterfaceOrientation:
    (UIInterfaceOrientation)toInterfaceOrientation
    duration:(NSTimeInterval)duration {

    if (toOrientation== UIInterfaceOrientationPortrait)
    {
        self.view = self.portrait;
        self.view.transform = CGAffineTransformIdentity;
        self.view.transform =
            CGAffineTransformMakeRotation(degreesToRadian(0));
        self.view.bounds = CGRectMake(0.0, 0.0, 300.0, 480.0);
    }
    else if (toOrientation== UIInterfaceOrientationLandscapeLeft)
    {
        self.view = self.landscape;
        self.view.transform =
            CGAffineTransformIdentity;
        self.view.transform =
            CGAffineTransformMakeRotation(degreesToRadian(-90));
        self.view.bounds = CGRectMake(0.0, 0.0, 460.0, 320.0);
    }
    else if (toOrientation== UIInterfaceOrientationPortraitUpsideDown)
    {
        self.view = self.portrait;
        self.view.transform = CGAffineTransformIdentity;
        self.view.transform =
            CGAffineTransformMakeRotation(degreesToRadian(180));
        self.view.bounds = CGRectMake(0.0, 0.0, 300.0, 480.0);
    }
    else if (toOrientation== UIInterfaceOrientationLandscapeRight)
    {
        self.view = self.landscape;
        self.view.transform = CGAffineTransformIdentity;
        self.view.transform =
            CGAffineTransformMakeRotation(degreesToRadian(90));
        self.view.bounds = CGRectMake(0.0, 0.0, 460.0, 320.0);
    }
}

- (IBAction)buttonPressed:(id)sender {

    if (sender == portraitFooButton || sender == landscapeFooButton)
    {
        portraitFooButton.hidden = YES;
        landscapeFooButton.hidden = YES;
    }
    else
    {
        portraitBarButton.hidden = YES;
    }
}
```



```

        landscapeBarButton.hidden = YES;
    }

}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
    return YES;
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [landscape release];

    [portrait release];
    [landscapeFooButton release];
    [portraitFooButton release];
    [landscapeBarButton release];
    [portraitBarButton release];

    [super dealloc];
}

@end

```

新代码中的第一个方法称为 `willAnimateFirstHalfOfRotationToInterfaceOrientation:duration:`。这个方法来自我们覆盖的一个超类，这个超类在旋转开始之后与旋转实际发生之前被调用。我们在此方法中采用的动作将是旋转动画的前半部分。

在此方法中，我们看一下旋转的目标方向，并根据新方向将 `view` 属性设置为 `landscape` 或 `portrait`。然后调用 `CGAffineTransformMakeRotation`（Core Graphics 框架的一部分）来创建一个旋转变换。变换是对对象大小、位置或角度的更改的数学描述。一般情况下，iPhone 在旋转电话时自动设置变换值。但是在这里，当切换到新视图时，我们必须确保为其提供了正确的值，以不至于使 iPhone 混淆。这就是 `willAnimateFirstHalfOfRotationToInterfaceOrientation:duration:` 方法在每次设置视图的 `transform` 属性时所做的工作。旋转视图之后，调整其框架，使其与处于当前方向的窗口完美结合。

接下来是 `buttonPressed:` 方法，该方法并没有太多令人惊奇的地方。我们按下一个按钮，隐藏它，然后在另一个视图中隐藏相应的按钮。

你应该熟悉我们在此类中编写的所有其他内容。新的 `shouldAutorotateToInterfaceOrientation:` 方法返回 YES，告诉 iPhone 我们支持旋转到任何方向，添加到 `dealloc` 方法中的代码用于清除内存。

现在，我们是否可以编译应用程序并使用它？还不行。在编译之前，我们还要做一件事情。请在继续之前务必先保存代码。

5.3.6 链接 Core Graphics 框架

前面已经提到，`CGAffineTransformMakeRotation`函数是Core Graphics框架的一部分。但是，在默认情况下，Core Graphics框架未包含在Xcode项目中。我们需要手动将该框架链接到项目中。使用项目窗口中Groups & Files窗格里的Frameworks文件夹完成此任务。

现在，我们如何知道需要链接到Core Graphics框架？在Xcode中，从**Build**菜单中选择**Build**，然后会看到`CGAffineTransformMakeRotation`出现了一个链接错误。我们知道，“CD”指的是“Core Graphics”，但是你并非每次都知哪个框架包含链接错误中报告的方法。这里有一个技巧可帮助找到出现问题的函数的框架。

首先，在源代码中找到`CGAffineTransformMakeRotation`调用。在我们的示例中，它位于`SwapViewController.m`中。双击函数名称，并将光标停留在选定函数上面。稍待片刻，函数名称右侧将出现一个下拉菜单箭头。将光标向右滑动到箭头处并单击它。从出现的上下文菜单选择**Jump to Definition**。

然后，我们将转到定义该函数或方法的头文件。一般而言，头文件顶部将指示它所属的框架。以下是定义`CGAffineTransformMakeRotation`的头文件顶部：

```
/* CoreGraphics - CGAffineTransform.h
 * Copyright (c) 1998-2008 Apple Inc.
 * All rights reserved. */

#ifndef CGAFFINETRANSFORM_H_
#define CGAFFINETRANSFORM_H_

typedef struct CGAffineTransform CGAffineTransform;

#include <CoreGraphics/CGBase.h>
#include <CoreGraphics/CGGeometry.h>

struct CGAffineTransform {
    ...
}
```

这些头文件中的第一部分注释通常包含头文件的文件名和所属框架。从上述内容中可以看出，`CGAffineTransform.h`是Core Graphics的一部分，因此我们知道，这就是我们要添加的框架。知道需要链接的库或框架之后，下一步是找到它，以便将其添加到项目窗口的Frameworks文件夹。

在Groups & Files窗格中单击Frameworks文件夹，从Project菜单中选择Add to Project。这将打开标准的Open File对话框。我们现在需要导航到正确的Core Graphics框架版本：适用于iPhone仿真器的版本。

要找到所需的框架版本，我们需要查看Mac的硬盘上的/Developer文件夹。该文件夹包含一个Platforms文件夹，而`iPhoneSimulator.platform`就在Platforms文件夹中。但是，找到该文件夹还不

够。

在iPhoneSimulator.platform文件夹内有一个名为Developer的文件夹，Developer文件夹包含SDKs文件夹。在SDKs文件夹内是iPhontSimulator2.1.sdk文件夹，其中又包含一个System文件夹。在System文件夹内是Library文件夹，而Library文件夹内是Frameworks文件夹。在Frameworks文件夹内是一个名为CoreGraphics.framework的框架，这就是我们需要添加的框架。

如果你在自己的计算机上执行这些步骤，则会注意到CoreGraphics.framework不仅仅是一个文件。在此框架下有许多项可供选择。不要继续展开文件夹，只需选择CoreGraphicsframework并单击Add按钮。

在本书中，我们将在许多项目中添加框架。你可能想在此处贴上一个书签，以便在以后添加框架时可以参考本小节提供的文件路径。你也可以折上本页的一角。

在项目中添加图像时，也会出现相同的对话框（参见图5-15）。在这里，与添加图片相比，选择正确的选项要重要得多。我们不想将该框架添加到项目中，因为我们只是进行链接，因此需要确保没有选中Copy items into destination group's folder (if needed)复选框。更重要的是，我们需要确保选择了正确的Reference Type，那就是Relative to Current SDK。此选项将支持在更改所使用的SDK时更改选定的框架。iPhone和iPhone仿真器都有自己的SDK，我们将在为iPhone自身构建应用程序时更改SDK。选定此选项之后，更改SDK将会自动更改我们链接到的框架版本。

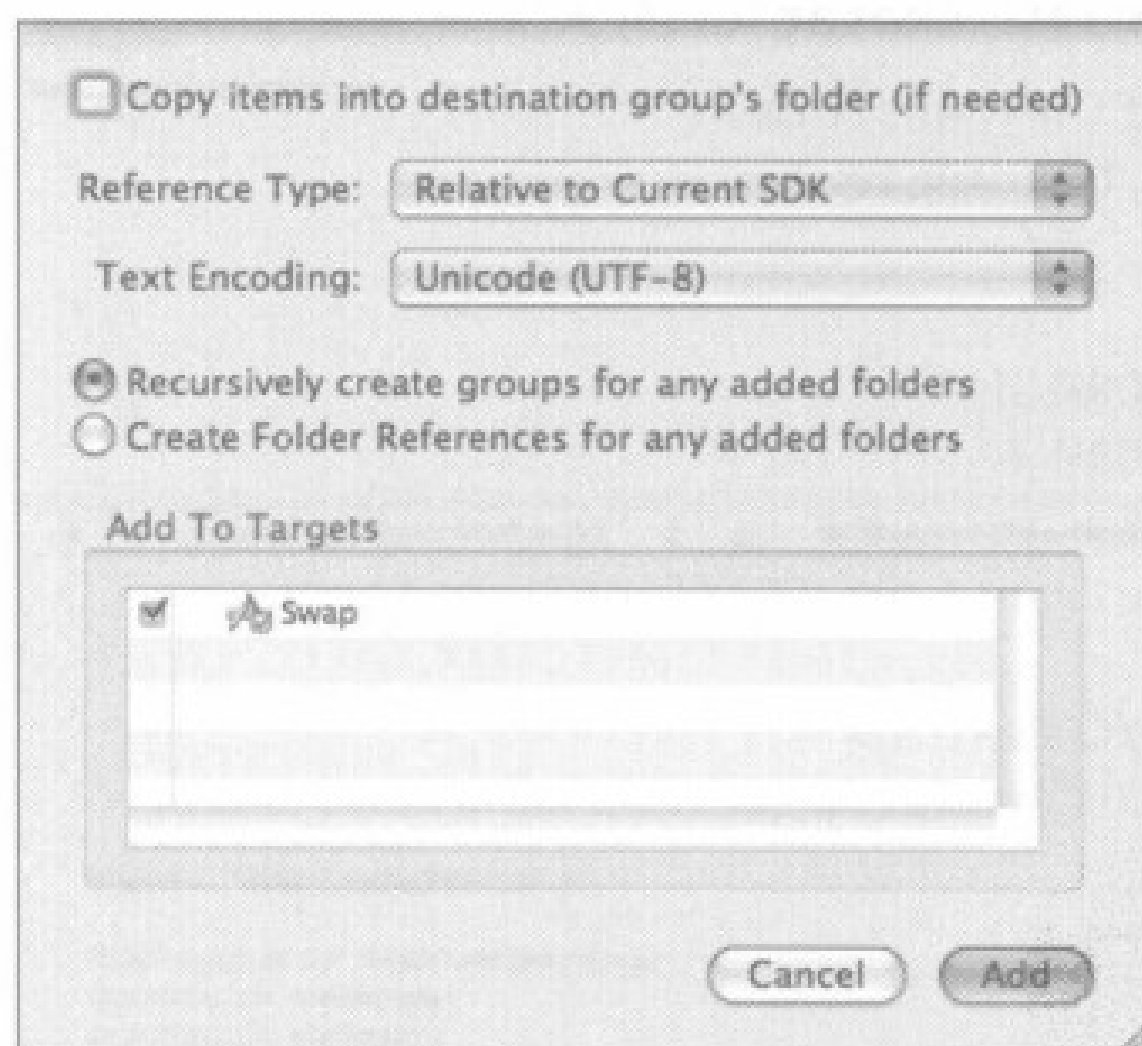


图5-15 添加资源对话框

说明 非常棒！如果你申请并加入了苹果公司的iPhone Developer Program，那么你想要做的第一件事就是构建应用程序，并在iPhone或iPod上运行它们。很棒的地方在于，在更改平台时无需更改框架。Xcode将自动进行正确操作。你只需在将框架添加到项目时，确保选择了Relative to Current SDK。

当出现如图5-15所示的对话框时，选择Add按钮，将项目链接到Core Graphics框架。现在可以编译并运行项目了。

提示 添加框架的一种更安全的方式是，右键单击Groups & Files窗格中的Resources。这样会弹出一个上下文菜单，其Add子菜单中包含一个Existing Frameworks...选项。选择此选项与从Project菜单选择Add to Project的效果完全一样，但此选项知道你不想将框架复制到项目中。

当项目运行时，仿真器窗口应该以纵向模式打开，且其中包含两个大按钮。单击顶部的按钮，该按钮将会消失。现在将电话更改为横向模式，它将切换到另一个视图。在纵向视图上单击的按钮在风景视图上应该仍然是隐藏的，即使它们不是同一个按钮。

棒极了！

注意，如果不小心单击了两个按钮，那么显示它们的唯一方式是退出仿真器并重新运行项目。不要在你自己的应用程序中使用此方法。

5.4 小结

本章介绍了在应用程序中支持自动旋转的3种完全不同的方法。你学习了自动调整属性，以及如何编写代码在旋转电话时重构视图。我们探讨了在旋转电话时如何在两个完全不同的视图之间切换，还学习了如何将新框架链接到应用程序中。

本章首次尝试了在一个应用程序中使用多个视图，在同一个nib文件中的两个视图之间进行切换。下一章将介绍真实的多视图应用程序。目前为止，我们编写的每个应用程序都仅使用了一个视图控制器，并且除最后一个应用程序以外，所有应用程序都仅使用了一个内容视图。然而，许多复杂的iPhone应用程序（比如Mail和Contracts）必须使用多个视图和视图控制器才能实现，我们将在第6章介绍如何使用多个视图和视图控制器。



目前为止，我们编写的应用程序都只有一个视图控制器。尽管使用一个视图可以实现许多功能，但是iPhone平台的真正强大之处在于可以根据用户输入切换视图。多视图应用程序具有各种不同的风格，但是它们的底层机制都是相同的，无论它们在屏幕上的显示方式如何。

严格来讲，我们在之前的应用程序中处理过多个视图，因为按钮、标签和其他控件都是UIView的子类，都是类层次结构的一部分。但是当苹果公司在文档中使用术语“视图”时，它通常指的是具有相应类控制器的UIView子类。这些视图类型有时也被称为内容视图，因为它们是应用程序内容的主要容器。

utility应用程序是最简单的多视图应用程序。utility应用程序主要使用一个视图，但是它还提供了一个视图用于配置应用程序或提供除主视图之外的更多信息。iPhone附带的Stocks应用程序就是一个很好的例子（参见图6-1）。单击右下角的小i图标可以弹出第二个视图，用于配置应用程序所跟踪的股票列表。



图6-1 iPhone随带的Stock应用程序包含两个视图，一个用于显示数据，另一个用于配置股票列表

iPhone还随带了几个标签栏应用程序，比如Phone应用程序（参见图6-2）和Clock应用程序。标签栏应用程序是一种多视图应用程序，它在屏幕底部显示了一行按钮。点击某个按钮就会激活一个新的视图控制器，并显示一个新视图。例如，在Phone应用程序中，点击Contacts时显示的视图与点击Keypad时显示的视图不同。

说明 标签栏和工具栏容易使人混淆。标签栏用于从两个或多个选项选择一个选项，且仅能选择一个选项。工具栏可以包含按钮和其他一些控件，但这些项并不是相互排斥的。在实际的应用程序中，标签栏几乎总是用于在两个或多个内容视图之间进行切换，而工具栏通常用于显示完成常见任务所需的按钮。

另一个常见的多视图iPhone应用程序类型是基于导航的应用程序，这类应用程序使用导航栏控制器向用户提供分层信息。Mail应用程序就是一个很好的例子（参见图6-3）。在Mail中，第一个视图是一个邮件账户列表。触摸某个邮件账户，将会进入一个文件夹列表。触摸某个文件夹将会显示该文件夹中的电子邮件，触摸电子邮件就会显示该邮件的内容。基于导航的应用程序非常适合用于提供分层信息。

由于视图在本质上是分层的，因此甚至可以在一个应用程序中结合使用不同的视图交换机制。例如，iPhone的iPod应用程序使用标签栏来切换组织音乐的不同方法，使用导航栏控制器来支持基于所选方法浏览音乐（参见图6-4）。



图6-2 Phone应用程序是使用标签栏的多视图应用程序的例子



图6-3 iPhone Mail应用程序是使用导航栏的多视图应用程序的例子



图6-4 iPod应用程序同时使用了导航栏和标签栏

所有这些多视图应用程序都使用UIKit提供的特定控制器类。标签栏界面使用UITabBar-

Controller类来实现，而导航界面使用UINavigationController实现。本章将从头构建一个多视图应用程序，主要介绍多视图应用程序的结构，以及交换内容视图的基础知识。我们将编写自定义的多视图控制器，该控制器将是充分利用苹果公司提供的各种多视图控制器的强大基础。

6.1 View Switcher 应用程序

本章将构建的View Switcher应用程序在外观上非常简单，但是从将要编写的代码上讲，它是目前为止我们碰到的最复杂的应用程序。View Switcher由3个不同的控制器、3个nib文件和1个应用程序委托组成。

在首次启动时，View Switcher将与图6-5类似，屏幕底部包含一个工具栏，工具栏中仅包含一个按钮。视图的其余部分包括一个蓝色的背景和一个等待按下的按钮。

当按下Switch Views按钮时，背景将会变为黄色，按钮的名称将会更改（参见图6-6）。

无论按下Press Me还是Press Me, Too按钮，都会弹出一个警告，指示按下了哪个视图的按钮（参见图6-7）。

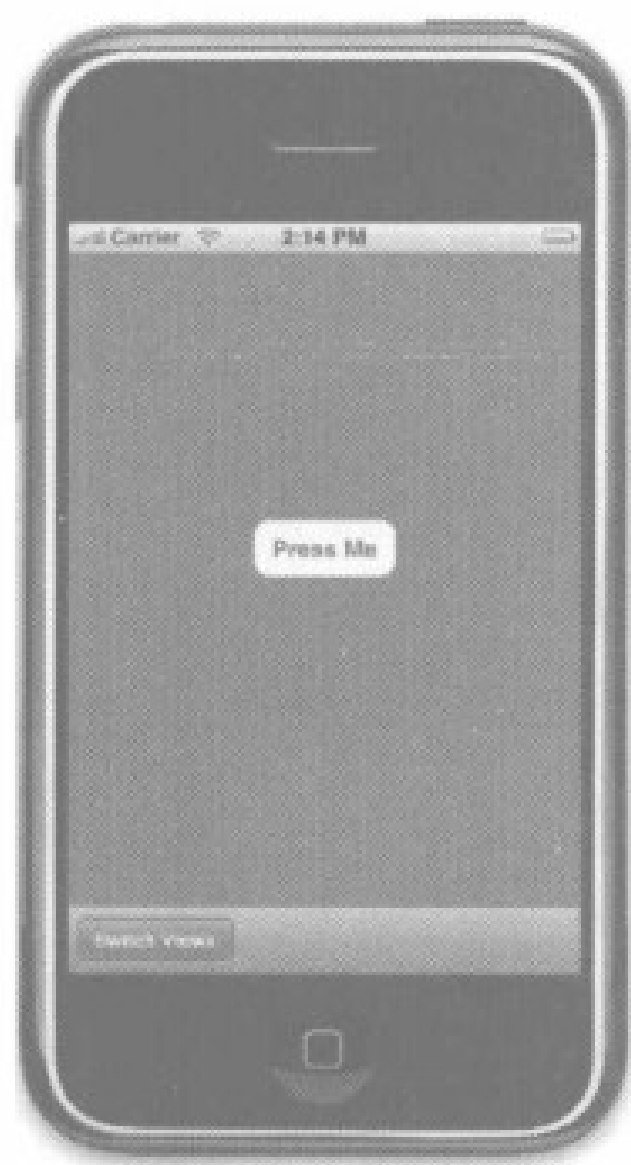


图6-5 启动之后的View Switcher

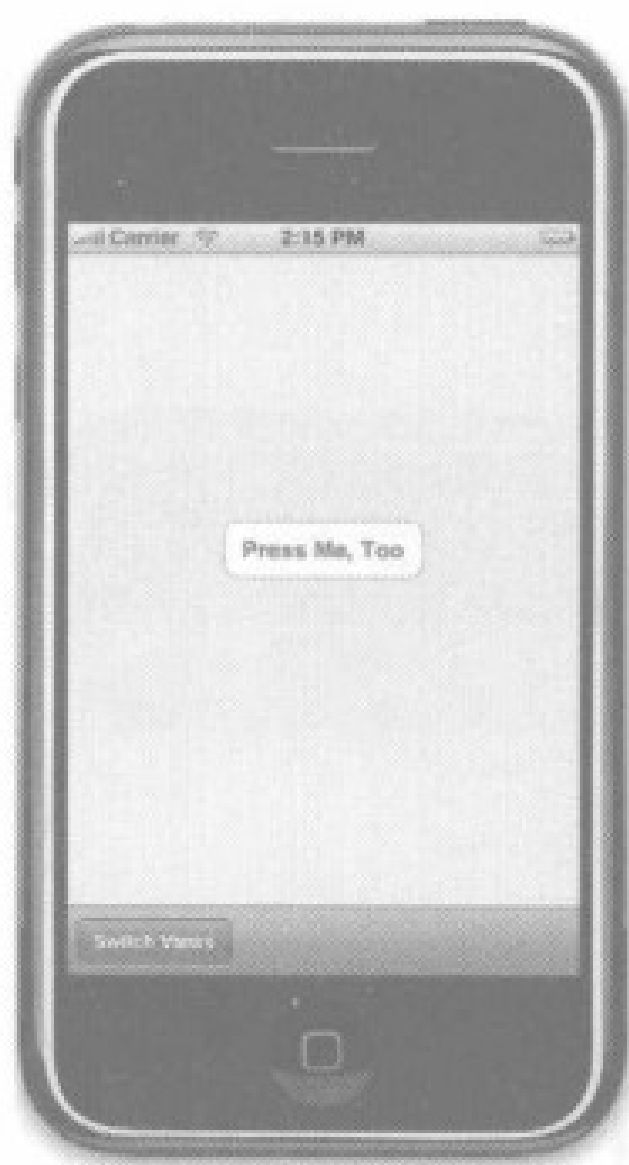


图6-6 按下Switch Views 按钮之后

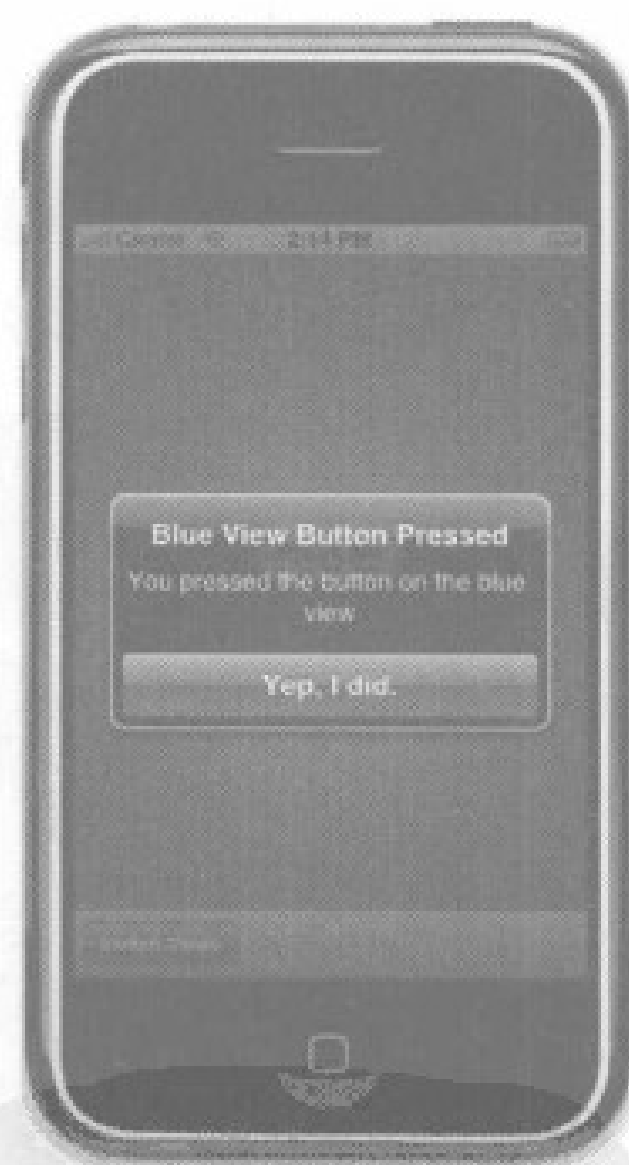


图6-7 按下中央的按钮将会显示一个警告

尽管可以编写一个单视图应用程序来实现相同的功能，但我们采用这种比较复杂的方法来演示多视图应用程序的机制。在这个简单的应用程序中，实际上有3个视图控制器在交互，1个控制器控制蓝色视图，一个控制黄色视图，而第三个特殊的控制器用于在按下Switch Views按钮时在这两个视图之间切换。

6.2 多视图应用程序的体系结构

在开始构建应用程序之前，我们先来了解一下iPhone多视图应用程序的组成方式。几乎所有

多视图应用程序都使用相同的基本模式。

在这里，nib文件扮演着重要角色。创建项目之后，你可以在项目窗口的Resources文件夹中找到MainWindow.xib。在该文件中，除了应用程序委托和应用程序的主窗口之外，还有一个控制器类实例，它负责管理当前向用户显示哪个视图。这个根控制器通常是一个UINavigationController或UITabBarController实例，但也可以是UIViewController的自定义子类。根控制器的任务是获取两个或更多其他视图，并根据用户输入向用户提供适当的视图。例如，标签栏控制器会根据最后点击的标签栏项，在不同的视图和视图控制器之间进行切换。当用户浏览分层数据时，导航控制器也具备相同的功能。

6.2.1 多视图控制器也是视图控制器

需要重点注意，每个多视图控制器都是一个视图控制器。甚至，已提供的多视图类UITabBarController和UINavigationController都是UIViewController的子类，并且能够执行其他视图控制器能够执行的所有工作。根控制器是应用程序的主要视图控制器，也是指定是否应该自动旋转到新方向的视图。

在多视图应用程序中，大部分屏幕都由一个内容视图组成，而每个内容视图都有自己的控制器以及输出口和动作。例如，在标签栏应用程序中，触摸标签栏所生成的事件将会转到标签栏控制器中，但是屏幕上其他位置发生的事件将会转到与当前显示的内容视图相对应的控制器中。

6.2.2 内容视图剖析

每个视图控制器（包括多视图控制器）都控制一个内容视图，应用程序的用户界面就是在这些内容视图中构建的。每个内容视图通常由2个或3个部分组成：视图控制器、nib文件以及一个可选的UIView子类。使用多视图模板创建新Xcode项目之后，始终可以看到所有这3个组件的文件。控制器和nib文件通常是必需的。尽管可以不使用nib文件，而在代码中创建界面，但很少有人选择这种方法，因为这种方法更加耗时且难以维护。

每个内容视图还有一个UIView子类，但是并不是始终都需要它。实际上，一般情况都不需要UIView子类，但是当需要更改内容视图的外观或行为，而使用Interface Builder中的属性检查器不能完成任务时，就需要使用这个子类。本章将为每个内容视图创建一个nib和一个控制器类。

要间接创建内容视图，可以实例化它的控制器类，并指定一个nib名称。nib通常使用控制器类的名称来命名。因此，名为MyController的类通常会加载一个名为MyController.xib的nib文件。

当应用程序加载MainWindow.xib时，也就创建了根控制器。

6.3 构建 View Switcher

你一定等不及想要使用除基于视图的应用程序之外的Xcode模板了。当然，你的耐心等待很快就会获得回报！从File菜单中选择New Project...或者按下⌘N。打开帮助窗口之后，选择Window-Based Application（参见图6-8），然后输入项目名称View Switcher。

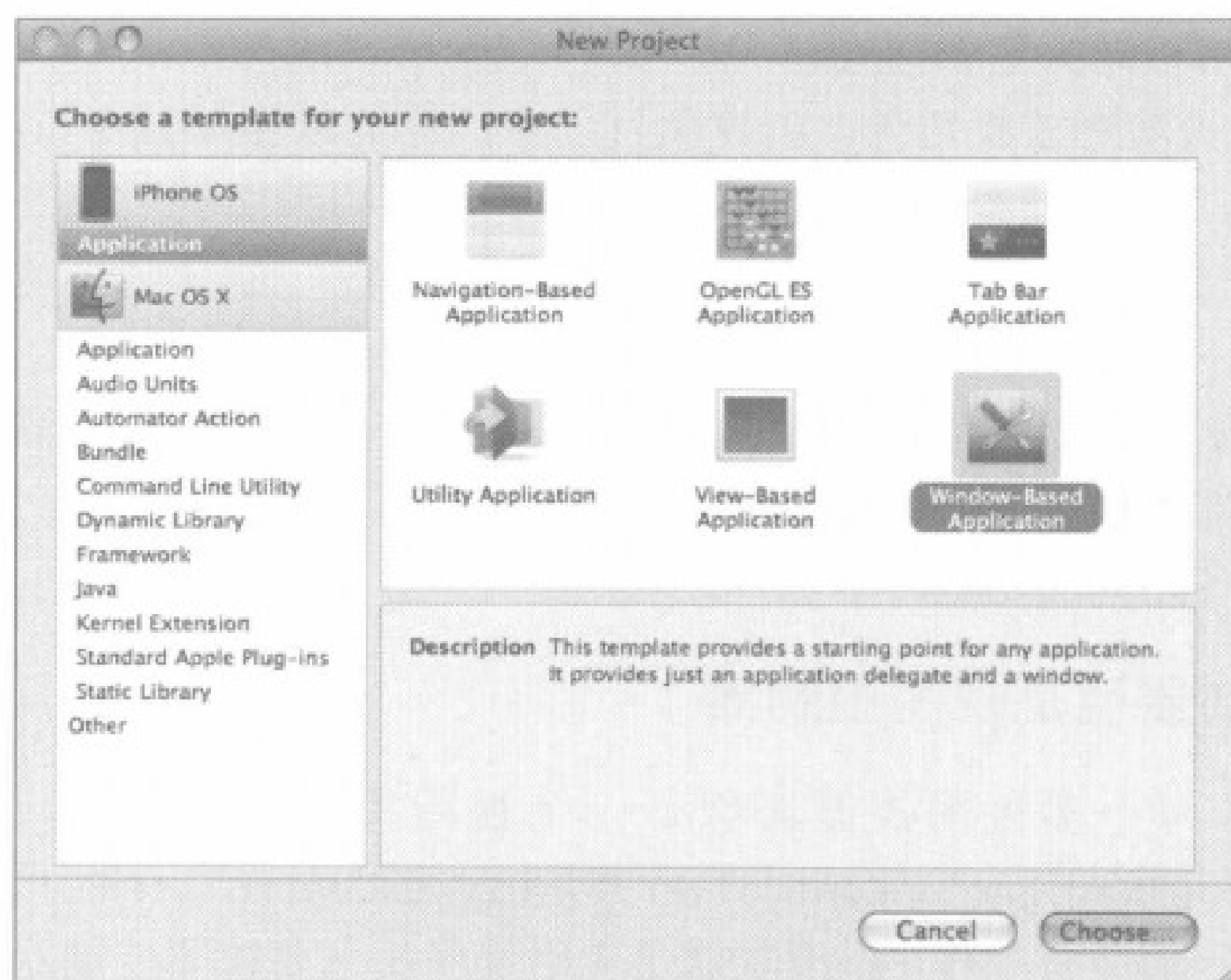


图6-8 选择新项目模板

刚才选择的模板实际上比我们以前使用的模板更加简单。这个模板仅提供一个窗口和一个应用程序委托，没有视图，没有控制器。此模板很少用于创建应用程序，但是，通过从头开始创建应用程序，你将能够很好地理解多视图应用程序的构造方式。

展开Groups & Files窗格中的Resources和Classes文件夹，看一下其中包含哪些内容。可以看到nib文件MainWindow.xib、Info.plist文件，以及类文件夹中用于实现应用程序委托的两个文件。应用程序所需的所有其他内容都需要自己创建。

6.3.1 创建视图控制器和 nib 文件

对于从头创建多视图应用程序，一个麻烦之处就是必须创建若干个互相连接的对象。我们将创建组成应用程序的所有文件，然后才在Interface Builder中进行操作以及编写代码。首先创建所有文件，我们可以使用Xcode的代码感知功能更快地编写代码。如果某个类未声明，代码感知功能将无法知道该类的信息，所以我们每次都必须键入完整的类，这会花费更长时间，并且容易出错。

幸运的是，除了项目模板，Xcode还为许多标准文件类型提供了文件模板，这使得创建应用程序的基本框架非常简单。单击Groups & Files窗格中的Classes文件夹，然后按下⌘N或从File菜单中选择New File...。看一下打开的窗口（参见图6-9）。

从左侧窗格选择Cocoa Touch Classes，可以看到大量用于常用的Cocoa Touch类的模板。选择UIViewController subclass，并单击Next。输入名称SwitchViewController.m，确保选中了Also create “SwitchViewController.h”，然后单击Finish。Xcode应该向Classes文件夹中添加两个文件，SwitchViewController类将充当根控制器。重复相同步骤两次，创建BlueViewController.m和YellowViewController.m，确保也为这两个控制器创建了相应的头文件。这两个内容视图将由SwitchViewController进行切换。

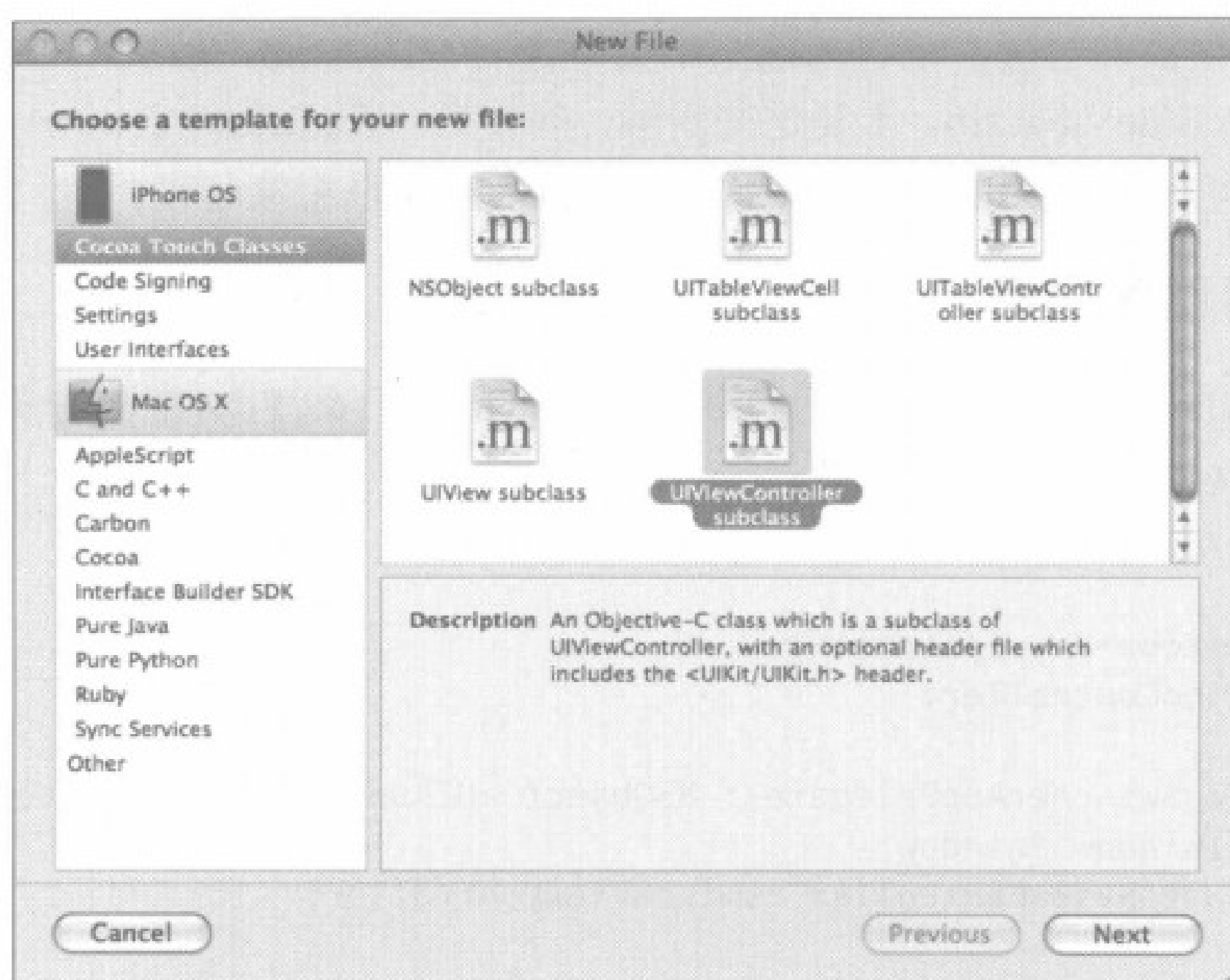


图6-9 创建新的视图控制器类

我们还需要另外两个nib文件，分别对应刚才创建的两个内容视图。要创建nib文件，就要单击Groups & Files窗格中的Resource文件夹，以便在正确的位置创建它们，然后再次按下⌘N或从File菜单中选择New File…。出现帮助窗口之后，在左侧窗格中的iPhone OS标题下选择User Interfaces（参见图6-10）。

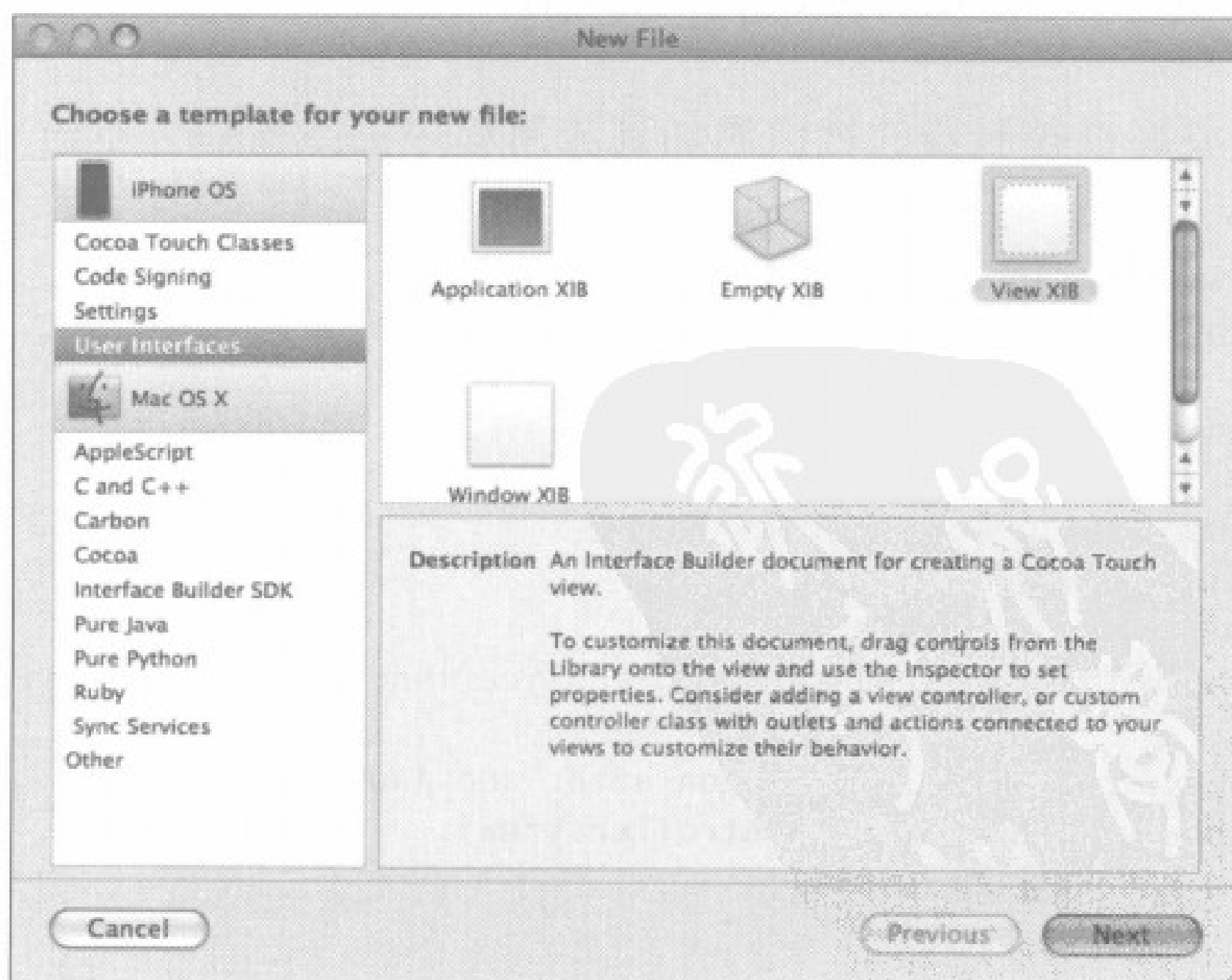


图6-10 为内容视图创建nib文件

选择View XIB模板的图标，这将创建一个带有内容视图的nib，然后单击Next按钮。当提示输入文件名时，输入BlueView.xib。重复这些步骤，创建另一个nib文件YellowView.xib。完成这些步骤之后，你将拥有所有必需的文件。接下来需要将所有文件衔接起来。

6.3.2 修改应用程序委托

多视图直通车的第一站是应用程序委托。单击Groups & Files窗格中的view_SwitchAppDelegate.h文件，并对文件进行以下更改：

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

@class View_SwitcherViewController;
@class SwitchViewController;

@interface View_SwitcherAppDelegate : NSObject <UIApplicationDelegate> {
    IBOutlet UIWindow *window;
    IBOutlet SwitchViewController *switchViewController;
}

@property (nonatomic, retain) UIWindow *window;
@property (nonatomic, retain) SwitchViewController *switchViewController;

@end
```

刚才输入的IBOutlet声明是一个输出口，它指向应用程序的根控制器。这个输出口必不可少，因为我们将编写代码来将根控制器的视图添加应用程序的主窗口。声明此输出口之后，当我们转到Interface Builder并在MainWindow.xib中添加一个SwitchViewController类实例之后，输出口将自动创建。

现在，我们需要将根控制器的视图添加到应用程序的主窗口。单击View_SwitcherAppDelegate.m并添加以下代码：

```
#import "View_SwitcherAppDelegate.h"
#import "SwitchViewController.h"

@implementation View_SwitcherAppDelegate

@synthesize window;
@synthesize switchViewController;

- (void)applicationDidFinishLaunching:(UIApplication *)application {

    // Override point for customization after app launch
    [window addSubview:switchViewController.view];
    [window makeKeyAndVisible];
}

- (void)dealloc {
    [window release];
    [switchViewController release];
}
```

```
    [super dealloc];
}
```

```
@end
```

除了实现switchViewController输出口之外，我们还将根控制器的视图添加到了窗口中。记住，窗口是通向用户的唯一途径，因此，需要向用户显示的任何内容都必须添加为窗口的子视图。

6.3.3 SwitchViewController.h

由于我们将在MainWindow.xib中添加一个SwitchViewController实例，因此现在是时候将所需的任何输出口或动作添加到SwitchViewController.h头文件中了。

我们需要通过一个动作方法在两个视图之间进行切换。我们不需要任何输出口，但是需要另外两个指针，分别指向将要交换的两个视图控制器。这些指针不需要输出口，因为我们将代码中而不是在nib中创建它们。将以下代码添加到SwitchViewController.h：

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

@class BlueViewController;
@class YellowViewController;

@interface SwitchViewController : UIViewController {
    YellowViewController *yellowViewController;
    BlueViewController *blueViewController;
}
@property (retain, nonatomic) YellowViewController *yellowViewController;
@property (retain, nonatomic) BlueViewController *blueViewController;

-(IBAction)switchViews:(id)sender;
@end
```

现在，我们已经声明了所需的动作，接下来可以将此类的一个实例添加到MainWindow.xib。

6.3.4 修改 MainWindow.xib

保存源代码，双击MainWindow.xib，在Interface Builder中打开它。nib的主窗口中应该显示4个按钮：File's Owner、First Responder、View_SwitcherAppDelegate以及Window（参见图6-11）。我们需要添加另外一个图标，该图标表示根控制器的一个实例。因为Interface Builder的库不包含SwitchViewController，所以必须添加一个视图控制器，并将其类改为SwitchViewController。

由于需要添加的类是UIViewController的一个子类，因此可以在View Controller中查看这些子类（参见图6-12），将其中一个子类拖到nib的主窗口中。

完成此操作之后，nib的主窗口将包含5个图标，以及一个包含灰色虚线圆角矩形的View窗口（参见图6-13）。

我们仅添加了一个UIViewController实例，但实际上我们需要一个SwitchViewController实例，因此，将视图控制器改为SwitchViewController。在nib的主窗口中单击View Controller图标，

并按下 $\text{⌘}4$ 打开身份检查器（参见图6-14）。

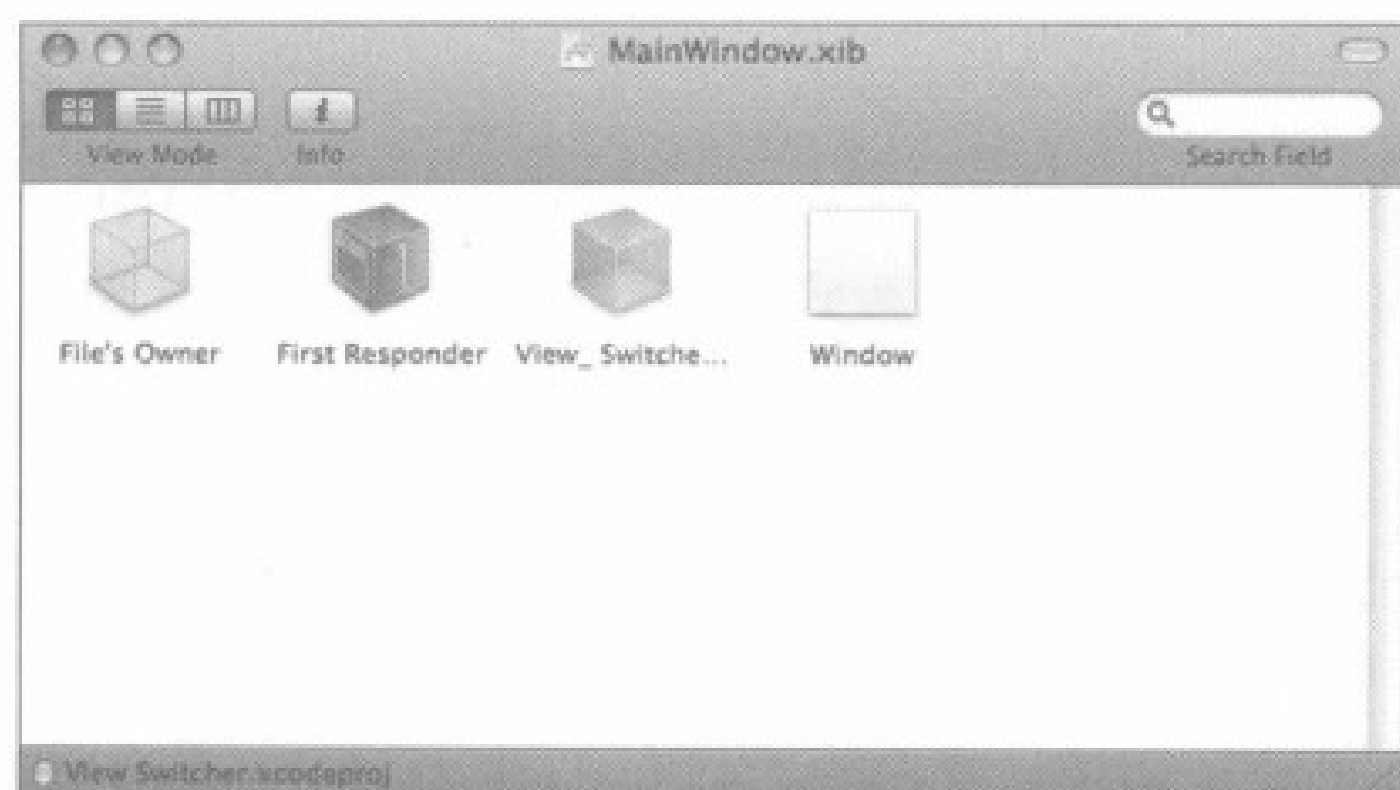


图6-11 MainMenu.xib

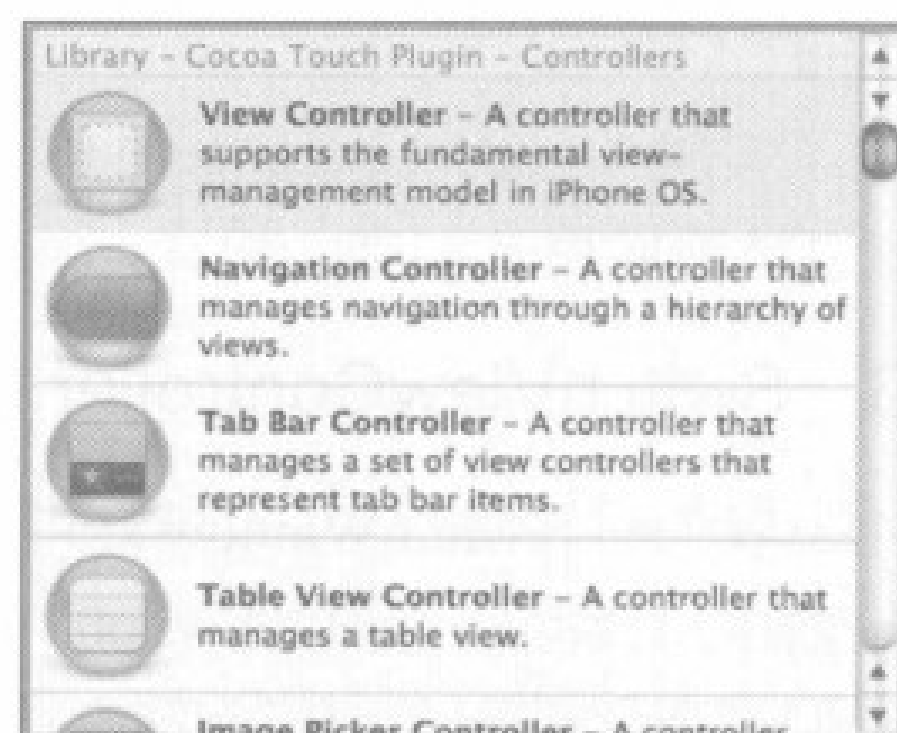


图6-12 库中的视图控制器

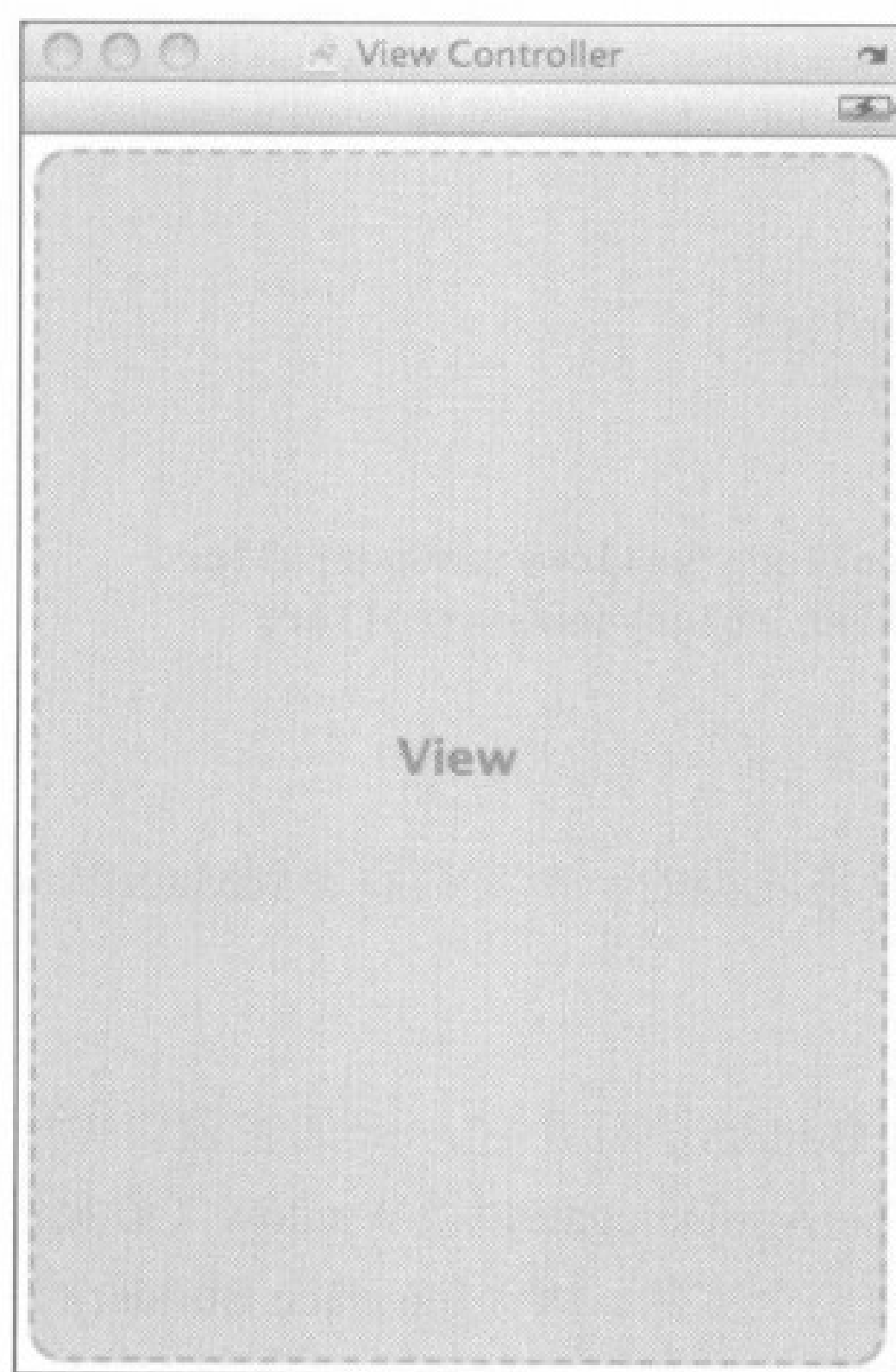


图6-13 Interface Builder中表示视图控制器的窗口

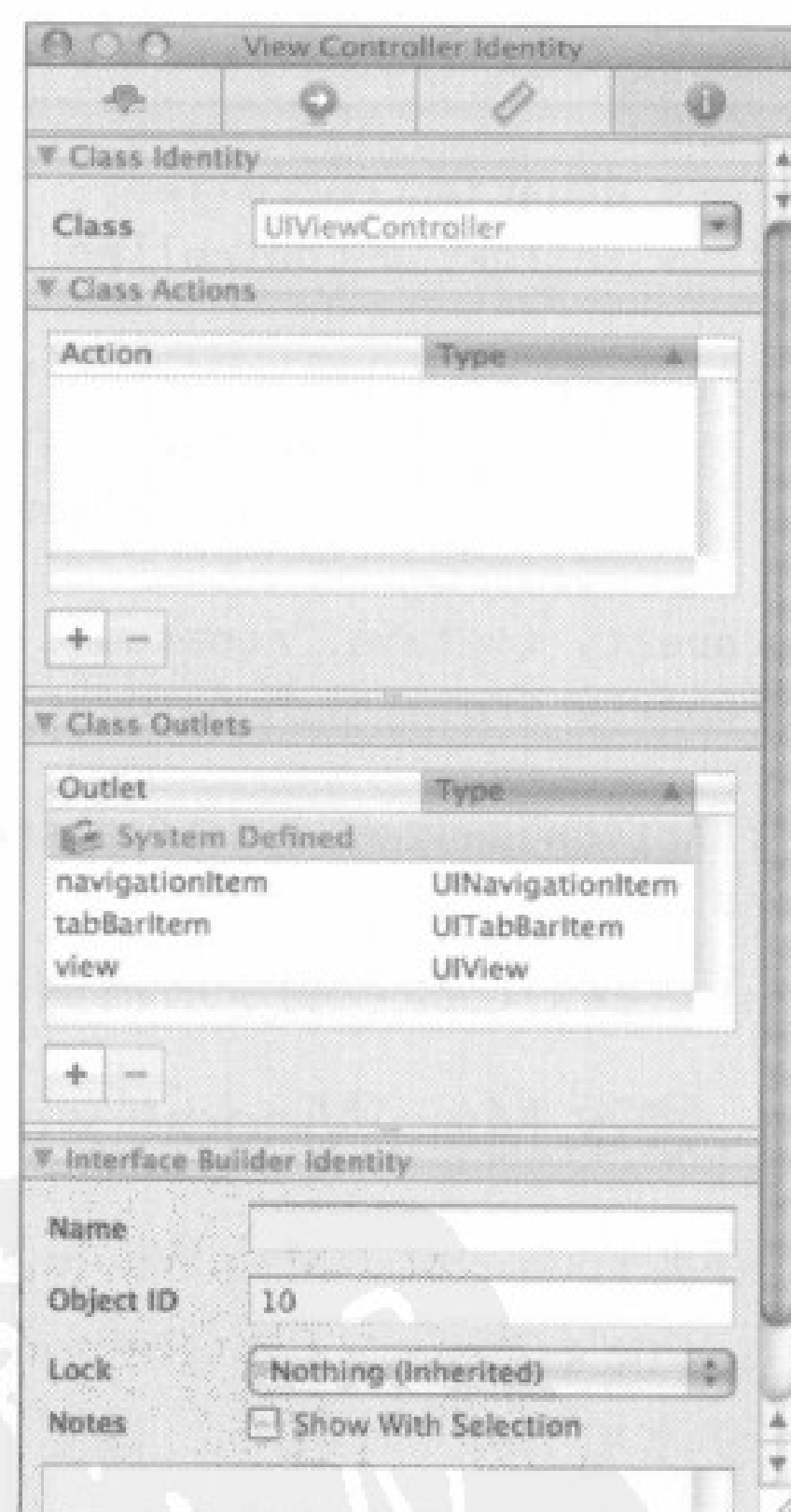


图6-14 身份检查器

借助身份检查器，你可以指定当前选定对象的类。当前指定的视图控制器是 `UIViewController`，且未定义任何动作。单击标签为 `Class` 的组合框，也就是检查器顶部现在显示为 `UIViewController` 的组合框。将 `Class` 改为 `SwitchViewController`。完成更改之后，`Class Actions` 部分将会显示 `switchViews:` 动作方法（参见图6-15）。还可以看到，在 nib 的主窗口中，新图标的名称由 `View Controller` 变为了 `Switch View Controller`。

现在，我们需要构建根控制器的视图。还记得将通用视图控制器拖到nib主窗口上时出现的新窗口吗（参见图6-13）？我们将在该窗口中构建SwitchViewController的视图。

回忆一下，SwitchViewController的任务是在蓝色视图和黄色视图之间进行切换。为此，它需要为用户提供一种方式来更改视图，所以我们将使用带有一个按钮的工具栏。现在让我们来构建这个工具栏。

从库中拖一个View到图6-13中显示的窗口上。记住：这是带有灰色背景，名为View的窗口。应将灰色背景替换为这个新视图。

现在，从库中拖动一个工具栏到视图上，将其放在底部，如图6-16所示。

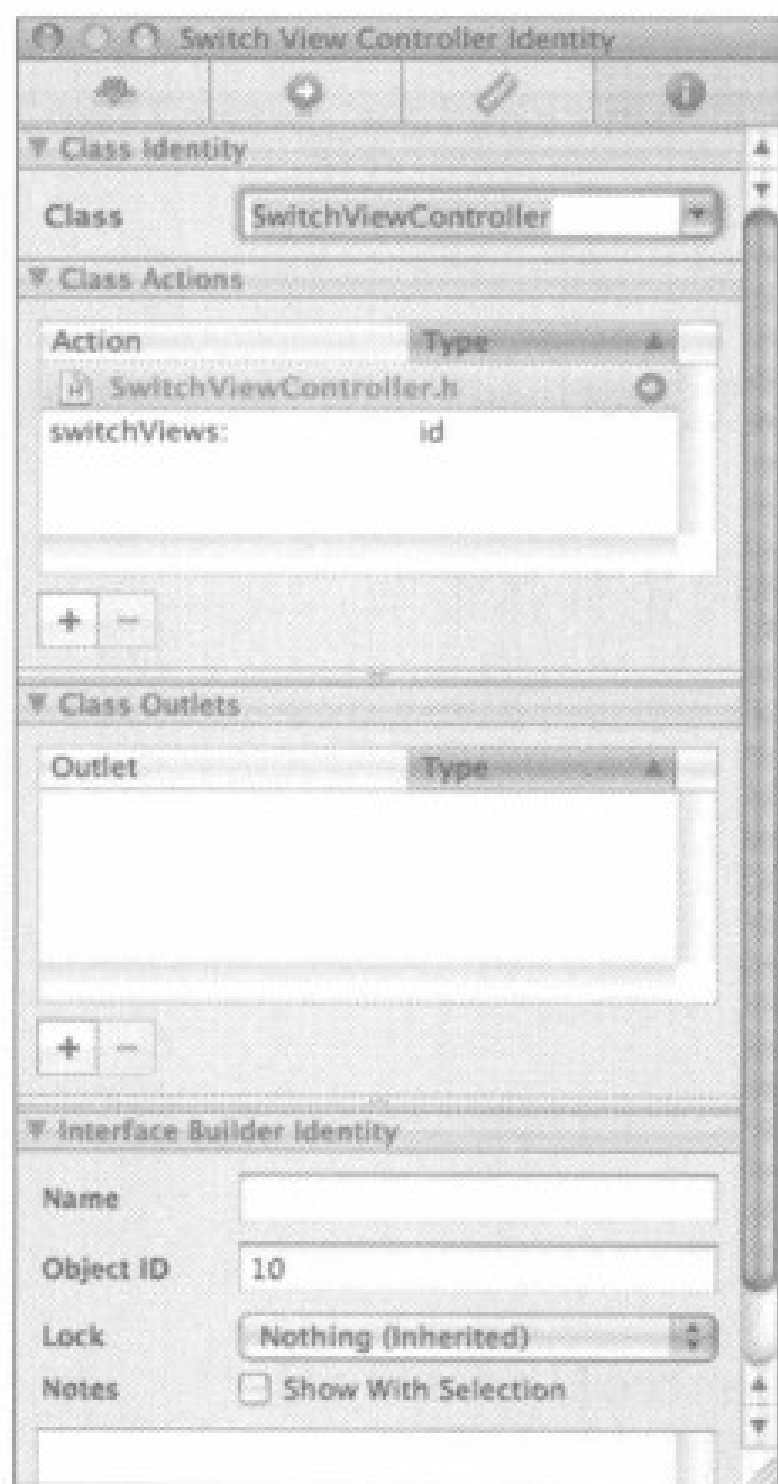


图6-15 将Class改为SwitchViewController之后的身份检查器

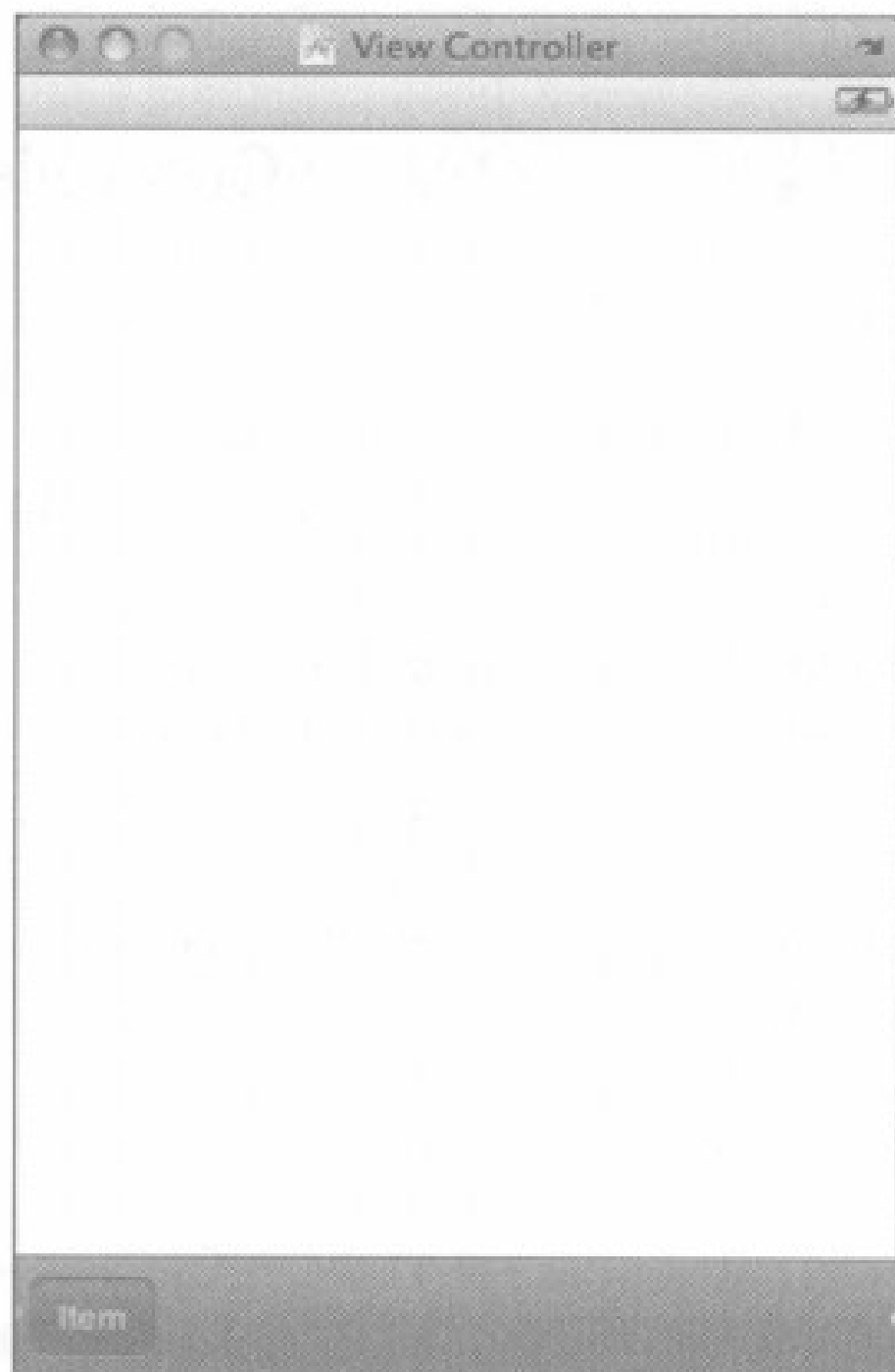


图6-16 在视图控制器的视图中添加一个工具栏

该工具栏带有一个按钮。我们使用该按钮来让用户在不同的内容视图之间切换。双击该按钮，将其标题改为Switch Views。按下回车键提交更改。

现在，我们可以将工具栏按钮链接到动作方法。在此之前，应该注意：工具栏按钮与其他iPhone控件不同。它们仅支持一个目标动作，并且仅符合条件时才触发该动作，相当于其他iPhone控件上的Touch Up Inside事件。

无需使用连接检查器将此按钮连接到我们的动作，只需单击Switch View按钮，等待片刻（避免双击），然后再次单击按钮将其选中。通过查看检查器的标题栏并确保它显示为Bar Button Item，可以验证是否选定了按钮。

选定Switch View按钮之后，按住Control键并将该按钮拖到Switch View Controller图标，然后

选择switchViews:动作。如果未弹出switchView:动作,而是看到一个名为delegate的输出口,这很可能是因为按住Control键拖动的是工具栏而不是按钮。要解决此问题,只需确保选定的是按钮而不是工具栏,然后重新按住Control键并进行拖动就可以了。

在前面,我们在View_SwitcherAppDelegate.h中创建了一个输出口,所以应用程序能够获得SwitchViewController实例并将其视图添加到应用程序主窗口。现在,我们需要在nib中将SwitchViewController实例连接到该输出口。按住Control键并将View_Switcher App Delegate图标拖到Switch View Controller图标,然后选择switchViewController输出口。可能还会看到第二个输出口,它具有一个类似的名称:viewController。如果确实看到该输出口,请确保连接到了switchViewController,而不是viewController。

至此任务就完成了,保存nib文件并返回到Xcode,以实现SwitchViewController。

6.3.5 编写 SwitchViewController.m

现在可以编写根视图控制器。当用户单击Switch Views按钮时,根视图控制器负责在黄色视图与蓝色视图之间进行切换。

将以下代码添加到SwitchViewController.m,然后我们将讨论具体细节:

```
#import "SwitchViewController.h"
#import "BlueViewController.h"
#import "YellowViewController.h"

@implementation SwitchViewController
@synthesize blueViewController;
@synthesize yellowViewController;

- (void)viewDidLoad
{
    BlueViewController *blueController = [[BlueViewController alloc]
                                           initWithNibName:@"BlueView" bundle:nil];
    self.blueViewController = blueController;
    [self.view addSubview:blueController.view atIndex:0];
    [blueController release];
}

- (IBAction)switchViews:(id)sender
{
    // Lazy load - we load the Yellow nib the first
    // time the button is pressed
    if (self.yellowViewController == nil)
    {
        YellowViewController *yellowController =
            [[YellowViewController alloc]
             initWithNibName:@"YellowView"
             bundle:nil];
        self.yellowViewController = yellowController;
    }
}
```

```

        [yellowController release];
    }

    if (self.blueViewController.view.superview == nil)
    {
        [yellowViewController.view removeFromSuperview];
        [self.view addSubview:blueViewController.view atIndex:0];\
    }
    else
    {
        [blueViewController.view removeFromSuperview];
        [self.view addSubview:yellowViewController.view atIndex:0];
    }
}

- (id)initWithNibName:(NSString *)nibNameOrNil
    bundle:(NSBundle *)nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil
        bundle:nibBundleOrNil]) {
        // Initialization code
    }
    return self;
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [yellowViewController release];
    [blueViewController release];
    [super dealloc];
}

@end

```

我们添加的第一个方法viewDidLoad是一个UIViewController方法，该方法将在载入nib时被调用。按住Option并双击方法名称，查看以下出现的文档。该方法在我们的超类中定义，当视图加载完成时，它将被覆盖为需要被通知的类。

我们将覆盖viewDidLoad以创建一个BlueViewController实例。我们使用initWithNibName方法从nib文件BlueView.xib加载BlueViewController实例。注意，为initWithNibName提供的文件名

未包含.xib扩展名。创建BlueViewController之后,我们将这个新实例分配给blueViewController属性。

```
BlueViewController *blueController = [[BlueViewController alloc]
    initWithNibName:@"BlueView" bundle:nil];
self.blueViewController = blueController;
```

接下来,我们插入蓝色视图作为根视图的一个子视图。将其插入在索引0的位置,这将告诉iPhone将此视图放在其他所有视图之后。将该视图放在其他所有视图之后可以确保刚才在Interface Builder中创建的工具栏在屏幕上始终可见,因为我们将内容视图放在了它的后面。

```
[self.view insertSubview:blueController.view atIndex:0];
```

那么,现在为何不在此处加载黄色视图呢?我们需要在某个时刻加载它,那么为什么不是现在呢?这个问题非常好。答案是,用户可能从不会点击Switch Views按钮。用户可能进入窗口,使用应用程序启动时出现的视图,然后退出。在这种情形下,为什么还要浪费资源来加载黄色视图及其控制器呢?

相反,我们将在第一次实际需要黄色视图的时候加载它。这种行为称为延迟加载(lazy loading),是降低内存开销的标准方式。黄色视图的实际加载在switchViews:方法中进行,那么让我们看一下该方法。

switchViews:首先检查属性yellowViewController是否为nil。如果是,则需要创建一个YellowViewController实例,就像在viewDidLoad方法中创建BlueViewController实例一样:

```
if (self.yellowViewController == nil)
{
    YellowViewController *yellowController =
        [[YellowViewController alloc]
            initWithNibName:@"YellowView"
            bundle:nil];
    self.yellowViewController = yellowController;
    [yellowController release];
}
```

除了在未按下Switch Views按钮时不使用黄色视图及其控制器的资源以外,当黄色视图当前未显示而且内存不足时,延迟加载还能够释放黄色视图以及相应的内存。因为我们知道,如果yellowViewController为nil,那么在下次调用此方法时,将会重新加载此视图。如果需要更多内存,可以安全地释放该视图以及相应的内存。延迟加载是iPhone上的一个关键的资源管理组件。

接下来,看一下blueViewController的视图的超视图,以确定要显示哪个视图以及退出哪个视图。当前未显示视图的超视图将为nil,这允许我们决定要显示和退出的视图。

```
if (self.blueViewController.view.superview == nil)
{
    [yellowViewController.view removeFromSuperview];
    [self.view insertSubview:blueViewController.view atIndex:0];
}
else
{
    [self.view insertSubview:blueViewController.view atIndex:0];
}
```

```

    [blueViewController.view removeFromSuperview];
    [self.view addSubview:yellowViewController.view atIndex:0];
}

```

当我们知道显示和退出的视图之后，可以删除当前显示的视图并插入另一个视图。现在，只需创建两个简单的内容视图，用于提供交换的内容。

6.3.6 实现内容视图

我们在此应用程序中创建的两个内容视图极其简单。每个视图都包含一个动作方法，该方法由一个按钮触发，而且两个视图都不需要输出口。这两个视图几乎完全相同。实际上，它们可以用一个类来表示。我们选择用两个独立的类来表示它们，因为大多数多视图应用程序就是这样构造的。在每个头文件中声明一个动作方法。首先，在BlueViewController.h中，添加以下声明：

```

#import <UIKit/UIKit.h>
@interface BlueViewController : UIViewController {

}
-(IBAction)blueButtonPressed:(id)sender;
@end

```

保存代码，在YellowViewController.h中添加以下代码：

```

#import <UIKit/UIKit.h>

@interface YellowViewController : UIViewController {

}
- (IBAction)yellowButtonPressed:(id)sender;
@end

```

保存代码，然后双击BlueView.xib，在Interface Builder中打开它，以进行一些更改。首先，我们需要告诉BlueView.xib，将从磁盘加载此nib的类是BlueViewController，因此单击File's Owner图标，并按下⌘4打开身份检查器。File's Owner默认为NSObject，请将其更改为BlueViewController。

接下来，在nib中更改视图的大小。双击View图标打开窗口，按下⌘3打开大小检查器。将此视图的高度由480改为416。为什么改为416呢？因为要从480像素中减去状态栏的20像素和工具栏的44像素，余下的416像素就是留给内容视图的空间。

按下⌘1打开属性检查器。单击标为Background的颜色，将此视图的背景颜色改为蓝色。接下来，从库中拖出一个Round Rect Button到窗口上。双击该按钮，将其标题改为Press Me。可以将该按钮放在任何合适的地方。然后，切换到连接检查器（按下⌘2），将Touch Up Inside事件拖到File's Owner图标，并连接到blueButtonPressed:动作方法。

我们还需要在此nib中做一件事情，那就是将BlueViewController's view输出口连接到nib中的视图。view输出口继承自父类UIViewController，为控制器提供了访问它所控制的视图的能力。更改文件所有者的底层类时，会破坏现有的输出口连接。因此，我们需要重新建立从控制器到其

视图的连接。要重新建立连接，就要按住Control键并将File's Owner图标拖到View图标，然后选择view输出口。

保存nib并返回到Xcode，双击YellowView.xib。我们对此nib文件进行几乎一样的更改。我们需要使用身份检查器将文件所有者从NSObject改为YellowViewController，使用大小检查器将视图的高度改为416像素，使用属性检查器将视图的背景改为黄色。还需要将一个圆角矩形添加到此视图中，将其标签更改为Press Me, Too，然后将该按钮的Touch Up Inside事件连接到File's Owner中的YellowButtonPressed:动作方法。最后，按住Control键并将File's Owner图标拖动到View图标，并连接到view输出口。

完成之后，保存nib并返回到Xcode。

我们将要实现的两个动作方法仅用于显示一个警告。这非常简单，将以下代码添加到BlueViewController.m即可：

```
#import "BlueViewController.h"

@implementation BlueViewController

- (IBAction)blueButtonPressed:(id)sender
{
    UIAlertView *alert = [[UIAlertView alloc]
        initWithTitle:@"Blue View Button Pressed"
        message:@"You pressed the button on the blue view"
        delegate:nil
        cancelButtonTitle:@"Yep, I did."
        otherButtonTitles:nil];
    [alert show];
    [alert release];
}

- (id)initWithNibName:(NSString *)nibNameOrNil
    bundle:(NSBundle *)nibBundleOrNil {
    ...
}
```

保存代码，切换到YellowViewController.m，将以下代码添加到该文件：

```
#import "YellowViewController.h"

@implementation YellowViewController

- (IBAction)yellowButtonPressed:(id)sender
{
    UIAlertView *alert = [[UIAlertView alloc]
        initWithTitle:@"Yellow View Button Pressed"
        message:@"You pressed the button on the yellow view"
        delegate:nil
        cancelButtonTitle:@"Yep, I did."
        otherButtonTitles:nil];
    [alert show];
    [alert release];
}
```

```
- (id)initWithNibName:(NSString *)nibNameOrNil
    bundle:(NSBundle *)nibBundleOrNil {
...

```

保存代码，现在可以运行我们的应用程序了。当应用程序启动时，它将显示在BlueView.xib中构建的视图，当点击Switch Views按钮时，它将显示在YellowView.xib中构建的视图。再次点击Switch Views按钮，将会重新显示在BlueView.xib中构建的视图。无论单击蓝色还是黄色视图上的按钮，都会获得一个警告视图，指示按下的是哪个按钮。此警告表明为显示的视图调用了正确的控制器类。

两个视图之间的转换过程比较生硬。那么，有没有使转换过程更加优美的方法呢？

6.4 制作转换动画

当然，有一种方法可以让转换过程更加优美。这就是制作转换动画，为用户提供可视化的更改反馈。可以调用UIView中的几个类方法来指示应该制作转换动画，指示应该使用的转换类型以及转换应该持续的时间。

回到SwitchViewController.m，将switchViews:方法替换为以下新版本：

```
- (IBAction)switchViews:(id)sender
{
    if (self.yellowViewController == nil)
    {
        YellowViewController *yellowController =
            [[YellowViewController alloc]
             initWithNibName:@"YellowView"
             bundle:nil];
        self.yellowViewController = yellowController;
        [yellowController release];
    }

    [UIView beginAnimations:@"View Flip" context:nil];
    [UIView setAnimationDuration:1.25];
    [UIView setAnimationCurve:UIViewAnimationCurveEaseInOut];
    if (blueViewController.view.superview == nil)
    {
        [UIView setAnimationTransition:
            UIViewAnimationTransitionFlipFromRight
            forView:self.view cache:YES];
        [blueViewController viewWillAppear:YES];
        [yellowViewController viewWillDisappear:YES];
        [yellowViewController.view removeFromSuperview];
        [self.view insertSubview:self.blueViewController.view atIndex:0];
        [yellowViewController viewDidDisappear:YES];
        [blueViewController viewDidAppear:YES];
    }
    else

```

```

{
    [UIView setAnimationTransition:
        UIViewAnimationTransitionFlipFromLeft
        forView:self.view cache:YES];
    [yellowViewController viewWillAppear:YES];
    [blueViewController viewWillDisappear:YES];
    [blueViewController.view removeFromSuperview];
    [self.view insertSubview:self.yellowViewController.view atIndex:0];
    [blueViewController viewDidDisappear:YES];
    [yellowViewController viewDidAppear:YES];
}
[UIView commitAnimations];
}

```

编译这个新版本并运行应用程序。单击Switch Views按钮之后,新视图将会以翻页的形式显示,而不只是简单地出现,如图6-17所示。

在第5章中,为了制作按钮的移动动画,我们首先创建一个动画块,执行要制作成动画的更改,然后提交这些更改。我们在这里也使用这种方法。首先声明一个动画块,并指定动画的持续时间:

```

[UIView beginAnimations:@"View Flip" context:nil];
[UIView setAnimationDuration:1.25];

```

然后,设置动画曲线,这决定了动画的持续时间。默认情况下,动画曲线是一条线性曲线,动画匀速地发生。我们在此处设置的选项指示动画应该更改其速度,开始和结束时速度较慢,中间速度较快。这使动画看起来更加自然,不再那么呆板。

```

[UIView setAnimationCurve:UIViewAnimationCurveEaseInOut];

```

接下来,需要指定要使用的转换类型。在编写本书时,iPhone上提供了4种视图转换类型:

- ☐ UIViewAnimationTransitionFlipFromLeft
- ☐ UIViewAnimationTransitionFlipFromRight
- ☐ UIViewAnimationTransitionCurlUp
- ☐ UIViewAnimationTransitionCurlDown

我们可以选择两种不同的效果,具体选择哪一种,取决于要显示的视图。为一种转换使用向左翻转,为另一种转换使用向右翻转,这会使人感觉视图在前后翻页。缓存选项在动画开始时生成一个快照,并在动画的每个步骤中使用该图像,而不是重新绘制视图,这能够加快绘制的速度。应该始终缓存动画,除非视图外观在动画期间需要改变。

```

[UIView setAnimationTransition:UIViewAnimationTransitionFlipFromRight
    forView:self.view cache:YES];

```

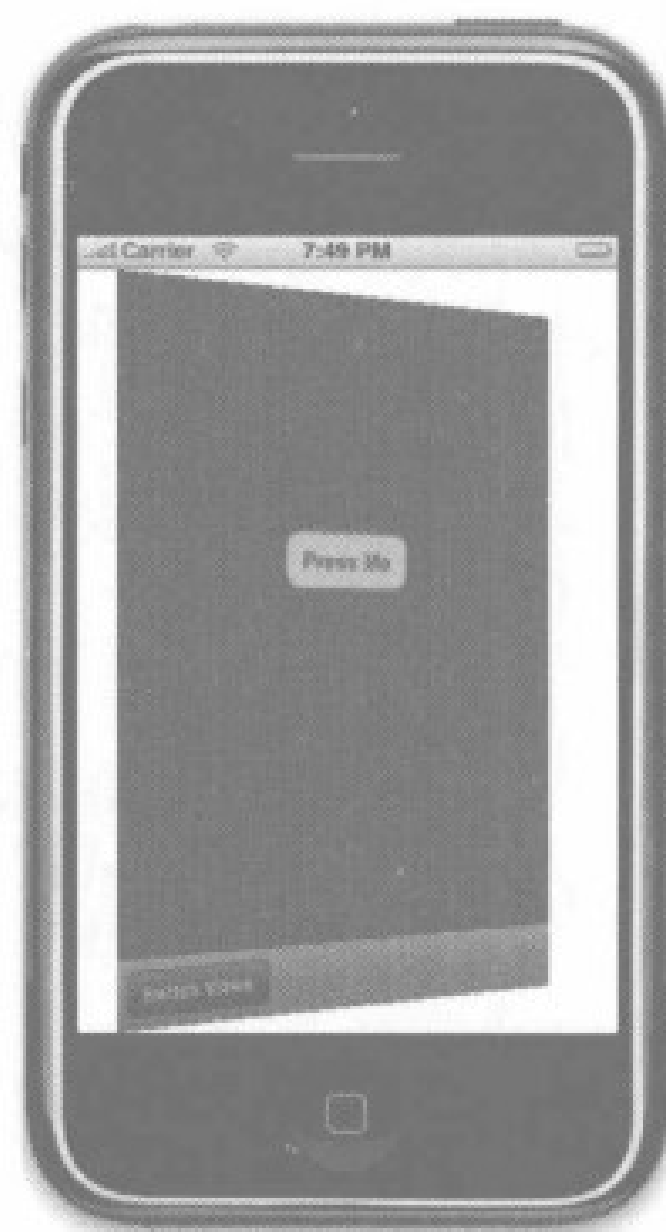


图6-17 使用翻页样式的视图转换动画

设置转换类型之后，进行两次调用，调用在转换过程中使用的两个视图如下：

```
[self.blueViewController viewWillAppear:YES];  
[self.yellowViewController viewWillDisappear:YES];
```

完成视图交换之后，在这两个视图上进行两次调用：

```
[self.yellowViewController viewDidDisappear:YES];  
[self.blueViewController viewDidAppear:YES];
```

UIView中这些方法的默认实现不会执行任何操作，所以我们对viewDidDisappear:和viewDidAppear:的调用也不会执行任何操作，因为我们的视图是UIView的实例。

既然这些调用不执行任何操作，为什么还要调用它们呢？虽然我们现在使用的是UIView，但我们仍然可以在开发过程中的某一时刻创建一个UIView子类，而且这个子类可能需要在转换前后执行某种操作。例如，当退出包含动画的视图时，这些视图往往会关闭这些动画，而当显示这些视图时，它们又会打开这些动画。执行这4次调用不会影响到应用程序的性能，因为它们不会触发任何代码。加入这些调用，我们就能够确保，即使在切换视图的类时，应用程序也能够继续工作。

当然，当指定了要制作成动画的所有更改之后，我们在UIView上调用commitAnimations:

```
[UIView commitAnimations];
```

由于CoCoa Touch在后台使用了Core Animation，因此我们能够使用极少的代码制作出非常复杂而精美的动画。

6.5 重构

现在，应用程序能够按照预期良好地运行。那么是不是该学习下一章了？不是，还有一个需要注意的地方。

看一下刚才编写的新版本的switchViews:。具体看一下此部分：

```
if (self.blueViewController.view.superview == nil)  
{  
    [UIView setAnimationTransition:  
        UIViewAnimationTransitionFlipFromRight  
        forView:self.view cache:YES];  
    [self.blueViewController viewWillAppear:YES];  
    [self.yellowViewController viewWillDisappear:YES];  
    [self.yellowViewController.view removeFromSuperview];  
    [self.view insertSubview: self.blueViewController.view atIndex:0];  
    [self.yellowViewController viewDidDisappear:YES];  
    [self.blueViewController viewDidAppear:YES];  
}  
else  
{  
    [UIView setAnimationTransition:  
        UIViewAnimationTransitionFlipFromLeft  
        forView:self.view cache:YES];
```

```
[self.yellowViewController viewWillAppear:YES];
[self.blueViewController viewWillDisappear:YES];
[self.blueViewController.view removeFromSuperview];
[self.view insertSubview:self.yellowViewController.view atIndex:0];
[self.blueViewController viewDidDisappear:YES];
[self.yellowViewController viewDidAppear:YES];
}
```

只要看到两个或更多代码块非常类似,就应该检查代码,考虑一下是否能够将这些代码合并为一个代码块。如果我们在一个代码块中发现错误,则必须在两个或更多位置修改这些错误,因为除了每个调用中使用的视图以外,这些代码块几乎完全相同。调整代码以改善其质量的流程,或者在不更改代码功能的前提下使其更具可维护性的流程称为**重构**。让我们重构这部分代码。

在大部分语言中,对于这种情形进行重构的方式应该是创建一个需要调用两次的新方法或函数。下面是在一个方法中处理这种情形的另一种方式。看一下这个新的switchViews版本:

```
- (IBAction)switchViews:(id)sender
{
    if (self.yellowViewController == nil) {
        YellowViewController *yellowController =
            [[YellowViewController alloc]
             initWithNibName:@"YellowView"
             bundle:nil];
        self.yellowViewController = yellowController;
        [yellowController release];
    }

    [UIView beginAnimations:@"View Flip" context:nil];
    [UIView setAnimationDuration:1.25];
    [UIView setAnimationCurve:UIViewAnimationCurveEaseInOut];

    UIViewController *coming = nil;
    UIViewController *going = nil;
    UIViewAnimationTransition transition;

    if (self.blueViewController.view.superview == nil) {
        coming = blueViewController;
        going = yellowViewController;
        transition = UIViewAnimationTransitionFlipFromLeft;
    }
    else{
        coming = yellowViewController;
        going = blueViewController;
        transition = UIViewAnimationTransitionFlipFromRight;
    }

    [UIView setAnimationTransition: transition forView:self.view
    cache:YES];
    [coming viewWillAppear:YES];
}
```



```
[going viewWillDisappear:YES];  
[going.view removeFromSuperview];  
[self.view addSubview: coming.view atIndex:0];  
[going viewDidDisappear:YES];  
[coming viewDidAppear:YES];  
  
[UIView commitAnimations];  
}
```

基本而言,我们仅声明了coming和going两个新指针,根据当前显示的视图,将blueViewController或yellowBlueController分配给合适的指针。现在,我们在一个代码块中包含了这两个指针,一种逻辑不会在两个地方重复出现,这使代码更容易维护。

如果发现多次输入了非常相似的代码,或者需要复制和粘贴较大的代码块,那就应该停下工作,然后想想还有没有更好的方式。

6

6.6 小结

终于大功告成!创建自己的多视图控制器是一项繁重的工作,对吗?我们从头构建了一个多视图应用程序,现在,你应该掌握了如何构建多视图应用程序。尽管Xcode包含3个最常用的多视图应用程序项目模板,但仍然需要理解这些应用程序的总体结构,这样才能按部就班地构建自己的应用程序。已提供的这些模板能够节省大量时间,但是有时候,它们并不符合要求。

在接下来的两章中,我们将继续构建多视图应用程序,以强化本章中介绍的概念,并介绍如何构建更加复杂的应用程序。第7章将构造一个标签栏应用程序,而第8章将构造一个基于导航栏的应用程序。



在上一章中，我们构建了第一个多视图应用程序。本章将构建一个完整的标签栏应用程序，它包含5个不同的标签和5个不同的内容视图。构建此应用程序能够巩固上一章介绍的知识，但是用一整章的篇幅来学习已经掌握的操作似乎有点浪费，所以本章将通过这5个内容视图展示如何使用至今未曾涉及的一种iPhone控件。该控件就是选取器视图（picker view），但是人们经常称之为选取器。

你可能还不熟悉这个名称，但它是iPhone中很常用的一个工具。选取器是带有能够旋转的刻度盘的控件。它们用于在Calendar应用程序中输入日期，或者在Clock应用程序中设置计时器（参见图7-1）。

选取器是本书目前为止介绍的最复杂的iPhone控件，因此，理所应当受到更多关注。选取器可以配置为显示一个或多个刻度盘。默认情况下，选取器显示文本列表，但是它们也能够显示图像。

7.1 Pickers 应用程序

本章的Pickers应用程序包含一个标签栏。在构建该应用程序时，我们将更改默认的标签栏，使其包含5个标签，向每个标签栏项添加一个图标，然后创建一系列内容视图，并将每个视图连接到每个标签。

我们将构建的第1个内容视图包含一个日期选取器（date picker），这是最容易实现的选取器类型（参见图7-2）。该视图还将有一个按钮，点击该按钮时，视图将弹出警告，显示选取的日期。

第2个标签中的选取器包含一组值（参见图7-3）。此选取器的实现比日期选取器稍微难一些。本章将介绍如何使用委托和数据源在选取器中指定要显示的值。

在第3个标签中，我们将创建带有两个独立滚轮的选取器。从技术上说，滚轮应该称为选取器组件，也就是说，我们将在其中创建带有两个组件的选取器。我们将了解如何使用数据源和委托向选取器提供两个独立的数据列表（参见图7-4）。此选取器的每个组件都可以在不影响对方的情况下进行更改。

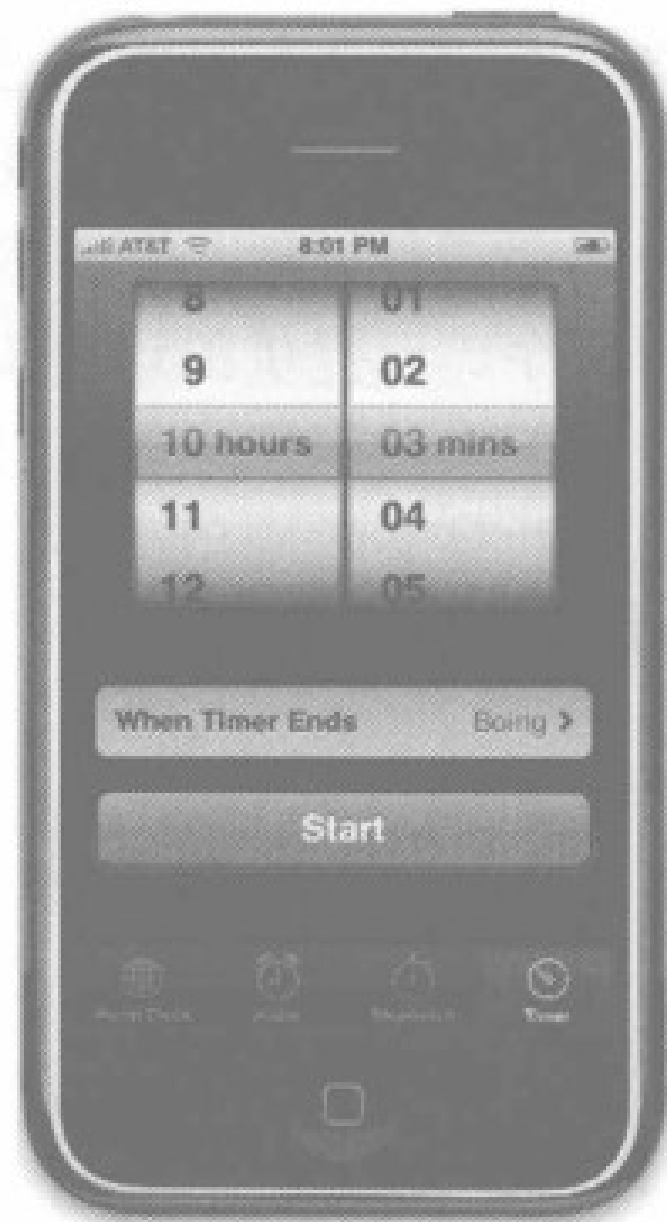


图7-1 Clock应用程序中的选取器

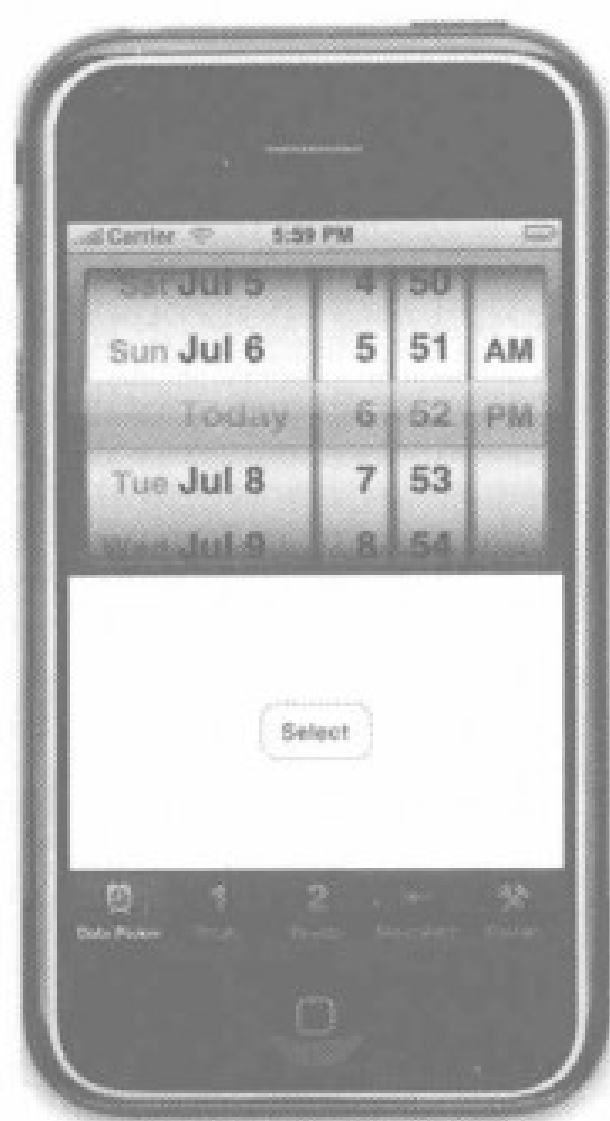


图7-2 第1个标签显示日期选取器

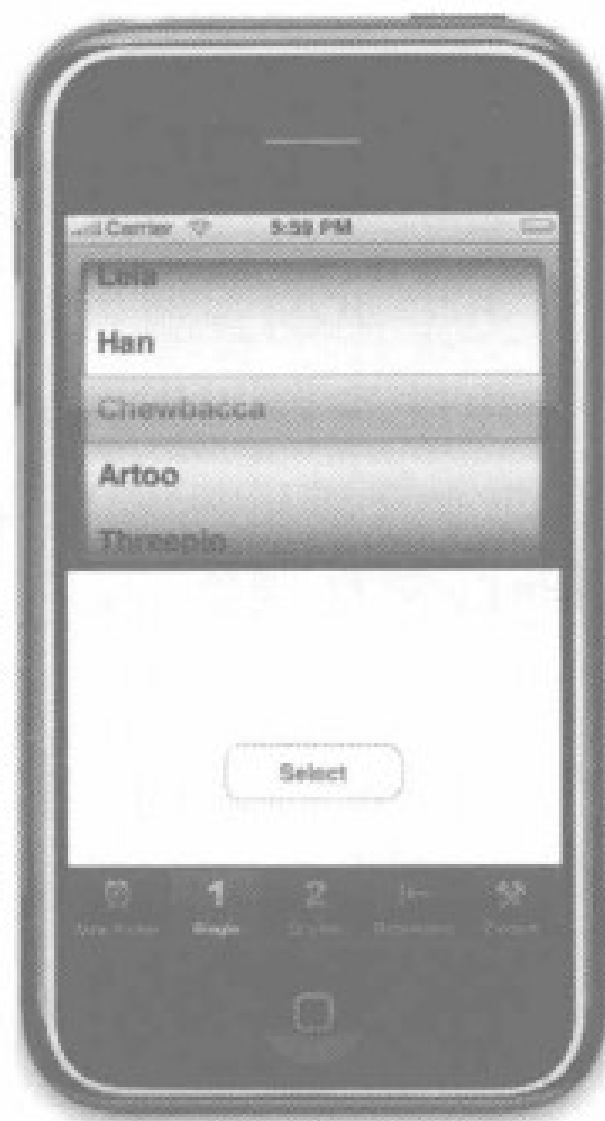


图7-3 此选取器显示一组值

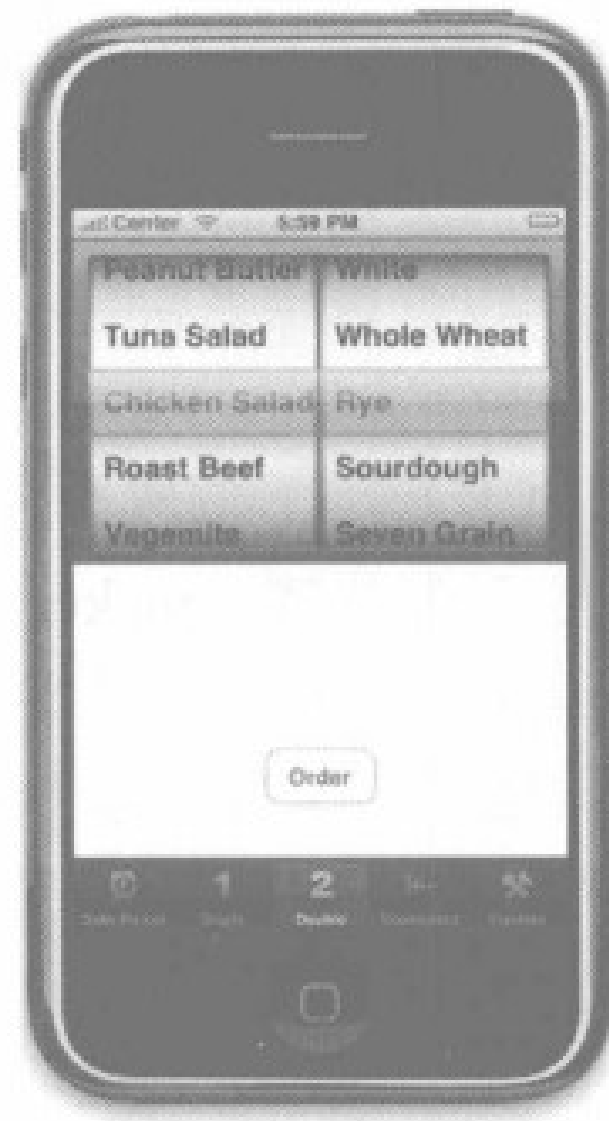


图7-4 此选取器包含两个组件

7

在第4个内容视图中，我们将创建另一个带有两个组件的选取器。但是这一次，右侧组件中显示的值将根据左侧组件中选定值的变化而变化。在我们的示例中，左侧组件中将显示一组州，右侧组件中将显示该州的ZIP编号（参见图7-5）。

最后，我们还将创建第5个内容视图。我们将了解如何将图像数据添加到选取器，并编写一个小游戏来进行演示，该游戏使用一个带有5个组件的选取器。在苹果公司的文档中的许多位置，都将选取器的外观描述为类似老虎机的样子。那么，还有比“小型老虎机”更适合的例子吗（参见图7-6）？对于此选取器，用户无法手动更改组件的值，但是能够选择Spin按钮来使5个滚轮旋转到一个新的、随机选定的值。如果在一行中出现了3个完全一样的图片，则表明用户获胜。

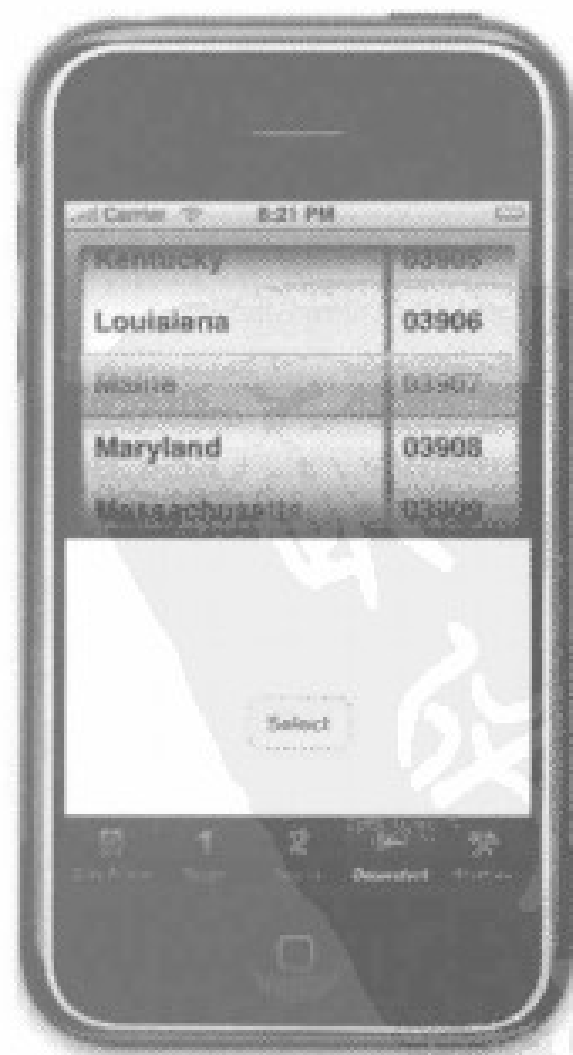


图7-5 在此选取器中，一个组件依赖于另一个组件。当在左侧组件中选定一个州时，右侧组件将显示该州的ZIP编号



图7-6 包含5个组件的选取器。注意，我们无意将你的iPhone变成一个玩具

7.2 委托和数据源

在开始构建应用程序之前，让我们看一下为什么选取器比之前所使用的控件都要复杂。更加复杂不仅仅意味着Interface Builder中有更多可配置属性需要设置。实际上，只有很少的选取器属性能够在Interface Builder中配置。要使用选取器，并不是从Interface Builder中挑选一个选取器并放在内容视图上，然后进行配置就行了，但是日期选取器是个例外。除此之外，还需要为选取器提供选取器代理（picker delegate）和选取器数据源（picker datasource）。

现在，你应该已经熟悉委托的用法了。我们已经使用过应用程序委托和操作表委托，这里的基本思想与它们一样。选取器将一些工作分配给它的委托。其中最重要的任务是，确定要为每个组件中的每一行绘制的实际内容。选取器要求委托在特定组件上的特定位置绘制一个字符串或一个视图。

除了委托之外，选取器还必须包含一个数据源。数据源的工作原理与委托类似，因为它的方法将在预先指定的时刻被调用。选取器使用数据源方法获取组件数和每个组件中的行数。如果未指定数据源和委托，选取器就无法工作，甚至无法绘制选取器。

在很多情况下，数据源和委托是同一个对象，该对象也是包含选取器的视图的视图控制器，我们将在本章的应用程序中采用这种方法。每个内容窗格的视图控制器都将是其选取器的数据源和委托。

说明 很多人存在这样的疑问：选取器数据源是应用程序的模型、视图，还是控制器部分呢？这是一个难以回答的问题。数据源似乎应该是模型的一部分，但是实际上，它是控制器的一部分。数据源并不总是一个用于保存数据的对象，它是为选取器提供数据的对象。

让我们打开Xcode，开始构建我们的应用程序。

7.3 建立工具栏框架

由于Xcode没有为标签栏应用程序提供模板，因此我们将从头构建自己的模板。这不需要太多工作，而且是一个很好的实践机会。创建一个新项目，再次选择Window-Based Application模板。输入Pickers作为模板名称。接下来将介绍构建应用程序的完整步骤，如果在构建过程中，遇到难以理解和掌握的步骤，一定要坚持下去。如果遇到难以解决的问题，随时可以返回再来。如果不愿意直接跳到后面的步骤也没有关系，我们将详细介绍每一步。

7.3.1 创建文件

在上一章中，我们创建了一个根视图控制器。这次还将创建一个根视图控制器，但是无需为其创建一个类，因为苹果公司提供了一个非常不错的类来管理标签栏视图，所以我们将使用UITabBarController实例来处理根控制器。稍后将在Interface Builder中创建该实例。

需要在Xcode中创建5个新类：根控制器进行切换的5个视图控制器。

展开Groups & Files窗格中的Classes和Resources文件夹, 接下来, 单击Classes文件夹, 按下⌘N 或从File菜单中选择New File....

在新建文件窗口的左侧窗格中选择Cocoa Touch Classes, 然后选择UIViewController subclass 图标, 并单击Next。将第一个类命名为DatePickerViewController.m, 确保选中了Also create “DatePickerViewController.h”。

重复此步骤, 将余下的类分别命名为SingleComponentPickerViewController.m、Double-ComponentPickerViewController.m、DependentComponentPickerViewController.m 和 Custom-PickerViewController.m。

接下来, 单击Resources文件夹, 再次按下⌘N, 或从File菜单中选择New File.... 这一次, 在新建文件帮助窗口的左侧窗格中选择User Interfaces, 然后单击View XIB图标。我们需要5个nib, 分别对应每个内容视图。将第一个nib命名为DatePickerView.xib。然后创建其他4个nib文件: SingleComponentPickerView.xib、DoubleComponentPickerView.xib、DependentComponentPickerView.xib和CustomPickerView.xib。

7

7.3.2 设置内容视图 nib

双击DatePickerView.xib, 在Interface Builder中打开该文件。我们实际上并不打算构建自己的内容视图。我们将搭建标签视图应用程序的框架, 让基本的应用程序能够正常运行。必须确保每个nib文件都指向正确的File's Owner图标, 文件所有者应该是对应的UIViewController子类。对于DatePickerView.xib, 控制器类必须是DatePickerViewController。单击File's Owner, 然后按下⌘4打开身份检查器。将类改为DatePickerViewController。

现在, 按住Control键并将File's Owner图标拖到View图标, 选择view输出口。最后, 单击View图标, 按下⌘3打开大小检查器。将视图的高度改为411像素, 这是减去状态栏和标签栏后剩下的空间。保存并关闭此nib。

对其余4个nib文件重复上述步骤, 将File's Owner指向对应的视图控制器。在完成之后一定要保存每个nib文件, 然后返回到Xcode。

7.3.3 添加根视图控制器

我们将在Interface Builder中创建自己的根视图控制器, 这将是一个UITabBarController实例。但是在创建根控制器之前, 我们应该为其声明一个输出口。单击PickersAppDelegate.h类, 添加以下代码:

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

@class PickersViewController;

@interface PickersAppDelegate : NSObject <UIApplicationDelegate> {
    IBOutlet UIWindow *window;
    IBOutlet UITabBarController *rootController;
}
```

```
@property (nonatomic, retain) UIWindow *window;
@property (nonatomic, retain) UITabBarController *rootController;
@end
```

在Interface Builder中创建根视图控制器之前，将以下代码添加到PickersAppDelegate.m:

```
#import "PickersAppDelegate.h"

@implementation PickersAppDelegate

@synthesize window;
@synthesize rootController;

- (void)applicationDidFinishLaunching:(UIApplication *)application {

    // Override point for customization after app launch
    [window addSubview:rootController.view];
    [window makeKeyAndVisible];
}

- (void)dealloc {
    [rootController release];
    [window release];
    [super dealloc];
}

@end
```

你应该不会对这些内容感到惊奇。我们在上一章也曾执行过同样的操作，但是这里没有使用苹果公司提供的控制器类，而是使用我们自己编写的控制器类。请保存这两个文件。

标签栏使用图标来表示每个标签，因此在转到Interface Builder之前，我们应该添加要使用的图标。可以在本书附带的项目归档文件中找到合适的图标，这些图标位于07 Picker/Tab Bar Icons/文件夹中。图标大小应该是24×24像素，并采用.png格式。图标文件应该有一个透明的背景。一般而言，浅灰色图标最适合标签栏。不要为设置合适的标签栏外观发愁。iPhone会自动设置合适的图标外观，就像设置应用程序的图标一样。

你现在应该熟悉如何将资源添加到项目了，接下来添加前面提供的5个图标，将它们从目前所在位置拖到Xcode项目的Resources文件夹，或者从Project菜单中选择Add to Project...。

添加图标之后，双击MainWindow.xib，在Interface Builder中打开该文件。从库中拖一个Tab Bar Controller到nib的主窗口（参见图7-7）。需要确保将其拖到MainWindow.xib的窗口，而不是Window窗口，因为后者不支持拖放操作。

