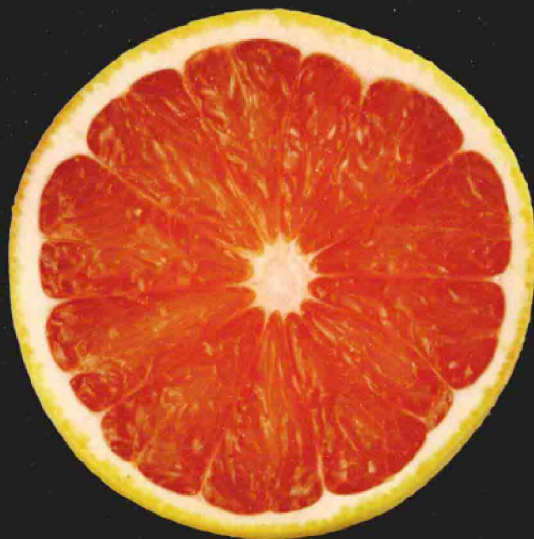


中文版累计销量逾50 000册!

全球数百万iOS开发者交口称赞的iOS开发圣经!



# 精通iOS开发

(第6版)

[瑞典] Jack Nutting [法] Fredrik Olsson  
[美] David Mark [美] Jeff LaMarche

著

周庆成 邓强 武海峰 译



人民邮电出版社

POSTS & TELECOM PRESS

## 亚马逊读者评论

“这本书简直太棒了！是我买过的性价比最高的一本书。作者非常清楚地解释了做出每种选择的原因以及每个iOS组件的特性，从而教你一种思维方式，而不是简单地完成每项任务。”

Beginning iOS 7 Development: Exploring the iOS SDK

# 精通iOS开发 (第6版)

还在iPhone和iPad应用开发的大门徘徊？还是已经投身iOS应用开发阵营，但希望迅速提升自己的功力？选择本书绝对能令你惊喜连连。四位作者均是资深移动开发专家，具有丰富的Mac、iOS、Cocoa及Objective-C开发经验。作者将多年的实战经验与智慧感悟汇集成本书，旨在帮助没有经验的读者顺利叩开iOS应用开发的大门，帮助有经验的读者迅速提升功力，从而在iOS开发的道路上所向披靡。

本书自问世以来就受到读者的交口赞誉，被奉为学习iOS平台开发的不二之选。中文版累计销量已超过5万册。新版做了大幅修订，力求使新老读者都能有最大收获。作者重写了所有项目代码，使之兼容新旧SDK，并对原有的各章内容进行更新，从而反映出技术的最新发展动态。

还等什么？立即展卷阅读，加入iOS开发的行列吧！

图灵iOS相关图书阅读路线图



Apress®

图灵社区: iTuring.cn

热线: (010)51095186转600

分类建议 计算机/移动开发

人民邮电出版社网址: www.ptpress.com.cn



ISBN 978-7-115-36826-3



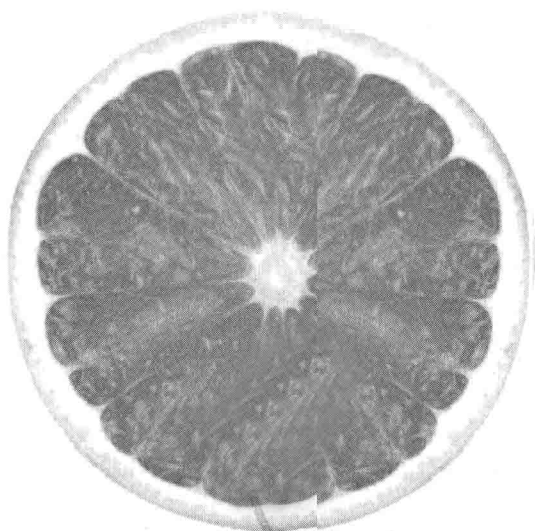
ISBN 978-7-115-36826-3

定价: 109.00元



**TURING**

图灵程序设计丛书



# 精通iOS开发

(第6版)

[瑞典] Jack Nutting [法] Fredrik Olsson 著  
[美] David Mark [美] Jeff LaMarche

周庆成 邓强 武海峰 译

人民邮电出版社

北京

## 图书在版编目 (C I P) 数据

精通iOS开发：第6版 / (瑞典) 纳丁 (Nutting, J.)  
等著；周庆成，邓强，武海峰译. — 北京：人民邮电  
出版社，2014.10

(图灵程序设计丛书)

ISBN 978-7-115-36826-3

I. ①精… II. ①纳… ②周… ③邓… ④武… III.  
①移动电话机—应用程序—程序设计 IV. ①TN929.53

中国版本图书馆CIP数据核字(2014)第191738号

## 内 容 提 要

本书是 iOS 应用开发基础教程，内容翔实，语言生动。作者结合大量实例，循序渐进地讲解了适用于 iPhone/iPad 开发的基本流程。新版介绍强大的 iOS 7 操作系统，涵盖 Xcode 4 以来的新功能，书中所有案例全部重新编写。

本书具有较强通用性，iOS 开发新手可通过本书快速入门进阶，经验丰富的 iOS 开发人员也能从中找到令人耳目一新的内容。

- 
- ◆ 著 [瑞典] Jack Nutting [法] Fredrik Olsson  
[美] David Mark [美] Jeff LaMarche  
译 周庆成 邓 强 武海峰  
责任编辑 朱 巍  
执行编辑 李 鑫  
责任印制 焦志炜
- ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号  
邮编 100164 电子邮件 315@ptpress.com.cn  
网址 <http://www.ptpress.com.cn>  
三河市中晟雅豪印务有限公司印刷
- ◆ 开本：800×1000 1/16  
印张：34.25  
字数：809千字 2014年10月第1版  
印数：1-3 000册 2014年10月河北第1次印刷
- 著作权合同登记号 图字：01-2014-5227号

---

定价：109.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

# 版 权 声 明

Original English language edition, entitled *Benginning iOS 7 Development: Exploring the iOS SDK* by Jack Nutting, Fredrik Olsson, David Mark, Jeff LaMarche, published by Apress, 2855 Telegraph Avenue, Suite600, Berkeley, CA94705 USA.

Copyright © 2014 by Jack Nutting, Fredrik Olsson, David Mark and Jeff LaMarche.Simplified Chinese-language edition copyright © 2014 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。

# 译者序

iOS 7 是 iOS 系统自 2007 年 iPhone 发布以来最大一次升级，由苹果公司在 2013 年 6 月 10 日举办的全球开发者大会（WWDC）上对外公布，其支持 iPhone 4 及以后机型、第 5 代 iPod touch、iPad 2 及以后机型，等等。

iOS 7 重新设计了用户界面并对操作系统的功能进行了改进，它采用全新的扁平化界面设计，总计有上百项改动，其中包括控制中心、通知中心、多任务处理能力等。iOS 7 还特别针对中国用户的输入习惯改进了中文输入法，增加了九宫格输入方式。就在翻译此书时，iOS 7 的安装率已经达到 91%，成为 iOS 设备上主流的操作系统。

iOS 7 在用户界面上有着与之前版本完全不同的视觉设计，应用程序的图标变得更锐利，在许多地方采用了较为纤细的字体，以往的拟物风格在 iOS 7 中不再出现。虽然扁平化风格界面并非苹果最早采用，但在视觉体验上，iOS 7 依然拥有自己的特色。新的界面色彩更为鲜艳，饱和度更加突出，通过渐变和半透明更好地阐释了苹果一贯的简单至上的设计原则。iOS 7 的画面采用类 3D 的效果，在锁定画面及主屏幕会有立体的效果。所有的内置应用程序、解锁画面与通知中心也经过重新设计。

此外 iOS 7 也新增了控制中心界面，让用户能够快速控制各种系统功能的开关（包括飞行模式、蓝牙、Wi-Fi 网络等）并调整屏幕亮度、播放或暂停音乐等。后台多任务处理功能也经过了强化，现在能够支持每一种应用程序，切换程序时也采用了新的交互方式。

本书全面介绍了 iOS 7 开发的基础知识，结构清晰，内容丰富，通过 22 章由浅入深涵盖了 iOS 7 所有的关键内容。与上一版相比，本书增加了游戏开发 SDK 教程，并对许多章节进行了大量的优化改进，其中第 9 章为了配合 iOS 7 的新特性重写了全部内容，从而能够更好地展示使用新系统开发的优势。

随着 iOS 的不断升级，这本书也迎来了第 6 版。我曾阅读过从本书最初到现在的所有版本，学习过程中难免会遇到一些翻译失当或难以理解的地方，也向图灵公司提出了一些勘误的意见。这次新版出来时，我向出版社自荐来改善这本书一些遗留的历史问题，希望让读者可以更轻松地理解原作者的想法，简化程序员向移动开发入门的难度。虽然在校订过程中修改了很多细节，尽可能地保证质量，但仍可能会有遗漏，希望读者能给出合理的意见。

这本书能够顺利翻译完成，首先要感谢我的家人对我的支持，以及出版社对我的信任。还要特别感谢图灵公司的编辑李鑫，他认真地对全稿进行了审阅。此外还要感谢好友蒋振华，他对书中许多内容提出了自己的看法。当然，还要感谢苹果公司所提供的如此便利的开发平台以及原作者的辛勤劳动。

——周庆成



# 关于本书

本书内容非常丰富。如果你读的是纸质书，就能明白我的意思，如果买的是电子书，就不容易发现这点。我只想说，它比我见过的各种经典书籍还要厚。当然，这并非是说，我是经典书籍或某方面的行家，但你们只需要明白它很大很重就对了。如果你想知道这本书是否比面包箱还大，我可以明确地告诉你，差不多真的有那么大。然而，本书结构划分得条理明确，各部分内容也十分简洁扼要，你完全可以在某个愉快的下午尽情揣摩某一章节的内容。这种“揣摩”并不仅仅指的是通过阅读来了解每章内容，还需要在你的 Mac 电脑上，用自己的方式来学习与实践本书的知识。如果在阅读每一章时，能够构建其中所有的示例应用，则有助于加深对使用模式和相关概念的理解，而光靠阅读显然是无法做到这一点的。如果完整通读此书，就会对 iOS 应用程序开发的基本概念有很好的理解，也就可以更好地构建自己的 iOS 应用了。

许多年前，我曾与 Torfrid Olsson（已故）见过一面，他是一位瑞典北方农村的雕刻家。对于他生活的某一方面，我表示羡慕和钦佩，但他的回答也令我不禁深思：“嗯，这些只是书上讲的。每个人的生活都是独特的，你怎么会认为你的生活是残缺的呢？”我希望你要善于利用书中的知识，不要满足于读懂。阅读当然是必要的，但也要实践和理解其中的内容，甚至要敢于质疑或推翻其中的内容，真正掌握这些知识。

——Jack Nutting  
斯德哥尔摩，2014 年

# 致 谢

感谢 Apress 所有员工，正是由于他们的努力工作，才能完成这本如此厚实的书。他们的付出与帮助是至关重要的。尤其需要感谢 Anamika Panchoo，直到出版的最后阶段，她都在鉴定全书插图的正确性——要知道，当时她已经不用负责这方面工作了，大部分工作的负责人是 Melissa Maldonado。没有这些编辑勤奋耐心的校正，本书是不可能完成的。Patrick Meader 修改了大量的文法错误，而 Nick Waynik 则把杂乱的代码排版得井井有条。Tom Welsh、Douglas Pundick 和 Matt Moodie 三人轮流担当执行编辑，确保了每章内容条理分明，易于理解。当然，对于负责编排索引、页面排版、印刷零售（以及很多我不知道的工作）的工作人员来说，尽管你们经常被人忽视，我们也要衷心对你们表达莫大的感激之情。尽管身为作者，我们没能和你们直接接触过，甚至也不知道你们的名字，但我明白你们做的事都非常有意义，再接再厉！

# 目 录

第 1 章 欢迎来到 iOS 世界 .....	1	第 3 章 实现基本交互 .....	34
1.1 关于本书 .....	1	3.1 MVC 方法 .....	34
1.2 必要条件 .....	1	3.2 创建项目 .....	35
1.2.1 开发者选项 .....	3	3.3 视图控制器 .....	36
1.2.2 必备知识 .....	3	3.3.1 输出接口和操作方法 .....	37
1.3 编写 iOS 应用程序有何不同 .....	4	3.3.2 清理视图控制器 .....	40
1.3.1 只能同时运行一个应用 .....	4	3.3.3 设计用户界面 .....	40
1.3.2 只有一个窗口 .....	4	3.3.4 运行应用 .....	50
1.3.3 有限的访问权限 .....	5	3.3.5 样式文本 .....	50
1.3.4 有限的响应时间 .....	5	3.4 应用程序委托 .....	51
1.3.5 有限的屏幕大小 .....	5	3.5 小结 .....	54
1.3.6 有限的系统资源 .....	5	第 4 章 更丰富的用户界面 .....	55
1.3.7 不支持垃圾回收 .....	6	4.1 满是控件的屏幕 .....	55
1.3.8 新功能 .....	6	4.2 活跃控件、静态控件和被动控件 .....	57
1.3.9 与众不同的交互方法 .....	6	4.3 创建应用程序 .....	58
1.4 本书内容 .....	7	4.4 实现图像视图和文本框 .....	59
1.5 这一版的新内容 .....	8	4.4.1 添加图像视图 .....	59
1.6 准备开始吧 .....	9	4.4.2 调整图像视图的大小 .....	62
第 2 章 创建项目 .....	10	4.4.3 设置视图属性 .....	63
2.1 在 Xcode 中创建项目 .....	10	4.4.4 添加文本框 .....	65
2.1.1 Xcode 项目窗口 .....	14	4.4.5 创建并关联输出接口 .....	71
2.1.2 深入研究项目 .....	22	4.5 关闭键盘 .....	73
2.2 界面构建器简介 .....	23	4.5.1 输入完成后关闭键盘 .....	73
2.2.1 文件格式 .....	24	4.5.2 通过触摸背景关闭键盘 .....	74
2.2.2 分镜 .....	25	4.5.3 添加滑动条和标签 .....	77
2.2.3 库 .....	26	4.5.4 添加顶部约束 .....	78
2.2.4 在视图中添加标签 .....	27	4.5.5 创建并关联操作方法和输出接口 .....	79
2.2.5 属性修改 .....	29	4.5.6 实现操作方法 .....	79
2.3 画龙点睛——美化 iPhone 应用 .....	31	4.6 实现开关、按钮和分段控件 .....	80
2.4 小结 .....	33		

4.6.1 添加两个带标签的开关 .....	81	6.3.3 修改 BIDSwitchView Controller.m .....	120
4.6.2 为开关创建并关联输出接口和 操作方法 .....	83	6.3.4 添加视图控制器 .....	120
4.6.3 实现开关的操作方法 .....	83	6.3.5 构建带有工具栏的视图 .....	122
4.7 美化按钮 .....	85	6.3.6 编写根视图控制器 .....	124
4.7.1 可拉伸图像 .....	85	6.3.7 实现内容视图 .....	127
4.7.2 控件状态 .....	86	6.3.8 转换过程的动画效果 .....	130
4.7.3 为按钮创建并关联输出接口和 操作方法 .....	87	6.4 小结 .....	132
4.8 实现分段控件的操作方法 .....	87	第 7 章 分页栏与选取器 .....	133
4.9 实现操作表单和警告视图 .....	88	7.1 Pickers 应用程序 .....	134
4.9.1 遵从操作表单委托方法 .....	88	7.2 委托和数据源 .....	136
4.9.2 显示操作表单 .....	89	7.3 创建 Pickers 应用程序 .....	136
4.9.3 最终调整 .....	92	7.3.1 创建视图控制器 .....	136
4.10 小结 .....	92	7.3.2 添加分镜 .....	137
第 5 章 自动旋转和自动调整大小 .....	93	7.3.3 创建分页栏控制器 .....	138
5.1 自动旋转机制 .....	93	7.3.4 初次运行 .....	140
5.1.1 点、像素和 Retina 显示屏 .....	94	7.4 实现日期选取器 .....	141
5.1.2 自动旋转的实现方式 .....	95	7.5 实现单滚轮选取器 .....	144
5.2 选择视图支持的方向 .....	95	7.5.1 构建视图 .....	144
5.2.1 应用级支持的方向 .....	96	7.5.2 将控制器实现为数据源和 委托 .....	145
5.2.2 单个控制器的旋转支持 .....	97	7.6 实现多滚轮取器 .....	149
5.3 使用约束设计界面 .....	99	7.6.1 声明输出接口和操作方法 .....	149
5.3.1 覆盖默认的约束 .....	101	7.6.2 构建视图 .....	149
5.3.2 与屏幕等宽的标签 .....	102	7.6.3 实现控制器 .....	150
5.4 旋转时重构视图 .....	104	7.7 实现内容取决于滚轮 .....	152
5.4.1 创建并关联输出接口 .....	106	7.8 使用自定义选取器创建一个简单 游戏 .....	158
5.4.2 旋转时移动按钮 .....	106	7.8.1 编写控制器头文件 .....	158
5.5 小结 .....	109	7.8.2 构建视图 .....	159
第 6 章 多视图应用 .....	110	7.8.3 添加图像资源 .....	160
6.1 多视图应用的常见类型 .....	110	7.8.4 实现控制器 .....	160
6.2 多视图应用的体系结构 .....	113	7.8.5 最后的细节 .....	163
6.2.1 根控制器 .....	115	7.9 小结 .....	166
6.2.2 内容视图剖析 .....	116	第 8 章 表视图简介 .....	167
6.3 构建 View Switcher 项目 .....	116	8.1 表视图基础 .....	167
6.3.1 创建视图控制器和分镜 .....	117	8.1.1 表视图和表视图单元 .....	168
6.3.2 修改应用委托 .....	119	8.1.2 分组表和无格式表 .....	168



8.2 实现一个简单表 .....	170	9.7 创建字体信息视图控制器 .....	225
8.2.1 设计视图 .....	170	9.7.1 设计字体信息视图控制器的 分镜 .....	226
8.2.2 编写控制器 .....	171	9.7.2 设置约束 .....	227
8.2.3 添加一个图像 .....	175	9.7.3 调整字体列表视图控制器的 转场 .....	228
8.2.4 表视图单元样式 .....	176	9.7.4 我的收藏字体 .....	229
8.2.5 设置缩进级别 .....	178	9.8 改善表视图 .....	229
8.2.6 处理行的选择 .....	179	9.8.1 实现轻扫删除 .....	230
8.2.7 更改字体大小和行高 .....	180	9.8.2 实现拖动排序 .....	231
8.3 定制表视图单元 .....	182	9.9 小结 .....	232
8.3.1 向表视图单元添加子视图 .....	182	第 10 章 集合视图 .....	233
8.3.2 创建 UITableViewCell 子类 .....	183	10.1 创建 DialogViewer 项目 .....	233
8.3.3 从 nib 文件加载 UITableViewCell .....	187	10.2 修补视图控制器类 .....	234
8.4 分组分区和索引分区 .....	191	10.3 自定义单元 .....	235
8.4.1 构建视图 .....	191	10.4 配置视图控制器 .....	237
8.4.2 导入数据 .....	191	10.5 内容单元 .....	239
8.4.3 实现控制器 .....	192	10.6 实现流式布局 .....	240
8.4.4 添加索引 .....	196	10.7 分区标题视图 .....	241
8.5 解决状态栏干扰 .....	197	10.8 小结 .....	243
8.6 实现搜索栏 .....	198	第 11 章 iPad 开发注意事项 .....	244
8.7 小结 .....	203	11.1 分割视图和浮动窗口 .....	244
第 9 章 导航控制器和表视图 .....	204	11.1.1 创建 SplitView 项目 .....	246
9.1 导航控制器 .....	204	11.1.2 在分镜中定义结构 .....	248
9.1.1 栈的概念 .....	204	11.1.3 使用代码定义功能 .....	250
9.1.2 控制器栈 .....	205	11.2 显示总统信息 .....	256
9.2 一个简单的字体浏览器: Fonts .....	206	11.3 创建浮动窗口 .....	261
9.2.1 子控制器 .....	207	11.4 小结 .....	267
9.2.2 Font 应用的基础框架 .....	209	第 12 章 应用设置及用户默认设置 .....	268
9.3 创建根视图控制器 .....	213	12.1 设置捆绑包入门 .....	268
9.4 初始化分镜 .....	216	12.2 应用: Bridge Control .....	269
9.5 第一个子控制器: 字体列表视图 .....	218	12.2.1 创建项目 .....	272
9.5.1 设定字体列表的分镜 .....	220	12.2.2 使用设置捆绑包 .....	273
9.5.2 对根视图控制器的转场进行 设置 .....	221	12.2.3 读取应用中的设置 .....	286
9.6 创建字体尺寸视图控制器 .....	222	12.2.4 在应用中修改默认设置 .....	289
9.6.1 设计字体尺寸视图控制器的 分镜 .....	224	12.2.5 注册默认值 .....	292
9.6.2 对字体列表视图控制器的转 场进行设置 .....	224	12.2.6 保证设置有效 .....	293
		12.3 小结 .....	295

第 13 章 数据持久化基础知识 .....	296	第 15 章 Grand Central Dispatch 和 后台处理 .....	359
13.1 应用的沙盒 .....	296	15.1 Grand Central Dispatch .....	359
13.1.1 获取 Documents 目录 .....	297	15.2 SlowWorker 简介 .....	360
13.1.2 获取 tmp 目录 .....	298	15.3 线程基础知识 .....	362
13.2 文件保存方案 .....	298	15.4 工作单元 .....	363
13.2.1 单文件持久化 .....	299	15.5 GCD: 底层队列 .....	364
13.2.2 多文件持久化 .....	299	15.5.1 傻瓜式操作 .....	364
13.3 属性列表 .....	299	15.5.2 改进 SlowWorker .....	365
13.3.1 属性列表序列化 .....	299	15.6 后台处理 .....	370
13.3.2 Persistence 应用的第一个 版本 .....	300	15.6.1 应用生命周期 .....	371
13.4 对模型对象进行归档 .....	305	15.6.2 状态更改通知 .....	372
13.4.1 遵循 NSCoding 协议 .....	305	15.6.3 创建 State Lab 项目 .....	373
13.4.2 实现 NSCopying 协议 .....	306	15.6.4 执行状态的变化 .....	374
13.4.3 对数据对象进行归档和取消 归档 .....	307	15.6.5 利用执行状态更改 .....	376
13.4.4 归档应用 .....	308	15.6.6 处理不活跃状态 .....	377
13.5 使用 iOS 内嵌的 SQLite3 .....	311	15.6.7 处理后台状态 .....	380
13.5.1 创建或打开数据库 .....	312	15.7 小结 .....	387
13.5.2 绑定变量 .....	313	第 16 章 使用 Core Graphics 绘图 .....	388
13.5.3 SQLite3 应用 .....	314	16.1 Quartz 2D 基础概念 .....	388
13.6 使用 Core Data .....	320	16.2 Quartz 2D 绘图方法 .....	388
13.6.1 实体和托管对象 .....	321	16.2.1 Quartz 2D 图形环境 .....	389
13.6.2 Core Data 应用 .....	324	16.2.2 坐标系统 .....	390
13.7 小结 .....	333	16.2.3 指定颜色 .....	391
第 14 章 iCloud 之旅 .....	334	16.2.4 在环境中绘制图像 .....	393
14.1 使用 UIDocument 管理文档存储 .....	334	16.2.5 绘制形状: 多边形、直线 和曲线 .....	393
14.1.1 构建 TinyPix .....	335	16.2.6 Quartz 2D 样例: 图案、渐 变色、虚线图 .....	394
14.1.2 创建 BIDTinyPixDocument 类 .....	336	16.3 QuartzFun 应用程序 .....	394
14.1.3 主控制器代码 .....	338	16.3.1 构建 QuartzFun 应用程序 .....	395
14.1.4 初始分镜 .....	345	16.3.2 添加 Quartz 2D 绘制代码 .....	404
14.1.5 创建 BIDTinyPixView 类 .....	347	16.3.3 优化 QuartzFun 应用程序 .....	409
14.1.6 设计分镜 .....	351	16.4 小结 .....	412
14.2 添加 iCloud 支持 .....	353	第 17 章 Sprite Kit 游戏框架 .....	413
14.2.1 创建授权文件 .....	353	17.1 基础入门 .....	413
14.2.2 如何查询 .....	354	17.1.1 自定义初始场景 .....	414
14.2.3 保存在哪里 .....	356	17.1.2 隐藏状态栏 .....	415
14.2.4 将首选项保存到 iCloud .....	357		
14.3 小结 .....	358		

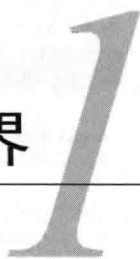
17.2 场景设置	415	18.6 Swipes 应用	454
17.3 玩家动作	418	18.6.1 自动手势识别	457
17.3.1 向场景中插入玩家	418	18.6.2 实现多指轻扫	459
17.3.2 触摸处理	419	18.7 检测多次轻点	460
17.3.3 玩家移动	420	18.8 检测捏合和旋转	465
17.3.4 几何运算	421	18.9 自定义手势	469
17.3.5 轻微摆动	421	18.9.1 CheckPlease 应用	470
17.4 创建你的敌人	422	18.9.2 CheckPlease 的触摸方法	472
17.5 在场景中放入敌人	423	18.10 小结	474
17.6 开始射击	425	第 19 章 Core Location 和 Map Kit	475
17.6.1 定义物理类别	425	19.1 位置管理器	476
17.6.2 创建 BIDBulletNode 类	425	19.1.1 设置精度	476
17.6.3 应用物理知识	427	19.1.2 设置距离筛选器	476
17.6.4 在场景中添加导弹	427	19.1.3 启动位置管理器	477
17.7 利用物理引擎攻击敌人	429	19.1.4 合理使用位置管理器	477
17.8 实现关卡	431	19.2 位置管理器委托	477
17.8.1 注意敌人	431	19.2.1 获取位置更新	477
17.8.2 进入下一关卡	431	19.2.2 使用 CLLocation 获取纬度和经度	477
17.9 自定义碰撞	433	19.2.3 错误通知	479
17.9.1 为 SKNode 添加类别	434	19.3 开始构建 Core Location	480
17.9.2 向敌人添加自定义碰撞行为	435	19.4 将移动路线展现在地图上	484
17.9.3 准确显示玩家生命	436	19.5 小结	488
17.10 粒子系统	437	第 20 章 陀螺仪和加速计	489
17.10.1 第一个粒子	437	20.1 加速计物理特性	489
17.10.2 向场景中加入粒子	440	20.2 陀螺仪旋转特性	490
17.11 游戏结束	441	20.3 Core Motion 和动作管理器	490
17.12 开始场景	443	20.3.1 基于事件的动作	491
17.13 播放音乐	445	20.3.2 主动动作访问	496
17.14 小结	446	20.3.3 加速计结果	498
第 18 章 轻点、触摸和手势	447	20.4 检测摇动	498
18.1 多点触控术语	447	20.4.1 内嵌的摇动检测	499
18.2 响应者链	448	20.4.2 摇动与击碎	500
18.2.1 响应事件	448	20.5 将加速计用做方向控制器	502
18.2.2 转发事件: 保持响应者链的活动状态	449	20.5.1 滚弹珠程序	503
18.3 多点触控体系结构	450	20.5.2 实现 BIDBallView 类	505
18.4 4 个手势通知方法	450	20.5.3 计算弹珠运动	508
18.5 TouchExplorer 应用	451	20.6 小结	510

第 21 章 摄像头和照片库.....	511	22.2 字符串文件.....	521
21.1 图像选取器和 UIImagePickerController- Controller.....	511	22.2.1 字符串文件.....	522
21.2 实现图像选取器控制器委托.....	513	22.2.2 本地化的字符串宏.....	522
21.3 实际测试摄像头和照片库.....	514	22.3 现实中的 iOS 本地化应用.....	523
21.3.1 设计界面.....	515	22.3.1 创建 LocalizeMe.....	523
21.3.2 实现摄像头视图控制器.....	515	22.3.2 测试 LocalizeMe.....	526
21.4 小结.....	519	22.3.3 本地化项目.....	527
第 22 章 应用本地化.....	520	22.3.4 初始化分镜.....	530
22.1 本地化体系结构.....	520	22.3.5 创建并本地化字符串文件.....	532
		22.3.6 应用显示名称的本地化.....	535
		22.4 小结.....	536



## 第 1 章

# 欢迎来到 iOS 世界



这么说，你想编写 iPhone、iPod touch 和 iPad 应用程序？哦，当然可以。作为所有苹果设备的核心软件，iOS 从 2007 年发布以来发展异常迅速，是个极具吸引力的平台。移动应用平台的崛起意味着人们无时无刻不在使用各种软件。随着 iOS 7、Xcode 5 以及最新 iOS 软件开发工具包（SDK）的发布，开发 iOS 应用正变得更加高效、更加有趣。

## 1.1 关于本书

本书是 iOS 应用编程的入门指南，旨在帮助你克服入门的困难，帮助你理解 iOS 应用程序的运行和构建方式。

在学习的过程中，你将会创建一系列小型应用程序，每个应用程序都会突出某些 iOS 特性，展示如何控制这些特性，以及如何与其交互。如果你扎实地掌握了本书的基础知识，充分发挥自己的创造力，并且持之以恒，再借助大量条理清晰的苹果文档，你就可以创建出专业的 iPhone 和 iPad 应用。

---

**提示** Dave、Jack、Jeff 和 Fredrik 为本书创办了一个论坛。志趣相投者可会聚于此，相互答疑解惑。一定要访问这个论坛哦！地址是 <http://forum.learncocoa.org>。

---

## 1.2 必要条件

开始编写 iOS 应用程序之前，需要做一些准备工作。初学者需要一台安装了 Mountain Lion（OS X 10.8）或 Mavericks（OS X 10.9）甚至更高版本的基于 Intel 架构的 Macintosh 计算机。任何最近上市的基于 Intel 架构的 Macintosh 计算机（台式机或笔记本）均可。

如果想访问到苹果更强大的全新开发工具，你需要注册成为 iOS 开发者。只需要访问 <http://developr.apple.com/ios/> 链接就可以创建你的开发者账号了。你会看到如图 1-1 所示的页面。

首先，点击 Log in 按钮，页面将提示你输入 Apple ID。如果没有 Apple ID，就点击 Join now 创建一个 ID，然后再登录。登录之后就进入了 iOS 开发中心的主页面。其中有各类文档、视频和示例代码等的链接，所有这些都可以帮助你更好地进行 iOS 应用开发。

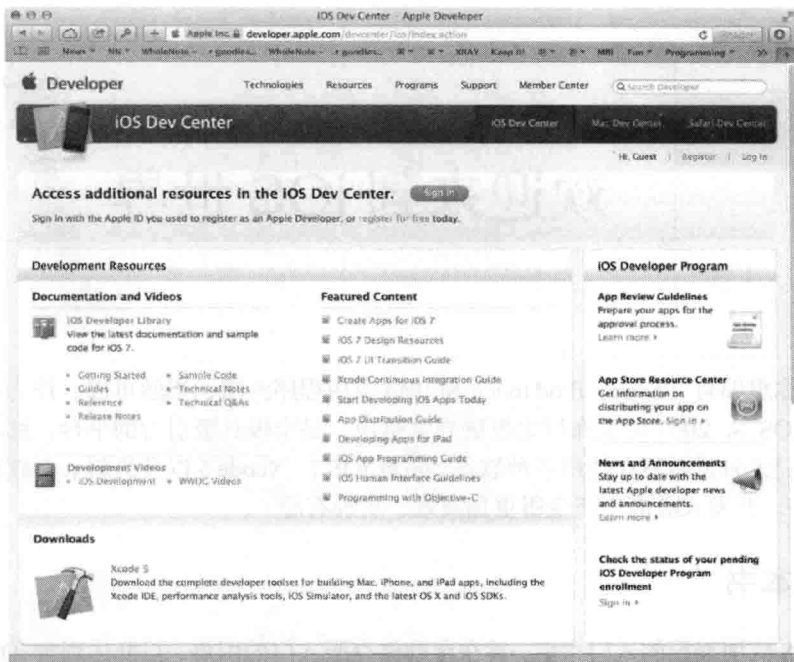


图 1-1 苹果的 iOS 开发中心网站

iOS 开发中最重要的工具是 Xcode，它是苹果的集成开发环境（IDE）。Xcode 提供了一些实用工具，用于创建和调试源代码、编译应用程序，以及对应用程序进行性能调优。

可以从 Mac App Store 下载 Xcode（可以通过 Mac 的 Apple 菜单访问 App Store）。

### 书中示例使用的 SDK 版本和源代码

随着 SDK 和 Xcode 版本的不断更新，它们的下载方式也在发生变化。在最近几年内，苹果已经开始把当前“稳定”版本的 Xcode 和 iOS SDK 放在 Mac App Store 中了，同时也会在开发者网站上提供预览版供开发者下载。总之，如果想下载 Xcode 和 iOS SDK 最新版本（非 beta 版），应该使用 Mac App Store。

本书面向最新版 SDK。在一些地方，我们会使用 iOS 7 中引入的新函数或方法，它们可能与旧版 SDK 不兼容。当然，出现这些情况时我们会特别指出。

请从 <http://learn.cocoa.org> 或者本书论坛 <http://forum.learn.cocoa.org> 上下载最新的源代码<sup>①</sup>。当有新版 SDK 发布时，我们会及时更新代码，所以你应该定期登录这些网站，看看是否有更新内容。

① 本书源代码也可从图灵社区本书页面“随书下载”部分下载，网址是 <http://www.it-ebooks.com.cn/book/1390>。

### 1.2.1 开发者选项

免费下载的 Xcode 中包含一个模拟器, 使用这个模拟器, 就可以在 Mac 上创建并运行 iPhone 和 iPad 应用。这个模拟器对于学习编写 iOS 程序极其有用。但是, 模拟器不支持那些依赖于硬件的特性, 比如加速计和摄像头。另外, 免费下载的 SDK 不支持把应用程序安装到 iPhone 等 iOS 真机设备中, 也不能在苹果的 App Store 上出售应用。如果想要这些功能, 需要注册一个付费选项。

- 标准版计划的价格为 99 美元/年。它提供了大量的开发工具和资源, 以及技术支持, 还可以通过苹果的 App Store 出售应用。最重要的是, 它允许在 iOS 设备上 (而不只是在模拟器上) 测试和调试代码。
- 企业版计划的价格为 299 美元/年。如果需要开发公司内部使用的私有 iOS 应用程序, 可以选择这个计划。

可以访问 <http://developer.apple.com/programs/ios> (标准版计划) 和 <http://developer.apple.com/programs/ios/enterprise> (企业版计划) 来查看这两个计划的详细信息。

对于 iOS 所支持的始终联网的移动设备 (比如 iPhone), 由于这种设备使用的是其他公司的无线基础设施, 因此苹果对 iOS 开发人员的限制比对 Mac 开发人员严格得多 (目前, Mac 开发人员无需经过苹果的审查或批准就可以编写并销售程序)。虽然 iPod touch 和那种只支持 Wi-Fi 的 iPad 不使用其他公司的基础设施, 但是它们也受到同样的限制。

苹果添加这些限制, 主要是为了尽量避免发布恶意程序和不良程序, 因为这类程序可能会在共享网络中降低性能。开发 iOS 应用似乎有很高的门槛, 但是苹果也为简化开发过程做出了巨大努力。值得一提的是, 99 美元比 Visual Studio (微软公司的软件开发 IDE) 的价格低得多。

显然, 你还需要一部 iPhone、iPod touch 或 iPad。虽然很多代码都可以通过 iOS 模拟器进行测试, 但并非所有程序都是如此。即便是那些可以在模拟器上运行的应用程序, 也需要在真实设备上进行全面测试后才能对外发布。

---

**注意** 如果你准备注册标准版计划或企业版计划, 最好马上注册。因为审批过程可能需要一些时间, 通过审批之后才能在真实设备上运行应用程序。不过不必担心, 前几章中的所有项目以及本书的大多数应用程序都可以在 iOS 模拟器上运行。

---

### 1.2.2 必备知识

学习本书应该具备一定的编程知识。我们假设你已经理解基本编程尤其是面向对象编程的基础知识 (例如, 知道类、对象、循环、变量这些基本概念)。你还应该熟悉 Objective-C 编程语言。本书大部分章节都需要用到 Cocoa Touch (它是 SDK 的一部分), Cocoa Touch 使用的是最新版的 Objective-C, 这个版本有一些新特性。不过不了解 Objective-C 的新特性也没有关系。本书在用到 Objective-C 的新特性时会特别指出, 并解释它们的工作原理和使用原因。

作为用户, 你还应该熟悉 iOS 系统本身。就像在其他平台中编写应用程序一样, 你需要熟悉

iPhone、iPad 或 iPod touch 的一些细微差别和怪异模式。花点时间去熟悉 iOS 界面以及 iPhone/iPad 应用的风格吧。

### Objective-C 的学习资源

如果你从未使用 Objective-C 编写程序，下面这些资源可以作为入门读物。

- 《Objective-C 基础教程(第2版)》一书浅显易懂，是非常优秀的 Objective-C 入门读物，作者是 Mac 编程专家 Scott Knaster、Waqar Malik 和 Mark Dalrymple。
- *Programming with Objective-C*，这本是苹果出的 Objective-C 语言入门书。你可以在 <https://developer.apple.com/library/mac/documentation/cocoa/conceptual/ProgrammingWithObjectiveC> 网站上找到更多的内容。

## 1.3 编写 iOS 应用程序有何不同

如果从未使用 Cocoa(或者它的前辈 NeXTSTEP 和 OpenStep)写过程序，你可能会发现 Cocoa Touch(用于编写 iOS 应用程序的应用程序框架)比较新奇。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)之间存在一些根本差异。起初你可能会有点不得要领，不过不必担心，只要多加练习，很快就能够对 Cocoa Touch 运用自如了。

如果曾经使用 Cocoa 或 NeXTSTEP 写过程序，你会发现 iOS SDK 中有许多熟悉的身影。有很多类都是从 Mac OS X 版本的 Cocoa 中原样移植过来的。即便是那些不同的类，也遵循相同的基本原则和相似的设计模式。但是，Cocoa 和 Cocoa Touch 之间还是有一些不同的。

无论你的知识背景如何，都需要时刻牢记 iOS 开发与桌面应用程序开发之间的重要差异。接下来几小节讨论这些差异。

### 1.3.1 只能同时运行一个应用

在 iOS 中，任意时刻只能有一个应用处于活动状态并显示在屏幕上。从 iOS 4 开始，用户按下 Home 键后，应用程序可以在后台继续运行，但这也只限于少数情况，而且必须专门为此编写代码。

当应用程序不处于活动状态也不是在后台运行时，它不会占用任何 CPU 资源，因此也会断开网络连接。iOS 允许在后台进行处理，但要做到这一点，开发者需要多做些努力。

### 1.3.2 只有一个窗口

在台式机及笔记本的操作系统中，多个程序可以同时运行，每个程序可以创建并控制多个窗口。而 iOS 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，而且这个窗口的大小就是 iOS 设备屏幕的大小，是固定的。



### 1.3.3 有限的访问权限

计算机上的程序可以访问属主用户（启动这个程序的用户）能够访问的任何内容。然而，iOS 严格限制了应用程序的访问权限。

iOS 的文件系统会为每个应用分配一块独立的区域，这块区域称为沙盒，每个应用只能对自己沙盒内的文件进行读写。沙盒就是应用用于存储文档、首选项等任何必要数据的地方。

应用程序还会受到其他方面的限制。比如，不能访问 iOS 中端口号较小的网络端口，也不能做那些在台式机中需要有根用户权限或管理员权限才能进行的操作。

### 1.3.4 有限的响应时间

由于使用方式特殊，iOS 需要能够快速响应各种事件，你的应用程序也应如此。启动应用程序时，需要先打开它，载入首选项和数据，并尽快把主视图显示到屏幕上，这些过程要在几秒钟内全部完成。

在应用程序运行中的任何时刻，都可以通过双击 Home 键查看最近使用的应用列表。如果用户按 Home 键，iOS 就会返回主屏幕，应用必须快速保存一切内容并退出。如果没有在 5 秒之内保存必要的数 据并放弃对系统资源的控制，无论是否已经保存完成，应用程序进程都会被终止。有一个 API 可以在应用程序终止前申请多一些的时间来完成必要的工作。你必须知道如何使用它才行。

### 1.3.5 有限的屏幕大小

iPhone 的屏幕显示效果非常出色，在相当长的一段时间里，iPhone 一直都是市场上分辨率最高的掌上电子设备。

但是，iPhone 的显示空间并不大，与现代计算机相比，在 iPhone 上能使用的屏幕空间要小很多。最初几代的 iPhone 屏幕分辨率只有 320 像素×480 像素，后来，从 iPhone 4 的视网膜屏幕开始，分辨率增加到了 640 像素×960 像素。iPhone 5 的分辨率进一步提高到了 640 像素×1136 像素。像素数量有了极大的提高，但是屏幕的显示空间却没有大的变化，因此还是不能在屏幕上摆放更多的控件之类。小屏幕极大地影响了 iPhone 所能提供的应用种类和交互性。

iPad 的分辨率是 1024 像素×768 像素，显示空间也增加了，但也不是非常大。下面来做一个有趣的对比，写作本书时，苹果最便宜的 iMac 的分辨率是 1920 像素×1080 像素，最便宜的笔记本电脑（11 寸的 MacBook Air）的分辨率是 1366 像素×768 像素，而苹果最大的显示器是 27 英寸的 LED Cinema Display 则支持高达 2560 像素×1440 像素的超高分辨率。请注意，对于 iPad 而言，如果不出意外的话，往后的 iPad 机型（iPad2 以后的全尺寸 iPad 以及 iPad Mini Retina）都将配备视网膜屏幕，横竖两个方向的分辨率都加倍了。但是，跟视网膜屏幕的 iPhone 一样，这块 2048 像素×1536 像素的屏幕的实际尺寸跟旧屏幕一样，仍无法像在传统的屏幕上那样使用这些像素。

### 1.3.6 有限的系统资源

一部拥有 512 MB 内存和 16 GB 存储空间 的机器竟然需要对资源使用做很多限制，很多资深

程序员该要笑话苹果的这种做法了。可事实确实如此。或许开发 iOS 应用程序并不像是在内存为 48 KB 的机器上编写复杂的电子表格应用，但是，由于 iOS 具备的图形特性和多种功能，内存很容易就会耗光。

目前上市的 iOS 设备的物理内存要么是 512 MB( iPhone 4S、iPad 2、一代 iPad mini、iPod touch 5)，要么是 1024 MB( iPhone 5c、iPhone 5S、iPad Air、iPad mini Retina)，而且以后还会不断加大。很大一部分内存被用于屏幕缓冲区和和其他一些系统进程。通常，只有不到一半的内存留给应用程序使用（实际可用内存可能会更少，尤其是现在其他的应用可以在后台运行了）。

虽然这些内存对于这种小型移动设备来说可能已经足够了，但谈到 iOS 的内存时，却还要考虑另一个因素。现代的计算机操作系统（比如 OS X）会将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的实际可用内存时，它仍然可以正常运行。但是，iOS 并不会将易失性内存（比如应用程序数据）写到交换文件中。因此，应用程序的可用内存大小受限于 iOS 设备中未使用的物理内存空间。

Cocoa Touch 提供了一种内置机制，可以在内存不足时通知应用程序。出现这种情况时，应用程序必须释放不必要的内存空间，否则就可能被强制退出。

### 1.3.7 不支持垃圾回收

之前提过，Cocoa Touch 使用的是 Objective-C，但是 iOS 却并不支持一个 Objective-C 早在本世纪初就已经有的关键特性：垃圾回收。是的，Cocoa Touch 不支持垃圾回收。为 iOS 编写程序时需要手动管理内存，许多刚刚接触这个平台（尤其是那些从支持垃圾回收的语言转过来）的开发者，还真有点不太适应。

但是，最新版本 iOS 支持的 Objective-C 基本解决了这个问题。这要归功于 ARC（Automatic Reference Counting，自动引用计数）功能，它解决了手动管理 Objective-C 对象占用内存的问题。ARC 并不仅仅是垃圾回收的替代品，实际上，它在很多方面比垃圾回收更优秀。因此，从 OS X 10.8 开始，ARC 就成为了针对 Mac 应用程序的默认内存管理机制，垃圾回收则被废弃了。当然 ARC 也是 iOS 应用程序的默认内存管理机制。第 3 章会详细介绍 ARC。

### 1.3.8 新功能

前面提过，Cocoa Touch 缺少 Cocoa 的一些特性，但 iOS SDK 中也有一些功能是 Cocoa 目前没有的，至少不是在每一部 Mac 上都可用。

- ❑ iOS SDK 中的 Core Location 框架可以帮助应用程序确定 iOS 设备的当前地理坐标。
- ❑ 大部分 iOS 设备都有内置的摄像头和照片库，SDK 允许应用程序访问它们。
- ❑ iOS 设备还有一个内置的运动传感器，用于检测设备的握持和移动方式。

### 1.3.9 与众不同的交互方法

iOS 设备没有物理键盘和鼠标，这意味着 iOS 应用与用户的交互方式跟通用计算机软件完全

不同。幸好，大部分的交互都会由 iOS 系统完成。例如，如果应用中用到了文本框，iOS 系统就会在用户单击这个文本框时调出软键盘，不需要开发者为此额外编写代码。

**注意** 所有的设备都支持通过蓝牙连接外置键盘，这提供了一种不错的键盘体验，并节省了屏幕空间，但使用这种键盘的人仍然非常少。而且，现在依然无法连接鼠标。

## 1.4 本书内容

下面是本书其余各章的简要概述。

- ❑ 第 2 章：讲述如何使用 Xcode 的搭档 Interface Builder 创建简单的界面，并在屏幕上显示一些文本。
- ❑ 第 3 章：展示与用户的交互，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，介绍更多的标准 iOS 用户界面控件。此外，还将介绍如何使用警告框和操作表单提醒用户作出决策，或者通知用户发生了一些异常事件。
- ❑ 第 5 章：解释如何处理自动旋转和自动改变大小属性，以及使 iOS 应用程序同时支持纵向和横向运行的机制。
- ❑ 第 6 章：分析更多高级用户界面，并讲述如何创建多视图应用程序。还会介绍如何在运行时改变展示给用户的视图，这样可以创造出更强大的应用。
- ❑ 第 7 章：讨论如何实现标签栏和选取器等界面元素，它们是标准 iOS 用户界面的一部分。
- ❑ 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。这一章还会介绍如何让用户搜索应用程序数据。
- ❑ 第 9 章：说明如何实现分层列表这一标准的界面类型。分层列表是最常用的 iOS 应用程序界面之一，可以通过它查看更多详细的数据。
- ❑ 第 10 章：展示集合视图的使用。各种 iOS 应用程序从一开始都在使用表视图展示动态的、垂直滚动的组件列表。在最近一段时间，苹果引入了 `UICollectionView` 类，用于实现集合视图。集合视图在展示动态、垂直滚动的组件列表方面功能更加强大。使用集合视图，开发者可以更灵活地处理视觉组件的排列。
- ❑ 第 11 章：讲述如何使用 SDK 中面向 iPad 的部分。iPad 的外形与其他 iOS 设备不同，它需要用不同的方法来显示 GUI（图形用户界面），SDK 中有一些专用于 iPad 的组件。
- ❑ 第 12 章：展示如何实现应用程序设置，iOS 中的这种机制允许用户设置应用程序级的首选项。
- ❑ 第 13 章：介绍 iOS 中的数据管理。将讨论如何创建对象来保存应用程序数据，以及如何将这些数据持久化到 iOS 文件系统中。这一章还会介绍使用 Core Data 的基础知识，使用 Core Data 可以很方便地保存和检索数据。

- ❑ 第 14 章：解释如何使用 iCloud。iCloud 是 iOS 5 引入的一项功能，文档数据可以保存到 iCloud 中，这样就可以在不同设备之间同步应用数据。
- ❑ 第 15 章：iOS 开发人员可以使用 Grand Central Dispatch 这种新方法进行多线程开发，它还可以让应用必要时在后台运行。这一章将展示如何使用 Grand Central Dispatch 和后台处理。
- ❑ 第 16 章：人们都喜欢画画，这一章介绍自定义绘图，我们将为你讲解 Core Graphics 的绘图机制。
- ❑ 第 17 章：在 iOS 7 中，苹果引入了一个叫作 Sprite Kit 的新框架，用来创建 2D 游戏。它包含了物理引擎和动画系统，也可以用来创建 OS X 系统上的游戏。在第 17 章中，你将会看到如何使用 Sprite Kit 创建一个简单的游戏。
- ❑ 第 18 章：iOS 设备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，还将介绍如何以及何时应该自定义新手势。
- ❑ 第 19 章：iOS 可以通过 Core Location 确定设备所处的纬度和经度。本章中会编写一些使用 Core Location 的代码，用于计算设备所在位置，并且使用位置信息来实现我们让世界连通的理想。
- ❑ 第 20 章：讨论如何与 iOS 的加速计和陀螺仪进行交互，通过它们可以确定设备的握持方式、运动速度，以及移动方向、设备所在位置。我们也会研究应用使用这类信息可以做哪些有趣的事情。
- ❑ 第 21 章：每台 iOS 设备都有自己的摄像头和照片库，获得用户允许之后，你的应用程序就可以访问摄像头和照片库。这一章介绍如何在自己的应用程序中使用它们。
- ❑ 第 22 章：iOS 设备现已遍及 90 多个国家。本章会介绍如何把应用的各个部分恰当地翻译为其他语言，这样有助于发掘应用的潜在用户。
- ❑ 至此，你已经掌握了 iPhone 和 iPad 应用的基本构建方法。附录将给出一些精通 iOS SDK 需要关注的其他资源。

## 1.5 这一版的新内容

自本书第 1 版上市以来，iOS 开发社区也取得了巨大发展。苹果不断更新 SDK，使它持续发展完善。

当然，我们也很忙。iOS 7 增强了许多新的功能并改变了显示内容的方式。Xcode 5 也添加了许多强大的功能，Interface Builder 还更好地改善了对自动布局机制的支持，采用了新的图片资源管理方式，全面地使用分镜（storyboard）来代替所有旧的 nib 项目模板（不过需要注意的是，nib 文件以及使用了它的项目仍然可以正常运行，以后的版本也一样）。我们竭尽全力地更新章节内容，使本书能完整地覆盖所有这些新的技术。每个项目都是我们从头开始重新构建的，以确保项目的代码不但能在最新版的 Xcode 和 iOS SDK 下编译，而且还能够充分利用 Cocoa Touch 提供的强大新功能。这一版内容进行了大量细微调整，也添加了许多实质性变动，包括新增一章内容用

来讲述 Sprite Kit 游戏框架。当然我们也对本书中的所有屏幕截图进行了更新。

## 1.6 准备开始吧

iOS 是一个不可思议的计算平台，是令人兴奋的新领域，令开发充满乐趣。编写 iOS 程序不同于为其他任何平台开发应用程序，将成为一种全新的体验。所有看似熟悉的功能都具有其独特之处，但只要深入理解本书中的代码，就能把这些概念紧密联系起来，并融会贯通。

应该谨记，完成本书中的所有练习并不能立即成为 iOS 开发专家。在进行下一个项目之前，请确保你已经完全理解了已完成的项目。不要害怕修改代码，熟悉 Cocoa Touch 这种编程环境的最佳方法就是不断实验并观察结果。

如果你已经安装了 iOS SDK，请继续阅读本书；如果还没有，请立即安装。明白吗？好，开始动手吧！

## 第 2 章

# 创建项目

# 2

你可能知道，任何编程书都习惯使用“Hello, World!”作为第一个项目的名称，这已然成为一种传统。我们曾考虑过打破这种常规，但又恐标新立异犯了众怒。因此，本书也以“Hello, World!”作为第一个项目。

本章将使用 Xcode 创建一个小型 iOS 应用，在模拟设备屏幕上显示文本“Hello, World!”。我们将讨论使用 Xcode 创建 iOS 应用的各方面内容，深入探究使用界面构建器设计应用用户界面的具体细节，最后在 iOS 模拟器上运行应用。随后，将为应用指定一个图标，让它看起来更像真正的 iOS 应用。

要做的事情很多，开始吧。

## 2.1 在 Xcode 中创建项目

现在，你应该已经在机器上安装了 Xcode 和 iOS SDK。还应该从本书的网站 (<http://www.learncocoa.com>) 下载本书的项目归档文件。还可以顺便看一下本书的论坛 (<http://forum.learncocoa.org/>)。本书的论坛是讨论 iOS 开发的好地方，你可以在这里提问，还能够结交到志同道合的朋友。

---

**注意** 即使你已经拥有了本书的完整项目文件，这里仍然建议你亲自动手创建每一个项目，而不是直接运行下载到的项目文件。只有通过实践，才能更好地熟悉并精通各种应用开发工具。软件开发可不是只靠观察就能学好的，除了亲自动手创建应用，别无他法。

---

在本书项目归档的 02-Hello World 文件夹中，可以找到本章要创建的项目。

在开始之前，需要启动 Xcode。本书中大部分编程工作都要使用 Xcode 完成，从 Mac App Store 下载到 Xcode 之后，你会发现 Xcode 与大多数 Mac 应用一样安装到了/Applications 文件夹下。因为要经常用到 Xcode，所以可以将它的图标拖到 Dock 上，以方便使用。

如果是第一次使用 Xcode，也不用担心，我们将详细介绍创建新项目的每一步骤。如果你很熟悉以前版本的 Xcode，但是还没用过 Xcode 5，就会发现有很多变化(大多是变得比旧版更好了)。

第一次启动 Xcode 时，会显示如图 2-1 所示的欢迎窗口。从这里可以创建一个新的项目，可

以连接到版本控制系统, 检查已有项目, 还可以从最近打开的项目列表中选择个项目。欢迎窗口为新手提供了一个非常好的入门指南, 它把最常见的一些任务列了出来。所有这些功能都可以在 Xcode 菜单里找到。浏览完成之后关闭这个窗口, 继续学习其他内容。如果不希望再次看到这个窗口, 只需在关闭该窗口前取消勾选底部的 Show this window when Xcode launches (当 Xcode 加载时显示该窗口) 选项就可以了。



图 2-1 Xcode 欢迎窗口

**注意** 如果你的 Mac 机器连接了 iPhone、iPad 或者 iPod touch 等设备, 那么第一次启动 Xcode 时可能会看到一个对话框, 询问你是否要使用该设备进行开发。就目前的学习来说, 点击 Ignore (忽略) 按钮就可以了。否则就会显示 Organizer (组织者) 窗口, 列出与机器同步的设备。这种情况下, 直接关闭 Organizer 窗口即可。如果已经加入了付费的 iOS 开发者计划 (iOS Developer Program), 那么你就能访问苹果的应用开发者计划网站, 了解如何使用 iOS 设备进行开发和测试。

要创建新项目, 可以选择 File>New>Project... (或者按下 shift+command+N)。这时会看到一个新建项目窗口, 其中显示了项目模板选择面板 (参见图 2-2)。可以从这个面板中选择一个项目模板作为构建应用的起点。窗口左侧的面板分为两个主要部分: iOS 和 OS X。由于我们要创建的是 iOS 应用, 所以选择 iOS 部分的 Application (应用) 类别, 以显示 iOS 应用模板。

图 2-2 右上方面板中的每一个图标都表示一个独立的项目模板, 这些模板可以用作构建 iOS 应用的基础。Single View Application (单视图应用) 是最简单的模板, 本书前几章将会用到它。而其他模板提供了额外的代码和资源来创建一些通用的 iPhone 和 iPad 应用界面, 这些将在后面的章节介绍。

单击 Single View Application 图标 (如图 2-2 所示), 然后单击 Next 按钮, 就会看到项目选项表单, 如图 2-3 所示。在这个表单中, 需要为项目指定 Product Name (产品名称) 和 Company Identifier (公司标识)。Xcode 会将这些内容结合起来, 为应用生成一个唯一的 Bundle Identifier



(包标识符)。还可以看到一个 Organization Name (组织名称) 字段, Xcode 会自动在你创建的每一个源代码文件中以这个名称插入版权声明。把 Product Name 设置为 Hello World, 组织名称为 Apress, 然后在 Company Identifier 字段填写 com.apress, 如图 2-3 所示。等你注册了开发者计划并且了解授权文件 (provisioning profile) 之后, 就可以使用自己的公司标识了。本章稍后将会更详细地讨论包标识符。

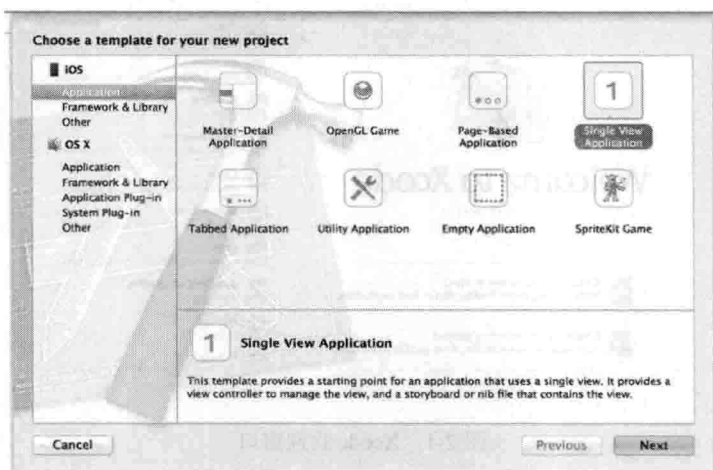


图 2-2 创建项目时, 可以从项目模板选择面板列出的模板中选择一个

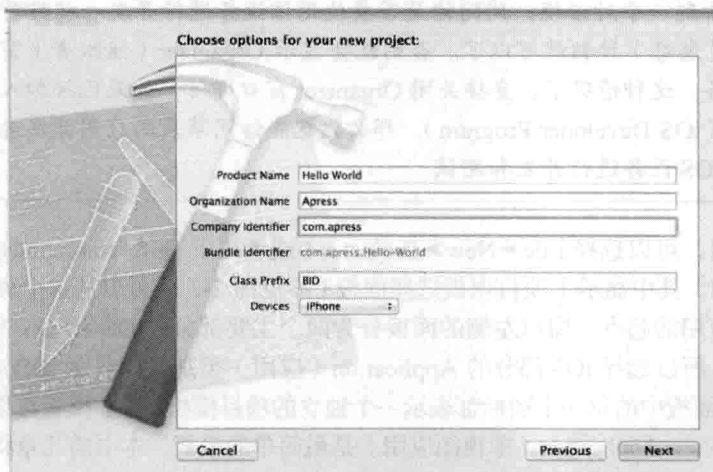


图 2-3 为项目选择 Product Name 和 Company Identifier。目前使用如图所示的设置

下一个文本框是 Class Prefix (类名前缀), 这里应该输入一个至少包含 3 个大写字母的字符串。这个字符串会被添加到 Xcode 为我们创建的所有类的类名前面。这么做是为了避免与苹果 (使用两个字符的前缀) 以及可能会用到的其他开发者的代码发生命名冲突。在 Objective-C 中,

两个类同名会导致应用构建失败。

本书的项目将使用 BID 作为前缀，BID 代表 Beginning iOS Development。例如，可能有很多类会被命名为 MyViewController，然而被命名为 BIDMyViewController 的类就很少了，这就大大降低了命名冲突的可能性。

我们还需要设置 Devices（即设备类别）的值。也就是说，Xcode 需要知道我们要创建的应用究竟是用于 iPhone、iPod touch，还是 iPad，或是能够在所有 iOS 设备上运行的通用应用。在设备类别中选择 iPhone（如果它还没有被选中）。这就告诉 Xcode 我们将创建的这个应用专用于 iPhone 和 iPod touch（它们的屏幕尺寸相同）。本书第一部分将始终使用 iPhone 作为设备类别，不过不用担心，后面会介绍 iPad 开发。

再次单击 Next 按钮，此时会看到一个标准的保存对话框，要求指定项目的保存位置，如图 2-4 所示。如果你尚未为本书的项目创建一个新的主目录，那就先转到 Finder 创建一个，然后回到 Xcode，切换至该目录。点击 Create 按钮之前，先注意一下 Create local git repository for this project（创建该项目的本地 git 仓库）选项。本书不打算讨论 Git，但是 Xcode 内置了对 Git 和其他版本控制系统的支持。如果比较熟悉 Git 并且希望在项目中使用它，就选中这个选项，否则就不要选中。

**注意** 版本控制系统（Source Control Management，简称 SCM）这个技术用于在构架应用时跟踪代码变更和资源变更。它提供了一些工具，可以解决多个开发者同时更改某一个应用时可能引起的冲突问题。Xcode 已经内置了对 git（当今最流行的版本控制系统）的支持。本书不使用版本控制系统，所以是否要勾选这个选项取决于你，选不选都可以。

无论是否勾选了 git 版本控制，点击 Create 按钮之后都会创建一个新的项目出来。

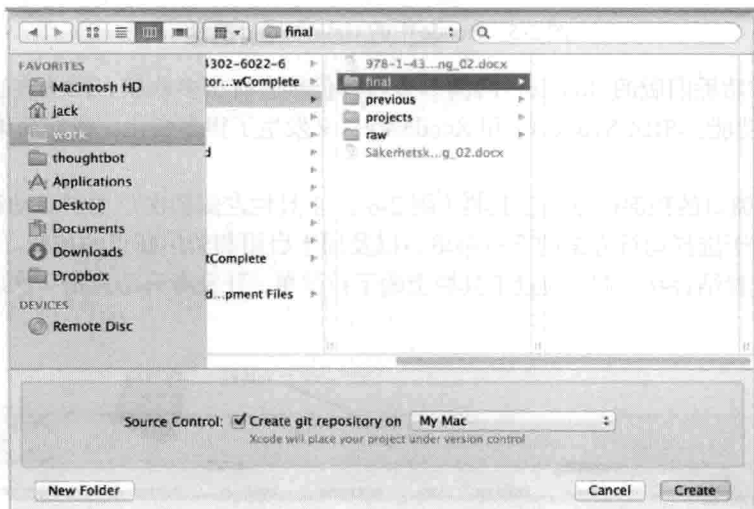


图 2-4 将项目保存到硬盘上的某个项目文件夹中

### 2.1.1 Xcode项目窗口

确定项目的保存路径之后, Xcode 就会创建并打开项目。这时会看到一个新的项目窗口, 如图 2-5 所示。该窗口包含许多信息, 它也是进行 iOS 开发用到的主要窗口。

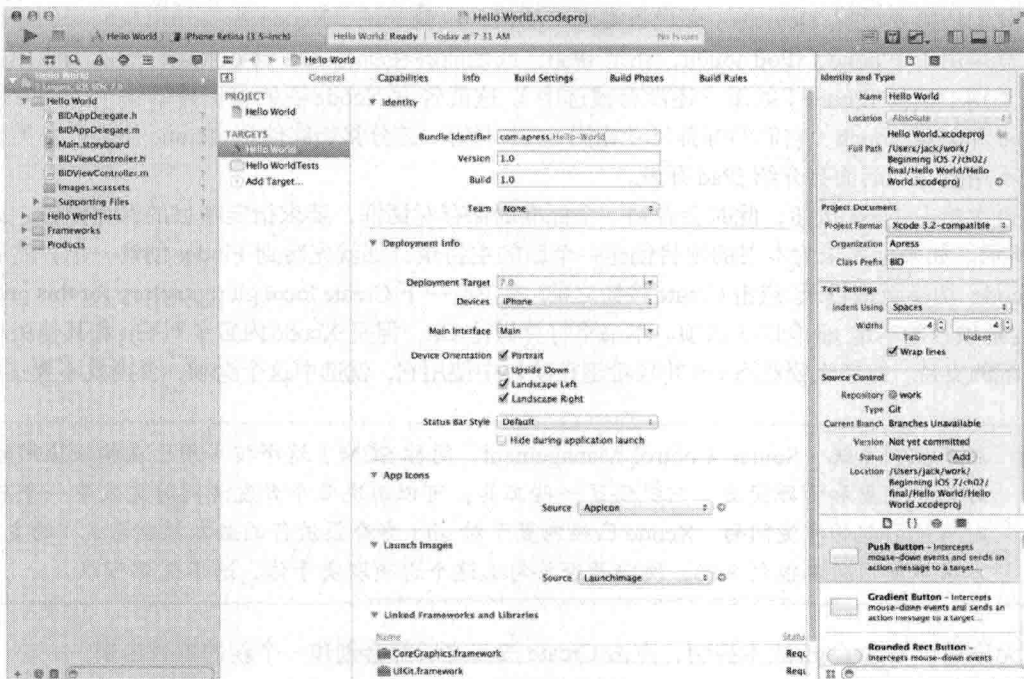


图 2-5 Xcode 中的 Hello World 项目

即使你非常熟悉旧版的 Xcode, 阅读本节内容仍然会有很多收获, 因为在这里介绍了很多 Xcode 5 上的新功能, 相比 Xcode 3.x 和 Xcode 4.x, 又发生了很多变化。我们简单了解一下。

#### 1. 工具栏

Xcode 项目窗口的顶部区域是工具栏 (图 2-6)。工具栏左侧依次是用于启动和停止项目运行的控制按钮、用于选择运行方案的下拉菜单, 以及用于启用和禁用断点的按钮。方案 (scheme) 将目标和构建设置结合在一起。通过工具栏上的下拉菜单, 开发者只需点击一次就能选择一种特定的设置。

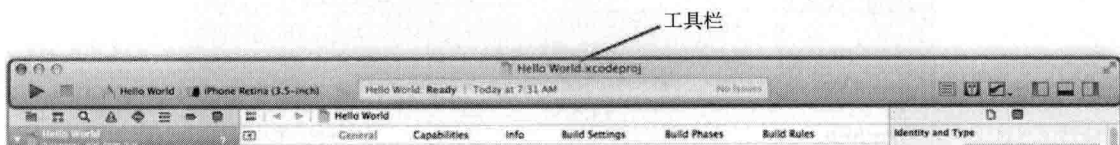


图 2-6 Xcode 工具栏

工具栏中间的大方框是**活动视图 (activity view)**。顾名思义, 活动视图显示着当前正在进行的操作或处理。例如, 运行项目时, 活动视图会显示应用构建过程中每一步的运行时说明。如果出现了错误或者警告, 这些信息也会显示在活动视图中。如果点击其中的警告或错误, 就会直接跳转到问题导航面板, 问题导航面板提供了更为详细的警告和错误信息, 下一节将会介绍这部分内容。

在工具栏右侧有两组按钮。左侧的一组按钮可以在三种不同的编辑器配置间进行切换。

- ❑ **标准编辑器 (standard editor)** 提供了一个面板, 用于编辑文件或者编辑项目相关的配置数值。
- ❑ **辅助编辑器 (assistant editor)** 非常强大, 它将编辑器面板分割为左右两个部分。右侧面板通常用于显示左侧面板中文件的关联文件, 或者是你在编辑左侧文件时可能会用到的文件。可以手动指定每个面板的内容, 也可以让 Xcode 自动判断进而显示与当前任务匹配度最高的内容。例如, 如果你正在编辑一个 Objective-C 类的实现文件 (.m 文件), Xcode 就会自动在右侧面板中显示这个类的头文件 (.h 文件)。如果你正在左侧面板中设计用户界面, Xcode 就会在右侧面板中显示能够与这个用户界面进行交互的代码。本书从头至尾都在使用辅助编辑器。
- ❑ **版本编辑器 (version editor)** 按钮将编辑器面板转换为一个与 Time Machine 类似的对比视图, 这个对比视图可以跟 Subversion 和 git 等版本控制系统协同工作。可以将一个源文件的当前版本与之前提交的版本进行比较, 或者对任意两个之前的版本进行比较。

编辑器按钮的右侧是另一组开关, 用于控制左侧的面板、右侧编辑器视图以及项目窗口底部的调试区域的显示与隐藏。动手点击这些按钮几次, 试试这组开关的功能。你很快就要学习如何使用这些按钮了。

## 2. 导航视图

工具栏下方, 项目窗口左侧就是**导航视图 (navigator)**。如果你之前曾使用导航视图开关把它隐藏的话, 请再次将其显示出来。导航视图共提供了 8 个面板, 供开发者以不同的方式查看项目。点击导航视图顶部的图标可以在不同类型的导航面板中进行切换。下面从左至右依次介绍它们。

- ❑ **项目导航面板 (project navigator)**: 这个面板列出了项目用到的所有文件, 如图 2-7 所示。可以把任何想要的内容引用放在这里, 从源代码文件到图片文件、数据模型、属性列表文件 (也叫 plist 文件, 2.1.2 节会介绍), 甚至是其他项目文件。在一个工作区中存放多个项目便于项目之间共享资源。在项目导航面板中点击任意一个文件, 该文件都会在编辑器面板中显示。不仅能查看文件, 还可以编辑 (只要 Xcode 知道如何编辑这种文件)。

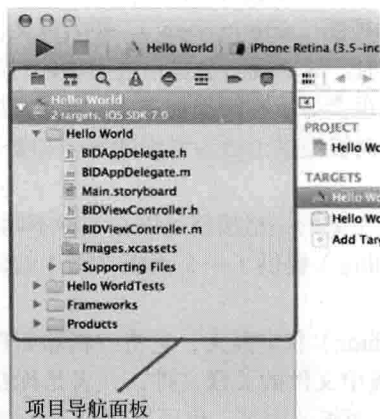


图 2-7 Xcode 项目导航面板。通过单击导航视图顶部的 8 个图标，就可以在不同的导航面板之间切换

- 符号导航面板 (symbol navigator)：顾名思义，这个导航面板集中了所有在工作区中定义的符号 (symbol)，如图 2-8 所示。从根本上说，符号就是那些编译器能识别的东西，例如 Objective-C 类、枚举类型、结构体和全局变量。

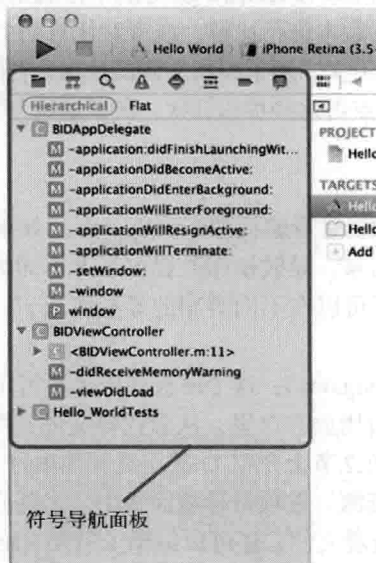


图 2-8 Xcode 符号导航面板。点击倒三角按钮可以看到每个分组中定义的文件和符号

- 搜索导航面板 (Find navigator)：使用这个导航面板可以对工作区中的所有文件执行搜索，如图 2-9 所示。面板顶部有多层下拉菜单，Find (查询) 功能也可以改成 Replace (替换)

功能，此外还可以对输入的文本采用不同的搜索方式。在文本框下方的其他控件可以让你选择搜索范围是整个项目还是其中一部分，或指定是否区分字母大小写。

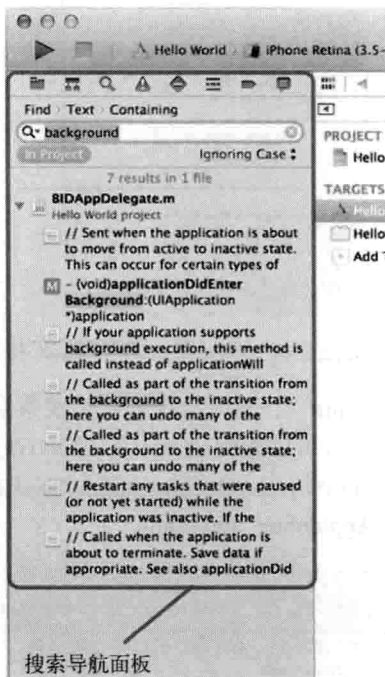


图 2-9 搜索导航面板。记得查看一下 Find 文本中和位于搜索框下面的按钮上的隐藏下拉菜单

- 问题导航面板 (issues navigator)：构建项目过程中出现的任何错误或者警告都会在这个导航面板中显示，同时窗口顶部的活动视图中会显示错误数量，如图 2-10 所示。点击问题导航面板中的任一错误，就会跳转到编辑器面板中相应的代码行。

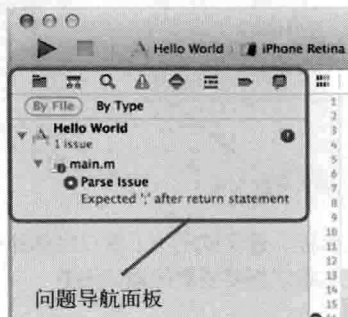


图 2-10 问题导航面板。可以在这里找到编译错误和警告

- ❑ 测试导航面板 (Test navigator): 如果你正在使用 Xcode 内置的单元测试功能 (很遗憾, 本书不会深入介绍这方面的内容), 那么你将会在这里看到测试的结果 (如图 2-11 所示)。



图 2-11 测试导航面板。单元测试的打印结果将显示在这里

- ❑ 调试导航面板 (debug navigator): 这个导航面板是观察调试过程的主要区域, 如图 2-12 所示。如果你对调试很陌生, 可以查阅 *Xcode 5 User Guide* 文档的相关部分, 网址是: [http://developer.apple.com/library/mac/#documentation/ToolsLanguages/Conceptual/Xcode4UserGuide/060-Debug\\_Your\\_App/debug\\_app.html](http://developer.apple.com/library/mac/#documentation/ToolsLanguages/Conceptual/Xcode4UserGuide/060-Debug_Your_App/debug_app.html)。

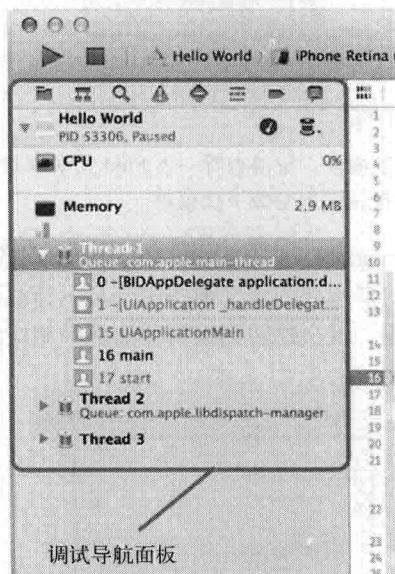


图 2-12 调试导航面板。建议试用一下窗口底部的细致程度滑块, 可以通过它指定想要查看的调试细节

调试导航面板列出了每个活动线程的栈帧。栈帧 (stack frame) 按调用序列出了之前调用过的函数或方法。点击某个方法, 与之对应的代码就会显示在编辑器面板中。在编辑



器中, 还有另一个面板, 可以用来控制调试过程, 显示和修改数据值以及访问底层调试器。调试导航面板底部还有一个滑块, 可以控制调试跟踪的细致程度。滑动到最右边可以看到所有内容, 包括所有的系统调用; 滑动到最左边的话就只能看到自己的调用了。默认位置为正中间, 这是个不错的位置。

- ❑ **断点导航面板 (breakpoint navigator)**: 可以在断点导航面板中查看已设置的所有断点, 如图 2-13 所示。顾名思义, 断点会指向导致应用停止运行 (或者跳出) 的代码部分, 这样就可以查看变量中的值, 做一些别的工作来调试应用。这个导航面板中的断点列表是以文件的形式组织的。在列表中点击一个断点, 编辑器面板中就会显示该断点所对应的代码行。位于断点导航面板时, 记得查看一下项目窗口左下角的弹出菜单。可以通过加号弹出菜单添加异常断点或者符号断点, 也可以通过减号弹出菜单删除选定的断点。

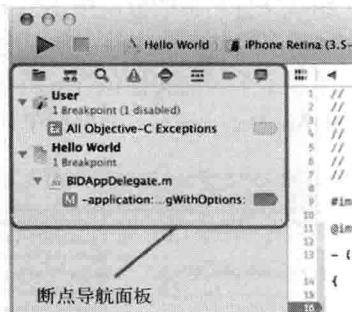


图 2-13 断点导航面板。断点列表以文件的形式进行组织

- ❑ **日志导航面板 (log navigator)**: 这个导航面板中保存着构建结果和运行日志的历史记录, 如图 2-14 所示。点击某条日志, 编辑器面板就会显示相应的构建指令和构建问题。

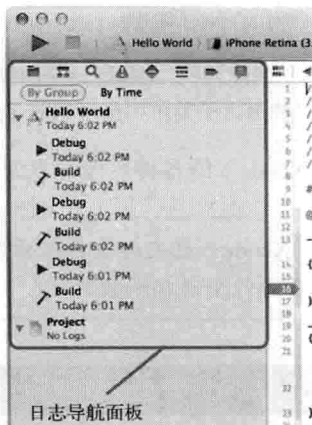


图 2-14 日志导航面板。它显示了一个项目运行列表, 选中某项后, 编辑器面板中就会出现对应的详细信息日志



### 3. 跳转栏

在编辑器的顶部，你会找到一个叫作跳转栏（jump bar）的特殊控件，你只需要单击一次，就能跳转到当前导航层次结构中的特定元素。例如，图 2-15 显示了一个正在编辑器面板中进行编辑的源代码文件。跳转栏就在源代码上方，它有以下几个组成部分。

- ❑ 最左侧有一个看起来很特别的图标，它实际上是一个弹出菜单，可显示的子菜单包括最近的文件（Recent Files）、未保存的文件（Unsaved Files）、关联文件（Counterparts）、父类（Superclasses）、子类（Subclasses）、兄弟类（Siblings）、类别（Categories）、引用头文件（Includes），等等，通过子菜单可以看到所有与编辑器当前文件相关的其他代码。
- ❑ 弹出菜单图标的右边是一对左右箭头，分别可以跳转到上一个文件和下一个文件。
- ❑ 跳转栏包含一组分段式的弹出菜单，显示了在项目中找到当前所选文件的层级路径。你可以试着点击任意一个显示了组或文件名称的分段按钮，来查看位于同一层级的所有其他文件和组。最后一个分段按钮展示了当前所选文件中各项内容的列表。在图 2-15 中，你可以看到跳转栏最右边的弹出菜单显示了当前文件包含的所有方法和用符号代表的其他内容。跳转栏当前显示的文件是 BIDAppDelegate.m，弹出菜单列出了该文件中定义的各项符号。



图 2-15 Xcode 编辑器面板显示的跳转栏，编辑器面板中则显示了跳转栏中选择的源代码文件。跳转栏的弹出子菜单列出了当前文件的所有方法

跳转栏是非常强大的。在查看 Xcode 5 的各种界面元素时一定要留意它。

**提示** 与大部分的 OS X 应用一样，Xcode 5 也支持全屏显示功能。点击项目窗口右上角的全屏按钮，感受一下不会分散注意力的全屏编码模式吧！

### Xcode 键盘快捷键

如果你喜欢使用键盘快捷键进行导航，而不是使用鼠标进行屏幕控制的话，那么你肯定会喜欢 Xcode 提供的快捷键。Xcode 中的大部分常用操作都有对应的快捷键，比如 `command+B`

可以构建应用，`command+N` 可以创建新文件。

可以自由更改所有的 Xcode 快捷键，也可以在 Xcode 首选项的 Key Bindings 标签中为未指定快捷键的命令指定一个快捷键。

一个非常好用的快捷键是 `shift+command+O`，对应的是 Xcode 的 Open Quickly 功能。按下该快捷键后，键入一个文件名、设置项名称或者符号名，Xcode 就会显示一个选项列表。找到想要的文件后按下 `return` 键就可以在编辑器面板中打开这个文件，这样只需要进行几次键盘操作就能够快速切换文件。

#### 4. 实用工具面板

之前提到，Xcode 工具栏右侧倒数第二个按钮用于打开、关闭实用工具面板。和检查器面板类似，实用工具区域是一个上下文相关的检查器面板，它的内容会随着编辑器面板的显示内容而变化。实用工具区域的下方显示了一些不同类型的资源，你可以将它们拖动到你的项目中。本书有很多这样的例子。

#### 5. 界面构建器

Xcode 的早期版本包含一个被称为界面构建器（Interface Builder）的界面设计应用，用于在项目中构建和自定义用户界面。后来，Xcode 就把界面构建器集成到了工作区中。界面构建器不再是一个独立的应用了。这就意味着，在编写代码和设计界面时，不需要在 Xcode 和界面构建器之间反复切换了。这个功能变化已经是很多年前的事情了，不过对于包括我们在内，那些经历过独立版界面构建器应用日子的人来说，现在的界面构建器能直接集成到 Xcode，确实是件让人觉得非常愉悦的事情。

本书中的例子会大量使用 Xcode 的界面创建功能，并深入探讨其中的细节。事实上，本章稍后就会创建我们的第一个界面。

#### 6. 新的编译器和调试器

Xcode 4 最重要的变化之一是内部机制的变化：一个全新的编译器（compiler）和底层调试器（low-level debugger）。它们都比之前更快更智能。

过去几年苹果一直使用 GCC（GNU C Compiler）作为底层编译器。不过最近几年，苹果已经全面切换到新的 LLVM（Low Level Virtual Machine，底层虚拟机）编译器。LLVM 的代码生成速度远比传统的 GCC 快。除了代码生成速度快之外，LLVM 还知道更多与代码相关的信息，所以它能生成更智能、更精确的错误信息和警告。

Xcode 对 LLVM 也有很好的集成，后者为前者注入了强大的能量。Xcode 可以提供精确的代码补全功能，而且当要产生警告或者弹出修复建议菜单时，它可以对代码片段的实际意图作出更准确的猜测。这样就可以很容易地找到并修正符号名称拼写错误、括号匹配错误、分号遗漏等问题。

此外，LLVM 还提供了一个复杂的静态分析器（static analyzer），它可以扫描你的代码以查找各种潜在问题，包括 Objective-C 的内存管理问题。事实上，LLVM 在这方面确实相当智能，它可以为你处理大多数内存管理任务，不过前提是编写代码时要遵守一些简单的规则。下一章开始讨论之前提到过的 ARC（自动引用计数）功能。

### 2.1.2 深入研究项目

我们已经讨论了 Xcode 项目窗口，现在来看看 Hello World 项目中的文件。单击 8 个导航面板图标（位于工作区左侧）中最左边的那个图标（参见本章前面的“导航视图”一节）或者按下 `command+1`，切换到项目导航面板。

---

**提示** 可以用 `command+1` 到 `command+8` 之间的快捷键在 8 个导航面板之间进行切换。从最左边的图标开始，数字与图标排序一一对应，所以，`command+1` 对应项目导航面板，`command+2` 对应符号导航面板，以此类推，`command+8` 对应日志导航面板。

---

项目导航面板中的第一个条目就是项目名，本例为 Hello World。这个条目表示整个项目，可以做些与项目相关的配置。单击这个条目，就可以在 Xcode 的编辑器中编辑项目的很多配置项。不过现在不用担心那些与具体项目相关的设置。目前来说，保留默认设置即可。

回过头来看一下图 2-7。Hello World 左侧的倒三角是展开的，下面显示了如下一些子文件夹（在 Xcode 中称为组）。

- ❑ **Hello World:** 这是第一个文件夹，它总是以项目名来命名，你会把大量时间花在这个文件夹上。它包含了应用的大部分代码以及用户界面文件。可以在这个文件夹下随意创建子文件夹，从而更好地组织代码，甚至可以使用其他分组代替这个默认的文件夹（如果你想换个方式组织代码的话）。这个文件夹中的大部分文件都将留到下一章再介绍，但是有一个文件在下一节使用界面构建器时会用到，下一小节就会加以介绍。
  - **Main.storyboard:** 这个文件包含了项目主视图控制器用到的用户界面元素。
- ❑ **Supporting Files:** Hello World 文件夹内的某个文件夹，包含了项目中必需的非 Objective-C 类的源代码文件和资源文件。通常，不会在 Supporting Files 文件夹上花费太多时间。创建一个新的 iPhone 应用项目后，这个文件夹就会包含 4 个文件。
  - **Hello World-Info.plist:** 这是一个属性列表，包含应用相关的各种信息。2.3 节会简单介绍这个文件。
  - **InfoPlist.strings:** 这是一个文本文件，包含可能被信息属性列表引用到的可读字符串。不同于信息属性列表，这个文件可以被本地化，这样就能在应用中包含多种语言（第 21 章会介绍这个话题）。
  - **main.m:** 这个文件包含应用的 `main()` 方法。通常不需要编辑或修改这个文件。事实上，最好不要碰这个文件，除非你真的知道自己在做什么。
  - **Hello World\_Prefix.pch:** 这个文件包含项目中用到的所有外部框架的头文件（扩展名 `.pch` 代表 `precompiled header`，意思是预编译头文件）。通常，这个文件中包含的头文件并不是项目的一部分，也不会经常更改。Xcode 会预先对这些头文件进行编译，之后构建应用时就直接使用预编译过的版本，这样就减少了编译项目所需的时间（点击 **Build** 或 **Run** 可以对项目进行编译）。暂时不需要关心这个文件，因为它已经包含了最常用的头文件。

- ❑ **Hello WorldTests**: 如果你想要为你的应用编写一些单元测试代码, 这个文件夹就会包含所需的初始化文件。我们不会在本书中讲解单元测试的内容, 但是你创建的每个项目 Xcode 都会替你设置好这些事情, 这点很不错。与 Hello World 文件夹类似, 它也包含了自己的 Supporting Files 文件夹, 里面是构建并运行单元测试代码必须用到的文件。
- ❑ **Frameworks**: 框架是一种特殊的库, 其中可以包含代码, 也可以包含图像和声音之类的资源。添加到 Frameworks 文件夹中的框架或者库都会被链接到应用, 这样你的代码就可以使用这些框架或库中的类、函数和其他资源。最常用的框架和库默认都会被链接到项目中, 所以大多数情况下不必向这个文件夹添加任何东西。如果确实需要使用那些不常用的库和框架, 可以很容易地把它们添加到 Frameworks 文件夹中。第 7 章将介绍如何添加框架。
- ❑ **Products**: 这个文件夹包含构建项目时生成的应用。展开 Products 文件夹, 可以看到一个名为 Hello World.app 的文件, 这就是这个项目创建出来的应用。它还包含了一个名字为 Hello WorldTests.xctest 的文件, 它表示测试代码。这些文件都被称为构建目标 (build target)。由于我们还没有构建这个应用, 所以它们都显示为红色, Xcode 利用这种方式告诉你这个文件并不存在。

**注意** 导航面板区域中的“文件夹”并不一定与 Mac 文件系统上的文件夹一一对应。它们只是 Xcode 中的逻辑分组, 用于对所有的东西进行组织, 以便在应用开发时可以更快更容易地找到需要的内容。通常, 这两个项目文件夹中的文件都直接保存在项目目录下, 但是也可以把它们保存到其他位置, 如果你愿意的话, 甚至可以把它们放到项目文件夹外部。Xcode 内部的层次结构与文件系统的层次结构完全无关, 比如, 在 Xcode 中把一个文件移出 Classes 文件夹并不会改变这个文件在硬盘上的位置。

使用实用工具面板可以对分组进行配置, 可以为这个分组分配一个具体的文件系统目录。但是, 默认情况下, 项目中新加入的分组完全独立于文件系统, 这些分组的内容可以保存在任何地方。

## 2.2 界面构建器简介

在项目窗口的项目导航面板中展开 Hello World 组 (如果尚未展开), 然后选择 Main.storyboard 文件。然后这个文件就会在编辑器面板中打开 (参见图 2-16)。你将在纯白色背景的中央看到一个完全空白的 iPhone 屏幕, 可以在这个背景上编辑界面。这就是 Xcode 的界面构建器 (有时被称为 IB), 可以在这里设计应用的用户界面。

界面构建器历史悠久, 它于 1988 年面世, 曾用于开发 NeXTSTEP、OpenStep、Mac OS X 应用, 现在也用于 iPhone 和 iPad 等 iOS 设备。之前已经提到过, 界面构建器过去曾是一个独立的应用 (安装 Xcode 时会自动安装界面构建器), 它与 Xcode 协同工作。而现在, 界面构建器被完全集成到了 Xcode 中。

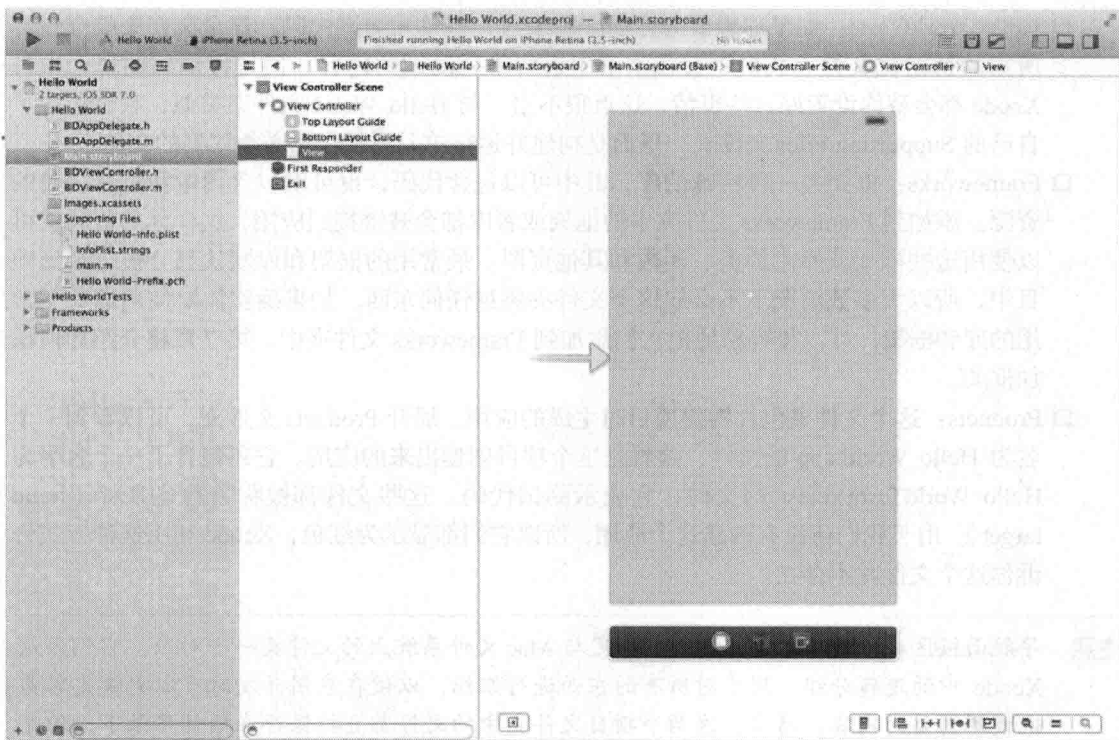


图 2-16 在项目导航面板中选择 Main.storyboard。这将在界面构建器中打开该文件。它看起来应该会是这样的

### 2.2.1 文件格式

界面构建器支持不同的文件类型：最初的版本使用扩展名为 `nib` 的二进制格式，后来改用扩展名为 `xib`，这是一种基于 XML 的衍生格式。两者包含了完全一样的文档内容，但 `xib` 格式的版本却是基于文本格式的，这样做有很多优势，尤其是在你使用版本控制系统的时候。

**注意** iOS 项目模板全都默认使用 `.xib` 扩展名，但是在最初的 20 年里，所有的界面构建器文件都使用 `.nib` 扩展名，结果就是大部分开发者都把界面构建器文件称为“`nib` 文件”。不管文件实际使用的是 `.xib` 扩展名还是 `.nib` 扩展名，都被称为“`nib` 文件”。事实上，苹果公司仍然在其文档中使用“`nib`”和“`nib 文件`”这两个术语。

一个 `nib` 文件可以包含任意数量的对象。但在 iOS 项目中，通常每个 `nib` 文件负责包含一个视图（一般都是全屏视图）以及相关联的控制器或对象。用这种方式对应用进行划分，只在某个视图需要显示的时候加载它的 `nib` 文件。这样做能为内存受限制的 iOS 设备节约应用运行时的内存。



最近几年, 界面构建器开始支持另一种文件类型, 即分镜 (storyboard)。你可以将分镜想成一个“元 nib 文件” (meta-nib file), 因为它可以包含多个视图和控制器, 以及如何在应用运行时进行相互连接的配置信息。与一次就加载完所有内容的 nib 文件不同, 分镜不会一次加载所有内容, 而是在你需要加载某视图和控制器时向它请求特定内容。

在本书中, 分镜和 nib 文件我们都会使用到。在当前的示例中, Xcode 为我们创建了一个分镜, 所以我们先来看看它的用法。

你现在研究的是将来构建 iOS 应用用户界面的首选工具。现在, 假设你想创建一个按钮实例。你可以通过编写代码来创建这个按钮, 不过更简洁的方法是从库中拖出一个按钮用来创建界面对象并指定它的各项属性, 这与在运行时创建的按钮完全一样。

Main.storyboard 文件会在应用启动时自动进行加载 (目前不需要关心这是如何实现的), 在这里添加对象就可以组成应用的用户界面。在界面构建器中创建的对象会在程序加载分镜或 nib 文件时被实例化。本书有很多这样的例子。

### 2.2.2 分镜

每个分镜都是由一组或多组相对应的视图和控制器构成的。视图就是你眼睛能看到并可以在界面构建器中进行编辑的部分, 而控制器则是你编写的应用代码, 用来处理用户的交互事件。应用的实际操作都是在控制器内执行的。

在界面构建器中, 你经常会看到一个表示 iPhone 屏幕尺寸 (虽然也可能是其他尺寸) 的矩形视图, 我们当前的示例也同样如此。点击矩形的任意位置, 你将看到底下一行出现了三个图标。把鼠标悬停在每个图标上面, 你会看到弹出的工具提示显示了它们各自的名称: View Controller (视图控制器)、First Responder (第一响应者) 和 Exit (离开)。目前请先忽略掉 Exit, 把注意力集中到更重要的两项上。

- ❑ 视图控制器代表会从某存储文件中加载控制器以及相关联的视图。

- ❑ 第一响应者简单地说, 就是用户当前正在进行交互的对象。如果用户正在向一个文本框中输入数据, 那么这个文本框就是当前的第一响应者。第一响应者会随着用户与用户界面的交互而变化, 而通过 First Responder 图标则可方便地与当前作为第一响应者的控件等对象进行通信, 不需要编写代码来判断到底哪个控件 (或视图) 是当前第一响应者。

从下一章开始会详细介绍这些对象, 所以你现在不必担心搞不清楚何时使用第一响应者以及视图控制器是如何加载的。

除了这些图标以外, 你在编辑区能看到的其余部分就是用来放置图形对象的空白区域。不过在了解它之前, 你需要注意界面构建器的编辑器中还有一个层级视图区域。点击编辑区左下角的小按钮, 你将会看到层级视图从左侧滑动出来。这里显示分镜的所有内容, 并由相关的场景 (scene) 作为容器来进行划分。在本示例中只有一个场景, 它的名称是 View Controller Scene。你会看到它包含了一个名称为 View Controller 的子项, 往下依次又包含了一个名称为 View 的子项 (还有其他一些内容, 你将在后面学到)。通过这种方式可以很方便地浏览全部的内容。你在主编辑区域看到的所有内容都能在这里找到。

View 图标代表 UIView 类的一个实例。UIView 对象是一块用户能够看到并与之交互的区域。本例只有一个视图，所以这个图标代表这个应用中用户能够看到的全部内容。之后我们会创建更为复杂的多视图应用。目前来说，就认为这是用户使用应用时能够看到的全部内容即可。

**注意** 从技术角度来说，这个应用实际上包含多个视图。屏幕上显示的所有用户界面元素（包括按钮、文本框、标签等）都继承自 UIView。但是，本书使用的术语视图（view）仅仅指 UIView 的实例，所以称本应用只有一个视图。

点击 View 图标，Xcode 会自动对我们之前所说的 iPhone 尺寸屏幕的矩形进行高亮。在这里可以使用图形化的方式设计用户界面。

### 2.2.3 库

工作区右侧的实用工具视图被分为了两部分，如图 2-17 所示。如果没有看到这个实用工具视图，可以单击工具栏上三个 View 按钮中最右边的那个，选择 View>Utilities>Show Utilities，或者按下 option+command+0。

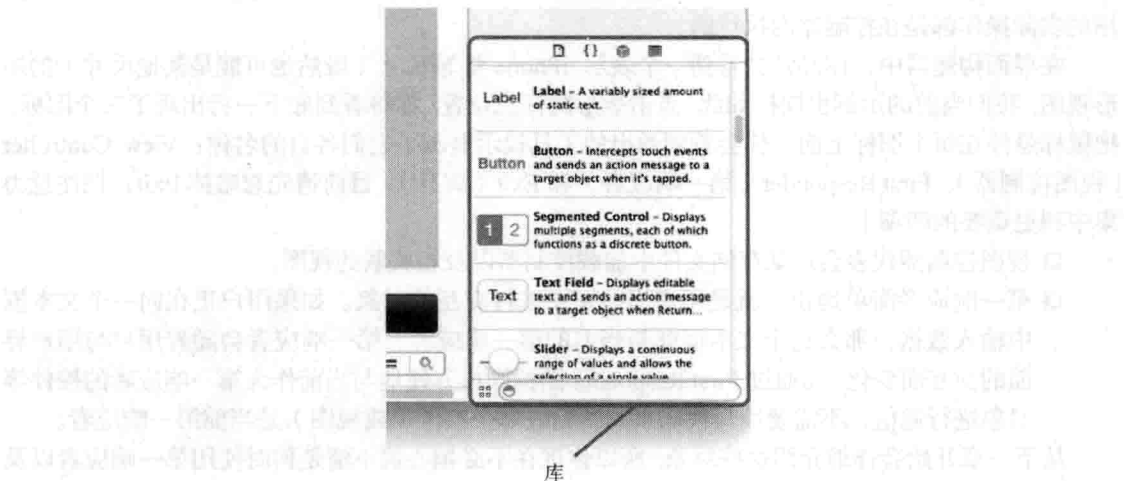


图 2-17 库包含了 UIKit 内置的各种对象，可以在界面构建器中使用它们。

库与工具栏之间的部分统称为检查器

实用工具视图的下半部分称为库面板（library pane），或者简称为库。库是可重用对象的集合，可以在自己的程序中使用它们。库面板顶部工具栏中的 4 个图标将它分成了 4 个部分。

❑ 文件模板库（file template library）：这部分包含一些文件模板，向项目中添加新文件时可以使用它们。例如，如果要向项目中添加一个新的 Objective-C 类，就可以从文件模板库中拖出一个 Objective-C 类文件。

- ❑ **代码片段库 (code snippet library)**: 这部分包含一些精选的代码片段, 可以直接把它们拖到源代码中使用。记不住 Objective-C 的快速枚举语法? 没关系, 就从库里拖出这个特定的代码片段, 不需要再去翻书了。写了一些希望以后能够再次使用的代码? 在文本编辑器中选中想要的代码, 然后把它拖到代码片段库中就行了。
- ❑ **对象库 (object library)**: 这部分包含各种可重用对象, 比如文本框、标签、滑块、按钮等可以用来设计 iOS 界面的任何对象。本书的示例程序会大量使用对象库来创建界面。
- ❑ **媒体库 (media library)**: 顾名思义, 这个库包括用户的所有媒体文件, 有图片、声音和影片文件等。

---

**注意** 对象库中的对象主要来自于 iOS 的 UIKit 框架, 这个框架中包含的对象可用于创建应用的用户界面。UIKit 在 Cocoa Touch 中的作用与 AppKit 在 Cocoa 中的作用相同。这两个框架在概念上很相似, 然而由于平台之间的差异, 它们也存在很多明显的不同。不过, NSString、NSArray 等属于 Foundation 框架的类, 是 Cocoa 和 Cocoa Touch 共有的。

---

注意库面板底部的搜索框。想找一个按钮控件? 那就在搜索框里输入 button, 这时库会只显示名字中含有“button”的项。搜索完成后记得要清空搜索框。

## 2.2.4 在视图中添加标签

现在试着使用界面构建器。单击库顶部的对象库图标 (看起来像个立方体) 打开对象库。向下滚动, 在库中寻找 Table View。继续滚动, 找到它了! 哦, 等等, 有个更好的方法: 只要在搜索框里键入 Table View 就可以了, 这不是更容易吗?

---

**提示** 按下快捷键 `control+option+command+3` 就能跳转到搜索栏, 并且高亮显示搜索框的内容。接下来你就可以输入想要查找的内容了。

---

在库中找到 Label。它大约位于列表的顶部。然后, 把标签控件拖放到之前介绍过的视图中。(如果在编辑器面板中看不到视图, 就在界面构建器的 dock 中单击一下 View 图标。) 当把光标移到视图上面时, 光标就会变成一个绿色的加号指示符 (在 Finder 中它表示“我正在复制某些内容”)。把标签拖到视图中央。标签位于视图中央时会看到两条蓝色的引导线, 一条垂直、一条水平。标签是否居中并不重要, 重要的是知道引导线的存在。图 2-18 显示了在释放鼠标之前工作区的情况。

用户界面中的对象是按照层次关系存储的。大多数视图都可以包含子视图, 当然也有例外, 按钮和其他很多控件就不能包含子视图。界面构建器很智能, 如果一个对象不接受子视图, 那就无法把其他对象拖到它上面。

把标签直接从库里拖动到正在编辑的视图中, 就能将其作为子视图添加到主视图 (名为 View 的视图) 中, 当主视图显示在用户面前时子视图会自动显示出来。从库中将一个 Label 拖到 View 视图中, 实际上是在应用主视图上添加了一个 UILabel 的实例作为子视图。



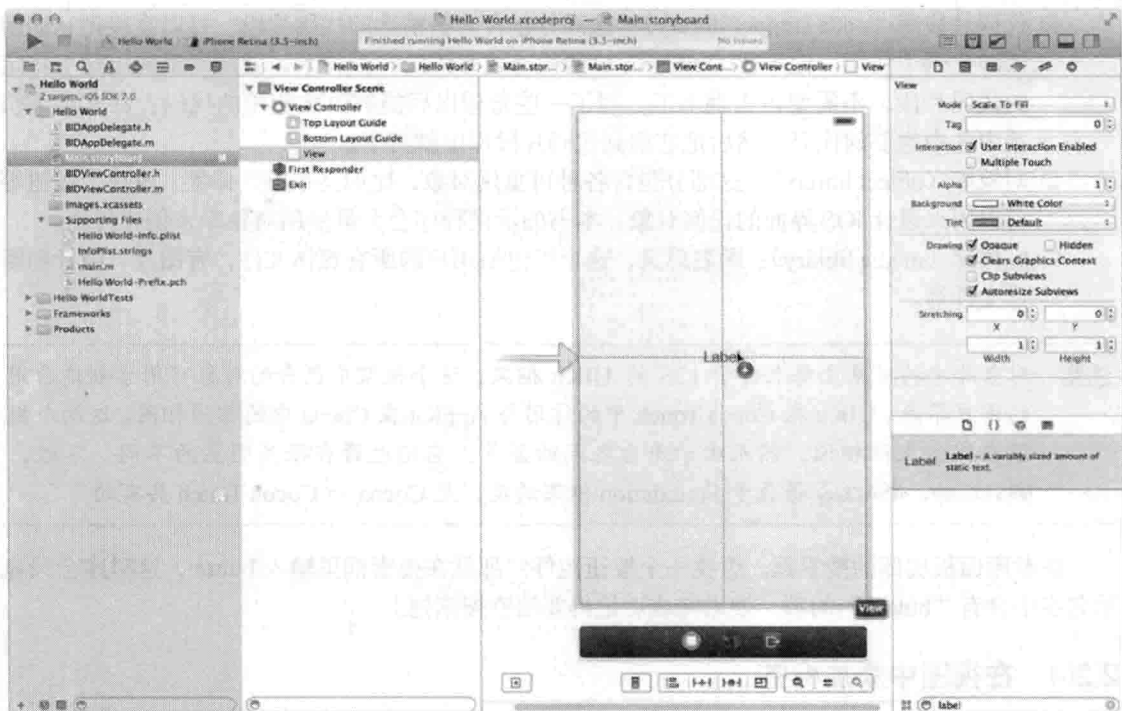


图 2-18 在库中找到标签并把它拖到视图中。注意，我们在库的搜索框里键入了 label，从而限制了对象列表只显示名称中含有 label 的对象

现在我们来编辑这个标签，让它显示一些有意义的内容。双击刚才创建的标签，键入文本 Hello, World!。在标签外点击鼠标，然后再重新选中标签，重新把标签拖到视图中央，或是放到屏幕上你希望的任何地方。

你知道吗？只要保存项目，这个应用就完成了！选择 **File**►**Save**（或者按下 **command+S**）。然后查看 Xcode 项目窗口左上方的弹出菜单，事实上它是一个多分段的控件。左侧用来选择编译目标以及其他一些设置，不过我们需要注意的是右侧，你可以在这里选择想要运行的设备。点击右侧按钮你将会看到可用的设备列表。如果你已经连接了任意 iOS 设备并做好准备的话，它就会出现在列表顶端。不然只会看到一个无意义的 **iOS Device** 选项。在它下面有一整段标题为 **iOS Simulator** 的列表，里面列出了你可以在 iOS 模拟器中使用的所有型号的设备。选择 **iPhone Retina (4 inch)**，这样我们的应用就可以在这个模拟器中以 iPhone 5 的配置运行了。如果参与了苹果公司的付费 iOS 开发者计划，还可以尝试在真实的 iPhone 上运行应用。本书尽量只使用模拟器，因为在模拟器中运行不需要任何费用。

准备好运行这个应用了吗？选择 **Product**►**Run**（或者按下 **command+R**）。Xcode 会编译这个应用并且在 iPhone 模拟器中启动它，如图 2-19 所示。



图 2-19 iPhone 中的“Hello, World!”程序

**注意** 在构建并运行应用时，如果已经有 iOS 设备连接到 Mac，情况可能会与上面讲的有所不同。总而言之，如果想要在 iPhone、iPad 或者 iPod touch 上构建并运行应用，就必须注册苹果公司的 iOS 开发者计划并支付一些费用，然后还要对 Xcode 做一些恰当的配置。加入开发者计划后，苹果公司会提供一些必要的信息，指导你完成在真实设备上运行应用的配置工作。不过，本书的大部分程序都能很好地在 iPhone 或者 iPad 模拟器中运行。

欣赏完你自己的作品后，就可以返回 Xcode 了。Xcode 和模拟器是两个互相独立的应用。

**提示** 检查完应用后，可以退出模拟器，但是可能随时需要再次启动它。如果让模拟器一直运行，那么要求 Xcode 再次运行应用时，Xcode 就会询问是否要先终止当前正在运行的应用再启动新的实例。如果这让你感到困惑，那就在每次测试完应用后退出模拟器吧。没人知道！

等一下！这就完成了？可是还没有编写代码呢！是的，已经完成了！  
是不是很赞？

如果想要修改标签的某些属性（比如字号或者字体颜色），应该如何实现呢？要编写代码吗？不需要！接下来就来看看修改属性是件多么容易的事。

### 2.2.5 属性修改

返回 Xcode，单击 Hello World 标签以选中它。现在把注意力转向库面板上方的区域。面板

的这一部分称为检查器。与库类似，检查器面板顶部也有一些图标，点击图标就可以切换检查器以显示特定类型的数据。要改变标签的属性，可以点击左起第四个图标，这样就会切换到对象属性检查器，如图 2-20 所示。

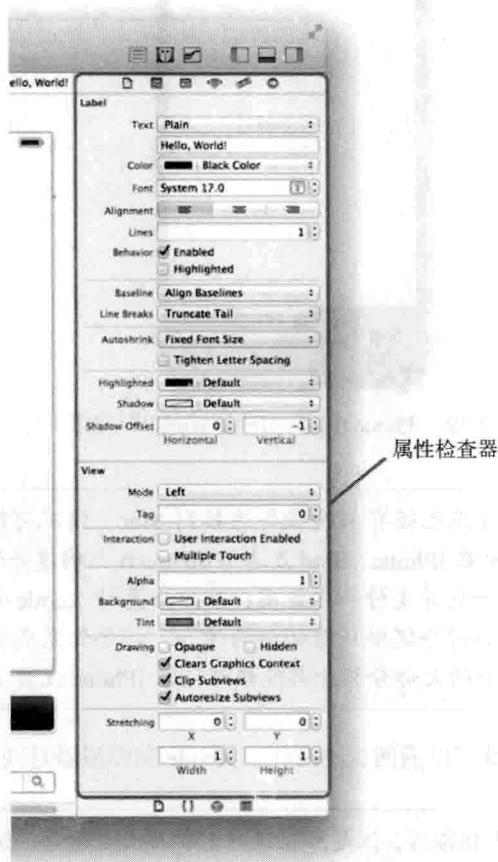


图 2-20 在对象属性检查器中查看标签属性

**提示** 检查器面板与项目导航面板类似，每个图标都有对应的键盘快捷键。option+command+1 对应检查器最左边的图标，option+command+2 对应第二个图标，以此类推。与项目导航面板不同的是，检查器面板中的图标个数会根据在导航面板或编辑器中选择的对象而改变。

接下来根据自己的喜好来改变标签的外观。可以随意修改文本的字体、字号以及颜色。注意，如果加大了字号，可能需要调整标签大小以容纳更大的文本。完成后，保存文件，再次选择 Run。刚才所做的修改就会在应用中显现出来了，同样，这次也没有编写任何代码。

**注意** 不用太过担心对象属性检查器中所有字段的含义，如果某一项更改没有生效也不用着急。在学习本书的过程中，能够学到很多对象属性检查器相关的内容，以及每个字段的作用。

界面构建器支持以图形化的方式设计用户界面，从而使你能够专注于编写具体应用的代码，而不用花时间编写冗长的代码来构建用户界面。

大多数现代应用开发环境都提供了一些工具，支持以图形化的方式构建用户界面。界面构建器与其他很多工具的一个区别就是，界面构建器不会生成任何需要手动维护的代码。事实上，界面构建器创建的是 Objective-C 对象（与在代码中所做的一样），然后把这些对象序列化到分镜或 nib 文件中，以便在运行时把它们直接加载到内存中。这样做避免了很多与代码生成相关的问题，总而言之，这是个更好更强大的方法。

## 2.3 画龙点睛——美化 iPhone 应用

现在只剩下最后一步，对应用进行美化，使它更像一个真实的 iPhone 应用。首先，运行项目。模拟器窗口出现后，按一下 iPhone 的主屏幕（窗口底部带有白色方框的黑色按钮）。这样就返回到 iPhone 的主屏幕了，参考图 2-21 中的内容。是不是觉得有点单调？

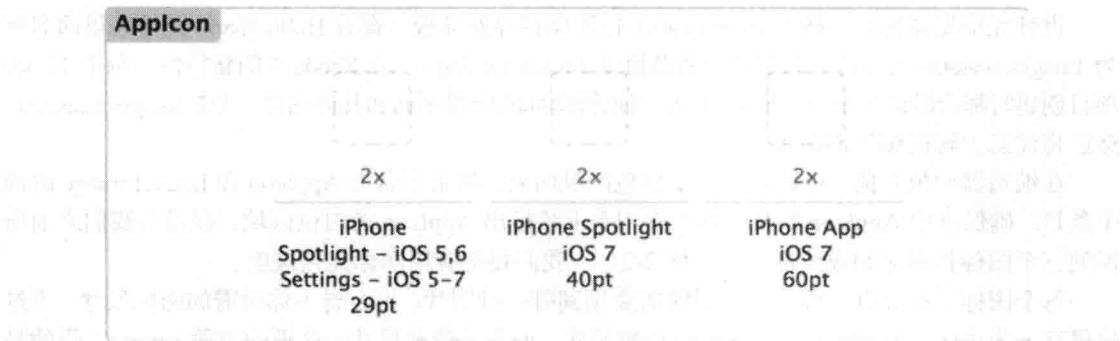


图 2-21 在项目资源目录中的 Applcon 面板。可以在这里设置你的应用图标

看一下屏幕顶部的 Hello World 图标。这样的图标真是拿不出手，是吧？要解决这个问题，需要创建一个图标并将它保存为.png（Portable Network Graphic，便携式网络图像）文件。事实上，最佳的方案是创建两个图标。一个大小为 120 像素×120 像素，另一个为 80 像素×80 像素，此外还需要 58 像素×58 像素。为什么要三个图标呢？第一个用来在 iPhone 的主屏幕上显示你的应用。第二个会在 iPhone 上使用 Spotlight 搜索到应用时出现。第三个则显示在“设置”应用中。如果你没有提供较小的图标，就会对大图标进行适当的缩放。不过为了最优效果，你（或者是你团队中的美工）需要预先缩放好大小。

---

**注意** 图标尺寸的问题要更复杂一些，在 iOS 7 之前，所有 iPhone 的图标尺寸都是 114 像素 × 114 像素。但如果还想支持老款的非视网膜屏的 iPhone，就还需要包含一个缩小一半的图标，57 像素 × 57 像素。iPad 上的应用也存在这种问题，无论是否配置有视网膜屏，iOS 7 和早先的 iOS 系统所用的图标尺寸也不相同。所以，我们现在避免讨论这个问题，只针对运行 iOS 7 的 iPhone 配置图标尺寸。

---

创建图标时不需要模仿 iPhone 上已有的图标风格，iPhone 会自动把图标调整为圆角矩形。只需要创建一个普通的正方形图像。如果不想创建自己的图标，可以直接使用我们提供的一组图标，它们位于项目归档文件 02-Hello World-icons 文件夹下，这些图片分别叫做 icon-120.png、icon-80.png 和 icon-58.png。

---

**注意** 必须使用 .png 图像作为应用的图标，实际上 iOS 项目中所有图像都应该使用这种格式。Xcode 在构建应用时会自动优化 .png 图像，这使 .png 格式成为 iOS 应用中最快、最有效的图像格式。尽管大多数其他的常用图像格式也可以正确显示，但是，除非理由特别充分，否则都应该使用 .png 文件。

---

设计完应用图标后，按下 `command+I` 打开项目导航面板，查看 Hello World 文件夹里面名字为 Images.xcassets 的文件。它被称为资源目录 (asset catalog)，是 Xcode 5 的新特性。每个 Xcode 项目创建时都会默认生成一个资源目录，用来管理你的应用图标和其他图片。选中 Images.xcassets，然后将注意力转向编辑器面板。

在编辑器面板左侧，可以看到一个白色的纵向列，列出了名为 AppIcon 和 LaunchImage 的两个条目。确保选中 AppIcon 条目。你会看到左上角标出 AppIcon 的白色区域，以及与我们之前所说的三个图标相对应的虚线框（参见图 2-21）。我们要把应用图标拖到这里。

每个图标下面都有一段文字来说明需要用到哪一种图片。也会告诉你所需的图标尺寸。不过这里有一个误区：Xcode 显示给你的是点数尺寸，而不是像素尺寸。这里的点数 (point) 指的是屏幕上特殊的尺寸。在旧的 iPhone 设备 (iPhone4 之前的所有手机) 以及 iPad 一代、iPad 2 和 iPad mini 上一点代表单个像素。而在其他 Retina 屏幕的设备上，一点表示的实际上是一个 2 像素 × 2 像素的格子。资源目录中使用写有 2× 的标签来进行提示，不过也只是标签而已。如果要算出某一项实际需要的尺寸，可以选中它并按下 `option+command+4` 在窗口右侧打开属性检查器。在此处能够同时看到 size (单位还是点数) 以及 scale 值。一部分图标在此处都会显示 2×。将尺寸与缩放值相乘，你将得到实际需要的像素尺寸大小。分别查看 AppIcon 面板中的每一项，检查器都会提供相关的详细信息。这些内容与我们之前所说的一致，但你永远不会知道苹果下一步打算做什么。从开始印刷直到你阅读本书为止的这段时间，苹果可能又发布了一些奇妙的设备，也许会增加所需要的图标。

把 icon-120.png 从 Finder 拖到标注为 iPhone App 的方框中 (应该位于右侧)。这样就把

icon-120.png 复制到项目中并将其设置为应用的图标了。接下来，再把 icon-80.png 拖到中间的方框，把它设置为 Spotlight 的图标。最后拖动 icon-58 到左边的方框内，它将成为 iOS 7 中“设置”程序中显示的图标。

现在，编译并运行应用。模拟器启动之后，按下带有白色方框的按钮回到主屏幕，现在可以看到漂亮的新图标了（参见图 2-22）。如果想要看到更小的图标，请在主屏幕上向下轻扫以调出 spotlight 搜索文本框，并输入单词 Hello，你将会立即看到新的应用图标。

---

**注意** 如果想把旧的应用从 iOS 模拟器的主屏幕上清除，可以从 iOS 模拟器的应用菜单中选择 iOS Simulator>Reset Content and Settings...。

---



图 2-22 现在应用有一个漂亮的图标了

## 2.4 小结

现在可以缓口气了。本章可能并没有做太多事情，但是也确实介绍了不少基础知识。你学习了 iOS 项目模板，了解了如何创建应用，掌握了很多 Xcode 5 相关内容，也开始使用界面构建器了，最后还学习了如何设置应用的图标。

但是，这个 Hello World 程序是一个完完全全的单向应用。我们只是向用户显示一些信息，却没能得到任何用户输入。让我们进入下一章，看看如何在 iOS 设备上获取用户输入，并根据用户输入进行相应的操作。准备好了吗？做个深呼吸，然后翻到下一页。

上一章的 Hello World 应用程序很好地展示了如何使用 Cocoa Touch 进行 iOS 开发,但它缺少了一个至关重要的功能——与用户交互。如果不能与用户交互,应用的功能会严重受限。

本章将编写一个稍微复杂一点儿的应用,它有两个按钮和一个标签,如图 3-1 所示。用户按下一个按钮时,标签上的文本会相应地改变。这看上去是一个相当简单的示例,但它展示了在 iOS 应用中实现交互功能所需的关键概念。为了增加趣味性,本章还会介绍如何使用 `NSAttributedString` 类,通过这个类可以对很多 CocoaTouch 的 GUI 元素使用带样式的文本 (styled text)。

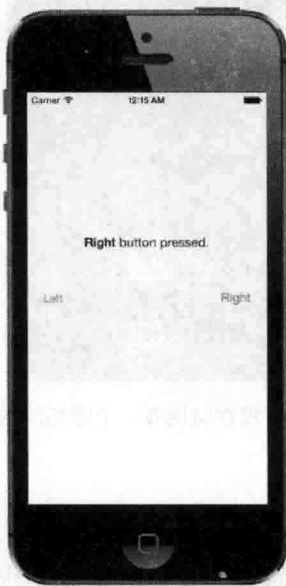


图 3-1 本章将构建的简单应用程序,它有两个按钮

### 3.1 MVC 方法

在深入学习之前,先介绍一些基本理论。Cocoa Touch 设计者们采用 MVC 模式 (Model-View-

Controller, 模型-视图-控制器)作为指导原则。在基于图形用户界面的应用程序中,使用 MVC 可以非常合乎逻辑地对代码进行拆分。目前,几乎所有面向对象编程框架都在一定程度上借鉴了 MVC 的设计理念,但很少有像 Cocoa Touch 这样忠实于 MVC 的。

MVC 模式把代码功能划分为 3 个不同类。

- 模型: 保存应用程序数据的类。
- 视图: 包括窗口、控件以及其他一些用户可以看到并能与之交互的元素。
- 控制器: 把模型和视图绑定在一起的代码, 包括处理用户输入的应用程序逻辑。

MVC 的目标是最大限度地分离这 3 类代码。创建的任何对象都应该非常清晰明确, 让人一看便知这个对象所属的分类(模型、视图或控制器), 尽量不要包含那些可能被认为属于多个分类的功能。例如, 实现按钮的对象不应该包含按钮点击时处理数据的代码, 实现银行账户的对象不应该包含绘制表格以显示交易数据的代码。

MVC 可以帮助确保代码的最大可重用性。一个实现通用按钮的类可以在任何应用程序中使用, 否则, 如果实现按钮的类要在点击按钮时进行一些特定计算, 那这个类就只能在最初实现它的应用程序中使用。

编写 Cocoa Touch 应用程序时, 主要使用界面构建器以可视化的方式创建视图组件, 但有时仍然需要在代码中修改(甚至创建)用户界面。

创建模型时, 可以编写一个 Objective-C 类, 保存应用程序数据, 也可以使用 Core Data 构建一个数据模型(第 13 章会介绍 Core Data)。本章的应用程序不会创建任何模型对象, 因为我们不需要保存数据。后面的章节会介绍模型对象, 以实现更复杂的应用程序。

控制器组件通常由应用程序的具体类组成。控制器可以是完全自定义的类(NSObject 子类), 但在多数情况下控制器是 UIKit 框架提供的那些通用控制器类(比如 UIViewController, 稍后就会介绍)的子类。这么说吧, 通过继承一个已有的类, 可以免费获取大量的实用功能, 这样就不用再花时间重复造轮子了。

随着对 Cocoa Touch 的深入学习, 很快就可以看到 UIKit 框架中的类遵循 MVC 原则的情况。在开发时牢记这个概念, 就能够创建出简洁而易于维护的代码。

## 3.2 创建项目

现在开始创建下一个 Xcode 项目。这个项目与上一章的项目使用相同的模板 Single View Application。从这个简单的模板入手, 更容易理解视图和控制器之间的协作。后面的章节会陆续介绍其他一些模板。

启动 Xcode, 选择 File>New>New Project...(或者按下 shift+command+N)。选择 Single View Application 模板, 然后点击 Next。

然后会看到一个与上一章相同的选项表单。在 Product Name 字段中填入 Button Fun 作为这个新应用程序的名字。Organization Name、Company Identifier 以及 Class Prefix 这三个字段的值默认与上一个项目相同(Apress、com.apress、BID), 可以不用管它们。



与“Hello, World”一样,我们要编写的是个 iPhone 应用程序,所以在 Devices 中选择 iPhone。最终的选项表单如图 3-2 所示。

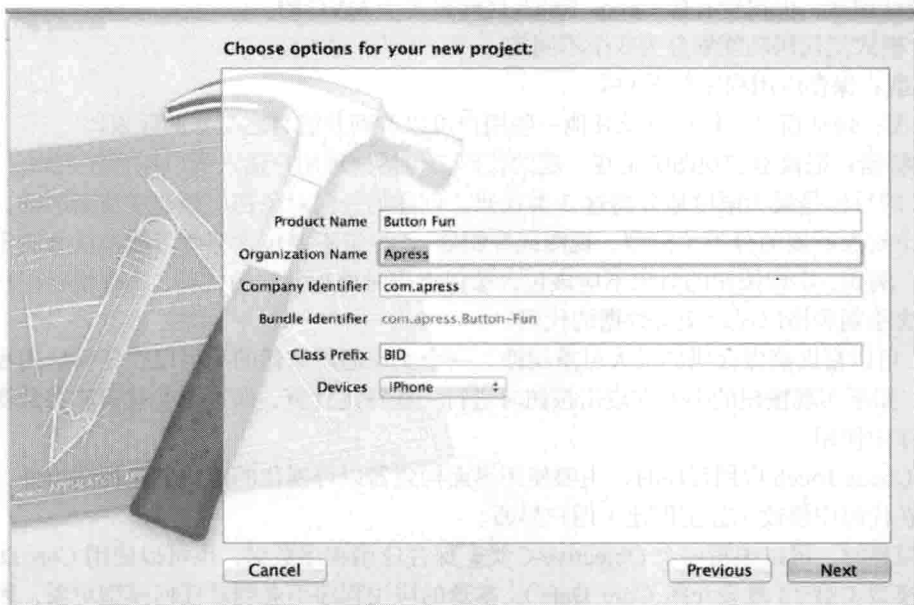


图 3-2 项目命名和项目选项

点击 Next, Xcode 会提示你选择项目的保存位置。可以把这个项目与本书其他项目保存在一起。对于 Create git repository 选项,可以根据自己的喜好来决定是否选择。

### 3.3 视图控制器

与上一章一样,本章稍后会使用界面构建器为应用程序设计一个视图(或者说是用户界面)。在此之前,先来看看那些 Xcode 自动创建的源代码文件,需要对它们进行一些修改。这一章终于要开始编写代码了。

开始修改之前,先来看看这些已经创建好的文件。在项目导航面板中,Button Fun 分组应该已经展开了(如果还没有展开,可以点击旁边的三角形按钮展开它),如图 3-3 所示。

Button Fun 文件夹包含 4 个源代码文件(以.h 或者.m 结尾)、一个 storyboard 文件和一个包含了应用程序所需全部图片的 xcassets 文件。这 4 个源代码文件实现了应用程序所需的两个类:应用程序委托(application delegate),以及用于这个应用仅有的一个视图的视图控制器。再次提醒一下, Xcode 为所有的类都自动添加了之前指定的类名前缀(BID)。

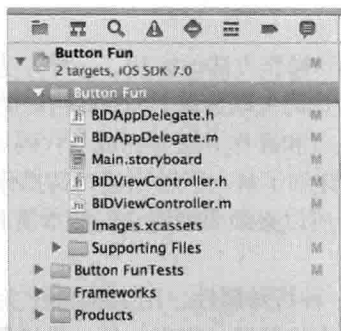


图 3-3 项目导航面板中显示了项目模板自动创建的类文件。注意，类文件的名称前都自动添加了类名前缀

本章稍后会介绍应用程序委托。先来看看 Xcode 自动创建的视图控制器类。

名为 `BIDViewController` 的控制器类负责管理应用程序的视图。名称中的 `BID` 部分就是之前指定的类名前缀，名称中的 `ViewController` 部分则表明这个类是一个视图控制器。点击分组和文件（Groups & Files）面板中的 `BIDViewController.h`，就可以查看这个类的头文件了：

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

```
@interface BIDViewController : UIViewController
```

```
@end
```

没什么内容，是吧。`BIDViewController` 是 `UIViewController` 的子类。`UIViewController` 是之前提到的通用控制器类中的一个，它是 `UIKit` 框架的一部分，通过继承这个类，可以获得大量的常用功能。Xcode 并不知道我们的应用程序会有哪些功能，但它知道应用程序必须具备哪些功能，所以它创建了这个类，可以在此基础上手动编写应用程序的具体功能。

### 3.3.1 输出接口和操作方法

第 2 章曾经使用 Xcode 的界面构建器创建了一个简单的用户界面。刚才也看到了视图控制器类的框架。一定有某种方式可以让视图控制器类中的代码与分镜文件中的对象交互，对不对？

当然可以！控制器类可以通过一种名为输出接口（outlet）的特殊属性来引用分镜或 nib 文件中的对象。可以把输出接口看做指向用户界面中对象的指针。例如，假设在界面构建器中创建了一个文本标签（像第 2 章那样），希望可以在代码中改变该标签的文本。声明一个输出接口，并且把它与标签对象关联起来，就可以在代码中使用这个输出接口来修改标签的显示文本了。本章会详细介绍这个方法。

另一方面，可以对 nib 文件中的界面对象进行设置，以触发控制器类中的某些特殊方法。这些特殊方法称为操作方法（action method），或者简称为操作（action）。例如，可以在界面构建器中进行设置，当用户点击一个按钮时，就调用代码中的某个相关操作方法。甚至还可以在界面构建器这样设置：当用户第一次触碰一个按钮时调用某个操作方法，当手指离开该按钮时调用另一

个操作方法。

Xcode 支持多种创建输出接口和操作方法的方法，一种方法是在源代码里先设定好，然后再使用界面构建器与它们与相应的代码关联起来。Xcode 的辅助视图提供了一种更快更直观的方式，只需要一步就可以创建输出接口和操作方法并完成与代码的关联。稍后就会介绍这个方法。不过，在进行关联之前，还需要再详细了解一下输出接口和操作方法。输出接口和操作方法是创建 iOS 应用的两个最基础的模块，所以必须要理解它们的本质和原理。

### 1. 输出接口

输出接口是 Objective-C 类的一种特殊属性，用 `IBOutlet` 关键字声明。输出接口可以在控制器类的头文件中声明，或者也可以在控制器的实现文件的类扩展（class extension）中的某个特定部分进行声明，如下所示：

```
@property (weak, nonatomic) IBOutlet UIButton *myButton;
```

这个例子声明了一个名为 `myButton` 的输出接口，可以让它指向界面构建器中的任何按钮。

`IBOutlet` 并不属于 Objective-C 内置的关键字，仅仅是一个简单的位于系统头文件中的，用 C 写成的预处理指令，它的定义如下所示：

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

是不是感到很困惑？对于编译器来说，`IBOutlet` 什么作用都没有。它唯一的作用就是告诉 Xcode，这个属性会与 nib 文件中的对象进行关联。任何需要与 nib 文件中的对象进行关联的属性，都必须使用 `IBOutlet` 关键字进行声明。幸好，Xcode 能够自动创建输出接口。

## 输出接口的变化

随着时间的推移，苹果公司改变了输出接口的声明方式和使用方式。鉴于有时可能会遇到比较老的代码，所以一起来看看输出接口的变化吧。

在本书的第 1 版中，我们需要为输出接口同时声明属性和相应的实例变量。那时，属性是 Objective-C 语言中的一个新概念，必须要为属性声明相应的实例变量，请看：

```
@interface MyViewController : UIViewController
{
    UIButton *myButton;
}
@property (weak, nonatomic) UIButton *myButton;
@end
```

在那时，我们是在实例变量的声明前使用 `IBOutlet` 关键字，就像这样：

```
IBOutlet UIButton *myButton;
```

当时苹果的示例代码使用的就是这种方式，这也是 `IBOutlet` 关键字在 Cocoa 和 NeXTSTEP 中的常规用法。

到编写本书第 2 版时，苹果把 `IBOutlet` 关键字从实例变量的声明移到了属性声明中，这

也是目前的标准做法，如下所示：

```
@property (weak, nonatomic) IBOutlet UIButton *myButton;
```

尽管这两种方法都可行（现在仍是如此），但本书遵循苹果的声明方式修改了书中的代码，把 `IBOutlet` 关键字从实例变量的声明中移除，放到了属性声明中。

近年来，苹果把默认的编译器从 GCC 改为了 LLVM（Low Level Virtual Machine，底层虚拟机）之后，就不再需要为属性声明相应的实例变量了。对于一个属性来说，如果 LLVM 找不到与之对应的实例变量，它就会自动创建一个实例变量。因此，在本书这一版中，不再专门为输出接口声明实例变量。

所有的这些方法实际上都在做同一件事，那就是让界面构建器知道输出接口的存在。苹果目前推荐的方式是把 `IBOutlet` 关键字放在属性声明中，所以我们遵循这一方式。希望你能了解这些历史背景，以防遇到比较老的代码（`IBOutlet` 关键字出现在实例变量的声明中）而迷惑不解。

要了解 Objective-C 属性的更多信息，可以阅读由 Scott Knaster、Waqar Malik 和 Mark Dalrymple 合著的《Objective-C 基础教程（第 2 版）》，或者是苹果开发者网站上的文档“Introduction to the Objective-C Programming Language”（网址是 <http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC>）。

## 2. 操作方法

简单来说，操作方法是一种返回类型为 `IBAction` 的特殊方法，返回类型 `IBAction` 告诉界面构建器这个方法可以被 nib 文件中的控件触发。操作方法的声明通常如下所示：

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

或者是：

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

操作方法的名称并没有什么特殊要求，可以使用任何你喜欢的方法名称，但是返回类型必须是 `IBAction`，实际上这相当于把返回类型声明为 `void`。返回类型为 `void` 的方法不返回任何值。而且，操作方法要么不接受任何参数，要么只接受一个参数（通常命名为 `sender`）。`sender` 参数是一个指针，在操作方法被调用时，`sender` 指向触发该方法的对象。例如，如果用户按下某个按钮时触发了这个操作方法，那么 `sender` 就指向这个被按下的按钮。`sender` 参数的存在使一个操作方法可以对多个控件作出响应。可以通过 `sender` 参数来确定到底是哪个控件触发了这个操作方法。

**提示** 事实上，还有一种不常用的 `IBAction` 声明方式，如下所示：

```
- (IBAction)doSomething:(id)sender  
    forEvent:(UIEvent *)event;
```

下一章会介绍控件事件。

如果声明了一个带有 `sender` 参数的操作方法，方法内部却没有使用到这个参数，也不会有任何问题。以后可能会看到很多这样的代码。`Cocoa` 和 `NeXTSTEP` 中的操作方法需要接受 `sender` 参数，不管会不会用到。所以很多 `iOS` 代码（尤其是早期的代码）都是用这种方式编写的。

现在，你已经了解了操作方法和输出接口的基本概念，接下来就会学习如何在设计用户界面时使用它们。在开始之前，需要先进行一些清理工作，以保持代码整洁有序。

### 3.3.2 清理视图控制器

在项目导航面板中单击 `BIDViewController.m`，打开这个实现文件。可以看到，文件中有一些包含 `viewDidLoad` 和 `didReceiveMemoryWarning` 等方法的样板代码，这是由创建项目时选择的项目模板提供的。在 `UIViewController` 的子类中通常会用到这些方法，所以 `Xcode` 提供了这些基本的代码片段，如果需要的话，我们可以直接在这些方法中添加自己的代码。但是，这个项目并不需要这些代码片段，而它们既占用了空间又使代码不易阅读。为了简化以后的工作，应该删除那些不需要的代码。

在这个文件的顶部，可以看到一个空的类扩展。类扩展是一种特殊的 `Objective-C` 分类（`category`）声明，可以在其中声明方法和属性，但是这些方法和属性只能在当前文件的 `implementation` 块中使用。本书后面会用到类扩展，但是现在并不需要，所以应该把这个空的 `@interface...@end` 块删除。完成之后，实现文件看起来应该是这个样子：

```
#import "BIDViewController.h"

@implementation BIDViewController

@end
```

现在就简洁多了，是不是？不用担心刚才删除的那些方法，本书后面会介绍这些方法中的大部分内容。

### 3.3.3 设计用户界面

记得保存刚才所做的修改，然后单击 `Main.storyboard` 文件，就可以在 `Xcode` 的界面构建器中打开应用程序的视图（参见图 3-4）。你可能还记得上一章提到，编辑器的白色窗口中展示了应用程序的唯一一个视图。现在需要在这个视图中添加两个按钮和一个标签，从而实现图 3-1 所示的效果。

先来思考一下这个应用程序。需要在用户界面中添加两个按钮和一个标签，这个过程与上一章类似。不过，还需要通过输出接口和操作方法才能使应用程序与用户交互。

每个按钮都需要在控制器上触发一个操作方法。可以选择让每个按钮调用不同的操作方法，但由于它们本质上做的是同一件事（更改标签的文本），所以调用同一个操作方法。我们使用 `sender` 参数（之前的“操作方法”一节讨论过）来区分这两个按钮。除了操作方法，还需要一个与标签关联的输出接口，修改标签的显示文本。

首先添加按钮，然后再添加标签。设计用户界面时创建对应的操作方法和输出接口。也可以手动声明操作方法和输出接口，然后将用户界面元素与它们关联起来，不过，既然 `Xcode` 会自动

完成这些工作，为什么要手动去做呢？

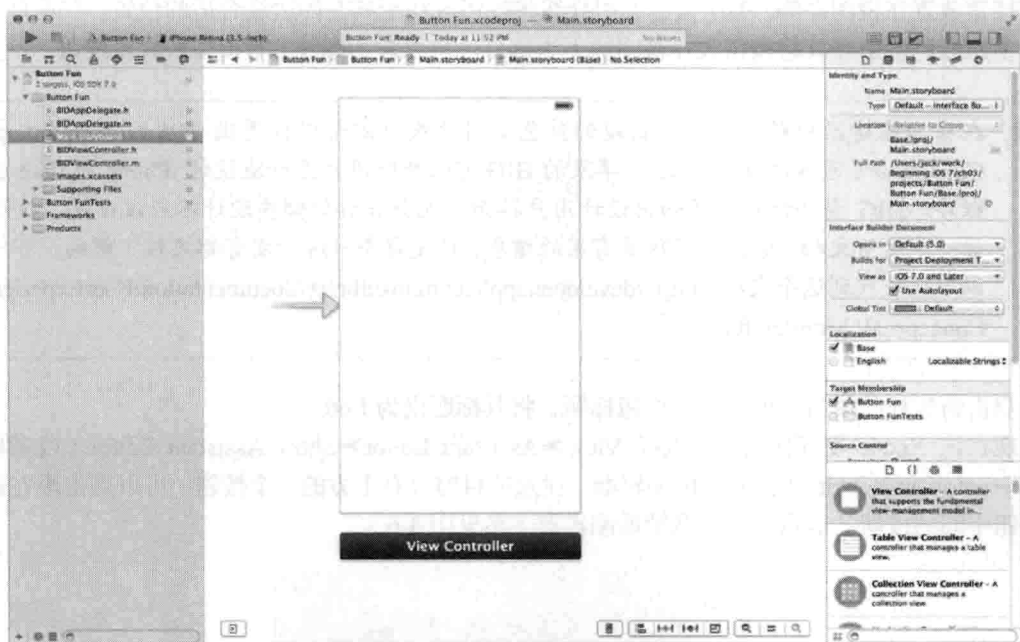


图 3-4 打开 Main.storyboard 文件就可以对 Xcode 的界面构建器进行编辑了

### 1. 添加按钮和操作方法

第一步要向用户界面添加两个按钮。随后让 Xcode 创建一个空的操作方法，我们可以把两个按钮都关联到这个操作方法。这样，用户点击按钮时就会调用这个操作方法，其中的代码就会执行。

选择 **View>Utilities>Show Object Library** (或按下 **control+option+command+3**) 打开对象库。在对象库的搜索框中输入 **UIButton** (实际上只需要输入开头的 4 个字母 **UIBu** 来筛选列表，就可以在对象列表的顶部看到 **UIButton** 了)，如图 3-5 所示。

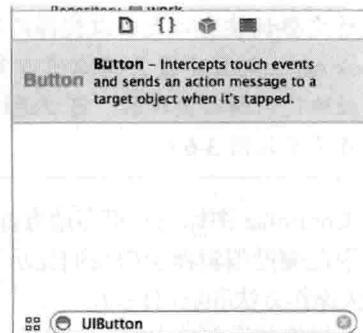


图 3-5 对象库中显示的按钮

把 Button 从库中拖到编辑区中白色的窗口中，这样就会在应用程序视图中添加一个按钮。将该按钮放置在视图左侧，参照蓝色的引导线将按钮放在距离左侧边缘合适的位置。在垂直方向上，参照蓝色的引导线将按钮放在视图的下半部分。可以参考图 3-1 的布局来放置按钮。

**注意** 在界面构建器中移动对象时出现的蓝色细引导线可以帮助你遵循 *iOS Human Interface Guidelines* (通常简称为 HIG)。苹果的 HIG 可以帮助用户更好地设计 iPhone 和 iPad 应用程序。HIG 会告诉你应该如何设计用户界面，也会告诉你哪些设计不应该出现。建议阅读一下这份文档，它包含了很多有用的信息，这是每个 iOS 开发者都应该了解的。可以在以下地址找到这个文档：<http://developer.apple.com/ios/library/documentation/UserExperience/Conceptual/MobileHIG/>。

双击新添加的按钮，可以编辑按钮标题，将其标题设为 Left。

现在该 Xcode 发挥作用了。选择 View>Assistant Editor>Show Assistant Editor (或者按下 option+command+return) 打开辅助编辑器。注意项目窗口右上方的 7 个按钮，可以点击最左边一组按钮中间的按钮，显示或者隐藏辅助编辑器 (参见图 3-6)。

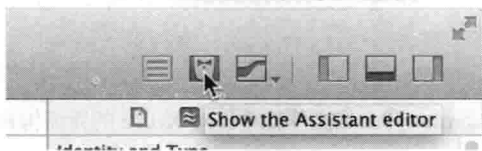


图 3-6 Show the Assistant editor 开关按钮

辅助编辑器会出现在编辑面板的右侧，除非你特别指定辅助编辑器的位置 (参见辅助编辑器菜单的选项)。辅助编辑器将出现在编辑面板的右侧，辅助编辑器左侧显示的内容仍然是界面构建器，而右侧显示的是 BIDViewController.h 或 BIDViewController.m (也就是当前查看的视图所属的视图控制器类的头文件以及实现文件)。

**提示** 打开辅助编辑器之后，或许需要调整窗口大小以获得足够的工作空间。如果你的显示器屏幕比较小 (比如 MacBook Air 上的显示器)，那么可以关闭 utility 视图或项目导航面板，从而获得足够的空间，有效地使用辅助编辑器。可以通过项目导航窗口右上角的 3 个视图按钮方便地完成这个操作 (参见图 3-6)。

还记得上一章提到过的 View Controller 图标吗？很多地方都与它有关。Xcode 知道我们的视图控制器类负责显示这个视图，因此辅助编辑器会向我们显示视图控制器类的头文件或实现文件，我们只可能在这里创建并连接操作方法和输出接口。

前面提到，BIDViewController 类中并没多少内容。它只是一个空的 UIViewController 子类，但它很快就不再是空的子类了！



现在要让 Xcode 自动创建一个新的操作方法，并把它与之前创建的按钮关联起来。这些定义代码要加在头文件中。如果辅助编辑器当前显示的不是头文件，请在上面的跳转栏中选择 `BIDViewController.h` 文件。

首先，点击新添加的按钮以选中它。然后，按住键盘上的 `control` 键不放开，接着用鼠标把按钮拖向辅助编辑器的源代码中。这个过程中会看到一条蓝色的直线，从按钮一直连到光标（参见图 3-7）。我们就是通过这条蓝色的直线把界面构建器中的对象连接到代码或其他对象上。

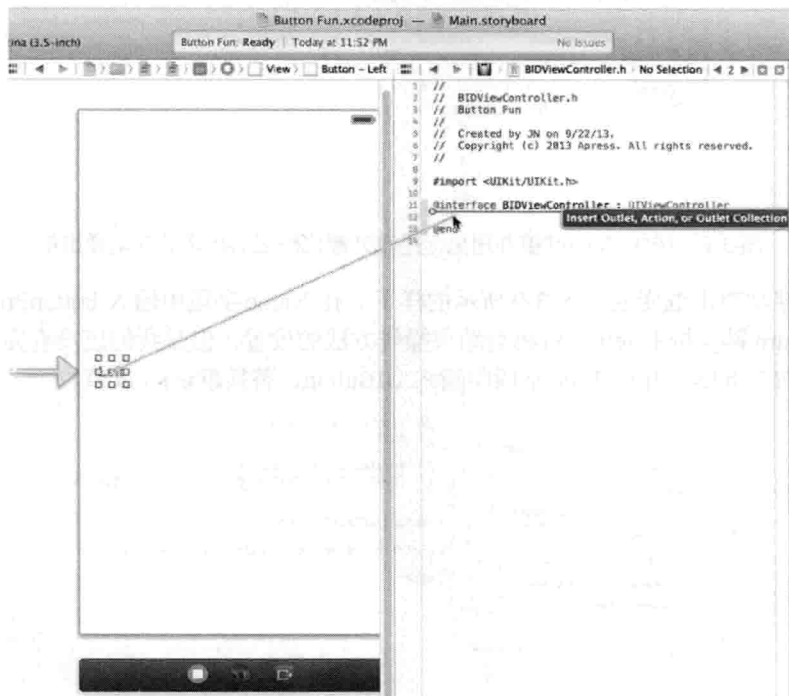


图 3-7 按住 `control` 键并用鼠标拖动到源代码上，可以看到创建输出接口、操作方法或者输出接口集合的选项

**提示** 可以将这条蓝色直线拖动到任何地方与按钮建立关联：可以拖动到辅助编辑器中的头文件、File's Owner 图标、编辑面板左侧的任何其他图标，甚至是编辑区域中的其他对象。

如果把指针移动到 `@interface` 和 `@end` 关键字之间（参见图 3-7），就会弹出一个灰色的提示框，告诉你如果在这里放开鼠标的话可以插入输出接口、操作方法，或输出接口集合。

**注意** 本书会使用操作方法和输出接口，但是不会用到输出接口集合。使用输出接口集合可以把多个同类型的对象关联到一个 `NSArray` 属性，而不必为每个对象单独创建一个属性。

请放开鼠标来完成关联，此时会出现一个浮动的弹出框，如图 3-8 所示。可以在这个窗口中定制新操作方法。在这个窗口中，点击名为 **Connection** 的弹出菜单，把当前的选择 **Outlet** 改为 **Action**。这就告诉 Xcode，我们要创建的是一个操作方法，而不是输出接口。

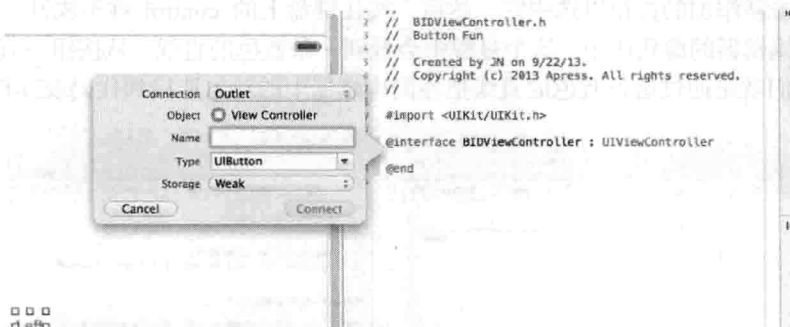


图 3-8 按住 Control 键并用鼠标拖动到源代码之后出现的浮动弹出框

现在这个浮动弹出框变成了图 3-9 所示的样子。在 **Name** 字段中输入 **buttonPressed**。输入完成后不要按 return 键。按下 return 键将会结束操作方法的设置，但是我们还没有完成这个操作方法的定制。请按 Tab 键，并在 **Type** 字段中输入 **UIButton**，替换默认的值。

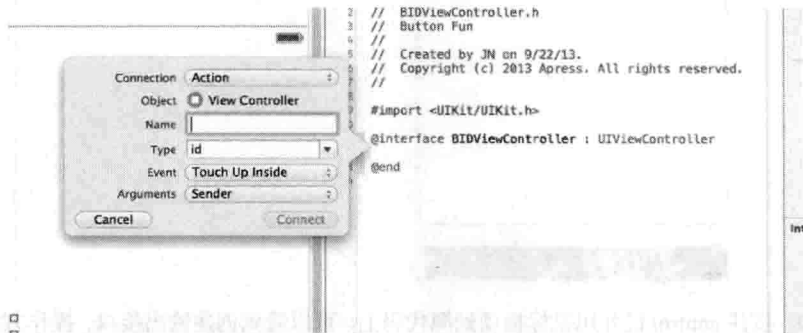


图 3-9 将关联类型改成 Action 后，弹出框的外观发生了改变

**注意** 也许你还记得，id 是一种泛型指针，它能指向任何 Objective-C 类。在这里保持 **Type** 字段中的值为 id 不变，这样也能正常工作，但是，如果把它更改为期望调用该方法的类，当试图通过一个错误类型的对象调用该方法时编译器就能发出警告。有时你希望利用这种灵活性，以便通过不同类型的控件调用同一个操作方法，这种情况下保留默认设置 id 即可。在本例中，我们只会通过按钮对象调用这个方法，所以应该将其改为 **UIButton**，让 Xcode 和 LLVM 知道我们的意图。现在，如果不小心把其他对象关联到这个方法，就会得到警告信息。

在 Type 下面还有两个字段，保留它们的默认值即可。在 Event 字段中指定什么时候调用这个方法。默认值 Touch Up Inside 仅会在用户的手指离开屏幕（且用户的手指在离开屏幕之前依然位于按钮内部）时触发相应的操作方法。这是按钮使用的标准事件。这就给了用户一个重新考虑的机会。如果用户的手指在离开屏幕之前从按钮上移到了别处，那么就不会触发这个方法。

在 Arguments 字段中选择操作方法要用到的方法签名（三选一）。本例中需要使用 sender 参数来区分触发方法的按钮。默认值就是 sender，所以不需要更改。

按下 return 键，或者点击弹出框中的 Connect 按钮，Xcode 就会自动插入操作方法。现在 BIDViewController.h 文件应如下所示：

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

@interface BIDViewController : UIViewController
- (IBAction)buttonPressed:(UIButton *)sender;

@end
```

Xcode 在类的头文件中添加了一个方法声明。使用辅助编辑器顶端的跳转栏切换到 BIDViewController.m 文件中，可以看到 Xcode 已经自动添加了一个方法存根。

```
- (IBAction)buttonPressed:(UIButton *)sender {
}
```

很快我们会再次讨论这个方法存根，编写一些代码用于响应按钮的点击事件。除了创建方法声明和方法实现，Xcode 还对按钮和这个操作方法进行了关联，并且把这些信息存储在分镜文件中。这意味着，不需要再做任何额外的工作，就可以保证按钮在应用程序运行时调用这个操作方法。

再来看 Main.storyboard 文件，再从对象库中拖出一个按钮，这次把新的按钮放在屏幕右边。然后双击按钮，把它的标题改为 Right。这时会出现蓝色的引导线，帮助你把按钮放到距离右侧边缘恰当的位置上，也可以帮你把新按钮与另一个按钮在垂直方向上对齐。

---

**提示** 除了从库中拖出一个新对象，还可以按住 option 键不放并且拖曳原始对象（本例中为 Left 按钮）。按住 option 键就是告诉界面构建器创建一个被拖曳对象的副本。

---

这次不必为新按钮创建新的操作方法。可以直接把这个新按钮关联到之前 Xcode 为我们创建的操作方法。应该怎么做呢？关联的过程与之前为第一个按钮创建操作方法的过程非常相似。

修改完按钮名称之后，按住键不放开，把按钮拖向辅助编辑器。无所谓是当前是.h 文件还是.m 文件，只需要把鼠标拖向 buttonPressed: 方法的位置，当指针接近 buttonPressed: 方法声明时，这个方法会高亮显示，同时会弹出一个显示 Connect Action 的灰色提示框（参见图 3-10）。看到这个弹出提示框之后，松开鼠标，Xcode 就会把按钮与这个已有的操作方法关联起来。现在，与另一个按钮一样，点击这个新添加的按钮就会触发 buttonPressed: 方法。

再说一遍，如果用这种方式把按钮拖曳到实现文件（.m 文件）中的操作方法，效果也是一样的。也就是说，按住 control 键拖曳按钮时，既可以拖向 BIDViewController.h 文件中的 buttonPressed

方法声明上,也可以拖向 BIDViewController.m 文件中的 `buttonPressed` 方法实现上。Xcode 真是太智能了!

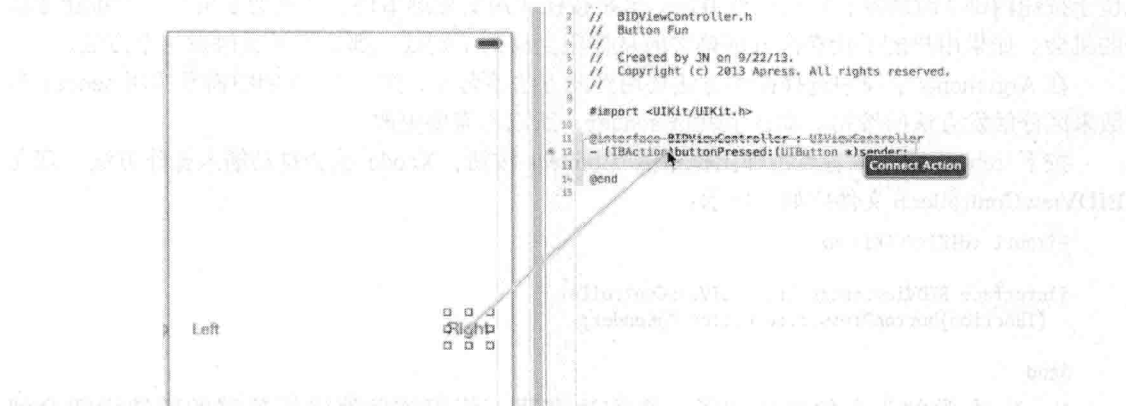


图 3-10 把按钮拖曳到一个已有的操作方法上,就可以将按钮与该操作方法关联起来

## 2. 添加标签和输出接口

在对象库的搜索框里输入 Label,可以找到用户界面元素 Label (参见图 3-11)。把 Label 拖曳到用户界面中,放置在两个按钮的上方。然后,调整标签的大小,从屏幕左侧边缘拉伸到右侧边缘。这样标签就有足够的空间来容纳需要显示给用户的文本。

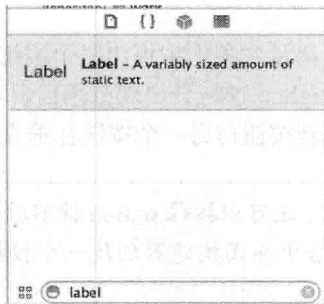


图 3-11 对象库中的标签

标签中的文本默认是左对齐的,但是我们希望它居中对齐。选择 `View > Utilities > Show Attributes Inspector` (或者按下 `option+command+4`) 打开属性检查器 (参见图 3-12)。确保选中了这个标签,然后在属性检查器中找到 `Alignment` 按钮集合。选择中间的按钮就可以使标签中的文本居中显示。

用户点击按钮之前,我们不希望标签显示任何文本,所以双击标签 (这样就选中了标签的文本),按下键盘上的 `delete` 键。这会删除标签当前显示的文本。按下 `return` 键提交更改。这样一来,标签在未选中时就看不到了,不过不用担心,它仍然在那里。

**提示** 如果有不可见的用户界面元素，比如空白标签，而你又希望能看到它们所在的位置，可以从 Editor 菜单中选择 Canvas，然后从弹出的子菜单中勾选 Show Bounds Rectangles。

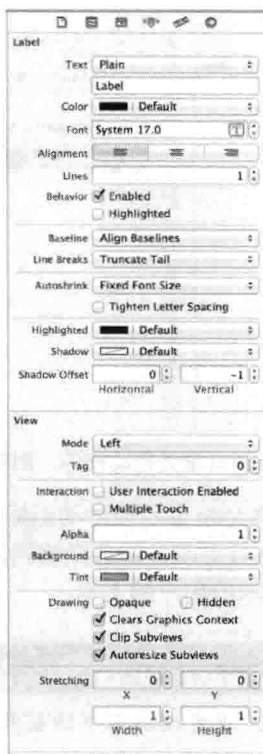


图 3-12 标签的属性检查器

最后剩下的工作就是为标签创建输出接口。这与之前创建并且关联操作方法的方式完全相同。确保打开了辅助编辑器，并且显示着 BIDViewController.h 文件。如果需要切换当前显示的文件，可以使用辅助编辑器上方跳转栏中的弹出菜单。

接着，选中界面构建器中的标签控件，按住 control 键把它从标签拖向头文件。当光标位于已有的操作方法的上方，看到如图 3-13 所示的画面时，松开鼠标和键盘，你将再次看到图 3-8 所示的弹出窗口。

我们要创建一个输出接口，所以保留 Connection 中的默认值 Outlet。并且要为这个输出接口指定一个描述性的名称，以便在编写代码时能够知道这个输出接口的作用。在 Name 字段中输入 statusLabel。Type 字段使用默认值 UILabel。对于最后一个名为 Storage 的字段，保留默认值即可。

按下 return 键提交更改，Xcode 将会在代码中插入一个输出接口属性。现在控制器类的头文件应该是下面这个样子：

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

