

- 数百个项目案例 + 两个真实商业项目开发全过程
- 涵盖 iOS 平台架构设计、测试驱动开发、性能优化、版本控制和程序调试等内容
- 精彩手绘原型草图，艺术与科技的结合



iOS 开发指南

从零基础到App Store上架

关东升 编著

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。



关东升

国内知名 iOS 技术作家，iOS 技术顾问，高级培训讲师，移动开发专家。精通 iOS、Android 和 Windows Phone 及 HTML5 等移动开发技术。曾先后主持开发大型网络游戏神农诀的 iOS 和 Android 客户端开发，国家农产品追溯系统的 iPad 客户端开发，酒店预订系统的 iPhone 客户端开发，金融系统微博的 iOS、Windows Phone 7、Android 客户端开发。在 App Store 上发布多款游戏和应用软件，擅长移动平台的应用和游戏类项目开发。近期为中国移动研究院、云南移动、东软、方正科技、大唐电信、中石油、深圳康拓普、上海财富 168、天津港务局等企事业单位授课。

著有《iOS 网络编程与云端应用最佳实践》、《iPhone 与 iPad 开发实战——iOS 经典应用剖析》、《Android 开发案例驱动教程》、《Android 网络游戏开发实战》、《移动平台用户体验设计》以及《JSP 网络程序设计》等图书。

本书配套网站：<http://www.iosbook1.com>

iOS 开发指南

从零基础到App Store上架

关东升 编著

人 民 邮 电 出 版 社

图灵社区会员 叶清泉(qqyadf@126.com) 专享 尊重版权

图书在版编目 (C I P) 数据

iOS开发指南：从零基础到App Store上架 / 关东升
编著. -- 北京：人民邮电出版社，2013.7
(图灵原创)
ISBN 978-7-115-32444-3

I. ①i… II. ①关… III. ①移动终端—应用程序—
程序设计—指南 IV. ①TN929.53-62

中国版本图书馆CIP数据核字(2013)第150303号

内 容 提 要

本书共4部分：第一部分为基础篇，介绍了iOS的一些基础知识；第二部分为网络篇，介绍了iOS网络开发相关的知识；第三部分为进阶篇，介绍了iOS高级内容、商业思考等；第四部分为实战篇，从无到有地介绍了两个真实的iOS应用——MyNotes应用和2016里约热内卢奥运会应用。书中包括了100多个完整的案例项目源代码，大家可以到本书网站<http://www.iOSBook1.com>下载。

本书适合iOS开发人员阅读。

-
- ◆ 编 著 关东升
责任编辑 王军花
责任印制 焦志炜
 - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京 印刷
 - ◆ 开本：850×1168 1/16
印张：44
字数：1 376千字
印数：1—3 500册
- 2013年7月第1版
2013年7月北京第1次印刷

定价：99.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223

反盗版热线：(010)67171154

广告经营许可证：京崇工商广字第0021号

前言

2012年9月19日，苹果公司开放 iOS 6 下载。2012年12月14日，iPhone 5 在中国发售，此时 iOS 6 正渐入佳境。本书在 iOS 6 第一个 beta 阶段就开始了编写，想在本书中包括 iOS 6 的新增特性，并将其作为第一本国内原创 iOS 6 图书奉献给广大读者。几个月过去了，我们终于在 2013 年元旦之前将书稿提交给出版社。几个月来，我们智捷 iOS 课堂团队夜以继日，我几乎推掉一切社交活动，推掉很多企业邀请我去讲课的机会，每天工作 12 小时，不敢有任何松懈，不敢有任何模棱两可，只做一件事情——编写此书。

内容和组织结构

本书是我们团队编写 iOS 系列丛书中的一本，目的是使一个有 Objective-C 基础的程序员通过学习本书，从零基础学习如何在 App Store 上发布一款应用。全书共 4 部分。

第一部分为基础篇，共 11 章内容，介绍了 iOS 的一些基础知识。

第 1 章介绍了 iOS 的开发背景以及本书约定。

第 2 章使用 nib 和故事板技术创建了 HelloWorld，同时讨论了 iOS 工程模板、应用的运行机制和生命周期、视图器的生命周期等，最后介绍了如何使用 API 帮助文档和官方案例。

第 3 章讨论了 iOS 开发中 4 种常用的设计模式，分别为单例模式、委托模式、观察者模式和 MVC 模式。在介绍每种设计模式时，我们按照问题提出、实现原理、应用案例的结构介绍了其适用情况、实现原理及其用法。

第 4 章首先学习了视图和控件之间的关系以及应用界面的构建层次，然后介绍了标签、按钮、文本框和导航栏等基本控件，接着介绍了屏幕布局的内容以及一个较为复杂的控件——选择器，最后探讨了 iOS 6 中的集合视图。

第 5 章探讨了表视图的组成、表视图类的构成和表视图的分类，使我们对表视图有了一个整体上的认识。接下来介绍了如何实现简单表视图和分节表视图，以及表视图中索引、搜索栏和分组的使用，然后学习了如何对表视图单元格进行删除、插入、移动等操作，最后介绍了表视图 UI 设计模式方面的内容。

第 6 章讨论了如何判断应用是不是需要一个导航功能，并且知道在什么情况下选择平铺导航、标签导航、树形结构导航，或者同时综合使用这 3 种导航模式。

第 7 章首先介绍了 iPhone 和 iPad 设备使用场景上的差异，然后深入介绍了 iPad 专用 API，接着使用 nib 和故事板技术分别实现了两个重要的程序模板，最后介绍了 iOS 平台的分层架构设计。

第 8 章介绍了设置和配置的概念，然后通过对二者差异的探讨，介绍了什么样的项目适合放在设置里，什么样的项目适合放在配置里。

第 9 章介绍了本地化概念、内容和目录结构，接下来详细阐述了文本信息、nib 及故事板、资源文件的本地化。

第 10 章讨论了 iOS 本地数据持久化的问题。首先分析了数据存取的几种方式以及每种数据存取方式适合什么样的场景，然后分别举例介绍了每种存取方式的实现。

第 11 章首先介绍了访问通讯录所需要的框架，然后介绍了使用 AddressBook 框架如何读取联系人的信息，具体包括联系人记录、单值多值属性、图片属性的读取；接着介绍了如何使用该框架将联系人信息写入数据库，具体包括联系人的创建、修改和删除；最后介绍了如何使用 AddressBook 高级 API 实现选择联系人、显示和修改联系人以及创建联系人的操作。

第二部分为网络篇，共两章，介绍了 iOS 网络开发的相关知识。

第 12 章介绍了数据交换格式，其中 XML 和 JSON 是主要的方式。这里重点介绍了 Web Service 的访问以及 ASIHTTPRequest 框架。

第 13 章讨论了 iOS 中的定位服务技术，包括地理信息编码和反编码查询。之后介绍了 iOS 6 苹果地图的使用，包括了显示地图、在地图上添加标注以及跟踪用户位置的变化等。最后，介绍了程序外地图的使用，如何调用 iOS 6 苹果地图和谷歌 Web 地图。

第三部分为进阶篇，共 6 章，介绍了 iOS 高级内容和商业思考等，相关内容如下所示。

第 14 章介绍了 iOS 中的商业模式，其中的收费策略值得广大读者借鉴。此外，我们还介绍了植入广告和应用内购买的 API，其中植入广告包括苹果自己的 iAd 和谷歌的 AdMob 广告。

第 15 章首先介绍了有哪些调试工具，然后重点介绍了几个常用的调试工具，具体包括日志与断言的输出、LLDB 调试工具、异常堆栈报告分析，接下来讲解了如何在真机上调试应用，最后分析了 Xcode 设备管理工具的使用。

第 16 章讨论了测试驱动的 iOS 开发，介绍了测试驱动开发流程。此外，我们还学习了单元测试框架 OCMock、GKUnit 和 OCMock 的用法。

第 17 章介绍了 iOS 中的性能优化方法，其中包括内存优化、资源文件优化、延迟加载、持久化优化、使用可重用对象、多线程以及程序编译参数等。

第 18 章介绍了如何使用 Git 进行代码版本控制，其中包括 Git 服务器的搭建、Git 常用命令和协同开发。此外，还介绍了在 Xcode 中如何配置和使用 Git 工具。

第 19 章探讨了如何在 App Store 上发布应用，介绍了应用的发布流程以及应用审核不通过的一些常见原因。

第四部分为实战篇，共两章，从无到有地介绍了两个真实的 iOS 应用：MyNotes 应用和 2016 里约热内卢奥运会应用。

第 20 章通过重构 MyNotes 应用，把 MyNotes 应用的数据由原来的本地存储变成云存储。在这个过程中，我们介绍了移动网络通信应用中分层架构设计的必要性和重要性。我们重点介绍了基于委托模式和观察者模式通知机制实现的分层架构设计。

第 21 章介绍了完整的 iOS 应用分析设计、编程、测试和发布过程，其中采用了敏捷开发方法。此外，该项目采用分层架构设计，这对于学习 iOS 架构是非常重要的。

书中并没有包括多媒体等知识，我们会在另外一本 iOS 游戏开发书中介绍，具体进展请读者关注我们智捷 iOS 课堂官方网站 <http://www.51work6.com>。

本书网站

为了更好地为广大读者提供服务，我们专门为本书建立了一个网站 <http://www.iosbook1.com>，大家可以查看相关出版进度，并对书中内容发表评论，提出宝贵意见。

源代码

书中包括了 100 多个完整的案例项目源代码，大家可以到本书网站 <http://www.iosbook1.com> 下载或者到图灵社区（www.it-ebooks.com）本书主页免费注册下载。

勘误与支持

我们在网站 <http://www.iosbook1.com> 中建立了一个勘误专区，可以及时地把书中的问题、失误和纠正反馈给广大读者。如果你发现了任何问题，均可以在网上留言，也可以发送电子邮件到 eeorient@sina.com，我们会在第

一时间回复你。此外，你也可以通过新浪微博与我们联系，我的微博为@tony_关东升。

致谢

在此感谢图灵的王军花责编给我们提供的宝贵意见，感谢智捷 iOS 课堂团队的李玉超和贾云龙参与内容讨论和审核，感谢赵大羽老师手绘了书中全部草图，并从专业的角度修改书中图片，力求更加真实完美地奉献给广大读者。此外，还要感谢我的家人容忍我的忙碌，以及对我的关心和照顾，使我能抽出这么多时间，投入全部精力专心编写此书。

由于时间仓促，书中难免存在不妥之处，请读者原谅。

关东升
2012 年 12 月于北京

目 录

第一部分 基础篇

第 1 章 开篇综述	2
1.1 iOS 概述	2
1.1.1 iOS 介绍	2
1.1.2 iOS 6 新特性	2
1.2 开发环境及开发工具	3
1.3 本书中的约定	4
1.3.1 案例代码约定	4
1.3.2 图示的约定	5
第 2 章 第一个 iOS 应用程序	7
2.1 创建基于 nib 的 HelloWorld 工程	7
2.1.1 创建工程	7
2.1.2 Xcode 中的 iOS 工程模板	10
2.1.3 应用剖析	11
2.2 基于故事板的 HelloWorld 工程	13
2.2.1 使用故事板重构 HelloWorld	14
2.2.2 nib、xib 与故事板	15
2.2.3 故事板中的 Scene 和 Segue	16
2.3 应用生命周期	17
2.3.1 非运行状态——应用启动场景	18
2.3.2 点击 Home 键——应用退出场景	19
2.3.3 挂起重新运行场景	20
2.3.4 内存清除——应用终止场景	21
2.4 视图生命周期	21
2.4.1 视图生命周期与视图控制器关系	22
2.4.2 iOS 6 UI 状态保持和恢复	23
2.5 设置产品属性	25
2.5.1 Xcode 中的 Project 和 Target	25
2.5.2 设置常用的产品属性	27
2.6 iOS API 简介	29
2.6.1 API 概述	29
2.6.2 如何使用 API 帮助	31
2.7 小结	33

第 3 章 iOS 常用设计模式	34
3.1 单例模式	34
3.1.1 问题提出	34
3.1.2 实现原理	34
3.1.3 应用案例	35
3.2 委托模式	37
3.2.1 问题提出	37
3.2.2 实现原理	39
3.2.3 应用案例	41
3.3 观察者模式	44
3.3.1 问题提出	44
3.3.2 实现原理	45
3.3.3 通知机制和 KVO 机制	47
3.4 MVC 模式	53
3.4.1 MVC 模式概述	53
3.4.2 Cocoa Touch 中的 MVC 模式	54
3.5 小结	56
第 4 章 UIView 与控件	57
4.1 视图“始祖”——UIView	57
4.1.1 UIView “家族”	57
4.1.2 应用界面的构建层次	59
4.1.3 视图分类	60
4.2 标签控件和按钮控件	60
4.2.1 标签控件	61
4.2.2 按钮控件	61
4.2.3 动作和输出口	63
4.3 TextField 控件和 TextView 控件	66
4.3.1 TextField 控件	66
4.3.2 TextView 控件	67
4.3.3 键盘的打开和关闭	67
4.3.4 关闭和打开键盘的通知	68
4.3.5 键盘的种类	69
4.4 开关控件、滑块控件和分段控件	70
4.4.1 开关控件	71

4.4.2 滑块控件	71	5.3 分节表视图	138
4.4.3 分段控件	71	5.3.1 添加索引	138
4.5 网页控件 WebView	72	5.3.2 分组与静态表	141
4.5.1 WebView 介绍	72	5.4 修改单元格	144
4.5.2 使用 WebView 构建 Hybrid 应用	75	5.4.1 删除和插入单元格	144
4.6 屏幕滚动控件 ScrollView	80	5.4.2 移动单元格	150
4.6.1 ScrollView 属性的设置	80	5.5 表视图 UI 设计模式	152
4.6.2 键盘与其他控件的协同	84	5.5.1 分页模式	152
4.7 等待相关的控件与进度条	86	5.5.2 下拉刷新模式	152
4.7.1 活动指示器 ActivityIndicatorView	87	5.5.3 iOS 6 下拉刷新控件	153
4.7.2 进度条 ProgressView	88	5.6 小结	155
4.8 警告框和操作表	88	第 6 章 视图控制器与导航模式	156
4.8.1 警告框 UIAlertView	89	6.1 概述	156
4.8.2 操作表 ActionSheet	90	6.1.1 视图控制器的种类	156
4.9 工具栏和导航栏	92	6.1.2 导航模式	156
4.9.1 工具栏	92	6.1.3 模态视图	157
4.9.2 导航栏	94	6.2 平铺导航	163
4.10 屏幕布局	98	6.2.1 应用场景	163
4.10.1 iPad 与 iPhone 屏幕布局	98	6.2.2 基于分屏导航的实现	165
4.10.2 绝对布局和相对布局	99	6.2.3 基于分页导航的实现	168
4.10.3 使用 AutoLayout 布局	102	6.3 标签导航	173
4.10.4 旋转你的屏幕	103	6.3.1 应用场景	174
4.11 选择器	108	6.3.2 nib 实现	174
4.11.1 日期选择器	108	6.3.3 故事板实现	179
4.11.2 普通选择器	110	6.4 树形结构导航	182
4.11.3 数据源协议与委托协议	112	6.4.1 应用场景	182
4.12 iOS 6 中的集合视图	113	6.4.2 nib 实现	183
4.12.1 集合视图介绍	114	6.4.3 故事板实现	189
4.12.2 集合视图单元格	116	6.5 组合使用导航模式	193
4.12.3 数据源协议与委托协议	118	6.5.1 应用场景	193
4.13 小结	119	6.5.2 故事板实现	194
第 5 章 表视图	120	6.6 小结	200
5.1 概述	120	第 7 章 iPhone 与 iPad 应用开发的差异	201
5.1.1 表视图的组成	120	7.1 概述	201
5.1.2 表视图的相关类	121	7.1.1 应用场景差异	201
5.1.3 表视图分类	122	7.1.2 设计和开发需注意的问题	201
5.1.4 单元格的组成和样式	123	7.1.3 构建自适应的 iPhone 和 iPad 工程	204
5.1.5 数据源协议与委托协议	124	7.2 iPad 专用 API	206
5.2 简单表视图	125	7.2.1 UIPopoverController 控制器	206
5.2.1 创建简单表视图	125	7.2.2 UISplitViewController 控制器	209
5.2.2 自定义单元格	129	7.2.3 模态视图专用属性	215
5.2.3 添加搜索栏	133	7.3 Master-Detail 应用程序模板	219

7.3.1 nib 实现	220	10.2 属性列表	275
7.3.2 故事板实现	225	10.3 对象归档	280
7.4 Utility 应用程序模板	228	10.4 使用 SQLite 数据库	285
7.4.1 nib 实现	229	10.4.1 SQLite 数据类型	285
7.4.2 故事板实现	232	10.4.2 创建数据库	285
7.5 移动平台的分层架构设计	234	10.4.3 查询数据	287
7.5.1 低耦合企业级系统架构设计	234	10.4.4 修改数据	290
7.5.2 移动平台的分层架构设计	235	10.5 Core Data	292
7.5.3 基于同一工程的分层	235	10.5.1 ORM	292
7.5.4 基于一个工作空间不同工程的分层	241	10.5.2 Core Data 堆栈	293
7.6 小结	244	10.5.3 建模和生成实体	296
第 8 章 应用程序设置	245	10.5.4 采用 Core Data 分层架构设计	299
8.1 概述	245	10.5.5 查询数据	301
8.1.1 设置	245	10.5.6 修改数据	302
8.1.2 配置	246	10.6 小结	304
8.2 应用程序设置包	247	第 11 章 访问通讯录	305
8.3 设置项目种类	248	11.1 概述	305
8.3.1 文本字段	251	11.2 读取联系人信息	306
8.3.2 开关	253	11.2.1 查询联系人记录	307
8.3.3 滑块	254	11.2.2 读取单值属性	309
8.3.4 值列表	256	11.2.3 读取多值属性	311
8.3.5 子界面	257	11.2.4 读取图片属性	313
8.4 读取设置	259	11.3 写入联系人信息	313
8.5 小结	260	11.3.1 创建联系人	315
第 9 章 应用程序本地化	261	11.3.2 修改联系人	317
9.1 概述	261	11.3.3 删除联系人	318
9.1.1 本地化内容	261	11.4 高级 API	319
9.1.2 本地化目录结构	263	11.4.1 选择联系人	319
9.2 文本信息本地化	263	11.4.2 显示和修改联系人	322
9.2.1 系统按钮和信息本地化	263	11.4.3 创建联系人	324
9.2.2 应用名称本地化	265	11.5 小结	328
9.2.3 程序代码输出的静态文本本地化	266		
9.2.4 使用 genstring 工具	268	第二部分 网络篇	
9.3 nib 和故事板文件本地化	268	第 12 章 访问 Web Service	330
9.3.1 添加本地化	269	12.1 概述	330
9.3.2 开关使用 ibtool 工具	270	12.2 数据交换格式	330
9.4 资源文件本地化	271	12.2.1 XML 文档结构	332
9.5 小结	273	12.2.2 解析 XML 文档	333
第 10 章 数据持久化	274	12.2.3 JSON 文档结构	340
10.1 概述	274	12.2.4 JSON 数据解码	341
10.1.1 沙箱目录	274	12.3 REST Web Service	343
10.1.2 持久化方式	275	12.3.1 HTTP 和 HTTPS 协议	343

12.3.2	同步 GET 请求方法	344	14.2.1	横幅广告	396
12.3.3	异步 GET 请求方法	348	14.2.2	插页广告	401
12.3.4	POST 请求方式	349	14.2.3	查看你的收入	406
12.3.5	调用 REST Web Service 的插入、 修改和删除方法	350	14.3	使用谷歌 AdMob 广告	408
12.4	使用 ASIHTTPRequest 框架	355	14.3.1	注册 AdMob 账号和管理应用	408
12.4.1	安装和配置 ASIHTTPRequest 框架	355	14.3.2	下载谷歌 AdMob Ads SDK 和示 例代码	412
12.4.2	同步请求	356	14.3.3	添加 AdMob 横幅广告	414
12.4.3	异步请求	358	14.3.4	添加 AdMob 插页广告	418
12.4.4	使用请求队列	359	14.3.5	为广告提交用户和位置信息	421
12.4.5	上传数据	362	14.3.6	搜索广告	422
12.5	反馈网络信息改善用户体验	364	14.3.7	查看你的收入	424
12.5.1	iOS 6 表视图刷新控件的使用	364	14.4	应用内购买	425
12.5.2	使用等待指示器控件	367	14.4.1	概述	425
12.5.3	使用网络等待指示器	369	14.4.2	测试环境搭建	426
12.6	小结	370	14.4.3	在程序中实现应用内购买	431
第 13 章	定位服务与地图应用	371	14.4.4	测试应用内购买	437
13.1	定位服务	371	14.5	小结	439
13.1.1	定位服务编程	371	第 15 章	找出程序中的 bug——调试	440
13.1.2	地理信息反编码	376	15.1	Xcode 调试工具	440
13.1.3	地理信息编码查询	377	15.1.1	定位编译错误	440
13.1.4	关于定位服务的测试	379	15.1.2	查看和显示日志	441
13.2	使用 iOS 6 苹果地图	382	15.1.3	设置和查看断点	442
13.2.1	显示地图	382	15.1.4	调试工具栏	446
13.2.2	添加标注	384	15.1.5	输出窗口	447
13.2.3	跟踪用户位置变化	387	15.1.6	变量查看窗口	448
13.3	使用程序外地图	388	15.1.7	查看线程	449
13.3.1	调用 iOS 6 苹果地图	388	15.2	日志与断言输出	450
13.3.2	调用谷歌 Web 地图	391	15.2.1	使用 NSLog 函数	450
13.4	小结	392	15.2.2	使用 NSAssert 宏	451
			15.2.3	移除 NSLog 和 NSAssert	452
			15.3	LLDB 调试工具	455
			15.3.1	断点命令	455
			15.3.2	观察点命令	457
			15.3.3	查看变量和计算表达式命令	458
			15.4	异常堆栈报告分析	461
			15.4.1	跟踪异常堆栈	461
			15.4.2	分析堆栈报告	463
			15.5	在 iOS 设备上调试	464
			15.5.1	创建开发者证书	465
			15.5.2	设备注册	468
			15.5.3	创建 App ID	470
			15.5.4	创建配置概要文件	471
			15.5.5	设备调试	473

第三部分 进阶篇

第 14 章	iOS 中的商业模式	394
14.1	收费策略	394
14.1.1	iOS 如何赚钱	394
14.1.2	避免定价策略误区	395
14.1.3	免费软件的艺术	395
14.1.4	在适当的时间、适当的地点 植入广告	395
14.1.5	尝试不同的盈利模式	395
14.2	使用苹果 iAd 广告	396

15.6	Xcode 设备管理工具	474	17.4.1	使用文件	546
15.6.1	管理设备配置概要文件	474	17.4.2	使用 SQLite 数据库	549
15.6.2	查看设备上的应用程序	475	17.4.3	使用 Core Data	550
15.6.3	设备控制台	477	17.5	可重用对象的使用	552
15.6.4	设备日志	477	17.5.1	表视图中的可重用对象	553
15.7	小结	479	17.5.2	集合视图中的可重用对象	554
第 16 章	基于测试驱动的 iOS 开发	480	17.5.3	地图视图中的可重用对象	555
16.1	测试驱动的软件开发概述	480	17.6	并发处理与多核 CPU	556
16.1.1	测试驱动的软件开发流程	480	17.6.1	主线程阻塞问题	556
16.1.2	测试驱动的软件开发案例	481	17.6.2	选择 NSThread 还是 GCD	557
16.1.3	iOS 单元测试框架	486	17.7	编译器和编译参数	558
16.2	使用 OCMock 测试框架	486	17.7.1	GCC、LLVM GCC 与 Apple LLVM 比较	558
16.2.1	添加 OCMock 到工程中	486	17.7.2	ARM 架构	559
16.2.2	应用测试和逻辑测试	489	17.7.3	Optimization Level	561
16.2.3	编写 OCMock 测试方法	489	17.8	小结	562
16.2.4	分析测试报告	494	第 18 章	管理好你的程序代码——代码版本 控制	563
16.3	使用 GHUnit 测试框架	495	18.1	概述	563
16.3.1	添加 GHUnit 到工程	496	18.1.1	版本控制历史	563
16.3.2	编写 GHUnit 测试用例	498	18.1.2	基本概念	564
16.3.3	分析测试报告	500	18.2	Git 代码版本控制	564
16.4	使用伪对象	502	18.2.1	服务器搭建	564
16.4.1	添加 OCMock 到工程	503	18.2.2	Gitolite 服务器管理	566
16.4.2	使用 OCMock 对象	505	18.2.3	Git 常用命令	568
16.5	iOS 单元测试最佳实践	507	18.2.4	Git 分支	570
16.5.1	iOS 单元测试策略	507	18.2.5	Git 协同开发	574
16.5.2	测试数据持久层	507	18.2.6	Xcode 中 Git 的配置与使用	576
16.5.3	测试业务逻辑层	512	18.3	GitHub 代码托管服务	583
16.5.4	测试表示层	515	18.3.1	创建和配置 GitHub 账号	584
16.6	小结	522	18.3.2	创建代码库	586
第 17 章	让你的程序“飞”起来——性能 优化	523	18.3.3	派生代码库	589
17.1	内存优化	523	18.3.4	使用 GitHub 协同开发	591
17.1.1	内存泄漏问题的解决	523	18.3.5	管理组织	596
17.1.2	查找和解决僵尸对象	531	18.4	小结	599
17.1.3	autorelease 的使用问题	534	第 19 章	把你的应用放到 App Store 上	600
17.1.4	响应内存警告	534	19.1	收官	600
17.1.5	选择 nib 还是故事板	536	19.1.1	添加图标	600
17.2	优化资源文件	537	19.1.2	添加启动界面	602
17.2.1	图片文件优化	537	19.1.3	调整 Application Target 属性	604
17.2.2	音频文件优化	538	19.1.4	为发布进行编译	605
17.3	延迟加载	539	19.1.5	应用打包	609
17.3.1	资源文件的延迟加载	540	19.2	发布流程	610
17.3.2	故事板和 nib 文件的延迟加载	543	19.2.1	创建应用及基本信息	611
17.4	数据持久化的优化	546			

19.2.2	应用定价信息	612
19.2.3	最后信息输入	613
19.2.4	上传应用	616
19.3	常见审核不通过的原因	618
19.4	小结	619

第四部分 实战篇

第 20 章 重构 MyNotes 应用——iOS 网络

通信中的设计模式与架构设计

20.1	移动网络通信应用的分层架构设计	622
20.2	基于委托模式实现	623
20.2.1	网络通信与委托模式	623
20.2.2	在异步网络通信中使用委托模式实现分层架构设计	623
20.2.3	类图	624
20.2.4	时序图	626
20.2.5	数据持久层的代码实现	629
20.2.6	业务逻辑层的代码实现	631
20.2.7	表示层的代码实现	632
20.3	基于观察者模式的通知机制实现	637
20.3.1	观察者模式的通知机制回顾	637
20.3.2	异步网络通信中通知机制的分层架构设计	638
20.3.3	类图	638
20.3.4	时序图	639
20.3.5	数据持久层的代码实现	642
20.3.6	业务逻辑层的代码实现	643
20.3.7	表示层的代码实现	644
20.4	小结	649

第 21 章 iOS 敏捷开发项目实战——

2016 里约热内卢奥运会应用

开发及 App Store 发布

21.1	应用分析与设计	650
21.1.1	应用概述	650
21.1.2	需求分析	650
21.1.3	原型设计	651
21.1.4	数据库设计	652
21.1.5	架构设计	652
21.2	iOS 敏捷开发	653
21.2.1	敏捷开发宣言	653
21.2.2	iOS 适合敏捷开发?	654
21.2.3	iOS 敏捷开发最佳实践	654

21.3	任务 1: 创建应用基本工作空间	656
21.4	任务 2: 信息系统层与持久层开发	657
21.4.1	迭代 2.1: 编写数据库 DDL 脚本	657
21.4.2	迭代 2.2: 插入初始数据到数据库	658
21.4.3	迭代 2.3: 编写实体类	658
21.4.4	迭代 2.4: DAO 类 GUnit 单元测试	659
21.4.5	迭代 2.5: 编写 DAO 类	663
21.4.6	迭代 2.6: 发布到 GitHub	666
21.5	任务 3: 业务逻辑层开发	667
21.5.1	迭代 3.1: 比赛项目业务逻辑类 GUnit 单元测试	667
21.5.2	迭代 3.2: 编写比赛项目业务逻辑类	668
21.5.3	迭代 3.3: 比赛日程业务逻辑类 GUnit 单元测试	669
21.5.4	迭代 3.4: 编写比赛日程业务逻辑类	670
21.5.5	迭代 3.5: 发布到 GitHub	672
21.6	任务 4: 表示层开发	672
21.6.1	迭代 4.1: 根据原型设计初步设计 iPad 故事板	673
21.6.2	迭代 4.2: 根据原型设计初步设计 iPhone 故事板	674
21.6.3	迭代 4.3: 首页模块	674
21.6.4	迭代 4.4: 比赛项目模块	675
21.6.5	迭代 4.5: 比赛日程模块	679
21.6.6	迭代 4.6: 倒计时模块表示层	681
21.6.7	迭代 4.7: 关于我们模块表示层	683
21.6.8	迭代 4.8: 发布到 GitHub	683
21.7	任务 5: 收工	684
21.7.1	迭代 5.1: 添加图标	684
21.7.2	迭代 5.2: 设计和添加启动界面	684
21.7.3	迭代 5.3: 植入谷歌 AdMob 横幅广告	685
21.7.4	迭代 5.4: 性能测试与改善	686
21.7.5	迭代 5.5: 发布到 GitHub	687
21.7.6	迭代 5.6: 在 App Store 上发布应用	687
21.8	小结	690

Part 1

第一部分

基础篇

本 部 分 内 容

- 第 1 章 开篇综述
- 第 2 章 第一个 iOS 应用程序
- 第 3 章 iOS 常用设计模式
- 第 4 章 UIView 与控件
- 第 5 章 表视图
- 第 6 章 视图控制器与导航模式
- 第 7 章 iPhone 与 iPad 应用开发的差异
- 第 8 章 应用程序设置
- 第 9 章 应用程序本地化
- 第 10 章 数据持久化
- 第 11 章 访问通讯录



自从App Store上线以来，它创造了很多神话，给我们这些程序员提供了展示自己的舞台，给了我们创意的空间，给了我们创业的机会。下面让我们从这里开始iOS开发之旅吧。

1.1 iOS 概述

在本节中，我们将了解什么是iOS以及iOS 6有哪些新特性。

1.1.1 iOS介绍

iOS的系统架构分为4层——Cocoa Touch层、Media层、Core Services层和Core OS层，相关内容可参见2.6.1节。下面我们简要介绍一下iOS的一些功能，具体如下所示。

- ❑ **多点触摸和手势。**触摸功能在iOS设备之前就被采用，但基本都是单点触摸，即只能用一个手指，而iOS设备能够感应多个手指的触摸。为了配合这种多点触摸，iOS上的触摸分为多种手势：触击、双击、滑动、长期间触击、轻拂、刷屏和手指合拢张开等。
- ❑ **统一的屏幕尺寸。**目前，iOS屏幕尺寸有4套：iPhone和iPod touch是3.5英寸，iPhone 5和第5代iPod touch是4英寸，iPad是9.7英寸，iPad mini是7.9英寸。统一的屏幕尺寸给应用软件开发带来很多好处，开发人员可以不用关心屏幕尺寸适配的问题，从而把精力集中在其他方面。
- ❑ **高分辨率。**iPhone 4S的屏幕分辨率是960 × 640，iPhone 5和第5代iPod touch的屏幕分辨率是1136 × 640，第1、2代iPad的屏幕分辨率是1024 × 768，第3代iPad的屏幕分辨率是2048 × 1536，而iPad mini的屏幕分辨率是1024 × 768。
- ❑ **重力加速计。**iOS内置了重力加速计。有了重力加速计，用户能够玩很多有意思的游戏（如极品飞车，它可以把iPhone作为方向盘，通过重力加速计感应方向的变化）。此外，还有很多与重力加速计有关的应用软件，如水平尺应用等。
- ❑ **指南针。**iOS内置了指南针设备。很多应用基于指南针，例如导航软件和地图应用软件。
- ❑ **蓝牙和Wi-Fi连接。**iOS内置了蓝牙和Wi-Fi通信模块。iOS设备之间可以采用Wi-Fi互相连接，也可以采用蓝牙进行连接，很多基于局域网的游戏就是通过这个功能实现的。当然，也可以通过Wi-Fi上网，这可以节约用户的上网费用。此外，iOS还可以与电脑连接。

1.1.2 iOS 6 新特性

iOS的最新版本为iOS 6.0。苹果公司于2012年9月20日凌晨1点开放其正式版的下载，它支持iPhone 3GS、iPhone 4、iPhone 4S、iPhone 5、iPad 2、iPad 3、iPod touch 4和iPod touch 5等设备。据苹果发布的更新文档显示，iOS 6新增了200多项功能，更好地支持了中国市场，很多新特性或将成为将来的焦点。

现在我们先简要介绍一下iOS 6几个重要的新性能。

- ❑ **地图**。苹果放弃了以前使用的谷歌地图，转而使用苹果自己的地图。MapKit框架本身没有变化，只是地图内容不再是谷歌地图。
- ❑ **社交网络（social network）**。iOS 5集成了Twitter，iOS 6新集成了Facebook和新浪微博，还提供了一个新的视图控制器UIActivityViewController。使用这个视图控制器，可以非常方便地发送短消息和邮件，而且将内容复制到剪贴板之后，还可以利用视图控制器发消息到Twitter、Facebook以及新浪微博。
- ❑ **Pass Kit**。它是开发Passbook的API。Passbook可以集中管理各种优惠券、打折卡、登机牌，这样以后就不必在钱包中放一堆卡了。个人认为该新功能在未来电子商务的发展中会有不容小视的贡献。
- ❑ **游戏中心（Game Center）**。Game Center是苹果在发布iOS 4时引入的API，目的是开发基于苹果Game Center的游戏。iOS 6对Game Center的API进行了升级。
- ❑ **提醒（Reminder）**。在iOS 6中，苹果开放了访问设备中的提醒API，开发者可以通过Event Kit读写提醒信息。
- ❑ **应用内购买（In-App Purchase，缩写为IAP）**。使用这个功能，可以在应用中购买付费道具，增加新功能，订阅杂志，也可以购买和下载iTunes Store上的音乐、电影和图书。
- ❑ **集合视图（Collection View）**。它是一种增强网格视图，虽然这种网格视图在开源社区中已经有开源代码，但是使用起来不是很方便，而有了iOS 6的集合视图API后，使用起来变得非常方便。
- ❑ **界面状态保持（UI State Preservation）**。iOS 6之前，在应用退出，进入后台并且被终止的情况下，如果需要保持界面中UI控件的状态，则需要通过自己编写代码读写数据来实现，而在iOS 6之后这些工作就变得非常简单了。
- ❑ **自动布局（Auto Layout）**。这个功能首先应用于Mac OS X 10.7下的开发，现在也可以在iOS 6中使用，它为视图布局定义一套约束，而约束定义了两个界面中视图之间的关系。
- ❑ **数据隐私（Data Privacy）**。为了防止隐私泄漏，在iOS 6中访问联系人、日历、提醒和照片库时，需要经过用户允许。以前只有访问位置信息时，才需要经过用户允许。

此外，iOS 6还对已有的框架进行了不同程度的增强和删减。

1.2 开发环境及开发工具

苹果公司于2008年3月6日发布了iPhone和iPod touch的应用程序开发包，其中包括Xcode开发工具、iPhone SDK和iPhone手机模拟器。第一个Beta版本是iPhone SDK 1.2b1（build 5A147p），发布后立即就能使用，但是同时推出的App Store所需要的固件更新直到2008年7月11日才发布。编写本书时，iOS SDK 6.1.2版本已经发布。

iOS开发工具主要是Xcode。自从Xcode 3.1发布以后，Xcode就成为iPhone软件开发工具包的开发环境。Xcode可以开发Mac OS X和iOS应用程序，其版本是与SDK相互对应的。例如，Xcode 3.2.5与iOS SDK 4.2对应，Xcode 4.1与iOS SDK 4.3对应，Xcode 4.2与iOS SDK 5对应，Xcode 4.5与iOS SDK 6对应。

在Xcode 4.1之前，还有一个配套使用的工具Interface Builder，它是Xcode套件的一部分，用来设计窗体和视图，通过它可以“所见即所得”地拖曳控件并定义事件等，其数据以XML的形式被存储在xib文件中。在Xcode 4.1之后，Interface Builder成为了Xcode的一部分，与Xcode集成在一起。

打开Xcode 4.5工具，看到的主界面如图1-1所示。

该界面主要分成3个区域，①号区域是工具栏，其中的按钮可以完成大部分工作。②号区域是导航栏，主要是对工作空间中的内容进行导航。在导航栏上面还有一排按钮，如图1-2所示，默认选中的是“文件”导航面板。关于各按钮的具体用法，我们会在以后用到的时候详细介绍。

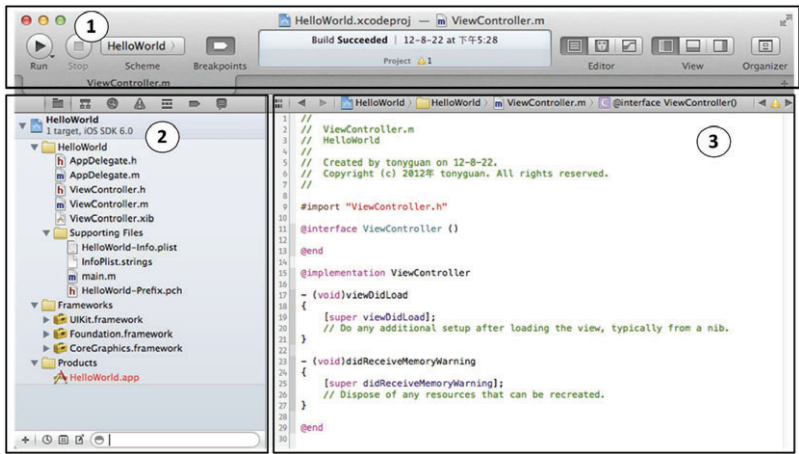


图1-1 Xcode主界面



图1-2 Xcode导航面板

在选中“文件”导航面板时，导航栏下面也有一排按钮，如图1-3所示。这是辅助按钮，它们的功能如图1-3所标注。需要注意的是，对于不同的导航面板，这些按钮是不同的。

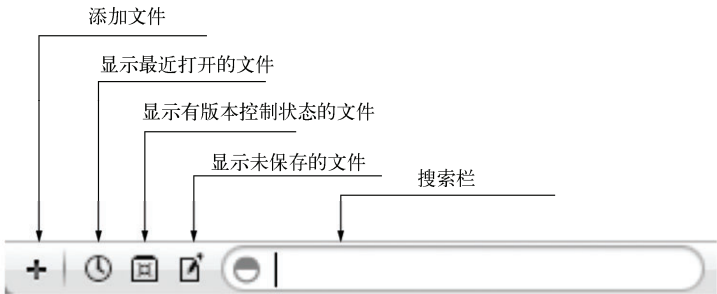


图1-3 “文件”导航面板的辅助按钮

图1-1所示的③号区域是代码编辑区，我们的编码工作就是在这里完成的。其背景颜色可以在Xcode的使用偏好（该功能可以在图1-4所示菜单栏中的苹果图标中找到）中设置。



图1-4 Xcode工具菜单栏

1.3 本书中的约定

为了方便大家阅读本书，本节介绍一下书中案例代码和图示的相关约定。

1.3.1 案例代码约定

作为一本编程方面的书，书中有很多案例代码，我们可以从图灵网站（www.ituring.com.cn）本书主页免费注册

册下载或者从智捷教育提供的本书服务网站（www.iosbook1.com/code.html）下载，解压后会看到如图1-5所示的目录结构。

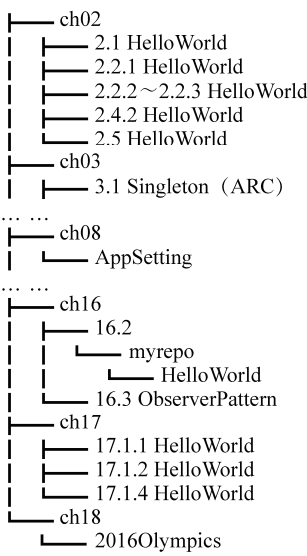


图1-5 源代码文件目录

ch02 ~ ch18代表第2章到第18章的案例代码或一些资源文件，其中工程或工作空间的命名有如下几种形式。

- ❑ 二级目录标号，如“2.1 HelloWorld”说明是2.1节中使用的HelloWorld工程（或工作空间）案例。
- ❑ 三级目录标号，如“2.2.1 HelloWorld”说明是2.2.1节中使用的HelloWorld工程（或工作空间）案例。
- ❑ 有~的情况，如“2.2.2 ~ 2.2.3 HelloWorld”说明是2.2.2节到2.2.3节共同使用的HelloWorld工程（或工作空间）案例。
- ❑ 对于没有标号的情况，由其所在的父目录说明是哪个章节的案例工程（或工作空间），如“2016Olympic”说明是在第18章中使用的。

1.3.2 图示的约定

为了更形象有效地说明知识点或描述操作，本书添加了很多图示，下面简要说明图示中一些符号的含义。

- ❑ 图中的圈框。有时读者会看到如图1-6所示的圈框，其中的内容是选中的内容或重点要说明的内容。

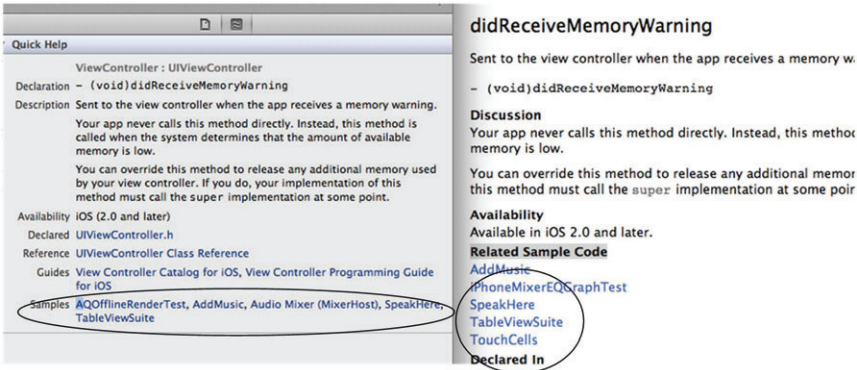


图1-6 图中圈框

- 图中的箭头。如图1-7和图1-8所示，箭头用于说明用户的动作，一般箭尾是动作开始的地方，箭头指向动作结束的地方。图1-8所示的虚线箭头在书中用得比较多，常用来描述设置控件的属性等操作，箭头指向代表打开XXX检测器。

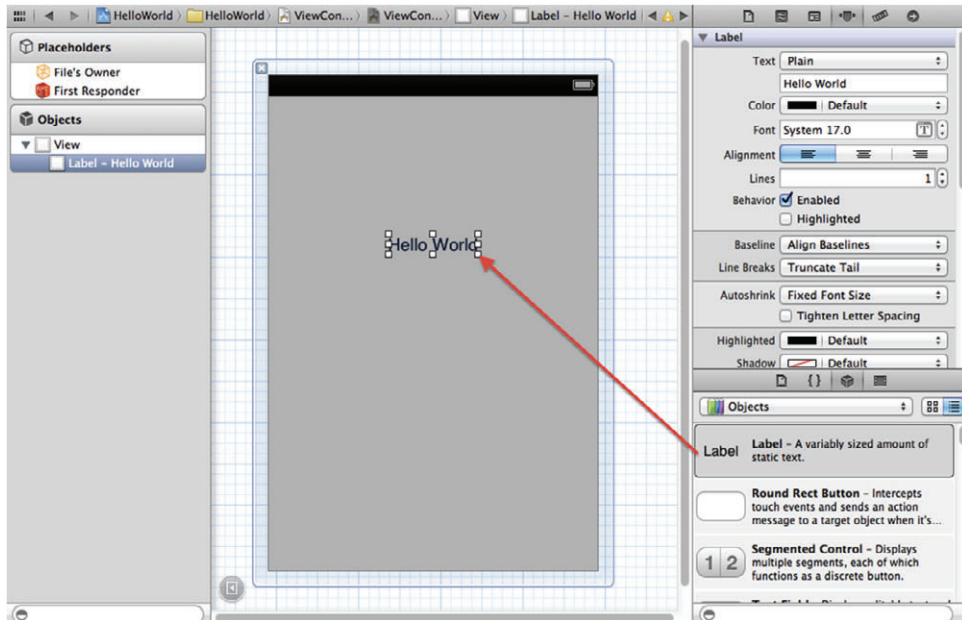


图1-7 图中箭头1

- 图中手势。为了描述操作，我们在图中放置了👉等手势符号，这说明点击了该处的按钮。如图1-9所示，屏幕下方的“更多...”按钮上面就有这个手势，说明用户点击了“更多...”按钮。

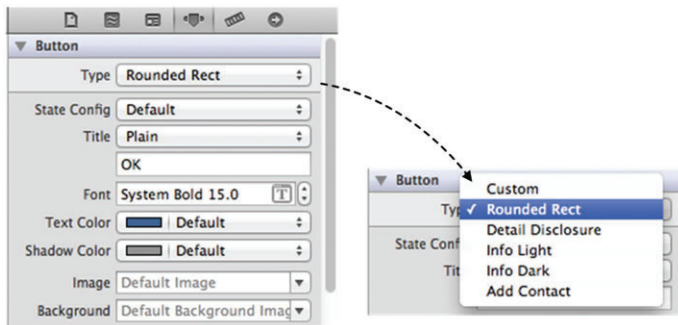


图1-8 图中箭头2

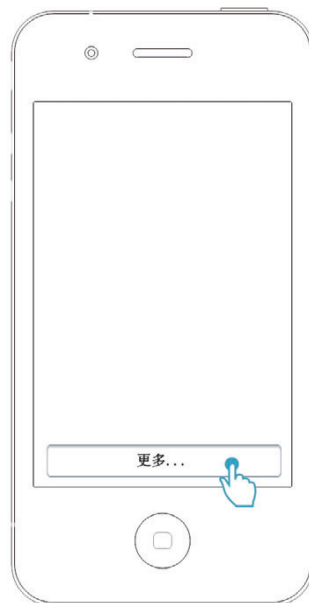


图1-9 图中手势

从控制台输出Hello World是我学习C语言的第一步，也是我人生中非常重要的一步。多年后的今天，我希望仍以HelloWorld作为第一步，为大家开启一个神奇、瑰丽的世界——iOS。

本章以HelloWorld作为切入点，向大家系统介绍什么是iOS应用以及如何使用Xcode创建iOS应用。

2.1 创建基于 nib 的 HelloWorld 工程

在学习之初，我们有必要对使用Xcode创建iOS工程做一个整体概览，这里我们通过创建一个基于nib的HelloWorld iPhone应用来详述其中涉及的知识点。

实现HelloWorld应用后，会在界面上展示字符串Hello World（效果如图2-1所示），其中主要包含Label（标签）控件。



图2-1 HelloWorld的iPhone界面

2.1.1 创建工程

启动Xcode，然后点击File→New→Project菜单，在打开的Choose a template for your new project界面中选择Single View Application工程模板（如图2-2所示）。

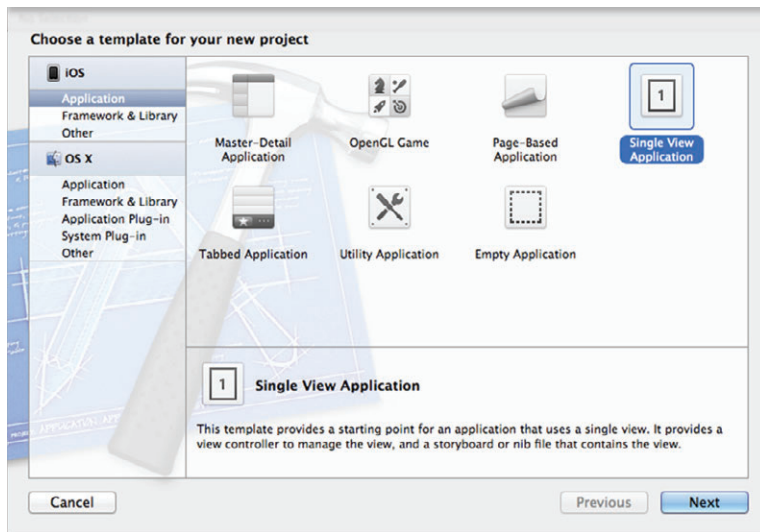


图2-2 选择工程模板

接着点击Next按钮，随即出现图2-3所示的界面。

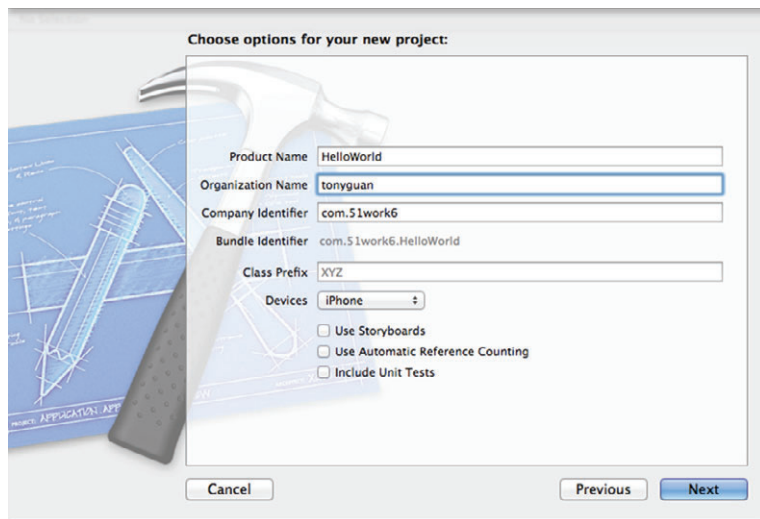


图2-3 新工程中的选项

这里我们可以按照提示并结合自己的实际情况和需要输入相关内容。下面简要说明图2-3中的选项。

- ❑ **Product Name**。工程名字。
- ❑ **Organization Name**。组织名字。
- ❑ **Company Identifier**。公司标识（很重要）。一般情况下，这里输入的是公司的域名（如com.51work6），这类似于Java中的包命名。
- ❑ **Bundle Identifier**。捆绑标识符（很重要）。该标识符由Product Name+ Company Identifier构成。因为在App Store发布应用的时候会用到它，所以它的命名不可重复。
- ❑ **Class Prefix**。类的前缀。为生成的类加前缀（如XYZViewController）。

- ❑ **Devices**。选择设备。可以构建基于iPhone或iPad的工程，也可以构建通用工程。通用工程是指一个工程在iPhone和iPad上都可以正常运行。
- ❑ **Use Storyboards**。工程是否采用故事板技术。
- ❑ **Use Automatic Reference Counting**。工程是否采用ARC（自动引用计数）技术。
- ❑ **Include Unit Tests**。是否产生单元测试相关的类。

设置完相关的工程选项后，点击Next按钮，进入下一级界面。根据提示选择存放文件的位置，然后点击Create按钮，将出现如图2-4所示的界面。

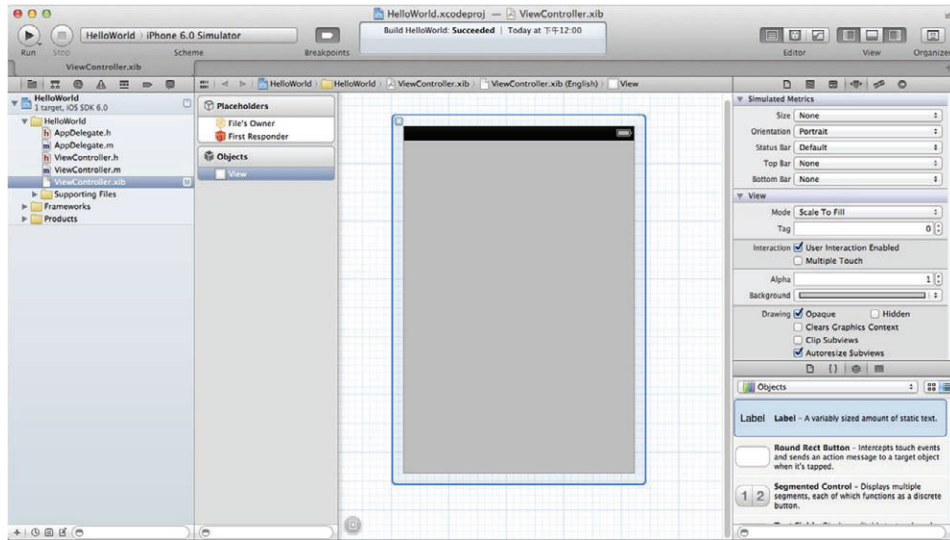


图2-4 新创建的工程

在右下角的对象库中选择Label控件，将其拖曳到View上并调整其位置。双击Label控件，使其处于编辑状态（也可以通过控件的属性来设置），在其中输入Hello World，如图2-5所示。

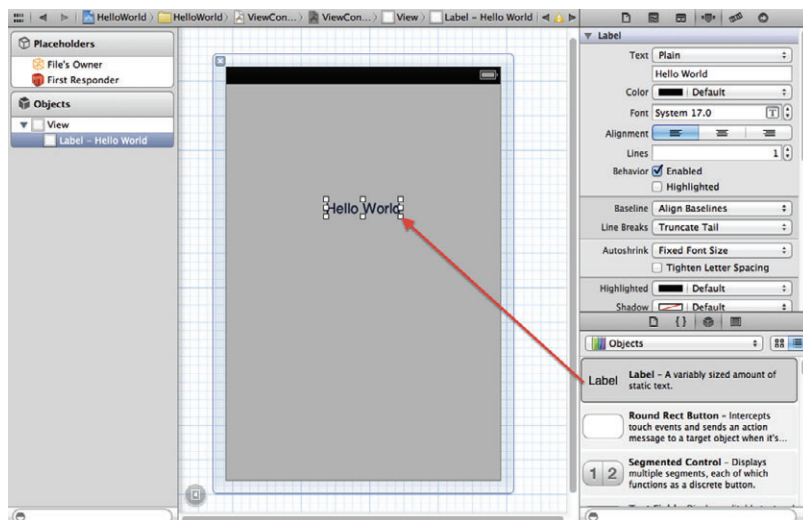



图2-5 添加Label控件

至此，整个工程创建完毕。

点击图2-4左上角的Run按钮，即可看到运行结果。

我们在没有输入任何代码的情况下，就已经利用Xcode工具的Single View Application模板创建了一个工程，并成功运行，Xcode之强大可见一斑。

2.1.2 Xcode中的iOS工程模板

从图2-2中可以看出，iOS工程模板分为3类——Application、Framework & Library和Other，下面将分别详细介绍这3类模板。

1. Application类型

我们大部分的开发工作都是从使用Application类型模板创建iOS程序开始的。该类型共包含7个模板，具体如下所示。

- ☐ Master-Detail Application。可以构建树形结构导航模式应用，生成的代码中包含了导航控制器和表视图控制器等。
- ☐ OpenGL Game。可以构建基于OpenGL ES的游戏应用。
- ☐ Page-Based Application。可以构建类似于电子书效果的应用，这是一种平铺导航。
- ☐ Single View Application。可以构建简单的单个视图应用。
- ☐ Tabbed Application。可以构建标签导航模式的应用，生成的代码中包含了标签控制器和标签栏等。
- ☐ Utility Application。可以构建实用型应用程序，它会生成两个视图控制器——主视图控制器和子视图控制器。在iPhone中子视图以模态方式呈现，在iPad中子视图以浮动窗口（popover）的形式呈现。
- ☐ Empty Application。可以构建一个空应用程序，需要我们自己添加视图等对象。该模板很少使用。

2. Framework & Library类型

Framework & Library类型的模板如图2-6所示，它可以构建基于Cocoa Touch的静态库。

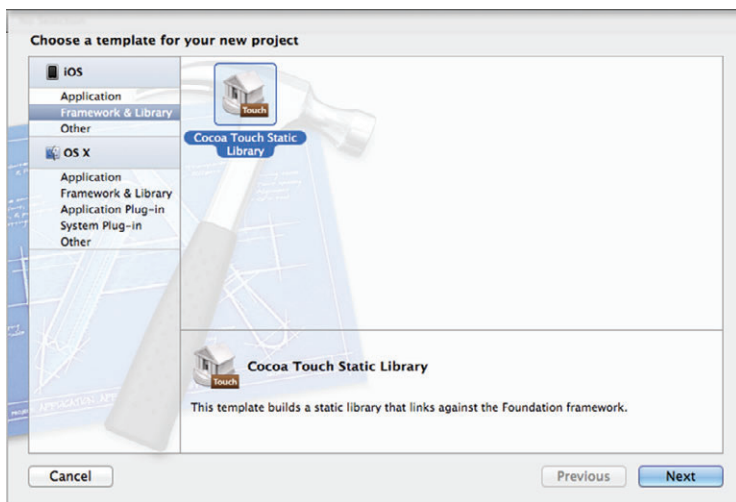


图2-6 Framework & Library类型模板

说明 出于代码安全和多个工程重用代码的考虑，我们需要将一些类或者函数编写成静态库。静态库不能独立运行，编译成功时会生成名为libXXX.a的文件（例如libHelloWorld.a）。

3. Other类型

利用该类型，我们可以构建应用的内置付费内容包（In-App Purchase Content）和空工程，如图2-7所示。使用内置付费内容包，可以帮助我们构建具有内置收费功能的应用。

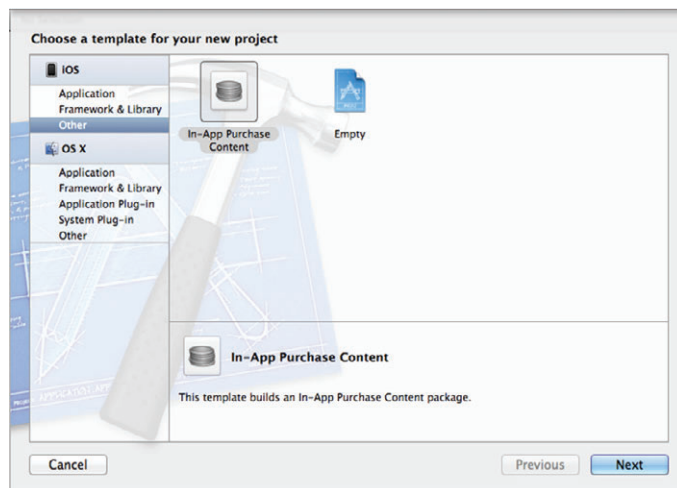


图2-7 Other类型模板

我们可以根据需要选用不同的工程模板，这可以大大减少我们的工作量。

2.1.3 应用剖析

在创建HelloWorld的过程中，生成了很多文件（展开Xcode左边的项目导航视图可以看到，如图2-8所示），它们各自的作用是什么？彼此间又是怎样的一种关系呢？

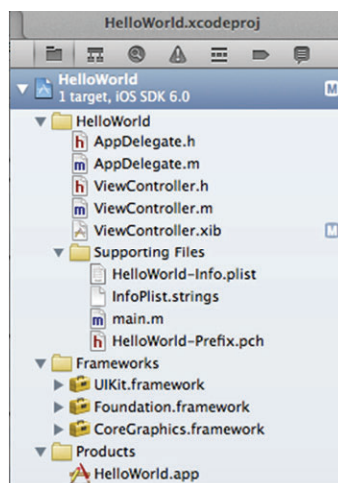


图2-8 项目导航视图

如图2-8所示，导航视图下有HelloWorld、Frameworks和Products三个组。

在HelloWorld组中共有两个类：AppDelegate和ViewController，以及一个组Supporting Files。我们主要的编码工作就是在AppDelegate和ViewController这两个类中进行的，它们的类图如图2-9所示。

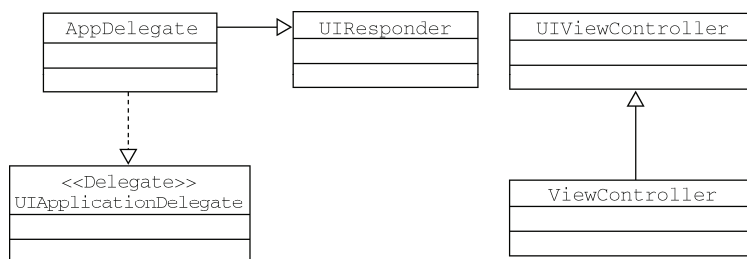


图2-9 HelloWorld工程中的类图

AppDelegate是应用程序委托对象，它继承了UIResponder类，并实现了UIApplicationDelegate委托协议。UIResponder类可以使子类AppDelegate具有处理相应事件的能力，而UIApplicationDelegate委托协议使AppDelegate能够成为应用程序委托对象，这种对象能够响应应用程序的生命周期。相应地，AppDelegate的子类也可以实现这两个功能。

ViewController类继承自UIViewController类，它是视图控制器类，在工程中扮演着根视图和用户事件控制类的角色。需要特别指出的是，ViewController.xib文件也是视图控制文件，起描述作用，与ViewController配套存在。

AppDelegate和ViewController类与main代码模块的主函数存在一种直接的调用关系，下面我们借助UML时序图来进行详细说明，如图2-10所示。

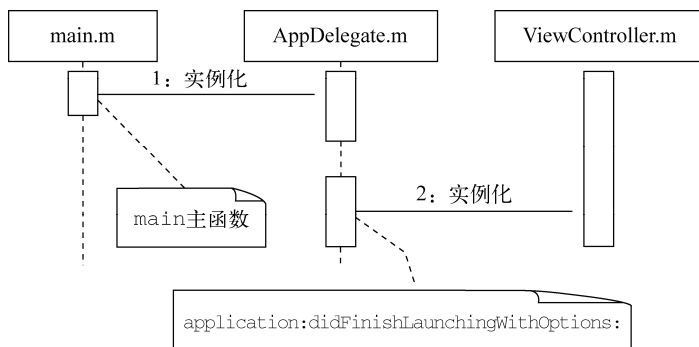


图2-10 HelloWorld启动时序图

可以看到，在HelloWorld启动过程中，首先调用main.m代码模块的main()主函数进行AppDelegate的实例化，具体代码如下所示：

```

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

AppDelegate类是应用程序委托对象，这个类中继承的一系列方法在应用生命周期的不同阶段会被回调。启动HelloWorld时，首先会调用application:didFinishLaunchingWithOptions:方法，该方法的代码如下：

```

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

```
self.viewController = [[[ViewController alloc]
    initWithNibName:@"ViewController" bundle:nil] autorelease];
self.window.rootViewController = self.viewController;
[self.window makeKeyAndVisible];
return YES;
}
```

在该方法中，共做了3件事：实例化UIWindow，实例化ViewController，然后把ViewController作为根视图控制器放到UIWindow上。每一个iOS应用都有一个UIWindow对象，而每一个UIWindow对象上面都有一个根视图，它所对应的控制器为根视图控制器（ViewController）。UIWindow对象与根视图UIView之间的关系如图2-11所示。在根视图上，我们可以添加子视图。各种控件（包括UIWindow）都继承了UIView。



图2-11 UIWindow对象与根视图

Supporting Files组共有4个文件。为了便于大家理解，经过分析和提炼，我们将各文件的作用以表的形式向大家总结说明，见表2-1。

表2-1 Supporting Files组文件说明表

文 件 名	说 明
HelloWorld-Info.plist	工程属性描述文件，它的命名必须是“工程名+Info.plist”
InfoPlist.strings	工程本地化的字符串文件
main.m	应用程序的入口程序文件，它不是一个类，其中只有一个main()主函数，应用的运行都是由此开始
HelloWorld-Prefix.pch	在这个文件中可以引入一些头文件，这样工程中的其他文件就不需要再引入了。它的命名必须是“工程名+ Prefix.pch”

Frameworks组包含工程里引用到的框架或类库，而Products组是工程将要生成的产品包。

2.2 基于故事板的 HelloWorld 工程

故事板（storyboard）是用来替代xib的技术，也是iOS 5最重要的新特性之一。本节中，我们将用故事板重构HelloWorld。

2.2.1 使用故事板重构HelloWorld

参见2.1节描述的HelloWorld创建过程，我们在图2-3中勾选Use Storyboards复选框。工程创建完成之后，通过导航进入MainStoryboard.storyboard，界面如图2-12所示。

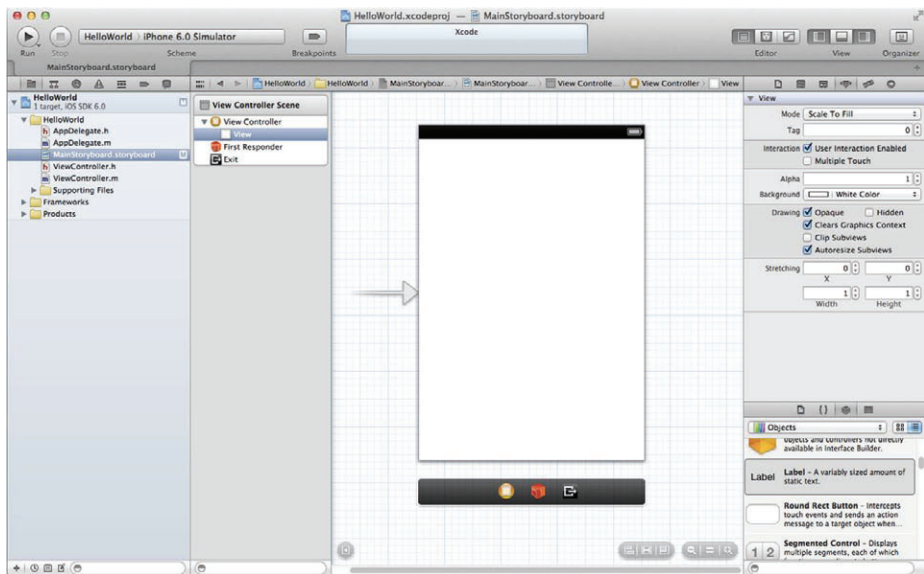


图2-12 创建故事板工程

添加Label控件的操作与2.1节相同，如图2-13所示。

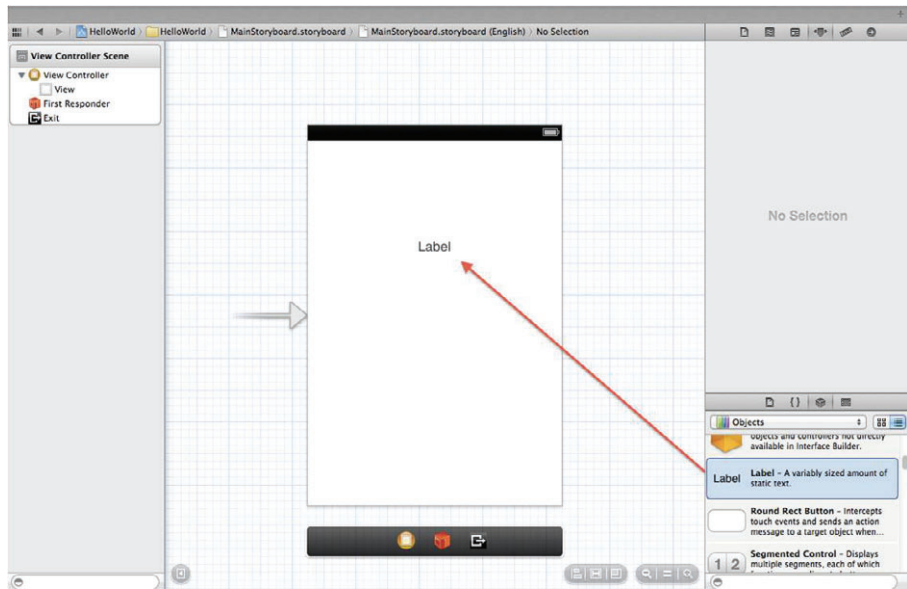



图2-13 添加Label控件

点击按钮运行，运行结果与2.1节相同。

2.2.2 nib、xib与故事板

大家一定会发现某些资料上会提到nib文件，那么nib与xib是怎样的一种关系呢？

最初只有nib文件，后来将其更名为xib文件，但大家一直沿袭nib这个叫法（即称xib文件为nib文件），所以到目前为止，nib等同于xib。xib文件采用xml格式。

那么故事板与xib比较，是否只是文件后缀名不同呢？当然不是，一般而言，一个工程中可以有多个xib文件，一个xib文件对应着一个视图控制器和多个视图。而使用故事板时，一个工程只需要一个主故事板文件就可以了。因此，在包含多个视图控制器的情况下，采用故事板管理比较方便，而且故事板还可以描述界面之间的导航关系。

下面我们举例说明故事板的用法。

我们要做这样一个应用：两个不同的界面，有两个标签分别与其对应，点击标签，实现两个界面的互相切换。该应用采用标签栏导航模式，设计原型草图见图2-14。

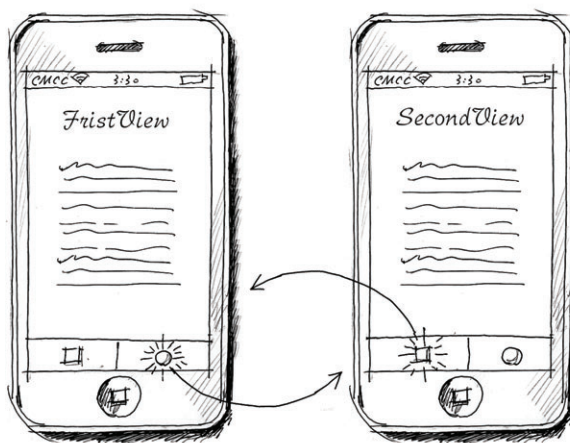


图2-14 设计原型草图

选择Tabbed Application模板，分别采用xib和故事板文件实现，生成的文件如图2-15和图2-16所示。

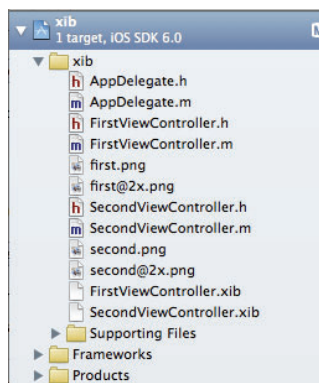


图2-15 采用xib构建文件

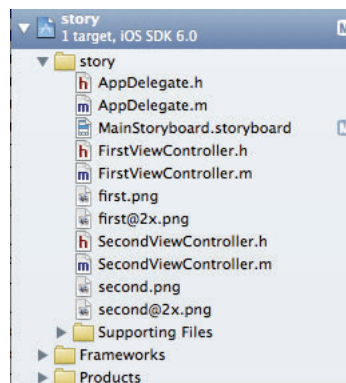


图2-16 采用故事板构建文件

从图2-15和图2-16中可以看到，采用xib技术时，这两个界面有两个xib文件，而采用故事板时，这两个界面只有一个MainStoryboard.storyboard文件。

打开MainStoryboard.storyboard文件，我们会看到如图2-17所示的设计视图。

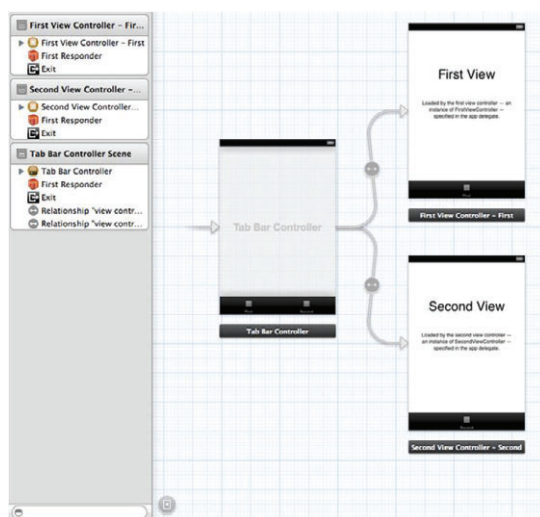


图2-17 故事板设计视图

可以看到，该应用包含两个视图，两个视图存在切换关系。事实上，故事板是多个xib文件集合的描述文件，也采用xml格式。

需要特别提出的是，虽然苹果官方主张使用故事板，但我们要根据具体情况、具体问题对故事板和xib做以取舍，而不是一概而论。当应用数据量很大、界面很多、关系很复杂时，如果使用故事板技术，那我们在Interface Builder设计器中的工作就会变得庞大而复杂，并且整个工程的性能也会受到一定影响。

2.2.3 故事板中的Scene和Segue

Scene和Segue（参见图2-18）是故事板中非常重要的两个概念。每个视图控制器都会对应一个Scene，Scene翻译为“场景”，可以理解为应用的一个界面或屏幕，在这个屏幕中有很多视图或控件，相当于一个xib。这些Scene之间通过Segue连接，Segue不但定义了Scene之间的跳转（或导航）方式，还体现了Scene之间的关系。跳转的类型分为：Push、Modal、Popover和自定义方式。Scene跳转类型还要跟具体的控制器结合使用。Push是树形导航模式，Modal是模态导航模式，Popover是呈现浮动窗口，这些导航模式我们会在后面介绍。

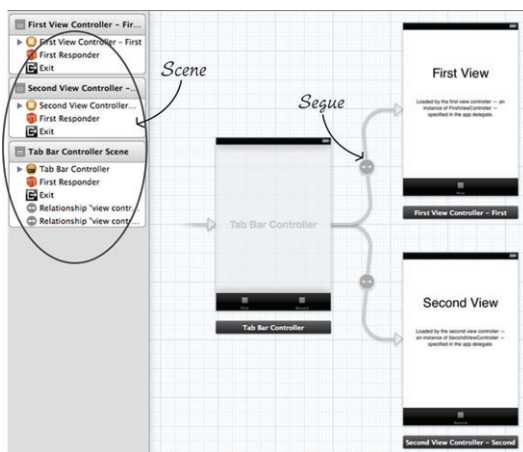


图2-18 故事板Scene和Segue

除了Scene和Segue以外，故事板中还有关于表视图单元格的一些新东西，我们也将后面逐一介绍。

2.3 应用生命周期

作为应用程序的委托对象，AppDelegate类在应用生命周期的不同阶段会回调不同的方法。首先，让我们先了解一下iOS应用的不同状态及它们彼此间的关系，见图2-19。

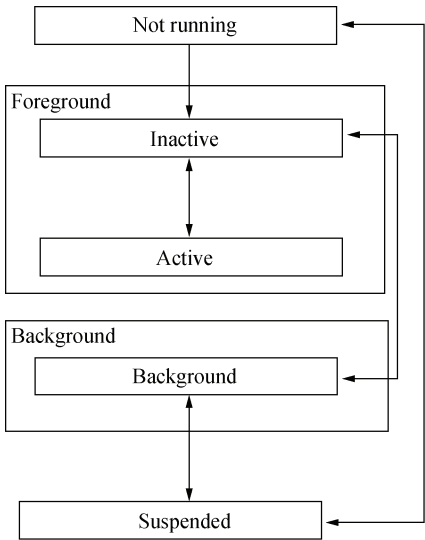


图2-19 iOS应用状态图

下面简要介绍一下iOS应用的5种状态。

- ❑ Not Running（非运行状态）。应用没有运行或被系统终止。
- ❑ Inactive（前台非活动状态）。应用正在进入前台状态，但是还不能接受事件处理。
- ❑ Active（前台活动状态）。应用进入前台状态，能接受事件处理。
- ❑ Background（后台状态）。应用进入后台后，依然能够执行代码。如果有可执行的代码，就会执行代码，如果没有可执行的代码或者将可执行的代码执行完毕，应用会马上进入挂起状态。
- ❑ Suspended（挂起状态）。处于挂起的应用进入一种“冷冻”状态，不能执行代码。如果系统内存不够，应用会被终止。

在应用状态跃迁的过程中，iOS系统会回调AppDelegate中的一些方法，并且发送一些通知。实际上，在应用的生命周期中用到的方法和通知很多，我们选取了几个主要的方法和通知进行详细介绍，具体如表2-2所述。

表2-2 状态跃迁过程中应用回调的方法和本地通知

方 法	本地通知	说 明
application:didFinishLaunchingWithOptions:	UIApplicationDidFinishLaunching Notification	应用启动并进行初始化时会调用该方法并发出通知。这个阶段会实例化根视图控制器
applicationDidBecomeActive:	UIApplicationDidBecomeActive Notification	应用进入前台并处于活动状态时调用该方法并发出通知。这个阶段可以恢复UI的状态（例如游戏状态等）
applicationWillResignActive:	UIApplicationWillResignActive Notification	应用从活动状态进入到非活动状态时调用该方法并发出通知。这个阶段可以保存UI的状态（例如游戏状态等）

(续)

方 法	本地通知	说 明
applicationDidEnterBackground:	UIApplicationDidEnterBackgroundNotification	应用进入后台时调用该方法并发出通知。这个阶段可以保存用户数据，释放一些资源（例如释放数据库资源等）
applicationWillEnterForeground:	UIApplicationWillEnterForegroundNotification	应用进入到前台，但是还没有处于活动状态时调用该方法并发出通知。这个阶段可以恢复用户数据
applicationWillTerminate:	UIApplicationWillTerminateNotification	应用被终止时调用该方法并发出通知，但内存清除时除外。这个阶段释放一些资源，也可以保存用户数据

为了便于观察应用程序的运行状态，我们为AppDelegate.m中的方法添加一些日志输出，具体代码如下：

```
@implementation AppDelegate

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSLog(@"%@", @"application:didFinishLaunchingWithOptions:");
    .....
    return YES;
}

- (void)applicationWillResignActive:(UIApplication *)application
{
    NSLog(@"%@", @"applicationWillResignActive:");
}

- (void)applicationDidEnterBackground:(UIApplication *)application
{
    NSLog(@"%@", @"applicationDidEnterBackground:");
}

- (void)applicationWillEnterForeground:(UIApplication *)application
{
    NSLog(@"%@", @"applicationWillEnterForeground:");
}

- (void)applicationDidBecomeActive:(UIApplication *)application
{
    NSLog(@"%@", @"applicationDidBecomeActive:");
}

- (void)applicationWillTerminate:(UIApplication *)application
{
    NSLog(@"%@", @"applicationWillTerminate:");
}

@end
```

为了让大家更直观地了解各状态与其相应的方法、通知间的关系，下面我们以几个应用场景为切入点进行系统的分析。

2.3.1 非运行状态——应用启动场景

场景描述：用户点击应用图标的时候，可能是第一次启动这个应用，也可能是应用终止后再次启动。该场景的状态跃迁过程见图2-20，共经历两个阶段3个状态：Not running→Inactive→Active。

- ❑ 在Not running→Inactive阶段。调用application:didFinishLaunchingWithOptions:方法，发出UIApplicationDidFinishLaunchingNotification通知。
- ❑ 在Inactive→Active阶段。调用applicationDidBecomeActive:方法，发出UIApplicationDidBecomeActiveNotification通知。

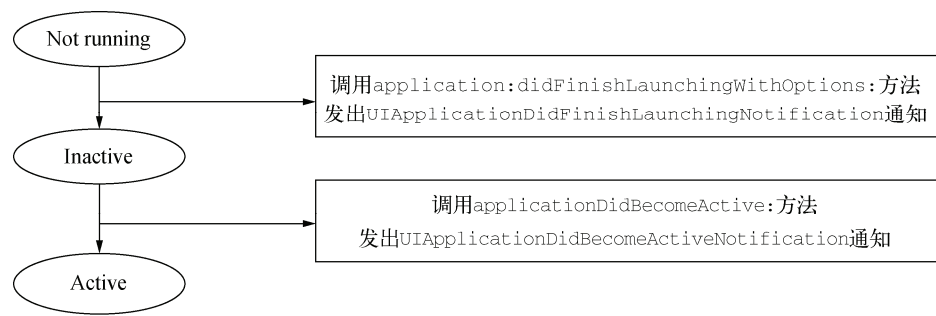


图2-20 应用启动场景的状态跃迁过程

2.3.2 点击Home键——应用退出场景

场景描述：应用处于运行状态（即Active状态）时，点击Home键或者有其他的应用导致当前应用中断。该场景的状态跃迁过程可以分成两种情况：可以在后台运行或者挂起，不可以在后台运行或者挂起。根据产品属性文件（如HelloWorld-Info.plist）中的相关属性Application does not run in background（如图2-21所示）是与否可以控制这两种状态。如果采用文本编辑器打开HelloWorld-Info.plist文件该设置项对应的键是UIApplicationExitsOnSuspend。

Application requires iPhone environment	Boolean	YES
▶ Required device capabilities	Array	(1 item)
Application does not run in background	Boolean	YES
▶ Supported interface orientations	Array	(3 items)

图2-21 UIApplicationExitsOnSuspend键设定

状态跃迁的第一种情况：应用可以在后台运行或者挂起，该场景的状态跃迁过程见图2-22，共经历3个阶段4个状态：Active → Inactive → Background→Suspended。

- ❑ 在Active→Inactive阶段。调用applicationWillResignActive:方法，发出UIApplicationWillResignActiveNotification通知。
- ❑ 在Inactive→Background阶段。应用从非活动状态进入到后台（不涉及我们要重点说明的方法和通知）。
- ❑ 在Background→Suspended阶段。调用applicationDidEnterBackground:方法，发出UIApplicationDidEnterBackgroundNotification通知。

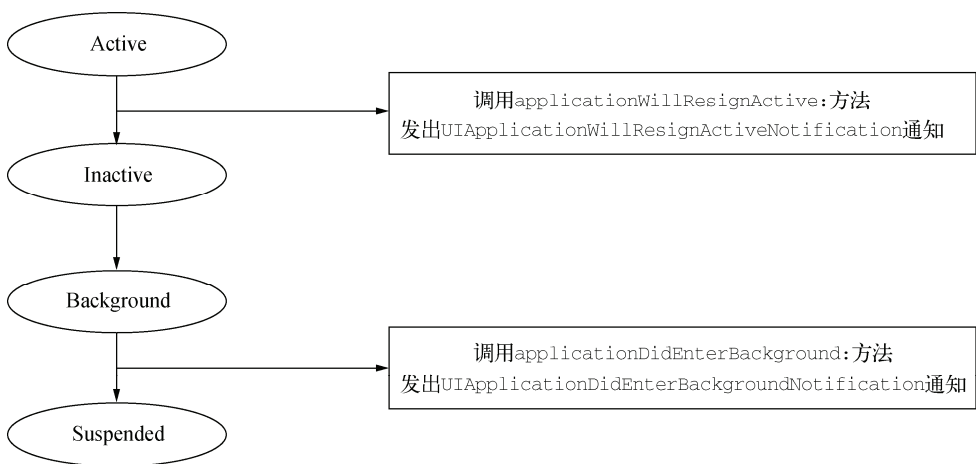


图2-22 点击Home键应用退出场景

状态跃迁的第二种情况：应用不可以在后台运行或者挂起，其状态跃迁情况见图2-23，共经历4个阶段5个状态：Active → Inactive → Background → Suspended → Not running。

- ❑ 在Active→Inactive阶段。应用由活动状态转为非活动状态（不涉及我们要重点说明的方法和通知）。
- ❑ 在Inactive→Background阶段。应用从非活动状态进入到后台（不涉及我们要重点说明的方法和通知）。
- ❑ 在Background→Suspended阶段。调用applicationDidEnterBackground:方法，发出UIApplicationDidEnterBackgroundNotification通知。
- ❑ 在Suspended→Not running阶段。调用applicationWillTerminate:方法，发出UIApplicationWillTerminateNotification通知。

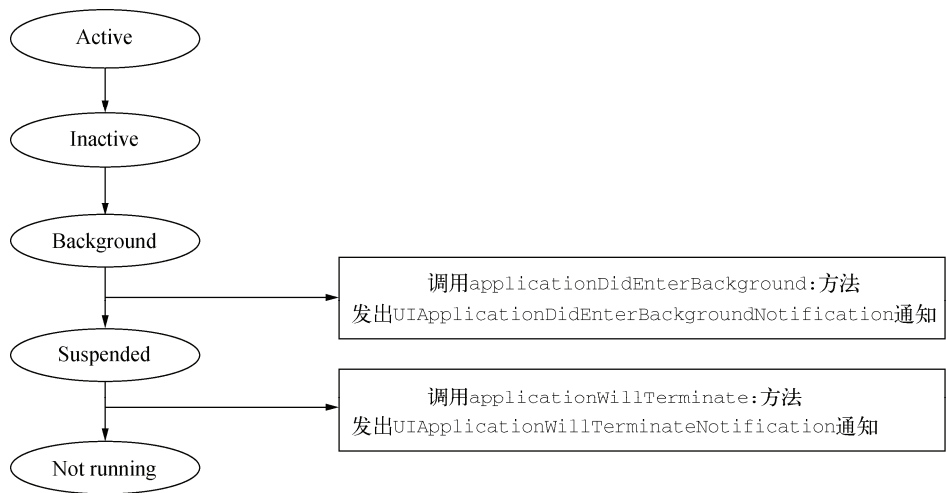


图2-23 点击Home键，应用退出场景

iOS在iOS 4之前不支持多任务，点击Home键时，应用会退出并中断；而在iOS 4之后（包括iOS 4），操作系统能够支持多任务处理，点击Home键应用会进入后台但不会中断（内存不够的情况除外）。

应用在后台也可以进行部分处理工作，处理完成则进入挂起状态。

说明 双击Home键可以快速进入iOS多任务栏，如图2-24所示。在此处可以看到处于后台运行或挂起状态的应用，也可能有处于终止状态的应用驻留在这里。长按这些图标，可以删除这些应用以手动释放内存。



图2-24 iOS多任务栏

2.3.3 挂起重新运行场景

场景描述：挂起状态的应用重新运行。该场景的状态跃迁过程如图2-25所示，共经历3个阶段4个状态：Suspended → Background → Inactive → Active。

- ❑ **Suspended→Background阶段**。应用从挂起状态进入后台（不涉及我们讲述的这几个方法和通知）。
- ❑ **Background→Inactive阶段**。调用`applicationWillEnterForeground:`方法，发出`UIApplicationWillEnterForegroundNotification`通知。
- ❑ **Inactive→Active阶段**。调用`applicationDidBecomeActive:`方法，发出`UIApplicationDidBecomeActiveNotification`通知。

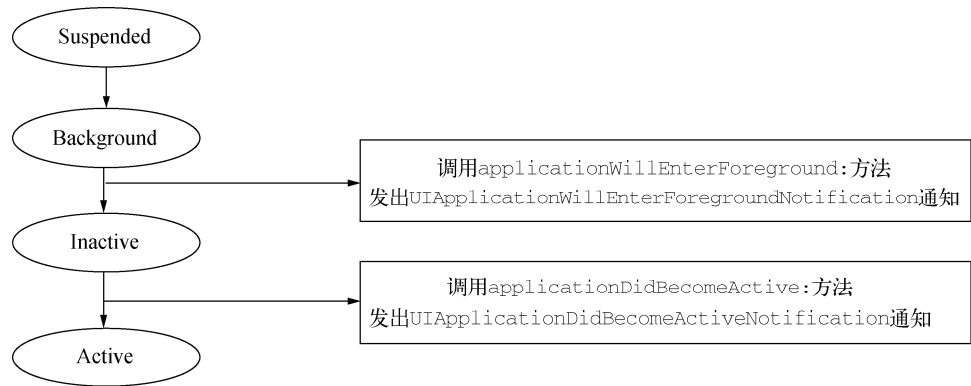


图2-25 重新运行场景的状态跃迁过程

2.3.4 内存清除——应用终止场景

场景描述：应用在后台处理完成时进入挂起状态（这是一种休眠状态），如果这时发出低内存警告，为了满足其他应用对内存的需要，该应用就会被清除内存从而终止运行，该场景的状态跃迁见图2-26。

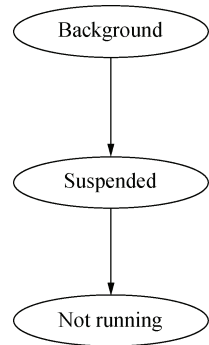


图2-26 内存清除终止场景

内存清除的时候应用终止运行。内存清除有两种情况，可能是系统强制清除内存，也可能是由使用者从任务栏中手动清除（即删掉应用）。内存清除后如果应用再次运行，上一次的运行状态不会被保存，相当于应用第一次运行。

在内存清除场景下，应用不会调用任何方法，也不会发出任何通知。

2.4 视图生命周期

视图是应用的一个重要组成部分，功能的实现与其息息相关，而视图控制器控制着视图，其重要性在整个应用中不言而喻。

2.4.1 视图生命周期与视图控制器关系

以视图的4种状态为基础，我们来系统了解一下视图控制器的生命周期。在视图不同的生命周期中，视图控制器会回调不同的方法，具体如图2-27所示。

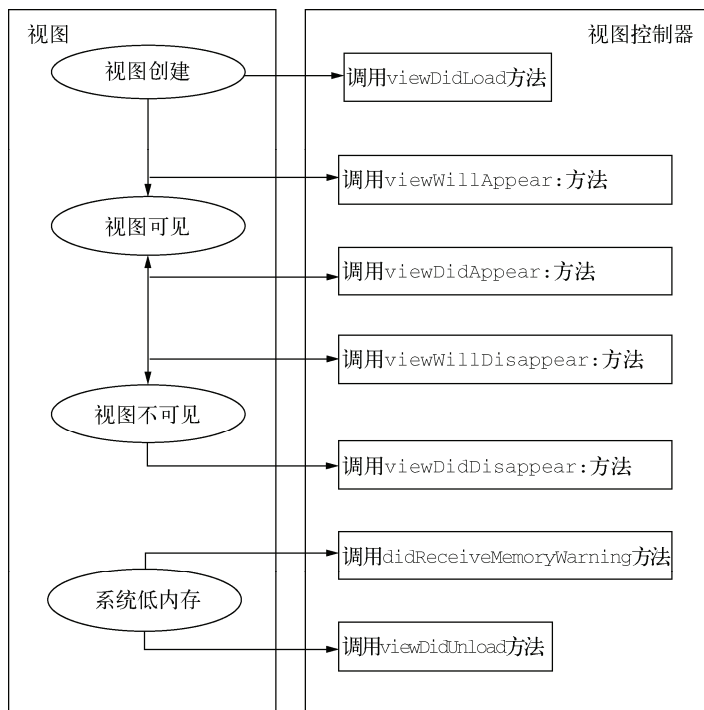


图2-27 视图控制器的一些主要方法

在视图控制器已被实例化，视图被加载到内存中时调用viewDidLoad方法，这个时候视图并未出现。在该方法中，通常进行的是对所控制的视图进行初始化处理。

视图可见前后会调用viewWillAppear:方法和viewDidAppear:方法；视图不可见前后会调用viewWillDisappear:方法和viewDidDisappear:方法。4个方法调用父类相应的方法以实现其功能，编码时该方法的位置可根据实际情况做以调整，参见如下代码：

```
-(void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:YES];
    .....
}
```

viewDidLoad方法在应用运行的时候只调用一次，而这上述4个方法可以被反复调用多次，它们的使用很广泛但同时也具有很强的技巧性。例如，有的应用会使用重力加速计，重力加速计会不断轮询设备以实时获得设备在z轴、x轴和y轴方向的重力加速度。不断的轮询必然会耗费大量电能进而影响电池使用寿命，我们通过利用这4个方法适时地打开或者关闭重力加速计来达到节约电能的目的。怎么使用这4个方法才能做到“适时”是一个值得思考的问题。

在低内存情况下，iOS会调用didReceiveMemoryWarning:和viewDidUnload:方法。在iOS 6之后，就不再使用viewDidUnload:，而仅支持didReceiveMemoryWarning:。didReceiveMemoryWarning:方法的主要职能是释放内存，包括视图控制器中的一些成员变量和视图的释放。现举例如下：

```

- (void)didReceiveMemoryWarning {
    self.button = nil;
    self.myStringD = nil;
    [myStringC release];
    [super didReceiveMemoryWarning];
}

```

除了上述5个方法视图控制器外，还有很多其他方法，随着学习的深入，我们会逐一向大家介绍。

2.4.2 iOS 6 UI状态保持和恢复

iOS设计规范中要求，当应用退出的时候（包括被终止运行的时候），需要保持界面中UI元素的状态，当再次进来的时候看到的状态与退出时是一样的。在iOS 6之后，苹果提供以下API使得UI状态保持和恢复变得很容易。

在iOS 6中，我们可以在以下3种地方实现状态保持和恢复：

- ❑ 应用程序委托对象
- ❑ 视图控制器
- ❑ 自定义视图

为了演示这个功能，我们把基于故事板的HelloWorld工程改造一下。在界面中添加一个文本框，操作与在2.1节中添加Label控件类似，如图2-28所示。关于文本框的技术细节，我们会在后面的章节中介绍。

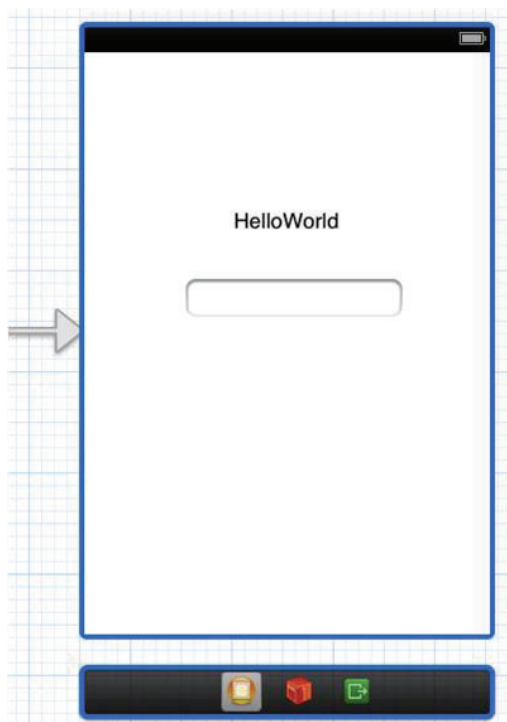



图2-28 视图添加文本框

用户在文本框中输入一些内容，应用程序退出并且终止，当用户再次进来的时候，文本框中还会保持原来输入的内容。然后在Interface Builder的Scene中选中View Controller，打开右边的标识检查器，如图2-29所示，设置Restoration ID（恢复标识）为viewController。

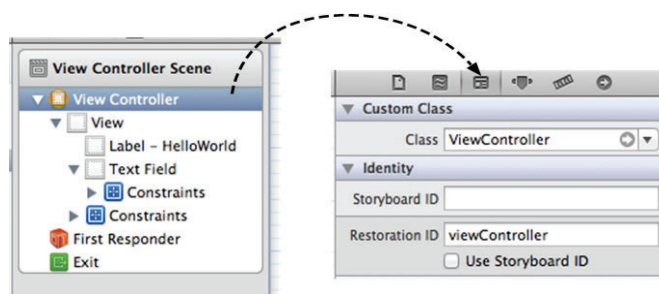


图2-29 设置控制器的恢复标识

恢复标识是iOS 6为了实现UI状态保持和恢复添加的设置项目。我们还需要在应用程序委托对象AppDelegate代码部分做一些修改，添加的代码如下：

```

- (BOOL) application: (UIApplication *) application shouldSaveApplicationState:
    (NSCoder *) coder
{
    return YES;
}

- (BOOL) application: (UIApplication *) application shouldRestoreApplicationState:
    (NSCoder *) coder
{
    return YES;
}

- (void) application: (UIApplication *) application willEncodeRestorableStateWithCoder:
    (NSCoder *) coder
{
    [coder encodeFloat:2.0 forKey:@"Version"];
}

- (void) application: (UIApplication *) application didDecodeRestorableStateWithCoder:
    (NSCoder *) coder
{
    float lastVer = [coder decodeFloatForKey:@"Version"];
    NSLog(@"lastVer = %f", lastVer);
}

```

其中application:shouldSaveApplicationState:方法在应用退出时调用，负责控制是否允许保存状态，返回YES情况是可以保存，NO是不保存。

application:shouldRestoreApplicationState:方法在应用启动时调用，负责控制是否恢复上次退出时的状态，返回YES表示可以恢复，返回NO表示不可以恢复。

application:willEncodeRestorableStateWithCoder:方法在保存时调用，在这个方法中实现UI状态或数据的保存，其中[coder encodeFloat:2.0 forKey:@"Version"]语句是保存简单数据。

application:didDecodeRestorableStateWithCoder:方法在恢复时调用，在这个方法中实现UI状态或数据的恢复，其中[coder decodeFloatForKey:@"Version"]语句用于恢复上次保存的数据。

想要实现具体界面中控件的保持和恢复，还需要在它的视图控制器中添加一些代码。我们在ViewController.m中添加的代码如下：

```

- (void) encodeRestorableStateWithCoder: (NSCoder *) coder
{
    [super encodeRestorableStateWithCoder:coder];
    [coder encodeObject:self.txtField.text forKey:kSaveKey];
}

```

```

- (void)decodeRestorableStateWithCoder: (NSCoder *)coder
{
    [super decodeRestorableStateWithCoder:coder];
    self.txtField.text = [coder decodeObjectForKey:kSaveKey];
}

```

在iOS 6之后,视图控制器都添加了两个方法——`encodeRestorableStateWithCoder:`和`decodeRestorableStateWithCoder:`,用来实现该控制器中的控件或数据的保存和恢复。其中`encodeRestorableStateWithCoder:`方法在保存时候调用, `[coder encodeObject:self.txtField.text forKey:kSaveKey]`语句是按照指定的键保存文本框的内容,`decodeRestorableStateWithCoder:`方法在恢复时调用, `[coder decodeObjectForKey:kSaveKey]`在恢复文本框内容时调用,保存和恢复事实上就是向一个归档文件中编码和解码的过程。

为了测试是否能够保持和恢复,我们可以将工程属性文件HelloWorld-Info.plist中的相关属性Application does not run in background设置为YES,使应用退出时终止程序的运行。

2.5 设置产品属性

在前面讲解应用生命周期时,为了禁止应用在后台运行,我们将HelloWorld-Info.plist文件中的Application does not run in background属性修改为YES (即`UIApplicationExitsOnSuspend = YES`),这项操作就属于产品属性的设置。在Xcode中,产品与Target直接相关,而Target与Project直接相关。

2.5.1 Xcode中的Project和Target

打开HelloWorld工程时,我们会看到如图2-30所示的界面。产品属性包括Project和Target两块内容。一个工程只有一个Project,但可以有一个或多个Target。

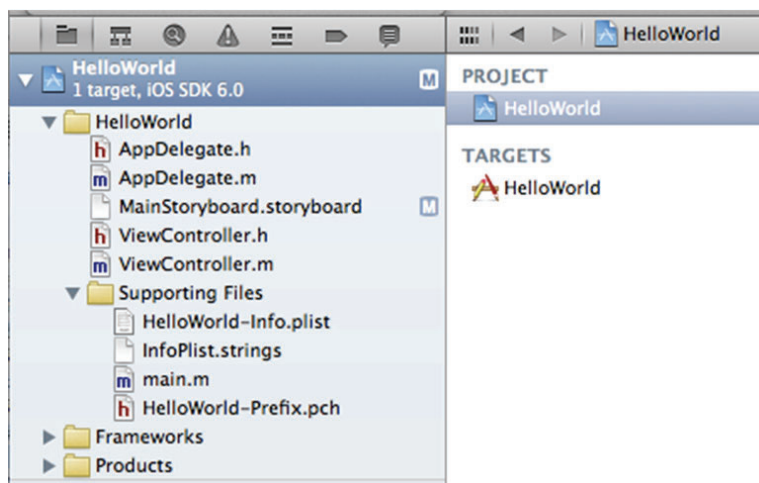


图2-30 Xcode的Project和Target

我们所创建的HelloWorld只有一个Target,下面我们为之前使用故事板实现的HelloWorld工程增加一个Target。首先,依次选择File→New→Target菜单项,此时会弹出一个模板选择对话框,如图2-31所示。

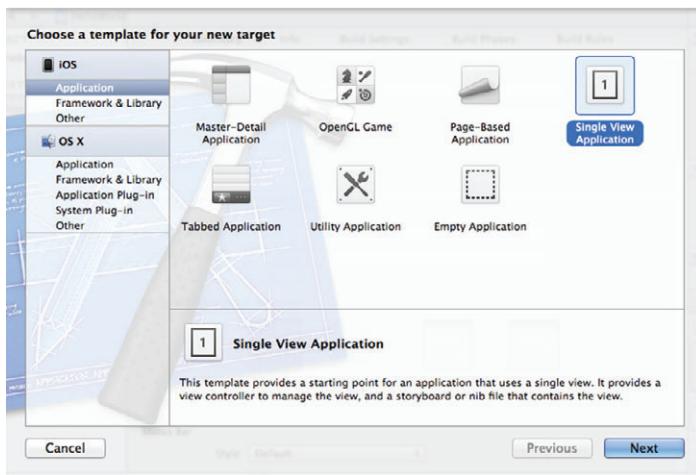


图2-31 选择Target模板对话框

这里选择的模板与新建工程时选择的模板完全一样，然后点击Next按钮，将出现如图2-32所示的对话框。

根据情况逐一设定后，点击Finish按钮，现在我们已经成功为HelloWorld新增了一个Target。查看左边的导航面板，可以发现右边有两个Target，并同时生成一套完整的文件——main.m、AppDelegate、ViewController和MainStoryboard.storyboard，它们独立于原来的Target而存在，如图2-33所示。

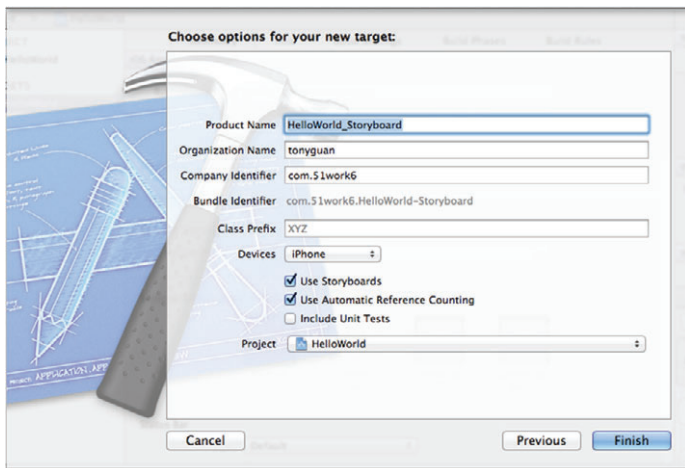


图2-32 Target的一些选项设定

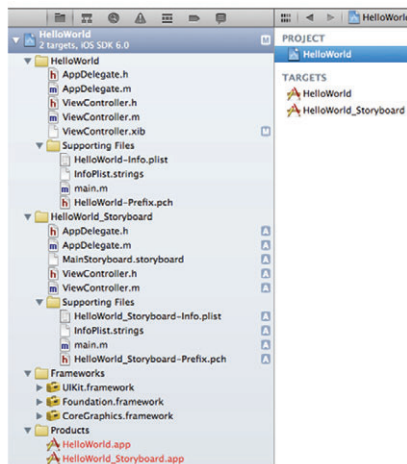


图2-33 新创建Target

要指定运行哪一个Target，可以通过选择不同的Scheme来实现。如图2-34所示，在Xcode的左上角选择HelloWorld_Storyboard→iPhone 6.0 Simulator，就可以在iPhone 6.0 Simulator上运行HelloWorld_Storyboard Target了。

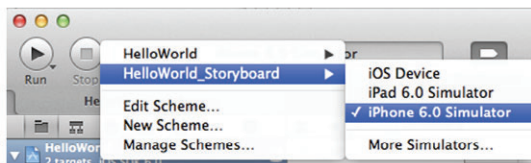


图2-34 选择Scheme

2.5.2 设置常用的产品属性

Target继承了Project。对于Target和Project下都有的设置项，可根据需要对Target进行再设置，此设置可覆盖Project的设置。

Project中的属性设置相对比较简单，大家可以参考官方的相关资料。这里为大家介绍Target下4个常用的产品属性。

1. 设定屏幕方向

如图2-35所示，在导航面板中选择HelloWorld_Storyboard Target，然后在右侧选择Summary选项卡，此时可以发现下面的Supported Interface Orientations区域中有4个按钮，它们代表设备支持的4个方向，其中按下状态（深色）代表支持该指定方向。

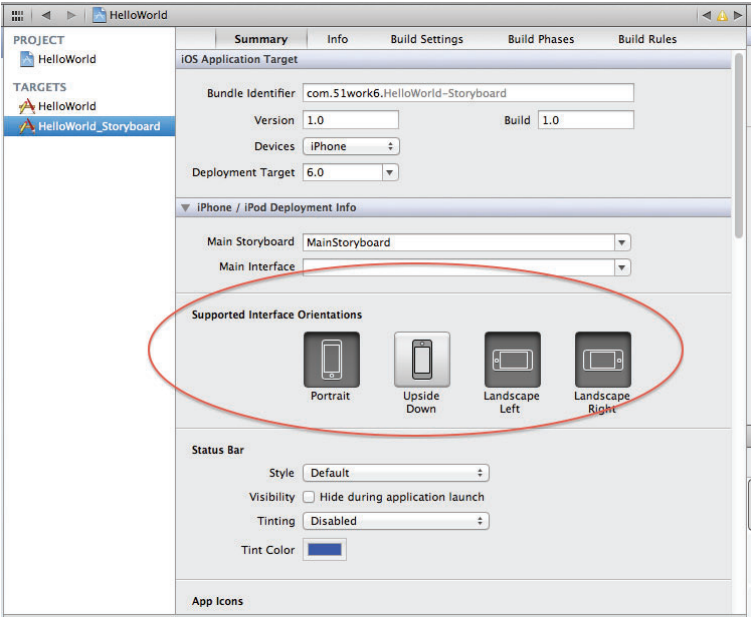


图2-35 设定支持的屏幕方向

说明 鉴于Target的属性存储在HelloWorld_Storyboard-Info.plist文件中，也可以直接修改HelloWorld_Storyboard-Info.plist文件以完成属性各项的设置，如图2-36所示。

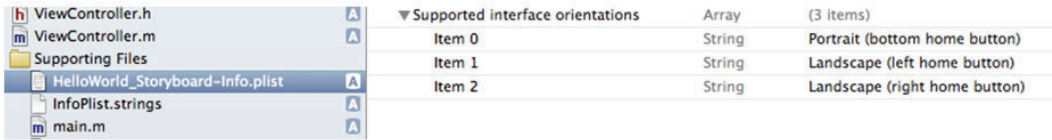


图2-36 在HelloWorld_Storyboard-Info.plist文件中设定屏幕方向

2. 设置应用图标

如图2-37所示，在App Icons区域中，右击选择图片文件，然后为应用添加图标。后面的App Icons选择区域用来为具备Retina显示技术（iPhone 4以后才支持）的设备提供图标，操作同前。