将标签栏控制器放置到nib的主窗口上之后，将会出

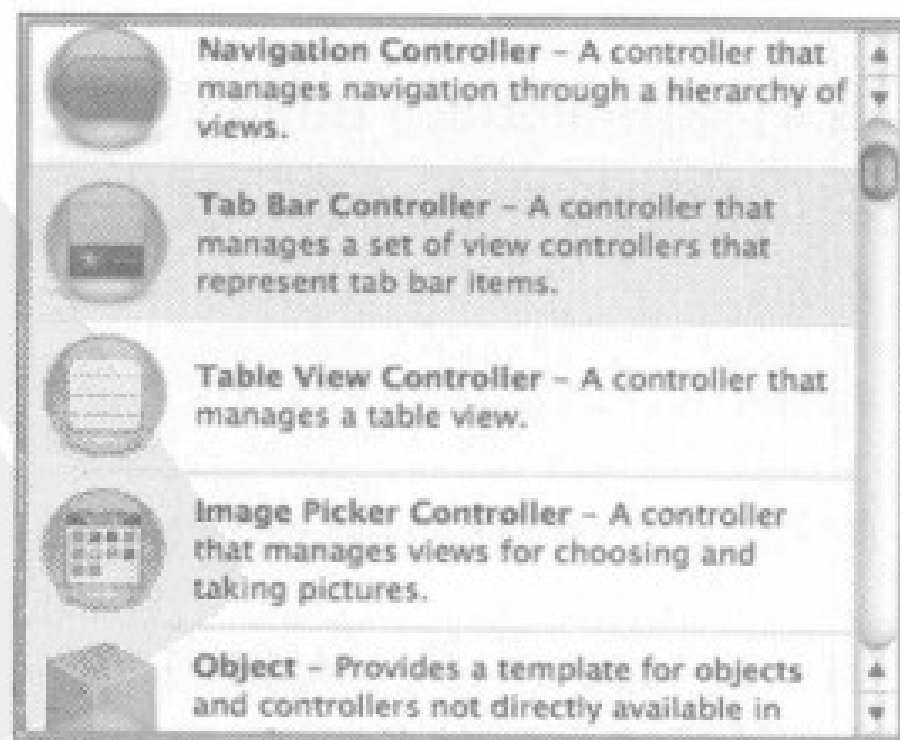


图7-7 库中的标签栏控制器

现一个新窗口，如图7-8所示。这个标签栏控制器将是我们的根视图控制器。回想一下，根视图控制器控制用户在应用程序运行之后看到的第一个视图。

在nib的主窗口中单击Tab Bar Controller图标，并按下⌘1打开它的属性检查器。标签栏控制器的属性检查器如图7-9所示。

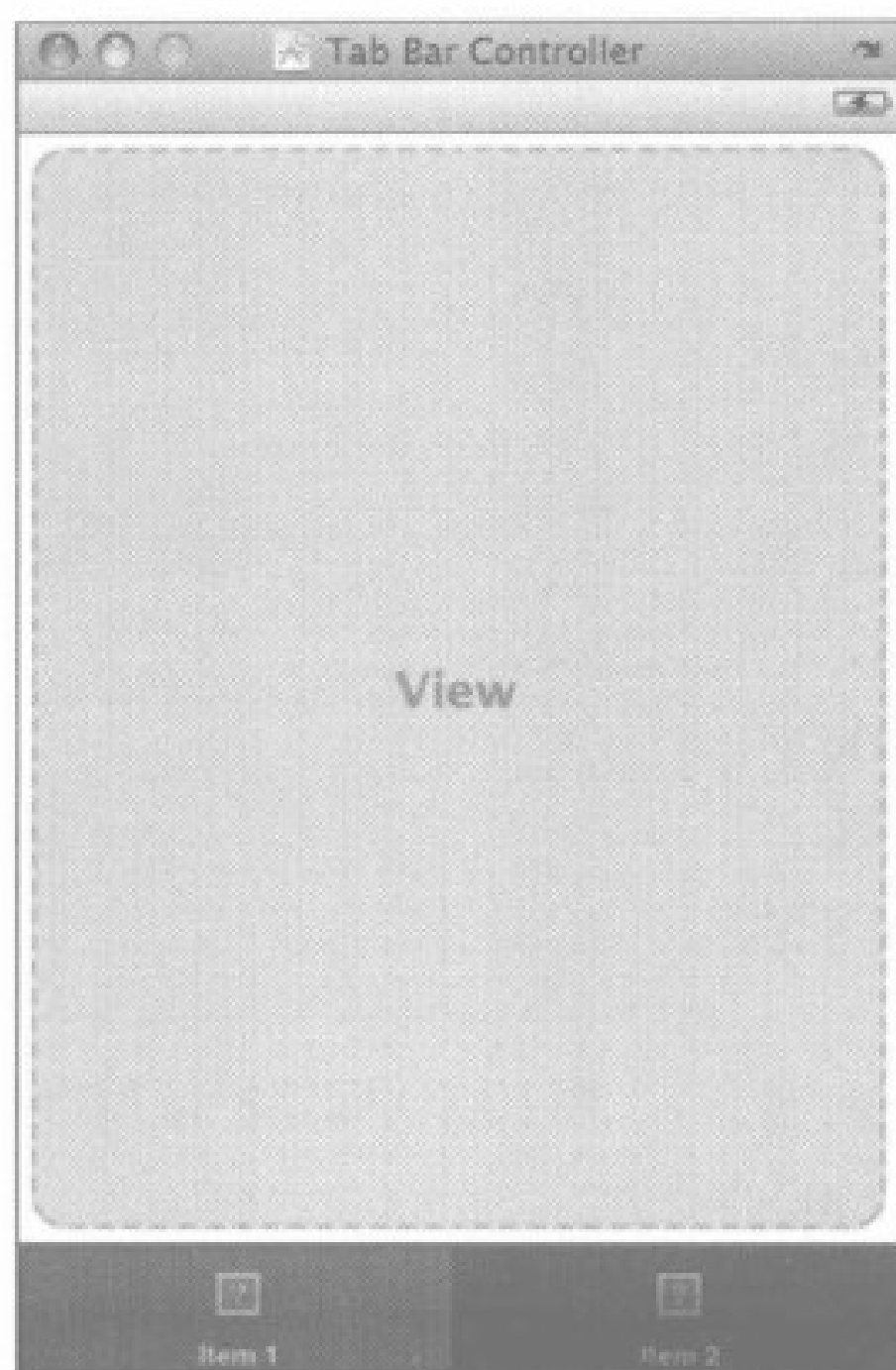


图7-8 标签栏控制器的窗口

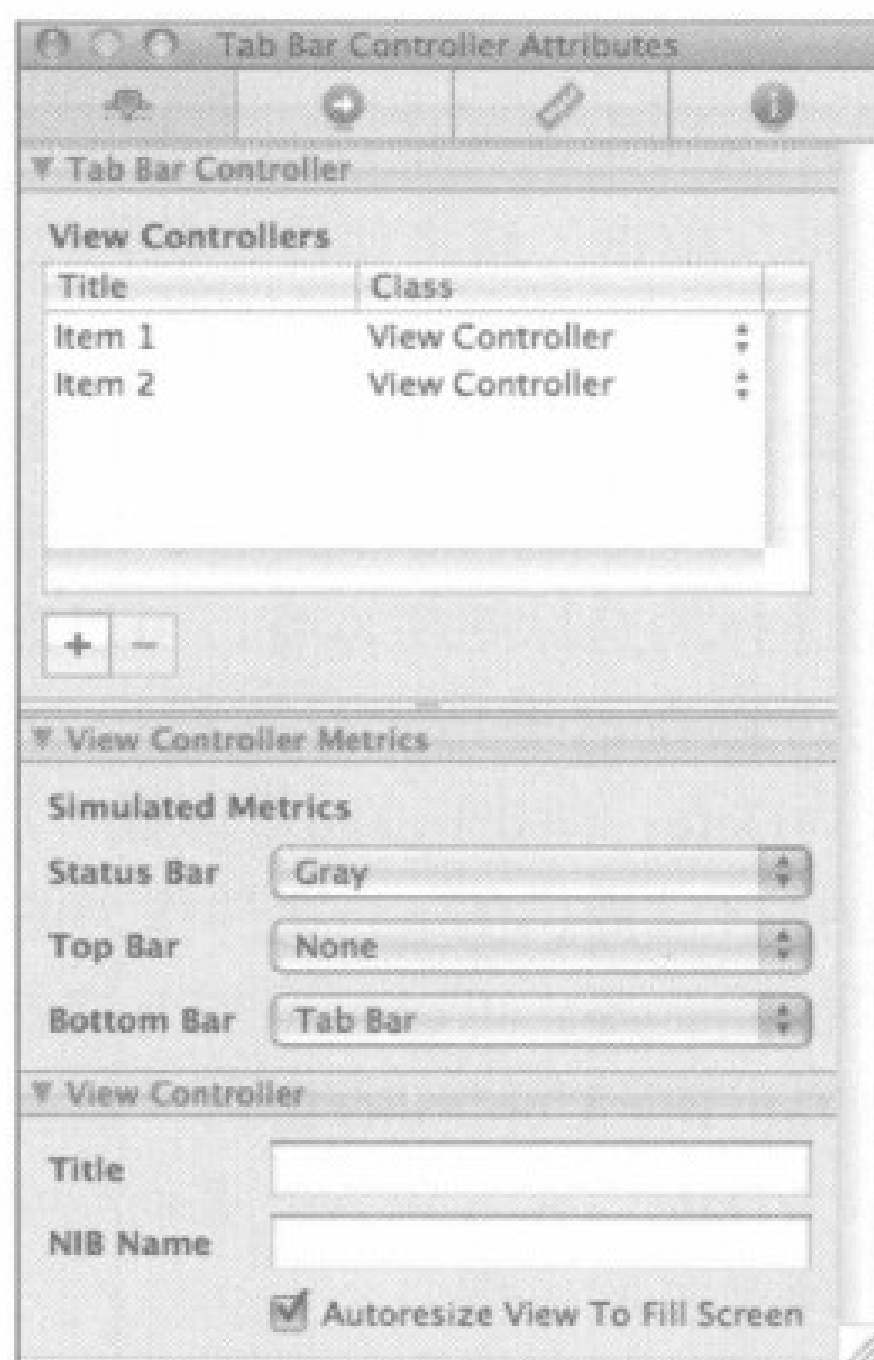


图7-9 标签栏控制器的属性检查器

我们关注的部分位于顶部，也就是View Controllers部分。当所有视图控制器都设置完成之后，标签控制器的每个标签都将拥有一个视图控制器。回顾一下图7-2。可以看到，我们的程序拥有5个标签，分别对应5个子视图——5个子视图需要5个视图控制器。

将注意力转向标签栏控制器的属性检查器。我们需要更改标签栏控制器，使其拥有5个标签，而不是两个。单击带有加号(+)的按钮3次，总共创建5个控制器。属性检查器将显示5个项，如果查看Tab View Controller窗口，将会看到它现在拥有5个按钮，而不是2个。

单击Tab Bar Controller窗口底部的标签栏。一定要单击最左边的标签。这应该选定与最左侧标签相对应的控制器，检查器应该更改为对应的外观，如图7-10所示。

我们在这里将每个标签的视图控制器与合适的nib相关联。最左侧的标签将启动5个子视图中的第1个。保留Title字段为空，将NIB Name指定为DatePickerView。不要包含.xib扩展名。

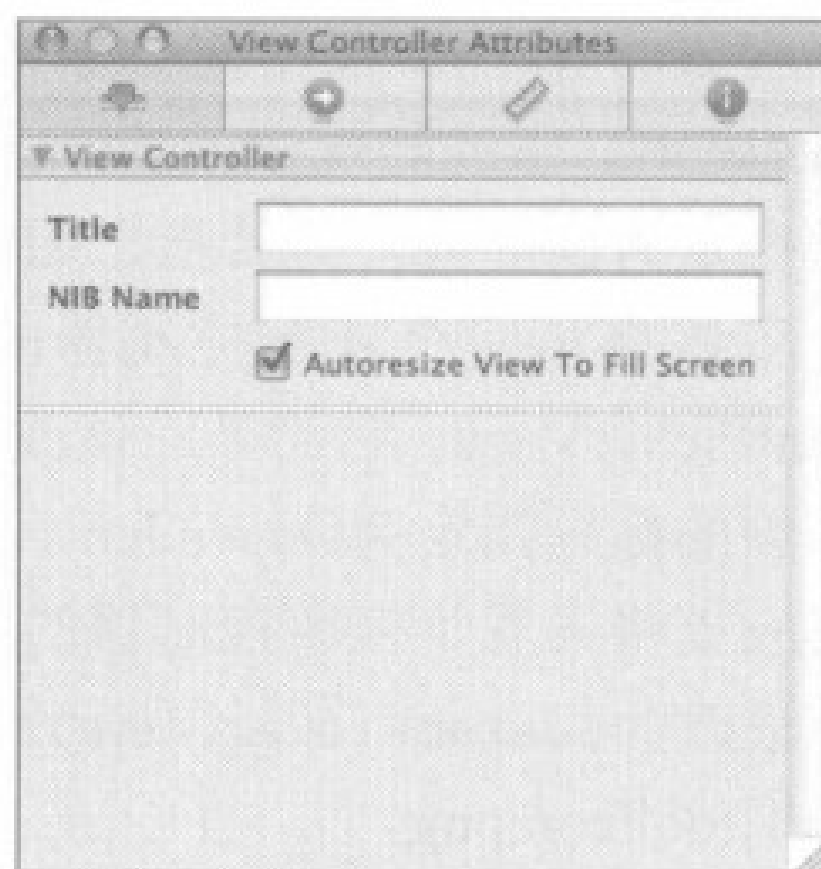


图7-10 视图控制器属性检查器

完成之后按下⌘4，打开与最左侧标签相关联的视图控制器的身份检查器。将类改为DatePickerviewController，并按下return或tab键进行设置。

说明 在此处保留Title字段为空的原因在于，标签栏控制器不会使用此字段，尽管导航控制器和许多其他类型的视图控制器会使用它。

按下⌘1返回到属性检查器。单击标签栏中的第一个标签，片刻之后再次单击。这将使检查器再次更改，如图7-11所示。

通过在同一位置再次单击标签栏，对视图控制器（与标签栏项相关联的）的选择就改成了对标签栏项本身的选择。换句话说，第一次单击选择的是5个子视图控制器中的第一个。第二次单击选择的是标签栏本身，可以设置它的标题和图标。

我们可以在这里指定标签栏项的图标和标题。将Title由Item 1改为Date；单击Image复选框；选择clockicon.png图像。也可以选择自己提供的.png文件作为图标。本章稍后部分将讨论我们提供的资源。可以根据需要调整自己的媒体文件。

查看Tab Bar Controller窗口，将会看到最左侧的标签栏项现在显示为Date，并且包含一张时钟图片。现在需要对另外4个标签栏项重复此过程。

应该将第二个视图控制器的标题指定为Single，并将其nib名称指定为SingleComponentPickerView。在身份检查器中，它的类应该改为SingleComponentPickerviewController。应该将第二个标签栏项的标题指定为Single，它应该使用图标singleicon.png。

应该将第三个视图控制器的标题指定为Double，并将其nib名称指定为DoubleComponentPickerView。它的类应该改为DoubleComponentPickerviewController。应该将第三个标签栏项的标题指定为Double，为其指定图像doubleicon.png。

应该将第四个视图控制器的标题指定为Dependent，并将其nib名称指定为DependentComponentPickerView。它的类应该改为DependentComponentPickerviewController。应该将第四个标签栏项的标题指定为Dependent，为其指定图像dependenticon.png。

应该将第五个视图控制器的标题指定为Custom，将其nib名称指定为CustomPickerView。它的类应该改为CustomPickerviewController。应该将第五个标签栏项的标题指定为Custom，为其指定图像toolicon.png。

接下来要在nib中执行的操作就是，按住Control键并将Pickers App Delegate图标拖到Tab Bar Controller图标，选择rootController输出口。保存nib文件，返回到Xcode。

现在，标签栏和内容视图应该都衔接起来，并且能够正常工作了。编译并运行应用程序，启动之后的应用程序应该包含一个能够正常工作的工具栏，单击一个标签应该会将其选中。现在，内容视图中还没有任何内容，因此，更改将不会很大。但是如果所有组成部分都运行良好，

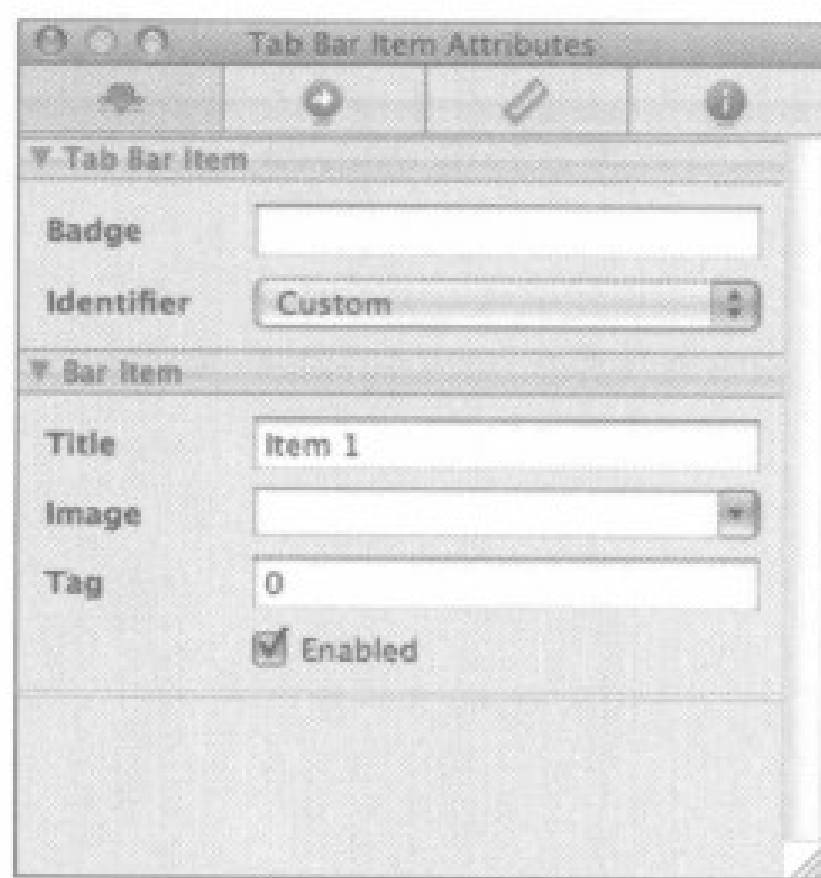


图7-11 标签栏项属性检查器

那么多视图应用程序的基本框架现在就已经建立并能够运行了。接下来可以开始设置每个内容视图了。

提示 如果在单击某个标签时，仿真器出了问题，请不要惊慌！这很可能是因为漏掉了一个步骤，或者出现了输入错误。返回去检查一下所有nib文件名；确保所有连接都正确；并确保所有类名都设置正确。

如果想要进一步确保所有元素都能够正常工作，可以在重新启动应用程序之前，向每个内容视图添加另一个标签或某个其他对象。如果所有元素都运行良好，将会看到不同视图的内容会在选择不同标签时发生改变。

7.4 实现日期选取器

要实现日期选取器，需要一个输出口和一个操作。输出口用于从日期选取器提取值。操作将由一个按钮触发并抛出一个警告，显示从选取器抓取的日期值。单击DatePickerViewController.h，并添加以下代码：

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

@interface DatePickerViewController : UIViewController {
    IBOutlet UIDatePicker *datePicker;
}
@property (nonatomic, retain) UIDatePicker *datePicker;
-(IBAction)buttonPressed;
@end
```

保存此文件，双击DatePickerView.xib，在Interface Builder中打开第一个标签的内容视图。我们首先需要的是一个日期选择器，因此在库中查找Date Picker（参见图7-12），并将其拖到View窗口。如果没有打开View窗口，就在nib的主窗口中双击View图标将其打开。

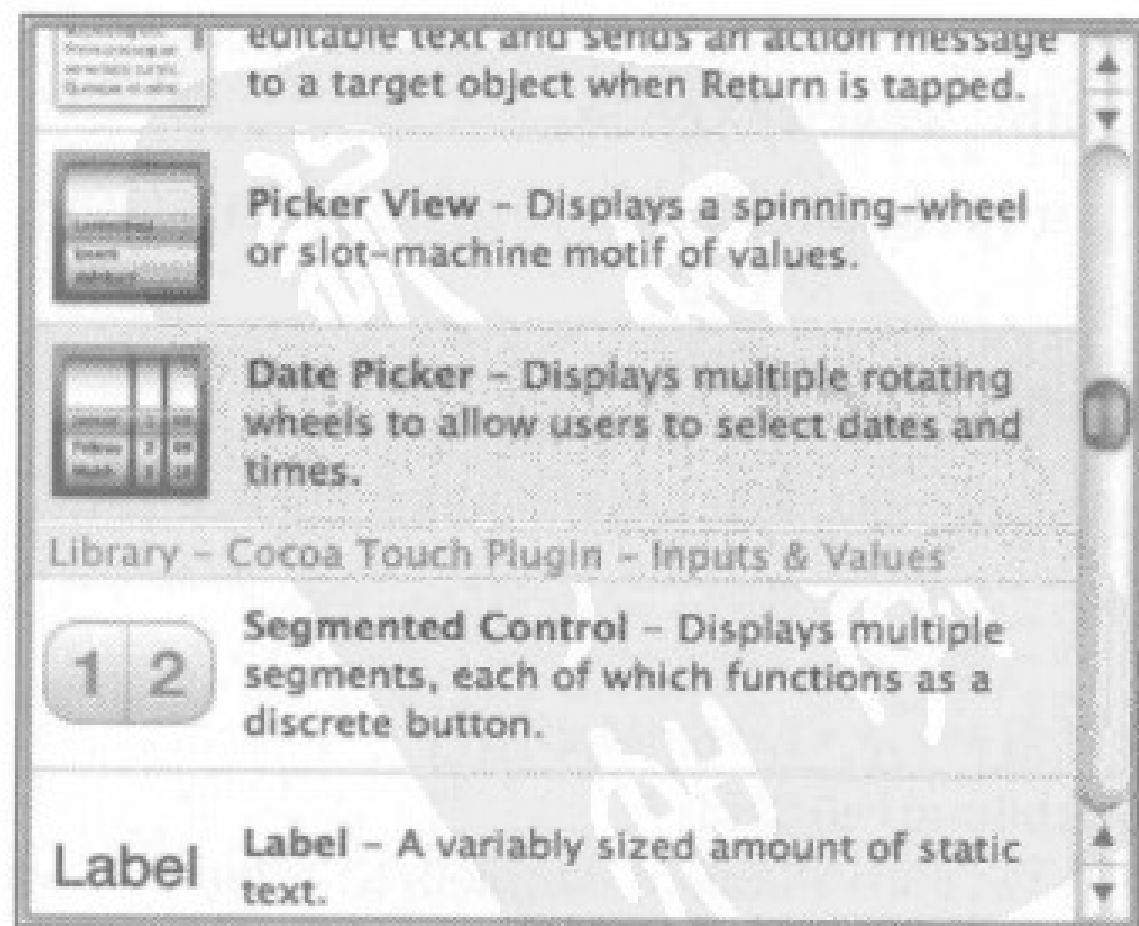


图7-12 库中的Date Picker

将日期选取器放在视图顶部。它应该会占用内容视图的大部分空间。不要为选取器使用蓝色的引导线，它应该会与视图边缘完美接合（参见图7-13）。

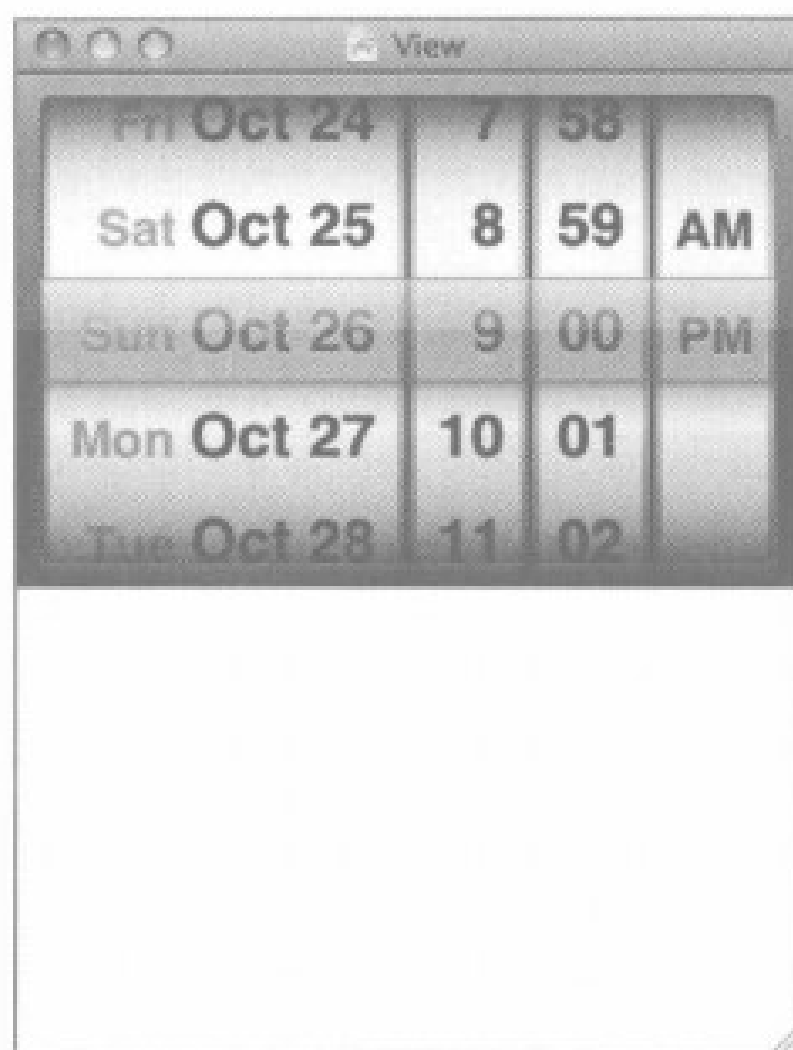


图7-13 将所有选取器都放在视图边缘的正确位置

通过单击操作选中日期选取器，并按下⌘1打开属性检查器。从图7-14中可以看到，可以配置日期选取器的许多属性。在其余选取器中也会碰到这些选项，所以应该仔细了解一下它们的用途。我们将保留大部分值的默认设置，如果要查看每个选项的用途，可以自由选择各种选项。接下来，将选取器的范围限制为合理的日期。将Minimum日期值改为1/1/1900，将Maximum改为12/31/2200。

接下来，从库中拖出一个Round Rect Button，并将其放在日期选取器之下。双击它，将其标题设置为Select，并按下⌘2切换到连接检查器。将Touch Up Inside事件旁边的圆拖到File's Owner图标，并连接到buttonPressed图标。然后按住Control键，并将File's Owner图标拖到日期选取器，选择datePicker输出口。关闭nib，返回到Xcode。

现在只需要实现DatePickerViewController，因此单击DatePickerViewController.m并添加以下代码：

```
#import "DatePickerViewController.h"

@implementation DatePickerViewController
@synthesize datePicker;

- (id)initWithNibName:(NSString *)nibNameOrNil
  bundle:(NSBundle *)nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil])
    {
        // Initialization code
    }
}
```

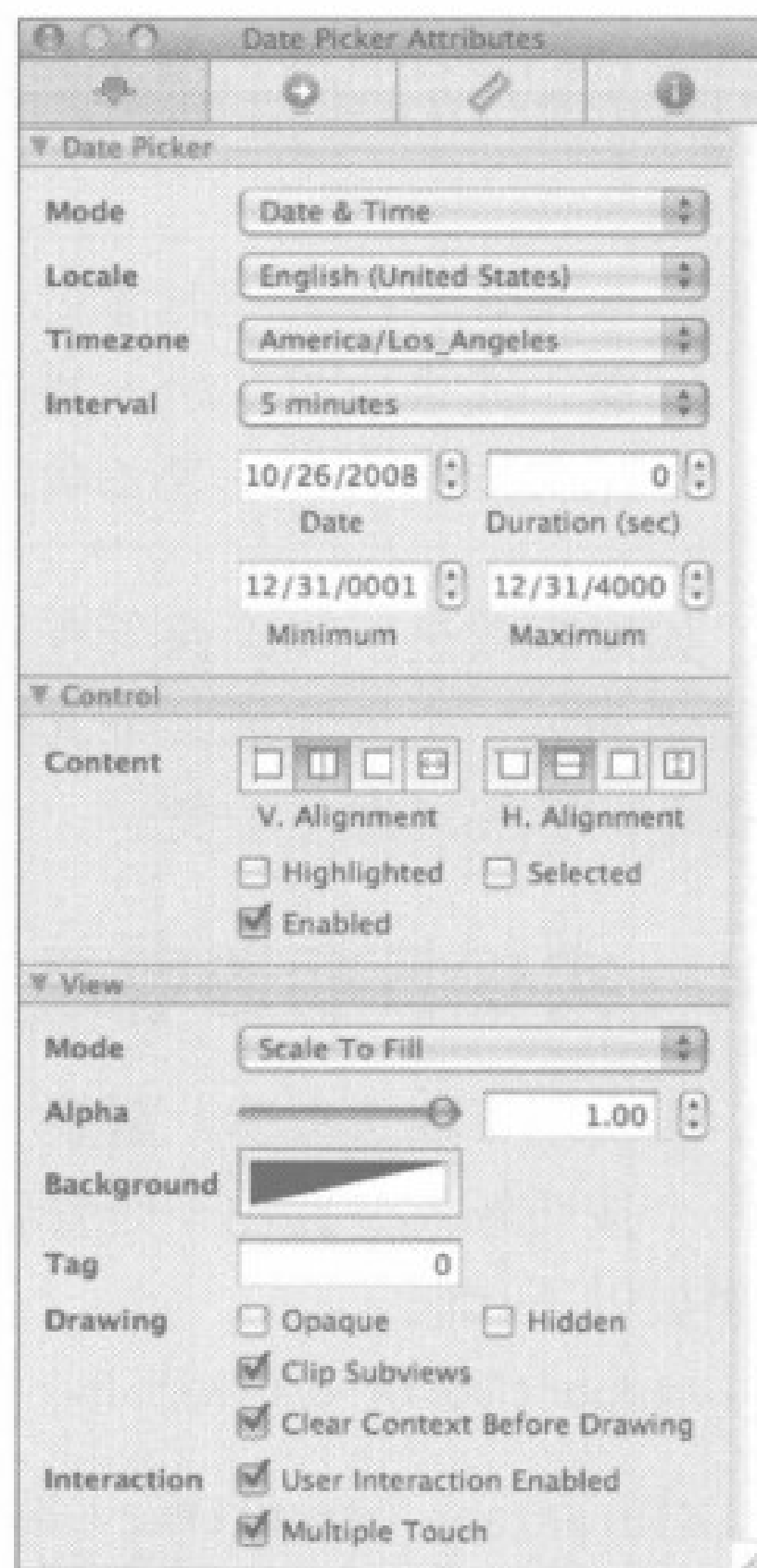


图7-14 日期选取器的属性检查器

```
    }  
    return self;  
}  
-(IBAction)buttonPressed {  
    NSDate *selected = [datePicker date];  
    NSString *message = [[NSString alloc] initWithFormat:  
        @"The date and time you selected is: %@", selected];  
    UIAlertView *alert = [[UIAlertView alloc]  
        initWithTitle:@"Date and Time Selected"  
        message:message  
        delegate:nil  
        cancelButtonTitle:@"Yes, I did."  
        otherButtonTitles:nil];  
    [alert show];  
    [alert release];  
    [message release];  
}  
-(void)viewDidLoad {  
    NSDate *now = [[NSDate alloc] init];  
    [datePicker setDate:now animated:YES];  
    [now release];  
}  
  
-(BOOL)shouldAutorotateToInterfaceOrientation:  
    (UIInterfaceOrientation)interfaceOrientation {  
    // Return YES for supported orientations  
    return (interfaceOrientation == UIInterfaceOrientationPortrait);  
}  
  
-(void)didReceiveMemoryWarning {  
    [super didReceiveMemoryWarning];  
    // Releases the view if it doesn't have a superview  
    // Release anything that's not essential, such as cached data  
}  
  
-(void)dealloc {  
    [datePicker release];  
    [super dealloc];  
}
```

我们添加了buttonPressed的实现，并覆盖了viewDidLoad。在buttonPressed中，我们使用datePicker输出口从日期选取器获取当前的日期值，然后根据该日期构造一个字符串，并使用该字符串显示警告。

在viewDidLoad中，我们创建了一个新的NSDate对象。通过这种方式创建的NSDate对象将包含当前的日期和时间。然后将datePicker设置为该日期，这可以确保每次加载此视图时，选取器都会重置为当前的日期和时间。

编译并运行应用程序，确保选中了日期选取器。如果一切都运行良好，那么运行中的应用程序应该与图7-2中所示类似。如果单击Select按钮，将会弹出一个警告，显示当前在日期选取器中

选定的日期和时间。尽管日期选取器不允许指定秒或时区，但显示选定日期和时间的警告也会显示秒和时区偏移值。我们可以添加一个格式化器来简化警告中显示的字符串，但由于本章篇幅所限，将不再赘述这方面内容。

7.5 实现单个组件选取器

日期选取器非常简单，接下来看一下如何使用支持从一组值中进行选择的选取器。在本示例中，我们将创建一个NSArray来保存想要在选取器中显示的值。选取器本身不会保存任何数据。它们调用其数据源和委托上的方法来获取需要显示的数据。选取器不会关心底层数据位于何处。它在需要时才会请求数据，数据源和委托将通过相互协作来提供该数据。因此，数据可以来自一个静态列表，比如本节中使用的数据，也可以从一个文件或URL载入，甚至随时地组合或计算而来。

7.5.1 声明输出口和操作

通常，在Interface Builder中工作之前，我们需要确保输出口和操作已经位于控制器的头文件中。在Xcode中，单击SingleComponentPickerViewController.h。此控制器类将同时充当选取器的数据源和委托，因此我们需要确保它符合这两个角色的协议。此外，还需要声明一个输出口和一个操作。添加以下代码：

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

@interface SingleComponentPickerViewController : UIViewController
    <UIPickerViewDelegate, UIPickerViewDataSource> {
    IBOutlet UIPickerView *singlePicker;
    NSArray *pickerData;
}
@property (nonatomic, retain) UIPickerView *singlePicker;
@property (nonatomic, retain) NSArray *pickerData;
- (IBAction)buttonPressed;
@end
```

首先，确保控制器类符合UIPickerViewDelegate和UIPickerViewDataSource两个协议。然后，为选取器声明一个输出口和一个指向NSArray的指针，NSArray将用于保存在选取器中显示的数据项。最后，声明按钮的操作方法，就像对日期选取器的操作一样。

7.5.2 构建视图

双击SingleComponentPickerView.xib，打开标签栏的第二个标签的内容视图。从库中选择一个Picker View（参见图7-15），将其添加到nib的View窗口，使其与视图顶部

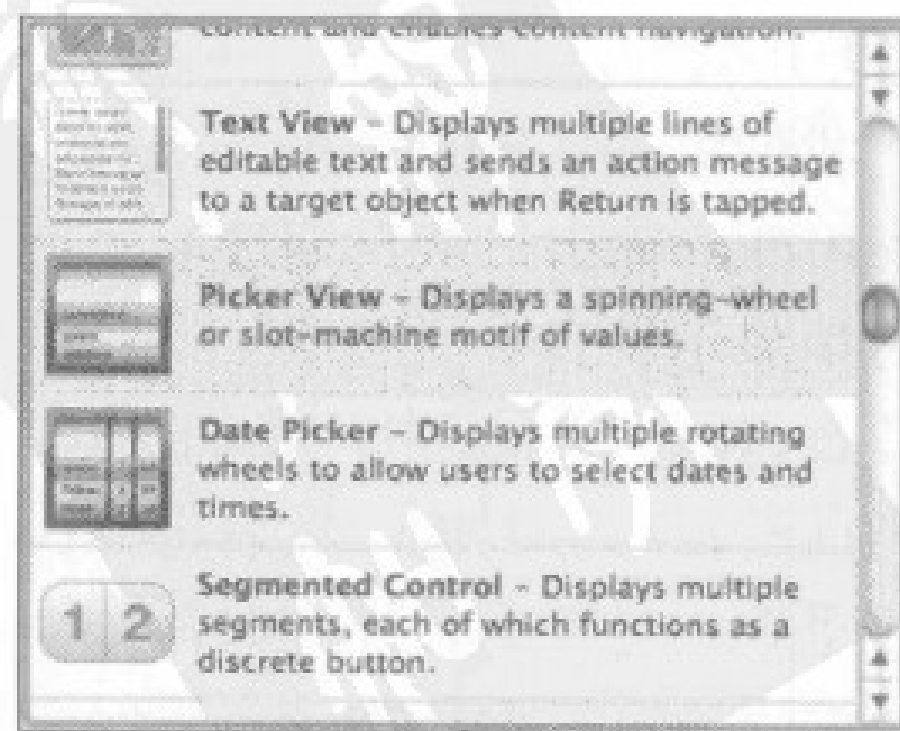


图7-15 库中的选取器视图

吻合，就像日期选取器视图一样。

放置好选取器之后，按住Control键并将File's Owner拖动到选取器视图，然后选择singlePicker输出口。接下来，如果未选中选取器，则单击选中它，按下⌘2打开连接检查器。如果查看选取器视图可用的连接，将会看到前两项是DataSource和Delegate。将DataSource旁边的圆拖到File's Owner图标。然后再次将Delegate旁边的圆拖到File's Owner图标。现在，此选取器知道SingleComponentPickerViewController类的实例就是它的数据源和委托，并且会要求该实例提供要显示的数据。换句话说，当选取器需要将要显示的数据信息时，它将向控制此视图的SingleComponentPickerViewController实例请求该信息。

拖动一个Round Rect Button到该视图，双击该按钮，将其标题设置为Select。按下回车键提交更改。在连接检查器中，将Touch Up Inside旁边的圆拖到File's Owner图标，选择buttonPressed操作。保存并关闭nib文件，然后返回到Xcode。

7.5.3 将控制器实现为数据源和委托

要让控制器充当选取器的数据源和委托，我们需要实现一些你从未见过的新方法。单击SingleComponentPickerViewController.m，并添加以下代码：

```
#import "SingleComponentPickerViewController.h"

@implementation SingleComponentPickerViewController
@synthesize singlePicker;
@synthesize pickerData;

- (id)initWithNibName:(NSString *)nibNameOrNil
  bundle:(NSBundle *)nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil
      bundle:nibBundleOrNil]) {
      // Initialization code
    }
    return self;
}

- (IBAction)buttonPressed
{
    NSInteger row = [singlePicker selectedRowInComponent:0];
    NSString *selected = [pickerData objectAtIndex:row];
    NSString *title = [[NSString alloc] initWithFormat:
      @"You selected %@!", selected];
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:title
      message:@"Thank you for choosing."
      delegate:nil
      cancelButtonTitle:@"You're Welcome"
      otherButtonTitles:nil];
    [alert show];
    [alert release];
    [title release];
}
```



```

}
- (void)viewDidLoad {
    NSArray *array = [[NSArray alloc] initWithObjects:@"Luke", @"Leia",
        @"Han", @"Chewbacca", @"Artoo", @"Threepio", @"Lando", nil];
    self.pickerData = array;
    [array release];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [singlePicker release];
    [pickerData release];
    [super dealloc];
}
#pragma mark -
#pragma mark Picker Data Source Methods
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView
{
    return 1;
}

- (NSInteger)pickerView:(UIPickerView *)pickerView
    numberOfRowsInComponent:(NSInteger)component
{
    return [pickerData count];
}
#pragma mark Picker Delegate Methods
- (NSString *)pickerView:(UIPickerView *)pickerView
    titleForRow:(NSInteger)row
    forComponent:(NSInteger)component
{
    return [pickerData objectAtIndex:row];
}
@end

```

你现在应该熟悉前两个方法了。`buttonPressed`方法与日期选取器中使用的对应方法几乎一样。与日期选取器不同，常规的选取器无法告诉我们它包含的数据，因为它没有维护这些数据。它将此工作交由委托和数据源处理。我们需要询问选取器哪一行已被选中，然后从`pickerData`数组提取相应的数据。

以下是询问选取器所选行的方法：

```
NSInteger row = [singlePicker selectedRowInComponent:0];
```

注意，我们需要指定想要了解的组件。此选取器中只有一个组件，因此我们传入0，这是第一个组件的索引。

注意 注意到NSInteger与Row之间没有星号了吗？在iPhone中，虽然前缀“NS”通常表示来自基础框架的Objective-C类，但此处是这个一般规则的一个例外。NSInteger始终定义为整数数据类型，无论是int还是long。我们使用NSInteger，而没有使用int或long，因为当使用NSInteger时，编译器将自动选择最适合目标平台的整数类型。当针对32位处理器编译时，编译器将创建一个32位int，当针对64位体系结构编译时，它将创建一个64位long。目前还没有64位的iPhone，但是谁知道以后会不会有呢？也可以为iPhone应用程序编写类便于以后在面向Mac OS X的Cocoa应用程序中使用，Mac OS X能够同时支持32位和64位应用程序。

7

在viewDidLoad中，我们创建了一个包含几个对象的数组，用于向选取器提供数据。通常而言，数据来自于其他数据源，比如项目的Resources文件夹中的属性列表。利用此处的方式在代码中嵌入一组项，将难以更新此列表或将应用程序转换为其他语言。但出于演示目的，这种方法是将数据获取到数组中的最快和最简单的方式。即使你通常未采用这种方式创建数组，也会总是将使用的数据缓存到viewDidLoad方法中的一个数组中，这样，不必在选取器每次请求数据时都访问磁盘或网络。

提示 如果不打算像我们刚才在viewDidLoad中做的那样，在代码中创建一个包含一组对象的数组，那么应该如何操作呢？将对象列表嵌入到属性列表文件中，并将这些文件添加到项目的Resources文件夹中。无需重新编译源代码就可以更改属性列表文件，这意味着在更改时不会引入新的错误。你也可以为不同语言提供不同的列表版本，第17章将介绍这种方法。可以使用位于/Developer/Applications/Utilities/Property List Editor.app中的Property List Editor应用程序来创建属性列表，或者在Xcode中创建属性列表，Xcode支持在编辑器窗格中编辑属性列表。NSArray和NSDictionary都提供了一个名为initWithContentsOfFile:的方法，以支持初始化属性文件中的实例，本章稍后在实现Dependent标签时将采用这种方法。

文件底部包含实现选取器所需的新方法。dealloc之后的前两个方法来自UIPickerView-DataSource协议，所有选取器（除了日期选取器）都需要它们。下面是第一个方法：

```
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView
{
    return 1;
}
```

选取器可以包含多个旋转滚轮或组件，这就是选取器会询问应该显示几个组件的原因。这一次，我们只想显示一个列表，因此返回值1。注意，`UIPickerView`将作为参数传入。此参数指向询问此问题的选取器视图，这使同一个数据源能够控制多个选取器。在本例中，我们知道只有一个选取器，可以放心地忽略此参数，因为我们已经知道需要哪个选取器。

选取器使用第二种数据源方法询问给定组件包含多少行数据：

```
- (NSInteger)pickerView:(UIPickerView *)pickerView
    numberOfRowsInComponent:(NSInteger)component
{
    return [pickerData count];
}
```

再一次，我们知道哪个选取器视图正在询问问题，以及该选取器询问的是关于哪个组件的信息。因为我们知道只有一个选取器和一个组件，因此无需理会这两个参数，只需返回数据数组中的对象计数。

在两个数据源方法之后，我们实现了一个委托方法。与数据源方法不同，所有委托方法都是可选的。术语“可选”具有一定的欺骗性，因为必须至少实现一个委托方法。你通常会实现我们在此处实现的方法。在处理自定义选取器时，如果想要在选取器中显示除文本之外的其他内容，则必须实现一个不同的方法。

```
- (NSString *)pickerView:(UIPickerView *)pickerView
    titleForRow:(NSInteger)row
    forComponent:(NSInteger)component
{
    return [pickerData objectAtIndex:row];
}
```

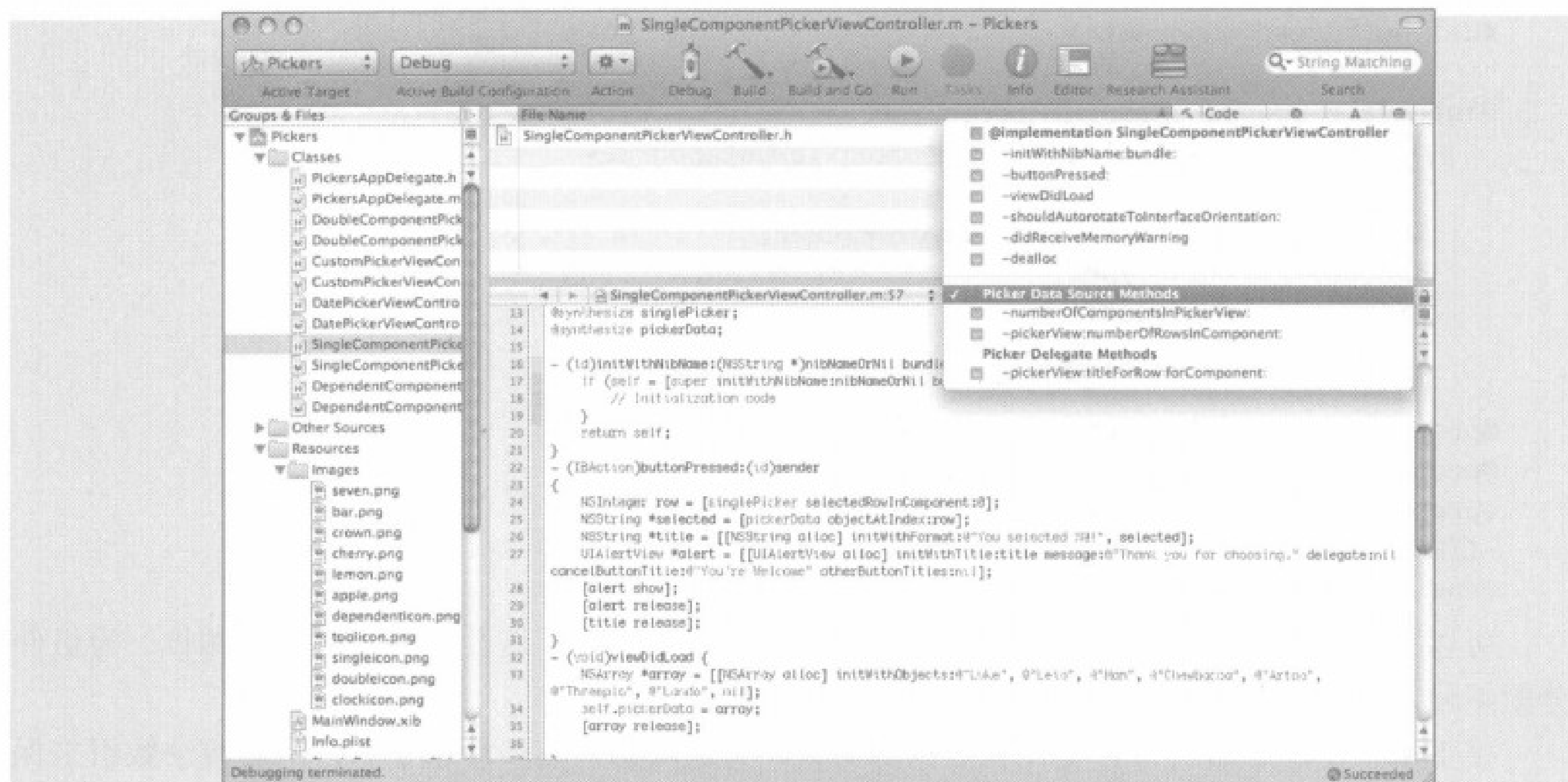
#pragma是什么

注意到SingleComponentPickerViewController.m中的以下代码了吗？

```
#pragma mark -
#pragma mark Picker Data Source Methods
```

从技术上讲，任何以`#pragma`开头的代码都是一条编译器指令，具体来讲，是一个特定于程序或特定于编译器的指令，它们不一定适用于其他编译器或其他环境。如果编译器不能识别该指令，则会将其忽略，但可能会生成一个警告。在这种情况下，`#pragma`指令实际上是针对IDE的指令，而与编译器无关，它们告诉Xcode的编辑器，要在编辑器窗格顶部的方法和函数弹出菜单中将代码分隔开，如下面的屏幕截图所示。第一个指令在菜单中添加了一个分隔符。第二个指令创建了一个粗体条目。

一些类（尤其是一些控制器类）可能很长，方法和函数弹出菜单可以便于代码导航。加入`#pragma`指令并对代码进行逻辑组织，可以使弹出菜单变得更加有效。



在此方法中，选取器要求提供关于指定组件中指定行的数据。我们提供一个指向请求数据的选取器的指针，以及它请求的组件和行。因为我们的视图只有一个选取器，且该选取器只有一个组件，因此可以忽略除row参数之外的其他参数，并使用row参数返回数据数组中的合适项。

再次编译并运行应用程序。当仿真器出现时，切换到第二个标签（标为Single的标签），并检查新的自定义选取器，它应该与图7-3类似。

温习一下刚才介绍的所有内容，然后返回Xcode，我们将讨论如何实现带有两个组件的选取器。如果想要挑战一下自己，那么第二个内容视图实际上是一个很好的演练机会。你已经知道此选取器需要的所有方法，因此可以实际练习一下。你可能首先想实现图7-4所示的外观，那么只需回顾刚才介绍的方法。然后继续阅读，你将会看到我们如何实现这个成果。

7.6 实现多组件选取器

下一个内容窗格将包含一个带有两个组件或滚轮的选取器。每个滚轮之间彼此独立。左侧的滚轮将包含一个三明治馅料列表，右侧的滚轮包含各种面包类型。刚才已经提到，要编写的数据源和委托方法与为单个组件选取器编写的方法相同。我们只需在一些方法中编写少量代码，以确保为每个组件返回正确的值和行数。

7.6.1 声明输出口和操作

单击`DoubleComponentPickerController.h`，并添加以下代码：

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

```
#define kFillingComponent 0
```

```
#define kBreadComponent 1

@interface DoubleComponentPickerViewController : UIViewController
    <UIPickerViewDelegate, UIPickerViewDataSource>
{
    IBOutlet UIPickerView *doublePicker;
    NSArray *fillingTypes;
    NSArray *breadTypes;

}
@property(n nonatomic, retain) UIPickerView *doublePicker;
@property(n nonatomic, retain) NSArray *fillingTypes;
@property(n nonatomic, retain) NSArray *breadTypes;
-(IBAction)buttonPressed;
@end
```

可以看到，我们首先定义了两个常量，它们表示两个组件，这会使代码更容易阅读。为组件分配编号，最左侧的组件的编号为0，向右依次递增。

接下来，使控制器类符合委托和数据源协议，为选取器声明一个输出口，声明两个数组来保存两个选取器组件的数据。声明了每个实例变量的属性之后，为按钮声明一个操作方法，就像在前面两个内容窗格中那样。保存工作，双击DoubleComponentPickerView.xib，在Interface Builder中打开该nib文件。

7.6.2 构建视图

在视图中添加一个选取器和一个按钮，然后创建必要的连接。我们不再讨论连接过程，如果需要逐步指导，可以参考7.5节，因为两个应用程序的nib文件都是一样的。下面总结需要做的工作。

- ❑ 将File's Owner上的doublePicker连接到选取器。
- ❑ 将选取器视图上的DataSource和Delegate连接到File's Owner（使用连接检查器）。
- ❑ 将按钮的Touch Up Inside事件连接到File's Owner上的buttonPressed操作（使用连接检查器）。

确保保存并关闭了nib文件，然后返回Xcode。可以在这一页上做个标记（或者如果喜欢，添加一个书签）。稍后将会参考本页内容。

7.6.3 实现控制器

单击DoubleComponentPickerViewController.m，并添加以下代码：

```
#import "DoubleComponentPickerViewController.h"

@implementation DoubleComponentPickerViewController
@synthesize doublePicker;
@synthesize fillingTypes;
@synthesize breadTypes;
- (id)initWithNibName:(NSString *)nibNameOrNil
```

```
bundle:(NSBundle *)nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil
        bundle:nibBundleOrNil]) {
        // Initialization code
    }
    return self;
}

- (IBAction)buttonPressed
{
    NSInteger breadRow = [doublePicker selectedRowInComponent:
        kBreadComponent];
    NSInteger fillingRow = [doublePicker selectedRowInComponent:
        kFillingComponent];

    NSString *bread = [breadTypes objectAtIndex:breadRow];
    NSString *filling = [fillingTypes objectAtIndex:fillingRow];

    NSString *message = [[NSString alloc] initWithFormat:
        @"Your %@ on %@ bread will be right up.", filling, bread];

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:
        @"Thank you for your order"
        message:message
        delegate:nil
        cancelButtonTitle:@"Great!"
        otherButtonTitles:nil];
    [alert show];
    [alert release];
    [message release];
}

- (void)viewDidLoad {
    NSArray *breadArray = [[NSArray alloc] initWithObjects:@"White",
        @"Whole Wheat", @"Rye", @"Sourdough", @"Seven Grain", nil];
    self.breadTypes = breadArray;
    [breadArray release];

    NSArray *fillingArray = [[NSArray alloc] initWithObjects:@"Ham",
        @"Turkey", @"Peanut Butter", @"Tuna Salad",
        @"Chicken Salad", @"Roast Beef", @"Vegemite", nil];
    self.fillingTypes = fillingArray;
    [fillingArray release];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
```



```

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [doublePicker release];
    [breadTypes release];
    [fillingTypes release];
    [super dealloc];
}

#pragma mark -
#pragma mark Picker Data Source Methods
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView
{
    return 2;
}

- (NSInteger)pickerView:(UIPickerView *)pickerView
    numberOfRowsInComponent:(NSInteger)component
{
    if (component == kBreadComponent)
        return [self.breadTypes count];

    return [self.fillingTypes count];
}

#pragma mark Picker Delegate Methods
- (NSString *)pickerView:(UIPickerView *)pickerView
    titleForRow:(NSInteger)row
    forComponent:(NSInteger)component
{
    if (component == kBreadComponent)
        return [self.breadTypes objectAtIndex:row];

    return [self.fillingTypes objectAtIndex:row];
}

@end

```

这一次，`buttonPressed`方法稍微有点复杂，但是其中绝大部分代码都是我们所熟悉的，我们只需使用前面定义的`kBreadComponent`和`kFillingComponent`常量指定选定行所对应的组件。

```

NSInteger breadRow = [doublePicker selectedRowInComponent:
    kBreadComponent];
NSInteger fillingRow = [doublePicker selectedRowInComponent:
    kFillingComponent];

```

可以看到，这里使用了两个常量来代替0和1，这使代码更具有可读性。后面使用的`buttonPressed`方法与我们编写的上一个版本基本相同。

`biewDidLoad:`也与前一节中编写的版本非常类似。唯一的区别在于，我们载入了两个包含数据的数组，而不是一个。再一次，我们从硬编码的字符串列表创建数组，你一般不应该在自己的

应用程序中这么做。

接下来看一下数据源方法，从这里开始，将对代码进行较大的更改。在第一个方法中，指定选取器应该拥有两个组件，而不是一个：

```
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView
{
    return 2;
}
```

非常简单。这一次，当要求提供行数时，我们必须检查选取器询问的是哪个组件，并返回相应数组的正确行数：

```
- (NSInteger)pickerView:(UIPickerView *)pickerView
    numberOfRowsInComponent:(NSInteger)component
{
    if (component == kBreadComponent)
        return [self.breadTypes count];

    return [self.fillingTypes count];
}
```

然后，在委托方法中执行相同的操作。检查组件并使用正确的数组供被请求的组件提取和返回正确的值。

```
- (NSString *)pickerView:(UIPickerView *)pickerView
    titleForRow:(NSInteger)row
    forComponent:(NSInteger)component
{
    if (component == kBreadComponent)
        return [self.breadTypes objectAtIndex:row];

    return [self.fillingTypes objectAtIndex:row];
}
```

这不是很难，对吗？编译并运行应用程序，并确保Double内容窗格与图7-4类似。注意，每个滚轮之间是完全独立的。旋转一个滚轮不会影响到另一个。这正适合此处的情形。但有时候，一个组件将依赖于另一个。日期选取器就是一个恰当的例子。当更改月份时，显示每月天数的刻度盘可能需要更改，因为不是所有月份都拥有相同的天数。只要知道操作方法，实现这项功能实际上并不难，但是独自解决此问题并不容易，所以接下来我们将看一下如何操作。

7.7 实现独立组件

在本节中，我们不打算详细讨论前面已经介绍过的内容。我们将主要介绍一些新知识。新选取器将在左侧组件中显示一组美国的州，在右侧组件中显示与当前在左侧选定的州相对应的ZIP编号。

左侧组件中的每个项都需要一个独立的ZIP编号列表。与上一节一样，我们将声明两个数组，分别对应每个组件。还需要一个NSDictionary字典。在字典中，每个州都有一个对应的NSArray

(参见图7-16)。随后,实现一个委托方法,该方法将在选取器的选定项改变时通知我们。如果左侧的值改变,我们将从字典中提取正确的数组,并将其分配给右侧组件所使用的数组。如果未能获取所有数组,不要担心,我们将在深入分析代码时讨论这一点。

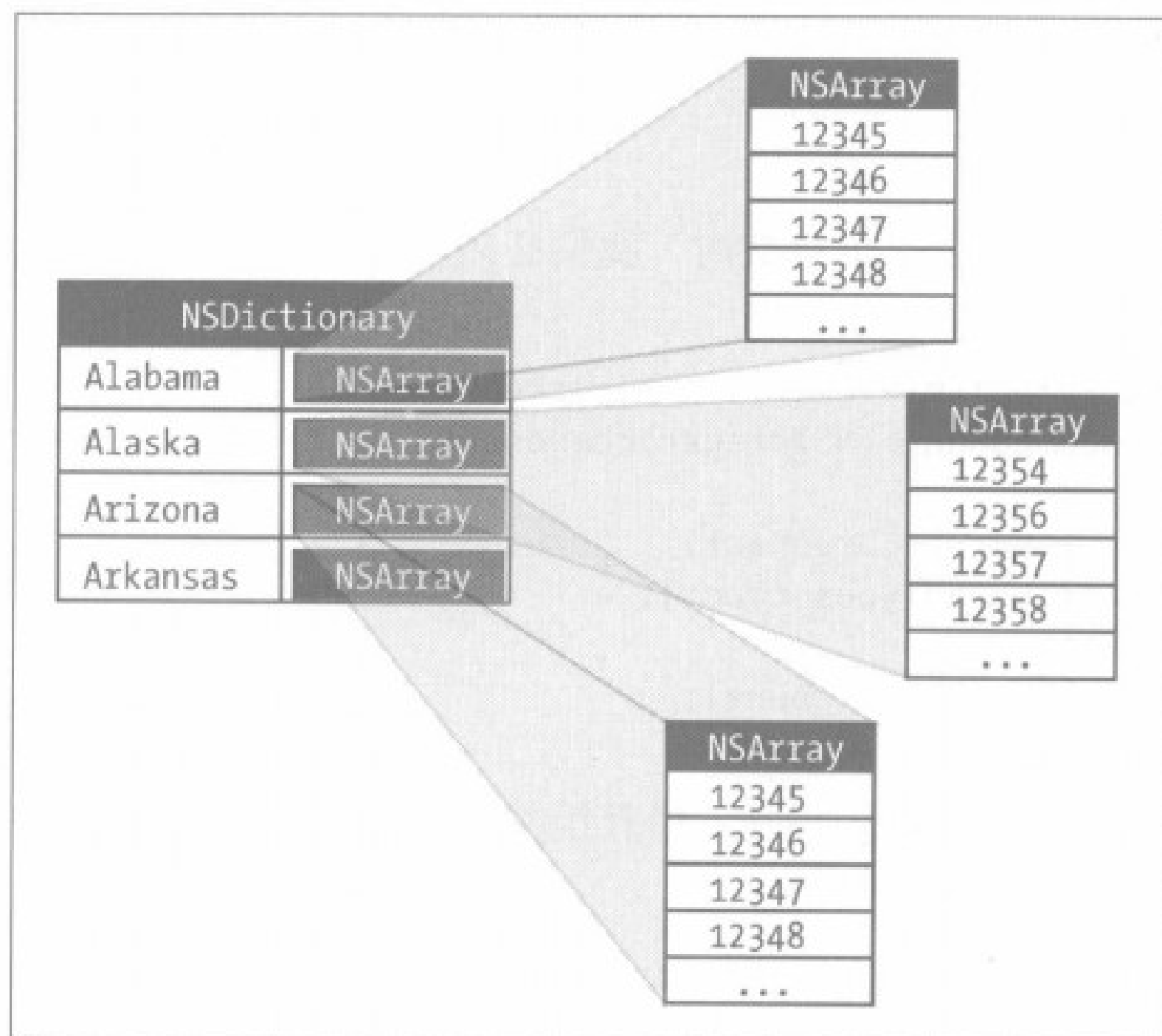


图7-16 应用程序的数据:对于每个州,字典里都有与之对应的、使用州名作为键的条目。该键字存储的是一个NSArray实例,其中包含该州的所有ZIP编号

将以下代码添加到DependentComponentPickerViewController.h文件中:

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

```
#define kStateComponent 0
```

```
#define kZipComponent 1
```

```
@interface DependentComponentPickerViewController : UIViewController
    <UIPickerViewDelegate, UIPickerViewDataSource>
```

```
{
    IBOutlet UIPickerView *picker;
```

```
    NSDictionary *stateZips;
```

```
    NSArray *states;
```

```
    NSArray *zips;
```

```
}
```

```
@property (retain, nonatomic) UIPickerView *picker;
```

```
@property (retain, nonatomic) NSDictionary *stateZips;
```

```
@property (retain, nonatomic) NSArray *states;
```

```
@property (retain, nonatomic) NSArray *zips;
```

```
- (IBAction)buttonPressed;
```

```
@end
```

现在,使用Interface Builder构建一个内容视图。构建过程与前面构建两个组件视图的过程大

体相同。如果忘记了具体操作，可以查看7.5.2节，并按照其中的分步说明进行操作。这里要注意一点：首先应打开DependentComponentPickerView.xib。完成之后，保存并关闭nib文件，然后返回Xcode。

接下来，我们将实现这个控制器类。刚开始可能会觉得此实现有点怪异。通过让一个组件依赖于另一个组件，我们使控制器类的复杂度变得更高了。虽然选取器一次只能显示两个列表，但控制器类必须管理51个列表。此处所使用的技巧可以简化这个过程。数据源方法看起来与实现DoublePicker视图的方法几乎相同。所有增加的复杂度都在viewDidLoad和一个新的委托方法pickerView:didSelectRow:inComponent:之间进行处理。

在编写代码之前，我们需要创建要显示的数据。目前为止，我们已经通过指定字符串列表在代码中创建了一些数组。但是现在，我们不再打算采用这种方式。由于我们不希望输入数千个值，并且希望通过合适的方式解决这个问题，所以我们将从一个属性列表载入数据。前面已经提到，NSArray和NSDictionary对象都可以通过属性列表创建。我们已经在项目归档文件的07 Pickers文件夹中包含了一个属性列表statedictionary.plist。将该文件导入Xcode项目中并单击它，可以查看甚至编辑其中的数据（参见图7-17）。

State	Type	Items
Root	Dictionary	(50 items)
Alabama	Array	(657 items)
Alaska	Array	(251 items)
Arizona	Array	(376 items)
Arkansas	Array	(618 items)
California	Array	(1757 items)
Colorado	Array	(501 items)
Connecticut	Array	(276 items)
Delaware	Array	(68 items)
Florida	Array	(972 items)
Georgia	Array	(736 items)
Hawaii	Array	(92 items)
Item 1	String	96701
Item 2	String	96703
Item 3	String	96704
Item 4	String	96705
Item 5	String	96706
Item 6	String	96707
Item 7	String	96708
Item 8	String	96710
Item 9	String	96712
Item 10	String	96713
Item 11	String	96714
Item 12	String	96716
Item 13	String	96717
Item 14	String	96718
Item 15	String	96719
Item 16	String	96720

图7-17 statedictionary.plist文件

现在编写一些代码。将以下代码添加到DependentComponentPickerViewController.m，然后我们将分块进行讨论：

```
#import "DependentComponentPickerViewController.h"

@implementation DependentComponentPickerViewController
@synthesize picker;
```

```
@synthesize stateZips;
@synthesize states;
@synthesize zips;

- (id)initWithNibName:(NSString *)nibNameOrNil
  bundle:(NSBundle *)nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil
      bundle:nibBundleOrNil]) {
      // Initialization code
    }
    return self;
}

- (IBAction)buttonPressed
{
    NSInteger stateRow = [picker selectedRowInComponent:kStateComponent];
    NSInteger zipRow = [picker selectedRowInComponent:kZipComponent];

    NSString *state = [self.states objectAtIndex:stateRow];
    NSString *zip = [self.zips objectAtIndex:zipRow];
    NSString *title = [[NSString alloc] initWithFormat:
      @"You selected zip code %@.", zip];
    NSString *message = [[NSString alloc] initWithFormat:
      @"%@ is in %@", zip, state];

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:title
      message:message
      delegate:nil
      cancelButtonTitle:@"OK"
      otherButtonTitles:nil];
    [alert show];
    [alert release];
    [title release];
    [message release];
}

- (void)viewDidLoad {

    NSBundle *bundle = [NSBundle mainBundle];
    NSString *plistPath = [bundle pathForResource:
      @"statedictionary" ofType:@"plist"];

    NSDictionary *dictionary = [[NSDictionary alloc]
      initWithContentsOfFile:plistPath];
    self.stateZips = dictionary;
    [dictionary release];

    NSArray *components = [self.stateZips allKeys];
    NSArray *sorted = [components sortedArrayUsingSelector:
      @selector(compare:)];
    self.states = sorted;
```

```
NSString *selectedState = [self.states objectAtIndex:0];
NSArray *array = [stateZips objectForKey:selectedState];
self.zips = array;
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [picker release];
    [stateZips release];
    [states release];
    [zips release];
    [super dealloc];
}

#pragma mark -
#pragma mark Picker Data Source Methods
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView
{
    return 2;
}

- (NSInteger)pickerView:(UIPickerView *)pickerView
    numberOfRowsInComponent:(NSInteger)component
{
    if (component == kStateComponent)
        return [self.states count];
    return [self.zips count];
}

#pragma mark Picker Delegate Methods
- (NSString *)pickerView:(UIPickerView *)pickerView
    titleForRow:(NSInteger)row
    forComponent:(NSInteger)component
{
    if (component == kStateComponent)
        return [self.states objectAtIndex:row];
    return [self.zips objectAtIndex:row];
}

- (void)pickerView:(UIPickerView *)pickerView
```



```

        didSelectRow:(NSInteger)row
        inComponent:(NSInteger)component
    {
        if (component == kStateComponent)
        {
            NSString *selectedState = [self.states objectAtIndex:row];
            NSArray *array = [stateZips objectForKey:selectedState];
            self.zips = array;
            [picker selectRow:0 inComponent:kZipComponent animated:YES];
            [picker reloadComponent:kZipComponent];
        }
    }
@end

```

无需再讨论这里的buttonPressed方法了，它与上一个版本基本一样。但是，我们应该看一下viewDidLoad方法。这里有一些内容需要理解，所以我们将仔细讨论。

在这个新的viewDidLoad方法中，我们做的第一件事情上就提取对应用程序的主束（main bundle）的引用。

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

那么什么是束呢？束只是一种特定的文件类型，其中的内容遵循特定的结构。应用程序和框架都是束，此调用返回的束对象表示我们的应用程序。NSBundle的一个主要作用是获取添加到项目的Resources文件夹的资源。在构建应用程序时，这些文件将被复制到应用程序的束中。我们已经在项目中添加了图像等资源，但是到现在为止，我们还只是在Interface Builder中使用它们。如果想要在代码中使用这些资源，则必须使用NSBundle。我们将使用主束来获取需要的资源路径：

```
NSString *plistPath = [bundle pathForResource:@"statedictionary"
ofType:@"plist"];
```

这将返回一个字符串，其中包含statedictionary.plist文件的位置。然后可以使用该路径创建一个NSDictionary对象。当这样做时，属性列表的所有内容将被载入到新创建的NSDictionary对象中。然后，将该对象分配给stateZips。

```
NSDictionary *dictionary = [[NSDictionary alloc]
initWithContentsOfFile:plistPath];
self.stateZips = dictionary;
[dictionary release];
```

刚才载入的字典使用州名作为键，并且包含一个NSArray，其中包含所有州的ZIP编号。为了填充左侧组件的数组，我们从字典获取所有键的列表，并将这些键分配给states数组。在分配数组之前，对其中的值按字母顺序进行排序。

```
NSArray *components = [self.stateZips allKeys];
NSArray *sorted = [components sortedArrayUsingSelector:
@selector(compare:)];
self.states = sorted;
```

除非将选择设置为另一个值，否则选取器将从选择的第一行开始（行0）。为了获取与states数组中的第一行相对应的zips数组，我们从states数组提取索引为0的对象。这将返回启动时默

认选择的州名。然后使用这个州名提取该州的ZIP编号数组，将这个数组分配给zip数组，zip数组将用于向右侧组件提供数据。

```
NSString *selectedState = [self.states objectAtIndex:0];
NSArray *array = [stateZips objectForKey:selectedState];
self.zip = array;
```

两个数据源方法实际上都与其上一个版本相同，用于返回合适数组中的行数。我们实现的一个委托方法也与其上一个版本相同。而第二个委托方法是全新的，这正是魔力所在：

```
- (void)pickerView:(UIPickerView *)pickerView
    didSelectRow:(NSInteger)row
    inComponent:(NSInteger)component
{
    if (component == kStateComponent)
    {
        NSString *selectedState = [self.states objectAtIndex:row];
        NSArray *array = [stateZips objectForKey:selectedState];
        self.zip = array;
        [picker selectRow:0 inComponent:kZipComponent animated:YES];
        [picker reloadComponent:kZipComponent];
    }
}
```

只要选取器的选择发生变化，就会调用这个方法，我们看一下该组件，并看看左侧的组件是否发生了改变。如果它改变了，我们就提取对应于新选择的数组，并将其分配给zip数组。然后，将右侧组件设置为第一行，并告诉他重新加载自己。通过在州改变时交换zip数组，可以使余下的代码保持与DoublePicker示例中的代码一样。

我们的工作还没完成。编译并运行应用程序，检查Dependent标签，如图7-18所示。是否存在不合意的地方？

两个组件的大小相同。即使ZIP编号不超过5个字符，它也会与州占用同样的空间。一半的选取器宽度无法完全显示Mississippi（密西西比州）和Massachusetts（马萨诸塞州）这样的州，这似乎不太理想。幸运的是，可以实现另一个委托方法来指定每个组件应该占用的宽度。在纵向模式下，选取器组件的可用宽度大约为295像素，但是对于添加的每个附加组件，可能没有空间绘制新组件的边缘。也许需要调整组件的值以获得最佳的显示效果。将以下方法添加到DependentComponent-



图7-18 是否要将两个组件设置成相同的大小

PickerViewController.m的委托部分:

```
- (CGFloat)pickerView:(UIPickerView *)pickerView
    widthForComponent:(NSInteger)component
{
    if (component == kZipComponent)
        return 90;
    return 200;
}
```

在这个方法中, 我们返回了一个数字, 代表每个组件应该具有的宽度(以像素为单位), 选取器将尽可能适应这个宽度值。保存应用程序, 编译并运行, **Dependent**标签上的选取器将与图7-5更加类似。

现在, 你应该对选取器和标签栏应用程序有了一定的了解。对于选取器, 我们还有一些工作要做, 接下来的工作将更加有趣。下一节将创建一个简单的老虎机游戏。

7.8 使用自定义选取器创建简单游戏

现在, 我们将创建一个实际的老虎机游戏。当然, 虽然这个老虎机不会给我们吐出大把钱来, 但它确实是一个不错的游戏。继续之前看一下图7-6, 了解一下将要构建的视图是什么样子的。

7.8.1 编写控制器头文件

将以下代码添加到CustomPickerViewController.h:

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

@interface CustomPickerViewController : UIViewController
    <UIPickerViewDataSource, UIPickerViewDelegate>
{
    IBOutlet    UIPickerView *picker;
    IBOutlet    UILabel *winLabel;

    NSArray *column1;
    NSArray *column2;
    NSArray *column3;
    NSArray *column4;
    NSArray *column5;
}
@property(n nonatomic, retain) UIPickerView *picker;
@property(n nonatomic, retain) UILabel *winLabel;
@property(n nonatomic, retain) NSArray *column1;
@property(n nonatomic, retain) NSArray *column2;
@property(n nonatomic, retain) NSArray *column3;
@property(n nonatomic, retain) NSArray *column4;
@property(n nonatomic, retain) NSArray *column5;
-(IBAction)spin;
@end
```

我们将声明两个输出口，一个用于选取器视图，另一个用于标签。该标签将用于在用户获胜之后告诉他们，也就是在同一行中得到3个相同的符号时。

我们还将创建5个指向NSArray对象的指针。这些指针用于保存图像视图，这些视图包含我们想要选取器绘制的图像。虽然所有5列都使用相同的图像，但我们仍然需要将每列对应的数组分开，使其拥有自己的图像视图集，因为每个视图只能在选取器中的一个位置绘制。我们也声明了一个操作方法，这次将其命名为spin。

7.8.2 构建视图

尽管图7-6中的图片比我们构建的视图更加漂亮，但设计nib的方式实际上没有太大的区别。所有其他工作都在控制器的委托方法中完成。

确保保存了新的源代码，然后双击CustomPickerView.xib，在Interface Builder中将其打开。添加一个标签和一个按钮。将按钮的标题命名为Spin。然后选定标签，在Fonts面板（按下⌘T）中调整标签文本的外观并将字体放大。可能还需要将标签本身放大，以适应新的大文本的大小。也可以使用属性检查器为标签分配一个漂亮的颜色。设置字体时，确保将文本设置为中间对齐。得到想要的文本之后，删除文字Label，因为我们不想在用户首次胜利之前显示任何文本。

然后，建立所有到输出口和操作的连接。需要将文件所有者的picker输出口连接到选取器视图，将文件所有者的winLabel输出口连接到标签，将按钮的Touch Up Inside事件连接到旋转操作。然后，确保指定了选取器的Delegate和DataSource。

最后，还有一件事情需要做。选择选取器并打开属性检查器。需要取消选中User Interaction Enabled的复选框，这样，用户就不能够手动更改刻度盘进行作弊了。完成之后，保存并返回Xcode。

注意 在Interface Builder中设计iPhone界面时，一定要小心使用Fonts面板。Interface Builder允许将Mac上的任何字体分配给标签，但是iPhone支持的字体类型很少。应该将字体选择限制为以下字体系列：American Typewriter、AppleGothic、Arial、Arial Rounded MT Bold、Arial Unicode MS、Courier、Courier New、DB LCD、Temp、Georgia、Helvetica、Helvetica Neue、Hiragino Kaku Gothic ProN W3、Hiragino Kaku Gothic ProN W6、Marker Felt、STHeiti J、STHeiti K、STHeiti SC、STHeiti TC、Times New Roman、Trebuchet MS、Verdana或者Zapfino。

7.8.3 添加图像资源

返回Xcode之后，需要添加将要在游戏中使用的图像。我们在项目归档文件的07 Pickers/Custom Picker Images文件夹中包含了6个图像文件（seven.png、bar.png、crown.png、cherry.png、lemon.png和apple.png）。将所有这些文件添加到项目的Resources文件夹。在提示是否需要创建副本时，最好将这些文件复制到项目文件夹中。

7.8.4 实现控制器

在这个控制器的实现过程中，我们添加了许多新内容。将以下代码添加到CustomPickerView-

Controller.m文件。然后，我们将依次讨论每部分新内容：

```
#import "CustomPickerViewController.h"

@implementation CustomPickerViewController
@synthesize picker;
@synthesize winLabel;
@synthesize column1;
@synthesize column2;
@synthesize column3;
@synthesize column4;
@synthesize column5;
- (id)initWithNibName:(NSString *)nibNameOrNil
  bundle:(NSBundle *)nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil
      bundle:nibBundleOrNil]) {
        // Initialization code
    }
    return self;
}
-(IBAction)spin
{
    BOOL win = NO;
    int numInRow = 1;
    int lastVal = -1;
    for (int i = 0; i < 5; i++)
    {
        int newValue = random() % [self.column1 count];

        if (newValue == lastVal)
            numInRow++;
        else
            numInRow = 1;

        lastVal = newValue;
        [picker selectRow:newValue inComponent:i animated:YES];
        [picker reloadComponent:i];
        if (numInRow >= 3)
            win = YES;
    }

    if (win)
        winLabel.text = @"WIN!";
    else
        winLabel.text = @"";
}

- (void)viewDidLoad {

    UIImage *seven = [UIImage imageNamed:@"seven.png"];
```

```

UIImage *bar = [UIImage imageNamed:@"bar.png"];
UIImage *crown = [UIImage imageNamed:@"crown.png"];
UIImage *cherry = [UIImage imageNamed:@"cherry.png"];
UIImage *lemon = [UIImage imageNamed:@"lemon.png"];
UIImage *apple = [UIImage imageNamed:@"apple.png"];

for (int i = 1; i <= 5; i++)
{
    UIImageView *sevenView = [[UIImageView alloc] initWithImage:seven];
    UIImageView *barView = [[UIImageView alloc] initWithImage:bar];
    UIImageView *crownView = [[UIImageView alloc] initWithImage:crown];
    UIImageView *cherryView = [[UIImageView alloc]
        initWithImage:cherry];
    UIImageView *lemonView = [[UIImageView alloc] initWithImage:lemon];
    UIImageView *appleView = [[UIImageView alloc] initWithImage:apple];

    NSArray *imageViewArray = [[NSArray alloc] initWithObjects:
        sevenView, barView, crownView, cherryView, lemonView,
        appleView, nil];
    NSString *fieldName =
        [[NSString alloc] initWithFormat:@"column%d", i];
    [self setValue:imageViewArray forKey:fieldName];
    [fieldName release];
    [imageViewArray release];

    [sevenView release];
    [barView release];
    [crownView release];
    [cherryView release];
    [lemonView release];
    [appleView release];
}

srandom(time(NULL));
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [picker release];
}

```

```

    [winLabel release];
    [column1 release];
    [column2 release];
    [column3 release];
    [column4 release];
    [column5 release];
    [super dealloc];
}
#pragma mark -
#pragma mark Picker Data Source Methods
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView
{
    return 5;
}

- (NSInteger)pickerView:(UIPickerView *)pickerView
    numberOfRowsInComponent:(NSInteger)component
{
    return [self.column1 count];
}
#pragma mark Picker Delegate Methods

- (UIView *)pickerView:(UIPickerView *)pickerView
    viewForRow:(NSInteger)row
    forComponent:(NSInteger)component reusingView:(UIView *)view
{
    NSString *arrayName = [[NSString alloc] initWithFormat:@"column%d",
        component+1];
    NSArray *array = [self valueForKey:arrayName];
    return [array objectAtIndex:row];
}
@end

```

这段代码中包含许多新内容。接下来逐一分析这些新方法。

7.8.5 spin 方法

spin方法将在用户触摸Spin按钮时被触发。在该方法中，我们首先声明了一些变量，这些变量有助于跟踪用户的胜负情况。使用win确定一行中的3个图像是否一样，如果是，则将win设置为YES。使用numInRow跟踪到目前为止，我们在一行中获得同一个值的次数，我们还将在lastVal中跟踪以前的组件的值，以便比较当前值与以前的值。将lastVal初始化为-1，因为我们知道，-1不会与任何真实的值匹配：

```

    BOOL win = NO;
    int numInRow = 1;
    int lastVal = -1;

```

接下来，通过循环将所有5个组件都设置为一个新的、随机生成的行选择。我们使用column1数组的计数来完成此工作，这是一种非常便捷的方法，因为我们知道，这5列都具有相同数量的值：


```
for (int i = 0; i < 5; i++)  
{  
    int newValue = random() % [self.column1 count];
```

将新值与上一个值进行比较，如果它们匹配，则将numInRow加1。如果它们不匹配，则将numInRow重置为1。然后将新值分配给lastVal，这样，可以在下一次循环中使用它来进行比较：

```
if (newValue == lastVal)  
    numInRow++;  
else  
    numInRow = 1;  
lastVal = newValue;
```

然后，将相应的组件设置为新值，告诉该组件制作更改动画，并告诉选取器重新载入该组件：

```
[picker selectRow:newValue inComponent:i animated:YES];  
[picker reloadComponent:i];
```

每次循环要做的最后一件事就是，查看是否在一行中得到了3个相同的图像，如果是，则将win设置为YES：

```
if (numInRow >= 3)  
    win = YES;  
}
```

完成循环之后，设置显示结果是胜还是负的标签：

```
if (win)  
    winLabel.text = @"Win!";  
else  
    winLabel.text = @"";
```

7.8.6 viewDidLoad 方法

新的viewDidLoad方法稍微有点复杂。但是不要担心，当我们将其分解之后，它就不再那么令人生畏了。我们做的第一件事情是载入6个不同的图像。UIImage类提供的imageName:便利方法可以轻松地完成此任务。

```
UIImage *seven = [UIImage imageNamed:@"seven.png"];  
UIImage *bar = [UIImage imageNamed:@"bar.png"];  
UIImage *crown = [UIImage imageNamed:@"crown.png"];  
UIImage *cherry = [UIImage imageNamed:@"cherry.png"];  
UIImage *lemon = [UIImage imageNamed:@"lemon.png"];  
UIImage *apple = [UIImage imageNamed:@"apple.png"];
```

前面已经提到过，在使用便捷的类方法初始化对象时要谨慎，因为它们使用了自动释放池。但是这里是一个例外，原因有两点。首先，这段代码仅在应用程序启动时触发；其次，它非常方便使用。通过使用此方法，无需在iPhone上确定每个图像的位置，然后使用该信息来载入每个图像。这可以节省数10行代码，而且不会增加太多的内存开销。

载入6个图像之后，需要创建一些UIImageView实例，分别对应每个图像以及5个选取器组件中的每一个。我们将通过一个循环来完成此任务：

```

for (int i = 1; i <= 5; i++)
{
    UIImageView *sevenView = [[UIImageView alloc] initWithImage:seven];
    UIImageView *barView = [[UIImageView alloc] initWithImage:bar];
    UIImageView *crownView = [[UIImageView alloc] initWithImage:crown];
    UIImageView *cherryView = [[UIImageView alloc]
        initWithImage:cherry];
    UIImageView *lemonView = [[UIImageView alloc] initWithImage:lemon];
    UIImageView *appleView = [[UIImageView alloc] initWithImage:apple];
}

```

创建了图像视图之后，将这些视图放到一个数组中。此数组将用于向选取器的每个组件提供数据。

```

NSArray *imageViewArray = [[NSArray alloc] initWithObjects:
    sevenView, barView, crownView, cherryView, lemonView,
    appleView, nil];

```

现在，只需将此数组分配给5个数组之一。为此，我们将创建一个字符串，该字符串与一个数组的名称匹配。第一次循环时，此字符串为column1，这是一个数组的名称，我们将使用该数组为选取器中第一个组件提供数据。第二次循环时，字符串将变为column2，依此类推：

```

NSString *fieldName = [[NSString alloc]
    initWithFormat:@"column%d", i];

```

为5个数组指定名称之后，可以使用一个非常方便的setValue:forKey:方法将此数组分配给该属性。此方法允许根据属性名称设置属性。因此，如果使用值“column1”调用此方法，其结果将与调用仿真器方法setColumn1:完全一样。

```

[self setValue:imageViewArray forKey:fieldName];

```

然后进行内存清理：

```

[fieldName release];
[imageViewArray release];

[sevenView release];
[barView release];
[crownView release];
[cherryView release];
[lemonView release];
[appleView release];
}

```

此方法的最后一项任务是提供一个随机数生成器。如果不提供随机数生成器，则每次游戏的结果都是一样的，这就失去了游戏的意义了。

```

srandom(time(NULL));
}

```

但是，向这5个数组填充了图像视图之后，我们还能够对它们做什么呢？如果向下浏览刚才输入的代码，将会发现，两个数据源方法与前面的版本几乎一样，但是如果继续往下查看委托方法，将会发现我们使用了一个完全不同的委托方法来向选取器提供数据。我们前面使用的委托方

法返回一个NSString *, 但是这个方法返回UIView *。

使用此方法, 我们可以为选取器提供任何能够在UIView中绘制的内容。当然, 由于选取器的尺寸较小, 所以既能够正常工作又比较美观的内容少之又少。但是此方法使我们在选择显示的内容上拥有更多自由, 只是需要做更多的工作。

```
- (UIView *)pickerView:(UIPickerView *)pickerView
    viewForRow:(NSInteger)row
    forComponent:(NSInteger)component
    reusingView:(UIView *)view
{
```

此方法返回5个数组中的某个图像视图。为此, 我们再次使用某个数组的名称创建一个NSString。由于component的索引为0, 所以我们将component加1, 这会提供一个介于column1与column5之间的值, 这个值与一个组件对应, 选取器将请求这个组件的数据。

```
    NSString *arrayName = [[NSString alloc] initWithFormat:@"column%d",
        component+1];
```

有了要使用的数组名称之后, 我们使用方法valueForKey:检索该数组。该方法相当于我们在viewDidLoad中使用的setValue:forKey:方法。使用valueForKey:方法相当于调用所指定的属性的访问方法。因此, 调用valueForKey:并指定“column1”的结果与使用column1访问方法相同。有了与组件对应的正确的数组之后, 返回该数组中与选定的行对应的图像视图。

```
    NSArray *array = [self valueForKey:arrayName];
    return [array objectAtIndex:index];
}
```

现在该放松一下了。我们一口气学习了viewDidLoad方法的所有内容, 接下来将在旋转中应用该方法。

7.8.7 最后的细节

这个小游戏非常有趣, 但它的构建方法却非常简单。接下需要对两个地方进行一些调整。现在, 还有两个地方不太令人满意。第一个就是它没有声音, 老虎机竟然如此安静! 第二个地方是, 刻度盘旋转还未结束, 游戏就告诉我们已经获胜了, 这虽然是个小问题, 但是它使游戏缺少了猜测的乐趣。

首先解决第一个问题, 本书附带的项目归档文件中的07 Pickers/Custom Picker Sounds文件夹包含两个声音文件: crunch.wav和win.wav。将这两个文件都添加到项目的Resources文件夹中。这两个声音分别在用户点击旋转按钮和获胜时播放。

要添加以上声音, 我们需要访问iPhone的Audio Toolbox类。在CustomPickerViewController.m的顶部插入下面这行代码:

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

接下来, 我们需要添加一个输出口, 该输出口将指向旋转按钮。当滚轮旋转时, 我们将隐藏该按钮。我们不希望在当前的旋转过程完成之前让用户再次点击该按钮。将以下代码添加到

CustomPickerviewController.h:

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

@interface CustomPickerViewController : UIViewController
    <UIPickerViewDataSource, UIPickerViewDelegate> {
    IBOutlet    UIPickerView *picker;
    IBOutlet    UILabel *winLabel;
    IBOutlet    UIButton *button;

    NSArray *column1;
    NSArray *column2;
    NSArray *column3;
    NSArray *column4;
    NSArray *column5;
}
-(IBAction)spin;
@property(n nonatomic, retain) UIPickerView *picker;
@property(n nonatomic, retain) NSArray *column1;
@property(n nonatomic, retain) NSArray *column2;
@property(n nonatomic, retain) NSArray *column3;
@property(n nonatomic, retain) NSArray *column4;
@property(n nonatomic, retain) NSArray *column5;
@property(n nonatomic, retain) UILabel *winLabel;
@property(n nonatomic, retain) UIButton *button;
@end
```

输入代码之后，保存nib，双击CustomPickerView.xib，在Interface Builder中打开该文件。打开该文件之后，按住Control键并将File's Owner拖到Spin按钮，并将其连接到我们刚才创建的新按钮输出口。保存nib文件并返回Xcode。

现在，在实现控制器类的过程中，我们需要做一些事情。首先，需要将访问方法和新输出口的修改方法结合起来，因此添加以下代码：

```
@implementation CustomPickerViewController
@synthesize picker;
@synthesize column1;
@synthesize column2;
@synthesize column3;
@synthesize column4;
@synthesize column5;
@synthesize winLabel;
@synthesize button;
...
```

我们还需要向控制器类添加两个方法。将以下两个方法添加到CustomPickerViewController.m，作为该类的前两个方法：

```
-(void)showButton
{
    button.hidden = NO;
```

```

}
-(void)playWinSound
{
    NSString *path = [[NSBundle mainBundle] pathForResource:@"win"
        ofType:@"wav"];
    SystemSoundID soundID;
    AudioServicesCreateSystemSoundID((CFURLRef)[NSURL URLWithString:path]
        , &soundID);
    AudioServicesPlaySystemSound (soundID);
    winLabel.text = @"WIN!";
    [self performSelector:@selector(showButton) withObject:nil
        afterDelay:1.5];
}

```

第一个方法用于显示旋转按钮。我们将在用户点击该按钮之后将其隐藏，因为如果滚轮仍在旋转，则不可能让它们在停止之前继续旋转。

第二个方法将在用户获胜之后被调用。此方法的第一行向主束请求声音win.wav的路径，就像在载入Dependent选取器视图的属性列表时的操作一样。获取该资源的路径之后，后面的3行代码将载入声音文件并播放它。然后，我们将标签设置为WIN!，并调用showButton方法，但是我们使用一个名为performSelector:withObject:afterDelay:的方法，通过一种特殊方式调用显示按钮方法。所有对象都可以非常方便地使用此方法，它允许在未来某个时候调用该方法，在本例中，将在1.5秒之后调用该方法，这会使游戏在刻度盘旋转到最终位置之后才告诉用户结果。

我们还需要对spin:方法进行一些更改。需要编写代码来播放声音，并在游戏者获胜之后调用playerwon方法。现在对代码进行以下更改：

```

-(IBAction)spin
{
    BOOL win = NO;
    int numInRow = 1;
    int lastVal = -1;
    for (int i = 0; i < 5; i++)
    {
        int newValue = random() % [self.column1 count];

        if (newValue == lastVal)
            numInRow++;
        else
            numInRow = 1;

        lastVal = newValue;
        [picker selectRow:newValue inComponent:i animated:YES];
        [picker reloadComponent:i];
        if (numInRow >= 3)
            win = YES;
    }
}

```

```

button.hidden = YES;
NSString *path = [[NSBundle mainBundle] pathForResource:@"crunch"
ofType:@"wav"];
SystemSoundID soundID;
AudioServicesCreateSystemSoundID((CFURLRef)[NSURL URLWithString:path]
, &soundID);
AudioServicesPlaySystemSound (soundID);

if (win)
    [self performSelector:@selector(playWinSound)
withObject:nil
afterDelay:.5];
else
    [self performSelector:@selector(showButton)
withObject:nil
afterDelay:.5];

winLabel.text = @"";
}

```

我们添加的第一行代码用于隐藏Spin!按钮。接下来的4行代码用于播放声音，让游戏者知道他们已经旋转了滚轮。然后，我们并不是在知道用户胜利时将标签设置为WIN!，而是采用了另外一种技巧。我们调用刚才创建的两个方法之一，但是在使用performSelector:afterDelay:进行延迟之后才调用。如果用户获胜了，程序则在0.5秒之后调用playerwon方法，这提供了足够的时间让刻度盘旋转到终点；如果用户失败了，程序将等待0.5秒，然后重新启用Spin!按钮。

最后需要做的就是确保释放了按钮输出口，因此对dealloc方法进行以下更改：

```

- (void)dealloc {
    [picker release];
    [winLabel release];
    [column1 release];
    [column2 release];
    [column3 release];
    [column4 release];
    [column5 release];
    [button release];
    [super dealloc];
}

```

7.8.8 链接 Audio Toolbox 框架

如果现在尝试编译应用程序，将会得到另一个链接错误。事实表明，这个错误与用于载入和播放声音的函数有关。是的，它们未包含在任何默认链接的框架中。按住Command并双击AudioServicesCreateSystemSoundID函数，这将转到声明该函数的头文件。在该文件中，我们可以看到此函数是Audio Toolbox框架的一部分。

从Project菜单中选择Add to Project...，然后导航到iPhone仿真器的框架文件夹，位于/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator2.1.sdk/System/Libr-

ary/Frameworks/中，并将AudioToolbox.framework添加到项目中，确保没有将该框架复制到项目中，并选择Relative to Current SDK。完成之后，应用程序应该能够正常编译了，进行游戏时也应该会有声音了。

7.9 小结

现在，你应该已经掌握了标签栏应用程序和选取器的基础知识。在本章中，我们从头构建了一个完善的标签栏应用程序，它包含5个不同的内容视图。你学习了如何在各种不同配置下使用选取器，如何创建带有多个组件的选取器，以及如何使某个组件中的值依赖于另一个组件中的选定值。最后还学习了如何让选取器显示图像，而不仅仅是文本。

本章还介绍了选取器委托和数据源，介绍了如何载入图像、载入声音以及通过属性列表创建字典。恭喜你掌握了本章的知识！如果已经准备好学习表视图，那么请继续学习下一章。



下一章将构建一个基于分层导航的应用程序，它类似于iPhone随带的电子邮件应用程序。通过这个应用程序，用户可以访问数据嵌套列表和编辑数据。不过，在此之前，需要先掌握表视图的基本概念。这正是本章将要介绍的内容。

表视图是用于向用户显示数据的一种最常见的机制。它们是高度可配置的对象，可以被配置为用户所需的任何形式。电子邮件使用表视图显示账户、文件夹和消息的列表，但是表视图并不仅限于显示文本数据。还可以在YouTube、Settings和iPod应用程序中使用表视图，尽管这些应用程序具有十分不同的外观（参见图8-1）。



图8-1 虽然看起来各自不同，但Settings、iPod和YouTube应用程序都使用表视图来显示数据

8.1 表视图基础

表用于显示数据列表。数据列表中的每项都由行表示。iPhone表没有限制行的数量，其数量仅受可用存储空间的限制。iPhone表可以只有一列。

表视图是显示表数据的视图对象，它是UITableViewController类的一个实例。表中的每个可见行都由UITableViewCell类实现。因此，表视图是显示表中可见部分的对象，表视图单元负责显示表中的一行（参见图8-2）。



图8-2 每个表视图都是UITableView的一个实例，每个可见行都是UITableViewCell的一个实例

表视图并不负责存储表中的数据。它们只存储足够绘制当前可见行的数据。表视图从遵循UITableViewDelegate协议的对象获取配置数据，从遵循UITableViewDataSource协议的对象获得行数据。本章稍后介绍的示例应用程序将展示如何实现这些操作。

如前所述，所有的表都只有1列。但是，图8-1右边所示的YouTube应用程序外观确实至少拥有2列，如果数一数图标，甚至会发现原来有3列。不过情况并非如此。表中的每一行都由一个UITableViewCell表示，可以使用一个图像、一些文本和一个可选的辅助图标来配置每个UITableViewCell对象。其中辅助图标是指位于右边的一个小图标，下一章将对它进行详细的介绍。

如果需要的话，可以在一个单元中放置更多的数据。可以通过两种基本方法来完成此操作。一种方法是向UITableViewCell添加子视图；另一种方法是通过子类化UITableViewCell。你可以按照任何喜欢的方式展示表视图单元，也可以添加任何想要的子视图。这样看来，单列限制并不像开始听起来那样可怕。如果这些让你感到迷惑，别担心，本章稍后将介绍这方面的技巧。

分组表和索引表

表视图分为两种基本样式。第一种是**分组表**。分组表中的每个组都由嵌入在圆角矩形中的多个行组成，如图8-3最左边的图片所示。注意，一个分组表可以只包含一个组。

另一种类型是**索引表**（在某些地方又称为**无格式表**）。索引表是默认的样式。任何没有圆角矩形属性的表都是索引表视图。

如果数据源提供了必要的信息，通过表视图，用户可以使用右侧的索引来导航列表。图8-3显示了一个分组表、一个不带索引的索引表（无格式表）和一个带有索引的索引表。



图8-3 相同的表视图分别显示为分组表（左）、不带索引的索引表（通常指无格式表，中间）、和一个带有索引的索引表（右）

表中的每个部分被称为数据源中的分区（section）。在分组表中，每个分组都是一个分区（参见图8-4）。在索引表中，数据的每个索引分组都是一个分区。例如，在图8-3所示的索引表中，以“A”开头的所有名称都是一个分区，以“B”开头的那些名称则是另一个分区，依此类推。



图8-4 分组表中的分区和行显而易见，不过所有的表都支持分区和行

分区主要有两个作用。在分组表中，每个分区表示一个组。在索引表中，每个分区对应一个索引条目。因此，如果你希望显示一个按字母顺序列出索引且每个字母作为一个索引条目的列表，那么你将拥有26个分区，每个分区包含以特定字母开头的值。

注意 从技术上来说，可以创建带有索引的分组表。即便如此，也不应该为分组表视图提供索引。iPhone Human Interface Guideline (iPhone人性化界面指南) 这本书中明确指出分组表不应该提供索引。

在本章中，我们将创建这两种类型的表。

8.2 实现一个简单的表

下面通过一个最简单的示例来了解表视图的工作原理。本示例将显示一个文本值列表。

在Xcode中创建一个新项目。对于本章来说，我们将使用基于视图的应用程序模板。选择这一项，然后将项目命名为Simple Table。

8.2.1 设计视图

展开Resources和Classes文件夹。这是一个极为简单的应用程序，它不需要任何输出口或操作，双击Simple_TableViewController.xib，在Interface Builder中打开该文件。View窗口应该已经打开，因此，只需要在库中找到Table View（参见图8-5），并将它拖到View窗口中即可。

表视图会自动将其高度和宽度调整为View窗口的高度和宽度（参见图8-6）。这正是我们所希望的。将表视图设计为占据屏幕的整个宽度，以及除应用程序导航栏、工具栏或标签栏之外的高度。

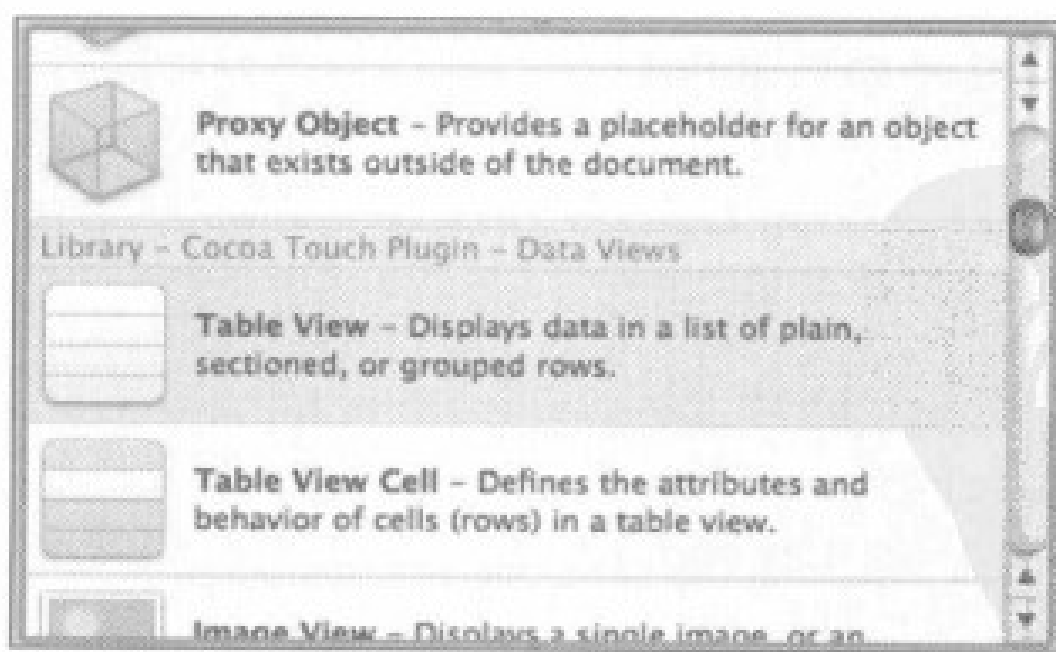


图8-5 库中的表视图

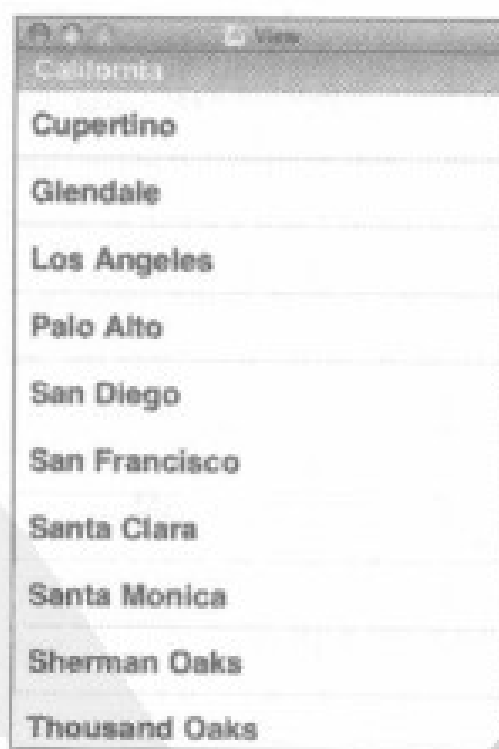


图8-6 添加表视图之后的View窗口

将表视图拖到View窗口上之后，它应该仍然处于选中状态，如果不是，那么通过单击操作来选定它，然后按下⌘2打开连接检查器。你会注意到，表视图的前两个可用连接和选取器视图的前两个连接一样，都是数据源和委托。从每个连接旁边的圆圈拖到File's Owner图标。这样一来，控制器类就成为此表的数据源和委托。完成上述操作之后，保存并关闭，然后返回到Xcode。

8.2.2 编写控制器

下面是控制器类的头文件。单击Simple_TableViewController.h，并添加以下代码：

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

@interface Simple_TableViewController : UIViewController
    <UITableViewDelegate, UITableViewDataSource>        {
    NSArray *listData;
}
@property (nonatomic, retain) NSArray *listData;
@end
```

上述代码的作用是让类遵从两个协议，类需要使用这两个协议来充当表视图的委托和数据源，然后声明一个数组用于放置将要显示的数据。

现在切换到Simple_TableViewController.m，添加更多的代码：

```
#import "Simple_TableViewController.h"

@implementation Simple_TableViewController
@synthesize listData;
#pragma mark Table View Controller Methods
- (void)viewDidLoad {
    NSArray *array = [[NSArray alloc] initWithObjects:@"Sleepy", @"Sneezy",
        @"Bashful", @"Happy", @"Doc", @"Grumpy", @"Dopey", @"Thorin",
        @"Dorin", @"Nori", @"Ori", @"Balin", @"Dwalin", @"Fili", @"Kili",
        @"Oin", @"Gloin", @"Bifur", @"Bofur", @"Bombur", nil];
    self.listData = array;
    [array release];
    [super viewDidLoad];
}
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}
- (void)dealloc {
    [listData release];
    [super dealloc];
}
#pragma mark -
#pragma mark Table View Data Source Methods
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
```

```

{
    return [self.listData count];
}
- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *SimpleTableIdentifier = @"SimpleTableIdentifier";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        SimpleTableIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
            reuseIdentifier: SimpleTableIdentifier] autorelease];
    }

    NSUInteger row = [indexPath row];
    cell.text = [listData objectAtIndex:row];
    return cell;
}
@end

```

我们为控制器添加了3个方法。你可能对第一个方法viewDidLoad感到很熟悉，因为我们前面使用过类似的方法。此方法只创建了一个要传递给表的数据的数组。在实际应用程序中，此数组很可能来自于另一个源，比如文本文件、属性列表或URL。

继续往下看，你会看到我们添加了两个数据源方法。第一个方法是tableView:numberOfRowsInSection:，表使用它来查看指定分区中有多少行。正如你所希望的，默认的分区数量为1，此方法用于返回组成列表的表分区中的行数。只需返回数组中数组项的数量即可。

下一个方法可能需要一些解释，让我们更仔细地看一下此方法：

```

- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath
{

```

当表视图需要绘制其中一行时，则会调用此方法。你会注意到此方法的第二个参数是一个NSIndexPath实例。表视图正是使用此机制把分区和行绑定到一个对象中的。要从NSIndexPath中获得一行或一个分区，只需要调用行方法或分区方法就可以了，这两个方法都返回一个int值。

第一个参数tableView是对发起请求的表的引用。通过它，我们可以创建充当多个表的数据源的类。

下面，声明一个静态字符串实例。

```
static NSString *SimpleTableIdentifier = @"SimpleTableIdentifier";
```

此字符串充当表示某种表单元的键。在此表中，我们将只使用一种单元，因此定义一种标识符就可以了。表视图在iPhone的小屏幕上一次只能显示几行，但是表自身能够保存相当多的数据。记住，表中的每一行都由一个UITableViewCell实例表示，该实例是UIView的一个子类，这就意味着每一行都能拥有子视图。对于大型表来说，如果视图为表中的每一行都分配一个表视图单元，

不管该行当前是否正被显示，这都将带来大量开销。幸好表并不是这样工作的。

相反，因滚动操作离开屏幕的一些表视图单元，将被放置在一个可以被重用的单元序列中。如果系统运行比较慢，表视图就从序列中删除这些单元，以释放存储空间，不过，只要有可用的存储空间，表视图就会重新获取这些单元，以便以后再次使用它们。

当一个表视图单元滚出屏幕时，另一个表视图单元就会从另一边滚动到屏幕上。如果滚动到屏幕上的新行重新使用从屏幕上滚动下来的其中一个单元，系统就会避免与不断创建和释放那些视图相关的开销。要充分利用此机制，我们需要让表视图给定一个出列单元，该出列单元正是我们需要的类型。注意，我们现在正在使用前面声明的NSString标识符。实际上，我们需要一个SimpleTableIdentifier类型的可重用单元：

```
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
    SimpleTableIdentifier];
```

现在，表视图中可能没有任何多余的单元了，我们来检查这些cell，看一下它是否为零(nil)。如果是，则使用上面提到的标识符字符串手动创建一个新的表视图单元。从某种程度上来说，我们将不可避免地重复使用此处创建的单元，因此需要确保它具有相同的类型。

```
if (cell == nil) {
    cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
        reuseIdentifier: SimpleTableIdentifier] autorelease];
}
```

现在，我们拥有了一个可以返回到表视图的表视图单元。下面所有要做的就是将需要显示的信息放在该表视图单元中。在表的一行内显示文本是很常见的任务，因此表视图单元提供了一个名称为text的属性，我们可以设置此属性以显示字符串。对于这种情况，我们需要从listData数组中获取正确的字符串，然后使用它设置表视图单元的text属性。

要完成上述操作，需要知道表视图需要显示哪些行。可以从indexPath变量获取该信息，如下所示：

```
NSInteger row = [indexPath row];
```

我们使用这个值从数组获取正确的字符串，将它分配给单元的text属性，然后返回该单元。

```
cell.text = [listData objectAtIndex:row];
return cell;
}
```

情况并不那么糟糕，对吗？下面编译并运行应用程序。等一下，是不是有地方弄错了？出现了一个链接错误？你认为我们应该怎么做呢？

如果你被难住了，那么给你一个提示：我们使用了一个名为CGRectZero的常量，它是Core Graphics框架的一部分。默认情况下，Core Graphics框架没有连接到项目。如果你忘记如何将它连接进来的话，可以参见第5章，其中逐步地介绍了此过程。解决这个问题后，就可以成功完成编译了。运行应用程序，你将看到显示在表视图中的数组值（参见图8-7）。

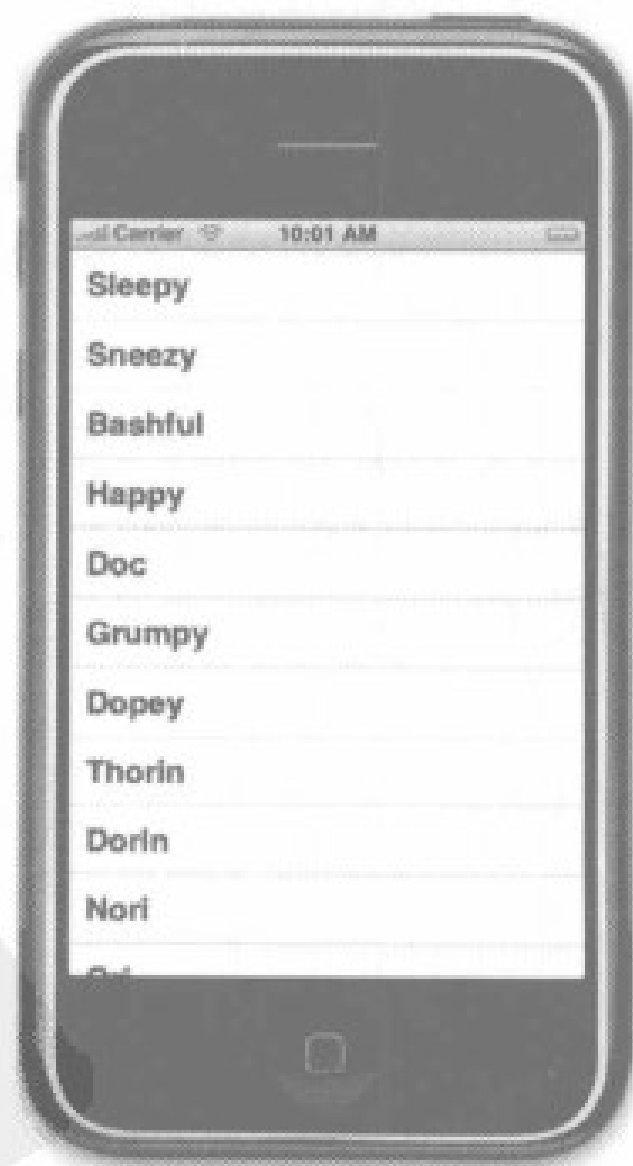


图8-7 简单的表应用程序

8.3 添加一个图像

要是可以向每一行添加一个图像就好了。我们需要创建一个UITableViewController子类来添加图像吗？不用。实际上，如果能够让图像位于每一行的左侧就不需要这么做了。默认的表视图单元会把这个情况处理好。下面我们来看一看。

在项目归档文件的08 Simple Table文件夹中，找到名为star.png的文件，然后把它添加到项目的Resources文件夹中。star.png是为此项目准备的一个小图标。

下面看看代码部分。在SimpleTableViewController.m文件中，在tableView:cellForRowAtIndexPath:方法中添加以下代码：

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *SimpleTableIdentifier = @" SimpleTableIdentifier ";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        SimpleTableIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
            reuseIdentifier: SimpleTableIdentifier] autorelease];
    }

    NSUInteger row = [indexPath row];
    cell.text = [listData objectAtIndex:row];
    UIImage *image = [UIImage imageNamed:@"star.png"];
    cell.image = image;
    return cell;
}
@end
```