```
@interface BIDViewController : UIViewController
@property (weak, nonatomic) IBOutlet UILabel *statusLabel;
- (IBAction)buttonPressed:(UIButton *)sender;
@end
```

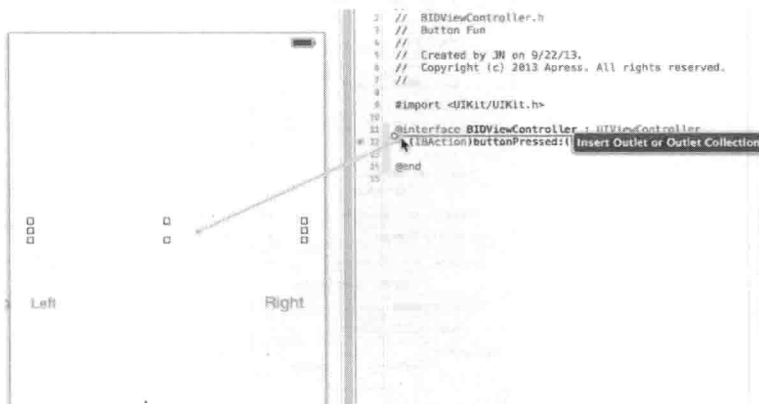


图 3-13 按住 control 键并拖动控件，创建一个输出接口

现在，有了一个输出接口，而且 Xcode 也已经自动把标签与这个输出接口关联了起来。这意味着，如果在代码中对 statusLabel 进行更改，就会影响到用户界面中的标签。例如，如果设置 statusLabel 的 text 属性，就会改变显示给用户的文本。

### 自动引用计数

如果你已经很熟悉 Objective-C，或者读过本书的前几版，就会发现，这里没有 dealloc 方法。我们没有释放实例变量！

警报！警报！这很危险！

事实上，你大可以放松。这里没有问题。一点儿也不危险，真的。

已经不必再释放对象了。其实这么说并不完全正确。需要释放对象，但是苹果公司在 Xcode 中使用的 LLVM 编译器相当智能，可以自动释放对象。它通过 ARC 功能来完成这个繁重工作。这就意味着不再频繁地调用 dealloc 方法了，也不需要再关心 release 或者 autorelease 的调用了。ARC 的出现是一个巨大的进步，本书所有的例子都使用 ARC。过去两年内 ARC 在 Xcode 中是可选的内容，不过到了现在你创建的每个项目都会默认启用它。

ARC 只适用于 Objective-C 对象，并不能用于 Core Foundation 对象或是使用 malloc() 分配的内存以及类似的东西，也会有一些陷阱和注意事项，但是大多数情况下，已经没有必要担心内存管理了。

要了解更多关于 ARC 的信息，可以在以下网址查看 ARC 的发布说明：

<http://developer.apple.com/library/ios/#releasenotes/ObjectiveC/RN-TransitioningToARC/>

ARC 确实很酷,但它并不是万能的。仍然需要理解 Objective-C 内存管理的基本规则,才能避免使用 ARC 时遇到麻烦。要想重温 Objective-C 内存管理机制,可以阅读苹果公司的 *Memory Management Programming Guide*,地址是: <http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/MemoryMgmt/>。

### 3. 编写操作方法

至此,已经完成了用户界面的设计,并且将用户界面与输出接口和操作方法都进行了关联。最后要做的就是,点击按钮时,使用这些操作方法和输出接口来设置标签的显示文本。在项目导航面板中单击 `BIDViewController.m` 文件,并在编辑器中打开它。找到之前 Xcode 为我们创建的空的 `buttonPressed:` 方法。

为了区分这两个按钮,需要用到 `sender` 参数。使用 `sender` 参数可以获取到被点击的按钮标题,根据这个标题创建一个新的字符串,然后把这个字符串作为标签的文本。在这个空的方法中添加以下加粗代码:

```
- (IBAction)buttonPressed:(UIButton *)sender {
    NSString *title = [sender titleForState:UIControlStateNormal];
    NSString *plainText = [NSString stringWithFormat:@"%@@ button pressed.", title];
    _statusLabel.text = plainText;
}
```

这段代码非常直观。第一行使用 `sender` 参数获取被点击的按钮的标题。由于按钮在不同状态下可以有不同的标题,因此使用 `UIControlStateNormal` 参数表明我们需要获取的是按钮在正常状态时(未被按下时)的标题。获取控件(按钮是控件的一种)标题时,通常都是使用这种状态。第 4 章会详细介绍控件状态。

接下去的一行代码创建了一个新的字符串,新字符串的内容是在按钮标题(上一行代码获取到的标题)末尾添加“button pressed.”文本拼接而成的。因此,如果点击标题为 `Left` 的按钮,就会创建一个内容为“Left button pressed.”的字符串。最后一行代码就是把这个新的字符串赋给标签的 `text` 属性,这样就可以改变标签的显示文本。

### 消息嵌套

一些开发者经常会嵌套使用 Objective-C 消息。在学习过程中你可能会看到类似这样的代码:

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

这一行代码的功能与之前 `buttonPressed:` 方法中的前两行代码完全等价。这是因为 Objective-C 方法可以嵌套,这样就不必为每一个嵌套的方法调用都创建一个对象来保存返回值。

为使代码清晰,本书的示例代码中通常不会嵌套 Objective-C 消息,除非是对 `alloc` 和 `init` 的调用,因为它们的嵌套调用已经是长期以来的一种标准约定。



### 3.3.4 运行应用

到现在为止，这个项目基本上就完成了。准备好运行这个应用了吗？我们来试试！

选择 **Product** ➤ **Run**，如果碰到了编译或链接错误，请参照本章前面部分对比一下你的代码。代码正确构建之后，Xcode 将会启动 iOS 模拟器运行这个应用程序。点击右边的按钮，标签应该会显示 “Right button pressed.”（参见图 3-1）。如果再点击左边的按钮，标签文本将会变为 “Left button pressed.”。

目前为止一切顺利，但是，如果回头看一下图 3-1，会发现有一件事情忘记做了。最终结果屏幕截图中的按钮名称是用粗体显示的，但是目前只显示了纯文本。可以使用 `NSAttributedString` 加粗文本。

### 3.3.5 样式文本

`NSAttributedString` 可以对字符串附加格式信息，比如字体和段落对齐。可以在整个字符串上使用这些元数据，也可以对字符串的不同部分使用不同的属性。如果你想在文字处理程序中对不同文本片段使用不同的格式，那么 `NSAttributedString` 正是你想要的！

然而，苹果的 `UIKit` 中一直都没有一个类能够处理属性字符串（attributed string）。如果希望展现一个同时包含加粗文本和正常文本的标签，要么使用两个标签，要么直接在自己的视图上绘制文本。这些方法并不是无法实现，但是使用起来非常复杂，让大多数开发者都望而却步。在 iOS 6 中对于样式文本有了很大的改善，大部分主要的 `UIKit` 控件都允许使用属性字符串了。对这个例子里的 `UILabel` 来说，只需要创建一个属性字符串，然后把它赋给标签的 `attributedText` 属性就可以了。

那么，修改 `buttonPressed:` 方法，删除划掉的代码行，添加加粗显示的代码行，如下所示：

```
- (IBAction)buttonPressed:(UIButton *)sender {
    NSString *title = [sender titleForState:UIControlStateNormal];
    NSString *plainText = [NSString stringWithFormat:@"%@" button pressed.", title];
    _statusLabel.text = plainText;

    NSMutableAttributedString *styledText = [[NSMutableAttributedString alloc]
                                              initWithString:plainText];

    NSDictionary *attributes =
    @{
        NSFontAttributeName : [UIFont boldSystemFontOfSize:_statusLabel.font.pointSize]
    };

    NSRange nameRange = [plainText rangeOfString:title];

    [styledText setAttributes:attributes range:nameRange];
    _statusLabel.attributedText = styledText;
}
```

新代码做的第一件事情就是基于我们要显示的文本创建一个属性字符串（`NSMutableAttributedString` 的一个实例）。这里需要使用可变的属性字符串，因为需要改变它的属性。

接下来，创建一个用于保存字符串属性的字典。现在只需要一种属性，所以这个字典只包含一个键值对。通过名为 `NSFontAttributeName` 的键可以为属性字符串的一部分指定字体。这里传入的值是一种名为“bold system font”的字体，字体大小与标签当前使用的字体大小一致。相对于使用硬编码的字体名称，通过这种方式指定字体更加灵活，因为系统总是能够恰当地展现加粗字体。

**提示** 如果刚接触 Objective-C 不久，你可能还不熟悉新的字典创建语法，但是这种语法非常简单。Xcode 内置的 LLVM 提供了非常简洁的字典创建语法，与显式地在 `NSDictionary` 上调用方法相比，新语法更简单易用。它的基本形式如下所示：

```
@{
    key1 : value1,
    key2 : value2
}
```

使用这种方式创建字典时，除了不再需要输入冗长的类名和方法名之外，它还能自动以“正确的顺序”来排列这些键值对，与很多脚本语言类似，比如 Ruby、Python、Perl、JavaScript。

除了新的字典语法，2012 年的时候也添加了新的数组语法和数值语法。本书的代码示例使用这些新的语法片段。

然后，取得 `plainText` 字符串中待改变的子字符串（也就是这里的 `title`）所属的范围（由一个起始索引和一个长度组成）。把属性应用到属性字符串，然后把属性字符串赋给标签。

现在，点击 Run 按钮运行应用，可以看到标签中的按钮名称是以粗体显示的。

## 3.4 应用程序委托

很好，应用程序运行起来了！在进入下一个主题之前，我们花点时间来看看两个还没有介绍过的源代码文件：`BIDAppDelegate.h` 和 `BIDAppDelegate.m`。这两个文件实现了应用程序委托（application delegate）。

Cocoa Touch 广泛地使用了委托（delegate），委托是负责为其他对象处理特定任务的对象。通过应用程序委托，能够在某些预定义时间点为 `UIApplication` 类做一些工作。每个 iOS 应用程序都有且仅有一个 `UIApplication` 实例，它负责应用程序的运行循环，以及处理应用程序级的功能（比如把输入信息分发给恰当的控制类）。`UIApplication` 是 `UIKit` 的标准组成部分，它主要在后台处理任务，所以一般来说不用管它。

在应用程序执行过程中的某些特定时间点，`UIApplication` 将会调用特定的委托方法（如果委托对象存在，并且实现了相应的委托方法）。例如，如果需要在程序退出时触发一段代码，可以在应用程序委托中实现 `applicationWillTerminate:` 方法，在这个方法内编写想要的代码即可。

通过这种委托, 开发者可以实现通用的应用程序级行为, 而不需要继承 `UIApplication` 类, 也不需要了解它的任何内部机制。

在项目导航面板中单击 `BIDAppDelegate.h`, 查看应用程序委托的头文件, 它应该如下所示:

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

@interface BIDAppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;

@end
```

其中有一行代码需要特别注意:

```
@interface BIDAppDelegate : UIResponder <UIApplicationDelegate>
```

注意到尖括号里面的值了吗? 它表示这个类遵循 `UIApplicationDelegate` 协议。按住 `option` 键, 光标应该变成了十字形。把光标移到 `UIApplicationDelegate` 这个单词上, 现在光标应该变成了问号, 并且 `UIApplicationDelegate` 被高亮显示了, 类似于浏览器中的链接 (参见图 3-14)。

```
8
9  #import <UIKit/UIKit.h>
10
11  @interface BIDAppDelegate : UIResponder <UIApplicationDelegate>
12
13  @property (strong, nonatomic) UIWindow *window;
14
15  @end
16
```

图 3-14 在 Xcode 中按住 `Option` 键, 把光标指向代码中的某个符号, 该符号就会高亮显示, 同时光标会变成问号

仍然按住 `option` 键不放开, 单击这个链接。这时候会弹出一个窗口, 其中显示了 `UIApplicationDelegate` 协议的简单概述, 如图 3-15 所示。

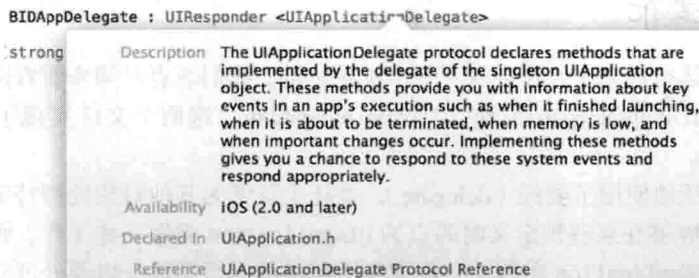


图 3-15 按住 `option` 并单击源代码中的 `<UIApplicationDelegate>`, Xcode 就会弹出这个被称为 Quick Help 的面板, 其中简单描述了协议的相关内容

注意这个新弹出的文档窗口 (参见图 3-15) 底部的两个链接。单击 `Reference` 链接可以查看

这个符号的完整文档，单击 [Declared In](#) 链接可以查看这个符号在头文件中的定义。这个技巧也适用于类、协议、分类名称，以及编辑器面板中显示的方法名称。只要按住 `option` 键并单击某个单词，Xcode 就会在文档浏览器中搜索这个词。

了解如何快速查找文档中的内容绝对大有裨益，但是查看这个协议的定义可能更为重要。开发者可以通过这个协议的定义来了解这个应用程序委托能够实现哪些方法，以及这些方法分别会在什么时候调用。有必要花些时间来阅读这些方法的说明。

**注意** 如果你之前使用过 Objective-C，但是没有用过 Objective-C 2.0，那么你需要知道的是，现在可以为协议指定可选方法了。UIApplicationDelegate 包含很多可选方法。不过，不需要在应用程序委托中实现任何可选方法，除非真的有必要。

回到项目导航面板，单击 `BIDAppDelegate.m` 来查看应用程序委托的实现。它应该如下所示：

```
#import "BIDAppDelegate.h"

@implementation BIDAppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
(NSDictionary *)launchOptions
{
    // 在程序启动之后，重写自定义设置的位置
    return YES;
}

- (void)applicationWillResignActive:(UIApplication *)application
{
    // 应用即将从活动状态切换到不活动状态时会触发这个方法。在出现某种临时中断（比如有
    // 来电或者短信）或用户退出应用程序时都会触发这个方法，然后应用就会转换为后台运行
    // 可以在这个方法中暂停正在进行的任务，禁用定时器，降低 OpenGL ES 帧率。如果是游戏应
    // 用，应该在这个方法中暂停游戏
}

- (void)applicationDidEnterBackground:(UIApplication *)application
{
    // 在该方法中释放共享资源，保存用户数据，清除定时器，并存储足够的应用状态信息，目
    // 的是当应用终止时，将它恢复到当前状态
    // 如果你的应用支持在后台执行，那么当用户退出时调用这个方法而不是 applicationWill
    // Terminate：方法
}

- (void)applicationWillEnterForeground:(UIApplication *)application
{
    // 这个方法会在应用程序从后台运行状态转换到活动状态的过程中被调用，可以在这里恢复
    // 应用正常运行所需的信息
}
```

```
}  
  
- (void)applicationDidBecomeActive:(UIApplication *)application  
{  
    // 当应用程序处于非活动状态时重新启动暂停（或尚未启动）的任务。如果程序之前在后台  
    // 运行，那么可以选择刷新用户界面  
  
}  
  
- (void)applicationWillTerminate:(UIApplication *)application  
{  
    // 程序即将终止时调用该方法。如果有必要，那么保存数据，参见 applicationDid  
    // EnterBackground:  
  
}  
  
@end
```

在这个文件的顶部，可以看到应用程序委托实现了文档中列出的一个协议方法，即 `application:didFinishLaunchingWithOptions:`。也许你已经猜到了，当应用程序完成所有的初始化工作，并且准备好与用户进行交互时，就会调用这个方法。这个方法经常用来创建要在应用程序运行的整个生命周期内活动的对象。

本书后面你将看到更多相关的内容。我们只是希望在本章结束之前你能够了解一些应用程序委托的背景知识，以及所有这些内容之间的联系。

### 3.5 小结

本章通过一个简单的应用程序介绍了 MVC，并学习了如何创建并连接输出接口和操作方法，实现视图控制器，以及使用应用程序委托。还学习了如何在点击按钮时触发操作方法，以及如何在运行时更改标签的文本。虽然这只是一个简单的应用程序，但是其中用到的基本概念与 iOS 中的所有控件都是一致的，不仅仅是按钮控件。事实上，本章对按钮和标签的使用方式适用于 iOS 中的大部分标准控件。

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！如果尚未完全理解这些内容，本书后面创建比较复杂的界面时，你就会感到更加困惑。

下一章将介绍其他一些标准 iOS 控件。你还可以学到如何使用警告来把重要的事项通知给用户，以及如何通过操作表单指示用户需要在程序继续运行之前作出一个选择。做好准备之后，深吸一口气，给自己一点鼓励，开始学习下一章的内容吧！

第3章介绍了MVC，并且构建了一个简单的MVC应用程序。我们学习了输出接口和操作方法，并使用它们将按钮控件与文本标签绑定在一起。本章要构建一个复杂一些的应用程序，这会加深你对控件的理解。

本章将实现一个图像视图、一个滑动条、两个不同的文本框、一个分段控件、两个开关控件和一个更符合iOS风格的按钮。你将了解如何设置和获取各种控件的值。本章还会介绍如何使用操作表单强制用户作出选择，以及使用警告视图向用户显示重要的反馈信息。并将讨论控件状态，以及如何使用可拉伸图像让按钮更加美观。

本章的应用程序将使用大量的用户界面元素，因此讲解方式会与前两章有所不同。本章会将应用程序分成若干小块，每次实现其中的一块，需要不断地在Xcode和iOS模拟器之间进行切换，我们对每一块都要进行测试，然后再继续实现下一块。把构建复杂界面的过程划分为小块，这样可以简化它，也更加接近于实际的应用程序构建流程。“编码—编译—调试”是软件开发人员日常工作的主要内容。

## 4.1 满是控件的屏幕

前面已经提到，本章将要构建的应用程序比第3章稍微复杂一些。我们仍然只使用一个视图和控制器，但这个视图上有很多控件，如图4-1所示。

iPhone屏幕顶部的Apress标志是一个图像视图。在这个应用程序中，它的作用仅仅是显示一个静态图像。标志下方是两个文本框，一个允许输入字母和数字，另一个只允许输入数字。文本框下方有一个滑动条。当用户移动滑动条时，其左侧标签的值将会随之改变，它反映了滑动条的当前值。

滑动条下方是一个分段控件和两个开关控件。分段控件在这里用于控制其下方空间显示哪种类型的控件。当应用程序首次启动时，分段控件下方会有两个开关控件。更改任一开关的值都会导致另一个开关随之变动，两个开关的值始终保持一致。当然，在真实应用程序中不太可能这么做，但它的确说明了如何通过代码更改控件的值，以及Cocoa Touch如何在不需要任何人为控制的情况下，实现特定操作的动画效果。

图4-2显示了当用户单击分段控件右侧按钮时发生的情况。分段控件下方的两个开关控件消

失不见了，由一个按钮取而代之。点击这个 Do Something 按钮时，将弹出一个操作表单，询问用户是否确定要点击这个按钮(参见图 4-3)。对于具有潜在危险或可能导致严重后果的用户输入，这是标准的响应方式，可以多给用户一次选择的机会，避免潜在危险的发生。如果选择 Yes, I'm Sure!，应用程序会弹出一个警告视图通知用户一切正常(参见图 4-4)。

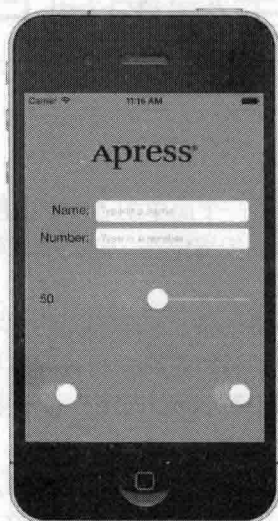


图 4-1 Control Fun 应用程序，包含文本框、标签、滑动条和其他几个 iOS 控件

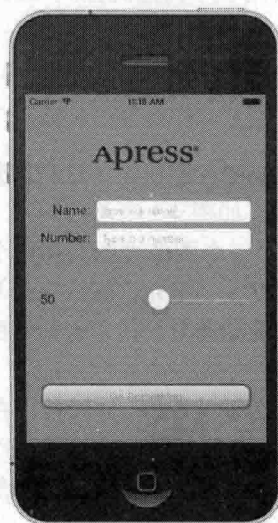


图 4-2 点击分段控件左侧按钮，在分段控件下方显示一组开关控件；点击分段控件右侧按钮，则在分段控件下方只显示一个按钮



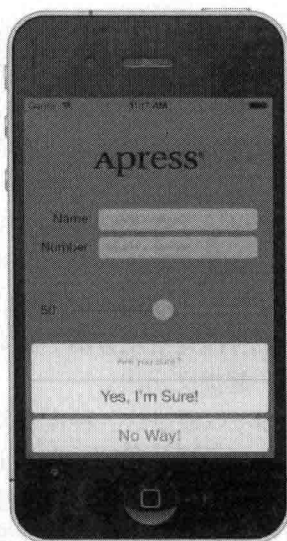


图 4-3 应用通过操作表单来请求用户的输入响应

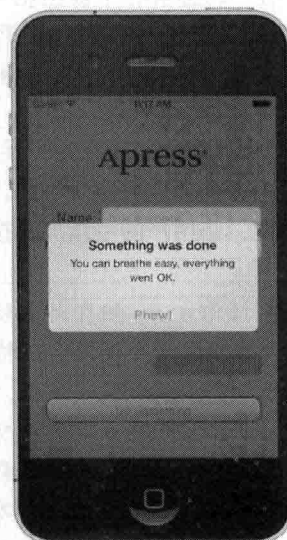


图 4-4 在可能导致严重后果时，利用警告视图来通知用户。

不过，这里用它是为了通知用户一切正常

## 4.2 活跃控件、静态控件和被动控件

用户界面控件共有三种基本模式：活跃、静态（又叫非活跃）和被动。上一章使用的按钮是

典型的活跃控件，点击它们时会发生一些事情——通常是触发一段自己编写的事件代码。

虽然大多数控件都能直接触发操作方法，但并非所有控件都是如此。本章将要实现的图像视图就是静态控件的一个典型例子。尽管可以对 `UIImageView` 控件进行一些配置使其能够触发操作方法，但在本章的应用程序中，图像视图是被动的，用户不能对其执行任何操作。标签和图像控件通常都采用这种方式。

一些控件可以在被动状态下工作，仅用于存储用户输入的值，以备后续使用。这些控件不会触发任何操作方法，但是用户可以与之交互，并修改它们的值。被动控件的一个典型例子是网页上的文本框。虽然可以在离开文本框时触发验证代码，但网页上的大多数文本框都只是保存数据的容器，这些数据在用户点击提交按钮时提交给服务器。文本框本身不会触发任何代码，但是在点击提交按钮时可以把文本框的数据一起提交出去。

在 iOS 设备上，大多数可用控件都可以通过这三种模式使用，并且几乎所有的控件都支持一种以上的模式，可以根据自己的需求选择合适的模式。所有 iOS 控件都是 `UIControl` 的子类，因此它们能够触发操作方法。大多数控件都支持被动模式，并且所有控件都支持静态或者不可见模式。例如，使用某个控件时可能会触发另一个静态控件成为活跃控件。但是，包括按钮在内的一些控件，除了在活跃模式下用来触发代码以外，实际上并没有其他用途。

iOS 和 Mac 上的控件在行为上存在一些差异，下面给出一些例子。

- ❑ 由于多点触控界面的引入，所有 iOS 控件都可以根据被触控的方式触发多个不同的操作方法。可以在用户点击按钮时触发一个操作方法，当用户手指在按钮上滑动时，再触发另一个操作方法。
- ❑ 可以在用户按下按钮时触发一个操作方法，而在用户手指离开按钮时触发另一个操作方法。
- ❑ 可以让单个控件对单一事件调用多个操作方法。例如，你可以让 `Touch Up Inside` 事件触发两个不同的操作方法，也就是说，当用户的手指离开按钮时这两个操作方法都会被调用。

---

**注意** 虽然在 iOS 中控件可以触发多个操作方法，但在大多数情况下，最好是对控件的每一个特殊用途实现唯一的操作方法。虽然通常不会使用这项特性，但是在使用界面构建器时最好牢记这一点。如果一个控件已经关联了某个操作方法，而后又在界面构建器中把同一控件的同一事件关联到其他操作方法，此时并不会断开该控件与前一个操作方法的关联。如果一个控件可以触发多个操作方法，可能会导致应用出现意料之外的行为。在界面构建器中重新关联事件时务必要留心，确保在关联到新的操作方法之前断开之前的关联。

---

iOS 与 Mac 之间的另一个主要区别是，iOS 设备没有物理键盘。iOS 的标准键盘实际上是由系统提供的一个满是按钮控件的视图。你的代码可能永远都不会直接与 iOS 的键盘交互。

## 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 Single View Application 模板，仍然按照前两章的方法创建项目。

项目创建完成之后，要先找到将在图像视图中使用的图片。需要先将图片导入到 Xcode 中，然后才能在界面构建器中使用它，所以现在先导入图片。可以在 04 - Control Fun 文件夹的项目归档中找到两张图片，名为分别为 `apress_logo_344.png` 和 `apressL_logo_172.png`，它们是同一张图片的 Retina（视网膜）版和普通版。我们要将这些图片添加到新项目的图像资源目录中，并确定程序运行时所使用的图片。如果使用自己的两张图片，要确保它的格式是 .png，并且图片尺寸要与屏幕空间相适应。大的那一张图片的高度应小于 200 像素，宽度应小于 600 像素，这样不需要调整图像大小，视图的顶部也能够放得下。小的那张宽高则各为一半。

在 Xcode 中选中 Images.xcassets 文件并点击编辑区左下角的加号按钮。将弹出一个选项菜单，你应该选择 New Image Set 那项。这将出现一个新的位置来添加你的实际图片文件。目前它的名字是 Image，不过我们想给它取个唯一的名字，这样我们就可以在项目的任意位置引用到它。选择 Image 图标，调出属性检查器（option+command+3），并在里面将图像的名字更改为 `apress_logo`。

现在可以把图片从 Finder 拖动到图像内容方框中，这样就把图片添加到 `apress_logo` 图像中了。小的图片拖到标题是 1x 的位置，大的图片拖到标题是 2x 的位置。

## 4.4 实现图像视图和文本框

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

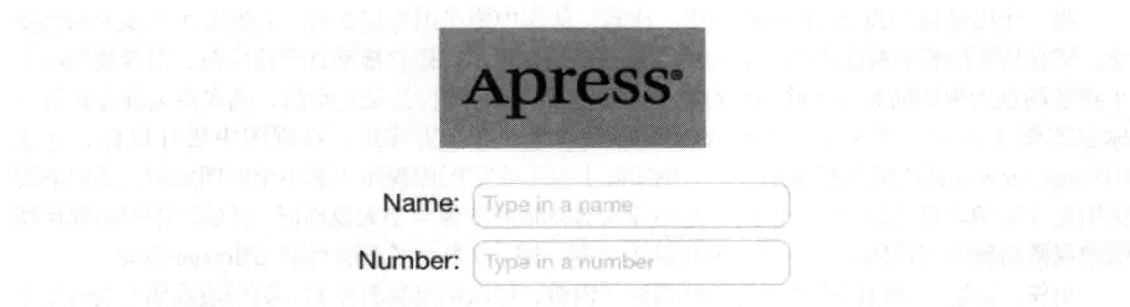


图 4-5 首先实现图像视图、标签和文本框

### 4.4.1 添加图像视图

在项目导航面板中单击 Main.storyboard，在界面构建器中打开它。可以看到熟悉的白色背景和一个 iPhone 尺寸大小的视图，可以在这个视图上放置应用程序的界面元素。

如果对象库还没有打开，那么选择 View > Utilities > Show Object Library 打开它。把滚动条向下拖动大约 1/4 的距离应该就能找到 Image View 了（参见图 4-6），也可以直接在搜索框中输入 image view。请记住，库面板顶部的第三个图标代表对象库。在其他面板中无法找到 Image View。

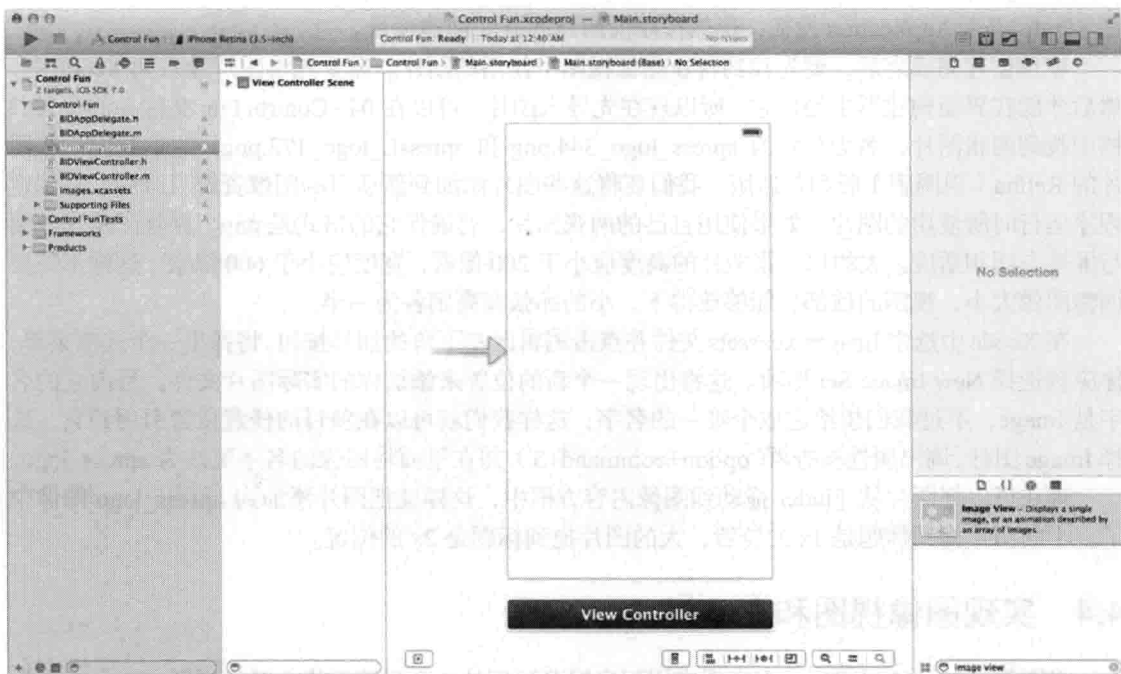


图 4-6 界面构建器库面板中的 Image View

将一个图像视图拖到 nib 编辑器中。注意，从库中拖出图像视图时，它的大小会发生两次变化。把它从库面板中拖出来时，它的形状是一个横向矩形。把它拖动到视图中时，图像视图的大小将被调整为视图的大小（除去顶部状态栏的大小）。这种行为是正常的，确实在大部分情况下你也需要这么做，因为第一个放入视图的图像通常都作为背景。在视图中松开鼠标，注意 UIImageView 与视图的边缘要对齐。本例实际上并不希望图像视图占满整个视图空间，所以需要使使用拖动手柄调整它的大小，使其与刚刚导入 Xcode 的图像大小大致相同。目前不用担心如何使图像视图精确匹配原图，下一节会介绍这个主题。图 4-7 显示了调整后的 UIImageView。

记住，如果在编辑区中选择控件时遇到了困难，可以调出编辑器的层级列表视图（点击左下方的小三角形图标）。现在就可以在列表中点击想要选择的界面元素，这个元素就会在编辑器中被选中。

如果要选取一个嵌套在另一个对象中的对象，可以点击包含该嵌套对象的对象左侧的三角形图标，从而把其中包含的对象都显示出来。本例要选取图像视图，首先点击 view 对象左边的三角形图标，然后点击出现在 dock 面板中的 image view，这样，编辑器中相应的图像视图就被选中了。

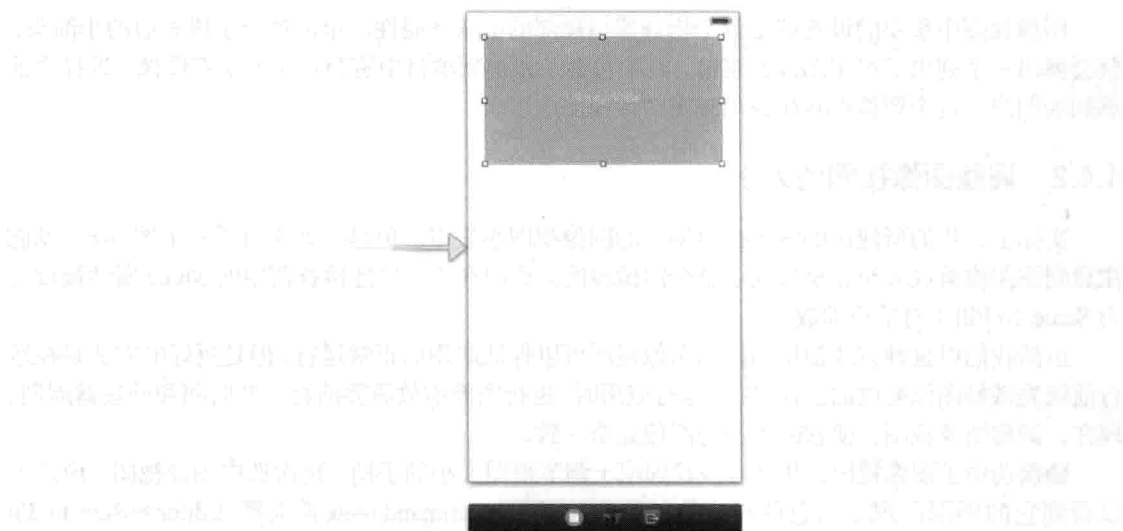


图 4-7 调整后的 UIImageView，大小与将要放入的图片相匹配

选中图像视图之后，按 `option+command+4` 调出属性检查器，这样可以看到 UIImageView 类的可编辑选项，参见图 4-8。

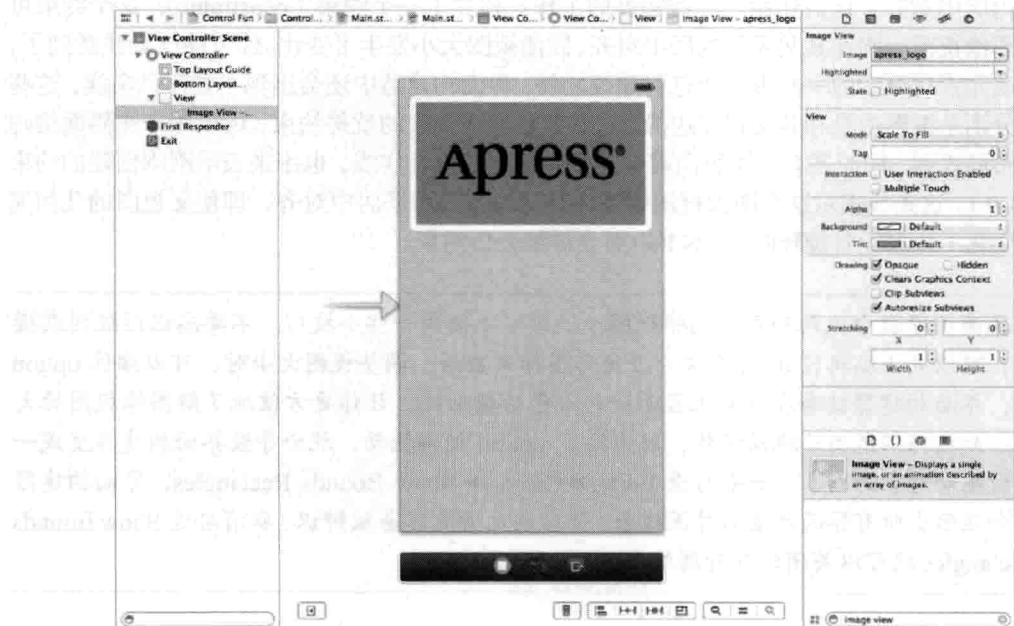


图 4-8 图像视图的属性检查器。在检查器顶部的 Image 弹出菜单中选中想要使用的图像，这个图像就会出现在图像视图中

图像视图中重要的设置就是位于检查器最顶部的 Image 属性。单击这个字段右边的小箭头,就会弹出一个列出了可用图像的菜单,其中包含了添加到项目中资源目录的所有图像。选择之前添加的图像。这个图像现在应该出现在图像视图中了。

#### 4.4.2 调整图像视图的大小

实际上,我们所使用的图像比容纳它的图像视图小很多。但是,如果再看一下图 4-8,就能注意到该图像被放大至完全填满了整个图像视图。原因在于,属性检查器中的 Mode 属性被设置为 Scale To Fill (自适应缩放)。

虽然我们以这种方式使用图像和图像视图可以保证应用的正常运行,但是更好的方法是在运行前就先做好图像缩放的工作,因为运行应用时,进行图像缩放需要消耗一些时间和处理器周期。现在,调整图像视图,使它的大小与图像完全一致。

确保选中了图像视图,并且可以看到用于调节视图大小的手柄。再次选中图像视图,应该可以看到它的周围出现了灰色的粗边框。最后,按下 `command+=` 或者选择 `Editor>Size to Fit Content`。这样就可以将图像视图的大小自动调整到与其中的图像完全一致。

图像视图已经调整好了,现在将它移动到目标位置。需要首先取消选择,然后点击它以再次选中。现在拖动视图图像,使其顶部与视图顶部的蓝色引导线对齐,并且使用蓝色引导线使其居中对齐(参见图 4-9)。注意,还可以通过 `Editor>Align>Horizontal Center in Container` 使视图在其父视图内居中对齐,同时还做了一些额外的工作:建立了一个约束(constraint),这个约束可以保证该图像视图始终在其父视图内居中对齐,即便视图大小发生了变化。你可能已经注意到了,除了在拖动元素时显示的一些虚的蓝色引导线之外,界面构建器中还会出现一些蓝色实线,这些线的两端分别是视图边缘和其父视图边缘。这些蓝色实线代表的就是约束,可以用来在界面构建器中表示布局规则。同时还有一条纵向贯穿整个主视图的橙色实线,也用来表示刚刚创建的约束(参见图 4-9)。这条线表示这个图像视图始终在其父视图内水平居中对齐,即使父视图的几何属性发生了变化(比如设备旋转时)。本书后面会详细介绍约束。

---

**提示** 在界面构建器中拖动和重新调整视图大小时可以使用一些小技巧。不要忘记层级列表模式,可以点击编辑器 dock 底部的三角形图标来激活。调整视图大小时,可以按住 `option` 键,界面构建器就会在屏幕上显示一些红色的辅助线,让你更方便地了解图像视图的大小。此技巧不适用于拖动操作,因为按下 `option` 键再拖动,就会导致界面构建器生成一份被拖动对象的副本。如果勾选 `Editor>Canvas>Show Bounds Rectangles`,界面构建器还会显示出所有界面元素的外围线条,使这些元素更容易被辨识。取消勾选 `Show Bounds Rectangles` 就可以关闭这些外围线条。

---

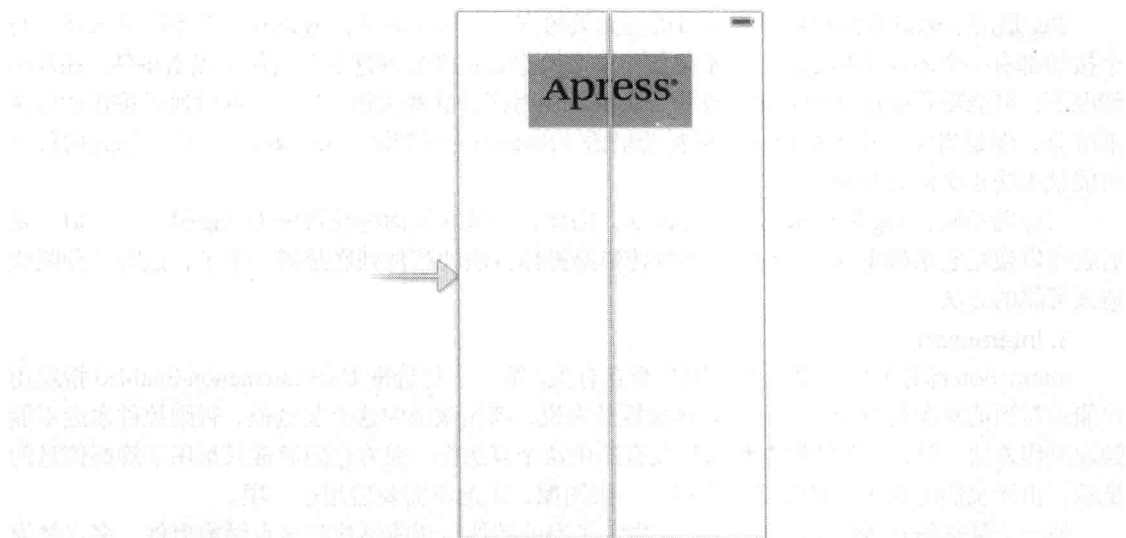


图 4-9 把图像视图调整到与图像尺寸大小一致之后，根据蓝色引导线把视图拖动到合适位置，并创建了一个保证它始终位于中央的约束

### 4.4.3 设置视图属性

选中图像视图，再次将注意力转向属性检查器。在检查器的 Image View 部分下面的是 View 部分。你也许已经猜到了，检查器顶部显示的是特定于当前所选对象的属性，之后则是更为通用的属性（它们适用于所选对象的父类）。在本例中，UIImageView 的父类是 UIView，所以，下一部分的标签是 View，其中包含了任何视图类都具有的属性。

#### 1. Mode

图像视图检查器中的第一个属性是名为 Mode 的弹出菜单。Mode 菜单用于选择内容在视图内部的显示方式。这决定了图像在视图内的对齐方式，以及是否缩放图像以适应视图大小。可以随意尝试各种选项，以查看效果，不过，目前来说，使用默认值 Scale To Fill 就可以。

记住，选择任何导致图像缩放的选项都可能增加处理开销，因此最好避免使用这些选项，尽量在导入图像之前就调整好它们的大小。如果希望以多种尺寸显示同一图像，最好是在项目中导入该图像不同尺寸的多个副本，而不是强制让 iOS 设备在运行时对它们执行缩放。当然，有时在运行时缩放图像可能更好，我们这里只是一般情况下的做法，而不是硬性规定。

#### 2. Tag

下一个值得注意的属性是 Tag，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 tag 属性，该属性只是一个数值，可以在这里设置，也可以在代码中设置。Tag 是开发者使用的，系统永远不会设置或修改它的值。如果为某控件或视图设置了一个 Tag 值，那么这个值会一直不变，除非你又修改了它。



Tag 是用于标识界面对象的一种与语言无关的方法，非常简便。假设有 5 个不同的按钮，每个按钮都有一个不同的标题，但你希望使用一个操作方法来处理这 5 个按钮的点击事件。在这种情况下，可能需要通过某种方式在操作方法被调用时区分这些按钮。当然，可以通过按钮的标题来区分，但是当应用程序被翻译为斯瓦希里语（Swahili）或梵语（Sanskrit）之后，原先的代码可能就无法正确区分按钮了。

与标题不同，Tag 属性永远都不会改变，因此，如果在界面构建器中为 Tag 赋了一个值，随后就可以使用它来确定通过 sender 参数传递给操作方法的控件到底是哪一个了，这是一种既快速又可靠的方法。

### 3. Interaction

Interaction 部分的两个复选框与用户交互有关。第一个复选框 User Interaction Enabled 指定用户能否与当前对象进行交互。对于大多数控件来说，都应该选中这个复选框，否则控件永远不能触发操作方法。但是，图像视图默认都没有选中这个复选框，因为它们通常只是用于静态信息的显示。由于我们的例子只是在屏幕上显示一张图像，因此不需要启用这一项。

另一个复选框是 Multiple Touch，它决定了当前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，比如许多 iOS 应用程序中用于缩放的双指捏合操作。第 13 章会详细讨论手势和多点触摸事件。由于我们例子中的图像视图完全不接受用户交互，因此也没有必要开启多点触摸事件，所以不勾选这个复选框就行了。

### 4. Alpha

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，也就是图像背后内容的可见度。Alpha 是取值范围为 0.0 ~ 1.0 的浮点数，0.0 是完全透明，1.0 是完全不透明。如果使用任何小于 1.0 的值，iOS 设备就会将视图绘制成具有一定的透明度，这样视图背后的任何对象都可以显示出来。如果值小于 1.0，即使图像背后没有任何内容，应用程序也会在运行时占用处理器周期来叠加半透明视图后面的空白区域。因此，除非有非常合适的理由，否则一般要将该值设置为 1.0。

### 5. Background

再下面一项是 Background，这个属性继承自 UIView，用于确定视图的背景颜色。对于图像视图来说，只有当图像没有填满整个视图、显示在屏幕纵横比为 4 : 3 的 iPad 上或者图像某些部分透明的情况下，这个属性才起作用。由于我们已经调整了视图大小，使其与显示的图像完全匹配，因此设置这个属性的值并不会产生任何可见效果，保留默认值即可。

### 6. Tint

接下来的控件可以让你指定所选视图的高光颜色。这个用来显示 GUI 组件在高亮或选中状态时的颜色。UIImageView 不会用到这个属性，因此目前只需要忽略它。之后我们会遇到其他使用此颜色的 GUI 组件。

### 7. Drawing

Tint 属性下方是一系列的 Drawing 复选框。第一个复选框名为 Opaque。这个复选框在默认情况下应该是选中的，如果没有，请单击选中它。Opaque 选中时相当于告诉 iOS 当前视图的背后

没有需要绘制的内容，同时允许 iOS 的绘图方法通过一些优化来加速当前视图的绘制。

你可能想知道为何在把 Alpha 设置为 1.0（不透明）之后还需要选中 Opaque 复选框。其原因是，Alpha 属性对需要绘制图像的视图部分起作用，但是如果某个图像并不能完全填充图像视图，或者图像上存在一些洞（由 Alpha 通道所致），那么视图下方的对象将仍然可见，不管 Alpha 的值是多少。而选中 Opaque 复选框之后，iOS 就会知道这个视图背后没有需要绘制的内容，因此就不必在视图对象背后的东西上浪费处理时间了。可以放心地选中 Opaque 复选框，因为之前已经选中了 Size To Fit（这使图像视图与它所包含的图像大小完全匹配）。

Hidden 复选框的作用显而易见，选中它之后，用户就看不到这个对象了。有时隐藏某个对象非常有用，比如本章稍后介绍的例子就需要隐藏开关和按钮。但在大多数情况下，都不会选中此选项。

下一个复选框是 Clears Graphics Context，这一项基本上不需要选中。如果选中它，iOS 会在实际绘制对象之前先使用透明的黑色绘制被对象覆盖的所有区域。考虑到性能问题，并且很少有这种需求，所以通常将其设置为关闭状态。你应确保这个复选框未被选中（默认情况下可能是选中的）。

Clip Subviews 是一个非常有趣的选项。如果你的视图包含子视图，并且这些子视图没有完全包含在其父视图的边界之内，那么这个复选框的值可以决定子视图的绘制方式。如果选中了 Clip Subviews，那么只有位于父视图边界之内的子视图部分会被绘制出来。如果不选中 Clip Subviews，那么子视图就会被完全绘制出来，不管子视图是否超出了父视图的边界。

Clip Subviews 默认是未选中的状态。这一行为似乎与实际情况刚好相反。从数学的角度来说，计算裁剪区域并显示子视图的部分区域是比较耗费资源的操作，而且在大多数情况下子视图不会超出父视图的边界。如果确实需要，可以启用 Clip Subviews，但考虑到性能，这个选项默认是关闭的。

最后要介绍的是 Autoresize Subviews 复选框，它告诉 iOS，在当前视图的大小发生变化时自动调整子视图的大小。这一项默认选中，保留默认值即可，因为我们例子中的视图大小不会发生改变，所以选不选中这一项都无所谓。

## 8. Stretching

下一部分名为 Stretching（拉伸）。可以忽略这一部分，因为只有在屏幕上调整矩形视图大小导致重绘视图时，才需要拉伸。该选项用于将视图的外边缘（例如按钮的边框）保持不变，仅拉伸中间部分，而不是均匀拉伸视图的全部内容。

这里需要设置 4 个浮点值，用于指定一个矩形可拉伸区域的左上角坐标以及大小，这 4 个浮点数的取值范围都是 0.0~1.0，代表整个视图大小的一部分。例如，如果希望每个边缘最外边的 10%是不可拉伸的，那么就将 X 和 Y 都设为 0.1，同时将 Width 和 Height 都设为 0.8。本例保留默认值：X 和 Y 为 0.0，Width 和 Height 为 1.0。大多数情况下都不需要改变这些值。

### 4.4.4 添加文本框

完成图像视图之后，该添加文本框了。从库中拖出一个文本框并将其移动到 View 中，将其放置

在图像视图的下方。使用蓝色引导线使文本框与右边缘对齐，同时紧挨着图像视图（参见图 4-10）。

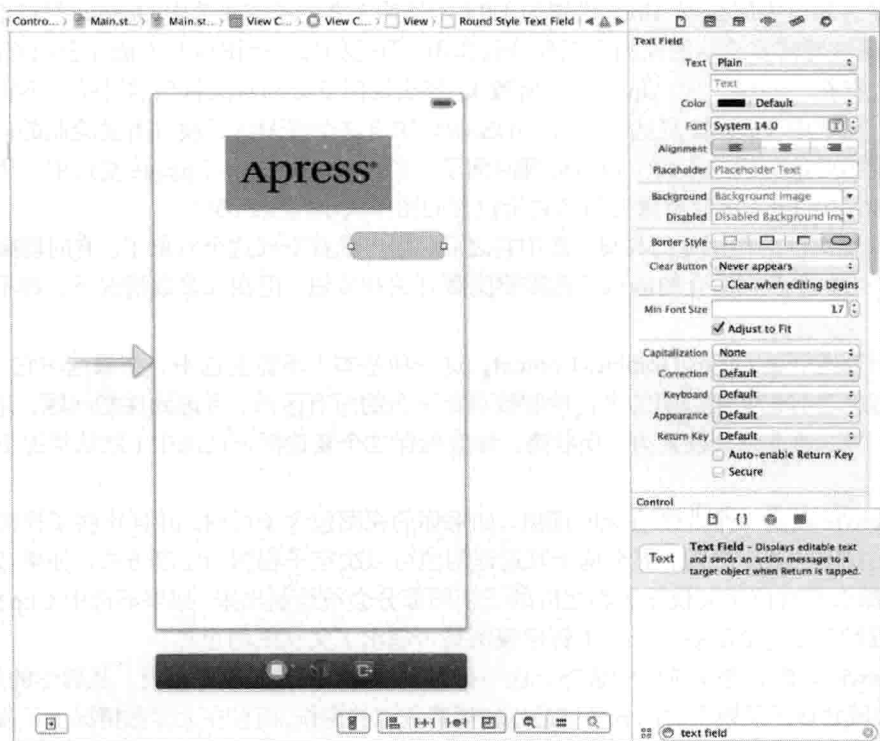


图 4-10 从库中拖出文本框，将其放置到 View 上位于图像视图下方的位置，并使其与右边的蓝色引导线对齐

将文本框移动到非常接近图像视图底部时，文本框的上方将出现一条水平的蓝色引导线。这条引导线告诉你已经离另一个对象足够近了。虽然可以将文本框放在此位置上，但是为了获得更加平衡的外观，将它下移一点会更好。请记住你随时可以使用界面构建器再次修改界面元素的位置和大小，而不需要更改代码或重新建立关联。

放置好文本框之后，从库中拖出一个标签放置到视图上，使标签与视图左边缘对齐，同时与之前放置的文本框水平对齐。注意，移动标签时会出现多条蓝色引导线，这样可以方便地让标签的顶部、底部、中间与文本框对齐。这里使用底部对齐的引导线（在中间的引导线附近拖动就会看到底线）来对齐标签和文本框，如图 4-11 所示。

双击刚才放置的标签，删除默认的 Label，输入 Name:（注意加上冒号），然后按 return 键提交更改。

接下来，再从库中拖动一个文本框放到视图中，并使用引导线将它放置在第一个文本框的正下方（参见图 4-12）。

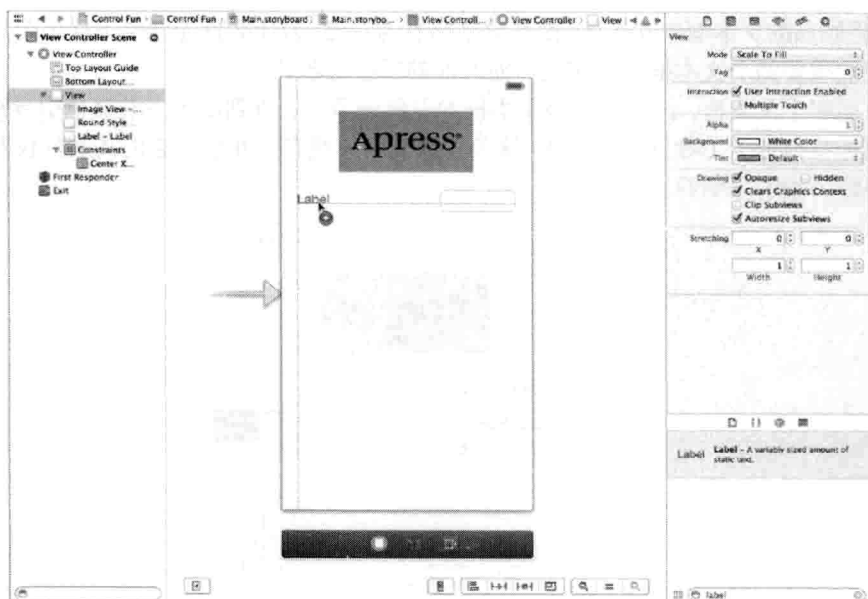


图 4-11 使用引导线对齐标签和文本框

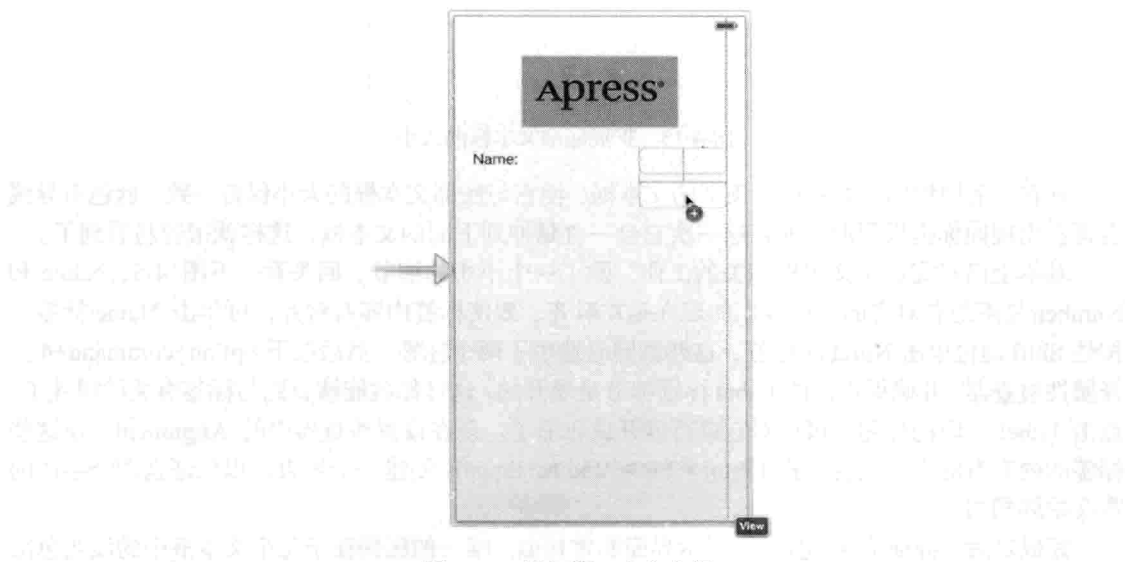


图 4-12 添加第二个文本框

放置好第二个文本框之后，从库中再拖出一个标签，将其置于左边第一个标签的正下方。再次使用蓝色中间引导线将它与第二个文本框对齐。双击新添加的标签，输入 Number:（再重申一遍，不要忘记冒号）。

现在,将底部的文本框向左扩展,使其紧靠标签的右侧。为什么首先调整下面那个文本框呢?这是因为我们想要两个文本框大小相同,而底部的标签比较长。

单击下面那个文本框,向左拖动该文本框的左侧调节点,直到出现一条蓝色引导线,表示该字段已经离标签很近了(参见图 4-13)。这条特殊的引导线有点儿小,它的高度仅仅和文本框一样,所以要睁大眼睛仔细看。

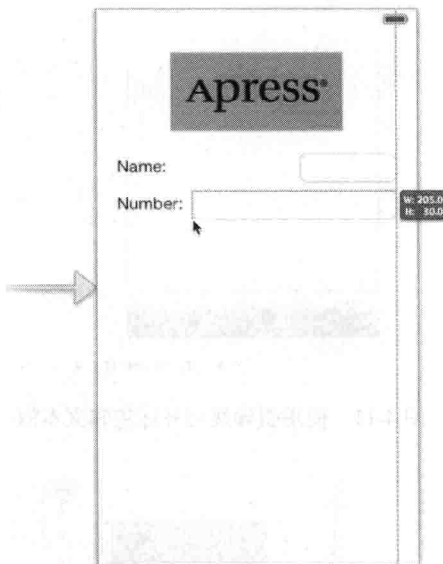


图 4-13 扩展底部文本框的大小

现在,采用相同的方法扩展顶部的文本框,使它与底部文本框的大小保持一致。蓝色引导线会再次出现向你提供帮助,而且这一次它会一直延伸到下面的文本框,这样就很容易看到了。

基本上已经完成了文本框相关的工作,除了一个小小的细节。回头看一下图 4-5, Name:和 Number:是不是右对齐的?而我们的现在是左对齐。要使标签内容右对齐,可单击 Name:标签,按住 Shift 键再单击 Number:标签,这样就同时选中了两个标签。然后按下 `option+command+4` 打开属性检查器,并确保里面的 Label 标题部分是展开的,这样你就能够看到与标签有关的属性了。点击 Label 分段的标题就可以对它进行展开或收缩了。现在设置检查器中的 Alignment,使这些标签的内容右对齐。然后选择 Editor>Pin>Widths Equally 创建一个约束,以保证这两个字段的宽度始终相同。

完成之后,界面应该与图 4-5 所示界面非常相似。唯一的区别在于每个文本框中的浅灰色文本,这是我们接下来要添加的。

选中上面那个文本框(位于 Name:标签的旁边),按下 `option+command+4` 打开属性检查器(参见图 4-14)。文本框是最复杂的 iOS 控件之一,同时也是最常用的一个控件。来看一下其中的设置项,从检查器的顶部开始。



图 4-14 文本框的属性检查器，当前显示的都是默认值

### 1. 文本框检查器设置

第一部分最上方是 Text 属性，它旁边的两个控件可以让你控制文本框中显示的内容。上方的下拉列表按钮中有纯文本（Plain Text）和属性文本（Attributed Text）两种类型供你选择，在属性文本中可以包含各种字体和不同的属性。目前保留下拉列表按钮为默认值 Plain。在下方可以设置文本框的默认值。输入的任何内容都将在应用程序启动时显示在文本框中，而不会是空白的。

之后是一系列用于设置字体和字体颜色的控件。保留 Color 的默认值黑色。注意，Color 弹出菜单分成两部分。右边的部分用于从一些预设的颜色中选择一种你想要的颜色，而左边的部分是一个调色板，可以在这里更精确地指定需要的颜色。

Font 的设置分为三部分。右边的控件可以用来增大或减小文本大小（每次增大或减小一个字号）。左边的部分可以用来手动编辑字体名称和字体大小。最后，点击带有 T 字样的图标可以打开一个弹出窗口，可以在这里设置各种字体属性。我们将 Font 的设置保留为默认的 System 14.0。

位于这些字段下方的是 3 个按钮，用于控制文本框中文本的对齐方式。我们保留这个字段的默认值，即左对齐（最左侧的按钮）。

接下来的部分是 Placeholder（占位符），可以在这里输入一些文本，文本框的内容为空时，Placeholder 的内容会以灰色文本显示在文本框中。如果空间不足，可以使用占位符来代替标签，或者使用占位符来告诉用户应在这个字段中输入什么内容。对于我们当前选中的文本框，可以输入 Type in a name 作为占位符，然后按 return 键提交更改。

接下来的两个字段是 Background 和 Disabled，仅在需要定制文本框的外观时使用，多数情况下，完全不必要也不建议使用它们。用户希望文本框以预期的方式显示。因此，可以直接跳过这些字段，保留它们的默认值即可。

再之后是4个名为 **Border Style** 的按钮，它们用于更改文本框边框的绘制方式。默认值（最右边的按钮）所创建的文本框样式是 iOS 应用中最惯用的。可以随意尝试这4种不同的样式。但在尝试完之后，记得将其重新设为最右边的按钮。

在边框设置下面是 **Clear Button** 弹出按钮，可以在这里设置何时出现清除按钮（clear button）。清除按钮是出现在文本框最右边的一个小的 X 形按钮。清除按钮通常用在搜索框和其他需要频繁改变内容的字段中，对于需要持久保存数据的文本框，一般不包含清除按钮。所以这里保留默认值 **Never appears**。

**Clear when editing begins** 复选框指定用户触摸此字段时是否清除已有的文本。如果选中了该复选框，则之前该字段中的任何内容都将被删除，用户需要重新输入。如果未选中该复选框，则之前的内容仍然保留在此字段中，用户能够编辑之前的内容。我们的例子中可以不选中这个复选框。

接下来的控件可以用来设置文本框在显示文本时可以使用的最小字号。目前保留默认值即可。

**Adjust to Fit** 复选框可以指定显示文本是否应随文本框尺寸的变化而变化。如果选中，那么整个文本在视图中都是可见的，即使文本大于所分配的空间。这一项会与最小字号的设置协同工作。无论文本框的大小如何，其中的文本都不会小于最小字号。指定最小字号可以确保文本不会因为过小而影响可读性。

接下来的部分，用于定义使用此文本框时键盘的外观及行为。我们期望用户输入一个名字，因此将 **Capitalization** 的值更改为 **Words**。这可以保证每个输入的单词都会自动转换为首字母大写，而这正符合名字的要求。

之后三个弹出选项 **Correction**、**Keyboard**、**Appearance**，都可以保留为默认值。可以花点时间看看这些设置的作用。

接着是 **Return Key** 弹出选项。**return** 键是键盘右下方的一个键，它的标签会根据用户正在进行的操作发生变化。例如，如果你正在向 Safari 的搜索框中输入文本，它就会显示 **Search**。而在我们这种文本框与其他控件共享屏幕的应用程序中，**Done** 是最合适的选择。这里就改为 **Done**。

如果选中了 **Auto-enable Return Key** 复选框，那么在文本框内容为空时 **return** 键将被禁用，直到至少在文本框中输入一个字符。取消选中此复选框，因为我们希望允许文本框保留为空（用户什么也不输入）。

**Secure** 复选框指定是否在文本框中显示已输入的字符。如果此文本框要用作一个密码字段，那么应该选中此复选框。我们这里保留其未选中状态即可。

接下来的部分用于设置继承自 **UIControl** 的控件属性，但它们通常不适用于文本框（除了 **Enabled** 复选框），也不会影响文本框的外观。我们希望启用这些文本框，以使用户能够与它们交互。这里保留默认的所有设置即可。

检查器上的最后一部分是 **View**，你应该比较熟悉。它与之前介绍的图像视图检查器上的同名部分相同。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，它们都具有此部分中的属性。跟之前所做的一样，选中 **Opaque**，不要选中 **Clears Graphics Context** 和 **Clip Subviews**，原因之前我们已经讨论过了。



## 2. 设置第二个文本框的属性

接下来,单击 View 窗口中下方的文本框(位于 Number:标签的旁边),然后返回检查器。在 Placeholder 字段中,输入 Type in a number 并且确保 Clear When Editing Begins 未选中。再往下一点,单击 Keyboard 弹出菜单。我们希望用户只输入数字,而不含字母,因此选中 Number Pad。完成这些设置之后,用户所使用的键盘将只有数字,这意味着用户不能输入字母、符号等非数字内容。不需要为数字键盘设置 Return Key,因为这种样式的键盘没有 return 键,所以检查器上的其他选项都可以保留默认值。跟之前一样,选中 Opaque,而 Clears Graphics Context 和 Clip Subviews 都不需要选中。

### 4.4.5 创建并关联输出接口

目前已经基本准备好对该应用进行第一次测试了。对于界面设计的第一部分,剩下的工作就是创建和关联输出接口。界面上的图像视图和标签并不需要输出接口,因为无需在运行时改变它们。而两个文本框是被动控件,其中保存了代码中需要用到的数据,所以需要为它们分别创建输出接口。

你可能还记得上一章提到,Xcode 支持在辅助编辑器中同时完成输出接口的创建和关联工作。选择名为 Editor 的工具栏中间的按钮或者选择 View>Assistant Editor>Show Assistant Editor 就可以打开辅助编辑器。

确保在项目导航面板中选中了分镜文件。如果没有足够的屏幕空间,可能需要通过 View>Utilities>Hide Utilities 在这个环节中隐藏实用工具面板。打开辅助编辑器后,编辑面板被分成了两部分,一半是界面构建器,另一半是 BIDViewController.h 或 BIDViewController.m(参见图 4-15)。右侧的新编辑区域就是辅助区域。

可以看到辅助编辑器的顶端部分包含一个跳转栏(类似于普通的编辑器面板)。这个跳转栏有一个非常重要的辅助功能,就是它能“智能地”提供一组选项,Xcode 根据主视图中内容让你能在相关的各个文件中进行切换。默认情况下,它显示一组标为“Automatic”的文件,其中包含了编辑器中所有与当前所选内容有关的.h 和.m 文件。在本示例中它包含了控制器类的两个源代码文件。花些时间随意点击辅助编辑器顶部跳转栏中的内容,以便对其有所认识。对跳转栏以及其中所显示的文件有一定了解之后,再继续读下面的内容。

现在就到好玩的部分了。确保辅助编辑器中显示的是 BIDViewController.m 文件(如果需要的话,可以使用跳转栏返回该文件)。然后按住 control 键,用鼠标将视图中上面那个文本框拖向 BIDViewController.m 源代码,拖到@interface 这一行的下面,可以看到一条灰色的弹出信息,信息内容是 Insert Outlet, Action, or Outlet Collection(参见图 4-16)。松开鼠标按键,会看到一个与上一章相同的弹出窗口。我们想要创建一个名为 nameField 的输出接口,因此在 Name 字段中输入 nameField,然后按下 return 键或点击 Connect 按钮。

现在,BIDViewController 中有了一个 nameField 属性,而且它与上面那个文本框关联了起来。以同样的方式为第二个文本框创建并关联输出接口,将对应的属性命名为 numberField。

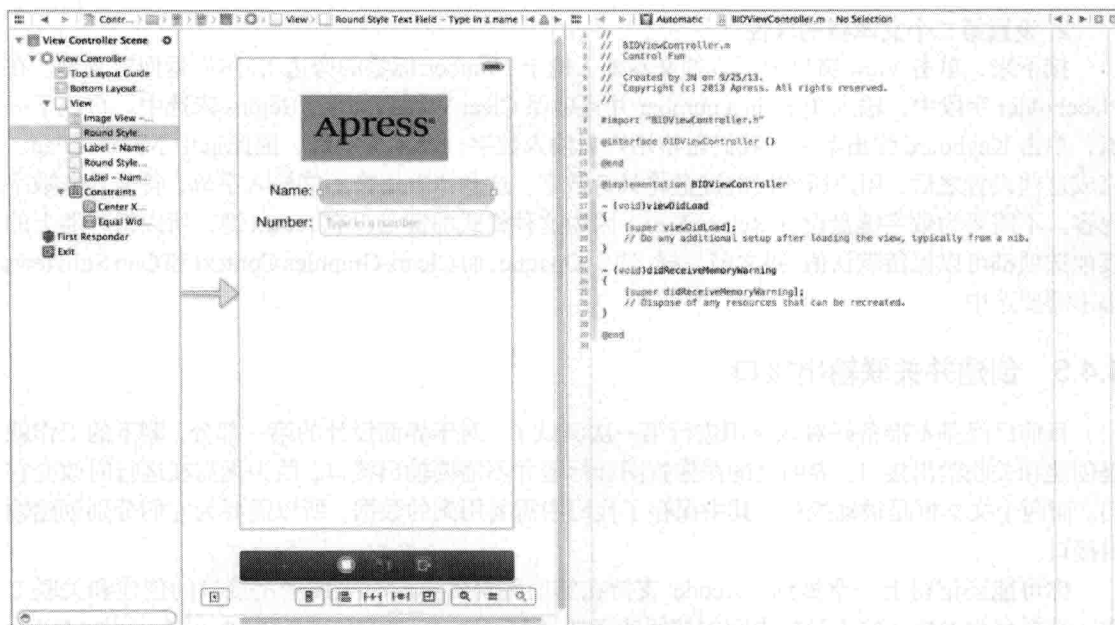


图 4-15 打开了辅助编辑器的编辑面板。可以在右边看到辅助区域，其中显示了 BIDViewController.h 中的代码

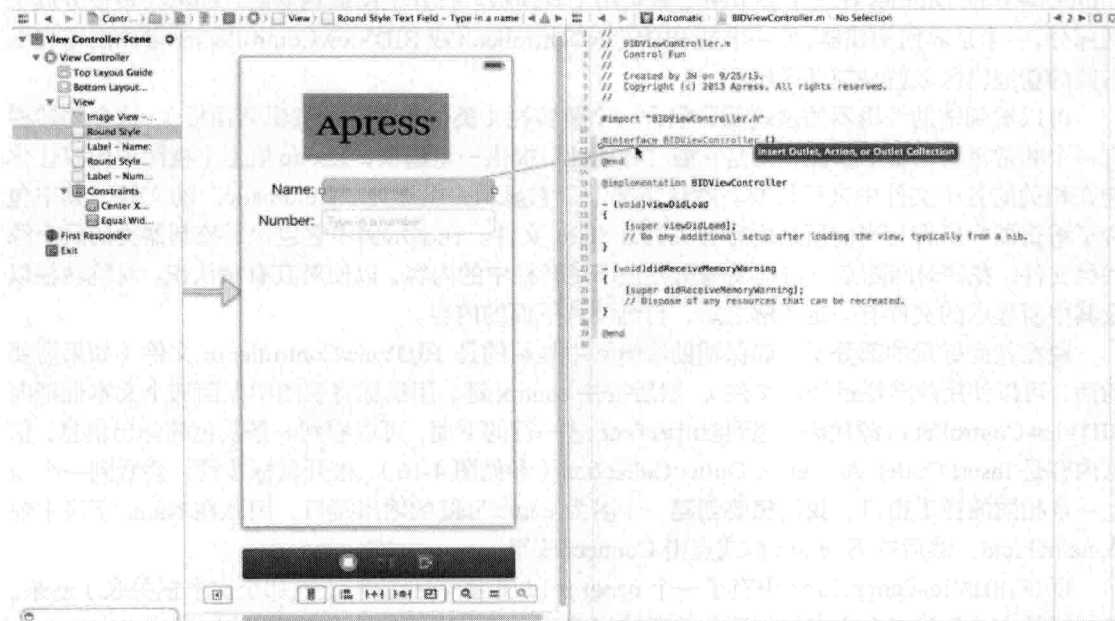


图 4-16 打开辅助编辑器，按下 control 键拖动鼠标，将 nameField 字段与输出接口关联起来

## 4.5 关闭键盘

下面再来看看应用的运行情况。从 Xcode 的 Product 菜单中选择 Run，应用程序就会出现在 iPhone 模拟器中了。单击 Name 文本框，此时会出现传统的键盘，输入一个名字。现在，单击 Number 字段，键盘将切换为纯数字键盘（参见图 4-17）。Cocoa Touch 免费提供了所有这些功能，而我们只需要把文本框添加到用户界面上就可以了。



图 4-17 触摸文本框或数字字段时键盘将自动显示

非常顺利！但是，还有一个小问题。应该如何关闭键盘呢？自己先想一下，下一小节就会介绍解决方案。

### 4.5.1 输入完成后关闭键盘

iOS 设备上的键盘是虚拟的，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。用户按下键盘上的 Done 按钮时，会产生一个 `Did End On Exit` 事件，此时需要让文本框交出控制权，以关闭键盘。为了实现这个功能，需要在控制器类中添加一个操作方法。

在项目导航面板中选择 `BIDViewController.h`，添加以下粗体显示的代码：

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

@interface BIDViewController : UIViewController

- (IBAction)textFieldDoneEditing:(id)sender;

@end
```

在项目导航面板中选中该头文件之后，你可能会发现，先前打开的辅助编辑器能根据主编辑面板中选择的源代码文件自动切换，显示了与所选文件相对应的文件。如果选择的是 `.h` 文件，那

么辅助编辑器会自动显示与之对应的.m文件,反之亦然。这是Xcode中一项非常便捷的功能。现在辅助视图中显示了BIDViewController.m文件,可以在这里实现刚刚添加的操作方法。

在BIDViewController.m的最后(@end之前)添加这个操作方法:

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

在第2章中已经了解到,第一响应者就是当前正在与用户进行交互的控件。在这个新方法中,我们通知该控件放弃作为第一响应者的控制权,将其返还给用户之前操作的控件。一个文本框失去了第一响应者状态后,与之关联的键盘也将消失。

保存刚才编辑的两个文件,回到分镜,把这个操作方法与两个文本框关联起来。

在项目导航面板中选择Main.storyboard,单击Name文本框,按下option+command+6打开连接检查器。这次不使用上一章用过的Touch Up Inside事件,而是使用Did End On Exit事件,因为这个事件会在用户按下文本键盘上的Done按钮时触发。

将Did End On Exit旁边的圆圈拖动到黄色的视图控制器图标上(位于你当前正在使用的视图下方)。这时会出现一个小的弹出菜单,从中可以看到刚刚添加的操作方法。单击textFieldDoneEditing:方法以选中它。也可以将Did End On Exit旁边的圆圈拖向辅助编辑器视图中的textFieldDoneEditing:方法。对另一个文本框重复以上步骤,然后保存所做的修改,按下command+R再次运行该应用。

当模拟器出现之后,单击Name字段,输入一些内容,然后按下Done按钮。不出所料,键盘随之消失了。那么Number字段也是这样吗?可是这个字段的Done按钮在哪里呢(参见图4-17)?

并非所有的键盘布局都有Done按钮。可以强制用户按下Name字段,然后再按Done按钮,但是这样的用户体验非常糟糕。我们显然希望应用程序具有很好的用户体验。下面就看看如何解决这个问题。

## 4.5.2 通过触摸背景关闭键盘

还记得苹果公司的iPhone应用是如何处理这种情况的吗?在大部分保存文本框的地方,点击视图中没有活跃控件的任何地方键盘都会消失。我们如何实现此功能呢?

答案可能会令你惊讶,因为它非常简单。视图控制器有一个view属性,是从UIViewController继承来的。这个view属性对应于nib文件中的View。view属性指向分镜中的一个UIView实例,这个实例是用户界面中所有元素的容器。它在用户界面中没有可见的外观,但它覆盖了整个iPhone窗口,位于所有其他用户界面对象“背后”。它有时也称为容器视图(container view),因为它的主要用途是保存其他视图和控件。总而言之,容器视图就是用户界面的背景。

使用界面构建器,可以更改view所指的对象所属的类,将它的底层类由UIView更改为UIControl。因为UIControl是UIView的子类,所以非常适用于将view属性连接到UIControl实例。请记住,当一个类继承自另一个类时,子类其实是父类的一个更加具体的版本,所以UIControl是一个UIView。如果从UIView类创建实例更改为从UIControl类创建实例,就获得了触发操作

方法的能力。但在此之前，需要创建在点击背景时需要调用的操作方法。

需要在控制器类中再添加一个操作方法。将以下代码添加到 `BIDViewController.m` 文件中 `@end` 语句上面的位置：

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

这个方法只是告诉两个文本框放弃第一响应者状态（如果它们处于该状态的话）。即使控件并非第一响应者，对其调用 `resignFirstResponder` 方法也是非常安全的，所以可以在这两个文本框上都调用该方法，而不需要检查它们是否为第一响应者。请注意，与之前同时在头文件和实现文件中添加操作方法不同，这次我们只将它放在了实现文件中。现在的 Xcode 已经足够智能了，即便跳过多余的头文件声明也可以对 GUI 和代码之间进行关联。不过假如我们要在其他类中也能使用这个方法的话，仍然需要在头文件中声明这个方法。

**提示** 在编码时将多次在头文件和实现文件之间切换。幸好，除了辅助编辑器所提供的方便性之外，Xcode 还提供了一个组合键，用于在一个文件与其对应的另一个文件之间快速切换。默认的组合键是 `control+command+Up`，可以使用 Xcode 的首选项将它更改为自己想要的任何组合键。

保存文件，再次选中分镜。确保文件大纲是展开状态（可以点击编辑区域左下方的三角图标打开或关闭它）。单击 **View** 以选中它。不要选择视图的子项，我们需要的是容器视图本身。

接下来，按下 `option+command+3` 打开身份检查器（参见图 4-18）。在这里，可以更改分镜中任何对象实例的底层类。

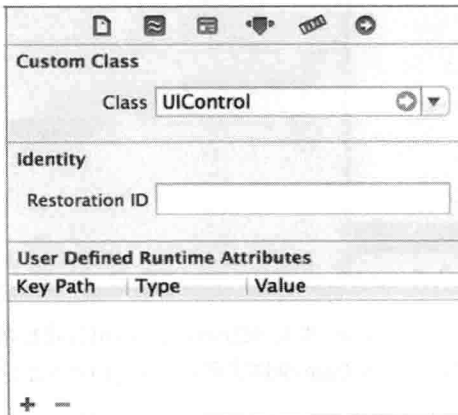


图 4-18 将界面构建器切换到列表视图，选择 **View**。然后切换到身份检查器，在这里可以修改分镜中任何对象实例的底层类

名为 Class 的字段目前的值为 UIView，如果不是，可能是你没有选中容器视图。现在，将其更改为 UIControl，按下 return 提交修改。所有能够触发操作方法的控件都是 UIControl 的子类，所以通过更改底层类，这个视图就可以触发操作方法。可以按 option+command+6 调出关联检查器来验证这一点。现在应该会看到上一章将按钮连接到操作时看到的所有事件。

把 Touch Down 事件拖到 View Controller 图标上（参见图 4-19），然后选择 backgroundTap: 操作方法。现在，触摸视图中没有活跃控件的任何位置都将触发新的操作方法，这样就可以关闭键盘了。用这种方式关联到视图控制器与关联到代码中的方法是完全等价的。对于视图控制器的分镜来说，View Controller 就是视图控制器类的一个实例对象，所以这只是一种达到相同目的的不同方式而已。

**注意** 你可能想知道为什么我们选择 Touch Down，而不是像上一章那样选择 Touch Up Inside。因为背景并不是按钮。它并不是用户眼中的控件，所以大多数用户不会试图把手指滑动到其他地方来取消操作。



图 4-19 通过将视图的 class 从 UIView 改为 UIControl，就可以在标准事件上触发操作方法了。

这个例子中把视图的 Touch Down 事件关联到 backgroundTap: 操作方法

保存分镜文件，再次编译和运行应用程序。这一次，点击键盘上的 Done 按钮或者是触摸没有活跃控件的任何地方都可以关闭键盘了，用户比较习惯后一种方式。

非常好！这一小节到此就完成了，准备好学习下一组控件了吗？

### 4.5.3 添加滑动条和标签

现在是时候添加一个滑动条及其附属的标签了。记住，移动滑动条位置时，标签的值将会随之改变。在项目导航面板中选择 Main.storyboard，我们将为应用程序用户界面添加更多控件。

在添加滑动条之前，先为我们的设计腾出一点空间。用于确定顶部文本框与其上方图像之间间距的蓝色引导线，实际上已给出了最小的建议间距。换句话说，蓝色引导线告诉你“间距不要小于此距离”。参照图 4-1，将两个文本框和它们的标签稍微向下拖动一点。接下来添加滑动条。

从对象库中拖出一个滑动条，并将其放置在 Number 文本框下方，以最右侧的蓝色引导线为停止点，在底部的文本框下方留一点空间给标签。滑动条在水平方向上大约位于视图的中间位置。单击新添加的滑动条以选中它，这时候应该可以看到属性检查器了，如果没有看到，可以按下 option+command+4 打开对象的属性检查器，如图 4-20 所示。

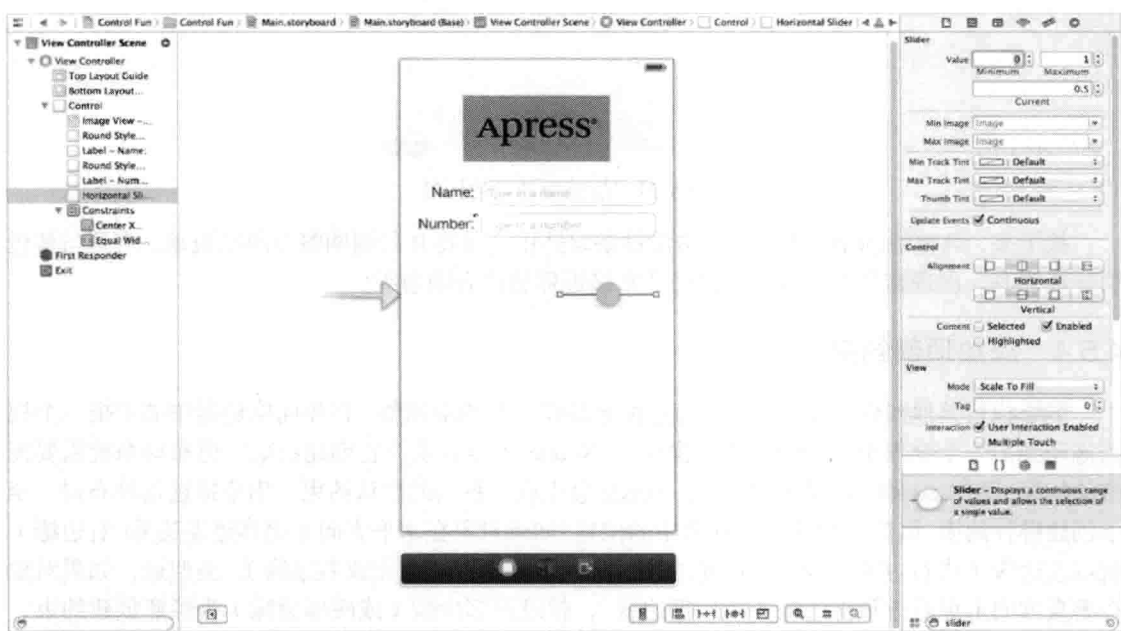


图 4-20 显示着滑动条默认属性的检查器

可以通过滑动条选择特定范围内的数值。使用检查器可以设置初始值，最小值为 1，最大值为 100，当前值是 50。选中 Update Events 和 Continuous 复选框，这样可以确保滑动条的值改变时可以触发一系列连续的事件。目前只需要了解这些设置就可以了。

在滑动条旁边放置一个标签，需使用蓝色引导线保持它与滑动条水平对齐，并保持其左侧边缘与视图的左侧边缘对齐（参见图 4-21）。

双击新添加的标签，将其文本从 Label 更改为 100。这是滑动条的最大值，也可以使用它确定滑动条的正确宽度。由于“100”比“Label”短，界面构建器会自动把标签的宽度减小，就如



同你拖着右边正中间的调整点向左移动一样。虽然这种行为是自动的,但是你仍然可以调整标签的大小。如果希望使用工具自动确定一个最合适的大小,可以按下 `command+=` 或者是选择 `Editor > Size fo Fit Content`。

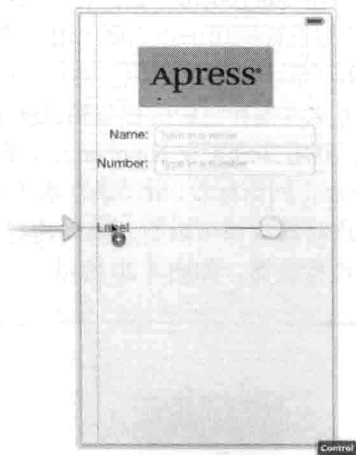


图 4-21 放置滑动条的标签

接下来,调整滑动条的大小,单击滑动条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。出现蓝色引导线表明已经非常接近标签的右侧边缘。

#### 4.5.4 添加顶部约束

继续执行其他操作之前,需要先为这种布局作一些约束调整。在界面构建器中如果把一个视图拖动到另一个视图中(类似刚才的操作),Xcode 不会自动为它创建约束。但布局系统需要完整的约束,所以编译应用程序的时候,Xcode 会生成一系列的默认约束,用于描述这种布局。至于创建哪种约束,取决于对象在父视图中的位置。如果对象在水平方向上更靠近左边缘(右边缘),就以左边缘(或右边缘)为参照创建约束,将其固定在左边缘(或右边缘)。类似地,如果对象在垂直方向上更靠近顶部边缘(或底部边缘),就以顶部边缘(或底部边缘)为参照创建约束,将其固定在顶部边缘(或底部边缘)。如果对象是完全居中的,它就会被约束固定在中间位置。

让事情更趋复杂的是,Xcode 还可能根据同一父视图中某个对象的一个或多个兄弟对象的位置,自动创建约束,固定这个对象。这个自动行为也许不是你想要的,因此通常你最好在应用程序编译之前就在界面构建器中创建好整套的约束。

我们来看看现在已经有了一些东西了。找找针对特定视图的所有约束,试着选中它并打开尺寸检查器。如果你选中了标签、文本框或者滑动条中的任一项,你会看到尺寸检查器提示了所选视图没有约束的一条信息。事实上我们构建的 GUI 上只有一个约束:用来绑定图像视图在容器视图中水平对齐。选中容器视图或图像视图就可以在检查器中看到这个约束。

我们实际上需要一整套的约束以在编译时能告诉布局系统如何精确地管理所有视图和控件。

好在这项工作很容易就能完成。从容器视图的左上角点击并向右下角拖拽出一个选择框,以选中所有的视图和控件。所有的内容选中之后,执行 `Editor>Resolve Auto Layout Issues>Add Missing Constraints` command 菜单项。完成这步之后,你会看到所有的视图和控件现在都有一些细的蓝色实线连接着其他控件或容器视图。每条实线都代表着一个约束。我们没有让 Xcode 在编译时生成约束,此时创建它们的好处在于可以根据需要修改每个约束。在本书中我们会讨论很多关于如何调整约束的内容。

通常情况下,并不需要对这些约束进行特别修改,就能保证这种布局适用于所有设备。但是,自从 iPhone 5 发布之后,就不一样了。现在的 iPhone 和 iPod touch 拥有两种不同的屏幕大小,这意味着,系统在加载分镜或 nib 文件中 GUI 一系列组件时,需要针对不同的屏幕调整用户界面的内容,并且重新应用所有的约束。Xcode 中创建的用户界面默认都是以 iPhone 5 的尺寸为基准的,也叫做 iPhone Retina 4 (因为屏幕是 4 英寸大)。因此,如果某些视图的位置是参照父视图的顶部,而另外一些视图是参照父视图的底部或者中心,把应用放到 iPhone 4 (iPhone 4 的屏幕也称为 iPhone Retina 3.5) 上运行时,很可能会出现意想不到的情况——某些控件可能在垂直方向上相互覆盖了。

对于我们当前的 GUI 来说,还没遇到这个问题。通过以下步骤可以证实了。在编辑用户界面时你会在编辑区域的右下看到一行按钮,使用它们可以执行一些通用操作。当你把鼠标指针移到最左边的按钮上会看到 `Apply Retina 3.5-inch Form Factor` 的提示信息。点击它就会看到容器视图的尺寸发生了改变,但所有的视图和控件都停留了合适的位置上。再次点击按钮就会切换到视网膜 4 英寸的布局格式,你会看到一切依然正常。在本书的后面,我们将会处理需要在这块区域调整一些 GUI 的情况。

#### 4.5.5 创建并关联操作方法和输出接口

最后要为这两个控件关联输出接口和操作方法。我们需要一个指向该标签的输出接口,以便在滑动条滑动时能够更新标签的值,另外还需要一个操作方法,它将在滑动条位置发生变化时被调用。

确保正在使用辅助编辑器编辑 `BIDViewcontroller.m` 文件,然后按住 `control` 键从滑动条拖动到辅助编辑器中 `@end` 声明上方的位置。弹出窗口出现后,将 `Connection` 弹出菜单改为 `Action`,在 `name` 字段中键入 `sliderChanged`。再把 `Type` 字段的值置为 `UISlider`,然后按下 `return` 就完成了操作方法的创建和关联。

接着按住 `control` 键,从新添加的标签(显示文字 100 的那个)拖向辅助编辑器。这次,拖到顶端 `@interface` 和 `@end` 之间,最后一个属性的下方位置。弹出窗口出现后,在 `Name` 文本框中输入 `sliderLabel`,然后按下 `return` 就完成了输出接口的创建和关联。

#### 4.5.6 实现操作方法

虽然 Xcode 为我们创建并关联了操作方法,但是仍然需要我们自己编写操作方法的实现代码,这样操作方法才能执行既定工作。保存你的文件,然后在项目导航面板中单击 `BIDView-`

Controller.m, 找到 sliderChanged: 方法 ( 它应该是空的 ), 添加如下代码:

```
- (IBAction)sliderChanged:(UISlider *)sender {  
    int progress = lroundf(sender.value);  
    self.sliderLabel.text = [NSString stringWithFormat:@"%d", progress];  
}
```

这个方法的第一行是获取滑动条的当前值, 将其四舍五入到最接近的整数, 然后把这个整数赋给一个整型变量。第二行代码是根据这个整型变量创建一个字符串, 然后把这个字符串赋给标签。

这样就可以处理控制器对滑动条滑动做出的响应了, 但是为了更好地一致性, 需要保证用户触碰滑动条之前标签也能正确显示滑动条的值。在 viewDidLoad 方法中添加如下代码:

```
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
    // 视图加载完成之后 (通常是从分镜加载), 做一些额外的设置  
    self.sliderLabel.text = @"50";  
}
```

这个方法会在 app 加载完分镜的视图之后, 显示在屏幕上之前执行。刚刚添加的这行代码能够保证用户立即看到正确的初始值。

保存文件, 然后按下 command+R 在 iPhone 模拟器中构建并运行这个应用, 试用一下滑动条。移动滑动条时, 应该可以看到标签文本的实时变化。这样又实现了一小块功能。

但是, 如果把滑动条向左移动 (使其值小于 10) 或者是向右移动 (使其值变为 100), 就会有奇怪的事情发生。当显示的数值减少到只有一位数时, 标签会在水平方向上向左收缩, 而当显示的数值变成三位数时, 标签会在水平方向上向右扩张。现在, 除了标签包含的文本, 我们根本看不到这个标签本身, 所以看不到标签大小的改变, 但是能够看到滑动条的大小随着标签的变化在变小或者变大。滑动条维护着一个约束, 这个约束可以保证滑动条与标签之间的距离始终不变。

我们并没有要求滑动条这么做, 是吧? 并非如此。实际上这是界面构建器在创建响应式可移动的图形用户界面时产生的一个副作用。之前我们创建了一些默认的约束, 这里就是其中一个约束在起作用。界面构建器创建的某个约束, 会使这些元素在水平方向的间距始终保持不变。

幸好, 可以创建自己的约束来覆盖这种行为。回到 Xcode, 在分镜中选中滑动条, 然后从菜单中选择 Editor>Pin>Width, 这样就创建了一个高优先级的约束, 这个约束告诉布局系统 “不要改变滑动条的宽度”。现在再次按下 command+R 来构建并运行应用, 可以看到, 在前后移动滑动条时, 滑动条已经不再扩张和收缩了。

本书中还会看到很多使用约束的例子。但是现在先来看一下如何实现开关控件。

## 4.6 实现开关、按钮和分段控件

再次返回到 Xcode。是否有点头晕目眩了? 这种来回切换可能看起来有点奇怪, 但在开发过程中这是非常普遍的, 需要经常在 Xcode 的源代码文件、分镜或 nib 文件之间进行切换, 也需要经常在 iOS 模拟器中测试开发中的应用。

应用程序将有两个开关，开关是一种比较小的控件，它只有开和关两种状态。还要添加一个分段控件来控制开关的隐藏和显示。除此之外，还需要添加一个按钮，当点击分段控件的右侧时显示该按钮。下面来实现这些功能。

回到分镜，从对象库中拖出一个分段控件（参见图 4-22），将其放置在 View 窗口中滑动条下方的位置。

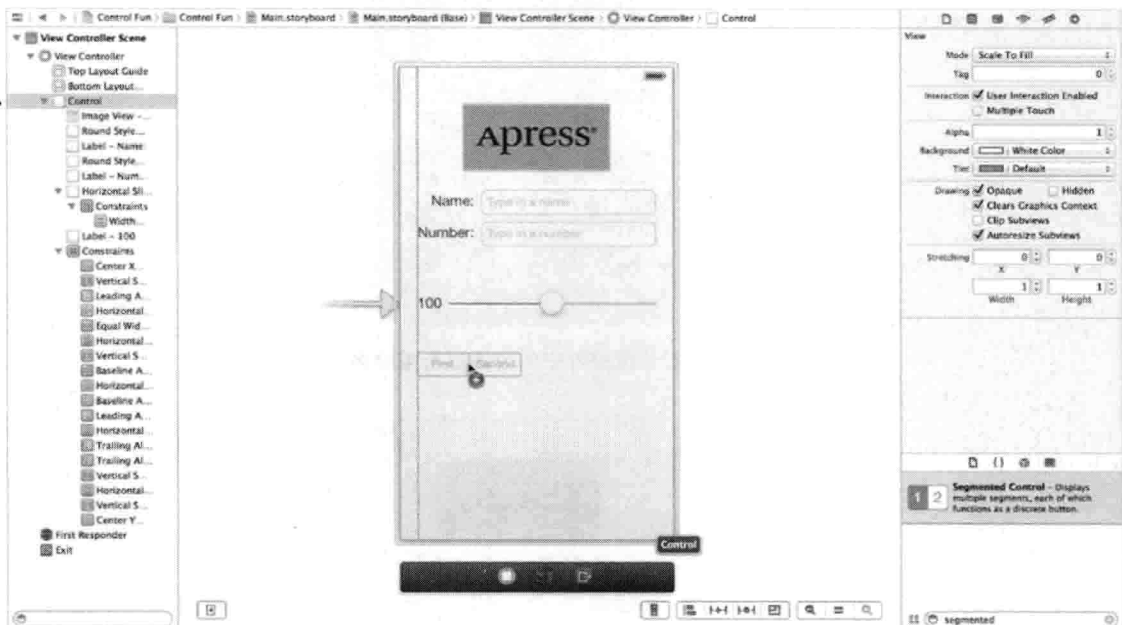


图 4-22 这里我们从库中拖出一个分段控件，将其置于父视图的左侧。然后调整分段控件的大小，将其拉伸到视图的右侧

**提示** 关于控件之间的间距给你提个建议，来看一下带有 Apress 商标的图像视图。在图像视图的上方和下方保留有同样大小的空间。对滑动条也要这么做，在滑动条的上方和下方保留同样大小的空间。

扩展分段控件的宽度，使其从视图左边缘一直拉伸到右边缘。双击分段控件上的 First 分段，将其重命名为 Switches。之后对 Second 段重复这个过程，将其重命名为 Button（参见图 4-23）。

#### 4.6.1 添加两个带标签的开关

接下来，从库中拖出一个开关，将其放在视图上分段控件下方靠近左侧边缘的位置。然后拖出第二个开关并放在靠近右侧边缘的位置，与第一个开关水平对齐（参见图 4-24）。

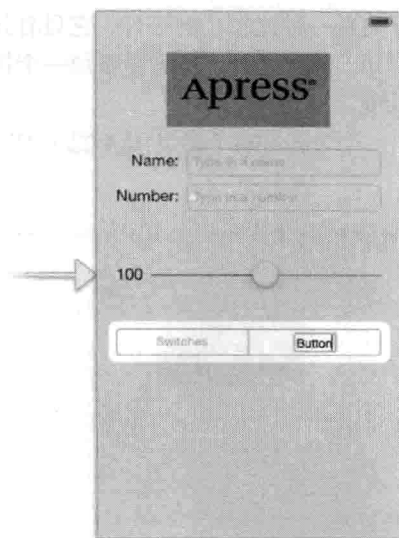


图 4-23 对分段控件中的分段进行重命名

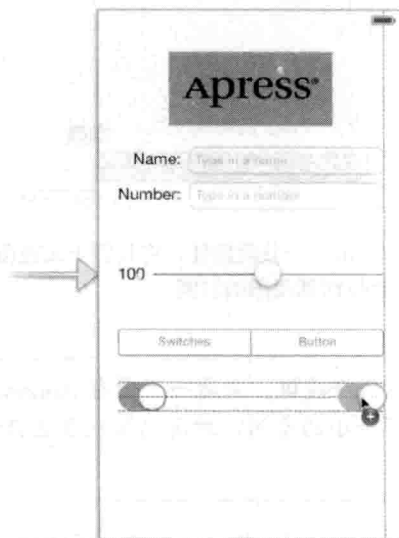


图 4-24 把开关添加到视图中

**提示** 在界面构建器中，按住 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多个实例，只需从库中拖出一个对象，然后再按住 option 键拖出多个副本即可，这种方式比较快捷。

### 4.6.2 为开关创建并关联输出接口和操作方法

添加按钮之前,先为两个开关控件创建输出接口并进行关联。我们稍后要添加的按钮实际上将位于开关控件上层,按钮会把开关遮盖住,这样一来就很难按住 control 并拖动开关了,所以我们在添加该按钮之前就处理好开关控件的关联。由于按钮和开关不会同时可见,所以将它们放在同一个物理位置不会存在问题。

使用辅助编辑器,按下 control 键,把左边的开关拖向 BIDViewController.m 文件中最后一个输出接口的下方。出现弹出窗口后,将输出接口命名为 leftSwitch,然后按下 return。对另一个开关重复此步骤,将其输出接口命名为 rightSwitch。

现在,单击左边的开关再次选中它,再次按住 control 键并将其拖向辅助编辑器。这一次,拖到@end 声明的上方然后松开鼠标。弹出窗口出现后,将新的操作方法命名为 switchChanged:,把 sender 参数的 Type 改为 UISwitch。然后按下 return 就创建了一个新的操作方法。对右边的开关重复这个过程,但有一点差异:不要为右边的开关创建新的操作方法,而是将它拖向之前创建的 switchChanged:操作方法以进行关联。和上一章一样,我们将使用同一个操作方法来处理这两个开关的事件。

最后,如之前所做的一样,按住 control 键并把分段控件拖向辅助编辑器中的@end 声明上方,插入一个名为 toggleControls:的操作方法。这一次,将 sender 参数的 Type 设置为 UISegmentedControl。

### 4.6.3 实现开关的操作方法

保存分镜,并将注意力转移到 BIDViewController.m(在辅助视图中已经打开了),找到 Xcode 自动添加的 switchChanged:方法,添加如下代码:

```
- (IBAction)switchChanged:(UISwitch *)sender {
    BOOL setting = sender.isOn;
    [self.leftSwitch setOn:setting animated:YES];
    [self.rightSwitch setOn:setting animated:YES];
}
```

用户按下任何一个开关都会调用 switchChanged:方法。在该方法中,我们简单地获取 sender 参数的 isOn 属性值(sender 代表被按下的那个开关),然后使用这个值来设置两个开关。此处的逻辑是设置一个开关的值会同时改变另一个开关的值,让它们始终保持同步。

在这个例子中,sender 始终是 leftSwitch 或 rightSwitch 的其中之一,所以你可能会感到奇怪:为什么要同时设置它们两个?原因之一是考虑到实践性。比起判断调用该方法的开关,然后设置另一个开关,每次都同时设置两个开关所需的工作量更少。不管是哪个开关调用这个方法都能够被设置为正确的值,对一个开关重复设置相同的值并不会造成任何影响。

#### 添加按钮

接下来,从库中拖出一个 Button 放到视图中。直接将这个按钮叠放到最左侧的开关控件上面,将其与左侧边缘被覆盖的开关对齐,并将其与两个开关在垂直方向上顶部对齐(参见图 4-25)。

现在按住按钮最右边中间位置的调整手柄并一直向右拖动,直到按钮的右侧边缘与用于标示

屏幕右侧边缘的蓝色引导线对齐。按钮应该完全覆盖这两个开关的所处的空间，但因为按钮默认是透明的，因此你还是能看到开关的（参见图 4-26）。

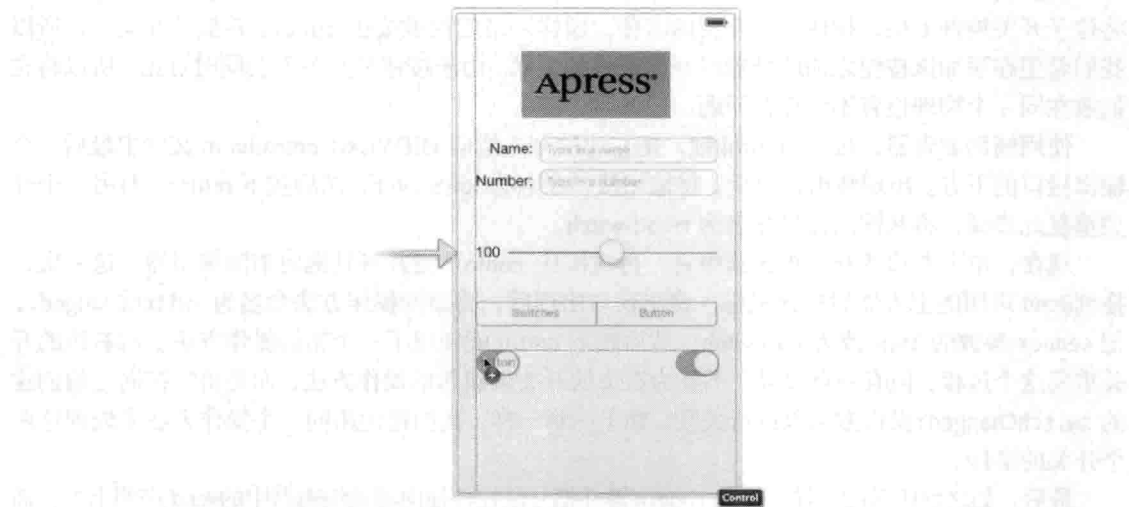


图 4-25 将一个圆角矩形按钮覆盖到现有开关上面

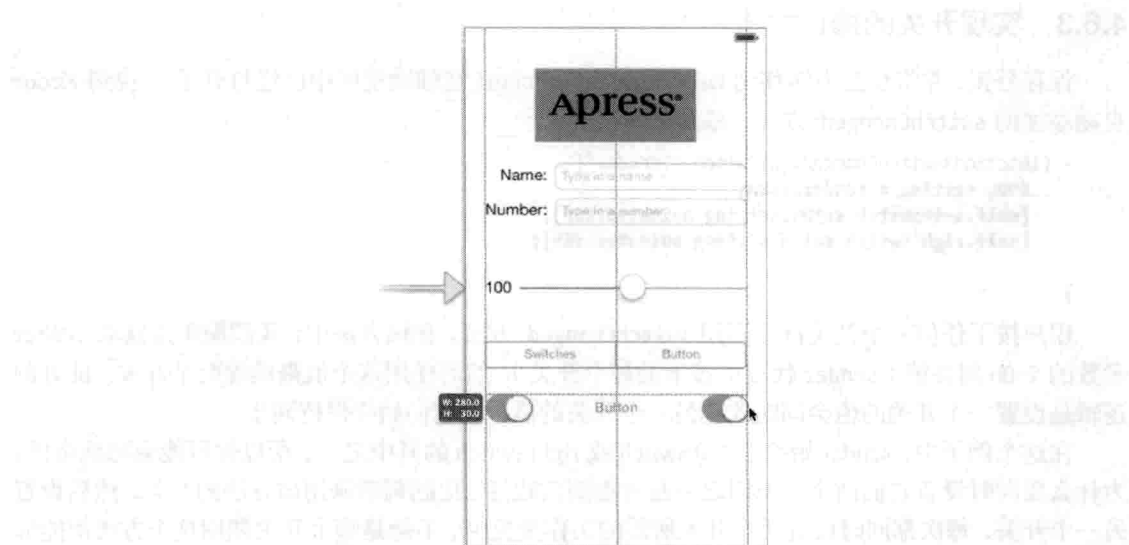


图 4-26 按钮调整好位置和尺寸后，将填满两个开关所占的空间

双击新添加的按钮并输入 Do Something 作为按钮标题。



## 4.7 美化按钮

将你的应用程序与图 4-2 作比较,可能会注意到一个有趣的差异。Do Something 按钮的外观和图中的不太一样。这是因为 iOS 7 中默认的按钮的外观非常简单,只是一段纯文本,没有外形、边框、背景颜色以及其他修饰。这点符合苹果在 iOS 7 中的全新设计理念。不过在有些情况下你仍然需要使用自定义的按钮,因此我们会向你展示如何去做到。

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

需要记住,应用程序是在沙盒中运行的。你不能访问 iOS 设备上其他应用程序使用的模板图像,也不能访问 iOS 本身使用的图像,因此,需要确保所有需要的图像都位于应用程序包中。那么,从哪里获取这些图像模板呢?

幸好,苹果公司提供了一个大礼包。可以从名为 UICatalog 的 iOS 示例应用程序中获取它们,可以在 iOS 开发者库里下载:

<http://developer.apple.com/library/ios/#samplecode/UICatalog/index.html>

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中特别说明允许开发者使用和发布它们。

所以,从 04 - Control Fun 文件夹或者 UICatalog 项目的 images 子文件夹中,找到 blueButton.png 和 whiteButton.png 两张图。在 Xcode 中,选择 Images.xcassets (即我们之前添加图标的图片时使用的资源目录),只需将两张图片从 Finder 文件夹拖动到 Xcode 窗口的编辑区域中。图片就会添加到你的项目中,并可以在应用中使用。

### 4.7.1 可拉伸图像

如果你查看了刚刚添加的两张按钮图片大小,你可能会惊讶于它们的尺寸大小。它们非常小,而且看起来根本没法填满之前添加到分镜的按钮。不过这些图像是可以用来拉伸的。UIKit 可以拉伸图像至任何你需要的尺寸大小。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,它知道如何智能地调整自身大小以维持恰当的外观。对于这些按钮模板,我们不希望图像边缘跟其他部分一样被均匀拉伸。**边缘图像** (Edge inset) 是一个图像的一部分,以像素为单位,它不会被改变大小。我们希望边缘圆角能够保持原样,不随按钮尺寸的改变而改变,因此我们需要指定所有边缘不会被拉伸的区域范围。

过去只能在代码中对它进行设置。你必须使用某图形工具来测量图像边界的像素值,然后在代码中利用这些数字来设置边缘图像。Xcode 5 帮你节省了这样的操作,你能够可视地“切分”资源目录中的任意已有的图像。接下来我们就会开始进行讲解。

在 Xcode 中选中 Images.xcassets 资源目录,并在里面选择 whiteButton 图像。在编辑区域的底部你会看到一个标题是 Show Slicing 的按钮。点击它就会启用切分操作,一开始会在你的图片上面出现 Start Slicing 按钮,点击它就会产生效果。你将会看到三个新按钮并让你选择是否让图

片可以垂直拉伸、水平拉伸或两者都要。请选择中间的按钮使两个方向都可以拉伸。Xcode 会快速分析你的图像，并找到边缘上像素不连续的区域，而位于中间的垂直和水平方向切图则可以不断延伸。你会看到虚线所代表的边界，如图 4-27 所示。如果你的图像很特别，则需要手动调整（这很简单，只需要用鼠标拖动），不过对于这张图，自动分析出的边缘图像已经足够了。

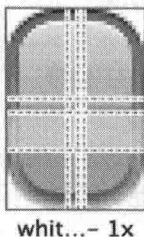


图 4-27 此图就是 white button 图像的默认切分

接下来选择 blueButton 图像并同样为它执行自动切分操作。完成之后就可以使用这些图像了。

回到之前的分镜界面并点中 Do Something 按钮。因为我们将其标记为隐藏的，所以按钮现在是不可见的，不过你还是有办法看到虚影的，你也可以在左边的层级视图（如果你已经展开了）中点选按钮。

选中按钮，按下 option+command+4 来打开属性检查器。在属性检查器中，找到下面的 View 标题部分并取消勾选 Hidden 复选框，这样我们才能看到所做的变化。然后回到顶部位置并将第一个弹出菜单的值由 System 改为 Custom。检查器能让你为按钮指定 Image 和 Background 内容。我们将使用 Background 来展示那张可以改变尺寸的图像，因此点击 Background 下拉列表并选择 whiteButton。你会看到按钮现在显示的是白色的图像，并且是完美地拉伸至覆盖整个按钮的轮廓。做得很好！

现在我们想要使用蓝色按钮来定义这个按钮在高亮时的外观（也就是按钮被按下时看到的）。我们会在本章的下一小节讨论更多关于控件状态的内容。不过目前只是稍微看看顶端第二个弹出菜单（标题是 State Config）。一个 UIButton 按钮可以有多个状态，每个都有自己的文字与图像。目前我们只设定了默认状态，所以把弹出菜单切换到 Highlighted，这样我们就可以设定这个状态了。你会看到 Background 下拉框已经被清空了。点击并选择 blueButton 就可以了。

新的按钮外观有一个问题：默认的 UIButton 按钮尺寸没有足够的高度来显示渐变按钮图像的全部内容。向下拖动按钮底边的缩放手柄来增加它的高度。接近合适的高度时它会自动贴到那个位置，这样就会比一开始的长度多一些像素。

通过设定这个按钮引入了两个新的概念：可拉伸图像（stretchable image）和控件状态（control state）。我们已经讨论过前面那个了，现在来谈谈后面那个。

## 4.7.2 控件状态

每个 iOS 控件都有如下 4 种状态，任何时候都处于并仅能处于其中的一种状态。

- ❑ 普通 (Normal): 最常见的状态是默认的正常状态。控件在未处于其他状态时都为这种状态。
- ❑ 突出显示 (Highlighted): 突出显示状态是控件正被使用时的状态。对于按钮来说, 这表示用户手指正在按钮上。
- ❑ 禁用 (Disabled): 禁用状态是控件被关闭时的状态。要禁用控件, 可以在界面构建器中取消选中 Enabled 复选框, 或者将控件的 enabled 属性设置为 NO。
- ❑ 选中 (Selected): 只有一部分控件支持选中状态。它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 iOS 控件的属性可以根据控件的不同状态接受不同的值。举例来说, 可以为 UIControlStateNormal 状态指定一个图像, 为 UIControlStateHighlighted 状态指定另一个图像, 这样一来, 当用户将手指放在按钮上时, iOS 会使用后一个图像, 而其他情况则使用前者。本质上也就是我们之前在分镜中为按钮设定的两个不同背景状态。

### 4.7.3 为按钮创建并关联输出接口和操作方法

按住 control 键, 从新创建的按钮拖向辅助编辑器, 拖至文件顶端部分中最后一个输出接口下方位置处。弹出窗口出现后, 创建一个新的输出接口, 将其命名为 doSomethingButton。完成之后, 再次按住 control, 把这个按钮拖至文件底部的 @end 声明上方。在这里创建一个操作方法, 名为 buttonPressed:。不需要特别设置 Type, 因为这个方法不会用到这个特性。

如果现在保存更改, 并且测试该应用程序, 你将发现分段控件是可用的, 但它并没有做什么实质性工作。我们需要为它添加一些逻辑, 以实现控制按钮和开关的隐藏和显示。

还需要在应用启动时就把按钮标记为隐藏。之前没有这么做, 因为这样会使关联输出接口和操作方法变得更加困难。而现在, 既然已经关联好了, 就可以隐藏这个按钮了。用户按下分段控件右边部分时, 按钮将会显示。而应用刚启动时, 按钮是隐藏的。按下 option+command+4 打开属性检查器, 向下滚动到 View 的部分, 选中 Hidden 复选框。这样, 按钮在界面构建器中仍然可见, 但是看上去是淡出并且透明的, 这表示它处于隐藏状态。

## 4.8 实现分段控件的操作方法

保存分镜, 并将注意力重新放回到 BIDViewController.m, 找到 Xcode 为我们创建的 toggleControls: 方法, 添加以下粗体显示的代码:

```
- (IBAction)toggleControls:(UISegmentedControl *)sender {
    // 0 == switches index
    if (sender.selectedSegmentIndex == 0) {
        self.leftSwitch.hidden = NO;
        self.rightSwitch.hidden = NO;
        self.doSomethingButton.hidden = YES;
    }
    else {
        self.leftSwitch.hidden = YES;
        self.rightSwitch.hidden = YES;
    }
}
```

```

        self.doSomethingButton.hidden = NO;
    }
}

```

这段代码检查 sender 的 `selectedSegmentIndex` 属性，这样就可以知道当前选中的是分段控件的哪一部分。第一部分（名为 `switches`）的索引值是 0（已经在注释中注明了，这样以后重读这段代码时，就能知道它的作用）。根据当前选中的分段来隐藏或者显示合适的控件。

现在，保存所做的修改，并尝试在 iOS 模拟器中运行该应用。如果你的代码输入没有错误，应该就能使用分段控件在按钮和一对开关之间进行切换了。点击任何一个开关，另一个开关的值也会随之改变。而这个按钮，仍然什么也没有做。在实现这个按钮的功能之前，需要先了解一下如何操作表单和警告视图。

## 4.9 实现操作表单和警告视图

操作表单（action sheet）和警告视图（alert）都用于向用户提供反馈。

- ❑ 操作表单的作用是要求用户在两个以上选项之间作出选择。操作表单从屏幕底部出现，显示一系列按钮供用户选择（参见图 4-3）。用户必须点击其中一个按钮之后才能继续使用应用程序。操作表单通常用于向用户确认有潜在危险的或者无法撤销的操作，比如删除对象。
- ❑ 警告视图以圆角矩形的形式出现在屏幕中央（参见图 4-4）。与操作表单类似，警告视图也要求用户必须做出一个回应，然后才能继续使用应用程序。警告视图通常用于通知用户发生了一些重要的或者不寻常的事情。与操作表单不同，警告视图可以只显示一个按钮，但是如果需要接收多个回应的话，也允许显示多个按钮。

---

**注意** 要求用户必须先作出选择，然后才能继续使用应用的视图称为模态视图（modal view）。

---

### 4.9.1 遵从操作表单委托方法

还记得第 3 章讨论的应用程序委托吗？`UIApplication` 并不是 Cocoa Touch 中唯一使用委托的类。实际上，委托模式是 Cocoa Touch 中一个非常通用的设计模式。操作表单和警告视图都使用了委托，所以它们知道在用户做出选择之后应该通知哪个对象。在当前的应用程序中，需要在操作表单消失时获得通知。但是并不需知道警告视图何时消失，因为这里只是使用它来通知用户某件事情，而不是要求用户作出选择。

为了让控制器类充当操作表单的委托对象，控制器类需要遵从 `UIActionSheetDelegate` 协议。这可以通过在类声明部分的超类后面的尖括号中添加协议名称来实现。在 `BIDViewController.h` 中添加以下协议声明代码：

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

```
@interface BIDViewController : UIViewController <UIActionSheetDelegate>
```

```
...
```

### 4.9.2 显示操作表单

现在切换回 BIDViewController.m, 准备实现按钮的操作方法。除了现有的操作方法, 实际上还需要实现另一个方法, 即 `UIActionSheetDelegate` 方法, 操作表单消失时会调用这个方法通知我们。

首先, 找到 Xcode 创建的空的 `buttonPressed:` 方法, 向该方法添加以下粗体显示的代码, 创建并显示操作表单:

```
- (IBAction)buttonPressed:(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];
}
```

接下来, 在 BIDViewControl.m 中的 `buttonPressed:` 方法后面添加一个新方法:

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

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

这段代码究竟做了些什么呢? 首先, 在 `doSomething:` 操作方法中分配了一个 `UIActionSheet` 对象并进行了初始化, 这个对象用于表示操作表单 (也许你自己想不到这一点):

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

这个初始化方法接受多个参数。下面依次介绍这些参数。

第一个参数是要显示的标题。如图 4-3 所示，这里提供的标题将显示在操作表单的顶部。

第二个参数是操作表单的委托对象。按下操作表单上的某个按钮时，委托对象会收到通知。更确切地说，是委托对象的 `actionSheet:didDismissWithButtonIndex:` 方法会被调用。将 `self` 作为委托对象参数传递给这个方法，可以确保调用我们实现的 `actionSheet:didDismissWithButtonIndex:` 方法。

下一个参数是 `cancel` 按钮（取消按钮）的标题，用户可以单击这个按钮以表明不希望继续操作。所有操作表单都应该有一个 `cancel` 按钮，但是可以根据需要为它指定合适的标题。如果不打算提供选择给用户，那就没必要使用操作表单。如果只希望通知用户，而不需要用户作出选择的话，则使用警告视图更合适。

下一个参数是 `destructive` 按钮（可以将它理解为“是的，请继续”按钮），同样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任意数量的其他按钮，这些按钮也会显示在操作表单上。最后这个参数可以接受数量不确定的值，这是 Objective-C 语言中一个非常好的特性。如果希望操作表单上还有另外两个按钮，可以像下面这样编写代码：

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

这段代码会产生一个拥有四个按钮的操作表单。可以在 `otherButtonTitles` 参数中传递任意数量的参数，只要最后一个参数是 `nil` 即可。当然，由于可用屏幕空间的限制，按钮的数量实际上是有限制的。

创建操作表单之后，使用以下代码显示它。

```
[actionSheet showInView:self.view];
```

操作表单始终有一个父视图，即当前对用户可见的视图。在本例中，我们希望使用在界面构建器中设计的视图作为父视图，因此使用 `self.view`。注意 Objective-C 点表示法的使用。`self.view` 使用存取方法返回视图属性的值，等价于 `[self view]`。

为什么不简单地使用 `view` 而是使用 `self.view` 呢？因为 `view` 是 `UIViewController` 类的一个私有实例变量，即便对子类来说也是不可见的，所以不能直接访问，必须通过存取方法来访问。

很好，这并不难，是吧？在寥寥数行代码中，我们显示了一个操作表单，并且要求用户作出决定。iOS 甚至为显示这个操作表单创建了动画效果，而这并不需要我们做任何额外的工作。现在，只需要确定用户按下的是哪一个按钮。刚才实现的另一个方法 `actionSheet:didDismissWithButtonIndex:`，是 `UIAlertSheetDelegate` 协议中的一个方法，由于已经指定了 `self` 作为操作表单的委托，所以这个方法将会在用户按下按钮时自动被操作表单调用。

使用 `buttonIndex` 参数可以知道用户实际按下的是哪个按钮。但是，如何知道哪个按钮索引代表 `cancel` 按钮，哪个按钮索引代表 `destructive` 按钮呢？幸好，该委托方法接受一个指向



UIAlertActionSheet 对象（它代表操作表单）的指针，而该操作表单对象知道哪个按钮是 cancel 按钮。我们只需要查看它的 cancelButtonTitle 属性：

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

这行代码确保用户按下的不是 cancel 按钮。由于只给用户提供了两个选项，因此可以知道，如果用户没有按下 cancel 按钮，那么一定是按下了 destructive 按钮，也就是表示继续操作。知道用户并没有取消操作之后，首先做的是创建一个将要显示给用户的新字符串。在一个实际的应用程序中，会在这里处理用户的请求。本例假设已经执行了一些操作，然后使用警告视图来通知用户。

如果用户在顶部的文本框中键入了姓名，就获取这个值，并且在警告视图的消息中使用它；否则，就只显示一条普通的消息：

```
NSString *msg = nil;
```

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

接下来的代码你应该不陌生。警告视图和操作表单的创建方式非常类似：

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

同样，我们传递了一个需要显示的标题。还有一条详细消息，也就是刚才创建的字符串。警告视图也使用委托，如果需要知道用户何时关闭了警告视图，或者用户按下了哪个按钮，可以将 self 指定为委托对象（正如之前在操作表单中所做的那样）。如果要这么做，那就需要让我们的类遵循 UIAlertViewDelegate 协议，并且实现该协议中的一个或多个方法。本例只向用户通知一些消息，而且只提供一个按钮。并不关心用户何时按下按钮，而且也已经知道哪个按钮将被按下（因为只有一个按钮），所以这里把 delegate 参数指定为 nil，表示当用户结束对警告视图的操作后，我们不需要做任何处理。

与操作表单不同，警告视图并没有与特定的视图绑定在一起，所以只需要把警告视图显示出来，而不用指定父视图。现在所有工作都完成了。保存文件，然后构建、运行并且测试一下完整的应用程序吧。

### 4.9.3 最终调整

在 iOS 7 中，苹果加入了新的 GUI 规范。其中屏幕顶端的状态栏在 iOS 7 应用中是透明显示



的，因此你的内容会穿透它。现在 Apress 图像的黄色在应用程序的白色背景前过于显眼，所以我们要让黄色覆盖整个视图。在 Main.storyboard 中，选择主内容视图，按下 option+command+4 打开属性检查器。点击标题为 Background 的颜色按钮打开 OS X 系统标准的颜色选取器。这个颜色选取器的一个特点就是你还可以选取屏幕上可见的任意颜色。只需要点击左上角的放大镜图标，然后点击 Apress 图像上任意黄色的区域就可以用它来设置整个视图的背景颜色。完成之后请关闭颜色选取器。

你会发现屏幕上的背景和 Apress 图像的颜色看起来有细微的差别，但在模拟器或真机中运行应用时就会完全一样。在界面构建器中的颜色有差别是因为 OS X 会根据你使用的显示器自动调整颜色进行适配。而在 iOS 设备或模拟器中则不会发生。

现在运行你的应用，将会看到黄色铺满了整个屏幕，状态栏和应用内容之间也没有明显的区别。如果你没有使用全屏滚动的内容或其他需要在顶端使用导航栏或其他控件的内容，这将会是一个极好的方式来展示全屏内容并不会受到状态栏多少影响。

## 4.10 小结

本章内容比较多。我们并没有讲述太多新概念，而是着重介绍了许多控件的用法，以及各种实现的细节。你做了不少输出接口和操作方法的实践，并了解了如何利用视图的层次性提供方便，而且还初步了解了一些约束。你学习了控件状态和可拉伸图像，以及如何使用操作表单和警告视图。

这个小应用包含许多知识点，回头你好好把玩一下。可以尝试更改各项属性的值，或者尝试添加和修改代码，看界面构建器中的设置会有哪些不同效果。我们无法逐一介绍 iOS 中的所有控件，但本章中的应用是一个非常好的开端，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切换 iOS 设备时会发生什么。你可能已经知道许多应用会根据用户握持设备的方式来更改其显示风格，接下来就会介绍如何在你自己的应用中实现这样的功能。

应该说, iPhone、iPad 是工程设计的杰作。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, 支持在旋转设备时更改应用程序的方向。这种行为称为自动旋转(autorotation), iOS 中的 Web 浏览器, 移动版 Safari, 就是一个典型的例子(参见图 5-1)。

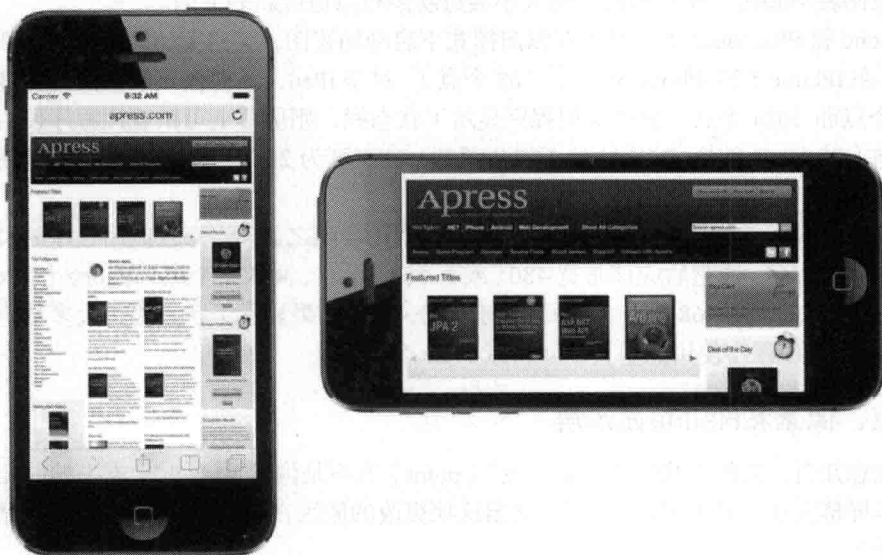


图 5-1 与许多 iOS 应用程序一样, Mobile Safari 能够根据握持设备的方式改变显示方式, 以充分利用可用的屏幕空间

本章将详细介绍自动旋转。首先来看看它的复杂细节, 然后讨论在应用中实现自动旋转的几种不同方法。

## 5.1 自动旋转机制

自动旋转并不是对所有应用程序都适用。苹果公司的某些 iPhone 应用程序仅支持单方向模

式。例如，通讯录（Contact）只能在纵向模式下编辑。但 iPad 应用程序并非如此，苹果公司建议 iPad 上的大部分应用程序都应该支持所有的方向（某些沉浸式应用当属例外，比如面向特殊布局设计的游戏）。

实际上苹果公司自己几乎全部的 iPad 应用程序在两个方向上都能很好地运行。其中许多应用程序使用不同的方向来显示不同的数据视图。例如，邮件（Mail）和备忘录（Notes）应用使用横向模式在左侧显示一个项目列表（文件夹、消息或备忘录），在右侧显示所选的项目；而应用的纵向模式使你能够将注意力集中在所选项目的细节上。

对于 iPhone 应用，基本原则是，如果自动旋转能够增强用户体验，那么应该将它添加到应用中。而对于 iPad 应用，应该添加自动旋转功能，除非有充分的理由不这么做。幸好，苹果公司在 iOS 和 UIKit 中非常好地隐藏了自动旋转的复杂细节，因此开发者在自己的 iOS 应用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，如果用户旋转设备，活动视图控制器将被问及是否可以旋转到新的方向（本章稍后将介绍如何实现）。如果视图控制器给予肯定的响应，那么应用程序的窗口和视图就会旋转，窗口和视图的大小会重新调整以适应新的方向。

在 iPhone 和 iPod touch 上，对于在纵向模式下启动的视图，其宽度和高度分别为 320 个点和 480 个点（在 iPhone 5 和 iPhone 5S 上是 568 个点）。对于 iPad，纵向模式下视图的宽度和高度分别为 768 个点和 1024 个点。如果应用程序显示了状态栏，则屏幕上可供应用程序使用的空间将在垂直方向上减少 20 个点。状态栏位于屏幕顶部，其高度为 20 个点（参见图 5-1），用于显示信号强度、时间以及电池电量等信息。

将设备切换到横向模式时，应用程序的窗口和视图会随之旋转，而且应用程序的尺寸会重新调整以适应新的方向，调整后应该是宽 480（或者 568）个点、高 320 个点（iPhone 和 iPod touch），或者是宽 1024 个点、高 768 个点（iPad）。与之前一样，如果显示了状态栏（大多数应用都会显示），实际可供应用程序使用的垂直空间将减少 20 个点。

### 5.1.1 点、像素和 Retina 显示屏

你可能想知道，为什么我们说的是“点”（point）而不是像素（pixel）。本书的前几版一直用像素来表示屏幕大小，并不是用“点”。做出这项更改的原因在于苹果引入了 Retina 显示屏（视网膜显示屏）。

Retina 显示屏是苹果公司的销售术语，指的是 iPhone 4、iPhone 4S、iPhone 5 以及后续的 iPod touch 和 Retina 显示屏的 iPad 使用的高分辨率屏幕。Retina 显示屏将 iPhone 屏幕的分辨率将原来的 320 像素×480 像素翻了一倍，达到了 640 像素×960 像素（iPhone 5 已经达到了 640×1136），iPad 屏幕的分辨率从原来的 1024×768 提高到了 2048×1536。

幸好，大多数情况下，你不需要为此做任何工作。我们操作屏幕上的界面元素时，使用点而非像素来指定其尺寸和距离。对于早期的 iPhone（以及第一代 iPad、iPad 2 和第一代 iPad Mini）来说，点和像素是等价的。一个点就是一个像素。然而，在较新的 iPhone 和 iPod touch 上，一个点相当于 4 像素的面积（宽和高都是 2 像素），虽然屏幕宽度实际上是 640 像素，但是依然是 320

个点。同样，iPad 屏幕的几何尺寸是 1024×768 个点，虽然实际上拥有 2048 像素×1536 像素。鉴于 iOS 自动把点映射到了屏幕的物理像素，所以可以把点看做“虚拟分辨率”（virtual resolution）。第 16 章将深入讨论这些内容。

在一般的应用程序中，在屏幕中调整像素的大部分工作都由 iOS 负责。而应用程序的主要工作是，确保所有的界面元素能够很好地适应尺寸调整后的窗口，并且比例合适。

### 5.1.2 自动旋转的实现方式

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合适的约束。调整视图时，约束可以告诉 iOS 设备应该如何对控件进行调整。如果你曾在 OS X 上使用过 Cocoa，那么应该熟悉这个基本过程，因为这与调整窗口尺寸时窗口内控件的行为方式是一样的。这个系统称为 Cocoa Autolayout，但是我们使用“约束”这个术语来描述这些东西。约束是 Cocoa Autolayout 的一部分，多数情况下可以直接与约束进行交互并对它进行配置。

使用约束最简单的方式就是在界面构建器中进行配置。可以在界面构建器中定义约束，这些约束用于描述当父视图发生变化或者其他视图发生位置变更时，GUI 组件应该如何进行位置调整和大小调整。第 4 章已经初步接触过约束了，本章会深入介绍这个主题。可以把约束理解为描述视图几何属性的方程式，而 iOS 视图系统就是求解程序，它在必要时对视图进行调整，使其满足方程式所描述的几何属性。

约束是 iOS 6 中新添加的功能，但是在 Mac 中已经出现一段时间了。在 iOS 和 OS X 中，约束可以用来替代之前的“springs and struts”系统。相比这个旧系统的功能，约束有过之而无不及。

在界面构建器中配置约束快捷方便，但是这种方法并非对所有应用程序都适用。对于较复杂的界面必须采用不同的方式来处理自动旋转。而对于较复杂的视图，你需要重写视图控制器类中继承于 `UIViewController` 的方法。这样做就能按你想要的方式进行布局了。我们会在本章的末尾向你展示如何做到。

开始吧，准备好了吗？在使用不同的方式对 GUI 的排列进行配置之前，先来看一下如何指定应用支持的方向。

## 5.2 选择视图支持的方向

我们将创建一个简单的应用来演示如何为应用选择支持的方向。在 Xcode 中创建一个 Single View Application 项目，将其命名为 Orientations。在 Device 弹出菜单中选择 iPhone，并与其他项目文件夹存储在同一目录下。

在分镜中设计 GUI 之前，要先告诉 iOS 我们的视图支持自动旋转。有两种方式可以做到，既可以创建一个应用级的设置（这会成为所有视图控制器的默认支持方向），也可以为每个独立的视图控制器做进一步调整。接下来就看一下这两种方式，首先看一下应用级的设置。

### 5.2.1 应用级支持的方向

首先,需要指定应用程序所支持的屏幕方向。新的 Xcode 项目窗口出现后,应该已经打开了项目设置。如果尚未打开,点击项目导航面板中的第一行(以项目名称命名的那一项),然后确保位于 General 界面中。在提供的选项中,应该可以看到有一部分的标题为 Deployment Info,其中包含了一组名为 Device Orientation 的复选框(参见图 5-2)。

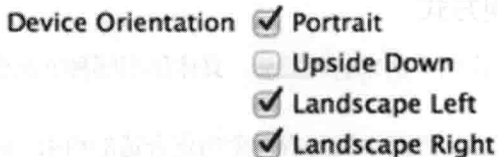


图 5-2 项目的 General 界面中显示了支持的设备方向

这就是指定应用程序所支持方向的方法。这并非意味着应用中的每一个视图都要支持被选中的方向,但如果想让所有的应用程序视图都支持某个方向,就必须在这里选中该方向。

---

**注意** 图 5-2 所示的 4 个复选框实际上只是一种在应用的 Info.plist 文件中添加、删除相关项的便捷方式。如果单击项目导航面板中 Supporting Files 文件夹下的 Orientations-Info.plist 文件,应该能看到一项名为 Supported interface orientations 的属性,其中包含三个子项,对应当前选择的三个方向。在 General 界面中选中或是取消选中那些复选框,就是在这个数组中添加、删除对应的项。使用这些复选框更便捷,而且不容易犯错。所以强烈建议使用复选框,当然你应该知道它们的作用。

---

你是否注意到,默认情况下 Upside Down 方向(Home 键位于上方的那种方向)是不被选中的。这是因为,如果在 iPhone 倒置时来电,那么在接听电话时,手机仍然可能处于倒置状态。而 iPad 应用项目默认支持所有方向,因为在任何方向上都应该能够使用 iPad。由于这个项目是针对 iPhone 的,所以可以保留这些复选框的默认设置。

现在,选中 Main.storyboard,在对象库中拖出一个标签,将其放置到视图中心偏上一些的位置,如图 5-3 所示。选中标签的文本,将其更改为 This Way Up。更改文本内容可能会移动标签的位置,所以要再次把它拖到水平中心位置。

现在,按下 Command+R 构建并运行这个简单的应用。当应用出现在模拟器中之后,试着使用 command+左方向键或者 command+右方向键来旋转设备。可以看到整个视图(包括刚刚添加的标签)会自动旋转到每一个方向(除了 Upside Down),我们刚刚就是这样配置的。

已经确定了应用将支持的方向,但是所要做的并不止这些。还可以为每一个视图控制器指定它自己支持的方向,这样就可以更具体地控制应用的不同部分在不同方向上的表现。

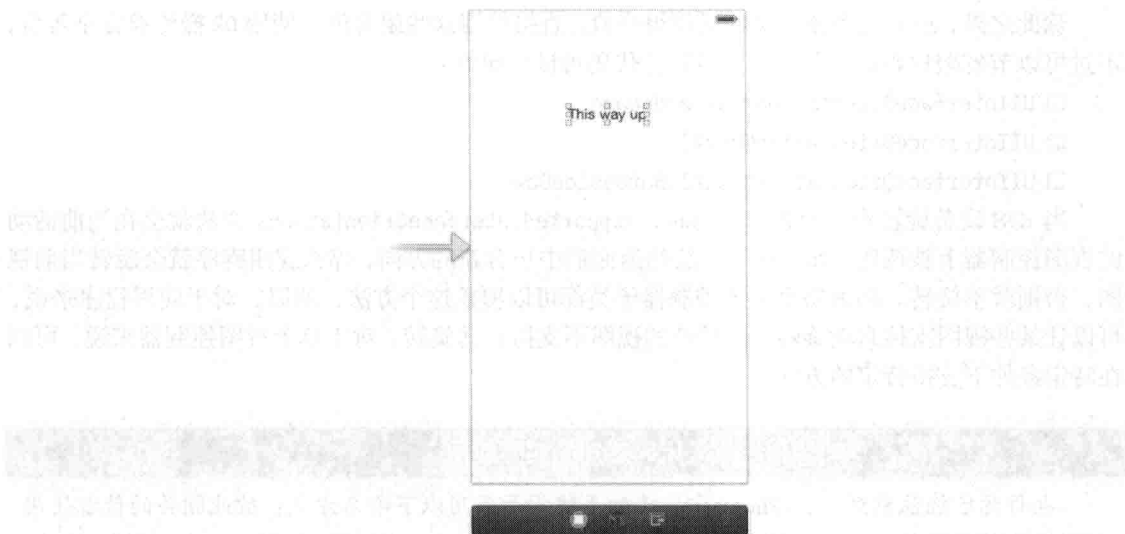


图 5-3 标签位置发生中心偏移时会提示你

### 5.2.2 单个控制器的旋转支持

现在对视图控制器进行配置，使它支持不同的方向（应用级支持方向的子集）。注意到我们在应用的全局配置中指定应用所支持方向的绝对上限。如果全局配置不支持 Upside-Down 方向，那么任何一个视图控制器都不能迫使系统旋转到 Upside-Down 方向。在视图控制器能做的就是在可接受的范围内做进一步限制。

单击 BIDViewController.m。这里要实现一个在父类 UIViewController 中定义的方法，这个方法可以指定当前视图控制器所支持的方向。

```
- (NSUInteger)supportedInterfaceOrientations {
    return (UIInterfaceOrientationMaskPortrait |
           UIInterfaceOrientationMaskLandscapeLeft);
}
```

这个方法可以使用 C 语言风格的掩码返回所支持的方向。这就是 iOS 询问一个视图控制器是否支持旋转到某个指定方向的方法。在这个例子中，返回值表示可以接受两种方向：默认的纵向方向和顺时针旋转 90 度之后的横向（也就是手机左侧边缘位于上方时的方向）。我们使用 OR 操作符（外观为一个竖杠）把这两个方向掩码组合在一起作为返回值。

UIApplication.h 定义了如下的方向掩码，可以使用之前提到的 OR 操作符任意组合这些掩码：

- UIInterfaceOrientationMaskPortrait
- UIInterfaceOrientationMaskLandscapeLeft
- UIInterfaceOrientationMaskLandscapeRight
- UIInterfaceOrientationMaskPortraitUpsideDown

除此之外，还有几个预定义的通用组合值。直接使用这些组合值与使用 OR 操作符完全等价，不过可以节省敲代码的时间，而且可以让代码可读性更好：

- ❑ `UIInterfaceOrientationMaskLandscape`
- ❑ `UIInterfaceOrientationMaskAll`
- ❑ `UIInterfaceOrientationMaskAllButUpsideDown`

当 iOS 设备旋转到一个新的方向时，`supportedInterfaceOrientations` 方法就会在当前活动的视图控制器上被调用。如果这个方法的返回值中包含新的方向，那么应用程序就会旋转当前视图，否则就不旋转。因为每个视图控制器子类都可以覆盖这个方法，所以，对于应用程序来说，可以让某些视图支持自动旋转，而另外的视图不支持自动旋转；对于单个视图控制器来说，可以在特定条件下支持特定的方向。

### 代码感知的实际应用

也许你已经注意到了，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序列作为开头。`UIInterfaceOrientationMaskPortrait`、`UIInterfaceOrientationMaskPortraitUpsideDown`、`UIInterfaceOrientationMaskLandscapeLeft` 和 `UIInterfaceOrientationMaskLandscapeRight` 都以 `UIInterfaceOrientationMask` 作为开头，是为了充分利用 Xcode 的代码自动补全（Code Completion）特性。

你可能已经注意到，在 Xcode 中输入代码时，Xcode 经常会尝试自动完成正在输入的单词。这就是代码自动补全特性的一种实际应用。

开发者不可能记住系统中定义的所有常量，但可以记住常用常量集合的通用开头。需要指定方向时，只需输入 `UIInterfaceOrientationMask`（或者只是 `UIInterf`），就会看到弹出了所有匹配项的列表（可以在 Xcode 的偏好设置中设置只有按下 `esc` 键才弹出）。可以使用方向键在匹配列表中进行移动，按下 `tab` 或 `return` 键进行选择。这比在文档或头文件中查找值要快得多。

可以随意修改这个方法的返回值来看看不同方向的效果。可以要求系统在应用支持的某个方向上压缩视图的显示空间，但是不要忘记之前说过的全局配置！请记住，如果没有在全局配置中提供对 `Upside-Down` 的支持，那么任何视图都不会支持 `Upside-Down` 方向，不管视图的 `supportedInterfaceOrientations` 方法的返回值是什么。

**注意** iOS 实际上有两种不同类型的方向。这里讨论的是界面方向（interface orientation）。另一个独立但相关的概念是设备方向（device orientation）。设备方向表示设备当前的持握方式，而界面方向则是指屏幕上视图的旋转方向。如果把一个 iPhone 上下颠倒过来，那么设备的方向就是向下倒置的，但是界面方向却只能是其他的三个方向之一，因为 iPhone 应用默认不支持 `Upside-Down` 方向。



## 5.3 使用约束设计界面

在 Xcode 中,新建一个基于 Single View Application 模板的项目,命名为 Autosize。选中 Main.storyboard 文件,在界面构建器中编辑界面。使用约束的好处之一就是,只需要编写少量的代码就可以完成大量的工作。虽然仍需要在代码中指定支持的方向,但是其余的自动旋转相关实现可以在界面构建器中完成。

从对象库中拖出 4 个标签放置到视图上,布局可以参考图 5-4。利用蓝色虚线将每个标签靠近各自所在的角落对齐。在这个例子中,使用 UILabel 类的实例来展示如何对 GUI 布局使用约束,但同样的规则适用于所有 GUI 对象。

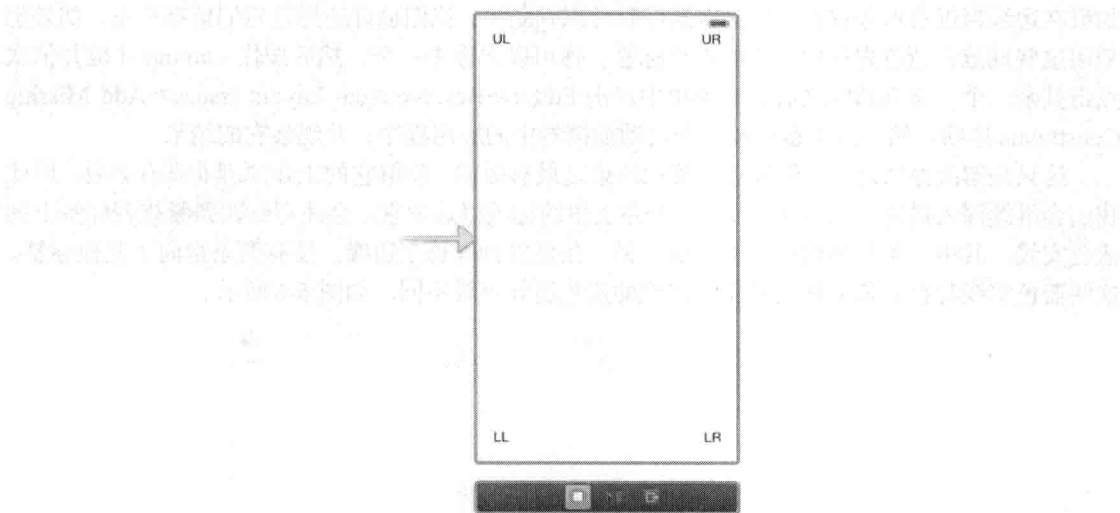


图 5-4 向界面中添加四个标签

双击每一个标签,为各标签指定一个标题以方便区分。使用 UL 作为左上角标签的标题,使用 UR 作为右上角标签的标题,使用 LL 做为左下角标签的标题,使用 LR 作为右下角标签的标题。设置完所有的标签文本后,把它们全部再拖回到与容器视图各个角落对齐的位置。

现在已经指定了支持的方向但是还没设置自动旋转相关特性,看看会发生什么。构建并运行应用。iOS 模拟器出现之后,选择 Hardware>Rotate Left, 这样会模拟出 iPhone 旋转到横向模式的状态,如图 5-5 所示。

可以看到,所有东西都不太对劲。左边的标签旋转后还在正确的位置,但右上角的标签却向中央偏移了,而且底部的标签则完全不见了!所有的对象都维持了与视图左上角相对应的距离,为什么会这样?

我们真正想要的是每个标签旋转后都紧紧地固定在最近的角落。右边的标签应当向右漂移以适配视图的新宽度,而底部的标签应当向上移以适配新的高度,不然会看不见。还好我们可以在界面构建器中简单地设置这些约束来实现这些变化。

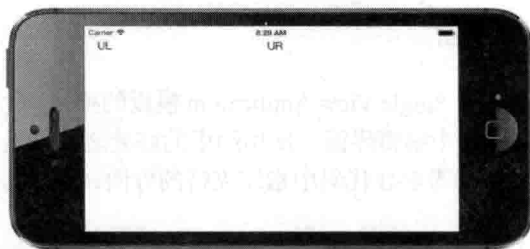


图 5-5 到目前为止并不正常。为什么会是这种效果

事实上界面构建器已经可以智能地检测这些对象并创建一系列默认约束来达到我们的要求。如果在边缘附近有对象存在，它会根据经验常识判断出，我们也许还想让它们留在那里。如果想采用这些规范，请首先选中所有的四个标签。你可以先选中一个，然后按住 **command** 键并依次点击其余三个。全部选中之后，在菜单中点击 **Editor>Resolve Auto Layout Issues>Add Missing Constraints** 选项。然后点击运行按钮来启动模拟器中的应用程序，并观察它的结果。

这只是解决办法之一，但像这样使用约束是最有效的，理解它的工作原理非常有必要。因此我们会继续深入讲解。回到 Xcode，点击左上角的标签以选中它。会注意到四条紧挨着标签上的蓝色实线，其中一条延伸到屏幕左边缘，另一条延伸到屏幕上边缘。还有两条指向了其他标签。这些蓝色实线与在屏幕上拖曳对象时出现的蓝色引导虚线不同，如图 5-6 所示。

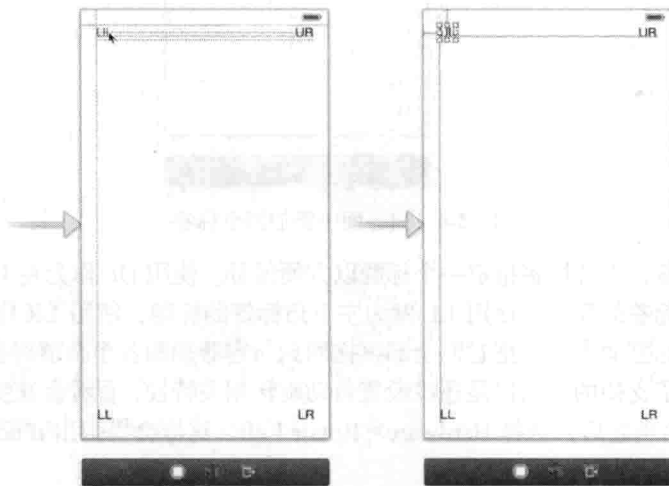


图 5-6 左图显示的是拖曳对象时出现的蓝色引导虚线，用于对齐。  
右图的蓝色实线显示为所选对象配置的约束

每条蓝色实线代表一个约束。如果按下 **option+command+5** 来打开尺寸检查器 (size inspector)，可以看到里面有包含了四个约束的列表，其中的两个负责处理当前标签位置与父视图 (即容器视图) 之间的关系：一个用于控制当前视图顶部与其父视图顶部之间的距离，另一个用

于控制前置空白 (leading space), 也就是距左边的间距。父视图的尺寸发生变化时, 这些约束会使标签与父视图顶部和左边的间距不变。另外两个约束指向了其他标签并与它们保持对齐。

注意, 某些语言的阅读顺序是从右向左, 对于这些语言, “前置空白” 指的是右边的间距, 所以, 如果用户为 iPhone 选择的语言是阿拉伯语, 前置空白约束可能会使这个 GUI 元素被放置到相反方向。目前来说, 把 “前置空白” 理解为 “左边间距” 就可以了。

现在, 选中放置在视图右上角的标签, 可以看到有一点儿不同。这个标签上的蓝色实线延伸到父视图的右侧边缘, 另外两条则延伸至其他标签, 再看一下尺寸检查器中的 Constraints 部分, 这里显示的约束用于控制底线对齐 (影响垂直位置) 和后置空白 (trailing space, 这里的意思是 “右边间距”, 上文提到过的相关内容同样也适用于这里)。当父视图尺寸发生变化时, 这些约束能使标签始终依附在父视图的右上角。之后检查一下其余标签都包含了哪些约束。

### 5.3.1 覆盖默认的约束

再从对象库中拖出一个标签放置到视图上。这一次, 不要再把标签拖到角落, 而是拖到视图的左侧边缘, 与其他标签的左侧边缘对齐, 并且在视图的中心位置垂直对齐。可以使用出现的蓝色虚线作为辅助。图 5-7 展示了效果。

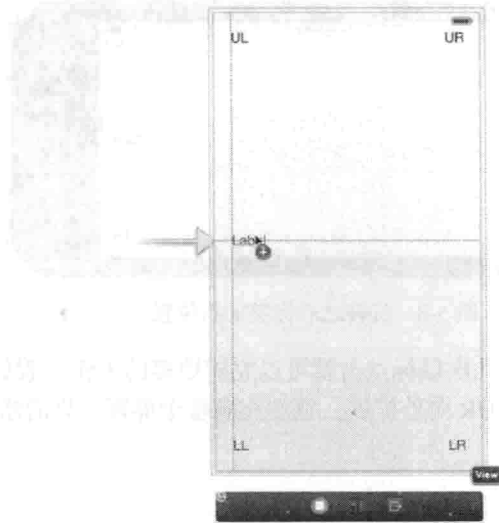


图 5-7 放置左边的标签

左边的标签放置好完成之后, 将其标题改为 Left。按下 `command+R` 在模拟器中运行这个应用, 旋转到横向模式, 可以看到标签与父视图的顶部间距保持不变 (问题跟左上角的标签一样), 标签这时位于屏幕中间偏下的位置, 又来了!

需要增加一个新的约束保证应用的正常运行, 回到 Xcode 并且在分镜中选中左边的标签。添加一个约束保证标签在垂直方向上始终位于屏幕中间, 这非常容易, 只需选择 `Editor>Align>`

Vertical Center in Container 就可以了。完成之后，Xcode 会创建一个新的约束并且会在编辑器视图中选中这个新的约束。你可能会有点儿迷惑，不过不用担心。点击标签再次将其选中。按下 `option+command+5` 调出尺寸检查器，可以看到这个标签有一个与父视图中心 Y 值对齐的约束。按下 `Command+R` 再次运行这个应用，进行旋转，可以看到现在所有的标签都会很完美地移动到合适的位置。非常好！

现在，再拖出一个新的标签，放置到视图右侧，与其他位于右侧的标签右边对齐，并与 Left 标签垂直对齐。这样标签大家族就全员到齐了。把新标签的标题改为 Right，然后将标签再稍微移动一点，使用蓝色引导虚线确保右边与其他两个右侧的标签重新对齐。我们可以使用 Xcode 所提供的自动约束功能，请选择 `Editor>Resolve Auto Layout Issues>Add Missing Constraints` 菜单项来生成它们。

选中这个标签，看一下 Xcode 创建的约束。应该可以看到有两个约束，其中一个将标签与在上面的 UR 标签绑定到一起，另一个的底线（假设标签中的文本下面有一条底线）与 Left 标签对齐。Xcode 确实算得非常准！再次构建并运行，然后旋转屏幕，可以看到屏幕上所有标签的相对位置都非常正确（参见图 5-8）。如果再旋转回来，它们又都回到原来的位置。这个技术对于大多数的应用程序都有用处。



图 5-8 旋转之后标签的新位置

这样已经可以了，但是只需几次鼠标点击就可以完成更多的工作。假如我们现在突发奇想，决定让最顶部的两个标签 UL 和 UR 向外扩展，宽度充满整个屏幕。只需略微调整尺寸大小并增加几个约束，就完成了。

### 5.3.2 与屏幕等宽的标签

接下来要创建一些约束，确保设备旋转后视图顶部两个标签的宽度始终相同、间距不变。完成之后的效果如图 5-9 所示。

这里最难的地方在于要用眼睛来确认是否达到了我们想要的结果，每个标签都要在屏幕各自一半的区域里完全位于中间。为了方便看到结果是否正确，我们暂时设置一下标签的背景颜色。同时选中 UL 和 UR 标签，打开属性检查器，找到下面的 View 部分。使用 Background 控件选择一个鲜艳明亮的颜色。你将看到整个标签都会填满这个颜色。

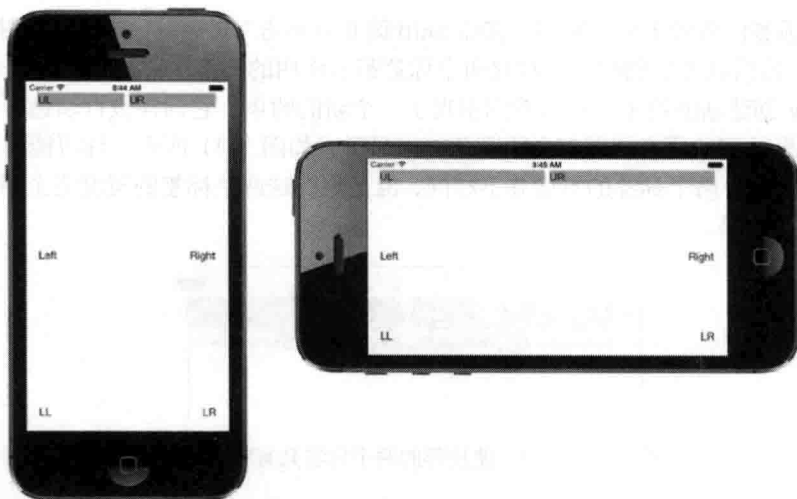


图 5-9 无论在纵向模式还是横向模式下，顶部两个标签都与显示屏等宽

现在请注意力回到 UL 标签并拖动其右侧边缘上的尺寸调整手柄，一直拉伸到接近视图水平方向中心位置。不需要准确地拉伸到中心位置，至于原因嘛，你很快就会明白了。完成之后，再调整 UR 标签的尺寸，将其左侧边缘上的尺寸调整控件向左拉伸，一直到蓝色引导虚线出现为止（也就是说已经离左边的标签很近了，蓝色引导虚线所在的位置就是建议的间距）。现在，我们要添加约束，使这些标签能够占满父视图的宽度。同时选中 UL 和 UR 标签，并在菜单中选择 Editor>Pin>Horizontal Spacing 选项。这个约束会告诉布局系统让这些标签之间始终保持与当前一样的水平间距。构建并运行应用程序，看看会发生什么。旋转设备，很可能会看到图 5-10 所示的情况。

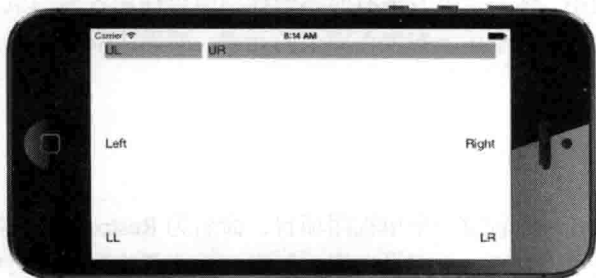


图 5-10 标签被拉伸到占满整个显示屏的宽度，但是并不均匀

已经很接近了，但是还没有达到我们想要的效果。哪里做错了呢？我们定义了控制每个标签与父视图之间的相对位置以及标签之间的位置，但还完全没有设定标签的尺寸。这让布局系统会以它自己的方式自由地改变尺寸（如我们之前所看到的，肯定不能这样）。为了解决这个问题，我们还需要添加一个约束。

确保 UL 标签仍然处于选中状态, 按住 Shift 键并且单击 UR 标签, 这样就同时选中了 UL 和 UR 两个标签, 然后就可以创建一个对这两个标签都起作用的约束。从菜单中选择 **Editor>Pin>Widths Equally** 创建新的约束。可以看到出现了一个新的约束, 它同样被自动选中了。一个约束被选中时, 它影响到的所有视图都会用黄色突出显示, 如图 5-11 所示。你可能注意到了, 在创建这个新约束之前, 两个标签的宽度并不相同, 但是现在这两个标签的宽度完全相同, 这正是这个新约束带来的效果。

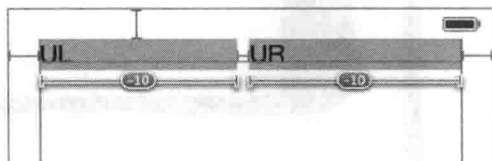


图 5-11 新约束使顶部的两个标签具有相同的宽度

如果这时再次运行应用, 应该能够在旋转设备时看到顶部的两个标签在水平方向上扩展到整个屏幕, 参见图 5-9。

在这个例子中, 所有的标签在多个方向上都是可见的并且位置正确, 但是屏幕上仍然有很多空间未使用。如果设置另外两行标签占满视图宽度或者允许改变标签的高度, 可能会更好一些, 这样可以减少界面上的空白空间。可以随意添加或者更改这 6 个标签上的约束实现不同效果, 或者也可以再多增加几个标签。除目前已经介绍过的之外, 在 **Editor>Pin** 菜单下可以看到更多创建约束的选项。如果错误地添加了一个不想要的约束, 可以选中这个约束然后按 **Backspace** 键删除, 或者在属性检查器中对其进行配置。可以多试几次, 直到对约束的基本工作原理比较熟悉为止。从现在开始直到本书结束, 都会大量使用约束, 如果你希望了解更详细的内容, 可以在 Xcode 的文档窗口中搜索 **auto layout** (自动布局)。

在动手实验的过程中, 你一定会发现有时不管如何组合使用各种约束, 都无法实现想要的效果。在这种情况下, 就需要完全重新布置界面上的元素, 而无法依赖约束, 这时编写少量的代码是必要的。下面就来看一下。

## 5.4 旋转时重构视图

回到 Xcode, 如之前一样新建一个单视图项目, 命名为 **Restructure**。接下来就介绍通过手动指定每个组件的形状尺寸, 构造一个在纵向模式和横向模式切换时尺寸会稍有变化的布局。仍然使用界面构建器配置 GUI 并且与对象进行关联, 但是需要使用代码把 GUI 的各个部分放置到目标位置。

我们现在要构造的 GUI 由一个大的内容区域和一组实现各种操作的小按钮组成。如果设备是纵向模式, 按钮会均匀分布在屏幕底部。而在横向模式时, 它们会在右边排成一列。图 5-12 展示了这两种状态。

选中 **Main.storyboard** 开始编辑 GUI。因为没有什么特别想要显示的内容, 索性就放一个带颜

色的大矩形。从对象库中拖出一个 UIView 到容器视图中。你会发现它会自动拉伸直到完全覆盖容器视图，这不是你想要的结果。所以在它仍然被选中时，使用尺寸检查器将新视图的宽度和高度都调整为 280 像素。接下来切换到属性检查器并使用 Background 弹出菜单选取一个背景颜色。你可以选择任何你喜欢的颜色，只要不是白色就行，以便视图在背景前能够突出明显。最后将视图拖动到父视图顶端中心的位置（参照图 5-13）。

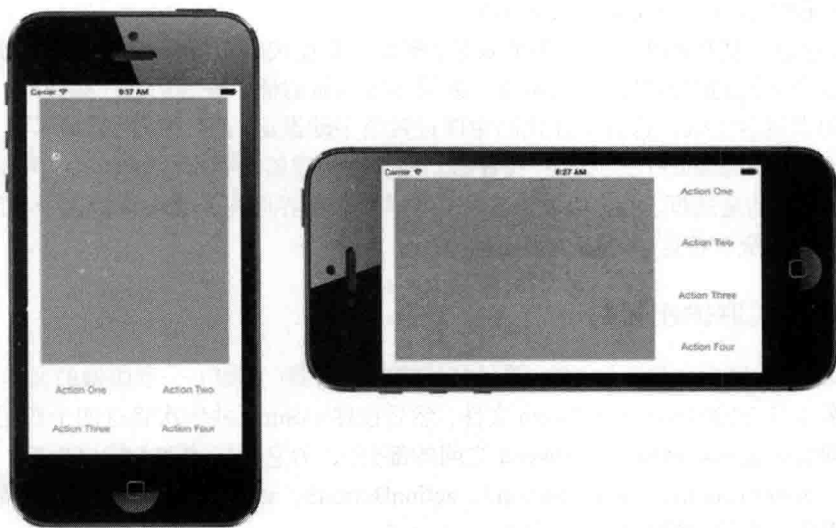


图 5-12 Restructure 应用程序在两个方向上的的最终界面布局

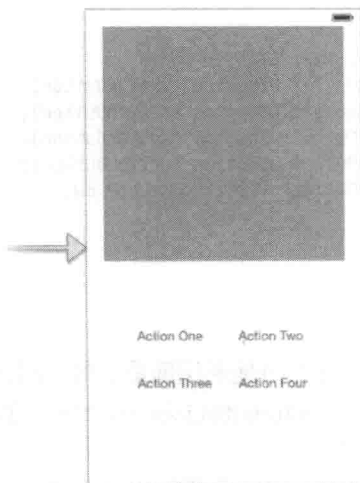


图 5-13 Restructure 视图基本的纵向布局

现在从对象库中拖出一个按钮并将它放在视图的底部。双击以选中里面的文本并将它改成



Action One。现在按住 option 键拖动这个按钮复制出三个拷贝,并将它们均匀地铺开(参考图 5-13)。你不需要现在就去进行精准的对齐,因为我们将代码中调整它们的位置。其他三个按钮的标题也分别改成 Action Two、Action Three 和 Action Four。

如果现在旋转屏幕,你能猜到会发生什么吗?继续在模拟器中运行应用程序并观察结果。出现的样子你肯定不会满意。通常来说,每个视图的位置都是绑定与左上角之间的距离来决定的,这也意味着小视图有可能会又要再在底部消失了。

事实上,这里只使用约束无法达预期效果,所以需要在代码中调整布局。编写布局代码之前,首先需要对这个分镜的内容禁用约束系统。如果不这么做的话,在应用程序编译时,Xcode 自动创建的自动约束就会生效,这会使在代码中通过数值手动设定 GUI 的努力前功尽弃。

按下 option+command+1 打开文件检查器。在界面构建器文档部分的中间,可以看到一个名为 Use Autolayout 的复选框。单击以取消选中,这样就会使界面构建器不再对这个界面使用约束。现在,你就可以完全掌控这个界面中视图的布局了。

### 5.4.1 创建并关联输出接口

确保选中了 Main.storyboard 文件,并且打开辅助编辑器(正如上一章所做的那样)。确保 GUI 布局区域的旁边是 BIDViewController.m 文件,然后按住 Control 并依次将这四个按钮拖向代码的类扩展中(即文件顶部@interface 和@end 之间的部分),为它们创建四个输出接口,并分别命名为 bigButton、actionButton1、actionButton2、actionButton3、actionButton4。现在对较大的视图也进行同样的操作,并给其输出接口命名为 contentView。

将这 5 个视图都关联到输出接口之后,BIDViewController.m 文件的顶部代码应如下所示:

```
#import "BIDViewController.h"

@interface BIDViewController ()
@property (weak, nonatomic) IBOutlet UIButton *actionButton1;
@property (weak, nonatomic) IBOutlet UIButton *actionButton2;
@property (weak, nonatomic) IBOutlet UIButton *actionButton3;
@property (weak, nonatomic) IBOutlet UIButton *actionButton4;
@property (weak, nonatomic) IBOutlet UIView *contentView;

@end
```

### 5.4.2 旋转时移动按钮

如果要移动这些视图的位置以便充分地利用屏幕上的空间,则需要覆盖 BIDViewController.m 中的 willAnimateRotationToInterfaceOrientation:duration: 方法。这个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。

在 BIDViewController.m 最后的@end 之前添加以下方法:

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

```

        [self doLayoutForOrientation:toInterfaceOrientation];
    }

```

可以看到，这个方法只是把新的方向传递给另一个方法（暂时还没有实现这个方法）。至于为什么要调用一个新的方法来实现这个功能，一会儿你就会明白了。新方法的内容如下：

```

- (void)doLayoutForOrientation:(UIInterfaceOrientation)orientation {
    if (UIInterfaceOrientationIsPortrait(orientation)) {
        [self layoutPortrait];
    } else {
        [self layoutLandscape];
    }
}

```

doLayoutForOrientation:方法仅仅将操作权分给了另外两个方法的其中一个，它们都能将这些视图摆放到正确的位置上。我们还没有实现它们的代码，在这里我们会使用几个常量来进行一些布局设定。

```

static const CGFloat buttonHeight = 40;
static const CGFloat buttonWidth = 120;
static const CGFloat spacing = 20;

- (void)layoutPortrait {
    CGRect b = self.view.bounds;

    CGFloat contentWidth = CGRectGetWidth(b) - (2 * spacing);
    CGFloat contentHeight = CGRectGetHeight(b) - (4 * spacing) -
        (2 * buttonHeight);

    self.contentView.frame = CGRectMake(spacing, spacing,
                                         contentWidth, contentHeight);

    CGFloat buttonRow1y = contentHeight + (2 * spacing);
    CGFloat buttonRow2y = buttonRow1y + buttonHeight + spacing;

    CGFloat buttonCol1x = spacing;
    CGFloat buttonCol2x = CGRectGetWidth(b) - buttonWidth - spacing;

    self.actionButton1.frame = CGRectMake(buttonCol1x, buttonRow1y,
                                           buttonWidth, buttonHeight);

    self.actionButton2.frame = CGRectMake(buttonCol2x, buttonRow1y,
                                           buttonWidth, buttonHeight);

    self.actionButton3.frame = CGRectMake(buttonCol1x, buttonRow2y,
                                           buttonWidth, buttonHeight);

    self.actionButton4.frame = CGRectMake(buttonCol2x, buttonRow2y,
                                           buttonWidth, buttonHeight);
}

- (void)layoutLandscape {
    CGRect b = self.view.bounds;

```

```
CGFloat contentWidth = CGRectGetWidth(b) - buttonWidth - (3 * spacing);
CGFloat contentHeight = CGRectGetHeight(b) - (2 * spacing);
```

```
self.contentView.frame = CGRectMake(spacing, spacing,
                                     contentWidth, contentHeight);
```

```
CGFloat buttonX = CGRectGetWidth(b) - buttonWidth - spacing;
```

```
CGFloat buttonRow1y = spacing;
```

```
CGFloat buttonRow4y = CGRectGetHeight(b) - buttonHeight - spacing;
```

```
CGFloat buttonRow2y = buttonRow1y + floor((buttonRow4y - buttonRow1y)
                                           * 0.333);
```

```
CGFloat buttonRow3y = buttonRow1y + floor((buttonRow4y - buttonRow1y)
                                           * 0.667);
```

```
self.actionButton1.frame = CGRectMake(buttonX, buttonRow1y,
                                       buttonWidth, buttonHeight);
```

```
self.actionButton2.frame = CGRectMake(buttonX, buttonRow2y,
                                       buttonWidth, buttonHeight);
```

```
self.actionButton3.frame = CGRectMake(buttonX, buttonRow3y,
                                       buttonWidth, buttonHeight);
```

```
self.actionButton4.frame = CGRectMake(buttonX, buttonRow4y,
                                       buttonWidth, buttonHeight);
```

```
}
```

这些方法都很相似，不过会出现不同的结果。它们首先将父视图的位置尺寸存储在一个本地变量中，方便之后使用。接下来它们通过矩形的尺寸和常量值计算得出大内容视图的最佳位置，然后设置视图的正确位置和尺寸。之后计算那些按钮的位置，并同样将它们放置在合适的位置。

这里有些深奥，不过也有很多有趣的内容。所有视图（包括按钮等控件）的尺寸和大小都是由名称为 `frame` 和 `bound` 的属性来指定的，两者都是 `CGRect` 类型的结构体。它们的不同之处在于：视图的 `frame` 属性用来描述它在父视图坐标系中的位置，而视图的 `bound` 属性则用来描述在其自身坐标系中的位置（在计算子视图的父级归属时会很有用）。它们有很多种用途，但最普遍的用途就像我们在这里所用的那样：使用多个对象的位置来得出另一个的位置。按照惯例，如果你想要得到某个视图的位置，你可以先获得父视图的 `bound` 属性或同层级视图的 `frame` 属性作为基本值，然后设置目标视图的 `frame` 属性以放到正确的位置。在整本书中会遇到很多这样的例子。

此外，苹果公司提供了 `CGRectMake()` 函数来让你通过指定 `x` 和 `y` 值以及宽度和高度来轻松创建一个 `CGRect` 值。我们也会使用 `CGRectGetHeight()` 和 `CGRectGetWidth()` 函数从已有的 `CGRect` 值中轻而易举地获取到高度或宽度。

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` 的结构体。`CGRectMake` 是苹果公司提供的一个函数，通过指定 `x` 和 `y` 的位置以及 `width` 和 `height` 就可以方便地创建一个 `CGRect`。

**注意** 你可能已经注意到这段代码中有些之前没有见过的东西。还记得我们使用界面构建器的“拖曳生成代码”特性为所有按钮创建属性吗？这些属性同时声明了存取方法，可以使用存取方法或者 Objective-C 的点操作符来访问属性的值（比如，`[self contentView]`或者 `self.contentView`）。过去，必须在.m 文件中实现这些方法，或者自己编写方法体代码，或者使用 `@synthesize` 声明自动实现。如果你在 2012 年以前曾做过 Objective-C 开发，可能还会记得这类事情。最近几年内，默认的行为有了一点儿变化。如果没有使用 `@synthesize` 声明，当使用“点击生成代码”特性时，编译器会自动创建这些方法。同时也为各个属性创建了相应的实例变量，实例变量的命名规则是下划线（`_`）+属性名称。作为例子，看一下这行代码：

```
@synthesize bigButton = _bigButton;
```

在 Xcode 4.4 及之后的版本中，这行代码是完全多余的。如果已经声明了一个名为 `bigButton` 的属性，那么完全可以省略这行代码。可以完全省略对合成存取方法的显式声明，除非你希望对实例变量使用不同的命名约定。

结束之前，还有一件事要做。因为我们使用代码为所有的 GUI 元素设置精确位置，所以对所有的 GUI 布局都应该使用代码完成，包括从分镜或 nib 文件中加载视图对象的时刻。可以在 `viewDidLoad` 方法中添加如下加粗显示的代码来实现：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后（通常是从分镜或 nib 文件加载），做一些额外的设置
    UIApplication *app = [UIApplication sharedApplication];
    UIInterfaceOrientation currentOrientation = app.statusBarOrientation;
    [self doLayoutForOrientation:currentOrientation];
}
```

这里获取了 `UIApplication` 类的共享实例并且访问它的 `statusBarOrientation`。这样就可以知道视图加载时的屏幕方向。把当前方向传递给之前编写的 `doLayoutForOrientation:` 方法，这样所有的界面元素都会被放置到正确的位置。

保存代码。现在构建并运行代码，查看其实际效果。尝试着旋转屏幕，看看按钮是如何移动到新位置的，旋转完成之后这些按钮应该非常完美地对齐在屏幕右边。屏幕旋转回来之后，它们也会回到一开始所在的位置。相当流畅！你还可以试试在其他模拟器中运行。Xcode 能让你选择使用 3.5 英寸还是 4 英寸 iPhone 模拟器上运行。它们的运行结果都会是一样的。

## 5.5 小结

本章介绍了在应用程序中实现自动旋转的几种完全不同的方法。我们学习了使用约束来定义视图布局，以及如何使用代码在 iOS 设备发生旋转时重构视图。

在下一章中，我们将会开始接触真正的多视图应用程序。我们目前编写的所有应用程序都只使用了一个内容视图。然而，许多复杂的 iOS 应用程序（如邮件和通讯录）必须使用多个视图和视图控制器才能实现，第 6 章就会进行相关介绍。

到目前为止,我们编写的应用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,但是只有能够根据用户输入在不同视图之间切换,才会彰显 iOS 平台的真正威力。多视图应用具有各种不同的风格,但是无论它们在屏幕上的显示方式如何,它们的底层机制都是相同的。

本章将重点介绍多视图应用的结构和切换内容视图的基本知识,从头开发一个多视图应用。我们将编写一个自定义的控制器类,用于在两个不同的内容视图之间切换,打好基础,为之后能够充分利用苹果公司提供的各种多视图控制器。

在开始开发应用之前,我们先来看看多视图应用有哪些用途。

## 6.1 多视图应用的常见类型

严格来讲,我们在之前的应用中处理过多个视图,因为按钮、标签和其他控件都是 `UIView` 的子类,都是视图层次结构的一部分。但是苹果公司在文档中使用的术语视图,通常指具有相应视图控制器的 `UIView` 或其子类。这些视图类型有时也称为内容视图,因为它们是应用内容的主要容器。

实用工具应用是最简单的多视图应用。它主要使用一个视图,还提供了一个视图用于配置应用或提供除主视图之外的更多信息。iPhone 自带的股票 (Stocks) 应用就是一个很好的例子 (参见图 6-1)。点击右下角的按钮可以切换到配置视图,用于配置应用所跟踪的股票列表。

iPhone 还自带了几个标签栏应用,包括电话应用 (参见图 6-2) 和时钟应用。标签栏应用是一种多视图应用,它在屏幕底部显示一行按钮,称为标签栏 (Tab Bar)。单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 应用中,单击通讯录时显示的视图与单击拨号键盘时显示的视图不同。

另一种常见的多视图 iPhone 应用是基于导航的应用,这类应用拥有一个导航控制器,使用导航栏控制一系列分层的视图。设置应用就是一个很好的例子。在设置应用中,第一个视图是一系列行,每行对应一类设置或某个应用。点击其中的某一行将会进入一个新的视图,可以在这里定制某一类型的设置。有些视图显示一个列表,点击某一行就可以进入更深层次的视图。导航控制器跟踪所在的视图深度,并且向你提供控制权,让你可以回到之前的视图。



图 6-1 iPhone 自带的股票应用包含两个视图，一个用于显示数据，另一个用于配置股票列表



图 6-2 电话应用是使用标签栏的多视图应用的一个例子

例如，如果选择“声音”，就会显示一个包含声音相关选项列表的视图。该视图的顶部是一个导航栏，其中包含一个标题为“设置”的左箭头，点击它可返回上一个视图。声音选项中有一行名为“电话铃声”。单击这一行就会进入一个新的视图（参见图 6-3），其中显示了一个铃声列表和一个导航栏，导航栏可用于返回“声音”首选项主视图。在希望显示具有不同层次结构的视图时就可以使用这种基于导航的应用形式。



图 6-3 iPhone 的设置应用是使用导航栏的多视图应用的一个例子

在 iPad 上,大部分基于导航的应用都是使用分割视图 (split view) 实现的,比如邮件应用。在分割视图中,导航元素在屏幕左侧显示,被选中的项在右侧显示。第 10 章将更详细地介绍分割视图和其他 iPad 特有的 GUI 元素。

由于视图在本质上是分层的,因此甚至可以在一个应用中结合使用不同的视图交换机制。例如,iPhone 的音乐应用使用标签栏来切换音乐的不同组织方法,使用导航控制器及其关联的导航栏实现按照所选方法浏览音乐。标签栏和导航栏分别位于屏幕的底部和顶部,如图 6-4 所示。

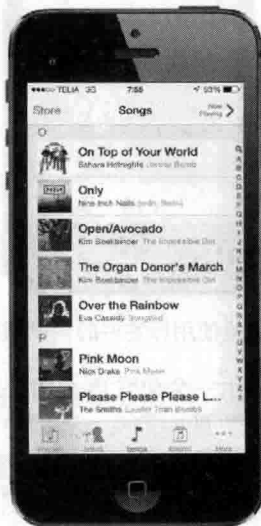


图 6-4 Music 应用同时使用了导航栏和标签栏



一些应用使用了工具栏，人们通常会将它与标签栏混淆。标签栏用于从两个或更多选项中选择一项，而且只能选择一项。工具栏可以包含按钮和其他一些控件，但这些项并不是互斥的。Safari 主视图的底部就有一个不错的工具栏（参见图 6-5）。如果将 Safari 视图底部的工具栏与电话或音乐应用底部的标签栏进行比较，将会发现它们很容易区分。标签栏有多个分段，只有被选中的那个才会有颜色，而工具栏上通常每个能点的按钮都是有颜色的。

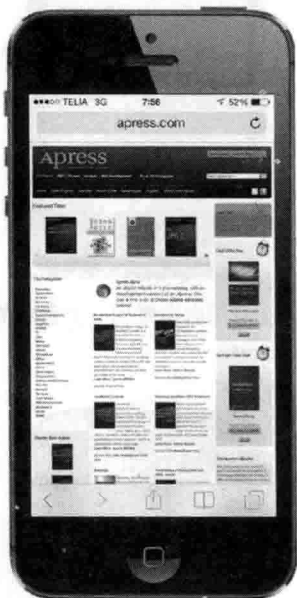


图 6-5 移动版 Safari 底部有一个工具栏，可以在工具栏上随意放置各种控件

所有这些多视图应用都使 UIKit 提供的某个具体的控制器类。标签栏界面是使用 UITabBarController 类实现的，而导航界面是使用 UINavigationController 实现的。我们将在下一章中详细描述如何使用它。

## 6.2 多视图应用的体系结构

本章将构建 View Switcher（视图切换器）应用，它的外观非常简单，但是从代码的角度来讲，它是目前为止本书所介绍的最复杂的应用。View Switcher 由 3 个不同的控制器、1 个分镜和 1 个应用委托组成。

首次启动时，View Switcher 看上去与图 6-6 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。

按下 Switch Views（切换视图）按钮时，背景将会变为黄色，按钮的名称也会生发变化（参见图 6-7）。

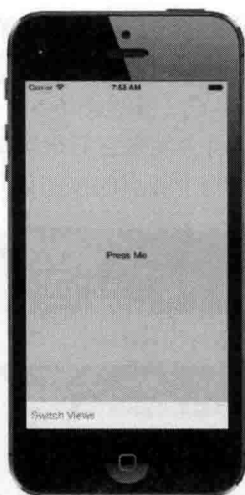


图 6-6 首次启动 View Switcher 应用后会看到一个蓝色的视图，这个视图上有一个按钮和一个带按钮的工具栏

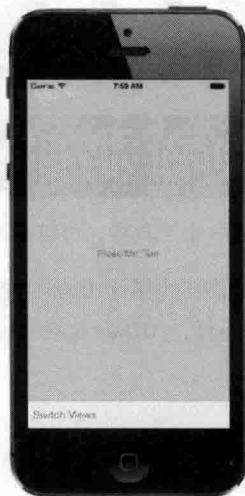


图 6-7 按下 Switch Views 按钮之后，蓝色视图会翻转过去，然后显示一个黄色视图

无论按下 Press Me（请按下按钮）还是 Press Me, Too（也请按下这个按钮）按钮，都会弹出一个警告对话框，指出按下了哪个视图的按钮（参见图 6-8）。

尽管编写一个单视图应用也能实现相同的功能，但我们采用这种比较复杂的方法演示多视图应用的机制。在这个简单的应用中，实际上有 3 个视图控制器在进行交互：一个用于控制蓝色视图，一个用于控制黄色视图，还有一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。

开始构建应用之前，先来了解一下 iPhone 多视图应用的组织方式。几乎所有的多视图应用都使用相同的基本模式。

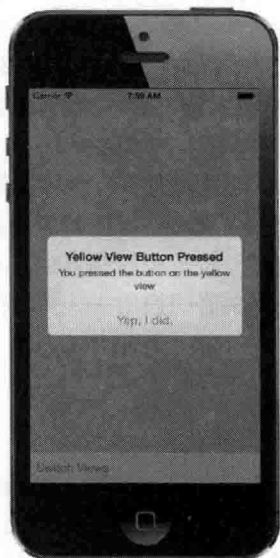


图 6-8 按下 Press Me 或者 Press Me, Too 按钮就会显示一个警告对话框

### 6.2.1 根控制器

在这里，分镜扮演着重要角色，它包含了应用的所有视图和视图控制器。我们先创建一个空白的分镜，然后再添加一个控制器类的实例，它负责管理当前向用户显示哪个视图。我们将这个控制器称为根控制器，因为它是用户看到的第一个控制器，也是应用加载时加载的控制器。这个根控制器通常是 UINavigationController 或 UITabBarController 的一个实例，但也可以是 UIViewController 的自定义子类。

在多视图应用中，根控制器的任务是接受两个或更多其他视图，并根据用户输入将合适的视图呈现给用户。例如，标签栏控制器会根据最后单击的标签项，在不同的视图和视图控制器之间进行切换。用户浏览分层数据时，导航控制器也具备相同的功能。

---

**注意** 根控制器是应用的主视图控制器，因此，它也是用于指定是否应该自动旋转到新方向的视图。但是，根控制器可以将这类任务转交给当前活动的控制器。

---

在多视图应用中，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接口和操作方法。例如，在标签栏应用中，点击标签栏将转到标签栏控制器中，但是点击屏幕上其他任何位置都将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

在多视图应用中,每个视图控制器控制一个内容视图,应用的用户界面就是在这些内容视图中构建的。这样的组合在分镜中被称为**场景 (scene)**。每个场景都是由一个视图控制器和一个内容视图 (可能是 `UIView` 或其子类的实例对象) 构成的。除非所做的工作非常特殊,否则内容视图始终具有一个相关联的视图控制器,而且有时会继承 `UIView` 类。尽管可以不使用界面构建器,而是直接在代码中创建界面,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。

在这个项目中,我们将为每个内容视图创建一个新的控制器。根控制器控制着一个内容视图,这个内容视图包含一个位于屏幕底部的工具栏。根控制器会加载一个蓝色的视图控制器,将蓝色的内容视图作为根控制器视图的子视图。按下根控制器的 **Switch Views** 按钮 (这个按钮位于工具栏中) 时,根控制器会将蓝色的视图控制器切换出去,同时把黄色的视图控制器切换进来,必要时还会实例化这个黄色的视图控制器。糊涂了? 别担心,看看代码就明白了。

## 6.3 构建 View Switcher 项目

理论已经足够了。现在开始着手构建项目。选择 **File>New>New Project...** 或者按下 **Shift+Command+N**。模板选择表单打开后,选择 **Empty Application** (参见图 6-9), 然后点击 **Next** 按钮。在向导的下一页输入 **View Switcher** 作为 **Product Name** 的值, **Class Prefix** 保持 **BID** 不变,然后将 **Device Family** 弹出按钮设置为 **iPhone**。另外确保不要选中 **Use Core Data** 复选框。全部都完成之后请点击 **Next** 按钮继续。在下个界面中,指定这个项目在硬盘上的保存位置,最后点击 **Create** 按钮,创建一个新的项目目录。

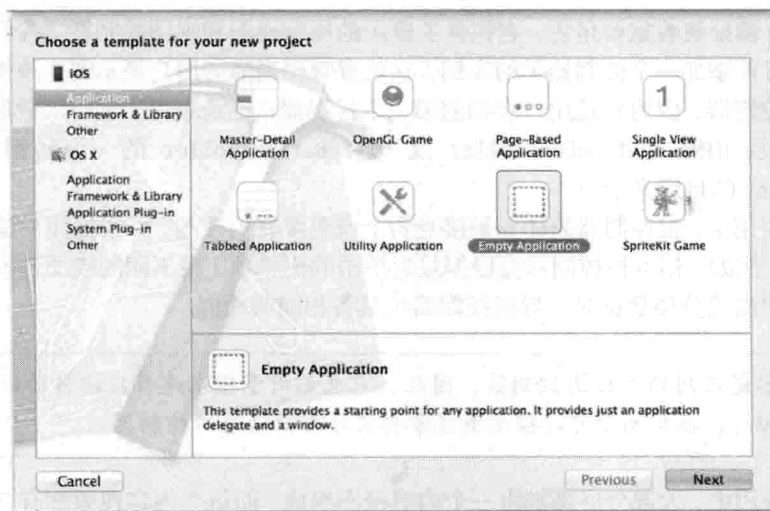


图 6-9 使用 Empty Application 项目模板创建一个新项目

刚才选择的模板实际上比我们以前使用的 Single View Application 模板更加简单。这个模板仅提供一个用来生成自身窗口应用委托和一个资源目录，没有视图，没有控制器，什么都没有。

**注意** 窗口是 iOS 中最基本的容器。虽然可以同时屏幕上看到多个窗口，但每个应用仅拥有一个属于它的窗口。例如，如果应用在运行时收到了一条 SMS 或者 iMessage 消息，可以看到这条消息在它自己的窗口中显示，而且是显示在当前运行的应用窗口的前面。你的应用无法访问这个覆盖在自己上面的窗口，因为它属于 Messages 应用。

创建应用时，Empty Application 模板并不常用，但是，在本例中从无到有开始创建应用，能够更好地体会多视图应用的组织形式。

展开项目导航面板中的 View Switcher 文件夹以及其中的 Supporting Files 文件夹，看一下其中包含哪些内容。在 View Switcher 文件夹中，可以看到两个用于实现应用委托的文件。在 Supporting Files 文件夹中，可以看到 View Switcher-Info.plist 文件和 InfoPlist.strings 文件（其中包含 Info.Plist 文件的本地化版本），还有标准的 main.m 文件和预编译的头文件（View Switcher-Prefix.pch）。应用所需的其他内容都需要自己创建。

### 6.3.1 创建视图控制器和分镜

从头创建多视图应用，一个麻烦之处就是必须创建一些互相关联的对象。先来创建组成应用的所有文件，然后再使用 Interface Builder 编写代码。首先创建所有文件，这样就可以使用 Xcode 的代码感知功能更快地编写代码。如果某个类未声明，代码感知功能就无法知道这个类的信息，这样就必须每次都输入完整的类名，这样会花费很多时间，而且容易出错。

幸好，除了项目模板，Xcode 还为许多标准文件类型提供了文件模板，这极大简化了创建应用基本框架的过程。

单击项目导航面板中的 View Switcher 文件夹，然后按下 Command+N 或者选择 File>New>File...。然后会看到图 6-10 所示的窗口。

在左侧面板中选中 Cocoa Touch，可以看到一些用于常用的 Objective-C 结构的模板，比如类（class）、分类（category）等。选择 Objective-C class，然后点击 Next。在向导的下一页将看到两个名称分别为 Class 和 Subclass of 的字段。在 Subclass of 字段中输入 UIViewController 作为新类的父类。然后可以看到 Xcode 自动在 Class 字段中填入了一个名称，将其改为 BIDSwitchViewController。然后来看其他 3 个选项，它们用于配置这个子类。

- ☐ 第二项是名为 Targeted for iPad 的复选框。如果默认已经选中了这一项，现在应该取消选中（因为我们并不是创建 iPad GUI）。
- ☐ 第三项是名为 With XIB for user interface 的复选框。如果这个复选框处于选中状态，则单击它以取消选择。如果选择该选项，Xcode 就会创建一个与此控制器类相关联的 nib 文件。我们的整个 GUI 都是由分镜文件管理的，因此取消它的勾选。

点击 Next。此时会出现一个窗口，让你选择一个特别的目录用于保存文件，并且为文件选择一个分组和目标。默认情况下，该窗口将会显示与你在项目导航面板中选中的文件夹最相关的目录。考虑到一致性，应该将这个新类保存在 View Switcher 文件夹下，这个文件夹是 Xcode 在创建项目时创建的，其中应该已经包含了 BIDAppDelegate 类。所有作为项目的一部分创建的 Objective-C 类都会被 Xcode 放置在这里。这个文件夹也是保存自定义类的理想位置。

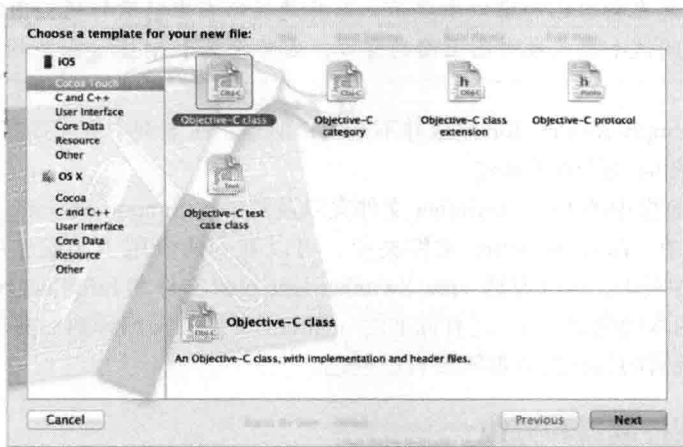


图 6-10 用于创建新的视图控制器子类的模板

在窗口的下半部分，可以找到 Group 弹出列表。可以将该新文件添加到 View Switcher 分组。最后，在 Targets 列表中选择 View Switcher 目标，然后点击 Create 按钮。

Xcode 会将两个文件（BIDSwitchViewController.h 和 BIDSwitchViewController.m）添加到 View Switcher 文件夹。BIDSwitchViewController 将作为根控制器，用于控制其他视图的切换。现在需要为两个内容视图（应用会在这两个视图间切换）创建控制器。重复之前的步骤两次，创建 BIDBlueViewController.m、BIDYellowViewController.m 和它们对应的.h 文件，将它们添加到项目层次结构中的相同位置。

---

**警告** 请确保拼写正确，因为这里的输入错误会导致创建的类与本章后面的源代码不匹配。

---

下一步是创建一个分镜文件，我们要在这里为之前添加的每个内容视图设定一个场景。单击项目导航面板中的 View Switcher 文件夹，然后再次按下 `command+N` 或选择 `File>New>File...`。这一次，选择左侧面板中 iOS 下面的 User Interface（参见图 6-11）。接下来，选择 Storyboard 模板的图标，这样将会创建一个空白的分镜。点击 Next。在接下来的窗口中，为 Device Family 弹出菜单选择 iPhone，然后单击 Next 按钮。

提示请求文件名时输入 Main.storyboard。和前面一样，在 Where 弹出菜单中选择 View Switcher 文件夹作为存储位置。选中 View Switcher，确保在 Group 弹出菜单中选择了 View Switcher，并

且勾选 Target 旁的 View Switcher 复选框，然后点击 Create。当 Main.storyboard 文件在项目导航面板的 View Switcher 分组中出现时，就表示创建成功了。

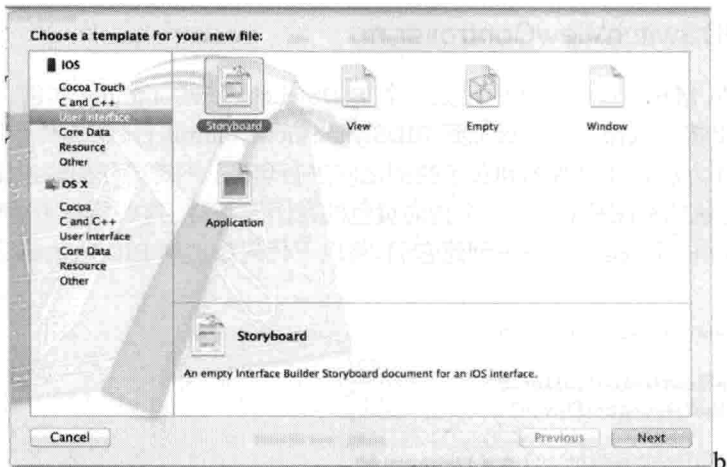


图 6-11 使用 User Interface 部分的模板创建一个新的分镜

完成之后，就拥有了所需的所有文件。接下来将所有部分衔接起来。第一步要做的就是让 Xcode 知道用刚刚创建的分镜为基础来设定应用 GUI。方法是选中项目导航栏最上面的 View Switcher 条目，然后切换到编辑区域的 General 标签项。这样会调出一个多分段的配置视图。在 Deployment Info 部分中，Main Interface 弹出菜单的值选为 Main.storyboard。完成之后，应用就会在启动时自动通过分镜的内容创建初始化界面。我们之前没有涉及过这些，但前面创建的每一个项目都采用了同样的起始设定，能够节省这么多功夫，这都要感谢 Xcode 的 Single View Application 项目模板。

### 6.3.2 修改应用委托

多视图之旅的第一站是应用委托。单击项目导航面板中的 BIDAppDelegate.m 文件（确保点击的是应用委托而不是 BIDSwitchViewController.m），并对文件进行如下更改：

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:
        [[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

你需要清除标有删除线的代码。在空白的应用中（就像我们现在使用的），需要在代码中手



动创建应用窗口。因为我们要从分镜从加载 GUI，所以这些代码是用不到的，可以放心大胆地删除它们。

### 6.3.3 修改BIDSwitchViewController.m

由于我们将在 Main.storyboard 中建立一个 BIDSwitchViewController 实例，因此现在是时候将所需的每个输出接口或操作方法添加到 BIDSwitchViewController.m 头文件中了。

需要一个操作方法在蓝色视图和黄色视图之间进行切换。不需要任何输出接口，但需要两个指针，一个指向蓝色的视图控制器，一个指向黄色的视图控制器。这些指针不需要成为输出接口，因为我们将在代码中而不是在分镜中创建它们。将以下代码添加到 BIDSwitchViewController.m 文件靠上的位置：

```
#import "BIDSwitchViewController.h"

#import "BIDYellowViewController.h"
#import "BIDBlueViewController.h"

@interface BIDSwitchViewController ()

@property (strong, nonatomic) BIDYellowViewController *yellowViewController;
@property (strong, nonatomic) BIDBlueViewController *blueViewController;
```

```
@end
```

接下来请在文件末尾的@end 语句上方添加没有实现代码的空白操作方法，如下所示。

```
- (IBAction)switchViews:(id)sender
{
}
```

```
@end
```

之前我们曾在界面构建器中直接添加操作方法，而这里我们使用另一种方法也能做到，因为在 Interface Builder 中是可以看到已经在源代码中定义的输出接口和操作方法。现在我们已经声明了所需的操作方法，可以在分镜中设置它的控制器。

### 6.3.4 添加视图控制器

保存源代码，点击 Main.storyboard，编辑这个应用的主 GUI。你将看到一片完全空白的编辑区域，没有任何视图和控制器。通过对象库找到 View Controller 并将其拖动到编辑区域中。你会看到熟悉的 iPhone 尺寸大小的矩形视图，以及下面的一排图标。

默认情况下，这个场景中的视图控制器被设定为 UIViewController 的一个实例。需要将它改为 BIDSwitchViewController，这样界面构建器才能让我们与 BIDSwitchViewController 的输出接口和操作方法进行关联。单击场景下面的视图控制器图标，然后按下 option+command+3 打开身份检查器（参见图 6-13）。

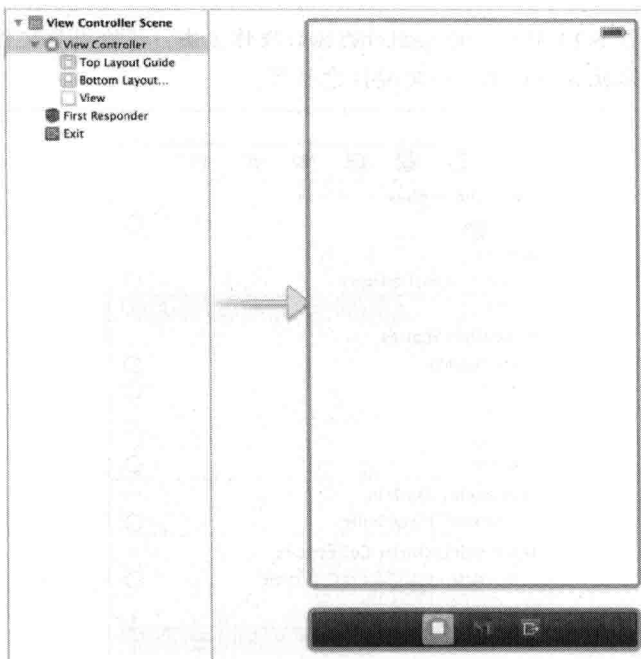


图 6-12 Main.storyboard，展示了应用会使用的第一个场景

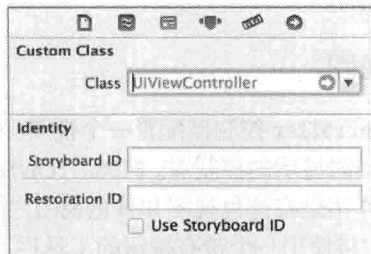


图 6-13 注意，Custom Class 字段当前在身份检查器中被设置为 UIViewController，需要将它替换为 BIDSwitchViewController

借助身份检查器，可以指定当前选定对象所属的类。我们的视图控制器目前被指定为 UIViewController 类，这个类没有定义任何操作方法。单击 Custom Class 区域中标题为 Class 的下拉框（位于检查器顶部，内容为 UIViewController）。将其改为 BIDSwitchViewController。

完成更改之后按下 option+command+6 切换到关联检查器，会看到 switchViews: 操作方法出现在 Received Actions 部分（参见图 6-14）。关联检查器的 Received Actions 部分显示了当前类中定义的所有操作方法。将视图控制器改为 BIDSwitchViewController 时，BIDSwitchViewController 的操作方法 switchViews: 就可以用来进行关联了。下一节将介绍如何使用这个操作方法。

**警告** 如果没有看到图 6-14 中所示的 switchViews:操作方法, 请检查类文件名的拼写。如果名称不正确, 内容就会不匹配。一定要注意拼写!

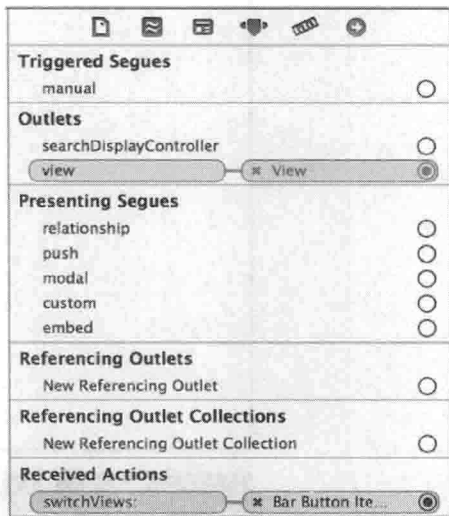


图 6-14 在关联检查器中可以看到 switchViews:操作方法已经添加到 Received Actions 部分保存分镜文件并继续下一步操作。

### 6.3.5 构建带有工具栏的视图

现在需要为 BIDSwitchViewController 控制器配置一个视图。提示一下, 这个新的视图控制器将作为根视图控制器——应用启动时显示的控制器。BIDSwitchViewController 的内容视图包含一个位于屏幕底部的工具栏。它的作用是在蓝色视图和黄色视图之间进行切换, 所以需要提供一种方式让用户更改视图。为此, 我们将使用一个带有按钮的工具栏。现在开始构建这个工具栏视图。

在界面构建器编辑区域中, 点击你之前添加到场景中的视图。这个视图是 UIView 类的一个实例, 如图 6-15 所示, 它目前是一片空白。我们将在这里构建应用 GUI。

现在, 向视图底部添加一个工具栏。从库中拖出一个 Toolbar 放置到视图底部, 如图 6-16 所示。我们希望无论视图尺寸为多少, 这个工具栏都能够一直固定在视图底部。解决方法是选择 Editor>Pin>Bottom Space to Superview 菜单项。这样就会创建一个能起到作用的约束。

为了确保我们没有出现问题, 请点击运行按钮并在模拟器中启动应用。你将会在打开时看到一个纯白界面的应用, 底部是角落有一个按钮的浅灰色工具栏。假如没有显示出这样的场景, 请向上回溯之前的步骤并检查究竟漏掉了哪一步。

该工具栏带有一个按钮。用户使用该按钮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 return 键提交更改。

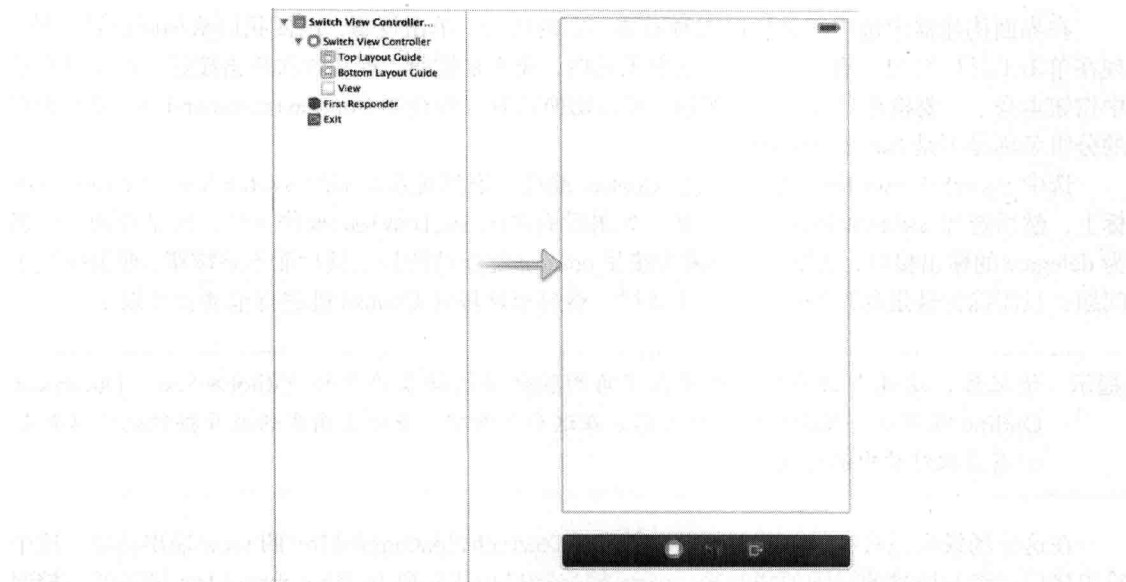


图 6-15 分镜中的新空白视图，很快就会向其中添加一些好玩的东西

现在，将工具栏按钮关联到操作方法。在此之前，应该注意：工具栏上的按钮与其他 iOS 控件不同。它们仅支持一个目标操作方法，并且只在特定时刻触发这个方法——相当于其他 iOS 控件上的 Touch Up Inside 事件。

6



图 6-16 将一个工具栏拖到新视图上。请注意该工具栏包含一个标签为 Item 的按钮

在界面构建器中选中工具栏按钮时需要一定的技巧。单击视图，以便我们从相同位置开始。现在单击工具栏按钮。请注意，这会选中工具栏，而不是按钮。现在再次单击按钮。这样就会选中按钮本身了。要检查是否选中了按钮，可以切换到属性检查器（option+command+4）看看顶部的分组名称是不是 Bar Button Item。

选中 Switch Views 按钮之后，按住 Control 键将它拖到场景底部的 Switch View Controller 图标上，然后选择 switchViews:操作方法。如果没有弹出 switchView:操作方法，而是看到一个名为 delegate 的输出接口，这很可能是因为按住 control 键拖动的是工具栏而不是按钮。要解决这个问题，只需确保选定的是按钮而不是工具栏，然后重新按住 Control 键进行拖动就可以了。

---

**提示** 请记住，通过点击界面构建器左下角的按钮或选择菜单中的 Editor►Show Document Outline 就可以一直浏览到文件大纲。在这个界面中，点击三角形的展开按钮就可以查看视图层次结构中的任意元素。

---

在这个场景中还有一件事需要提及，就是 BIDSwitchViewController 的 view 输出接口。这个输出接口已经与场景的视图关联起来，view 输出接口是从父类 UIViewController 继承的，控制器使用 view 属性访问它管理的视图。在我们从对象库中拖动出 View Controller 场景的时候，界面构建器就立即构建了控制器和视图，并为我们做好了关联。太好了！

至此任务就完成了，保存文件。下一步就来实现 BIDSwitchViewController 类。

### 6.3.6 编写根视图控制器

现在是时候编写根视图控制器了。当用户单击 Switch Views 按钮时，根视图控制器负责在黄蓝两个视图之间进行切换。

在项目导航栏中选择 BIDSwitchViewController.m，并在 viewDidLoad 方法中添加下面粗体所显示的代码进行修改。

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // 加载完视图后，执行其他的额外设置
    self.blueViewController = [self.storyboard
                           instantiateViewControllerWithIdentifier:
                           @\"Blue\"];
    [self.view addSubview:self.blueViewController.view atIndex:0];
}
```

现在通过添加以下粗体代码来实现你之前创建的 switchViews:方法。

```
- (IBAction)switchViews:(id)sender
{ 如果该视图没有父视图，则释放它
    if (!self.yellowViewController.view.superview) {
        if (!self.yellowViewController) {
            self.yellowViewController = [self.storyboard
```

```

        instantiateViewControllerWithIdentifier:@"Yellow"];
    }
    [self.blueViewController.view removeFromSuperview];
    [self.view insertSubview:self.yellowViewController.view atIndex:0];
} else {
    if (!self.blueViewController) {
        self.blueViewController = [self.storyboard
            instantiateViewControllerWithIdentifier:@"Blue"];
    }
    [self.yellowViewController.view removeFromSuperview];
    [self.view insertSubview:self.blueViewController.view atIndex:0];
}
}
}

```

将以下代码添加到已有的 `didReceiveMemoryWarning` 方法中：

```

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // 释放没有用到的缓存的数据、图片等
    if (!self.blueViewController.view.superview) {
        self.blueViewController = nil;
    } else {
        self.yellowViewController = nil;
    }
}
}

```

我们修改的第一个方法 `viewDidLoad` 覆盖了 `UIViewController` 中的同名方法，这个方法会在 nib 加载时调用。我们是如何知道的呢？按住 `option` 键并单击方法名 `viewDidLoad`。查看出现的文档弹出窗口（参见图 6-17）。也可以在菜单中选择 `View>Utilities>Show Quick Help Inspector`，在 Quick Help 面板中查看类似的信息。`viewDidLoad` 方法是在超类 `UIViewController` 中定义的，视图加载完成时需要通知哪个类，就应该在哪个类中覆盖这个方法。

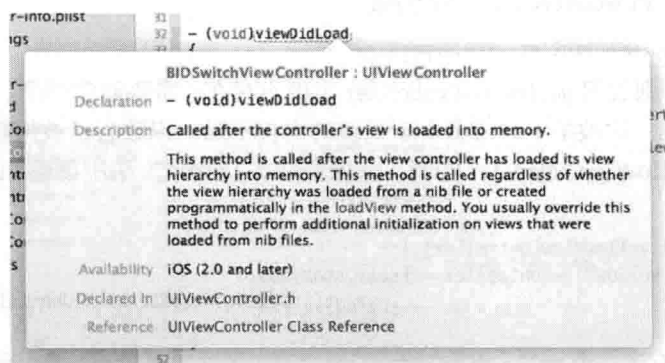


图 6-17 按住 `Option` 键并单击 `viewDidLoad` 方法名称时，就会显示这个文档窗口

这个 `viewDidLoad` 创建了一个 `BIDBlueViewController` 实例，使用 `instantiateViewControllerWithIdentifier:` 方法从根视图控制器所属的分镜中加载 `BIDBlueViewController` 实例。为了访问分镜中特定的视图控制器，我们将使用一个字符串来作为标识符（在本示例中是 `Blue`，我们将在

之后为分镜进行更多配置时设定此项)。BIDBlueViewController 实例创建之后,就将其赋给 blueView Controller 属性:

```
self.blueViewController = [self.storyboard
    instantiateViewControllerWithIdentifier:@"Blue"];
```

接下来,插入蓝色视图,作为根视图的一个子视图。将其插入到索引为 0 的位置,这将告诉 iOS 将这个视图放在其他所有视图之后。将这个视图作为背景视图,可以确保刚才在界面构建器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:self.blueViewController.view atIndex:0];
```

那么,为什么不在这里加载黄色视图呢?我们会在某个时刻加载黄色视图,但为什么不是现在呢?这个问题非常好。答案是,用户可能永远不会点击 Switch Views 按钮。用户可能只使用应用启动时出现的视图,然后就退出了。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,应该在真正需要黄色视图时再加载它。这种行为称为延迟加载 (lazy loading),这是一种降低内存开销的常用方式。对黄色视图的实际加载发生在 switchViews:方法中,来看一下这个方法。

switchViews:首先通过检查属性 yellowViewController 的 view 的父视图是否为 nil,判断出当前显示的是哪个视图。以下两种情况将返回 true。

- ❑ 如果 yellowViewController 存在,但是它的视图当前并没有显示给用户,这样它的视图就没有父视图,因为它的视图并没有在视图层次结构中显示,表达式的结果为 true。
- ❑ 如果 yellowViewController 不存在,因为还没有创建,或者从内存中清除了,这样也会返回 true。

然后检查 yellowViewController 是否存在。

```
if (!self.yellowViewController.view.superview) {
```

如果是空指针,则表明 yellowViewController 实例不存在,需要创建一个。发生这种情况可能是因为这是该按钮第一次被按下,或者由于系统的内存不足,以致它已被擦除。在这种情况下,需要创建一个 BIDYellowViewController 实例,就像在 viewDidLoad 方法中创建 BIDBlueViewController 实例一样:

```
if (!self.yellowViewController) {
    self.yellowViewController = [self.storyboard
        instantiateViewControllerWithIdentifier:@"Yellow"];
}
```

现在,我们已经有了一个 yellowViewController 实例(要么本来就有,要么刚才已创建了一个)。然后,从视图层次结构中删除 blueViewController 的视图,将 yellowViewController 的视图添加进来:

```
[self.blueViewController.view removeFromSuperview];
[self.view insertSubview:self.yellowViewController.view atIndex:0];
```

如果 self.yellowViewController.view.superview 不为 nil,就需要对 blueViewController 进行同样的处理。尽管在 viewDidLoad 中创建了一个 blueViewController 实例,但它仍然有可能因为内



存不足而被清除。在这个应用中，内存不足的可能性很小，但是我们仍然要合理使用内存，确保在继续操作之前拥有了一个 `blueViewController` 实例：

```
    } else {
        if (!self.blueViewController) {
            self.blueViewController = [self.storyboard
                                       instantiateViewControllerWithIdentifier:@"Blue"];
        }

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

除了在按下 Switch Views 按钮之前不会为黄色视图及其控制器浪费资源以外，延迟加载还能够释放当前未显示的视图所占的内存。当内存减少到系统设定的一个水平时，iOS 将调用 `UIViewController` 的 `didReceiveMemoryWarning` 方法，这个方法会被每个视图控制器继承。

既然知道了下次向用户显示视图时，会重新加载，那就可以安全地释放每个控制器。可以在现有的 `didReceiveMemoryWarning` 方法中添加几行代码完成此任务：

```
- (void)didReceiveMemoryWarning { // 如果该视图没有父视图，则释放它
    [super didReceiveMemoryWarning];

    // 释放没有用到的缓存的数据、图片等
    if (!self.blueViewController.view.superview) {
        self.blueViewController = nil;
    } else {
        self.yellowViewController = nil;
    }
}
```

新添加的代码用于检查当前向用户显示的是哪个视图，并释放另一个视图的控制器（通过将其属性设置为 `nil` 来实现）。这样会释放控制器及其视图，从而释放它们占用的内存。

---

**提示** 延迟加载是 iOS 中一个关键的资源管理组件，应该尽可能在合适的地方使用。在复杂的多视图应用中，负责任地从内存中移除未使用的对象，可以保证应用顺利运行，以免应用因为内存不足而间歇崩溃。

---

### 6.3.7 实现内容视图

我们在此应用中创建的两个内容视图都极其简单。每个内容视图都包含一个操作方法，这个操作方法由一个按钮触发，而且两个视图都不需要输出接口。实际上，这两个视图太相似了，它们甚至可以用同一个类来表示。我们选择用两个不同的类来表示它们，是因为大多数多视图应用就是这样构造的。

我们接下来要实现的两个操作方法仅仅是用来显示警告视图（就像第 4 章的 Control Fun 应用中所做的一样），在 `BIDBlueViewController.m` 中继续添加以下代码：

```
#import "BIDBlueViewController.h"

@implementation BIDBlueViewController

- (IBAction)blueButtonPressed {
    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];
}
...
```

保存代码，然后切换到 BIDYellowViewController.m，添加以下非常相似的代码：

```
#import "BIDYellowViewController.h"

@implementation BIDYellowViewController

- (IBAction)yellowButtonPressed {
    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];
}
}
```

保存代码。

接下来选中 Main.storyboard，在界面构建器中打开它，以进行一些更改。首先，需要为 BIDBlueViewController 添加一个新的场景。到目前为止，我们所使用的每个分镜都只包含一个视图控制器和一个视图，但分镜的能力远远不止这些，其中就包括了管理多个场景。从对象库中再拖出一个 View Controller 对象并放在编辑区中已有的视图控制器旁边。现在你的分镜中包含了两个场景，每一个都可以在应用运行时各自动态进行加载。单击新场景下面那行图标中的 View Controller 图标并按下 option+ command+3 打开身份检查器。在 Custom Class 区域中，Class 默认为 UIViewController，请将其更改为 BIDBlueViewController。

我们还需要为这个新的视图控制器创建一个标识符，这样代码就可以在分镜中找到它了。在身份检查器的 Custom Class 标题下方，你会看到一个 Storyboard ID 文本框。点中并在里面输入 Blue，与我们代码中的内容完全一样。

现在你拥有两个场景。我们在之前向你展示了如何让应用在启动时加载这个分镜，但却并没有任何提到关于场景的信息。应用如何知道应该显示两个视图中的哪一个呢？答案就是图 6-18 中指向第一个场景的大箭头。箭头指向了分镜的默认场景，即应用最开始显示的内容。如果你想要选择另一个默认场景，只需将箭头拖到想要指向的场景上。

单击刚刚添加的新场景上面全屏的矩形视图，然后按 option+command+4 调出属性检查器。

在属性检查器的 View 部分，单击标有 Background 的颜色选项，使用弹出的颜色选取器将此视图的背景颜色改为蓝色。如果你对自己选择的蓝色满意，就关闭颜色选取器。

从库中拖出一个 Button 放到视图上，通过蓝色引导线让按钮在视图的水平方向和垂直方向上都居中对齐。我们想要按钮始终位于中心，因此要创建两个约束来达到效果。首先在菜单中选择 Editor>Align>Horizontal Center in Container 选项。然后再次选中按钮，在菜单中选择 Editor>Align>Horizontal Center in Container 选项。

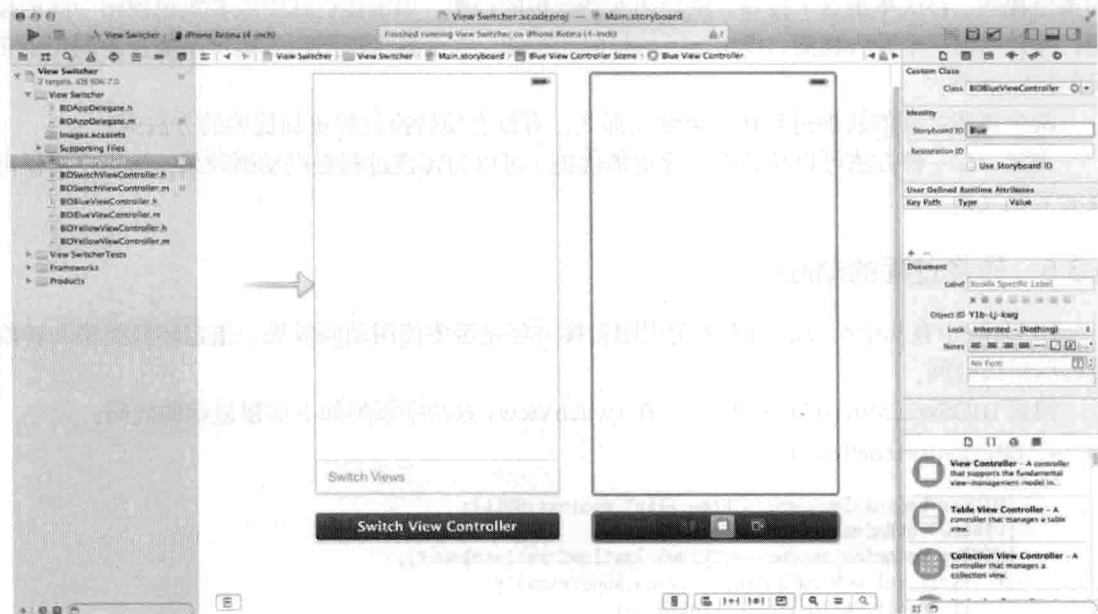


图 6-18 我们在分镜中添加了一个场景。大箭头指向了默认场景

双击这个按钮，将其标题改为 Press Me。然后，选中按钮的同时切换到关联检查器（按下 option+command+6），将 Touch Up Inside 事件拖到视图控制器图标，并关联到 blueButtonPressed: 操作方法。你可能会注意到按钮的文字默认是蓝色的，而背景也是蓝色的。这样就会出现按钮的文字很难看清楚的问题。通过 option+command+4 快捷键切换到属性检查器，并使用颜色选取器弹出按钮来更改文字为其他颜色。根据背景颜色的深浅程度，你可以选择白色或黑色。

之后我们要对 BIDYellowViewController 进行几乎一样的更改。从对象库中再拖出一个 View Controller 控制器并放在编辑区里。不必担心界面元素会不会过于密集，你可以把场景盖在另一个上面，这不会有影响的。点击 dock 栏中的 View Controller 图标并使用身份检查器将它的 class 属性更改为 BIDYellowViewController 并设定其 Storyboard ID 为 Yellow。

接着，选择视图并切换到属性检查器。在属性检查器中，单击 Background 颜色选项并选择亮黄色，然后关闭颜色选取器。

接下来，从库中拖出一个 Button 按钮并通过引导线将其添加到此视图的中心位置。通过使

用菜单来创建水平和垂直中心对齐的约束（就像之前的按钮那样）。并将其标题改为 **Press Me, Too**。选中这个按钮，然后使用关联检查器将按钮的 **Touch Up Inside** 事件拖动到视图控制器的图标上，并关联 **yellowButtonPressed** 操作方法。

完成之后，保存分镜，现在可以试着运行这个应用了。点击 Xcode 上的运行按钮，应用启动后将会显示出整个屏幕的蓝色。

应用启动时，将显示我们构建的蓝色视图。单击 **Switch Views** 按钮时，会切换到我们所构建的黄色视图。再次单击这个按钮，就会切回到蓝色的视图。单击蓝色视图或者黄色视图中间的按钮，都会弹出一个警告视图，指出按下的是哪个按钮。这个警告视图用于提示当前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？

当然，有一种方法可以让转换过程更加优美。可以为转换过程提供动画效果，为用户提供可视的变化反馈。

### 6.3.8 转换过程的动画效果

UIView 中有几个类方法用于指定视图转换过程是否要使用动画效果，指定转换类型和转换过程的持续时间。

回到 **BIDSwitchViewController.m**，在 **switchViews:** 方法中添加如下加粗显示的代码：

```
- (IBAction)switchViews:(id)sender
{
    [UIView beginAnimations:@"View Flip" context:NULL];
    [UIView setAnimationDuration:0.4];
    [UIView setAnimationCurve:UIViewAnimationCurveEaseInOut];
    if (!self.yellowViewController.view.superview) {
        if (!self.yellowViewController) {
            self.yellowViewController = [self.storyboard
                instantiateViewControllerWithIdentifier:@"Yellow"];
        }
        [UIView setAnimationTransition:UIViewAnimationTransitionFlipFromRight
            forView:self.view cache:YES];
        [self.blueViewController.view removeFromSuperview];
        [self.view insertSubview:self.yellowViewController.view atIndex:0];
    } else {
        if (!self.blueViewController) {
            self.blueViewController = [self.storyboard
                instantiateViewControllerWithIdentifier:@"Blue"];
        }
        [UIView setAnimationTransition:UIViewAnimationTransitionFlipFromLeft
            forView:self.view cache:YES];
        [self.yellowViewController.view removeFromSuperview];
        [self.view insertSubview:self.blueViewController.view atIndex:0];
    }
    [UIView commitAnimations];
}
```

编译这个新版本并运行应用。单击 **Switch Views** 按钮之后，新视图将会以翻页的形式显示出

来，而不只是简单地忽然出现，如图 6-19 所示。

为了告诉 iOS 我们希望在转换过程中使用动画效果，需要声明一个动画块（animation block）并指定动画的持续时间。动画块使用 UIView 的类方法 `beginAnimations:context:` 声明，例如：

```
[UIView beginAnimations:@"View Flip" context:NULL];
[UIView setAnimationDuration:0.4];
```

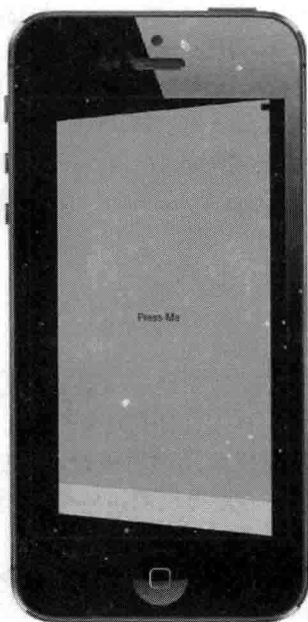


图 6-19 使用翻页动画，从一个视图转换到另一个视图

`beginAnimations:context:` 接受两个参数，第一个是动画块标题。只有在更直接地使用 Core Animation（即用于实现动画的框架）时才会用到这个标题。对于我们这个示例而言，可以使用 `nil` 作为这个参数的值。第二个参数是一个（`void *`）指针，指向关联到这个动画块的对象（或者任何 C 语言数据类型）。这里使用 `NULL`，因为没有必要指定对象。我们还设定了动画的持续时间，它告诉了 UIView 动画应该运行多长时间（以秒为单位）。

然后，设置动画曲线（animation curve），这决定了动画的运行速度。默认情况下，动画曲线是一条线性曲线，使动画匀速运行。此处设置的选项（`UIViewAnimationCurveEaseInOut`）指定动画以较慢的速度开始，中间加速，然后慢慢停止。这样动画看起来会更加自然，不那么呆板。

```
[UIView setAnimationCurve:UIViewAnimationCurveEaseInOut];
```

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

- ☐ `UIViewAnimationTransitionFlipFromLeft`
- ☐ `UIViewAnimationTransitionFlipFromRight`
- ☐ `UIViewAnimationTransitionCurlUp`

❑ `UIViewAnimationTransitionCurlDown`

根据要显示的视图选择使用不同的效果。为一种转换使用左侧翻入，为另一种转换使用右侧翻入，这样可以让人感觉视图在不断地前后翻动。

`cache` 选项会在动画开始时生成一个快照，并在动画执行过程的每一步使用这个快照，而不是重新绘制视图。应该始终对动画进行缓存，除非视图外观在动画执行过程中需要改变。

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

然后从控制器的视图中移除当前显示的视图，添加另一个视图。

指定了要制作成动画的所有更改之后，在 `UIView` 上调用 `commitAnimations` 方法。从动画块开始一直到调用 `commitAnimations` 方法之间的所有动作都会被制成动画。

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

## 6.4 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，现在，你应该充分理解了多视图应用的组织结构。

尽管 Xcode 包含了很多常用的多视图应用项目模板，但仍然需要理解这些应用的整体结构，这样才能从零开始构建自己的应用。Xcode 提供的这些项目模板能够节省大量时间，但有时，它们并不能满足要求。

接下来几章将继续构建多视图应用，强化本章介绍的概念，让你可以更好地理解复杂应用的组织结构。第7章将构造一个分页栏应用，准备开始吧！

上一章构建了第一个多视图应用程序。本章将构建一个完整的分页栏应用程序，它包含 5 个不同的分页和 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 个内容视图展示选取器视图（picker view，简称选取器）的用法，这是一种还未介绍过的 iOS 控件。

你可能还不熟悉这个名字，但只要用过 iPhone 或 iPod touch 10 分钟以上，就应该使用过选取器了。选取器是带有能够旋转的刻度盘的控件。它们用于在日历（Calendar）应用程序中输入日期，或者在时钟（Clock）应用程序中设置计时器（参见图 7-1）。在 iPad 上，选取器视图不太常用，因为 iPad 屏幕较大，你可以用其他方式在多个项中作出选择，但即便是 iPad 中的日历应用程序，也使用了选取器。

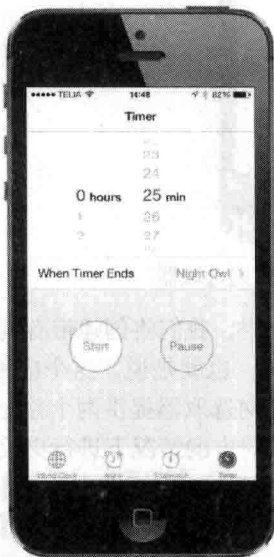


图 7-1 Clock 应用程序中的选取器

选取器是本书目前为止介绍的最复杂的 iOS 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 应用程序包含一个分页栏。构建该应用程序时，我们将更改默认的分页栏，使它包含 5 个分页，向每个分页项添加一个图标，然后创建一系列内容视图，并将每个视图连接到一个分页。

应用程序的内容视图将有 5 种不同的选取器。

- ❑ **日期选取器**：将要构建的第一个内容视图包含一个日期选取器，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，单击该按钮时，视图中将弹出一个警告视图，用于显示选取的日期。
- ❑ **单滚轮选取器**：第二个分页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。

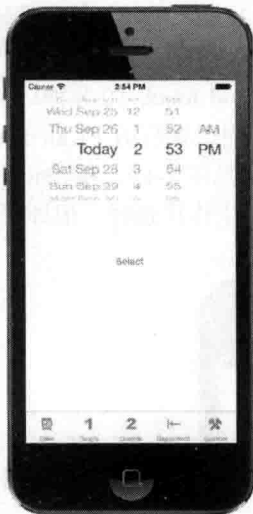


图7-2 第一个分页显示日期选取器

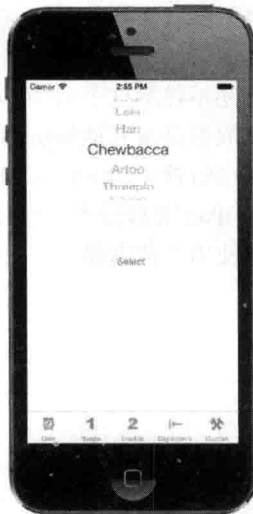


图7-3 这个选取器显示了一组值

- ❑ **多滚轮选取器**：在第三个分页中，我们将创建带有两个独立滚轮的选取器。从技术上说，每个滚轮都是一个选取器滚轮，也就是说，这个应用将创建带有两个滚轮的选取器。你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）。此选取器的每个滚轮都可以在不影响对方的情况下进行更改。
- ❑ **包含依赖滚轮的选取器**：在第四个内容视图中，我们将创建另一个带有两个滚轮的选取器。但是这一次，右侧滚轮中显示的值将根据左侧滚轮中选定值的变化而变化。在本示例中，左侧滚轮中将显示一组州，右侧滚轮中将显示该州的邮政编码（参见图 7-5）。
- ❑ **包含图像的自定义选取器**：最后，创建同样重要的第五个内容视图。你将了解如何将图像数据添加到选取器，并编写一个小游戏进行演示，该游戏使用带有 5 个滚轮的选取器。苹果公司的文档在描述选取器的外观时，多次提到它类似于老虎机。那么，还有比“小

型老虎机”更适合的例子吗（参见图 7-6）？对于这个选取器，用户无法手动更改滚轮的值，但是能够按 Spin 按钮使 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连续 3 个完全一样的图片，用户就获胜。

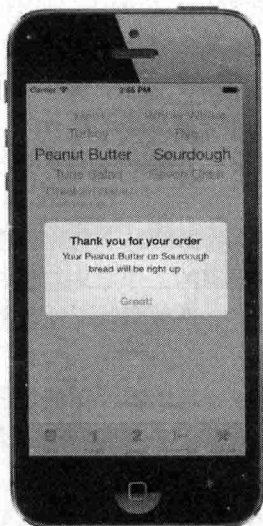


图7-4 包含两个滚轮的选取器，警告视图显示了所选的值



图7-5 在这个选取器中，一个滚轮的内容受另一个滚轮的影响。在左侧滚轮中选定一个州时，右侧滚轮将显示该州的邮政编码列表



图 7-6 第五个选取器。但我们并不赞成你使用 iPhone 来进行赌博

## 7.2 委托和数据源

在开始构建应用程序之前,先看一下为什么选取器比之前使用的控件都复杂。要使用选取器,仅仅从对象库中拖出一个选取器并放在内容视图上然后进行配置是不行的(日期选取器例外)。除此之外,还需要为选取器提供选取器委托和选取器数据源。

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单委托,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,确定要为每个滚轮中的每一行绘制的实际内容。选取器要求委托在特定滚轮上的特定位置绘制一个字符串或一个视图。选取器从委托获取数据。

除了委托之外,选取器还需要包含一个数据源。在本例中,数据源这个名称可能有点词不达意。选取器通过数据源获知滚轮数量和每个滚轮中的行数。数据源的工作原理与委托类似,它的方法会在预先指定的某些时刻被调用。如果未指定数据源和委托,选取器就无法工作,甚至无法绘制出来。

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视图的视图控制器,本章的应用程序将采用这种方法。每个内容面板的视图控制器都将是其选取器的数据源和委托。

---

**注意** 很多人存在这样的疑问:选取器数据源是应用程序的模型、视图或者控制器的某一部分吗?这个问题很难回答。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。数据源通常并不是用于保存数据的对象。虽然在简单的应用程序中,数据源可能用于保存数据,但它真正的作用是从模型中检索数据,并传递给选取器。

---

打开 Xcode,开始构建我们的应用程序。

## 7.3 创建 Pickers 应用程序

虽然 Xcode 为分页栏应用程序提供了模板,但是我们将从头开始构建这个应用程序。这并不需要太多额外的工作,而且是一个很好的实践机会。

创建一个新项目,再次选择 Empty Application 模板,点击 Next 进入下一个界面。在 Product Name 字段键入 Pickers。确保 Use Core Data 复选框未选中,并且把 Devices 弹出菜单设置为 iPhone。再次点击 Next, Xcode 会要求你选择一个文件夹用于保存项目。

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,你想在不看书内容的情况下尝试自己进行挑战,很好,大胆去试试吧!如果遇到难以解决的问题,随时可以翻书。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 创建视图控制器

上一章创建了一个根控制器(“根视图控制器”的简称)来管理切换应用程序其他视图的过

程。这次仍然需要使用根控制器来控件视图的切换，但是不需要自己创建。因为苹果公司提供了一个非常不错的类来管理分页栏视图，所以我们直接使用 `UITabBarController` 实例作为根控制器。

首先，需要在 Xcode 中创建 5 个新类，根控制器会控制这 5 个视图控制器之间的切换。

在项目导航面板中展开 Pickers 文件夹。这里有 Xcode 为项目创建的初始代码文件。单击 Pickers 文件夹，按 `command+N` 或者选择 `File>New>File...`。

在新文件向导的左侧面板中选择 Cocoa Touch，然后选择 Objective-C class 图标并点击 Next 继续。在向导下一页的 Class 字段输入 `BIDDatePickerViewController`。在为类文件命名时要仔细检查拼写，拼写错误会导致生成新类的类名不正确。还会看到一个控件，让你选择或输入新类的父类，确保这里的值是 `UIViewController`。再往下，可以看到一组 Targeted for iPad 和 With XIB for user interface 的复选框，单击 Next 之前请确保两者都取消了勾选。

最后，会看到一个文件夹选择窗口，可以选择一个位置保存新建的类。选择 Pickers 目录，这个目录中已经包含了 `BIDAppDelegate` 类和其他一些文件。确保在 Group 中选择了 Pickers 文件夹，并且在 Targets 中勾选了 Pickers。

单击 Create 按钮之后，Pickers 文件夹中将出现 2 个新文件：`BIDDatePickerViewController.h` 和 `BIDDatePickerViewController.m`。

重复上述步骤四次，把新类分别命名为 `BIDSingleComponentPickerViewController`、`BIDDoubleComponentPickerViewController`、`BIDDependentComponentPickerViewController`、`BIDCustomPickerViewController`。全部完成之后，Pickers 文件夹中就会包含所有的新文件，如图 7-7 所示。

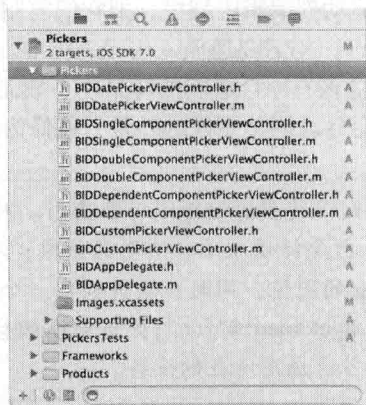


图 7-7 创建完 5 个视图控制器类之后，项目导航面板中应该包含这些文件

### 7.3.2 添加分镜

我们将在界面构建器中创建根视图控制器，它是一个 `UITabBarController` 实例。首先按下 `command+N` 来创建一个新的分镜，你会在文件创建面板的 User Interface 分区找到它。将它的 Devices 选项设定为 iPhone，将其命名为 `Main.storyboard`，并确保勾选了把它添加到 Pickers 目标

的复选框。

现在创建好了一个分镜，我们需要告诉 Xcode 在应用程序启动时的 GUI 由它定义。因此点击项目导航器最上面的 Pickers 文件夹，并在编辑区域中切换到 General 分页项，在 Deployment Info 位置的 Main Interface 下拉列表中选择 Main.storyboard。

因为我们使用 Empty Application 模板创建的这个项目，所以构建时不包含分镜，而是在应用程序委托的代码中创建了一个基本的空视图。我们接下来就是删除这些代码以改用分镜来加载。选择项目导航器中的 BIDAppDelegate.m 文件并将 application:didFinishLaunchingWithOptions: 方法中除最后一行以外的代码全部删掉：

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen]
                                                    bounds]];
    // 应用启动之后的一些自定义设置
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

分页栏可以使用图标来表示每个分页，因此在编辑分镜之前，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 Pickers/Tab Bar Icons/文件夹中。在 Xcode 的项目导航器中，选择已经包含了一个默认图标和启动图片的 Image.xcassets。然后将 Finder 文件夹中的 5 个图标全部拖动到编辑区域里，把它们复制到项目中。

如果你想要使用自定义的图标，则需要遵循一些相关标准。图标大小应该是 24 像素×24 像素，以.png 格式保存。图标文件应该有一个透明的背景。一般而言，中灰色图标最适合分页栏。不需要担心分页栏的外观。iOS 会自动调整图像的外观，就像为应用程序设置图标时一样。

---

**提示** 实际上，24 像素×24 像素大小的图标供普通屏幕使用。对于 iPhone 4 和 new iPad 及后续版本的 Retina 屏幕来说，需要提供双倍分辨率的图像，不然就会出现马赛克。这非常简单，对于一个名为 foo.png 的图像，同时应该再提供一个名为 foo@2x.png 的分辨率加倍的版本。调用 [UIImage imageNamed:@"foo"] 会自动返回标准大小的图像或者是分辨率加倍的图片，以更好地适应当前应用所在的设备。

---

### 7.3.3 创建分页栏控制器

现在开始创建分页栏控制器。回到 Main.storyboard 界面并从对象库中拖出一个 Tab Bar Controller 对象并放到编辑区域中（参考图 7-8）。

在拖动时你会发现，与我们之前让你从对象库中拖出的其他控制器不同，这一次其实拖出了三个完整的视图控制器以及视图，它们彼此通过曲线连接起来。实际上这里不仅仅是一个分页栏控制器，还有两个已经关联好的子控制器。

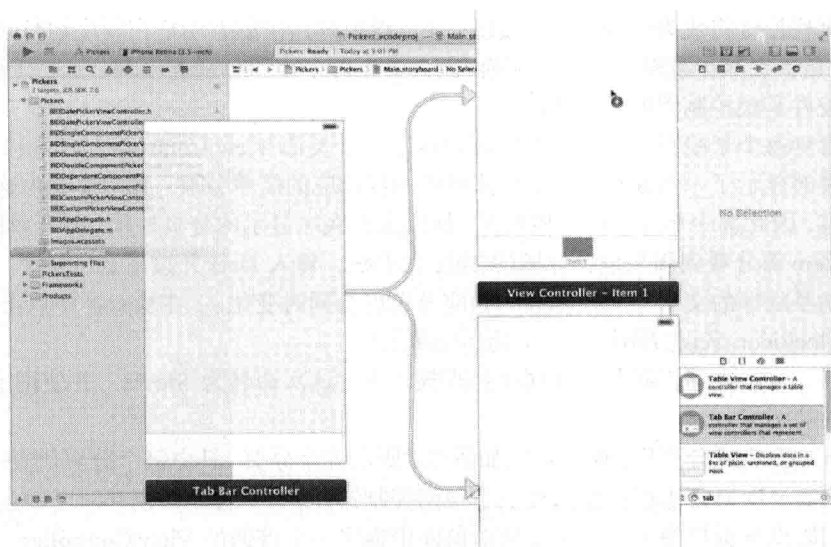


图 7-8 从库中拖出一个 Tab Bar Controller 放到编辑区中。你拖出来的东西块头真大啊

将分页栏控制器放到编辑区域中之后，你能看到分镜中所有场景的详细情况（参见图 7-9）。你会发现连接着分页栏控制器和子控制器的曲线依然还在。你可以任意地移动场景，这些线会自动进行调整并保持连接。分镜中的场景在屏幕上的位置与应用运行时所显示的没有关系。

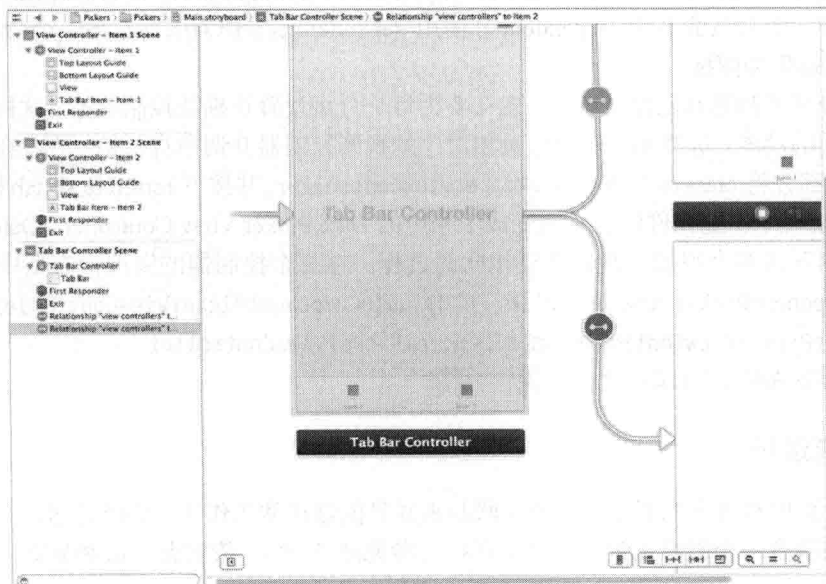


图 7-9 分页栏控制器的场景和两个子场景。请注意视图底部的分页栏，它包含了两个独立的分页。以及连接每个子视图控制器的曲线

这个分页栏控件将成为根控制器。回想一下，根控制器控制着用户在程序运行时看到的一个视图。同时负责在其他视图之间进行切换。因为要把每个视图分别连接到分页栏上的一个分页，所以分页栏控件是根控制器的合理选择。

你可以看到每个子视图控制器都在底部显示了一个类似 View Controller - Item 1 的名字，并且都在视图底部显示了一条横栏，以及与分页栏内容相应的简单分页。我们需要为这两个名字设置正确的内容，因此选中 Item 1 视图控制器，然后点击底端显示的分页栏项。打开属性检查器后，你会在 Bar Item 部分看到用来设置标题的 Title 文本框。输入 Date 并按下 return 键。这样能立即更改视图控制器底部的文本（父控制器的相应分页也会同时变化）。在检查器中点击 Image 弹出菜单并选择 clockicon 来设置图标。这不能再简单了！

现在对第二个子视图控制器重复同样的步骤，不过这次命名为 Single，并使用 singleicon 图像作为分页项的图标。

我们接下来要完成分页栏，使其对应如图 7-2 所示五个分页。其中每个分页代表一个选取器。我们的办法只需要向分镜中再添加另外三个视图控制器（分页栏控制器创建时已经包含了两个），然后连接它们以供分页栏使用。首先要从对象库中拖出一个普通的 View Controller。接下来按住 control 键从分页栏控制器拖动到你的新视图控制器上后松开鼠标，并在弹出的面板中选择 view controllers。这样就会告诉分页栏控制器它还有一个新的子控制器需要管理，分页栏会立即出现一个新的分页，而且新控制器的视图底部也会出现一条横栏（和之前所见的一样）。最后对新的视图控制器执行和之前一样的步骤，将分页的标题改为 Double，并使用 doubleicon 图像作为图标。

我们现在已经开始习惯了。像之前那样再次拖出两个视图控制器并将它们连接到分页栏控制器上。为其中一个分页命名为 Dependent 并使用 dependenticon 作为图标，另一个命名为 Custom 并使用 toolicon 作为图标。

所有的分页控制器都已经就位了，现在要为每个分页设置正确的控制器类。这样每一个视图都会执行不同的功能。选择和最左边分页相对应的视图控制器并调出身份检查器。在检查器中的 Custom Class 部分将 class 改为 BIDDatePickerViewController，并按下 return 键或 tab 键提交设置。你将看到 dock 栏中所选控件的名字也变成了相应的 Date Picker View Controller - Date。

现在对剩下的四个视图控制器重复相同的过程。在每个控制器的属性检查器中，分别输入 BIDSingleComponentPickerViewController、BIDDoubleComponentPickerViewController、BIDDependentComponentPickerViewController 和 BIDCustomPickerViewController。

在继续接下来的 GUI 编辑之前，保存分镜。

### 7.3.4 初次运行

现在，分页栏和内容视图都已经被关联起来并且能够正常工作了。编译并运行，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（参见图 7-10）。依次点击这些分页，每一个分页应该都可点击。

现在，内容视图中还没有任何内容，因此，分页的切换其实并不明显。事实上除了分页按钮



上的高光之外完全没有任何区别。但是如果所有组成部分都运行良好,那么这个多视图应用程序的基本框架现在就已经建立起来了,而且能够运行了。接下来开始设置每个独立的内容视图。

**提示** 如果在单击某个分页时,模拟器出了问题,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。返回去确保所有的关联和类名都正确设置了。



图 7-10 应用程序拥有 5 个空白的分页,但是每个分页都是可点击的

如果想进一步确保所有元素都能够正常工作,可以为每个内容视图添加一个分页或者其他对象,然后重新运行应用程序。如果所有元素都运行良好,点击不同分页时会看到不同的视图内容。

## 7.4 实现日期选取器

要实现日期选取器,需要一个输出接口和一个操作方法。输出接口获取日期选取器的值。操作方法由一个按钮触发并弹出一个警告视图,用于显示在选取器中选择的日期。我们将在界面构建器中编辑时添加,如果当前编辑区域中的不是 `Main.storyboard`,请在项目导航器中选中它。

我们要做的第一件事是在库中找到 `Date Picker` 并将其拖动到编辑区中的 `Date Picker View Controller` 上。把日期选取器放在视图的顶端,直到贴住屏幕的上面。不必担心它会遮住状态栏,因为在 iOS 7 中,这个控件顶端的垂直部分有一些没人留意的固定空白。选取器会占去整个内容视图的宽度和一部分高度(参见图 7-11)。

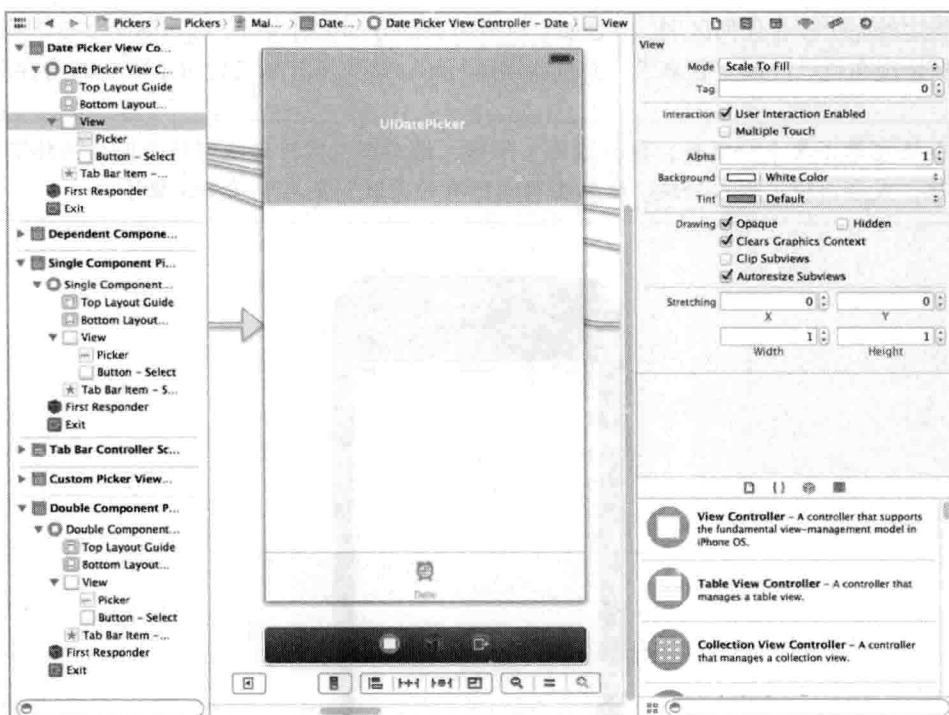


图 7-11 从库中拖出一个 Date Picker。它会占据整个视图宽度。我们将它放在了视图顶部，并盖住状态栏

单击以选中这个日期选取器，回到属性检查器。从图 7-12 中可以看到，日期选取器有许多属性可以配置。我们将保留大部分的默认设置(本章结束之后，可以随意更改这些选项查看效果)。我们要做的一件事是，将选取器限定在合理日期范围内。找到名为 Constraints 的项，选中 Minimum Date 复选框，保留默认值 1/1/1970。还要选中 Maximum Date 复选框，并将其设置为 12/31/2200。

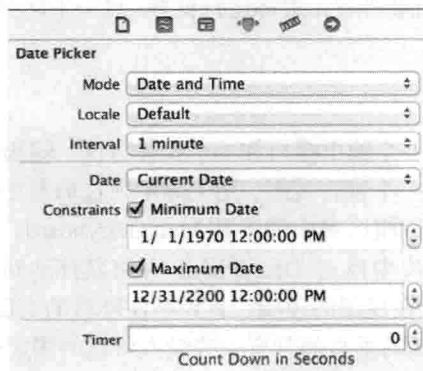


图 7-12 日期选取器的属性检查器，设置 maximum date，其余各项保留默认值

我们现在要将选取器连接到它的控制器上。请按下 `option+command+return` 打开辅助编辑器并确保辅助编辑器上方的跳转栏上的是 Automatic。这样就会显示 `BIDDatePickerViewController.m`。接下来请按住 `control` 键从选取器拖到 `BIDDatePickerViewController.m` 上方 `@interface` 和 `@end` 语句之间的位置并在 Insert Outlet, Action, or Outlet Collection 气泡提示出现时松开鼠标。确保弹出窗口中的 Connection 值为 Outlet, 并在 Name 文本框中输入名字 `datePicker`, 然后按下 `return` 以创建输出接口并关联到选取器上。

接下来, 从库中拖出一个 Button 按钮, 并放到日期选取器下方。双击按钮, 将其标题设置为 Select。现在按住 `control` 键从按钮拖辅助编辑器的源代码中, 这次要拖到底端的 `@end` 语句上面, 看到 Insert Action 这样的气泡提示就可以松开了。给新的操作方法命名为 `buttonPressed` 并按下 `return` 键进行关联。这样会创建一个名字为 `buttonPressed:` 的空方法, 你可以编写以下代码来实现它:

```
- (IBAction)buttonPressed:(id)sender {
    NSDate *selected = [self.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:@"That's so true!"
        otherButtonTitles:nil];
    [alert show];
}
```

然后, 向 `viewDidLoad:` 方法中添加几行代码来完成这个控制器类:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view.
    NSDate *now = [NSDate date];
    [self.datePicker setDate:now animated:NO];
}
```

这里, 首先添加了 `buttonPressed` 方法的实现, 并且覆盖了 `viewDidLoad` 方法。在 `buttonPressed` 方法中, 使用 `datePicker` 输出接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, 最后把这个字符串显示在警告视图中。

在 `viewDidLoad` 中, 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` 设置为这个日期, 这可以确保每次从分镜中加载此视图时, 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, 看看选取器是否可以正常工作。如果一切都进展顺利, 那么应用程序应该与图 7-2 类似。如果单击 Select 按钮, 会弹出一个警告视图, 显示当前在日期选取器中选定的日期和时间。

**注意** 日期选取器不允许指定秒或时区。警告视图上显示的格林威治标准时间（GMT）是带有秒的。我们可以添加一些代码来简化警告中显示的字符串，但由于本章篇幅限制，这里不再赘述这方面内容。如果有兴趣自己定制日期格式，可以看一下 `NSDateFormatter` 类。

## 7.5 实现单滚轮选取器

接下来看一下如何使用选取器让用户从一系列值中进行选择。在这个例子中，我们将创建一个 `NSArray` 来保存显示在选取器中的值。

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（实际上，它们通常是同一个对象）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，甚至通过动态组合或计算得到。

因为选取器会向它的控制器请求数据，我们必须确保控制器正确地实现了方法。其中一环是在控制器的接口中声明将要实现的两个协议。请在项目导航面板中，单击 `BIDSingleComponentPickerViewController.h`。此控制器类同时充当选取器的数据源和委托，因此我们需要确保它遵循这两个角色的协议。添加以下代码：

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

@interface BIDSingleComponentPickerViewController : UIViewController
    <UIPickerViewDelegate, UIPickerViewDataSource>

@end
```

### 7.5.1 构建视图

再次点击 `Main.storyboard`，这次编辑的是分页栏中第二个分页的内容视图。在文件大纲中，点击 `Single Component Picker View` 场景。假如你看不到里面的内容，请点击展开三角，并选中小的黄色图标以调出视图。接下来，从库中拖出一个 `Picker View`（参见图 7-13），将其添加到视图中，放置在靠近视图顶部的位置，就像之前操作日期选取器视图一样。

现在我们将选取器关联到它的控制器上。此处的步骤和之前的相似：打开辅助编辑器，设置跳转栏以显示 `.m` 文件，按住 `control` 键从选取器拖到 `BIDSingleComponentPickerViewController.m` 的顶部 `@interface` 部分，以创建一个名为 `singlePicker` 的输出接口。

接下来选中选取器，按下 `option+command+6` 打开关联检查器。如果查看选取器视图的可用关联，将会看到前两项是 `dataSource` 和 `delegate`。如果没看到这些输出接口，再检查一下，确保选中的是选取器而不是包含选取器的 `UIView`。将 `dataSource` 旁边的圆圈拖到视图控制器图标上。然后把 `delegate` 旁边的圆圈也拖到视图控制器图标上。现在，选取器知道分镜中 `BIDSingleComponentPickerViewController` 类的实例就是它的数据源和委托，而选取器会要求这个实例提供数据进行显示。换句话说，当选取器需要用到将要显示的数据信息时，它将向控制此

视图的 `BIDSingleComponentPickerController` 实例请求相关信息。

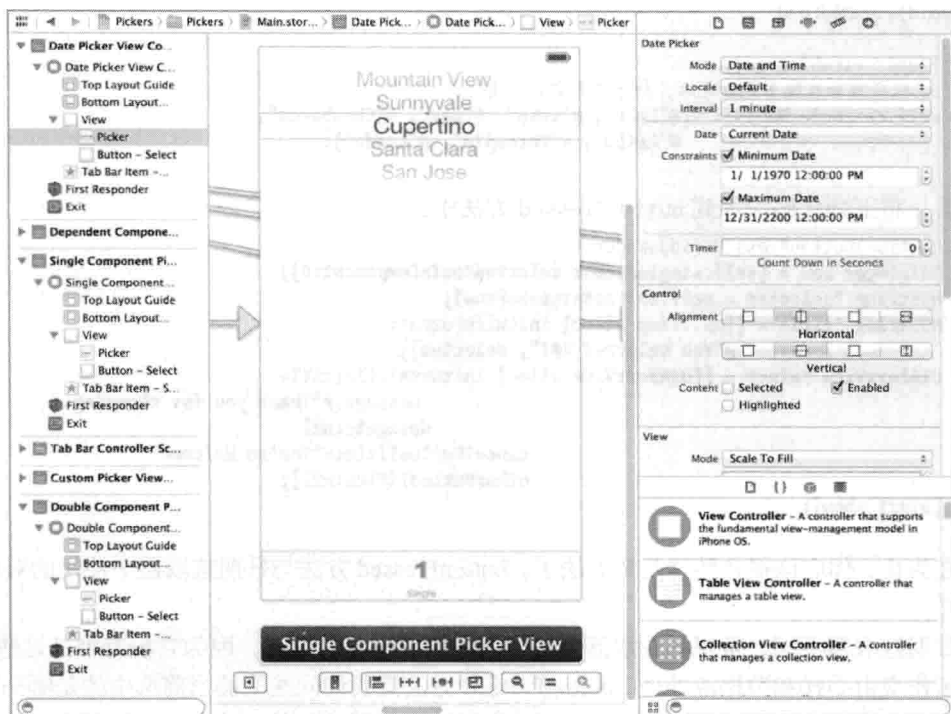


图 7-13 从库中拖出一个 Picker View 放到第二个视图上

把一个 Button 拖到这个视图上，双击该按钮，将其标题设置为 `Select`。按下 `return` 键提交更改。在关联检查器中，将 `Touch Up Inside` 旁边的圆圈拖到辅助编辑器的代码底端 `@end` 上并松开鼠标，以创建一个新的操作方法，将其命名为 `buttonPressed`，你会看到 Xcode 自动编写了一个空的方法。现在我们完成了 GUI 第二个分页的创作，保存并关闭分镜，然后继续编写代码。

## 7.5.2 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，先从你熟悉的代码开始，再添加一些你从未见过的新方法。

单击 `BIDSingleComponentPickerController.m`，并在文件开头的 `@interface` 部分添加以下属性变量。这样我们就有一个指向了某个数组的指针，它包含了几个著名的电影角色名字：

```
@interface BIDSingleComponentPickerController ()
```

```
@property (weak, nonatomic) IBOutlet UIPickerView *singlePicker;
```

```
@property (strong, nonatomic) NSArray *characterNames;
```

```
@end
```

接下来在 `viewDidLoad` 方法中添加这些初始化代码来设定角色数组的内容：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // 从分镜加载视图完成之后，执行其他额外设置
    self.characterNames = @[@"Luke", @"Leia", @"Han", @"Chewbacca",
                           @"Artoo", @"Threepio", @"Lando"];
}
```

然后，将下列代码添加到 `buttonPressed` 方法中：

```
- (IBAction)buttonPressed:(id)sender {
    NSInteger row = [self.singlePicker selectedRowInComponent:0];
    NSString *selected = self.characterNames[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];
}
```

现在为止，你应该很熟悉这两个方法了。`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。`buttonPressed:` 方法需要询问选取器当前选中的是哪一行，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 = [self.singlePicker selectedRowInComponent:0];
```

注意，需要指定想要询问的滚轮。此选取器中只有一个滚轮，因此我们传入 0，这是第一个滚轮的索引。

---

**注意** 注意到 `NSInteger` 与 `row` 之间没有星号了吗？在大部分 iOS SDK 中，虽然前缀 `NS` 通常表示来自 Foundation 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`。现在苹果已经发布了 64 位的 iOS 设备，这具有极大意义。也有可能为 iOS 应用程序编写的类打算将来能在面向 Mac OS X 的 Cocoa 应用程序中重复利用。Mac OS X 应用程序最近几年都能够同时运行在 32 位和 64 位机器上。

---

在 `viewDidLoad` 中，创建了一个包含几个对象的数组，用于向选取器提供数据，并将其赋给 `characterNames` 属性。通常而言，数据来自于其他数据源，比如项目的 Resources 文件夹 web 服务请求中的属性列表。利用此处的方式在代码中嵌入一系列元素，需要对这个列表进行更新或者把

应用程序转换为其他语言时就会比较麻烦。但出于演示目的，这种方法是将数据获取到数组中的最快捷、最简单的方式。但你通常不会采用这种方式创建数组，而是将使用的数据缓存到 `viewDidLoad` 方法的一个数组中，这样选取器请求数据时就不必每次都访问磁盘或网络。

**提示** 如果不打算像刚才在 `viewDidLoad` 中那样，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？可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程序 bug。你也可以为不同语言提供不同版本的列表，第 20 章将介绍这种方法。可以在 Xcode 中创建属性列表，Xcode 在新建文件向导页面的 `Resource` 部分提供了创建属性列表所需的模板，也支持在编辑器面板中编辑属性列表。`NSArray` 和 `NSDictionary` 都提供了一个名为 `initWithContentsOfFile:` 的方法，可以根据属性列表文件的内容对实例进行初始化，本章稍后在实现 `Dependent` 分页时将采用这种方法。

最后，将以下代码插入到文件末尾：

```
#pragma mark -
#pragma mark Picker Data Source Methods
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView
{
    return 1;
}

- (NSInteger)pickerView:(UIPickerView *)pickerView
numberOfRowsInComponent:(NSInteger)component
{
    return [self.characterNames count];
}

#pragma mark Picker Delegate Methods
- (NSString *)pickerView:(UIPickerView *)pickerView
titleForRow:(NSInteger)row
forComponent:(NSInteger)component
{
    return self.characterNames[row];
}

@end
```

文件底部包含实现选取器所需的新方法。前两个方法来自 `UIPickerViewDataSource` 协议，所有选取器（除了日期选取器）都必须实现这两个方法。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滚轮，这就是选取器会询问应该显示几个滚轮的原因。这一次只想显示一个列表，因此返回 1。注意，有一个 `UIPickerView` 作为参数传到这个方法中。这个



参数指向触发这个方法的选取器视图，这样一来，同一个数据源就能够控制多个选取器。在本例中，只有一个选取器，可以直接忽略这个参数，因为已经知道到底是哪个选取器在调用这个方法。

选取器使用第二个数据源方法询问给定的选取器滚轮应该包含多少行数据：

```
- (NSInteger)pickerView:(UIPickerView *)pickerView
numberOfRowsInComponent:(NSInteger)component
{
    return [self.characterNames count];
}
```

这次又可以通过参数知道当前询问的是哪个选取器视图，以及选择器询问的是哪个滚轮。因为我们只有一个选取器，而且只包含一件滚轮，所以并不需要关心这些参数，可以直接返回数组中的对象数量。

### #pragma 是什么

注意到 BIDSingComponentPickerViewController.m 中的如下代码了吗？

```
#pragma mark -
#pragma mark Picker Data Source Methods
```

从技术角度来说，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。在我们的例子中，#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，要在编辑器面板顶部的方法和函数弹出菜单中插入一条分隔线。第一个指令在菜单中添加了一条分隔线。第二条指令创建一个文本条目，其中包含该行剩余的内容，可以将该文本条目用作源代码中各组方法的某种描述性标题。

一些类（尤其是一些控制器类）可能很长，使用方法和函数弹出菜单便于代码导航。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。可选（optional）这个词带有一定的欺骗性，因为实际上可以一个委托方法都不实现。通常需要实现这里实现的这种方法。但是，如果想在选取器中显示除文本以外的内容，必须实现另一个方法，本章后面介绍自定义选取器时会讲述。

```
#pragma mark Picker Delegate Methods
- (NSString *)pickerView:(UIPickerView *)pickerView
titleForRow:(NSInteger)row
forComponent:(NSInteger)component
{
    return self.characterNames[row];
}
```

在此方法中，选取器要求提供指定滚轮中指定行的数据。参数提供了一个指向正在请求数据的选取器的指针，以及它请求的滚轮和行。由于我们的视图只有一个选取器，且该选取器只有一

个滚轮，因此可以忽略除 row 参数之外的其他参数，使用 row 参数作为索引返回数据数组中相应的元素。

再次编译并运行应用程序。当模拟器出现时，切换到第二个分页（名为 Single 的分页），并检查新的自定义选取器，它应该与图 7-3 类似。

回顾完《星球大战》(Star Wars) 之后，返回 Xcode，接下来讨论如何实现带有两个滚轮的选取器。如果想要自我挑战，那么第二个内容视图实际上是一个很好的尝试机会。你已经知道实现选取器需要的所有方法，可以试着自己实现。如果遇到问题可以随时再回到这里。你可能想先实现图 7-4 所示的优美界面，那么只需回顾刚才介绍的方法。然后继续阅读，一起来看看如何解决这个问题。

## 7.6 实现多滚轮取器

下一个内容面板将包含一个选取器，这个选取器带有两个相互独立的滚轮（或滚轮）。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，要编写的数据源方法和委托方法与为单个滚轮选取器编写的方法相同。我们只需在这些方法中编写少量额外的代码，以确保为每个滚轮返回正确的值和行数。

### 7.6.1 声明输出接口和操作方法

单击 BIDDoubleComponentPickerViewController.h，并添加以下代码：

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

@interface BIDDoubleComponentPickerViewController : UIViewController
<UIPickerViewDelegate, UIPickerViewDataSource>

@end
```

这里我们只是让控制器遵循了委托和数据源协议。保存代码并点击 Main.storyboard 以编辑 GUI。

### 7.6.2 构建视图

在视图中添加一个选取器和一个按钮，将按钮分页改为 Select，然后创建必要的关联。我们不再讨论关联过程，如果需要逐步指导，可以参考上一小节，因为这两个视图控制器在分镜中的关联方式都是一样的。下面简单说明一下需要做的工作。

- (1) 创建一个名为 doublePicker 输出接口并将视图控制器关联到选取器。
- (2) 使用关联检查器，将选取器视图上的 DataSource 和 Delegate 关联到视图控制器。
- (3) 使用关联检查器，将按钮的 Touch Up Inside 事件关联到视图控制器上名为 buttonPressed 的新操作方法。

确保保存并关闭了分镜，然后返回 Xcode。可以把这一页上折起来（也可以使用书签），以后用得着。稍后还需要参考这些内容。

### 7.6.3 实现控制器

选中 BIDDoubleComponentPickerViewController.m, 并在文件开头添加以下代码:

```
#import "BIDDoubleComponentPickerViewController.h"

#define kFillingComponent 0
#define kBreadComponent 1

@interface BIDDoubleComponentPickerViewController ()

@property (weak, nonatomic) IBOutlet UIPickerView *doublePicker;
@property (strong, nonatomic) NSArray *fillingTypes;
@property (strong, nonatomic) NSArray *breadTypes;

@end
```

如你所见, 我们一开始定义了代表各自选取器滚轮的两个常量, 这样能使代码更易于阅读。用数字来为滚轮赋值, 最左边的滚轮赋值为零, 向右数字依次加一。接下来我们声明了两个数组的属性变量, 用来包含两个选取器滚轮的数据内容。

现在来实现 buttonPressed: 按钮方法, 代码如下所示:

```
- (IBAction)buttonPressed:(id)sender {
    NSInteger fillingRow = [self.doublePicker selectedRowInComponent:
                             kFillingComponent];
    NSInteger breadRow = [self.doublePicker selectedRowInComponent:
                           kBreadComponent];

    NSString *filling = self.fillingTypes[fillingRow];
    NSString *bread = self.breadTypes[breadRow];
    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];
}
```

然后, 将以下代码添加到 viewDidLoad 方法中:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // 从分镜加载视图完成之后, 执行其他额外设置
    self.fillingTypes = @[@"Ham", @"Turkey", @"Peanut Butter", @"Tuna Salad",
                          @"Chicken Salad", @"Roast Beef", @"Vegemite"];
    self.breadTypes = @[@"White", @"Whole Wheat", @"Rye", @"Sourdough",
                        @"Seven Grain"];
}
```

最后，将委托方法和数据源方法添加到文件底部最后的@end 语句之上：

```
#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];
    } else {
        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[row];
    } else {
        return self.fillingTypes[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熟悉的。请求选中的行时，需要使用前面定义的常量 kBreadComponent 和 kFillingComponent 指定所请求的行所属的滚轮。

```
NSInteger fillingRow = [self.doublePicker selectedRowInComponent:
                        kFillingComponent];
NSInteger breadRow = [self.doublePicker selectedRowInComponent:
                      kBreadComponent];
```

可以看到，这里使用了两个常量来代替 0 和 1，使得代码具有非常好的可读性。后面使用的 buttonPressed 方法都与这个基本相同。

viewDidLoad: 也与前一个选取器中编写的版本非常类似。唯一的区别在于，载入了两个包含数据的数组，而不是一个。这次还是使用硬编码的字符串列表创建数组，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，代码与之前有一些不同。在第一个方法中，指定选取器应该拥有两个滚轮，而不是一个：

```
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView
{
    return 2;
}
```

这一次要求提供行数时，必须检查选取器正在询问的是哪个滚轮，然后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];
    } else {
        return [self.fillingTypes count];
    }
}
```

然后，在委托方法中也要进行同样的处理。检查滚轮，根据指定的滚轮返回相应数组中的正确数据。

```
- (NSString *)pickerView:(UIPickerView *)pickerView
titleForRow:(NSInteger)row
forComponent:(NSInteger)component
{
    if (component == kBreadComponent) {
        return self.breadTypes[row];
    } else {
        return self.fillingTypes[row];
    }
}
```

这不是很难，是吧？编译并运行应用程序，并确保 Double 分页的内容面板与图 7-4 类似。

注意，各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。这正适合本例的需求。但有时，一个滚轮依赖于另一个。日期选取器就是一个典型的例子。当更改月份时，显示每月天数的刻度盘可能需要更改，因为不是所有月份对应的天数都相同。只要知道基本的方法，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，所以接下来一起看看如何操作。

## 7.7 实现内容取决于滚轮

我们会逐渐加快学习速度。本节不再详细讨论前面已经介绍过的内容，主要介绍一些新知识。新选取器将在左侧滚轮中显示一个包含美国所有州名的列表，在右侧滚轮中显示左侧选定的州对应的邮政编码。

左侧滚轮中的每个项都需要一个独立的邮政编码列表。与上一节一样，我们将声明两个数组，分别对应每一个滚轮。还需要一个 `NSDictionary` 字典。在该字典中，每个州都有一个对应的 `NSArray`（参见图 7-14）。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，并将它分配给右侧滚轮所使用的数组。如果不是很明显，不要担心，深入分析代码时会说明这一点。

将以下代码添加到 `BIDDependentComponentPickerViewController.h` 文件中：

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

```
@interface BIDDependentComponentPickerViewController : UIViewController
```

```
<UIPickerViewDelegate, UIPickerViewDataSource>
```

```
@end
```

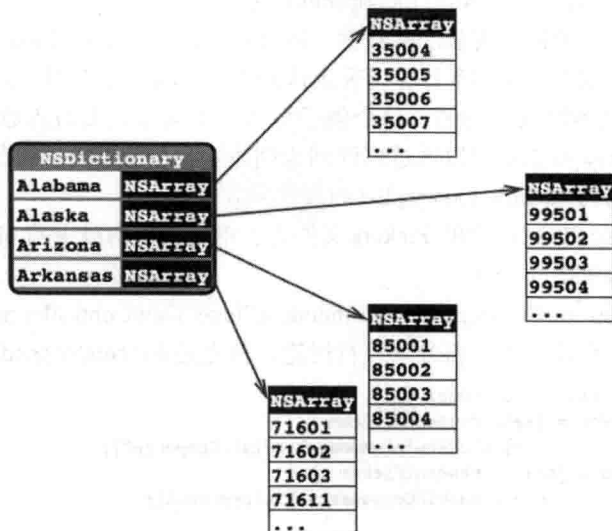


图 7-14 应用程序的数据：对于每个州，字典里都有与之对应的、使用州名作为键的条目，这个键对应的值是一个 NSArray 实例，其中包含该州的所有邮政编码

接下来将以下代码添加到 BIDDependentComponentPickerViewController.m 文件中：

```
#import "BIDDependentComponentPickerViewController.h"

#define kStateComponent 0
#define kZipComponent 1

@interface BIDDependentComponentPickerViewController ()

@property (strong, nonatomic) NSDictionary *stateZips;
@property (strong, nonatomic) NSArray *states;
@property (strong, nonatomic) NSArray *zips;

@end
```

现在开始构建内容视图。构建过程与构建前面两个滚轮视图的过程大体相同。如果忘记了具体操作，可以查看 7.5.1 节，并按照其中的步骤进行操作。这里要注意一点：首先应打开 Main.storyboard，找到 BIDDependentComponentPickerViewController 类的视图，然后重复本章对其他内容视图所做的基本操作。你还要将名称为 dependentPicker 的属性变量关联到选取器，一个空的 buttonPressed: 方法关联到按钮上，并将选取器的 delegate 和 dataSource 输出接口都关联到视图控制器上。完成之后，保存分镜。

接下来，深呼吸，马上开始实现这个控制器类。刚开始可能会觉得这个实现有点怪异。通过让一个滚轮依赖于另一个滚轮，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。此处使用的技巧可以简化这个过程。数据源方法看起来与为 DoublePicker 视图实现的方法几乎相同。增加的所有复杂度都在 viewDidLoad 和一个新的委托方法 pickerView:didSelectRow:inComponent: 之间处理。

在编写代码之前，需要创建要显示的数据。到目前为止，已经通过指定一系列字符串在代码中创建了一些数组。但是现在，我们不打算再采用这种方式，因为不希望输入数千个值，所以希望通过其他更合适的方式解决这个问题，这个例子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名为 statedictionary.plist 的属性列表文件。

将该文件复制到 Xcode 项目中的 Pickers 文件夹。单击项目窗口中的 plist 文件，可以查看甚至编辑其中的数据（参见图 7-15）。

现在编写一些代码。在 BIDDependentComponentPickerViewController.m 文件中，我们先会把完整的实现方法都展示出来，然后再分块进行讨论。首先是 buttonPressed: 方法的实现：

```
- (IBAction)buttonPressed:(id)sender {
    NSInteger stateRow = [self.dependentPicker
                          selectedRowInComponent:kStateComponent];
    NSInteger zipRow = [self.dependentPicker
                       selectedRowInComponent:kZipComponent];

    NSString *state = self.states[stateRow];
    NSString *zip = self.zips[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];
}
```

将以下代码添加到现有的 viewDidLoad 方法中：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // 从分镜加载视图完成之后，执行其他额外设置
    NSBundle *bundle = [NSBundle mainBundle];
    NSURL *plistURL = [bundle URLForResource:@"statedictionary"
                                             withExtension:@"plist"];

    self.stateZips = [NSDictionary
                     dictionaryWithContentsOfURL:plistURL];

    NSArray *allStates = [self.stateZips allKeys];
    NSArray *sortedStates = [allStates sortedArrayUsingSelector:
```

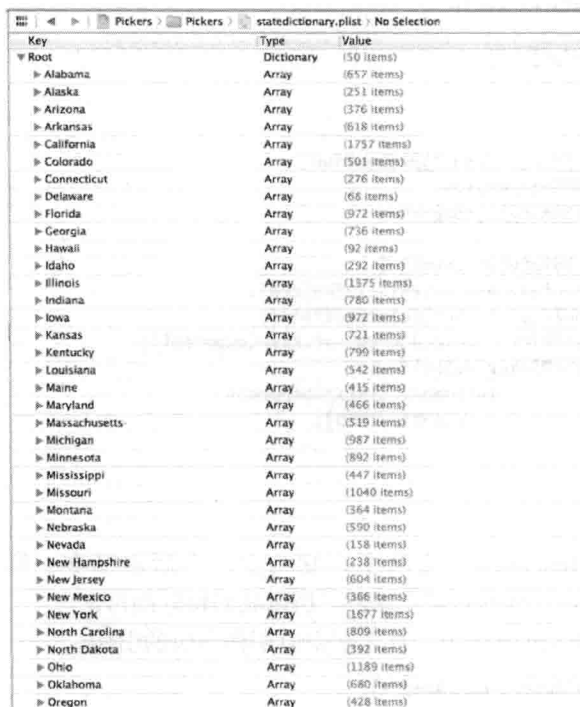


```

        @selector(compare:));
self.states = sortedStates;

NSString *selectedState = self.states[0];
self.zips = self.stateZips[selectedState];
}

```



Key	Type	Value
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	(66 items)
Florida	Array	(972 items)
Georgia	Array	(736 items)
Hawaii	Array	(92 items)
Idaho	Array	(292 items)
Illinois	Array	(1375 items)
Indiana	Array	(780 items)
Iowa	Array	(972 items)
Kansas	Array	(721 items)
Kentucky	Array	(799 items)
Louisiana	Array	(542 items)
Maine	Array	(415 items)
Maryland	Array	(466 items)
Massachusetts	Array	(519 items)
Michigan	Array	(987 items)
Minnesota	Array	(892 items)
Mississippi	Array	(447 items)
Missouri	Array	(1040 items)
Montana	Array	(364 items)
Nebraska	Array	(590 items)
Nevada	Array	(158 items)
New Hampshire	Array	(238 items)
New Jersey	Array	(604 items)
New Mexico	Array	(386 items)
New York	Array	(1677 items)
North Carolina	Array	(809 items)
North Dakota	Array	(392 items)
Ohio	Array	(1189 items)
Oklahoma	Array	(680 items)
Oregon	Array	(428 items)

图 7-15 statedictionary.plist 文件，显示一个列表，列出了美国所有的州名。

在 Hawaii（夏威夷）这项中可以看到一组邮政编码的开头

最后，将委托方法和数据源方法添加到文件底部：

```

#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];
    } else {
        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[row];
    } else {
        return self.zips[row];
    }
}

- (void)pickerView:(UIPickerView *)pickerView
  didSelectRow:(NSInteger)row
  inComponent:(NSInteger)component
{
    if (component == kStateComponent) {
        NSString *selectedState = self.states[row];
        self.zips = self.stateZips[selectedState];
        [self.dependentPicker reloadComponent:kZipComponent];
        [self.dependentPicker selectRow:0
                               inComponent:kZipComponent
                               animated:YES];
    }
}

@end

```

这里不再讨论 `buttonPressed` 方法了，它与上一个版本基本一样。但是，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，首先获得应用程序包的引用。

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

那么什么是包呢？包（bundle）只是一种特定类型的文件夹，其中的内容遵循特定的结构。应用程序和框架都是包，这个调用返回的包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` 文件夹中的资源。构建应用程序时，这些文件将被复制到应用程序的包中。我们已经在项目中添加过图像等资源，但是到现在为止，只是在界面构建器中使用它们。如果想在代码中使用这些资源，通常需要使用 `NSBundle`。我们将使用包获取所需资源的路径：

```
NSURL *plistURL = [bundle URLForResource:@"statedictionary"
                                       withExtension:@"plist"];
```

这将返回一个 `URL`，内容是 `statedictionary.plist` 文件的位置。然后可以根据这个 `URL` 创建一个 `NSDictionary` 对象。之后，属性列表的所有内容将被载入到新创建的 `NSDictionary` 对象中，然后将这个对象分配给 `stateZips`。

```
self.stateZips = [NSDictionary
                 dictionaryWithContentsOfURL:plistURL];;
```

刚才载入的字典使用州名作为键，键对应的值是一个 `NSArray`，其中包含所选州的邮政编码。

为了填充左侧滚轮的数组，我们从字典获取一个包含所有键的列表，并将这个列表赋给 `states` 数组。在赋值之前，先对这个列表按字母顺序进行排序。

```
NSArray *allStates = [self.stateZips allKeys];
NSArray *sortedStates = [allStates sortedArrayUsingSelector:
    @selector(compare:)];
self.states = sortedStates;
```

除非特别指定初始时选中的值，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对应的邮政编码数组，将这个数组赋给 `zips` 数组，`zips` 数组用于向右侧滚轮提供数据。

```
NSString *selectedState = self.states[0];
self.zips = self.stateZips[selectedState];
```

两个数据源方法实际上都与上一个版本相同。在合适的数组中返回行数。我们实现的第一个委托方法也与上一个版本相同。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[row];
        self.zips = self.stateZips[selectedState];
        [self.dependentPicker reloadComponent:kZipComponent];
        [self.dependentPicker selectRow:0
                                inComponent:kZipComponent
                                animated:YES];
    }
}
```

只要选取器的选择发生变化，就会调用这个方法，在这个方法中检查 `component` 参数可以知道发生变化的是否是左侧的滚轮。如果确实是左侧的滚轮发生了变化，就提取新选择的项对应的数组，并将它赋给 `zips` 数组。然后，将右侧滚轮设置为选中第一行，并重新加载右侧滚轮。通过在州发生变化时交换 `zips` 数组，可以使其他的代码与 `DoublePicker` 示例中的代码相同。

我们的工作还没完成。编译并运行应用程序，查看 `Dependent` 分页的视图，如图 7-16 所示。有没有什么不满意的地方？

两个滚轮的大小相同。即使邮政编码不超过 5 个字符，它也会与州名占用同样的空间。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）等州名，这似乎不太理想。幸好可以实现另一个委托方法来指定每个滚轮应该占用的宽度。在纵向模式下，选取器滚轮的可用宽度大约为 295 点数，每添加一个新滚轮，都需要占用一些空间来绘制新滚轮的边缘。也许需要亲自动手尝试调整滚轮的宽度值以获得最佳的显示效果。将以下方法添加到 `BIDDependentComponentPickerController.m` 的委托部分：

```
- (CGFloat)pickerView:(UIPickerView *)pickerView
widthForComponent:(NSInteger)component
```

```

{
    if (component == kZipComponent) {
        return 90;
    } else {
        return 200;
    }
}

```



图 7-16 两个滚轮的宽度一定要相同吗？注意，有个比较长的州名被截断了

这个方法返回一个数字，代表每个滚轮的宽度应该为多少像素，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，此时 Dependent 分页的视图与图 7-5 更加类似。

现在，你应该非常熟悉选取器和分页栏应用程序了。我们还要展示选取器的另一种用法，接下来的学习过程会非常有趣。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一个简单游戏

接下来，我们将创建一个简单的老虎机游戏。虽然这个老虎机不会吐给我们大把的钱，但它确实是一个不错的游戏。先看看图 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

首先将以下代码添加到 BIDCustomPickerViewController.h 中：

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

```
@interface BIDCustomPickerViewController : UIViewController
<UIPickerViewDataSource, UIPickerViewDelegate>
```

```
@end
```

接下来请切换到 `BIDCustomerPickerViewController.m` 并在接近文件顶部的类扩展中添加以下属性变量：

```
#import "BIDCustomPickerViewController.h"

@interface BIDCustomPickerViewController ()

@property (strong, nonatomic) NSArray *images;

@end
```

此时我们向类中添加的是一个 `NSArray` 对象属性，它包含了显示在屏幕上的象征图像。代码剩余内容后面会继续补充。

## 7.8.2 构建视图

尽管图 7-6 中的选取器看起来比我们构建的其他视图更漂亮，但是界面的设计方式实际上没有太大区别。额外的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，然后在项目导航面板中选中 `Main.storyboard` 并选中 `Custom Picker View Controller` 以对 GUI 进行编辑。添加一个选取器视图，在选取器视图下方添加一个分页，在分页下添加一个按钮。然后将分页和按钮垂直居中对齐。将按钮的标题改为 `Spin`。

现在，移动分页以使它与视图的左侧引导线对齐，同时对齐到选取器视图底部的引导线。接下来，调整分页，使其右侧边缘扩展至右侧引导线，同时使其底部边缘与按钮顶部上方的引导线对齐。

选中分页，调出属性检查器。将 `Alignment` 设置为居中对齐。然后单击 `Text Color` 以更改文本颜色，将颜色设置为比较喜庆的颜色，比如亮紫红色（实际上我们并不知道这是什么颜色，但它看起来比较喜庆）。

接下来，要将文本设置得大一些。在检查器中查找 `Font` 设置，点击它内部的图标（看起来像字母 T 被圈在一个小方框中）会弹出字体选择器。使用该控件可将设备的标准系统字体设为你喜欢的其他类型，或者仅仅改变字号。在这个例子中，只是简单地将字体大小更改为 48。获得了想要的文本格式之后，删除单词“Label”，因为我们不希望在用户第一次获胜之前显示任何文本。

然后，建立界面元素到输出接口和操作方法的关联。创建一个名称为 `picker` 的新输出接口并将视图控制器关联到选取器视图，另一个名称为 `winLabel` 的输出接口将视图控制器关联到分页，将按钮的 `Touch Up Inside` 事件关联到名为 `spin` 的新操作方法。然后，确保为选取器指定了委托和数据源。

最后，还有一件事要做。选中选取器并打开属性检查器。需要取消选择 `View` 设置底部的 `User Interaction Enabled` 复选框，这样，用户就不能够手动更改刻度盘作弊了。完成之后，保存对分镜的修改。

## iOS 设备支持的字体

请谨慎使用界面构建器中的字体面板设计 iOS 界面。属性检查器的字体选择器允许开发者从大量字体中指定字体类型，但不同的 iOS 设备可用的字体集可能不同。例如，编写本书时，一些字体在 iPad 上可用，但是在 iPhone 和 iPod touch 上却不可用。应该将字体选项限制为目标 iOS 设备上的一个字体集。Jeff LaMarche 有一篇优秀的 iOS 博文介绍了如何使用程序获取设备的可用字体列表：<http://iphonedevdevelopment.blogspot.com/2010/08/fonts-and-font-families.html>。

简单来讲，只需创建一个基于视图的应用程序，并将下面这段代码添加到应用程序委托中的 `application:didFinishLaunchingWithOptions:` 方法即可：

```
for (NSString *family in [UIFont familyNames]) {
    NSLog(@"%@", family);
    for (NSString *font in [UIFont fontNamesForFamilyName:family]) {
        NSLog(@"\t%@", font);
    }
}
```

在恰当的模拟器中运行项目，可用的字体就会显示在项目的控制台日志中。

### 7.8.3 添加图像资源

现在需要添加将在游戏中使用的图像。项目归档文件的 07 Pickers/Custom Picker Images 文件夹中包含了 6 组图像文件（seven.png、bar.png、crown.png、cherry.png、lemon.png、apple.png 和所有图像的 @2x 版本）。和之前对分页栏图标进行的处理一样，将所有这些文件拖动到 Xcode 的 Images.xcasset 文件夹中。

### 7.8.4 实现控制器

在实现这个控制器的过程中，会学习到许多新内容。选中 BIDCustomPickerViewController.m 文件并补充 `spin:` 方法的内容：

```
- (IBAction)spin:(id)sender {
    BOOL win = NO;
    int numInRow = 1;
    int lastVal = -1;
    for (int i = 0; i < 5; i++) {
        int newValue = random() % [self.images count];

        if (newValue == lastVal) {
            numInRow++;
        } else {
            numInRow = 1;
        }
        lastVal = newValue;

        [self.picker selectRow:newValue inComponent:i animated:YES];
    }
}
```

```

        [self.picker reloadComponent:i];
        if (numInRow >= 3) {
            win = YES;
        }
    }
    if (win) {
        self.winLabel.text = @"WIN!";
    } else {
        self.winLabel.text = @"";
    }
}

```

接下来，将以下代码添加到 viewDidLoad 方法中：

```

- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后（从它的分镜加载），执行其他额外设置
    self.images = @[
        [UIImage imageNamed:@"seven"],
        [UIImage imageNamed:@"bar"],
        [UIImage imageNamed:@"crown"],
        [UIImage imageNamed:@"cherry"],
        [UIImage imageNamed:@"lemon"],
        [UIImage imageNamed:@"apple"]];

    srand(time(NULL));
}

```

最后，将以下代码添加到文件末尾 @end 语句之上：

```

#pragma mark -
#pragma mark Picker Data Source Methods
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView
{
    return 5;
}

- (NSInteger)pickerView:(UIPickerView *)pickerView
numberOfRowsInComponent:(NSInteger)component
{
    return [self.images count];
}

#pragma mark Picker Delegate Methods
- (UIView *)pickerView:(UIPickerView *)pickerView
viewForRow:(NSInteger)row
forComponent:(NSInteger)component reusingView:(UIView *)view
{
    UIImage *image = self.images[row];
    UIImageView *imageView = [[UIImageView alloc] initWithImage:image];
    return imageView;
}

@end

```

这段代码中包含许多新内容。接下来依次分析各个方法。



### 1. spin 方法

spin 方法将在用户点击 Spin 按钮时触发。在该方法中，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 判断一行中是否出现了 3 个连续相同的行，如果是，则将 win 设置为 YES。使用 numInRow 跟踪到目前为止一行中连续相同的值出现的次数，同时在 lastVal 中记录前一个滚轮的值，以便与当前值进行比较。将 lastVal 初始化为 -1，因为我们知道，-1 不会与任何真实的值相匹配：

```
BOOL win = NO;
int numInRow = 1;
int lastVal = -1;
```

接下来，通过循环将所有 5 个滚轮的当前行都设置为一个新的随机行。根据 images 数组的元素个数来选取随机数，这是一种非常便捷的方法，因为我们知道，这 5 列具有相同数量的值：

```
for (int i = 0; i < 5; i++) {
    int newValue = random() % [self.images count];
```

将新值与上一个值进行比较，如果它们匹配，则将 numInRow 加 1。如果不匹配，则将 numInRow 重置为 1。然后将新值赋给 lastVal，这样，就可以在下一循环中用它进行比较：

```
if (newValue == lastVal) {
    numInRow++;
} else {
    numInRow = 1;
}
lastVal = newValue;
```

然后，将相应的滚轮设置为新值，通知这个滚轮使用动画效果进行改变，并通知选取器重新加载这个滚轮：

```
[self.picker selectRow:newValue inComponent:i animated:YES];
[self.picker reloadComponent:i];
```

每次循环要做的最后一件事就是，查看一行中是否出现了 3 个连续相同的图像，如果是，则将 win 设置为 YES：

```
if (numInRow >= 3) {
    win = YES;
}
```

完成循环之后，设置分页的文本以显示是否获胜：

```
if (win) {
    self.winLabel.text = @"WIN!";
} else {
    self.winLabel.text = @"";
}
```

### 2. viewDidLoad 方法

回顾一下我们在这里添加的代码，首先载入 6 个不同的图像。我们使用了 UIImage 类提供的 imageNamed: 便利方法。

```
self.images = @[UIImage imageNamed:@"seven",
                UIImage imageNamed:@"bar", UIImage imageNamed:@"crown"],
```

```
[UIImage imageNamed:@"cherry"], [UIImage imageNamed:@"lemon"],  
[UIImage imageNamed:@"apple"]];
```

这个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,那么每次游戏过程都完全一样,这样就失去游戏的乐趣。

```
srandom(time(NULL));
```

其实很简单,是吧?但是,这6个图像是用来做什么的呢?如果继续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,但是如果再继续查看委托方法,将会发现我们使用了一个完全不同的委托方法向选取器提供数据。在此之前使用的委托方法都是返回一个 `NSString *`,但是这个委托方法返回的是 `UIView *`。

使用这个委托方法,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实际上是有限制的。虽然这个方法需要多编写一些代码,但它为我们选择显示的内容提供了更多自由。

```
- (UIView *)pickerView:(UIPickerView *)pickerView  
    viewForRow:(NSInteger)row  
    forComponent:(NSInteger)component reusingView:(UIView *)view  
{  
    UIImage *image = self.images[row];  
    UIImageView *imageView = [[UIImageView alloc] initWithImage:image];  
    return imageView;  
}
```

这个方法首先初始化6个图像中的其中一个,得到一个 `UIImageView` 对象,然后将这个对象返回。为此,首先获得与目标行对应的图像。然后使用这个图像创建一个图像视图并返回。因为视图比图像要复杂,建议预先将所需的视图创建出来(比如可以在 `viewDidLoad:` 方法中创建),然后就可以在需要时直接将预创建的视图返回。但是对于这个简单的示例,我们可以在需要时才创建视图。

现在,深呼吸。你一口气学完了所有内容,接下来可以试玩一个这个游戏。

## 7.8.5 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。接下来通过一些小的调整改进这个游戏。现在,还有两个地方不尽人意。

- 它没有声音,老虎机竟然如此安静!
- 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,但会降低游戏的趣味性。可以再次运行应用程序亲自观察一下。虽然不是很明显,但分页确实在转轮停止旋转之前就出现了。

本书附带项目归档文件中的 07 Pickers/Custom Picker Sounds 文件夹中包含两个声音文件: `crunch.wav` 和 `win.wav`。将这个文件夹添加到项目的 Pickers 文件夹中。这两个声音分别在用户单击 `Spin` 按钮和获胜时播放。

要使用声音,就需要访问 `iOS Audio Toolbox` 中的类。在 `BIDCustomPickerViewController.m` 开头插入这样一行代码:

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

接下来,需要添加一个关联到 Spin 按钮的输出接口。在转轮旋转的过程中,需要将按钮隐藏。我们不希望用户在当前循环全部完成之前再次点击该按钮。将以下代码添加到 BIDCustomPickerViewController.m:

```
@interface BIDCustomPickerViewController ()
```

```
@property (strong, nonatomic) NSArray *images;
@property (weak, nonatomic) IBOutlet UIPickerView *picker;
@property (weak, nonatomic) IBOutlet UILabel *winLabel;
@property (weak, nonatomic) IBOutlet UIButton *button;
```

```
@end
```

输入完成之后保存文件,点击 Main.storyboard,开始编辑 GUI。打开该文件之后,按住 control 键并从视图控制器拖到 Spin 按钮上,将按钮关联到刚才创建的按钮输出接口。保存分镜。

现在,需要在控制器类的实现文件中做一些工作。首先,需要一个实例变量来保存声音的引用。打开 BIDCustomPickerViewController.m 文件添加如下代码:

```
@implementation BIDCustomPickerViewController {
```

```
    SystemSoundID winSoundID;
```

```
    SystemSoundID crunchSoundID;
```

```
}
```

```
:
```

```
:
```

```
:
```

还需要向控制器类添加两个方法。将以下两个方法添加到 BIDCustomPickerViewController.m 文件,作为这个类的前两个方法:

```
- (void)showButton
```

```
{
```

```
    self.button.hidden = NO;
```

```
}
```

```
- (void)playWinSound
```

```
{
```

```
    if (winSoundID == 0) {
```

```
        NSURL *soundURL = [[NSBundle mainBundle] URLForResource:@"win"
                                                                    withExtension:@"wav"];
        AudioServicesCreateSystemSoundID((__bridge CFURLRef)soundURL,
```

```
                                         &winSoundID);
    }
```

```
    AudioServicesPlaySystemSound(winSoundID);
```

```
    self.winLabel.text = @"WINNING!";
```

```
    [self performSelector:@selector(showButton)
```

```
        withObject:nil
```

```
        afterDelay:1.5];
```

```
}
```

第一个方法用于显示按钮。如之前所说,将在用户单击按钮之后将其隐藏,因为如果滚轮还在旋转,就不应该让它们在停止之前再次开始旋转。

第二个方法将在用户获胜时调用。首先，检查声音是否已经加载完成。实例变量会初始化为 0，而有效的声音标识符不会是 0，所以通过比较实例变量的值是否为 0 就能知道声音是否已经加载完成。为了加载声音，首先向程序包请求名为 win.wav 的声音文件路径，就和之前在 Dependent 选取器视图中加载属性列表时一样。获取到该资源的路径后，接下来的 3 行代码就加载该声音文件并播放。然后，将分页设置为“WINNING!”，并调用 showButton 方法，但这里是通过 performSelector:withObject:afterDelay: 方法来调用 showButton 方法。这是一个非常方便的方法，所有对象都可以调用它。它支持在未来的某个时间对方法进行调用。以本例来说，将在 1.5 秒之后调用 showButton 方法，这就可以保证在刻度盘旋转到最终位置之后才将结果告知用户。

**注意** 你可能注意到 AudioServicesCreateSystemSoundID 方法的调用方式有点奇怪。该方法接受一个 URL 作为第一个参数，但它需要的并非是 NSURL 的实例，而是一个 CFURLRef 结构。苹果通过 Core Foundation 框架为很多常用滚轮（比如 URL、数组、字符串等）提供了 C 语言接口。通过这种方式，就算是完全用 C 编写的应用程序也能访问一些基于 Objective-C 的功能。有趣的是，这些 C 滚轮通过“桥接”（bridged）来使用对应的 Objective-C 副本。例如，CFURLRef 在功能上与 NSURL 指针相同。这就意味着某些使用 Objective-C 创建的对象可以通过桥接来使用 C 语言的 API，反之亦然。这是通过 C 语言的强制类型转换实现的，将你需要的类型放在一对括号中，置于待转换的变量名之前。从 iOS 5 开始，如果使用 ARC，必须在类型名之前加上 `_bridge` 关键字，这样 ARC 才能知道应该如何处理这个被传递给 C 语言 API 的 Objective-C 对象。

还需要对 spin: 方法进行一些更改。需要编写代码来播放声音，并在玩家获胜之后调用 playerWon 方法。现在对代码进行以下更改：

```
- (IBAction)spin:(id)sender {
    BOOL win = NO;
    int numInRow = 1;
    int lastVal = -1;
    for (int i = 0; i < 5; i++) {
        int newValue = random() % [self.images count];

        if (newValue == lastVal) {
            numInRow++;
        } else {
            numInRow = 1;
        }
        lastVal = newValue;

        [self.picker selectRow:newValue inComponent:i animated:YES];
        [self.picker reloadComponent:i];
        if (numInRow >= 3) {
            win = YES;
        }
    }
    if (crunchSoundID == 0) {
```

```

NSString *path = [[NSBundle mainBundle] pathForResource:@"crunch"
                                                         ofType:@"wav"];
NSURL *soundURL = [NSURL fileURLWithPath:path];
AudioServicesCreateSystemSoundID((__bridge CFURLRef)soundURL,
                                  &crunchSoundID);
}
AudioServicesPlaySystemSound(crunchSoundID);

if (win) {
    [self performSelector:@selector(playWinSound)
                withObject:nil
                afterDelay:.5];
} else {
    [self performSelector:@selector(showButton)
                withObject:nil
                afterDelay:.5];
}
self.button.hidden = YES;
self.winLabel.text = @"";

——if (win){
——    self.winLabel.text = @"WIN!";
——} else {
——    self.winLabel.text = @"";
——}
}

```

首先，像之前处理获胜时的声音一样，先加载按钮点击时的效果声音（如果需要）。然后播放这个声音，以使玩家知道他们已经转动了滚轮。接下来，使用了一个小技巧，不是在知道玩家获胜之后立即将分页显示为“WINNING?!”。而是调用了之前创建的两个方法其中的一个，但是通过 `performSelector:afterDelay:` 方法进行延迟调用。如果玩家获胜，程序会在 0.5 秒之后调用 `playerWon` 方法，这样就有足够的时间让刻度盘旋转到终点；如果玩家失败了，程序将等待 0.5 秒，然后重新启用 `Spin` 按钮，以便玩家可以再次点击。我们在转轮的旋转过程中隐藏了按钮并且清空了分页的文本。

现在你已经全部完成了！点击运行按钮并点中最后一个分页来检验这个老虎机吧。单击 `Spin` 按钮时会播放一种声音，胜利时会播放胜利音乐。非常好！

## 7.9 小结

现在，你应该已经掌握了分页栏应用程序和选取器的基础知识。本章从头构建了一个非常完整的分页栏应用程序，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，如何创建带有多个滚轮的选取器，以及如何使某个滚轮中的值依赖于另一个滚轮中选定的值。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和声音，如何通过属性列表创建字典。本章内容非常多，恭喜你掌握了本章的知识！如果你已经准备好了学习表视图，那么翻到下一页，我们继续学习新的内容。

本章将构建一个基于导航的分层应用，它类似于 iOS 设备自带的邮件应用。通过这个应用，用户可以访问嵌套的数据列表并进行编辑。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。

表视图是向用户显示数据列表的一种最常见机制。它们是高度可配置的对象，可以根据用户需要随意配置，邮件应用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。设置、音乐、时钟应用同样会使用表视图，尽管这些应用的外观差别很大（参见图 8-1）。



图 8-1 虽然外观不同，但设置、音乐和时钟应用都使用表视图显示数据

## 8.1 表视图基础

表用于显示数据列表。数据列表中的每一项对应表的一行。iOS 没有限制表的行数，行数仅受可用存储空间的限制。iOS 的表只能有一列。

### 8.1.1 表视图和表视图单元

表视图是用于显示表数据的视图对象，它是 `UITableView` 类的一个实例。表中的每个可见行都由 `UITableViewCell` 类实现。因此，表视图是用于显示表中可见部分的对象，表视图单元则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。本章稍后讲解示例程序时你可以了解到这些工作原理。

前面提到，所有表都只有 1 列。但是，图 8-1 右边的时钟应用，从外观看确实至少拥有 2 列，如果把时钟图标也算进去，会发现其实是 3 列。不过情况并非如此。表中的每一行都由一个 `UITableViewCell` 表示。每个 `UITableViewCell` 对象都可以包含图像、文本，还有一个可选的附加图标（最右边的小图标，下一章会详细介绍附加图标）。

如果需要的话，可以向 `UITableViewCell` 添加子视图，从而在一个单元中放置更多的数据。你可以通过两种基本方法来完成此操作。一种方法是创建单元时在程序中添加子视图，另一种方法是从镜或 nib 文件中加载它们。你可以按照喜欢的方式展示表视图单元，也可以添加想要的子视图。这样看来，单列限制并不像开始听起来那样可怕。如果你觉得听晕了，别担心，本章稍后将介绍这方面的技术。

### 8.1.2 分组表和无格式表

表视图有以下两种基本样式。



- ❑ 分组表 (grouped table): 分组表包含了一个或一个以上的行分组。在分组中的每行都紧密地贴合在一起, 而分组之间有明显的间距, 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分组。
- ❑ 无格式表 (plain table): 无格式表是默认的风格。在这个样式中的分组间距非常细微, 每个分组的标题可以选择自定义样式。如果使用了索引, 这种表又称为索引表 (indexed table), 如图 8-3 最右边的图片所示。

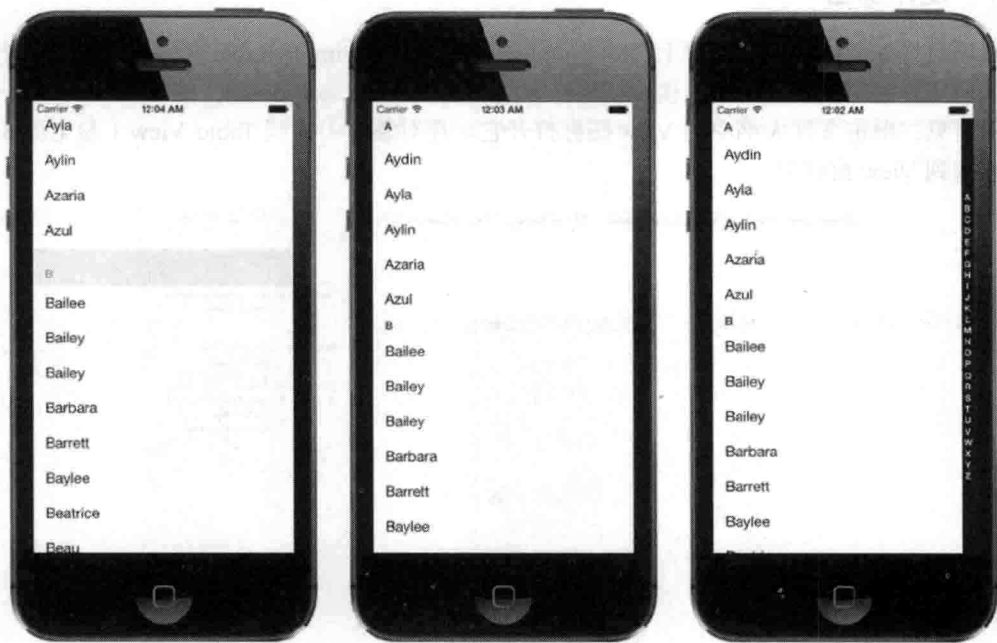


图 8-3 同一个表视图, 分别使用分组表 (左)、不带索引的无格式表 (中间) 和一个带索引的无格式表 (右) 显示

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在列表中进行导航。

表中的每个部分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。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, 以此类推。

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。

**警告** 从技术上说, 可以创建带有索引的分组表, 但是不应该这么做。iPhone Human Interface Guidelines 文档中明确指出分组表不应该使用索引。

## 8.2 实现一个简单表

下面通过一个最简单的示例解释表视图的工作原理。本示例将显示一个文本值列表。

在 Xcode 中创建一个新项目。本章将使用 Single View Application 模板，因此选择这一项。将项目命名为 Simple Table，在 Class Prefix 中输入 BID，并将 Device Family 设为 iPhone。

### 8.2.1 设计视图

在项目导航面板中，展开最上层的 Simple Table 项目和 Simple Table 文件夹。这是一个极为简单的应用，它不需要任何输出接口或操作方法。选中 Main.storyboard，编辑 GUI。如果 View 窗口不可见，单击文件大纲中的 View 图标打开它。在对象库中找到 Table View（参见图 8-4），并将它拖到 View 窗口中。

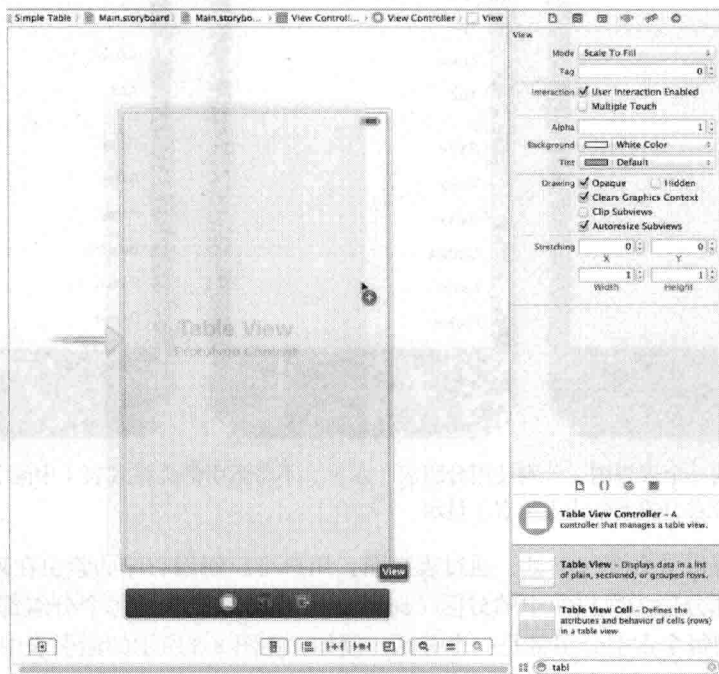


图 8-4 从库中拖出一个表视图放到主视图上。注意，表视图会自动调整尺寸以占满整个视图

表视图会自动设置高度和宽度以占满整个视图。这正是我们所希望的。表视图的宽度是整个屏幕的宽度，高度是除应用导航栏、工具栏和分页栏之外的整个屏幕高度。将表视图放置在 View 窗口中并与父视图中心对齐。

在继续下一步之前，有一个问题需要先解决。当前的视图尺寸是固定的，与父视图尺寸一致。但假如应用运行在屏幕尺寸与分镜设定不同的设备上，父视图的尺寸会发生变化。比如说，假定

你在界面构建器中将这个视图设定为 4 英寸的 Retina 屏幕，但却是在 iPhone 4 或 3.5 英寸 Retina 模式的模拟器中运行应用，表视图会维持原先的尺寸，这意味着它将超过屏幕的大小并越出底部。

好在我们可以使用约束来简单地解决这个问题。在之前的章节里，我们通过 Editor 菜单中的多个选项来添加约束，不过现在将展示另一种方法。你可以看到，在界面构建器的编辑区域底部有一排小按钮。其中一部分按钮与约束相关。在表视图还是选中状态时，将你的鼠标移动到底部的每一个按钮之上来查看它们的内容。我们想让表视图的边缘贴住父视图的边缘，因此找到气泡提示为 Pin 的按钮并点击它。图 8-5 显示了随后的界面。

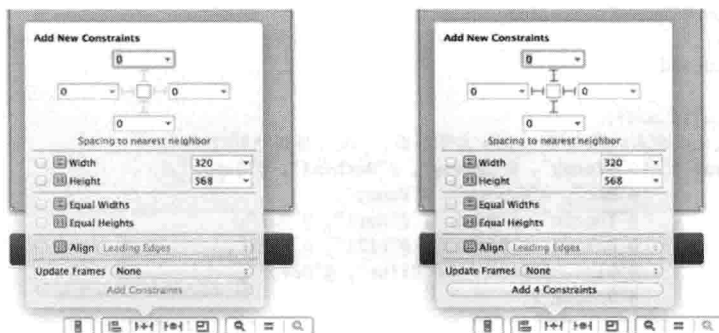


图 8-5 这里展示了 Add New Constraints 面板设置前和设置后的状态

这个面板可以让你为所选视图设置新的约束。在这个示例中，我们想要创建新的约束来固定视图与父视图的边缘。你只需要依次点击面板上面小正方形四周的虚线相联符号就轻松解决了。点击之后它们就都变成实线了，而且底下的按钮会更新数目来告诉你添加了多少个约束。四条线都点好后，请点击 Add 4 Constraints 按钮进行布置。

保持表视图的选中状态，按下 `option+command+6` 打开关联检查器。你会注意到，表视图的前两个可用关联和选取器视图的前两个关联是一样的，都是 `dataSource` 和 `delegate`。把每个关联旁边的小圆圈拖到 View Controller 图标上。这样一来，控制器类就成为表的数据源和委托了。

继续保持表视图的选中状态，打开属性检查器（`option+command+4`），然后在 View 分区中的 Tag 文本框中输入 1。如果我们为视图指定了唯一的标记值，这些值将来可以在代码中获取视图时使用到。我们之后将对表视图执行这样的操作。

完成上述关联之后，保存分镜，然后开始深入分析 UITableView 代码。

## 8.2.2 编写控制器

下面是控制器类的头文件。单击 `BIDViewController.h`，并添加以下代码：

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

@interface BIDViewController : UIViewController
<UITableViewDataSource, UITableViewDelegate>

@end
```

上述代码的作用是让类遵循两个协议，类只有遵循这两个协议才能成为表视图的委托和数据源。保存文件。现在切换到 BIDViewController.m，将以下代码添加到文件的开头：

```
#import "BIDViewController.h"

@interface BIDViewController ()

@property (copy, nonatomic) NSArray *dwarves;

@end

@implementation BIDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后（通常是从 nib 文件加载），做一些额外的设置
    self.dwarves = @[@"Sleepy", @"Sneezy", @"Bashful", @"Happy",
                    @"Doc", @"Grumpy", @"Dopey",
                    @"Thorin", @"Dorin", @"Nori", @"Ori",
                    @"Balin", @"Dwalin", @"Fili", @"Kili",
                    @"Oin", @"Gloin", @"Bifur", @"Bofur",
                    @"Bombur"];

    UITableView *tableView = (id)[self.view viewWithTag:1];
    UIEdgeInsets contentInset = tableView.contentInset;
    contentInset.top = 20;
    [tableView setContentInset:contentInset];
}
```

最后，将以下代码添加到文件的末尾：

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
    return [self.dwarves count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *SimpleTableIdentifier = @"SimpleTableIdentifier";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
                             SimpleTableIdentifier];

    if (cell == nil) {
        cell = [[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleDefault
                reuseIdentifier:SimpleTableIdentifier];
    }

    cell.textLabel.text = self.dwarves[indexPath.row];
    return cell;
}

@end
```

首先声明一个用于放置所要显示数据的数组，之后我们为控制器添加了 3 个方法。你可能很熟悉第一个方法 `viewDidLoad`，因为前面使用过类似的方法。此方法只创建了一个数据数组，这些数据要在表中显示。在实际的应用中，此数组很可能来自另一个源，如文本文件、属性列表或 web 服务。这里还执行了一个新的操作：我们调整了表视图顶部边缘的偏移值，这样初始屏幕就不会影响透明的状态栏。我们在这里使用了之前在分镜中设置的 `tag` 标记值以访问表视图。

继续往下看，你会看到添加了两个数据源方法。第一个方法是 `tableView:numberOfRowsInSection:`，供表用来获取指定分区中的行数。你可能猜到了，默认的分区号为 1，此方法用于返回列表中某一分区中的行数。只需返回数组的元素数量即可。

下一个方法可能需要稍加解释，因此详细介绍一下该方法：

```
(UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
```

当表视图需要绘制其中一行时，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的索引绑定到一个对象中。要从 `NSIndexPath` 中获得行或分区的索引，只需要使用 `row` 属性或 `section` 属性即可，这两个属性都返回一个整形值。

第一个参数 `tableView` 是一个引用，指向触发这个方法的表。这样，就可以创建一个类来充当多个表的数据源。

然后，声明一个静态的字符串实例：

```
static NSString *SimpleTableIdentifier = @"SimpleTableIdentifier";
```

此字符串将作为键使用，用来表示某种表单元。此表将只使用一种单元。

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，如果表试图为表中的每一行都分配一个表视图单元，而不管该行当前是否正在显示，那么这样做会造成大量的开销。幸好，表并不是这样工作的。

相反，当表视图单元滚离屏幕时，它们将被放置在一个可重用的单元队列中。如果系统运行较慢，表视图就从队列中删除这些单元，以释放存储空间，不过，只要有可用的存储空间，队列就会一直保存这些单元，以便日后再次使用它们。

每当表视图单元滚离屏幕时，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滚离屏幕的其中一个单元，系统就能避免不断创建和释放那些视图的开销。要充分利用此机制，我们需要表视图给出前面使用过的指定类型的单元。注意，这里需要使用前面声明的 `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，表视图中完全有可能没有任何闲置单元，所以需要在这个调用之后，检查得到的 `cell` 是否为 `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，因此需要确保它是使用 `SimpleTableIdentifier`

创建的。

```
if (cell == nil) {  
    cell = [[UITableViewCell alloc]  
            initWithStyle:UITableViewCellStyleDefault  
            reuseIdentifier:SimpleTableIdentifier];  
}
```

对 `UITableViewCellStyleDefault` 很好奇吧？再忍耐一下，我们介绍表视图单元样式时会深入探讨它。

现在，我们拥有了一个可以返回给表视图的表视图单元。下面要做的就是将需要在单元中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名为 `textLabel` 的 `UILabel` 属性，可以设置此属性以显示字符串。对于这种情况，需要从 `listData` 数组中获取正确的字符串，然后用它设置表视图单元的 `textLabel` 属性。

要获得正确的值，需要知道表视图正在请求的是哪一行。可以从 `indexPath` 的 `row` 属性获取当前行。使用表的行号从数组中获得相应的字符串，然后将其赋给表单元的 `textLabel.text` 属性，最后将表单元返回即可。

```
cell.textLabel.text = self.dwarves[indexPath.row];  
return cell;
```

并不难，是吧？

编译并运行应用，应该可以看到数组中的值显示在表视图中（参见图 8-6）。

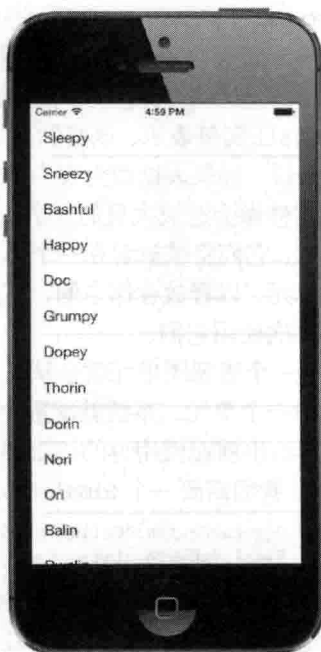


图 8-6 简单的表应用

### 8.2.3 添加一个图像

要是可以为每一行添加一个图像就好了。是不是需要创建一个 `UITableViewCell` 子类或使用子视图添加图像？不用。实际上，如果能接受图像位于每一行的左侧的话，就不需要做额外的工作了。默认的表视图单元会把这个情况处理好。下面来看一看。

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 `Images.assets` 目录中。`star.png` 是为该项目准备的一个小图标。

下面看看代码部分。在 `BIDViewController.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]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:SimpleTableIdentifier];
    }

    UIImage *image = [UIImage imageNamed:@"star"];
    cell.imageView.image = image;

    cell.textLabel.text = self.dwarves[indexPath.row];
    return cell;
}
```

大功告成！每个单元都有一个 `imageView` 属性。每个 `imageView` 都有一个 `image` 属性，以及一个 `highlightedImage` 属性。图像出现在单元文本左侧，并且在选中单元时会被替换为 `highlightedImage`（如果已提供）。可以把单元的 `imageView.image` 属性设置为想要显示的任何图像。

如果编译并运行应用，出现的列表每一行左侧都有一个漂亮的小星星图标（参见图 8-7）。当然，如果愿意的话，可以为表中的每一行设置不同的图像。或者，稍微多做一点儿工作，就可以为每一行分别应用不同的图标。

如果愿意，可以复制 `star.png`，调节一下它的颜色并将它添加到项目中，使用 `imageNamed:` 加载这个图像，然后将它赋给 `imageView.highlightedImage`。现在，如果单击表单元，新的图像就会显示出来。如果不打算调节颜色，可以使用项目归档文件中的 `star2.png`。

---

**注意** `UIImage` 使用一种基于文件名的缓存机制，所以它不会在每次调用 `imageNamed:` 时都重新加载新的图像，而是使用已经缓存的版本。

---





图 8-7 使用表视图单元的 `image` 属性，为每个表视图单元添加一个图像

## 8.2.4 表视图单元样式

目前对表视图进行的处理使用了图 8-7 所示的默认单元格样式(由常量 `UITableViewCellStyleDefault` 表示)。但 `UITableViewCell` 类包含其他几个预定义的单元格样式，可以很方便地向表视图添加更多样式。这些单元格样式使用了 3 种不同的单元格元素。

- **图像**：如果指定的样式中包含图像，那么该图像将显示在单元的文本左侧。
- **文本标签**：这是单元的主要文本。在之前使用的样式 `UITableViewCellStyleDefault` 中，文本标签是唯一在单元中显示的文本。
- **详细文本标签**：这是单元的辅助文本，通常用作解释性的说明或标签。

要查看这些新样式的外观，将以下代码添加到 `BIDViewController.m` 的 `tableView:cellForRowAtIndexPath` 中：

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *SimpleTableIdentifier = @"SimpleTableIdentifier";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        SimpleTableIdentifier];

    if (cell == nil) {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:SimpleTableIdentifier];
    }
}
```

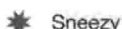
```