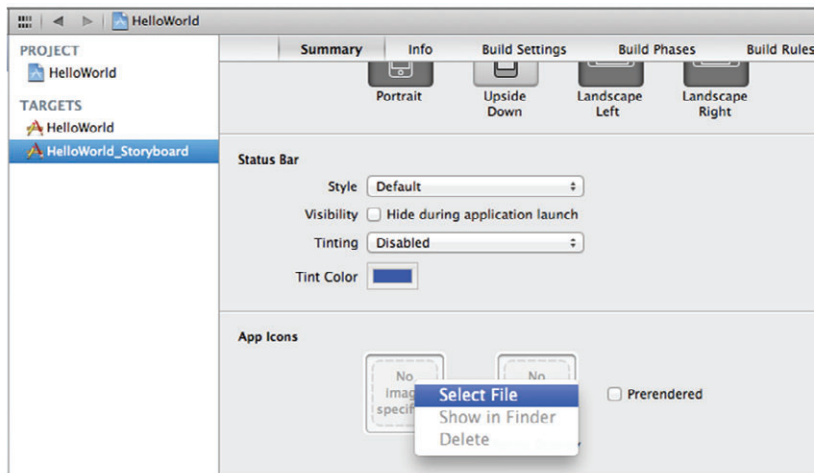


图2-37 应用图标的设定

图标对其图片文件的尺寸和名称要求很严格,普通App Icons的图片大小为 57×57 像素,名称为Icon.png; Retina显示屏App Icons的图片大小为 114×114 像素,名称为Icon@2x.png。

说明 应用图标名可以不做修改,直接采用原来的自定义名称,因为Xcode会自动给图标改名,但文件的格式(PNG)和大小必须按照要求设定。

3. 设置启动屏幕

启动屏幕是应用启动时一闪而过的图片,它也在Summary选项卡中设置。如图2-38所示,在Launch Images区域中,右击选择图片区域,在弹出界面中选择Select File,选择我们为启动屏幕准备好的图片文件。对于普通的iPhone设备,这个图片的大小为 320×480 像素,文件以Default.png命名。右边的选择框是为Retina显示技术设备提供图片的,大小是 640×960 像素,文件命名为Default@2x.png。

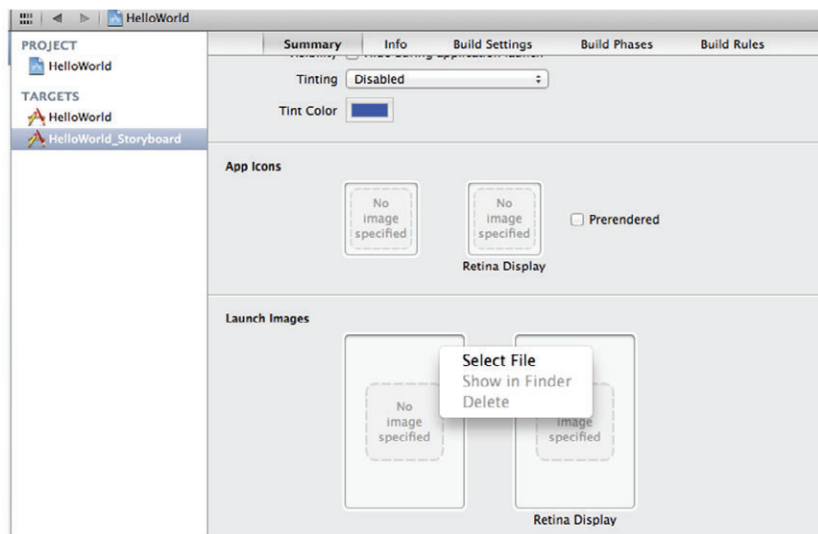


图2-38 应用屏幕的设定

说明 启动屏幕图片的名字可以不做修改，因为Xcode会自动给它改名，但文件的格式（PNG）和大小必须按照要求进行设定。iPad设备会有4种图片，命名也与iPhone不同。

4. 设置设备支持情况

我们可以让应用支持iPhone设备或iPad设备，或者同时支持iPhone和iPad设备。如图2-39所示，在Summary选项卡中的iOS Application Target区域中单击Device下拉列表，从中选择iPhone、iPad或者Universal选项，其中Universal表示同时支持iPhone和iPad设备。

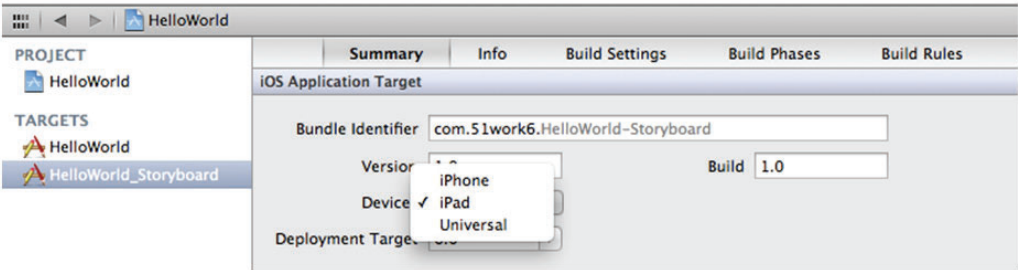


图2-39 设置设备支持情况

事实上，产品的相关属性还有很多，我们会在后面继续了解。

2.6 iOS API 简介

苹果的iOS API在不同版本间有很多变化，本书采用的是iOS 6。本节中，我们会介绍iOS 6有哪些API，如何使用这些API的帮助文档以及如何使用官方案例。

2.6.1 API概述

iOS的整体架构图如图2-40所示，分为4层——Cocoa Touch层、Media层、Core Services层和Core OS层，下面概要介绍一下这4层。



图2-40 iOS整体架构图

1. Cocoa Touch层

该层提供了构建iOS应用的一些基本系统服务（如多任务、触摸输入和推送通知等）和关键框架（见表2-3）。

表2-3 Cocoa Touch层包括的框架

框 架	前 缀	说 明
Address Book UI	AB	访问用户的联系人信息
Event Kit UI	EK	访问用户的日历事件数据
Game Kit	GK	提供能够进行点对点的网络通信的API
iAd	AD	在应用中嵌入广告
Map Kit	MK	在应用中嵌入地图和地理信息编码等
Message UI	MF	提供与发送E-mail相关的API
Twitter	TW	提供发送Twitter的接口
UIKit	UI	提供UI类

2. Media层

Media层提供了图形、音频、视频和AirPlay技术，包括的框架如表2-4所示。

表2-4 Media层包括的框架

框 架	前 缀	说 明
Assets Library	AL	提供访问用户的图片和视频的接口
AudioToolbox	Audio	录制或播放音频、音频流以及格式转换
AudioUnit	Audio, AU	提供使用内置音频单元服务，以及音频处理模块
AV Foundation	AV	提供播放与录制音频和视频的Objective-C接口
Core Audio	Audio	提供录制、制作、播放音频的C语言接口
Core Graphics	CG	提供Quartz 2D接口
Core Image	CI	提供操作视频和静态图像的接口
Core MIDI	MIDI	提供用于处理MIDI数据低层的API
Core Text	CT	提供渲染文本和处理字体的简单、高效的C语言接口
Core Video	CV	提供用于处理音频和视频的API
Image I/O	CG	包含一些读写图像数据类
GLKit	GLK	包含了构建复杂OpenGL ES应用的Objective-C实用类
Media Player	MP	包含全屏播放接口
OpenAL	AL	包含了OpenAL（跨平台的音频）的C语言接口
OpenGL ES	EAGL, GL	包含OpenGL ES（跨平台的2D/3D图形库）的C语言接口
Quartz Core	CA	提供动画接口类

3. Core Services层

该层提供了iCloud、应用内购买、SQLite数据库和XML支持等技术，包括的主要框架如表2-5所示。

表2-5 Core Services层包括的框架

框 架	前 缀	说 明
Accounts	AC	用于访问用户的Twitter账户（iOS 5之后才有此API）
AddressBook	AB	访问用户的联系人信息
AdSupport	AS	获得iAD广告标识
CFNetwork	CF	提供了访问Wi-Fi网络和蜂窝电话网络的API
Core Data	NS	提供管理应用数据的ORM接口
CoreFoundation	CF	它是iOS开发中最基本的框架，包括数据集
Core Location	CL	提供定位服务的API
CoreMedia	CM	提供AV Foundation框架使用的底层媒体类型。可以精确控制音频或视频的创建及展示
CoreMotion	CM	接收和处理重力加速计以及其他的运动事件
CoreTelephony	CT	提供访问电话基本信息的API
Event Kit	EK	访问用户的日历事件数据
Foundation	NS	为Core Foundation框架的许多功能提供Objective-C封装，是Objective-C最为基本框架
MobileCoreServices	UT	定义统一类型标识符（UTI）使用的底层类型
Newsstand Kit	NK	提供在后台下载杂志和新闻的API接口（iOS 5之后才有此API）
Pass Kit	PK	提供访问各种优惠券的API（iOS 6之后才有此API）
QuickLook	QL	该框架可以预览无法直接查看的文件内容，例如打开PDF文件
Social	SL	提供社交网络访问API，中国区提供新浪微博API（iOS 6之后才有此API）
Store Kit	SK	提供处理应用内置收费的资金交易
SystemConfiguration	SC	用于确定设备的网络配置（例如，使用该框架判断Wi-Fi或者蜂窝连接是否正在使用中），也可以用于判断某个主机服务是否可以使用

4. Core OS层

该层提供了一些低级功能，开发中一般不直接使用它。该层包括的主要框架如表2-6所示。

表2-6 Core OS层包括的框架

框 架	前 缀	说 明
Accelerate	AC	访问重力加速计API
Core Bluetooth	CB	访问低能耗蓝牙设备API
External Accessory	EA	访问外围配件API接口
Generic Security Services	gss	提供一组安全相关的服务
Security	CSSM, Sec	管理证书、公钥、私钥和安全信任策略API

2.6.2 如何使用API帮助

对于一个初学者来说，学会在Xcode中使用API帮助文档是非常重要的。下面我们通过一个例子来介绍API帮助文档的用法。

在编写HelloWorld程序时，我们可以看到ViewController.m的代码，具体如下所示：

```
@implementation ViewController


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

```

}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
}
@end

```

对于didReceiveMemoryWarning方法，我有很多困惑，这时候需要自己查找帮助文档。如果只是简单查看帮助信息，可以双击选择didReceiveMemoryWarning方法，然后选择右边的快捷帮助检查器，如图2-41所示。

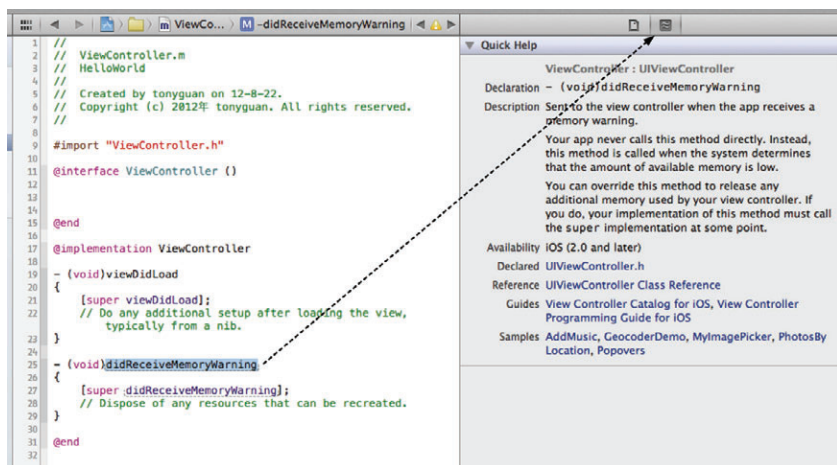


图2-41 Xcode快捷帮助检查器

在打开的Xcode快捷帮助检查器窗口中，可以看到这个方法的描述，其中包括使用的iOS版本、相关主题以及一些相关案例。这里需要说明的是，如果需要查看官方的案例，直接从这里下载即可。

如果想查询比较完整的、全面的帮助文档，可以按住Alt键双击didReceiveMemoryWarning方法名，这样就会打开一个Xcode的API帮助文档，如图2-42所示。

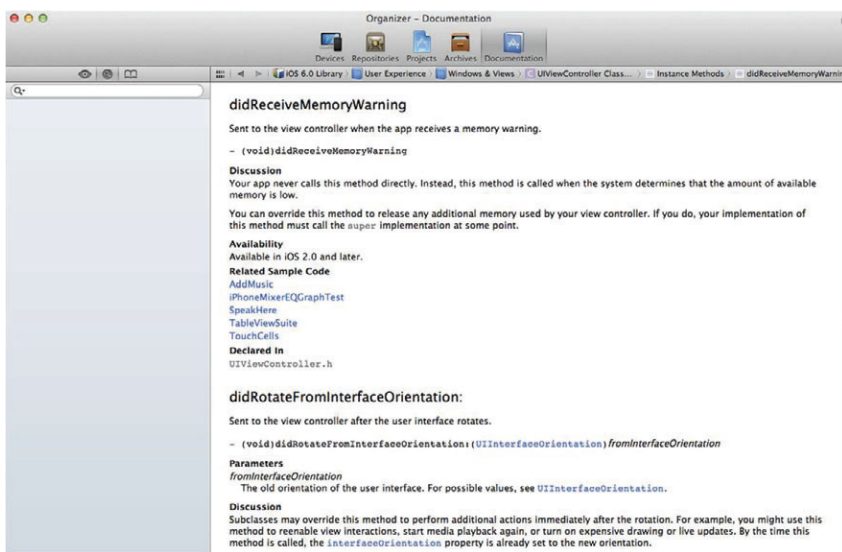


图2-42 Xcode帮助窗口

API帮助文档还提供给我们一些官方案例，选择案例会打开一个新的窗口，如图2-43所示，这里可以查看案例介绍。点击Open Project按钮，可以打开并下载这个案例工程。

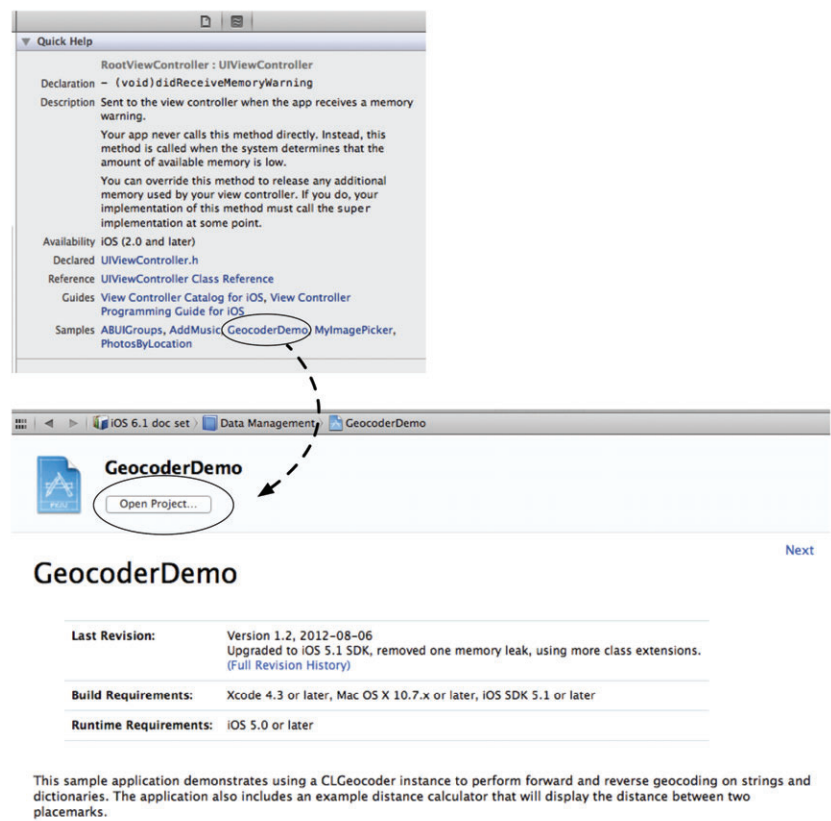


图2-43 官方案例

2.7 小结

在本章中，我们首先介绍了如何使用nib和故事板技术创建工程，了解了这二者的优缺点和彼此间的差异。接着通过HelloWorld工程讨论了iOS工程模板、应用的运行机制和生命周期、视图控制器的生命周期，然后介绍了4项常用产品属性的设置。最后，我们向大家介绍了API帮助文档和官方案例的用法。

谈起设计模式，大家在兴奋之余难免心生畏惧。兴奋的原因是我们能够灵活而有机地运用设计模式，这就意味着编程工作的高效性和产品健壮性、安全性的提高。很多人自豪于能够透彻掌握“某某设计模式”，而面试官们也常常把对“某某设计模式”的掌握程度作为考评求职者的重要标准之一。设计模式的重要性和技巧性可想而知。那么，畏惧从何说起呢？设计模式是个很庞杂的知识体系，即便是同一设计模式在不同开发语言环境下也存在很大的差异，而真正能驾驭设计模式的开发者的确不多。

那么，什么是设计模式呢？设计模式是在特定场景下对特定问题的解决方案，这些解决方案是经过反复论证和测试总结出来的。实际上，除了软件设计，设计模式也被广泛应用于其他领域，比如UI设计和建筑设计等。

软件设计模式大都来源于GoF^①的23种设计模式。该书的设计模式都是面向对象的，在C++、Java和C#领域都有广泛的应用。Cocoa和Cocoa Touch框架中的设计模式也基本上是这23种设计模式的演变，但是具体来说，Cocoa和Cocoa Touch中的设计模式仍然存在着差异。

关于iOS开发，我们将重点分析Cocoa框架下的几个设计模式。当然，Cocoa框架下关于设计模式的内容远不止这些，但为了能在尽量短的时间内让其为我所用，我们经过审慎地思考并结合多年的开发经验，选择了如下4种设计模式：单例模式、委托模式、观察者模式和MVC模式。本章中，我们主要按照“问题提出”、“实现原理”和“应用案例”的结构来介绍各模式。

3.1 单例模式

单例模式的作用是解决“应用中只有一个实例”的一类问题。

3.1.1 问题提出

在一个iOS应用的生命周期中，有时候我们只需要某个类的一个实例。例如，iOS设备都有一个重力加速计硬件设备，要访问设备在x轴、y轴和z轴上的重力加速度，就必然要有一个类能够与硬件设备沟通来实时获得这些数据，这个类就是UIAccelerometer。除了实时地获得数据，该类还能够保持x轴、y轴和z轴的状态。但是这个类只需要一个实例就够了，如果有多个实例，就会占用过多的内存。

再有，当应用程序启动时，应用的状态由UIApplication类的一个实例维护，这个实例代表了整个“应用程序对象”，它只能是一个实例，其作用是实现应用程序中一些共享资源的访问和状态的保持等。

3.1.2 实现原理

单例模式一般会封装一个静态属性，并提供静态实例的创建方法，其UML类图如图3-1所示。

^① *Design Patterns: Elements of Reusable Object-Oriented Software*（中文版《设计模式》）一书由Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 合著（Addison-Wesley，1995），这四位作者常被称为“四人组”（Gang of Four，GoF）。

实现的参考代码如下：

```
//
//Singleton.h
//
@interface Singleton : NSObject
+ (Singleton*)sharedManager;
@property (nonatomic ,strong) NSString* singletonData;
@end

//
//Singleton.m
//
#import "Singleton.h"
@implementation Singleton
@synthesize singletonData = _singletonData;
static Singleton *sharedManager = nil;
+ (Singleton*)sharedManager
{
    static dispatch_once_t once;
    dispatch_once(&once, ^{
        sharedManager = [[self alloc] init];
    });
    return sharedManager;
}
@end
```

Singleton
属性： static sharedInstance
操作： +(Singleton*)sharedInstance

图3-1 单例设计模式类图

其中static Singleton *sharedManager为静态变量，类方法为+ (Singleton*)sharedManager。sharedManager方法采用了GCD（Grand Central Dispatch）技术，这是一种基于C语言的多线程访问技术。在上述代码中，dispatch_once函数就是由GCD提供的，它的作用是在整个应用程序生命周期中只执行一次代码块（^ {...}）。dispatch_once_t是GCD提供的结构体，使用时需要将GCD地址传给dispatch_once函数。dispatch_once函数能够记录该代码块是否被调用过。

dispatch_once函数不仅意味着代码仅会被运行一次，而且还意味着此运行还是线程同步的。也就是说，当我们使用了dispatch_once函数时，就不再需要使用诸如@synchronized之类的语句。

说明 由于自iOS 5开始，内存计数上用ARC（自动引用计数）替代了原来的MRC（手动引用计数），同时实现方式也发生了很大的变化，所以本书中的代码也都采用ARC技术的形式。如果读者想了解在iOS 5之前非ARC的实现方式，可以参考苹果公司的代码：https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaObjects/CocoaObjects.html#//apple_ref/doc/uid/TP40002974-CH4-SW32。

3.1.3 应用案例

在Cocoa Touch框架中，有UIApplication、UIAccelerometer、NSUserDefaults和NSNotificationCenter等单例类。另外，NSFileManager和NSBundle类虽然属于Cocoa框架的内容，但也可以在Cocoa Touch框架中使用（Cocoa框架中的单例类有NSFileManager、NSBundle、NSWorkspace和NSApplication等）。

1. UIApplication

UIApplication类的实例提供了应用程序的集中控制点来保持应用的状态。UIApplication实例总是分配给应用程序委托对象（UIApplicationDelegate），通过应用程序委托对象来响应低内存、应用启动、后台运行和应用终止等事件。在HelloWorld案例中，AppDelegate就是这个应用程序的委托对象，它实现了UIApplicationDelegate协议。

UIApplication类有很多方法和属性，下面我们重点介绍其中几个。

❑ **+ sharedApplication**方法。创建和获得UIApplication实例的方法。

❑ **idleTimerDisabled**属性。设定和获得“空闲时间禁止”的状态。idleTimerDisabled属性的默认值是NO，即默认情况下系统会锁定屏幕。当idleTimerDisabled = YES时，则不会开启“空闲时间禁止”状态，系统不会锁定屏幕。开启这项设定需要谨慎，它会使你的应用比较耗电。

❑ **- openURL:**方法。可以打开一些内置的iOS应用，其中包括打开浏览器、打开Google地图、拨打电话、发送短信和发送E-mail等。

打开浏览器的示例代码如下：

```
NSURL *url = [NSURL URLWithString:@"http://www.51work6.com"];
[[UIApplication sharedApplication] openURL:url];
```

打开Google地图时，实际上是通过内置浏览器来打开，示例代码如下：

```
NSString* searchQuery = @"清华大学";
searchQuery = [searchQuery stringByAddingPercentEscapesUsingEncoding:
    NSUTF8StringEncoding];
NSString* urlString = [NSString stringWithFormat:
    @"http://maps.google.com/maps?q=%@", searchQuery];
[[UIApplication sharedApplication] openURL:[NSURL URLWithString:urlString]];
```

其中NSString的stringByAddingPercentEscapesUsingEncoding方法将字符串转换为URL编码，例如“%E6%B8%85%E5%8D%8E%E5%A4%A7%E5%AD%A6”是“清华大学”的URL编码。

拨打电话时，苹果官方要求使用该方法调用内置拨号程序，示例代码如下：

```
[[UIApplication sharedApplication] openURL:[NSURL URLWithString:@"tel://10086"]];
```

发送短信时，苹果官方要求使用该方法调用内置发送短信程序，示例代码如下：

```
[[UIApplication sharedApplication] openURL:[NSURL URLWithString:@"sms:10086"]];
```

发送E-mail时，这种方式可以发送简单的不带附件的E-mail，示例代码如下：

```
[[UIApplication sharedApplication] openURL:[NSURL URLWithString:@"mailto://eorient@sina.com"]];
```

2. UIAccelerometer

单例类UIAccelerometer前面也讲过，它可以访问重力加速计硬件设备，实时获得设备在x轴、y轴和z轴方向上的重力加速度。

+ sharedAccelerometer方法是创建和获得UIAccelerometer实例的共享方法。

与UIApplication类似，UIAccelerometer也有对应的委托对象，其委托对象为UIAccelerometerDelegate。UIAccelerometer将实例分配给委托对象UIAccelerometerDelegate，然后由委托对象响应重力加速计事件。

3. NSUserDefaults

单例类NSUserDefaults可以很方便地读取应用设置项目。

+ standardUserDefaults方法是创建和获得NSUserDefaults实例的静态方法。

4. NSNotificationCenter

单例类NSNotificationCenter提供信息广播通知，它采用观察者模式的通知机制。

+ defaultCenter方法是创建和获得NSNotificationCenter实例的共享方法。

5. NSFileManager

NSFileManager提供了访问文件系统的通用操作，可以定位、创建、复制文件和文件夹。在iOS 5和Mac OS X v10.7之后，它还可以管理存储在 iCloud 上的数据。

+ defaultManager方法是创建和获得NSFileManager实例的方法。除了该方法外，创建NSFileManager对象时还可以使用实例构造方法- init。这两种方法有着比较大的差别，+ defaultManager方法总是返回相同

的NSFileManager对象，但如果要使用委托（NSFileManagerDelegate）完成基于文件的操作并接收通知，应该使用- initWithURL方法创建一个新的实例，而不是使用共享的对象。

6. NSBundle

NSBundle提供了动态加载（或卸载）可执行代码、定位资源文件以及资源本地化、访问文件系统等功能。

+ mainBundle方法是创建和获得NSBundle实例的共享方法。

3.2 委托模式

委托模式从GoF装饰（Decorator）模式、适配器（Adapter）模式和模板方法（Template Method）模式等演变而来。几乎每一个应用都会或多或少地用到委托模式。不只是Cocoa Touch框架，在Cocoa框架中，委托模式也得到了广泛的应用。

3.2.1 问题提出

对于应用生命周期的非运行状态应用启动场景，我们把从点击图标到启动第一个界面的过程细化了一下，具体如图3-2所示。

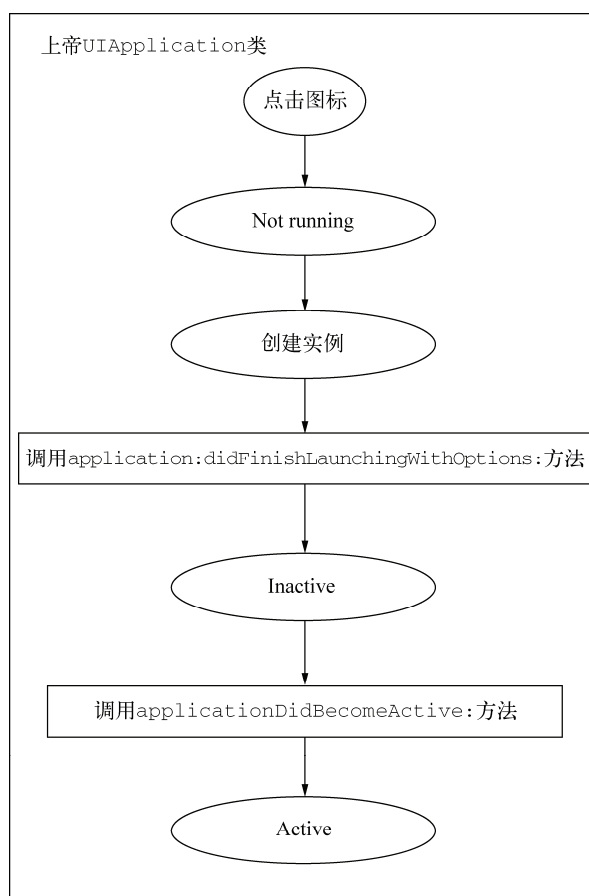


图3-2 非运行状态应用启动场景的流程图

假设这一系列的处理都是在上帝类UIApplication中完成的。之所以叫“上帝类(god class)”，是因为它“无所不能”、“包含所有”。在面向对象的软件设计中，“上帝类”不是很友好，需要重构。在编程过程中，要尽量避免使用上帝类，因为上帝类是高耦合的，职责不清，难以维护。我们需要“去除上帝类”，把看似功能很强且很难维护的类，按照职责将它的属性或方法分派到各自的类中或分解成功能明确的类。

幸运的是，苹果没有把UIApplication类设计成“上帝类”，而是将它们分割到两个不同的角色类中：其中一个扮演框架类角色，框架类具有通用、可重复使用、与具体应用无关等特点；另一个扮演应用相关类的角色，应用相关类与具体应用有关。由于受到框架类的控制，应用相关类常常被设计为“协议”，在Java中称为“接口”。开发人员需要在具体的应用中实现这个“协议”。

如图3-3所示，将一些功能提取出来放在application:didFinishLaunchingWithOptions:和applicationDidBecomeActive:方法中完成，定义在UIApplicationDelegate协议中，这样UIApplication类就变成了框架类。

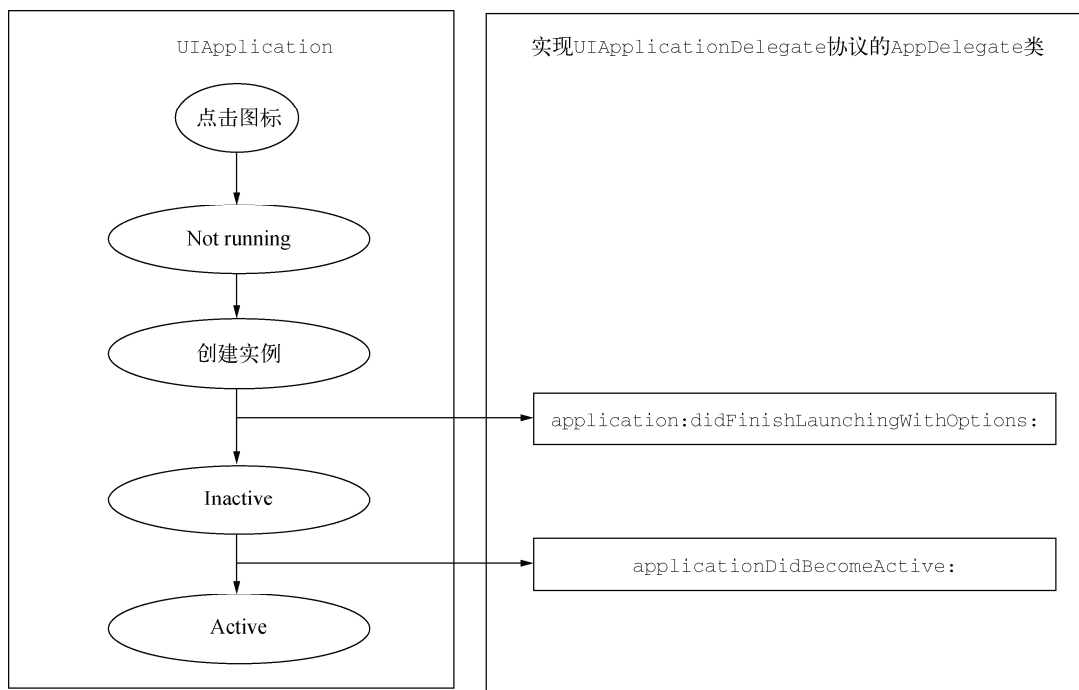


图3-3 去“上帝”化的非运行状态启动场景流程图

在具体使用时，需要实现UIApplicationDelegate协议。HelloWorld应用的类图如图3-4所示。

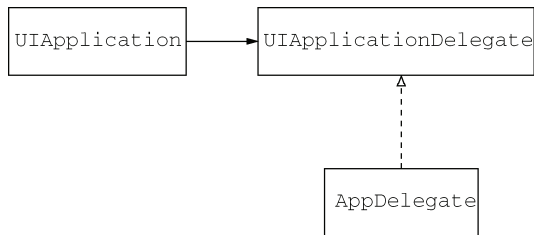


图3-4 去“上帝”化的HelloWorld应用类图

UIApplication不直接依赖于AppDelegate类，而是依赖于UIApplicationDelegate协议，这在面向对象软件

设计原则中叫做“面向接口的编程”。AppDelegate类实现协议UIApplicationDelegate，它是委托类。

委托是为了降低一个对象的复杂度和耦合度，使其能够更具通用性而将其中一些处理置于委托对象中的编码方式。通用类因为通用性（与具体应用的无关性）而变为框架类，框架类保持委托对象的指针，并在特定时刻向委托对象发送消息。消息可能只是通知委托对象做一些事情，也可能是对委托对象进行控制。

3.2.2 实现原理

下面我们通过一个案例介绍委托设计模式的实现原理和应用场景，重新绘制的委托设计模式类图如图3-5所示。

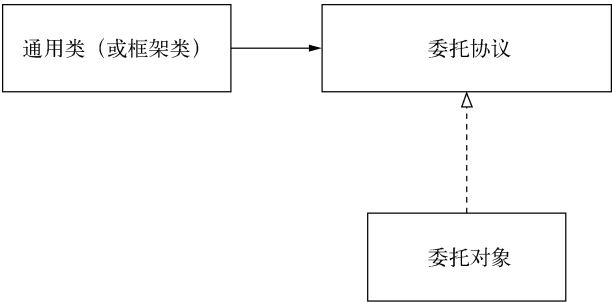


图3-5 委托设计模式类图

在古希腊有一个哲学家，他毕生只做三件事情：“睡觉”、“吃饭”和“工作”。为了更好地生活，提高工作效率，他会找一个徒弟，把这些事情委托给徒弟做。然而要成为他的徒弟，需要实现一个协议，协议要求能够处理“睡觉”、“吃饭”和“工作”的问题。三者的关系如类图3-6所示。

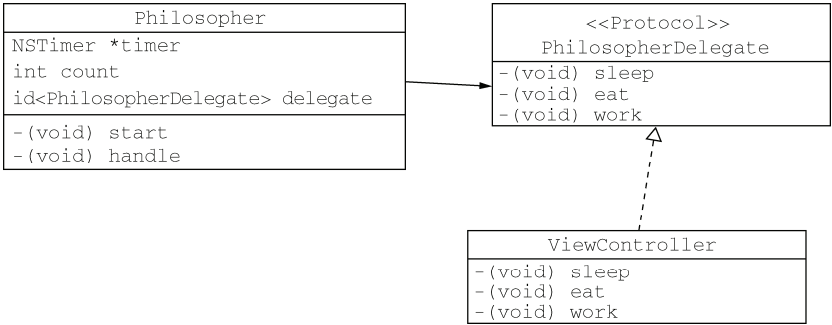


图3-6 委托设计模式哲学家案例类图

从图3-6所示的哲学家类图中可以看到，通用类（Philosopher）保持指向委托对象（ViewController）的“弱引用”（id<PhilosopherDelegate> delegate），委托对象（ViewController）就是哲学家的“徒弟”，它实现了协议PhilosopherDelegate。PhilosopherDelegate规定了3个方法：-(void) sleep、-(void) eat和-(void) work方法。

下面我们看看实现代码，委托协议PhilosopherDelegate.h的代码如下：

```
@protocol PhilosopherDelegate

@required
-(void) sleep;
-(void) eat;
-(void) work;

@end
```

可以看到，委托协议PhilosopherDelegate定义了3个方法。如果该委托协议没有.m文件，它的定义可以放在别的.h文件中。它的实现类就是委托类ViewController，相关代码如下：

```
//
//ViewController.h
//

@interface ViewController : UIViewController<PhilosopherDelegate>
@end
//
//ViewController.m
//
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    Philosopher *obj = [[Philosopher alloc ] init];
    obj.delegate = self;
    [obj start];
}
#pragma -- PhilosopherDelegate方法实现
- (void) sleep
{
    NSLog(@"sleep...");
}
- (void) eat
{
    NSLog(@"eat...");
}
- (void) work
{
    NSLog(@"work...");
}
@end
```

委托对象如何与通用类建立引用关系呢？我们通过viewDidLoad方法中的obj.delegate = self语句来指定委托对象和通用类间的引用关系。一般情况下，通用类由框架直接提供。在这个例子中，我们根据需要自己实现了通用类Philosopher。Philosopher.h的代码如下：

```
//
//Philosopher.h
//DelegatePattern
//
#import "PhilosopherDelegate.h"
@interface Philosopher : NSObject
{
    NSTimer *timer;
    int count;
}
@property (nonatomic, weak) id<PhilosopherDelegate> delegate;
- (void) start;
- (void) handle;
@end
```

在上述代码中，我们定义了delegate属性，它的类型是id<PhilosopherDelegate>，它可以保存委托对象的引用，其中属性weak说明是“弱引用”。这里使用弱引用方式是为了防止内存引用计数增加而导致委托对象无法释放的问题。Philosopher.m文件的代码如下：

```
//
//Philosopher.m
//DelegatePattern

#import "Philosopher.h"
@implementation Philosopher
```

```

@synthesize delegate;

-(void) start
{
    count= 0;
    timer = [NSTimer scheduledTimerWithTimeInterval:3.0
        target:self selector:@selector(handle)userInfo:nil repeats:YES];
}

-(void)handle
{
    switch (count)
    {
        case 0:
            [self.delegate sleep];
            count++;
            break;
        case 1:
            [self.delegate eat];
            count++;
            break;
        case 2:
            [self.delegate work];
            [timer invalidate];
            break;
    }
}
@end

```

在本例中，Philosopher模拟一些通用类发出调用，这个调用通过NSTimer每3秒发出一个，依次向委托对象发出消息sleep、eat和work。self.delegate是指向委托对象ViewController的指针，[self.delegate sleep]是调用ViewController中的sleep方法。

3.2.3 应用案例

我们以UITextFieldDelegate为例来说明一下委托的用法。UITextFieldDelegate是控件UITextField的委托，它主要负责响应控件事件或控制其他对象。除了UITextField，WebView和UITableView等控件也有相应的委托对象。

打开UITextFieldDelegate的API文档（如图3-7所示），可以发现其中有4个与编辑有关的方法，还有3个其他方法。

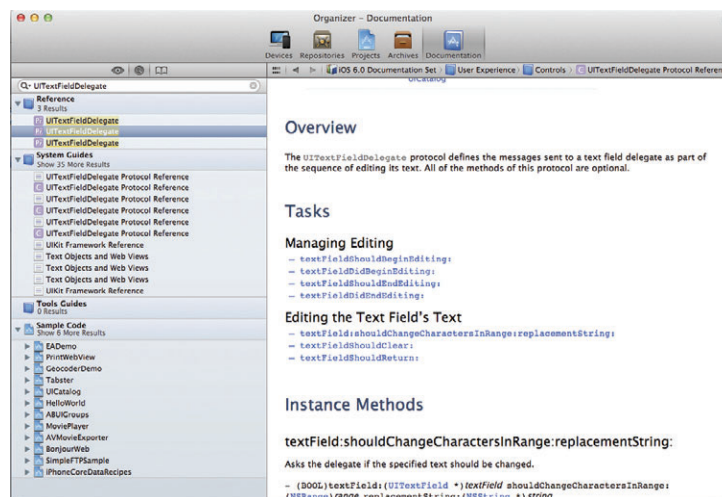


图3-7 UITextFieldDelegate的API文档

这里我们重点介绍在编辑过程中消息的发送以及UITextField控件与UITextFieldDelegate委托对象之间的交互过程，如图3-8所示。

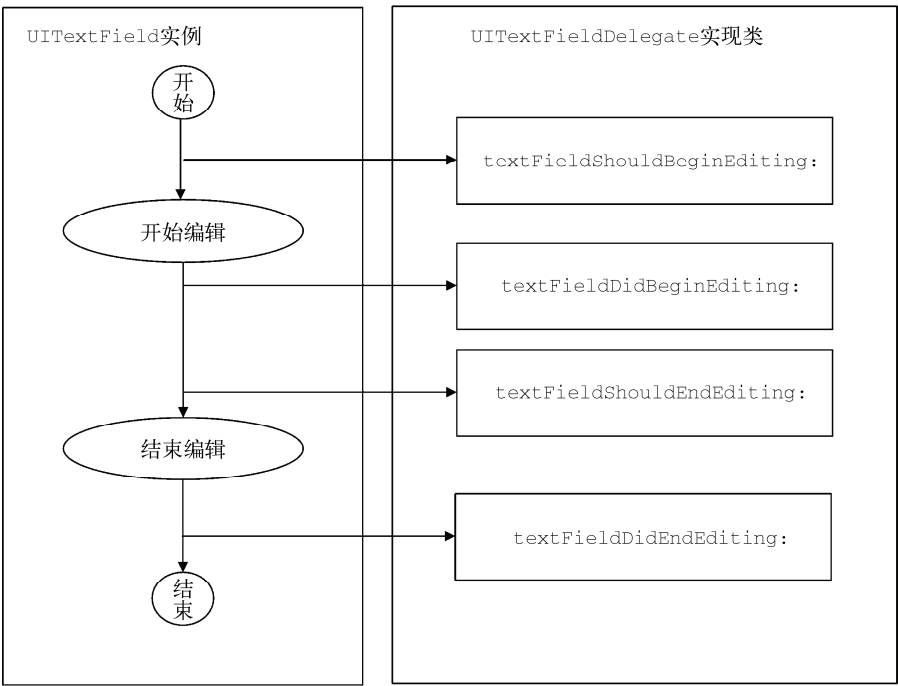


图3-8 UITextField控件与UITextFieldDelegate委托对象之间的交互过程

在文本框开始编辑前后，会分别发出消息`textFieldShouldBeginEditing:`和`textFieldDidBeginEditing:`，编辑结束前后会分别发出消息`textFieldShouldEndEditing:`和`textFieldDidEndEditing:`。

注意 委托消息命名有一定的约定性，如果是UITextField发出的消息，就以`textField`开头，后面跟3个词之一——`Should`、`Will`或`Did`。在使用`Should`消息时，应该返回一个布尔值，这个返回值用于确定委托是否会响应消息；当使用`Will`后缀时，没有返回值，表示改变前要做的事情；当使用`Did`后缀时，也没有返回值，表示改变之后要做的事情。这3种方法都会把发送消息的对象以参数的形式回传回来，例如`textFieldShouldBeginEditing:(UITextField *) textField`消息中的参数`textField`。

为了演示文本框编辑前后发生了什么，我们需要编写一个简单的文本框工程，如图3-9所示，其中界面中只包含一个文本框。

我们在视图控制器`ViewController`中实现`UITextFieldDelegate`，而`ViewController`是`UITextField`的委托对象。`ViewController.h`的代码如下：

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

@interface ViewController : UIViewController<UITextFieldDelegate>

@property (weak, nonatomic) IBOutlet UITextField *textField;

@end
```

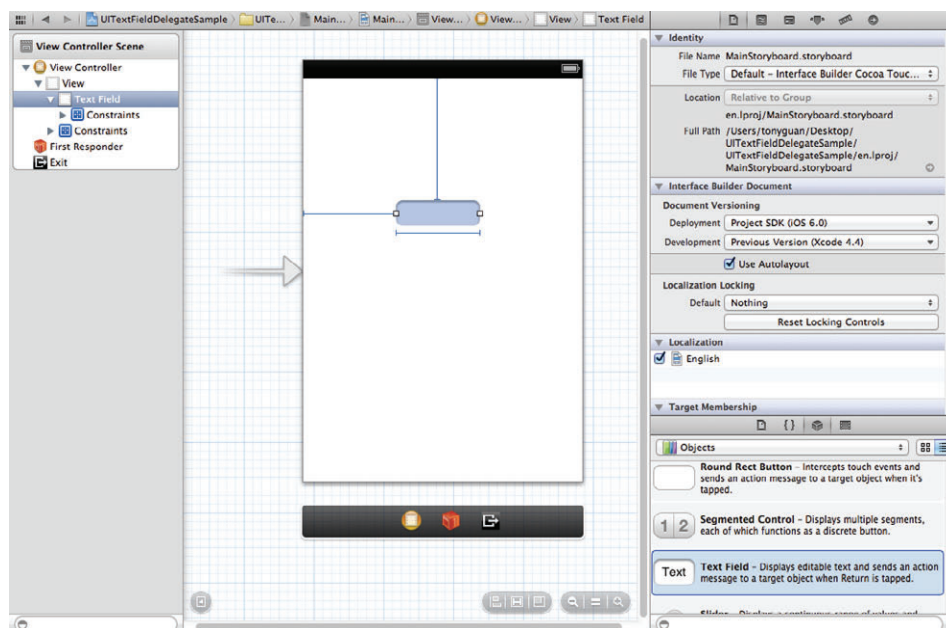


图3-9 文本框工程界面

在上述代码中，我们实现了UITextFieldDelegate协议，把UITextField定义为一个弱引用的“输出口”（“输出口”的概念我们将在第4章中详细介绍）。

ViewController.m的代码如下：

```
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.textField.delegate = self;
}

#pragma mark -- UITextFieldDelegate method
- (BOOL)textFieldShouldBeginEditing:(UITextField *)textField
{
    NSLog(@"call textFieldShouldBeginEditing:");
    return YES;
}

- (void)textFieldDidBeginEditing:(UITextField *)textField
{
    NSLog(@"call textFieldDidBeginEditing:");
}

- (BOOL)textFieldShouldEndEditing:(UITextField *)textField
{
    NSLog(@"call textFieldShouldEndEditing:");
    return YES;
}

- (void)textFieldDidEndEditing:(UITextField *)textField
{
    NSLog(@"call textFieldDidEndEditing:");
}

- (BOOL)textFieldShouldReturn:(UITextField *)textField
{
    NSLog(@"call textFieldShouldReturn:");
    [textField resignFirstResponder];
    return YES;
}

@end
```

在viewDidLoad方法中, `self.textField.delegate = self`语句极为重要, 它将委托对象ViewController分配给文本框对象。除了代码, 我们也可以通过Interface Builder工具进行连线分配。如图3-10所示, 打开故事板文件, 右击文本框控件, 从弹出的快捷菜单中, 将位于Outlets (输出口) 下面的delegate后面的圆圈用鼠标拖曳到View Controller上, 然后释放鼠标。

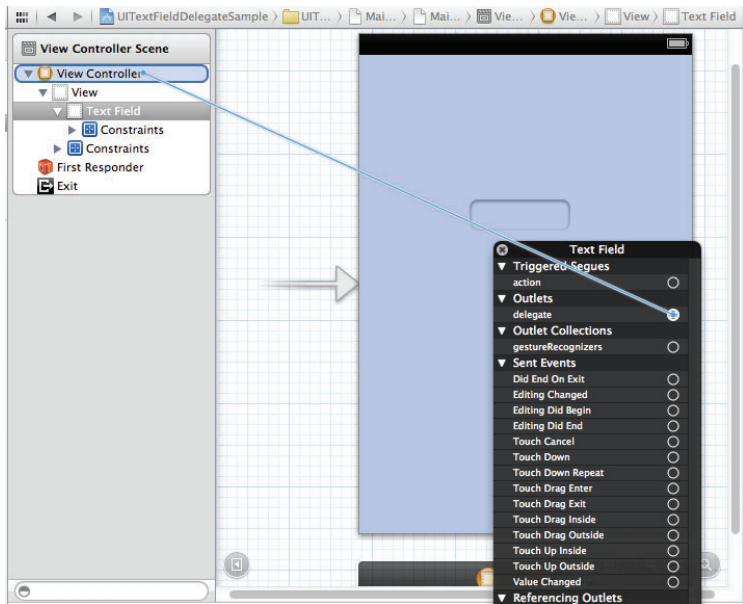


图3-10 定义委托输出口

运行代码, 输出的日志如下:

```
call textFieldShouldBeginEditing:
call textFieldDidBeginEditing:
```

输入完成后, 点击return键, 关闭键盘, 结束编辑状态, 此时在日志中的输出结果如下:

```
call textFieldShouldReturn:
call textFieldShouldEndEditing:
call textFieldDidEndEditing:
```

其中textFieldShouldReturn:是点击return键时发出的消息, 我们借助该消息通过textFielddesignFirstResponder方法关闭键盘。

对于一些更复杂的控件 (如UITableView), 除了需要实现委托协议外, 还需要实现数据源协议。数据源与委托一样, 都是委托设计模式的具体应用, 委托对象主要对控件对象的事件和状态变化作出响应, 而数据源对象是为控件对象提供数据。需要注意的是, 委托中的方法在实现时是可选的, 而数据源中的方法一般必须实现。

3.3 观察者模式

观察者 (Observer) 模式也叫发布/订阅 (Publish/Subscribe) 模式, 是 MVC (模型-视图-控制器) 模式的重要组成部分。

3.3.1 问题提出

天气一直是英国人喜欢讨论的话题, 而最近几年天气的变化也成为中国人非常关注的话题。我会根据天气预

报决定是坐地铁还是开车上班，我的女儿也会根据天气预报决定明天穿哪件衣服。于是我在移动公司为我的手机定制了天气预报短信通知服务，它的工作模型如图3-11所示。

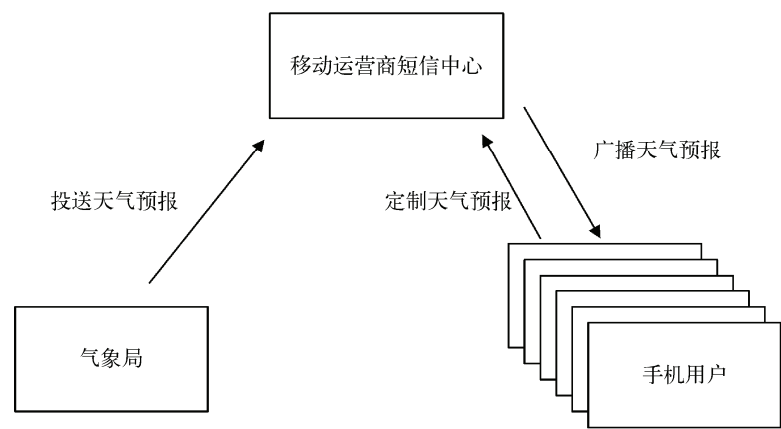


图3-11 定制天气预报短信通知服务

每天气象局将天气预报信息投送给移动运营商，移动运营商的短信中心负责把天气预报发送给定制过这项服务的手机。

在软件系统中，一个对象状态改变也会连带影响其他很多对象的状态发生改变。能够实现这一需求的设计方案有很多，但能够做到复用性强且对象之间匿名通信的，观察者模式是其中最为适合的一个。

3.3.2 实现原理

观察者模式的类图如图3-12所示。

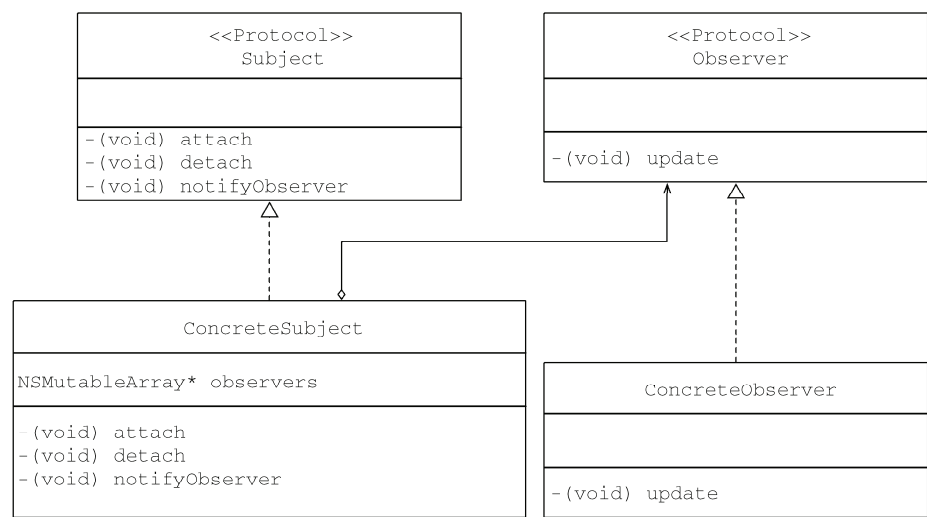


图3-12 观察者模式的类图

它有4个角色，具体如下所示。

❑ 抽象主题（**Subject**）。在Objective-C中，抽象主题是一个协议，它是一个观察者集合容器，定义了添加观察者（attach）方法、移除观察者（detach）方法和为所有观察者发送通知的方法（notifyObserver）。

□ 抽象观察者 (**Observer**)。在Objective-C中,抽象观察者是一个协议,它是一个更新(update)方法。

□ 具体观察者 (**ConcreteObserver**)。Observer协议的具体实现。

□ 具体主题 (**ConcreteSubject**)。Subject协议的具体实现。

引入Subject和Observer这两个协议后,不仅提高了系统的可复用性,还降低了耦合度。下面我们给出观察者模式中Observer和Subject的Objective-C实现代码。

抽象观察者(Observer)的实现代码如下:

```
//
//Observer.h
//ObserverPattern
//
@protocol Observer
@required
-(void)update;
@end

//
//Subject.h
//ObserverPattern
//
@class Observer;
@protocol Subject
@required
-(void)attach:(Observer*) observer;
-(void)detach:(Observer*) observer;
-(void)notifyObservers;
@end
```

具体观察者(ConcreteObserver)的实现代码如下:

```
//
//ConcreteObserver.h
//ObserverPattern
//
#import "Observer.h"
@interface ConcreteObserver : NSObject <Observer>
@end
//
//ConcreteObserver.m
//ObserverPattern
//
#import "ConcreteObserver.h"
@implementation ConcreteObserver
-(void)update
{
    NSLog(@"ConcreteObserver : %@",self);
}
@end
```

这里的update方法只是进行日志输出处理。

下面是具体主题(ConcreteSubject)的实现代码:

```
//
//ConcreteSubject.h
//ObserverPattern
//
#import "Subject.h"
@class Observer;
@interface ConcreteSubject : NSObject <Subject>
{
    NSMutableArray* observers;
}
@property (nonatomic ,strong) NSMutableArray* observers;
```

```

+(ConcreteSubject*)sharedConcreteSubject;
@end

//
//ConcreteSubject.m
//ObserverPattern
//
#import "ConcreteSubject.h"
@implementation ConcreteSubject

@synthesize observers;
static ConcreteSubject *sharedConcreteSubject = nil;
+(ConcreteSubject*)sharedConcreteSubject
{
    static dispatch_once_t once;
    dispatch_once(&once, ^{
        sharedConcreteSubject = [[self alloc] init];
        sharedConcreteSubject.observers = [[NSMutableArray alloc] init];
    });
    return sharedConcreteSubject;
}
-(void)attach:(Observer*) observer
{
    [self.observers addObject:observer];
}
-(void)detach:(Observer*) observer
{
    [self.observers removeObject:observer];
}
-(void)notifyObservers
{
    for (id obs in self.observers)
    {
        [obs update];
    }
}
@end

```

因为ConcreteSubject只需要一个实例，所以我们采用单例设计模式实现。观察者模式还可以有其他变形，若要深入了解，可以参考GoF。

3.3.3 通知机制和KVO机制

在Cocoa Touch框架中，观察者模式的具体应用有两个——通知（notification）机制和KVO（Key-Value Observing）机制，下面简要介绍这两种机制。

1. 通知机制

通知机制与委托机制不同的是，前者是“一对多”的对象之间的通信，后者是“一对一”的对象之间的通信。

说明 在iOS中，通知一词多次出现过，归纳一下主要有广播通知（broadcast notification）、本地通知（local notification）和推送通知（push notification），本节介绍的是广播通知。事实上，除了名字相似，广播通知与其他两个通知完全不同：广播通知是Cocoa Touch框架中实现观察者模式的一种机制，它可以在一个应用内部的多个对象之间发送消息；本地通知和推送通知中的“通知”是给用户一种“提示”，它的“提示”方式有警告对话框、发出声音、振动和在应用图标上显示数字等。在计划时间达到时，本地通知由本地iOS发出。推送通知由第三方案程序发送给苹果的远程服务器，再由远程服务器推送给iOS的特定应用。

如图3-13所示，在通知机制中对某个通知感兴趣的所有对象都可以成为接收者。首先，这些对象需要向通知中心（NSNotificationCenter）发出addObserver:selector:name:object:消息进行注册，在投送对象投送通知给通知中心时，通知中心就会把通知广播给注册过的接收者。所有的接收者都不知道通知是谁投送的，更不关心它的细节。投送对象与接收者是一对多的关系。接收者如果对通知不再关注，会给通知中心发出removeObserver:name:object:消息解除注册，以后不再接收通知。

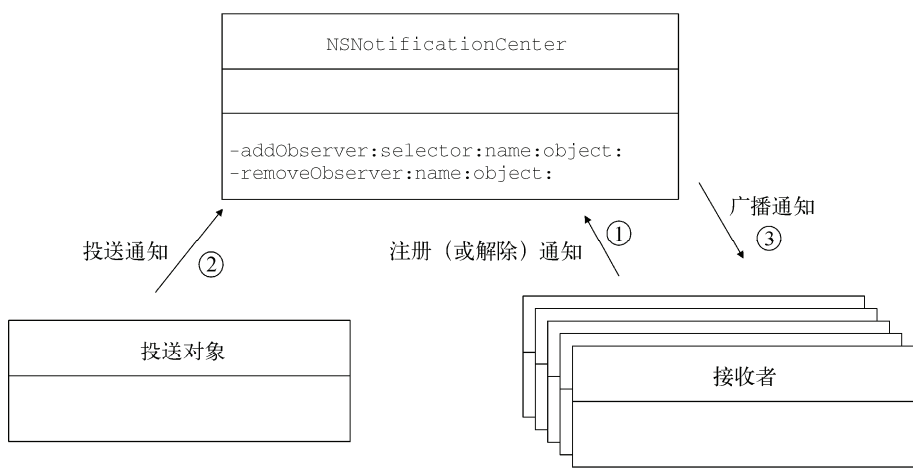


图3-13 通知机制图

下面我们介绍一下通知机制的使用过程。这里我们用图3-14所示的Utility Application模板创建一个工程。

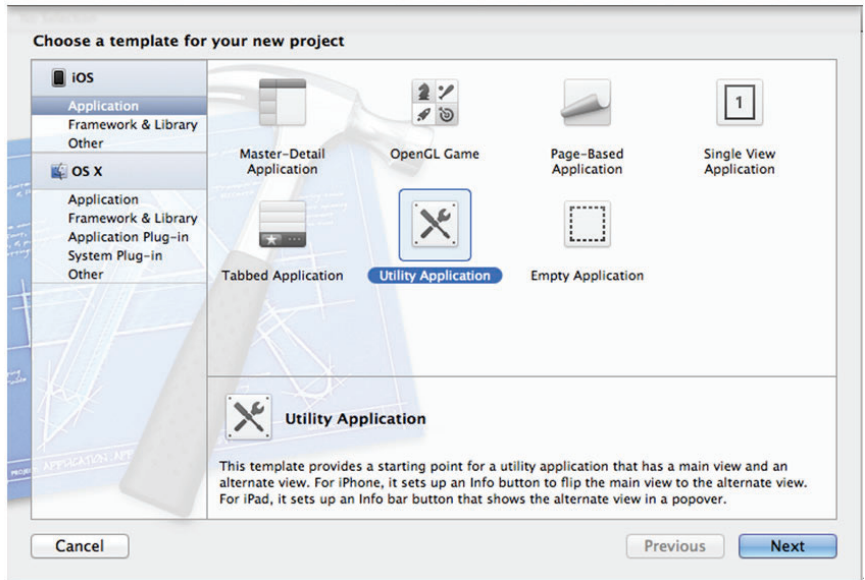


图3-14 选择Utility Application模板

用Utility Application的iPhone工程模板所构建的应用程序会有两个界面——主界面和翻转界面，如图3-15所示。为什么通知案例采用这种工程模板呢？这是因为它有两个视图控制器——主界面控制器MainViewController和翻转界面控制器FlipsideViewController。此外，还有应用程序委托对象AppDelegate，如图3-16所示。

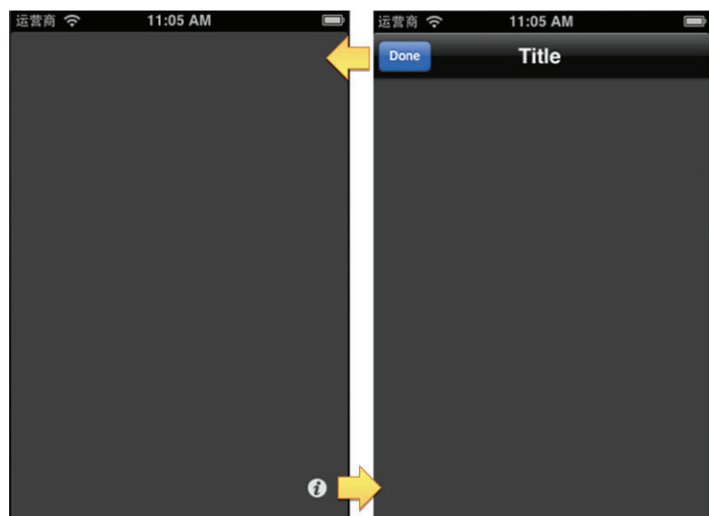


图3-15 使用Utility Application模板创建的工程的“主界面”和“翻转界面”

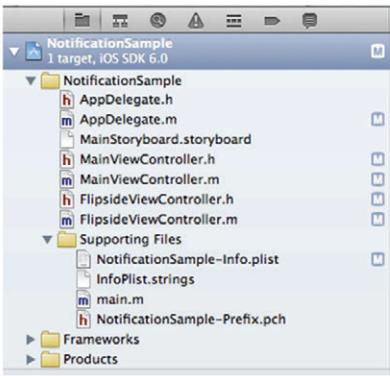


图3-16 工程生成文件

可以将两个视图控制器作为通知的接收者，应用程序委托对象作为通知投送对象，如图3-17所示。

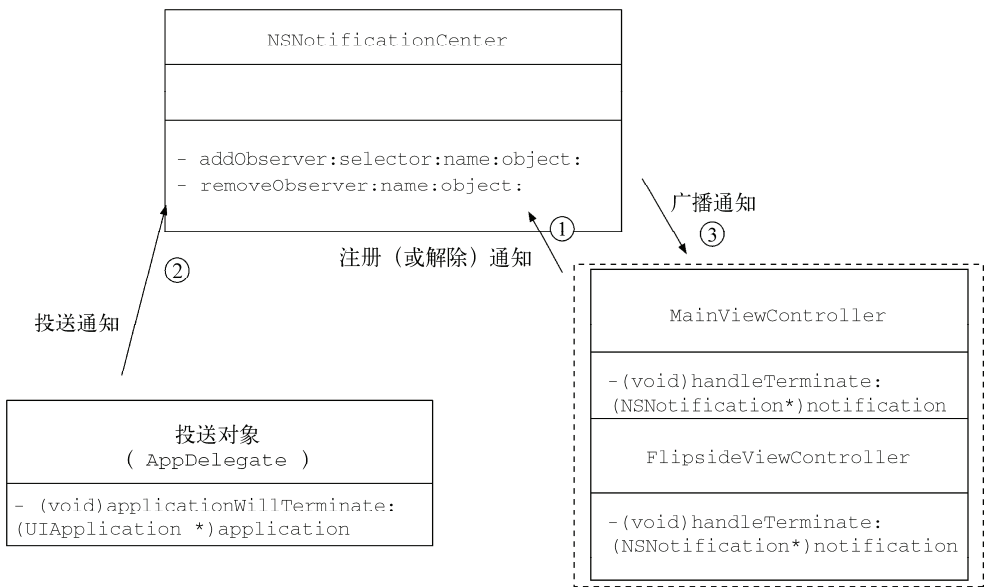


图3-17 通知机制图

在MainViewController和FlipsideViewController这两个视图控制器中，注册通知接收者的代码如下：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(handleTerminate:)
        name:@"AppWillTerminateNotification"
        object:nil];
}
```

NSNotificationCenter是单例模式,创建和获得共享实例的方法是defaultCenter,NSNotificationCenter的消息addObserver:selector:name:object:注册通知接收者,当接收到AppWillTerminateNotification通知时,就会调用handleTerminate:方法。

解除注册代码也类似,通过NSNotificationCenter发出removeObserver消息实现。对于一般的Objective-C对象,可以在dealloc方法中发出消息。对于视图控制器,也可以在didReceiveMemoryWarning方法或viewDidUnload方法中发出消息,具体代码如下:

```
- (void)dealloc
{
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```

MainViewController 和 FlipsideViewController 处理通知的方法基本相同,这里我们只看FlipsideViewController的处理通知方法handleTerminate:,其代码如下:

```
#pragma mark - 处理通知
-(void)handleTerminate:(NSNotification*)notification
{
    NSDictionary *theData = [notification userInfo];
    if (theData != nil) {
        NSDate *date = [theData objectForKey:@"TerminateDate"];
        NSLog(@"FlipsideViewController App Terminate Date: %@", date);
    }
}
```

这个方法可以接收一个NSNotification的参数。NSNotification类中有3个重要的属性——name、object和userInfo,这3个属性与通知中心投送方法中的参数有一定的对应关系,如图3-18所示。

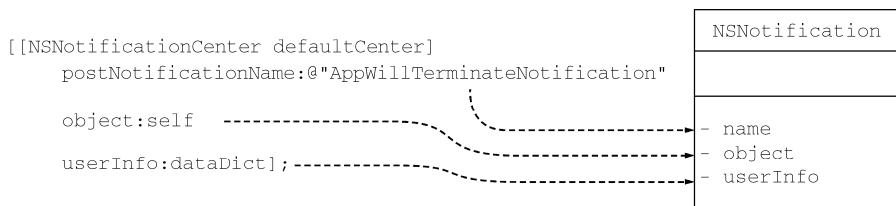


图3-18 NSNotification类和通知中心中投送方法参数的关系

其中name是通知的名字,object是投送通知时传递过来的对象,userInfo是投送通知时定义的字典对象,可借助于该参数传递数据。

如果我们想在应用程序终止时投送通知,需要重写AppDelegate中的方法applicationWillTerminate:,相关代码如下:

```
- (void)applicationWillTerminate:(UIApplication *)application
{
    NSDate *date = [NSDate date];
    NSDictionary *dataDict = [NSDictionary dictionaryWithObject:date
        forKey:@"TerminateDate"];
    [[NSNotificationCenter defaultCenter]
        postNotificationName:@"AppWillTerminateNotification"
        object:self
        userInfo:dataDict];
}
```

NSNotificationCenter的投送方法除了代码中所示外,还有另外一个重载方法:

```
[[NSNotificationCenter defaultCenter]
    postNotificationName:@"AppWillTerminateNotification" object:self];
```

它可以投送不带userInfo参数的通知,我们可以根据需要进行选择。还要注意的,object参数未必是self

对象，我们可以根据需要传递一个对象，如果接收者不需要，可以将其设为nil。

当我们运行代码，从“主界面”进入“翻转界面”之后，点击iPhone的Home键，应用就会终止（需要把NotificationSample-Info.plist中的Application does not run in background设为YES），在退出之前日志输出如下：

```
NotificationSample[2578:c07] MainViewController App Terminate Date:
    2012-08-31 04:21:06 +0000
NotificationSample[2578:c07] FlipsideViewController App Terminate Date:
    2012-08-31 04:21:06 +0000
```

我们在applicationWillTerminate:方法中投送一个应用终止通知。事实上，Cocoa Touch框架提供了通知，当应用终止事件发生时，由iOS系统自动投送。这些系统提供的通知在上一章介绍应用生命周期时已介绍了，具体内容大家可以参考表2-2。

使用iOS系统自动投送通知后，我们需要修改AppDelegate的applicationWillTerminate:方法，删除其中的代码：

```
- (void)applicationWillTerminate:(UIApplication *)application
{
    NSDate *date = [NSDate date];
    NSDictionary *dataDict = [NSDictionary dictionaryWithObject:date
        forKey:@"TerminateDate"];
    [[NSNotificationCenter defaultCenter]
        postNotificationName:@"AppWillTerminateNotification"
        object:self
        userInfo:dataDict];
}
```

修改MainViewController和FlipsideViewController中注册通知接收者的代码，具体如下：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(handleTerminate:)
        name:UIApplicationWillTerminateNotification
        object:nil];
}
```

这里把通知的name由@"AppWillTerminateNotification"修改为UIApplicationWillTerminateNotification，这是为了能够接收UIApplicationWillTerminateNotification通知。

除了应用生命周期的不同阶段有不同的通知外，很多控件也会在某些事件发生时投送通知，例如UITextField控件。在编辑过程的不同阶段，UITextField控件会分别发出如下通知：UITextFieldTextDidBeginEditingNotification、UITextFieldTextDidChangeNotification和UITextFieldTextDidEndEditingNotification。

2. KVO机制

KVO不像通知机制那样通过一个通知中心通知所有观察者对象，而是在对象属性变化时通知会被直接发送给观察者对象。图3-19为KVO机制解析图。

可以看到，属性发生变化的对象需要发出消息addObserver:forKeyPath:options:context:给注册观察者，使观察者关注它的某个属性的变化。当对象属性变化时，观察者就会接收到通知，观察者需要重写方法observeValueForKeyPath:ofObject:change:context:以响应属性的变化。

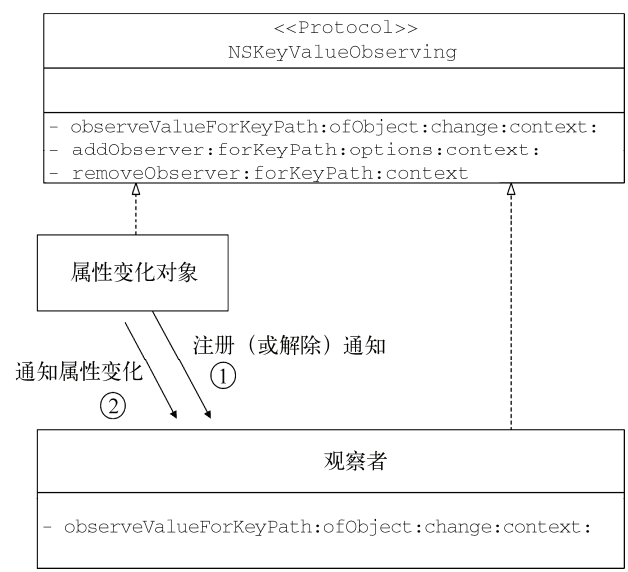


图3-19 KVO机制图

下面我们来看一个实际的案例。我们使用KVO机制来监视应用程序的状态变化。应用程序委托对象的AppDelegate.h文件的代码如下：

```
@interface AppDelegate : UIResponder <UIApplicationDelegate>
@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) NSString *appStatus;
@property (strong, nonatomic) AppStatusWatcher *watcher;
@end
```

其中appStatus属性用来记录应用程序状态的变化，AppStatusWatcher *watcher是定义ppStatusWatcher类型的观察者对象属性。AppDelegate.m的代码如下：

```
@implementation AppDelegate
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.watcher = [AppStatusWatcher new];

    [self addObserver:self.watcher forKeyPath:@"appStatus"
        options:NSKeyValueObservingOptionNew |
        NSKeyValueObservingOptionOld context:@"Pass Context"];
    self.appStatus = @"launch";
    return YES;
}
- (void)applicationWillResignActive:(UIApplication *)application
{
    self.appStatus = @"inactive";
}
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    self.appStatus = @"background";
}
- (void)applicationWillEnterForeground:(UIApplication *)application
{
    self.appStatus = @"inactive";
}
- (void)applicationDidBecomeActive:(UIApplication *)application
{
    self.appStatus = @"active";
}
```

```

}
- (void)applicationWillTerminate:(UIApplication *)application
{
    self.appStatus = @"terminate";
}
@end

```

其中application:didFinishLaunchingWithOptions:方法通过addObserver:forKeyPath:options:context:语句注册观察者，其中参数addObserver是要被关注的对象，forKeyPath是被关注对象的属性。options是为属性变化设置的选项，本例中被设定为NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld，把属性新旧两个值都传递给观察者。context参数是上下文内容，类型为(void*)（C语言形式的任何指针类型），因此，如果传递“空”，应该是“NULL”而非“nil”。

观察者AppStatusWatcher的代码如下：

```

@interface AppStatusWatcher : NSObject
@end

@implementation AppStatusWatcher

- (void)observeValueForKeyPath:(NSString*)keyPath ofObject:(id)object
  change:(NSDictionary*)change context:(void*)context
{
    NSLog(@"Property '%@' of object '%@' changed: %@",
          context: %@",keyPath,object,change,context);
}
@end

```

因为NSObject类实现了NSKeyValueObserving协议，所以只需声明AppStatusWatcher继承了NSObject类，而无需实现NSKeyValueObserving协议。

observeValueForKeyPath:ofObject:change:context:方法的observeValueForKeyPath参数是被关注的属性。ofObject是被关注的对象，change是字典类型，包含了属性变化的内容，这些内容与注册时属性变化设置的选项（options参数）有关。context是注册时传递的上下文内容。

第一次运行程序到界面时，会有两个状态的变化，日志结果如下：

```

KVOSample[6597:c07] Property 'appStatus' of object '
  <AppDelegate: 0x7521430>' changed: {
    kind = 1;
    new = launch;
    old = "<null>";
  }
context: Pass Context
KVOSample[6597:c07] Property 'appStatus' of object '
  <AppDelegate: 0x7521430>' changed: {
    kind = 1;
    new = active;
    old = launch;
  }
context: Pass Context

```

3.4 MVC 模式

MVC（Model-View-Controller，模型-视图-控制器）模式是相当古老的设计模式之一，它最早出现在Smalltalk语言中。现在，很多计算机语言和架构都采用了MVC模式。

3.4.1 MVC模式概述

MVC模式是一种复合设计模式，由“观察者”（Observer）模式、“策略”（Strategy）模式和“合成”（Composite）

模式等组成。MVC模式由3个部分组成，如图3-20所示，其中这3个部分的作用如下所示。

- ❑ **模型**。保存应用数据的状态，回应视图对状态的查询，处理应用业务逻辑，完成应用的功能，将状态的变化通知视图。
- ❑ **视图**。为用户展示信息并提供接口。用户通过视图向控制器发出动作请求，然后再向模型发出查询状态的申请，而模型状态的变化会通知给视图。
- ❑ **控制器**。接收用户请求，根据请求更新模型。另外，控制器还会更新所选择的视图作为对用户请求的回应。控制器是视图和模型的媒介，可以降低视图与模型的耦合度，使视图和模型的权责更加清晰，从而提高开发效率。

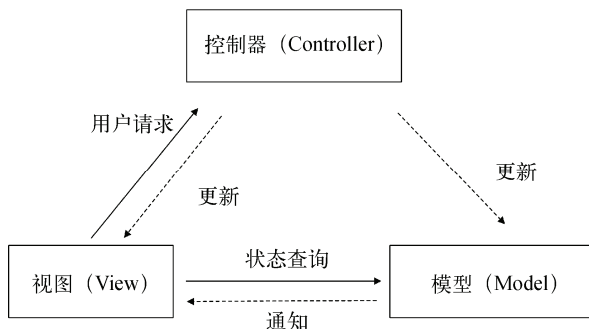


图3-20 MVC模式

对应于哲学中的“内容”与“形式”，在MVC模型中，模式是“内容”，它存储了视图所需要的数据，视图是“形式”，是外部表现方式，而控制器是它们的媒介。

3.4.2 Cocoa Touch中的MVC模式

在上一节中，我们讨论的是通用的MVC模式，而Cocoa和Cocoa Touch框架中的MVC模式与传统的MVC模式略有不同，前者的模型与视图不能进行任何通信，所有的通信都是通过控制器完成的，如图3-21所示。

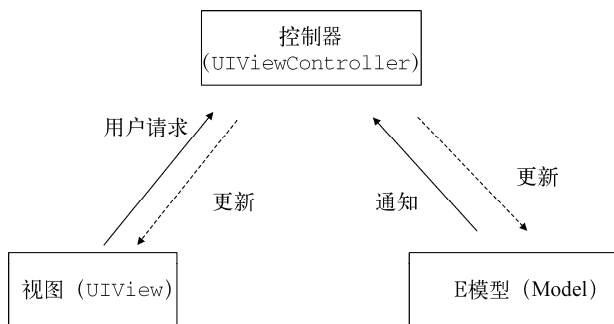


图3-21 Cocoa Touch的MVC模式

在Cocoa Touch框架的UIKit框架中，UIViewController是所有控制器的根类，如UITableViewController、UITabBarController和UINavigationController。UIView是视图和控件的根类，模型一般继承于NSObject的子类。

下面我们通过一个iOS的案例来分析Cocoa Touch中MVC模式的运作过程，这个案例的界面如图3-22所示。

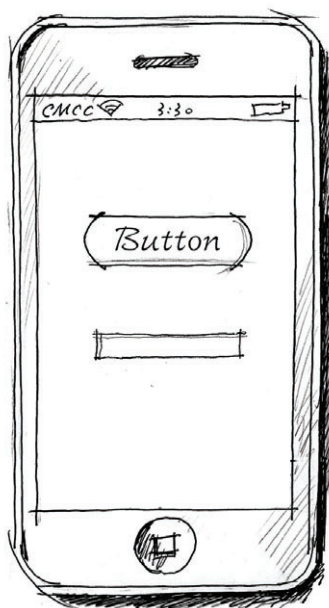


图3-22 MVC案例界面

这里我们就不过多介绍案例的编写过程，而是直接看一下代码。打开MVCSample工程，其中包括的文件有AppDelegate.h、AppDelegate.m、ViewController.h、ViewController.m和MainStoryboard.storyboard。

AppDelegate是应用程序委托对象，ViewController是视图控制器、MainStoryboard.storyboard是故事板文件。我们只看到了视图控制器，没有看到视图和模型。打开故事板MainStoryboard.storyboard文件，可以看到View Controller Scene如图3-23所示。

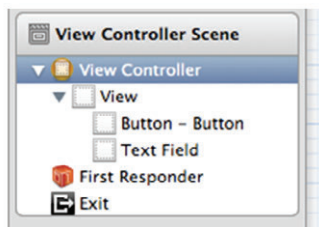


图3-23 View Controller Scene

打开View Controller，就可以看到View，其中直接使用了UIKit框架中的UIView，因此在MVCSample组中没有视图。此外，属于视图的还有Button和Text Field，它们是View的子视图。

那么，模型对象在哪儿呢？模型对象很特殊，其本质是视图的“数据”。Text Field输入的内容，Button上的便签，都可以说是模型，但是模型与视图一样，有的时候我们未必需要自己创建一个模型类。因此，我们做开发工作时，主要是编写视图控制器。下面我们看看视图控制器ViewController.h文件的代码：

```
@interface ViewController : UIViewController <UITextFieldDelegate>

@property (weak, nonatomic) IBOutlet UIButton *myButton;
@property (weak, nonatomic) IBOutlet UITextField *myTextField;
- (IBAction)myAction:(id)sender;

@end
```

可以看到，ViewController.h为两个控件myButton和myTextField定义了两个IBOutlet（输出口）类型的属性。因为要通过控制器更新这些视图（控件也属于视图），所以我们需要把这些视图定义成输出口类型的属性。

此外，ViewController.h还定义了- (IBAction)myAction:(id)sender方法以响应myButton按钮的触摸事件。该方法的返回类型是IBAction（动作事件），这说明该方法是可以响应控件事件的。关于IBOutlet（输出口）和IBAction（动作事件），我们将在第4章中详细介绍。

另外，ViewController还实现了UITextFieldDelegate协议，这样ViewController就变成了UITextField控件的委托对象，它们之间的运作关系如图3-24所示。

如图3-24所示，视图包含了myButton和myTextField两个控件。现在我们按照编号对图3-24解释如下。

- ① 当用户触摸myButton的时候，会触发ViewController中的- (IBAction)myAction:(id)sender方法。
- ② 视图控制器会实现一些控件委托和数据源协议，这要看具体的控件。在此案例中，ViewController实现了UITextFieldDelegate协议，在UITextFieldDelegate中定义了一些响应UITextField事件的方法。
- ③ 视图控制器通过属性myButton和myTextField来改变控件的状态。
- ④ 模型对象可以通过通知和KVO机制来通知数据的变化。
- ⑤ 视图控制器可以保存一个模型成员变量或属性，并通过它们改变模型的状态。

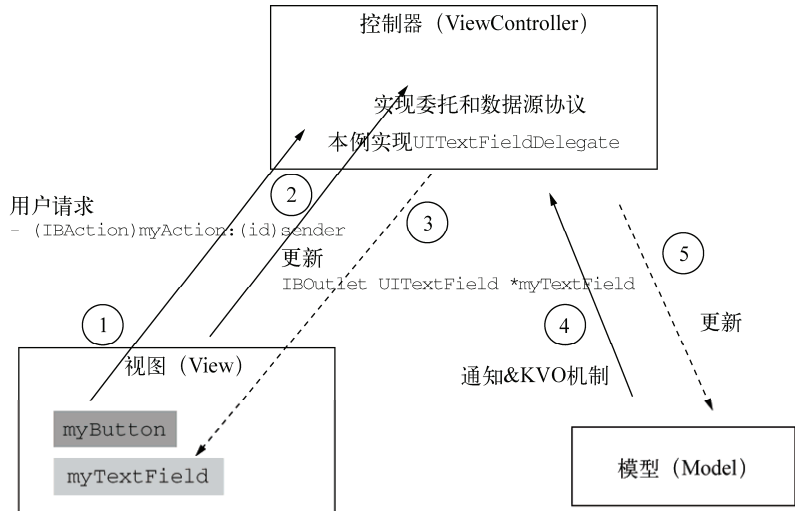


图3-24 MVC案例运作图

3.5 小结

本章首先介绍了软件设计模式的由来，然后重点介绍了iOS开发中常用的4种模式——单例模式、委托模式、观察者模式和MVC模式。在这4种模式的介绍中，我们介绍了每种模式的适用情况和实现原理，然后用具体案例来说明其用法。

视图和控件是应用的基本元素。在学习iOS之初，我们要掌握一些常用的视图和控件的特点及它们的使用方式。

4.1 视图“始祖”——UIView

在Objective-C中，NSObject是所有类的“根”类。同样，在UIKit框架中，也存在一个如此神奇的类UIView。从继承关系上看，UIView是所有视图的根，我们形象地称其为“始祖”。本节中，我们将向你详细解读UIView的神奇所在。

4.1.1 UIView“家族”

UIView“家族”大体分为“控件”和“视图”两类，二者均继承于UIView。UIView类的继承层次如图4-1所示。

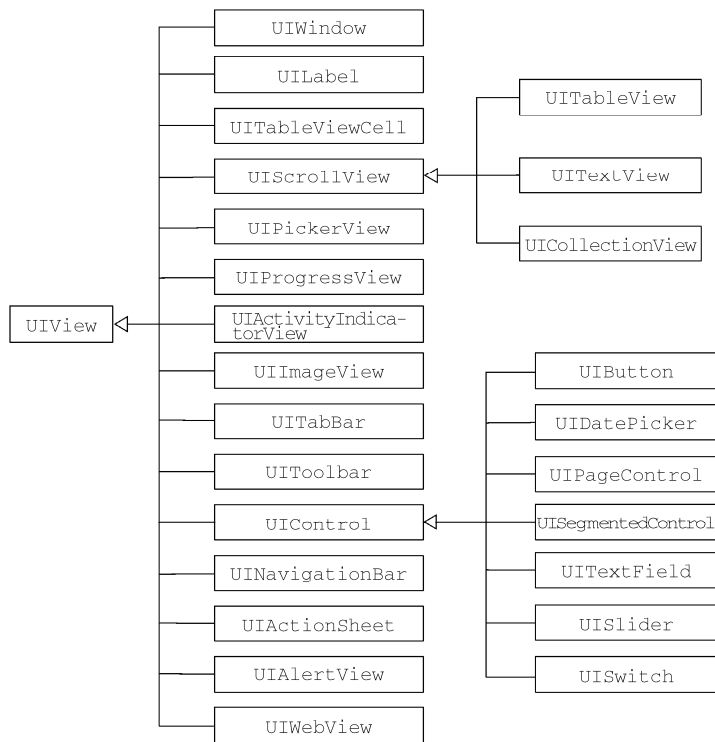



图4-1 UIView的继承层次图

UIControl类是控件类，其子类有UIButton、UITextField和UISlider等。之所以称它们为“控件类”，是因为它们都有能力响应一些高级事件。为了查看这些事件，我们可以在Interface Builder中拖曳一个UIButton到设计界面，然后选中这个Button，单击右上角的按钮，打开连接检查器，如图4-2所示。

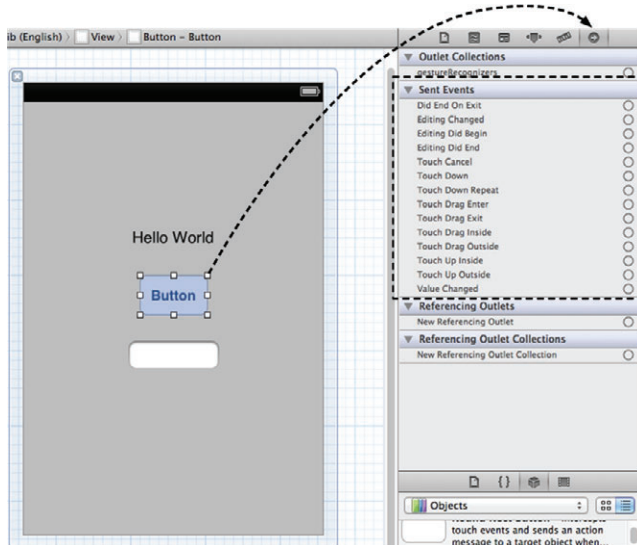


图4-2 UIButton的事件

其中Send Events栏中的内容就是UIButton相对应的高级事件。UIControl类以外的视图没有这些高级事件，这可以借助HelloWorld工程中的Label控件验证一下。选中UILabel控件，打开连接检查器，如图4-3所示。可以发现UILabel的连接检查器中没有Send Events栏，即没有高级事件，不可以响应高级事件。

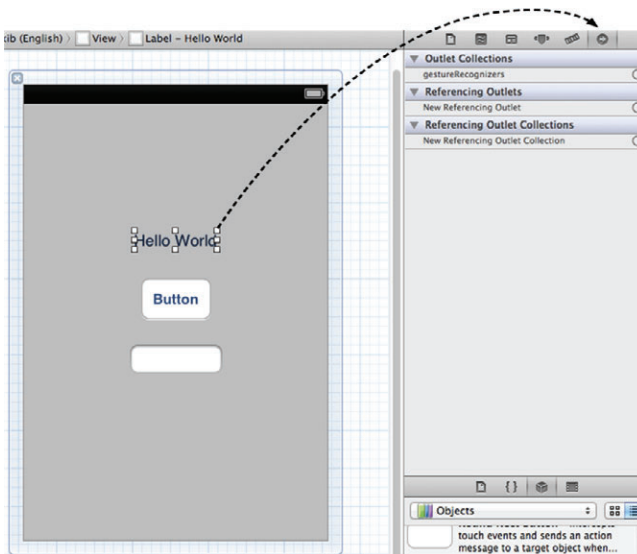


图4-3 UILabel没有高级事件

事实上，视图也可以响应事件，但这些事件比较低级，需要开发人员自己进行处理。很多手势的开发都以这些低级事件为基础的。

4.1.2 应用界面的构建层次

图4-4是一个应用界面的构建层次图，该应用有一个UIWindow，其中包含一个UIView根视图。根视图下又有3个子视图——Button1、Label2和UIView(View2)，其中子视图UIView(View2)中存在一个按钮Button3。

一般情况下，应用中只包含一个UIWindow。从视图构建层次上讲，UIWindow包含了一个根视图UIView。根视图一般也只有一个，放于UIWindow中。根视图的类型决定了应用程序的类型。图4-4中各对象间的关系如图4-5所示。

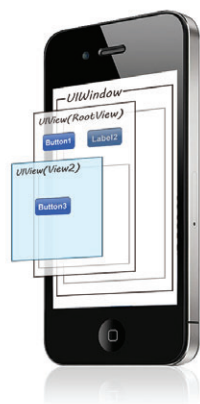


图4-4 应用界面的构建层次图

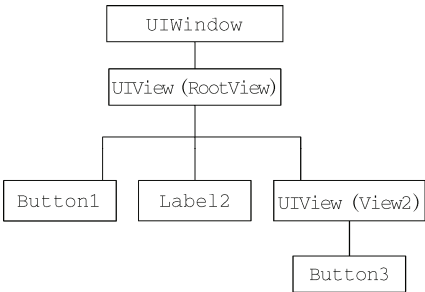


图4-5 各对象间的关系

应用界面的构建层次是一种树形结构，UIWindow是“树根”，根视图是“树干”，其他对象为树冠。在层次结构中，上下两个视图是“父子关系”。除了UIWindow，每个视图的父视图有且只有一个，子视图可以有多个。它们间的关系涉及3个属性，如图4-6所示。

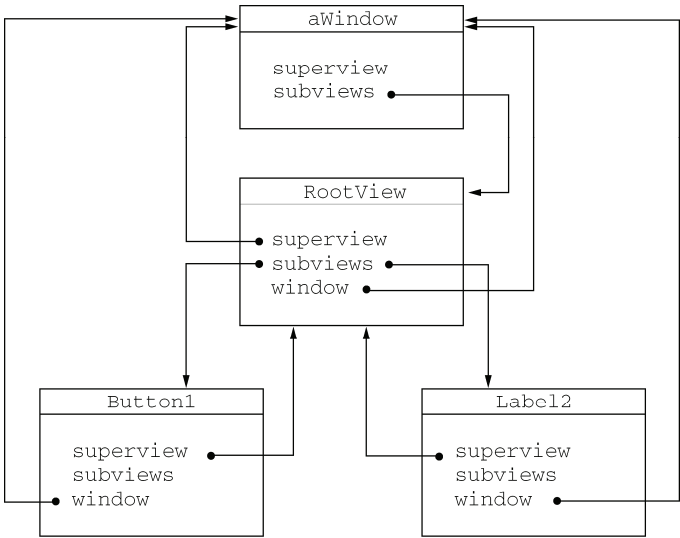


图4-6 视图中的superview、subviews和window属性

下面简要介绍这3个属性的含义。

❑ **Superview**。获得父视图对象。

- ❑ **Subviews**。获得子视图对象集合。
- ❑ **Window**。获得视图所在的UIWindow对象。

4.1.3 视图分类

为了便于开发，苹果将UIKit框架中的视图分成以下几个类别。

- ❑ **控件**。继承自UIControl类，能够响应用户高级事件。
- ❑ **窗口**。它是UIWindow对象。一个iOS应用只有一个UIWindow对象，它是所有子视图的“根”容器。
- ❑ **容器视图**。它包括了UIScrollView、UIToolbar以及它们的子类。UIScrollView的子类有UITextView、UITableView和UICollectionView，在内容超出屏幕时，它们可以提供水平或垂直滚动条。UIToolbar是非常特殊的容器，它能够包含其他控件，一般置于屏幕底部，特殊情况下也可以置于屏幕顶部。
- ❑ **显示视图**。用于显示信息，包括UIImageView、UILabel、UIProgressView和UIActivityIndicatorView等。
- ❑ **文本和Web视图**。提供了能够显示多行文本的视图，包括UITextView和UIWebView，其中UITextView也属于容器视图，UIWebView是能够加载和显示HTML代码的视图。
- ❑ **导航视图**。为用户提供从一个屏幕到另外一个屏幕的导航（或跳转）视图，它包括UITabBar和UINavigationController。
- ❑ **警告框和操作表**。用于给用户提供一种反馈或者与用户进行交互。UIAlertView视图是一个警告框，它会以动画形式弹出来；而UIActionSheet视图给用户提供可选的操作，它会从屏幕底部滑出。

注意 在后面章节中，很多视图（如UILabel、文本视图和进度条等）并未继承UIControl类，但我们也习惯称为“控件”，这是开发中约定俗成的一种常用归类方式，与严格意义上的概念性分类有差别。

4.2 标签控件和按钮控件

标签控件和按钮控件是两个常用的控件，下面我们通过一个具备用户交互功能的工程进一步学习这两个控件。另外，鉴于用户交互涉及动作和输出口，本节也将简单介绍动作和输出口的用法。

该案例的设计原型草图如图4-7所示，其中包含一个标签和一个按钮，当点击按钮的时候，标签文本会从初始的Label1替换为HelloWorld。

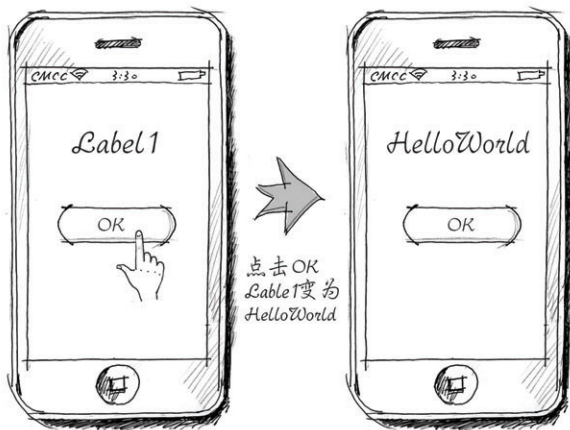


图4-7 设计原型草图

4.2.1 标签控件

使用Single View Application模板创建一个名为Label_ButtonSample的工程（具体创建过程请参见2.1.1节）。打开MainStoryboard.storyboard文件，从对象库中拖曳一个Label控件（其属性检查器如图4-8所示），双击该控件，将其文本设置为Label1。

提示 若无特别说明，本书中所有的案例都采用ARC和故事板技术。

由图4-8可以看出，标签的属性检查器包括Label和View两个组。Label组主要是文本相关的属性，而View组主要是从视图的角度对控件进行设置。

前文提到过，我们可以通过双击或者属性来实现Label控件的文本输入，这里的属性指的就是Label下的Text属性。当然，我们也可以的代码来实现文本的编辑。

需要说明的是，对象库中包含了控制器、基本控件、高级控件和手势等很多对象。随着版本的升级，对象库还在不断扩充和完善，短时间内可能无法找到指定的控件，此时我们可以借助对象库下方的搜索栏来查找，如图4-9所示。

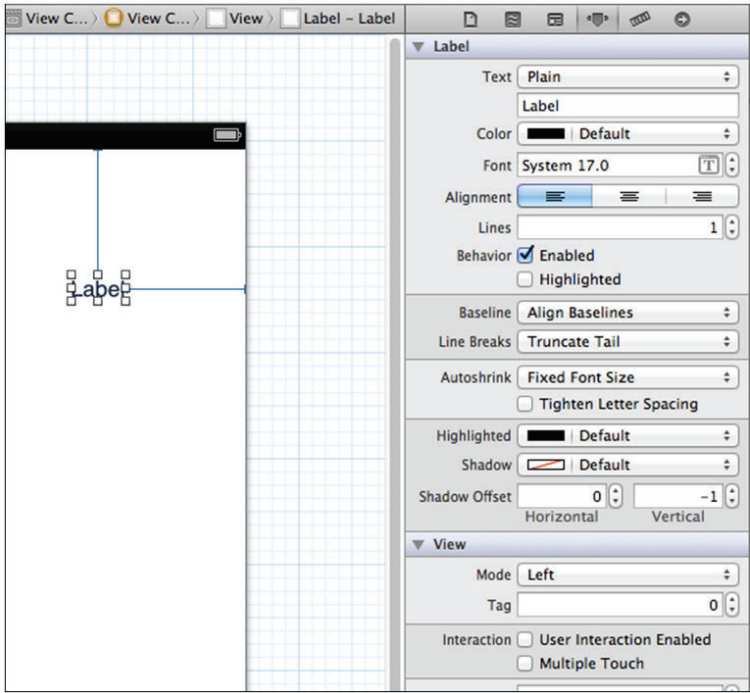


图4-8 属性检查器

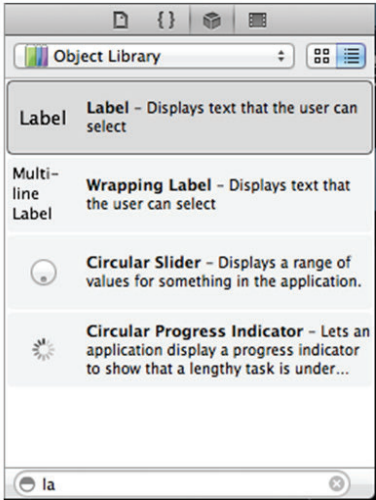


图4-9 对象库搜索栏

4.2.2 按钮控件

从对象库中拖曳一个Button控件并将其摆放到标签的正下方，如图4-10所示。

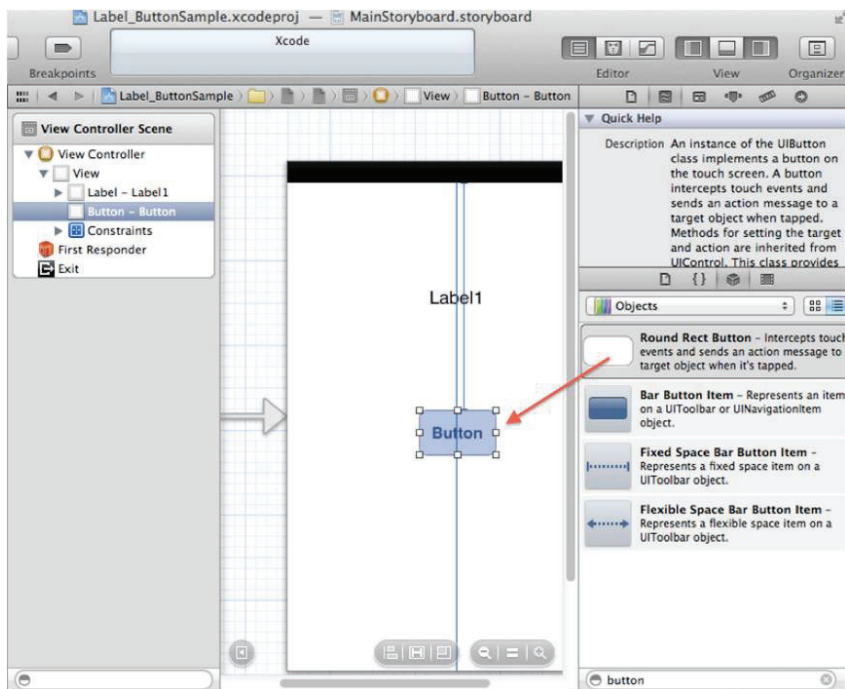


图4-10 摆放按钮控件

双击按钮，输入文本OK。现在按钮的状态是默认状态，我们可以运行一下，看看效果。

为了美观，往往还要通过属性检查器优化一下按钮。打开其属性检查器，如图4-11左图所示，单击Type下拉列表（如图4-11右图所示），其中各选项的含义如下所示。

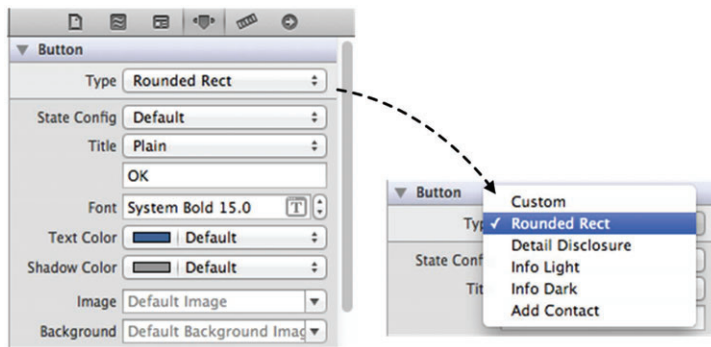


图4-11 按钮的Type属性

- ❑ Custom。自定义类型。如果我们不喜欢圆角按钮，可以使用该类型。
- ❑ Rounded Rect。系统默认属性，表示该按钮为圆角矩形。
- ❑ Detail Disclosure。细节展示按钮 ⓘ，主要用于表视图中的细节展示。
- ❑ Info Light和Info Dark。这两个是信息按钮，用于实用性应用程序。
- ❑ Add Contact。添加联系人按钮 ⓘ。

State Config下拉列表中有4种状态，分别是Default（默认）状态、Highlighted（高亮）状态、Selected（选择）状态和Disabled（不可用）状态，如图4-12所示。

选择不同的State Config选项，可以设置不同状态下的属性。

如果希望点击按钮时按钮中央会高亮显示，我们可以勾选Drawing中的Shows Touch On Highlight复选框，如图4-13所示。

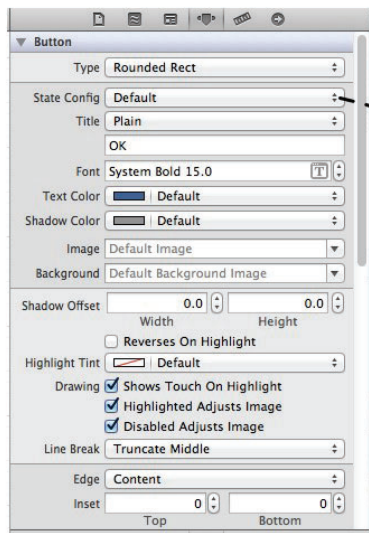


图4-12 按钮的State Config属性

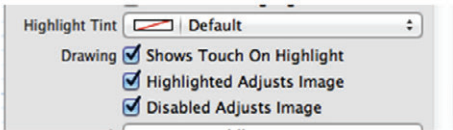
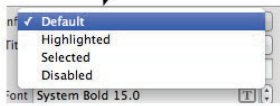



图4-13 高亮状态的设置

是点击按钮时的高亮效果，其中按钮中央会出现一个光圈。

UIKit至少有两种按钮：一种是UIButton类的普通按钮，该按钮可以有文字，也可以有图片；另一种是放置于工具栏或导航栏中的UIBarButtonItem，它虽然可以当按钮用，但是从类的继承关系上看，它不是UIView的子类，详情可参见4.8节。

4.2.3 动作和输出口

为了将事件和控件联系到一起，我们引入了动作和输出口的概念，它们的作用机制如图4-14所示。

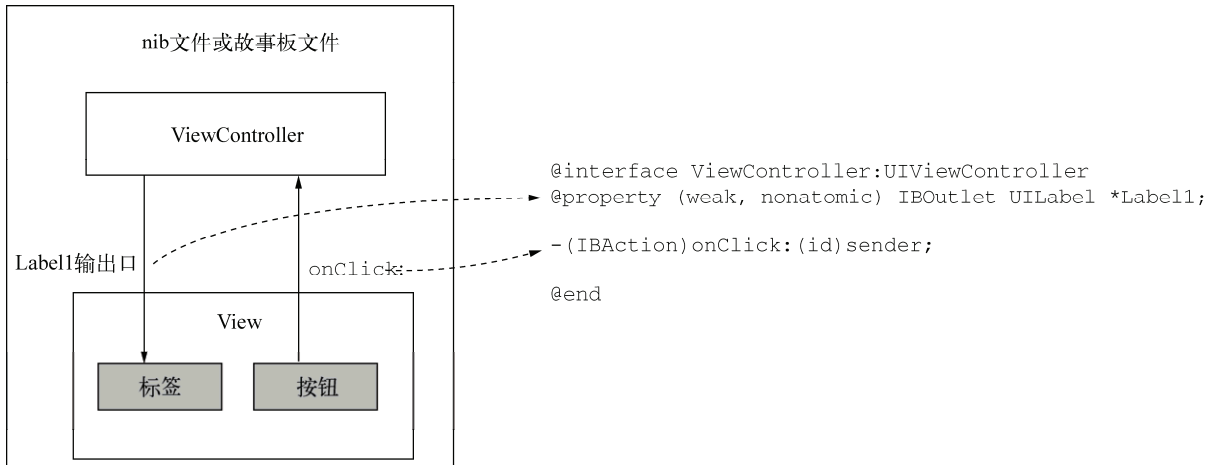


图4-14 动作和输出口的作用机制

动作是为了响应一个控件的事件而定义的方法，返回值类型为IBAction。此处列举一个动作的代码：

```
- (IBAction)onClick:(id)sender
```

该方法的返回类型是IBAction，说明这个方法是一个动作。sender是参数，是事件源，是发出事件的控件对象，可以省略如下：

```
- (IBAction)onClick
```

为了使控件的某个事件与定义的动作关联在一起，我们可以通过Interface Builder或者代码建立关联，本章中我们重点使用第一种方式。

为了能访问控件，我们需要为其定义输出口，关键字为IBOutlet。输出口可以在声明控件变量时声明，也可以在定义控件属性时声明。

在声明控件变量时声明输出口的代码如下：


```
@interface ViewController : UIViewController {
    IBOutlet UILabel * label;
}
```

在定义控件属性时声明输出口的代码如下：

```
@property (weak, nonatomic) IBOutlet UILabel *Label1;
```

跟动作一样，我们也需要为输出口和控件建立联系，这也可以通过Interface Builder或者代码来实现。

提示 Interface Builder设计器就是Interface Builder，在Xcode 4之后被集成到Xcode工具中。打开故事板或者xib文件，就会自动打开Interface Builder设计器。

为了使按钮能够控制标签，我们需要给标签定义并连接输出口，给按钮实现动作。点击左上角第一组按钮中的“打开辅助编辑器”按钮，打开如图4-15所示的界面。

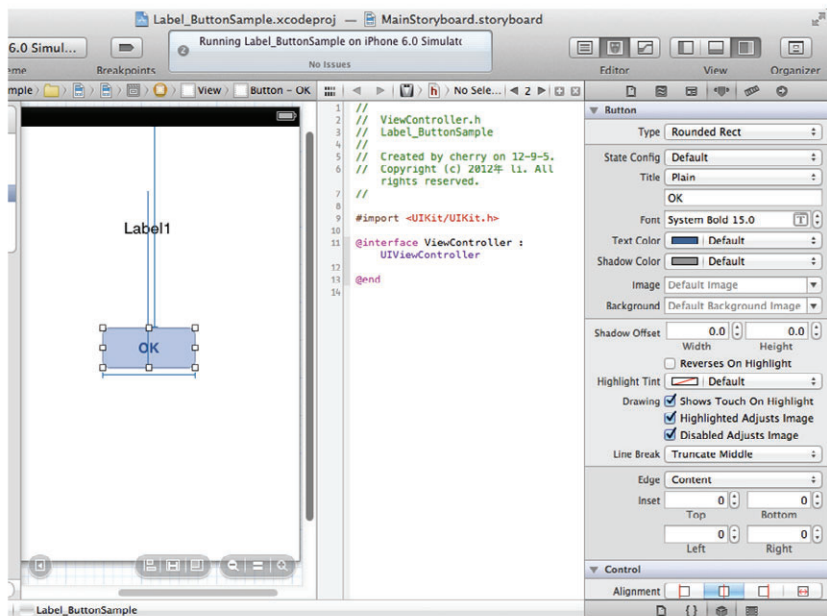


图4-15 辅助编辑器

选中标签，同时按住control键，将标签Label1拖曳到如图4-16所示的位置。

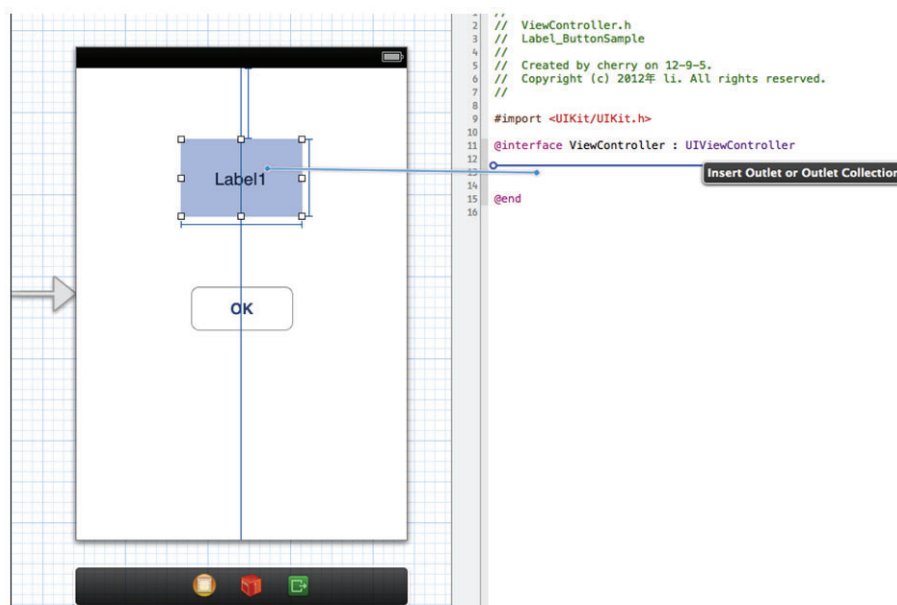


图4-16 拖曳标签Label1

释放鼠标，会弹出一个对话框。在Connection栏中选择Outlet，将输出口命名为Label1，如图4-17所示。

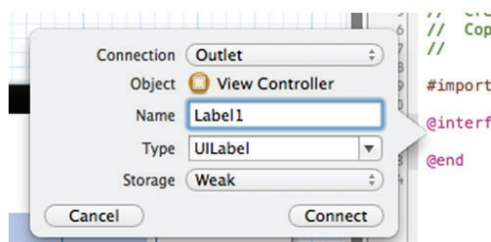


图4-17 设置输出口

点击Connect按钮，右边的编辑界面将自动添加如下代码：