好，这就完成了。刚才我们把单元的image属性设置为想要显示的任何图像。现在，如果编译并运行应用程序，出现的列表每一行的左侧都有一个漂亮的小星星图标（参见图8-8）。当然，如果愿意的话，可以为表中的每一行设置一个不同的图像。或者，费些工夫，为所有行分别应用不同的图标。

8.4 附加配置

你会注意到我们使用控制器作为此表视图的数据源和委托，不过到现在为止，还没有真正实现UITableViewDelegate的任何方法。与选取器视图不同，较简单的表视图不需要委托代替它们完成一些功能。数据源提供了绘制表所需要的所有数据。委托只是用于配置表视图的外观并处理某些用户交互。现在，让我们看一

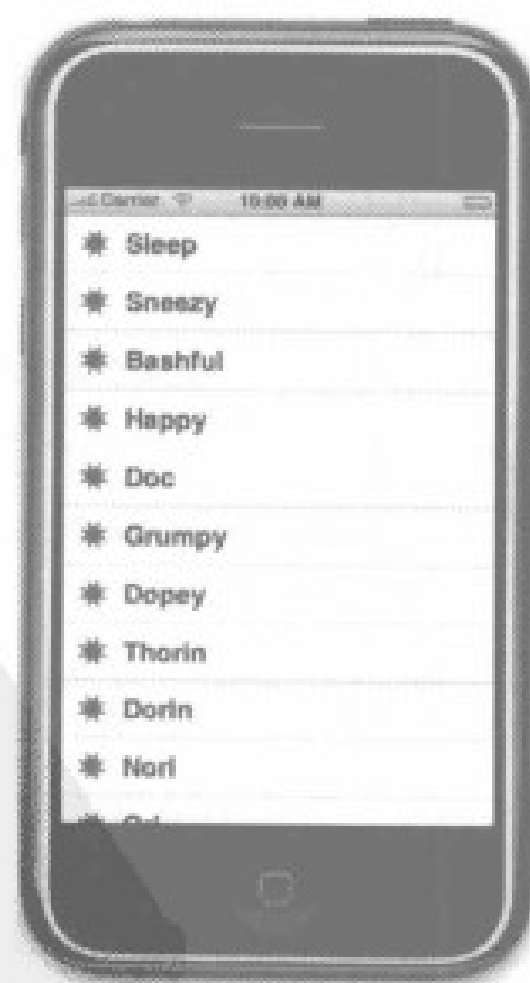


图8-8 使用单元的image属性，为每个表视图单元添加一个图像

下几个配置选项。下一章将更详细地介绍此内容。

8.4.1 设置缩进级别

可以使用委托指定缩进某些行。在Simple_TableViewController.m文件中，在代码中的@end declaration上方添加以下方法：

```
#pragma mark -
#pragma mark Table Delegate Methods

- (NSInteger)tableView:(UITableView *)tableView
  indentationLevelForRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSInteger row = [indexPath row];
    return row;
}
```

此方法把每一行的缩进级别设置为其行号，所以第0行的缩进级别为0，第1行的缩进级别为1，依此类推。缩进级别是一个整数，它会告诉表视图把一行向右移动一点。缩进级别的数量越大，行向右缩进得就越多。例如，可以使用这项技术来表示一行从属于另一行，就好像在电子邮件中表示子文件夹一样。

再次运行应用程序，可以看到每一行都在上一行的基础上向右移动了一些距离（参见图8-9）。

8.4.2 处理行的选择

表的委托可以使用两个方法确定用户是否选择了特定的行。一个方法在一行被突出显示之前调用，并且可以用于阻止选中此行，甚至改变被选中的行。让我们来实现这个方法，并指定第一行是不能被选中的。将以下方法添加到Simple_TableViewController.m的尾部，位于在@end声明之前：

```
-(NSIndexPath *)tableView:(UITableView *)tableView
  willSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSInteger row = [indexPath row];
    if (row == 0)
        return nil;

    return indexPath;
}
```

这个方法获取传递过来的indexPath，它用于表示哪项将被选中。我们的代码着眼于哪一行将被选中。如果这一行是第一行，其索引将始终为零，那么它将返回nil，表示实际上没有行被

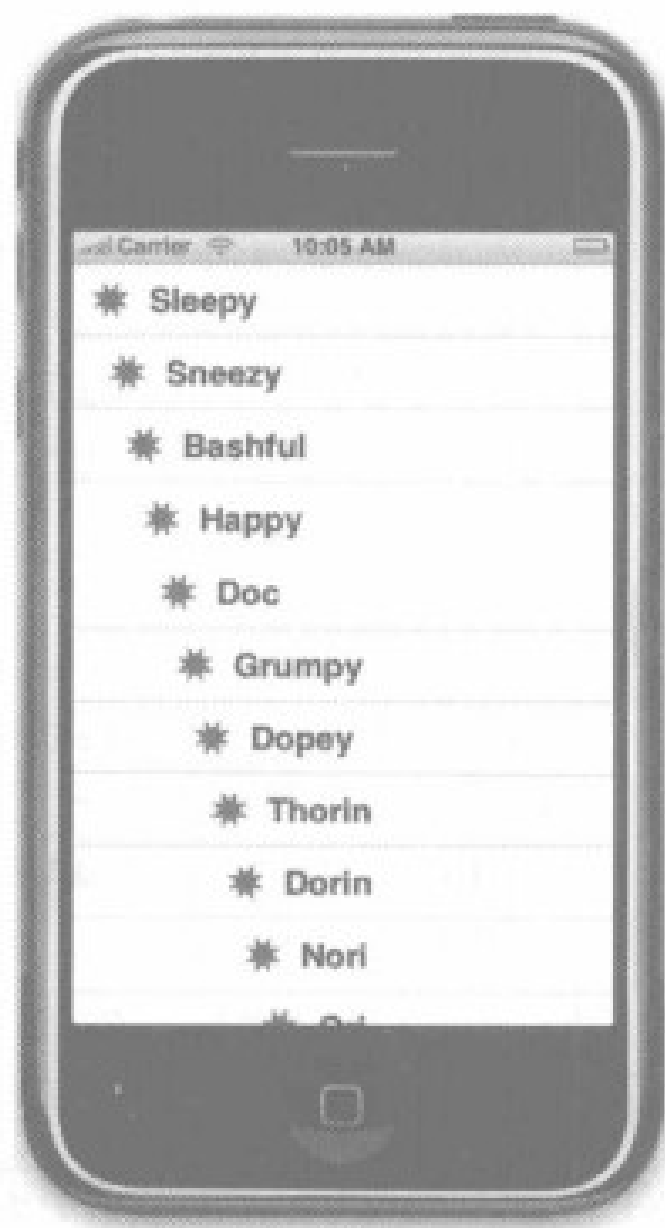


图8-9 表中的每一行都比上一行具有更高的缩进级别

选中。否则，它返回indexPath，表示选择可以继续进行。

在编译和运行应用程序之前，我们还要实现委托方法，在一行被选中之后调用该方法，通常它也是实际处理选择的地方。用户选中一行时，可以在这里做任何操作。在下一章中，我们将使用此方法处理更深入的问题。本章将只使用此方法抛出一个警告以显示某一行被选中了。将下面的方法添加到Simple_TableViewController.m的尾部，位于@end声明之前。

```
- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSUInteger row = [indexPath row];
    NSString *rowValue = [listData objectAtIndex:row];

    NSString *message = [[NSString alloc] initWithFormat:
        @"You selected %@", rowValue];
    UIAlertView *alert = [[UIAlertView alloc]
        initWithTitle:@"Row Selected!"
        message:message
        delegate:nil
        cancelButtonTitle:@"Yes I Did"
        otherButtonTitles:nil];
    [alert show];

    [message release];
    [alert release];
}
```

添加此方法之后，编译并运行应用程序。看一下你是否能够选中第一行（应该不能），然后选择其他行。被选中的行应该会突出显示，然后弹出警告通知你选择的是哪一行（参见图8-10）。

注意，你还可以在传递回indexPath之前修改索引路径，这将导致不同的行和/或分区被选中。你不会经常这样做，因为没有什么理由需要更改用户的选择。在大多数情况下，使用此方法时将返回indexPath或nil，以允许或禁止某个选择。

8.4.3 更改字体大小和行高

假设我们希望更改表视图中使用的字体大小。在大多数情况下，不应该覆盖默认的字体，那是用户所希望看到的。不过有时候我们有合适的理由这样做。在tableView:cellForRowAtIndexPath:方法中添加下面的代码行，然后编译并运行：



图8-10 在本示例中，第一行是不可选的。当选中其他任意行时会显示一个警告。此功能是使用委托方法完成的

```

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *SimpleTableIdentifier = @"SimpleTableIdentifier";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        SimpleTableIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
            reuseIdentifier:SimpleTableIdentifier] autorelease];
    }

    NSUInteger row = [indexPath row];
    cell.text = [listData objectAtIndex:row];
    cell.font = [UIFont boldSystemFontOfSize:80];
    UIImage *image = [UIImage imageNamed:@"star.png"];
    cell.image = image;
    return cell;
}

```

现在运行应用程序，列表中的值会变得很大，但是它们的大小并不适合行（参见图8-11）。

好，现在靠表视图委托来援救吧！表视图委托可以指定表中的行高。实际上，如果需要的话，它可以为每一行指定唯一值。下面向控制器类中添加此方法，代码位于@end之前。

```

#pragma mark -
#pragma mark Table View Delegate Methods
- (CGFloat)tableView:(UITableView *)tableView
heightForRowAtIndexPath:(NSIndexPath *)indexPath
{
    return 180;
}

```

在上面的代码中，我们使表视图把所有行高都设置为行180像素。编译并运行应用程序，现在表中的行应该高多了（参见图8-12）。



图8-11 大字体效果不错！不过如果我们能看见所有内容就更好了



图8-12 使用委托更改行大小。太大了吗？可能吧

8.4.4 委托还能做什么？

委托还能处理更多任务，下一章在介绍分层数据时会用到这些任务中的大多数。要了解更多内容，请使用文档浏览器查看UITextViewDelegate协议，然后看一下还有什么可用的其他方法。

8.5 定制表视图单元

你可以直接为表视图做许多事情，不过一般来说，你会希望以不受UITableViewCell直接支持的方式格式化每一行中的数据。对于这种情况，可以采用两种基本方法。一种方法是向UITableViewCell添加子视图，另一种方法是创建一个UITableViewCell子类。下面我们来看一下这两种方法。

8.5.1 单元应用程序

要展示如何使用自定义单元，我们将创建一个新的应用程序，使用另一个表视图，然后向用户显示两行信息（参见图8-13）。应用程序将显示一系列常见计算机模型的名称和颜色。通过向表视图单元添加子视图，我们将在同一个表单元中显示这两组信息。

8.5.2 向表视图单元添加子视图

默认的表视图单元只显示一行文本。即使你试图通过指定一个包含回车符的字符串让单元显示多个行，它也会删除回车符，并在下一个单元的行中显示数据。现在我们要创建一个项目，向单元添加子视图以摆脱这种束缚，这样就可以在每个单元中显示两行数据。

使用基于视图的应用程序模板创建一个新的Xcode项目，将它命名为Cells。双击CellsViewController.xib，添加一个TableView，然后像我们在上一章所做的，把委托和数据源设置为File's Owner。保存并回到Xcode。如果需要，可以参考8.6.1节的内容。

1. 修改控制器头文件

单击CellsViewController.h，添加以下代码：

```
#import <UIKit/UIKit.h>
#define kNameValueTag 1
#define kColorValueTag 2

@interface CellsViewController : UIViewController
    <UITableViewDataSource, UITableViewDelegate>
{
    NSArray *computers;
```



图8-13 向表视图单元添加子视图，可以让一个单元支持多行数据

```

}
@property (nonatomic, retain) NSArray *computers;
@end

```

你首先会注意到这里我们定义了两个常量。我们稍后将使用这两个常量为将要添加到表视图单元中的子视图分配标记 (tag)。下面将向一个单元中添加4个子视图, 其中2个需要在每一行进行更改。为此, 需要在使用特定的行数据更新单元时, 通过某种机制检索单元中的两个字段。如果为需要再次使用的每个标签设置唯一的标记值, 那么将能够从表视图单元检索它们并设置它们的值。

2. 实现控制器代码

在控制器中, 我们需要设置要用到的一些数据, 然后通过实现表数据源方法将这些数据反馈给表。单击CellsViewController.m, 然后添加以下代码:

```

#import "CellsViewController.h"

@implementation CellsViewController
@synthesize computers;

- (void)viewDidLoad {

    NSDictionary *row1 = [[NSDictionary alloc] initWithObjectsAndKeys:
        @"MacBook", @"Name", @"White", @"Color", nil];
    NSDictionary *row2 = [[NSDictionary alloc] initWithObjectsAndKeys:
        @"MacBook Pro", @"Name", @"Silver", @"Color", nil];
    NSDictionary *row3 = [[NSDictionary alloc] initWithObjectsAndKeys:
        @"iMac", @"Name", @"White", @"Color", nil];

    NSArray *array = [[NSArray alloc] initWithObjects:row1, row2,
        row3, nil];
    self.computers = array;

    [row1 release];
    [row2 release];
    [row3 release];
    [array release];

}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

```

```

- (void)dealloc {
    [computers release];
    [super dealloc];
}
#pragma mark -
#pragma mark Table Data Source Methods
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    return [self.computers count];
}
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellTableIdentifier = @"CellTableIdentifier ";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        CellTableIdentifier];
    if (cell == nil) {
        CGRect cellFrame = CGRectMake(0, 0, 300, 65);
        cell = [[[UITableViewCell alloc] initWithFrame: cellFrame
            reuseIdentifier: CellTableIdentifier] autorelease];

        CGRect nameLabelRect = CGRectMake(0, 5, 70, 15);
        UILabel *nameLabel = [[UILabel alloc] initWithFrame:nameLabelRect];
        nameLabel.textAlignment = UITextAlignmentRight;
        nameLabel.text = @"Name:";
        nameLabel.font = [UIFont boldSystemFontOfSize:12];
        [cell.contentView addSubview: nameLabel];
        [nameLabel release];

        CGRect colorLabelRect = CGRectMake(0, 26, 70, 15);
        UILabel *colorLabel = [[UILabel alloc] initWithFrame:
            colorLabelRect];
        colorLabel.textAlignment = UITextAlignmentRight;
        colorLabel.text = @"Color:";
        colorLabel.font = [UIFont boldSystemFontOfSize:12];
        [cell.contentView addSubview: colorLabel];
        [colorLabel release];

        CGRect nameValueRect = CGRectMake(80, 5, 200, 15);
        UILabel *nameValue = [[UILabel alloc] initWithFrame:
            nameValueRect];
        nameValue.tag = kNameValueTag;
        [cell.contentView addSubview:nameValue];
        [nameValue release];

        CGRect colorValueRect = CGRectMake(80, 25, 200, 15);
        UILabel *colorValue = [[UILabel alloc] initWithFrame:

```



```

        colorValueRect];
        colorValue.tag = kColorValueTag;
        [cell.contentView addSubview:colorValue];
        [colorValue release];
    }

    NSUInteger row = [indexPath row];
    NSDictionary *rowData = [self.computers objectAtIndex:row];
    UILabel *name = (UILabel *)[cell.contentView viewWithTag:
        kNameValueTag];
    name.text = [rowData objectForKey:@"Name"];

    UILabel *color = (UILabel *)[cell.contentView viewWithTag:
        kColorValueTag];
    color.text = [rowData objectForKey:@"Color"];
    return cell;
}
@end

```

在这里，viewDidLoad方法创建了一系列字典。每个字典都包含表中一行的名称和颜色信息。某一行中的名称在字典的Name键下，颜色在Color键下。我们把所有的字典放到了同一个数组里，这就是此表的数据。

让我们着重看一下tableView:cellForRowAtIndexPath:，此方法中真正添加了新内容。代码的前两行像前面介绍过的一样，先创建一个标识符，然后，如果表提供了出列的表视图单元，则要求表将该单元退出队列。

如果表没有任何可以重用的单元，就必须创建一个新的单元。创建新单元时，还需要创建和添加将要用到的子视图，以实现每单元两行的表。让我们更仔细地研究一下代码。首先，创建一个单元，其实这跟前面一样，除非不使用表视图进行计算，而是手动指定单元的大小。

```

CGRect cellFrame = CGRectMake(0, 0, 300, 65);
cell = [[[UITableViewCell alloc] initWithFrame: cellFrame
    reuseIdentifier: CellTableIdentifier] autorelease];

```

接下来需要创建4个UILabel，并把它们添加到表视图单元。表视图单元已经有了一个名为contentView的UIView子视图，用于对它的所有子视图进行分组，就像第4章中我们对UIView中的两个开关进行分组一样。因此，我们不用把标签作为子视图直接添加到表视图单元，而是添加到contentView。

```

[cell.contentView addSubview:colorValue];

```

其中两个标签包含静态文本。NameLabel标签包含Name:文本，colorLabel标签包含Color:文本。这些是不需要更改的静态标签。另两个标签用于显示指定行的数据。记住，稍后我们需要检索这些字段，所以要为这两个字段分配值。例如，把常量kNameValueTag分配给nameValue的tag字段：

```

nameValue.tag = kNameValueTag;

```

稍后，我们将使用该标记从单元中检索正确的标签。

创建新单元之后，使用传入的indexPath参数确定表正在请求单元的哪一行，然后使用该行的值为请求的行获取正确的字典。记住，该字典有两个键/值对，一个是名称，一个是颜色。

```
NSUInteger row = [indexPath row];
NSDictionary *rowData = [self.computers objectAtIndex:row];
```

还记得我们前面设置的标记吗？在这里，我们使用那些标记检索需要设置值的标签。

```
UILabel *name = (UILabel *)[cell.contentView viewWithTag:kNameValueTag];
```

检索到标签以后，只需将标签文本设置为从表示此行的字典里获取的一个值。

```
name.text = [rowData objectForKey:@"Name"];
```

编译并运行应用程序，你会得到带有两行数据的行，如图8-13所示。向表视图添加视图比单独使用标准的表视图单元具有更大的灵活性，不过，通过编程创建、定位和添加所有的子视图是一项单调乏味的工作。如果我们能在Interface Builder中设计表视图单元就好了，不是吗？

8.5.3 使用 UITableViewCell 的自定义子类

我们很幸运，可以使用Interface Builder设计表视图单元。我们将使用Interface Builder重新创建与刚才使用代码构建的界面相同的两行界面。要达到此目的，可以创建一个UITableViewCell子类和一个包含表视图单元的新nib文件。然后，当我们需要一个表视图单元来表示一行时，不是向标准的表视图单元添加子视图，而是从nib文件加载子类，并使用将添加的两个输出口来设置名称和颜色。有道理吗？让我们开始付诸行动吧。

在Xcode中右键单击（或Ctrl+单击）Classes文件夹，从出现的Add子菜单中选择New File...，或者按下⌘N键。新建文件向导出现后，从左侧窗格选择Cocoa Touch Classes，然后从右上窗格选择UITableViewCell子类。单击Next按钮，将新文件命名为CustomCell.m，并确保选中了Also Create “CustomCell.h”复选框。

创建文件之后，在Xcode中右键单击Resources文件夹，再次选择Add→New File...。这一次，在新建文件向导的左侧窗格中单击User Interfaces，在右上窗格中选择Empty XIB。当提示输入名称时，输入CustomCell.xib。

1. 创建UITableViewCell子类

创建所有必要的新文件之后，下面让我们继续创建UITableViewCell的新子类。

我们将在子类中使用输出口，这会简化对需要在每一行更改的值的设置。我们可以再次使用标记（tag），这完全避免了创建子类，而使用这种方式，代码会更加简明并容易阅读，因为我们可以仅通过设置属性来设置每一行单元上的标签，如下所示：

```
cell.nameLabel = @"Foo";
```

单击CustomCell.h，添加以下代码：

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

```
@interface CustomCell : UITableViewCell {
    IBOutlet UILabel *nameLabel;
    IBOutlet UILabel *colorLabel;
}
@property (nonatomic, retain) UILabel *nameLabel;
@property (nonatomic, retain) UILabel *colorLabel;
@end
```

这就是我们需要添加的所有内容，下面切换到CustomCell.m，并添加两行代码：

```
#import "CustomCell.h"

@implementation CustomCell
@synthesize nameLabel;
@synthesize colorLabel;
- (id)initWithFrame:(CGRect)frame
    reuseIdentifier:(NSString *)reuseIdentifier {
    if (self = [super initWithFrame:frame
        reuseIdentifier:reuseIdentifier]) {
        // Initialization code
    }
    return self;
}
- (void)setSelected:(BOOL)selected animated:(BOOL)animated {

    [super setSelected:selected animated:animated];

    // Configure the view for the selected state
}

- (void)dealloc {
    [super dealloc];
}

@end
```

保存上面的代码，并准备好自定义子类。

2. 在Interface Builder中设计表视图单元

下一步，双击CustomCell.xib，在Interface Builder中打开文件。这个主窗口中只有两个图标：File's owner和First Responder。在库中找到表视图单元（参见图8-14），然后把它拖动到主窗口中。

确保选中了表视图单元，然后按下⌘4，打开标识检查器。将类从UITableViewCell改为CustomCell。

然后，按下⌘3，打开大小检查器，将表视图单元的高度从44改为65。这会让我们有更多的活动空间。

最后，按下⌘1，打开属性检查器。其中第一个字段是

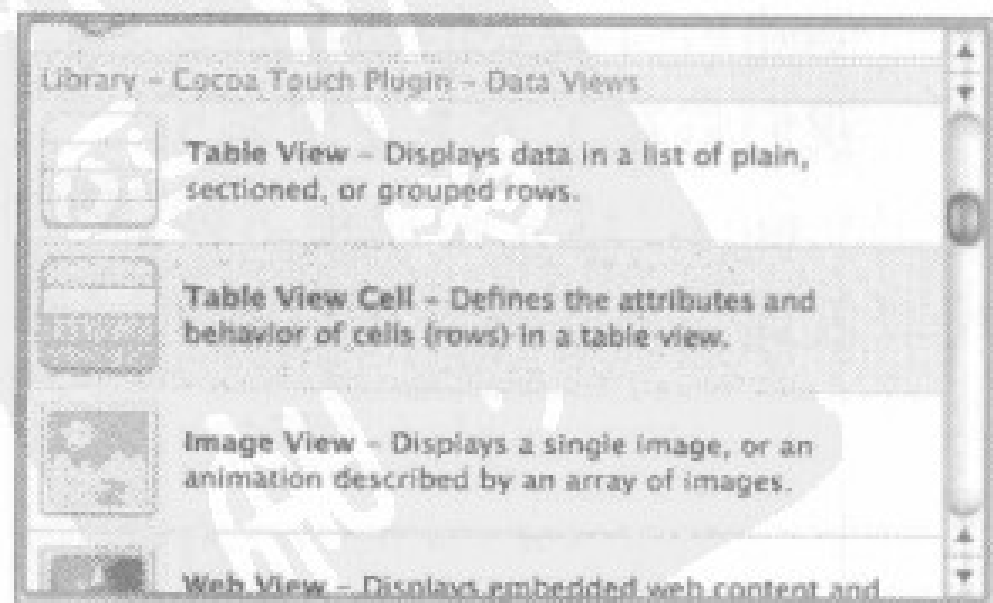


图8-14 库中的表视图单元

Identifier, 它是在代码中使用过的可重用的标识符。如果记不起来这一内容, 请查阅本章前面的内容并找到 SimpleTableIdentifier。将 Identifier 设置为 CustomCell-Identifier。下一步, 找到名称为 Accessory 的弹出按钮, 把 Detail Disclosure 改为 None (参见图 8-15)。扩展图标会在单元中占用一些空间, 但是我们希望整个单元空间都归自己所用。下一章将详细地讨论扩展图标。

记住, 即使 UITableViewCell 是 UIView 的子类, 它仍然使用内容视图对子视图进行保存和分组。双击 Custom Cell 图标, 将打开一个新的窗口, 你会发现一个标记为 Content View 的灰色虚线圆角矩形 (参见图 8-16)。

Interface Builder 通过这种方式告诉你应该添加一些内容, 因此在库中找到 View 并把它拖到 Custom Cell 窗口中。

发布视图之后会发现它的大小和窗口大小不一致, 还要进行调整。选中新的视图, 打开大小检查器。通过将 x 设置为 0, y 设置为 0, w 设置为 320, h 设置为 65, 把 View 的大小和位置改为符合 Custom Cell 窗口的大小。

现在所有项都设置完成了。我们有了一个画布, 可以使用它在 Interface Builder 中设计表视图单元。下面就开始吧。

从库中拖动 4 个标签到 Custom Cell 窗口, 按照图 8-17 所示进行布局 and 重命名。要将 Name: 和 Color: 设置为黑体, 选中它们并按下 **⌘B**。下一步, 选中右上方的标签, 增加其宽度, 将其右边缘拖到右边的蓝线。按同样方式更改右下位置的标签。这样做的目的是让名称和颜色数据拥有更大的显示空间。

现在, 按下 Control 键, 并将 Custom Cell 图标拖到视图右上位置的标签, 为它指定 nameLabel 输出口。然后, 按下 Control 键, 把 Custom Cell 图标再次拖到右下位置的标签, 为它指定 colorLabel 输出口。

你可能很奇怪为什么我们没有做任何和 File's Owner 图标有关的事情。原因是根本不需要。我们使用这个表单元显示数据, 不过与用户的所有交互都是通过表视图来完成的, 因此它不需要自己的控制器类。我们实际上只是使用 nib 作为一种模板, 以便可视地设计表单元。

保存 nib 并将其关闭, 然后返回 Xcode。

3. 使用新的表视图单元

要使用我们设计的单元, 必须对 CellsViewController.m 中的 tableView:cellForRowAtIndexPath 方法做一些大的改动。删除当前的 tableView:cellForRowAtIndexPath 方法, 用下面

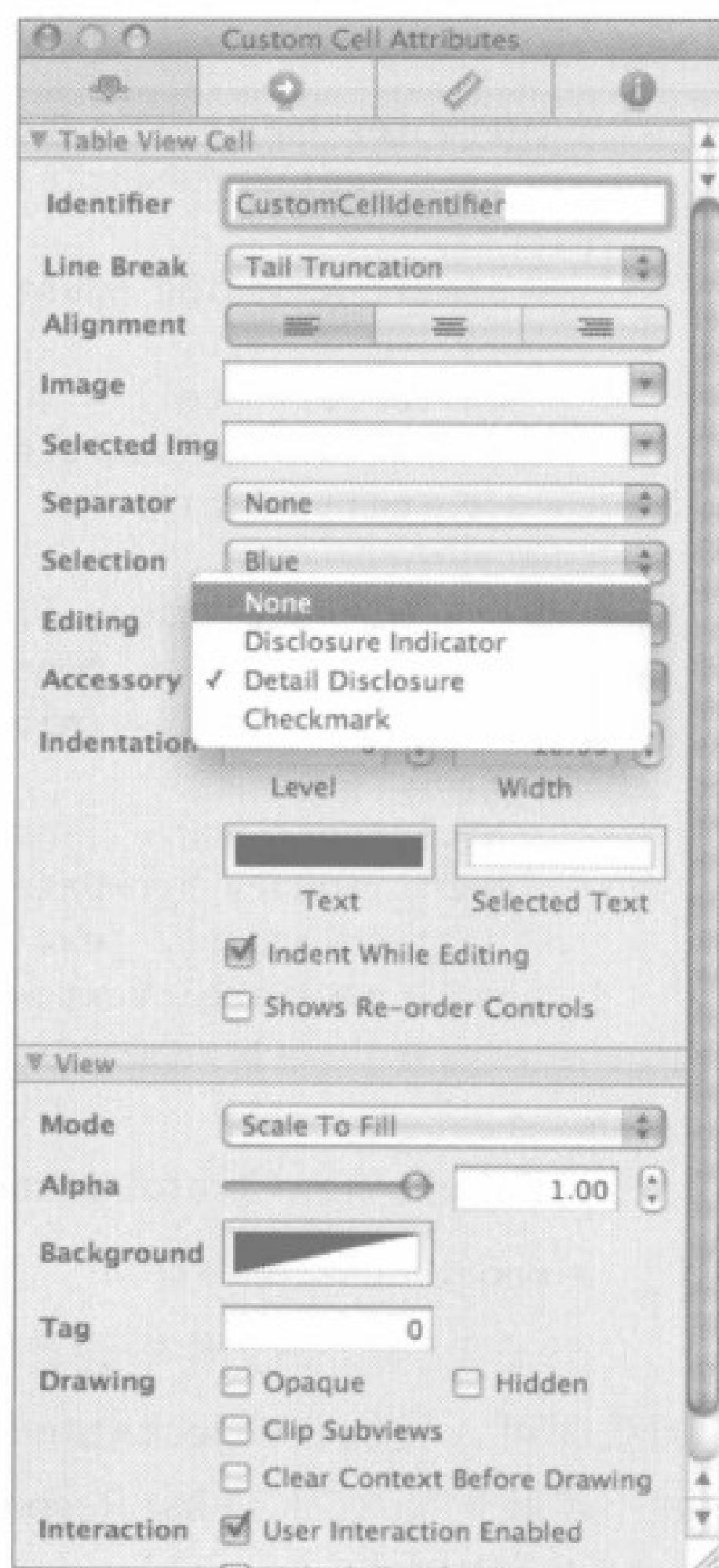


图 8-15 关闭扩展图标

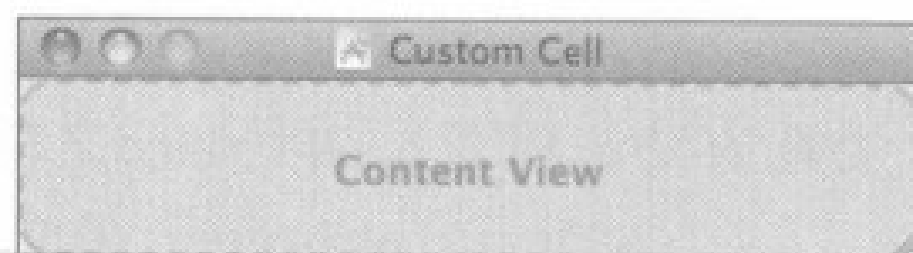


图 8-16 表视图单元的窗口



图 8-17 表视图单元的设计

的新版本代替：

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CustomCellIdentifier = @"CustomCellIdentifier ";

    CustomCell *cell = (CustomCell *)[tableView
        dequeueReusableCellWithIdentifier: CustomCellIdentifier];
    if (cell == nil)
    {
        NSArray *nib = [[NSBundle mainBundle] loadNibNamed:@"CustomCell"
            owner:self options:nil];
        cell = [nib objectAtIndex:1];
    }
    NSInteger row = [indexPath row];
    NSDictionary *rowData = [self.computers objectAtIndex:row];
    cell.colorLabel.text = [rowData objectForKey:@"Color"];
    cell.nameLabel.text = [rowData objectForKey:@"Name"];
    return cell;
}
```

更改CellsViewController.m时，在接近顶部的位置添加此行：

```
#import "CustomCell.h"
```

由于我们在nib文件中设计了表视图单元，因此，如果没有可重用的单元，只需从nib文件中加载即可。我们在objectAtIndex:调用中使用索引值1而不是0，因为对象0是文件的所有者，它并不是我们想要的。First Responder不是由loadNibNamed:owner:options:返回的，因此表视图单元的索引值为1。

下面还要添加另一项内容。由于更改了表视图单元默认的高度值，我们必须告知表视图这一事实；否则，表视图单元不会留下足够的显示空间。为此，在CellsViewController.m中添加以下委托方法，将其添加到@end之前：

```
- (CGFloat)tableView:(UITableView *)tableView
    heightForRowAtIndexPath:(NSIndexPath *)indexPath
{
    return
}
```

然后，我们不能从单元获得这个值，因为此委托方法可能在单元存在之前被调用，因此我们必须硬编码这个值。把这个常量定义添加到CustomCell.h的顶部，然后删除那些不再需要的tag常量。

```
#define kTableViewRowHeight 66
#define kNameValueTag 1
#define kColorValueTag 2
```

这就好了。编译并运行程序。现在，两行的表单元都是基于Interface Builder设计技术的。

8.6 分组分区和索引分区

下一个项目将探讨表的另一个基本内容。仍然使用一个表视图（没有分层），不过我们将把数据分为几个分区。再次使用基于视图的应用程序模板创建一个新的Xcode项目，这一次将它命名为Sections。

8.6.1 构建视图

打开Classes和Resources文件夹，在Interface Builder中双击SectionsViewController.xib，打开文件。与之前一样，把表视图拖到View窗口中。然后按下⌘2，将数据源和委托连接到File's Owner图标。

下一步，确保选中表视图，按下⌘1，打开属性检查器。把表视图的Style从Indexed改为Grouped（参见图8-18）。这里可以提示一下，我们在本章开头讨论过索引类型和分组类型之间的差别。保存并返回Xcode。

8.6.2 导入数据

要完成此项目需要大量的数据。我们提供了另一个属性列表，为你节省几个小时敲键盘的时间。从本书随附的项目归档文件的08 Sections文件夹中找到名为sortednames.plist文件，把它添加到项目的Resource文件夹中。

完成添加以后，单击sortednames.plist，看一下它到底是什么（参见图8-19）。它是包含字典的一个属性列表，其中字母表中的每个字母都有一个条目。每个字母下面是以该字母开头的名称列表。

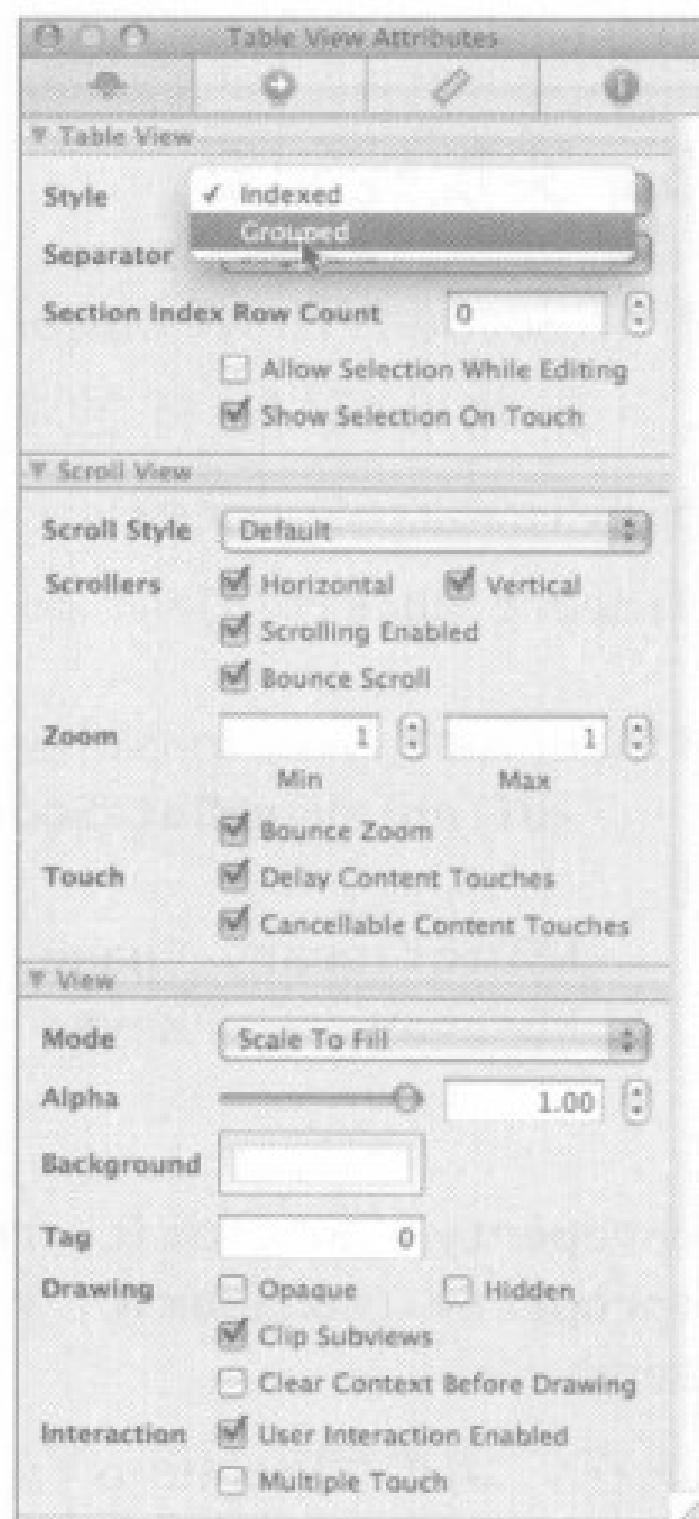


图8-18 表视图的属性检查器



图8-19 sortednames.plist属性列表文件

我们将使用这个属性列表中的数据填充表视图，并为每个字母创建一个分区。

8.6.3 实现控制器

单击SectionsViewController.h文件，添加NSDictionary和NSArray实例变量及其相应的属性声明。字典将保存所有数据。数组将保存以字母顺序排序的分区。还需要让类遵循UITableViewDataSource和UITableViewDelegate协议：

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

@interface SectionsViewController : UIViewController
    <UITableViewDataSource, UITableViewDelegate>
{
    NSDictionary *names;
    NSArray      *keys;
}
@property (nonatomic, retain) NSDictionary *names;
@property (nonatomic, retain) NSArray *keys;
@end
```

现在，切换到SectionsViewController.m，并添加以下代码：

```
#import "SectionsViewController.h"

@implementation SectionsViewController
@synthesize names;
@synthesize keys;
#pragma mark -
#pragma mark UIViewController Methods

- (void)viewDidLoad {
    NSString *path = [[NSBundle mainBundle] pathForResource:@"sortednames"
        ofType:@"plist"];
    NSDictionary *dict = [[NSDictionary alloc]
        initWithContentsOfFile:path];
    self.names = dict;
    [dict release];

    NSArray *array = [[names allKeys] sortedArrayUsingSelector:
        @selector(compare:)];
    self.keys = array;
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
```



```
- (void)didReceiveMemoryWarning {  
    [super didReceiveMemoryWarning];  
    // Releases the view if it doesn't have a superview  
    // Release anything that's not essential, such as cached data  
}  
  
- (void)dealloc {  
    [names release];  
    [keys release];  
    [super dealloc];  
}  
#pragma mark -  
#pragma mark Table View Data Source Methods  
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView  
{  
    return [keys count];  
}  
- (NSInteger)tableView:(UITableView *)tableView  
    numberOfRowsInSection:(NSInteger)section  
{  
    NSString *key = [keys objectAtIndex:section];  
    NSArray *nameSection = [names objectForKey:key];  
    return [nameSection count];  
}  
- (UITableViewCell *)tableView:(UITableView *)tableView  
    cellForRowAtIndexPath:(NSIndexPath *)indexPath  
{  
    NSUInteger section = [indexPath section];  
    NSUInteger row = [indexPath row];  
  
    NSString *key = [keys objectAtIndex:section];  
    NSArray *nameSection = [names objectForKey:key];  
  
    static NSString *SectionsTableIdentifier = @"SectionsTableIdentifier";  
  
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:  
        SectionsTableIdentifier];  
    if (cell == nil) {  
        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero  
            reuseIdentifier: SectionsTableIdentifier] autorelease];  
    }  
  
    cell.text = [nameSection objectAtIndex:row];  
    return cell;  
}  
- (NSString *)tableView:(UITableView *)tableView  
    titleForHeaderInSection:(NSInteger)section  
{
```

```

        NSString *key = [keys objectAtIndex:section];
        return key;
    }
@end

```

上面的大部分代码和我们以前看到的没有多大区别。在viewDidLoad方法中，从添加到项目的属性列表中创建了一个NSDictionary实例，并为它指定名称names。然后，获取字典中的所有键，按照字典中字母表的顺序对键值进行排序，将会得到一个有序的NSArray。记住，NSDictionary使用字母表中的字母作为它的键，因此这个数组将拥有26个字母，按照顺序从A到Z。我们将通过这个数组来了解分区。

下面看一下数据源方法。我们添加的第一个方法指定了分区的数量。上次没有实现此方法是因为默认设置1已经很合适了。这一次，我们需要告诉表视图字典中的每个键都有一个分区。

```

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return [keys count];
}

```

第二个方法用于计算特定分区中的行数。上一次，只有一个分区，所以只返回数组中拥有的行数。这次，我们将以分区为单位返回行数。可以检索对应于正被考虑的分区数组，并从该数组返回行的数量，从而达到目的。

```

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    NSString *key = [keys objectAtIndex:section];
    NSArray *nameSection = [names objectForKey:key];
    return [nameSection count];
}

```

在tableView:cellForRowAtIndexPath:方法中，必须从索引路径获取分区和行，用它们来确定要使用哪一个值。分区会告诉我们从名称字典中取出哪个数组，然后可以使用行指出要使用该数组中的哪个值。方法中的其他内容基本上和Simple Table应用程序代码中的相同。

可以通过tableView:titleForHeaderInSection方法为每个分区指定一个可选的标题值，然后只返回这一组的字母就可以了。

```

- (NSString *)tableView:(UITableView *)tableView
    titleForHeaderInSection:(NSInteger)section
{
    NSString *key = [keys objectAtIndex:section];
    return key;
}

```

最后，由于在SectionViewController.m中引用了CGRectZero，我们需要连接到Core Graphics框架。单击Groups & Files窗格中的Frameworks文件夹，从Project菜单中选择Add to Project。如果你忘记了如何导航到框架，请返回到第5章的末尾部分，该部分详细介绍了框架的位置。

现在可以编译并运行项目，然后慢慢地欣赏它了。记住，我们已经把表的样式改为Grouped

了，因此最终拥有一个带有26个分区的分组表，如图8-20所示。

作为比较，让我们把表视图再次改为索引风格，然后看一下带有多个分区的索引表视图是什么样子。在Interface Builder中双击SectionViewController.xib，打开该文件。选中表视图，使用属性检查器将视图改为Indexed。保存并返回Xcode，编译并运行，得到的是相同的数据和不一样的外观（参见图8-21）。

8.6.4 添加索引

当前表的一个问题是行数太多了。此列表中有2000个名称，要找到Zacharian或Zebedian，你的手指会非常累，更别说Zojirishu了。

这个问题的一个解决方案是，在表视图的右侧添加一个索引。既然我们已经把表视图样式改成了索引类型，要添加一个索引相对来说也很容易。在SectionsViewController.m文件尾部，@end前面添加如下方法：

```
- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)tableView
{
    return keys;
}
```

这就完成了。在这个方法中，委托请求一个值数组在索引中显示。必须使表视图中的多个分区使用索引，而且此数组中的条目必须对应于这些分区。返回的数组拥有的条目数必须与拥有的分区数相同，且值必须对应于适当的分区。也就是说，此数组中的第一项带给用户的是第一个分区，即分区0。

再次编译并运行，你将得到一个很好的索引（参见图8-22）。

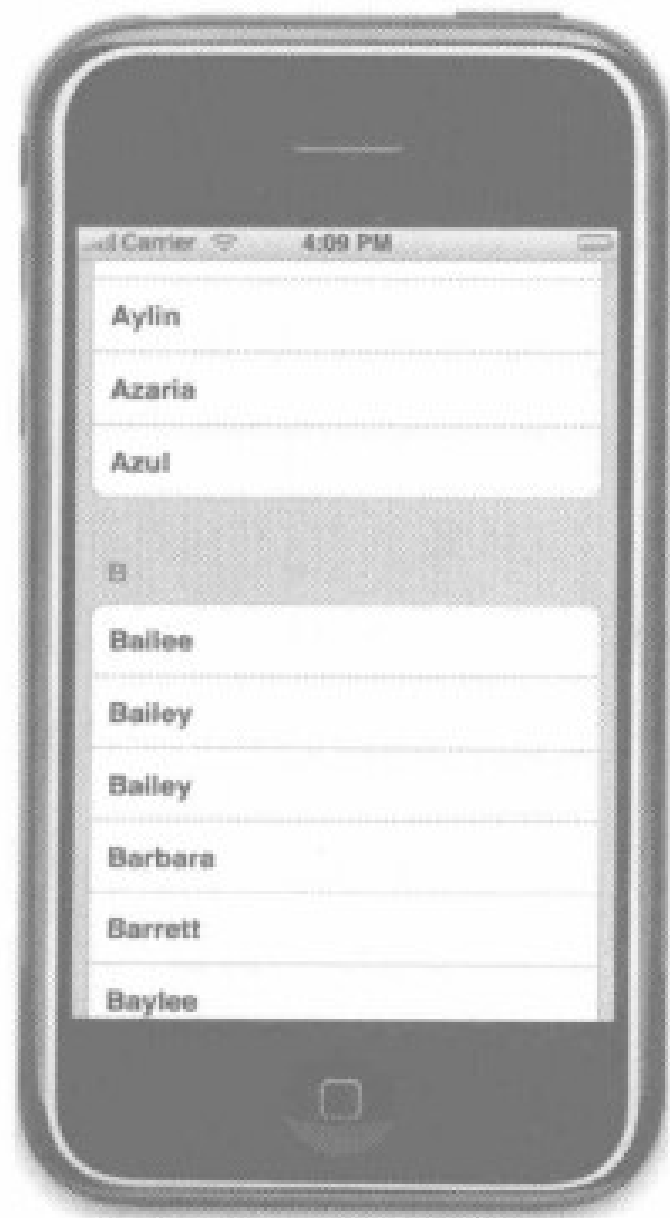


图8-20 有多个分区的分组表

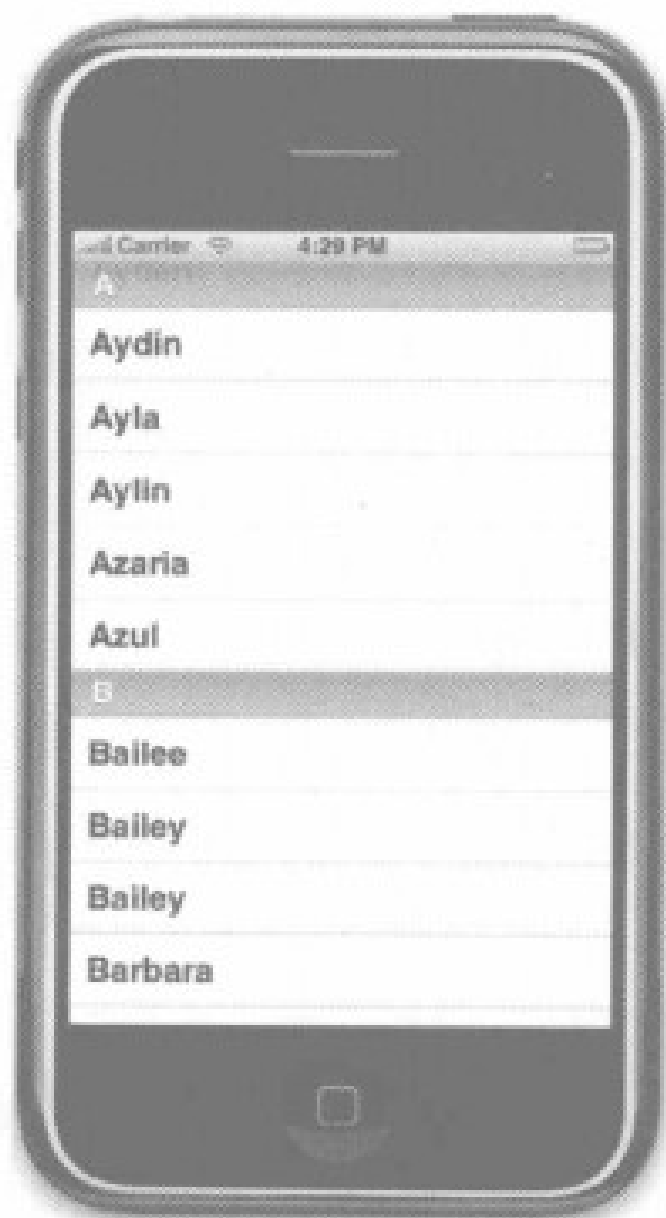


图8-21 带有分区的索引表视图

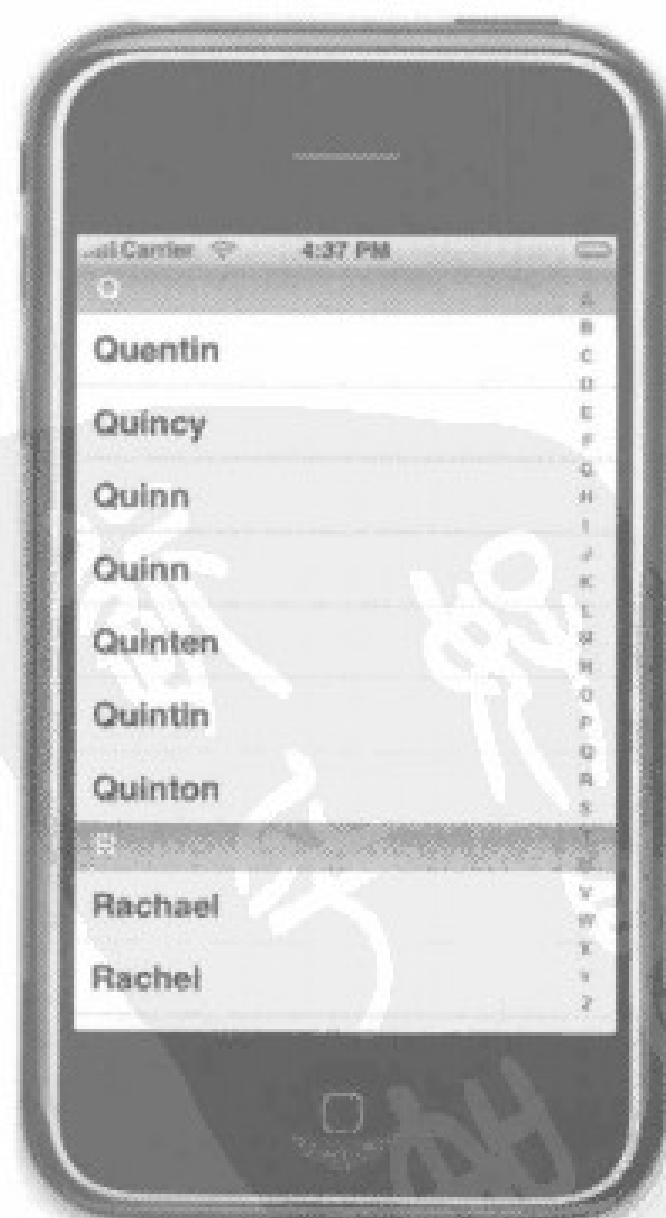


图8-22 带有一个索引的索引表视图

8.7 实现搜索栏

索引是很有用的,即便如此,此处我们仍然有非常多的名称。例如,如果要查看名称Arabella是否存在于列表中,使用索引之后仍然需要拖动滚动条。如果能够通过指定搜索项简化该列表就好了,对不对?这样对用户更友好。那只是一点额外的工作,并不是特别糟糕。我们将实现一个标准的iPhone搜索栏,如图8-23所示。

8.7.1 重新考虑设计

在我们开始着手做之前,需要考虑一下应用程序是如何工作的。当前,我们拥有一个包含多个数组的字典,其中字母表中的每个字母都占有一个数组。该字典是不可改变的,这就意味着不能从字典添加或删除值,它包含的数组也是如此。当用户取消搜索或者更改搜索项时,还必须能够返回到源数据集。

我们能做的就是创建两个新的字典:一个包含完整数据集的不可改变的字典、一个可以从中删除行的可变的字典副本。委托和数据源将从可变字典进行读取,当搜索标准更改或者取消搜索时,可以从不可改变的字典刷新可变字典。听起来像个好计划,下面就开始吧。



图8-23 带有搜索栏的应用程序

注意 下一个项目有些复杂,阅读得太快会让你感到有些不知所措。如果某些概念让你感到头疼,请查阅Mark Dalrymple和Scott Knaster所著的*Learn Object-C* (Apress, 2009),阅读此书中与类别和易变性有关的内容。

8.7.2 深层可变副本

现在存在一个问题, `NSDictionary` 遵循 `NSMutableCopying` 协议,该协议返回一个 `NSMutableDictionary`,但是这个方法创建的是浅副本。也就是说,调用 `mutableCopy` 方法时,它将创建一个新的 `NSMutableDictionary` 对象,该对象拥有源字典所拥有的所有对象。它们并不是副本,而是相同的实际对象。如果我们所处理的字典仅存储字符串,这会很好,因为从副本中删除一个值不会对源字典产生任何影响。然而,由于字典中存有数组,如果从副本的数组中删除对象,这些对象也将从源字典的数组中被删除,因为副本和源都指向相同的对象。

要解决这一问题,需要为存有数组的字典创建一个深层可变副本。这并不难,不过我们应该把这项功能放在哪儿呢?

如果你说“在类别中”,那么好极了,你的想法是正确的。如果还没有想到,别担心,习惯这种语言需要花费一定的时间。通过类别可以向现有对象添加附加方法,而不需要子类化这些对象。类别经常会被Objective-C的新手忽略,因为这些特性是大多数其他语言所没有的。

通过类别,我们可以向 `NSDictionary` 添加一个方法来实现深层副本,返回的 `NSMutableDictionary`

Dictionary拥有相同的数据，但不包含相同的实际对象。

在项目窗口中，选择Classes文件夹，然后按下⌘N创建一个新文件。向导出现之后，从最底端的左侧选择Other。然而，类别没有文件模板，我们要创建几个空文件来保存它。选择Empty File图标，将第一个文件命名为NSDdictionary-MutableDeepCopy.h。重复此过程，将第二个文件命名为NSDdictionary-MutableDeepCopy.m。

提示 创建类别所需的两个文件的一个更快的方法就是，选择NSObject子类模板，然后删除文件内容。此选项将同时提供头文件和实现文件，为你省了一步。

将下面的代码添加到NSDdictionary-MutableDeepCopy.h中：

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

```
@interface NSDictionary(MutableDeepCopy)
- (NSMutableDictionary *)mutableDeepCopy;
@end
```

切换到NSDdictionary-MutableDeepCopy.m，并添加以下实现：

```
#import "NSDictionary-MutableDeepCopy.h"
```

```
@implementation NSDictionary (MutableDeepCopy)
- (NSMutableDictionary *) mutableDeepCopy
{
    NSMutableDictionary *ret = [NSMutableDictionary dictionaryWithCapacity:
        [self count]];
    NSArray *keys = [self allKeys];
    for (id key in keys)
    {
        id oneValue = [self valueForKey:key];
        id oneCopy = nil;

        if ([oneValue respondsToSelector:@selector(mutableDeepCopy)])
            oneCopy = [oneValue mutableDeepCopy];
        else if ([oneValue respondsToSelector:@selector(mutableCopy)])
            oneCopy = [oneValue mutableCopy];
        if (oneCopy == nil)
            oneCopy = [oneValue copy];
        [ret setValue:oneCopy forKey:key];
    }
    return ret;
}
@end
```

此方法创建一个新的可变字典，然后在源字典中的所有键中进行迭代，为它遇到的每个数组创建可变副本。由于此方法将表现得好像它是NSDictionary的一部分，所以对self的任何引用都是对调用到此方法上的字典的一个引用。此方法首先尝试创建深层可变副本，如果对象没有响应mutableDeepCopy消息，那么它将尝试创建可变副本，如果对象没有响应mutableCopy消息，它就

回过头再创建常规副本，以确保对字典中包含的所有对象都创建了副本。通过这种方式，如果我们有一个包含字典（或支持深层可变副本的其他对象）的字典，则也将对所包含的内容创建深层副本。

对于大多数读者来说，这可能是第一次在Objective-C中看到如下语法：

```
for (id key in keys)
```

Objective-C 2.0还有一种新特性，叫做快速枚举。快速枚举是NSEnumerator的语言级替代，《Learn Objective C》一书对它进行了介绍。可以通过快速枚举对集合（如NSArray）进行迭代，从而避免了创建附加对象的麻烦。

所有已交付的Cocoa集合类，包括NSDictionary、NSArray和NSSet，都支持快速枚举。而且可以在任何需要的时候使用此语法对集合进行迭代。它将确保你获得效率最高的循环。

你可能注意到了，我们正在使用一个便利类方法创建ret字典，虽然我们以前曾警告过不要使用不必要的便利方法。使用便利方法的问题是，由便利方法返回的对象被放到自动释放池，并至少要等待当前事件循环结束，即使它的保留计数下跌到零。很显然，这里的问题是内存的低使用率。

在这种情况下，当方法结束后返回值时，必须在该方法上调用autorelease，以确保它依然为调用此方法的代码而存在。因为不管怎么样，它也要到自动释放池中，我们可以利用便利方法，省去了键入一行或两行代码的工作。

如果在其他类中包含NSDictionary-MutableDeepCopy.h头文件，我们将能够在任何喜欢的NSDictionary对象上调用mutableDeepCopy。现在让我们使用这一功能。

8.7.3 更新控制器头文件

下一步，我们需要向控制器头文件添加一些输出口。表视图需要一个输出口。目前为止，我们还没有用到数据源方法之外指向表视图的指针，现在就需要一个，因为我们需要根据搜索结果通知表重新加载它自己。

我们还需要一个指向搜索栏的输出口，它是一个用于搜索的控件。除了这两个输出口之外，还将需要一个附加字典。现有的字典和数组都是不可改变的对象，我们需要把它们改为相应的可变版本，这样，NSArray就变成了NSMutableArray，NSDictionary就变成了NSMutableDictionary。

在控制器中不再需要任何新的操作方法，不过我们需要几个新的方法。目前，仅对它们进行声明，输入代码之后再对它们进行详细的讨论。

还需要让类遵循UISearchBarDelegate协议。除了充当表视图的委托之外，我们还需要让它充当搜索栏的委托。

在SectionsViewController.h文件中做如下更改：

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

```
@interface SectionsViewController : UIViewController
    <UITableViewDataSource, UITableViewDelegate, UISearchBarDelegate>
```

```

{
    IBOutlet    UITableView *table;
    IBOutlet    UISearchBar *search;
    NSDictionary *allNames;
    NSMutableDictionary *names;
    NSMutableArray *keys;
    NSDictionary *names;
    NSArray *keys;
}

@property (nonatomic, retain) NSDictionary *names;
@property (nonatomic, retain) NSArray *keys;
@property (nonatomic, retain) UITableView *table;
@property (nonatomic, retain) UISearchBar *search;
@property (nonatomic, retain) NSDictionary *allNames;
@property (nonatomic, retain) NSMutableDictionary *names;
@property (nonatomic, retain) NSMutableArray *keys;
- (void)resetSearch;
- (void)handleSearchForTerm:(NSString *)searchTerm;
@end

```

这就是我们所做的更改。table输出口将指向表视图；search输出口将指向搜索栏；allNames字典将存有所有数据集；names字典将存有那些与当前搜索标准匹配的数据集；keys将存有索引值和分区名称。如果清楚地理解了这些内容，下面让我们在Interface Builder中修改视图。

8.7.4 修改视图

在Interface Builder中双击SectionsViewController.xib，打开该文件。打开之后，选中表视图，然后使用顶部的调整大小的工具将视图缩短一些，为顶部的搜索栏腾出一些空间。不要担心如何精确地进行调整，你在一秒钟之内就会调整到最好。提示一下，如果你不知道如何显示调整大小工具，可以将nib主窗口改为列表模式，然后双击Table View，这会选中Table View并显示调整大小的工具。

下一步，从库中选择Search Bar（参见图8-24），然后把它添加到视图的顶部。

调整搜索栏和表视图，使它们看起来好看一些。表视图的顶部应该正好挨着搜索栏的底部，这两个控件应该正好占据视图的全部空间，如图8-25所示。

现在，按下Control键并将File's Owner图标拖到表视图，然后选中表输出口。同样对搜索栏重复上述操作，然后选中搜索输出。单击搜索栏，按下⌘1键，打开属性检查器，如图8-26所示。

选中Shows Cancel Button复选框。在Placeholder字段中键入search，搜索字段的右侧会出现一个Cancel按钮，用户可以使用此按钮取消搜索。搜索字段中将出现以灰色字母显示的占位符文本search。

按下⌘2，转到连接检查器，从delegate连接拖到File's Owner图标，以告知搜索栏视图控制器也是搜索栏的委托。

这就是所有我们所需要的，因此一定要进行保存。下面让我们回到Xcode。

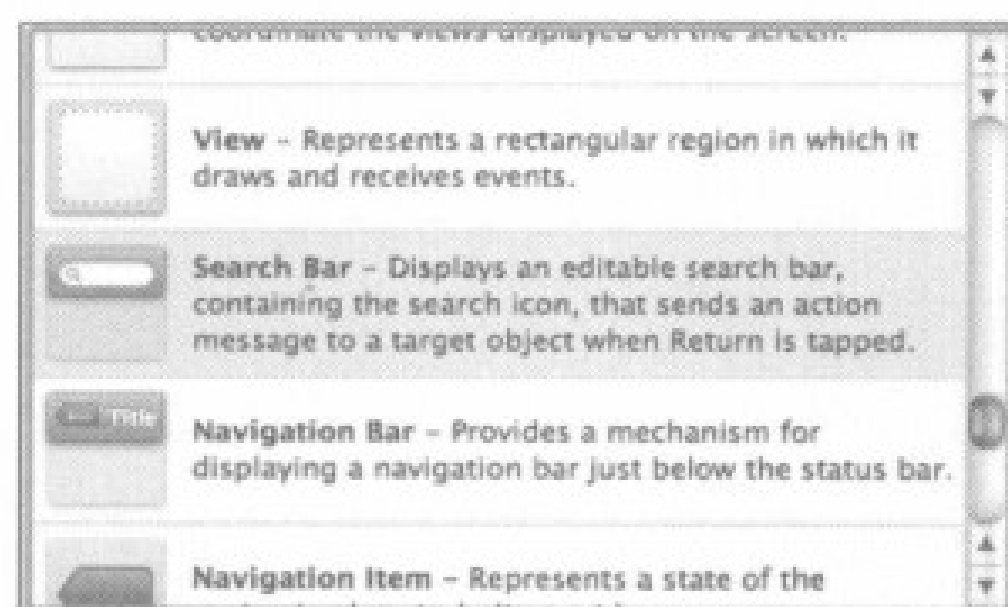


图8-24 库中的Search Bar

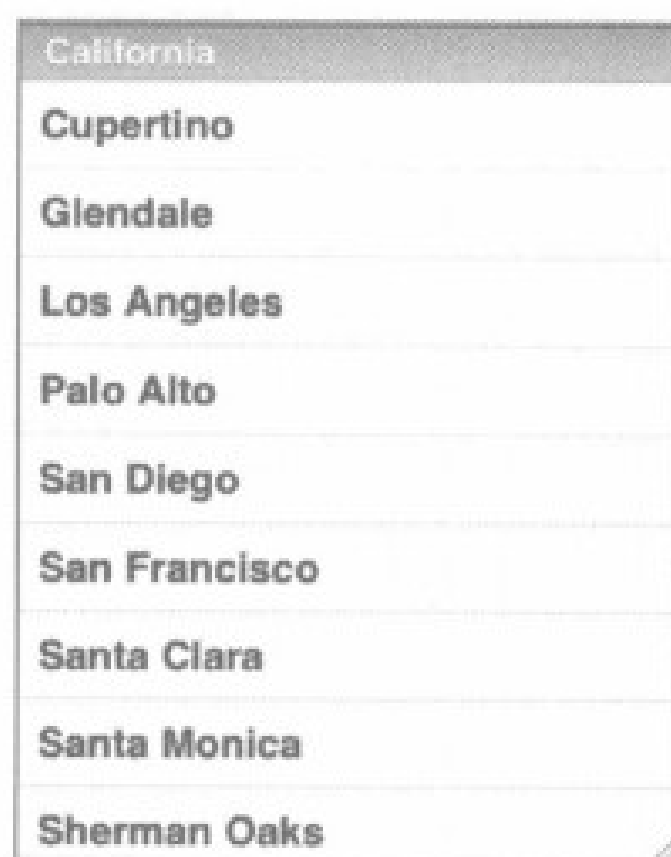


图8-25 带有表视图和搜索栏的新视图

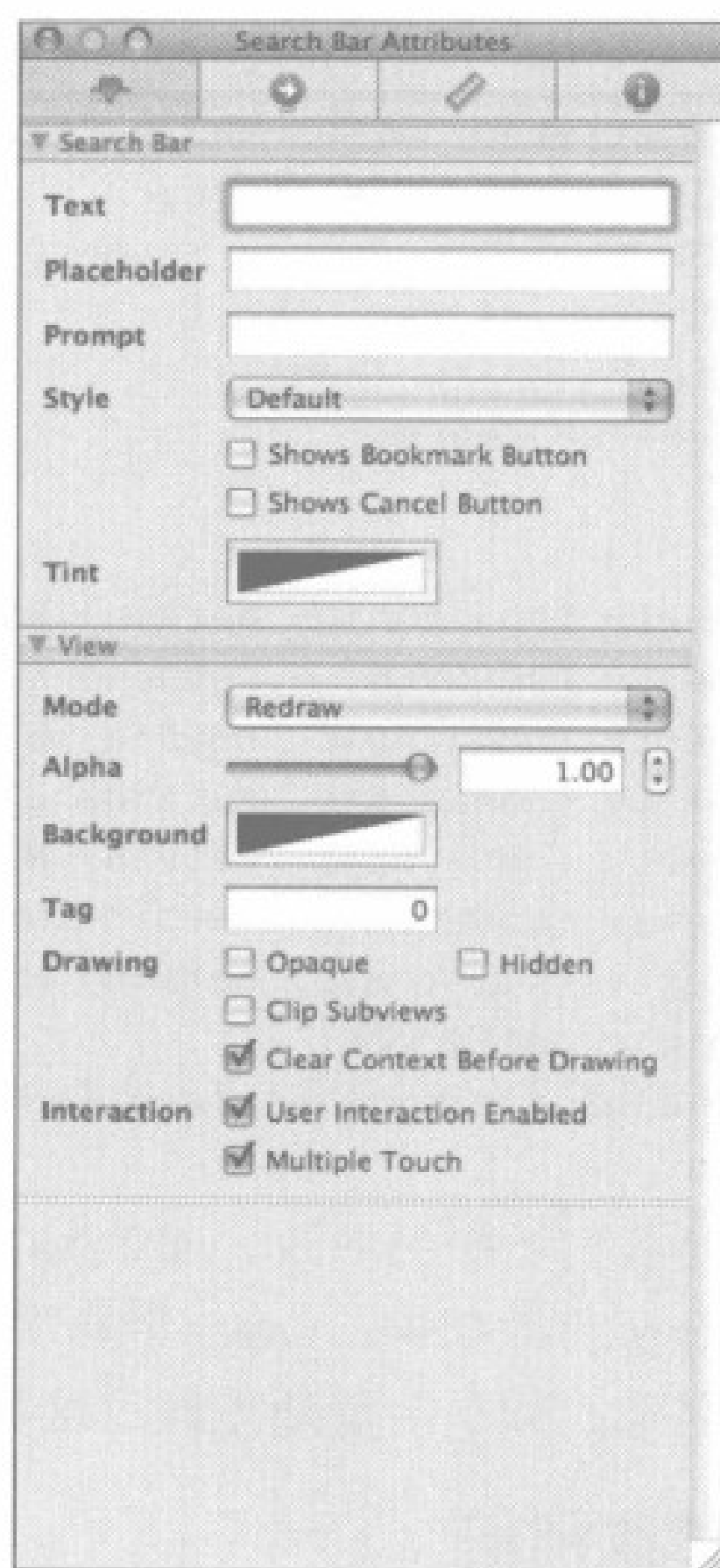


图8-26 搜索栏的属性检查器

8.7.5 修改控制器实现

对搜索栏进行的更改非常大。对SectionsViewController.m做如下更改，然后快速返回，查看这些更改。

```
#import "SectionsViewController.h"
#import "NSDictionary-MutableDeepCopy.h"

@implementation SectionsViewController
@synthesize names;
@synthesize keys;
@synthesize table;
@synthesize search;
@synthesize allNames;
#pragma mark -
#pragma mark Custom Methods
- (void)resetSearch
{
    self.names = [self.allNames mutableDeepCopy];
    NSMutableArray *keyArray = [[NSMutableArray alloc] init];
```

```

        [keyArray addObjectsFromArray:[self.allNames allKeys]
            sortedArrayUsingSelector:@selector(compare:)];
        self.keys = keyArray;
        [keyArray release];
    }
    - (void)handleSearchForTerm:(NSString *)searchTerm
    {
        NSMutableArray *sectionsToRemove = [[NSMutableArray alloc] init];
        [self resetSearch];

        for (NSString *key in self.keys) {
            NSMutableArray *array = [names valueForKey:key];
            NSMutableArray *toRemove = [[NSMutableArray alloc] init];
            for (NSString *name in array) {
                if ([name rangeOfString:searchTerm
                    options:NSCaseInsensitiveSearch].location == NSNotFound)
                    [toRemove addObject:name];
            }

            if ([array count] == [toRemove count])
                [sectionsToRemove addObject:key];
            [array removeObjectsInArray:toRemove];
            [toRemove release];
        }
        [self.keys removeObjectsInArray:sectionsToRemove];
        [sectionsToRemove release];
        [table reloadData];
    }
#pragma mark -
#pragma mark UIViewController Methods

    - (void)viewDidLoad {
        NSString *path = [[NSBundle mainBundle] pathForResource:@"sortednames"
            ofType:@"plist"];
        NSDictionary *dict = [[NSDictionary alloc]
            initWithContentsOfFile:path];
        self.names = dict;
        self.allNames = dict;

        [dict release];

        NSArray *array = [self.names allKeys] sortedArrayUsingSelector:
            @selector(compare:);
        self.keys = array;

        [self resetSearch];
        search.autocapitalizationType = UITextAutocapitalizationTypeNone;
        search.autocorrectionType = UITextAutocorrectionTypeNo;
    }

```

```

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {;
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [table release];
    [search release];
    [allNames release];
    [keys release];
    [names release];
    [super dealloc];
}
#pragma mark -
#pragma mark Table View Data Source Methods
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return [keys count];
    return ([keys count] > 0) ? [keys count] : 1;
}
- (NSInteger)tableView:(UITableView *)aTableView
    numberOfRowsInSection:(NSInteger)section
{
    if ([keys count] == 0)
        return 0;
    NSString *key = [keys objectAtIndex:section];
    NSArray *nameSection = [names objectForKey:key];
    return [nameSection count];
}
- (UITableViewCell *)tableView:(UITableView *)aTableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSUInteger section = [indexPath section];
    NSUInteger row = [indexPath row];

    NSString *key = [keys objectAtIndex:section];
    NSArray *nameSection = [names objectForKey:key];

    static NSString *sectionsTableIdentifier = @"sectionsTableIdentifier";

```

```
UITableViewCell *cell = [aTableView dequeueReusableCellWithIdentifier:
    sectionsTableIdentifier];
if (cell == nil) {
    cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
        reuseIdentifier: sectionsTableIdentifier] autorelease];
}

cell.text = [nameSection objectAtIndex:row];
return cell;
}

- (NSString *)tableView:(UITableView *)tableView
    titleForHeaderInSection:(NSInteger)section
{
    if ([keys count] == 0)
        return @"";

    NSString *key = [keys objectAtIndex:section];
    return key;
}

- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)tableView
{
    return keys;
}

#pragma mark -
#pragma mark Table View Delegate Methods
- (NSIndexPath *)tableView:(UITableView *)tableView
    willSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [search resignFirstResponder];
    return indexPath;
}

#pragma mark -
#pragma mark Search Bar Delegate Methods
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
    NSString *searchTerm = [searchBar text];
    [self handleSearchForTerm:searchTerm];
}

- (void)searchBar:(UISearchBar *)searchBar
    textDidChange:(NSString *)searchTerm
{
    if ([searchTerm length] == 0)
    {
        [self resetSearch];
        [table reloadData];
        return;
    }
}
```

```

        [self handleSearchForTerm:searchTerm];
    }
    - (void)searchBarCancelButtonClicked: (UISearchBar *)searchBar
    {
        search.text = @"";
        [self resetSearch];
        [table reloadData];
        [searchBar resignFirstResponder];
    }
@end

```

1. 从allNames复制数据

键入上述所有代码之后，你还跟得上进度吗？让我们停下来看一看刚才的代码。先看一下添加的两个新方法。这是第一个方法：

```

- (void)resetSearch
{
    self.names = [self.allNames mutableDeepCopy];
    NSMutableArray *keyArray = [[NSMutableArray alloc] init];
    [keyArray addObjectsFromArray:[self.allNames allKeys]
        sortedArrayUsingSelector:@selector(compare:)];
    self.keys = keyArray;
    [keyArray release];
}

```

取消搜索或更改搜索条件时将调用此方法。它所做的是创建allNames的可变副本，将它赋值为names，然后刷新keys数组，使它包含字母表中的所有字母。我们必须刷新keys数组，因为如果某搜索排除了某分区中的所有值，那么还需要除去这一分区。否则，屏幕将被标题和空的分区充满，这看起来很糟糕。我们也不希望为不存在的内容提供索引，因此根据搜索短语挑选名称时，还需要去除空的分区。

2. 实现搜索

另一个方法实现了实际的搜索：

```

- (void)handleSearchForTerm:(NSString *)searchTerm
{
    NSMutableArray *sectionsToRemove = [[NSMutableArray alloc] init];
    [self resetSearch];

    for (NSString *key in self.keys)
    {
        NSMutableArray *array = [names valueForKey:key];
        NSMutableArray *toRemove = [[NSMutableArray alloc] init];
        for (NSString *name in array) {
            if ([name rangeOfString:searchTerm
                options:NSCaseInsensitiveSearch].location == NSNotFound)
                [toRemove addObject:name];
        }
    }
}

```

```

        if ([array count] == [toRemove count])
            [sectionsToRemove addObject:key];
        [array removeObjectsWithIdentifiers:toRemove];
        [toRemove release];
    }
    [self.keys removeObjectsWithIdentifiers:sectionsToRemove];
    [sectionsToRemove release];
    [table reloadData];
}

```

虽然我们将在搜索栏委托方法中进行搜索，但仍然要把`handleSearchForTerm:`拖到自己的方法中，因为我们将需要在两个不同的委托方法中使用相同的功能。通过在`handleSearchForTerm:`方法中嵌入搜索，我们将功能固定在一个位置，这样便于维护，然后只在需要时调用这个新的方法即可。

这是此部分真正有趣的地方，让我们把这个方法分为几小段来看一下。首先，创建一个数组，它将存有我们找到的空分区。后面使用此数组删除这些空分区，因为在一个集合中进行迭代时，从该集合中删除对象是不安全的。由于我们正使用快速枚举，这样做将引发异常。因此，我们不能在键中进行迭代时删除键，就把将要删除的分区存储在一个数组中，在完成所有枚举之后一次删除所有的对象。分配数组之后，重置搜索：

```

NSMutableArray *sectionsToRemove = [[NSMutableArray alloc] init];
[self resetSearch];

```

下一步，枚举新存储的`keys`数组中的所有键。

```

for (NSString *key in self.keys)
{

```

每次进行循环时，获取对应于当前键的名称数组，并创建存有需要从`names`数组中删除的值的数组。记住，我们删除的是名称和分区，因此需要知道哪些键是空的，以及哪些名称与搜索条件不匹配。

```

    NSMutableArray *array = [names valueForKey:key];
    NSMutableArray *toRemove = [[NSMutableArray alloc] init];

```

接下来，在当前数组中对所有名称进行迭代。因此，如果当前正处理键“A”，那么此循环将迭代所有以“A”开头的名称。

```

    for (NSString *name in array)
    {

```

此循环使用了一个返回字符串中子字符串位置的`NSString`方法。通过指定`NSCaseInsensitiveSearch`选项，表明我们不关心搜索短语的大小写。也就是说，“A”相当于“a”。此方法返回的值是一个`NSRange`结构，它带有两个成员：`location`和`length`。如果没有找到搜索短语，`location`将被设置为`NSNotFound`，因此只要检查有没有`NSNotFound`就可以了。如果返回的`NSRange`包含`NSNotFound`，就把名称添加到将要删除的对象数组中。

```

        if ([name rangeOfString:searchTerm
            options:NSCaseInsensitiveSearch].location == NSNotFound)

```

```

        [toRemove addObject:name];
    }

```

对给定字母的所有名称迭代完成之后，检查将要删除的名称数组的长度是不是和名称数组的长度相同，如果相同，则这个分区就是空的，我们将把它添加到键的数组中，以备将来删除。

```

    if ([array count] == [toRemove count])
        [sectionsToRemove addObject:key];

```

下一步，从此分区的数组中删除不匹配的名称，然后释放用于存储名称的数组。尽可能地避免在这样的循环中使用便利方法很重要，因为它们会在每次循环中将一些东西放入自动释放池。然而，直到完成循环，自动释放池才会充满。

```

        [array removeObjectsWithIdenticalObjects:toRemove];
        [toRemove release];
    }

```

最后，删除空分区，释放用于存储空分区的数组，并告知表重新加载数据。

```

        [self.keys removeObjectsWithIdenticalObjects:sectionsToRemove];
        [sectionsToRemove release];
        [table reloadData];
    }

```

3. 修改viewDidLoad

在viewDidLoad中，我们进行了一些更改。首先，把属性列表加载到allNames字典而不是names字典，删除加载keys数组的代码，因为现在使用resetSearch方法完成加载。然后调用resetSearch方法，它填充names可变字典和keys数组。下一步，在搜索栏上发起两个调用，以进行在Interface Builder中不能设置的一些配置，因为受影响的设置在属性检查器中不可用：

```

search.autocapitalizationType = UITextAutocapitalizationTypeNone;
search.autocorrectionType = UITextAutocorrectionTypeNo;

```

由于搜索是不区分大小写的，因此不需要把用户键入到搜索栏的内容变为大写。我们还不希望使用自动校正，因为许多搜索短语可能是名称的一部分，我们不希望搜索对它们进行自动校正。

4. 修改数据源方法

如果跳转到数据源方法，你会发现我们对该方法做了一些微小的修改。因为names字典和keys数组依然用于提供数据源，这些方法基本上和以前相同。我们必须说明这样一个事实：表视图始终拥有分区的基本部分，而且搜索可能把所有名称排除在所有分区之外。因此，我们添加了一些代码来检查删除了所有分区的情况，在那种情况下，我们为表视图提供了一个没有行且只有一个空白名称的分区。这避免了所有问题，不会给用户任何错误的反馈。

5. 添加表视图委托方法

在数据源方法下面添加一个委托方法。如果用户在使用搜索栏时单击一行，我们希望键盘不再起作用。这是通过实现tableView:willSelectRowAtIndexPath:并告知搜索栏放弃第一响应者状态（这将关闭键盘）来完成的。下一步，返回未改变的indexPath。我们还可以在tableView:didSelectRowAtIndexPath:中完成，不过由于这里是这样做的，键盘会取消得稍微快一些。


```

- (NSIndexPath *)tableView:(UITableView *)tableView
willSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [search resignFirstResponder];
    return indexPath;
}

```

6. 添加搜索栏委托方法

搜索栏有许多在其委托上调用的方法。当用户点击键盘上的返回按钮或搜索键时，将调用 `searchBarSearchButtonClicked:`。此方法从搜索栏获取搜索短语，并调用我们的搜索方法，这个搜索方法将删除 `names` 中不匹配的名称和 `keys` 中的空分区。

```

- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
    NSString *searchTerm = [searchBar text];
    [self handleSearchForTerm:searchTerm];
}

```

每次使用搜索栏的时候都应该实现 `searchBarSearchButtonClicked:` 方法。除此之外，我们还实现了另一个搜索栏委托方法，不过实现这个方法需要谨慎一些。这个方法用于实现现场搜索 (live search)。每次更改搜索短语时，不管用户是否选中了搜索按钮或点击了返回，我们都重新进行搜索。这种行为具有非常高的用户友好性，因为用户在键入时可以看见结果的改变。如果用户在键入第3个字符后减少了足够长的列表，那么他们可以停止键入，然后选择想要的行。

实现现场搜索会减弱应用程序的性能，尤其是在显示图像或拥有复杂数据模型的时候。在这种情况下，如果只有2000个字符串，没有图像或扩展图标，程序会运行得非常好，即使是在第一代iPhone或iPod Touch上。

不要以为仿真器中的高性能可以转换为你的设备上的高性能。如果你打算实现这样的现场搜索，就需要在实际硬件上做大量测试，以确保应用程序保持响应。拿不定主意的时候不要使用它。或许用户很乐意敲击搜索按钮呢。

现在，你已经获得了足够的警告，下面的代码将处理一个现场搜索。实现搜索栏委托方法 `searchBar:textDidChange:` 的代码为如下所示：

```

- (void)searchBar:(UISearchBar *)searchBar
textDidChange:(NSString *)searchTerm
{
    if ([searchTerm length] == 0)
    {
        [self resetSearch];
        [table reloadData];
        return;
    }
    [self handleSearchForTerm:searchTerm];
}

```

注意，这里我们查找一个空的字符串。如果字符串为空，那么所有名称都将与它匹配，因此我们只要重置搜索并重新加载数据就可以了，而不需要枚举所有名称。

最后，我们实现一个方法，当用户在搜索栏上单击Cancel按钮时，我们能得到通知。

```
- (void)searchBarCancelButtonClicked:(UISearchBar *)searchBar
{
    search.text = @"";
    [self resetSearch];
    [table reloadData];
    [searchBar resignFirstResponder];
}
```

当用户单击Cancel时，程序会将搜索短语设置为空字符串，然后重置搜索，并重新加载数据以显示所有名称。此外，还需要让搜索栏生成第一响应者状态，这样键盘就不再起作用，以便于用户重新处理表视图。

8.8 小结

感觉怎么样？这一章的内容极其重要，你已经学习了很多！你应该对平面表（flat table）的工作方式有了非常好的理解，并了解了如何自定义表和表视图单元，以及如何配置表视图。你还了解了如何实现搜索栏，在任何呈现大量数据的iPhone应用程序中，这都是一个至关重要的工具。要确保真正理解了本章中的所有内容，因为本章是后面章节的基础。

下一章将继续介绍表视图，你将学习如何使用表视图呈现分层数据。你将了解如何创建内容视图，以使用户能够编辑表视图中选中的数据，以及如何在表中呈现检验表，在表的行中嵌入控件和删除行。



在第8章中，你已经掌握了有关表视图的基础知识。在本章中，我们将共同探讨导航控制器，并完成更多的练习。导航控制器和表视图密不可分。严格来说，要完成导航控制器的功能并不需要表视图。然而，在实际的应用程序中实现导航控制器时，几乎总是要实现至少一个表，并且通常是多个表，因为导航控制器的强大之处在于它能处理复杂的分层数据。在iPhone的小屏幕上，连续的表视图是表示分层数据最理想的方式。

本章将逐步地构建一个应用程序，这与第7章构建的标签视图（tab view）应用程序一样。我们先让导航控制器和第一个视图控制器工作起来，然后开始向分层结构添加更多的控制器和层。我们创建的每一个视图控制器都将增强表或配置的某些方面。你将了解如何把表视图中（的数据）写入到子表中，以及如何把表视图中的数据写入可以读取甚至编辑详细数据的内容视图中。你还将了解如何通过使用表列表来从多个值中进行选择，以及如何使用编辑模式从表视图中删除行。

内容非常丰富吧？那我们还等什么，开始吧。

9.1 导航控制器

UINavigationController是用于构建分层应用程序的主要工具，它在管理以及换入和换出多个内容视图方面与UITabBarController较为类似。两者之间的主要不同在于，UINavigationController是作为栈（stack）来实现的，这让它非常适合用于处理分层数据。

如果你已经了解了栈的来龙去脉，请快速浏览9.1.1节的内容，然后开始学习9.1.2节。你是第一次接触栈吗？非常幸运，它是一个非常简单的概念。

9.1.1 栈的性质

栈是一种常用的数据结构，采用后进先出的原则。不管你是否相信，Pez糖果盒（dispenser）是栈的一个极好的例子。想试一试吗？根据每个Pez糖果盒随带的说明书，一共有几个简单的步骤。第一步，打开Pez糖果包装。第二步，直接敲击糖果盒的顶部，打开它。第三步，抓紧糖果栈（注意这里我们聪明地使用了“栈”这个字），在食指和拇指之间牢牢地握住它，然后将糖果栈插入打开的糖果盒内。第四步，捡起洒落一地的糖果，因为说明书根本没用。

这个例子并不是特别实用。不过，下面发生的事情是：当你捡起糖果并一次一个地把它们塞进糖果盒时，你所操作的就是一个栈。还记得吗？我们说过栈是后进先出的，也可以说是先进后

出。放到糖果盒的第一个糖果将是最后一个出来的，最后一个被塞入的糖果将是第一个出来的。

计算机栈遵循同样的规则。向栈中添加对象的操作称为入栈（push），即把对象推到栈中；从栈中删除对象的操作称为出栈（pop）。要让一个对象出栈时，这个对象往往最后一个入栈的。第一个入栈的对象往往最后一个出栈。

9.1.2 控制器栈

导航控制器维护一个视图控制器栈。任何类型的视图控制器都可以放入栈中。在设计导航控制器时，你需要指定用户看到的第一个视图。该视图是视图层次结构中最底层的视图，其控制器称为根视图控制器（root view controller）。根视图控制器是被导航控制器推入到栈中的第一个视图控制器。当用户选择查看下一个视图时，栈中将加入一个新的视图控制器，它所控制的视图将展示给用户。我们把这些新的视图控制器称为子控制器（subcontroller）。可以看出，本章的Nav应用程序由一个导航控制器和6个子控制器组成。

查看图9-1。注意当前视图左上角的导航按钮。这个导航按钮类似于网页浏览器的后退按钮。当用户单击该按钮时，当前的视图控制器出栈，下一个视图成为当前视图。

我们热衷于这种设计模式。通过它，我们可以反复地构建复杂的分层应用程序。你不需要了解整个分层结构的复杂性。每个控制器只需要知道其子控制器，以便在用户做出选择时把适当的新的控制器对象加入到栈中。通过这种方式，你可以把若干小部件组合成一个大型应用程序，这正是本章所要介绍的内容。

9.2 由6个部分组成的分层应用程序：Nav

我们构建的应用程序将向你展示有关如何显示分层数据的常见问题。应用程序运行后将显示一个选项列表（如图9-2所示）。此顶级视图中的每一行分别表示一个不同的视图控制器，当选中其中一行时，对应的视图控制器将被加入到导航控制器栈中。

每行右侧的灰色箭头是扩展图标（accessory icon）。这种特别的扩展图标被称为扩展指示器（disclosure indicator），

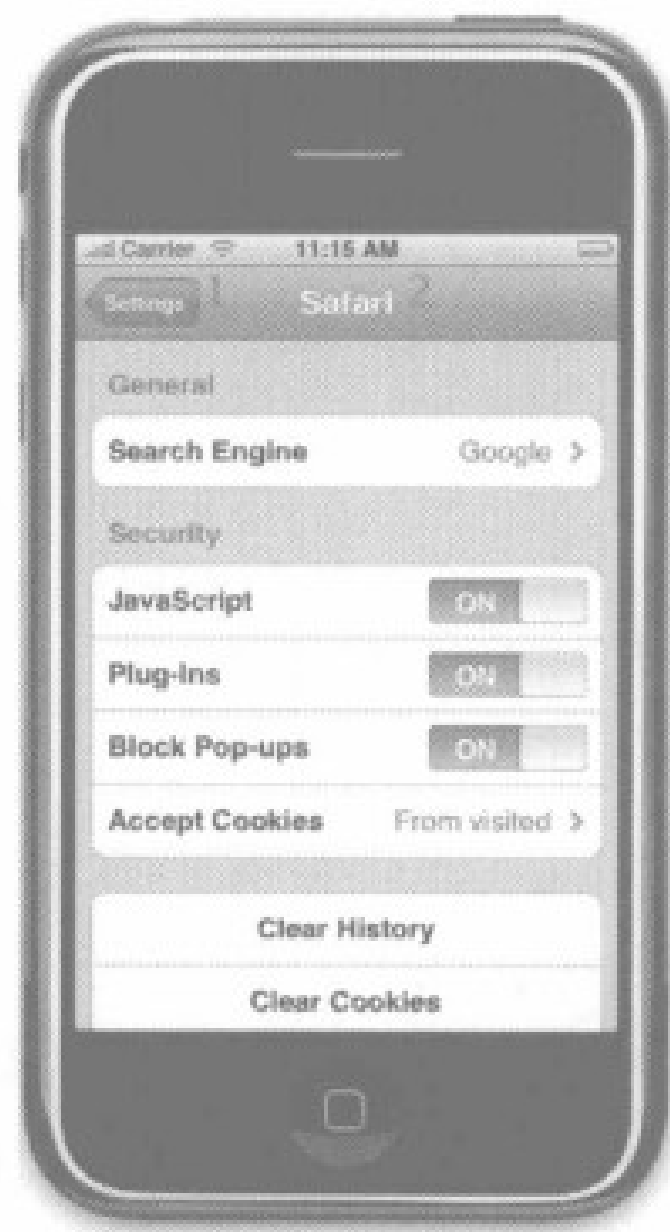


图9-1 Settings应用程序使用了一个导航控制器。左上方（1）是用于让当前视图控制器出栈的导航按钮，它返回分层结构的上一级。还显示了当前内容视图控制器的标题（2）

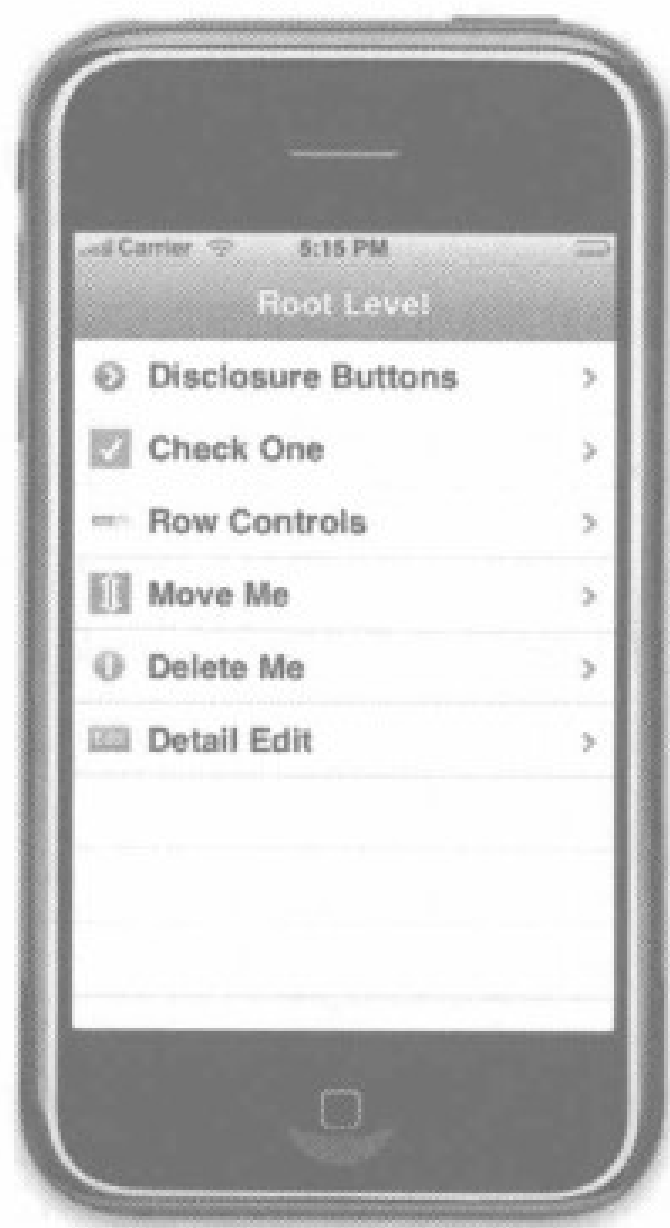


图9-2 本章应用程序的顶级视图。注意视图右侧的扩展图标。这种特别的扩展图标被称为扩展指示器，它用于告知用户触摸这一行将切换到另一个表视图

用于告知用户触摸该箭头将切换到另一个表视图。

使用扩展指示器切换到包含选中行详细信息的视图是不合适的。此处，我们使用的是**细节展示按钮**（detail disclosure button），如图9-3所示，它显示应用程序6个子控制器中的第一个。从图9-2所示的顶部视图中选择Disclosure Buttons就会出现这个视图。细节展示按钮告诉你，选择该行将显示有关当前行的更多详细信息，并允许你对它们进行编辑。

与扩展指示器不同，细节展示按钮不仅仅是一个图标，它还是一个可单击的控件，因此一个给定的行可以有两个不同的选项。当用户选择该行时触发一个操作；当用户单击展示按钮时触发另一个操作。

iPhone应用程序是正确使用细节展示按钮的一个较好的例子。选择Favorites标签中的联系人将对该联系人发起呼叫，而选择联系人名字旁边的展示按钮将显示其详细联系信息。YouTube应用程序是另一个恰到好处的例子。选择某行会播放相应视频，而单击细节展示按钮则会显示有关该视频的更多详细信息。

在Contacts应用程序中，虽然选择一行会显示详细视图，但联系人列表中没有细节展示按钮。由于在Contact应用程序中每一行只有一个可用的选项，因此不显示任何扩展图标。

再说一次，单击某行将显示该行的详细视图，但如果希望该行支持两个不同的选项，则不必使用扩展图标，而应使用细节展示按钮。如果希望单击某行时显示另一个完全不同的视图，且该视图并不是该行的详细视图，则应使用扩展指示器（灰色箭头）标记该行。

应用程序的第2个子控制器如图9-4所示。在图9-2中选择Check One将出现该视图。

该视图可用于呈现“多选一”列表。在iPhone中，该列表的作用就如同Mac OS X中的单选按钮。此列表使用选中标记来标记当前选中的行。

图9-5显示了应用程序的第3个子控制器。此视图在每一行的扩展视图中添加了一个开关控件。扩展视图位于表视图单元的最右侧，它通常用于存放扩展图标，但其用途远不止于此。在讨论应用程序的这一部分时，你将了解如何创建和检索放置在扩展视图中的控件的值。

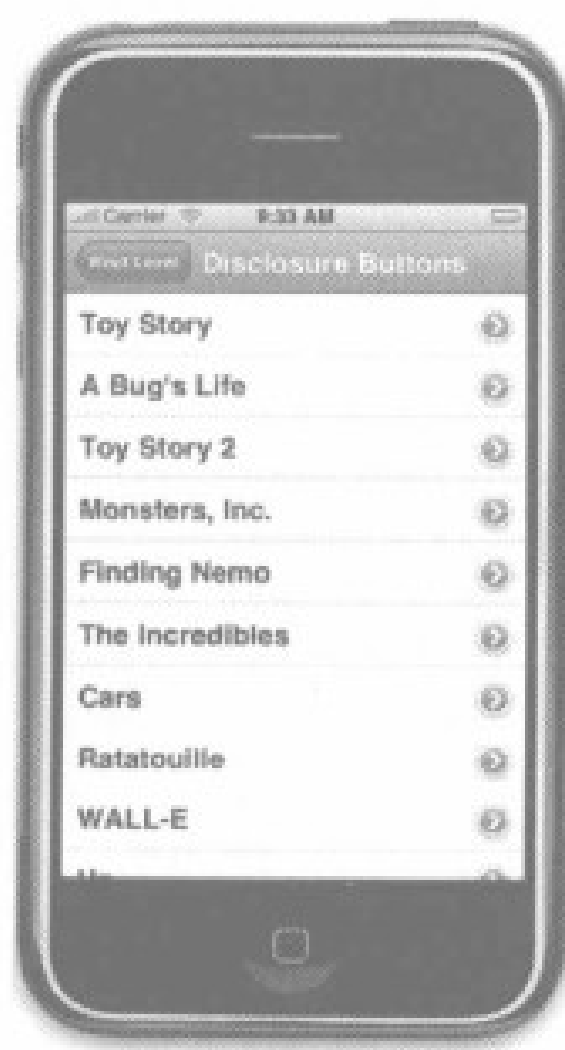


图9-3 Nav应用程序的第1个子控制器实现了一个表，表中的每一行都包含一个细节展示按钮



图9-4 Nav应用程序的第2个子控制器允许执行“多选一”操作



图9-5 Nav应用程序的第3个子控制器在每个表视图单元的扩展视图都添加了一个开关

图9-6显示了应用程序的第4个子控制器。在这个视图中，通过将表切换为编辑模式（本章稍后将对这一概念进行详细介绍），用户可以对列表中的行重新排序。

图9-7显示了应用程序的第5个子控制器。在这个视图中，我们将展示编辑模式的另一种用法，即删除表中的行。

图9-8显示了应用程序的第6个子控制器，同时也是最后一个。它使用分组表显示了一个可编辑的详细视图。详细视图这项技术在iPhone应用程序中得到了广泛应用。

原来有这么多工作要做。现在让我们开始吧！



图9-6 Nav应用程序的第4个子控制器允许用户通过触摸和拖动移图标对列表中的行重新排序



图9-7 Nav应用程序的第5个子控制器实现了允许用户从表中删除项的编辑模式

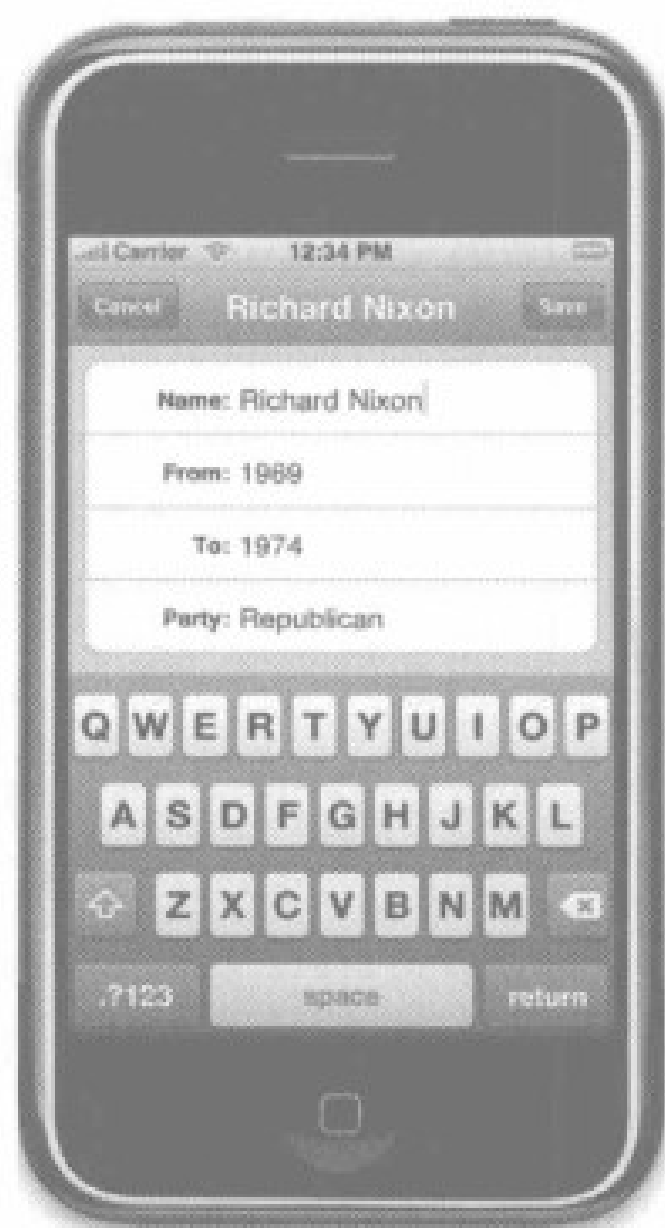


图9-8 Nav应用程序的第6个子控制器使用分组表实现了一个可编辑的详细视图

9.3 构建 Nav 应用程序的骨架

Xcode为创建基于导航的应用程序提供了一个极好的模板，常用于创建分层应用程序。但是，我们没有使用这个模板。我们将从零开始构建基于导航的应用程序，以便于读者了解所有内容是如何组合起来的。这与第7章中构建标签栏控制器（tab bar controller）的方式没有多大区别，所以你能轻松地跟上我们的节奏。

在Xcode中，按下⌘N创建一个新的项目，然后从iPhone模板列表中选择Window-Based Application。将新项目命名为Nav。单击Classes和Resources文件夹，可以看到此模板只提供一个应用程序委托，即MainWindow.xib。我们需要向MainWindow.xib添加一个导航控制器，它将是应用程序的根控制器。由于所有的导航控制器必须拥有自己的根视图控制器，因此还需要创建它们。我们已经在Xcode中了，下面创建实现根视图控制器所需的文件。

9.3.1 创建根视图控制器

在项目窗口中，选择Groups & Files窗格中的Classes文件夹，然后按下⌘N或从File菜单中选择New File…。在出现新建文件向导时，选择Cocoa Touch Classes和UIViewController subclass，然后单击Next。将此文件命名为RootViewController.m，并确保选中了Also create “RootViewController.h”复选框。

刚才创建的这些文件将包含导航控制器的根视图控制器的控制器类，其中，根视图控制器将在应用程序运行时显示。那么现在我们需要创建一个nib文件，对吗？

不，在本章中，我们将子类化UITableViewController而不是UIViewController。如果子类化UITableViewController，它将创建一个表视图，而不需要nib文件。如果需要多个表，使用这种方法显然是不可行的，比如在上一章中添加搜索栏时就不能使用这个方法。不过，如果只需要一个表，则可以采用这种方法。

9.3.2 设置导航控制器

在编写代码之前，让我们了解一下组成应用程序的多个控制器的名称。应用程序的根控制器处于最高级，它的视图将被添加到窗口中。本应用程序的根控制器是换入和换出组成视图层次的所有其他视图的导航控制器。

你可能有些不解。事实是，导航控制器拥有一个名称为rootViewController的属性，它是栈中最底层视图控制器的控制器。在我们的应用程序中，它就是图9-2中的分为6行的视图。问题在于根视图控制器有一个rootViewController属性。为了避免弄混，这里不再像以前一样将根控制器命名为rootController，而是将它命名为navController。

花一些时间理解上面的所有内容，并整理好自己的思路。现在，回到正题。

在NavAppDelegate.h中，添加以下代码：

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

@class NavViewController;

@interface NavAppDelegate : NSObject <UIApplicationDelegate> {
    IBOutlet UIWindow *window;
    IBOutlet UINavigationController *navController;
}

@property (nonatomic, retain) UIWindow *window;
@property (nonatomic, retain) UINavigationController *navController;
@end
```

接下来，我们需要切换到实现文件，并为navController添加@synthesize语句。然后，将navController的视图添加为应用程序窗口的子视图，以便能将它显示给用户。单击NavAppDelegate.m，做如下更改：

```
#import "NavAppDelegate.h"

@implementation NavAppDelegate
```



```

@synthesize window;
@synthesize navController;
- (void)applicationDidFinishLaunching:(UIApplication *)application {

    [window addSubview: navController.view];
    [window makeKeyAndVisible];
}

- (void)dealloc {
    [navController release];
    [window release];
    [super dealloc];
}

@end

```

保存上面两个文件。下一步，创建一个导航控制器，将它连接到刚才声明的navController输出口，然后告知导航控制器使用什么作为它的根视图控制器。

在Interface Builder中双击MainWindow.xib，打开该文件。在库中找到Navigation Controller（参见图9-9），将它拖到nib的主窗口中，也就是标签为MainWindow.xib的窗口，而不是标签为Window的窗口。

按下Control键，并从Nav App Delegate图标拖到新的Navigation Controller图标，选择navController输出口。如果有viewController输出口的话，小心不要选中它，因为它不是你想要的，将它连接到导航控制器会导致运行时异常。

我们的工作已经接近尾声了，不过下一项任务有些棘手。我们需要告知导航控制器在哪儿能找到它的根视图控制器。最简单的方法是，使用窗口工具栏中间的View Mode按钮将nib的主窗口改为列表模式（参见图9-10）。

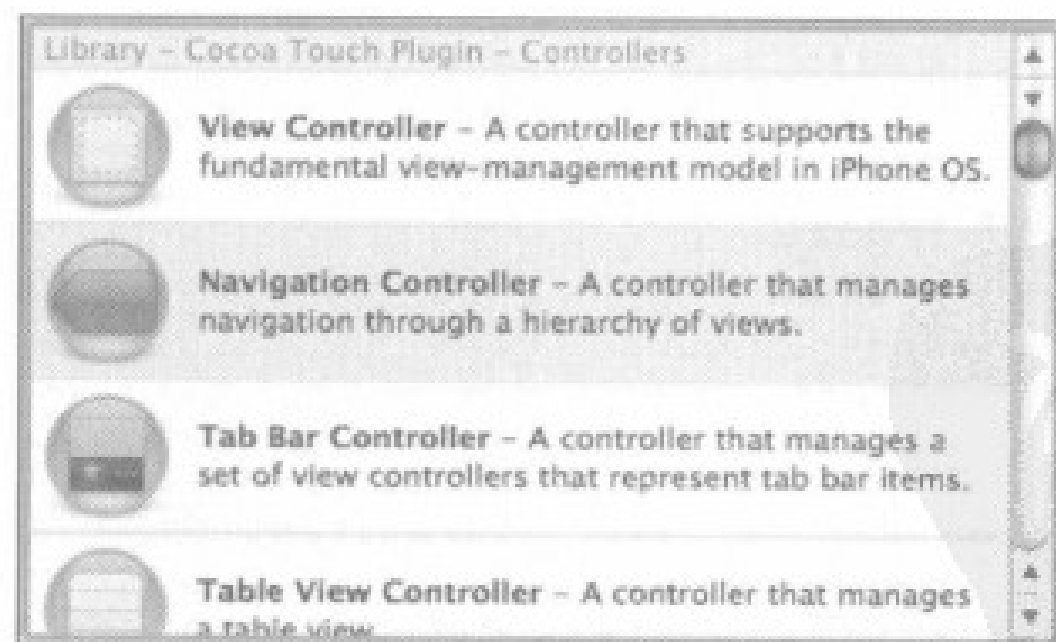


图9-9 库中的Navigation Controller

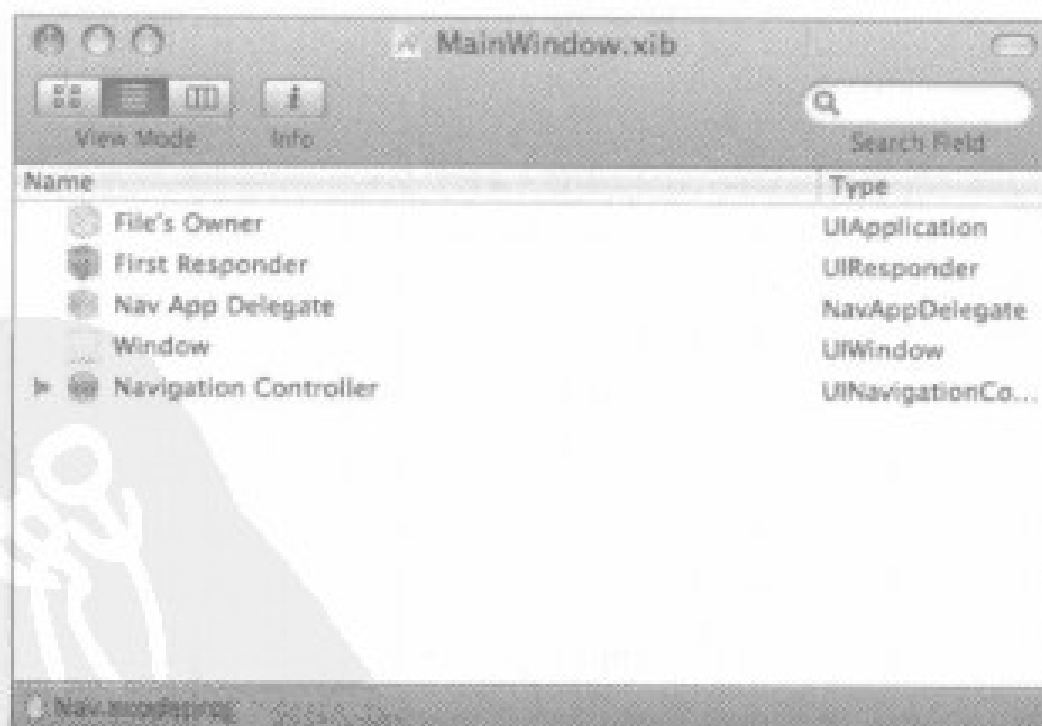


图9-10 将MainWindow.xib的主窗口改为列表模式

单击图9-10中Navigation Controller左侧的小三角形，展开它，可以看到两个项目：Navigation Bar和View Controller（Navigation Item）。

单击View Controller（Navigation Item）图标，按下⌘4打开身份检查器。将基础类更改为RootViewController，然后按下返回键提交更改。按下⌘1切换到属性检查器。此处，如果愿意，我们还可以指定一个nib文件，从中加载根级视图。不过，我们将保留NIB Name字段为空，以此

说明表视图控制器应该创建一个表视图实例。此处需要的所有更改已经完成了，下面保存、关闭窗口并返回到Xcode。

当然，现在我们需要一个用于显示根视图的列表。第8章使用了简单的字符串数组。在此应用程序中，根视图控制器将管理即将构建的子控制器列表。单击任何行都会导致所选视图控制器的实例被添加到导航控制器的栈中。我们还希望能够在每一行旁边显示一个图标，因此创建一个UITableViewController子类（因为它的UIImage属性可以存放行图标），而不是将UIImage属性添加到创建的每个子控制器中。然后，我们将子类化这个新的类，而不是直接子类化UITableViewController，结果是所有的子类都拥有UIImage属性，这会让代码更加简洁明了。

我们不会真正创建这个新类的实例。它是单独存在的，这样便可以向接下来编写的其他控制器添加常规项目。在许多语言中，可以把它声明为一个抽象类，不过Objective-C不支持抽象类。我们可以创建不需要实例化的类，但编译器不会阻止我们采用与其他许多语言相同的方式创建这些类。与其他多数流行语言相比，Objective-C是一种比较宽松的语言，你可能不太习惯这一点。

在Xcode中单击Classes文件夹，然后按下⌘N打开新建文件向导。从左侧窗格中选择Cocoa Touch Classes，然后选择NSObject subclass。将这个新文件命名为SecondLevelViewController。创建新文件之后，选择SecondLevelViewController.h，并做如下更改：

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

@interface SecondLevelViewController : NSObject {
@interface SecondLevelViewController : UITableViewController {
    UIImage *rowImage;
}
@property (nonatomic, retain) UIImage *rowImage;
@end
```