UIImage *image = [UIImage imageNamed:@"star.png"];
cell.imageView.image = image;

cell.textLabel.text = self.dwarves[indexPath.row];
if (indexPath.row < 7) {
    cell.detailTextLabel.text = @"Mr. Disney";
} else {
    cell.detailTextLabel.text = @"Mr. Tolkien";
}
return cell;
}

```

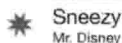
这里所做的只是设置单元的详细文本。前 7 行使用字符串 @"Mr. Disney", 其余行使用字符串 @"Mr. Tolkien"。运行这段代码时, 每个单元看起来仍然与以前一样 (参见图 8-8)。这是因为我们使用了样式 UITableViewCellStyleDefault, 这个样式并不包含详细文本。



★ Sneezzy

图 8-8 默认单元样式在一行中显示图像和文本标签

现在, 将 UITableViewCellStyleDefault 更改为 UITableViewCellStyleSubtitle 并再次运行。对于子标题样式, 两个文本元素都会显示, 其中一个位于另一个下方 (参见图 8-9)。



★ Sneezzy  
Mr. Disney

图 8-9 子标题样式在文本标签下方以较小的灰色字体显示详细文本

将 UITableViewCellStyleSubtitle 更改为 UITableViewCellStyleValue1, 然后编译并运行。此样式将文本标签和详细文本标签放在同一行, 分别位于单元的两端 (参见图 8-10)。



★ Sneezzy      Mr. Disney

图 8-10 Style Value 1 样式将文本标签放在左侧, 文本颜色为黑色。

将详细文本放在右侧, 文本颜色为蓝色

最后将 UITableViewCellStyleValue1 更改为 UITableViewCellStyleValue2。这种格式通常用于在信息旁边显示一个描述性的标签。这个样式并不显示表视图单元的图标, 而是将详细文本标签放在文本标签的左侧 (参见图 8-11)。在此布局中, 详细文本标签用于描述文本标签中保存的数据类型。



Sneezzy Mr. Disney

图 8-11 Style Value 2 样式不显示图像, 用蓝色显示详细文本标签, 并将其放在文本标签左侧

现在你已经了解了可用的单元样式, 继续学习之前, 将 UITableViewCellStyleValue2 改回之

前使用的 `UITableViewCellStyleDefault` 样式。稍后你将会看到如何定制表格的外观。但在此之前，请先确认有没有哪种自带的样式能够满足需求。

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，让我们看一下几个配置选项。下一章将更详细地介绍此内容。

## 8.2.5 设置缩进级别

可以使用委托指定缩进某些行。在 `BIDViewController.m` 文件中，在代码中的 `@end` 声明上方添加以下方法：

```
- (NSInteger)tableView:(UITableView *)tableView  
indentationLevelForRowAtIndexPath:(NSIndexPath *)indexPath  
{  
    return indexPath.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，以此类推。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。缩进级别的数值越大，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，就好像在邮件应用中表示子文件夹一样。

再次运行应用，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 8-12）。



图 8-12 表中每一行的缩进级别都比上一行高

## 8.2.6 处理行的选择

表的委托对象可以使用两个方法来确定用户是否选择了特定的行。其中一种方法会在一行被点击但是高亮显示之前就调用，并且可以用于阻止选中这一行，甚至改变被选中的行。我们来实现这个方法，并指定不能选中第一行。将以下方法添加到 BIDViewController.m 的尾部，位于@end 声明之前：

```
- (NSIndexPath *)tableView:(UITableView *)tableView
willSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (indexPath.row == 0) {
        return nil;
    } else {
        return indexPath;
    }
}
```

这个方法获取传递过来的 indexPath，它代表即将被选中的行对应的索引。我们的代码首先检查哪一行将被选中。如果这一行是第一行（索引值是 0），就将返回 nil，表示实际上不应该选中任何一行。否则，它返回 indexPath，表示可以继续选中 indexPath 对应的行。

在编译和运行应用之前，还要实现另一个委托方法，这个委托方法会在一行被选中之后调用，通常也是在这个委托方法中对选择进行实际处理。用户选中一行时，可以在这里执行任何操作。下一章将在这个方法中处理视图转换。本章只在这个方法中弹出一个警告视图，用于显示被选中的行。将下面的方法添加到 BIDViewController.m 的尾部，位于@end 声明之前。

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSString *rowValue = self.dwarves[indexPath.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];

    [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

添加此方法之后，编译并运行应用。看一下你是否能够选中第一行（应该不能），然后试着选择其他行。选中的行应该会高亮显示，然后会弹出一个警告视图，指出当前选中的是哪一行，同时选中的行会淡入到背景中（参见图 8-13）。

注意，还可以在将索引路径传递回来之前修改它，这样就会选中一个不同的行或分区。通常不应该这样做，除非你有充分的理由更改用户的选择。在大多数情况下，使用此方法时将返回未修改的 indexPath 或 nil，分别代表允许或禁止某个选择。

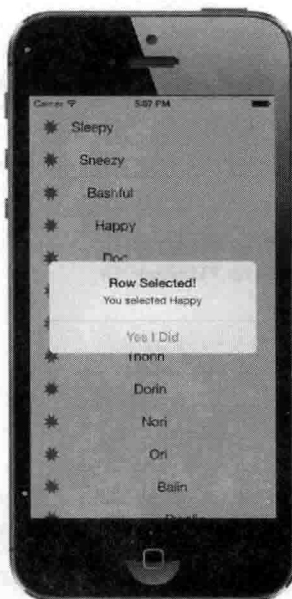


图 8-13 在本示例中，第一行是不可选的。选中其他任意一行都会显示一个警告。这个功能是使用委托方法完成的

## 8.2.7 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。大多数情况下，不应该覆盖默认的字体，那是用户希望看到的。不过有时我们有合适的理由这样做。在 `tableView:cellForRowAtIndexPath:` 方法中添加下列代码：

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *SimpleTableIdentifier = @"SimpleTableIdentifier";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        SimpleTableIdentifier];

    if (cell == nil) {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:SimpleTableIdentifier];
    }

    UIImage *image = [UIImage imageNamed:@"star.png"];
    cell.imageView.image = image;

    cell.textLabel.text = self.dwarves[indexPath.row];
    cell.textLabel.font = [UIFont boldSystemFontOfSize:50];
}
```

```

if (indexPath.row < 7) {
    cell.detailTextLabel.text = @"Mr. Disney";
} else {
    cell.detailTextLabel.text = @"Mr. Tolkien";
}
return cell;
}

```

现在运行应用，列表中值的字体会变得很大，与该行并不匹配（参见图 8-14）。

好，现在只能靠表视图委托了！表视图委托可以指定表视图的行高。实际上，如果需要的话，它可以为每一行指定不同值。下面向控制器类中添加此方法，代码位于@end 之前。

```

- (CGFloat)tableView:(UITableView *)tableView
heightForRowAtIndexPath:(NSIndexPath *)indexPath
{
    return 70;
}

```

在上面的代码中，表视图把所有行高都设置为 70 像素。编译并运行应用，现在表中的行应该高多了（参见图 8-15）。



图8-14 大字体效果不错！不过，有些内容被盖住了



图8-15 使用委托更改行的高度

委托能够处理的任务还有很多，但是，它的大部分功能会保留到下一章处理多层次数据时再使用。要了解更多内容，请使用文档浏览器查看 UITableViewDelegate 协议，然后看一下还有哪些可用方法。

## 8.3 定制表视图单元

你可以直接对表视图执行很多操作,不过一般来说,UITableViewCell 默认支持的格式通常不能很好地满足需要。在这种情况下,可以采用两种基本方法:一种方法是在创建单元时通过程序向 UITableViewCell 添加子视图,另一种方法是从 nib 文件中加载一组子视图。下面我们来看一下这两种方法。

### 8.3.1 向表视图单元添加子视图

要展示如何使用自定义单元,我们将使用另一个表视图创建一个新的应用。每一行将显示两行信息和两个标签(参见图 8-16)。应用将显示一系列常见计算机型号的名称和颜色。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。



图 8-16 向表视图单元添加子视图,可以让一个单元持有多行数据

使用 Single View Application 模板创建一个新的 Xcode 项目。它的设置跟前一个项目完全相同,将它命名为 Cells。双击 Main.storyboard,在界面构建器中编辑 GUI 界面。

向主视图添加一个 Table View,然后像在 Simple Table 应用中那样,使用关联检查器把委托和数据源设置为 File's Owner。选中表视图并且调出属性检查器(option+command+4),将 View 部分的 Tag 属性值设为 1。就像我们之前所做的那样,不需要添加任何特定的属性就可以在代码



中访问到表视图了。以及使用窗口底部的 Pin 按钮来创建视图与父视图边缘之间的约束。最后，保存分镜。

### 8.3.2 创建 UITableViewCell 子类

至此，我们使用的标准表视图单元处理了所有单元布局的细节。控制器代码并不关心应在何处放置标签和图像等细节问题，只是将需要显示的值传递给表单元，从而将表示逻辑与控制器分离，开发者应该始终坚持这种优良设计。在这个项目中，我们将要创建一个新的表视图单元子类，处理新布局的细节内容，从而使控制器尽可能简洁。

#### 1. 添加新单元

在项目导航面板中选择 Cells 文件夹，按下 `command+N` 创建一个新文件。在弹出的向导中，从 Cocoa Touch 部分选择 Objective-C class，然后点击 Next。在下一个页面键入 `BIDNameAndColorCell` 作为新类的名字，在 Subclass of 弹出列表中选择 `UITableViewCell`，然后再次点击 Next。在最后一个页面中，选择已经包含了其他源代码的 Cells 文件夹，确保在下面的 Group 和 Target 选项中都选择了 Cells，最后点击 Create。

选择 `BIDNameAndColorCell.h`，添加如下代码：

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

@interface BIDNameAndColorCell : UITableViewCell

@property (copy, nonatomic) NSString *name;
@property (copy, nonatomic) NSString *color;

@end
```

这里我们向表单元的 `interface` 文件添加了两个属性，控制器将使用它们向每一个单元传递值。注意，我们使用 `copy` 语义而不是 `strong` 语义声明 `NSString` 属性。对于 `NSString` 类型的属性，这种做法通常更好，因为传递给属性设置方法的字符串实际上有可能是 `NSMutableString` 类型，而这种类型的字符串值之后可能会被修改，以致引起一些问题。将每个传递给属性的字符串复制一份副本，这样就得到了一个不变的字符串副本，其中的内容始终是设置方法被调用时的值。

现在切换到 `BIDNameAndColorCell.m` 并添加如下代码：

```
#import "BIDNameAndColorCell.h"

@interface BIDNameAndColorCell ()

@property (strong, nonatomic) UILabel *nameLabel;
@property (strong, nonatomic) UILabel *colorLabel;

@end
```

我们在这里添加了一个类扩展，里面定义了 2 个用来访问子视图（之后我们会添加到表单元）的属性变量。表单元包含了 4 个子视图，其中的 2 个标签在每行的内容都发生有变化，因此我们创建了 2 个用来关联标签的属性变量。

我们需要添加所有的属性,请切换到@implementation 部分。在 initWithStyle:reuseIdentifier: 方法中添加代码以创建我们想要显示的视图:

```
- (id)initWithStyle:(UITableViewCellStyle)style reuseIdentifier:(NSString *)reuseIdentifier
{
    self = [super initWithStyle:style reuseIdentifier:reuseIdentifier];
    if (self) {
        // 初始化代码
        CGRect nameLabelRect = CGRectMake(0, 5, 70, 15);
        UILabel *nameMarker = [[UILabel alloc] initWithFrame:nameLabelRect];
        nameMarker.textAlignment = NSTextAlignmentRight;
        nameMarker.text = @"Name:";
        nameMarker.font = [UIFont boldSystemFontOfSize:12];
        [self.contentView addSubview:nameMarker];

        CGRect colorLabelRect = CGRectMake(0, 26, 70, 15);
        UILabel *colorMarker = [[UILabel alloc] initWithFrame:colorLabelRect];
        colorMarker.textAlignment = NSTextAlignmentRight;
        colorMarker.text = @"Color:";
        colorMarker.font = [UIFont boldSystemFontOfSize:12];
        [self.contentView addSubview:colorMarker];

        CGRect nameValueRect = CGRectMake(80, 5, 200, 15);
        UILabel *_nameLabel = [[UILabel alloc] initWithFrame:
                               nameValueRect];
        [self.contentView addSubview:_nameLabel];

        CGRect colorValueRect = CGRectMake(80, 25, 200, 15);
        UILabel *_colorLabel = [[UILabel alloc] initWithFrame:
                                colorValueRect];
        [self.contentView addSubview:_colorLabel];
    }
    return self;
}
```

这段代码相当直观,我们创建了 4 个 UILabel,并将它们添加到表视图单元。表视图单元已经拥有了一个名为 contentView 的 UIView 子视图, contentView 用于管理它的所有子视图,这种方式类似于第 4 章在一个 UIView 中包含两个开关的方式。因此,我们并没有将标签作为子视图直接添加到表视图单元,而是将其添加到 contentView。

这些标签中有两个用于存放静态文本,名为 nameMarker 的标签包含文本 Name:, 而 colorMarker 标签包含文本 Color:。我们不会修改它们。这两个标签都通过 NSTextAlignmentRight 设置为右对齐。

另外两个标签用于显示特定于行的数据。记住,我们稍后需要通过某种方式获取这两个字段,所以需要将这两个视图的引用保存在之前声明的属性变量中。

现在,对 BIDNameAndColorCell 作最后一处修改,在@end 之前添加两个赋值方法。

```
- (void)setName:(NSString *)n
{
    if (![n isEqualToString:_name]) {
        _name = [n copy];
    }
}
```

```

        self.nameLabel.text = _name;
    }
}

- (void)setColor:(NSString *)c
{
    if (![c isEqualToString:_color]) {
        _color = [c copy];
        self.colorLabel.text = _color;
    }
}

```

你已经知道使用@property 会为每个属性创建取值方法和赋值方法。然而，这里为 name 和 color 定义了自己的设置方法。这么做没有任何问题，如果一个类定义了自己的取值方法和赋值方法，就会覆盖默认的方法。在这个类中，使用了默认生成的取值方法，但是定义了我们自己的赋值方法，因此，当为 name 或 color 属性传递一个新的值时，就会更新之前创建的标签显示的内容。

## 2. 实现控制器代码

现在来实现这个简单的控制器，在新的表视图单元中显示内容。选择 BIDViewController.h，在其中添加如下代码：

```

#import <UIKit/UIKit.h>

@interface BIDViewController : UIViewController
<UITableViewDataSource, UITableViewDelegate>

```

```
@end
```

在控制器中准备一些要用到的数据，然后实现表的数据源方法将这些数据反馈给表。单击 BIDViewController.m，然后在文件开头添加以下代码：

```

#import "BIDViewController.h"
#import "BIDNameAndColorCell.h"

static NSString *CellTableIdentifier = @"CellTableIdentifier";
@interface BIDViewController ()

@property (copy, nonatomic) NSArray *computers;

@end

@implementation BIDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后（通常是从 nib 文件加载），做一些额外的设置

    self.computers = @[
        @{@"Name" : @"MacBook Air", @"Color" : @"Silver"},
        @{@"Name" : @"MacBook Pro", @"Color" : @"Silver"},
        @{@"Name" : @"iMac", @"Color" : @"Silver"},
        @{@"Name" : @"Mac Mini", @"Color" : @"Silver"},
        @{@"Name" : @"Mac Pro", @"Color" : @"Black"}];
}

```

```

UITableView *tableView = (id)[self.view viewWithTag:1];
[tableView registerClass:[BIDNameAndColorCell class]
 forCellReuseIdentifier:CellTableIdentifier];

UIEdgeInsets contentInset = tableView.contentInset;
contentInset.top = 20;
[tableView setContentInset:contentInset];

}

```

在这里，viewDidLoad 方法将一个字典数组赋给了 computers 属性。每个字典都包含表中某行的名称和颜色信息。行名称在字典的 Name 键下，颜色在 Color 键下。在结尾处使用了一个 tag 数字来寻找表视图，并注册表单元类以供后期重复使用。这些详细的内容之后很快就会看到。

---

**注意** 还记得 Mac 早期有各种不同的颜色（比如米色、银灰色、黑色、白色）吗？还不算拥有漂亮彩虹色的早期 iMac 和 iBook 系列。现在除了最新的 Mac Pro，只剩下一种颜色：银白色。此外现在也有彩色的 iPhone 了。

---

现在将以下代码添加到文件结尾处@end 声明的上面：

```

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
    return [self.computers count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    BIDNameAndColorCell *cell = [tableView dequeueReusableCellWithIdentifier:
                                   CellTableIdentifier
                                   forIndexPath:indexPath];

    NSDictionary *rowData = self.computers[indexPath.row];

    cell.name = rowData[@"Name"];
    cell.color = rowData[@"Color"];

    return cell;
}

@end

```

我们着重看一下 tableView:cellForRowAtIndexPath:，这个方法中出现了一些新东西。我们在这里使用了一个有趣的特性：表视图能够利用某种类注册机在需要时创建一个新单元。也就是说，如果注册了表视图用到的所有可重用标识符，就总是能访问到一个可用的表单元。在前面的示例中，我们使用 dequeueReusableCellWithIdentifier:方法执行了类似的事情，也使用了注册机，但如果注册机内没有标识符就会返回 nil。而现在使用的 dequeueReusableCellWithIdentifier:forIndexPath:方法则有一点不同，因为这个方法不会返回 nil。如果我们传递了一个没有注册过的

标识符，这个方法会直接崩溃。崩溃听起来很可怕，不过在当前环境，这意味着在开发中有一个小 bug 需要立即找到。因此，可以移除检查单元是否为 nil 的代码，因为得到的表单元绝对不会是 nil。

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 `Name`，一个是 `Color`。

```
NSDictionary *rowData = self.computers[indexPath.row];
```

最后要做的就是使用从所选行中获取的数据来填充单元，使用在子类中定义的属性即可实现这一目的。

```
cell.name = rowData[@"Name"];
cell.color = rowData[@"Color"];
```

编译并运行应用，就会看到一个表视图，表视图的每一行都显示了两行数据，如图 8-16 所示。

向表视图添加视图比仅仅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、定位和添加所有子视图是一项单调乏味的工作。如果我们能使用 Xcode 的 GUI 编辑工具来设计表视图单元就好了，不是吗？幸运的是，事实正如你所愿，你可以使用界面构建器设计表视图单元，然后在创建新单元时从 nib 文件中加载视图。

### 8.3.3 从 nib 文件加载 UITableViewCell

我们将使用界面构建器的可视化布局功能，重新创建刚才使用代码构建的两行界面。为此，可以创建一个包含表视图单元的新 nib 文件，使用界面构建器布局视图外观。然后，如果需要一个表视图单元来表示一行，那么不再创建一个标准的表视图单元，而是从 nib 文件加载子类，并使用单元子类中已定义的属性设置名称和颜色。除了使用界面构建器的可视化布局，我们还将将在其他一些方面简化代码。

首先，在 `BIDNameAndColorCell.m` 文件中修改 `BIDNameAndColorCell` 类。第一步是要将属性变量标记为输出接口，这样就可以在界面构建器中使用到了。在顶部的类扩展中作出以下更改：

```
@interface BIDNameAndColorCell ()

@property (strong, nonatomic) IBOutlet UILabel *nameLabel;
@property (strong, nonatomic) IBOutlet UILabel *colorLabel;

@end
```

现在还记得 `initWithStyle:reuseIdentifier:` 方法中用于创建标签的代码吗？这些都不需要了。事实上，应该删除整个方法，因为这些标签的所有创建工作都将在界面构建器中完成。

完成之后，这个单元子类就比之前简洁多了。它现在唯一的功能就是向标签中填入数据。现在需要在界面构建器中重新创建标签。

在 Xcode 中右击 `Cells` 文件夹，然后在弹出的关联菜单中选择 `New File...`。在新文件向导的左侧窗格中点击 `User Interface`（确保在 `iOS` 部分中选择，而不是在 `Mac OS X` 中选择）。在右上方的窗格中选择 `Empty`，然后点击 `Next`。在下一个页面中保留 `Device Family` 的值 `iPhone`，再次点击 `Next`。然后键入 `BIDNameAndColorCell.xib` 作为该 nib 的名称。确保在文件浏览器中选择了

主项目目录，并且在 Group 弹出框中选择了 Cells 组。

### 1. 在界面构建器中设计表视图单元

下一步，选中项目导航面板中的 BIDNameAndColorCell.xib，打开文件进行编辑。目前为止，我们都是分镜中设计 GUI 的，不过这次我们要用到 nib 文件。很多东西看起来都非常相似，不过还是有一些区别。其中一个主要区别就是分镜文件中陈列的场景都是视图控制器与视图一体的，而 nib 文件中则没有这样的强制关联。事实上 nib 文件通常不会包含一个真正的控制器，而是一个叫做 File's Owner 的代理者。如果你展开文件大纲就会在 First Responder 的上面找到它。

在做其他事情之前，我们先要关闭这个 nib 文件的自动布局功能。请你回想一下，自动布局就是在运行时决定视图位置和尺寸根据父视图或兄弟视图的几何变化而做何种改变的约束系统。因为我们定义的是这个视图的固定布局，所以不需要这个功能。只需要调出文件检查器（option+command+1）并取消界面构建器 Document 分区中 Use Autolayout 复选框的勾选就行了。

在库中找到一个 Table View Cell（参见图 8-17）并将其拖动到 GUI 布局区域上。

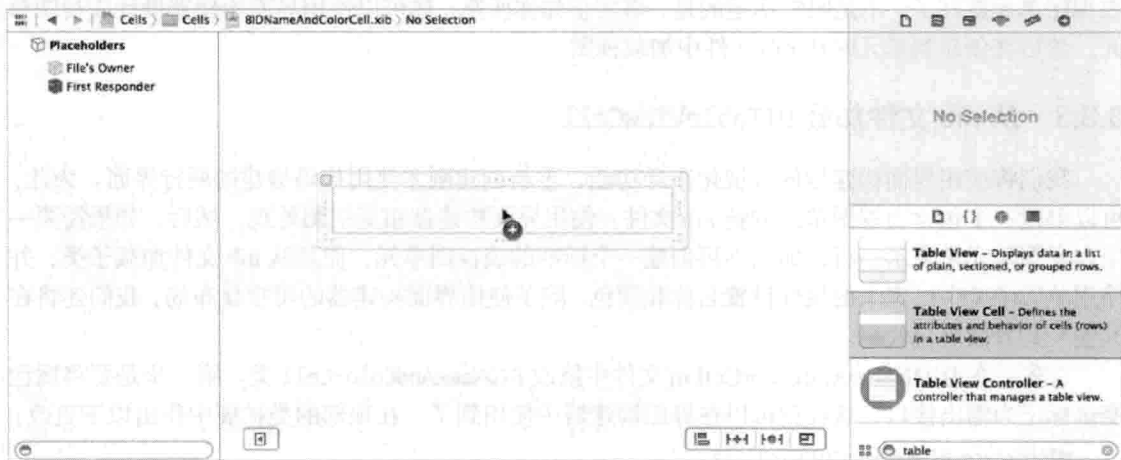


图 8-17 从库中拖出表视图单元，放到 nib 编辑器的 GUI 布局区域

确保选中了表视图单元，然后按下 option+command+5，打开尺寸检查器，将表视图单元的高度从 44 改为 65。这样我们就有更多的空间。

接下来，按下 option+command+4，打开属性检查器（参见图 8-18）。其中第一个字段是 Identifier，它是我们在代码中使用过的可重用标识符。如果记不起这一内容，请回头在本章中查找 CellTableIdentifier。将 Identifier 设置为 CellTableIdentifier。



图 8-18 表视图单元的属性检查器

这里的想法是，当获取一个单元格以便重用时，可能由于要将新单元格滚动到视图中，我们希望确保获得了正确的单元格类型。当这个特定的单元格从 nib 文件实例化时，可以向它的重用标识符实例变量预先填充你在 Identifier 字段中输入的 NSString——本例中是 CellTableIdentifier。

想象这样一种场景，你创建一个具有标题的表，然后创建一系列“中间”单元格。如果将一个中间单元格滚动到视图中，需要获取一个中间单元格以供重用，而不是获取标题单元格。Identifier 字段用于适当地标记单元格。

下一步是编辑表单元格的内容视图。从库中拖出 4 个 Label 控件，将它们放在内容视图中，参照图 8-19。由于标签之间靠得很近，顶部和底部的引导线基本上没有什么用，但左侧引导线和对齐引导线仍然能正常工作。你也可以拖出一个标签，然后按住 option 键创建其副本，或许你会喜欢这种简单的方法。



图 8-19 表视图单元的内容视图，其中拖入了 4 个标签

接下来，双击左上方的标签并将它更改为 Name:，将左下方的标签更改为 Color:。

现在将 Name: 和 Color: 标签都选中，按下属性检查器中 Font 字段旁边的小 T 按钮，这会打开



一个包括 Font 弹出按钮的小型面板。点击 Font 按钮，选择 System Bold 作为字体。如果有必要，选中右侧的两个未更改的标签字段，将它们稍微向右拖动，从而使设计更加合理。

最后，调整右侧的两个标签，将它们拉伸至右侧的引导线。从图 8-20 中应该可以了解我们最终的单元格内容视图。

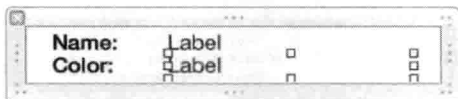


图 8-20 表视图单元格的内容视图，将左侧标签名称设置为粗体，对右侧标签进行了细微的移动和调整

现在，我们需要告诉界面构建器，这个表视图单元不是一个普通的单元，而是一个特殊的子类。否则，我们无法将输出接口关联至相关的标签。选择这个表视图单元，按下 option+command+3 打开身份检查器，在 Class 选项中选择 BIDNameAndColorCell。

接着按下 option+command+6 打关联检查器，你将在这里看到 colorLabel 和 nameLabel 输出接口。分别将它们拖至 GUI 中的相应标签上。

## 2. 使用新的表视图单元

要使用刚刚设计的表视图单元，只需要在 BIDViewController.m 中对 viewDidLoad: 方法作一些简单的修改：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.

    self.computers = @[
        @{@"Name" : @"MacBook Air", @"Color" : @"Silver"},
        @{@"Name" : @"MacBook Pro", @"Color" : @"Silver"},
        @{@"Name" : @"iMac", @"Color" : @"Silver"},
        @{@"Name" : @"Mac Mini", @"Color" : @"Silver"},
        @{@"Name" : @"Mac Pro", @"Color" : @"Black"}];

    UITableView *tableView = (id)[self.view viewWithTag:1];
    [tableView registerClass:[BIDNameAndColorCell class]
        forCellReuseIdentifier:CellTableIdentifier];
    tableView.rowHeight = 65;
    UINib *nib = [UINib nibWithNibName:@"BIDNameAndColorCell" bundle:nil];
    [tableView registerNib:nib forCellReuseIdentifier:CellTableIdentifier];

    UIEdgeInsets contentInset = tableView.contentInset;
    contentInset.top = 20;
    [tableView setContentInset:contentInset];
}
```

这里的第一个变动就是将表视图的行高设置为 65。我们已经在 CustomCell.xib 中修改过表视图单元的默认高度了，但是仍然不够。还需要将这个变化告诉表视图；否则，表视图不会为单元预留合适的显示空间。rowHeight 属性对所有行都生效，除非实现了 tableView:heightForRow-

`AtIndexPath:`委托方法。在这个委托方法中可以为每一行分别设置高度,不过我们现在用不到它,因此使用 `rowHeight` 属性以快速改变所有行的高度。

将类关联到可重用标识符之后,表视图就可以跟踪与特定的可重用标识符相关联的 nib 文件。这样,就可以为使用类或者 nib 文件创建的每一行注册表视图单元, `dequeueReusableCellWithIdentifier:forIndexPath:`方法总是会返回一个可用的单元。

这就好了。编译并运行程序。现在,两行的表单元都是基于界面构建器设计的。

现在你已经看到了两种设计表视图的方式,觉得如何?很多研究 iOS 开发的人起初都会因为关注界面构建器而感到困惑,但是你看懂了,它确实做了很多工作。除了支持可视化设计 GUI 这一显著的好处之外,使用界面构建器还促进了 nib 文件的合理使用,这样可以帮助你更好地遵循 MVC 架构模式。另外,你也可以使应用代码更简洁、更具模块化、更容易编写。正如我们的好朋友 Mark Dalrymple 所说:“不写代码是最好的编程方式!”

## 8.4 分组分区和索引分区

下一个项目将探讨表的另一项基本内容。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 Single View Application 模板创建一个新的 Xcode 项目,这一次将它命名为 Sections。

### 8.4.1 构建视图

打开 Sections 文件夹,单击 Main.storyboard,编辑文件。与之前一样,把表视图拖到 View 窗口中。然后按下 `option+command+6`,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 `option+command+4`,打开属性检查器。把表视图的 Style 从 Plain 改为 Grouped (参见图 8-21)。将表视图的 Tag 属性设置为一个唯一值(本例中是 1),以便之后再重新获得表视图对象。最后像之前那样再次使用 Pin 按钮以设置新表视图的约束,保存分镜然后继续。(本章开头讨论过索引类型和分组类型之间的差别。)

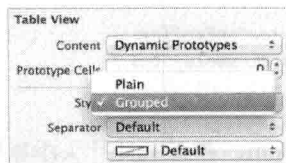


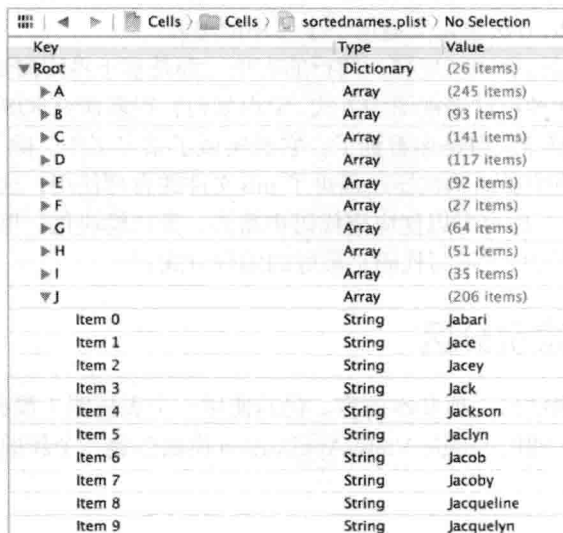
图 8-21 表视图的属性检查器,显示的 Style 弹出窗口中默认选中了 Grouped

### 8.4.2 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,为你节省几小时敲键盘的时间。从本书附带的项目归档文件 08 Sections/Sections 文件夹中找到 `sortednames.plist` 文件,把它添加

到项目的 Sections 文件夹中。

完成添加以后,单击 sortednames.plist,看一下它到底是什么样子(参见图 8-22)。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。



Key	Type	Value
▼ Root	Dictionary	(26 items)
▶ A	Array	(245 items)
▶ B	Array	(93 items)
▶ C	Array	(141 items)
▶ D	Array	(117 items)
▶ E	Array	(92 items)
▶ F	Array	(27 items)
▶ G	Array	(64 items)
▶ H	Array	(51 items)
▶ I	Array	(35 items)
▼ J	Array	(206 items)
Item 0	String	Jabari
Item 1	String	Jace
Item 2	String	Jacey
Item 3	String	Jack
Item 4	String	Jackson
Item 5	String	Jaclyn
Item 6	String	Jacob
Item 7	String	Jacoby
Item 8	String	Jacqueline
Item 9	String	Jacquelyn

图 8-22 sortednames.plist 属性列表文件。展开字母 F 这一项,可以看到字典的内容

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。

### 8.4.3 实现控制器

单击 BIDViewController.h 文件,添加以下粗体显示的代码,这样类就能够遵循 UITableView DataSource 和 UITableViewDelegate 协议:

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

```
@interface BIDViewController : UIViewController
    <UITableViewDataSource, UITableViewDelegate>
@end
```

```
@end
```

现在,切换到 BIDViewController.m,并在文件开头添加以下代码:

```
#import "BIDViewController.h"
```

```
static NSString *SectionsTableIdentifier = @"SectionsTableIdentifier";
```

```
@interface BIDViewController ()
```

```
@property (copy, nonatomic) NSDictionary *names;
```

```
@property (copy, nonatomic) NSArray *keys;
```

```
@end
```

```

@implementation BIDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    //视图加载完成之后（通常是从 nib 文件加载），做一些额外的设置

    UITableView *tableView = (id)[self.view viewWithTag:1];
    [tableView registerClass:[UITableViewCell class]
        forCellReuseIdentifier:SectionsTableIdentifier];

    NSString *path = [[NSBundle mainBundle] pathForResource:@"sortednames"
                                                            ofType:@"plist"];
    self.names = [NSDictionary dictionaryWithContentsOfFile:path];

    self.keys = [[self.names allKeys] sortedArrayUsingSelector:
        @selector(compare:)];
}

```

将以下代码添加到文件末尾@end 声明的上面：

```

#pragma mark -
#pragma mark Table View Data Source Methods
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return [self.keys count];
}

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    NSString *key = self.keys[section];
    NSArray *nameSection = self.names[key];
    return [nameSection count];
}

- (NSString *)tableView:(UITableView *)tableView
    titleForHeaderInSection:(NSInteger)section
{
    return self.keys[section];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:SectionsTableIdentifier
            forIndexPath:indexPath];

    NSString *key = self.keys[indexPath.section];
    NSArray *nameSection = self.names[key];

    cell.textLabel.text = nameSection[indexPath.row];
    return cell;
}

@end

```

上面大部分代码和我们以前看到的没有多大区别。在顶部的类扩展中，我们分别添加了 `NSDictionary` 和 `NSArray` 类型的属性变量。字典将保存所有数据，而数组将保存以字母顺序排列的分区。在 `viewDidLoad` 方法中，我们使用前面声明的标识符注册了每行都会显示的默认表视图单元类。之后我们通过之前添加到项目中的属性列表创建一个 `NSDictionary` 实例变量并将其赋值给 `names` 属性变量。接下来，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，得到一个有序的 `NSArray`。请记住，`NSDictionary` 使用字母表中的字母作为它的键，因此这个数组包含了从 A 到 Z 的 26 个字母。我们通过这个数组来匹配相应的分区。

你可能会注意到，我们省略了一个在前面的示例中都执行过的步骤：我们并没有指定表视图顶部边缘的偏移值。这是因为当你使用分组表视图的时候（就像现在这样），苹果的开发环境会自动将所有内容向下移动一些距离，因此你不需要担心初始表视图的内容会影响到状态栏。

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 很合适了。这一次需要告诉表视图，字典中的每个键都有一个分区。

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return [self.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，所以只返回数组中的行数。这次，我们将以分区为单位返回行数。可以检索相关分区对应的数组，并从该数组返回行数，从而达到目的。

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
    NSString *key = self.keys[section];
    NSArray *nameSection = self.names[key];
    return [nameSection count];
}
```

可以通过 `tableView:titleForHeaderInSection` 方法为每个分区指定一个可选的标题，为简单起见，本例返回这一组对应的字母。

```
- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section
{
    return self.keys[section];
}
```

在 `tableView:cellForRowAtIndexPath:` 方法中，必须从索引路径中获取分区属性和行属性，用它们来确定要使用哪个值。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本章之前构建的 Cells 应用一致。

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，因此最终得到一个带有 26 个分区的分组表，如图 8-23 所示。

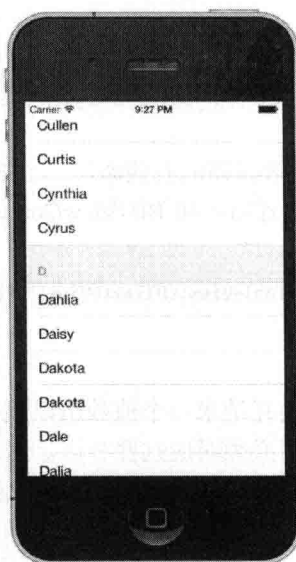


图 8-23 有多个分区的分组表

为进行比较，把表视图再次改为普通风格，然后看一下带有多个分区的无格式表视图是什么样子。在界面构建器中选 `Main.storyboard` 以进行编辑。选中表视图，使用属性检查器将视图改为 `Plain`。保存项目，编译并运行，得到的表数据相同，外观不一样（参见图 8-24）。你会注意到当前的无格式表顶端影响了状态栏，我们之后会对它进行处理。

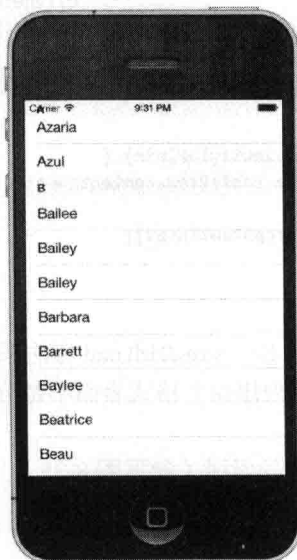


图 8-24 带有分区但是没有索引的无格式表

### 8.4.4 添加索引

当前表的一个问题是行太多了。此列表中有 2000 个名称，要查找 Zachariah 或 Zayne，你的手指会非常累，更别说 Zoie 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改成普通样式了，要添加索引相对来说也很容易。在 BIDViewController.m 文件尾部@end 前面添加如下方法：

```
- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)tableView
{
    return self.keys;
}
```

这就完成了。在这个方法中，委托请求一个值数组以便在索引中显示。表视图有多个分区才能使用索引，而且此数组中的条目必须对应这些分区。返回的数组拥有的条目数必须与拥有的分区数相同，且值必须对应于正确的分区。也就是说，此数组中的第一项对应第一个分区，即分区 0。

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后（通常是从 nib 文件加载），做一些额外的设置
    UITableView *tableView = (id)[self.view viewWithTag:1];
    [tableView registerClass:[UITableViewCell class]
        forCellReuseIdentifier:SectionsTableIdentifier];

    NSString *path = [[NSBundle mainBundle] pathForResource:@"sortednames"
                                                            ofType:@"plist"];
    self.names = [NSDictionary dictionaryWithContentsOfFile:path];

    self.keys = [[self.names allKeys] sortedArrayUsingSelector:
        @selector(compare:)];

    if (tableView.style == UITableViewStylePlain) {
        UIEdgeInsets contentInset = tableView.contentInset;
        contentInset.top = 20;
        [tableView setContentInset:contentInset];
    }
}
```

因为切换到了无格式表，所以还要向 viewDidLoad 方法添加一些代码来修复顶端边缘偏移的问题。现在要让它稍微智能一些。因为想在无格式表而不是分组表中改变偏移，所以代码要判断当前处理的表是哪种类型的。

再次编译并运行，得到一个很好的索引（参见图 8-25）。



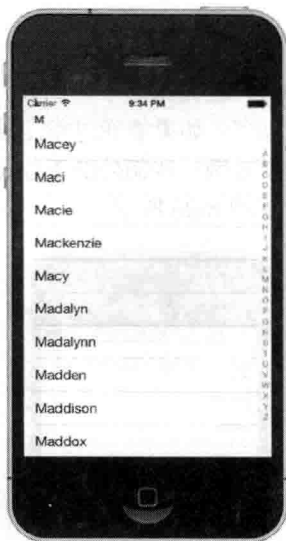


图 8-25 带有索引的表视图

## 8.5 解决状态栏干扰

在体验的时候你一定会发现一个显眼的问题：虽然表视图的顶端偏移了，但只要开始滚动，表视图的内容就会干扰状态栏。目前为止我们构建的所有表视图应用都受到了这个问题影响。但是现在有一个不透明的分区标题总是会贴住屏幕顶端，这样会更加别扭。只要你开始滚动，文本就会穿过分区标题并从状态栏后面滚出。这实在是太丑了！

在下一章中，你将会看到苹果通过使用一种被称为导航控制器的方式，就能够自动地妥善处理这种情况。不过目前我们还没学到它，所以要寻找一种简单的方法能够看起来没有问题。在 `viewDidLoad` 方法中添加以下代码可以创建一个与状态栏相同尺寸的简单 `UIView` 实例，设置为白色并且完全不透明，添加到视图中。因为我们想让它只在非分组表中起作用，所以要将它放在之前添加的 `if` 条件判断代码中。

```
if (tableView.style == UITableViewStylePlain) {
    UIEdgeInsets contentInset = tableView.contentInset;
    contentInset.top = 20;
    [tableView setContentInset:contentInset];

    UIView *barBackground = [[UIView alloc] initWithFrame:CGRectMake(0, 0, 320, 20)];
    barBackground.backgroundColor = [UIColor colorWithWhite:1.0 alpha:0.9];
    [self.view addSubview:barBackground];
}
```

如果你现在运行应用，会发现文字穿过表视图的分区标题并滚出来时，几乎看不到了。如果你想要尝试不同的不透明度值，可以更改 `alpha` 程度并观看效果。如果你将其设定为 1.0，添加的视图会完全不透明，滚动文字也看不到了；设定为 0.0 则会使它完全透明。

## 8.6 实现搜索栏

索引很有用，即便如此，这里的名称还是太多了。例如，如果要查看 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。当然，实现搜索需要做一些额外的工作，但并不是太难。我们将使用搜索显示控件实现一个标准的 iOS 搜索栏，如图 8-26 所示。

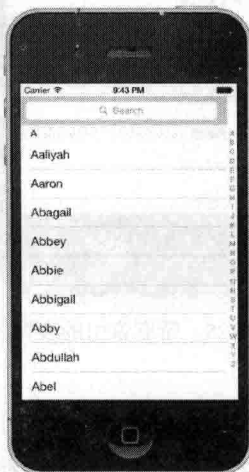


图 8-26 带有搜索栏的应用

首先需要更新 BIDViewController.h，使视图控制器类遵循 UISearchDisplayDelegate 协议。

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

```
@interface BIDViewController : UIViewController
    <UITableViewDataSource, UITableViewDelegate, UISearchDisplayDelegate>
```

```
@end
```

接下来在视图控制器中添加两个实例变量，其中一个用于保存与搜索关键字相匹配的名字，另一个供 UISearchDisplayController 使用。向 BIDViewController.m 中添加如下代码。

```
@implementation BIDViewController {
    NSMutableArray *filteredNames;
    UISearchDisplayController *searchController;
}
```

接下来，我们需要在 viewDidLoad 方法的末尾进行其他的更改，代码如下所示：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后（通常是从 nib 文件加载），做一些额外的设置
```

```

UITableView *tableView = (id)[self.view viewWithTag:1];
[tableView registerClass:[UITableViewCell class]
 forCellReuseIdentifier:SectionsTableIdentifier];

NSString *path = [[NSBundle mainBundle] pathForResource:@"sortednames"
                                                         ofType:@"plist"];
self.names = [NSDictionary dictionaryWithContentsOfFile:path];

self.keys = [[self.names allKeys] sortedArrayUsingSelector:
             @selector(compare:)];

if (tableView.style == UITableViewStylePlain) {
    UIEdgeInsets contentInset = tableView.contentInset;
    contentInset.top = 20;
    [tableView setContentInset:contentInset];

    UIView *barBackground = [[UIView alloc] initWithFrame:CGRectMake(0, 0, 320, 20)];
    barBackground.backgroundColor = [UIColor colorWithWhite:1.0 alpha:0.9];
    [self.view addSubview:barBackground];
}

filteredNames = [NSMutableArray array];
UISearchBar *searchBar = [[UISearchBar alloc]
                          initWithFrame:CGRectMake(0, 0, 320, 44)];
tableView.tableHeaderView = searchBar;
searchController = [[UISearchDisplayController alloc]
                    initWithSearchBar:searchBar
                    contentsController:self];
searchController.delegate = self;
searchController.searchResultsDataSource = self;
}

```

首先，将 `filteredNames` 初始化为一个空数组。这个变量用来保存基于搜索条件过滤后的结果。之后，创建一个 `UISearchBar` 对象，并将其作为顶部视图添加到表中。顶部视图是表的特殊行，始终显示在表的顶部。接下来创建一个搜索显示控制器（`search display controller`），用于显示搜索到的内容。使用用于输入的搜索栏来初始化搜索控制器，并且把视图控制器类本身作为自己的所有者。同时把当前的控制器设置为搜索显示控制器的委托，以便根据搜索条件对数据进行筛选。最后，将视图控制器设置为搜索结果的数据源，用于显示搜索结果。

搜索显示控制器自带了一个表，但是其中的单元由我们来提供，以便进行显示。需要在 `search DisplayController:didloadSearchResultsTableView:` 委托方法中注册一个表视图单元类。将以下代码添加到 `@end` 声明之前。

```

- (void)searchDisplayController:(UISearchDisplayController *)controller
didloadSearchResultsTableView:(UITableView *)tableView
{
    [tableView registerClass:[UITableViewCell class]
 forCellReuseIdentifier:SectionsTableIdentifier];
}

```

搜索显示控制器和我们自己的表使用同一个视图控制器作为数据源，用于填充表视图，它们调用同样的数据源方法。如果调用者是我们自己的表，那么处理方式与现在一样，如果调用者是

搜索显示控制器的表，那就应该显示相匹配的名字。需要使用表的 `tag` 属性来区分到底是哪个表在调用数据源方法，然后进行恰当的处理。修改数据源方法，如下所示：

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    if (tableView.tag == 1) {
        return [self.keys count];
    } else {
        return 1;
    }
}

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
    if (tableView.tag == 1) {
        NSString *key = self.keys[section];
        NSArray *nameSection = self.names[key];
        return [nameSection count];
    } else {
        return [filteredNames count];
    }
}

- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section
{
    if (tableView.tag == 1) {
        return self.keys[section];
    } else {
        return nil;
    }
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
                             SectionsTableIdentifier
                             forIndexPath:indexPath];

    if (tableView.tag == 1) {
        NSString *key = self.keys[indexPath.section];
        NSArray *nameSection = self.names[key];

        cell.textLabel.text = nameSection[indexPath.row];
    } else {
        cell.textLabel.text = filteredNames[indexPath.row];
    }
    return cell;
}

- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)tableView
{
    if (tableView.tag == 1) {
```

```

        return self.keys;
    } else {
        return nil;
    }
}

```

最后要做的就是实现 `searchDisplayController:shouldReloadTableForSearchString:` 委托方法。将以下代码添加到 `@end` 声明之前。

```

- (BOOL)searchDisplayController:(UISearchDisplayController *)controller
shouldReloadTableForSearchString:(NSString *)searchString
{
    [filteredNames removeAllObjects];
    if (searchString.length > 0) {
        NSPredicate *predicate =
        [NSPredicate
        predicateWithBlock:^(BOOL(NSString *name, NSDictionary *b) {
            NSRange range = [name rangeOfString:searchString
                                options:NSCaseInsensitiveSearch];
            return range.location != NSNotFound;
        }]];
        for (NSString *key in self.keys) {
            NSArray *matches = [self.names[key]
                                filteredArrayUsingPredicate: predicate];
            [filteredNames addObjectsFromArray:matches];
        }
    }
    return YES;
}

```

这个委托方法会在用户每次修改搜索栏中的搜索条件时调用，根据方法返回结果决定是否重新加载匹配结果的显示。目前来说，我们始终返回 YES，但是可以添加更多的逻辑，以便在搜索条件真正发生变化时才重新加载。

首先，将之前的搜索结果清空。

```
[filteredNames removeAllObjects];
```

然后，检查搜索条件是否为空。搜索条件为空时，不显示任何匹配结果。

```
if (searchString.length > 0) {
```

现在，定义了一个谓词，用于判断名字与搜索字符串是否匹配。谓词是一种对象，用于对输入的值进行测试。如果值匹配，就返回 YES；如果不匹配，就返回 NO。这里的测试方式是检查名字中是否包含搜索字符串。如果能够在名字中找到搜索字符串的起始位置，就说明这个名字与搜索字符串匹配。

```

NSPredicate *predicate =
[NSPredicate
predicateWithBlock:^(BOOL(NSString *name, NSDictionary *b) {
    NSRange range = [name rangeOfString:searchString
                                options:NSCaseInsensitiveSearch];
    return range.location != NSNotFound;
}]];

```

最后,对所有的键进行迭代,对于每一个键,都使用这个谓词进行测试,并将匹配的名字添加到 `filteredNames` 数组中。

```
for (NSString *key in self.keys) {
    NSArray *matches = [self.names[key]
                        filteredArrayUsingPredicate:predicate];
    [filteredNames addObjectsFromArray:matches];
}
```

现在可以运行应用并且试着对名字进行过滤,结果如图 8-27 所示。

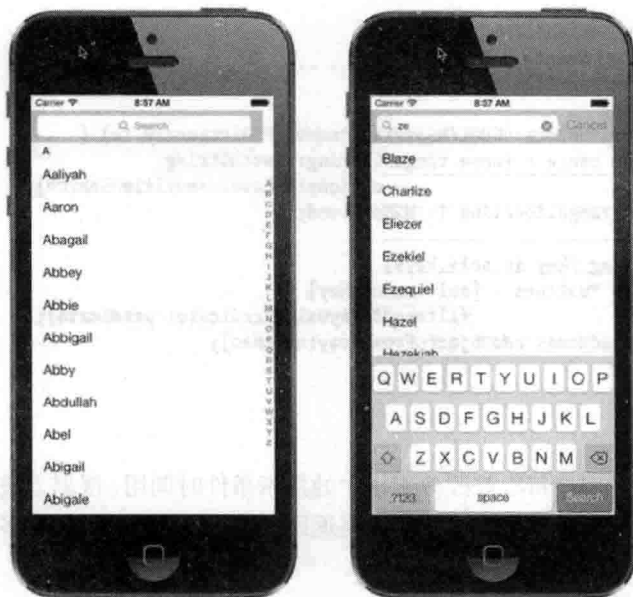


图 8-27 带有搜索栏的表视图应用。请注意在点击搜索栏之前,它在屏幕右侧有些向里缩短了

如你所见,这里有点小问题:搜索栏在右侧边缘看起来像是离奇地切断了。你所看到的实际上是右侧分区索引栏的垂直最顶端。搜索栏是表视图的一部分(因为我们将其设置为顶部视图)。当一个表视图显示分区索引,它会自动从右向内移动其他视图。因为默认的分区索引背景是白色的,与表视图的行颜色混在了一起,这样它在搜索栏旁边就会看起来很违和。

为了修复这个问题,我们可以设置分区索引的颜色。使用反色可以让它在表视图中从上到下都很显眼,用户就能更容易看明白了。你只需要在 `viewDidLoad` 方法的末尾添加以下语句:

```
tableView.sectionIndexBackgroundColor = [UIColor blackColor];
tableView.sectionIndexTrackingBackgroundColor = [UIColor darkGrayColor];
tableView.sectionIndexColor = [UIColor whiteColor];
```

首先我们设置分区索引在用户没有触摸的时候会看到的主要背景颜色,然后设置了滚动背景颜色,它会在用户在边缘触摸并拖动时让整个索引栏高亮,最后设置了索引项本身的文字颜色。图 8-28 展示了最终效果。



图 8-28 使用明显的分区索引，用户会更容易明白它实际上是控件的界面

## 8.7 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，应该能很好地理解表的工作方式，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。你还学习了如何实现搜索栏，在任何呈现大量数据的 iOS 应用中，它都是一个至关重要的工具。要确保真正理解了本章的所有内容，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。你还将学到如何创建内容视图，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查表，在表的行中嵌入控件和删除行。

在上一章中，你已经掌握了如何简单地操作表视图。而在本章中，由于需要学习导航控制器，你将获得更多与表视图相关的练习。

表视图与导航控制器两者作用是紧密配合的。严格来说，并不一定必须用表视图来实现导航控制器。然而在实际情况中，你总会使用到至少一张表（通常会更多）来实现导航控制器，因为导航控制器的强大之处在于它可以轻而易举地处理复杂的层级数据。而在 iPhone 的小屏幕上，利用一系列的表视图是展示层级数据的最理想方式。

本章将逐步构建一个应用，就像之前在第 7 章的选取器应用中所做的那样。先实现可以使用的导航控制器以及根视图控制器，然后再为层级结构添加更多的控制器和层。添加的每一个视图控制器都将会对表的用途以及配置的某些方面进行完善：

- 如何从当前表视图切换到子表视图；
- 如何从当前表视图切换到内容视图，在里面可以浏览详细内容甚至还可以进行编辑；
- 如何将一个表视图拆分成多个分区；
- 如何启用编辑模式从表视图中删除行；
- 如何启用编辑模式使用户可以在表视图中重新排列行的顺序。

信息量有些大吧？那我们先从导航控制器的基础开始讲起吧。

## 9.1 导航控制器

`UINavigationController` 是用于构建层级应用的主要工具。与 `UITabBarController` 在管理以及互相切换各个内容视图的方式类似。两者之间的主要区别在于 `UINavigationController` 是作为栈（stack）来实现的，因此它非常适合用于处理层级结构。

如果你对栈的了解已经胸有成竹，请快速浏览（也可以直接略过）9.1.1 节的内容，然后开始学习 9.1.2 节，如果你是第一次接触栈也不必担心，它是一个非常简单的概念。

### 9.1.1 栈的概念

栈是一种常用的数据结构，采用后进先出的原则。你也许不会相信，佩兹糖果盒就是栈的一个很好的例子。想试试看吗？每个糖果盒都会在使用说明中列出几个简单的步骤。第一步，打开



佩兹糖果盒的包装；第二步，把盒子的卡通头部向后扳开；第三步，用食指和拇指牢牢地捏住糖果栈（注意这里我们巧妙地使用了“栈”这个字），然后用一根小棍儿往里面捅。第四步，捡起散落一地的糖块（哦，看起来这和使用说明上讲得不一样）。

好吧，上面这个例子不算数。不过接下来发生的事情可以形象地说明栈的概念：当捡起散落的糖果，一次一个地把它们塞进糖果盒时，你其实就在操作一个栈。还记得吗？我们说过栈是“后进先出的”，当然也可以说是“先进后出”。放入糖果盒的第一个糖块将是最后一个弹出来的，最后一个被塞入的糖块将会是第一个出来的。栈也遵循同样的规则：

- 向栈中添加对象的操作被称作入栈（push），即把对象推到栈中。
- 第一个入栈的对象叫做栈底（base）。
- 最后一个入栈的对象叫做栈顶（top），下一个被推入的对象将取代它成为新的栈顶。
- 从栈中删除对象的操作被称为出栈（pop）。出栈的那个对象永远是最后被推入栈的。同理第一个入栈的对象也永远是最后一个出栈。

### 9.1.2 控制器栈

导航控制器维护着一个视图控制器栈。在设计导航控制器时，你需要制定用户看到的第一个视图。该视图控制器就是前面几章提到过的根视图控制器（或简称根控制器），它也是导航控制器栈中所有视图控制器的栈底。当用户选择查看下一个视图时，栈中将插入一个新的视图控制器，由它所控制的视图内容也会显示出来。我们把这些新的视图控制器称为子控制器。可以看出，本章的应用 Fonts 就是由一个导航控制器和几个子控制器组成的。

导航栏正中的标题（Title）和左侧的返回按钮（Back Button）如图 9-1 所示。这里，我们利用导航栏控制器栈顶层视图控制器的 title 属性值来定义导航栏的标题，返回按钮的标题则是以之前的视图控制器的 title 属性值来定义。返回按钮和浏览器中的后退按钮作用效果是一样的，当用户点击该按钮时，当前的视图控制器就被推出栈，之前的视图变为当前视图。



图 9-1 iOS 上的设置应用使用了导航控制器。左上角的返回按钮用于将当前视图控制器推出栈外，并返回到层级结构的上一级。还显示了当前内容视图控制器的标题

我们热衷于这种设计模式。可以使用它一层层地构建应用中复杂的层级结构。我们不需要了解整个层级结构。每个控制器只需要知道其子控制器，以便在用户做出操作时把相应的新控制器对象加入到栈中。通过这种方式可以把若干单一的内容组合成一个完整的应用，这正是我们在本章中所要研究的。

导航控制器是许多 iPhone 应用最核心的部分，但在 iPad 应用中，导航控制器的作用则没有那么重要。一个典型的例子就是邮件应用，它实现了层级导航控制器，用户可以通过它在所有的邮件服务器、文件夹以及信息中进行选择。而在 iPad 版的邮件应用中，导航控制器却没有铺满屏幕，而是显示为侧边栏（偶尔会遮住主视图的一部分）。我们将在第 11 章介绍 iPad 专有的 GUI 功能时再详细讲解这一用法。

## 9.2 一个简单的字体浏览器：Fonts

从构建应用的过程中可以了解到为显示层级数据所需要做的基本工作。应用运行后将显示一个包含了 iOS 中所有字体系列的列表，如图 9-2 所示。字体系列是一组风格接近的字体（也有可能是其中的某个字体是另一个的风格变体）。比如说，Helvetica、Helvetica-Bold、Helvetica-Oblique 以及其他变体都是 Helvetica 字体大家族中的一员。



图 9-2 本章应用的根视图。需要注意视图右侧的辅助图标。这种特别的辅助图标则被称为扩展指示器，用户看到它就会知道，触摸这一行则将切换到另一个表视图

当选中这个顶级视图中任意一行时，相应的视图控制器将会被推入到导航控制器的栈中。每行右侧的小图标都被称为辅助图标。这里这种特别的辅助图标（即灰色箭头）则被称为扩展指示器（disclosure indicator），用户如果看到它出现就会知道，触摸这一行将切换到另一个表视图中。

### 9.2.1 子控制器

在开始构建 Fonts 应用前,先快速预览一下每个子控制器所将展示的视图内容。

#### 1. 字体列表控制器

触摸图 9-2 中显示的表里面任意一行,将会切换到图 9-3 中所示的子视图中。



图 9-3 Fonts 应用的第一个子控制器实现了一个表,表中的每一行都包含了一个详情扩展按钮

图 9-3 中每一行右侧的辅助图标和之前的略有不同。它们被称为详情扩展按钮 (detail disclosure button)。与展开标识不一样,详情扩展按钮不仅仅是一个图标,它还是一个用户可以轻点的控件。这意味着你针对给定的行可以有两种不同的操作:当用户选中该行时会触发一个操作;当用户轻点这个按钮时,则会触发另一个操作。点击辅助图标中的小圆形信息按钮,用户就可以浏览 (或许能够编辑) 当前行所对应的更加详细的信息。而辅助图标中右键头则提示了用户,点击这行的任意位置将会导航到更深一层的位置。

#### 2. 字体尺寸视图控制器

触摸图 9-3 中显示的表里的任意一行,将会切换到如图 9-4 所示的子视图中。

下面概述了使用扩展指示器与详情扩展按钮的适当情境:

- ❑ 如果点击某行,仅会打开该行的详情视图,那么请不要使用辅助图标;
- ❑ 如果点击某行,会切换到一个带有更多项列表的新视图 (不是详情视图),那么可以使用扩展指示器 (即右键头);
- ❑ 如果想要为某行提供两种操作,则可以为该行添加详情扩展指示器或详情按钮,这样用户就能够点击该行切换到新的视图或点击详情按钮以获取更多详情。



图 9-4 当前位于字体列表视图控制器的下一层：字体尺寸视图控制器。控制器中的每一行都展示了所选字体的各种尺寸大小

### 3. 字体信息视图控制器

图 9-5 中展示的是我们应用的最后一个（也是唯一不是表视图的）子控制器。轻点图 9-2 中字体列表控制器任意一行上的信息图标，就会出现这个视图。



图 9-5 Fonts 应用的最后一个视图控制器，可以让所选字体以想要的任何尺寸显示出来

这个视图可以让用户通过拖动滑动条以调整显示字体的尺寸。它还包含一个让用户指定此字体是否会出现于用户收藏中的开关。如果有字体被设置为收藏状态,它们将在根视图控制器中作为独立的一个分组出现。

## 9.2.2 Font应用的基础框架

Xcode 为创建基于导航器的应用提供了一个极好的模板,以后创建层级应用时可能会经常用到它。不过我们现在不会用到这个模板,而是从零开始构建一个基于导航器的应用,以便于读者了解所有内容是如何协作的。我们将全面地概括所有知识,你应该能够轻松地跟上节奏。

在 Xcode 中,按下 `command+shift+N` 创建一个新的项目,在 iOS Application 模板列表中选择 Empty Application,然后点击 Next 按钮继续。将 Product Name 设为 Fonts, Organization Name 设为 Apress, Company Identifier 设为 com.apress, Class Prefix 设为 BID。同样,确保没有勾选 Use Core Data 复选框,将 Devices 设为 iPhone,之后点击 Next 按钮并选择项目要存储的位置。

选择项目导航器并检查 Fonts 文件夹,你会看到此模板除了应用委托并没有提供任何其他东西。此时还没有视图控制器和导航控制器。

要让应用运行,我们需要添加一个导航控制器,其中包含一个导航栏。我们还需要添加一系列视图和视图控制器供导航栏显示。其中第一个视图就是图 9-2 中所示的顶级视图。

### 1. 创建分镜

应用 GUI 界面由单个分镜来容纳。这是管理预定义层级的视图控制器最好、最简洁的方法。按下 `command+N` 打开文件创建向导并在 User Interface 部分选择 Storyboard。在之后弹出的界面中将 Device Family 设置为 iPhone,然后在后面的界面中为文件命名 Main.storyboard 并保存到你的项目中。

现在你需要确保应用是从分镜中加载 GUI 配置信息,请在项目导航栏中选择顶级的 Fonts 项。在编辑区域左侧的 Targets 位置处选中 Fonts 条目。接下来在 General 分页下找到 Deployment Info 区域,你会看到一个 Main Interface 的下拉菜单。选择 Main.storyboard 就可以完成这一部分的配置了。

完成之后,你需要从应用委托中移除在没有加载分镜或 nib 文件时的基本 GUI 设置代码,请选择 BIDAppDelegate.m 文件。第一个方法是我们熟悉的 `application:didFinishLaunchingWithOptions:` 方法,之前曾遇到过。删除该方法内几乎全部的代码,只留下最后一行用来返回 YES 的语句。

### 2. 设置导航控制器

现在我们需要创建应用导航器的基础架构。其核心之处就是 UINavigationController,它管理着用户可以切换的视图控制器栈,和一个我们用来显示顶级列表的 UITableViewController。通过界面构建器可以很轻松地做到这些。

选中 Main.storyboard 并使用对象库搜索 UINavigationController。拖动它到编辑区域中,你将会看到实际上得到的并不只是一个场景,而是两个,这和第 7 章中创建分页视图控制器的情况非常类似。在左侧的是 UINavigationController,它有一个连接指向了第二个场景,其中包含了一个 UITableViewController。你会看到表的标题是 Root View Controller。点击标题,调出属性检

查器，并将标题设置为 Fonts。

可以停下来稍微想一想。通过配置应用从分镜中加载初始场景能得到什么？首先，我们得到了由导航控制器创建的视图，它集成了顶部导航栏（通常包含各种标题和在左侧出现的某些返回按钮）和导航控制器中当前视图控制器中显示的任何内容。在我们的示例中，底下显示的所有内容将是与导航控制器一同创建的表视图。

在阅读过程中，你还将学到更多关于如何控制导航控制器中导航条中显示的内容，还将了解到导航控制器如何将焦点从一个子视图控制器切换到其他控制器。现在已经有了足够的准备工作，可以开始定义自定义视图控制器了。

这时的应用框架实质上已经完成了。存储所有文件，然后构建并运行应用。如果一切正常，应用将会启动，而一个标题为 Fonts 的导航条也会出现。你还没有给表视图任何关于需要显示的信息，所以此时没有行会显示出来（如图 9-6 所示）。



图 9-6 应用基础框架的效果

### 3. 收藏喜爱的字体

在应用中，用户可以维持一个收藏字体列表，可以添加自己喜欢的字体，浏览所有已经选好的收藏字体，也可以从列表中移除它们。为了能使用一致的方式管理这个列表，我们将要创建一个新类，它包含了一个收藏内容的数组并将它们存储到应用的用户偏好设置中。你将会在第 12 章中学到更多关于用户偏好设置的内容，不过这里我们只会接触到一些皮毛。

现在开始创建一个新类。在项目导航栏中选中 Fonts 文件夹并按下 `command+N` 调出新建文件向导。在左侧面板中选择 Cocoa Touch，再选择 Objective-C class，然后点击 Next 按钮。在接下来的界面中，选择从 NSObject 继承子类并将新类命名为 BIDFavoritesList。创建完这个类的文件之后，选择 BIDFavoritesList.h 并添加如下所示的粗体代码：

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

@interface BIDSFavoritesList : NSObject

+ (instancetype)sharedFavoritesList;

- (NSArray *)favorites;

- (void)addFavorite:(id)item;
- (void)removeFavorite:(id)item;

@end
```

在上面的代码片段中，我们声明了新类的 API 接口。首先我们声明了一个名称为 `sharedFavoritesList` 的工厂方法，它会返回一个类的实例。无论什么时候调用这个方法，总是会返回同一个实例。这是因为 `BIDSFavoritesList` 没有使用多个实例，而是以单例的方式运行，我们在应用中只会用到一个实例。

**注意** 你可能难以理解 `sharedFavoritesList` 声明返回的类型：`instancetype`。这是 Objective-C 最近才新增的特性。推荐所有以前使用 `id` 作为返回类型的工厂方法和 `init` 方法应改用 `instancetype` 为返回类型。使用 `id` 类型会引起类型安全问题。在过去，你总是会头脑发热写出像 `NSString *s = [NSArray array]` 这样的代码，而编译器不会警告（虽然之后当你创建的 `NSArray` 对象发送 `NSString` 类专有的方法时会引发崩溃）。使用 `instancetype` 能帮你保证一定程度上的通用性，它会始终要求编译器令返回值类型必须是消息接受者（或其子类）的类型。

`instancetype` 现在还未使用在所有的苹果代码中，不过 iOS 7 的 SDK 中一些类已经更新并采用它了。事实上，像上面那个例子中，错误的数组分配现在会生成一个编译器警告，因为 `NSArray` 类的 `array` 方法现在是以 `instancetype` 作为返回类型的。鉴于这个特性未来会用得更多，所以我们现在就该把习惯转变过来。

接下来要定义用来访问数组以及添加删除项目的方法。

现在我们想要切换到 `BIDSFavoritesList.m` 文件并开始实现代码。首先我们在顶部的类扩展中添加这个属性变量：

```
#import "BIDSFavoritesList.h"

@interface BIDSFavoritesList ()

@property (strong, nonatomic) NSMutableArray *favorites;

@end
```

请注意我们声明了一个 `NSMutableArray` 类型名为 `favorites` 的属性变量。在头文件中，我们也声明了一个名为 `favorites` 但返回 `NSArray` 的方法。由于声明一个属性变量便会自动声明 `getter` 和 `setter` 方法，这样做不会引起冲突吗？好在不必担心。因为这个属性类型是头文件所用类型

的子类，所以它能够正常运行。这意味着可以在类里面使用这个可变数组，但是我们希望通过 API 公开的是一个看似不可变的 NSArray 数组。所有使用这段代码的人都应该对这个类所提供的 API 有一致的理解（约定）。如果有些代码过于深入，从而发现这个类实际上返回的是一个 NSMutableArray，继而跳过 API 直接使用它的话，就会破坏这种约定，这可不是我们的问题。

继续往下，此处是 sharedFavoritesList 工厂方法：

```
+ (instancetype)sharedFavoritesList {
    static BIDFavoritesList *shared = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        shared = [[self alloc] init];
    });
    return shared;
}
```

这段代码看起来可能很复杂，不过它实际上只做了一件事：它创建了一个类的实例，并将其返回。传递给 dispatch\_once() 函数的代码块中包含了用来创建的代码部分。这样做可以确保这些代码只会运行一次。首次调用这个方法之后，每当再调用它，由于实例已经创建了，所以会直接返回。

现在我们来看看 init 方法：

```
- (instancetype)init {
    self = [super init];
    if (self) {
        NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
        NSArray *storedFavorites = [defaults objectForKey:@"favorites"];
        if (storedFavorites) {
            self.favorites = [storedFavorites mutableCopy];
        } else {
            self.favorites = [NSMutableArray array];
        }
    }
    return self;
}
```

这个方法使用 NSUserDefaults 类（详情参见第 12 章）来判断偏好设置里是否已经有收藏的内容。如果有的话，将收藏内容复制成一个可变副本并赋给 favorites 属性，否则赋给它一个空的可变数组。

剩下的就是实现添加和移除收藏项的方法，以及两者都会调用的一个方法，用来快速存储更改：

```
- (void)addFavorite:(id)item {
    [_favorites insertObject:item atIndex:0];
    [self saveFavorites];
}

- (void)removeFavorite:(id)item {
    [_favorites removeObject:item];
    [self saveFavorites];
}
```



```
- (void)saveFavorites {
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    [defaults setObject:self.favorites forKey:@"favorites"];
    [defaults synchronize];
}
```

addFavorite: 和 removeFavorite: 都非常简单。这里唯一值得注意的是, 我们没有通过 self.favorites 来访问数组(全书中都优先使用这种方式), 而是访问底层的实例变量 favorites。这样做的原因很巧妙: 虽然在类扩展中将属性变量定义为 NSMutableArray 类型, 但编译器会在处理 self.favorites 时发现头文件的@interface 中声明的是一个不可变的 NSArray! 这种变相的解决办法与我们平常的编码风格不一样, 不过能正常使用就行。

这些方法都调用了 saveFavorites 方法, 它使用 NSUserDefaults 类将数组存储到用户的偏好设置中。第 12 章将介绍更多相关的工作原理, 不过现在只需知道在这里获取的 defaults 对象就像是一种永久性字典, 放入其中的任何东西在下次访问时仍将存在, 即便应用曾经停止或重启过。

## 9.3 创建根视图控制器

现在我们要开始创建第一个视图控制器。上一章使用一个简单的字符串数组来作为表的行。这里也使用类似的方法, 不过这次将使用 UIFont 类来获取字体系的列表, 然后使用这些字体系列的名称作为每一行。我们还将使用字体本身格式来显示字体名称, 这样每行都将能够简单预览字体系列的内容。

开始为应用创建第一个控制器类。选中项目导航栏中的 Fonts 文件夹并按下 command+N 以调出新建文件向导。在左侧面板选择 Cocoa Touch, 再选择 Objective-C 类, 然后点击 Next 按钮。在之后的界面的 Subclass of 文本框中输入 UITableViewController, 并将新类命名为 BIDRootViewController。点击 Next 按钮, 再点击 Create 按钮创建新类。最后选择 BIDRootViewController.m 并添加以下粗体所示的代码片段, 来为收藏列表导入头文件并添加一些属性变量:

```
#import "BIDRootViewController.h"
#import "BIDFavoritesList.h"

@interface BIDRootViewController ()

@property (copy, nonatomic) NSArray *familyNames;
@property (assign, nonatomic) CGFloat cellPointSize;
@property (strong, nonatomic) BIDFavoritesList *favoritesList;

@end
```

一开始就会给这些属性变量赋值, 之后会在类中多次使用它们。familyNames 数组将包含所要显示的所有字体系的列表; cellPointSize 表示想要在表视图单元中使用的字体大小, favoritesList 将包含指向 BIDFavoritesList 单例的指针。

**注意** 你可能会注意到 `familyNames` 属性变量使用的是 `copy` 关键字而非 `strong`。这是为什么呢？为什么非要复制数组？这是因为它有可能会是可变数组，这么做就是为了避免这种情况。想象一下，如果我们使用 `strong` 来声明属性变量，而外部的某段代码传入了一个 `NSMutableArray` 类型的实例来设置 `familyNames` 的值。假如此调用者之后要更改数组的内容，`familyNames` 将与屏幕上的内容不再同步，`BIDRootViewController` 实例会变得不一致！使用 `copy` 可以消除这个危险，因为对 `NSArray`（也包括任何可变的子类）调用 `copy` 总是会得到一个不可变的副本。此外我们不需要过多担心性能问题。事实上，给不可变对象发送 `copy` 并不会复制对象。而是添加一个引用计数后再返回原值。其实对不可变对象调用 `copy` 和调用 `retain` 的效果一样（你在设定一个 `strong` 属性变量时，ARC 就会在后台调用 `retain`）。因此这样做对象便不会被改变，这对所有人都有效。

这种情况适用于所有基类不可变，但有可变质类的值类（value classes）。这些值类包括 `NSArray`、`NSDictionary`、`NSSet`、`NSString` 以及 `NSData` 等。如果想用这些类的实例作为属性，可能需要使用 `copy` 来声明这些属性的值来避免麻烦，而不能用 `strong`。

在 `viewDidLoad` 方法中添加如下粗体显示的代码以设置全部的类型属性变量：

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

    self.familyNames = [[UIFont familyNames]
                      sortedArrayUsingSelector:@selector(compare:)];
    UIFont *preferredTableViewFont = [UIFont preferredFontForTextStyle:
                                     UIFontTextStyleHeadline];
    self.cellPointSize = preferredTableViewFont.pointSize;
    self.favoritesList = [BIDFavoritesList sharedFavoritesList];
}
```

在上面的代码中，`familyNames` 中放置了从 `UIFont` 类获取的所有已知字体名称，并对结果数组进行排序。然后再次使用 `UIFont` 获取了在标题处使用的字体。通过 iOS 7 的新功能可以得到设置应用中指定的字体大小。这个动态字体尺寸功能能让用户设置一个系统的全局字体大小。在这里我们使用字体的 `pointSize` 来建立一个在视图控制器中都会采用的标准字体大小。最后我们还要获取收藏列表单例。

在继续之前，我们先删除 `initWithStyle:` 和 `didReceiveMemoryWarning` 方法，以及所有被注释的表视图委托和数据源方法（通常是 `tableView:canEditRowAtIndexPath:` 方法、`tableView:commitEditingStyle:` 方法、`tableView:moveRowAtIndexPath:toIndexPath:` 方法和 `tableView:canMoveRowAtIndexPath:` 方法），我们不会在这个类中用到它们。

这个视图控制器的目的是显示两段分区。第一个分区是全部有效字体系列的列表，每一项代表这个系列中的所有字体。第二个分区是收藏列表，它只包含了一个项目，用户通过它可以看到收藏的字体。不过，假如没有用户喜欢的收藏项目（例如应用第一次启动的时候），将不会显示第二个分区，因为用户通过它得到的只是一张空列表。因此我们还有一些工作要做，才能最终完

成这个类。首先是实现这个方法：

```
- (void)viewWillAppear:(BOOL)animated {
    [self.tableView reloadData];
}
```

之所以要这样做是因为，当把一个视图切换到另一个时，需要显示的内容可能也会发生改变。比如说，用户一开始时没有收藏项，但当它切换到视图后，浏览某个字体并将其设置为收藏项，然后回到根视图。此时我们需要重新加载表视图，这样第二个分区便会显示出来。

接下来我们要实现在该类中使用的一个工具方法。很多时候，例如通过数据源方法配置表视图时，需要能够计算出表单元中需要显示哪种字体。将用下面这个方法来实现此功能：

```
- (UIFont *)fontForDisplayAtIndexPath:(NSIndexPath *)indexPath {
    if (indexPath.section == 0) {
        NSString *familyName = self.familyNames[indexPath.row];
        NSString *fontName = [[UIFont fontNamesForFamilyName:familyName]
                               firstObject];
        return [UIFont fontWithName:fontName size:self.cellPointSize];
    } else {
        return nil;
    }
}
```

前面的方法使用了 UIFont 类，首先根据已有的字体系列名称找到所有的字体名称，然后获取系列中首个字体的名称。不需知道字体系列中的首个字体来代表整个系列是否最为合适，姑且就这样认为吧。

现在把注意力集中到视图控制器的主体：表视图数据源方法。首先我们来看看分区的数量：

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    #warning Potentially incomplete method implementation.
    // 返回分区数
    if ([self.favoritesList.favorites count] > 0) {
        return 2;
    } else {
        return 1;
    }
    //return 0;
}
```

我们根据收藏列表来判断是否要显示第 2 个分区。接下来，我们要处理每个分区的行数：

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
    #warning Incomplete method implementation.
    // 返回分区中的行数
    if (section == 0) {
        return [self.familyNames count];
    } else {
        return 1;
    }
    //return 0;
}
```

这段代码也很简单。我们只需根据分区数的索引来判断分区究竟是显示所有字体系列的名称，还是收藏列表的单个表单元。下面我们来定义另一个方法，UITableViewDataSource 协议中的一个可选方法，它能使我们指定每个分区的标题。

```
- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section {
    if (section == 0) {
        return @"All Font Families";
    } else {
        return @"My Favorite Fonts";
    }
}
```

这段代码实现了另一个简单的方法。它根据分区索引决定使用什么作为标题。最后是每个表视图都必须实现的核心数据源方法，用来配置每个表单元，代码如下所示：

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifierWithIdentifier:CellIdentifier
                                                                    forIndexPath:indexPath];

    static NSString *FamilyNameCell = @"FamilyName";
    static NSString *FavoritesCell = @"Favorites";
    UITableViewCell *cell = nil;

    // 配置表单元
    if (indexPath.section == 0) {
        cell = [tableView dequeueReusableCellWithIdentifierWithIdentifier:FamilyNameCell
                                                                    forIndexPath:indexPath];
        cell.textLabel.font = [self fontForDisplayAtIndexPath:indexPath];
        cell.textLabel.text = self.familyNames[indexPath.row];
        cell.detailTextLabel.text = self.familyNames[indexPath.row];
    } else {
        cell = [tableView dequeueReusableCellWithIdentifierWithIdentifier:FavoritesCell
                                                                    forIndexPath:indexPath];
    }

    return cell;
}
```

这段代码中定义了两个不一样的表单元标识符，用它们指向分镜中的两个不同的表单元。现在还没有开始配置，不过很快就会了。接着根据分区索引来判断针对当前 indexPath 索引路径将显示哪个表单元。如果表单元用来容纳字体系列名称，则要在 label 和 detailLabel 上显示字体系列名称。还在文本标签中使用了系列中的字体（通过 fontForDisplayAtIndexPath:方法获取），这样就可以让字体系列名称以本身的字体显示出来，此外还有一个较小的标准系统字体。

## 9.4 初始化分镜

现在已经实现了一个视图控制器。为了让其显示一些内容，需要设置分镜。在项目导航栏中

选择 Main.storyboard。你将看到我们之前添加的导航控制器和表视图控制器。我们首先需要配置表视图控制器。默认的控制类是 `UITableViewController`。需要更改它并连接输出接口，这样表视图就能成功加载。

点击场景以选中表视图控制器，然后选中代表视图控制器的黄色图标。使用身份检查器将视图控制器的 Class 更改为 `BIDRootViewController`。之后按住鼠标右键，将鼠标指针从表视图拖到控制器图标上，在弹出的面板中选择 `dataSource`，再次重复此操作并选择 `delegate`。这样你便将表视图的两个输出接口连接到控制器，之后它就能从中获取到表单元的信息了。

我们现在还需要配置两个表单元样本，与代码中会用到的单元标识符相配。表视图在一开始就拥有一个表单元样本。选中它并按下 `command+D` 键对它进行复制，之后将看到两个表单元。选中第 1 个表单元，使用特性检查器将其 Style 设为 `Subtitle`，Identifier 设为 `FamilyName`。接下来选择第 2 个表单元样本，将其 Style 设为 `Basic`，Identifier 设为 `Favorites`。还需要双击表单元中的标题并将文本中的 Title 改为 `Favorites`。

现在构建应用，并在设备或模拟器上运行它，应该会看到一张不错的字体列表。不过这里有点问题！可能在向下滚动到 `Damascus` 字体时就已经发觉了，不过请先继续看下去，直到可以看见底部的 `Zapfino` 字体。你可能会看到类似如图 9-7 所示的样子。

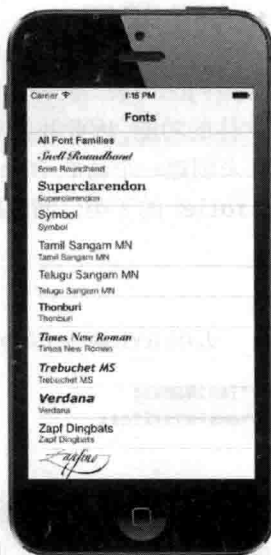


图 9-7 某些字体比其他字体要大

这是因为 `Zapfino` 以及列表中其他一些字体尺寸过高，无法全部显示出来。虽然我们指定了所有字体使用同样的点数尺寸，但事实上每种字体都可能会有一些字符的大小超出预期范围。

如果看过了列表中所有字体的小写字母，我们会发现小写字符大致都是同样的高度。你可能还会注意到一些字体中尺寸过高的字母与其他字符相比，其位置略微靠下。在印刷行业中，这些垂直的超出距离被称为升部（`ascender`）和降部（`descender`）。幸运的是，`UIFont` 类为我们提供了

检测这些字体高度的方法。而且 `UITableViewDelegate` 协议中包含了一个方法，我们可以通过实现它来为每个表单元指定不同的高度，这样就可以分别计算出合适的高度出来。请在 `BIDRootViewController.m` 中添加这个方法：

```
- (CGFloat)tableView:(UITableView *)tableView
heightForRowAtIndexPath:(NSIndexPath *)indexPath {
    if (indexPath.section == 0) {
        UIFont *font = [self fontForDisplayAtIndex:indexPath];
        return 25 + font.ascender - font.descender;
    } else {
        return tableView.rowHeight;
    }
}
```

这个方法通过计算字体的 `ascender` 和 `descender` 属性之间的差异，算出了表视图第一个分区中每个单元的高度。我们给这个数字加上 25 来为下方的详情文本标签腾出显示区域。这里的 25 仅仅是多次尝试后而确定的一个数字。因为详情文本标签总是同样的高度，这样做就能正常显示了。

添加完这个方法后，你可以再次构建并运行，你将会看到每行现在都已经能完美地适配里面字体的尺寸了。非常好！

## 9.5 第一个子控制器：字体列表视图

我们的应用当前只显示了一个字体系列的列表，没有其他的内容。我们想添加让用户点击某个字体系列就可以看到其中包含所有字体的功能，因此我们要创建一个新的视图控制器来管理此字体列表。通过 Xcode 的新建文件向导来创建一个新的 Objective-C 类，将其命名为 `BIDFontListViewController` 并作为 `UITableViewController` 的子类。创建好这个类后，选择它的头文件并添加以下属性变量：

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

@interface BIDFontListViewController : UITableViewController

@property (copy, nonatomic) NSArray *fontNames;
@property (assign, nonatomic) BOOL showsFavorites;

@end
```

`fontNames` 属性变量用来告诉这个视图控制器显示何种内容。另外，还创建了一个 `showsFavorites` 属性变量，将通过它让视图控制器知道显示的是收藏列表，而不是字体系列列表——它在后面会非常有用。

现在切换到 `BIDFontListController.m`，并删除 `initWithStyle:` 和 `didReceiveMemoryWarning` 方法，这里不会用到它们。接下来需要在文件顶部导入一个头文件并声明一个属性变量：

```
#import "BIDFontListViewController.h"
#import "BIDFavoritesList.h"

@interface BIDFontListViewController ()
```

```
@property (assign, nonatomic) CGFloat cellPointSize;
```

```
@end
```

我们将使用 `cellPointSize` 属性变量存储每个字体的优先显示尺寸，再次通过 `UIFont` 找到优先尺寸。在 `viewDidLoad` 方法中通过以下方式实现：

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

    // 将下面代码注释掉，以便保留显示选择
    // self.clearsSelectionOnViewWillAppear = NO;

    // 注释掉下列代码，以便在这个 view controller 的导航栏中显示编辑按钮
    // self.navigationItem.rightBarButtonItem = self.editButtonItem;

    UIFont *preferredTableViewFont = [UIFont preferredFontForTextStyle:
        UIFontTextStyleHeadline];
    self.cellPointSize = preferredTableViewFont.pointSize;
}
```

这时我们首先要创建一个简单的工具方法来选择每行的字体，与我们在 `BIDRootViewController` 中所做的类似。但这里有些区别，在这个视图控制器中我们要得到字体的名称列表，而不是字体系列的列表，使用 `UIFont` 类来获取每个字体的名称，就像这样：

```
- (UIFont *)fontForDisplayAtIndex:(NSIndexPath *)indexPath {
    NSString *fontName = self.fontNames[indexPath.row];
    return [UIFont fontWithName:fontName size:self.cellPointSize];
}
```

现在要以实现 `viewWillAppear:` 方法的形式进行其他改动。你是否还记得，在 `BIDRootViewController` 中，实现这个方法是为了在收藏项变动时进行刷新？现在也是同样道理。这个视图控制器可能会显示收藏列表，而且用户可能会切换到另一个视图控制器，更改收藏项（我们之后会进行处理），然后再回到这里。我们需要重新加载表视图，这个方法会对其进行处理：

```
- (void)viewWillAppear:(BOOL)animated {
    if (self.showsFavorites) {
        self.fontNames = [BIDFavoritesList sharedFavoritesList].favorites;
        [self.tableView reloadData];
    }
}
```

其基本原理是，在进行平常操作时，视图控制器会在显示出来之前传入一个字体名称列表，只要视图控制器仍在，该列表就将始终保存不变，并且这个视图控制器有时要负责重新加载它的列表。

接着，删除整个 `numberOfSectionsInTableView:` 方法。此处只有一个分区，只要忽略这个方法就相当于实现它并返回了 1。接着要实现另外两个主数据源方法，就像这样：

```
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
```



```

{
#warning Incomplete method implementation.
// 返回分区的行数
return [self.fontNames count];
return 0;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    static NSString *CellIdentifier = @"FontName";
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier
        forIndexPath:indexPath];

    // 配置表单元
    cell.textLabel.font = [self fontForDisplayAtIndexPath:indexPath];
    cell.textLabel.text = self.fontNames[indexPath.row];
    cell.detailTextLabel.text = self.fontNames[indexPath.row];

    return cell;
}

```

第一个方法不需要解释，第二个应该也不需要——与 BIDRootViewController 中的做法类似，而且还更简单一些。

你应该还记得，不同的字体拥有不同的显示高度。为了让每个表视图单元的高度都合适，需要像 BIDRootViewController 那样实现同样的委托方法：

```

- (CGFloat)tableView:(UITableView *)tableView
  heightForRowAtIndexPath:(NSIndexPath *)indexPath {
    UIFont *font = [self fontForDisplayAtIndexPath:indexPath];
    return 25 + font.ascender - font.descender;
}

```

我们之后要向这个类添加更多的内容，不过首先要让它能运行。为此我们还需要配置一些分镜，并对 BIDRootViewController 做一些修改。请切换到 Main.storyboard 开始吧。

### 9.5.1 设定字体列表的分镜

现在分镜的导航控制器中拥有一个表视图控制器。需要添加一个新的层来显示字体列表，因此在对象库中找到 Table View Controller（表视图控制器），拖出一个放在编辑区域，可以放在已有的表视图控制器右边。选择新的表视图控制器并使用身份检查器将它的类设置为 BIDFontListViewController。确保表视图的委托和数据源都连接到了控制器。在表视图中选择单元样本并打开特性检查器以做出一些调整。将它的 Style 改为 Subtitle，Identifier 设为 FontName，而它的 Accessory 改为 Detail Disclosure。使用详情扩展按钮可以让这类行能对两种轻点作出响应，这样用户就可以根据轻点的是扩展还是其他位置而触发两种不同的操作。

有一种方法，可以让用户在一个视图控制器内操作，能够实例化并显示另一个视图控制器，



那就是创建一个在两者之间相连的转场（segue）。转场实际上是一种“过渡”，作家和电影人有时会用它来描述一种能使段落或场景平滑地与后续内容衔接起来的行为。苹果本来可以直呼其名为“过渡”，不过在 UIKit 的 API 接口中随处可见过渡（transition）一词，所以苹果可能为了避免混淆而使用了一个独特的术语。需要告诉你的是，segue 与个人用交通工具 Segway 的发音完全一样（现在你可能终于明白它为什么要叫 Segway 了吧）。

通常转场完全是在界面构建器中创建的。运行原理是某个场景中的操作可以触发转场去加载并显示另一个场景。如果正在使用导航控制器，转场可以将下一个控制器自动推入导航栈。我们将在应用中使用这个功能。现在就开始吧！

为了让根视图控制器中的表单元可以显示出字体列表视图控制器，需要创建一对转场来连接两个场景。只需要按住鼠标右键，将鼠标指针从第一个单元样本拖到新场景上，鼠标悬停时将看到整个场景会高亮显示，表示可以连接了。释放鼠标右键并选择弹出的浮动菜单中 Selection Segue 位置的 push 选项。现在对另一个单元样本执行同样的操作。创建这些转场意味着，只要用户轻点这些表视图，另一端相连的视图控制器便会分配内存空间并准备切换。

## 9.5.2 对根视图控制器的转场进行设置

保存更改并切换回 BIDRootViewController.m。请注意不是刚刚用到的类 BIDFontListViewController，而是它的“父”控制器。你需要在这里响应用户在根表视图中的触摸行为，让新的 BIDFontListViewController（由你刚刚创建的其中一个转场所指定）可以显示，并传递需要显示的值。首先要导入新类的头文件：

```
#import "BIDRootViewController.h"
#import "BIDFavoritesList.h"
#import "BIDFontListViewController.h"
```

新视图控制器实际的准备工作是由 prepareForSegue:sender: 方法来执行的。可以在 @implementation 部分的底部找到这个方法，它在这个文件创建时会以注释的状态存在。移除注释标记，并用如下的方式来实现这个方法：

```
/*
#pragma mark - Navigation

// 在基于分镜的应用中，经常需要在导航间作一些准备工作
// 将选定对象传入新的视图控制器
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    // 使用 segue.destinationViewController 获取新的视图控制器
    // 将选定对象传入新的视图控制器
    NSIndexPath *indexPath = [self.tableView indexPathForCell:sender];
    BIDFontListViewController *listVC = segue.destinationViewController;

    if (indexPath.section == 0) {
        NSString *familyName = self.familyNames[indexPath.row];
        listVC.fontNames = [[UIFont fontNamesForFamilyName:familyName]
                           sortedArrayUsingSelector:@selector(compare:)];
    }
}
```

```

        listVC.navigationItem.title = familyName;
        listVC.showsFavorites = NO;
    } else {
        listVC.fontNames = self.favoritesList.favorites;
        listVC.navigationItem.title = @"Favorites";
        listVC.showsFavorites = YES;
    }
}

*/

```

这个方法通过 sender（即轻点的 UITableViewCell）来判断哪一行被点击，并向 segue 请求它的 destinationViewController，即要显示的 BIDFontListViewController 实例。然后根据用户轻点的是字体系列（section 为 0）还是收藏单元（section 为 1）直接向新的视图控制器传入一些值。我们不仅要设置这个特定视图控制器的自定义属性变量，还要访问每个控制器的 navigationItem 属性以设置它的 title 标题。

navigationItem 属性是 UINavigationController 类的一个实例，这是一个 UIKit 类，它包含了显示在视图控制器的导航条上内容相关的信息。

现在运行应用，你将看到点击字体系列的任一名字将显示出它包含所有的独立字体列表，如图 9-3 所示。

## 9.6 创建字体尺寸视图控制器

应用的功能还没有全部完成。图 9-4 和图 9-5 展示了能以多种方式浏览所选字体的界面，不过现在还没实现。但很快就会了！下面要创建一个如图 9-4 所示的视图，它一次展示了多个字体尺寸。使用 Xcode 的新建文件向导创建一个 UITableViewController 子类的新 Objective-C 类，并命名为 BIDFontSizesViewController。这个类只需要父控制器传入的字体参数，你需要在 BIDFontSizesViewController.h 中添加它，就像这样：

```

#import <UIKit/UIKit.h>

@interface BIDFontSizesViewController : UITableViewController

@property (strong, nonatomic) UIFont *font;

@end

```

现在切换到 BIDFontSizesViewController.m。这是一个非常简单的表视图控制器，只实现了一些基本的表视图委托和数据源方法，还有几个私有内部方法。首先删除 initWithStyle: 方法、viewDidLoad 方法、didReceiveMemoryWarning、numberOfSectionsInTableView: 方法，以及所有位于底部被注释的方法。这次同样用不到它们。

所需的是两个内部私有方法。其中一个将返回一个点数列表，它定义了所选字体将要显示的所有尺寸。另一个将根据 indexPath 索引路径返回相应的字体，它与我们在其他视图控制器中的用法类似。

```

- (NSArray *)pointSizes {
    static NSArray *pointSizes = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        pointSizes = @[
            @9,
            @10,
            @11,
            @12,
            @13,
            @14,
            @18,
            @24,
            @36,
            @48,
            @64,
            @72,
            @96,
            @144];
    });
    return pointSizes;
}

- (UIFont *)fontForDisplayAtIndexPath:(NSIndexPath *)indexPath {
    NSInteger *pointSize = self.pointSizes[indexPath.row];
    return [self.font fontWithSize:pointSize.floatValue];
}

```

请注意，pointSizes 方法使用的 dispatch\_once() 方法和我们之前所用的一样，用来确保这段代码只会运行一次。在这个示例中，它初始化了一个数字列表，将用来指定表中每行字体的属性。

对于这个视图控制器，我们要忽略用来指定有多少分区要显示的方法，这样就只会采用默认的数字（1）了。不过我们还是必须实现用来指定有多少行以及每个表单元内容的方法。此外还要跟之前一样实现用来指定每行高度的方法。以下就是提及的三个方法内容：

```

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
    #warning Incomplete method implementation.
    // 返回分区的行数
    return [self.pointSizes count];
    return 0;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    static NSString *CellIdentifier = @"FontNameAndSize";
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier
        forIndexPath:indexPath];

    // 配置表单元
    cell.textLabel.font = [self fontForDisplayAtIndexPath:indexPath];
}

```

```

        cell.textLabel.text = self.font.fontName;
        cell.detailTextLabel.text = [NSString stringWithFormat:@"%@" point",
                                     self.pointSizes[indexPath.row]];
    }
    return cell;
}

- (CGFloat)tableView:(UITableView *)tableView
heightForRowAtIndexPath:(NSIndexPath *)indexPath {
    UIFont *font = [self fontForDisplayAtIndexPath:indexPath];
    return 25 + font.ascender - font.descender;
}

```

这里所有的代码都是之前遇到过的，所以我们直接来看如何设置它的 GUI 界面。

### 9.6.1 设计字体尺寸视图控制器的分镜

回到 Main.storyboard 并将另一个 Table View Controller 拖到编辑区域中。使用身份检查器将它的类改为 BIDFontSizesViewController。需要将它与父控制器 Font List View Controller 用转场连接起来。因此请找到控制器并按住鼠标右键从它的单元样本拖动到最新的视图控制器上，然后在出现的菜单中 Selection Segue 部分选择 push。接下来选择你最新添加场景中的单元样本，然后使用特性检查器将其 Style 设为 Subtitle，将 Identifier 设为 FontNameAndSize。

### 9.6.2 对字体列表视图控制器的转场进行设置

现在我们需要切换到父视图控制器以设置它的子控制器，就像之前扩展分镜的导航层级那样。需要在 BIDFontListViewController.m 中导入新的子控制器头文件：

```

#import "BIDFontListViewController.h"
#import "BIDFavoritesList.h"
#import "BIDFontSizesViewController.h"

```

接下来移到 @implementation 部分的底部，移去 prepareForSegue:sender: 方法外面的注释标记，并按以下方式实现它：

```

/*
#pragma mark - Navigation

// 在基于分镜的应用中，经常需要在导航间作一些准备工作
// 将选定对象传入新的视图控制器
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    // 使用 segue destinationViewController 获取新的视图控制器
    // 将选定对象传入新的视图控制器
    NSIndexPath *indexPath = [self.tableView indexPathForCell:sender];
    UIFont *font = [self fontForDisplayAtIndexPath:indexPath];
    [segue.destinationViewController navigationItem].title = font.fontName;

    BIDFontSizesViewController *sizesVC = segue.destinationViewController;
    sizesVC.font = font;
}
*/

```

这些看起来应该很熟悉，所以我们就不过多深入了。

运行应用，选择某个字体系列，再选择某字体（在行内，轻点除了右侧辅助图标以外的任意位置），你将会看到如图 9-4 所示的多尺寸列表。

## 9.7 创建字体信息视图控制器

我们要创建的最后一个视图就是图 9-6 中所示的视图。它并不基于表视图，包含一个大的文本标签，一个用来设置文字尺寸的滑动条，以及一个用来处理这个字体是否要加入收藏的切换开关。在项目中创建一个新的 Objective-C 类，使用 `UIViewController` 作为父类，然后将其命名为 `BIDFontInfoViewController`。与应用中大多数其他控制器一样，需要有两个由父控制器传来的参数。在 `BIDFontInfoViewController.h` 中定义这些属性变量：

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

@interface BIDFontInfoViewController : UIViewController

@property (strong, nonatomic) UIFont *font;
@property (assign, nonatomic) BOOL favorite;
```

```
@end
```

现在切换到 `BIDFontInfoViewController.m` 并在顶部添加一个头文件导入和几个 `IBOutlet` 属性变量：

```
#import "BIDFontInfoViewController.h"
#import "BIDFavoritesList.h"

@interface BIDFontInfoViewController ()

@property (weak, nonatomic) IBOutlet UILabel *fontSampleLabel;
@property (weak, nonatomic) IBOutlet UISlider *fontSizeSlider;
@property (weak, nonatomic) IBOutlet UILabel *fontSizeLabel;
@property (weak, nonatomic) IBOutlet UISwitch *favoriteSwitch;

@end
```

接下来删除模版中的 `initWithNibName:bundle:` 方法和 `didReceiveMemoryWarning` 方法，因为这里不会用到它们。在此处实现 `viewDidLoad` 方法以及两个分别由滑动条和开关触发的操作方法：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // 加载视图后，进行额外设置

    self.fontSampleLabel.font = self.font;
    self.fontSampleLabel.text = @"AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVv"
                                "WwXxYyZz 0123456789";
    self.fontSizeSlider.value = self.font.pointSize;
    self.fontSizeLabel.text = [NSString stringWithFormat:@"%f",
                              self.font.pointSize];
    self.favoriteSwitch.on = self.favorite;
}
```

```

- (IBAction)slideFontSize:(UISlider *)slider {
    float newSize = roundf(slider.value);
    self.fontSampleLabel.font = [self.font fontWithSize:newSize];
    self.fontSizeLabel.text = [NSString stringWithFormat:@"%f", newSize];
}

- (IBAction)toggleFavorite:(UISwitch *)sender {
    BIDFavoritesList *favoritesList = [BIDFavoritesList sharedFavoritesList];
    if (sender.on) {
        [favoritesList addFavorite:self.font.fontName];
    } else {
        [favoritesList removeFavorite:self.font.fontName];
    }
}

```

这些方法都很简单。viewDidLoad 方法根据所选字体设置显示内容；slideFontSize:方法根据滑动条的值改变 fontSampleLabel 标签中字体的尺寸；toggleFavorite:会根据开关的值将当前字体添加或移除收藏列表。

### 9.7.1 设计字体信息视图控制器的分镜

现在回到 Main.storyboard 以构建这个应用最后的视图控制器 GUI 界面。通过对象库找到一个原始的 View Controller。将其拖动到编辑区域，并使用身份检查器将它的类设为 BIDFontInfo ViewController。接下来通过对象库找到其他对象并将它们拖入新场景中。需要三个标签、一个开关和一个滑动条。将它们按照图 9-8 的布局放在大致位置。

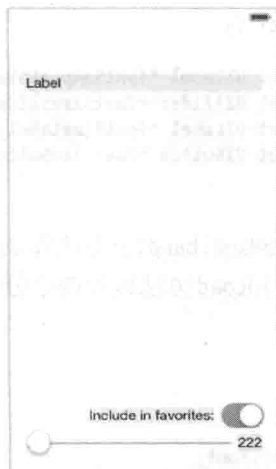


图 9-8 这里的每个标签都是浅灰色背景颜色，以突出显示。当然也可以使用白色背景

请注意我们在顶端标签的上面留了一些空间，这是因为要在这里放置导航条。此外，想要顶端标签能够显示多行大量文本，但标签默认的设置只能显示一行。如果要改变它，请选中标签，打开特性检查器，并将 Lines 文本框中的数字设为 0。

图 9-8 还展示了下方的两个标签中改过的文本。请参照它做同样的改动。图中并没有显示它们在特性检查器中都设为了右对齐。你需要这么做，因为它们的布局方式都是贴住右边的。此外选择底部的滑动条，然后使用特性检查器将它的 Minimum 设为 1，将 Maximum 设为 200。

现在要建立 GUI 中的所有连接。首先选择视图控制器并打开连接检查器。检查器中展示了我们要连接的许多内容。通过拖动 favoriteSwitch、fontSampleLabel、fontSizeLabel 和 fontSizeSlider 旁边的小圆圈到场景中相应的对象以创建每个输出接口的连接。如果觉得复杂，我们可以提示一下：fontSampleLabel 应该连接顶部的标签，fontSizeLabel 连接右下角的标签，而 favoriteSwitch 和 fontSizeSlider 输出接口都只有一个地方能连。为了连接操作方法和控件，你还需要用到连接检查器。拖动 slideFontSize: 旁边的小圆圈到滑动条上并释放鼠标按键，然后在显示的弹出菜单中选择 Value Changed。接下来拖动 toggleFavorite: 旁边的小圆圈到开关上并再次选择 Value Changed。

还需要做的一件事就是创建一个转场，否则该视图无法显示。请记住，当用户轻点详情辅助图标（圆圈中蓝色的 i）来显示字体列表视图控制器时，该视图便会出现。因此请找到控制器，按住鼠标右键从它的单元样本拖到之前所用的新字体信息视图控制器，在出现的菜单中 Accessory Action 部分选择 push。请注意我们说的是 Accessory Action（辅助操作），而不是 Selection Segue（选中转场）。辅助操作是当用户轻点详情辅助图标时触发的转场，而选中转场是点击行内任何位置时触发的转场。我们已经设置了这个表单元的选中转场为打开一个 BIDFontSizesViewController 控制器。

现在行中有两处不同的区域可以通过轻点来触发不同的转场。因为它们将会显示不同的视图控制器，以及拥有不同的属性变量，所以要用某种方式对它们进行区分。幸运的是，代表转场的 UIStoryboardSegue 类有一个解决的办法：我们可以使用标识符，就像对表视图单元所做的那样！

你所要做的是在编辑区域中选择一个转场并使用特性检查器设置设置它的 Identifier。你可能需要稍微移动你的场景位置，以便看到两条指向 Font List View Controller 右侧的转场。选择其中指向 Font Sizes View Controller 的并设置它的 Identifier 为 ShowFontSizes。接下来选择其中指向 Font Info View Controller 的并设置它的 Identifier 为 ShowFontInfo。

### 9.7.2 设置约束

设置转场可以让界面构建器知道新场景会像其他场景一样在导航控制器的环境中使用，因此场景会在顶端自动获取一个空白的导航条。现在视图的实际范围已经确定了，是时候设置它的约束了。这是一个相对复杂的视图，包含了多个子视图，尤其是底部附近，因此我们不能依赖系统的自动约束能为我们做到。我们将使用编辑区域底部的 Pin 按钮和它调出的弹出窗口来创建大部分所需的约束。

首先是最上面的标签。点击 Pin 按钮，然后在弹出窗口中选中小正方形上方、左侧和右侧的红色实线（不要选中下方的）。现在点击底部的 Add 3 Constraints 按钮。

接下来选择底部的滑动条并点击 Pin 按钮。这次请选中小正方形下方、左侧和右侧的红色实线（不要选中上方的）。再次点击 Add 3 Constraints 以进行布置。

针对剩下的两个标签和开关，都使用这个步骤：选择对象，点击 Pin 按钮，选中小正方形下方和右侧的红色实线，勾选 Width 和 Height 的复选框，最后点击 Add 4 Constraints。设置这三个对象的约束使它们限定在右下角。

此处还要创建一个约束。我们想让顶端标签可以延长以容纳更多的文本，但是不能延长过大以致覆盖底部的视图。我们可以通过一个约束来解决这个问题。按住鼠标右键从上方的标签拖到 Include in Favorites 标签，释放鼠标按钮并在出现的弹出菜单中选择 Vertical Spacing。接下来点击新的约束以选中它并打开特性检查器，你将在这里看到这个约束的一些配置。将 Relation 弹出菜单改为 Greater Than or Equal，然后设置 Constant 的值为 10。这样能确保上方延长的标签不会越过底部的其他视图。

### 9.7.3 调整字体列表视图控制器的转场

回到之前的 BIDFontListViewController.m。因为这个类现在能够触发两个不同子视图控制器的转场，所以它现在需要导入最后一个视图控制器的头文件：

```
#import "BIDFontListViewController.h"
#import "BIDFavoritesList.h"
#import "BIDFontSizesViewController.h"
#import "BIDFontInfoViewController.h"

- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    // 使用 segue destinationViewController 获取新的视图控制器
    // 将选定对象传入新的视图控制器
    NSIndexPath *indexPath = [self.tableView indexPathForCell:sender];
    UIFont *font = [self fontForDisplayAtIndexPath:indexPath];
    [segue.destinationViewController navigationItem].title = font.fontName;

    if ([segue.identifier isEqualToString:@"ShowFontSizes"]) {
        BIDFontSizesViewController *sizesVC = segue.destinationViewController;
        sizesVC.font = font;
    } else if ([segue.identifier isEqualToString:@"ShowFontInfo"]) {
        BIDFontInfoViewController *infoVC = segue.destinationViewController;
        infoVC.font = font;
        infoVC.favorite = [[BIDFavoritesList sharedFavoritesList].favorites
                           containsObject:font.fontName];
    }
}
```

现在运行应用并看看我们做到哪了！选择一个包含了多个字体的字体系列（比如 Gill Sans），然后轻点任意一行字体的中间位置，就将进入前面看到过的列表，它用多种尺寸来显示字体。按下左上角的导航按钮以返回，然后轻点另一行，不过这次请点击右侧显示的详情辅助图标。这样便会进入最后的视图控制器，显示了一个字体样本，以及底部一个可以让你选择想要尺寸的滑动条。

现在还可以使用 Include in favorites 开关来将这个字体标为收藏字体。完成后点击左上角的导航按钮两次以回到根控制器视图。



### 9.7.4 我的收藏字体

滚动到根视图控制器的底部，你将看到一些新的内容：第二个分区出现了，如图 9-9 所示。

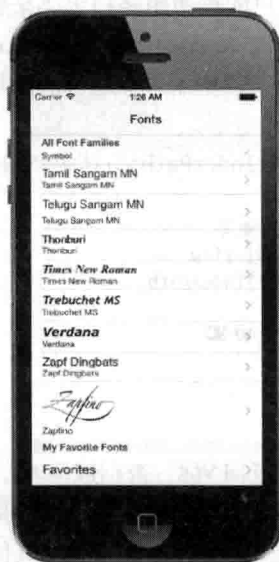


图 9-9 现在我们至少可以选择一个收藏字体，轻点根视图控制器底部的下行可以看到一个列表

轻点 Favorites 行，你将看到一个列表，包含了你所选择的所有收藏字体。在这里，可以做和其他字体列表一样的行为：轻点一行来看多个字体尺寸的列表，或者轻点详情辅助图标来察看可用滑动条调整字体的视图和收藏切换开关，甚至还可以尝试取消勾选开关并点击返回按钮，你将发现之前浏览的字体已经不在列表中了。

## 9.8 改善表视图

现在应用的基本功能已经完成了。但还没结束，仍有一些功能需要实现。如果曾经使用过 iOS 一段时间，可能就会发现经常可以在表视图中从右向左轻扫来删除某行。比如说在邮件应用中，可以使用这种技术来删除信息列表中的信息。使用这个手势会在表视图行中的右侧出现一个小的 GUI 界面。这个 GUI 会询问你是否确认要删除，确认后此行消失并且剩下的行会向上滑动填充空隙。所有的交互操作（包括处理轻扫手势、显示确认 GUI 以及与行有关的动画）都是由表视图自身完成的。你需要做的就是控制器中实现两个方法就能使其生效了。

表视图还提供了简易的功能，能让用户在表视图中通过向上或向下拖动行来重新为它们排序。与轻扫删除类似，表视图会处理所有的用户交互。需要做的只是先用一行代码进行设置（创建一个开启重新排序 GUI 的按钮），然后实现一个方法，它会在用户完成拖动后调用。表视图为我们节省了很多精力，不用它可真说不过去！

### 9.8.1 实现轻扫删除

在这个应用中, `BIDFontListViewController` 类就是一个该功能的典型例子。无论应用什么时候会显示收藏列表, 我们都应该让用户能够使用轻扫手势来删除收藏, 以省去要轻点详情辅助图标, 然后关闭切换开关的麻烦。首先在 Xcode 中选择 `BIDFontListController.m`。我们需要实现的方法已经默认在每个视图控制器的源文件中了, 不过它们都是注释状态。现在解除它们的注释并对它们进行实现。

首先移除 `tableView:canEditRowAtIndexPath:` 方法外的注释标记并对它进行实现:

```
/*
// 为了支持表视图额外的编辑操作而进行重写
- (BOOL)tableView:(UITableView *)tableView
canEditRowAtIndexPath:(NSIndexPath *)indexPath
{
    // 如果不希望特定项目被编辑, 则返回 NO
    return self.showsFavorites;
    // return YES;
}
*/
```

如果这里显示的是收藏列表就会返回 YES, 否则返回 NO。这意味着能够删除行的编辑功能只会在显示收藏列表时启用。如果你只做了这个更改再运行应用并试着删除行, 会发现没有任何变化。这个表视图不会处理轻扫手势, 因为我们还没有实现其他能完成删除的方法。所以要继续实现。请删除 `tableView:commitEditingStyle:forRowAtIndexPath:` 方法外的注释标记并根据以下方法中的内容进行更改:

```
/*
// 为了支持表视图额外的编辑操作而进行重写
- (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (!self.showsFavorites) return;

    if (editingStyle == UITableViewCellEditingStyleDelete) {
        // 从数据源中删除该行
        NSString *favorite = self.fontNames[indexPath.row];
        [[BIDFavoritesList sharedFavoritesList] removeFavorite:favorite];
        self.fontNames = [BIDFavoritesList sharedFavoritesList].favorites;

        [tableView deleteRowsAtIndexPaths:@[indexPath]
                        withRowAnimation:UITableViewRowAnimationFade];
    }
    else if (editingStyle == UITableViewCellEditingStyleInsert) {
        // 创建相关类的新实例, 将其插入到数组中, 为表视图添加新的一行
        // array, and add a new row to the table view
    }
}
*/
```

这个方法非常简单, 不过有些内容很微妙。首先要通过判断以确保显示的是收藏列表, 如果

不是的话直接跳出。通常这种情况不可能发生，因为在上一个方法中指定了只有收藏列表可以被编辑。尽管如此，我们还是在这里做了一些防御性编程。之后我们检查了编辑类型以确保我们现在处理的是删除操作。在表视图中还可以进行插入操作，不过在这里我们不会使用它，所以我们不需要考虑其他情况。接下来我们算出哪一个字体需要被删除，将其从 `BIDFavoritesList` 单例中移除，并更新收藏列表的本地副本。

最后我们告诉表视图删除行并使其通过一段渐隐动画后消失。你应当理解告诉表视图删除一行时，会发生什么。从直觉上你可能认为调用此方法会删除一些数据，但并不是这样的。事实上，我们已经删完了数据！最后一个方法的调用其实是以一种方式告诉表视图：“嘿，我做了点变动，并想让你用动画移除这行。执行其他操作时再问我”。当它发生时，表视图会开始对被删那行下面的行使用动画，使它们上升，这意味着之前不在屏幕内的一行或多行现在将进入屏幕，这时它将通过控制器的通用方法询问表单元数据。因为这个原因，`tableView:commitEditingStyle:forRowAtIndexPath:`方法的实现中对数据模型（在这个示例中即 `BIDFavoritesList` 单例）进行必要的改动需要在告诉表视图删除某行之前执行。

现在再次运行应用，确保你已经设定了一些收藏字体，然后进入 Favorites 列表并通过从右向左轻扫删除某行。这一行部分会移出屏幕，并在右侧显示一个删除按钮。轻点删除按钮，当前行就会消失。

## 9.8.2 实现拖动排序

要为字体列表添加最后一个功能，使用户能通过向上或向下拖动收藏项就能对它们进行排序。为了完成这个功能，我们要向 `BIDFavoritesList` 类添加一个方法，它能让我们对内容进行各种排序。打开 `BIDFavoritesList.h` 并在 `@interface` 部分添加如下声明语句：

```
- (void)moveItemAtIndex:(NSInteger)from toIndex:(NSInteger)to;
```

接下来切换到 `BIDFavoritesList.m` 并在 `@implementation` 部分添加这个方法：

```
- (void)moveItemAtIndex:(NSInteger)from toIndex:(NSInteger)to {
    id item = _favorites[from];
    [_favorites removeObjectAtIndex:from];
    [_favorites insertObject:item atIndex:to];
    [self saveFavorites];
}
```

它为我们要做的事提供基本功能。现在选择 `BIDFontListViewController.m` 并在 `viewDidLoad` 方法末尾添加以下语句：

```
if (self.showsFavorites) {
    self.navigationItem.rightBarButtonItem = self.editButtonItem;
}
```

我们之前提到过导航项。它是一个保存视图控制器导航条上应该显示什么信息的对象。它有一个叫做 `rightBarButtonItem` 的属性，它是 `UIBarButtonItem`（只能用在导航条或工具栏上的某种特殊按钮）实例。这里我们使用的是 `editButtonItem`，它是 `UIViewController` 为我们提供的特殊按钮，已经预设好了会启用表视图的编辑或排序 GUI 界面。

完成之后，试着再次运行应用并进入 Favorites 列表。你将看到右上角现在有一个编辑按钮。按下这个按钮就会启用表视图的编辑 GUI 界面，即每行左侧有一个删除按钮，而内容会向右移动一点以腾出空间。这是另一种让用户删除行的方式，使用的是我们已经实现的同样方法。

不过我们现在的主要目的是添加重新排序功能。因此我们需要做的是实现这个方法。确保已移除了默认的注释标记，然后添加粗体显示的代码：

```
/*
// 为了重新整理表视图而重写
- (void)tableView:(UITableView *)tableView
moveRowAtIndexPath:(NSIndexPath *)fromIndexPath
toIndexPath:(NSIndexPath *)toIndexPath
{
    [[BIDFavoritesList sharedFavoritesList] moveItemAtIndexPath:fromIndexPath.row
toIndexPath:toIndexPath.row];
    self.fontNames = [BIDFavoritesList sharedFavoritesList].favorites;
}
*/
```

这个方法会在用户完成一行的拖动时调用。我们在这里告诉 BIDFavoritesList 单例进行重新排序，然后刷新字体名称的列表，和完成删除一行后所做的相似。为了观看实际效果，请运行应用，进入 Favorites 列表，并轻点编辑按钮。你将看到编辑模式现在包含了小“拖动手柄”图标位于每行右侧，你可以使用拖动手柄来重新排序。

这样我们的应用就完成了！至少将此书中提到的功能完成了。如果你能为这些字体想出更多有用的功能，请试试看！

## 9.9 小结

这一章真是无比漫长。如果你坚持到这儿了，应该会觉得非常折磨人。在这些神奇的表视图和导航控制器对象上纠结这么久是有必要的，因为它们是大量 iOS 应用的支柱，如果没有真正理解它们，会被它们的复杂性困住的。

开始构建自己的表时，可以参考这一章以及上一章，并且不要害怕苹果的官方文档。表视图异常复杂，而且它不可能考虑到每种情况。不过你现在肯定了解了很多用来构建表视图的代码，这些代码都可以免费用在你自己的应用——这是来自我们的馈赠，希望你能喜欢。

本章要介绍的是 UIKit 最近新增的 `UICollectionView` 类，这个类与大家已经非常熟悉的 `UITableView` 有一些相似之处，也有一些不同之处，可以用来做一些使用 `UITableView` 时做梦都想不到的事情。

这些年来，iOS 开发者使用 `UITableView` 组件创建了大量各式各样的用户界面。使用 `UITableView`，我们可以定义多种不同类型的表视图单元（在程序运行中，可以在需要的时候即时创建表视图单元），可以很方便地进行垂直滚动，`UITableView` 是许许多多应用的关键组件。这些年来，苹果也一直非常用心地改进表视图类，在每个主要的 iOS 发布版中都会为 `UITableView` 类增加一些新内容。

然而，`UITableView` 并非是所有大数据集合的终极解决方案。例如，如果希望以多列的形式来展示数据，就需要将每行的数据以纵列的形式嵌入到一个表视图单元中。我们也无法让 `UITableView` 的内容进行水平滚动。总之，`UITableView` 的强大能力是一种折中的结果：开发者无法控制表视图的整体布局。每个独立表视图单元的外观是可以定制的，但直到今天，每个表视图单元也只能堆叠在一个滚动列表中！

显然苹果也意识到了这个问题。iOS 6 中增加了一个名为 `UICollectionView`（集合视图类）的新类，它就是用来弥补表视图的这些不足的。与表视图一样，集合视图也可以用来显示大量的数据“单元”，也具备一些与表视图同样的特性，比如将单元放在队列中以便之后复用。与表视图不同的是，`UICollectionView` 并不是将这些单元垂直堆叠起来。事实上，`UICollectionView` 根本就不处理这些单元的布局，而是使用一个辅助类（helper class）处理布局，稍后你就会看到。

## 10.1 创建 DialogViewer 项目

为了展示 `UICollectionView` 的能力，接下来我们要用它处理一些文本段落的布局。每一个单词都位于独立的单元中，每个文本段落的所有单元都被集中在一个分区（section）中。每个分区都有自己的标题（header）。可能这并不足以令你激动，因为 UIKit 提供了处理文本布局的更完美方式，但这个例子非常具有教育意义，能够让你很直观地感受到 `UICollectionView` 的灵活性。如果使用表视图，你是不可能得到图 10-1 所示布局的。

为了达到这个目的，需要自定义两个单元类，这里将使用 `UICollectionViewFlowLayout` 类（当

前版本的 UIKit 中唯一的一个布局辅助类), 并与往常一样, 使用视图控制器将这些东西组织在一起。现在就开始吧!

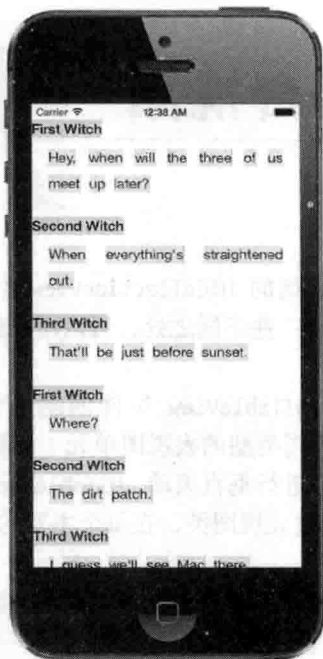


图 10-1 每个单词都位于独立的单元中。只是使用 UICollectionView 就可以实现这种布局, 不需要进行显式的几何计算

使用 Xcode 创建一个基于 Single View Application 模板的项目, 如之前你多次做过的那样。我偿将项目命名为 DialogViewer, 其他设置都使用本书中的通用设置( Class Prefix 的值设为 BID, 在 Devices 弹出菜单中选择 iPhone)。

## 10.2 修补视图控制器类

在这个应用中, 我们不需要对应用委托做任何修改, 所以可以直接跳到 BIDViewController.h 文件做一处简单的修改, 将超类更改为 UICollectionView:

```
@interface BIDViewController : UIViewController
@interface BIDViewController : UICollectionView
```

然后, 打开 Main.storyboard 文件。我们需要对视图进行一些相应的设置, 以便它与头文件中的修改相匹配。在编辑区域中选中唯一的视图控制器场景并将其删除。然后在对象库中找到 Collection View Controller, 将其拖曳到编辑区。选中你刚刚拖进来的视图控制器图标并使用身份检查器将它的类更改为 BIDViewController。

## 10.3 自定义单元

接下来定义一些单元类。如图 10-1 所示，这里需要显示两种基本的单元：一种用于包含单词的“普通”单元，还有一种用作段落标题的单元。任何需要在 `UICollectionView` 中创建并使用的单元都必须是系统自带的 `UICollectionViewCell` 类的子类。`UICollectionViewCell` 提供的功能与 `UITableViewCell` 非常相似：都有 `backgroundView` 和 `contentView` 等内容。由于这两种单元有很多共同的功能，所以我们可以让其中一个继承另一个，然后覆盖某些功能。

首先在 Xcode 中创建一个 Objective-C 类，将其命名为 `BIDContentCell`，作为 `UICollectionViewCell` 的子类使用。我们选中新类的头文件，添加两个属性声明和一个类方法：

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

@interface BIDContentCell : UICollectionViewCell

@property (strong, nonatomic) UILabel *label;
@property (copy, nonatomic) NSString *text;

+ (CGSize)sizeForContentString:(NSString *)s;

@end
```

`label` 属性指向一个用于显示内容的 `UILabel`，`text` 属性就是单元将要显示的内容。对于特定的字符串，我们可以使用 `sizeForContentString:` 方法来计算需要多大的单元才能将字符串完整显示出来。在创建并配置单元类实例的时候，它们会非常有用。

现在，切换到 `BIDContentCell.m` 文件，在这里有一些琐碎的工作要做。首先，我们建立 `initWithFrame:` 方法，如下所示：

```
- (id)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        // 初始化代码
        self.label = [[UILabel alloc] initWithFrame:self.contentView.bounds];
        self.label.opaque = NO;
        self.label.backgroundColor = [UIColor colorWithRed:0.8
                                                             green:0.9
                                                             blue:1.0
                                                             alpha:1.0];
        self.label.textColor = [UIColor blackColor];

        self.label.textAlignment = NSTextAlignmentCenter;
        self.label.font = [[self class] defaultFont];
        [self.contentView addSubview:self.label];
    }
    return self;
}
```

这段代码非常简单。它只是创建一个标签，设置标签的显示属性，然后将标签添加到单元的 `contentView`。这里唯一值得注意的是，它使用 `defaultFont` 类方法获取字体，然后设置标签字体。



这么做的原因是类需要使用这个方法确定显示内容所使用的字体,同时也可以让任何子类通过覆盖 `defaultFont` 方法定义各自用于显示的字体。现在就来创建这个方法:

```
+ (UIFont *)defaultFont {
    return [UIFont preferredFontForTextStyle:UIFontTextStyleBody];
}
```

这里的内容非常简单直观。这里利用了 iOS 7 的一个新功能,可以让用户在设置应用中指定字体的尺寸。比起使用硬编码的字体尺寸,这种体验对用户更加友好。

```
+ (CGSize)sizeForContentString:(NSString *)string {
    CGSize maxSize = CGSizeMake(300, 1000);

    NSStringDrawingOptions opts = NSStringDrawingUsesLineFragmentOrigin |
                                   NSStringDrawingUsesFontLeading;

    NSMutableParagraphStyle *style = [[NSMutableParagraphStyle alloc] init];
    [style setLineBreakMode:NSLineBreakByCharWrapping];

    NSDictionary *attributes = @{ NSFontAttributeName : [self defaultFont],
                                   NSParagraphStyleAttributeName : style };

    CGRect rect = [string boundingRectWithSize:maxSize
                                       options:opts
                                       attributes:attributes
                                       context:nil];

    return rect.size;
}
```

为了完成这个类,我们还要添加之前在头文件中提到过的方法,用来计算表单单元的合适尺寸。

这个方法执行了很多内容,来简单介绍一下。首先我们声明了一个最大尺寸,以防止单词的宽度超过屏幕的空间。接下来设定了一些选项,以帮助系统计算出所处理字符串的正确尺寸。我们还创建了一个显示方式为 `character wrapping` 的段落类型,这样假如字符串的宽度超过了之前所定的允许大小,多余的文本内容将移至下一行。之后创建了一个 `attributes` 字典来保存为这个类指定的默认字体和刚刚创建的段落类型。最后我们使用一些 UIKit 提供的 `NSString` 函数来计算字符串所占的屏幕尺寸。将最大允许尺寸和其他设定好的选项与属性传给这个方法,就会获取到一个尺寸值。

剩下的就是对 `text` 属性的一些特殊处理了。相对于之前只是使用隐式的实例变量,这次我们要自己为 `text` 属性定义获取方法和设置方法,用之前创建的 `UILabel` 存储显示的值,基于 `UILabel` 来设置 `text` 属性的值。这样的话,当文本发生变化时,我们还可以在设置方法中重新计算单元的几何尺寸。代码如下所示:

```
- (NSString *)text {
    return self.label.text;
}

- (void)setText:(NSString *)text {
    self.label.text = text;
    CGRect newLabelFrame = self.label.frame;
    CGRect newContentFrame = self.contentView.frame;
```



```

CGSize textSize = [[self class] sizeForContentString:text];
newLabelFrame.size = textSize;
newContentFrame.size = textSize;
self.label.frame = newLabelFrame;
self.contentView.frame = newContentFrame;
}

```

获取器方法并没有什么特别的，但是设置器方法做了一些额外的工作。简单来说，它根据显示当前字符串所需的单元尺寸修改了标签和内容视图的 frame。

对于基本的单元类来说，这就是全部内容了。现在再为段落标题创建一个单元类。这里再使用 Xcode 创建一个新的 Objective-C 类，命名为 BIDHeaderCell，让其继承 BIDContentCell。这里不需要修改头文件，直接对 BIDHeaderCell.m 做一些修改。在这个类中，我们需要做的仅仅是覆盖几个方法以改变单元的外观，让它与普通的内容单元看起来不同：

```

- (id)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        // 初始化代码
        self.label.backgroundColor = [UIColor colorWithRed:0.9
                                                             green:0.9
                                                             blue:0.8
                                                             alpha:1.0];
        self.label.textColor = [UIColor blackColor];
    }
    return self;
}

+ (UIFont *)defaultFont {
    return [UIFont preferredFontForTextStyle:UIFontTextStyleHeadline];
}

```

这样，我们就为标题单元设置了颜色和字体，让它有了不同的外观。

## 10.4 配置视图控制器

现在将注意力集中到视图控制器上。我们选中 BIDViewController.m 文件，首先导入自定义单元的头文件，并且声明一个数组，用于保存待显示的内容：

```

#import "BIDViewController.h"
#import "BIDContentCell.h"
#import "BIDHeaderCell.h"

@interface BIDViewController ()
@property (copy, nonatomic) NSArray *sections;
@end

```

然后，在 viewDidLoad 方法中定义数据。sections 数组需要包含一个字典列表，每个字典有两个键：header 和 content。我们需要根据键所关联的值来定义需要显示的内容，而实际上要使用的内容来自于一个非常有名的游戏。

```

- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后 (通常是从 nib 加载), 做一些额外的设置
    self.sections =
    @[
        @{@"header" : @"First Witch",
          @"content" : @"Hey, when will the three of us meet up later?" },
        @{@"header" : @"Second Witch",
          @"content" : @"When everything's straightened out." },
        @{@"header" : @"Third Witch",
          @"content" : @"That'll be just before sunset." },
        @{@"header" : @"First Witch",
          @"content" : @"Where?" },
        @{@"header" : @"Second Witch",
          @"content" : @"The dirt patch." },
        @{@"header" : @"Third Witch",
          @"content" : @"I guess we'll see Mac there." },
    ];
}

```

与 UITableView 非常像, UICollectionView 允许基于标识符来注册需要复用的单元类。这样一来,当后面需要使用单元的时候,我们就可以直接调用一个出队方法,而如果没有可用的单元,集合视图会自动创建一个。与 UITableView 一样,我们将下面的代码添加到 viewDidLoad 方法的最后:

```

[self.collectionView registerClass:[BIDContentCell class]
    forCellWithReuseIdentifier:@"CONTENT"];

```

UICollectionView 默认拥有一个黑色的背景。但我们希望这个应用程序看上去能清爽一些,所以将其颜色替换为白色:

```

self.collectionView.backgroundColor = [UIColor whiteColor];

```

我们还要对 viewDidLoad 方法作另一个更改。因为这个应用程序没有导航栏,主视图依然会影响到状态栏。在 viewDidLoad 方法的末尾添加以下代码就能防止这样的状况(在前几章中我们对其他视图做过很多相似的事情)。

```

UIEdgeInsets contentInset = self.collectionView.contentInset;
contentInset.top = 20;
[self.collectionView setContentInset:contentInset];

```

目前来说, viewDidLoad 方法需要做的就是这么多。在编写生成集合视图的代码之前需要写一个简单的辅助方法。所有内容都包含在一个冗长的字符串中,但是需要每次处理一个单词,以便将其放入相应的单元中。因此,我们来创建一个自己的内部方法,用于将字符串分离为单词。这个方法接受一个分区号码作为参数,从分区数据中提取出相应的内容字符串,然后将其分离为单词:

```

- (NSArray *)wordsInSection:(NSInteger)section {
    NSString *content = self.sections[section][@"content"];
    NSString *space = [NSString whitespaceAndNewlineCharacterSet];
    NSArray *words = [content componentsSeparatedByCharactersInSet:space];
    return words;
}

```

## 10.5 内容单元

现在是时候实现那些真正用于生成集合视图的方法了。接下来的 3 个方法与 UITableView 中对应的方法非常相似。首先，我们需要一个方法让集合视图知道一共有多少个分区需要显示：

```
- (NSInteger)numberOfSectionsInCollectionView:(UICollectionView *)collectionView {
    return [self.sections count];
}
```

第二个方法用于指明每个分区中有多少个条目。这个方法调用了我们之前定义的 wordsInSection:方法：

```
- (NSInteger)collectionView:(UICollectionView *)collectionView
numberOfItemsInSection:(NSInteger)section {
    NSArray *words = [self wordsInSection:section];
    return [words count];
}
```

我们用第三个方法得到一个单元，对其进行配置让每个单元都包含一个单词，然后将单元返回。这个方法也调用了之前定义的 wordsInSection:方法。如下代码所示，它会在 UICollectionView 上调用一个出队方法，这也与 UITableView 非常相似。由于我们之前已经为这里使用的标识符注册了单元，所以这里的出队方法总是能够返回一个有效的单元实例：

```
- (UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
cellForItemAtIndexPath:(NSIndexPath *)indexPath {
    NSArray *words = [self wordsInSection:indexPath.section];

    BIDContentCell *cell = [self.collectionView
                           dequeueReusableCellWithReuseIdentifier:@"CONTENT"
                           forIndexPath:indexPath];
    cell.text = words[indexPath.row];
    return cell;
}
```

如果根据使用 UITableView 的经历来判断，你可能会认为现在这个应用至少已经可以正常工作了。构建并运行应用，你会发现事实并非如此，参见图 10-2。



图 10-2 这样的应用并没有什么用

我们可以看到一些单词，但是布局非常混乱，所有的东西都混在一起。原因是，还有一些委托方法没有实现。

## 10.6 实现流式布局

目前为止，我们一直在介绍 `UICollectionView`，但是正如之前提到的，这个类使用一个辅助类处理布局。`UICollectionViewFlowLayout` 是 `UICollectionView` 默认的布局辅助类，它有一些委托方法。接下来，我们实现其中的一个委托方法。布局对象会为每个单元调用这个方法，以便得到这个单元的尺寸。这里又一次用到了之前定义的 `wordsInSection:` 方法，它用于得到分区中的单词，还调用了 `BIDContentCell` 类中一个方法来计算单元的尺寸。

```
- (CGSize)collectionView:(UICollectionView *)collectionView
    layout:(UICollectionViewLayout*)collectionViewLayout
    sizeForItemAtIndexPath:(NSIndexPath *)indexPath {
    NSArray *words = [self wordsInSection:indexPath.section];
    CGSize size = [BIDContentCell sizeForContentString:words[indexPath.row]];
    return size;
}
```

我们再次构建并运行应用，可以看到效果好多了，参见图 10-3。

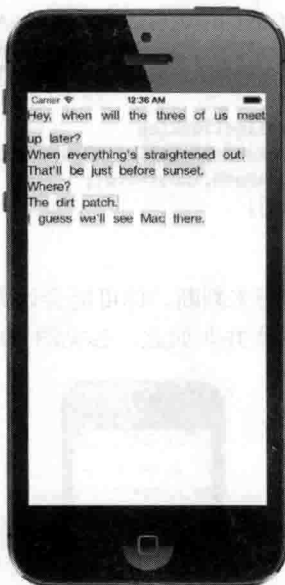


图 10-3 文本段落的流式布局已经初步成形了

可以看到，现在的文本已经比之前更适合阅读了，不同分区之间也有了一点点间隔。但是，每一个分区内部的单元仍然混为一团。有一些单元紧贴着视图边缘，这样看起来并不舒服。接下来再做一些配置修复这个问题。我们在 `viewDidLoad` 方法最后添加如下代码：

```
UICollectionViewLayout *layout = self.collectionView.collectionViewLayout;
UICollectionViewFlowLayout *flow = (UICollectionViewFlowLayout *)layout;
flow.sectionInset = UIEdgeInsetsMake(10, 20, 30, 20);
```

这段代码首先从集合视图得到布局对象。首先使用一个临时的 `UICollectionViewLayout` 对象来保存这个布局对象，这主要是为了突出一个点：`UICollectionView` 仅仅好像是“知道”布局类的大致情况，但并不能在运行时直接使用它。在实践中，除非特别指定，否则我们都是使用 `UICollectionViewFlowLayout` 实例处理布局。由于知道布局对象的真正类型，所以可以使用类型转换将其赋给另一个变量，这样就可以访问相应的布局对象子类的方法了。

再次构建并运行，我们可以看到各个文本单元之间有了一些空隙，视图看起来让人舒服多了，参见图 10-4。

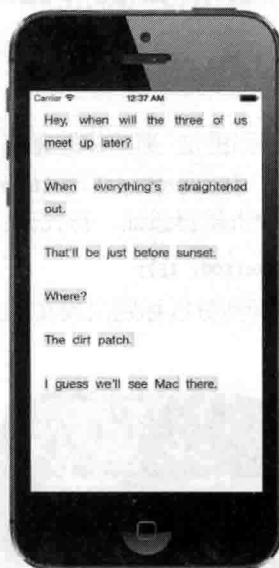


图 10-4 各个文本单元之间有了一些空隙，看起来让人舒服多了

## 10.7 分区标题视图

现在，剩下的就是标题对象的显示了，我们一起来搞定它吧。回忆一下之前的内容，`UITableView` 有一个表头和表尾视图，可以为每个分区单独指定。在 `UICollectionView` 中，这个概念变得更加通用了，使得布局更加灵活。除了可以使用委托方法访问普通的单元视图，我们还有一套类似的系统用于访问额外的视图，比如表头、表尾等。请在 `viewDidLoad` 中添加如下代码，注册集合视图的分区标题：

```
[self.collectionView registerClass:[BIDHeaderCell class]
forSupplementaryViewOfKind:UICollectionViewElementKindSectionHeader
withReuseIdentifier:@"HEADER"];
```

如上所示，示例不仅指定了单元类和标识符，而且指定了视图的“种类”。这么做的原因是，不同的布局可以定义不同“种类”的辅助视图，然后要求委托方法提供相应的视图。UICollectionViewLayout 需要为集合视图中的每一个分区提供一个分区标题，代码如下所示：

```
- (UICollectionViewReusableView *)collectionView:(UICollectionView *)collectionView
viewForSupplementaryElementOfKind:(NSString *)kind
atIndexPath:(NSIndexPath *)indexPath {
    if ([kind isEqual:UICollectionViewElementKindSectionHeader]) {
        BIDHeaderCell *cell = [self.collectionView
                               dequeueReusableSupplementaryViewOfKind:kind
                               withReuseIdentifier:@"HEADER"
                               forIndexPath:indexPath];

        cell.text = self.sections[indexPath.section][@"header"];
        return cell;
    }
    return nil;
}
```

构建并运行，你会看到……等等，标题呢？那些标题跑哪里去了？它们没有出现的原因在于，除非明确指定标题视图的尺寸，否则 UICollectionViewLayout 不会为标题视图提供任何显示空间。所以，我们在 viewDidLoad 方法的末尾再添加一行代码：

```
flow.headerReferenceSize = CGSizeMake(100, 25);
```

再次构建并运行这个应用，可以看到分区标题出现在了正确的位置，如之前的图 10-1 和图 10-5 所示。



图 10-5 完成后的 DialogViewer 应用

## 10.8 小结

本章中，我们只是稍微了解了一下 `UICollectionView` 以及默认的布局类 `UICollectionViewFlowLayout`。通过定义自己的布局类，我们可以实现更加精美的布局，不过本书不对此作深入介绍，读者可以自己参考苹果的官方文档进行深入学习。

现在你已经熟练掌握了所有主要的设计元素，是时候学习如何在全新的屏幕尺寸上通过完全不同的方式使用它们了。我们说的就是 iPad，在第 11 章中会对它进行讲解。

从技术角度讲，编写 iPad 程序与编写面向其他任何 iOS 设备的程序非常相似。除了屏幕大小，3G iPad 与 iPhone 或 Wi-Fi iPad 与 iPod touch 之间几乎没什么区别。尽管 iPhone 和 iPad 在本质上很相似，但从用户角度看这些设备之间具有很大的不同。幸好苹果公司一开始就充分认识到了这一点，为 iPad 配备了额外的 UIKit 组件，以帮助开发者创建可以更好地利用 iPad 的屏幕大小和使用模式的应用。本章将介绍如何使用这些组件。我们开始吧！

## 11.1 分割视图和浮动窗口

第 9 章花了较长篇幅介绍基于表视图中的选择实现应用导航，其中每个选择都会导致顶级视图（填满整个屏幕的视图）滑到左边并调出层次结构中的下一个视图，也可能是另一个表视图。许多 iPhone 和 iPod touch 应用都以这种方式工作，包括苹果公司自己的应用和第三方应用。

一个典型的例子是邮件应用，它支持在服务器和文件夹中不断向下展开，直到最终找到邮件。从技术上讲，此方法也适用于 iPad，但它会导致用户交互问题。

在 iPhone 或 iPod touch 这样大小的屏幕上，让一个屏幕大小的视图滑走以显示另一个屏幕大小的视图，这没有什么问题。然而在 iPad 这样大小的屏幕上，同样的交互让人感觉不太对劲，有点夸张，甚至有点让人窒息。此外，在这样大的显示屏上仅显示一个表视图在多数情况下都有点浪费。所以，你会看到内置的 iPad 应用没有按此方式工作。相反，任何向下导航功能（比如邮件应用中使用的），都归入了一个较窄的列中，当用户向下导航或者返回时，它的内容会向左或向右滑动。当 iPad 处于横向模式时，导航栏位于左侧固定位置，而所选项的内容则在右侧显示。这在 iPad 中称为分割视图，如图 11-1 所示。

左侧分割视图的宽度始终为 320 点（与纵向模式下 iPhone 的宽度相同），分割视图本身并列显示导航栏和内容，并且仅在横向模式下显示。如果将 iPad 切换为纵向，分割视图仍然有效，但它的显示方式将发生变化。导航视图不再固定在某个位置，可以点击一个工具栏按钮来激活它，这会导致导航视图弹出一个视图，该视图漂浮在屏幕上所有其他内容的前方（参见图 11-2）。这就是所谓的浮动窗口（popover）。





图 11-1 此 iPad 处于横向模式，显示了一个分割视图。导航栏位于左侧。点击导航栏中的一项（在本例中为特定的邮件账户），该项的内容会在右侧区域中显示

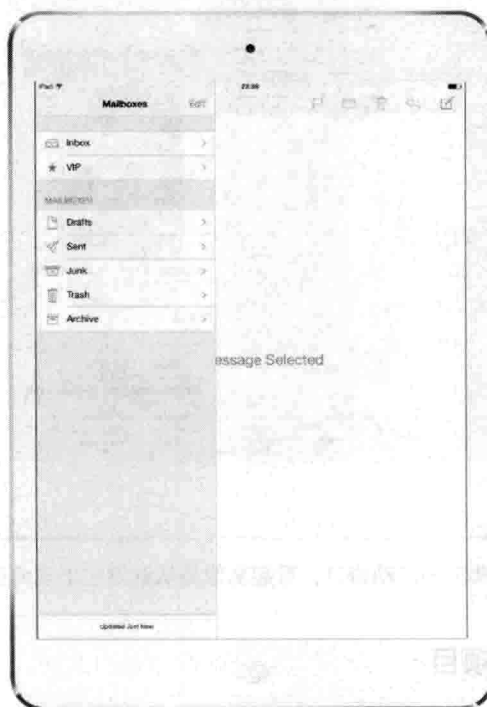


图 11-2 此 iPad 处于纵向模式，没有出现横向模式下的分割视图。横向模式中分割视图左侧的信息到了纵身视图中就变成了一个浮动窗口

然而有些应用不会严格遵循这个规范。iPad 上的设置应用就是分割视图在任何状态下都可见的。左侧的视图不会消失，也不会覆盖右侧的视图。在本章中，我们仍然会坚持遵循这个标准方案。

以分割视图形式显示的浮动窗口外观与你以前见过的许多浮动窗口都不一样。这个浮动窗口总会出现在屏幕的左侧并盖住屏幕，但不是所有浮动窗口都这样。它们可以通过设置以不一样的尺寸出现在屏幕的任意位置。在你阅读这一章的时候会看到一个悬挂在工具栏按钮上的小型浮动窗口（参见图 11-3）。本章的示例项目将会介绍如何创建同时使用分割视图和与之相应浮动窗口的 iPad 应用，以及如何创建和显示自定义浮动窗口。你还将学到如何创建并显示不会依附在任何分割视图上的浮动窗口。



图 11-3 一个典型的浮动窗口，看起来像是从触发它出现的按钮中弹出的

### 11.1.1 创建SplitView项目

我们将先从简单的工作开始，利用 Xcode 预定义的一个模板创建一个分割视图项目。我们将构建一个应用，功能是列出美国历届总统并显示你所选择的总统的维基百科条目。

转到 Xcode 并选择 **File>New>Project...** 菜单。在 iOS 下的 **Application** 分类中, 选择 **Master-Detail Application**, 然后点击 **Next**。在下一个页面中, 将新项目命名为 **Presidents**, 将 **Class Prefix** 设置为 **BID**, 而将 **Devices** 设为 **iPad**。确保 **Use Core Data** 复选框没有勾选。点击 **Next**, 为该项目选择存储位置, 最后点击 **Create**。Xcode 将会完成一些常规工作, 为你创建一些类以及分镜文件, 然后显示该项目。如果还没有打开 **Presidents** 文件夹, 那么展开它, 查看其中的内容。

项目最初包含一个应用委托 (和平常一样)、**BIDMasterViewController** 类和 **BIDDetailViewController** 类。这两个视图控制器分别表示将在分割视图左侧和右侧显示的视图。**BIDMasterViewController** 定义导航结构的顶级视图, **BIDDetailViewController** 定义在选择某个导航元素时, 在较大的区域中显示的内容。应用启动时, 这两部分都包含在分割视图内, 你可能还记得, 旋转设备时它们的形状会略微发生变化。

要查看这个应用模板提供了哪些功能, 可以在模拟器中构建并运行它。在横向模式 (参见图 11-4) 和纵向模式 (参见图 11-5) 之间切换, 就会看到分割视图的实际应用。在横向模式下, 分割视图在左侧显示导航视图, 在右侧显示详细信息视图。在纵向模式下, 详细信息视图占据了屏幕的大部分空间, 导航元素被限制在浮动窗口中, 点击视图左上角的按钮就会调出该窗口。

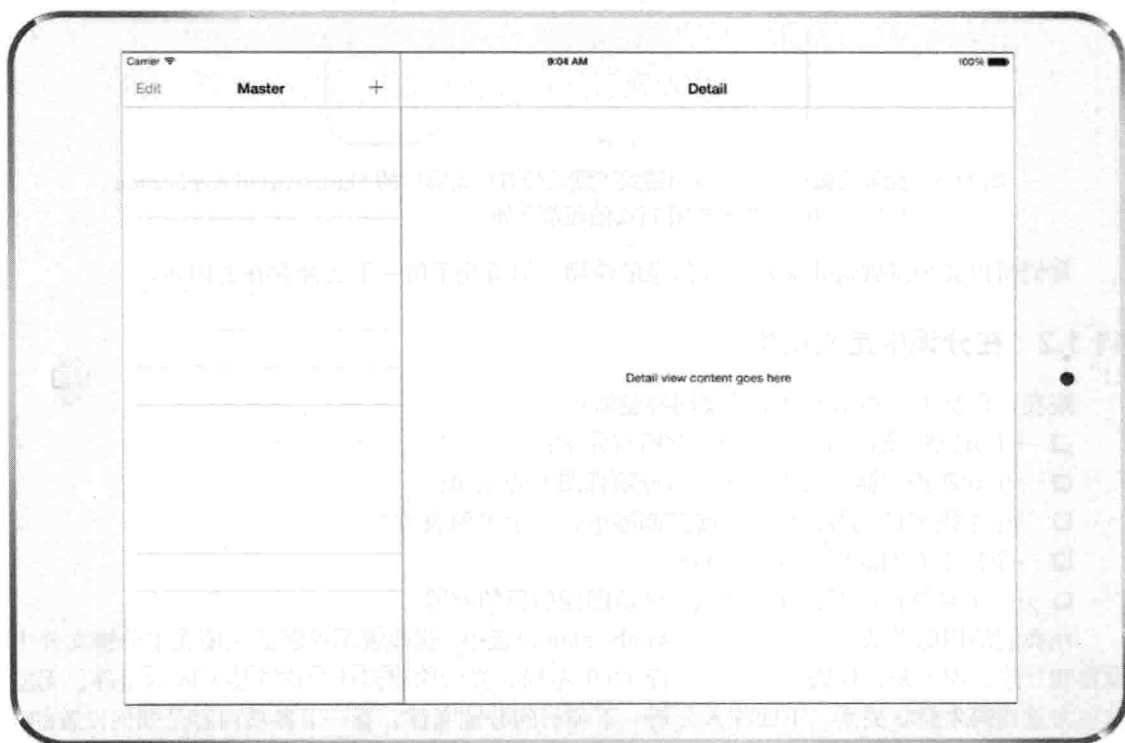


图 11-4 此屏幕截图显示了横向模式下默认的 Master-Detail Application 模板。请注意此图与图 11-1 的相似之处



图 11-5 此屏幕截图显示了纵向模式下默认带有浮动窗口的 Master-Detail Application 模板。请注意此图与图 11-2 的相似之处

我们将以此为基础构建展示总统信息的应用，但首先了解一下已经存在的内容。

### 11.1.2 在分镜中定义结构

现在，你有了一些相当复杂的视图控制器：

- 一个分割视图控制器，其中包含所有元素；
- 一个导航控制器，用于处理左侧分割视图上的操作；
- 一个主视图控制器，位于导航控制器中，显示主列表项；
- 一个位于右侧的详情视图控制器。
- 另一个导航控制器，作为右侧详情视图控制器的容器。

在我们使用的默认的 Master-Detail Application 模板中，这些视图控制器主要在主分镜文件中设置和互连，而不是在代码中。除了进行 GUI 布局，界面构建器还允许连接不同的组件，无需编写大量代码来建立关系。下面深入分析一下项目的分镜文件，看一下各项内容是如何设置的。

选择 MainStoryboard.storyboard，在界面构建器中打开它，这个分镜包含了大量内容，你应该切换到文件大纲来达到最佳查看效果（参见图 11-6）。使用编辑器右下角的控制器放大视角也可以帮助你纵观全局。

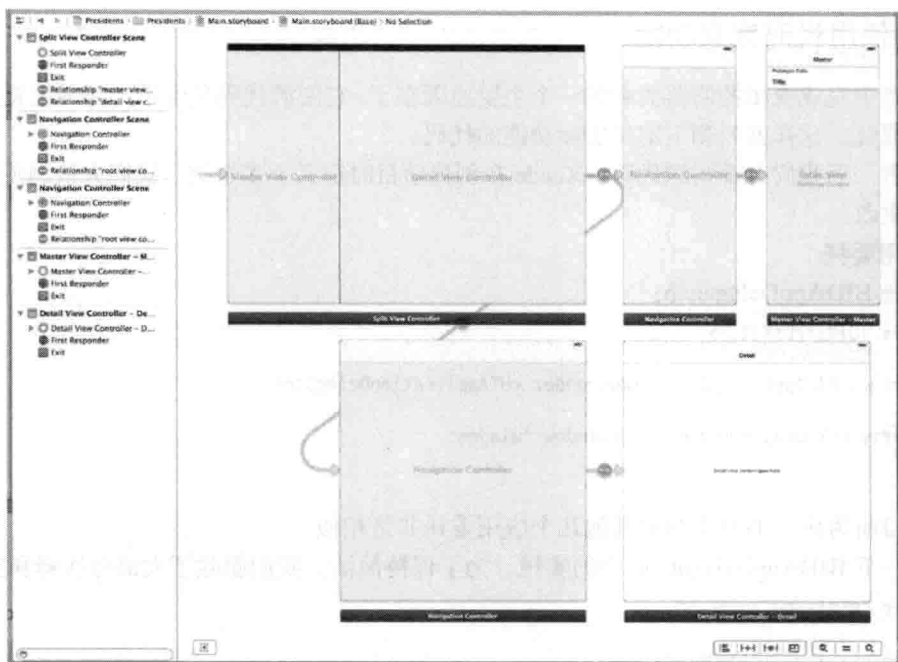


图 11-6 在界面构建器中打开的 MainStoryboard.storyboard, 最好在列表模式下查看这些复杂的对象层次

打开连接检查器, 花些时间点击每个视图控制器, 以便更清晰地了解它们之间的关联关系。

分割视图控制器和两个导航控制器对象一开始都与一个或多个其他的控制器相关联, 如连接检查器中 Storyboard Segues 部分所示。通过第 9 章的学习, 你已经熟悉了这些关联类型, 包括每个 UINavigationController 都有的 `rootViewController` 关系。这里你可以发现, `UISplitViewController` 实际上与其他控制器存在两个关联关系: `masterViewController` 和 `detailViewController`。`UISplitViewController` 用它们来显示左边栏或者弹出窗口中的内容 (`masterViewController`), 以及右边更大的显示区域中的内容 (`detailViewController`)。请注意 `masterViewController` 和 `detailViewController` 是 `UINavigationController` 的实例对象, 而不是包含了应用 GUI 的特定类。每个导航控制器分别拥有一个 `rootViewController` 变量关联到其他显示实际内容的视图控制器。通过这种方式使用导航控制器是很普遍的, 即便在像这种实际上没有怎么利用导航控制器关键特性的应用中也是如此。这是因为除了能够让控制器入栈出栈, 导航控制器在顶端提供了内置的导航栏, 它可以用来显示标题以及我们在示例中的一些按钮。导航栏还包含了 iOS 7 标志性的毛玻璃半透明背景效果。

目前, `MainStoryboard.storyboard` 中的内容实际上定义了应用中各个控制器之间的关联, 很多使用分镜的例子都有所体现。使用分镜可以减少大量的代码, 这通常都是好事。如果你喜欢使用代码来做这些配置, 那么可以这么做, 但是在这个例子中, 我们坚持使用 Xcode 提供的内容。

### 11.1.3 使用代码定义功能

在分镜中完成视图控制器关联的一个主要原因在于，它使源代码免受不必要的配置信息的影响而变得混乱，这样就只剩下定义实际功能的代码。

首先看一下我们有哪些源代码。Xcode 在创建项目时定义了多个类，我们大致浏览一下，然后再进行更改。

#### 1. 应用委托

首先是 BIDAppDelegate.h:

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

@interface BIDAppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;

@end
```

它与目前为止本书中介绍的其他几个应用委托非常相似。

再看一下 BIDAppDelegate.m 中的实现。为了保持简洁，我们删除了大部分注释和空方法):

```
#import "BIDAppDelegate.h"

@implementation BIDAppDelegate

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // 程序启动之后的一些自定义设置
    UISplitViewController *splitViewController =
        (UISplitViewController *)self.window.rootViewController;
    UINavigationController *navigationController =
        [splitViewController.viewControllers lastObject];
    splitViewController.delegate =
        (id)navigationController.topViewController;
    return YES;
}

@end
```

这段代码实际上就做了一事：设置 UISplitViewController 的 delegate 属性，使其指向主要显示部分（图 11-6 中名为 Detail 的视图）的控制器。本章稍后深入讨论分割视图时，将会分析 UISplitViewController 委托的关联逻辑。但是为什么这里要在代码中创建连接，而不是直接在分镜中创建？毕竟前面段落才刚说过能够去除乏味的代码（例如“将这个物体关联至那个物体”）是 nib 和分镜的主要优势之一。而且你也已经多次看到在界面构建器中进行这类关联了，那么为什么这里不能这么做呢？

要理解这里不能使用分镜来建立关联的原因，需要了解分镜和 nib 文件的差别。因为大部分时候使用的是分镜，本书中只有很少的地方用到了 nib 文件，不过它们仍然是存在的，并且两者有很多区别需要注意。

nib 文件实际上是静态对象图 (frozen object graph)。当向运行中的应用加载 nib 时, 它所包含的对象全都会加载并且一直存在, 包括 nib 文件中指定的所有连接。系统将依次为 nib 文件中每个单独对象创建一个全新的实例, 以及关联所有的输出接口和对象。

而分镜则不完全是这样。你可以认为分镜中的每个场景大致上都是相应的 nib 文件。当添加了元数据以描述场景如何通过转场互相关联之后, 就得到了一个分镜。不像单个 nib 文件, 一个复杂的分镜通常不会一次加载所有内容。相反, 任何使新场景获得焦点的行为都会导致分镜停止加载特定场景的静态对象图。这就意味着你在使用分镜时看到的对象并不一定同时都存在。

由于界面构建器不知道哪些场景将会共存, 所以它其实禁止你从一个场景中的对象向另一个场景中的对象关联任何输出接口或者目标/操作。事实上能在不同场景之间进行关联的只有转场。

不要光听我们说, 自己动手实践一下吧! 首先, 在分镜中选择 Split View Controller (也可以在 dock 中的 Split View Controller Scene 中找到它)。打开连接检查器, 试着从 delegate 输出接口拖向另一个视图控制器或者对象。可以将它拖到布局视图和列表视图上面, 但是你无法找到任何高亮显示 (这代表该项能够接受一项关联) 的项。

所以, 我们需要在代码中将 delegate 输出接口从 UISplitViewController 关联至目的控制器。回到 BIDAppDelegate.m, 接下来是:

```
UISplitViewController *splitViewController =
    (UISplitViewController *)self.window.rootViewController;
```

这行代码用于获取窗口的 rootViewController, 它就是在分镜中通过自由浮动箭头指向的 UISplitViewController 实例。代码如下所示:

```
UINavigationController *navigationController =
    [splitViewController.viewControllers lastObject];
```

在这一行中, 我们获取了 UISplitViewController 的 viewControllers 数组, 我们恰好知道它总是包含 2 个视图控制器: 一个用于左边栏, 一个用于右边栏 (稍后将详细介绍)。获取右边那个包含详情视图的控制器。再看最后的代码:

```
splitViewController.delegate =
    (id)navigationController.topViewController;
```

这最后一行仅仅将详情视图控制器赋值给了 UISplitViewController 的 delegate。

总的来说, 这一点额外代码相对于使用分镜所省去的其他大量代码来说, 确实微不足道。

## 2. 主视图控制器

现在, 我们来看一下 BIDMasterViewController, 它用于创建包含应用导航的表视图。BIDMasterViewController.h 如下所示:

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

@class BIDDetailViewController;

@interface BIDMasterViewController : UITableViewController

@property (strong, nonatomic) BIDDetailViewController *detailViewController;

@end
```

该头文件所对应的 BIDMasterViewController.m 文件开头部分如下所示(我们现在只看最前面的几个方法,剩下的后面再讨论):

```
#import "BIDMasterViewController.h"

#import "BIDDetailViewController.h"

@interface BIDMasterViewController () {
    NSMutableArray *_objects;
}
@end

@implementation BIDMasterViewController

- (void)awakeFromNib
{
    self.clearsSelectionOnViewWillAppear = NO;
    self.preferredContentSize = CGSizeMake(320.0, 600.0);
    [super awakeFromNib];
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后 (通常是从 nib 文件加载), 做一些额外的设置
    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    UIBarButtonItem *addButton =
        [[UIBarButtonItem alloc]
         initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
         target:self
         action:@selector(insertNewObject:)];
    self.navigationItem.rightBarButtonItem = addButton;
    self.detailViewController =
        (BIDDetailViewController *)[self.splitViewController.viewControllers
                                     lastObject] topViewController];
}
.
.
.
@end
```

这里进行了大量配置,幸好,Xcode 将这些配置作为分割视图模板的一部分提供给你了。这段代码包含了一些之前可能没见过的 iPad 相关内容。

首先,awakeFromNib 方法的第一行如下所示:

```
self.clearsSelectionOnViewWillAppear = NO;
```

clearsSelectionOnViewWillAppear 属性在 UITableViewController 类中 (BIDMasterViewController 的父类) 定义,用于微调控制器的行为。默认情况下,UITableViewController 在每次显示时都会取消选择所有行。这在 iPhone 应用中可能没问题,因为每个表视图的显示通常只依赖于其自身,但是在 iPad 应用中,表视图表现为一个分割视图,也许你并不希望所选项消失。回顾一



下前面的例子,考虑邮件应用,用户在左侧栏选择一封邮件,并且希望即使在邮件列表消失后(可能因为旋转了 iPad,或者关闭了包含该列表的弹出窗口),该选项也能保持选中状态。这一行代码就能解决这个问题。

`awakeFromNib` 方法中有一行用于设置视图的 `preferredContentSize` 属性。如果要使用该视图控制器负责显示其他允许设定尺寸的视图控制器,可以通过这个属性来设置视图的尺寸。在这个示例中,显示的是纵向模式下包含了主视图控制器的弹出窗口控制器。这个矩形的宽度必须至少为 320 像素,除了这个限制,你可以随意设置其大小。本章稍后将更详细地讨论弹出窗口的内容。

最后需要提一下 `viewDidLoad` 方法。在前几章里,当你要实现一个与用户选择的行对应的表视图控制器时,典型做法是创建一个新的视图控制器,然后将其压入导航控制器栈。然而在这个应用中,我们想要显示的视图控制器一开始就存在,而且每当用户在左侧栏中选择某项时,都会重用该视图控制器。它是包含在分镜文件中的 `BIDDetailViewController` 的实例。这里,我们获取该 `BIDDetailViewController` 实例并将其赋给一个属性变量,当有一些内容需要显示时,会用到它。

这个类的模板中还包含了很多方法,不过目前不需要担心它们。当完成了详情视图控制器的章节之后,我们会删除一部分内容并重新编写剩余的代码。

### 3. 详情视图控制器

Xcode 创建的最后一个类是 `BIDDetailViewController`,它负责实际显示用户所选项。`BIDDetailViewController.h` 文件类的内容如下所示:

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

@interface BIDDetailViewController : UIViewController <UISplitViewControllerDelegate>

@property (strong, nonatomic) id detailItem;

@property (weak, nonatomic) IBOutlet UILabel *detailDescriptionLabel;
@end
```

除了前面引用的 `detailItem` 属性(在 `BIDMasterViewController` 类中),`BIDDetailViewController` 有一个输出接口用于连接到分镜中的标签(`detailDescriptionLabel`)。

切换到 `BIDDetailViewController.m`,在其中可以找到以下内容(再说一遍,这里进行了一定的删改):

```
#import "BIDDetailViewController.h"

@interface BIDDetailViewController ()
@property (strong, nonatomic) UIPopoverController *masterPopoverController;
- (void)configureView;
@end

@implementation BIDDetailViewController

#pragma mark - Managing the detail item

- (void)setDetailItem:(id)newDetailItem
{
    if ( detailItem != newDetailItem) {
```

```

        _detailItem = newDetailItem;
        [self configureView];
    }
    if (self.masterPopoverController != nil) {
        [self.masterPopoverController dismissPopoverAnimated:YES];
    }
}

- (void)configureView
{
    if (self.detailItem) {
        self.detailDescriptionLabel.text = [self.detailItem description];
    }
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    [self configureView];
}

#pragma mark - Split view

- (void)splitViewController:(UISplitViewController *)splitController
willHideViewController:(UIViewController *)viewController
withBarButtonItem:(UIBarButtonItem *)barButtonItem
forPopoverController:(UIPopoverController *)popoverController
{
    barButtonItem.title = NSLocalizedString(@"Master", @"Master");
    [self.navigationItem setLeftBarButtonItem:barButtonItem animated:YES];
    self.masterPopoverController = popoverController;
}

- (void)splitViewController:(UISplitViewController *)splitController
willShowViewController:(UIViewController *)viewController
invalidatingBarButtonItem:(UIBarButtonItem *)barButtonItem
{
    [self.navigationItem setLeftBarButtonItem:nil animated:YES];
    self.masterPopoverController = nil;
}

@end

```

你应该熟悉这段代码中的大部分内容，但是这个类包含一些值得注意的新内容。首先是类扩展，它在靠近文件顶部的地方声明：

```

@interface BIDDetailViewController ()
@property (strong, nonatomic) UIPopoverController *masterPopoverController;
- (void)configureView;
@end

```

前面提到过类扩展，不过有必要再提一下它们的用途。可创建类扩展来定义这样一些方法和属性：它们将在一个类中使用，但你不希望向头文件中的其他类公开它们。这里声明了 `masterPopoverController` 属性和一个实用程序方法，在需要更新显示时将会调用该实用程序方法。我们还未告诉你 `masterPopoverController` 属性的用途，但你很快就会看到！

继续往下看，可以看到这样一个方法：

```
- (void)setDetailItem:(id)newDetailItem
{
    if (_detailItem != newDetailItem) {
        _detailItem = newDetailItem;
        [self configureView];
    }
    if (self.masterPopoverController != nil) {
        [self.masterPopoverController dismissPopoverAnimated:YES];
    }
}
```

你可能会对 `setDetailItem:` 方法感到惊奇。毕竟我们将 `detailItem` 定义为一个属性，同时自动合成了赋值方法和取值方法，那么为什么在代码中创建赋值方法呢？本例需要能在用户调用赋值方法时作出反应（选择左侧主列表中的一行），以便可以更新显示，这是达到此目的的一种不错的途径。该方法的第一部分看起来很简单，但在最后它采用了一个调用来解除当前的 `masterPopoverController`（如果存在）。虚构的 `masterPopoverController` 来自何处？答案就在下一个方法中：

```
- (void)splitViewController:(UISplitViewController *)splitController
willHideViewController:(UIViewController *)viewController
withBarButtonItem:(UIBarButtonItem *)barButtonItem
forPopoverController:(UIPopoverController *)popoverController
{
    barButtonItem.title = NSLocalizedString(@"Master", @"Master");
    [self.navigationItem setLeftBarButtonItem:barButtonItem animated:YES];
    self.masterPopoverController = popoverController;
}
```

这是 `UISplitViewController` 的一个委托方法。当分割视图控制器不再固定显示分割视图左侧时（也就是当 iPad 旋转到纵向时）将调用它。这个方法首先使用 `NSLocalizedString` 函数配置 `barButtonItem` 中显示的标题，你可以通过该函数使用其他语言的文本字符串（如果你准备好了的话）。第 22 章将详细讨论本地化问题，但是目前，你只需知道其中一个参数实际上是个键，该函数使用它从一个字典中获取本地化字符串，而另一个参数则只是个注释。

左侧的分割视图消失之前，分割视图控制器在委托中调用此方法并传递两个有趣的项：`UIPopoverController` 和 `UIBarButtonItem`。已预先配置 `UIPopoverController` 来包含分割视图左侧的内容，设置 `UIBarButtonItem` 显示相同的浮动窗口。这意味着，如果 GUI 包含 `UIToolbar` 或者 `UINavigationController`（由 `UINavigationController` 显示的标准工具栏），我们只需在工具栏上添加一个按钮，用户点击该按钮即可调出包含在浮动窗口中的导航视图。

在这个例子中，由于该控制器本身封装在 `UINavigationController` 中，所以可以直接访问 `UINavigationController`，在其中放置按钮。如果 GUI 未包含 `UINavigationController` 或 `UIToolbar`，还有传入的浮动窗口控制器，可以将它分配给 GUI 的其他某个元素，使该元素可以弹出该浮动窗口。我们还传入了包装好的 `UIViewController` 本身（在本例中为 `BIDMasterViewController`），以防需要以完全不同的方式显示它的内容。

浮动窗口控制器就是这么出现的。也许你已经猜到，下一个方法将解除该浮动窗口：

```

- (void)splitViewController:(UISplitViewController *)splitController
willShowViewController:(UIViewController *)viewController
invalidatingBarButtonItem:(UIBarButtonItem *)barButtonItem
{
    [self.navigationItem setLeftBarButtonItem:nil animated:YES];
    self.masterPopoverController = nil;
}

```

当用户切换回横向模式时将调用该方法,这时分割视图控制器希望重新在固定位置绘制左侧视图,所以它告诉我们删除之前提供的 `UIBarButtonItem`。

Xcode 的 Master-Detail Application 模板提供的内容就大体介绍完了。乍看起来可能很难掌握,但通过一次展示一部分,我们希望你已经理解各个部分是如何结合在一起的。

## 11.2 显示总统信息

前面介绍了此项目的基本布局,是时候“填补空白”并将这个自动生成的应用转换为我们自己的东西了。首先看一下本书的源代码归档文件,其中的文件夹 11-Presidents 包含一个名为 `PresidentList.plist` 的文件。将该文件拖到 Xcode 中你的项目的 Presidents 文件夹中,将它添加到项目中,确保告诉 Xcode 复制文件本身的复选框处于选中状态。这个 `plist` 文件包含到目前为止美国历届总统的信息,由每位总统的姓名和维基百科条目的 URL 组成。

现在看一下 `BIDMasterViewController` 类,并看看要如何修改它从而恰当处理总统数据。只需要加载总统列表,在表视图中显示它们,以及向详细信息视图传递一个 URL 供显示。在 `BIDMasterViewController.m` 中,向类扩展添加以下以粗体显示的代码并移除有删除线的部分:

```

#import "BIDMasterViewController.h"

#import "BIDDetailViewController.h"

@interface BIDMasterViewController () {
    NSMutableArray *_objects;
}
@property (copy, nonatomic) NSArray *presidents;
@end

```

我们创建了一个自己的数组并将它作为属性变量。而没有使用 Xcode 自动生成的可变数组来存放总统列表的数据。

现在将你的注意力转移到 `viewDidLoad` 方法上,这里的更改稍微复杂一点(不过并不是太难)。你要添加一些代码来加载总统列表,然后移除设置工具栏上编辑按钮和插入按钮的其他语句:

```

- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后 (通常是从 nib 文件加载), 做一些额外的设置
    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    NSString *path = [[NSBundle mainBundle] pathForResource:@"PresidentList"
                                                            ofType:@"plist"];
    NSDictionary *presidentInfo = [NSDictionary

```

```

        dictionaryWithContentsOfFile:path];
        self.presidents = [presidentInfo objectForKey:@"presidents"];

        UIBarButtonItem *addButton =
        [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
        target:self
        action:@selector(insertNewObject:)];

        self.navigationItem.rightBarButtonItem = addButton;
        self.detailViewController =
        (BIDDetailViewController *)[self.splitViewController.viewControllers
        lastObject] topViewController];
    }

```

模板生成的类还包含一个名称为 `insertNewObject:` 的方法,它用来向 `_objects` 数组添加内容。现在这个数组已经不存在了,所以我们将这个方法整段删除:

```

- (void)insertNewObject:(id)sender
{
    if (!_objects) {
        _objects = [[NSMutableArray alloc] init];
    }
    [_objects insertObject:[NSDate date] atIndex:0];
    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:0 inSection:0];
    [self.tableView insertRowsAtIndexPaths:@[indexPath]
    withRowAnimation:UITableViewRowAnimationAutomatic];
}

```

这里还有两个可以让用户编辑表视图中的行的数据源方法。在这个应用中我们不会允许用户编辑行,因此在添加自己的代码之前,先来移除这些代码:

```

- (BOOL)tableView:(UITableView *)tableView
canEditRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Return NO if you do not want the specified item to be editable.
    return YES;
}

- (void)tableView:(UITableView *)tableView commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (editingStyle == UITableViewCellEditingStyleDelete) {
        [_objects removeObjectAtIndex:indexPath.row];
        [tableView deleteRowsAtIndexPaths:@[indexPath]
        withRowAnimation:UITableViewRowAnimationFade];
    } else if (editingStyle == UITableViewCellEditingStyleInsert) {
    }
}

```

现在都是些主要的表视图数据源方法,要针对我们的应用进行调整。首先需要让表视图知道每个分区有多少行数据需要显示,因此先修改下面这个方法:

```

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
    return [_objects count];
}

```

```

    return [self.presidents count];
}

```

然后修改 tableView:cellForRowAtIndexPath:方法, 让每个表视图单元显示一位总统的姓名:

```

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:@"Cell"
        forIndexPath:indexPath];
    NSDate *object = _objects[indexPath.row];
    cell.textLabel.text = [object description];

    NSDictionary *president = self.presidents[indexPath.row];
    cell.textLabel.text = president[@"name"];
    return cell;
}

```

最后, 更新 tableView:didSelectRowAtIndexPath:的行为, 以便将 URL 传递给详细信息视图控制器:

```

- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSDate *object = _objects[indexPath.row];
    self.detailViewController.detailItem = object;
    NSDictionary *president = self.presidents[indexPath.row];
    NSString *urlString = president[@"url"];
    self.detailViewController.detailItem = urlString;
}

```

这就是需要对 BIDMasterViewController 进行的所有修改。现在就可以构建并运行这个应用了。旋转至横向模式, 或点击左上角的 Master 按钮打开一个包含总统列表的弹出窗口 (参见图 11-7), 然后点击一位总统的名字, 在详情视图中显示该总统的维基百科页面 URL。

在本节最后, 让详情视图对该 URL 执行一些更加实用的事情。首先, 在 BIDDetailView-Controller.h 中, 添加一个 Web 视图输出接口, 用于显示所选总统的维基百科页面。添加以下粗体显示的代码:

```

#import <UIKit/UIKit.h>

@interface BIDDetailViewController : UIViewController <UISplitViewControllerDelegate>

@property (strong, nonatomic) id detailItem;
@property (weak, nonatomic) IBOutlet UILabel *detailDescriptionLabel;
@property (weak, nonatomic) IBOutlet UIWebView *webView;
@end

```

然后切换到 BIDDetailViewController.m, 这里需要做的事情稍微多一些 (实际上也不是很多)。向下滚动到 configureView 方法, 添加以下粗体显示的方法:

```

- (void)configureView
{
    NSURL *url = [NSURL URLWithString:self.detailItem];
}

```

```

NSURLRequest *request = [NSURLRequest requestWithURL:url];
[self.webView loadRequest:request];
if (self.detailItem) {
    self.detailDescriptionLabel.text = [self.detailItem description];
}
}

```

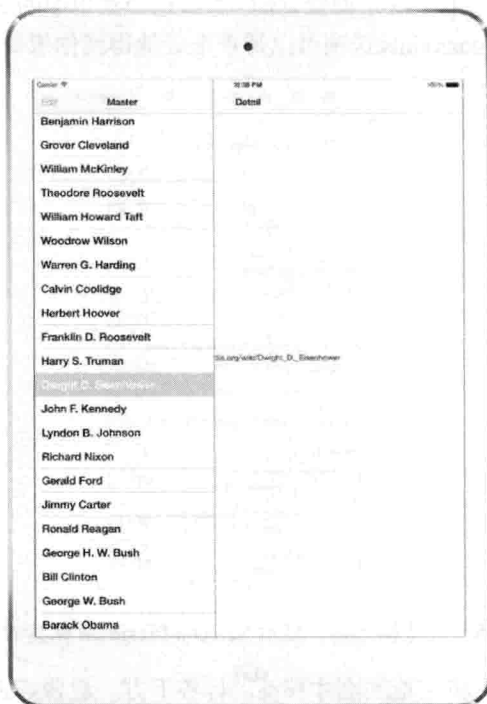


图 11-7 Presidents 应用的第一次运行。注意，我们按下 Master 按钮打开了浮动弹出窗口。点击一位总统的名字，就会显示该总统的维基百科链接

这些新代码用于让 Web 视图加载请求的页面。

接下来请移动到 `splitViewController:willHideViewController:withBarButtonItem:forPopoverController:` 方法，我们将在其中为 `UIBarButtonItem` 提供一个更合适的标题：

```

—— barButtonItem.title = NSLocalizedString(@"Master", @"Master");
   barButtonItem.title = NSLocalizedString(@"Presidents", @"Presidents");

```

无论你信不信，这些就是到目前为止我们需要编写的所有代码。

最后需要在 `MainStoryboard.storyboard` 中进行更改。打开它，在右下方找到详情视图，首先看一下 GUI 中的标签（它显示“Detail view content goes here”）。

首先选择该标签。在文件大纲中选择它是最容易的，它位于 Detail View Controller-Detail Scene 部分。在文件大纲的搜索框中键入 label 可以很快找到它。

选择该标签后，将其拖到窗口顶部。注意，这个标签应该从左侧开始一直延伸到右侧蓝色引

导线处，并且恰好位于导航栏下方。这个标签用于显示当前 URL。但是当应用刚启动、用户选择一位总统之前，我们想要用它提示用户进行操作。

双击该标签，将其内容改为 **Select a President**。你应该使用尺寸检查器，以确保将该标签的位置约束在父视图的左侧边缘、右侧边缘以及顶部边缘处（参见图 11-8）。如果你需要调整这里的约束，可以使用前面描述过的方法来设置它们。通过选择菜单中的 **Editor>Resolve Auto Layout Issues>Reset to Suggested Constraints** 选项可以近乎准确地得到你想要的效果。

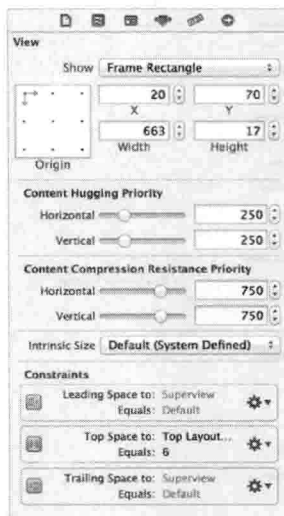


图 11-8 尺寸检查器，显示 **Select a President** 标签的设置

从库中找到 **UIWebView**，将它拖到刚才移动的标签下方。放置好之后，使用调整手柄使它适合标签下方剩余的视图空间。将它与左右两边对齐，覆盖从标签底部的蓝色参考线到窗口底部的空间。接下来，使用尺寸检查器将这个 **Web** 视图约束在父视图的左侧边缘、右侧边缘和底部边缘，还需要约束顶部边缘与标签之间的距离（参见图 11-9）。因为很重要，所以再说一次，通过选择菜单中的 **Editor>Resolve Auto Layout Issues>Reset to Suggested Constraints** 选项可以近乎准确地得到你想要的效果。

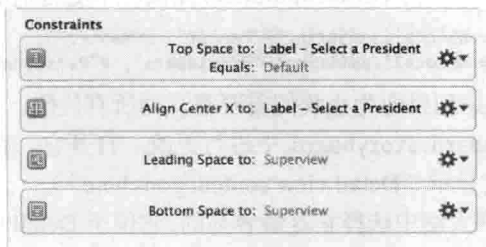


图 11-9 尺寸检查器中所显示的 **Web** 视图的约束设置



最后还有一处需要修改。要连接刚刚创建的输出接口，按住 control 键并从 dock 中的 Detail View Controller 图标（在 Detail View Controller-Detail 部分，就在 First Responder 图标下面）拖到新的 Web 视图，连接 webView 输出接口（在同一部分中，就在标签下方）。保存更改之后，你的工作就完成了！

现在可以构建并运行应用了，它将显示每位总统的维基百科条目。在两个方向上旋转显示屏，将会看到分割视图控制器为你处理了所有事务，在一定程度上还借助了详情视图控制器来处理显示浮动窗口所需的工具栏项（就像在更改之前的原始应用中一样）。

本节最后要进行的更改实际上就是进行美化。在横向模式下运行此应用时，左侧导航视图上方的标题为 Master。切换到纵向模式并单击 Presidents 工具栏按钮，也会看到相同的标题。

要更改标题，可打开 MainStoryboard.storyboard，双击窗口右上方表视图上的导航栏，然后双击视图上显示的文本，将它更改为 Presidents（参见图 11-10）。保存分镜文件，构建并运行应用，应该会看到更改生效了。

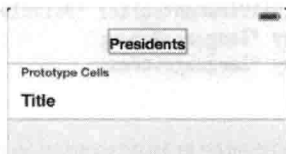


图 11-10 MainStoryboard.storyboard 的当前状态。请注意我们将 master-detail 视图的表视图标题改为了 Presidents

## 11.3 创建浮动窗口

前面进展非常顺利，但 iPad GUI 技术中还有一部分介绍得不够详细：浮动窗口的创建和显示。到目前为止，我们仅仅从 UISplitView 委托方法获得了一个 UIPopoverController，它使我们能够在实例变量中跟踪它，以便能够强制解除它，但你希望呈现自己的视图控制器时，浮动窗口才会真正发挥作用。

为了熟悉它的工作原理，我们将自行添加一个浮动窗口，它将由一个始终存在的工具栏项激活（与 UISplitView 委托方法提供的浮动窗口不同，它只会自行显示并消失）。此浮动窗口将显示一个表视图，其中包含一个语言列表。如果用户从该列表中选择一种语言，Web 视图将加载已使用这种新语言显示的维基百科条目。这非常简单，因为要在维基百科中从一种语言切换到另一种，只需更改 URL 中嵌入的国家代码部分。

**注意** 本例中两种浮动窗口的用途都是显示 UITableView，但不要让这误导了你，UIPopoverController 可用于处理你想要的任何视图控制器内容的显示！我们在本例中坚持使用表视图，这是因为它是一种常见的使用情形，很容易使用相对较少的代码来显示，并且你应该已经对它很熟悉。

首先右击 Xcode 中的 Presidents 文件夹, 从上下文菜单中选择 New File ...。当出现向导时, 依次选择 Cocoa Touch 和 Objective-C class, 然后点击 Next。在下一个页面中, 将新类命名为 BIDLanguageListController, 在 Subclass of 字段选择 UITableViewController。勾选 Targeted for iPad 复选框, 取消选中 With XIB for user interface 复选框。单击 Next 按钮, 再次检查保存文件的路径, 点击 Create。

BIDLanguageListController 是一个非常标准的表视图控制器类。它将显示一组列表项, 使用详细信息视图控制器指针, 让详细信息视图控制器知道何时进行了选择。编辑 BIDLanguageListController.h, 添加以下粗体显示的代码:

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

@class BIDDetailViewController;

@interface BIDLanguageListController : UITableViewController

@property (weak, nonatomic) BIDDetailViewController *detailViewController;
@property (copy, nonatomic) NSArray *languageNames;
@property (copy, nonatomic) NSArray *languageCodes;

@end
```

这里添加的代码定义了一个指向详细信息视图控制器的指针(我们将在显示语言列表时在详细信息视图控制器自身的代码中设置), 以及两个数组, 其中包含将显示的值(English、French 等)以及对应的用于构建 URL 的值(en、fr 等)。请注意我们声明的这些数组使用 copy 标识代替了 strong。这意味着代码中无论何时调用某个 setter 方法, 参数会发送一个 copy 消息, 因此保存的并不是作为 strong 类型的指针。这样可以防止某一个类传递的不是 NSArray, 而是 NSMutableArray 类型, 并可能在我们不知情时改变数组内容的情况。发送 copy 给 NSMutableArray 实例变量总是会返回一个不可变的 NSArray 数组, 这样我们就能知道当前使用的这个数组不会受其他因素而改变。如果向一个 NSArray 发送 copy, 因为它已经是不可变的, 所以实际不会创建一个新的拷贝, 只是返回一个指向 self 的 strong 类型指针, 由此任何时候发送 copy 消息都不会引起多余消耗。

如果从本书源代码归档文件(或电子版)中将该代码复制并粘贴到你自己的项目中, 或者如果自行键入它时未加留意, 你可能注意不到 detailViewController 属性在声明方式上的重要区别。与大多数引用对象指针的属性不同, 我们使用 weak 而不是 strong 声明此属性。必须这样做才能避免保留连环(retain cycle)。

什么是保留连环? 它指的是这样一种情形: 两个或更多对象以一种连环的方式相互持有。每个对象有一个值为 1 或更大值的引用计数, 因此它包含的指针永远不会释放, 所以保留连环中的对象也永远不会被销毁。谨慎考虑对象的创建, 通常通过确定哪个对象“拥有”谁, 可以避免大多数潜在的保留连环。在这种意义上, BIDDetailViewController 的实例“拥有”BIDLanguageListController 的实例, 因为正是这个 BIDDetailViewController 实际创建 BIDLanguageListController 来完成部分工作。如果有两个需要相互引用的对象, 通常希望“所有者”对象持有另一个对象, 但另一个对象

绝对不能持有它的所有者。由于我们使用了苹果在 Xcode 4.2 中引入的 ARC 特性, 编译器会为我们做很多工作。不用关心对象释放和对对象保持的细节问题, 我们要做的就是用 `weak` 关键字 (而不是 `strong`) 来声明一个属性, 余下的事交给 ARC 就可以了!

现在切换到 `BIDLanguageListController.m`, 进行以下更改。在文件顶部, 首先导入 `BIDDetailViewController` 的头文件:

```
#import "BIDLanguageListController.h"
#import "BIDDetailViewController.h"
```

```
.
```

然后向下滚动到 `viewDidLoad` 方法, 并添加一些设置代码:

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

    self.languageNames = @[@"English", @"French", @"German", @"Spanish"];
    self.languageCodes = @[@"en", @"fr", @"de", @"es"];
    self.clearsSelectionOnViewWillAppear = NO;
    self.preferredContentSize = CGSizeMake(320.0,
                                             [self.languageCodes count] * 44.0);

    [self.tableView registerClass:[UITableViewCell class]
      forCellReuseIdentifier:@"Cell"];
}
```

这段代码设置了语言数组, 还定义了此视图在浮动窗口中显示时 (我们知道它将在浮动窗口中显示) 使用的尺寸。如果没有定义这个尺寸, 得到的浮动窗口会垂直拉伸至几乎填满整个屏幕, 即便它显示的全部内容是自己要小的视图。最后, 如在第 8 章中那样, 注册一个默认的表视图单元类。

接下来是两个由 Xcode 模板生成的方法, 但是其中没有包含有效的代码, 只是一个警告和一些占位符文本。我们将这些文本替换为实际的代码:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    #warning Potentially incomplete method implementation.
    return 0;
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
  numberOfRowsInSection:(NSInteger)section
{
    #warning Incomplete method implementation.
    // 返回分区中的行数
    return 0;
    return [self.languageCodes count];
}
```

然后 `tableView:cellForRowAtIndexPath:` 末尾添加一行代码, 将语言名称添加到单元格中:

```
// 配置单元格
cell.textLabel.text = self.languageNames[indexPath.row];
return cell;
```

接下来, 实现 `tableView:didSelectRowAtIndexPath:` 方法, 使其能回应用户的触摸, 传递所选语言到详情视图控制器中:

```
-(void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    self.detailViewController.languageString =
        self.languageCodes[indexPath.row];
}
```

---

**注意** `BIDDetailViewController` 实际上没有 `languageString` 属性, 因此会得到一个编译器错误。我们稍后将探讨这一主题。

---

现在对 `BIDDetailViewController` 进行一些必要的更改以便处理浮动窗口, 以及在用户更改显示语言或选中不同的总统时生成正确的 URL。首先在 `BIDDetailViewController.h` 中进行以下更改:

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

@interface BIDDetailViewController : UIViewController <UISplitViewControllerDelegate,
    UIPopoverControllerDelegate>

@property (strong, nonatomic) id detailItem;

@property (weak, nonatomic) IBOutlet UILabel *detailDescriptionLabel;
@property (weak, nonatomic) IBOutlet UIWebView *webView;
@property (strong, nonatomic) UIBarButtonItem *languageButton;
@property (strong, nonatomic) UIPopoverController *languagePopoverController;
@property (copy, nonatomic) NSString *languageString;

@end
```

这里我们声明这个类遵循 `UIPopoverControllerDelegate` 协议 (可以处理弹出窗口控制器之后会发来的通知) 并添加一些属性变量以记录弹出窗口所需的 GUI 元素和用户所选的语言。现在需要做的是修改 `BIDDetailViewController.m`, 使它可以处理语言浮动窗口和 URL 的构造。首先将以下代码添加到顶部的某个位置:

```
#import "BIDLanguageListController.h"
```

我们要添加的下一项是一个函数, 它接受指向维基百科页面的 URL 以及一个双字母语言代码作为参数, 并返回一个结合了这两部分的 URL。我们稍后将在控制器代码中的恰当位置调用这个方法。可以将此函数放在任何地方, 包括类的实现内部。编译器非常聪明, 总是能以正确的方式识别以及处理函数。我们选择将它放在 `setDetailItem:` 方法之后。

```
static NSString * modifyUrlForLanguage(NSString *url, NSString *lang) {
    if (!lang) {
        return url;
    }

    // 下面的代码很脆弱，因为它依赖于特定的维基百科 URL 格式
    NSRange codeRange = NSMakeRange(7, 2);
    if ([[url substringWithRange:codeRange] isEqualToString:lang]) {
        return url;
    } else {
        NSString *newUrl = [url stringByReplacingCharactersInRange:codeRange
                                                                    withString:lang];
        return newUrl;
    }
}
```

为什么要将它创建为函数，而不是方法呢？有几点原因。但最重要的是，类中的实例方法通常用于执行涉及一个或多个实例变量的操作（也可能是通过 getter 和 setter 方法访问或者通过实例变量方式直接访问对象的内部状态）。此函数不使用任何实例变量。它仅在两个字符串上执行一项操作并返回另一个字符串。我们可以将它创建为类方法，但这让人感觉不太对劲，因为该方法执行的操作实际上与控制器类没有明确的关联。有时所需的只是一个函数。

接下来需要更新 `setDetailItem:` 方法。此方法将使用刚才定义的函数，将传入的 URL 与所选的 `languageString` 结合在一起，生成正确的 URL。它还可以确保第二个浮动窗口（如果显示的话）像第一个浮动窗口（我们之前定义的浮动窗口）一样消失。

```
- (void)setDetailItem:(id)newDetailItem
{
    if (self.detailItem != newDetailItem) {
        _detailItem = newDetailItem;
        _detailItem = modifyUrlForLanguage(newDetailItem, self.languageString);

        // 更新视图
        [self configureView];
    }

    if (self.masterPopoverController != nil) {
        [self.masterPopoverController dismissPopoverAnimated:YES];
    }
}
```

现在更改 `viewDidLoad` 方法。这里将创建一个 `UIBarButtonItem`，并将其置于屏幕顶部的 `UINavigationController` 中。

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.languageButton =
        [[UIBarButtonItem alloc] initWithTitle:@"Choose Language"
                                             style:UIBarButtonItemStyleBordered
                                             target:self
                                             action:@selector(toggleLanguagePopover)];
}
```

```

        self.navigationItem.rightBarButtonItem = self.languageButton;
        [self configureView];
    }

```

接下来实现 `setLanguageString:`。它也会调用 `modifyUrlForLanguage()` 函数, 从而立即重新生成一个 URL (并加载新页面)。将此方法添加到文件底部的 `@end` 上方。

```

- (void)setLanguageString:(NSString *)newString {
    if (![newString isEqualToString:self.languageString]) {
        _languageString = [newString copy];
        self.detailItem = modifyUrlForLanguage(_detailItem, self.languageString);
    }
    if (self.languagePopoverController != nil) {
        [self.languagePopoverController dismissPopoverAnimated:YES];
        self.languagePopoverController = nil;
    }
}

```

下面对用户点击 Choose Language 按钮时的事件进行处理。为简单起见, 我们创建一个 `BIDLanguageListController`, 将它包装在 `UIPopoverController` 中并显示。将此方法添加到 `viewDidLoad` 方法之后。

```

- (void)toggleLanguagePopover
{
    if (self.languagePopoverController == nil) {
        BIDLanguageListController *languageListController =
            [[BIDLanguageListController alloc] init];
        languageListController.detailViewController = self;
        UIPopoverController *poc =
            [[UIPopoverController alloc]
             initWithContentViewController:languageListController];
        [poc presentPopoverFromBarButtonItem:self.languageButton
         permittedArrowDirections:UIPopoverArrowDirectionAny
         animated:YES];
        self.languagePopoverController = poc;
    } else {
        if (self.languagePopoverController != nil) {
            [self.languagePopoverController dismissPopoverAnimated:YES];
            self.languagePopoverController = nil;
        }
    }
}

```

最后我们需要实现另一个方法来处理用户点击打开了语言弹出窗口, 然后再点击弹出窗口外其他地方以关闭它的情况。在这个示例中, `toggleLanguagePopover` 方法并没有被调用。不过我们可以实现在 `UIPopoverControllerDelegate` 协议中声明的方法。事件发生时它会进行通知, 并且关闭语言弹出窗口。

```

- (void)popoverControllerDidDismissPopover:
    (UIPopoverController *)popoverController
{
    if (popoverController == self.languagePopoverController) {
        self.languagePopoverController = nil;
    }
}

```

大功告成！现在应该能够完美地运行该应用，在总统和语言之间随意切换了。从一种语言切换到另一种应该始终保持所选的总统不变，同理从一位总统切换到另一位总统应该保持所选语言不变。

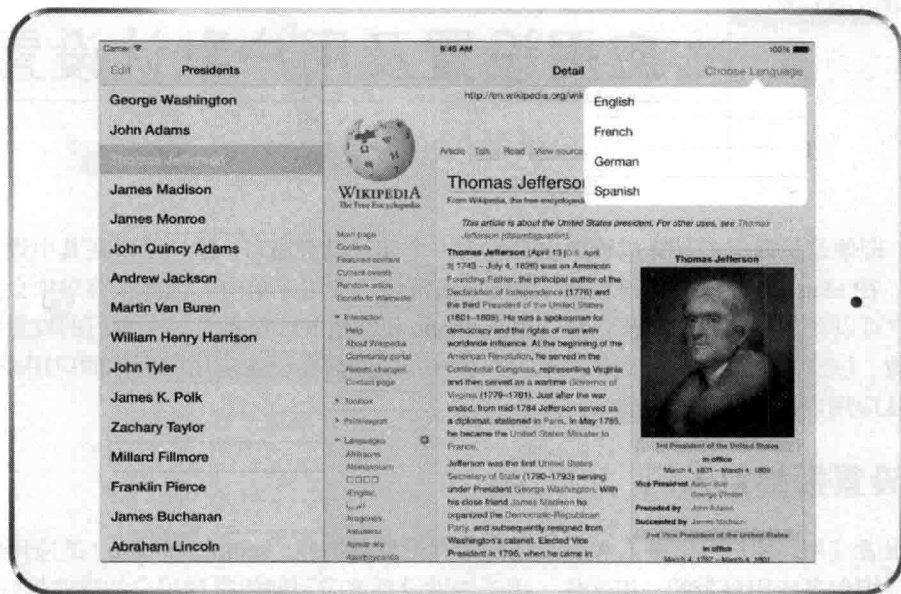


图 11-11 详情视图中显示托马斯·杰斐逊的维基百科页面，右上角显示语言列表浮动窗口

## 11.4 小结

本章介绍了仅可用于 iPad 的主要 GUI 组件——浮动窗口和分割视图，还介绍了一个完全在界面构建器中配置并且包含多个相互关联的视图控制器的复杂 iPad 应用示例。有了这些宝贵的知识，你应该可以非常顺利地开发第一个优秀的 iPad 应用了。如果你想更进一步了解 iPad 开发的细节，可以阅读 David Mark、Jack Nutting 和 Dave Wooldridge 编写的《iPad 开发基础教程》。

接着，我们进入下一章：应用设置及用户默认设置。

现今，即便是最简单的计算机程序也会包含一个偏好设置窗口，用户可以在其中设置应用专属的选项。在 Mac OS X 中，偏好设置...菜单项通常位于应用菜单中。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 和其他 iOS 设备有一个专门的设置应用来进行各种设置。该应用你肯定用过很多次了。本章将介绍如何在设置应用中为你的应用添加设置，以及如何从应用内部访问这些设置。

## 12.1 设置捆绑包入门

通过设置应用，用户可以输入和更改任何带有设置捆绑包（settings bundle）的应用的偏好设置。设置捆绑包是应用自带的一组文件，用于告诉设置该应用期望得到用户的哪些偏好设置。

先在 iOS 设备上找到设置应用的图标。点击该图标，启动设置应用。在我们的系统上，看起来如图 12-1 所示。



图 12-1 设置应用



设置应用充当着 iOS 用户默认设置（User Defaults）机制的通用用户界面的角色。用户默认设置是保存和获取偏好设置的系统的一部分。

在 iOS 应用中，用户默认设置由 `NSUserDefaults` 类实现。如果你在 Mac 上开发过 Cocoa 应用，那么可能比较熟悉 `NSUserDefaults`，因为在 Mac 上也是用这个类来保存和读取偏好设置。应用通过 `NSUserDefaults` 用键值对的方式来读取和保存偏好设置数据，就跟通过键从 `NSDictionary` 对象中获取数据一样。不同之处在于 `NSUserDefaults` 数据会被持久保存到文件系统中，而不是存储到内存中的对象实例中。

本章将开发一个应用，添加并配置一个设置捆绑包，然后从应用中访问并编辑这些偏好设置。

设置应用的优势之一是不用再为偏好设置设计用户界面。创建可描述应用有效设置的属性列表后，设置应用会自动创建用户界面。

游戏等沉浸式应用，通常应自行提供偏好设置视图，这样用户更改设置后不用退出应用就能使设置生效。即使是实用工具和生产效率应用，也应该允许用户在不离开应用的情况下更改偏好设置。我们还将介绍如何直接从应用用户处收集偏好设置，以及如何将其保存在 iOS 的用户默认设置中。

另一种情况是用户可以切换到设置应用，修改偏好设置，然后再切回到还在运行的应用。我们将在本章末尾演示如何处理这种情况。

## 12.2 应用：Bridge Control

本章中，我们构建一个简单的应用，来模拟控制星际飞船舰桥的一些功能，我敢肯定你会觉得它非常实用。首先创建一个设置捆绑包，这样当用户启动设置应用时，界面会显示一个指向这个应用的入口（参见图 12-2）。

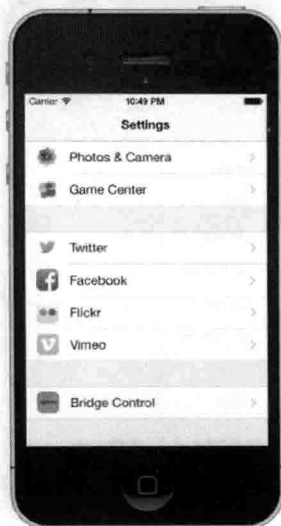


图 12-2 在模拟器中，设置应用显示了 Bridge Control 应用的一个入口

如果用户点击了应用的入口，系统会显示一个跟该应用有关的偏好设置视图。如图 12-3 所示，设置应用为用户提供了文本框、安全文本框、开关和滑动条。

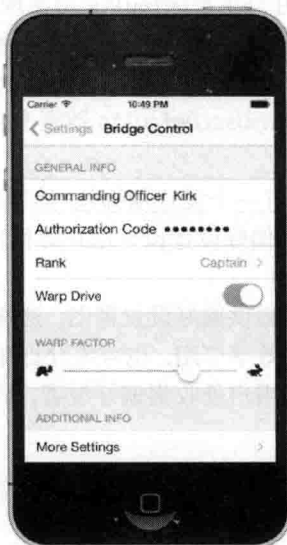


图 12-3 应用的主设置视图

此视图中还有两项包含了展开指示器。第一项是 Rank，通过它用户可以进入另一个表视图，其中显示该项目的可用选项。用户只能在该表视图选择单个值（参见图 12-4）。



图 12-4 从列表中选择单个偏好设置选项

在设置应用的主视图上还有一个展开指示器，那就是“更多设置”，利用它，用户可打开另一组偏好设置（参见图 12-5）。该子视图拥有的控件类型可以和主设置视图相同，甚至还可以有自己的子视图。设置应用需要用到导航控制器，因为它支持构建多级偏好设置视图。



图 12-5 应用的子设置视图

启动应用后，它会显示一组从设置应用中得到的偏好设置（参见图 12-6）。

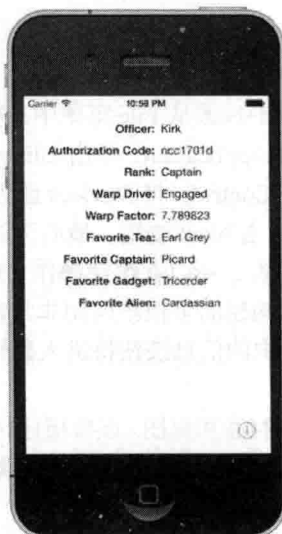


图 12-6 应用的主视图

为了演示如何在应用中更新偏好设置，我们还在右下角提供了一个较小的信息按钮，它会将用户带到另一个视图，以使用户直接在应用中修改其他偏好设置（参见图 12-7）。

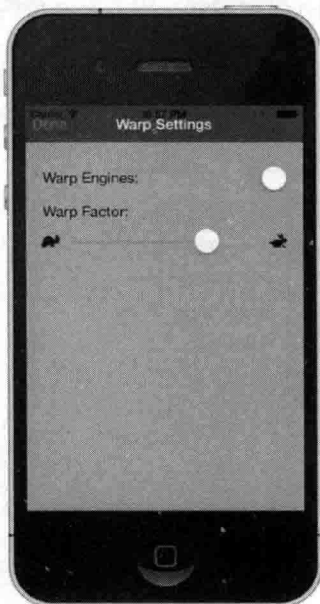


图 12-7 直接在应用中设置部分偏好设置

开始构建 Bridge Control 应用吧，准备好了吗？

### 12.2.1 创建项目

在 Xcode 中，按下 shift+command+N 或从 File 菜单中选择 New>Project…。当新项目向导出现后，选择左侧窗格中 iOS 标题下的 Application，单击 Utility Application 图标，点击 Next。在下一个页面中，将新项目命名为 Bridge Control。将 Devices 设为 iPhone。确保 BID 为类的前缀并取消勾选 Use Core Data 复选框，然后点击 Next 按钮。最后为该项目选择保存位置，点击 Create。

我们以前没有使用过这个项目模板，所以在继续操作之前，先熟悉一下这个新模板。Utility Application 模板创建的应用与第 6 章构建的多视图应用非常相似。此应用有一个主视图和一个翻转视图（flipside view）。单击主视图中的信息按钮将进入翻转视图，单击翻转视图中的 Done 按钮将返回主视图。

实现这种类型的应用需要几个控制器和视图。该模板已将这些控制器和视图组织到一些分组中。展开 Bridge Control 文件夹，你会看到常用的应用委托类以及另外两个控制器类和一个用来存储 GUI 的故事板文件，如图 12-8 所示。

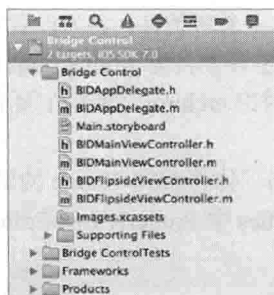


图 12-8 通过 Utility Application 模板创建的项目。注意应用委托、故事板以及主视图控制器和翻转视图控制器

## 12.2.2 使用设置捆绑包

设置应用使用每个应用的设置捆绑包的内容构建出一个应用的设置视图。如果应用没有设置捆绑包，则设置应用不会显示出它的任何信息。每个设置捆绑包必须包含一个名为 Root.plist 的属性列表，它定义了根级偏好设置视图。此属性列表必须遵循一种非常严格的格式，我们在设定应用设置捆绑包的属性列表时会讨论这种格式。

当设置应用启动时，它会检查每个应用的设置捆绑包并为包含设置捆绑包的每个应用添加设置组。如果你希望偏好设置包含子视图，那就必须向设置捆绑包添加属性列表，并在 Root.plist 中为每个子视图添加一个条目。本章稍后将详细介绍如何操作。

### 1. 在项目中添加设置捆绑包

在项目导航面板中，点击 Bridge Control 文件夹，然后选择 File>New>File...或者按下 command+N。在左侧窗格中，选择 iOS 部分中的 Resource，然后选择设置 Bundle 图标（参见图 12-9）。点击 Next 按钮，名字保留默认的设置.bundle，最后点击 Create。

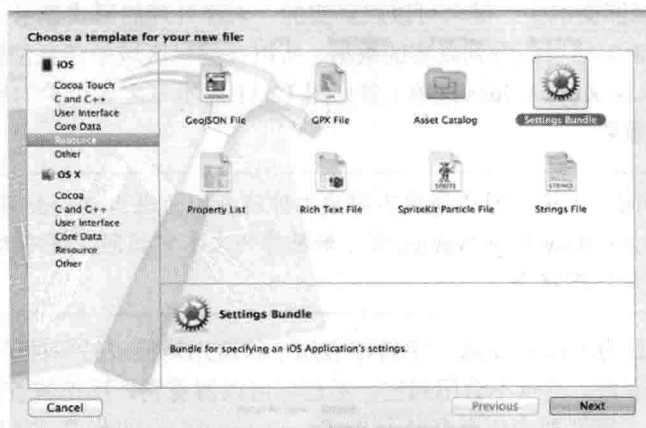


图 12-9 在 Xcode 中创建设置捆绑包

现在你应该能在项目窗口中看到一个新条目，名为设置.bundle。展开设置.bundle，应该能看到一个名为 en.lproj 的文件夹（其中还有个名为 Root.strings 的文件）以及一个名为 Root.plist 的图标。第 22 章介绍本地化应用时会讨论 en.lproj。现在主要介绍 Root.plist。

## 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（参见图 12-10）。此编辑器与 /Developer/Applications/Utilities 中 Property List Editor 应用的功能相同。

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
▶ Preference Items	Array	(4 items)
Strings Filename	String	Root

图 12-10 属性列表编辑器窗格中的 Root.plist。如果你的编辑窗格看起来不太一样，不要惊慌，只需要按住 control 键单击编辑窗格，从弹出的上下文菜单中选中 Show Raw Keys/Values

注意 plist 中各项的组织结构。属性列表在本质上就是字典，存储条目的类型和值，要通过键来检索它们，就跟使用 NSDictionary 一样。

属性列表可以包含几种不同类型的节点。Boolean、Data、Date、Number 和 String 节点类型可以保存数据，也可以通过多种方式来处理整个节点集合。除了能保存字典的 Dictionary 节点外，还有 Array 节点。它会存储一个含有其他节点的有序列表，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。

---

**注意** 虽然在 NSDictionary 中可以使用大多数对象作为键，但属性列表 Dictionary 节点中的键必须为字符串类型。但你可以选择任意节点类型作为该键的值。

---

创建设置属性列表时，必须遵循特定的格式。所幸，当在项目中添加设置捆绑包时，应用会创建一个符合这种格式的属性列表，名为 Root.plist。下面就来介绍它。

在 Root.plist 编辑器窗格中，键名可以以真实的、未经处理的形式显示，也可以以更易于阅读的形式显示。我们希望尽可能看到真实的数据，所以在编辑区域中任意位置右击鼠标，在弹出的菜单中选择 Show Raw Keys/Values 选项（参见图 12-11）。本章之后讨论的所有键都使用真实的键名，所以这一步很重要。

---

**警告** 在写作本书期间，不论是因为编辑不同的文件还是因为退出 Xcode 而离开属性列表，都会取消选中 Show Raw Keys/Values 项。如果你的文本突然间变得不太一样，再次检查一下该项是否处于选中状态。

---

字典中有一项条目为 StringsTable。字符串表用于将应用转换成另一种语言。第 22 章介绍本地化时，将讨论字符串表。本章不会用到它，不过你可以留着它，反正留着也没什么坏处。

除了 StringsTable，属性列表还有个名为 PreferenceSpecifiers 的节点，它是一个数组。这个数组节点用于保存一组 Dictionary 节点，每个 Dictionary 节点都代表用户可修改的一个偏好设置项

或用户可以访问的一个子视图。

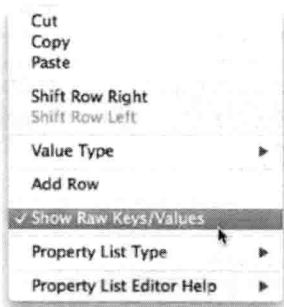


图 12-11 按住 control 键，在属性列表编辑窗格中的任意位置右击，确保选中了 Show Raw Keys/Values 选项。这将保证在属性列表编辑器中使用真实键名，从而使你的编辑工作更为精确

你会注意到 Xcode 的模板中提供了 4 个节点（参见图 12-12）。这些节点与我们实际的偏好设置没有任何联系，所以分别单击 Item 1、Item 2 和 Item 3，按 delete 键依次将它们删除，只留下 Item 0。

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
PreferenceSpecifiers	Array	(4 items)
▶ Item 0 (Group - Group)	Dictionary	(2 items)
▶ Item 1 (Text Field - Name)	Dictionary	(8 items)
▶ Item 2 (Toggle Switch - Enabled)	Dictionary	(4 items)
▶ Item 3 (Slider)	Dictionary	(7 items)

图 12-12 编辑器窗格中的 Root.plist，这次展开了 PreferenceSpecifiers

**注意** 要在属性列表中选择一项最好是单击 Key 列的一端或另一端，否则容易打开 Key 列的下拉菜单。

单击 Item 0，但不要展开它。你只需要在 Xcode 的属性列表中按下 return 键添加新行。当前的选择状态（包括选中了哪行以及该行是否展开）决定了新行的插入位置。如果选择了一个没有展开的数组或者字典，按下 return 键将会在该行后面添加一个同级节点。也就是说，将会添加一个与当前所选项同级的新节点。如果此时按下了 return 键（现在不要这么做），你将得到一个名为 Item 1 的新行，紧跟 Item 0。图 12-13 展示了按下 return 键创建新行的例子。注意，可以通过下拉菜单指定该项显示的偏好设置标识符的类型（稍后将会详细介绍）。

展开 Item 0 看看其中包含的内容（参见图 12-14）。现在可以在编辑器中为选中的条目添加子节点，这时按下 return（现在不要按）就会在 Item 0 中创建一个新的子节点。

Item 0 下面有一个 Type 键，并且 PreferenceSpecifiers 数组中的每一个属性列表节点都必须有一个包含此键的条目。它通常是第二个条目，但顺序在字典中并不重要，因此 Type 键不必一定

在第二位。Type 键可以让设置应用知道与此条目关联的是哪种类型的数据。

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
▼ PreferenceSpecifiers	Array	(2 items)
▶ Item 0 (Group - Group)	Dictionary	(2 items)
▶ Item 1 (Text Field - )	Dictionary	(3 items)
String	String	Root
Group		
Multi Value		
Slider		
✓ Text Field		
Title		
Toggle Switch		

图 12-13 选中 Item 0、按下 return 键可创建同级新行。注意弹出的下拉菜单，可以通过它指定该项显示的偏好设置标识符的类型

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
▼ PreferenceSpecifiers	Array	(1 item)
▼ Item 0 (Group - Group)	Dictionary	(2 items)
Title	String	Group
Type	String	PSGroupSpecifier

图 12-14 展开 Item 0，显示键为 Type 和 Title 的两行，Item 0 代表 Group 组

在 Item 0 中，此 Type 键的值 PSGroupSpecifier 用来说明该条目是一个新分组的开始。其后的每个条目都将是此分组的一部分，直到下一个 Type 对应的值为 PSGroupSpecifier 的条目为止。

从图 12-3 中可以看出，设置应用在一个分组的表视图中显示应用设置信息。PreferenceSpecifiers 数组中的 Item 0 在设置捆绑包属性列表中应始终为 PSGroupSpecifier 类型，这样设置便会在一个新的分组中开始，这很重要，因为每个设置表中需要至少一个分组。

Item 0 中剩下的一个条目包含一个名为 Title 的键，它用于在这个组上设置一个可选题。

现在仔细观察 Item 0 行本身，可以看到它实际显示为 Item 0 (Group-Group)。圆括号里的值代表 Type 项的值（第一个 Group）和 Title 项的值（第二个 Group）。这是 Xcode 提供的一种便捷方式，有助于直观地观察设置捆绑包的内容。

如图 12-3 所示，我们将第一个组命名为 General Info。双击 Title 旁边的值，将它从 Group 改为 General Info（参见图 12-15）。键入这个新的标题后，你可能会注意到 Item 0 发生了些细微的变化。它现在显示为 Item 0 (Group-General Info)，反映了新标题的值。

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
▼ PreferenceSpecifiers	Array	(1 item)
▼ Item 0 (Group - Group)	Dictionary	(2 items)
Title	String	General Info
Type	String	PSGroupSpecifier

图 12-15 我们将 Item 0 组的标题从 Group 改为 General Info



### 3. 添加文本框设置

现在,需要在此数组中添加另一个条目,它表示第一个实际的偏好设置字段。我们将从一个简单的文本框开始。

如果在编辑窗格中单击了 PreferenceSpecifiers (不要这么做,继续往下读),按下 return 键添加一个子项,那么新行将会插在列表的开头,这不是我们想要的,我们希望在数组末尾添加一行。

要添加该行,首先点击 Item 0 左边展开的三角形,把它关闭,然后选择 Item 0 按下 return 键,此时将会在当前行下方添加一个新的同级行(参见图 12-16)。同样,添加了新条目后,将会出现一个下拉菜单,显示了默认值 Text Field。

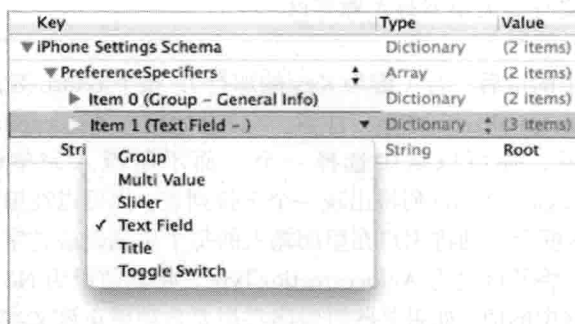


图 12-16 添加一个与 Item 0 同级的新行

点击下拉菜单外部使其消失,然后点击 Item 1 旁边的展开三角形展开它。可以看到它包含了一个值为 PSTextFieldSpecifier 的 Type 行,用于告诉设置应用我们希望用户在一个文本框中编辑这项设置。此外它还包含了键为 Title 和 Key 的两个空行(参见图 12-17)。

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
▼ PreferenceSpecifiers	Array	(2 items)
▶ Item 0 (Group - General Info)	Dictionary	(2 items)
▼ Item 1 (Text Field -)	Dictionary	(5 items)
Type	String	PSTextFieldSpecifier
Title	String	Commanding Officer
Key	String	officer

图 12-17 我们的文本框条目,展开后,其中显示了 Type、Title 和 Key

选择 Title 行,双击 Value 列的空白部分,键入 commanding Officer 来设置该 Title 的值。该文本将会出现在设置应用中。

现在对 Key 行进行同样的设置(这不是印刷错误,确实是将该键设置为了 Key),将其值设为 officer(注意第一个字母小写)。回想一下,我们提到过,用户默认设置的工作方式与 Dictionary 相似。这个新条目告诉设置应用,在存储此文本框中输入的值时使用什么键。

还记得我们讲过的关于 NSUserDefaults 的内容吗?它允许用户使用键保存值,这与 NSDictionary 类似。设置应用将对替你保存的每个偏好设置进行同样的操作。如果你为它提供了

一个键值 `foo`，则稍后可以在应用中请求 `foo` 值，它会返回用户为该偏好设置输入的值。稍后，我们将使用这个键值从应用中的用户默认设置获取此设置。

**注意** `Title` 的值为 `commanding Officer`，而 `Key` 的值为 `officer`。这种大小写差异将会经常出现，并且这里我们为显示标题使用了两个单词，而键只使用了一个。`Title` 是在屏幕上显示的内容，所以用大写字母 `C` 和 `O` 比较合适。而 `Key` 是一个文本字符串，用于从用户默认设置中获取偏好设置，所以所有字母采用小写形式比较合适。是否可以将 `Title` 的内容全部小写，而将 `Key` 的内容全部大写？当然可以。只要保存和检索时使用相同的大小写形式，为偏好设置键指定什么大小写形式都可以。

现在，选择 `Item 1` 中的最后一行（键为 `Key` 的那行），按下 `return` 键，在 `Item 1` 字典中添加另一项，将其键设为 `AutocapitalizationType`。注意，当你开始键入 `AutocapitalizationType` 时，Xcode 会向你显示一些匹配项，你可以从中选择一个，而不用键入完整的名字。当你完成了 `AutocapitalizationType` 输入后，`Value` 列将出现一个下拉列表，你可以在里面选择有用的选项。请选择 `Words`，它表示文本框会自动将用户在里面输入每个单词改成首字母大写。

创建最后一个新行，将其键设为 `AutocorrectionType`，将其值设为 `No`，它告诉设置应用不要自动更正输入到该文本框中的值。如果某些时候确实想要自动更正该文本框的值，则需将其值更改为 `Yes`。同样，输入 `AutocorrectionType` 后，Xcode 会为你提供匹配项列表，并在下拉列表中显示有效的选项。

完成这些操作后，属性列表应该如图 12-18 所示。

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
▼ PreferenceSpecifiers	Array	(2 items)
▶ Item 0 (Group - General Info)	Dictionary	(2 items)
▼ Item 1 (Text Field -	Dictionary	(5 items)
Type	String	PSTextFieldSpecifier
Title	String	Commanding Officer
Key	String	officer
AutocapitalizationType	String	Words
AutocorrectionType	String	No
StringsTable	String	Root

图 12-18 完成后的 `Root.plist` 中指定的文本框属性

#### 4. 添加应用图标

在体验新设置之前，我们向项目中添加一个应用图标。前面已经这么做过了。

保存 `Root.plist` 属性文件。然后在项目导航栏中选中 `Images.xcassets`，并选择里面的 `AppIcon` 项。你会看到那里有一组通过拖动来放置图标的方框。

在 `Finder` 中找到源代码归档文件并打开 `12-Bridge Control` 文件夹。将文件 `Icon.png` 拖到项目的配置编辑器中，将它拖动到左侧的应用图标拖动目标上。

这就做好了。现在从 `Product` 菜单中选择 `Run` 编译并运行应用。由于我们尚未给应用构建任

何 GUI，你看到的是个很粗糙的启动画面，只有一个暗灰色的背景和在一角出现的一个小小的蓝色信息图标。按下主屏幕按钮，单击设置应用图标，应该能够看到一个我们应用的条目，它用的是刚才添加的应用图标（参见图 12-2）。单击 Bridge Control 会显示一个简单的设置视图，其中包含一个文本框，如图 12-19 所示。

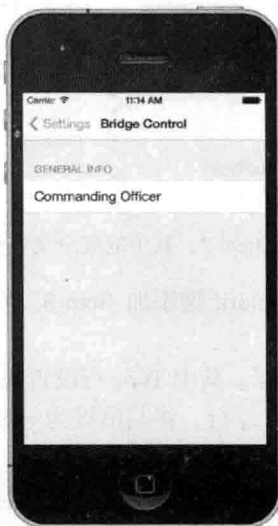


图 12-19 设置应用中添加了组和文本框的根视图

退出模拟器，返回 Xcode。虽然我们的工作还没有完成，但你应该发现为应用添加偏好设置很容易。现在添加根设置视图的其他字段。我们添加的第一项是用于输入用户验证码的安全文本框。

### 5. 添加安全文本框设置

单击 Root.plist，返回设置标识符（不要忘了选中 Show Raw Keys/Values，有可能 Xcode 重置过这一项）。收起 Item 0 和 Item 1。然后选择 Item 1，按 `command+C` 将其复制到剪贴板，再按 `command+V` 将其粘贴回原来的位置，这将创建一个与 Item 1 相同的新条目 Item 2。展开 Item 2，将 Title 改为 Authorization Code，将 Key 改为 `authorizationCode`。（记住标题是用来显示在屏幕上的，键是用来存储值的。）

接下来，向 Item 2 中添加一个子条目。记住，条目的顺序无关紧要，将新条目放置在 Key 条目下方即可。只需要点选键值为 Key / `authorizationCode` 的那行，按下 `return` 键就可以了。

将这个新条目的 Key 设为 `IsSecure`（注意开头的大写字母 I），Xcode 会自动将其 Type 改为 Boolean。现在将其 Value 从 NO 改为 YES。它会告诉设置应用，该框应该是一个隐藏用户输入文字的密码框，而不是一个普通的文本框。最后将 `AutocapitalizationType` 的值更改为 `None`。完成后的 Item 2 如图 12-20 所示。

### 6. 添加多值字段

我们将添加的下一个条目是一个多值字段。这种字段类型会自动生成带有展开指示器的行。

单击它将切换到另一个表，你可以在它的多行数据中选择某一项。

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
▼ PreferenceSpecifiers	Array	(3 items)
▶ Item 0 (Group - General Info)	Dictionary	(2 items)
▶ Item 1 (Text Field - Commanding	Dictionary	(5 items)
▼ Item 2 (Text Field - Authorization	Dictionary	(6 items)
Type	String	PSTextFieldSpecifier
Title	String	Authorization Code
Key	String	authorizationCode
IsSecure	Boolean	YES
AutocapitalizationType	String	None
AutocorrectionType	String	No

图 12-20 完成后的 Item 2，其中的安全文本框用于接收验证码

折叠 Item 2 并选中该行，按下 return 键添加 Item 3。在 Key 字段的弹出菜单中选择 Multi Value，单击展开三角形展开 Item 3。

展开了的 Item 3 已经包含一些行了。其中 Type 行设置成了 PSMultiValueSpecifier。找到 Title 行并将其值设置为 Rank。然后找到 Key 行，将其值设为 rank。下一部分会有点麻烦，所以在操作之前先讨论一下。

我们将向 Item 3 中添加另外两个子条目，但它们的节点类型是 Array，而不是 String。

□ Titles 数组，用于保存可供用户选择的一组值。

□ Values 数组，用于保存用户默认设置中存储的一组值。

Values 列表中的第一项与 Titles 数组中的第一项对应。因此，如果用户选择第一项，设置应用实际保存的是 Values 数组中的第一个值。这种 Titles/Values 对非常方便，它能为用户提供易于理解的文本，而实际上却保存其他内容，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，然后复制粘贴并更改副本的键，这样就会得到具有相同内容但保存在不同键下的两个数组。实际操作时，我们会采用这种方法。

选择 Item 3（保留它的展开状态），然后按下 return 键添加一个新的子条目。你将再次发现，Xcode 知道我们正在编辑的文件类型，而且似乎也预料到了我们想做的事情，因为这个新子行的 Key 值已经设置成了 Titles，而它本身也已被配置为一个 Array。这正是我们希望的！展开这个 Titles 行，按下 return 键添加一个子节点，重复 5 次，这样你总共就拥有了 6 个子节点。将这 6 个节点全部设置为 String 类型，并将其值分别设置为：Ensign、Lieutenant、Lieutenant commander、Captain 和 Commodore。

创建完所有 6 个节点并输入相应值后，折叠 Titles 行并将其选中，按 command+C 进行复制，然后按 command+V 进行粘贴。这将创建一个新项，其键为 Titles - 2。双击 Titles - 2，将它改为 Values。

到此，我们基本完成了多值字段的操作。唯一缺少的是 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）。我们创建项目时，Xcode 自动添加了 DefaultValue 行，将其值设为 Ensign 即可。图 12-21 显示了最终完成的 Item 3。

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
▼ PreferenceSpecifiers	Array	(4 items)
▶ Item 0 (Group - General Info)	Dictionary	(2 items)
▶ Item 1 (Text Field - Commanding	Dictionary	(5 items)
▶ Item 2 (Text Field - Authorization	Dictionary	(6 items)
▼ Item 3 (Multi Value - Rank)	Dictionary	(6 items)
▼ Titles	Array	(6 items)
Item 0	String	Ensign
Item 1	String	Lieutenant
Item 2	String	Lieutenant Commander
Item 3	String	Commander
Item 4	String	Captain
Item 5	String	Commodore
▼ Values	Array	(6 items)
Item 0	String	Ensign
Item 1	String	Lieutenant
Item 2	String	Lieutenant Commander
Item 3	String	Commander
Item 4	String	Captain
Item 5	String	Commodore
Type	String	PSMultiValueSpecifier
Title	String	Rank
Key	String	rank
DefaultValue	String	Ensign

图 12-21 最终完成的 Item 3，其中的多值字段用于从 5 个可能值中选择一个

检验一下我们的工作。保存属性列表，编译并再次运行应用。应用启动时，按下主屏幕按钮，并启动设置应用。选择 Bridge Control 条目之后，根级视图上应该显示 3 个字段（参见图 12-22）。尝试使用一下新建的多值字段，然后学习下一项任务。

### 7. 添加拨动开关设置

需要从用户处获取的下一项内容是一个 Boolean 值，该值表示拨动开关是否打开。为了获取偏好设置中的 Boolean 值，我们将向 PreferenceSpecifiers 数组添加另一个类型为 PSToggleSwitchSpecifier 的条目，告诉设置应用使用 UISwitch。

如果 Item 3 当前处于展开状态，那么将它收起。单击，将其选中。按下 return 键创建 Item 4。在下拉菜单中选择 Toggle Switch，单击展开三角形展开 Item 4，它已经创建了一个默认的子行，键和值分别设置为 Type 和 PSToggleSwitchSpecifier。将空 Title 行的值设置为 Warp Drive，Key 行的值设置为 warp。

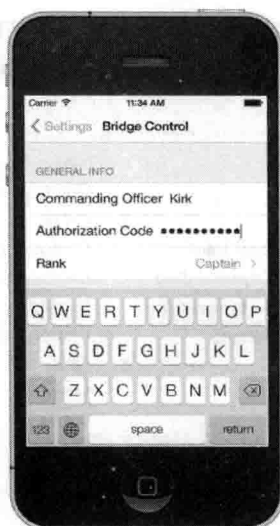


图 12-22 现在有三个字段了，看起来不错

这个字典中还有一个必填项：默认值。和 Multi Value 一样，Xcode 已经为我们创建了 DefaultValue 行，我们将 DefaultValue 的值设置为 YES，默认开启曲速引擎。图 12-23 显示了设置好的 Item 4。

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
▼ PreferenceSpecifiers	Array	(5 items)
▶ Item 0 (Group - General Info)	Dictionary	(2 items)
▶ Item 1 (Text Field - Commanding	Dictionary	(5 items)
▶ Item 2 (Text Field - Authorization	Dictionary	(6 items)
▶ Item 3 (Multi Value - Rank)	Dictionary	(6 items)
▼ Item 4 (Toggle Switch - Warp	Dictionary	(4 items)
Type	String	PSToggleSwitchSpecifier
Title	String	Warp Drive
Key	String	warp
DefaultValue	Boolean	YES

图 12-23 设置好的 Item 4，这个开关用于打开和关闭曲速引擎

### 8. 添加滑动条设置

接下来我们将添加的项是一个滑动条。在设置应用中，滑动条两端可以各有一个小图像，但它不能有标签。我们将滑动条放置在一个带有自己标题的组中，以使用户了解滑动条的作用。

首先将 Item 4 收起，然后单击 Item 4 并按下 return 键添加一个新行。使用弹出菜单将新项改为 Group，然后点击旁边的展开三角形展开它。可以看到，Type 已经设置成了 PSGroupSpecifier。这是告诉设置应用，在这个位置开始一个新组。双击 Title 行中的值，将该值改为 Warp Factor。

收起 Item 5 并将其选中，按下 return 添加一个新的同级行。通过弹出菜单将新项更改为 Slider，它指示设置应用应该使用 UISlider 从用户处获取此信息。展开 Item 6，将另一行的键和值分别设

为 Key 和 warpFactor, 这样, 设置应用就能知道存储该值时使用什么键。

我们允许用户输入 1~10 的一个值, 并将默认值设置为 warp 5。滑动条需要有一个最小值、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) 的形式保存到属性列表中。幸好, Xcode 替这些值创建了相应的行, 我们只需将 DefaultValue 行的值设置为 5, MinimumValue 行的值设置为 1, MaximumValue 行的值设置为 10。

如果想要测试一下该滑动条, 那就赶快动手吧。测试之后我们还要立即回来, 对滑动条进行一些定制工作。

大家应该注意到了, 滑动条可以有图片。滑动条两端允许分别放置一个 21 像素×21 像素的小图像。我们将提供一些图标来说明向左滑动会降低速度, 向右滑动会提高速度。

### 9. 为设置捆绑包添加图标

在本书附带的项目归档文件 12 - Bridge Control 文件夹中有两个图标, 分别为 rabbit.png 和 turtle.png。我们需要将这两个图标添加到设置捆绑包中。设置应用需要使用这两个图标, 因此不能仅将它们放在 Bridge Control 文件夹中, 还需要将它们放在设置捆绑包中, 这样设置应用才能使用它们。

为此, 在项目导航面板中找到 Settings.bundle, 我们需要在 Finder 中打开此捆绑包。按住 control 单击 Settings.bundle 图标, 此时会弹出上下文菜单, 选择 Show in Finder (如图 12-24 所示) 在 Finder 中显示该捆绑包。

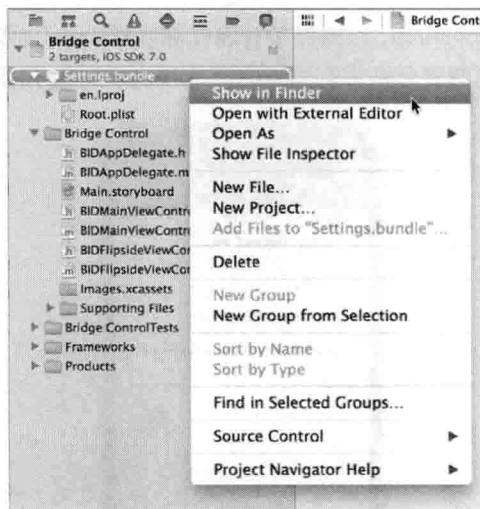


图 12-24 Settings.bundle 的右键菜单

记住, 在 Finder 中, 捆绑包看起来像文件, 但实际上它们是文件夹, 可以按住 control 键单击捆绑包的图标, 然后在出现的上下文菜单中选择 Show Package Contents 来访问捆绑包的内容。这将打开一个新的 Finder 窗口来显示设置捆绑包, 你应该能够在 Xcode 的 Settings.bundle 中看到那两个图标。将图标文件 rabbit.png 和 turtle.png 从 12 - Bridge Control 文件夹中复制到 Finder 窗

口的 Settings.bundle 包内容中。

在 Finder 中将此窗口保留为打开状态,因为稍后我们还要将另外一个文件复制到这里。现在,返回 Xcode,告诉滑动条使用这两个图像。

返回 Root.plist,在 Item 6 下添加两个子行,将它们的键和值分别设置为 MinimumValueImage 和 turtle、MaximumValueImage 和 rabbit。完成后的 Item 6 如图 12-25 所示。

Key	Type	Value
iPhone Settings Schema	Dictionary	(2 items)
Preference Items	Array	(7 items)
Item 0 (Group - General Info)	Dictionary	(2 items)
Item 1 (Text Field -	Dictionary	(5 items)
Item 2 (Text Field -	Dictionary	(6 items)
Item 3 (Multi Value - Rank)	Dictionary	(6 items)
Item 4 (Toggle Switch - Warp	Dictionary	(4 items)
Item 5 (Group - Warp Factor)	Dictionary	(2 items)
Item 6 (Slider)	Dictionary	(7 items)
Type	String	Slider
Identifier	String	warpFactor
Default Value	Number	5
Minimum Value	Number	1
Maximum Value	Number	10
Min Value Image Filename	String	turtle
Max Value Image Filename	String	rabbit

图 12-25 设置好的 Item 6,其中的滑动条分别用海龟和兔子来表示慢和快

保存属性列表,编译并运行应用,以确保所有属性都能够生效。如果所有属性都能正常设置了,导航到设置应用,应该能找到两端分别带有酣睡的海龟和快乐的兔子图标 of 的滑动条(参见图 12-26)。



图 12-26 我们已经拥有文本框、多值字段、拨动开关和滑动条。应用就要做好了



### 10. 添加子设置视图

接下来将添加另一个偏好设置标识符,告诉设置应用,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展开指示器的行,单击该展开指示器会调出一个全新的偏好设置视图。

由于我们不希望新的偏好设置与滑动条分到同一组,因此在添加此节点之前,复制 Item 0 中的组标识符,并将它粘贴到 PreferenceSpecifiers 数组的末尾,为子设置视图创建一个新组。

在 Root.plist 中,收起所有展开的项,然后单击 Item 0 将其选中,并按 command+C 将其复制到剪贴板。接下来,选择 Item 6,然后按 command+V 粘贴新项 Item 7。展开 Item 7,双击键 Title 旁边的 Value 列,将它由 General Info 改为 Additional Info。

现在,再次收起 Item 7。选择 Item 7,按下 return 键添加 Item 8,它将是实际的子视图。单击展开三角形将其展开。找到 Type 行,将它的值设置为 PSChildPaneSpecifier。然后将 Title 行的值设置为 More 设置。

我们需要向 Item 8 添加最后一行,它将告诉设置应用,为 More 设置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置为 File 和 More(参见图 12-27)。假定文件扩展名为.plist,并且.plist 不应包含在文件名中,否则设置应用将无法找到此属性列表文件。

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
▼ PreferenceSpecifiers	Array	(9 items)
▶ Item 0 (Group - General Info)	Dictionary	(2 items)
▶ Item 1 (Text Field -	Dictionary	(5 items)
▶ Item 2 (Text Field -	Dictionary	(6 items)
▶ Item 3 (Multi Value - Rank)	Dictionary	(6 items)
▶ Item 4 (Toggle Switch - Warp	Dictionary	(4 items)
▶ Item 5 (Group - Warp Factor)	Dictionary	(2 items)
▶ Item 6 (Slider)	Dictionary	(7 items)
▼ Item 7 (Group - Additional Info)	Dictionary	(2 items)
Title	String	Additional Info
Type	String	PSGroupSpecifier
▼ Item 8 (Child Pane - More	Dictionary	(3 items)
Type	String	PSChildPaneSpecifier
Title	String	More Settings
File	String	More

图 12-27 设置好了 Item 7 和 Item 8, 创建了新的 Additional Info 设置组, 将子窗格链接到 More.plist 文件

现在,我们需要向主偏好设置视图添加一个子视图。这项设置是在 More.plist 文件中指定的。我们需要将 More.plist 复制到设置捆绑包中。不能在 Xcode 中向设置捆绑包添加新文件,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捆绑包中。因此,必须创建一个新的属性列表,将它保存到其他某个地方,然后通过 Finder 将它拖入 Settings.bundle 窗口。

现在,你已经知道了所有能在设置捆绑包属性列表文件中使用的偏好设置字段类型。为了节约输入代码的时间,可以使用本书附带的项目归档文件 12 - Bridge Control 文件夹中的 More.plist,将它拖入到之前打开的 Settings.bundle 窗口。

---

**提示** 创建子设置视图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偏好设置标识符，将需要的所有偏好设置标识符添加到此新文件中。

---

现在我们已经完成设置捆绑包的相关操作。你可以编译、运行和测试设置应用。你应该能够进入该子视图并设置所有其他字段的值。可以动手试试，还可以随便更改属性列表。

---

**提示** 这里介绍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 iOS 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。可以在以下网页中找到该文档以及许多其他有用的参考文档：<http://developer.apple.com/library/ios/navigation/>。

---

在继续讨论之前，请在 Xcode 的项目导航栏中选中 `Images.xcasset`，把项目归档文件 `12 - Bridge Control` 文件夹中的 `rabbit.png` 和 `turtle.png` 复制到编辑区域的左侧。将这些图标作为新图像资源添加到项目中。我们将会应用中使用它们显示当前设置的值。

你可能注意到了，刚才添加的两个图标与之前添加到设置捆绑包中的图标完全相同。这是为什么呢？记住，iOS 上的应用不能从其他应用的沙盒中读取文件。设置捆绑包并不是我们应用沙盒的一部分，而是设置应用沙盒的一部分。我们还要在自己的应用中使用这些图标，因此需要单独将它们添加到 `Bridge Control` 文件夹中，这样它们便会复制到我们的应用沙盒中。

### 12.2.3 读取应用中的设置

现在问题解决了一半。用户能够使用设置应用来声明他们的偏好设置，但我们如何在应用中获取用户的偏好设置呢？非常简单。

#### 1. 获取用户设置

我们将使用 `NSUserDefaults` 类访问用户设置。`NSUserDefaults` 作为单例类，意味着应用中只能有一个 `NSUserDefaults` 实例在运行。为了访问这个实例，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，我们可以像使用 `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`。如果我们要以标量（如整型、浮点型或布尔型）的形式获取该值，可以使用 `intValueForKey:`、`floatForKey:`、`boolForKey:` 等其他方法。

创建应用的属性列表时，请在属性表文件中添加一个 `PreferenceSpecifiers` 数组，其中一些标识符在设置应用里用于创建组。另一些标识符用于创建用户进行交互时使用的界面对象。这些才是我们真正感兴趣的标识符，因为它们保存了实际的设置数据的键。绑定到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，滑动条键的值为 `warpfactor`，`Authorization`

Code 字段的键为 password...我们通过这些键获取用户设置。

我们将使用一些预编译程序的#define 语句来表示每个键的文本值，而不是直接在方法中使用它们。这样我们在代码中可以使用这些临时常量以体改内置文本，避免出现输入错误导致的一些问题。因为我们之后要在其他类中使用它们，所以要在头文件中对它们进行设定。那么请打开 BIDMainViewController.h 并在文件顶端添加这些粗体显示的代码：

这些常量是我们在属性列表文件中为不同的偏好设置字段使用的键值。现在我们已经拥有了显示设置的地方，接下来使用一组标签快速设置一下主视图。进入界面构建器之前，先为我们需要的所有标签创建输出接口。单击 BIDMainViewController.m，并进行如下更改：

```
#import "BIDFlipsideViewController.h"

#define kOfficerKey           @"officer"
#define kAuthorizationCodeKey @"authorizationCode"
#define kRankKey             @"rank"
#define kWarpDriveKey        @"warp"
#define kWarpFactorKey       @"warpFactor"
#define kFavoriteTeaKey      @"favoriteTea"
#define kFavoriteCaptainKey  @"favoriteCaptain"
#define kFavoriteGadgetKey   @"favoriteGadget"
#define kFavoriteAlienKey    @"favoriteAlien"
#import "BIDMainViewController.h"

@interface BIDMainViewController ()

@property (weak, nonatomic) IBOutlet UILabel *officerLabel;
@property (weak, nonatomic) IBOutlet UILabel *authorizationCodeLabel;
@property (weak, nonatomic) IBOutlet UILabel *rankLabel;
@property (weak, nonatomic) IBOutlet UILabel *warpDriveLabel;
@property (weak, nonatomic) IBOutlet UILabel *warpFactorLabel;
@property (weak, nonatomic) IBOutlet UILabel *favoriteTeaLabel;
@property (weak, nonatomic) IBOutlet UILabel *favoriteCaptainLabel;
@property (weak, nonatomic) IBOutlet UILabel *favoriteGadgetLabel;
@property (weak, nonatomic) IBOutlet UILabel *favoriteAlienLabel;

@end
```

代码中没有涉及什么新知识。之后我们声明了九个属性变量，它们都是标签，而且都带有能在界面构建器中进行关联的 IBOutlet 关键字。

保存修改。现在我们已经声明了各个输出接口，下面转到故事板文件来创建 GUI。

## 2. 创建主视图

双击 MainStoryboard.storyboard，在界面构建器中编辑它。视图出现后，你会发现主视图在左，翻转视图在右，由转场连接。注意，主视图的背景为暗灰色。我们将它改为白色。

单击属于 Main View Controller 的 View 图标，打开属性查看器。使用 Background 取色板中的颜色将背景改为白色。请注意如果点击取色板右侧将会显示弹出菜单，也可以从该菜单中选择 White Color。

现在将向 View 中添加一些标签，如图 12-28 所示。我们需要使用 18 个标签。其中有一半位

于屏幕左侧、粗体、右对齐；另一半位于屏幕右侧，用于显示从用户默认设置获取的实际值，并使输出接口指向这些标签。

依照图 12-28 创建此视图。你创建的视图外观无需与图 12-29 完全一样，但视图上必须为我们声明的每个输出接口分别提供对应的标签。这些你自己可以做到，完成后返回，然后进行其他操作。提一下，这里所有的标签使用的字体都是 15 点 System Font（或者 System Font Bold），你可以根据自己的喜好随便使用其他字体。

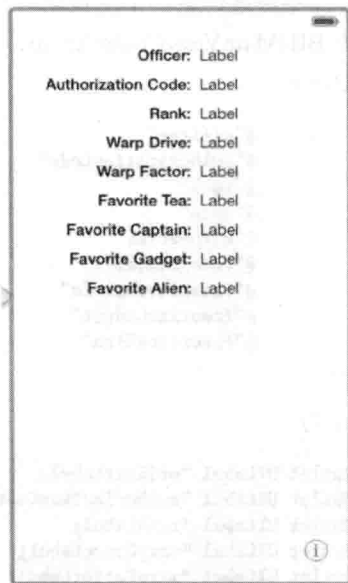


图 12-28 界面构建器中的 View 窗口，显示了我们添加的 18 个标签

接下来要做的是按住 control 键并从 Main View Controller 图标拖动到每个用于显示设置值的标签上。你一共需要按住 control 键并拖动 9 次，将每个标签都设成指向不同的输出接口。将所有 9 个输出接口与标签连接好后，保存修改。

### 3. 更新主视图控制器

在 Xcode 中，选择 BIDMainViewController.m，在类的@implementation 位置添加以下代码：

```
- (void)refreshFields {
   NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    self.officerLabel.text = [defaults objectForKey:kOfficerKey];
    self.authorizationCodeLabel.text = [defaults
                                         objectForKey:kAuthorizationCodeKey];
    self.rankLabel.text = [defaults objectForKey:kRankKey];
    self.warpDriveLabel.text = [defaults boolForKey:kWarpDriveKey
                                         ? @"Engaged" : @"Disabled"];
    self.warpFactorLabel.text = [[defaults objectForKey:kWarpFactorKey]
                                stringValue];
    self.favoriteTeaLabel.text = [defaults objectForKey:kFavoriteTeaKey];
}
```

```

self.favoriteCaptainLabel.text = [defaults
                                objectForKey:kFavoriteCaptainKey];
self.favoriteGadgetLabel.text = [defaults objectForKey:kFavoriteGadgetKey];
self.favoriteAlienLabel.text = [defaults objectForKey:kFavoriteAlienKey];
}

- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    [self refreshFields];
}
.
.
.

```

上面的代码中需要解释的内容不是很多。新方法 `refreshFields` 做了两件事情。首先它获取了标准用户默认设置，其次使用我们输入到 `plist` 文件中的相同键值，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`，我们在返回的对象上调用 `stringValue`。所有其他偏好设置都是字符串，它们都以 `NSString` 对象的形式从用户默认设置返回。但是，滑动条存储的偏好设置以 `NSNumber` 的形式返回，因此我们调用该对象的 `stringValue` 方法来获取它存储的值的字符串表示。

然后，覆盖了父类的 `viewDidAppear:` 方法，它会调用 `refreshFields` 方法。

在获取翻转视图控制器被解除的通知之后，再次调用了 `refreshFields` 方法。由于翻转视图会被当成模态视图来处理（因为其父视图是模态视图），因此 `BIDMainViewController` 的 `viewDidAppear:` 方法不会在翻转视图被解除时调用。幸好我们选择的 `Utility Application` 模板提供了一个委托方法，可以实现此目的。将下面的代码添加到已有的 `flipsideViewControllerDidFinish:` 方法中：

```

- (void)flipsideViewControllerDidFinish:
    (BIDFlipsideViewController *)controller
{
    [self refreshFields];
    [self dismissViewControllerAnimated:YES completion:nil];
}

```

有了这段代码，在视图加载和翻转视图被换出时，我们显示的字段就会设置为对应的偏好设置值。

### 12.2.4 在应用中修改默认设置

现在我们已经构建好主视图了，下一步构建翻转视图。如图 12-29 所示，翻转视图显示的是我们的曲速引擎开关，以及曲速层级滑动条。我们会用跟设置应用相同的控件：一个开关和一个滑动条。除了声明输出接口，我们还会声明一个 `refreshFields` 方法（与 `BIDMainViewController` 中的做法一样）以及两个在用户点击控件时触发的方法。



图 12-29 在界面构建器中设计翻转视图

选定 BIDFlipsideViewController.h, 做如下修改:

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

@class BIDFlipsideViewController;

@protocol BIDFlipsideViewControllerDelegate
- (void)flipsideViewControllerDidFinish:
    (BIDFlipsideViewController *)controller;
@end

@interface BIDFlipsideViewController : UIViewController

@property (weak, nonatomic) id <BIDFlipsideViewControllerDelegate> delegate;
@property (weak, nonatomic) IBOutlet UISwitch *engineSwitch;
@property (weak, nonatomic) IBOutlet UISlider *warpFactorSlider;

- (void)refreshFields;
- (IBAction)engineSwitchTapped;
- (IBAction)warpSliderTouched;
- (IBAction)done:(id)sender;

@end
```

**注意** 不必担心此处出现的其他代码。跟我们前面见过的一样, Utility Application 模板会将 BIDMainViewController 作为 BIDFlipsideViewController 的一个委托。这里出现的没有在其他文件模板中用过的代码实现了委托关系。

现在, 保存更改。选中 MainStoryboard.storyboard, 在界面构建器中编辑 GUI, 这次我们主要看一下文件大纲中的 Flipside View Controller Scene。首先, 按下 option 键, 点击展开三角来展开 Flipside View Controller 及其子项, 然后双击标题栏的标题并将它改为 Warp Settings。

接着,选择 Flipside View Controller Scene 中的 View,打开属性查看器。先在 Background 弹出菜单中将背景色改为 Light Gray Color。默认的翻转视图背景颜色太黑,不利于黑色文本的显示,但对白色文本来说又太亮。

然后,从库中拖出两个 Label,并将它们放置在 View 窗口中。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 12-29 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 Flipside View Controller 图标拖到新开关,将其连接到 engineSwitch 输出接口。然后按住 control 键再将开关拖回 Flipside View Controller 图标,将其连接到 engineSwitchTapped 操作方法。

从库中拖出一个 Slider,将它放置在 Warp Factor: 标签的下方。调整滑动条的大小,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 Flipside View Controller 图标拖到滑动条,将其连接到 warpFactorSlider 输出接口。然后按住 control 键并从滑动条拖到 Flipside View Controller,选择 warpSliderTouched 操作。

如果未选中滑动条,则单击滑动条将其选中,然后调出属性查看器。将 Minimum、Maximum 和 Current 分别设置为 1.00、10.00 和 5.00。然后,分别选择乌龟和兔子作为 Min Image 和 Max Image 的图标。如果它们没有出现在弹出菜单中,请确认是否将图片拖入了 Images.xcassets 资源目录中。

接下来完成翻转视图控制器。选择 BIDFlipsideViewController.m,在文件的开头添加如下信息:

```
#import "BIDMainViewController.h"
```

并在类实现中做如下修改:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后 (通常是从 nib 文件加载), 做一些额外的设置。
    [self refreshFields];
}

- (void)refreshFields {
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    self.engineSwitch.on = [defaults boolForKey:kWarpDriveKey];
    self.warpFactorSlider.value = [defaults floatForKey:kWarpFactorKey];
}

- (IBAction)engineSwitchTapped {
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    [defaults setBool:self.engineSwitch.on forKey:kWarpDriveKey];
}

- (IBAction)warpSliderTouched {
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    [defaults setFloat:self.warpFactorSlider.value forKey:kWarpFactorKey];
}
```



```

}

```

我们在 `viewDidLoad` 方法中添加了对 `refreshFields` 方法的调用。这个方法中的 3 行代码获取了对标准用户默认设置的引用, 并通过开关和滑动条的输出接口将用户默认设置中存储的值显示出来。我们还实现了 `engineSwitchTapped` 和 `warpSliderTouched` 操作方法, 这样用户更改的控件中的值就能回填到用户默认设置中。

现在这个应用应该能够运行了。你可以翻到背面, 修改那里的值, 然后在翻回正面时, 在主视图中看到修改后的值。

### 12.2.5 注册默认值

我们已经创建了一个设置捆绑包, 其中包括一些值的默认设置, 这样设置应用能够直接访问我们应用的偏好设置。我们还对应用进行了其他设置, 使其能够访问相同的信息, 并为用户提供了一个 GUI 来查看和修改偏好设置。不过, 百密一疏啊: 我们的应用完全不知道设置捆绑包中指定的那些默认值。你可以在 iOS 模拟器中或设备上先删掉 Bridge Control 应用 (这样就删除了已保存的该应用的偏好设置), 然后再从 Xcode 运行一次, 验证一下。第一次运行时, 这个应用会将所有设置的值都显示为空值。即使是在设置捆绑包中为 warp drive 指定的默认值也不会显示出来。如果你转到设置应用, 就能看到默认值, 但除非你在设置应用中实际修改了这些值, 否则我们的 Bridge Control 应用还是不会显示它们。

这些设置之所以没显示出来, 是因为这个应用完全不知道它包含的设置捆绑包。所以, 当尝试从 `NSUserDefaults` 中为 `warpFactor` 读取值而未在该键下找到任何已保存的值时, 它就不会显示任何值。幸好, `NSUserDefaults` 包含了一个名为 `registerDefaults:` 的方法, 如果我们尝试查找一个尚未设置的键 / 值, 但该键至少应该有默认值, 就可以通过该方法指定默认值。为了使该设置在整个应用中都有效, 最好在应用启动时就立即调用它。选择 `BIDAppDelegate.m` 文件, 在该文件顶部引入以下头文件, 以便获取之前定义的键名:

```

#import "BIDMainViewController.h"

```

然后对 `application:didFinishLaunchingWithOptions:` 方法作如下修改:

```

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

```

```

    // 程序启动之后的一些自定义设置。

```

```

    NSDictionary *defaults = @{kWarpDriveKey : @YES,
                               kWarpFactorKey : @5,
                               kFavoriteAlienKey : @"Vulcan"};

```

```

    [[NSUserDefaults standardUserDefaults] registerDefaults:defaults];
    return YES;
}

```



首先我们创建了一个包含 3 个键/值对的字典，其中每一项都对应一个在设置中提供的需要默认值的键。我们使用与之前定义的键相同的键名，以避免输入错误的键名。注意，除了初始化字典的 `@{}` 语法，我们还在创建包含布尔值 YES 和整数 5 用到的 `NSNumber` 实例时，用到了新的 `@<数值>` 语法。

我们将整个字典传递给标准的 `NSUserDefaults` 实例的 `registerDefaults:` 方法。由此，只要没有在我们的应用或设置应用中设置不同的值，`NSUserDefaults` 就会提供这里指定的值。

完了！现在你可以编译并运行该应用了。运行结果如图 12-6 所示，当然，你的应用将会显示你在设置应用中设置的值。非常简单，不是吗？

### 12.2.6 保证设置有效

现在你可以运行这个应用、查看设置信息并按下主屏幕按钮打开设置应用来修改一些值了。然后再按一次主屏幕按钮，重新启动这个应用，结果可能令你大吃一惊。当你回到该应用时，你会看到设置并未改变！它们依然跟之前一样，显示的是以前的值。

原因在于：iOS 中，当应用正在运行时点击主屏幕按钮并不会退出该应用，而是由操作系统在后台将其暂停，这样它就随时都能快速启动了。这点在用户做应用间切换时非常有用，因为重新唤醒一个暂停的应用比从头启动一个应用省很多时间。不过，在这个例子中，我们需要做一点额外的工作，这样当应用被唤醒时，它能够很快清醒，重新加载用户偏好设置并重新显示它们的值。

第 15 章会介绍更多有关后台应用的知识，但这里会简单介绍一下如何在切换回它时，让应用知道自己已经起死回生。要实现这个功能，我们将注册所有控制器类，以便接收从暂停执行状态唤醒的应用发送出来的通知。

**通知**是对象之间进行通信的轻量级机制。任何对象都能定义一个或多个发送到应用通知中心的通知。**通知中心**是一个单例对象，作用是在对象之间传送通知。通知通常是某些事件发生时发送出的说明，而发布通知信息的对象会在它们的文档中包含一系列通知。`UIApplication` 类会发送大量的通知（你可以在 Xcode 的文档阅读器中找到这些内容，在 `UIApplication` 页面的底部）。大多数通知的用途从命名就能看出来，但当你发现有的通知的用途不甚明了时，可以在文档中找到更详细的信息。

应用需要在应用回到前台时刷新一下它显示的内容，所以我们需要关注一下名为 `UIApplicationWillEnterForegroundNotification` 的通知。编写 `viewWillAppear:` 方法时，我们会订阅该通知，并告诉通知中心在通知出现时调用该方法。将这个方法添加到 `BIDMainViewController.m` 和 `BIDFlipsideViewController.m` 中：

```
- (void)applicationWillEnterForeground:(NSNotification *)notification {
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    [defaults synchronize];
    [self refreshFields];
}
```

这个方法本身很简单。首先，它会获得一个到标准的用户默认值对象的引用，然后调用该对象的 `synchronize` 方法，强制用户默认值系统保存尚未保存的修改，然后从存储中重新加载所有

未修改的偏好设置。实际上,这是在强制它重新读取已保存的偏好设置,从而获得设置应用中所作的修改。然后它会调用 `refreshFields` 方法——每个类都用它来更新显示的内容。

现在,我们需要实现 `BIDMainViewController.m` 和 `BIDFlipsideViewController.m` 中的 `viewWillAppear:` 方法,使每个控制器都订阅我们关注的通知。下面是在两个类中做的修改:

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];

    UIApplication *app = [UIApplication sharedApplication];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(applicationWillEnterForeground:)
        name:UIApplicationWillEnterForegroundNotification
        object:app];
}
```

我们先获得一个对应用实例的引用,通过默认的 `NSNotificationCenter` 实例和名为 `addObserver:selector:name:object:` 的方法订阅 `UIApplicationWillEnterForegroundNotification`,然后将如下内容传给该方法。

- ❑ 对于 `observer`,我们给它传递 `self`,也就是我们的控制器类(每个控制器类,因为它们都会包含这段代码)需要得到通知。
- ❑ 对于 `selector`,我们会给刚写的 `applicationWillEnterForeground:` 方法传一个选择器,告诉通知中心在该通知发出时调用该方法。
- ❑ 第三个参数 `UIApplicationWillEnterForegroundNotification` 是我们要接收的通知的名称。
- ❑ 最后一个参数 `app` 是我们关心的获得通知的来源对象。如果给最后一个参数传递的是 `nil`,只要有方法发出 `UIApplicationWillEnterForegroundNotification`,我们就会得到通知。

它会负责更新显示的内容,但我们还需要考虑当用户在我们的应用中操作控件时,那些值要怎么放到用户默认值中。要保证在控制权转移到其他应用前将它们保存到存储中。最简单的办法就是只要设置被修改了,就调用 `synchronize` 方法。我们可以在 `BIDFlipsideViewController.m` 中的每个新操作方法中添加一行:

```
- (IBAction)engineSwitchTapped {
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    [defaults setBool:self.engineSwitch.on forKey:kWarpDriveKey];
    [defaults synchronize];
}
- (IBAction)warpSliderTouched {
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    [defaults setFloat:self.warpFactorSlider.value forKey:kWarpFactorKey];
    [defaults synchronize];
}
```

**注意** 调用 `synchronize` 方法的开销可能会很大,因为要比较内存中和存储中的所有用户偏好默认值。如果你一次对大量用户默认值做了修改而要保证所有内容都保持同步,最好尽量减少对 `synchronize` 的调用,这样整个比较就不会一次又一次地重复进行。不过,像这里这样,在响应每个用户操作时调用它一次,不会造成任何显著的性能问题。

还有一件事需要注意，它能让整个工作过程更清晰。你知道，当不会再到属性时，必须将属性设置为 nil 来清理内存，其他清理工作也是一样。通知系统是另外一处需要清理的地方。你可以告诉默认的 `NSNotificationCenter` 不想收到其他任何通知。在这个例子中，我们将每个视图控制器都注册了，以便在 `viewWillAppear:` 方法中监听该通知，所以我们应该在对应的 `viewDidDisappear:` 方法中撤销注册。因此，在 `BIDMainViewController.m` 和 `BIDFlipsideViewController.m` 中，将下面几行放到 `viewDidDisappear:` 方法的顶部：

```
- (void)viewDidDisappear:(BOOL)animated {
    [super viewDidDisappear:(BOOL)animated];
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```

注意，也可以使用 `removeObserver:name:object:` 方法来撤销对特定通知的订阅，将前面注册 observer 时用的参数值传递给该方法就行。但前面这种方法是确保通知中心彻底忘记我们的 observer 的更简便方法，不管注册它是为了接收多少种通知。

做完以上这些工作，你就可以构建和运行该应用并查看在应用和设置应用之间切换时会发生什么了。当切换回你的应用时，你在设置应用中所做的修改应该会立即体现出来。

## 12.3 小结

此时此刻，你应该已经对设置应用和用户默认值机制有了深刻的了解。你知道如何为应用添加捆绑包，如何为应用偏好设置构建结构化视图。你还学习了如何通过 `NSUserDefaults` 读写偏好设置，以及如何让用户在应用内修改偏好设置。你甚至还了解了如何在 Xcode 中使用新项目模板。总之，你已经掌握了应用偏好设置的所有知识。

下一章会继续介绍如何在应用退出时持久化你的应用数据。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。尽管我们的几个应用已经从应用包中读取了数据，但是还没有一个应用能将数据持久地保存起来。持久存储是一种非易失性存储，在重新启动计算机或设备时也不会丢失数据。到目前为止，除了设置应用外（参见第 12 章），其他示例应用都没有存储数据，也没有使用易失性（非持久化）存储。启动其中任意一个示例应用，显示的数据都与第一次启动时完全相同。

不持久保存数据目前来看对我们还无所谓。但是现实中的应用是需要持久存储数据的，只有这样，用户才能在更改数据之后再次启动程序时，看得到自己的更改。

可以使用多种机制将数据持久存储在 iOS 设备上。用 Mac OS X 平台的 Cocoa 写过程序的朋友可能已经接触过这方面的部分或者全部技术了。

本章将介绍 4 种将数据持久存储到 iOS 文件系统的机制：

- 属性列表；
- 对象归档；
- iOS 的嵌入式关系数据库（SQLite3）；
- 苹果公司提供的持久化工具 Core Data。

我们将使用这 4 种机制编写一些示例应用。

---

**注意** 在 iOS 开发中，持久化数据的方法并不限于属性列表、对象归档、SQLite3 和 Core Data。它们只是 4 种最常用且最简单的方法。你可以使用传统的 C 语言 I/O 调用（比如 `fopen()`）读取和写入数据，也可以使用 Cocoa 的底层文件管理工具。只不过以上两种方法都需要多写很多代码，并且也没有必要这么做。当然，如果确实需要的话，选择它们是没问题的。

---

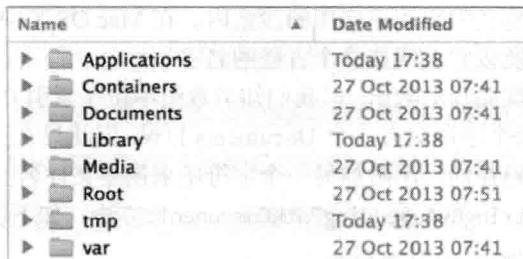
## 13.1 应用的沙盒

本章介绍的 4 种数据持久化机制都涉及一个共同要素，即应用的 /Documents 文件夹。每个应用都有自己的 /Documents 文件夹，且仅能读写各自的 /Documents 目录中的内容。

为便于讲解，我们先来看一下 iPhone 模拟器使用的文件夹布局，从而了解 iOS 中应用是如

何组织的。为此,需要看主目录中所包含的 Library (资源库) 目录。在 Mac OS X 10.6 及之前的版本中,这没有任何问题,但是从 10.7 开始,苹果默认隐藏了 Library 文件夹,需要几个额外的操作才能找到这个目录。打开 Finder 窗口,找到主目录,如果可以看见 Library 文件夹,那太好了。如果没有,按住 option 键,并选择前往→资源库。如果不按下 option 键就看不到 Library 选项。

在 Library 文件夹中,向下找到 Application Support/iPhone Simulator/。在该目录中可以看到一些子目录,分别对应当前 Xcode 所支持的 iOS 版本。例如,你可能看到名为 7.0,或者 iOS 7 的其他版本,或许还能看到 iOS 6。展开代表最新 iOS 版本的目录,应该可以看到几个子文件夹,其中包含一个名为 Applications 的文件夹(参见图 13-1)。



Name	Date Modified
Applications	Today 17:38
Containers	27 Oct 2013 07:41
Documents	27 Oct 2013 07:41
Library	Today 17:38
Media	27 Oct 2013 07:41
Root	27 Oct 2013 07:51
tmp	Today 17:38
var	27 Oct 2013 07:41

图 13-1 显示 Applications 文件夹的 Library/Application Support/iPhone Simulator/7.0.3/ 目录的布局

**注意** 如果安装了多个版本的 SDK,你会在 iPhone Simulator 目录中看到其他一些文件夹,其名称表示 iOS 版本号,这很正常。

虽然这是模拟器的目录,但实际设备上的文件结构与此相似。显而易见,Applications 文件夹就是 iOS 存储其应用的文件夹。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 自动生成的全局唯一标识符。每个文件夹都包含一个应用及其支持文件夹。

打开其中一个 Applications 的子目录,应该会看到一些比较熟悉的内容。在这里,可以找到你之前构建并在模拟器中运行的 iOS 应用以及 3 个支持文件夹。

- Documents: 应用将数据存储在 Documents 中,但基于 UserDefaults 的首选项设置除外。
- Library: 基于 UserDefaults 的首选项设置存储在 Library/Preferences 文件夹中。
- tmp: tmp 目录供应用存储临时文件。当 iOS 设备执行同步时,iTunes 不会备份 tmp 中的文件,但在不需要这些文件时,应用要负责删除 tmp 中的文件,以免占用文件系统的空间。

### 13.1.1 获取 Documents 目录

既然我们的应用位于一个名称看上去随机的文件夹中,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 C 函数 `NSSearchPathForDirectoriesInDomain()` 来查找各种目录。它是基础函数，因此可以与基于 Mac OS X 平台的 Cocoa 共享。它的很多可用选项都是专门为 OS X 设计的，在 iOS 上不会返回任何值。原因在于 iOS 上并不存在这些位置（如“下载”文件夹），或者应用因为 iOS 的沙盒机制而没有访问该位置的权限。

下面是检索 Documents 目录路径的一些代码：

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
    NSUserDomainMask, YES);
NSString *documentsDirectory = paths[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在应用的沙盒内。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，所以它才叫这么个古怪的名字。

虽然返回的是一个匹配路径的数组，但我们知道数组中位于索引 0 处的一定是 Documents 目录，为什么呢？我们知道每个应用只有一个 Documents 目录，因此只有一个目录符合指定的条件。

可以在刚刚检索到的路径的结尾附加另一个字符串来创建文件名。为此要使用专为该目的设计的 `NSString` 方法，即 `stringByAppendingPathComponent:` 方法，如下所示：

```
NSString *filename = [documentsDirectory
    stringByAppendingPathComponent:@"theFile.txt"];
```

完成此调用之后，`filename` 就是指向应用的 Documents 目录中 `theFile.txt` 文件的完整路径。然后，我们就可以使用 `filename` 来创建、读取和写入文件了。

### 13.1.2 获取tmp目录

获取对应用临时目录的引用比获取对 Documents 目录的引用容易。名为 `NSTemporaryDirectory()` 的 Foundation 函数将返回一个字符串，该字符串包含到应用临时目录的完整路径。若要创建一个将会存储在临时目录中的文件，首先要找到该临时目录：

```
NSString *tempPath = NSTemporaryDirectory();
```

然后，在路径的结尾附上文件名就可以创建指向该目录下文件的路径，例如：

```
NSString *tempFile = [tempPath
    stringByAppendingPathComponent:@"tempFile.txt"];
```

## 13.2 文件保存方案

本章将介绍 4 种实现数据持久化的方法，这 4 种方法都使用 iOS 的文件系统。使用 SQLite3 将创建一个 SQLite3 数据库文件，并让 SQLite3 去存储和检索数据。Core Data 则以其最简单的形式帮助开发者完成所有文件系统的管理工作。使用属性列表和归档则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。

### 13.2.1 单文件持久化

把数据保存在一个文件中是最简单的方法，且对于许多应用，这也是完全可以接受的方法。首先，创建一个根对象，通常是 `NSArray` 或 `NSDictionary`（使用归档文件的情况下根对象可以基于某个自定义类）。接下来，使用所有需要保存的程序数据填充根对象。真正保存的时候，代码会将该根对象的全部内容重新写入单个文件。应用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。这就是本章将要使用的方法。

使用单文件的缺点是必须将全部数据加载到内存中，并且不管更改多少也必须将所有数据全部重新写入文件系统。如果应用管理的数据不超过几兆字节，此方法可能非常好，而且它非常简单，一点也不麻烦。

### 13.2.2 多文件持久化

使用多个文件是另一种实现持久化的方法。比如，电子邮件应用可能会将每封邮件都单独存储在一个文件中。

这种方法有明显的优势，比如应用可以只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。用户当前未查看的任何数据都可以从内存中删除，下次需要时再从文件系统重新加载即可。

多文件持久化的缺点是它大大增加了应用的复杂性。到目前为止，我们还是坚持使用单文件持久化。

接下来，我们将分别详细介绍属性列表、对象归档、SQLite3 和 Core Data 这几种持久化方法。我们在探讨每种方法时都会用相应的方法构建一个应用，将数据保存到设备的文件系统。首先从属性列表开始。

## 13.3 属性列表

我们的许多示例应用都使用了属性列表，比如说使用属性列表来指定应用首选项。属性列表非常方便，因为可以使用 Xcode 或 Property List Editor 应用手动编辑它们，并且只要字典或数组包含特定可序列化对象，就可以将 `NSDictionary` 和 `NSArray` 实例写入属性列表或者从属性列表创建它们。

### 13.3.1 属性列表序列化

序列化对象（serialized object）是指可以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的对象。虽然说任何对象都可被序列化，但只有某些对象才能被放置到某个集合类中（如 `NSDictionary` 或 `NSArray` 中），然后才使用该集合类的 `writeToFile:atomically:` 方法或 `writeToURL:atomically:` 方法将它们存储到属性列表中。可以按照该方法序列化下面的 Objective-C 类：

□ `NSArray`



- NSMutableArray
- NSDictionary
- NSMutableDictionary
- NSData
- NSMutableData
- NSString
- NSMutableString
- NSNumber
- NSDate

如果只使用这些对象构建数据模型，就可以使用属性列表来方便地保存和加载数据。

如果你打算使用属性列表持久保存应用数据，则可以使用 `NSArray` 或 `NSDictionary`。假设放到 `NSArray` 或 `NSDictionary` 中的所有对象都是前面列出的可序列化对象，则可以通过对字典或数组实例调用 `writeToFile:atomically:` 方法来写入属性列表，如下所示：

```
[myArray writeToFile:@"some/file/location/output.plist" atomically:YES];
```

---

**注意** 稍微解释一下，这里的 `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，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，因为如果应用在保存期间崩溃，则现有文件（如果有）不会被破坏。尽管这增加了一点开销，但是多数情况下还是值得的。

---

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，另外也不能使用没有在前面的可序列化对象列表中指定的 Cocoa Touch 交付的其他类，这意味着无法直接使用 `NSURL`、`UIImage` 和 `UIColor` 等类。

且不说序列化问题，将这些模型对象保存到属性列表中还意味着你无法轻松创建派生的或需要计算的属性（例如，等于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。但多数情况下，如果创建了专用的模型类，则应用更容易维护。

但是，在复杂的应用中，简单的属性列表仍然非常有用。它们是将静态数据包含在应用中的最佳方法。例如，当应用包含一个选取器时，创建一个属性列表文件并将其放在项目的 `Resources` 文件夹中，就是将项目列表包含到选取器中的最佳方法，这样能把项目列表编译到应用中。

下面让我们构建一个使用属性列表存储数据的简单应用。

### 13.3.2 Persistence应用的第一个版本

本节将构建一个程序，可让你在 4 个文本框中输入数据，应用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这些数据（参见图 13-2）。



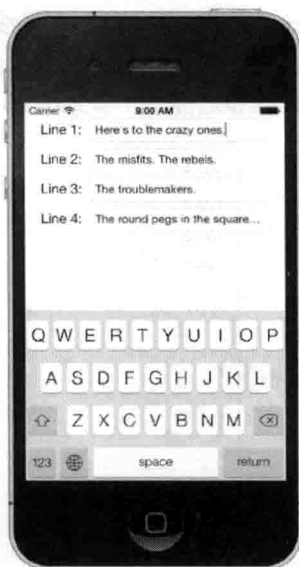


图 13-2 Persistence 应用

**注意** 对于本章的应用，我们不会花费时间设置所有用户界面细节（前几章中也是这么做的）。例如，按 **return** 键既不会让键盘消失，也不会进入下一个文本框。如果你希望向应用添加这类功能，这非常好，我们鼓励你自己动手实现。

### 1. 创建 Persistence 项目

在 Xcode 中，使用 Single View Application 模板创建一个新项目，命名为 Persistence。项目包含了构建应用所需的所有文件，以便可以直奔主题。

稍后将构建一个具有 4 个文本框的视图，但构建前需先创建所需的输出接口。打开 Classes 文件夹中的 BIDViewController.m 文件，并进行以下更改：

```
#import "BIDViewController.h"

@interface BIDViewController ()

@property (strong, nonatomic) IBOutletCollection(UITextField) NSArray *lineFields;

@end
```

接下来选中 BIDViewController.xib，将其打开以编辑 GUI。

### 2. 设计 Persistence 应用的视图

启动界面构建器之后，你会在编辑面板中看到 View Controller Scene。从库中拖出一个 Text Field，然后根据顶部和右侧的蓝色引导线放置它。打开属性检查器，取消选中标签为 Clear When Editing Begins 的复选框。

现在，向窗口中拖入一个 Label，使用左侧的蓝色引导线将其置于文本框的左边，并且使用水平居中引导线将该标签与文本框对齐。双击标签将其值改为“Line 1:”。最后，使用文本框左侧的调节手柄调整该字段的大小，使其靠近标签（参见图 13-3）。

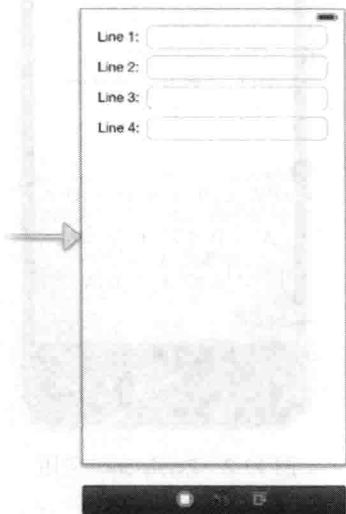


图 13-3 设计 Persistence 应用的视图

接着，同时选中标签和文本框，按下 option 键，往下拖动以复制一份副本。使用蓝色引导线将其放置在适当位置。然后同时选中标签和文本框，再次按下 option 向下拖动，现在就有了 4 个标签和旁边对应的 4 个文本框。依次双击复制的标签，将它们的名字分别改为“Line 2:”、“Line 3:”、“Line 4:”，参见图 13-3。

添加 4 个文本框和标签之后，按住鼠标右键从 View Controller 图标拖到每个文本框中。将所有文本框连接到 lineFields 输出接口集合，确保连接顺序为从顶部到底部。最后保存对 Main.storyboard 所作的修改。

### 3. 编辑 Persistence 类

在项目导航面板中，单击 BIDViewController.m 并将以下代码添加到 @implementation 位置：

```
@implementation BIDViewController
```

```
- (NSString *)dataFilePath
{
    NSArray *paths = NSSearchPathForDirectoriesInDomains(
        NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];
    return [documentsDirectory stringByAppendingPathComponent:@"data.plist"];
}
.
.
.
```

我们添加的第一个方法是 `dataFilePath`，它会查找 Documents 目录并在其后附加数据文件的文件名，这样就得到了数据文件的完整路径。需要加载或保存数据的任何代码都可以调用该方法。

再往下一点，找到 `viewDidLoad` 方法，然后向其中添加一些代码，并在后面添加一个名为 `applicationWillResignActive:` 的方法，用于接收通知，代码如下所示：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后（通常是从 nib 文件加载），做一些额外的设置
    NSString *filePath = [self dataFilePath];
    if ([[NSFileManager defaultManager] fileExistsAtPath:filePath]) {
        NSArray *array = [[NSArray alloc] initWithContentsOfFile:filePath];
        for (int i = 0; i < 4; i++) {
            UITextField *theField = self.lineFields[i];
            theField.text = array[i];
        }
    }

    UIApplication *app = [UIApplication sharedApplication];
    [[NSNotificationCenter defaultCenter]
     addObserver:self
     selector:@selector(applicationWillResignActive:)
     name:UIApplicationWillResignActiveNotification
     object:app];
}

- (void)applicationWillResignActive:(NSNotification *)notification
{
    NSString *filePath = [self dataFilePath];
    NSArray *array = [self.lineFields valueForKey:@"text"];
    [array writeToFile:filePath atomically:YES];
}
```

在 `viewDidLoad` 方法中，我们做了几件事情。第一件事是检查数据文件是否存在，如果不存在就不加载它了；如果存在，就用该文件的内容实例化数组，然后将数组中的对象复制到 4 个文本框。由于数组是按顺序排列的列表，因此只要根据保存顺序（之后会看到保存相关代码）来复制数组，就一定可以确保相应的字段获得正确的值。

```
NSString *filePath = [self dataFilePath];
if ([[NSFileManager defaultManager] fileExistsAtPath:filePath]) {
    NSArray *array = [[NSArray alloc] initWithContentsOfFile:filePath];
    for (int i = 0; i < 4; i++) {
        UITextField *theField = self.lineFields[i];
        theField.text = array[i];
    }
}
```

从属性列表中加载数据之后，我们获得了对应用实例的引用，并使用该引用订阅 `UIApplicationWillResignActiveNotification`，使用默认的 `NSNotificationCenter` 实例以及一个名为 `addObserver:selector:name:object:` 的方法。我们将 `self` 传递给第一个参数，这样 `BIDViewController` 就会作为观察者接收通知。对于第二个参数，传入 `applicationWillResignActive:` 方法，告知通知

中心在发布该通知后调用这个方法。第三个参数 `UIApplicationWillResignActiveNotification` 是我们希望接收的通知名称，它是由 `UIApplication` 类定义的字符串常量。最后一个参数 `app` 是要从中获取通知的对象。

```
UIApplication *app = [UIApplication sharedApplication];
[[NSNotificationCenter defaultCenter]
 addObserver:self
 selector:@selector(applicationWillResignActive:)
 name:UIApplicationWillResignActiveNotification
 object:app];
```

最后一个新方法是 `applicationWillResignActive:`。请注意，它接受一个指向 `NSNotification` 的指针作为参数。还记得吗？第 12 章讲过这种模式。`applicationWillResignActive:` 是一个通知方法，所有通知都接受一个 `NSNotification` 实例作为参数。

应用应该在终止运行或者进入后台之前保存数据，所以我们需要使用名为 `UIApplicationWillResignActiveNotification` 的通知。这样，只要这个应用不再是当前正与用户进行交互的应用就会发布通知，包括用户按下 Home 键（主屏幕按钮），以及其他事件（比如来了个电话）导致应用进入后台运行。之前在 `viewDidLoad` 方法中，我们使用了通知中心来订阅这个具体的通知。发生该通知时就会调用此方法：

```
- (void)applicationWillResignActive:(NSNotification *)notification
{
    NSString *filePath = [self dataFilePath];
    NSArray *array = [self.lineFields valueForKey:@"text"];
    [array writeToFile:filePath atomically:YES];
}
```

此方法非常简短，不过实际上调用了很多方法。我们通过调用 `lineFields` 数组中每个文本框的 `text` 方法构建一个字符串数组。我们利用了一个便捷的方法，没有迭代数组中的文本框获取每个 `text` 值，并添加到新的数组中；而只是对数组调用了 `valueForKey:` 方法，并传递 `"text"` 作为参数。`NSArray` 类的 `valueForKey:` 方法为我们实现了迭代获取实例变量的 `text` 值，返回包含这些值的数组。然后将该数组的内容写入一个属性列表文件。使用属性列表保存数据就这么简单。

还好，不是太难吧？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框中。接下来，应用将在终止（退出运行或进入后台）时通知我们。当应用终止时，我们收集 4 个文本框中的值，将它们保存到可变数组中，并将该可变数组写入属性列表。

为什么不编译和运行一下应用呢？现在构建并在模拟器中运行这个应用。应用在模拟器上运行之后，你应该能够在这 4 个文本框中键入文本。在其中键入某些内容后，按下主屏幕按钮（模拟器窗口底部包含一个圆角矩形的圆形按钮）。按主屏幕按钮非常重要，如果只是退出模拟器（不按主屏幕按钮），等同于强制退出应用，视图控制器不会收到应用终止的通知，并且绝对不会保存你的数据。按下主屏幕按钮之后，就可以退出模拟器，或者是在 Xcode 中结束应用，然后再次运行。应用再次运行时，之前的文本数据仍然存在。

**注意** 需要知道按下主屏幕按钮通常不会退出应用，至少不会立即退出。应用将进入后台状态，并准备在用户切换回来之后迅速重新激活。第 15 章将深入介绍这些状态和它们对运行和退出应用的影响。与此同时，如果希望确认数据确实已保存，可以完全退出 iPhone 模拟器，然后从 Xcode 重新启动应用。退出模拟器基本上相当于重新启动 iPhone，所以当它再次启动时，你相当于经历了一次手机重启。

属性列表序列化非常实用，也非常好用，但它有一点限制，即只能将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。

## 13.4 对模型对象进行归档

在第 9 章的最后，我们在构建 Presidents 数据模型对象之后，给出了一个使用 NSCoder 加载归档数据的示例。在 Cocoa 世界中，归档（archiving）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的任何模型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）或都是遵循 NSCodering 协议的某个类的实例，你就可以对整个对象进行完全的归档。由于大多数支持存储数据的 Foundation 和 Cocoa Touch 类都遵循 NSCodering 协议（不过有一些例外，如 UIImage），因此对于大多数类来说，归档相对而言比较容易实现。

尽管对归档的使用没有严格要求，但还有一个协议应该与 NSCodering 一起实现，即 NSCopying 协议。后者允许复制对象，这使你在使用数据模型对象时具备了较大的灵活性。例如，在第 9 章的 Presidents 应用中，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，而是可以生成 president 对象的副本，并将更改存储在副本中。如果用户按下 Save，我们只需复制更改后的版本来替换原来的版本。

### 13.4.1 遵循 NSCodering 协议

NSCodering 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，使用方式与第 12 章的 UserDefaults 非常相似。也可以使用 KVC（Key-Value Coding，键值编码）对对象和原生数据类型（如 int 和 float）进行编码和解码。

对某个对象进行编码的方法可能看起来如下所示：

```
- (void)encodeWithCoder:(NSCoder *)encoder
{
    [encoder encodeObject:foo forKey:kFooKey];
    [encoder encodeObject:bar forKey:kBarKey];
    [encoder encodeInt:someInt forKey:kSomeIntKey];
    [encoder encodeFloat:someFloat forKey:kSomeFloatKey]
}
```

若要在我们的项目中支持归档，必须使用正确的编码方法将所有实例变量编码成 `encoder`。如果要子类化某个也遵循 `NSCoding` 的类，还需要确保对超类调用 `encodeWithCoder:` 方法，你的方法将如下所示：

```
- (void)encodeWithCoder:(NSCoder *)encoder
{
    [super encodeWithCoder:encoder];
    [encoder encodeObject:foo forKey:kFooKey];
    [encoder encodeObject:bar forKey:kBarKey];
    [encoder encodeInt:someInt forKey:kSomeIntKey];
    [encoder encodeFloat:someFloat forKey:kSomeFloatKey];
}
```

我们还需要实现一个通过 `NSCoder` 解码的对象初始化方法，恢复我们之前归档的对象。实现 `initWithCoder:` 方法比实现 `encodeWithCoder:` 方法稍微复杂一些。如果直接对 `NSObject` 进行子类化，或者对某些不遵循 `NSCoding` 的其他类进行子类化，则你的方法将看起来如下所示：

```
- (id)initWithCoder:(NSCoder *)decoder
{
    if (self = [super init]) {
        foo = [decoder decodeObjectForKey:kFooKey];
        bar = [decoder decodeObjectForKey:kBarKey];
        someInt = [decoder decodeIntForKey:kSomeIntKey];
        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 `NSCoder` 的实例中传递的值来设置其属性。当为某个具有超类且遵循 `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:`，像这样：

```
- (id)initWithCoder:(NSCoder *)decoder
{
    if (self = [super initWithCoder:decoder]) {
        foo = [decoder decodeObjectForKey:kFooKey];
        bar = [decoder decodeObjectForKey:kBarKey];
        someInt = [decoder decodeIntForKey:kSomeIntKey];
        someFloat = [decoder decodeFloatForKey:kAgeKey];
    }
    return self;
}
```

基本就这些。只要实现这两个方法，就可以对所有对象的属性进行编码和解码，然后便可以对对象进行归档，并且可以将其写入归档或者从归档中读取它们。

### 13.4.2 实现 `NSCopying` 协议

如前所述，遵循 `NSCopying` 对于任何数据模型对象来说都是非常好的事情。`NSCopying` 有一个 `copyWithZone:` 方法，可用来复制对象。实现 `NSCopying` 与实现 `initWithCoder:` 非常相似，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:

```
- (id)copyWithZone:(NSZone *)zone
{
    MyClass *copy = [[[self class] allocWithZone:zone] init];
    copy.foo = [self.foo copyWithZone:zone];
    copy.bar = [self.bar copyWithZone:zone];
    copy.someInt = self.someInt;
    copy.someFloat = self.someFloat;
    return copy;
}
```

13

**注意** 不要过于担心 NSZone 参数。它指向系统用于管理内存的 struct。只有在极少数情况下,开发人员才需要关注 zone 或者创建自己的 zone。而在目前,还没有使用多个 zone 的说法。对某个对象调用 copy 的方法与使用默认 zone 调用 copyWithZone:的方法完全相同,几乎始终能满足你的需求。事实上,现在的 iOS 上完全可以忽略 zone。NSCopying 会用到 zone 本质上是考虑向后兼容性所致。

### 13.4.3 对数据对象进行归档和取消归档

从遵循 NSCoder 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,像这样:

```
[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。

```
[archiver finishEncoding];
BOOL success = [data writeToFile:@"path/to/archive" atomically:YES];
```

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。

从归档重组对象的步骤类似。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/to/archive"];
NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc]
    initWithReadingWithData:data];
```



然后，使用之前对对象进行归档的同一个键从解压程序中读取对象：

```
self.object = [unarchiver decodeObjectForKey:@"keyValueString"];
```

最后，告知归档程序已经完成了该操作：

```
[unarchiver finishDecoding];
```

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 应用添加归档功能，以便于你理解其内部原理。操作几次之后，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。

### 13.4.4 归档应用

本节将改进 Persistence 应用，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，因此在继续下述步骤之前先为整个项目文件夹创建一个副本。

#### 1. 实现 BIDFourLines 类

准备好后，在 Xcode 中打开 Persistence 项目的副本。然后单击 Persistence 文件夹，并按 Command+N 或从 File 菜单中选择 New>New File...。出现新建文件向导后，选择 Cocoa Touch，然后选择 Objective-C class，单击 Next。将新类命名为 BIDFourLines，并在 Subclass of 控件中选择 NSObject，再次单击 Next。选择 Persistence 文件夹保存文件，单击 Create。该类将作为我们的数据模型，并且它将容纳当前存储在属性列表应用的字典中的数据。

单击 BIDFourLines.h，并进行以下更改：

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

```
@interface BIDFourLines : NSObject <NSCoding, NSCopying>
```

```
@property (copy, nonatomic) NSArray *lines;
```

```
@end
```

这是一个非常简单的数据模型类，它具有一个数组类型的属性，数组中包含 4 个字符串。注意，我们已经让该类遵循 NSCoder 和 NSCopying 协议了。现在切换到 BIDFourLines.m，并添加以下代码：

```
#import "BIDFourLines.h"
```

```
static NSString * const kLinesKey = @"kLinesKey";
```

```
@implementation BIDFourLines
```

```
#pragma mark - Coding
```

```
-(id)initWithCoder:(NSCoder *)aDecoder
```

```
{
```

```
    self = [super init];
```

```
    if (self) {
```

```
        self.lines = [aDecoder decodeObjectForKey:kLinesKey];
```



```

    }
    return self;
}

- (void)encodeWithCoder:(NSCoder *)aCoder;
{
    [aCoder encodeObject:self.lines forKey:kLinesKey];
}

#pragma mark - Copying

- (id)copyWithZone:(NSZone *)zone;
{
    BIDFourLines *copy = [[[self class] allocWithZone:zone] init];
    NSMutableArray *linesCopy = [NSMutableArray array];
    for (id line in self.lines) {
        [linesCopy addObject:[line copyWithZone:zone]];
    }
    copy.lines = linesCopy;
    return copy;
}

@end

```

我们刚才实现了遵循 NSCoding 和 NSCopying 所需的所有方法。在 encodeWithCoder: 中对 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 BIDFourLines 对象, 并将 4 个字符串复制到其中。看见了吗? 这一点儿也不难。复制代码时不要忘记做必要的修改。

## 2. 实现 BIDViewController 类

创建可归档的数据对象之后, 让我们使用它来持久化应用数据。单击 BIDViewController.m, 并进行以下更改:

```

#import "BIDViewController.h"
#import "BIDFourLines.h"

static NSString * const kRootKey = @"kRootKey";

@interface BIDViewController ()

@property (strong, nonatomic) IBOutletCollection(UITextField) NSArray *lineFields;

@end

@implementation BIDViewController

- (NSString *)dataFilePath
{
    NSArray *paths = NSSearchPathForDirectoriesInDomains(
        NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];
    return [documentsDirectory stringByAppendingPathComponent:@"data.plist"];
    return [documentsDirectory stringByAppendingPathComponent:@"data.archive"];
}

```

```

}

- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后 (通常是从 nib 文件加载), 做一些额外的设置
    NSString *filePath = [self dataFilePath];
    if ([[NSFileManager defaultManager] fileExistsAtPath:filePath]) {
        NSArray *array = [[NSArray alloc] initWithContentsOfFile:filePath];
        for (int i = 0; i < 4; i++) {
            UITextField *textField = self.lineFields[i];
            textField.text = array[i];
        }
        NSData *data = [[NSMutableData alloc]
                        initWithContentsOfFile:filePath];
        NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc]
                                         initWithReadingWithData:data];
        BIDFourLines *fourLines = [unarchiver decodeObjectForKey:kRootKey];
        [unarchiver finishDecoding];

        for (int i = 0; i < 4; i++) {
            UITextField *textField = self.lineFields[i];
            textField.text = fourLines.lines[i];
        }
    }
    UIApplication *app = [UIApplication sharedApplication];
    [[NSNotificationCenter defaultCenter]
     addObserver:self
     selector:@selector(applicationWillResignActive:)
     name:UIApplicationWillResignActiveNotification
     object:app];
}

- (void)applicationWillResignActive:(NSNotification *)notification
{
    NSString *filePath = [self dataFilePath];
    NSArray *array = [self.lineFields valueForKey:@"text"];
    [array writeToFile:filePath atomically:YES];

    BIDFourLines *fourLines = [[BIDFourLines alloc] init];
    fourLines.lines = [self.lineFields valueForKey:@"text"];
    NSMutableData *data = [[NSMutableData alloc] init];
    NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc]
                                 initWithWritingWithMutableData:data];
    [archiver encodeObject:fourLines forKey:kRootKey];
    [archiver finishEncoding];
    [data writeToFile:filePath atomically:YES];
}

@end

```

保存更改, 先试运行一下这个版本的 Persistence。

更改的东西不多。首先我们指定了一个新的文件名, 以避免应用加载旧的属性列表作为归档。

我们还定义了一个新的常量，作为编码和解码的键。然后重新编写加载和保存的代码，使用 `BIDFourLines` 保存数据，并且使用 `NSCoding` 的方法完成实际的加载和保存工作。GUI 与上一个版本完全相同。

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，答案非常简单：实际上没什么优势。但是回想第 9 章中的最后一个示例，我们在那个例子中允许用户编辑总统列表，每个总统有 4 个可编辑字段。要使用属性列表处理对总统列表的归档，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，以及将该数组写入属性列表文件。当然，这是假设我们只能使用可序列化的属性。否则，不做大量转换工作根本就不能使用属性列表序列化。

另一方面，如果我们拥有一个包含可归档对象的数组（像刚才构建的 `BIDFourLines` 类），则可以对数组实例本身进行归档来归档整个数组。对集合类（如 `NSArray`）进行归档时，也会归档其包含的所有对象。只要放入数组或字典中的对象遵循 `NSCoding`，你就可以归档数组或字典并还原它。这样，对其进行归档时，其中所有对象都将位于已还原的数组或字典中。

换句话说，该方法具有非常好的伸缩性（至少从代码量看是这样），因为无论添加多少对象，将这些对象写入磁盘的方式（假设使用单文件持久化）都完全相同。但使用属性列表，工作量会随着添加对象而增加。

## 13.5 使用 iOS 内嵌的 SQLite3

第三个持久化选项是 iOS 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。

考虑这样两种情况，假设应用需要计算其中所有对象的特殊字段的总和，或者需要只符合特定条件的对象的总和，SQLite3 可以不需要将所有对象加载到内存中就获取到这些信息。从 SQLite3 获取聚合比将所有对象加载到内存，然后计算它们值的总和要快几个数量级。作为一个功能比较完善的嵌入式数据库，SQLite3 还可以通过一些工具进一步提升速度（如创建表索引加快查询速度）。

---

**注意** 有关“SQL”和“SQLite”的发音，有两派意见。大多数官方文档将“SQL”读作“S-Q-L”，将“SQLite”读作“S-Q-L-Light”。但也有很多人分别将它们读作“Sequel”和“Sequel Light”。少数非主流还喜欢读作“Squeal”和“Squeal Light”。请选择你喜欢的叫法（如果选择“Squeal”，那么其他人可能会觉得你很古怪，请三思而行）。

---

SQLite3 使用 SQL（Structured Query Language，结构化查询语言）。SQL 是与关系数据库交互的标准语言。本书是全部采用 SQL 语法（实际上有几百个）以及 SQLite 本身编写的。因此，

如果你还不了解 SQL 并且想在应用中使用 SQLite3,则需要提前做点儿工作。我们将介绍如何在 iOS 应用中进行设置并与 SQLite 数据库交互,我们会在本章中展示一些基本语法。但是,要真正充分利用 SQLite3 需要进行更为深入的研究和探索。<http://www.sqlite.org/cintro.html> 上的 “An Introduction to the SQLite3 C/C++ Interface” 和 <http://www.sqlite.org/lang.html> 上的 “SQL As Understood by SQLite” 都是不错的起点。

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。这些方法差异很大,因而出现了在两者之间进行转换的各种技术以及很多库和工具。这些技术统称为 ORM (Object-Relational Mapping, 对象关系映射)。目前有多种 ORM 工具可用于 Cocoa Touch。实际上,我们将在下一节讨论苹果公司提供的 ORM 解决方案,即 Core Data。

在此之前,我们将重点介绍基础知识,包括设置 SQLite3、创建容纳数据的表以及利用应用中的数据库。很明显,现实世界中像我们的示例这么简单的应用,不值得去兴师动众地用 SQLite3。但是,正是它的简单性使它成为一个非常好的学习示例。

### 13.5.1 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是 `sqlite3_open()`,这样将打开一个现有数据库,如果指定位置上不存在数据库,则函数便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`,就表示数据库已成功打开。此处你应该记住,数据库文件的路径必须以 C 字符串(而非 `NSString`)的形式进行传递。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,有个 `NSString` 方法,该方法能从 `NSString` 实例生成 C 字符串:

```
const char *stringPath = [pathString UTF8String];
```

对 SQLite3 数据库执行完所有操作后,调用以下内容来关闭数据库:

```
sqlite3_close(database);
```

数据库将所有数据存储在表中。可以通过 SQL 的 `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);
```

---

**提示** 如果两个字符串之间除了空白(包括换行符)之外没有其他的分隔字符,那么这两个字符串会被连接为一个字符串。

---

如之前所做的一样,需要检查 `result` 是否等于 `SQLITE_OK` 以确保命令成功运行。如果命令未

成功运行, `errorMsg` 将对所发生的问题进行描述。

函数 `sqlite3_exec` 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, 必须首先向其输入 SQL 的 `SELECT` 命令来准备该语句:

```
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);
```

13

**注意** 所有接受字符串的 SQLite3 函数都要求使用旧样式的 C 字符串。在示例中, 我们可以创建并传递一个 C 字符串, 也可以创建一个 `NSString` 并通过它的方法 (名为 `UTF8String`) 派生一个 C 字符串。这两个方法都行。如果需要操纵字符串, 则使用 `NSString` 或 `NSMutableString` 比较容易, 但将 `NSString` 转换为 C 字符串会导致一些额外开销。

如果 `result` 等于 `SQLITE_OK`, 则语句准备成功, 可以开始遍历结果集。下面的例子将遍历结果集并从数据库中检索 `int` 和 `NSString`:

```
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
}
sqlite3_finalize(statement);
```

## 13.5.2 绑定变量

虽然可以通过创建 SQL 字符串来插入值, 但常用的方法是使用绑定变量 (bind variable) 来执行数据库插入操作。正确处理字符串并确保它们没有无效字符 (以及引号处理过的属性) 是非常烦琐的事情。借助绑定变量, 这些问题将迎刃而解。

要使用绑定变量插入值, 只需按正常方式创建 SQL 语句, 但要在 SQL 字符串中添加一个问号。每个问号都表示一个需要在语句执行之前进行绑定的变量。然后, 准备好 SQL 语句, 将值绑定到各个变量并执行命令。

下面这个示例使用两个绑定变量预处理 SQL 语句, 它将 `int` 绑定到第一个变量, 将字符串绑定到第二个变量, 然后执行并结束语句:

```
char *sql = "insert into foo values (?, ?);";
sqlite3_stmt *stmt;
if (sqlite3_prepare_v2(database, sql, -1, &stmt, nil) == SQLITE_OK) {
    sqlite3_bind_int(stmt, 1, 235);
    sqlite3_bind_text(stmt, 2, "Bar", -1, NULL);
}
if (sqlite3_step(stmt) != SQLITE_DONE)
    NSLog(@"This should be real error checking!");
sqlite3_finalize(stmt);
```

根据希望使用的数据类型,可以选择不同的绑定语句。大部分绑定函数都只有 3 个参数。

- ❑ 无论针对哪种数据类型,任何绑定函数的第一个参数都指向之前在 `sqlite3_prepare_v2()` 调用中使用的 `sqlite3_stmt`。
- ❑ 第二个参数是所绑定的变量的索引。它是一个有序索引的值,这表示 SQL 语句中的第一个问号是索引 1,而其后的每个问号都依次按序增加 1。
- ❑ 第三个参数始终表示应该替换问号的值。

有些绑定函数(比如说用于绑定文本和二进制数据的绑定函数)拥有另两个参数。

- ❑ 一个参数是在上面的第三个参数中传递的数据的长度。对于 C 字符串,可以传递 -1 来代替字符串的长度,则函数将使用整个字符串。对于所有其他情况,需要指定所传递数据的长度。
- ❑ 另一个参数是可选的函数回调,用于在语句执行后完成内存清理工作。通常,这种函数使用 `malloc()` 释放已分配的内存。

绑定语句后面的语法看起来可能有点奇怪,因为我们执行了一个插入操作。当使用绑定变量时,会将相同语法同时用于查询和更新。如果 SQL 字符串包含一个 SQL 查询(而不是更新),我们需要多次调用 `sqlite3_step()`,直到它返回 `SQLITE_DONE`。因为这里是更新,所以仅调用一次。

### 13.5.3 SQLite3 应用

在 Xcode 中,使用 Single View Application 模板创建一个新项目,命名为 SQLite Persistence。这个项目的开始部分与前一个项目相同,所以,先打开 `BIDViewController.m` 并进行如下修改:

```
#import "BIDViewController.h"

@interface BIDViewController ()

@property (strong, nonatomic) IBOutletCollection(UITextField) NSArray *lineFields;

@end
```

然后选中 `Main.storyboard`。根据本章前面的“设计 Persistence 应用的视图”(13.3.2 节第 2 条)的内容设计视图并关联输出接口。设计完成之后,保存分镜文件。

我们已经讲述了基本知识,下面付诸实践。再次改进 Persistence 应用,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表中 4 个行中。我们为每一行提供一个与其字段相对应的行号,例如,第一行的值存储在表中行号为 0 的行中,第二行的值存储在行号为 1 的行中,以此类推。下面让我们开始吧!