```
@property (weak, nonatomic) IBOutlet UILabel *Label1;
```

对OK按钮进行同样的操作。在图4-15所示的对话框中选择Action并将其命名为onClick，其他选项用默认值就可以。点击Connect按钮，会生成如下代码：

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

现在我们进入ViewController.m文件，实现onClick:方法，具体代码如下：

```
- (IBAction)onClick:(id)sender {
    self.Label1.text = @"HelloWorld";
}
```

此时点击OK按钮，标签的文本内容从原来的Label1成功切换为Hello World。

4.3 TextField 控件和 TextView 控件

与Label控件一样，TextField和TextView控件也是文本类控件，是可以编辑文本内容的。

在编辑方面，三者都可以利用代码、双击控件和属性检查器中的Text属性来实现，但是TextField和TextView比Label多了一个键盘的使用。另外，TextField和TextView控件还各有一个委托协议。考虑到这些，我们将二者单列在一节。下面我们通过图4-18所示的案例向大家展示TextField控件和TextView控件的用法，其中包括两个标签、一个TextField和一个TextView。该案例的功能是向用户展示书的简介，TextField展示书名，TextView展示简介的内容。在TextField和TextView进入编辑状态时，键盘会从屏幕下方滑出来，点击return键关闭键盘。

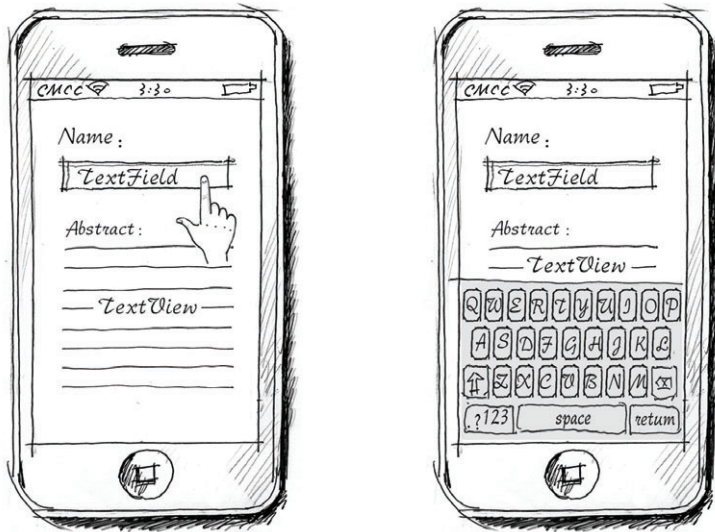




图4-18 设计原型草图

4.3.1 TextField控件

在UIKit框架中，TextField控件由UITextField类创建。此外，它还有对应的UITextFieldDelegate委托协议。委托可以帮助响应事件处理。TextField继承了UIControl，隶属于真正的“控件”，而TextView继承了UIScrollView。下面我们使用Single View Application模板创建一个名为 TextField_TextViewSample的工程。

打开MainStoryboard.storyboard设计界面，从对象库中拖曳两个标签控件到界面，分别将其命名为Name:和Abstract:。在Name:标签下摆放一个TextField。打开TextField属性检查器，在Placeholder属性中输入enter your book name作为提示，运行时该文本是浅灰色，当有输入动作时文本消失。可以利用TextField后面的清除按钮（Clear Button）清除TextField的内容，如图4-19所示。

现在我们就为TextField添加清除按钮：打开TextField的属性检查器，进入Clear Button属性的下拉列表，从中选择Is always visible，如图4-20所示。



图4-19 TextField的清除按钮

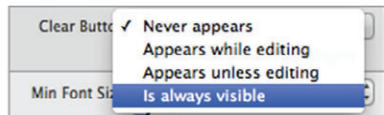


图4-20 选择清除按钮属性

4.3.2 UITextView控件

TextView 是一个可展示和编辑多行文本的控件，由 UITextView 类创建。TextView 控件有对应的 UITextViewDelegate 委托协议，我们可以借助委托来响应事件。

回到 Interface Builder 设计界面，在第二个标签 Abstract: 下面放置一个 UITextView 控件。打开视图控制器 ViewController.h 文件，代码如下：

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

@interface ViewController : UIViewController<UITextFieldDelegate, UITextViewDelegate>

@end
```

在 ViewController.h 文件中，UIViewController 实现了两个委托 UITextFieldDelegate 和 UITextViewDelegate。我们需要将委托对象 ViewController 分配给 UITextView 和 UITextField 控件的委托属性 delegate，这可以通过代码或者 Interface Builder 设计器来实现。这里我们用 Interface Builder 设计器进行分配，具体实现过程如下。

在 Interface Builder 中打开故事板文件，右击 UITextView 控件，弹出的快捷菜单如图 4-21 所示，用鼠标拖曳 Outlets→delegate 后面的小圆点到左边的 View Controller 上。以同样的方式将 UITextField 控件 Outlets →delegate 后面的小圆点拖曳到左边的 View Controller 上。

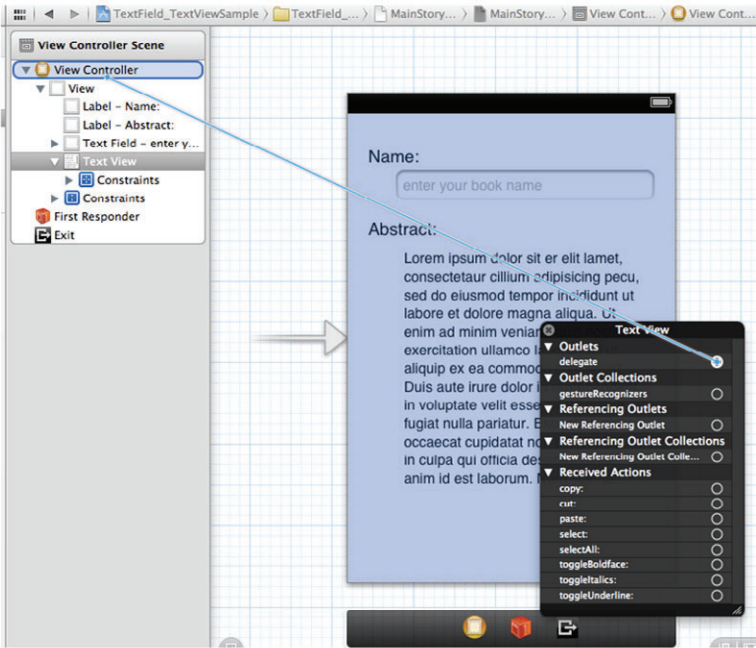


图4-21 在Interface Builder中分配委托

4.3.3 键盘的打开和关闭

一旦 UITextField 和 UITextView 等控件处于编辑状态，系统就会智能地弹出键盘，而不需要做任何额外的操作。但是，关闭键盘就不像打开键盘这样顺利了，我们需要用代码去实现。

首先我们要了解键盘不能自动关闭的原因。当 UITextField 或 UITextView 处于编辑状态时，这些控件变成了“第一

响应者”。要关闭键盘，就要放弃“第一响应者”的身份。在iOS中，事件沿着响应者链从一个响应者传到下一个响应者，如果其中一个响应者没有对事件做出响应，那么该事件会重新向下传递。

顾名思义，“第一响应者”是响应者链中的第一个，不同的控件成为“第一响应者”之后的“表现”不太一致。TextField和TextView等输入类型的控件会出现键盘，而我们只有让这些控件放弃它们的“第一响应者”身份，键盘才会关闭。

要想放弃“第一响应者”身份，需要调用UIResponder类中的resignFirstResponder方法，此方法一般在点击键盘的return键或者是背景视图时触发。本例采用点击return键关闭键盘的方式要实现这个操作，可以利用TextField和TextView的委托协议实现。相关的实现代码是在ViewController.m文件中完成的，具体如下所示：

```
@implementation ViewController

//通过委托来放弃“第一响应者”
#pragma mark - UITextField Delegate Method
- (BOOL)textFieldShouldReturn:(UITextField *)textField
{
    [textField resignFirstResponder];
    return YES;
}

//通过委托来放弃“第一响应者”
#pragma mark - UITextView Delegate Method
- (BOOL)textView:(UITextView *)textView shouldChangeTextInRange:
(NSRange)range replacementText:(NSString *)text
{
    if([text isEqualToString:@"\n"]) {
        [textView resignFirstResponder];
        return NO;
    }
    return YES;
}

@end
```

其中textFieldShouldReturn:方法是UITextFieldDelegate委托协议中定义的方法，在用户点击键盘时调用，其中的[textField resignFirstResponder]这条语句用于关闭键盘。与此类似，textView:shouldChangeTextInRange:replacementText:是由UITextViewDelegate委托协议提供的方法，它也是在用户点击键盘时被调用。

另外，如果界面中有很多控件，或者控件的位置比较靠近屏幕下方，控件就很可能被弹出的键盘挡住，此时可以添加UIScrollView控件来解决，详情可参见4.6节。

4.3.4 关闭和打开键盘的通知

在关闭和打开键盘时，iOS系统分别会发出如下广播通知：UIKeyboardDidHideNotification和UIKeyboardDidShowNotification。

使用广播通知的时候需要注意在合适的时机注册和解除通知，而ViewController.m中的有关代码如下：

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

    //注册键盘出现通知
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(keyboardDidShow:)
        name:UIKeyboardDidShowNotification object:nil];
    //注册键盘隐藏通知
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(keyboardDidHide:)
        name:UIKeyboardDidHideNotification object:nil];
    [super viewWillAppear:animated];
}
```



```
- (void) viewWillDisappear:(BOOL)animated {
    //解除键盘出现通知
    [[NSNotificationCenter defaultCenter] removeObserver:self
        name: UIKeyboardDidShowNotification object:nil];
    //解除键盘隐藏通知
    [[NSNotificationCenter defaultCenter] removeObserver:self
        name: UIKeyboardDidHideNotification object:nil];

    [super viewWillDisappear:animated];
}

- (void) keyboardDidShow: (NSNotification *)notif {
    NSLog(@"键盘打开");
}

- (void) keyboardDidHide: (NSNotification *)notif {
    NSLog(@"键盘关闭");
}
}
```

注册通知在viewWillAppear:方法中进行，解除通知在viewWillDisappear:方法中进行。
keyboardDidShow:消息是在键盘打开时发出的，keyboardDidHide:消息是在键盘关闭时发出的。

4.3.5 键盘的种类

我们之前所看到的键盘都是系统默认的类型。在iOS中，打开有输入动作的控件的属性检测器，可以发现Keyboard的下拉选项有10种类型键盘，如图4-22所示，我们可以根据需要进行选择。

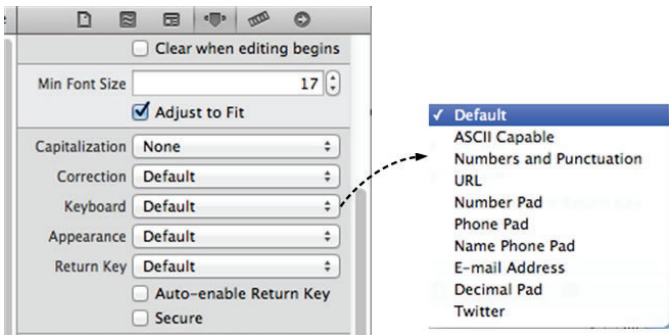


图4-22 选择键盘类型

选择不同的键盘类型，会在iOS上弹出不同的键盘，这些键盘的样式如图4-23 ~ 图4-26所示。



图4-23 ASCII键盘

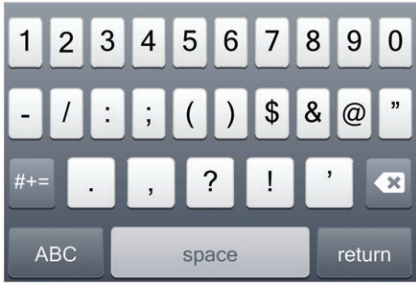


图4-24 数字和标点符号键盘



图4-25 邮箱键盘

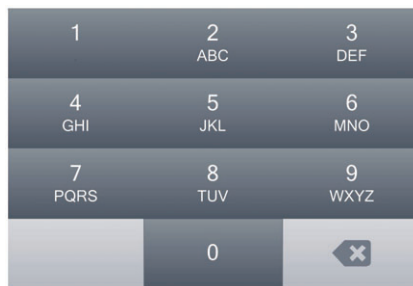


图4-26 电话拨号键盘

除了可以为控件选择合适的键盘类型外，我们还可以自定义return键的文本，而文本的内容根据有输入动作的控件而定。如果控件内输入的是查询条件，我们可以将return键的文本设置为Go或者Search，示意接下来进行的就是查找动作。return键的文本设置如图4-27所示。

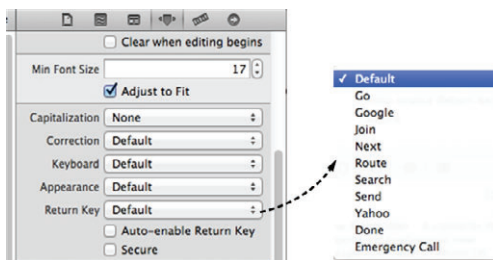


图4-27 选择return键的类型

4.4 开关控件、滑块控件和分段控件

开关控件、滑块控件和分段控件都是UIControl的子类，下面我们将通过一个示例为大家讲解这3个控件。

如图4-28所示，该案例包括两个开关控件、一个分段控件、两个标签控件和一个滑块控件。两个开关控件的值保持一致，点击其中一个，令其值为ON，另一个也会随之改变；一个有两段的分段控件，左侧和右侧的段分别命名为Left和Right，点击Right时两个开关控件消失，点击Left时两个开关显示；后面的滑块控件可以改变标签SliderValue的内容，把滑块变化的数值显示在后面。

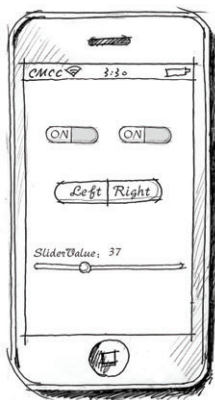


图4-28 案例原型设计图

4.4.1 开关控件

使用 Single View Application 模板创建一个名为 SwitchSliderSegmentedControlSample 的工程。打开 MainStoryboard.storyboard 文件，从对象库中拖曳两个开关控件到界面，然后为两个开关控件指定输出口，分别命名为 LeftSwitch 和 RightSwitch。在 ViewController.h 文件中声明一个 valueChanged: 方法，该方法的作用是同时设置两个开关的值，使它们的值保持一致，其实现代码如下：

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

开关控件的功能类似于 Windows 中的复选框，它只有两种状态：TRUE（或 YES）和 FALSE（或 NO），两种状态的切换方法是 setOn:animated:。上面代码中的 self.leftSwitch setOn:setting animated:YES 就是状态切换方法。

4.4.2 滑块控件

在视图上拖曳一个滑块控件，然后将其水平放置。打开它的属性检查器，将其最小值、最大值、初始值依次设定为 0.0、1.0、0.5。在 iOS 开发中，滑块的值是 0.0f~1.0f 之间的浮点数，值的设定方法如下：

```
-(void)setValue:(float)value animated:(BOOL)animated
```

在滑块上方拖曳两个标签，将左侧标签的文本改为 SliderValue:，将右侧标签的文本清除，并为其实现输出口，命名为 SliderValue。右侧的标签用于显示滑块的值，也就是滑块控制着标签的值，这里我们为滑块实现一个动作，将其命名为 sliderValueChange:。这些内容是在 ViewController.h 头文件中是这样定义的：

```
@property (weak, nonatomic) IBOutlet UILabel *SliderValue;

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

在 ViewController.m 文件中实现 sliderValueChange: 的代码如下：

```
- (IBAction) sliderValueChange:(id)sender {
    UISlider *slider = (UISlider *)sender;
    int progressAsInt = (int)(slider.value + 0.5f);
    NSString *newText = [[NSString alloc] initWithFormat:@"%d", progressAsInt];
    self.SliderValue.text = newText;
}
```

4.4.3 分段控件

分段控件也是一种选择控件，其功能类似于 Windows 中的单选按钮。它由两段或更多段构成，每个段相当于一个独立的按钮。它有两种样式——Plain&Bordered 样式和 Bar 样式，见图 4-29 和图 4-30，可以通过属性检查器中的 Style 属性设置。



图4-29 Plain&Bordered样式



图4-30 Bar样式

Bordered 样式是在 Plain 样式上加上一个边框，所以它们被归为一类。Plain&Bordered 样式中的每一段都可以设置文本、添加图片。Bar 样式整体看来比较窄，段中一般不放置图片。

我们在开关控件下方拖曳一个分段控件，双击使其处于编辑状态，依次输入文本Left和Right。然后回到代码中，在ViewController.h头文件中定义如下方法：

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

在ViewController.m文件中，实现touchDown:的代码如下：

```
- (IBAction)touchDown:(id)sender
{
    if (_leftSwitch.hidden == YES) {
        self.rightSwitch.hidden = NO;
        self.leftSwitch.hidden = NO;
    }else
    {
        self.leftSwitch.hidden = YES;
        self.rightSwitch.hidden = YES;
    }
}
```

4.5 网页控件 WebView

UIKit中的UIWebView类能够为用户提供显示多行文本的视图，能够冠以“Web”就说明它可以使用Web等技术进行显示HTML、解析CSS和执行JavaScript等操作。事实上，UIWebView的内核是开源的WebKit浏览器引擎。

从桌面到移动应用开发，一直有两大阵营的争论，关于是否本地好还是Web好的话题本书不做过多讨论，只是给出技术解决方案。事实上，除了本地和Web应用，我们还有第三条路可以走——Hybrid。Hybrid是本地+Web的混合产物，而WebView控件是Hybrid应用的关键技术，它不仅是负责解析HTML的控件，更是本地和Web进行沟通的桥梁。

4.5.1 WebView介绍

WebView控件可以加载本地HTML代码或者网络资源。

本地资源的加载采用同步方式，数据可以来源于本地文件或者是硬编码的HTML字符串，具体方法如下。

❑ **loadHTMLString:baseUrl**。设定主页文件的基本路径，通过一个HTML字符串加载主页数据。

❑ **loadData:MIMETYPE:textEncodingName:baseUrl**。指定MIME类型、编码集和NSData对象加载一个主页数据，并设定主页文件基本路径。

使用这两个方法时，需要注意字符集问题，而采用什么样的字符集取决于HTML文件。

加载网络资源时，我们采用的是异步加载方式，使用的方法是loadRequest:(NSURLRequest *)request，该方法要求提供一个NSURLRequest对象，该对象在构建的时候必须严格遵守某种协议格式，例如：

❑ **http://www.sina.com.cn**，HTTP协议；

❑ **file://localhost/Users/tonyguan/.../index.html**，文件传输协议；

其中http://和file://是协议名，不能省略。上网的时候我们常常将http://省略，一般的浏览器仍然可以解析你输入的URL，但是在WebView的loadRequest:方法中，该字符一定不能省略！

由于我们采用异步请求加载WebView，所以还要实现相应的UIWebViewDelegate委托协议，通过实现UIWebViewDelegate协议响应WebView在加载的不同阶段的事件。

下面我们通过一个案例（如图4-31所示）来了解一下WebView这3个方法的用法。该案例有3个按钮，分别为loadHTMLString、loadData和loadRequest，点击这3个按钮会分别触发WebView的3个加载方法。

使用Single View Application模板创建一个名为WebViewSample的工程，然后打开Interface Builder设计界面，按图4-32摆放控件。

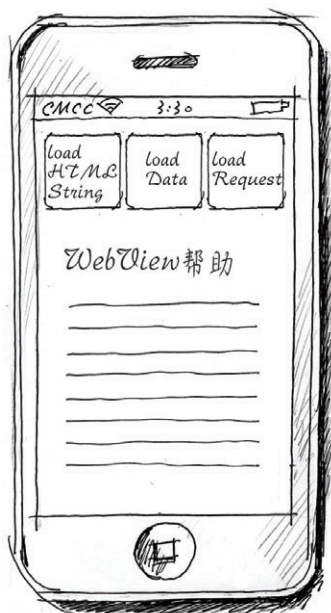


图4-31 案例原型设计图

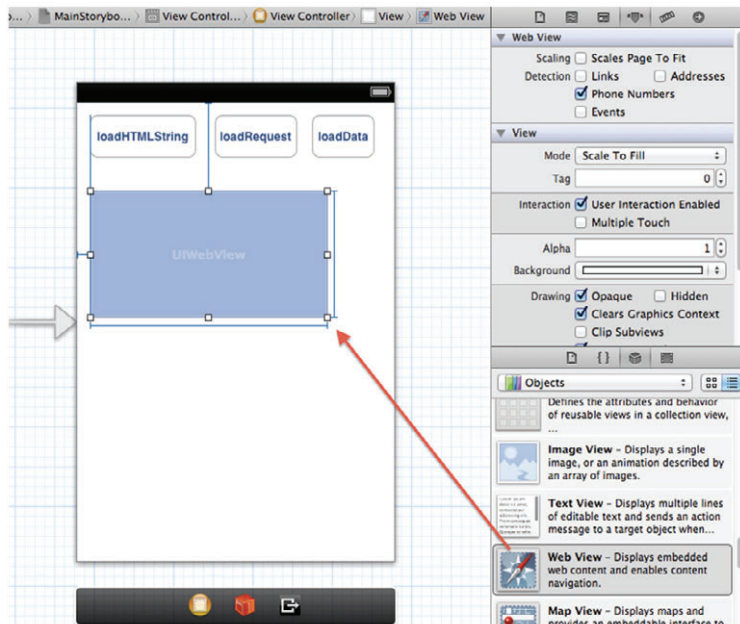


图4-32 Interface Builder设计界面

在ViewController.h文件中定义输出口和动作，具体代码如下：

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

@interface ViewController : UIViewController <UIWebViewDelegate>

@property (weak, nonatomic) IBOutlet UIWebView *webView;

- (IBAction)testLoadHTMLString:(id)sender;
- (IBAction)testLoadData:(id)sender;
- (IBAction)testLoadRequest:(id)sender;

@end
```

在上述代码中，我们定义了3个动作方法和一个UIWebView输出口属性。在ViewController.m文件中，testLoadHTMLString:和testLoadData:方法的代码如下：

```
- (IBAction)testLoadHTMLString:(id)sender {
    NSString *htmlPath = [[NSBundle mainBundle] pathForResource:
        @"index" ofType:@"html"];
    NSURL *bundleUrl = [NSURL fileURLWithPath:[NSBundle mainBundle] bundlePath]];
    NSError *error = nil;

    NSString *html = [[NSString alloc] initWithContentsOfFile:
        htmlPath encoding: NSUTF8StringEncoding error:&error];

    if (error == nil) { //数据加载没有错误的情况下
        [self.webView loadHTMLString:html baseURL:bundleUrl];
    }
}

- (IBAction)testLoadData:(id)sender {
    NSString *htmlPath = [[NSBundle mainBundle] pathForResource:
```



```

        @"index" ofType:@"html"];
    NSURL *bundleUrl = [NSURL fileURLWithPath:[NSBundle mainBundle] bundlePath]];
    NSError *error = nil;

    NSData *htmlData = [[NSData alloc] initWithContentsOfFile: htmlPath];

    if (error == nil) { //数据加载没有错误的情况下
        [self.webView loadData:htmlData MIMEType:@"text/html"
        textEncodingName:@"UTF-8"baseURL: bundleUrl];
    }
}

```

这两个方法用于加载本地资源文件index.html，并将其显示在WebView上。在testLoadHTMLString:方法中，通过NSString的initWithContentsOfFile:encoding:error:方法将index.html文件的内容读取到NSString对象中。在读取过程中，需要使用encoding参数将字符集指定为NSUTF8StringEncoding，其中NSUTF8StringEncoding是枚举类型NSStringEncoding的一个常量，error参数用于判断读取是否成功，如果error == nil则说明读取成功，否则失败。在self.webView loadHTMLString:html baseURL:bundleUrl语句中，baseURL参数用于设定主页文件的基本路径，即index.htm所在的资源目录，这可以用NSURL fileURLWithPath:[NSBundle mainBundle] bundlePath]语句来获取。

在方法testLoadData:中，我们使用NSData而没有采用NSString来读取index.html文件，这是因为NSData是一种二进制的字节数组类型。它没有字符集的概念，但用它来装载WebView的时候必须指定字符集，对应的方法是loadData:MIMEType:textEncodingName:baseURL:。

触摸loadRequest按钮时，WebView会发起异步调用，此时就会用到UIWebViewDelegate委托协议，相关代码如下：

```

- (IBAction)testLoadRequest:(id)sender {
    NSURL * url = [NSURL URLWithString: @"http://www.51work6.com"];
    NSURLRequest * request = [NSURLRequest requestWithURL:url];
    [self.webView loadRequest:request];
    self.webView.delegate = self;
}

#pragma mark -- UIWebViewDelegate委托定义方法
- (void)webViewDidFinishLoad: (UIWebView *) webView {
    NSLog(@"%@", [webView stringByEvaluatingJavaScriptFromString:
    @"document.body.innerHTML"]);
}

```

在testLoadRequest:方法中，我们先创建了一个NSURL对象，指定要请求的网址，其中网址必须是严格的HTTP格式，然后再构建NSURLRequest对象。获得NSURLRequest对象以后，就可以通过WebView的loadRequest:方法发起异步请求。异步调用不会导致主线程堵塞，并且会获得较好的用户体验。self.webView.delegate = self是必不可少的，该语句把WebView控件的委托对象（当前的视图控制器）分配给WebView。

UIWebViewDelegate委托协议定义的方法如下所示。

- ❑ **webView:shouldStartLoadWithRequest:navigationType:**。该方法在WebView开始加载新的界面之前调用，可以用来捕获WebView中的JavaScript事件。
- ❑ **webViewDidStartLoad:**。该方法在WebView开始加载新的界面之后调用。
- ❑ **webViewDidFinishLoad:**。该方法在WebView完成加载新的界面之后调用。
- ❑ **webView:didFailLoadWithError:**。该方法在WebView加载失败时调用。

本案例只使用了webViewDidFinishLoad:方法，其中使用WebView的stringByEvaluatingJavaScriptFrom:方法调用JavaScript的语句，使用document.body.innerHTML获得页面中HTML代码的JavaScript语句，日志输出结果如图4-33所示。



图4-33 案例日志输出结果

4.5.2 使用WebView构建Hybrid应用

Hybrid应用同时融合了本地技术和Web技术，它能够同时发挥本地技术和Web技术各自的优势。纯Web技术的应用虽然禁止在App Store中发布，但Hybrid却可以获得发布许可。有一个Hybrid框架——PhoneGap (<http://phonegap.com/>) 用于移动平台的开发。

提示 PhoneGap现在被Adobe收购了，它为开发者提供了13个调用本地的API，采用的是JavaScript、CSS3和HTML5等Web技术。PhoneGap的理想情况是不需要本地技术，而只用Web技术就可以完成本地移动应用的开发。

这里我们简要介绍一下PhoneGap的基本核心原理，包括内容：

- ❑ 本地代码调用JavaScript
- ❑ JavaScript调用本地代码

WebView是实现Hybrid应用的核心，也是实现本地技术和Web技术融合的核心。所有移动平台都有类似WebView的控件：在iOS平台中是UIWebView，Android平台中是WebView，Windows Phone平台中是WebBrowser。

下面我们通过一个案例来了解如何通过本地代码调用JavaScript代码以及如何通过JavaScript来调用本地代码。如图4-34所示，该界面是WebView。在案例启动时候，WebView页面加载的时候，本地代码会调用JavaScript函数，并在WebView页面中输出“从iOS对象中调用JS Ok.”字符串。在WebView页面可以点击“调用iOS对象”按钮（这是一个HTML按钮），来调用iOS本地代码在日志输出一些内容。

使用Single View Application模板，创建一个名为MyPhoneGap的工程。打开Interface Builder设计界面，拖曳一个WebView控件并使其覆盖整个界面。

1. 使用本地代码调用JavaScript

为了通过本地代码调用JavaScript，我们需要编写一个HTML文件index.html，其代码如下：

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport"
          content="width=device-width, height=device-height, initial-scale=1.0,
                  maximum-scale=1.0, user-scalable=no;" />
```

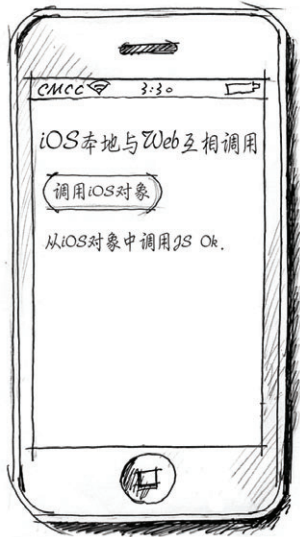


图4-34 案例设计图


```

<meta http-equiv="Content-type" content="text/html; charset=utf-8"/>
<script>
    //从iOS对象中调用
    function helloWorld(msg) {
        document.getElementById('message').innerHTML = msg;
    }
</script>

</head>
<body>
    <h2>iOS本地与Web互相调用</h2>
    <button>调用iOS对象</button>
    <br><br>
    <div id='message'></div>
</body>
</html>

```

在移动设备上开发Web应用时，需要指定页面的viewport属性。通过viewport属性，可以使页面适合于任何一种屏幕尺寸的移动设备。helloWorld(msg)函数会在iOS本地代码（即视图控制器）中调用，该函数的作用是调用document.getElementById('message').innerHTML = msg语句，在一个id为message的div标签上显示字符串，这个字符串由视图控制器传递过来。

编写完index.html文件后，需要把它放到MyPhoneGap工程中，具体操作过程是这样的：首先到MyPhoneGap工程所在的Finder目录下创建一个www文件夹，将index.html文件复制到www文件夹下面，如图4-35所示。

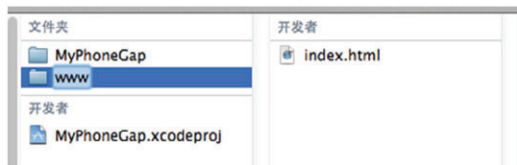


图4-35 创建www文件夹

回到工程，右击工程名MyPhoneGap，在弹出的快捷菜单中选择Add File to “MyPhoneGap”，此时会弹出如图4-36所示的对话框。

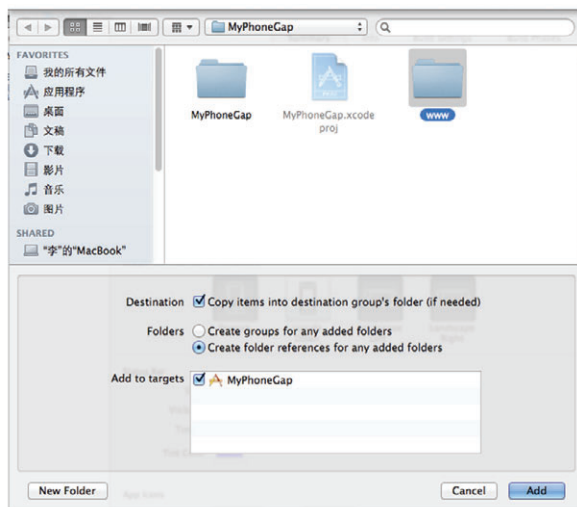


图4-36 “添加文件”对话框

选择www文件夹，在Folders项中选择Create folder references for any added folders单选按钮，然后点击Add按钮，这里选择这个单选按钮的原因是为了在工程中创建一个www文件夹，而不是www组。在Xcode工程中，文件夹和组的颜色是不同的，文件夹是灰色的，组是黄色的，如图4-37所示。

在使用过程中，文件夹和组也有着本质的差别。如果将文件放入到文件夹中，则访问文件的路径是“文件夹/index.html”，而如果将文件放入到组中，访问路径则为“index.html”。

下面我们来看看使用本地代码调用JavaScript的HelloWorldViewController的代码，具体如下：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.webView.delegate = self;
    NSString *path = [[NSBundle mainBundle] pathForResource:
        @"www/index" ofType:@"html"];
    [self.webView loadRequest:[NSURLRequest requestWithURL:
        [NSURL fileURLWithPath: path]]];
}

#pragma mark -
#pragma mark UIWebViewDelegate
- (void)webViewDidFinishLoad:(UIWebView *)webView
{
    [self.webView stringByEvaluatingJavaScriptFromString:
        @"helloWorld('从iOS对象中调用JS Ok.')"];
}
```

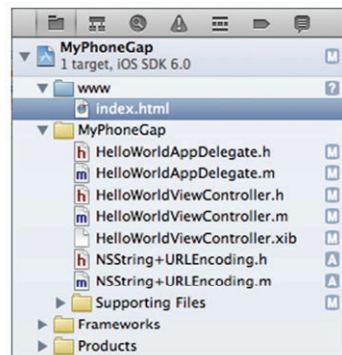


图4-37 Xcode中的文件夹和组

在viewDidLoad方法中，我们加载了index.html文件到WebView控件。index.html虽然是本地文件，但是这里使用的方法是loadRequest:方法，该方法可以进行异步加载，在页面加载的不同阶段会发出不同的消息给UIWebViewDelegate委托对象（HelloWorldViewController），我们就在这些消息中编写代码。由于我们的目的是在Web加载新界面完成之后调用JavaScript，所以我们要使用webViewDidFinishLoad:方法。下面是调用helloWorld函数的语句：

```
[self.webView stringByEvaluatingJavaScriptFromString:@"helloWorld('从iOS对象中调用JS Ok.')'];
```

这样，当WebView加载完成后，界面上就会输出“从iOS对象中调用JS Ok.”字符串。

2. 使用JavaScript调用本地代码

下面我们再看看通过JavaScript调用本地代码的过程。修改index.html文件，添加JavaScript函数showAndroidDialog，并在button按钮中实现showAndroidDialog的调用，相关代码如下：

```
<!DOCTYPE html>
<html>
  <head>
    .....
    <script>
      //从iOS对象中调用
      function helloWorld(msg) {
        document.getElementById('message').innerHTML = msg;
      }
      //调用iOS对象
      function showAndroidDialog(msg) {
        var myJSONObject = new Object();
        myJSONObject.title = 'HelloWorld';
        myJSONObject.message = msg;
        var jsonString = JSON.stringify(myJSONObject);
        var uri = 'gap://XXXClass.XXXmethod#' + jsonString;
        window.location = uri;
      }
    </script>
  </head>
  <body>
    <div id="message"></div>
    <button id="showAndroidDialog">Show Android Dialog</button>
  </body>
</html>
```

```

    }
</script>

</head>
<body>
    <h2>iOS本地与Web互相调用</h2>
    <button onclick='showAndroidDialog("JS to iOS 对象")'>调用iOS对象</button>
    .....
</body>
</html>

```

从JavaScript调用本地代码的关键是要在JavaScript函数中调用`window.location = uri`语句,然后在iOS的HelloWorldViewController中就会收到`webView:shouldStartLoadWithRequest:navigationType:`消息,这是因为WebView在开始加载新的界面之前会调用该方法。JavaScript中的`window.location`语句就是界面跳转,所以会触发`webView:shouldStartLoadWithRequest:navigationType:`消息。在`showAndroidDialog`函数中,为了把一个JSON对象传递给HelloWorldViewController,需要序列化JSON对象:

```

var myJSONObject = new Object();
myJSONObject.title = 'HelloWorld';
myJSONObject.message = msg;
var jsonString = JSON.stringify(myJSONObject);

```

而函数中的uri是'`gap://XXXClass.XXXmethod#`' + jsonString,这是我们自己定义的协议,目的是与iOS中的HelloWorldViewController对象沟通。本例中的uri如下:

```
gap://XXXClass.XXXmethod#{"title":"HelloWorld","message":"JS to iOS 对象"}
```

提示 URI (Uniform Resource Identifier, 通用资源标识符), 具备全球唯一性, 一般用于定位Web资源。我们在Web上经常会遇到下面形式的URI: `http://www.acme.com/icons/logo.gif`。它的格式: `<scheme> : <host> [? <query>] [# <fragment>]`

其中scheme是gap, 这个名字是我们自定义的, 如果在HelloWorldViewController中判断scheme是gap, 则说明这是由点击Web界面中的按钮调用的; host是XXXClass.XXXmethod, 这部分是我们自定义的, 用于表示在HelloWorldViewController中接下来的处理是调用哪个类的哪个方法, 在本例中并没有使用它; query部分没有使用; fragment为`{"title":"HelloWorld","message":"JS to iOS 对象"}`, 这是序列化的JSON对象。

下面我们来看看使用本地代码HelloWorldViewController调用JavaScript的有关代码:

```

- (BOOL)webView:(UIWebView *)webView shouldStartLoadWithRequest:(NSURLRequest *)request
    navigationType:(UIWebViewNavigationType)navigationType
{
    NSString *actionType = request.URL.host;                                ①
    NSString *scheme = request.URL.scheme;                                ②
    NSString *fragment = [request.URL.fragment URLDecodedString];          ③
    NSData *responseData = [fragment dataUsingEncoding:NSUTF8StringEncoding]; ④

    if ( [scheme isEqualToString:@"gap"] ) {                               ⑤
        if ( [actionType isEqualToString:@"XXXClass.XXXmethod"] ) {        ⑥

            NSError* error;
            NSDictionary* json = [NSJSONSerialization
                                  JSONObjectWithData:responseData
                                  options:NSJSONReadingAllowFragments
                                  error:&error];

            NSLog(@"title: %@ , message: %@", [json objectForKey:@"title"],
                [json objectForKey:@"message"] );

```

```

    }
    return true;
}

```

使用NSURLRequest类的参数request的URL属性,可以获得NSURL对象。在NSURL中,共有3个属性,它们分别是scheme、host和fragment,获取方式如下所示。

- ❑ 第①行request.URL.scheme,可以获得scheme属性,本例中的值是gap。
- ❑ 第②行request.URL.host,可以获得host属性,本例中的值是XXXClass.XXXmethod。
- ❑ 第③行request.URL.fragment,可以获得host的属性fragment。fragment的内容是URL编码的字符串,它的作用是将字符转化为可在因特网上安全传输的格式,例如“%22”是双引号字符。

那么,在iOS中如何进行字符串的URL编码和解码呢?在本例中,[request.URL.fragment URLDecodedString]就采用了NSString分类方法URLDecodedString对URL字符串进行了解码。我们编写的NSString+URLEncoding分类的代码如下:

```

#import "NSString+URLEncoding.h"
@implementation NSString (URLEncoding)
- (NSString *)URLEncodedString
{
    NSString *result = (NSString *)
    CFURLCreateStringByAddingPercentEscapes(kCFAllocatorDefault,
        (CFStringRef)self,
        NULL,
        CFSTR("!*'();:@&+=,/?%#[] "),
        kCFStringEncodingUTF8);
    return result;
}
- (NSString*)URLDecodedString
{
    NSString *result = (NSString *)
    CFURLCreateStringByReplacingPercentEscapesUsingEncoding(kCFAllocatorDefault,
        (CFStringRef)self,
        CFSTR(""),
        kCFStringEncodingUTF8);
    return result;
}
@end

```

在上述代码中,共涉及两个方法,其中URLDecodedString是URL字符串解码方法,URLEncodedString是URL字符串编码方法。在URLEncodedString方法中,我们使用一个C语言函数CFURLCreateStringByAddingPercentEscapes编码URL字符串,该函数的第一个参数是分配内存的方式,kCFAllocatorDefault表示采用默认的模式,第二个参数是原始字符串,第三个参数是保留的不进行转义的字符,NULL表示全部的合法字符串都转义,第四个参数是要转义的非合法字符,NULL表示所有的非法字符串都被替换,第五个参数是字符集。在URLDecodedString方法中,我们使用一个C语言函数CFURLCreateStringByReplacingPercentEscapesUsingEncoding解码URL字符串,该函数的第一个参数是分配内存的方式,kCFAllocatorDefault表示采用默认模式,第二个参数是原始字符串,第三个参数是保留的不进行%字符的转换,NULL表示全部的%字符串都不转换,第四个参数是字符集。

在HelloWorldViewController调用JavaScript时,使用NSJSONSerialization类来解析JSON字符串,相关代码如下:

```

NSDictionary* json = [NSJSONSerialization
    JSONObjectWithData:responseData
    options:NSJSONReadingAllowFragments
    error:&error];

```

NSJSONSerialization是iOS 5之后提供的新API，通过它可以解析JSON字符串或序列化JSON对象。本例中，我们采用JSONObjectWithData:options:error:方法解析JSON字符串，其中options参数指定了解析JSON的模式，它是枚举类型NSJSONReadingOptions中定义了的3个常量之一。

- ❑ **NSJSONReadingMutableContainers**。指定解析返回的是可变的数组或字典。如果以后需要修改结果，这个常量是合适的选择。
- ❑ **NSJSONReadingMutableLeaves**。指定解析返回的是可变字符串。
- ❑ **NSJSONReadingAllowFragments**。指定顶级节点不能是数组或字典对象。

4.6 屏幕滚动控件 ScrollView

ScrollView在UIKit中是UIScrollView类，是容器类型的视图。它有两个子类——UITextView和UITableView，它们在内容超出屏幕时提供水平或垂直滚动条。

4.6.1 ScrollView属性的设置

ScrollView的属性有很多，但最为重要的就是与显示相关的属性：contentSize、contentInset和contentOffset。

1. contentSize属性

contentSize属性表示ScrollView中内容视图（Content View）的大小，它返回CGSize结构体类型，该结构体包含width和height两个成员。如图4-38所示，内容视图是图中灰色部分（320×544），而ScrollView视图大小（frame指定的范围）只有320×460。正是因为内容视图超出了ScrollView视图大小，才会有滚动屏幕的必要。

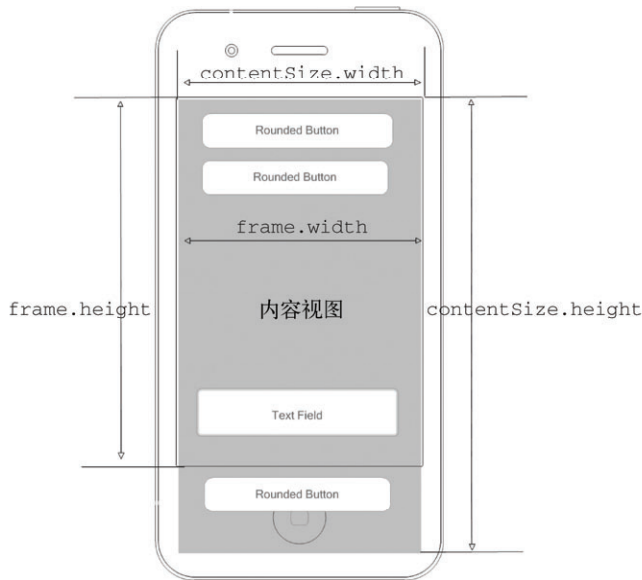


图4-38 contentSize属性

对于图4-38，相关说明如下所示。

- ❑ **contentSize.width**。ScrollView内容视图的宽。
- ❑ **contentSize.height**。ScrollView内容视图的高。
- ❑ **frame.width**。ScrollView的宽。


❑ `frame.height`。ScrollView的高。

2. `contentInset`属性

`contentInset`属性用于在ScrollView中的内容视图周围添加边框，这往往为了留出空白以放置工具栏、标签栏或导航栏等。图4-39所示的是`contentInset.top`属性。



图4-39 `contentInset`属性

`contentInset`属性有4个分量，分别是Top、Bottom、Left和Right，分别代表顶边距离、底边距离、左边距离和右边距离。在Interface Builder中选中ScrollView，打开其尺寸检查器，即可在Content Insets中设定`contentInset`属性的这4个分量，如图4-40所示。

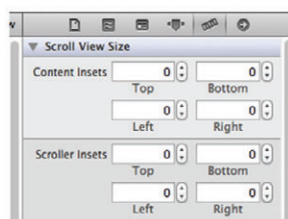


图4-40 尺寸检查器的`contentInset`属性

3. `contentOffset`属性

`contentOffset`属性是内容视图坐标原点与ScrollView坐标原点的偏移量，返回CGPoint结构体类型，这个结构体类型包含x和y两个成员。如图4-41所示，内容视图沿y轴负偏移（或者说ScrollView视图沿y轴正偏移），x轴方向没有偏移。

偏移量可以通过ScrollView方法或属性设定。设定ScrollView视图沿y轴正偏移110点的代码如下：

```
[self.scrollView setContentOffset:CGPointMake(0, 110) animated:YES];
```

或者

```
self.scrollView.contentOffset = CGPointMake(0, 110);
```

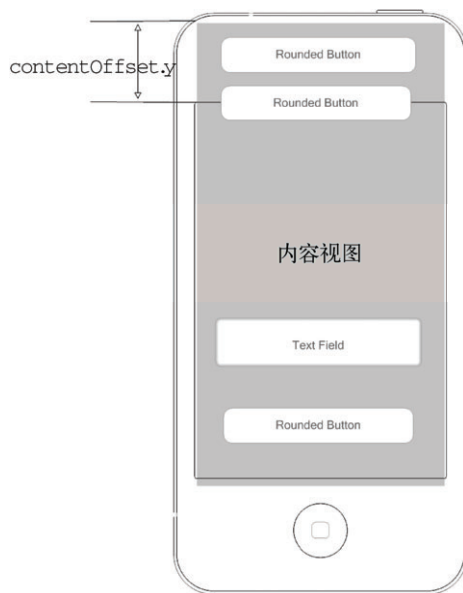


图4-41 内容视图沿y轴负偏移

如果使用`self.scrollView setContentOffset:CGPointMake(0, 110) animated:YES`方法设定，在偏移的同时可以出现动画效果。

下面我们实现图4-36所示的界面，界面中共有3个按钮、一个文本框和一个ScrollView。首先，使用Xcode创建工程，工程模板采用Single View Application，工程名为ScrollViewSample。打开Interface Builder设计界面，从对象库中拖曳ScrollView控件到设计界面，如图4-42所示。

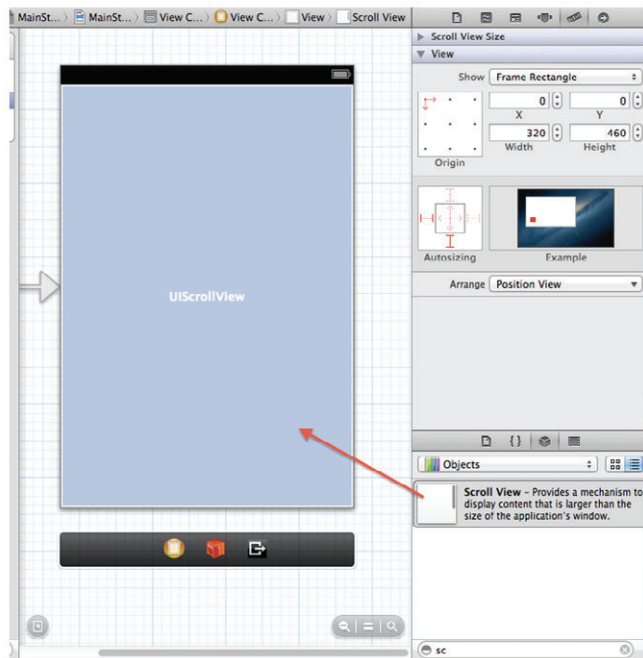



图4-42 在Interface Builder设计ScrollView

在设计界面中选择ScrollView，打开其尺寸检查器，设定其Width属性为320，Height属性为460，这刚好与它的父视图View的大小一样。然后再依次拖曳3个按钮和一个文本框到ScrollView中，如图4-43所示。这几个控件的摆放位置不用很精确，但是最后一个按钮需要超出屏幕之外，因此将它的y坐标设定为500。

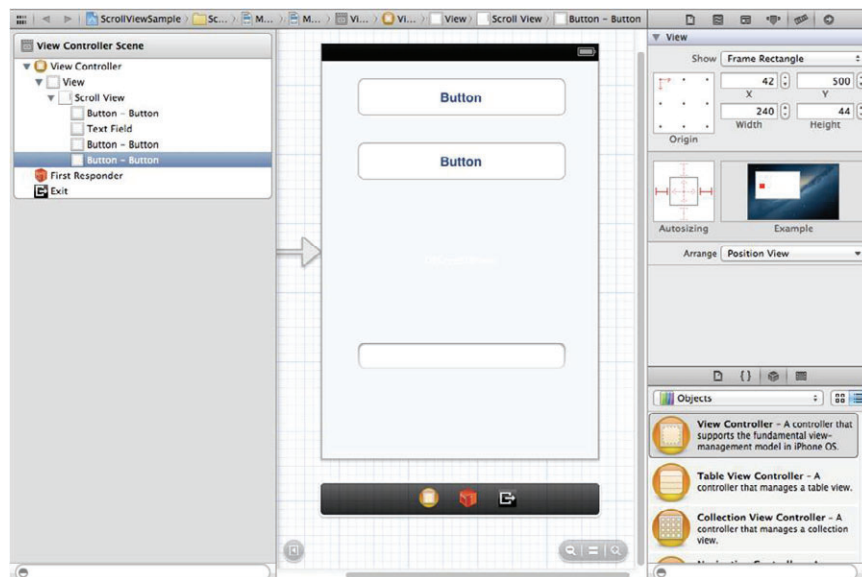


图4-43 在Interface Builder设计按钮和文本框

然后把ScrollView控件设定为输出口，细节就不再介绍了。下面我们看看ViewController.h文件的代码：

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

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UIScrollView *scrollView;

@end
```

scrollView属性是与输出口ScrollView控件对应的。下面我们看看ViewController.m文件的代码如下：

```
@implementation ViewController
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.scrollView.contentSize = CGSizeMake(320, 600);
}
- (void)viewDidAppear:(BOOL)animated
{
    [self.scrollView setContentOffset:CGPointMake(0, 110) animated:YES];
    //self.scrollView.contentOffset = CGPointMake(0, 110);
    [super viewDidAppear:YES];
}
@end
```

在viewDidLoad方法中，我们一般要设定ScrollView控件的contentSize属性，这个属性一般都是通过程序代码动态设定的，这是因为很多情况下它的尺寸是计算出来的。本例中设定为320×600，其中高度600能够把最后一个按钮包含在内容视图中。

4.6.2 键盘与其他控件的协同

在iOS开发中，键盘是很麻烦的。显示键盘时会遮挡一些控件，如图4-44所示。那么如何使界面中的众多控件与键盘很好地协同，给用户很好的体验呢？这里的关键是在键盘打开前后，要摆放好ScrollView和控件（例如，文本框）的位置。



图4-44 键盘遮挡住TextField控件

现在我们对上个例子ScrollViewSample进行修改，使ScrollView中的控件与上边框的距离为绝对距离。选择一个控件，打开其尺寸检查器，如图4-45所示，选择View的Autosizing属性中矩形顶部的“I”型虚线线段，将其改成实线线段（这说明该控件与父视图的上边距是绝对的）。

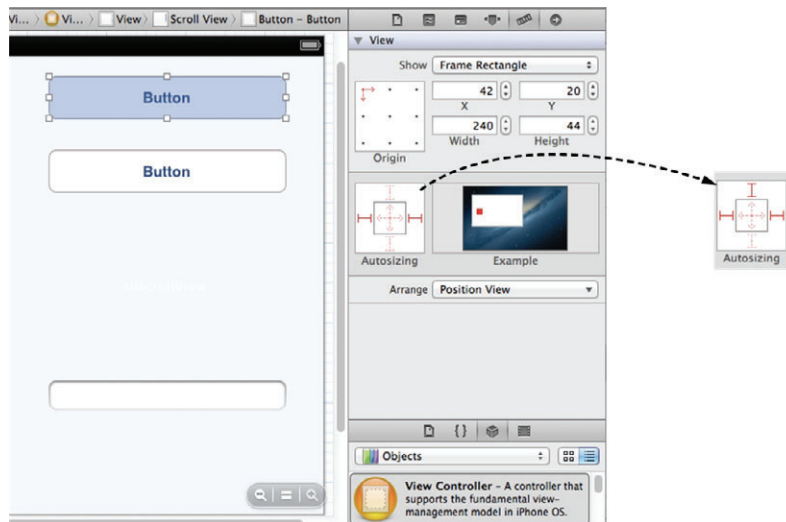



图4-45 修改View的Autosizing属性

提示 默认情况下，当我们使用Xcode 4.5创建故事板时，我们采用iOS 6的Autolayout布局技术。关于Autolayout布局，我们会在4.9节中介绍，但在4.9节之前我们还是不采用Autolayout布局技术。不采用Autolayout布局时，需要选择xib或故事板文件，打开文件检查器，取消选中Use Autolayout复选框。

依次将ScrollView中的所有控件都改成上边距绝对定位，这样它们的y坐标不会因为父视图高度的变化而变化。

我们再看一下ScrollViewSample案例的代码部分，其中ViewController.h文件代码如下：

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

@interface ViewController : UIViewController <UITextFieldDelegate>
{
    BOOL keyboardVisible;    // 键盘打开标识
}
@property (weak, nonatomic) IBOutlet UIScrollView *scrollView;
@property (weak, nonatomic) IBOutlet UITextField *textField;

@end
```

通过上述代码，我们可以发现除了为UIScrollView控件定义属性外，还定义了UITextField控件的属性，并且都设置为输出类型。

此外，ViewController还实现了UITextFieldDelegate，我们需要在Interface Builder中把ViewController分配给文本框的委托属性，具体操作方法如下：在Interface Builder中选择Text Field控件，打开连接检查器，将Outlets中delegate后面的小圆点拖曳到View Controller即可，如图4-46所示。

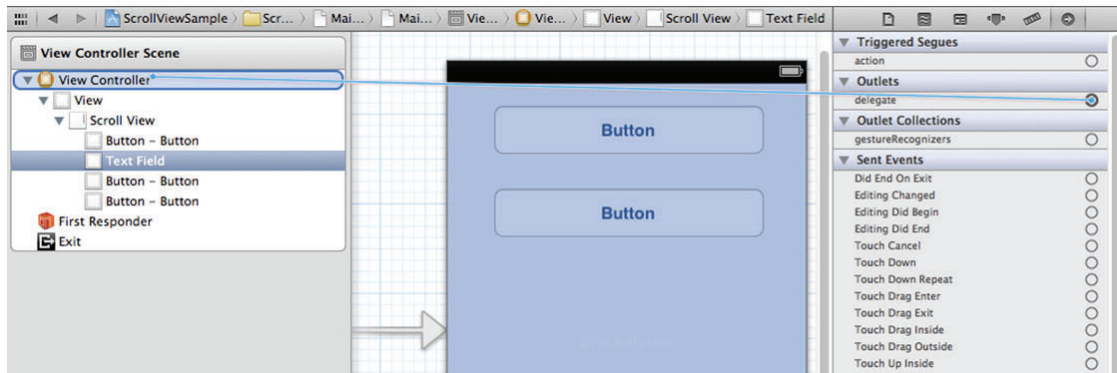


图4-46 把View Controller分配给文本框的委托属性

下面我们再看看ViewController.m文件中打开与关闭键盘的部分代码：

```
-(void) keyboardDidShow: (NSNotification *)notif {
    if (keyboardVisible) { // 如果键盘已经出现，要忽略通知
        return;
    }
    // 获得键盘尺寸
    NSDictionary* info = [notif userInfo];
    NSValue* aValue = [info objectForKey:UIKeyboardFrameEndUserInfoKey];
    CGSize keyboardSize = [aValue CGRectValue].size;

    // 重新定义ScrollView的尺寸
    CGRect viewFrame = self.scrollView.frame;
    viewFrame.size.height -= (keyboardSize.height);
    self.scrollView.frame = viewFrame;
}
```

```

//滚动到当前文本框
CGRect textFieldRect = [self.textField frame];
[self.scrollView scrollRectToVisible:textFieldRect animated:YES];

keyboardVisible = YES;
}

-(void) keyboardDidHide: (NSNotification *)notif {

    NSDictionary* info = [notif userInfo];
    NSValue* aValue = [info objectForKey:UIKeyboardFrameEndUserInfoKey];
    CGSize keyboardSize = [aValue CGRectValue].size;

    CGRect viewFrame = self.scrollView.frame;
    viewFrame.size.height += keyboardSize.height;
    self.scrollView.frame = viewFrame;

    if (!keyboardVisible) {
        return;
    }

    keyboardVisible = NO;
}

```

其中keyboardDidShow:消息是键盘打开时发出的，这需要注册键盘打开通知（UIKeyboardDidShowNotification）；keyboardDidHide:消息是键盘关闭时发出的，这需要注册键盘关闭通知（UIKeyboardDidHideNotification）。

在keyboardDidShow:方法中，我们主要从UIKeyboardDidShowNotification通知中获得键盘的尺寸，然后根据键盘的尺寸重新计算和设定ScrollView的高度，最后将屏幕滚动到当前的文本框。获得键盘尺寸的语句为代码第①~③行。

广播通知可以传递NSDictionary类型的参数，这个由系统发出的UIKeyboardDidShowNotification通知可以传递字典类型的数据，可以使用userInfo来获得该数据。使用UIKeyboardFrameEndUserInfoKey取出其中的键盘的Frame值，然后再用CGRectValue方法获得CGSize结构体类型的键盘尺寸数据。

这里我们修改ScrollView尺寸的原因是因为我们始终能够看到ScrollView区域，但是键盘打开后，这个ScrollView区域就被键盘遮挡了，所以我们需要重新计算减掉键盘后的尺寸。

将ScrollView滚动到当前控件，这可以通过scrollRectToVisible:animated:来实现，其中scrollRectToVisible参数用于指定滚动到一个矩形区域，这个矩形区域是CGRect结构体。每个视图的frame方法可以获得CGRect结构体数据。

键盘关闭的处理要比键盘打开简单一些，不需要滚动到当前文本框，只要重新设定ScrollView的高度加上键盘的高度就可以了。

4.7 等待相关的控件与进度条

在请求完成之前，经常会用到活动指示器ActivityIndicatorView和进度条ProgressView，其中活动指示器可以消除用户的心理等待时间，而进度条可以指示请求的进度。

下面我们通过一个案例讲解一下这两个控件，其原型图如图4-47所示，其中有两个按钮——Upload和Download，分别对应于活动指示器和进度条。点击Upload按钮时，活动指示器开始旋转，再次点击该按钮时停止旋转。点击Download按钮，进度条开始前进，完成时弹出一个对话框。

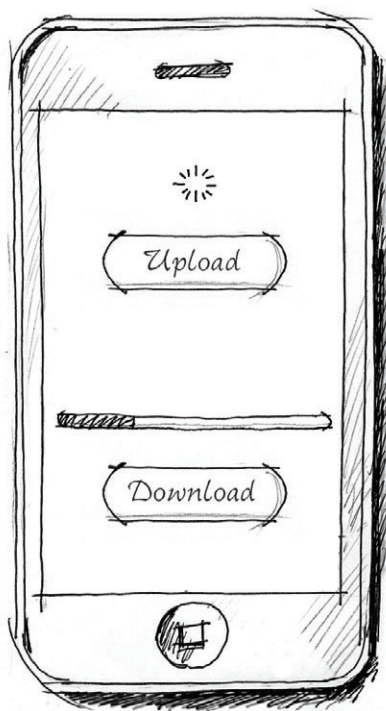


图4-47 原型图

4.7.1 活动指示器ActivityIndicatorView

使用Single View Application模板，创建一个名为ActivityIndicatorViewProgressViewSample的工程。

打开Interface Builder设计界面，在视图上拖曳一个ActivityIndicatorView控件和一个按钮，将按钮命名为Upload。为了与白色的活动指示器区分，我们将后面的视图背景设置为黑色。

打开Interface Builder，实现按钮的动作和活动指示器的输出口。ViewController.h文件中的相关代码如下：

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

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UIActivityIndicatorView *myActivityIndicatorView;

- (IBAction)startToMove:(id)sender;

@end
```

在ViewController.m文件中，点击Upload按钮的实现代码如下：

```
- (IBAction)startToMove:(id)sender
{
    if ([self.myActivityIndicatorView isAnimating]) {
        [self.myActivityIndicatorView stopAnimating];
    } else {
        [self.myActivityIndicatorView startAnimating];
    }
}
```

在上述代码中，isAnimating方法用于判断ActivityIndicatorView是否处于运动状态，stopAnimating方法用于停止旋转，startAnimating方法用于开始旋转。

4.7.2 进度条ProgressView

进度条体现了任务进行的进度，同活动指示器一样，也有消除用户心理等待时间的作用。

为了模拟真实的任务进度的变化，我们在案例中引入了定时器（NSTimer）。定时器继承于NSObject类，可以在特定的时间间隔后向某对象发出消息。

打开Interface Builder，实现按钮的动作和进度条的输出口，ViewController.h文件中的相关代码如下：

```
@interface ViewController : UIViewController
{
    NSTimer *myTimer;
}
@property(n nonatomic, strong) NSTimer *myTimer;
@property (weak, nonatomic) IBOutlet UIProgressView *myProgressView;
- (IBAction)downloadProgress:(id)sender;

@end
```

其中Download按钮的实现代码如下：

```
- (IBAction)downloadProgress:(id)sender
{
    myTimer = [NSTimer scheduledTimerWithTimeInterval:1.0
                                                target:self
                                                selector:@selector(download)
                                                userInfo:nil
                                                repeats:YES];

- (void)download{
    self.myProgressView.progress=self.myProgressView.progress+0.1;
    if (self.myProgressView.progress==1.0) {
        [myTimer invalidate];
        UIAlertView*alert=[[UIAlertView alloc] initWithTitle:@"download completed! "
                                                    message:@" "
                                                    delegate:nil
                                                    cancelButtonTitle:@"OK"
                                                    otherButtonTitles: nil];

        [alert show];
    }
}
```

可以看到，NSTimer的类方法是+ (NSTimer *)scheduledTimerWithTimeInterval:(NSTimeInterval)seconds target:(id)target selector:(SEL)aSelector userInfo:(id)userInfo repeats:(BOOL)repeats，其中seconds参数用于设定间隔时间，target用于指定发送消息给哪个对象，aSelector指定要调用的方法名，相当于一个函数指针，userInfo可以给消息发送参数，repeats表示是否重复。

download方法是定时器调用的方法，在定时器完成任务后一定要停止它，这可以通过语句[myTimer invalidate]来实现。

4.8 警告框和操作表

应用如何与用户交流呢？警告框（AlertView）和操作表（ActionSheet）就是为此而设计的。

本节案例的原型草图如图4-48所示，其中有两个按钮“Test警告框”和“Test操作表”，点击“Test警告框”按钮时弹出警告框，它有两个按钮。当点击“Test操作表”按钮时，屏幕下方将滑出操作表。

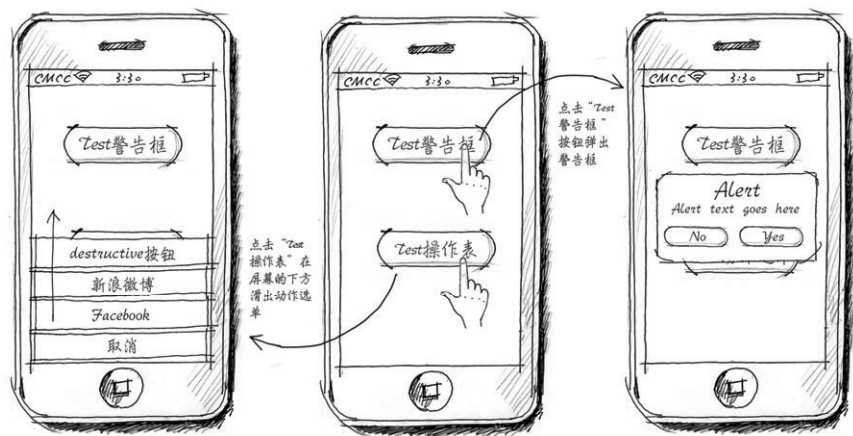


图4-48 案例原型草图

4.8.1 警告框UIAlertView

警告框是UIAlertView创建的，用于给用户以警告或提示，最多有两个按钮，超过两个就应该使用操作表。由于在iOS中，警告框是“模态”的^①，因此不应该随意使用。一般情况下，警告框的使用场景有如下几个。

- ❑ 应用不能继续运行。例如，无法获得网络数据或者功能不能完成的时候，给用户一个警告，这种警告框只需一个按钮。
- ❑ 询问另外的解决方案。好多应用在不能继续运行时，会给出另外的解决方案，让用户去选择。例如，Wi-Fi网络无法连接时，是否可以使用3G网络。
- ❑ 询问对操作的授权。当应用访问用户的一些隐私信息时，需要用户授权，例如用户当前的位置、通讯录或日程表等。

下面看看图4-48所示案例中警告框的实现过程。使用Single View Application模板，创建一个名为UIAlertViewActionSheetSample的工程。打开Interface Builder设计界面，按照图4-49所示摆放两个按钮控件。

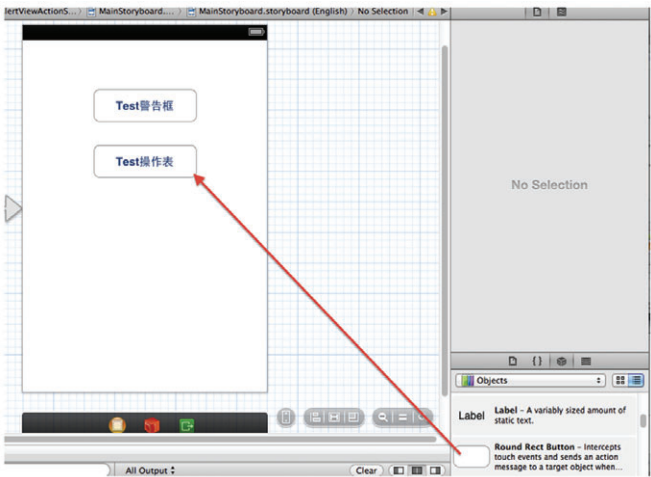


图4-49 设计界面

① “模态”表示的是不关闭它就不能做别的事情。

下面我们为这两个按钮定义动作事件，ViewController.h中的相关代码如下：

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

@interface ViewController : UIViewController <UIAlertViewDelegate>

- (IBAction)testAlertView:(id)sender;
- (IBAction)testActionSheet:(id)sender;

@end
```

可以看到，视图控制器实现了UIAlertViewDelegate协议，这个协议是UIAlertView委托协议。点击UIAlertView中的按钮，会给委托对象发送alertView:clickedButtonAtIndex:消息，ViewController.m中的相关代码如下：

```
- (IBAction)testAlertView:(id)sender {

    UIAlertView *alertView = [[UIAlertView alloc]
                               initWithTitle:@"Alert"
                               message:@"Alert text goes here"
                               delegate:self
                               cancelButtonTitle:@"No"
                               otherButtonTitles:@"Yes", nil];

    [alertView show];
}

#pragma mark-- 实现UIAlertViewDelegate
- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)buttonIndex {
    NSLog(@"buttonIndex = %i",buttonIndex);
}
```

在testAlertView:方法中实例化UIAlertView对象时，最常用的构造函数是initWithTitle:message:delegate:cancelButtonTitle:otherButtonTitles:，其中delegate参数在本例中设定为self，即该警告框的委托对象为当前的视图控制器（ViewController）；cancelButtonTitle参数用于设置“取消”按钮的标题，它是警告框的左按钮；otherButtonTitles参数是其他按钮，它是一个字符串数组，该字符串数组以nil结尾。从技术层面上讲，警告框可以多于两个按钮，这都是通过otherButtonTitles参数设定的，但是从用户体验上讲，警告框最多有两个按钮。如果警告框只有一个按钮，可以采用下面的语句构造警告框：

```
UIAlertView *alertView = [[UIAlertView alloc]
                           initWithTitle:@"Alert"
                           message:@"Alert text goes here"
                           delegate:nil
                           cancelButtonTitle:@"OK"
                           otherButtonTitles: nil];
```

此时警告框只是给用户一些警告信息，当用户点击OK按钮时，只是为了关闭警告框，因此不需要指定委托参数。但是有两个按钮的情况下，为了响应点击警告框按钮的需要，我们在视图控制器中实现了alertView:clickedButtonAtIndex:方法，其中clickedButtonAtIndex参数是按钮索引，cancelButton按钮的索引是0，从左到右依次是1，2…。

4.8.2 操作表ActionSheet

如果想给用户多于两个的选择，比如想把应用中的某个图片发给新浪微博或者Facebook等平台，就应该使用操作表。操作表是UIActionSheet创建的，在iPhone下运行会从屏幕下方滑出来，如图4-50所示，其布局是最下面是一个“取消”按钮，它离用户的大拇指最近，最容易点击到。如果选项中有一个破坏性的操作，将会放在最上面，是大拇指最不容易碰到的位置，并且其颜色是红色的。

在iPad中，操作表的布局与iPhone有所不同，如图4-51所示。在iPad中，操作表不是在底部滑出来的，而是随

机出现在触发它的按钮的周围。此外，它还没有“取消”按钮，即便是在程序代码中定义了“取消”按钮，也不会显示它。



图4-50 iPhone中的操作表



图4-51 iPad中的操作表

下面我们看看图4-50所示的操作表的代码部分。修改ViewController.h的代码，具体如下：

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

@interface ViewController : UIViewController <UIAlertViewDelegate,UIActionSheetDelegate>

- (IBAction)testalertView:(id)sender;
- (IBAction)testActionSheet:(id)sender;

@end
```

在ViewController.h中，我们声明了UIActionSheetDelegate协议。虽然也可以直接在.m文件中实现该协议，但是从代码规范上讲，这样做程序的可读性会更好一些。UIActionSheetDelegate协议提供了actionSheet:clickedButtonAtIndex:方法，该方法在点击操作表中的按钮时调用。ViewController.m中的相关代码如下：

```
- (IBAction)testActionSheet:(id)sender {

    UIActionSheet *actionSheet = [[UIActionSheet alloc]
                                   initWithTitle:nil
                                   delegate:self
                                   cancelButtonTitle:@"取消"
                                   destructiveButtonTitle:@"破坏性按钮"
                                   otherButtonTitles:@"Facebook",@"新浪微博",nil];

    actionSheet.actionSheetStyle = UIActionSheetStyleAutomatic;
    [actionSheet showInView:self.view];

}

#pragma mark-- 实现UIActionSheetDelegate
- (void)actionSheet:(UIActionSheet *)actionSheet clickedButtonAtIndex:(NSInteger)buttonIndex {
    NSLog(@"buttonIndex = %i",buttonIndex);
}
```

在testActionSheet:方法中实例化UIActionSheet对象时，最常用的构造函数是initWithTitle:delegate: cancelButtonTitle:destructiveButtonTitle:otherButtonTitles:，本例中将delegate参数设定为self，即该操作表的委托对象为当前的视图控制器（ViewController）。cancelButtonTitle参数用于设置“取消”按钮的标题，在iPhone中它在最下面。destructiveButtonTitle参数用于设置“破坏性”按钮，它的颜色是红色的，如果没有“破坏性”按钮，可以将该参数设定为nil。“破坏性”按钮只能有一个，在最上面。otherButtonTitles参数是其他按钮，它是一个字符串数组，以nil结尾。

UIActionSheet的actionSheetStyle属性用于设定操作表的样式，这些样式如下所示。

- ❑ **UIActionSheetStyleAutomatic**。自动样式。
- ❑ **UIActionSheetStyleDefault**。默认样式。
- ❑ **UIActionSheetStyleBlackTranslucent**。半透明样式。
- ❑ **UIActionSheetStyleBlackOpaque**。透明样式。

为了响应点击按钮，需要在视图控制器上实现actionSheet:clickedButtonAtIndex:方法，其中clickedButtonAtIndex参数是按钮索引，从上到下依次是0,1,2…。

4.9 工具栏和导航栏

工具栏和导航栏的应用有很大的差别，但是有一个共同的特性，那就是其中都可以放置UIBarButtonItem。UIBarButtonItem是工具栏和导航栏中的按钮，在事件响应方面与UIButton类似。

4.9.1 工具栏

工具栏类为UIToolbar。在iPhone中，工具栏位于屏幕底部，其按钮数不能超过5个，如果超过5个，则第5个按钮（即最后一个）则是“更多”按钮，如图4-52所示。在iPad中，工具栏位于屏幕顶部，按钮的数量没有限制。

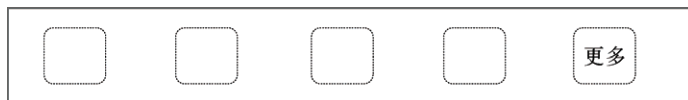


图4-52 在iPhone中，工具栏中的按钮

工具栏是工具栏按钮（UIBarButtonItem）的容器。在UIBarButtonItem中，除了我们看到的按钮外，还有“固定空格”和“可变空格”，它们的作用是在各个按钮之间插入一定的空间，如图4-53所示。这样处理以后，工具栏给用户的视觉效果会更好。

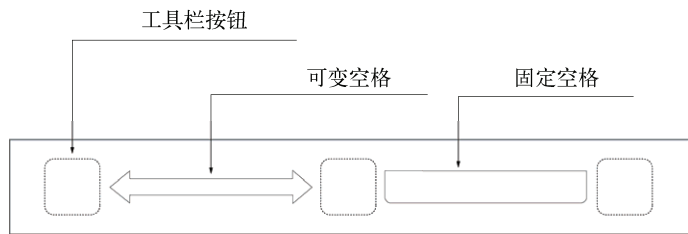


图4-53 工具栏中的“固定空格”和“可变空格”

在工具栏中，我们除了可以放置UIBarButtonItem外，还可以放置其他自定义视图，但这种操作只在特殊情况下才使用。下面我们用一个案例（其原型设计图如图4-54所示）来介绍一下工具栏的用法，其中工具栏中有两个按钮Save和Open，界面中央有一个标签，点击Save和Open按钮均会改变标签的内容。

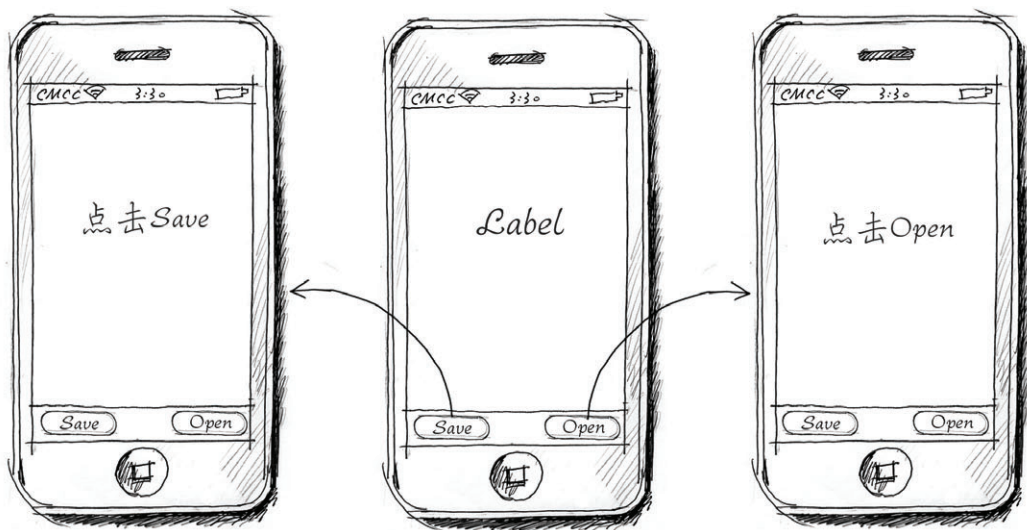


图4-54 工具栏案例原型设计图

使用Single View Application模板创建一个名为ToolbarSample的工程。打开Interface Builder设计界面，摆放两个按钮控件，如图4-55所示，从对象库中拖曳一个Toolbar到设计界面底部并将其摆放到合适的位置。

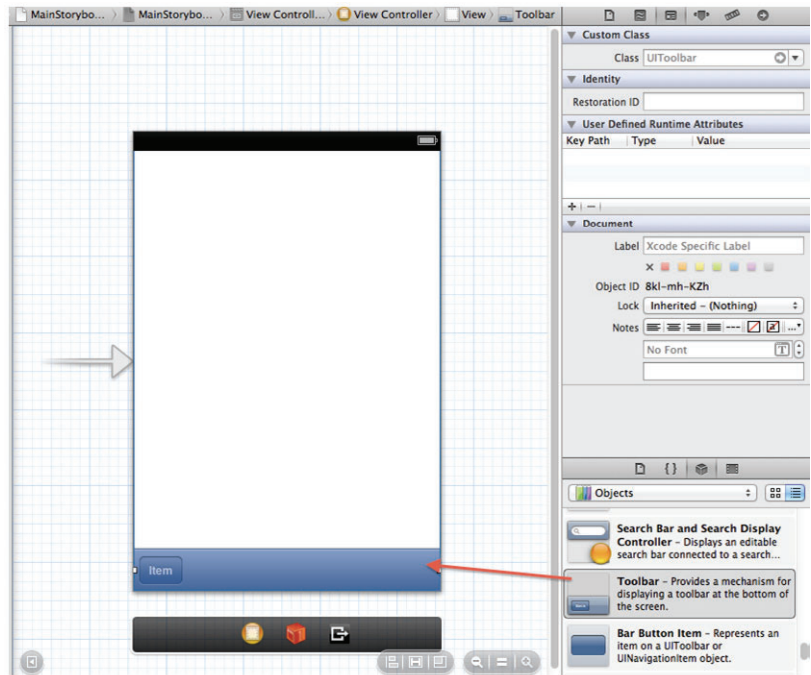


图4-55 在Interface Builder中添加工具栏

再拖曳一个工具栏按钮到工具栏，然后拖曳一个“可变空格”到两个按钮之间，如图4-56所示。
双击选中按钮，修改按钮上的标题。当然，也可以打开如图4-57所示的属性检查器，直接编辑Bar Item下的Title属性。如果想添加图片按钮，直接在属性检查器中修改Image属性即可。

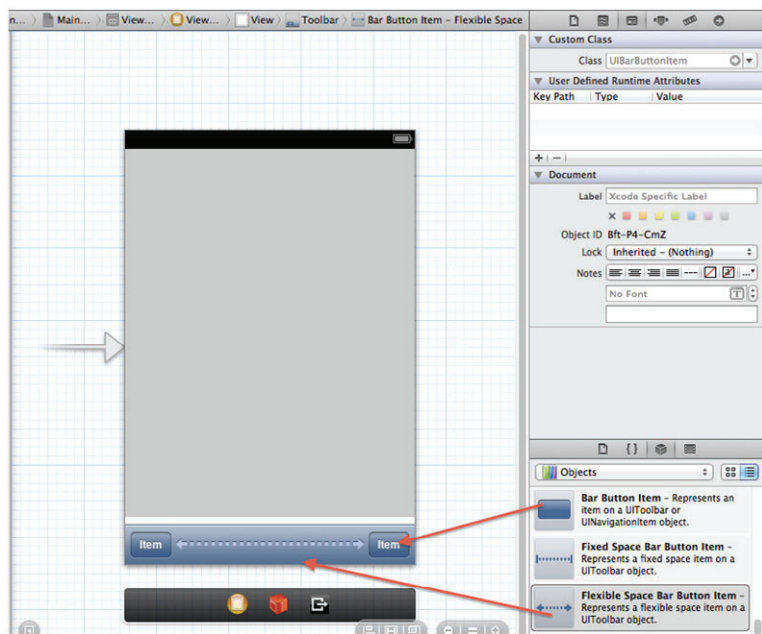


图4-56 在工具栏中添加按钮和“可变空格”

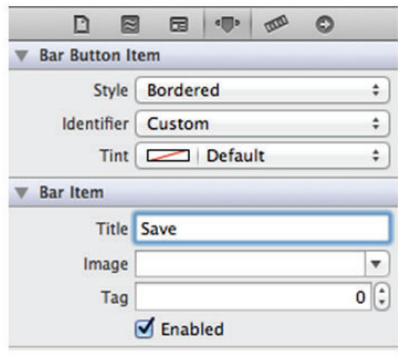


图4-57 工具栏按钮属性检查器

下面我们看看ViewController.h文件中的相关代码：

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

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UILabel *label;

- (IBAction)save:(id)sender;
- (IBAction)open:(id)sender;

@end
```

在上述代码中，我们定义了一个输出口类型的UILabel属性label、一个用于响应Save按钮点击事件的动作save:，以及用于响应Open按钮点击事件的动作open:。编写完ViewController.h代码后，还需要Interface Builder为输出口和动作事件连线。下面我们再看看相关的代码：

```
@implementation ViewController

- (IBAction)save:(id)sender {
    self.label.text = @"点击Save";
}

- (IBAction)open:(id)sender {
    self.label.text = @"点击Open";
}

@end
```

这两个方法对应于ViewController.h文件所定义的save:和open:方法，它们所做的事情是改变标签的内容。

4.9.2 导航栏

导航栏主要用于导航，考虑的是整个应用，而工具栏应用于当前界面，考虑的是局部界面。相关类和概念如下所示。

- ❑ **UINavigationController**。导航控制器，可以构建树形导航模式应用的根控制器，这将在后面章节中介绍。
- ❑ **UINavigationController**。导航栏，它与导航控制器是一一对应的关系。它管理一个视图栈，用来显示树形结构中的视图。
- ❑ **UINavigationController**。导航栏项目，在每个界面中都会看到。它分为左、中、右3个区域，左侧区域一般放置一个返回按钮（设定属性是backBarButtonItem）或左按钮（设定属性是leftBarButtonItem），右侧区域一般放置一个右按钮（设定属性是rightBarButtonItem），中间区域是标题（属性是title）或者提示信息（属性是prompt）。导航栏与导航栏项目是一对多的关系，如图4-58所示。导航栏的栈中存放的就是导航栏项目，处于栈顶的导航栏项目就是我们当前看到的导航栏项目。
- ❑ **UIBarButtonItem**。与工具栏中的按钮一样，它是导航栏中的左右按钮。

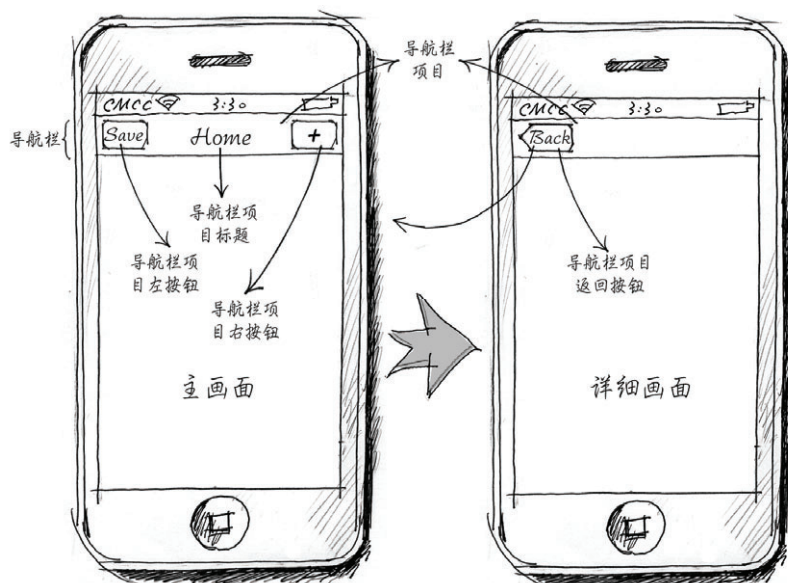


图4-58 导航栏和导航栏项目

下面我们用一个案例介绍一下导航栏的用法，该案例的原型设计图如图4-59所示。

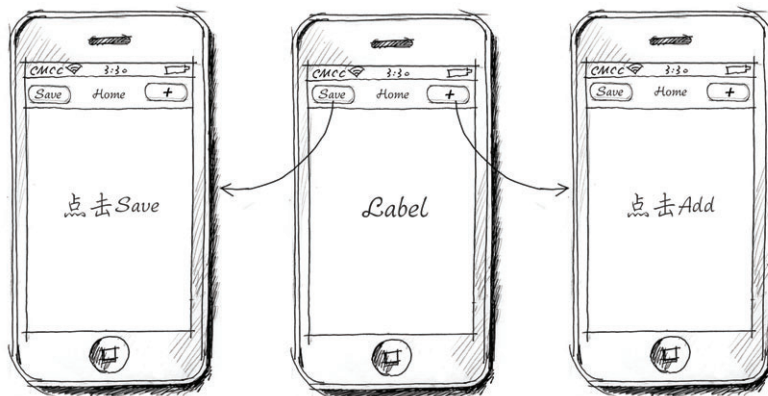


图4-59 导航栏案例原型设计图

在该导航栏中，共有两个按钮Save和+，界面中央有一个标签。点击Save和+按钮将改变标签的内容。需要说明的是，这里的Save和+按钮是iOS系统提供的标准按钮。标准按钮有着标准的用途和样式，这些按钮的用途可以在苹果HIG（iOS人机交互开发指南）文档中找到：http://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/MobileHIG/UIElementGuidelines/UIElementGuidelines.html#//apple_ref/doc/uid/TP40006556-CH13-SW41。

下面看看该案例的实现过程。使用Single View Application模板创建工程名为NavigationBarSample的应用，然后打开Interface Builder设计界面，并从对象库中拖曳一个Navigation Bar到设计界面顶部并将其摆放到合适的位置，如图4-60所示。

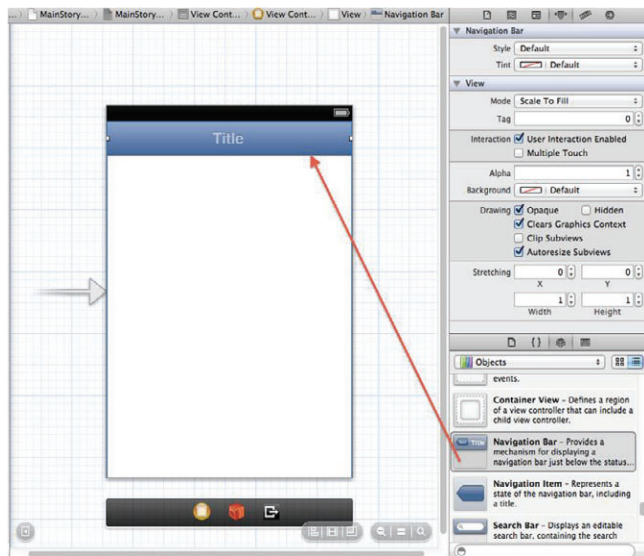


图4-60 在Interface Builder中添加导航栏

然后在导航栏项目中的左右两个区域分别拖曳一个Bar Button Item，为导航栏项目添加左右按钮，如图4-61所示。

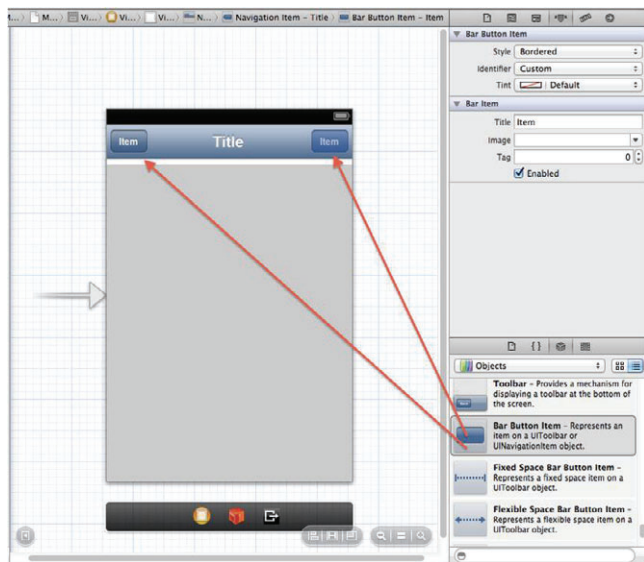



图4-61 导航栏项目左右按钮

选择左按钮，打开其属性检查器，在Bar Button Item选项组中，从Identifier中选择按钮类型，如图4-62所示。

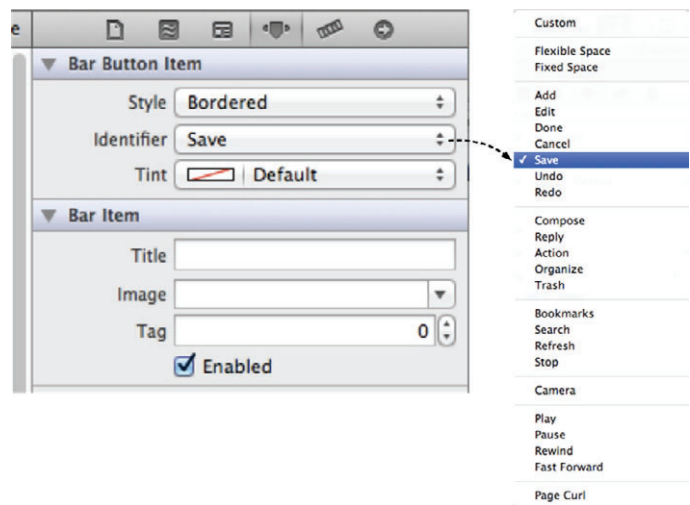


图4-62 选择导航按钮类型

需要说明的是，导航栏和工具栏中按钮的类型都可以通过这个属性检查器来选择，不需要我们自己设定名字。从苹果UI设计规范的角度来讲，这些按钮与要完成的功能一致，不能随意使用。

选择导航栏项目，打开其属性检查器，将Title属性修改为Home，如图4-63所示。

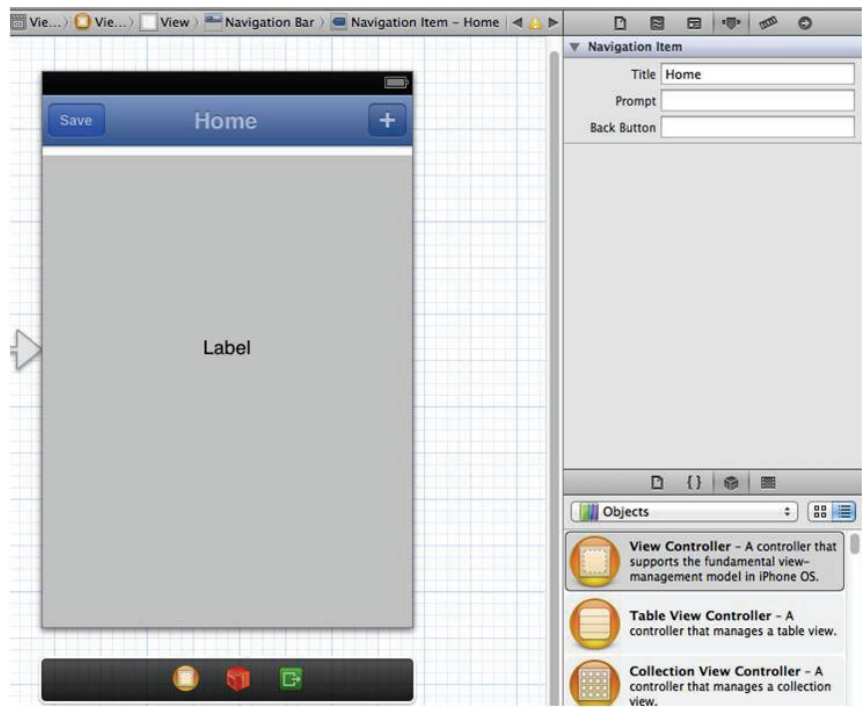


图4-63 修改导航栏项目标题

案例实现代码ViewController.h文件的内容如下：

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

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UILabel *label;

- (IBAction)save:(id)sender;
- (IBAction)add:(id)sender;

@end
```

在上述代码中，我们定义了一个输出口类型的UILabel属性label，用于响应Save按钮的点击事件save:，用于响应+按钮的点击事件add:。编写完代码后，还需要Interface Builder为输出口和动作事件连线。ViewController.m文件的内容如下：

```
@implementation ViewController

- (IBAction)save:(id)sender {
    self.label.text = @"点击Save";
}

- (IBAction)add:(id)sender {
    self.label.text = @"点击Add";
}

@end
```

这两个方法对应于ViewController.h文件中定义的save:和add:方法，主要用于改变标签的内容。一般情况下，如果涉及导航栏，都是多界面的应用，这是因为导航栏的用途就是导航，而单界面不需要导航。但是在本案例中，只有一个界面，主要关注导航栏和导航栏项目的用法。关于导航模式的相关内容，我们以后再介绍。

4.10 屏幕布局

移动应用比桌面应用更难开发，其中一个主要的原因是屏幕尺寸小。在iPhone和iPad这样小的设备上放置控件，需要缜密地思考。同样一个应用的iPad版本不应该是简单地将iPhone界面放大，而是需要重新布局。

4.10.1 iPad与iPhone屏幕布局

作为iOS开发者，需先了解一下iPad与iPhone的屏幕。第1代和第2代iPad屏幕的物理尺寸是9.7英寸，分辨率是1024×768像素，第3代iPad采用视网膜屏幕技术，分辨率达到了2048×1536像素，而iPad mini的分辨率则达到了1024×768像素。在iPhone 4之前，屏幕的物理尺寸为3.5英寸，屏幕分辨率是480×320像素。iPhone 4和iPhone 4S采用视网膜屏幕技术，屏幕的物理尺寸为3.5英寸，分辨率达到了960×640像素。iPhone 5采用视网膜屏幕技术，屏幕物理尺寸为4英寸，分辨率是1136×640像素。

注意 在Interface Builder设计器中，屏幕或控件的尺寸以点（point）为单位。在视网膜屏幕技术中，1个点包括了4个像素，而没有采用视网膜屏幕技术的还是1个点包括1个像素。因此，也可以说iPhone 5之前的手机的分辨率是480×320点，iPhone 5的分辨率是568×320点。为了方便设计，若无特殊说明，屏幕或控件尺寸的单位是“点”。

在iPad和iPhone屏幕布局中，一般会有状态栏、工具栏、导航栏以及内容视图部分，它们的尺寸也是固定的。

如图4-64所示，在iPhone竖屏幕中，状态栏占用20点，导航栏（或工具栏）占用44点，标签栏占用49点。实际上，这些尺寸在iPhone横屏幕和iPad上也保持不变。

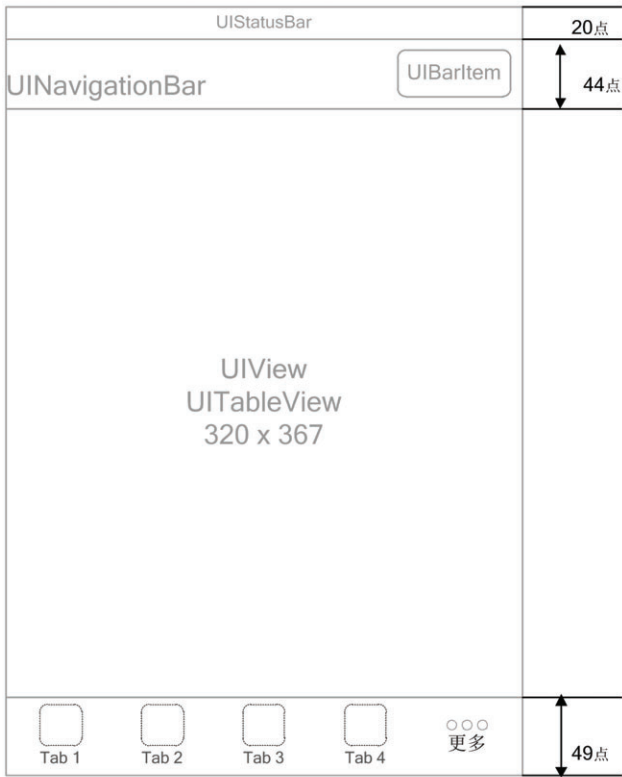


图4-64 状态栏、工具栏、导航栏以及内容视图的尺寸

在开发应用时，确定屏幕布局之后，内容视图区域的大小也就能确定下来了，然后再根据这个大小设计图片等。

4.10.2 绝对布局和相对布局

在iOS中，布局管理主要涉及绝对布局和相对布局这两个问题。绝对布局就是视图或控件在屏幕中的位置绝对，它的大小也是绝对的。即使它所在的父容器视图大小变化或者屏幕旋转了，它的位置也不变。相对布局在上述情况下，它的位置是变化的。

使用绝对布局还是相对布局主要取决于应用场景。对于下面的场景：一个开关控件被摆放在`UIView`中，它与父容器（`UIView`）的上边距是绝对400点，如图4-65左图所示。当屏幕旋转时，由于开关控件与它的父容器的上边距是绝对400点，因此开关控件会超出屏幕，我们就看不到这个控件了，如图4-65右图所示，此时我们需要使用相对布局技术。

下面我们看另外一个场景。如图4-66左图所示，一个开关控件被摆放在`UIView`中，它与父容器（`UIView`，320×480）的上边距是相对400点，当父容器视图的高度增大为500时，开关控件就会下移，上边距大于400点，如图4-66中图所示。如果这个变化足够大，这个控件就会超出屏幕。相反，如果将父容器视图的高度降低为400，开关控件就会上移，上边距小于400点，如图4-66右图所示，如果这个变化足够大，屏幕中的控件都会挤压一起。

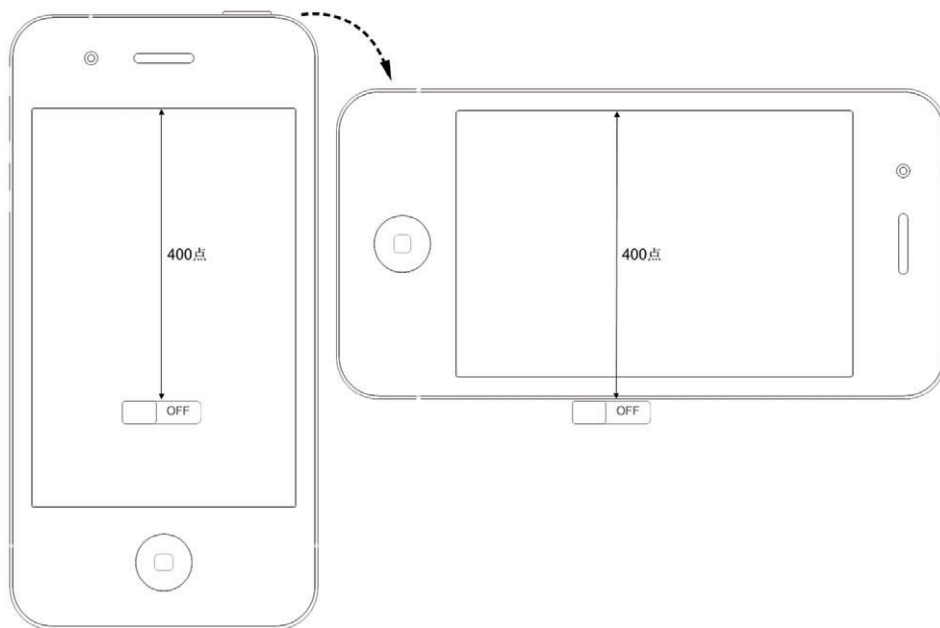


图4-65 使用绝对布局会出问题的场景

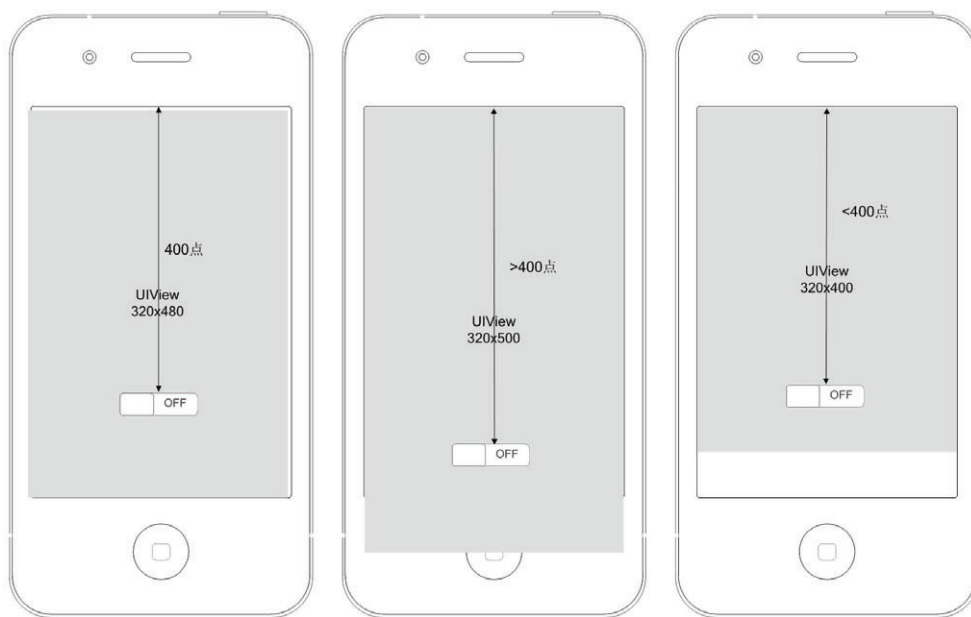


图4-66 使用相对布局会出问题的场景

那么,如何来设定视图的相对布局和绝对布局?事实上,在iOS 6之后,我们推出了新的解决方案——Autolayout布局技术,这会在4.10.3节中介绍,本节还是采用以前的设定方式。使用Xcode创建的xib或故事板文件默认采用了Autolayout布局,本节设定的时候需要去掉这个选项。如图4-67所示,选择xib或故事板文件,打开文件检查器,取消选中Use Autolayout复选框。

然后选中改变布局的控件，打开其尺寸检查器，如图4-68所示。在View的Autosizing属性中，虚线线段代表的是相对距离，实线线段代表的是绝对距离，点击可以互相切换。

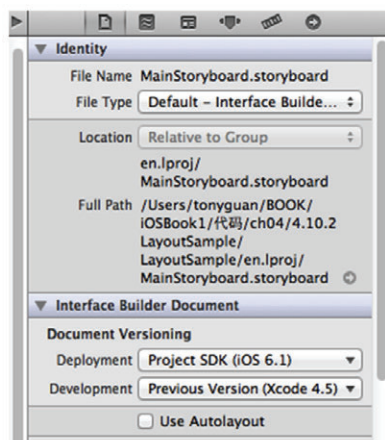


图4-67 取消选中Use Autolayout复选框

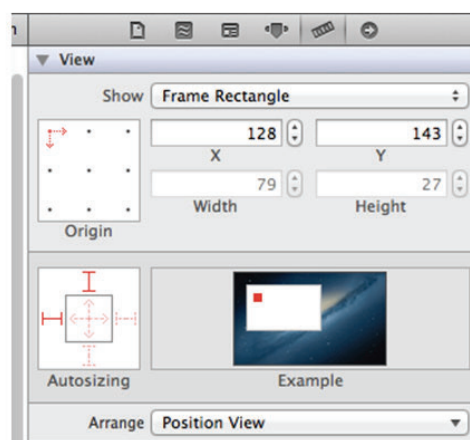


图4-68 View的Autosizing属性

Autosizing属性旁边是效果，其中红色小方块代表当前控件的位置。按照上文所述的方法，将图4-65所示的开关控件修改为相对布局，如图4-69所示。

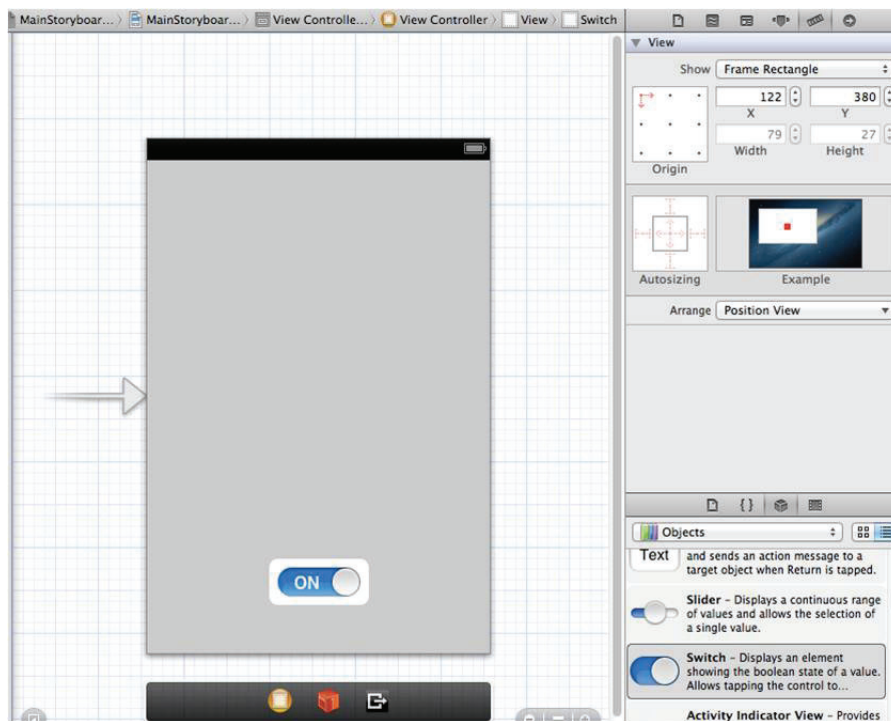


图4-69 将开关控件修改为相对布局

运行一下，结果如图4-70所示，再旋转屏幕（command+方向键），看看开关控件是否超出屏幕！

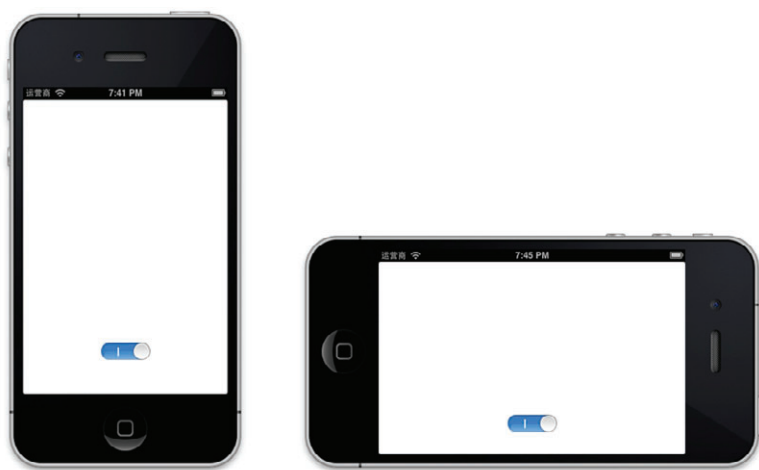


图4-70 运行结果

4.10.3 使用AutoLayout布局

AutoLayout (Cocoa Auto Layout) 技术最早应用于Mac OS X 10.7 下的开发, 现在成为了iOS 6的新特性, 它可以帮助我们解决相对布局和绝对布局的问题。AutoLayout为空间布局定义了一套约束 (constraint), 约束定义了控件与视图之间的关系。约束定义可以通过Interface Builder或代码实现, 因为通过Interface Builder设定约束相对简单直观, 所以本书重点向读者推荐这种方式。