在SecondLevelViewController.m中添加以下代码行：

```
#import "SecondLevelViewController.h"

@implementation SecondLevelViewController
@synthesize rowImage;

@end
```

要作为二级控制器实现的任何控制器类（也就是说，用户可以从应用程序根级别直接导航到的任何控制器类）都应该子类化SecondLevelViewController，而不是UITableViewController。由于使用SecondLevelViewController作为父类，因此所有子类都将拥有一个可用于存储行图像的属性，我们可以先在RootViewController中编写代码，然后再实际编写具体的二级控制类。

下面开始写代码吧。首先，在RootViewCotroller.h中声明一个数组，并将父类改为UITableViewController：

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

@interface RootViewController : UIViewController
@interface RootViewController : UITableViewController
```

```
<UITableViewDelegate, UITableViewDataSource> {
    NSArray *controllers;
```

```
}
@property (nonatomic, retain) NSArray *controllers;
@end
```

我们刚才添加的数组将存放二级视图控制器的实例，并为表提供数据。即便使用 `UITableViewController` 作为父类，我们仍然需要遵循表视图委托和表视图数据源协议，此处也不例外。

在项目中添加新文件时，你会注意到一个名为 `UITableViewController subclass` 的文件模板。创建自己的应用程序时，请随意使用此选项。我们并未使用它，因为 `UITableViewController subclass` 模板提供了实现表时所需的大量方法和方法桩，而我们并不需要这些。我们不希望把时间浪费在处理这些不需要的模板方法上，而是让代码尽量保持简单。通过选择 `UIViewController` 并将它的超类改为 `UITableViewController`，我们得到了一个小得多的文件，这样更易于添加方法，而不会在一堆方法中摸不着头脑。

在 `RootViewController.m` 中添加以下代码。稍后，我们将具体解释它们。

```
#import "RootViewController.h"
#import "SecondLevelViewController.h"
#import "NavAppDelegate.h"

@implementation RootViewController
@synthesize controllers;

- (id)initWithStyle:(UITableViewStyle)style {
    if (self = [super initWithStyle:style]) {
    }
    return self;
}

- (void)viewDidLoad {
    self.title = @"Root Level";
    NSMutableArray *array = [[NSMutableArray alloc] init];
    self.controllers = array;
    [array release];
    [super viewDidLoad];
}

- (void)dealloc {
    [controllers release];
    [super dealloc];
}

#pragma mark -
#pragma mark Table Data Source Methods
- (NSInteger)tableView:(UITableView *)tableView
```

```

    numberOfRowsInSection:(NSInteger)section {
        return [self.controllers count];
    }
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *RootViewControllerCell= @"RootViewControllerCell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        RootViewControllerCell];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
            reuseIdentifier: RootViewControllerCell] autorelease];
    }
    // Configure the cell
    NSUInteger row = [indexPath row];
    SecondLevelViewController *controller =
        [controllers objectAtIndex:row];
    cell.text = controller.title;
    cell.image = controller.rowImage;
    return cell;
}

#pragma mark -
#pragma mark Table View Delegate Methods
- (UITableViewCellAccessoryType)tableView:(UITableView *)tableView
    accessoryTypeForRowWithIndexPath:(NSIndexPath *)indexPath
{
    return UITableViewCellAccessoryDisclosureIndicator;
}

- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    NSUInteger row = [indexPath row];
    SecondLevelViewController *nextController = [self.controllers
        objectAtIndex:row];

    NavAppDelegate *delegate =
        [[UIApplication sharedApplication] delegate];
    [delegate.navController pushViewController:nextController
        animated:YES];
}

@end

```

首先需要指出的是，这里导入了新的SecondLevelViewController.h头文件。这样，我们可以在代码中使用SecondLevelViewController类，以便编译器能知道rowImage属性。我们还导入了应用程序委托类的头文件，因为需要访问应用程序委托。应用程序委托拥有一个指向应用程序导航控制器的输出口，因此无论何时需要访问导航控制器，都可以通过应用程序委托来完成。

下面是viewDidLoad方法。我们所做的第一件事是设置self.title。通过询问当前活动控制器的标题，导航控制器知道要在导航栏的标题中显示什么。因此，为基于导航应用程序的所有控制器实例设置标题很重要，因为这能让用户了解自己处于哪个阶段。

然后创建一个可变数组，并把它分配给前面声明的controllers属性。稍后，当我们准备好向表中添加行时，将向此数组添加视图控制器，并且它们会自动显示。选择其中一行会自动将对应的视图呈现给用户。

viewDidLoad方法的最后一部分是对[super viewDidLoad]的调用。这是因为此处使用的父类是UITableViewController，而不是UIViewController。UIViewController的viewDidLoad方法为空，但UITableViewController的viewDidLoad方法可能不是。在覆盖viewDidLoad方法时始终调用[super viewDidLoad]是不错的主意，因为调用UIViewController的空方法对程序不会有任何影响。在子类化除UIViewController之外的任何视图控制器类时，确保在覆盖viewDidLoad时调用超类的方法是非常重要的，因为超类需要在加载视图时做一些事情。

tableView:numberOfRowsInSection:方法与以前所看到的相同，它只返回数组的计数。tableView:cellForRowAtIndexPath:与以前写的方法也类似，它获取一个可重用的单元，或者如果该单元不存在的话，就创建一个新的单元，然后从对应于查询行的数组中获取控制器对象。最后，将单元的文本设置为控制器的标题，并返回单元。

注意，此处我们假定从数组检索的对象是SecondLevelViewController的一个实例，并将控制器的rowImage属性分配给UIImage。稍后，在声明第一个具体的二级控制器并将其添加到数组时，此步骤将发挥作用。

添加的最后一个方法也是此处最重要的一个，它是新增的唯一功能。当然，你已经了解了tableView:didSelectRowAtIndexPath:方法。它是用户单击某行后调用的方法。如果需要在单击某行时触发信息展开，则可以使用此方法。首先，从indexPath中获取行：

```
NSInteger row = [indexPath row];
```

下一步，从对应于该行的数组中获取正确的控制器：

```
SecondLevelViewController *nextController =  
    [self.controllers objectAtIndex:row];
```

由于导航控制器是由应用程序委托来维护的，因此使用共享的UIApplication实例获取到该委托的引用。

```
NavAppDelegate *delegate = [[UIApplication sharedApplication] delegate];
```

下一步，我们使用委托的navController输出口（指向应用程序的导航控制器）将下一个控制器（从数组中取出的）放入到导航控制器栈中。

```
[delegate.navigationController pushViewController:nextController animated:YES];
```

这就是所有内容。层中的每个控制器只需要知道其子控制器。当选中一行时，活动的控制器负责获取或创建一个新的子控制器，如有必要，还会设置其属性（这里不需要设置），然后将新的子控制器加入到导航控制器栈中。完成这些操作之后，导航控制器就可以自动处理其他所有事

情了。

至此，应用程序的骨架已经完成了。你需要将Core Graphics骨架连接到项目中。如果不记得如何连接，请参见第5章中的详细步骤。

保存所有文件，构建并运行应用程序。确保输入的所有代码都是正确的。如果一切正常，应用程序将启动，并显示一个带有Root Level标题的导航栏。由于当前数组是空的，因此没有显示任何行（参见图9-11）。

现在，我们已经准备好开始开发二级视图了。在此之前，从09 Nav文件夹中取出图像图标。名称为Image的子文件夹中包含6个.png图像，我们可以使用它们作为行图像。在处理之前首先将这6个图像全部添加到Xcode项目的Resources文件夹中。

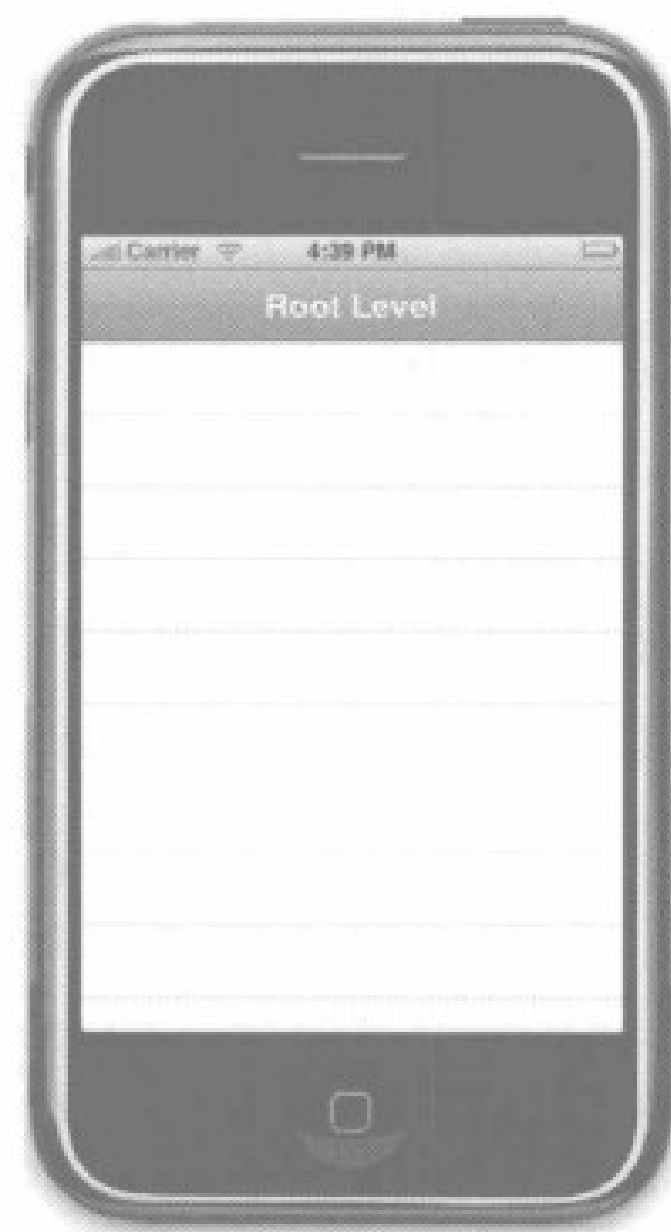


图9-11 正在运行的应用程序骨架

9.4 第1个子控制器：展示按钮视图

现在，实现第一个二级视图控制器。首先需要创建一个Second-LevelViewController子类。

在Xcode中选择Classes文件夹，按下⌘N再次打开新建文件向导。这次，从左侧窗格中选择Cocoa Touch Classes，从右上窗格中选择UIViewController subclass。将文件命名为DisclosureButton-Controller.m，并确保选中了创建头文件的复选框。在顶层视图中单击Disclosure Buttons时，此类用于管理将显示的影片表（参见图9-3）。

当用户单击任意影片标题时，应用程序将展开另一个视图，这个视图会报告选中了哪一行。因此，我们还需要创建一个可展开的详细视图，重复上面的步骤创建另一个文件，将它命名为DisclosureDetailController.m。确保选中了创建头文件的复选框。

详细视图是一个非常简单的视图，我们只能在这个视图中设置一个标签。它是不可编辑的，我们使用它展示如何将值传递到子控制器中。因为这个控制器不对表视图负责，所以还要为控制类创建一个nib文件。在创建nib之前，先为标签添加一个输出口。在DisclosureDetailController.h中，添加以下代码：

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

@interface DisclosureDetailController : UIViewController {
    IBOutlet UILabel *label;
    NSString *message;
}
@property (nonatomic, retain) UILabel *label;
@property (nonatomic, retain) NSString *message;
@end
```

为什么要同时添加一个标签和一个字符串呢？还记得延迟加载（lazy loading）的概念吗？没错，视图控制器将应用延迟加载。当我们创建自己的控制器时，它不会加载nib文件，直到nib文件被实际显示。当控制器被加入到导航控制器的栈中时，我们不能指望拥有要设置的label。如

果未加载nib文件，则label只是指向nil的一个指针。不过没关系，我们会将message设置为需要的值。并且，在viewWillAppear:方法中，将根据message中的值设置标签。

为什么此处使用viewWillAppear:进行更新，而不是像以前一样使用viewDidLoad方法呢？问题是viewDidLoad只在第一次加载其视图的时候得到调用。而在此处，我们重新使用了DisclosureDetailController的视图。不管怎样，当你单击展示按钮时，详细消息就出现在相同的DisclosureDetailController视图中。如果我们使用viewDidLoad来管理更新，该视图将只在DisclosureDetailController视图第一次出现的时候得到更新。在选取第二个Pixar按钮时，我们仍会看见来自第一个Pixar按钮的详细消息。这很不好，因为每次拖动视图时都调用viewWillAppear:方法，我们可以使用它进行更新。

在DisclosureDetailController.m中添加以下代码：

```
#import "DisclosureDetailController.h"

@implementation DisclosureDetailController
@synthesize label;
@synthesize message;
- (id)initWithNibName:(NSString *)nibNameOrNil
  bundle:(NSBundle *)nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil
      bundle:nibBundleOrNil]) {
      // Initialization code
    }
    return self;
}

- (void)viewWillAppear:(BOOL)animated {
    label.text = message;
    [super viewWillAppear:animated];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
  (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [label release];
    [message release];
    [super dealloc];
}
```



```
}
```

```
@end
```

这很简单，是吗？很好，下面为此文件创建nib。请确保保存了对源代码的更改。

在Xcode中选择Resources文件夹，然后按下⌘N创建另一个新文件。这一次，从左侧窗格中选择User Interfaces，从右上窗格中选择View XIB。将此nib文件命名为DisclosureDetail.xib。

首先设置nib。在Xcode中双击DisclosureDetail.xib，在Interface Builder中打开该文件。文件打开后，单击File's Owner，并按下⌘4打开身份检查器。将基础类改为DisclosureDetailController。现在，按下Control键并从File's Owner图标拖到View图标，选择view输出口重建控制器与视图之间的连接，因为这个连接在更改控制器类的时候中断了。

从库中拖出一个Label，并把它放置在View窗口上。调整大小，使它占据视图大部分宽度，使用蓝色引导线将它放到正确位置，然后使用属性检查器将文本对齐方式改为居中。从File's Owner拖到标签，然后选择label输出口。保存并关闭nib，返回Xcode。

在本例中，列表将只显示来自数组的多个行，因此我们要声明一个名称为list的NSArray。还需要声明一个实例变量，用它来存放子控制器的一个实例，它指向刚才构建的DisclosureDetailController类的一个实例。用户每次单击详细展示按钮时，我们都会为该控制器类分配一个新实例。不过创建一个实例然后重复使用的效率会更高。下面对DisclosureButtonController.h做如下更改：

```
#import <UIKit/UIKit.h>
#import "SecondLevelViewController.h"
@class DisclosureDetailController;

@interface DisclosureButtonController : UIViewController {
@interface DisclosureButtonController : SecondLevelViewController
    <UITableViewDelegate, UITableViewDataSource> {
        NSArray *list;
        DisclosureDetailController *childController;
    }
@property (nonatomic, retain) NSArray *list;
@end
```

注意，此处没有为childController声明属性。我们在类内部使用此实例变量，不希望把它呈现给其他人，因此不会通过声明属性来宣扬它的存在。

现在看看下面有趣的部分。对DisclosureButtonController.m做如下更改。稍后，我们将对它进行讨论。

```
#import "DisclosureButtonController.h"
#import "NavAppDelegate.h"
#import "DisclosureDetailController.h"

@implementation DisclosureButtonController
@synthesize list;
- (id)initWithStyle:(UITableViewStyle)style {
```

```
    if (self = [super initWithStyle:style]) {
        return self;
    }
- (void)dealloc {
    [list release];
    [childController release];
    [super dealloc];
}

- (void)viewDidLoad {
    NSArray *array = [[NSArray alloc] initWithObjects:@"Toy Story",
        @"A Bug's Life", @"Toy Story 2", @"Monsters, Inc.",
        @"Finding Nemo", @"The Incredibles", @"Cars",
        @"Ratatouille", @"WALL-E", @"Up", nil];
    self.list = array;
    [array release];
    [super viewDidLoad];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

#pragma mark -
#pragma mark Table Data Source Methods
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
    return [list count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString * DisclosureButtonCellIdentifier =
        @"DisclosureButtonCellIdentifier";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        DisclosureButtonCellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
            reuseIdentifier: DisclosureButtonCellIdentifier]
            autorelease];
    }

    NSUInteger row = [indexPath row];
    NSString *rowString = [list objectAtIndex:row];
    cell.text = rowString;
    [rowString release];
    return cell;
}
```

```

#pragma mark -
#pragma mark Table Delegate Methods
- (UITableViewCellAccessoryType)tableView:(UITableView *)tableView
    accessoryTypeForRowWithIndexPath:(NSIndexPath *)indexPath
{
    return UITableViewCellAccessoryDetailDisclosureButton;
}

- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:
        @"Hey, do you see the disclosure button?"
        message:@"If you're trying to drill down, touch that instead"
        delegate:nil
        cancelButtonTitle:@"Won't happen again"
        otherButtonTitles:nil];
    [alert show];
    [alert release];
}

- (void)tableView:(UITableView *)tableView
    accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath
{
    if (childController == nil)
        childController = [[DisclosureDetailController alloc]
            initWithNibName:@"DisclosureDetail" bundle:nil];

    childController.title = @"Disclosure Button Pressed";
    NSUInteger row = [indexPath row];

    NSString *selectedMovie = [list objectAtIndex:row];
    NSString *detailMessage = [[NSString alloc]
        initWithFormat:@"You pressed the disclosure button for %@.",
        selectedMovie];
    childController.message = detailMessage;
    childController.title = selectedMovie;
    [detailMessage release];
    NavAppDelegate *delegate =
        [[UIApplication sharedApplication] delegate];
    [delegate.navigationController pushViewController:childController
        animated:YES];
}
@end

```

到目前为止，你应该对每个问题都感到非常轻松，包括刚才添加的3个数据源方法。下面，让我们看一下这3个新的委托方法。

第一个是

。由于我们希望每一行都显示一个展示按钮，因此不管传入的参数是什么，此方法都返回UITableViewCellAccessoryDetailDisclosureButton。我们还使用了indexPath来查看这一行的数据，并只为可展开的行返回UITableViewCell-

`cellAccessoryDetailDisclosureButton`。对于那些不可展开的行，返回 `UITableViewCellAccessoryNone`，这将导致出现不带展示按钮的行。我们知道所有的行都可以展开，因此可以为每一行都返回相同的值。

第二个方法 `tableView:didSelectRowAtIndexPath:` 在选中一行时被调用，它会委婉地告诉用户要单击展示按钮而不是选中行。如果用户真的单击了细节展示按钮，则调用最后一个新增的委托方法 `tableView:accessoryButtonTappedForRowWithIndexPath:`。下面，让我们近距离看一下这个方法。

此方法所做的第一件事情是检查 `childController` 实例变量，查看它是否为 `nil`。如果是，则说明还没有分配和初始化 `DetailDisclosureController` 的新实例，接下来就会执行分配和初始化操作。这为我们提供了一个新的控制器，可以将它放入到导航栈中，就像前面在 `RootViewController` 中所做的一样。在将它放入栈中之前，需要为它分配所显示的文本。这样，我们将设置 `message` 以反应单击的是哪些行的展示按钮。我们还根据选中的行设置了新视图的标题。

现在，二级控制器已经完成了，它是我们的细节控制器。接下来的任务就是创建一个二级控制器实例，并将它添加到 `RootViewController` 的控制器中。单击 `RootViewController.m`，在 `viewDidLoad` 方法中添加以下代码：

```
- (void)viewDidLoad {
    self.title = @"Root Level";
    NSMutableArray *array = [[NSMutableArray alloc] init];

    // Disclosure Button
    DisclosureButtonController *disclosureButtonController =
        [[DisclosureButtonController alloc]
         initWithStyle:UITableViewStylePlain];
    disclosureButtonController.title = @"Disclosure Buttons";
    disclosureButtonController.rowImage = [UIImage
        imageNamed:@"disclosureButtonControllerIcon.png"];
    [array addObject:disclosureButtonController];
    [disclosureButtonController release];

    self.controllers = array;
    [array release];
    [super viewDidLoad];
}
```

上面的代码创建了一个新的 `DisclosureButtonController` 实例。通过指定 `UITableViewStylePlain`，我们表示需要一个索引表，而不是分组表。下一步，设置标题，并将图像设置为添加到项目中的一个 `.png` 文件，将控制器添加到数组，并释放控制器。在文件的顶部，需要添加一行代码为新文件导入标头类（header class）。在 `@implementation` 声明上面插入以下代码：

```
#import "DisclosureButtonController.h"
```

保存文件，然后构建项目。如果一切正常，你的项目将可以通过编译，并在仿真器中运行。运行完成之后，表中应该只有一行（参见图9-12）。

如果触摸这一行，将导航到刚才实现的表视图（参见图9-13）。

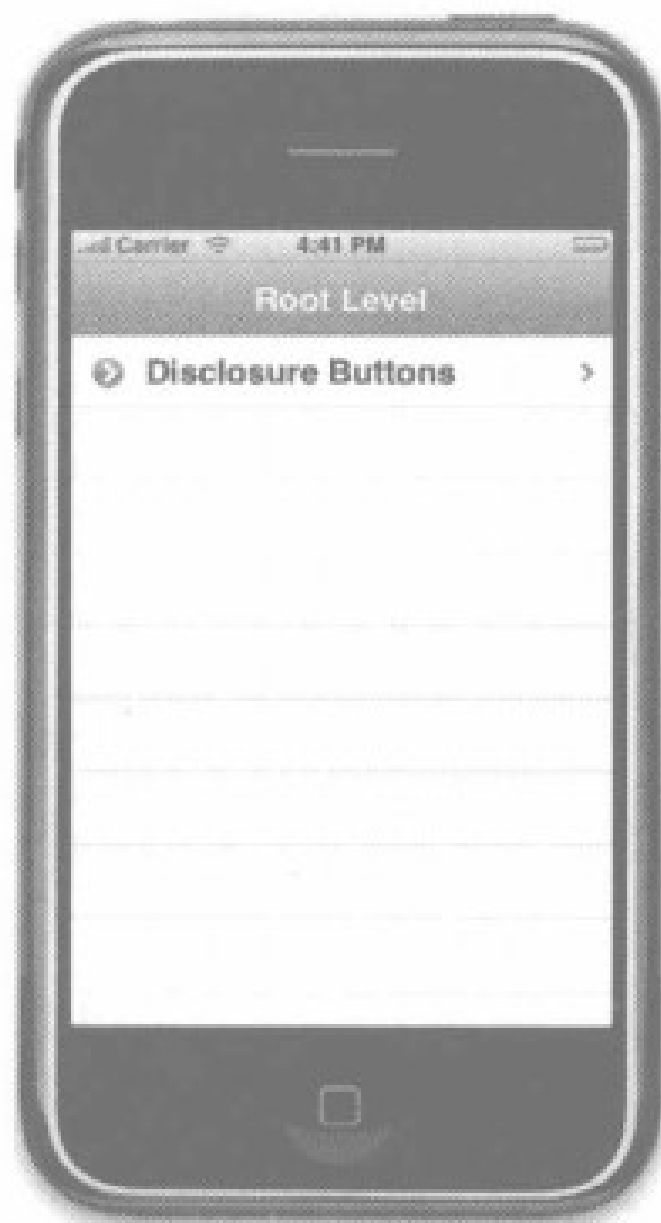


图9-12 添加了第一个二级控制器之后的应用程序

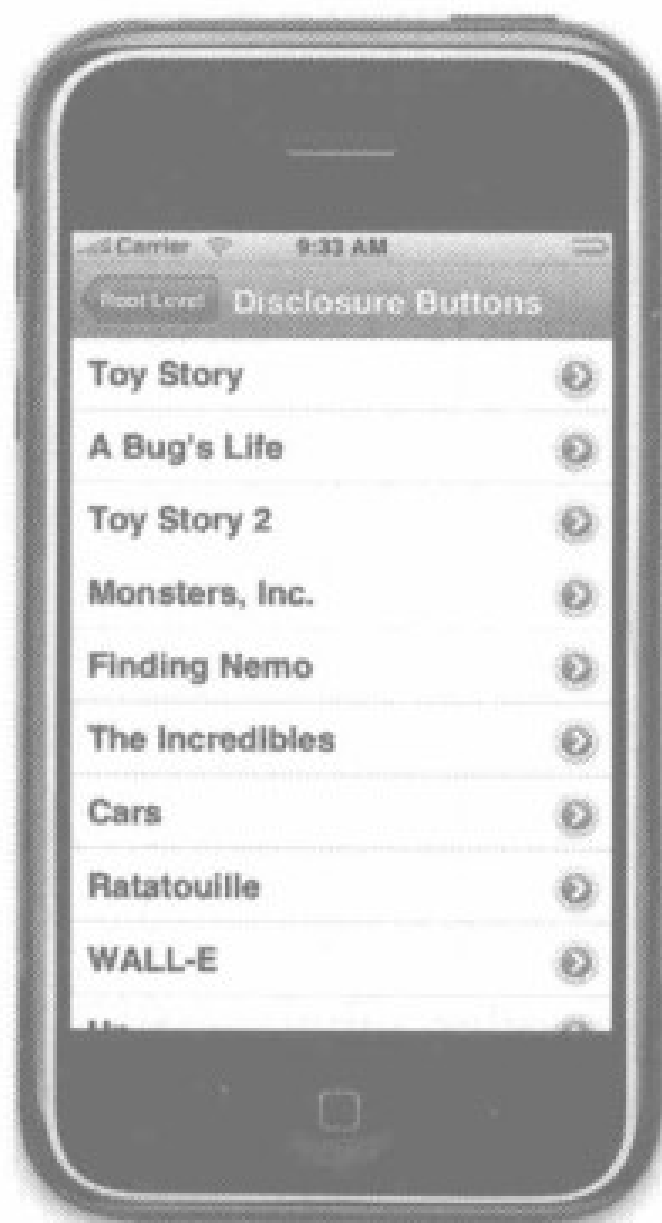


图9-13 展示按钮视图

注意，为控制器设置的标题现已显示在导航栏中，视图控制器的标题（Root Level）也包含在一个导航按钮中。单击该按钮将使用户返回到第一个级别。选中此表中的任意一行都会出现一个警告，提示你如果要展开视图，可以使用细节展示按钮（参见图9-14）。

触摸细节展示按钮会展开另一个视图。新视图（参见图9-15）将显示我们传递给它的信息。虽然这是一个简单的例子，但任何时候需要显示详细视图时都应使用这个技巧。

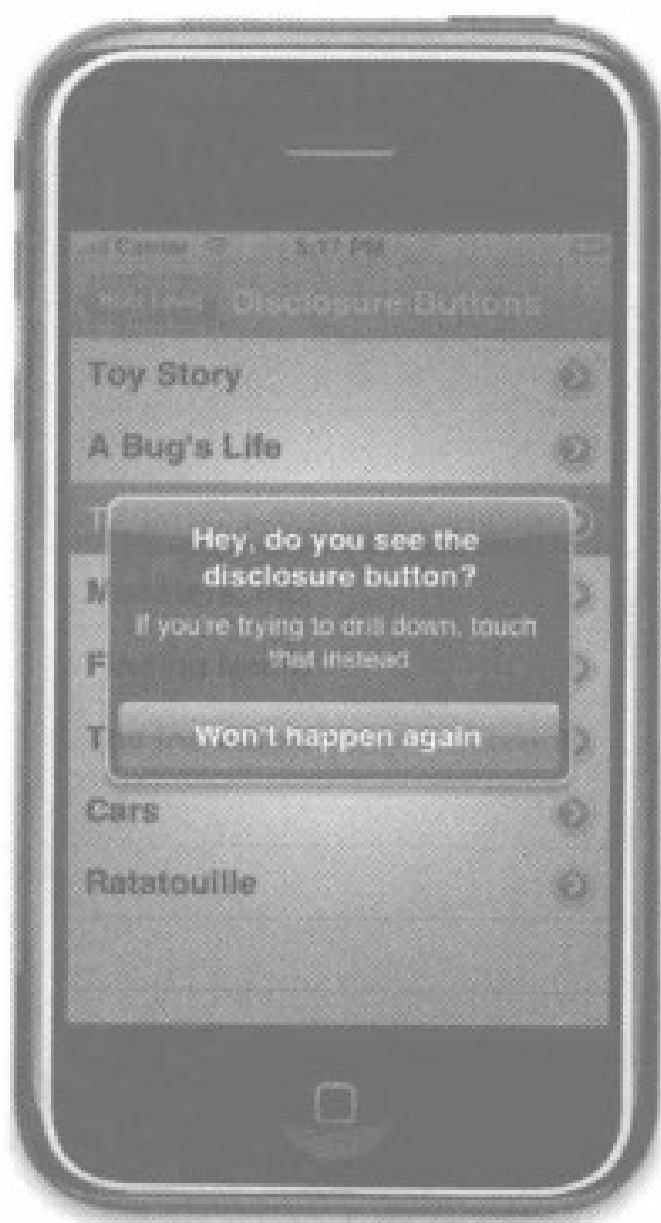


图9-14 当细节展示按钮可见时，选中行不会展开详细视图

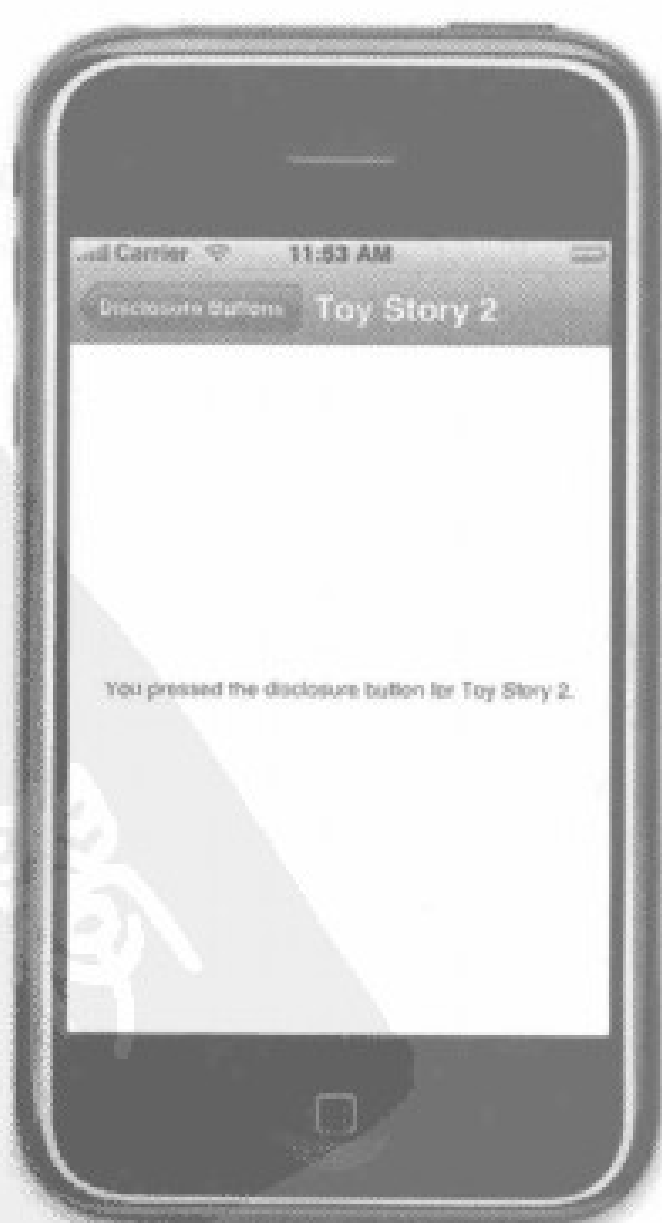


图9-15 详细视图

注意，当我们展开详细视图时，标题再次更改，返回按钮也是如此。现在单击返回按钮将回到上一个视图而不是根视图。这就完成了第一个视图控制器。现在，你已经了解了苹果公司设计导航控制器的方式，它使在小区块中构建应用程序成为可能。这看起来非常酷，不是吗？

9.5 第2个子控制器：校验表

我们将要实现的下一个二级视图也是一个表视图，不过这一次将使用扩展图标，以允许用户能且仅能从列表中选择个项目。我们将使用扩展图标在当前选中行的旁边放置一个选中标记，而且当用户单击另一行时，将更改选项。

由于这个视图是一个表视图，它没有任何详细视图，因此不需要创建新nib，不过我们确实需要创建另一个SecondLevelViewController子类。在Xcode的Groups & Files窗格中选择Classes文件夹，然后按下⌘N或从File菜单中选择New File...。选择Cocoa Touch Classes，并选择UIViewController subclass图标。单击Next按钮，当提示输入名称时，键入CheckListController.m，并确保同时也创建了头文件。

除了更改超类并遵循两个协议之外，我们还需要通过一个方法来跟踪当前所选中的行。下面将声明一个NSIndexPath属性来跟踪最后选中的行。单击CheckListController.h，并添加以下代码：

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

@interface CheckListController : UIViewController {
@interface CheckListController : SecondLevelViewController
    <UITableViewDelegate, UITableViewDataSource> {
        NSArray *list;
        NSIndexPath *lastIndexPath;
    }
@property (nonatomic, retain) NSIndexPath *lastIndexPath;
@property (nonatomic, retain) NSArray *list;
@end
```

现在，切换到CheckListController.m，并做如下更改：

```
#import "CheckListController.h"

@implementation CheckListController
@synthesize list;
@synthesize lastIndexPath;
- (id)initWithStyle:(UITableViewStyle)style {
    if (self = [super initWithStyle:style]) {

    }
    return self;
}

- (void)dealloc {
```

```

    [list release];
    [lastIndexPath release];
    [super dealloc];
}

- (void)viewDidLoad {

    NSArray *array = [[NSArray alloc] initWithObjects:@"Who Hash",
        @"Bubba Gump Shrimp Étouffée", @"Who Pudding", @"Scooby Snacks",
        @"Everlasting Gobstopper", @"Green Eggs and Ham", @"Soylent Green",
        @"Hard Tack", @"Lembas Bread", @"Roast Beast", @"Blancmange", nil];
    self.list = array;
    [array release];

    [super viewDidLoad];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}
#pragma mark -
#pragma mark Table Data Source Methods
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
    return [list count];
}
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CheckMarkCellIdentifier = @"CheckMarkCellIdentifier";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        CheckMarkCellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
            reuseIdentifier:CheckMarkCellIdentifier] autorelease];
    }
    NSUInteger row = [indexPath row];
    NSUInteger oldRow = [lastIndexPath row];
    cell.text = [list objectAtIndex:row];
    cell.accessoryType = (row == oldRow && lastIndexPath != nil) ?
        UITableViewCellAccessoryCheckmark : UITableViewCellAccessoryNone;

    return cell;
}

#pragma mark -

```



```

#pragma mark Table Delegate Methods
- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

    int newRow = [indexPath row];
    int oldRow = [lastIndexPath row];

    if (newRow != oldRow)
    {
        UITableViewCell *newCell = [tableView cellForRowAtIndexPath:
            indexPath];
        newCell.accessoryType = UITableViewCellAccessoryCheckmark;

        UITableViewCell *oldCell = [tableView cellForRowAtIndexPath:
            lastIndexPath];
        oldCell.accessoryType = UITableViewCellAccessoryNone;

        lastIndexPath = indexPath;
    }

    [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
@end

```

首先看一下tableView:cellForRowAtIndexPath:方法,因为这个方法中有一些值得注意的新问题。你应该对前面的几行很熟悉:

```

static NSString *CheckMarkCellIdentifier = @"CheckMarkCellIdentifier";

UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
    CheckMarkCellIdentifier];
if (cell == nil) {
    cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
        reuseIdentifier:CheckMarkCellIdentifier] autorelease];
}

```

不过,这正是有趣的地方。首先,我们从这个单元和当前选项中提取行:

```

NSUInteger row = [indexPath row];
NSUInteger oldRow = [lastIndexPath row];

```

我们从数组中获得这一行的值,并将它分配给单元的标题:

```

cell.text = [list objectAtIndex:row];
cell.text = rowTitle;

```

然后,根据两行是否相同,将扩展图标设置为显示检验标记或者不显示任何东西。换句话说,如果某行的表正在请求单元,而这行正好是当前选中的行,我们就将扩展图标设置为一个选中标记;否则,将它设置为不显示任何东西。注意,还需要检查lastIndexPath来确保它不为nil。这样做是因为值为nil的lastIndexPath不表示任何选项。但是,在nil对象上调用row方法将返回0,它是一个有效行,不过我们不希望在0行上放置一个检验标记,因为实际上没有任何选项。

```
cell.accessoryType = (row == oldRow && lastIndexPath != nil) ?
    UITableViewCellStyleAccessoryCheckmark :
    UITableViewCellStyleAccessoryNone;
```

然后，释放声明的字符串并返回单元。

```
[rowTitle release];
return cell;
```

现在跳转到最后一个方法。你之前看到过tableView:didSelectRowAtIndexPath:方法，不过这里有些新内容。我们不仅获取了刚才选中的行，还获取了上一次选中的行。

```
int newRow = [indexPath row];
int oldRow = [lastIndexPath row];
```

这样做是因为如果新的行和旧的行相同，就不做任何更改：

```
if (newRow != oldRow)
{
```

下一步，获取刚才选中的单元，并指定一个检验标记作为它的扩展图标：

```
UITableViewCell *newCell = [tableView
    cellForRowAtIndexPath:indexPath];
newCell.accessoryType = UITableViewCellStyleAccessoryCheckmark;
```

然后，获取上一次选中的单元，将它的扩展图标设置为无：

```
UITableViewCell *oldCell = [tableView cellForRowAtIndexPath:
    lastIndexPath];
oldCell.accessoryType = UITableViewCellStyleAccessoryNone;
```

之后，存储刚才在lastIndexPath中选中的索引路径，以便在下一次选中一行时使用：

```
lastIndexPath = indexPath;
}
```

完成之后，告知表视图取消选中刚才选中的行，因为我们不希望该行一直保持突出显示。我们已经用选中标记标记了该行，把它保留为蓝色将是很麻烦的一件事。

```
[tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

接下来，只需要添加此控制器的一个实例到RootViewController的controllers数组。我们通过向RootViewController.m中的viewDidLoad方法添加以下代码来实现此目的：

```
- (void)viewDidLoad {
    self.title = @"Root Level";
    NSMutableArray *array = [[NSMutableArray alloc] init];

    // Disclosure Button
    DisclosureButtonController *disclosureButtonController =
        [[DisclosureButtonController alloc]
            initWithStyle:UITableViewStylePlain];
    disclosureButtonController.title = @"Disclosure Buttons";
    disclosureButtonController.rowImage = [UIImage imageNamed:
        @"disclosureButtonControllerIcon.png"];
    [array addObject:disclosureButtonController];
    [disclosureButtonController release];

    // Check List
    CheckListController *checkListController = [[CheckListController alloc]
```

```

        initWithStyle:UITableViewStylePlain];
checkListController.title = @"Check One";
checkListController.rowImage = [UIImage imageNamed:
    @"checkmarkControllerIcon.png"];
[array addObject:checkListController];
[checkListController release];

self.controllers = array;
[array release];
[super viewDidLoad];
}

```

最后，还需要导入新的头文件，因此在所有其他#import语句之前添加以下代码：

```
#import "CheckListController.h"
```

你还等什么呢？保存所有代码，编译并运行。如果一切正常，应用程序将再次在仿真器中运行，这一次更加让人高兴。屏幕上将出现两行（参见图9-16）。

如果点击Check One，就会转到刚才实现的视图控制器，如图9-17所示。当它第一次出现时，没有被选择的行，也没有可见的选中标记。如果点击某行，就会出现一个选中标记。如果再点击不同的行，选中标记就会转到新行。

9.6 第3个子控制器：表行上的控件

上一章展示了如何向表视图添加子视图以自定义其外观，但是没有在内容视图中放置除标签之外的任何活动控件。这一次，我们试着在表视图单元中添加控件。在本例中，我们将向每一行添加一个开关，不过对大多数控件的操作方法基本相同。这一次将向扩展窗格添加控件，这就意味着当单击扩展窗格时，用户将更改开关的值。另外，在其他任何地方单击一行都将弹出一个警告，告知我们此行开关的状态是开还是关。此技巧将向你展示如何检索在表视图单元上使用的控件的值——这确实非常有用。

要在根视图的表中再添加一行，需要再创建一个控制器。你应该知道创建的步骤了：在Xcode中的Groups & Files窗格中选择Classes文件夹，然后按下⌘N，或从File菜单中选择New File...。选择Cocoa Touch Classes，并选择UIViewController subclass图标。当提示输入名称时，键入RowControlsController.m，并确保选中了创建头文件的复选框。就像最后一个选项一样，一个表视图可以完全实现此控制器，不需要任何nib文件。

单击RowControlsController.h，并添加以下代码：

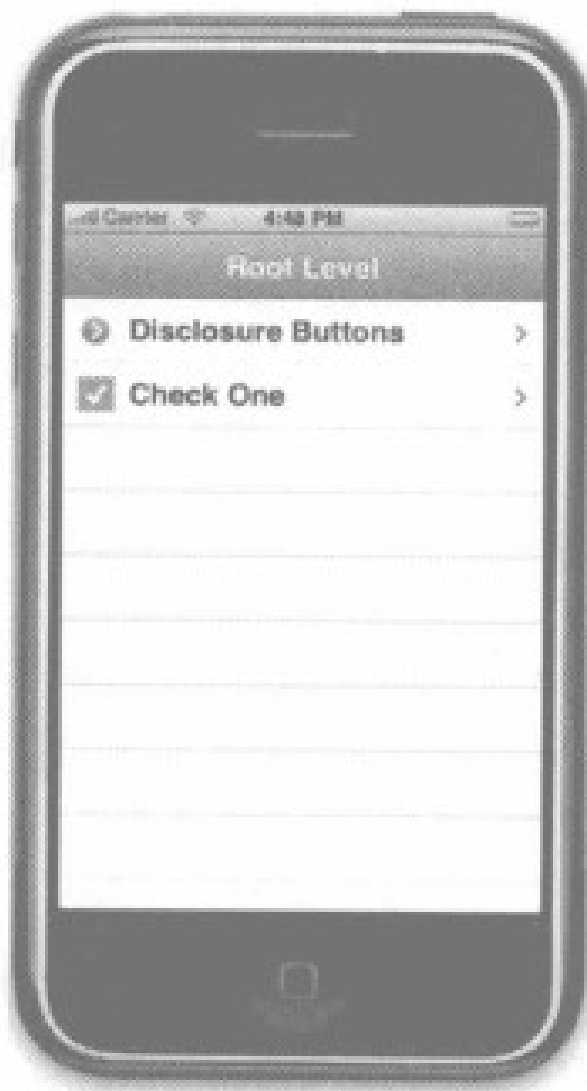


图9-16 两个二级控制器，两行信息

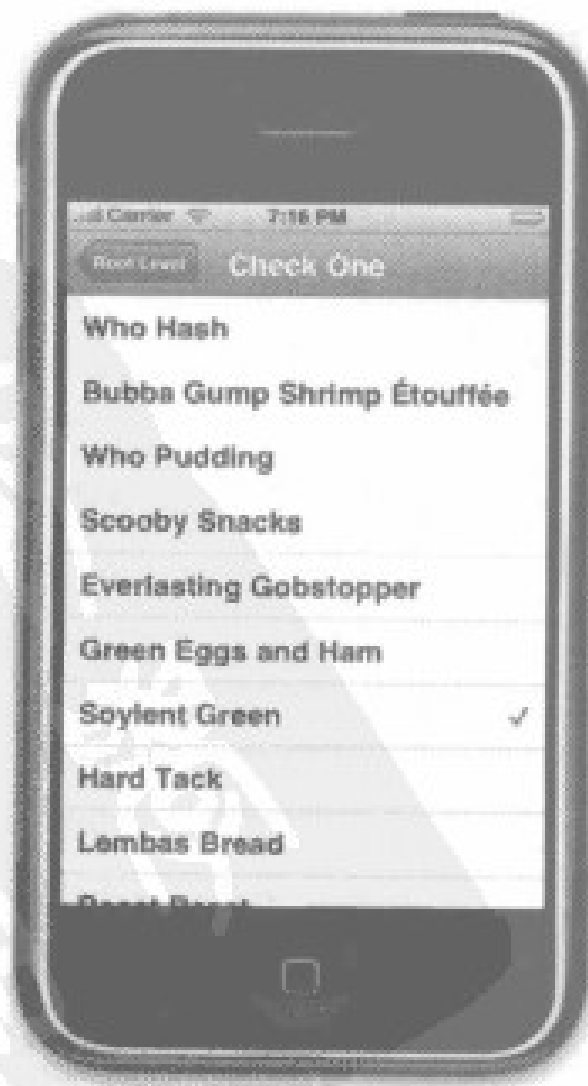


图9-17 检验表视图。注意，一次只能选中一个项目

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

#define kSwitchTag 100

@interface RowControlsController : UIViewController {
@interface RowControlsController : SecondLevelViewController
    <UITableViewDelegate, UITableViewDataSource> {
    NSArray *list;
}
@property (nonatomic, retain) NSArray *list;
@end
```

代码并不多，是吗？我们定义了一个常量，设置将被添加到表单元视图的开关的标记时将使用它。然后，我们可以使用该标记在代码的其他部分检索开关。此处更改了父类，使类遵循表数据源和表委托方法，并创建了一个数组来存放表数据。

切换到RowControlsController.m，并做如下更改：

```
#import "RowControlsController.h"

@implementation RowControlsController
@synthesize list;
- (id)initWithStyle:(UITableViewStyle)style {
    if (self = [super initWithStyle:style]) {
    }
    return self;
}

- (void)dealloc {
    [list release];
    [super dealloc];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

- (void)viewDidLoad {
    NSArray *array = [[NSArray alloc] initWithObjects:@"R2-D2",
        @"C3PO", @"Tik-Tok", @"Robby", @"Rosie", @"Uniblab",
        @"Bender", @"Marvin", @"Lt. Commander Data",
        @"Evil Brother Lore", @"Optimus Prime", @"Tobor", @"HAL",
        @"Orgasmatron", nil];
    self.list = array;
    [array release];
    [super viewDidLoad];
}

#pragma mark -
#pragma mark Table Data Source Methods
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
```

```

    return [list count];
}
- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *ControlRowIdentifier = @"ControlRowIdentifier";

    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:ControlRowIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
            reuseIdentifier:ControlRowIdentifier] autorelease];
        UISwitch *switchView = [[UISwitch alloc] init];
        switchView.tag = kSwitchTag;
        cell.accessoryView = switchView;
        [switchView release];
    }
    NSUInteger row = [indexPath row];
    NSString *rowTitle = [list objectAtIndex:row];
    cell.text = rowTitle;

    return cell;
}
#pragma mark -
#pragma mark Table Delegate Methods
- (void)tableView:(UITableView *)tableView
  didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSUInteger row = [indexPath row];
    UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexPath];
    UISwitch *switchView = (UISwitch *)[cell viewWithTag:kSwitchTag];

    NSString *baseString = @"%@ %@";
    NSString *onString = (switchView.on) ? @"IS on" : @"IS NOT on";
    NSString *robot = [list objectAtIndex:row];
    NSString *messageString = [[NSString alloc] initWithFormat:baseString,
        robot, onString];
    UIAlertView *alert = [[UIAlertView alloc]
        initWithTitle:@"Row Selected."
        message:messageString
        delegate:nil
        cancelButtonTitle:@"Thanks!"
        otherButtonTitles:nil];
    [alert show];
    [alert release];
    [messageString release];
}
@end

```

找到委托方法，查看tableView:cellForRowAtIndexPath:的实现，这也是我们设置表视图单

元显示控件的地方。此方法开头和往常一样，声明了一个标识符，然后使用它请求一个可重用的单元：

```
static NSString *ControlRowIdentifier = @"ControlRowIdentifier";
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
    ControlRowIdentifier];
```

如果没有可重用的单元，则创建一个：

```
if (cell == nil) {
    cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
        reuseIdentifier:ControlRowIdentifier] autorelease];
}
```

下面，创建一个开关，设置其标记，并将它分配给单元的accessoryView属性：

```
UISwitch *switchView = [[UISwitch alloc] init];
switchView.tag = kSwitchTag;
cell.accessoryView = switchView;
[switchView release];
}
```

现在，指出表视图的哪一行需要一个单元，根据其行号为它分配行标题，然后返回单元：

```
NSIndexPath *indexPath = [indexPath row];
NSString *rowTitle = [list objectAtIndex:indexPath.row];
cell.text = rowTitle;
```

```
return cell;
```

实现的最后一个方法是tableView:didSelectRowAtIndexPath:，正如你现在所知道的，它是一个委托方法，在用户选中一行后进行调用。这里我们所做的是查明哪一行被选中并获取其单元：

```
NSIndexPath *indexPath = [indexPath row];
UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexPath];
```

然后，使用以前指定的开关tag（标记）从单元中获取开关视图：

```
UISwitch *switchView = (UISwitch *)[cell viewWithTag:kSwitchTag];
```

我们应该指出，由于将这个值分配给了行的扩展视图，在这里还可以通过如下调用获得开关视图：

```
UISwitch *switchView = (UISwitch *)cell.accessoryView;
```

为什么不这样做呢？实际上，用上面的方式获取开关视图没有什么问题，只不过使用标记稍微安全一些。在后面的示例中，如果其他一些代码段为扩展视图分配了其他一些对象，则会造成在错误类型的对象上进行调用。这些bug很难调试，甚至会导致应用程序崩溃。另一方面，如果某个人用一个新的视图替换了扩展视图，viewWithTag:方法将返回nil。在Objective-C中，向nil传递消息是允许的，而且一般不会导致应用程序崩溃，虽然无任何意义，但的确是很好的精神食粮。在现实中，这两行代码之间的区别可能更多地取决于个人喜好。

获取指向开关的指针之后，使用其on属性来设置一个字符串并在警告中显示该字符串：

```

NSUInteger row = [indexPath row];
UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexPath];
UISwitch *switchView = (UISwitch *)[cell viewWithTag:kSwitchTag];

NSString *baseString = @"%@ %@.";
NSString *onString = (switchView.on) ? @"IS on" : @"IS NOT on";
NSString *robot = [list objectAtIndex:row];
NSString *messageString = [[NSString alloc] initWithFormat:baseString,
    robot, onString];

UIAlertView *alert = [[UIAlertView alloc]
    initWithTitle:@"Row Selected."
    message:messageString
    delegate:nil
    cancelButtonTitle:@"Thanks!"
    otherButtonTitles:nil];
[alert show];
[alert release];
[messageString release];

```

现在，将此控制器添加到RootViewController的数组中。单击RootViewController.m，并在viewDidLoad中添加以下代码：

```

- (void)viewDidLoad {
    self.title = @"Root Level";
    NSMutableArray *array = [[NSMutableArray alloc] init];

    // Disclosure Button
    DisclosureButtonController *disclosureButtonController =
        [[DisclosureButtonController alloc]
            initWithStyle:UITableViewStylePlain];
    disclosureButtonController.title = @"Disclosure Buttons";
    disclosureButtonController.rowImage = [UIImage
        imageNamed:@"disclosureButtonControllerIcon.png"];
    [array addObject:disclosureButtonController];
    [disclosureButtonController release];

    // Check List
    CheckListController *checkListController = [[CheckListController alloc]
        initWithStyle:UITableViewStylePlain];
    checkListController.title = @"Check One";
    checkListController.rowImage = [UIImage
        imageNamed:@"checkmarkControllerIcon.png"];
    [array addObject:checkListController];
    [checkListController release];

    // Table Row Controls
    RowControlsController *rowControlsController =
        [[RowControlsController alloc]

```



```

initWithStyle:UITableViewStylePlain];
rowControlsController.title = @"Row Controls";
rowControlsController.rowImage = [UIImage imageNamed:
@"rowControlsIcon.png"];
[array addObject:rowControlsController];
[rowControlsController release];

self.controllers = array;
[array release];
[super viewDidLoad];
}

```

要编译此代码，必须为RowControlsController类导入头文件。所以，在相同的文件中的@implementation语句之前添加以下代码行：

```
#import "RowControlsController.h"
```

保存并编译程序。这一次，假定一切正常，应用程序运行后将出现另一行（参见图9-18）。

如果单击这个新行，它将导航到一个新的列表，其中每一行的右侧都有一个开关控件。单击任意开关将显示其值（参见图9-19）。

单击一行中除开关外的任意位置将显示一个警告，告知该行开关的状态是开还是关。从这一点来说，你应该对于整个应用程序感到很轻松。那么让我们尝试一个稍微复杂一些的情况吧，好吗？下面看一下如何让用户对表中的行重新排序。

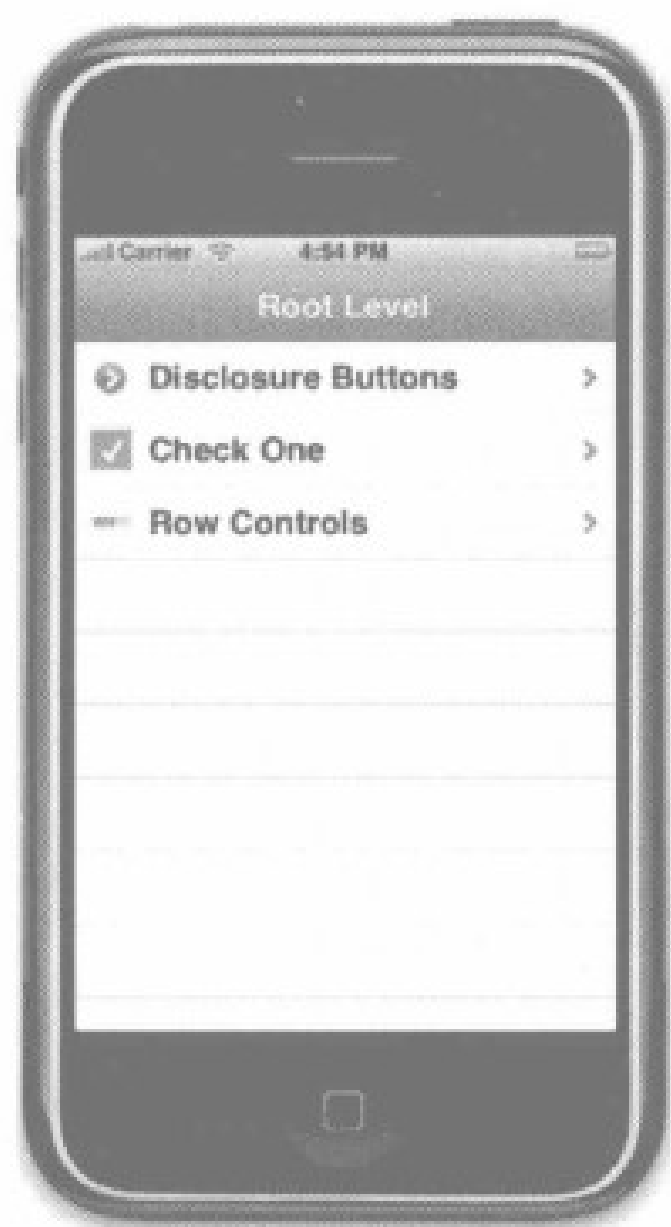


图9-18 添加到根级控制器的行控件控制器



图9-19 扩展视图中带有开关控件的表。单击开关，改变设备的作用

9.7 第4个子控制器：可移动的行

感觉怎么样？还在继续坚持吗？本章的内容很多，到现在你已经读完大部分了。为什么不休

息一下，听听音乐吃点零食呢？我们也一样。当你再次精神焕发并准备继续时，我们将构建另一个二级视图控制器，并将它添加到应用程序中。

9.7.1 编辑模式

移动并删除行，以及在表的指定位置插入行，所有这些任务都可以相当轻松地实现。可以通过使用表视图上的`setEditing:animated:`方法打开编辑模式（editing mode）来完成以上3个任务。此方法带有两个`Boolean`类型的参数。第一个参数指示编辑模式是打开还是关闭，第二个参数指示表是否进行动画转换。如果把编辑模式设置为当前状态（也就是说，当编辑模式开启时仍为开启，当编辑模式关闭时仍为关闭），则不管在第二个参数中指定了什么，都不会执行动画转换。

在下一个控制器中，我们将再次使用编辑模式，用户可以从表中删除行。允许对行重新排序是最简单的编辑模式任务，所以我们先来解决它。

开启编辑模式后，大量新的委托方法就开始发挥作用。表视图使用它们询问某一行是否可以被移除或编辑，并告知用户是否真正移除或编辑了特定行。这听起来有些复杂。让我们实际看一看吧。

9.7.2 创建一个新的二级控制器

由于我们不需要显示详细视图，因此实现的Move Me视图控制器可以不带nib文件，而只带有一个控制器类。在Xcode的Groups & Files窗格中选择Classes文件夹，然后按下⌘N或从File菜单中选择New File...。选择Cocoa Touch Classes，并选择UIViewController subclass图标。当提示输入名称时，键入MoveMeController.m。

在头文件中，需要做两件事情。首先，需要用一个可变数组来存放数据和跟踪行顺序。它必须是可变的，因为我们要在获取移除通知时才能够移除项目。还需要一个操作方法在编辑模式的开启和关闭之间切换。此操作方法将由随后创建的导航栏按钮调用。单击MoveMeController.h，并做如下更改：

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

@interface MoveMeController : UIViewController {
@interface MoveMeController : SecondLevelViewController
    <UITableViewDelegate, UITableViewDataSource> {
        NSMutableArray *list;
    }
@property (nonatomic, retain) NSMutableArray *list;
-(IBAction)toggleMove;
@end
```

切换到MoveMeController.m，并添加以下代码：

```
#import "MoveMeController.h"

@implementation MoveMeController
@synthesize list;
```

```

- (IBAction)toggleMove
{
    [self.tableView setEditing:!self.tableView.editing animated:YES];
}
- (id)initWithStyle:(UITableViewStyle)style {
    if (self = [super initWithStyle:style]) {
    }
    return self;
}

- (void)dealloc {
    [list release];
    [super dealloc];
}

- (void)viewDidLoad {
    NSMutableArray *array = [[NSMutableArray alloc] initWithObjects:
        @"Eeny", @"Meeny", @"Miney", @"Moe", @"Catch", @"A",
        @"Tiger", @"By", @"The", @"Toe", nil];
    self.list = array;
    [array release];

    UIBarButtonItem *moveButton = [[UIBarButtonItem alloc]
        initWithTitle:@"Move"
        style:UIBarButtonItemStyleBordered
        target:self
        action:@selector(toggleMove)];
    self.navigationItem.rightBarButtonItem = moveButton;
    [moveButton release];
    [super viewDidLoad];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

#pragma mark -
#pragma mark Table Data Source Methods
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSectionInSection:(NSInteger)section {
    return [list count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *MoveMeCellIdentifier = @"MoveMeCellIdentifier";

    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:MoveMeCellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
            reuseIdentifier:MoveMeCellIdentifier] autorelease];
        cell.showsReorderControl = YES;
    }
}

```

```

    }
    NSInteger row = [indexPath row];
    cell.text = [list objectAtIndex:row];

    return cell;
}
- (UITableViewCellEditingStyle)tableView:(UITableView *)tableView
    editingStyleForRowAtIndexPath:(NSIndexPath *)indexPath {
    return UITableViewCellEditingStyleNone;
}
- (BOOL)tableView:(UITableView *)tableView
    canMoveRowAtIndexPath:(NSIndexPath *)indexPath {
    return YES;
}
- (void)tableView:(UITableView *)tableView
    moveRowAtIndexPath:(NSIndexPath *)fromIndexPath
    toIndexPath:(NSIndexPath *)toIndexPath {
    NSInteger fromRow = [fromIndexPath row];
    NSInteger toRow = [toIndexPath row];

    id object = [[list objectAtIndex:fromRow] retain];
    [list removeObjectAtIndex:fromRow];
    [list insertObject:object atIndex:toRow];
    [object release];
}
@end

```

让我们逐步地进行介绍。添加的第一段代码是操作方法的实现：

```

- (IBAction)toggleMove
{
    [self.tableView setEditing:!self.tableView.editing animated:YES];
}

```

此处所做的就是切换编辑模式，很简单，是吗？

下一个方法是viewDidLoad。你对该方法的前一部分应该已经非常熟悉。它创建了一个已填充值的可变数组，以便表可以显示一些数据。后面的部分是新增内容。

```

UIBarButtonItem *moveButton = [[UIBarButtonItem alloc]
    initWithTitle:@"Move"
    style:UIBarButtonItemStyleBordered
    target:self
    action:@selector(toggleMove)];
self.navigationItem.rightBarButtonItem = moveButton;
[moveButton release];

```

此处创建了一个按钮栏项目，该按钮将被放置在导航栏上。将其标题设置为Move，并指定常量UIBarButtonItemStyleBordered表示需要一个简单的按钮。最后两个变量target和action，告知按钮被单击时应该做什么。通过传递self作为目标，并为其提供一个到toggleMove方法的选择器作为操作，我们告知按钮无论何时被单击时都调用toggleMove方法。因此，任何时候用户单

击按钮时，都将切换编辑模式。创建按钮之后，将它添加到导航栏的右侧，然后释放它。

现在，找到到刚才添加的`tableView:cellForRowAtIndexPath:`方法。你注意到这一行新的代码了吗？

```
cell.showsReorderControl = YES;
```

可以通过实现`tableView:accessoryTypeForRowWithIndexPath:`方法指定标准扩展图标。但是，重新排序控件不是标准扩展图标，它是一个特殊的例子，只在表处于编辑模式时才得到显示。要启动重新排序控件，必须在单元上设置一个属性。不过，需要注意的是，将此属性设置为YES不会真正显示重新排序控件，除非表进入编辑模式。此方法中的其他内容是以前看到过的。

下一个新方法虽然简短，但很重要。在表视图中，我们希望能够对行进行重新排序，不过不希望用户能够删除或插入行。因此，我们实现了`tableView:editingStyleForRowWithIndexPath:`方法。通过这个方法，表视图可以询问指定行是否可以被删除，或是否可以将新行插入到指定位置。通过为每一行返回`UITableViewEditingStyleNone`，我们表示不支持插入或删除任何行。

下面是`tableView:canMoveRowAtIndexPath:`方法。每一行都将调用此方法，可以通过它禁止移动指定的行。如果某行的此方法返回NO，那么该行将不显示重新排序控件，用户不能够从当前位置移动该行。此处允许对所有行进行重新排序，所以全部返回YES。

最后一个方法`tableView:moveRowAtIndexPath:fromIndexPath:`，是当用户移动一行时真正调用的方法。`tableView`旁边的两个参数都是`NSIndexPath`实例，它们指定被移动的行和行的新位置。表视图已经移动了表中的行，用户应该看到了正确的结果，不过我们需要更新数据模型以保持两者同步，并避免显示问题的出现。

首先，检索需要移动的行。然后检索行的新位置。

```
NSUInteger fromRow = [fromIndexPath row];
NSUInteger toRow = [toIndexPath row];
```

现在，我们需要从数组中移除指定的对象，并在新位置重新插入该对象。不过在此之前，我们需要检索一个指向将要移动的对象指针并保留该指针，以便从数组中移除对象时，该对象不会被释放。如果数组是保留我们将要移除对象的唯一对象（这里确实如此），从数组中移除选中的对象将使其保留数量变为0，这意味着它可能会消失。通过首先保留它，可以阻止这样的事情发生。

```
id object = [[list objectAtIndex:fromRow] retain];
[list removeObjectAtIndex:fromRow];
```

移除它之后，需要将它重新插入到指定的新位置：

```
[list insertObject:object atIndex:toRow];
```

最后，由于我们已经保留了它，因此还需要释放它以避免内存泄漏：

```
[object release];
```

很好，现在已经大功告成了。我们实现了一个允许对行重新排序的表。现在，只需要向`RootViewController`的控制器数组添加这个新类的一个实例。现在，你对此可能已经非常熟悉了，不过此处将再次复习这一过程。

在RootViewController.m文件中，通过在@implementation声明之前添加以下代码行导入新视图的头文件：

```
#import "MoveMeController.h"
```

下面，向同一个文件的viewDidLoad方法中添加以下代码：

```
- (void)viewDidLoad {
    self.title = @"Root Level";
    NSMutableArray *array = [[NSMutableArray alloc] init];

    // Disclosure Button
    DisclosureButtonController *disclosureButtonController =
        [[DisclosureButtonController alloc]
         initWithStyle:UITableViewStylePlain];
    disclosureButtonController.title = @"Disclosure Buttons";
    disclosureButtonController.rowImage = [UIImage
        imageNamed:@"disclosureButtonControllerIcon.png"];
    [array addObject:disclosureButtonController];
    [disclosureButtonController release];

    // Check List
    CheckListController *checkListController = [[CheckListController alloc]
        initWithStyle:UITableViewStylePlain];
    checkListController.title = @"Check One";
    checkListController.rowImage = [UIImage
        imageNamed:@"checkmarkControllerIcon.png"];
    [array addObject:checkListController];
    [checkListController release];

    // Table Row Controls
    RowControlsController *rowControlsController =
        [[RowControlsController alloc]
         initWithStyle:UITableViewStylePlain];
    rowControlsController.title = @"Row Controls";
    rowControlsController.rowImage = [UIImage imageNamed:
        @"rowControlsIcon.png"];
    [array addObject:rowControlsController];
    [rowControlsController release];

    // Move Me
    MoveMeController *moveMeController = [[MoveMeController alloc]
        initWithStyle:UITableViewStylePlain];
    moveMeController.title = @"Move Me";
    moveMeController.rowImage = [UIImage imageNamed:@"moveMeIcon.png"];
    [array addObject:moveMeController];
    [moveMeController release];

    self.controllers = array;
    [array release];
}
```

```

    [super viewDidLoad];
}

```

接下来，编译程序，看一下会发生什么。如果一切正常，在仿真器中运行应用程序后，根级表中将包含4行。如果单击名为Move Me的新行，程序将导航到一个行列表。如果要移动这些行，单击Move按钮，将出现重新排序控件（参见图9-20）。

如果单击重新排序控件并拖动它，行会按照你拖动的方式进行移动，如图9-6所示。根据自己的喜好移动行。行会很好地适应它的新位置。你甚至可以回到顶级视图，然后再返回，这些行将会留在原来你调整的位置。如果退出并重新回来，这些行将恢复到原来的位置，不过别担心，在后面几章中，我们将向你介绍如何保存和恢复数据。



图9-20 第一次展开视图时的Move Me视图控制器

说明 如果你发现触摸移动控件有些困难，不要惊慌。如果你非常小心地单击移动控件像素，应该能够体验到移动的好处。此处的问题是，你面对的是使用单像素热区域光标的仿真器。如果将应用程序下载到iPhone或iPod Touch上，你将使用自己硕大的手指（至少有几个像素宽吧？），触摸移动控件就不会有问题了。

你可能注意到了，本章有些像马拉松。如果你感到有些压抑或不知所措，可以适当休息一下。本章需要理解的内容很多，而且都很重要。大部分iPhone应用程序在某些方面来说都将使用表视图。当你准备好继续下去时，我们来看一看编辑模式的另一种用法。这一次，我们将允许用户删除行。

9.8 第5个子控制器：可删除的行

实际上，允许用户删除行比允许用户移动行复杂不了多少。下面让我们看一下其过程。这一次，我们并没有通过对象的硬编码列表来创建数组，而是加载属性列表文件，用于保存一些键入信息。你可以从本书随附的项目归档文件的09 Nav文件夹中获取名为computers.plist的文件，并将它添加到Xcode项目的Resources文件夹中。

在Xcode中的Groups & Files窗格中选择Classes文件夹，然后按下⌘N，或从File菜单中选择New File...。选择Cocoa Touch Classes，然后选择UIViewController subclass图标。当提示输入名称时，键入DeleteMeController.m。

首先编辑DeleteMeController.h文件。要做的更改看起来应该很熟悉，因为它们和上一个视图控制器中的更改几乎相同。更改如下：

```

#import <UIKit/UIKit.h>
#import "SecondLevelViewController.h"

```



```

@interface DeleteMeController : UIViewController {
@interface DeleteMeController : SecondLevelViewController
    <UITableViewDelegate, UITableViewDataSource> {
        NSMutableArray *list;
    }
@property (nonatomic, retain) NSMutableArray *list;
-(IBAction)toggleEdit:(id)sender;
@end

```

不奇怪，对吗？首先把超类从UIViewController改为SecondLevelViewController，并让类遵循两个协议，我们需要这两个协议充当表的数据源和委托。之后，声明一个存放数据的可变数组和一个切换编辑模式的操作方法。在应用程序的上一个版本中，我们使用编辑模式让用户对行重新排序。在这一版本中，编辑模式将允许用户删除行。如果喜欢，可以把两者结合到相同的表中。此处，我们将分开介绍这两个方面，以便于读者的理解。不过将删除和重新排序操作结合起来的确实很好。当表处于编辑模式时，支持重新排序的行将显示重新排序图标。单击行左侧的红色圆形图标时（如图9-7所示），将弹出Delete按钮，这会遮蔽重新排序图标，不过只是暂时的。

切换到DeleteMeController.m，并添加以下代码：

```

#import "DeleteMeController.h"

@implementation DeleteMeController
@synthesize list;
-(IBAction)toggleEdit:(id)sender {
    [self.tableView setEditing:!self.tableView.editing animated:YES];
}
#pragma mark -
- (id)initWithNibName:(NSString *)nibNameOrNil
    bundle:(NSBundle *)nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil
        bundle:nibBundleOrNil]) {
        // Initialization code
    }
    return self;
}
- (void)viewDidLoad {
    NSString *path = [[NSBundle mainBundle] pathForResource:@"computers"
        ofType:@"plist"];
    NSMutableArray *array = [[NSMutableArray alloc]
        initWithContentsOfFile:path];
    self.list = array;
    [array release];

    UIBarButtonItem *editButton = [[UIBarButtonItem alloc]
        initWithTitle:@"Delete"
        style:UIBarButtonItemStyleBordered
        target:self
        action:@selector(toggleEdit)];
    self.navigationItem.rightBarButtonItem = editButton;
}

```

```

        [editButton release];

        [super viewDidLoad];
    }
    - (BOOL)shouldAutorotateToInterfaceOrientation:
        (UIInterfaceOrientation)interfaceOrientation {
        // Return YES for supported orientations
        return (interfaceOrientation == UIInterfaceOrientationPortrait);
    }

    - (void)didReceiveMemoryWarning {
        [super didReceiveMemoryWarning];
        // Releases the view if it doesn't have a superview
        // Release anything that's not essential, such as cached data
    }

    - (void)dealloc {
        [list release];
        [super dealloc];
    }
#pragma mark -
#pragma mark Table Data Source Methods
    - (NSInteger)tableView:(UITableView *)tableView
        numberOfRowsInSection:(NSInteger)section {
        return [list count];
    }

    - (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath {

        static NSString *DeleteMeCellIdentifier = @"DeleteMeCellIdentifier";

        UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
            DeleteMeCellIdentifier];
        if (cell == nil) {
            cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
                reuseIdentifier>DeleteMeCellIdentifier] autorelease];
        }
        NSInteger row = [indexPath row];
        cell.text = [self.list objectAtIndex:row];
        return cell;
    }

#pragma mark -
#pragma mark Table Delegate Methods
    - (void)tableView:(UITableView *)tableView
        commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
        forRowAtIndexPath:(NSIndexPath *)indexPath {

        NSInteger row = [indexPath row];
        [self.list removeObjectAtIndex:row];
        [tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]

```

```

        withRowAnimation:UITableViewRowAnimationFade];
    }
@end

```

让我们看一看上面的代码。新的操作方法toggleEdit:和上一版本中的完全相同。如果当前编辑模式状态为关闭，它将开启编辑模式；反之亦然。viewDidLoad方法也和上一个视图控制器中的类似。唯一的区别是从属性列表中加载数组，而不是为数组提供字符串的硬编码列表。我们使用的属性列表包含一个由字符串组成的平面数组，该数组包含许多常用的计算机模型名称。这一次，我们还为编辑按钮指定了一个不同的名称Delete，这样用户就能明显地知道这个按钮的作用。

两个数据源方法没有添加任何新内容，不过类中的最后一个方法是以前没有见过的，因此我们着重看一下这段代码：

```

- (void)tableView:(UITableView *)tableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
    forRowAtIndexPath:(NSIndexPath *)indexPath {

```

当用户完成一项编辑（删除或插入）操作时，表视图将调用此方法。第一个参数指出在哪个表视图的行上进行编辑。第二个参数editingStyle是一个常量，它指示编辑的类型。目前，我们定义了3种编辑类型。其中一个是UITableViewCellEditingStyleNone，我们在最后一部分中使用它指示行不能被编辑。另外两个类型是UITableViewCellEditingStyleDelete（默认选项）和UITableViewCellEditingStyleInsert。UITableViewCellEditingStyleNone不会传递到此方法中，因为它用于表示这一行不允许被编辑。

本例忽略了此参数，因为行默认的编辑类型是删除类型，因此每一次调用此方法时，它都请求一项删除操作。可以使用此参数在一个表中同时允许插入和删除操作，不过在同一行中不允许。另一个编辑类型UITableViewCellEditingStyleInsert，通常用于需要让用户在列表中的一个指定位置插入行的情形。在顺序由系统来维护的列表中，比如一个按字母顺序排序的名称列表，用户通常会单击工具栏或导航栏来请求系统在详细视图中创建一个新的对象。只要用户指定好新的对象，系统就会把它放在合适的行。这里不再对插入的使用做介绍，不过插入的功能基本上和我们将要实现的删除操作相同。唯一的区别就是，需要创建一个新的对象并把它插入到指定位置，而不是从数据模型中删除指定的行。

最后一个参数indexPath，表示当前正在编辑哪一行。对于删除操作，此索引路径表示将要被删除的行。对于插入操作，它表示新行插入位置的索引。

在我们的方法中，首先在indexPath中检索正被编辑的行：

```
NSInteger row = [indexPath row];
```

然后，从前面创建的可变数组中删除该对象：

```
[self.list removeObjectAtIndex:row];
```

最后，通过指定常量UITableViewRowAnimationFade（它表示删除行时iPhone将使用的一种动画类型）通知表删除该行。除了它，还有其他几个选项可以让行消失。可以查看Xcode文档浏览器中的UITableViewRowAnimation，看一下还有什么其他可用的动画。


```

[moveMeController release];

// Delete Me
DeleteMeController *deleteMeController = [[DeleteMeController alloc]
initWithStyle:UITableViewStylePlain];
deleteMeController.title = @"Delete Me";
deleteMeController.rowImage = [UIImage imageNamed:@"deleteMeIcon.png"];
[array addObject:deleteMeController];
[deleteMeController release];

self.controllers = array;
[array release];
[super viewDidLoad];
}

```

保存并编译。仿真器出现后，根级视图总共有5行信息。如果选中新的Delete Me行，程序将呈现一个计算机模型列表（参见图9-21）。你现在拥有多少计算机模型呢？

注意，导航栏右侧也有一个按钮，它被标记为Delete。如果单击该按钮，表将进入编辑模式，如图9-22所示。



图9-21 第一次运行时的Delete Me视图



图9-22 处于编辑模式的Delete Me视图

每个可编辑的行旁边都有一个小图标，看上去类似“禁止进入”的路标。如果单击该图标，它会转向一边，并出现一个标记为Delete的按钮（参见图9-7）。单击该按钮将使用指定的动画类型删除底层模型和表中的相应行。

9.9 第6个子控制器：可编辑的详细窗格

我们正处于最后一个拐弯处，终点线已经在望，不过还有一定的距离。如果你还跟随着我们，

拍拍自己的背给自己打气。因为本章的内容冗长而复杂。

我们将要探讨的下一个主题是，如何实现一个可编辑的详细视图。你在浏览iPhone上的多种应用程序时会注意到，许多应用程序（包括Contacts应用程序）都把它们详细视图实现为一个分组表（参见图9-23）。

现在让我们看看这是如何做到的。开始之前，需要显示一些数据，我们需要的不仅仅是一个字符串列表。在前两章中，当我们需要更复杂的数据（比如第7章中的多行表或第6章中的ZIP代码选择器）时，都使用NSArray来保存一组已填充数据的NSDictionary实例。这种方式具有很大的灵活性，不过实现起来有些困难。而对于此表中的数据，我们将创建一个自定义Object-C数据对象来存放将在列表中显示的单个实例。

9.9.1 创建数据模型对象

应用程序所使用的属性列表包含有关美国总统的数据：每个总统的姓名、所属的政党、就职年份及卸任年份。下面将创建存放这些数据的类。

再次在Xcode中单击Classes文件夹选中它，然后按下⌘N打开新建文件向导。从左侧窗格中选择Cocoa Touch Classes，然后从右上窗格中选择NSObject subclass。将此类命名为President.m，并确保选中了创建头文件的复选框。

单击President.h，并做如下更改：

```
#define kPresidentNumberKey      @"President"
#define kPresidentNameKey       @"Name"
#define kPresidentFromKey       @"FromYear"
#define kPresidentToKey         @"ToYear"
#define kPresidentPartyKey      @"Party"

#import <Foundation/Foundation.h>

@interface President : NSObject <NSCoding> {
    int         number;
    NSString    *name;
    NSString    *fromYear;
    NSString    *toYear;
    NSString    *party;
}

@property int number;
@property (nonatomic, retain) NSString *name;
@property (nonatomic, retain) NSString *fromYear;
@property (nonatomic, retain) NSString *toYear;
@property (nonatomic, retain) NSString *party;
@end
```



图9-23 一个用于呈现可编辑表视图的分组表视图示例

从文件系统中读取字段时，这5个常量用于标识这些字段。通过让此类遵循NSCoding协议，可以将此对象写入文件，或者从文件中读取它。此头文件中添加的其余新内容用来实现存放数据所需的属性。切换到President.m，并做如下更改：

```
#import "President.h"

@implementation President
@synthesize number;
@synthesize name;
@synthesize fromYear;
@synthesize toYear;
@synthesize party;

-(void)dealloc{
    [name release];
    [fromYear release];
    [toYear release];
    [party release];
    [super dealloc];
}

#pragma mark -
#pragma mark NSCoding
- (void)encodeWithCoder:(NSCoder *)coder
{
    [coder encodeInt:self.number forKey:kPresidentNumberKey];
    [coder encodeObject:self.name forKey:kPresidentNameKey];
    [coder encodeObject:self.fromYear forKey:kPresidentFromKey];
    [coder encodeObject:self.toYear forKey:kPresidentToKey];
    [coder encodeObject:self.party forKey:kPresidentPartyKey];
}

- (id)initWithCoder:(NSCoder *)coder
{
    if (self = [super init])
    {
        self.number = [coder decodeIntForKey:kPresidentNumberKey];
        self.name = [coder decodeObjectForKey:kPresidentNameKey];
        self.fromYear = [coder decodeObjectForKey:kPresidentFromKey];
        self.toYear = [coder decodeObjectForKey:kPresidentToKey];
        self.party = [coder decodeObjectForKey:kPresidentPartyKey];
    }
    return self;
}

@end
```

不要过于担心encodeWithCoder:和initWithCoder:方法。我们稍后将更详细地介绍有关内容。你只需要知道这两个方法是NSCoding协议的一部分。encodeWithCoder:方法把对象编码为归档文件；initWithCoder:方法用于从归档文件创建新的对象。通过这两个方法，我们可以通过属性列表归档文件创建President对象。这个类中的其他内容应该不再需要解释了。

我们为你提供了一个属性列表文件，它包含所有美国总统的数据，可用于创建刚才编写的President对象的新实例。我们将在下一节中使用它，所以不必键入大量的数据。在项目归档文件的09 Nav文件夹中找到Presidents.plist文件，并将它添加到项目的Resources文件夹中。

下面，开始编写两个控制器类。

9.9.2 创建控制器

应用程序的这一部分需要两个新的控制器，一个用于显示被编辑的列表，另一个用于查看和编辑该列表中选中项目的详细信息。由于这两个视图控制器都以表为基础，因此不需要创建任何nib文件，不过我们需要两个独立的控制器类。现在，让我们开始创建这两个类的文件并实现它们。

在Xcode的Groups & Files窗格中选择Classes文件夹，然后按下⌘N或从File菜单中选择New File...。选择Cocoa Touch Classes，然后选择UIViewController subclass图标。提示输入名称时，键入PresidentsViewController.m，并确保同时创建了头文件。再次重复上述步骤，键入名称PresidentDetailController.m。

说明 你可能会感到疑惑，PresidentDetailController是单数（与PresidentsDetailController相反），因为它只处理与一个总统相关的详细信息。是的，关于这个小细节，你和我之间确实有些冲突，不过以后就会冰释前嫌。

首先创建显示总统列表的视图控制器。单击PresidentsViewController.h，并做如下更改：

```
#import <UIKit/UIKit.h>
#import "SecondLevelViewController.h"
@class PresidentDetailController;

@interface PresidentsViewController : UIViewController {
@interface PresidentsViewController : SecondLevelViewController
    <UITableViewDelegate, UITableViewDataSource> {
        NSMutableArray *list;
    }
@property (nonatomic, retain) NSMutableArray *list;
@end
```

对于你这样一个经验丰富的老手，这里没有什么新东西。

切换到PresidentsViewController.m，并做如下更改。完成更改后，我们将进行讨论：

```
#import "PresidentsViewController.h"
#import "PresidentDetailController.h"
#import "President.h"
#import "NavAppDelegate.h"

@implementation PresidentsViewController
@synthesize list;

- (id)initWithStyle:(UITableViewStyle)style {
    if (self = [super initWithStyle:style]) {
```

```
    }
    return self;
}
- (void)dealloc {
    [list release];
    [super dealloc];
}

- (void)viewDidLoad {
    NSString *path = [[NSBundle mainBundle] pathForResource:@"Presidents"
        ofType:@"plist"];

    NSData *data;
    NSKeyedUnarchiver *unarchiver;

    data = [[NSData alloc] initWithContentsOfFile:path];
    unarchiver = [[NSKeyedUnarchiver alloc] initWithReadingWithData:data];

    NSMutableArray *array = [unarchiver decodeObjectForKey:@"Presidents"];
    self.list = array;
    [unarchiver finishDecoding];
    [unarchiver release];
    [data release];

    [super viewDidLoad];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

#pragma mark -
#pragma mark Table Data Source Methods
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
    return [list count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *PresidentListCellIdentifier =
        @"PresidentListCellIdentifier";

    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:PresidentListCellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
            reuseIdentifier:PresidentListCellIdentifier]
            autorelease];
    }

    NSUInteger row = [indexPath row];
```

```

    President *thePres = [self.list objectAtIndex:row];
    cell.text = thePres.name;
    return cell;
}
#pragma mark -
#pragma mark Table Delegate Methods
- (void)tableView:(UITableView *)tableView
  didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

    NSUInteger row = [indexPath row];
    President *prez = [self.list objectAtIndex:row];

    PresidentDetailController *childController =
        [[PresidentDetailController alloc]
         initWithStyle:UITableViewStyleGrouped];

    childController.title = prez.name;
    childController.president = prez;

    UINavigationController *delegate =
        [[UIApplication sharedApplication] delegate];
    UINavigationController *navController = [delegate navigationController];
    [navController pushViewController:childController animated:YES];
    [childController release];
}
@end

```

刚才键入的大部分代码都是以前出现过的。仅viewDidLoad方法中有一项新内容，此处使用NSKeyedUnarchiver方法通过属性列表文件创建了一个填充有President类实例的数组。不必完全理解这些代码，你只要知道我们加载了一个填满President实例的数组就可以了。

首先，获取属性列表的路径：

```

NSString *path = [[NSBundle mainBundle] pathForResource:@"Presidents"
ofType:@"plist"];

```

下一步，声明一个数据对象和一个NSKeyedUnarchiver。该数据对象将临时存放未编码的归档文件；NSKeyedUnarchiver将用于实际存储归档文件中的对象：

```

NSData *data;
NSKeyedUnarchiver *unarchiver;

```

将属性列表加载到data中，然后使用data初始化unarchiver：

```

data = [[NSData alloc] initWithContentsOfFile:path];
unarchiver = [[NSKeyedUnarchiver alloc] initWithReadingWithData:data];

```

现在，从归档文件中解码数组数据。“Presidents”键与创建此归档文件的值相同：

```

NSMutableArray *array = [unarchiver decodeObjectForKey:@"Presidents"];

```

然后，把此解码数组分配给list属性、结束解码过程、清理内存并调用super：

```

self.list = array;
[unarchiver finishDecoding];

```

```
[unarchiver release];
[data release];

[super viewDidLoad];
```

上次创建详细视图之后，还有一处需要更改。这项更改在最后一个方法

中。创建Disclosure Button视图时，每次都会重用相同的子控制器，而只需更改它的值。当得到的nib文件带有输出口时，这种方式相对来说比较容易。当你使用表视图来实现详细视图时，第一次触发和后面触发的方法是不同的。另外，用于显示和更改数据的表单元也将被重用。如果你试图让它每次都以相同的方式表现，并确保能够跟踪所有的更改，那么这两个细节的结合意味着你的代码会非常非常复杂。因此，花费一些额外开销是值得的，即通过分配和释放新控制器对象来减小控制器类的复杂性。

让我们看一下这个细节控制器，它是新添加的内容。当用户单击PresidentsViewController表中的行以允许输入总统的有关数据时，这个新的控制器将被加入到导航栈中。下面将实现这个详细视图。

9.9.3 创建详细视图控制器

女士们先生们，请系好安全带，前方的路有些麻烦。

下一个控制器有些复杂，不过我们会顺利地排除障碍。请坐在你的座位上不要动。单击PresidentDetailController.h，并做如下更改：

```
#define kNumberOfEditableRows      4
#define kNameRowIndex              0
#define kFromYearRowIndex          1
#define kToYearRowIndex            2
#define kPartyIndex                3

#define kLabelTag                  4096
#import <UIKit/UIKit.h>
@class President;

@interface PresidentDetailController : UIViewController {
@interface PresidentDetailController : UITableViewController
    <UITableViewDelegate, UITableViewDataSource, UITextFieldDelegate> {
        President *president;
        NSArray *fieldLabels;
        NSMutableDictionary *tempValues;
        UITextField *textFieldBeingEdited;
    }
@property (nonatomic, retain) President *president;
@property (nonatomic, retain) NSArray *fieldLabels;
@property (nonatomic, retain) NSMutableDictionary *tempValues;
@property (nonatomic, retain) UITextField *textFieldBeingEdited;

- (IBAction)cancel:(id)sender;
```

```

- (IBAction)save:(id)sender;
- (IBAction)textFieldDone:(id)sender;
@end

```

究竟发生了什么事？这是全新的内容。在以前所有的表视图示例中，表中的每一行都对应数组中的一行。数组提供表所需要的所有数据。因此，例如，Pixar影片的表由一个字符串数组来控制，每个字符串都包含一个Pixar影片的标题。

前面的总统信息示例有两个不同的表。一个是总统的姓名列表，数组的每一行对应一个总统姓名。第二个表实现选中总统后显示的详细视图。由于这个表包含固定数量的字段，因此我们没有使用数组为此表提供数据，而是定义了一系列常量，我们将在表数据源方法中使用它们。这些常量定义了可编辑字段的数量，以及保存这些属性的行的索引值。

还有一个名称为kLabelTag的常量，我们将使用它从单元中检索UILabel，以便可以正确地设置行的标签。UITextField还应该有一个标记吧？通常是这样。不过我们需要将文本字段的tag属性用于另一个目的。在设置文本字段的值时，我们必须使用另一个稍微方便一些的机制来获取文本字段。如果这看起来很让人迷惑，请不要担心。实际编写代码的时候，一切都会变得很清楚。

你应该注意到了，该类此次遵循3个协议：表数据源和委托协议，以及一个新的协议UITextFieldDelegate。通过遵循这个新协议，当用户对某个文本字段做出更改时，我们会得到通知，以保存字段的值。此应用程序没有足够的行能让表上下滚动，不过在许多应用程序中，文本字段可以移出屏幕，而且可以被再次分配和再次使用。如果文本字段没有了，它存储的值也会相应地消失。因此用户做出更改时需要保存文本字段的值。

再往下，我们声明了一个指向President对象的指针。我们将实际使用此视图编辑这个对象，并根据选中的行在父控制器的tableView:didSelectRowAtIndexPath:中设置它。当用户单击Thomas Jefferson这一行时，PresidentsViewController将创建一个PresidentDetailController实例。然后，PresidentsViewController将该实例的president属性设置为代表Thomas Jefferson的对象，并把新创建的PresidentDetailController实例加入到导航栈中。

第二个实例变量fieldLabels是用于存放标签列表的数组。这些标签对应常量kNameRowIndex、kFromYearRowIndex、kToYearRowIndex和kPartyIndex。例如，kNameRowIndex被定义为0。那么显示总统姓名那一行的标签在fieldLabel数组中被存储在索引0的位置。你将在稍后的代码中看到其实际应用。

下面，我们定义了一个可变字典tempValues，用于存入用户更改后的字段值。我们不希望直接对president对象做出更改，因为如果用户选择了Cancel按钮，还需要让用户能够返回源数据。我们要做的是将更改后的任意值存储在新可变字典中tempValues。例如，如果用户编辑了Name:字段，然后单击Party:字段并开始编辑它，PresidentDetailController将在Name:字段编辑完成后得到通知，因为它是文本字段的委托。

当PresidentDetailController得到更改的通知后，它把这个新的值存储在字典中，并以属性名称作为键。在我们的示例中，使用键@"name"存储对Name:字段的更改。这样，不管用户进行保存还是取消，我们都拥有需要的数据。如果用户取消，我们只需丢弃此字典即可；如果用户进行保存，我们就把更改的值复制到president中。

下面是一个指向UITextField的指针，名称为textFieldBeingEdited。当用户在一个PresidentDetailController文本字段中单击时，textFieldBeingEdited被设置为指向该文本字段。为什么需要这个文本字段指针呢？因为这里存在一个有趣的时间问题，而 textFieldBeingEdited正是解决方法。

用户可以采取两种基本方法来结束对一个文本字段的编辑。第一种方法是，用户可以触摸另一个成为第一响应者的控件或文本字段。这样，正被编辑的文本字段就失去了第一响应者状态，且委托方法textFieldDidEndEditing:被调用。后面为PresidentDetailController.m输入代码时，你将看到textFieldDidEndEditing:。textFieldDidEndEditing:获取文本字段的新值并把它存储在tempValues中。

用户结束对一个文本字段编辑的第二种方法是通过单击Save或Cancel按钮。进行这样的操作时，将调用save:或cancel:操作方法。在这两个方法中，必须使PresidentDetailController视图出栈，因为保存和取消动作都会结束当前的编辑会话。这就出现了一个问题。save:和action:操作方法很难找到刚才编辑的文本字段并保存数据。

我们前面讨论的委托方法textFieldDidEndEditing:，确实可以访问文本字段，因为文本字段是作为一个参数传入的。这就是使用textFieldBeingEdited的原因所在。cancel:操作方法忽略了textFieldBeingEditing，因为用户不需要保存更改，所以可以丢掉它们。但是，save:方法关心这些更改，它需要通过一种方法来保存它们。

由于textFieldBeingEdited被用作一个指向当前正被编辑的文本字段的指针，save:方法使用该指针把文本字段中的值复制到tempValues中。现在，save:可以进行工作并使PresidentDetailController视图出栈了，这将把我们的总统列表放回栈的顶部。当视图出栈时，文本字段和它的值就丢失了。没关系，我们已经对它们进行了保存。

单击PresidentDetailController.m，并做如下更改：

```
#import "PresidentDetailController.h"
#import "President.h"
#import "NavAppDelegate.h"

@implementation PresidentDetailController
@synthesize president;
@synthesize fieldLabels;
@synthesize tempValues;
@synthesize textFieldBeingEdited;
#pragma mark -
-(IBAction)cancel:(id)sender{

    NavAppDelegate *delegate =
        [[UIApplication sharedApplication] delegate];
    [delegate.navigationController popViewControllerAnimated:YES];
}
-(IBAction)save:(id)sender
{
```



```
if (textFieldBeingEdited != nil)
{
    NSNumber *tagAsNum= [[NSNumber alloc]
                        initWithInt:textFieldBeingEdited.tag];
    [tempValues setObject:textFieldBeingEdited.text forKey: tagAsNum];
    [tagAsNum release];
}
for (NSNumber *key in [tempValues allKeys])
{
    switch ([key intValue]) {
        case kNameRowIndex:
            president.name = [tempValues objectForKey:key];
            break;
        case kFromYearRowIndex:
            president.fromYear = [tempValues objectForKey:key];
            break;
        case kToYearRowIndex:
            president.toYear = [tempValues objectForKey:key];
            break;
        case kPartyIndex:
            president.party = [tempValues objectForKey:key];
        default:
            break;
    }
}

NavAppDelegate *delegate =
    [[UIApplication sharedApplication] delegate];

UINavigationController *navController = [delegate navController];
[navController popViewControllerAnimated:YES];

NSArray *allControllers = navController.viewControllers;
UITableViewController *parent = [allControllers lastObject];
[parent.tableView reloadData];
}
-(IBAction)textFieldDone:(id)sender
{
    [sender resignFirstResponder];
}
#pragma mark -
- (void)viewDidLoad {

    NSArray *array = [[NSArray alloc] initWithObjects:@"Name:", @"From:",
        @"To:", @"Party:", nil];
    self.fieldLabels = array;
    [array release];

    UIBarButtonItem *cancelButton = [[UIBarButtonItem alloc]
```



```

        initWithTitle:@"Cancel"
        style:UIBarButtonItemStylePlain
        target:self
        action:@selector(cancel:));
self.navigationItem.leftBarButtonItem = cancelButton;
[cancelButton release];

UIBarButtonItem *saveButton = [[UIBarButtonItem alloc]
    initWithTitle:@"Save"
    style:UIBarButtonItemStyleDone
    target:self
    action:@selector(save:)];
self.navigationItem.rightBarButtonItem = saveButton;
[saveButton release];

NSMutableDictionary *dict = [[NSMutableDictionary alloc] init];
self.tempValues = dict;
[dict release];
[super viewDidLoad];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [textFieldBeingEdited release];
    [tempValues release];
    [president release];
    [fieldLabels release];

    [super dealloc];
}

#pragma mark -
#pragma mark Table Data Source Methods
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
    return kNumberOfEditableRows;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *PresidentCellIdentifier = @"PresidentCellIdentifier";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        PresidentCellIdentifier];
    if (cell == nil) {

```

```
cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
        reuseIdentifier:PresidentCellIdentifier] autorelease];
UILabel *label = [[UILabel alloc] initWithFrame:
    CGRectMake(10, 10, 75, 25)];
label.textAlignment = UITextAlignmentRight;
label.tag = kLabelTag;
label.font = [UIFont boldSystemFontOfSize:14];
[cell.contentView addSubview:label];
[label release];

UITextField *textField = [[UITextField alloc] initWithFrame:
    CGRectMake(90, 12, 200, 25)];
textField.clearsOnBeginEditing = NO;
[textField setDelegate:self];
textField.returnKeyType = UIReturnKeyDone;
[textField addTarget:self
    action:@selector(textFieldDone:)
    forControlEvents:UIControlEventEditingDidEndOnExit];
[cell.contentView addSubview:textField];
}
NSUInteger row = [indexPath row];

UILabel *label = (UILabel *)[cell viewWithTag:kLabelTag];
UITextField *textField = nil;
for (UIView *oneView in cell.contentView.subviews)
{
    if ([oneView isKindOfClass:[UITextField class]])
        textField = (UITextField *)oneView;
}
label.text = [fieldLabels objectAtIndex:row];
NSNumber *rowAsNum = [[NSNumber alloc] initWithInt:row];
switch (row) {
    case kNameRowIndex:
        if ([[tempValues allKeys] containsObject:rowAsNum])
            textField.text = [tempValues objectForKey:rowAsNum];
        else
            textField.text = president.name;
        break;
    case kFromYearRowIndex:
        if ([[tempValues allKeys] containsObject:rowAsNum])
            textField.text = [tempValues objectForKey:rowAsNum];
        else
            textField.text = president.fromYear;
        break;
    case kToYearRowIndex:
        if ([[tempValues allKeys] containsObject:rowAsNum])
            textField.text = [tempValues objectForKey:rowAsNum];
        else
            textField.text = president.toYear;
}
```

```

        break;
    case kPartyIndex:
        if ([[tempValues allKeys] containsObject:rowAsNum])
            textField.text = [tempValues objectForKey:rowAsNum];
        else
            textField.text = president.party;
    default:
        break;
}
if (textFieldBeingEdited == textField)
    textFieldBeingEdited = nil;

textField.tag = row;
[rowAsNum release];
return cell;
}
#pragma mark -
#pragma mark Table Delegate Methods
- (NSIndexPath *)tableView:(UITableView *)tableView
    willSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    return nil;
}
#pragma mark Text Field Delegate Methods
- (void)textFieldDidBeginEditing:(UITextField *)textField
{
    self.textFieldBeingEdited = textField;
}
- (void)textFieldDidEndEditing:(UITextField *)textField
{
    NSNumber *tagAsNum = [[NSNumber alloc] initWithInt:textField.tag];
    [tempValues setObject:textField.text forKey:tagAsNum];
    [tagAsNum release];
}
@end

```

第一个新方法是cancel:操作方法。用户单击Cancel按钮时会调用此方法。单击Cancel按钮时，当前视图将出栈，下一个视图就上升到栈的顶部。通常，这项任务由导航控制器处理，不过在稍后的代码中，我们将手动设置左边栏的按钮项目。可以通过获取对导航控制器的一个引用，来告知它使当前视图出栈。

```

- (IBAction)cancel:(id)sender {
    NavAppDelegate *delegate =
        [[UIApplication sharedApplication] delegate];
    [delegate.navigationController popViewControllerAnimated:YES];
}

```

下一个方法是save:，用户单击Save按钮时会调用此方法。单击Save按钮时，用户输入的值已经存储在tempValues字典中了，除非键盘仍然可见，并且光标仍然位于一个文本字段中。如果是那样的话，可能还没有把对该文本字段的一些更改放入到tempValues字典中。为了搞清楚这一

点，`save:`方法做的第一件事就是检查当前是否有正被编辑的文本字段。只要用户开始编辑文本字段，我们就把指向该文本字段的指针存储在`textFieldBeingEdited`中。如果`textFieldBeingEdited`不为`nil`，就获取它的值并放入`tempValues`中。

```
if (textFieldBeingEdited != nil)
{
    NSNumber *tfKey= [[NSNumber alloc] initWithInt:
        textFieldBeingEdited.tag];
    [tempValues setObject:textFieldBeingEdited.text forKey:tfKey];
    [tfKey release];
}
```

然后，通过快速枚举遍历字典中的所有键值，它使用行号作为键。我们不能在`NSDictionary`中存储`int`这样的数据类型，因此创建了基于行号的`NSNumber`对象，并使用它们。我们使用`intValue`把`key`表示的数字转换为`int`类型，然后使用前面定义的常量在该值上使用一个`switch`，并把来自`tempValues`数组的适当的值分配给`president`对象上指定的字段。

```
for (NSNumber *key in [tempValues allKeys])
{
    switch ([key intValue]) {
        case kNameRowIndex:
            president.name = [tempValues objectForKey:key];
            break;
        case kFromYearRowIndex:
            president.fromYear = [tempValues objectForKey:key];
            break;
        case kToYearRowIndex:
            president.toYear = [tempValues objectForKey:key];
            break;
        case kPartyIndex:
            president.party = [tempValues objectForKey:key];
        default:
            break;
    }
}
```

现在，已经更新了`president`对象，我们需要在视图层中上升一个级别。在详细视图上单击`Save`或`Done`按钮，用户会升到上一级，因此我们获取应用程序委托，并使用它的`rootController`输出口使自己离开导航栈，同时把用户传递回总统列表：

```
NavAppDelegate *delegate =
    [[UIApplication sharedApplication] delegate];
[delegate.navigationController popViewControllerAnimated:YES];
```

现在，我们还有另一个问题需要处理，即告知父视图表重新加载数据。因为用户可以编辑的其中一个字段是名称字段，它显示在`PresidentsViewController`表中，如果不让表重新加载其数据，那么它将继续显示原始值。

```
UINavigationController *navController = [delegate navigationController];
NSArray *allControllers = navController.viewControllers;
```

```
UITableViewController *parent = [allControllers lastObject];
[parent.tableView reloadData];
```

用户单击键盘上的Done按钮时会调用第3个操作方法。如果没有此方法,当用户单击Done后,键盘不会关闭。此方法在我们的应用程序中并不是十分必要,因为此处可被编辑的4行正好适合键盘上面的区域。也就是说,如果再添加一行或者未来的应用程序需要更多屏幕空间时,可以采用此方法。保持应用程序之间行为的一致性是一个好主意,尽管这对应用程序的功能无关紧要。

```
-(IBAction)textFieldDone:(id)sender
{
    [sender resignFirstResponder];
}
```

viewDidLoad方法还是和以前一样,没有任何令人惊奇的内容。创建字段名称数组,将把fieldLabels属性分配给它。

```
NSArray *array = [[NSArray alloc] initWithObjects:@"Name:",
    @"From:", @"To:", @"Party:", nil];
self.fieldLabels = array;
[array release];
```

下一步,创建两个按钮并将它们添加到导航栏。我们把Cancel按钮放置在左栏按钮项目的位置,自动取代导航按钮。我们把Save按钮放置在右边的位置,并把它指定为UIBarButtonItemStyleDone类型。这一类型专用于一种按钮:当用户对他们的更改很满意,并准备离开视图时单击的按钮。此类型的按钮是蓝色的,不是灰色的,而且通常带有Save或Done标签。

```
UIBarButtonItem *cancelButton = [[UIBarButtonItem alloc]
    initWithTitle:@"Cancel"
    style:UIBarButtonItemStylePlain
    target:self
    action:@selector(cancel:)];
self.navigationItem.leftBarButtonItem = cancelButton;
[cancelButton release];

UIBarButtonItem *saveButton = [[UIBarButtonItem alloc]
    initWithTitle:@"Save"
    style:UIBarButtonItemStyleDone
    target:self
    action:@selector(save:)];
self.navigationItem.rightBarButtonItem = saveButton;
[saveButton release];
```

最后,创建一个新的可变字典并将它分配给tempValues,更改后的值将存放在此处。如果直接对president对象进行更改,当用户单击Cancel按钮时,我们将很难回滚到源数据。

```
NSMutableDictionary *dict = [[NSMutableDictionary alloc] init];
self.tempValues = dict;
[dict release];
[super viewDidLoad];
```

我们可以跳过dealloc方法和第一个数据源方法,因为这两个方法中没有什么新内容。不过,

我们确实需要讨论一下tableView:cellForRowAtIndexPath:方法，因为这里有几个难点。方法的第一部分和前面编写的其他tableView:cellForRowAtIndexPath:方法非常类似。

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *PresidentCellIdentifier = @"PresidentCellIdentifier";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        PresidentCellIdentifier];
    if (cell == nil) {

        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
            reuseIdentifier:PresidentCellIdentifier] autorelease];
```

在创建一个新单元时，我们会创建一个标签，使它右对齐并为它分配一个标记，以便我们以后能够再次对它进行检索。下一步，把该标签添加到单元的contentView中，并释放它。这非常简单：

```
UILabel *label = [[UILabel alloc] initWithFrame:
    CGRectMake(10, 10, 75, 25)];
label.textAlignment = UITextAlignmentRight;
label.tag = kLabelTag;
label.font = [UIFont boldSystemFontOfSize:14];
[cell.contentView addSubview:label];
[label release];
```

之后，创建一个新的文本字段。用户在此字段中输入值。我们将它设置为：编辑时不清除当前值，这样我们就不会丢失现有数据。然后将self设置为文本字段的委托。通过将文本字段的委托设置为self，文本字段会告知我们，实现来自UITextFieldDelegate协议的适当的方法时发生的某些事件。稍后你会看到，在这个类中，我们实现了两个文本字段委托方法。当用户开始或结束编辑文本字段包含的文本时，这些方法将被所有行上的文本字段所调用。我们还设置了键盘的返回键类型（return key type），通过它指定键盘右下方的键的文本。默认值为Return，不过由于只有一行文本，我们希望返回键类型为Done，因此此处传递UIReturnKeyDone。

```
UITextField *textField = [[UITextField alloc] initWithFrame:
    CGRectMake(90, 12, 200, 25)];
textField.clearsOnBeginEditing = NO;
[textField setDelegate:self];
textField.returnKeyType = UIReturnKeyDone;
```

之后，我们告知文本字段针对Did End on Exit事件调用textFieldDone:方法。这样做的效果类似于：从Interface Builder的连接检查器中的Did End on Exit事件拖到File's Owner，并选择一个操作方法。因为我们没有nib文件，因此必须用编程的方法来实现，不过结果是一样的。

配置文本字段之后，将它添加到单元的内容视图中。不过，值得注意的是，我们在将它添加到该视图之前没有设置tag。

```
[textField addTarget:self
    action:@selector(textFieldDone:)]
```

```

        forControlEvents:UIControlEventEditingDidEndOnExit];
        [cell.contentView addSubview:textField];
    }

```

至此，我们知道已经得到了一个全新的单元或者一个重用的单元，但不知道是哪一个。要做的第一件事是指出这个单元将要表示哪一行：

```
NSInteger row = [indexPath row];
```

下一步，我们需要从此单元内部获取对标签和文本字段的引用。标签很简单，只要使用分配给它的标记从cell中检索就可以了：

```
UILabel *label = (UILabel *)[cell viewWithTag:kLabelTag];
```

然而，文本字段就不是那么简单了。因为我们需要通过标记告知文本字段委托，哪一个文本字段正在调用它们。我们将依赖于这样一个事实：只有一个文本字段是单元contentView的子视图。此处将通过快速枚举遍历所有的子视图，在找到文本字段时，就将它分配给前面声明的指针。循环完成之后，textField指针将指向此单元中仅有的一个文本字段。

```
UITextField *textField = nil;
```

```

for (UIView *oneView in cell.contentView.subviews)
{
    if ([oneView isKindOfClass:[UITextField class]])
        textField = (UITextField *)oneView;
}

```

既然有了指向标签和文本字段的指针，就可以根据此行表示的是来自president对象的哪个字段，来为它们分配正确的值。标签再次从fieldLabels数组中获取它的值：

```
label.text = [fieldLabels objectAtIndex:row];
```

为文本字段分配值并不是那么容易。首先必须检查tempValues字典中是否有对应于此行的值。如果有，则将它分配给文本字段。如果tempValues中没有任何对应的值，那么我们可以得知此字段未经过更改，因此将president中对应的值分配给它。

```

NSNumber *rowAsNum = [[NSNumber alloc] initWithInt:row];
switch (row) {
    case kNameRowIndex:
        if ([[tempValues allKeys] containsObject:rowAsNum])
            textField.text = [tempValues objectForKey:rowAsNum];
        else
            textField.text = president.name;
        break;
    case kFromYearRowIndex:
        if ([[tempValues allKeys] containsObject:rowAsNum])
            textField.text = [tempValues objectForKey:rowAsNum];
        else
            textField.text = president.fromYear;
        break;
    case kToYearRowIndex:

```



```

        if ([[tempValues allKeys] containsObject:rowAsNum])
            textField.text = [tempValues objectForKey:rowAsNum];
        else
            textField.text = president.toYear;
        break;
    case kPartyIndex:
        if ([[tempValues allKeys] containsObject:rowAsNum])
            textField.text = [tempValues objectForKey:rowAsNum];
        else
            textField.text = president.party;
    default:
        break;
}

```

如果正在使用的字段是当前正在编辑的字段，这就表示textFieldBeingEdited中存放的值不再有效，因此将textFieldBeingEdited设置为nil。如果文本字段没有被释放或重用，则调用文本字段委托方法，且正确的值已经存在于tempValues字典中。

```

if (textFieldBeingEdited == textField)
    textFieldBeingEdited = nil;

```

下一步，将文本字段的tag设置为它所表示的行，这样我们就能知道哪一个字段正在调用文本字段委托方法：

```
textField.tag = row;
```

最后，释放rowAsNum并返回cell：

```

    [rowAsNum release];
    return cell;
}

```

这里确实实现了一个表委托方法，即tableView:willSelectRowAtIndexPath:。记住，此方法将在行被选中时调用，并且允许禁止选中行。在此视图中，我们不希望将要显示的行被选中。我们需要知道用户选中了一行，以便可以在旁边放置一个检验标记，但是不希望该行真正突出显示。别担心。要使某一行的文本字段可编辑，不需要选中此行，此方法的作用仅仅是不要让行在选中之后呈突出显示。

```

- (NSIndexPath *)tableView:(UITableView *)tableView
    willSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    return nil;
}

```

现在剩下的就是两个文本字段委托方法了。我们实现的第一个方法是textFieldDidBeginEditing:，当文本字段成为第一响应者时调用此方法。因此，如果用户单击某字段并打开键盘，我们将获取通知。在这个方法中，我们在当前正被编辑的字段中存放了一个指针，以便能够获取单击Save按钮之前所做的最后更改。

```

(void)textFieldDidBeginEditing:(UITextField *)textField
{
    self.textFieldBeingEdited = textField;
}

```

当用户通过单击不同的文本字段或按下Done按钮来停止编辑当前文本字段时,或者当另一个字段成为第一响应者(例如,当用户导航回总统列表时就会发生)时,会调用最后一个方法。此处,我们将该字段的值保存在tempValues字典中,以便用户单击Save按钮以确认更改时拥有这些值。