#### 1. 链接到 SQLite3 库

通过一个过程 API 来访问 SQLite3,该 API 提供对很多 C 函数调用的接口。要使用此 API,我们需要将应用链接到一个名为 `libsqlite3.dylib` 的动态库。在 Mac OS X 和 iOS 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链接完全相同。

选中项目导航列表(最左边的面板)顶部的 Persistence,然后在主区域的 TARGETS 部分(即

图 13-4 中间的面板) 选择 SQLite Persistence。注意要从 TARGETS 部分选择 Persistence, 而不是从 PROJECT 部分选择。

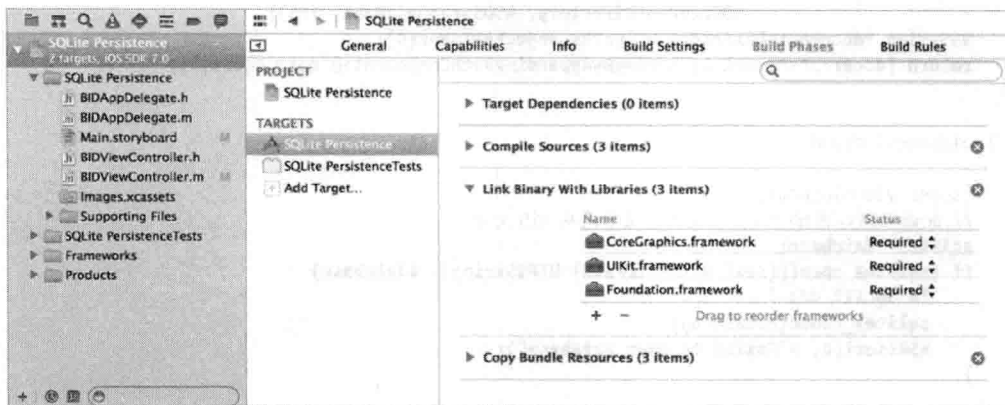


图 13-4 在项目导航面板中选择 SQLite Persistence 项目, 然后选择 Persistence 目标, 最后选择 Build Phases 标签

在选中了 Persistence 目标后, 在最右边的面板中点击 Build Phases 标签。其中包含一些列表项, 初始都是收起的, 它们代表 Xcode 构建应用的各个步骤。展开一行名为 Link Binary With Libraries 的项。其中包括了我们的应用创建时默认链接的标准框架: CoreGraphics.framework、UIKit.framework 和 Foundation.framework。

现在, 我们来向项目添加 SQLite3 库。点击链接框架列表底部的+按钮, 将会列出所有可用框架和库。在其中找到 libsqlite3.dylib (也可以使用搜索框), 然后点击 Add 按钮。注意, 目录中可能有多个以 libsqlite3 开头的条目, 务必选择 libsqlite3.dylib, 它是始终指向最新版本的 SQLite3 库的。在项目中添加了该库后, 将其拖到项目的 Frameworks 文件夹中。

## 2. 修改 Persistence 视图控制器

现在是时候来修改代码了。这次我们使用 SQLite 来替换 NSCoding 的相应代码。同样, 再次修改文件名, 以免跟上一个版本所用的文件重名, 我们希望文件名能适当反映了它所保存的数据的类型。然后我们修改用于保存和加载数据的方法。

选择 BIDViewController.m, 对其作如下修改:

```
#import "BIDViewController.h"
#import <sqlite3.h>

@interface BIDViewController ()

@property (strong, nonatomic) IBOutletCollection(UITextField) NSArray *lineFields;

@end

@implementation BIDViewController
```



```

- (NSString *)dataFilePath
{
    NSArray *paths = NSSearchPathForDirectoriesInDomains(
        NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];
    return [documentsDirectory stringByAppendingPathComponent:@"data.sqlite"];
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    // 加载视图后进行一些其他设置, 通常是从 nib 文件
    sqlite3 *database;
    if (sqlite3_open([[self dataFilePath] UTF8String], &database)
        != SQLITE_OK) {
        sqlite3_close(database);
        NSLog(@"Failed to open database");
    }

    // 有用的 C 语言小知识:
    // 如果两个内联的字符串之间只有空白 (包括换行符) 而没有其他字符,
    // 那么这两个字符串会被连接为一个字符串
    NSString *createSQL = @"CREATE TABLE IF NOT EXISTS FIELDS "
        "(ROW INTEGER PRIMARY KEY, FIELD_DATA TEXT);";

    char *errorMsg;
    if (sqlite3_exec(database, [createSQL UTF8String],
        NULL, NULL, &errorMsg) != SQLITE_OK) {
        sqlite3_close(database);
        NSLog(@"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 *fieldValue = [[NSString alloc]
                initWithUTF8String:rowData];
            UITextField *field = self.lineFields[row];
            field.text = fieldValue;
        }
        sqlite3_finalize(statement);
    }
    sqlite3_close(database);

    UIApplication *app = [UIApplication sharedApplication];
    [[NSNotificationCenter defaultCenter]
        addObserver:self
        selector:@selector(applicationWillResignActive:)
        name:UIApplicationWillResignActiveNotification
        object:app];
}

```



```

}

- (void)applicationWillResignActive:(NSNotification *)notification
{
    sqlite3 *database;
    if (sqlite3_open([[self dataFilePath] UTF8String], &database)
        != SQLITE_OK) {
        sqlite3_close(database);
        NSAssert(0, @"Failed to open database");
    }
    for (int i = 0; i < 4; i++) {
        UITextField *field = self.lineFields[i];
        // 内联字符串的连接, 又一次派上用场了
        char *update = "INSERT OR REPLACE INTO FIELDS (ROW, FIELD_DATA) "
            "VALUES (?, ?)";
        char *errorMsg = NULL;
        sqlite3_stmt *stmt;
        if (sqlite3_prepare_v2(database, update, -1, &stmt, nil)
            == SQLITE_OK) {
            sqlite3_bind_int(stmt, 1, i);
            sqlite3_bind_text(stmt, 2, [field.text UTF8String], -1, NULL);
        }
        if (sqlite3_step(stmt) != SQLITE_DONE)
            NSAssert(0, @"Error updating table: %s", errorMsg);
        sqlite3_finalize(stmt);
    }
    sqlite3_close(database);
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // 删除可以重新创建的资源
}

```

@end

新增的第一段代码位于 `viewDidLoad` 方法中。我们首先打开数据库, 如果在打开时遇到了问题, 则关闭它并抛出一个断言错误。

```

sqlite3 *database;
if (sqlite3_open([[self dataFilePath] UTF8String], &database)
    != SQLITE_OK) {
    sqlite3_close(database);
    NSAssert(0, @"Failed to open database");
}

```

接下来, 需要确保有一个表来保存我们的数据。可以使用 `CREATE TABLE` 完成此任务。通过指定 `IF NOT EXISTS`, 可以防止数据库覆盖现有数据。如果已有一个具有相同名称的表, 此命令会直接退出, 不执行任何操作, 所以可以在应用每次启动时安全地调用它, 无需显式检查表是否存在。

```

NSString *createSQL = @"CREATE TABLE IF NOT EXISTS FIELDS "
    "(ROW INTEGER PRIMARY KEY, FIELD_DATA TEXT)";

```

```

char *errorMsg;
if (sqlite3_exec(database, [createSQL UTF8String],
                 NULL, NULL, &errorMsg) != SQLITE_OK) {
    sqlite3_close(database);
    NSAssert(0, @"Error creating table: %s", errorMsg);
}

```

最后，需要加载数据。为此，使用 SELECT 语句。在这个简单的例子中，我们创建一个 SELECT 来从数据库请求所有行并要求 SQLite3 准备我们的 SELECT。要告诉 SQLite3 按行号排序各行，以便我们总是以相同顺序获取它们。否则，SQLite3 将按内部存储顺序返回各行。

```

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) {

```

然后遍历返回的每一行：

```

while (sqlite3_step(statement) == SQLITE_ROW) {
    抓取行号并将它存储在一个 int 变量中，然后抓取字段数据保存到 C 语言字符串中。

```

```

    int row = sqlite3_column_int(statement, 0);
    char *rowData = (char *)sqlite3_column_text(statement, 1);

```

接下来，利用从数据库获取的值设置相应的字段。

```

    NSString *fieldValue = [[NSString alloc]
                           initWithUTF8String:rowData];
    UITextField *field = self.lineFields[row];
    field.text = fieldValue;

```

最后关闭数据库连接，所有操作到此结束。

```

    }
    sqlite3_finalize(statement);
}
sqlite3_close(database);

```

请注意，我们在创建表和加载它所包含的所有数据后立即关闭了数据库连接，而不是在应用运行的整个过程中保持打开状态。这是管理连接最简单的方式，对于这个小应用，我们可以在需要连接时再打开它。在其他需要频繁使用数据库的应用中，可能有必要始终打开连接。

其他更改是在 `applicationWillResignActive:` 方法中进行的，我们需要把应用数据保存在这里。由于数据库中的数据存储在表中，存储后应用的数据看起来跟表 13-1 类似。

表13-1 存储在数据库的FIELDS表中的数据

行	FIELD_DATA
0	专为疯狂的人准备
1	专为不合时宜的人、叛逆者准备
2	专为麻烦制造者准备
3	专为异类准备

`applicationWillResignActive:`方法首先会再次打开数据库。然后保存数据,在4个字段中进行循环,生成4条独立的命令来更新数据库中的每一行。

```
for (int i = 0; i < 4; i++) {
    UITextField *field = self.lineFields[i];
```

我们在循环中要做的第一件事就是创建一个字段名称,以便可以检索到正确的文本框输出口。记住,使用 `valueForKey:` 可以根据名称检索属性。同时为错误消息声明一个指针,在出现错误时使用。

我们设计了一条带两个绑定变量的 `INSERT OR REPLACE` 的 SQL 语句。第一个变量代表所存储的行,第二个变量代表要存储的实际字符串值。使用 `INSERT OR REPLACE`,而不是更标准的 `INSERT`,就不需要担心某个行是否已经存在。

```
char *update = "INSERT OR REPLACE INTO FIELDS (ROW, FIELD_DATA) "
               "VALUES (?, ?);";
```

接下来声明一个指向语句的指针,然后为语句添加绑定变量,并将值绑定到两个绑定变量:

```
sqlite3_stmt *stmt;
if (sqlite3_prepare_v2(database, update, -1, &stmt, nil)
    == SQLITE_OK) {
    sqlite3_bind_int(stmt, 1, i);
    sqlite3_bind_text(stmt, 2, [field.text UTF8String], -1, NULL);
}
```

然后调用 `sqlite3_step` 来执行更新,检查并确定其运行正常,然后完成语句,结束循环:

```
if (sqlite3_step(stmt) != SQLITE_DONE)
    NSLog(@"Error updating table: %s", errorMsg);
sqlite3_finalize(stmt);
```

注意,此处使用了一个断言来检查错误条件。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。如果某个错误条件是用户正常情况下可能遇到的条件,则应该使用其他形式的错误检查。

---

**注意** 有一个条件可能导致前面的 SQLite 代码出现错误,但不是程序员错误。如果设备的存储区已满,SQLite 无法将其更改保存到数据库,那么这里也会发生错误。但是,这种情况很少见,并可能为用户带来更深层次的问题,不过这已超出了应用数据的范围。如果系统处于这一状态,我们的应用甚至可能无法成功启动。所以我们可以完全不用考虑这个问题。

---

完成循环之后,关闭数据库:

```
sqlite3_close(database);
```

为什么不编译、运行一下呢?输入一些数据,按 iPhone 模拟器的 Home 按钮。然后退出模拟器,重新启动 SQLite Persistence 应用,启动后,该数据应该处于原来的位置。在用户看来,这 3

个版本的应用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化机制。

## 13.6 使用 Core Data

本章演示的最后一项技术是如何使用苹果的 Core Data 框架实现持久化。Core Data 是一款稳定、功能全面的持久化工具。这里，我们将展示如何使用 Core Data 重新创建与 Persistence 应用相同的持久化。

---

**注意** 有关 Core Data 较为全面的讨论，请阅读 Kevin Kim、Alex Horovitz 和 David Mark 合著的 *More iOS 7 Development* (Apress, 2012)，其中有几章是专门讨论 Core Data 的。你也可能会对 Michael Privet 和 Robert Warner 合著的 *Pro Core Data for iOS* (Apress, 2011) 感兴趣。

---

在 Xcode 中创建一个新项目。这一次选择 Empty Application 模板，单击 Next。将产品命名为 Core Data Persistence，从 Device Family 弹出框中选择 iPhone，但先不要单击 Next 按钮。查看 Device Family 弹出窗口的下面，应该能看到一个标签为 Use Core Data 的复选框。将 Core Data 添加到已有项目的操作有一定复杂性，因此苹果公司为此选项提供了一些应用项目模板，帮助你完成大部分工作。

选中 Use Core Data 复选框（参见图 13-5），然后单击 Next 按钮。在提示窗口中选择保存项目的目录，然后单击 Create 按钮。

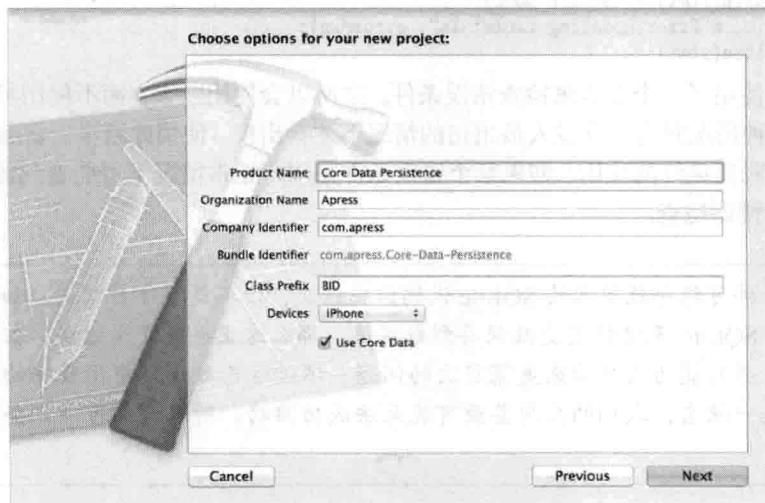


图 13-5 一些项目模板（包括 Empty Application）提供了使用 Core Data 的选项以便于持久化

在讨论代码之前，我们先来看项目窗口，其中包括一些全新的元素。展开 Core Data Persistence 文件夹（参见图 13-6）。

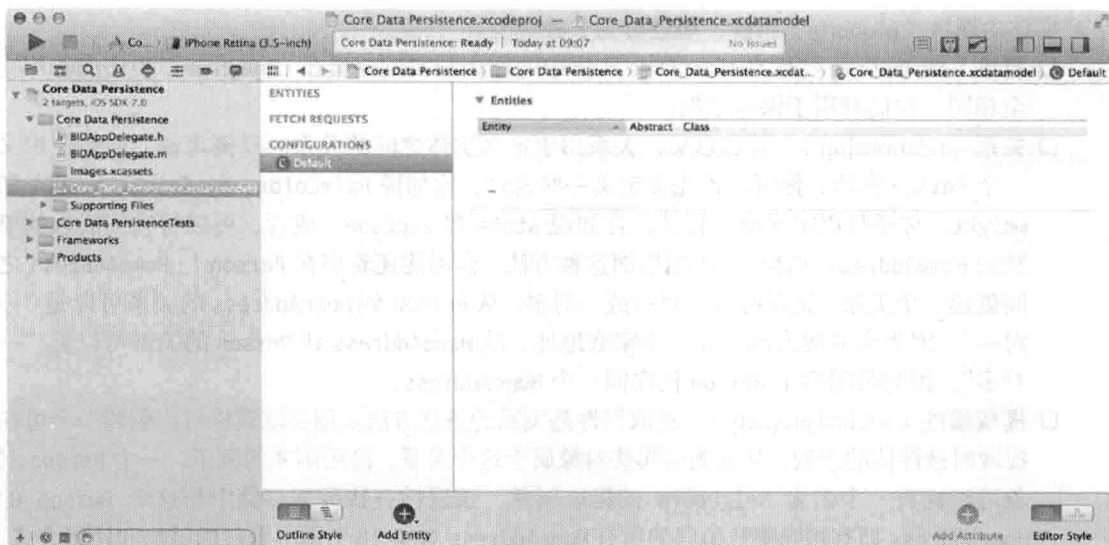


图 13-6 项目模板提供了 Core Data 所需的文件。当前已经选中了 Core Data 模型，数据模型编辑器显示在编辑面板中

### 13.6.1 实体和托管对象

项目导航面板中显示的大部分内容你都应该很熟悉：应用委托、图像资源目录。此外，还有一个名为 `Core_Data_Persistence.xcdatamodeld` 的文件，这个文件包含我们的数据模型。在 Xcode 中，Core Data 可用于直观地设计数据模型而无需编写代码，并将该数据模型存储在 `.xcdatamodeld` 文件中。

单击 `.xcdatamodeld` 文件，会显示出数据模型编辑器，如图 13-6 的右面部分所示。数据模型编辑器为数据模型提供了两种视图，可以在项目窗口右下角的选项处进行设置。Table 模式如图 13-6 所示，数据模型中包含的数据项将会显示为一系列可编辑的表；在 Graph 模式中，数据项是以图形方式来表示的。目前，这两个视图都显示了同样的空数据模型。

在 Core Data 之前，创建数据模型的传统方式是创建一个 `NSObject` 的子类并让它们遵循 `NSCoding` 和 `NSCopying`，以便能够像本章之前那样对它们进行归档。Core Data 使用了一种完全不同的方法。你不需要创建类，而是先在数据模型编辑器中创建一些实体（Entity），然后在代码中为这些实体创建托管对象（managed object）。

**注意** 术语“实体”和“托管对象”可能有点令人困惑，因为两者都表示数据模型对象。术语“实体”表示对对象的描述，而“托管对象”表示在运行时创建的该实体的具体实例。因此，在数据模型编辑器中，你将创建实体；而在代码中，你将创建并检索托管对象。实体和托管对象之间的差异类似于类与类的实例。

实体由属性（property）组成，属性分为 3 种类型。

- 特性（attribute）：特性在 Core Data 实体中的作用与实例变量在 Objective-C 类中的作用完全相同，它们都用于保存数据。
- 关系（relationship）：顾名思义，关系用于定义实体之间的关系。举例来说，假设要定义一个 Person 实体，你可能首先会定义一些特性，比如说 hairColor、eyeColor、height 和 weight。你还可以定义地址特性，比如说 state 和 zipCode。或者，可以将它们嵌入到单独的 HomeAddress 实体中。使用后面这种方法，你可能还希望在 Person 与 HomeAddress 之间创建一个关系。关系可以一对一或一对多。从 Person 到 HomeAddress 的关系可以是“一对一”，因为大多数人都只有一个家庭地址。从 HomeAddress 到 Person 的关系可以是“一对多”，因为可能多个 Person 住在同一个 HomeAddress。
- 提取属性（fetched property）：提取属性是关系的备选方法。用提取属性可以创建一个可在提取时被评估的查询，从而确定哪些对象属于这个关系。沿用刚才的例子，一个 Person 对象可以拥有一个名为 Neighbors 的提取属性，该属性查找数据存储中与这个 Person 的 HomeAddress 拥有相同邮政编码的所有 HomeAddress 对象。由于提取属性的结构和使用方式，它们通常都是一对一关系。提取属性也是唯一一种能够让你跨越多个数据存储的关系。

通常，特性、关系和提取属性都是使用 Xcode 的数据模型编辑器定义的。在 Core Data Persistence 应用中，我们将构建一个简单的实体，以便于你了解其运行原理。

### 1. 键-值编码

我们的代码中不再使用存取方法和修改方法，而是使用键-值编码来设置属性或检索它们的已有值。键-值编码看上去有点令人生畏，但本书已经在多处使用了这种方法。举例来说，每次在使用 NSDictionary 时，我们都需要使用键-值编码，因为字典中的每个对象都保存在一个唯一的键值中。与 NSDictionary 相比，Core Data 所使用的键-值编码更加复杂一些，但它们的基本概念都是相同的。

在操作托管对象时，用于设置和检索属性值的键就是希望设置的特性的名称。因此，要从托管对象中检索存储在 name 特性中的值，需要调用以下方法：

```
NSString *name = [myManagedObject valueForKey:@"name"];
```

同样，要为托管对象的属性设置新值，可以执行以下操作：

```
[myManagedObject setValue:@"Gregor Overlander" forKey:@"name"];
```

### 2. 在上下文中结合它们

那么，这些托管对象的活动区域在哪里呢？它们位于所谓的持久存储中，有时也称为支持存储（backing store）。持久存储可以采用多种不同的形式。默认情况下，Core Data 应用将支持存储实现为存储在应用 Documents 目录中的 SQLite 数据库。虽然数据是通过 SQLite 存储的，但 Core Data 框架中的类将完成与加载和保存数据相关的所有工作。如果使用 Core Data，则不需要编写任何 SQL 语句。你只需要操作对象，而内部的工作将由 Core Data 完成。

除了 SQLite 之外，支持存储还可以作为二进制文件实现，甚至以 XML 形式存储。还有一种

选择是创建一个内存库，编写缓存机制时可以采用这种方法，但它在当前会话结束后无法保存数据。在几乎所有情况下，你都应该采用默认设置，并使用 SQLite 作为持久存储。

虽然大多数应用都只有一个持久存储，但也可以在同一应用中使用多个持久存储。如果你对支持存储的创建和配置方式感到好奇，可以查看 Xcode 项目中的 `BIDAppDelegate.m` 文件。我们选择的 Xcode 项目模板提供了为应用设置单个持久存储所需的所有代码。

除了创建它之外（在应用委托中实现），我们通常不会直接操作持久存储，而是使用所谓的托管对象上下文（通常称为上下文）。上下文协调对持久存储的访问，同时保存自上次保存对象以来修改过的属性的信息。上下文还能通过撤销管理器来注册所有更改，这意味着你可以撤销单个操作或者回滚到上次保存的数据。

---

**注意** 可以将多个上下文指向相同的持久存储，但大多数 iOS 应用都只会使用一个。有关使用多个上下文和撤销管理器的更多信息，请阅读 *More iOS 7 Development* 一书（Apress, 2014）。

---

许多核心数据调用都需要 `NSManagedObjectContext` 作为参数，或者需要在上下文中执行。除了一些更加复杂、多线程的 iOS 应用之外，应用委托中都可以只使用 `managedObjectContext` 属性——它是 Xcode 项目模板自动为应用创建的默认上下文。

你可能会发现，除了托管对象上下文和持久存储协调者之外，所提供的应用委托还包含一个 `NSManagedObjectContext` 实例。该类负责在运行时加载和表示使用 Xcode 中的数据模型编辑器创建的数据模型。通常，你不需要直接与该类交互。该类由其他 Core Data 类在后台使用，因此它们可以确定数据模型中定义了哪些实体和属性。只要使用所提供的文件创建数据模型，就完全不需要担心这个类。

### 3. 创建新的托管对象

创建托管对象的新实例非常简单，但没有使用 `alloc` 和 `init` 创建常规对象实例那么直观。相反，这里使用 `NSEntityDescription` 类中的 `insertNewObjectForEntityForName:inManagedObjectContext:` 工厂方法。`NSEntityDescription` 的工作是跟踪在应用的数据模型中定义的所有实体并能够让你创建这些实体的实例。此方法创建并返回一个实例，表示内存中的单个实体。它返回使用该特定实体的正确属性设置的 `NSManagedObject` 实例，或者如果将实体配置为使用 `NSManagedObject` 的特定子类实现，则返回该类的实例。请记住，实体类似于类。实体是对象的描述，用于定义特定的实体拥有哪些属性。

创建新对象的方法如下：

```
NSManagedObject *thing = [NSEntityDescription
                           insertNewObjectForEntityForName:@"Thing"
                           inManagedObjectContext:context];
```

这个方法名称为 `insertNewObjectForEntityForName:inManagedObjectContext:`，因为除了创建新对象外，它还将此新对象插入到上下文，并返回这个对象。调用结束后，对象存在于上下文中，但还不是持久存储的一部分。下一次托管对象上下文的 `save:` 方法被调用时，这个对象将



被添加到持久存储内。

#### 4. 获取托管对象

要从持久存储中获取托管的对象，可以使用获取请求（fetch request），这是 Core Data 处理预定义的查询的方式。例如，可以要求“返回所有 eyeColor 为蓝色的 Person”。

首次创建获取请求之后，为它提供一个 `NSEntityDescription`，指定希望检索的一个或多个对象实体。下面是一个创建获取请求的例子：

```
NSFetchRequest *request = [[NSFetchRequest alloc] init];
NSEntityDescription *entityDescr = [NSEntityDescription
    entityWithName:@"Thing" inManagedObjectContext:context];
[request setEntity:entityDescr];
```

也可以使用 `NSPredicate` 类为获取请求指定条件。谓语句（predicate）类似于 SQL 的 WHERE 子句，可定义条件让获取请求得出结果。下面是一个简单的谓语句示例：

```
NSPredicate *pred = [NSPredicate predicateWithFormat:@"(name = %@)", nameString];
[request setPredicate: pred];
```

第一行代码创建的谓语句告诉获取请求，无需获取指定实体的所有托管对象，它应仅获取那些 name 属性被设置为当前存储在 nameString 变量中的值的托管对象。所以，如果 nameString 是一个包含值 @"Bob" 的 NSString，则会告诉获取请求仅返回其 name 属性设置为 "Bob" 的托管对象。这是一个简单的例子，谓语句可能复杂得多，还可以使用布尔逻辑来指定在大部分情形下可能需要的准确条件。

---

**注意** Scott Knaster、Waqar Maliq 和 Mark Dalrymple 合著的《Objective-C 基础教程（第 2 版）》专门用一章介绍了 `NSPredicate` 的使用。

---

创建了获取请求并为它提供实体描述之后（可以选择为它指定一个谓语句），使用 `NSManagedObjectContext` 中的实例方法来执行获取请求：

```
NSError *error;
NSArray *objects = [context executeFetchRequest:request error:&error];
if (objects == nil) {
    // 错误处理
}
```

`executeFetchRequest:error:` 将从持久存储中加载指定对象，并在一个数组中返回它们。如果遇到错误，则会获得一个 nil 数组，并且你所提供的错误指针将指向描述特定问题的 `NSError` 对象。如果没有遇到错误，则会获得一个有效的数组，但其中可能没有任何对象，因为可能没有对象满足指定标准。此后，托管对象上下文（对它执行了请求）将跟踪对该数组中返回的托管对象的所有更改。向该上下文发送一条 `save:` 消息可保存更改。

### 13.6.2 Core Data 应用

接下来，暂时放下 Core Data，将注意力放回到 Xcode，开始创建数据模型。



## 1. 设计数据模型

单击 `Core_Data_Persistence.xcdatamodeld`，打开 Xcode 的数据模型编辑器。数据模型编辑面板中列出了数据模型中的所有实体、获取请求和配置。

13

**注意** Core Data 的配置允许开发者定义一个或多个包含在数据模型中的实体的命名子集，在某些特定场合下，这一点很有用。例如，如果你想创建一些共享相同数据模型的应用，而另一些应用不能获取这些数据模型（比如一个应用提供给普通用户使用，而另一个给系统管理员使用），则通过这种方式就可以做到。也可以在一个应用中使用多个配置，在不同操作模式之间进行切换。本书中，我们并不涉及配置，但是由于配置列表（包括在你的模型中包含所有内容的默认配置）使你得以接触实体和获取请求的底层内容，所以我们认为有必要在这里提一下。

在图 13-6 中，这些列表都是空的，因为我们还没有创建任何内容。单击实体面板左下方的加号图标（标为 `Add Entity`），选择创建一个名称为 `Entity` 的实体（如图 13-7 所示）。

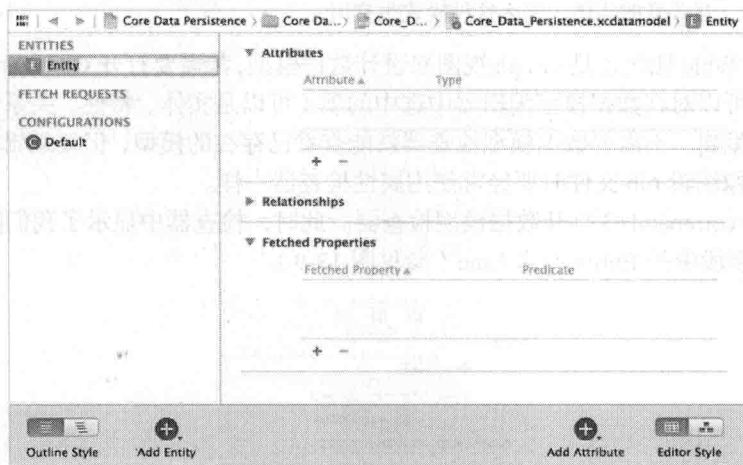


图 13-7 数据模型编辑器，其中显示了刚添加的实体

构建数据模型时，需要使用编辑区域右下方的 `Editor Style` 控件在 `Table` 视图和 `Graph` 视图之间进行切换。现在切换到 `Graph` 视图。`Graph` 视图显示一个代表实体的小方框，它包含用于显示该实体的特性和关系的部分，现在也是空的（参见图 13-8）。如果你的模型包含多个实体，那么 `Graph` 视图会非常有用，它以图形化方式显示了所有实体之间的关系。

**注意** 如果你倾向于使用图形化方式工作，那么可以在 `Graph` 视图下构建实体模型。本章使用 `Table` 视图，因为这种模式更容易解释。当你创建自己的数据模型时，可以随意使用 `Graph` 视图（如果这种方式更适合你的话）。

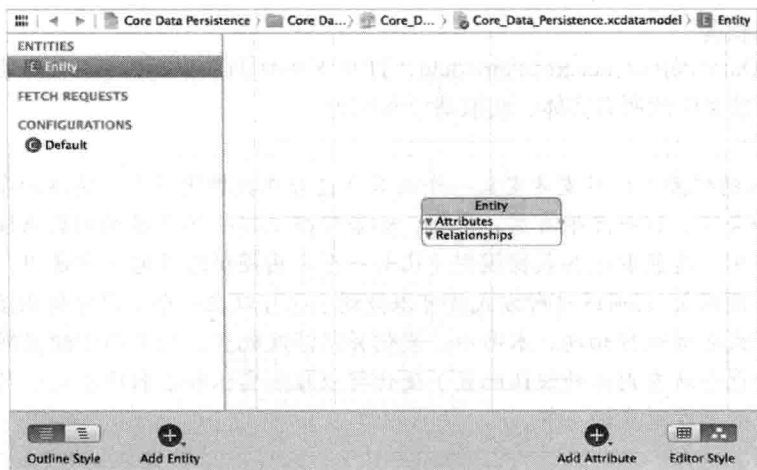


图 13-8 使用右下方的控件将数据模型编辑器切换到 Graph 模式。注意，Graph 模式显示的实体与 Table 模式下的相同，只是以图形化方式显示。如果有多个相互关联的实体，那么这种模式很有用

无论是使用 Table 视图还是 Graph 视图来设计数据模型，都需要打开 Core Data 数据模型检查器。这个检查器可以对在数据模型编辑器中选中的项（可以是实体、特性、关系等内容）的相关细节进行查看和编辑。不需要数据模型检查器就能查看已存在的模型，但要编辑模型，就得使用这个检查器，就像编辑 nib 文件时要经常使用属性检查器一样。

按下 `option+command+3` 打开数据模型检查器。此时，检查器中显示了我们刚添加的实体的信息。将 Name 字段中的 Entity 改为 Line（参见图 13-9）。

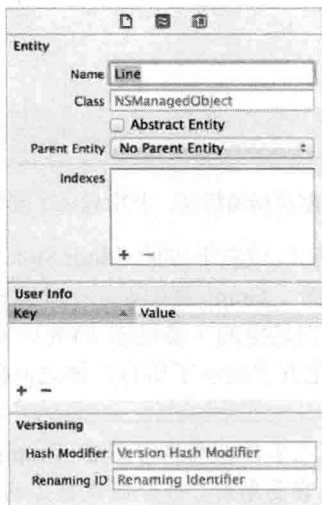


图 13-9 使用数据模型检查器将实体的名称改为 Line

如果你现在位于 Graph 视图中,那么请切换到 Table 视图。Table 视图显示了当前编辑实体的更为详细的信息,所以在创建一个新实体时 Table 视图通常比 Graph 视图更有用。在 Table 视图中,数据模型编辑器大部分都用于显示该实体的特性、关系和提取属性。我们就在这里设置实体。

注意,在编辑区域的右下方有一个加号图标,与左下方用于创建实体的图标很相似。如果你选择实体,然后按住鼠标按键并点击那个加号图标,将会出现一个弹出菜单,可用来向实体添加特性、关系或者提取属性(参见图 13-10)。

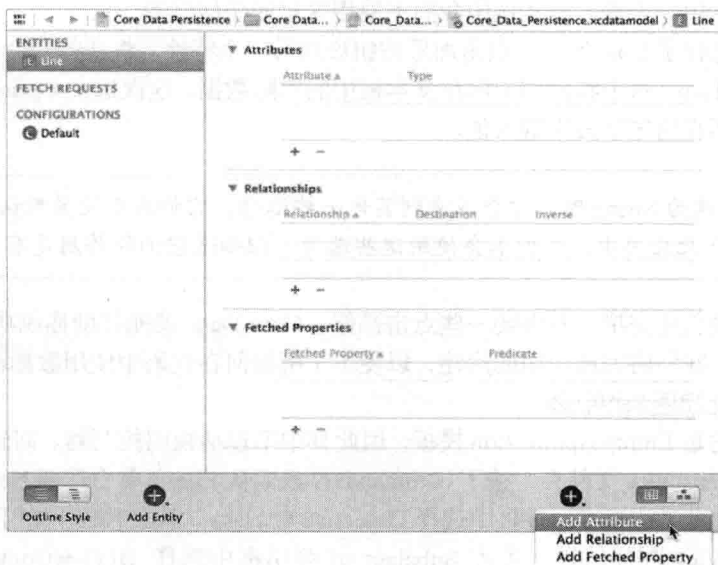


图 13-10 选择一个实体,按住右下方加号图标,为该实体添加一个特性、关系或提取属性

**注意** 不需要通过点击并按住鼠标按键来添加特性,直接点击加号图标就可以了。这是种快捷方式!

用这种方式为 Line 实体添加一个特性。新创建的特性初始名称为 attribute,添加在表和所选项中的 Attributes 部分。从表中可以看到,不仅是该行被选中了,特性名也选中了。这意味着点击加号图标之后,可以立即为新特性键入名字,而不用另外再点击它。

将这个新特性的名字从 attribute 改为 lineNumber,然后点击名字旁边的弹出框,将其 Type 从 Undefined 改为 Integer16,从而将该特性转换为保存整型值的类型。我们将使用这个特性来标识托管对象中保存的 4 个字段中的某一个。由于我们只有 4 个选项,所以选择最小的整型类型。

现在将注意力转向数据模型检查器,这里可以配置额外的详细信息。Name 字段右下方的复选框 Optional,默认处于选中状态,点击它以取消选择。我们不希望这个特性是可选的(在界面中没有对应的标签的行是没有用处的)。

选择 **Transient** 复选框可以创建一个瞬态特性，这个特性用于将一个值指定为这样的：当应用运行时托管对象会持有该值，但是不会将它保存在数据存储区中。因为我们想要行号保存在数据存储区中，所以不能选中 **Transient** 复选框。

选择 **Indexed** 复选框会在底层 SQL 数据库中为保存此特性的字段创建索引。同样不要选择该复选框，因为数据量很小，而且我们不会向用户提供搜索功能，所以不需要索引。

这些选项的下面还有更多设置，你可以在这里做一些简单的数据验证，为整数指定最小值和最大值，设置默认值，等等。在本例中我们不使用这里的任何设置。

现在，确保选择了 **Line** 实体，点击加号按钮添加另一个特性。将新特性命名为 **lineText**，并将其 **Type** 改为 **String**。这个特性用于保存文本框中的实际数据。这次选中 **Optional** 复选框，因为用户完全有可能不在给定字段中键入值。

---

**注意** 将 **Type** 更改为 **String** 时，你会注意到其他一些选项，它们用于设置默认值或限制字符串的长度。在此应用中，我们不会使用这些选项，但知道它们的作用是有好处的。

---

数据模型已经创建完毕。只需要一些点击操作，**Core Data** 就能帮助你成功创建一个应用数据模型。接下来，我们将完成应用的构建，以便于了解如何在代码中使用数据模型。

## 2. 创建持久化视图和控制器

由于我们选的是 **Empty Application** 模板，因此其中不包括视图控制器。回到项目导航面板，单击 **Core Data Persistence** 文件夹，按下 **command+N** 或者从 **File** 菜单中选择 **New > New File...**，打开新文件向导。从 **Cocoa Touch** 栏中选择 **Objective-C class**，单击 **Next**。在下一个表单中，将新类命名为 **BIDViewController**，并从 **Subclass of** 弹出框中选择 **UIViewController**。确保标有 **Targeted for iPad** 的框处于未选中状态。还需要确保选中了 **With XIB for user interface** 复选框，这样系统会自动为你创建 **nib** 文件。我们在整本书中几乎只用到了分镜，不过 **nib** 文件对于简单的 GUI 仍然有效，所以有必要了解一下它的工作方式。单击 **Next** 按钮，选择文件保存目录。完成这些后，**BIDViewController.h**、**BIDViewController.m** 和 **BIDViewController.xib** 应该位于 **Core Data Persistence** 文件夹中。

单击 **BIDViewController.h**，并作出以下更改（你对此应该非常熟悉了）：

```
#import "BIDViewController.h"
```

```
@interface BIDViewController ()
```

```
@property (strong, nonatomic) IBOutletCollection(UITextField) NSArray *lineFields;
```

```
@end
```

保存文件，双击 **BIDViewController.xib** 以在界面构建器中编辑 GUI。设计视图，并依照 13.3.2 节第 2 条“设计 Persistence 应用的视图”中的步骤连接输出接口。回头看图 13-3 时你会发现它很有用。你会发现最大的区别是这里没有用 **View Controller** 图标来表示我们的视图控制器，而是用 **File's Owner** 图标。**nib** 文件与分镜场景不同，后者的 **View Controller** 图标会在每个全屏视图的

下方。而在 nib 文件中, File's Owner 图标仅出现在左侧的文件大纲内。设计完成之后, 请保存 nib 文件。

回到 BIDViewController.m 文件, 按以下方式更改代码:

```
#import "BIDViewController.h"
#import "BIDAppDelegate.h"

static NSString * const kLineEntityName = @"Line";
static NSString * const kLineNumberKey = @"lineNumber";
static NSString * const kLineTextKey = @"lineText";

@interface BIDViewController ()

@property (strong, nonatomic) IBOutletCollection(UITextField) NSArray *lineFields;

@end

@implementation BIDViewController

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        // 自定义初始化
    }
    return self;
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    BIDAppDelegate *appDelegate = [UIApplication sharedApplication].delegate;
    NSManagedObjectContext *context = [appDelegate managedObjectContext];
    NSFetchedRequest *request = [[NSFetchRequest alloc]
                                initWithEntityName:kLineEntityName];

    NSError *error;
    NSArray *objects = [context executeFetchRequest:request error:&error];
    if (objects == nil) {
        NSLog(@"There was an error!");
        // 做一些适当的错误处理
    }

    for (NSManagedObject *oneObject in objects) {
        int lineNum = [[oneObject valueForKey:kLineNumberKey] intValue];
        NSString *lineText = [oneObject valueForKey:kLineTextKey];

        UITextField *theField = self.lineFields[lineNum];
        theField.text = lineText;
    }

    UIApplication *app = [UIApplication sharedApplication];
    [[NSNotificationCenter defaultCenter]
     addObserver:self
```

```

        selector:@selector(applicationWillResignActive:)
        name:UIApplicationWillResignActiveNotification
        object:app];
    }

- (void)applicationWillResignActive:(NSNotification *)notification
{
    BIDAppDelegate *appDelegate = [UIApplication sharedApplication].delegate;
    NSManagedObjectContext *context = [appDelegate managedObjectContext];
    NSError *error;
    for (int i = 0; i < 4; i++) {
        UITextField *theField = self.lineFields[i];

        NSFetchedRequest *request = [[NSFetchedRequest alloc]
                                     initWithEntityName:kLineEntityName];
        NSPredicate *pred = [NSPredicate
                             predicateWithFormat:@"%K = %d", kLineNumberKey, i];
        [request setPredicate:pred];

        NSArray *objects = [context executeFetchRequest:request error:&error];
        if (objects == nil) {
            NSLog(@"There was an error!");
            // 做一些适当的错误处理
        }

        NSManagedObject *theLine = nil;
        if ([objects count] > 0) {
            theLine = [objects objectAtIndex:0];
        } else {
            theLine = [NSEntityDescription
                      insertNewObjectForEntityForName:kLineEntityName
                      inManagedObjectContext:context];
        }
        [theLine setValue:[NSNumber numberWithInt:i] forKey:kLineNumberKey];
        [theLine setValue:theField.text forKey:kLineTextKey];
    }
    [appDelegate saveContext];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // 删除可以重新创建的资源
}

@end

```

现在看一下 `viewDidLoad` 方法。我们需要确定持久存储中是否已经存在数据，如果有则加载数据并使用它填充字段。该方法首先获取对应用委托的引用，我们将使用这个引用获得为我们创建的托管对象上下文：

```

BIDAppDelegate *appDelegate = [UIApplication sharedApplication].delegate;
NSManagedObjectContext *context = [appDelegate managedObjectContext];

```

下一个步骤是创建一个获取请求并将实体描述传递给它，以便请求知道要检索的对象类型：

```
NSFetchRequest *request = [[NSFetchRequest alloc]
    initWithEntityName:kLineEntityName];
```

由于我们希望检索持久存储中的所有 Line 对象，因此没有创建谓词。通过执行没有谓词的请求，上下文将返回库中的每一个 Line 对象。确保返回的是有效的数组，如果不是则记录相应的日志。

```
NSError *error;
NSArray *objects = [context executeFetchRequest:request error:&error];
if (objects == nil) {
    NSLog(@"There was an error!");
    // 做一些适当的错误处理
}
```

接下来，我们使用快速枚举遍历已获取托管对象的数组，从中提取每个托管对象的 lineNumber 和 lineText 值，并使用该信息更新用户界面上的一个文本框。

```
for (NSManagedObject *oneObject in objects) {
    int lineNum = [[oneObject valueForKey:kLineNumberKey] intValue];
    NSString *lineText = [oneObject valueForKey:kLineTextKey];

    UITextField *theField = self.lineFields[lineNum];
    theField.text = lineText;
}
```

然后，与本章中的所有其他应用一样，我们需要在应用即将终止（无论是转到后台还是完全退出）的时候获取通知，以便能够保存用户对数据作出的任何更改：

```
UIApplication *app = [UIApplication sharedApplication];
[[NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(applicationWillResignActive:)
    name:UIApplicationWillResignActiveNotification
    object:app];
```

我们接下来讨论 applicationWillResignActive:。这里使用的方法和前面一样，先获取对应应用委托的引用，然后使用此引用获取指向应用的默认上下文的指针。

```
BIDAppDelegate *appDelegate = [UIApplication sharedApplication].delegate;
NSManagedObjectContext *context = [appDelegate managedObjectContext];
```

然后，使用循环语句为每个标签执行 1 次，共 4 次，获得每一个字段对应的索引。

```
for (int i = 0; i < 4; i++) {
    UITextField *theField = self.lineFields[i];
```

接下来，为 Line 实体创建获取请求。需要确认持久存储中是否已经有一个与这个字段对应的托管对象，因此创建一个谓词，用于为字段标识正确的对象。

```
NSFetchRequest *request = [[NSFetchRequest alloc]
    initWithEntityName:kLineEntityName];

NSPredicate *pred = [NSPredicate
    predicateWithFormat:@"%K = %d", kLineNumberKey, i];
[request setPredicate:pred];
```

此时在上下文中执行获取请求并且检查 `objects` 是否为 `nil`。如果为 `nil`，则表示遇到错误，我们应该为应用执行合适的错误处理。对于此示例应用，我们仅仅记录错误并继续运行。

```
NSArray *objects = [context executeFetchRequest:request error:&error];
if (objects == nil) {
    NSLog(@"There was an error!");
    // 做一些适当的错误处理
}
```

现在声明一个指向 `NSManagedObject` 的指针并将它设置为 `nil`。执行此操作的原因是，我们还不知道是要从持久存储中加载托管对象，还是创建新的托管对象。因此，可以检查与条件匹配的返回对象。如果返回了有效的对象，就加载，否则就创建一个新的托管对象来保存这个字段的文本。

```
NSManagedObject *theLine = nil;
if ([objects count] > 0) {
    theLine = [objects objectAtIndex:0];
} else {
    theLine = [NSEntityDescription
                insertNewObjectForEntityForName:kLineEntityName
                inManagedObjectContext:context];
}
```

接着，使用键-值编码来设置行号以及此托管对象的文本：

```
[theLine setValue:[NSNumber numberWithInt:i] forKey:kLineNumberKey];
[theLine setValue:theField.text forKey:kLineTextKey];
```

最后，完成循环之后，通知上下文保存其更改：

```
[appDelegate saveContext];
```

### 3. 将持久化视图控制器设置为应用的根控制器

由于我们使用 `Empty Application` 模板，而不是 `Single View Application` 模板，因而还需要一个步骤才能让新的 `Core Data` 应用正常运行。需要创建一个 `BIDViewController` 实例来充当应用的根控制器，并将其视图添加为应用主窗口的子视图。现在就开始行动吧！

要将根控制器的视图设置为应用窗口的子视图以便用户与之交互，打开 `BIDAppDelegate.m` 并在该文件顶部作如下更改：

```
#import "BIDAppDelegate.h"
#import "BIDViewController.h"

@implementation BIDAppDelegate

@synthesize managedObjectContext = _managedObjectContext;
@synthesize managedObjectModel = _managedObjectModel;
@synthesize persistentStoreCoordinator = _persistentStoreCoordinator;

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];
    // 程序启动之后的一些自定义设置
    BIDViewController *controller = [[BIDViewController alloc] init];
    self.window.rootViewController = controller;
```



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

大功告成。编译并运行以确保程序正常运行。Core Data 版本的应用应该与之前的版本功能完全相同。

Core Data 需要的工作量很大，对于这种简单的应用来说，它并没有提供明显的优势。但在比较复杂的应用中，Core Data 可以显著减少设计和编写数据模型所需的时间。

## 13.7 小结

现在，你应该牢固掌握了在会话之间保存应用数据的 4 种方法，如果包括上一章介绍的用户默认值方法，则为 5 种。我们使用属性列表构建了持久保存数据的应用，并将该应用修改为使用对象归档来保存数据。然后进行更改，使用 iOS 内置的 SQLite3 机制来保存应用数据。最后，使用 Core Data 重新构建了同样的应用。几乎所有 iOS 应用都使用这些机制来保存和加载数据。

近年来，iOS 中添加的一大新特性是苹果公司的 iCloud 服务，iCloud 为 iOS 设备以及运行 OS X 系统的计算机提供云存储服务。大部分 iOS 用户都能在购买新设备或者将旧设备升级到新版的 iOS 后，获得 iCloud 设备备份选项，他们很快就能发现它的自动备份优势——甚至不需要使用计算机就能完成备份。

不需要借助计算机就能完成备份是一项很强大的特性，但它对 iCloud 来说仅是小菜一碟。或许，iCloud 更强大的特性是为应用程序开发者提供了一种能够轻松向苹果公司的云服务器透明存储数据的机制。你可以将应用中的数据存储在 iCloud 中，然后自动传输给同一个 iCloud 用户的其他设备。用户可能在 iPad 上创建了一份文档，而后就可以在他们的 iPhone 或 Mac 上浏览同一份文档。这不需要任何干预措施，文档就能出现在其他设备上了。

由一个系统进程负责验证用户是否正确登录了 iCloud，并且管理文件传输，所以你完全不需要担心网络或者身份验证。只需要少量的应用配置，在保存文件、定位可用文件方面对方法进行一些小修改，就能创建支持 iCloud 的应用。

iCloud 文件归档系统的一个关键组成部分是 `UIDocument` 类。`UIDocument` 通过处理一些读取、写入文件方面的常规工作，省去了一部分创建基于文档的应用的工作。通过这种方式，开发者可以更多地关注特定于应用的功能，而不是为所有应用构建相同的内容。

无论你是否使用 iCloud，`UIDocument` 都为在 iOS 中管理文档文件提供了一些强大的工具。为了演示这些特性，本章的第一部分将创建 TinyPix 项目，这是一个将文件保存在本地存储器上的基于文档的简单应用。这种方式能够很好地运用到 iOS 上所有类型的应用。

本章稍后将介绍如何让 TinyPix 支持 iCloud。为此，你需要有一个或多个连接到 iCloud 的 iOS 设备。你还需要一个付费的 iOS 开发者账号，以便在设备上安装该应用，这是因为模拟器中运行的应用程序不能访问 iCloud 服务。

## 14.1 使用 `UIDocument` 管理文档存储

任何使用过台式计算机（除了仅用来网上冲浪）的人都应该用过基于文档的应用，比如 `TextEdit`、`Microsoft Word`、`GarageBand`、`Xcode` 等。这些可以处理多个数据集并将每个数据集保存到独立文件中的应用，都可以看做基于文档的应用程序。通常，屏幕窗口和它所包含的文档是一一对应

的,但有时(比如 Xcode),一个窗口可以通过某种方式显示多个相互关联的文档。

iOS 设备上不支持多个窗口,但是大部分应用仍然受益于基于文档的方式。现在多亏有了 UIDocument 类来负责文档文件存储方面的大部分常规工作,iOS 开发者可以更加轻松地存储文档了。开发者不需要直接处理文件(只要 URL),所有必要的文件读写工作都在后台线程中进行,所以就算正在访问文件,应用也能保持响应。另外,它还自动定期保存编辑过的文档,当应用程序挂起时(比如关闭设备,按下主屏幕按钮等)也会自动保存,所以不再需要任何类型的保存按钮。所有这些都有助于使应用程序的行为更符合用户的期望。

### 14.1.1 构建 TinyPix

我们将要构建一个名为 TinyPix 的应用,该应用可以使用 1 位颜色编辑简单的 8×8 图像(参见图 14-1)。为了方便用户,每个图像都在全屏方式下编辑。当然,我们将使用 UIDocument 来表示每个图像的数据。

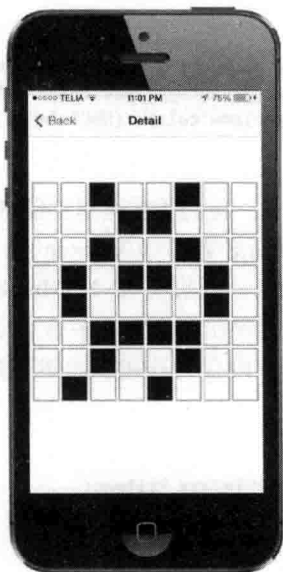


图 14-1 在 TinyPix 中编辑极低分辨率的图标

首先在 Xcode 中创建一个新项目。从 iOS Application 部分选择 Master-Detail Application 模板,然后单击 Next。将这个新项目命名为 TinyPix,将 Devices 设为 iPhone,确保没有选中 Use Core Data 复选框。然后再次单击 Next,为该项目选择保存位置。

Xcode 的项目导航面板中包含了 BIDAppDelegate、BIDMasterViewController 和 BIDDetailViewController 的相关文件,以及 Main.storyboard 文件。这些文件中的大部分都需要作一些修改,此外,我们还将创建一些新的类。

### 14.1.2 创建BIDTinyPixDocument类

首先要创建的是一个文档类，用于包含每个从文件存储器中加载的 TinyPix 图像的数据。在 Xcode 中选择 TinyPix 文件夹，按下 `command+N` 创建一个新文件。在 iOS Cocoa Touch 部分，选择 Objective-C class，然后点击 Next。在 Class 文本框内键入 BIDTinyPixDocument，Subclass of 文本框内键入 UIDocument，然后点击 Next。最后点击 Create 创建文件。

在深入实现细节之前，我们先来考虑一下该类的公共 API。这个类要显示一个 8×8 的像素网格，每个像素由一个表示开启或关闭的值构成。所以我们这里要创建 3 个方法：用一个方法来接收一对行和列索引作为参数，返回 BOOL 值，用另一个方法为指定的行和列设置特定的状态。为了方便起见，我们再用一个方法来单纯负责切换特定位置处的状态。

选择 BIDTinyPixDocument.h 以编辑这个新类的头文件，添加如下粗体所示代码：

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

@interface BIDTinyPixDocument : UIDocument

// 行和列的索引范围都是 0 到 7
- (BOOL)stateAtRow:(NSUInteger)row column:(NSUInteger)column;
- (void)setState:(BOOL)state atRow:(NSUInteger)row column:(NSUInteger)column;
- (void)toggleStateAtRow:(NSUInteger)row column:(NSUInteger)column;

@end
```

现在转到 BIDTinyPixDocument.m，我们要在这里存储 8×8 像素网格，实现在公共 API 中定义的方法，以及实现加载、保存文档所必需的 UIDocument 方法。

首先为 8×8 位图数据定义存储区。我们将在 NSMutableData 的实例中保存该数据，通过 NSMutableData，我们可以直接使用包含在对象中的字节数据的数组，这样当我们不再使用该数据时，Cocoa 内存管理将会为我们释放内存。现在添加如下类扩展：

```
#import "BIDTinyPixDocument.h"

@interface BIDTinyPixDocument ()
@property (strong, nonatomic) NSMutableData *bitmap;
@end

@implementation BIDTinyPixDocument
```

UIDocument 类指定了所有子类都应该使用的初始化方法。我们将要在该方法中创建初始位图。在真实的位图类型中，我们需要使用单字节来包含每一行，尽量减少内存的使用。字节中的每一位代表了行中每一个列索引的开/关值。我们的文档总共只有 8 字节。

---

**注意** 这部分包含少量位操作，以及一些 C 指针和数组操作。这些内容对 C 开发者来说很平常，但是如果你没有 C 语言开发经验，可能会感到迷惑或者完全无法理解。如果是这样的话，你可以纯粹地复制和使用这里所提供的代码（它们都可以正常运行）。如果你想要彻底理解其原理，先要深入理解 C 语言，我们推荐你阅读 Dave Mark 编著的 *Learn C on the Mac* (Apress, 2009) 一书。

---

在文档实现文件中添加如下方法，将其添加在文件底部@end之前：

```
- (id)initWithFileURL:(NSURL *)url {
    self = [super initWithFileURL:url];
    if (self) {
        unsigned char startPattern[] = {
            0x01,
            0x02,
            0x04,
            0x08,
            0x10,
            0x20,
            0x40,
            0x80
        };

        self.bitmap = [NSMutableData dataWithBytes:startPattern length:8];
    }
    return self;
}
```

这段代码将每个位图初始化为从一个角延伸到另一个角的对角线图案。

现在开始实现在头文件中定义的公共 API 方法。首先来实现读取单个位的状态的方法。实现这个方法只要从字节数组中获取相关字节，然后对其进行移位操作和 AND 操作，检查是否设置了给定位，相应地返回 YES 或 NO。在@end之前添加如下方法：

```
- (BOOL)stateAtRow:(NSUInteger)row column:(NSUInteger)column {
    const char *bitmapBytes = [self.bitmap bytes];
    char rowByte = bitmapBytes[row];
    char result = (1 << column) & rowByte;
    if (result != 0) {
        return YES;
    } else {
        return NO;
    }
}
```

下一个方法正好相反，它用于为给定行和列的位设置值。这里，我们再一次获取指向指定行相关字节的指针，并且进行一些移位操作。但是这一次，我们不是用移位来检查行的内容，而是用来设置或者清空行中某一位的值。在@end之前添加如下代码：

```
- (void)setState:(BOOL)state atRow:(NSUInteger)row column:(NSUInteger)column {
    char *bitmapBytes = [self.bitmap mutableBytes];
    char *rowByte = &bitmapBytes[row];

    if (state) {
        *rowByte = *rowByte | (1 << column);
    } else {
        *rowByte = *rowByte & ~(1 << column);
    }
}
```

现在我们来添加辅助方法，外部代码使用该方法来切换单个单元的状态。在@end前添加以下方法：

```

- (void)toggleStateAtRow:(NSUInteger)row column:(NSUInteger)column {
    BOOL state = [self stateAtRow:row column:column];
    [self setState:!state atRow:row column:column];
}

```

要成为基于文档的应用，我们的文档类还需要实现最后两个用于读取和写入的方法。以前也提到过，我们不需要直接处理文件，甚至也不用担心之前传递给 `initWithFileURL:` 方法的 URL 参数。只需要实现两个方法：一个用于将文档的数据结构转换成 NSData 对象，以便存储；另一个获取最近加载的 NSData 对象，从中取出对象的数据结构。由于我们文档的内部结构已经包含在 NSMutableData 对象中了，而 NSMutableData 是 NSData 的子类，所以实现过程非常简单。在 @end 之前添加如下两个方法：

```

- (id)contentsForType:(NSString *)typeName error:(NSError **)outError {
    NSLog(@"saving document to URL %@", self.fileURL);
    return [self.bitmap copy];
}

- (BOOL)loadFromContents:(id)contents ofType:(NSString *)typeName
    error:(NSError **)outError {
    NSLog(@"loading document from URL %@", self.fileURL);
    self.bitmap = [contents mutableCopy];
    return true;
}

```

第一个方法 `contentsForType:error:` 在保存文档时调用。它返回位图数据的一份不可变副本，之后系统负责存储它。

当系统从存储区加载了数据，并且准备将这个数据提供给文档类的一个实例时，则会调用第二个方法 `loadFromContents:ofType:error:`。这里，我们只是获取一份传递给该方法的数据的不可变副本。我们还编写了一些日志语句，便于后续在 Xcode 日志中查看执行结果。

你可以在这些方法中完成一些我们在应用中忽略的工作。它们都提供了 `typeName` 参数，你的文档可以从不同类型的数据存储器中加载数据，或者向它们保存数据，该参数就用于区分这些数据存储的类型。这两个方法还有一个 `outError` 参数，如果向文档的内存数据结构保存数据或者加载数据时发生了错误，可以使用这个参数来指定。然而在本例中，我们所做的实在很简单，无需关心这些。

这些就是文档类所需的所有内容了。我们的文档类严格遵循 MVC 原则，是个完完全全的模型类，它对自己是如何被显示的一无所知。而且有了 `UIDocument` 父类，这个文档类甚至不必知道其存储方式的细节。

### 14.1.3 主控制器代码

现在我们已经完成了文档类，是时候开始编写用户在运行该应用时最先看到的视图了：一个现存 TinyPix 文档的列表，它由 `BIDMasterViewController` 类负责。我们需要让这个类知道如何获取可用文档的列表，让用户选择一个已存在的文档，然后创建并命名新的文档。当用户创建或者选择一个文档后，则将它传递给详细控制器以便显示。



```

    NSURL *directoryURL = urls[0];
    NSURL *fileURL = [directoryURL URLByAppendingPathComponent:filename];
    return fileURL;
}

```

第二个私有方法有些长。它也用到了 Documents 目录,这次是用于查找代表现存文档的文件。该方法获取它所找到的文件,并将它们根据创建时间来排序,以便用户可以以“博客风格”(blog-style)的顺序来查看文档列表(第一个文档是最新的)。文档文件名被存放在 documentFileNames 属性中,然后重新加载表视图(我们尚未处理)。在@end 前添加如下方法:

```

- (void)reloadFiles {
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
        NSUserDomainMask, YES);
    NSString *path = paths[0];
    NSFileManager *fm = [NSFileManager defaultManager];

    NSError *dirError;
    NSArray *files = [fm contentsOfDirectoryAtPath:path error:&dirError];
    if (!files) {
        NSLog(@"Error listing files in directory %@: %@",
            path, dirError);
    }
    NSLog(@"found files: %@", files);

    files = [files sortedArrayUsingComparator:
        ^NSComparisonResult(id filename1, id filename2) {
        NSDictionary *attr1 = [fm attributesOfItemAtPath:
            [path stringByAppendingPathComponent:filename1]
            error:nil];
        NSDictionary *attr2 = [fm attributesOfItemAtPath:
            [path stringByAppendingPathComponent:filename2]
            error:nil];
        return [attr2[NSFileCreationDate] compare: attr1[NSFileCreationDate]];
    }];
    self.documentFileNames = files;
    [self.tableView reloadData];
}

```

现在我们来添加一些表视图的数据源方法。你应该对此已经相当熟悉了,在@end 之前添加以下 3 个方法:

```

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
    return [self.documentFileNames count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        @"FileCell"];
}

```



```

NSString *path = self.documentFileNames[indexPath.row];
cell.textLabel.text = path.lastPathComponent.stringByDeletingPathExtension;
return cell;
}

```

这些方法都基于存储在 `documentFileNames` 属性中的数组内容。`tableView:cellForRowAtIndexPath:`方法依赖于一个标识符设为"FileCell"的表视图单元，所以我们必须保证之后在分镜中创建该单元。

现在只差分镜还没有设计，不然当前就可以运行至此所编写的代码并且查看其运行结果了。但是如果没有先前创建好的 TinyPix 文档，表视图就没有任何可以显示的内容。而且到目前为止，也还无法创建新文档。另外，我们也没有实现颜色选择控件（接下来就要处理这部分内容）。所以在运行该应用之前，我们先来实现这些内容。

用户选择的高亮色彩会立即对应用程序生效。这是 iOS 7 的新特性，可以让你立即定义应用程序的部分或全部高亮色彩。UIView 类拥有一个 `tintColor` 属性，如果设置了某个视图的色彩，这个值将会向下继承到所有子视图中。这意味着如果设置了应用程序 UIWindows（每个应用程序只有一个）的这个值，应用程序的所有子视图都将获得同样的值。

同时我们会将这个值存储在 `NSUserDefaults` 中以便之后获取。下面的操作方法实现了这项工作，它将分段控件选择的索引保存到 `NSUserDefaults`。此外还有一个用来设置顶层视图色彩的方法。在 `@end` 上方添加如下方法：

```

- (IBAction)chooseColor:(id)sender {
    NSInteger selectedColorIndex = [(UISegmentedControl *)sender
                                     selectedSegmentIndex];
    [self setTintColorForIndex:selectedColorIndex];

    NSUserDefaults *prefs = [NSUserDefaults standardUserDefaults];
    [prefs setInteger:selectedColorIndex forKey:@"selectedColorIndex"];
    [prefs synchronize];
}

- (void)setTintColorForIndex:(NSInteger)selectedColorIndex {
    switch (selectedColorIndex) {
        case 0:
            self.view.window.tintColor = [UIColor redColor];
            break;
        case 1:
            self.view.window.tintColor = [UIColor colorWithRed:0
                                                                green:0.6
                                                                blue:0
                                                                alpha:1];
            break;
        case 2:
            self.view.window.tintColor = [UIColor blueColor];
            break;
        default:
            break;
    }
}

```

我们还没有在分镜中创建分段控件，但很快就会进行这项工作。我们还需要一个方法用来确

保 GUI 中的分段控件被显示时, 应该显示 `NSUserDefaults` 中的当前色彩值。在 `viewDidAppear:` 方法中使用代码最为合适, 因为这个方法调用时, 视图已经加载到窗口里了。这样我们就可以访问顶层对象并设置其色彩:

```
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];

    NSUserDefaults *prefs = [NSUserDefaults standardUserDefaults];
    NSInteger selectedColorIndex = [prefs integerForKey:@"selectedColorIndex"];
    [self setTintColorForIndex:selectedColorIndex];
    [self.colorControl setSelectedSegmentIndex:selectedColorIndex];
}
```

现在, 创建一个新的 `viewDidLoad` 方法。首先调用父类的实现, 然后在导航栏的右侧添加一个按钮, 用户按下该按钮可创建新的 TinyPix 文档。在方法的最后, 调用之前实现的 `reloadFiles` 方法。对 `viewDidLoad:` 方法作如下修改:

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

    UIBarButtonItem *addButton = [[UIBarButtonItem alloc]
        initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
        target:self
        action:@selector(insertNewObject)];
    self.navigationItem.rightBarButtonItem = addButton;

    [self reloadFiles];
}
```

你可能已经注意到了, 在这个方法中创建 `UIBarButtonItem` 时, 我们在该按钮按下时告诉它调用 `insertNewObject` 方法。我们还没有编写该方法, 现在就来实现它。在 `@end` 前添加如下方法:

```
- (void)insertNewObject {
    // 获取文档名称
    UIAlertView *alert =
        [[UIAlertView alloc] initWithTitle:@"Filename"
        message:
            @"Enter a name for your new TinyPix document."
        delegate:self
        cancelButtonTitle:@"Cancel"
        otherButtonTitles:@"Create", nil];
    alert.alertViewStyle = UIAlertViewStylePlainTextInput;
    [alert show];
}
```

这个方法创建了一个包含文本输入字段的警告面板, 然后显示它。而创建新项的任务就交给了警告视图的委托方法 (当从警告视图退出时将会调用该方法), 现在我们就来实现它, 在 `@end` 前添加如下方法:

```
- (void)alertView:(UIAlertView *)alertView
didDismissWithButtonIndex:(NSInteger)buttonIndex {
    if (buttonIndex == 1) {
```

```

NSString *filename = [NSString stringWithFormat:@"%$.tinypix",
    [alertView textFieldAtIndex:0].text];
NSURL *saveUrl = [self urlForFilename:filename];
self.chosenDocument = [[BIDTinyPixDocument alloc]
    initWithFileURL:saveUrl];
[self.chosenDocument saveToURL:saveUrl
    forSaveOperation:UIDocumentSaveForCreating
    completionHandler:^(BOOL success) {
    if (success) {
        NSLog(@"save OK");
        [self reloadFiles];
        [self performSegueWithIdentifier:@"masterToDetail"
            sender:self];
    } else {
        NSLog(@"failed to save!");
    }
}];
}
}

```

14

这个方法前面部分相当简单。它检查 `buttonIndex` 来确定用户按下的是哪个按钮（0 代表 Cancel）。然后根据用户的输入创建一个文件名，根据该文件名创建一个 URL（使用之前我们编写的 `urlForFilename:` 方法），然后使用该 URL 创建一个新的 `BIDTinyPixDocument` 实例。

接下来的部分就有些复杂了。这里有一点非常重要：仅仅使用一个给定的 URL 创建新文档，并不会创建真正的文件。事实上，在调用 `initWithFileURL:` 方法时，文档并不知道这个给定的 URL 是指向一个已存在的文件，还是指向一个需要创建的新文件。所以我们必须告诉它应该怎么做。在本例中，我们通过以下代码告诉它使用这个给定的 URL 来保存一个新文件：

```

[self.chosenDocument saveToURL:saveUrl
    forSaveOperation:UIDocumentSaveForCreating
    completionHandler:^(BOOL success) {
    .
    .
    .
}];

```

我们来看一下作为最后一个参数传递给 `saveToURL:forSaveOperation:completionHandler:` 方法的代码块，它的目的和用法很有意思。这个方法没有提供返回值来告诉我们它是如何完成的，事实上，这个方法在调用后将会立即返回（远在文件实际保存之前），它先进行文件保存工作，完成后调用我们传递的代码块，使用 `success` 参数告诉我们是否成功。为了使它尽可能稳定地工作，文件保存实际上是在后台线程中执行的。而我们传递的代码块是在主线程中调用的，所以可以安全地使用需要在主线程中执行的资源，比如 `UIKit`。记住这些，然后再次看以下这段代码块的作用：

```

if (success) {
    NSLog(@"save OK");
    [self reloadFiles];
    [self performSegueWithIdentifier:@"masterToDetail" sender:self];
} else {
    NSLog(@"failed to save!");
}

```

这就是我们向文件保存方法传递的代码块内容，当文件操作完成后将会调用它。我们检查该操作是否成功，如果成功了，就立即重新加载这个文件，然后初始化一个转向另一个视图控制器的转场。之前没有过多涉及转场的这方面功能，但是这些内容相当简单。

分镜文件中的转场可以拥有一个标识符，和表视图单元一样，可以使用该标识符通过编程触发转场。在本例中，我们只要记得在分镜中配置该转场。在进行配置前，先在这个类中添加最后一个必需的方法，用来处理这个转场。在 @end 之前插入以下方法：

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if (sender == self) {
        // 如果 sender == self, 说明刚刚创建了一个新文档,
        // 并且 chosenDocument 属性已经被设置好了
        UIViewController *destination = segue.destinationViewController;
        if ([destination respondsToSelector:@selector(setDetailItem:)]) {
            [destination setValue:self.chosenDocument forKey:@"detailItem"];
        }
    } else {
        // 查找表视图中选中的文档
        NSIndexPath *indexPath = [self.tableView indexPathForSelectedRow];
        NSString *filename = self.documentFileNames[indexPath.row];
        NSURL *docUrl = [self urlForFilename:filename];
        self.chosenDocument = [[BIDTinyPixDocument alloc]
                               initWithFileURL:docUrl];
        [self.chosenDocument openWithCompletionHandler:^(BOOL success) {
            if (success) {
                NSLog(@"load OK");
                UIViewController *destination = segue.destinationViewController;
                if ([destination respondsToSelector:@selector(setDetailItem:)]) {
                    [destination setValue:self.chosenDocument
                                         forKey:@"detailItem"];
                }
            } else {
                NSLog(@"failed to load!");
            }
        }];
    }
}
```

这个方法有两条清晰的执行路径，由顶部的条件来决定。从第 10 章中对于分镜的讨论可知，一个新的控制器压入导航栈时，将会对视图控制器调用该方法。sender 参数指向初始化这个转场的对象，这里我们只用它来指出接着要做什么。如果转场是通过我们在警告视图委托方法中执行的方法调用来初始化的，那么 sender 参数为 self。在这种情况下，我们知道 chosenDocument 属性已经被设置好了，我们只要将它的值传递给目标视图控制器就可以了。

否则，我们知道用户触碰了表视图的某一行，需要对此作出响应，这种情况则稍微复杂一些。构建一个 URL（这与创建文档时所做的类似），创建文档类的一个新实例，然后尝试打开该文件。可以看到，用于打开文件的方法 openWithCompletionHandler：与之前用来保存文件的方法很相似，我们向它传递一个代码块，用于之后执行。和文件保存方法相同，加载过程发生在后台，而传入的代码块将会在加载完成后在主线程中执行。此时，如果加载成功，我们就将该文档传递给详情

视图控制器。

注意，这些方法都使用了键—值编码技术，我们在之前已经用过多次，通过这种技术，我们甚至不用包含转场的目标控制器的头文件，就能设置它的 `detailItem` 属性。这样就足够了，因为 `BIDDetailViewController`（作为 Xcode 项目的一部分创建的详情视图控制器类）正好包含了一个名为 `detailItem` 的属性，所以这样就可以了。

我们已经有了足够的代码，现在是时候来配置分镜了，以便能够运行应用进行测试。保存代码，然后继续。

#### 14.1.4 初始分镜

在 Xcode 项目导航面板中选择 `Main.storyboard`，首先看一下里面已经存在的内容。其中包括导航控制器、主视图控制器和详情视图控制器（参见图 14-2）。你可以完全忽略导航控制器，我们只会使用另外两个。

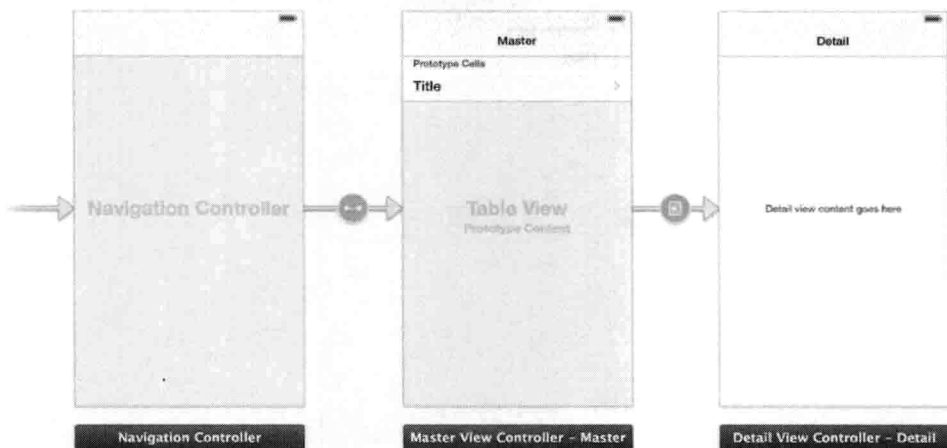


图 14-2 TinyPix 项目的分镜，显示了导航控制器、主视图控制器和详情视图控制器

首先我们来处理主视图控制器场景。我们在这里配置显示所有 TinyPix 文档的表视图。默认情况下，这个场景的表视图使用静态单元，而非动态单元（如果要复习这两种单元类型的区别，可以回顾第 10 章）。我们要让这个表视图从之前实现的数据源方法中获取内容，所以使用默认的设置就可以了。但我们需要配置单元原型，所以选中它并打开属性检查器。将单元的 `Identifier` 属性设置为 `FileCell`。这样，我们之前编写的数据源代码就可以访问表视图单元了。

还需要创建一个在代码中触发的转场。按住 `control` 键拖动鼠标，从 `master detail view controller` 的图标（场景底部的橙色圆圈或者是 dock 中的 `Master View Controller-Master` 图标）拖到 `detail view controller` 上，从分镜的转场菜单中选择 `Push`。

现在，你可以看到有两个转场关联了这两个场景。分别选择这两个转场，可以看看它们的起始场景。一个会高亮显示整个主场景，而另一个只会高亮显示表视图单元。选择那个高亮显示整

个场景的转场，使用属性检查器将它的 Identifier 设为 masterToDetail。

最后要为主视图控制器场景做的是：让用户选择一种颜色，用来在详情视图中代表一个“开启”的点。我们并不想实现某种复杂的颜色选取器，只是添加一个分段控件，让用户从预定义的颜色中进行选取。

从对象库中找到一个 Segmented Control，将它拖至主视图顶部的导航栏中（参见图 14-3）。

确保选中了分段控件，然后打开属性检查器。在检查器顶部的 Segmented Control 部分，使用 stepper 选项将 Segments 的数量从 2 改为 3。然后依次双击每个分段的标题，将它们分别改为 Red、Green 和 Blue。设置完这些标题后，调整分段控件的尺寸到合适的宽度。

接着，按下 control 键，从分段控件拖向代表主控制器的图标（控制器底部的橙色圆圈，或者 dock 中标为 Master View Controller - Master 的图标），选择 chooseColor: 方法。然后再次按下 control 键，从主视图控制器拖向分段控件，选择 colorControl 输出接口。

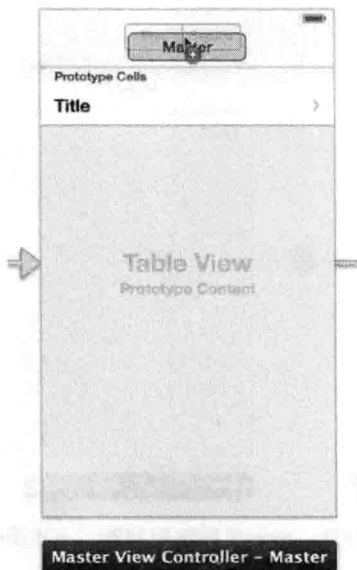


图 14-3 TinyPix 项目的分镜，显示了一个主视图控制器，在该控制器的导航栏中放置了一个分段控件

现在我们可以运行这个应用了，看看我们辛苦工作的成果！运行应用，它启动了，显示了一个空的表视图，视图顶部有一个分段控件，右上角有一个加号按钮（参见图 14-4）。

点击这个加号按钮，应用要求你为新文档输入名称。键入一个名字，然后按下 Create，现在应用切换到详情视图，好吧，它现在还在建设中。详情视图控制器所做的默认实现只是在一个标签中显示 detailItem 的描述。当然，在控制台窗格中还有更多信息。内容并不多，但也不错了！

点击返回按钮回到主列表，你可以在这里看到刚刚添加的项。继续添加一两个项，看看它们是否能正确地添加到列表。最后回到 Xcode，我们还有很多工作要做。



图 14-4 第一次启动后的 TinyPix 应用。单击加号图标添加一个新文档。应用会提示你为新文档输入名称。目前，详情视图所做的只是在标签中显示文档名

### 14.1.5 创建 BIDTinyPixView 类

接下来要创建一个视图类，用于显示一个用户可编辑的网格。在项目导航面板中选择 TinyPix 文件夹，按下 `command+N` 创建一个新文件。在 iOS 下面的 Cocoa Touch 部分中选择 Objective-C class，然后点击 Next。将新类命名为 BIDTinyPixView，在 Subclass of 弹出菜单中选择 UIView。点击 Next，确认保存位置正确，最后点击 Create。

**注意** 这个视图类的实现包括一些我们尚未提及的内容，比如绘图和触屏处理。这里暂时不讨论这些主题的细节，我们只是先向你展示这些代码。第 16 章会详细讨论使用 Core Graphics 绘图的相关话题，第 18 章中会讨论触屏和拖动。

选中 BIDTinyPixView.h，作如下修改：

```
#import <UIKit/UIKit.h>
@class BIDTinyPixDocument;

@interface BIDTinyPixView : UIView

@property (strong, nonatomic) BIDTinyPixDocument *document;

@end
```

这里我们只是添加了一个属性，这样控制器就可以传递文档。

现在，转向 BIDTinyPixView.m，这里要做大量的工作。首先在文件顶部添加一个类扩展：

```

#import "BIDTinyPixView.h"
#import "BIDTinyPixDocument.h"

typedef struct {
    NSUInteger row;
    NSUInteger column;
} GridIndex;

@interface BIDTinyPixView ()

@property (assign, nonatomic) CGSize blockSize;
@property (assign, nonatomic) CGSize gapSize;
@property (assign, nonatomic) GridIndex selectedBlockIndex;

@end

@implementation BIDTinyPixView
.
.
.

```

这里我们定义了一个名为 GridIndex 的 C 语言结构体，作为一种便捷方式来处理行/列对。我们还定义了一个类扩展，包含一些之后要用到的属性。

默认的空 UIView 子类包含一个 initWithFrame: 方法，它是 UIView 类的默认初始化方法。然而，由于该类将从分镜中加载，所以需要使用 initWithCoder: 方法来进行初始化。我们将同时实现这两个方法，在每个方法中都调用第三个方法来初始化属性。对 initWithFrame: 作如下修改，并在后面再添加一些代码：

```

- (id)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        // 初始化代码
        [self commonInit];
    }
    return self;
}

- (id)initWithCoder:(NSCoder *)aDecoder {
    self = [super initWithCoder:aDecoder];
    if (self) {
        [self commonInit];
    }
    return self;
}

- (void)commonInit{
    _blockSize = CGSizeMake(34, 34);
    _gapSize = CGSizeMake(5, 5);
    _selectedBlockIndex.row = NSNotFound;
    _selectedBlockIndex.column = NSNotFound;
}

```

\_blockSize 和 \_gapSize 值是根据横向 310 点数的视图来调整的。如果我们想在这里采用更为



灵活的方式，可以根据视图的实际框架动态定义它们的值，但这里采用了符合该示例的最简单的方式，所以我们就用这种方式。

现在，来看看绘图工作。我们重写了标准的 `UIView` 方法 `drawRect:`，用于绘制网格中的所有方格，并且为每一个方格调用另一个方法。添加如下粗体所示代码，不要忘了删除 `drawRect:` 方法周围的注释：

```
/*
// 除非是要进行自定义绘图，否则不要重写 drawRect:方法
// 即使是一个空的 drawRect:方法，也会影响动画性能
- (void)drawRect:(CGRect)rect
{
    // 绘图代码
    if (!_document) return;

    for (NSUInteger row = 0; row < 8; row++) {
        for (NSUInteger column = 0; column < 8; column++) {
            [self drawBlockAtRow:row column:column];
        }
    }
}
*/

- (void)drawBlockAtRow:(NSUInteger)row column:(NSUInteger)column {
    CGFloat startX = (_blockSize.width + _gapSize.width) * (7 - column) + 1;
    CGFloat startY = (_blockSize.height + _gapSize.height) * row + 1;
    CGRect blockFrame = CGRectMake(startX, startY,
                                   _blockSize.width, _blockSize.height);
    UIColor *color = [_document stateAtRow:row column:column] ?
        [UIColor blackColor] : [UIColor whiteColor];
    [color setFill];
    [self.tintColor setStroke];
    UIBezierPath *path = [UIBezierPath bezierPathWithRect:blockFrame];
    [path fill];
    [path stroke];
}
```

最后，我们添加一组响应用户触摸事件的方法。`touchesBegan:withEvent:`和 `touchesMoved:withEvent:`方法是所有 `UIView` 子类都能实现的标准方法，它们用于捕获视图框架中发生的触摸事件。这两个方法用到了之前我们在类扩展中定义的方法，根据触摸位置来计算网格位置，并切换文档中指定值的状态。在文件底部 `@end` 之前添加以下 4 个方法：

```
- (GridIndex)touchedGridIndexFromTouches:(NSSet *)touches {
    GridIndex result;
    UITouch *touch = [touches anyObject];
    CGPoint location = [touch locationInView:self];
    result.column = 8 - (location.x * 8.0 / self.bounds.size.width);
    result.row = location.y * 8.0 / self.bounds.size.height;
    return result;
}

- (void)toggleSelectedBlock {
    [_document toggleStateAtRow:_selectedBlockIndex.row
```

```

        column:_selectedBlockIndex.column];
[[_document.undoManager prepareWithInvocationTarget:_document]
 toggleStateAtRow:_selectedBlockIndex.row column:_selectedBlockIndex.column];
[self setNeedsDisplay];
}
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    self.selectedBlockIndex = [self touchedGridIndexFromTouches:touches];
    [self toggleSelectedBlock];
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    GridIndex touched = [self touchedGridIndexFromTouches:touches];
    if (touched.row != _selectedBlockIndex.row
        || touched.column != _selectedBlockIndex.column) {
        _selectedBlockIndex = touched;
        [self toggleSelectedBlock];
    }
}
}

```

细心的读者可能会注意到, toggleSelectedBlock 方法做了一些特别的工作。在调用文档类的 toggleStateAtRow:column: 方法改变特定网格点的值后, 它还做了一些其他工作。我们再来看一下:

```

- (void)toggleSelectedBlock {
    [_document toggleStateAtRow:_selectedBlockIndex.row
                          column:_selectedBlockIndex.column];
    [[_document.undoManager prepareWithInvocationTarget:_document]
     toggleStateAtRow:_selectedBlockIndex.row column:_selectedBlockIndex.column];
    [self setNeedsDisplay];
}

```

\_document.undoManager 的调用返回一个 NSUndoManager 实例。本书中我们还没有直接处理过这个类, NSUndoManager 在 iOS 和 Mac OS X 中都是撤销/重做功能的结构基础。它的思想是, 无论用户何时在 GUI 中执行一项操作, 开发者都可以使用 NSUndoManager 来“记录”一个用于撤销用户先前操作的方法调用, 从而保留某种路径导航。NSUndoManager 会将该方法存储在一个特殊的撤销栈 (undo stack) 中, 当用户激活系统的撤销功能时, 可以使用它回溯文档的状态。

它的工作原理是 prepareWithInvocationTarget: 方法返回一种特殊的委托对象, 你可以向它发送任何消息, 而这个消息会和目标一起被打包, 压入撤销栈。因此, 虽然看起来好像在一行里调用了两次 toggleStateAtRow:column: 方法, 实际上第二次没有调用, 仅是放入队列, 以备后用。这种出色的动态行为是 Objective-C 超越 C++ 等静态语言的一个方面, 静态语言不支持, 也几乎不可能将一个对象实现为另一个对象的代理, 或者将一个方法调用打包起来以供后续使用 (因此, 很多工作实现起来都相当麻烦, 如撤销功能)。

那么, 为什么这里我们要这么做呢? 至此, 我们还没有考虑过任何撤销/重做问题, 为什么现在提出来? 这是因为: 用文档的 undoManager 注册一个可被撤销的操作可以将该文档标记为“脏的”, 并且确保稍后会被自动保存。事实上, 支持用户撤销操作只能算锦上添花, 起码在这个应用中是这样。而在一个拥有较复杂文档结构的应用中, 支持文档层面的撤销是颇有益处的。

保存你所作的修改。现在视图类已经完成了, 我们回到分镜, 为详情视图配置 GUI。

### 14.1.6 设计分镜

选择 Main.storyboard，找到详情视图场景，首先来看看已有的内容。

GUI 只有一个标签（Detail view content goes here，查看详情视图内容），它就是之前运行应用时看到的包含文档描述的标签。这个标签没什么特别用处，所以在详情视图控制器中选择该标签，按下 delete 键删除它。

使用对象库找到一个 View，将它拖入详情视图中。界面构建器会将它填满整个视图区域。放好后，使用尺寸检查器将它的宽度和高度都设为 310。最后拖动该视图，使用引导线将它放置在视图中央（参见图 14-5）。



图 14-5 我们在详情视图使用一个 310 像素×310 像素的视图替换了标签，并将该视图放置在详情视图中央

现在转到身份检查器，在这里我们可以将 UIView 实例改为我们自定义类的实例。在检查器顶部的 Custom Class 部分，选择 Class 弹出列表，然后选择 BIDTinyPixView。

继续之前，需要调整这个新视图的约束。我们希望无论屏幕尺寸是多少，它都能够保持中央位置（即在 iPhone4 和 iPhone5 尺寸的屏幕上能正常运行），但我们也希望它能维持自身的尺寸。因此在视图仍然选中的时候，点击编辑区域底部的 Align 图标以显示相关选项。勾选 Horizontal Center in Container 和 Vertical Center in Container 的复选框，然后点击 Add 2 Constraints 按钮。之后点击编辑区域底部的 Pin 图标以调出它的控制器。点击 Width 和 Height 复选框，然后点击 Add 2 Constraints 按钮使其生效。

现在，我们需要将自定义视图和详情视图控制器关联起来。我们还没有为自定义视图准备输出接口，但是没问题，因为 Xcode 的拖动以生成代码（drag-to-code）功能可以帮我们轻松做到。

打开辅助编辑器，文本编辑器将会滑至 GUI 编辑器的旁边，显示 BIDDetailViewController.m 的内容。如果它显示的是其他内容，可以使用文本编辑器顶部的跳转栏将 BIDDetailViewController.m 显示出来。

要建立关联，按下 control 键将 Tiny Pix View 拖到代码中，在文件顶部的类扩展中 configureView 声明附近的位置释放鼠标，在出现的弹出窗口中，确保 Connection 设为 Outlet，并将这个新的输出接口命名为 pixView，然后点击 Connect 按钮。

现在应该看到 BIDDetailViewController.m 中增加了如下代码：

```
@property (weak, nonatomic) IBOutlet BIDDTinyPixView *pixView;
```

然而还有一项内容没有添加，那就是自定义视图类在源代码中的信息。我们需要在 BIDDetailView-Controller.m 的顶部添加如下粗体所示的代码：

```
#import "BIDDetailViewController.h"
#import "BIDDTinyPixView.h"

@interface BIDDetailViewController ()
```

现在修改 configureView 方法。它并非标准的 UIViewController 方法，只是项目模板包含在这个类中的一个私有方法，以便编写发生变化后更新视图的代码。因为我们不再使用描述标签，所以删除设置该标签的那行代码。然后再添加一些代码，用来将所选文档传递给我们的自定义视图，接着调用 setNeedsDisplay 方法通知它重绘自身。

```
- (void)configureView
{
    // 更新详情视图的用户界面

    if (self.detailItem) {
        self.detailDescriptionLabel.text = [self.detailItem description];
        self.pixView.document = self.detailItem;
        [self.pixView setNeedsDisplay];
    }
}
```

我们很快就能完成这个类，但还有一处需要修改。还记得之前提过的自动保存吗？文档被告知发生了一些修改后（通过注册可撤销操作来触发）会进行自动保存。保存工作通常在编辑后约 10 秒内发生。这和之前我们在本章讨论过的其他保存和加载过程相同，也是在后台线程中发生的，用户通常不会注意到。然而，这仅在文档还存在时才有效。

当前的设置存在一些风险，当用户按下返回按钮回到主列表时，文档实例将在没有进行任何保存操作的情况下被销毁，用户的最终更改也将丢失。为了确保不会发生这种情况，我们需要在 viewWillDisappear: 方法中添加一些代码，一旦用户离开详情视图，就关闭文档。关闭文档会导致该文档被自动保存。同样，保存工作发生在后台线程中。本例中我们不需要在保存完成后做任何事情，所以只要传递 nil 即可，不必传递代码块。

对 viewWillDisappear: 方法作如下修改：

```

- (void)viewWillDisappear:(BOOL)animated
{
    [super viewWillDisappear:animated];
    UIDocument *doc = self.detailItem;
    [doc closeWithCompletionHandler:nil];
}

```

现在，我们第一个真正的基于文档的应用就完成了。运行一下看看效果。你可以创建几个新文档，编辑它们，返回文档列表，选择另一个文档（或同一个文档），这些操作都可执行。如果你在测试应用时打开了 Xcode 控制台，那么可以看到每次加载或者保存文档时都会输出记录。使用自动保存体系无法直接控制保存工作发生的时刻（关闭文档除外），但看看这些日志，感受一下它们是何时发生的也挺有趣。

14

## 14.2 添加 iCloud 支持

现在有了一个完整的基于文档的应用，但我们不想止步于此。我们之前承诺过要在本章支持 iCloud，现在是时候履行诺言了。

要将 TinyPix 修改为支持 iCloud 相当简单。考虑到所有工作都在后台发生，要做的修改简直少得出乎意料。我们需要修改用于加载可用文件列表的方法，以及用于为加载新文件指定 URL 的方法，就是这些了。

除了代码修改，还需要处理一些额外的管理事项。只有应用嵌入了配置为允许使用 iCloud 的授权文件（provisioning profile），苹果公司才允许该应用向 iCloud 存储文件。这意味着，要让应用支持 iCloud，你必须是付费 iOS 开发者，并且安装了开发者证书。另外，该应用必须在真实设备上运行，而不是模拟器上。所以你必须至少有一台注册了 iCloud 的 iOS 设备，以运行新的支持 iCloud 的 TinyPix 应用。如果有两台设备，那就更有意思了，你可以看到一台设备上的更改如何传给另一台设备。

### 14.2.1 创建授权文件

首先，为 TinyPix 创建一个支持 iCloud 的授权文件。过去需要在苹果的开发者网站上执行大量复杂的步骤，但 Xcode 5 简化了这些操作。在项目导航栏的顶端选择 TinyPix 条目，然后点击编辑区域的 Capabilities 分页。你应该会看到如图 14-6 所示的内容。

图 14-6 中展示的功能列表都可以通过 Xcode 直接设定，不需要登录网站来创建和下载 provisioning 等文件。针对 TinyPix，我们想要启用 iCloud，即列表中第一个功能，因此点击云图标旁边的展开三角。这里你将看到一些关于此功能的信息。点击右侧的开关来启用它。这一步需要登录你的 Apple ID 账号，显然还要保证电脑是联网状态。启用之后，点击 key-value store 复选框以勾选它，如图 14-7 所示。

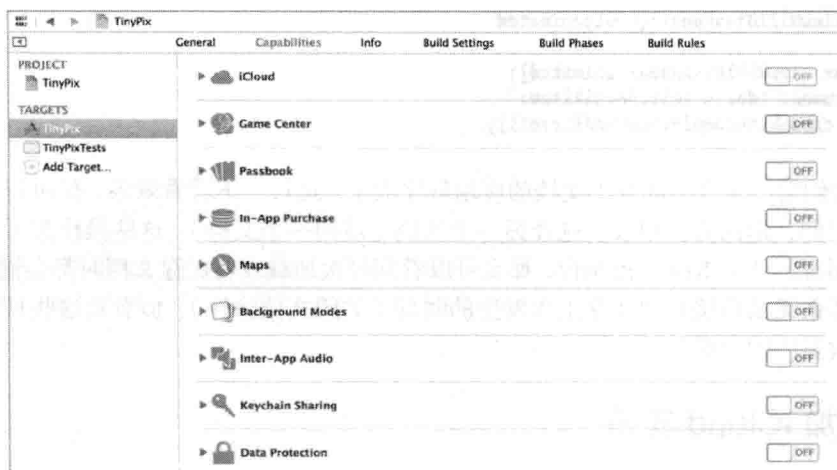


图 14-6 Xcode 5 可以设定的应用程序技术和服务界面

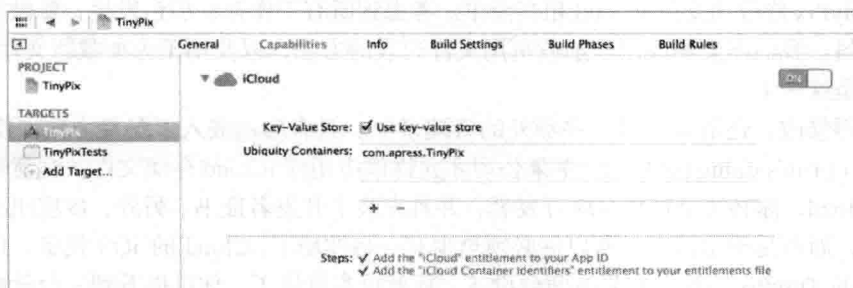


图 14-7 应用程序现在设定为启用了 iCloud。因为这个简单的配置让我们可以在这一章中少写几页的纸，这也许会拯救一两棵树的生命。感谢你，苹果！

你竟然做完了！你的应用程序现在拥有了通过代码访问 iCloud 的必要权限。接下来就是一些简单的编程。

## 14.2.2 如何查询

选择 BIDMasterViewController.m, 开始为 iCloud 功能修改代码。最大的一处改动是查询可用文档的方式。TinyPix 的第一个版本使用 `NSFileManager` 来查看本地文件系统中的可用文档。而这次，我们的做法有所不同，要使用一种特殊的查询方式来查找文档。

首先在类扩展中添加一对属性：一个用于保存指针，该指针指向当前正在进行的查询；另一个属性用于保存查询到的所有文档的列表。

```
@interface BIDMasterViewController () <UIAlertViewDelegate>
```

```
@property (weak, nonatomic) IBOutlet UISegmentedControl *colorControl;
```

```

@property (strong, nonatomic) NSArray *documentFileNames;
@property (strong, nonatomic) BIDTinyPixDocument *chosenDocument;
@property (strong, nonatomic) NSMetadataQuery *query;
@property (strong, nonatomic) NSMutableArray *documentURLs;
- (NSURL *urlForFilename:(NSString *)filename;
- (void)reloadFiles;
@end

```

接着是新的文件列表方法。删除整个 reloadFiles 方法，用以下方法替换：

```

- (void)reloadFiles {
    NSFileManager *fileManager = [NSFileManager defaultManager];
    // 在这里传入 nil 也是可以的，会匹配第一条授权
    NSURL *cloudURL = [fileManager URLForUbiquityContainerIdentifier:nil];
    NSLog(@"got cloudURL %@", cloudURL); // 在模拟器中返回 nil
    self.query = [[NSMetadataQuery alloc] init];
    _query.predicate = [NSPredicate predicateWithFormat:@"%K like '*.tinypix'",
                      NSMetadataItemFSNameKey];
    _query.searchScopes = [NSArray arrayWithObject:
                          NSMetadataQueryUbiquitousDocumentsScope];
    [[NSNotificationCenter defaultCenter]
     addObserver:self
     selector:@selector(updateUbiquitousDocuments:)
     name:NSMetadataQueryDidFinishGatheringNotification
     object:nil];
    [[NSNotificationCenter defaultCenter]
     addObserver:self
     selector:@selector(updateUbiquitousDocuments:)
     name:NSMetadataQueryDidUpdateNotification
     object:nil];
    [_query startQuery];
}

```

这里有一些新内容需要提一下。首先是下面这行：

```
NSURL *cloudURL = [fileManager URLForUbiquityContainerIdentifier:nil];
```

这个方法名真拗口。Ubiquity？这是什么？对于 iCloud 存储中的资源，苹果公司的术语都包括类似于 ubiquity 和 ubiquitous 这样的单词，用以表明这些资源“无处不在”，也就是说可以使用同一个 iCloud 登录凭证在任何设备上访问它们。

在本例中，我们向文件管理器请求一个基本的 URL，以便访问与特定的容器标识符（container identifier）相关的 iCloud 目录。容器标识符通常是包含你公司的唯一的包种子 ID（bundle seed ID）和应用程序标识符的字符串，容器标识符用于选取包含在应用中的一个 iCloud 授权。这里传递 nil 是一种快捷方式，意味着“给我列表中的第一项”。由于我们的应用只包含一项（前一节中所创建的），所以这完全符合我们的需要。

之后，我们创建并配置一个 NSMetadataQuery 的实例：

```

self.query = [[NSMetadataQuery alloc] init];
_query.predicate = [NSPredicate predicateWithFormat:@"%K like '*.tinypix'",
                  NSMetadataItemFSNameKey];
_query.searchScopes = [NSArray arrayWithObject:
                      NSMetadataQueryUbiquitousDocumentsScope];

```



这个类起初是用在 Mac OS X 上的 Spotlight 搜索工具上的,现在它还能让 iOS 应用查找 iCloud 目录。我们为这个查询设置了一个谓词,将搜索结果限制为仅包含那些正确的文件名类型,另外我们还指定了搜索范围,只在应用的 iCloud 存储中的 Documents 文件夹中进行查找。然后我们设置了一些通知,以便获知查询是何时完成的,这样我们就可以启动查询。

现在,我们需要实现当查询完成时的那些通知调用。在 reloadFiles 方法之后添加如下方法:

```
- (void)updateUbiquitousDocuments:(NSNotification *)notification {
    self.documentURLs = [NSMutableArray array];
    self.documentFileNames = [NSMutableArray array];

    NSLog(@"updateUbiquitousDocuments, results = %@", self.query.results);
    NSArray *results = [self.query.results sortedArrayUsingComparator:
        ^NSComparisonResult(id obj1, id obj2) {
            NSMetadataItem *item1 = obj1;
            NSMetadataItem *item2 = obj2;
            return [[item2 valueForKey:NSMetadataItemFSCreationDateKey]
                compare:
                [item1 valueForKey:NSMetadataItemFSCreationDateKey]];
        }];

    for (NSMetadataItem *item in results) {
        NSURL *url = [item valueForKey:NSMetadataItemURLKey];
        [self.documentURLs addObject:url];
        [(NSMutableArray *)_documentFileNames addObject:[url lastPathComponent]];
    }

    [self.tableView reloadData];
}
```

查询的结果包含一个 NSMetadataItem 对象的列表,从中我们可以获取文件 URL 和创建日期等数据项。我们根据创建日期来排列这些项,然后获取所有的 URL 以供之后使用。

### 14.2.3 保存在哪里

下一个要修改的是 urlForFilename:方法,这个方法也完全不同了。这里,我们使用一个普通的 URL 来为给定的文件名创建一个完整的 URL 路径。我们还在生成的路径中插入 "Documents", 以确保使用应用的 Documents 目录。删除原来的方法,用下面这个新的替换:

```
- (NSURL *)urlForFilename:(NSString *)filename {
    // 确保将"Documents"插入到路径中
    NSURL *baseURL = [[NSFileManager defaultManager]
        URLForUbiquityContainerIdentifier:nil];
    NSURL *pathURL = [baseURL URLByAppendingPathComponent:@"Documents"];
    NSURL *destinationURL = [pathURL URLByAppendingPathComponent:filename];
    return destinationURL;
}
```

现在,在真实的 iOS 设备上(不是模拟器)构建、运行该应用。如果你已经在这个设备上运行了老版本的应用,就会发现之前创建的 TinyPix 文档现在不见了。新版本忽略了应用的本地 Documents 目录,而完全依赖 iCloud。但你应该能够创建新的文档,在退出和重启应用后新文档



仍然存在。不仅如此,你甚至可以从设备上删除 TinyPix 应用,然后再次从 Xcode 运行它,你会发现所有保存在 iCloud 上的文档都立刻可用了。如果你还有采用同一个 iCloud 用户配置的 iOS 设备,使用 Xcode 在该设备上运行应用,可以看到同样的文档也出现在了这个设备上,这真是太好了!此外,你还能在 iOS 设备 Settings 应用的 iCloud 部分查找这些文档;如果运行的是 OS X 10.8 以上版本,你也能在你的 Mac 机的 System Preferences 应用的 iCloud 部分查找这些文档。

### 14.2.4 将首选项保存到 iCloud

只要再稍微花点儿工夫,就能实现另一项 iCloud 功能了。iOS 5 包含了一个新类 `NSUbiquitousKeyValueStore`,它非常类似于 `NSDictionary` (或者 `NSUserDefaults`),不同的是 `NSUbiquitousKeyValueStore` 的键-值都存放在“云”上。对应用程序的首选项、登录令牌,以及其他任何不属于同一个文档,但需要在用户的所有设备上共享的数据来说,`NSUbiquitousKeyValueStore` 非常实用。

在 TinyPix 项目中,我们将使用这项功能来保存用户首选的高亮颜色。这样的话,用户就不需要在每个设备上进行了配置,只要设置一次,就可以在所有设备上出现了。

选中 `BIDMasterViewController.m`,进行一些小修改。首先找到 `chooseColor:` 方法,作如下修改:

```
- (IBAction)chooseColor:(id)sender {
    NSInteger selectedColorIndex = [(UISegmentedControl *)sender selectedSegmentIndex];
    —NSUserDefaults *prefs = [NSUserDefaults standardUserDefaults];
    —[prefs setInteger:selectedColorIndex forKey:@"selectedColorIndex"];
    —[prefs synchronize];
    NSUbiquitousKeyValueStore *prefs = [NSUbiquitousKeyValueStore defaultStore];
    [prefs setLongLong:selectedColorIndex forKey:@"selectedColorIndex"];
}
```

这里,我们获取了一个与 `NSUserDefaults` 稍有不同的对象。这个新类没有 `setInteger:` 方法,我们使用了 `setLongLong:` 方法,它的作用与 `setInteger:` 方法相同。

然后找到 `viewDidAppear:` 方法,作如下修改:

```
- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    —NSUserDefaults *prefs = [NSUserDefaults standardUserDefaults];
    —NSInteger selectedColorIndex = [prefs integerForKey:@"selectedColorIndex"];
    NSUbiquitousKeyValueStore *prefs = [NSUbiquitousKeyValueStore defaultStore];
    NSInteger selectedColorIndex = (int)[prefs longLongForKey:
        @"selectedColorIndex"];
    [self setTintColorForIndex:selectedColorIndex];
    [self.colorControl setSelectedSegmentIndex:selectedColorIndex];
}
```

完成!现在,你可以在配置了同一个 iCloud 用户的多个设备上运行这个应用了,可以看到,在一个设备上设置的颜色将会出现在另一个设备上(下一次在该设备上打开文件时就能看到)。小菜一碟!

## 14.3 小结

现在,我们已经掌握了创建、运行一个支持 iCloud、基于文档的应用程序的基本知识,但还有更多需要考虑的问题。本书中我们并不打算讨论这些主题,不过如果真想开发一个出色的基于 iCloud 的应用,应该注意以下问题。

- ❑ 存储在 iCloud 中的文档有可能会发生冲突。在多个设备上同时编辑一个 TinyPix 文件会怎么样? 很幸运,苹果公司已经考虑到了这个问题,而且提供了一些方式来处理应用中的冲突。是要忽略冲突,还是尝试自动修正它们,或者让用户挑出这个问题,这些完全取决于你。详细内容可在 Xcode 文档查看器中搜索标题为“resolving document version conflicts”的文档。
  - ❑ 苹果公司建议将应用设计为能够在完全离线的模式下运行,以防用户因为某种原因不使用 iCloud。同时苹果公司也建议你为用户提供一种在 iCloud 存储和本地存储之间移动文件的方式。不幸的是,苹果公司并没有提供或者建议任何标准的 GUI 来帮助用户对此进行管理。而目前提供了这项功能的应用,比如苹果公司的 iWork 应用,在处理这个问题上看起来并没有做到很好的用户体验。更多内容可以查阅苹果公司 Xcode 文档中的“Managing the Life Cycle of a Document”。
  - ❑ 苹果公司支持为 Core Data 存储使用 iCloud,甚至提供了一个 `UIManagedDocument` 类,想使用这项功能,可以继承该类。想要了解更多信息,可以查看 `UIManagedDocument` 类,或者 Kevin Kim、Alex Horowitz、Dave Mark 和 Jeff LaMarche 合著的 *More iOS 7 Development: Further Explorations of the iOS SDK* (Apress 2014),该书是构建支持 iCloud 的 Core Data 应用的实用指南。我们需要指出这个架构比普通的 iCloud 文档存储更加复杂和难以掌握。苹果在 iOS 7 中改善了一部分,但仍然不是最完美的,使用时请三思而行。
- 你问接下来的内容? 第 15 章要讨论如何让应用多线程、多任务环境中正确运行。

# Grand Central Dispatch 和后台处理

尝试过多线程编程的读者（无论是在何种环境中）很可能体验过恐惧、惊骇或者更糟的感觉。所幸，技术不断在发展，最近苹果公司推出了一种新方法，大大简化了多线程编程。此方法称为 Grand Central Dispatch，本章就来学习使用它。我们还将深入了解 iOS 的多任务功能，介绍如何调整应用来利用这些新功能进一步完善应用。

## 15.1 Grand Central Dispatch

开发人员如今面临的一个最大的挑战是编写这样的软件：它可以执行复杂的操作来响应用户输入，同时保持迅速响应，确保用户不会在处理器执行某些后台任务时长时间等待。回想一下就会发现，这一挑战一直以来都伴随着我们，即使计算技术的进步使 CPU 越来越快，这一问题也始终存在。想要证据吗？只要看看眼前的计算机屏幕就行了。很可能就在上次你使用计算机时，你的工作流就被一个不断旋转的光标（表示系统繁忙）或者其他事件中中断过。

既然系统体系结构不断在发展，为什么这一问题还一直困扰着我们呢？部分原因是软件的典型编写方式——软件编写为一个按顺序执行的事件序列。这种软件可能随着 CPU 速度的提高而相应地变快，但这种改善只能是在一定程度上的。只要程序开始等待外部资源（比如文件或网络连接），整个事件序列都会暂停。所有现代操作系统目前都支持在一个程序中使用多个执行线程，因此，即使一个线程在等待特定的事件，其他线程仍然可以继续运行。即便如此，许多开发人员仍然将多线程编程视为某种歪门邪道而不屑使用。

开发人员希望不要花太多工夫熟悉系统的线程层就能将代码分解为并发执行的程序块（block），而苹果公司带来了一个好消息。这个好消息就是 GCD（Grand Central Dispatch），它提供了一套全新的 API，可以将应用需要执行的工作拆分为可分散在多个线程和多个 CPU（使用合适的硬件）上的更小的块。

这个新 API 的大部分可以使用程序块访问，程序块是苹果公司的另一项创新，它向 C 和 Objective-C 添加了某种简单的内联函数功能。程序块与 Ruby 和 Lisp 等语言中的类似功能具有很多相同点，它们可以提供有趣的新方式，在不同对象之间建立交互性，同时确保相关代码紧密地结合在方法中。

## 15.2 SlowWorker 简介

作为演示 GCD 工作原理的平台，我们将创建一个名为 SlowWorker 的简单应用，它有个简单的界面，包含一个按钮和一个文本视图。单击按钮会立即启动一个同步任务，将应用锁定 10 秒。任务完成后，会在文本视图中显示一些文本（参见图 15-1）。

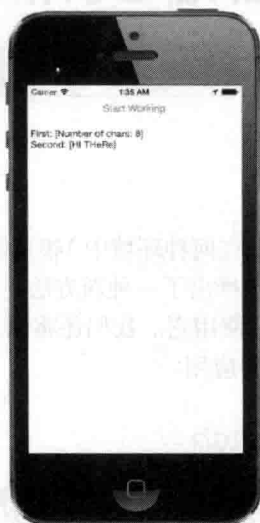


图 15-1 SlowWorker 应用将其界面隐藏在一个按钮之后。点击按钮，应用便执行工作，界面就会挂起大约 10 秒

首先像以前一样，在 Xcode 中使用 Single View Application 构建一个新应用，将它命名为 SlowWorker，将 Devices 设置为 iPhone，点击 Next 按钮存储项目。向 BIDViewController.m 中添加以下代码：

```
#import "BIDViewController.h"

@interface BIDViewController ()

@property (weak, nonatomic) IBOutlet UIButton *startButton;
@property (weak, nonatomic) IBOutlet UITextView *resultsTextView;

@end
```

这段代码定义了 GUI 上可见的两个对象的输出接口。

现在继续在 BIDViewController.m 的 @implementation 部分中添加以下粗体显示代码：

```
@implementation BIDViewController

- (NSString *)fetchSomethingFromServer
{
    [NSThread sleepForTimeInterval:1];
```

```

    return @"Hi there";
}

- (NSString *)processData:(NSString *)data
{
    [NSThread sleepForTimeInterval:2];
    return [data uppercaseString];
}

- (NSString *)calculateFirstResult:(NSString *)data
{
    [NSThread sleepForTimeInterval:3];
    return [NSString stringWithFormat:@"Number of chars: %d",
        [data length]];
}

- (NSString *)calculateSecondResult:(NSString *)data
{
    [NSThread sleepForTimeInterval:4];
    return [data stringByReplacingOccurrencesOfString:@"E"
        withString:@"e"];
}

- (IBAction)doWork:(id)sender
{
    NSDate *startTime = [NSDate date];
    NSString *fetchedData = [self fetchSomethingFromServer];
    NSString *processedData = [self processData:fetchedData];
    NSString *firstResult = [self calculateFirstResult:processedData];
    NSString *secondResult = [self calculateSecondResult:processedData];
    NSString *resultsSummary = [NSString stringWithFormat:
        @"First: %@\nSecond: %@", firstResult,
        secondResult];
    self.resultsTextView.text = resultsSummary;
    NSDate *endTime = [NSDate date];
    NSLog(@"Completed in %f seconds",
        [endTime timeIntervalSinceDate:startTime]);
}

```

如你所见，这个类的工作被拆分为许多小代码块。此代码只是为了模拟一些较慢的活动，这些方法不会真正执行任何耗时的操作。为了增添一些趣味，每个方法包含对 `NSThread` 中的 `sleepForTimeInterval:` 类方法的一次调用，这会使程序（具体来讲是调用该方法的线程）“暂停”几秒，不执行任何操作。`doWork:` 方法还在开头和末尾包含了一些代码，计算完成所有工作所花的时间。

现在打开 `Main.storyboard`，将一个 `Button` 和 `Text View` 控件拖到空白的 `View` 窗口中，按照图 15-2 所示布置这些控件。按住 `control` 拖动 `View Controller` 图标，将视图控制器的两个输出接口关联到按钮和文本视图。

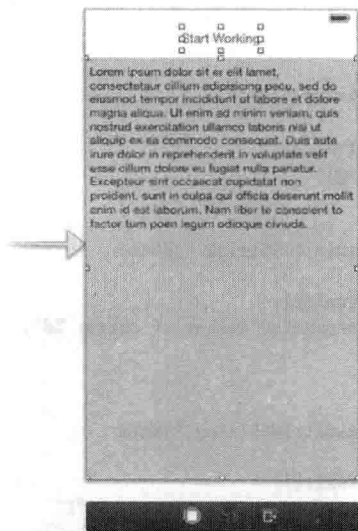


图 15-2 SlowWorker 界面包含一个按钮和一个文本视图。确保取消了文本视图的 Editable 复选框并删除了它的所有文本

接下来, 选择按钮, 转到关联检查器以将按钮的 Touch Up Inside 事件连接到 View Controller, 选择视图控制器的 doWork: 方法。最后, 选择文本视图, 使用属性检查器取消选择 Editable 复选框 (位于右上角), 从文本框删除默认文本。

现在保存工作, 点击 Run。应用应该启动, 按下按钮将使它运行大约 10 秒 (所有这些休眠时间量的总和), 然后显示结果。在等待期间, 可以看到 Start Working! 按钮颜色变淡, 只有在“工作”完成后才转为正常的颜色。另外, 在“工作”完成之前, 应用的视图无法响应。点击屏幕上的任何位置都没有反应。事实上, 在此期间与应用交互的唯一方式是点击主屏幕按钮从它切换出来, 而这正是我们希望避免的事件状态。

在这个例子中, 等待时间还可以忍受, 因为应用似乎仅会挂起几秒钟, 但如果应用经常以这种方式长时间“挂起”, 会让人相当郁闷。在最坏的情况下, 如果应用太长时间未响应, 那么操作系统可能会结束它。无论如何, 最终都会导致用户不爽, 甚至可能吓跑用户!

### 15.3 线程基础知识

在开始实现解决方案之前, 我们先介绍一下并发 (concurrency) 的一些基础知识。这里不会对 iOS 中的线程或一般线程知识做完整介绍, 只介绍理解本章中的具体操作所需的知识。

大部分现代操作系统 (当然包括 iOS) 都支持执行线程的概念。每个进程可以包含多个线程, 它们可以同时运行。如果只有一个处理器核心, 操作系统将在所有执行线程之间切换, 非常类似于在所有执行进程之间切换。如果拥有多个核心, 线程将像进程一样, 分散到几个核心上去执行。

一个进程中的所有线程共享可执行程序代码和全局数据。每个线程也可以拥有一些独有的数

据。线程可以使用一种称为互斥量（mutex）或锁的特殊结构，这种结构可以确保特定的代码块无法一次被多个线程运行。在多个线程同时访问相同数据时，这有助于保证正确的结果，在一个线程更新某个值（在代码中称为临界区）时锁定其他线程。

处理线程的过程中我们通常会关注线程安全（thread-safe）问题。一些软件库在编写时考虑了线程并发性，并使用互斥量恰当地保护它们的所有临界区。也有一些代码库不是线程安全的。

举例来说，在 Cocoa Touch 中，Foundation 框架（包含适用于所有 Objective-C 编程类型的基本类，如 NSString、NSArray 等）通常被视为是线程安全的。但是，UIKit 框架（包含专门用于构建 GUI 应用的类，如 UIApplication、UIView 及其所有子类等）在很大程度上被视为非线程安全的。这意味着在一个运行的 iOS 应用中，处理任何 UIKit 对象的所有方法调用都应从相同线程执行，该线程通常称为主线程（main thread）。如果从另一个线程访问 UIKit 对象，那结果就不堪设想了！你还可能会遇到一些莫名其妙的 bug，甚至更糟的是，你自己不会遇到任何问题，但发布之后一些用户却遭殃了。

默认情况下，主线程执行 iOS 应用的所有操作（比如处理由用户事件触发的操作），所以对于简单应用，没有什么需要担心的。用户触发的动作方法已在主线程中运行。本书到目前为止，代码全部在主线程上运行，但情况很快就会不一样了。

---

**提示** 有许多关于线程安全的著作，你有必要花时间深入理解和掌握相关知识。可以先看看苹果公司的相关文档，花几分钟阅读一下此页面，绝对有帮助：<http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/Multithreading/ThreadSafetySummary/ThreadSafetySummary.html>。

---

## 15.4 工作单元

一般程序员都会遇到前面介绍的线程模型问题，编写没有错误的多线程代码几乎是不可能的。这里并不是在对行业或普通程序员的能力进行批判，只是在说一种客观现象。在跨多个线程同步数据和操作时，需要在代码中考虑的复杂交互确实太多了，大部分人都无法应付。假设 5% 的人能够编写软件，那么其中只会有一小部分人真正能胜任大型多线程应用的编写任务。甚至成功处理了多线程问题的人通常也会建议其他人不要这样做！

幸好，还有一线希望。因为我们现在无需做太多底层线程编码就可以实现一定的并发性。这就像无需直接将每个比特放入视频 RAM 即可在屏幕上显示数据，无需直接与磁盘控制器交互即可从磁盘读取数据，我们也可以使用软件抽象确保我们无需直接对线程执行太多处理即可在多个线程上运行代码。

苹果公司推荐使用的解决方案体现了这样一种理念：将长期运行的任务拆分为多个工作单元，并将这些单元添加到执行队列中。系统会为我们管理这些队列，为我们在多个线程上执行工作单元。我们不需要直接启动和管理后台线程，而可以从通常实现多线程应用所涉及的太多“登记”工作中脱离出来，系统会为我们完成这些工作。



## 15.5 GCD：底层队列

这种将工作单元放到可在后台执行的队列中，以及让系统管理线程的理念确实很强大，而且显著简化了许多需要并发性的开发工作。在几年前的 OS X 中，GCD 开始展露锋芒，提供了执行此任务的基础架构。之后几年，iOS 平台也引入了 GCD。此技术不仅适用于 Objective-C，也适用于 C 和 C++。

GCD 在 C 接口中添加了一些优秀的概念，比如工作单元、无痛后台处理 (painless background processing)、自动线程管理，它们可在所有基于 C 的语言中使用。为了进一步完善，苹果公司开源了 GCD 的实现，所以它也可以移植到其他类 Unix 操作系统。

GCD 的一个重要概念是队列。系统提供了许多预定义的队列，包括可以保证始终在主线程上执行其工作的队列。它非常适合非线性安全的 UIKit。开发人员也可以创建自己的队列，按照自己的需求创建任意多个。GCD 队列严格遵循 FIFO（先进先出）原则。添加到 GCD 队列的工作单元将始终按照加入队列的顺序启动。这样看来，它们不会总是按相同顺序完成，因为如果可能，GCD 队列将自动在多个线程之间分配工作。

GCD 能够访问一个线程池，该线程池可在应用整个生命周期内重用。GCD 会尽量维护一些适合机器体系结构的线程。在有工作需要处理时，它将自动利用更多的处理器核心，以充分利用更强大的机器性能。以前的 iOS 设备都是单核的，所以线程池的用处不大，不过最近几年发布的所有 iOS 设备都采用了双核处理器，GCD 终于可以大显身手了！

### 15.5.1 傻瓜式操作

除了 GCD，苹果公司还向 C 语言本身（以及 Objective-C 和 C++）添加了一点儿新语法，以实现程序块的语言功能（在其他一些语言中也称为闭包或 lambda），这对于尽量充分利用 GCD 非常重要。程序块的理念是像任何其他 C 语言类型一样对待特定的代码块。程序块可以分配给一个变量，以参数的形式传递给函数或方法，当然也可以执行（不同于其他大部分类型）。通过这种方式，程序块可替代 Objective-C 中的委托模式或 C 中的回调函数。

跟方法或函数很像，程序块可以接受一个或多个参数并指定一个返回值。要声明程序块变量，可以使用“^”符号以及其他一些放在圆括号内的代码来声明参数和返回类型。要定义程序块本身，执行的操作大体相同，但要在后面添加定义程序块的实际代码（包含在花括号中）。

```
// 声明一个块变量 loggerBlock,
```

```
// 这个块没有参数也没有返回值
```

```
void (^loggerBlock)(void);
```

```
// 将一个块赋给上面声明的变量
```

```
// 没有参数和返回值的块（比如这个）
```

```
// 不需要在变量声明之前使用 void 对返回值 进行修饰
```

```
loggerBlock = ^{ NSLog(@"I'm just glad they didn't call it a lambda"); };
```

```
// 执行这个块，跟函数调用一样
```

```
loggerBlock(); // 这会在控制台上输出一些内容
```



如果进行过大量 C 编程，可能会发现这段代码类似于 C 中的函数指针概念。但是，它们之间存在一些重要区别。或许最大的区别（也是最明显的区别）是，程序块可以在代码内以内联方式定义，可以在代码块被传递给另一个方法或函数时再定义它。

另一个比较大的区别是，程序块可以访问在创建它的范围内所有可用的变量。默认情况下，程序块通过这种方式“捕获”了你访问的任何变量，将值复制到一个新的同名变量中，保留原始变量不变。Objective-C 对象会自动在复制基本类型值（如 `int` 和 `float`）时发送一个 `retain` 消息（之后代码块结束时，会向里面的 `strong` 类型变量发送 `release` 消息）。然而也可以在变量声明之前添加存储修饰符 `__block`，进行外部变量“读/写”。请注意 `block` 前面有两条下划线，而不只是一条。或者如果你想要传递一个 `weak` 类型的对象指针，你可以加上 `__weak` 前缀：

```
// 定义一个能够被块访问并修改的变量
__block int a = 0;

// 定义一个块，这个块会修改它作用域中的一个变量
void (^sillyBlock)(void) = ^{ a = 47; };

// 对块进行调用之前，先检查变量的值
NSLog(@"a == %d", a); // 这里会输出"a == 0"

// 执行块
sillyBlock();

// 调用块之后再次检查变量的值
NSLog(@"a == %d", a); // 这里会输出"a == 47"
```

之前提到，程序块在与 GCD 结合使用时才真正发挥作用，有了程序块，只需一步就可以将它添加到队列中。如果在定义块后立即将它添加到队列，而不是在块存储到变量之后，就多了个优势：能在使用代码的上下文中看到相关代码。

## 15.5.2 改进 SlowWorker

为了查看这是如何实现的，我们回头看一下 `SlowWorker` 的 `doWork:` 方法。它目前跟下面的代码差不多：

```
- (IBAction)doWork:(id)sender
{
    NSDate *startTime = [NSDate date];
    NSString *fetchedData = [self fetchSomethingFromServer];
    NSString *processedData = [self processData:fetchedData];
    NSString *firstResult = [self calculateFirstResult:processedData];
    NSString *secondResult = [self calculateSecondResult:processedData];
    NSString *resultsSummary = [NSString stringWithFormat:
        @"First: [%@]\nSecond: [%@]", firstResult,
        secondResult];
    self.resultsTextView.text = resultsSummary;
    NSDate *endTime = [NSDate date];
    NSLog(@"Completed in %f seconds",
        [endTime timeIntervalSinceDate:startTime]);
}
```

我们可以让该方法完全在后台运行,只需将所有代码包装在一个程序块中并将它传递给一个名为 `dispatch_async` 的 GCD 函数。此函数接受两个参数:一个 GCD 队列和一个分配给该队列的程序块。对 `doWork` 的副本进行以下更改,一定不要忘记在方法最后加上大括号和圆括号。

```
- (IBAction)doWork:(id)sender
{
    NSDate *startTime = [NSDate date];
    dispatch_queue_t queue =
        dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_async(queue, ^{
        NSString *fetchedData = [self fetchSomethingFromServer];
        NSString *processedData = [self processData:fetchedData];
        NSString *firstResult = [self calculateFirstResult:processedData];
        NSString *secondResult = [self calculateSecondResult:processedData];
        NSString *resultsSummary = [NSString stringWithFormat:
            @"First: %@\nSecond: %@", firstResult,
            secondResult];
        self.resultsTextView.text = resultsSummary;
        NSDate *endTime = [NSDate date];
        NSLog(@"Completed in %f seconds",
            [endTime timeIntervalSinceDate:startTime]);
    });
}
```

第一行使用 `dispatch_get_global_queue()` 函数,抓取一个已经存在并始终可用的全局队列。该函数接受两个参数:第一个指定优先级;第二个目前未使用,应该始终为 0。如果在第一个参数中指定了不同的优先级,比如 `DISPATCH_QUEUE_PRIORITY_HIGH` 或 `DISPATCH_QUEUE_PRIORITY_LOW`,那么实际上会得到一个不同的全局队列,系统将对该队列分配不同的优先级。目前,我们仍然使用默认的全局队列。

然后将该队列以及它后面的代码块一起传递给 `dispatch_async()` 函数,GCD 然后获取整个程序块,并将它传递给一个后台线程,程序块将在这里一次执行一步,就像在主线程中运行一样。

注意,我们在程序块创建之前定义了名为 `startTime` 的变量,而后在程序块最后使用了它的值。这看起来似乎没什么特别的意义,但因为程序块被执行时 `doWork` 方法已经退出了,所以 `startTime` 变量所指向的 `NSDate` 实例应该已经释放了!这是程序块用法的关键点:如果一个程序块在执行过程中访问任何“外部”变量,那么当该程序块被创建时,会进行一些特殊的设置工作,以允许程序块访问那些变量。这些变量所包含的值要么被复制(如果是普通的 C 类型变量,如 `int` 或 `float`),要么被保存(如果是指向对象的指针变量),这样它们所包含的值就可以在程序块内部使用了。当在 `doWork` 的第 2 行调用 `dispatch_async`,而代码中所示的程序块被创建后,`startTime` 实际上就发送了一条 `retain` 消息,其返回值赋给了程序块内部的一个同名(`startTime`)的新的不可变变量。

程序块内部的 `startTime` 变量必须是不可变的,这样程序块内部的代码就不会意外地与外部定义的变量混淆了。否则,只会使所有人困惑。然而有的时候,你确实想让一个程序块向一个外部定义的变量写入数据,这时 `_block` 存储修饰符(之前提到过)就派上用场了。如果使用 `_block` 定义一个变量,那么任何在相同作用域内定义的程序块都能直接访问它。一个有趣的副作用是:

`_block` 修饰的变量在程序块中使用不会被复制或保留。

### 1. 不要忘记主线程

这里有一个问题, 即 UIKit 的线程安全性。请记住, 从后台线程向任何 GUI 对象 (包括我们的 `resultsTextView`) 发送消息都是不可能的。幸好, GCD 也提供了一种方式来处理此问题。在程序块内部, 可以调用另一个分派函数, 将工作传回主线程! 为此, 可以再次调用 `dispatch_async()`, 这一次传入 `dispatch_get_main_queue()` 函数返回的队列, 该函数总是提供主线程上的特定队列, 并准备执行需要使用主线程的程序块。对你的 `doWork`: 再进行一项更改:

```
- (IBAction)doWork:(id)sender
{
    NSDate *startTime = [NSDate date];
    dispatch_queue_t queue =
        dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_async(queue, ^{
        NSString *fetchedData = [self fetchSomethingFromServer];
        NSString *processedData = [self processData:fetchedData];
        NSString *firstResult = [self calculateFirstResult:processedData];
        NSString *secondResult = [self calculateSecondResult:processedData];
        NSString *resultsSummary = [NSString stringWithFormat:
            @"First: [%@]\nSecond: [%@]", firstResult,
            secondResult];

        dispatch_async(dispatch_get_main_queue(), ^{
            self.resultsTextView.text = resultsSummary;
        });
        NSDate *endTime = [NSDate date];
        NSLog(@"Completed in %f seconds",
            [endTime timeIntervalSinceDate:startTime]);
    });
}
```

### 2. 提供反馈

如果现在构建并运行应用, 会看到它似乎能更加流畅地运行, 至少在一定程度上是这样。按钮在触摸之后不再突出显示, 这样可能导致用户不断地重复点击按钮。检查 Xcode 的控制台日志, 会看到每次点击的结果, 但只有最后一次点击的结果会在文本视图中显示。

我们真正希望做的是改进 GUI, 以便在用户按下按钮时, 显示界面会立即更新, 表明一个操作正在运行, 我们还想要该按钮在此过程中被禁用。为此, 我们将向显示界面添加 `UIActivityIndicatorView`。此类提供了在许多应用和网站上看到过的“旋转指示器”(spinner)。首先在 `BIDViewController.m` 文件顶部的类扩展中声明它:

```
@interface BIDViewController ()

@property (weak, nonatomic) IBOutlet UIButton *startButton;
@property (weak, nonatomic) IBOutlet UITextView *resultsTextView;
@property (weak, nonatomic) IBOutlet UIActivityIndicatorView *spinner;

@end
```

然后打开 `Main.storyboard` 文件, 在库中找到一个 Activity Indicator View, 并将它拖到视图中的按钮旁边 (参见图 15-3)。

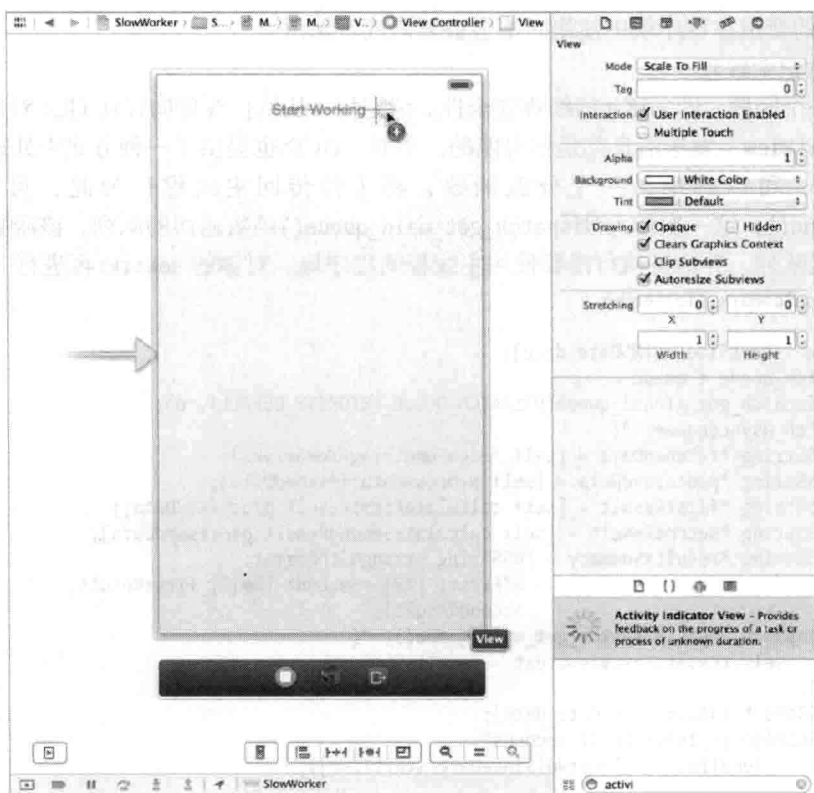


图 15-3 在界面构建器中将活动指示器视图拖到主视图中

选中活动旋转指示器，在属性检查器中勾选 **Hide When Stopped** 复选框，这样旋转指示器仅在我们告诉它开始旋转时才出现（没人希望自己 GUI 中有一个不会旋转的旋转指示器）。

接下来，按住 **control** 键从 **View Controller** 图标拖到旋转指示器，将旋转指示器关联到输出口，保存更改。

打开 **BIDViewController.m**。这里我们首先处理 **doWork:** 方法，添加一些代码，管理在用户单击和工作完成时按钮和旋转指示器的外观。首先将按钮的 **enabled** 属性设置为 **NO**，这会阻止它注册任何点击操作并通过让其文字变成灰色和透明使按钮呈现出被禁用状态。接下来我们让旋转指示器转动。

```

- (IBAction)doWork:(id)sender
{
    NSDate *startTime = [NSDate date];
    self.startButton.enabled = NO;

    [self.spinner startAnimating];
    dispatch_queue_t queue =
        dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

```

```

dispatch_async(queue, ^{
    NSString *fetchedData = [self fetchSomethingFromServer];
    NSString *processedData = [self processData:fetchedData];
    NSString *firstResult = [self calculateFirstResult:processedData];
    NSString *secondResult = [self calculateSecondResult:processedData];
    NSString *resultsSummary = [NSString stringWithFormat:
        @"First: [%@]\nSecond: [%@]", firstResult,
        secondResult];
    dispatch_async(dispatch_get_main_queue(), ^{
        self.resultsTextView.text = resultsSummary;
        self.startButton.enabled = YES;
    });
    [self.spinner stopAnimating];
});
NSDate *endTime = [NSDate date];
NSLog(@"Completed in %f seconds",
    [endTime timeIntervalSinceDate:startTime]);
});
}

```

15

构建并运行应用，然后按下按钮。效果是不是更逼真了？即使完成“工作”需要几秒，用户也不会感到在盲目等待，按钮被禁用时同样如此，动画式的旋转指示器告诉用户应用没有真正挂起，有望在某个时刻返回正常状态。

### 3. 并发程序块

到现在为止一切进展顺利，但是我们的程序还没有完成！眼尖的读者会注意到，在经历这些动作后，我们仍然没有真正更改算法的基本顺序布局（你甚至可以回想起算法中这个简单的步骤列表）。我们所做的只是将此方法的一部分转移到一个后台线程，然后在主线程中完成它，Xcode 控制台输出证明了这一点：此“工作”的运行花了 10 秒，就像最初一样。最重要的问题在于，`calculateFirstResult:`和 `calculateSecondResult:`不需要顺序执行，并发执行可以显著提高速度。

幸而 GCD 提供了一种途径来完成此任务：使用所谓的分派组（dispatch group）。将在一个组的上下文中通过 `dispatch_group_async()` 函数异步分派的所有程序块设置为松散的，以尽可能快地执行，如果可能，将它们分发给多个线程来同时执行。也可以使用 `dispatch_group_notify()` 指定一个额外的程序块，让它在组中的所有程序块运行完成时再执行。

对 `doWork:` 作如下修改，确保在方法最后正确加上了大括号和圆括号。

```

- (IBAction)doWork:(id)sender
{
    NSDate *startTime = [NSDate date];
    self.startButton.enabled = NO;
    self.startButton.alpha = 0.5f;
    [self.spinner startAnimating];
    dispatch_queue_t queue =
        dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_async(queue, ^{
        NSString *fetchedData = [self fetchSomethingFromServer];
        NSString *processedData = [self processData:fetchedData];
        NSString *firstResult = [self calculateFirstResult:processedData];
        NSString *secondResult = [self calculateSecondResult:processedData];
    });
}

```

```

    __block NSString *firstResult;
    __block NSString *secondResult;
    dispatch_group_t group = dispatch_group_create();
    dispatch_group_async(group, queue, ^{
        firstResult = [self calculateFirstResult:processedData];
    });
    dispatch_group_async(group, queue, ^{
        secondResult = [self calculateSecondResult:processedData];
    });
    dispatch_group_notify(group, queue, ^{
        NSString *resultsSummary = [NSString stringWithFormat:
            @"First: %@\nSecond: %@",
            firstResult,
            secondResult];
        dispatch_async(dispatch_get_main_queue(), ^{
            self.resultsTextView.text = resultsSummary;
            self.startButton.enabled = YES;
            self.startButton.alpha = 1;
            [self.spinner stopAnimating];
        });
        NSDate *endTime = [NSDate date];
        NSLog(@"Completed in %f seconds",
            [endTime timeIntervalSinceDate:startTime]);
    });
}

```

这里的一个难题是，每个 `calculate` 方法都会返回一个我们希望抓取的值，所以必须首先使用 `__block` 存储修饰符创建变量，这可以确保在程序块内设置的值可被以后运行的代码使用。

完成之后，再次构建并运行应用，你会看到自己的努力得到了回报。以前需要花 10 秒的操作现在只需要 7 秒，这得益于我们同时进行了两种计算。

显然，我们设计的示例获得了最好的效果，因为这两种“计算”实际不会执行任何操作，只会导致运行它们的线程休眠。在真实的应用中，加速程度取决于所执行的工作和可用的资源。只有在有多个 CPU 核心可用时，执行 CPU 资源密集型的计算才能从这种技术受益，在未来配备更多 CPU 的 iOS 设备上，这项技术会变得更好。对于其他用途（比如一次从多个网络连接抓取数据），即使只有一个 CPU 时，也能加快速度。

可以看到，GCD 不是万能的。使用 GCD 不会自动加速所有应用，但是，在应用中速度至关重要或对用户的响应迟缓的部分谨慎地应用这些技术，即使无法改进真实性能，也能够提供更出色的用户体验。

## 15.6 后台处理

处理并发性的另一项重要功能是后台处理，后台处理支持在后台运行应用，在一些情形下甚至可以在用户按下主屏幕按钮之后还在后台运行。

不要将此功能跟现代桌面操作系统提供的真正的多任务混淆，在桌面操作系统中启动的所有程序将保留在系统 RAM 中，直到确实退出系统为止。iOS 设备的 RAM 仍然太少，无法胜任这

一职责。不过,后台处理功能意味着需要特定系统功能的应用可以在受限方式下继续运行。例如,如果应用播放来自互联网广播站的音频流,iOS将允许该应用在用户切换到另一个应用时继续运行。除此之外,在应用播放音频时,iOS甚至还在iOS控制中心(当你从屏幕的底部向上滑动时会出现的半透明面板)上提供了标准的暂停和音量调节控件。

假设你要创建播放音频(即便用户此时正在运行其他应用)、持续请求位置更新(响应特殊类型的推送请求,让其从服务器端加载新数据)或实现VoIP来让用户在互联网上拨打和接听电话的应用,那么这其中的任何一种情况,都可以在应用的Info.plist文件中声明此情形,系统将以一种特殊方式处理应用。这种用途尽管有趣,但可能不是本书的大部分读者将处理的问题,所以我们不打算在这里赘述。

除了在后台运行应用,iOS还能够在用户按下主屏幕按钮之后将应用添加到挂起(suspended)状态中。挂起状态在概念上类似于将Mac设置为休眠模式。应用的所有工作内存都在RAM中,在挂起时它完全不执行。因此,切换回这样的应用的速度非常快。这种行为不仅限于特殊的应用,事实上是使用Xcode构建的任何应用的默认行为(但可以通过Info.plist文件中的另一项设置禁用)。要查看此行为的实际应用,打开设备的邮件应用并打开一封邮件,然后按下主屏幕按钮,打开一款笔记类应用并选择一条笔记。现在双击主屏幕按钮并切换回邮件应用。几乎感觉不到延迟,它会立即滑到视图中,就像它一直在运行一样。

对于大部分应用,这种形式的自动挂起和恢复正是你所需要的。但是,在一些情形下,应用可能需要知道它何时被挂起和被唤醒。系统提供了多种方式,通过UIApplication类通知应用改变其执行状态。UIApplication类针对此用途提供了许多委托方法和通知,本章稍后将介绍如何使用它们。

应用即将被挂起时,它可以做的一件事是请求在后台运行一段时间(无论它是否属于可在后台运行的特殊应用类型)。这样做是为了确保应用有足够的时间来关闭已经打开的文件、网络资源等。稍后将给出一个相关例子。

### 15.6.1 应用生命周期

在详细介绍如何处理对应用执行状态的变更之前,我们探讨一下它的生命周期中有哪些状态。

- ❑ 未运行(Not Running): 此状态表明所有应用都位于一个刚刚重新启动的设备上。在设备打开状态下,不论应用在何时启动,只有遇到以下状况应用才返回未运行状态:
  - 应用的Info.plist包含UIApplicationExitsOnSuspend键(并且其值设置为YES);
  - 应用之前被挂起且系统需要清除一些内存;
  - 应用在运行过程中崩溃。
- ❑ 活跃(Active): 这是应用在屏幕上显示时的正常运行状态。它可以接收用户输入并更新显示。



- ❑ 后台 (Background): 在此状态, 应用获得了一定的时间来执行一些代码, 但它无法直接访问屏幕或获取任何用户输入。在用户按下主屏幕按钮后不久, 所有应用都会进入此状态, 它们中的大部分会迅速进入挂起状态。需要在后台执行各种操作的应用会一直处于此状态, 直到被再次激活。
- ❑ 挂起 (Suspended): 挂起的应用被冻结执行。普通应用在处于后台状态后不久就会转变为此状态。应用在活跃时使用的所有内存将原封不动地得以保留。如果用户将应用切换回活跃状态, 它将恢复到之前的状态。如果系统需要为当前活跃的应用提供更多内存, 任何挂起的应用都可能被终结 (并返回到未运行状态), 它们的内存将被释放用于其他用途。
- ❑ 不活跃 (Inactive): 应用仅在其他状态之间的临时过渡阶段处于不活跃状态。应用可以在任意时间内处于不活跃状态的唯一前提是, 用户正在处理系统提示 (比如显示的传入呼叫或 SMS 提示) 或用户锁定了屏幕。这基本上是一种中间过渡状态。

## 15.6.2 状态更改通知

为了管理这些状态之间的变更, `UIApplication` 定义了它的委托可以实现的一些方法。除了委托方法, `UIApplication` 还定义了一个匹配的通知名称集合 (参见表 15-1)。这使其他对象也像应用委托一样, 可以在应用状态更改时注册通知。

表 15-1 跟踪应用执行状态的委托方法和相应的通知名称

委托方法	通知名称
<code>application:didFinishLaunchingWithOptions:</code>	<code>UIApplicationDidFinishLaunchingNotification</code>
<code>applicationWillResignActive:</code>	<code>UIApplicationWillResignActiveNotification</code>
<code>applicationDidBecomeActive:</code>	<code>UIApplicationDidBecomeActiveNotification</code>
<code>applicationDidEnterBackground:</code>	<code>UIApplicationDidEnterBackgroundNotification</code>
<code>applicationWillEnterForeground:</code>	<code>UIApplicationWillEnterForegroundNotification</code>
<code>applicationWillTerminate:</code>	<code>UIApplicationWillTerminateNotification</code>

请注意, 这些委托方法和通知都直接与某种“运行”状态相关: 活跃、不活跃和后台。每个委托方法仅在一中状态中调用 (每个通知也仅在一中状态中出现)。最重要的状态过渡发生在活跃状态与其他状态之间, 一些过渡 (比如从后台到挂起) 不会出现任何通知。我们分析一下这些方法, 看看应该如何使用它们。

第一个方法 `application:didFinishLaunchingWithOptions:` 已在本书出现多次, 它是在应用启动后直接进行应用级编码的主要方式。

接下来的 `applicationWillResignActive:` 和 `applicationDidBecomeActive:` 会在许多情况下使用。用户按下主屏幕按钮将调用 `applicationWillResignActive:`, 如果他们稍后将应用切换回前台, 将调用 `applicationDidBecomeActive:`。如果用户接听电话, 也会发生相同序列的事件。最神奇的是, 应用启动时也会调用 `applicationDidBecomeActive:`! 一般而言, 这两个方法代表着应用从活跃状态过渡到不活跃状态, 可以用它们来启用或禁用任何动画、应用内的音频或其他用于向用户展示的应用内元素。由于使用 `applicationDidBecomeActive:` 的情况很多, 我们可能想在其中添加一些应用初始



化代码,而不是在 `application:didFinishLaunchingWithOptions:` 中。请注意,不应该在 `applicationWillResignActive:` 中假设应用将进入后台状态,因为它只是一种临时变化,最终将恢复到活跃状态。

接下来是 `applicationDidEnterBackground:` 和 `applicationWillEnterForeground:`, 这两个方法的适用范围稍有不同,它们处理肯定会进入后台状态的应用。应用应该在 `applicationDidEnterBackground:` 中释放所有可在以后重新创建的资源,保存所有用户数据,关闭网络连接等。如果需要,也可以在这里请求在后台运行更长时间,稍后将会演示这一点。如果在 `applicationDidEnterBackground:` 中花了太长时间(超过 5 秒),系统将断定应用的行为异常并终止它。应该实现 `applicationWillEnterForeground:` 来重新创建在 `applicationDidEnterBackground:` 中销毁的内容,比如重新加载用户数据、重新建立网络连接等。请注意,当调用 `applicationDidEnterBackground:` 时,可以安全地假设最近也调用了 `applicationWillResignActive:`。类似地,当调用 `applicationWillEnterForeground:` 时,可以认为即将调用 `applicationDidBecomeActive:`。

列表中的最后一个方法是 `applicationWillTerminate:`, 你可能很少使用它。只有在应用已进入后台,并且系统出于某种原因决定跳过暂停状态并终止应用时,才会真正调用它。

现在你对应用状态过渡的理论有了基本了解。我们在一个简单的应用(只是在每次调用这些方法时向 Xcode 的控制台日志写入一条消息)中试用一下这些知识。然后我们通过各种方式操作正在运行的应用,就像用户一样,看一下将发生哪些过渡。

### 15.6.3 创建 State Lab 项目

在 Xcode 中,以 Single View Application 模板为基础创建一个新项目,将它命名为 State Lab。除了自带的默认灰色屏幕,此应用不会显示任何信息。它所生成的所有输出最终都将进入 Xcode 控制台。BIDAppDelegate.m 文件已经包含我们感兴趣的所有方法,其他要做的就是添加一些日志,如以下粗体代码所示。请注意,我们也删除了这些方法中的注释,以保持简洁。

```
#import "BIDAppDelegate.h"

#import "BIDViewController.h"

@implementation BIDAppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSLog(@"%@", NSStringFromSelector(_cmd));
    return YES;
}

- (void)applicationWillResignActive:(UIApplication *)application
{
    NSLog(@"%@", NSStringFromSelector(_cmd));
}
```

```

- (void)applicationDidEnterBackground:(UIApplication *)application
{
    NSLog(@"%@", NSStringFromSelector(_cmd));
}
- (void)applicationWillEnterForeground:(UIApplication *)application
{
    NSLog(@"%@", NSStringFromSelector(_cmd));
}
- (void)applicationDidBecomeActive:(UIApplication *)application
{
    NSLog(@"%@", NSStringFromSelector(_cmd));
}
- (void)applicationWillTerminate:(UIApplication *)application
{
    NSLog(@"%@", NSStringFromSelector(_cmd));
}

@end

```

你可能希望了解我们在所有这些方法中使用的 NSLog 调用。Objective-C 提供了一个方便的内置变量，名为 `_cmd`，它始终包含当前方法的选择器。回想一下，选择器只是 Objective-C 引用方法的一种方式，`NSStringFromSelector()` 函数返回给定选择器的 `NSString` 表示。这里只是给出了输出当前方法名的快捷方式，无需重新键入或复制粘贴。

### 15.6.4 执行状态的变化

现在构建并运行应用。模拟器将显示并启动我们的应用。切换回 Xcode 并查看一下控制台 (View>Debug Area>Activate Console)，应该会看到以下类似信息：

```

2013-11-15 19:12:36.953 State Lab[12751:70b] application:didFinishLaunchingWithOptions:
2013-11-15 19:12:36.957 State Lab[12751:70b] applicationDidBecomeActive:

```

可以看到，应用已成功启动并进入了活跃状态。现在返回到模拟器并按下主屏幕按钮，应该会在控制台看到以下信息：

```

2013-11-15 19:13:10.378 State Lab[12751:70b] applicationWillResignActive:
2013-11-15 19:13:10.386 State Lab[12751:70b] applicationDidEnterBackground:

```

这两行显示了应用两个状态之间的实际过渡。它首先转变为不活跃状态，然后进入后台。在这里无法看到的是，应用还切换到了第三个状态：挂起。前面提到过，你不会被告知发生了这一过程，它完全在你的控制之外。请注意，从某种意义上讲该应用仍然是“活跃的”，Xcode 仍然与它相连，即使它实际上没有占用任何 CPU 时间。要验证这一点，可以点击应用的图标重新启动它，这应该生成以下输出：

```

2013-11-15 19:13:55.739 State Lab[12751:70b] applicationWillEnterForeground:
2013-11-15 19:13:55.739 State Lab[12751:70b] applicationDidBecomeActive:

```

应用又开始正常运行了。应用之前被挂起，然后唤醒到不活跃状态，最后再次返回活跃状态。

那么当应用真正被终结时会发生什么？再次点击主屏幕按钮，你会看到：

```
2013-11-15 19:14:35.035 State Lab[12751:70b] applicationWillResignActive:
2013-11-15 19:14:35.036 State Lab[12751:70b] applicationDidEnterBackground:
```

然后双击主屏幕按钮，会出现一个能够横向滚动的应用列表，按住并向上滑动 State Lab 截屏直到飞出屏幕以结束 State Lab。发生了什么呢？你可能很惊讶，NSLog 调用没有向控制台打印任何信息。相反，应用卡在了 main.m 中的 UIApplication 调用上，并且有一个错误消息“Thread 1: singal SIGKILL”。单击 Xcode 左上角的 Stop 按钮，现在 State Lab 就真的完全终止了。

事实证明，在系统将应用从挂起转为未运行状态时，没有正常地调用 applicationWillTerminate: 方法。当应用挂起时，无论系统决定转储它以回收内存，还是开发人员手动强制退出它，它都会直接消失，无法执行任何操作。applicationWillTerminate: 方法仅在被终结的应用处于后台状态时才会被调用。例如，如果应用在后台状态下运行，以之前预定义的某种方式（音频播放、GPS 使用等）使用系统资源，并被用户或系统强制退出，那么可能调用此方法。在 State Lab 例子中，应用处于挂起状态，而不是后台状态，所以应用立即终止，没有任何通知。

这里还有另一种有趣的交互需要介绍，就是当系统在显示警告对话框时，临时接管来自应用的输入流，并且将它设为不活跃状态。这种状态只有在真实设备上（而不是模拟器）运行时才会被触发（使用内置的信息应用）。信息应用和其他很多应用一样，可以从外部接收消息，并能通过多种方式显示。

要看看这是如何建立的，可以在你的设备上运行设置应用，在列表中选择通知中心，然后从应用列表里选择信息应用。iOS 5 中首次引入显示消息的全新方式称为“横幅”。它显示一条覆盖在屏幕顶部的小通知栏，而不会中断当前正在运行的应用。但是，我们这里想要显示的是不太理想的旧式“提醒”方法，它会在当前应用前弹出一个窗口，要求用户进行操作。选择这种方式后，信息应用将会回到早期的形式（使用 iOS 4 及早期版本的用户所不得不处理的恼人情形）。

现在回到 Mac 中。在 Xcode 中，使用左上方的弹出菜单从模拟器切换到你的设备，然后点击 Run 按钮在你的设备上构建和运行该应用。现在，你要做的是向设备发送一条消息。如果使用的是 iPhone，可以用另一个手机给它发送一条短信。如果是 iPod touch 或者 iPad，只能用 Apple 自带的 iMessage 进行通信，它在所有 iOS 设备上都可使用（包括 OS X 中的“信息”应用）。根据你的情况，通过短信或者 iMessage 向 iOS 设备上发送一条消息。当设备显示收到消息的提醒时，将会在 Xcode 控制台中出现以下信息：

```
2013-11-18 00:04:28.295 State Lab[16571:60b] applicationWillResignActive:
```

请注意，应用没有被发送到后台，它处于不活跃状态，并且仍然可以在系统提醒背后看到。如果此应用是一个游戏或正在运行着视频、音频或动画，那么这时需要暂停它们。

按下提醒上的“关闭”按钮，将得到以下信息：

```
2013-11-18 00:05:23.830 State Lab[16571:60b] applicationDidBecomeActive:
```

现在看一下如果决定回复短信会发生什么。发送另一条短信到你的设备，这会生成以下信息：

```
2013-11-18 00:05:55.487 State Lab[16571:60b] applicationWillResignActive:
```

这一次单击“回复”，切换回信息应用，应该会看到以下一系列活动：

```
2013-11-18 00:06:10.513 State Lab[16571:60b] applicationDidBecomeActive:
2013-11-18 00:06:11.137 State Lab[16571:60b] applicationWillResignActive:
2013-11-18 00:06:11.140 State Lab[16571:60b] applicationDidEnterBackground:
```

非常有趣！我们的应用迅速地再次激活，然后变为不活跃，最后进入后台（接下来默默地被挂起）。

### 15.6.5 利用执行状态更改

那么，我们应该如何对待这些状态呢？基于刚才演示的例子，看起来在处理这些状态更改时有一条明确的策略可以遵循。

#### 1. 活跃➤不活跃

使用 `applicationWillResignActive:` 或者 `UIApplicationWillResignActiveNotification` 来“暂停”应用的显示。如果应用是游戏，你可能已经能够通过某种方式暂停游戏。对于其他类型的应用，确保工作中对用户输入没有严格的时间要求，因为应用在一段时间内不会获得任何用户输入。

#### 2. 不活跃➤后台

使用 `applicationDidEnterBackground:` 或者 `UIApplicationDidEnterBackgroundNotification` 释放在应用处于后台状态时不需要保留的任何资源（比如缓存的图像或其他可轻松重新加载的数据），或者无法保存在后台状态的任何资源（比如活跃的网络连接）。在这里避免过度使用内存可确保应用最终的挂起快照更小，从而降低了应用从 RAM 中完全清除的风险。还应该借此机会保存任何必要的应用数据，这些数据将有助于用户在下一次重新启动应用时找到上次离开时的进度。如果应用返回到活跃状态，这通常没什么问题，但在应用被清除并必须重新启动时，用户会非常希望从相同位置恢复。

#### 3. 后台➤不活跃

使用 `applicationWillEnterForeground:` 或者 `UIApplicationWillEnterForeground` 恢复从不活跃切换到后台状态时所执行的任何操作。例如，可以从这里重新建立持久网络连接。

#### 4. 不活跃➤活跃

使用 `applicationDidBecomeActive:` 或者 `UIApplicationDidBecomeActive` 恢复从活跃切换到不活跃状态时执行的所有操作。请注意，如果是游戏类应用，这可能不会直接从暂停状态返回到游戏，应该让用户自行返回到游戏。另外请记住，这个方法和通知在应用全新启动时使用，所以这里执行的任何操作也必须在该上下文中有有效。

对于从不活跃到后台状态的过渡，还有一个需要注意的特殊因素。相比其他过渡，它除了描述文字最多，还可能是大部分应用中使用代码最多和耗时最长的过渡，因为你希望应用执行的“登记”操作量可能很大。在执行此过渡的过程中，系统不会提供大量时间来保存这里的更改，仅提供大约 5 秒。如果应用从委托方法返回（或处理已经注册的任何通知）的时间超过 5 秒，应用将立刻从内存中清除并进入未运行状态！如果这看起来不公平，不要担心，因为可以采用一种推迟方法。在处理该委托方法或通知时，可以要求系统在后台队列中执行一些额外的工作，这会争取

到更多时间。下一节将介绍这种技术。

### 15.6.6 处理不活跃状态

应用遇到的最简单的状态更改是从活跃过渡到不活跃,然后再返回到活跃。回想一下,iPhone 在应用运行时收到短信并显示就是这种情况。本节将让 State Lab 执行一些有意思的操作,这样我们可以看到忽略该状态更改会发生什么,然后了解如何修复它。

我们还要将一个 UILabel 添加到显示视图,使用 Core Animation (iOS 中制作对象动画的方法)来移动它。

首先在 BIDViewController.m 中添加一个 UILabel 作为实例变量和属性:

```
#import "BIDViewController.h"

@interface BIDViewController ()

@property (strong, nonatomic) UILabel *label;

@end
```

现在设置视图加载时的标签。向 viewDidLoad 方法添加以下粗体显示的代码行:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // 加载视图后进行一些其他设置,通常是从 nib 文件
    CGRect bounds = self.view.bounds;
    CGRect labelFrame = CGRectMake(bounds.origin.x, CGRectGetMidY(bounds) - 50,
                                   bounds.size.width, 100);
    self.label = [[UILabel alloc] initWithFrame:labelFrame];
    self.label.font = [UIFont fontWithName:@"Helvetica" size:70];
    self.label.text = @"Bazinga!";
    self.label.textAlignment = NSTextAlignmentCenter;
    self.label.backgroundColor = [UIColor clearColor];
    [self.view addSubview:self.label];
}
```

然后设置一些动画。我们将定义两个方法,一个用于将标签旋转为倒置,另一个将它旋转回正常位置:

```
- (void)rotateLabelDown
{
    [UIView animateWithDuration:0.5
        animations:^(
            self.label.transform = CGAffineTransformMakeRotation(M_PI);
        )
        completion:^(BOOL finished){
            [self rotateLabelUp];
        }];
}
```

```

- (void)rotateLabelUp
{
    [UIView animateWithDuration:0.5
        animations:^(
            self.label.transform = CGAffineTransformMakeRotation(0);
        )
        completion:^(BOOL finished){
            [self rotateLabelDown];
        }];
}

```

这段代码需要解释一下。UIView 定义一个名为 `animateWithDuration:animations:completion:` 的类方法，该方法设置一个动画。我们在动画块内设置的任何可制定动画的属性都不会立即在接收程序上实现动画效果。Core Animation 会将该属性从其当前值流畅地过渡到我们指定的新值。这就是所谓的隐式动画（implicit animation），是 Core Animation 的主要功能之一。最后完成的程序块可指定在动画完成后执行何种操作。

所以，这些方法将标签的 `transform` 属性设置为特定的旋转角度（以弧度为单位指定）。它们还设置一个完成程序块来调用其他方法，使文本继续不停地反复显示动画。

最后，我们需要设置一种方式来启动动画。在 `viewDidLoad` 末尾添加以下代码来实现该功能（但在后面会更改此代码，其原因到时候会说明）：

```
[self rotateLabelDown];
```

现在构建并运行应用，应该看到 Bazinga! 标签不停地旋转（参见图 15-4）。



图 15-4 State Lab 应用旋转标签

要测试活跃→不活跃过渡，需要在真正的 iPhone 上再次运行此应用，从其他设备向它发送短信。不幸的是，苹果公司目前发布的所有版本的 iOS 模拟器都无法模拟此行为。如果还不能在设备上构建并安装这个应用，或者没有 iPhone，你就无法亲自尝试此应用，这种情况下请尽可能继续学习。

在 iPhone 上构建并运行应用，可以看到动画一直运行。现在向设备发送一条短信，系统警告显示该短信时会看到动画仍在运行！这可能很有趣，但却不是用户想要的。我们将使用过渡通知在发生这种情况时停止动画。

我们的控制器类将需要有某种内部状态来记录它是否应该在给定时刻显示动画。出于此用途，我们向 BIDViewController.m 添加一个实例变量。因为这是一个简单的布尔值，不需要外部类访问，所以我们跳过了文件头部，将它添加到@implementation 部分中。

```
@implementation BIDViewController {
    BOOL animate;
}
```

因为我们的类不是应用委托，所以无法实现委托方法并期望它们生效，但我们可以注册以接收在执行状态更改时来自应用的通知。为此，在 BIDViewController.m 文件中的 viewDidLoad 方法末尾添加以下代码：

```
NSNotificationCenter *center = [NSNotificationCenter defaultCenter];
[center addObserver:self
    selector:@selector(applicationWillResignActive)
    name:UIApplicationWillResignActiveNotification
    object:nil];
[center addObserver:self
    selector:@selector(applicationDidBecomeActive)
    name:UIApplicationDidBecomeActiveNotification
    object:nil];
```

这段代码设置了两个通知，将分别在恰当的時刻调用我们的类中的一个方法。可以在@implementation 程序块中的任意位置定义这些方法：

```
- (void)applicationWillResignActive
{
    NSLog(@"VC: %@", NSStringFromSelector(_cmd));
    animate = NO;
}

- (void)applicationDidBecomeActive
{
    NSLog(@"VC: %@", NSStringFromSelector(_cmd));
    animate = YES;
    [self rotateLabelDown];
}
```

这段代码包含了与以前相同的方法记录日志，所以可以在 Xcode 控制台中看到它们在何处发生。注意，我们在 NSLog() 调用开头添加了"VC:"以区别在委托中调用该方法。第一个方法关闭 animate 标记，第二个打开该标记，然后再次实际地启动动画。要使第一个方法有效果，我们必须添加一些代码来检查 animate 标记，并仅在它启用时保持动画效果。



```

- (void)rotateLabelUp
{
    [UIView animateWithDuration:0.5
        animations:^(
            self.label.transform = CGAffineTransformMakeRotation(0);
        )
        completion:^(BOOL finished){
            if (animate) {
                [self rotateLabelDown];
            }
        }];
}

```

我们将这段代码添加到 `rotateLabelUp` 的完成程序块中，只有添加到这里，动画才仅在文本旋转到正常位置时停止。

现在再次构建并运行，看看会发生什么。可能会看到屏幕在不停闪动，标签在迅速上下翻转，甚至没有旋转！出现这种情况的原因很简单，但可能不那么明显（我们在前面提示过）。

还记得我们在 `viewDidLoad` 末尾通过调用 `rotateLabelDown` 来启动动画吗？我们现在也在 `applicationDidBecomeActive` 中调用 `rotateLabelDown`。请记住，`applicationDidBecomeActive` 不仅会在从不活跃切换到活跃状态时调用，而且会在应用启动并首次变为活跃状态时调用。这意味着我们启动了动画两次，Core Animation 似乎不能很好地处理多个动画，确实有两个动画都试图同时更改相同特性的动画。解决方案很简单，就是删除前面在 `viewDidLoad` 末尾添加的代码：

```

-[self rotateLabelDown];

```

现在构建并运行，应该会看到动画运行正常。再次向 iPhone 发送短信，这一次当系统警告出现时将会看到，只要文本转到正确位置，后台运行的动画就会停止。点击 Close 按钮，动画将重新开始。

前面介绍了如何处理从活跃切换到不活跃状态并切换回来的简单情形。更大型（或许也是更重要）的任务是处理切换到后台，然后切换回前台的过程。

### 15.6.7 处理后台状态

我们前面已经提到，切换到后台状态对于确保最佳用户体验非常重要。我们需要在这里丢弃可轻松地重新获取（或在应用进入静默状态时一定会丢失）的资源，保存与应用当前状态相关的信息，所有这些操作都不应占用主线程超过 5 秒钟。

为了演示部分行为，我们将通过多种方式扩展 State Lab。首先，向显示视图添加一个图像，以便可以在以后展示如何删除内存中的图像。然后，展示如何保存与应用状态相关的信息，以便可以在以后轻松地还原它。最后，我们将展示如何将所有这些工作放入后台队列中，确保这些活动不会占用主线程太长时间。

#### 1. 进入后台时删除资源

首先从本书的源文件归档中将 `smiley.png` 添加到项目的 State Lab 文件夹，一定要勾选告诉 Xcode 将文件复制到项目目录的复选框。不要将它添加到 `Images.xcassets` 资源目录中，因为这样



会产生自动缓存，它将妨碍我们将要实现的资源管理。

现在，将图像和图像视图的属性添加到 BIDViewController.m 中：

```
@interface BIDViewController ()

@property (strong, nonatomic) UILabel *label;
@property (strong, nonatomic) UIImage *smiley;
@property (strong, nonatomic) UIImageView *smileyView;
```

```
@end
```

然后设置图像视图，并通过修改 viewDidLoad 方法将它放在屏幕上，如下所示：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // 完成视图加载后进行一些其他设置，通常是从 nib 文件
    CGRect bounds = self.view.bounds;
    CGRect labelFrame = CGRectMake(bounds.origin.x, CGRectGetMaxY(bounds) - 50,
                                    bounds.size.width, 100);
    self.label = [[UILabel alloc] initWithFrame:labelFrame];
    self.label.font = [UIFont fontWithName:@"Helvetica" size:70];
    self.label.text = @"Bazinga!";
    self.label.textAlignment = NSTextAlignmentCenter;
    self.label.backgroundColor = [UIColor clearColor];

    // smiley.png 的尺寸是 84 像素 x 84 像素
    CGRect smileyFrame = CGRectMake(CGRectGetMidX(bounds) - 42,
                                    CGRectGetMidY(bounds)/2 - 42,
                                    84, 84);
    self.smileyView = [[UIImageView alloc] initWithFrame:smileyFrame];
    self.smileyView.contentMode = UIViewContentModeCenter;
    NSString *smileyPath = [[NSBundle mainBundle] pathForResource:@"smiley"
                                                                ofType:@"png"];
    self.smiley = [UIImage imageWithContentsOfFile:smileyPath];
    self.smileyView.image = self.smiley;
    [self.view addSubview:self.smileyView];

    [self.view addSubview:self.label];

    NotificationCenter *center = [NotificationCenter defaultCenter];
    [center addObserver:self
               selector:@selector(applicationWillResignActive)
               name:UIApplicationWillResignActiveNotification
               object:nil];
    [center addObserver:self
               selector:@selector(applicationDidBecomeActive)
               name:UIApplicationDidBecomeActiveNotification
               object:nil];
}
```

构建并运行应用，会在屏幕中的旋转文本的上方看到一个非常开心的笑脸（参见图 15-5）。

现在，按下主屏幕按钮将应用切换到后台，然后点击它的图标再次启动它。可以看到，应用重启后，仍然保持着它进入后台之前的状态。这对用户来说很好，但我们没有尽可能地优化系统资源。



```

    self.smiley = [UIImage imageNamed:@"smileyPath"];
    self.smileyView.image = self.smiley;
}

```

现在构建并运行应用，执行与让应用进入后台并切换回来相同的步骤。从用户角度看，应用的行为应该大体相同。如果希望亲自确认发生了此行为，可以注释掉 `applicationWillEnterForeground` 方法的内容，再次构建并运行应用，应该会看到图像真的消失了。

## 2. 进入后台时保存状态

前面的例子展示了如何在进入后台时释放一些资源，现在是时候考虑保存状态了。请记住，我们的想法是保存与用户执行的操作相关的所有信息，这样一来，如果应用在以后从内存转储，用户下次回来时仍然可以恢复到他们离开时的进度。

我们这里介绍的状态类型与具体的应用密切相关。你可能希望知道用户在查看哪个文档，他们的光标在文本字段中的位置，打开了哪个应用视图，等等。在我们的例子中，将跟踪用户在分段控件中的选择。

首先在 `BIDViewController.m` 中添加一个新属性：

```

#import "BIDViewController.h"

@interface BIDViewController ()

@property (strong, nonatomic) UILabel *label;
@property (strong, nonatomic) UIImage *smiley;
@property (strong, nonatomic) UIImageView *smileyView;
@property (strong, nonatomic) UISegmentedControl *segmentedControl;

@end

```

然后移到 `BIDViewController.m` 中的 `viewDidLoad` 方法的中间，我们将在这里创建分段控件并将它添加到视图：

```

.
.
.
    self.smileyView.image = self.smiley;

    self.segmentedControl = [[UISegmentedControl alloc] initWithItems:
        [NSArray arrayWithObjects:
            @"One", @"Two", @"Three", @"Four", nil]] ;
    self.segmentedControl.frame = CGRectMake(bounds.origin.x + 20,
        50,
        bounds.size.width - 40, 30);

    [self.view addSubview:self.segmentedControl];
    [self.view addSubview:self.smileyView];
    [self.view addSubview:self.label];
.
.

```

构建并运行该应用。现在应该可以看到这个分段控件，并且能够点击其分段，每次选择其中一个。再次按下主屏幕按钮让应用进入后台运行，然后双击主屏幕按钮打开任务栏，接着结束你

的应用，然后再重新启动它。你会发现每次都会回到初始状态，没有分段被选中。接下来就修复这个问题。

保存分段选择是非常简单的。我们需要在 BIDViewController.m 文件中的 applicationDidEnterBackground 方法末尾添加几行代码：

```
- (void)applicationDidEnterBackground
{
    NSLog(@"VC: %@", NSStringFromSelector(_cmd));
    self.smiley = nil;

    self.smileyView.image = nil;
    NSInteger selectedIndex = self.segmentedControl.selectedSegmentIndex;
    [[NSUserDefaults standardUserDefaults] setInteger:selectedIndex
                                     forKey:@"selectedIndex"];
}
```

但是应该在哪里恢复分段选择的索引并用于配置分段控件呢？applicationWillEnterForeground 方法并不是我们想要的。这个方法被调用的时候应用已经处于运行状态了，但设置还是最原始的状态。应该在应用重新运行之后就进行设置，也就是说，应该在 viewDidLoad 方法中进行处理。在 viewDidLoad 方法中添加如下代码：

```
.
.
.
[self.view addSubview:self.label];

NSNumber *indexNumber = [[NSUserDefaults standardUserDefaults]
                        objectForKey:@"selectedIndex"];

if (indexNumber) {
    NSInteger selectedIndex = [indexNumber intValue];
    self.segmentedControl.selectedSegmentIndex = selectedIndex;
}
.
.
.
```

我们必须包含一种合理性检查，查看是否为 selectedIndex 键存储了值，以涵盖首次启动应用（这时没有选择任何分段）等情形。

现在构建并运行应用，触摸一个分段，然后执行完整的“后台—结束—重新启动”步骤，所做的选择保持不变！

显然，我们这里所介绍的概念非常简单，但可以将该概念扩展到所有的应用状态。你可以自行决定应用它的程度，从而让用户感觉应用仍然在运行并在等待他们回来！

### 3. 请求更多后台时间

前面提过，如果进入后台状态花费了太长时间，应用可能会从内存中移出。例如，你的应用可能正在进行文件传输工作，如果没能完成的话则很遗憾，但试图强制 applicationDidEnterBackground 方法在应用真正进入后台前完成这项工作，并不是一个很好的选择。相反，你应该将 applicationDidEnterBackground 作为平台，告诉系统你还有额外的工作要做，然后启动一个程序块，真正地执行该工作。假设用户在执行其他操作时系统仍然有足够的 RAM 将你的应

用保存在内存中，那么系统会强制保留你的应用继续运行一段时间。

我们将要演示这一点，不过不是真正的文件传输，而是一个简单的睡眠呼叫。我们会再次使用刚刚学习的 GCD 和程序块，让 `applicationDidEnterBackground` 方法的内容运行在一个单独的队列中。

在 `BIDViewController.m` 中，修改 `applicationDidEnterBackground` 方法，如下所示：

```
(void)applicationDidEnterBackground
{
    NSLog(@"VC: %@", NSStringFromSelector(_cmd));
    UIApplication *app = [UIApplication sharedApplication];

    __block UIBackgroundTaskIdentifier taskId;
    taskId = [app beginBackgroundTaskWithExpirationHandler:^(
        NSLog(@"Background task ran out of time and was terminated.");
        [app endBackgroundTask:taskId];
    )];

    if (taskId == UIBackgroundTaskInvalid) {
        NSLog(@"Failed to start background task!");
        return;
    }

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
    ^{
        NSLog(@"Starting background task with %f seconds remaining",
            app.backgroundTimeRemaining);
        self.smiley = nil;
        self.smileyView.image = nil;
        NSInteger selectedIndex = self.segmentedControl.selectedSegmentIndex;
        [[NSUserDefaults standardUserDefaults] setInteger:selectedIndex
            forKey:@"selectedIndex"];

        // 模拟一个 25 秒的过程
        [NSThread sleepForTimeInterval:25];

        NSLog(@"Finishing background task with %f seconds remaining",
            app.backgroundTimeRemaining);
        [app endBackgroundTask:taskId];
    });
}
```

我们详细分析一下这段代码。首先抓取共享的 `UIApplication` 实例，因为我们将在此方法中多次使用它。然后是以下代码：

```
__block UIBackgroundTaskIdentifier taskId;
taskId = [app beginBackgroundTaskWithExpirationHandler:^(
    NSLog(@"Background task ran out of time and was terminated.");
    [app endBackgroundTask:taskId];
)];
```

对 `beginBackgroundTaskWithExpirationHandler:` 的调用返回一个标识符，我们需要跟踪它以供后续使用。我们声明了 `taskId` 变量，并使用 `__block` 存储修饰符进行存储，因为希望确保该方法返回的标识符在方法创建的所有程序块中可以共享。

调用 `beginBackgroundTaskWithExpirationHandler:` 基本是在告诉系统：我们需要更多时间来完成某件事，并承诺在完成时告诉它。如果系统断定我们运行了太长时间并决定停止运行，可以调用我们作为参数提供的程序块。

请注意，我们提供的程序块最后会调用 `endBackgroundTask:`，传入 `taskId`。这告诉系统我们完成了之前请求额外时间来完成的工作。一定要权衡对 `beginBackgroundTaskWithExpirationHandler:` 的每次调用和对 `endBackgroundTask:` 的匹配调用，以便让系统知道我们何时完成工作。

---

**注意** 这里说的“任务”一词可能使人联想起那个计算机术语，类似我们通常所说的“进程”，即包含多个线程的程序。在本例中使用的“任务”不是专业术语，只是表示“某件需要完成的事情”。这里创建的任何“任务”都在你执行的应用中运行。

---

接下来，添加以下代码：

```
if (taskId == UIBackgroundTaskInvalid) {
    NSLog(@"Failed to start background task!");
    return;
}
```

如果前面对 `beginBackgroundTaskWithExpirationHandler:` 的调用返回特殊值 `UIBackgroundTaskInvalid`，则表明系统没有为我们提供任何多余的时间。在这种情况下，可以尝试完成必须完成的操作中最快的部分，希望它能在应用终止之前完成。例如在较旧设备（比如 iPhone 3G）上运行时，这很可能无法完成。但是，在本例中，我们只是让它滑动一下。

接下来是完成工作本身的有趣部分：

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
^ {
    NSLog(@"Starting background task with %f seconds remaining",
        app.backgroundTimeRemaining);
    self.smiley = nil;
    self.smileyView.image = nil;
    NSInteger selectedIndex = self.segmentedControl.selectedSegmentIndex;
    [[NSUserDefaults standardUserDefaults] setInteger:selectedIndex
        forKey:@"selectedIndex"];

    // 模拟一个 25 秒的过程
    [NSThread sleepForTimeInterval:25];

    NSLog(@"Finishing background task with %f seconds remaining",
        app.backgroundTimeRemaining);
    [app endBackgroundTask:taskId];
});
```

这段代码所做的是获取我们的方法最初所做的工作，并将它放在一个后台队列中。在该程序块末尾，我们调用 `endBackgroundTask:` 来让系统知道我们已完成。

添加此代码后构建并运行应用，然后按下主屏幕按钮让应用进入后台，观察 Xcode 控制台以及 Xcode 窗口底部的状态栏。这次将会看到，应用一直在运行（不会在状态栏上获得“Debugging terminated”消息），25 秒过后将会在输出中看到最后的日志。现在完整地运行应用将会得到包含

以下内容的控制台输出：

```
2013-11-18 01:30:08.194 State Lab[12158:70b] application:didFinishLaunchingWith
Options:
2013-11-18 01:30:08.209 State Lab[12158:70b] applicationDidBecomeActive:
2013-11-18 01:30:08.210 State Lab[12158:70b] VC: applicationDidBecomeActive
2013-11-18 01:30:17.010 State Lab[12158:70b] applicationWillResignActive:
2013-11-18 01:30:17.011 State Lab[12158:70b] VC: applicationWillResignActive
2013-11-18 01:30:17.018 State Lab[12158:70b] applicationDidEnterBackground:
2013-11-18 01:30:17.019 State Lab[12158:70b] VC: applicationDidEnterBackground
2013-11-18 01:30:17.021 State Lab[12158:3a03] Starting background task with
179.988868 seconds remaining
2013-11-18 01:30:42.027 State Lab[12158:3a03] Finishing background task with
154.986797 seconds remaining
```

可以看到，在后台执行操作与它在应用的主线程中相比，系统提供了更多的时间，所以如果还有任何正在运行的任务要处理，此步骤真正有助于完成工作。

请注意，我们仅请求了一个后台任务标识符，但实际上如果需要，可以请求尽可能多的标识符。例如，如果在后台发生了多个网络传输任务，并且需要完成它们，可以为每个任务请求一个标识符，并允许每项任务在后台队列中继续运行，以便可以轻松地允许多个操作在可用的时间内并行运行。另外考虑收到的任务标识符是一个普通的 C 语言值（不是对象），除了存储在局部 `__block` 变量中，它也可以存储为实例变量，只要这样更适合你的类设计。

## 15.7 小结

本章的内容非常丰富，介绍了大量新概念。不仅介绍了苹果公司添加到 C 语言的完整的新功能，还介绍了处理并发的一种新的概念范式，无需考虑线程即可处理并发性。此外，我们还阐述了确保应用在 iOS 的多线程世界中良好运行的技术。现在我们已经解决了一些重要问题，下一章将介绍绘图。请准备好铅笔，开始绘画！

到目前为止,我们创建的所有应用程序都是通过使用 UIKit 框架中的视图以及控件来构造的。UIKit 框架的用处很大,仅仅利用这些其中的预定义对象,就完全可以构造出许多优质的应用程序。然而,对于有些可视化元素来说,仅仅使用 UIKit 框架内置的组件是无法完全实现的。

例如,有时应用程序需要能够进行自定义绘图。幸好 iOS 内置了 Core Graphics 框架,能帮助我们进行一系列的各类绘图工作。在本章中,我们将要探索这个功能强大的图形绘制环境。通过构建一个示例应用程序来深度展示 Core Graphics 的主要功能,并详细讲解其核心概念。

## 16.1 Quartz 2D 基础概念

Core Graphics 中的关键部分是一个名为 Quartz 2D 的 API 集合,它包含了各种函数、数据类型以及对象,能让你在内存中直接绘制视图和图像。Quartz 2D 将正在进行绘制的视图或图像视为一个虚拟的画布,并遵循**绘画模式**。之所以用这个词,是因为使用绘图命令的方式很大程度上与在画布上使用颜料的方式相同。

如果绘画者将整个画布涂成红色,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以及一半蓝色或紫色(如果颜料是不透明的,那应该为蓝色;而如果颜料是半透明的,那应该为紫色)。Quartz 2D 的虚拟画布采用了相同的工作方式。如果将整个视图涂为红色,然后将视图的下半部分涂为蓝色,那么你将拥有一个一半是红色,而另一半是蓝色(或紫色)的视图,这要取决于第二个绘图操作是完全不透明的还是部分透明的。画布上的每一个绘图操作都会覆盖在之前的绘图操作之上。

Quartz 2D 提供了各种直线、形状以及图像的绘图函数。虽然使用方便,但 Quartz 2D 仅限于二维绘图。尽管许多 Quartz 2D 函数会在绘图时利用硬件加速,但无法保证在 Quartz 2D 中执行的所有操作都会得到加速。