下面我们通过一个例子向读者介绍约束的用法。如图4-71左图所示, 界面中有三个按钮, 将屏幕向右旋转至横屏时, 效果图如4-71右图所示, 可以发现, 这3个按钮仍然能够很好地摆放在屏幕之中。

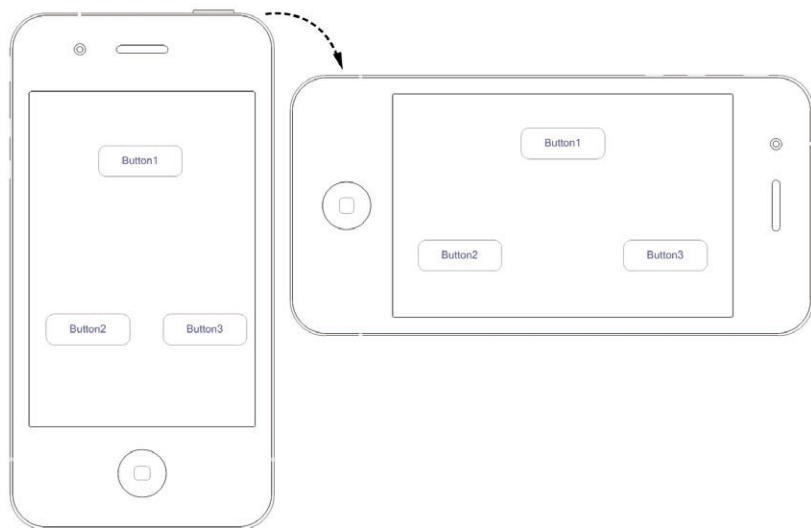


图4-71 案例原型设计图

这个案例的具体实现为: 使用Single View Application模板创建一个名为AutolayoutSample的工程, 打开Interface Builder设计界面, 从对象库中拖曳3个按钮到设计界面。打开View Controller Scene视图中View下面的Constraints项, 我们发现其中有6个约束, 如图4-72所示。

这6个AutoLayout视图约束的说明如下所示。

- ❑ 第一个约束。用于说明Button1按钮与父视图在垂直方向上的距离为91点。
- ❑ 第二个约束。说明Button1按钮在父视图的X轴上居中。
- ❑ 第三个约束。说明Button2按钮与父视图在垂直方向上的距离为44点。
- ❑ 第四个约束。说明Button2按钮与父视图在水平方向上的距离为46点。
- ❑ 第五个约束。说明Button3按钮与Button2按钮在垂直方向上的位置相同。
- ❑ 第六个约束。说明Button3按钮与父视图在水平方向上的距离为28点。

要改动它们之间的约束关系，直接在Interface Builder中拖动即可。如果想更加精确，可以通过约束的属性检查器设定，这里我们选择了第三个约束作为演示，如图4-73所示。

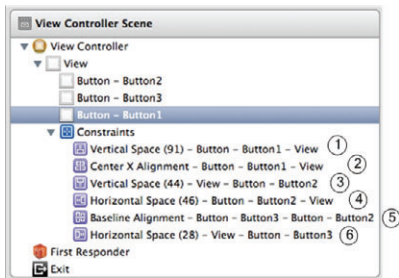


图4-72 AutoLayout视图约束

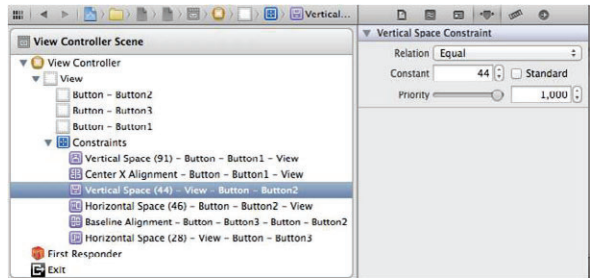


图4-73 约束属性检查器

在垂直距离的约束属性设定中，Relation是指设定的距离之间的关系，包括3个选项——等于、大于等于和小于等于，Constant是约束数值，Priority是约束等级。当有相同的约束作用于两个视图之间时，等级高的约束优先。设定好约束之后，我们就可以运行一下看看是否可以达到预期的效果。

4.10.4 旋转你的屏幕

在iPhone官方提供的应用中，有一个计算器应用，竖屏时它是一个简单的计算器，如图4-74所示。横屏时，它是一个复杂的带有科学计算功能的计算器，如图4-75所示。



图4-74 竖屏时的计算器应用



图4-75 横屏时的计算器应用

从计算器的应用可以看出，横屏和竖屏时分别采用了不同的视图，而不是同一个视图界面元素的重新布局。这里我们对图4-71所示的场景进行一些修改，如图4-76所示，竖屏时屏幕中有3个按钮，横屏时屏幕中有4个按钮。

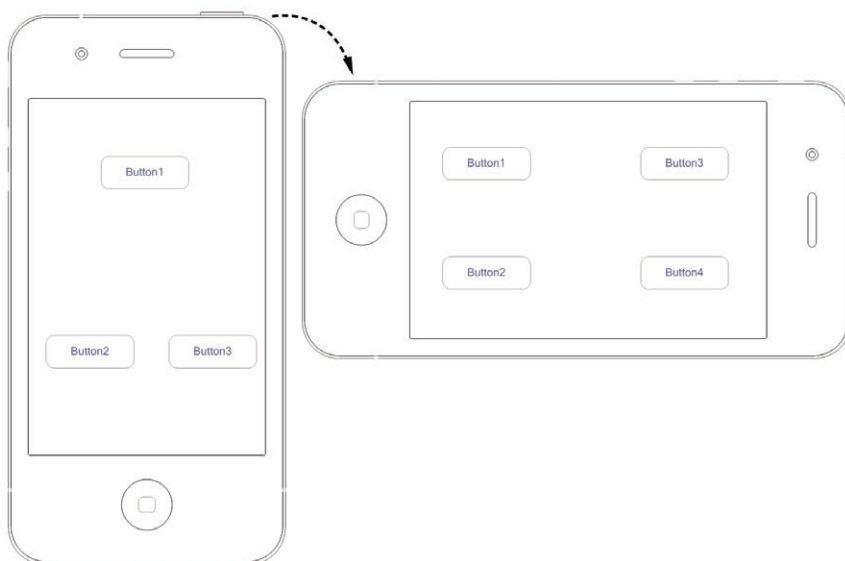


图4-76 案例原型设计图

下面我们看看如何实现这个应用。使用Single View Application模板创建一个名为ScreenRotateSample的工程。打开MainStoryboard.storyboard设计界面，从对象库中拖曳3个按钮到设计界面，将其作为竖屏界面。

那么，横屏界面如何实现呢？事实上，就是创建了一个新视图，具体实现方式有如下3种。

- ❑ 创建一个视图的xib文件，并使用Interface Builder设计这个视图的xib文件，然后从xib文件实例化视图对象。
- ❑ 创建一个带有xib文件的视图控制器，并使用Interface Builder设计这个xib文件，通过initWithNibName:bundle:构造函数实例化视图控制器，然后通过视图控制器的view属性获得视图对象。
- ❑ 创建一个故事板和一个视图控制器，并使用Interface Builder设计故事板文件，通过故事板的storyboardWithName:bundle:方法实例化视图控制器，然后通过视图控制器的view属性获得视图对象。

由于本书案例主要采用故事板编写，所以这里我们只介绍故事板的实现过程。首先，我们需要创建一个故事板文件，然后选择菜单File→New→File...，此时会打开Choose a template for your new file界面，在User Interface中选择Storyboard文件模板（如图4-77所示）。

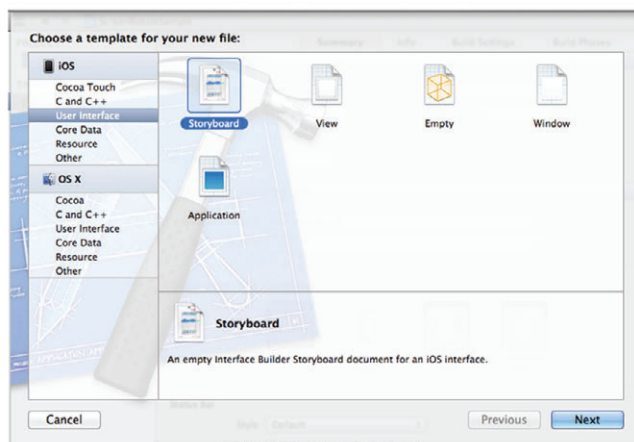



图4-77 选择文件模板

点击Next按钮，在打开的界面中选择平台为iPhone，再点击Next按钮，输入文件名LandscapeStoryboard，创建完成后，在Interface Builder中打开LandscapeStoryboard.storyboard文件的设计界面。选择视图，打开其属性检查器，选择Simulated Metrics→Orientation属性为Landscape，如图4-78所示，这样设计视图就会横屏摆放了。

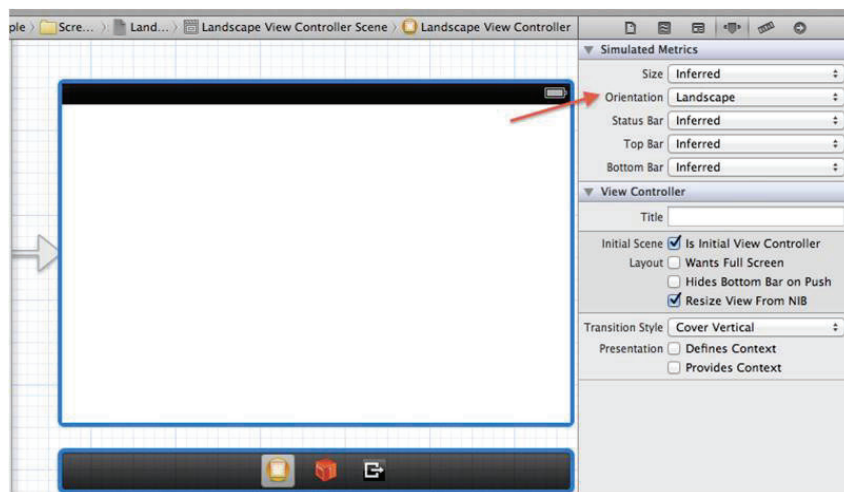


图4-78 选择横屏摆放

从对象库中拖曳4个按钮到设计界面，接着再创建视图控制器文件LandscapeViewController，注意不要带有xib文件。再用Interface Builder打开LandscapeStoryboard.storyboard文件，点选Landscape View Controller Scene中的Landscape View Controller，打开其标识检查器，如图4-79所示，在Custom Class中选择Class为LandscapeViewController，在下面的Storyboard ID中输入LandscapeViewController。

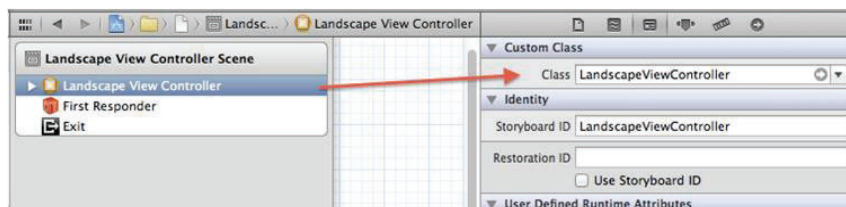


图4-79 选择视图控制器

设计界面工作完成了，我们再来看看代码部分。ViewController.h的相关代码如下：

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

@interface ViewController : UIViewController

@property (nonatomic, strong) UIView *mainPortraitView;
@property (nonatomic, strong) UIView *mainLandscapeView;

@end
```

可以发现，其中定义了竖屏视图属性mainPortraitView和横屏视图属性mainLandscapeView。我们再来看看代码部分，ViewController.m的相关代码如下：

```
@implementation ViewController
- (void)viewDidLoad
{
    [super viewDidLoad];
```



```

LandscapeViewController *landscapeViewController = [[UIStoryboard
    storyboardWithName:@"LandscapeStoryboard" bundle:NULL]
    instantiateViewControllerWithIdentifier:@"LandscapeViewController"];

(self.mainLandscapeView) = landscapeViewController.view;
self.mainPortraitView = self.view;
}

- (BOOL)shouldAutorotate
{
    return YES;
}

- (NSUInteger)supportedInterfaceOrientations
{
    return UIInterfaceOrientationMaskAllButUpsideDown;
}

- (void)willAnimateRotationToInterfaceOrientation:(UIInterfaceOrientation)
    toInterfaceOrientation duration:(NSTimeInterval)duration
{
    if (toInterfaceOrientation == UIInterfaceOrientationLandscapeRight) {
        self.view=self.mainLandscapeView;
        self.view.transform=CGAffineTransformMakeRotation(deg2rad*(90));
        self.view.bounds=CGRectMake(0.0, 0.0, 480.0, 300.0);
    } else if (toInterfaceOrientation == UIInterfaceOrientationLandscapeLeft) {
        self.view=self.mainLandscapeView;
        self.view.transform=CGAffineTransformMakeRotation(deg2rad*(-90));
        self.view.bounds=CGRectMake(0.0, 0.0, 480.0, 300.0);
    } else {
        self.view=self.mainPortraitView;
        self.view.transform=CGAffineTransformMakeRotation(deg2rad*(0));
        self.view.bounds=CGRectMake(0.0, 0.0, 320.0, 460.0);
    }
    [super willRotateToInterfaceOrientation:toInterfaceOrientation
        duration:duration];
}

@end

```

在viewDidLoad方法中，我们总是需要初始化竖屏视图属性和横屏视图属性。在横屏视图中，我们需要先创建视图控制器（LandscapeViewController）对象，然后才能得到它的视图。创建视图控制器（LandscapeViewController）对象时，我们使用UIStoryboard的类方法storyboardWithName: bundle:（它的第一个参数是故事板的名字）获得故事板，然后再通过UIStoryboard的实例方法instantiateViewControllerWithIdentifier:（其参数是视图控制器在故事板中的标识LandscapeViewController）获得视图控制器实例。

shouldAutorotate方法用于指定当前视图是否支持旋转，supportedInterfaceOrientations方法具体指定视图支持哪个方向的旋转。在iPad中，shouldAutorotate方法的返回值默认是UIInterfaceOrientationMaskAll常量，表示支持所有方向。在iPhone中，其返回值默认是UIInterfaceOrientationMaskAllButUpsideDown常量，支持除了竖直向下以外的其他3个方向。视图支持某个方向的前提是设备也支持该方向。

willAnimateRotationToInterfaceOrientation:duration:方法在屏幕旋转之前触发，其参数toInterfaceOrientation可以判断屏幕的旋转方向，是枚举UIInterfaceOrientation中定义的常量。UIInterfaceOrientation枚举包括如下成员变量。

❑ **UIInterfaceOrientationPortrait**（或**UIDeviceOrientationPortrait**）。垂直向上，如图4-80中图1所示。

- ❑ **UIInterfaceOrientationPortraitUpsideDown** (或**UIDeviceOrientationPortraitUpsideDown**)。垂直向下, 如图4-80中图2所示。
- ❑ **UIInterfaceOrientationLandscapeLeft** (或**UIDeviceOrientationLandscapeLeft**)。水平向左, 如图4-80中图3所示。
- ❑ **UIInterfaceOrientationLandscapeRight** (或**UIDeviceOrientationLandscapeRight**)。水平向右, 如图4-80中图4所示。

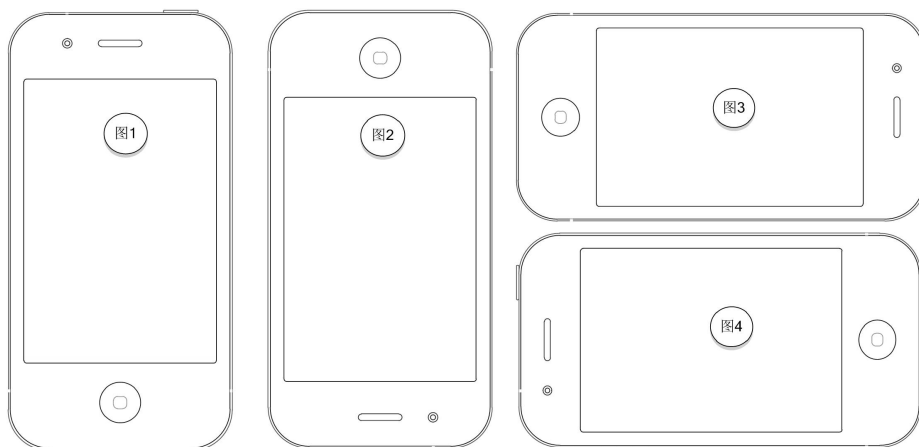


图4-80 iPhone设备方向

在判断向某个方向的旋转时, 不仅将要旋转的视图替换为当前视图, 还要把视图进行旋转变换并重新调整视图的边界。我们可以将其中一个判断 (如`toInterfaceOrientation == UIInterfaceOrientationLandscapeRight`) 中的视图旋转变换和调整视图语句去掉:

```
if (toInterfaceOrientation == UIInterfaceOrientationLandscapeRight) {
    self.view=self.mainLandscapeView;

    self.view.transform=CGAffineTransformMakeRotation(deg2rad*(90));
    self.view.bounds=CGRectMake(0.0, 0.0, 480.0, 300.0);
}
```

此时我们的运行结果如图4-81所示, 设备虽然成功旋转但是里面的视图没有旋转。

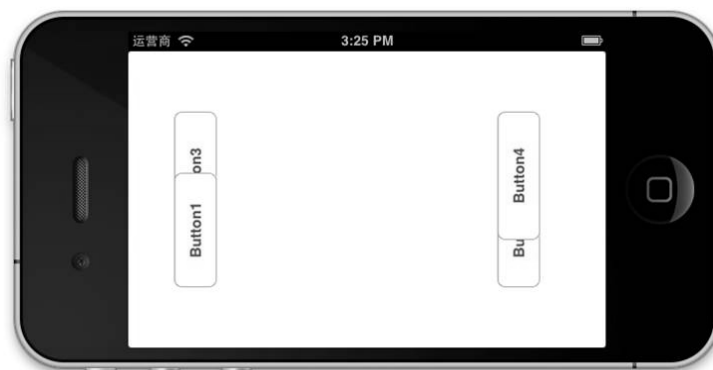


图4-81 旋转变换

在上述代码中，CGAffineTransformMakeRotation函数是放射变换函数，用于视图的二维坐标变换，类似的还有缩放和位移放射变换函数，其中的参数是弧度。本例中我们定义了一个宏deg2rad，用于实现度与弧度的转换。

4.11 选择器

玩过老虎机吗？选择器的外形很像是一台老虎机，当拨动选择器时，它还会像老虎机一样发出“咔咔”的声音，还有真实的拨盘旋转的感觉。虽然它看起来像老虎机，但是它并不是用来娱乐的，而是iOS中的标准控件，主要用于为用户提供选择。在软件领域，有句话很经典：“有输入的地方，就要验证。”当你的界面是用户注册界面时，其中有一个“出生日期”字段，你是给用户一个文本框吗？如果是，他可能会输入类似“2012-1-32”这样不合法的日期，我们需要确保输入内容的合法性。为了方便操作，我们希望用户以选择的方式完成信息输入，此时选择器便应运而生了。

4.11.1 日期选择器

日期是最复杂的，为此iOS推出了UIDatePicker日期选择器，它可以实现对日期的选择。日期选择器有4种模式：日期、日期时间、时间和倒计时定时器，如图4-82 ~ 图4-85所示。



图4-82 日期模式

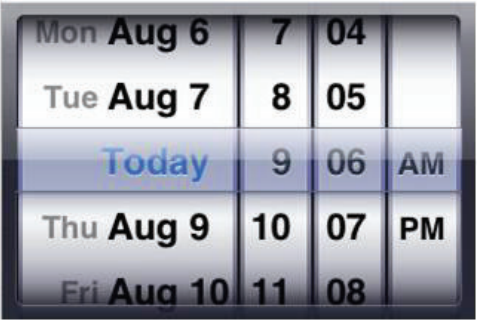


图4-83 日期时间模式



图4-84 时间模式



图4-85 倒计时定时器模式

下面我们通过图4-86所示的案例来学习日期选择器，界面中有日期选择器、一个标签和一个按钮。点击该按钮，会将选中的日期显示在标签上，具体的实现方式如下。

使用Single View Application模板创建一个名为DatePickerSample的工程。打开MainStoryboard.storyboard设计界面，从对象库中拖曳控件到设计界面，如图4-87所示。

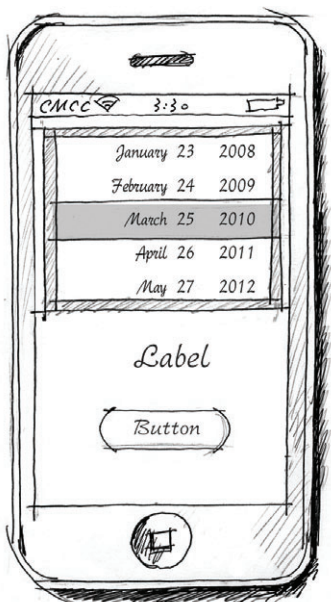


图4-86 案例原型草图

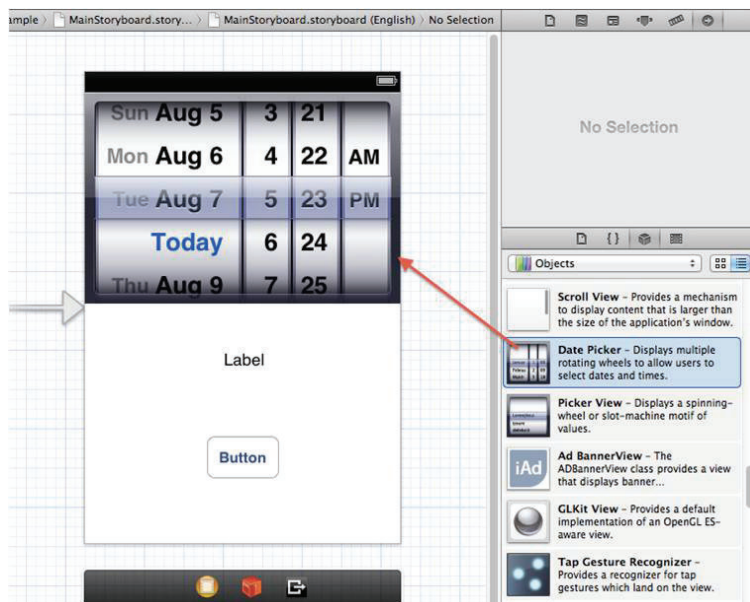



图4-87 Interface Builder设计界面

选择Date Picker，打开其属性检查器，如图4-88所示。

这些属性项的含义如下所示。

- ☐ **Mode**。设定日期选择器的模式。
- ☐ **Local**。设定本地化，日期选择器会按照本地习惯和文字显示日期。
- ☐ **Interval**。设定间隔时间，单位为分钟。
- ☐ **Date**。设定开始时间。
- ☐ **Constraints**。设定能显示的最大和最小日期。
- ☐ **Timer**。在倒计时定时器模式下倒计时的秒数。

下面我们看看ViewController.h中的相关代码，具体如下：

```
@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UIDatePicker *datePicker;
@property (weak, nonatomic) IBOutlet UILabel *label;

- (IBAction)onclick:(id)sender;

@end
```

在上述代码中，我们定义了输出口属性UIDatePicker和UILabel，以及一个按钮点击事件onclick:。下面我们看看ViewController.m中的相关代码，具体如下：

```
@implementation ViewController

- (IBAction)onclick:(id)sender {

    NSDate * theDate = self.datePicker.date;
    NSLog(@"the date picked is: %@", [theDate descriptionWithLocale:
    [NSLocale currentLocale]]);
    NSDateFormatter * dateFormatter = [[NSDateFormatter alloc] init] ;
```



图4-88 Date Picker属性检查器

```

[dateFormatter setDateFormat:@"%YYYY-MM-dd HH:mm:ss"];
NSLog(@"the date formate is: %@", [dateFormatter stringFromDate:theDate]);
self.label.text = [dateFormatter stringFromDate:theDate];
}
@end

```

UIDatePicker的date属性返回NSDate数据,就是控件中选中的时间,NSDate的descriptionWithLocale:返回基于本地化的日期信息。日期显示一般需要格式化输出的,本例中设定为YYYY-MM-dd HH:mm:ss。

4.11.2 普通选择器

有时候,我们可能还需要输入除了日期之外的其他内容,比如籍贯。籍贯要选择省,省下面还要有市等信息,普通选择器UIPickerView就能够满足用户的这些需要。UIPickerView是UIDatePicker的父类,它非常灵活,拨盘的个数可以设定,每一个拨盘的内容也可以设定。与UIDatePicker不同的是,UIPickerView需要两个非常重要的协议——UIPickerViewDataSource和UIPickerViewDelegate。

下面我们通过一个案例学习一下普通选择器的用法。图4-89是选择“籍贯”界面,其中有一个选择器、一个标签和一个按钮,第一个拨轮是所在的省,第二个拨轮是这个省下面可以选择的市。点击其中的按钮,可以将选择器中选中的两个拨轮内容显示在标签上,具体的实现过程如下所示。

使用Single View Application模板创建一个名为PickerViewSample的工程。打开MainStoryboard.storyboard设计界面,从对象库中拖曳控件到设计界面,如图4-90所示。

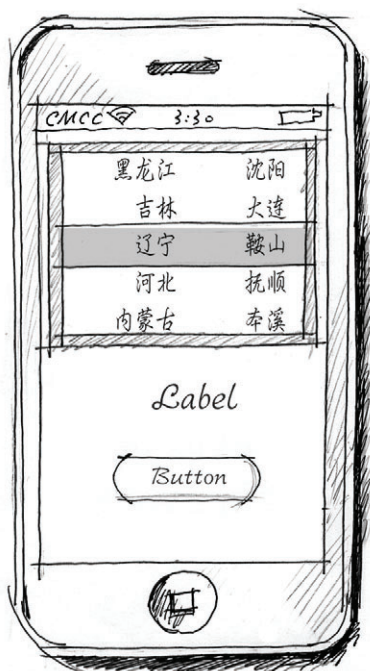


图4-89 案例原型草图

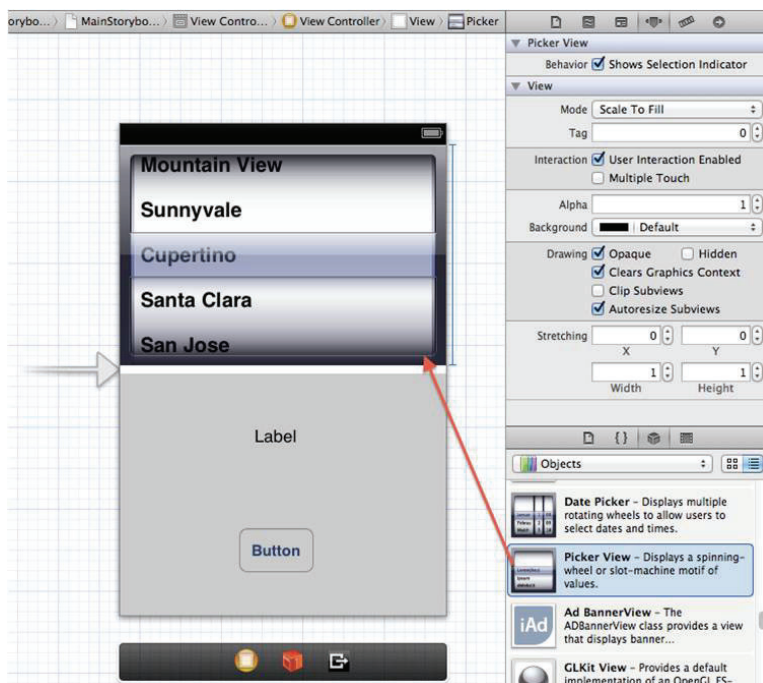


图4-90 Interface Builder设计界面

该案例中省份和市的数据是联动的,即选择了省份后,与它对应的市也会跟着一起变化,省市的信息放在provinces_cities.plist文件中,这个文件采用字典结构,如图4-91所示。

Key	Type	Value
▼ Root	Dictionary	(3 items)
▼ 吉林省	Array	(8 items)
Item 0	String	长春
Item 1	String	吉林
Item 2	String	四平
Item 3	String	辽源
Item 4	String	通化
Item 5	String	白山
Item 6	String	松原
Item 7	String	白城
▶ 黑龙江省	Array	(12 items)
▶ 辽宁省	Array	(14 items)

图4-91 provinces_cities.plist文件

下面我们看看ViewController.h文件中的相关代码：

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

@interface ViewController : UIViewController
<UIPickerViewDelegate, UIPickerViewDataSource>

@property (weak, nonatomic) IBOutlet UIPickerView *pickerView;
@property (weak, nonatomic) IBOutlet UILabel *label;

@property (nonatomic, strong) NSDictionary *pickerData;    //保存全部数据
@property (nonatomic, strong) NSArray *pickerProvincesData; //当前的省数据
@property (nonatomic, strong) NSArray *pickerCitiesData;   //当前省下面的市数据

- (IBAction)onclick:(id)sender;

@end
```

这里定义了输出口属性UIPickerView和UILabel，还有一个动作事件onclick:，用于响应按钮点击事件。装载数据的属性pickerData是字典类型，用来保存从provinces_cities.plist文件中读取的全部内容。

pickerProvincesData是数组类型，保存了全部的省份信息。pickerCitiesData也是数组类型，保存了当前选中省份下的市的信息。我们再看看ViewController.m中数据加载部分的代码：

```
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    NSBundle *bundle = [NSBundle mainBundle];
    NSString *plistPath = [bundle pathForResource:@"provinces_cities"
                                                ofType:@"plist"];

    //获取属性列表文件中的全部数据
    NSDictionary *dict = [[NSDictionary alloc] initWithContentsOfFile:plistPath];
    self.pickerData = dict;

    //省份名数据
    self.pickerProvincesData = [self.pickerData allKeys];

    //默认取出第一个省的所有市的数据
    NSString *seletedProvince = [self.pickerProvincesData objectAtIndex:0];
    self.pickerCitiesData = [self.pickerData objectForKey:seletedProvince];
}

@end
```


viewDidLoad方法实现了加载数据到成员变量中。当用户点击按钮时的代码如下：

```
- (IBAction)onclick:(id)sender {

    NSInteger row1 = [self.pickerView selectedRowInComponent:0];
    NSInteger row2 = [self.pickerView selectedRowInComponent:1];
    NSString *selected1 = [self.pickerProvincesData objectAtIndex:row1];
    NSString *selected2 = [self.pickerCitiesData objectAtIndex:row2];

    NSString *title = [[NSString alloc] initWithFormat:@"%@@, %@市",
                      selected1,selected2];

    self.label.text = title;

}
```

UIPickerView的Component就是指拨盘，selectedRowInComponent方法返回拨盘中被选定的行的索引，索引是从0开始的。

4.11.3 数据源协议与委托协议

与UITextField控件不同，UIPickerView和UITableView等复杂控件除了委托协议外，还有数据源协议。UIPickerView的委托协议是UIPickerViewDelegate，数据源是UIPickerViewDataSource。

在上一章中，我们介绍委托设计模式的时候也提到过数据源。数据源与委托一样，都是委托设计模式的具体实现，只不过它们的角色不同：委托对象负责控制控件外观，如选择器的宽度、选择器的行高等信息，此外，还负责对控件的事件和状态变化作出反应。数据源对象是控件与应用数据（模型）的桥梁，如选择器的行数、拨轮数等信息。委托中的方法在实现时是可选的，而数据源中的方法一般是必须实现的。

UIPickerViewDataSource中的方法有如下两种。

❑ **numberOfComponentsInPickerView:**。为选择器中拨轮的数目。

❑ **pickerView:numberOfRowsInComponent:**。为选择器中某个拨轮的行数。

在ViewController.m中，UIPickerViewDataSource的实现代码如下：

```
#pragma mark 实现协议UIPickerViewDataSource方法
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView {
    return 2;
}

- (NSInteger)pickerView:(UIPickerView *)pickerView
  numberOfRowsInComponent:(NSInteger)component {
    if (component == 0) { //省份个数
        return [self.pickerProvincesData count];
    } else { //市的个数
        return [self.pickerCitiesData count];
    }
}
```

UIPickerViewDelegate中的常用方法有如下两个。

❑ **pickerView:titleForRow:forComponent:**。为选择器中某个拨轮的行提供显示数据。

❑ **pickerView:didSelectRow:inComponent:**。选中选择器的某个拨轮中的某行时调用。

在ViewController.m中，UIPickerViewDelegate的实现代码如下：

```
#pragma mark 实现协议UIPickerViewDelegate方法
- (NSString *)pickerView:(UIPickerView *)pickerView
  titleForRow:(NSInteger)row forComponent:(NSInteger)component {
    if (component == 0) { //选择省份名
        return [self.pickerProvincesData objectAtIndex:row];
    } else { //选择市名
        return [self.pickerCitiesData objectAtIndex:row];
    }
}
```



```

    }
}

- (void)pickerView:(UIPickerView *)pickerView
  didSelectRow:(NSInteger)row inComponent:(NSInteger)component {
    if (component == 0) {
        NSString *seletedProvince = [self.pickerProvincesData objectAtIndex:row];
        NSArray *array = [self.pickerData objectForKey:seletedProvince];
        self.pickerCitiesData = array;
        [self.pickerView reloadComponent:1];
    }
}

```

最后，不要忘记将委托和数据源的实现对象ViewController分配给UIPickerView的委托属性delegate和数据源属性dataSource，这可以通过代码或Interface Builder进行分配。下面的代码就是来实现分配的：

```

- (void)viewDidLoad
{
    .....
    self.pickerView.dataSource = self;
    self.pickerView.delegate = self;
}

```

在Interface Builder中分配的过程是：打开故事板文件，右击选择器，弹出的右键菜单如图4-92所示，将Outlets→dataSource后面的小圆点拖曳到左边的View Controller上，然后以同样的方式将Outlets→delegate后面的小圆点拖曳到左边的View Controller上。

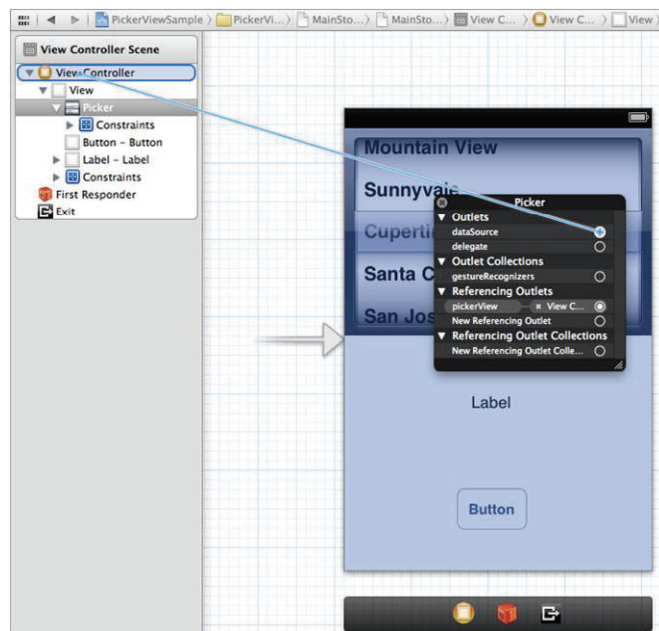


图4-92 在Interface Builder中分配数据源和委托

数据源和委托是非常重要的内容，在学习表视图时我们还会用到它们。

4.12 iOS 6 中的集合视图

为了增强网格视图开发，iOS 6中开放了集合视图API。这种网格视图的开源代码在开源社区中很早就有，但是都比较麻烦，而iOS 6的集合视图API使用起来却非常方便。

4.12.1 集合视图介绍

图4-93显示了集合视图的组成，它有4个重要的组成部分。

- ❑ 单元格。它是集合视图中的一个单元格。
- ❑ 节。它是集合视图中的一个行数据，由多个单元格构成。
- ❑ 补充视图。它是节的头和脚。
- ❑ 装饰视图。集合视图中的背景视图。

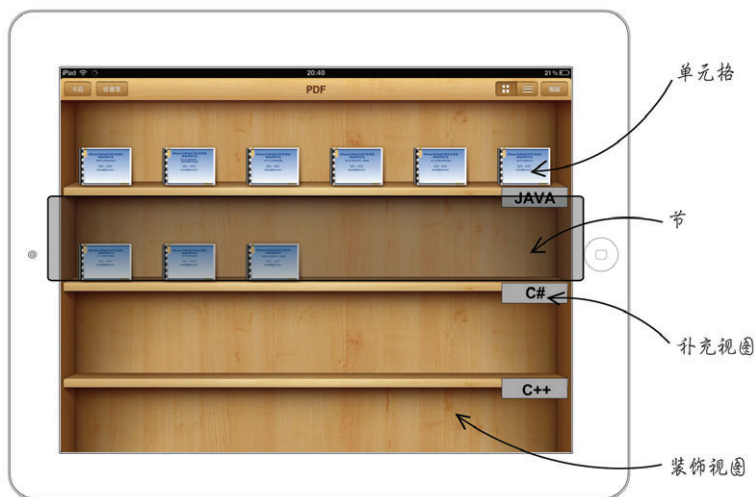


图4-93 集合视图组成

集合视图类的构成如图4-94所示，可以看到，UICollectionView继承UIScrollView。与选择器类似，集合视图也有两个协议：UICollectionViewDelegate委托协议和UICollectionViewDataSource数据源协议。UICollectionViewCell是单元格类，它的布局是由UICollectionViewLayout类定义的，它是一个抽象类。UICollectionViewFlowLayout类是UICollectionViewLayout类的子类。对于复杂的布局，可以自定义UICollectionViewLayout类。UICollectionView对应的控制器是UICollectionViewController类。

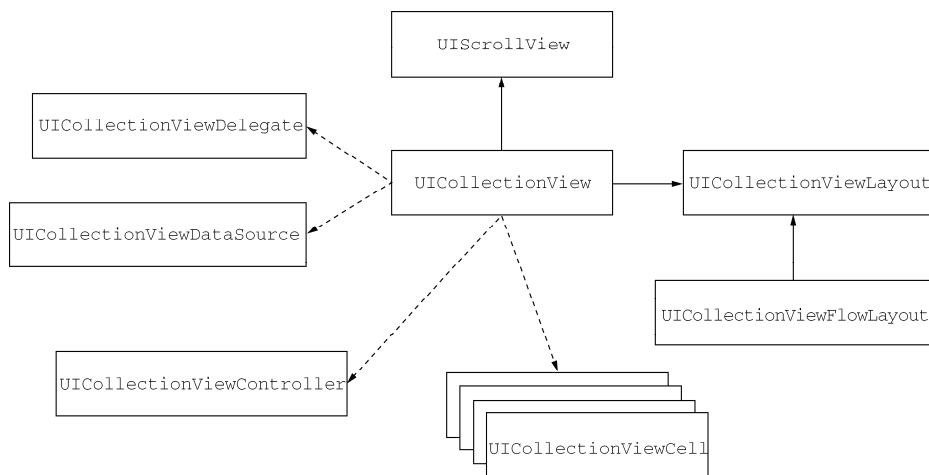


图4-94 集合视图类的构成

图4-95是使用集合视图展示奥运会比赛项目的案例，其中有8行2列，共16个比赛项目，点击其中一个，会有NSLog输出信息。该案例的具体实现过程如下所示。

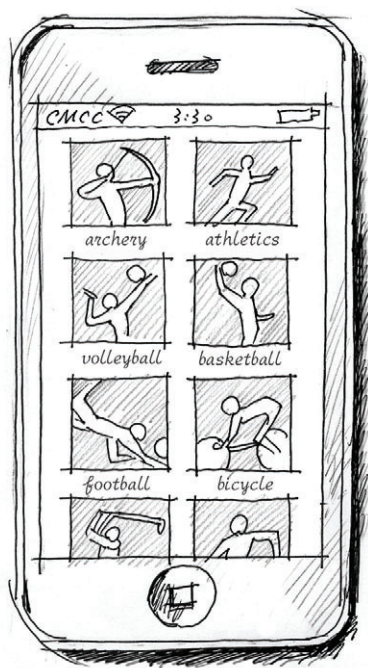


图4-95 案例原型草图

使用Single View Application模板创建一个名为CollectionViewSample的工程。打开ViewController.h，将ViewController父类UIViewController修改为UICollectionViewViewController，具体代码如下：

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

@interface ViewController : UICollectionViewViewController

@property (strong, nonatomic) NSArray * events;
@end
```

将资源文件夹Olympics_Pic添加到工程中作为一个组，如图4-96所示。它包含了比赛项目的图片和一个描述比赛项目信息的文件events.plist。

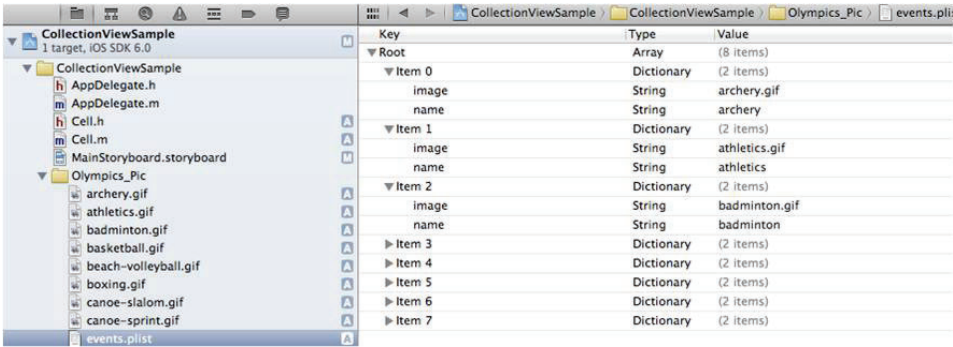


图4-96 添加Olympics_Pic组

events.plist文件的结构如图4-96所示，是在数组中放置字典，每个字典描述一个比赛项目信息，其中包括两个键值对：name和image。这个文件的读取是在viewDidLoad方法中实现的，相关代码如下：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    NSBundle *bundle = [NSBundle mainBundle];
    NSString *plistPath = [bundle pathForResource:@"events"
                                                ofType:@"plist"];
    //获取属性列表文件中的所有数据
    NSArray *array = [[NSArray alloc] initWithContentsOfFile:plistPath];
    self.events = array;
}
```

4.12.2 集合视图单元格

集合视图单元格是集合视图中最为重要的组成部分，没有样式和风格定义，它可以在故事板中设计，也可以通过程序代码来设定。单元格就是一个视图，可以在它的内部放置其他视图或控件。

首先，我们需要自定义一个单元格类，它继承自UICollectionViewCell，并且需要两个输出口属性UIImageView和UILabel，具体代码如下：

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

@interface Cell : UICollectionViewCell

@property (strong, nonatomic) IBOutlet UIImageView *imageView;
@property (strong, nonatomic) IBOutlet UILabel *label;

@end

#import "Cell.h"

@implementation Cell

- (id)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        //初始化代码
    }
    return self;
}

@end
```

在这个定义的单元格中，我们基本上不需要实现什么内容，这是因为这个案例是通过Interface Builder设计的。如果通过代码实现，就需要在initWithFrame:构造函数中实现UIImageView和UILabel这两个控件的初始化代码。

在Interface Builder中打开故事板设计界面，将View Controller Scene→View Controller→View删除，然后重新从对象库中拖曳一个Collection View视图到View Controller下面，如图4-97所示。

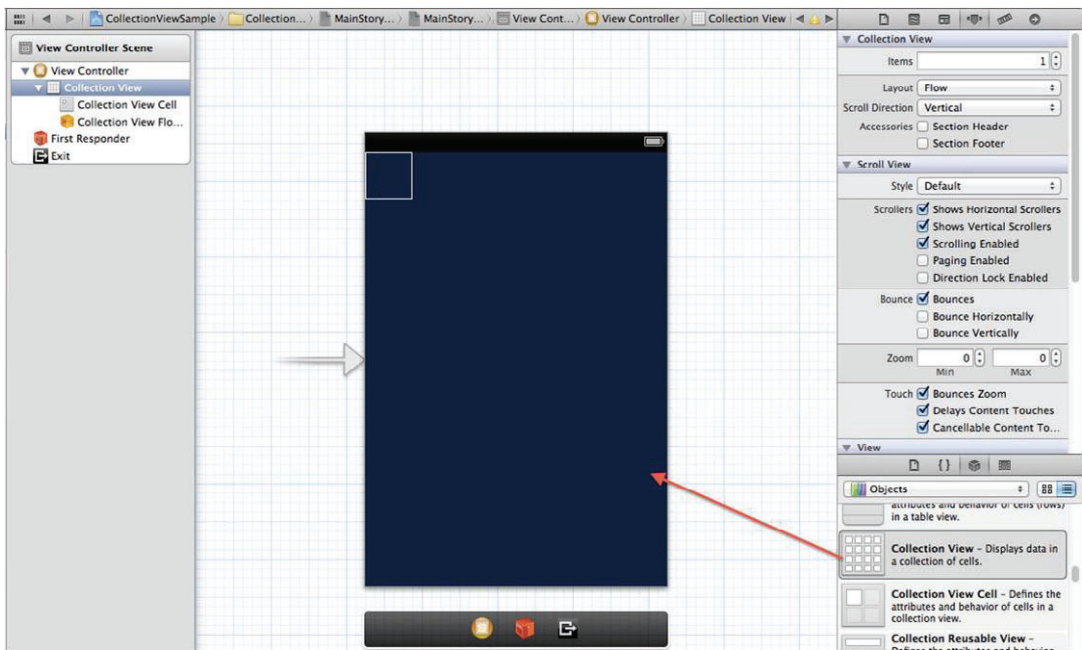



图4-97 替换视图

选择Collection View，将其背景改为白色，然后选中Collection View Cell，打开其标识检查器，将Custom Class→Class设为Cell；打开其属性检查器，在Collection Reusable View→Identifier中输入Cell，它是可重用单元格标识；打开其尺寸检查器，将Size修改为Custom，尺寸为150×150，如图4-98所示。

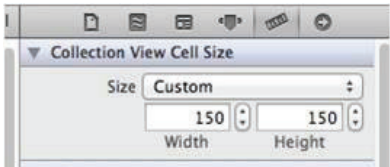


图4-98 尺寸检查器

提示 可重用单元格是为了节约内存开销而设计的，当屏幕在翻动时，旧的单元格退出屏幕，新的单元格进入屏幕，如果每次都实例化单元格，必然增加内存开销，可重用单元格就是不去实例化新的单元格，而是先使用可重用单元格标识到视图去找，找到就使用，没有则创建。iOS 6对可重用单元格进行优化，不再需要做判断了，代码如下所示：

```
Cell *cell = [collectionView dequeueReusableCellWithReuseIdentifier:@"Cell" forIndexPath:indexPath];
if (!cell) {
    // ...
}
```

回到Interface Builder界面，需要将UIImageView和UILabel拖曳到单元格中，如图4-99所示。

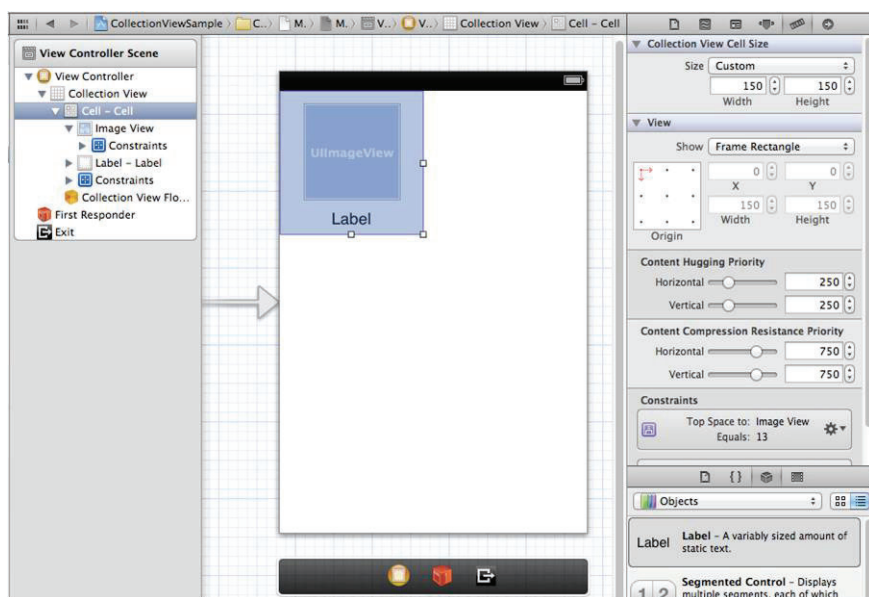


图4-99 拖曳控件到单元格

然后将单元格中的输出口控件与属性连线，具体操作方法是：首先选中单元格，然后按住control键将鼠标拖曳到UIImageView控件上，此时会弹出一个菜单，从中选择Image View，如图4-100所示。最后，使用同样的方法将Label控件与输出口属性label连接好。

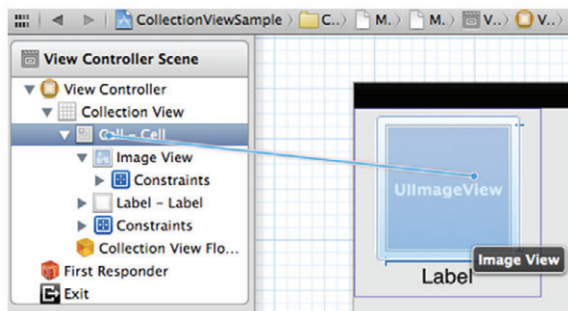


图4-100 连接输出口控件与属性

4.12.3 数据源协议与委托协议

与UIPickerView和UITableView类似，集合视图也有数据源协议与委托协议，它们的作用都是类似的。集合视图的委托协议是UICollectionViewDelegate，数据源协议是UICollectionViewDataSource。

UICollectionViewDataSource中的方法有如下4个。

- ❑ **collectionView:numberOfItemsInSection:**。提供某个节中的列数目。
- ❑ **numberOfSectionsInCollectionView:**。提供视图中节的个数。
- ❑ **collectionView:cellForItemAtIndexPath:**。为某个单元格提供显示数据。
- ❑ **collectionView:viewForSupplementaryElementOfKind:atIndexPath:**。为补充视图提供显示数据。

在ViewController.m中，UICollectionViewDataSource的实现代码如下：

```
- (NSInteger)numberOfSectionsInCollectionView:(UICollectionView *)collectionView
{
    return [self.events count] / 2;
}
- (NSInteger)collectionView:(UICollectionView *)collectionView
    numberOfItemsInSection:(NSInteger)section
{
    return 2;
}
- (UICollectionViewCell *)collectionView:(UICollectionView *)
    collectionView cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
    Cell *cell = [collectionView dequeueReusableCellWithReuseIdentifier:
        @"Cell" forIndexPath:indexPath];
    NSDictionary *event = [self.events objectAtIndex:(indexPath.section*2 + indexPath.row)];
    cell.label.text = [event objectForKey:@"name"];
    cell.imageView.image = [UIImage imageNamed:[event objectForKey:@"image"]];
    return cell;
}
```

UICollectionViewDelegate中的方法很多，这里我们选择几个较为重要的加以介绍。

❑ **collectionView:didSelectItemAtIndexPath:**。选择单元格之后触发。

❑ **collectionView:didDeselectItemAtIndexPath:**。不选择单元格之后触发。

在ViewController.m中，UICollectionViewDelegate的实现代码如下：

```
- (void)collectionView:(UICollectionView *)collectionView
    didSelectItemAtIndexPath:(NSIndexPath *)indexPath
{
    NSDictionary *event = [self.events objectAtIndex:(indexPath.section*2 + indexPath.row)];
    NSLog(@"select event name : %@", [event objectForKey:@"name"]);
}
```

运行上述代码，得到的输出结果如下：

```
select event name : basketball
select event name : athletics
select event name : archery
```

4.13 小结

本章首先向大家介绍了视图和控件之间的关系以及应用界面的构建层次，然后介绍了标签、按钮、文本框、文本视图、开关、滑块、分段控件、网页控件、屏幕滚动控件、活动指示器、进度条、警告、操作表、工具栏、导航栏等基本控件，接着介绍了屏幕布局的内容，然后我们向大家介绍了一个较为复杂的控件——选择器，最后讨论了iOS 6中集合视图方面的内容。

表视图是iOS开发中使用最频繁的视图。一般情况下，我们都会选择以表的形式来展现数据，比如通讯录和频道列表等。在表视图中，分节、分组和索引等功能使我们所展示的数据看起来更规整、更有条理。更令人兴奋的是，表视图还可以利用细节展示等功能多层次地展示数据。但与其他控件相比，表视图的使用相对比较复杂。

5.1 概述

在本节中，我们将了解表视图的一些概念、相关类、表视图的分类、单元格的组成和样式以及表视图的两个协议——UITableViewController委托和UITableViewDataSource数据源。

5.1.1 表视图的组成

在iOS中，表视图是最重要的视图，它有很多概念，这些概念之间的关系如图5-1所示。



图5-1 表视图组成图

下面我们简要介绍一下这些概念。

- ❑ **表头视图 (table header view)**。表视图最上边的视图，用于展示表视图的信息，例如表视图刷新信息，如图5-2所示。
- ❑ **表脚视图 (table footer view)**。表视图最下边的视图，用于展示表视图的信息，例如表视图分页时显示“更多”等信息，如图5-2所示。

- ❑ 单元格（cell）。它是组成表视图每一行的单位视图。
- ❑ 节（section）。它由多个单元格组成，有节头（section header）和节脚（section footer）。
- ❑ 节头。节的头，描述节的信息，如图5-3所示，文字左对齐。
- ❑ 节脚。节的脚，也可描述节的信息和声明，如图5-3所示，文字居中对齐。



图5-2 表头视图和表脚视图



图5-3 节头和节脚

5.1.2 表视图的相关类

表视图（UITableView）继承自UIScrollView，它有两个协议：UITableViewDelegate委托协议和 UITableViewDataSource数据源协议。此外，表视图还包含很多其他类，其中UITableViewCell类是单元格类，UITableViewController类是UITableView的控制器，UITableViewHeaderFooterView类用于为节头和节脚提供视图，它是iOS 6之后才有的新类，这些类的构成如图5-4所示。

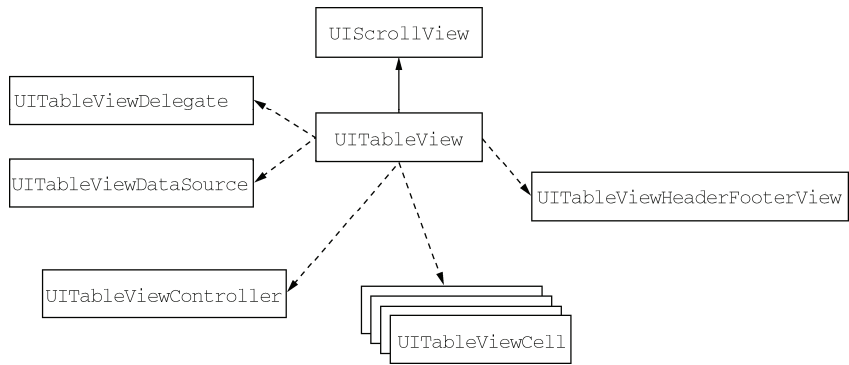


图5-4 UITableView类的结构图

图5-4所示的类只是表视图使用过程中涉及的几个主要类，其他的类和常量我们将在使用过程中逐一介绍。

5.1.3 表视图分类

iOS中的表视图主要分为普通表视图（如图5-5所示）和分组表视图（如图5-6所示），下面简要介绍一下这两种视图。

- ❑ 普通表视图。主要用于动态表，而动态表一般在单元格数目未知的情况下使用。
- ❑ 分组表视图。一般用于静态表，会将表分成很多“孤岛”，这个“孤岛”由一些类似的单元格组成。静态表一般用于控件的界面布局，它是在iOS 5之后由故事板提供的。

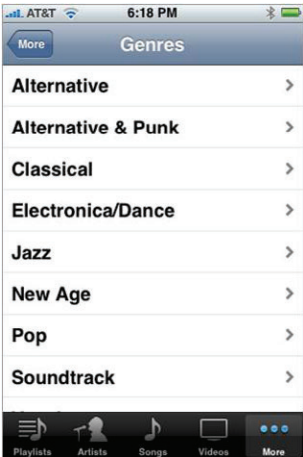


图5-5 普通表视图



图5-6 分组表视图

此外，在表视图中还可以带有索引列、选择列和搜索栏等，下面介绍一下具有这种特征的表视图情况。
图5-7所示的是索引表视图。一般情况下，在表视图超过一屏时应该添加索引列。图5-8所示的是选择表视图，用于给用户提供一个选择列表。由于iOS标准控件没有复选框控件，所以一般使用选择表视图来替代其他平台的控件。

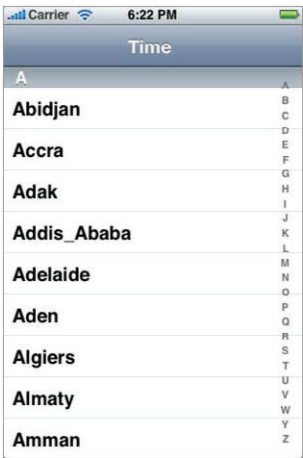


图5-7 索引表视图

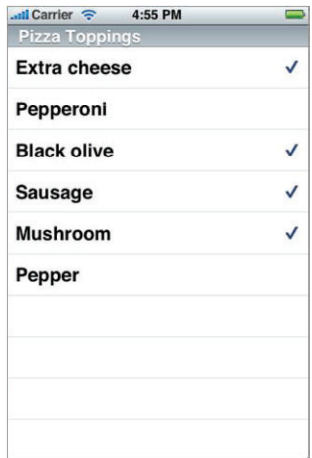


图5-8 选择表视图

图5-9所示的是带有搜索栏的表视图。由于单元格很多，所以我们需要借助搜索栏进行过滤。搜索栏一般放在表头，也就是说，只有表视图翻到最顶端时才会看到搜索栏。图5-10所示的是分页表视图。一般情况下，Twitter、

微博等需要网络请求的列表会使用分页表视图。分页表视图的表头中有刷新和加载等待标识，表脚中会有“更多”按钮或“加载更多”标识。对于此功能，iOS 6提供了分页刷新控件。

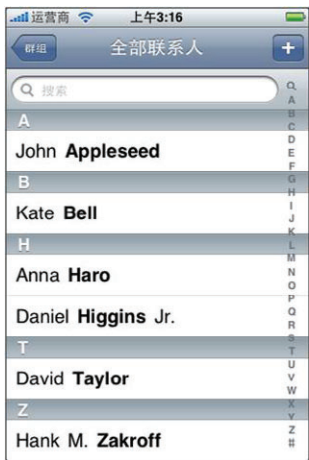


图5-9 搜索栏表视图



图5-10 分页表视图

表视图的分类不是绝对的。苹果提供了一些表视图的使用模式，使用时我们应首先考虑这些使用模式。当然，必要的话，我们还要根据业务需要进行合理的创新。

5.1.4 单元格的组成和样式

如图5-11所示，单元格由图标、标题和扩展视图等组成。

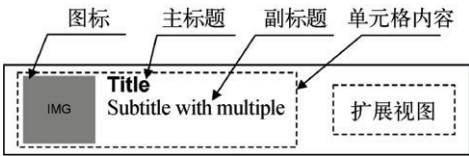


图5-11 单元格的组成

当然，单元格可以有很多样式，我们可以根据需要进行选择。图标、标题和副标题可以有选择地设置，扩展视图可以内置或者自定义，其中内置的扩展视图是在枚举类型UITableViewCellAccessoryType中定义的。枚举类型UITableViewCellAccessoryType中定义的常量如下所示。

- ❑ UITableViewCellAccessoryNone。没有扩展图标。
 - ❑ UITableViewCellAccessoryDisclosureIndicator。扩展指示器，触摸该图标➤将切换到下一级表视图。
 - ❑ UITableViewCellAccessoryDetailDisclosureButton。细节展示按钮，触摸该单元格的时候，表视图会以视图的方式显示当前单元格的更多详细信息，图标为⦿。
 - ❑ UITableViewCellAccessoryCheckmark。选中标志，表示该行被选中，图标为✔。
- 在开发过程中，我们应该首要考虑苹果公司提供的一些固有的单元格样式。iOS API提供的单元格样式是在枚举类型UITableViewCellStyle中定义的，而UITableViewCellStyle枚举类型中定义的常量如下所示。
- ❑ UITableViewCellStyleDefault。默认样式，如图5-12所示，只有图标和主标题。
 - ❑ UITableViewCellStyleSubtitle。带有副标题的样式，如图5-13所示，有图标、主标题和副标题。

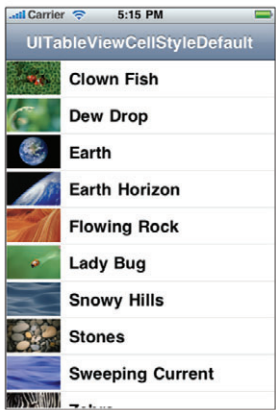


图5-12 默认样式

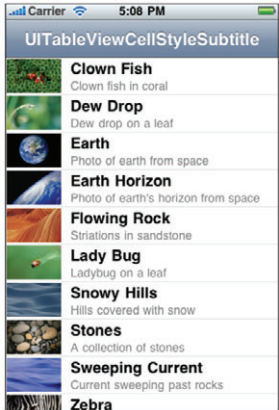


图5-13 带有副标题的样式

- ❑ **UITableViewCellStyleValue1**. 无图标带副标题样式1，如图5-14所示，有主标题和子标题。
- ❑ **UITableViewCellStyleValue2**. 无图标带副标题样式2，如图5-15所示，有主标题和子标题。

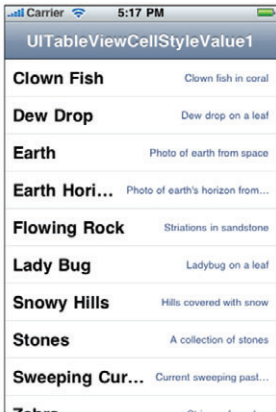


图5-14 无图标带副标题样式1

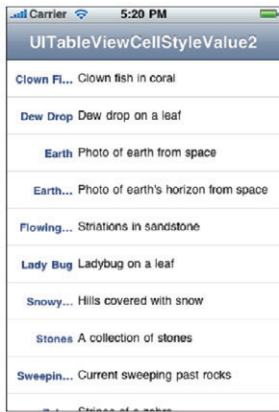


图5-15 无图标带副标题样式2

如果以上单元格样式都不能满足业务需求，可以考虑自定义单元格。

5.1.5 数据源协议与委托协议

与UIPickerView等复杂控件类似，表视图在开发过程中也会使用委托协议和数据源协议，而表视图UITableView的数据源协议是UITableViewDataSource，委托协议是UITableViewDelegate。UITableViewDataSource协议中的主要方法如表5-1所示，其中必须要实现的方法有tableView:numberOfRowsInSection:和tableView:cellForRowAtIndexPath:。

表5-1 UITableViewDataSource协议的主要方法

方 法	返回类型	说 明
tableView:cellForRowAtIndexPath:	UITableViewCell*	为表视图单元格提供数据，该方法是必须实现的方法
tableView:numberOfRowsInSection:	NSInteger	返回某个节中的行数

(续)

方 法	返回类型	说 明
tableView:titleForHeaderInSection:	NSString	返回节头的标题
tableView:titleForFooterInSection:	NSString	返回节脚的标题
numberOfSectionsInTableView:	NSInteger	返回节的个数
sectionIndexTitlesForTableView:	NSArray*	提供表视图节索引标题
tableView:commitEditingStyle:forRowAtIndexPath:	void	为删除或修改提供数据

UITableViewDelegate协议主要用来设定表视图中节头和节脚的标题，并响应一些动作事件，主要的方法见表5-2，它们都是可选择的。

表5-2 UITableViewDelegate协议的主要方法

方 法	返回类型	说 明
tableView:viewForHeaderInSection:	UIView *	为节头准备自定义视图，iOS 6之后可以使用UITableViewHeaderFooterView
tableView:viewForFooterInSection:	UIView *	为节脚准备自定义视图，iOS 6之后可以使用UITableViewHeaderFooterView
tableView:didEndDisplayingHeaderView:forSection:	void	该方法在节头从屏幕中消失时触发（iOS 6之后的方法）
tableView:didEndDisplayingFooterView:forSection:	void	当节脚从屏幕中消失时触发（iOS 6之后的方法）
tableView:didEndDisplayingCell:forRowAtIndexPath:	void	当单元格从屏幕中消失时触发（iOS 6之后的方法）
tableView:didSelectRowAtIndexPath:	void	响应选择表视图单元格时调用的方法

此外，相关的方法还有很多，随着学习的深入，我们会在一些案例和项目中进行进一步接触。

5.2 简单表视图

表视图的形式灵活多变，本着由浅入深的原则，我们先从简单表视图开始学习。

5.2.1 创建简单表视图

鉴于要创建的是一个最基本的表，我们只需实现UITableViewDataSource协议中必须要实现的方法tableView:numberOfRowsInSection:和tableView:cellForRowAtIndexPath:即可。简单表视图的时序图如5-16所示，其中构造方法initWithFrame:style:在实例化表视图时调用。

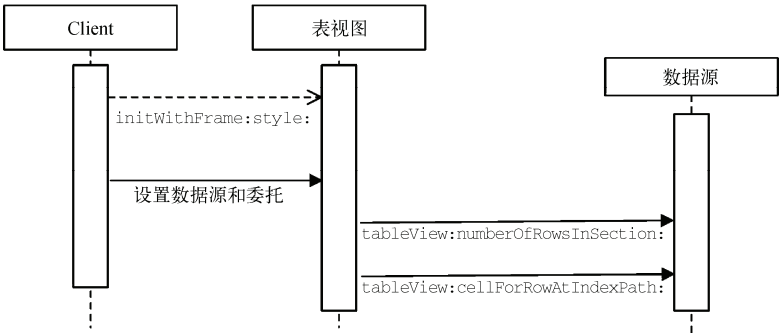


图5-16 简单表视图的时序图

如果采用xib或故事板来设计表视图,那么表视图的创建是在实例化表视图控制器的时候完成的,表视图显示的时候会发出tableView:numberOfRowsInSection:消息询问当前节中的行数,表视图单元格显示的时候会发出tableView:cellForRowAtIndexPath:消息为单元格提供显示数据。

下面我们创建一个类似图5-17所示的简单表视图,其中单元格使用默认样式,有图标和主标题,具体创建步骤如下所示。

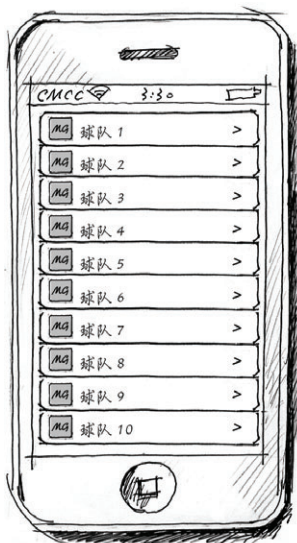


图5-17 简单表设计原型图

使用Single View Application模板创建一个工程,工程名为SimpleTable。打开Interface Builder设计界面,由于模板生成的视图控制器不是表视图控制器,因此需要在View Controller Scene中删除View Controller,然后再从控件库中拖曳一个Table View Controller到设计界面,如图5-18所示。

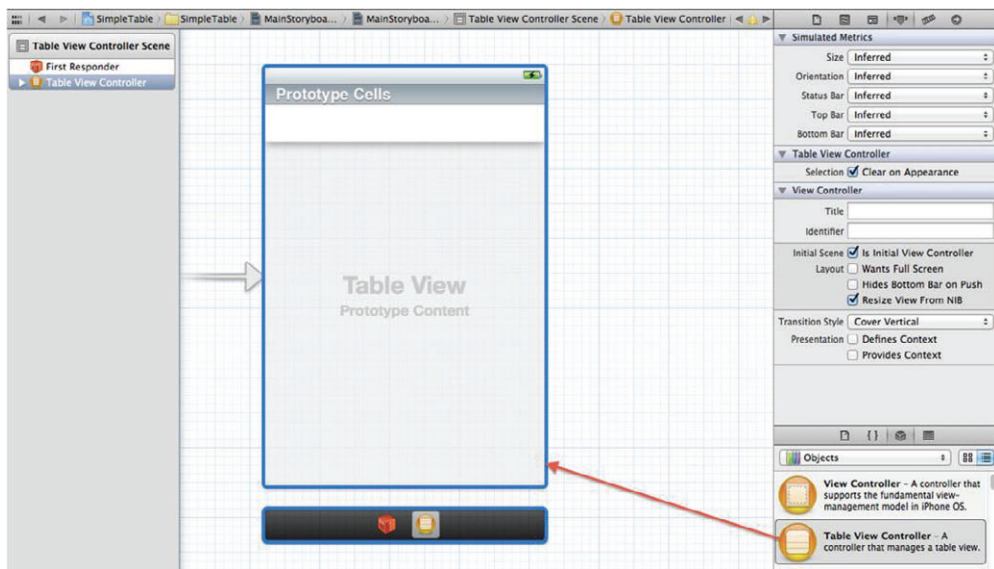




图5-18 拖曳 Table View Controller到设计界面

将.h文件中ViewController的父类从原来的UIViewController修改为UITableViewController。

在Interface Builder设计界面左侧的Scene列表中选择Table View Controller Scene → Table View Controller，打开表视图控制器的标识检查器，如图5-19所示，在Class下拉列表中选择ViewController，这是我们自己编写的视图控制器。

然后在Scene列表中选择Table View Controller Scene → Table View Controller → Table View，打开表视图的属性检查器，如图5-20所示。可以发现，Content下有两个选项——Dynamic Prototypes和Static Cells，这两个选项只有在故事板中才有。Dynamic Prototypes用于构建“动态表”，而Static Cells的相关内容我们会在“静态表”中详细介绍。

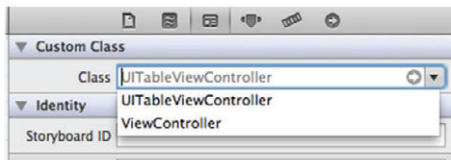


图5-19 表视图控制器的标识检查器

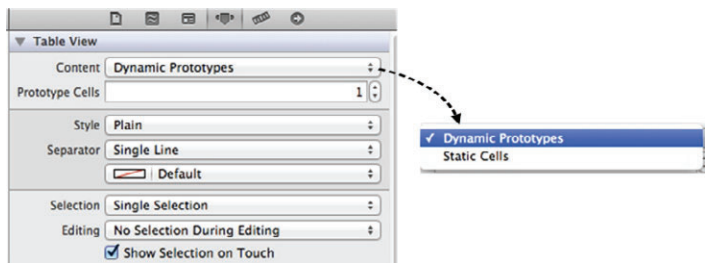


图5-20 表视图属性检查器

如果通过代码来实现单元格的创建，图5-20中的Prototype Cells项要设为0，相关的模式代码如下：

```
static NSString *CellIdentifier = @"CellIdentifier";
UITableViewCell *cell = [tableView
    dequeueReusableCellWithIdentifier:CellIdentifier];
if (cell == nil) {
    cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
        reuseIdentifier:CellIdentifier];
}
```

在上述代码中，CellIdentifier是可重用单元格的标识符，其中这个可重用单元格与集合视图中可重用单元格的概念一样。首先，在表视图中查找是否有可以重用的单元格，如果没有，就通过initWithStyle:reuseIdentifier:构造方法创建一个。

如果要利用故事板设计单元格，首先要选择Table View Controller Scene → Table View Controller → Table View → Table View Cell，打开单元格的属性检查器，如图5-21所示。可以看到，Style下有很多选项，这些选项与5.1节中描述的表视图单元格的样式一致，而Identifier是指可重用单元格的标识符。

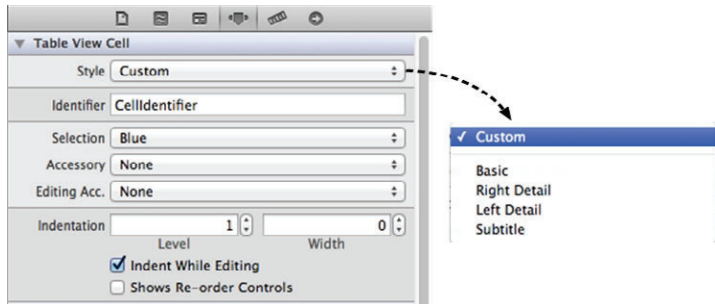


图5-21 表单元格属性检查器

这样操作以后，我们就不需要在代码中实例化单元格了。这里我们直接通过图5-21中设定的Identifier取得单元格的实例，以此达到重用单元格的目的。获得单元格对象的代码可以修改如下：

```
static NSString *CellIdentifier = @"CellIdentifier";
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
if (cell == nil) {
    cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
    reuseIdentifier:CellIdentifier];
}
}
```

此外，我们需要将team.plist和“球队图片”添加到工程中，ViewController.h文件的相关代码如下：

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

@interface ViewController : UITableViewController

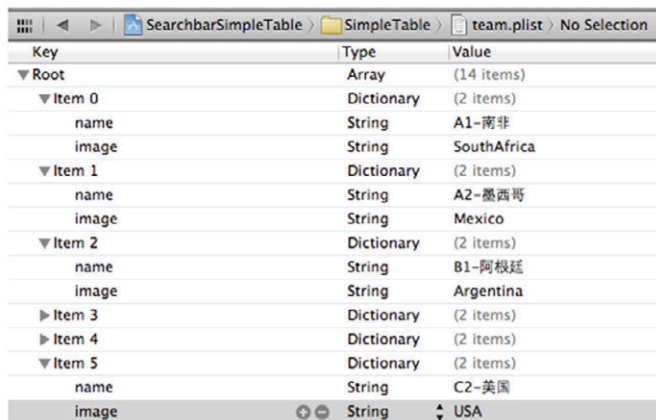
@property (nonatomic, strong) NSArray *listTeams;
@end
```

这里我们将ViewController的父类修改为UITableViewController。此外，还定义了NSArray*类型的属性listTeams，这个属性用来装载从文件中读取的数据。读取属性列表文件team.plist（其结构如图5-22所示）的操作是在viewDidLoad方法中实现的，相关代码如下：

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

    NSBundle *bundle = [NSBundle mainBundle];
    NSString *plistPath = [bundle pathForResource:@"team"
    ofType:@"plist"];

    // 获取属性列表文件中的全部数据
    self.listTeams = [[NSArray alloc] initWithContentsOfFile:plistPath];
}
```



Key	Type	Value
Root	Array	(14 items)
Item 0	Dictionary	(2 items)
name	String	A1-南非
image	String	SouthAfrica
Item 1	Dictionary	(2 items)
name	String	A2-墨西哥
image	String	Mexico
Item 2	Dictionary	(2 items)
name	String	B1-阿根廷
image	String	Argentina
Item 3	Dictionary	(2 items)
Item 4	Dictionary	(2 items)
Item 5	Dictionary	(2 items)
name	String	C2-英国
image	String	USA

图5-22 属性列表文件team.plist

我们再看看ViewController.m中实现UITableViewDataSource协议的方法，相关代码如下：

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection: (NSInteger)section
{
    return [self.listTeams count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath: (NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"CellIdentifier";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
```

```

        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];

        NSIndexPath row = [indexPath row];
        NSDictionary *rowDict = [self.listTeams objectAtIndex:row];
        cell.textLabel.text = [rowDict objectForKey:@"name"];

        NSString *imagePath = [rowDict objectForKey:@"image"];
        imagePath = [imagePath stringByAppendingString:@".png"];
        cell.imageView.image = [UIImage imageNamed:imagePath];

        cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;
        return cell;
    }

```

由于当前的这个表事实上只有一个节,因此不需要对节进行区分,在tableView:numberOfRowsInSection:方法中直接返回listTeams属性的长度即可。

tableView:cellForRowAtIndexPath:方法中NSIndexPath参数的row方法可以获得当前的单元格行索引。cell.accessoryType属性用于设置扩展视图类型。

运行之后的效果如图5-23所示。这里我们可以将单元格的样式UITableViewCellStyleDefault替换为其他3种来体验一下其他3种单元格样式的效果。

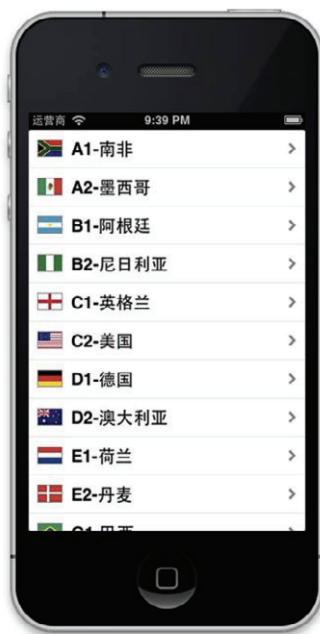


图5-23 简单表案例运行结果

5.2.2 自定义单元格

当苹果公司提供的单元格样式不能满足业务需求时,我们可以自定义单元格。在iOS 5之前,自定义单元格有两种实现方式:通过代码实现和用xib技术实现。用xib技术实现相对比较简单:创建一个.xib文件,然后再自定义一个继承UITableViewCell的单元格类即可。在iOS 5之后,我们又有了新的选择,用故事板实现,这种方式比xib方式更简单一些。

这里我们把5.2.1节所示案例的原型图修改一下,改后的原型如图5-24所示。

采用Single View Application工程模板创建一个名为CustomCell的工程,将Table View属性的Prototype Cells项设为1(除此之外,其他的操作过程与5.2.1节一样),如图5-25所示。

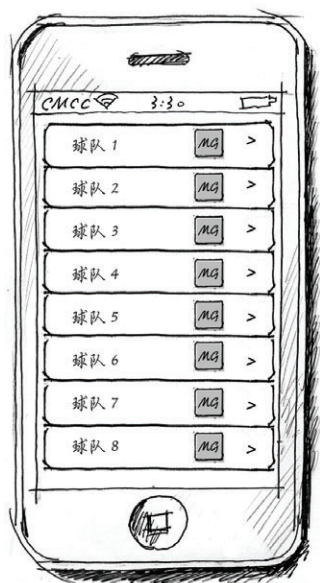


图5-24 自定义单元格设计原型图

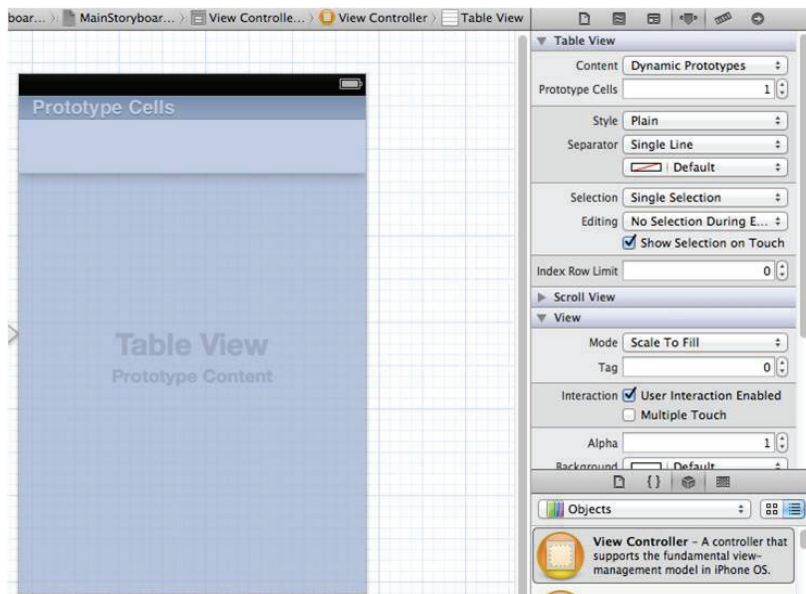


图5-25 设定Table View属性的Prototype Cells项

在设计界面中上部,一般会有一个单元格设计界面,我们可以在这个位置进行单元格布局的设计。从对象库中拖曳一个Label和Image View控件到单元格设计界面,如图5-26所示,调整好它们的位置。

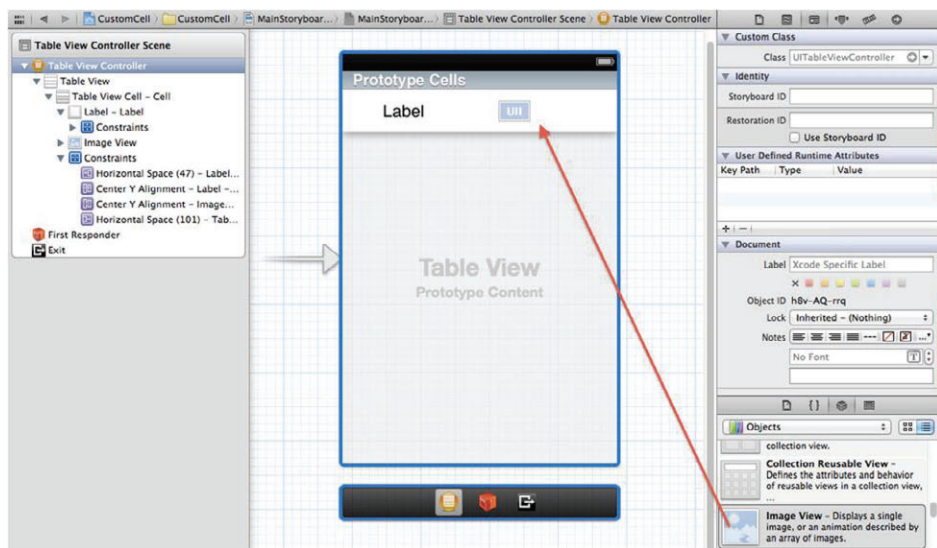


图5-26 设计表视图单元格

创建自定义单元格类CustomCell,具体操作方法为:右击工程名,在弹出的快捷菜单中选择Add File to CustomCell,此时界面中会弹出如图5-27所示的对话框,在Subclass of中选择UITableViewCell为其父类。

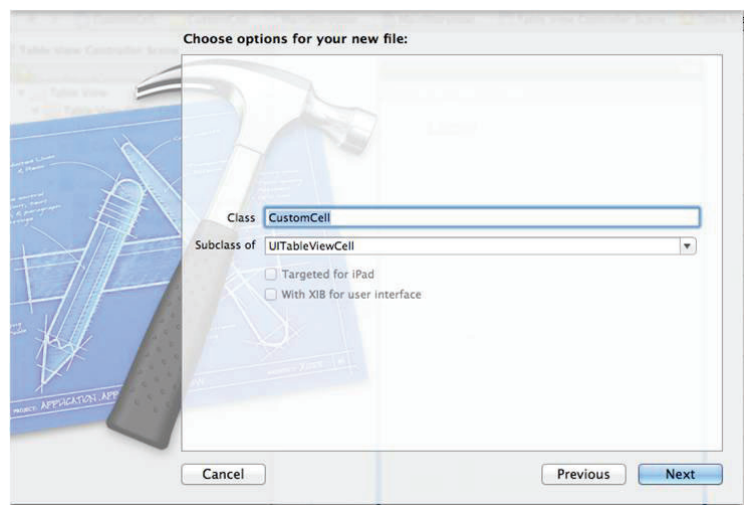



图5-27 创建自定义单元格类CustomCell

再回到Interface Builder设计界面，在左边选择Table View Controller Scene→Table View Controller→ Table View →Table View Cell，打开单元格的标识检查器，如图5-28所示，在Class下拉列表中选择CustomCell类。

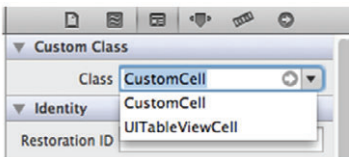


图5-28 选择CustomCell类

接着，为Label和ImageView控件连接输出口，如图5-29所示。

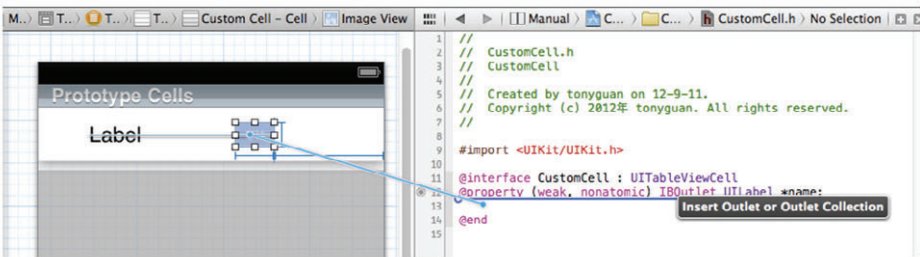


图5-29 输出口连线

本案例的代码如下：

```
//
//CustomCell.h
//CustomCell

#import <UIKit/UIKit.h>

@interface CustomCell : UITableViewCell
@property (weak, nonatomic) IBOutlet UILabel *name;
@property (weak, nonatomic) IBOutlet UIImageView *image;
```

```

@end

//
//CustomCell.m
//CustomCell

#import "CustomCell.h"

@implementation CustomCell
@end

```

其中CustomCell类的代码比较简单，在有些业务中还需要定义动作。

修改视图控制器ViewController.m中的tableView:cellForRowAtIndexPath:方法，相关代码如下：

```

- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    CustomCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier];
    }

    NSUInteger row = [indexPath row];
    NSDictionary *rowDict = [self.listTeams objectAtIndex:row];
    cell.name.text = [rowDict objectForKey:@"name"];

    NSString *imagePath = [rowDict objectForKey:@"image"];
    imagePath = [imagePath stringByAppendingString:@".png"];
    cell.image.image = [UIImage imageNamed:imagePath];

    cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;
    return cell;
}

```

我们看到代码if (cell == nil) {}被移除了，这是因为我们在Interface Builder中已经将重用标识设定为Cell了。其他代码与简单表一致，此处不再赘述。运行一下上述代码，得到的效果如图5-30所示。

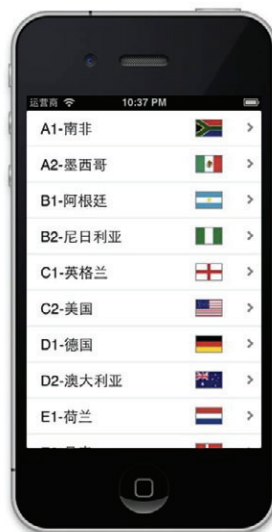


图5-30 自定义单元格案例的运行结果

5.2.3 添加搜索栏

当表视图中的数据量比较大的时候，要找到指定的数据并不是件轻而易举的事情，幸好iOS给我们提供了一个搜索栏控件（UISearchBar）。一般情况下，搜索栏置于表视图的表头，只有翻到顶部搜索栏才会出现。但是很多开发者会把搜索栏固定放置于屏幕之上，不随表视图的翻动而移动。将搜索栏一直放在屏幕上必然导致屏幕的部分空间一直被占用，而iPhone的屏幕本来就很小，这样设计不会获得太好的用户体验。

搜索栏有多种样式，如表5-3所示。

表5-3 搜索栏样式说明

样 式	说 明
	基本搜索栏。里面灰色的Search文字用于提示用户输入查询关键字，搜索栏的Placeholder属性可以设置这个提示信息
	带有清除按钮的搜索栏。在输入框中键入文字时，会在后面出现灰色清除按钮，点击清除按钮可以清除输入框中的文字
	带有查询结果按钮的搜索栏。显示最近搜索结果，显示设定如图5-31所示，选中Options下的Shows Search Results Button复选框，事件响应由UISearchBarDelegate对象中的searchBarResultsListButtonClicked:方法管理
	带有书签按钮的搜索栏。显示用户收藏的书签列表，显示设定如图5-31所示，选中Options中的Shows Bookmarks Button复选框，事件响应由UISearchBarDelegate对象中的searchBarBookmarkButtonClicked:方法管理
	带有取消按钮的搜索栏。显示设定如图5-31所示，选中Options下的Show Cancel Button复选框，事件响应由UISearchBarDelegate对象中的searchBarCancelButtonClicked:方法管理
	带有Scope的搜索栏。显示设定如图5-31所示，选中Options下的Shows Scope Bar复选框，同时需要设定下面的Scope Titles。选中这个选项时，搜索栏一出现就会在下面显示Scope Titles。如果初始化时不想显示，但在搜索栏获得焦点时显示，则可以在视图控制器的viewDidLoad方法中加入下面的代码： <pre>[self.searchBar setShowsScopeBar:NO];</pre> <pre>[self.searchBar sizeToFit];</pre> 其次，事件响应由UISearchBarDelegate或UISearchBarDisplayDelegate对象管理

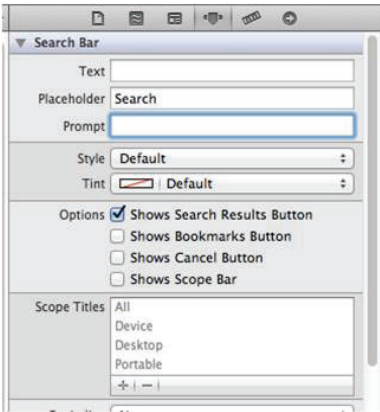


图5-31 搜索栏属性检查器

搜索栏是一个比较复杂的控件，它所涉及的类和协议如图5-32所示。

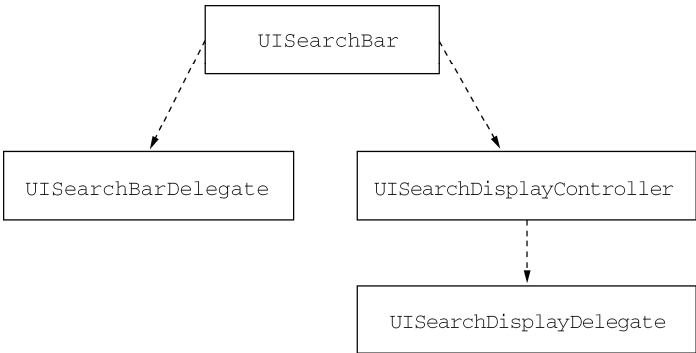


图5-32 搜索栏类结构图

UISearchBarDelegate是搜索栏控件的委托协议，UISearchDisplayController用来管理搜索栏并显示搜索结果视图。事件处理由UISearchDisplayDelegate协议的委托对象来管理。

下面我们通过一个名为SearchbarSimpleTable的工程来介绍一下如何在表视图中添加搜索栏，案例原型图见图5-33。我们在上一节简单表视图的基础上添加搜索栏，工程的创建过程参照上一节。有别于简单表视图的是，本案例中的单元格样式采用了有副标题的样式，其中副标题用于展示球队的英文名称，主标题是该球队的中文名称。在输入查询内容时，搜索栏下面会出现Scope Titles，它有按中文查询和按英文查询两种查询方式。

打开故事板设计界面，如图5-34所示，从对象库中拖曳一个Search Bar and Search Display Controller到设计界面，注意不是Search Bar控件。Search Bar and Search Display Controller的好处在于它可以把UISearchDisplayController也添加到搜索栏，并且将委托和数据源连线完毕。

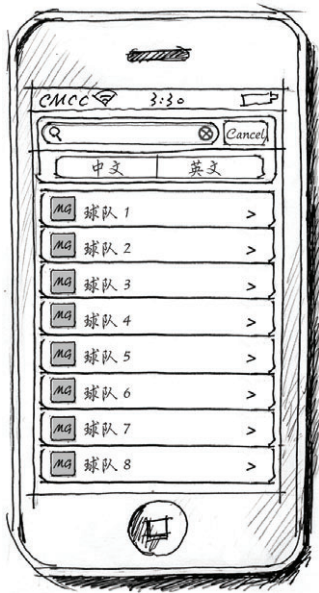


图5-33 添加搜索栏案例设计原型图

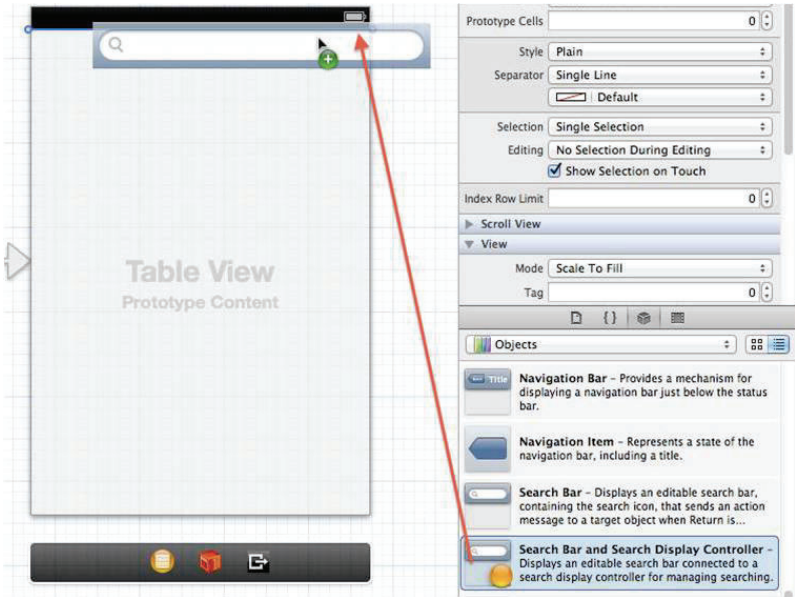


图5-34 拖曳搜索栏到设计界面

然后在设计界面中选择搜索栏，打开其属性检查器，然后将属性Placeholder设定为Search for Name，选中Shows Scope Bar复选框，并在Scope Titles中添加“中文”和“英文”，如图5-35所示。

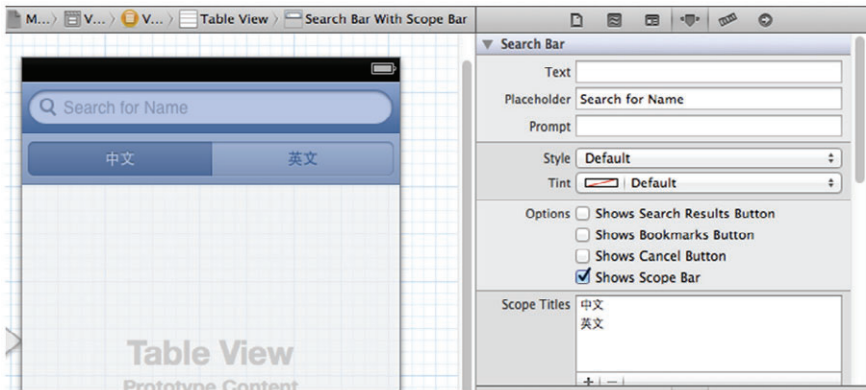


图5-35 搜索栏属性检查器

连接UISearchBar输出口，如图5-36所示。定义UISearchBar的属性，将其名称设置为searchBar。

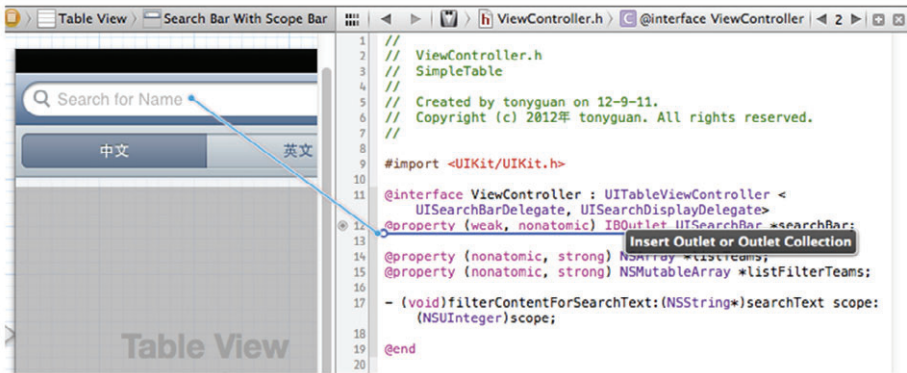


图5-36 连接UISearchBar输出口

下面我们看一下ViewController.h文件的代码：

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

@interface ViewController : UITableViewController
    <UISearchBarDelegate, UISearchDisplayDelegate>
@property (weak, nonatomic) IBOutlet UISearchBar *searchBar;

@property (nonatomic, strong) NSArray *listTeams;
@property (nonatomic, strong) NSMutableArray *listFilterTeams;

- (void)filterContentForSearchText:(NSString*)searchText scope:(NSInteger)scope;
@end
```

其中属性listTeams是为了装载全部球队的信息，listFilterTeams是查询之后的球队信息。

filterContentForSearchText:scope:方法用于查询过滤结果集，其中searchText是查询条件，scope是查询范围的索引。该方法在ViewController.m文件中的实现代码如下：

```
- (void)filterContentForSearchText:(NSString*)searchText scope:(NSInteger)scope;
{
    if([searchText length]==0)
    {
        // 查询所有
```

```

        self.listFilterTeams = [NSMutableArray arrayWithArray:self.listTeams];
        return;
    }

    NSPredicate *scopePredicate;
    NSArray *tempArray ;

    switch (scope) {
        case 0: //英文
            scopePredicate = [NSPredicate predicateWithFormat:@"SELF.name
                contains[c] %@",searchText]; ①
            tempArray =[self.listTeams filteredArrayUsingPredicate:scopePredicate]; ②
            self.listFilterTeams = [NSMutableArray arrayWithArray:tempArray];

            break;
        case 1:
            scopePredicate = [NSPredicate predicateWithFormat:
                @"SELF.image contains[c] %@",searchText]; ③
            tempArray =[self.listTeams filteredArrayUsingPredicate:scopePredicate];
            self.listFilterTeams = [NSMutableArray arrayWithArray:tempArray];

            break;
        default:
            //查询所有
            self.listFilterTeams = [NSMutableArray arrayWithArray:self.listTeams];
            break;
    }
}

```

第①行代码中的NSPredicate定义了一个逻辑查询条件，用来在内存中过滤集合对象；类方法predicateWithFormat:通过Predicate字符串创建了一个NSPredicate对象。本例中的@"SELF.name contains[c] %"是Predicate字符串，它有点像SQL语句或是HQL（Hibernate Query Language），其中SELF代表要查询的对象，SELF.name是查询对象的name字段。SELF对象可能是字段，此时name就是它的键。当SELF对象为实体类的时候，name就是它的属性。contains[c]是包含字符的意思，其中c是不区分大小写的。

提示 关于Predicate字符串的语法，可以参考<https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Predicates/Articles/pSyntax.html>。

在第②行代码中，filteredArrayUsingPredicate:方法是NSArray类中的方法，用来按照上面的条件进行过滤，结果返回的还是NSArray对象。我们需要重新构建一个NSMutableArray对象才可以将结果放到属性listFilterTeams中。

下面还有两个NSPredicate的例子：

```

NSMutableArray *array = [NSMutableArray arrayWithObjects:@"Bill",
    @"Ben", @"Chris", @"Melissa", nil];
NSPredicate *bPredicate = [NSPredicate predicateWithFormat:
    @"SELF beginswith[c] 'b'"];
NSArray *beginWithB = [array filteredArrayUsingPredicate:bPredicate];
//beginWithB 包含 { @"Bill", @"Ben" }

NSPredicate *sPredicate = [NSPredicate predicateWithFormat:
    @"SELF contains[c] 's'"];
[array filterUsingPredicate:sPredicate];
//数组包含 { @"Chris", @"Melissa" }

```

在搜索栏中输入查询条件，会触发UISearchBarDelegate委托对象的searchBar:textDidChange:方法和UISearchDisplayDelegate委托对象的searchDisplayController:shouldReloadTableForSearchString:方法，我们实现它们其中之一就可以达到搜索的基本目的。searchDisplayController:shouldReloadTableForSearchString:方法的优势在于可以控制表视图数据源是否重新加载。本例中实现了searchDisplayController:

shouldReloadTableForSearchString:方法, 它在ViewController.m文件中的实现代码如下:

```
- (BOOL)searchDisplayController:(UISearchDisplayController *)controller
    shouldReloadTableForSearchString:(NSString *)searchString
{
    [self filterContentForSearchText:searchString
        scope:self.searchBar.selectedScopeButtonIndex];
    //如果返回YES, 则表视图可以重新加载
    return YES;
}
```

如果采用UISearchBarDelegate委托对象的searchBar:textDidChange:方法实现搜索功能, 我们还需要自己调用[tableView reloadData]方法重新加载表视图数据。

类似地, 点击Scope Bar进行切换, 会触发UISearchBarDelegate委托对象的searchBar:selectedScopeButtonIndexDidChange:方法和UISearchDisplayDelegate委托对象的searchDisplayController:shouldReloadTableForSearchScope:方法, 我们也只需要实现二者之一即可。searchDisplayController:shouldReloadTableForSearchScope:的优势在于它可以控制表视图数据源是否重新加载。本例实现了searchDisplayController:shouldReloadTableForSearchScope:方法, 该方法在ViewController.m文件中的实现代码如下:

```
- (BOOL)searchDisplayController:(UISearchDisplayController *)controller
    shouldReloadTableForSearchScope:(NSInteger)searchOption
{
    [self filterContentForSearchText:self.searchBar.text scope:searchOption];
    //如果返回YES, 则表视图可以重新加载
    return YES;
}
```

当用户点击“取消”按钮时, 会触发UISearchBarDelegate委托对象的searchBarCancelButtonClicked:方法, 该方法在ViewController.m文件中的代码如下:

```
- (void)searchBarCancelButtonClicked:(UISearchBar *)searchBar
{
    //查询所有
    [self filterContentForSearchText:@" " scope:-1];
}
```

另外, 点击“取消”按钮还会查询所有的数据, 这可以通过[self filterContentForSearchText: @" " scope:-1]语句实现, 其中第二个参数scope是Scope Bar的索引。本例中有两个scope, 将它们的值分别设置为0、1, 当传递其他任何值时, 应用都是查询所有结果。我们看看视图加载方法viewDidLoad, 其代码如下:

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

    //设定搜索栏ScopeBar隐藏
    [self.searchBar setShowsScopeBar:NO];
    [self.searchBar sizeToFit];

    NSBundle *bundle = [NSBundle mainBundle];
    NSString *plistPath = [bundle pathForResource:@"team"
                                                ofType:@"plist"];

    //获取属性列表文件中的所有数据
    self.listTeams = [[NSArray alloc] initWithContentsOfFile:plistPath];

    //初次进入界面, 查询所有数据
    [self filterContentForSearchText:@" " scope:-1];
}
```

注意 如果我们是做iPhone开发, 搜索栏ScopeBar需要设置为隐藏。如果是进行iPad开发, 可以不进行此设置。

5.3 分节表视图

上一节中的简单表视图只有一个节，它实际上是分节表视图的一个特例。一个表可以有多个节，节也有头有脚，分节是添加索引和分组的前提。

在简单表视图的例子中，我们省略了如下代码：

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}
```

其中numberOfSectionsInTableView:方法的返回值是节的个数，一旦返回值大于1，其他很多方法都要相应地有所变化。另外，我们还可能会用到tableView:titleForHeaderInSection:和tableView:titleForFooterInSection:方法来设置节头和节脚的标题。

5.3.1 添加索引

当表视图中有大量数据集合时，除了添加搜索栏，我们还可以通过添加索引来辅助查询。

为一个表视图建立索引的规则与在数据库表中建立索引的规则是类似的，但也有一定差别。对于图5-37所示的表，索引列中的索引标题几乎与显示的标题完全一样，这种情况下我们还需要索引吗？该表的另一个问题就是索引列与扩展视图发生了冲突，当你点击索引列时，往往会点击到扩展视图的图标。索引列表的正确使用方式应该像英文字典的索引一样，A字母代表A开头的所有单词，如图5-38所示。

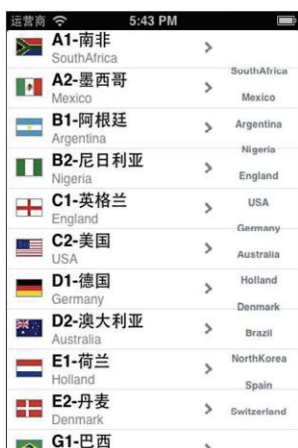


图5-37 错误使用索引



图5-38 正确使用索引

索引的正确使用原则如下所示。

- ❑ 索引标题不能与显示的标题完全一样。如果与要显示的标题一致，索引就变得毫无意义，如图5-37所示。
- ❑ 索引标题应具有代表性，能代表一个数据集合。如图5-38所示，索引标题A下有一系列符合要求的数据。
- ❑ 如果采用了索引列表视图，一般情况下就不再使用扩展视图。索引列表视图与扩展视图并存的时候，两者会存在冲突。当点击索引标题时，很容易点击到扩展视图。

接下来我们通过一个案例来演示正确使用索引的方式。

使用Single View Application模板创建一个名为SearchbarSimpleTable的工程。除了数据结构，其他操作与上一节完全相同。为了方便，我们将数据放到team_dictionary.plist文件中，该文件的数据结构与上一小节有所区别，具体如图5-39所示。

Key	Type	Value
▼ Root	Dictionary	(8 items)
▼ A组	Array	(4 items)
Item 0	String	A1-南非
Item 1	String	A2-墨西哥
Item 2	String	A3-乌拉圭
Item 3	String	A4-法国
▼ B组	Array	(4 items)
Item 0	String	B1-阿根廷
Item 1	String	B2-尼日利亚
Item 2	String	B3-韩国
Item 3	String	B4-希腊
► C组	Array	(4 items)
► D组	Array	(4 items)
► E组	Array	(4 items)
► F组	Array	(4 items)
► G组	Array	(4 items)
► H组	Array	(4 items)

图5-39 属性列表文件team_dictionary.plist

我们先看看ViewController.h的代码，其中dictData是从属性列表文件team_dictionary.plist中读取字典类型数据，listGroupname保存了小组名的集合，具体如下所示：

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

@interface ViewController : UITableViewController <UISearchBarDelegate>

//从team_dictionary.plist文件中读取出来的数据
@property (nonatomic, strong) NSDictionary *dictData;
//小组名集合
@property (nonatomic, strong) NSArray *listGroupname;

@end
```

读取属性列表文件team_dictionary.plist的实现代码位于ViewController.m中的viewDidLoad方法中，具体如下所示：

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

    NSBundle *bundle = [NSBundle mainBundle];
    NSString *plistPath = [bundle pathForResource:@"team_dictionary"
                                                ofType:@"plist"];