```
- (void)textFieldDidEndEditing:(UITextField *)textField
{
    NSNumber *tagAsNum = [[NSNumber alloc] initWithInt:textField.tag];
    [tempValues setObject:textField.text forKey:tagAsNum];
    [tagAsNum release];
}
```

就这么简单。我们完成了这两个视图控制器,接下来要做的就是向顶级视图控制器添加此类的一个实例。现在你应该知道怎么做了。单击RootViewController.m。

首先,在@implementation声明之前添加以下代码行,从新的二级视图中导入头文件:

```
#import "PresidentsViewController.h"
```

然后,在viewDidLoad方法中添加以下代码:

```
- (void)viewDidLoad {
    self.title = @"Top Level";
    NSMutableArray *array = [[NSMutableArray alloc] init];

    // Disclosure Button
    DisclosureButtonController *disclosureButtonController =
        [[DisclosureButtonController alloc]
         initWithStyle:UITableViewStylePlain];
    disclosureButtonController.title = @"Disclosure Buttons";
    disclosureButtonController.rowImage = [UIImage
        imageNamed:@"disclosureButtonControllerIcon.png"];
    [array addObject:disclosureButtonController];
    [disclosureButtonController release];

    // Check List
    CheckListController *checkListController = [[CheckListController alloc]
        initWithStyle:UITableViewStylePlain];
    checkListController.title = @"Check One";
    checkListController.rowImage = [UIImage
        imageNamed:@"checkmarkControllerIcon.png"];
    [array addObject:checkListController];
    [checkListController release];

    // Table Row Controls
    RowControlsController *rowControlsController =
        [[RowControlsController alloc]
         initWithStyle:UITableViewStylePlain];
    rowControlsController.title = @"Row Controls";
    rowControlsController.rowImage =
        [UIImage imageNamed:@"rowControlsIcon.png"];
```

```

[array addObject:rowControlsController];
[rowControlsController release];

// Move Me
MoveMeController *moveMeController = [[MoveMeController alloc]
    initWithStyle:UITableViewStylePlain];
moveMeController.title = @"Move Me";
moveMeController.rowImage = [UIImage imageNamed:@"moveMeIcon.png"];
[array addObject:moveMeController];
[moveMeController release];

// Delete Me
DeleteMeController *deleteMeController = [[DeleteMeController alloc]
    initWithStyle:UITableViewStylePlain];
deleteMeController.title = @"Delete Me";
deleteMeController.rowImage = [UIImage imageNamed:@"deleteMeIcon.png"];
[array addObject:deleteMeController];
[deleteMeController release];

// President View/Edit
PresidentsViewController *presidentsViewController =
    [[PresidentsViewController alloc]
    initWithStyle:UITableViewStylePlain];
presidentsViewController.title = @"Detail Edit";
presidentsViewController.rowImage = [UIImage imageNamed:
    @"detailEditIcon.png"];
[array addObject:presidentsViewController];
[presidentsViewController release];
self.controllers = array;
[array release];
[super viewDidLoad];
}

```

保存，然后构建应用程序。如果一切正常，仿真器将运行，并显示第6行，也就是最后一行信息，如图9-2所示。单击此行将显示美国总统列表（参见图9-24）。

单击任何一行将显示刚才构建的详细视图（参见图9-8），你可以编辑值。如果单击键盘中的Done按钮，键盘就会关闭。单击可编辑的值，键盘会再次出现。做一些更改并单击Cancel，应用程序将返回总统列表。如果再次访问刚才取消更改的总统信息，更改将消失。另一方面，如果做出更改并单击Save按钮，这些更改将反映在父表中，当你返回到详细视图时，新的值将仍然存在。



图9-24 最后一个子控制器显示美国总统列表。单击其中一行将显示详细视图

9.10 更多内容

我们还需要添加一项内容，以使应用程序能够按照预期运行。在刚才构建的版本中，键盘加入了一个Done按钮，单击此按钮会让键盘关闭。如果视图上有用户可能需要的其他控件，这样做是合适的。然而，由于表视图上的每一行都是一个文本字段，我们需要采用一种稍微不同的解决方案。键盘应该拥有一个Return按钮而不是Done按钮。单击Return按钮时，用户将导航到下一行文本字段。

要完成上述过程，需要做的第一件事就是用Return按钮代替Done按钮。可以通过从PresidentDetailController.m中删除一行代码来实现此目的。在tableView:cellForRowAtIndexPath:方法中，删除以下代码行：

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *PresidentCellIdentifier = @"PresidentCellIdentifier";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        PresidentCellIdentifier];
    if (cell == nil) {

        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
            reuseIdentifier:PresidentCellIdentifier] autorelease];
        UILabel *label = [[UILabel alloc] initWithFrame:
            CGRectMake(10, 10, 75, 25)];
        label.textAlignment = UITextAlignmentRight;
        label.tag = kLabelTag;
        label.font = [UIFont boldSystemFontOfSize:14];
        [cell.contentView addSubview:label];
        [label release];

        UITextField *textField = [[UITextField alloc] initWithFrame:
            CGRectMake(90, 12, 200, 25)];
        textField.clearsOnBeginEditing = NO;
        [textField setDelegate:self];
        textField.returnKeyType = UIReturnKeyDone;
        [textField addTarget:self
            action:@selector(textFieldDone:)
            forControlEvents:UIControlEventEditingDidEndOnExit];
        [cell.contentView addSubview:textField];
    }
    NSUInteger row = [indexPath row];
    ...
}
```

下一个步骤并不是那么简单。在textFieldDone:方法中，不是简单地告知sender放弃第一响应者状态，而是通过某种方式指出下一个字段并让该字段成为第一响应者。用这个新版本代替当

前的textFieldDone:版本, 然后我们将讨论其运行原理的:

```
-(IBAction)textFieldDone:(id)sender
{
    UITableViewCell *cell =
        (UITableViewCell *)[[sender superview] superview];
    UITableView *table = (UITableView *)[cell superview];
    NSIndexPath *textFieldIndexPath = [table indexPathForCell:cell];
    NSUInteger row = [textFieldIndexPath row];
    row++;
    if (row >= kNumberOfEditableRows)
        row = 0;
    NSUInteger newIndex[] = {0, row};
    NSIndexPath *newPath = [[NSIndexPath alloc] initWithIndexes:newIndex
        length:2];
    UITableViewCell *nextCell = [self.tableView
        cellForRowAtIndexPath:newPath];
    UITextField *nextField = nil;
    for (UIView *oneView in nextCell.contentView.subviews) {
        if ([oneView isKindOfClass:[UITextField class]])
            nextField = (UITextField *)oneView;
    }
    [nextField becomeFirstResponder];
}
```

遗憾的是, 单元并不知道它们表示哪一行。但是, 表视图知道给定单元当前表示的是哪一行。因此, 我们获取对表视图单元的引用。我们知道触发此操作方法的文本字段是表单元视图的内容视图的一个子视图, 因此只需要获取sender的超视图的超视图。

如果上面的内容感到疑惑, 可以这样考虑。在本例中, sender是正在编辑的文本字段。sender的超视图是对文本字段及其标签进行分组的内容视图。sender超视图的超视图是包含该内容视图的单元。

```
UITableViewCell *cell = (UITableViewCell *)[(UIView *)sender superview]
    superview];
```

我们还需要访问单元的封闭表视图, 这很简单, 因为它是单元的超视图:

```
UITableView *table = (UITableView *)[cell superview];
```

然后向表询问: 该单元所表示的是哪一行。表的回答是NSIndexPath, 因此从中获取行:

```
NSIndexPath *textFieldIndexPath = [table indexPathForCell:cell];
NSUInteger row = [textFieldIndexPath row];
```

下一步, 使row增加1, 这表示表中的下一行。如果新行号大于最后一行的行号, 则将row重置为0:

```
row++;
if (row >= kNumberOfEditableRows)
    row = 0;
```

然后，构建一个新的`NSIndexPath`来表示下一行，并使用该索引路径，获取对当前表示下一行的单元的引用：

```
NSUInteger newIndex[] = {0, row};
NSIndexPath *newPath = [[NSIndexPath alloc] initWithIndexes:newIndex
length:2];
UITableViewCell *nextCell = [self.tableView cellForRowAtIndexPath:newPath];
```

对于文本字段来说，`tag`已经用于另一个用途，因此我们需要遍历单元内容视图的子视图来查找文本字段，而不是使用`tag`来检索：

```
UITextField *nextField = nil;
for (UIView *oneView in nextCell.contentView.subviews) {
    if ([oneView isKindOfClass:[UITextField class]])
        nextField = (UITextField *)oneView;
}
```

最后，通知新的文本字段成为第一响应者：

```
[nextField becomeFirstResponder];
```

现在，编译并运行应用程序。你展开详细视图之后，单击`Return`按钮会导航到表中的下一个字段，这样更便于用户输入数据。

9.11 小结

本章就像是一场马拉松比赛，如果你还没累趴下，应该会感到非常惬意。仔细研究这些神秘的表视图和导航控制器对象很重要，因为它们是所有大型iPhone应用程序的支柱，但如果没有真正理解它们，其复杂性一定会给你带来许多麻烦。

开始构建自己的表时，请参考本章和上一章的内容，以及苹果公司的文档。表视图极其复杂，我们不可能涵盖所有内容，不过你在设计和构建自己的应用程序时，可以使用本章提供的表视图构建块。与往常一样，你可以随意在应用程序中重用这些代码。它是我们送给你的礼物。好好享受吧！

下一章将介绍应用程序设置，也就是iPhone用于收集和存储用户喜好的机制。等你完全冷静下来之后，多喝点水，然后继续下一章的学习。还有，不要忘记伸个懒腰。

应用程序设置和用户默认设置

现在，甚至最简单的计算机程序都包含首选项窗口，用户可以在其中设置特定的应用选项。在Mac OS X中，Preferences...菜单项通常包含在应用程序菜单中，选择该菜单项会弹出一个窗口，用户可以在其中输入和更改各种选项。iPhone有一个专用的Settings应用程序，你一定已经使用过多次了。本章将介绍如何在Settings应用程序中添加设置，以及如何从应用程序内部访问这些设置。

说明 苹果公司关于Settings的术语及其底层机制，可以在Application Preferences中找到。

10.1 了解设置束

通过Settings应用程序（参见图10-1），用户可以输入和更改任何带有设置束（settings bundle）的应用程序的首选项。设置束是构建到应用程序中的一组文件，它告诉Settings应用程序，主应用程序希望从用户那里收集到哪些首选项。

在iPhone或iPod Touch上，Settings图标位于主屏幕上。点击此图标将启动Settings应用程序，如图10-2所示。

在iPhone的用户默认设置机制下，Settings应用程序充当着一个通用的用户界面。用户默认设置是应用程序首选项的一部分，由NSUserDefaults类实现，用于保存和获取首选项。如果在Mac上开发过Cocoa程序，那么你可能很熟悉NSUserDefaults，因为在Mac上就是使用这个类保存和读取首选项的。与使用键从NSDictionary获取数据一样，应用程序通过NSUserDefaults使用键值读取和保存首选项数据。不同之处在于NSUserDefaults数据被持久化到文件系统中，而没有存储在内存中的对象实例中。

本章将创建一个应用程序，添加并配置一个设置束，然后从应用程序中访问并编辑这些首选项。

Settings应用程序的优势之一是无需为首选项设计用户界面。创建属性列表来定义应用程序的可用设置后，Settings应用程序会自动创建用户界面。但是，使用Settings应用程序也有一些限

制。当应用程序正在运行时，用户可能需要更改的任何首选项都不应该受到Settings应用程序的限制，因为用户可能被强制退出应用程序以更改这些值。

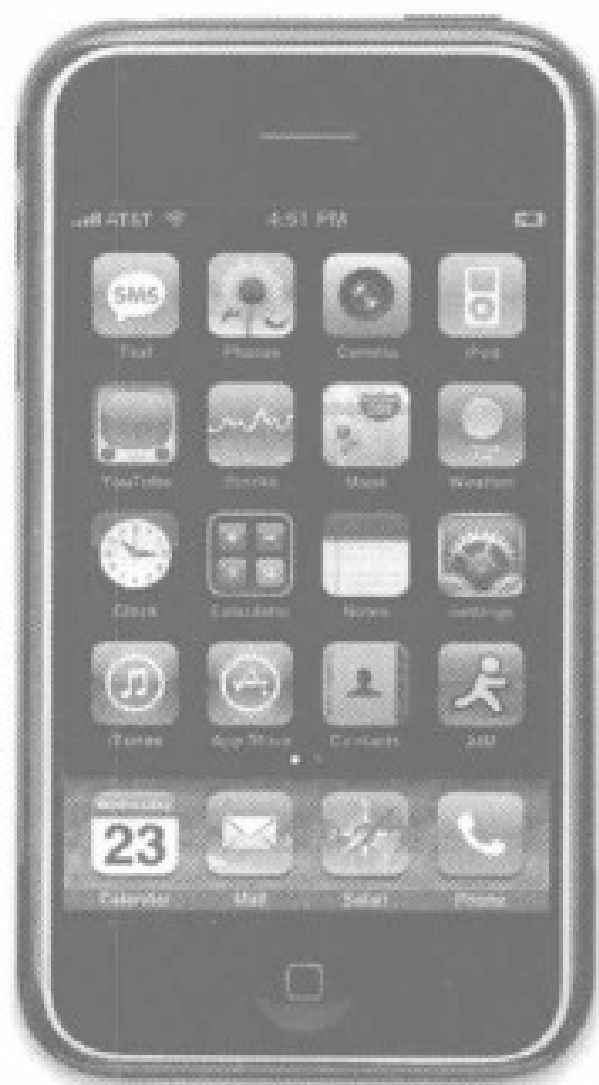


图10-1 最后一列第3个图标就是Settings应用程序图标。该图标在其他iPhone或iPod Touch上的位置可能稍有不同，但始终存在



图10-2 Settings应用程序

互动式应用程序，如游戏，通常应该提供首选项视图，使用户更改设置时无需退出应用程序。甚至实用工具和生产应用程序的首选项，也应该能够让用户在不离开应用程序的情况下进行更改。我们还将介绍如何从应用程序的用户处收集首选项，以及如何将其保存在iPhone的用户默认设置中。

10.2 AppSettings 应用程序

接下来，我们将构建一个简单的应用程序。首先实现一个设置束，这样，当用户启动Settings应用程序时，其中将包含我们的应用程序的一个条目（参见图10-3）。

如果用户选择我们的应用程序，他将看到一个视图，其中显示与我们的应用程序相关的首选项。如图10-4所示，Settings应用程序为用户提供了文本字段、安全文本字段、开关和滑块。

此视图中有两项包含显示指示符。第一个是Protocol，它导向另一个表视图，其中显示该项目的可用选项。用户只能在表视图中选择一个值（参见图10-5）。

在应用程序的Settings应用程序的主视图上还有一个显示指示符，那就是More Settings，它将用户引导至另一组首选项（参见图10-6）。该子视图可能与主设置视图拥有相同类型的控件，它还可以有自己的子视图。Settings应用程序需要使用导航控制器，这是因为它支持构建多级首选项视图。

实际启动AppSettings应用程序后，将会显示一组从Settings应用程序收集来的首选项（参见图10-7）。

为了介绍如何更新应用程序的首选项，我们还在右下角提供了一个较小的信息按钮，它将用

户引导至另一个视图，以设置应用程序的两个首选项值（参见图10-8）。



图10-3 Settings应用程序显示了一个AppSettings条目

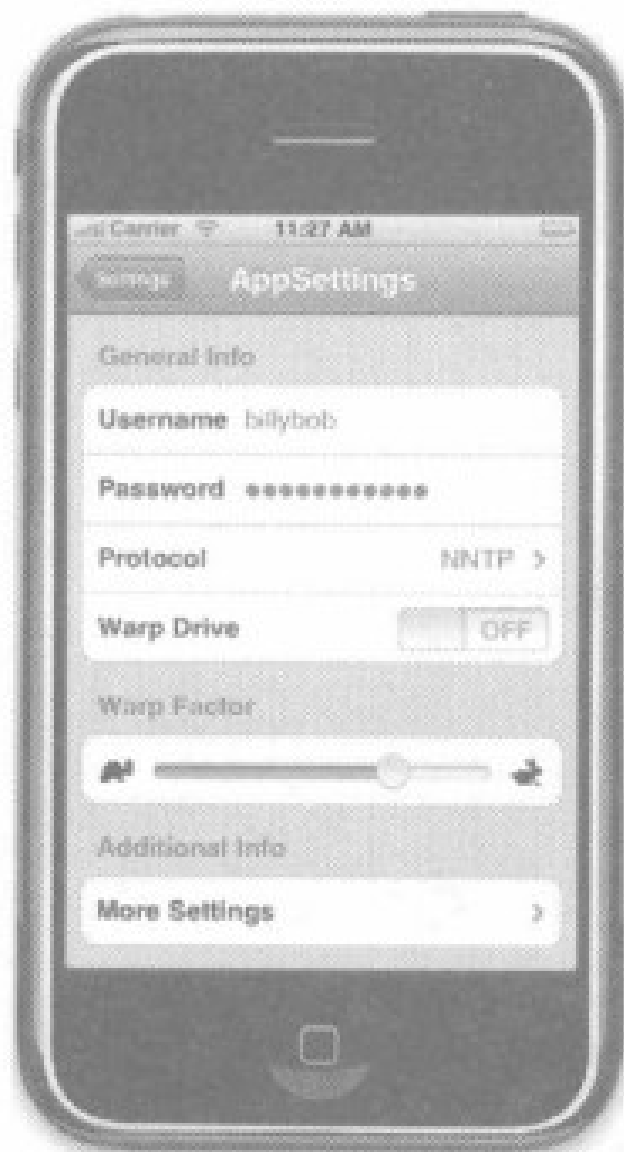


图10-4 AppSettings应用程序的主设置视图

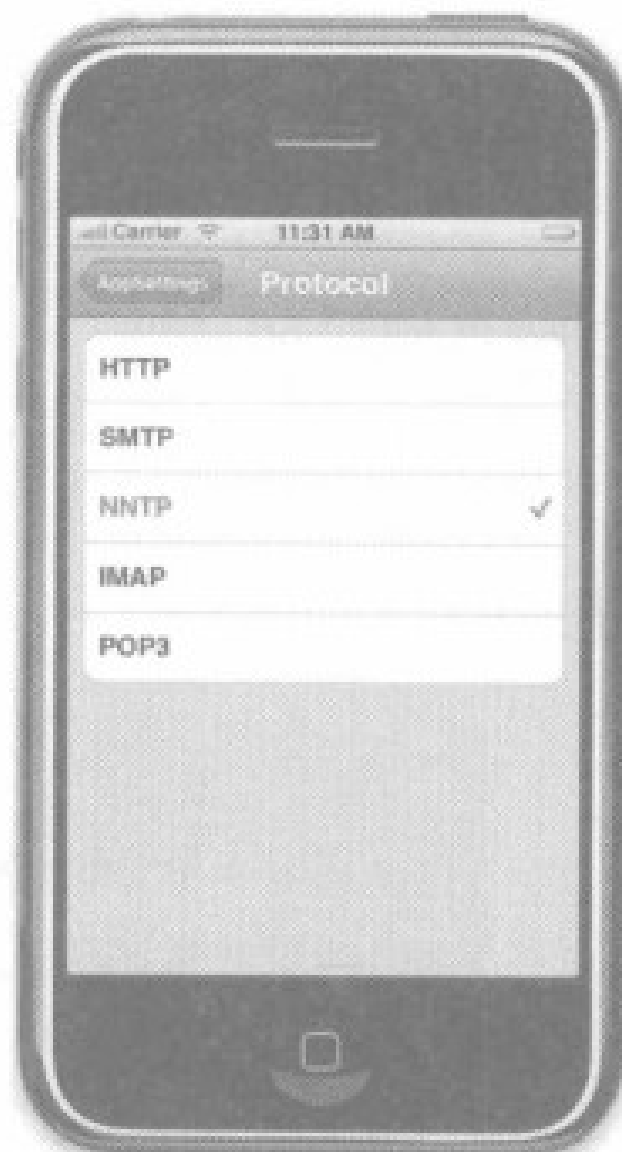


图10-5 从列表选择一个首选项



图10-6 AppSettings应用程序的子设置视图



图10-7 AppSettings应用程序主视图

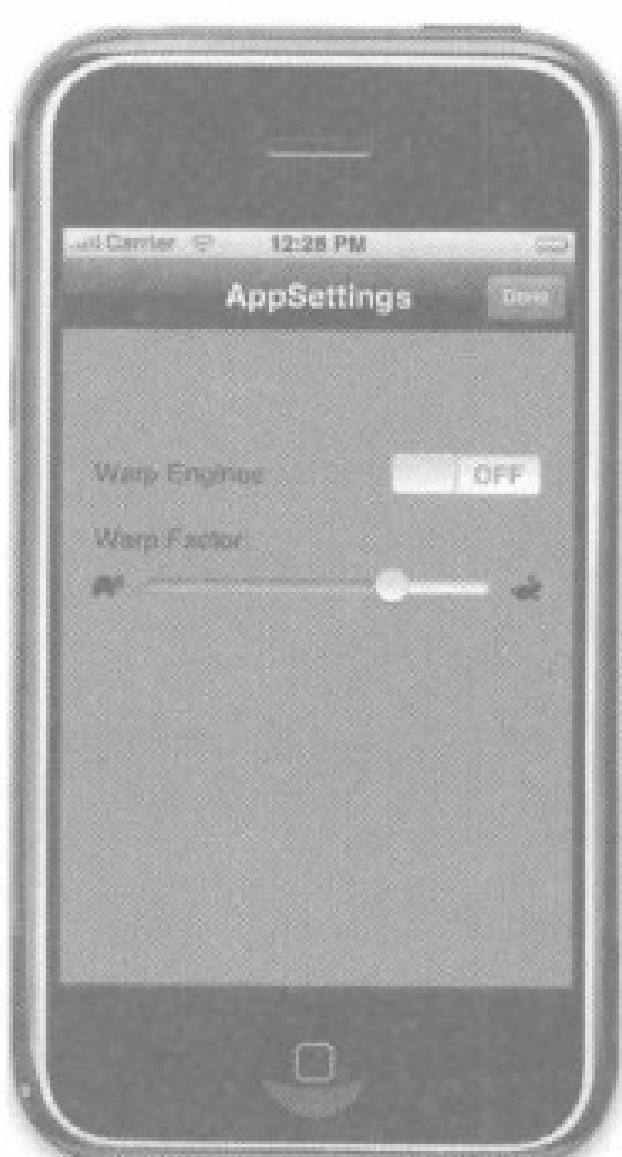


图10-8 设置应用程序中的首选项

让我们开始吧，准备好了吗？

10.3 创建项目

在Xcode中，按下⌘N或从File菜单中选择New Project...。当新项目帮助窗口出现后，选择左侧窗格中iPhone标题下的Application，单击Utility Application图标，然后单击Choose...按钮。将

新项目命名为AppSettings。

我们以前没有使用过这个项目模板，所以在继续操作之前，让我们先熟悉一下这个新模板。该模板创建的应用程序与第6章中构建的应用程序非常相似。此应用程序有一个主视图和一个辅助视图（称为flipside视图）。点击主视图中的信息按钮将进入flipside视图，点击flipside视图中的Done按钮将返回主视图。

你将注意到，Xcode项目中没有Classes文件夹（参见图10-9）。因为实现这种类型的应用程序需要许多文件，该模板已将这些文件组织到一些分组中。展开文件夹Main View、Flipside View和Application Controllers，然后将Resources也打开。

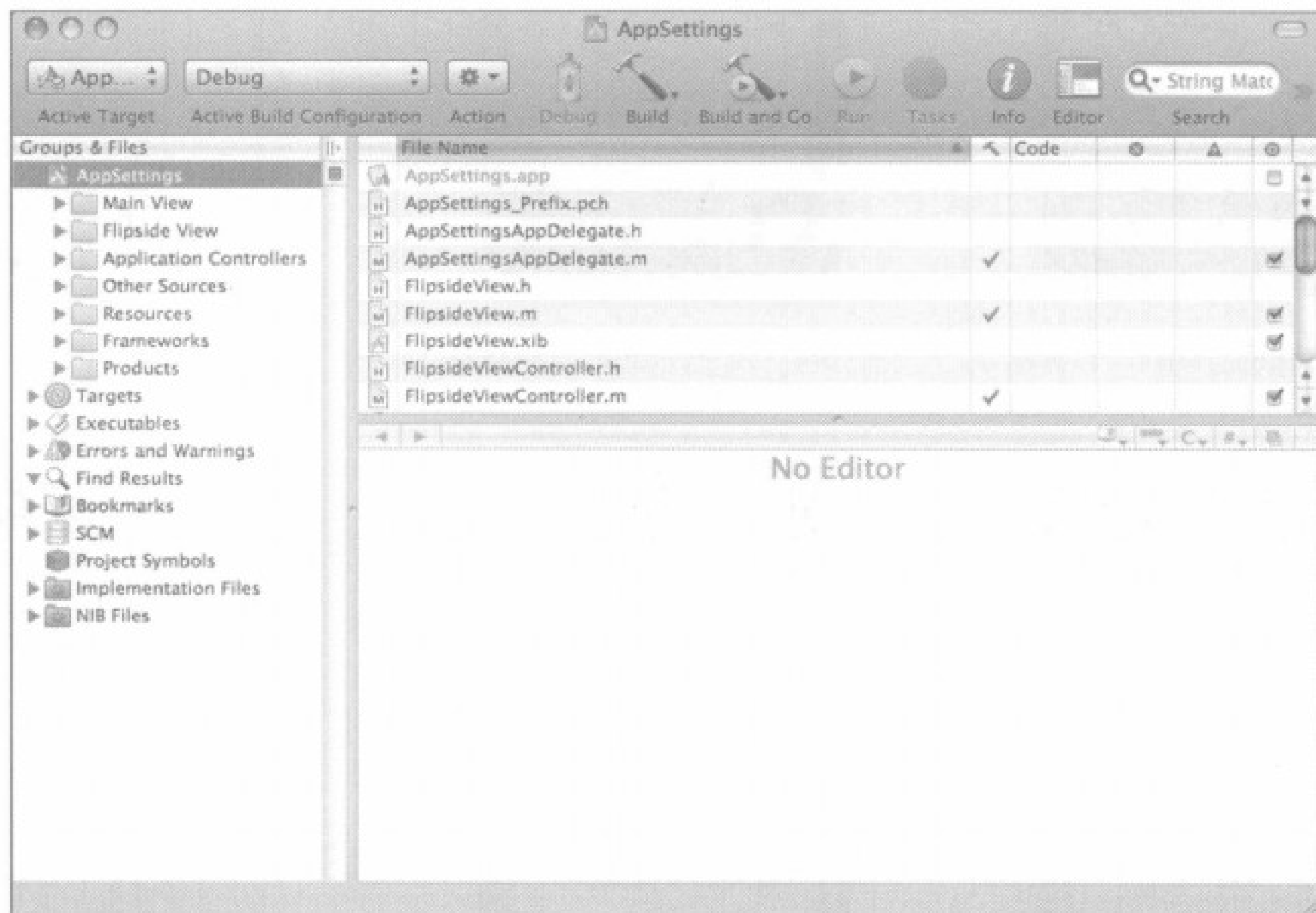


图10-9 使用Utility Application模板创建的项目

组成主视图的所有类，包括视图控制器和一个UIView子类，都包含在Main View文件夹中。同样，实现flipside视图所需的所有源代码文件都包含在Flipside View文件夹中。最后，应用程序委托和根控制器类都包含在Application Controllers文件夹中。

此模板为主视图和flipside视图提供了一个自定义的UIView子类。实际上，在此应用程序中，主视图和flipside视图都不需要UIView子类，但是我们会将FlipsideView和MainView保留在项目中。保留它们并没有什么坏处，但如果删除它们，我们就必须重新连接nib文件，使其指向UIView。

接下来快速更改一下MainWindow.xib。双击MainWindow.xib，在Interface Builder中打开该文件。将主窗口（标题为MainWindow.xib）设置为列表模式（中间的View Mode按钮）。然后，单击Root View Controller图标左侧的显示三角符号，显示View图标。接下来单击View图标左侧的显示三角形，显示Light Info Button图标（参见图10-10）。

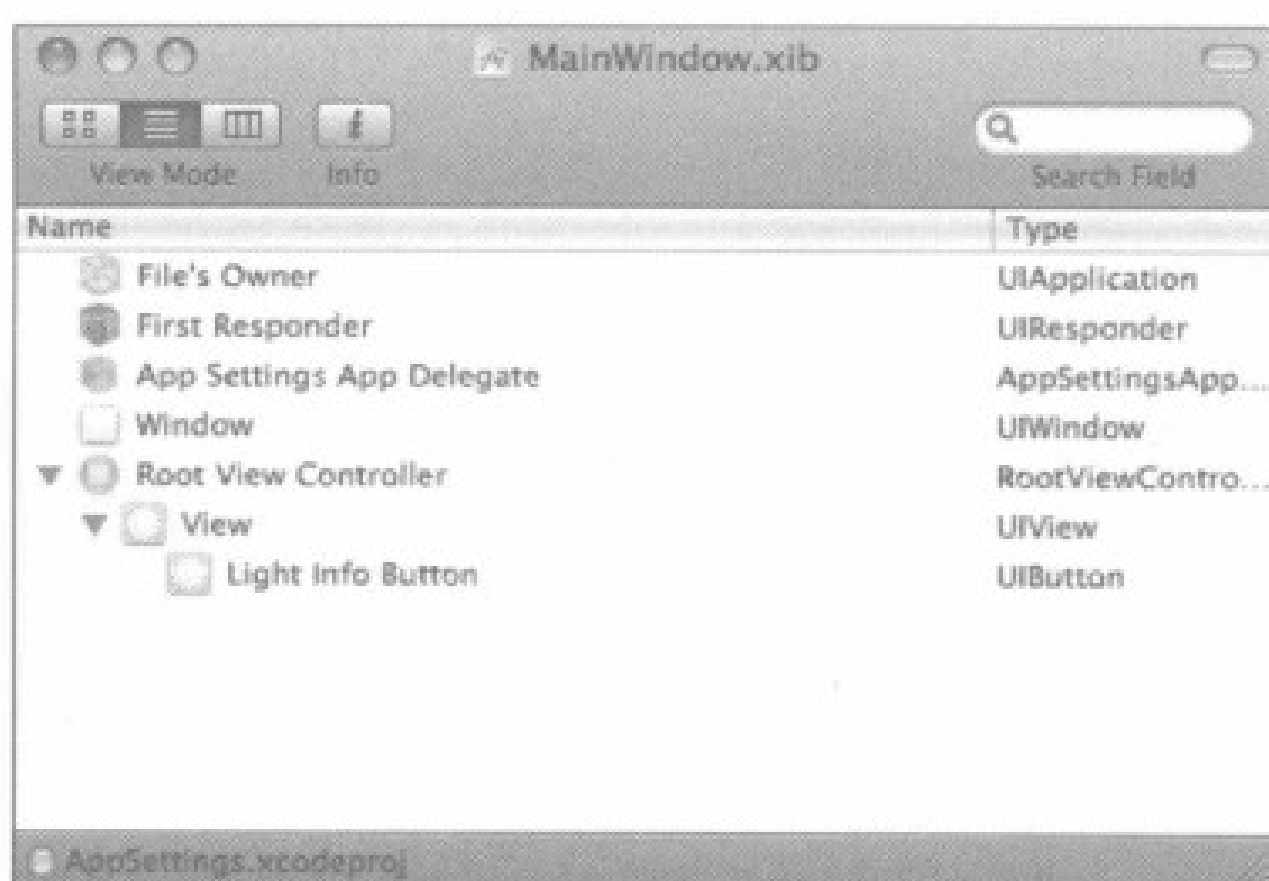


图10-10 使用列表视图模式

提示 无需逐一展开每项，可以按下option键并单击Root View Controller图标旁边的显示三角形，在Interface Builder中展开整个层次结构。

接下来，更改Light Info Button图标，让它在白色背景中显得更为美观。单击Light Info Button图标将其选中，然后按下⌘I调出属性检查器，将按钮的Type由Info Light改为Info Dark。完成之后保存，关闭MainWindow.xib并返回Xcode。

本章的代码已经包含了一些图标，可以确保你创建的应用程序与我们的相似。首先，打开项目归档文件中的10 AppSettings文件夹，抓取icon.png文件，将它添加到项目的Resources文件夹中。

然后，单击Resources文件夹中的info.plist文件，将Icon文件行的值设置为icon.png。

10

10.4 使用设置束

Settings应用程序根据给定应用程序内部的设置束内容来显示该应用程序的首选项。每个设置束必须包含一个名为Root.plist的属性列表，它定义根级首选项视图。此属性列表必须遵循一种非常精确的格式，稍后我们将讨论这种格式。如果Settings应用程序发现设置束中包含一个合适的Root.plist文件，它将根据属性列表的内容为应用程序构建一个设置视图。如果想要首选项包含任何子视图，则必须向设置束中添加其他属性列表，并为每个子视图添加一个Root.plist条目。本章稍后将详细介绍如何操作。

此过程的一个缺点是，无法在Xcode中为设置束添加或删除项。可以从Xcode中更改设置束中已有文件的内容，但是如果需要真正地添加或删除项，则必须在Finder中完成。不必担心，我们将深入介绍如何执行此操作。

10.4.1 在项目中添加设置束

在Groups & Files窗格中，单击根对象（名为AppSettings，应该位于列表最顶端），从File菜单中选择New File...或按下⌘N。选择左窗格中iPhone OS标题下的Settings，然后选择Settings Bundle图标（参见图10-11），单击Next按钮，单击回车键以选择默认名称Settings Bundle。

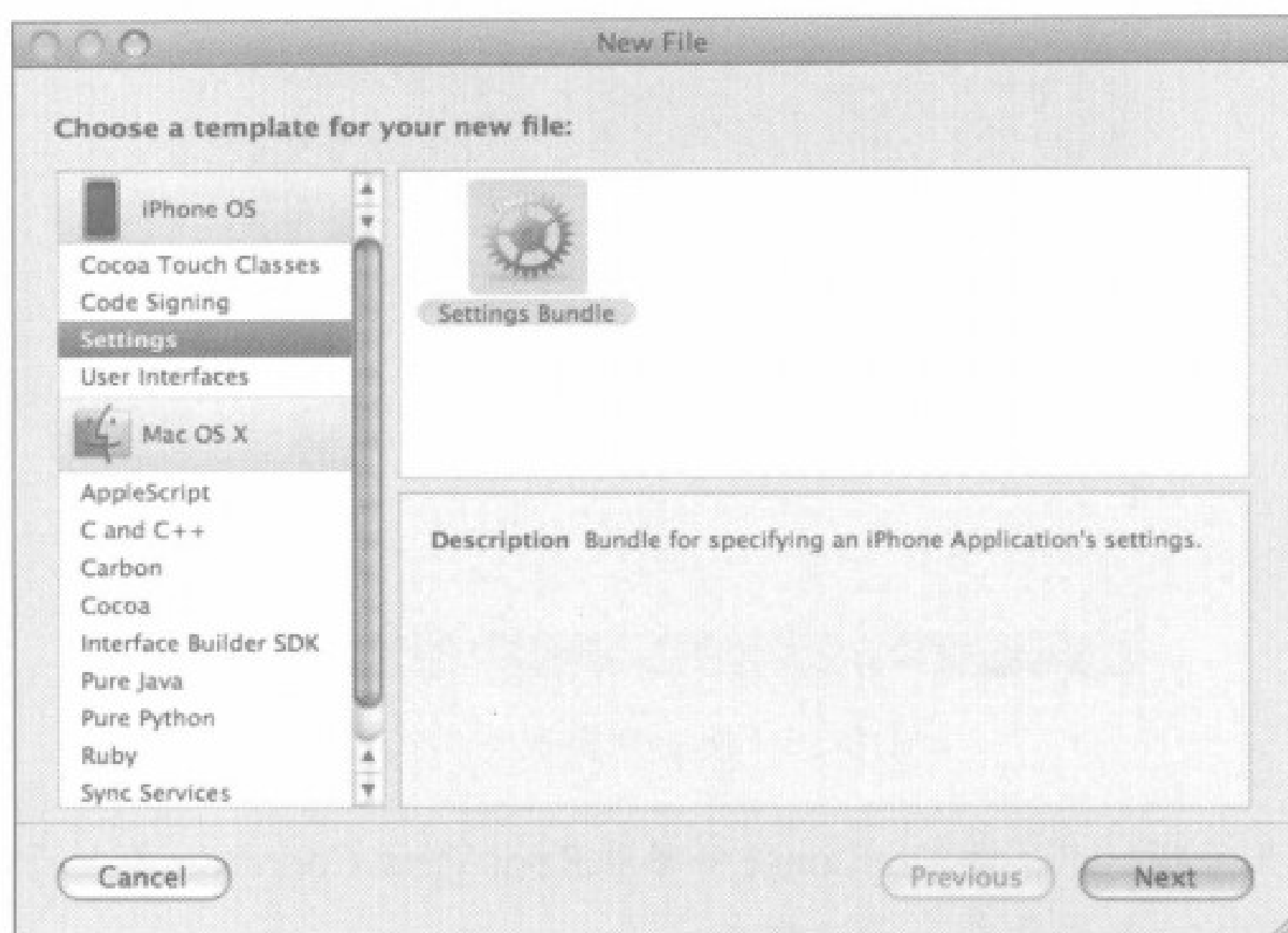


图10-11 创建设置束

现在应该在Xcode的Groups & Files窗格中看到一个新项，名为Settings.bundle。展开Settings.bundle，应该看到Root.plist图标和en.lproj文件夹。我们将在第17章介绍本地化应用程序时讨论en.lproj。现在，主要介绍Root.plist。

10.4.2 设置属性列表

单击Root.plist，查看编辑器窗格，你将看到Xcode的属性列表编辑器。此编辑器与/Developer/Applications/Utilities中Property List Editor应用程序的功能相同。

所有属性列表都有一个类型为Dictionary的根节点，该节点使用键值来保存项，与NSDictionary相同。Dictionary节点的所有子节点都必须有一个键和一个值。任何给定的属性列表都只能有一个根节点，并且所有其他节点都必须位于该节点之下。

属性列表可以包含几种不同类型的节点。除Dictionary节点（它允许使用键存储其他节点）外，还有Array节点，这类节点存储其他节点的一个有序列表，与NSArray类似。Dictionary和Array是唯一能够包含其他节点的属性列表节点类型。还有其他一些节点类型可以保存数据，如Boolean、Data、Date、Number和String。

提示 在NSDictionary中可以使用任何对象作为键，但属性列表Dictionary节点中的键必须为字符串类型，可以选择任何节点类型作为该键的值。

创建一个设置属性列表时，必须遵循已定义的格式。所幸，当在项目添加设置束时，应用程序会创建一个具有适当格式的属性列表，名为Root.plist，它就是你刚才在设置束中单击的Root.plist。

在Root.plist编辑器窗格中，展开PreferenceSpecifiers节点（参见图10-12）。

Key	Type	Value
▼ Root	Dictionary	(3 items)
Title	String	YOUR_PROJECT_NAME
StringsTable	String	Root
▼ PreferenceSpecifiers	Array	(4 items)
▶ Item 1	Dictionary	(2 items)
▶ Item 2	Dictionary	(8 items)
▶ Item 3	Dictionary	(6 items)
▶ Item 4	Dictionary	(7 items)

图10-12 编辑器窗格中的Root.plist

添加首选项标识符之前，先了解一下属性列表以及要求的格式。Root节点下的第一项是键Title，这个名称将出现在应用程序的Settings应用程序部分中。双击Title旁边的当前值YOUR_PROJECT_NAME，将它改为AppSettings。

接下来讨论第二项StringsTable，第17章中也会讲到它。还使用了一个字符串表，用于将应用程序翻译为其他语言。由于这一项是可选的，所以你可以单击StringsTable并按delete键删除此条目。它不影响任何操作，因此你也可以保留它，但是如果删除它，表中就只有一项了。

根节点下的下一项是PreferenceSpecifiers，它是一个数组。单击其显示三角形可以显示它的子项。这个数组节点用于保存一组Dictionary节点，每个Dictionary节点都代表用户可输入的一个首选项或用户可以访问的一个子视图。你将注意到Xcode的模板有4个节点。这些节点与我们实际的首选项没有任何联系，所以分别单击Item 2、Item 3和Item 4，并按delete键将它们删除。

单击Item 1，但不要展开它。注意此行右边带有加号图标的按钮，它用于在这行后面添加一个同级节点。换句话说，它可以在同一级添加另一个节点。如果单击此图标，我们将得到新行Item 2，它位于Item 1之后。

现在展开Item 1，此时加号图标改为一个带有三条横线的图标。此新图标表示单击该按钮将添加一个子节点，因此如果现在单击该按钮，我们将在Item 1下方获得一个新行。

展开Item 2，它下方的第一行有一个键Type；PreferenceSpecifiers数组中的每个属性列表节点都必须有一个带有此键的条目。它通常位于第一项的位置，但在Dictionary节点中顺序并不重要，所以Type键无需是第一个键。当前Item 1的值PSGroupSpecifier用于表示应该启动一个新组。返回图10-4，你会看到Settings应用程序在一个分组表中显示各种设置。设置束属性列表中PreferenceSpecifiers数组中的Item 1应该始终具有此类型，因为每个表至少需要一个组。

Item 1中包含一个名为Title的键，它用于在启动的组上设置一个可选标题。再返回图10-4，你会看到第一组的名称为General Info。双击Title旁边的值，将其由Group改为General Info。

10.4.3 添加文本字段设置

现在，我们需要在此数组中添加另一个项，它将表示第一个实际的首选项字段。我们将从一个简单的文本字段开始。如果在编辑器窗格中单击PreferenceSpecifiers行，然后单击按钮添加一个子项，新行将被插入到列表的顶端，这不是我们想要的结果。我们希望在数组的末尾添加一个项。为此，单击Item 1左侧的显示三角形将其折叠，然后选择Item 1，并单击此行末尾的加号按钮，这将在当前行之后添加一个同级的行（参见图10-13）。

Key	Type	Value
▼ Root	Dictionary	(2 items)
Title	String	AppSettings
▼ PreferenceSpecifiers	Array	(2 items)
▶ Item 1	Dictionary	(2 items)
Item 2	String	

图10-13 添加Item 1的同级项目

新行默认为String节点类型，这不是我们需要的类型。记住，PreferenceSpecifiers数组中的每一项都必须为Dictionary类型，因此单击String，将节点类型改为Dictionary。单击Item 2旁边的显示三角形展开Item 2，此时它还不包含任何内容，唯一的区别是显示三角形指向下方，添加同级节点的按钮变为添加子节点的按钮。现在单击add child node按钮，向此Dictionary节点中添加第一个条目。

有一个新行出现，其类型默认为String，这是我们所需要的类型。新行的键值默认为New item，将它改为Type，然后双击Value列，键入PSTextFieldSpecifier，它告诉Settings应用程序，我们希望用户在文本字段中编辑此设置。

在此例中，PSTextFieldSpecifier是一种类型，更具体地讲，它是一种特定的首选项字段类型。找到Key列中的Type，我们定义将用于编辑首选项的字段类型。

单击Type行右侧带加号图标按钮，向Dictionary添加另一项。这一行指定将在文本字段旁边显示的标签。将键New item改为Title。现在按下tab键。注意，此时要编辑Value列中的值，将它设置为Username。现在按下Title行末尾的加号按钮，向Dictionary添加另一项。

将新条目的键改为Key（这不是印刷错误，确实是将该键设置为“Key”），将其值设为username。回想一下，我们介绍过用户默认设置的工作方式与Dictionary相似。这个新条目告诉Settings应用程序，在存储此文本字段中输入的值时使用什么键。再想一下NSUserDefaults的属性。它允许用户使用键保存值，这与NSDictionary类似。Settings应用程序将对为你保存的每个首选项进行同样的操作。如果你为其提供了一个键值foo，则稍后可以在应用程序中请求foo值，它会显示用户为该首选项输入的值。稍后我们将使用这个键值从应用程序中的用户默认设置获取此设置。

说明 Title拥有值Username，而Key拥有值username。这种大小写差异将会经常出现。Title是在屏幕上显示的内容，所以使用大写字母“U”比较合适。而Key是一个文本字符串，用于从用户默认设置获取首选项，所以所有字母采用小写形式比较合适。我们可以将Title的内容全部小写吗？当然可以。那我们可以将Key的内容全部大写吗？答案是肯定的，但小写才是我们使用的风格。

向Dictionary中添加另一项，将其键设置为AutocapitalizationType，将其值设为None，这指定文本字段不要尝试自动大写用户输入的内容。

创建最后一个新行，将其键设置为AutocorrectionType，将其值设为No，它告诉Settings应用程序不要尝试自动更正输入到该文本字段中的值。如果确实想要自动更改该文本字段的值，则需将其值更改为Yes。完成这些操作后，属性列表应该如图10-14所示。

Key	Type	Value
▼ Root	Dictionary	(3 items)
Title	String	AppSettings
StringsTable	String	Root
▼ PreferenceSpecifiers	Array	(2 items)
▶ Item 1	Dictionary	(2 items)
▼ Item 2	Dictionary	(5 items)
Type	String	PSTextFieldSpecifier
Title	String	Username
Key	String	username
AutocapitalizationType	String	None
AutocorrectionType	String	No

图10-14 在Root.plist中指定的完成后的文本字段

保存属性文件，检查所有属性是否已正确设置并能够生效。现在我们应该能够编译并运行应用程序了。即使应用程序还未实现任何功能，我们也应该能够单击iPhone仿真器上的主按钮，选择Settings应用程序查看应用程序的条目（参见图10-3）。

试验一下，从Build菜单中选择Build and Run。单击主按钮，然后单击Settings应用程序的图标，应该能够看到AppSettings条目，它使用了我们先前添加的应用程序图标。如果单击AppSettings行，应该出现一个简单的设置视图，其中含有一个文本字段，如图10-15所示。

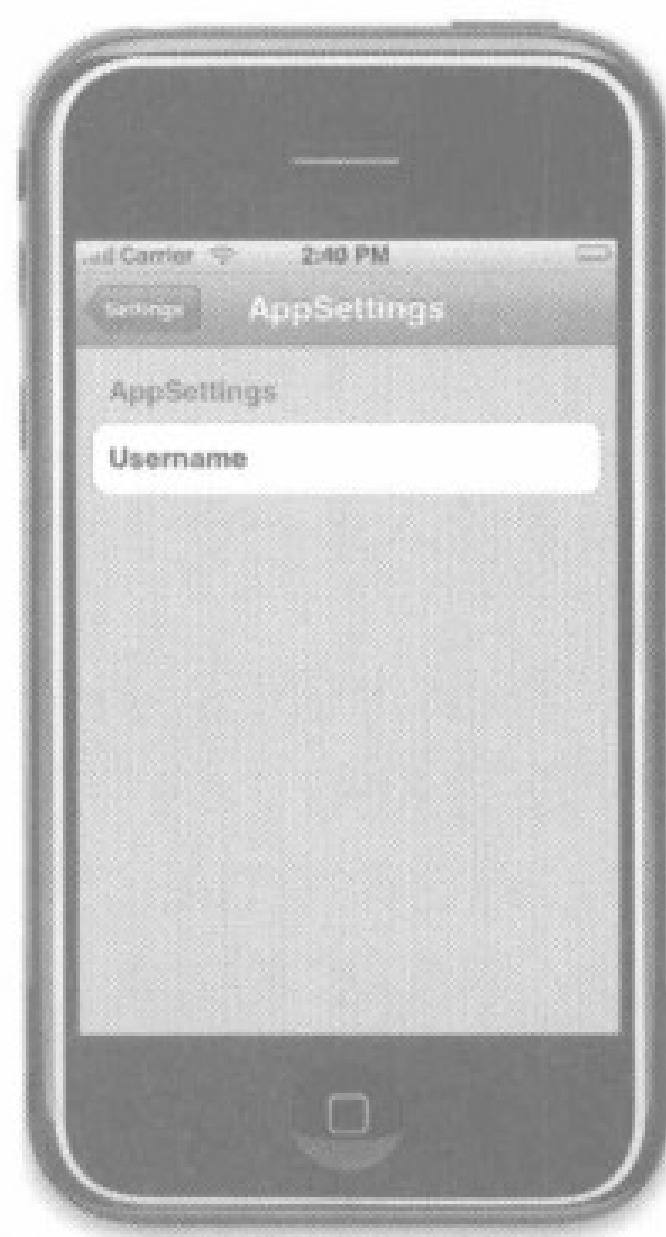


图10-15 Settings应用程序中添加了组和文本字段的根视图

10.4.4 添加安全文本字段设置

退出仿真器，返回Xcode。虽然我们的工作还没有完成，但你应该已经意识到为应用程序添加首选项是很容易的。现在添加根设置视图的其他字段。我们将添加的第一项是用于设置用户密码的安全文本字段。

添加此节点有一种简单方法：折叠PreferenceSpecifiers数组中的Item 2，然后选择Item 2，按⌘C将其复制到剪贴板，再按下⌘V将其粘贴回原来的位置，这将创建一个与Item 2相同的新项Item 3。展开Item 3，将Title改为Password，将Key改为password。

接下来，向Item 3中添加一个子项。记住，项的顺序无关紧要，将新项放置在Key项目下方即可，将其Key设置为IsSecure，将其Type改为Boolean。完成这些操作后，平常用于输入值的空间将变成一个复选框。单击复选框将其选中，这告诉Settings应用程序此字段应该是一个密码字段，而非普通的文本字段。

10.4.5 添加多值字段

我们将添加的下一项是一个多值字段。这种字段类型会自动生成带有显示指示符的行。单击显示指示符将进入另一个表，你可以在多行中进行选择。折叠Item 3并选中该行，单击此行右端的加号图标，添加Item 4。将Item 4的Type改为Dictionary，单击显示三角形展开Item 4。

向Item 4中添加3个子行。每一行的键和值分别为:Type和PSMultiValueSpecifier、Title和Protocol、Key和protocol。接下来的操作有些麻烦,所以操作之前先介绍一下。

我们将向Item 4中添加另外两个子项,但它们的节点类型是Array,而不是String。其中一项名为Titles,用于保存可供用户选择的一组值。另一项是Values,用于保存用户默认设置中实际存储的一组值。Values列表中的第一项与Titles数组中的第一项相对应。因此,如果用户选择第一项,Settings应用程序实际保存的是Values数组中的第一个值。这种Titles/Values对非常方便,为用户提供了易于理解的文本,而实际保存的却是其他文本,如数字、日期、或不同的字符串。这两个数组都是必需的。如果希望两个数组的内容相同,可以只创建一个数组,然后进行复制粘贴并更改副本的键,这样就会得到具有相同的内容但保存在不同键下的两个数组。我们将采用这种方法。

向Item 4中添加一个新的子项,将其键改为Values,并将其类型设置为Array。展开数组,添加5个子节点,所有节点必须为String类型且应该包含以下值:HTTP、SMTP、NNTP、IMAP和POP3。

提示 如果输入第一个值并按下回车键,将会编辑它下面的值。这是一种快捷方式。

输入上述5个值后,折叠Values并将其选中,按⌘C进行复制,然后按⌘V进行粘贴。这将创建一个新项,其键为Values-2。双击Values-2,将它改为Titles。

到此,我们基本完成了多值字段的操作。唯一缺少是Dictionary中的一个必需值,即默认值。多值字段必须有且只有一行被选中,所以我们必须指定要使用的默认值,以防没有值被选中。而且默认值需要与Values数组中的项相对应(如果这两个数组不同的话,就不是Titles数组)。向Item 4中添加另一个子项,将其键设置为DefaultValue,将其值设为SMTP。

检验一下我们的工作。保存属性列表,编译并再次运行。当应用程序启动时,按下主按钮启动Settings应用程序。选择AppSettings,根级视图上应该显示3个字段(参见图10-16)。继续检验创建的多值字段,然后学习下一项任务。



图10-16 3个字段

10.4.6 添加拨动开关设置

需要从用户处获取的下一项是一个Boolean值,该值表示拨动开关是否打开。为了获取首选项中的Boolean值,我们将告诉Settings应用程序使用一个UISwitch,向PreferenceSpecifiers数组添加另一个类型为PSToggleSwitchSpecifier的项。

如果Item 4当前处于展开状态,折叠Item 4,单击以将其选中。单击此行右端的加号图标创建Item 5。将Item 5的类型更改为Dictionary,然后展开Item 5,并向其中添加3个子行,每行的键和值分别设置为:Type和PSToggleSwitchSpecifier、Title和Warp Drive、Key和warp。

默认情况下，拨动开关产生一个Boolean值YES或NO，该值将保存到用户默认设置中。如果想在打开和关闭位置上指定不同的值，可以指定可选的键TrueValue和FalseValue。你可以在打开位置（TrueValue）或关闭位置（FalseValue）指定字符串、日期或数字，这样，Settings应用程序就会保存你指定的字符串，而不是保存YES或NO。我们将打开位置设置为保存字符串Engaged，将关闭位置设置为保存字符串Disabled。

完成此操作需要向Item 5中再添加两个子项，将其键和值分别设置为：TrueValue和Engaged、FalseValue和Disabled。

此Dictionary节点中还有一项是必需的，即默认值。如果我们没有提供FalseValue和TrueValue项，则需要创建一个新行，将其键设置为DefaultValue，并将其类型由String改为Boolean。但是，由于我们确实已经添加了这两项，所以我们输入到DefaultValue中的值必须与传入TrueValue或FalseValue的值相匹配。

我们将拨动开关的默认状态设置为打开，向Item 5中添加最后一个子项，并将其键和值分别设为DefaultValue和Engaged。注意，字符串“Engaged”是将保存在用户默认设置中的内容，而不是将显示在屏幕上的内容。

10.4.7 添加滑块设置

接下来我们将添加的项是一个滑块。在Settings应用程序中，滑块两端可以各有一个小图像，但它不能有标签。我们将滑块放置在一个带有自己的标题的组中，以便用户了解滑块的作用。

单击PreferenceSpecifiers下的Item 1，按⌘C将它复制到剪贴板。现在，选择Item 5，同时确保它处于折叠状态，然后按⌘V进行粘贴。由于Item 1是一个组标识符，所以我们刚才粘贴进来的新项Item 6也是一个组标识符，它将告诉Settings应用程序，应该在此位置生成一个新组。

展开Item 6，双击标为Title的行中的值，并将它改为Warp Factor。

折叠Item 6并将其选中，单击其所在行右端的按钮，添加一个新的同级行。将新行Item 7的Type由String改为Dictionary，然后展开新行。向Item 7中添加两个子行，将第一行的键和值分别设为Type和PSSliderSpecifier，它指示Settings应用程序应该使用UISlider从用户处获取此信息。将另一行的键和值分别设为Key和warpfactor，这样，Settings应用程序就会知道存储该值时使用什么键。

我们允许用户输入1到10之间的一个值，并将默认值设置为warp 5。滑块需要有一个最小值、一个最大值和一个起始（或默认）值，所有这些值都需要以数字（而非字符串）的形式保存在属性列表中。为此，向Item 7中添加3个子行，将它们的Type由String改为Number，并将它们的键和值分别设为DefaultValue和5、MinimumValue和1、MaximumValue和10。

如果想要测试一下滑块，那就测试吧，但测试之后应立即返回来。我们将对滑块进行一些自定义。滑块两端允许分别放置一个21×21像素的小图像。我们将提供一些图标来指示向左滑动会降低速度，向右滑动会提高速度。

在本书附带的项目归档文件中的10 AppSettings文件夹中有两个分别名为rabbit.png和turtle.png的图标。我们需要将这两个图标添加到设置束中。因为Settings应用程序需要使用这两个

图标，因此不能仅将它们放在Resources文件夹中，而需要将其放在设置束中，这样Settings应用程序才能获取它们。为此，进入Finder并导航到存储Xcode项目的文件夹。这个文件夹中包含一个名为Settings.bundle的图标。

记住，在Finder中，束看起来像文件，但实际上它们是文件夹，可以通过按住Control并单击（或右键单击）束的图标，然后选择Show Package Contents来访问束的内容。这将打开一个新窗口，你应该能够看到在Xcode的Settings.bundle中看到的两个图标。将图标文件rabbit.png和turtle.png从10 AppSettings文件夹中复制到此文件夹中。

将刚才复制到设置束中的两个文件添加到Xcode项目的Resources文件夹中。无需将它们复制到项目中，只需创建首选项即可。这使我们能够在应用程序中使用这两个图标。即使设置束中的图像将被编译到应用程序中，它们也不能被我们的应用程序访问，只能供Settings应用程序访问，除非我们执行了这一步。

在Finder中将此窗口保留为打开状态，因为稍后我们还要在其中复制另一个文件。现在，返回Xcode，告诉滑块使用这两个图像。

返回到Root.plist，在Item 7下添加两个子行，将它们的键和值分别设为MinimumValueImage和turtle.png、MaximumValueImage和rabbit.png。保存属性列表，编译并运行应用程序，以确保所有属性都能够生效。如果所有属性都能正常运行，则应该能够导航到Settings应用程序，找到两端分别带有酣睡的海龟和快乐的兔子图标的滑块（参见图10-17）。



图10-17 我们已经拥有文本字段、多值字段、拨动开关和滑块。我们的应用程序快要完成了

10.4.8 添加子设置视图

我们将添加另一个首选项标识符，告诉Settings应用程序，我们希望它显示一个子设置视图。此标识符将呈现一个带有显示指示符的行，点击该显示指示符会调出一个全新的首选项视图。由于我们不希望新的首选项与滑块分到同一组，因此在添加此节点之前，我们将复制Item 1中的组标识符，并将其粘贴到PreferenceSpecifiers数组的末尾，为子设置视图创建一个新组。在Root.plist中，如果Item 1已展开，将其折叠。然后单击以将其选中，并按⌘C将其复制到剪贴板。接下来，如果Item 7已展开，将其折叠。然后单击以将其选中，然后按⌘V粘贴新项Item 8。展开Item 8，双击键Title旁边的值列，将它由General Info改为Additional Info。

现在，再次折叠Item 8。选择Item 8，并按下此行右端的add sibling按钮以添加Item 9，它将是实际的子视图。将新行的类型由String更改为Dictionary，并单击显示三角形将其展开。向其中添加两个子行，将它们的键和值分别设为Type和PSChildPaneSpecifier、Title和More Settings。

我们需要添加最后一行，它将告诉Settings应用程序，为More Settings视图加载哪个属性列表。

添加另一个子行，并将其键和值分别设为File和More。假定文件扩展名为.plist，且不应包含在文件名中，否则Settings应用程序将找不到此属性列表文件。

现在，我们需要向设置束中添加另一个属性列表，以描述应该在该子视图上获取的首选项值。Settings应用程序中的每个视图都以一个属性列表文件为基础，因此我们需要向设置束中添加另一个属性列表文件来定义该子视图。不能在Xcode中向设置束添加新文件，且属性列表编辑器的Save对话框不允许将新文件保存到设置束中。因此，我们必须创建一个新的属性列表，将它保存到其他地方，然后使用Finder将它拖入Settings.bundle窗口。

现在，你可以看到能够在设置束属性列表文件中使用的所有不同类型的首选项字段，因此为了节约编写代码的时间，可以使用本书附带的项目归档文件中的10 AppSettings文件夹中的More.plist，将它拖入到之前打开的Settings.bundle窗口。完成这些操作后，返回Xcode，这样，我们就能够将此新属性列表添加为Root.plist文件的子项。

提示 创建子设置视图的最简单的方法是复制Root.plist，并重新命名副本。然后删掉除第一项以外的所有现有首选项标识符，将需要的任何首选项标识符添加到此新文件中。

保存工作，我们现在已经完成设置束的相关操作。你可以编译、运行和测试Settings应用程序。你应该能够进入该子视图并设置所有其他字段的值。继续测试，还可以任意更改属性列表。我们涵盖了几乎所有可用的配置选项（至少在撰写本书时是这样），你也可以在iPhone开发中心的Settings Application Schema Reference文档中找到设置属性列表格式的完整文档。可以在以下网页找到该文档以及许多其他有用的参考文档：

<http://developer.apple.com/iphone/library/navigation/Reference.html>

10.5 读取应用程序中的设置

现在问题解决了一半。用户能够获取我们的首选项，但我们如何获取用户的首选项呢？方法非常简单。

我们将使用NSUserDefaults类读取用户设置。NSUserDefaults作为单一实例实现，这意味着应用程序中只有一个NSUserDefaults实例在运行。为了访问这个单一实例，我们调用类方法standardUserDefaults，如下所示：

```
NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
```

有了指向标准用户默认设置的指针之后，可以像使用NSDictionary一样使用它。要获取标准用户默认设置的值，可以调用objectForKey:，它会返回一个Objective-C对象，如NSString、NSDate或NSNumber。如果想要以标量（如整型、浮点型或布尔型）的形式获取该值，我们可以使用intValueForKey:、floatForKey:、boolForKey:等其他方法。

在创建应用程序的属性列表时，将创建一个PreferenceSpecifiers数组。其中一些标识符用于创建组。另一些标识符用于创建用户进行设置时使用的接口对象。这些才是我们真正感兴趣的标识符，因为它们才是实际数据所在的地方。绑定到一个用户设置的每个标识符都有一个名为Key

的键。回顾一下前面的内容。例如，滑块的键拥有值warpfactor。Password字段的键为password。我们将使用这些键获取用户设置。

现在我们已经拥有了显示设置的位置，接下来使用一组标签快速设置一下主视图。进入Interface Builder之前，先为我们需要的所有标签创建输出口。单击MainViewController.h，并进行如下更改：

```
#define kUsernameKey      @"username"
#define kPasswordKey     @"password"
#define kProtocolKey     @"protocol"
#define kWarpDriveKey    @"warp"
#define kWarpFactorKey   @"warpfactor"
#define kFavoriteTeaKey  @"favoriteTea"
#define kFavoriteCandyKey @"favoriteCandy"
#define kFavoriteGameKey @"favoriteGame"
#define kFavoriteExcuseKey @"favoriteExcuse"
#define kFavoriteSinKey  @"favoriteSin"

#import <UIKit/UIKit.h>

@interface MainViewController : UIViewController {
    IBOutlet UILabel *usernameLabel;
    IBOutlet UILabel *passwordLabel;
    IBOutlet UILabel *protocolLabel;
    IBOutlet UILabel *warpDriveLabel;
    IBOutlet UILabel *warpFactorLabel;

    IBOutlet UILabel *favoriteTeaLabel;
    IBOutlet UILabel *favoriteCandyLabel;
    IBOutlet UILabel *favoriteGameLabel;
    IBOutlet UILabel *favoriteExcuseLabel;
    IBOutlet UILabel *favoriteSinLabel;
}
@property (nonatomic, retain) UILabel *usernameLabel;
@property (nonatomic, retain) UILabel *passwordLabel;
@property (nonatomic, retain) UILabel *protocolLabel;
@property (nonatomic, retain) UILabel *warpDriveLabel;
@property (nonatomic, retain) UILabel *warpFactorLabel;

@property (nonatomic, retain) UILabel *favoriteTeaLabel;
@property (nonatomic, retain) UILabel *favoriteCandyLabel;
@property (nonatomic, retain) UILabel *favoriteGameLabel;
@property (nonatomic, retain) UILabel *favoriteExcuseLabel;
@property (nonatomic, retain) UILabel *favoriteSinLabel;

-(void)refreshFields;
@end
```

代码中没有涉及什么新知识。我们声明了一些常量。这些常量是我们在属性列表文件中为不同的首选项字段使用的键值。然后我们声明了10个输出口及其所有标签，并创建了每个标签的属性。最后，我们声明了一个方法，它将读取用户默认设置中的设置，并将这些值传送至各个标签。

我们将此函数放在一个独立的方法中，因为我们要在多个位置完成这项任务。现在我们已经声明了各个输出口，下面转到Interface Builder。

双击MainView.xib，在Interface Builder中将其打开。视图出现后，你会发现它的背景为暗灰色。我们将它改为白色。在nib的主窗口中单击Main View图标，按⌘1调出属性检查器。使用Background中的颜色将背景改为白色。如果Main View窗口还未打开，那么双击Main View图标将其打开。

在库中查找Label。我们需要使用20个标签。其中一半是静态标签，它们为粗体且右对齐；另一半用于显示从用户默认设置获取的实际值，并使输出口指向这些标签。依照图10-18创建此视图。创建的视图无需与图10-18精确匹配，但视图上必须有一个标签与我们声明的每个输出口对应。继续设计此视图。完成后返回到Interface Builder，然后进行其他操作。

我们要完成的下一个任务是按住Control键并从File's Owner拖到每个用于显示设置值的标签。要将每个标签都设置为指向不同的输出口，需要按住Control并拖动10次。将所有10个输出口与标签相连之后，保存工作并关闭MainView.xib窗口，然后返回Xcode。

单击MainViewController.m，添加以下代码。完成之后，我们将检查一下我们的工作成果。

```
#import "MainViewController.h"
#import "MainView.h"

@implementation MainViewController
@synthesize usernameLabel;
@synthesize passwordLabel;
@synthesize protocolLabel;
@synthesize warpDriveLabel;
@synthesize warpFactorLabel;
@synthesize favoriteTeaLabel;
@synthesize favoriteCandyLabel;
@synthesize favoriteGameLabel;
@synthesize favoriteExcuseLabel;
@synthesize favoriteSinLabel;

-(void)refreshFields
{
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    usernameLabel.text = [defaults objectForKey:kUsernameKey];
    passwordLabel.text = [defaults objectForKey:kPasswordKey];
    protocolLabel.text = [defaults objectForKey:kProtocolKey];
    warpDriveLabel.text = [defaults objectForKey:kWarpDriveKey];
    warpFactorLabel.text = [[defaults objectForKey:kWarpFactorKey]
        stringValue];
    favoriteTeaLabel.text = [defaults objectForKey:kFavoriteTeaKey];
```

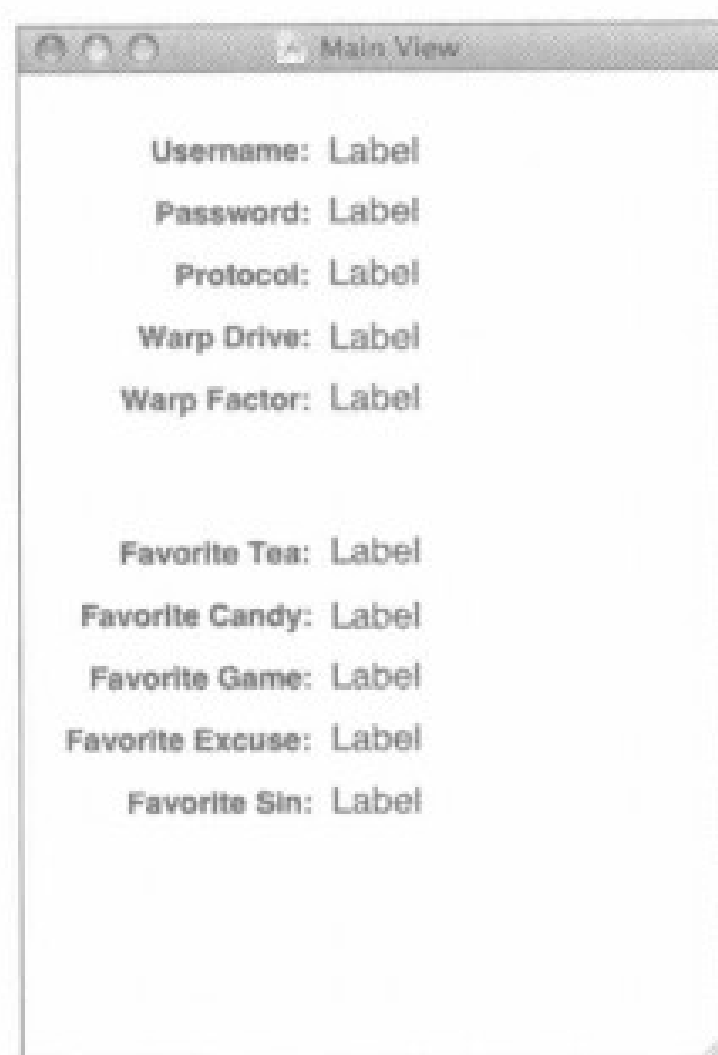


图10-18 Interface Builder中的Main View窗口


```
        favoriteCandyLabel.text = [defaults objectForKey:kFavoriteCandyKey];
        favoriteGameLabel.text = [defaults objectForKey:kFavoriteGameKey];
        favoriteExcuseLabel.text = [defaults objectForKey:kFavoriteExcuseKey];
        favoriteSinLabel.text = [defaults objectForKey:kFavoriteSinKey];
    }

- (id)initWithNibName:(NSString *)nibNameOrNil
  bundle:(NSBundle *)nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil
      bundle:nibBundleOrNil]) {
        // Custom initialization
    }
    return self;
}

- (void)viewDidAppear:(BOOL)animated {
    [self refreshFields];
    [super viewDidAppear:animated];
}

- (void)viewDidLoad {
    [self refreshFields];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
  interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [usernameLabel release];
    [passwordLabel release];
    [protocolLabel release];
    [warpDriveLabel release];
    [warpFactorLabel release];
    [favoriteTeaLabel release];
    [favoriteCandyLabel release];
    [favoriteGameLabel release];
    [favoriteExcuseLabel release];
    [favoriteSinLabel release];
    [super dealloc];
}

@end
```



上面的代码中需要解释的内容不是很多。新方法refreshFields仅用于抓取标准用户默认设置，并使用我们输入到属性文件中的键值，将所有标签的文本属性设置为用户默认设置中的适当对象。注意，对于warpFactorLabel，我们在返回到对象上调用stringValue。所有其他首选项都是字符串，它们都以NSString对象的形式从用户默认设置返回。但是，滑块存储的首选项以NSNumber形式返回，因此我们调用该对象上的stringValue来获取它存储的字符串表示。

随后，我们添加viewDidLoad和viewDidAppear:方法，我们将在这两个方法中调用refreshFields方法。这使得在视图载入时，显示的字段将设置为合适的首选项值，然后在视图更换首选项值时，显示的字段将被刷新。这样操作是因为稍后我们将实现flipside视图，用户将能够在其中更改一些设置。如果不刷新viewDidAppear:中的字段，任何时候在flipside视图上更改的值都不会在前端显示。

你可能在想为什么我们选择viewDidAppear:，而不是viewWillAppear:，原因是我们想在将视图呈现给用户之前进行更改。具体原理涉及到时限。我们将在FlipsideViewController的viewWillDisappear:方法中将更改返回到用户默认设置中。该控制器中的viewWillAppear:方法将在FlipsideViewController控制器的viewWillDisappear:方法之前被调用，因此对字段的更新不会生效。但是不用担心，因为滑动动画在1秒之后才会发生，当主要的辅助视图再次显示时，更新已经完成，用户不会看到更改过程。

完成这个类之后，现在应该能够编译并运行应用程序了。除非你在Settings应用程序中输入了其他值，否则运行结果将与图10-7类似。操作过程很容易，不是吗？

10.6 更改应用程序中的默认设置

现在我们的主视图已经完成并能够运行了，接下来创建flipside视图。如图10-19所示，flipside视图具有拨动开关和滑块。我们要使用的控件与Settings应用程序为这两项使用的控件相同：开关和滑块。首先，我们需要声明输出口，单击FlipsideViewController.h，并进行如下更改：

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

@interface FlipsideViewController : UIViewController {
    IBOutlet    UISwitch *engineSwitch;
    IBOutlet    UISlider *warpFactorSlider;
}
@property (nonatomic, retain) UISwitch *engineSwitch;
@property (nonatomic, retain) UISlider *warpFactorSlider;
@end
```

现在，双击FlipsideView.xib，在Interface Builder中将其打开。如果Flipside View窗口未打开，双

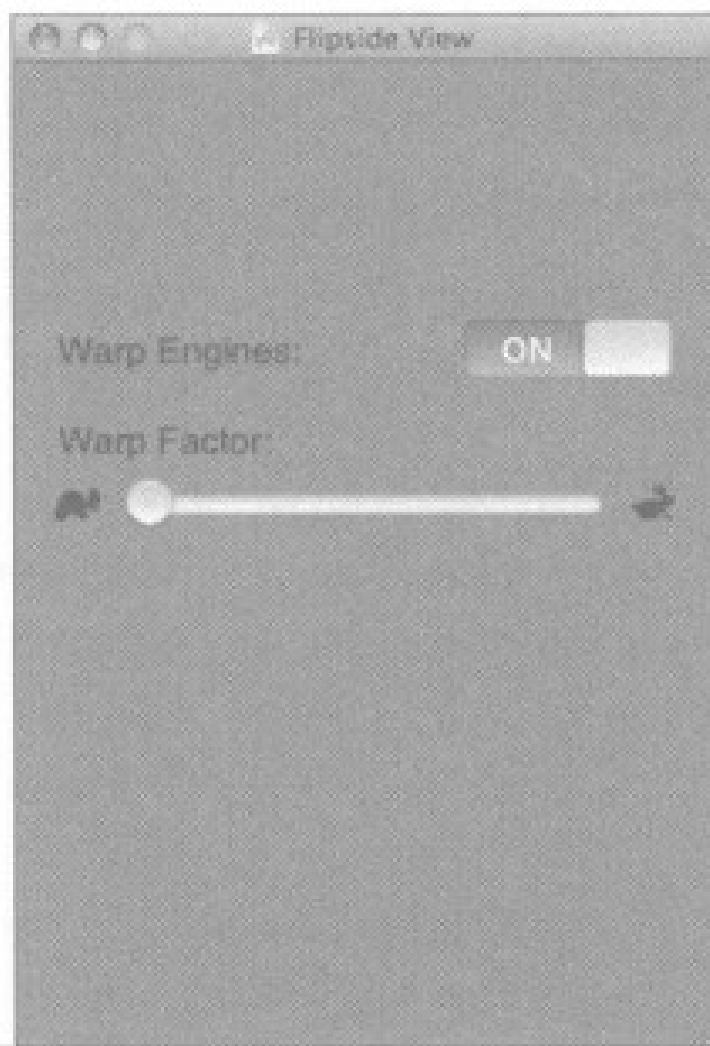


图10-19 在Interface Builder中设计flipside视图

击nib主窗口中的Flipside View图标将其打开。从库中拖出两个Label,并将它们放置在Flipside View窗口。双击一个标签并将其名称改为Warp Engines:。双击另一个标签并将其名称改为Warp Factor:。可以参考图10-19进行布局。

接下来,从库中拖出一个Switch,将它放置在视图右侧的Warp Engines标签旁边。按住Control键并从File's Owner图标拖到新开关,将其连接到engineSwitch输出口。

现在从库中拖出一个Slider,将它放置在Warp Factor标签下方。调整滑块的大小,将其从左侧的蓝色引导线拉伸到右侧的引导线,然后按住Control键并从File's Owner图标拖到滑块,将其连接到warpFactorSlider输出口。

如果未选中滑块,则单击滑块将其选中,按下⌘1调出属性检查器。将Minimum、Maximum和Initial分别设为1.00、10.00和5.00。然后,分别选择turtle.png和rabbit.png作为Min Image和Max Image的图标。完成这些操作后,属性检查器应该如图10-20所示。

保存并关闭nib,返回Xcode,以便完成flipside视图控制器。单击FlipsideViewController.m,进行如下更改:

```
#import "FlipsideViewController.h"
#import "MainViewController.h"

@implementation FlipsideViewController
@synthesize engineSwitch;
@synthesize warpFactorSlider;

- (void)viewDidLoad {
    self.view.backgroundColor = [UIColor viewFlipsideBackgroundColor];

    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    engineSwitch.on = ([[defaults objectForKey:kWarpDriveKey]
        isEqualToString:@"Engaged"]) ? YES : NO;
    warpFactorSlider.value = [defaults floatForKey:kWarpFactorKey];
}

- (void)viewWillDisappear:(BOOL)animated
{
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    NSString *prefValue = (engineSwitch.on) ? @"Engaged" : @"Disabled";
    [defaults setObject:prefValue forKey:kWarpDriveKey];
    [defaults setFloat:warpFactorSlider.value forKey:kWarpFactorKey];
    [super viewWillDisappear:animated];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
    interfaceOrientation {
    // Return YES for supported orientations

```

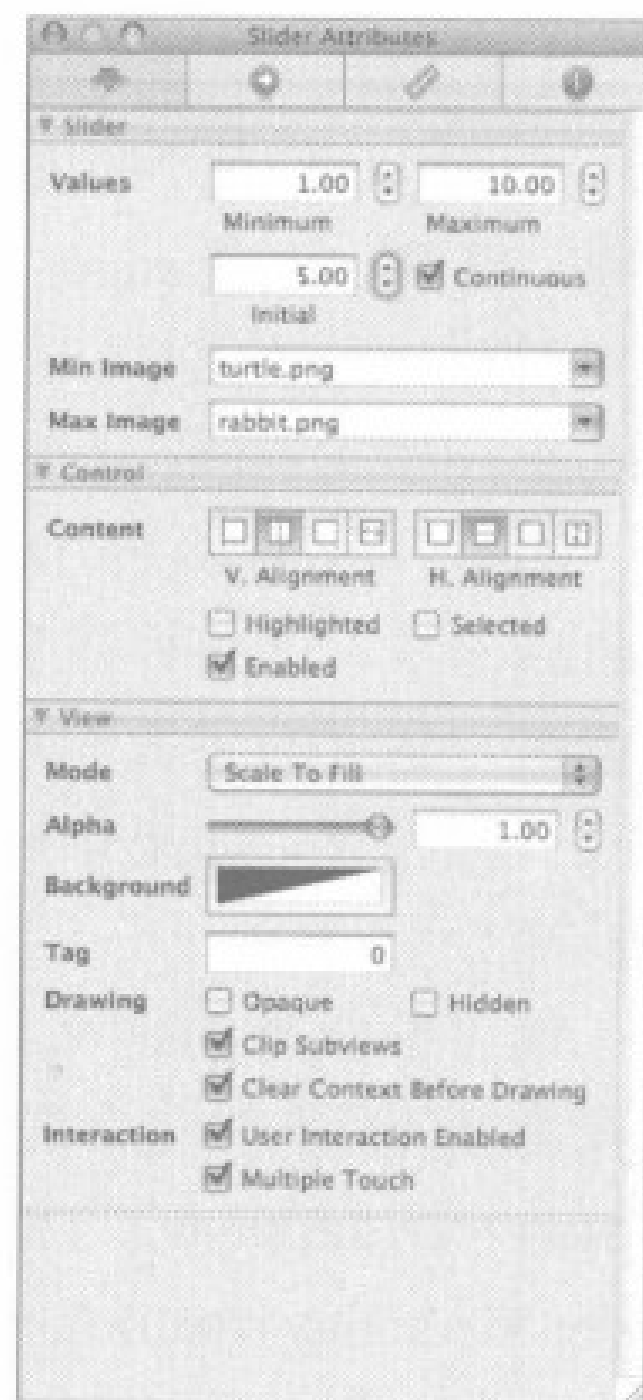


图10-20 Warp Factor滑块的属性检查器

```

        return (interfaceOrientation == UIInterfaceOrientationPortrait);
    }

    - (void)didReceiveMemoryWarning {
        [super didReceiveMemoryWarning];
        // Releases the view if it doesn't have a superview
        // Release anything that's not essential, such as cached data
    }

    - (void)dealloc {
        [engineSwitch release];
        [warpFactorSlider release];
        [super dealloc];
    }
@end

```

在viewDidLoad方法中，我们删除了一行代码并添加了3行代码（其实添加了4行，因为其中一行太长，被分为两行显示）。删除的一行代码不太重要。模板中的代码使用一个类方法设置视图的背景色，这行代码将flipside视图的外观设置为具有纹理的暗灰色，而没有使用在Interface Builder中设置的背景。具有纹理的背景使得难以看清文字和滑块图片。删除这行代码可以使Interface Builder中的背景色更加明亮，使文字和图标更容易看清。

添加的4行代码获取标准用户默认设置，并使用开关和滑块的输出口将它们设置为用户默认设置中存储的值。由于我们选择以字符串形式（而不是Boolean值）存储拨动开关的值，并且UISwitch实例使用BOOL属性进行设置，所以我们必须在代码中进行转换。

```

NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
engineSwitch.on = ([[defaults objectForKey:kWarpDriveKey]
    isEqualToString:@"Engaged"]) ? YES : NO;
warpFactorSlider.value = [defaults floatForKey:kWarpFactorKey];

```

我们还覆盖了父类的viewWillDisappear:方法，以便在主视图再次显示之前，将控件的值填充到用户默认设置中。因为控制器的viewWillDisappear:方法将在主视图的viewWillAppear:方法之前触发，更改的值已经保存在用户默认设置中以供视图检索，所以主视图将使用正确的新值进行更新。

10.7 小结

现在，你应该已经牢固掌握了Settings应用程序和用户默认设置，了解了如何向应用程序添加设置束，如何为应用程序的首选项构建多级视图。你还学会了如何使用NSUserDefaults读取和写入首选项，如何让用户从应用程序中更改首选项，你甚至还有机会在Xcode中使用新的项目模板。现在你应该能够处理许多应用程序首选项了。

下一章将介绍管理iPhone上的文件的不同方法。我们将涵盖将对象持久化到文件系统的不同技术，还将了解如何使用iPhone的嵌入式数据库SQLite。准备好了吗？我们出发吧！

到目前为止，我们重点介绍了模型-视图-控制器范型的控制器和视图方面。尽管我们的几个应用程序已经从应用程序包中读取了数据，但是没有任何一个应用程序将其数据保存到任何形式的持久性存储，持久性存储就是某种形式的非易失性存储，这种存储在重新启动计算机或设备时也不会丢失数据。到目前为止，每个示例应用程序都没有存储数据，也没有使用易失性存储或非持久性存储。每次启动其中一个示例应用程序时，显示的数据都与第一次启动时完全相同。

到现在为止，我们一直使用这种方法。但是在现实世界中，应用程序需要持久存储数据，以便用户进行更改时能够保存这些更改，并能在再次启动该程序时仍然位于原处。我们可以使用多种不同的机制将数据持久存储在iPhone上。如果采用Cocoa for Mac OS X编程，则可能会使用其中一些或所有这些技术。

本章将介绍将数据持久存储到iPhone的文件系统的三种不同的机制。并介绍如何使用属性列表、对象归档以及iPhone的嵌入式关系数据库（称为SQLite3）。我们将编写使用这三种机制的示例应用程序。

说明 要将数据持久存储在iPhone上，并不是只有属性列表、对象归档以及SQLite3这几种方法。它们只是三种最常用且最简单的方法。你可以始终选择使用传统的C I/O调用（如fopen()）读取和写入数据。也可以使用Cocoa的低级文件管理工具。在这两种情况下，持久存储数据都将导致编写更多代码，而且这样做是不必要的，但是如果需要的话，也可以使用这些工具。

11.1 应用程序的沙盒

本章介绍的三种数据持久性的机制都具备一个重要的共有元素，即应用程序的/Documents文件夹。每个应用程序都有自己的/Documents文件夹（但Apple应用程序例外，比如说Settings）并且应用程序仅能读取各自的/Documents目录中的内容。

为了给你提供一些上下文，我们看一下iPhone上的应用程序。打开一个Finder窗口并导航到主目录。然后在该目录中，向下展开到Library/Application Support/iPhone Simulator/User/。此时，

你应该看到5个子文件夹，其中一个文件夹名为Applications（参见图11-1）。

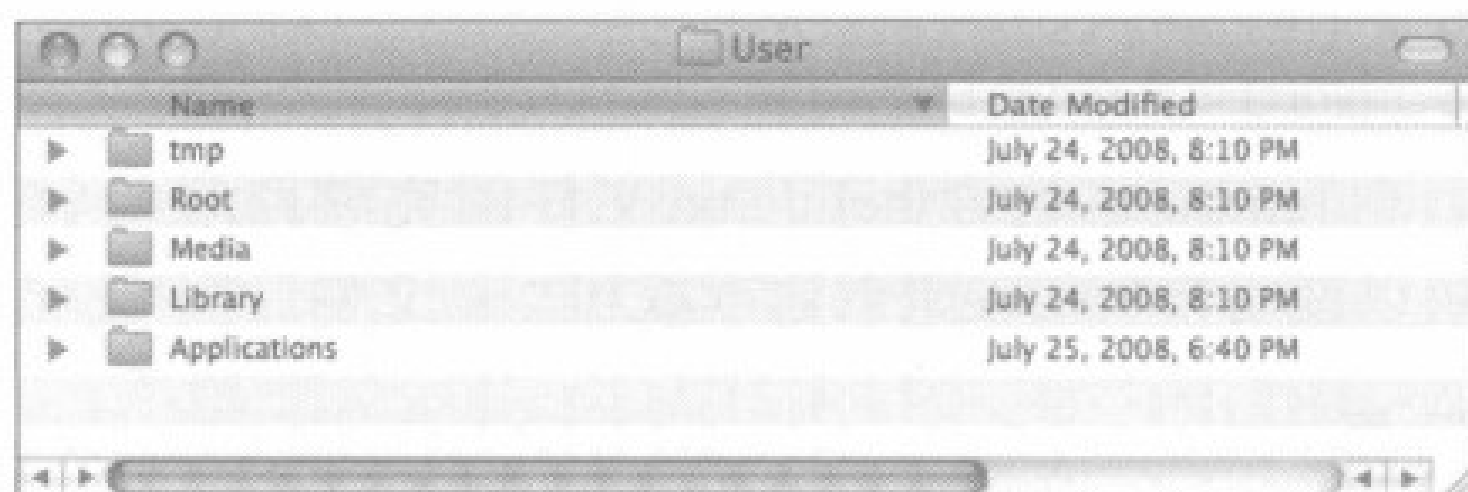


图11-1 显示Applications文件夹的User目录的布局

显而易见，Applications文件夹就是iPhone存储其应用程序的文件夹。如果打开Applications文件夹，可以看到一系列文件夹和文件，它们的名称是较长的字符串。这些名称都是由Xcode自动生成的。这些文件夹中的每个文件夹都包含一个应用程序及其支持的文件夹。

.sb文件包含仿真器用于启动具备相同名称的程序的设置。你应该从不需要接触这些内容。但是如果打开其中一个应用程序目录，应该会看到一些比较熟悉的内容。在这里，可以找到你构建的其中一个iPhone应用程序及其支持的3个文件夹：Documents、Library和tmp。应用程序将其数据存储在Documents中（但基于NSUserDefaults的首选设置除外，它存储在Library/Preferences文件夹中）。tmp目录供应用程序存储临时文件。当iPhone执行同步时，iTunes不会备份/tmp中的文件，但当不再需要这些文件时，应用程序需要负责删除/tmp中的文件，以避免占用文件系统的空间。

11.1.1 获取 Documents 目录

既然我们的应用程序位于一个名称看上去是随机名称的文件夹中，那么如何检索Documents目录的完整路径以便读取和写入文件呢？实际上这非常容易。可以使用名为NSSearchPathForDirectoriesInDomain的C函数来查找各种目录。它是Foundation函数，因此它可以与Cocoa for Mac OS X共享。它的很多可用选项都是专门为OS X设计的，在iPhone上不会返回任何值。其原因在于，这些位置并不存在于iPhone（Downloads文件夹）上，或者你的应用程序由于iPhone的沙盒机制而没有访问该位置的权限。

下面是检索文档目录路径的一些代码：

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
    NSUserDomainMask, YES);
NSString *documentsDirectory = [paths objectAtIndex:0];
```

常量NSDocumentDirectory表明我们正在查找Documents目录的路径。第二个常量NSUserDomainMask表明我们希望将搜索限制于我们应用程序的沙盒。在Mac OS X中，此常量表示我们希望该函数查看用户的主目录，这解释了其名称古怪的原因。

尽管返回了一个匹配路径的数组，但是我们可以算出数组中位于索引0处的Documents目录。为什么呢？我们知道每个应用程序只有一个Documents目录，因此只有一个目录符合我们指定的条件。我们可以通过在刚刚检索到的路径的结尾附加另一个字符串来创建一个文件名，用于执行读取或写入操作。我们将使用专为该目的设计的NSString方法，即stringByAppendingPath-

Component:，如下所示：

```
NSString *filename = [documentsDirectory  
    stringByAppendingPathComponent:@"theFile.txt"];
```

完成此调用之后，filename将包含theFile.txt文件的完整路径，该文件位于应用程序的Documents目录，我们可以使用filename来创建读取和写入文件。

11.1.2 获取 tmp 目录

获取对应用程序临时目录的引用比获取对Documents目录的引用更加容易。名为NSTemporaryDirectory()的Foundation函数将返回一个字符串，该字符串包含到应用程序临时目录的完整路径。若要为将存储在临时目录中的某个文件创建一个文件名，我们首先要找到临时目录：

```
NSString *tempPath = NSTemporaryDirectory();
```

然后，通过在该路径的结尾附加一个文件名在该目录中创建一个到该文件的路径，例如：

```
NSString *tempFile = [tempPath  
    stringByAppendingPathComponent:@"tempFile.txt"];
```

11.2 文件保存策略

在本章中，我们将介绍实现数据持久性的三种不同方法。所有这三种方法都使用iPhone的文件系统。

如果是SQLite3，你将创建一个SQLite3数据库文件，并让SQLite3负责存储和检索数据。使用其他两种持久性机制，即属性列表和归档，你将需要考虑是将数据存储在一个文件中，还是存储在多个文件中。

11.2.1 单个文件持久性

使用单个文件是最简单的方法，并且对于许多应用程序，这是非常容易接受的方法。首先创建一个根对象，通常是NSArray或NSDictionary，但也可以让根对象基于某个自定义类。接下来，使用程序数据填充根对象。需要保存时，代码会将该根对象的全部内容重新写入单个文件。应用程序在启动时会将该文件的全部内容读入内存，并在退出时注销全部内容。这就是本章将要使用的方法。

使用单个文件的缺点在于，你必须将全部应用程序数据加载到内存中，并且必须将所有数据全部写入文件系统，即使更改再少也是如此。如果应用程序不可能管理超过几兆字节的数据，则此方法可能非常好，而且它这么简单，一定会使我们的生活更加轻松。

11.2.2 多个文件持久性

使用多个文件一定非常复杂。假设你要编写一个电子邮件应用程序，该程序将每封电子邮件都存储在其自己的文件中。该方法具有明显的优势。它允许应用程序仅加载用户请求的数据（另一种形式的延迟加载），当用户进行更改时，只需保存更改的文件。此方法允许开发人员在收到

内存不足通知时释放内存，因为可以刷新用于存储用户当前未查看的数据的任何内存，并且只需在下次需要时从文件系统重新加载即可。多个文件持久性的缺点是它大大增加了应用程序的复杂性。到目前为止，我们还是坚持使用单个文件持久性。

11.3 持久保存应用程序数据

本节介绍属性列表、对象归档以及SQLite3这三种持久性方法的详细信息。我们将依次探讨每种方法，并构建一个分别使用各机制将一些数据保存到iPhone文件系统的应用程序。我们将从属性列表开始。

属性列表序列化

我们的许多应用程序都使用了属性列表，比如说使用属性列表来指定应用程序首选项。属性列表非常方便，因为可以使用Xcode或Property List Editor应用程序手动编辑它们，并且只要字典或数组仅包含特定可序列化的对象，就可以将NSDictionary和NSArray实例写入属性列表以及从属性列表创建它们。序列化对象已被转换为字节流，以便于存储到文件中或通过网络进行传输。尽管可以让任何对象可序列化，但是只能将某些对象放置到某个集合类（如NSDictionary或NSArray）中，然后使用该集合类的writeToFile:atomically:方法将它们存储到属性列表。可以按照该方法进行序列化的Objective-C类如下：

- ☐ NSArray
- ☐ NSMutableArray
- ☐ NSDictionary
- ☐ NSMutableDictionary
- ☐ NSData
- ☐ NSMutableData
- ☐ NSString
- ☐ NSMutableString
- ☐ NSNumber
- ☐ NSDate

如果可以只从这些对象构建数据模型，则可以使用属性列表轻松保存和加载数据。实际上，我们已经在很多示例应用程序中使用这种机制来提供示例数据了。

如果你打算使用属性列表持久保存应用程序数据，则可以使用NSArray或NSDictionary容纳需要持久保存的数据。假设你放到NSArray或NSDictionary中的所有对象都是可序列化对象，则可以通过对字典或数组实例调用writeToFile:atomically:方法来编写属性列表，如下所示：

```
[myArray writeToFile:@"~/some/file/location/output.plist" atomically:YES];
```

说明 如果愿意，可以通过atomically参数让该方法将数据写入辅助文件，而不是写入指定位置。

成功写入该文件之后，该辅助文件将被复制到第一个参数指定的位置。这是更安全的写入文件的方法，因为如果应用程序在保存期间崩溃，则现有文件（如果有）不会被破坏。尽管这样增加了一点开销，但是多数情况下，它还是值得的。

属性列表方法的一个问题是无法将自定义对象序列化到属性列表中。也不能使用通过Cocoa Touch交付且未在之前的可序列化对象列表中指定的其他类，这意味着无法直接使用NSURL、UIImage和UIColor等类。

不能序列化这些对象还意味着你无法轻松创建派生或计算的属性（例如，某两个属性之和的属性），并且必须将实际上应该包含在模型类中的某些代码移动到控制器类。而且，这些限制也适用于简单数据模型和简单应用程序。但是多数情况下，如果创建了专用的模型类，则应用程序更容易维护。

但是，在复杂的应用程序中，简单的属性列表仍然非常有用。它们是将数据包含在应用程序中的最佳方法。例如，当应用程序包含一个选取器时，将项目列表包含到选取器中的最佳方法是，创建一个属性列表文件并将其包含在项目的Resources文件夹中，这会将其编译到应用程序中。

让我们构建一个使用属性列表存储数据的简单应用程序。

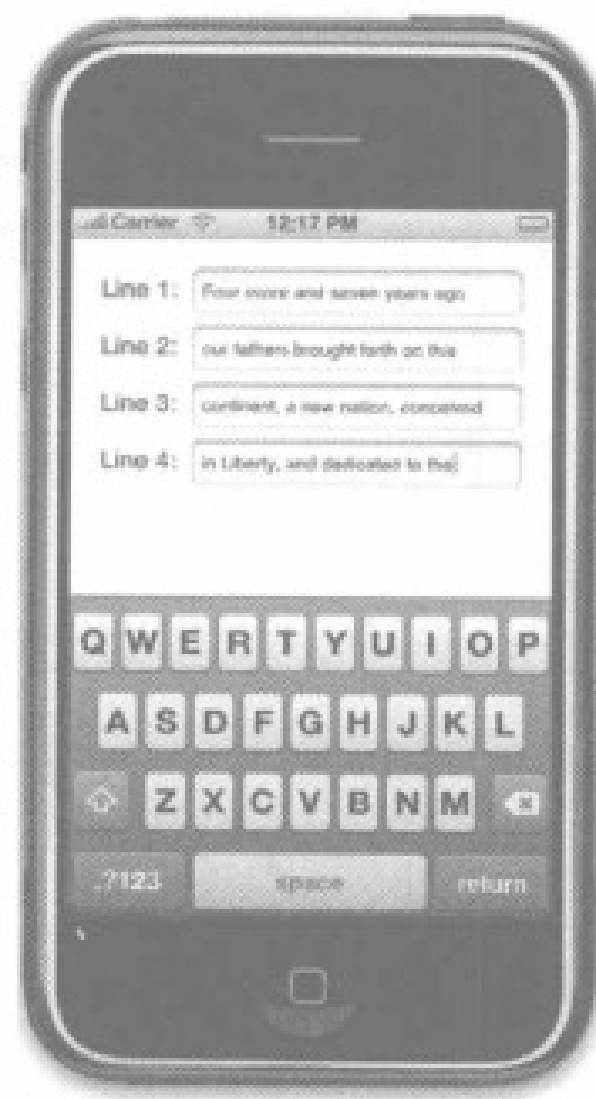


图11-2 属性列表应用程序

11.4 持久性应用程序

我们将构建一个程序，该程序允许你在4个文本字段中输入数据，应用程序退出时会将这些字段保存到属性列表文件，然后在下次启动时从该属性列表文件中重新加载该数据（参见图11-2）。

说明 在本章的应用程序中，我们不会花费时间设置任何用户界面细节（我们曾经这样做过）。例如，轻击return键既不会离开键盘，也不会带你进入下一字段。如果你希望向应用程序中添加此修饰，则这样做是最佳做法，但这不是本章的主题，因此我们不会对此进行介绍。

11.4.1 创建持久性项目

在Xcode中，使用基于视图的应用程序模板创建一个新项目，并用名称Persistence保存该项目。该项目包含构建应用程序所需的所有文件，这样我们可以潜心研究特定的事情。稍后，我们将构建一个具有4个文本字段的视图。创建所需的输出口，然后进入Interface Builder。展开Classes文件夹。然后单击PersistenceViewController.h文件，并进行以下更改：

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

```
#define kFilename @"data.plist"
```

```

@interface PersistenceViewController : UIViewController {
    IBOutlet UITextField *field1;
    IBOutlet UITextField *field2;
    IBOutlet UITextField *field3;
    IBOutlet UITextField *field4;
}
@property (nonatomic, retain) UITextField *field1;
@property (nonatomic, retain) UITextField *field2;
@property (nonatomic, retain) UITextField *field3;
@property (nonatomic, retain) UITextField *field4;
- (NSString *)dataFilePath;
- (void)applicationWillTerminate:(NSNotification *)notification;
@end

```

除了定义4个文本字段输出口之外，我们还为将要使用的文件名定义了一个常量，以及两个其他的方法。一个方法是dataFilePath，该方法可以将kFilename串联到Documents目录的路径，以创建并返回数据文件的完整路径名。第二个方法是applicationWillTerminate:（之后我们将讨论该方法），应用程序将在退出时调用该方法，并且将数据保存到属性列表文件。

接下来，展开Resources文件夹，双击PersistenceViewController.xib，在Interface Builder中将其打开。

11.4.2 设计持久性应用程序视图

Interface Builder出现之后，应该还会打开View窗口。如果该窗口未打开，则双击View图标打开它。从库中拖出一个Text Field，然后根据右上角的蓝色引导线放置它。将其向左展开以便它到达大约横跨窗口路线的三分之二，然后按⌘1打开属性检查器。取消选中标签为Clear When Editing Begins的复选框。

接下来，按下option键并向下拖动文本框，这将创建该文本框的副本。重复此步骤两次以上，以便拥有4个文本字段。现在，将4个标签拖到窗口并使用图11-3作为布局和设计指导。注意，我们已将文本字段放置在视图的顶部，以便为键盘留出空间。

添加4个文本字段和标签之后，按下Control的同时，将File's Owner图标拖放到每个文本字段中。将最上面的文本字段连接到名为field1的输出口，将下面一个文本字段连接到field2，将第3个文本字段连接到field3，将最底下的文本字段连接到field4。最后，保存、关闭PersistenceViewController.xib，并返回Xcode。

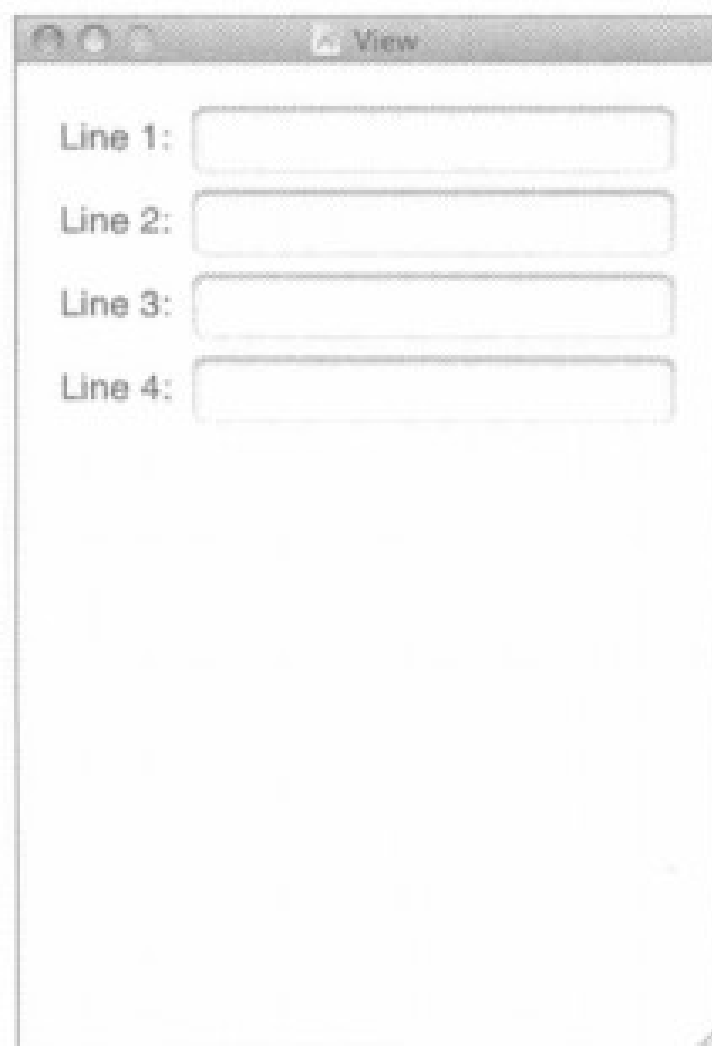


图11-3 设计持久性应用程序的视图

11.4.3 编辑持久性类

单击PersistenceViewController.m并进行以下更改，完成键入之后，我们将对此进行讨论：

```
#import "PersistenceViewController.h"

@implementation PersistenceViewController
@synthesize field1;
@synthesize field2;
@synthesize field3;
@synthesize field4;

- (NSString *)dataFilePath
{
    NSArray *paths = NSSearchPathForDirectoriesInDomains(
        NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];
    return [documentsDirectory stringByAppendingPathComponent:kFilename];
}

- (void)applicationWillTerminate:(NSNotification *)notification
{
    NSMutableArray *array = [[NSMutableArray alloc] init];
    [array addObject:field1.text];
    [array addObject:field2.text];
    [array addObject:field3.text];
    [array addObject:field4.text];
    [array writeToFile:[self dataFilePath] atomically:YES];
    [array release];
}

#pragma mark -

- (void)viewDidLoad {

    NSString *filePath = [self dataFilePath];
    if ([[NSFileManager defaultManager] fileExistsAtPath:filePath])
    {
        NSArray *array = [[NSArray alloc] initWithContentsOfFile:filePath];
        field1.text = [array objectAtIndex:0];
        field2.text = [array objectAtIndex:1];
        field3.text = [array objectAtIndex:2];
        field4.text = [array objectAtIndex:3];
        [array release];
    }

    UIApplication *app = [UIApplication sharedApplication];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(applicationWillTerminate:)
        name:UIApplicationWillTerminateNotification
        object:app];
    [super viewDidLoad];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
```

```

    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [field1 release];
    [field2 release];
    [field3 release];
    [field4 release];
    [super dealloc];
}

@end

```

我们添加的第一个方法 `dataFilePath` 用于返回数据文件的完整路径名。它通过查找文档目录并对其附加 `kFilename` 来执行该操作。我们将通过需要加载或保存数据的任何代码来调用该方法。

```

- (NSString *)dataFilePath
{
    NSArray *paths = NSSearchPathForDirectoriesInDomains(
        NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];
    return [documentsDirectory stringByAppendingPathComponent:kFilename];
}

```

第二个新方法称为 `applicationWillTerminate:`。注意，它使用指向 `NSNotification` 的指针作为参数。`applicationWillTerminate:` 是通知方法，并且所有通知都使用一个 `NSNotification` 实例作为其参数。

通知是一种对象可以用于彼此通信的轻量级机制。任何对象都可以定义一个或多个通知，以发布到应用程序通知中心，它是一个单独的对象，它存在的唯一目的是在对象之间传递这些通知。通常，通知是发生了某些事件的指示，被传递的发布通知的对象在其文档中包含一个通知列表。例如，如果查看图11-4，则可以看到发布很多通知的 `UIApplication` 类。

通常，大多数通知的作用都可以从其名称中明显看出，但是如果你发现某个通知的作用不明确，可以查看包含详细信息的文档。我们的应用程序需要在退出之前保存其数据，因此我们对 `UIApplicationWillTerminate-`

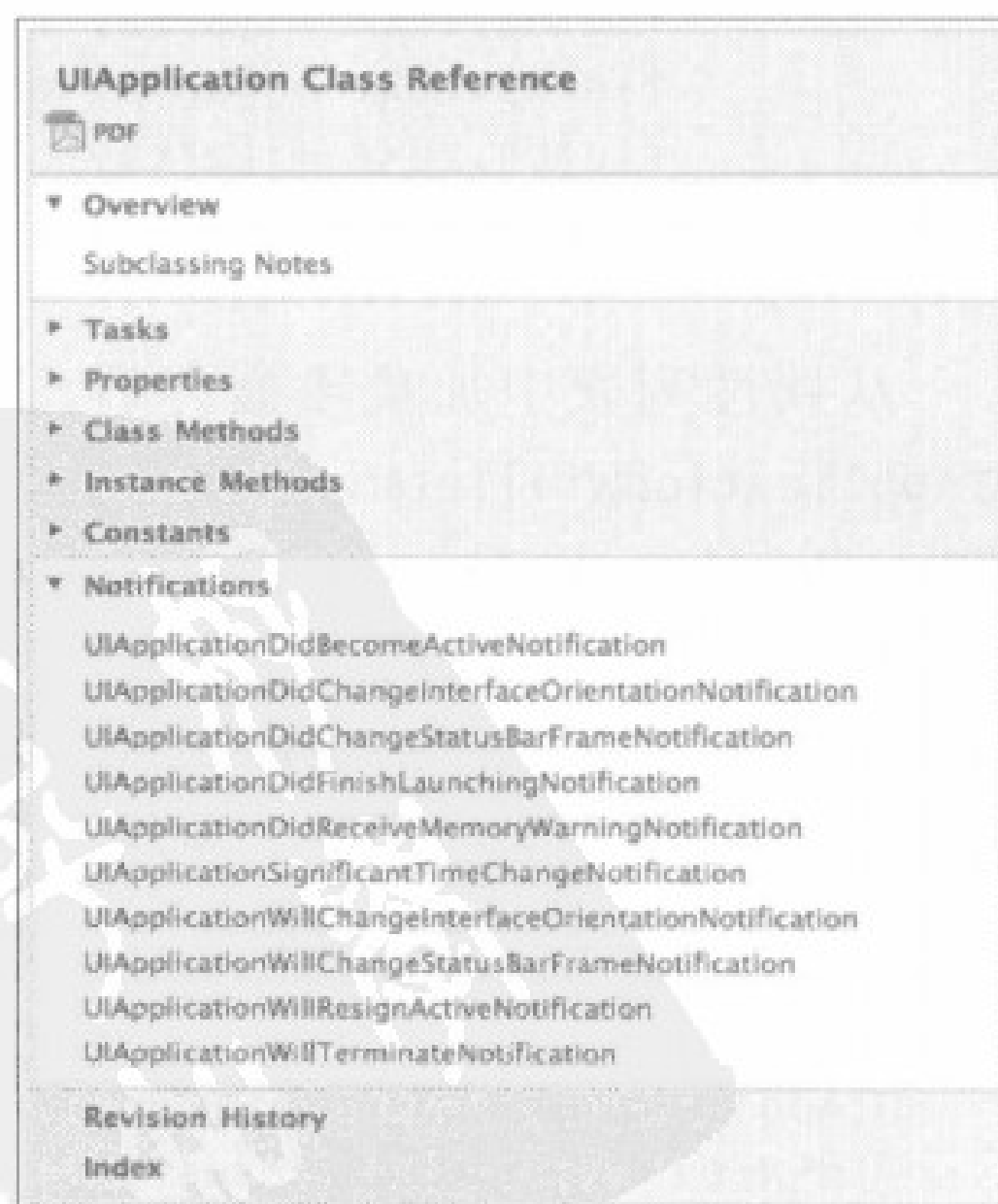


图11-4 UIApplication文档列出了其发布的所有通知

Notification通知更加感兴趣。随后在编写viewDidLoad方法时，我们将订阅该通知并告知通知中心调用该方法：

```
- (void)applicationWillTerminate:(NSNotification *)notification
{
    NSMutableArray *array = [[NSMutableArray alloc] init];
    [array addObject:field1.text];
    [array addObject:field2.text];
    [array addObject:field3.text];
    [array addObject:field4.text];
    [array writeToFile:[self dataFilePath] atomically:YES];
    [array release];
}
```

该方法本身非常简单。我们创建了一个可变数组，将4个字段中的文本添加到该数组中，然后将该数组的内容写入一个属性列表文件。此处所有要做的就是使用属性列表保存我们的数据。

在viewDidLoad方法中，我们只做几件事情。第一件事是检查数据文件是否存在。如果不存在，我们不希望尝试加载它。如果文件存在，就用该文件的内容实例化数组，然后将该数组中的对象复制到4个文本字段。由于数组是按顺序排列的列表，因此只要根据保存顺序来复制数组，就一定可以确保正确的字段获得正确的值。

```
- (void)viewDidLoad {

    NSString *filePath = [self dataFilePath];
    if ([[NSFileManager defaultManager] fileExistsAtPath:filePath])
    {
        NSArray *array = [[NSArray alloc] initWithContentsOfFile:filePath];
        field1.text = [array objectAtIndex:0];
        field2.text = [array objectAtIndex:1];
        field3.text = [array objectAtIndex:2];
        field4.text = [array objectAtIndex:3];
        [array release];
    }
}
```

从属性列表中加载数据之后，我们获得了对应用程序实例的引用，并使用该引用订阅UIApplicationWillTerminateNotification，使用默认的NSNotificationCenter实例以及一个名为addObserver:selector:name:object:的方法。我们传递一个observer，即self，这意味着PersistenceViewController是需要通知的对象。对于selector，我们将一个selector传递给刚才编写的applicationWillTerminate:方法，告知通知中心在发布该通知时调用该方法。第三个参数name:是我们对接收感兴趣的通知的名称，最后一个参数object:是我们对从中获取通知感兴趣的对象。如果我们为最后一个参数传递nil，则通知我们发布UIApplicationWillTerminateNotification的时间和方式。

```
UIApplication *app = [UIApplication sharedApplication];
[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(applicationWillTerminate:)
    name:UIApplicationWillTerminateNotification
    object:app];
```