基本了解 Quartz 2D 之后,下面就可以开始进行尝试了。我们将从 Quartz 2D 工作原理的基础讲起,并随后构建一个简单的应用程序。

## 16.2 Quartz 2D 绘图方法

使用 Quartz 2D(也可以简称 Quartz)绘制图形时,通常需要向图形所在的视图中添加绘图



代码。比如说创建一个 `UIView` 的子类，并向该类的 `drawRect:` 方法中添加对 Quartz 函数的调用。`drawRect:` 方法是 `UIView` 类定义的一部分，视图每次需要自身重新绘制时都会调用该方法，所以如果在 `drawRect:` 中插入 Quartz 代码，那么该段代码就会在视图重新绘制之前先被调用。

### 16.2.1 Quartz 2D图形环境

与 Core Graphics 中其他部分一样，Quartz 的绘制是在图形环境（graphics context）中进行的，通常简称它为环境（context）。每个视图都有相关的环境。你需要先获取当前环境，通过它可以调用各类 Quartz 绘图函数，由环境负责将图形绘制在视图上。你可以认为环境是一种画布。系统提供了默认的环境来将显示内容呈现在屏幕上。然而你也可以创建一个自己的环境，来创建不想立刻显现在屏幕上的绘制内容，以待后续使用。我们现在主要研究默认的环境，可以使用下面的语句来获取它。

```
CGContextRef context = UIGraphicsGetCurrentContext();
```

**说明** 在这里我们使用的是 Core Graphics 框架的 C 语言函数来进行绘制，而不是 Objective-C 对象。某些函数拥有面向对象的特性，我们将在后面章节中详细谈到。

定义图形环境之后，就可以通过传递该环境给各种 Core Graphics 绘图函数来进行绘制。例如，以下代码将会创建并绘制一条由直线组成的路径：

```
CGContextSetLineWidth(context, 4.0);
CGContextSetStrokeColorWithColor(context, [UIColor redColor].CGColor);
CGContextMoveToPoint(context, 10.0, 10.0);
CGContextAddLineToPoint(context, 20.0, 20.0);
CGContextStrokePath(context);
```

第一个函数调用指定了之后所有参与创建当前路径的绘制命令所使用的画笔宽度必须为 4 点。可以将其想像成选择绘图时所用的笔刷尺寸大小。在你再次调用该函数设置一个不同的值之前，所有的直线在绘制时宽度都应该为 4 点数。然后，指定笔刷颜色应该为红色。Core Graphics 的绘图操作中涉及以下两种颜色：

- 笔刷颜色，用于绘制直线以及形状的轮廓；
- 填充颜色，用于填充形状。

可以认为，环境是通过一支看不见的画笔来绘制线条的。随着绘制命令的执行，画笔的移动会形成一段路径。当调用 `CGContextMoveToPoint()` 时，会将虚拟画笔移动到指定的位置，事实上这样做不会执行任何绘图操作。无论接下来执行何种操作，它都将以画笔所移动到的点为参照物执行自己的工作。以前面的代码为例，我们首先将画笔移动到 (10, 10)。下一个函数调用绘制一条从当前的画笔位置 (10, 10) 到指定位置 (20, 20) 的线条，而 (20, 20) 会成为画笔的新位置。

在 Core Graphics 中绘图时，你并没有绘制任何实际可见的内容——至少目前是。你创建了一段路径，也可以是形状、线条或其他对象，但它们不包含颜色或其他可见的内容。就像是用隐形

墨水在书写一样。在执行某些使其可见的操作之前，你的路径是看不到的。因此，下一步是调用 `CGContextStrokePath()` 函数告诉 Quartz 来绘制路径。该函数将使用我们之前设置的线宽和笔刷颜色对此路径进行涂色并使其可见。

### 16.2.2 坐标系统

在前面的代码中，我们将一对浮点数作为参数传递给 `CGContextMoveToPoint()` 函数和 `CGContextLineToPoint()` 函数。这些浮点数表示的是 Core Graphics 坐标系中的位置。点在此坐标系中的位置由其横坐标和纵坐标表示，我们通常用  $(x, y)$  来表达。环境的左上角为  $(0, 0)$ 。向下移动时， $y$  值会增加，而向右移动时， $x$  值会增加。

在上一段代码中，我们绘制了一条从  $(10, 10)$  到  $(20, 20)$  的对角线，看起来应该像图 16-1 所示的那样。

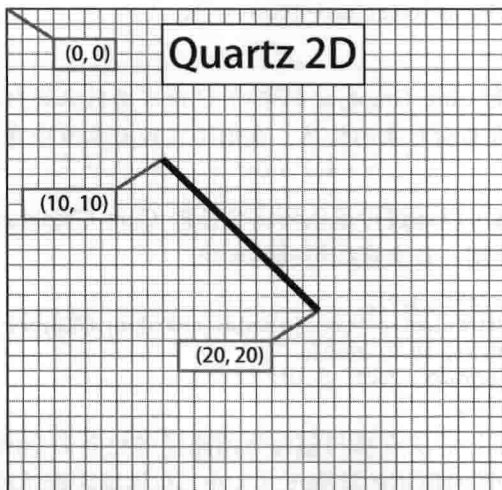


图 16-1 使用 Quartz 2D 的坐标系绘制一条直线

在 iOS 上使用 Quartz 绘图时，坐标系会是其中一个误区，因为它的垂直轴向与许多图形库使用的坐标系以及传统笛卡儿坐标系（由笛卡儿于 17 世纪制定）上下相反。在其他坐标系统中（比如 OpenGL 或者 OS X 版本的 Quartz），坐标  $(0, 0)$  位于左下角。若  $y$  轴坐标上升，你将向着环境或视图的顶端移动，如图 16-2 所示。

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值作为参数。而另一些 Quartz 函数则要求将该坐标值封装在 `CGPoint` 中，`CGPoint` 是一个包含了两个浮点值（即  $x$  和  $y$ ）的结构体。若要描述视图或其他对象的尺寸，Quartz 则会使用 `CGSize`，`CGSize` 也是一个拥有两个浮点值（即 `width` 和 `height`）的结构体。此外 Quartz 还声明了一个名为 `CGRect` 的数据类型，它能用于在坐标系中定义矩形。`CGRect` 中包含了两个元素：一个是名为 `origin` 的 `CGPoint`，它的  $x$  与  $y$  值确定了矩形的左上角位置；和一个名为 `size` 的 `CGSize`，它确定了矩形的宽度与高度。

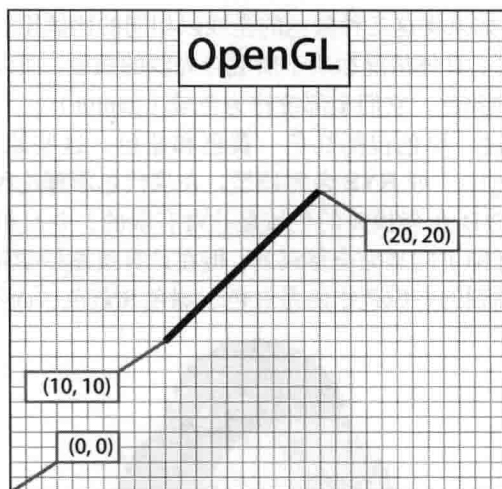


图 16-2 在许多图形库中（包括 OpenGL），从(10, 10)到(20, 20)绘制所生成的直线应该看起来如此图中所示，而不是图 16-1 的那种

### 16.2.3 指定颜色

颜色是绘图的一个重要部分，因此理解颜色在 iOS 上的实现机制是非常重要的。UIKit 框架提供了一个 Objective-C 类：UIColor。你不能在 Core Graphics 函数调用中直接使用 UIColor 对象。因为 UIColor 仅仅是 CGColor 的一个容器（这也正是 Core Graphics 需要的），因此你可以像之前那段代码展示的那样，使用 UIColor 实例的 CGColor 属性来获取一个 CGColor 引用：

```
CGContextSetStrokeColorWithColor(context, [UIColor redColor].CGColor);
```

我们使用名为 redColor 的便利方法创建了一个 UIColor 实例，然后获取它的 CGColor 属性值，并将该值传递给了函数。

#### 1. iOS 设备上的色彩理论

在现代计算机图形学中，显示在屏幕上所有颜色的数据都会被保存下来，并遵循色彩模型（color model）。色彩模型有时也被称为色彩空间（color space），是将现实世界中的颜色以计算机可以理解的数值来表示的方法。一种常见的方法是使用四种元素（红、绿、蓝和不透明度）来表示颜色。在 Quartz 中，这些值都是 CGFloat 类型（与 float 类型一样都是占 4 字节的浮点值），并且只能在 0.0 到 1.0 之间取值。

**说明** 取值范围为 0.0 到 1.0 的浮点值通常被称为限定浮点变量（clamped floating-point variable），有时简称为限定变量（clamp）。

红绿蓝这三个元素很容易理解，因为它们代表加法三原色（additive primary colors）和 RGB

色彩模型（参见图 16-3）。如果将这三种颜色的光线以相同的比例投射在一处，在眼前出现的结果会是白色或某种灰色光影，具体情况取决于所混合光线的强度。而以不同的比例混合这三种颜色，将会生成一系列不同的颜色，它们全部统称为色域（gamut）。

你可能在小学里学到过，原色包括红色、黄色和蓝色。这些原色被称为历史减法三原色（historical subtractive primaryies）或 RYB 色彩模型，在现代色彩理论种很少采用，在计算机图形学中也几乎从未使用过。RYB 色彩模型的色域是非常有限的，并且很难用数学的方式来表示。我并没有说你的小学美术老师讲的是完全错误的，但在计算机图形学领域中，不应该继续依赖过去的知识。因此在本书中提到的三原色指的是红色、绿色和蓝色，而不是红色、黄色和蓝色。

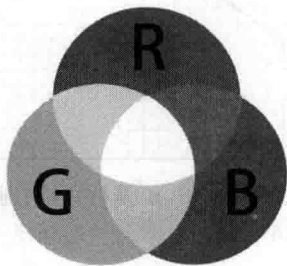


图 16-3 组成 RGB 色彩模型的加法三原色简单图示

除了红色、绿色和蓝色之外，Quartz 还使用了另一个叫做 **alpha** 的颜色元素，它表示颜色的不透明度。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** 的值为 1.0，则绘制的颜色为 100% 不透明，它下面的任何颜色都无法看清楚。如果它的值小于 1.0，则下面的颜色将能透过并混合上层的颜色显示出来。如果 **alpha** 的值为 0.0，则当前颜色完全不可见，下面的颜色也会全部透视出来。有时会将使用了 **alpha** 值的色彩模型称为 **RGBA 色彩模型**，虽然从技术上来说，**alpha** 实际上并不是颜色的一部分，它只是定义了绘制时颜色与其他色彩的交互方式。

## 2. 其他色彩模型

虽然在计算机图形学中最常用的是 RGB 模型，但是它不是唯一的色彩模型。比如下面的几项模型也得到了使用：

- 色相、饱和度、色明度（HSV）
- 色调、饱和度、亮度（HSL）
- 蓝绿色、洋红色、黄色、黑色（CMYK），被应用于四色平版印刷
- 灰度级

不仅如此，这些模型（包括 RGB 模型）还都有各种不同的版本变化，从而令情况变得更加混乱。

幸运的是，对于大多数操作来说，我们不必担心所使用的色彩模型。我们只需从 `UIColor` 对象中调用 `CGColor` 属性，在大多数情况下 Core Graphics 将会处理任何需要的数值转换。

## 3. 颜色便利方法

`UIColor` 提供了许多便利方法，可以返回被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` 方法将颜色初始化为红色。

这些便利方法创建的 `UIColor` 实例对象恰好大部分采用的都是 RGBA 色彩模型。唯一的例外则表示灰度级的预定义 `UIColor` (例如 `blackColor`、`whiteColor` 和 `darkGrayColor`) , 它们只根据白色程度和不透明度来定义。在本书的示例中, 我们不会使用到它, 因此可以假定现在使用的是 RGBA 模型。

如果你需要更精确地控制颜色, 可以不使用这些与颜色名称相关的便利方法, 而是通过指定所有的四个元素来创建某个颜色。以下是示例代码:

```
UIColor *red = [UIColor colorWithRed:1.0 green:0.0 blue:0.0 alpha:1.0];
```

## 16.2.4 在环境中绘制图像

通过 Quartz 可以在环境中直接绘制图像。这是 Objective-C 类 `UIImage` 的另一种用法, 你可以使用它作为 Core Graphics 中数据结构 (`CGImage`) 的替代方式。此 `UIImage` 类包含了将图像绘制到当前环境中的方法。你需要使用下面任意一种方法来确定此图像出现在环境中的位置:

- ❑ 指定一个 `CGPoint` 以确定图像的左上角;
- ❑ 指定一个 `CGRect` 以确定图像的范围, 并根据需要调整到适当的尺寸大小。

你可以在当前环境中绘制一个 `UIImage`, 代码如下所示:

```
UIImage *image; // assuming this exists and points at a UIImage instance
CGPoint drawPoint = CGPointMake(100.0, 100.0);
[image drawAtPoint:drawPoint];
```

16

## 16.2.5 绘制形状: 多边形、直线和曲线

Quartz 提供了许多函数, 这些函数使我们更加容易创建复杂的形状。如果需要绘制矩形或多边形, 并不需要计算角度、绘制线段或者执行任何数学运算。你只需要调用一个 Quartz 函数就能帮你完成工作。比如说, 假如要绘制一个椭圆形, 你需要定义一个刚好能包住它的矩形, 接下来的全都可以交给 Core Graphics 来完成:

```
CGRect theRect = CGRectMake(0, 0, 100, 100);
CGContextAddEllipseInRect(context, theRect);
CGContextDrawPath(context, kCGPathFillStroke);
```

对于矩形也是类似的方法。此外还有许多方法可用于创建更为复杂的形状, 比如弧形和贝塞尔路径 (Bezier path)。

**说明** 本章的示例中不会介绍复杂的形状。若要了解有关 Quartz 中弧形和 Bezier 路径的详细信息, 请查看位于 <http://developer.apple.com/documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/> 链接上 iOS Dev Center 中的 Quartz 2D Programming Guide 文档, 也可以在 Xcode 中的在线文档中找到:

<https://developer.apple.com/library/ios/documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/>。

### 16.2.6 Quartz 2D 样例：图案、渐变色、虚线图

Quartz 提供了许多强大的工具。例如，Quartz 除了纯色以外，还支持使用渐变色填充多边形。此外除了能够绘制实线，也可以使用各种各样的虚线图形。请浏览图 16-4 中截取自苹果公司 QuartzDemo 官方样例代码实现的屏幕截图，以了解 Quartz 可以达到的各种图形效果。

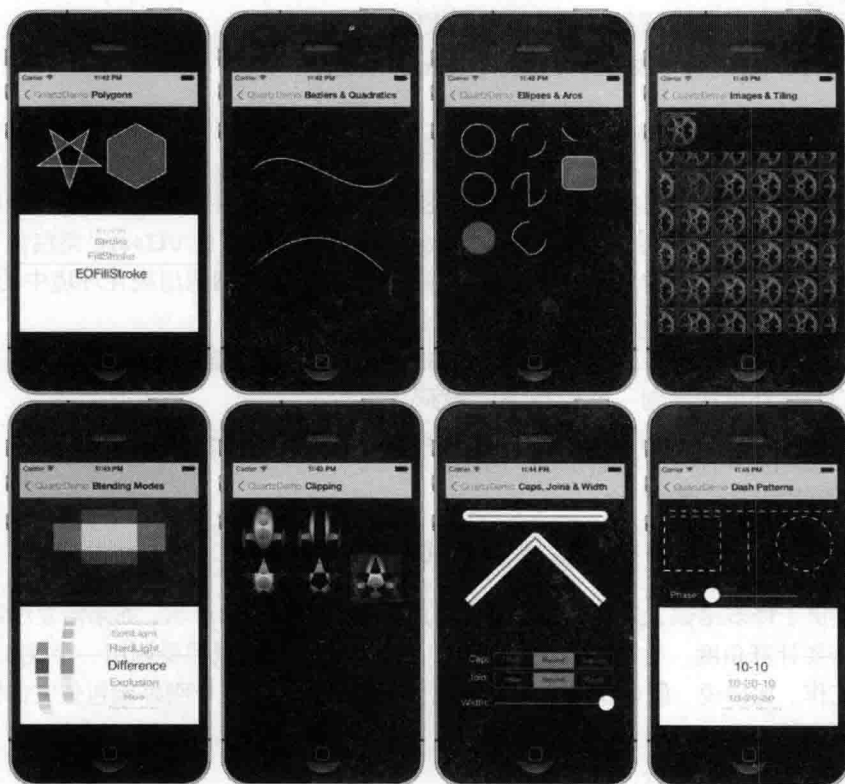


图 16-4 一些 Quartz 2D 的示例，来自于苹果公司提供的 QuartzDemo 样例项目

现在你已经基本理解了 Quartz 的工作原理以及它的功能，那么马上开始使用它吧。

## 16.3 QuartzFun 应用程序

我们接下来的应用程序是一个简单的绘图程序（参见图 16-5）。为了能让你更好地结合之前所描述的概念并获得实际的体验效果，我们将使用 Quartz 来构建此应用程序。

该应用程序的顶部和底部分别有一个工具条，它们各包含一个分段控件。顶部的控件可以用于更改绘图颜色，底部的控件可以用于更改要绘制的形状。当你触摸并拖动屏幕，将使用所选的颜色绘制所选的形状。为了尽量降低应用程序的复杂性，每次都只能绘制一个形状。

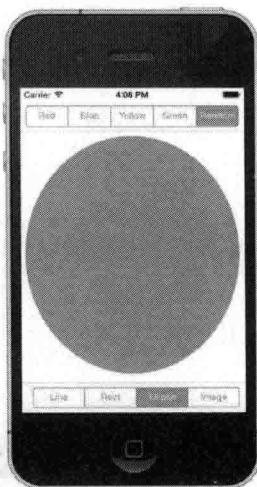


图 16-5 本章的绘图应用程序示例正在运行中

### 16.3.1 构建QuartzFun应用程序

在 Xcode 中, 使用 Single View Application 模板创建一个新的 iPhone 项目, 并将其命名为 QuartzFun。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。因为我们将在视图中执行自定义绘图, 所以需要创建一个自定义的 UIView 子类。在该子类中, 我们将通过重载 drawRect: 方法进行绘图。

选中 QuartzFun 文件夹 (即应用程序委托与视图控制器文件当前位于的文件夹), 按下 command+N 打开新文件向导, 然后从 Cocoa Touch 分栏中选择 Objective-C class 图标。将新类命名为 BIDQuartzFunView, 并设置其为 UIView 的子类。

与之前的项目一样, 我们还需要定义一些常量, 不过这次定义的常量是多个类所需要的。因此我们还要创建一个只针对常量的头文件。

再次选中 QuartzFun 组, 并按下 command+N 打开新文件向导, 从 C and C++ 边栏中选择 Header File 模板, 并将其命名为 BIDConstants.h。

我们还需要再创建两个文件。查看图 16-5, 你可以看到我们提供了一个选择随机颜色的选项, 但 UIColor 没有提供返回随即颜色的方法, 因此我们必须编写代码来实现功能。我们可以将该代码放在控制器类中, 不过作为有经验的 Objective-C 程序员, 我们还是决定通过 UIColor 类的类别特性来处理这段代码。

再一次选中 QuartzFun 文件夹并按下 command+N 调出新文件向导。从 Cocoa Touch 分类中选择 Objective-C 类别, 然后单击 Next 按钮。出现提示框时将该目录命名为 BIDRandom, 并设置其为 UIColor 类的类别。点击 Next 按钮, 最后将文件保存到你的项目文件夹中。

现在你应该已经拥有两个名称分别为 UIColor+BIDRandom.h 和 UIColor+BIDRandom.m 的类别文件。



## 1. 创建随机颜色

首先处理这两个类别文件。在 UIColor+BIDRandom.h 文件中, 添加以下代码:

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

```
@interface UIColor (BIDRandom)
+ (UIColor *)randomColor;
@end
```

然后切换到 UIColor+BIDRandom.m 文件中并添加以下内容:

```
#import "UIColor+BIDRandom.h"
```

```
#define ARC4RANDOM_MAX 0x100000000LL
```

```
@implementation UIColor (BIDRandom)
+ (UIColor *)randomColor {
    CGFloat red = (CGFloat)arc4random() / (CGFloat)ARC4RANDOM_MAX;
    CGFloat blue = (CGFloat)arc4random() / (CGFloat)ARC4RANDOM_MAX;
    CGFloat green = (CGFloat)arc4random() / (CGFloat)ARC4RANDOM_MAX;
    return [UIColor colorWithRed:red green:green blue:blue alpha:1.0];
}
@end
```

这里非常简单。我们使用 arc4random() 函数为每个颜色元素生成一个随机浮点数, 然后再除以某个表示 arc4random() 可以返回的最大值的常量。这样做的话, 就能保证每个元素的值都是 0.0 到 1.0 之间的随机数。我们接下来使用这三个原色比例创建一个新的颜色。并设置 alpha 值为 1.0, 这样新生成的颜色就会是完全不透明的。

## 2. 定义应用程序常量

接下来, 我们将为用户在分段控制器上会选到的每个选项定义相对应的常量。单击 BIDConstants.h 文件并添加以下代码:

```
#ifndef QuartzFun_BIDConstants_h
#define QuartzFun_BIDConstants_h
```

```
typedef NS_ENUM(NSInteger, ShapeType) {
    kLineShape = 0,
    kRectShape,
    kEllipseShape,
    kImageShape
};
```

```
typedef NS_ENUM(NSInteger, ColorTabIndex) {
    kRedColorTab = 0,
    kBlueColorTab,
    kYellowColorTab,
    kGreenColorTab,
    kRandomColorTab
};
```

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

```
#endif
```



为了使代码更具有可读性, 我们使用 `typedef` 关键字和 `NS_ENUM` 宏声明了两个枚举类型。一个类型用于表示应用程序中可用的形状选项, 另一个类型用于表示应用程序中可用的各种颜色选项。这些常量中存放的值与我们在应用程序中创建的两个分段控件上的各个选项都能对应起来。

**说明** 可能你以前没有见过 `#ifndef` 这种形式, 这条编译指令目的首先是检测 `QuartzFun_BIDConstants.h` 是否已经定义了, 如果还没有定义, 那么定义它。那为什么不直接用 `#define` 呢? 这是因为如果一个 `.h` 文件被导入了一次以上 (无论是直接导入或通过其他 `.h` 文件间接导入), 该指令也只会执行一次。通常在编译使用了 `#import` 指令的 Objective-C 代码时不需要采用这种方式, 不过这是使用 C 语言以及旧版 `#include` 指令进行开发时常用的方案, 可以有效地避免多次导入同一个头文件。

### 3. 实现 QuartzFunView 框架

由于我们将在 `UIView` 的某个子类中进行绘图, 因此我们要将这个类的所有内容都设置好, 不过关键的绘图代码要最后再填上。首先单击 `BIDQuartzFunView.h` 文件并在起始处添加以下代码:

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

@interface BIDQuartzFunView : UIView
@property (assign, nonatomic) CGPoint firstTouchLocation;
@property (assign, nonatomic) CGPoint lastTouchLocation;
@property (assign, nonatomic) ShapeType shapeType;
@property (assign, nonatomic) BOOL useRandomColor;
@property (strong, nonatomic) UIColor *currentColor;
@property (strong, nonatomic) UIImage *drawImage;
@end
```

我们做的第一件事情就是导入刚刚创建的 `BIDConstants.h` 头文件, 这样就能够使用这些枚举值了。然后声明属性变量。前两个变量将跟踪用户在屏幕上拖动的手指。我们会将用户第一次触摸屏幕的位置存放在 `firstTouchLocation` 变量中, 并将拖动时手指的当前位置和拖动结束时手指的位置存放在 `lastTouchLocation` 变量中。绘图代码通过使用这两个变量来确定在什么位置绘制所需的形状。

接下来, 我们定义一个 `ShapeType` 变量来存放用户想要绘制的形状, 和一个记录用户是否想要使用随机颜色的布尔值变量。然后我们定义一个 `UIColor` 属性变量来存放当前所选的颜色。最后我们定义一个 `UIImage` 属性变量, 当用户选择底部工具栏最右边选项时, 可用来控制将要绘制在屏幕上的图片 (参考图 16-6)。注意前四个属性都是底层 C 类型变量, 所以使用 `assign` 关键字声明, 而后两个对象是通过 `strong` 关键字声明的。

切换到 `BIDQuartzFunView.m` 文件, 我们要在这个文件中作几处改动。首先, 导入 `UIColor+BIDRandom.h` 头文件, 在文件起始处的其他 `import` 指令下面添加这行代码就可以生成随机颜色了:

```
#import "UIColor+BIDRandom.h"
```

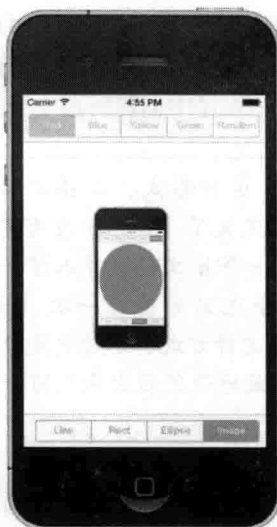


图 16-6 如果在屏幕上绘制 UIImage，颜色控件则会消失。你能告诉我屏幕上的小 iPhone 里运行的是哪个应用程序吗？

继续修改这个实现文件。在模板中已经替我们写好了一个名叫 `initWithFrame:` 的方法，不过我们并不需要用到它。请记住 nib 文件和分镜中的对象实例是作为归档对象存储的，这与我们在第 13 章中归档和加载到磁盘所使用的机制相同。因此，从 nib 文件或分镜中加载对象实例时，`initWithFrame:` 方法都不会被调用。实际被调用的是 `initWithCoder:` 方法，所以需要在这里添加所有的初始化代码。在这个示例中，我们将颜色的初始值设置为红色，`useRandomColor` 变量初始值为 NO，并加载本章后面需要绘制的图像文件。请删除已有的 `initWithFrame:` 模板代码，将其替换为如下方法：

```
- (id)initWithCoder:(NSCoder*)coder {
    if (self = [super initWithCoder:coder]) {
        _currentColor = [UIColor redColor];
        _useRandomColor = NO;
        _drawImage = [UIImage imageNamed:@"iphone.png"];
    }
    return self;
}
```

`initWithCoder` 方法之后还要添加三个能响应用户触摸事件的方法：

**#pragma mark - Touch Handling**

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    if (self.useRandomColor) {
        self.currentColor = [UIColor randomColor];
    }
    UITouch *touch = [touches anyObject];
    self.firstTouchLocation = [touch locationInView:self];
    self.lastTouchLocation = [touch locationInView:self];
}
```

```

    [self setNeedsDisplay];
}

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

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    self.lastTouchLocation = [touch locationInView:self];
    [self setNeedsDisplay];
}

```

这三个方法都继承自 `UIView` 类（实际上是在 `UIView` 的父类 `UIResponder` 中声明的），可以通过重载这些方法来识别用户触摸了屏幕的什么位置。以下是它们的执行过程。

- ❑ `touchesBegan:withEvent:` 方法会在用户的手指第一次触摸到屏幕上时被调用。在这个方法中，假如用户选择了随机颜色选项，我们会使用之前为 `UIColor` 新增的 `randomColor` 方法改变当前颜色。然后存储当前坐标，这样就能知道用户第一次触摸到屏幕的位置。之后我们调用 `setNeedsDisplay` 方法将视图标记为需要重新绘制的。
- ❑ `touchesMoved:withEvent:` 方法会当用户的手指在屏幕上拖动时被持续调用。我们在这里要作的只是每次都要在 `lastTouchLocation` 变量中存储最新的当前位置，并标记为需要重新绘制屏幕。
- ❑ `touchesEnded:withEvent:` 方法会在用户的手指离开屏幕时被调用。与我们在 `touchesMoved:withEvent:` 方法中所做的一样，只需要在 `lastTouchLocation` 变量中存储最终的位置并将视图标记为需要重新绘制。

即便你不能完全理解这些代码的内容也不必担心。我们将在第 18 章中详细地讲解触摸事件以及这些触摸方法的工作原理。

完成应用程序框架并且能运行之后，我们要回来继续完善这个类。我们将会用 `drawRect:` 方法中实现应用程序的主要功能，这段代码当前是被注释的状态，还没有写任何内容。我们先完成应用程序的设置，之后将在这里添加绘图代码。

#### 4. 向视图控制器中添加输出接口和操作方法

开始绘图之前，我们需要向图形用户界面（GUI）中添加分段控件，然后连接操作方法和输出接口。单击 `Main.storyboard` 图标来设置这些内容。

要做的第一件事就是修改视图所代表的类。在文件大纲中展开场景的内容，并在里面的视图控制器中单击选中 `View` 图标。按下 `command+3` 打开身份检查器，将 `class` 文本框中的内容由 `UIView` 改成 `BIDQuartzFunView`。

在对象库中查找导航栏（`Navigation Bar`）。注意，你要找到的是导航栏，而不是导航控制器（`Navigation Controller`）。我们需要让导航栏位于视图上方，请将导航栏放在靠近视图顶部仅低于状态栏的位置。请注意你无法在这里看到实际的状态栏，因此你需要使用尺寸检查器将它的 `y` 值

设置为 20 点数。

接下来，在库中找到分段控件（Segmented Control）并将其直接拖动到导航栏的顶部，请放在中间位置，而不要靠左或靠右（参见图 16-7）。

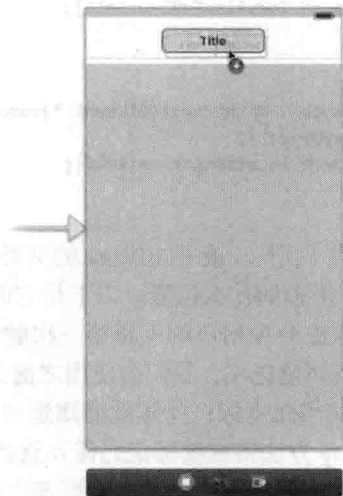


图 16-7 拖出一个分段控件，确保放在导航栏的顶部

放下该控件之后，它应该仍然为选中状态。拖动分段控件任何一侧的缩放点来调整它的大小，使它能占据整个导航栏的宽度。你不会看到任何蓝色引导线，不过界面构建器会限制它在导航栏中的尺寸，因此只需要拖动它直到不能再放大为止。

确保分段控件是选中状态，打开属性检查器，并将分段数量从 2 更改为 5。依次双击各个分段，将它们的标签分别改为（从左至右）Red、Blue、Yellow、Green 和 Random。此时，View 视图应该类似于图 16-8 所示。

如果辅助编辑器界面还没有显示，请现在打开它，然后在文件跳转栏中选择 BIDViewController.m。现在按住 control 键并将鼠标指针从 dock 栏中的分段控件拖动到右边的 BIDViewController.m 文件上的空白处（在 @interface 与靠近顶端用来表示类扩展的 @end 语句之间）。当你的指针停留在 @interface 和 @end 声明之间时，请释放鼠标以创建一个新的输出接口。给新的输出接口命名为 colorControl，并保留所有其他选项为默认值。请确保你是从分段控件开始拖动的，而不是导航栏或导航按钮。

接着，我们添加一个操作方法。再次按住 control 键，仍然从分段控件拖向之前的文件，停留在文件底部的 @end 声明的上面。这次使用 changeColor: 作为操作方法的名称，下拉列表默认的值应该刚好就是你需要的 Value Changed 事件，此外还须将 Type 值设置为 UISegmentedControl。

现在，请在库中找到一个工具栏（Toolbar，不是 Navigation Bar 导航栏）并将其拖出到视图窗口的底部。库中的工具栏上有一个我们不需要的按钮，因此请选中按钮并按下键盘上的 delete 键。这样按钮就会消失，只留下一个空白的工具栏。

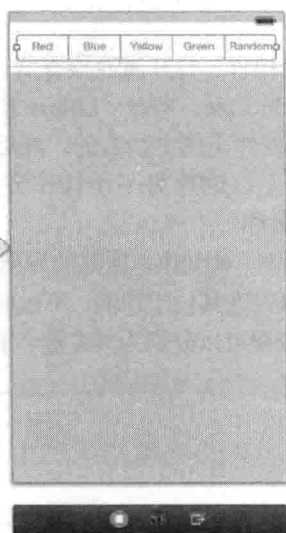


图 16-8 完成后的导航栏

放好工具栏后，请再拖出另一个分段控件并将其放置在工具栏上（参见图 16-9）。

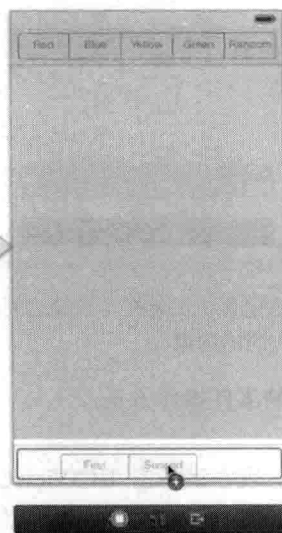


图 16-9 视图窗口底部显示一个工具栏，工具栏上拖动放置一个分段控件

现在来调整分段控件的尺寸。在 dock 栏中选择 Bar Button 项目（其子项目即分段控件），编辑区域中分段控件的右侧将出现调节控制柄。拖动该手柄来调整分段控件的尺寸，使其占满整个工具栏，只在左右两侧留下一小部分空间。界面构建器不会显示引导线，也不会像前面的导航栏那样限制你将分段控件大小拉伸到工具栏之外，因此你需要尽量小心地将分段控件放大

到合适的尺寸。

接着，在 dock 栏中选择分段控件，打开属性检查器，并将分段数量由 2 更改为 4。现在双击每个分段并将它们的名称依次改为 Line、Rect、Ellipse 和 Image。

完成这些之后，请确保分段控件是选中状态的，然后按住 control 键并将鼠标指针从分段控件拖动到 BIDViewController.m 上以创建另一个操作方法。将连接类型改为 Action，并使用 changeShape: 作为新的操作方法名称。

最后我们要在这里为界面创建一些约束。顶部的导航栏是正常的，因为我们在构建应用程序的时候，Xcode 会默认创建约束以保证其位于顶端。不过底部的工具栏还需要一些调整，选中工具栏之后点击编辑区域下方的 Pin 按钮可以分别为其左边、右边以及底端添加约束（参见图 16-10）。

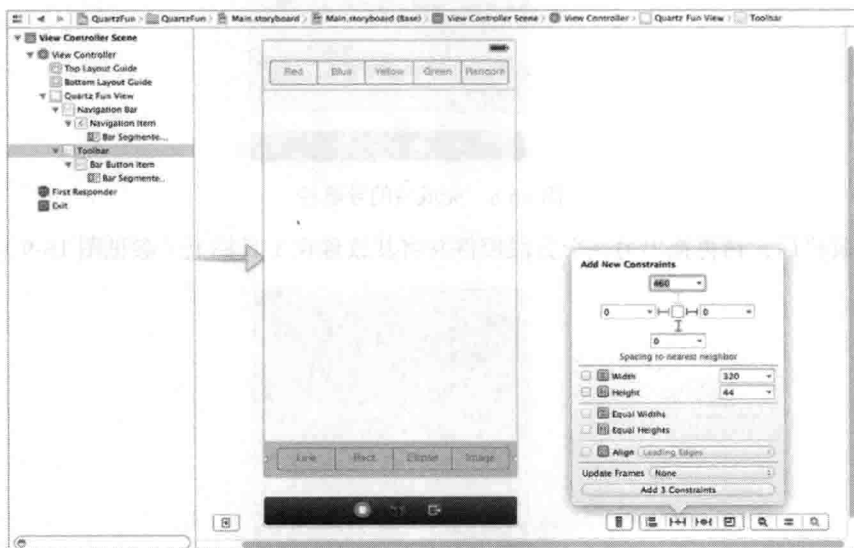


图 16-10 请注意浮动窗口也包含了工具栏与顶端之间距离的设置项，我们可以无视它，只添加其他三个方向的约束

接下来的任务就是实现我们的事件操作方法。

### 5. 实现操作方法

保存分镜并关闭辅助编辑器。现在请单击 BIDViewController.m 文件。首先我们需要导入常量文件，以便访问之前定义的枚举值。因为我们要与自定义视图之间进行交互，所以还需要导入它的头文件。在 BIDViewController.m 文件顶部，紧接着已有的 import 语句之后，添加如下代码：

```
#import "BIDConstants.h"
#import "BIDQuartzFunView.h"
```

然后找到 Xcode 为你自动创建的 changeColor: 方法代码块的位置，并添加如下代码：

```
-(IBAction)changeColor:(id)sender {
    UISegmentedControl *control = sender;
    ColorTabIndex index = [control selectedSegmentIndex];
```

```

    BIDQuartzFunView *funView = (BIDQuartzFunView *)self.view;

    switch (index) {
        case kRedColorTab:
            funView.currentColor = [UIColor redColor];
            funView.useRandomColor = NO;
            break;
        case kBlueColorTab:
            funView.currentColor = [UIColor blueColor];
            funView.useRandomColor = NO;
            break;
        case kYellowColorTab:
            funView.currentColor = [UIColor yellowColor];
            funView.useRandomColor = NO;
            break;
        case kGreenColorTab:
            funView.currentColor = [UIColor greenColor];
            funView.useRandomColor = NO;
            break;
        case kRandomColorTab:
            funView.useRandomColor = YES;
            break;
        default:
            break;
    }
}

```

16

这段代码非常简单，只是检查了究竟选中了哪个分段按钮，并根据用户的选择创建了新的颜色作为当前的绘制颜色。为了使编译器不感到疑惑，我们将 view 的类型（在父类中被声明为 UIView 的实例）强制转换为 QuartzFunView。然后我们设置了 currentColor 属性，这样该类在绘制时就能知道应该使用哪种颜色了——除非你选择的是随机颜色。如果选择的是随机颜色，视图将检索 useRandomColor 属性，所以我们要分别为每一个选项设置一个合适的值。由于所有绘制代码都包含在视图自身中，所以我们不需要在这个方法中执行任何其他操作。

接下来找到已有的 changeShape: 方法代码块，并添加以下代码：

```

- (IBAction)changeShape:(id)sender {
    UISegmentedControl *control = sender;
    [(BIDQuartzFunView *)self.view setShapeType:[control
                                                    selectedSegmentIndex]];

    if ([control selectedSegmentIndex] == kImageShape) {
        self.colorControl.enabled = NO;
    } else {
        self.colorControl.enabled = YES;
    }
}

```

这个方法所要做的就是根据所选择的分段控件按钮来设置图形形状。你还记得 ShapeType 枚举类型吗？枚举中的四个元素分别对应于应用程序视图底部的四个工具栏分段。我们将形状设置为当前分段按钮所选代表的形状，并根据是否选择了 Image 分段来显示或隐藏 colorControl 控件。

**说明** 你可能想知道为什么我们要将导航栏放置在视图的顶部，而把工具栏放置在视图的底部。根据苹果公司发布的《人机界面指南》(*Human Interface Guidelines*)，导航栏是专门放在屏幕顶部的，而工具栏则是专门放在底部的。如果你阅读了界面构建器的库窗口中关于工具栏和导航栏的描述，你就会明白这样做的设计意图。

请编译并运行应用程序，以确保以上所有步骤都已正常完成。目前你还不能在屏幕上绘制图形，不过分段控件已经能够正常工作了。点击底部的 Image 分段控件按钮时，颜色控件便会消失。

到现在为止一切都顺利，我们开始剩下的绘制内容吧。

### 16.3.2 添加 Quartz 2D 绘制代码

现在要添加执行绘图的代码。我们将绘制一条直线、一些图形和一张图片。整个工作将循序渐进地进行，先编写一小段代码，然后运行应用程序以查看代码所实现的内容。

#### 1. 绘制直线

首先绘制最简单的一条直线。选择 BIDQuartzFunView.m 文件，然后将被注释掉的 drawRect: 方法改为如下所示：

```
- (void)drawRect:(CGRect)rect {
    CGContextRef context = UIGraphicsGetCurrentContext();

    CGContextSetLineWidth(context, 2.0);
    CGContextSetStrokeColorWithColor(context, self.currentColor.CGColor);

    switch (self.shapeType) {
        case kLineShape:
            CGContextMoveToPoint(context,
                                   self.firstTouchLocation.x,
                                   self.firstTouchLocation.y);
            CGContextAddLineToPoint(context,
                                     self.lastTouchLocation.x,
                                     self.lastTouchLocation.y);
            CGContextStrokePath(context);
            break;
        case kRectShape:
            break;
        case kEllipseShape:
            break;

        case kImageShape:
            break;
        default:
            break;
    }
}
```

首先要获取当前环境的引用，这样就能够知道在哪里进行绘制。



```
CGContextRef context = UIGraphicsGetCurrentContext();
```

接下来将直线的宽度设置为 2.0，这意味着画的任何直线都是 2 个点数宽：

```
CGContextSetLineWidth(context, 2.0);
```

随后设置所画直线的颜色。由于 UIColor 类的 CGColor 属性正是函数所需的参数，因此使用 currentColor 实例变量的这个属性将正确的颜色传递给该函数：

```
CGContextSetStrokeColorWithColor(context, self.currentColor.CGColor);
```

使用 switch 语句根据每个形状的类型跳转到对应的代码位置。如之前所说，先从处理绘制直线的 kLineShape 代码开始，完成之后再依次为示例中其余的形状添加代码：

```
switch (self.shapeType) {
    case kLineShape:
```

为了绘制一条直线，我们要让图形环境在用户触摸的第一个位置开始创建路径。请回忆起曾在 touchesBegan: 方法中存储了这个值，因此能获取最近一次触摸或拖动时的起点位置。

```
CGContextMoveToPoint(context,
                    self.firstTouchLocation.x,
                    self.firstTouchLocation.y);
```

接下来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停留在屏幕上，则 lastTouch 变量包含的是用户手指的位置。如果用户的手指离开了屏幕，则 lastTouch 变量包含的是用户手指离开时的位置。

```
CGContextAddLineToPoint(context,
                    self.lastTouchLocation.x,
                    self.lastTouchLocation.y);
```

然后要绘制出这条路径。以下函数会将画的直线以之前设置好的颜色与宽度绘制出来。

```
CGContextStrokePath(context);
```

随后只需要完成此 switch 语句就可以了，代码如下所示：

```
        break;
    case kRectShape:
        break;
    case kEllipseShape:
        break;
    case kImageShape:
        break;
    default:
        break;
}
```

目前就是这些内容。此时你应该能够再次进行编译并运行应用程序了。Rect、Ellipse 和 Image 选项都不可用，不过你可以使用选中的任意颜色来很好地绘制出直线（参见图 16-11）。

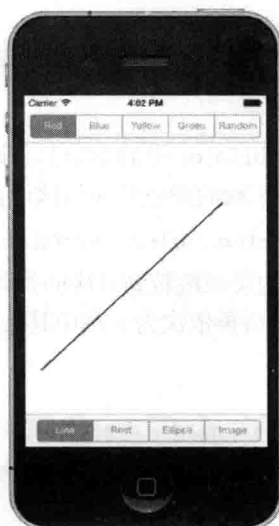


图 16-11 应用程序中绘制直线的部分现在已经完成。在该图中，我们使用的是红色

## 2. 绘制矩形和椭圆形

这次我们同时实现绘制矩形和椭圆形的代码，因为 Quartz 实现这两种对象的方法基本类似。将以下粗体显示的代码添加到已有的 `drawRect:` 方法中：

```
(void)drawRect:(CGRect)rect {
    CGContextRef context = UIGraphicsGetCurrentContext();

    CGContextSetLineWidth(context, 2.0);
    CGContextSetStrokeColorWithColor(context, self.currentColor.CGColor);

    CGContextSetFillColorWithColor(context, self.currentColor.CGColor);
    CGRect currentRect = CGRectMake(self.firstTouchLocation.x,
                                   self.firstTouchLocation.y,
                                   self.lastTouchLocation.x -
                                   self.firstTouchLocation.x,
                                   self.lastTouchLocation.y -
                                   self.firstTouchLocation.y);

    switch (self.shapeType) {
        case kLineShape:
            CGContextMoveToPoint(context,
                                self.firstTouchLocation.x,
                                self.firstTouchLocation.y);
            CGContextAddLineToPoint(context,
                                self.lastTouchLocation.x,
                                self.lastTouchLocation.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;
}
}

```

由于想要给椭圆形和矩形涂上纯色，所以要添加一个方法调用，并使用 `currentColor` 变量来设置填充颜色：

```
CGContextSetFillColorWithColor(context, self.currentColor.CGColor);
```

接下来声明一个 `CGRect` 变量，做这个步骤是因为矩形和椭圆形都是基于矩形绘制的。我们将使用 `currentRect` 存放的值来描述用户拖出的矩形信息。请记住，`CGRect` 内包含两个成员变量：`size` 以及 `origin`。通过 `CGRectMake()` 函数，我们可以通过指定 `x`、`y`、`width` 和 `height` 的值来创建一个 `CGRect`，我们就用它来生成矩形。

创建矩形的代码非常简单。我们使用存储在 `firstTouch` 中的点创建左上角的起点。然后通过计算两点之间 `x` 值与 `y` 值的差来获取尺寸大小。请注意，根据拖动方向，其中一个或两个尺寸的值可能会为负数，不过这没关系。具有负值的 `CGRect` 将从起点按相反方向渲染（对于负宽度值，向左绘制，而负高度值则向上绘制）：

```

CGRect currentRect = CGRectMake(self.firstTouchLocation.x,
                                self.firstTouchLocation.y,
                                self.lastTouchLocation.x -
                                    self.firstTouchLocation.x,
                                self.lastTouchLocation.y -
                                    self.firstTouchLocation.y);

```

将矩形的内容定义好之后，只需要简单调用两个函数就可以绘制矩形或椭圆形了。其中一个函数是在我们定义的 `CGRect` 中绘制矩形或椭圆形，另一个函数用来绘画并填充它：

```

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，试着用一下矩形和椭圆形工具来看看你的杰作。别忘了多换几种颜色试试，以及随机颜色。

### 3. 绘制图像

我们要做的最后一件事就是绘制图像。16-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项目的 Image.xcassets 文件中，或者也可以使用其他想要的 png 文件，只要记住将下面代码中的文件名改为所选图像的名称即可。

向 drawRect: 方法中添加以下代码：

```
(void)drawRect:(CGRect)rect {
    CGContextRef context = UIGraphicsGetCurrentContext();

    CGContextSetLineWidth(context, 2.0);
    CGContextSetStrokeColorWithColor(context, self.currentColor.CGColor);

    CGContextSetFillColorWithColor(context, _currentColor.CGColor);
    CGRect currentRect = CGRectMake(self.firstTouchLocation.x,
                                    self.firstTouchLocation.y,
                                    self.lastTouchLocation.x -
                                        self.firstTouchLocation.x,
                                    self.lastTouchLocation.y -
                                        self.firstTouchLocation.y);

    switch (self.shapeType) {
        case kLineShape:
            CGContextMoveToPoint(context,
                                   self.firstTouchLocation.x,
                                   self.firstTouchLocation.y);
            CGContextAddLineToPoint(context,
                                     self.lastTouchLocation.x,
                                     self.lastTouchLocation.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 = self.drawImage.size.width / 2;
            CGFloat verticalOffset = self.drawImage.size.height / 2;
            CGPoint drawPoint = CGPointMake(self.lastTouchLocation.x -
                                              horizontalOffset,
                                              self.lastTouchLocation.y -
                                              verticalOffset);

            [self.drawImage drawAtPoint:drawPoint];
            break;
        }
        default:
            break;
    }
}
```

**说明** 注意在 `switch` 语句中，我们将 `case kImageShape:` 下的代码用花括号圈起来了。这是因为在 `case` 语句下第一行的变量声明会让编译器报错。使用花括号可以告诉编译器停止当前的警告。也可以通过改在 `switch` 语句之前声明 `horizontalOffset` 变量来避免这个问题，不过为了保持代码之间的相关性，我们还是使用了如上方法。

首先要计算该图像的中心，因为我们希望绘制的图像以用户最后触摸的点为中心。如果不这样调整的话，则会以用户的手指作为左上角绘制图像，当然这也是一种方案。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 = self.drawImage.size.width / 2;
CGFloat verticalOffset = self.drawImage.size.height / 2;
CGPoint drawPoint = CGPointMake(self.lastTouchLocation.x -
                                horizontalOffset,
                                self.lastTouchLocation.y -
                                verticalOffset);
```

现在我们通知图像让它将自身绘制出来，以下代码将执行此工作：

```
[self.drawImage drawAtPoint:drawPoint];
```

16

### 16.3.3 优化QuartzFun应用程序

应用程序的功能如预想的那样，不过仍需要考虑进行一些优化。在这个简单的应用程序中，你不会注意到速度有所减慢，但如果是在早期 iOS 设备上运行更加复杂的应用程序，可能会看到一些延迟。

该问题由 `BIDQuartzFunView.m` 中的 `touchesMoved:` 和 `touchesEnded:` 方法引起。这两个方法都包含了下面这行代码：

```
[self setNeedsDisplay];
```

顾名思义，我们告诉视图有些内容发生了变化了，需要重新绘制自身。该代码能正常工作，但它导致整个视图被擦除并重新绘制，即使只有非常微小的更改也是如此。准备拖动新形状时，我们希望能擦除该屏幕，但并不想要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，可以改用 `setNeedsDisplayInRect:` 方法。它是一个 `UIView` 对象的方法，该方法会将视图区域的某一块矩形部分标记为需要重新绘制。通过使用此方法，可以仅标记受当前绘图操作影响而需要重新绘制视图的某一部分，从而提高效率。

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所有屏幕部分。假设用户触摸屏幕，并在屏幕上到处乱画，如果只重新绘制 `firstTouch` 和 `lastTouch` 之间的部分，则重新绘制之后屏幕上仍会残留许多我们不希望看到的内容。

解决办法是跟踪受 `CGRect` 实例变量中的特定拖动影响的整个区域。在 `touchesBegan:` 方法中，先将该实例变量重新设置仅为用户触摸的点。然后在 `touchesMoved:` 和 `touchesEnded:` 方法中，使用一个 `Core Graphics` 函数获取当前矩形和已存储矩形的并集，然后再存储所得到的矩形。我们

还需要使用它来指定需要重新绘制的视图部分。通过该解决办法，可以获得受当前拖动影响而变化的所有区域。

现在需要在 `drawRect:` 方法中计算当前矩形，以便绘制椭圆形和矩形。将该计算相关的代码移动到新的方法中，这样就可以在 3 个地方使用到它，以避免编写重复的代码。准备好了吗？开始吧。

对 BIDQuartzFunView.h 文件进行以下更改:

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

@interface BIDQuartzFunView : UIView
@property (assign, nonatomic) CGPoint firstTouchLocation;
@property (assign, nonatomic) CGPoint lastTouchLocation;
@property (assign, nonatomic) ShapeType shapeType;
@property (assign, nonatomic) BOOL useRandomColor;
@property (strong, nonatomic) UIColor *currentColor;
@property (strong, nonatomic) UIImage *drawImage;
@property (readonly, nonatomic) CGRect currentRect;
@property (assign, nonatomic) CGRect redrawRect;
@end
```

我们声明了一个名为 `redrawRect` 的 `CGRect` 变量,并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,该属性将返回之前在 `drawRect:` 方法中计算的矩形。

切换到 BIDQuartzFunView.m 文件，将以下代码添加到文件顶部（放在已有的@Synthesize 语句的下面）：

```
- (CGRect)currentRect {
    return CGRectMake (self.firstTouchLocation.x,
                      self.firstTouchLocation.y,
                      self.lastTouchLocation.x - self.firstTouchLocation.x,
                      self.lastTouchLocation.y - self.firstTouchLocation.y);
}
```

现在 `drawRect:` 方法中所有对 `currentRect` 变量的引用都改成 `self.currentRect` 属性，这样代码就会用到刚刚创建的新存取方法。

[illegible]

```

switch (self.shapeType) {
    case kLineShape:
        CGContextMoveToPoint(context,
                               self.firstTouchLocation.x,
                               self.firstTouchLocation.y);
        CGContextAddLineToPoint(context,
                                  self.lastTouchLocation.x,
                                  self.lastTouchLocation.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: {
        CGFloat horizontalOffset = self.drawImage.size.width / 2;
        CGFloat verticalOffset = self.drawImage.size.height / 2;
        CGPoint drawPoint = CGPointMake(self.lastTouchLocation.x -
                                         horizontalOffset,
                                         self.lastTouchLocation.y -
                                         verticalOffset);
        [self.drawImage drawAtPoint:drawPoint];
        break;
    }
    default:
        break;
}
}
}

```

16

还需要对 `touchesEnded:withEvent:` 和 `touchesMoved:withEvent:` 方法作一些修改。需要重新计算受当前操作影响的范围，并用它来指示需要重新绘制的视图部分。将已有的这两个方面替换为以下的新版本：

```

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    self.lastTouchLocation = [touch locationInView:self];

    if (self.shapeType == kImageShape) {
        CGFloat horizontalOffset = self.drawImage.size.width / 2;
        CGFloat verticalOffset = self.drawImage.size.height / 2;
        self.redrawRect = CGRectUnion(self.redrawRect,
                                       CGRectMake(self.lastTouchLocation.x -
                                                    horizontalOffset,
                                                    self.lastTouchLocation.y -
                                                    verticalOffset,
                                                    self.drawImage.size.width,
                                                    self.drawImage.size.height));
    } else {
        self.redrawRect = CGRectUnion(self.redrawRect, self.currentRect);
    }
}

```

```

self.redrawRect = CGRectInset(self.redrawRect, -2.0, -2.0);
[self setNeedsDisplayInRect:self.redrawRect];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    self.lastTouchLocation = [touch locationInView:self];

    if (self.shapeType == kImageShape) {
        CGFloat horizontalOffset = self.drawImage.size.width / 2;
        CGFloat verticalOffset = self.drawImage.size.height / 2;
        self.redrawRect = CGRectUnion(self.redrawRect,
                                      CGRectMake(self.lastTouchLocation.x -
                                                  horizontalOffset,
                                                  self.lastTouchLocation.y -
                                                  verticalOffset,
                                                  self.drawImage.size.width,
                                                  self.drawImage.size.height));
    }
    self.redrawRect = CGRectUnion(_redrawRect, self.currentRect);
    [self setNeedsDisplayInRect:self.redrawRect];
}

```

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）。这样妥善处理 iOS 设备宝贵的处理器周期，可以在应用程序性能方面产生巨大的差别，尤其是遇到复杂的应用程序时。

---

**说明** 如果希望更深入地学习 Quartz 2D 主题，可以查阅由 Jack Nutting、Dave wooldridge 和 David Mark 编写的《iPad 开发基础教程》，其中介绍了大量 Quartz 2D 绘图知识。该书中的所有绘图代码和说明都同时适用于 iPhone 和 iPad。

---

## 16.4 小结

在本章中，我们事实上只学习了 iOS 绘图功能的一点皮毛。现在你应该逐渐适应了 Quartz 2D。通过参考苹果公司的文档，你还可以处理遇到的大多数绘图需求。

现在是时候更多地提高你的图形技能了。在第 17 章将会向你介绍 iOS 7 中全新的 Sprite Kit 框架，它能让你快速地执行位图渲染，以创建游戏或其他可以交互的内容。



在 iOS 7 中，苹果引入了一个用来进行高效率渲染的 2D 图形框架——Sprite Kit。听起来有点像 Core Graphics 和 Core Animation，那么有哪些新功能呢？与主要通过绘图模式绘制图形的 Core Graphics 以及主要用来管理 GUI 元素动画属性的 Core Animation 不同，Sprite Kit 专注于一个完全不同的领域：视频游戏！Sprite Kit 是基于 OpenGL 构建的，后者是通用计算机平台的技术实现，能够让现代图形硬件将位图图形以不可思议的速度写入视频缓存内。通过 Sprite Kit，你可以不必深入研究 OpenGL 底层编码技术，就能利用其良好的性能特征。

这是苹果公司在 iOS 系统上首次尝试进军游戏开发的图形部分。它同时在 iOS 7 和 OS X 10.9 (Mavericks) 上发布，并在两个平台上提供相同的 API 接口，因此应用都只需要编写一次就可以轻松移植到另一个平台上。尽管苹果此前从未提供过像 Sprite Kit 这样的框架，而它却与多个开源库（比如 Cocos2D）非常相似。如果你过去曾使用 Cocos2D 或其他类似的工具，那么你就会觉得学起来很轻松。

Sprite Kit 并没有像 Core Graphics 那样去实现一个灵活的通用绘图系统。它没有绘制路径、渐变或颜色填充的方法。而你使用到的是一个场景图表（scene graph），它与 UIKit 视图的层级关系有些相似。它可以对每一个图表节点的位置、缩放和旋转进行变换，每个节点还可以绘制自身。大部分绘图都是通过 SKSprite 类或子类的实例对象而执行的，它用来表示即将放置在屏幕上的某张图像的图形实例。

本章将通过 Sprite Kit 构建一个简单的射击游戏，名字叫做 TextShooter。不必制作图像，我们将使用文本来构建我们游戏中的对象，通过 SKSprite 的某个子类就可以专门实现这个功能。采用这种方式，你就不需要从项目库或类似资源中提取图片了。我们创建的应用外观很简洁，很容易修改，也适合娱乐。

## 17.1 基础入门

我们开始吧。在 Xcode 中的 iOS Application 边栏里选择 SpriteKit Game 应用模板，创建一个新的应用。给你的新项目命名为 TextShooter，其余的设置不做更改，并与你的其他项目存储在同一目录下。

现在浏览一下 Xcode 创建的项目。它包含了最基本的 BIDAppDelegate 类和一个用来进行 SKView 初始化配置的简单 BIDViewController 类。SKView 对象是从应用分镜中加载，用来显示

所有 Sprite Kit 内容。完成配置后 SKView 就可以向我们展示一些运行时的性能特征了，viewDidLoad 方法创建了一个新的 BIDMyScene 实例对象并告诉 SKView 显示这个场景。

某种程度上，SKView 与我们在本书中使用 UIViewController 类有些相似。SKView 类扮演的角色有点像 UINavigationController，它有点类似总控制器，可以管理对其他控制器显示。不过之后就有些区别了，与 UINavigationController 不同，SKView 管理的顶层对象并不是 UIViewController 的子类，而是 SKScene 的子类，它们分别各自管理对象的关系（如何显示以及物理引擎效果等）。使用 Sprite Kit 开发时，你可能会创建一个新的 SKScene 子类来管理应用中各个部分。场景可以用来表现有许多对象在屏幕上移动的快节奏游戏，也可以用来表现简单的开始菜单。在本章中我们将看到 SKScene 的多种使用方式。

你还需要浏览一下通过模板创建的 BIDMyScene 类。它包含了两个方法：一个用来创建屏幕上标签的初始化函数和一个触摸事件处理器。后者会在用户每次触摸屏幕时创建一个由位图表示的精灵（sprite）。运行这个应用就可以看到它的工作方式（运行界面如图 17-1 所示）。



图 17-1 正在运行的默认 Sprite Kit 应用。文本显示在屏幕中央，每次点击屏幕都会在当前位置放置一个旋转的喷气式飞机图片

### 17.1.1 自定义初始场景

已经有一些东西了，不过我们想要的应用和它完全不一样。我们要改动 Xcode 自动设置的两个方法。首先删除 initWithSize: 方法的全部内容（之后会编写一个全新的），然后移除 touchesBegan:withEvent: 方法的大部分内容，只留下 for 循环语句和里面的第一行代码，就像这样：

```
(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    /* 触摸时调用 */

    for (UITouch *touch in touches) {
        CGPoint location = [touch locationInNode:self];
    }
}
```

接下来我们要修改这个模板在命名风格上的小缺陷：BIDMyScene 类自身的名字。Sprite Kit 游戏中每个场景都代表一种特定的用户游戏体验，它的名称应当与之对应。场景名字中像 my 这样的词语不能表达任何意义，所以要为其重新命名。实际上，Xcode 有一个便利的功能，可以帮助我们做到这一点。请将鼠标停在这个类的.h 或.m 文件中 BIDMyScene 文本的上面，右击（如果使用的是苹果的触控板可以按住 control 键点击）以调出弹出菜单，并选择 Refactor（重构）>Rename...（重命名...）选项。之后在窗口顶部出现的面板中，输入 BIDLevelScene 作为新的类名，并确保 Rename related files（重命名相关文件）复选框已被选中，然后点击 Preview（预览）按钮。一个新的面板将会弹出，它显示了 Xcode 按照你的意愿将做出的所有更改，如图 17-2 所示。

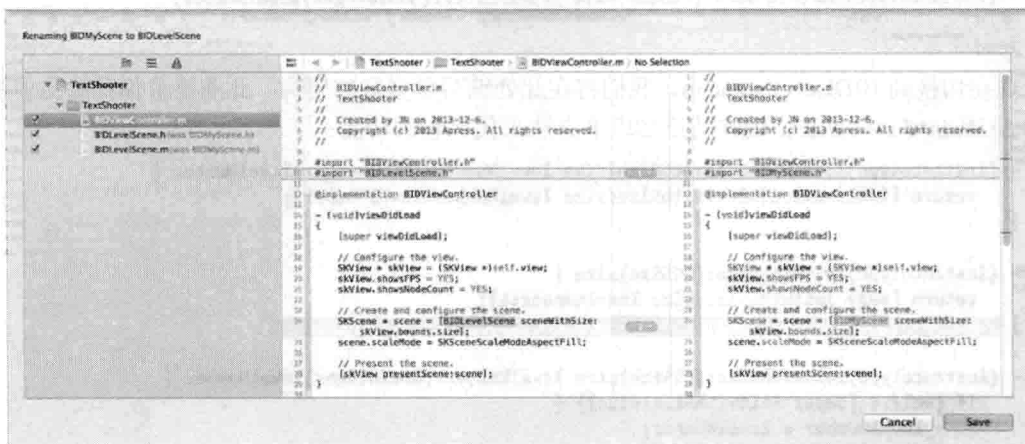


图 17-2 我们所做更改的两侧比较界面

点击 Save 按钮，它将触发窗口从顶部弹出另一个面板。它询问是否要开启 Xcode 中一个叫做 snapshots（快照）的功能，你可以选择 Disable。之后你将看到 Xcode 更改了源文件中的名称，和源代码文件本身的名字，还更改了 BIDViewController.m 中所有相关的名称。非常好！

### 17.1.2 隐藏状态栏

进入 BIDViewController.m 后，对它进行更改：禁用 iOS 的状态栏。只需要在 @implementation 部分的任意位置添加这个方法：

```
- (BOOL)prefersStatusBarHidden {
    return YES;
}
```

这个更改将让 iOS 状态栏在游戏运行时消失，通常在像这样的动作游戏中应该这么做。

## 17.2 场景设置

现在切换到 BIDLevelScene.h。这个类可以重写父类的 initWithSize: 方法，将其作为初始化

函数。不过我们要编写两个新方法来创建实例，这样就可以包含关卡序数。同时要添加关卡序数和玩家拥有生命条数的属性变量，以及一个能让我们知道此关是否通关的 flag（布尔值标记）。请添加以下粗体显示的代码：

```
@interface BIDLevelScene : SKScene

@property (assign, nonatomic) NSUInteger levelNumber;
@property (assign, nonatomic) NSUInteger playerLives;
@property (assign, nonatomic) BOOL finished;

+ (instancetype)sceneWithSize:(CGSize)size levelNumber:(NSUInteger)levelNumber;
- (instancetype)initWithSize:(CGSize)size levelNumber:(NSUInteger)levelNumber;

@end
```

现在切换到 BIDLevelScene.m，我们将在此处编写更多基础架构。此前要求你移除由模板创建的 initWithSize: 方法，现在请添加以下方法来代替它的功能：

```
+ (instancetype)sceneWithSize:(CGSize)size levelNumber:(NSUInteger)levelNumber {
    return [[self alloc] initWithSize:size levelNumber:levelNumber];
}

- (instancetype)initWithSize:(CGSize)size {
    return [self initWithSize:size levelNumber:1];
}

- (instancetype)initWithSize:(CGSize)size levelNumber:(NSUInteger)levelNumber {
    if (self = [super initWithSize:size]) {
        _levelNumber = levelNumber;
        _playerLives = 5;

        self.backgroundColor = [SKColor whiteColor];

        SKLabelNode *lives = [SKLabelNode labelNodeWithFontNamed:@"Courier"];
        lives.fontSize = 16;
        lives.fontColor = [SKColor blackColor];
        lives.name = @"LivesLabel";
        lives.text = [NSString stringWithFormat:@"Lives: %lu",
            (unsigned long)_playerLives];
        lives.verticalAlignmentMode = SKLabelVerticalAlignmentModeTop;
        lives.horizontalAlignmentMode = SKLabelHorizontalAlignmentModeRight;
        lives.position = CGPointMake(self.frame.size.width,
            self.frame.size.height);

        [self addChild:lives];
        SKLabelNode *level = [SKLabelNode labelNodeWithFontNamed:@"Courier"];
        level.fontSize = 16;
        level.fontColor = [SKColor blackColor];
        level.name = @"LevelLabel";
        level.text = [NSString stringWithFormat:@"Level: %lu",
            (unsigned long)_levelNumber];
        level.verticalAlignmentMode = SKLabelVerticalAlignmentModeTop;
        level.horizontalAlignmentMode = SKLabelHorizontalAlignmentModeLeft;
        level.position = CGPointMake(0, self.frame.size.height);
        [self addChild:level];
    }
}
```

```

    }
    return self;
}

```

第一个方法 `sceneWithSize:levelNumber:` 是一个工厂方法，它可以快速创建一个关卡并立即设置它的序数。在第二个方法 `initWithSize:` 中，我们重写了类的默认初始化函数，将控制权交给第三个函数（并传递了一个默认的关卡序数值）。第三个方法由下往上调用了父类实现的特定初始化函数。这可能看起来有些绕，不过在为某个类添加新的初始化函数时，通常仍然会使用类中特定的初始化函数。

我们在添加的第三个方法 `initWithSize:levelNumber:` 中设置了关卡场景的基本配置。首先设置了两个实例变量的值为传入的参数，然后设置了场景的背景颜色。请注意这里使用了一个名称为 `SKColor` 的类，而不是 `UIColor`。事实上，`SKColor` 并不是一个真正的类，它是一种替身，可以代替 iOS 应用中的 `UIColor` 以及 OS X 应用中的 `NSColor`。这就可以轻松地在 iOS 和 OS X 之间进行移植。

之后我们创建了两个 `SKLabelNode` 类的实例。这是一个有些类似 `UILabel` 的便利类，便于选择字体、设置文本并指定其对齐方式。我们创建一个标签来显示屏幕右上角的生命条数，另一个用来显示屏幕左上角的关卡序数。

如果你认为传入的点是那些标签的位置，那么你会惊讶地发现传入的是场景的高度。在 `UIKit` 中，将位置设为 `UIView` 的高度会将它放在视图的底部，不过在 `Sprite Kit` 中  $y$  轴是相反的，因此场景高度最大值的位置是在屏幕的顶端。

你还会注意到每个标签都有名字。它的工作方式与 `UIKit` 中的 `tag` 标记或标识符类似，之后可以通过访问它们的名字来获取标签。

现在运行游戏，可以看到已经搭建了一个非常基础的构造，如图 17-3 所示。



图 17-3 游戏目前还不算有趣，不过至少帧数很给力

## 17.3 玩家动作

现在要添加一些交互性。我们来创建一个代表玩家的新类。它会知道如何使用内部元素来绘制自身，以及如何通过一个平滑的动画移动到新的位置。接下来要向场景中插入新类的实例并编写一些代码，以便让玩家可以通过触摸屏幕来移动对象。

场景中的每个对象都必须是 `SKNode` 的子类。因此你需要使用 Xcode 的 File 菜单创建一个名为 `BIDPlayerNode` 的新 Objective-C 类，并且是 `SKNode` 的子类。在新建的几乎全空的 `BIDPlayerNode.m` 文件中添加以下方法：

```
- (instancetype)init {
    if (self = [super init]) {
        self.name = [NSString stringWithFormat:@"Player %p", self];
        [self initNodeGraph];
    }
    return self;
}

- (void)initNodeGraph {
    SKLabelNode *label = [SKLabelNode labelNodeWithFontNamed:@"Courier"];
    label.fontColor = [SKColor darkGrayColor];
    label.fontSize = 40;
    label.text = @"v";
    label.zRotation = M_PI;
    label.name = @"label";

    [self addChild:label];
}
```

`BIDPlayerNode` 自身不会显示任何东西。而是通过 `init` 方法来设置一个子节点来进行实际绘制。这个子节点是 `SKLabelNode` 的另一个实例，与之前创建用来关卡序数和剩余生命条数所用的相似。我们没有设置标签的位置，这表示它的位置是坐标(0, 0)。与视图类似，坐标系统中的每个节点都继承父节点的位置。节点的位置为零意味着会在屏幕上 `BIDPlayerNode` 实例的位置出现。而任何非零值都会在这个位置上产生位移。

我们还要设定标签的旋转值，这样里面的小写字母 `v` 就可以上下颠倒显示出来。旋转属性 `zRotation` 的名字看起来可能有些奇怪，它只是代表 Sprite Kit 中坐标空间的  $z$  轴。你只能在屏幕上看到  $x$  轴和  $y$  轴，而  $z$  轴可以用来为显示内容的深度排序，以及旋转节点。赋给 `zRotation` 的值需要是弧度而不是角度，因此我们赋给值 `M_PI`，它与数学概念的值圆周率  $\pi$  相等。 $\pi$  的弧度等于 180 角度，这刚好就是你要的效果。

### 17.3.1 向场景中插入玩家

现在切换到 `BIDLevelScene.m`。我们要在这里向场景插入 `SKPlayerNode` 的实例。首先导入新类的头文件并在新的扩展中添加一个属性变量：

```
#import "BIDLevelScene.h"
#import "BIDPlayerNode.h"
```

```

@interface BIDLevelScene ()

@property (strong, nonatomic) BIDPlayerNode *playerNode;

@end

```

接着在 `initWithSize:levelNumber:` 方法末尾处附近添加以下粗体显示的代码，要把它放在 `return self` 及其上面的花括号之前：

```

    _playerNode = [BIDPlayerNode node];
    _playerNode.position = CGPointMake(CGRectGetMidX(self.frame),
                                       CGRectGetHeight(self.frame) * 0.1);

    [self addChild:_playerNode];
}
return self;
}

```

如果现在构建并运行应用，应该会看到玩家出现在屏幕底部中间的位置附近，如图 17-4 所示。



图 17-4 一个倒置的 v 要开始战斗了

### 17.3.2 触摸处理

接下来要在之前清空的 `touchesBegan:withEvent:` 方法中添加一些逻辑。请插入以下粗体显示的代码：

```

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    /* 触摸时调用 */
}

```

```

for (UITouch *touch in touches) {
    CGPoint location = [touch locationInNode:self];
    if (location.y < CGRectGetHeight(self.frame) * 0.2 ) {
        CGPoint target = CGPointMake(location.x,
                                      self.playerNode.position.y);
        CGFloat duration = [self.playerNode moveToward:target];
    }
}
}

```

上面的代码片段采用屏幕底部五分之一区域中的任意触摸位置作为新的位置目标, 你的玩家节点将向此处移动。在当前环境中, 编译器会警告出错, 因为还没有定义玩家节点的 `moveToward:` 方法。所以要在 `BIDPlayerNode.h` 中声明方法, 就像这样:

```

#import <SpriteKit/SpriteKit.h>

@interface BIDPlayerNode : SKNode

// 返回将来移动所用的时间
- (CGFloat)moveToward:(CGPoint)location;

@end

```

### 17.3.3 玩家移动

接下来切换到 `BIDPlayerNode.m` 并添加以下实现代码:

```

- (CGFloat)moveToward:(CGPoint)location {
    [self removeActionForKey:@"movement"];

    CGFloat distance = BIDPointDistance(self.position, location);
    CGFloat pixels = [UIScreen mainScreen].bounds.size.width;
    CGFloat duration = 2.0 * distance / pixels;

    [self runAction:[SKAction moveTo:location duration:duration]
               withKey:@"movement"];

    return duration;
}

```

我们现在先跳过第一行, 稍后再回来探讨它。这个方法比较了新位置与当前位置并计算出了距离和要移动的像素。接下来它根据通过常量设定的总体位移速度算出了需要多少时间来移动。最后它创建了一个 `SKAction` 来执行动作。`SKAction` 是 `Sprite Kit` 中的一个元素, 它知道如何使节点随着时间变化, 可以让你简单地使用节点的位移、缩放、旋转和透明度渐变动画等等。我们在这里所做的是告诉玩家节点在特定的时间内运行一个简单的位移动作, 然后为动作附上关键字 `"movement"`。如你所见, 这个关键字与方法中第一行移除的动作关键字一样。我们在方法一开始移除了任何已有的同关键字动作, 这样用户就可以快速连续轻点多个位置而不会因多个不同方向的动作而产生冲突。



### 17.3.4 几何运算

注意，现在触发了另一个编译器错误，原因是 Xcode 无法找到名称为 `BIDPointDistance()` 的方法。这是我们应用将使用的几个几何函数中的一个，它们通过点、向量和浮点值进行运算。我们现在开始编写。使用 Xcode 创建一个新文件，这次请选择 C and C++ 部分的 Header File。将它命名为 `BIDGeometry.h` 并添加以下内容。

```
#ifndef TextShooter_BIDGeometry_h
#define TextShooter_BIDGeometry_h

// 接受 CGVector 和 CGFloat 为参数
// 返回一个新的 CGVector，其中每个 v 元素都已经乘以 m
static inline CGVector BIDVectorMultiply(CGVector v, CGFloat m) {
    return CGVectorMake(v.dx * m, v.dy * m);
}

// 接受两个 CGPoint 为参数
// 返回一个表示 p1 到 p2 距离的 CGVector
static inline CGVector BIDVectorBetweenPoints(CGPoint p1, CGPoint p2) {
    return CGVectorMake(p2.x - p1.x, p2.y - p1.y);
}

// 接受 1 个 CGVector 为参数
// 通过勾股定理计算出向量的长度并返回 CGFloat 值
// Pythagoras' theorem.
static inline CGFloat BIDVectorLength(CGVector v) {
    return sqrtf(powf(v.dx, 2) + powf(v.dy, 2));
}

// 根据两个 CGPoints 坐标，通过勾股定理计算出两者间的距离
// 并返回 CGFloat 类型的长度值
static inline CGFloat BIDPointDistance(CGPoint p1, CGPoint p2) {
    return sqrtf(powf(p2.x - p1.x, 2) + powf(p2.y - p1.y, 2));
}

#endif
```

这都是一些通用操作的实现代码，在许多游戏中会用到：向量相乘、创建两点之间的向量以及计算距离。为了能使用它们，需要在 `BIDPlayerNode.m` 的顶端附近添加以下导入代码：

```
#import "BIDGeometry.h"
```

现在构建并运行应用。当玩家飞船出现后，轻点屏幕底部区域的任意位置可以看到飞船向左或向右靠近你点击的位置。你可以在飞船到达终点前再次点击，它将会立即开始新的动画并移动到新的位置。这很不错，不过如果玩家飞船在移动时能稍微真实一些就更好了。

### 17.3.5 轻微摆动

我们为飞船加入另一个位移动画时，使其移动时出现一些轻微的摆动。在 `BIDPlayerNode` 的 `moveToward:` 方法中添加以下粗体显示的代码。

```

- (CGFloat)moveToward:(CGPoint)location {
    [self removeActionForKey:@"movement"];
    [self removeActionForKey:@"wobbling"];

    CGFloat distance = BIDPointDistance(self.position, location);
    CGFloat pixels = [UIScreen mainScreen].bounds.size.width;
    CGFloat duration = 2.0 * distance / pixels;

    [self runAction:[SKAction moveTo:location duration:duration]
        withKey:@"movement"];

    CGFloat wobbleTime = 0.3;
    CGFloat halfWobbleTime = wobbleTime * 0.5;
    SKAction *wobbling = [SKAction
        sequence:@[[SKAction scaleXTo:0.2
                    duration:halfWobbleTime],
                    [SKAction scaleXTo:1.0
                    duration:halfWobbleTime]
        ]];
    NSInteger wobbleCount = duration / wobbleTime;

    [self runAction:[SKAction repeatAction:wobbling count:wobbleCount]
        withKey:@"wobbling"];

    return duration;
}

```

这里所做的与之前创建的位移动作类似，不过很多地方不一样。对于基本的位移，我们只计算了位移持续时间，然后在单一步骤中创建并运行位移动作。这次有一点复杂。首先我们定义了单次摆动的时间（飞船在移动时可能会多次摆动，不过摆动的频率是一致的）。摆动先是在飞船的  $x$  轴发生，飞船的宽度缩小到正常尺寸的  $2/10$ ，然后放大到完整尺寸。所有单一动作被封装到另一种被称为序列的动作中，它将依次执行包含的所有动作。接下来计算出在飞船移动的时间内要发生多少次摆动，并将 wobbling 序列包含在一个重复动作中，告诉它要执行多少次连续摆动。与之前一样，我们在方法的起始处取消之前的所有摆动动作，这样摆动就不会冲突了。

现在运行应用，你将看到飞船在向前或向后移动时会发生摆动。看起来就像它在飞行一样。

## 17.4 创建你的敌人

目前一切正常，不过这个游戏需要一些让玩家射击的敌人。我们将使用 Xcode 创建一个名称为 BIDEnemyNode 的新 Objective-C 类，使用 SKNode 作为父类。我们暂时不会为敌人类提供任何真实的行为，不过会让它出现。我们采用之前针对玩家的同样技术，使用文本来构建敌人。可以确定，没有文本字符比字母 X 更让人害怕，所以我们的敌人就是由小写  $x$  组成的一个字母 X！请尝试克服恐惧并添加这些方法：

```

- (instancetype)init {
    if (self = [super init]) {

```

```

        self.name = [NSString stringWithFormat:@"Enemy %p", self];
        [self initNodeGraph];
    }
    return self;
}

- (void)initNodeGraph {
    SKLabelNode *topRow = [SKLabelNode
                           labelNodeWithFontNamed:@"Courier-Bold"];
    topRow.fontColor = [SKColor brownColor];
    topRow.fontSize = 20;
    topRow.text = @"x x";
    topRow.position = CGPointMake(0, 15);
    [self addChild:topRow];

    SKLabelNode *middleRow = [SKLabelNode
                              labelNodeWithFontNamed:@"Courier-Bold"];
    middleRow.fontColor = [SKColor brownColor];
    middleRow.fontSize = 20;
    middleRow.text = @"x";
    [self addChild:middleRow];

    SKLabelNode *bottomRow = [SKLabelNode
                              labelNodeWithFontNamed:@"Courier-Bold"];
    bottomRow.fontColor = [SKColor brownColor];
    bottomRow.fontSize = 20;
    bottomRow.text = @"x x";
    bottomRow.position = CGPointMake(0, -15);
    [self addChild:bottomRow];
}

```

这里没有什么新的内容,我们只是通过调整每个文本位置的  $y$  坐标值,使其变成了多行文本。

17

## 17.5 在场景中放入敌人

现在要更改 BIDLevelScene.m 让一些敌人出现在场景中。首先在顶端添加以下粗体显示的代码:

```

#import "BIDLevelScene.h"
#import "BIDPlayerNode.h"
#import "BIDEnemyNode.h"

#define ARC4RANDOM_MAX    0x100000000

@interface BIDLevelScene ()

@property (strong, nonatomic) BIDPlayerNode *playerNode;
@property (strong, nonatomic) SKNode *enemies;

@end

```

我们导入了新敌人类的头文件,还定义了 arc4random() 函数的最大返回值,之后将会用到它。使用随机数生成器可以使同一游戏关卡在每一次玩时都不一样,而 arc4random() 可以做到随机。

最后我们添加一个新的属性变量来存储所有添加到关卡中的敌人。你可能认为我们会使用一个 NSMutableArray, 但事实上使用一个基本的 SKNode 更加有效。SKNode 可以拥有任意数量的子节点。而且我们总是要向场景中添加所有的敌人, 使用 SKNode 包含它们也易于访问。

接下来要创建 spawnEnemies 方法, 如下所示:

```
- (void)spawnEnemies {
    NSUInteger count = log(self.levelNumber) + self.levelNumber;
    for (NSUInteger i = 0; i < count; i++) {
        BDEnemyNode *enemy = [BDEnemyNode node];
        CGSize size = self.frame.size;
        CGFloat x = (size.width * 0.8 * arc4random() / ARC4RANDOM_MAX) +
                    (size.width * 0.1);
        CGFloat y = (size.height * 0.5 * arc4random() / ARC4RANDOM_MAX) +
                    (size.height * 0.5);
        enemy.position = CGPointMake(x, y);
        [self.enemies addChild:enemy];
    }
}
```

最后在 initWithSize:levelNumber:方法的末尾处添加这些代码以创建一个空的敌人节点, 然后调用 spawnEnemies 方法:

```
[self addChild:_playerNode];
_enemies = [SKNode node];
[self addChild:_enemies];
[self spawnEnemies];
```

现在运行应用, 你将会看到一个可怕的敌人位于屏幕的随机位置 (参考图 17-5)。你难道不想射它吗?



图 17-5 我能肯定你想射掉由 x 组成的 X

## 17.6 开始射击

现在要实现的游戏开发的下一个逻辑步骤：让玩家攻击敌人。我们想让玩家能够轻点屏幕上方 80% 的区域来发出射向敌人的导弹。可以使用 Sprite Kit 中的物理引擎来移动玩家的导弹并判断导弹与敌人是否发生了碰撞。

不过首先要知道物理引擎是什么东西。简单来说，物理引擎是一个可以记录场景中多个物理对象和它们之间产生的作用力的软件组件。它还可以确保所有对象能以现实中的方式进行运作。它可以考虑进重力的影响、处理对象之间碰撞（因此对象不会同时占据相同位置），甚至还能模拟像摩擦力和反弹力这样的物理特性。

一定要明白，物理引擎和图像引擎通常都是分开的。苹果提供了便利的 API 接口，使我们可以联合使用这两种引擎，不过它们本质上是分开的。显示对象是很普遍的事，就像显示等级数和剩余声明的标签，而这些内容与物理引擎是没有关系的。可能有时会创建一个拥有物理特性的对象，但却不会显示任何东西。

### 17.6.1 定义物理类别

Sprite Kit 物理引擎需要为对象设置不同的物理类别（physics categories）。物理类别与 Objective-C 的类别毫无关系。物理类别是一种集合相关对象的方式，这样物理引擎就可以用不同的方式处理它们之间的碰撞。例如在这个示例中，我们将创建 3 个类别：敌人、玩家、玩家的导弹。明显要让物理引擎关注敌人和玩家导弹之间的碰撞，不过可能要忽略玩家导弹和玩家自身之间的碰撞。使用物理类别可以很轻松地设置。

我们开始创建所需的类别。使用 Xcode 创建一个名称为 BIDPhysicsCategories.h 的新 C 头文件并添加以下内容：

```
#ifndef TextShooter_BIDPhysicsCategories_h
#define TextShooter_BIDPhysicsCategories_h

typedef NS_OPTIONS(uint32_t, BIDPhysicsCategory) {
    PlayerCategory      = 1 << 1,
    EnemyCategory       = 1 << 2,
    PlayerMissileCategory = 1 << 3
};

#endif
```

在这里声明了 3 个类别常量。请注意类别是位掩码，因此它们都是 2 的乘方。可以通过移位轻松做到。设置位掩码是为了简化物理引擎的 API。使用位掩码，我们可以使用 OR 运算符将多个值并在一起。这样就可以使用一个单独的 API 调用来告诉物理引擎如何处理多个不同层之间的碰撞。很快就能看到实例了。

### 17.6.2 创建BIDBulletNode类

现在要做些基础工作，首先创建一些导弹，这样就可以开始射击了。

创建一个名称为 `BIDBulletNode` 的新类，再次使用 `SKNode` 作为父类。在头文件中声明两个共有方法：

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

@interface BIDBulletNode : SKNode

+ (instancetype)bulletFrom:(CGPoint)start toward:(CGPoint)destination;
- (void)applyRecurringForce;

@end
```

第一个方法是工厂方法，用来创建类的新实例。第二个方法需要在场景的每一帧调用，以告诉导弹进行移动。现在切换到 `BIDBulletNode.m` 并实现这个类。

要做的第一件事是导入特定几何函数和物理类别所在的头文件，然后是添加单个属性的类扩展，它包含了这个导弹发射的向量：

```
#import "BIDBulletNode.h"
#import "BIDPhysicsCategories.h"
#import "BIDGeometry.h"

@interface BIDBulletNode ()

@property (assign, nonatomic) CGVector thrust;

@end
```

接下来实现一个 `init` 方法。与这个应用中其他 `init` 方法相似，这里创建导弹的对象图形，由一个点组成。我们还要通过创建并配置一个 `SKPhysicsBody` 实例，并将其赋给 `self` 以设置这个类的物理特性。在这个过程中，我们要告诉新的物理形体：它属于什么类别，并且应该会与什么类别的对象进行碰撞检测。

```
@implementation BIDBulletNode

- (instancetype)init {
    if (self = [super init]) {
        SKLabelNode *dot = [SKLabelNode labelNodeWithFontNamed:@"Courier"];
        dot.fontColor = [SKColor blackColor];
        dot.fontSize = 40;
        dot.text = @".";
        [self addChild:dot];
        SKPhysicsBody *body = [SKPhysicsBody bodyWithCircleOfRadius:1];
        body.dynamic = YES;
        body.categoryBitMask = PlayerMissileCategory;
        body.contactTestBitMask = EnemyCategory;
        body.collisionBitMask = EnemyCategory;
        body.mass = 0.01;

        self.physicsBody = body;
        self.name = [NSString stringWithFormat:@"Bullet %p", self];
    }
    return self;
}
```

### 17.6.3 应用物理知识

接下来要实现工厂方法，以创建一个新的导弹并给予它一个发射的向量，物理引擎将用它来使导弹朝目标前进：

```
+ (instancetype)bulletFrom:(CGPoint)start toward:(CGPoint)destination {
    BIDBulletNode *bullet = [[self alloc] init];

    bullet.position = start;

    CGVector movement = BIDVectorBetweenPoints(start, destination);
    CGFloat magnitude = BIDVectorLength(movement);
    if (magnitude == 0.0f) return nil;

    CGVector scaledMovement = BIDVectorMultiply(movement, 1 / magnitude);

    CGFloat thrustMagnitude = 100.0;
    bullet.thrust = BIDVectorMultiply(scaledMovement, thrustMagnitude);

    return bullet;
}
```

基本运算非常简单。首先设定它由起点位置指向终点的位移向量 `movement`，然后再设定它的 `magnitude`（长度）。用 `movement` 向量除以 `magnitude` 产生一个标准化的单位向量（unit vector），这个向量所指的方向与原先的相同，不过只有一个单位的长度（这里单位的概念与屏幕上的一个点相同，在 Retina 设备上就是两个像素）。创建一个单位向量是非常有用的，因为无论用户点击屏幕的位置有多远，都可以用它乘以一个固定的长度（这个示例中是 100）来算出同等速度的发射向量。

向这个类添加的最后一段代码是这个方法，它推动物理形体发射。在场景的每一帧中都要调用它：

```
- (void)applyRecurringForce {
    [self.physicsBody applyForce:self.thrust];
}
```

### 17.6.4 在场景中添加导弹

现在切换到 `BIDLevelScene.m` 以向场景自身添加导弹。一开始在顶部附近导入新类的头文件。接下来添加其他属性变量，以在一个 `SKNode` 中包含所有导弹，就像之前对敌人所做的那样：

```
#import "BIDLevelScene.h"
#import "BIDPlayerNode.h"
#import "BIDEnemyNode.h"
#import "BIDBulletNode.h"

#define ARC4RANDOM_MAX    0x100000000

@interface BIDLevelScene ()
```

```

@property (strong, nonatomic) BIDPlayerNode *playerNode;
@property (strong, nonatomic) SKNode *enemies;
@property (strong, nonatomic) SKNode *playerBullets;

```

```

@end

```

找到之前添加敌人的 `initWithSize:levelNumber:` 方法的位置。我们还要在这里设置 `playerBullets` 节点。

```

_playerBullets = [SKNode node];
[self addChild:_playerBullets];

```

现在准备编写实际的导弹发射代码。在 `touchesBegan:withEvent:` 方法中添加 `else` 语句，这样所有在屏幕上方区域的点击都会发射一个导弹，而不是移动飞船：

```

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    for (UITouch *touch in touches) {
        CGPoint location = [touch locationInNode:self];
        if (location.y < CGRectGetHeight(self.frame) * 0.2) {
            CGPoint target = CGPointMake(location.x,
                                         self.playerNode.position.y);
            [self.playerNode moveToward:target];
        } else {
            BIDBulletNode *bullet = [BIDBulletNode
                                     bulletFrom:self.playerNode.position
                                     toward:location];
            if (bullet) {
                [self.playerBullets addChild:bullet];
            }
        }
    }
}

```

这样可以添加导弹，不过添加的导弹事实上不会移动，除非在每一帧中都给它们推力。场景已经包含了一个名称为 `update:` 的空方法，这个方法会在每一帧被调用，这里是用来放置每一帧都会调用的游戏逻辑的合适位置。我们不会直接在这个方法中更新所有的导弹，而是将代码放在由 `update:` 方法调用的独立方法中：

```

- (void)update:(CFTimeInterval)currentTime {
    [self updateBullets];
}

- (void)updateBullets {
    NSMutableArray *bulletsToRemove = [NSMutableArray array];
    for (BIDBulletNode *bullet in self.playerBullets.children) {
        // 清除所有移动到屏幕外部的导弹
        if (!CGRectContainsPoint(self.frame, bullet.position)) {
            // 标记将予以清除的导弹
            [bulletsToRemove addObject:bullet];
            continue;
        }
        // 将推力作用于剩下的导弹
        [bullet applyRecurringForce];
    }
}

```



```

    }
    [self.playerBullets removeChildrenInArray:bulletsToRemove];
}

```

在告诉每个导弹给予推力之前，我们还要检查是否每个导弹仍在屏幕上显示。所有移出屏幕的导弹要放入一个临时数组中，在最后它们要在 `playerBullets` 节点中清除。请注意这两步过程是必需的，因为 `for` 循环还在这个方法中执行 `playerBullets` 节点的子节点遍历。正在迭代时改变集合的内容永远不是一个正常的行为，它很容易引起崩溃。

现在构建并运行应用，你将看到除了玩家飞船的移动，还可以通过在屏幕上轻点来向上发射导弹（参考图 17-6）。干得漂亮！



图 17-6 射出你的金属风暴吧！

## 17.7 利用物理引擎攻击敌人

游戏中两个重要的元素仍然没有实现。敌人不会攻击我们，而我们也无法通过射击而消灭掉它们。现在先完成后者。我们将实现这个功能，这样射中一个敌人会让它当前在屏幕上的位置消失。这个功能将使用物理引擎来担当所有工作，并且会更改 `BIDPlayerNode`、`BIDEnemyNode` 和 `BIDLevelScene`。

首先为还没有物理形体的节点进行添加。在 `BIDEnemyNode.m` 的顶部附近添加这些 `#import` 语句：

```

#import "BIDPhysicsCategories.h"
#import "BIDGeometry.h"

```

接下来在 `init` 方法中添加以下代码：

```

- (instancetype)init {
    if (self = [super init]) {
        self.name = [NSString stringWithFormat:@"Enemy %p", self];
        [self initNodeGraph];
        [self initPhysicsBody];
    }
    return self;
}

```

现在添加实际用来设置物理形体的代码。这与我们对 BIDPlayerBullet 类所做的非常相似：

```

- (void)initPhysicsBody {
    SKPhysicsBody *body = [SKPhysicsBody bodyWithRectangleOfSize:
                           CGSizeMake(40, 40)];
    body.affectedByGravity = NO;
    body.categoryBitMask = EnemyCategory;
    body.contactTestBitMask = PlayerCategory|EnemyCategory;
    body.mass = 0.2;
    body.angularDamping = 0.0f;
    body.linearDamping = 0.0f;
    self.physicsBody = body;
}

```

然后选中 BIDPlayerNode.m，对它进行几乎一样的设置。首先在顶部附近添加#import 语句：

```
#import "BIDPhysicsCategories.h"
```

之后在 init 方法中添加粗体显示的代码：

```

- (instancetype)init {
    if (self = [super init]) {
        self.name = [NSString stringWithFormat:@"Player %p", self];
        [self initNodeGraph];
        [self initPhysicsBody];
    }
    return self;
}

```

最后添加全新的 initPhysicsBody 方法：

```

- (void)initPhysicsBody {
    SKPhysicsBody *body = [SKPhysicsBody bodyWithRectangleOfSize:
                           CGSizeMake(20, 20)];
    body.affectedByGravity = NO;
    body.categoryBitMask = PlayerCategory;

    body.contactTestBitMask = EnemyCategory;
    body.collisionBitMask = 0;

    self.physicsBody = body;
}

```

此时可以运行应用并发现导弹现在可以消灭太空中的敌人了。不过这里你还会发现一些问题。在开始游戏并将最后一个敌人消灭后，就无限卡住了！这时要在游戏中添加关卡管理功能。

## 17.8 实现关卡

我们需要改善 `BIDLevelScene`，这样就会知道什么时候进入下一关卡。它只要寻找现有敌人的数量就能算出来了。如果它发现屏幕上已经没有敌人了，当前关卡就完成了，游戏就会进入下一关卡。

### 17.8.1 注意敌人

首先添加这个 `updateEnemies` 方法。它的工作方式与之前添加的 `updateBullets` 方法相似：

```
- (void)updateEnemies {
    NSMutableArray *enemiesToRemove = [NSMutableArray array];
    for (SKNode *node in self.enemies.children) {
        // 清除移动到屏幕外的敌人
        if (!CGRectContainsPoint(self.frame, node.position)) {
            // 标记将予以清除的敌人
            [enemiesToRemove addObject:node];
            continue;
        }
    }
    if ([enemiesToRemove count] > 0) {
        [self.enemies removeChildrenInArray:enemiesToRemove];
    }
}
```

它会将所有屏幕外的敌人从关卡的 `enemies` 数组中移除。现在修改 `update:` 方法，让它调用 `updateEnemies` 方法，以及一个还没有实现的新方法：

```
- (void)update:(CFTimeInterval)currentTime {
    /* 渲染每帧时调用 */
    if (self.finished) return;

    [self updateBullets];
    [self updateEnemies];
    [self checkForNextLevel];
}
```

我们在方法的开头检查了 `finished` 属性变量。因为添加的方法会结束一个等级，所以需要确保在关卡结束后没有重复执行多余的操作！然后，在每一帧中检查是否有导弹或敌人在屏幕之外，将在每一帧调用 `checkForNextLevel` 来判断当前等级关卡是否已完成。添加这个方法：

```
- (void)checkForNextLevel {
    if ([self.enemies.children count] == 0) {
        [self goToNextLevel];
    }
}
```

### 17.8.2 进入下一关卡

我们还没有实现 `checkForNextLevel` 方法中依次调用的另一个方法。`goToNextLevel` 方法将这个关卡标为已完成，在屏幕上显示一些文本以让玩家知道，然后开始下一关卡：

```

- (void)goToNextLevel {
    self.finished = YES;

    SKLabelNode *label = [SKLabelNode labelNodeWithFontNamed:@"Courier"];
    label.text = @"Level Complete!";
    label.fontColor = [SKColor blueColor];
    label.fontSize = 32;
    label.position = CGPointMake(self.frame.size.width * 0.5,
                                self.frame.size.height * 0.5);
    [self addChild:label];

    BIDLevelScene *nextLevel = [[BIDLevelScene alloc]
                                initWithSize:self.frame.size
                                levelNumber:self.levelNumber + 1];
    nextLevel.playerLives = self.playerLives;
    [self.view presentScene:nextLevel
                       transition:[SKTransition flipHorizontalWithDuration:1.0]];
}

```

goToNextLevel 方法之后创建了一个新的 BIDLevelScene 实例并初始化了它需要的值。然后告诉视图通过转场来展示新的场景。SKTransition 类可以让我们使用各种转场。运行应用并完成一个关卡来观察效果（如图 17-7 所示）。

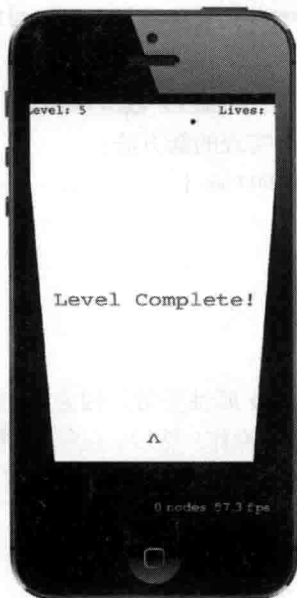


图 17-7 结束关卡后屏幕翻转的转场快照

这里使用的转场看起来像是绕着水平轴翻转卡片，不过你有更多的选择！浏览文档或 SKTransition 的头文件，试试其他方式。我们将在本章后面使用更多的类型。

## 17.9 自定义碰撞

现在，游戏基本上可以玩了。你可以通过将敌人向上击出屏幕来结束关卡。这就可以了，不过这一点儿也没有挑战性！我们之前说过，敌人攻击玩家的代码还没有写，现在是时候实现它了。我们要稍微复杂一点，敌人被导弹击中，或是和其他敌人碰撞时，都会向下坠落。我们还要让玩家被坠落的敌人碰撞时扣去一条生命。你可能还注意到了导弹在撞击敌人之后会曲线绕过敌人再继续向上，这种效果是不合理的。我们要在 `BIDLevelScene.m` 中实现碰撞处理程序来处理这些情况。

处理碰撞检测的方法是 `SKPhysicsWorld` 类的委托方法。场景默认拥有一个物理世界，不过事先需要稍微进行一些设置。一开始最好就让编译器知道要实现一个委托协议，因此在文件顶部附近的类扩展中添加这个声明语句：

```
@interface BIDLevelScene () <SKPhysicsContactDelegate>
```

我们仍然需要稍微设置物理世界（稍微减少重力）并告诉它委托是哪个。完成后在 `init` 方法结尾附近添加这些粗体代码，在这里还要添加所有其他设置：

```
self.physicsWorld.gravity = CGVectorMake(0, -1);  
self.physicsWorld.contactDelegate = self;
```

现在设置了物理世界的 `contactDelegate` 为 `BIDLevelScene`，可以实现相关的委托方法。方法的核心部分应该是这样的：

```
- (void)didBeginContact:(SKPhysicsContact *)contact {  
    if (contact.bodyA.categoryBitMask == contact.bodyB.categoryBitMask) {  
        // 两种物理形体都属于同一物理类别  
        SKNode *nodeA = contact.bodyA.node;  
        SKNode *nodeB = contact.bodyB.node;  
  
        // 这些节点能干什么用  
    } else {  
        SKNode *attacker = nil;  
        SKNode *attackee = nil;  
  
        if (contact.bodyA.categoryBitMask > contact.bodyB.categoryBitMask) {  
            // Body A 正在攻击 Body B  
            attacker = contact.bodyA.node;  
            attackee = contact.bodyB.node;  
        } else {  
            // Body B 正在攻击 Body A  
            attacker = contact.bodyB.node;  
            attackee = contact.bodyA.node;  
        }  
        if ([attackee isKindOfClass:[BIDPlayerNode class]]) {  
            self.playerLives--;  
        }  
        // 应该怎么处理攻击者和受攻击者的逻辑机制  
    }  
}
```

继续添加方法, 不过如果你刚刚看到这里, 会发现还没有做完。事实上, 这个方法的具体结果是每当坠落的敌人碰撞玩家的飞船时扣除玩家的生命。不过敌人还无法坠落!

要想实现这一点, 需要根据两个碰撞对象计算出它们是否属于同一类别(这样它们就是“伙伴”)。如果它们是不同的类别, 必须判断谁是攻击者, 谁是受攻击者。

如果浏览了 `BIDPhysicsCategories.h` 中声明的类别顺序, 你会看到它们指定的顺序的攻击性是增加的: `Player` 节点可以被 `Enemy` 节点攻击, 而 `Enemy` 则被 `PlayerMissile` 节点攻击。这意味着使用一个简单的大小比较来算出谁是攻击者。

考虑到简洁和模块化, 我们不会让场景来决定每个对象在被敌人或其他对象攻击时的行为。最好在受影响的节点类自身中构建这些细节。不过在已有的方法中, 我们可以确定的是它们都有一个 `SKNode` 实例。与其编写一个繁杂的 `if-else` 语句结构来访问每个属于 `SKNode` 子类的节点, 不如使用常规的多态性来让每个节点类来以自己的方式进行处理。为了让它能够执行, 必须在 `SKNode` 中添加方法, 默认的实现不执行任何操作, 让子类可以适当地重写。这个特性就是类别! 这次不是 `Sprite Kit` 物理类别, 而是真正的 Objective-C 的 `@category` 定义。

### 17.9.1 为 SKNode 添加类别

为了向 `SKNode` 添加类别, 请右击 Xcode 项目导航栏中的 `TextShooter` 文件夹并在右键菜单中选择 `New File...` 选项。在向导的 `iOS/Cocoa Touch` 部分中选择 `Objective-C category`, 然后点击 `Next`。将其命名为 `Extra` 并在 `Category on` 文本框中输入 `SKNode`。现在再次点击 `Next` 以创建文件。选择 `SKNode+Extra.h` 并添加以下粗体显示的方法声明:

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

@interface SKNode (Extra)

- (void)receiveAttacker:(SKNode *)attacker contact:(SKPhysicsContact *)contact;
- (void)friendlyBumpFrom:(SKNode *)node;
```

@end

切换到相应的 .m 文件并输入以下空方法定义:

```
#import "SKNode+Extra.h"

@implementation SKNode (Extra)

- (void)receiveAttacker:(SKNode *)attacker contact:(SKPhysicsContact *)contact {
    // 默认的实现不执行任何操作
}

- (void)friendlyBumpFrom:(SKNode *)node {
    // 默认的实现不执行任何操作
}

@end
```

现在回到 `BIDLevelScene.m` 以完成这部分的碰撞处理。首先在顶部添加新的头文件:

```
#import "BIDLevelScene.h"
#import "BIDPlayerNode.h"
#import "BIDEnemyNode.h"
#import "BIDBulletNode.h"
#import "SKNode+Extra.h"
```

接下来回到 `didBeginContact:` 方法，你将添加这些实际上执行相同工作的代码：

```
- (void)didBeginContact:(SKPhysicsContact *)contact {
    if (contact.bodyA.categoryBitMask == contact.bodyB.categoryBitMask) {
        // 两种物理形体都属于同一物理类别
        SKNode *nodeA = contact.bodyA.node;
        SKNode *nodeB = contact.bodyB.node;

        // 这些节点能干什么用
        [nodeA friendlyBumpFrom:nodeB];
        [nodeB friendlyBumpFrom:nodeA];
    } else {
        SKNode *attacker = nil;
        SKNode *attackee = nil;

        if (contact.bodyA.categoryBitMask > contact.bodyB.categoryBitMask) {
            // Body A 正在攻击 Body B
            attacker = contact.bodyA.node;
            attackee = contact.bodyB.node;
        } else {
            // Body B 正在攻击 Body A
            attacker = contact.bodyB.node;
            attackee = contact.bodyA.node;
        }
        if ([attackee isKindOfClass:[BIDPlayerNode class]]) {
            self.playerLives--;
        }
        // 应该怎么处理攻击者和受攻击者的逻辑机制
        if (attacker) {
            [attackee receiveAttacker:attacker contact:contact];
            [self.playerBullets removeChildrenInArray:@[attacker]];
            [self.enemies removeChildrenInArray:@[attacker]];
        }
    }
}
```

17

我们在这里添加的全部都是新方法的调用。如果碰撞是“伙伴”，比如两个敌人撞在一起，我们就要告诉它们这是伙伴的碰撞。否则，在计算出谁攻击谁受攻击之后，我们要告诉受攻击者：它被其他对象攻击了。最后我们在 `playerBullets` 或 `enemies` 节点中移除攻击者。我们告诉那些节点移除攻击者，虽然它只可能是其中某一个的子节点，不过这样也行。告诉节点移除一个不存在的子节点不会导致错误——只是不产生任何效果。

### 17.9.2 向敌人添加自定义碰撞行为

现在所有内容已经就绪，我们可以通过重写为 `SKNode` 添加的类别来实现节点的一些特定行为。

选择 BIDEnemyNode.m 并添加以下两个方法：

```
- (void)friendlyBumpFrom:(SKNode *)node {
    self.physicsBody.affectedByGravity = YES;
}

- (void)receiveAttacker:(SKNode *)attacker contact:(SKPhysicsContact *)contact {
    self.physicsBody.affectedByGravity = YES;
    CGVector force = BIDVectorMultiply(attacker.physicsBody.velocity,
                                       contact.collisionImpulse);
    CGPoint myContact = [self.scene convertPoint:contact.contactPoint
                                       toNode:self];
    [self.physicsBody applyForce:force
                      atPoint:myContact];
}
```

第一个方法 friendlyBumpFrom: 首先使敌人受到重力影响。因此，如果某个敌人在移动时碰到了其他敌人，第二个敌人将会立即受到重力影响而向下坠落。

receiveAttacker:contact: 方法会在敌人被导弹击中时会调用，首先会开启敌人的重力影响。此外它会使用传入的接触数据计算出接触发生的位置并向这个点传送力，给发射的导弹朝这个方向额外的推力。

### 17.9.3 准确显示玩家生命

运行游戏，你将会看到可以射击敌人并将它们击落。你还会看到其他敌人被坠落的敌人撞到时也会向下坠落。

---

**注意** 在每个关卡开始时，物理世界会通过物理模拟来确保没有任何物理形体是重叠的。在高等级关卡中，会发生这样一种有趣的副作用：因为成倍的敌人在随机位置出现，重叠在一起的几率会增加。若这种情况发生，敌人会立即位移，这样就不会重叠了，而我们的碰撞处理代码将会被触发，随后将开启重力并向下坠落！在我们开始构建这个游戏时没有想到这种情况，不过这个意外惊喜会使高等级关卡更加困难，我们暂不考虑这种情况！

---

如果敌人在坠落时击中了你，玩家的生命条数会减少，不过……请等下，始终都显示的是 5！在创建关卡时会设置 Lives 显示，不过之后它从未更新过。所幸，通过实现 setPlayerLives: 的 setter 方法来取代自动合成的 setter 可以轻松解决这个问题，就像这样：

```
- (void)setPlayerLives:(NSUInteger)playerLives {
    _playerLives = playerLives;
    SKLabelNode *lives = (id)[self childNodeWithName:@"LivesLabel"];
    lives.text = [NSString stringWithFormat:@"Lives: %lu",
                                                (unsigned long)_playerLives];
}
```

前面的代码片段使用之前对标签关联的名称（在 init 方法中）来再次找到标签并设置新的文本值。再次运行游戏，你将会看到如果敌人多次连续击中了玩家，生命条数会减少到零。而游



戏没有结束。在下次被击中后，你会看到一个非常巨大的生命条数量，如图 17-8 所示。

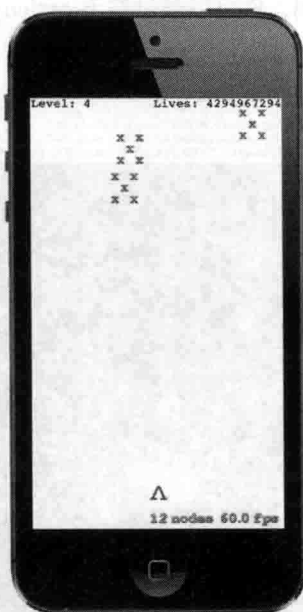


图 17-8 太多条命了

17

这是怎么回事？因为我们使用了一个无符号整型值来存储生命条数。当你使用无符号整型并且值在零以下，会溢出零的界限并显示出无符号整型能够显示的最大值！

问题出现的原因是因为我们没有写任何代码来检测游戏是否结束，也就是当玩家生命条数变成零时。我们将很快完成它，不过首先要让屏幕上的碰撞更刺激一些。

## 17.10 粒子系统

Sprite Kit 的优秀之处在于它集成了粒子系统。粒子系统在游戏中可以用来创建视觉特效，模拟烟雾、火焰和爆炸，等等。目前当导弹撞击到敌人，或敌人撞到玩家时，效果都还只是攻击对象消失而已。我们来创建两个粒子系统来改善这种情况。

首先按下 `command+N` 以创建一个新文件。选择左侧的 iOS 下的 Resource 部分，然后在右侧选择 SpriteKit Particle File。点击 Next，在之后的界面中选择 Spark 粒子模板。再次点击 Next 并将文件命名为 `MissileExplosion.sks`。

### 17.10.1 第一个粒子

你将会看到 Xcode 创建了粒子文件，并在项目中添加了名为 `spark.png` 的新资源。同时整个 Xcode 编辑区域切换到了新的粒子文件，并显示了一个巨大的动画爆炸效果。

我们不希望导弹击中敌人的特效如此夸张庞大，因此要重新配置它。定义粒子动画的所有属性都可以在 SKNode 的检查器中看到，你可以通过按下 option+command+7 来调出它。图 17-9 显示了巨大的爆炸和检查器。

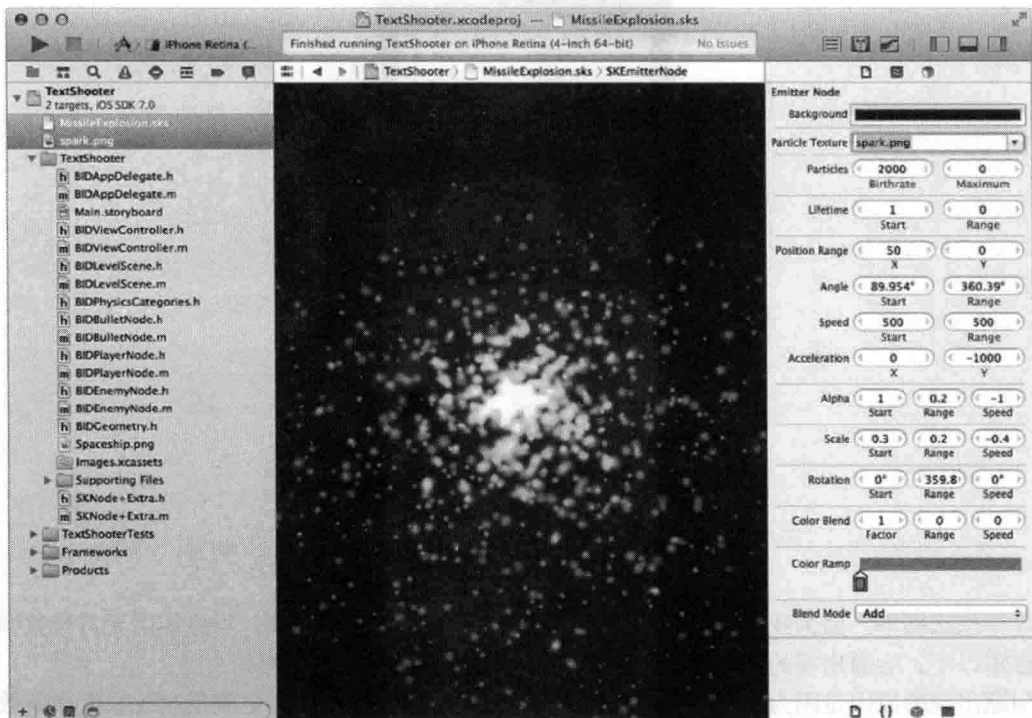


图 17-9 城市毁灭了！右侧显示的参数定义了默认的粒子效果

现在为导弹击中配置一个小型的爆炸。检查器中所有的参数都要重新设置。首先让颜色与我们的游戏看起来匹配，点击 Background 颜色并将其设置为白色。接下来点击底部的 Color Ramp（颜色过渡）并将小方块的颜色设置为黑色。并且将 Blend Mode 设置为 Alpha，现在你就可以看到像墨水一样的爆炸喷射效果。

其余的参数都是数字。将它们都设置为图 17-10 中所示。每设置一步，你就会看到粒子特效逐渐接近最终效果（参见图 17-10）。

现在创建另一个粒子特效，再次使用 Spark 模板。将其命名为 EnemyExplosion.sks 并设置它的参数如图 17-11 所示。

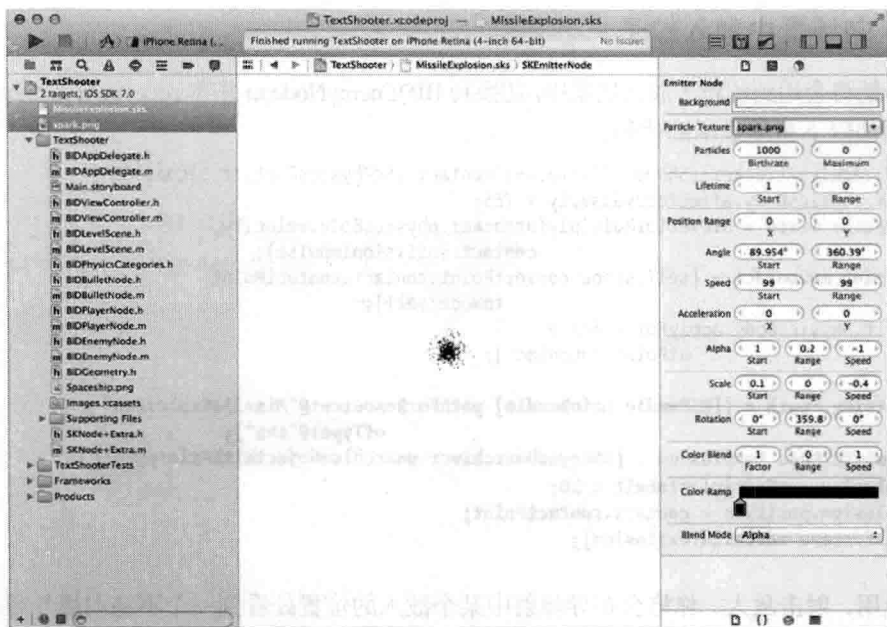


图 17-10 这就是我们想要的最终导弹爆炸效果

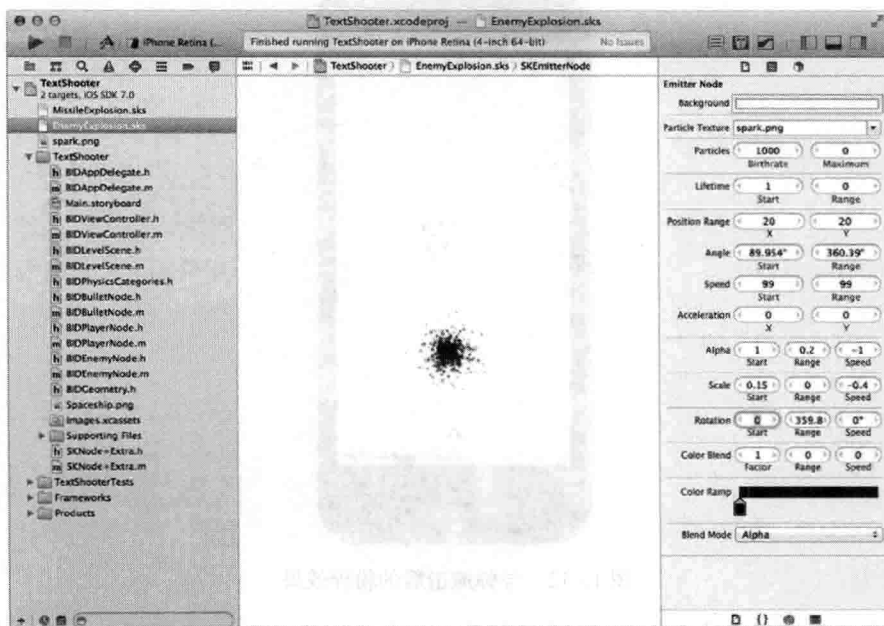


图 17-11 这是我们想要创建的敌人爆炸效果。你在纸质书上看到的是黑色或白色，而我们在 Color Ramp 实际选择的是深红

### 17.10.2 向场景中加入粒子

现在开始将会用到的粒子放入场景中。切换到 `BIDEnemyNode.m` 并在 `receiveAttacker:contact:` 方法底部添加以下粗体显示的代码：

```
- (void)receiveAttacker:(SKNode *)attacker contact:(SKPhysicsContact *)contact {
    self.physicsBody.affectedByGravity = YES;
    CGVector force = BIDVectorMultiply(attacker.physicsBody.velocity,
                                       contact.collisionImpulse);
    CGPoint myContact = [self.scene convertPoint:contact.contactPoint
                                              toNode:self];
    [self.physicsBody applyForce:force
                       atPoint:myContact];

    NSString *path = [[NSBundle mainBundle] pathForResource:@"MissileExplosion"
                                                           ofType:@"sks"];
    SKEmitterNode *explosion = [NSKeyedUnarchiver unarchiveObjectWithFile:path];
    explosion.numParticlesToEmit = 20;
    explosion.position = contact.contactPoint;
    [self.scene addChild:explosion];
}
```

运行应用，射击敌人，你将会在导弹射中某个敌人的位置处看到一个不错的爆炸效果，如图 17-12 所示。

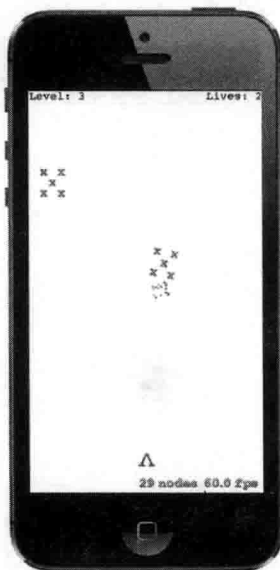


图 17-12 导弹撞击后的粉碎效果

很好！现在我们要对敌人撞击到玩家飞船执行类似的操作。选择 `BIDPlayerNode.m` 并添加这个方法：

```

- (void)receiveAttacker:(SKNode *)attacker contact:(SKPhysicsContact *)contact {
    NSString *path = [[NSBundle mainBundle] pathForResource:@"EnemyExplosion"
                                                         ofType:@"sks"];
    SKEmitterNode *explosion = [NSKeyedUnarchiver unarchiveObjectWithFile:path];
    explosion.numParticlesToEmit = 50;
    explosion.position = contact.contactPoint;
    [self.scene addChild:explosion];
}

```

再次运行，你将在每次敌人撞击到玩家时看到一个不错的红色特效，如图 17-13 所示。



图 17-13 好痛啊！

这些改动都很简单，不过却改善了游戏整体的体验。现在对象碰撞时，就会出现视觉特效。

## 17.11 游戏结束

我们之前提到过，游戏当前还有一个小问题。当生命条数为零时，我们需要结束游戏。游戏结束后要创建一个新的场景类以进行转场。你之前见过我们从某一关卡进入下一关卡的转场。这次也相似，不过需要一个新的类。

请创建一个新的 Objective-C 类。使用 SKScene 作为父类并将新类命名为 BIDGameOverScene。

我们将要开始实现一个非常简单的功能，仅显示 Game Over 文本。我们在 BIDGameOverScene.m 的 @implementation 部分添加以下代码来完成这个功能：

```

- (instancetype)initWithSize:(CGSize)size {
    if (self = [super initWithSize:size]) {
        self.backgroundColor = [SKColor purpleColor];
        SKLabelNode *text = [SKLabelNode labelNodeWithFontNamed:@"Courier"];
    }
}

```

```

        text.text = @"Game Over";
        text.fontColor = [SKColor whiteColor];
        text.fontSize = 50;
        text.position = CGPointMake(self.frame.size.width * 0.5,
                                    self.frame.size.height * 0.5);

        [self addChild:text];
    }
    return self;
}

```

现在切换回 BIDLevelScene.m。需要在顶端导入新场景的头文件：

```

#import "BIDLevelScene.h"
#import "BIDPlayerNode.h"
#import "BIDEnemyNode.h"
#import "BIDBulletNode.h"
#import "SKNode+Extra.h"
#import "BIDGameOverScene.h"

```

游戏结束时的基本操作由这个新方法来定义。这里显示了一个爆炸效果并转场到刚刚创建的新场景中：

```

- (void)triggerGameOver {
    self.finished = YES;

    NSString *path = [[NSBundle mainBundle] pathForResource:@"EnemyExplosion"
                                                            ofType:@"sks"];
    SKEmitterNode *explosion = [NSKeyedUnarchiver unarchiveObjectWithFile:path];
    explosion.numParticlesToEmit = 200;
    explosion.position = _playerNode.position;
    [self addChild:explosion];
    [_playerNode removeFromParent];
    SKTransition *transition = [SKTransition doorsOpenVerticalWithDuration:1.0];
    SKScene *gameOver = [[BIDGameOverScene alloc] initWithSize:self.frame.size];
    [self.view presentScene:gameOver transition:transition];
}

```

接下来创建这个新的方法来检测游戏是否结束，如果是的话则调用 triggerGameOver 方法，返回 YES 表示结束，返回 NO 表示游戏仍在进行：

```

- (BOOL)checkForGameOver {
    if (self.playerLives == 0) {
        [self triggerGameOver];
        return YES;
    }
    return NO;
}

```

最后在已有的 update:方法中添加检测方法。它会检测游戏的结束状态，只有在游戏仍在进行时才会检测是否要进入下一等级。否则就会出现错误，如果关卡内的最后一个敌人取走了玩家的最后一条生命，就会同时触发两个转场！

```

- (void)update:(CFTimeInterval)currentTime {
    if (self.finished) return;
}

```

```

[self updateBullets];
[self updateEnemies];
if (![self checkForGameOver]) {
    [self checkForNextLevel];
}
}

```

现在再次运行应用,让坠落的敌人攻击你的飞船5次,你将会看到游戏结束的界面,如图 17-14 所示。



图 17-14 胜败乃兵家常事,大侠请重新来过

## 17.12 开始场景

这样就会有另一个问题:游戏结束之后应该做什么?我们可以让玩家轻点屏幕以重新开始游戏,不过在想到这里时,一个想法穿过大脑。为什么不让游戏有一个开始界面?这样玩家就不用启动时间后直接开始游戏。而且游戏的结束屏幕可以再回到这里。很明显这个方法是可行的。现在创建另一个新的 Objective-C 类,再次使用 SKScene 作为父类,这次将其命名为 BIDStartScene。

我们要在这里创建一个非常简单的开始界面。它只是显示一些文本,并在用户轻点任意位置时开始游戏。在这里添加所有粗体代码来完成这个类:

```

#import "BIDStartScene.h"
#import "BIDLevelScene.h"

@implementation BIDStartScene

- (instancetype)initWithSize:(CGSize)size {

```

```

if (self = [super initWithSize:size]) {
    self.backgroundColor = [SKColor greenColor];

    SKLabelNode *topLabel = [SKLabelNode labelNodeWithFontNamed:@"Courier"];
    topLabel.text = @"TextShooter";
    topLabel.fontColor = [SKColor blackColor];
    topLabel.fontSize = 48;
    topLabel.position = CGPointMake(self.frame.size.width * 0.5,
                                    self.frame.size.height * 0.7);
    [self addChild:topLabel];

    SKLabelNode *bottomLabel = [SKLabelNode labelNodeWithFontNamed:
                                @"Courier"];
    bottomLabel.text = @"Touch anywhere to start";
    bottomLabel.fontColor = [SKColor blackColor];
    bottomLabel.fontSize = 20;
    bottomLabel.position = CGPointMake(self.frame.size.width * 0.5,
                                       self.frame.size.height * 0.3);
    [self addChild:bottomLabel];
}
return self;
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    SKTransition *transition = [SKTransition doorwayWithDuration:1.0];
    SKScene *game = [[BIDLevelScene alloc] initWithSize:self.frame.size];
    [self.view presentScene:game transition:transition];
}

@end

```

现在回到 BIDGameOverScene.m, 这样我们可以让游戏结束界面转场到开始界面。导入这个头文件:

```

#import "BIDGameOverScene.h"
#import "BIDStartScene.h"

```

然后添加以下两个方法:

```

- (void)didMoveToView:(SKView *)view {
    [self performSelector:@selector(goToStart) withObject:nil afterDelay:3.0];
}

- (void)goToStart {
    SKTransition *transition = [SKTransition flipVerticalWithDuration:1.0];
    SKScene *start = [[BIDStartScene alloc] initWithSize:self.frame.size];
    [self.view presentScene:start transition:transition];
}

```

didMoveToView:方法实现之后就会在场景中被调用。这里仅仅触发一个三秒钟的暂停, 接着转场回到开始场景。

还有一个问题就是要让场景能正常地转场到另一场景。我们需要更改应用的启动程序, 这样它就不会直接进入游戏, 而是显示开始界面。回到 BIDViewController.m, 在这里先导入开始场景



的头文件：

```
#import "BIDViewController.h"
#import "BIDLevelScene.h"
#import "BIDStartScene.h"
```

然后在 `viewDidLoad` 方法中，替换场景的名称：

```
// 创建并配置场景
— SKScene * scene = [BIDLevelScene sceneWithSize:skView.bounds.size];
SKScene * scene = [BIDStartScene sceneWithSize:skView.bounds.size];
```

现在来试试看吧！启用应用，你会很开心地看到开始界面。触摸屏幕，进行游戏，失去生命，你将会进入游戏结束场景。等待几秒钟后，你将回到开始界面，如图 17-15 所示。

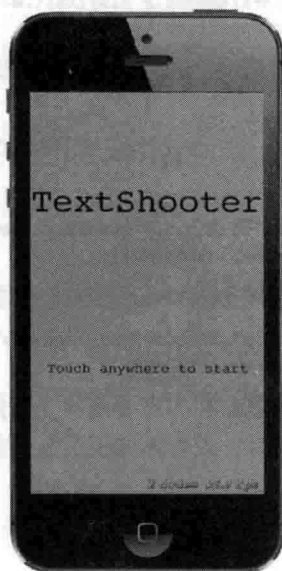


图 17-15 最后我们创建了开始界面

## 17.13 播放音乐

好吧，还有一件事。我们做的是视频游戏，视频游戏应该要有声音，不过我们的游戏却非常安静！所幸 `Sprite Kit` 包含了音频播放代码，它非常容易使用。首先找到本章的代码归档，并找到准备好的音频文件：`enemyHit.wav`、`gameOver.wav`、`gameStart.wav`、`playerHit.wav` 和 `shoot.wav`。将它们都拖入 Xcode 的项目导航栏中。

**注意** 这些音效使用了非常棒的开源 CFXR 应用创建（可以从<https://github.com/nevyn/cfxr>中找到）。如果需要奇特的音效，恐怕没有比 CFXR 更合适的了！

现在我们将简单播放这些音效。首先在 BIDBulletNode.m 的 bulletFrom:toward:方法末尾的 return 语句之前添加以下粗体显示的代码:

```
[bullet runAction:[SKAction playSoundFileNamed:@"shoot.wav"  
                             waitForCompletion:NO]]];
```

接下来切换到 BIDEnemyNode.m, 在 receiveAttacker:contact:方法的末尾添加这些代码:

```
[self runAction:[SKAction playSoundFileNamed:@"enemyHit.wav"  
                             waitForCompletion:NO]]];
```

现在对 BIDPlayerNode.m 执行非常类似的操作, 在 receiveAttacker:contact:方法的末尾添加以下代码:

```
[self runAction:[SKAction playSoundFileNamed:@"playerHit.wav"  
                             waitForCompletion:NO]]];
```

这些游戏内的声音足够满足此时需要。这时继续运行游戏。我觉得你也会同意给粒子加上声音会给游戏更好的体验。

现在为游戏开始和游戏结束添加一些音效。在 BIDStartScene.m 的 touchesBegan:withEvent:方法结尾添加这些代码:

```
[self runAction:[SKAction playSoundFileNamed:@"gameStart.wav"  
                             waitForCompletion:NO]]];
```

最后在 BIDLevelScene.m 的 triggerGameOver 方法末尾添加这些代码。

```
[self runAction:[SKAction playSoundFileNamed:@"gameOver.wav"  
                             waitForCompletion:NO]]];
```

现在当你运行游戏, 会被各种哔哩哔哩的声音淹没, 就像当你还是孩子时游戏所发出的声音那样! 不过, 也许是你父母还是孩子时, 或者你的祖父! 请相信我, 过去所有游戏的声音都是像这样。

## 17.14 小结

虽然 TextShooter 外观上很简单, 不过你在本章中学到的技术是所有使用 Sprite Kit 开发游戏的基础。你学到了如何在多个节点类中管理代码, 使用节点将群组对象归纳在一起等知识。你还尝试了逐步构建游戏的功能, 探索了每个步骤。当然我们没有向你展示经历的失误, 这本书可有几百页呢。不过即便算上这些, 这个应用都是从无到有构建出来的, 只用了几个小时, 得以在这一章中展示出来。

使用 Sprite Kit 可以在短时间内构建许多架构。如你所见, 你可以在没有方便的图片时使用基于文本的 sprite, 而且如果想在后面使用真实的图像来替换它也没有问题。之前的读者还发觉了可以插入苹果浏览输入源的 emoji 符号来代替旧的 ASCII 普通字符文本。这个就作为测试留给读者来完成吧!

清晰明亮并且支持触摸操作的 iPhone、iPod touch 和 iPad 屏幕确实很漂亮，绝对是工程设计的精巧杰作，并且所有 iOS 设备共有的多点触控屏幕赋予了平台无与伦比的易用性。由于该屏幕可以同时检测多点触控并且可以各自跟踪这些触控，因此应用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。

假设在某个电子邮件应用中浏览收件箱，并且决定删除某封电子邮件。你有多个选择，比如可以分别轻点每封电子邮件，轻点废纸篓图标删除该邮件，然后等待下载下一封邮件，像这样依次删除每封邮件。如果希望在删除每封电子邮件之前阅读它们，你最好使用此方法。

另外一种方式是从电子邮件列表轻点右上角的“编辑”按钮，轻点每个电子邮件行以进行标记，然后轻点“删除”按钮删除所有标记的邮件。如果不需要在删除每封电子邮件之前阅读它们，你最好使用此方法。还可以在列表中某封电子邮件行上，用轻扫的手势由右至左轻扫。此手势将会生成针对这封邮件的一个“更多”按钮和一个“删除”按钮。若你轻点“删除”按钮，该邮件将会被删除。

本例只介绍了通过多点触控屏幕可以完成的无数手势中的几个。还可以将手指捏合在一起来缩小图片，或者分开手指放大图片，也可以在主屏幕上长时间按住一个图标，开启“抖动”模式，以便从 iOS 设备上删除应用。

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。

## 18.1 多点触控术语

在钻研体系结构之前，先来看一下一些基本词汇。手势指的是从你用一个或多个手指接触屏幕时开始，直到手指离开屏幕为止所发生的所有事件。无论手势持续多长时间，只要一个或多个手指仍然在屏幕上，这个手势就存在（除非传入电话呼叫等系统事件中断了该手势）。注意，Cocoa Touch 没有公开任何代表手势的类或结构。从某种意义上来说，手势就是一个动作，运行中的应用可以从用户输入流知道是否出现某种手势。

手势通过一系列事件在系统内传递信息。用户在与设备的多点触控屏幕交互时会触发一系列事件。事件包含与发生的一次或多次触摸相关的信息。

触摸是指把手指放到 iOS 设备的屏幕上，从屏幕上拖动或抬起的这样一种行为。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 5 个手指都放到屏幕上，只要这些手指彼此不是靠太近，iOS 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，但是你应该知道 iOS 能够处理这种情况（如有必要的话）。实验表明，iPad 可以处理同时发生的 11 处触摸。看起来似乎太多了，但这可能是有用的。例如，在多个玩家参与的游戏里，可能同时会有好几个玩家与屏幕进行交互。

当用一个手指触摸屏幕，然后立即将该手指从屏幕移开（而不是来回移动），这就是轻点。iOS 设备能够跟踪轻点的数量，并且可以区分用户究竟是轻点了 2 次还是 3 次，甚至 20 次！例如，它能够通过区分两次单击和一次双击，来处理所有的计时工作以及其他必要的工作。

手势识别器是一个对象，它知道如何观察用户生成的事件流，并能够识别用户何时以与预定义的手势相匹配的方式进行了触摸和拖动。在检测常见手势时，`UIGestureRecognizer` 类及其各种子类可节省大量工作。`UIGestureRecognizer` 类很好地封装了查找手势的功能，而且可以方便地应用于应用中的任何视图。

## 18.2 响应者链

由于手势是在事件之内传递到系统的，而事件会通过响应者链（responder chain）进行传递，因此你需要了解响应者链的工作方式，以便能够正确地处理手势。如果你使用过 Cocoa for Mac OS X，很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。如果这对于你来说是新知识，也不必担心，我们会解释它的工作原理。

### 18.2.1 响应事件

在本书中，我们已经多次提到过第一个响应者，该响应者通常是用户当前正在交互的对象。第一个响应者是响应者链的起点，除此之外，响应者链中通常还有其他响应者。在一个运行的应用中，响应者链是一个可变的能够响应用户事件的对象集合。以 `UIResponder` 作为超类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，这意味着它也是响应者，其所有子类（如 `UINavigationController` 和 `UITabBarController`）也都是响应者。响应者就是这样命名的，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），那么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的传递过程。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，则该事件被丢弃。

下面让我们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，那么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，那么该事件被传递给第一个响应者的父视图。如果父视图没有响应，则该事件将被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，任何视图或控制器都没有对其进行处理，那么该事件将会被传递给应用的窗口。如果窗口不处理该事件，则该窗口会将该事件传递给应用的对象实例 `UIApplication`。

如果 `UIApplication` 也不响应该事件，那么还有一个地方可以构建一个全局响应者作为响应链的最后一环，那就是应用委托。如果应用委托是 `UIResponder` 的子类（如果你是通过苹果公司的应用模板来创建项目的，那么通常都是如此），那么应用会尝试将任意尚未处理的事件传递给它。最后，如果应用委托不是 `UIResponder` 的子类，或者不处理这个事件，那么这个事件将会被丢弃。

这个过程非常重要，原因有多个。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，那么该视图或控件将有机会进行响应。如果它没有响应，则表视图单元则将有机会进行响应。在某个应用（如邮件应用）中，我们可以使用轻扫操作删除某封邮件，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并不响应手势，如果它们不响应，那么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响应者链的结尾为止。

## 18.2.2 转发事件：保持响应者链的活动状态

让我们回顾邮件应用中的表视图单元。我们不知道苹果公司邮件应用的内部细节，但是暂且可以认为表视图单元支持（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（稍后介绍）相关的方法，以便它可以检查判断该事件是否为轻扫手势的一部分。如果该事件与表视图相应的轻扫手势一致，那么表视图单元会采取操作，该事件将停止传递。

如果该事件与表视图单元的轻扫手势不相符，那么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，那么表和链上的其他对象将永远不会获得响应的机会，并且该应用可能无法如用户所期望地那样正常工作。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。如果某个对象截获了无法处理的事件，那么就需要手动地将该对象继续向下传递，并在下一个响应者上调用相同的方法。下面是其中一小部分代码：

```
- (void)respondToFictionalEvent:(UIEvent *)event
{
    if ([self shouldHandleEvent:event]) {
        [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。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，需要确保将事件传递给响应者链中接下来的节点，这一点非常重要。

## 18.3 多点触控体系结构

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，封装着相关手势的事件沿着响应者链传递。这意味着响应者链的对象中需要包含代码来处理与多点触控屏幕进行的任意种类的交互。一般来说，这意味着我们可以将该代码嵌入到 `UIView` 的子类中，也可以将该代码嵌入到 `UIViewController` 中。

那么该代码属于视图还是视图控制器？

如果视图需要根据用户的触摸对自己执行某些操作，那么代码可能属于定义该视图的类。例如，很多控件类（如 `UISwitch` 和 `UISlider`）都能够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` 可以响应触摸动作。

但是，通常当正在处理的手势影响正在触摸的多个对象时，该手势代码实际上应属于相关的视图控制器类。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，那么应该由视图控制器中的代码处理该手势。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。

## 18.4 4 个手势通知方法

我们可以使用 4 个方法通知响应者有关触摸和手势的情况，它们是 `touchesBegan:withEvent:`、`touchesMoved:withEvent:`、`touchesEnded:withEvent:` 和 `touchesCancelled:withEvent:`。当用户第一次触摸屏幕时，iOS 设备将查找 `touchesBegan:withEvent:` 方法的响应者。若要查清用户第一次开始手势或轻点屏幕的时间，请在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];

    // 在此处进行一些处理。
}
```

该方法以及所有与触摸有关的方法都接受一个名为 `touches` 的 `NSSet` 实例和一个 `UIEvent` 实例。你可以通过获取 `touches` 中的对象数确定当前按压屏幕的手指数量。`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点的一部分，那么我们可以通过询问任意 `UITouch` 对象查询轻点数量。在前面的示例中，`numTaps` 为 2 代表快速连续轻点屏幕两次；同理如果 `numTouches` 为 2，那么用户只是同时用两个手指轻点屏幕一次；如果两个值均为 2，这代表用户用两个手指进行了双击操作。



`touches` 中的所有对象都可能与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。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`：

```
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，你甚至可以将点转换为视图的本地坐标系：

```
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:` 获得通知。在长时间的拖动过程中应用会多次调用该方法，并且每次调用该方法时都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，你还可以查清该触摸原来的位置，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应用会调用另一个事件，即 `touchesEnded:withEvent:`。调用该方法表示用户在结束某个手势。

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（如来电呼叫）导致手势中断时，该方法会被调用。你可以在此处进行任何清理工作，以便可以重新开始一个新手势。该方法被调用时，对于当前手势，应用将不会调用 `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。

## 18.5 TouchExplorer 应用

我们将构建一个小应用，让你更好地体会 4 个与触摸有关的响应者方法的调用时机。在 Xcode 中，我们使用 Single View Application 模板创建新项目，将 Product 名设为 TouchExplorer，在 Devices 弹出菜单中选择 iPhone。

每次调用与触摸有关的方法时，TouchExplorer 都会将触摸和轻点计数的消息显示到屏幕中（参见图 18-1）。

---

**注意** 尽管本章中的应用将在模拟器上运行，但是 you 无法看到所有可用的多点触控功能，除非是在真实的 iOS 设备上运行这些应用。如果你已经是苹果的 iOS 开发者计划的付费会员，那么可以在自己选择的设备上运行你编写的程序。苹果公司网站详细说明了如何将 Xcode 连接到具体设备的准备过程。

---



图 18-1 TouchExplorer 应用

我们需要为该应用提供 3 个标签：一个用于指示最后调用的方法，一个用于报告当前的轻点计数，一个用于报告触摸数量。请单击 BIDViewController.m 文件，并在顶部的类扩展中添加 3 个输出接口。

```
#import "BIDViewController.h"
```

```
@interface BIDViewController ()
```

```
@property (weak, nonatomic) IBOutlet UILabel *messageLabel;
```

```
@property (weak, nonatomic) IBOutlet UILabel *tapsLabel;
```

```
@property (weak, nonatomic) IBOutlet UILabel *touchesLabel;
```

```
@end
```

现在，请选择 Main.storyboard 以编辑 GUI 界面。你会看到这个模板的新项目中都会包含的空视图。我们将一个标签拖至该视图，使用蓝色引导线将标签放置在视图左上角，然后使用标签右侧边缘调节手柄调整它的大小，将其拉伸至右边的蓝色引导线处。接着，使用属性检查器将标签的对齐方式设为居中。最后，我们按下 option 键从这个原始标签另外再拖出两个标签，依次将它们放置在前一个标签的下方，这样你就有了 3 个标签（参见图 18-1）。

接下来，我们按住鼠标右键从 View Controller 图标分别拖到这 3 个标签上，将一个连接到 messageLabel 输出接口，一个连接到 tapsLabel 输出接口，一个连接到 touchesLabel 输出接口。

如果对外观有要求，可以随意修改字体和颜色。完成之后，请分别双击它们，按下 delete 键将其中的文本删除。

最后，单击当前视图的背景或文件大纲中的 View 图标，然后打开属性检查器（参见图 18-2）。在属性检查器中，在 View 部分的底部，确保同时选中 User Interacting Enabled 和 Multiple Touch。如果未选中 Multiple Touch，控制器类的触摸方法将始终接受一个手指的触摸并且只接受一个手



指的触摸，无论实际上有多少个手指在触摸手机的屏幕。



图 18-2 在视图属性检查器中，确保同时选中 User Interacting Enabled 和 Multiple Touch

完成后，请回到 BIDViewController.m 并在文件开头添加以下代码：

```
@implementation BIDViewController
```

```
-(void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后（通常是从 nib 文件加载），做一些额外的设置
}
```

```
-(void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // 删除可以重新创建的资源
}
```

```
-(void)updateLabelsFromTouches:(NSSet *)touches
{
    NSUInteger numTaps = [[touches anyObject] tapCount];
    NSString *tapsMessage = [[NSString alloc]
                             initWithFormat:@"%d taps detected", numTaps];
    self.tapsLabel.text = tapsMessage;
    NSUInteger numTouches = [touches count];
    NSString *touchMsg = [[NSString alloc] initWithFormat:
                          @"%d touches detected", numTouches];
    self.touchesLabel.text = touchMsg;
}
```

```
#pragma mark - Touch Event Methods
```

```
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    self.messageLabel.text = @"Touches Began";
    [self updateLabelsFromTouches:touches];
}
```

```
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
{
    self.messageLabel.text = @"Touches Cancelled";
    [self updateLabelsFromTouches:touches];
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    self.messageLabel.text = @"Touches Ended.";
    [self updateLabelsFromTouches:touches];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    self.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那 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，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点计数，通过查看 touches 集的计数计算出触摸的数量，并用该信息更新标签。

编译并运行该应用。如果在模拟器中运行，请尝试反复单击屏幕以增加轻点计数，并在视图中拖动时尝试单击并按住鼠标按钮，以模拟触摸和拖动。注意，拖动与轻点不同，所以开始拖动时，应用将报告轻点次数为 0。

可以在用鼠标单击并进行拖动时按下 option 键，以此在 iOS 模拟器中模仿两个手指捏合的手势。另外，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。你将无法使用需要 3 个或更多个手指才能完成的手势，但可以在模拟器上使用 option 和 shift 键的组合进行大多数用到两个手指的手势。

如果能够在某个设备上运行该程序，请看看这个程序最多能够同时识别多少个手指的触摸。请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，并请尝试轻点两次和轻点 3 次屏幕，看看通过用两个手指轻点是否可以增加轻点计数。

尝试使用 TouchExplorer 应用，直到了解并适应 4 个触摸方法的工作方式为止。完成之后，再来看看如何检测最常用的一个手势：轻扫。

## 18.6 Swipes 应用

这里将要构建的应用叫 Swipes，它只能检测水平和垂直轻扫这两种手势。如果将手指从左到右、从右到左、从上到下或从下到上滑过屏幕，Swipes 就会在屏幕顶部显示一条消息（并持续几秒钟），提示已检测到轻扫（参见图 18-3）。

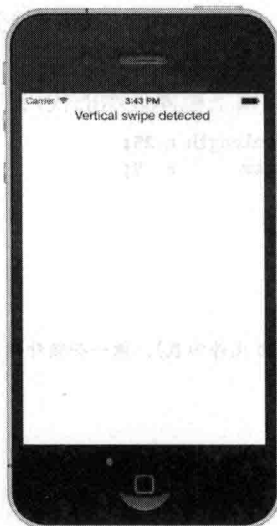


图 18-3 Swipes 应用将检测水平和垂直方向上的轻扫动作

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，也就是将该手势算作轻扫之前，用户必须轻扫过的距离。此处还将定义一个偏差，即用户可以最多偏离某条直线多远，其操作仍然可以算作水平或垂直轻扫。通常我们不将偏移到对角线方向的手势算作轻扫，但是如果只是与水平或垂直方向偏离一点，就仍会将其视为轻扫手势。

当用户触摸屏幕时，第一次触摸的位置将被保存在变量中。然后，当用户手指滑过屏幕时，我们将进行检查，看它是否达到某个点以及这个点是否足够远且足够直，以至于能够将其算作轻扫。来构建该应用吧。

我们再次使用 Single View Application 模板在 Xcode 中创建一个新项目，在 Devices 弹出菜单中选择 iPhone，这次将该项目命名为 Swipes。

请单击 BIDViewController.m 并在顶部附近的类扩展中添加以下代码：

```
#import "BIDViewController.h"

@interface BIDViewController ()

@property (weak, nonatomic) IBOutlet UILabel *label;
@property (nonatomic) CGPoint gestureStartPoint;

@end
```

我们为一个标签声明了一个输出接口和一个容纳用户触摸的第一个点的变量。然后声明一个方法，当文本显示几秒钟之后，该方法会将其清除。

请选中 Main.storyboard，打开该文件进行编辑。请一定使用属性检查器设置视图，同时选中 Users Interaction Enabled 和 Multiple Touch，并从库中拖出一个标签到 View 窗口的上面位置。我们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）。请随意使用文本属性使

该标签易读。按住鼠标右键的同时从 View Controller 图标拖到该标签上，并将其连接到该标签输出接口。最后，请双击该标签并删除其文本。

回到 BIDViewController.m 并添加以下粗体显示的代码：

```
static CGFloat const kMinimumGestureLength = 25;
static CGFloat const kMaximumVariance      = 5;

@implementation BIDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后（通常是从 nib 文件加载），做一些额外的设置
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // 删除可以重新创建的资源
}

- (void)eraseText
{
    self.label.text = @"";
}

#pragma mark - Touch Handling

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    self.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(self.gestureStartPoint.x - currentPosition.x);
    CGFloat deltaY = fabsf(self.gestureStartPoint.y - currentPosition.y);

    if (deltaX >= kMinimumGestureLength && deltaY <= kMaximumVariance) {
        self.label.text = @"Horizontal swipe detected";
        [self performSelector:@selector(eraseText)
            withObject:nil afterDelay:2];
    } else if (deltaY >= kMinimumGestureLength &&
        deltaX <= kMaximumVariance){
        self.label.text = @"Vertical swipe detected";
        [self performSelector:@selector(eraseText) withObject:nil
            afterDelay:2];
    }
}

@end
```

我们首先来看 `touchesBegan:withEvent:` 方法。此处所要做的就是从 `touches` 集中获得触摸并存储它的点。现在主要关注一个手指的轻扫，因此不必关注触摸数量；我们只需要获取其中之一即可。

```
UITouch *touch = [touches anyObject];
self.gestureStartPoint = [touch locationInView:self.view];
```

在接下来的方法 `touchesMoved:withEvent:` 中，我们进行实际的工作，首先获取用户手指的当前位置：

```
UITouch *touch = [touches anyObject];
CGPoint currentPosition = [touch locationInView:self.view];
```

之后，计算从起始位置开始，用户手指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，它能够返回一个类型为 `float` 的绝对值。这允许我们从一个值中减去另一个，而不必关心哪个值较高：

```
CGFloat deltaX = fabsf(self.gestureStartPoint.x - currentPosition.x);
CGFloat deltaY = fabsf(self.gestureStartPoint.y - currentPosition.y);
```

获得两个增量之后，判断用户在两个方向上所移动过的距离，检测用户是否在一个方向上移动得足够远，但在另一个方向上移动得却不够，以便形成轻扫动作。如果是这样，请设置标签的文本以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手势还是最近的手势：

```
if (deltaX >= kMinimumGestureLength && deltaY <= kMaximumVariance) {
    self.label.text = @"Horizontal swipe detected";
    [self performSelector:@selector(eraseText)
        withObject:nil afterDelay:2];
} else if (deltaY >= kMinimumGestureLength &&
    deltaX <= kMaximumVariance){
    self.label.text = @"Vertical swipe detected";
    [self performSelector:@selector(eraseText) withObject:nil
        afterDelay:2];
}
```

继续，请编译并运行应用。如果发现自己进行了单击和拖动，但却没有看到结果，请耐心一点。单击并垂直向下或水平拖动，直到熟悉轻扫操作。

### 18.6.1 自动手势识别

用于检测轻扫手势的过程并不是太糟糕。所有复杂性都包含在 `touchesMoved:withEvent:` 方法中，并且它不是那么令人费解。不过，还可以采用另一种更加轻松的方法完成此工作。iOS 现在包含一个名为 `UIGestureRecognizer` 的类，从而根本不必再用观察所有事件来查看手指如何移动了。其原理是不直接使用 `UIGestureRecognizer` 类，而是创建它的一个子类的实例；其每个子类用于查找特定类型的手势，比如轻扫、捏合、双击、三击等。

看一下如何修改 Swipes 应用，使用手势识别器委托代替手工过程。与平常一样，此处可以

复制 Swipes 项目文件夹并开始修改。

首先, 请选择 BIDViewController.m, 删除 touchesBegan:withEvent: 和 touchesMoved:withEvent: 方法, 因为这里不再需要它们了。之后, 在它们的位置上添加两个新方法:

```
- (void)reportHorizontalSwipe:(UIGestureRecognizer *)recognizer
{
    self.label.text = @"Horizontal swipe detected";
    [self performSelector:@selector(eraseText) withObject:nil afterDelay:2];
}

- (void)reportVerticalSwipe:(UIGestureRecognizer *)recognizer
{
    self.label.text = @"Vertical swipe detected";
    [self performSelector:@selector(eraseText) withObject:nil afterDelay:2];
}
```

这些方法实现轻扫手势的实际“功能”(假设可以这样称呼它), 就像 touchesMoved:withEvent: 方法之前所做的一样。现在, 我们向 viewDidLoad 方法添加以下代码:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后 (通常是从 nib 文件加载), 做一些额外的设置
    UISwipeGestureRecognizer *vertical =
        [[UISwipeGestureRecognizer alloc]
         initWithTarget:self action:@selector(reportVerticalSwipe)];
    vertical.direction = UISwipeGestureRecognizerDirectionUp|
        UISwipeGestureRecognizerDirectionDown;
    [self.view addGestureRecognizer:vertical];

    UISwipeGestureRecognizer *horizontal =
        [[UISwipeGestureRecognizer alloc]
         initWithTarget:self action:@selector(reportHorizontalSwipe)];
    horizontal.direction = UISwipeGestureRecognizerDirectionLeft|
        UISwipeGestureRecognizerDirectionRight;
    [self.view addGestureRecognizer:horizontal];
}
```

就这么简单! 要进一步改进该应用, 你也可以删除 BIDViewController.h 和 BIDViewController.m 中引用 gestureStartPoint 的代码行 (但将它们保留在那里也没有任何害处)。得益于 UIGestureRecognizer, 这里只需要创建和配置一些手势识别器, 并将它们添加到视图中。当用户以识别器可识别的方式与屏幕交互时, 我们指定的操作方法就会被调用。

在代码总量方面, 对于像本例这样的简单情况, 使用手势识别器与使用以前的方法没有太大区别。但手势识别器的使用无疑更易于理解和编程。甚至无需考虑计算手指运动的问题, 因为 UISwipeGestureRecognizer 可以完成此任务。更好的是, 苹果的手势识别系统是可扩展的, 这意味着假如应用需要用到苹果的手势识别器所没有涉及的复杂手势, 你可以自己对识别器进行自定义, 将识别器中复杂的代码 (我们之前看到的那几行) 替换成你的视图控制器代码。我们将在本章后面构建一个这样的示例。

## 18.6.2 实现多指轻扫

在 Swipes 应用中,我们仅关心单指轻扫,因此只从 `touches` 集中获取某个对象来计算在轻扫期间用户手指的位置。如果只对单指轻扫感兴趣(这是最常见的轻扫类型),则该方法非常合适。

但是,如果我希望处理双指或三指轻扫,该怎么办?在本书最早的那几版中,我们为此专门编写了大约 50 行代码以及大量的说明,通过跨多个触摸事件跟踪多个 `UITouch` 实例来实现这些手势。幸好,现在手势识别器可以解决这些问题了。经过配置后,`UISwipeGestureRecognizer` 可识别同时执行的任意数量手指的触摸。默认情况下,每个实例关注一个手指,但可以将它配置为关注同时按压屏幕的任意数量的手指。每个实例仅响应所指定准确数量的触摸。所以,为了更新应用,将在一个循环中创建大量手势识别器。

再次复制 Swipes 项目文件夹。

请编辑 `BIDViewController.m` 并修改 `viewDidLoad` 方法,将它替换为如下所示的代码:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后 (通常是从 nib 文件加载),
    // 做一些额外的设置。
    for (NSUInteger touchCount = 1; touchCount <= 5; touchCount++) {
        UISwipeGestureRecognizer *vertical;
        vertical = [[UISwipeGestureRecognizer alloc]
                    initWithTarget:self action:@selector(reportVerticalSwipe:)];
        vertical.direction = UISwipeGestureRecognizerDirectionUp
        | UISwipeGestureRecognizerDirectionDown;
        vertical.numberOfTouchesRequired = touchCount;
        [self.view addGestureRecognizer:vertical];

        UISwipeGestureRecognizer *horizontal;
        horizontal = [[UISwipeGestureRecognizer alloc]
                     initWithTarget:self action:@selector(reportHorizontalSwipe:)];
        horizontal.direction = UISwipeGestureRecognizerDirectionLeft
        | UISwipeGestureRecognizerDirectionRight;
        horizontal.numberOfTouchesRequired = touchCount;
        [self.view addGestureRecognizer:horizontal];
    }
}
```

注意,在真实的应用中可能需要不同数量的手指在屏幕上轻扫来触发不同的行为。我们可以使用手势识别器轻松完成此任务,让每个识别器调用不同的操作方法即可。

现在,我们所需做的是更改日志,添加一个方法来提供对触摸数量的描述并将它用于“报告”方法中,如下所示。请将此方法添加到 `BIDViewController` 类的底部,两个轻扫报告方法之前:

```
- (NSString *)descriptionForTouchCount:(NSUInteger)touchCount
{
    switch (touchCount) {
        case 1:
            return @"Single";
        case 2:
            return @"Double ";
    }
}
```

```

        case 3:
            return @"Triple ";
        case 4:
            return @"Quadruple ";
        case 5:
            return @"Quintuple ";
        default:
            return @"";
    }
}

```

接下来，修改这两个轻扫报告方法：

```

- (void)reportHorizontalSwipe:(UIGestureRecognizer *)recognizer
{
    self.label.text = @"Horizontal swipe detected";
    self.label.text = [NSString stringWithFormat:@"%sHorizontal swipe detected",
        [self descriptionForTouchCount:
            [recognizer numberOfTouches]]];
    [self performSelector:@selector(eraseText) withObject:nil afterDelay:2];
}

- (void)reportVerticalSwipe:(UIGestureRecognizer *)recognizer
{
    self.label.text = @"Vertical swipe detected";
    self.label.text = [NSString stringWithFormat:@"%sVertical swipe detected",
        [self descriptionForTouchCount:
            [recognizer numberOfTouches]]];
    [self performSelector:@selector(eraseText) withObject:nil afterDelay:2];
}

```

请编译并运行应用。你应该能够在两个方向上触发 2 个手指和 3 个手指的轻扫，并且仍然能够触发单指轻扫。如果你的手指比较小，甚至可以触发 4 个手指和 5 个手指的轻扫。

---

**提示** 在模拟器中，如果你按着 option 键，将会出现一对小圆点，它们代表了两个手指。若你将它们靠近，然后按下 shift 键，这样会使它们之间的相对位置就会保持不变，你可以将它们在屏幕上任意移动。现在，请点击并向下拖动屏幕，这样就可以模拟双指轻扫了。这真是太棒了！

---

在多手指轻扫时，需要注意一件事：手指不能彼此太过靠近。如果两个手指彼此靠得非常近，那么它们可能被注册为一个单指触摸。因此，我们建议不要依赖 4 个手指或 5 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 个或 5 个手指的轻扫。不过，iPad 上一些四指和五指手势是系统默认开启的，用来在应用之间切换或回到主屏幕。这些可以在“设置”应用中关闭，如果应用中不会用到这些手势的话，推荐关闭这个设置。

## 18.7 检测多次轻点

在 TouchExplorer 应用中，我们将轻点次数打印到了屏幕上，看到了吗，检测多次轻点是多



么简单！但是，它并不像看上去那样简单，因为我们通常希望根据轻点的数量采取不同的操作。如果用户连续轻点 3 次，那么程序会分 3 次单独通知你。你将得到轻点 1 次、轻点 2 次，最后是轻点 3 次的通知。如果你想对 2 次轻点执行某些完全不同于 3 次轻点的操作，3 个独立的通知可能会引起问题。