    //获取属性列表文件中的全部数据
    self.dictData = [[NSDictionary alloc] initWithContentsOfFile:plistPath];

    NSArray* tempList = [self.dictData allKeys];
    //对key进行排序
    self.listGroupname = [tempList sortedArrayUsingSelector:@selector(compare:)];
}
```

小组名的集合属性listGroupname是从dictData属性中取出的，它是dictData的键的集合。如果直接从字典中取出来，它的顺序是混乱状态（D组,C组,B组,H组,A组,G组,F组,E组），这是因为它是哈希结构，内部结构是无序的，我们需要使用[tempList sortedArrayUsingSelector: @selector(compare:)]语句对其重新进行排序。

此外，我们还需要修改数据源方法tableView:numberOfRowsInSection:和tableView:cellForRowAtIndexPath:，具体代码如下所示：

```

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection:
    (NSInteger)section
{
    //按照节索引从小组名数组中获得组名
    NSString *groupName = [self.listGroupname objectAtIndex:section];
    //将组名作为key, 从字典中取出球队数组集合
    NSArray *listTeams = [self.dictData objectForKey:groupName];
    return [listTeams count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"CellIdentifier";
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleSubtitle
            reuseIdentifier:CellIdentifier];
    }

    //获得选择的节
    NSUInteger section = [indexPath section];
    //获得选择节中选中的行索引
    NSUInteger row = [indexPath row];
    //按照节索引从小组名数组中获得组名
    NSString *groupName = [self.listGroupname objectAtIndex:section];
    //将组名作为key, 从字典中取出球队数组集合
    NSArray *listTeams = [self.dictData objectForKey:groupName];

    cell.textLabel.text = [listTeams objectAtIndex:row];

    return cell;
}

```

在表视图分节时, 需要实现数据源中numberOfSectionsInTableView:和tableView:titleForHeaderInSection:方法, 具体实现代码如下:

```

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return [self.listGroupname count];
}

- (NSString *)tableView:(UITableView *)tableView
    titleForHeaderInSection:(NSInteger)section
{
    NSString *groupName = [self.listGroupname objectAtIndex:section];
    return groupName;
}

```

上面这几个方法已实现了分节。分节只是添加索引的前提, 数据源的sectionIndexTitlesForTableView:方法才与索引直接相关。我们在该方法的listGroupname集合中存放的数据是A组, B组, C组, D组, E组, F组, G组, H组, 这些数据在索引列中显示的结果是A, B, C, D, E, F, G, H, 将后面的“组”截取掉:

```

- (NSArray *) sectionIndexTitlesForTableView: (UITableView *) tableView
{
    NSMutableArray *listTitles = [[NSMutableArray alloc]
        initWithCapacity:[self.listGroupname count]];
    //把“A组”改为“A”
    for (NSString *item in self.listGroupname) {
        NSString *title = [item substringToIndex:1];
        [listTitles addObject:title];
    }
    return listTitles;
}

```

此时再看看运行结果。

5.3.2 分组与静态表

在Interface Builder设计器中选择表视图，打开其属性检查器，从Style属性下拉列表中选择Grouped选项，如图5-40所示。

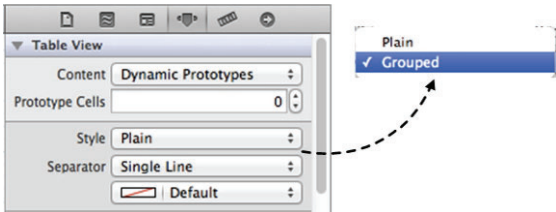


图5-40 表视图属性检查器

运行一下，得到的结果如图5-41所示，这个结果你是否满意呢？我们分析一下：我们分组的目的是让相关单元格放在“孤岛”上（这个功能我们已经实现），但是界面中“孤岛”的间距比较大，并不适合大量数据集的展示。需要说明的是，在数据量较小的情况下，没必要使用索引。

事实上，分组表视图的另外一个重要应用就是控制布局，这类似于我们在HTML网页中使用Table标签进行页面布局。图5-42是苹果官方的即时聊天工具iMessage应用的登录界面，如果这个界面没有采用表视图来控制布局，界面会非常难看。



图5-41 分组表视图的运行结果



图5-42 iMessage应用登录界面

可以看到，图5-42中的表视图是一个静态表，使用我们刚刚学到的知识就可以实现。我们需要将表视图分为三组，第一组有两个单元格，每一个单元格有一个文本框，文本框有输出口；第二组有一个单元格，其中放置一个登录按钮；第三组有一个单元格，其中包含标签控件和扩展指示器。这些工作基本上都是通过代码实现的，包括每一个控件的位置、动作事件等，这是一项比较繁重的工作。幸运的是，iOS 5之后的故事板技术可以帮助我们构建静态表。

下面我们吧图5-42的界面简化一下，采用静态表技术实现如图5-43所示的案例。

使用Single View Application模板创建一个名为StaticTableGroup的工程。打开Interface Builder设计界面，在View Controller Scene中删除View Controller，然后从控件库中拖曳一个Table View Controller到设计界面，接着选择View

Controller Scene→Table View，打开其属性检查器，如图5-44所示，从Content下拉列表中选择Static Cells（即静态表），将Sections的值设为3（即3节），从Style下拉列表中选择Grouped。



图5-43 登录界面

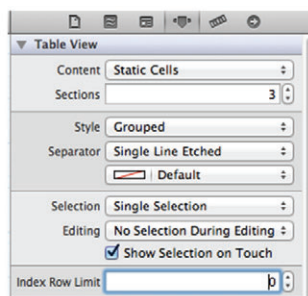


图5-44 静态表属性检查器

然后再选择View Controller Scene→Table View→Table View Section，选中第一节，打开它的属性检查器，如图5-45所示，将Rows的值设为2，即该节中包含两个单元格。我们还可以根据需要设定Header（节头）和Footer（节脚）。

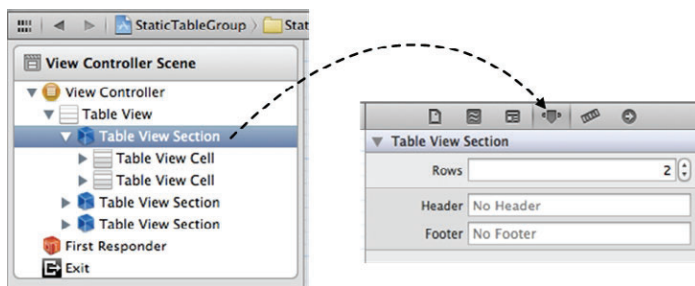


图5-45 静态表中的节属性检查器

将两个TextField控件分别拖曳到这个节中的单元格上，调整布局设定属性。在静态表第二节中，有一个按钮，可以按照上面的方法设定。第三节中的单元格中有标签控件和扩展指示器，其中扩展指示器的设定如图5-46所示。选择View Controller Scene→Table View→Table View Section，打开其属性检查器，从Accessory下拉列表中选择Disclosure Indicator（扩展指示器）。

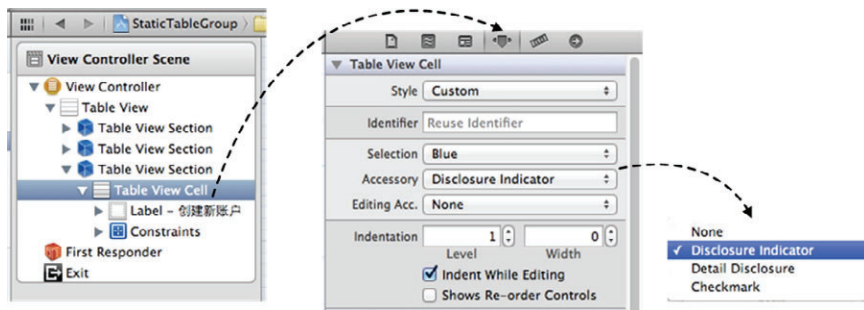


图5-46 为单元格选择扩展图标

这样整个界面就设计好了，如图5-47所示，你可以与图5-44的效果对比一下。要完成该案例还需要为登录按钮定义动作事件，为TextField定义输出口，这些操作与普通控件一致，这里不再赘述。



图5-47 设计完成的界面

相关的ViewController.h代码如下：

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

@interface ViewController : UITableViewController

@property (weak, nonatomic) IBOutlet UITextField *txtUserName;
@property (weak, nonatomic) IBOutlet UITextField *txtPwd;

- (IBAction)login:(id)sender;

@end
```

相关的ViewController.m代码如下：

```
@implementation ViewController

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

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

- (IBAction)login:(id)sender {

    if ([self.txtPwd.text isEqualToString:@"123"] &&
        [self.txtUserName.text isEqualToString:@"tony"]) {
        NSLog(@"登录成功。");
        [self.txtPwd resignFirstResponder];
        [self.txtUserName resignFirstResponder];
    }
}

@end
```

在上述代码中，login:方法用于响应登录按钮的点击事件，这里我们将登录验证规则“硬编码”了。不知道大家是否发现，上面的代码没有实现表视图数据源的tableView:numberOfRowsInSection:和tableView:cellForRowAtIndexPath:方法。是的，在静态表中可以不实现数据源和委托协议的方法。

5.4 修改单元格

对于表视图，我们不仅需要浏览数据，有时还需要修改其中的数据，本节简要介绍一下如何删除、插入和移动单元格等。

5.4.1 删除和插入单元格



表视图一旦进入删除和插入状态，单元格的左边就会出现一个“编辑控件”，如图5-48所示。这个区域会显示删除控件或插入控件，具体显示哪个图标在表视图委托协议的tableView:editingStyleForRowAtIndexPath:方法中设定。



图5-48 单元格编辑控件

为了防止用户操作失误，删除过程需要确认。删除控件时，删除控件从图5-49变成图5-50所示的样式，同时右侧会出现一个Delete按钮，点击该按钮数据才会成功删除。



图5-49 单元格删除控件

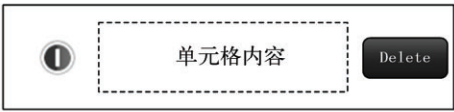


图5-50 单元格删除确认控件

提示 在iOS中，还有一个鲜为人知的删除手势，那就是在单元格中左右滑动手势，也会在单元格右边出现一个Delete按钮。

插入数据时，新插入的单元格会出现在表视图的最后，如图5-51所示。当点击插入控件时，会增加一行数据，此操作可重复进行。

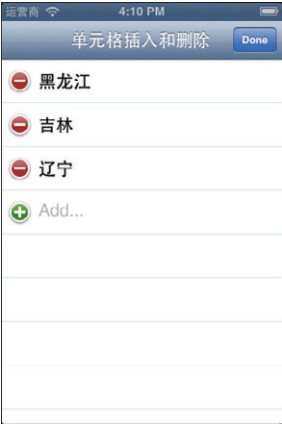


图5-51 单元格插入

删除和插入单元格操作的核心是如下两个方法：表视图委托对象的tableView:editingStyleForRowAtIndexPath:方法和表视图数据源对象的tableView:commitEditingStyle:forRowAtIndexPath:方法。删除和插入单元格的时序图如图5-52所示。

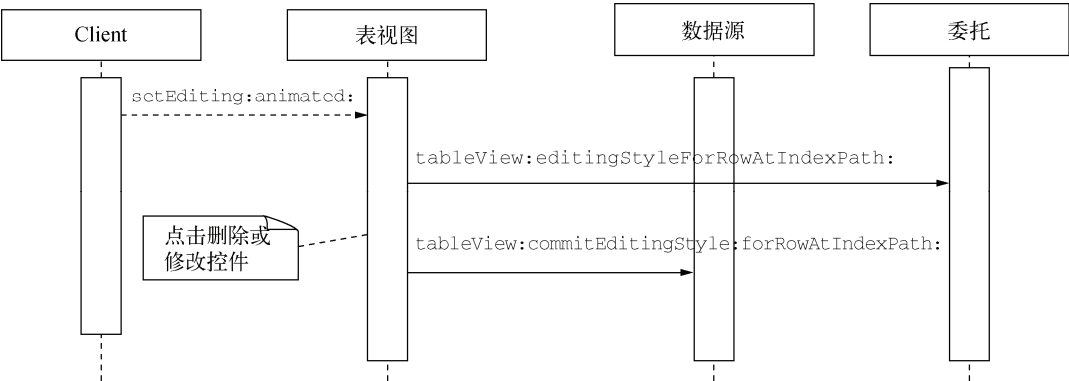


图5-52 删除和插入单元格的时序图

setEditing:animated:方法设定视图能否进入编辑状态，然后调用委托协议中的tableView:editingStyleForRowAtIndexPath:方法进行单元格编辑图标 的设置，当用户删除或修改控件时，委托方法向数据源发出tableView:commitEditingStyle:forRowAtIndexPath:消息实现删除或插入的处理。

下面我们实现图5-51所示的案例。使用Single View Application模板创建一个名为 DeleteAddCell的工程。打开Interface Builder设计界面，从控件库中拖曳一个Navigation Item控件和一个Table View控件（这两个控件同属于View的子视图）到View设计界面，如图5-53所示。注意，不要把Navigation Item拖曳到Table View里面，使其变成Table View的子视图。

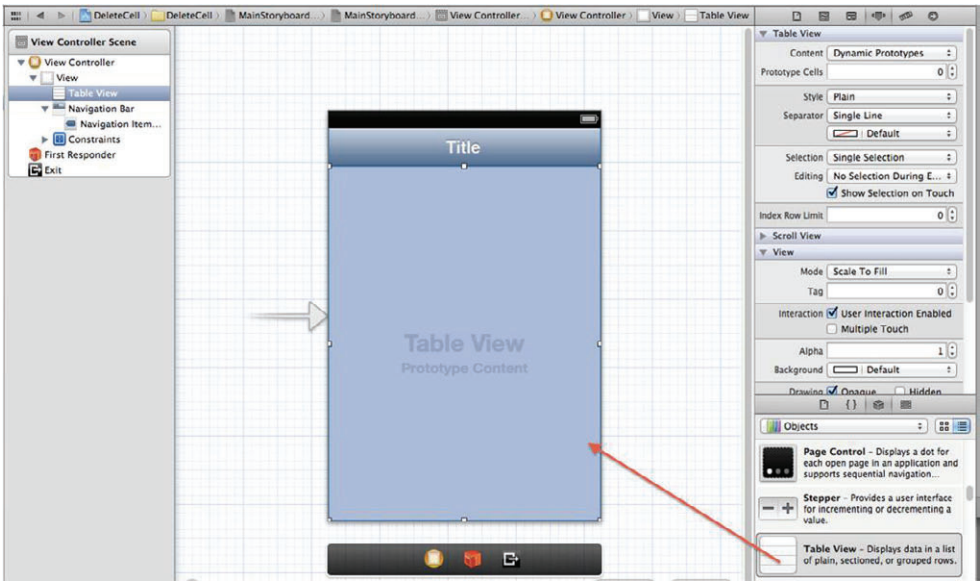


图5-53 拖曳Navigation Item和Table View到View设计界面

当插入单元格时，应该有一个控件能够接收用户输入的信息，这个控件应该是TextField文本输入框，所以我们在插入的单元格里放置了一个文本框。但是在Interface Builder中，把文本框放入到单元格中是比较困难的，我

们可以通过程序代码`cell.contentView addSubview: TextField`来实现。因此,我们可以先在View Controller Scene中添加一个TextField文本输入框,然后打开其属性检查器,将Font属性设置为System 20.0,将Placeholder属性设置为Add...,将Border Style属性设置为如图5-54所示的无边框样式。

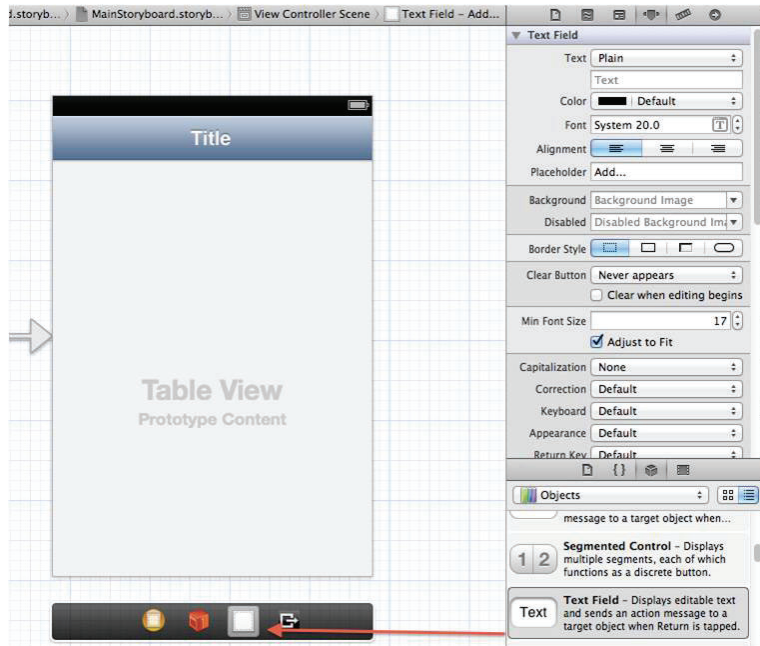


图5-54 Text Field属性检查器

将TextField文本输入框拖曳到View Controller Scene设计界面,其中是看不到设计界面的,这是因为它不是View或者Table View的子视图,还没有添加到任何视图中去,需要通过代码将其添加到单元格的内容视图(`contentView`)上。

然后我们需要为刚才拖曳的Navigation Item、Table View和Text Field控件定义输出口,并将其与视图控制器连线。连接过程这里就不介绍了,连接好之后的ViewController.h的代码如下:

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

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UINavigationController *navigationItem;
@property (weak, nonatomic) IBOutlet UITableView *tableView;
@property (strong, nonatomic) IBOutlet UITextField *txtField;

@end
```

在ViewController.h文件中添加表视图委托协议、数据源协议和文本框委托协议,相关代码如下:

```
@interface ViewController : UIViewController
<UITableViewDataSource, UITableViewDelegate, UITextFieldDelegate>
.....
@property (nonatomic, strong) NSMutableArray *listTeams;

@end
```

其中`listTeams`属性是可变数组集合,用于装载表视图的数据。这里将其声明为可变的,是为了可以对它进行删除或修改。下面我们看看ViewController.m中`viewDidLoad`方法的实现代码:

```

- (void)viewDidLoad
{
    [super viewDidLoad];



    //设置导航栏
    self.navigationItem.rightBarButtonItem = self.editButtonItem;
    self.navigationItem.title = @"单元格插入和删除";

    //设置单元格文本框
    self.txtField.hidden = YES;
    self.txtField.delegate = self;

    //将当前视图控制器分配给表视图的委托和数据源
    self.tableView.delegate = self;
    self.tableView.dataSource = self;

    self.listTeams = [[NSMutableArray alloc] initWithObjects:@"黑龙江",
        @"吉林", @"辽宁", nil];
}

```

在上述代码中，`self.navigationItem.rightBarButtonItem = self.editButtonItem`这行代码将编辑按钮设置为导航栏右边的按钮。编辑按钮是视图控制器中已经定义好的按钮，用`self.editButtonItem`可以取得编辑按钮的对象指针。编辑按钮的样式可以在  和  之间切换，如何切换取决于当前的视图是否处于编辑状态。点击编辑按钮时，会调用`setEditing:animated:`方法，其代码如下：

```

#pragma mark -- UIViewController生命周期方法，用于响应视图编辑状态变化
- (void)setEditing:(BOOL)editing animated:(BOOL)animated {
    [super setEditing:editing animated:animated];

    [self.tableView setEditing:editing animated:YES];
    if (editing) {
        self.txtField.hidden = NO;
    } else {
        self.txtField.hidden = YES;
    }
}

```

该方法是`UIViewController`生命周期的方法，用于响应视图编辑状态的变化。当表视图处于编辑状态时，文本框需要显示出来；当表视图处于非编辑状态时，我们应该将文本框隐藏。在`ViewController.m`中，还需要实现`UITableViewDataSource`协议中的`numberOfRowsInSection:`和`tableView:cellForRowAtIndexPath:`方法，它们的代码如下：

```

#pragma mark --UITableViewDataSource协议方法
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection:
    (NSInteger)section
{
    return [self.listTeams count] + 1;
}
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
    (NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    BOOL b_addCell = (indexPath.row == self.listTeams.count);

    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    }

    if (!b_addCell) {

```

```

        cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;
        cell.textLabel.text = [self.listTeams objectAtIndex:indexPath.row];
    } else {
        self.txtField.frame = CGRectMake(10,0,300,44);
        self.txtField.text = @"";
        [cell.contentView addSubview:self.txtField];
    }

    return cell;
}

```

numberOfRowsInSection:方法返回的不是listTeams集合的长度,而是“listTeams集合的长度+1”,这是因为我们需要为插入准备一个空的单元格,必须在此处预先指定。

在tableView:cellForRowAtIndexPath:方法中,要注意的是单元格要分两种情况来处理:一种是普通单元格,另一种是要插入的那个单元格,在插入的单元格中需要在其内容视图中添加文本框。

tableView:editingStyleForRowAtIndexPath:方法用于单元格编辑图标の設定,其代码如下:

```

#pragma mark --UITableViewDelegate协议方法
- (UITableViewCellEditingStyle)tableView:(UITableView *)tableView
    editingStyleForRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (indexPath.row == [self.listTeams count]) {
        return UITableViewCellEditingStyleInsert;
    } else {
        return UITableViewCellEditingStyleDelete;
    }
}

```

tableView:commitEditingStyle:forRowAtIndexPath:方法用于实现删除或插入处理,其代码如下:

```

#pragma mark --UITableViewDataSource协议方法
- (void)tableView:(UITableView *)tableView commitEditingStyle:
    (UITableViewCellEditingStyle)editingStyle
    forRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (editingStyle == UITableViewCellEditingStyleDelete)
    {
        [self.listTeams removeObjectAtIndex:indexPath.row];
        [self.tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
            withRowAnimation:UITableViewRowAnimationFade];
        [self.tableView reloadData];
    } else if (editingStyle == UITableViewCellEditingStyleInsert) {
        [self.listTeams insertObject:self.txtField.text atIndex:[self.listTeams count]];

        [self.tableView insertRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
            withRowAnimation:UITableViewRowAnimationFade];

        [self.tableView reloadData];
    }
}

```

在删除单元格数据时(即if (editingStyle == UITableViewCellEditingStyleDelete){}分支),本例中删除的是内存对象listTeams集合中的数据,但是如果数据来源于数据库,则应该删除的是数据库里的数据。下面是删除表视图单元格的方法,其中withRowAnimation:参数可以设置删除时的动画效果:

```

[self.tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
    withRowAnimation:UITableViewRowAnimationFade];

```

删除单元格后,要通过self.tableView reloadData语句重新加载表视图数据。插入单元格的情况与此类似,也需要重新加载数据。

上述代码足以完成单元格的删除和插入, 为了更加友好, 我们还添加了一些方法: UITableViewDelegate 协议中的 tableView:shouldHighlightRowAtIndexPath: 和 tableView:heightForRowAtIndexPath: 方法, 其代码如下:

```
- (BOOL)tableView:(UITableView *)tableView shouldHighlightRowAtIndexPath:
(NSIndexPath *)indexPath
{
    if (indexPath.row == [self.listTeams count]) {
        return NO;
    } else {
        return YES;
    }
}
- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:
(NSIndexPath *)indexPath
{
    return 50;
}
```

其中tableView:shouldHighlightRowAtIndexPath:设定单元格是否能在选择时处于高亮状态。一般情况下, 我们不希望用户能够选择表视图的最后一个单元格, 因为它没有内容, 如图5-55所示。如果tableView:shouldHighlightRowAtIndexPath:方法返回NO, 就能够选择最后一个单元格, 但这样用户就感觉不到它的存在了。



图5-55 单元格选择

最后, 还有一些键盘的问题需要处理。下面的两个方法是与键盘有关的, 一个是放弃第一响应者以关闭键盘, 另外一个避免键盘遮挡文本框:

```
#pragma mark -- UITextFieldDelegate委托方法, 关闭键盘
- (BOOL)textFieldShouldReturn:(UITextField *)textField
{
    [textField resignFirstResponder];
    return YES;
}

#pragma mark -- UITextFieldDelegate委托方法, 避免键盘遮挡文本框
- (void) textFieldDidBeginEditing:(UITextField *)textField {
    UITableViewCell *cell = (UITableViewCell*) [[textField superview] superview];
    [self.tableView setContentOffset:CGPointMake(0.0,
        cell.frame.origin.y) animated:YES];
}
```

在解决键盘遮挡的方法中, UITableViewCell *cell = (UITableViewCell*) [[textField superview] superview]; 这行代码返回 textField 所在的单元格。需要注意的是, textField 的父视图不是 UITableViewCell, 而 textField 的父视图的父视图才是 UITableViewCell, 大家可以参考图5-56。

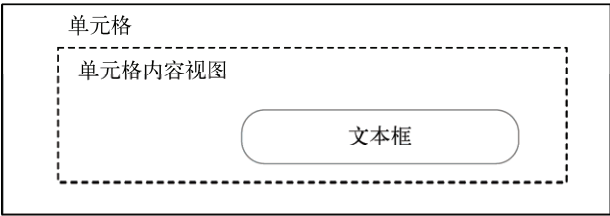


图5-56 单元格选择

5.4.2 移动单元格

在表视图中，单元格的顺序可以重新排列，书中将其称为移动单元格。移动单元格与插入和删除单元格类似，在单元格的后面会有重排序控件，如图5-57所示。



图5-57 重排序控件

图5-58是处于编辑状态的单元格，图5-59是处于移动状态的单元格。



图5-58 处于编辑状态的单元格



图5-59 处于移动状态的单元格

移动单元格时，需要实现数据源的tableView:canMoveRowAtIndexPath:和tableView:moveRowAtIndexPath:toIndexPath:方法，其时序图如图5-60所示。

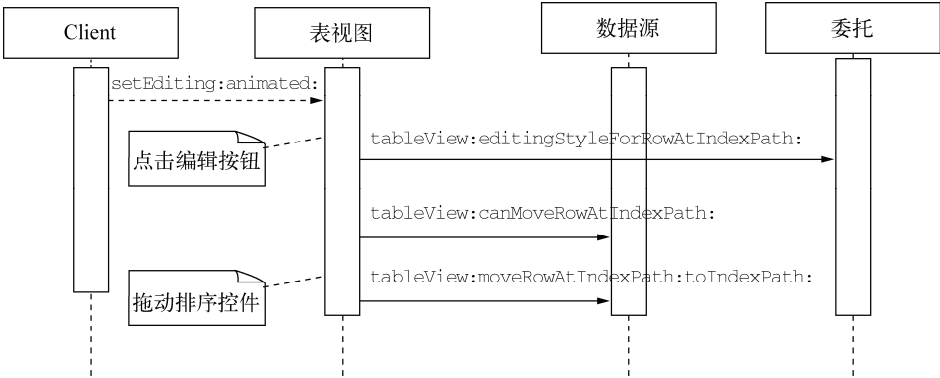


图5-60 移动单元格时序图

时序图反映的是当用户点击编辑按钮时，系统通过`setEditing:animated:`方法设定视图能否进入编辑状态，这与删除和插入单元格的操作一致，代码如下：

```
#pragma mark -- UIViewController生命周期方法，用于响应视图编辑状态变化
- (void)setEditing:(BOOL)editing animated:(BOOL)animated {

    [super setEditing:editing animated:animated];

    [self.tableView setEditing:editing animated:YES];

}
```

然后调用委托方法`tableView:editingStyleForRowAtIndexPath:`进行单元格编辑图标の設定，这个方法不是必需的。如果不设定它，默认情况下删除和重排序控件同时存在，如图5-61所示。本例中设定的是`UITableViewCellEditingStyleNone`，它是在枚举类型`UITableViewCellEditingStyle`中定义的。枚举类型`UITableViewCellEditingStyle`中的常量有：

- ☐ `UITableViewCellEditingStyleNone`
- ☐ `UITableViewCellEditingStyleDelete`
- ☐ `UITableViewCellEditingStyleInsert`



图5-61 单元格有删除和重排序控件

`tableView:editingStyleForRowAtIndexPath:`方法的代码如下：

```
- (UITableViewCellEditingStyle)tableView:(UITableView *)tableView
    editingStyleForRowAtIndexPath:(NSIndexPath *)indexPath
{
    return UITableViewCellEditingStyleNone;
}
```

当`tableView:canMoveRowAtIndexPath:`方法返回YES时，表示可以移动单元格，返回NO时，表示不能移动单元格，其代码如下：

```
- (BOOL)tableView:(UITableView *)tableView canMoveRowAtIndexPath:
    (NSIndexPath *)indexPath {
    return YES;
}
```

当用户拖动排序控件时，会触发`tableView:moveRowAtIndexPath:toIndexPath:`方法，该方法会对`listTeams`数据进行重新排序：

```
- (void)tableView:(UITableView *)tableView moveRowAtIndexPath:(NSIndexPath*)
    sourceIndexPath toIndexPath:(NSIndexPath *)destinationIndexPath
{
    NSString *stringToMove = [self.listTeams objectAtIndex:sourceIndexPath.row];
    [self.listTeams removeObjectAtIndex:sourceIndexPath.row];
    [self.listTeams insertObject:stringToMove atIndex:destinationIndexPath.row];
}
```

5.5 表视图 UI 设计模式

在iOS中，表视图应用极其广泛，本节将向大家介绍表视图中的两个UI设计模式——分页模式和下拉刷新（Pull-to-Refresh）模式，这两种模式已经成为移动平台开发的标准。

5.5.1 分页模式

想一想，你的新浪微博是否可以一次将所有的微博信息返回到你的设备屏幕里？当数据量很大时，一次返回所有信息这种方式会严重影响应用的性能，造成网络堵塞。通常，我们利用分页模式来解决请求大量数据的问题。

分页模式是先请求少量数据，例如一次50条，当翻动屏幕已显示50条数据之后，应用会再次请求50条。

根据触发方式的不同，请求分为主动请求和被动请求。图5-62所示为主动请求模式，即当条件满足时，再次请求下50条数据是自动发出的，并且一般在表视图的表脚会出现活动指示器，请求结束后活动指示器会隐藏起来。图5-63所示为被动请求模式，当条件满足时，表视图的表脚中会显示出一个响应点击事件的控件。这个控件一般是一个按钮，按钮标签一般会设为“更多”。当点击“更多”按钮时，应用会向服务器请求，请求结束后“更多”按钮会隐藏起来。

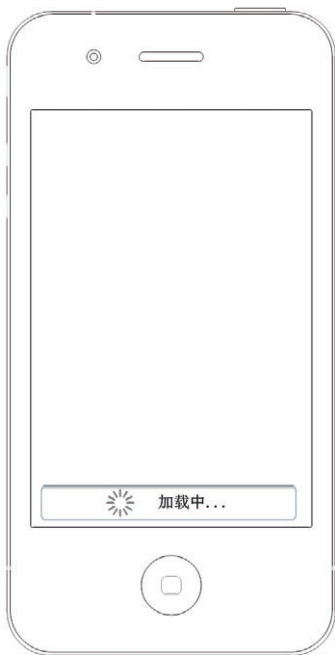


图5-62 主动请求

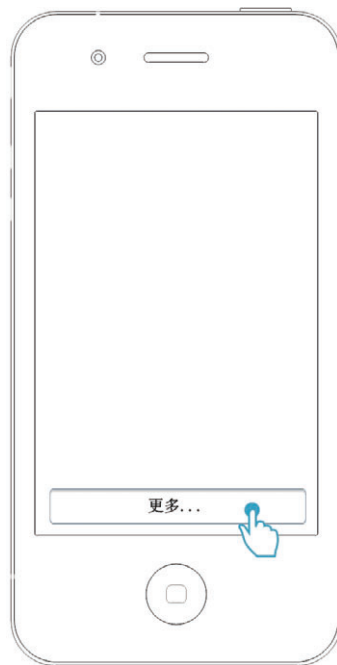


图5-63 被动请求

5.5.2 下拉刷新模式

下拉刷新是重新刷新表视图或列表，以便重新加载数据，这种模式广泛用于移动平台。下拉刷新与分页相反，当翻动屏幕到顶部时，再往下拉屏幕，程序就开始重新请求数据，此时表视图的表头部分会出现活动指示器，请求结束后表视图表头消失，如图5-64所示。可以看到，下拉刷新模式带有箭头动画效果。

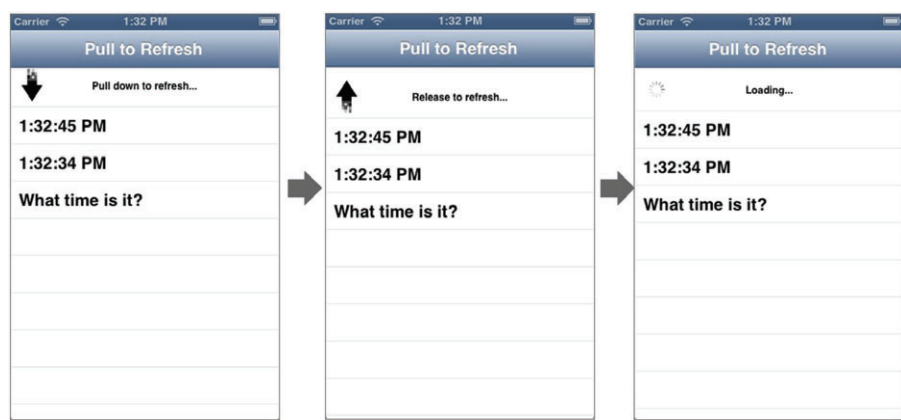


图5-64 下拉刷新模式

在很多开源社区中，都有下拉刷新的实现代码供大家参考，比如Github上的git：<https://github.com/leah/PullToRefresh.git>。

5.5.3 iOS 6 下拉刷新控件

随着下拉刷新模式的影响力越来越大，苹果不得不考虑把它列入自己的规范之中，并在iOS 6 API中推出了下拉刷新控件。图5-65所示的是iOS 6中的下拉刷新，有点像是在拉“胶皮糖”，当这个“胶皮糖”拉断时，就会出现活动指示器。

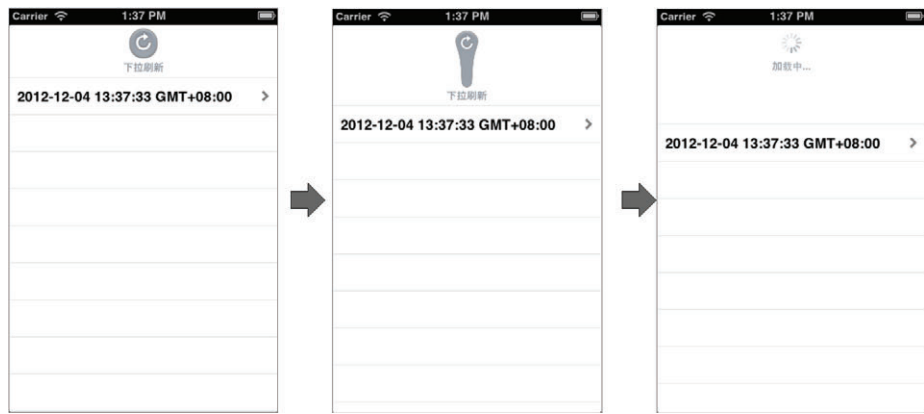


图5-65 iOS 6的下拉刷新

在iOS 6之后，UITableViewController添加了一个refreshControl属性，这个属性保持了UIRefreshControl的一个对象指针。UIRefreshControl就是iOS 6为表视图实现下拉刷新而提供的类，目前该类只能应用于表视图界面。UIRefreshControl的refreshControl属性与UITableViewController配合使用，关于下拉刷新布局等问题可以不必考虑，UITableViewController会将其自动放置于表视图中。

下面我们通过一个例子来了解一下UIRefreshControl控件的用法。参考创建简单表视图的案例创建工程RefreshControlSample，然后修改代码ViewController.h：

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

@interface ViewController : UITableViewController
```

```
@property (nonatomic, strong) NSMutableArray* Logs;

@end
```

其中Logs属性存放了NSDate日期列表，用于在表视图中显示需要的数据。

ViewController.m中的初始化代码如下：

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

    //初始化变量和时间
    self.Logs = [[NSMutableArray alloc] init];
    NSDate *date = [[NSDate alloc] init];
    [self.Logs addObject:date];

    //初始化UIRefreshControl
    UIRefreshControl *rc = [[UIRefreshControl alloc] init];
    rc.attributedTitle = [[NSAttributedString alloc] initWithString:@"下拉刷新"];
    [rc addTarget:self action:@selector(refreshTableView)
     forControlEvents:UIControlEventValueChanged];
    self.refreshControl = rc;
}
}
```

在上述代码中，我们初始化了当前时间的一条模拟数据。UIRefreshControl的构造方式是init。attributedTitle属性用于显示下拉控件的标题。UIRefreshControl的addTarget:forControlEvents:方法能够通过编程方式为UIControlEventValueChanged的事件添加处理方法。refreshTableView是UIControlEventValueChanged事件的处理方法，相关代码如下：

```
- (void) refreshTableView
{
    if (self.refreshControl.refreshing) {
        self.refreshControl.attributedTitle = [[NSAttributedString
        alloc] initWithString:@"加载中..."];
        //添加新的模拟数据
        NSDate *date = [[NSDate alloc] init];
        //模拟请求完成之后，回调方法callBackMethod
        [self performSelector:@selector(callBackMethod:) withObject:
        date afterDelay:3];
    }
}
```

UIRefreshControl的refreshing属性可以判断控件是否处于刷新状态，刷新状态的图标是我们常见的活动指示器，在这个阶段要将显示标题设置为“加载中...”。接下来，应该进行网络请求或者数据库查询的操作。然后，应用会回调callBackMethod方法。本案例涉及云端的技术，我们使用self performSelector:@selector(callBackMethod:) withObject:date afterDelay:3语句延时调用callBackMethod方法来模拟实现。

回调方法callBackMethod:的代码如下：

```
- (void)callBackMethod:(id) obj
{
    [self.refreshControl endRefreshing];
    self.refreshControl.attributedTitle = [[NSAttributedString alloc]
    initWithString:@"下拉刷新"];
    [self.Logs addObject:(NSDate*)obj];

    [self.tableView reloadData];
}
```

在请求完成的时候，endRefreshing方法可以停止下拉刷新控件，回到初始状态，显示的标题文本为“下拉刷新”。[self.tableView reloadData]语句用于重新加载表视图。

实现UITableViewDataSource的方法的代码如下：

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection:
(NSInteger)section {
    return [self.Logs count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleValue1
            reuseIdentifier:CellIdentifier];
    }

    NSDateFormatter *dateFormat = [[NSDateFormatter alloc] init];
    [dateFormat setDateFormat: @"yyyy-MM-dd HH:mm:ss zzz"];

    cell.textLabel.text = [dateFormat stringFromDate: [self.Logs
        objectAtIndex:indexPath row]];
    cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;

    return cell;
}
```

运行一下，看看效果。在应用能够实现用户需求的前提下，良好的用户体验是我们更高的追求，因为我们的产品不仅仅是具备某种功能的工具，更是一件艺术品，是我们心、智、力的结晶。

5.6 小结

在本章中，首先我们对表视图有了整体的认识，了解了表视图的组成、表视图类的构成、表视图的分类以及表视图的两个重要协议（委托协议和数据源协议），接着讨论了如何实现简单表视图和分节表视图，以及表视图中索引、搜索栏、分组的用法，然后学习了如何对表视图单元格进行删除、插入、移动等操作，最后向大家介绍了表视图UI设计模式方面的内容。

几乎每个应用都会用到导航，本章将为大家介绍平铺导航、标签导航、树形结构导航的使用方式。另外，本章还为大家讲解了3种导航模式的综合用法。这些知识点基本囊括了开发工作中的大部分导航需求，希望大家能有所收获。

6.1 概述

在Cocoa Touch的MVC设计模式中，处于重要地位的视图控制器有很多种，其中有些视图控制器与导航息息相关。

导航指引用户使用你的应用，没有有效的导航，用户就会迷失方向。

6.1.1 视图控制器的种类

在UIKit中，视图控制器有很多，其中有些负责显示视图，而有些起到导航的作用，有些还有其他用途，下面我们将与导航相关的视图控制器整理如下。

- ❑ **UINavigationController**。用于自定义视图控制器的导航。例如，对于两个界面的跳转，我们可以用一个UINavigationController来控制另外两个UIViewController。
- ❑ **UITableViewController**。导航控制器，它与UITableViewController结合使用，能够构建树形结构导航模式。
- ❑ **UITabBarController**。标签栏控制器，用于构建树标签导航模式。
- ❑ **UIPageViewController**。呈现电子书导航风格的控制器。
- ❑ **UISplitViewController**。可以把屏幕分割成几块的视图控制器，主要为iPad屏幕设计。
- ❑ **UIPopoverController**。呈现“气泡”风格视图的控制器，主要为iPad屏幕设计。

视图控制器随着iOS版本的变化而变化，例如UISplitViewController和UIPopoverController是随着iPad的出现而推出的，UIPageViewController则是iOS 5新推出的，主要用于构建电子书和电子杂志应用。

6.1.2 导航模式

如果火车站没有导航标牌，高速公路上没有路标，情况会怎样？毫无疑问，我们会无所适从，甚至手足无措。你的应用是否具备这些“标牌”和“路标”呢？完美的导航能够清晰地指引用户完成任务。导航是应用软件开发中极为重要的部分，想做好也是存在一定难度的。从内容组织形式上考虑，iPhone有3种导航模式，每一种导航模式都对应于不同的视图控制器。

- ❑ **平铺导航模式**。内容没有层次关系，展示的内容都放置在一个主屏幕上，采用分屏或分页控制器进行导航，可以左右或者上下滑动屏幕查看内容。图6-1展示了iPod touch中自带的天气预报应用，它采用分屏进行导航。

❑ 标签导航模式。内容被分成几个功能模块，每个功能模块之间没有什么关系。通过标签管理各个功能模块，点击标签可以切换功能模块。图6-2展示了iPod touch中自带的时钟应用，它采用的就是标签导航模式。



图6-1 平铺导航模式



图6-2 标签导航模式

❑ 树形结构导航模式。内容是有层次的，从上到下细分或者具有分类包含等关系，例如黑龙江省包含了哈尔滨，哈尔滨又包含了道里区、道外区等。图6-3展示了iPod touch中自带的邮件应用，它采用的就是树形结构导航模式。



图6-3 树形结构导航模式

这3种导航模式基本可以满足大部分应用的导航需求。在实际应用中，有时会将几种导航模式组合在一起使用。

6.1.3 模态视图

在导航过程中，有时候需要放弃主要任务转而做其他次要任务，然后再返回到主要任务，这个“次要任务”就是在“模态视图”中完成的。图6-4为模态视图示意图，该图中的主要任务是登录后进入主界面，如果用户没有注册，就要先去“注册”。“注册”是次要任务，当用户注册完成后，他会关闭注册视图，回到登录界面继续进行主任务。

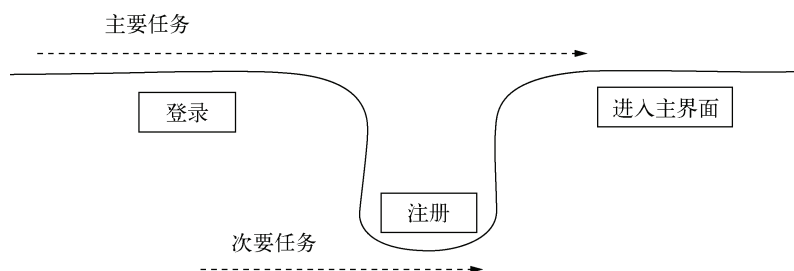


图6-4 模态视图示意图

默认情况下，模态视图是从屏幕下方滑出来的。当完成的时候需要关闭这个模态视图，如果不关闭，就不能做别的事情，这就是“模态”的含义，它具有必须响应处理的意思。因此，模态视图中一定会有“关闭”或“完成”按钮，其根本原因是iOS只有一个Home键。Android和Window Phone就不会遇到这些问题，因为在这两个系统中遇到上述情况时，可以通过Back键返回。

负责控制模态视图的控制器，被称为“模态视图控制器”。“模态视图控制器”并非一个专门的类，它可以是上面提到的控制器的子类。负责主要任务视图的控制器称为“主视图控制器”，它与模态视图控制器之间是“父子”关系。在UIViewController类中，主要有如下两个方法。

❑ **presentViewController:animated:completion.** 呈现模态视图。

❑ **dismissViewControllerAnimated:completion.** 关闭模态视图。

下面我们通过一个案例来介绍模态视图。这个案例有一个登录界面和一个注册界面，在登录界面点击“注册”按钮，会从屏幕下方滑出注册模态视图，如图6-5所示，点击Save或Done按钮后关闭注册视图。此外，点击Save按钮，还可以把数据回传给登录视图。



图6-5 模态视图案例

在此之前，我们学习的应用都只是单视图的，涉及导航必然是多视图的，因此，这个案例会涉及多视图控制器的创建和管理，还有视图之间参数的传递问题。

使用Xcode创建工程ModalViewSample，相关选项如下：模板采用Single View Application，Devices选择iPhone，选择Use Storyboards和Use Automatic Reference Counting复选框。

注意 我们也可以采用Utility Application工程模板，这种模板可以很快帮助我们创建主视图控制器和子视图控制器，其中子视图控制器是模态视图控制器。在本章中，我们不建议采用模板构建应用，这是因为模板会屏蔽技术实现的细节。关于模板使用的问题，我们会在下一章中介绍。

用Interface Builder打开故事板MainStoryboard.storyboard，从对象库中拖曳控件到设计界面，如图6-6所示。

使用 Single View Application 工程模板只能创建一个视图控制器 ViewController 类，在故事板 MainStoryboard.storyboard中有一个视图控制器（ViewController）。现在拖曳视图控制器到设计界面，如图6-7所示。



图6-6 Interface Builder设计登录界面

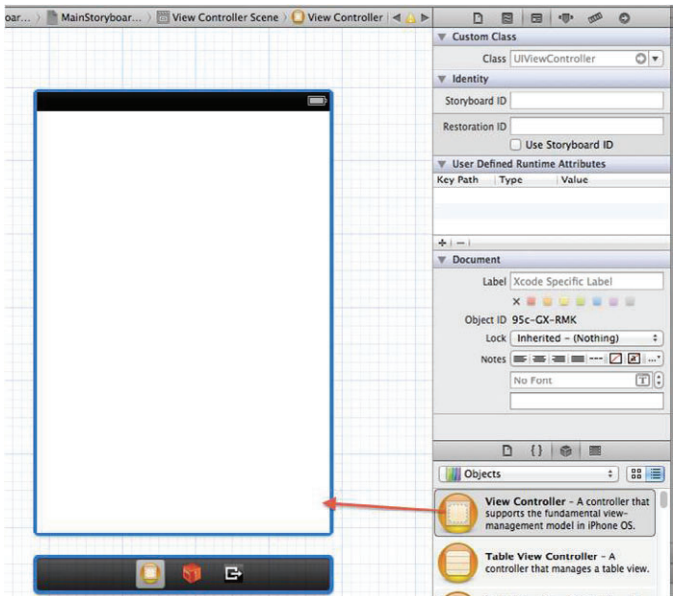


图6-7 拖曳视图控制器到设计界面

选择刚才拖曳进来的视图控制器，打开其标识检查器，修改Storyboard ID为registerViewController，这个ID用于在编码中获得该视图控制器对象。然后，拖曳控件到视图，如图6-8所示。

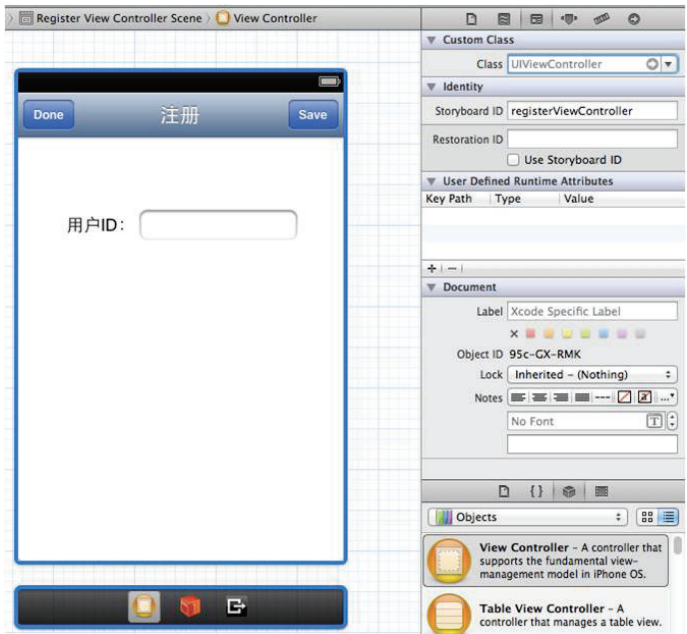


图6-8 模态视图设计界面

然后添加注册视图控制器类，具体操作步骤如下。

(1) 选择File→New→File...菜单项，在打开的Choose a template for your new file对话框中选择Objective-C class文件模板（如图6-9所示）。

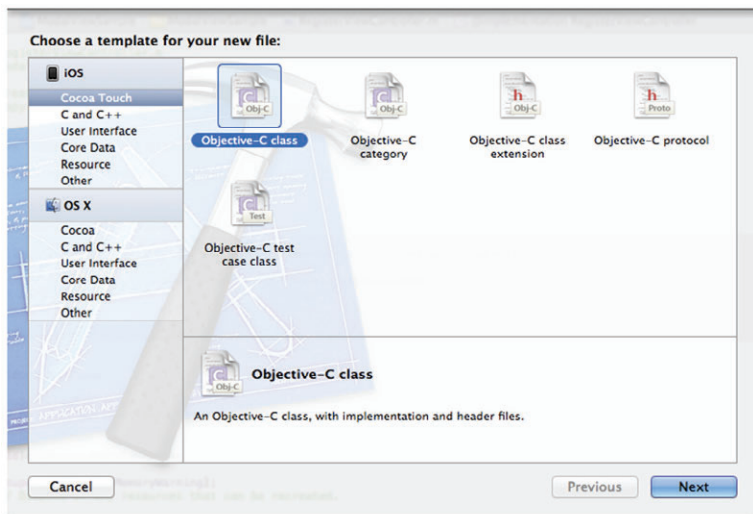


图6-9 选择文件模板

(2) 点击Next按钮，得到的界面如图6-10所示，在Class中输入RegisterViewController类名，从Subclass of下拉列表中选择UIViewController，再取消选择With XIB for user interface复选框。事实上，如果读者对xib比较熟悉，也可以勾选这个选项来创建一个具有xib文件的视图控制器，不过在实例化视图控制器时二者是有区别的。

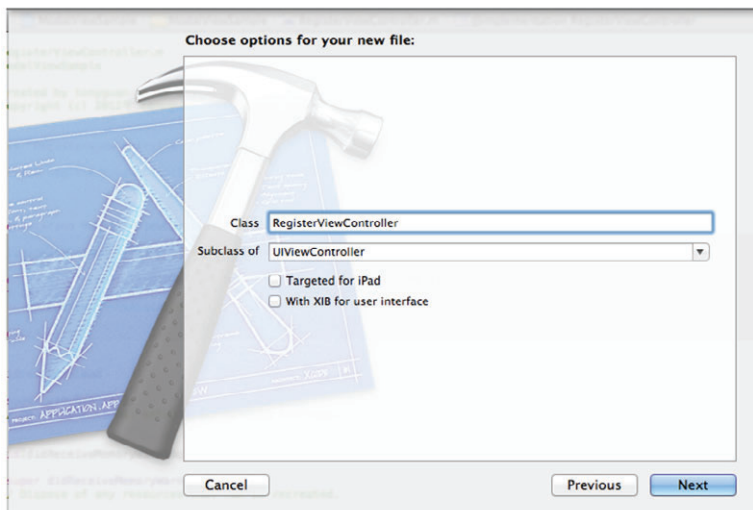


图6-10 输入类名

(3) 点击Next按钮，此时RegisterViewController类就创建好了。然后回到Interface Builder中，选择注册视图控制器，打开其标识检查器，重新选择Class为RegisterViewController，如图6-11所示，此时故事板中的这个视图控制器就与代码中的RegisterViewController对应起来了。

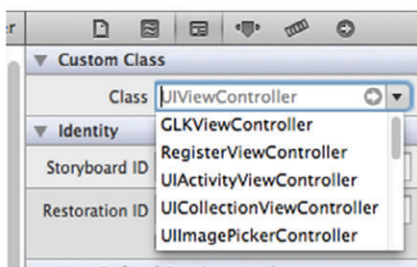


图6-11 重新选择Class选项

下面是两个视图控制器的.h文件，我们还需要为输出口和动作连线：

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

@interface ViewController : UIViewController

- (IBAction)regonclick:(id)sender;

@end

//
//RegisterViewController.h
//ModalViewSample

#import <UIKit/UIKit.h>

@interface RegisterViewController : UIViewController

@property (weak, nonatomic) IBOutlet UITextField *txtUsername;

- (IBAction)done:(id)sender;

@end
```

关于上述代码，我们分两个步骤给大家介绍，第一步是实现模态视图的呈现和关闭，第二步是实现模态视图的参数回传。

1. 模态视图的呈现和关闭

在ViewController.m中，如下代码实现了模态视图的呈现：

```
- (IBAction)regonclick:(id)sender {

    UIStoryboard* mainStoryboard = [UIStoryboard storyboardWithName:@"MainStoryboard"
        bundle:nil];
    UIViewController *registerViewController = [mainStoryboard
        instantiateViewControllerWithIdentifier:@"registerViewController"];

    registerViewController.modalTransitionStyle =
        UIModalTransitionStyleCoverVertical;
    [self presentViewController:registerViewController animated:YES completion:^(
        NSLog(@"Present Modal View");
    )];
}
```

其中前两条语句通过故事板 instantiateViewControllerWithIdentifier 使用 registerViewController 的ID获得视图控制器对象。如果创建 RegisterViewController 时采用了 xib 技术，我们可以通过如下代码创建视图控制器：

```
UIViewController *registerViewController = [[UIViewController alloc]
initWithNibName::@"<RegisterViewController xib文件>" bundle:nil];
```

modalTransitionStyle属性是UIViewController类提供的，用于设定模态视图呈现和关闭时的动画效果，它们是由UIModalTransitionStyle枚举中的常量定义的。UIModalTransitionStyle枚举包含以下常量。

❑ **UIModalTransitionStyleCoverVertical**。呈现时沿垂直方向由底向上退出，覆盖原来视图，关闭时从上往底部退回。

❑ **UIModalTransitionStyleFlipHorizontal**。水平翻转，呈现的时候从右往左翻转，关闭的时候从左往右翻转回来。

❑ **UIModalTransitionStyleCrossDissolve**。两个视图交叉淡入淡出。

❑ **UIModalTransitionStylePartialCurl**。呈现时模态视图卷起一个边角翻页，关闭时模态视图翻回来。

呈现模态视图presentViewController:animated:completion:是iOS 5之后的方法，相较于之前版本的presentModalViewController:方法，iOS 5之后的版本可以在呈现完成时调用completion代码块。completion代码块非常类似于Java中的匿名类，或是JavaScript中的回调函数：

```
completion:^(
    NSLog(@"Present Modal View");
)
```

关闭模态视图是在RegisterViewController.m中实现的，具体代码如下：

```
- (IBAction)done:(id)sender {

    [self dismissViewControllerAnimated:YES completion:^(
        NSLog(@"Modal View done");
    )];
}
```

其中dismissViewControllerAnimated:completion:是iOS 5之后的方法。与之前版本的dismissModalViewControllerAnimated:方法相比较，iOS 5之后的版本可以在关闭完成时调用completion代码块，这一点与呈现方法类似。

2. 模态视图参数回传

很多情况下，在导航过程中我们需要在多个视图之间传递参数。传递方式有很多种，要具体问题具体分析。就本例的模态视图控制器而言，我们可以采用第3章介绍的委托设计模式或广播通知机制进行回传。采用委托设计模式方式回传比较麻烦，需要定义一个协议，很多开源社区都对此有所介绍，这里不再详细介绍。本章就介绍一下如何以广播通知机制实现参数传递。

将ViewController.m的代码修改如下：

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

    [[NSNotificationCenter defaultCenter] addObserver:self
                                              selector:@selector(registerCompletion:)
                                              name:@"RegisterCompletionNotification"
                                              object:nil];
}

- (void)registerCompletion:(NSNotification*)notification {

    NSDictionary *theData = [notification userInfo];
    NSString *username = [theData objectForKey:@"username"];

    NSLog(@"username = %@",username);
}
```

在上述代码中，我们在viewDidLoad方法中注册一个自定义通知RegisterCompletionNotification，通

知到来的时候发出registerCompletion:消息，其参数notification中可以包含回传的参数，它们都放在NSDictionary字典中。

修改RegisterViewController.m代码，在done方法中的关闭完成completion代码块中添加代码投送通知RegisterCompletionNotification，其中通知的参数放在字典dataDict中：

```
- (IBAction)done:(id)sender {  
  
    [self dismissViewControllerAnimated:YES completion:^(  
        NSLog(@"Modal View done");  
  
        NSDictionary *dataDict = [NSDictionary dictionaryWithObject:self.txtUsername.text  
            forKey:@"username"];  
        [[NSNotificationCenter defaultCenter]  
            postNotificationName:@"RegisterCompletionNotification"  
            object:nil  
            userInfo:dataDict];  
  
    }];  
  
}
```

在UIViewController中，与模态视图控制器相关的属性还有modalPresentationStyle。但是与iPad不同的是，modalPresentationStyle在iPhone和iPod touch中都是全屏显示的。

6.2 平铺导航

平铺导航模式是非常重要的导航模式，一般用于简单的扁平化信息浏览。扁平化信息是指这些信息之间没有从属的层次关系，如北京、上海和哈尔滨之间就没有从属关系，而哈尔滨市与黑龙江省之间就是从属的层次关系。

6.2.1 应用场景

图6-12是iPod touch自带的天气应用程序，每一个屏幕展示一个城市最近的天气信息。它是基于分屏导航实现的平铺导航模式。基于分屏导航实现的平铺导航模式可以构建iOS中的实用型应用程序。



图6-12 平铺导航的天气应用

提示 实用型应用程序完成的简单任务对用户输入要求很低。用户打开实用型应用程序是为了快速查看信息摘要或是在少数对象上执行简单任务。天气程序就是一个实用型应用程序的典型例子，它在一个易读的摘要中显示了重点明确的信息。——引自于苹果HIG (iOS Human Interface Guidelines, iOS人机界面设计指导手册)

图6-13是iPad中平铺导航的iBooks电子书应用，它是基于分页导航实现的平铺导航模式，用户可以像翻书一样在页面之间导航，而且在翻动书页时还可以看到下一页或背面的内容，完全模拟真书的效果。



图6-13 iPad中平铺导航的iBooks应用（横屏双页显示）

分页导航在iPad和iPhone横屏情况下是单页显示。图6-14所示是iBooks应用竖屏时的单页显示情况。

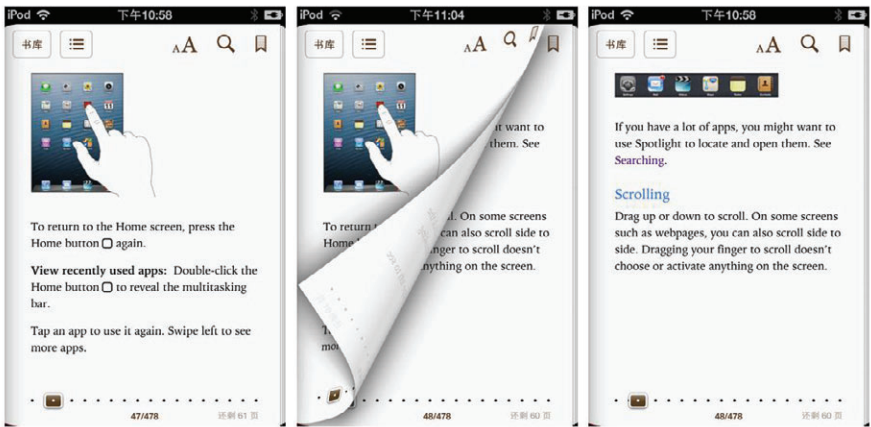



图6-14 iPhone中平铺导航的iBooks应用（竖屏单页显示）

为了进一步了解平铺导航，我们先从一个需求开始介绍平铺导航。如果我想开发一个基于iPhone的“画廊”应用，目前只有3幅名画（如图6-15所示，左图是达芬奇的《蒙娜丽莎》、中图是罗丹的《思想者》、右图是保罗·克利的《肖像》）收录到应用中。这3幅名画之间没有层次关系而是扁平关系。




图6-15 3幅名画

6.2.2 基于分屏导航的实现

基于分屏导航是平铺导航模式的主要实现方式，主要涉及的控件有分屏控件（UIPageControl）和滚动视图（UIScrollView），其中分屏控件是iOS标准控件，图6-12屏幕下方的就是分屏控件，高亮的小点是当前屏幕的位置。

基于分屏导航的手势有两种，一个是点击小点的左边（上边）或右边（下边）实现翻页，另一个是用手在屏幕上滑动实现翻页。屏幕的总数应该限制在20个以内，超过20个小点的分屏控件就会溢出。事实上，当一个应用超过10屏时，使用基于分屏导航的平铺导航模式已经不是很方便了。

下面我们基于分屏导航模式实现“画廊”应用。使用Single View Application模板创建一个名为PageControlNavigation的工程。将UIScrollView和PageControl控件拖曳到设计界面，并按图6-16所示将其摆放到合适的位置，通过属性检查器将视图背景设置为黑色。

在Interface Builder中选中文档中的UIScrollView控件，打开其属性检查器，按照图6-17所示设置Scrollers中的属性，此时该UIScrollView控件不显示水平和垂直滚动条，但可以滚动也可以分屏。

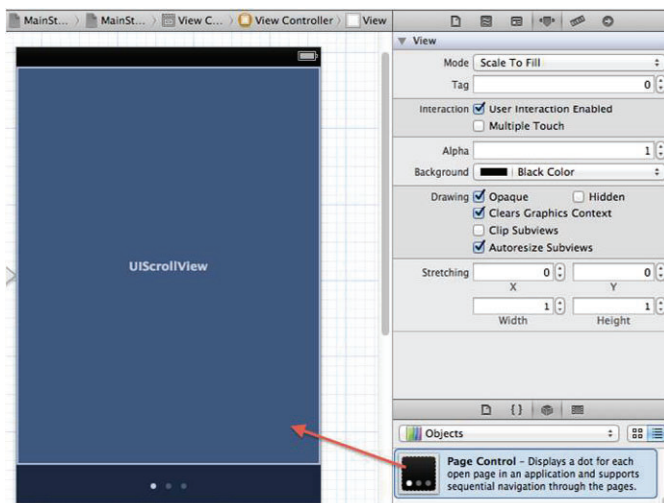


图6-16 拖曳控件到设计界面

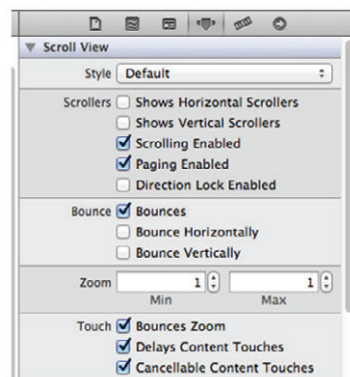



图6-17 设置UIScrollView控件属性

在Interface Builder中选中文档中的PageControl控件，打开其属性检查器，设置Pages中# of Pages（总屏数）属性为3，Current（当前屏）属性为0，如图6-18所示。再打开其尺寸检查器，如图6-19所示，修改Width（宽度）属性为200，将这个属性设置大一些是为了便于手指点击。

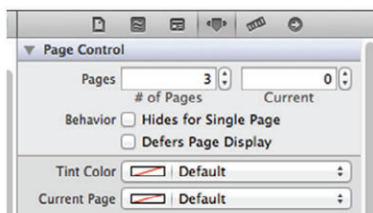


图6-18 分屏控件属性检查器

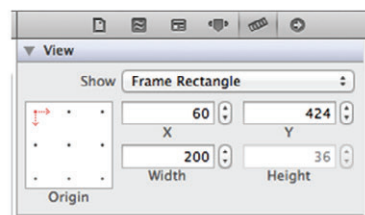


图6-19 分屏控件尺寸检查器

最后，还需要为这两个控件定义输出口并连线，而且要为分屏控件定义响应屏幕变化事件的方法changePage：并连线。

完成之后，ViewController.h文件中增加的代码如下：


```
@property (weak, nonatomic) IBOutlet UIScrollView *scrollView;
@property (weak, nonatomic) IBOutlet UIPageControl *pageControl;
- (IBAction)changePage:(id)sender;
```

下面设计3个视图，将3个View Controller视图控制器拖曳到MainStoryboard.storyboard的设计界面中，然后再分别拖曳3个Image View到3个不同的视图上，如图6-20所示。

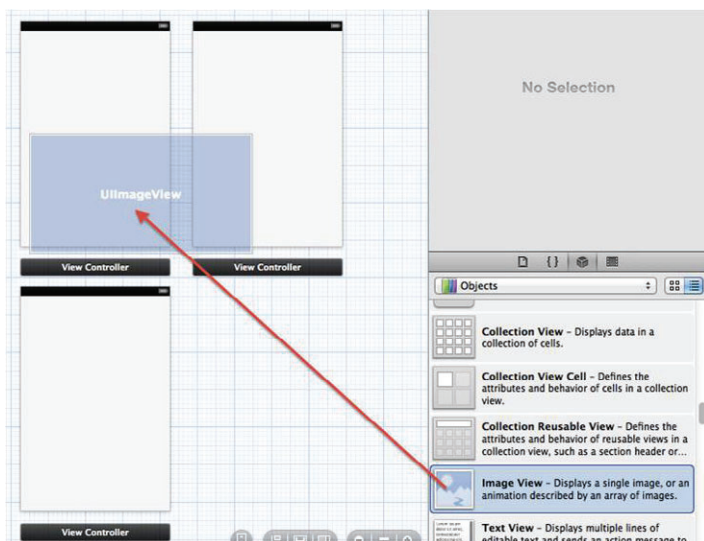


图6-20 拖曳Image View到设计界面

然后再分别修改3个Image View的Image属性为名画的文件名，如图6-21所示。

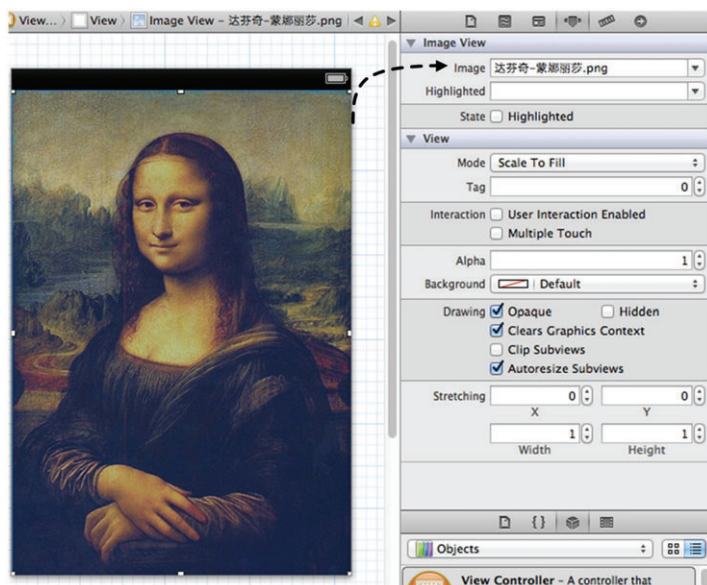


图6-21 修改Image View的Image属性

然后再依次选中视图控制器，将Storyboard ID分别修改为page1、page2和page3。与模态视图的例子不同，我们不需要再创建视图控制器的子类。就本例而言，我们只需展示一些图片。如果需要处理动作事件，则需要自定义

义视图控制器的子类。

设计完成后，我们看看程序代码ViewController.h:

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

@interface ViewController : UIViewController <UIScrollViewDelegate>

@property (strong, nonatomic) UIView *page1;
@property (strong, nonatomic) UIView *page2;
@property (strong, nonatomic) UIView *page3;

@property (weak, nonatomic) IBOutlet UIScrollView *scrollView;
@property (weak, nonatomic) IBOutlet UIPageControl *pageControl;

- (IBAction)changePage:(id)sender;

@end
```

由于需要响应UIScrollView的事件，我们在ViewController中实现了UIScrollViewDelegate协议。下面我们看看ViewController.m中viewDidLoad方法的代码：

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

    self.scrollView.contentSize = CGSizeMake(self.view.frame.size.width*3,
        self.scrollView.frame.size.height);
    self.scrollView.frame = self.view.frame;

    UIStoryboard *mainStoryboard = [UIStoryboard
        storyboardWithName:@"MainStoryboard" bundle:nil];

    UIViewController* page1ViewController = [mainStoryboard
        instantiateViewControllerWithIdentifier:@"page1"];
    self.page1 = page1ViewController.view;
    self.page1.frame = CGRectMake(0.0f, 0.0f, 320.0f, 420.0f);

    UIViewController* page2ViewController = [mainStoryboard
        instantiateViewControllerWithIdentifier:@"page2"];
    self.page2 = page2ViewController.view;
    self.page2.frame = CGRectMake(320.0f, 0.0f, 320.0f, 420.0f);

    UIViewController* page3ViewController = [mainStoryboard
        instantiateViewControllerWithIdentifier:@"page3"];
    self.page3 = page3ViewController.view;
    self.page3.frame = CGRectMake(2 * 320.0f, 0.0f, 320.0f, 420.0f);

    self.scrollView.delegate = self;

    [self.scrollView addSubview:self.page1];
    [self.scrollView addSubview:self.page2];
    [self.scrollView addSubview:self.page3];
}
```

该方法主要进行控件初始化，由于需要计算各个控件的位置，代码比较多，但是基本上没什么难点。
self.scrollView.delegate = self;用于把当前视图控制器（UIScrollViewDelegate实现对象）分配给ScrollView控件，以便响应事件处理，其他事件是scrollViewDidScroll:，其代码如下：

```
- (void) scrollViewDidScroll: (UIScrollView *) aScrollView
{
    CGPoint offset = aScrollView.contentOffset;
    self.pageControl.currentPage = offset.x / 320.0f;
}
```

当左右滑动屏幕，ScrollView控件滚动完成时，需要计算和设定分屏控件的当前屏currentPage。当点击分

屏控件时，屏幕发生变化，此时会触发changePage:方法，其代码如下：

```
- (IBAction)changePage:(id)sender
{
    [UIView animateWithDuration:0.3f animations:^(
        int whichPage = self.pageControl.currentPage;
        self.scrollView.contentOffset = CGPointMake(320.0f * whichPage, 0.0f);
    )];
}
```

在上述代码中，我们根据分屏控件的当前屏幕属性（currentPage）重新调整了ScrollView控件的偏移量，而且为了使屏幕变化产生动画效果，我们使用了[UIView animateWithDuration:0.3f animations:^(...)]代码，重新调整了控件的偏移量。

运行代码，得到的效果如图6-22所示。



图6-22 运行效果

6.2.3 基于分页导航的实现

在iOS 5之后，可以使用分页控制器（UIPageViewController）构建类似于电子书效果的应用，我们称为基于分页的应用。一个分页应用有很多相关的视图控制器，如图6-23所示。

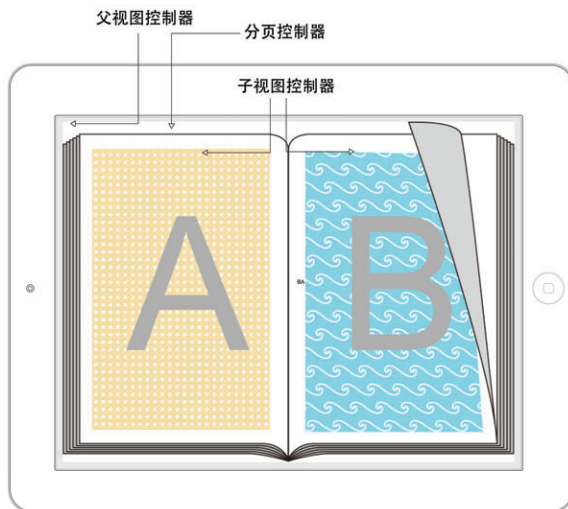


图6-23 分页应用相关的视图控制器

分页控制器需要放置在一个父视图控制器中，在分页控制器下面还要有子视图控制器，每个子视图控制器对应图中的一个页面。

在基于分页导航实现的应用中，需要的类和协议有 `UIPageViewControllerDataSource` 协议、`UIPageViewControllerDelegate` 协议和 `UIPageViewController` 类，其中 `UIPageViewController` 类没有对应的视图类。

`UIPageViewControllerDataSource` 数据源协议中必须要实现的方法有以下两个。

- ❑ **`pageViewController:viewControllerBeforeViewController:`**。返回当前视图控制器之前的视图控制器，用于上一个页面的显示。
- ❑ **`pageViewController:viewControllerAfterViewController:`**。返回当前视图控制器之后的视图控制器，用于下一个页面的显示。

在 `UIPageViewControllerDelegate` 委托协议中，最重要的方法为 `pageViewController:spineLocationForInterfaceOrientation:`，它根据屏幕旋转方向设置书脊位置（Spine Location）和初始化首页。

在 `UIPageViewController` 中，共有两个常用的属性：双面显示（`doubleSided`）和书脊位置（`spineLocation`）。

- ❑ **双面显示**。指在页面翻起时偶数页面会在背面显示。图6-13右图为 `doubleSided` 设置为 YES 的情况，图6-14中图为 `doubleSided` 设置为 NO（单面显示）的情况。单面显示在页面翻起的时候，能够看到页面的背面，背面的内容是当前页面透过去的，与当前内容是相反的镜像。
- ❑ **书脊位置**。书脊位置也是很重要的属性，但是它的 `spineLocation` 属性是只读的，要设置它，需要通过 `UIPageViewControllerDelegate` 委托协议中的 `pageViewController:spineLocationForInterfaceOrientation:` 方法来实现。书脊位置由枚举 `UIPageViewControllerSpineLocation` 定义，该枚举类型下的成员变量如下所示。
 - **`UIPageViewControllerSpineLocationMin`**。定义了书脊位置在书的最左边（或最上面），如图6-24所示，书将从右向左翻（或从下往上翻）。
 - **`UIPageViewControllerSpineLocationMax`**。定义了书脊位置在书的最右边（或最下面），如图6-25所示，书将从左向右翻（或从上往下翻）。

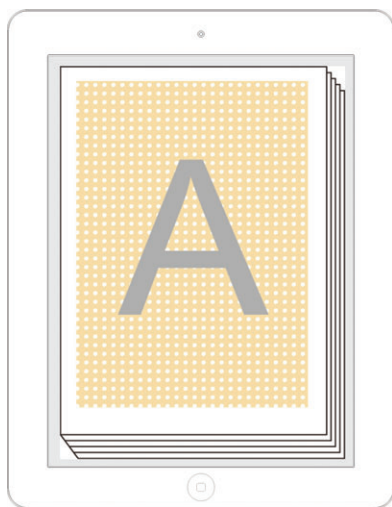


图6-24 书脊在最左边

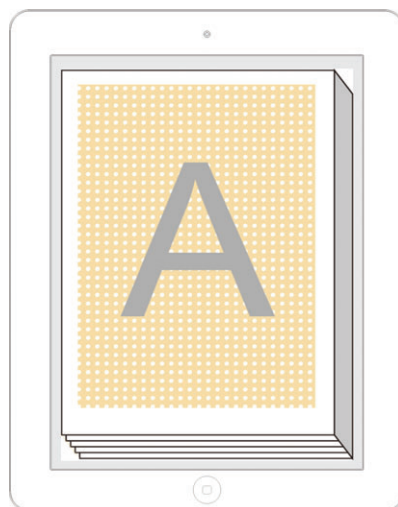


图6-25 书脊在最右边

- **`UIPageViewControllerSpineLocationMid`**。定义了书脊位置在书的中间，如图6-26所示，一般会在横屏下显示，屏幕分成两个页面。

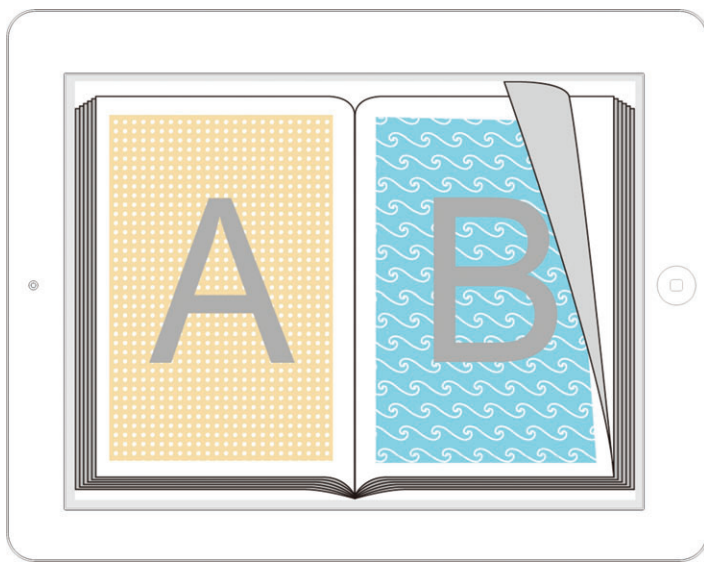


图6-26 书脊在中间

下面我们使用页面导航实现城市信息这个应用。使用Single View Application模板创建一个名为PageNavigation的工程。

注意 也可以采用Xcode工程模板Page-Based Application构建分页应用程序，但是会产生很多类，它们之间的关系比较复杂，这不利于初学者学习，因此不建议采用。

按照上一节基于分屏控件的导航实现中的方法创建3个视图控制器。另外，一个简单的方法是直接从PageControlNavigation工程中复制过来。打开MainStoryboard.storyboard，选中3个视图控制器，按下control+C组合键复制，再到PageNavigation中打开MainStoryboard.storyboard，按下control+V组合键粘贴即可。

这样UI设计工作就结束了。下面的工作都是由代码完成的。我们先看看ViewController.h的代码：

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

@interface ViewController : UIViewController
    <UIPageViewControllerDataSource,UIPageViewControllerDelegate>
{
    // 当前页面的索引
    int pageIndex;
}
@property (strong, nonatomic) UIPageViewController *pageViewController;

@end
```

在上述代码中，ViewController实现了UIPageViewControllerDataSource和UIPageViewControllerDelegate协议。成员变量pageIndex保存了当前页面的索引，pageViewController属性保存了UIPageViewController实例。

下面我们看看ViewController.m中的viewDidLoad方法，其代码如下所示：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.view.frame = CGRectMake(0.0f, 0.0f, 320.0f, 440.0f);
}
```

```

self.pageViewController = [[UIPageViewController alloc]
    initWithTransitionStyle:UIPageViewControllerTransitionStylePageCurl
    navigationOrientation:UIPageViewControllerNavigationOrientationHorizontal
    options:nil];

self.pageViewController.delegate = self;
self.pageViewController.dataSource = self;

UINavigationController *mainStoryboard = [UINavigationController
    storyboardWithName:@"MainStoryboard" bundle:nil];
UIViewController* page1ViewController = [mainStoryboard
    instantiateViewControllerWithIdentifier:@"page1"];

// 第一个视图，作为PageViewController首页
NSArray *viewControllers = @[page1ViewController];

[self.pageViewController setViewControllers:viewControllers
    direction:UIPageViewControllerNavigationDirectionForward animated:YES
    completion:NULL];

[self.view addSubview:self.pageViewController.view];

pageIndex = 0;
}

```

在上述代码中，initWithTransitionStyle:navigationOrientation:options:构造方法用于创建UIPageViewController实例，initWithTransitionStyle用于设定页面翻转的样式。UIPageViewControllerTransitionStyle枚举类型定义了如下两个翻转样式。

❑ **UIPageViewControllerTransitionStylePageCurl**。翻书效果样式。

❑ **UIPageViewControllerTransitionStyleScroll**。滑屏效果样式。

navigationOrientation设定了翻页方向，UIPageViewControllerNavigationOrientation枚举类型定义了以下两种翻页方式。

❑ **UIPageViewControllerNavigationOrientationHorizontal**。水平方向。

❑ **UIPageViewControllerNavigationOrientationVertical**。垂直方向。

代码NSArray *viewControllers = @[page1ViewController] 相当于 NSArray *viewControllers = [NSArray arrayWithObject: page1ViewController, nil]。

在UIPageViewController中，setViewControllers:direction:animated:completion:方法用于设定首页中显示的视图。首页中显示几个视图与书脊类型有关，如果书脊类型是UIPageViewControllerSpineLocationMin或UIPageViewControllerSpineLocationMax，首页中将显示一个视图；如果是UIPageViewControllerSpineLocationMid，首页中显示两个视图。

[self addChildViewController:self.pageViewController] 语句是将PageViewController添加到父视图控制器中去。

我们再看看ViewController.m中数据源协议UIPageViewControllerDataSource的实现代码：

```

- (UIViewController *)pageViewController:(UIPageViewController *)pageViewController
    viewControllerBeforeViewController:(UIViewController *)viewController
{
    pageIndex--;

    if (pageIndex < 0){
        pageIndex = 0;
        return nil;
    }
}

```

```

    UIStoryboard *mainStoryboard = [UIStoryboard
        storyboardWithName:@"MainStoryboard" bundle:nil];
    NSString *pageId = [NSString stringWithFormat:@"page%i", pageIndex+1];
    UIViewController* pvController = [mainStoryboard
        instantiateViewControllerWithIdentifier:pageId];

    return pvController;
}

- (UIViewController *)pageViewController:(UIPageViewController *)pageViewController
    viewControllerAfterViewController:(UIViewController *)viewController
{
    pageIndex++;

    if (pageIndex > 2){
        pageIndex = 2;
        return nil;
    }

    UIStoryboard *mainStoryboard = [UIStoryboard
        storyboardWithName:@"MainStoryboard" bundle:nil];
    NSString *pageId = [NSString stringWithFormat:@"page%i", pageIndex+1];
    UIViewController* pvController = [mainStoryboard
        instantiateViewControllerWithIdentifier:pageId];

    return pvController;
}

```

在ViewController.m中, 有关委托协议UIPageViewControllerDelegate实现方法的代码如下:

```

- (UIPageViewControllerSpineLocation)pageViewController:(UIPageViewController *) pageViewController
    spineLocationForInterfaceOrientation:(UIInterfaceOrientation)orientation
{
    self.pageViewController.doubleSided = NO;
    return UIPageViewControllerSpineLocationMin;
}

```

由于spineLocation属性是只读的, 所以只能在这个方法中设置书脊位置。该方法可以根据屏幕旋转方向的不同来动态设定书脊的位置, 实现代码可以参考下面的代码:

```

- (UIPageViewControllerSpineLocation)pageViewController:(UIPageViewController *) pageViewController
    spineLocationForInterfaceOrientation:(UIInterfaceOrientation)orientation
{
    UIStoryboard *mainStoryboard = [UIStoryboard
        storyboardWithName:@"MainStoryboard" bundle:nil];
    UIViewController* page1ViewController = [mainStoryboard
        instantiateViewControllerWithIdentifier:@"page1"];
    UIViewController* page2ViewController = [mainStoryboard
        instantiateViewControllerWithIdentifier:@"page2"];

    if (orientation == UIInterfaceOrientationLandscapeLeft || orientation ==
        UIInterfaceOrientationLandscapeRight)
    {
        NSArray *viewControllers = @[page1ViewController, page2ViewController];
        [self.pageViewController setViewControllers:viewControllers
            direction:UIPageViewControllerNavigationDirectionForward animated:YES
            completion:NULL];
        self.pageViewController.doubleSided = NO;
        return UIPageViewControllerSpineLocationMid;
    }
    NSArray *viewControllers = @[page1ViewController];
    [self.pageViewController setViewControllers:viewControllers

```



```

        direction:UIPageViewControllerNavigationDirectionForward animated:YES
        completion:NULL];
self.pageViewController.doubleSided = NO;
return UIPageViewControllerSpineLocationMin;
}

```

这只是一个基本的实现，要根据具体的应用具体再定。用平铺导航实现时，UIPageViewController往往不需要实现屏幕旋转的支持，而且书脊的位置也不会设置在中间。

代码编写完毕后，我们可以运行一下看看效果，如图6-27所示。



图6-27 运行效果

6.3 标签导航

标签导航模式是非常重要的导航模式。使用标签栏时，有一定的指导原则：标签栏位于屏幕下方，占有49点的屏幕空间，有时可以隐藏起来；为了点击方便，标签栏中的标签不能超过5个，如果超过5个，则最后一个显示为“更多”，点击“更多”标签会出现更多的列表，如图6-28所示。

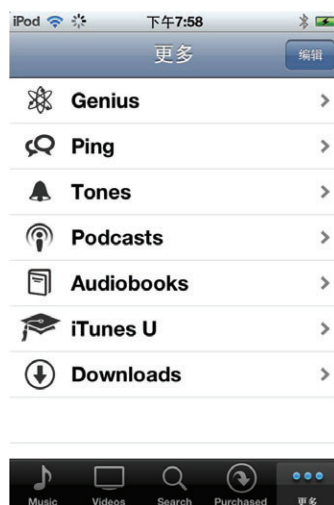


图6-28 “更多”标签

6.3.1 应用场景

对于中国东北三省的城市信息数据，如果把它们分成三组，你会怎么分呢？首先，考虑的是按照行政区划。

❑ **第一组：**哈尔滨、齐齐哈尔、鸡西、鹤岗、双鸭山、大庆、伊春、佳木斯、七台河、牡丹江、黑河、绥化，这12个城市归黑龙江省管辖。

❑ **第二组：**长春、吉林、四平、辽源、通化、白山、松原、白城，这8个城市归吉林省管辖。

❑ **第三组：**沈阳、大连、鞍山、抚顺、本溪、丹东、锦州、营口、阜新、辽阳、盘锦、铁岭、朝阳、葫芦岛，这14个城市归辽宁省管辖。

小组内部的数据有一定的关联关系，它们同属于一个行政管辖区域，小组之间互相独立，这就是标签导航模式适用的情况。

按照这样的分组方式在iPhone上摆放这些城市，仍然会分成3个屏幕，如图6-29所示，标签名就是省的名字，当选中某个省的标签时，屏幕会显示出该省的城市信息，而且标签是高亮显示的。

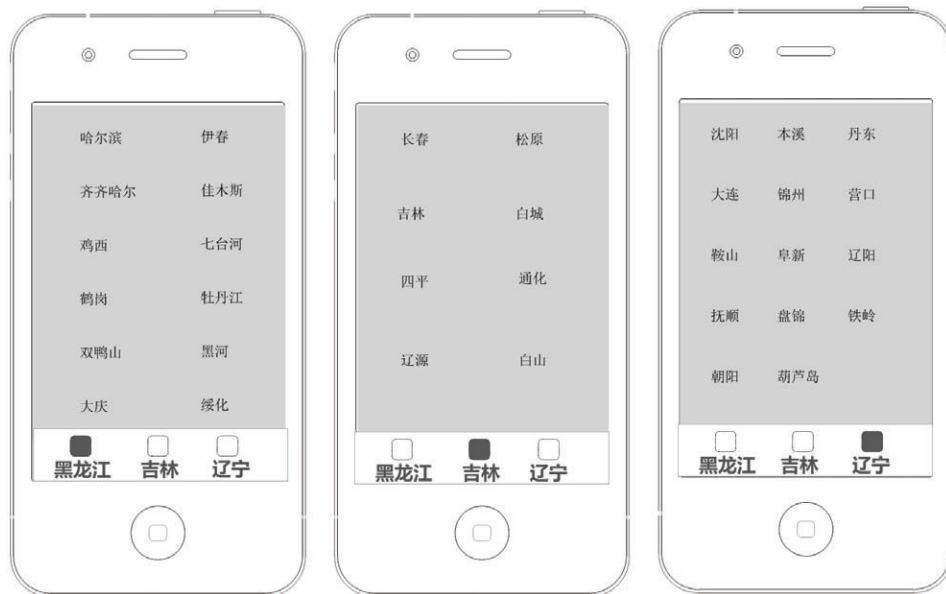


图6-29 标签导航模式

在开发具体应用的时候，标签导航模式的各个标签分别代表一个功能模块，各功能模块之间相对独立。

6.3.2 nib实现

在Xcode 4.5中，可以使用工程模板Tabbed Application创建标签导航模式的应用，这可以通过故事板或nib技术实现。故事板创建简单但屏蔽了标签导航的很多技术细节。为了展示两者的区别，在实现环节我们分别采用了nib技术和故事板技术。首先我们来看看用nib技术实现的过程。

使用Tabbed Application模板创建一个名为TabNavigationNib的工程。注意不要选中Use Storyboards复选框，如图6-30所示。

创建的工程如图6-31所示，其中有两个视图控制器。FirstViewController.h和FirstViewController.m是第一个视图控制器，first.png和first@2x.png是标签上的图标，其中first.png是普通屏幕需要的图标，first@2x.png是视网膜显示屏需要的图标。Second View的界面与First View类似。

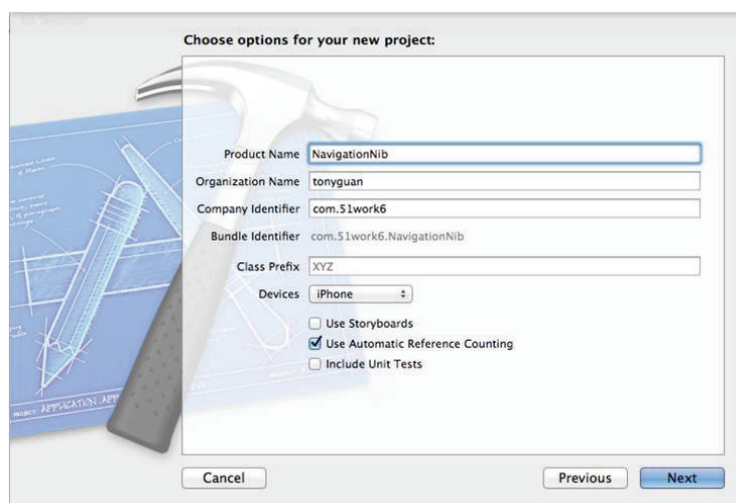


图6-30 选择Tabbed Application工程模板

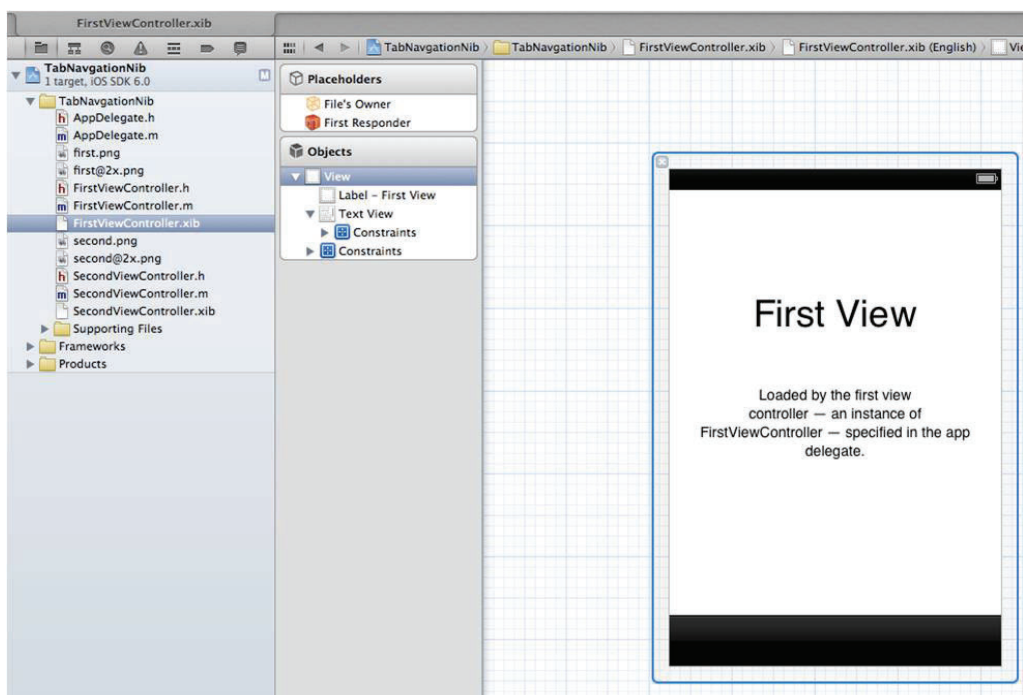


图6-31 创建的工程

下面我们对模板生成的视图控制器做一些修改。首先，修改视图控制器的名称，此时最好使用Xcode提供的工具，它会修改所有依赖关系的相关代码（程序中硬编码的字符串部分除外）。打开FirstViewController.h文件，在代码中双击鼠标，选中要重新命名的类名FirstViewController并右击，从弹出的快捷菜单中选择Refactor→Rename菜单项，如图6-32所示，此时将打开重新输入名字的对话框，输入要修改的类名HeiViewController，接着还有一个预览对话框，如果没有问题，点击Save按钮保存即可。采用同样的方法将SecondViewController改为JiViewController。

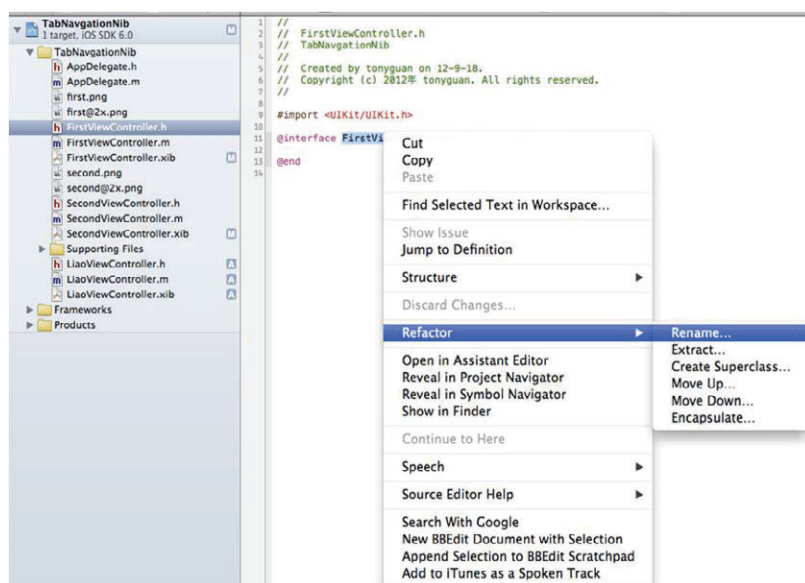


图6-32 重新命名类

由于使用工程模板预先生成了两套视图控制器，而城市信息需要3个视图控制器，我们可以在模板生成的两个视图控制器的基础上再增加一个视图控制器LiaoViewController，具体操作方法是：在菜单栏中选择File→New→File...，然后在文件模板中选择iOS→Objective-C，此时会出现新建文件对话框，如图6-33所示，在Class项目中输入LiaoViewController，在Subclass of中选择UIViewController，然后选中With XIB for user interface复选框。

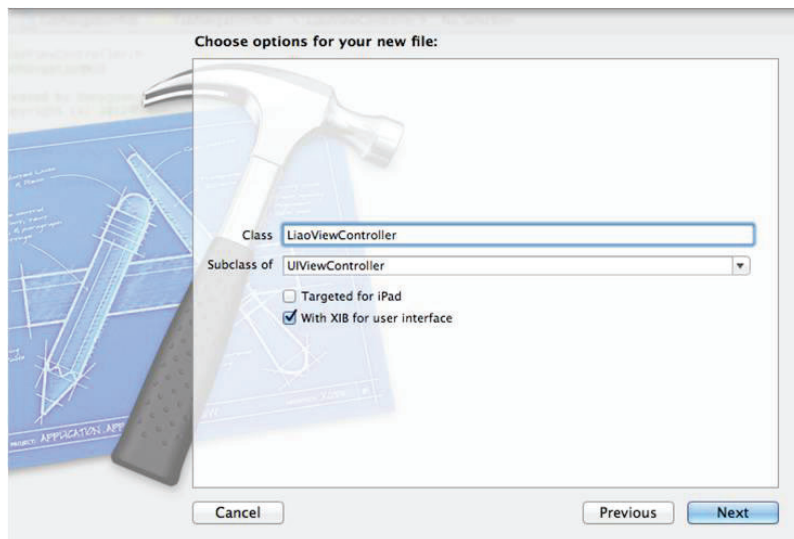


图6-33 新建文件对话框

为了把视图很好地放到标签栏应用中，需要在视图底部添加标签栏控件，这个控件占有49点。如图6-34所示，选中视图，打开其属性检查器，选择Bottom Bar为Tab Bar，这样视图下面就会出现一个黑框。这样做的目的是在设计界面时提醒我们不要把其他控件放到标签栏中，否则运行时该部分控件就会被标签栏遮挡。

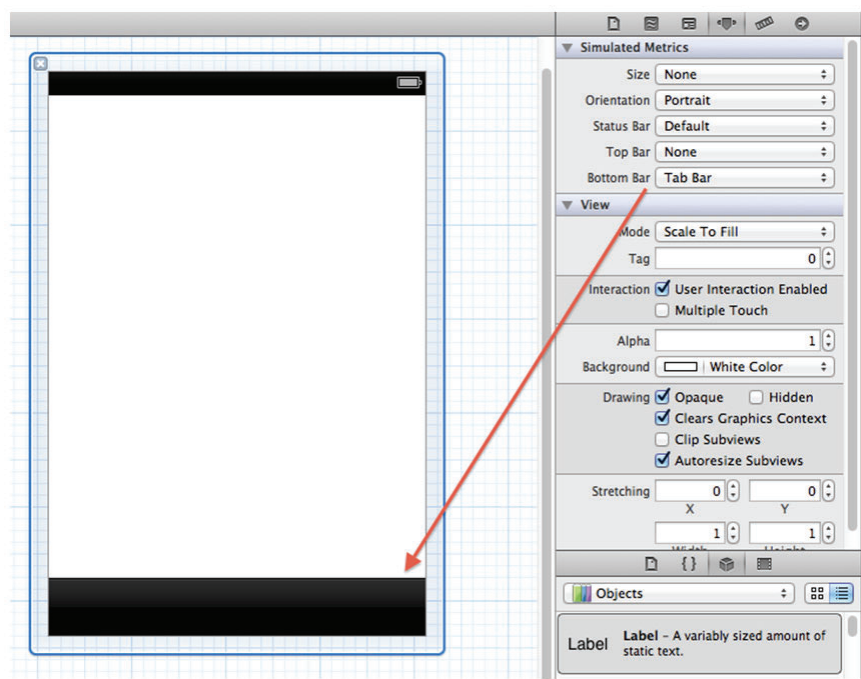


图6-34 添加标签栏控件

将3个视图设计为图6-35所示的效果：拖曳一些Label控件，摆放好位置，修改成图中所示的城市的名字，然后再修改视图背景的颜色。

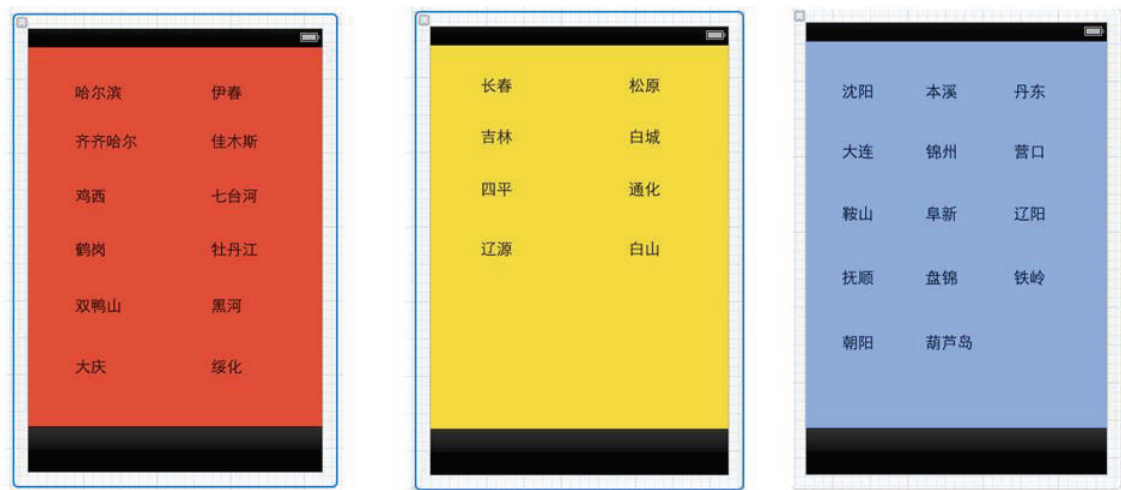


图6-35 设计3个视图

完成时还需要更换图标，删除原来的4个图标文件，将本书提供的源代码工程的icons文件夹添加到本工程中，如图6-36所示。

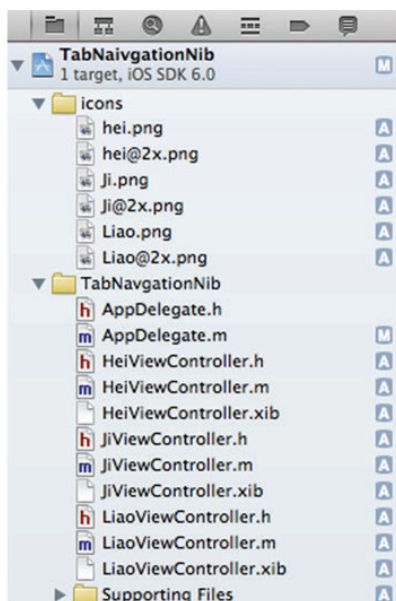


图6-36 添加icons文件夹到工程中

下面我们看看代码部分。应用程序委托对象AppDelegate.m的有关代码如下：

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

    UIViewController *viewController1 = [[HeiViewController alloc]
        initWithNibName:@"HeiViewController" bundle:nil];
    UIViewController *viewController2 = [[JiViewController alloc]
        initWithNibName:@"JiViewController" bundle:nil];
    UIViewController *viewController3 = [[LiaoViewController alloc]
        initWithNibName:@"LiaoViewController" bundle:nil];

    self.tabBarController = [[UITabBarController alloc] init];
    self.tabBarController.viewControllers = @[viewController1,
        viewController2, viewController3];
    self.window.rootViewController = self.tabBarController;
    [self.window makeKeyAndVisible];
    return YES;
}
```

这部分代码是在应用启动时调用的。如果采用故事板构建的话，该方法中这部分代码可以省略。委托对象有一个属性tabBarController，它是UITabBarController类型。UITabBarController是标签栏视图控制器，它的一个重要属性是viewControllers，它是NSArray数组类型，用于存放所有标签栏视图控制器所控制的各个模块的视图控制器。通过self.window.rootViewController = self.tabBarController这条语句，设置标签栏视图控制器为应用的根视图控制器。AppDelegate的window属性、tabBarController属性和模块视图控制器之间的关系如图6-37所示。

在应用程序委托对象中，window属性是UIWindow的实例。每个应用只有一个UIWindow对象来作为应用的“窗口”，“窗口”中的根视图控制器是标签栏控制器。

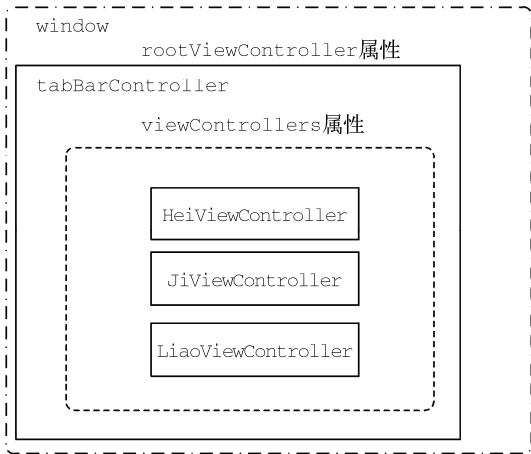


图6-37 window属性、tabBarController属性和模块视图控制器之间的关系

我们再看一个模块的视图控制器HeiViewController.m的有关代码：