订阅通知之后，超类正好可以借此机会响应viewDidLoad。

```
[super viewDidLoad];  
}
```

这样做没有什么不好，不是吗？当主视图完成加载后，我们查找属性列表文件。如果该文件存在，则将其中的数据复制到文本字段中。接下来，应用程序将在终止时通知我们。当应用程序终止时，我们收集4个文本字段中的值，将它们粘贴在可变数组中，并将该可变数组写入属性列表。

为什么没有编译和运行应用程序呢？应该先构建，然后在仿真器中启动。实现之后，你应该能够在4个文本字段中的任何一个字段中键入文本。在其中键入某些内容后，按home按钮（仿真器窗口底部具有圆角矩形的圆形按钮）。按home按钮非常重要。如果你刚刚退出仿真器，即等同于强制退出应用程序，则从不会收到应用程序正在终止的通知，并且绝对不会保存你的数据。

属性列表序列化非常实用且易于使用，但它有一点限制，它只能将选择的一小部分对象存储在属性列表中。下面让我们看看比较强大的方法。

11.4.4 对模型对象进行归档

在第9章的最后一个部分中，我们在构建Presidents数据模型对象之后，给出了一个使用NSCoder加载归档数据的过程示例。在Cocoa世界中，术语“归档”是指另一种形式的序列化，但它是任何对象都可以实现的更常规的类型。专门编写用于保存数据（模型对象）的任何对象都应该支持归档。使用对模型对象进行归档的技术可以轻松将复杂的对象写入文件，然后再从中读取它们。只要在类中实现的每个属性都是标量（如int或float）或都是符合NSCoding协议的某个类的实例，你就可以对你的对象进行完整归档。由于大多数支持存储数据的Foundation和Cocoa Touch类都符合NSCoding，因此对于大多数类来说，归档相对而言比较容易实现。

尽管对使用归档没有严格要求，但是应该与NSCoding一起实现另一个协议，即NSCopying协议，该协议允许复制对象。能够复制对象使你在使用数据模型对象时具备了更多的灵活性。例如，在第9章的Presidents应用程序中，我们不必编写复杂代码来存储用户所做的更改以便可以处理Cancel和Save按钮，我们可以生成president对象的副本，并将更改存储在该副本中。如果用户轻击Save，我们只需复制更改后的版本来替换原来的版本。

符合NSCoding

NSCoding协议声明了两个方法，这两个方法都是必需的。一个方法将对象编码到归档中；另一个方法通过对归档解码来创建一个新对象。这两个方法都传递一个NSCoder实例，使用方式与上一章中的NSUserDefaults非常相似。你可以使用键-值编码对对象和标量进行编码和解码。

对某个对象编码的方法可能类似于以下内容：

```
- (void)encodeWithCoder:(NSCoder *)encoder  
{  
    [encoder encodeObject:foo forKey:kFooKey];  
    [encoder encodeObject:bar forKey:kBarKey];  
    [encoder encodeInt:someInt forKey:kSomeIntKey];  
    [encoder encodeFloat:someFloat forKey:kSomeFloatKey];  
}
```


要在对象中支持归档，我们必须使用适当的编码方法将每个实例变量编码成encoder。因此，我们需要实现一个方法来初始化NSCoder中的对象，以还原以前归档的对象。实现initWithCoder:方法比实现encodeWithEncoder:方法稍微复杂一些。如果你直接对NSObject进行子类化，或者对某些不符合NSCoding的其他类进行子类化，则你的方法将类似以下内容：

```
- (id)initWithCoder:(NSCoder *)decoder
{
    if (self = [super init])
    {
        self.foo = [decoder decodeObjectForKey:kFooKey];
        self.bar = [decoder decodeObjectForKey:kBarKey];
        self.someInt = [decoder decodeIntForKey:kSomeIntKey];
        self.someFloat = [decoder decodeFloatForKey:kAgeKey];
    }
    return self;
}
```

该方法使用[super init]初始化对象实例，如果初始化成功，则它通过解码在NSCoder的实例中传递的值来设置其属性。当为某个具有超类且符合NSCoding的类实现NSCoding时，initWithCoder:方法应稍有不同。它不再对super调用init，而是调用initWithCoder:，例如：

```
- (id)initWithCoder:(NSCoder *)encoder
{
    if (self = [super initWithCoder:decoder])
    {
        self.foo = [decoder decodeObjectForKey:kFooKey];
        self.bar = [decoder decodeObjectForKey:kBarKey];
        self.someInt = [decoder decodeIntForKey:kSomeIntKey];
        self.someFloat = [decoder decodeFloatForKey:kAgeKey];
    }
    return self;
}
```

基本就这些。只要你实现这两个方法，就可以对所有对象的属性进行编码和解码，然后便可以对你的对象进行归档，并且可以写入归档中以及从归档中读取。

11.4.5 实现 NSCopying

如前所述，符合NSCopying对于任何数据模型对象来说都是一个非常好的主意。NSCopying有一个copyWithZone:方法，可用于复制对象。实现NSCopying与实现initWithCoder:非常相似。你只需创建一个同一类的新实例，然后将该新实例的所有属性都设置为与该对象属性相同的值。此处copyWithZone:方法的内容类似于以下内容：

```
- (id)copyWithZone:(NSZone *)zone
{
    MyClass *copy = [[[self class] allocWithZone: zone] init];
    copy.foo = [self.foo copy];
    copy.bar = [self.bar copy];
}
```

```

        copy.someInt = self.someInt;
        copy.someFloat = self.someFloat;
        return copy;
    }

```

注意，我们并没有释放或自动释放创建的新对象。复制的对象都被隐式保留，并且应该在名为copy的代码中释放或自动释放。

1. 对数据对象进行归档

从符合NSCoding的一个或多个对象创建归档相对比较容易。首先，创建一个NSMutableData实例，用于包含编码的数据，然后创建一个NSKeyedArchiver实例，用于将对象归档到此NSMutableData实例中：

```

NSMutableData *data = [[NSMutableData alloc] init];
NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc]
    initWithWritingWithMutableData:data];

```

创建这两个实例之后，我们使用键-值编码来对希望包含在归档中的所有对象进行归档，例如：

```
[archiver encodeObject:myObject forKey:@"keyValueString"];
```

对所有要包含的对象进行编码之后，我们只需告知归档程序已经完成了这些操作，将NSMutableData实例写入文件系统，并对对象进行内存清理。

```

[archiver finishEncoding];
BOOL success = [data writeToFile:@"/path/to/archive" atomically:YES];
[archiver release];
[data release];

```

如果写入文件时出现错误，会将success设置为NO。如果success为YES，则数据已成功写入指定文件。从该归档创建的任何对象都将是过去写入该文件的对象的精确副本。

2. 对数据对象取消归档

要从归档重新组成对象，我们需要经历类似的过程。从归档文件创建一个NSData实例，并创建一个NSKeyedUnarchiver以对数据进行解码：

```

NSData *data = [[NSData alloc] initWithContentsOfFile:path];
NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc]
    initWithReadingWithData:data];

```

然后，使用之前用于对对象进行归档的相同密钥从解压程序中读取对象：

```
self.object = [unarchiver decodeObjectForKey:@"keyValueString"];
```

说明 自动释放由decodeObjectForKey:返回的对象，因此如果需要身边经常备用该对象，则需要保留该对象。将其分配给使用retain关键字声明的属性通常可以为我们完成这个任何，但是如果你未将其分配给属性，并且需要让该对象位于当前事件循环的结尾，则需要保留该对象。

最后，我们告知归档程序已经完成了该操作，并且进行内存清理：

```
[unarchiver finishDecoding];
[unarchiver release];
[data release];
```

如果你感觉对归档有点不知所措，不要担心，实际上它非常简单。我们将为Persistence应用程序添加归档功能，以便于你理解其内部原理。完成几次之后，归档将变成第二天性，因为你所有实际执行的操作就是使用键-值编码存储和检索对象的属性。

11.5 归档应用程序

我们将改进Persistence应用程序，让它使用归档而不是属性列表。我们将对Persistence源代码进行一些非常重要的更改，因此你可能希望在继续之前为项目创建一个副本。

11.5.1 实现 FourLines 类

准备好继续执行操作并在Xcode中打开Persistence项目之后，单击Classes文件夹，并按⌘N或从File菜单中选择New File...。当出现新建文件向导时，选择NSObject subclass，并将该文件命名为FourLines.m，确保选中创建头文件的复选框。该文件将是我们的数据模型，并且它将容纳当前存储在属性列表应用程序的字典中的数据。

单击FourLines.h，并进行以下更改：

```
#define kField1Key @"Field1"
#define kField2Key @"Field2"
#define kField3Key @"Field3"
#define kField4Key @"Field4"

#import <UIKit/UIKit.h>

@interface FourLines : NSObject <NSCoding, NSCopying> {
    NSString *field1;
    NSString *field2;
    NSString *field3;
    NSString *field4;
}
@property (nonatomic, retain) NSString *field1;
@property (nonatomic, retain) NSString *field2;
@property (nonatomic, retain) NSString *field3;
@property (nonatomic, retain) NSString *field4;
@end
```

这是一个非常简单数据模型类，它具有4个字符串属性。注意，我们已经让该类符合NSCoding和NSCopying协议了。现在切换到FourLines.m，并添加以下代码。

```
#import "FourLines.h"

@implementation FourLines
@synthesize field1;
@synthesize field2;
```

```

@synthesize field3;
@synthesize field4;
#pragma mark NSCoder
- (void)encodeWithCoder:(NSCoder *)encoder
{
    [encoder encodeObject:field1 forKey:kField1Key];
    [encoder encodeObject:field2 forKey:kField2Key];
    [encoder encodeObject:field3 forKey:kField3Key];
    [encoder encodeObject:field4 forKey:kField4Key];
}
- (id)initWithCoder:(NSCoder *)decoder
{
    if (self = [super init])
    {
        self.field1 = [decoder decodeObjectForKey:kField1Key];
        self.field2 = [decoder decodeObjectForKey:kField2Key];
        self.field3 = [decoder decodeObjectForKey:kField3Key];
        self.field4 = [decoder decodeObjectForKey:kField4Key];
    }
    return self;
}

#pragma mark -
#pragma mark NSCopying
- (id)copyWithZone:(NSZone *)zone
{
    FourLines *copy = [[[self class] allocWithZone: zone] init];
    field1 = [self.field1 copy];
    field2 = [self.field2 copy];
    field3 = [self.field3 copy];
    field4 = [self.field4 copy];

    return copy;
}
@end

```

我们刚才实现了符合NSCoding和NSCopying所需的所有方法。我们在encodeWithCoder:中对所有4个属性进行编码，并在initWithCoder:中使用相同的4个键值对这些属性进行解码。在copyWithZone:中，我们创建了一个新的FourLines对象，并将所有4个字符串复制到其中。看见了吗？这一点也不难。

11.5.2 实现 PersistenceViewController 类

创建可归档的数据对象之后，让我们使用它来持久存储应用程序数据。单击PersistenceViewController.h，并进行以下更改：

```

#import <UIKit/UIKit.h>

#define kFilename          @"data.plist"

```

```

#define kFilename      @"archive"
#define kDataKey       @"Data"

@interface PersistenceViewController : UIViewController {
    IBOutlet UITextField *field1;
    IBOutlet UITextField *field2;
    IBOutlet UITextField *field3;
    IBOutlet UITextField *field4;
}
@property (nonatomic, retain) UITextField *field1;
@property (nonatomic, retain) UITextField *field2;
@property (nonatomic, retain) UITextField *field3;
@property (nonatomic, retain) UITextField *field4;
- (NSString *)dataFilePath;
- (void)applicationWillTerminate:(NSNotification *)notification;
@end

```

此处所有要做的就是指定一个新的文件名，以便我们的程序不会将旧的属性列表作为归档加载。我们还定义了一个新的常量，该常量将是我们用于对对象进行编码和解码的键值。

让我们切换到PersistenceViewController.m，并进行以下更改：

```

#import "PersistenceViewController.h"
#import "FourLines.h"

@implementation PersistenceViewController
@synthesize field1;
@synthesize field2;
@synthesize field3;
@synthesize field4;
- (NSString *)dataFilePath
{
    NSArray *paths = NSSearchPathForDirectoriesInDomains(
        NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];
    return [documentsDirectory stringByAppendingPathComponent:kFilename];
}
- (void)applicationWillTerminate:(NSNotification *)notification
{
    NSMutableArray *array = [[NSMutableArray alloc] init];
    [array addObject:field1.text];
    [array addObject:field2.text];
    [array addObject:field3.text];
    [array addObject:field4.text];
    [array writeToFile:[self dataFilePath] atomically:YES];
    [array release];

    FourLines *fourLines = [[FourLines alloc] init];
    fourLines.field1 = field1.text;
    fourLines.field2 = field2.text;
    fourLines.field3 = field3.text;
    fourLines.field4 = field4.text;
}

```

```

NSMutableData *data = [[NSMutableData alloc] init];
NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc]
    initWithWritingWithMutableData:data];
[archiver encodeObject:fourLines forKey:kDataKey];
[archiver finishEncoding];
[data writeToFile:[self dataFilePath] atomically:YES];
[fourLines release];
[archiver release];
[data release];
}
#pragma mark -
- (void)viewDidLoad {

    NSString *filePath = [self dataFilePath];
    if ([[NSFileManager defaultManager] fileExistsAtPath:filePath])
    {
        NSMutableArray *array = [[NSMutableArray alloc]
            initWithContentsOfFile:filePath];
        field1.text = [array objectAtIndex:0];
        field2.text = [array objectAtIndex:1];
        field3.text = [array objectAtIndex:2];
        field4.text = [array objectAtIndex:3];
        [array release];

        NSData *data = [[NSMutableData alloc]
            initWithContentsOfFile:[self dataFilePath]];
        NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc]
            initWithReadingWithData:data];
        FourLines *fourLines = [unarchiver decodeObjectForKey:kDataKey];
        [unarchiver finishDecoding];

        field1.text = fourLines.field1;
        field2.text = fourLines.field2;
        field3.text = fourLines.field3;
        field4.text = fourLines.field4;

        [unarchiver release];
        [data release];
    }
    UIApplication *app = [UIApplication sharedApplication];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(applicationWillTerminate:)
        name:UIApplicationWillTerminateNotification
        object:app];
    [super viewDidLoad];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:

```

```

    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [field1 release];
    [field2 release];
    [field3 release];
    [field4 release];
    [super dealloc];
}
@end

```

实际更改并不多，并且此新版本需要比属性列表序列化多实现几行代码，因此你可能想知道使用归档是否比使用序列化属性列表更有优势。对于该应用程序，答案非常简单：实际上并非如此。但是回想第9章中的最后一个示例，在该示例中，我们允许用户编辑总统列表，每个总统共有4个不同的可以编辑的字段。要使用属性列表处理对总统列表的归档，需要涉及迭代总统列表、为每个总统创建一个NSDictionary实例、将每个字段中的值复制到NSDictionary实例、将该实例添加到另一个数组，然后将该数组写入属性列表文件。当然，这是假设我们将自己局限于只使用可序列化的属性。如果不这样，就根本不能使用属性列表序列化。

另一方面，如果我们拥有一个可归档对象的数组（如我们刚才构建的FourLines类），则可以通过对数组实例本身进行归档来归档整个数组。归档集合类（如NSArray）时，也归档其包含的所有对象。只要放入数组或字典中的每个对象都符合NSCoding，你就可以归档数组或字典并还原它。另外，当你对其进行归档时，其中的所有对象都将位于已还原数组或字典中。换句话说，该方法可以适当伸缩，因为无论你添加多少对象，将这些对象写入磁盘的方式（假设你使用单个文件持久性）都完全相同。但使用属性列表，工作量随着添加的每个对象而增加。

11.6 使用 iPhone 的嵌入式 SQLite3

我们将要讨论的第三个持久性选项是iPhone的嵌入式SQL数据库，名为SQLite3。SQLite3在存储和检索大量数据方面非常有效。它能够对数据进行复杂的聚合，与使用对象执行这些操作相比，获得结果的速度更快。例如，如果应用程序需要计算应用程序中所有对象中特殊字段的总和，或者如果需要只符合特定条件的对象的总和，SQLite3将可以执行该操作，而不会将每个对象加载到内存中。从SQLite3获取聚合比将所有对象加载到内存中，然后计算它们值的总和要快几个数量级。作为一个羽翼丰满的嵌入式数据库，SQLite3包含使其速度更快（例如，通过创建

可以加快查询速度的表索引)的工具。

SQLite3使用SQL (Structured Query Language, 结构化查询语言)。SQL是用于与关系数据库交互的标准语言,并且它是具有其自己的语法的语言,很多细微之处已超出本书的范围。整本书都是采用SQL (实际上有几十个)语法以及SQLite本身编写的。因此,如果你还不了解SQL并且想在应用程序中使用SQLite3,则需要提前做点工作。我们将介绍如何在iPhone应用程序中进行设置并与SQLite数据库交互,并且你将在本章中看到某些基本语法。但是,要真正地充分利用SQLite3,你就需要进行某些额外的研究和探讨。

提示 有关“SQL”和“SQLite”的发音,有两派意见。大多数正式文档将“SQL”读为“Ess-Queue-Ell”,将“SQLite”读为“Ess-Queue-Ell-Light”。很多人分别将它们读为“Sequel”和“Sequel Light”。

关系数据库(包括SQLite3)和面向对象的编程语言使用完全不同的方法来存储和组织数据。这些方法非常不同,以至于出现了用于在两者之间转换的各种技术以及很多库和工具。这些不同的技术统称为对象关系映射(object-relational mapping, ORM)。Core Data为Mac OS X提供了基于Cocoa的ORM解决方案。但是,iPhone没有任何ORM库或工具。因此,程序员有责任设计自己的方法将数据库中的信息映射到对象以及再映射回来。本章将重点介绍基础知识,包括设置SQLite3,创建容纳数据的表,保存数据以及从数据库中检索值。很明显,在现实世界中,这么一个简单的应用程序无法保证在SQLite3方面的投资。但是,它的简单性确实使它成为一个非常好的学习示例。

如果你对SQL完全陌生,你可能希望查找有关SQLite3和SQL语言的详细信息,然后再继续本章。两个良好的起点是<http://www.sqlite.org/cintro.html>上的“Introduction to the SQLite3 C API”以及<http://www.sqlite.org/lang.html>上的“SQLite SQL Language Guide”。

创建或打开数据库

使用SQLite3之前,必须打开数据库。用于执行此操作的命令`sqlite3_open()`将打开一个现有数据库,如果指定位置上不存在数据库,则它会创建一个新的数据库。下面是打开新数据库的代码:

```
sqlite3 *database;
int result = sqlite3_open("/path/to/database/file", &database);
```

如果`result`等于常量`SQLITE_OK`,则表示数据库已成功打开。此处你应该记住的一件事情就是,数据库文件的路径必须作为C字符串(而非`NSString`)传递。SQLite3是采用可移植的C(而非Objective-C)编写的,它不知道什么是`NSString`。所幸,有一个`NSString`方法,该方法从`NSString`实例生成C字符串:

```
char * cStringPath = [pathString UTF8String];
```

当你对SQLite3数据库执行完所有操作时,通过调用以下内容来关闭数据库:

```
sqlite3_close(database);
```

数据库将其所有数据存储在表中。你可以通过SQL `CREATE`语句创建一个新表,并使用函数

sqlite3_exec将其传递到打开的数据库，如下所示：

```
char * errorMsg;
const char *createSQL = "CREATE TABLE IF NOT EXISTS PEOPLE ➡
    (ID INTEGER PRIMARY KEY AUTOINCREMENT, FIELD_DATA TEXT)";
int result = sqlite3_exec(database, createSQL, NULL, NULL, &errorMsg);
```

执行该操作之前，需要检查SQLITE_OK的result以确保命令成功运行。如果命令未成功运行，errorMsg将包含对所发生问题的描述。

函数sqlite3_exec用于针对SQLite3运行任何不返回数据的命令。它用于执行更新、插入和删除操作。从数据库中检索数据有点复杂。你必须首先通过向其输入SQL SELECT命令来准备该语法：

```
NSString *query = @"SELECT ID, FIELD_DATA FROM FIELDS ORDER BY ROW";
sqlite3_stmt *statement;
int result = (sqlite3_prepare_v2(database, [query UTF8String],
    -1, &statement, nil));
```

说明 所有接受字符串的SQLite3函数都要求使用旧样式的C字符串。在创建示例中，我们创建并传递一个C字符串，但在该示例中，我们创建一个NSString并通过调用其中一个NSString的方法（名为UTF8String）派生一个C字符串。这两个方法都可以接受。如果你需要对字符串进行操纵，则使用NSString或NSMutableString将比较容易，但是将NSString转换为C字符串会导致一些额外开销。

如果result等于SQLITE_OK，则你的语句已成功准备，并且你可以开始单步调试结果集。下面是一个单步调试结果集并从数据库中检索int和NSString的示例：

```
while (sqlite3_step(statement) == SQLITE_ROW) {
    int rowNum = sqlite3_column_int(statement, 0);
    char *rowData = (char *)sqlite3_column_text(statement, 1);
    NSString *fieldValue = [[NSString alloc] initWithUTF8String:rowData];
    // Do something with the data here
    [fieldValue release];
}
sqlite3_finalize(statement);
```

设置项目使用 SQLite3

我们已经讲述了基本知识，下面介绍在实践中的工作原理。我们将再次改进Persistence应用程序，这次使用SQLite3来存储它的数据。它将使用一个表并将字段值存储在该表中4个不同的行中。我们将为每个行提供一个与其字段相对应的行号，例如，field1中的值将存储在表中行号为1的行中。下面让我们开始吧。

通过一个过程API来访问SQLite 3，该API提供对很多C函数调用的接口。要使用此API，我们需要将应用程序链接到一个名为libsqlite3.dylib的动态库。在Mac OS X和iPhone上，该库位于/usr/lib中。

将动态库链接到项目中的过程与在框架中的此链接完全相同。

返回Xcode并打开Persistence应用程序（如果尚未打开）。在Groups & Files窗格中选择Frameworks。接下来，立即从Project菜单中选择Add to Project...。然后，导航到/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator2.1.sdk/usr/lib，并找到名为libsqlite3.dylib的文件。当系统弹出提示，确保取消选中标签为Copy items into destination group's folder (if needed)的复选框。还要确保将Reference Type改为Relative to Current SDK。注意，该目录中可能有多多个以libsqlite3开头的其他条目。务必选择libsqlite3.dylib。它是始终指向最新版本的SQLite3库的别名。

提示 如果选择Absolute Path作为Reference Type，则可以直接链接到/usr/lib/libsqlite3.dylib。该位置很容易记，而绝对路径比较容易被破坏，因此通常不提倡使用。相对路径比较安全，并且在将来版本中不太可能被破坏，即使是libsqlite3.dylib，使用绝对路径链接可能也会比较安全。

接下来，对PersistenceViewController.h进行以下更改：

```
#import <UIKit/UIKit.h>
#import "/usr/include/sqlite3.h"

#define kFilename    @"dataarchive.plist"
#define kDataKey     @"Data"
#define kFilename    @"data.sqlite3"

@interface PersistenceViewController : UIViewController {
    IBOutlet UITextField *field1;
    IBOutlet UITextField *field2;
    IBOutlet UITextField *field3;
    IBOutlet UITextField *field4;

    sqlite3    *database;
}
@property (nonatomic, retain) UITextField *field1;
@property (nonatomic, retain) UITextField *field2;
@property (nonatomic, retain) UITextField *field3;
@property (nonatomic, retain) UITextField *field4;

- (NSString *)dataFilePath;
- (void)applicationWillTerminate:(NSNotification *)notification;
@end
```

再次更改文件名，以便于区分我们将使用的文件与之前版本中使用的文件，并能正确反映其容纳的数据类型。还要声明一个实例变量database，它将指向应用程序的数据库。

切换到PersistenceViewController.m并进行以下更改：

```
#import "PersistenceViewController.h"
#import "FourLines.h"
```

```

@implementation PersistenceViewController
@synthesize field1;
@synthesize field2;
@synthesize field3;
@synthesize field4;

- (NSString *)dataFilePath
{
    NSArray *paths = NSSearchPathForDirectoriesInDomains(
        NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];
    return [documentsDirectory stringByAppendingPathComponent:kFilename];
}

- (void)applicationWillTerminate:(NSNotification *)notification
{
    FourLines *fourLines = [[FourLines alloc] init];
    fourLines.field1 = field1.text;
    fourLines.field2 = field2.text;
    fourLines.field3 = field3.text;
    fourLines.field4 = field4.text;

    NSMutableData *data = [[NSMutableData alloc] init];
    NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc]
        initWithWritingWithMutableData:data];
    [archiver encodeObject:fourLines forKey:kDataKey];
    [archiver finishEncoding];
    [data writeToFile:[self dataFilePath] atomically:YES];
    [fourLines release];
    [archiver release];
    [data release];

    for (int i = 1; i <= 4; i++)
    {
        NSString *fieldName = [[NSString alloc]
            initWithFormat:@"field%d", i];
        UITextField *field = [self valueForKey:fieldName];
        [fieldName release];
        NSString *update = [[NSString alloc] initWithFormat:
            @"INSERT OR REPLACE INTO FIELDS (ROW, FIELD_DATA) →
            VALUES (%d, '%@');", i, field.text];
        char * errorMsg;

        if (sqlite3_exec (database, [update UTF8String], NULL, NULL,
            &errorMsg) != SQLITE_OK) {
            NSLog(@"Error updating tables: %s", errorMsg);
            sqlite3_free(errorMsg);
        }
    }
    sqlite3_close(database);
}

```

```

}
#pragma mark -
- (void)viewDidLoad {

    NSString *filePath = [self dataFilePath];
    if ([[NSFileManager defaultManager] fileExistsAtPath:filePath])
    {
        NSData *data = [[NSMutableData alloc]
            initWithContentsOfFile:[self dataFilePath]];
        NSKeyedUnarchiver *unarchiver =
            [[NSKeyedUnarchiver alloc] initWithReadingWithData:data];
        FourLines *fourLines = [unarchiver decodeObjectForKey:kDataKey];
        [unarchiver finishDecoding];

        field1.text = fourLines.field1;
        field2.text = fourLines.field2;
        field3.text = fourLines.field3;
        field4.text = fourLines.field4;

        [unarchiver release];
        [data release];
    }
    if (sqlite3_open([self dataFilePath] UTF8String), &database)
        != SQLITE_OK) {
        sqlite3_close(database);
        NSAssert(0, @"Failed to open database");
    }

    char *errorMsg;
    NSString *createSQL = @"CREATE TABLE IF NOT EXISTS FIELDS ↗
        (ROW INTEGER PRIMARY KEY, FIELD_DATA TEXT)";
    if (sqlite3_exec(database, [createSQL UTF8String],
        NULL, NULL, &errorMsg) != SQLITE_OK) {
        sqlite3_close(database);
        NSAssert1(0, @"Error creating table: %s", errorMsg);
    }
    NSString *query = @"SELECT ROW, FIELD_DATA FROM FIELDS ORDER BY ROW";
    sqlite3_stmt *statement;
    if (sqlite3_prepare_v2(database, [query UTF8String],
        -1, &statement, nil) == SQLITE_OK) {
        while (sqlite3_step(statement) == SQLITE_ROW) {
            int row = sqlite3_column_int(statement, 0);
            char *rowData = (char *)sqlite3_column_text(statement, 1);

            NSString *fieldName = [[NSString alloc]
                initWithFormat:@"field%d", row];
            NSString *fieldValue = [[NSString alloc]
                initWithUTF8String:rowData];
            UITextField *field = [self valueForKey:fieldName];
            field.text = fieldValue;
        }
    }
}

```

```
        [fieldName release];
        [fieldValue release];
    }
    sqlite3_finalize(statement);
}

UIApplication *app = [UIApplication sharedApplication];
[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(applicationWillTerminate:)
    name:UIApplicationWillTerminateNotification
    object:app];
[super viewDidLoad];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [field1 release];
    [field2 release];
    [field3 release];
    [field4 release];
    [super dealloc];
}
@end
```

下面让我们看一下这些更改吧。

我们首先更改的地方是，在applicationWillTerminate:方法中需要保存应用程序数据的地方。由于数据库中的数据存储在一个表中，因此存储后应用程序的数据将看起来类似于表11-1。

表11-1 存储在数据库的FIELDS表中的数据

行	FIELD表中的数据	行	FIELD表中的数据
1	Four score and seven years ago	3	Continent, a new nation, conceived
2	Our fathers brought forth on this	4	In Liberty, and dedicated to the

保存数据时，需要遍历所有4个字段并且发出了一个分隔命令以更新数据库的每一行。下面就是我们的循环，我们在循环中要做的第一件事情就是创建一个字段名称，以便我们可以检索到正确的文本字段输出口。记住，使用valueForKey:，可以根据其名称检索属性。

```
for (int i = 1; i <= 4; i++)
{
    NSString *fieldName = [[NSString alloc]
        initWithFormat:@"field%d", i];
    UITextField *field = [self valueForKey:fieldName];
```

然后，我们使用文本字段中的值来创建INSERT OR REPLACE SQL语句。该语句将数据插入数据库（如果尚未插入），或者更新其行号匹配的现有行（如果已存在）：

```
NSString *update = [[NSString alloc] initWithFormat:
    @"INSERT OR REPLACE INTO FIELDS (ROW, FIELD_DATA) ➡
    VALUES (%d, '%@')"; i, field.text];
```

接下来，我们针对数据库执行SQL INSERT OR REPLACE：

```
char * errorMsg;
if (sqlite3_exec (database, [update UTF8String], NULL, NULL,
    &errorMsg) != SQLITE_OK) {
    NSLog(@"Error updating tables: %s", errorMsg);
}
}
```

注意，我们在此处使用了一个断言（遇到错误时）。我们之所以会使用断言，而不使用异常或手动错误检查，是因为这种情况只有在我们开发人员出错的情况下才会出现。使用此断言宏将有助于我们调试代码，并且可以脱离最终的应用程序。如果某个错误条件是用户正常情况下可能遇到的条件，则可能应该使用某些其他形式的错误检查。

完成循环之后，关闭数据库并完成对该方法的更改：

```
sqlite3_close(database);
```

此外，唯一的新代码位于viewDidLoad方法中。我们要做的第一件事情就是打开数据库。如果打开数据库时遇到问题，则关闭数据库并发起一个断言：

```
if (sqlite3_open([[self dataFilePath] UTF8String], &database)
    != SQLITE_OK) {
    sqlite3_close(database);
    NSLog(@"Failed to open database");
}
```

接下来，我们必须确保拥有一个容纳数据的表。我们可以使用SQL CREATE TABLE来执行该操作。通过指定IF NOT EXISTS，我们可以防止数据库改写现有数据。如果相同名称的表已经存在，则该命令平静退出，不执行任何操作，因此每次应用程序启动时调用（不必显式检查表是否存在）是比较安全的。

```
char *errorMsg;
NSString *createSQL = @"CREATE TABLE IF NOT EXISTS FIELDS ➡
    (ROW INTEGER PRIMARY KEY, FIELD_DATA TEXT)";
if (sqlite3_exec (database, [createSQL UTF8String], NULL, NULL,
    &errorMsg) != SQLITE_OK) {
    sqlite3_close(database);
    NSLog(@"Error creating table: %s", errorMsg);
}
```


最后，需要加载数据。我们使用SQL SELECT语句来执行该操作。在这个简单的示例中，我们创建了一个SQL SELECT，它请求数据库中的所有行，并要求SQLite3准备SELECT。还告知SQLite按照行号对行排序，以便我们可以按照相同的顺序取回它们。缺少此项，SQLite将按照在内部存储它们的顺序返回行。

```
NSString *query = @"SELECT ROW, FIELD_DATA FROM FIELDS ORDER BY ROW";
sqlite3_stmt *statement;
if (sqlite3_prepare_v2(database, [query UTF8String],
    -1, &statement, nil) == SQLITE_OK) {
```

然后，我们单步调试每个返回的行：

```
while (sqlite3_step(statement) == SQLITE_ROW) {
```

我们获取了行号并将其保存为int类型，然后将字段数据作为C字符串：

```
int row = sqlite3_column_int(statement, 0);
char *rowData = (char *)sqlite3_column_text(statement, 1);
```

接下来，我们根据行号创建一个字段名称（例如，对于行号1为field1），将C字符串转换为NSString，并使用它用从数据库中检索的值设置适当的字段：

```
NSString *fieldName = [[NSString alloc]
    initWithFormat:@"field%d", row];
NSString *fieldValue = [[NSString alloc]
    initWithUTF8String:rowData];
UITextField *field = [self valueForKey:fieldName];
field.text = fieldValue;
```

最后，执行内存清理，至此工作已全部完成：

```
    [fieldName release];
    [fieldValue release];
}
```

为什么你没有编译、运行和试用它呢？输入一些数据，按iPhone仿真器的home按钮。然后重新启动Persistence应用程序，启动后，该数据应该处于原来的位置。就用户所关心的内容而言，这三个不同版本的应用程序之间绝对没有任何差别，但每个版本都使用了截然不同的持久性机制。

11.7 小结

现在，你应该牢固掌握了在会话之间保存应用程序数据的三种不同方法，如果包括最后一章介绍的用户默认值方法，则为四种方法。我们使用属性列表构建了持久保存数据的应用程序，并将该应用程序修改为使用对象归档来保存其数据。然后，我们进行了最后一项更改，使用iPhone内置的SQLite3机制来保存应用程序数据。几乎所有iPhone应用程序都使用这些机制来保存和加载数据的基本构建块。

准备好进行更多操作了吗？拿出你的画笔吧，因为下一章将介绍如何绘图。非常棒的！

到 目前为止，本书中的所有应用程序都是通过UIKit框架中的视图和控件来构造的。借助这些常备组件，我们可以执行许多操作，并且可以构造各式各样的应用程序界面。但是，如果不能高瞻远瞩，某些应用程序会无法完全实现。例如，有时应用程序需要能够进行自定义绘图。幸而，我们可以依靠两个不同的库来满足我们的绘图需要。一个库是Quartz 2D，它是Core Graphics框架的一部分；另一个库是OpenGL ES，它是跨平台的图形库。OpenGL ES是跨平台图形库OpenGL的简化版。OpenGL ES是OpenGL的一个子集，OpenGL ES是专门为iPhone之类的嵌入式系统（因此缩写为字母“ES”）设计的。本章将介绍这两个功能强大的图形环境。我们将在这两种环境中构建示例应用程序，并尝试了解什么时候使用哪个环境。

12.1 图形世界的两个视图

尽管Quartz和OpenGL有许多共性，但它们之间存在明显差别。Quartz是一组函数、数据类型以及对象，专门设计用于直接在内存中对视图或图像进行绘制。

Quartz将正在绘制的视图或图像视为一个虚拟的画布，并遵循所谓的**绘画者模型**。这只是一种奇特的方式，之所以这么说，是因为应用绘图命令的方式很大程度上与将颜料应用于画布的方式相同。如果绘画者将整个画布涂为红色，然后将画布的下半部分涂为蓝色，那么画布将变为一半红色、一半蓝色或紫色。如果颜料是不透明的，应该为蓝色，如果颜料是半透明的，应该为紫色。

Quartz的虚拟画布采用相同的工作方式。如果将整个视图涂为红色，然后将视图下半部分涂为蓝色，你将拥有一个一半红色、一半蓝色或紫色的视图，这取决于第二个绘图操作是完全不透明的还是部分透明的。每个绘图操作都将被应用于画布，并且处于之前所有绘图操作之上。

另一方面，OpenGL ES以**状态机**的形式实现。这个概念可能有点不好理解，因为不能将其归结为一个简单的比喻，如在虚拟画布上绘画。OpenGL ES不允许执行直接影响视图、窗口或图像的操作，它维护一个虚拟的三维世界。当向这个世界中添加对象时，OpenGL会跟踪所有对象的状态。虽然OpenGL ES没有提供虚拟画布，但是却提供了一个进入其世界的虚拟窗口。可以向该世界添加对象并定义虚拟窗口相对于该世界的位置。然后，OpenGL根据配置方式以及各种对象彼此相对的位置绘制视图，并通过该窗口呈现给用户。这个概念有点儿抽象，因此如果你感到困惑，也不必担心。本章稍后将通过示例详细说明这个概念。

Quartz相对比较容易使用。它提供了各种直线、形状以及图像绘制函数。尽管易于使用，但Quartz 2D仅限于二维绘图。尽管许多Quartz函数会在绘图时利用硬件加速，但无法保证在Quartz中执行的任何操作都得到了加速。

尽管OpenGL非常复杂，并且概念上也比较难理解，但是它的强大性是毫无疑问的。它同时提供了二维和三维绘图工具。它经过专门设计，目的是为了充分利用硬件加速。由于它可以跟踪虚拟世界的状态，因此还非常适合用于编写游戏和其他复杂的、图形密集的程序。

12.2 本章的绘图应用程序

下一个应用程序是一个简单的绘图程序（参见图12-1）。我们将分别使用Quartz 2D和OpenGL ES来构建该应用程序，因此你会真正感受到它们之间的差别。

该应用程序的特点是顶部和底部各有一个工具栏，每个工具栏都有一个分段控件。顶部的控件用于更改图形颜色，底部的控件用于更改要绘制的形状。当用户触击和拖动对象时，程序将用所选颜色绘制所选形状。为了最大程度地降低应用程序的复杂性，一次只绘制一种形状。

12.3 Quartz 绘图方法

使用Quartz绘制图形时，通常会向绘制图形的视图中添加绘图代码。例如，可能会创建UIView的子类，并向该类的drawRect:方法中添加Quartz函数调用。drawRect:方法是UIView类定义的一部分，并且每次需要重绘视图时都会调用该方法。如果在drawRect:中插入Quartz代码，则会先调用该代码，然后重绘视图。

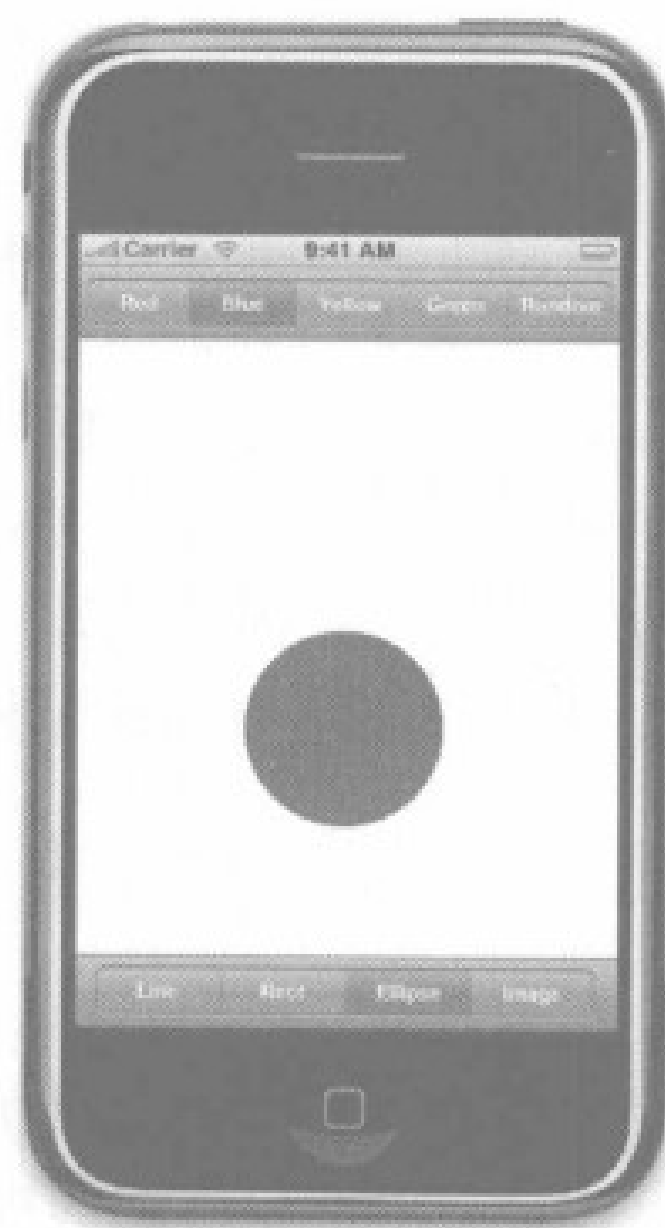


图12-1 本章中的简单绘图应用程序

12.3.1 Quartz 2D 的图形上下文

在Quartz 2D中，和在其他Core Graphics中一样，绘图是在图形上下文中进行的，通常，只称为上下文。每个视图都有相关联的上下文。要在某个视图中绘图时，你将检索当前上下文，使用此上下文进行各种Quartz图形调用，并且让此上下文负责将图形呈现到视图上。

这面这行代码将检索当前上下文：

```
CGContextRef context = UIGraphicsGetCurrentContext();
```

说明 我们使用Core Graphics C函数，而不是使用Objective-C对象来绘图。Core Graphics的API是基于C的，因此在本章的此部分中编写的大多数代码将由C函数调用组成。

定义图形上下文之后，可以将该上下文传递给各种Core Graphics函数来进行绘图。例如，以下代码将在上下文中绘制一条2像素宽的直线：

```
CGContextSetLineWidth(context, 2.0);
CGContextSetStrokeColorWithColor(context, [UIColor redColor].CGColor);
CGContextMoveToPoint(context, 100.0f, 100.0f);
CGContextAddLineToPoint(context, 200.0f, 200.0f);
CGContextStrokePath(context);
```

第一个调用指定任何绘制都应该创建一条2像素宽的直线。然后，我们指定笔划颜色应该为红色。在Core Graphics中，有两种颜色与绘图操作关联：笔划颜色和填充颜色。笔划颜色用于绘制直线以及形状的轮廓，填充颜色用于填充形状。

上下文具有一种与绘制直线关联的不可见的“画笔”。调用CGContextMoveToPoint()时，会将这个不可见的画笔移动到新位置，而不是实际绘制任何内容。这表示我们指定要绘制的直线从位置(100, 100)处开始（参见12.3.2节有关位置的说明）。下一个函数实际上绘制了一条从当前画笔位置到指定位置（该位置将成为新的画笔位置）的直线。在Core Graphics中绘图时，我们没有绘制任何实际可见内容。我们创建了形状、直线或某些其他对象，但它们不包含颜色或者任何使其可见的内容。就像用不可见的墨水在书写一样。在执行某些操作使其可见之前，我们看不到直线。因此，下一步是告知Quartz使用CGContextStrokePath()绘制直线。该函数将使用之前我们设置的线宽和笔划颜色对此直线进行涂色并使其可见。

12.3.2 坐标系

在上面的代码块中，我们将一对浮点数作为参数传递给CGContextMoveToPoint()和CGContextLineToPoint()。这些浮点数表示在Core Graphics坐标系中的位置。此坐标系中的位置由其x和y坐标表示，我们通常用(x, y)来表示。上下文左上角为(0, 0)。向下移动时，y增加。向右移动时，x增加。

在最后一个代码片段中，我们绘制了一条从(100, 100)到(200, 200)的对角线，绘制的直线类似于图12-2所示的直线。

在iPhone绘图时需经常使用的一个概念就是坐标系，它借鉴了许多图形库的绘图机制以及传统的几何学。例如，在OpenGL ES中，(0, 0)位于左下角，当y坐标增加时，你将移向上下文或视图的顶部，如图12-3所示。使用OpenGL时，必须将位置从视图坐标系转换为OpenGL坐标系。这非常容易，在本章稍后的部分中，你将了解如何使用OpenGL。

若要在坐标系中指定一个点，某些Quartz函数需要使用两个浮点数作为参数。其他Quartz函数要求该点嵌入在CGPoint中，CGPoint是一个包含两个浮点值（即x和y）的struct。若要描述视图或其他对象的大小，Quartz将使用CGSize。CGSize也是一个拥有两个浮点值（即width和height）的struct。Quartz还声明一个名为CGRect的数据类型，它用于在坐标系中定义矩形。CGRect包含两个元素，一个是名为origin的CGPoint，它确定矩形的左上角，另一个是名为size的CGSize，它确定矩形的宽度（width）和高度（height）。

12.3.3 指定颜色

颜色是绘图的一个重要因素，因此理解颜色在iPhone上的运行原理是非常重要的。UIKit为此

提供了一个Objective-C类: `UIColor`。你不能在Core Graphic调用中直接使用`UIColor`对象,但可以像我们之前在以下代码片段中所做的一样,使用它的`CGColor`属性从`UIColor`实例中检索`CGColor`引用(Core Graphic函数要求这样做):

```
CGContextSetStrokeColorWithColor(context, [UIColor redColor].CGColor);
```

我们使用`redColor`便利方法创建了一个`UIColor`实例,然后检索它的`CGColor`属性,并将该属性传递给函数。

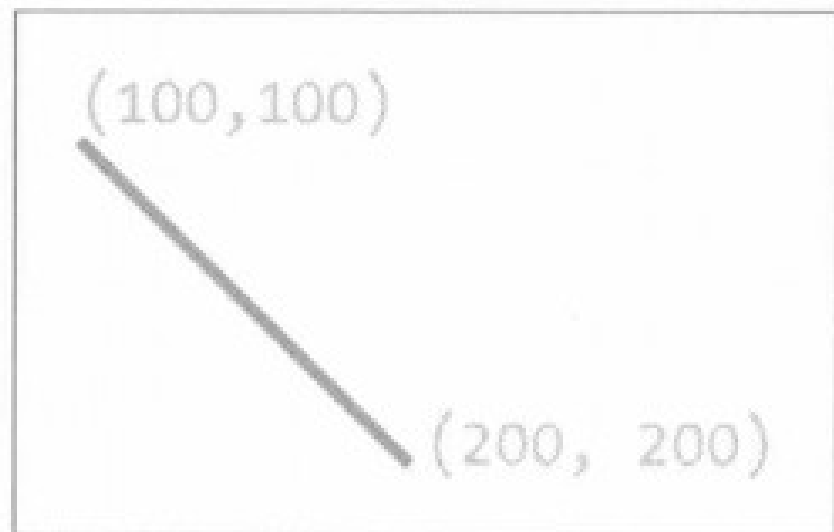


图12-2 在视图坐标系中绘制一条直线

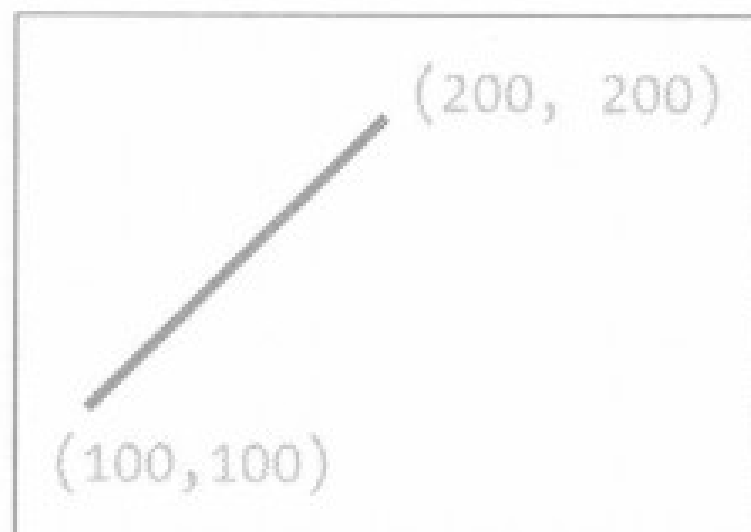


图12-3 在许多图形库(包括OpenGL)中,从(100, 100)到(200, 200)绘制一条直线应该与此类似,而不是与图12-2中的直线类似

1. iPhone显示的颜色理论

在现代计算机图形中,通常用4个要素(即红色、绿色、蓝色和透明度)表示颜色。在Quartz 2D中,这些值都是`CGFloat`类型(与iPhone上的`float`相同)类型,并且只能在0和1中取值。前3个要素很容易理解,因为它们表示加法三原色或RGB颜色模型(参见图12-4)。以不同比例组合这3种颜色可以产生不同的颜色。如果以相同的比例将这3种级别的原色放到一起,出现在你眼前的结果将是白色或某种灰度,具体情况取决于所混合原色的饱和度。以不同比例组合这3种原色,你可以获得一系列不同的颜色,称为色域。

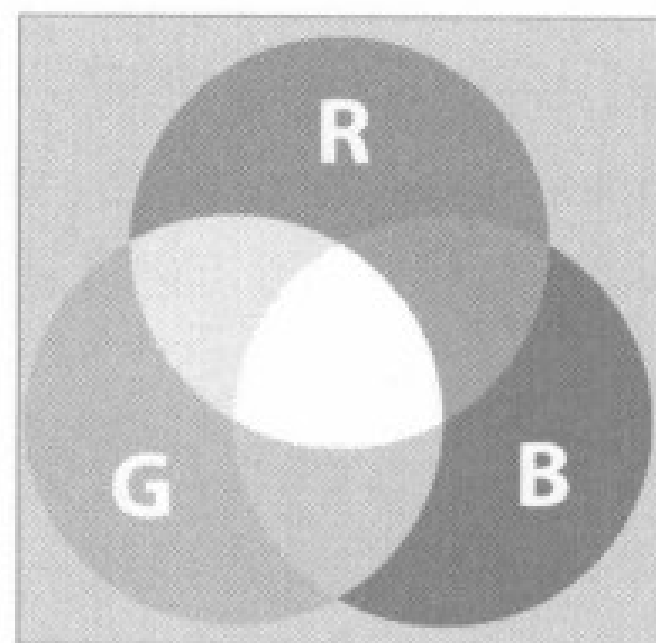


图12-4 组成RGB颜色模型的加法三原色的简单表示

你可能在小学学习过,原色包括红色、黄色和蓝色。这些原色(称为历史减法三原色或RYB颜色模型)在现代颜色理论中很少应用,几乎从来没有在计算机图形中使用。RYB颜色模型的色域是非常有限的,并且该模型不容易进行数学定义。你可能从来没有怀疑过小学的美术教师,但他们的理解至少不适用于计算机图形环境。之后我们会重复地讲“原色包括红色、绿色和蓝色”。

2. 比所看到的颜色还多

除了红色、绿色和蓝色之外,Quartz 2D(以及OpenGL ES)还有另一个组件:即`alpha`,它表示颜色的透明程度。当在一种颜色的上面绘制另一种颜色时,`Alpha`用于确定绘制的最终颜色。如果`alpha`为1.0,则绘制的颜色为100%不透明,它的下面的任何颜色都无法看清楚。如果它的值为任何小于1.0的值,则它下面的颜色将能够透过它显示出来,最后获得混合的颜色。当使用`alpha`组件时,有时颜色模型称为RGBA颜色模型,但是从技术上来讲,`alpha`实际上并不是颜色的一部分;它只是定义绘制时颜色与其他颜色的交互方式。

尽管在计算机图形中最常用的是RGB模型，但是它不是唯一的颜色模型。其他一些模型也得到了使用，包括色调、饱和度、值（HSV）；色调、饱和度、亮度（HSL）；蓝绿色、洋红色、黄色、黑色（CMYK），它们用于实现四色打印；灰度级。此外，不同版本的RGB颜色空间使这一切变得更加复杂。所幸，对于大多数操作来说，我们不必担心所使用的颜色模型。我们只需从UIColor对象中传递CGColor，Core Graphics即可处理任何所需的转换。在使用OpenGL ES时，记住由于OpenGL ES需要采用RGBA来指定颜色，因此Quartz支持其他颜色模型，这一点非常重要。

UIColor提供了许多便利方法，可以返回初始化为特定颜色的UIColor对象。在上一个代码示例中，我们使用redColor方法来获取初始化为红色的颜色。幸运的是，这些便利方法创建的UIColor实例都使用RGBA颜色模型。

如果你需要对颜色进行更多控制，但不使用便利方法，则可以通过指定所有这4个组件来创建一种颜色。下面是一个示例：

```
return [UIColor colorWithRed:1.0f green:0.0f blue:0.0f alpha:1.0f];
```

12.3.4 在上下文中绘制图像

使用Quartz 2D，可以在上下文中直接绘制图像。这是Objective-C类（UIImage）的另一个示例，你可以使用此类作为操作Core Graphics数据结构（CGImage）的备用选项。此UIImage类包含将图像绘制到当前上下文中的方法。你需要确定此图像出现在上下文中的位置，方法是：指定一个CGPoint来确定图像的左上角或者指定一个CGRect来框住图像，并根据需要调整图像大小使其合适该框。可以在当前上下文中绘制一个UIImage，如下所示：

```
CGPoint drawPoint = CGPointMake(100.0f, 100.0f);
[image drawAtPoint:drawPoint];
```

12.3.5 绘制形状：多边形、直线和曲线

Quartz 2D提供了许多函数，这些函数简化了复杂形状创建。若要绘制一个矩形或一个多边形，实际上你不必计算角度，绘制直线或者根本不必进行任何数学计算。你只需调用一个Quartz函数即可实现该操作。例如，绘制椭圆形的方法是，定义它所适合的矩形并且让Core Graphics执行以下任务：

```
CGRect theRect = CGRectMake(0,0,100,100);
CGContextAddEllipseInRect(context, theRect);
CGContextDrawPath(context, kCGPathFillStroke);
```

对于矩形也是类似的方法。此外，还有许多方法用于创建更为复杂的形状（如弧形和Bezier路径）。若要了解有关Quartz中弧形和Bezier路径的详细信息，请查看http://developer.apple.com/documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/dq_intro/chapter_1_section_1.html上iPhone Dev Center中或Xcode联机文档中的*Quartz 2D Programming Guide*（Quartz 2D编程指南）。

12.3.6 Quartz 2D 工具示例：模式、梯度、虚线模式

Quartz 2D不像OpenGL那么昂贵，却提供了许多吸引人的工具，尽管这些工具中的许多工具

不在本书的讨论范围之内，但你应该知道它们的存在。例如，Quartz 2D支持用梯度填充多边形，而不只是用纯色，并且不仅仅支持实线，而且还支持虚线模式。浏览图12-5中截取自苹果公司 QuartzDemo示例代码的屏幕截图，了解Quartz 2D的实际操作示例。

现在你已经基本了解了Quartz 2D的工作原理以及它的功能，让我们尝试使用它吧。

12.4 构建 QuartzFun 应用程序

在Xcode中，使用基于视图的应用程序模板创建一个新项目，并将其命名为QuartzFun。创建项目之后，展开Classes和Resources文件夹，单击Classes文件夹，以便我们可以添加类。这个模板已经为我们提供了一个应用程序委托和一个视图控制器。我们将在视图中执行自定义绘图，因此需要创建一个UIView子类。在该子类中，我们将通过覆盖drawRect:方法进行绘图。创建一个新的Cocoa Touch Classes文件，并选择UIView subclass模板。将该文件命名为QuartzFunView.m，并确保创建了头文件。

与之前一样，我们将定义一些常量，但这次定义的常量是多个类所需要的，并且不是特定于某个类的。我们将只为常量创建头文件，因此通过以下访问创建一个新文件：从Other栏中选择Empty File模板，并将其命名为Constants.h。

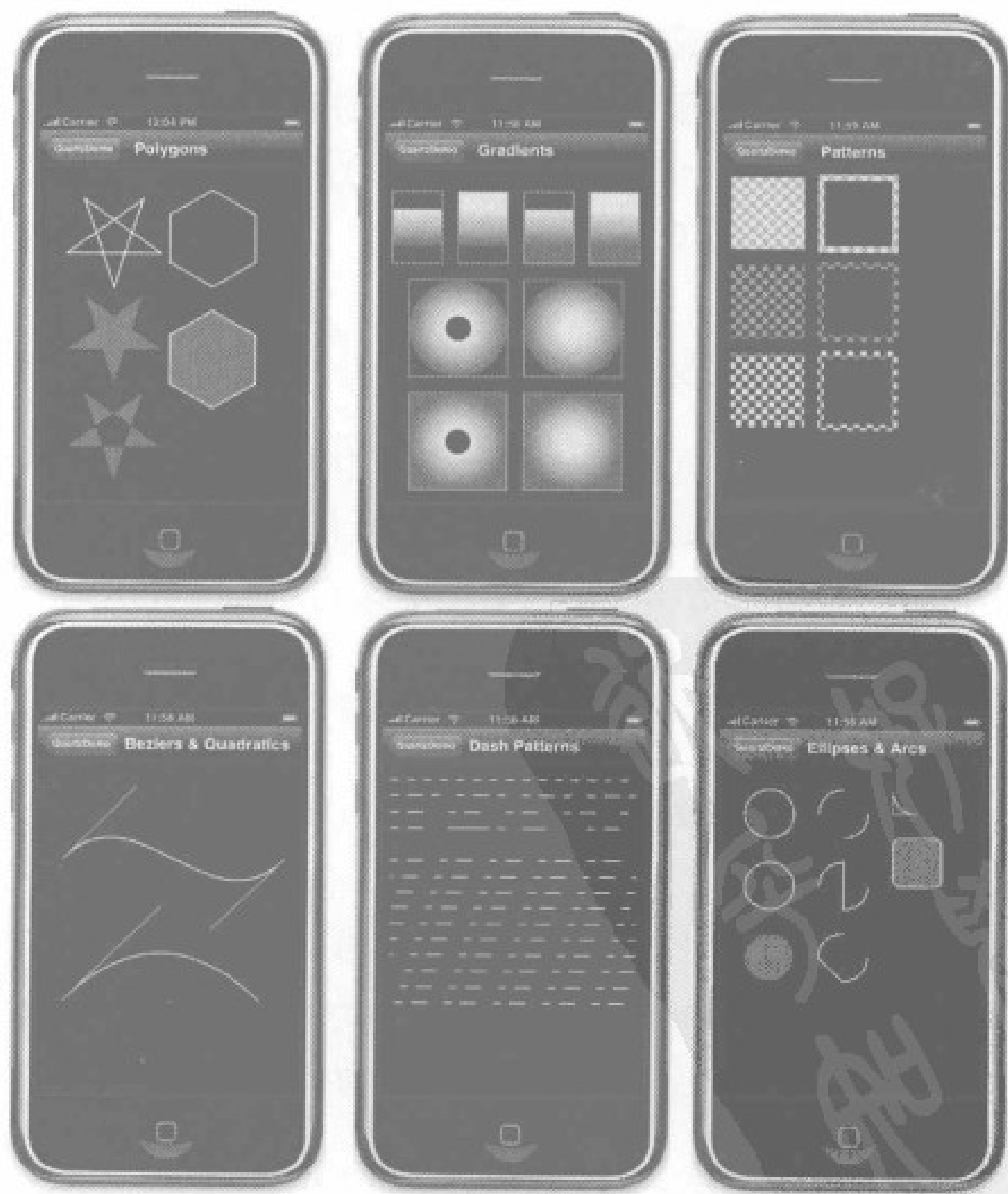


图12-5 一些Quartz 2D示例，来自于苹果公司提供的Quartz Demo示例项目

我们还需要创建两个文件。查看图12-1，你可以看到我们提供了一个选择随机颜色的选项。但UIColor没有提供返回随机颜色的方法，因此我们必须编写代码来执行该操作。当然，我们将该代码放置到控制器类中，但是由于我们是了解Objective-C的程序员，因此将该代码放置到UIColor上的某个类别中。使用Empty File模板创建两个文件，将它们分别命名为UIColor-Random.h和UIColor-Random.m。或者，使用NSObject subclass模板创建UIColor-Random.m，并让该模板为你自动创建UIColor-Random.h；然后删除这两个文件的内容。

12.4.1 创建随机颜色

让我们首先处理此类别。在UIColor-Random.h中，添加以下代码：

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

@interface UIColor(Random)
+ (UIColor *)randomColor;
@end
```

现在，切换到UIColor-Random.m并添加以下内容：

```
#import "UIColor-Random.h"

@implementation UIColor(Random)
+ (UIColor *)randomColor
{
    static BOOL seeded = NO;
    if (!seeded) {
        seeded = YES;
        srand(time(NULL));
    }
    CGFloat red = (CGFloat)random()/(CGFloat)RAND_MAX;
    CGFloat blue = (CGFloat)random()/(CGFloat)RAND_MAX;
    CGFloat green = (CGFloat)random()/(CGFloat)RAND_MAX;
    return [UIColor colorWithRed:red green:green blue:blue alpha:1.0f];
}
@end
```

这非常简单。我们声明一个静态变量，该变量告诉我们该方法是否是第一次调用。应用程序运行期间第一次调用该方法时，我们将运行随机数字生成器。在此处执行此操作意味着：我们不必依赖应用程序在其他地方执行该操作，因此，我们可以在其他iPhone项目中重用此类别。

在运行随机数字生成器之后，生成三个随机的CGFloat，其值介于0.0和1.0之间，使用这3个值来创建新的颜色。我们将alpha设置为1.0，以便所有生成的颜色都是不透明的。

12.4.2 定义应用程序常量

我们将使用分段控制器为用户可以选择的每个选项定义常量。单击Constants.h并添加以下内容：

```
typedef enum {
    kLineShape = 0,
    kRectShape,
    kEllipseShape,
    kImageShape
} ShapeType;

typedef enum {
    kRedColorTab = 0,
    kBlueColorTab,
    kYellowColorTab,
    kGreenColorTab,
    kRandomColorTab
} ColorTabIndex;

#define degreesToRadian(x) (3.14159265358979323846 * x / 180.0)
```

为了使代码更具有可读性，我们使用typedef声明了两个枚举类型。

12.4.3 实现 QuartzFunView 框架

由于我们将在UIView的某个子类中进行绘图，因此让我们用其所需的所有内容设置该类，但进行绘图的代码除外，我们稍后将添加这些代码。单击QuartzFunView.h并进行以下更改：

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

@interface QuartzFunView : UIView {
    CGPoint      firstTouch;
    CGPoint      lastTouch;
    UIColor      *currentColor;
    ShapeType     shapeType;
    UIImage      *drawImage;
    BOOL         useRandomColor;
}
@property CGPoint firstTouch;
@property CGPoint lastTouch;
@property (nonatomic, retain) UIColor *currentColor;
@property ShapeType shapeType;
@property (nonatomic, retain) UIImage *drawImage;
@property BOOL useRandomColor;
@end
```

我们做的第一件事情就是导入刚才创建的Constants.h头文件，这样便可以使用枚举。然后，声明实例变量。前两个变量将跟踪用户拖过屏幕的手指。我们将用户第一次触摸屏幕的位置存储在firstTouch中，将拖动时手指的位置以及拖动结束时手指的位置存储在lastTouch中。我们的绘图代码将使用这两个变量来确定在哪里绘制请求的形状。

接下来，我们定义某种颜色来存放用户的颜色选择，并定义一个ShapeType以跟踪用户想绘制的形状。然后，定义一个UIImage属性，该属性存放用户选择底部工具栏中最右侧项目时在屏幕上绘制的图像（参见图12-6）。我们定义的最后一个属性是Boolean，它用于跟踪用户是否请

求随机颜色。

切换到QuartzFunView.m并进行以下更改：

```
#import "QuartzFunView.h"
#import "UIColor-Random.h"
@implementation QuartzFunView
@synthesize firstTouch;
@synthesize lastTouch;
@synthesize currentColor;
@synthesize shapeType;
@synthesize drawImage;
@synthesize useRandomColor;

- (id)initWithCoder:(NSCoder*)coder
{
    if ( ( self = [super initWithCoder:coder] ) ) {
        self.currentColor = [UIColor redColor];
        self.useRandomColor = NO;
        if (drawImage == nil)
            self.drawImage = [UIImage imageNamed:@"iphone.png"];
    }
    return self;
}

- (void)drawRect:(CGRect)rect {

    // Drawing code

}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    if (useRandomColor)
        self.currentColor = [UIColor randomColor];
    UITouch *touch = [touches anyObject];
    firstTouch = [touch locationInView:self];
    lastTouch = [touch locationInView:self];
    [self setNeedsDisplay];
}

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

    [self setNeedsDisplay];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    lastTouch = [touch locationInView:self];

    [self setNeedsDisplay];
}
}
```

```

- (void)dealloc {
    [currentColor release];
    [drawImage release];
    [super dealloc];
}

```

```
@end
```

由于该视图是从nib中加载的，因此我们首先实现 `initWithCoder:`。请记住nib中的对象实例将存储为归档对象，这与我们在上一章将对象归档和加载到磁盘所使用的机制完全相同。因此，从nib中加载对象实例时，`initWithFrame:`都不会调用。而是使用 `initWithCoder:`，因此任何初始化代码都需要在这里添加。若要将初始颜色值设置为红色，则将 `useRandomColor` 初始化为NO，并加载我们将要绘制的图像文件。你不必完全理解此处的其他代码。我们将在第13章详细讨论使用 `touchesBegan:withEvent:`、`touchesMoved:withEvent:` 和 `touchesEnded:withEvent:` 方法的技巧和具体细节。简单地说，可以覆盖这3个 `UIView` 方法以确定用户触摸iPhone屏幕的位置。

当用户手指第一次触摸屏幕时会调用 `touchesBegan:withEvent:`。在该方法中，如果用户已经使用之前添加到 `UIColor` 中的新 `randomColor` 方法选择了某个随机颜色，则我们需要更改此颜色。之后，我们存储当前位置，这样便可以知道用户第一次触摸屏幕的位置，并指出：需要通过在 `self` 上调用 `setNeedsDisplay` 来重新绘制视图。

当用户在屏幕上拖动手指时会连续调用接下来的 `touchesMoved:withEvent:` 方法。此处，我们所要做的就是将新位置存储在 `lastTouch` 中，并指出需要重新绘制该屏幕。

当用户将手指从屏幕上抬起时会调用最后一个方法，即 `touchesEnded:withEvent:`。就像在 `touchesMoved:withEvent:` 方法中一样，我们所要做的就是将最后一个位置存储在 `lastTouch` 变量中，并指出需要重新绘制该视图。

如果你还没有全面了解这3个方法在触摸过程中所执行的操作，请不用担心，我们将在后续章节中更详细地介绍这些方法。

完成应用程序骨架并运行之后，我们将重新回顾这些方法。`drawRect:` 方法就是此应用程序的主体部分，目前仅包含一条注释，因为我们尚未编写该方法。我们首先需要完成应用程序设置，然后再添加绘图代码。

12.4.4 向视图控制器中添加输出口和操作

如果参考图12-1，你会看到我们的界面包括两个分段控制器，一个位于屏幕顶部，另一个位于屏幕底部。使用顶部的控制器，用户可以选择颜色，但该分段控制器仅应用于底部4个选项中的3个选项，因此我们需要一个到顶部分段控制器的输出口，以便当它不起作用时将它隐藏。我



图12-6 当在屏幕上绘制UIImage时，颜色控件消失

们还需要两个方法，一个方法是在选择新颜色时调用，另一个方法是在选择新形状时调用。

单击QuartzFunViewController.h并进行以下更改：

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

@interface QuartzFunViewController : UIViewController {
    IBOutlet    UISegmentedControl *colorControl;
}
@property (nonatomic, retain) UISegmentedControl *colorControl;
- (IBAction)changeColor:(id)sender;
- (IBAction)changeShape:(id)sender;
@end
```

所做的更改一目了然。接下来，切换到QuartzFunViewController.m并进行以下更改：

```
#import "QuartzFunViewController.h"
#import "QuartzFunView.h"
#import "UIColor-Random.h"
#import "Constants.h"

@implementation QuartzFunViewController
@synthesize colorControl;

- (IBAction)changeColor:(id)sender {
    UISegmentedControl *control = sender;
    NSInteger index = [control selectedIndex];

    QuartzFunView *quartzView = (QuartzFunView *)self.view;

    switch (index) {
        case kRedColorTab:
            quartzView.currentColor = [UIColor redColor];
            quartzView.useRandomColor = NO;
            break;
        case kBlueColorTab:
            quartzView.currentColor = [UIColor blueColor];
            quartzView.useRandomColor = NO;
            break;
        case kYellowColorTab:
            quartzView.currentColor = [UIColor yellowColor];
            quartzView.useRandomColor = NO;
            break;
        case kGreenColorTab:
            quartzView.currentColor = [UIColor greenColor];
            quartzView.useRandomColor = NO;
            break;
        case kRandomColorTab:
            quartzView.useRandomColor = YES;
            break;
        default:
    }
```

```

        break;
    }
}
- (IBAction)changeShape:(id)sender {
    UISegmentedControl *control = sender;
    [(QuartzFunView *)self.view setShapeType:[control
        selectedSegmentIndex]];

    if ([control selectedSegmentIndex] == kImageShape)
        colorControl.hidden = YES;
    else
        colorControl.hidden = NO;
}

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

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [colorControl release];
    [super dealloc];
}

@end

```

这段代码也非常简单。在 `changeColor:` 方法中，我们确定已选择的分段，并根据选择内容创建一个新颜色。我们将 `view` 转换为 `QuartzFunView`。接下来，设置它的 `currentColor` 属性，以便它知道绘图所使用的颜色。但选择随机颜色时除外，如果选择随机颜色，我们只需将视图的 `useRandomColor` 属性设置为 `YES`。由于所有绘图代码将包含在视图内部，因此我们不必对其他方法执行任何操作。

在 `changeShape:` 方法中，我们的做法类似。但是，由于不必创建对象，因此我们只需将视图的 `shapeType` 属性设置为来自 `sender` 的分段索引。还记得 `ShapeType` `enum` 吗？`enum` 的4个元素与应用程序视图底部的4个工具栏分段相对应。我们将形状设置为与当前所选择的分段相同，并根据是否选择了 `Image` 分段来隐藏 `colorControl` 和取消隐藏 `colorControl`。

12.4.5 更新 QuartzFunViewController.xib

开始绘图之前,我们需要向nib中添加分段控件,然后连接操作和输出口。双击QuartzFunViewController.xib,在Interface Builder中将其打开。第一件事是更改视图的类,因此在标签为QuartzFunViewController.xib的窗口中单击View图标,并按⌘4打开身份检查器。将该类从UIView改为QuartzFunView。

接下来,在库中找到Navigation Bar。确保你控制的是Navigation Bar,而非Navigation Controller。我们只是希望该工具栏位于视图顶部。将Navigation Bar紧贴在视图窗口的顶部。

接下来,在库中找到Segmented Control,并将该控件拖动到Navigation Bar的顶部。放下该控件之后,它应该仍然为选中状态。捕捉此分段控件任何一侧的调整大小的点,调整它的大小,以便它占据导航栏的整个宽度。你不会看到任何蓝色引导线,但这种情况下,Interface Builder会限制该栏的最大大小,因此只需拖动它直到不再进一步展开为止。

按⌘1调出属性检查器,并将分段数量从2更改为5。依次双击各分段,将它们的标签分别改为(从左到右)Red、Blue、Yellow、Green和Random。此时,你的View窗口应该类似于图12-7。

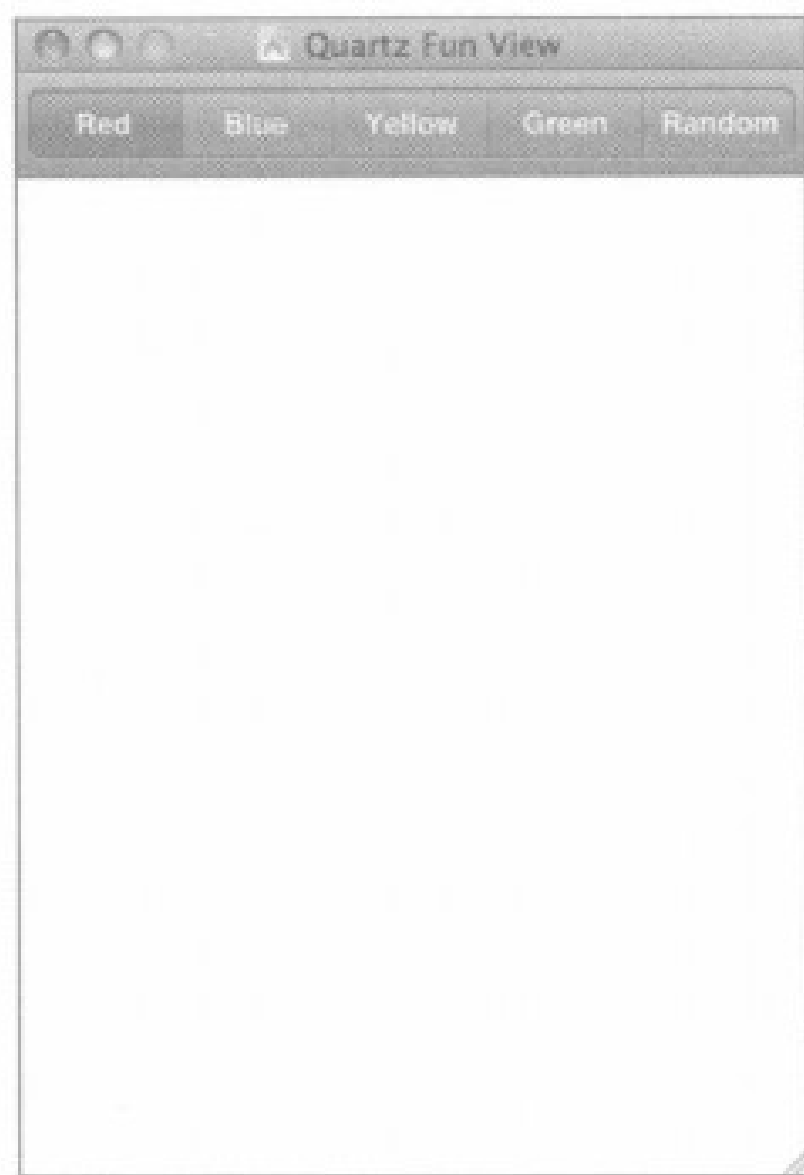


图12-7 完成后的导航栏

按住Control的同时,将File's Owner图标拖到分段控件上,并选择colorControl输出口。接下来,确保选中分段控件,并按⌘2调出连接检查器。从Value Changed事件拖到File's Owner并选择changeColor:操作。

现在,在库中找到Toolbar,从中拖出一个工具栏并将其放置在窗口底部。库中的Toolbar上有一个我们并不需要的按钮,因此选择该按钮并按下Delete键。放置控件并删除按钮之后,选中另一个Segmented Control,并将其拖到工具栏上。

结果是,分段控件在工具栏中居中有点困难,因此我们将提供一点帮助。将Flexible Space Bar Button Item从库中拖到位于分段控件左侧的工具栏上。接下来,将另一个Flexible Space Bar Button Item拖到位于分段控件右侧的工具栏上。当我们调整该工具栏的大小时,这些项目将使分段控件位于工具栏的中心。单击分段控件以将其选中,并调整其大小以使它适合此工具栏,其中左右两侧各留有一点空间。

接下来,按⌘1打开属性检查器,并将分段数从2更改为4。按顺序将4个分段的标题改为Line、Rect、Ellipse和Image。切换到连接检查器,并将Value Changed事件连接到File's Owner的changeShape:操作方法。保存并关闭nib,然后返回Xcode。

说明 你可能想知道为什么我们将一个导航栏放置在视图的顶部,将一个工具栏放置在视图的底部。根据苹果公司发布的“iPhone人机界面指南”,导航栏是经过专门设计的,它在屏

幕顶部看起来比较美观，而工具栏则是专门为底部而设计。如果你在Interface Builder的库窗口中阅读了Toolbar和Navigation Bar的描述，你将会看到对此设计意图的说明。

编译并运行应用程序，确保一切正常。目前你还不能在屏幕上绘制形状，但分段控件可以工作，当在底部控件中轻击Image分段时，颜色控件会消失。一切都开始工作之后，我们进行绘图。

12.4.6 绘制直线

返回Xcode，编辑QuartzFunView.m，并用以下代码替换空的drawRect:方法：

```
- (void)drawRect:(CGRect)rect {

    CGContextRef context = UIGraphicsGetCurrentContext();

    CGContextSetLineWidth(context, 2.0);
    CGContextSetStrokeColorWithColor(context, currentColor.CGColor);
    CGContextSetFillColorWithColor(context, currentColor.CGColor);

    switch (shapeType) {
        case kLineShape:
            CGContextMoveToPoint(context, firstTouch.x, firstTouch.y);
            CGContextAddLineToPoint(context, lastTouch.x, lastTouch.y);
            CGContextStrokePath(context);
            break;
        case kRectShape:
            break;
        case kEllipseShape:
            break;
        case kImageShape:
            break;
        default:
            break;
    }
}
```

首先，检索对当前上下文的引用，以便知道要绘图的位置：

```
CGContextRef context = UIGraphicsGetCurrentContext();
```

接下来，将线宽设置为2.0，这意味着我们画的任何直线都是2个像素宽：

```
CGContextSetLineWidth(context, 2.0);
```

随后，设置所画直线的颜色。由于UIColor有该方法所需的CGColor属性，因此我们使用currentColor实例变量的这个属性将正确的颜色传递给该函数：

```
CGContextSetStrokeColorWithColor(context, currentColor.CGColor);
```

我们使用switch跳转到每个形状类型的相应代码。我们将从处理kLineShape的代码开始，使其正常工作，然后依次为每个形状添加代码：

```
switch (shapeType) {
    case kLineShape:
```

要绘制直线，我们将不可见画笔移动到用户触摸的第一个位置。请记住，我们将该值存储在 `touchesBegan:` 方法中，以便它总是反映用户上次触摸或拖动时触摸的第一个点。

```
CGContextMoveToPoint(context, firstTouch.x, firstTouch.y);
```

接下来，我们绘制一条从该点到用户触摸的最后一个点的直线。如果用户的手指仍然与屏幕接摸，则 `lastTouch` 包含用户手指的当前位置。如果用户的手离开了屏幕，则 `lastTouch` 包含用户手指离开屏幕时的位置。

```
CGContextAddLineToPoint(context, lastTouch.x, lastTouch.y);
```

然后，画出这条路径。以下函数将画出一条直线，这是我们使用之前设置的颜色和宽度绘制成的：

```
CGContextStrokePath(context);
```

随后，我们只需完成此 `switch` 语句就可以了，如下所示：

```
        break;
    case kRectShape:
        break;
    case kEllipseShape:
        break;
    case kImageShape:
        break;
    default:
        break;
}
```

在构建和运行之前，还有最后一个步骤。由于 Quartz 2D 是 Core Graphics 的一部分，因此我们需要将 Core Graphics 框架添加到项目中。在项目窗口中，单击 Groups & Files 窗格中的 Frameworks，并添加此框架。如果你忘记了具体细节，请参考第 5 章。你将此书页折起角了，对吗？

此时，你应该能够进行编译和运行了。Rect、Ellipse 和 Shape 选项将不可用，但你应该能够很好地绘制直线（参见图 12-8）。

12.4.7 绘制矩形和椭圆形

让我们实现同时绘制矩形和椭圆形的代码，因为 Quartz 2D 基本上采用相同的方法实现这两个对象。对 `drawRect:` 方法进行以下更改：



图 12-8 应用程序中绘制直线的部分现在已经完成。在该图像中，我们使用随机颜色进行绘制

```

- (void)drawRect:(CGRect)rect {

    if (currentColor == nil)
        self.currentColor = [UIColor redColor];

    CGContextRef context = UIGraphicsGetCurrentContext();

    CGContextSetLineWidth(context, 2.0);
    CGContextSetStrokeColorWithColor(context, currentColor.CGColor);

    CGContextSetFillColorWithColor(context, currentColor.CGColor);
    CGRect currentRect = CGRectMake (
        (firstTouch.x > lastTouch.x) ? lastTouch.x : firstTouch.x,
        (firstTouch.y > lastTouch.y) ? lastTouch.y : firstTouch.y,
        fabsf(firstTouch.x - lastTouch.x),
        fabsf(firstTouch.y - lastTouch.y));

    switch (shapeType) {
        case kLineShape:
            CGContextMoveToPoint(context, firstTouch.x, firstTouch.y);
            CGContextAddLineToPoint(context, lastTouch.x, lastTouch.y);
            CGContextStrokePath(context);
            break;
        case kRectShape:
            CGContextAddRect(context, currentRect);
            CGContextDrawPath(context, kCGPathFillStroke);
            break;
        case kEllipseShape:
            CGContextAddEllipseInRect(context, currentRect);
            CGContextDrawPath(context, kCGPathFillStroke);
            break;
        case kImageShape:
            break;
        default:
            break;
    }
}

```

由于我们希望将椭圆形和矩形涂上纯色，因此我们添加一个使用 `currentColor` 设置填充颜色的调用：

```
CGContextSetFillColorWithColor(context, currentColor.CGColor);
```

接下来，我们声明一个 `CGRect` 变量。我们将使用 `currentRect` 来存放由用户拖动描述的矩形。请记住，`CGRect` 包含两个成员：`size` 和 `origin`。通过 `CGRectMake()` 函数，我们可以通过指定 `x`、`y`、`width` 和 `height` 值来创建 `CGRect`，因此可用于绘制矩形。绘制矩形的代码乍看上去有点吓人，但实际上并没有那么复杂。用户可以向任何方向拖动，因此起点因拖动方向而异。我们使用两个点

中较小的x值和两个点中较小的y值来创建起点。然后通过获得两个x值和两个y值之差的绝对值来计算出大小：

```
CGRect currentRect = CGRectMake (
    (firstTouch.x > lastTouch.x) ? lastTouch.x : firstTouch.x,
    (firstTouch.y > lastTouch.y) ? lastTouch.y : firstTouch.y,
    fabsf(firstTouch.x - lastTouch.x),
    fabsf(firstTouch.y - lastTouch.y));
```

定义此矩形之后，绘制矩形或椭圆形就像调用两个函数一样轻松，一个函数是绘制矩形或在我们定义的CGRect中绘制椭圆形，另一个函数是绘画并填充它。

```
case kRectShape:
    CGContextAddRect(context, currentRect);
    CGContextDrawPath(context, kCGPathFillStroke);
    break;
case kEllipseShape:
    CGContextAddEllipseInRect(context, currentRect);
    CGContextDrawPath(context, kCGPathFillStroke);
    break;
```

编译并运行应用程序，并试用Rect和Ellipse工具，看看你有多喜欢它们。不要忘记不时更改颜色和试用随机颜色。

12.4.8 绘制图像

我们的最后一件事是绘制图像。12 QuartzFun文件夹中包含一个名为iphone.png的图像，你可以将该图像添加到Resources文件夹中，或者也可以添加你要使用的任何.png文件，只要记得将代码中的文件名改为所选图像即可。

向drawRect:方法中添加以下代码：

```
- (void)drawRect:(CGRect)rect {

    if (currentColor == nil)
        self.currentColor = [UIColor redColor];

    CGContextRef context = UIGraphicsGetCurrentContext();

    CGContextSetLineWidth(context, 2.0);
    CGContextSetStrokeColorWithColor(context, currentColor.CGColor);

    CGContextSetFillColorWithColor(context, currentColor.CGColor);
    CGRect currentRect;
    currentRect = CGRectMake (
        (firstTouch.x > lastTouch.x) ? lastTouch.x : firstTouch.x,
        (firstTouch.y > lastTouch.y) ? lastTouch.y : firstTouch.y,
        fabsf(firstTouch.x - lastTouch.x),
        fabsf(firstTouch.y - lastTouch.y));

    switch (shapeType) {
```

```

    case kLineShape:
        CGContextMoveToPoint(context, firstTouch.x, firstTouch.y);
        CGContextAddLineToPoint(context, lastTouch.x, lastTouch.y);
        CGContextStrokePath(context);
        break;
    case kRectShape:
        CGContextAddRect(context, currentRect);
        CGContextDrawPath(context, kCGPathFillStroke);
        break;
    case kEllipseShape:
        CGContextAddEllipseInRect(context, currentRect);
        CGContextDrawPath(context, kCGPathFillStroke);
        break;
    case kImageShape: {
        CGFloat horizontalOffset = drawImage.size.width / 2;
        CGFloat verticalOffset = drawImage.size.height / 2;
        CGPoint drawPoint = CGPointMake(lastTouch.x - horizontalOffset,
                                         lastTouch.y - verticalOffset);
        [drawImage drawAtPoint:drawPoint];
        break;
    }
    default:
        break;
}
}
}

```

提示 注意，在switch语句中，我们在case kImageShape:下的代码两侧添加了花括号。GCC在case语句之后的第一行中声明变量时遇到了问题。这些花括号是我们告诉GCC停止抱怨的一种方式。我们还在switch语句之前声明了horizontalOffset，该方法将相关代码放到了一起。

首先，我们计算该图像的中心，因为我们希望绘制的图像以用户上次触摸的点为中心。如果不进行调整，则会在用户手指的左上角绘制该图像，这也是一个有效的选项。然后通过从lastTouch中的x和y值中减去这些偏移量来生成一个新的Cgpoint。

```

CGFloat horizontalOffset = drawImage.size.width / 2;
CGFloat verticalOffset = drawImage.size.height / 2;
CGPoint drawPoint = CGPointMake(lastTouch.x - horizontalOffset,
                                lastTouch.y - verticalOffset);

```

现在，我们通知图像绘制自身。此行代码将进行这项工作：

```
[drawImage drawAtPoint:drawPoint];
```

应用程序如预期执行，但我们应该考虑进行一些优化。在该应用程序中，你不会注意到速度减慢，但是在更复杂的应用程序（在速度较慢的处理器上运行）中，你会看到某些延迟。该问题由touchesMoved:和touchesEnded:方法中的QuartzFunView.m引起。这两个方法都包含下面这行

代码：

```
[self setNeedsDisplay];
```

很明显，这是我们告知视图重新绘制自身的方式。该代码正常工作，但它导致整个视图被擦除并重新绘制，即使只是非常微小的更改也是如此。当我们准备拖动新形状时，我们希望擦除该屏幕，但我们不希望在拖动形状时一秒钟清除屏幕好几次。

为避免在拖动期间多次强制重新绘制整个视图，我们可以使用 `setNeedsDisplayInRect:`。`setNeedsDisplayInRect:` 是一个 `NSView` 方法，该方法会将视图区域的一个矩形部分标记为需要重新显示。我们需要重新绘制的不仅仅是 `firstTouch` 和 `lastTouch` 之间的矩形，还有当前拖动所包围的任何屏幕部分。如果用户触摸屏幕，然后在屏幕上到处乱画，则只需重新绘制 `firstTouch` 和 `lastTouch` 之间的部分，将许多不需要的已绘制的内容留在屏幕上。

答案是跟踪受 `CGRect` 实例变量中的特定拖动影响的整个区域。在 `touchesBegan:` 中，我们将该实例变量重置为仅用户触摸的点。然后在 `touchesMoved:` 和 `touchesEnded:` 中，使用一个 `Core Graphics` 函数获取当前矩形和存储的矩形的并集，然后存储所得到的矩形。此外，还使用该函数指定需要重新绘制的视图部分。该方法为我们提供了受当前拖动影响的正在运行的全部区域。

我们立刻在 `drawRect:` 方法中计算当前矩形，以便绘制椭圆形和矩形形状。我们将该计算结果移动到新方法中，以便在所有3个位置中使用此新方法，而没有重复代码。准备好了吗？让我们开始吧。对 `QuartzFunView.h` 进行以下更改：

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

@interface QuartzFunView : UIView {
    CGPoint      firstTouch;
    CGPoint      lastTouch;
    UIColor      *currentColor;
    ShapeType    shapeType;
    UIImage      *drawImage;
    CGRect       redrawRect;
}
@property CGPoint firstTouch;
@property CGPoint lastTouch;
@property (nonatomic, retain) UIColor *currentColor;
@property ShapeType shapeType;
@property (nonatomic, retain) UIImage *drawImage;
@property (readonly) CGRect currentRect;
@property CGRect redrawRect;
@end
```

我们声明了一个名为 `redrawRect` 的 `CGRect`，我们将使用它来跟踪需要重新绘制的区域。我们还声明了一个名为 `currentRect` 的只读属性，该属性将返回我们之前在 `drawRect:` 中计算的矩形。注意，这个属性没有底层实例变量。

切换到 `QuartzFunView.m` 并进行以下更改：

```

#import "QuartzFunView.h"

@implementation QuartzFunView
@synthesize firstTouch;
@synthesize lastTouch;
@synthesize currentColor;
@synthesize shapeType;
@synthesize drawImage;
@synthesize redrawRect;
@dynamic currentRect;
- (id)initWithFrame:(CGRect)frame {
    if (self = [super initWithFrame:frame]) {
    }
    return self;
}
- (CGRect)currentRect {
    return CGRectMake (
        (firstTouch.x > lastTouch.x) ? lastTouch.x : firstTouch.x,
        (firstTouch.y > lastTouch.y) ? lastTouch.y : firstTouch.y,
        fabsf(firstTouch.x - lastTouch.x),
        fabsf(firstTouch.y - lastTouch.y));
}
- (void)drawRect:(CGRect)rect {

    if (currentColor == nil)
        self.currentColor = [UIColor redColor];

    CGContextRef context = UIGraphicsGetCurrentContext();

    CGContextSetLineWidth(context, 2.0);
    CGContextSetStrokeColorWithColor(context, currentColor.CGColor);

    CGContextSetFillColorWithColor(context, currentColor.CGColor);
    CGRect currentRect = CGRectMake (
        (firstTouch.x > lastTouch.x) ? lastTouch.x : firstTouch.x,
        (firstTouch.y > lastTouch.y) ? lastTouch.y : firstTouch.y,
        fabsf(firstTouch.x - lastTouch.x),
        fabsf(firstTouch.y - lastTouch.y));

    switch (shapeType) {
        case kLineShape:
            CGContextMoveToPoint(context, firstTouch.x, firstTouch.y);
            CGContextAddLineToPoint(context, lastTouch.x, lastTouch.y);
            CGContextStrokePath(context);
            break;
        case kRectShape:
            CGContextAddRect(context, self.currentRect);
            CGContextDrawPath(context, kCGPathFillStroke);
            break;
    }
}

```



```

        case kEllipseShape:
            CGContextAddEllipseInRect(context, self.currentRect);
            CGContextDrawPath(context, kCGPathFillStroke);
            break;
        case kImageShape:
            if (drawImage == nil)
                self.drawImage = [UIImage imageNamed:@"iphone.png"];

            CGFloat horizontalOffset = drawImage.size.width / 2;
            CGFloat verticalOffset = drawImage.size.height / 2;
            CGPoint drawPoint = CGPointMake(lastTouch.x - horizontalOffset,
                                             lastTouch.y - verticalOffset);
            [drawImage drawAtPoint:drawPoint];
            break;
        default:
            break;
    }
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    firstTouch = [touch locationInView:self];
    lastTouch = [touch locationInView:self];

    if (shapeType == kImageShape) {
        CGFloat horizontalOffset = drawImage.size.width / 2;
        CGFloat verticalOffset = drawImage.size.height / 2;
        redrawRect = CGRectMake(firstTouch.x - horizontalOffset,
                                firstTouch.y - verticalOffset,
                                drawImage.size.width, drawImage.size.height);
    }
    else
        redrawRect = CGRectMake(firstTouch.x, firstTouch.y, 0, 0);
    [self setNeedsDisplay];
}

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

    [self setNeedsDisplay];
    if (shapeType == kImageShape) {
        CGFloat horizontalOffset = drawImage.size.width / 2;
        CGFloat verticalOffset = drawImage.size.height / 2;
        redrawRect = CGRectUnion(redrawRect, CGRectMake(lastTouch.x -
                                                         horizontalOffset, lastTouch.y - verticalOffset,
                                                         drawImage.size.width, drawImage.size.height));
    }
}

```

```

    else
        redrawRect = CGRectUnion(redrawRect, self.currentRect);
        redrawRect = CGRectInset(redrawRect, -2.0, -2.0);
        [self setNeedsDisplayInRect:redrawRect];
    }
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    lastTouch = [touch locationInView:self];

    [self setNeedsDisplay];
    if (shapeType == kImageShape) {
        CGFloat horizontalOffset = drawImage.size.width / 2;
        CGFloat verticalOffset = drawImage.size.height / 2;
        redrawRect = CGRectUnion(redrawRect,
            CGRectMake(lastTouch.x - horizontalOffset,
                lastTouch.y - verticalOffset, drawImage.size.width,
                drawImage.size.height));
    }
    redrawRect = CGRectUnion(redrawRect, self.currentRect);
    [self setNeedsDisplayInRect:redrawRect];
}
- (void)dealloc {
    [currentColor release];
    [drawImage release];
    [super dealloc];
}
@end

```

仅增加了几行代码，我们就减少了重新绘制视图所需的大量工作（不再需要擦除和重新绘制未受当前拖动影响的视图部分）。像这样妥善处理iPhone宝贵的处理器周期，可以在应用程序性能方面产生巨大差别，尤其是当应用程序变得更加复杂时。

12.5 一些 OpenGL ES 基础知识

如前所述，OpenGL ES和Quartz 2D采用完全不同的方法进行绘图。对OpenGL ES的详细介绍本身就是一本书，因此我们在此不对其进行讨论。我们使用OpenGL ES重新创建我们的Quartz 2D应用程序，只是为了让你对其有个基本了解，并且向你提供一些示例代码，你可以依据这些代码实现自己的OpenGL应用程序。

说明 准备在你自己的应用程序中添加OpenGL时，请顺便浏览一下<http://www.khronos.org/opengles/>，该网页是OpenGL ES标准组的主页。更好的做法是访问此页并搜索单词“tutorial”：<http://www.khronos.org/developers/resources/opengles/>。

让我们开始创建应用程序吧。

构建 GLFun 应用程序

在Xcode中，创建一个基于视图的新应用程序，并将其命名为GLFun。为了节省时间，将文件Constants.h、UIColor-Random.h、UIColor-Random.m和iphone.png从Quartz-Fun项目复制到这个新项目中。打开GLFunViewController.h并进行以下更改。你应该能够识别它们，因为它们与我们之前对QuartzFunViewController.h进行的更改相同：

```
#import <UIKit/UIKit.h>
#import "Constants.h"
@interface GLFunViewController : UIViewController {
    IBOutlet UISegmentedControl *colorControl;
}
@property (nonatomic, retain) UISegmentedControl *colorControl;
- (IBAction)changeColor:(id)sender;
- (IBAction)changeShape:(id)sender;
@end
```

切换到QuartzFunViewController.m并进行以下更改。你应该对它们非常熟悉：

```
#import "GLFunViewController.h"
#import "GLFunView.h"
#import "UIColor-Random.h"

@implementation GLFunViewController
@synthesize colorControl;
- (IBAction)changeColor:(id)sender {
    UISegmentedControl *control = sender;
    NSInteger index = [control selectedSegmentIndex];

    GLFunView *glView = (GLFunView *)self.view;

    switch (index) {
        case kRedColorTab:
            glView.currentColor = [UIColor redColor];
            glView.useRandomColor = NO;
            break;
        case kBlueColorTab:
            glView.currentColor = [UIColor blueColor];
            glView.useRandomColor = NO;
            break;
        case kYellowColorTab:
            glView.currentColor = [UIColor yellowColor];
            glView.useRandomColor = NO;
            break;
        case kGreenColorTab:
            glView.currentColor = [UIColor greenColor];
            glView.useRandomColor = NO;
            break;
        case kRandomColorTab:
            glView.useRandomColor = YES;
    }
}
```

```

        break;
    default:
        break;
    }
}
- (IBAction)changeShape:(id)sender {
    UISegmentedControl *control = sender;
    [(GLFunView *)self.view setShapeType:[control selectedSegmentIndex]];
    if ([control selectedSegmentIndex] == kImageShape)
        [colorControl setHidden:YES];
    else
        [colorControl setHidden:NO];
}
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [colorControl release];
    [super dealloc];
}

@end

```

这与QuartzFunController.m之间的唯一差别是，我们引用一个名为GLFunView的视图，而不是名为QuartzFunView的视图。进行绘图的代码包含在UIView的子类中。由于这次我们进行绘图的方式完全不同，因此使用一个新类来包含绘图代码比较有意义。

继续操作之前，你需要在项目中添加几个文件。在12 GLFun文件夹中，你可以找到4个文件，名称分别为Texture2D.h、Texture2D.m、OpenGL ES2DView.h和OpenGL ES2DView.m。这些文件中的代码是由苹果公司编写的，它们使得在OpenGL ES中绘制图像更加容易。如果你愿意，可以在自己的程序中随意使用这些文件。

OpenGL ES本身并没有sprite或图像；它具有纹理，但纹理必须绘制到对象上。在OpenGL ES中绘制图像的方法是绘制一个多边形，然后将纹理映射到该多边形上，以便它与多边形的大小完全匹配。Texture2D将相对比较复杂的过程封装到一个易于使用的类中。

OpenGL ES2Dview是一个UIView子类，它使用OpenGL进行绘图。我们设置此视图的目的是便

于在一对一的基础上映射OpenGL ES的坐标系和视图的坐标系。OpenGL ES是一个三维系统。OpenGL ES2Dview将OpenGL 3-D世界映射到2-D视图的像素。注意，尽管视图和OpenGL上下文之间是一对一的关系，但是y坐标仍然是翻转的，因此我们必须将y坐标从视图坐标系（y增加表示向下移动）转换为OpenGL坐标系（y增加表示向上移动）。

若要使用OpenGL ES2Dview类，首先要将其子类化，然后实现draw方法进行实际绘图。还可以在视图中实现所需的任何其他方法，如与触摸有关的方法。

使用UIView子类模板创建一个新文件，并将其命名为GLFunView.m，必须创建它的头文件。现在，你可以双击GLFunViewController.xib，然后设计其界面。这次我们不会指导你进行该过程，但是如果你不明白的话，可以参考12.4.5节，以便获得具体步骤。

完成之后，保存并返回到Xcode。单击GLFunView.h并进行以下更改：

```
#import <UIKit/UIKit.h>
#import "Constants.h"
#import "Texture2D.h"
#import "OpenGL ES2DView.h"

@interface GLFunView : UIView {
    CGPoint firstTouch;
    CGPoint lastTouch;
    UIColor *currentColor;
    BOOL useRandomColor;

    ShapeType shapeType;

    Texture2D *sprite;
}
@property CGPoint firstTouch;
@property CGPoint lastTouch;
@property (nonatomic, retain) UIColor *currentColor;
@property BOOL useRandomColor;
@property ShapeType shapeType;
@property (nonatomic, retain) Texture2D *sprite;
@end
```

此类与QuartzFunView.h非常相似，但它不使用UIImage来存放图像，我们使用Texture2D来简化将图像绘制到OpenGL ES上下文中的过程。我们还将超类从UIView改为OpenGL ES2Dview，以便视图支持使用OpenGL ES进行二维绘图。

切换到GLFunView.m并进行以下更改。

```
#import "GLFunView.h"

@implementation GLFunView
@synthesize firstTouch;
@synthesize lastTouch;
@synthesize currentColor;
@synthesize useRandomColor;
```

```
@synthesize shapeType;
@synthesize sprite;
- (id)initWithCoder:(NSCoder*)coder
{
    if (self = [super initWithCoder:coder]) {
        self.currentColor = [UIColor redColor];
        self.useRandomColor = NO;
    }
    return self;
}

- (void)draw
{
    glLoadIdentity();

    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    CGColorRef color = currentColor.CGColor;
    const CGFloat *components = CGColorGetComponents(color);
    CGFloat red = components[0];
    CGFloat green = components[1];
    CGFloat blue = components[2];

    glColor4f(red, green, blue, 1.0);

    switch (shapeType) {
        case kLineShape: {
            if (sprite){
                [sprite release];
                self.sprite = nil;
            }

            GLfloat vertices[4];

            // Convert coordinates
            vertices[0] = firstTouch.x;
            vertices[1] = self.frame.size.height - firstTouch.y;
            vertices[2] = lastTouch.x;
            vertices[3] = self.frame.size.height - lastTouch.y;
            glLineWidth(2.0);
            glVertexPointer (2, GL_FLOAT , 0, vertices);
            glDrawArrays (GL_LINES, 0, 2);

            break;
        }
        case kRectShape:{
            if (sprite){
                [sprite release];
                self.sprite = nil;
            }
        }
    }
}
```

```

    }
    // Calculate bounding rect and store in vertices
    GLfloat vertices[8];
    GLfloat minX = (firstTouch.x > lastTouch.x) ?
        lastTouch.x : firstTouch.x;
    GLfloat minY = (self.frame.size.height - firstTouch.y >
        self.frame.size.height - lastTouch.y) ?
        self.frame.size.height - lastTouch.y :
        self.frame.size.height - firstTouch.y;
    GLfloat maxX = (firstTouch.x > lastTouch.x) ?
        firstTouch.x : lastTouch.x;
    GLfloat maxY = (self.frame.size.height - firstTouch.y >
        self.frame.size.height - lastTouch.y) ?
        self.frame.size.height - firstTouch.y :
        self.frame.size.height - lastTouch.y;

    vertices[0] = maxX;
    vertices[1] = maxY;
    vertices[2] = minX;
    vertices[3] = maxY;
    vertices[4] = minX;
    vertices[5] = minY;
    vertices[6] = maxX;
    vertices[7] = minY;

    glVertexPointer (2, GL_FLOAT , 0, vertices);
    glDrawArrays (GL_TRIANGLE_FAN, 0, 4);
    break;
}
case kEllipseShape: {
    if (sprite){
        [sprite release];
        self.sprite = nil;
    }
    GLfloat vertices[720];
    GLfloat xradius = (firstTouch.x > lastTouch.x) ?
        (firstTouch.x - lastTouch.x)/2 :
        (lastTouch.x - firstTouch.x)/2;
    GLfloat yradius = (self.frame.size.height - firstTouch.y >
        self.frame.size.height - lastTouch.y) ?
        ((self.frame.size.height - firstTouch.y) -
            (self.frame.size.height - lastTouch.y))/2 :
        ((self.frame.size.height - lastTouch.y) -
            (self.frame.size.height - firstTouch.y))/2;
    for (int i = 0; i <= 720; i+=2)
    {
        GLfloat xOffset = (firstTouch.x > lastTouch.x) ?
            lastTouch.x + xradius
            : firstTouch.x + xradius;

```



```

        GLfloat yOffset = (self.frame.size.height - firstTouch.y >
            self.frame.size.height - lastTouch.y) ?
            self.frame.size.height - lastTouch.y + yradius :
            self.frame.size.height - firstTouch.y + yradius;
        vertices[i] = (cos(degreesToRadian(i))*xradius) + xOffset;
        vertices[i+1] = (sin(degreesToRadian(i))*yradius) +
            yOffset;

    }
    glVertexPointer (2, GL_FLOAT , 0, vertices);
    glDrawArrays (GL_TRIANGLE_FAN, 0, 360);
    break;

}
case kImageShape:
    if (sprite == nil) {
        self.sprite = [[Texture2D alloc] initWithImage: [UIImage
            imageNamed:@"iphone.png"]];
        glBindTexture(GL_TEXTURE_2D, sprite.name);
    }
    [sprite drawAtPoint:CGPointMake(lastTouch.x,
        self.frame.size.height - lastTouch.y)];

    break;
default:
    break;
}

glBindRenderbufferOES(GL_RENDERBUFFER_OES, viewRenderbuffer);
[context presentRenderbuffer:GL_RENDERBUFFER_OES];
}

- (void)dealloc {
    [currentColor release];
    [sprite release];
    [super dealloc];
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    if (useRandomColor)
        self.currentColor = [UIColor randomColor];

    UITouch* touch = [[event touchesForView:self] anyObject];
    firstTouch = [touch locationInView:self];
    lastTouch = [touch locationInView:self];
    [self draw];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event

```

```

{
    UITouch *touch = [touches anyObject];
    lastTouch = [touch locationInView:self];

    [self draw];
}

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

    [self draw];
}
@end

```

不难看出，使用OpenGL或使用任何方法都不如使用Quartz 2D那么简洁直观。尽管它比Quartz更加强大，但是同时也更加复杂。有时OpenGL可能会令人畏缩。

由于该视图是从nib中加载的，因此我们添加了一个initWithCoder:方法，在该方法中，我们创建了一个UIColor并将其分配给currentColor。我们还将useRandomColor的默认值设置为NO。但是，我们尚未创建Texture2D对象。由于我们绘制的形状没有纹理，因此不需要加载纹理。如果加载纹理，OpenGL ES将在绘制多边形时尝试使用纹理。因此，我们需要采取一些步骤以确保在绘制其他形状时不加载纹理。处理此问题的首选方法是延迟加载纹理。

initWithCoder:方法后面是draw方法，你可以在该方法中看到两个库之间的实际差别。让我们看一看绘制直线的过程。下面就是在Quartz版本（我们已经删除了与绘图无关的代码）中绘制直线的方法：

```

CGContextRef context = UIGraphicsGetCurrentContext();
CGContextSetLineWidth(context, 2.0);
CGContextSetStrokeColorWithColor(context, currentColor.CGColor);
CGContextMoveToPoint(context, firstTouch.x, firstTouch.y);
CGContextAddLineToPoint(context, lastTouch.x, lastTouch.y);
CGContextStrokePath(context);

```

下面是在OpenGL中绘制相同的直线所需采取的步骤。首先，我们重置虚拟机世界以便删除任何旋转、转换或已经应用于它的其他变换：

```

glLoadIdentity();

```

接下来，我们将背景清除为白色：

```

glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);

```

之后，我们必须通过分割UIColor并从中拖出各个RGB组件来设置OpenGL绘图颜色。所幸，我们不必担心UIColor使用的是哪种颜色模型。我们可以安全地假设它将使用RGBA颜色空间：

```
CGColorRef color = currentColor.CGColor;
const CGFloat *components = CGColorGetComponents(color);
CGFloat red = components[0];
CGFloat green = components[1];
CGFloat blue = components[2];
glColor4f(red, green, blue, 1.0);
```

若要绘制直线，我们需要两个顶点，这意味着我们需要一个包含4个元素的数组。前面讨论过，二维空间中的点由两个值（即x和y）表示。在Quartz中，我们使用一个CGPoint struct来存放这些点。在OpenGL中，点未嵌入到struct中。相反，我们用组成需要绘制的形状的所有点来填充数组。因此，若要在OpenGL ES中绘制一条从点(100, 150)到点(200, 250)的直线，我们将创建一个如下所示的顶点数组：

```
vertex[0] = 100;
vertex[1] = 150;
vertex[2] = 200;
vertex[3] = 250;
```

我们的数组格式为{x1, y1, x2, y2, x3, y3}。该方法中的下一段代码将两个CGPoint结构转换为顶点数组：

```
GLfloat vertices[4];
vertices[0] = firstTouch.x;
vertices[1] = self.frame.size.height - firstTouch.y;
vertices[2] = lastTouch.x;
vertices[3] = self.frame.size.height - lastTouch.y;
```

定义描述我们需要绘制的内容（在本例中为直线）的顶点数组之后，指定线宽，使用方法glVertexPointer()将该数组传递到OpenGL ES中，并通知OpenGL ES绘制数组：

```
glLineWidth(2.0);
glVertexPointer(2, GL_FLOAT, 0, vertices);
glDrawArrays(GL_LINES, 0, 2);
```

在OpenGL ES中完成绘图之后，我们必须告诉它渲染其缓冲器，并且告诉我们的视图上下文显示新渲染的缓冲器：

```
glBindRenderbufferOES(GL_RENDERBUFFER_OES, viewRenderbuffer);
[context presentRenderbuffer:GL_RENDERBUFFER_OES];
```

为了阐明在OpenGL中绘图的过程由3个步骤组成。首先，在上下文中绘图。其次，完成所有绘图之后，将上下文呈现到缓冲器中。第三，呈现渲染缓冲器，即当像素实际绘制到屏幕上时。

正如你所见，OpenGL示例比较长。当查看绘制椭圆的过程时，Quartz 2D和OpenGL ES之间的差别变得更加明显。OpenGL ES不知道如何绘制椭圆。OpenGL是OpenGL ES的老大哥甚至前辈，它有许多生成常见的二维和三维形状的便利函数，而这些便利函数只是从OpenGL ES分离出来的一部分功能，这使得OpenGL更加精简并且更加适合在嵌入式设备（如iPhone）中使用。因此，更多责任落在了开发人员的身上。

作为提示，下面是我们使用Quartz 2D绘制椭圆的方法：

```

CGContextRef context = UIGraphicsGetCurrentContext();
CGContextSetLineWidth(context, 2.0);
CGContextSetStrokeColorWithColor(context, currentColor.CGColor);
CGContextSetFillColorWithColor(context, currentColor.CGColor);
CGRect currentRect;
CGContextAddEllipseInRect(context, self.currentRect);
CGContextDrawPath(context, kCGPathFillStroke);

```

对于OpenGL ES版本, 开始的步骤与之前相同, 重置任何移动或旋转, 将背景清除为白色以及基于currentColor设置绘图颜色。

```

glLoadIdentity();
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
CGColorRef color = currentColor.CGColor;
const CGFloat *components = CGColorGetComponents(color);
CGFloat red = components[0];
CGFloat green = components[1];
CGFloat blue = components[2];
glColor4f(red, green, blue, 1.0);

```

由于OpenGL ES不知道如何绘制椭圆, 因此我们必须自己绘制, 这意味着需要面对复杂的几何理论。我们将定义一个顶点数组, 该数组存放720个GLfloat, 这将存放360个点的x和y位置, 围绕圆一度一个。我们可以更改点数来提高或降低此圆的平滑度。这种方法在将适合iPhone屏幕的任何视图上看起来都不错, 但如果你绘制的所有内容是比较小的圆, 则严格来讲可能需要进行更多处理。

```
GLfloat vertices[720];
```

接下来, 我们将根据存储在firstTouch和lastTouch中的两个点计算此椭圆的水平半径和垂直半径。

```

GLfloat xradius = (firstTouch.x > lastTouch.x) ?
    (firstTouch.x - lastTouch.x)/2 :
    (lastTouch.x - firstTouch.x)/2;
GLfloat yradius = (self.frame.size.height - firstTouch.y >
    self.frame.size.height - lastTouch.y) ?
    ((self.frame.size.height - firstTouch.y) -
    (self.frame.size.height - lastTouch.y))/2 :
    ((self.frame.size.height - lastTouch.y) -
    (self.frame.size.height - firstTouch.y))/2;