幸好，苹果公司的工程师预料到了这一情形，提供了一种机制来让多个手势识别器协同运行——即使出现看起来可能触发它们中的任何一个的模糊输入，也能正常运作。这里的基本理念是：在一个手势识别器上设置一个限制，告诉它：除非其他某个手势识别器未能识别关联的方法，否则不要触发自己的关联方法。

这看起来有点抽象，我们具体分析一下。一种常用的手势识别器是使用 `UITapGestureRecognizer` 类表示的。轻点手势识别器可配置为在发生特定数量的轻点时执行某种操作。假设我们有一个视图，并希望将其定义当用户进行一次或两次轻点时产生不同行为，那么可以首先编写以下代码：

```
UITapGestureRecognizer *singleTap = [[UITapGestureRecognizer alloc]
                                       initWithTarget:self
                                       action:@selector(doSingleTap)];

singleTap.numberOfTapsRequired = 1;
[self.view addGestureRecognizer:singleTap];

UITapGestureRecognizer *doubleTap = [[UITapGestureRecognizer alloc]
                                       initWithTarget:self
                                       action:@selector(doDoubleTap)];

doubleTap.numberOfTapsRequired = 2;
[self.view addGestureRecognizer:doubleTap];
```

这段代码的问题在于，两个识别器不会感知到彼此，而且无法知道用户的操作可能更适合于另一个识别器。使用前面的代码，如果用户两次轻点视图，`doDoubleTap` 方法将会被调用，但 `doSingleMethod` 也会被调用两次，每次调用针对每次轻点。

此问题的解决方式是建立失败需求。我们告诉 `singleTap` 希望它仅在 `doubleTap` 未识别时触发自己的操作，并使用以下这行代码响应用户输入：

```
[singleTap requireGestureRecognizerToFail:doubleTap];
```

这意味着当用户轻点一次时，`singleTap` 不会立即执行自己的工作。相反，`singleTap` 会等到获知 `doubleTap` 已决定停止关注当前的手势（用户没有轻点两次）时执行。我们将在下一个项目中进一步探讨此主题。

在 Xcode 中，请使用 Single View Application 模板创建一个新的项目，将此新项目命名为 TapTaps，并在 Devices 弹出菜单中选中 iPhone。

该应用将拥有 4 个标签，当它检测到轻点 1 次、轻点 2 次、轻点 3 次以及轻点 4 次时，它们会分别通知我们（参见图 18-4）。

我们需要为 4 个标签提供输出接口，并且还需要为每个轻点方案提供单独的方法，以便模拟在实际应用中面临的状况。我们还将引入一个擦除文本字段的方法。请打开 `BIDViewController.m` 并在顶部的类接口中进行以下更改：

```
#import "BIDViewController.h"

@interface BIDViewController ()

@property (weak, nonatomic) IBOutlet UILabel *singleLabel;
@property (weak, nonatomic) IBOutlet UILabel *doubleLabel;
@property (weak, nonatomic) IBOutlet UILabel *tripleLabel;
@property (weak, nonatomic) IBOutlet UILabel *quadrupleLabel;

@end
```

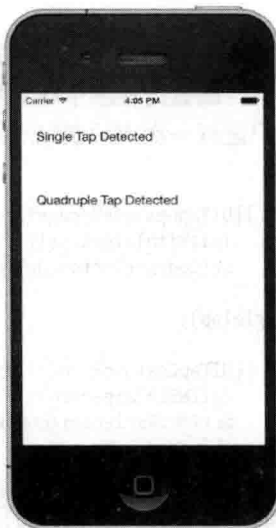


图 18-4 同时检测 4 种轻点类型的 TapTaps 应用

请保存文件，然后选择 Main.storyboard 以编辑 GUI 界面。打开该文件之后，从库中向视图添加 4 个标签，让这 4 个标签从蓝色引导线拉伸到另一侧的蓝色引导线，设对齐方式为居中，然后调整它们的格式，直到看着合适为止。我们可以使每个标签具有不同的颜色，但这不是必须的。完成后，请在按下 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 singleLabel、doubleLabel、tripleLabel 和 quadrupleLabel。现在，请一定双击每个标签并按 delete 键删除所有文本。

选中 BIDViewController.m 对代码作如下更改：

```
@implementation BIDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后（通常是从 nib 文件加载），做一些额外的设置
    UITapGestureRecognizer *singleTap =
    [[UITapGestureRecognizer alloc] initWithTarget:self
                                             action:@selector(singleTap)];
    singleTap.numberOfTapsRequired = 1;
```

```

singleTap.numberOfTouchesRequired = 1;
[self.view addGestureRecognizer:singleTap];

UITapGestureRecognizer *doubleTap =
[[UITapGestureRecognizer alloc] initWithTarget:self
                                     action:@selector(doubleTap)];
doubleTap.numberOfTapsRequired = 2;
doubleTap.numberOfTouchesRequired = 1;
[self.view addGestureRecognizer:doubleTap];
[singleTap requireGestureRecognizerToFail:doubleTap];

UITapGestureRecognizer *tripleTap =
[[UITapGestureRecognizer alloc] initWithTarget:self
                                     action:@selector(tripleTap)];
tripleTap.numberOfTapsRequired = 3;
tripleTap.numberOfTouchesRequired = 1;
[self.view addGestureRecognizer:tripleTap];
[doubleTap requireGestureRecognizerToFail:tripleTap];

UITapGestureRecognizer *quadrupleTap =
[[UITapGestureRecognizer alloc] initWithTarget:self
                                     action:@selector(quadrupleTap)];
quadrupleTap.numberOfTapsRequired = 4;
quadrupleTap.numberOfTouchesRequired = 1;
[self.view addGestureRecognizer:quadrupleTap];
[tripleTap requireGestureRecognizerToFail:quadrupleTap];
}
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // 删除可以重新创建的资源
}

- (void)singleTap
{
    self.singleLabel.text = @"Single Tap Detected";
    [self performSelector:@selector(clearLabel:)
                    withObject:self.singleLabel
                    afterDelay:1.6f];
}

- (void)doubleTap
{
    self.doubleLabel.text = @"Double Tap Detected";
    [self performSelector:@selector(clearLabel:)
                    withObject:self.doubleLabel
                    afterDelay:1.6f];
}

- (void)tripleTap
{
    self.tripleLabel.text = @"Triple Tap Detected";
    [self performSelector:@selector(clearLabel:)
                    withObject:self.tripleLabel
                    afterDelay:1.6f];
}

```

```

    }

    - (void)quadrupleTap
    {
        self.quadrupleLabel.text = @"Quadruple Tap Detected";
        [self performSelector:@selector(clearLabel:)
            withObject:self.quadrupleLabel
            afterDelay:1.6f];
    }

    - (void)clearLabel:(UILabel *)label
    {
        label.text = @"";
    }

@end

```

在这个应用里，这 4 种轻点方法的作用只是设置 4 个标签中的一个，并在 1.6 秒之后使用 `performSelector:withObject:afterDelay:` 擦除这个标签上的文本。`clearLabel:` 方法会擦除传递给它的任何标签的文本。

此代码中有趣的部分在于 `viewDidLoad` 方法中所发生的操作。开始部分很简单，我们设置一个轻点手势识别器并将它附加到视图。

```

UITapGestureRecognizer *singleTap =
[[UITapGestureRecognizer alloc] initWithTarget:self
                                     action:@selector(singleTap)];

singleTap.numberOfTapsRequired = 1;
singleTap.numberOfTouchesRequired = 1;
[self.view addGestureRecognizer:singleTap];

```

注意，这里将触发操作所需的轻点数（依次触摸相同位置的次数）和触摸数（同时触摸屏幕的手指数）设置为了 1。然后，我们设置了另一个轻点手势识别器来处理两次轻点。

```

UITapGestureRecognizer *doubleTap =
[[UITapGestureRecognizer alloc] initWithTarget:self
                                     action:@selector(doubleTap)];

doubleTap.numberOfTapsRequired = 2;
doubleTap.numberOfTouchesRequired = 1;
[self.view addGestureRecognizer:doubleTap];
[singleTap requireGestureRecognizerToFail:doubleTap];

```

这非常类似于上一个手势识别器，只不过最后一行为 `singleTap` 提供了一些附加的上下文。我们实际上是在告诉 `singleTap`：仅在其他某个手势识别器（在本例中为 `doubleTap`）断定当前的用户输入的不是其想要的手势时，`singleTap` 才应该触发自己的操作。

这是什么意思呢？有了这两个轻点手势识别器，视图中的单次轻点将立即让 `singleTap` 认为这是它寻找的手势。与此同时，`doubleTap` 将认为这看起来适合它，但它需要等待另外一次轻点。因为 `singleTap` 被设置为等待 `doubleTap` “失败”才开始运行，所以它不会立即发送自己的操作方法，而是等待 `doubleTap` 的结果。

第一次轻点之后，如果立即发生了另一次轻点，那么 `doubleTap` 会认为这完全是自己需要的手势，并触发自己的操作。在这时，`singleTap` 将认识到所发生的事情并放弃该手势。另一方面，

如果经过了特定的时间（系统规定的两次轻点之间的最大时间长度），doubleTap 将放弃，singleTap 将看到 doubleTap 失败并最终将触发自己的操作。

该方法剩余的部分为 3 次和 4 次轻点定义手势识别器，每一次配置的手势将依赖于下一个手势的失败。

```
UITapGestureRecognizer *tripleTap =
[[UITapGestureRecognizer alloc] initWithTarget:self
                                     action:@selector(tripleTap)];

tripleTap.numberOfTapsRequired = 3;
tripleTap.numberOfTouchesRequired = 1;
[self.view addGestureRecognizer:tripleTap];
[doubleTap requireGestureRecognizerToFail:tripleTap];

UITapGestureRecognizer *quadrupleTap =
[[UITapGestureRecognizer alloc] initWithTarget:self
                                     action:@selector(quadrupleTap)];

quadrupleTap.numberOfTapsRequired = 4;
quadrupleTap.numberOfTouchesRequired = 1;
[self.view addGestureRecognizer:quadrupleTap];
[tripleTap requireGestureRecognizerToFail:quadrupleTap];
```

注意，我们不需要将每个手势显式配置为依赖于每个轻点次数更多的手势的失败。这种多重依赖关系是在代码中建立的失败链的自然结果。因为 singleTap 需要 doubleTap 失败，doubleTap 需要 tripleTap 失败，而 tripleTap 需要 quadrupleTap 失败，所以引申可知 singleTap 需要所有其他手势都失败。

编译并运行此版本，当轻点 1 次、2 次、轻点 3 次以及轻点 4 次时，你应该只会看到一个标签显示。

## 18.8 检测捏合和旋转

18

另一个常见的手势是双指捏合。在很多应用（比如移动版 Safari、邮件和照片）中，人们使用它来执行放大（手指分开）或缩小（手指捏合）操作。

检测双指捏合非常简单，这得益于 UIPinchGestureRecognizer。此识别器称为连续手势识别器，因为它在双指捏合期间反复调用自己的操作方法。当发生该手势时，双指捏合手势识别器经历多个状态。我们唯一希望观察的是 UIGestureRecognizerStateBegan，它是识别器在检测到双指捏合之后首次调用操作方法时所处的状态。这时，双指捏合手势识别器的 scale 属性始终设置为 1.0，对于手势的其他状态，该数值将随着用户手指相对于起始位置的移动上升或下降。我们将使用 scale 值调整标签中文本的大小。

另一种常用的手势是双指旋转，它对应 UIRotationGestureRecognizer——一个连续的手势识别器。UIRotationGestureRecognizer 有一个 rotation 属性，在手势开始时值默认为 0.0，在用户旋转手指时，这个属性的值可以在 0.0 到  $2.0 \times \pi$  之间变化。

在 Xcode 中，我们再次使用 Single View Application 模板创建一个新项目，将此项目命名为 PinchMe。从本书项目归档文件的 18- PinchMe 文件夹中找到 yosemite-meadow.png（或者是你自

己喜欢的照片), 将其拖曳到项目中, 并记得勾选 Copy items into the destination groups folder(if needed)。接着, 展开 PinchMe 文件夹, 单击 BIDViewControoler.h, 并做如下更改:

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

@interface BIDViewController : UIViewController <UIGestureRecognizerDelegate>
```

```
@end
```

这里最大的一处改变就是让 BIDViewController 遵循 UIGestureRecognizerDelegate 协议, 以便让多个手势识别器能够同时对手势进行识别。

现在, 切换到 BIDViewController.m, 添加如下代码:

```
#import "BIDViewController.h"

@interface BIDViewController ()

@property (strong, nonatomic) UIImageView *imageView;

@end

@implementation BIDViewController {
    CGFloat scale, previousScale;
    CGFloat rotation, previousRotation;
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后 (通常是从 nib 文件加载), 做一些额外的设置
    previousScale = 1;

    UIImage *image = [UIImage imageNamed:@"yosemite-meadows.png"];
    self.imageView = [[UIImageView alloc] initWithImage:image];
    self.imageView.userInteractionEnabled = YES;
    self.imageView.center = self.view.center;
    [self.view addSubview:self.imageView];

    UIPinchGestureRecognizer *pinchGesture =
    [[UIPinchGestureRecognizer alloc]
     initWithTarget:self action:@selector(doPinch:)];
    pinchGesture.delegate = self;
    [self.imageView addGestureRecognizer:pinchGesture];

    UIRotationGestureRecognizer *rotationGesture =
    [[UIRotationGestureRecognizer alloc]
     initWithTarget:self action:@selector(doRotate:)];
    rotationGesture.delegate = self;
    [self.imageView addGestureRecognizer:rotationGesture];
}

- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer
shouldRecognizeSimultaneouslyWithGestureRecognizer:
(UIGestureRecognizer *)otherGestureRecognizer
```

```

{
    return YES;
}

- (void)transformImageView
{
    CGAffineTransform t = CGAffineTransformMakeScale(scale * previousScale,
                                                         scale * previousScale);
    t = CGAffineTransformRotate(t, rotation + previousRotation);
    self.imageView.transform = t;
}

- (void)doPinch:(UIPinchGestureRecognizer *)gesture
{
    scale = gesture.scale;
    [self transformImageView];
    if (gesture.state == UIGestureRecognizerStateChanged) {
        previousScale = scale * previousScale;
        scale = 1;
    }
}

- (void)doRotate:(UIRotationGestureRecognizer *)gesture
{
    rotation = gesture.rotation;
    [self transformImageView];
    if (gesture.state == UIGestureRecognizerStateChanged) {
        previousRotation = rotation + previousRotation;
        rotation = 0;
    }
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // 删除可以重新创建的资源
}

@end

```

这里首先定义了4个实例变量，分别表示当前缩放比例、先前缩放比例、当前旋转角度、先前旋转角度。先前的值来自于先前触发并已经结束的手势识别器；需要跟踪这些值，因为 UIPinchGestureRecognizer（用于缩放）和 UIRotationGestureRecognizer（用于旋转）开始的默认值都是缩放比例 1.0、旋转角度 0.0。

```

@implementation BIDViewController {
    CGFloat scale, previousScale;
    CGFloat rotation, previousRotation;
}

```

接下来，在 viewDidLoad 方法中，首先创建了一个会被缩放和旋转的 UIImageView。记住，一定要对图像视图启用用户交互功能，因为 UIImageView 是少数几个默认关闭用户交互功能的 UIKit 类。

```
UIImage *image = [UIImage imageNamed:@"yosemite-meadows.png"];
self.imageView = [[UIImageView alloc] initWithImage:image];
self.imageView.userInteractionEnabled = YES;
self.imageView.center = self.view.center;
[self.view addSubview:self.imageView];
```

然后，建立一个捏合手势识别器和一个旋转手势识别器，让它们分别通过 `doPinch:` 和 `doRotation:` 方法来通知我们。这两个对象都需要将 `self` 作为委托对象使用。

```
UIPinchGestureRecognizer *pinchGesture =
[[UIPinchGestureRecognizer alloc]
 initWithTarget:self action:@selector(doPinch:)];
pinchGesture.delegate = self;
[self.imageView addGestureRecognizer:pinchGesture];

UIRotationGestureRecognizer *rotationGesture =
[[UIRotationGestureRecognizer alloc]
 initWithTarget:self action:@selector(doRotate:)];
rotationGesture.delegate = self;
[self.imageView addGestureRecognizer:rotationGesture];
```

在 `gestureRecognizer:shouldRecognizeSimultaneouslyWithGestureRecognizer:` 方法中，我们始终返回 YES，以便允许捏合手势和旋转手势同时工作；否则的话，先开始的手势识别器就会屏蔽掉另一个：

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

接下来，实现一个辅助方法，用于根据从手势识别器中获得的缩放比例和旋转角度对图像视图进行变换。注意，这里将缩放比例乘以之前的缩放比例，并且对旋转角度加上了之前的旋转角度，这样就能让新手势从默认的缩放比例 1.0、旋转角度 0.0 开始的时候对先前已经结束的捏合和旋转进行调整。

```
- (void)transformImageView
{
    CGAffineTransform t = CGAffineTransformMakeScale(scale * previousScale,
                                                         scale * previousScale);
    t = CGAffineTransformRotate(t, rotation + previousRotation);
    self.imageView.transform = t;
}
```

最后，我们实现这两个操作方法，对手势识别器的输入进行处理，并且对图像视图进行变换。在 `doPinch:` 和 `doRotate:` 方法中，我们首先提取新的 `scale` 和 `rotation` 值，然后更新图像视图的变换。最后，如果手势结束（`state` 属性和 `UIGestureRecognizerStateEnded` 值相等的时候手势就结束了），我们就把当前的缩放比例和旋转角度值保存起来，并且将当前的缩放比例和旋转角度分别重置为默认的 1.0 和 0.0。

```
- (void)doPinch:(UIPinchGestureRecognizer *)gesture
{
```



```

scale = gesture.scale;
[self transformImageView];
if (gesture.state == UIGestureRecognizerStateEnded) {
    previousScale = scale * previousScale;
    scale = 1;
}
}

- (void)doRotate:(UIRotationGestureRecognizer *)gesture
{
    rotation = gesture.rotation;
    [self transformImageView];
    if (gesture.state == UIGestureRecognizerStateEnded) {
        previousRotation = rotation + previousRotation;
        rotation = 0;
    }
}

```

捏合和旋转的检测到这里就结束了。请编译和运行应用自己尝试一下。你在尝试捏合和旋转操作时，将会看到图像的变化（参见图 18-5）。如果是在模拟器上运行应用，请记住可以通过按下 option 键并在模拟器窗口中使用鼠标单击拖动来模拟捏合手势。

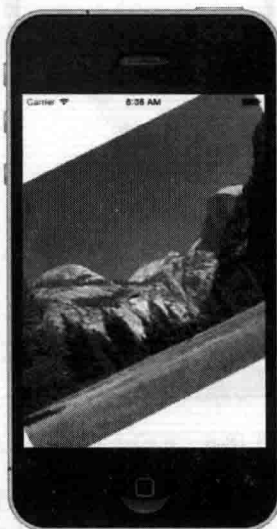


图 18-5 在 PinchMe 应用中检测捏合和旋转手势

## 18.9 自定义手势

现在，我们已经了解了如何检测最常用的 iPhone 手势。当你开始自定义手势时，才是真刀真枪的实战的开始！相信你已经知道如何使用 `UIGestureRecognizer` 的一些子类，现在是时候学习创建自己的手势了，这些手势可以轻松地关联到任何视图。

自己定义手势时你需要一些技巧。你现在已经掌握了基本的原理，因此这并不是太困难，但在定义手势的组成时，你需要灵活处事。

大多数人做出的手势并不精确。还记得之前实现轻扫手势时所使用的误差范围吗？当轻扫手势不是完全水平或垂直的时候，我们可以根据这个误差范围来判断手势是否有效。这是一个需要你向自己的手势定义中添加细微的手势误差的完美示例。如果你将手势定义得太严格，那么它将没有什么用处。如果将误差范围定的太大，就会将某些非轻扫手势判为有效的轻扫手势，这会让用户感到迷惑。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（如数字 8），那么检测该手势背后的数学计算也将非常复杂。

### 18.9.1 CheckPlease应用

在本例中，我们将定义一个形状像对勾（check-mark）的手势（参见图 18-6）。



图 18-6 对勾手势的图示

定义此对勾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，用户的手指沿直线移动一点距离。在图 17-6 中，对勾的两根分支以某个锐角相交，仅小于  $90^\circ$ 。严格的  $85^\circ$  角的手势很难有人做对，因此需要定义一个可接受的角度范围。

在 Xcode 中，请使用 Single View Application 模板创建一个新项目并将它命名为 CheckPlease。在此项目中，我们需要执行一些标准的几何分析，计算两点之间的距离和两条线之间的夹角等信息。如果回忆不起太多的几何知识，不要担心，我们提供了计算将用到的函数。

请在本书项目归档文件的 18-CheckPlease 文件夹中查找两个文件（CGPointUtils.h 和 CGPointUtils.c）并将它们都拖到项目的 CheckPlease 文件夹中。我们可以在自己的应用中自由使用这些实用的程

序函数。

在 Xcode 中右键点击 CheckPlease 文件夹，并向项目添加一个新文件，然后使用新建文件向导创建一个名为 BIDCheckMarkRecognizer 的 Objective-C 类。请在 Subclass of 控件中输入 UIGestureRecognizer，然后打开 BIDCheckMarkRecognizer.h 进行以下修改：

```
#import "BIDCheckMarkRecognizer.h"
#import "CGPointUtils.h"
#import <UIKit/UIGestureRecognizerSubclass.h>

static CGFloat const kMinimumCheckMarkAngle = 50;
static CGFloat const kMaximumCheckMarkAngle = 135;
static CGFloat const kMinimumCheckMarkLength = 10;

@implementation BIDCheckMarkRecognizer {
    CGPoint lastPreviousPoint;
    CGPoint lastCurrentPoint;
    CGFloat lineLengthSoFar;
}

@end
```

导入前面提到的 CGPointUtils.h 文件之后，再导入一个名为 UIGestureRecognizerSubclass.h 的特殊的头文件，其中包含仅应由一个子类使用的声明。这样做的一个重要目的是使手势识别器的 state 属性可写。子类将使用这一机制断言我们所观察的手势已成功完成。

然后，定义一些参数，用于确定用户的手指曲线是否与对勾定义相匹配。可以看到，这里定义了一个 50° 的最小角和 135° 的最大角。这个角度范围很宽，你可以根据具体需要缩小角度范围。我们对角度进行了大量试验，发现所采用的对勾手势涵盖很宽的范围，这是选择相对较大容错率的原因。我们做出的对勾手势非常不规则，所以相信至少有一部分用户也是如此。一位智者曾经说过：“严以律己，宽以待人。”

另外，还声明了 lastPreviousPoint、lastCurrentPoint 和 lineLengthSoFar 这 3 个实例变量。我们每次接到触摸事件通知时，都会知晓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，可以判断是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（是否实际上在绘制对勾手势）。

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。

我们还声明了一个实例变量用于保存用户手指拖动的距离。如果手指移动的距离不超过 10 个像素（在 kMinimumCheckMarkLength 中定义的值），那么角度是否落在正确的范围之内并不重要。如果没有这个距离限制，我们就可能会得到很多无效的触摸线段。

## 18.9.2 CheckPlease的触摸方法

接下来，我们添加两个方法，它们用于处理发送到手势识别器的触摸事件：

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    [super touchesBegan:touches withEvent:event];
    UITouch *touch = [touches anyObject];
    CGPoint point = [touch locationInView:self.view];
    lastPreviousPoint = point;
    lastCurrentPoint = point;
    lineLengthSoFar = 0.0;
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    [super touchesMoved:touches withEvent: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) {
        self.state = UIGestureRecognizerStateEnded;
    }
    lineLengthSoFar += distanceBetweenPoints(previousPoint, currentPoint);
    lastPreviousPoint = previousPoint;
    lastCurrentPoint = currentPoint;
}
```

可以看到，这些方法中的每一个都首先调用超类的实现，之前从未在触摸方法中这么做过。需要在 `UIGestureRecognizer` 子类中执行此操作，以便超类对于事件与我们具有同样的认识。现在，来看代码本身。

在 `touchesBegan:withEvent:` 中，确定用户当前触摸的点并将该值存储在 `lastPreviousPoint` 和 `lastCurrentPoint` 中。因为此方法在手势开始时被调用，我们知道无需关注之前的点，所以在两个地方都存储当前的点。我们还将跟踪的线条长度重设为 0。

然后，在 `touchesMoved:withEvent:` 中计算从当前触摸手势的前一个位置到其当前位置的线条的角度，两点之间的线条存储在 `lastPreviousPoint` 和 `lastCurrentPoint` 实例变量中。有了这个角度之后，我们可以检查它是否是可接受的角度，确认用户的手指在急转弯之前是否轻扫了足够长的距离。如果两个条件都满足，那么我们将标签设置为表明识别了一个对勾手势。接下来，请计算触摸的位置与其前一个位置之间的距离，将该值添加到 `lineLengthSoFar` 中，并将 `lastPreviousPoint` 和 `lastCurrentPoint` 中的值替换为来自当前触摸的两个点，以便下次可通过此方法获得它们。

现在有了一个手势识别器，是时候像其他手势一样将它与视图进行关联了。切换到 `BIDView Controller.m` 并添加以下粗体显示的代码：

```
#import "BIDViewController.h"
#import "BIDCheckMarkRecognizer.h"

@interface BIDViewController ()

@property (weak, nonatomic) IBOutlet UILabel *label;

@end
```

这里只是导入了自定义的手势识别器头文件并添加了一个标签的输出接口，用于在检测到对勾手势时通知用户。

请选中 Main.storyboard 以编辑 GUI 界面。请从库中添加一个 Label 到视图上，通过蓝色引导线将其放在视图左上角，然后调整标签大小，使其从左侧蓝色引导线拉伸到右侧蓝色引导线，并将其对齐方式设为居中。接下来按住鼠标右键，从 View Controller 图标拖到该标签，以将其连接到此 label 输出接口。我们双击该标签以删除其文本。

现在，请切换回 BIDViewController.m 并将以下代码添加到文件的 @implementation 部分：

```
@implementation BIDViewController

- (void)doCheck:(BIDCheckMarkRecognizer *)check
{
    self.label.text = @"Checkmark";
    [self performSelector:@selector(eraselabel)
        withObject:nil afterDelay:1.6];
}

- (void)eraselabel
{
    self.label.text = @"";
}
```

这提供了一个操作方法来连接识别器，进而触发我们熟悉的 eraselabel 方法。接下来，请向 viewDidLoad 方法添加以下代码，将新识别器的一个实例关联到视图上：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后（通常是从 nib 文件加载），做一些额外的设置。
    BIDCheckMarkRecognizer *check = [[BIDCheckMarkRecognizer alloc]
        initWithTarget:self
        action:@selector(doCheck:)];
    [self.view addGestureRecognizer:check];
}
```

请编译并运行应用，试用该手势。

当要为自己的应用定义新的手势时，请确保对它们进行了彻底的测试；如果可以的话，最好找其他人和你一起测试。需要确保手势对于用户来说易于使用，但也要保证它们不过于简单，以免被用户无意地触发。另外，你还需要确保它不会与应用中的其他手势发生冲突。例如，系统中的捏合手势不应该再作为自定义的其他手势使用。

## 18.10 小结

现在，相信你已经理解了 iOS 向应用通知有关触摸、轻点、手势的信息的机制。你应该还学习了如何检测最常用的 iOS 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 iOS 开发时，你会希望拥有这些技术。

当你准备好继续学习时，翻过此页，我们将告诉你如何使用 Core Location 得到你在世界上的位置。

每种 iOS 设备都可以使用 Core Location 框架确定它的物理位置。iOS 还有一个名为 Map Kit 的框架，可以用于创建实时交互的地图来显示你想要设备显示的任何位置，当然，也包括用户自己所处的位置。本章会介绍这两种框架的使用。

实际上，Core Location 可以利用 3 种技术来实现该功能：GPS、蜂窝基站 ID 定位（cell ID Location）和 WPS（Wi-Fi Positioning Service，Wi-Fi 位置服务）。GPS 是这 3 种技术中最精确的，但在第一代 iPhone、iPod touch 和只支持 Wi-Fi 的 iPad 上不可用。总之，任何至少具有 3G 数据连接功能的设备一般都还包含一个 GPS 单元。GPS 通过读取来自多个卫星的微波信号来确定当前位置。

---

**注意** 从技术上讲，苹果公司使用的 GPS 为 Assisted GPS（辅助全球卫星定位系统），也称 A-GPS。A-GPS 使用网络资源来帮助改进纯粹的 GPS 的性能，其基本原理是通信运营商部署网络服务，移动设备会自动寻找并从中收集数据，这样移动设备能够比只依靠 GPS 卫星更快地确定它的起始位置。

---

蜂窝基站 ID 定位根据设备所属范围内的蜂窝基站的位置计算得出设备的当前位置。由于每个基站可以覆盖相当广的范围，因此这种定位方式会有比较大的误差。蜂窝基站 ID 定位需要一个无线电连接，所以它只能用在 iPhone（所有款型，包括最早款）和有 3G 数据连接功能的 iPad 上。

最后一种技术 WPS 使用附近 Wi-Fi 接入点的 MAC 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。

这 3 种方法都很耗电，大家在使用 Core Location 时需要注意这点，尽量只在必要时进行定位。使用 Core Location 时，我们可以根据需要指定精度。注意，你在指定绝对最低精度级别时要谨慎，以避免不必要的电力消耗。

Core Location 所依赖的技术对于应用来说是隐藏的。我们不需要指定 Core Location 是使用 GPS、蜂窝基站 ID 定位还是 WPS，只是指定精度级别，然后它将自动从可用的技术中选择可以更好地满足你请求的那种技术。

## 19.1 位置管理器

Core Location API 实际上非常易于使用。这里将使用的主类是 `CLLocationManager`，通常称为位置管理器（location manager）。为了与 Core Location 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];
```

这样就创建了位置管理器的一个实例，但它不会真正开始轮询我们的位置。此处还必须创建一个遵循 `CLLocationManagerDelegate` 协议的对象，并将其作为位置管理器的委托。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，甚至多达几秒钟。

### 19.1.1 设置精度

设置委托之后，你还需要设置所需的精度。前面讲过，我们建议避免指定任何大于绝对需要的精度。如果你编写的应用只需要知道手机当前位置所在的州或国家，则不需要指定较高级别的精度。记住，Core Location 的精度越高，电量消耗就越大。另外，也不一定总能获得所需级别的精度。

下面是设置委托和请求特定精度级别的示例：

```
locationManager.delegate = self;  
locationManager.desiredAccuracy = kCLLocationAccuracyBest;
```

精度通过设定 `CLLocationAccuracy` 的值进行指定，这里值的类型为 `double`。该值的单位为米（m），因此如果你指定 `desiredAccuracy` 的值为 10，就表示要求 Core Location 在尝试确定当前位置时尽量达到 10 米的精度。如之前一样指定 `kCLLocationAccuracyBest`，或者指定 `kCLLocationAccuracyBestForNavigation`（它也使用其他传感器数据）表示要求 Core Location 使用当前可用的具有最高精度的方法。除了 `kCLLocationAccuracyBestForNavigation`，我们还可以使用 `kCLLocationAccuracyNearestTenMeters`、`kCLLocationAccuracyHundredMeters`、`kCLLocationAccuracyKilometer` 和 `kCLLocationAccuracyThreeKilometers`。

### 19.1.2 设置距离筛选器

默认情况下，位置管理器会把检测到的位置更改通知给委托。指定距离筛选器意味着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大小时通知你。设置距离筛选器可以减少应用所执行的轮询数量。

距离筛选器也是以米为单位进行设置的。若指定距离筛选器为 1000，则表示直到 iPhone 已经偏移以前报告的位置至少 1000 米之后，才会通知位置管理器的委托。下面是一个示例：

```
locationManager.distanceFilter = 1000;
```

如果你曾经希望将位置管理器恢复为没有筛选器的默认设置，可以使用常量 `kCLDistanceFilterNone`，如下所示：



```
locationManager.distanceFilter = kCLLocationDistanceFilterNone;
```

就像在指定特定精确度时一样，这里应该注意避免过于频繁地获取更新，否则会浪费电量。基于用户的位置计算用户速度的加速计应用可能需要尽可能快地获取更新，但显示附近快餐店的应用则完全可以让频率更新得慢一些。

### 19.1.3 启动位置管理器

当准备好开始轮询位置时，通知位置管理器启动，然后位置管理器开始启动，然后在定位到当前位置时调用委托方法。在它停止之前，只要感知到任何超过当前距离筛选器的更改，就会继续调用委托方法。

下面是启动位置管理器的方法：

```
[locationManager startUpdatingLocation];
```

### 19.1.4 合理使用位置管理器

如果只需要确定当前位置而不需要持续轮询位置，那么当位置管理器获取应用所需的信息之后，就应该让位置委托停止位置管理器。如果需要持续轮询，那么则需要确保只要有可能就尽量停止轮询。请记住，只要从位置管理器获取更新，就一定会消耗电量。

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];
```

## 19.2 位置管理器委托

位置管理器委托必须遵循 `CLLocationManagerDelegate` 协议，该协议定义了一些方法，这些方法都是可选的。位置管理器在确定了当前位置或者检测到位置更改时调用其中某个方法。当位置管理器遇到错误时将调用另一个方法。我们会在应用中同时实现这两个方法。

### 19.2.1 获取位置更新

当位置管理器希望将当前位置通知给委托时，它将调用 `locationManager:didUpdateLocations` 方法。该方法接受两个参数。

- ❑ 第一个参数是调用该方法的位置管理器。
- ❑ 第二个参数是一个 `CLLocation` 对象数组，用于描述设备的当前位置，可能还有之前的几个位置。如果在一段比较短的时间内发生了多次位置更新，这几次位置更新有可能会被一次性全部上报（通过调用一次这个方法）。无论何时，数组的最后一项都表示当前位置。

### 19.2.2 使用 `CLLocation` 获取纬度和经度

位置信息是位置管理器使用 `CLLocation` 类的实例进行传递的。该类具有应用可能感兴趣的 5

个属性：

- ❑ `coordinate`（地理坐标）
- ❑ `horizontalAccuracy`（水平精度）
- ❑ `altitude`（海拔高度）
- ❑ `verticalAccuracy`（垂直精度）
- ❑ `timestamp`（时间戳）

纬度和经度存储在一个名为 `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` 对象还可以显示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` 为圆心的圆的半径。`horizontalAccuracy` 的值越大，Core Location 就越确定不了准确的位置；半径越小，位置精度就越高。

我们来看 `horizontalAccuracy` 在地图应用中的图形表示（参见图 19-1）。当检测到你所处位置时，地图中显示的圆将 `horizontalAccuracy` 作为半径。位置管理器认为你位于该圆的中心；如果不是这样，但几乎可以肯定你位于圆之内的某个位置。`horizontalAccuracy` 为负值时，则表明由于某些原因，而导致无法信任 `coordinate` 的值。



图 19-1 地图应用使用 Core Location 来确定你的当前位置。外面的圆是水平精度的可视化表示

`CLLocation` 对象还具有一个名为 `altitude` 的属性，该属性用于描述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 在描述海拔高度时的精确程度。海拔高度值可能与 `verticalAccuracy` 的值相差很多米，并且如果 `verticalAccuracy` 为负值，那么 Core Location 就是在告诉你它无法确定有效的海拔高度。

`CLLocation` 对象具有一个时间戳，它描述位置管理器确定位置的时间。

除了这些属性，`CLLocation` 还有一个非常有用的实例方法，该方法允许你确定两个 `CLLocation` 对象之间的距离。该方法是 `distanceFromLocation:`，其工作方式如下：

```
CLLocationDistance distance = [fromLocation distanceFromLocation:toLocation];
```

这行代码将返回两个 `CLLocation` 对象（`fromLocation` 和 `toLocation` 之间的距离）。返回的 `distance` 值将是 大圆计算的结果，该计算忽略了 `altitude` 属性，并且在假设这两个点处于同一海平面的情况下计算该距离。对于大多数情况，大圆计算已经能够满足需求，但是如果在计算距离时需要考虑海拔高度，那么你必须自己编写代码来进行该操作。

---

**注意** 如果不确定大圆距离的含义，你可能需要回想一下地理课上学到的大圆路线的概念。地球表面上任何两点之间的最短距离都可沿着一条围绕整个地球的路线（“大圆”）计算得出。显然，地图上的赤道和经线就是大圆。然而，地球表面上的任意两点之间都存在这样的一个圆弧。`CLLocation` 负责计算两点之间沿这样一条路线的距离，并且考虑了地球的弯曲度。如果不考虑该弯曲度，我们最终将得到连接两点的直线，这没多大用处，因为该直线一定会从地球内部“穿过”！

---

### 19.2.3 错误通知

如果 Core Location 无法确定你的当前位置，它会调用一个名为 `locationManager:didFailWithError:` 的委托方法。最有可能的错误原因是用户拒绝访问。位置管理器的使用必须由用户进行授权，因此应用在第一次确定位置时会在屏幕上弹出一个提示，询问用户是否确定让当前应用访问其位置（参见图 19-2）。

如果用户轻点 **Don't Allow** 按钮，那么位置管理器会使用包含错误代码 `kCLErrorDenied` 的 `locationManager:didFailWithError:` 通知你的委托。位置管理器支持的另一常见错误代码为 `kCLErrorLocationUnknown`，它表示 Core Location 无法确定位置，但它将不断尝试。`kCLErrorLocationUnknown` 错误表示问题可能是临时的，而 `kCLErrorDenied` 或其他错误通常表示当前会话的其余时间内，应用都将无法访问 Core Location。

---

**注意** 在模拟器中工作时，模拟器窗口外部会出现一个提示框，询问是否可以使用你的当前位置。这种情况下，你的位置会模拟位于库比蒂诺的苹果公司总部中。

---

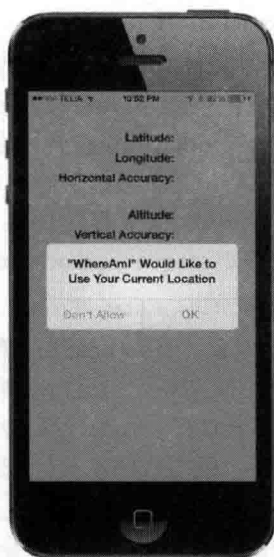


图 19-2 位置管理器的访问必须经过用户授权

### 19.3 开始构建 Core Location

让我们构建一个小型应用来检测 iPhone 的当前位置以及该程序运行期间所移动的总距离。最终的应用如图 19-3 所示。

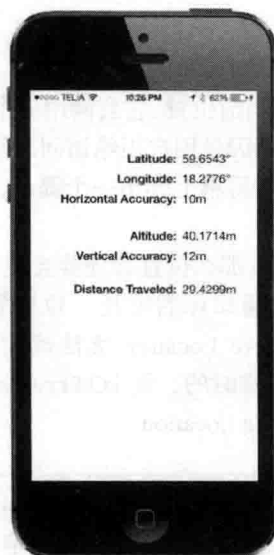


图 19-3 运行中的 WhereAmI 应用

在 Xcode 中, 请使用 Single View Application 模板新建一个项目, 并将该项目命名为 WhereAmI, 然后将 Devices 设为 iPhone。单击 BIDViewController.h 并进行以下更改:

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

@interface BIDViewController :
    UIViewController <CLLocationManagerDelegate>

@end
```

需要注意的第一件事情是这里已经包含了 Core Location 头文件。Core Location 不是 UIKit 或 Foundation 的一部分, 因此我们需要手动包含头文件。接下来, 使该类遵循 CLLocationManagerDelegate 协议, 以便可以从位置管理器接收位置信息。

现在选择 BIDViewController.m 并在文件顶部附近的类扩展中添加这些属性变量声明:

```
#import "BIDViewController.h"

@interface BIDViewController ()

@property (strong, nonatomic) CLLocationManager *locationManager;
@property (strong, nonatomic) CLLocation *previousPoint;
@property (assign, nonatomic) CLLocationDistance totalMovementDistance;
@property (weak, nonatomic) IBOutlet UILabel *latitudeLabel;
@property (weak, nonatomic) IBOutlet UILabel *longitudeLabel;
@property (weak, nonatomic) IBOutlet UILabel *horizontalAccuracyLabel;
@property (weak, nonatomic) IBOutlet UILabel *altitudeLabel;
@property (weak, nonatomic) IBOutlet UILabel *verticalAccuracyLabel;
@property (weak, nonatomic) IBOutlet UILabel *distanceTraveledLabel;

@end
```

之后, 我们声明一个 CLLocationManager 指针, 使用它来存放指向我们将要创建的 Core Location 实例的指针。我们还声明了一个指向 CLLocation 的指针, 将其设置为最后一次更新时从位置管理器接收的位置。这样每当用户的移动足够触发更新的一段距离, 我们就能够将最后移动的距离添加到总移动距离中。

其他属性都是输出接口, 用于更新用户界面上的标签。

选中 Main.storyboard 开始创建 GUI 界面。将图 19-3 作为向导, 我们从库中拖出 12 个标签到 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, 然后分别将这 6 个粗体标签的值设置为 Latitude:、Longitude:、Horizontal Accuracy:、Altitude:、Vertical Accuracy: 和 Distance Traveled:。因为 Horizontal Accuracy: 标签最长, 所以我们选择先放它, 然后按住 option 拖出复制出另外 5 个粗体标签。右边的 6 个标签应该采用左对齐, 并且紧挨着放在每个左侧粗体标签的旁边。

右侧的每个标签应该关联到之前到头文件中定义的适当输出接口。一旦将上述 6 个标签关联到输出接口之后, 依次双击每个标签, 删除其包含的文本, 然后保存修改。

接下来, 请选中 BIDViewController.m, 并在 viewDidLoad 中插入以下代码来配置位置管理器:

```

- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后 (通常是从 nib 文件加载), 做一些额外的设置。
    self.locationManager = [[CLLocationManager alloc] init];
    self.locationManager.delegate = self;
    self.locationManager.desiredAccuracy = kCLLocationAccuracyBest;
    [self.locationManager startUpdatingLocation];
}

```

在 `viewDidLoad` 方法中, 我们分配并初始化一个 `CLLocationManager` 实例, 并将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 `Location Manager` 实例开始提供位置更新。

现在在 `@implementation` 部分末尾插入以下新的委托方法, 用来处理从位置管理器所接受的信息:

```

#pragma mark - CLLocationManagerDelegate Methods
- (void)locationManager:(CLLocationManager *)manager
    didUpdateLocations:(NSArray *)locations {
    CLLocation *newLocation = [locations lastObject];
    NSString *latitudeString = [NSString stringWithFormat:@"%g\u00B0",
                                newLocation.coordinate.latitude];
    self.latitudeLabel.text = latitudeString;

    NSString *longitudeString = [NSString stringWithFormat:@"%g\u00B0",
                                newLocation.coordinate.longitude];
    self.longitudeLabel.text = longitudeString;

    NSString *horizontalAccuracyString = [NSString stringWithFormat:@"%gm",
                                newLocation.horizontalAccuracy];
    self.horizontalAccuracyLabel.text = horizontalAccuracyString;

    NSString *altitudeString = [NSString stringWithFormat:@"%gm",
                                newLocation.altitude];
    self.altitudeLabel.text = altitudeString;

    NSString *verticalAccuracyString = [NSString stringWithFormat:@"%gm",
                                newLocation.verticalAccuracy];
    self.verticalAccuracyLabel.text = verticalAccuracyString;

    if (newLocation.verticalAccuracy < 0 ||
        newLocation.horizontalAccuracy < 0) {
        // 无效的精度
        return;
    }

    if (newLocation.horizontalAccuracy > 100 ||
        newLocation.verticalAccuracy > 50) {
        // 这里不使用过大的精度值
        return;
    }

    if (self.previousPoint == nil) {
        self.totalMovementDistance = 0;
    }
}

```

```

    } else {
        self.totalMovementDistance += [newLocation
            distanceFromLocation:self.previousPoint];
    }
    self.previousPoint = newLocation;

    NSString *distanceString = [NSString stringWithFormat:@"%gm",
        self.totalMovementDistance];
    self.distanceTraveledLabel.text = distanceString;
}

- (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];
}

```

## 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，而且我们知道，如果实现委托方法 `locationManager:didUpdateLocations:`，就可以在发生位置更新时获得通知。因此，让我们看一看该方法的实现。

在委托方法中，首先从 `locations` 参数获得一个 `CLLocation` 对象（示例代码中命名为 `newLocation`），然后根据这个对象的值更新前 5 个标签：

```

NSString *latitudeString = [NSString stringWithFormat:@"%g\u00B0",
    newLocation.coordinate.latitude];
self.latitudeLabel.text = latitudeString;

NSString *longitudeString = [NSString stringWithFormat:@"%g\u00B0",
    newLocation.coordinate.longitude];
self.longitudeLabel.text = longitudeString;

NSString *horizontalAccuracyString = [NSString stringWithFormat:@"%gm",
    newLocation.horizontalAccuracy];
self.horizontalAccuracyLabel.text = horizontalAccuracyString;

NSString *altitudeString = [NSString stringWithFormat:@"%gm",
    newLocation.altitude];
self.altitudeLabel.text = altitudeString;

NSString *verticalAccuracyString = [NSString stringWithFormat:@"%gm",
    newLocation.verticalAccuracy];
self.verticalAccuracyLabel.text = verticalAccuracyString;

```

**注意** 经度和纬度都以格式字符串显示，包含\u00B0，看起来十分神秘。这是角度符号(°)的 Unicode 表示形式的十六进制值。将不是 ASCII 字符的任何其他东西直接放入源代码文件绝不是个好主意，但在字符串中包含十六进制值是可以的，这里即是如此。

接下来，检查位置管理器所提供值的精确度。负的精度值表明位置是无效的，过大的精度值表明位置管理器并不确定位置是否准确。用米表示的精度值表示位置管理器提供的一个圆半径，这意味着真实的位置可能会在这个圆内部的任何一点。在代码中检查这些精度值是否能够接受，如果不是，就直接返回，而不要继续再处理垃圾数据。

```
if (newLocation.verticalAccuracy < 0 ||
    newLocation.horizontalAccuracy < 0) {
    // 无效的精度
    return;
}

if (newLocation.horizontalAccuracy > 100 ||
    newLocation.verticalAccuracy > 50) {
    // 这里不使用过大的精度值
    return;
}
```

然后，检查 `previousPoint` 是否为 `nil`。如果是，则该更新是来自位置管理器的第一个有效更新，我们将 `distanceFromStart` 归零；如果不是，就将最后的位置与上一个位置之间的距离添加到总距离中。无论是哪一种情况，我们都要更新 `previousPoint` 为当前位置。

```
if (self.previousPoint == nil) {
    self.totalMovementDistance = 0;
} else {
    self.totalMovementDistance += [newLocation
                                   distanceFromLocation:self.previousPoint];
}
self.previousPoint = newLocation;
```

之后，我们将从初始位置开始移动的总距离更新到最后那个标签上。这个应用程序运行的时候，如果用户移动的距离足够远，从而使得位置管理器能够检测到位置变动，那么 `Distance Traveled` 字段就会不断被更新为从应用程序启动时的位置开始移动的距离。

```
NSString *distanceString = [NSString stringWithFormat:@"%gm",
                             self.totalMovementDistance];
self.distanceTraveledLabel.text = distanceString;
```

就是这么简单。Core Location 是非常简单易用的。

编译并运行该应用，然后尝试一下。如果有条件，你也应在自己的 iPhone 或 iPad 上运行它，可以尝试在开车时运行该程序，看看数据值的变化。哦，对了，最好让别人来开车！

## 19.4 将移动路线展现在地图上

目前为止，这个应用还是相当简洁的。但是，如果能够将移动路线展现在地图上，不是更好



吗？幸运的是，iOS 提供的 Map Kit 框架可以帮助实现这个目的。Map Kit 与苹果的地图应用使用同样的后端服务，也就是说，Map Kit 提供的地图服务是非常健壮的，而且会持续改进。它包含一个主要的用于显示地图的视图类，可以对用户手势作出非常好的响应，就如当前的地图应用一样。我们可以在这个视图上为地图上的任何位置添加想要显示的消息（默认使用大头针来表示，可以点击以显示更多详细信息）。接下来我们将扩展 WhereAmI 应用，把用户的初始位置和当前位置显示到地图上。

首先，我们在 Xcode 中选中的 BIDViewController.h，然后在文件顶部导入地图框架的头文件：

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

接下来切换到 BIDViewController.m 并在类扩展中的其他属性下面再添加一个属性：

```
@property (weak, nonatomic) IBOutlet MKMapView *mapView;
```

然后，选中 Main.storyboard 开始编辑视图。我们希望保留所有的标签，不过需要它们看起来像是地图视图的弹出式界面。一个比较好的方式是使用一个半透明的方形 UIView。所以，我们选中所有的标签，然后从菜单中选择 Editor>Embed In>View。

确保选中新的视图，然后使用属性检查器禁用 User Interaction Enabled 复选框（这样一来在此视图上触摸任何东西的事件都会被忽略，作为替代，将所有的触摸事件都传递到地图视图上），并且将它的背景颜色设为半透明。这时，我们将该视图拖动到父视图的底部。现在我们需要创建一些约束，这样此视图将始终位于底部并且无论屏幕尺寸如何改变高度都将固定。点击编辑区域底部的 Pin 按钮调出约束创建面板。在面板的上半部分点中小正方形左边、右边的底下的实线。在中间部分勾选 Height 复选框。如果设置完后的界面和图 19-4 的相同，请点击位于底部的 Add 4 Constraints 按钮：

现在，在对象库中找到一个 MKMapView 对象，将其拖曳到主视图上。调整它的尺寸，使其占满整个视图，然后选择 Editor>Arrange>Send to Back，使其显示在已有的标签背后。按住鼠标右键，从 View Controller 图标拖曳鼠标到地图视图上，然后选择 mapView 输出接口。再次使用 Pin 面板为地图视图创建约束，这次要在先将小正方形四周的所有四条实线都点中后再点击 Add 4 Constraints 按钮。

这些准备工作完成之后，就可以编写代码让地图为我们服务了。与控制器打交道之前，我们需要建立一些模型类用于表示起始点。MKMapView 是作为 MVC 体系结构中的视图（View）部分构建的，使用其他类来表示地图上的标记点更为合适。可以将模型对象传递给地图视图，通过 Map Kit 框架中定义的协议来查询它们的坐标、标题等。

请在 Xcode 中新建一个 Objective-C 类，继承于 NSObject 类，将其命名为 BIDPlace。我们选中 BIDPlace.h 并且进行如下修改。需要导入 Map Kit 头文件、指定遵循的协议、添加一些属性变量：

```
#import <Foundation/Foundation.h>
#import <MapKit/MapKit.h>
@interface BIDPlace : NSObject <MKAnnotation>
...
@property (copy, nonatomic) NSString *title;
```

```

@property (copy, nonatomic) NSString *subtitle;
@property (assign, nonatomic) CLLocationCoordinate2D coordinate;

@end

```

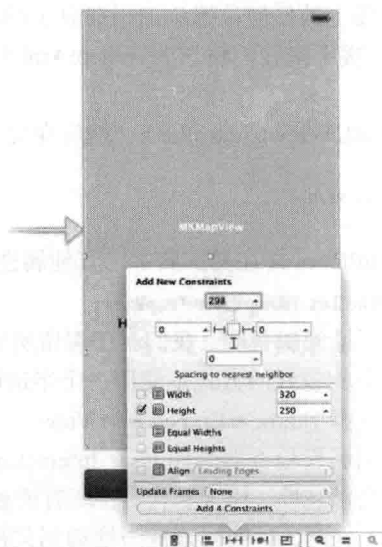


图 19-4 现在所有的标签都位于新的方形视图中，并且移动到了屏幕底部。设置约束使视图保持形状

这个类有点儿简单，仅仅用于保存这些属性变量，甚至都不需要修改.m 文件。在实际的开发中，你可能有实用的模型类需要作为标示显示在地图上，这时可以使用 MKAnnotation 框架方便地为类添加这种能力，而不需要打乱现有的类层级。

我们选中 BIDViewController.m，首先导入新类的头文件：

```
#import "BIDPlace.h"
```

然后，找到 viewDidLoad 方法，在方法末尾添加如下代码：

```
self.mapView.showsUserLocation = YES;
```

可能你已经想到了，这里所做的就是在用户移动时自动在地图上绘制用户的位置，而不需要每次都手动去做。

现在，再次访问 locationManager:didUpdateLocations: 方法。这里面已经有一些代码了，用于得到第一个有效位置数据并建立起始点。我们将分配一个新的 BIDPlace 类实例，设置其属性变量，指定一个位置，以及添加当位置标记显示出来时所需要显示的标题和子标题，最后将这个对象传给地图视图。

这里还创建了一个 MKCoordinateRegion 对象，它是 Map Kit 中一个特殊的结构体，用于告诉视图需要显示地图的哪一部分，MKCoordinateRegion 使用新的位置坐标以及一对以米为单位的距离 (100, 100) 来指定需要显示的地图部分的宽度和高度。我们将其传递给地图视图，告诉视图

对这个变化使用动画效果。如下粗体所示的代码就是用来执行这些工作的：

```
if (self.previousPoint == nil) {
    self.totalMovementDistance = 0;

    BIDPlace *start = [[BIDPlace alloc] init];
    start.coordinate = newLocation.coordinate;
    start.title = @"Start Point";
    start.subtitle = @"This is where we started!";

    [self.mapView addAnnotation:start];
    MKCoordinateRegion region;
    region = MKCoordinateRegionMakeWithDistance(newLocation.coordinate,
                                                100, 100);
    [self.mapView setRegion:region animated:YES];
} else {
    self.totalMovementDistance += [newLocation
                                   distanceFromLocation:self.previousPoint];
}
self.previousPoint = newLocation;
```

现在，地图视图知道有一个标示（比如一个可见的位置标记）要显示给用户，但是如何显示它呢？在复杂的应用中，地图视图请求它的委托来确定应该为每一个标示显示哪种视图。但是在这个例子中，我们并没有将自身设置为委托，因为对于这种简单的应用场景来说根本不需要这样做。与 UITableView 需要数据源提供单元进行显示不同，MKMapView 使用不同的策略：如果委托没有提供标示视图，就使用默认的视图（在地图上显示一个红色的大头针，点击后可以显示更多信息）。多么简单！

现在构建并运行应用，你将看到地图视图加载出来。当得到有效的位数据之后，我们可以看到地图向右滚动，在起始位置处放置了一枚大头针，而你的当前位置会使用蓝色的圆点来标记，如图 19-5 所示。效果还不错，毕竟只用了这么少的代码。



图 19-5 红色的大头针标记了起始位置，蓝色的圆点显示了当前位置

## 19.5 小结

现在，你已经了解了 Core Location 的方方面面，还尝试了 Map Kit 的基本操作。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了起来，使我们可以非常方便地为应用添加位置相关的功能和地图功能，以便了解用户的当前位置、在其移动时进行通知，并且在地图上标记他们的位置（或者其他位置）。

准备好了吗？请阅读下一章，了解如何使用 iPhone 的内置加速计。

内置加速计是 iPhone、iPad 和 iPod touch 最酷的特性之一，iOS 可以通过这个小设备知道用户握持设备的方式，以及用户是否移动了设备。iOS 使用加速计处理自动旋转，并且许多游戏都将它作为游戏操控机制。它还可以用于检测摇动和其他突发的运动。此功能在 iPhone 4 上得到了进一步的扩展，它是第一款内置陀螺仪的 iPhone，可让开发者确定设备的方向与每条坐标轴之间的夹角。最新生产的 iPad 和 iPod touch 上都内置了陀螺仪和加速计。本章将介绍如何在应用中使用 Core Motion 框架来访问这些值。

## 20.1 加速计物理特性

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。iOS 设备内的加速计是一个三轴加速计，也就是说它能够检测到三维空间中的运动或重力。也就是说，加速计不但可以指示用户握持设备的方式（如自动旋转功能），而且还可以在设备被放在桌子上时指示其是正面朝下还是朝上。

加速计可以测量重力，因此加速计返回值为 1.0 时，表示在特定方向上感知到的重力是 1 g，下面是几个例子。

- 如果是静止握持设备而没有任何运动，那么地球引力对其施加的力大约为 1 g。
- 如果是纵向竖直地握持，那么设备会检测并报告在其 y 轴上的力大约为 1 g。
- 如果是以一定角度握持，那么 1 g 的力会分布到不同的轴上，这取决于握持的方式。在以 45°角握持时，1 g 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 1 g，那么我们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 1 g 的值。如果摇动、坠落或投掷设备，那么加速计便会在一个或多个轴上检测到很大的力。请不要为了测试这一理论而坠落或投掷自己的 iOS 设备，除非你正打算找个借口换最新的机型。

图 20-1 展示了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，这与第 16 章讨论的 Quartz 2D 的坐标系相反。如果加速计将 Quartz 2D 作为控制机制，那么必须要转换 y 坐标轴。使用 Sprite Kit 时（使用加速计控制动画时通常会用到），则不需要转换。

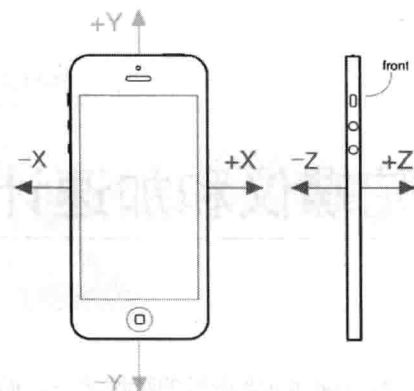


图 20-1 三维方向上 iPhone 加速计的轴，左边 iPhone 的正视图展示的是  $x$  轴和  $y$  轴，右边的侧视图展示的是  $z$  轴

## 20.2 陀螺仪旋转特性

如前所述，现在所有 iOS 设备都有一个陀螺仪传感器，可用于读取描述设备围绕其轴旋转的值。

如果此传感器与加速计之间的区别看起来不那么明显，我们可以想象 iPhone 平放在桌面上的情况。如果在保持手机平放的同时旋转它，加速计的值不会更改。这是因为让手机移动的力（在这种情况下只有重力直接施加在  $z$  轴上）没有改变。（实际的情况比这更难理解，你的手碰到手机时肯定会触发细微的加速计操作。）但是，在相同的运动过程中，设备的旋转值将改变，具体来讲就是  $z$  轴的旋转值将改变。顺时针旋转设备将生成负值，逆时针旋转它将生成正值。停止旋转后， $z$  轴旋转值将恢复为 0。

无需注册绝对的旋转值，在设备旋转值发生变化时陀螺仪会告诉你。本章的第一个示例将介绍这是如何实现的。

## 20.3 Core Motion 和动作管理器

加速计和陀螺仪的值是通过 Core Motion 框架访问的。此框架提供了 `CMMotionManager` 类（当然还有其他内容），该类提供的所有数据都是用来描述用户如何移动设备的。应用程序创建一个 `CMMotionManager` 实例，然后通过以下某种模式使用它：

- 它可以在动作发生时执行一些代码；
- 它可以时刻监视一个持续更新的结构，使你随时能够访问最新的值。

后一种方法是游戏和其他高度交互性应用程序的理想选择，这类应用程序需要能够在游戏循环的每一关轮询设备的最新状态。我们将介绍如何实现这两种方法。

注意，`CMMotionManager` 类实际上不是一个单例，但应用程序应该将它视为单例。我们应该仅为每个应用创建一个 `CMMotionManager` 实例，使用普通的 `alloc` 和 `init` 方法。所以，如果需要

从应用中的多个位置访问动作管理器，可能需要在应用程序委托中创建它并提供从这里访问它的权限。

除了 `CMMotionManager` 类，Core Motion 还提供了其他一些类，比如 `CMAccelerometerData` 和 `CMGyroData`，它们是一些简单容器，用于让应用程序访问动作数据。我们在遇到这些类时再一一介绍它们。

### 20.3.1 基于事件的动作

我们提到动作管理器可以在这样一种模式下运行：它在动作数据每次更改时执行一些代码。其他的大部分 Cocoa Touch 类提供此类功能的方式是在时机来临时允许你连接到一个委托来获取消息，但 Core Motion 的实现方式稍有不同。

`CMMotionManager` 没有使用委托方法，而是让你在发生移动时传递给一个代码块来执行的。本书中已经多次使用代码块，但现在你将会看到此技术的另一项应用。

使用 Xcode 创建一个新的 Single View Application 项目，将其命名为 `MotionMonitor`。这是一个非常简单的应用，它读取加速计数据和陀螺仪数据（如果可用）并在屏幕上显示。

---

**注意** 本章中的应用程序不适用于模拟器，因为模拟器没有加速计。太遗憾了！

---

现在我们选中 `BIDViewController.h` 文件，进行以下更改：

```
#import "BIDViewController.h"

@interface BIDViewController ()

@property (weak, nonatomic) IBOutlet UILabel *accelerometerLabel;
@property (weak, nonatomic) IBOutlet UILabel *gyroscopeLabel;

@end
```

这段代码提供了一个访问动作管理器的指针，以及两个将显示信息的标签的输出接口。这里没有太多需要解释的，保存更改即可。

接下来在界面构建器中打开 `Main.storyboard` 文件。然后从库中将一个标签拖到视图中。调整标签，使其与左右两侧的蓝色引导线对齐，高度调整为整个视图的一半，然后将标签的顶部与顶部的蓝色引导线对齐。

现在打开属性检查器并将 `Lines` 字段从 1 更改为 0。`Lines` 属性用于指定标签中可以出现多少行文本，提供一个硬性上限。如果将它设置为 0，则不会应用限制，那么标签可以包含任意行数的文本。接下来按住 `option` 键并拖动标签以创建一个副本，将该副本与视图下半部分中的蓝色引导线对齐。现在按住鼠标右键并从 `View Controller` 图标拖到每个标签，将 `accelerometerLabel` 关联到上方的标签，将 `gyroscopeLabel` 关联到下方的标签。

最后，双击这两个标签，删除原来的文本。

这个简单的 GUI 就制作完成了，我们保存工作并准备编写一些代码。

接下来，选中 `BIDViewController.m`。现在到有趣的部分了。我们添加以下内容：

```

#import "BIDViewController.h"
#import <CoreMotion/CoreMotion.h>

@interface BIDViewController ()

@property (weak, nonatomic) IBOutlet UILabel *accelerometerLabel;
@property (weak, nonatomic) IBOutlet UILabel *gyroscopeLabel;

@property (strong, nonatomic) CMMotionManager *motionManager;
@property (strong, nonatomic) NSOperationQueue *queue;

@end

@implementation BIDViewController

- (NSUInteger)supportedInterfaceOrientations
{
    return UIInterfaceOrientationMaskPortrait;
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后 (通常是从 nib 文件加载), 做一些额外的设置
    self.motionManager = [[CMMotionManager alloc] init];
    self.queue = [[NSOperationQueue alloc] init];
    if (self.motionManager.accelerometerAvailable) {
        self.motionManager.accelerometerUpdateInterval = 1.0 / 10.0;
        [self.motionManager startAccelerometerUpdatesToQueue:self.queue
                                     withHandler:
        ^(CMAccelerometerData *accelerometerData, NSError *error) {
            NSString *labelText;
            if (error) {
                [self.motionManager stopAccelerometerUpdates];
                labelText = [NSString stringWithFormat:
                    @"Accelerometer encountered error: %@", error];
            } else {
                labelText = [NSString stringWithFormat:
                    @"Accelerometer\n---\n"
                    "x: %.2f\ny: %.2f\nz: %.2f",
                    accelerometerData.acceleration.x,
                    accelerometerData.acceleration.y,
                    accelerometerData.acceleration.z];
            }
            dispatch_async(dispatch_get_main_queue(), ^{
                self.accelerometerLabel.text = labelText;
            });
        });
    } else {
        self.accelerometerLabel.text = @"This device has no accelerometer.";
    }
    if (self.motionManager.gyroAvailable) {
        self.motionManager.gyroUpdateInterval = 1.0 / 10.0;
        [self.motionManager startGyroUpdatesToQueue:self.queue withHandler:
        ^(CMGyroData *gyroData, NSError *error) {
            NSString *labelText;

```



```

        if (error) {
            [self.motionManager stopGyroUpdates];
            labelText = [NSString stringWithFormat:
                @"Gyroscope encountered error: %@", error];
        } else {
            labelText = [NSString stringWithFormat:
                @"Gyroscope\n---\n"
                "x: %.2f\ny: %.2f\nz: %.2f",
                gyroData.rotationRate.x,
                gyroData.rotationRate.y,
                gyroData.rotationRate.z];
        }
        dispatch_async(dispatch_get_main_queue(), ^{
            self.gyroscopeLabel.text = labelText;
        });
    }];
} else {
    self.gyroscopeLabel.text = @"This device has no gyroscope";
}
}

-(void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // 删除可以重新创建的资源
}

```

@end

首先，导入了 Core Motion 框架的头文件，并且在类扩展部分添加了额外两个属性。

```
@interface BIDViewController ()
```

```

@property (weak, nonatomic) IBOutlet UILabel *accelerometerLabel;
@property (weak, nonatomic) IBOutlet UILabel *gyroscopeLabel;

@property (strong, nonatomic) CMMotionManager *motionManager;
@property (strong, nonatomic) NSOperationQueue *queue;

```

@end

然后，覆盖 supportedInterfaceOrientations 方法，以免在试用动作传感器时屏幕发生旋转。

```

- (NSUInteger)supportedInterfaceOrientations
{
    return UIInterfaceOrientationMaskPortrait;
}

```

然后，在 viewDidLoad 方法中添加触发传感器所需的全部代码，告诉传感器每隔 0.1 秒向我们报告一次，然后更新屏幕显示的内容。

得益于代码块的强大功能，代码非常简单和紧凑。我们不用把功能分散在各个委托方法中，你可以在代码块中定义行为，并在配置代码块的方法中就能看到它！我们逐步讲解一下这一过程。首先从这里开始：

```

self.motionManager = [[CMMotionManager alloc] init];
self.queue = [[NSOperationQueue alloc] init];

```

这段代码首先创建一个 `CMMotionManager` 实例，我们将使用它监测动作事件。然后代码会创建一个操作队列，也就是一些需要完成的工作的容器（回想一下第 15 章）。

**警告** 动作管理器需要有一个队列，以便在每次发生事件时在其中放入一些要完成的工作，这些工作由你将提供给它的代码块指定。可能你会想把系统的默认队列用在这里，但 `CMMotionManager` 的文档明确警告不要这么做！原因在于，默认队列最终可能会被这些事件填满，并因而无法处理其他重要的系统事件。

下一步是配置加速计。我们首先确保设备确实拥有加速计。目前为止，所有 iOS 设备都有加速计，但仍然需要检查一下，因为未来的设备可能没有加速计。然后，设置更新的时间间隔（以秒为单位）。在这里，我们要求每 0.1 秒更新一次。注意，该设置无法保证将在准确地每隔 0.1 秒收到更新。实际上，该设置只是一种限制，可以指定允许动作管理器提供更新的最佳速率。现实中，它的更新频率可能低于该值。

```
if (self.motionManager.accelerometerAvailable) {
    self.motionManager.accelerometerUpdateInterval = 1.0 / 10.0;
```

接下来，告诉动作管理器开始报告加速计更新。我们传入队列和代码块；队列中放置着每次发生更新时要完成的工作，代码块定义这些工作。记住，一个代码块始终以 `^` 符号开始，后跟一个包含在圆括号中的参数列表，列表中包含在执行代码块时要填充到其中的参数（在本例中为加速计数据，可能还有一个提醒我们出现故障的错误），最后为花括号部分，其中包含要执行的代码。

```
[self.motionManager startAccelerometerUpdatesToQueue:self.queue
                    withHandler:
^ (CMAccelerometerData *accelerometerData, NSError *error) {
```

后面的就是代码块的内容。它基于当前的加速计值创建一个字符串，或者在出现问题时生成一条错误消息。然后，它将该字符串值推入 `accelerometerLabel` 中。这里无法直接这么做，因为像 `UILabel` 这样的 `UIKit` 类通常仅在从主线程访问时才能很好地运行。因为此代码将从 `NSOperationQueue` 内部执行，所以我们不知道将执行的特定线程。因此，我们要在设置标签的文本属性之前，使用 `dispatch_async()` 方法将控制权交给主线程。

注意，加速计值通过传给它的 `accelerometerData` 的 `acceleration` 属性进行访问。`acceleration` 属性的类型为 `CMAcceleration`，它是一个包含 3 个 `float` 值的简单 `struct`。`accelerometerData` 本身是 `CMAccelerometerData` 类的一个实例，该类实际上是一个 `CMAcceleration` 包装器！如果你认为传递 3 个 `float` 参数是对类和类型的一种浪费，这么想的人可不止你一个。不管怎样，它的用法就是像下面这样：

```
NSString *labelText;
if (error) {
    [self.motionManager stopAccelerometerUpdates];
    labelText = [NSString stringWithFormat:
        @"Accelerometer encountered error: %@", error];
} else {
    labelText = [NSString stringWithFormat:
        @"Accelerometer\n---\n"]
```

```

        "x: %.2f\ny: %.2f\nz: %.2f",
        accelerometerData.acceleration.x,
        accelerometerData.acceleration.y,
        accelerometerData.acceleration.z];
    }
    dispatch_async(dispatch_get_main_queue(), ^{
        self.accelerometerLabel.text = labelText;
    });
}];

```

然后，我们完成代码块，完成方括号中的方法调用，首先在其中传递该代码块。最后，我们提供一条完全不同的代码路径，因为要应对设备没有加速计的情况：

```

} else {
    self.accelerometerLabel.text = @"This device has no accelerometer.";
}

```

之前已经提过，目前为止所有 iOS 设备都拥有加速计，但仍要对此进行监测，谁知道未来的设备会怎样呢？

你一定已经注意到，陀螺仪与加速计的代码在结构上是相同的，不同之处只是在于调用哪些方法和如何访问报告的值。它们非常类似，所以这里没有必要再介绍一遍。

现在，构建应用并在你的 iOS 设备上尝试使用它（参见图 20-2）。在用不同方式倾斜设备的过程中，我们可以看到加速度值是如何调整来适应每个新位置的，只要握住设备不动，加速计值也会保持不变。

如果在安装了陀螺仪的设备上运行该应用，我们也将看到这些值是如何变化的。只要设备保持静止，无论它处于哪个方向，陀螺仪值都将接近 0。当旋转设备时，陀螺仪值会发生变化，其变化取决于它是如何围绕各个轴旋转的。当停止移动设备时，各个值将恢复为 0。



图 20-2 在 iPhone 设备上运行的 MotionMonitor。遗憾的是，如果在模拟器中运行此应用，你只会得到两条错误消息

## 20.3.2 主动动作访问

前面介绍了如何通过传递将在动作发生时调用的 `CMMotionManager` 代码块来访问动作数据。对于一般的 Cocoa 应用, 这种事件驱动的动作处理就足够了, 但是有时无法很好地满足应用程序的特定需要。例如, 交互式游戏通常拥有一个始终在运行的循环, 该循环处理用户输入、更新游戏状态和重新绘制屏幕。在这种情况下, 事件驱动的方法就不足以满足需求了, 因为我们需要实现一个对象来等待动作事件, 记住每个传感器报告的最新位置, 在必要时向主游戏循环报告数据。

幸运的是, `CMMotionManager` 有一个内置的解决方案。无需传入代码块, 我们只需告诉它使用 `startAccelerometerUpdates` 和 `startGyroUpdates` 方法激活传感器, 这样的话, 我们就可以在任何时候直接从动作管理器读取相应值!

更改一下 `MotionMonitor` 应用来使用此方法, 这样就可以看到它是如何工作的了。复制 `MotionMonitor` 项目文件夹。关闭当前的 Xcode 项目, 并打开新的副本并找到 `BIDViewController.m` 文件。首先, 移除 `queue` 属性, 再添加一个新属性 (指向一个 `NSTimer`, 用于触发屏幕更新):

```
#import "BIDViewController.h"
#import <CoreMotion/CoreMotion.h>

@interface BIDViewController ()

@property (weak, nonatomic) IBOutlet UILabel *accelerometerLabel;
@property (weak, nonatomic) IBOutlet UILabel *gyroscopelabel;

@property (strong, nonatomic) CMMotionManager *motionManager;
@property (strong, nonatomic) NSOperationQueue *queue;
@property (strong, nonatomic) NSTimer *updateTimer;

@end
```

接下来, 删除已有的整个 `viewDidLoad` 方法, 将它替换为下面这个更简单的版本, 它设置动作管理器并为缺乏传感器的设备提供信息标签:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后 (通常是从 nib 文件加载), 做一些额外的设置
    self.motionManager = [[CMMotionManager alloc] init];
    if (self.motionManager.accelerometerAvailable) {
        self.motionManager.accelerometerUpdateInterval = 1.0 / 10.0;
    } else {
        self.accelerometerLabel.text = @"This device has no accelerometer.";
    }
    if (self.motionManager.gyroAvailable) {
        self.motionManager.gyroUpdateInterval = 1.0/10.0;
    } else {
```

```

        self.gyroscopeLabel.text = @"This device has no gyroscope.";
    }
}

```

我们需要让定时器和动作管理器在很短的某个时机被激活，也就是视图真正被显示的时候。这样就可以将主“游戏循环”的使用率降到最低。为此，我们可以按如下方式实现 `viewWillAppear:` 和 `viewDidDisappear:` 方法。代码如下所示：

```

- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    [self.motionManager startAccelerometerUpdates];
    [self.motionManager startGyroUpdates];
    self.updateTimer = [NSTimer
        scheduledTimerWithTimeInterval:1.0 / 10.0
        target:self
        selector:@selector(updateDisplay)
        userInfo:nil
        repeats:YES];
}

- (void)viewDidDisappear:(BOOL)animated {
    [super viewDidDisappear:animated];
    [self.motionManager stopAccelerometerUpdates];
    [self.motionManager stopGyroUpdates];
    [self.updateTimer invalidate];
    self.updateTimer = nil;
}

```

`viewWillAppear:` 中的代码创建了一个新的计时器，并计划每隔 0.1 秒触发一次，也就是调用 `updateDisplay` 方法（还未创建它）。我们将此方法添加到 `viewDidDisappear:` 下方：

```

- (void)updateDisplay {
    if (self.motionManager.accelerometerAvailable) {
        CMAccelerometerData *accelerometerData =
            self.motionManager.accelerometerData;
        self.accelerometerLabel.text = [NSString stringWithFormat:
            @"Accelerometer\n---\n"
            "x: %.2f\ny: %.2f\nz: %.2f",
            accelerometerData.acceleration.x,
            accelerometerData.acceleration.y,
            accelerometerData.acceleration.z];
    }
    if (self.motionManager.gyroAvailable) {
        CMGyroData *gyroData = self.motionManager.gyroData;
        self.gyroscopeLabel.text = [NSString stringWithFormat:
            @"Gyroscope\n---\n"
            "x: %.2f\ny: %.2f\nz: %.2f",
            gyroData.rotationRate.x,
            gyroData.rotationRate.y,
            gyroData.rotationRate.z];
    }
}

```

在设备上构建并运行应用，你应该会看到它的行为与第一个版本完全一样。现在，你已经看到了两种访问动作数据的方法，实践中请选择最适合具体应用程序的方法。

### 20.3.3 加速计结果

前面已经提到, iPhone 的加速计沿 3 个轴检测加速度, 它使用 `CMAcceleration` 结构体提供此信息。每个 `CMAcceleration` 拥有一个 `x`、`y` 和 `z` 字段, 每个字段保存一个浮点值。值 0 表示加速计在特定轴上未检测到移动, 正值或负值表示一个方向上的力。例如, 负 `y` 值表示感知到向下的运动, 这可能表示手机在纵向上被直立地握着。正 `y` 值表示在相反方向上存在一个力, 这可能表示手机被倒拿着或在朝下运动。

记住图 20-1 所示的内容, 我们再来看一下图 20-3 中的一些加速计结果。注意, 在现实生活中我们几乎不会获得这么理想化的值, 因为加速计非常灵敏, 能够感知非常细微的运动, 通常在所有 (3 个) 轴上存在微弱的力。这是真实世界中的物理现象, 不属于中学所学的物理学知识。

通常, 加速计在第三方应用程序中被用作游戏的控制器。本章后面将创建的一个程序使用加速计进行输入, 不过我们先看一下另一个常见的加速计用途: 检测摇动。

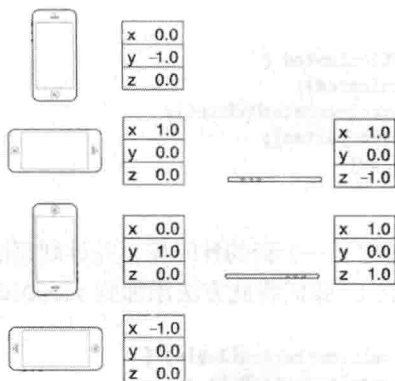


图 20-3 不同设备方向上理想化的加速度值

## 20.4 检测摇动

像手势一样, 摇动可用作应用程序的一种输入形式。例如, 绘图程序 `GLPaint` (一个苹果的 iOS 示例代码项目) 允许用户摇动 iOS 设备来擦除图像, 类似于 `Etch A Sketch`。

检测摇动相对来说是件小事。它只是检查一个轴上比特定阈值大的绝对值。在正常使用中, 3 个轴上的注册值常常高达 1.3 g, 但获取比该值更大的值通常需要特意施加力量。加速计不太可能注册比 2.3 g 更大的值 (至少我还从未遇到过), 所以不需要设置比该值更大的值。

要检测摇动, 我们可以通过检查比 1.5 大的绝对值来检测细微摇动, 通过检查比 2.0 更大的值来检测强烈摇动, 比如:

```
CMAccelerometerData *accelerometerData =
    self.motionManager.accelerometerData;
```

```

if (fabsf(accelerometerData.acceleration.x) > 2.0
    || fabsf(accelerometerData.acceleration.y) > 2.0
    || fabsf(accelerometerData.acceleration.z) > 2.0) {
    // 在这里做一些处理
}

```

前面的方法检测任何轴上力大于 2 g 的任何运动。

我们可以实现更复杂的摇动检测，要求用户来回摇动一定次数以注册为摇动，比如：

```

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 = [[NSDate alloc] init];
}

CMAccelerometerData *accelerometerData =
    self.motionManager.accelerometerData;
if (fabsf(accelerometerData.acceleration.x) > 2.0
    || fabsf(accelerometerData.acceleration.y) > 2.0
    || fabsf(accelerometerData.acceleration.z) > 2.0) {
    shakeCount++;
    if (shakeCount > 4) {
        // 做一些处理
        shakeCount = 0;
        shakeStart = [[NSDate alloc] init];
    }
}

```

此方法跟踪加速计报告大于 2.0 的值的次数，如果加速计在 1.5 秒内报告了 4 次，该运动就会被注册为摇动。

### 20.4.1 内嵌的摇动检测

实际上还有另一种检测摇动的方法，这种方法被结合到了响应者链中。还记得在第 18 章中我们是如何实现 `touchesBegan:withEvent:` 这样的方法来检测触摸的吗？iOS 提供了 3 个类似的响应程序方法来检测动作：

- ❑ 当动作开始时，`motionBegan:withEvent:` 方法会被发送到第一响应者，然后通过响应者链，如第 18 章所述；
- ❑ 当动作结束时，`motionEnded:withEvent:` 方法会被发送到第一响应者；
- ❑ 如果在摇动期间电话振铃或发生了其他某个干扰动作，`motionCancelled:withEvent:` 消息会被发送到第一响应者。

这意味着无需直接使用 `CMMotionManager` 即可检测摇动；只需要重写视图或视图控制器中相应的动作感知方法，在用户摇动手机时这些方法将被自动调用。除非明确需要对摇动手势进行更

多控制，否则我们应该使用内嵌的动作检测方法，而不是前面介绍的手动方法，之前介绍手动方法的基础知识是为满足你需要进行更多控制的需要。

现在你对如何检测摇动有了基本的理解，接下来我们将击碎你的手机。

### 20.4.2 摇动与击碎

好吧，我们并不是真的要将手机摔碎，只是要编写一个应用程序，它在检测到摇动之后会使手机在视觉和听觉上呈现破碎的效果。

启动此应用程序后，程序会显示一张图片，它看起来像是 iPhone 的主屏幕（参见图 20-4）。以足够大的力气摇动手机时，我们就可以听到手机发出一种不好声音，任何人都不会想听到这样的声音从消费性电子设备中发出来。此外，屏幕看起来如图 20-5 所示。我们为什么要做这种坏事呢？不要担心，只需触摸屏幕你就可以将 iPhone 重置为初始状态。



图20-4 ShakeAndBreak应用程序看起来平淡无奇

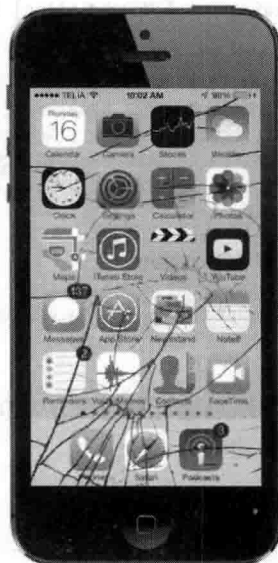


图20-5 但如果摇动太猛烈的话……哦，不

我们在 Xcode 中使用 Single View Application 模板新建一个项目，将新建项目命名为 ShakeAndBreak。在项目归档文件的 20-ShakeAndBreak 文件夹中，我们已经为此应用程序提供了两张图和一个声音文件。我们将 home.png、homebroken.png 和 glass.wav 文件拖到项目中。

现在，开始创建视图控制器。我们需要创建一个指向图像视图的输出接口，用于在后面改变显示的图像。请单击 BIDViewController.m，并在类扩展中添加以下属性声明：

```
#import "BIDViewController.h"
```

```
@interface BIDViewController ()
```



```
@property (weak, nonatomic) IBOutlet UIImageView *imageView;
```

```
@end
```

保存文件。现在单击 Main.storyboard，在界面构建器中编辑此文件。然后，请单击选中 View Controller，在属性检查器里将 Simulated Metrics 下面的 Status Bar 弹出框从 Inferred 更改为 None。接下来，我们从库中拖出一个图像视图放到视图上的布局区域。图像视图能够自动调整大小以占满整个窗口，所以将它放置到窗口中即可。

我们按下鼠标右键并从 View Controller 图标拖到图像视图，并选择 imageView 输出接口，然后保存分镜文件。

接下来，请回到 BIDViewController.m 文件。

我们要为两张需要显示的图像添加一些额外的属性变量，以检测当前显示的是否为已破碎的图像。我们还要添加一个音频播放器对象来播放玻璃破碎的音效。将以下代码添加到文件顶部：

```
#import "BIDViewController.h"  
#import <AVFoundation/AVFoundation.h>
```

```
@interface BIDViewController ()
```

```
@property (weak, nonatomic) IBOutlet UIImageView *imageView;  
@property (strong, nonatomic) UIImage *fixed;  
@property (strong, nonatomic) UIImage *broken;  
@property (assign, nonatomic) BOOL brokenScreenShowing;  
@property (strong, nonatomic) AVAudioPlayer *crashPlayer;
```

```
@end
```

然后，在 viewDidLoad 中添加如下实现代码：

```
@implementation BIDViewController  
  
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
    // 视图加载完成之后（通常是从 nib 文件加载），做一些额外的设置  
  
    NSURL *url = [[NSBundle mainBundle] URLForResource:@"glass"  
        withExtension:@"wav"];  
  
    NSError *error = nil;  
    self.crashPlayer = [[AVAudioPlayer alloc] initWithContentsOfURL:url  
        error:&error];  
  
    if (!self.crashPlayer) {  
        NSLog(@"Audio Error! %@", error.localizedDescription);  
    }  
  
    self.fixed = [UIImage imageNamed:@"home.png"];  
    self.broken = [UIImage imageNamed:@"homebroken.png"];  
  
    self.imageView.image = self.fixed;  
  
}
```

在这里我们创建了一个 NSURL 对象以指向声音文件，并初始化了一个 AVAudioPlayer 实例。这是一个简易的可以播放音乐的类。之后的检测用以确保音频播放器是正确设置的，我们加载了两张需要使用的图像并把第一张显示出。

接下来添加以下新方法：

```
- (void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event;
{
    if (!self.brokenScreenShowing && motion == UIEventSubtypeMotionShake) {
        self.imageView.image = self.broken;
        [self.crashPlayer play];
        self.brokenScreenShowing = YES;
    }
}
```

这个方法会在摇动发生时被调用。在检测后确定破碎的图像还没有显示，而且接收的确实是摇动事件，这个方法会显示破碎的图像并播放碎裂的声音。

对于最后一个方法，你已经非常熟悉了。这一方法会在触摸屏幕这一事件发生时被调用。在此方法中，我们只需要将图像设置回未破坏的屏幕，并将 brokenScreenShowing 置为 NO。

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    self.imageView.image = self.fixed;
    self.brokenScreenShowing = NO;
}
```

我们编译并运行应用程序，并对它进行摇动测试。无法在 iOS 设备上运行此应用程序的人也可以尝试。模拟器无法模拟加速计硬件，但它有一个可以模拟摇动事件的菜单项，因此模拟器上也能运行。

现在可以体验应用程序了。完成之后，我们再看看如何将加速计用做游戏或其他程序中的控制器。

## 20.5 将加速计用做方向控制器

游戏开发者一般不会采用按钮来控制游戏角色或对象的移动，而是使用加速计来实现这个功能。例如，在赛车游戏中，像方向盘一样转动 iOS 设备可以让汽车转弯，而向前倾斜可以加速，向后倾斜可以制动。

如何将加速计用做控制器？这在很大程度上取决于特定的游戏操作方法。在最简单的情况下，我们可以获取一个轴上的值，将它乘以一个数，并将结果添加到受控对象的坐标上，但在较复杂的游戏（它们更逼真地模拟了物理特性）需要根据加速计返回的值调整受控对象的速度。

将加速计作为控制器使用有一个棘手问题：委托方法无法保证按指定的间隔回调。如果告诉动作管理器每秒读取加速计 60 次，那么唯一能够确定的是它不会在 1 秒内更新超过 60 次。我们无法保证每秒更新均等时间间隔的 60 次。所以如果制作基于加速器输入的动画，我们必须跟踪更新之间的时间间隔，将它作为一个考虑因素来确定对象的移动速度。

## 20.5.1 滚弹珠程序

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极简示例。此处使用 Quartz 2D 来处理动画。

**注意** 在处理游戏或其他需要平滑动画的程序时，我们通常使用 Sprite Kit 或 OpenGL ES。此处的应用程序使用 Quartz 2D，这是因为它比较简单，并且可以减少与使用加速计无关的代码。

在此应用程序中，如果倾斜 iPhone，弹珠就会来回滚动，就像是在桌面上一样（参见图 20-6）。iPhone 向左倾斜，小球就会向左滚动；倾斜得越厉害，小球就滚动得越快。若再反方向倾斜，则小球的滚动会慢下来并开始向另一个方向滚动。

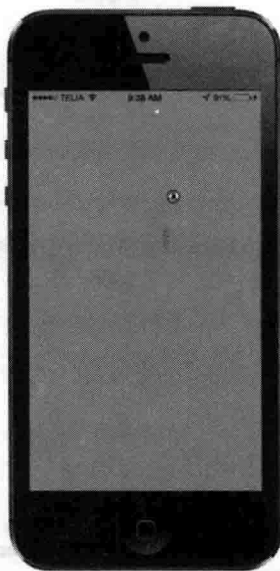


图 20-6 使用滚弹珠应用程序在屏幕上滚动弹珠

在 Xcode 中，请使用 Single View Application 模板新建一个项目，将它命名为 Ball。在项目归档文件的 19 - Ball 文件夹中，我们可以找到一个名为 ball.png 的图像，请将它拖到新项目中。

接下来，在项目导航面板中选中 Ball 项目，然后点击 Targets 下 Ball 的 General 分页。在 Deployment Info 区域，取消除 Portrait 之外的所有 Device Orientations 复选框的选中状态（如图 20-7 所示）。这样就禁用了默认的界面方向变化方式，我们希望在移动设备让滚动弹珠时界面方向不会发生改变。

然后，单击 Ball 文件夹，并从 File 菜单中选择 New > New File...。从 Cocoa Touch 目录中选择 Objective-C class，单击 Next，将新类命名为 BIDBallView，然后从 Subclass of 弹出菜单中选择 UIView，点击 Next，然后单击 Create 保存类文件。稍后我们会回来编辑这个类。

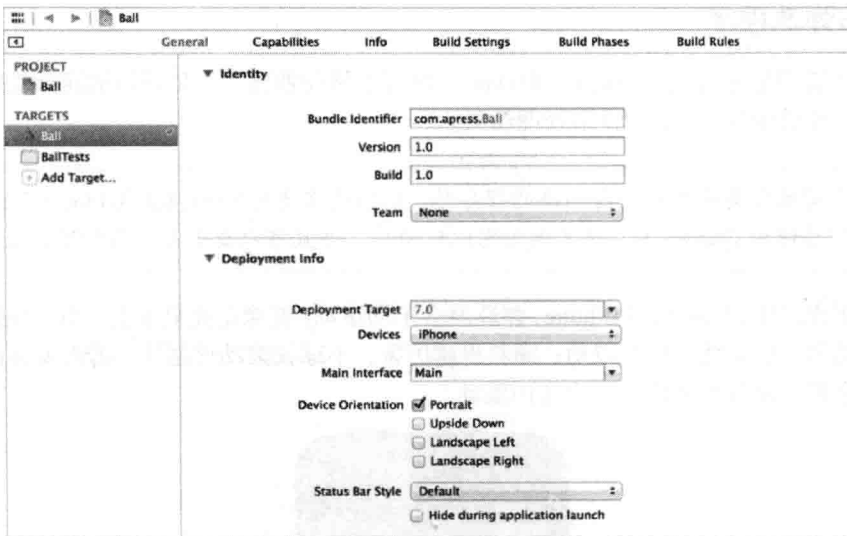


图 20-7 在目标的 Summary 标签中禁用除 Portrait 之外的所有界面方向

请选中 Main.storyboard，在界面构建器中编辑文件。请单击 View 图标，并使用身份检查器将视图的 class 由 UIView 改为 BIDBallView。然后，切换到属性检查器，将视图的 Background 更改为 Black Color，最后保存分镜文件。

接下来，编辑 BIDViewController.m，在文件顶部添加以下代码：

```
#import "BIDViewController.h"
#import "BIDBallView.h"
#import <CoreMotion/CoreMotion.h>

#define kUpdateInterval    (1.0f / 60.0f)

@interface BIDViewController ()
@property (strong, nonatomic) CMMotionManager *motionManager;
@property (strong, nonatomic) NSOperationQueue *queue;
@end
@implementation BIDViewController
{
    .
    .
    .
}
```

我们用以下代码填充 viewDidLoad 方法：

```
-(void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后（通常是从 nib 文件加载），做一些额外的设置
    self.motionManager = [[CMMotionManager alloc] init];
    self.queue = [[NSOperationQueue alloc] init];
    self.motionManager.accelerometerUpdateInterval = kUpdateInterval;
    [self.motionManager startAccelerometerUpdatesToQueue:self.queue

```

```

        withHandler:
^((CMAccelerometerData *accelerometerData, NSError *error) {
    [(id)self.view setAcceleration:accelerometerData.acceleration];
    [self.view performSelectorOnMainThread:@selector(update)
        withObject:nil
        waitUntilDone:NO];
});
}

```

注意 你可能会在这里看到一个错误提示说 BIDBallView 不完整。这里的大部分工作都在 BIDBallView 类中进行，稍后我们就来处理它。

这里的 `viewDidLoad` 方法非常类似于本章其他地方执行的某项操作。主要的区别在于，我们声明了每秒 60 次的较高更新频率。需要报告加速计更新时，应告诉动作管理器执行一个代码块，在这个代码块中将加速对象直接传入到视图中，然后调用一个名为 `update` 的方法，该方法基于加速度和自上一次更新以来经过的时间来更新弹珠在视图中的位置。由于代码块可在任何线程上执行，而 UIKit 对象（包括 UIView）中的方法仅能从主线程安全地使用，所以我们再次强制在主线程中调用 `update` 方法。

## 20.5.2 实现BIDBallView类

选中 BIDBallView.h。你需要在这里导入 Core Motion 头文件并添加属性变量，控制器将传递这些加速度值：

```

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

@interface BIDBallView : UIView
@property (assign, nonatomic) CMAcceleration acceleration;

```

@end

切换到 BIDBallView.m 并在顶部附近的类扩展中进行以下更改：

```

#import "BIDBallView.h"

@interface BIDBallView ()

@property (strong, nonatomic) UIImage *image;
@property (assign, nonatomic) CGPoint currentPoint;
@property (assign, nonatomic) CGPoint previousPoint;
@property (assign, nonatomic) CGFloat ballXVelocity;
@property (assign, nonatomic) CGFloat ballYVelocity;

```

@end

现在来看一下这些属性以及它们各自的作用。第一个实例变量是 UIImage，它指向屏幕上滚动的弹珠：

```
UIImage *image;
```

然后,跟踪两个 `CGPoint` 变量。`currentPoint` 变量用于保持小球当前的位置。同样,我们也要跟踪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;
```

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。下一节将计算滚动速度。我们从加速计获得加速度值并跟踪这些变量在两个轴上的速度。

```
CGFloat ballXVelocity;
CGFloat ballYVelocity;
```

现在我们来编写在屏幕上绘制并移动小球的代码。首先,在 `BIDBallView.m` 的 `@implementation` 部分顶部添加如下方法:

```
@implementation BIDBallView
```

```
- (void)commonInit
```

```
{
```

```
    self.image = [UIImage imageNamed:@"ball.png"];
```

```
    self.currentPoint = CGPointMake((self.bounds.size.width / 2.0f) +
```

```
                                   (self.image.size.width / 2.0f),
```

```
                                   (self.bounds.size.height / 2.0f) +
```

```
                                   (self.image.size.height / 2.0f));
```

```
}
```

```
- (id)initWithCoder:(NSCoder *)coder
```

```
{
```

```
    self = [super initWithCoder:coder];
```

```
    if (self) {
```

```
        [self commonInit];
```

```
    }
```

```
    return self;
```

```
}
```

```
- (id)initWithFrame:(CGRect)frame
```

```
{
```

```
    self = [super initWithFrame:frame];
```

```
    if (self) {
```

```
        [self commonInit];
```

```
    }
```

```
    return self;
```

```
}
```

```
.
```

```
.
```

```
.
```

`initWithCoder:`方法和 `initWithFrame:`方法都会调用 `commonInit` 方法。我们在分镜文件中创建的视图会通过 `initWithCoder:`方法进行初始化。在两个初始化器方法中都调用 `commonInit` 方法是为了保证视图类既可以从代码创建也可以从 nib 文件创建。对于任何可能会被重用的视图类(比如这个好玩的弹珠滚动视图)来说,这样做是非常好的。

现在，我们对之前注释掉的 `drawRect:` 方法取消注释，按如下方式实现该方法：

```
- (void)drawRect:(CGRect)rect
{
    // 绘图代码
    [self.image drawAtPoint:self.currentPoint];
}
```

然后，将下列新方法添加到类的末尾：

```
.
.
#pragma mark -

- (void)setCurrentPoint:(CGPoint)newPoint
{
    self.previousPoint = self.currentPoint;
    _currentPoint = newPoint;

    if (self.currentPoint.x < 0) {
        _currentPoint.x = 0;
        self.ballXVelocity = 0;
    }

    if (self.currentPoint.y < 0){
        _currentPoint.y = 0;
        self.ballYVelocity = 0;
    }

    if (self.currentPoint.x > self.bounds.size.width - self.image.size.width) {
        _currentPoint.x = self.bounds.size.width - self.image.size.width;
        self.ballXVelocity = 0;
    }

    if (self.currentPoint.y >
        self.bounds.size.height - self.image.size.height) {
        _currentPoint.y = self.bounds.size.height - self.image.size.height;
        self.ballYVelocity = 0;
    }

    CGRect currentRect =
    CGRectMake(self.currentPoint.x, self.currentPoint.y,
               self.currentPoint.x + self.image.size.width,
               self.currentPoint.y + self.image.size.height);
    CGRect previousRect =
    CGRectMake(self.previousPoint.x, self.previousPoint.y,
               self.previousPoint.x + self.image.size.width,
               self.currentPoint.y + self.image.size.height);
    [self setNeedsDisplayInRect:CGRectUnion(currentRect, previousRect)];
}

- (void)update
{
    static NSDate *lastUpdateTime = nil;

    if (lastUpdateTime != nil) {
        NSTimeInterval secondsSinceLastDraw =
```

```

[[NSDate date] timeIntervalSinceDate:lastUpdateTime];

self.ballYVelocity = self.ballYVelocity -
    (self.acceleration.y * secondsSinceLastDraw);
self.ballXVelocity = self.ballXVelocity +
    (self.acceleration.x * secondsSinceLastDraw);

CGFloat xAccel = secondsSinceLastDraw * self.ballXVelocity * 500;
CGFloat yAccel = secondsSinceLastDraw * self.ballYVelocity * 500;

self.currentPoint = CGPointMake(self.currentPoint.x + xAccel,
                                self.currentPoint.y + yAccel);
}
// 用当前时间更新最后时间
lastUpdateTime = [[NSDate alloc] init];
}

@end

```

### 20.5.3 计算弹珠运动

`drawRect:`方法非常简单。我们只是把在 `commonInit:`方法中加载的图像绘制在 `currentPoint` 存储的位置处。`currentPoint` 存取器是一个标准的存取器方法。`setCurrentPoint:`方法是自定义的设置方法。

在 `setCurrentPoint:`中, 我们首先将旧的 `currentPoint` 值存储在 `previousPoint` 中, 并将新值赋给 `currentPoint`。

```

self.previousPoint = self.currentPoint;
self.currentPoint = newPoint;

```

下面来做边界检查。如果弹珠的  $x$  或  $y$  位置小于 0, 或分别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 (self.currentPoint.x < 0) {
    _currentPoint.x = 0;
    self.ballXVelocity = 0;
}
if (self.currentPoint.y < 0) {
    _currentPoint.y = 0;
    self.ballYVelocity = 0;
}
if (self.currentPoint.x > self.bounds.size.width - self.image.size.width) {
    _currentPoint.x = self.bounds.size.width - self.image.size.width;
    self.ballXVelocity = 0;
}
if (self.currentPoint.y >
    self.bounds.size.height - self.image.size.height) {
    _currentPoint.y = self.bounds.size.height - self.image.size.height;
    self.ballYVelocity = 0;
}

```



**提示** 希望弹珠能够更加自然地从墙面弹起，而不是仅仅停止在墙上？这相当简单。只需要将 `setCurrentPoint:` 中的 `self.ballXVelocity = 0;` 更改为 `self.ballXVelocity = - (self.ballXVelocity / 2.0);`，并将 `self.ballYVelocity = 0;` 更改为 `self.ballYVelocity = - (self.ballYVelocity / 2.0);`。完成以上更改之后，弹珠的速度将减半并且它将以相反的方向运动，而不是停止（速度为零）。现在，弹珠会在反方向上拥有一半的速度。

之后，我们根据图像的大小计算两个 `CGRect`。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。这两个矩形可以确保在绘制新弹珠的同时擦除旧的弹珠。

```
CGRect currentRect =
CGRectMake(self.currentPoint.x, self.currentPoint.y,
            self.currentPoint.x + self.image.size.width,
            self.currentPoint.y + self.image.size.height);
CGRect previousRect =
CGRectMake(self.previousPoint.x, self.previousPoint.y,
            self.previousPoint.x + self.image.size.width,
            self.currentPoint.y + self.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(currentRect, previousRect)];
```

本类中的最后一个实质性方法是 `update`，它用于计算小球的正确位置。此方法在为视图提供了新的加速对象之后，会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，此变量用于跟踪距离上次调用 `update` 方法的时间。第一次执行此方法，当 `lastUpdateTime` 是 `nil` 时，我们不需要做任何事，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少了一帧。

```
static NSDate *lastUpdateTime = nil;
```

```
if (lastUpdateTime != nil) {
```

每次执行此方法时，我们都计算出距离上次调用此方法的时间。通过 `[NSDate date]` 返回的 `NSDate` 实例代表当前时间。

通过请求从 `lastUpdateDate` 开始的时间间隔，获取了表示当前时间和 `lastUpdateTime` 之间相隔的秒数：

```
NSTimeInterval secondsSinceLastDraw =
[[NSDate date] timeIntervalSinceDate:lastUpdateTime];
```

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` 相乘是因为加速度是与时间一致的。以同样的角度倾斜手机总可以得到相同的加速度。

```
self.ballYVelocity = self.ballYVelocity -  
    (self.acceleration.y * secondsSinceLastDraw);  
self.ballXVelocity = self.ballXVelocity +  
    (self.acceleration.x * secondsSinceLastDraw);
```

之后，我们根据速度计算出上次调用此方法之后发生的像素变化。这里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，就像弹珠粘上了糖浆一样。

```
CGFloat xAccel = secondsSinceLastDraw * self.ballXVelocity * 500;  
CGFloat yAccel = secondsSinceLastDraw * self.ballYVelocity * 500;
```

知道了发生的像素变化之后，我们将当前位置与计算出的加速度相加，并赋值给 `currentPoint`，这样即可创建一个新点。通过使用 `self.currentPoint`，我们便可以使用前面编写的存取器方法，而不必将数值直接赋给实例变量。

```
self.currentPoint = CGPointMake(self.currentPoint.x + xAccel,  
    self.currentPoint.y + yAccel);
```

至此，计算已经完成了。剩余的工作是将 `lastUpdateTime` 更新为当前时间：

```
lastUpdateTime = [[NSDate alloc] init];
```

在编译之前，我们使用之前提到的技术添加 Core Motion 框架。添加之后，我们编译并运行这个应用。

如果一切顺利，应用程序就会开始运行。现在你应该可以通过倾斜手机控制弹珠的滚动了。弹珠到达屏幕的边缘时会停止。如果向另一面倾斜手机，弹珠应该会向另一个方向滚动。成功！

## 20.6 小结

我们已经在本章享受到了物理和奇妙的 iOS 加速计以及陀螺仪所带来的乐趣。我们创建了一个非常棒的愚人节应用，看到了将加速计用作控制设备的基础知识。我们可以使用加速计和陀螺仪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 iOS 硬件：内置摄像头。

众所周知，iPhone、iPad 和 iPod touch 都提供了内置摄像头和照片应用。照片应用可以帮助用户管理自己拍摄的各式照片和视频。但也许你还不知道，自己的应用可以使用内置摄像头来拍摄照片，而且，还可以让用户从这些设备的照片库中选择照片。本章就来看看这些功能。

## 21.1 图像选取器和 UIImagePickerController

由于 iOS 应用受到沙盒机制的限制，因此通常不能获取照片或自己沙盒之外的其他数据。幸运的是，应用可以通过**图像选取器**（image picker）使用摄像头和照片库。

顾名思义，图像选取器是从特定源中选择图片的一种机制。这个类最先出现于 iOS，只用于图像，但现在也可以用于捕捉视频。

通常来说，图像选取器会将一系列图像或视频作为它的源（参见图 21-1 左侧的图片）。不过，我们也可以指定摄像头作为源（参见图 21-1 右侧的图片）。

图像选取器界面是通过名为 UIImagePickerController 的模式控制器类实现的。首先，我们创建此类的一个实例，指定委托（如果没有的话），指定其图像源，指定希望选择图像还是视频，然后以模式方式启动。图像选取器会控制设备让用户从已有的媒体库中选择图片或视频，或者用户可以使用摄像头拍摄新照片或视频。用户选择之后，就可以对所选图像做一些基本的编辑，如缩放或裁剪。所有行为都是 UIImagePickerController 实现的，所以你不用费什么事儿。

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或视频会被传送到委托。无论用户选择了一个媒体文件还是点了取消，委托都有责任解除 UIImagePickerController，让用户返回到应用。

UIImagePickerController 的创建非常简单。我们按照对多数的类操作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 iOS 设备都有摄像头。老式 iPod touch 便是一个例子，第一代 iPad 也是，但是将来这类苹果设备会越来越少。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，在让用户使用摄像头拍摄照片之前，我们应先确保程序所在的设备上有摄像头。这一点可以使用 UIImagePickerController 的类方法进行检查，如下所示：

```
if ([UIImagePickerController isSourceTypeAvailable:
    UIImagePickerControllerSourceTypeCamera]) {
```

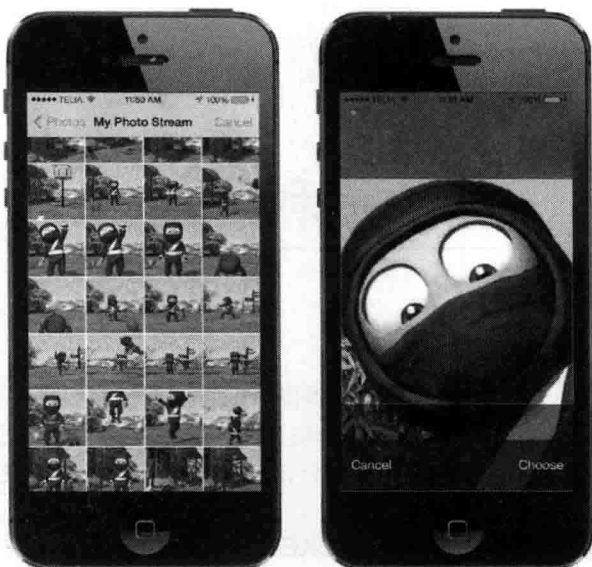


图 21-1 图像选取器的实际应用。左边呈现的是一列图像，右边呈现的是选择某个图像之后的样子，用户可以在这里对它进行移动和缩放。没错，在我的相册中有时都是 Clumsy Ninja（《笨拙的忍者》）的图片。这都是我那淘气的孩子捣的乱

本例中传递 UIImagePickerControllerSourceTypeCamera 表示想让用户使用内置摄像头拍照或录像。如果所指定的源当前可用，方法 isSourceTypeAvailable: 将返回 YES。除了 UIImagePickerControllerSourceTypeCamera，我们还可以指定另外两个值。

- ❑ UIImagePickerControllerSourceTypePhotoLibrary 指定用户将从现有的媒体库中选取照片或视频。照片将被返回到委托。
- ❑ UIImagePickerControllerSourceTypeSavedPhotosAlbum 指定了用户将从现有照片库中选择照片，但选择范围仅限于最近使用的相册。此选项也可以在没有摄像头的设备上运行，虽然用处不大，但是仍然可以用来选取之前保存的屏幕快照。

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很多：

```
UIImagePickerController *picker = [[UIImagePickerController alloc] init];
picker.delegate = self;
picker.sourceType = UIImagePickerControllerSourceTypeCamera;
[self presentViewController:picker animated:YES completion:nil];
```

在创建并配置了 UIImagePickerController 之后，我们使用类从 UIView 继承的 presentViewController:animated:completion: 方法将图像选取器呈现给用户。

---

**提示** presentViewController:animated:completion: 方法并不局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，我们可以将任何视图控制器以模态的方式呈现给用户。

---

## 21.2 实现图像选取器控制器委托

为了知道用户何时退出图像选取器界面，你需要实现 `UIImagePickerControllerDelegate` 协议。此协议定义了两个方法：`imagePickerController:didFinishPickingMediaWithInfo:`和 `imagePickerController:didCancel:`。

当用户成功拍摄了照片和视频，或从照片库中选择了相应项之后，`imagePickerController:didFinishPickingMediaWithInfo:`将被调用。第一个参数是指向之前创建的 `UIImagePickerController` 的指针。第二个参数是一个 `NSDictionary` 实例，包含用户所选照片或当前所选视频的 URL；如果允许在图像选取器中编辑（并且用户确实对图像或视频进行了编辑），那么第二个参数还包括可选的编辑信息。此字典还包含存储在键 `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示例：

```
(void)imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    UIImage *selectedImage = info[UIImagePickerControllerEditedImage];
    UIImage *originalImage = info[UIImagePickerControllerOriginalImage];

    // 对 selectedImage 和 originalImage 做一些处理

    [picker dismissViewControllerAnimated:YES completion:nil];
}
```

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。我们可以将这个 `NSValue` 实例转换为 `CGRect`：

```
NSValue *cropValue = info[UIImagePickerControllerCropRect];
CGRect cropRect = [cropValue CGRectValue];
```

完成转换之后，`cropRect` 将指定在编辑过程中所选定的原始图像部分。如果你不需要此信息，则可以忽略。

---

**警告** 如果返回到委托的图像来自摄像头，那么此照片不会被自动存储在照片库中。保存图像（如果需要）的工作将由应用负责。

---

在用户决定取消此过程，不再拍照或选择媒体时，另一个委托方法（`imagePickerController:didCancel:`）将被调用。当图像选取器调用此委托方法时，它就会通知委托说用户已经结束对选取器的使用，没有选择任何图像。

在 `UIImagePickerControllerDelegate` 协议中的两种方法都被标记为可选，但实际上不是，原因是：必须委托才能解除图像选取器这样的模态视图。事实是，在用户取消图像选取器时，即使不需要采取任何应用特定的操作，我们也仍然需要解除选取器。至少，`imagePickerController:didCancel:`方法要像这样才能保证程序正确运行：

```

- (void)imagePickerControllerDidCancel:(UIImagePickerController *)picker
{
    [picker dismissViewControllerAnimated:YES completion:NULL];
}

```

## 21.3 实际测试摄像头和照片库

本章构建的应用允许用户使用摄像头拍摄照片和视频，或从照片库中选择图片或视频，然后在图像视图中显示所选图片或视频（参见图 21-2）。如果用户使用的设备没有摄像头，我们就隐藏 New Photo or Video 按钮，只允许用户从照片库中选择图片。

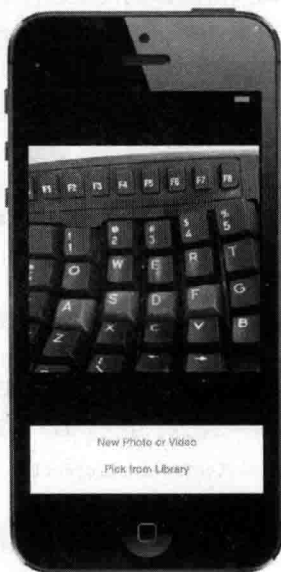


图 21-2 运行中的 Camera 应用

在 Xcode 中，请使用 Single View Application 模板创建一个新项目并将其命名为 Camera。首先要做的就是添加两个输出接口。我们需要一个指向图像视图的输出接口，以便可以使用从图像选取器返回的图像来更新它，还需要一个指向 New Photo or Video（新照片或视频）按钮的输出接口，以便可以在设备没有摄像头时隐藏该按钮。

我们还需要两个操作方法，一个用于 New Photo or Video 按钮，一个用于让用户从照片库中选择现有的照片。

我们要展开 Camera 文件夹，这里可以看到所有相关文件，然后选中 BIDViewController.h，并在类扩展中添加如下协议遵循声明和属性变量：

```

#import "BIDViewController.h"

@interface BIDViewController ()
<UIImagePickerControllerDelegate, UINavigationControllerDelegate>

```

```
@property (weak, nonatomic) IBOutlet UIImageView *imageView;
@property (weak, nonatomic) IBOutlet UIButton *takePictureButton;
```

```
@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，我们在使用图像选取器时不需要它们，但是必须遵循这个协议，否则编译器后面会发出警告。

你应该还会注意到的另一点：尽管将使用一个 UIImageView 实例显示所选图像，但没有类似的控件可以用于显示所选视频。UIKit 没有包含像 UIImageView 这样的公开的类用来显示视频内容，所以我们必须使用另一种方式来实现视频。需要的时候将使用一个 MPMoviePlayerController 实例，抓取它的 view 属性并将其插入到视图层次结构中。这是使用任何视图控制器的一种非常独特的方式，但也是苹果公司实际上已认可的在视图层次结构中显示视频的方式。

还要添加两个操作方法并连接到按钮上。目前我们只创建空的实现代码，这样界面构建器就可以找到它们了。之后将填充里面的代码：

```
- (IBAction)shootPictureOrVideo:(id)sender {
}

- (IBAction)selectExistingPictureOrVideo:(id)sender {
}
```

### 21.3.1 设计界面

请从库中拖出两个按钮，并将它们放置在标签为 View 的窗口中，将它们上下排列放置，底部按钮与底部的蓝色引导线对齐。我们双击最上面的按钮并将标题命名为 New Photo or Video，然后双击下面的按钮并将标题命名为 Pick from Library（从库中选取）。然后，请从库中拖出一个图像视图，将它放置在其他按钮上方，最后拉伸视图使它占据按钮上方的所有空间，如图 21-2 所示。

此时，请按住鼠标右键并从 View Controller 图标拖至图像视图，选择 imageView 输出接口。这后，请再次按下 Control 键并从 File's Owner 拖至 New Photo or Video 按钮，并选择 takePictureButton 输出接口。

然后，选择 New Photo or Video 按钮，并打关联检查器。我们从 Touch Up Inside 事件拖至 View Controller，并选择 shootPictureOrVideo:操作方法。接着，单击 Pick from Library 按钮，从关联检查器上的 Touch Up Inside 事件拖至 View Controller，并选择 selectExistingPictureOrVideo:操作方法。

完成这些关联之后，请保存修改。

### 21.3.2 实现摄像头视图控制器

请选择 BIDViewController.m，这里要做不少更改。因为我们要允许用户录像，所以需要有一个 MPMoviePlayerController 实例的属性变量。另外两个属性变量记录最后选择的照片和视频，还有一个字符串用来判断最后选择的是视频还是照片。我们还需要导入一些头文件使它们能正常执



行。添加以下粗体显示的语句：

```
#import "BIDViewController.h"
#import <MediaPlayer/MediaPlayer.h>
#import <MobileCoreServices/UTCoreTypes.h>

@interface BIDViewController ()
<UIImagePickerControllerDelegate, UINavigationControllerDelegate>

@property (weak, nonatomic) IBOutlet UIImageView *imageView;
@property (weak, nonatomic) IBOutlet UIButton *takePictureButton;

@property (strong, nonatomic) MPMoviePlayerController *moviePlayerController;
@property (strong, nonatomic) UIImage *image;
@property (strong, nonatomic) NSURL *movieURL;
@property (copy, nonatomic) NSString *lastChosenMediaType;

@end
```

现在我们要修改 `viewDidLoad` 方法，如果运行的设备没有摄像头，就要隐藏 `takePictureButton` 按钮。我们还要实现 `viewDidAppear:` 方法，让它调用 `updateDisplay` 方法，我们马上就会实现它。首先做出如下更改：

```
@implementation BIDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后（通常是从 nib 文件加载），做一些额外的设置。
    if (![UIImagePickerController isSourceTypeAvailable:
        UIImagePickerControllerSourceTypeCamera])
    {
        self.takePictureButton.hidden = YES;
    }
}

- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    [self updateDisplay];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // 删除可以重新创建的资源
}
```

你一定要理解 `viewDidLoad` 和 `viewDidAppear:` 方法之间的区别。前者仅在刚将视图加载到内存中时被调用，而后者可以在每次显示视图时被调用，这既包括启动时，也包括在显示另一个全屏视图（比如图像选取器）之后返回到控制器时。

接下来是 3 个工具方法，第一个是 `updateDisplay`，它会在 `viewDidAppear:` 方法中被调用，而 `viewDidAppear:` 方法会在视图初次创建时，以及每次用户选取图像或视频之后从图像选取器返



回时被调用。由于这里的双重用途，我们需要进行一些检查以确定选择的是图像还是视频，进而正确地设置 GUI。MPMoviePlayerController 类不允许更改它使用的 URL，所以每次要显示一段视频时都需要创建一个新的 MPMoviePlayerController 控制器实例。这个方法会处理所有的这些事情。只要在文件底部添加如下代码：

```
- (void)updateDisplay
{
    if ([self.lastChosenMediaType isEqual:(NSString *)kUTTypeImage]) {
        self.imageView.image = self.image;
        self.imageView.hidden = NO;
        self.moviePlayerController.view.hidden = YES;
    } else if ([self.lastChosenMediaType isEqual:(NSString *)kUTTypeMovie]) {
        [self.moviePlayerController.view removeFromSuperview];
        self.moviePlayerController = [[MPMoviePlayerController alloc]
                                       initWithContentURL:self.movieURL];
        [self.moviePlayerController play];
        UIView *movieView = self.moviePlayerController.view;
        movieView.frame = self.imageView.frame;
        movieView.clipsToBounds = YES;
        [self.view addSubview:movieView];
        self.imageView.hidden = YES;
    }
}
```

第二个工具方法是 pickMediaFromSource:，它会在两个操作方法中被调用。这个方法非常简单，它只是创建并配置一个图像选取器，使用传入的 sourceType 来确定应该显示摄像头还是媒体库。我们在文件底部添加如下代码：

```
- (void)pickMediaFromSource:(UIImagePickerControllerSourceType)sourceType
{
    NSArray *mediaTypes = [UIImagePickerController
                           availableMediaTypesForSourceType:sourceType];
    if ([UIImagePickerController
         isSourceTypeAvailable:sourceType] && [mediaTypes count] > 0) {
        NSArray *mediaTypes = [UIImagePickerController
                               availableMediaTypesForSourceType:sourceType];
        UIImagePickerController *picker = [[UIImagePickerController alloc] init];
        picker.mediaTypes = mediaTypes;
        picker.delegate = self;
        picker.allowsEditing = YES;
        picker.sourceType = sourceType;
        [self presentViewController:picker animated:YES completion:NULL];
    } else {
        UIAlertView *alert =
            [[UIAlertView alloc] initWithTitle:@"Error accessing media"
                                           message:@"Unsupported media source."
                                           delegate:nil
                                           cancelButtonTitle:@"Drat!"
                                           otherButtonTitles:nil];
        [alert show];
    }
}
```

第三个工具方法是 `shrinkImage:toSize:`，用于对图像进行缩小以适应显示它的图像视图。这样可以减小要使用的 `UIImage` 的尺寸，以及 `imageView` 为了显示图像所需的内存量。因为我们并不想要图像拉伸，所以要调整它的尺寸，这样就能以原始图像比率绘制。在文件最后添加如下代码：

```
- (UIImage *)shrinkImage:(UIImage *)original toSize:(CGSize)size
{
    UIGraphicsBeginImageContextWithOptions(size, YES, 0);

    CGFloat originalAspect = original.size.width / original.size.height;
    CGFloat targetAspect = size.width / size.height;
    CGRect targetRect;

    if (originalAspect > targetAspect) {
        // original is wider than target
        targetRect.size.width = size.width;
        targetRect.size.height = size.height * targetAspect / originalAspect;
        targetRect.origin.x = 0;
        targetRect.origin.y = (size.height - targetRect.size.height) * 0.5;
    } else if (originalAspect < targetAspect) {
        // original is narrower than target
        targetRect.size.width = size.width * originalAspect / targetAspect;
        targetRect.size.height = size.height;
        targetRect.origin.x = (size.width - targetRect.size.width) * 0.5;
        targetRect.origin.y = 0;
    } else {
        // original and target have same aspect ratio
        targetRect = CGRectMake(0, 0, size.width, size.height);
    }

    [original drawInRect:targetRect];
    UIImage *final = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();

    return final;
}
```

这里可以看到一系列调用，它们用于根据指定的尺寸创建一个新的图像，然后将旧图像渲染到新的图像中。

接下来，我们插入如下的操作方法：

```
- (IBAction)shootPictureOrVideo:(id)sender {
    [self pickMediaFromSource:UIImagePickerControllerSourceTypeCamera];
}

- (IBAction)selectExistingPictureOrVideo:(id)sender {
    [self pickMediaFromSource:UIImagePickerControllerSourceTypePhotoLibrary];
}
```

每一个对之前定义的工具方法的调用都会传入一个在 `UIImagePickerController` 中定义的值，用于指定照片或者视图的来源。

最后我们来实现选取器视图的委托方法：

```

#pragma mark - Image Picker Controller delegate methods
- (void)imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    self.lastChosenMediaType = info[UIImagePickerControllerMediaType];
    if ([self.lastChosenMediaType isEqual:(NSString *)kUTTypeImage]) {
        UIImage *chosenImage = info[UIImagePickerControllerEditedImage];
        self.image = [self shrinkImage:chosenImage
                               toSize:self.imageView.bounds.size];
    } else if ([self.lastChosenMediaType isEqual:(NSString *)kUTTypeMovie]) {
        self.movieURL = info[UIImagePickerControllerMediaURL];
    }
    [picker dismissViewControllerAnimated:YES completion:NULL];
}

- (void)imagePickerControllerDidCancel:(UIImagePickerController *)picker
{
    [picker dismissViewControllerAnimated:YES completion:NULL];
}

```

第一个委托方法检查用户是否选中了一张照片或者一个视频，用所选的文件做一些处理（比如缩小所选图像的尺寸以适应图像视图），然后解除图像选取器的模态显示。第二个委托方法仅仅用于解除图像选取器。

所需要做事情仅此而已。然后就可以编译并运行程序了。如果应用在模拟器上运行，则没有拍摄照片的选项，只能在照片库中选择（前提是模拟器照片库中有照片）。如果你有机会在实际设备上运行应用，请尝试这一操作；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。这个应用第一次需要在 iOS 上访问用户的照片时，系统会向用户询问是否允许进行这一访问，这是在 iOS 6 中用来确保应用不会没有经过用户同意而偷偷获取照片的一个新的隐私特性。

选择好照片之后，如果我们在单击 Use Photo 按钮之前放大或旋转图片，在委托方法中返回给应用的图像将会是裁剪后的图像。

## 21.4 小结

真难以置信，就这么简单，我们就可以让用户在应用使用摄像头拍摄并使用照片，甚至可以允许用户对拍摄的图像进行一些简单的编辑。

在下一章中，我们要学习的是将 iOS 应用翻译为其他语言，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！

本书写作之时，iPhone 已经遍及 90 个不同的国家，并且显而易见，此数字将会随时间的推移不断增长。现在，你可以在南极洲以外的任何大陆购买和使用 iPhone。iPad 和 iPod touch 也和 iPhone 一样在全世界都销售得炙手可热。

如果你计划通过 App Store 发布应用，那么潜在的用户远远不只是那些自己的国家中与你说同样语言的人们。恰好 iOS 拥有健壮的本地化体系结构，你不但可以轻松地将应用（或者由其他人将它）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用语风格吗？没问题。

一点问题都没有，只要你正确地编写了代码。改进现有的应用以支持本地化，比从头编写支持本地化的应用要困难得多。本章中，我们会讲述如何编写代码以方便地实现本地化，然后对一个应用示例进行本地化。

## 22.1 本地化体系结构

非本地化应用运行时，其所有文本都会以开发人员的语言呈现，也就是**开发基础语言**（development base language）。

当开发人员决定使其应用支持本地化时，他们会在应用包中为每种支持的语言创建一个子目录。每种语言的子目录都包含一个翻译为此种语言的应用资源子集。每个子目录都称为一个**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 扩展名。

在 iOS 的 Settings 应用中，用户可以设置设备的偏好语言和区域格式。例如，如果用户的母语是英语，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。

当本地化的应用需要载入某一资源（如图像、属性列表或 nib 文件）时，它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，那么它就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 iOS 语言，选择法国作为地区的用户，应用会先查找名为 fr\_FR.lproj 的本地化文件夹。文件夹名称的前两个字母是 ISO 国家代码，表示法语。下划线后的两个字母是 ISO 代码，表示法国。

如果应用找不到匹配的带有两个字母的代码，就会查找匹配的带有三个字母的 ISO 代码。在

我们的示例中, 如果应用找不到名为 `fr_FR.lproj` 的文件夹, 就会查找名为 `fre_FR` 或 `fra_FR` 的本地化文件夹。

所有语言都至少有一个三个字母的代码。某些语言有两个三个字母的代码, 一种是此语言的英语拼写, 一种是本地拼写。只有部分语言有二个字母的代码。当一种语言既有两个字母的代码又有三个字母的代码时, 我们最好使用两个字母的代码。

---

**注意** 你可以在 ISO 网站上找到当前的 ISO 国家(地区)代码列表。两位和三位代码都是 ISO 3166 标准的一部分: [http://www.iso.org/iso/country\\_codes.htm](http://www.iso.org/iso/country_codes.htm)。

---

如果应用找不到精确匹配的文件夹, 那么它会随即查找应用包中仅语言代码匹配(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`。如果这些项目都找不到, 它就会查找 `French.lproj`。最后一种结构是为了支持旧式 Mac OS X 应用而存在的, 一般来说, 我们应该避免使用它。

如果应用找不到与语言/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, 那么对于任何所需资源, 它将总是先查找这里。例如, 若通过 `imageName:` 方法载入一个 `UIImage`, 它会先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, 就使用它, 否则将会退回到基础语言资源。

如果某个应用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`。这样你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, 仅对受到不同方言或地理地区影响的资源实现本地化。

你只需要对受语言或国家(地区)影响的资源实现本地化。如果应用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。

## 22.2 字符串文件

在源代码中, 字符串字面量和字符串常量有何作用? 下面参考第 20 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];
```

如果已经努力完成了对特定受众的应用的本地化工作, 当然不想看到以开发基础语言编写的警告出现。上面采用的方法是, 将这些字符串存储到特定的文本文件中, 即存储到字符串文件(String file)中。

## 22.2.1 字符串文件

字符串文件实际上只是 Unicode 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中字符串文件的格式。

```
/* 用于表示用户的名 */
"LABEL_FIRST_NAME" = "First Name";
```

```
/* 用于表示用户的姓 */
"LABEL_LAST_NAME" = "Last Name";
```

```
/* 用于表示用户的生日 */
"LABEL_BIRTHDAY" = "Birthday";
```

在 `/*` 和 `*/` 字符之间的值只是注释。它们对应用来说没有用处，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在程序中的用处。

有人可能注意到了，在每一行代码中，相同的字符串出现了两次：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；等号右侧的值是翻译成的本地语言。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：

```
/* 用于表示用户的名 */
"LABEL_FIRST_NAME" = "Prénom";
```

```
/* 用于表示用户的姓 */
"LABEL_LAST_NAME" = "Nom de famille";
```

```
/* 用于表示用户的生日 */
"LABEL_BIRTHDAY" = "Anniversaire";
```

## 22.2.2 本地化的字符串宏

人们不会通过手动输入创建字符串文件。相反，我们总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。

下面的代码显示了宏的工作原理，首先以传统的字符串声明开始：

```
NSString *myString = @"First Name";
```

要本地化此字符串，我们需要这样做：

```
NSString *myString = NSLocalizedString(@"LABEL_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包中查找名为 `localizable.strings` 的字

符串文件。如果没有找到此文件，它返回其第一个参数，而此字符串会出现在开发基础语言中。在开发过程中，字符串通常只使用基础语言进行显示，因为应用还没有被本地化。

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，`NSLocalizedString` 将在字符串文件中搜索字符串 `"LABEL_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 `NSLocalizedString` 宏的第一个参数。

我们可以将基础语言文本用作 `NSLocalizedString` 宏的键，因为这个宏会在找不到匹配的本地化文本时返回第一个键参数。如果是这样的话，上面的例子看起来会是这样：

```
NSString *myString = NSLocalizedString(@"First Name",
    @"Used to ask the user his/her first name");
```

我们并不建议这么使用，这有两个原因。第一个原因是，你通常不太可能在第一次时就能为应用选则到完美的文本。返回字符串文件进行修改是非常麻烦的事情，而且很容易出现错误，这就意味着在应用中很可能找不到与键相匹配的文本。第二个原因是，若使用由大写字母组成的键，如果你忘记在字符串文件中添加相应的本地化文本，就很容易在运行应用时发现。

了解完本地化结构和字符串文件是怎样工作的，下面我们来看一看实际的使用效果。

## 22.3 现实中的 iOS 本地化应用

现在创建一个显示用户当前区域设置的小应用。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、货币和时间等信息。创建应用之后，我们需要将它本地化为其他语言。在此，你可以学习到如何实现分镜文件、字符串文件、图像，甚至是应用图标本地化。

图 22-1 展示了应用的外观。顶部的名称来自用户的区域设置。左侧的序数单词为静态标签，可以通过本地化分镜文件来设置它们的值。而右侧的单词和屏幕底部的国旗图像都会在运行时根据用户的区域由应用代码来选择。

现在让我们来开始本地化吧！

### 22.3.1 创建 LocalizeMe

在 Xcode 中使用 Single View Application 模板新建一个项目，并将其命名为 LocalizeMe。

查看本书项目归档中的 22 - LocalizeMe 文件夹，你可以找到一个名为 Images 的文件夹，在其中又可以找到两张名为 `flag_usa.png` 和 `flag_france.png` 图片。在 Xcode 中选中 Images.xcassets 文件夹，然后把 `flag_usa.png` 和 `flag_france.png` 两张图片拖入。

现在我们向项目的视图控制器添加输出接口，其中一个用于横跨视图顶部的蓝色标签，另一个用来显示国旗的图像视图，还有一个输出接口集合用来表示右侧所有的词条（参见图 22-1）。单击 `BIDViewController.h` 并作如下更改：



```
#import "BIDViewController.h"

@interface BIDViewController ()

@property (weak, nonatomic) IBOutlet UILabel *localeLabel;
@property (weak, nonatomic) IBOutlet UIImageView *flagImageView;
@property (strong, nonatomic) IBOutletCollection(UILabel) NSArray *labels;

@end
```

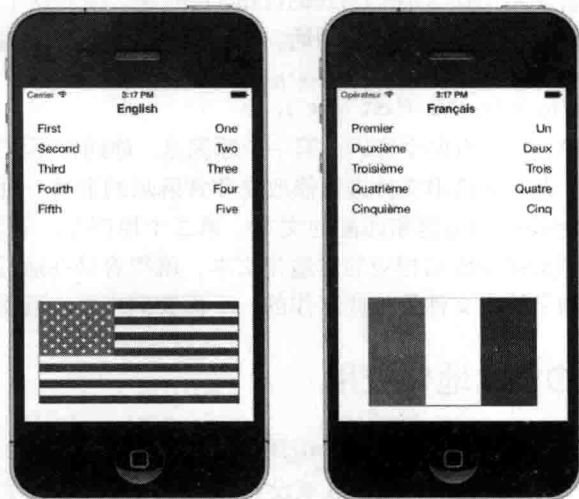


图 22-1 使用两种语言/地区设置进行显示的 LocalizeMe 应用

然后,选中 Main.storyboard,在界面构建器中编辑 GUI 界面,从库中拖出一个标签放置在窗口顶部,使之与顶部的蓝色引导线对齐。然后,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。然后,选定此标签,打开属性检查器,找到 Font 控件,单击其中包含的小 T 图标,调出一个小的字体选择弹出框,选择 System Bold 字体以使标签文本能够突出显示。接着使用属性检查器将文本改为居中对齐,并将字体颜色设置为亮蓝色。根据需要这里可以使用字体选择器加大字号。只要在属性检查器中选定了 Autoshrink (自动收缩),文本就会在太长不能适配时自动调整大小。

放置好标签之后,我们按下鼠标右键,将鼠标指针从 View Controller 图标拖到此新标签上,然后选择 localeLabel 输出接口。

然后,请在库中使用蓝色引导线将其他 5 个标签左对齐,上下依次放置,如图 22-1 所示。我们调整标签,使它们占据将近视图的一半(或更小),然后双击顶部的标签,把其中的文本由 Label 更改为 First。然后,我们对其他 4 个刚刚添加的标签重复此步骤,分别将标签文本设置为 Second、Third、Fourth 和 Fifth。

从库中再拖出 5 个标签,这次采用右对齐方式。然后,请使用属性检查器将文本对齐方式更改为右对齐,并增大标签的宽度,使之从右边的蓝色引导线伸展至视图中部。接着,我们按下鼠



标右键，将鼠标指针从 View Controller 图标拖至 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。以后我们会在程序运行过程中设置这些值。

最后，从库中拖出一个图像视图放到视图底部，使之紧挨底部和左侧的蓝色引导线。在属性检查器中，我们在视图的 Image 属性中选择 flag\_usa，调整图像大小使之位于两条蓝色引导线之间。然后，我们在属性检查器中将 Mode 属性由 Center（居中）改为 Aspect Fit（适配纵横比）。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，但这样还可以维持正确的纵横比（高度与宽度之比）。接下来将国旗调整得更宽，直至其边缘接触到右侧的蓝色引导线。最后右键从视图控制器拖动到这个图像视图并选择 flagImage 输出口。

保存分镜文件。然后切换到 BIDViewController.m，将以下代码添加到 viewDidLoad 方法：

```
(void)viewDidLoad
{
    [super viewDidLoad];
    // 视图加载完成之后（通常是从 nib 文件加载），做一些额外的设置。
    NSLocale *locale = [NSLocale currentLocale];
    NSString *currentLangID = [[NSLocale preferredLanguages] objectAtIndex:0];
    NSString *displayLang = [locale displayNameForKey:NSLocaleLanguageCode
                                                value:currentLangID];
    NSString *capitalized = [displayLang capitalizedStringWithLocale:locale];
    self.localeLabel.text = capitalized;

    [self.labels[0] setText:NSLocalizedString(@"LABEL_ONE", @"The number 1")];
    [self.labels[1] setText:NSLocalizedString(@"LABEL_TWO", @"The number 2")];
    [self.labels[2] setText:NSLocalizedString(@"LABEL_THREE",
                                                @"The number 3")];
    [self.labels[3] setText:NSLocalizedString(@"LABEL_FOUR", @"The number 4")];
    [self.labels[4] setText:NSLocalizedString(@"LABEL_FIVE", @"The number 5")];

    NSString *flagFile = NSLocalizedString(@"FLAG_FILE", @"Name of the flag");
    self.flagImageView.image = [UIImage imageNamed:flagFile];
}
```

在代码中，首先要获取一个代表用户当前区域设置的 NSLocale 实例。这个实例包含了用户的语言和地区首选项，即 iPhone 的设置应用里的设置。

```
NSLocale *locale = [NSLocale currentLocale];
NSString *currentLangID = [[NSLocale preferredLanguages] objectAtIndex:0];
```

关于代码的下一行可能需要多一点解释。NSLocale 的工作原理类似于字典，其中包含关于当前用户首选项的成批信息，包括所使用的货币的名称和期望的日期格式。NSLocale 的 API 参考中有这些信息的完整列表。

在代码的下一行中，我们使用名称为 displayNameForKey:value: 的方法来获取所选语言的实例名称，翻译为当地语言版本。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。我们可以使用此方法找到任何关于区域设置的数据，以便对任意用户进行正确的显示。在此例中我们想让用户语言的显示名称以当前使用语言来表示，所以为第 2 个变量传递了 currentLangID。这个字符串是两个字

母的语言编码，与之前创建语言项目时所使用的类似。对于美国的英语使用者来说，它是 `en`；对于法国的法语使用者来说，它是 `fr`：

```
NSString *displayLang = [locale displayNameForKey:NSLocaleLanguageCode
                                value:currentLangID];
```

我们所得到的名称应该像 `English` 或 `français` 这样——而且它只有用户语言经常要首字母大写时才会有大写。英语就是这种情况，而法语则不是。我们想要标题显示的名称是首字母大写的。幸运的是，`NSString` 包含了对字符串首字母大写的方法，并且遵循当地字符串首字母大写的规范！我们现使用它将 `français` 转换成 `Français`。

```
NSString *capitalized = [displayLang capitalizedStringWithLocale:locale];
```

有了显示名称，我们就可以用它设置视图顶部的标签：

```
self.localeLabel.text = capitalized;
```

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，我们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被转换为注释，我们可以使用这些注释方便地与相应的翻译人员进行沟通：

```
[self.labels[0] setText:NSLocalizedString(@"LABEL_ONE", @"The number 1")];
[self.labels[1] setText:NSLocalizedString(@"LABEL_TWO", @"The number 2")];
[self.labels[2] setText:NSLocalizedString(@"LABEL_THREE",
                                         @"The number 3")];
[self.labels[3] setText:NSLocalizedString(@"LABEL_FOUR", @"The number 4")];
[self.labels[4] setText:NSLocalizedString(@"LABEL_FIVE", @"The number 5")];
```

最后我们要查找国旗图像的名称，字符串对应的图像会放到图像视图中。

```
NSString *flagFile = NSLocalizedString(@"FLAG_FILE", @"Name of the flag");
self.flagImageView.image = [UIImage imageNamed:flagFile];
```

现在，请运行此应用。

### 22.3.2 测试 LocalizeMe

我们可以使用模拟器或某台设备测试 `LocalizeMe`。模拟器会缓存某些语言和地区设置，也许你更希望在真实设备（如果有的话）上完成测试。启动后的应用如图 22-2 所示。

我们使用 `NSLocalizedString` 宏代替静态字符串，已经做好了本地化的准备，但还没有开始本地化（通过右侧标签的大写内容和底部缺少的国旗图像就可以很明显地看出来）。如果使用模拟器或 iPhone 上的设置应用更改为另一种语言或另一个地区，那么结果看上去应该基本相同，视图顶部的标签除外（参见图 22-3）。

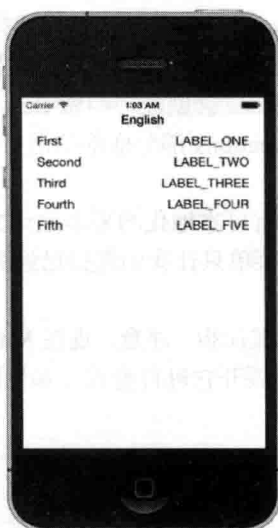


图 22-2 系统将在作者的基本语言下运行。应用针对本地化进行了设置，但尚未经过本地化处理



图 22-3 将 iPhone 的语言和地区设置为在法国使用法语，非本地化的应用在运行时看起来就成了这个样子

### 22.3.3 本地化项目

现在开始对项目进行本地化处理。我们在 Xcode 的项目导航面板中点击 LocalizeMe，在编辑

区域（不是 Targets 区域）中点击 LocalizeMe 项目，然后选择项目的 Info 标签。

请在 Info 标签中找到 Localizations 部分。可以看到，其中显示了一个本地化选项：Base 和 English。在创建一个新项目时，Xcode 会创建一个叫做 Base 的本地化项，指定为开发者的语言。我们想要添加法语，所以点击 Localizations 部分底部的加号（+）按钮，然后从出现的弹出列表中选择 French(fr)（参见图 22-4）。

接下来，Xcode 会要求你选择所有可本地化的需本地化文件以及基础语言，以便进行法语本地化。（参见图 22-5）此时每个弹出菜单只让我们选择已经列出的选项（Base 或 English），因此请选中所有的文件，然后点击 Finish。

添加好之后，你要看一下项目导航面板。注意，现在 Main.storyboard 文件旁有一个展开三角形，表明它是一个组或文件夹。我们展开它进行查看（参见图 22-6）。

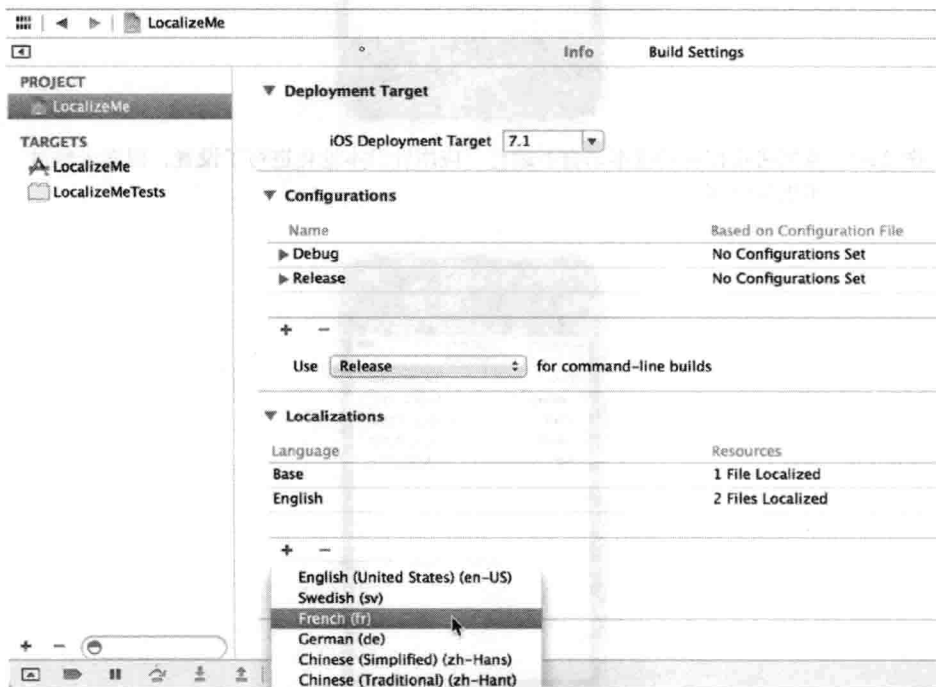


图 22-4 项目的 Info 设置中显示了本地化设置以及其他信息

在该项目中，Main.storyboard 显示为包含两个子项的组。第一个名称是 Main.Storyboard 并标记为 Base，第二个名称是 Main.strings 并标记为 French。Base 是在你创建项目时自动创建的，它代表的是开发基础语言。

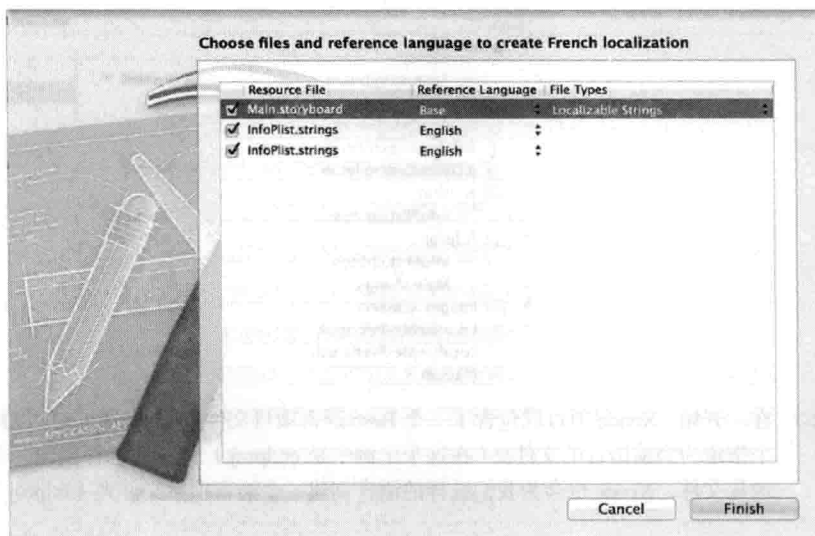


图 22-5 选择需要进行本地化的文件



图 22-6 在可本地化文件中，展开一个展开三角形，就能显示所添加的每种语言和地区的子值

这些文件分别位于一个单独的文件夹中，名为 `Base.lproj` 和 `fr.lproj`。我们进入 Finder，打开 `LocalizeMe` 项目文件夹中的 `LocalizeMe` 文件夹。除了所有的项目文件，应该还能看到名为 `Base.lproj` 和 `fr.lproj` 的文件夹，如此你也应该看到一个名称为 `en.lproj` 的文件夹。（参见图 22-7）。

注意，`Base.lproj` 文件夹始终都是存在的，其中包括 `Main.storyboard` 的副本。`en.lproj` 也是始终存在的，并包含了一个空的字符串文件。当 Xcode 发现资源只有一个本地化版本时，就作为一个单独项显示。如果一个文件拥有两个或多个本地化版本，它们就作为一个组显示。



图 22-7 在一开始, Xcode 项目就包含了一个 Base 语言项目文件夹 (Base.lproj), 以及一个指定为当前语言的文件夹 (在这个示例中是 en.lproj)。当我们选择创建一个本地化文件, Xcode 也会为我们选择的语言创建一个语言项目文件夹 (fr.lproj)

**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。iOS 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。

在 Xcode 中创建法语本地化文件时, Xcode 将在名为 fr.lproj 的项目文件夹中创建一个新的本地化项目, 并将两个字符串文件放到此处。其中一个是从 en.lproj 文件夹内 InfoPlist.strings 的副本, 另一个字符串文件包含了从 Base.lproj 或 Main.storyboard 提取的值。Xcode 没有复制整个分镜文件, 而只是提取了分镜中的每个文本字符串并创建一个用以本地化的字符串文件。之后应用编译并运行时, 字符串文件中的值会替换分镜中的值。

### 22.3.4 初始化分镜

在 Xcode 的项目导航栏中, Main.storyboard 应该有两个子项: Main.storyboard(Base) 和 Main.strings(French)。如果你已经展开了 Supporting File 分组, 你将会看到 InfoPlist.string 文件也有一个类似的结构。选择 Main.strings(French) 以打开字符串文件, 里面的值将会插入分镜中以显示给说法语的人。你会看到像下面这样的文本:

```
/* Class = "UILabel"; text = "Second"; ObjectID = "Agv-gm-Tho"; */
"Agv-gm-Tho.text" = "Second";

/* Class = "UILabel"; text = "Fourth"; ObjectID = "HiM-7A-r08"; */
"HiM-7A-r08.text" = "Fourth";

/* Class = "UILabel"; text = "Label"; ObjectID = "JHX-Zt-53a"; */
"JHX-Zt-53a.text" = "Label";
```

```
/* Class = "UILabel"; text = "Label"; ObjectID = "KVR-s0-2C5"; */
"KVR-s0-2C5.text" = "Label";
.
.
.
```

这里的每一行表示分镜中的一个字符串。注释内容提示的是：究竟是哪个类的对象包含了该字符串和原始字符串以及每个对象唯一的 ObjectID。注释下面那行的右侧就是你实际想要改变的值。你会看到一些像 First 这样的序数单词。它们是位于左侧的那些标签，在分镜中已有了名称。其他为 Label 的词对应的是右侧保留默认标题的那些标签。找到每个包含序数单词的标签文本 First、Second、Third、Fourth、Fifth，并将等号右侧的字符串分别改为 Premier、Deuxième、Troisième、Quatrième 和 Cinquième。最后请保存文件。

现在，分镜文件已经本地化为法语。我们编译并运行此程序。如果你已经在设置应用中将语言变为法语，应该能在左侧看到翻译后的标签。没有的话，打开设置应用，将语言切换到法语，并再次从 Xcode 启用应用。如果你不太确定如何修改，没关系，下面我们带你详细过一遍。

在模拟器中，我们打开设置应用并选择通用（general）行，然后选择标签为多语言设置（International）的行。此处可以更改语言和地区首选项（参见图 22-8）。



图 22-8 更改语言和地区，这两项设置会影响用户的区域和语言设置

你希望首先更改区域格式（Region Format），因为一旦你更改语言，iOS 就会重置并且返回主屏幕。我们将区域格式由 United States 改为 France（先选择 French，然后从弹出的新表中选择 France），然后将语言由 English 改为 Français。现在请按下完成按钮，模拟器将重置其语言。现在 iPhone 已经被设置为使用法语。

再次运行应用。这次，左侧的文字应该会以法语显示（参见图 22-9）。现在的问题是，国旗和右边的文本依然没有翻译。我们会在下一章节中进行更改。

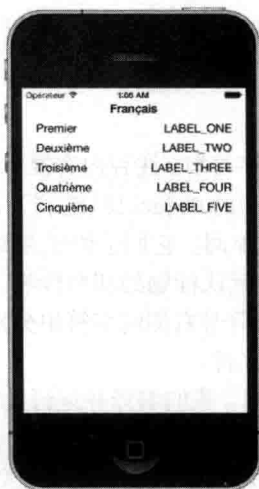


图 22-9 现在应用已经部分翻译为法语

### 22.3.5 创建并本地化字符串文件

图 22-9 显示视图右侧的单词仍然是全大写格式；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，我们需要暂时脱离 Xcode 的“温柔陷阱”。

请启动/Applications/Utilities/中的 Terminal.app。当终端窗口打开时，我们键入 `cd` 及一个空格。请不要按 `return` 键。

现在，打开 Finder，将存储 LocalizeMe 项目的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会显示在命令行上。这时再按 `return` 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 `genstrings` 程序，使之查找 `localizeMe` 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 `return` 键：

```
genstrings ./LocalizeMe/*.m
```

命令执行完之后（在这个小项目中只需半秒钟），你将返回至命令行。在 Finder 中，我们在项目文件夹中找到一个名为 `Localizable.strings` 的新文件，将它拖到 Xcode 项目导航面板的 `LocalizeMe` 文件夹中，但在弹出提示时，先不要单击 `Add` 按钮。首先，取消选中 `Copy items into destination group's folder(if needed)` 复选框，因为该文件已经在项目文件夹中了。点击 `Finish` 导入该文件。

---

**警告** 我们可以随时返回 `genstrings`，重新创建基础语言文件，但是在将字符串文件本地化为另一种语言之后，就要避免修改任何 `NSString()` 宏中所使用的文本。基础语言版本的字符串充当检索翻译的键，因此如果修改了文本，就再也找不到翻译后的版本了，那样将需要更新本地化的字符串文件或重新翻译它。

---



导入文件之后，单击 `Localizable.strings` 并查看它。它应该包含 6 个条目，因为我们对 5 个不同的值使用了 6 次 `NSLocalizedString`。传递给第 2 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。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方式，但在多数情况下，按字母排序会很有帮助。

```
/* 数字 5 */
"FLAG_FILE" = "FLAG_FILE";

/* 数字 5 */
"LABEL_FIVE" = "LABEL_FIVE";

/* 数字 4 */
"LABEL_FOUR" = "LABEL_FOUR";

/* 数字 1 */
"LABEL_ONE" = "LABEL_ONE";

/* 数字 3 */
"LABEL_THREE" = "LABEL_THREE";

/* 数字 2 */
"LABEL_TWO" = "LABEL_TWO";
```

现在对它进行本地化。

首先，我们单击 `Localizable.strings`，重复执行进行其他本地化时的步骤。

❑ 打开文件检查器，如果它还没有被打开的话。

❑ 在 `Localizations` 部分，点击 `localized...` 按钮，会调出一个小的面板，你可以在此选择原始的本地化版本，将现存的文件移动过去；你应该保留这个设置为 `Base`，然后点击面板上的 `Localize` 按钮。

❑ 回到文件检查器的 `Localization` 部分，检查 `French` 和 `English` 项并对它们的语言进行本地化。

由于 Xcode 中的某 bug，你可能会发觉无法完全按照上述内容执行步骤。大概就是完成第二步之后，Xcode 会丢失所选文件的响应，你必须在项目导航栏中再次点击 `Localizable.strings`。这倒也没什么。

返回至项目导航面板，点击 `Localizable.strings` 旁边的展开三角形并选择文件的 `English` 本地化内容，在编辑窗口中作如下更改：

```
/* 国旗名称 */
"FLAG_FILE" = "flag_usa";

/* 数字 5 */
"LABEL_FIVE" = "Five";

/* 数字 4 */
"LABEL_FOUR" = "Four";

/* 数字 1 */
"LABEL_ONE" = "One";

/* 数字 3 */
"LABEL_THREE" = "Three";
```

```
/* 数字 2 */
"LABEL_TWO" = "Two";
```

接下来选择文件的 French 本地化内容。在编辑器中做如下更改：

```
/* 国旗名称 */
"FLAG_FILE" = "flag_france";
```

```
/* 数字 5 */
"LABEL_FIVE" = "Cinq";
```

```
/* 数字 4 */
"LABEL_FOUR" = "Quatre";
```

```
/* 数字 1 */
"LABEL_ONE" = "Un";
/* 数字 3 */
"LABEL_THREE" = "Trois";
```

```
/* 数字 2 */
"LABEL_TWO" = "Deux";
```

在现实生活中（除非你掌握了多种语言），你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。得益于多年观看《芝麻街》学到的知识，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，可以看到右侧的标签也都被翻译为法语了（参见图 22-10）。在屏幕底部，现在应该会看到法国国旗了。



图 22-10 终于实现了完全的本地化

使用设置应用切换回 English 并返回应用，你将看到现在全部是英文，底部也是美国国旗了。



图 22-11 恢复成英文翻译和美国国旗了

### 22.3.6 应用显示名称的本地化

最后来看一种常见的本地化操作：本地化在主屏幕上和其他地方显示的应用名称。苹果公司对多个内置的应用都做了名称的本地化，你可能也希望这么做。

用于显示的应用名称存储在应用的 Info.plist 文件中，本例中的文件名是 LocalizeMe-Info.plist，它就在 Supporting Files 文件夹中。选择此文件以便编辑，我们将看到它包含的 Bundle display name 目前被设置为 `{PRODUCT_NAME}`。

在 Info.plist 文件所使用的语法中，任何以美元符号开头的实体都可以执行变量替换。在本例中，这意味着当 Xcode 编译该应用时，此项的值将替换为此 Xcode 项目中的产品（product）名称，也就是应用本身的名称。我们希望在本地化时，将 `{PRODUCT_NAME}` 替换为每种语言的本地化名称。但是，这并不像预料中那么简单。

Info.plist 文件比较特殊，但这并不意味着它不需要实现本地化。相反，如果希望本地化 Info.plist 的内容，需要创建 InfoPlist.strings 文件的本地化版本。这里有个好消息：Xcode 创建的每一个项目中都已经包含了这个文件，这里只需将它本地化。

我们在 Supporting Files 文件夹找到 InfoPlist.strings 文件，在文件检查器的 Localizations 部分按照以前的本地化步骤创建一个法语本地化版本（先是创建了 en.lproj 文件夹中的英语版本）。

现在需要添加一行代码来定义应用的显示名称。在 LocalizeMe-Info.plist 文件中，我们可以看到显示名称与一个名为 Bundle display name 的字典的键相关联，但这不是真正的键名！它只是 Xcode 的一项人性化功能，用于提供更加友好和易懂的名称。真正的名称是 CFBundleDisplayName，要进行证实，你可以选择 LocalizeMe-Info.plist，右键单击视图中的任何地方，然后选择

Show Raw Keys/Values (显示原始键/值), 这将显示所使用的键的真实名称。

所以, 选中英语本地化版本的 InfoPlist.strings, 添加以下代码:

```
CFBundleDisplayName = "Localize Me";
```

现在, 选中法语本地化版本的 InfoPlist.strings 文件, 编辑该文件, 为应用提供一个合适的法语名称:

```
CFBundleDisplayName = "Localisez Moi";
```

编译并运行这个应用, 然后按下主屏幕按钮以回到启动屏幕。如果你现在是以英语运行的, 还要切换当前设备或模拟器到法语环境。你会在应用图标下面看到本地化名称, 不过有时它不会立即出现。iOS 可能会在添加新应用时缓存这些信息, 但在将现有应用替换为新版本时不一定会更改该信息, 至少 Xcode 执行替换时不会这么做。所以如果你使用法语版本运行, 但没有看到新名称, 不要担心。我们只需从启动屏幕删除该应用, 返回到 Xcode 再次编译并运行即可。

现在, 应用已针对法语实现了全面的本地化。

## 22.4 小结

若要让 iOS 应用热销, 你就应该尽可能地实现本地化。好在 iOS 的本地化体系结构使应用可以轻松地支持多种语言, 甚至是同一种语言的多种方言。通过本章我们可以了解到, 几乎所有能够被添加到应用的文件都可以按需要进行本地化。

如果不打算对应用进行本地化, 那么也请在代码中使用 `NSLocalizedString`, 而不是只使用静态字符串。有了 Xcode 的 Code Sense 功能, 输入时间的差异可以忽略; 这样, 一旦你准备将应用翻译为其他语言, 只需很少的工作就可以了。若你很久之后再回到项目中查找所有需要被本地化的文本字符串, 这会是非常烦人而且容易出错的事儿。预先做一点工作就可以避免这种情况了。

至此, 我们已经到达了旅途的终点。是时候说再见了。

我们在本书中探讨的编程语言和框架是经历了 25 年演化的成果。苹果公司的工程师们夜以继日绞尽脑汁地研究下一个神奇的新产品。iOS 平台仅仅是一个开端, 还有更多优秀的东西即将到来。

通过整本书的内容, 你已经打下了扎实的基础, 对 Objective-C、Cocoa Touch 以及通过这些技术协调来创建全新的 iPhone、iPod touch 和 iPad 应用有了全面的了解。你也理解了 iOS 的软件架构——创建 Cocoa Touch 的设计方案。总而言之, 你学成了所有的课程, 我们为此感到骄傲!

很高兴能同你一起共度这段旅程, 希望你能一路顺利并能像我们这样喜欢 iOS 编程开发。