```
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        self.title = @"黑龙江";//NSLocalizedString(@"First", @"First");
        self.tabBarItem.image = [UIImage imageNamed:@"hei"];
    }
    return self;
}
```

这是视图控制器的构造方法，其中self.title设定了标签栏中标签的文本，self.tabBarItem.image设定了标签栏中的图标。其他两个视图控制器与此类似，这里不再介绍。

代码编写完毕后，得到的结果如图6-38所示。



图6-38 运行效果

6.3.3 故事板实现

在上一节中，我们介绍的是用nib技术实现标签导航模式，这一节我们将用故事板技术来实现标签导航模式。用故事板技术实现标签导航很简单，不需要编写任何代码。

使用Tabbed Application模板创建一个名为TabNavigationStoryborad的工程,其中需要选中Use Storyboards和Use Automatic Reference Counting复选框。创建完成之后,打开MainStoryboard.storyboard,如图6-39所示。

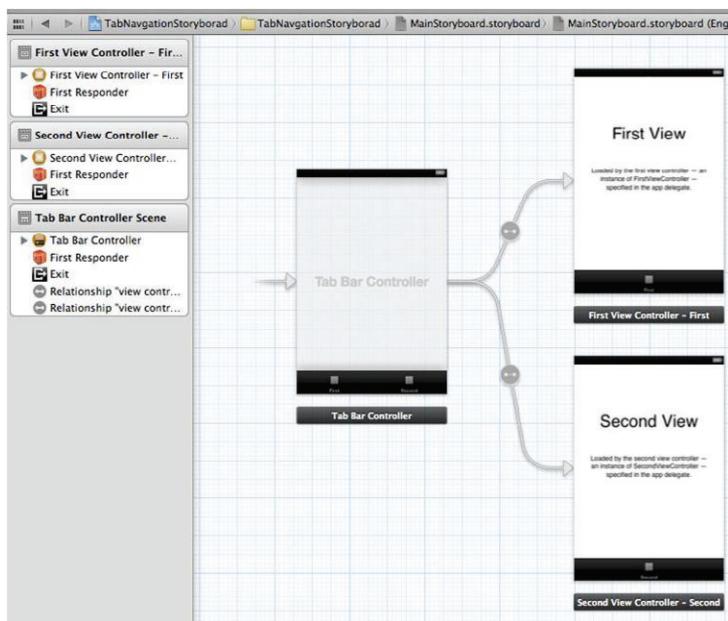


图6-39 使用故事板创建TabNavigationStoryborad工程

图6-39所示的3个Controller Scene会由一些线连接起来,这些线就是Segue。故事板开始的一端是Tab Bar Controller Scene,它是根视图控制器。图中有两个Segue,用来描述Tab Bar Controller Scene与First View Controller Scene和Second View Controller Scene之间的关系。

我们需要先修改两个现有的场景(Scene),然后再添加一个场景,才能满足我们的业务需求。修改两个现有的场景很简单,直接修改视图控制器名就可以了,然后场景就会跟着变化。添加一个场景到设计界面中,然后从对象库中拖曳一个View Controller到设计界面中,如图6-40所示。

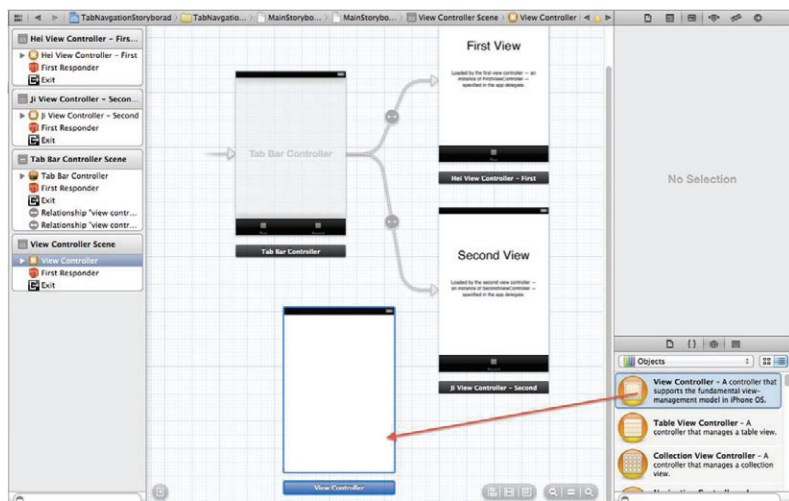


图6-40 添加一个场景到设计界面

此外, 还需要连线添加的场景和Tab Bar Controller Scene, 具体操作是: 按住control键从Tab Bar Controller Scene 拖曳鼠标到View Controller Scene, 释放鼠标, 从弹出菜单中选择view controllers项, 此时连线就做好了, 如图6-41所示。

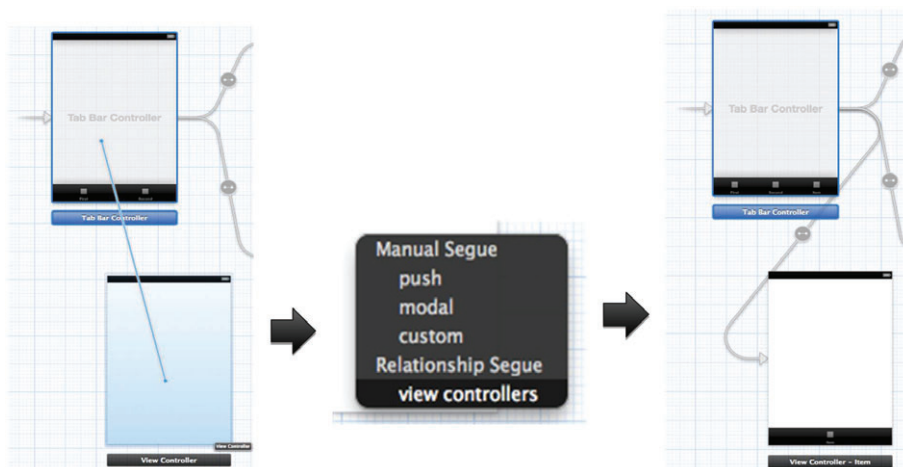


图6-41 连线两个场景

然后我们再添加一个视图控制器类LiaoViewController。在菜单栏中选择File→New→File... , 在文件模板中选择iOS→Objective-C, 此时将弹出“新建文件”对话框, 在Class项目中输入LiaoViewController, 从Subclass of下拉列表中选择UIViewController, 不选中With XIB for user interface复选框。再回到Interface Builder中, 选中View Controller Scene, 打开其标识检查器, 将Custom Class中的Class设为LiaoViewController。

然后添加图标到工程中, 修改标签栏项目中的图标和文本, 具体操作方法为: 选择Hei View Controller Scene→Hei View Controller→Tab Bar Item, 打开其属性检查器, 如图6-42所示, 将Bar Item下的Title设为“黑龙江”, 从Image下拉列表中选择hei.png。按照同样的办法修改其他两个视图控制器。

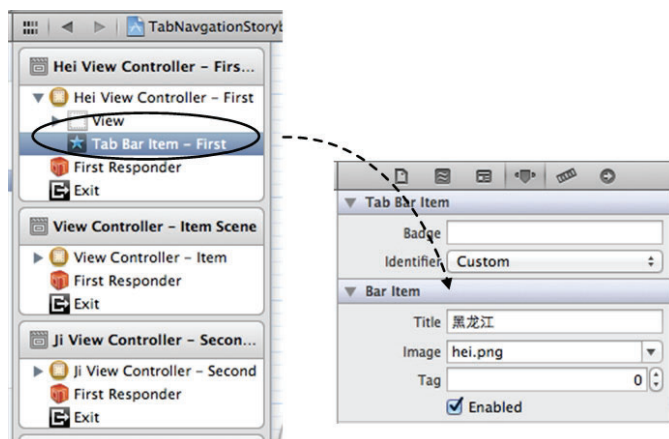


图6-42 修改标签栏项目中图标和文本

三个视图的设计可以参考nib实现部分, 拖曳一些Label控件, 摆放好位置, 修改城市名字, 然后再修改视图背景颜色。

此时基于故事板的标签导航模式就实现了, 整个过程中我们没有编写一行代码。

6.4 树形结构导航

树形结构导航模式也是非常重要的导航模式，它将导航视图控制器（UINavigationController）与表视图结合使用，主要用于构建有从属关系的导航。这种导航模式采用分层组织信息的方式，可以帮助我们构建iOS效率型应用程序。

提示 效率型应用程序具有组织和操作具体信息的功能，通常用于完成比较重要的任务。效率型应用程序通常分层组织信息，相册应用是其典型例子。——引自于苹果HIG（iOS Human Interface Guidelines，iOS人机界面设计指导手册）

6.4.1 应用场景

这里的应用场景同样是按照行政区划来展示东北三省的城市信息，详情可参见6.3.1节的分组。

对于每一个城市，如果还想看到更加详细的信息，比如想知道长春市在百度百科上的信息网址<http://baike.baidu.com/view/2172.htm>，这种情况下吉林省→长春→网址就构成了一种从属关系，是一种层次模型，此时就可以使用树形导航模式。如果按照这样的分组在iPhone上展示这些城市信息，需要使用三级视图，如图6-43所示。



图6-43 树形导航模式

iPod touch自带的邮件应用如图6-44所示，它采用的就是树形结构的导航模式，所有界面的顶部都有一个导航栏。第一个界面是树形结构中的“树根”，我们称为“根视图”；第二个界面是二级视图，它是“树干”；第三个界面是三级视图，是“树叶”。“树根”和“树干”采用表视图，因为表视图在分层组织信息方面的优势尤为突出。从理论上讲，“树干”还可以有多级，但是注意不要太多，“树叶”一般是一个普通的视图，它能够完成具体展示的功能。

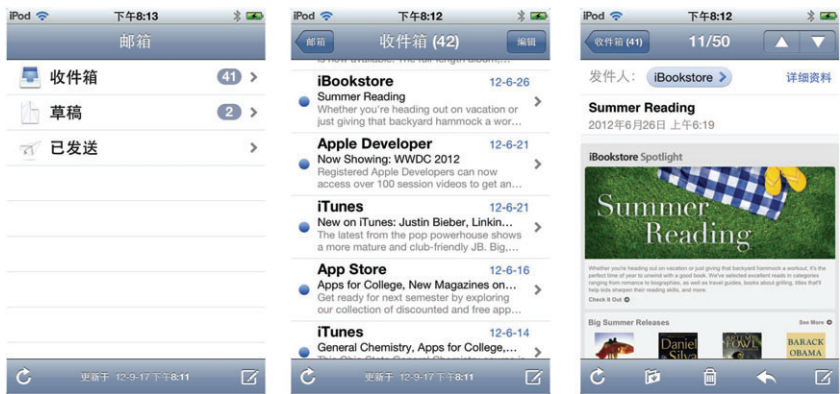


图6-44 iPod touch自带的邮件应用

我们可以为“根视图”的导航栏添加左右按钮，但是二级和三级视图的左按钮是由导航控制器自己添加的，它是汉泽尔与格莱特散在路上的“面包屑”^①，你没有权利自己定义这个按钮，否则用户就会迷失在你的应用之中。树形结构导航模式的缺点是你怎样进来，就要怎样原路返回，这一点与标签导航模式不同，后者可以很快在各个模块之间切换。

6.4.2 nib实现

使用Single View Application模板和nib技术创建一个名为TreeNavigationNib的工程。打开ViewController.xib文件，然后从对象库中拖曳一个Table View到View上面，如图6-45所示，并为其定义输出口。

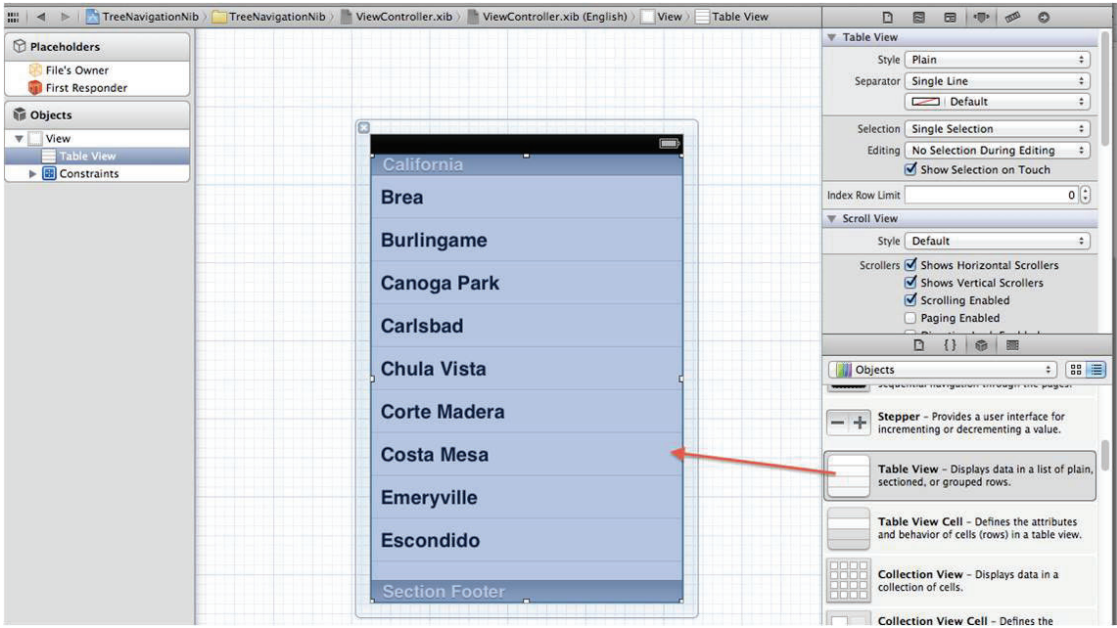


图6-45 拖曳Table View到View上面

^① 引自于格林兄弟所收录的德国童话《糖果屋》(德语: Hänsel und Gretel), 又译《汉泽尔与格莱特》。

ViewController是根视图控制器，下面新建一个二级视图控制器CitiesViewController，具体操作方法是：选择菜单File→New→File...，在打开的Choose a template for your new file对话框的User Interface中选择Objective-C class文件模板，选择父类UITableViewController，不需要选中With XIB for user interface复选框，因为表视图控制器一般不需要nib文件。

新建三级视图控制器DetailViewController，具体操作方法是：选择菜单File→New→File...，在打开的Choose a template for your new file对话框的User Interface中选择Objective-C class文件模板，选择父类UIViewController，需要勾选With XIB for user interface。生成之后，在Interface Builder中打开DetailViewController.xib，然后从对象库中拖曳一个Web View到View上面来，并为其定义输出口连线。

下面我们看看代码部分。由于需要编写应用程序委托对象，所以将AppDelegate.h代码修改如下：

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

@interface AppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;

@property (strong, nonatomic) UINavigationController *navigationController;

@end
```

在上述代码中，我们定义了两个属性——window和navigationController，其中navigationController作为window的根视图控制器，控制整个应用的其他控制器。

将应用程序启动方法的代码修改如下：

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

    ViewController* viewController = [[ViewController alloc]
        initWithNibName:@"ViewController" bundle:nil];
    self.navigationController = [[UINavigationController alloc]
        initWithRootViewController:viewController];

    self.window.rootViewController = self.navigationController;
    [self.window makeKeyAndVisible];
    return YES;
}
```

在上述代码中，我们使用initWithNibName:bundle:构造方法从xib文件中创建viewController对象。UINavigationController的initWithRootViewController:构造方法使用viewController作为导航控制器的根视图控制器来创建一个导航控制器对象，并将导航控制器对象放到属性self.navigationController中。self.window.rootViewController = self.navigationController将导航控制器作为应用的根视图控制器。

注意 在树形结构导航模型中，会有两个根视图控制器：一个是应用程序根视图控制器，它是UINavigationController的实例，通过self.window.rootViewController属性指定；另一个是导航控制器根视图控制器，通过UINavigationController的构造方法initWithRootViewController:指定，用于提供和呈现导航控制器的一级视图，即我们看到的第一个界面。

window属性、initWithRootViewController属性和各级视图控制器之间的关系如图6-46所示。

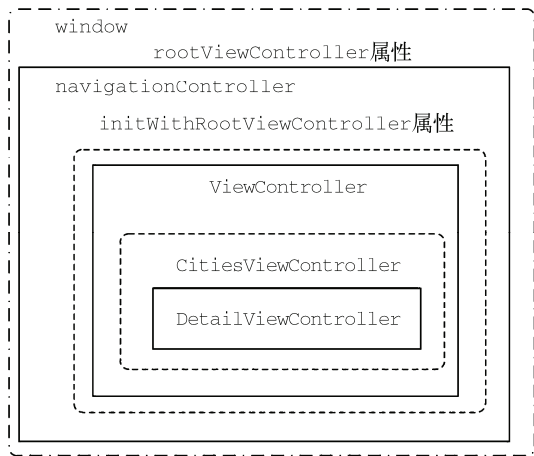


图6-46 window属性、initWithRootViewController属性和各级视图控制器之间的关系

下面看看下一级视图控制器ViewController.h的代码，具体如下：

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

@interface ViewController : UIViewController
    <UITableViewDataSource, UITableViewDelegate>

@property (weak, nonatomic) IBOutlet UITableView *tableView;

@property (strong, nonatomic) NSDictionary *dictData;
@property (strong, nonatomic) NSArray *listData;

@end
```

由于ViewController本身不是UITableViewController的子类，所以它需要实现UITableViewDataSource和UITableViewDelegate协议。dictData是从.plist文件中读取的内容，listData是一级视图显示需要的列表数据。plist文件结构如图6-47所示，从中可以看出listData是dictData的键的集合。

Key	Type	Value
▼ Root	Dictionary	(3 items)
▼ 吉林省	Array	(8 items)
▼ Item 0	Dictionary	(2 items)
name	String	长春
url	String	http://baike.baidu.com/view/2172.htm
▼ Item 1	Dictionary	(2 items)
name	String	吉林
url	String	http://baike.baidu.com/view/2171.htm
▶ Item 2	Dictionary	(2 items)
▶ Item 3	Dictionary	(2 items)
▶ Item 4	Dictionary	(2 items)
▶ Item 5	Dictionary	(2 items)
▶ Item 6	Dictionary	(2 items)
▶ Item 7	Dictionary	(2 items)
▶ 黑龙江省	Array	(12 items)
▶ 辽宁省	Array	(14 items)

图6-47 .plist文件结构

再看下一级视图控制器ViewController.m的代码，具体如下：

```

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.tableView.delegate = self;
    self.tableView.dataSource = self;

    NSBundle *bundle = [NSBundle mainBundle];
    NSString *path = [bundle pathForResource:@"provinces_cities"
        ofType:@"plist"];

    self.dictData = [[NSDictionary alloc] initWithContentsOfFile:path];

    self.listData = [self.dictData allKeys];

    self.title = @"城市信息";
}

#pragma mark - 实现表视图数据源方法
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection:(NSInteger)section
{
    return [self.listData count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    }

    NSInteger row = [indexPath row];
    cell.textLabel.text = [self.listData objectAtIndex:row];
    cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;

    return cell;
}

#pragma mark - 实现表视图委托方法
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSInteger row = [indexPath row];

    CitiesViewController *citiesViewController = [[CitiesViewController
        alloc] initWithStyle:UITableViewStylePlain];
    NSString *selectName = [self.listData objectAtIndex:row];
    citiesViewController.listData = [self.dictData objectForKey:selectName];
    citiesViewController.title = selectName;
    [self.navigationController pushViewController:citiesViewController
        animated:YES];
}

```

①

表视图委托方法tableView:didSelectRowAtIndexPath:的实现前面没有介绍过,它是选择表视图单元格时触发的方法。使用该方法,可以进入下一级视图。导航控制器用于将下一级视图控制器压入到栈中,处于栈顶的视图控制器就是当前视图控制器。如果要进入下一级视图,就将下一级视图控制器压栈,第①行的pushViewController:animated:方法就是实现这一目的的。回到上级视图,可以使用出栈方法。共有如下3个出栈方法。

- ❑ **popViewControllerAnimated:**。回到上一级视图。
- ❑ **popToRootViewControllerAnimated:**。回到根视图。
- ❑ **popToViewController:animated:**。回到指定视图。

ViewController.m中的其他方法以前都介绍过，这里不再详述。下面看一下二级视图控制器CitiesViewController.h的代码：

```
@interface CitiesViewController : UITableViewController

@property (weak, nonatomic) NSArray *listData;

@end
```

虽然都显示表视图，但与ViewController不同的是，CitiesViewController继承了UITableViewController类，这样可以避免很多麻烦，例如可以不必声明实现表视图控制器的两个协议，也不需要设置输出的连线，连nib文件也不需要。CitiesViewController.h定义的listData属性用来接收上一个视图控制器传递过来的参数。CitiesViewController.m中的主要方法如下：

```
#pragma mark - 实现表视图数据源方法
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection:
    (NSInteger)section
{
    return [self.listData count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    }

    NSInteger row = [indexPath row];
    NSDictionary *dict = [self.listData objectAtIndex:row];

    cell.textLabel.text = [dict objectForKey:@"name"];
    cell.accessoryType = UITableViewCellAccessoryDetailDisclosureButton;

    return cell;
}

#pragma mark - 实现表视图委托方法
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
    (NSIndexPath *)indexPath
{
    NSInteger row = [indexPath row];

    DetailViewController *detailViewController = [[DetailViewController alloc]
        initWithNibName:@"DetailViewController" bundle:nil];

    NSDictionary *dict = [self.listData objectAtIndex:row];

    detailViewController.url = [dict objectForKey:@"url"];
    NSString *name = [dict objectForKey:@"name"];
    detailViewController.title = name;
}
```

```
[self.navigationController pushViewController:detailViewController
    animated:YES];
```

```
}
```

下面看一下详细视图控制器DetailViewController.h的代码:

```
@interface DetailViewController : UIViewController <UIWebViewDelegate>

@property (weak, nonatomic) IBOutlet UIWebView *webView;

@property (weak, nonatomic) NSString *url;

@end
```

此外,还需要在DetailViewController中实现UIWebViewDelegate协议,webView是UIWebView类型的属性,并定义为输出口,对应于nib文件中的WebView控件。url属性是接收上一个视图控制器传递过来的参数,这里是选中城市的百度百科网址。

详细视图控制器DetailViewController.m的代码如下:

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

    self.webView.delegate = self;
    NSURL * url = [NSURL URLWithString: self.url];
    NSURLRequest * request = [NSURLRequest requestWithURL:url];
    [self.webView loadRequest:request];
}

#pragma mark WebView 委托
#pragma mark --
- (void)webViewDidFinishLoad: (UIWebView *) webView {
    NSLog(@"finish");
}
- (void)webView: (UIWebView *)webView didFailLoadWithError: (NSError *)error
{
    NSLog(@"%@", [error description]);
}
```

当视图加载时,就开始请求传递过来的网址,请求成功时,回调webViewDidFinishLoad:方法,请求失败的情况下回调webView: didFailLoadWithError:方法。

代码编写完成,此时运行一下。得到的效果如图6-48所示。



图6-48 运行效果

6.4.3 故事板实现

在Xcode 4.5中创建树形结构导航的应用，可以使用工程模板Master-Detail Application。

在树形结构导航中，采用故事板实现和采用nib实现有很大的差别。

使用Single View Application模板并利用故事板技术创建一个名为TreeNavigationStoryboard的工程。先创建二级视图控制器CitiesViewController和三级视图控制器DetailViewController，其中创建DetailViewController时不要选择With XIB for user interface复选框，其过程可以参考nib实现。

打开MainStoryboard.storyboard文件，然后从对象库中拖曳一个Table View到View上面，如图6-49所示，并为其定义输出连线。

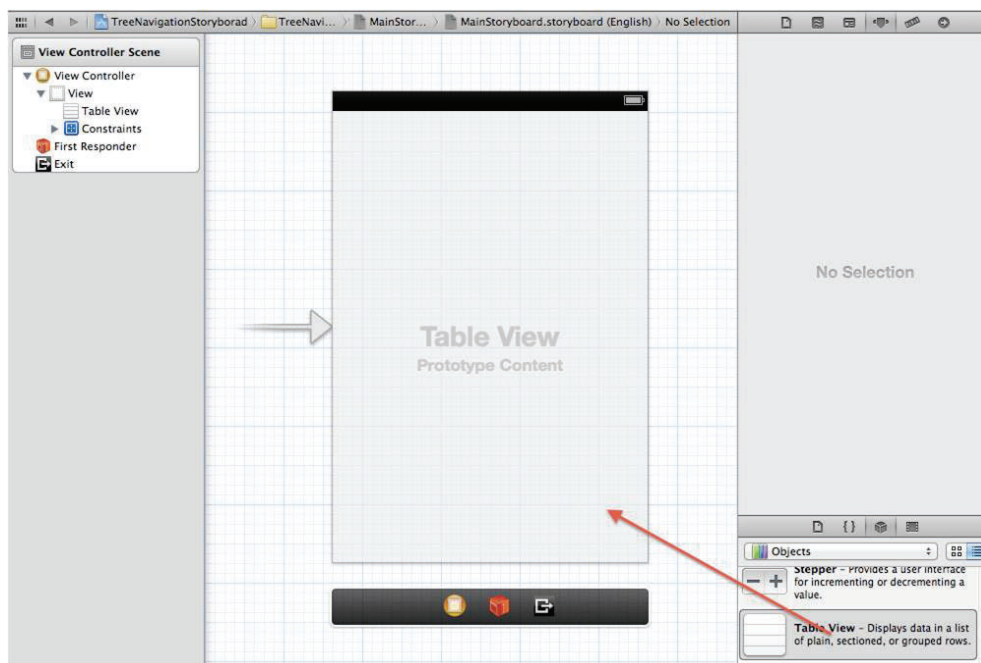



图6-49 拖曳Table View到View上面

选择View Controller Scene中的Table View，打开其属性检查器，将Prototype Cells属性设置为1，将Style属性设置为Plain，如图6-50所示。

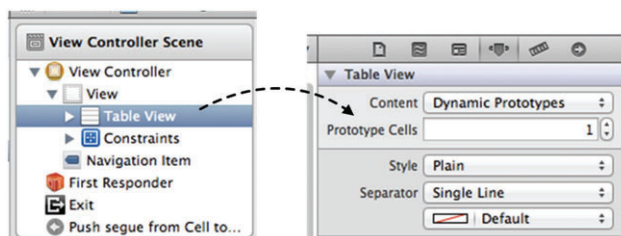


图6-50 设置Table View的属性

选择View Controller Scene中的Table View Cell，打开其属性检查器，将Identifier属性设置为Cell，将Accessory设置为Disclosure Indicator，如图6-51所示。

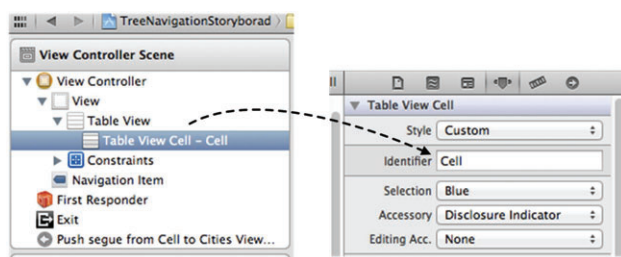


图6-51 设置Table View Cell的属性

由于我们选择的工程模板不是导航控制器模板，所以需要自己添加导航控制器并将其作为根视图控制器。选中View Controller，然后打开菜单Editor→Embed In→Navigation Controller，得到的界面如图6-52所示。

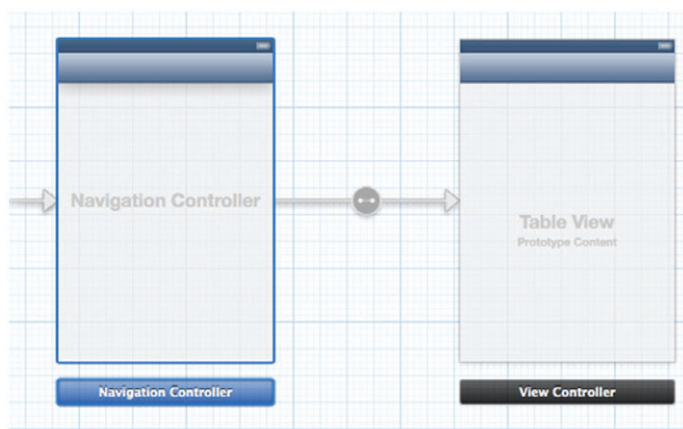


图6-52 添加导航控制器

从对象库中拖曳一个Table View Controller对象到Interface Builder设计界面，作为二级视图控制器。然后按住control键从上一个Table View Controller拖动鼠标到当前添加的Table View Controller，从弹出界面中选择push，然后就会出现两个控制器的连线，如图6-53所示。

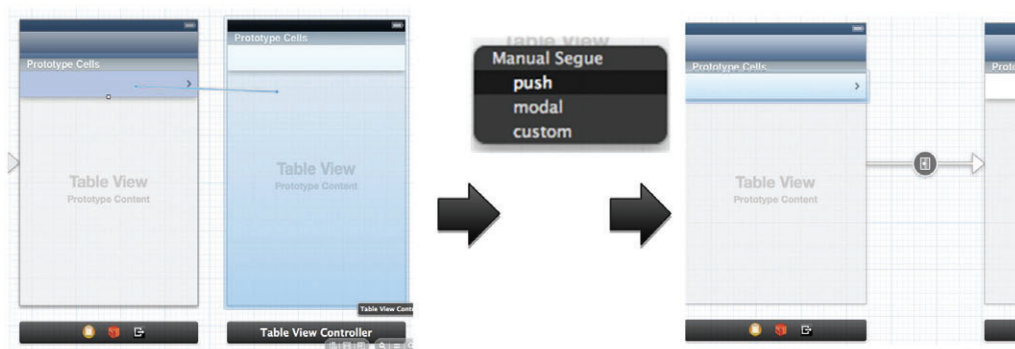


图6-53 两个视图控制器的连线

选中连线中间的Segue，打开其属性检查器，然后在Identifier属性中输入ShowSelectedProvince，如图6-54所示。这个Identifier属性将在代码中用于查询Segue对象。

选择Table View Controller，打开其标识检查器，在Custom Class的Class下拉列表中选择CitiesViewController，如图6-55所示。

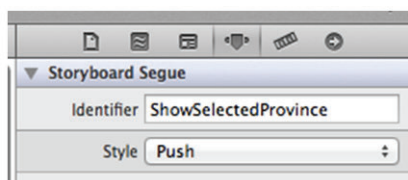


图6-54 设置Segue的Identifier属性

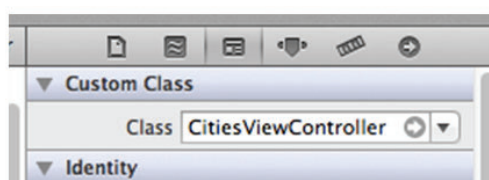


图6-55 设置Table View Controller的Class属性

选择Cities View Controller Scene中的Table View，打开其属性检查器，将Prototype Cells属性设置为1，将Style设置为Plain。选中Table View Cell，打开其属性检查器，将Identifier属性设置为Cell，选择Accessory为Detail Disclosure。

从对象库中拖曳一个View Controller对象到Interface Builder设计界面，作为三级视图控制器。然后按住control键将鼠标从上一个Table View Controller拖动到当前添加的View Controller，此时从弹出界面中选择push，此时就会出现两个控制器的连线。选中连线中间的Segue，打开其属性检查器，在Identifier属性中输入ShowSelectedCity。选择View Controller，打开其标识检查器，点击Custom Class→Class，将其设置为DetailViewController。最后，拖曳一个WebView控件到View上面，并为WebView连接输出口。

到此，烦琐的设计工作就完成了，下面我们看一下代码部分。与nib实现方式不同，在故事板中AppDelegate部分的代码不需要修改。ViewController.h的相关代码如下：

```
@interface ViewController : UIViewController <UITableViewDataSource, UITableViewDelegate>

@property (weak, nonatomic) IBOutlet UITableView *tableView;

@property (strong, nonatomic) NSDictionary *dictData;
@property (strong, nonatomic) NSArray *listData;

@end

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.tableView.delegate = self;
    self.tableView.dataSource = self;

    NSBundle *bundle = [NSBundle mainBundle];
    NSString *path = [bundle pathForResource:@"provinces_cities" ofType:@"plist"];

    self.dictData = [[NSDictionary alloc] initWithContentsOfFile:path];
    self.listData = [self.dictData allKeys];
    self.title = @"城市信息";
}

#pragma mark - 实现表视图数据源方法
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection:(NSInteger)section
{
    return [self.listData count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier forIndexPath:indexPath];
```

```

    NSInteger row = [indexPath row];
    cell.textLabel.text = [self.listData objectAtIndex:row];
    return cell;
}

//选择表视图行时触发
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    if([segue.identifier isEqualToString:@"ShowSelectedProvince"])
    {
        CitiesViewController *citiesViewController =
            segue.destinationViewController;
        NSInteger selectedIndex = [[self.tableView indexPathForSelectedRow] row];
        NSString *selectName = [self.listData objectAtIndex:selectedIndex];
        citiesViewController.listData = [self.dictData objectForKey:selectName];
        citiesViewController.title = selectName;
    }
}

```

需要注意一下，在tableView:cellForRowAtIndexPath:方法中，查找可重用单元格的方法dequeueReusableCellWithIdentifier:forIndexPath:比iOS 6之前的方法dequeueReusableCellWithIdentifier:多出forIndexPath:部分，意思是可以指定indexPath。

原来的表视图委托方法tableView:didSelectRowAtIndexPath:被替换为prepareForSegue:sender:方法。prepareForSegue:sender:方法是专门供故事板使用的方法，它是UIViewController中的方法。当两个视图跳转的时候，连接两个视图的Segue就会触发该方法。segue.destinationViewController属性用于获得要跳转到的视图控制器对象。

二级视图控制器CitiesViewController.m与nib代码的不同部分如下：

```

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier forIndexPath:indexPath];

    NSInteger row = [indexPath row];
    NSDictionary *dict = [self.listData objectAtIndex:row];

    cell.textLabel.text = [dict objectForKey:@"name"];

    return cell;
}

//选择表视图行时触发
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    if([segue.identifier isEqualToString:@"ShowSelectedCity"])
    {
        DetailViewController *detailViewController = segue.destinationViewController;

        NSInteger selectedIndex = [[self.tableView indexPathForSelectedRow] row];
        NSDictionary *dict = [self.listData objectAtIndex:selectedIndex];

        detailViewController.url = [dict objectForKey:@"url"];

        NSString *name = [dict objectForKey:@"name"];
        detailViewController.title = name;
    }
}

```

三级视图控制器DetailViewController与nib代码完全一样。代码编写完成后，我们可以运行一下，看看效果是不是与使用nib实现的一样。

6.5 组合使用导航模式

有些情况下，我们会将3种导航模式综合到一起使用，其中还会用到模态视图。例如，Tweet是编写Twitter的应用，如图6-56所示。Tweet主要采用了标签导航模式和树形结构导航模式，有些地方（右边图片中的Bill Couch）还采用了平铺导航模式。点击导航栏右边的按钮，会打开一个模态视图，可以编辑Twitter。



图6-56 Tweet应用

6.5.1 应用场景

同样是划分东北三省的城市信息，我们可以采用组合方式实现，如图6-57所示。标签栏上是省名，标签导航可以进行省的切换。省信息中又采用树形结构导航，只不过树形结构中只有两级视图，二级视图（城市信息）导航栏右边的按钮可以实现添加城市信息的功能。



图6-57 组合导航模式

6.5.2 故事板实现

下面我们介绍用故事板实现组合导航模式。为了能够进一步了解故事板的原理，我们选择Xcode的Empty Application工程模板，该工程模板内只有应用程序委托对象。使用Empty Application创建一个名为NavigationComb的工程，注意创建过程中不要选中故事板选项。

1. 添加故事板文件

鉴于在创建过程中没有选中故事板选项，我们要自己添加故事板文件。选择File→New→File...菜单项，在打开的Choose a template for your new file对话框中选择User Interface→Storyboard，如图6-58所示。点击Next按钮，从弹出的对话框中选择Device Family为iPhone，将文件命名为Storyboard.storyboard。

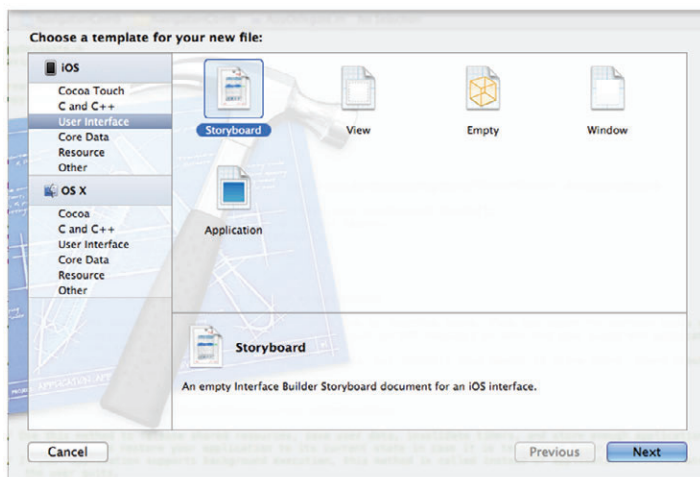


图6-58 添加故事板文件

进入工程，将TARGETS→NavigationComb中的Main Storyboard设置为Storyboard，如图6-59所示，这一步操作是为应用选择主故事板。

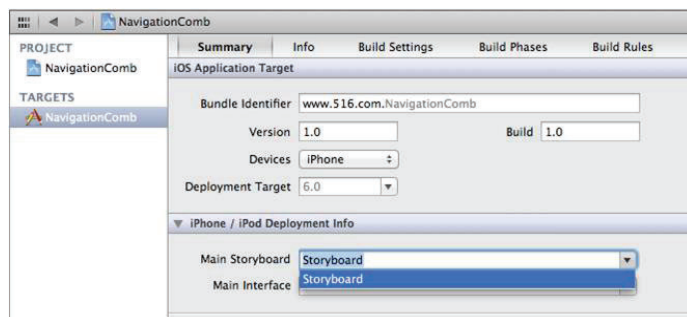



图6-59 选择主故事板

2. 创建视图控制器

我们需要预先创建好3个视图控制器文件：一级视图控制器FirstViewController、二级视图控制器DetailViewController和模态视图控制器ModalViewController，其中FirstViewController的父类是UITableViewController，DetailViewController和ModalViewController的父类都是UIViewController。在它们的创建过程中，都不要选中With XIB for user interface复选框。

3. 设计一级视图控制器场景

从对象库中拖曳一个Table View Controller到故事板设计界面，选择Table View Controller Scene→ Hei-FirstViewController，打开其标识检查器，将Class属性设置为FirstViewController，Label属性设置为Hei-FirstViewController，如图6-60所示，其中Label属性的作用是便于在故事板中查看这个Sence。

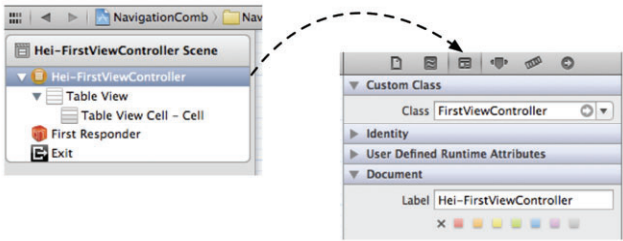


图6-60 设置FirstViewController

选中Hei-FirstViewController Scene，再复制出两个，然后将它们的Label属性分别修改为Ji-FirstViewController和Liao-FirstViewController，如图6-61所示。

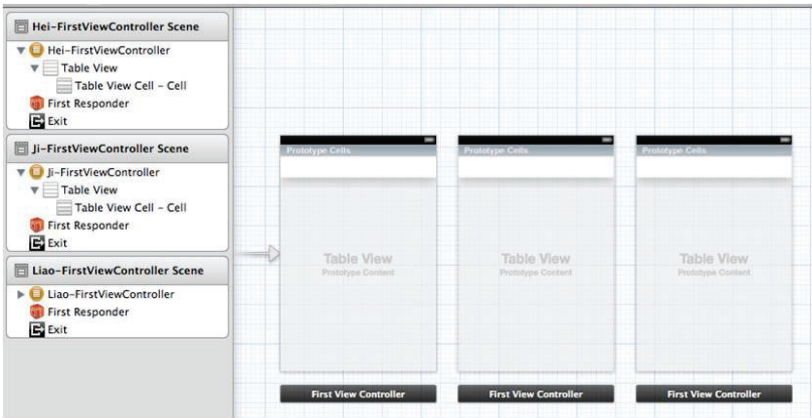


图6-61 复制场景

4.设计二级视图控制器场景

从对象库中拖曳一个View Controller到故事板设计界面,选择Table View Controller Scene→Hei-FirstViewController，打开其标识检查器，将Class属性设置为DetailViewController，将Label属性设置为Hei-DetailViewController，如图6-62所示。

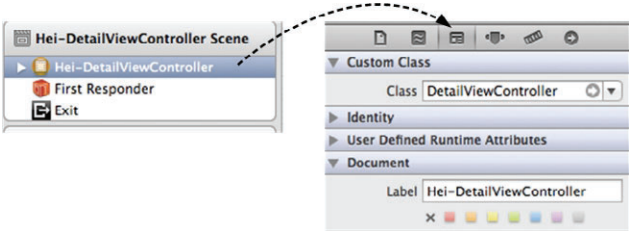


图6-62 设置标识检查器

接着从对象库中拖曳一个WebView控件到DetailViewController的View界面中，并为其定义输出口连线。

选择Hei-DetailViewController Scene, 再复制出两个, 然后再将它们的Label属性分别修改为: Ji-DetailViewController和Liao-DetailViewController。

5. 设计模态视图控制器场景

从对象库中拖曳一个View Controller到故事板设计界面, 选择View Controller Scene并打开其标识检查器, 将Class属性设置为ModalViewController。按照图6-63设计模态视图, 具体操作方法为: 从对象库中拖曳一个TextView和一个导航栏到视图上, 将导航栏标题修改为“添加信息”。另外, 我们要为导航栏设置左右两个按钮: 左按钮选择iOS API提供的Done按钮, 并为其定义动作事件连线; 右按钮选择iOS API提供的Save按钮, 本例中没有对该按钮编写代码, 因此不要为其定义动作事件连线, 读者可以根据自己的需要来实现保存功能。

6. 添加标签控制器

接下来, 我们要连线这些场景。选中Hei-FirstViewController Sence, 然后打开Editor→Embed In→Tab bar Controller菜单, 如图6-64所示。然后按住control键将鼠标从上一个Tab bar Controller拖动到Ji-FirstViewController, 在弹出的对话框中选择view controllers。再以同样的方式连接Tab bar Controller到Liao-FirstViewController。

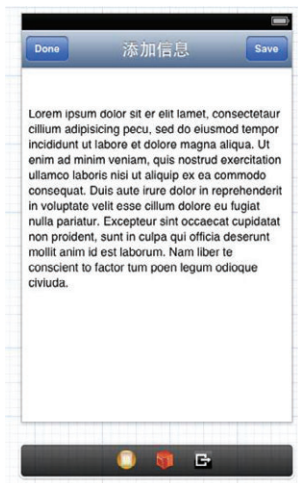


图6-63 模态视图

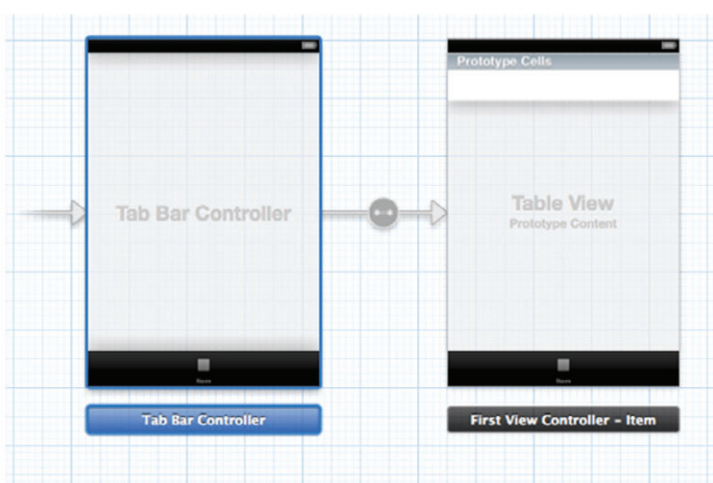


图6-64 标签控制器

选择Hei-FirstViewController Scene中的Tab bar Item, 打开其属性检查器中的Bar Item, 将其Title设置为“黑龙江”, Image设置为hei.png, 如图6-65所示。用同样方法设定Ji-FirstViewController Scene和Liao-FirstViewController Scene中的Tab bar Item属性。

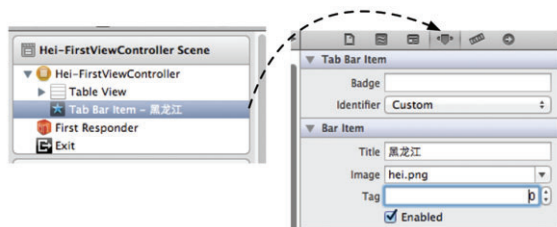


图6-65 Tab bar Item属性检查器

7. 添加导航控制器

选中Hei-FirstViewController Sence, 然后打开菜单Editor→Embed In→Navigation Controller, 如图6-66所示。用同样的方法为Ji-FirstViewController Scene和Liao-FirstViewController Scene添加导航控制器。

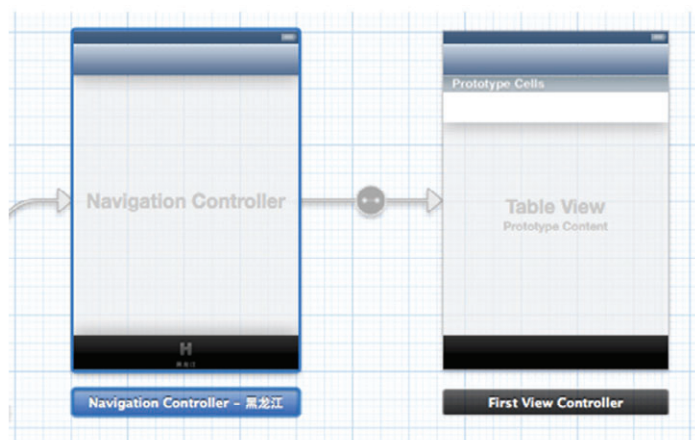


图6-66 导航控制器

选中Hei-FirstViewController Scene→Table View Cell - Cell，然后按住control键将鼠标拖动到Hei-DetailViewController，在弹出界面中选择push。选中它们之间的Segue，打开其属性检查器，将Identifier属性设置为ShowDetail，如图6-67所示。

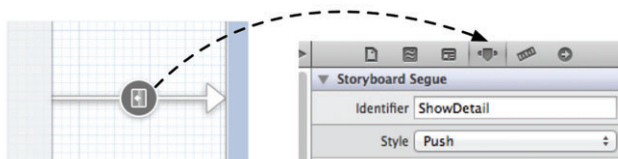


图6-67 设定Segue的Identifier属性

选中Hei-FirstViewController，为它的导航栏添加右按钮，选择按钮的Identifier属性为 Add，如图6-68所示。

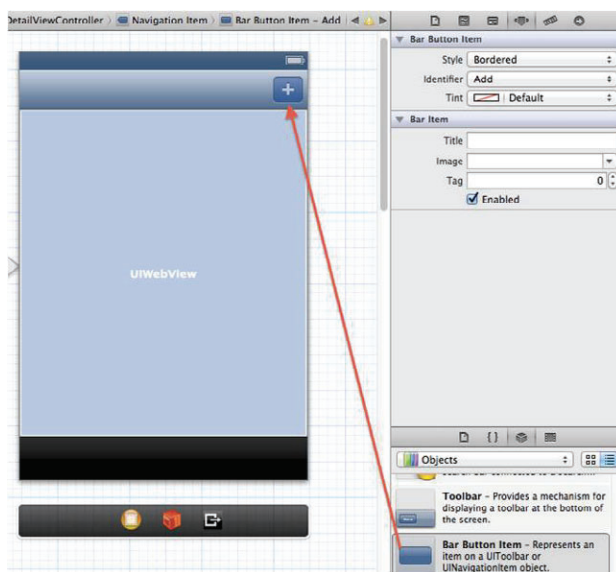


图6-68 为导航栏添加右按钮

按照上面的同样步骤设置Ji-FirstViewController Scene和Liao-FirstViewController Scene。

8. 连接模态视图控制器

选中Hei-FirstViewController Sence中导航栏的右按钮，然后按住control键拖动鼠标到ModalViewController Sence，从弹出界面中选择modal，如图6-69所示。用同样的方法分别连接Ji-FirstViewController Scene和Liao-FirstViewController Scene到ModalViewController Sence。

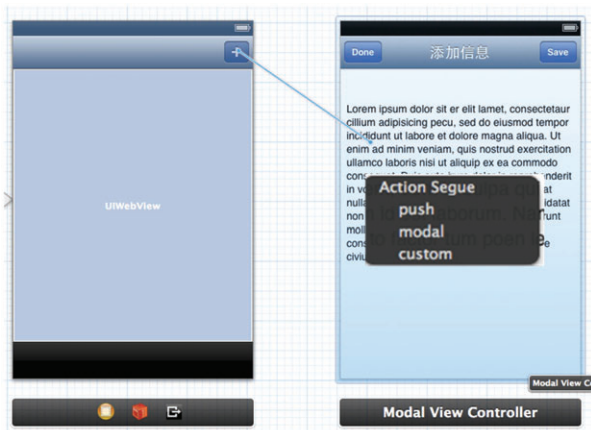


图6-69 连接模态视图控制器

到此为止，我们故事板的设计工作就完成了。完成之后的故事板如图6-70所示，其中有11个场景，很复杂吧！我们的业务还不是很复杂，就已经有这么多的Sence和Segue，故事板的用意是想减少代码量，但是与此同时也增加了设置环节的工作量，如果这些设置出了问题，目前还无法调试。虽然苹果主推使用故事板技术，但它并不是iOS解决编程问题的银弹^①。

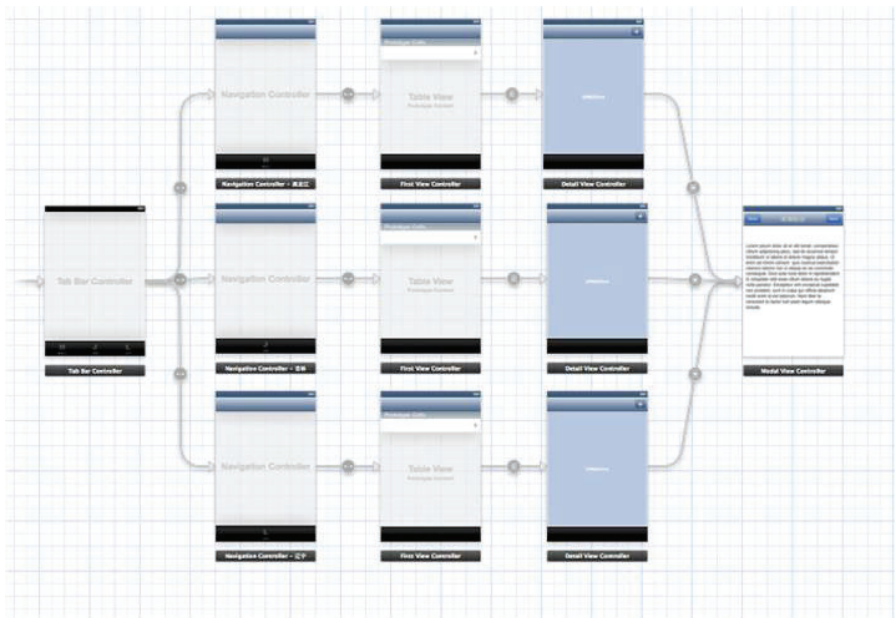


图6-70 完成的故事板

^① 在西方古老的传说里，狼人是不会死的，但是银弹可以杀死狼人，详情可参见<http://baike.baidu.com/view/3413847.htm>。

下面看一下代码部分。AppDelegate.m文件中需要修改应用的启动方法，原有的代码只留下return YES，这是因为window设置已经在故事板中设置好了：

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

一级视图控制器FirstViewController.h的代码如下：

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

@interface FirstViewController : UITableViewController

@property (strong, nonatomic) NSDictionary *dictData;
@property (strong, nonatomic) NSArray *listData;

@end
```

一级视图控制器FirstViewController.m中的视图加载代码如下：

```
@implementation FirstViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    NSBundle *bundle = [NSBundle mainBundle];
    NSString *path = [bundle pathForResource:@"provinces_cities" ofType:@"plist"];

    self.dictData = [[NSDictionary alloc] initWithContentsOfFile:path];
    UINavigationController *navigationController =
        (UINavigationController*)self.parentViewController;
    NSString *selectProvinces = navigationController.tabBarItem.title;

    NSLog(@"%@", selectProvinces);

    if ([selectProvinces isEqualToString:@"黑龙江"]) {
        self.listData = [self.dictData objectForKey:@"黑龙江省"];
        self.navigationItem.title = @"黑龙江省信息";
    } else if ([selectProvinces isEqualToString:@"吉林"]) {
        self.listData = [self.dictData objectForKey:@"吉林省"];
        self.navigationItem.title = @"吉林省信息";
    } else {
        self.listData = [self.dictData objectForKey:@"辽宁省"];
        self.navigationItem.title = @"辽宁省信息";
    }
}
```

FirstViewController是3个省共同使用的类，当然我们可以为每一个省创建一个视图控制器，但是就本例而言，完全没有必要创建3个不同的类。如何区分是点击了哪个标签进入的呢？下面的这两条语句可以获得选中的标签栏的标签名字，通过这个标签名字能够识别是点击哪个标签进入的：

```
UINavigationController *navigationController =
    (UINavigationController*)self.parentViewController;
NSString *selectProvinces = navigationController.tabBarItem.title;
```

在FirstViewController.m中，选择表视图行时会触发Segue方法：

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
```

```
if([segue.identifier isEqualToString:@"ShowDetail"]){
    DetailViewController *detailViewController = segue.destinationViewController;
    NSInteger selectedIndex = [[self.tableView indexPathForSelectedRow] row];
    NSDictionary *dict = [self.listData objectAtIndex:selectedIndex];
    detailViewController.url = [dict objectForKey:@"url"];
    detailViewController.title = [dict objectForKey:@"name"];
}
```

其他视图控制器与上一节中的案例是一样的，没有变化，此处就不再介绍了。代码编写完毕后，运行一下，效果如图6-71所示。



图6-71 运行效果

6.6 小结

通过本章的学习，你已经可以判断你的应用是不是需要一个导航功能，并且知道在什么情况下选择平铺导航、标签导航、树形结构导航，或者同时综合使用这3种导航模式。针对标签导航和树形导航这两种相对复杂的导航模式，本章为大家提供了nib和故事板两种实现方式，大家可以通过nib实现方式掌握导航模式的原理，同时通过故事板实现的方式掌握如何灵活使用这两种导航方式。

iPhone与iPad应用开发的差异



对于有些应用，我们会想在iPhone^①和iPad这两个平台上运行，但简单地把iPhone程序拿过来直接放在iPad上，那是不能运行的。由于它们尺寸不同，应用场景也不同，因此在导航模式上有比较大的差异，一些控件在展现方式上会有很多差别，所以iPad有一些特有的API。

7.1 概述

首先，我们回顾一下iPhone和iPad的几个参数：iPhone 4的屏幕尺寸为3.5英寸，分辨率为960 × 640像素；iPhone 5的屏幕尺寸为4英寸，分辨率为1136 × 640像素；iPad 1和iPad 2的屏幕尺寸是9.7英寸，分辨率为1024 × 768像素。屏幕尺寸的不同导致了应用场景的不同，应用场景的不同直接导致了设计和开发的不同。

7.1.1 应用场景差异

作为iOS开发者，我们应该熟悉iPhone和iPad应用的场景，然后才能开发出好的应用。iPhone是让用户一只手使用的设备，因此它适合在等车时拿出来看看天气、收发邮件、看看周围有哪些银行或者饭店，等等。而iPad是两只手使用的设备，它不太适合处理iPhone用户的场景。据调查，iPad多数用在家里，用来浏览网页、收发电子邮件、看照片、看视频、听音乐、玩电子游戏和看电子书等。作为平板电脑，它比笔记本电脑更轻便、更适合移动使用。

基于应用场景的不同，同样一款应用在iPhone和iPad上的功能选取和界面布局有着明显的不同。有些应用只能做成iPhone版本的，有些应用只能做成iPad版本的。与iPhone用户相比，iPad用户更期待具有高保真的、艺术品般的、高品质的应用，而绝非简单地放大iPhone应用的屏幕尺寸。

7.1.2 设计和开发需注意的问题

在设计和开发时，需要注意的有如下几个方面：API、导航模式和设计层面。

1. API

iPhone和iPad都使用一个操作系统——iOS，因此，它们的API基本上是一样的，但有一些是iPad专用的，比如UIPopoverController控制器和UISplitViewController控制器，其中UIPopoverController控制器用于呈现“漂浮”类型的视图，UISplitViewController控制器用于将屏幕分栏。这两个控制器在E-mail应用中都用过。

^① 本章中所说的iPhone设备包括了iPod touch。

2. 导航模式

在上一章中，我们介绍了iPhone的3种导航模式。在iPad中，平铺导航模式和标签导航模式与iPhone基本一样，但树形结构导航模式与iPhone差别比较大。下面我们以E-mail应用为例介绍一下iPad的树形结构导航模式。

图7-1是iPhone的E-mail应用界面，它不支持横屏，导航采用了树形结构导航模式，新邮件编辑采用模式视图导航模式。



图7-1 iPhone的E-mail应用界面

iPad横屏时，其E-mail应用如图7-2所示。对比可以发现，iPhone版导航分成两个屏幕，而iPad版采用一个屏幕分成左右两栏，左栏是用于导航的菜单，占用固定的320点，右栏是详细内容。

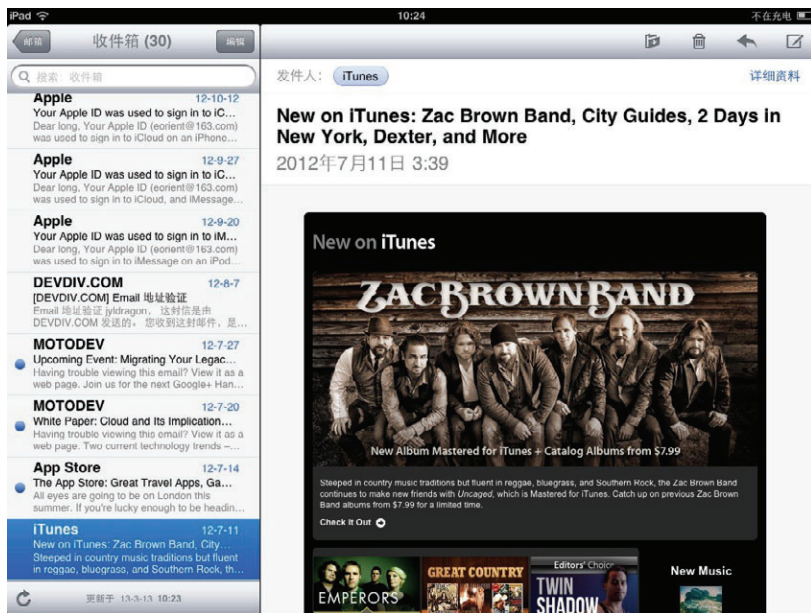


图7-2 横屏时iPad的E-mail应用界面

iPad竖屏时，其E-mail应用如图7-3所示，默认只显示详细内容。左边的导航栏是隐藏的，需要时点击左上角的“收件箱”按钮，它会以Popover方式显示出来。

此外，两个设备上的模式视图导航也是不同的。图7-4是iPhone的模式视图界面，默认情况下会从屏幕下方滑出，占有整个屏幕。

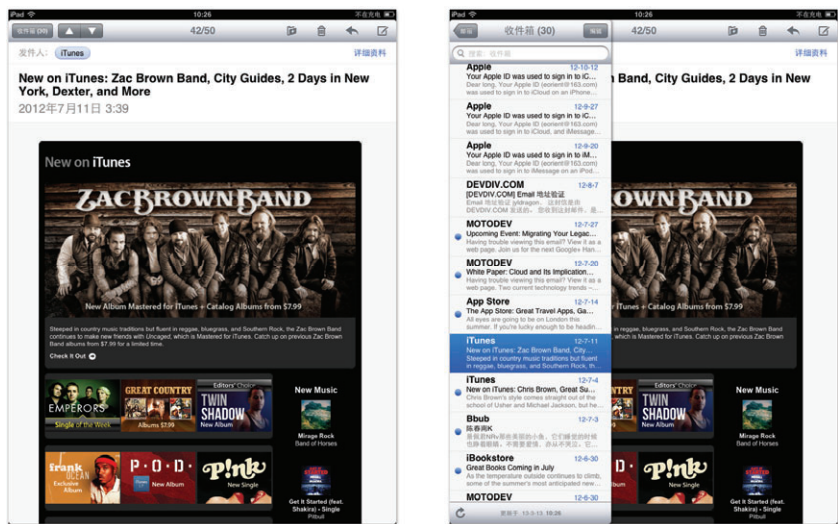


图7-3 竖屏时iPad的E-mail应用界面



图7-4 iPhone模式视图

图7-5是iPad的横屏模式视图界面，默认情况下会从屏幕下方滑出，显示在屏幕中间。图7-6是iPad的竖屏模式视图界面，默认情况下会从屏幕下方滑出，占有整个屏幕。



图7-5 iPad的横屏模式视图



图7-6 iPad的竖屏模式视图

3. 设计层面

如果一个应用同时要开发iPhone版和iPad版，需要怎么设计它呢？软件设计的基本原则是：代码可复用性和可扩展性。事实上，iPhone和iPad的区别在于表示层（或展示层），其他层是一样的。例如，你要做一个微博应用，iPhone和iPad版本的差别只在于界面和导航的不同，而网络通信和数据持久化应该是一样的，它们应该封装在一

个层之中。因此，iOS应用要考虑分层设计问题。我们会在下一节中详细讨论设计层面的问题。

提示 从软件系统架构上讲，应用系统可以“分层设计”，“层”是具有相似职责功能的一组类的集合。例如，表示层是与用户交互、展示信息、接收用户请求的层，由一些UI类组成。在iOS系统中，表示层由UIView及其子类、UIViewController及其子类等构成。

7.1.3 构建自适应的iPhone和iPad工程

有时候，应用需要能够在iPhone和iPad上运行，这时我们可以选择做两个完全不同的工程，然后共用一些类，也可以只创建一个工程，然后编译生成一个产品。这个产品能够自适应iPhone和iPad设备，在App Store发布时，会有两套不同的截图展示给用户，比如图7-7，它展示了App Store上的Evernote应用。

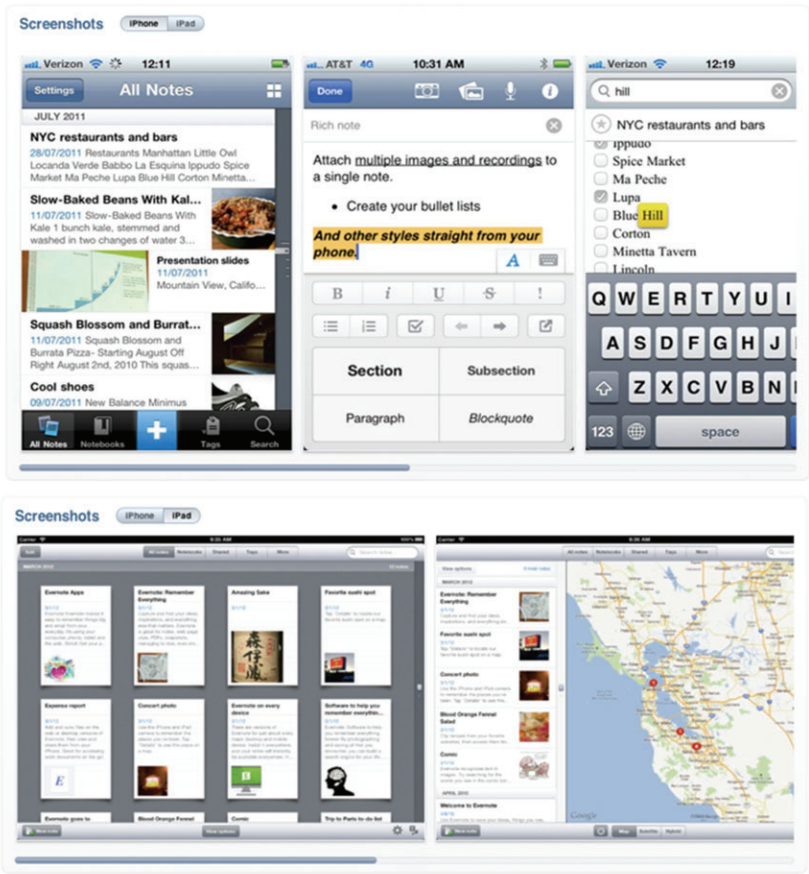


图7-7 App Store上的Evernote应用

下面我们来做一个名为UniversalSample的自适应iPhone和iPad设备的工程。

首先，在创建时选择Devices为Universal，如图7-8所示。我们以前一直选择的是iPhone，如果开发iPad专用的应用，需要勾选iPad。

创建好后，就会生成有两套故事板，如图7-9所示。用nib实现时，也会有两套文件。

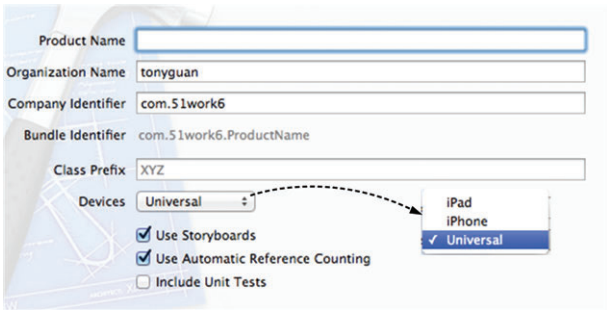


图7-8 选择通用设备

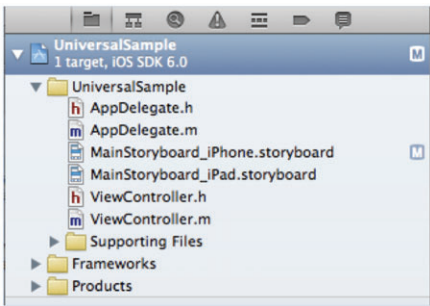


图7-9 两套故事板

选择TARGETS，可以发现，iPhone的Main Storyboard（主故事板）设置的是MainStoryboard_iPhone，如图7-10所示。主故事板在应用启动时首先被加载，它设定应用的根视图和启动的第一个界面。图7-11是iPad的主故事板，其中Main Storyboard设置的是MainStoryboard_iPad。

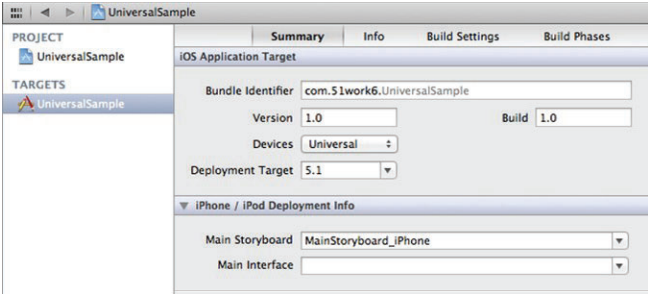


图7-10 iPhone中的主故事板

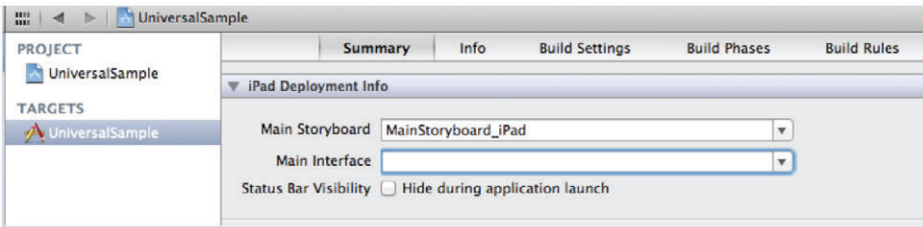


图7-11 iPad中的主故事板

运行时需要选择运行设备，有iPad 6.0 Simulator、iPhone 6.0 Simulator和iOS Device（真机运行）这3个选项，如图7-12所示，这里我们选择iPhone 6.0 Simulator，说明应用要在iPhone 6.0模拟器上运行。



图7-12 选择运行设备

在编程时可以通过下面语句判断设备，其中常量 `UIUserInterfaceIdiomPhone` 用于判断是否为 iPhone 设备，`UIUserInterfaceIdiomPad` 用于判断是否为 iPad 设备：

```
if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPhone) {
    //iPhone设备
} else {
    //iPad设备
}
```

7.2 iPad 专用 API

前面提到过，`UIPopoverController` 控制器和 `UISplitViewController` 控制器是 iPad 专用的视图控制类。此外，在模态视图呈现的时候，对于 iPad 不同的呈现样式，苹果也给出了一些特别的属性设置。

7.2.1 UIPopoverController 控制器

`UIPopoverController` 是 iPad 特有的类，不能在 iPhone 上使用，它负责控制 Popover 视图。Popover 视图是一种临时视图，它以“漂浮”的形式出现在视图表面，如图 7-13 所示。触摸 Popover 视图的外边，则关闭视图。

由于 Popover 视图不会占用全屏，而且有一个箭头指向其他视图或按钮，所以 Popover 内容视图中也常常包含一些控件，类似表单一样。图 7-14 是 iPad 中 Safari 浏览器的打印机选项。



图7-13 Popover视图



图7-14 Safari浏览器的打印机选项

iOS API 提供了 `UIPopoverController` 和 `UIPopoverControllerDelegate`，但没有提供与 `UIPopoverController` 对应的视图类。`UIPopoverController` 类的常用方法和属性如下所示。

- ❑ **setContentViewController:animated:**。设定内容视图大小的方法。
- ❑ **presentPopoverFromRect:inView:permittedArrowDirections:animated:**。指定一个矩形区域的位置作为锚点来呈现 Popover 视图的方法。
- ❑ **presentPopoverFromBarButtonItem:permittedArrowDirections:animated:**。指定一个按钮作为锚点来呈现 Popover 视图的方法。
- ❑ **dismissPopoverAnimated:**。关闭 Popover 视图的方法。
- ❑ **popoverVisible**。判断 Popover 视图是否可见。
- ❑ **popoverArrowDirection**。判断 Popover 视图箭头的方向。

Popover 视图可以用故事板连线实现，也可以通过代码实现。使用故事板实现时，不必编写任何代码即可，而使用代码实现比较灵活。下面我们通过一个案例介绍一下 Popover 视图的用法。如图 7-15 所示，在 iPad 界面的导航

栏中，有左右两个按钮，点击左边的Show按钮，会弹出Popover视图（其中可以设置打印机的相关项），这是一个Popover表单视图，是通过故事板设定的，不用编写任何代码。点击右边的Coding Show按钮，会弹出Popover视图，这是一个选择列表，通过代码实现。



图7-15 Popover视图案例

采用Single View Application模板创建PopoverViewSample工程，其中Devices选择iPad，勾选Use Storyboards和Use Automatic Reference Counting复选框。打开MainStoryboard.storyboard文件，设计iPad主界面，从对象库中拖曳导航栏项目以及左右按钮，设计样式如图7-16所示；然后，为右按钮Coding Show定义动作事件连线。



图7-16 iPad主界面

从对象库中拖曳一个新的Table View Controller，将其作为用于设置打印机的Popover视图控制器，如图7-17所示，它是一个静态表视图。再从对象库中拖曳一个新的Table View Controller，将其作为颜色选择Popover视图控制器，如图7-18所示，它是一个动态表视图，接着用尺寸属性检查器来调整它们的大小，参数请参照本案例。



图7-17 用于设置打印机的Popover视图

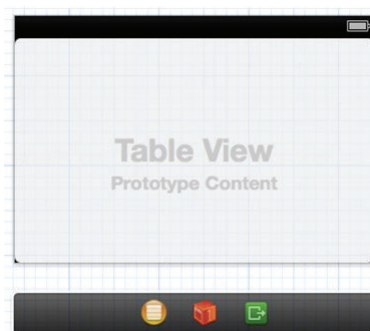


图7-18 用于选择颜色的Popover视图

对于设置打印机的Popover视图，我们不是通过代码而是在故事板中设计的。选中故事板主界面中的左按钮Show，按住control键拖曳设置打印机的Popover视图，此时从弹出界面中选择popover，如图7-19所示。注意，需要设定它的Storyboard ID属性为SelectViewController。

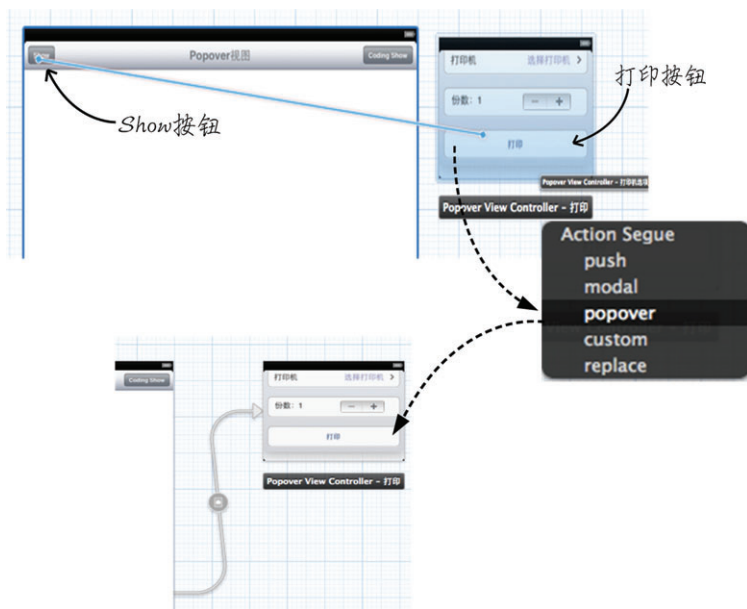


图7-19 使用故事板实现Popover视图

这时我们也可以先运行一下，看看这样不写一行代码是否可以弹出图7-15所示的窗口。对于选择颜色的Popover视图，我们通过代码来实现。首先要为它创建一个控制器SelectViewController，然后再看看ViewController.h的代码如下：

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

@interface ViewController : UIViewController

@property (nonatomic, strong) UIPopoverController *poc;

- (IBAction)show:(id)sender;

@end
```

其中poc是UIPopoverController类型的属性，它负责保存一个UIPopoverController对象。show:方法用于响应右按钮Coding Show的点击事件。ViewController.m的代码如下：

```
- (IBAction)show:(id)sender {

    SelectViewController *popoverViewController = [self.storyboard
        instantiateViewControllerWithIdentifier:@"SelectViewController"];

    if (self.poc == nil) {
        popoverViewController.title = @"选择你喜欢的颜色";
        UINavigationController *nav = [[UINavigationController alloc]
            initWithRootViewController:popoverViewController];
        self.poc = [[UIPopoverController alloc] initWithContentViewController:nav];
    }

    [self.poc presentPopoverFromBarButtonItem:sender
        permittedArrowDirections:UIPopoverArrowDirectionUp
        animated:YES];
}
```


第①行代码进行了if判断，目的是防止多次实例化UIPopoverController。在if语句中，第②行代码用于创建UINavigationController对象，第③行代码把这个UINavigationController对象作为内容视图放入到UIPopoverController中，因此UIPopoverController的内容视图是UINavigationController对象不是SelectViewController对象。这样做的目的是为Popover视图添加标题。没有标题的Popover视图如图7-20所示。



图7-20 没有标题的Popover视图

通常，构建没有标题的Popover视图的代码如下，需要把SelectViewController直接作为UIPopoverController的内容视图就可以了：

```
if (self.poc == nil) {

    popoverViewController.title = @"选择你喜欢的颜色";
    UINavigationController *nav = [[UINavigationController alloc]
    initWithRootViewController:popoverViewController];
    self.poc = [[UIPopoverController alloc]
    initWithContentViewController:popoverViewController];

}
```

SelectViewController是颜色选择视图控制器，它是一个动态表视图控制器，其代码不再介绍。如果需要，读者可以通过本书源代码查看全部代码。

从上面的实现过程可见，Popover视图实现起来比较简单。UIPopoverController还有委托协议UIPopoverControllerDelegate。委托协议UIPopoverControllerDelegate有如下两个方法。

- ❑ **popoverControllerShouldDismissPopover:**。控制是否关闭Popover视图。
- ❑ **popoverControllerDidDismissPopover:**。关闭Popover视图之后触发。

7.2.2 UISplitViewController控制器

前面我们分析过iPad自带的E-mail应用，它采用UISplitViewController控制器，该控制器是iPad中构建导航模式应用的基础，可以呈现屏幕分栏视图的效果。由于iPad要比iPhone大很多，所以不能简单地采用iPhone的导航模式。图7-21是横屏SplitView视图，此时屏幕被分割为左右两个视图，右侧是DetailView，负责显示详细信息，左侧是MasterView，其中有一个导航列表，用于为右侧的DetailView导航。需要说明的是，MasterView的导航列表占有320点的固定大小。在竖屏的情况下，MasterView会隐藏起来，如图7-22所示。

有时候，我们会根据需要在MasterView或DetailView中添加导航栏控制器（UINavigationController），以便在自己的视图中采用树形导航模式。图7-23是iPad自带的E-mail应用，左右视图都带有导航栏。MasterView内部就采用树形导航模式，这样会承载大量的信息而不混乱。

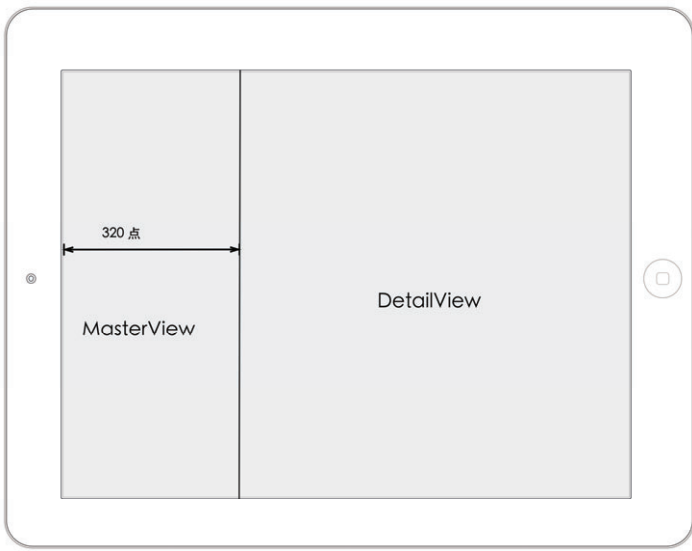


图7-21 横屏SplitView视图

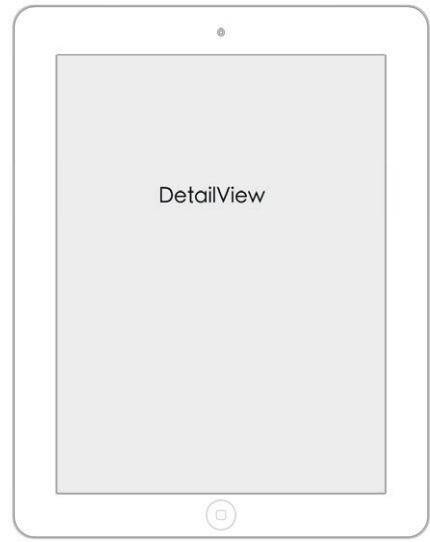


图7-22 竖屏SplitView视图

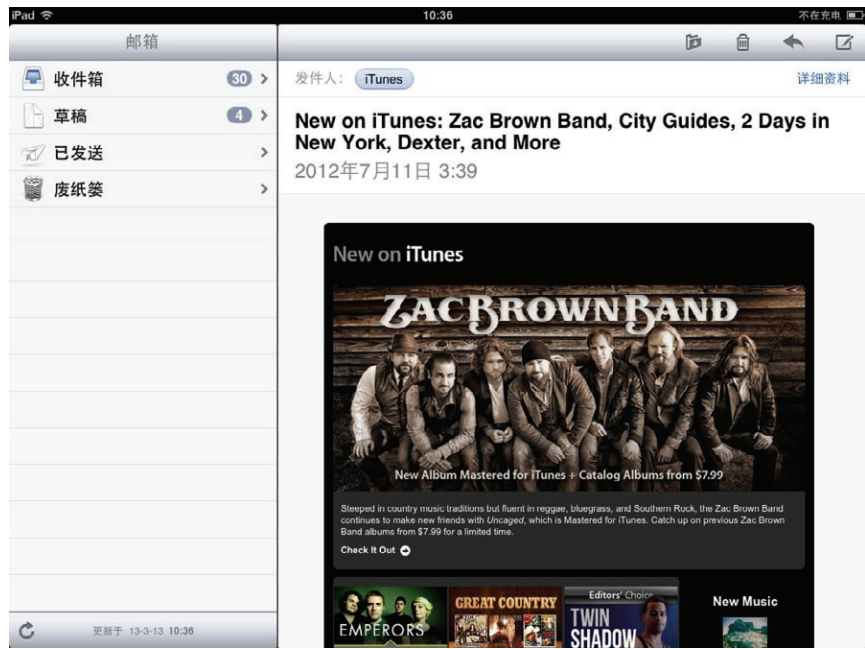


图7-23 iPad自带的E-mail应用

下面我们通过一个案例来熟悉一下UISplitViewController控制器。图7-24是横屏情况的SplitView视图，显示了MasterView和DetailView，其中MasterView中有Blue View和Yellow View选择项目，当选择其中的单元格时，右边的DetailView就会显示相应的蓝色和黄色视图。点击Press按钮，会弹出AlertView警告框提示为蓝色还是黄色视图。

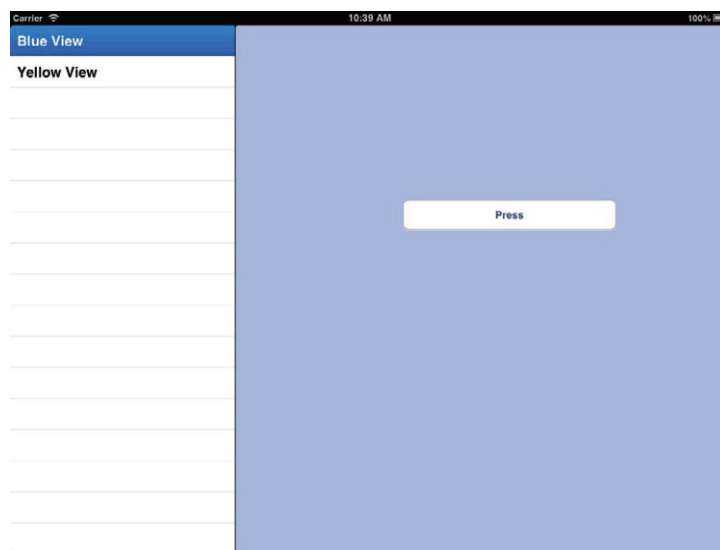


图7-24 SplitView视图案例

使用Xcode创建工程SplitViewSample, 模板采用Single View Application, Devices选择iPad, 选中Use Storyboards和Use Automatic Reference Counting复选框。删除由模板生成的ViewController.h和ViewController.m文件, 然后用Xcode工具创建DetailViewController、BlueViewController和YellowViewController视图控制器, 它们的父类是UIViewController, 不用选择xib文件。再创建MasterViewController视图控制器, 其父类是UITableViewController, 不用选择xib文件。

打开MainStoryboard.storyboard文件, 删除由模板生成的View Controller Scene, 从对象库中拖曳Split View Controller到设计界面, 如图7-25所示。

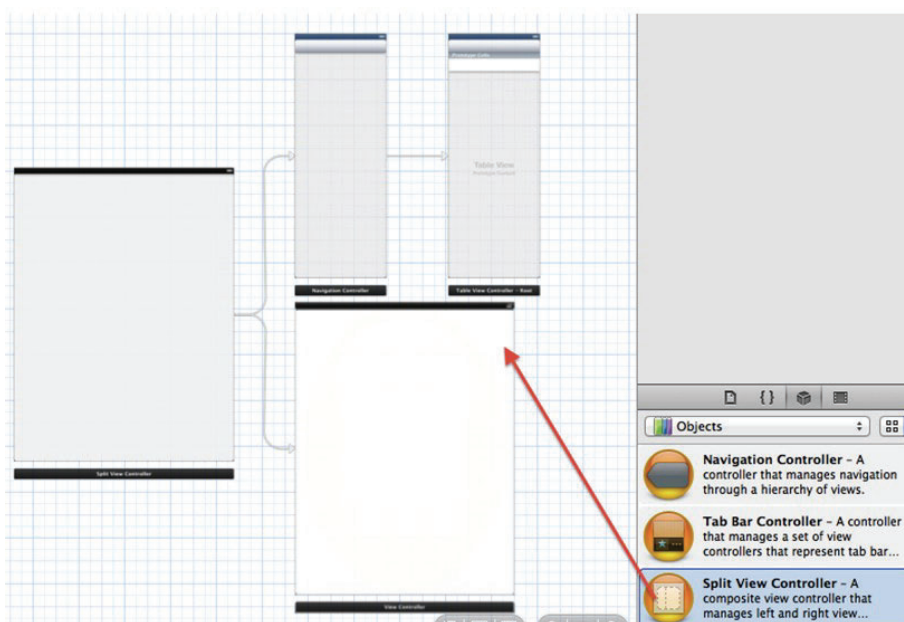


图7-25 从对象库中拖曳Split View Controller到设计界面

从图7-25中可见，共生成了4个视图控制器。默认情况下，DetailView内部采用导航控制器（Navigation Controller）作为它的根视图控制器，MasterView采用普通视图控制器作为它的根视图控制器。选中Split View Controller，打开其属性检查器，将Orientation属性选择为Landscape（横屏），如图7-26所示，这样就可以横屏设计界面了。

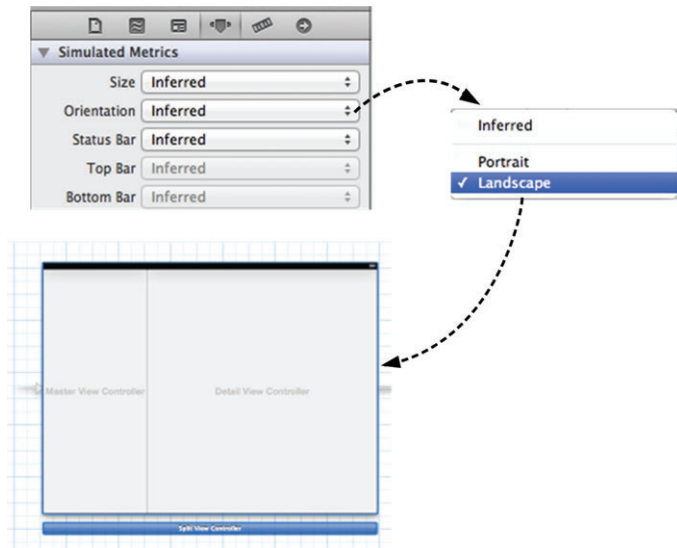


图7-26 横屏方向的设计视图

本例中的MasterView没有采用导航控制器，因此删除生成的导航控制器，重新将Split View Controller拖曳到Table View Controller，此时从弹出界面中选择master view controller项，重新连接视图控制器，如图7-27所示。

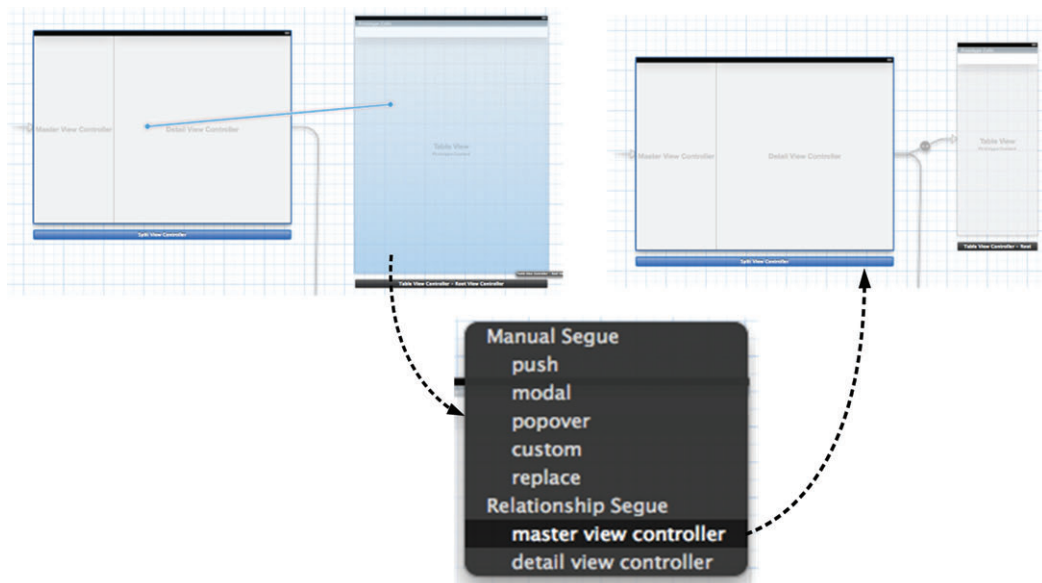


图7-27 重新连接视图控制器

从对象库中拖曳两个View Controller到设计界面，按照图7-28在每个视图中放置一个按钮。

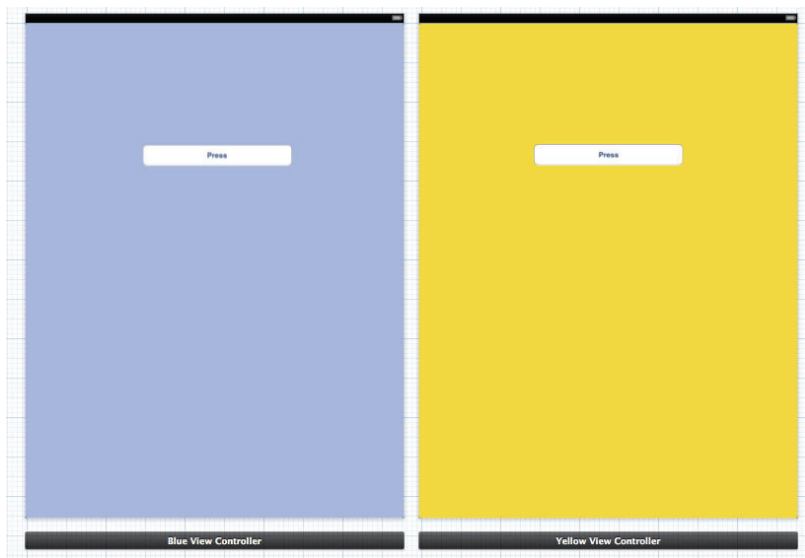




图7-28 视图控制器设计界面

在设计界面中选择Blue View Controller，打开其标识检查器，修改Class为BlueViewController，修改Storyboard ID为blueViewController，如图7-29所示；再打开其属性检查器，修改Size为Detail，如图7-30所示；再选择View，将其背景改为蓝色。

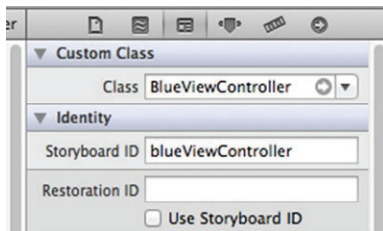


图7-29 标识检查器

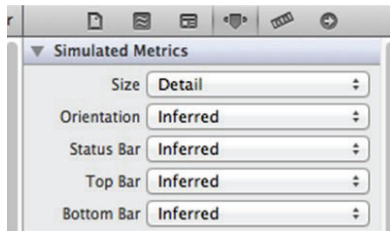


图7-30 属性检查器

按照上面的方法设定Yellow View Controller，修改Class为YellowViewController，Storyboard ID为yellowViewController；然后，再设定Detail View Controller，修改Class为DetailViewController，Storyboard ID为detailViewController。

下面我们看看代码部分，MasterViewController.h的代码如下：

```
@interface MasterViewController : UITableViewController

@property (nonatomic, strong) NSArray *listData;
@property (strong, nonatomic) DetailViewController *detailViewController;

@end
```

其中listData属性用于存放MasterView中的导航列表标题，detailViewController属性用于存放DetailViewController指针。MasterViewController.m中视图加载方法的代码如下：

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

```

self.listData = [[NSArray alloc] initWithObjects:@"Blue View", @"Yellow View", nil];

self.detailViewController = (DetailViewController *)
    [self.splitViewController.viewControllers lastObject];
}

```

在视图加载方法中,我们需要初始化detailViewController属性,这里的self.splitViewController用于获得它们所在的分栏视图控制器。splitViewController属性由UIViewController类提供,在iPad的UISplitViewController作为根视图控制器时使用。UISplitViewController的viewControllers属性是NSArray指针类型,只能存放两个视图控制器。viewControllers集合的第一个元素是MasterView的根视图控制器,第二个元素是DetailView的根视图控制器。viewControllers集合的lastObject方法用于获得最后一个元素(第二个元素),表示DetailViewController的指针类型。

关于获得DetailView的根视图控制器方法,有的读者会想到通过Storyboard ID得到,类似于下面的代码:

```

DetailViewController * detailViewController = [self.storyboard
    instantiateViewControllerWithIdentifier:@" detailViewController "];

```

很多情况下,我们都是这样获得故事板中的视图控制器的。使用上面的代码,我们可以创建一个新的视图控制器,这个新的视图控制器对象不是应用程序初始时创建的。

MasterView的视图控制器是表视图控制器,它实现的数据源和委托协议方法如下:

```

#pragma mark - Table view数据源方法
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    return 2;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    }

    int row = [indexPath row];
    cell.textLabel.text = [self.listData objectAtIndex:row];
    return cell;
}

#pragma mark - Table view委托方法
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath: (NSIndexPath *)indexPath
{
    int row = [indexPath row];

    [self.detailViewController updateView:row];
}

```

这里实现表视图委托方法tableView:didSelectRowAtIndexPath:的目的是根据选择的行号更新DetailView,其中updateView:方法是在DetailViewController中定义的方法,用于更新视图。接下来我们看看DetailViewController.h的代码:


```

@interface DetailViewController : UIViewController

@property (nonatomic, strong) YellowViewController *yellowViewController;
@property (nonatomic, strong) BlueViewController *blueViewController;

//根据行号更新视图
- (void)updateView:(int)row;

@end

```

在上述代码中，属性yellowViewController和blueViewController是Detail View中要展示视图的控制器。DetailViewController.m代码中的视图加载方法如下：

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.blueViewController = [self.storyboard
        instantiateViewControllerWithIdentifier:@"blueViewController"];
    self.yellowViewController = [self.storyboard
        instantiateViewControllerWithIdentifier:@"yellowViewController"];

    [self.view addSubview: self.blueViewController.view ];
}

```

在该方法中，我们通过Storyboard ID分别创建蓝色视图控制器和黄色视图控制器，然后通过addSubview:方法把蓝色视图放入到MasterView中作为初始视图。在DetailViewController.m代码中，视图方法updateView:的代码如下：

```

//根据行号更新视图
- (void)updateView:(int)row
{
    if (row == 0) { //蓝色

        if (self.yellowViewController.view.superview) { //黄色视图存在，则移除
            [self.yellowViewController.view removeFromSuperview];
        }
        if (self.blueViewController.view.superview == nil) { //添加蓝色视图
            [self.view addSubview: self.blueViewController.view ];
        }
    } else {
        if (self.blueViewController.view.superview) { //蓝色视图存在，则移除
            [self.blueViewController.view removeFromSuperview];
        }
        if (self.yellowViewController.view.superview == nil) { //添加黄色视图
            [self.view addSubview: self.yellowViewController.view ];
        }
    }
}

```

关于BlueViewController和YellowViewController的代码，就没有什么新内容了，本章就不再介绍了。

7.2.3 模态视图专用属性

在图7-5所示的iPad自带的E-mail应用中，横屏模态视图只是显示在屏幕中间，而非占有整个屏幕，这些是由模态视图控制器的modalPresentationStyle属性来控制的，该属性由枚举类型UIModalPresentationStyle定义，主要包括如下4个常量。

- ❑ **UIModalPresentationFullScreen**。全屏状态，是默认呈现样式。iPhone只能全屏呈现。
- ❑ **UIModalPresentationPageSheet**。它的宽度是固定的768点，因此，iPad横屏时的模态视图如图7-31所示，iPad竖屏时则全屏呈现。



图7-31 PageSheet呈现

- ❑ **UIModalPresentationFormSheet**。表示尺寸为固定的540×620点，无论是横屏还是竖屏情况下，呈现尺寸都不会变化，如图7-32所示。
- ❑ **UIModalPresentationCurrentContext**。表示与父视图控制器有相同的呈现方式。

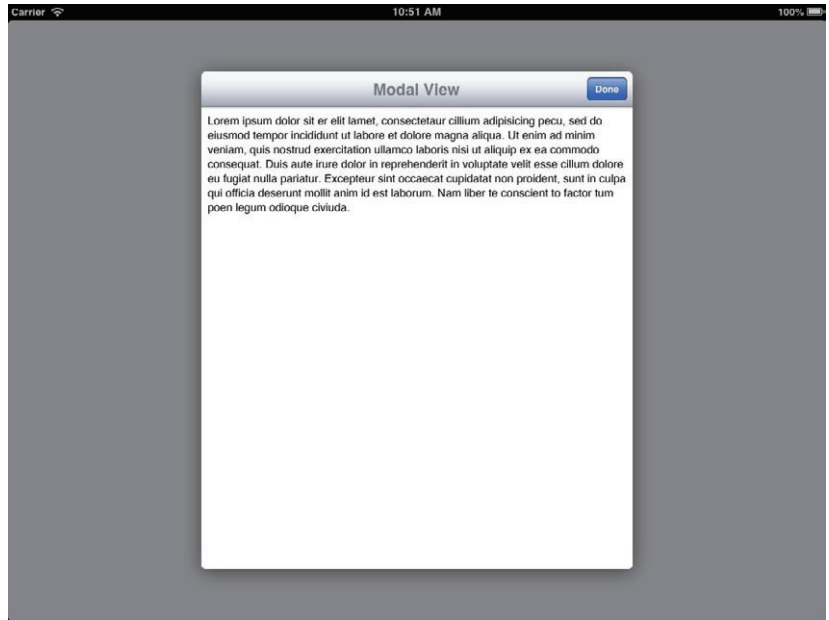


图7-32 FormSheet呈现

下面我们通过一个案例来熟悉一下这个属性。使用Xcode创建工程ModalViewSample，模板采用Single View Application，Devices选择iPad，勾选Use Storyboards和Use Automatic Reference Counting复选框。从对象库中拖曳分段控件和按钮控件到主视图设计界面，按照图7-33所示，分段控件分为4个段：Full Screen、Page Sheet、Form Sheet和Current Context。当用户选择不同段并点击Show Modal View按钮时，应用会根据选择的段呈现模态视图。

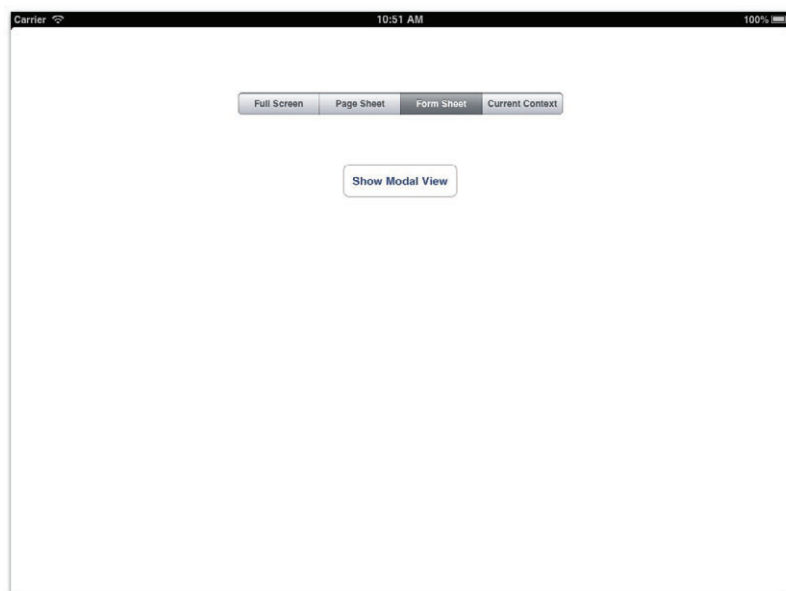


图7-33 主视图设计界面

从对象库中拖曳View Controller到设计界面，按照图7-34所示，在其中放置一个导航栏和一个TextView控件，其中导航栏右边为导航栏按钮，我们将其设置为系统Done样式。然后使用Xcode工具创建ModalViewController视图控制器，并将其作为模态视图控制器，其父类是UIViewController，不用选择xib文件。

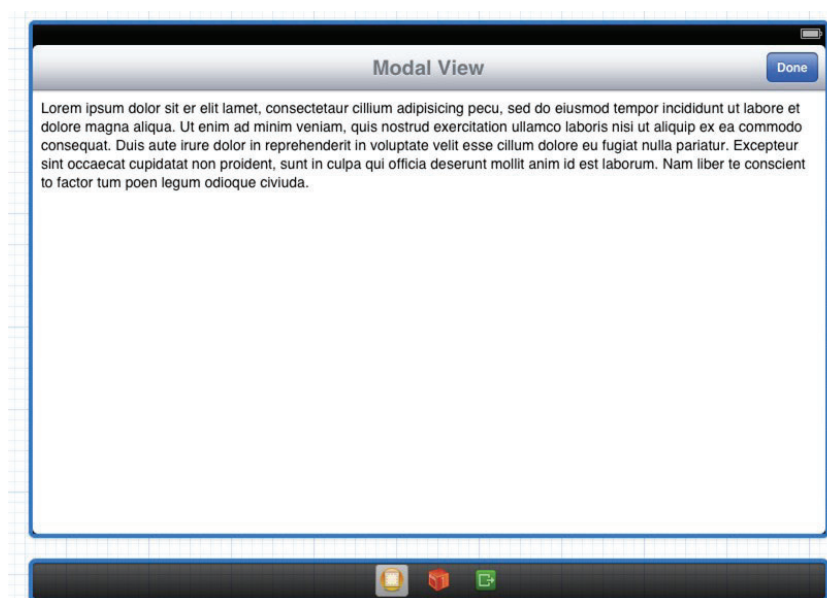


图7-34 模态视图设计界面

接着为Done按钮定义动作事件并连线，然后再打开视图控制器的标识检查器，如图7-35所示，修改Class属性为ModalViewController，Storyboard ID属性为modalViewController。

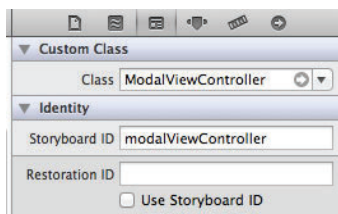


图7-35 ModalViewController标识检查器

然后为主视图中的Show Modal View按钮呈现模态视图，这可以通过故事板连线模态方式。将鼠标从Show Modal View按钮拖曳到模态视图控制器时，会弹出菜单，此时选择modal，如图7-36所示。



图7-36 通过故事板设计模态视图

选择Segue，打开其属性检查器，如图7-37所示，其中Presentation属性表示呈现样式，Transition属性表示界面跳转动画。

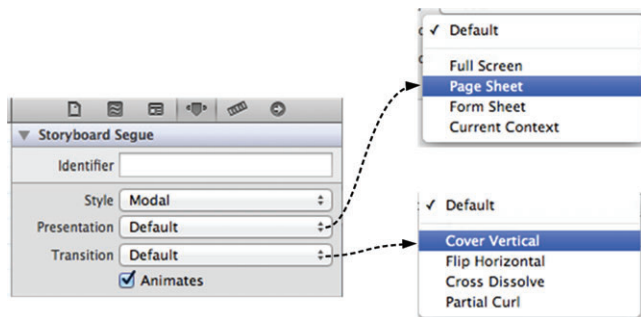


图7-37 Segue属性检查器

这时模态视图就设计完毕了，其中没有编写一行代码。但是，你对于模态视图的原理又了解多少呢？这个问题本质上是“要用故事板设计实现，还是用代码实现”的问题。

下面我们再看看如何用代码来实现，首先我们看一下ViewController.h的代码：

```
@interface ViewController : UIViewController

- (IBAction)onclick:(id)sender;

@property (weak, nonatomic) IBOutlet UISegmentedControl *segControl;

@end
```

ViewController.m中的按钮点击事件代码如下：

```
- (IBAction)onclick:(id)sender {

    ModalViewController *modalViewController = [self.storyboard
        instantiateViewControllerWithIdentifier:@"modalViewController"];

    modalViewController.modalTransitionStyle = UIModalTransitionStyleCoverVertical;

    switch (self.segControl.selectedSegmentIndex) {
        case 0:
            modalViewController.modalPresentationStyle = UIModalPresentationFullScreen;
            break;
        case 1:
            modalViewController.modalPresentationStyle = UIModalPresentationPageSheet;
            break;
        case 2:
            modalViewController.modalPresentationStyle = UIModalPresentationFormSheet;
            break;
        default:
            modalViewController.modalPresentationStyle = UIModalPresentationCurrentContext;
            break;
    }
    [self presentViewController:modalViewController animated:YES completion:nil];
}
```

此外，ModalViewController.m中还有一个关闭模态视图的方法，其代码如下：

```
- (IBAction)onclick:(id)sender {
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

7.3 Master-Detail 应用程序模板

前面我们提到过Master-Detail应用程序模板，它是一个非常重要也比较复杂的模板，可以帮助我们构建树形结构导航模式的应用程序。

图7-38是使用Master-Detail模板创建的iPhone应用。采用导航控制器与表视图结合的树形结构导航模式，可以浏览、删除、修改和添加数据。点击Master视图中的单元格，可以进入Detail视图。

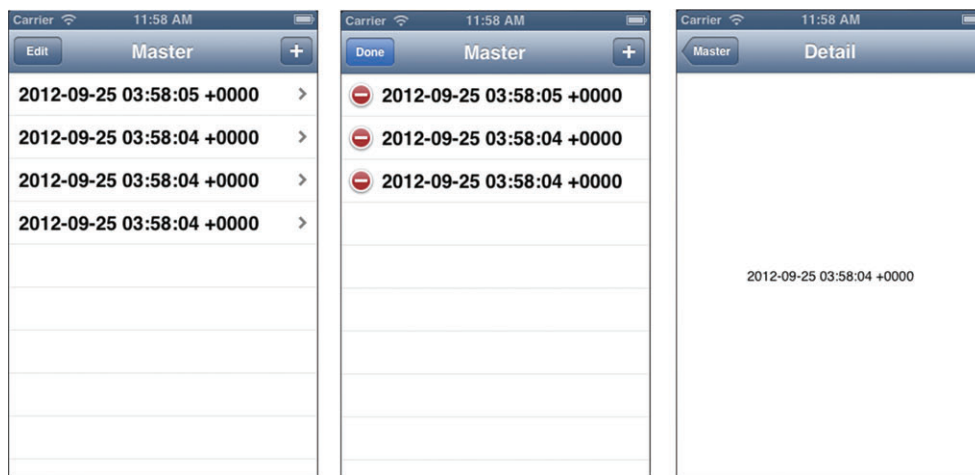


图7-38 使用Master-Detail模板创建的iPhone应用

图7-39是使用Master-Detail模板创建的iPad应用的横屏界面，它采用了导航控制器、表视图、SplitView等构建的iPad树形结构导航模式。图7-40是iPad竖屏的情况，点击Master按钮，会从侧面滑出PopoverView。

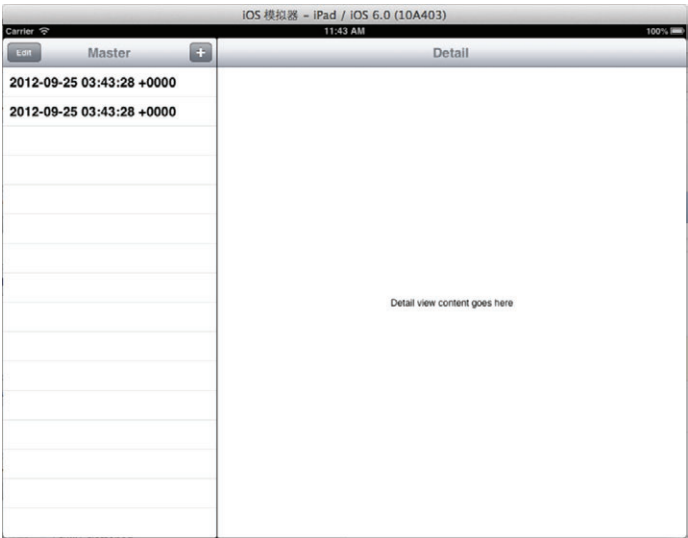


图7-39 iPad横屏界面

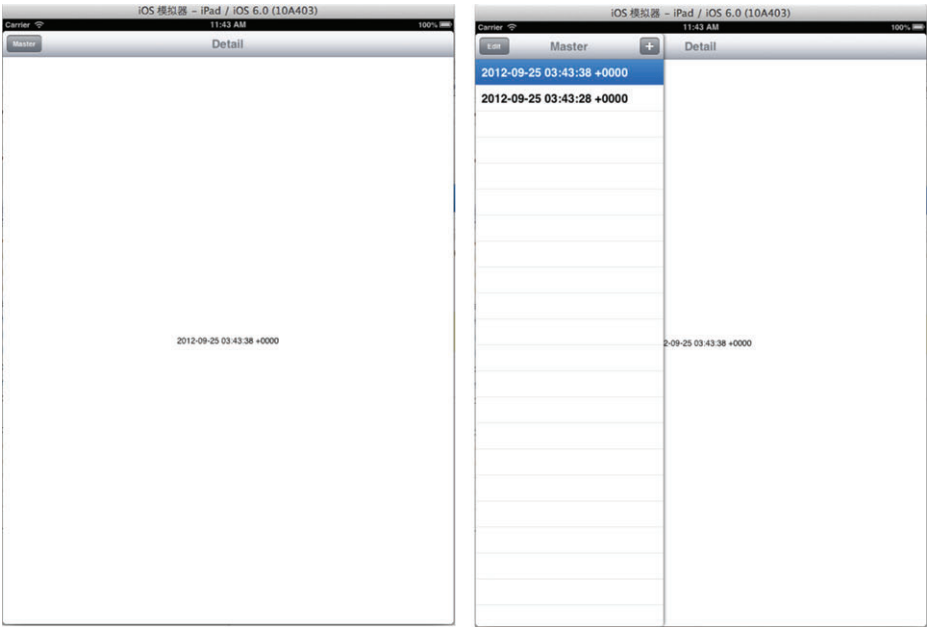


图7-40 iPad竖屏界面

下面我们分别介绍使用nib和故事板技术的实现方式。

7.3.1 nib实现

在本节中，我们将介绍如何用nib技术实现Master-Detail应用程序模板。

使用Xcode创建工程MasterDetailNib，如图7-41所示，其中模板采用Master-Detail Application，Devices选择Universal，不选中Use Storyboards复选框，选中Use Automatic Reference Counting复选框。由于本工程可以自适应iPad和iPhone两个设备，所以xib文件有两套，共10个文件，下面简要说明这10个文件。