```

然后, 我们将围绕圆进行循环, 计算围绕圆的正确的点:

```

for (int i = 0; i <= 720; i+=2) {
    GLfloat xOffset = (firstTouch.x > lastTouch.x) ?
        lastTouch.x + xradius : firstTouch.x + xradius;
    GLfloat yOffset = (self.frame.size.height - firstTouch.y >
        self.frame.size.height - lastTouch.y) ?
        self.frame.size.height - lastTouch.y + yradius :
        self.frame.size.height - firstTouch.y + yradius;

```

```
vertices[i] = (cos(degreesToRadian(i))*xradius) + xOffset;  
vertices[i+1] = (sin(degreesToRadian(i))*yradius) + yOffset;  
}
```

最后，我们将顶点数组提供给OpenGL ES，通知OpenGL ES绘制并渲染它，然后通知上下文呈现新渲染的图像：

```
glVertexPointer (2, GL_FLOAT , 0, vertices);  
glDrawArrays (GL_TRIANGLE_FAN, 0, 360);  
glBindRenderbufferOES(GL_RENDERBUFFER_OES, viewRenderbuffer);  
[context presentRenderbuffer:GL_RENDERBUFFER_OES];
```

我们不会审查矩形方法，因为它使用的基本技巧相同；我们定义一个顶点数组，该数组中的4个顶点用于定义矩形，然后我们渲染并呈现它。我们也不对绘制图像进行过多描述，因为苹果公司提供的可爱的Texture2D类，使得绘制一个小精灵就像在Quartz 2D中绘制一样容易。

在draw方法之后，我们拥有与之前版本相同的与触摸有关的方法。唯一差别是它不告知视图它需要显示，而是调用我们刚才定义的draw方法。不需要告知OpenGL ES将更新屏幕的哪些部分。它会计算出来并且利用硬件加速以最高效的方式绘制。但是，在编译和运行此程序之前，需要将这两个框架链接到你的项目。按照第5章有关添加Core Graphics框架的说明，然后选择OpenGL ES.framework和QuartzCore.framework（而不是选择CoreGraphics.framework）。

现在，你已经看到了OpenGL ES入门的足够内容。如果你对在iPhone应用程序中使用OpenGL ES感兴趣，可以在<http://www.khronos.org/opengles/>上查找OpenGL ES规范以及与OpenGL ES相关的书籍、文档和论坛的链接。

提示 如果你想创建一个全屏的OpenGL ES应用程序，不必手动构建它。Xcode为你提供了一个实用的模板。该模板为你设置了屏幕和缓冲器，甚至将一些示例绘图和动画代码放置到类中，以便你可以看到放置你的代码的位置。想尝试吗？新建一个iPhone OS应用程序，然后选择OpenGL ES Application模板。

12.6 小结

在本章中，我们真的只抓到了iPhone绘图功能的一点皮毛。现在，你应该逐渐适应了Quartz 2D。并且，通过参考苹果公司的文档，你可以满足遇到的大多数绘图需求。你还应该对什么是OpenGL ES及其如何与iPhone的视图系统集成有个基本了解。

下一步做什么呢？你将学习如何在应用程序中添加手势支持。

简洁明亮、每英寸160像素、支持触摸操作的iPhone屏幕确实很漂亮，并且其多触摸屏幕为iPhone赋予了无法比拟的可用性。由于该屏幕可以同时检测多个触摸并且可以单独跟踪这些触摸，因此应用程序能够检测到大范围的手势，从而为用户提供超出该界面之外的功能。

假设你在邮件应用程序中浏览收件箱，并且决定删除某封电子邮件。你可以轻击Edit按钮，选择该行，然后轻击Delete按钮，就这3个步骤。或者，你只是用手指轻扫要删除的行，然后轻击弹出的Delete按钮，就这两个步骤。

本例只介绍了通过iPhone的多触摸屏幕可以完成的无数手势之一。你可以将手指捏在一起来放大图片，或者松开手指缩小图片。可以双击Mobile Safari中的某个框进行缩放，以便该框占据你的整个屏幕。你可以用两个手指轻扫一个可滚动的视图，如长长的网页或电子邮件消息，视图将会随着你的手指上下滚动。

本章将介绍用于检测手势的底层体系结构。你将了解如何检测最常用的手势以及如何创建和检测全新的手势。

13.1 多触摸术语

在钻研体系结构之前，让我们复习一下一些基本词汇。首先，**手势**（gesture）是指从你用一个或多个手指接触屏幕时开始，直到你的手指离开屏幕为止所发生的所有事件。无论它花费多长时间，只要一个或多个手指仍然在屏幕上，你就仍然位于某个手势之中（除非传入电话呼叫等系统事件中断该手势）。手势在事件中传递到系统。事件将在用户与iPhone的多触摸屏幕交互时生成，其中包含有关发生的一个或多个触摸的信息。

很明显，术语**触摸**（touch）是指手指放到iPhone的屏幕上。手势中涉及的触摸数量等于同时位于屏幕上的手指数量。实际上，你可以将所有五个手指都放到屏幕上，只要这些手指彼此不要靠太近，iPhone就能够识别并跟踪所有的手指。现在，还没有太多实用的五指手势，但是知道iPhone能够处理这种情况（如果需要）比较好。

当用一个手指触摸屏幕，然后立即将该手指离开屏幕（而不是来回移动）时发生**轻击**（tap）。iPhone跟踪轻击的数量，并且可以告诉你用户轻击了两次还是3次，甚至是20次。例如，它处理所有计时以及其他必要的工作来区分两个单击还是一次双击。iPhone只跟踪使用一个手指时的轻击，记住这一点非常重要。如果它检测到多个触摸，则会将轻击计数重置为1。

13.2 响应者链

由于手势是在事件之内传递到系统的，然后事件会传递到响应者链（responder chain），因此你需要了解响应者链的工作方式，以便正确处理手势。如果你使用过Cocoa for Mac OS X，你可能会熟悉响应者链的概念，因为Cocoa和Cocoa Touch中使用的基本机制相同。如果这是新知识，那也不必担心，我们会解释它的工作原理。

在本书中，我们已经多次提到过第一响应者，该响应者通常是用户当前正在交互的对象。第一响应者是响应者链的开始，还有其他响应者。让UIResponder作为其超类之一的任何类都是响应者。UIView是UIResponder的子类，UIControl是UIView的子类，因此所有视图和所有控件都是响应者。UIViewController也是UIResponder的子类，这意味着它是响应者，其所有子类（如UINavigationController和UITabBarController）也都是响应者。那么响应者就是这样命名的，因为它们响应系统生成的事件，如屏幕触摸。

如果第一个响应者不处理某个特殊事件（如某个手势），则它会将该事件传递到响应者链的下一级。如果该链中的下一个对象响应此特殊事件，则它通常会处理该事件，这将停止该事件沿着响应者链向前传递。在某些情况下，如果某个响应者只对某个事件进行部分处理，则该响应者将采取操作，并将该事件转发给链中的下一个响应者。但通常这不会发生这种情况。正常情况下，当对象响应事件时，即到达了该事件的行尾。如果事件通过整个响应者链并且没有对象处理该事件，则丢弃该事件。

下面让我们再更具体地看一下响应者链。第一个响应者几乎总是视图或控件，并且首先对事件进行响应。如果第一个响应者不处理该事件，则它会将该事件传递给其视图控制器。如果此视图控制器不处理该事件，则将该事件传递给第一个响应者的父视图。如果父视图没有响应，则该事件将转到父视图的控制器（如果有）。该事件将沿着每个视图的视图层次结构继续前进，然后该视图的控制器获得处理该事件的机会。如果该事件一直通过视图层次结构，则会将该事件传递给应用程序的窗口。如果窗口不处理该事件，则该窗口会将该事件传递给应用程序的对象实例UIApplication。如果UIApplication不响应该事件，则该事件逐渐进入睡眠状态。

这个过程非常重要，这有多个原因。首先它控制可以处理手势的方式。比如说，一个用户正在查看某个表，他用某个手指轻扫该表的某一行。哪个对象会处理该手势呢？

如果是在某个视图或控件之内轻扫，而该视图或控件是表视图单元的子视图，则该视图或控件将获得响应的机会。如果它没有响应，则表视图单元会获得机会。在某个应用程序（如邮件应用程序）中，可以使用轻扫操作来删除某个邮件，表视图单元可能需要查看该事件，看它是否包含轻扫手势。但是大多数表视图单元不响应手势，如果它们不响应，则该事件将继续通过表视图，然后通过其他响应者，直到某些内容响应该事件或者达到行的结尾为止。

转发事件：保持响应者链的活动状态

让我们回到邮件应用程序中的表视图单元。我们不知道苹果公司邮件应用程序的内部细节，但是我们暂且可以认为表视图单元支持轻扫式删除且仅支持轻扫式删除。该表视图单元必须实现

与接收触摸事件（你将在几分钟之后看到）相关的方法，以便它可以进行检查，看该事件是否包含轻扫手势。如果该事件包含轻扫，则表视图单元会采取操作，该事件将停止传递。

如果该事件不包含轻扫手势，则表视图单元负责将该事件手动转发给响应者链中的下一个对象。如果它没有进行转发的工作，则表和链上的其他对象将永远也不会获得响应的机会，并且该应用程序可能无法如用户所期望地正常工作。该表视图单元可能会阻止其他视图识别手势。

只要你响应触摸事件，就必须记住代码无法在真空中工作。如果某个对象截获某个它无法处理的事件，则也需要通过在下一个响应者上调用相同的方法来手动传递该对象。下面是其中一小部分代码：

```
-(void)respondToFictionalEvent:(UIEvent *)event {
    if (someCondition)
        [self handleEvent:event];
    else
        [self.nextResponder respondToFictionalEvent:event];
}
```

注意我们在下一个响应者上调用相同方法的方式。这就是成为响应者链良好公民的方式。好在大多数情况下响应事件的方法还处理事件，但是如果不是这种情况，你必要确保将事件推回到响应者链中，这一点非常重要。

13.3 多触摸体系结构

对响应者链有了一定的了解之后，让我们看一下处理手势的过程吧。如前所述，手势沿着响应者链传递，并且嵌入在事件中。这意味着在响应者链的对象中，需要包含代码来处理与多触摸屏幕进行的任何种类的交互。一般来说，这意味着我们可以选择将该代码嵌入到UIView的子类中，也可以将该代码嵌入到UIView Controller中。

那么该代码属于视图还是属于视图控制器？

如果视图需要根据用户的触摸来对自己执行某些操作，则代码可能属于定义该视图的类。例如，很多控件类（如UISwitch和UISlider）都响应与触摸有关的事件。UISwitch可能希望根据触摸来打开或关闭自身。创建UISwitch类的人将处理手势的代码嵌入到该类中，因此UISwitch可以响应某个手势。

但是，通常当正在处理的手势影响正在触摸的多个对象时，该手势代码才真正属于视图的控制器类。例如，如果用户对一行进行手势触摸，该触摸指出应该删除所有行，则应该由视图控制器中的代码来处理该手势。无论代码属于哪个类，在这两种情况下响应触摸和手势的方式都完全相同。

4 个手势通知方法

可以使用4个方法通知响应者有关触摸和手势的情况。当用户第一次触摸屏幕时，iPhone将查找touchesBegan:withEvent:方法的响应者。若要查清用户第一次开始进行手势或轻击屏幕的时间，请在你的视图或视图控制器中实现该方法。下面是该方法的示例：

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {  
  
    NSUInteger numTaps = [[touches anyObject] tapCount];  
    NSUInteger numTouches = [touches count];  
  
    // Do something here.  
}
```

该方法以及所有与触摸有关的方法都传递一个NSSet实例和一个UIEvent实例。你可以通过获取touches中的对象数来确定当前按压屏幕的手指数目。touches中的每个对象都是一个UITouch事件，该事件表示一个手指正在触摸屏幕。如果该触摸是一系列轻击的一部分，则可以通过询问任何UITouch对象来查询轻击数量。当然，如果触摸中有多个对象，则轻击计数必将为1，因为只有只使用一个手指来轻击屏幕时，系统才保留轻击计数。在前面的示例中，如果numTouches为2，则用户只是轻击屏幕两次。

touches中的所有对象都可能与你实现该方法的视图或视图控制器无关。例如，表视图单元可能并不关心其他行中的触摸或者导航栏中的触摸。你可以从事件中获得一个子集，它仅拥有位于特殊视图中的触摸的touches，如下所示：

```
NSSet *myTouches = [event touchesForView:self.view];
```

每个UITouch都表示不同的手指，并且每个手指都位于屏幕上的不同位置。你可以使用UITouch对象查询特定手指的位置。如果需要，甚至可以将点转换为视图的本地坐标系，如下所示：

```
CGPoint point = [touch locationInView:self];
```

当用户将手指移过屏幕时，你可以通过实现touchesMoved:withEvent:来获得通知。在长时间的拖动过程中会多次调用该方法，并且每次调用该方法时，将获得另一组触摸以及另一个事件。除了能够从UITouch对象获得每个手指的当前位置之外，还可以查清该触摸的原来的位置，这是上次调用touchesMoved:withEvent:或touchesBegan:withEvent:时手指的位置。

当用户的手指离开屏幕时，会调用另一个事件，即touchesEnded:withEvent:。调用该方法时，你知道用户是在使用手势。

响应者可以实现的最后一个与触摸有关的方法是touchesCancelled:withEvent:。当发生某些事件导致手势中断时会调用该方法。可以在此处进行任何所需的清理工作，以便你可以重新开始一个新手势。调用该方法时，对于当前手势，将不会调用touchesEnded:withEvent:。

现在，理论已经很充分了，下面让我们看看其中一些理论的实践吧。

13.4 触摸浏览器应用程序

我们将构建一个小应用程序，当调用4个与触摸有关的响应者方法时，该应用程序会使你感觉更好。在Xcode中，使用基于视图的应用程序模板创建新项目，并调用新项目TouchExplorer。每次调用与触摸有关的方法时，TouchExplorer都会将消息打印到屏幕中，包含触摸和轻击计数（参见图13-1）。

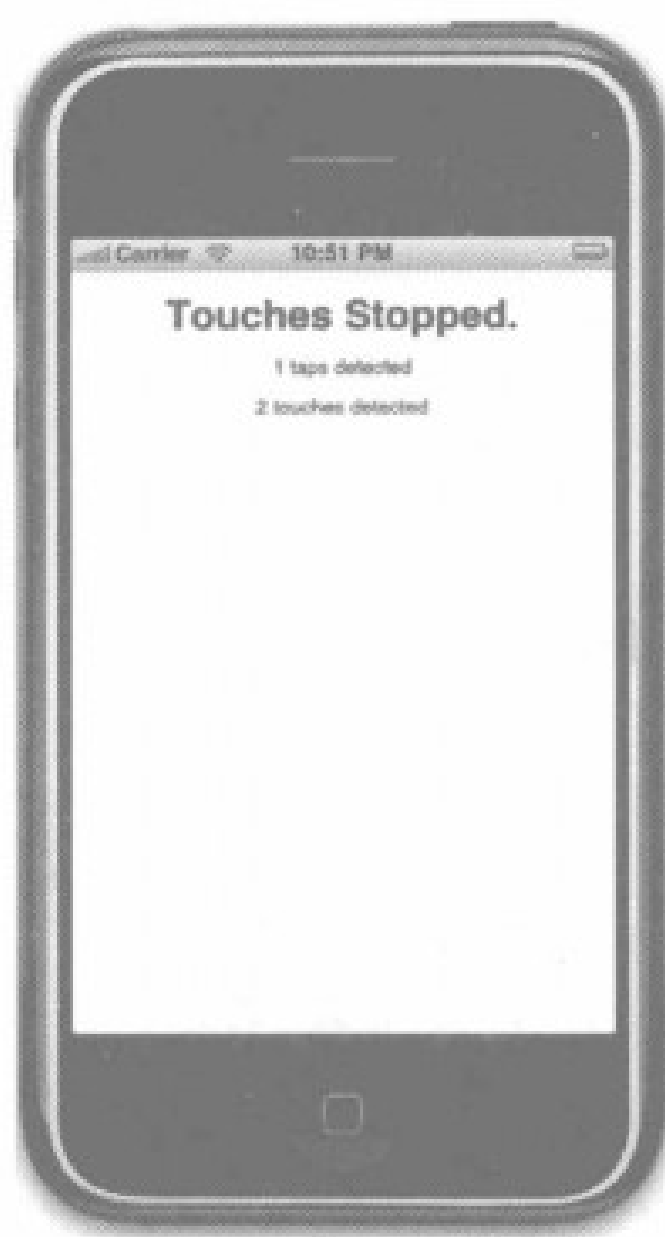


图13-1 触摸浏览器应用程序

我们需要为该应用程序提供3个标签：一个用于指示最后调用的方法，一个用于报告当前轻击计数，第三个用于报告触摸数量。单击TouchExplorerViewController.h并添加3个输出口和一个方法声明。将使用该方法更新多个位置中的标签。

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

```
@interface TouchExplorerViewController : UIViewController {
    IBOutlet UILabel *messageLabel;
    IBOutlet UILabel *tapsLabel;
    IBOutlet UILabel *touchesLabel;
}
@property (nonatomic, retain) UILabel *messageLabel;
@property (nonatomic, retain) UILabel *tapsLabel;
@property (nonatomic, retain) UILabel *touchesLabel;
- (void)updateLabelsFromTouches:(NSSet *)touches;
@end
```

说明 尽管本章中的应用程序将在仿真器上运行，但是你将无法看到所有可用的多触摸功能，除非你在iPhone或iPod Touch上运行这些应用程序。如果你已经加入开发人员计划，则可以在你选择的设备上运行你编写的程序。苹果公司网站进行了大量工作，这些工作可以引导你完成准备连接到你的设备所需的任何内容。可能你在入门时需要一点帮助，我们将一些有帮助的评论放在了附录中。

现在，双击TouchExplorerViewController.xib，在Interface Builder中打开该文件。如果未打开标题为View的窗口，则双击View图标以将其打开。从库中拖动3个Labels到View窗口中。你应该调整标签的大小，以便它们占满视图的整个宽度并且将文本居中，而标签的确切位置并不要紧。

如果你感觉有点过于单调，还可以设置字体和颜色。完成放置它们后，双击每个标签，并按delete键以删除其中的文本。

按住Control的同时，将File's Owner图标拖到这3个标签中的每个标签，从而将一个连接到messageLabel输出口，一个连接到tapsLabel输出口，最后一个连接到touchesLabel输出口。最后，单击View图标并按⌘1打开属性检查器（参见图13-2）。在检查器中，确保同时选中User Interacting Enabled和Multiple Touch。如果未选中Multiple Touch，你的控制器类的触摸方法将始终接受一个并且只接受一个触摸，无论实际上有多少手指触摸电话的屏幕都是如此。

完成后，保存并关闭nib，然后回到Xcode。

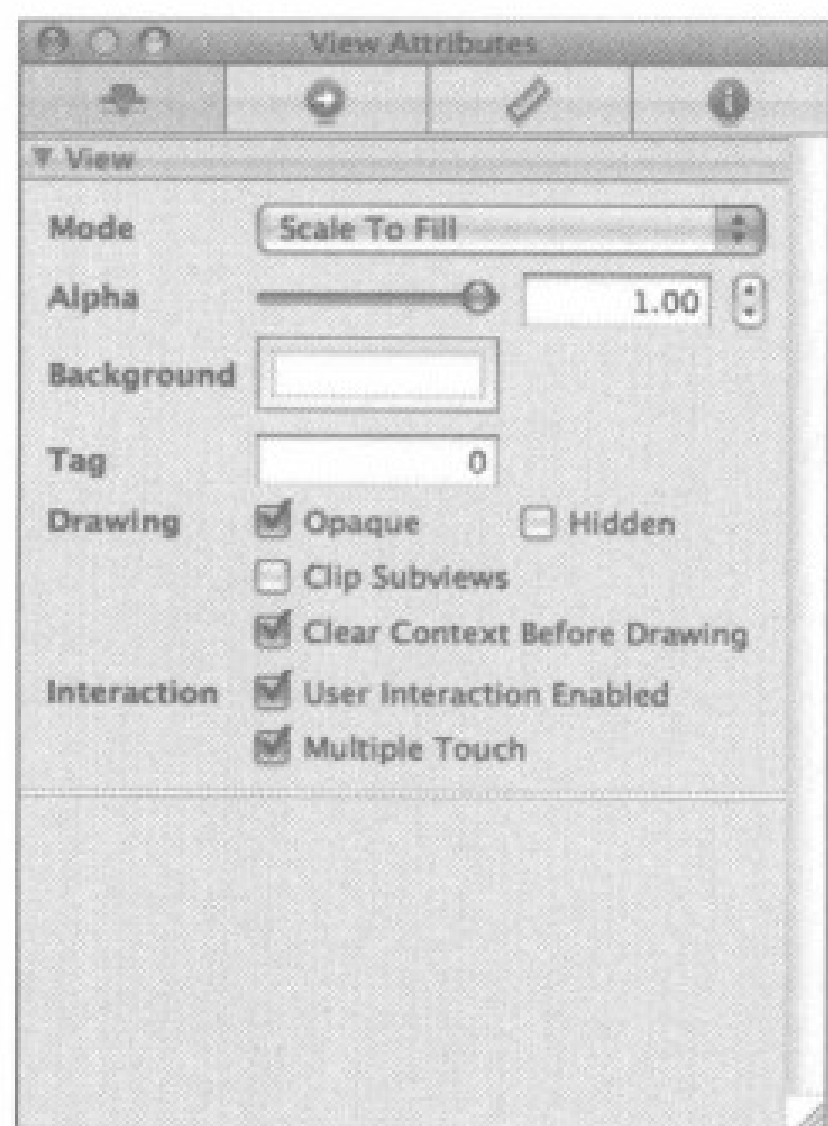
图13-2 确保将视图设置为接收多触摸事件

单击TouchExplorerViewController.m并添加以下代码：

```
#import "TouchExplorerViewController.h"

@implementation TouchExplorerViewController
@synthesize messageLabel;
@synthesize tapsLabel;
@synthesize touchesLabel;
- (void)updateLabelsFromTouches:(NSSet *)touches {
    NSUInteger numTaps = [[touches anyObject] tapCount];
    NSString *tapsMessage = [[NSString alloc]
        initWithFormat:@"%d taps detected", numTaps];
    tapsLabel.text = tapsMessage;
    [tapsMessage release];

    NSUInteger numTouches = [touches count];
    NSString *touchMsg = [[NSString alloc] initWithFormat:
        @"%d touches detected", numTouches];
    touchesLabel.text = touchMsg;
    [touchMsg release];
}
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}
```



```

- (void)dealloc {
    [messageLabel release];
    [tapsLabel release];
    [touchesLabel release];
    [super dealloc];
}
#pragma mark -
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {

    messageLabel.text = @"Touches Began";
    [self updateLabelsFromTouches:touches];

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

    messageLabel.text = @"Touches Cancelled";
    [self updateLabelsFromTouches:touches];
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {

    messageLabel.text = @"Touches Stopped.";
    [self updateLabelsFromTouches:touches];
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {

    messageLabel.text = @"Drag Detected";
    [self updateLabelsFromTouches:touches];

}

@end

```

在此控制器类中，我们实现了之前讨论的所有4个与触摸有关的方法。每个方法都设置了messageLabel，以便用户可以看到调用每个方法的时间。接下来，所有这4个方法都调用updateLabelsFromTouches:来更新其他两个标签。updateLabelsFromTouches:方法从其中一个触摸动作获得轻击计数，通过查看touches集的计数来计算出触摸的数量，并用该信息更新标签。

编译并运行该应用程序。如果是在仿真器中运行应用程序，则尝试重复单击屏幕来提高轻击计数，尝试在视图周围单击和在按住鼠标的情况下拖动鼠标来模拟触摸和拖动。可以在用鼠标单击并进行拖动时按下option键，以此在iPhone仿真器中模仿两个手指捏合的手势。还可以这样来模仿两个手指的轻扫手势：首先按下option键来模仿两个手指捏合，然后移动鼠标以便表示虚拟手指的两个点彼此相互靠近，然后再按下shift键（同时仍然按下option键）。按shift键将锁定两个手指相对于彼此的位置，并且可以进行轻扫和其他两个手指的手势。你将无法进行需要3个或多个手指的手势，但你可以使用option和shift键的组合来进行大多数两个手指的手势。

如果能够在iPhone或iPod Touch上运行该程序，则查看进行多少触摸才能同时注册。尝试使用一个手指进行拖动，再使用两个手指，然后使用3个手指。尝试轻击两次和轻击3次屏幕，看看

通过用两个手指轻击是否可以增加轻击计数。

尝试使用TouchExplorer应用程序，直到你适应4个触摸方法的工作方式为止。完成之后，让我们看看如何检测最常用的手势之一，即轻扫。

13.5 Swipe 应用程序

再次使用基于视图的应用程序模板在Xcode中创建一个新项目，这次将该项目命名为Swipes。我们将要构建的应用程序，除了检测水平和垂直轻扫之外，什么也不做（参见图13-3）。如果你用手指从左向右、从右到左、从上到下或从下到上轻扫屏幕，则Swipes将在屏幕顶部显示一条消息，保持几秒钟，用于告知你已检测到轻扫。

检测轻扫操作相对来说比较容易。我们将以像素为单位定义最小手势长度，即将该手势算作轻扫之前，用户必须轻扫的长度。我们还将定义一个偏差，即用户可以从某条直线偏离多远，仍然可以将该手势算作水平或垂直轻扫。通常不会将对角线算作轻扫，但是如果只是与水平或垂直轻扫偏离一点，也会将其当成轻扫。

当用户触摸屏幕时，我们将第一次触摸的位置保存在变量中，然后，当用户手指移动着通过屏幕时，我们将进行检查，看它是否达到某个点，这个点足够远且足够直，以便能够算作轻扫。让我们构建该应用程序吧。单击SwipesViewController.h并添加以下代码：

```
#define kMinimumGestureLength    25
#define kMaximumVariance        5

#import <UIKit/UIKit.h>

@interface SwipesViewController : UIViewController {
    IBOutlet UILabel *label;

    CGPoint gestureStartPoint;
}
@property (nonatomic, retain) UILabel *label;
@property CGPoint gestureStartPoint;
- (void)eraseText;
@end
```

首先，我们将最小手势长度定义为25像素，偏差定义为5。如果用户正在进行水平轻扫，则超过起始垂直位置上或下5像素时，该手势将结束，但只要用户水平移动了25像素，就仍然算作轻扫。

在实际的应用程序中，你可能需要使用这些数字来查清在应用程序界面上的工作内容。

我们还为一个标签声明了一个输出口和一个容纳用户触摸的第一个点的变量。我们所要做的最后一件事情就是声明一个方法，几秒钟之后将使用该方法擦除文本。

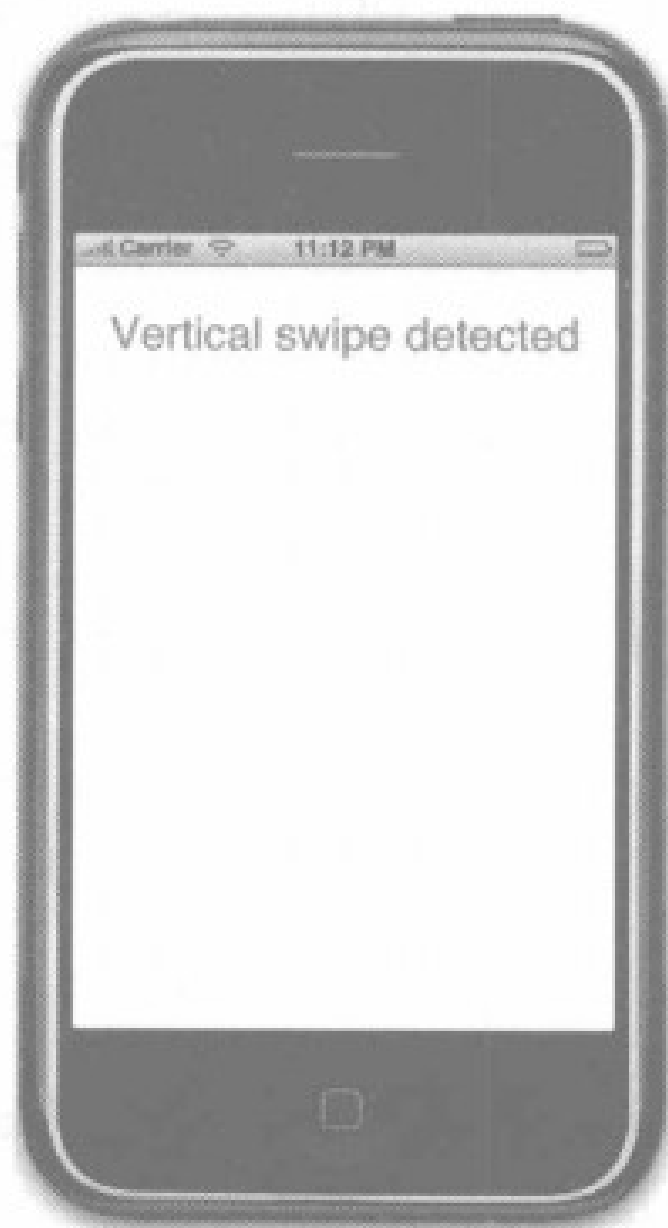


图13-3 Swipes应用程序

双击SwipesViewController.xib，在Interface Builder中打开该文件。一定要使用属性检查器将视图设置为接收多个触摸，并从库中拖出一个Label到View窗口上。设置该标签以便它占据视图的整个宽度，使用蓝色引导线作为参考，并随意使用文本属性使该标签易于阅读。接下来，双击该标签并删除其文本。按住Control的同时，将File's Owner图标拖到该标签上，并将其连接到该标签输出口。保存nib，关闭并返回到Xcode。

单击SwipesViewController.m并添加以下代码。完成后，我们将讨论该代码所执行的操作：

```
#import "SwipesViewController.h"

@implementation SwipesViewController
@synthesize label;
@synthesize gestureStartPoint;
- (void)eraseText
{
    label.text = @"";
}
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [label release];
    [super dealloc];
}
#pragma mark -
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    gestureStartPoint = [touch locationInView:self.view];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    CGPoint currentPosition = [touch locationInView:self.view];

    CGFloat deltaX = fabsf(gestureStartPoint.x - currentPosition.x);
    CGFloat deltaY = fabsf(gestureStartPoint.y - currentPosition.y);

    if (deltaX >= kMinimumGestureLength && deltaY <= kMaximumVariance) {
```



```

        label.text = @"Horizontal swipe detected";
        [self performSelector:@selector(eraseText)
            withObject:nil afterDelay:2];
    }
    else if (deltaY >= kMinimumGestureLength &&
        deltaX <= kMaximumVariance){
        label.text = @"Vertical swipe detected";
        [self performSelector:@selector(eraseText) withObject:nil
            afterDelay:2];
    }
}
@end

```

让我们从`touchesBegan:withEvent:`方法开始。此处，我们所有要做的就是从`touches`集中获得触摸并存储它的点。现在我们主要对一个手指轻扫感兴趣，因此我们不关心触摸数量有多少；我们只需要获取其中之一。

```

UITouch *touch = [touches anyObject];
gestureStartPoint = [touch locationInView:self.view];

```

在下一个方法`touchesMoved:withEvent:`中，我们进行实际的工作。首先，我们获取用户手指的当前位置：

```

UITouch *touch = [touches anyObject];
CGPoint currentPosition = [touch locationInView:self.view];

```

之后，我们计算用户手指从其起始位置开始，在水平和垂直方向上移动的距离。函数`fabsf()`来自标准C数学库，它返回一个类型为`float`的绝对值。这允许我们从一个中减去另一个，而不必关心哪个值较高：

```

CGFloat deltaX = fabsf(gestureStartPoint.x - currentPosition.x);
CGFloat deltaY = fabsf(gestureStartPoint.y - currentPosition.y);

```

实现两个增量之后，我们查看用户是否在一个方向上移动得足够远，但没有在另一个方向上移动太远，从而能够形成轻扫。如果是这样，我们将标签的文本设置为：它指出检测到的是水平轻扫还是垂直轻扫。我们还使用`performSelector:withObject:afterDelay:`在文本位于屏幕上2秒钟之后擦除文本。这样，用户便可以执行多个轻扫操作，而不必担心该标签是指之前的尝试还是指最近的尝试：

```

    if (deltaX >= kMinimumGestureLength && deltaY <= kMaximumVariance) {
        label.text = @"Horizontal swipe detected";
        [self performSelector:@selector(eraseText)
            withObject:nil afterDelay:2];
    }
    else if (deltaY >= kMinimumGestureLength &&
        deltaX <= kMaximumVariance){
        label.text = @"Vertical swipe detected";
        [self performSelector:@selector(eraseText)
            withObject:nil afterDelay:2];
    }
}

```

继续向前，编译并运行应用程序。如果你发现自己进行了单击和拖动，但没有看到结果。请耐心一点。单击并垂直向下或正对面拖动，直到你熟悉轻扫的用法。

13.6 实现多个轻扫

在Swipes应用程序中，我们仅关心一个手指的轻扫，因此我们只从触摸集中获取某个对象来计算出在轻扫期间用户手指的位置。如果你只对一个手指的轻扫感兴趣（这是所使用的最常见的轻扫类型），则该方法非常合适。

但是，如果我们想实现两个或三个手指的轻扫，则会遇到一点问题。这个问题就是：所提供的触摸将作为NSSet，而不是作为NSArray。这些集是没有顺序的集合，这意味着，当我们进行比较时，没有简单的方法来计算哪个手势使用了哪个手指。例如，我们不能假设该集中的第一次触摸所使用的手指是否与该手势开始后该集中的第一次触摸所使用的手指相同。

更糟糕的是，当用户执行两个或三个手指的手势时，完全有可能一个手指比另一个手指先触摸屏幕，这意味着在touchesBegan:withEvent:方法中，我们可能只能获得有关一个触摸的信息。

我们需要一种方法来检测多个手指的轻扫，而不会将其他手势（如捏合）错误识别为轻扫。解决方法非常简单。当touchesBegan:withEvent:通知某个手势已经开始时，我们保存一个手指的位置，就像我们之前做的那样。不需要保存所有手指的位置。其中任何一个手指都可以。

在检测轻扫时，我们遍历向touchesMoved:withEvent:方法提供的所有触摸，从而将每个触摸与已保存的点相比较。如果用户进行的是多手指轻扫，则当与保存的点进行比较时，至少我们在该方法中获得的其中一个触摸将指示轻扫。如果我们发现水平轻扫或垂直轻扫，则再次遍历触摸，并确保每个手指至少与第一个手指的水平或垂直位置保持最小距离（取决于轻扫类型）。现在让我们对Swipes应用程序进行改进，以便检测多手指轻扫。要实现此目的，我们需要对头文件进行少量更改，因此单击SwipesViewController.h并添加以下代码：

```
#define kMinimumGestureLength 25
#define kMaximumVariance 5
```

```
typedef enum {
    kNoSwipe = 0,
    kHorizontalSwipe,
    kVerticalSwipe
} SwipeType;
```

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

此枚举将提供一种方法，指出某个手势是水平轻扫还是垂直轻扫，或根本未检测到轻扫。现在，切换回SwipesViewController.m，并将touchesMoved:withEvent:方法完全替换为这个新版本：

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

    SwipeType swipeType = kNoSwipe;
    for (UITouch *touch in touches) {
```

```
CGPoint currentPosition = [touch locationInView:self.view];

CGFloat deltaX = fabsf(currentPosition.x-gestureStartPoint.x);
CGFloat deltaY = fabsf(currentPosition.y-gestureStartPoint.y);

if (deltaX >= kMinimumGestureLength &&
    deltaY <= kMaximumVariance)
    swipeType = kHorizontalSwipe;
else if (deltaY >= kMinimumGestureLength &&
    deltaX <= kMaximumVariance)
    swipeType = kVerticalSwipe;
}

BOOL allFingersFarEnoughAway = YES;
if (swipeType != kNoSwipe) {
    for (UITouch *touch in touches) {
        CGPoint currentPosition = [touch locationInView:self.view];

        CGFloat distance;
        if (swipeType == kHorizontalSwipe)
            distance = fabsf(currentPosition.x - gestureStartPoint.x);
        else
            distance = fabsf(currentPosition.y - gestureStartPoint.y);

        if (distance < kMinimumGestureLength)
            allFingersFarEnoughAway = NO;
    }
}
if (allFingersFarEnoughAway && swipeType != kNoSwipe)
{
    NSString *swipeCountString= nil;
    if ([touches count] == 2)
        swipeCountString = @"Double ";
    else if ([touches count] == 3)
        swipeCountString = @"Triple ";
    else if ([touches count] == 4)
        swipeCountString = @"Quadruple ";
    else
        swipeCountString = @"";

    NSString *swipeTypeString = (swipeType == kHorizontalSwipe) ?
        @"Horizontal" : @"Vertical";

    NSString *message = [[NSString alloc] initWithFormat:
        @"%@ Swipe Detected.", swipeCountString, swipeTypeString];
    label.text = message;
    [message release];
    [self performSelector:@selector(eraseText)
        withObject:nil afterDelay:2];
}
```

```
    }
}
```

编译并运行。你应该能够在两个方向触发两个手指和3个手指的轻扫，并且应该仍然能够触发一个手指的轻扫。如果你的手指比较小，你甚至可能触发4个手指的轻扫。

借助多手指轻扫，需要注意的一件事情就是，你的手指不能彼此太靠近。如果两个手指彼此靠得非常近，那么可能将它们注册为一个触摸。因此，你不应该依赖4个手指的轻扫来实现任何重要的手势，因为很多人的手指都比较大，不能有效地进行4个手指的轻扫。

13.7 检测多次轻击

在TouchExplorer应用程序中，我们将轻击计数打印到屏幕上，看到了吗，检测多次轻击是多么简单。但它并不像看上去那样简单，因为你通常希望根据轻击的数量来采取不同的操作。如果用户连续轻击3次，则程序会分3次单独通知你。你将得到轻击1次、轻击两次，最后得到轻击3次。如果你想对两次轻击执行某些完全不同于3次轻击的操作，拥有3个单独的通知可能会引起某个问题。让我们创建另一个应用程序进行演示并解决该问题。

在Xcode中，使用基于视图的应用程序模板创建一个新的项目。将此新项目命名为TapTaps。该应用程序将拥有4个标签，当它检测到轻击1次、轻击两次、轻击3次以及轻击4次时，它们会分别通知我们。在第一个版本的此应用程序中，所有4个字段将单独工作，因此如果轻击4次，你会得到所有4个轻击类型（参见图13-4）。

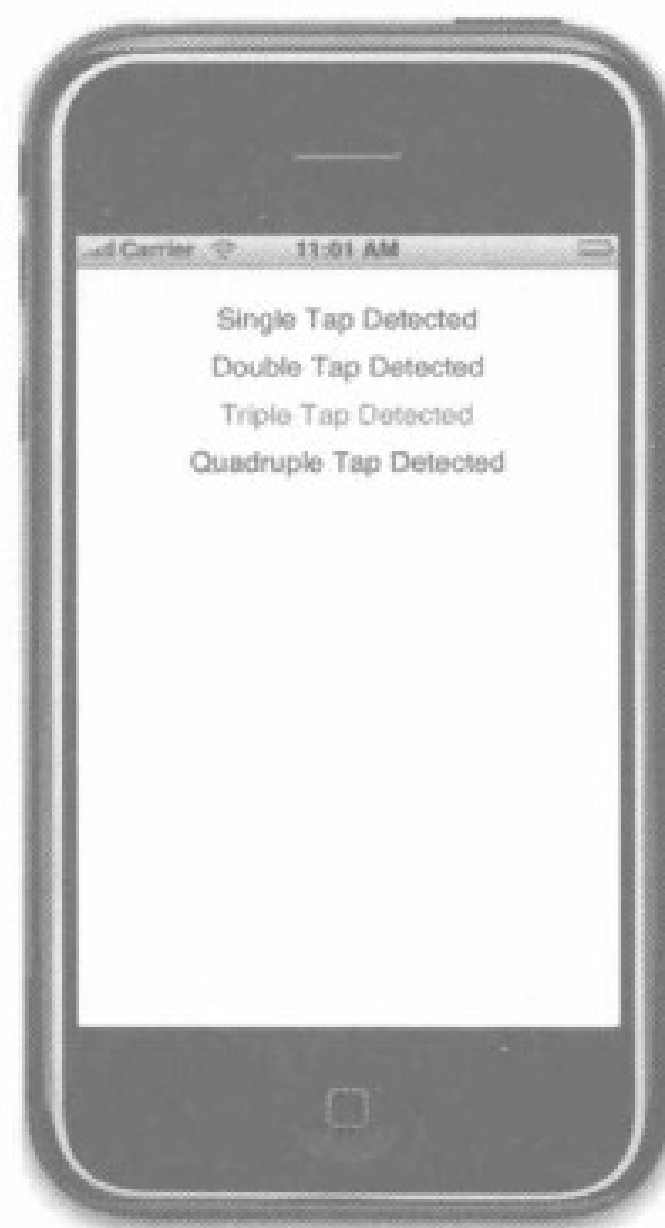


图13-4 同时检测所有轻击类型的TapTaps应用程序

此第一个版本运行之后，我们将查看如何更改它的行为，以便当用户停止轻击时只出现一个标签，该标签显示用户轻击的总数。

我们需要为4个标签提供输出口，并且我们还需要为每个轻击方案提供单独的方法，以便模拟你在实际应用程序中拥有的内容。还包括一个擦除文本字段的方法。展开Classes文件夹，单击TapTapsViewController.h并进行以下更改：

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

@interface TapTapsViewController : UIViewController {
    IBOutlet UILabel *singleLabel;
    IBOutlet UILabel *doubleLabel;
    IBOutlet UILabel *tripleLabel;
    IBOutlet UILabel *quadrupleLabel;
}
```

```

}
@property (nonatomic, retain) UILabel *singleLabel;
@property (nonatomic, retain) UILabel *doubleLabel;
@property (nonatomic, retain) UILabel *tripleLabel;
@property (nonatomic, retain) UILabel *quadrupleLabel;
- (void)singleTap;
- (void)doubleTap;
- (void)tripleTap;
- (void)quadrupleTap;
- (void)eraseMe:(UITextField *)textField ;
@end

```

保存它，然后展开Resources文件夹。双击TapTapsViewController.xib，在Interface Builder中打开该文件。打开该文件之后，确保将视图设置为接受多触摸，然后从库中向视图添加4个Labels。让这4个标签从蓝色引导线拉伸到蓝色引导线，然后安排它们的格式，直到你看着合适为止。我们选择使每个标签具有不同的颜色，但这不是必须的。完成后，一定要双击每个标签并按delete键删除所有文本。现在，按下Control的同时从File's Owner图标拖到每个标签，并将每个标签各自连接到single-Label、doubleLabel、tripleLabel和quadrupleLabel。完成该操作之后，保存并返回到Xcode。

在TapTapsViewController.m中进行以下更改：

```

#import "TapTapsViewController.h"

@implementation TapTapsViewController
@synthesize singleLabel;
@synthesize doubleLabel;
@synthesize tripleLabel;
@synthesize quadrupleLabel;
- (void)singleTap {
    singleLabel.text = @"Single Tap Detected";
    [self performSelector:@selector(eraseMe:)
        withObject:singleLabel afterDelay:1.6f];
}
- (void)doubleTap {
    doubleLabel.text = @"Double Tap Detected";
    [self performSelector:@selector(eraseMe:)
        withObject:doubleLabel afterDelay:1.6f];
}
- (void)tripleTap {
    tripleLabel.text = @"Triple Tap Detected";
    [self performSelector:@selector(eraseMe:)
        withObject:tripleLabel afterDelay:1.6f];
}
- (void)quadrupleTap {
    quadrupleLabel.text = @"Quadruple Tap Detected";
    [self performSelector:@selector(eraseMe:)
        withObject:quadrupleLabel afterDelay:1.6f];
}

```

```

- (void)eraseMe:(UITextField *)textField {
    textField.text = @"";
}
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [singleLabel release];
    [doubleLabel release];
    [tripleLabel release];
    [quadrupleLabel release];
    [super dealloc];
}
#pragma mark -
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    NSUInteger tapCount = [touch tapCount];
    switch (tapCount) {
        case 1:
            [self singleTap];
            break;
        case 2:
            [self doubleTap];
            break;
        case 3:
            [self tripleTap];
            break;
        case 4:
            [self quadrupleTap];
            break;
        default:
            break;
    }
}
@end

```

该应用程序中的4个轻击方法只是设置4个标签中的一个标签，并使用performSelector:withObject:afterDelay:在1.6秒之后擦除该标签。eraseMe:方法擦除传递给它的任何文本字段。

在下面的touchesBegan:withEvent:中，只要检测到适当的轻击数，就会调用这4个轻击方法。这非常简单，因此编译和运行应用程序。如果轻击两次，则会看到显示两个标签。如果轻击4次，

则会看到4个标签。在某些情况下，这可能没有问题，但是通常你希望基于用户结束操作时的轻击数量来采取操作。

注意，我们没有在该程序中实现`touchesEnded:withEvent:`或`touchesMoved:withEvent:`。没有通知我们用户已经停止轻击，这给我们出了一道难题。幸运的是，可以使用一个简单的方法来处理此问题。你已经熟悉方法`performSelector:withObject:afterDelay:`，它允许我们在将来的某个时间调用方法。另一个方法允许我们取消这些将来的调用。它是一个名为`cancelPreviousPerformRequestsWithTarget:selector:object:`的`NSObject`类方法。该方法将停止与传递到它的参数匹配的任何挂起的执行请求，并且它将帮助我们解决轻击的难题。在`TapTapsViewController.m`中，将`touchesBegan:withEvent:`方法替换为此新版本：

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    NSUInteger tapCount = [touch tapCount];

    switch (tapCount) {
        case 1:
            [self performSelector:@selector(singleTap)
                     withObject:nil
                     afterDelay:.4];
            break;
        case 2:
            [NSObject cancelPreviousPerformRequestsWithTarget:self
                     selector:@selector(singleTap)
                     object:nil];
            [self performSelector:@selector(doubleTap)
                     withObject:nil
                     afterDelay:.4];
            break;
        case 3:
            [NSObject cancelPreviousPerformRequestsWithTarget:self
                     selector:@selector(doubleTap)
                     object:nil];
            [self performSelector:@selector(tripleTap)
                     withObject:nil
                     afterDelay:.4];
            break;
        case 4:
            [NSObject cancelPreviousPerformRequestsWithTarget:self
                     selector:@selector(tripleTap)
                     object:nil];
            [self quadrupleTap];
            break;
        default:
            break;
    }
}
```


在此版本中，每次检测到多次轻击时，我们没有立即调用相应的方法，而是使用 `performSelector:withObject:afterDelay:` 在以后每秒的十分之四时调用这些方法，并且当接收到以前的轻击计数时，我们取消方法所作的执行请求。因此，当我们收到一个轻击时，我们在以后每秒的十分之四时调用 `singleTap` 方法。当我们收到一个轻击两次的通知时，我们取消对 `singleTap` 的调用，并在以后每秒的十分之四时调用 `doubleTap`。对于轻击3次、轻击4次，我们采取相同的操作，以便对于任何特殊的轻击序列，我们只调用4个方法中的一个。

编译并运行此版本，当轻击两次、轻击3次以及轻击4次时，你应该只看到显示的一个标签。

13.8 检测捏合操作

另一个常见的手势是两个手指的捏合。在很多应用程序（包括Mobile Safari、邮件和照片）中，使用它来放大（手指捏在一起）或缩小（手指松开）。

检测捏合非常简单。首先，当手势开始时，我们检查以确保存在两个触摸，因为捏合是两个手指的手势。如果有两个触摸，则存储它们之间的距离。然后，随着手势的进行，我们始终检测用户手指之间的距离，如果该距离增大或减小的量大于特定数量，我们便知道存在捏的手势。

再次使用基于视图的应用程序模板在Xcode中创建一个新项目，并将该项目命名为PinchMe。在该项目以及下一个项目中，我们将需要进行一些非常标准的解析几何运算，以计算某些内容，如两个点之间的距离（在该项目中）以及之后项目中的两条直线之间的角度。如果你不记得几何方法，也不用担心，我们为你提供了很多进行这些计算的函数。在13 PinchMe文件夹中找到两个名为CGPointUtils.h和CGPointUtils.c的文件。将这两个文件拖到你项目的Classes文件夹中。你可以在自己的应用程序中自由使用这些实用的函数。

PinchMe应用程序不仅需要有一个标签的一个输出口，而且还需要一个实例变量来存储手指之间的起始距离，以及一个用于擦除标签的方法（和以前的应用程序一样）。我们还将定义一个常量，该常量标识组成捏合手势的手指之间距离的最小更改。展开Classes文件夹，单击PinchMeViewController.h并进行以下更改：

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

#define kMinimumPinchDelta 100
@interface PinchMeViewController : UIViewController {
    IBOutlet UILabel *label;
    CGFloat initialDistance;
}
@property (nonatomic, retain) UILabel *label;
@property CGFloat initialDistance;
- (void)eraseLabel;
@end
```

创建输出口之后，现在展开Resources文件夹，并双击PinchMeView-Controller.xib。在Interface Builder中，确保将视图设置为接受多次触摸（在属性检查器上查找Multiple Touch Enabled复选

框)，并拖动一个标签到其上。你可以采用任何希望的方式放置、调整大小以及安排该标签的格式。对该标签执行完操作之后，双击该标签，并删除其所包含的文本。接下来，按住Control的同时从File's Owner图标拖到该标签，并将其连接到label输出口。保存并关闭nib，返回Xcode。

在PinchMeViewController.m中进行以下更改：

```
#import "PinchMeViewController.h"
#import "CGPointUtils.h"

@implementation PinchMeViewController
@synthesize label;
@synthesize initialDistance;
- (void)eraseLabel {
    label.text = @"";
}
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}
- (void)dealloc {
    [label release];
    [super dealloc];
}
#pragma mark -
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    if ([touches count] == 2) {
        NSArray *twoTouches = [touches allObjects];
        UITouch *first = [twoTouches objectAtIndex:0];
        UITouch *second = [twoTouches objectAtIndex:1];
        initialDistance = distanceBetweenPoints(
            [first locationInView:self.view],
            [second locationInView:self.view]);
    }
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    if ([touches count] == 2) {
        NSArray *twoTouches = [touches allObjects];
        UITouch *first = [twoTouches objectAtIndex:0];
        UITouch *second = [twoTouches objectAtIndex:1];
        CGFloat currentDistance = distanceBetweenPoints(
            [first locationInView:self.view],
            [second locationInView:self.view]);
```

```

        if (initialDistance == 0)
            initialDistance = currentDistance;
        else if (currentDistance - initialDistance > kMinimumPinchDelta) {
            label.text = @"Outward Pinch";
            [self performSelector:@selector(eraseLabel)
                withObject:nil
                afterDelay:1.6f];
        }
        else if (initialDistance - currentDistance > kMinimumPinchDelta) {
            label.text = @"Inward Pinch";
            [self performSelector:@selector(eraseLabel)
                withObject:nil
                afterDelay:1.6f];
        }
    }
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    initialDistance = 0;
}
@end

```

在`touchesBegan:withEvent:`方法中，我们查看此触摸是否涉及两个手指。如果涉及两个手指，我们将使用`CGPointUtils.c`中的一个方法计算这两个点之间的距离，并将结果存储在距离变量`initialDistance`中。

在`touchesMoved:withEvent:`中，我们再次检查是否有两次触摸，如果有的话，计算这两次触摸之间的距离：

```

if ([touches count] == 2) {
    NSArray *twoTouches = [touches allObjects];
    UITouch *first = [twoTouches objectAtIndex:0];
    UITouch *second = [twoTouches objectAtIndex:1];
    CGFloat currentDistance = distanceBetweenPoints(
        [first locationInView:self.view],
        [second locationInView:self.view]);
}

```

我们要做的下一件事是查看`initialDistance`是否为0。我们这样做是因为用户的手指可能在不同的时间触碰屏幕，因此，尽管涉及两个手指，但可能也不会调用`touchesBegan:withEvent:`。如果`initialDistance`为0，则表示这是两个手指接触屏幕的第一个点，并且我们将这两个点之间的当前距离存储为初始距离：

```

if (initialDistance == 0)
    initialDistance = currentDistance;

```

否则，我们查看从当前距离中减去初始距离之后，是否大于我们定义为算作捏合手势所需的最小更改的数量。如果大于的话，则判定为一个向外的捏合，因为现在该距离大于初始距离：

```

else if (currentDistance - initialDistance > kMinimumPinchDelta) {
    label.text = @"Outward Pinch";
    [self performSelector:@selector(eraseLabel)

```

```

        withObject:nil
        afterDelay:1.6f];
    }

```

如果不大于的话，我们通过查看初始距离减去当前距离之后，是否足够符合捏合手势，来进行另一项检查，看它是否是向内的捏：

```

else if (initialDistance - currentDistance > kMinimumPinchDelta) {
    label.text = @"Inward Pinch";
    [self performSelector:@selector(eraseLabel)
        withObject:nil
        afterDelay:1.6f];
}

```

捏合检测就是这些内容。编译和运行以进行尝试。如果你位于仿真器上，请记住你可以通过按下option键并在仿真器窗口中使用鼠标单击拖动来模仿捏合手势。

13.9 自己定义手势

现在，你已经了解了检测最常用的iPhone手势的方法。当你开始自定义手势时，才是真枪真炮地开始！

自己定义手势需要一些技巧。你现在已经掌握了基本的原理，因此这并不是太困难。在定义手势的组成时，需要技巧的部分非常灵活。大多数人无法准确知道使用手势的时间。记住我们在实现轻扫所使用的偏差，用于确定轻扫不是完全水平或垂直时是否仍然算数。这是一个需要你自己的手势定义中添加细微差别的完美示例。如果你将手势定义得太严格，那么它将没有什么用处。如果将它定义得太笼统，那么操作又会太灵活，这会使用户感到灰心。从某种意义上说，自己定义手势比较难，因为你必须确切知道某个手势的不精确之处。如果尝试捕获某个复杂的手势（比如说数字8），那么检测该手势背后的数学也将非常复杂。

在本例中，我们将定义一个形状像选中标记的手势（参见图13-5）。

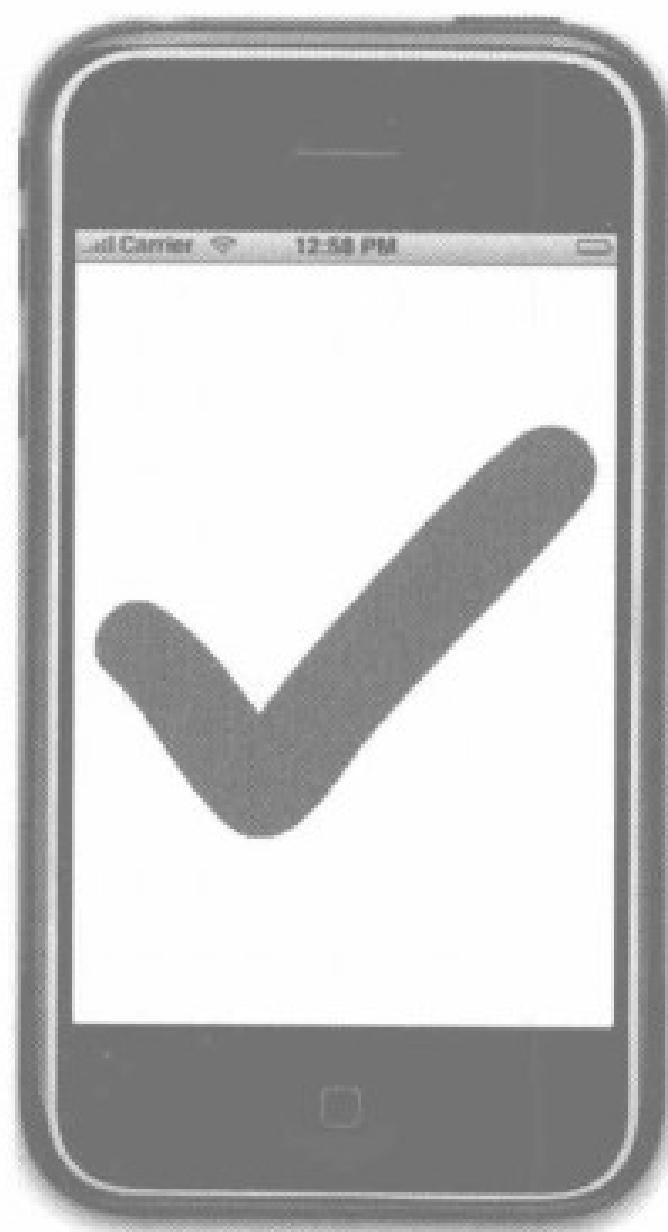


图13-5 选中标记手势的图例

定义此选中标记手势需要哪些属性呢？首要的一点是这两条直线之间角度的锐角变化。我们还希望确保在形成该锐角角度之前，用户的手指在直线上移动了一点距离。在图13-5中，选中标记的腿以某个尖锐的角度相交，仅小于90度。需要严格85度角的手势很难做对，因此我们将定义一个可接受角度的范围。

使用基于视图的应用程序模板在Xcode中创建一个新项目，并将该项目命名为CheckPlease。我们将需要来自CGPointUtils的一个函数，因此向该项目的Classes文件夹中添加CGPointUtils.h和CGPointUtils.c。

展开Classes文件夹，单击CheckPleaseViewController.h并进行以下更改：

```

#define kMinimumCheckMarkAngle 50
#define kMaximumCheckMarkAngle 135
#define kMinimumCheckMarkLength 10
#import <UIKit/UIKit.h>

@interface CheckPleaseViewController : UIViewController {
    IBOutlet UILabel *label;
    CGPoint lastPreviousPoint;
    CGPoint lastCurrentPoint;
    CGFloat lineLengthSoFar;
}
@property (nonatomic, retain) UILabel *label;
- (void)eraseLabel;
@end

```

可以看到，我们已经定义最小角度为50度且最大角度为135度。这是一个非常广的范围，根据需要，你可以决定限制该角度。我们对此进行了一些实验，并且发现实际的选中标记手势落入一个非常广泛的范围中，这就是为什么我们在这里选择一个相对比较大的容许量的原因。对于选中标记手势，我们有点粗糙，因此我们认为至少其中一些用户也是这样。

接下来，我们定义到某个标签的输出口，检测到选中标记手势之后，使用该输出口通知用户。我们还声明了lastPreviousPoint、lastCurrentPoint和lineLengthSoFar这3个变量。每次通知我们一个触摸时，就会提供之前的触摸点和当前触摸点。这两个点定义一个线段。下一个触摸增加另一条线段。我们将之前触摸的上一个点和当前点存储在lastPreviousPoint和lastCurrentPoint中，这两个点为我们提供了之前的线段。然后，我们可以将该线段与当前触摸的线段进行比较。通过比较，我们可以判断仍然在绘制一条直线，还是这两个线段之间有足够尖锐的角度（实际上我们正在绘制选中标记）。

记住，每个UITouch对象都知道其在视图中的当前位置以及其在视图中的之前位置。但是，要比较角度，我们需要知道之前两个点形成的直线，因此我们需要存储自上次用户触摸屏幕以来的当前点和之前的点。每次调用该方法时，我们都使用这两个变量存储这两个值，以便能够将当前直线与之前直线相比较并检测该角度。

我们还声明了一个变量，用于保存用户手指拖动的距离的运行计数。如果手指尚未行进到10个像素（kMinimumCheckMarkLength中的值），则角度是否落在正确的范围之内并不重要。如果我们不需要该距离，我们将得到很多主动错误信息。

展开Resources文件夹，双击CheckPleaseViewController.xib，在Interface Builder中将其打开。由于这是一个手指的手势，因此不要对视图启用多触摸支持，只需从库中添加一个Label，并按照所需的方式设置它。双击该标签以删除其文本，在按下Control的同时从File's Owner图标拖到该标签，以将其连接到此label输出口。保存nib文件。现在返回Xcode，单击CheckPleaseViewController.m并进行以下更改：

```

#import "CheckPleaseViewController.h"
#import "CGPointUtils.h"

```

```

@implementation CheckPleaseViewController
@synthesize label;
- (void)eraseLabel {
    label.text = @"";
}
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [label release];
    [super dealloc];
}
#pragma mark -
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    CGPoint point = [touch locationInView:self.view];
    lastPreviousPoint = point;
    lastCurrentPoint = point;
    lineLengthSoFar = 0.0f;
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {

    UITouch *touch = [touches anyObject];
    CGPoint previousPoint = [touch previousLocationInView:self.view];
    CGPoint currentPoint = [touch locationInView:self.view];
    CGFloat angle = angleBetweenLines(lastPreviousPoint,
                                      lastCurrentPoint,
                                      previousPoint,
                                      currentPoint);

    if (angle >= kMinimumCheckMarkAngle && angle <= kMaximumCheckMarkAngle
        && lineLengthSoFar > kMinimumCheckMarkLength) {
        label.text = @"Checkmark";
        [self performSelector:@selector(eraseLabel)
            withObject:nil afterDelay:1.6];
    }

    lineLengthSoFar += distanceBetweenPoints(previousPoint, currentPoint);
    lastPreviousPoint = previousPoint;
    lastCurrentPoint = currentPoint;
}

```

```
}  
@end
```

CheckPlease 触摸方法

让我们看一看触摸方法。首先，在`touchesBegan:withEvent:`中，我们确定哪个是用户当前正在触摸的点，并将该值存储在`lastPreviousPoint`和`lastCurrentPoint`中。由于该方法将在手势开始时调用，并且我们知道不需要担心之前的点，因此两个方法中都存储了当前点。我们还将正在运行的线长度计数设置为0。

然后，在`touchesMoved:withEvent:`中，我们计算从当前触摸的之前位置到其当前位置的直线，以及存储在`lastPreviousPoint`和`lastCurrentPoint`实例变量中两个点的直线之间的角度。计算出该角度之后，我们检查该角度是否处于可接受的角度范围之内，并确保在该锐角反向之前，用户的手指移开足够远。如果这两个条件都为真，则设置标签显示已经标识的选中标记手势。接下来，我们计算触摸的位置及其之前位置之间的距离，将该距离添加到`lineLengthSoFar`，并将`lastPreviousPoint`和`lastCurrentPoint`中的值替换为当前触摸中的两个点，以便我们下次通过该方法时能够拥有这两个点。

编译和运行应用程序，并试用该手势。

当为你自己的应用程序定义新手势时，确保仔细测试过它们，如果可以的话，还要让其他人帮助你进行测试。你一定希望确保你的手势对于用户来说容易操作，但也不是无意中就可以触发它那么容易。你还需要确保不与应用程序中的其他手势发生冲突。例如，不应该将一个手指的手势算作自定义手势和捏合手势。

13.10 小结

现在，你应该理解iPhone向应用程序通知有关触摸、轻击以及手势的信息的机制。你还了解了如何检测最常用的iPhone手势，以及如何自己定义手势。iPhone的接口在很大程度上依赖于手势，这要归功于手势的易用性。因此，在准备进行大多数iPhone开发时，你会希望拥有这些技术。

当你准备继续前进时，翻过此页，我们将告诉你如何使用Core Location计算你在世界中的位置。

我在哪里？使用 Core Location 定位功能

iPhone 可以使用 Core Location 框架确定它的物理位置。实际上，Core Location 可以利用三种技术来实现该功能：GPS、蜂窝基站三角网定位（cell tower triangulation）和 Wi-Fi 定位服务（WPS）。GPS 是三种技术中最精确的，但在第一代 iPhone 上不可用。GPS 从多个卫星读取微波信号来确定当前位置。蜂窝基站三角网定位根据手机所属范围内的手机基站的位置进行计算来确定当前位置。蜂窝基站三角网定位在城市和其他手机基站密度较高的区域非常精确，而在基站较为稀疏的区域则不太精确。最后一个选项 WPS 使用 iPhone 的 Wi-Fi 连接的 IP 地址，通过参考已知服务提供商及其服务区域的大型数据库来猜测你的位置。WPS 是不精确的，并且有时会有数英里的误差。

所有这 3 种方法都会显著消耗 iPhone 的电池，因此在使用 Core Location 时请记住这一点。除非绝对必要，否则不应该对你的位置进行多次轮询。使用 Core Location 时，可以根据需要指定精度。通过仔细指定所需的绝对最低精度级别，可以防止不必要的电池消耗。

Core Location 所依赖的技术隐藏在你的应用程序中。我们没有告知 Core Location 是使用 GPS、三角网还是 WPS。我们只是告知它我们希望的精度级别，它将决定哪种技术可以更好地满足你的请求。

14.1 位置管理器

Core Location API 实际上非常易于使用。我们将使用的主类是 `CLLocationManager`，通常称为位置管理器（Location Manager）。为了与 Core Location 进行交互，我们需要创建一个位置管理器实例，如下所示：

```
CLLocationManager *locationManager = [[CLLocationManager alloc] init];
```

这将为我们的位置管理器的一个实例，但是它实际上并未开始轮询我们的位置。我们创建一个委托，并将其分配给位置管理器。当位置信息可用时，位置管理器会调用我们的委托方法。这可能会花费一些时间，甚至需要几秒钟。我们的委托必须符合 `CLLocationManagerDelegate` 协议。

14.1.1 设置所需的精度

设置委托之后，你还会希望设置所需的精度。前面讲过，不要指定任何大于绝对需要的精度。如果你编写的应用程序只需要知道手机所在的州或国家，则不需要指定较高的精度级别。记住你要求的Core Location的精度越高，消耗的电量就会越多。还要记住，不能保证你会获得所需的精度级别。

下面是设置委托和请求指定精度级别的示例：

```
locationManager.delegate = self;
locationManager.desiredAccuracy = kCLLocationAccuracyBest;
```

精度是使用CLLocationAccuracy值进行设置的，类型定义为double。该值的单位为米（m），因此如果你指定的desiredAccuracy为10，则是告知Core Location你希望它尝试确定当前位置10m范围之内的区域。正如我们以前所做的一样，指定kCLLocationAccuracyBest会通知Core Location使用当前可用的最高精度的方法。除了kCLLocationAccuracyBest之外，还可以使用kCLLocationAccuracyNearestTenMeters、kCLLocationAccuracyHundredMeters、kCLLocationAccuracyKilometer和kCLLocationAccuracyThreeKilometers。

14.1.2 设置距离筛选器

默认情况下，位置管理器将通知委托任何检测到的在位置方面的更改。通过指定距离筛选器，告知位置管理器不要将每个更改都通知你，仅当位置更改超过特定数量时才通知你。设置距离筛选器可以减少应用程序所执行的轮询数量。距离筛选器也是以米为单位进行设置的。指定距离筛选器为1000，是告知位置管理器直到iPhone已经从其以前报告的位置移动至少1000m之后才通知委托。

下面是一个示例：

```
locationManager.distanceFilter = 1000.0f;
```

如果你曾经希望将位置管理器返回到没有筛选器的默认设置，则可以使用常量kCLDistanceFilterNone，如下所示：

```
locationManager.distanceFilter = kCLDistanceFilterNone;
```

14.1.3 启动位置管理器

当你准备好开始轮询位置时，通知位置管理器启动，然后它将离开并做自己的事情，然后在它已经确定当前位置时调用委托方法。在你告知它停止之前，只要它感知到任何超过当前距离筛选器的更改，它就会继续调用你的委托方法。下面是启动位置管理器的方法：

```
[locationManager startUpdatingLocation];
```

14.1.4 更明智地使用位置管理器

如果只需要确定当前位置而不需要连续轮询位置，则当它获取应用程序所需的信息之后，你

应该让位置委托停止位置管理器。如果需要连续轮询，则确保只要可能就停止轮询。请记住，只要从位置管理器获得更新，就会消耗用户的电池。若要告知位置管理器停止向其委托发送更新，请调用`stopUpdatingLocation`，如下所示：

```
[locationManager stopUpdatingLocation];
```

14.2 位置管理器委托

位置管理器委托必须符合`CLLocationManagerDelegate`协议，该协议定义了两种方法，这两种方法都是可选的。当位置管理器已经确定当前位置或者当它检测到位置的更改时将调用其中一个方法。当位置管理器遇到错误时将调用另一个方法。

14.2.1 获取位置更新

当位置管理器希望通知其委托当前位置时，它将调用`locationManager:didUpdateToLocation:fromLocation:`方法。该方法接受3个参数。第一个参数是调用该方法的位置管理器。第二个参数是定义iPhone的当前位置的一个`CLLocation`对象，第3个参数是从上次更新定义之前的位置的一个`CLLocation`对象。第一次调用该方法时，以前的位置对象将为`nil`。

14.2.2 使用 CLLocation 获取纬度和经度

使用`CLLocation`类的实例从位置管理器传递位置信息。该类具有有可能对你的应用程序感兴趣的5个属性。纬度和精度存储在一个名为`coordinate`的属性中。若要以度为单位获取纬度和经度，请执行以下操作：

```
CLLocationDegrees latitude = theLocation.coordinate.latitude;  
CLLocationDegrees longitude = theLocation.coordinate.longitude;
```

`CLLocation`对象还可以告诉你位置管理器在其纬度和经度计算方面的精确程度。`horizontalAccuracy`属性描述以`coordinate`作为其中心的一个圆的半径。`horizontalAccuracy`的值越大，Core Location所确定的位置就越不确定。非常小的半径表示在确定的位置方面具有较高的置信度。

你可以看到`horizontalAccuracy`在Maps应用程序中的图形表示（参见图14-1）。当检测到你的位置时，Maps中显示的蓝色圆使用`horizontalAccuracy`作为它的半径。位置管理器认为你位于该圆的中心。如果不是这样，几乎可以肯定地说你位于蓝色圆之内的某个位置。`horizontalAccuracy`为负值时，表示由于某些原因，你不能依赖`coordinate`的值。

`CLLocation`对象还具有一个名为`altitude`的属性，该属性可以告诉你你在海平面以上或以下多少米：

```
CLLocationDistance altitude = theLocation.altitude.
```

每个`CLLocation`对象都有一个名为`verticalAccuracy`的属性，该属性表示Core Location在其精确的海拔方面的精确程度。海拔的值可能与`verticalAccuracy`的值相差很多米，并且如果

verticalAccuracy值为负值，则Core Location会告诉你它无法确定有效的海拔。

CLLocation对象具有一个时间戳，它告知位置管理器确定位置的时间。

除了这些属性之外，CLLocation还有一个非常有用的实例方法，该方法将允许你确定两个CLLocation对象之间的距离。该方法称为getDistanceFrom:，它的工作方式如下所示：

```
CLLocationDistance distance = [fromLocation➤
getDistanceFrom:toLocation];
```

前面的一行代码将返回两个CLLocation对象即fromLocation和toLocation之间的距离。返回的distance值将是大地计算的结果，该计算忽略了海拔属性，并且假设这两个点处于同一海平面来计算该距离。对于大多数场合来说，大地计算已经足够了，但是如果在计算距离时需要考虑海拔，则必须编写你自己的代码来进行该操作。

14.2.3 错误通知

如果Core Location无法确定你的当前位置，则它会调用另一个名为locationManager:didFailWithError:的代理方法。最可能的错误原因是用户拒绝访问。Location Manager的使用必须由用户进行授权，因此应用程序在第一次确定位置时，会在屏幕上弹出一个警告，询问用户是否确定让当前程序访问你的位置（参见图14-2）。

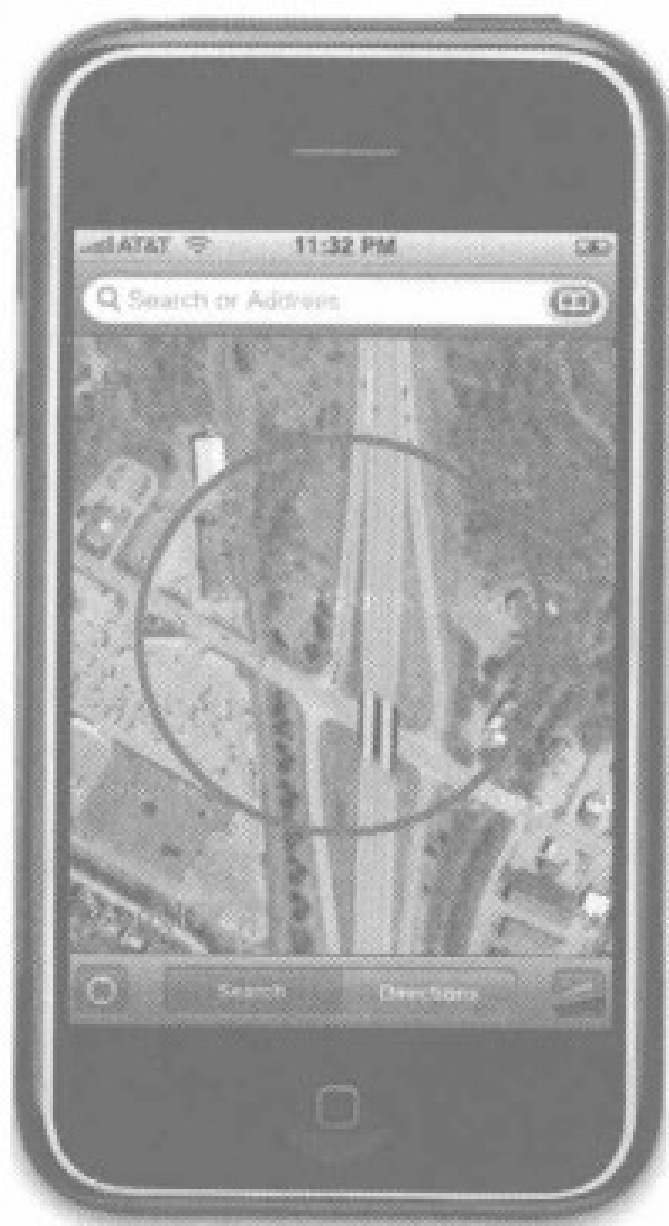


图14-1 Maps应用程序使用Core Location来确定你的当前位置。蓝色圆是水平精度的可视化表示

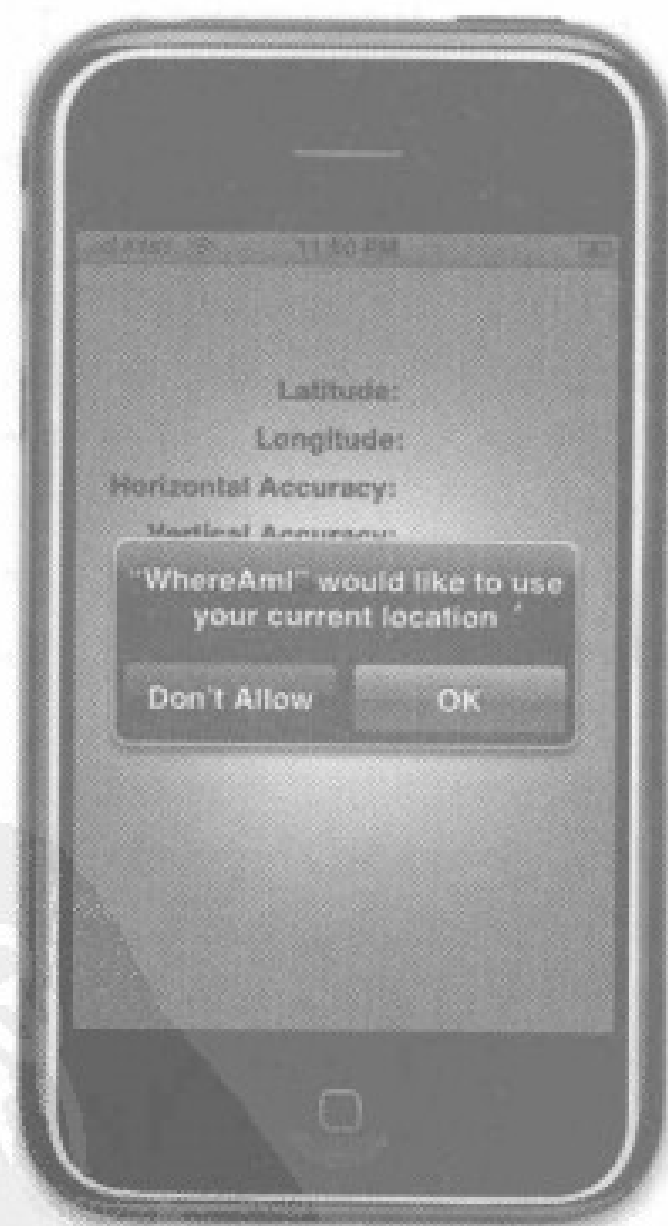


图14-2 Location Manager访问必须经过用户批准

如果用户单击Don't Allow按钮，则Location Manager会使用包含错误代码kCLErrorDenied的locationManager:didFailWithError:通知你的委托。编写此项目时，Location Manager唯一支持的其他错误代码为kCLErrorLocationUnknown，它表示Core Location无法确定位置，但它将不断尝

试。`kCLErrorDenied`错误通常表示,在当前会话的其余时间,应用程序都将无法访问Core Location。另一方面, `kCLErrorLocationUnknown`错误表示问题可能是临时的。

说明 当在仿真器中工作时,将不会提示你对Core Location的访问,并且将使用一个超级机密的算法来确定位置,该算法保留在位于库珀蒂诺的苹果公司总部的存储库中。

14.3 尝试使用 Core Location

让我们构建一个小型应用程序来检测iPhone的当前位置以及该程序运行期间所移动的总路程。你可以看到我们的最终应用程序将类似于图14-3。

在Xcode中,使用基于视图的应用程序模板创建一个新项目,并将该项目命名为WhereAmI。展开Classes和Resources文件夹,并单击WhereAmIViewController.h。进行以下更改,稍后我们将讨论这些更改:

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

@interface WhereAmIViewController :
    UIViewController <CLLocationManagerDelegate> {
    CLLocationManager    *locationManager;

    CLLocation          *startingPoint;

    IBOutlet    UILabel *latitudeLabel;
    IBOutlet    UILabel *longitudeLabel;
    IBOutlet    UILabel *horizontalAccuracyLabel;
    IBOutlet    UILabel *altitudeLabel;
    IBOutlet    UILabel *verticalAccuracyLabel;
    IBOutlet    UILabel *distanceTraveledLabel;
}
@property (retain, nonatomic) CLLocationManager *locationManager;
@property (retain, nonatomic) CLLocation *startingPoint;
@property (retain, nonatomic) UILabel *latitudeLabel;
@property (retain, nonatomic) UILabel *longitudeLabel;
@property (retain, nonatomic) UILabel *horizontalAccuracyLabel;
@property (retain, nonatomic) UILabel *altitudeLabel;
@property (retain, nonatomic) UILabel *verticalAccuracyLabel;
@property (retain, nonatomic) UILabel *distanceTraveled;
@end
```

需要注意的第一件事情是,我们已经包含了Core Location头文件。Core Location不是UIKit的一部分,因此我们需要手动包含头文件。接下来,我们使该类与CLLocationManagerDelegate方法一致,以便我们可以从Location Manager接收位置信息。

之后,我们声明一个CLLocationManager指针,使用该指针来存放我们创建的Core Location实例。我们还声明了一个指向CLLocation的指针,我们将其设置为第一次更新时从位置管理器接

收的位置。这样，如果用户运行我们的程序，并且移动足够远以触发某个更新，我们将能够计算用户移动的距离。

其余距离变量都是输出口，我们将使用它们来更新用户界面上的标签。

双击 WhereAmIViewController.xib 以打开 Interface Builder。使用图14-3作为向导，从库中拖出12个Labels到View窗口中。将其中6个标签放置在屏幕的左侧，将它们设置为右对齐并使用粗体字体。分别为6个粗体的标签设置值 Latitude:、Longitude:、Horizontal Accuracy:、Altitude:、Vertical Accuracy:和Distance Traveled:。其他6个标签应该采用左对齐，并且放在每个粗体标签的旁边。右侧的每个标签应该连接到我们之前在头文件中定义的适当输出口。将所有6个标签连接到输出口之后，依次双击每个标签，并删除其包含的文本。保存并返回到Xcode。

单击WhereAmIViewController.m并进行以下更改：

```
#import "WhereAmIViewController.h"

@implementation WhereAmIViewController
@synthesize locationManager;
@synthesize startingPoint;
@synthesize latitudeLabel;
@synthesize longitudeLabel;
@synthesize horizontalAccuracyLabel;
@synthesize altitudeLabel;
@synthesize verticalAccuracyLabel;
@synthesize distanceTraveled;

#pragma mark -
- (void)viewDidLoad {
    self.locationManager = [[CLLocationManager alloc] init];
    locationManager.delegate = self;
    locationManager.desiredAccuracy = kCLLocationAccuracyBest;
    [locationManager startUpdatingLocation];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
}
```

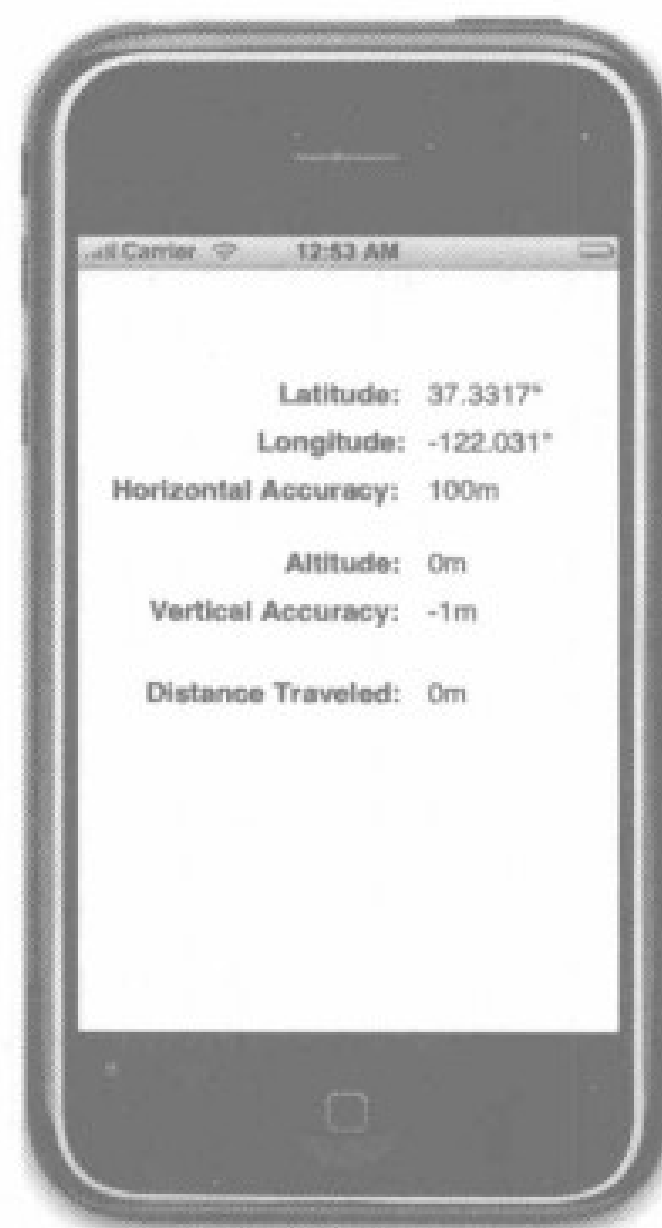


图14-3 实际运行的WhereAmI应用程序。此屏幕截图是在仿真器中获取的。注意垂直精度为负数，这告诉我们它无法确定海拔

```
// Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [locationManager release];
    [startingPoint release];
    [latitudeLabel release];
    [longitudeLabel release];
    [horizontalAccuracyLabel release];
    [altitudeLabel release];
    [verticalAccuracyLabel release];
    [distanceTraveled release];
    [super dealloc];
}

#pragma mark -
#pragma mark CLLocationManagerDelegate Methods
- (void)locationManager:(CLLocationManager *)manager
    didUpdateToLocation:(CLLocation *)newLocation
    fromLocation:(CLLocation *)oldLocation {

    if (startingPoint == nil)
        self.startingPoint = newLocation;

    NSString *latitudeString = [[NSString alloc] initWithFormat:@"%g°",
        newLocation.coordinate.latitude];
    latitudeLabel.text = latitudeString;
    [latitudeString release];

    NSString *longitudeString = [[NSString alloc] initWithFormat:@"%g°",
        newLocation.coordinate.longitude];
    longitudeLabel.text = longitudeString;
    [longitudeString release];

    NSString *horizontalAccuracyString = [[NSString alloc]
        initWithFormat:@"%gm",
        newLocation.horizontalAccuracy];
    horizontalAccuracyLabel.text = horizontalAccuracyString;
    [horizontalAccuracyString release];

    NSString *altitudeString = [[NSString alloc] initWithFormat:@"%gm",
        newLocation.altitude];
    altitudeLabel.text = altitudeString;
    [altitudeString release];

    NSString *verticalAccuracyString = [[NSString alloc]
        initWithFormat:@"%gm",
        newLocation.verticalAccuracy];
    verticalAccuracyLabel.text = verticalAccuracyString;
    [verticalAccuracyString release];
}
```



```

CLLocationDistance distance = [newLocation
    getDistanceFrom:startingPoint];
NSString *distanceString = [[NSString alloc]
    initWithFormat:@"%gm", distance];
distanceTraveledLabel.text = distanceString;
[distanceString release];
}

- (void)locationManager:(CLLocationManager *)manager
    didFailWithError:(NSError *)error {

    NSString *errorType = (error.code == kCLErrorDenied) ?
        @"Access Denied" : @"Unknown Error";
    UIAlertView *alert = [[UIAlertView alloc]
        initWithTitle:@"Error getting Location"
        message:errorType
        delegate:nil
        cancelButtonTitle:@"Okay"
        otherButtonTitles:nil];
    [alert show];
    [alert release];

}

@end

```

在viewDidLoad方法中，我们分配并初始化一个CLLocationManager实例，将我们的控制器类指定为委托，将所需的精度设置为可用的最佳精度，然后让我们的Location Manager实例开始提供位置更新：

```

- (void)viewDidLoad {
    self.locationManager = [[CLLocationManager alloc] init];
    locationManager.delegate = self;
    locationManager.desiredAccuracy = kCLLocationAccuracyBest;
    [locationManager startUpdatingLocation];
}

```

14.3.1 更新位置管理器

由于该类将其自己指定为位置管理器的委托，并且如果我们实现委托方法locationmanager:didUpdateToLocation:fromLocation:，则我们知道该类会进行位置更新，因此让我们看一看该方法的实现。我们在该方法中要做的第一件事情就是检查startingPoint是否为nil。如果是，则该更新是来自Location Manager的第一个更新，我们将当前位置指定给startingPoint属性。

```

if (startingPoint == nil)
    self.startingPoint = newLocation;

```

之后，我们用newLocation参数中传递的CLLocation对象的值来更新前6个标签：

```

NSString *latitudeString = [[NSString alloc] initWithFormat:@"%g°",
    newLocation.coordinate.latitude];

```

```
latitudeLabel.text = latitudeString;
[latitudeString release];

NSString *longitudeString = [[NSString alloc] initWithFormat:@"%g°",
    newLocation.coordinate.longitude];
longitudeLabel.text = longitudeString;
[longitudeString release];

NSString *horizontalAccuracyString = [[NSString alloc]
    initWithFormat:@"%gm",
    newLocation.horizontalAccuracy];
horizontalAccuracyLabel.text = horizontalAccuracyString;
[horizontalAccuracyString release];

NSString *altitudeString = [[NSString alloc] initWithFormat:@"%gm",
    newLocation.altitude];
altitudeLabel.text = altitudeString;
[altitudeString release];

NSString *verticalAccuracyString = [[NSString alloc]
    initWithFormat:@"%gm",
    newLocation.verticalAccuracy];
verticalAccuracyLabel.text = verticalAccuracyString;
[verticalAccuracyString release];
```

提示 你可以通过按 $\text{⌘}8$ 来键入度的符号(°)。

14.3.2 确定移动距离

最后，我们确定当前位置与存储在`startingPoint`中的位置之间的距离，并显示该距离。当运行该应用程序时，如果用户移动得足够远，以至于Location Manager能够检测到更改，则应用程序下次启动时会使用与用户原来位置之间的距离不断更新Distance Traveled:字段。

```
CLLocationDistance distance = [newLocation
    getDistanceFrom:startingPoint];
NSString *distanceString = [[NSString alloc]
    initWithFormat:@"%gm", distance];
distanceTraveledLabel.text = distanceString;
[distanceString release];
```

现在你已经实现了定位功能。Core Location非常简单而且易于使用。在编译该程序之前，你必须向项目中添加CoreLocation.framework。当你添加了CoreGraphics.framework之后，执行的操作与在前面第5章执行的操作完全相同，只是当你导航到适当的Frameworks文件夹时，选择的是CoreLocation.framework，而不是CoreGraphics.framework。编译并运行该应用程序，并尝试使用它。如果你能够在iPhone上运行该应用程序，则尝试运行该应用程序，并观察当你驾车时距离的变化情况。实际上，让他人来驾车会比较好！

14.4 小结

现在，你已经了解了Core Location的方方面面。尽管底层的技术非常复杂，但是苹果公司提供了一个简单的界面，将大部分复杂性隐藏在其中，并且极大地简化了在应用程序中添加定位功能的操作，因此你可以方便地确定用户移动时的位置。

准备好了吗？请继续阅读下一章，了解如何使用iPhone的内置加速计。



内置加速计是iPhone和iPod Touch中最酷的特性之一，iPhone可以通过这个小设备知道用户握持手机的方式，以及用户是否移动了手机。iPhone OS使用加速计处理自动旋转，并且许多游戏都使用它作为控制机制。它还可以用于检测摇动和其他突发的运动。

15.1 加速计物理学

通过感知特定方向的惯性力总量，加速计可以测量出加速度和重力。iPhone内的加速计是一个三轴加速计，这意味着它能够检测到三维空间中的运动或重力引力。因此，加速计不但可以指示握持电话的方式（如自动旋转功能），而且如果电话放在桌子上的话，还可以指示电话的正面朝下还是朝上。

加速计可以测量g引力（“g”代表重力），因此加速计返回值为1.0时，表示在特定方向上感知到1g。如果是静止握持iPhone而没有任何运动，那么地球引力对其施加的力大约为1g。如果是纵向竖直地握持iPhone，那么iPhone会检测并报告在其y轴上施加的力大约为1g。如果是以一定角度握持iPhone，那么1g的力会分布到不同的轴上，这取决于握持iPhone的方式。在以45度角握持时，1g的力会均匀地分解到两个轴上。

如果检测到的加速计值远大于1g，那么即可以判断这是突然运动。正常使用时，加速计在任一轴上都不会检测到远大于1g的值。如果摇动、坠落或投掷iPhone，那么加速计便会在一个或多个轴上检测到很大的力。请不要仅仅为了测试这一理论而坠落或投掷自己的iPhone。

图15-1展示了iPhone加速计所使用的三轴结构。需要注意的是，加速计对y坐标轴使用了更标准的惯例，即y轴伸长表示向上的力，这与Quartz 2D的坐标系相反。如果加速计使用Quartz 2D作为控制机制，那么必须要转换y坐标轴。使用OpenGL ES时（使用加速计控制动作时通常会用到），则不需要转换。

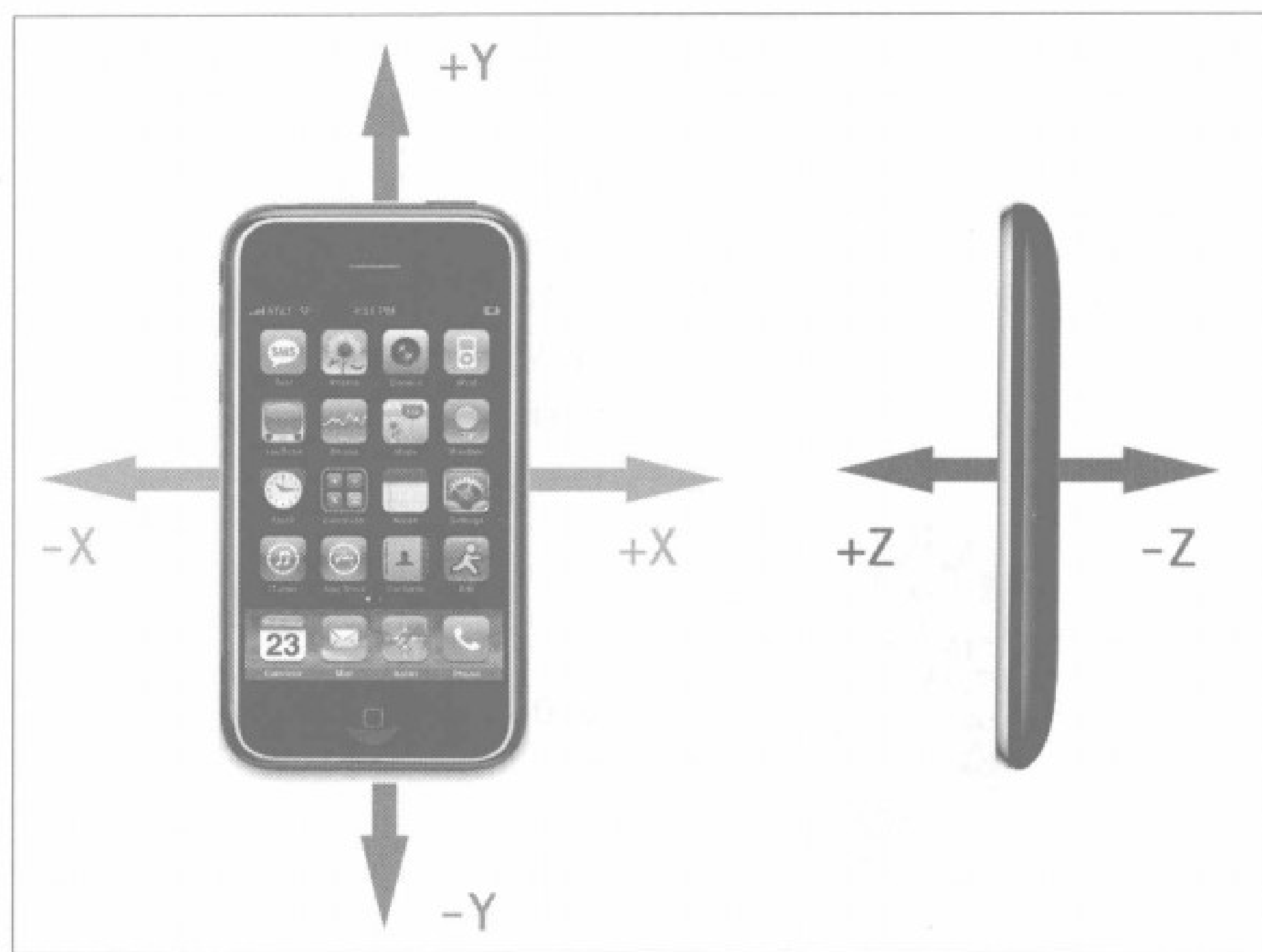


图15-1 三维方向上iPhone加速计的轴

15.2 访问加速计

UIAccelerometer类是单独存在的。要获取对此类的引用，请调用sharedAccelerometer方法，如下所示：

```
UIAccelerometer *accelerometer = [UIAccelerometer sharedAccelerometer];
```

从加速计获取信息与从Core Location获取信息相似。创建一个符合UIAccelerometerDelegate协议的类，执行可以获取加速计信息的方法，并指定此类的一个实例作为加速计的委托。

在分配委托时，需要以秒指定更新间隔。iPhone的加速计支持最高以每秒100次的频率进行轮询，但无法保证真正能够达到这么多次更新，或者可以精确均匀分隔这些更新。要分配委托或指定轮询间隔为每秒60次，可以这样做：

```
accelerometer.delegate = self;  
accelerometer.updateInterval = 1.0f/60.0f;
```

完成之后，剩余的事情是实现加速计用于更新委托的方法，`accelerometer:didAccelerate:`。此方法需要两个变量。第一个变量是对sharedUIAccelerometer实例的引用。第二个变量包含了来自加速计的真实数据，嵌入在类UIAcceleration的一个对象中。在查看委托方法之前，首先讨论用于把信息传递给委托的UIAcceleration对象。

15.2.1 UIAcceleration

如前所述，iPhone加速计可以检测3个轴上的加速度，并且对使用UIAcceleration类实例的委

托提供此信息。每个UIAcceleration实例都有x、y和z属性，分别有一个带符号的浮点值。值0表示加速计在此轴上没有检测到任何运动。正值或负值表示一个方向上的力。例如，y的负值表示感受到了向下的力，这可能表示电话是纵向竖直握持的。y的正值表示在向上的方向施加了某些力，这可能意味着电话是倒置的或者电话正在向下运动。

请在头脑中牢记图15-1的示意图，并查看加速计结果。注意，在现实生活中，几乎不可能获得如此精确的值，因为加速计非常敏感，可以感知非常微小的运动，而我们通常只能感知立体空间3个轴上的某一个微小受力。这是现实世界中的物理，而不是高中物理实验室。

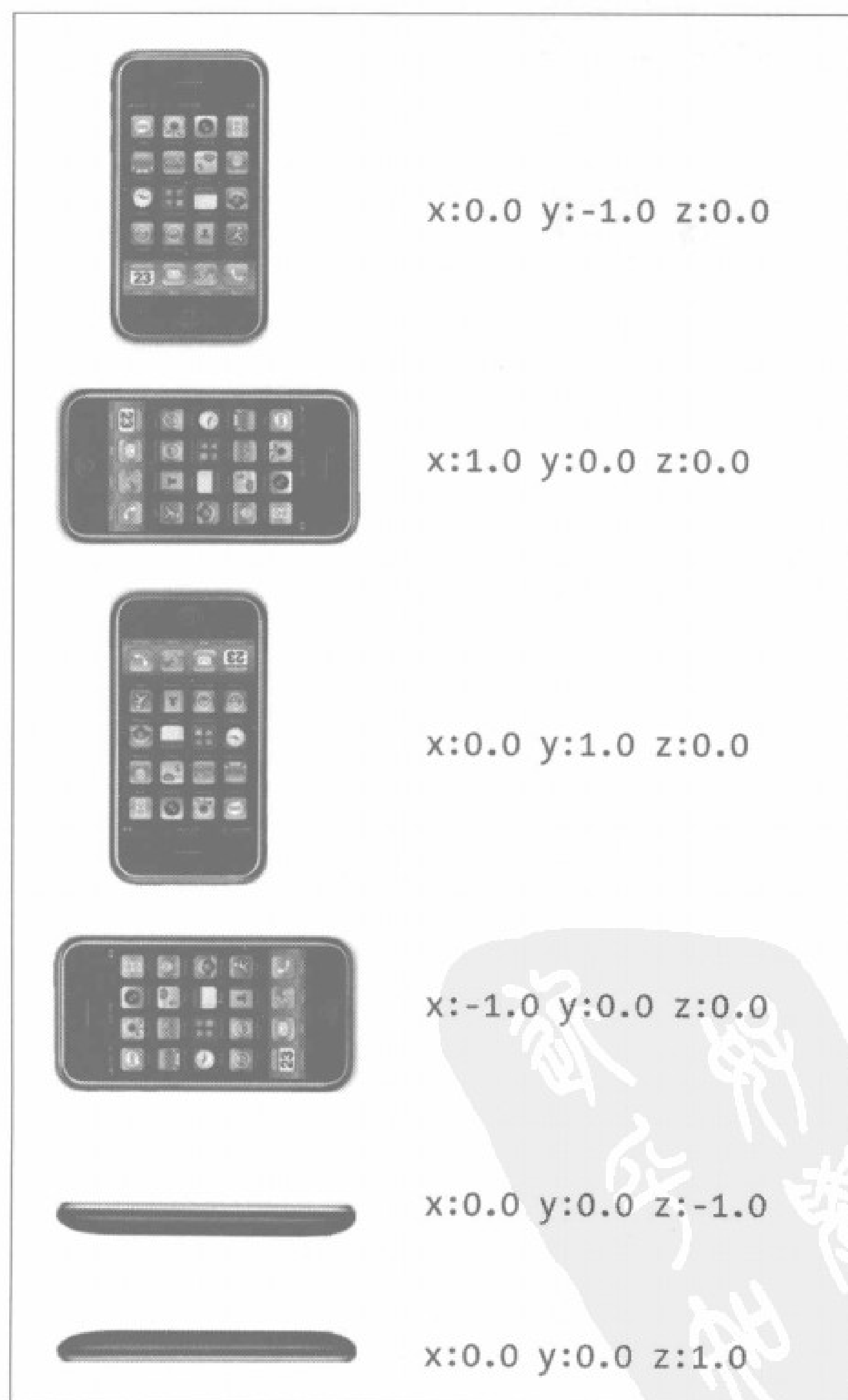


图15-2 设备不同朝向的理想化加速度值

15.2.2 实现 accelerometer:didAccelerate:方法

为了接收加速计信息，指定作为加速计委托的类需要实现accelerometer:didAccelerate:方法。如果想在UILabel中显示加速度值，那么需要这样实现此方法：

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
didAccelerate:(UIAcceleration *)acceleration {

    NSString *newText = [[NSString alloc]
        initWithFormat:@"Max: x: %g\ty:%g\tz:%g", acceleration.x,
        acceleration.y, acceleration.z];
    label.text = newText;
    [newText release];
}
```

此方法可以在每次调用时更改界面上的标签。调用此方法的频率取决于以前指定的updateInterval值。

1. 检测摇动

在应用程序中，加速计通常用于检测摇动。与手势相似，摇动可以作为应用程序输入的一种形式。例如，对于绘图程序GLPaint，iPhone的示例代码项目之一，用户可以通过摇动iPhone擦除绘图，就像Etch-a-Sketch一样。摇动检测功能相对来说是微不足道的，只需要检查某个轴上的绝对值是否大于设置阈值。在正常使用期间，3个轴之一所注册的值通常在1.3g左右，若要远高于此值，则需要刻意施加力量。加速计好像不能注册高于2.3g左右的值（至少在第一代iPhone上如此），因此，请勿将阈值设置得高于此值。

要检测摇动，请检查绝对值是否大于1.5（表示轻微摇动）和2.0（表示剧烈摇动），代码如下：

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
didAccelerate:(UIAcceleration *)acceleration {

    if (fabsf(acceleration.x) > 2.0
        || fabsf(acceleration.y) > 2.0
        || fabsf(acceleration.z) > 2.0) {
        // Do something here...
    }
}
```

上述方法可以检测到任一轴上任何超过2g力的运动。通过要求用户前后摇动一定次数才能注册为一次摇动，我们可以执行更复杂的摇动检测，代码如下：

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
didAccelerate:(UIAcceleration *)acceleration {
    static NSInteger shakeCount = 0;
    static NSDate *shakeStart;

    NSDate *now = [[NSDate alloc] init];
    NSDate *checkDate = [[NSDate alloc] initWithTimeInterval:1.5f
        sinceDate:shakeStart];
```



```

    if ([now compare:checkDate] ==
        NSOrderedDescending || shakeStart == nil) {
        shakeCount = 0;
        [shakeStart release];
        shakeStart = [[NSDate alloc] init];
    }
    [now release];
    [checkDate release];

    if (fabsf(acceleration.x) > 2.0
        || fabsf(acceleration.y) > 2.0
        || fabsf(acceleration.z) > 2.0) {
        shakeCount++;
        if (shakeCount > 4) {
            // Do something
            shakeCount = 0;
            [shakeStart release];
            shakeStart = [[NSDate alloc] init];
        }
    }
}

```

此方法可以查明加速计报告的值大于2的次数，如果在一秒半的时间内发生了4次，则注册为一次摇动。

2. 加速计用作方向控制器

或许加速计在第三方应用程序中最常见的用法是作为游戏控制器。在游戏中不是使用按钮控制字符或对象的移动，而是使用加速计。例如，在赛车游戏中，像转动方向盘那样转动iPhone也许就可以驾驶汽车，而向前倾斜表示加速，向后倾斜表示刹车。

具体如何将加速计用作控制器，这很大程度上取决于游戏的特定机制。在很简单的情况下，可能只需获取一个轴的值，乘以某个数，然后添加到所控制对象的坐标系中。在复杂的游戏中，因为所建立的物理模型更加真实，所以必须根据加速计返回的值调整所控制对象的速度。

使用加速计作为控制器的一个棘手问题是，委托方法并不能保证以指定的间隔回调。如果告诉加速计每秒钟更新60次委托类，所能确定的仅仅是它每秒钟更新的次数绝不会多于60次。因为不能保证每秒钟得到60次均匀分隔的更新，所以如果所做的动画是基于来自加速计的输入，那么必须要弄清楚委托方法调用之间的时间。

本章稍后将创建一个使用加速计作为输入的程序，但是首先，需要将电话击碎。

说明 本章中的应用程序在仿真器上不起作用，因为仿真器中没有加速计。

15.3 摇动与击碎

好吧，并不是真的要电话击碎，我们只是要编写一个应用程序，它在检测到摇动之后会使

电话看起来和听起来好像它因为摇动而破碎一样。启动此应用程序后，程序会显示一张图片，它看起来像是iPhone的首页（如图15-3所示）。

以足够大的力气摇动电话时，就可以听到可怜的电话发出一种声音，任何人都不想听到这样的声音从消费者电子设备中发出来。此外，屏幕看起来像图15-4所示。我们怎么忍心做这样的坏事呢？

不要担心。只需触摸屏幕即可以将iPhone重置到其初始状态。

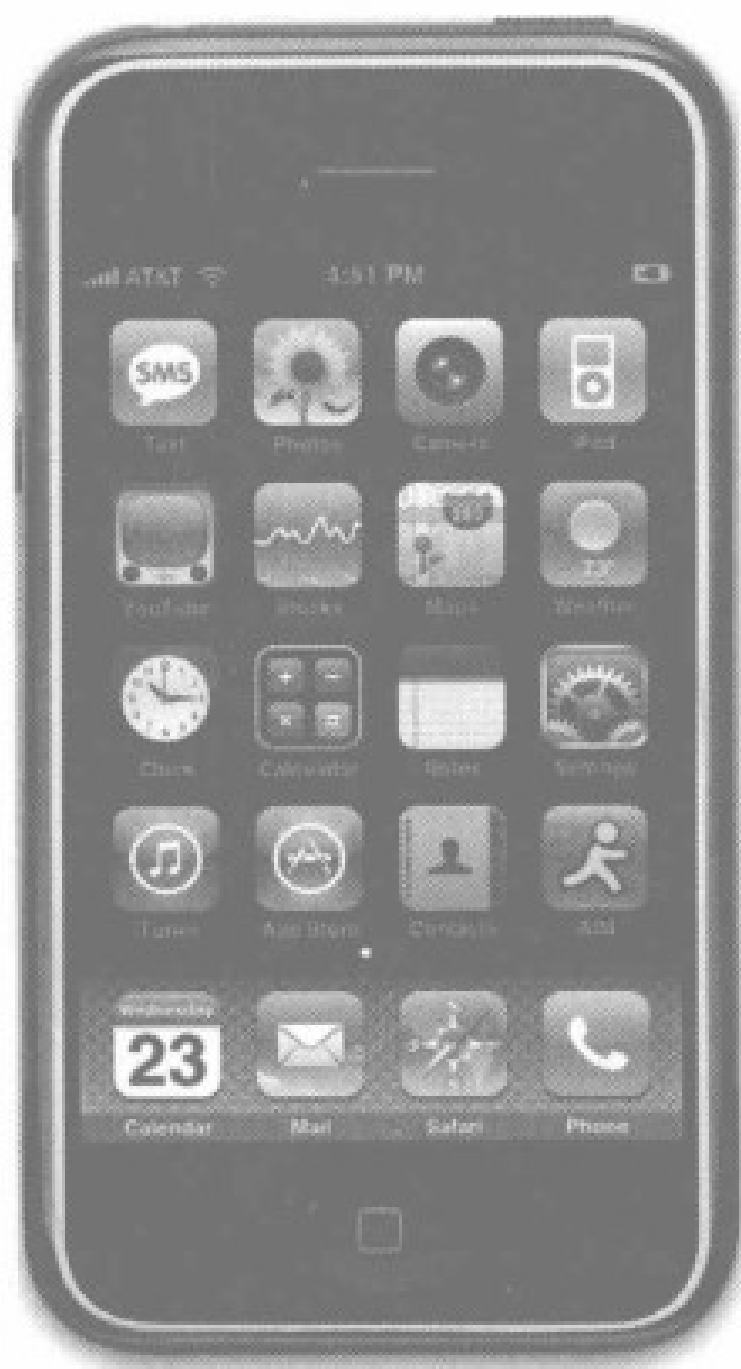


图15-3 ShakeAndBreak应用程序看起来平淡无奇

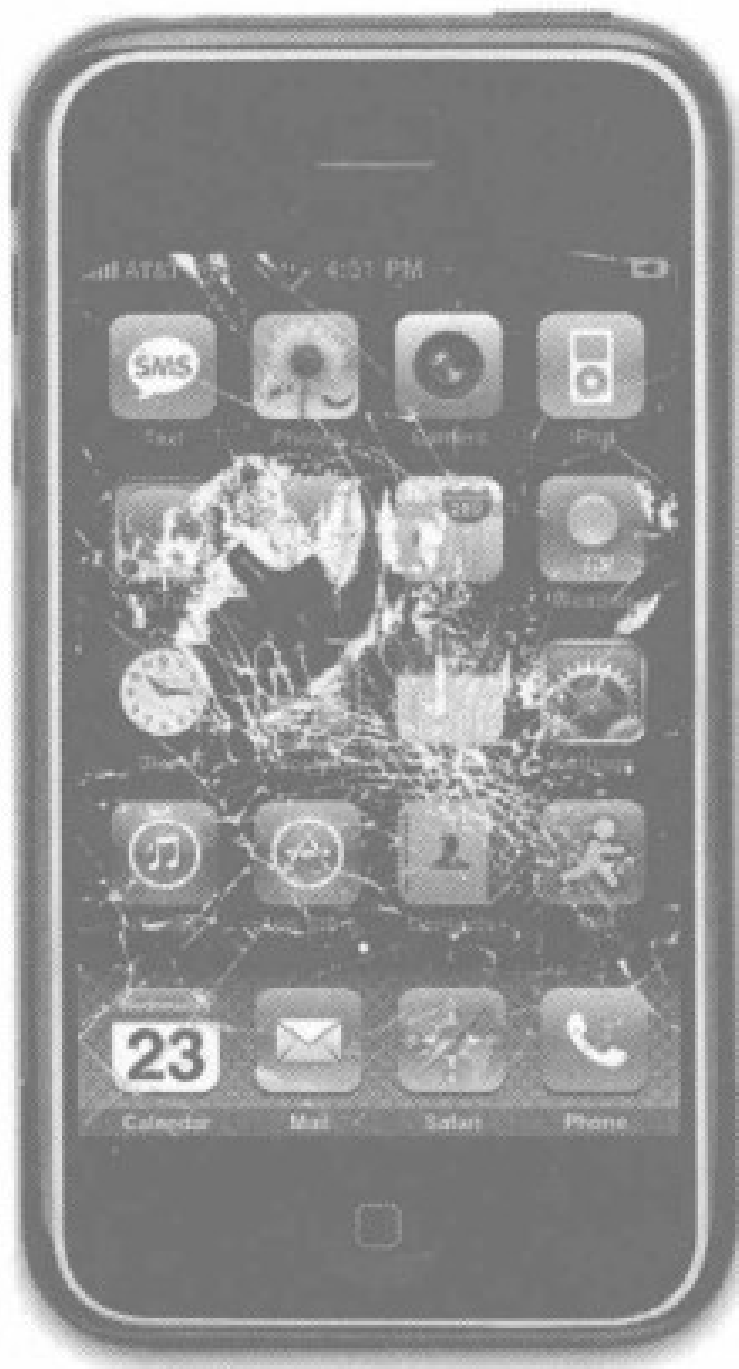


图15-4 但如果摇动太猛烈的话——噢，不！

15.3.1 用于击碎的代码

在Xcode中使用基于视图的应用程序模板新建一个项目。将新建项目命名为ShakeAndBreak。在项目存档文件的15 ShakeAndBreak文件夹中，我们已经为此应用程序提供了两张图像和一个声音文件，所以请将home.png、homebroken.png和glass.wav文件拖到项目的Resources文件夹中。文件夹中还包含一个icon.png文件。也将此文件添加到Resources文件夹中。

然后，展开Resources文件夹并单击info.plist。我们需要在属性列表中添加一个条目，告诉应用程序不要使用状态栏，所以请单击Information Property List行，并单击出现在本行末端的按钮，添加一个新的子值。将新行的Key改为UIStatusBarHidden。然后在刚才添加的行中按下Control键并单击（或者右击，如果有双键鼠标的話）空Value列。此时应该出现一个上下文菜单（如图15-5所示）。在此菜单中，选择Value Type子菜单并选择Boolean。此行应该变为一个复选框。单击选中此复选框。最后，在Icon file键旁的Value列中键入icon.png。

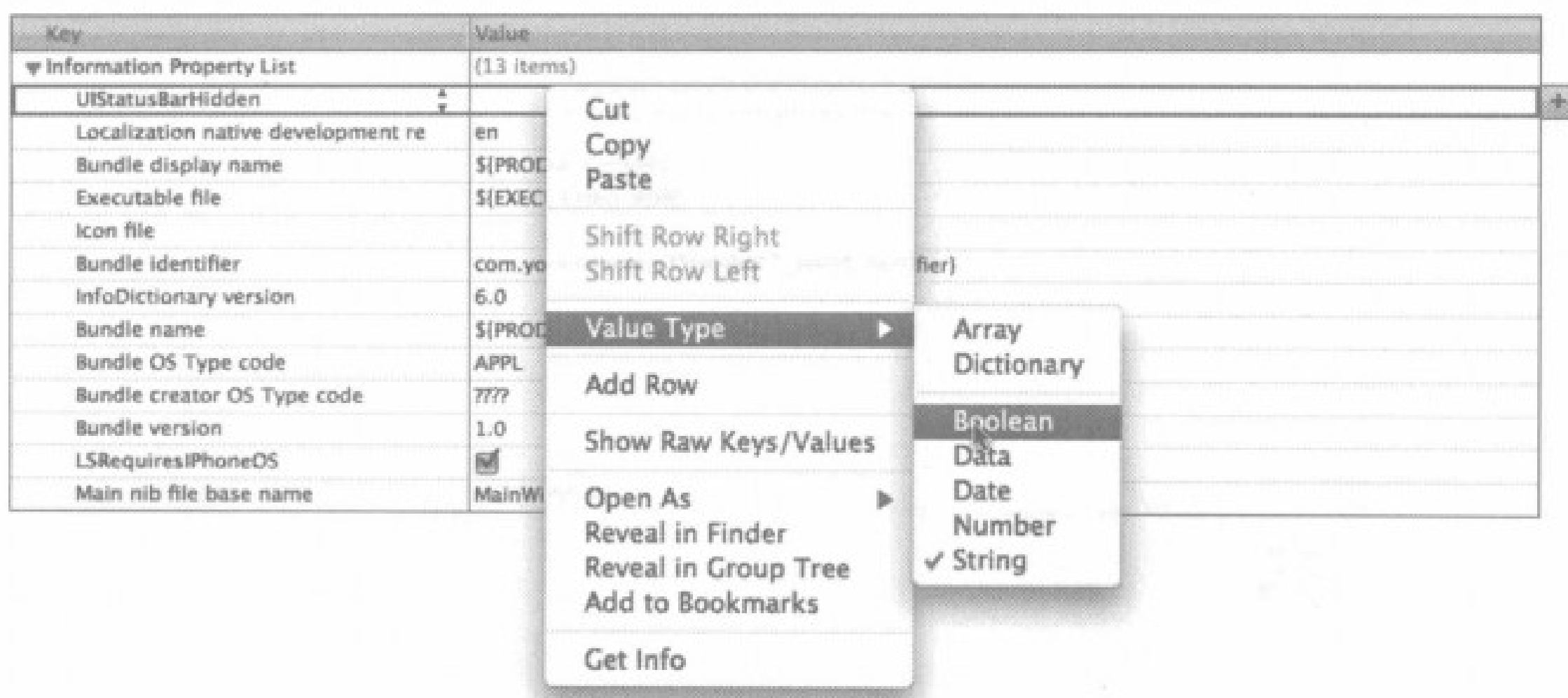


图15-5 更改UIStatusBarHidden的value Type

然后，展开Classes文件夹。我们现在要做的就是创建一个指向图像视图的输出口，以便更改显示图像。此外，还需要一对UIImage实例（用于保存两张图片）、一个声音ID（用于引用声音），以及一个Boolean（用于查明屏幕是否需要重置）。单击ShakeAndBreakViewController.h，并添加以下代码：