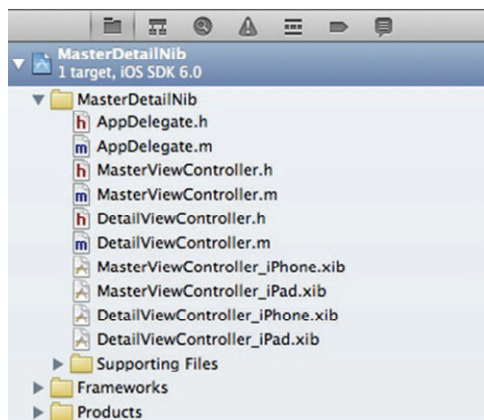


图7-41 模板生成文件

- ❑ AppDelegate.h和AppDelegate.m。应用程序委托对象。
- ❑ MasterViewController.h和MasterViewController.m。Master视图控制器。
- ❑ DetailViewController.h和DetailViewController.m。Detail视图控制器。
- ❑ MasterViewController_iPhone.xib。Master视图控制器的iPhone版nib文件。
- ❑ MasterViewController_iPad.xib。Master视图控制器的iPad版nib文件。
- ❑ DetailViewController_iPhone.xib。Detail视图控制器的iPhone版nib文件。
- ❑ DetailViewController_iPad.xib。Detail视图控制器的iPad版nib文件。

下面我们先从AppDelegate的代码开始介绍。AppDelegate.m中应用启动方法的代码如下：

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];
    if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPhone) {
        MasterViewController *masterViewController = [[MasterViewController alloc]
            initWithNibName:@"MasterViewController_iPhone" bundle:nil];
        self.navigationController = [[UINavigationController alloc]
            initWithRootViewController:masterViewController];
        self.window.rootViewController = self.navigationController;
    } else {
        MasterViewController *masterViewController = [[MasterViewController alloc]
            initWithNibName:@"MasterViewController_iPad" bundle:nil];
        UINavigationController *masterNavigationController =
            [[UINavigationController alloc]
                initWithRootViewController:masterViewController];

        DetailViewController *detailViewController = [[DetailViewController alloc]
            initWithNibName:@"DetailViewController_iPad" bundle:nil];
        UINavigationController *detailNavigationController =
            [[UINavigationController alloc]
                initWithRootViewController:detailViewController];

        //Detail视图控制器对象赋值给Master视图控制器的detailViewController属性
        masterViewController.detailViewController = detailViewController;

        self.splitViewController = [[UISplitViewController alloc] init];
```

```

        self.splitViewController.delegate = detailViewController;
        self.splitViewController.viewControllers = @[masterNavigationController,
            detailNavigationController];
        // 创建的SplitView控制器，用于作为window的根视图控制器
        self.window.rootViewController = self.splitViewController;
    }
    [self.window makeKeyAndVisible];
    return YES;
}

```

由于这个工程可以自适应iPad和iPhone两个设备，所以我们需要在程序中判断设备。下面的语句用于判断设备是否为iPhone设备：

```
if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPhone) {}
```

如果为iPhone设备，情况就比较简单，直接从nib文件中创建Master视图控制器，再来构建导航视图控制器，接着把导航控制器作为window的根视图控制器就可以了。如果为iPad设备，情况就比较复杂，由于Master和Detail视图都有导航控制器，所以先使用nib文件创建视图控制器，再分别来构建左右导航控制器对象，相关代码如下：

```

MasterViewController *masterViewController = [[MasterViewController alloc]
    initWithNibName:@"MasterViewController_iPad" bundle:nil];
UINavigationController *masterNavigationController = [[UINavigationController alloc]
    initWithRootViewController:masterViewController];

DetailViewController *detailViewController = [[DetailViewController alloc]
    initWithNibName:@"DetailViewController_iPad" bundle:nil];
UINavigationController *detailNavigationController = [[UINavigationController alloc]
    initWithRootViewController:detailViewController];

```

创建UISplitViewController之后，把上面创建的左右导航控制器放入viewControllers属性中。一定要注意的是，第一个元素是左侧的Master视图控制器，第二个元素是右侧的Detail视图控制器：

```
self.splitViewController.viewControllers = @[masterNavigationController,
    detailNavigationController];
```

在代码中，我们还将Detail视图控制器detailViewController的指针分配SplitView控制器的委托属性，相关代码如下：

```
self.splitViewController.delegate = detailViewController;
```

这里为什么让Detail视图控制器作为SplitView控制器的委托对象呢？这是因为UISplitViewControllerDelegate中定义的方法是为了响应视图显示和隐藏，而Detail视图无论在竖屏还是横屏情况下都能显示。

下面我们看看MasterViewController.h的代码，其中MasterViewController继承了UITableViewController（它是一个表视图控制器）：

```

#import <UIKit/UIKit.h>

@class DetailViewController;

@interface MasterViewController : UITableViewController

@property (strong, nonatomic) DetailViewController *detailViewController;

@end

```

MasterViewController.m中构造方法的代码如下：

```

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        self.title = NSLocalizedString(@"Master", @"Master");
        if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPad) {
            self.clearsSelectionOnViewWillAppear = NO;

```

```

        self.contentSizeForViewInPopover = CGSizeMake(320.0, 600.0);
    }
}
return self;
}

```

在上述代码中，`NSLocalizedString`是字符串本地函数。在iPad设备中，我们通过`self.contentSizeForViewInPopover = CGSizeMake(320.0, 600.0)`设定Popover视图的大小。MasterViewController.m中视图加载方法的代码如下：

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    UIBarButtonItem *addButton = [[UIBarButtonItem alloc]
        initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
        target:self action:@selector(insertNewObject:)];
    self.navigationItem.rightBarButtonItem = addButton;
}

```

在上述代码中，我们设定导航栏左按钮为编辑按钮，右按钮为系统定义的UIBarButtonItemAdd按钮。点击右按钮时，会调用insertNewObject:方法，其代码如下：

```

- (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];
}

```

该方法的作用是向集合中插入当前日期对象，然后使用表视图insertRowsAtIndexPaths:withRowAnimation:方法插入行。

在MasterViewController.m文件中，我们还实现了表视图数据源和委托方法，下面我们解释一下表视图数据源方法tableView:didSelectRowAtIndexPath:，其代码如下所示：

```

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSDate *object = _objects[indexPath.row];
    if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPhone) {
        if (!self.detailViewController) {
            self.detailViewController = [[DetailViewController alloc]
                initWithNibName:@"DetailViewController_iPhone" bundle:nil];
        }
        self.detailViewController.detailItem = object;
        [self.navigationController pushViewController:self.detailViewController
            animated:YES];
    } else {
        self.detailViewController.detailItem = object;
    }
}

```

该方法是点击单元格时调用的方法。在iPhone设备下，如果Detail视图控制器没有创建，就从nib文件中创建。无论是iPhone还是iPad设备，最后都需要从集合中取出日期对象并将其赋值给Detail视图控制器的detailItem属性，然后调用它的导航控制器的pushViewController:方法跳转到Detail视图。

我们在看看DetailViewController.h的代码。DetailViewController继承自UIViewController，实现了

UISplitViewControllerDelegate协议，相关代码如下：

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

@interface DetailViewController : UIViewController <UISplitViewControllerDelegate>

@property (strong, nonatomic) id detailItem;

@property (weak, nonatomic) IBOutlet UILabel *detailDescriptionLabel;
@end
```

在DetailViewController.m中，还有一些应该在.h文件中定义的内容，具体如下所示：

```
@interface DetailViewController ()
@property (strong, nonatomic) UIPopoverController *masterPopoverController;
- (void)configureView;
@end
```

其中DetailViewController()是定义DetailViewController的一个分类，添加了masterPopoverController属性和configureView方法。

此外，DetailViewController.m中还有setDetailItem:方法，事实上它与detailItem属性对应。属性本质上是封装了set方法和get方法，其中set方法的命名规则为“set+属性首字母大写+属性其他字母”。setDetailItem:方法的代码如下：

```
- (void)setDetailItem:(id)newDetailItem
{
    if (_detailItem != newDetailItem) {
        _detailItem = newDetailItem;

        //更新视图
        [self configureView];
    }

    if (self.masterPopoverController != nil) {
        [self.masterPopoverController dismissPopoverAnimated:YES];
    }
}
```

在上述代码中，我们调用了configureView方法更新视图，然后在满足self.masterPopoverController != nil这个条件时，关闭Popover视图。configureView方法用来更新Detail视图中的标签，其代码如下所示：

```
- (void)configureView
{
    if (self.detailItem) {
        self.detailDescriptionLabel.text = [self.detailItem description];
    }
}
```

在DetailViewController的视图加载方法viewDidLoad中，我们也调用了configureView方法：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    [self configureView];
}
```

DetailViewController构造方法的代码如下所示：

```
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        self.title = NSLocalizedString(@"Detail", @"Detail");
    }
}
```

```
    return self;
}
```

此外，在DetailViewController中，我们还实现了UISplitViewControllerDelegate委托协议的两个方法，相关代码如下：

```
- (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;
}
```

splitViewController:willHideViewController:withBarButtonItem:forPopoverController:方法在Master视图隐藏时调用，当屏幕由横屏变为竖屏时，Master视图会隐藏。barButtonItem参数负责呈现Popover视图按钮，我们把这个按钮设置成Detail视图导航栏的左侧按钮。另外，我们把popoverController参数赋值给masterPopoverController属性。

splitViewController:willShowViewController:invalidatingBarButtonItem:方法在Master视图显示时调用，当屏幕由竖屏变为横屏时，要清除Detail视图导航栏左侧的按钮，并将masterPopoverController属性赋值为nil。

7.3.2 故事板实现

Master-Detail应用程序模板也可采用故事板实现，其重点是故事板中的设置，代码要比nib少一些。

使用Xcode创建工程MasterDetailStoryborad，如图7-42所示，其中模板采用Master-Detail Application，Devices选择Universal，勾选Use Storyborads和Use Automatic Reference Counting复选框。由于本工程可以自适应iPad和iPhone两个设备，所以xib文件有两套，共8个文件，其中各个文件的含义如下所示。

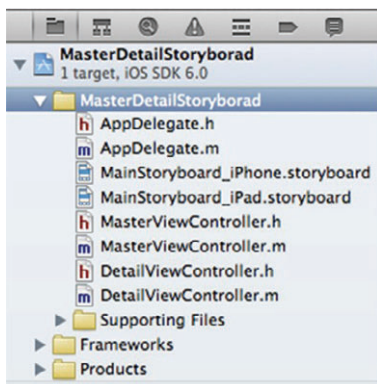


图7-42 模板生成文件

□ AppDelegate.h和AppDelegate.m。应用程序委托对象。

- ❑ MasterViewController.h和MasterViewController.m。Master视图控制器。
- ❑ DetailViewController.h和DetailViewController.m。Detail视图控制器。
- ❑ MainStoryboard_iPhone.storyboard。iPhone版的故事板文件。
- ❑ MainStoryboard_iPad.storyboard。iPad版的故事板文件。

打开MainStoryboard_iPhone.storyboard文件，里面共有3个视图控制器，如图7-43所示。

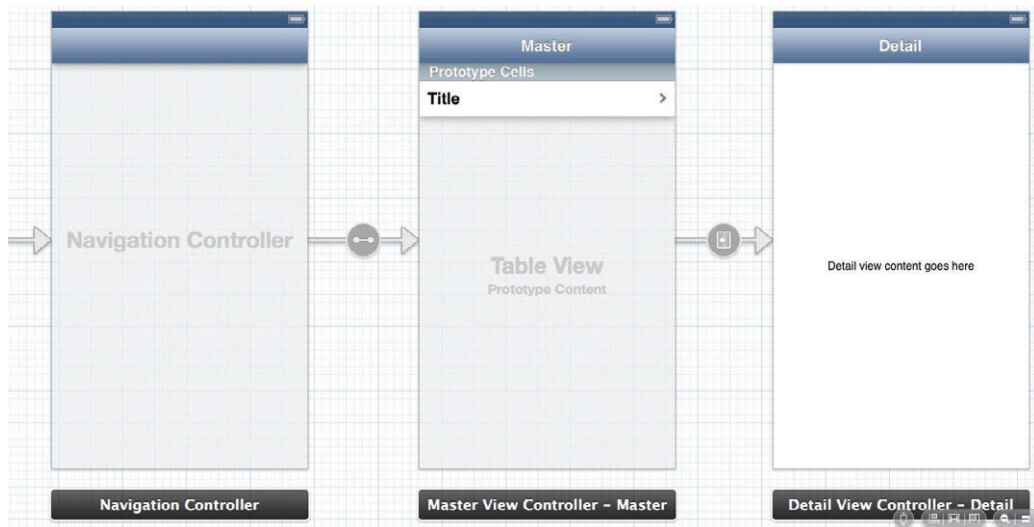


图7-43 MainStoryboard_iPhone.storyboard文件

初始视图控制器是导航控制器，Master视图控制器采用的是动态表视图。点击Master视图控制器中的单元格时，会打开Detail视图，图7-44中的Segue就是完成这个跳转的。图7-45是Segue属性检查器，它的Identifier属性是showDetail，Style属性为push（push相当于pushViewController:animated:方法）。

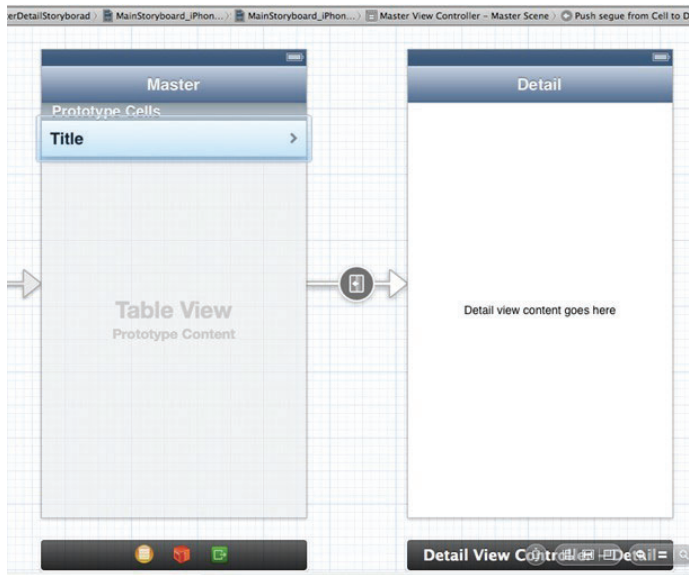


图7-44 Master视图控制器到Detail视图控制器的Segue

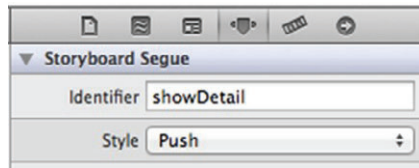


图7-45 Segue属性检查器

打开MainStoryboard_iPad.storyboard文件，里面共有5个视图控制器，如图7-46所示。由于视图控制器上的文字很小，我们对它们进行了编号。

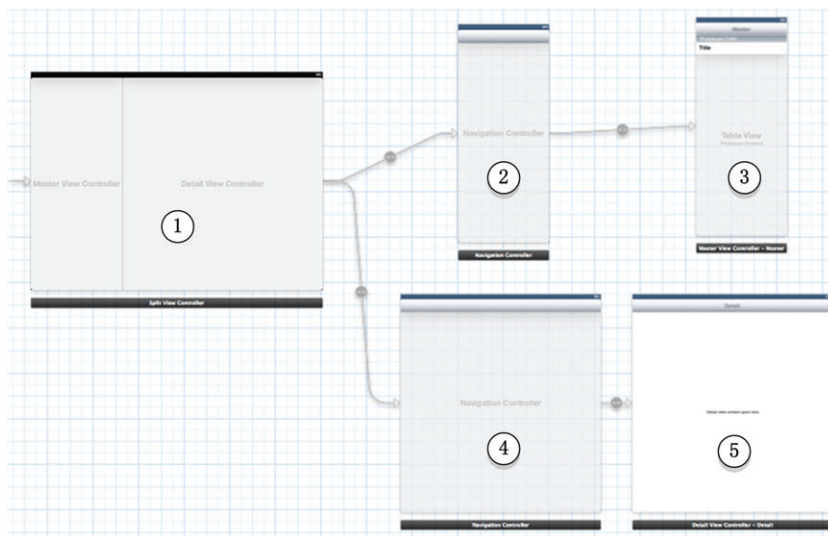


图7-46 MainStoryboard_iPad.storyboard文件

①号控制器是SplitViewController；②号控制器是NavigationController，它是Master视图的根视图控制器；③号控制器是Master视图控制器；④号控制器也是NavigationController，它是Detail视图的根视图控制器；⑤号控制器是Detail视图控制器。

下面我们先从AppDelegate代码开始介绍。AppDelegate.m中对应的应用启动方法的代码如下：

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
(NSDictionary *)launchOptions
{
    if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPad) {
        UISplitViewController *splitViewController = (UISplitViewController *)
            self.window.rootViewController;
        UINavigationController *navigationController =
            [splitViewController.viewControllers lastObject];
        splitViewController.delegate = (id)navigationController.topViewController;
    }
    return YES;
}
```

由于采用了故事板，iPhone设备不需要什么代码，因此该方法是针对iPad设备的。从应用程序委托对象的window属性中取出根视图控制器，这个根视图控制器是UISplitViewController类型。

[splitViewController.viewControllers lastObject]用于取出Detail视图中的导航控制器。topViewController属性用于获得导航控制器当前呈现的视图控制器，就本例而言它是DetailViewController，这里把它分配给UISplitViewController的delegate属性，即UISplitViewController视图控制器的委托对象。

对于MasterViewController视图控制器中的代码，其中与nib方式一样的部分就不再介绍，下面是不同的代码：

```
- (void)awakeFromNib
{
    if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPad) {
        self.clearsSelectionOnViewWillAppear = NO;
        self.contentSizeForViewInPopover = CGSizeMake(320.0, 600.0);
    }
}
```

```

    }
    [super awakeFromNib];
}

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPad) { ①
        NSDate *object = _objects[indexPath.row];
        self.detailViewController.detailItem = object;
    }
}

- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender ②
{
    if ([segue identifier] isEqualToString:@"showDetail"]) {
        NSIndexPath *indexPath = [self.tableView indexPathForSelectedRow];
        NSDate *object = _objects[indexPath.row];
        [[segue destinationViewController] setDetailItem:object];
    }
}

```

awakeFromNib方法与使用nib实现时构造方法initWithNibName:bundle:的作用一样,但需要注意的是在故事板中initWithNibName:bundle:构造方法不会被调用。awakeFromNib方法在加载故事板或nib文件时调用,可以在该方法中重新设置Popover视图的大小。

tableView:didSelectRowAtIndexPath:方法只处理了iPad设备,而iPhone设备的单元格选择处理是在第②行的prepareForSegue:sender:方法(该方法由Segue触发)中完成的。if ([[segue identifier] isEqualToString:@"showDetail"])语句用于判断Segue的identifier属性是否等于showDetail,这是在故事板中设定好的。

DetailViewController.m的代码与nib实现基本上一样,这里不再介绍。

7.4 Utility 应用程序模板

Utility应用程序模板比Master-Detail模板更简单,使用它可以帮助我们构建iOS中的“实用型”应用程序。

图7-47是使用Utility模板创建的iPhone应用,它采用模态视图导航模式,点击主视图(左边视图)右下角的i按钮,主视图会翻转到子视图(右边视图),在子视图中点击Done按钮又回到了主视图。

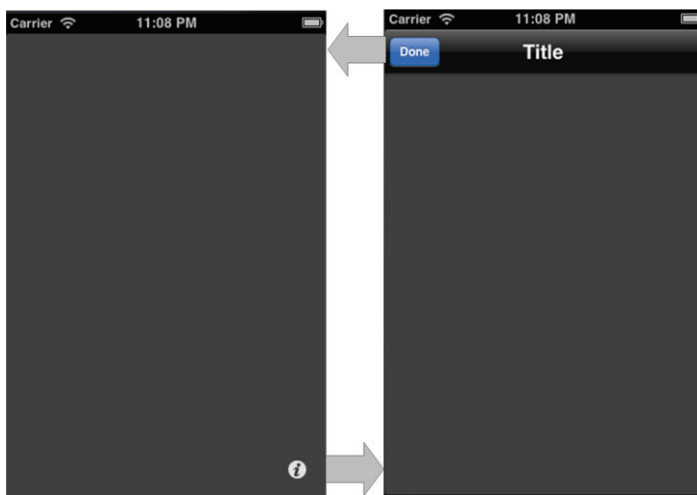


图7-47 使用Utility模板创建的iPhone应用

图7-48是使用Utility模板创建的iPad应用。与iPhone应用不同的是，iPad应用将呈现的内容放在一个Popover视图中，该视图在点击导航栏右侧的Info按钮时出现。



图7-48 iPad下Utility模板应用Popover视图

下面我们分别介绍使用nib和故事板技术的实现方式。

7.4.1 nib实现

使用Xcode创建工程UtilitySampleNib，如图7-49所示，其中模板采用Utility Application，Devices选择Universal，不选中Use Storyboards复选框，选中Use Automatic Reference Counting复选框。由于这个工程可以自适应iPad和iPhone两个设备，所以xib文件有两套，共9个文件，其中各个文件的含义如下所示。

- ❑ AppDelegate.h和AppDelegate.m。应用程序委托对象。
- ❑ MainViewController.h和MainViewController.m。主视图控制器。
- ❑ FlipsideViewController.h和FlipsideViewController.m。子视图控制器。
- ❑ MainViewController_iPhone.xib。主视图控制器的iPhone版nib文件。
- ❑ MainViewController_iPad.xib。主视图控制器的iPad版nib文件。
- ❑ FlipsideViewController.xib。子视图控制器的nib文件。

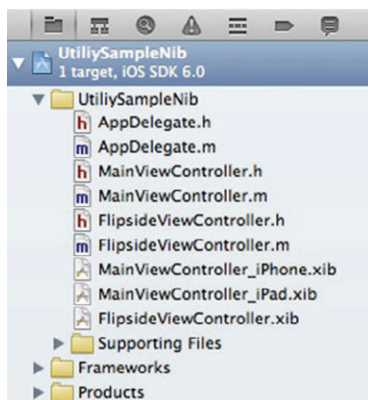


图7-49 模板生成文件

下面我们先从AppDelegate代码开始介绍。AppDelegate.m中对应的应用启动方法的代码如下：

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

    if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPhone) {
        self.mainViewController = [[MainViewController alloc]
            initWithNibName:@"MainViewController_iPhone" bundle:nil];
    } else {
        self.mainViewController = [[MainViewController alloc]
            initWithNibName:@"MainViewController_iPad" bundle:nil];
    }
    self.window.rootViewController = self.mainViewController;
    [self.window makeKeyAndVisible];
    return YES;
}
```

根据设备的不同选择不同的主视图控制器的nib文件来创建视图控制器，然后把它分配self.mainViewController作为window根视图控制器。

主视图控制器MainViewController.h的代码如下：

```
#import "FlipsideViewController.h"

@interface MainViewController : UIViewController <FlipsideViewControllerDelegate>

@property (strong, nonatomic) UIPopoverController *flipsidePopoverController;

- (IBAction)showInfo:(id)sender;

@end
```

MainViewController继承了UIViewController，实现了FlipsideViewControllerDelegate协议，这个协议是在FlipsideViewController.h文件中定义的。flipsidePopoverController属性用于在iPad设备中点击Info按钮时呈现Popover视图。showInfo:方法用于响应iPad的Info按钮和iPhone的按钮。在MainViewController.m中，FlipsideViewControllerDelegate协议方法的代码如下所示：

```
- (void)flipsideViewControllerDidFinish:(FlipsideViewController *)controller
{
    if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPhone) {
        [self dismissViewControllerAnimated:YES completion:nil];
    } else {
        [self.flipsidePopoverController dismissPopoverAnimated:YES];
    }
}
```

该方法的作用就是在iPhone设备上关闭模态视图，在iPad设备上关闭Popover视图。

在MainViewController.m中，showInfo:方法的代码如下：

```
- (IBAction)showInfo:(id)sender
{
    if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPhone) {
        FlipsideViewController *controller = [[FlipsideViewController alloc]
            initWithNibName:@"FlipsideViewController" bundle:nil];
        controller.delegate = self;
        controller.modalTransitionStyle = UIModalTransitionStyleFlipHorizontal;
        [self presentViewController:controller animated:YES completion:nil];
    } else {
        if (!self.flipsidePopoverController) {
            FlipsideViewController *controller = [[FlipsideViewController alloc]
                initWithNibName:@"FlipsideViewController" bundle:nil];
            controller.delegate = self;
```

```

        self.flipsidePopoverController = [[UIPopoverController alloc]
            initWithContentViewController:controller];
    }
    if ([self.flipsidePopoverController isPopoverVisible]) {
        [self.flipsidePopoverController dismissPopoverAnimated:YES];
    } else {
        [self.flipsidePopoverController presentPopoverFromBarButtonItem:sender
            permittedArrowDirections:UIPopoverArrowDirectionAny animated:YES];
    }
}
}

```

在上述代码中,如果设备是iPhone,则采用模态视图控制器,先从nib文件中创建FlipsideView-Controller实例,然后把当前视图控制器分配给FlipsideViewControllers的delegate属性,接着设定模态视图跳转样式为UIModalTransitionStyleFlipHorizontal,然后通过presentViewController:animated:completion:方法呈现模态视图。

如果设备是iPad设备,则采用Popover视图呈现。if (!self.flipsidePopoverController)用于判断Popover视图控制器是否为nil,如果为nil,则从nib文件中实例化FlipsideViewControllers,然后将其作为Popover视图的内容视图来实例化UIPopoverController并分配给flipsidePopoverController属性。为了防止多次呈现Popover视图,需要使用if ([self.flipsidePopoverController isPopoverVisible])语句判断是否已经呈现,如果已经呈现,则使用UIPopoverController的dismissPopoverAnimated:方法关闭,否则使用UIPopoverController的presentPopoverFromBarButtonItem:permittedArrowDirections:animated:方法呈现。

子视图控制器FlipsideViewController.h的代码如下:

```

#import <UIKit/UIKit.h>

@class FlipsideViewController;

@protocol FlipsideViewControllerDelegate
- (void)flipsideViewControllerDidFinish:(FlipsideViewController *)controller;
@end

@interface FlipsideViewController : UIViewController

@property (weak, nonatomic) id <FlipsideViewControllerDelegate> delegate;

- (IBAction)done:(id)sender;

@end

```

在这个文件中,我们定义了FlipsideViewControllerDelegate协议。该协议可以单独放在另外一个.h文件中,也可以与别的.h文件放在一起。这都没有关系,只是在使用之前要保证导入该.h文件。在这个协议中,我们定义了flipsideViewControllerDidFinish:方法,用于在子视图关闭时调用。

此外,我们还定义了delegate属性,它是id <FlipsideViewControllerDelegate>类型,即所有实现FlipsideViewControllerDelegate协议的类的类型。

子视图控制器FlipsideViewController.m的代码如下:

```

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self)
    {
        self.contentSizeForViewInPopover = CGSizeMake(320.0, 480.0);
    }
    return self;
}

```

```
- (IBAction)done:(id)sender
{
    [self.delegate flipsideViewControllerDidFinish:self];
}
```

在上述代码中, initWithNibName:bundle:是构造方法, contentViewInPopover属性只有在iPad下才有意义,而模板生成的代码没有关于是哪一种设备的判断, done:方法用于响应done按钮的事件。使用 delegate属性,能够回调主视图控制器的 flipsideViewControllerDidFinish:方法,这时候的 delegate保持的就是主视图控制器对象 MainViewController。

7.4.2 故事板实现

使用Xcode创建工程UtilitySampleStoryboard,如图7-50所示,模板采用Utility Application, Devices选择Universal,选中Use Storyboards和Use Automatic Reference Counting复选框。由于生成的工程可以自适应iPad和iPhone这两个设备,所以故事板文件有两套,共8个文件。

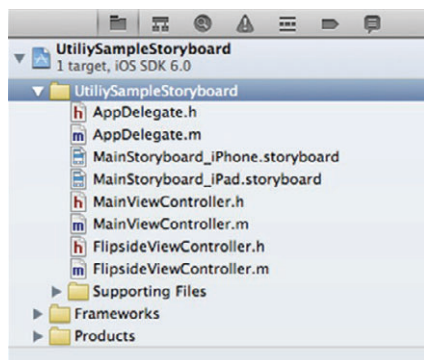


图7-50 UtilitySampleStoryboard工程

打开MainStoryboard_iPhone.storyboard文件,其中共有两个视图控制器,如图7-51所示。

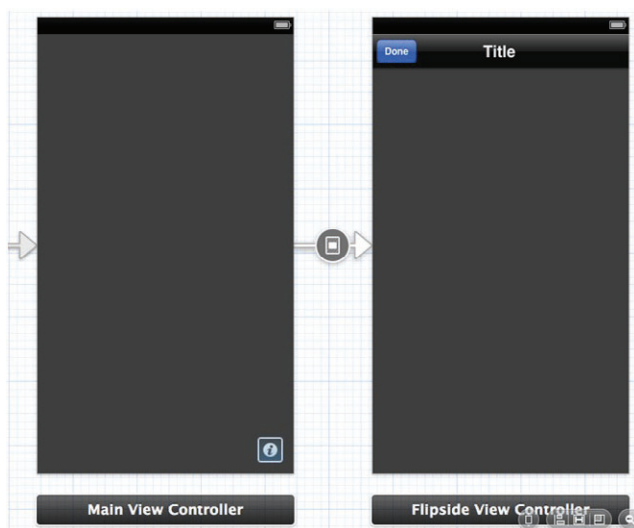


图7-51 MainStoryboard_iPhone.storyboard文件

初始视图控制器是MainViewController，它与FlipsideViewController之间的连接为Segue。打开MainStoryboard_iPad.storyboard文件，其中有两个视图控制器，如图7-52所示。

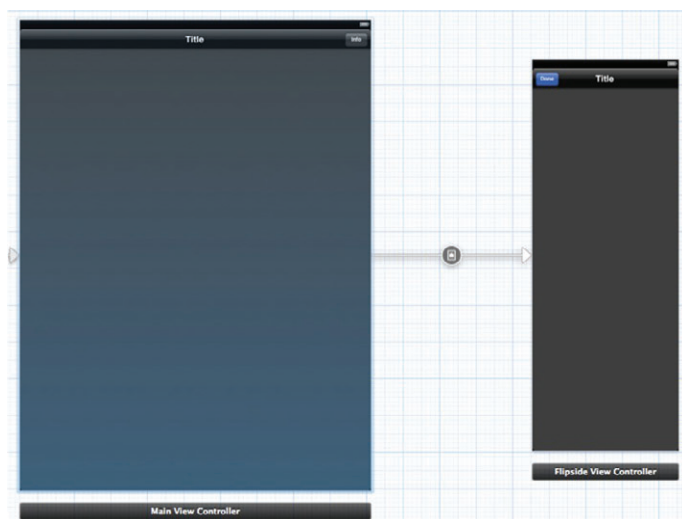


图7-52 MainStoryboard_iPad.storyboard文件

下面我们看看程序代码。AppDelegate没有修改任何内容，都是默认生成的代码，这里不再介绍。主视图控制器MainViewController.h的代码如下：

```
#import "FlipsideViewController.h"

@interface MainViewController : UIViewController <FlipsideViewControllerDelegate,
UIPopoverControllerDelegate>

@property (strong, nonatomic) UIPopoverController *flipsidePopoverController;

@end
```

在上述代码中，MainViewController除了继承UIViewController外，还实现了FlipsideViewControllerDelegate和UIPopoverControllerDelegate委托协议。主视图控制器MainViewController.m的代码如下：

```
- (void)flipsideViewControllerDidFinish:(FlipsideViewController *)controller
{
    if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPhone) {
        [self dismissViewControllerAnimated:YES completion:nil];
    } else {
        [self.flipsidePopoverController dismissPopoverAnimated:YES];
        self.flipsidePopoverController = nil;
    }
}

- (void)popoverControllerDidDismissPopover:(UIPopoverController *)popoverController
{
    self.flipsidePopoverController = nil;
}

- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    if ([segue.identifier] isEqualToString:@"showAlternate"]) {
        [segue.destinationViewController setDelegate:self];

        if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPad) {
```

```

        UIPopoverController *popoverController =
            [(UIStoryboardSegue *)segue popoverController];
        self.flipsidePopoverController = popoverController;
        popoverController.delegate = self;
    }
}

- (IBAction)togglePopover:(id)sender
{
    if (self.flipsidePopoverController) {
        [self.flipsidePopoverController dismissPopoverAnimated:YES];
        self.flipsidePopoverController = nil;
    } else {
        [self performSegueWithIdentifier:@"showAlternate" sender:sender];
    }
}

```

在上述代码中, popoverControllerDidDismissPopover:方法是UIPopoverControllerDelegate委托中定义的方法, 关闭Popover视图之后调用。

prepareForSegue:sender:方法是由Segue触发的。为了防止与其他Segue混淆, 要判断Segue的Identifier是否等于showAlternate。[[segue destinationViewController] setDelegate:self]语句用于将当前视图控制器分配给destinationViewController视图的delegate属性。下面的语句能够从Segue中获得Popover视图控制器对象:

```
UIPopoverController *popoverController = [(UIStoryboardSegue *)segue popoverController];
```

togglePopover:方法用于响应iPad的Info按钮和iPhone的i按钮。如果flipsidePopoverController非nil, 需要关闭Popover视图。performSegueWithIdentifier:sender:方法可以执行故事板中的Segue, 模板中的Identifier为showAlternate的Segue, 对于iPad(见图7-53)和iPhone(见图7-54)是不一样的。在iPad中, Segue中的Style项是Popover, 即Popover视图。在iPhone中, Segue中的Style项是Modal, 即模态视图。

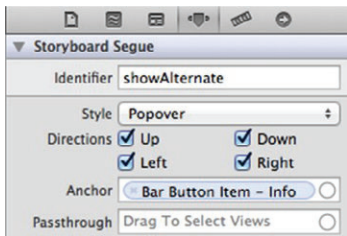


图7-53 iPad的Segue属性检查器

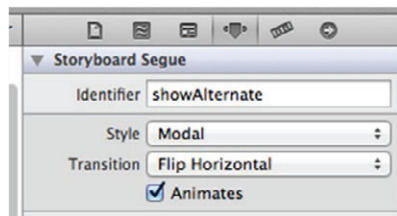


图7-54 iPhone的Segue属性检查器

子视图控制器FlipsideViewController的代码与nib版本基本一致, 这里我们就不再介绍了。

7.5 移动平台的分层架构设计

移动平台上的应用虽然复杂度不高, 但是也应该有架构设计。

7.5.1 低耦合企业级系统架构设计

首先, 我们来了解一下企业级系统架构设计。软件设计的原则是提高软件系统的“可复用性”和“可扩展性”, 系统架构设计采用层次划分方式, 这些层次之间是松耦合的, 层次内部是高内聚的。图7-55是通用低耦合的企业级系统架构图。

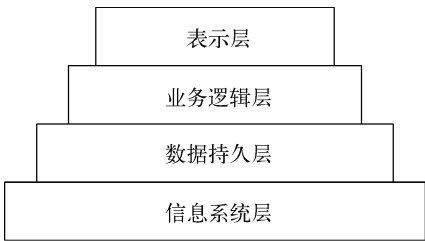


图7-55 通用低耦合的企业级系统架构图

- ❑ **表示层。**用户与系统交互的组件集合。用户通过这一层向系统提交请求或发出指令，系统通过这一层接收用户请求或指令，待指令消化吸收后再调用下一层，接着将调用结果展现到这一层。表示层应该是轻薄的，不应该具有业务逻辑。
- ❑ **业务逻辑层。**系统的核心业务处理层。负责接收表示层的指令和数据，待指令和数据消化吸收后，再进行组织业务逻辑的处理，并将结果返回给表示层。
- ❑ **数据持久层。**数据持久层用于访问信息系统层，从设计规范上讲为了降低耦合度，它不应该具有访问数据库或文件操作的代码，这些代码应该放到数据持久层中。
- ❑ **信息系统层。**系统的数据来源，可以是数据库、文件、遗留系统或者网络数据。

7.5.2 移动平台的分层架构设计

移动平台的应用也需要架构设计，但并非所有的应用都一定基于图7-55所示的通用低耦合企业级系统架构。一般而言，涉及信息处理的应用才使用这种架构设计模式，图7-56展示了iOS平台中信息处理应用的分层架构设计图。

- ❑ **表示层。**它由UIKit Framework构成，包括我们前面学习的视图、控制器、控件和事件处理等内容。
- ❑ **业务逻辑层。**采用什么框架要据具体的业务而定，但一般是具有一定业务处理功能的Objective-C和C++封装的类，或者是C封装的函数。
- ❑ **数据持久层。**提供本地或网络数据访问，它可能是访问SQLite数据的API函数，也可能是Core Data技术，或是访问文件的NSFileManager，或是网络通信技术。采用什么方式要看信息系统层是什么。
- ❑ **信息系统层。**它的信息来源分为本地和网络。本地数据可以放入文件中，也可以放在数据库中，目前iOS本地数据库采用SQLite3。网络可以是某个云服务，也可以是一般的Web服务。

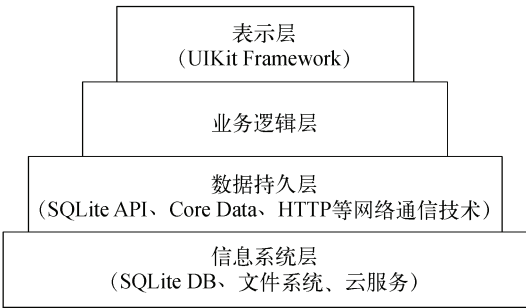


图7-56 iOS平台中信息处理应用的分层架构设计图

7.5.3 基于同一工程的分层

如果我们要编写一个基于iOS（iPhone和iPad两个平台）的MyNotes应用，它具有增加、删除和查询备忘录的

基本功能。图7-57是MyNotes应用的用例图。分层设计之后，表示层可以有iPhone版和iPad版本，而业务逻辑层、数据持久层和信息系统层可以公用，这样大大减少了我们的工作量。

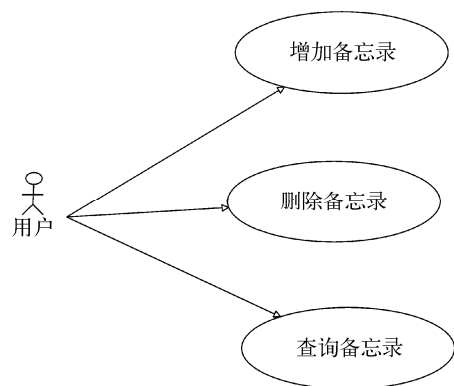


图7-57 MyNotes应用的用例图

考虑到iOS有iPhone和iPad两个平台，我们针对不同的平台绘制了相应的设计原型草图，如图7-58、图7-59和图7-60所示。

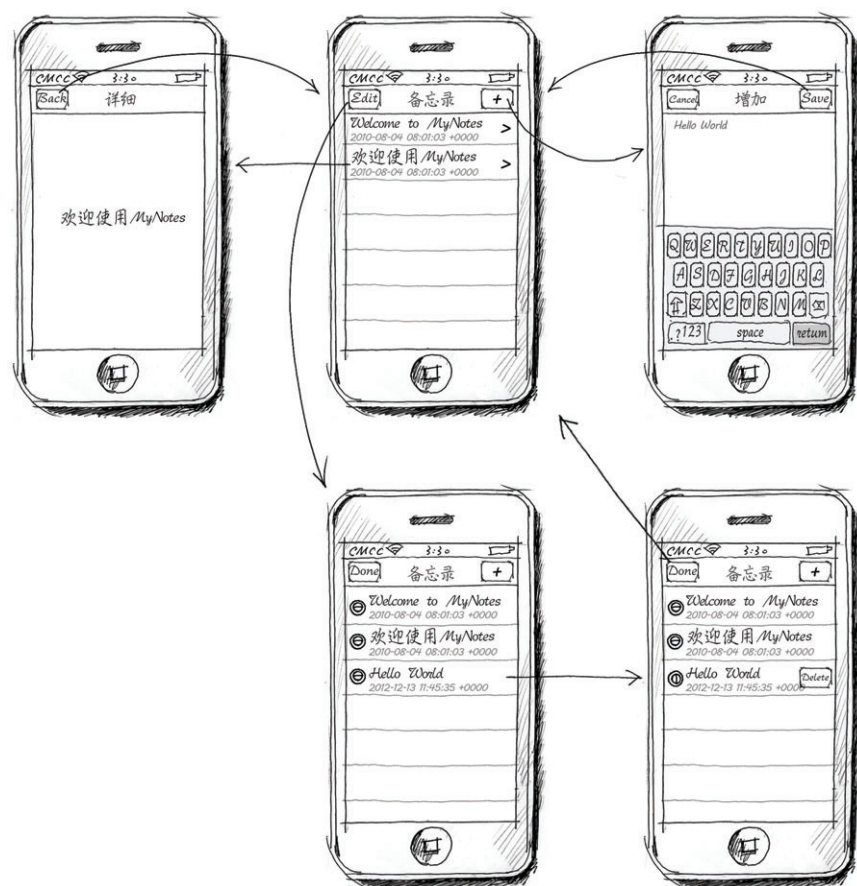


图7-58 iPhone版本的MyNotes设计原型草图

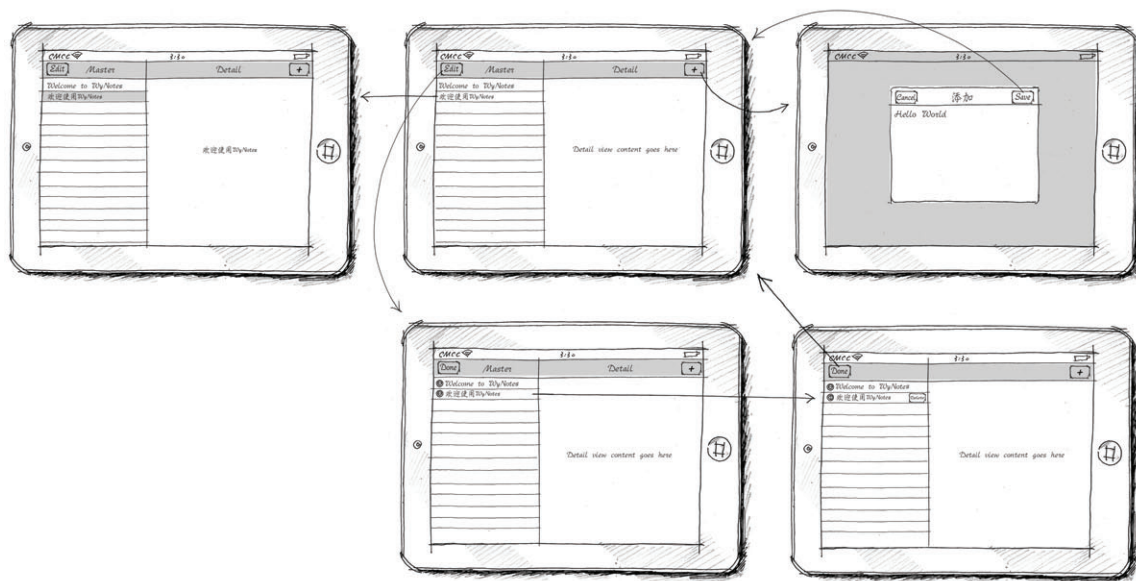


图7-59 iPad版本的MyNotes横屏设计原型草图

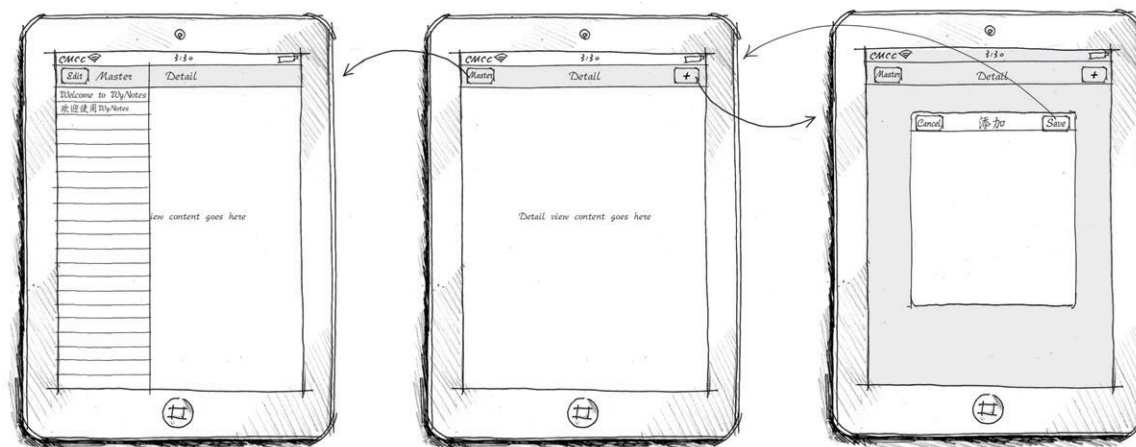


图7-60 iPad版本的MyNotes竖屏设计原型草图

在iOS平台中，分层架构设计有多种模式：基于同一工程的分层、基于一个工作空间不同工程的分层和静态链接库分层。本节简要介绍基于同一工程的分层。

前面构建的自适应iPhone和iPad工程采用的就是基于同一工程的分层模式。打开MyNotes工程，如图7-61所示。在Xcode工程导航面板中，共有3个组——PresentationLayer、BusinessLogicLayer和PersistenceLayer，其中PresentationLayer用于放置表示层相关的类，BusinessLogicLayer用于放置业务逻辑层相关的类，PersistenceLayer用于放置持久层相关的类。

在各个层下面，又是如何划分的呢？我们可以按照业务模块划分，也可以按照组件功能划分。在本应用中，PersistenceLayer层还要分成dao和domain两个组。dao用于放置数据访问对象，该对象中有访问数据的CRUD四类方法。为了降低耦合度，dao一般要设计成为协议（或Java接口），然后根据不同的数据来源采用不同的实现方式。domain组是实体类，实体是应用中的“人”、“事”、“物”等。

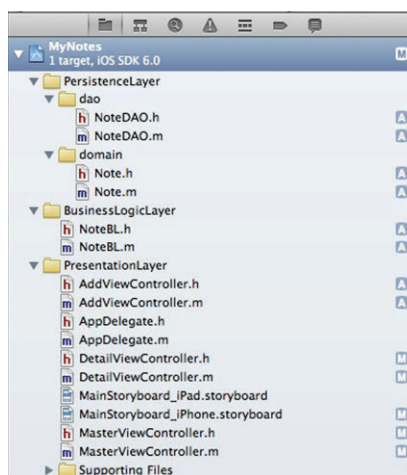


图7-61 MyNotes工程

提示 CRUD方法是访问数据的4个方法——增加、删除、修改和查询，C为Create，表示增加数据，R是Read，表示查询数据，U是Update，表示修改数据，D是Delete，表示删除数据。

在dao组中，NoteDAO.h中的代码如下：

```
@interface NoteDAO : NSObject
//保存数据列表
@property (nonatomic, strong) NSMutableArray* listData;
+ (NoteDAO*)sharedManager;
//插入备忘录的方法
- (int) create:(Note*)model;
//删除备忘录的方法
- (int) remove:(Note*)model;
//修改备忘录的方法
- (int) modify:(Note*)model;
//查询所有数据的方法
- (NSMutableArray*) findAll;
//按照主键查询数据的方法
- (Note*) findById:(Note*)model;
@end
```

在上述代码中，listData属性用于保存数据表中的数据，其中每一个元素都是Note对象。+ (NoteDAO*)sharedManager方法用于获得NoteDAO单例对象。在dao组中，NoteDAO.m中的代码如下：

```
@implementation NoteDAO
static NoteDAO *sharedManager = nil;
+ (NoteDAO*)sharedManager
{
    static dispatch_once_t once;
    dispatch_once(&once, ^{

        NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
        [dateFormatter setDateFormat:@"yyyy-MM-dd HH:mm:ss"];

        NSDate *date1 = [dateFormatter dateFromString:@"2010-08-04 16:01:03"];
        Note* note1 = [[Note alloc] init];
        note1.date = date1;
        note1.content = @"Welcome to MyNotes.";

        NSDate *date2 = [dateFormatter dateFromString:@"2011-12-04 16:01:03"];
```



```

        Note* note2 = [[Note alloc] init];
        note2.date = date2;
        note2.content = @"欢迎使用MyNotes。";

        sharedManager = [[self alloc] init];
        sharedManager.listData = [[NSMutableArray alloc] init];
        [sharedManager.listData addObject:note1];
        [sharedManager.listData addObject:note2];

    });
    return sharedManager;
}

//插入备忘录的方法
-(int) create:(Note*)model
{
    [self.listData addObject:model];
    return 0;
}

//删除备忘录的方法
-(int) remove:(Note*)model
{
    for (Note* note in self.listData) {
        //比较日期主键是否相等
        if ([note.date isEqualToDate:model.date]){
            [self.listData removeObject:note];
            break;
        }
    }
    return 0;
}

//修改备忘录的方法
-(int) modify:(Note*)model
{
    for (Note* note in self.listData) {
        //比较日期主键是否相等
        if ([note.date isEqualToDate:model.date]){
            note.content = model.content;
            break;
        }
    }
    return 0;
}

//查询所有数据的方法
-(NSMutableArray*) findAll
{
    return self.listData;
}

//按照主键查询数据的方法
-(Note*) findById:(Note*)model
{
    for (Note* note in self.listData) {
        //比较日期主键是否相等
        if ([note.date isEqualToDate:model.date]){
            return note;
        }
    }
    return nil;
}
@end

```

NoteDAO采用了单例设计模式来实现,这种模式与DAO设计模式没有关系,主要是出于访问数据方便的考虑。数据放置在listData属性中(这里本应该是从数据库中取出的,但是数据库访问技术我们还没有学习),CRUD方法也都是对listData而非数据库的处理。

在domain组中,Note的代码如下,它只有两个属性——date是创建备忘录的日期,content是备忘录的内容:

```
//
//Note.h

#import <Foundation/Foundation.h>

@interface Note : NSObject

@property(n nonatomic, strong) NSDate* date;
@property(n nonatomic, strong) NSString* content;

@end

//
//Note.m
#import "Note.h"

@implementation Note

@end
```

在业务逻辑层BusinessLogicLayer中,类一般是按照业务模块设计的,它的方法是业务处理方法。下面是NoteBL.h中的代码:

```
@interface NoteBL : NSObject
//插入备忘录的方法
-(NSMutableArray*) createNote:(Note*)model;

//删除备忘录的方法
-(NSMutableArray*) remove:(Note*)model;

//查询所有数据的方法
-(NSMutableArray*) findAll;

@end
```

下面是NoteBL.m中的代码:

```
@implementation NoteBL

//插入备忘录的方法
-(NSMutableArray*) createNote:(Note*)model
{
    NoteDAO *dao = [NoteDAO sharedManager];
    [dao create:model];

    return [dao findAll];
}

//删除备忘录的方法
-(NSMutableArray*) remove:(Note*)model
{
    NoteDAO *dao = [NoteDAO sharedManager];
    [dao remove:model];

    return [dao findAll];
}

//查询所有数据的方法
-(NSMutableArray*) findAll
```

```

{
    NoteDAO *dao = [NoteDAO sharedManager];
    return [dao findAll];
}

@end

```

PresentationLayer是表示层，其中的内容大家应该比较熟悉了，这里不再详细介绍了。

7.5.4 基于一个工作空间不同工程的分层

有时候，我们需要将某一层复用给其他的团队、公司或者个人，但由于某些原因，我们不能提供源代码，此时就可以将业务逻辑层和数据持久层编写成静态链接库（static library或statically-linked library）。

提示 库是一些没有main函数的程序代码的集合。除了静态链接库，还有动态链接库，它们的区别是：静态链接库可以编译到你的执行代码中，应用程序可以在没有静态链接库的环境下运行；动态链接库不能编译到你的执行代码中，应用程序必须在有链接库文件的环境下运行。

基于这种需求，我们可以把3个不同的层放置在不同的工程中，然后再把这3个工程放置到一个工作空间中，为它们设定依赖关系。

下面我们介绍一下详细的创建过程。首先使用Xcode创建工作空间，具体的操作方法是：选择File→New→Workspace...菜单，输入名称MyNotesWorkspace。工作空间是多个工程的集合，我们还需要创建其他3个工程。因为它们之间的依赖关系是：BusinessLogicLayer依赖于PersistenceLayer，PresentationLayer依赖于BusinessLogicLayer和PersistenceLayer，因此创建顺序应该是PersistenceLayer→BusinessLogicLayer→PresentationLayer。

1. PersistenceLayer工程

PersistenceLayer工程是静态链接库工程，具体创建过程如下：在Xcode中选择菜单File→New→Project...，在打开的对话框中选择Framework & Library→Cocoa Touch Static Library工程模板，如图7-62所示。

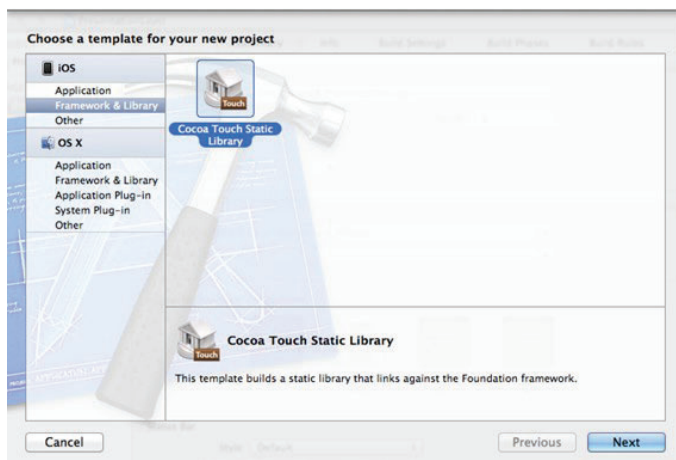


图7-62 静态链接库工程

输入名称PersistenceLayer，删除PersistenceLayer.h和PersistenceLayer.m文件，将上一节工程中的Note.h、Note.m、NoteDAO.h和NoteDAO.m文件添加到工程中。在iOS中，静态链接库文件为.a文件，在编译时要能找到这个.a文件，还要找到它们的.h文件（头文件）。在编译时静态链接库工程需要复制头文件，设定如下：打开PersistenceLayer工程，选择TARGETS→Build Phases，如图7-63所示，选择右下角的+按钮，从中选择Add Copy Headers项即可。

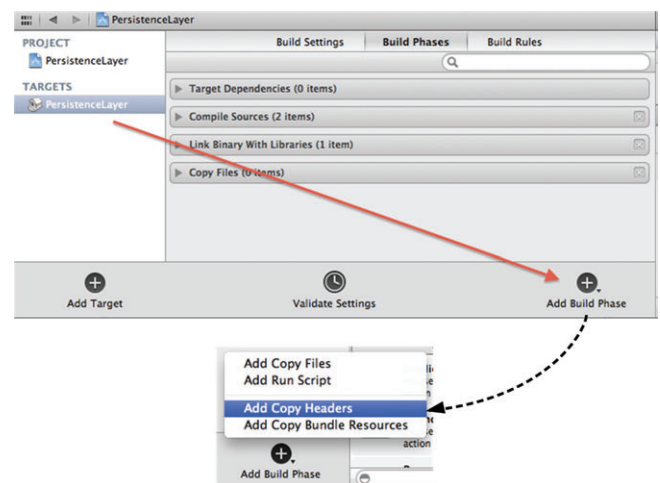


图7-63 复制头文件

此时会打开复制头文件窗口，选择图7-64中的+，从弹出界面中选择其中要复制的头文件，再点击Add按钮添加。

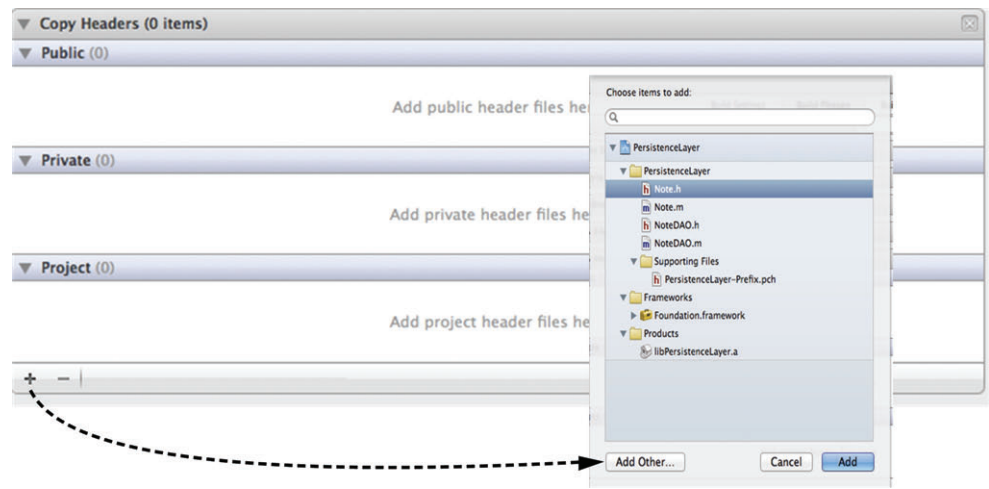


图7-64 选择要复制的头文件

接着用鼠标将复制的头文件从Project栏拖曳至Public栏中，如图7-65所示。

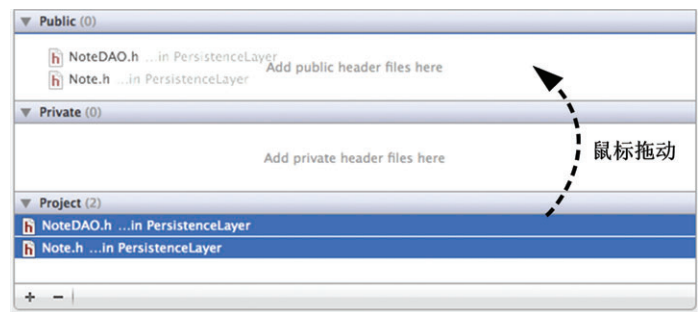


图7-65 拖曳复制的头文件到Public栏中

2. BusinessLogicLayer工程

BusinessLogicLayer工程也是静态链接工程，创建过程和复制头文件等设定都与PersistenceLayer工程类似，不同的是BusinessLogicLayer依赖于PersistenceLayer。打开BusinessLogicLayer工程，选择TARGETS→Build Phases→Link Binary With Libraries，如图7-66所示，选择左下角的+按钮，然后从弹出界面中选择libPersistenceLayer.a，再点击Add按钮，这样依赖关系就添加好了。

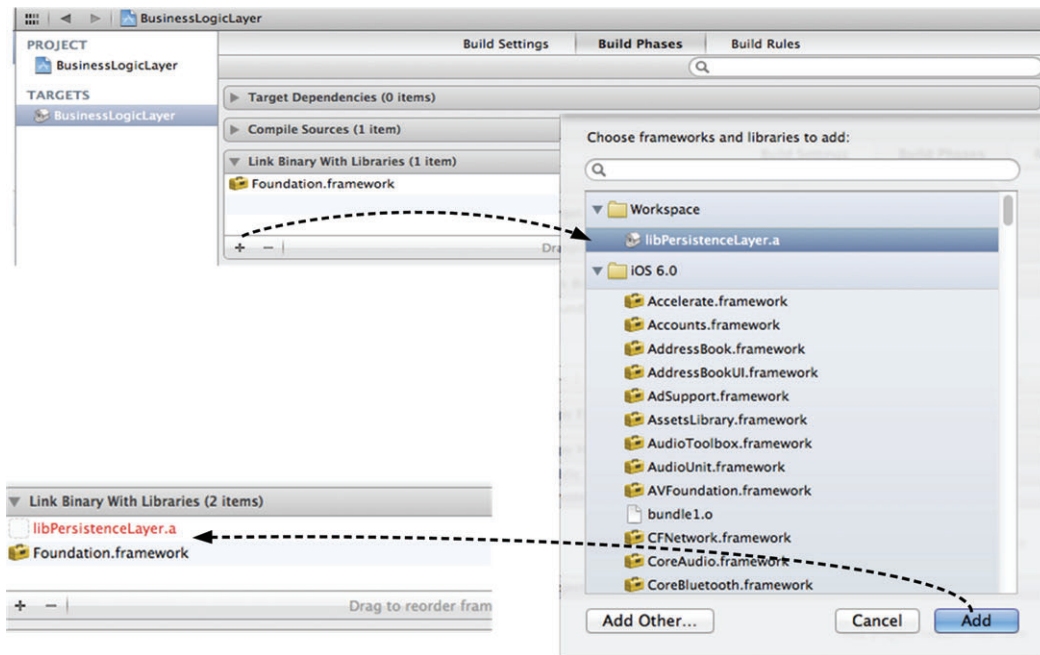


图7-66 添加依赖关系

此外，还要添加头文件搜索设置，具体操作方法是：打开BusinessLogicLayer工程，选择TARGETS→Build Settings→Search Paths→User Header Search Paths，输入内容\$(BUILT_PRODUCTS_DIR)，并选择recursive，如图7-67所示。

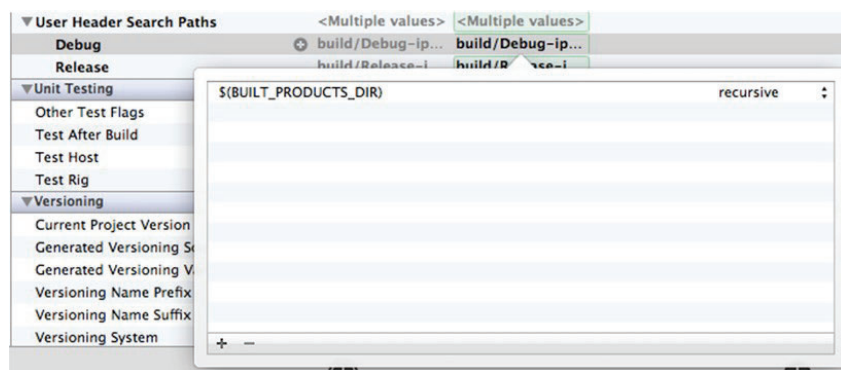


图7-67 添加头文件搜索设置

3. PresentationLayer工程

PresentationLayer不是静态链接工程，需要与BusinessLogicLayer和PersistenceLayer建立依赖关系，具体的操作

可方法参考BusinessLogicLayer工程。添加头文件搜索设置的具体方法，也可以参考BusinessLogicLayer工程。完成后的工作空间如图7-68所示。

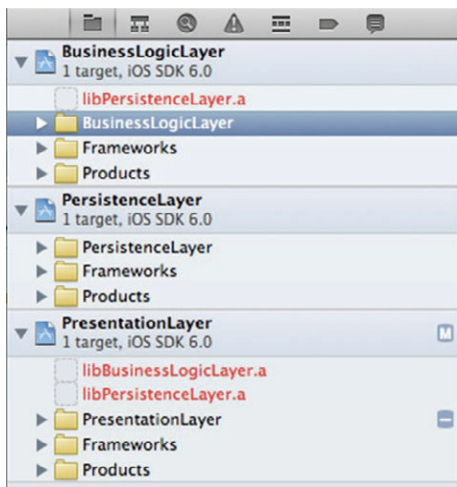


图7-68 完成后的工作空间

如果BusinessLogicLayer和PersistenceLayer工程中有内容要修改，因为不会自动编译，需要选择菜单Product→Clean清除一下再编译。

有时候业务比较复杂，用户对iPhone和iPad版本的UI要求差别很大，甚至功能上都有所不同，此时可以考虑将表示层分成iPhone版和iPad版两个不同的工程，它们之间的依赖关系如图7-69所示。

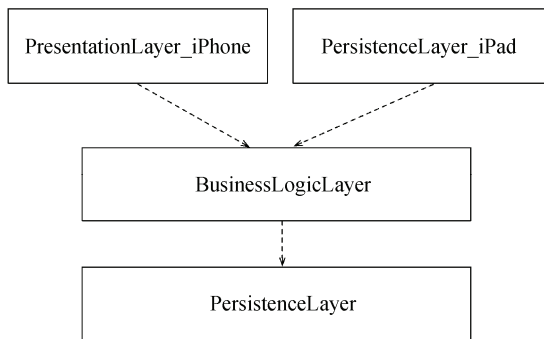


图7-69 iPhone版和iPad版的工程

7.6 小结

本章中，我们首先通过iPhone和iPad设备使用场景上的差异，了解了二者在设计和开发时存在差异的根由，然后深入介绍了iPad专用的API。为了掌握两种设备在应用开发中的不同，我们用nib和故事板技术分别实现了两个重要的程序模板。最后，我们介绍了iOS平台的分层架构设计技术，这种架构设计模式将贯穿全书（包括项目实战篇中的项目），希望读者能够重点学习。

我曾经见过我的同事将服务器的IP地址“硬编码”^①在程序中的尴尬事情，结果当用户服务器的IP地址发生变化时，程序就无法连接到服务器了，只能修改源代码，再重新编译。为了避免发生这种情况，我们需要在程序中添加一个能够让用户修改服务器IP的功能，这个功能叫做应用程序设置或配置。

8.1 概述

在iOS平台上，“设置”与“配置”是有区别的，下面我们将通过一些应用实例来进一步了解。

8.1.1 设置

设置中的项目在使用中是不经常变化的，它决定了应用的一些最基本的行为和特征，例如语言、货币、日期格式和新浪微博账户信息，等等。iOS将所有应用的设置集中在一起管理，可以通过iOS的“设置”应用进入。

图8-1是iPod touch自带的地图应用设置。点击桌面的“设置”图标，找到“地图”设置项目，点击进入地图应用设置，其中可以设置距离的单位。



图8-1 iPod touch自带的地图应用设置

在iPad中，无论是横屏还是竖屏，设置界面都分成两栏，如图8-2所示。设置项目分布在不同的“孤岛”中，功能类似的设置项目被集中在一个“孤岛”中。

^① 硬编码（hard code或hard coding）指的是在软件实现上，把输出或输入的相关参数（例如路径、输出的形式或格式）直接以常量的方式写在源代码中，而非在运行时由外界指定的设置、资源、数据或格式作出适当回应，具体可参见<http://zh.wikipedia.org/wiki/硬编码>。



图8-2 iPad的设置界面

8.1.2 配置

配置中的项目是经常变化的，例如游戏的音量、天气预报中的城市、地图的显示模式等。应用的配置和设置是不同的，设置是在应用之外单独有一个应用，而配置是在应用内部开辟出的一块功能。设置与配置之间的区别并非那么泾渭分明，一个项目放在配置或者设置中，并不会犯原则性错误。

在使用Xcode创建的实用型应用程序的主界面中，右下角有一个斜体的*i*按钮，它就是苹果官方给出的进入配置标准按钮。图8-3是iPod touch自带的天气预报应用配置，点击主界面的*i*按钮，界面会翻转到“背后”从而对天气预报的城市和温度单位（华氏度和摄氏度）进行修改。



图8-3 iPod touch自带的天气预报应用配置

图8-4为iPod touch自带的地图应用配置，它的右下角是翘起来的，用手指滑动它，就会像一张纸一样翻到一半，后面会露出来一些配置项目，包括设置地图模式和交通路口等内容。



图8-4 iPod touch自带的地图应用配置

配置是应用的一部分，它的界面布局、控件使用和事件处理等技术就是我们前面介绍的内容，而设置采用的技术是我们本章接下来要介绍的内容。

8.2 应用程序设置包

Settings Bundle是一个“包文件”，其中包含设置界面中所需的设置项目的描述、用到的图片、文字的本地化和子设置界面设置项目的描述等内容。

注意 本章中的“包文件”是Mac OS X下特有的文件，是可以用快捷菜单“显示包内容”打开的文件，包括本章要介绍的应用程序设置包文件和使用Xcode创建的工程文件。如图8-5所示，右击在Mac OS X的Finder中使用Xcode创建的工程文件（AppSetting.xcodeproj），从弹出的快捷菜单中选择“显示包内容”，就会打开包文件。

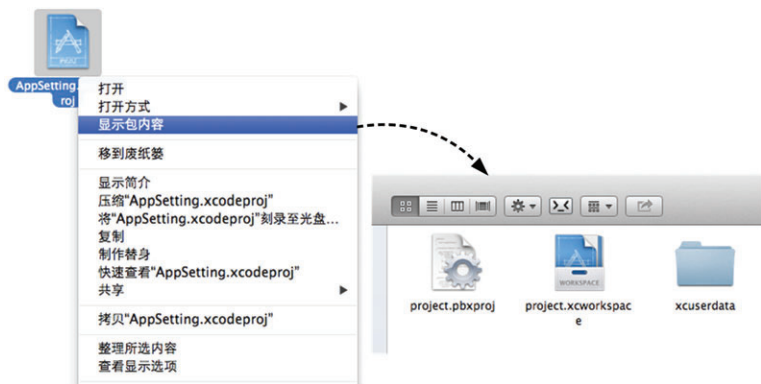


图8-5 打开包文件内容

我们可以使用Xcode工具在工程中创建应用程序设置包。在工程中选择File→New→File...菜单项，打开选择文件模板对话框，如图8-6所示，选择iOS→Resource→Settings Bundle模板，然后点击Next按钮，此时会弹出保存文件对话框，存放目录和文件名都采用默认值即可。

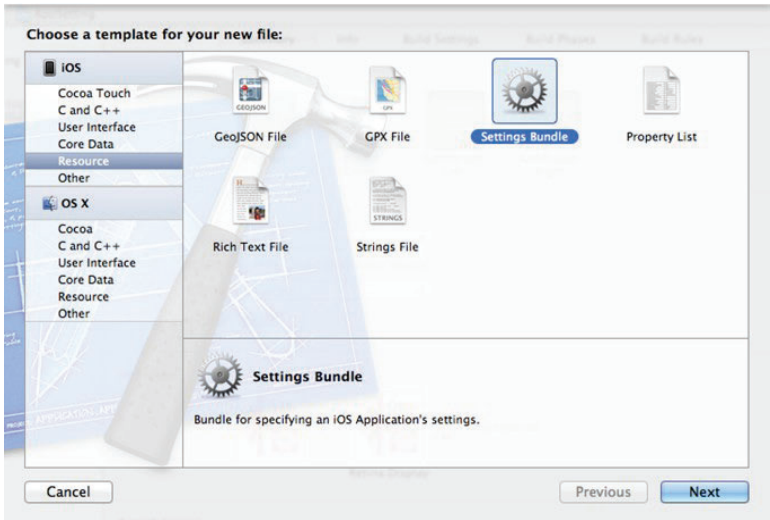


图8-6 创建应用程序设置包

此时在工程的导航面板中，就可以看到Settings.bundle文件（如图8-7所示）。这个文件是一个包文件，可以通过Xcode或Mac OS X操作系统中的Finder打开。

Settings.bundle文件中有Root.plist文件、en.lproj文件夹和Root.strings文件。Root.plist文件描述了根设置界面中设置的项目信息，它的命名必须是Root.plist。创建完Root.plist文件后，会生成一些内容，如图8-8所示，其中的每一个Item就是一个设置项目。关于设置项目的种类，我们会在下一节中介绍。如果需要，还可以有子设置界面描述信息文件，它的命名是可以自定义的。

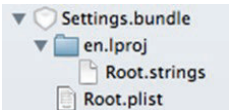
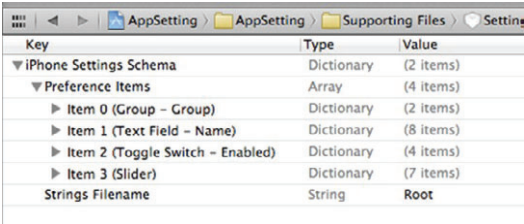


图8-7 Settings.bundle文件



Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
▼ Preference Items	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)
Strings Filename	String	Root

图8-8 模板生成的Root.plist文件的内容

en.lproj文件夹和Root.strings文件是与本地化有关的，用于设置界面文本信息的本地化。Root.strings文件的内容如下：

```
"Group" = "Group";
"Name" = "Name";
"none given" = "none given";
"Enabled" = "Enabled";
```

关于本地化相关的内容，我们会在后面的章节中介绍。

8.3 设置项目种类

运行我们刚刚创建的AppSetting应用，可以发现，屏幕上没有任何控件，这时按Home键退出，进入设置应用中。可以发现，在设置界面中出现了AppSetting的图标。点击AppSetting单元格，进入下一级设置界面，如图8-9所示。



图8-9 AppSetting应用的设置

在二级设置界面中,共有4个设置项,有些看起来像我们前面讲过的标准控件,那么二级界面是不是在Interface Builder中使用标准控件拖曳摆放的呢?答案是否定的,它们是由Settings.bundle中的Root.plist文件设定的。图8-10显示了AppSetting应用的设置项目与界面的对应关系。

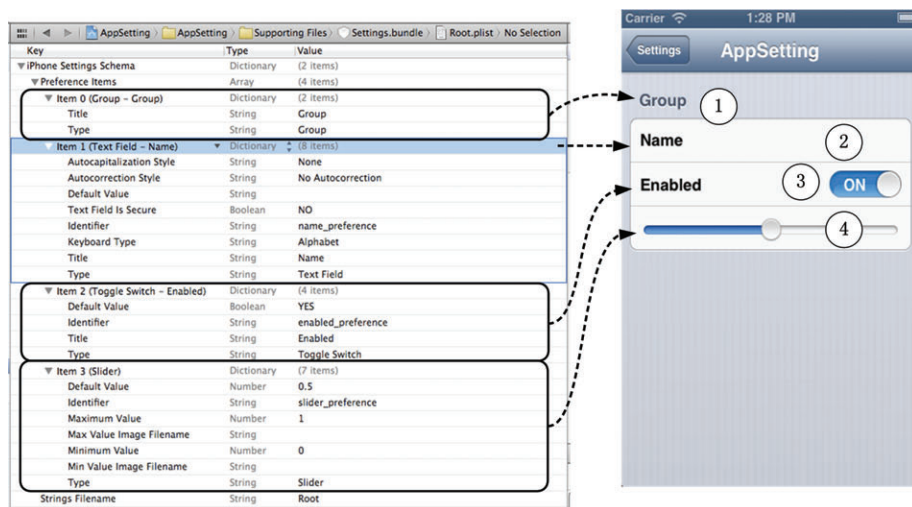


图8-10 AppSetting应用的设置项目与界面的对应关系

例如,④是滑块控件(不是UIKit中的UISlider类),它对应Root.plist文件中的Item3(Slider)。我们知道,.plist文件本质上是一种XML文件。使用记事本打开Root.plist文件时,可以发现其中描述这个滑块控件的代码如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>PreferenceSpecifiers</key>
  <array>
    .....
    <dict>
      <key>DefaultValue</key>
      <real>0.5</real>
      <key>Key</key>
```



```
        <string>slider_preference</string>
        <key>MaximumValue</key>
        <integer>1</integer>
        <key>MaximumValueImage</key>
        <string></string>
        <key>MinimumValue</key>
        <integer>0</integer>
        <key>MinimumValueImage</key>
        <string></string>
        <key>Type</key>
        <string>PSSliderSpecifier</string>
    </dict>
    .....
</array>
<key>StringsTable</key>
<string>Root</string>
</dict>
</plist>
```

AppSetting应用中的4个设置项目都有对应的XML文件内容加以描述，详见表8-1。

表8-1 设置项目

类 型	说 明
组	提供一个组，其他的设置项目放置在组中。当Type为PSGroupSpecifier时，指示该项目是一个新分组的开始，其后的每个项目都将是此分组的一部分，直到一个Type值为PSGroupSpecifier的项目之前
文本字段	提供一个文本字段。当Type为PSTextFieldSpecifier时，指示该项目是文本字段类型
标题	提供一个只读文本。当Type为PSTitleValueSpecifier时，指示该项目是标题类型
滑块	提供一个滑块控件设置项目。当Type为PSSliderSpecifier时，指示该项目是滑块类型
开关	提供一个开关控件设置项目。当Type为PSToggleSwitchSpecifier时，指示该项目是开关类型
值列表	给用户只能选择一个值的列表。当Type为PSMultiValueSpecifier时，指示该项目是值列表类型
子界面	会导航到下一级的设置界面。当Type为PSChildPaneSpecifier时，指示该项目是子界面类型

接下来，我们将重点介绍文本字段、滑块、开关、值列表和子界面设置项目。为了介绍这些设置项目，我们先看一个游戏应用需要设置的内容，见表8-2。

表8-2 游戏应用设置内容

设置的内容	采用设置项目	配置还是设置？
每月流量限制	滑块	设置
每月是否清除缓存	开关	设置
音效开启	开关	配置
背景音乐开启	开关	配置
服务器选择	值列表	设置
用户名	文本字段	设置
密码	文本字段	设置
通知（包括声音、振动开关）	子界面	设置

应用程序设置有一个最让人讨厌的问题，那就是你必须退出当前应用，从桌面进入设置应用对其进行设置，设置完毕之后再回到应用中去。因此，游戏音效和背景音乐等经常变化的内容应该在配置中。试想一下，如果你正在玩游戏，由于游戏的背景音乐比较吵，你想关掉它，此时背景音乐开关放在设置中，那么关闭它该是件多么麻烦的事情！由于配置是应用内部的一个功能，不用退出游戏就可以关闭了。

下面我们介绍这些设置项目的使用过程。图8-11是读取游戏应用的设置项目信息的界面，点击“读取设置数据”按钮就可以读取了。相关的设置界面如图8-12所示。



图8-11 读取游戏应用的设置项目



图8-12 游戏应用的设置界面

该案例中包含了文本字段、开关、滑块、值列表和子界面等类型，下面我们就开始介绍这些设置项目。

8.3.1 文本字段

在游戏案例中，用户名和密码属于文本字段，密码需要掩码显示。为了与其他设置相区别，我们把用户名和密码放到一个组中，如图8-13所示。对应的Root.plist文件中的设置项目如图8-14所示。



图8-13 文本字段设置项目

Item 0 (Group - 账户)	Dictionary	(2 items)
Title	String	账户
Type	String	Group
Item 1 (Text Field - 用户名)	Dictionary	(8 items)
Autocapitalization Style	String	None
Autocorrection Style	String	No Autocorrection
Default Value	String	
Text Field Is Secure	Boolean	NO
Identifier	String	name_preference
Keyboard Type	String	Alphabet
Title	String	用户名
Type	String	Text Field
Item 2 (Text Field - 密码)	Dictionary	(8 items)
Autocapitalization Style	String	None
Autocorrection Style	String	No Autocorrection
Default Value	String	
Text Field Is Secure	Boolean	YES
Identifier	String	password_preference
Keyboard Type	String	Alphabet
Title	String	密码
Type	String	Text Field

图8-14 Root.plist文本字段设置项目

在.plist文件中，有这样一个问题，那就是使用Xcode工具打开时，其内容与实际保存的内容不一致。下面的代码是使用记事本打开Root.plist时有关的3个设置项目：

```
<dict>
  <key>Title</key>
  <string>账户</string>
  <key>Type</key>
  <string>PSGroupSpecifier</string>
</dict>
<dict>
  <key>AutocapitalizationType</key>
  <string>None</string>
  <key>AutocorrectionType</key>
  <string>No</string>
  <key>DefaultValue</key>
  <string></string>
  <key>IsSecure</key>
  <false/>
  <key>Key</key>
  <string>name_preference</string>
  <key>KeyboardType</key>
  <string>Alphabet</string>
  <key>Title</key>
  <string>用户名</string>
  <key>Type</key>
  <string>PSTextFieldSpecifier</string>
</dict>
<dict>
  <key>AutocapitalizationType</key>
  <string>None</string>
  <key>AutocorrectionType</key>
  <string>No</string>
  <key>DefaultValue</key>
  <string></string>
  <key>IsSecure</key>
  <true/>
  <key>Key</key>
  <string>password_preference</string>
  <key>KeyboardType</key>
  <string>Alphabet</string>
  <key>Title</key>
  <string>密码</string>
  <key>Type</key>
  <string>PSTextFieldSpecifier</string>
</dict>
```

其中每个<dict>...</dict>对应着一个设置项目，其中的每个属性都是由一对<key>...</key>和<string>...</string>组成的，其中key是属性名，string是属性的取值。我们将图8-12所示的项目与上述代码进行比较，如图8-15和表8-3所示。

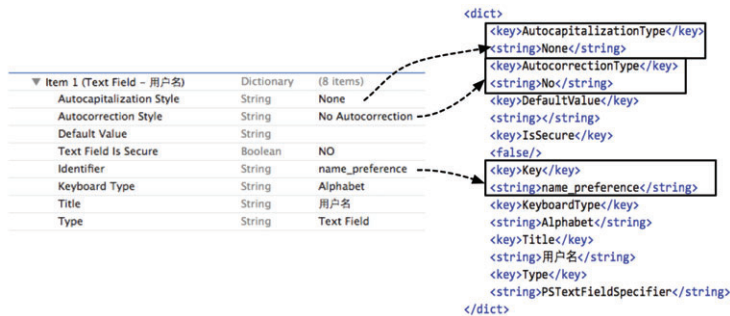


图8-15 Xcode工具和记事本打开的Root.plist文件

表8-3 Xcode工具和记事本打开的Root.plist文件的不同

Xcode工具	记 事 本
Autocapitalization Style	AutocapitalizationType
Autocorrection Style	AutocorrectionType
Identifier	Key

显而易见，Xcode工具并没有改变数据的本质，只是一种“障眼法”，它把这些项目变得易读。不仅是应用设置中的.plist是这样的，其他的一些.plist文件也是如此。根据用户的习惯，有的用户不喜欢这种“障眼法”，那怎么去掉呢？你只需在Xcode快捷菜单中选择Show Raw Keys/Values即可实现互相切换，如图8-16所示。

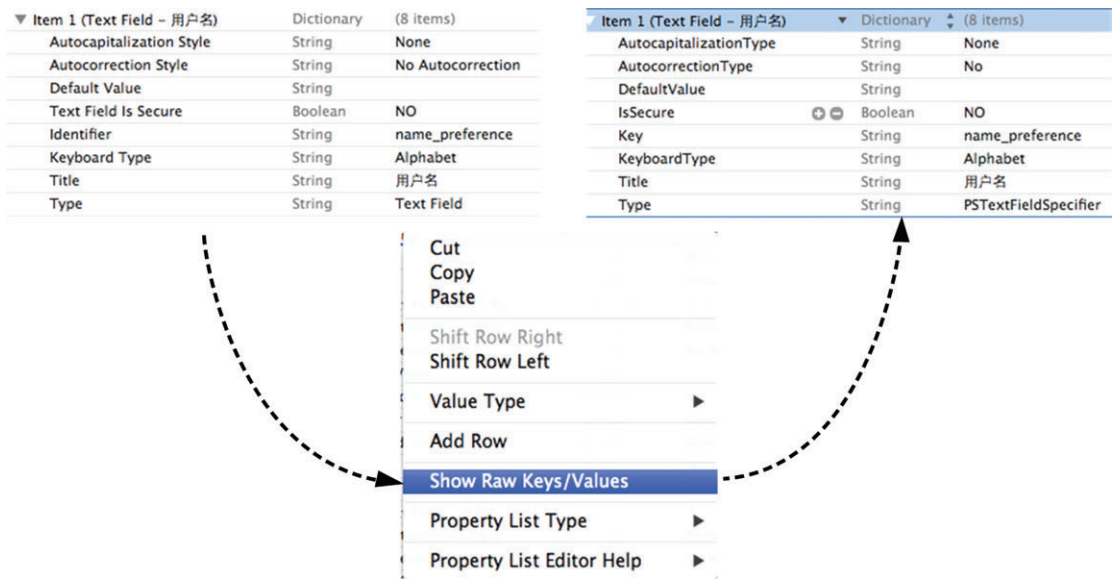


图8-16 切换设置显示

文本字段设置项目是可以输入内容的，它有很多属性，具体如表8-4所述。

表8-4 文本字段设置属性说明

Key属性	Value属性	说 明
Type	PSTextFieldSpecifier	文本字段类型
Title	Name	项目标题
Key	name_preference	项目的key
DefaultValue	空	默认值
IsSecure	false	是否为安全字段
KeyboardType	Alphabet	字母键盘
AutocapitalizationType	None	是否自动大写
AutocorrectionType	No	是否拼写纠正

8.3.2 开关

在游戏案例中，开关设置项目被单独放置在一组中，示例图如图8-17所示。

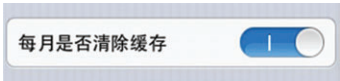


图8-17 开关设置项目

对应的Root.plist文件中的设置项目如图8-18所示。

▼ Item 3 (Group -)	Dictionary	(2 items)
Title	String	
Type	String	Group
▼ Item 4 (Toggle Switch - 每月是否清除缓存)	Dictionary	⬆ (4 items)
Default Value	Boolean	YES
Identifier	String	enabled_preference
Title	String	每月是否清除缓存
Type	String	Toggle Switch

图8-18 Root.plist开关设置项目

下面的代码是使用记事本打开Root.plist时有关的2个设置项目：

```
<dict>
    <key>Title</key>
    <string> </string>
    <key>Type</key>
    <string>PSGroupSpecifier</string>
</dict>
<dict>
    <key>DefaultValue</key>
    <true/>
    <key>Key</key>
    <string>enabled_preference</string>
    <key>Title</key>
    <string>每月是否清除缓存</string>
    <key>Type</key>
    <string>PSToggleSwitchSpecifier</string>
</dict>
```

开关设置项目很多属性，其说明如表8-5所述。

表8-5 开关设置属性说明

Key/属性	Value/属性	说 明
Type	PSToggleSwitchSpecifier	开关类型
Title	每月是否清除缓存	项目的标题
Key	enabled_preference	项目的key
DefaultValue	True	默认值

8.3.3 滑块

在游戏案例中，滑块设置项目也被单独放置在一组中，示例图如图8-19所示。

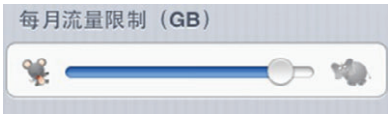


图8-19 滑块设置项目

对应的Root.plist文件中的设置项目如图8-20所示。

▼ Item 5 (Group – 每月流量限制)	Dictionary	(2 items)
Title	String	每月流量限制 (GB)
Type	String	Group
▼ Item 6 (Slider)	Dictionary	(7 items)
Default Value	Number	0.5
Identifier	String	slider_preference
Maximum Value	Number	1
Max Value Image Filename	String	elephant.png
Minimum Value	Number	0
Min Value Image Filename	String	mouse.png
Type	String	Slider

图8-20 Root.plist滑块设置项目

下面是使用记事本打开Root.plist时有关的2个设置项目的代码：

```
<dict>
  <key>Title</key>
  <string>每月流量限制 (GB) </string>
  <key>Type</key>
  <string>PSGroupSpecifier</string>
</dict>
<dict>
  <key>DefaultValue</key>
  <real>0.5</real>
  <key>Key</key>
  <string>slider_preference</string>
  <key>MaximumValue</key>
  <integer>1</integer>
  <key>MaximumValueImage</key>
  <string>elephant.png</string>
  <key>MinimumValue</key>
  <integer>0</integer>
  <key>MinimumValueImage</key>
  <string>mouse.png</string>
  <key>Type</key>
  <string>PSSliderSpecifier</string>
</dict>
```

滑块设置项目有很多属性，详见表8-6。

表8-6 滑块设置属性说明

Key/属性	Value/属性	说 明
Type	PSSliderSpecifier	滑块类型
Key	slider_preference	项目的key
DefaultValue	0.5	默认值
MinimumValue	0	最小值
MaximumValue	1	最大值
MinimumValueImage	mouse.png	最小值图片
MaximumValueImage	elephant.png	最大值图片

需要注意的是，将最大值图片和最小值图片添加到设置项目中的操作与以前在Xcode工程中添加文件有区别，我们要到Finder下面打开应用设置包Settings.bundle，如图8-21所示。

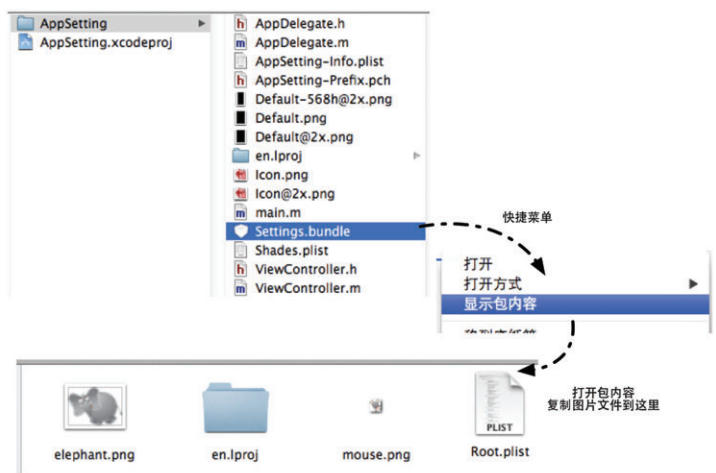


图8-21 为设置包添加图片文件

这种向设置包添加文件的方式也适用于其他文件添加，这里的图片格式最好是PNG。

8.3.4 值列表

值列表能够给用户提供了多选的列表。在游戏案例中，值列表设置项目与通知放置在一组中，如图8-22所示。选择值列表设置项目（比如服务器），就会导航到新的列表选择界面，这里只能单选。



图8-22 值列表设置项目

对应的Root.plist文件中的设置项目如图8-23所示。

▼ Item 8 (Multi Value – 服务器)	▼ Dictionary (6 items)
► Values	Array (3 items)
Type	String Multi Value
Title	String 服务器
► Titles	Array (3 items)
Identifier	String multivaule_preference
Default Value	String 上海服务器

图8-23 Root.plist值列表设置项目

下面是使用记事本打开Root.plist时有关设置项目的代码：

```
<dict>
  <key>Values</key>
  <array>
    <string>上海服务器</string>
    <string>北京服务器</string>
    <string>广州服务器</string>
  </array>
  <key>Type</key>
  <string>PSMultiValueSpecifier</string>
  <key>Title</key>
  <string>服务器</string>
  <key>Titles</key>
  <array>
    <string>上海服务器</string>
    <string>北京服务器</string>
    <string>广州服务器</string>
  </array>
  <key>Key</key>
  <string>multivaule_preference</string>
  <key>DefaultValue</key>
  <string>上海服务器</string>
</dict>
```

关于值列表设置项目的属性，详见表8-7。

表8-7 值列表设置属性说明

Key属性	Value属性	说 明
Type	PSMultiValueSpecifier	值列表类型
Key	multivaule_preference	项目的key
Title	服务器	项目的标题
Titles	字符串数组（上海服务器、北京服务器、广州服务器）	选项标题
Values	字符串数组（上海服务器、北京服务器、广州服务器）	选项值
DefaultValue	上海服务器	默认值

8.3.5 子界面

使用子界面，可以导航到下一级的设置界面。子界面的文件结构与Root.plist一样，它们之间构成“子父”关系，这就是多层设置界面了，如图8-24所示。

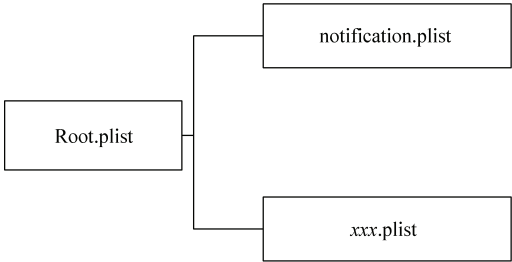


图8-24 多层设置界面

游戏案例中包含了子界面设置项目（比如通知），如图8-25所示，使用它可以导航到新的列表选择子界面。



图8-25 子界面设置项目

对应的Root.plist文件中的设置项目如图8-26所示。

Item 9 (Child Pane - 通知)		
Filename	String	Notification
Type	String	Child Pane
Title	String	通知

图8-26 Root.plist子界面设置项目

下面是使用记事本打开Root.plist时有关设置项目的代码：

```
<dict>
  <key>File</key>
  <string>Notification</string>
  <key>Type</key>
  <string>PSChildPaneSpecifier</string>
  <key>Title</key>
  <string>通知</string>
</dict>
```

关于子界面设置项目的属性，详见如表8-8所述。

表8-8 子界面设置属性说明

key属性	value属性	说 明
Type	PSChildPaneSpecifier	子界面类型
File	Notification	设置子界面的.plist文件
Title	通知	项目的标题

子界面设置文件Notification.plist如图8-27所示。

Key	Type	Value
Root	Dictionary (2 items)	
PreferenceSpecifiers	Array (3 items)	
Item 0	Dictionary (2 items)	
Title	String	
Type	String	PSGroupSpecifier
Item 1	Dictionary (4 items)	
DefaultValue	Boolean	YES
Key	String	sound_enabled_preference
Title	String	声音
Type	String	PSToggleSwitchSpecifier
Item 2	Dictionary (4 items)	
DefaultValue	Boolean	YES
Key	String	vibrate_enabled_preference
Title	String	振动
Type	String	PSToggleSwitchSpecifier
StringsTable	String	Root

图8-27 子界面设置文件Notification.plist

在这个文件中，描述了两个开关设置项目。此外，Notification.plist需要像图片一样放入设置包中。

8.4 读取设置

设置完成后，我们需编写代码读取这些内容，这可以通过NSUserDefaults类来实现。要想获得NSUserDefaults实例，可以采用单例设计模式，相关代码如下：

```
NSUserDefaults* defaults = [NSUserDefaults standardUserDefaults];
```

NSUserDefaults这个类我们在第3章中介绍过，接着我们将详细介绍一些与应用设置有关的方法，下面是它的一些取值方法。

- ❑ **boolForKey:**。根据键取出布尔值。
- ❑ **floatForKey:**。根据键取出float值。
- ❑ **integerForKey:**。根据键取出NSInteger值。
- ❑ **objectForKey:**。根据键取出id类型值。
- ❑ **stringForKey:**。根据键取出NSString类型值。
- ❑ **doubleForKey:**。根据键取出double类型值。

就应用设置而言，使用NSUserDefaults上面的取值方法就足够了，一般不需要使用NSUserDefaults赋值方法。

下面我们看看代码部分。ViewController.h中的代码如下：

```
@interface ViewController : UITableViewController

//用户名
@property (weak, nonatomic) IBOutlet UILabel *username;
//密码
@property (weak, nonatomic) IBOutlet UILabel *password;
//每月是否清除缓存
@property (weak, nonatomic) IBOutlet UILabel *clearCache;
//每月流量限制
@property (weak, nonatomic) IBOutlet UILabel *flowmeter;
//服务器
@property (weak, nonatomic) IBOutlet UILabel *serverName;
//声音
@property (weak, nonatomic) IBOutlet UILabel *notiSound;
```

```
//震动
@property (weak, nonatomic) IBOutlet UILabel *notiVibrate;
```

```
//读取设置数据
- (IBAction)getData:(id)sender;
@end
```

ViewController.m中的主要代码如下:

```
@implementation ViewController

- (void)viewWillAppear:(BOOL)animated
{
    [self getData:nil];
}

- (IBAction)getData:(id)sender {

    NSUserDefaults* defaults = [NSUserDefaults standardUserDefaults];

    self.username.text = [defaults stringForKey:@"name_preference"];
    self.password.text = [defaults stringForKey:@"password_preference"];

    if ([defaults boolForKey:@"enabled_preference"]) {
        self.clearCache.text = @"YES";
    } else
    {
        self.clearCache.text = @"NO";
    }
    self.flowmeter.text = [NSString stringWithFormat:@"%0.2fGB", [defaults
        doubleForKey:@"slider_preference"]];

    self.serverName.text = [defaults stringForKey:@"multivaule_preference"];

    if ([defaults boolForKey:@"sound_enabled_preference"]) {
        self.notiSound.text = @"YES";
    } else
    {
        self.notiSound.text = @"NO";
    }

    if ([defaults boolForKey:@"vibrate_enabled_preference"]) {
        self.notiVibrate.text = @"YES";
    } else
    {
        self.notiVibrate.text = @"NO";
    }
}
@end
```

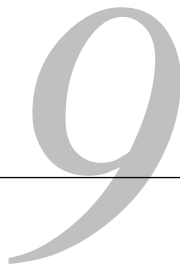
我们首先在viewWillAppear:方法中读取这些设置项目, 这样在界面一出现时就会读取出来。也可以通过点击按钮, 重新取得设置项目。

8.5 小结

在本章中, 我们首先介绍了应用中设置和配置的概念, 然后通过对二者差异的探讨, 介绍了什么样的项目适合放在设置里, 什么样的项目适合放在配置里, 并重点介绍了设置的实现及读取。

第9章

应用程序本地化



在本章中，我们从文本信息、nib和故事板文件、资源文件这3个方面来了解iOS本地化的相关技术。

9.1 概述

本地化就是使应用适合于本地区用户习惯的过程。应用软件是否需要本地化，需要提供多少种语言（或地区）的本地化支持，这些问题我们不能简单地一概而论。如果应用就是给国人开发的，例如开发的是一个新浪微博应用，只需要提供简体中文和繁体中文就可以了。但如果开发的是《愤怒的小鸟》这样的游戏，我们就需要提供几乎全世界所有语言的本地化支持了。事实上，本地化的工作是巨大而烦琐的，本地化到什么程度与应用本身有很大的关系。

9.1.1 本地化内容

本地化涉及很多方面，比如文字、货币、日期和图片，甚至语音都需要本地化。即便是同一种文字，也有不同的语言习惯，例如英国英语和美国英语是不同的。另外，同一个地区也可能有不同的语言。

目前，iOS中提供的本地化API包括文本信息本地化、nib和故事板文件本地化以及资源文件本地化。

❑ 文本信息本地化。它是首先被考虑的，包括应用的名称、按钮、警告提示信息以及界面中显示的静态文字等。

❑ nib和故事板文件本地化。同一个界面和场景可以提供多个本地化版本的nib和故事板文件。

❑ 资源文件本地化。它包括图片和音频等资源的本地化。图片本地化包括应用图标和一般图片的本地化。资源文件的本地化要根据具体情况而定，否则工作量还是很大的。

下面我们看看苹果提供的一个应用中的本地化情况。图9-1是iPod touch自带的天气预报应用，语言设置为“英文”，区域格式为“美国”。图9-2是语言设置为“简体中文”、区域格式为“中国”的天气预报应用。



图9-1 天气预报应用

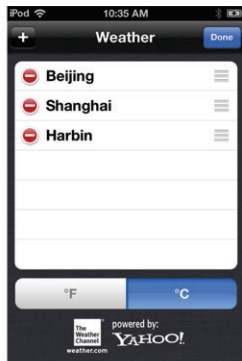


图9-2 语言设置为“简体中文”、区域格式为“中国”的天气预报应用

对比图9-1和图9-2可以看到，界面中的文字部分都本地化了。需要注意的是Done（“完成”）按钮，它是系统提供的按钮，也会自动本地化。

图9-3是iPod touch自带的日历应用程序，其中左图中语言设置为“简体中文”，区域格式为“中国”，右图中语言设置为“英文”，区域格式为“美国”。对比这两个图可以发现，除了要将文字部分本地化外，还要本地化日期格式，而日期格式的本地化与区域格式的设置有关。



图9-3 日历应用

提示 如何改变iOS语言设置和区域格式呢？它们是在“设置”应用中完成的，其中图9-4是语言设置，图9-5是区域格式设置。



图9-4 设置语言



图9-5 设置区域格式

9.1.2 本地化目录结构

进行本地化处理时，支持本地的文件都放在特定目录下，这些目录的命名是“<本地代号>.lproj”，例如en-US.lproj、en_GB.lproj和en.lproj等。

本地代号由“语言代号+国家（或地区）代号”组成，例如en-US中的en是语言代号，US是国家代号。语言代号可以遵守ISO 639-1和ISO 639-2代号规范^①，ISO 639-1是两位代号，ISO 639-2是三位代号，例如英语ISO 639-1是en，ISO 639-2是eng。事实上，语言代号可能会更加复杂。现在有一些新的规范，例如中文分为简体中文和繁体中文，简体中文为zh-Hans，繁体中文为zh-Hant。在OS X v10.4之后，中文主要采用这个规范。

国家（或地区）代号遵守ISO 3166-1代号规范^②，例如US是美国，CN是中国，FR是法国，GB是英国。有时候国家（或地区）代号有可能省略，例如en.lproj和zh-Hans.lproj只有语言代号，这样应用程序会先搜索最精确的en-US.lproj目录，没有找到，则找en.lproj目录。

下面是一个应用的目录结构：

```
L10N应用
├── L10N-Info.plist
├── L10N-Prefix.pch
├── L10NAppDelegate.h
├── L10NAppDelegate.m
├── L10NViewController.h
├── L10NViewController.m
├── en.lproj
│   ├── InfoPlist.strings
│   └── L10NViewController.xib
├── es.lproj
│   ├── InfoPlist.strings
│   └── L10NViewController.xib
├── fr.lproj
│   ├── InfoPlist.strings
│   └── L10NViewController.xib
├── main.m
├── zh-Hans.lproj
│   ├── InfoPlist.strings
│   └── L10NViewController.xib
└── zh-Hant.lproj
    ├── InfoPlist.strings
    └── L10NViewController.xib
```

9.2 文本信息本地化

在本地化工作中，文本信息本地化占有很大的比例，其中包括应用名称本地化、系统按钮和信息本地化以及静态文本信息本地化。

9.2.1 系统按钮和信息本地化

还记得天气预报应用背后的“完成”按钮吗？如图9-1和图9-2所示，它在中文环境下是“完成”，在英语环境下是Done。

此外，还有一些系统给我们的提示信息，比如图9-6是连接蓝牙设备时的系统提示，它在中文环境下是中文提

① 详情可参见http://www.loc.gov/standards/iso639-2/php/English_list.php。

② 详情可参见<http://www.iso.ch>。

示，在英语环境下是英文提示。

实际上，系统按钮上的文本和系统提示信息的文字都是不能修改的，所以如果我们不进行本地化设置，即便是这些基本信息，也一直都是英文显示的。我们可以尝试在故事板中创建如图9-7所示的界面，在导航栏中放置两个系统按钮Done和Edit，然后分别在英文和中文环境下运行，看看是否有变化。

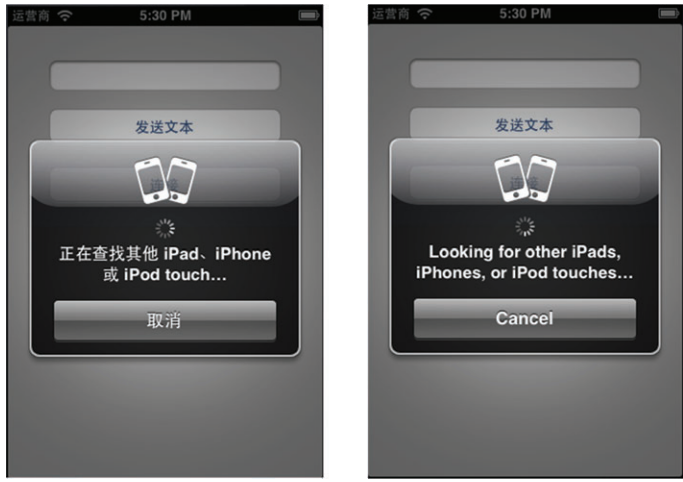


图9-6 连接蓝牙设备时的系统提示

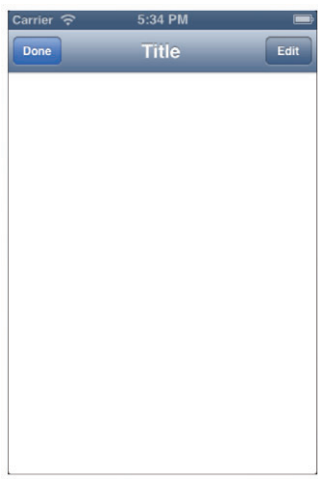


图9-7 系统按钮和信息的本地化

事实上，它们一直都没有变化，原因就在于我们没有对工程进行本地化设置。如图9-8所示，打开工程中的PROJECT项，选择L10N，点击Localizations下面的+，从弹出的快捷菜单中选择Chinese(zh-Hans)，这样就添加了简体中文本地化文件。

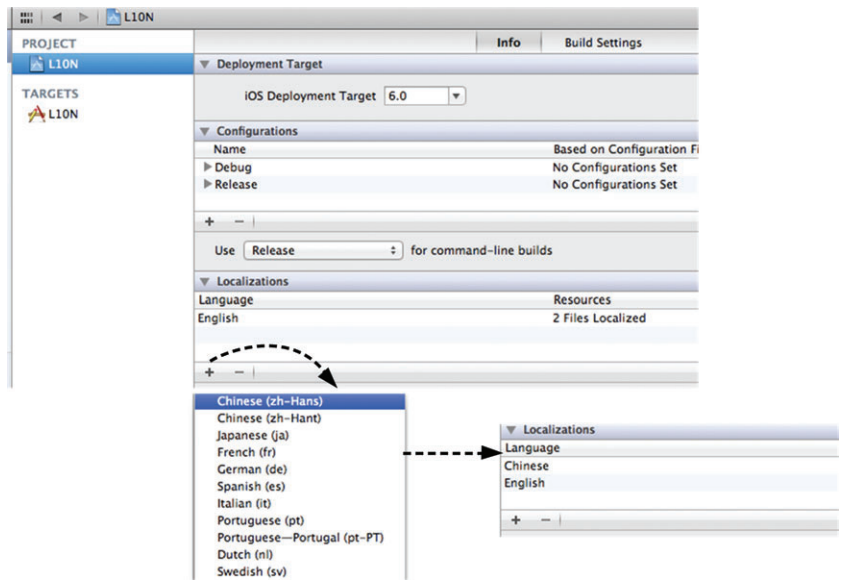


图9-8 对工程进行本地化设置

此时我们的工程就同时支持中文和英文的本地化了，系统按钮和提示信息等也都实现本地化了，这个过程不需要编写任何代码。

9.2.2 应用名称本地化

应用名称本地化是一个很重要的问题。如图9-9所示，左图是中文语言环境下的iPod touch桌面，右图是英文语言环境下的iPod touch桌面，我们看到日历、地图、股市等几个应用的名称都本地化了。



图9-9 应用名称本地化

下面我们编写一个应用，它的英文名是Localization，中文名是“本地化”。打开工程L10N应用，找到工程中的L10N-Info.plist文件，该文件是工程属性文件，应用程序名称就是在这个文件中定义的。如果我们要想本地化，必须借助另一个文件InfoPlist.strings。在本地化系统按钮和信息后，InfoPlist.strings下面会有两个文件：InfoPlist.strings(English)和InfoPlist.strings(Chinese)。打开Finder文件，可以看到en.lproj和zh-Hans.lproj这两个文件，其目录结构如下：