```
#define kAccelerationThreshold      2.2
#define kUpdateInterval            (1.0f/10.0f)

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

@interface ShakeAndBreakViewController :
    UIViewController <UIAccelerometerDelegate> {
    IBOutlet UIImageView *imageView;

    BOOL brokenScreenShowing;
    SystemSoundID soundID;
    UIImage *fixed;
    UIImage *broken;
}

@property (nonatomic, retain) UIImageView *imageView;
@property (nonatomic, retain) UIImage *fixed;
@property (nonatomic, retain) UIImage *broken;
@end
```

除了实例变量和属性之外，还应注意的是必须让类符合UIAccelerometerDelegate协议，并定义两个常量，一个用于更新频率，另一个用于定义加速计检测到多少g的力时才能限定为一次摇动。我们已经将更新频率定义到相当低的频率，即一秒钟10次，这已足够检测到一次摇动。通常来说，轮询时会使用能满足要求的最低频率。在将加速计用作控制器时，需要以相当快的速率轮询，通常达到每秒30到60次更新。

保存头文件，并双击ShakeAndBreakViewController.xib，在Interface Builder中打开此文件。单

击View图标，并按⌘3打开大小检查器。将视图的高度从460改为480，使之占据因为丢弃了状态栏而出现的额外屏幕空间。将Image View从库中拖到标签为View的窗口上。图像视图应该自动调整大小以占满整个窗口，所以只需将它放置到窗口中即可。

按下Control键并将File's Owner图标拖到图像视图，并选择imageView输出口。然后保存并关闭nib文件，返回到Xcode中。返回之后，单击ShakeAndBreakController.m文件，并做如下更改：

```
#import "ShakeAndBreakViewController.h"

@implementation ShakeAndBreakViewController
@synthesize imageView;
@synthesize fixed;
@synthesize broken;
- (void) viewDidLoad {
    UIAccelerometer *accel = [UIAccelerometer sharedAccelerometer];
    accel.delegate = self;
    accel.updateInterval = kUpdateInterval;

    NSString *path = [[NSBundle mainBundle] pathForResource:@"glass"
        ofType:@"wav"];
    AudioServicesCreateSystemSoundID((CFURLRef)[NSURL
        fileURLWithPath:path], &soundID);

    self.fixed = [UIImage imageNamed:@"home.png"];
    self.broken = [UIImage imageNamed:@"homebroken.png"];

    imageView.image = fixed;
    brokenScreenShowing = NO;
}
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}
- (void)dealloc {
    [imageView release];
    [fixed release];
    [broken release];
    [super dealloc];
}
#pragma mark -
- (void)accelerometer:(UIAccelerometer *)accelerometer
```

```

didAccelerate:(UIAcceleration *)acceleration {
    if (! brokenScreenShowing) {
        if (acceleration.x > kAccelerationThreshold
            || acceleration.y > kAccelerationThreshold
            || acceleration.z > kAccelerationThreshold) {
            imageView.image = broken;
            AudioServicesPlaySystemSound (soundID);
            brokenScreenShowing = YES;
        }
    }
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    imageView.image = fixed;
    brokenScreenShowing = NO;
}
@end

```

所实现的第一个方法是viewDidLoad，在此可以得到对共享加速计实例的引用，将self设置为加速计的委托，然后使用前面定义的常量设置更新频率：

```

UIAccelerometer *accel = [UIAccelerometer sharedAccelerometer];
accel.delegate = self;
accel.updateInterval = kUpdateInterval;

```

15.3.2 加载模拟文件

然后，将玻璃声音文件加载到内存中，并将分配的标识符保存到soundID实例变量中。

```

NSString *path = [[NSBundle mainBundle] pathForResource:@"glass"
    ofType:@"wav"];
AudioServicesCreateSystemSoundID((CFURLRef)[NSURL
    fileURLWithPath:path], &soundID);

```

然后，将两张图像加载到内存中：

```

self.fixed = [UIImage imageNamed:@"home.png"];
self.broken = [UIImage imageNamed:@"homebroken.png"];

```

最后，设置imageView显示未破坏的屏幕快照，并将brokenScreenShowing设置为NO以指示屏幕当前不需要重置：

```

imageView.image = fixed;
brokenScreenShowing = NO;

```

下一个新方法是加速计委托方法。检测其中的brokenScreenShowing。如果其值为NO，则表示屏幕已经显示了受破坏的图像，所以此时不需要做任何事。

```

if (! brokenScreenShowing) {

```

另外，检测所有3个轴，查看是否有轴超过了前面定义的加速度阈值。如果有任何一个轴超过了阈值，则将图像视图设置为显示受破坏的图像，播放声音，并将brokenScreenShowing设置为YES，以便在用户重置屏幕之前不重复这样做。

```

    if (acceleration.x > kAccelerationThreshold || acceleration.y >
        kAccelerationThreshold || acceleration.z >
        kAccelerationThreshold) {
        imageView.image = broken;
        AudioServicesPlaySystemSound (soundID);
        brokenScreenShowing = YES;
    }
}

```

15.3.3 完好如初——复原触摸

对于最后一个方法，你已经非常熟悉的了。在触摸屏幕时会调用此方法。在此方法中只需要将图像设置回未破坏的屏幕，并将brokenScreenShowing设置为NO：

```

imageView.image = fixed;
brokenScreenShowing = NO;

```

最后，添加AudioToolbox.framework以便播放声音文件。AudioToolbox.framework处在与CoreGraphics.framework相同的位置。如果忘记了操作步骤，请按照第5章中添加框架的步骤进行。

编译并运行此应用程序，然后进行测试。现在就可以体验应用程序了。完成之后返回，便可以知道如何将加速计用作游戏或其他程序中的控制器。

15.4 滚弹珠程序

我们的下一个小游戏，是通过倾斜电话在iPhone的屏幕上移动弹珠。这是使用加速计接收输入的一个非常简单的例子。此处，我们将使用Quartz 2D来处理动画。在处理游戏或其他需要平滑动画的程序时，通常的规则是使用OpenGL。在此应用程序中使用Quartz 2D，是因为它比较简单，并且可以减少与使用加速计无关的代码。也许动画效果不如使用OpenGL那样平滑，但却可以大大减少工作量。

在此应用程序中，弹珠会随着倾斜iPhone而来回滚动，就像是在桌面上一样（如图15-6所示）。将它向左倾斜，小球就会向左滚动。倾斜得更厉害，小球就会滚动得更快。返回倾斜，则小球会慢下来并开始向另一个方向滚动。

在Xcode中，使用基于视图的模式新建一个项目，将它命名为Ball。展开Classes和Resource文件夹，以便可以看到正在处理的文件。在项目存档文件的15 Ball文件夹中，可以找到一个名为ball.png的图像。将它拖到项目的Resource文件夹中。

然后单击Classes文件夹，并从File菜单中选择New File...。从Cocoa Touch目录中选择UIView subclass，然后将新文件命名为BallView.m，并确保创建了头文件。



图15-6 滚弹珠应用程序只有这样的功能——在屏幕上滚动弹珠

双击BallViewController.xib，在Interface Builder中打开文件。单击View图标，并使用身份检查器将视图的类由UIView改为BallView。然后切换到属性检查器，将视图的背景颜色更改为黑色。之后，按下Control键并从File's Owner图标拖至Ball View图标，然后选择view输出口，重新建立控制器与视图之间的链接。最后保存并关闭nib文件，返回至Xcode。

15.4.1 实现 Ball View 控制器

单击BallViewViewController.h。在此所要做的只是让类符合UIAccelerometerDelegate协议，所以要进行如下更改：

```
#define kUpdateInterval    (1.0f/60.0f)
#import <UIKit/UIKit.h>

@interface BallViewController :
    UIViewController <UIAccelerometerDelegate> {

}
```

```
@end
```

接下来，转到BallViewViewController.m，进行如下更改：

```
#import "BallViewController.h"
#import "BallView.h"

@implementation BallViewController

- (void)viewDidLoad {
    UIAccelerometer *accelerometer = [UIAccelerometer sharedAccelerometer];
    accelerometer.delegate = self;
    accelerometer.updateInterval = kUpdateInterval;
    [super viewDidLoad];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [super dealloc];
}
```



```
#pragma mark -
- (void)accelerometer:(UIAccelerometer *)accelerometer
  didAccelerate:(UIAcceleration *)acceleration {

    [(BallView *)self.view setAcceleration:acceleration];
    [(BallView *)self.view draw];
}
@end
```

此处的viewDidLoad方法与前面的相似。主要区别是这里声明的更新间隔更短，每秒60次。在accelerometer:didAccelerate:方法中，将加速对象传递到视图中并调用方法draw，此方法根据加速度和距离上次更新的时间量来更新视图中小球的位置。

15.4.2 编写 Ball view

既然我们大部分工作是在BallView类中处理，那就最好把它写出来。单击BallView.h并做出如下更改：

```
#define kVelocityMultiplier    500
#import <UIKit/UIKit.h>

@interface BallView : UIView {
    UIImage *image;

    CGPoint    currentPoint;
    CGPoint    previousPoint;

    UIAcceleration *acceleration;
    CGFloat    ballXVelocity;
    CGFloat    ballYVelocity;
}
@property (nonatomic, retain) UIImage *image;
@property CGPoint currentPoint;
@property CGPoint previousPoint;
@property (nonatomic, retain) UIAcceleration *acceleration;
@property CGFloat ballXVelocity;
@property CGFloat ballYVelocity;
- (void)draw;
@end
```

现在查看实例变量并讨论它们各自的用法。第一个实例变量是UIImage，它指向我们在屏幕上滚动的弹珠：

```
UIImage *image;
```

然后，查明两个CGPoint变量。currentPoint变量用于保持小球当前的位置。同样要查明的是绘制弹珠的最后一个点，以便建立一个更新矩形，此矩形包围住小球的新旧位置，在新位置进行绘制，并擦除旧位置。

```
CGPoint    currentPoint;  
CGPoint    previousPoint;
```

然后是指向加速对象的指针，通过它可以从控制器获得加速计信息：

```
UIAcceleration *acceleration;
```

还有两个变量用于在两个维度上跟踪小球的当前速度。虽然这并不是很复杂的模拟，但我们仍想让小球滚动的方式与真正的小球相似，所以在此使用方程 $velocity = velocity + acceleration$ 计算速度。我们从加速计获得加速度值并跟踪这些变量的速度。

```
CGFloat ballXVelocity;  
CGFloat ballYVelocity;
```

现在切换到BallView.m，并编写代码在屏幕上绘制并移动小球：

```
#import "BallView.h"
```

```
@implementation BallView  
@synthesize image;  
@synthesize currentPoint;  
@synthesize previousPoint;  
@synthesize acceleration;  
@synthesize ballXVelocity;  
@synthesize ballYVelocity;
```

```
- (id)initWithCoder:(NSCoder *)coder {  
    if (self = [super initWithCoder:coder]) {  
        self.image = [UIImage imageNamed:@"ball.png"];  
        self.currentPoint = CGPointMake((self.bounds.size.width / 2.0f) +  
                                         (image.size.width / 2.0f),  
                                         (self.bounds.size.height / 2.0f) + (image.size.height / 2.0f));  
  
        ballXVelocity = 0.0f;  
        ballYVelocity = 0.0f;  
    }  
    return self;  
}  
  
- (id)initWithFrame:(CGRect)frame {  
    if (self = [super initWithFrame:frame]) {  
        // Initialization code  
    }  
    return self;  
}  
  
- (void)drawRect:(CGRect)rect {  
    [image drawAtPoint:currentPoint];  
}
```

```

- (CGPoint)currentPoint {
    return currentPoint;
}
- (void)setCurrentPoint:(CGPoint)newPoint {
    previousPoint = currentPoint;
    currentPoint = newPoint;

    if (currentPoint.x < 0) {
        currentPoint.x = 0;
        ballXVelocity = 0;
    }
    if (currentPoint.y < 0){
        currentPoint.y = 0;
        ballYVelocity = 0;
    }
    if (currentPoint.x > self.bounds.size.width - image.size.width) {
        currentPoint.x = self.bounds.size.width - image.size.width;
        ballXVelocity = 0;
    }
    if (currentPoint.y > self.bounds.size.height - image.size.height) {
        currentPoint.y = self.bounds.size.height - image.size.height;
        ballYVelocity = 0;
    }
    CGRect currentImageRect = CGRectMake(currentPoint.x, currentPoint.y,
        currentPoint.x + image.size.width,
        currentPoint.y + image.size.height);
    CGRect previousImageRect = CGRectMake(previousPoint.x, previousPoint.y,
        previousPoint.x + image.size.width,
        currentPoint.y + image.size.width);
    [self setNeedsDisplayInRect:CGRectUnion(currentImageRect,
        previousImageRect)];
}

- (void)draw {
    static NSDate *lastDrawTime;

    if (lastDrawTime != nil) {
        NSTimeInterval secondsSinceLastDraw =
            -([lastDrawTime timeIntervalSinceNow]);

        ballYVelocity = ballYVelocity + -(acceleration.y *
            secondsSinceLastDraw);
        ballXVelocity = ballXVelocity + acceleration.x *
            secondsSinceLastDraw;

        CGFloat xAcceleration = secondsSinceLastDraw * ballXVelocity * 500;
        CGFloat yAcceleration = secondsSinceLastDraw * ballYVelocity * 500;
    }
}

```

```

        self.currentPoint = CGPointMake(self.currentPoint.x +
xAcceleration,
        self.currentPoint.y + yAcceleration);
    }
    // Update last time with current time
    [lastDrawTime release];
    lastDrawTime = [[NSDate alloc] init];
}
- (void)dealloc {
    [image release];
    [acceleration release];
    [super dealloc];
}

@end

```

首先要注意的是，需要将其中一个属性声明为@synthesize，因为我们已经在代码中为此属性执行了修改方法。就是这样。@synthesize指令不会改写访问方法或已编写的修改方法；它只会填充空白并提供你所不能提供的方法。

15.4.3 计算小球运动

因为是手动处理的currentPoint属性，所以当currentPoint更改时，需要做一点清理工作，比如确保小球不会滚出屏幕。稍后我们会研究此方法。现在，让我们先看看此类中的第一个方法initWithCoder:。记得在从nib文件载入视图时，始终不能调用类的init或initWithFrame:方法。nib文件包含了归档的对象，所以任何从nib文件中载入的实例都会使用initWithCoder:方法进行初始化。如果需要进行额外的初始化，就要使用此方法。

在此视图中，我们需要进行额外的初始化，所以我们覆盖了initWithCoder:。首先，载入ball.png图像。然后，计算视图的中心并将其设置为小球的起始点，并将两个轴上的速度设置为0。

```

self.image = [UIImage imageNamed:@"ball.png"];
self.currentPoint = CGPointMake((self.bounds.size.width / 2.0f) +
    (image.size.width / 2.0f), (self.bounds.size.height / 2.0f) +
    (image.size.height / 2.0f));

ballXVelocity = 0.0f;
ballYVelocity = 0.0f;

```

drawRect:方法极其简单。我们只需在currentPoint中存储的位置处绘制在initWithCoder:中载入的图像即可。currentPoint访问方法是一种标准的访问方法。然而，setCurrentPoint:修改方法却不同。

在setCurrentPoint:中首先要做的是将旧的currentPoint值存储在previousPoint中，并将新值赋给currentPoint:

```

previousPoint = currentPoint;
currentPoint = newPoint;

```

下面要做的是边界检查。如果小球的x或y位置小于0，或大于屏幕的宽度或高度（计算图像的宽度和高度），则停止在此方向上加速。

```
if (currentPoint.x < 0) {
    currentPoint.x = 0;
    ballXVelocity = 0;
}
if (currentPoint.y < 0){
    currentPoint.y = 0;
    ballYVelocity = 0;
}
if (currentPoint.x > self.bounds.size.width - image.size.width) {
    currentPoint.x = self.bounds.size.width - image.size.width;
    ballXVelocity = 0;
}
if (currentPoint.y > self.bounds.size.height - image.size.height) {
    currentPoint.y = self.bounds.size.height - image.size.height;
    ballYVelocity = 0;
}
```

之后，根据图像的大小计算两个CGRects。一个矩形包围了要绘制新图像的区域，另一个包围了上次绘制的区域。使用这两个矩形可以确保在擦除原来小球的同时绘制新球。

```
CGRect currentImageRect = CGRectMake(currentPoint.x, currentPoint.y,
    currentPoint.x + image.size.width,
    currentPoint.y + image.size.height);
CGRect previousImageRect = CGRectMake(previousPoint.x, previousPoint.y,
    previousPoint.x + image.size.width,
    currentPoint.y + image.size.width);
```

最后，创建一个新矩形，它包含了两个刚计算出的矩形，并将新矩形提供给setNeedsDisplayInRect:，以指示需要重新绘制的视图部分：

```
[self setNeedsDisplayInRect:CGRectUnion(currentImageRect,
    previousImageRect)];
```

本类中的最后一个实质性方法是draw，它用于指明小球的正确位置。此方法在为视图提供了新加速对象之后，被其控制器类的加速计方法调用。此方法首先声明一个静态NSDate变量，此变量用于查明距离上次调用draw方法的时间。

第一次执行此方法，当lastDrawTime是nil时，不需要做任何事，因为没有参考点。因为每秒钟大概有60次更新，所以没有人会注意到。

```
static NSDate *lastDrawTime;
```

```
if (lastDrawTime != nil) {
```

再次执行此方法时，我们可以计算出距离上次调用此方法的时间。对timeIntervalSinceNow返回的值求反，因为lastDrawTime是过去的某个时刻，所以返回的值将是一个负数，表示当前时间和lastDrawTime之间的秒数：

```
NSTimeInterval secondsSinceLastDraw =
    -([lastDrawTime timeIntervalSinceNow]);
```

然后，将当前的加速度与当前的速度相加，计算出两个方向上的新速度。将加速度与secondsSinceLastDraw相乘，因为加速度是与时间一致的。以同样的角度倾斜电话总可以得到相同的加速度。

```
ballYVelocity = ballYVelocity + -(acceleration.y *
    secondsSinceLastDraw);
ballXVelocity = ballXVelocity + acceleration.x *
    secondsSinceLastDraw;
```

之后，根据速度计算出调用此方法之后发生更改的像素。将速度和消耗时间的乘积再乘以500，以创建出自然移动的效果。如果不乘以某个数的话，加速度会非常小，就像小球粘上了蜜一样。

```
CGFloat xAcceleration = secondsSinceLastDraw * ballXVelocity *
    kVelocityMultiplier;
CGFloat yAcceleration = secondsSinceLastDraw * ballYVelocity *
    kVelocityMultiplier;
```

知道了发生更改的像素之后，将当前位置与计算出的加速度相加，并赋值给currentPoint，即可以创建一个新点。通过使用self.currentPoint，便可以使用前面编写的访问方法，而不必将数值直接赋给实例变量。

```
self.currentPoint = CGPointMake(self.currentPoint.x +
    xAcceleration, self.currentPoint.y + yAcceleration);
```

至此，计算已经完成了。剩余的工作是将lastDrawTime更新为现在的时间：

```
[lastDrawTime release];
lastDrawTime = [[NSDate alloc] init];
```

在编译和运行之前，还需要链接CoreGraphics.framework，此时这么做会觉得很轻松。然后，尝试在iPhone或iPod Touch上编译并运行Ball。

如果一切完好，则启动此应用程序。现在可以通过倾斜电话来控制小球的滚动。小球到达屏幕的边缘时应该停止。如果向另一面倾斜，它应该开始向另一个方向滚动。成功！

15.5 小结

我们已经在本章中享受到了物理和奇妙的iPhone加速计的乐趣，并且讲述了使用加速计作为控制设备的基础知识。使用加速计可以设计出无穷无尽的应用程序。因此，既然现在已经掌握了基础知识，那就创建一些好玩的东西带给我们惊喜吧！

对此功能驾轻就熟之后，我们开始学习使用另一种iPhone硬件：内置照相机。

iPhone提供了内置照相机（但iPod Touch缺少此硬件）和Photos应用程序，这在现在已经不足为奇。Photos程序可以帮助用户管理自己拍摄的精彩绝伦的照片。但鲜为人知的是，用户不但可以使用内置照相机拍摄照片，还可以从iPhone的照片库中选择照片。

由于iPhone应用程序受到其沙盒机制的限制，因此通常不能获取这些照片或自己沙盒之外的其他数据。幸而，应用程序可以通过**图像选取器**（image picker）来使用照相机和照片库。顾名思义，图像选取器是从特定源中选择图片的一种机制。通常来说，图像选取器会使用图像列表作为它的源（如图16-1左侧的图片所示）。不过也可以指定照相机作为源（如图16-1右侧的图片所示）。



图16-1 使用图像列表的图像选取器（左）和照相机（右）

16.1 使用图像选取器和 UIImagePickerController

图像选取器界面是通过名为UIImagePickerController的模式控制器类执行的。首先创建此类的一个实例，指定委托（如果没有的话），并指定其图像源，然后以模式化方式启动它。图像选取器会控制iPhone让用户从已有的图像集中选择一幅图片，或者使用照相机拍摄一幅新图片。用户拍摄或者选择图像之后，就可以对所选图像做一些基本的编辑，如缩放或裁剪。如果用户没

有按取消按钮，那么用户拍摄的或从库中选择的图像会传送到委托。

无论是否选择了图像，委托都有责任解除UIImagePickerController，让用户返回到应用程序。

创建UIImagePickerController非常简单。只需按照对多数的类使用的方式分配并初始化实例即可。然而有一点需要注意。并不是每一台运行iPhone OS的设备都有照相机。iPhone Touch便是第一个例子，但是将来，从Apple的装配线上出来的这样的设备会越来越少。在创建UIImagePickerController实例之前，需要先检查运行当前程序的设备是否支持要使用的图像源。例如，在用户可以使用照相机拍摄照片之前，应先确保程序所在的设备上有照相机。可以使用类方法检查UIImagePickerController，如：

```
if ([UIImagePickerController isSourceTypeAvailable:
    UIImagePickerControllerSourceTypePhotoLibrary]) {
```

在本例中，所传递的UIImagePickerControllerSourceTypePhotoLibrary表示我们想让用户使用现有照片库之外的图片。如果所指定的源当前可用，方法isSourceTypeAvailable:将返回YES。除了UIImagePickerControllerSourceTypePhotoLibrary之外，还可以指定另外两个值。

- ❑ UIImagePickerControllerSourceTypeCamera指定用户将使用内置的照相机拍摄图片。并将此图片返回到委托。
- ❑ UIImagePickerControllerSourceTypeSavedPhotosAlbum指定了用户将从现有照片库中选择图像，但选择范围仅限于最近的相册。此选项也可以在iPod Touch上运行，但是其作用不大。

在确保运行程序的设备支持要使用的图像源之后，启动图像选取器就相对容易多了：

```
UIImagePickerController *picker = [[UIImagePickerController alloc] init];
picker.delegate = self;
picker.sourceType = UIImagePickerControllerSourceTypePhotoLibrary;
[self presentViewController:picker animated:YES];
[picker release];
```

在创建并配置了UIImagePickerController之后，我们使用类从UIView中继承的presentModalViewController:animated:方法，将图像选取器呈现给用户。

提示 presentViewController:animated:方法并不仅限于呈现图像选取器，通过对当前可见视图的视图控制器调用此方法，可以按模式将任何视图控制器呈现给用户。

16.2 实现图像选取器控制器委托

用户退出图像选取器界面时，你希望获取的对象需要符合UIImagePickerControllerDelegate协议，此协议定义了两个方法，imagePickerController:didFinishPickingImage:editingInfo:和imagePickerControllerDidCancel:。

当用户成功拍摄了照片或从照片库中选择了一幅照片之后，将调用第一个方法

`imagePickerController:didFinishPickingImage:editingInfo:`。第一个参数是指向之前创建的 `UIImagePickerController` 的指针。第二个参数是包含用户所选照片的 `UIImage` 实例。最后一个参数是一个 `NSDictionary` 实例，如果允许编辑，并且用户裁剪或缩放了图像，那么就会传递此参数。此字典包含存储在键 `UIImagePickerControllerOriginalImage` 下未编辑的原始图像。下面给出了检索原始图像的委托方法的一个例子。

```
- (void)imagePickerController:(UIImagePickerController *)picker
    didFinishPickingImage:(UIImage *)image
    editingInfo:(NSDictionary *)editingInfo {

    UIImage *selectedImage = image;
    UIImage *originalImage = [editingInfo objectForKey:
        UIImagePickerControllerOriginalImage];

    // do something with selectedImage and originalImage

    [picker dismissModalViewControllerAnimated:YES];
}
```

通过存储在键 `UIImagePickerControllerCropRect` 下的 `NSValue` 对象，`editingInfo` 字典也可以指示在编辑期间选择了整个图像的哪一部分。也可以将此字符串转换至 `CGRect`：

```
NSValue * cropRect = [editingInfo
    objectForKey:UIImagePickerControllerCropRect];
CGRect theRect = [cropRect CGRectValue];
```

完成转换之后，`theRect` 可以指明在编辑过程中所选定的原始图像的部分。如果不需要此信息，则可以忽略。

注意 如果返回到委托的图像来自照相机，那么此图像不会存储在照片库中。在必要时保存此图像的工作将由应用程序负责。

在用户决定取消此过程而不拍照或选择图像时将调用另一个委托方法，`imagePickerControllerDidCancel`。当图像选取器调用此委托方法时，所通报的只是用户已经结束使用选取器并且没有选择任何图像。

在 `UIImagePickerControllerDelegate` 协议中的两种方法都标记为可选，但实际上不是，原因是：必须通过本身解除图像选取器这样的模式视图。因此，在用户取消图像选取器时，即使不需要采取任何应用程序特定的动作，也仍然需要解除选取器。至少，`imagePickerControllerDidCancel` 方法要像这样才能保证程序正确运行：

```
- (void)imagePickerControllerDidCancel:(UIImagePickerController *)picker {

    [picker dismissModalViewControllerAnimated:YES];
}
```

16.3 实际测试相机和库

在本章中，我们所构建的应用程序将允许用户使用照相机拍摄照片或从照片库中选择一幅图片，然后在图像视图中显示所选图片（如图16-2所示）。如果用户使用的设备没有照相机，则隐藏Take Picture按钮和Pick from Library按钮，只允许从照片库中选择图片。

在Xcode中，使用基于视图的应用程序模板创建一个新项目，将应用程序命名为Camera。此应用程序需要几个输出口。一个输出口指向图像视图，以便使用从图像选取器返回的图像对它进行更新。还需要一个指向Take New Picture按钮和Select from Camera Roll按钮的输出口，如果设备不包含照相机，则可以隐藏这两个按钮。此外，还需要两个操作方法，一个用于Take New Picture和Select from Camera Roll按钮，另一个用于允许用户从照片库中选择现有图片。展开Classes和Resources文件夹，可以看到所有相关文件。

单击CameraViewController.h，并做如下更改：

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

@interface CameraViewController : UIViewController
    <UIImagePickerControllerDelegate, UINavigationControllerDelegate> {
    IBOutlet UIImageView *imageView;
    IBOutlet UIButton *takePictureButton;
    IBOutlet UIButton *selectFromCameraRollButton;
}
@property (nonatomic, retain) UIImageView *imageView;
@property (nonatomic, retain) UIButton *takePictureButton;
@property (nonatomic, retain) UIButton *selectFromCameraRollButton;
- (IBAction)getCameraPicture:(id)sender;
- (IBAction)selectExistingPicture;
@end
```

首先需要注意，类必须遵循两个不同的协议：UIImagePickerControllerDelegate和UINavigationControllerDelegate。因为UIImagePickerController是UINavigationController的子类，所以类必须遵循这两个协议。UINavigationControllerDelegate中的方法都是可选的，使用图像选取器不一定需要它们，但是必须与此协议保持一致，否则编译器会发出警告。此处的其他内容都相当简单直观，完成更改后保存文件。此时，双击CameraViewController.xib，在Interface Builder中打开此文件。

16.3.1 设计界面

从库中拖出3个Round Rect Button，并将它们放置在标签为View的窗口中。将它们上下排列



图16-2 运行中的Camera应用程序

放置。双击最上面的按钮，将标题命名为Take New Picture。双击中间的按钮，将标题命名为Pick from Camera Roll。然后双击最下面的按钮，将标题命名为Pick from Library。然后，从库中拖出一个Image View，将它放置在其他按钮上方。拉伸视图使它占据按钮上方的所有空间，如图16-2所示。

此时，按下Control键并从File's Owner图标拖至图像视图，并选择imageView输出口。再次按下Control键并从File's Owner拖至Take New Picture按钮，并选择takePictureButton输出口。最后，从File's Owner拖至Pick from Camera Roll按钮，并选择selectFromCameraRollButton输出口。

然后，选择Take New Picture按钮，并按⌘2打开连接检查器。从Touch Up Inside事件拖至File's Owner，并选择getCameraPicture:操作。接着单击Pick from Camera Roll按钮，从连接检查器上的Touch Up Inside事件拖至File's Owner，并选择getCameraPicture:操作。之后选择Pick from Library按钮。从连接检查器上的Touch Up Inside事件拖至File's Owner，并选择selectExistingPicture操作。完成这些连接之后，保存并关闭nib文件，返回至Xcode。

16.3.2 实现照相机视图控制器

单击CameraViewController.m，并做如下更改：

```
#import "CameraViewController.h"

@implementation CameraViewController
@synthesize imageView;
@synthesize takePictureButton;
@synthesize selectFromCameraRollButton;
- (void)viewDidLoad {
    if (![UIImagePickerController isSourceTypeAvailable:
        UIImagePickerControllerSourceTypeCamera]) {
        takePictureButton.hidden = YES;
        selectFromCameraRollButton.hidden = YES;
    }
}
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [imageView release];
    [takePictureButton release];
}
```

```

        [selectFromCameraRollButton release];
        [super dealloc];
    }
#pragma mark -
    - (IBAction)getCameraPicture:(id)sender {
        UIImagePickerController *picker =
            [[UIImagePickerController alloc] init];
        picker.delegate = self;
        picker.allowsImageEditing = YES;
        picker.sourceType = (sender == takePictureButton) ?
            UIImagePickerControllerSourceTypeCamera :
            UIImagePickerControllerSourceTypeSavedPhotosAlbum;
        [self presentViewController:picker animated:YES];
        [picker release];
    }
    - (IBAction)selectExistingPicture {
        if ([UIImagePickerController isSourceTypeAvailable:
            UIImagePickerControllerSourceTypePhotoLibrary]) {
            UIImagePickerController *picker =
                [[UIImagePickerController alloc] init];
            picker.delegate = self;
            picker.sourceType = UIImagePickerControllerSourceTypePhotoLibrary;
            [self presentViewController:picker animated:YES];
            [picker release];
        }
        else {
            UIAlertView *alert = [[UIAlertView alloc]
                initWithTitle:@"Error accessing photo library"
                message:@"Device does not support a photo library"
                delegate:nil
                cancelButtonTitle:@"Drat!"
                otherButtonTitles:nil];
            [alert show];
            [alert release];
        }
    }
#pragma mark -
    - (void)imagePickerController:(UIImagePickerController *)picker
        didFinishPickingImage:(UIImage *)image
        editingInfo:(NSDictionary *)editingInfo {
        imageView.image = image;
        [picker dismissModalViewControllerAnimated:YES];
    }
    - (void)imagePickerControllerDidCancel:(UIImagePickerController *)picker {
        [picker dismissModalViewControllerAnimated:YES];
    }
@end

```

所编写的第一个方法是viewDidLoad:，其作用是检查应用程序所在的设备上是否有照相机：

```
if (![UIImagePickerController isSourceTypeAvailable:
    UIImagePickerControllerSourceTypeCamera]){
```

如果该设备上没有照相机，则隐藏两个与照相机相关的按钮：

```
    takePictureButton.hidden = YES;
    selectFromCameraRollButton.hidden = YES;
}
```

在第一个操作方法getCameraPicture:中，分配并初始化UIImagePickerController实例：

```
UIImagePickerController *picker =
    [[UIImagePickerController alloc] init];
```

然后将self赋值给图像选取器的委托，并指明用户可以在拍摄照片之后对它进行编辑：

```
picker.delegate = self;
picker.allowsImageEditing = YES;
```

然后，根据所按下的按钮设置sourceType。如果用户按下的是Take New Picture按钮，则告知选取器允许使用照相机。如果用户按下的是Pick from Camera Roll按钮，则使用UIImagePickerControllerSourceTypeSavedPhotosAlbum，后者在有照相机的设备上允许用户从当前相册中进行选择图像。

```
picker.sourceType = (sender == takePictureButton) ?
    UIImagePickerControllerSourceTypeCamera :
    UIImagePickerControllerSourceTypeSavedPhotosAlbum;
```

最后，呈现图像选取器并释放实例：

```
[self presentViewController:picker animated:YES];
[picker release];
```

我们再次检查此设备是否支持照相机，因为在这种情况下，触发此操作方法的按钮是不可见的。如果设备上没有照相机，则永远不要调用此方法。

第二个操作方法与第一个相似。此方法允许用户从照片库中选择图像。如果照片库存在，则创建一个图像选取器，其sourceType为UIImagePickerControllerSourceTypePhotoLibrary。

```
if ([UIImagePickerController isSourceTypeAvailable:
    UIImagePickerControllerSourceTypePhotoLibrary]) {
    UIImagePickerController *picker =
        [[UIImagePickerController alloc] init];
    picker.delegate = self;
    picker.sourceType = UIImagePickerControllerSourceTypePhotoLibrary;
    [self presentViewController:picker animated:YES];
    [picker release];
}
```

如果设备不包含照片库，则显示错误警告。需要注意的是，照片库为空和没有照片库是不同的。当前所有的iPhone OS设备都支持照片库，所以此代码应该会一直有用，但是从安全的角度考虑，既然显示警告要好于意外崩溃，因此这种编码方式是一个不错的主意：

```

else {
    UIAlertView *alert = [[UIAlertView alloc]
        initWithTitle:@"Error accessing photo library"
        message:@"Device does not support a photo library"
        delegate:nil
        cancelButtonTitle:@"Drat!"
        otherButtonTitles:nil];
    [alert show];
    [alert release];
}

```

下面是两个委托方法。第一个方法是 `imagePickerController:didFinishPickingImage:editingInfo`，用户完成使用图像选取器时会调用此方法。用户选择幅图片之后也会调用此方法，而不论使用的是哪种源类型。此方法的作用是设置 `imageView`，让它显示返回的图像：

```
imageView.image = image;
```

之后，让选取器解除自身，以使用户返回到应用程序视图：

```
[picker dismissModalViewControllerAnimated:YES];
```

在 `imagePickerControllerDidCancel` 中重复解除模式视图控制器的最后一步。如果用户取消的话，则不需要做其他的事情，只需关闭图像选取器，否则它会一直在那里挡住应用程序视图。

所需要做的仅仅如此。此时甚至不需要链接到任何其他库。然后编译并运行程序。如果应用程序在仿真器上运行，则没有拍摄照片的选项，且照片库也是空的。如果有机会在实际设备上运行程序，请尝试继续操作。此时应该可以拍摄照片，并可以使用手指捏合的姿势放大和缩小图片（如图16-3所示）。

如果在单击 `Use Photo` 按钮之前放大图片，则在委托方法中返回至应用程序的图像将会是裁剪后的图像。

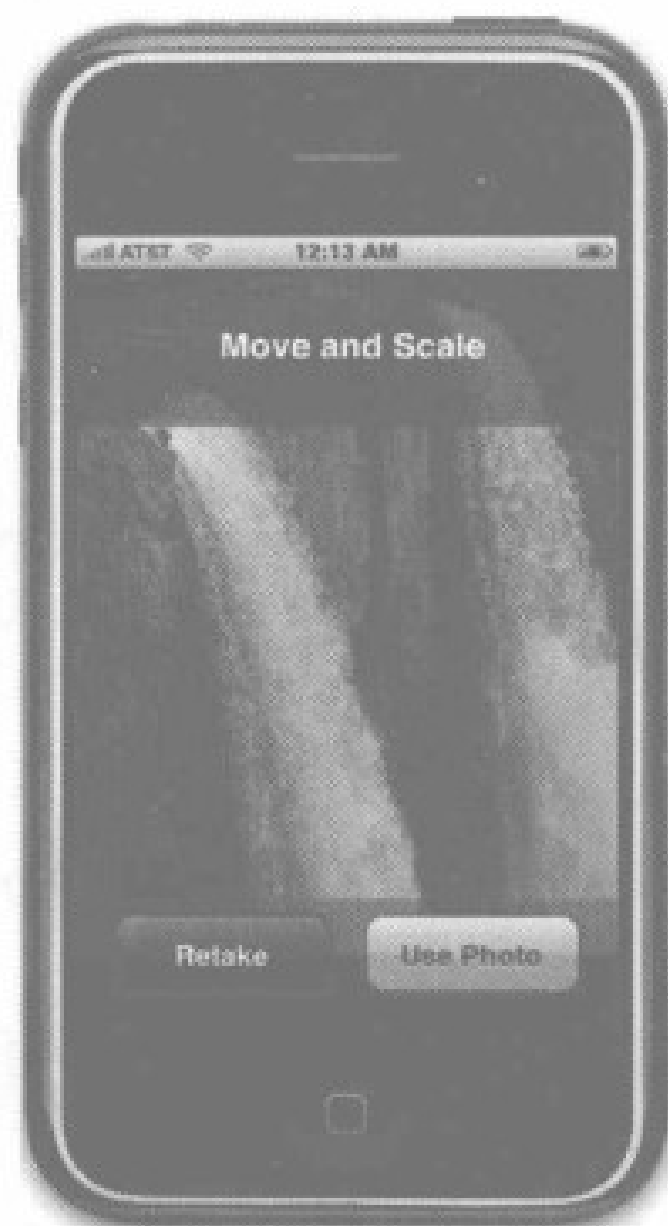


图16-3 如果将 `allowsEditing` 设置为 `YES`，则用户可以在返回应用程序之前缩放和裁剪图像

16.4 小结

用户已经可以使用iPhone照相机拍摄照片，并在应用程序中使用它们。没错，这就是本章的所有内容。如有必要，甚至还可以允许用户对拍摄的图像稍加编辑。

在下一章中，我们要学习的是将iPhone应用程序翻译为其他语言，让更广泛的用户群体接受它。准备好了吗？直接翻开新的一页。出发！

在写作本书时，iPhone已经（或者即将）遍及70个不同的国家，并且显而易见，此数字将会随时间不断增长。现在，你可以在南极洲以外的任何一个洲购买和使用iPhone。

如果计划通过iPhone App Store发布应用程序，那么潜在的市场将会远远大于仅在自己的国家说自己语言的人们。好在iPhone拥有健壮的本地化（localization）体系结构，使用它不但可以轻松地将应用程序（或者由其他程序将它）翻译成多种语言，甚至可以翻译成同一语言的多种方言。想为英式英语使用者和美式英语使用者提供不同的术语吗？没问题。

一点问题都没有，只要已经正确地编写了代码。翻新现有的应用程序以支持本地化，比起以同样的方式从头编写应用程序要困难得多。在本章中，我们会讲述如何编写代码可以更轻松地实现本地化，然后将对一个应用程序示例进行本地化。

17.1 本地化体系结构

在运行非本地化应用程序时，应用程序的所有文本都会以开发人员自己的语言呈现，也就是开发基础语言。

当开发人员决定对其应用程序进行本地化时，他们会在应用程序束中为每种支持的语言创建一个子目录。每种语言的子目录中都包含有一个翻译为此种语言的应用程序资源子集。每个子目录都被称为一个本地化项目，也称为本地化文件夹。本地化文件夹通常使用.lproj作为其扩展名。

在Settings应用程序中，用户可以设置语言和区域格式。例如，如果用户语言是英语，那么可选地区可以是美国或澳大利亚等——即所有讲英语的地区。

当本地化的应用程序需要载入某一资源时，如图像、属性列表或nib文件，应用程序会检查用户的语言和地区，并查找与此设置相匹配的本地化文件夹。如果找到了相应的文件夹，那么它会载入此资源的本地化版本而不是基础版本。

对于选择法语作为iPhone语言，选择法国作为地区的用户，应用程序会先查找名为fr_FR.lproj的本地化文件夹。文件夹名称的前两个字母是ISO二位代码，表示法语。下划线后的两个字母是ISO国家（地区）代码，表示法国。

如果应用程序找不到匹配的二位代码，那么它会查找匹配的ISO三位代码。所有的语言都有三位代码。只有一部分语言有二位代码。

说明 在ISO网站上可以找到当前ISO国家（地区）代码列表。二位和三位代码都是ISO3166标准的一部分：http://www.iso.org/iso/country_codes.htm。

在之前的示例中，如果应用程序找不到名为fr_FR.lproj的文件夹，它会查找名为fre_FR或fra_FR的本地化文件夹。所有语言都至少有一个三位代码。某些语言有两个三位代码，一种是此语言的英语拼写，一种是本地拼写。当一种语言既有二位代码又有三位代码时，则使用二位代码。

如果应用程序找不到精确匹配的文件夹，那么它会随即查找应用程序束中仅语言代码匹配（地区代码不匹配）的本地化文件夹。因此，对于来自法国的讲法语的人，应用程序随后会查找名为fr.lproj的本地化项目。如果找不到此名称的语言项目，它会尝试查找fre.lproj，然后查找fra.lproj。如果都找不到，它会查找French.lproj。最后一种结构是为了支持旧式Mac OS X应用程序，一般来说，应用程序会避免使用它（此规则也有例外，我们将会在本章后面部分讲述）。

如果应用程序找不到与语言/地区的组合相匹配或仅与语言相匹配的语言项目，那么它会使用开发基础语言中的资源。如果找到了适合的本地化项目，那么对于任何所需要的资源，它将总是先查找这里。例如，若载入一个使用imageName:的UIImage，它先会在本地化项目中查找使用指定名称的图像。如果找到了此图像，它就会使用它。如果没有找到，它将会退回到基础语言资源。

如果某个应用程序与多个本地化项目相匹配，例如，一个名为fr_FR.lproj的项目和一个名为fr.lproj的项目，那么它会先在更精确的匹配中查找，在本例中是fr_FR.lproj。如果在此处找不到资源，它将会查找fr.lproj。这样便可以在一个语言项目中对所有此语言的使用者提供共有的资源，仅本地化受到不同方言或地理地区影响的资源。

你只需要本地化受到语言或国家（地区）影响的资源。如果应用程序中的图像没有使用词汇并且其含义是通用的，那么就没有必要本地化此图像。

17.2 使用字符串文件

在源代码中，字符串文字和字符串常量有何作用？下面参考第16章中的一段源代码：

```
UIAlertView *alert = [[UIAlertView alloc]
    initWithTitle:@"Error accessing photo library"
    message:@"Device does not support a photo library"
    delegate:nil
    cancelButtonTitle:@"Drat!"
    otherButtonTitles:nil];
[alert show];
[alert release];
```

如果我们已经努力完成了对特定受众的应用程序本地化工作，我们当然不想看到出现以开发基础语言编写的警告。

上面问题的答案是，将这些字符串存储到特定的文本文件中，即字符串文件中。字符串文件实际上是Unicode（UTF-16）文本文件，其中包含了字符串配对列表，每项都标识了注释。

下面的示例描述了应用程序中字符串文件的格式。

```
/* Used to ask the user his/her first name */
"First Name" = "First Name";

/* Used to get the user's last name */
"Last Name" = "Last Name";

/* Used to ask the user's birth date */
"Birthdate" = "Birthdate";
```

在“/*”和“*/”字符之间的值是翻译者的注释。它们对应用程序来说没有用处，可以安全地删除，但最好不要这样做。因为它们给定了上下文，显示了一段特定的字符串如何应用于程序中。

有人会注意到每一行都列出了相同的字符串两次。等号左侧的字符串充当键，无论使用什么语言，它总是包含相同的值。等号右侧的值用于翻译为本地语言。因此，如果将前面的字符串文件本地化为法语，可能会是这样：

```
/* Used to ask the user his/her first name */
"First Name " = "Prénom";

/* Used to get the user's last name */
"Last Name " = "Nom de famille";

/* Used to ask the user's birth date */
"Birthdate" = "Anniversaire";
```

创建字符串文件

人们不会通过手动输入来创建字符串文件。而是，将所有本地化的文本字符串嵌入到代码内特定的宏中。完成源代码并做好本地化的准备工作之后，可以运行一个名为genstrings的命令行程序，它将在所有代码文件中搜索出现的宏，提取出所有的字符串，并将它们嵌入到本地化的字符串文件中。

下面显示了宏如何工作。我们从传统的字符串声明开始：

```
NSString *myString = @"First Name";
```

要本地化此字符串，需要这样做：

```
NSString *myString = NSLocalizedString(@"First Name",
    @"Used to ask the user his/her first name");
```

NSLocalizedString宏使用了两个参数。第一个参数是基础语言中字符串的值。在未本地化的情况下，应用程序将使用此字符串。第二个参数充当字符串文件中的注释。

NSLocalizedString在合适的本地化项目内部的应用程序束中查找名为localizable.strings的字符串文件。如果没有找到此文件，则返回其第一个参数，而此字符串会出现在开发基础语言中。如果此应用程序没有本地化，则字符串通常仅在开发时显示在基础语言中。

如果NSLocalizedString找到了字符串文件，则会搜索此文件中与第一个参数相匹配的行。

在前面的示例中，`NSString`将在字符串文件中搜索字符串“First Name”。如果在本地化项目中没有找到与用户语言设置相匹配的项，它会在基础语言中查找字符串文件并使用其中的值。如果没有字符串文件，它会只使用传递给`NSString`宏的第一个参数。

下面我们来看一看此过程。

17.3 现实中的 iPhone：本地化应用程序

现在创建一个显示用户当前区域设置的小应用程序。区域设置（`NSLocale`实例）同时描述了用户的语言和地区。在与用户交互时，系统使用区域设置确定使用哪种语言及如何显示日期、货币和时间信息等。创建应用程序之后，需要将它本地化为其他语言。在此可以学习到如何本地化 nib 文件、字符串文件、图像，甚至是应用程序图标。在图17-1中可以看到应用程序的外观。顶部的名称来自用户的区域设置。在 nib 文件中将视图下方左侧的词设置为静态标签。使用输出口将视图下方右侧的词设置为可编程的。屏幕底部的旗帜图像是一幅静态 `UIImageView`。

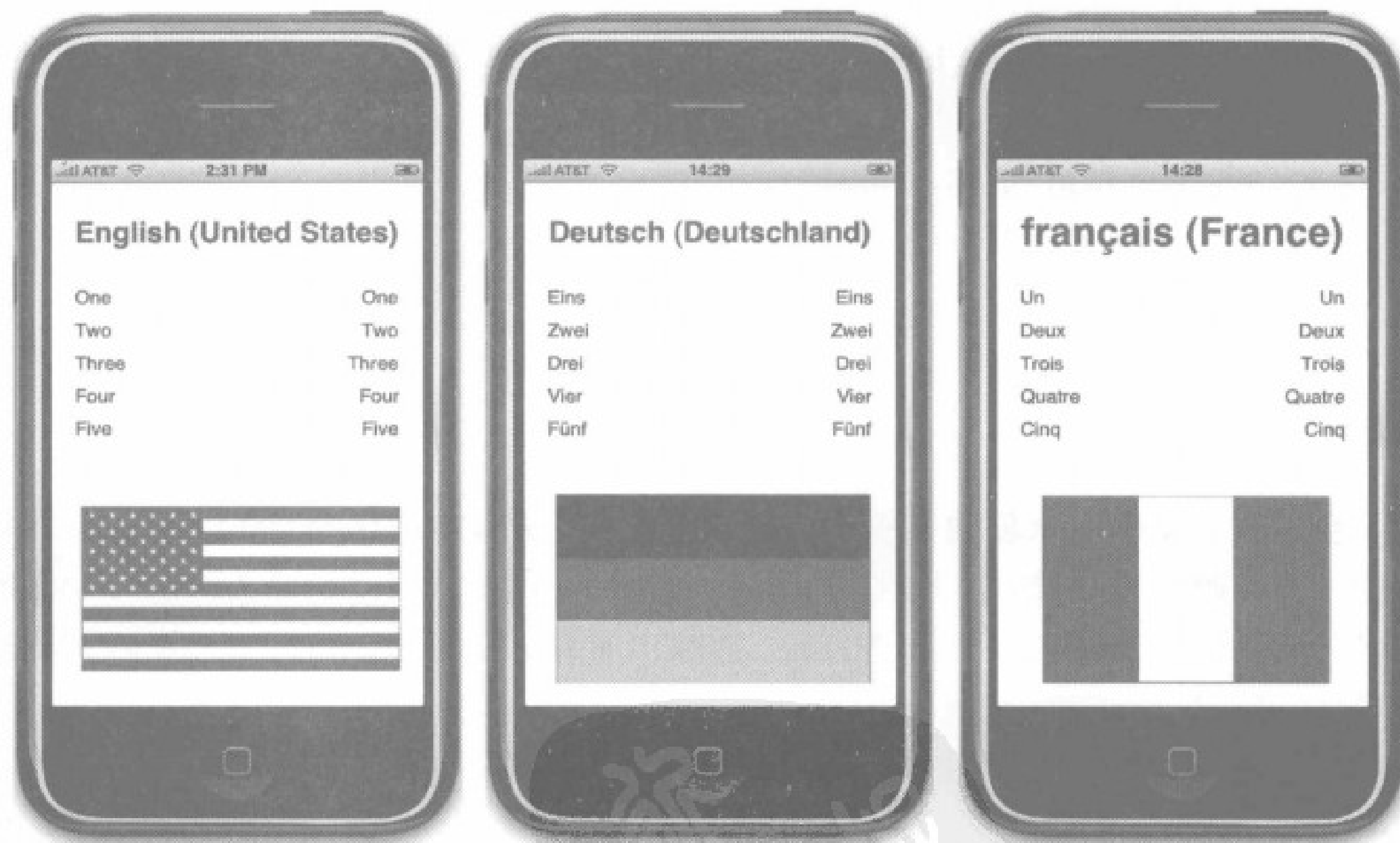


图17-1 使用3种不同语言/地区设置进行显示的LocalizeMe应用程序

现在开始实际操作。在Xcode中使用基于视图的应用程序模板新建一个项目，并将其命名为LocalizeMe。查看17 LocalizeMe文件夹，可以找到一个名为Resources的子文件夹。在Resources内部，可以找到名为Base Language的目录。在文件夹中，可以找到两幅图像：icon.png和flag.png。将两幅图像拖到项目的Resources文件夹中。然后单击Info.plist，并将Icon file值设置为icon.png，即将图标图像用作应用程序图标。

6个标签都需要创建输出口：1个用于视图顶部的蓝色标签，5个用于下方右侧的词条。打开Classes文件夹，单击LocalizeMeViewController.h并做如下更改：

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

@interface LocalizeMeViewController : UIViewController {
    IBOutlet UILabel *localeLabel;
    IBOutlet UILabel *label1;
    IBOutlet UILabel *label2;
    IBOutlet UILabel *label3;
    IBOutlet UILabel *label4;
    IBOutlet UILabel *label5;
}
@property (nonatomic, retain) UILabel *localeLabel;
@property (nonatomic, retain) UILabel *label1;
@property (nonatomic, retain) UILabel *label2;
@property (nonatomic, retain) UILabel *label3;
@property (nonatomic, retain) UILabel *label4;
@property (nonatomic, retain) UILabel *label5;
@end
```

然后双击LocalizeMeViewController.xib, 在Interface Builder中打开此文件。打开之后, 从库中拖一个Label放置在窗口顶部。重新调整其大小使之填满视图中两条蓝色引导线之间的整个宽度。选定此标签, 按⌘B加粗文本, 将文本改为居中对齐方式, 并使用属性检查器将文本设置为亮蓝色。

根据需要可以使字号变大。如需调整字号, 请在Font菜单中选择Show Fonts, 在此将字体设置为期望大小。如果在属性检查器中选定了Adjust to fit, 那么文本在太长不能适配时会对调整大小。

放置好标签之后, 按下Control键并将File's Owner图标拖到此新标签上, 然后选择localeLabel输出口。

然后, 在库中将其他5个Label拖到使用蓝色引导线的左边界旁, 上下依次放置, 如图17-1所示。双击顶部的标签, 把Label更改为One。然后对其他4个刚刚添加的标签重复此步骤, 使之涵盖数字1至5。

从库中再拖出5个Label, 这次放置在右边界旁。使用属性检查器将文本对齐方式更改为右对齐, 并增加标签大小, 使之从右边的蓝色引导线伸展至视图中部。按下Control键并将File's Owner拖至5个新标签上, 使它们分别连接到不同编号的标签输出口。然后依次双击这些新标签, 删除其文本。以后会将这些值设置为可编程的。

最后, 从库中拖出一个Image View到视图底部。在属性检查器中, 在视图的Image属性中选择flag.png, 调整图像大小使之位于两条蓝色引导线之间。然后, 在属性检查器中, 将Mode属性由Center改为Aspect Fit。这样做是为了确保本地化版本的图片看起来合适, 因为并非所有的国旗都有相同的纵横比。选择此选项会使图像视图调整置于其中的图像至合适大小, 但这样可以维持正确的纵横比(高度与宽度之比)。根据需要, 可以将国旗调整得更高, 直至其边缘接触到蓝色引导线。

保存并关闭nib文件, 返回Xcode。单击LocalizeMeViewController.m, 并做如下更改:

```
#import "LocalizeMeViewController.h"

@implementation LocalizeMeViewController
@synthesize localeLabel;
@synthesize label1;
@synthesize label2;
@synthesize label3;
@synthesize label4;
@synthesize label5;

- (void)viewDidLoad {

    NSLocale *locale = [NSLocale currentLocale];
    NSString *displayNameString = [locale
        displayNameForKey:NSLocaleIdentifier
        value:[locale localeIdentifier]];
    localeLabel.text = displayNameString;

    label1.text = NSLocalizedString(@"One", @"The number 1");
    label2.text = NSLocalizedString(@"Two", @"The number 2");
    label3.text = NSLocalizedString(@"Three", @"The number 3");
    label4.text = NSLocalizedString(@"Four", @"The number 4");
    label5.text = NSLocalizedString(@"Five", @"The number 5");
    [super viewDidLoad];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [localeLabel release];
    [label1 release];
    [label2 release];
    [label3 release];
    [label4 release];
    [label5 release];
    [super dealloc];
}
@end
```

在此类中唯一需要查看的是viewDidLoad方法。在此首先要做的事情是获取一个代表用户当前区域设置的NSLocale实例，区域设置在iPhone的Settings应用程序中进行设置，它包含了用户的

语言和地区偏好。

```
NSLocale *locale = [NSLocale currentLocale];
```

17.3.1 查看当前区域设置

关于代码的下一行可能需要一点解释。NSLocale的工作原理就像一本字典。其中包含关于当前用户偏好的成批信息，包括所使用的货币名称和期望的日期格式。在NSLocale的API参考中可以找到这些值的完整列表。

在代码的下一行中，可以找到区域设置标识符，即此区域设置所代表的语言和/或地区的名称。在此使用的函数叫`displayNameForKey:value:`。对于以特定语言请求的项，使用此方法可以返回此项的值。

例如，法语的显示名称在法语中是Français，但在英语中是French。使用此方法可以找到任何关于区域设置的数据，以便对任何用户进行正确的显示。在此例中，所获取的是使用区域设置语言的区域设置显示名称，所以为第二个变量传递了`[locale localeIdentifier]`。localeIdentifier是一个字符串，其格式是之前创建语言项目时所使用的格式。对于美式英语使用者来说，它是en_US；对于法国的法语使用者来说，它是fr_FR。

```
NSString *displayNameString = [locale
    displayNameForKey:NSLocaleIdentifier
    value:[locale localeIdentifier]];
```

有了显示名称，就可以用它设置视图顶部的标签。

```
localeLabel.text = displayNameString;
```

然后，以开发基础语言将其他5个标签依次编号为1至5。此处还有注释解释每个词的意思。如果词意很明显，也可以传递一个空字符串，但是传递给第二个变量的任何字符串在字符串文件中都会转换为注释，使用此注释可以与相应的翻译人员进行沟通。

```
label1.text = NSLocalizedString(@"One", @"The number 1");
label2.text = NSLocalizedString(@"Two", @"The number 2");
label3.text = NSLocalizedString(@"Three", @"The number 3");
label4.text = NSLocalizedString(@"Four", @"The number 4");
label5.text = NSLocalizedString(@"Five", @"The number 5");
```

17.3.2 测试 LocalizeMe

现在，运行此应用程序。可以使用仿真器或某台设备对此进行测试。仿真器似乎隐藏了某些语言和地区设置，因此人们也许更想在设备（如果有的话）上完成测试。启动后的应用程序如图17-2所示。

使用NSLocalizedString宏代替静态宏，为本地化做好了准备，但还没有开始本地化。如果使用仿真器或iPhone上的Settings应用程序更改为另一种语言或另一个地区，那结果看上去应该基本相同，除了视图顶部的标签（见图17-3）。

说明 如果在启动此应用程序时遇到了问题，可以尝试按照本章其他部分中的本地化步骤去。如果不起作用，可以参考网站<http://www.iphonedevbook.com>获得更多建议。

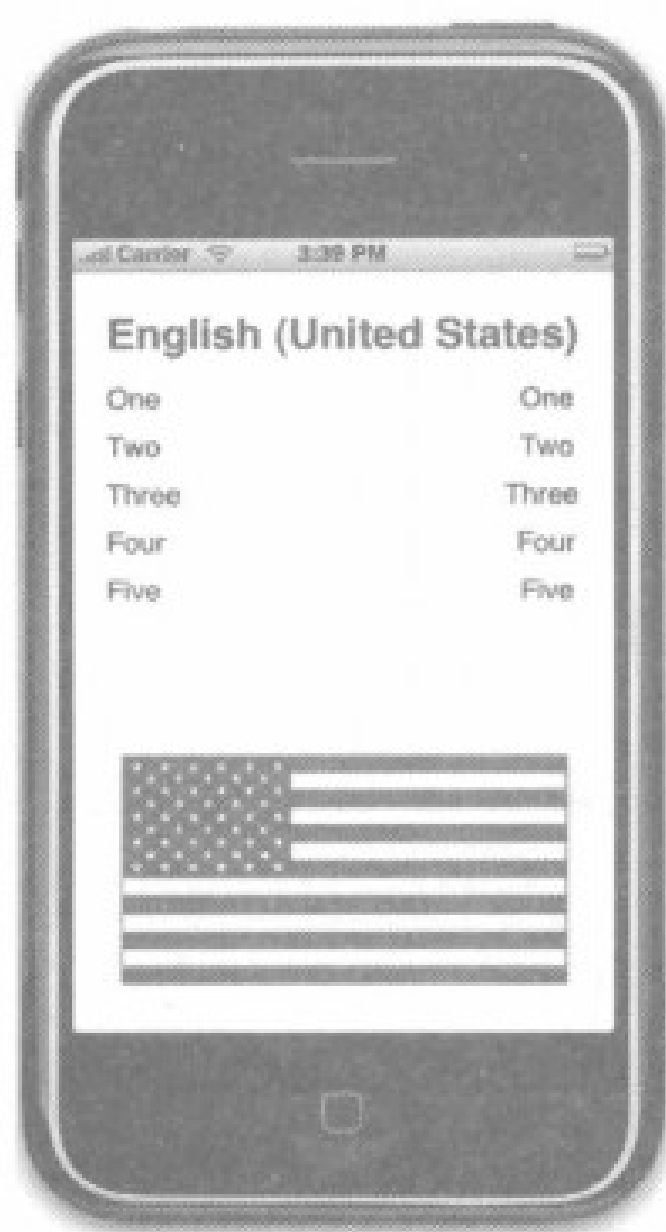


图17-2 运行基础语言的应用程序。应用程序已做好本地化的准备但还未本地化

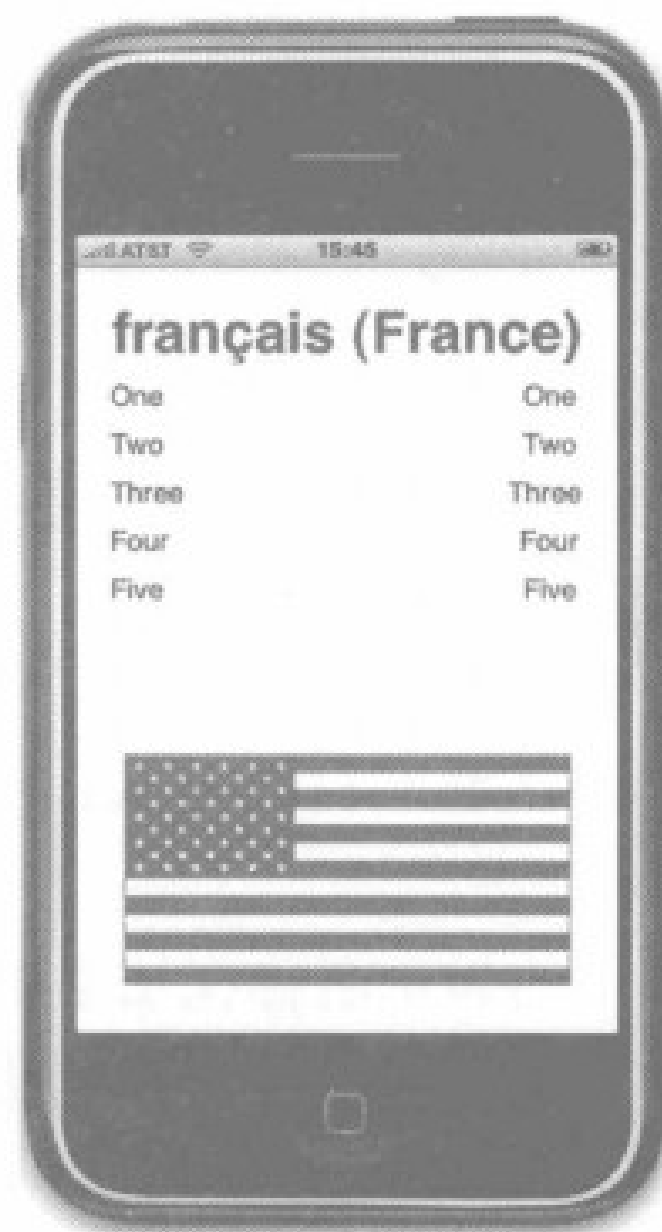


图17-3 设置为使用法语的iPhone上运行的非本地化的应用程序

17.3.3 本地化 nib 文件

现在开始本地化nib文件。本地化任何文件的基本步骤都是一样的。在Xcode中，请单击LocalizeMeViewController.xib，然后按 \mathbb{I} 为此文件打开Info窗口。如果当前窗口没有显示General选项卡，则选定General选项卡。单击窗口左下角的Make File Localizable按钮（如图17-4所示）。

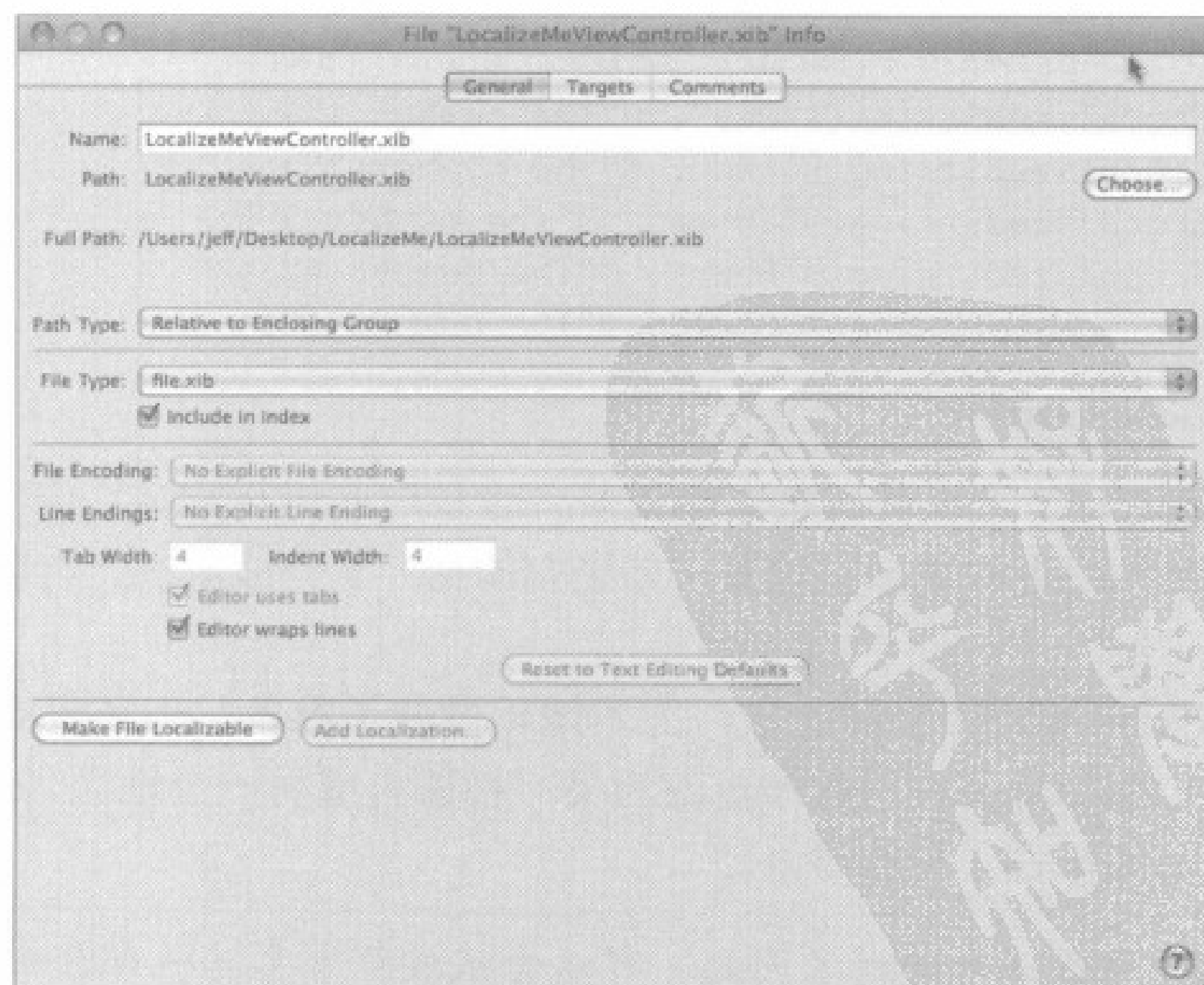


图17-4 LocalizeMeViewController.xib的Info窗口

单击Make File Localizable按钮之后,窗口会切换到Targets选项卡。关闭Info窗口,并在Xcode中查看Groups & Files窗格。注意,现在LocalizeMeViewController.xib文件旁有一个展开三角形,表明它是一个组或文件夹。展开它进行查看(如图17-5所示)。

17.3.4 查看本地化的项目结构

在项目中,LocalizeMeViewController.xib有一个子值——English。它由系统自动创建,代表开发基础语言。打开Finder并打开LocalizeMe项目文件夹。在此应该可以看到一个名为English.lproj的新文件夹(如图17-6所示)。

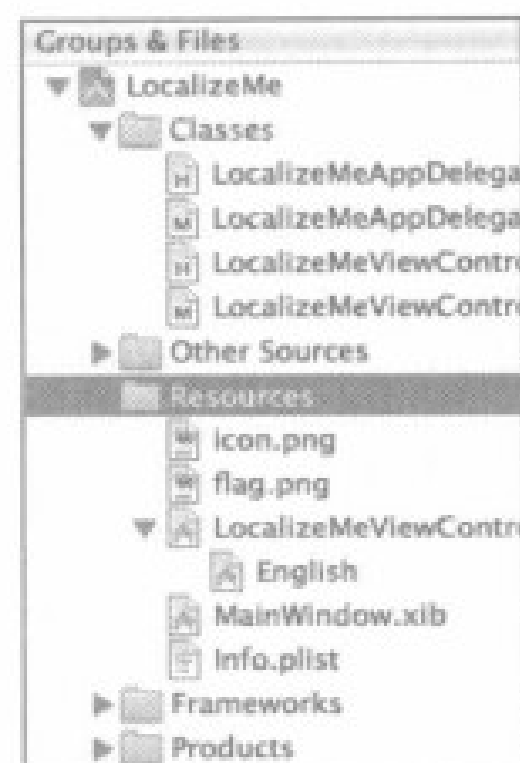


图17-5 可本地化的文件有一个展开三角形,并对所添加的每种语言和地区都有一个子值

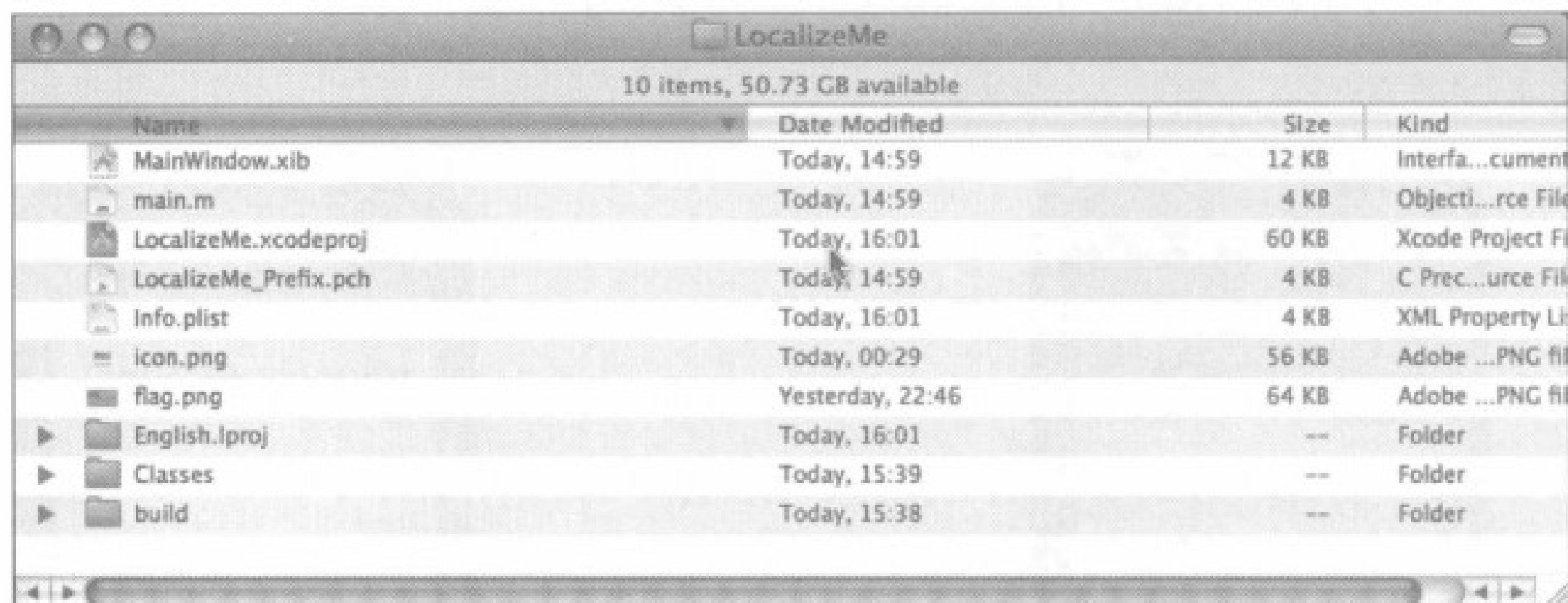


图17-6 通过使文件本地化,Xcode为基础语言创建了一个语言项目文件夹

在写作此书时,Xcode仍然对开发基础语言使用旧项目名称——English.lproj,而非沿用苹果公司使用ISO双字母语言代码的本地化传统,如果使用按照此传统,那么将会生成名为en.lproj的文件夹。此问题列于Xcode 3.1版本说明中。因为它运行得很好,所以没有必要更改此名称,但在添加新本地化时需要使用ISO代码。

再次单击Groups & Files窗格中的LocalizeMeViewController.xib,然后按⌘1返回到Info窗口。返回至General选项卡,并单击Add Localization按钮。此时会弹出一个下拉表单,用于输入新本地化的名称。在此要为法语添加本地化,所以键入fr。不要在下拉菜单中选择French,因为这样将会使用旧名称French。

提示 在处理区域设置时,语言代码是小写的,但是国家(地区)代码是大写的。因此,用于法语项目的正确名称是fr.lproj,但用于本土法语(在法国的人说的法语)的项目是fr_FR.lproj,而不是fr_fr.lproj或FR_fr.lproj。在Mac OS X中,文件系统不区分大小,因此所有的选项都可以起作用,但iPhone的文件系统是区分大小写的,因此正确的匹配大小写非常重要。

按下回车键之后，Xcode将在名为fr.lproj的项目文件夹中创建一个新本地化项目，并将LocalizeMeViewController.xib复制到此处。在Groups & Files窗格中，LocalizeMeViewController.xib应当有两个子值，English和fr。双击fr打开向法语使用者显示的nib文件。

在Interface Builder中打开的nib文件与以前建立的文件完全相同，因为这是它的一个副本。对此文件所做的任何更改都会向法语使用者显示，所以需要双击左侧的各个标签，将它们由One、Two、Three、Four、Five改为Un、Deux、Trois、Quatre、Cinq。完成更改之后，请保存此nib文件，并返回至Xcode。现在，nib文件已经本地化为法语。编译并运行此程序。在它启动之后，轻按主菜单按钮。打开Settings应用程序并选择General行，然后选择标签为International的行。在此处可以更改语言和地区偏好（如图17-7所示）。

将Region Format由United States改为France（在French行的下面），然后将Language由English改为French。一般先更改Region Format，因为一旦更改了语言，iPhone将重启并返回至主菜单。现在iPhone已经设置为使用法语，因此需要再次启动LocalizeMe。这时，左侧的单词应该显示为法语（如图17-8所示）。

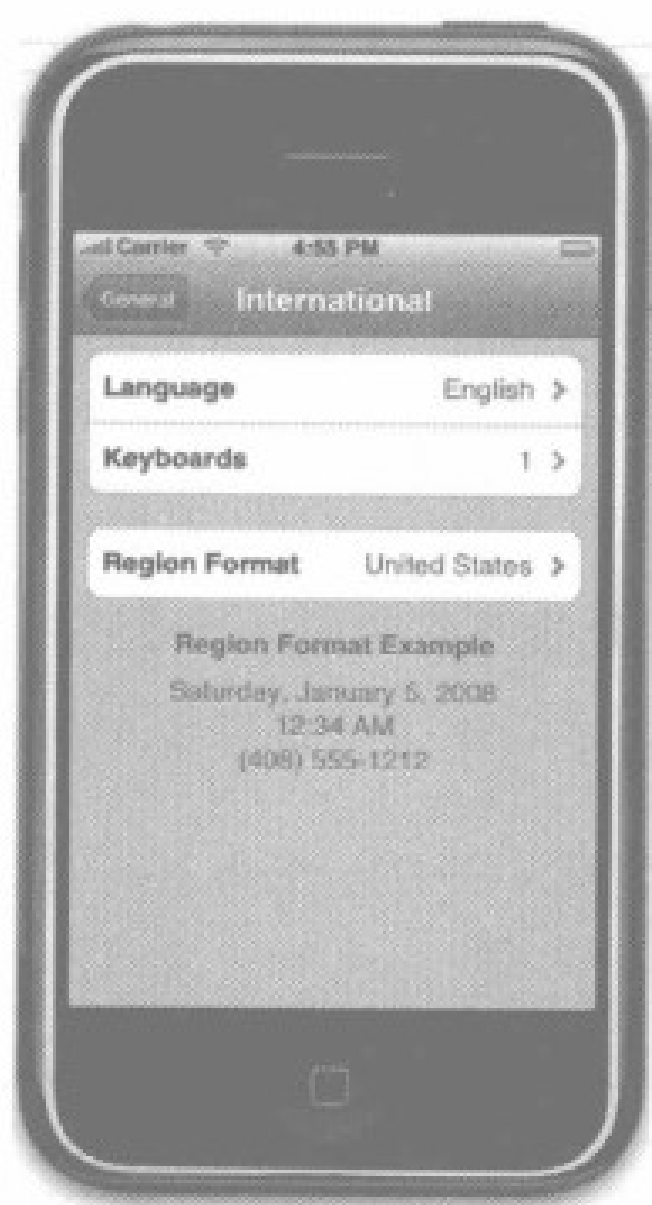


图17-7 更改语言和地区，这两项设置会影响用户的区域设置

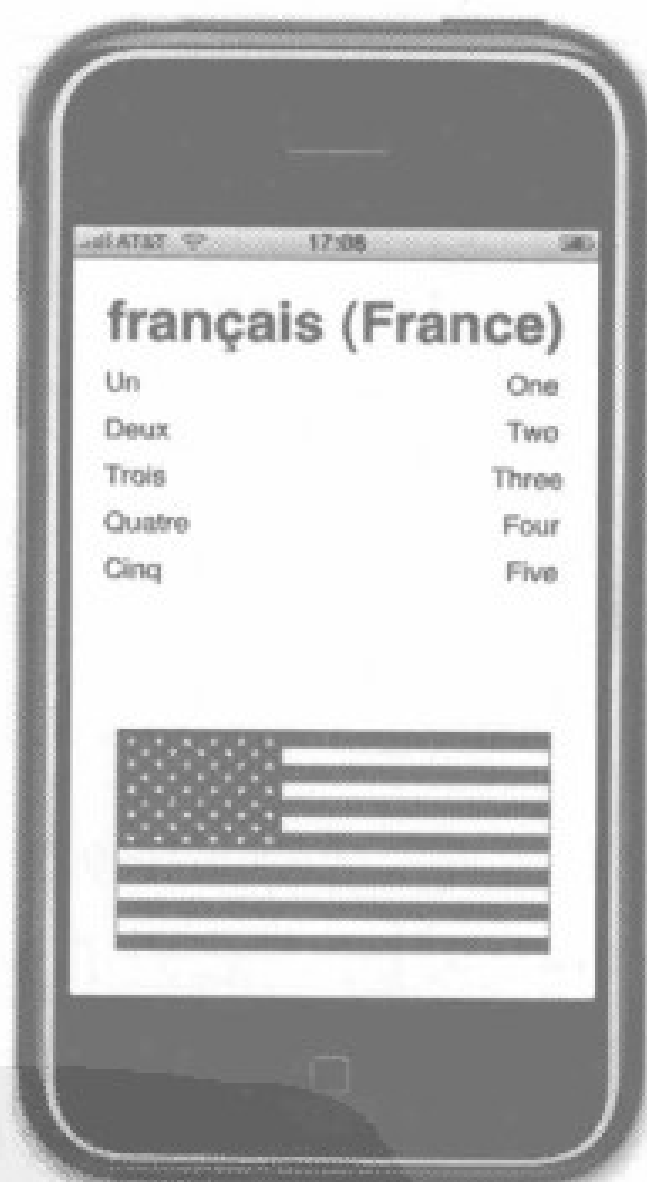


图17-8 现在应用程序已经部分翻译为法语

注意 iPhone和iPhone仿真器都隐藏了资源以提升性能。因此，有时需要手动移除旧版本的应用程序，进行清理及生成工作以显示本地化更改。在仿真器上有一个小技巧：退出仿真器。然后，在Xcode的Build菜单中选择Clean。然后，打开Finder，删除文件夹~/Library/Application Support/iPhone Simulator（此处~表示主目录）。现在从此书中打开一个旧项目，并在仿真器上运行。它启动之后，轻按主菜单按钮，启动Settings，将地区改为France，语言改为French。做出改之后，退出仿真器。最后，返回至LocalizeMe项目并在仿真器上运行。大功告成！

现在的问题是，国旗错了。通过在French本地化nib文件中选择一幅不同的图像即可以更改国旗图像。除此之外，也可以真正的本地化国旗本身。在本地化nib文件使用的一幅图像或其他资源时，nib文件会自动显示正确的语言版本（尽管在写作此书时还没有方言版本）。如果使用French版本本地化flag.png文件本身，那么在适当的时候，nib文件会自动显示正确的国旗。

17.3.5 本地化图像

现在开始对国旗图像进行本地化。首先在Xcode的Groups & Files窗格中单击flag.png。然后按 \mathbb{A} 1返回到此文件的Info窗口，单击Make File Localizable按钮。这样做可以让Xcode将flag.png复制到English.lproj文件夹（或是基础语言文件夹中，如果它们不同的话）。切换回General选项卡，并单击Add Localization按钮。在提示输入一种语言时，键入fr。此时，在LocalizeMe项目文件夹内的fr.lproj文件夹中应该有另一个文件，叫做flag.png，这是基础语言中flag.png文件的一个副本。显而易见，此图像并不正确。既然Xcode不允许编辑图像文件，那么在本地化项目内获取正确图像的最简便的方法是使用Finder将正确的图像复制到项目中。在17 LocalizeMe文件夹的Resources文件夹中，可以找到一个名为fr的文件夹。在此子文件夹中，可以找到一个flag.png文件，其中包含了法国国旗而不是美国国旗。将此flag.png文件复制到项目的fr.lproj子文件夹中，改写原文件。

工作完成了。因为iPhone可能会在上次运行应用程序时隐藏美国国旗，所以需要使用Xcode中的Organizer窗口将旧应用程序从iPhone中移除。如果是在仿真器上运行，则不必如此，只需直接跳到清理及生成步骤。

从Window菜单中选择Organizer，或按 $\wedge \mathbb{A} \mathbb{O}$ 打开它（如图17-9所示）。

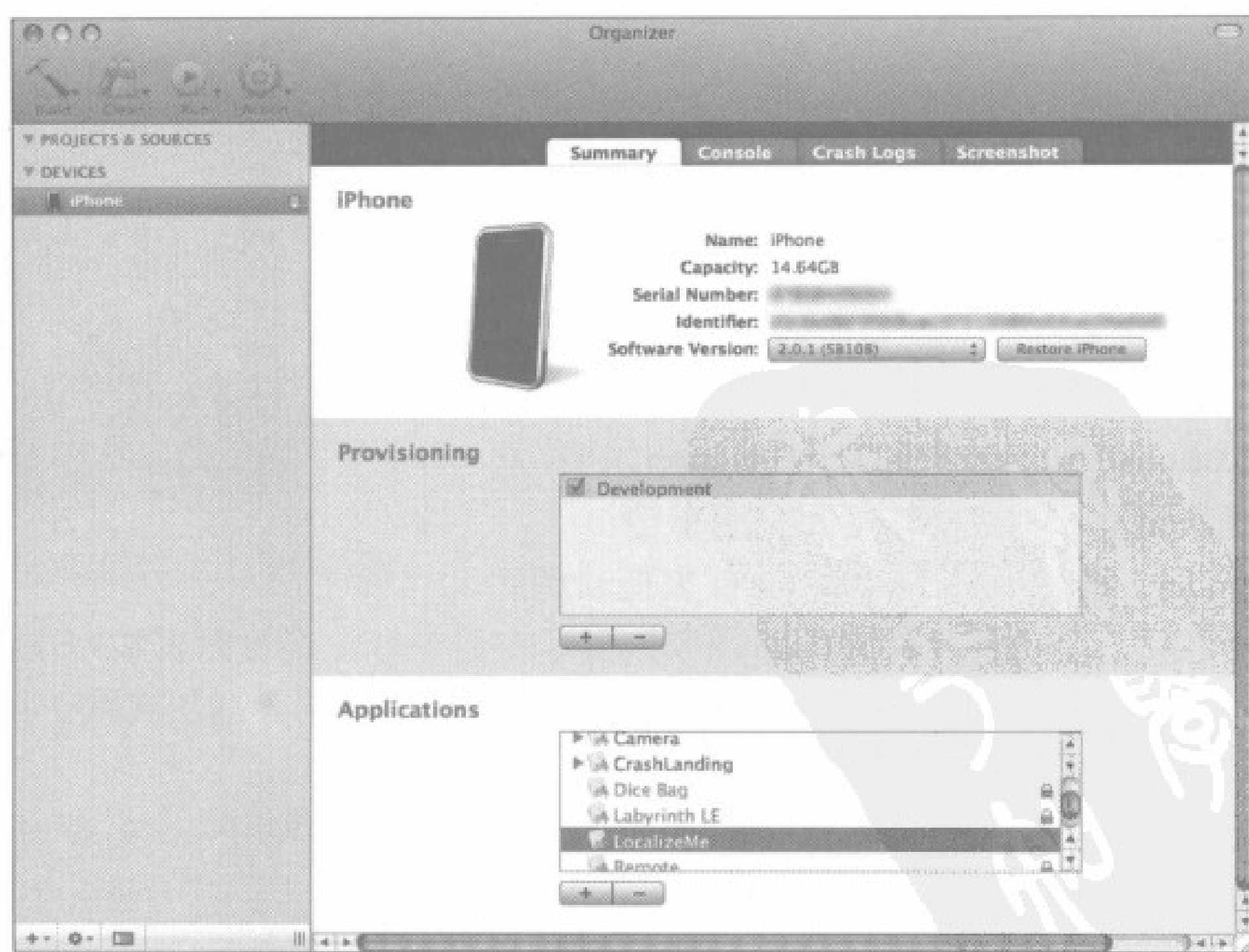


图17-9 使用Xcode的Organizer窗口可以手动移除应用程序

在Summary选项卡上,可以看到3个部分。最下面部分的标签是Applications。在应用程序列表中,找到LocalizeMe并选定它,然后单击减号按钮移除应用程序的旧版本和与之关联的隐藏程序。

现在从Build菜单中选择Clean,再次生成并运行此应用程序。启动应用程序后,除了下方左侧的法语单词之外,现在应该还显示了法国国旗,假设此时的手机或仿真器仍然设置为French(如图17-10所示)。

17.3.6 本地化应用程序图标

可以使用与本地化flag.png同样的方式来本地化应用程序的图标图像。首先单击Groups & Files窗格中的icon.png。打开Info窗口,并切换至General选项卡。单击Make Localizable按钮并返回至General选项卡。单击Add Localization按钮,当提示输入语言时,键入fr。

在17 LocalizeMe的Resources文件夹中的fr文件夹里(刚刚在此复制了flag.png文件),也可以找到本地化版本的icon.png。使用Finder将它复制到fr.lproj文件夹中,覆盖此处的原版本。现在,iPhone将自动探测并对法语用户显示此图标。

17.3.7 生成和本地化字符串文件

在图17-10中可以看到,视图右侧的单词仍然是英语。翻译它们需要先生成基础语言字符串文件,然后对它本地化。要完成这一任务,需要暂时脱离Xcode的温柔陷阱。

启动/Applications/Utilities/中的Terminal.app。当终端窗口打开时,键入cd及一个空格。不要按回车键。

现在打开Finder,将LocalizeMe项目文件夹拖动到终端窗口中。放置之后,至项目文件夹的路径应该显示在命令行上。这时再按回车键。

cd命令是“更改目录”的Unix说法,所以刚才所做的是将终端会话从其默认目录导航至项目目录。

下一步是运行程序genstring,使之查找Classes文件夹中.m文件中出现的所有NSString。为此,只需键入以下命令,并按回车键:

```
genstrings ./Classes/*.m
```



图17-10 现在已将图像和nib文件本地化

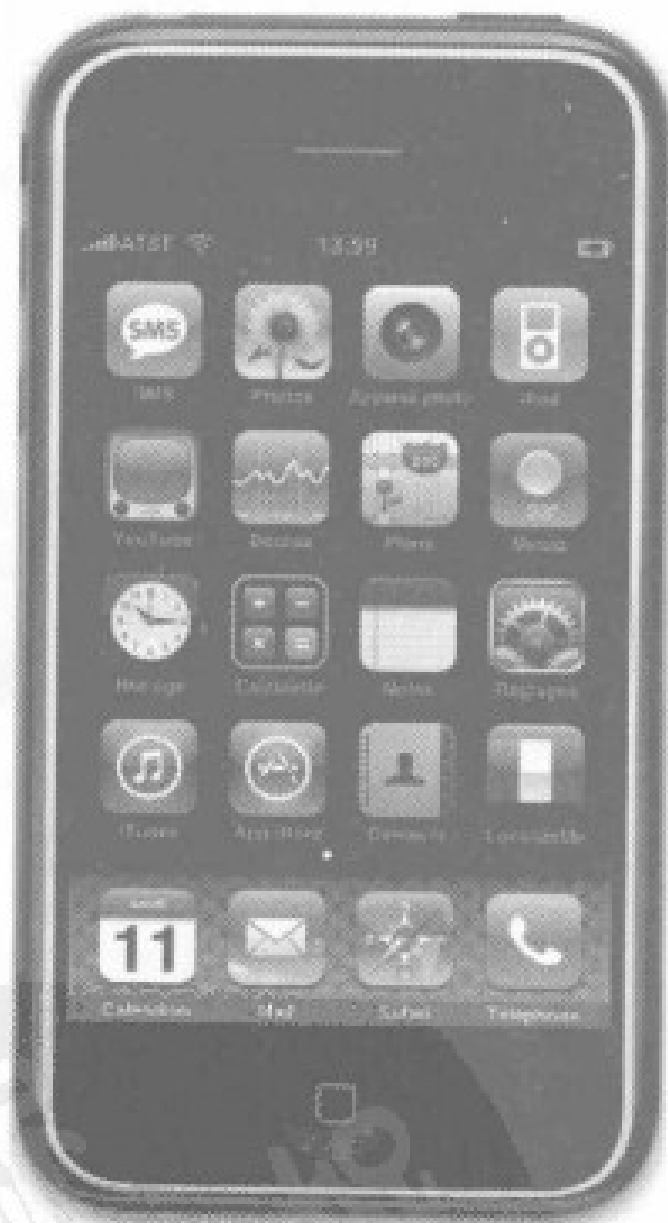


图17-11 已将应用程序图标本地化

命令执行完成之后（只需一秒钟），将返回至命令行。在Finder中，在项目文件夹中找到一个名为Localizable.strings的新文件。将它拖到Xcode中的Resources文件夹中，但在弹出提示时，先不要单击Add按钮。

提示 任何时间都可以安全地重新运行genstrings以重新创建基础语言文件。一般来说，不应该编辑基础语言文件。如果需要对任何用到的字符串做出更改，请在代码中查找它并在此处进行更改；然后使用genstrings重新生成Localizable.strings文件。

Localizable.strings文件采用UTF-16编码，这是Unicode的二字节版本。多数人可能使用UTF-8或当地语言编码方案作为Xcode中的默认编码。所以在项目中导入Localizable.strings文件时，需要考虑到这一点。首先，取消选中Copy items into destination group's folder (if needed)复选框，因为此文件已经在项目文件夹中。更重要的一点是，将文本编码更改为Unicode (UTF-16)（如图17-12所示）。如果不这样做的话，在Xcode中编辑此文件时，里面会显示乱码。

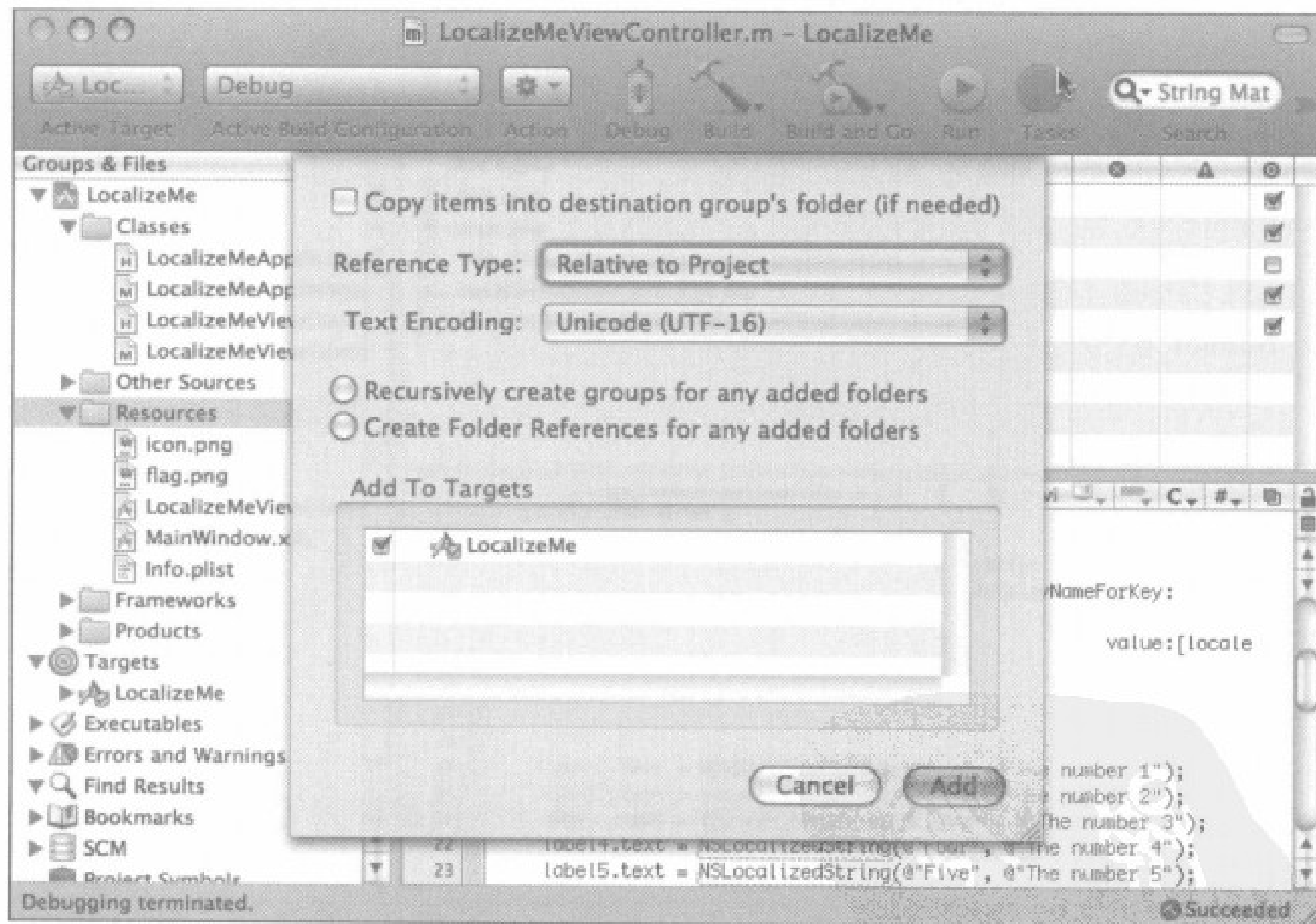


图17-12 导入Localizable.strings文件

然后单击Add按钮。导入文件之后，在Resources中单击Localizable.strings并查看它。它应该包含5个条目，因为我们对5个不同的值使用了5次NSLocalizedString。传递给第二个变量的值已经变成了各个字符串的注释。

字符串是以字母顺序生成的，这是一个好特性。因为在本例中处理的是数字，所以字母顺序并不是呈现它们的最直观的方式，但在多数情况下，按字母排序会很有帮助。


```
/* The number 5 */  
"Five" = "Five";
```

```
/* The number 4 */  
"Four" = "Four";
```

```
/* The number 1 */  
"One" = "One";
```

```
/* The number 3 */  
"Three" = "Three";
```

```
/* The number 2 */  
"Two" = "Two";
```

现在对它进行本地化。

首先单击Localizable.strings，并按⌘I打开Info窗口。如果切换到General选项卡，便可以发现其中的Make File Localizable按钮与本地化图像和nib文件时使用的按钮相同。单击此按钮。

切换回General选项卡并单击Add Localization。在提示输入语言时，键入fr表示对法语的所有方言进行本地化。返回至Xcode的Groups & Files窗格，单击Localizable.strings旁的展开三角形。单击fr，并在Xcode的编辑窗格中做如下更改：

```
/* The number 5 */  
"Five" = "Cinq";
```

```
/* The number 4 */  
"Four" = "Quatre";
```

```
/* The number 1 */  
"One" = "Un";
```

```
/* The number 3 */  
"Three" = "Trois";
```

```
/* The number 2 */  
"Two" = "Deux";
```

在现实生活中，通常是将此文件发送给翻译部门将这些值翻译到等号右边。在这个简单的示例中，我们可以自己翻译。

现在保存、编译并运行应用程序——应用程序已经完全本地化为法语。如果想做更多练习，可以使用17 LocalizeMe的Resources子文件夹中的信息进行德语本地化。如果想增加对此语言的支持，在此可以找到另一个icon.png、flag.png和Localizable.strings文件。

17.4 小结

若要让iPhone应用程序的销售最大化，则需尽可能地将它本地化。幸运的是，iPhone的本地化体系结构使应用程序可以轻松地支持多种语言，甚至是同一种语言的多种方言。如在本章中所

见，几乎添加到应用程序的任何类型的文件都可以按需要进行本地化。

如果不计划对应用程序本地化，那么请在代码中使用`NSString`而不是只使用静态字符串。因为Xcode的Code Sense属性，键入时间的不同可以忽略，所以随时都可以翻译应用程序，使得生活更轻松。

至此，整个iPhone旅程已经接近尾声。我们几乎同时到达了旅途的终点。在下一章之后，我们将说sayonara、au revoir、auf wiedersehen、avrio、arrivederci、adiós或再见。虽然现在你已经有了牢固的基础去构建有自己个性的iPhone应用程序，但是不要着急，还有一些有用的信息需要讲述。



至此，创建iPhone应用程序的旅程已经进入了尾声。通过阅读本书，你已经对iPhone有了全面的了解，但是谈到技术，尤其涉及到编程的层面，任何人永远都不可能做到全知。前面17章所使用的编程语言和框架是20多年不断发展的成果。并且，苹果公司的工程师们正在分秒必争地工作，思考下一个惊艳的崭新产品。iPhone平台刚刚进入繁盛期，未来将会有更多产品面世。

通过阅读本书，你可以为自己打下牢固的基础，并掌握扎实的知识——关于Objective-C、Cocoa Touch和能将这些技术融合在一起创建不可思议的新iPhone应用程序的工具。通过本书可以理解iPhone软件体系结构以及Cocoa Touch音乐播放的设计模式。简而言之，你已经可以为自己制订方针路线了。对此我们感到非常骄傲。那下一步呢？

18.1 答案揭晓

作为iPhone开发的核心，编程是用于解决问题和计算结果的。它很有趣，并且很值得去做。但有的时候，你可能会遇到一些看似不能克服的难题。

有时，在此类问题上少花一点时间反而可以得到答案。晚上良好的睡眠或抽出几个小时去做其他事通常就可以达到这样的效果。相信我们，你可能几个小时都盯着同一个问题，对其反复分析而使自己筋疲力尽，以至于错过了一个显而易见的解决方法。

然而有时候甚至峰回路转也毫无帮助。在这种情况下，站在高处的朋友会很有益处。下一节将概述一些有用的资源，在走投无路时，可以向它们求助。

18.1.1 苹果公司的文档

学会使用Xcode的文档浏览器。文档浏览器是一笔非常有价值的财富，其中包括示例源代码、概念指导、API参考、视频教程等等。苹果公司的文档几乎包括了所有可能遇到的问题。熟悉苹果公司的文档之后，便可以更轻松地去探索未知的领域和苹果公司开发的新技术。

说明 Xcode文档浏览器提供的信息与苹果公司 Developer Connection 网站 <http://developer.apple.com> 提供的信息相同。

18.1.2 邮件列表

可以登录并访问如下这些便捷的邮件列表。

<http://lists.apple.com/mailman/listinfo/cocoa-dev>，这份由苹果公司维护的列表主要是关于Mac OS X中的Cocoa的。然而，因为Cocoa和Cocoa Touch的传承性，此列表中的许多人仍然能够提供帮助。尽管如此，在提问之前一定要先搜索一下列表存档文件。

<http://lists.apple.com/mailman/listinfo/xcode-users>是另一份由苹果公司维护的列表，这份列表专门针对与Xcode相关的问题。

<http://lists.apple.com/mailman/listinfo/quartz-dev>，这份由苹果公司维护的邮件列表用于讨论Quartz 2D和Core Graphics技术。

18.1.3 论坛

将自己的问题发布到如下这些论坛上，与众多论坛读者共同交流。

<http://www.iphonedevsdk.com>，此网络论坛是iPhone编程者初学者和高手相互交流经验、共同解决问题的地方。

<http://discussions.apple.com/category.jspa?categoryID=164>，此链接可以连接到苹果公司的Mac和iPhone软件开发人员社区论坛。

<http://discussions.apple.com/category.jspa?categoryID=201>，此链接可以连接到苹果公司的iPhone社区论坛。

18.1.4 网站

访问如下这些网站可以获得编码建议。

<http://www.iphonedevbook.com>是本书的官方网站。随着人们向我们报告程序错误和错别字，我们会发布勘误表并维护所有书籍项目的最新版本。我们也会在上面说明最近的工作和成果。

<http://www.cocoadevcentral.com>可以链接到许多与Cocoa相关的网站和教程。

<http://cocoaheads.org> CocoaHeads是一个专注于支持和促进Cocoa的团体。它主要关注举行例会的当地团体，在例会上，Cocoa开发人员可以聚在一起，互帮互助，甚至结识朋友。没有什么比获得真实个人帮助更好的事情了，所以如果当地有CocoaHeads团体，请积极参与其中。如果没有的话，为什么不发起一个呢？

<http://nscodernight.com> NSCoder Nights是按周组织的会议，在此会议中，Cocoa编程人员可以聚在一起交流经验。与CocoaHeads会议相似，NSCoder Nights是独立组织的当地活动。

<http://cocoablogs.com>包含许多与Cocoa编程相关的博客链接。

<http://www.iphonedevcentral.org>专用于提供iPhone编程教程。

18.1.5 博客

如果仍然没有找到编码难题的解决方案，那么可以尝试阅读如下这些博客。

<http://theocacao.com>由Scott Stevenson维护，他是一位经验颇丰的Cocoa程序员。

<http://www.wilshipley.com/blog/>，Wil Shipley是世界上最有经验的Objective-C程序员之一。任何一位Objective-C程序员都应该拜读他的Pimp My Code系列博文。

<http://rentzsch.com>，Wolf Rentzsch是一位经验丰富的独立Cocoa程序员，并且是C4独立开发人员会议的创始人。

<http://chanson.livejournal.com>，Chris Hanson效力于苹果公司的Xcode团队，他的博客里充满了远见卓识和有关Xcode及相关主题的内容。

18.1.6 如果仍未解决问题

给Dave和Jeff发送电子邮件，地址为daveandjeff@iphonedevbook.com。关于本书的排版错误和程序问题，也应该给此地址发送电子邮件。我们不能保证对于每封邮件都给予回复，但一定会阅读所有邮件。请确认在点击发送之前已经阅读了勘误表。并且一定记得告诉我们你自己所开发了什么好的应用程序。

18.2 再会

感谢你陪伴我们走过整个旅程。祝你好运，并希望能像我们一样享受iPhone编程的乐趣。



[G e n e r a l I n f o r m a t i o n]

书名 = i P h o n e 开发基础教程

丛书名 = 图灵程序设计丛书

作者 = (美) D a v e M a r k , J e f f L a M a r c h e 著

出版社 = 人民邮电出版社

出版日期 = 2 0 0 9 . 0 4

形态项 = 3 9 2

页数 = 3 9 2

原书定价 = 6 5 . 0 0

读秀号 = 0 0 0 0 0 6 6 9 9 4 0 4

S S 号 = 1 2 1 9 4 3 9 6

I S B N = 9 7 8 - 7 - 1 1 5 - 1 9 7 3 3 - 7 / T N 9 2 9 . 5 3

分类号 = 1 8 1 6 1 1 1 2 0 3

主题词 =

参考文献格式 = (美) D a v e M a r k , J e f f L a M a r c h e
著 . i P h o n e 开发基础教程 . 人民邮电出版社 , 2 0 0 9 . 0 4 .

简介 = i P h o n e 是一种全新的移动平台 , 苹果公司为它推出了强大的软件开发工具包 i P h o n e S D K 。本书是一部关于 i P h o n e S D K 和 i P h o n e 开发的基础教程 , 内容翔实、语言生动。书中结合消费类设备上常见的实例 , 循序渐进地讲解了 i P h o n e 开发的基本流程 , 并介绍了最先进、时尚、受欢迎的 i P h o n e