```

├── en.lproj
│   ├── InfoPlist.strings
│   └── MainStoryboard.storyboard
└── zh-Hans.lproj
    ├── InfoPlist.strings
    └── MainStoryboard.storyboard
  
```

在InfoPlist.strings文件中，通过CFBundleDisplayName和CFBundleName键能够配置应用名字，其中CFBundleDisplayName键用于配置应用显示的名字，CFBundleName键配置应用的短名字，不超过16个字符，用于显示菜单栏和应用窗口信息。

InfoPlist.strings（Chinese）文件的内容如下：

```

CFBundleDisplayName="本地化";
CFBundleName="本地化";
  
```

InfoPlist.strings（English）文件的内容如下：

```

CFBundleDisplayName="Localization";
CFBundleName="L10N";
  
```

图9-10是运行结果，图标下的文字是CFBundleDisplayName键配置的名字。

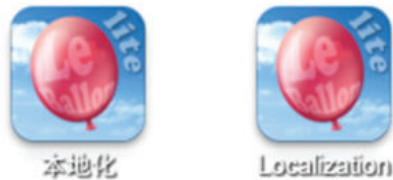


图9-10 应用在桌面上显示的名字

9.2.3 程序代码输出的静态文本本地化

应用中的静态文本都应该实现本地化，但是它们可能是通过程序代码输出的，也可能是通过Interface Builder在nib或故事板设计输出的。图9-11是采用Tabbed Application工程模板创建的标签应用程序，它的两个标签上的标题First和Second以及界面中的文字都属于静态文本。



图9-11 使用Tabbed Application模板创建的工程界面

同样都是这个工程，如果在创建过程中分别创建基于故事板和nib技术的两个版本，故事板版本的两个标签上的标题是通过Interface Builder在故事板文件中添加的，但在nib版本中两个标签上的标题通过程序代码输出的。关于故事板和nib中静态文本的本地化，我们会在下一节中介绍。

FirstViewController.m中构造方法的代码如下所示：

```
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        self.title = NSLocalizedString(@"First", @"First");
        self.tabBarItem.image = [UIImage imageNamed:@"first"];
    }
    return self;
}
```

SecondViewController.m中构造方法的代码如下所示：

```
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        self.title = NSLocalizedString(@"Second", @"Second");
        self.tabBarItem.image = [UIImage imageNamed:@"second"];
    }
    return self;
}
```

在这两个构造方法中设置标题属性时，我们使用了`NSLocalizedString`宏，该宏本质上是调用`NSBundle`的`localizedStringForKey:value:table:`方法，从默认字符串资源文件（`Localizable.strings`）中取出本地化的字符串。

字符串资源文件默认命名为`Localizable.strings`，采用UTF-16编码。如果静态文本不是很多，则可以自己创建`Localizable.strings`文件。选择Supporting Files组，打开File→New→File...菜单项，选择iOS→Resource→Strings File，输入文件名`Localizable.strings`，如图9-12所示。

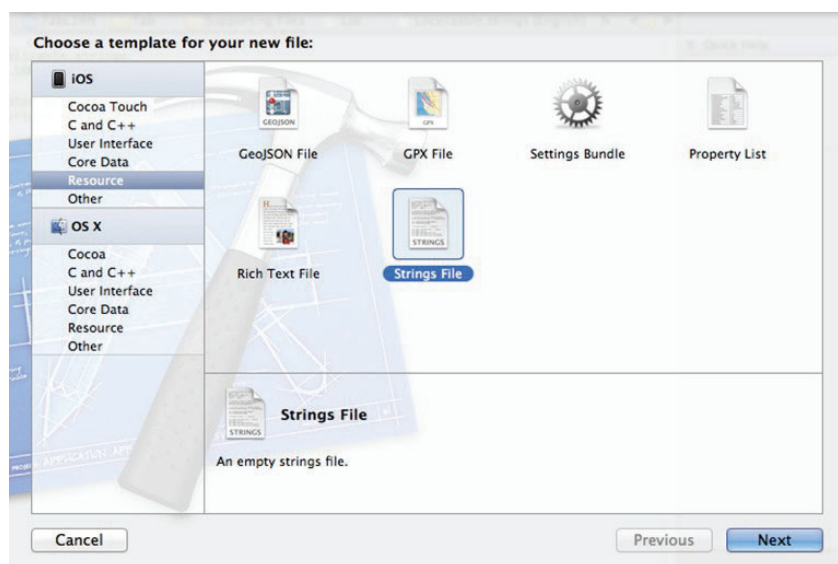


图9-12 Strings File文件模板

选择`Localizable.strings`文件，打开其文件检查器，接着点击Localization中的Make localized按钮（如图9-13所示），此时会弹出选择本地化语言对话框，如图9-14所示，这里我们选择English，然后点击Localize按钮。这可以帮助我们创建本地化的`Localizable.strings`文件。

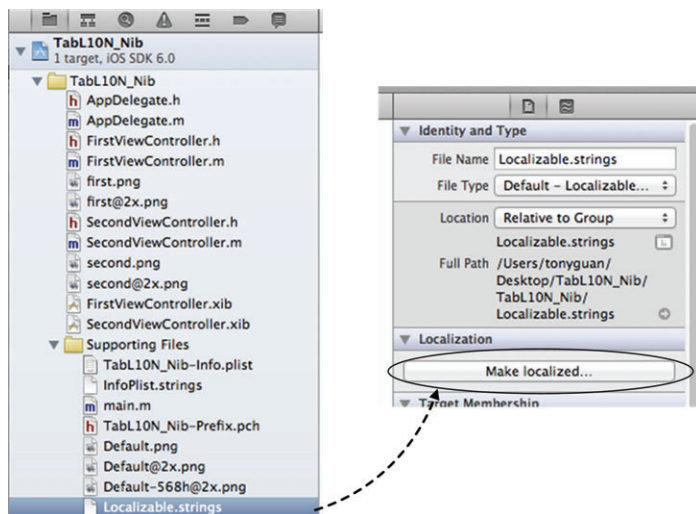


图9-13 文件检查器

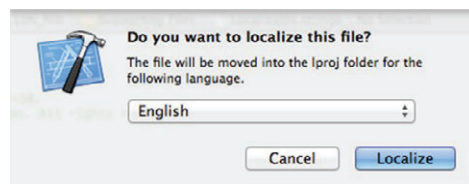


图9-14 选择本地化语言对话框

然后再按照图9-8所示，添加简体中文本地化文件，如图9-15所示。

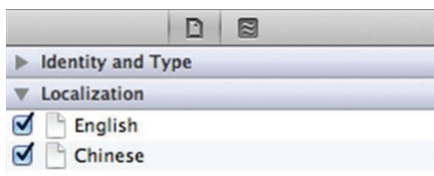


图9-15 添加Localizable.strings简体中文本地化

在英文版的Localizable.strings文件中添加以下内容：

```
/* First */
"First" = "First";
/* Second */
"Second" = "Second";
```

在中文版的Localizable.strings文件中添加以下内容：

```
/* First */
"First" = "第一";
/* Second */
"Second" = "第二";
```

9.2.4 使用genstring工具

在字符串很多时，可以借助命令行工具genstring，从.m或.mm文件中扫描下面宏中的某一个，并取出字符串，输出到本地化文件中：

```
CFCopyLocalizedString
CFCopyLocalizedStringFromTable
CFCopyLocalizedStringFromTableInBundle
CFCopyLocalizedStringWithDefaultValue
NSLocalizedString
NSLocalizedStringFromTable
NSLocalizedStringFromTableInBundle
NSLocalizedStringWithDefaultValue
```

在上面的宏中，以CF开头的宏和以NS的开头宏两两对应，后者属于Foundation框架，是基于Objective-C语言的，前者属于Core Foundation框架，是基于C语言的。NSLocalizedStringFromTable和NSLocalizedStringFromTableInBundle函数在自定义字符串资源文件名时使用。

下面是genstrings命名的基本语法：

```
genstrings [-a] [-q] [-o <outputDir>] sourcefile
```

其中参数-a用于在已有文件后面追加内容，-q用于关闭多个键/值对的警告，-o用于指定输出目录。

因此，如果想输出到en.lproj目录，可以使用如下代码：

```
genstrings -o en.lproj *.m
```

这样就在en.lproj目录下面产生了Localizable.strings文件。需要注意的是，每次运行上面的命令时，都会覆盖Localizable.strings文件。如果不想覆盖该文件，可以使用-a参数，然后在文件中进行修改。

9.3 nib 和故事板文件本地化

当采用Tabbed Application工程模板创建标签应用程序时，如果采用故事板创建，我们会发现标签上的标题First和Second是通过Interface Builder硬编码到故事板中的，如图9-16所示。

除了标签标题外，First View和Second View等静态文本也需要本地化，如图9-17所示。



图9-16 标签标题静态文本

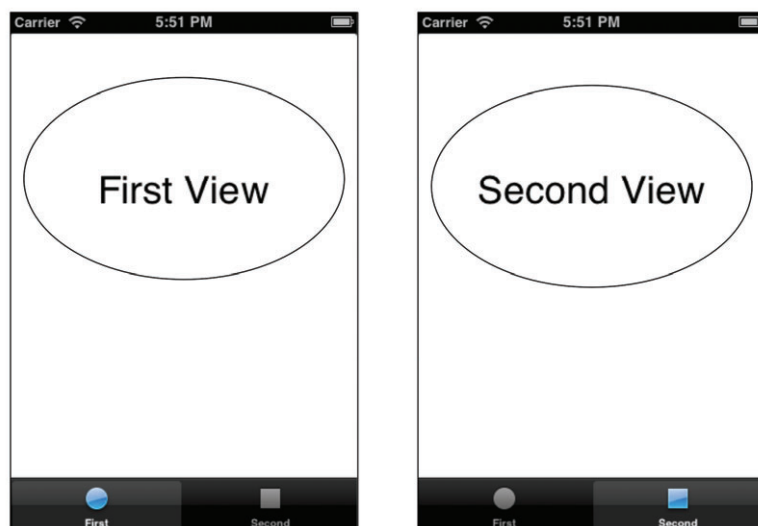


图9-17 故事板中的静态文本

与故事板一样，nib实现也可以进行本地化，每个本地化版本也可以有自己的静态文本。

9.3.1 添加本地化

不管是通过nib技术还是故事板技术创建的应用，添加本地化支持的操作是一样的，因此我们只介绍使用故事板技术的情况。按照图9-8所示添加简体中文本地化文件，此时生成的故事板文件如图9-18所示。

接下的事情就是分别在两个不同的故事板中进行设计，把静态文本修改成本地代码，如图9-19所示。

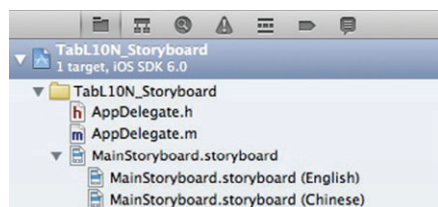


图9-18 故事板文件



图9-19 把静态文本修改成本地代码

9.3.2 开关使用ibtool工具

如果在nib或故事板中需要本地化的静态文本很多，而且需要支持的本地化语言也很多，那么工作量会变得很大，此时可以借助命令行工具ibtool。使用该工具，可以帮助我们提取出需要本地化的静态文本，并将其输出到一个文本文件中，然后再使用它生成其他语言版本的nib或故事板文件。

使用ibtool提取需要本地化的静态文本的代码如下：

```
ibtool --generate-strings-file MainStoryboard.strings en.lproj/  
MainStoryboard.storyboard
```

其中参数--generate-strings-file表示生成字符串资源文件，MainStoryboard.strings是要生成的资源文件，MainStoryboard.storyboard是英文版本的故事板文件。进入终端，保证在工程目录下运行上面代码命令，则生成MainStoryboard.strings，它的内容如下：

```
/* Class = "IBUIViewController"; title = "First"; ObjectID = "2"; */  
"2.title" = "First";  
  
/* Class = "IBUIViewController"; title = "Second"; ObjectID = "3"; */  
"3.title" = "Second";  
  
/* Class = "IBUITabBarItem"; title = "Second"; ObjectID = "6"; */  
"6.title" = "Second";  
  
/* Class = "IBUITabBarItem"; title = "First"; ObjectID = "7"; */  
"7.title" = "First";  
  
/* Class = "IBUILabel"; text = "First View"; ObjectID = "20"; */  
"20.text" = "First View";  
  
/* Class = "IBUILabel"; text = "Second View"; ObjectID = "22"; */  
"22.text" = "Second View";
```

这个文件不用解释，我们可以看出故事板中控件上面的静态文本信息，而且生成的代码中都有对应控件的注释，应该很容易能够找到对应的控件。

接下来，我们需要将其翻译成为另一个语言版本的文件。如果是简体中文，则MainStoryboard.strings中的内容如下：

```
/* Class = "IBUIViewController"; title = "First"; ObjectID = "2"; */  
"2.title" = "第一";  
  
/* Class = "IBUIViewController"; title = "Second"; ObjectID = "3"; */  
"3.title" = "第二";  
  
/* Class = "IBUITabBarItem"; title = "Second"; ObjectID = "6"; */  
"6.title" = "第二";  
  
/* Class = "IBUITabBarItem"; title = "First"; ObjectID = "7"; */  
"7.title" = "First";  
  
/* Class = "IBUILabel"; text = "First View"; ObjectID = "20"; */  
"20.text" = "第一视图";  
  
/* Class = "IBUILabel"; text = "Second View"; ObjectID = "22"; */  
"22.text" = "第二视图";
```

然后再使用ibtool生成其他语言版本的nib或故事板文件，相关代码如下：

```
ibtool --strings-file MainStoryboard.strings --write zh-Hans.lproj/  
MainStoryboard.storyboard en.lproj/MainStoryboard.storyboard
```

其中参数--strings-file表示生成文件，MainStoryboard.strings表示翻译之后的字符串资源文件，

--write表示写入到zh-Hans.lproj/MainStoryboard.storyboard故事板文件中，en.lproj/MainStoryboard.storyboard是原始故事板文件。也就是按照en.lproj/MainStoryboard.storyboard文件作为模板，使用MainStoryboard.strings资源文件重新生成一个简体中文版的故事板文件zh-Hans.lproj/MainStoryboard.storyboard。

这样我们打开zh-Hans.lproj/MainStoryboard.storyboard故事板文件时，会发现静态文本都已经变成了简体中文，如图9-20所示。

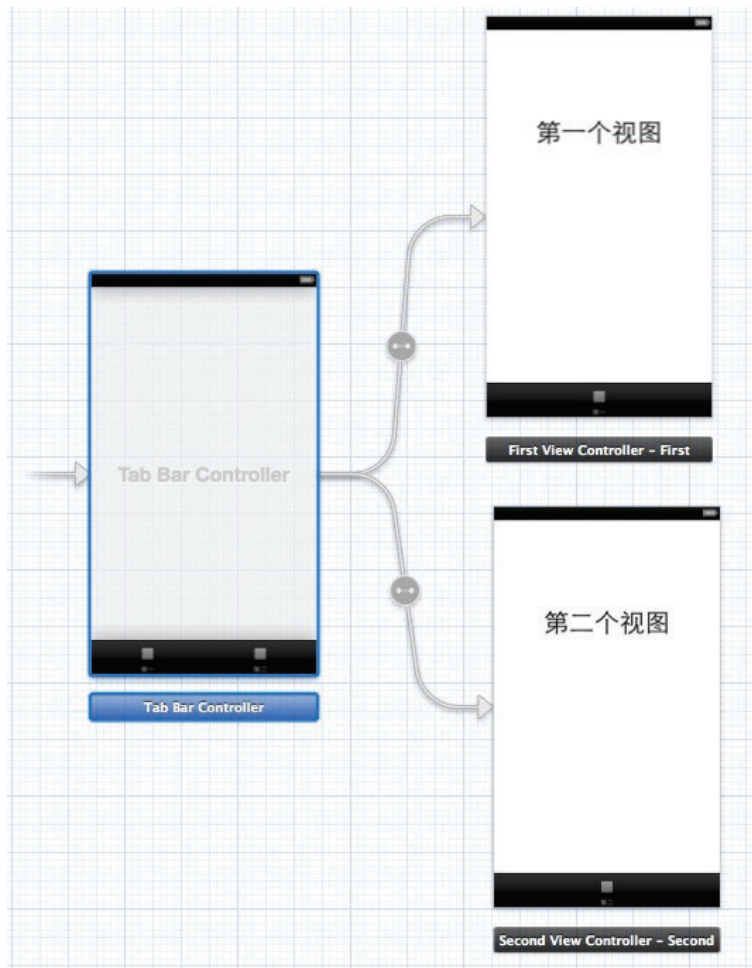


图9-20 zh-Hans.lproj/MainStoryboard.storyboard故事板文件


9.4 资源文件本地化

资源文件包括图片文件、音频文件以及前面提到的Localizable.strings等文件，它们的特点是随着应用一起打包发布。无论是图片文件还是音频文件，都必须实现本地化。它们的实现步骤都是类似的，因此我们重点介绍图片文件的本地化。

资源文件的本地化，也是需要准备好几个本地化版本的文件。例如，我们要实现一个游戏的控制界面，其中有控制关闭或者打开背景音乐的按钮和音效按钮，如图9-21所示。它们都是图片按钮，上面的文字是图片上的文字，因此需要本地化图片。



图9-21 本地化图片的案例

首先，将英文版图片添加到工程中，具体操作为：在故事板文件MainStoryboard.storyboard中正常添加两个UIImageView，接着选择对应的图片。然后为故事板添加本地化支持，这个过程请参考上一节内容。下面对图片进行本地化，选择背景按钮图片music_background.png文件，打开其文件检查器，点击Localization中的Make localized按钮（如图9-22所示），此时从弹出的界面中选择English，这样music_background.png文件就被移动到en.lproj目录下面了。

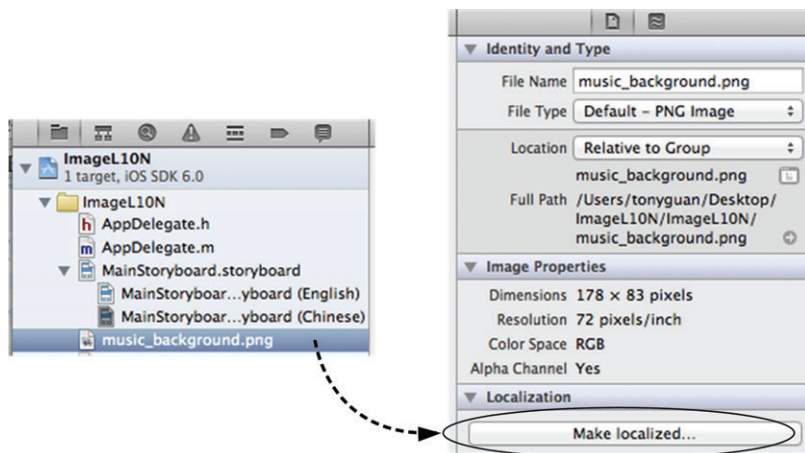


图9-22 文件检查器

添加简体中文版时，还需要按照上述操作选择Chinese选项才可以，如图9-23所示，这样music_background.png文件就复制到zh-Hans.lproj目录下面了。

此时就可以运行一下看看效果了。有时候，在Interface Builder打开的故事板设计界面中，往往会出现中文版本和英文版本混乱的情况，这是由于故事板加载的问题，只需重新打开工程即可。

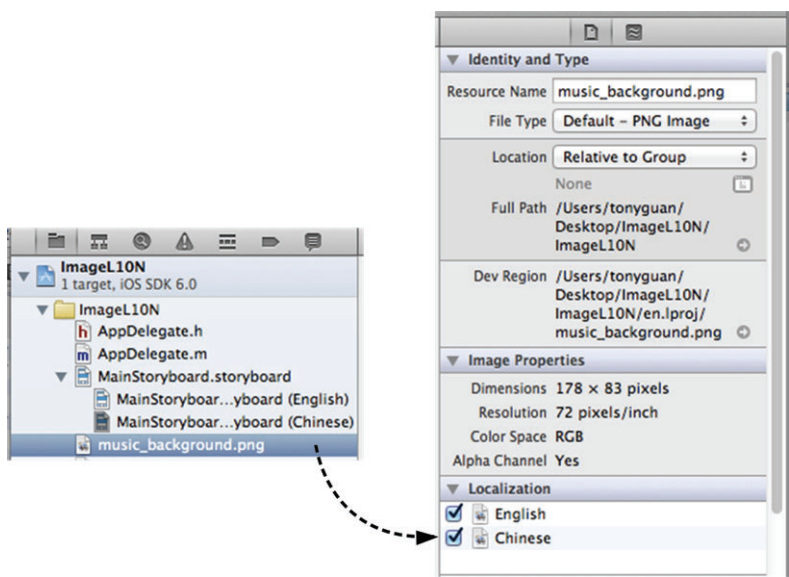


图9-23 选择Chinese选项

9.5 小结

在本章中，我们介绍了本地化的概念、内容和目录结构，并详细阐述了文本信息、nib及故事板、资源文件的本地化。从技术角度看，应用程序的本地化比较简单，但是工作量大而且比较烦琐。

信息和数据在现代社会中扮演着至关重要的角色，已成为我们生活中不可或缺的一部分。我们经常接触的信息有电话号码本、QQ通信录、消费记录等，而智能手机就是这些信息和数据的载体和传播工具。在本章中，我们将向大家介绍iOS系统中数据的多种持久化方式。

10.1 概述

iOS有一套完整的数据安全体系，iOS应用程序只能访问自己的目录，这个目录称为沙箱目录，而应用程序间禁止数据的共享和访问。访问一些特殊的应用，如：联系人应用，必须通过特定API访问。iOS支持现在主流数据持久化方式，本节将讨论这些持久化方式。

10.1.1 沙箱目录

沙箱目录是一种数据安全策略，很多系统都采用沙箱设计，实现HTML5规范的一些浏览器也采用沙箱设计。沙箱目录设计的原理就是只能允许自己的应用访问目录，而不允许其他的应用访问。在Android平台中，我们通过Content Provider技术将数据共享给其他应用。而在iOS系统中，特有的应用（联系人等）需要特定的API才可以共享数据，而其他的应用之间都不能共享数据。

下面的目录是iOS平台的沙箱目录，我们可以在模拟器下面看到，在真实设备上也是这样存储的：

```
/Users/<用户>/Library/Application Support/iPhone Simulator/6.0/  
Applications/A262B02A-1975-4A7A-AB8C-C181E2CC059A
```

其中A262B02A-1975-4A7A-AB8C-C181E2CC059A是应用程序ID，在安装时由系统分配。Documents、Library和tmp都是沙箱目录的子目录，其目录结构如下所示。

```
├── Documents  
│   └── NotesList.sqlite3  
├── Library  
│   ├── Caches  
│   └── Preferences  
├── tmp  
└── PresentationLayer.app
```

下面我们分别介绍这3个子目录，它们有不同的用途、场景和访问方式。

1. Documents目录

该目录用于存储非常大的文件或需要非常频繁更新的数据，能够进行iTunes或iCloud的备份。获取目录位置的代码如下所示：

```
NSArray * documentDirectory = NSSearchPathForDirectoriesInDomains  
(NSDocumentDirectory, NSUserDomainMask, YES);
```

其中documentDirectory是只有一个元素的数组，因此还需要使用下面的代码取出一个路径来：

```
NSString * myDocPath = [documentDirectory objectAtIndex:0];
```


或

```
NSString * myDocPath = [documentDirectory lastObject];
```

因为documentDirectory数组只有一个元素，所以取第一个元素和最后一个元素都是一样的，都可以取出Documents目录。

2. Library目录

在Library目录下面有Preferences和Caches目录，其中前者用于存放应用程序的设置数据，后者与Documents很相似，可以存放应用程序的数据，用来存储缓存文件。

3. tmp目录

这是临时文件目录，用户可以访问它。它不能够进行iTunes或iCloud的备份。要获取目录的位置，可以使用如下代码：

```
NSString *tmpDirectory = NSTemporaryDirectory();
```

10.1.2 持久化方式

持久化方式就是数据存取方式。iOS支持本地存储和云端存储，本章主要介绍本地存储，主要涉及如下4种机制。

- ❑ 属性列表。集合对象可以读写到属性列表文件中。
- ❑ 对象归档。对象状态可以保存到归档文件中。
- ❑ SQLite数据库。SQLite是一个开源嵌入式关系型数据库。
- ❑ Core Data。它是一种对象关系映射技术（ORM），本质上也是通过SQLite存储的。

采用什么技术，要看具体实际情况而定。属性列表文件和对象归档一般用于存储少量数据。属性列表文件的访问要比对象归档的访问简单，Foundation框架集合对象都有对应的方法读写属性列表文件，而对象归档是借助NSData实现的，使用起来比较麻烦。SQLite数据库和Core Data一般用于有几个简单表关系的大量数据的情况。如果是复杂表关系而且数据量很大，应该考虑把数据放在远程云服务器中。

10.2 属性列表

属性列表文件是一种XML文件，Foundation框架中的数组和字典等都可以与属性列表文件互相转换，如图10-1所示。

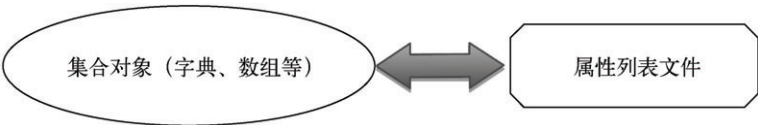


图10-1 集合与属性列表文件的对应关系

图10-2是属性列表文件NotesList.plist，它是一个数组，其中有两个元素，其元素结构是字典类型。图10-3是对应的NSArray，它是与NotesList.plist属性列表文件对应的集合对象。

Key	Type	Value
▼ Root	Array	(2 items)
▼ Item 0	Dictionary	(2 items)
date	String	2010-08-04 16:01:03
content	String	Welcome to MyNotes.
▼ Item 1	Dictionary	(2 items)
date	String	2011-12-04 16:01:03
content	String	欢迎使用MyNotes。

图10-2 属性列表文件NotesList.plist



图10-3 与NotesList.plist对应的NSArray集合对象

数组类NSArray和字典类NSDictionary提供了读写属性列表文件的方法,其中NSArray类的方法如下所示。

- ❑ **+ arrayWithContentsOfFile**。它是类级构造方法,用于从属性列表文件中读取数据,创建NSArray对象。
- ❑ **- initWithContentsOfFile**。它是实例构造方法,用于从属性列表文件读取数据,创建NSArray对象。
- ❑ **- writeToFile:atomically**。该方法把NSArray对象写入到属性列表文件中,它的第一个参数是文件名,第二个参数为是否使用辅助文件,如果为YES,则先写入到一个辅助文件,然后辅助文件再重新命名为目标文件,如果为NO,则直接写入到目标文件。

NSDictionary类的方法如下所示。

- ❑ **+ dictionaryWithContentsOfFile**。它是类级构造方法,用于从属性列表文件读取数据,创建NSDictionary对象。
- ❑ **- initWithContentsOfFile**。它是实例构造方法,用于从属性列表文件读取数据,创建NSDictionary对象。
- ❑ **- writeToFile:atomically**。将NSDictionary对象写入到属性列表文件中,它的第一个参数是文件名,第二个参数为是否使用辅助文件,如果为YES,则先写入到一个辅助文件,然后将辅助文件重新命名为目标文件,如果为NO,则直接写入到目标文件。

下面我们通过一个案例来学习属性列表文件的读写过程。还记得我们在第7章介绍的移动平台分层架构设计吗?其中一部分内容是基于一个工作空间不同工程的分层架构,在该种设计架构下创建了3个工程,如图10-4所示。

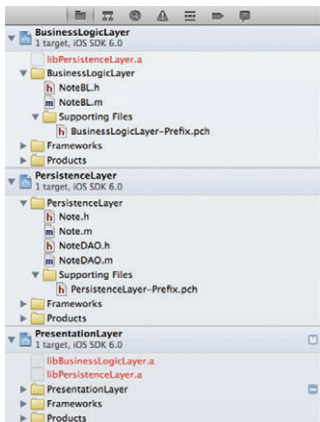


图10-4 基于一个工作空间不同工程的分层架构

当时采用的是数据持久层，但并没有真正将数据持久化，而是将数据临时保存在一个成员变量中，本章将采用不同的持久化技术完善这个案例。在该案例中，涉及一个实体类Note，其成员如图10-5所示。

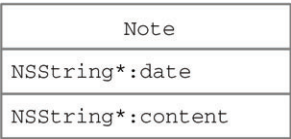


图10-5 案例中的实体类Note

Note类有两个成员变量——date和content，它们分别用于保存备忘录中的日期和内容。由于基于灵活的分层架构，我们只需要修改持久层工程（PersistenceLayer）中的NoteDAO类就可以了。其他的两个工程不需要做任何代码上的修改。

下面我们先看看NoteDAO.h文件的修改：

```
#import <Foundation/Foundation.h>
#import "Note.h"

@interface NoteDAO : NSObject

+ (NoteDAO*)sharedManager;

- (NSString *)applicationDocumentsDirectoryFile;

- (void)createEditableCopyOfDatabaseIfNeeded;

//插入备忘录的方法
- (int) create:(Note*)model;

//删除备忘录的方法
- (int) remove:(Note*)model;

//修改备忘录的方法
- (int) modify:(Note*)model;

//查询所有数据方法
- (NSMutableArray*) findAll;

//按照主键查询数据方法
- (Note*) findById:(Note*)model;

@end
```

在上述代码中，applicationDocumentsDirectoryFile方法获得放置在沙箱Documents目录下面的文件的完整路径。createEditableCopyOfDatabaseIfNeeded方法对数据文件进行预处理。在NoteDAO.m文件中，这两个方法的实现代码如下：

```
- (void)createEditableCopyOfDatabaseIfNeeded {

    NSFileManager *fileManager = [NSFileManager defaultManager];
    NSString *writableDBPath = [self applicationDocumentsDirectoryFile];

    BOOL dbexists = [fileManager fileExistsAtPath:writableDBPath];
    if (!dbexists) {
        NSString *defaultDBPath = [[[NSBundle mainBundle] resourcePath]
            stringByAppendingPathComponent:@"NotesList.plist"];

        NSError *error;
        BOOL success = [fileManager copyItemAtPath:defaultDBPath
            toPath:writableDBPath error:&error];
    }
}
```

```

        if (!success) {
            NSLog(@"错误写入文件: '%@'.", [error localizedDescription]);
        }
    }
}

- (NSString *)applicationDocumentsDirectoryFile {
    NSString *documentDirectory = [NSSearchPathForDirectoriesInDomain
                                   (NSDocumentDirectory, NSUserDomainMask, YES) lastObject];
    NSString *path = [documentDirectory
                      stringByAppendingPathComponent:@"NotesList.plist"];
    return path;
}

```

createEditableCopyOfDatabaseIfNeeded方法用于判断在Documents目录下是否存在NotesList.plist文件, 如果不存在则从资源目录下复制一个。在NotesList.plist这个文件中, 预先有两条数据, 如图10-2所示。NSFileManager的copyItemAtPath:toPath:error:方法实现文件复制。NSAssert1是Foundation框架提供的宏, 它在断言失败的情况下抛出异常, 类似的还有NSAssert和NSAssert2等。applicationDocumentsDirectoryFile方法中的stringByAppendingPathComponent:能够在目录后面追加文件名, 返回完整的文件路径。

在NoteDAO.m中, 我们也是采用单例设计模式。修改它的sharedManager方法, 具体如下:

```

+ (NoteDAO*)sharedManager
{
    static dispatch_once_t once;
    dispatch_once(&once, ^{

        sharedManager = [[self alloc] init];
        [sharedManager createEditableCopyOfDatabaseIfNeeded];
    });
    return sharedManager;
}

```

NoteDAO.m中插入方法的代码如下所示:

```

- (int) create:(Note*)model
{
    NSString *path = [self applicationDocumentsDirectoryFile];
    NSMutableArray *array = [[NSMutableArray alloc] initWithContentsOfFile:path];

    NSDateFormatter *dateFormat = [[NSDateFormatter alloc] init];
    [dateFormat setDateFormat: @"yyyy-MM-dd HH:mm:ss"];

    NSDictionary* dict = [NSDictionary
                          dictionaryWithObjects:@[[dateFormat stringFromDate:
                                                         model.date], model.content]
                          forKeys:@[@"date", @"content"]];

    [array addObject:dict];

    [array writeToFile:path atomically:YES];

    return 0;
}

```

通过NSMutableArray的initWithContentsOfFile方法可以将属性列表文件内容读取到array变量中。由于NSDate对象要转换成为yyyy-MM-dd HH:mm:ss格式的字符串, 所以需要NSDateFormatter设定格式, 然后使用stringFromDate:方法将其转换成为字符串, 最后用[array writeToFile:path atomically:YES]语句将其重新写入到属性列表文件中。

NoteDAO.m中删除备忘录方法的代码如下:

```

- (int) remove:(Note*)model
{

```

```

NSString *path = [self applicationDocumentsDirectoryFile];
NSMutableArray *array = [[NSMutableArray alloc] initWithContentsOfFile:path];

for (NSDictionary* dict in array) {

    NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
    [dateFormatter setDateFormat:@"%yyyy-MM-dd HH:mm:ss"];

    //Note* note = [[Note alloc] init];
    NSDate *date = [dateFormatter dateFromString:[dict objectForKey:@"date"]];

    //比较日期主键是否相等
    if ([date isEqualToDate:model.date]){
        [array removeObject: dict];
        [array writeToFile:path atomically:YES];
        break;
    }
}

return 0;
}

```

在上述代码中，需要注意if ([date isEqualToDate:model.date]){...}语句，它可以判断两个日期是否相等。由于备忘录的日期字段是主键，所以只有在日期相等的情况下，再从array集合中移除这个对象，最后再重新写回到属性列表文件中。

NoteDAO.m中修改备忘录方法的代码如下所示：

```

-(int) modify:(Note*)model
{
    NSString *path = [self applicationDocumentsDirectoryFile];
    NSMutableArray *array = [[NSMutableArray alloc] initWithContentsOfFile:path];

    for (NSDictionary* dict in array) {

        NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
        [dateFormatter setDateFormat:@"%yyyy-MM-dd HH:mm:ss"];

        NSDate *date = [dateFormatter dateFromString:[dict objectForKey:@"date"]];
        NSString* content = [dict objectForKey:@"content"];

        //比较日期主键是否相等
        if ([date isEqualToDate:model.date]){
            [dict setValue:content forKey:@"content"];
            [array writeToFile:path atomically:YES];
            break;
        }
    }
    return 0;
}

```

NoteDAO.m中查询所有数据方法的代码如下所示：

```

-(NSMutableArray*) findAll
{
    NSString *path = [self applicationDocumentsDirectoryFile];

    //[self.listData removeAllObjects];
    NSMutableArray *listData = [[NSMutableArray alloc] init];

    NSMutableArray *array = [[NSMutableArray alloc] initWithContentsOfFile:path];

    for (NSDictionary* dict in array) {

```

```

NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
[dateFormatter setDateFormat:@"%yyyy-MM-dd HH:mm:ss"];

Note* note = [[Note alloc] init];
note.date = [dateFormatter dateFromString:[dict objectForKey:@"date"]];
note.content = [dict objectForKey:@"content"];

[listData addObject:note];

}
return listData;
}

```

NoteDAO.m中按照主键查询数据方法的代码如下所示：

```

- (Note*) findById: (Note*) model
{
    NSString *path = [self applicationDocumentsDirectoryFile];
    NSMutableArray *array = [[NSMutableArray alloc] initWithContentsOfFile:path];

    for (NSDictionary* dict in array) {

        NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
        [dateFormatter setDateFormat:@"%yyyy-MM-dd HH:mm:ss"];

        Note* note = [[Note alloc] init];
        note.date = [dateFormatter dateFromString:[dict objectForKey:@"date"]];
        note.content = [dict objectForKey:@"content"];

        //比较日期主键是否相等
        if ([note.date isEqualToDate:model.date]){
            return note;
        }
    }
    return nil;
}

```

修改完成后，需要选择Product→Clean菜单项清除一些再编译，运行一下看看效果。

10.3 对象归档

对象归档是一种序列化方式。为了便于数据传输，先将归档对象序列化为一个文件，然后再通过反归档将数据恢复到对象中。归档技术可以实现数据的持久化，不过在大量数据和频繁读写的情况下，它就不太合适了。

对一个对象进行完整归档需要满足的条件为：该对象的类必须实现NSCoding协议，而且每个成员变量应该是基本数据类型或都是实现NSCoding协议的某个类的实例。

归档类NSKeyedArchiver和反归档类NSKeyedUnarchiver总与NSData关联在一起。NSData封装了字节数据的缓存类，提供了读写数据文件的方法，具体如下所示。

- ❑ **+ dataWithContentsOfFile**。它是类级构造方法，用于从文件读取数据创建NSData对象。
- ❑ **+ dataWithContentsOfFile:options:error**。它是类级构造方法，用于从文件读取数据创建NSData对象。
- ❑ **- initWithContentsOfFile**。它是实例构造方法，用于从文件读取数据创建NSData对象。
- ❑ **- initWithContentsOfFile:options:error**。它是实例构造方法，用于从文件读取数据创建NSData对象。
- ❑ **- writeToFile:atomically**。通过提供是否使用辅助文件，将NSData对象写入到文件中。
- ❑ **- writeToFile:options:error**。通过提供写入选项，将NSData对象写入到文件中。

归档过程是使用NSKeyedArchiver对象归档数据，具体过程为：首先将归档数据写入NSData对象，最后再

将NSData对象写入归档文件中。反归档过程是从归档文件中读取数据到NSData对象，再利用NSKeyedUnarchiver对象从NSData对象中反归档出数据。

下面我们使用归档技术实现备忘录案例。与属性列表文件的实现一样，我们只需要修改持久层工程(PersistenceLayer)中的NoteDAO类就可以了，其他的两个工程根本不需要修改任何代码。将NoteDAO.h文件修改如下：

```
#import <Foundation/Foundation.h>
#import "Note.h"

#define FILE_NAME @"NotesList.archive"
#define ARCHIVE_KEY @"NotesList"

@interface NoteDAO : NSObject

+ (NoteDAO*)sharedManager;

- (NSString *)applicationDocumentsDirectoryFile;
- (void)createEditableCopyOfDatabaseIfNeeded;

//插入备忘录的方法
- (int) create:(Note*)model;

//删除备忘录的方法
- (int) remove:(Note*)model;

//修改备忘录的方法
- (int) modify:(Note*)model;

//查询所有数据方法
- (NSMutableArray*) findAll;

//按照主键查询数据方法
- (Note*) findById:(Note*)model;

@end
```

因为程序中多次使用了归档文件名,我们定义了宏FILE_NAME,同理也定义了宏ARCHIVE_KEY归档数据的键。归档文件中有很多归档数据,为了在反归档时便于查找,在归档时我们为数据提供了键。

NoteDAO.m中createEditableCopyOfDatabaseIfNeeded方法的代码如下所示：

```
- (void)createEditableCopyOfDatabaseIfNeeded {

    NSFileManager *fileManager = [NSFileManager defaultManager];
    NSString *writableDBPath = [self applicationDocumentsDirectoryFile];

    BOOL dbexists = [fileManager fileExistsAtPath:writableDBPath];
    if (!dbexists) {

        NSString *path = [self applicationDocumentsDirectoryFile];

        NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
        [dateFormatter setDateFormat:@"%yyyy-MM-dd HH:mm:ss"];

        NSDate *date1 = [dateFormatter dateFromString:@"2010-08-04 16:01:03"];
        Note* note1 = [[Note alloc] init];
        note1.date = date1;
        note1.content = @"Welcome to MyNotes.";

        NSDate *date2 = [dateFormatter dateFromString:@"2011-12-04 16:01:03"];
        Note* note2 = [[Note alloc] init];
        note2.date = date2;
        note2.content = @"欢迎使用MyNotes。";
```

```

NSMutableArray* array = [[NSMutableArray alloc] init];

[array addObject:note1];
[array addObject:note2];

NSMutableData * theData = [NSMutableData data];
NSKeyedArchiver * archiver = [[NSKeyedArchiver alloc]
                               initWithWritingWithMutableData:theData];
[archiver encodeObject:array forKey:ARCHIVE_KEY];
[archiver finishEncoding];

[theData writeToFile:path atomically:YES];
}
}

```

该方法主要的作用与上一节中一样，就是判断在沙箱目录下是否存在归档文件NotesList.archive（如果不存在则创建一个），并且硬编码了两条初始数据。归档数据是放置在NSArray对象中的，将NSArray对象进行归档，NSArray对象内部每一个元素都是Note对象，而Note对象也必须能够被归档的，所以Note类实现了NSCoding协议，它的代码如下：

```

#import <Foundation/Foundation.h>

@interface Note : NSObject < NSCoding>

@property(nonatomic, strong) NSDate* date;
@property(nonatomic, strong) NSString* content;

@end

@implementation Note

-(void)encodeWithCoder:(NSCoder *)aCoder {
    [aCoder encodeObject:_date forKey:@"date"];
    [aCoder encodeObject:_content forKey:@"content"];
}

-(id)initWithCoder:(NSCoder *)aDecoder {
    self.date = [aDecoder decodeObjectForKey:@"date"];
    self.content = [aDecoder decodeObjectForKey:@"content"];
    return self;
}

@end

```

在NSCoding协议中，我们定义了两个方法：encodeWithCoder:编码方法和initWithCoder:反编码方法。NoteDAO.m中插入方法的代码如下：

```

-(int) create:(Note*)model
{
    NSString *path = [self applicationDocumentsDirectoryFile];
    NSMutableArray *array = [self findAll];

    [array addObject:model];

    NSMutableData * theData = [NSMutableData data];
    NSKeyedArchiver * archiver = [[NSKeyedArchiver alloc]
                                    initWithWritingWithMutableData:theData];
    [archiver encodeObject:array forKey:ARCHIVE_KEY];
    [archiver finishEncoding];
    [theData writeToFile:path atomically:YES];

    return 0;
}

```

NoteDAO.m中删除方法的代码如下：

```

-(int) remove:(Note*)model
{
    NSString *path = [self applicationDocumentsDirectoryFile];
    NSMutableArray *array = [self findAll];

    for (Note* note in array) {

        //比较日期主键是否相等
        if ([note.date isEqualToDate:model.date]){
            [array removeObject: note];

            NSMutableData * theData = [NSMutableData data];
            NSKeyedArchiver * archiver = [[NSKeyedArchiver alloc]
                                           initWithWritingWithMutableData:theData];
            [archiver encodeObject:array forKey:ARCHIVE_KEY];
            [archiver finishEncoding];
            [theData writeToFile:path atomically:YES];

            break;
        }
    }

    return 0;
}

```

NoteDAO.m中修改方法的代码如下:

```

-(int) modify:(Note*)model
{
    NSString *path = [self applicationDocumentsDirectoryFile];
    NSMutableArray *array = [self findAll];

    for (Note* note in array) {

        //比较日期主键是否相等
        if ([note.date isEqualToDate:model.date]){

            note.content = model.content;

            NSMutableData * theData = [NSMutableData data];
            NSKeyedArchiver * archiver = [[NSKeyedArchiver alloc]
                                           initWithWritingWithMutableData:theData];
            [archiver encodeObject:array forKey:ARCHIVE_KEY];
            [archiver finishEncoding];

            [theData writeToFile:path atomically:YES];

            break;
        }
    }

    return 0;
}

```

NoteDAO.m中查询所有数据方法的代码如下:

```

-(NSMutableArray*) findAll
{
    NSString *path = [self applicationDocumentsDirectoryFile];

    NSMutableArray *listData = [[NSMutableArray alloc] init];
    NSData * theData = [NSData dataWithContentsOfFile:path];

    if([theData length] > 0) {
        NSKeyedUnarchiver * archiver = [[NSKeyedUnarchiver alloc]
                                         initWithReadingWithData:theData];
        listData = [archiver decodeObjectForKey:ARCHIVE_KEY];
        [archiver finishDecoding];
    }
}

```

```

    }
    return listData;
}

```

NoteDAO.m中按照主键查询数据方法的代码如下：

```

- (Note*) findById: (Note*) model
{
    NSString *path = [self applicationDocumentsDirectoryFile];

    NSMutableArray *listData = [[NSMutableArray alloc] init];
    NSData * theData = [NSData dataWithContentsOfFile:path];

    if ([theData length] > 0) {
        NSKeyedUnarchiver * archiver = [[NSKeyedUnarchiver alloc]
                                         initWithReadingWithData:theData];
        listData = [archiver decodeObjectForKey:ARCHIVE_KEY];
        [archiver finishDecoding];

        for (Note* note in listData) {

            //比较日期主键是否相等
            if ([note.date isEqualToDate:model.date]){
                return note;
            }
        }
    }
    return nil;
}

```

其中归档的核心方法如下：

```

NSMutableData * theData = [NSMutableData data];
NSKeyedArchiver * archiver = [[NSKeyedArchiver alloc]
                               initWithWritingWithMutableData:theData];
[archiver encodeObject:array forKey:ARCHIVE_KEY];
[archiver finishEncoding];

[theData writeToFile:path atomically:YES];

```

在上述代码中，我们首先实例化NSMutableData（NSData的子类）对象theData，然后把theData作为参数，再实例化NSKeyedArchiver对象archiver。archiver对象的encodeObject:方法可以通过键对实现NSCoding协议的对象进行归档，[archiver finishEncoding]发出归档完成消息。最后，theData发送writeToFile:消息将NSData对象写入到文件中，这就实现了数据的归档，这个过程要比属性列表文件的实现复杂一些。

其中反归档的核心方法如下：

```

NSData * theData = [NSData dataWithContentsOfFile:path];
if ([theData length] > 0) {
    NSKeyedUnarchiver * archiver = [[NSKeyedUnarchiver alloc]
                                     initWithReadingWithData:theData];
    listData = [archiver decodeObjectForKey:ARCHIVE_KEY];
    [archiver finishDecoding];
}

```

在上述代码中，我们首先从以前的归档文件中创建NSData对象theData。紧接着，通过if ([theData length] > 0) {...}语句判断归档对象是否有数据。NSKeyedUnarchiver构造方法initWithReadingWithData:能够从对象theData中创建NSKeyedUnarchiver实例archiver。archiver的decodeObjectForKey:通过键反归档数据，它的返回值就是我们要的数据了。最后，使用[archiver finishDecoding]语句结束反归档。

10.4 使用 SQLite 数据库

2000年, D.理查德·希普开发并发布了嵌入式系统使用的关系数据库SQLite, 目前的主流版本是SQLite 3。SQLite是开源的, 它采用C语言编写, 具有可移植性强、可靠性高、小而容易使用的特点。SQLite运行时与使用它的应用程序之间共用相同的进程空间, 而不是单独的两个进程。

SQLite提供了对SQL-92标准的支持, 支持多表、索引、事务、视图和触发。SQLite是无数据类型的数据库, 就是字段不用指定类型。下面的代码在SQLite中是合法的:

```
CREATE TABLE mytable
(  a VARCHAR(10),
   b NVARCHAR(15),
   c TEXT,
   d INTEGER,
   e FLOAT,
   f BOOLEAN,
   g CLOB,
   h BLOB,
   i TIMESTAMP,
   j NUMERIC(10,5)
   k VARYING CHARACTER (24),
   l NATIONAL VARYING CHARACTER(16)
);
```

10.4.1 SQLite数据类型

虽然SQLite可以忽略数据类型, 但从编程规范上讲, 应该在Create Table语句中指定数据类型。因为数据类型可以告知这个字段的含义, 便于代码的阅读和理解。SQLite支持的常见数据类型如下所示。

- ❑ **INTEGER**。有符号的整数类型。
- ❑ **REAL**。浮点类型。
- ❑ **TEXT**。字符串类型, 采用UTF-8和UTF-16字符编码。
- ❑ **BLOB**。二进制大对象类型, 能够存放任何二进制数据。

在SQLite中没有Boolean类型, 可以采用整数0和1替代。在SQLite中, 也没有日期和时间类型, 它们存储在TEXT、REAL和INTEGER类型中。

为了兼容SQL-92中的其他数据类型, 可以将它们转换成为上述几种数据类型。

- ❑ VARCHAR、CHAR和CLOB转换成为TEXT类型。
- ❑ FLOAT、DOUBLE转换成为REAL类型。
- ❑ NUMERIC转换成为INTEGER或者REAL类型。

10.4.2 创建数据库

要创建数据库, 需要经过如下3个步骤。

- (1) 使用sqlite3_open函数打开数据库。
- (2) 使用sqlite3_exec函数执行Create Table语句, 创建数据库表。
- (3) 使用sqlite3_close函数释放资源。

在这个过程中, 我们使用了3个SQLite3函数, 它们都是纯C语言函数。通过Objective-C调用C函数当然不是什么问题, 但是也要注意Objective-C数据类型与C数据类型的兼容性问题。

下面我们使用SQLite技术实现备忘录案例。与属性列表文件的实现一样, 我们只需修改持久层工程(PersistenceLayer)中的NoteDAO类就可以了。首先, 我们需要添加SQLite3库到可以运行的工程环境中, 即表示

层工程 PresentationLayer。选择 PresentationLayer 工程中的 TARGETS→PresentationLayer→Link Binary With Libraries，点击左下角的+按钮，从弹出界面中选择libsqlite3.dylib或libsqlite3.0.dylib，在弹出的对话框中点击Add按钮添加，如图10-6所示。

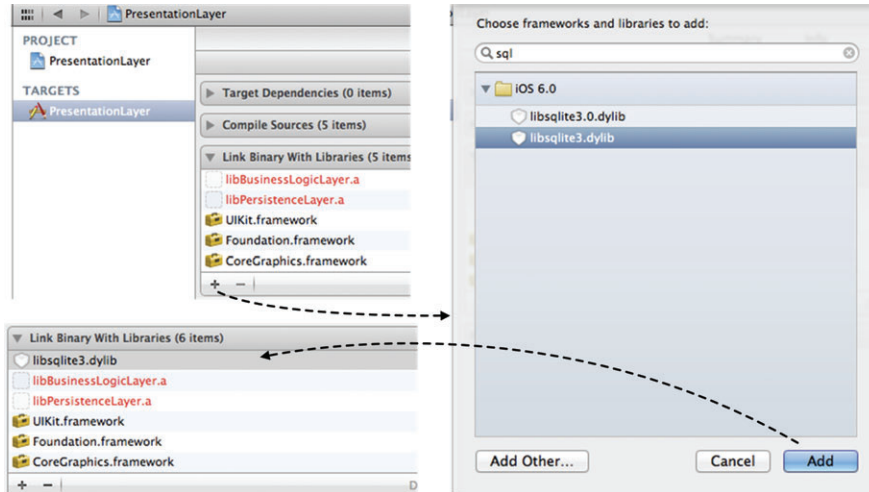


图10-6 添加SQLite3库到工程环境

将NoteDAO.h文件中的代码修改为如下的形式：

```
#import "Note.h"
#import "sqlite3.h"

#define DBFILE_NAME @"NotesList.sqlite3"

@interface NoteDAO : NSObject
{
    sqlite3 *db;
}

+ (NoteDAO*) sharedManager;

- (NSString *) applicationDocumentsDirectoryFile;
- (void) createEditableCopyOfDatabaseIfNeeded;

// 插入备忘录的方法
- (int) create: (Note*) model;

// 删除备忘录的方法
- (int) remove: (Note*) model;

// 修改备忘录的方法
- (int) modify: (Note*) model;

// 查询所有数据的方法
- (NSMutableArray*) findAll;

// 按照主键查询数据的方法
- (Note*) findById: (Note*) model;

@end
```

这里我们使用语句#import "sqlite3.h"引入了sqlite3头文件，而且定义了sqlite3*成员变量db。NoteDAO.m中createEditableCopyOfDatabaseIfNeeded方法的代码如下所示：


```

- (void)createEditableCopyOfDatabaseIfNeeded {

    NSString *writableDBPath = [self applicationDocumentsDirectoryFile];

    if (sqlite3_open([writableDBPath UTF8String], &db) != SQLITE_OK) {           ①
        sqlite3_close(db);                                                       ②
        NSAssert(NO, @"数据库打开失败。");
    } else {
        char *err;
        NSString *createSQL = [NSString stringWithFormat:@"CREATE TABLE IF NOT
            EXISTS Note (cdate TEXT PRIMARY KEY, content TEXT);"];           ③
        if (sqlite3_exec(db, [createSQL UTF8String], NULL, NULL, &err) !=
            SQLITE_OK) {                                                         ④
            sqlite3_close(db);                                                   ⑤
            NSAssert1(NO, @"建表失败, %s", err);                                ⑥
        }
        sqlite3_close(db);                                                       ⑦
    }
}

```

该方法用于创建数据库，第一步为打开数据库（见第①行的代码），其中sqlite3_open函数的第一个参数是数据库文件的完整路径，需要注意的是在SQLite3函数中接受的是char*的UTF-8类型数据，需要将NSString*类型的数据转换为UTF-8类型，这可以使用NSString*的UTF8String方法实现；第二个参数为sqlite3指针变量db的地址；返回值是int类型。在SQLite3中，我们定义了很多常量，如果返回值等于常量SQLITE_OK，则说明创建成功。

第二步为执行建表语句（见第④行的代码），其中语句sqlite3_exec(db, [createSQL UTF8String], NULL, NULL, &err)执行建表的SQL，该函数的第一个参数是sqlite3指针变量db的地址，第二个参数是要执行的SQL语句，第三个参数是要回调的函数，第四个参数是要回调函数的参数，第五个参数是执行出错的字符串。建表的SQL语句是CREATE TABLE IF NOT EXISTS Note (cdate TEXT PRIMARY KEY, content TEXT)，如果表Note存在，则不用创建。

第三步为使用sqlite3_close函数释放资源，见第②行、第⑤行和第⑦行的代码所示。该函数在数据库打开失败、Create Table执行失败和成功执行完成时调用。原则上，无论正常结束还是异常结束，必须使用sqlite3_close函数释放资源。

10.4.3 查询数据

数据查询一般会带有查询条件，这可以使用SQL语句的where子句实现，但是在程序中需要动态绑定参数给where子句。查询数据的具体操作步骤如下所示。

- (1) 使用sqlite3_open函数打开数据库。
- (2) 使用sqlite3_prepare_v2函数预处理SQL语句。
- (3) 使用sqlite3_bind_text函数绑定参数。
- (4) 使用sqlite3_step函数执行SQL语句，遍历结果集。
- (5) 使用sqlite3_column_text等函数提取字段数据。
- (6) 使用sqlite3_finalize和sqlite3_close函数释放资源。

NoteDAO.m中按照主键查询数据方法的代码如下：

```

- (Note*) findById: (Note*) model
{

    NSString *path = [self applicationDocumentsDirectoryFile];

    if (sqlite3_open([path UTF8String], &db) != SQLITE_OK) {           ①
        sqlite3_close(db);                                               ②
    }
}

```

```

        NSAssert(NO,@"数据库打开失败。");
    } else {

        NSString *qsql = @"SELECT cdate,content FROM Note where cdate =?";

        sqlite3_stmt *statement;
        //预处理过程
        if (sqlite3_prepare_v2(db, [qsql UTF8String], -1, &statement,NULL) == SQLITE_OK){ ③
            //准备参数
            NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init]; ④
            [dateFormatter setDateFormat:@"yyyy-MM-dd HH:mm:ss"];
            NSString *nsdate = [dateFormatter stringFromDate:model.date];
            //绑定参数开始
            sqlite3_bind_text(statement, 1, [nsdate UTF8String], -1, NULL); ⑤

            //执行
            if (sqlite3_step(statement) == SQLITE_ROW) { ⑥
                char *cdate = (char *) sqlite3_column_text(statement, 0); ⑦
                NSString *nscdate = [[NSString alloc]
                    initWithUTF8String: cdate];

                char *content = (char *) sqlite3_column_text(statement, 1);
                NSString * nscontent = [[NSString alloc]
                    initWithUTF8String: content];

                Note* note = [[Note alloc] init];
                note.date = [dateFormatter dateFromString:nscdate];
                note.content = nscontent;

                sqlite3_finalize(statement);
                sqlite3_close(db);
                return note;
            }

            sqlite3_finalize(statement); ⑧
            sqlite3_close(db); ⑨
        }

        return nil;
    }
}

```

该方法执行了6个步骤，其中第(1)个步骤如代码第①行所示，它与创建数据库的第一个步骤一样，这里就不再介绍了。

第(2)个步骤如代码第③行所示，语句`sqlite3_prepare_v2(db, [qsql UTF8String], -1, &statement, NULL)`是预处理SQL语句。预处理的目的是将SQL编译成二进制代码，提高SQL语句的执行速度。`sqlite3_prepare_v2`函数的第三个参数代表全部SQL字符串的长度，第四个参数是`sqlite3_stmt`指针的地址，它是语句对象，通过语句对象可以执行SQL语句，第五个参数是SQL语句没有执行的部分语句。

第(3)个步骤如代码第⑤行所示，语句`sqlite3_bind_text(statement, 1, [nsdate UTF8String], -1, NULL)`是绑定SQL语句的参数，其中第一个参数是`statement`指针，第二个参数为序号（从1开始），第三个参数为字符串值，第四个参数为字符串长度，第五个参数为一个函数指针。如果SQL语句中带有问号，则这个问号（它是占位符）就是要绑定的参数，示例代码如下所示：

```
NSString *qsql = @"SELECT cdate,content FROM Note where cdate =?";
```

第(4)个步骤为使用`sqlite3_step(statement)`执行SQL语句，如代码第⑥行所示。如果`sqlite3_step`函数返回`int`类型（即等于`SQLITE_ROW`），则说明还要其他的行没有遍历。

第(5)个步骤为提取字段数据，如代码第⑦行所示，它使用`sqlite3_column_text(statement, 0)`函数读取字符串类型的字段。需要说明的是，`sqlite3_column_text`函数的第二个参数用于指定select字段的索引（从

0开始)。同样, char*类型需要转换成NSString*类型, 这可以通过initWithUTF8String:构造方法实现。读取字段函数的采用与字段类型有关系, SQLite3中类似的常用函数还有:

- ❑ sqlite3_column_blob()
- ❑ sqlite3_column_double()
- ❑ sqlite3_column_int()
- ❑ sqlite3_column_int64()
- ❑ sqlite3_column_text()
- ❑ sqlite3_column_text16()

关于其他API, 读者可以参考<http://www.sqlite.org/cintro.html>。

第(6)个步骤是释放资源, 创建数据库过程不同,除了使用sqlite3_close函数关闭数据库, 代码第⑧行所示, 还要使用sqlite3_finalize函数释放语句对象statement代码第⑨行所示。

NoteDAO.m中的查询所有数据方法的代码如下:

```
-(NSMutableArray*) findAll
{
    NSString *path = [self applicationDocumentsDirectoryFile];
    NSMutableArray *listData = [[NSMutableArray alloc] init];

    if (sqlite3_open([path UTF8String], &db) != SQLITE_OK) {
        sqlite3_close(db);
        NSAssert(NO, @"数据库打开失败。");
    } else {

        NSString *qsql = @"SELECT cdate,content FROM Note";

        sqlite3_stmt *statement;
        //预处理过程
        if (sqlite3_prepare_v2(db, [qsql UTF8String], -1, &statement,
            NULL) == SQLITE_OK) {

            NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
            [dateFormatter setDateFormat:@"yyyy-MM-dd HH:mm:ss"];

            //执行
            while (sqlite3_step(statement) == SQLITE_ROW) {
                char *cdate = (char *) sqlite3_column_text(statement, 0);
                NSString *nscdate = [[NSString alloc] initWithUTF8String: cdate];

                char *content = (char *) sqlite3_column_text(statement, 1);
                NSString *nscontent = [[NSString alloc]
                    initWithUTF8String: content];

                Note* note = [[Note alloc] init];
                note.date = [dateFormatter dateFromString:nscdate];
                note.content = nscontent;

                [listData addObject:note];
            }

            sqlite3_finalize(statement);
            sqlite3_close(db);
        }
        return listData;
    }
}
```

查询所有数据方法与按照主键查询数据方法类似, 区别在于本方法没有查询条件不需要绑定参数。遍历的时候使用while循环语句, 不是if语句:

```
while (sqlite3_step(statement) == SQLITE_ROW) {
    .....
}
```

10.4.4 修改数据

修改数据时，涉及的SQL语句有insert、update和delete语句，这3个SQL语句都可以带参数。修改数据的具体步骤如下所示。

- (1) 使用sqlite3_open函数打开数据库。
 - (2) 使用sqlite3_prepare_v2函数预处理SQL语句。
 - (3) 使用sqlite3_bind_text函数绑定参数。
 - (4) 使用sqlite3_step函数执行SQL语句。
 - (5) 使用sqlite3_finalize和sqlite3_close函数释放资源。
- 这与查询数据少了提取字段数据这个步骤，其他步骤是一样的。下面我们看看代码部分。

NoteDAO.m中插入备忘录的代码如下：

```
-(int) create:(Note*)model
{
    NSString *path = [self applicationDocumentsDirectoryFile];

    if (sqlite3_open([path UTF8String], &db) != SQLITE_OK) {
        sqlite3_close(db);
        NSAssert(NO, @"数据库打开失败。");
    } else {

        NSString *sqlStr = @"INSERT OR REPLACE INTO note (cdate,content) VALUES (?,?)";

        sqlite3_stmt *statement;
        //预处理过程
        if (sqlite3_prepare_v2(db, [sqlStr UTF8String], -1, &statement,
            NULL) == SQLITE_OK) {
            NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
            [dateFormatter setDateFormat:@"yyyy-MM-dd HH:mm:ss"];
            NSString *nsdate = [dateFormatter stringFromDate:model.date];

            //绑定参数开始
            sqlite3_bind_text(statement, 1, [nsdate UTF8String], -1, NULL);
            sqlite3_bind_text(statement, 2, [model.content UTF8String], -1, NULL);

            //执行插入
            if (sqlite3_step(statement) != SQLITE_DONE) {
                NSAssert(NO, @"插入数据失败。");
            }

            sqlite3_finalize(statement);
            sqlite3_close(db);
        }

        return 0;
    }
}
```

第⑤行代码中的sqlite3_step(statement)语句执行插入语句，常量SQLITE_DONE表示执行完成。

NoteDAO.m中删除备忘录的代码如下：

```
-(int) remove:(Note*)model
{
    NSString *path = [self applicationDocumentsDirectoryFile];
```

```

if (sqlite3_open([path UTF8String], &db) != SQLITE_OK) {
    sqlite3_close(db);
    NSAssert(NO,@"数据库打开失败。");
} else {

    NSString *sqlStr = @"DELETE from note where cdate =?";

    sqlite3_stmt *statement;
    //预处理过程
    if (sqlite3_prepare_v2(db, [sqlStr UTF8String], -1, &statement,
        NULL) == SQLITE_OK) {
        NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
        [dateFormatter setDateFormat:@"yyyy-MM-dd HH:mm:ss"];
        NSString *nsdate = [dateFormatter stringFromDate:model.date];

        //绑定参数开始
        sqlite3_bind_text(statement, 1, [nsdate UTF8String], -1, NULL);
        //执行插入
        if (sqlite3_step(statement) != SQLITE_DONE) {
            NSAssert(NO, @"删除数据失败。 ");
        }
    }

    sqlite3_finalize(statement);
    sqlite3_close(db);
}

return 0;
}

```

NoteDAO.m中修改备忘录的代码如下:

```

-(int) modify:(Note*)model
{
    NSString *path = [self applicationDocumentsDirectoryFile];

    if (sqlite3_open([path UTF8String], &db) != SQLITE_OK) {
        sqlite3_close(db);
        NSAssert(NO,@"数据库打开失败。");
    } else {

        NSString *sqlStr = @"UPDATE note set content=? where cdate =?";

        sqlite3_stmt *statement;
        //预处理过程
        if (sqlite3_prepare_v2(db, [sqlStr UTF8String], -1, &statement,
            NULL) == SQLITE_OK) {

            NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
            [dateFormatter setDateFormat:@"yyyy-MM-dd HH:mm:ss"];
            NSString *nsdate = [dateFormatter stringFromDate:model.date];

            //绑定参数开始
            sqlite3_bind_text(statement, 1, [model.content UTF8String], -1, NULL);
            sqlite3_bind_text(statement, 2, [nsdate UTF8String], -1, NULL);
            //执行插入
            if (sqlite3_step(statement) != SQLITE_DONE) {
                NSAssert(NO, @"修改数据失败。");
            }
        }

        sqlite3_finalize(statement);
    }
}

```

```
        sqlite3_close(db);
    }
    return 0;
}
```

10.5 Core Data

Core Data是苹果为Mac OS X和iOS系统应用开发提供的数据库持久化技术。它基于高级数据库持久化API，它的底层最终是SQLite数据库、二进制文件和内存数据保存，这样开发人员不用再关心数据的存储细节问题，不再使用SQL语句，不用面对SQLite的C语言函数。

10.5.1 ORM

Core Data是一种ORM（对象关系映射）技术。听说过Hibernate^①的人对ORM不会感到陌生，ORM是关系模型数据和对象模型类之间的一个纽带。

无论哪一种模型，都是为了描述和构建应用系统。在应用系统中，一个非常基础的概念是“实体”。“实体”是应用系统中的“人”、“事”和“物”，它们能够在构造关系模型和对象模型中以不同的形态存在。如图10-7所示，实体在关系模型中代表表的一条数据，该表描述了实体的结构有哪些属性和关系。实体在对象模型中代表类的一个对象，类描述了实体的结构，实体是类的对象。因此，表是与类对应的概念，记录是与对象对应的概念。

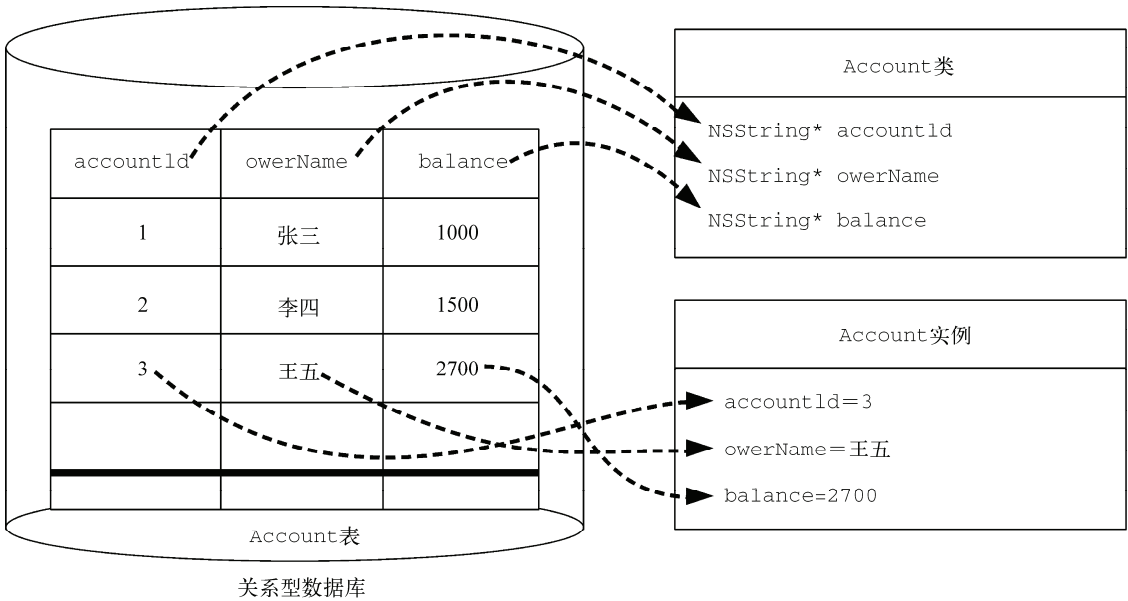


图10-7 ORM

关系模型和对象模型是有区别的，对象模型更加先进，能够描述继承、实现、关联、聚合和组成等复杂的关系，而关系模型只能描述一对一、一对多和多对多的关系。这两种模型之间的不和谐称为“阻抗不匹配”问题，而ORM可以解决“阻抗不匹配”问题。

① Hibernate是一个开放源代码的对象关系映射Java EE框架。

10.5.2 Core Data堆栈

使用Xcode工具，我们可以很方便地为工程添加Core Data支持。在Xcode的工程模板中，有3个模板（即Master-Detail Application、Utility Application和Empty Application模板）可以直接为工程添加Core Data支持，具体方法是创建工程时，选中Use Core Data复选框，如图10-8所示。

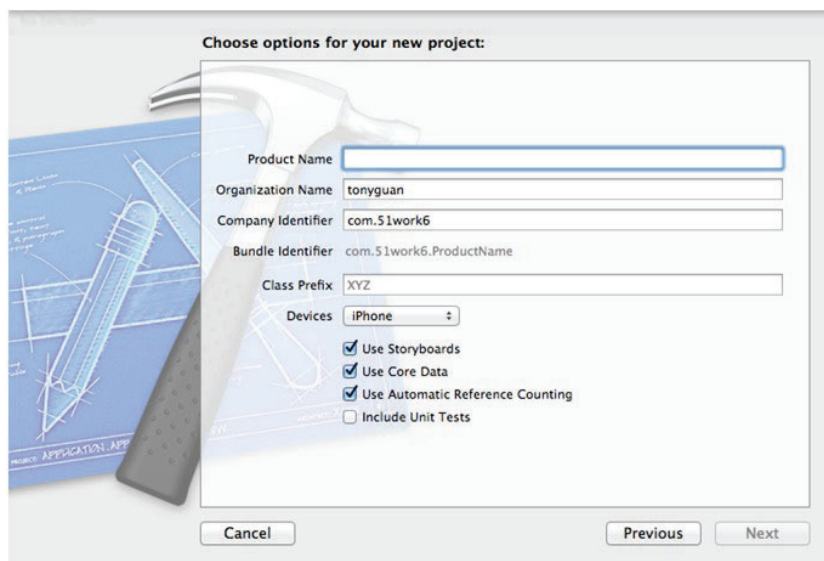


图10-8 添加Core Data支持

其他的模板需要自己添加Core Data支持。下面我们看看通过模板生成的代码，这些代码主要生成在AppDelegate中。AppDelegate.h中的代码如下：

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

@interface AppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;

@property (readonly, strong, nonatomic) NSManagedObjectContext
    *managedObjectContext;
@property (readonly, strong, nonatomic) NSManagedObjectModel
    *managedObjectModel;
@property (readonly, strong, nonatomic) NSPersistentStoreCoordinator
    *persistentStoreCoordinator;

- (void)saveContext;
- (NSURL *)applicationDocumentsDirectory;

@end
```

在AppDelegate.h中，我们定义了NSManagedObjectContext、NSManagedObjectModel、NSPersistentStoreCoordinator类型的属性，它们的含义如下所示。

- ❑ **NSManagedObjectContext**。它是被管理对象上下文（Managed Object Context，MOC）类，在上下文中可以查找、删除和插入对象，然后通过栈同步到持久化对象存储。
- ❑ **NSManagedObjectModel**。它是被管理对象模型（Managed Object Model，MOM）类，是系统中的“实体”，与数据库中的表等对象对应。

❑ **NSPersistentStoreCoordinator**。它是持久化存储协调器（Persistent Store Coordinator，PSC）类，在持久化对象存储之上提供了一个接口，可以把它考虑成为数据库的连接。

除了“被管理对象上下文”、“被管理对象模型”和“持久化存储协调器”外，还有“持久化对象存储”，它们一起构成了Core Data堆栈。

提示 持久化对象存储（Persistent Object Store，POS）执行所有底层的从对象到数据的转换，并负责打开和关闭数据文件。它有3种持久化实现方式：SQLite、二进制文件和内存形式。

Core Data堆栈如图10-9所示，有一个或多个被管理对象上下文，它连接到一个持久化存储协调器。一个持久化存储协调器连接到一个或多个持久化对象存储。持久化对象存储与底层存储文件关联。一个持久化存储协调器也可以管理多个被管理对象模型。一个持久化存储协调器就意味着一个Core Data堆栈。通过Core Data堆栈，可以实现数据查询、插入、删除和修改等操作。

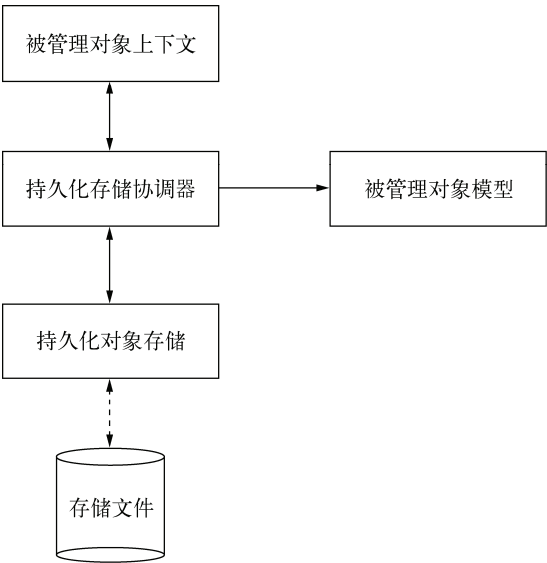


图10-9 Core Data堆栈

下面我们再看看AppDelegate.m中的saveContext方法，其代码如下所示：

```
- (void)saveContext
{
    NSError *error = nil;
    NSManagedObjectContext *managedObjectContext = self.managedObjectContext;
    if (managedObjectContext != nil) {
        if ([managedObjectContext hasChanges] && ![managedObjectContext
            save:&error]) {
            NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
            abort();
        }
    }
}
```

当插入、删除和修改数据之后，我们需要通过该方法保存被管理对象上下文，其中[managedObjectContext save:&error]语句是保存上下文的核心语句。

在AppDelegate.m中，applicationDocumentsDirectory方法的代码如下：

```

- (NSURL *)applicationDocumentsDirectory
{
    return [[[NSFileManager defaultManager]
        URLsForDirectory:NSDocumentDirectory inDomains:NSUserDomainMask] lastObject];
}

```

该方法返回应用程序沙箱Documents目录，它的返回类型是NSURL。它与我们之前返回应用程序沙箱Documents目录是类似的，只是原来返回的是字符串：

```

NSArray * documentDirectory = NSSearchPathForDirectoriesInDomains
    (NSDocumentDirectory, NSUserDomainMask, YES);
NSString * myDocPath = [documentDirectory lastObject];

```

在AppDelegate.m中，Core Data堆栈的方法如下：

```

//返回被管理对象上下文
- (NSManagedObjectContext *)managedObjectContext ①
{
    if (_managedObjectContext) {
        return _managedObjectContext;
    }

    NSPersistentStoreCoordinator *coordinator = ②
        [self persistentStoreCoordinator];
    if (coordinator != nil)
    {
        _managedObjectContext = [[NSManagedObjectContext alloc] init]; ③
        [_managedObjectContext setPersistentStoreCoordinator:coordinator]; ④
    }
    return _managedObjectContext;
}

//返回持久化存储协调器
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator ⑤
{
    if (_persistentStoreCoordinator )
    {
        return _persistentStoreCoordinator;
    }

    NSURL *storeURL = [[self applicationDocumentsDirectory]
        URLByAppendingPathComponent:@"<xcdatamodeld文件>.sqlite"];

    _persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
        initWithManagedObjectModel:[self managedObjectModel]]; ⑥
    [_persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType
        configuration:nil
        URL:storeURL
        options:nil
        error:nil]; ⑦
    return _persistentStoreCoordinator;
}

//返回被管理对象模型
- (NSManagedObjectModel *)managedObjectModel
{
    if (_managedObjectModel ) {
        return _managedObjectModel;
    }
    NSURL *modelURL = [[NSBundle mainBundle]
        URLForResource:@"<xcdatamodeld文件>" withExtension:@"momd"];
    _managedObjectModel = [[NSManagedObjectModel alloc]
        initWithContentsOfURL:modelURL];
    return _managedObjectModel;
}

```

上面的3个方法是与.h文件中定义的3个属性对应的getter方法。我们知道，属性事实上封装了getter方法和setter方法。它们的调用顺序如图10-10所示。

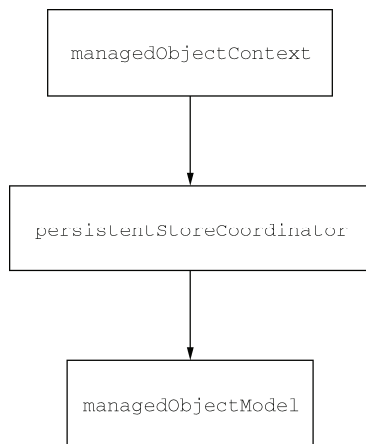


图10-10 Core Data堆栈方法的调用顺序

第①行代码`managedObjectContext`方法返回MOC对象，提供了访问`managedObjectContext`属性的getter方法。第②行代码是调用`persistentStoreCoordinator`方法获得PSC对象，第③行代码用于实例化MOC对象，第④行代码通过`[_managedObjectContext setPersistentStoreCoordinator:coordinator]`语句设置PSC数据持久化类型。

第⑤行代码中的`persistentStoreCoordinator`方法返回PSC对象，它提供了访问`persistentStoreCoordinator`属性的getter方法。第⑥行代码用于实例化PSC对象，第⑦行代码使用`addPersistentStoreWithType:configuration:URL:options:error:`方法为PSC对象添加新的持久化数据存储。`addPersistentStoreWithType`参数用于指定存储类型，它的取值可以是下面的3个常量。

- ❑ **NSSQLiteStoreType**。指数据持久化类型是SQLite数据。
- ❑ **NSBinaryStoreType**。指数据持久化类型是二进制文件。
- ❑ **NSInMemoryStoreType**。指数据持久化类型是内存形式。

10.5.3 建模和生成实体

在上一节的代码中，有`<xcdatamodeld文件>`，它是模型文件^①。Core Data可以利用它可视化设计数据库，生成实体类的Objective-C代码和SQLite数据库文件。下面我们介绍建模和生成实体这两个过程。

1. 建模

默认情况下，如果采用模板生成，会创建一个与工程名相同的数据模型文件——`<工程名>.xcdatamodeld`，但是如果不采用模板，则创建过程是选择File→New→File...菜单项，从打开的选择文件模板对话框中选择iOS→Core Data→Data Model，如图10-11所示。

然后点击Next按钮，输入相应的文件名，这里我们输入`CoreDataNotes`，这样创建的数据模型文件就是`CoreDataNotes.xcdatamodeld`。但是需要注意的是，在程序代码中加载的时候，`managedObjectModel`方法中会用到这个文件：

```

NSURL *modelURL = [[NSBundle mainBundle] URLForResource:@"<xcdatamodeld
文件>" withExtension:@"momd"];
_managedObjectModel = [[NSManagedObjectModel alloc]
initWithContentsOfURL:modelURL];
  
```

① 模型文件一般是在数据库设计阶段用可视化建模工具创建的数据库模型描述文件。

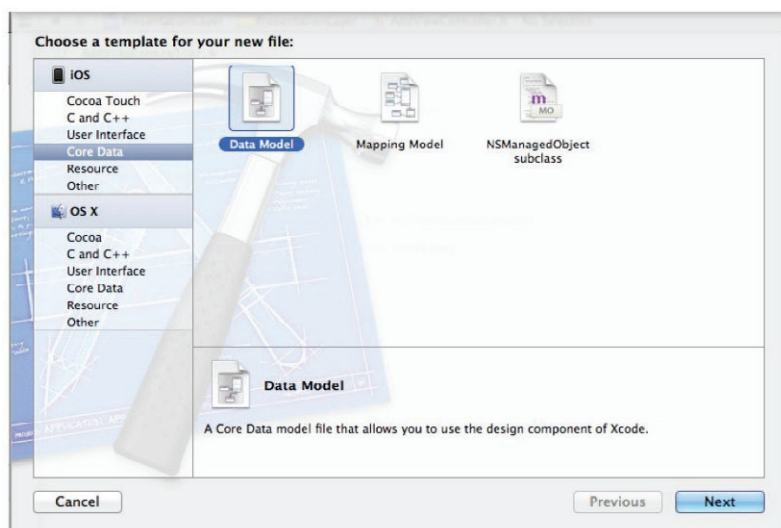


图10-11 创建数据模型文件

但是我们会发现程序的扩展名是momd而不是xcdatamodeld，这是因为CoreDataNotes.xcdatamodeld文件在编译发布时，变成了CoreDataNotes.momd，我们在编程时需要注意这个问题。

由于数据模型文件CoreDataNotes.xcdatamodeld属于资源文件，如果采用基于一个工作空间不同工程的分层架构，CoreDataNotes.xcdatamodeld文件应该创建并放置在表示层的工程中。

打开CoreDataNotes.xcdatamodeld文件，看到如图10-12所示的模型对话框，在这个对话框中可以创建实体(entity)、实体属性和实体关系等。

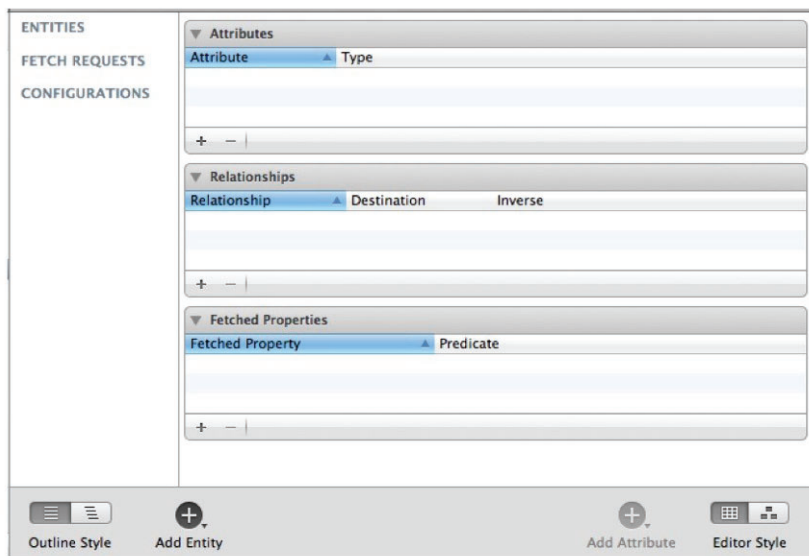


图10-12 模型对话框

创建实体的过程如图10-13所示：点击Add Entity按钮添加实体，将其名称修改为Note，在右边的Attributes列表框中添加属性date和content，并选择Type为Date和String。

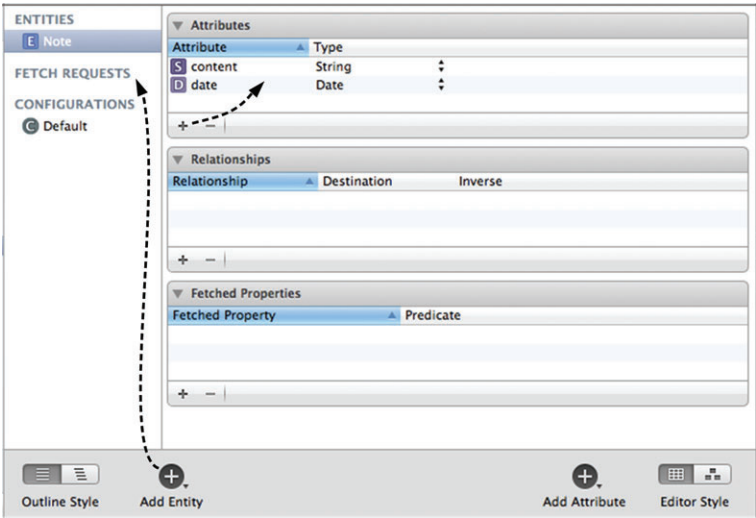


图10-13 创建实体

选择右下角的Editor Style按钮，改变后的建模样式对话框如图10-14所示。

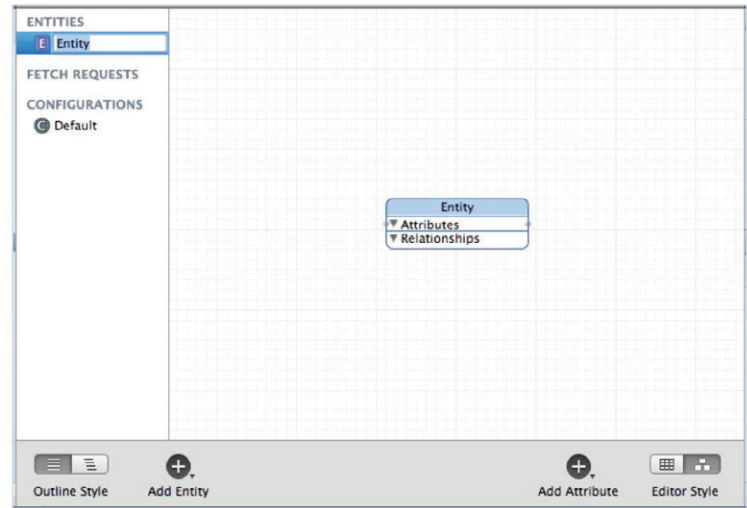



图10-14 建模样式对话框

2. 生成实体

选择实体Note，打开其数据模型检查器，如图10-15所示，在Entity的Class输入框中输入NoteManagedObject。

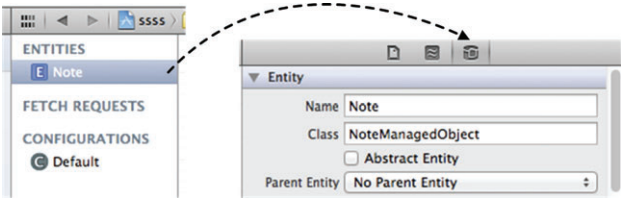


图10-15 输入实体类名

然后再选择File→New→File...菜单项,从打开的选择文件模板对话框中选择iOS→Core Data→NSManagedObject subclass,如图10-16所示,点击Next按钮生成实体类。

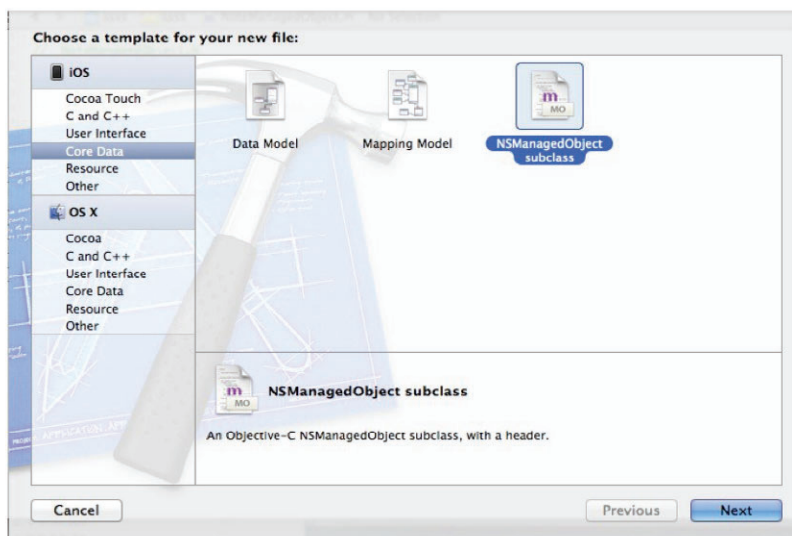


图10-16 生成实体类

生成之后的实体类的NoteManagedObject.h代码如下

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>
@interface NoteManagedObject : NSManagedObject

@property (nonatomic, retain) NSDate * date;
@property (nonatomic, retain) NSString * content;

@end
```

NoteManagedObject实体类需要被Core Data管理,因此需要继承NSManagedObject类。而之前的实体类直接继承NSObject类,它没有被Core Data管理,其代码如下:

```
@interface Note : NSObject

@property (nonatomic, strong) NSDate * date;
@property (nonatomic, strong) NSString * content;

@end
```

生成之后的实体类的NoteManagedObject.m代码如下,它们采用动态属性实现方式:

```
@implementation NoteManagedObject

@dynamic date;
@dynamic content;

@end
```

10.5.4 采用Core Data分层架构设计

堆栈创建都是在应用程序委托对象中实现的,而按照第7章iOS平台一般信息处理应用分层架构设计图所设计的Core Data应该只出现在数据持久层,Core Data中的对象NSManagedObjectContext、NSPersistentStoreCoordinator、

NSManagedObjectModel和NSManagedObjectContext等都不能出现在其他层中。采用分层设计必须遵守这个规范。

将重新将构建Core Data堆栈的代码移植到DAO类（CoreDataDAO）中，其中CoreDataDAO是所有DAO类的父类。CoreDataDAO.h的代码如下：

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>

@interface CoreDataDAO : NSObject

//被管理的对象上下文
@property (readonly, strong, nonatomic) NSManagedObjectContext *managedObjectContext;
//被管理的对象模型
@property (readonly, strong, nonatomic) NSManagedObjectModel *managedObjectModel;
//持久化存储协调器
@property (readonly, strong, nonatomic) NSPersistentStoreCoordinator
*persistentStoreCoordinator;

- (NSURL *)applicationDocumentsDirectory;

@end
```

CoreDataDAO.m的代码如下：

```
@implementation CoreDataDAO

@synthesize managedObjectContext = _managedObjectContext;
@synthesize managedObjectModel = _managedObjectModel;
@synthesize persistentStoreCoordinator = _persistentStoreCoordinator;

#pragma mark - Core Data 堆栈
//返回被管理对象上下文
- (NSManagedObjectContext *)managedObjectContext
{
    <参考10.5.2节>
    .....
}

//返回持久化存储协调器
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator
{
    <参考10.5.2节>
    .....
}

//返回被管理对象模型
- (NSManagedObjectModel *)managedObjectModel
{
    <参考10.5.2节>
    .....
}

#pragma mark - 应用程序沙箱
//返回应用程序Documents目录的NSURL类型
- (NSURL *)applicationDocumentsDirectory
{
    return [[[NSFileManager defaultManager] URLsForDirectory:NSDocumentDirectory
        inDomains:NSUserDomainMask] lastObject];
}

@end
```

然后让NoteDAO继承CoreDataDAO，并且增加了NoteManagedObject被管理实体类，这样数据持久层工程中的类如表10-1所述。

表10-1 数据持久层工程中的类

类 名	说 明
CoreDataDAO	DAO基类
NoteDAO	NoteDAO类
Note	未被管理的实体类
NoteManagedObject	被管理的实体类

Note和NoteManagedObject看起来有点重复，但是它们有不同的角色，这是一个非常重要的问题。如果不采用分层设计，我们完全可以采用NoteManagedObject，但是由于这里采用了分层设计，NoteManagedObject对象必须被严格限定在持久层中使用，而实体还会出现在表示层和业务逻辑层中，因此设计了Note类用在其他层中。在持久层中使用时，要在Note和NoteManagedObject之间转换。这个工作看起来比较麻烦，但是随着业务复杂度的增加，它的优点便会呈现出来。

10.5.5 查询数据

下面我们具体看看在分层架构下如何采用Core Data技术实现查询和修改(insert、update和delete)数据。先看看查询，查询分为无条件查询和有条件查询。

1. 无条件查询

NoteDAO.m中查询所有数据方法的代码如下：

```
-(NSMutableArray*) findAll
{
    NSManagedObjectContext *cxt = [self managedObjectContext];

    NSEntityDescription *entityDescription = [NSEntityDescription
        entityForName:@"Note" inManagedObjectContext:cxt]; ①

    NSFetchRequest *request = [[NSFetchRequest alloc] init]; ②
    [request setEntity:entityDescription]; ③

    NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
        initWithKey:@"date" ascending:YES]; ④
    [request setSortDescriptors:@[sortDescriptor]]; ⑤

    NSError *error = nil;
    NSArray *listData = [cxt executeFetchRequest:request error:&error]; ⑥

    NSMutableArray *resListData = [[NSMutableArray alloc] init];

    for (NoteManagedObject *mo in listData) { ⑦
        Note *note = [[Note alloc] init];
        note.date = mo.date;
        note.content = mo.content;
        [resListData addObject:note];
    }

    return resListData;
}
```

第①行代码中的NSEntityDescription是实体关联的描述类，通过指定实体的名字获得NSEntityDescription实例对象，实体的名字是在数据模型文件中定义的。第②行代码中的NSFetchRequest是数据提取请求类，用于查询。第③行代码把实体描述设定到请求对象中。第④行代码中的NSSortDescriptor是排序描述类，它可以指定排序字段以及排序方式。第⑤行代码把排序描述设定到请求对象中。

第⑥行中的[cxt executeFetchRequest:request error:&error]语句根据前面设置的请求对象执行查

询，返回NSArray集合。但是NSArray集合中放置的是被管理的NoteManagedObject实体对象，我们需要把它们转换到Note实体对象中，并把它们放置在NSMutableArray集合中。第⑦行循环体就实现了这个转换。

2. 有条件查询

NoteDAO.m中按照主键查询数据的代码如下：

```
-(Note*) findById:(Note*)model
{
    NSManagedObjectContext *cxt = [self managedObjectContext];

    NSEntityDescription *entityDescription = [NSEntityDescription
        entityForName:@"Note" inManagedObjectContext:cxt];

    NSFetchedRequest *request = [[NSFetchedRequest alloc] init];
    [request setEntity:entityDescription];

    NSPredicate *predicate = [NSPredicate predicateWithFormat:
        @"date = %@",model.date];
    [request setPredicate:predicate];

    NSError *error = nil;
    NSArray *listData = [cxt executeFetchRequest:request error:&error];

    if ([listData count] > 0) {
        NoteManagedObject *mo = [listData lastObject];

        Note *note = [[Note alloc] init];
        note.date = mo.date;
        note.content = mo.content;

        return note;
    }
    return nil;
}
```

与无条件查询不同的是，增加了如下语句：

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:
    @"date = %@",model.date];
[request setPredicate:predicate];
```

在上述代码中，NSPredicate用来定义一个逻辑查询条件，在内存中过滤集合对象。我们在5.2.3节介绍过它的用法，上面定义的查询条件是查询日期字段。[request setPredicate:predicate]语句用于把NSPredicate对象设定到提取请求对象中。

10.5.6 修改数据

这里的修改数据也是指insert、update和delete。NoteDAO.m中插入备忘录的方法如下：

```
-(int) create:(Note*)model
{
    NSManagedObjectContext *cxt = [self managedObjectContext];

    NoteManagedObject *note = [NSEntityDescription
        insertNewObjectForEntityForName:@"Note"
        inManagedObjectContext:cxt];
    [note setValue:model.content forKey:@"content"];
    [note setValue:model.date forKey:@"date"];

    note.date = model.date;
    note.content = model.content;
```

①
 ②
 ③

```

NSError *savingError = nil;
if ([self.managedObjectContext save:&savingError]){
    NSLog(@"插入数据成功");
} else {
    NSLog(@"插入数据失败");
    return -1;
}

return 0;
}

```

第①行代码中的`[NSEntityDescription insertNewObjectForEntityForName:@"Note" inManagedObjectContext:cxt]`语句用于创建一个被管理的Note实体对象，该对象不能通过init方法创建。第②行代码用于设定content属性值，也可以写成`note.content = model.content`，这是一种通过属性赋值的方法。第③行代码用于设定date属性值。第④行中的`[self.managedObjectContext save:&savingError]`语句保存修改，同步到持久化数据文件中。

NoteDAO.m中删除备忘录的方法如下：

```

-(int) remove:(Note*)model
{
    NSManagedObjectContext *cxt = [self managedObjectContext];

    NSEntityDescription *entityDescription = [NSEntityDescription
        entityForName:@"Note" inManagedObjectContext:cxt];

    NSFetchedRequest *request = [[NSFetchedRequest alloc] init];
    [request setEntity:entityDescription];

    NSPredicate *predicate = [NSPredicate predicateWithFormat:
        @"date = %@", model.date];
    [request setPredicate:predicate];

    NSError *error = nil;
    NSArray *listData = [cxt executeFetchRequest:request error:&error];
    if ([listData count] > 0) {
        NoteManagedObject *note = [listData lastObject];
        [self.managedObjectContext deleteObject:note];
        // ①

        NSError *savingError = nil;
        if ([self.managedObjectContext save:&savingError]){
            NSLog(@"删除数据成功");
        } else {
            NSLog(@"删除数据失败");
            return -1;
        }
        // ②
    }

    return 0;
}

```

进行删除操作时，首先要查询出要删除的实体，然后使用第①行语句删除实体，最后在第②行调用`[self.managedObjectContext save:&savingError]`语句保存修改，同步到持久化数据文件中。

NoteDAO.m中修改备忘录的方法如下：

```

-(int) modify:(Note*)model
{
    NSManagedObjectContext *cxt = [self managedObjectContext];

    NSEntityDescription *entityDescription = [NSEntityDescription
        entityForName:@"Note" inManagedObjectContext:cxt];

    NSFetchedRequest *request = [[NSFetchedRequest alloc] init];

```

```
[request setEntity:entityDescription];

NSPredicate *predicate = [NSPredicate predicateWithFormat:
    @"date = %@", model.date];
[request setPredicate:predicate];

NSError *error = nil;
NSArray *listData = [cxt executeFetchRequest:request error:&error];
if ([listData count] > 0) {
    NoteManagedObject *note = [listData lastObject];
    note.content = model.content;

    NSError *savingError = nil;
    if ([self.managedObjectContext save:&savingError]){
        NSLog(@"修改数据成功");
    } else {
        NSLog(@"修改数据失败");
        return -1;
    }
}
return 0;
}
```

要进行修改操作，首先需要查询出要修改的实体，然后修改实体中的属性，最后在第①行调用 `[self.managedObjectContext save:&savingError]` 语句保存修改，同步到持久化数据文件中。

10.6 小结

根据数据的规模和使用特点，可以将其放在本地或者云服务器中，而本章主要讨论了本地数据持久化。我们分析了数据存取几种方式以及每种数据存取方式适合什么样的场景，并分别举例介绍了每种存取方式的实现。

移动设备上都有一个很重要的内置数据库——通讯录，苹果把它扩展到了iCloud上，使苹果设备间可以共享通讯录信息。在iOS上，通讯录放在SQLite3数据库中，但是应用之间不能直接访问，也就是说我们自己编写的应用不能采用数据持久化技术直接访问通讯录数据库。为了实现通讯录数据库的访问，苹果开放了一些专门的API。

出于安全考虑，iOS 6之后的应用访问通讯录时，需要获得用户的授权，如图11-1所示。与其他应用（如定位服务授权）不同的是，通讯录对一个应用只授权一次，即便是这个应用删除后重新安装，也不必再次授权。

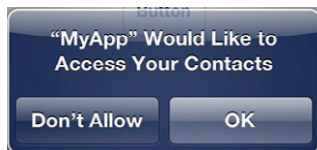


图11-1 用户授权对话框

此外，用户还可以在“设置”应用中禁止或允许某个应用访问通讯录，这也是在iOS 6中新追加的。在iOS 6之后，“设置”应用中有一个隐私设置项目，通过它可以设置通讯录的安全访问，如图11-2所示。



图11-2 设置通讯录的安全访问

11.1 概述

在开发访问通讯录的应用中，我们使用了两个框架：AddressBook和AddressBookUI。AddressBook框架主要提供了直接访问通讯录中记录和属性等API。由于使用这些API，需要自己构建UI界面，所以它们被称为“低级API”。

AddressBook框架中常用的类见表11-1。

表11-1 AddressBook框架中常用的类

类 名	说 明
ABAddressBook	封装访问通讯录接口。Core Foundation框架中对应的类型是ABAddressBookRef
ABPerson	封装通讯录个人信息数据，是数据库的一条记录。Core Foundation框架中对应的类型是ABPersonRef
ABGroup	封装通讯录组信息数据，一个组包含了多个人的信息，一个人也可以隶属多个组。Core Foundation框架中对应的类型是ABGroupRef
ABRecord	封装了数据库的一条记录，记录由属性组成。Core Foundation框架中对应的类型是ABRecordRef

提示 Core Foundation框架和Foundation框架紧密相关，它们为相同功能提供接口，但Foundation框架提供Objective-C接口，Core Foundation框架提供C接口。如果将Foundation对象和Core Foundation类型混合使用，则可利用两个框架之间的“无开销桥接”（toll-free bridging）。无开销桥接是说Core Foundation和Foundation框架中的某些类型可以互相转换，如表11-1中的ABAddressBook和ABAddressBookRef。

AddressBookUI框架提供了4个视图控制器和4个对应的委托协议，它们已经提供UI界面，不需要我们自己构建，因此称为“高级API”。这4个视图控制器和对应的委托协议如表11-2所述。

表11-2 AddressBookUI框架中的视图控制器

视图控制器	说 明
ABPeoplePickerNavigationController	它是从数据库中选取联系人的导航控制器，对应的委托协议为ABPeoplePickerNavigationControllerDelegate
ABPersonViewController	查看并编辑单个联系人信息，对应的委托协议为ABPersonViewControllerDelegate
ABNewPersonViewController	创建新联系人信息，对应的委托协议为ABNewPersonViewControllerDelegate
ABUnknownPersonViewController	呈现记录部分信息，这些信息可以创建新联系人信息，或添加到已经存在的联系人，对应的委托协议为ABUnknownPersonViewControllerDelegate

11.2 读取联系人信息

读取通讯录中联系人的一般过程是先查找联系人记录，然后再访问记录的属性，属性又可以分为单值属性和多值属性。本节通过图11-3所示的例子介绍联系人的查询、单值属性和多值属性的访问以及如何读取联系人中的图片数据。

本案例从iOS设备上读取通讯录中的联系人并将其显示在一个表视图中，可以在通讯录中查询联系人，点击联系人可以进入详细信息界面。这里我们必须做的两件事情如下所示。

- ❑ 添加AddressBook和AddressBookUI框架。为工程添加AddressBook.framework和AddressBookUI.framework，如图11-4所示。
- ❑ 引入头文件。在需要访问通讯录类的头文件中引入下面的头文件：

```
#import <AddressBook/AddressBook.h>
#import <AddressBookUI/AddressBookUI.h>
```

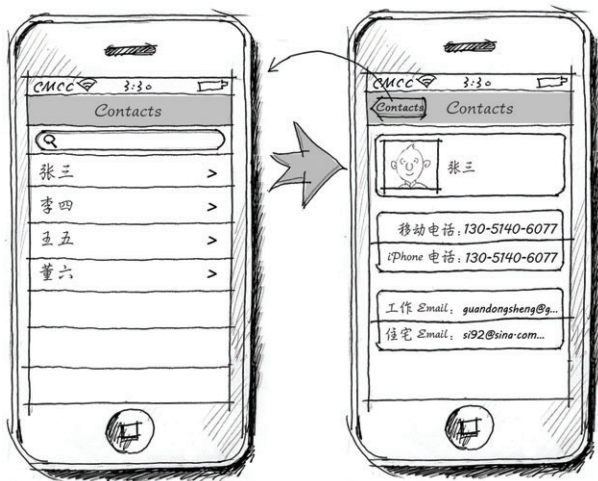


图11-3 访问通讯录中联系人信息的案例设计原型图

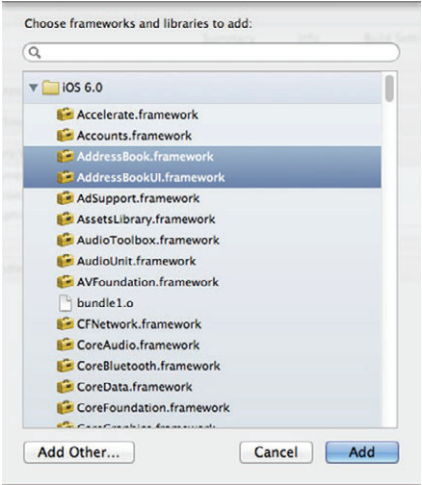


图11-4 添加框架

11.2.1 查询联系人记录

从通讯录数据库中查询联系人数据时，无法使用SQL语句，只能通过ABAddressBookCopyArrayOfAllPeople和ABAddressBookCopyPeopleWithName函数获得，它们的定义如下：

```
CFArrayRef ABAddressBookCopyArrayOfAllPeople (
    ABAddressBookRef addressBook
);

CFArrayRef ABAddressBookCopyPeopleWithName (
    ABAddressBookRef addressBook,
    CFStringRef name
);
```

前者用于查询所有的联系人数据，后者通过人名查询通讯录中的联系人，其中name参数就是查询的前缀关键字。两个函数中都有addressBook参数，它是我们要查询的通讯录对象，使用ABAddressBookCreateWithOptions函数（在iOS 6之前是ABAddressBookCreate函数）创建。该函数的定义如下：

```
ABAddressBookRef ABAddressBookCreateWithOptions (
    CFDictionaryRef options,
    CLErrorRef* error
);
```

其中options参数是保留参数，目前没有采用，使用时可以传递NULL值。error是错误对象，包含错误信息。

下面我们先看一下ViewController.h中的代码：

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

#import "DetailViewController.h"

@interface ViewController : UITableViewController
    <UISearchBarDelegate, UISearchDisplayDelegate>

@property (nonatomic, strong) NSArray *listContacts;

- (void)filterContentForSearchText:(NSString*) searchText;

@end
```

其中属性listContacts是装载联系人记录的数组集合, filterContentForSearchText:方法是用来过滤联系人信息的方法, 也就是查询方法。

ViewController.m中的viewDidLoad方法如下:

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

    CFErrorRef error = NULL;
    ABAddressBookRef addressBook = ABAddressBookCreateWithOptions(NULL, &error);           ①
    ABAddressBookRequestAccessWithCompletion(addressBook, ^(bool granted,
        CFErrorRef error) {
        if (granted) {
            //查询所有
            [self filterContentForSearchText:@""];
        }
    });
    CFRelease(addressBook);
}
```

在上述代码中, 我们首先在第①行代码处使用ABAddressBookCreateWithOptions函数创建addressBook对象, 然后在第②行又调用了函数ABAddressBookRequestAccessWithCompletion, 这个函数用于向用户请求访问通讯录数据库, 如果是第一次访问, 则会弹出一个用户授权对话框, 如果用户授权可以访问, 则会调用下面的代码块:

```
^(bool granted, CFErrorRef error) {
    if (granted) {
    }
};
```

请求和代码块的回调都是异步的, 表视图界面先出现, 然后才有查询出来的结果。在iOS 6之后必须有这个请求过程, 否则无法访问通讯录数据库。

ViewController.m中filterContentForSearchText:查询方法的代码如下:

```
- (void)filterContentForSearchText:(NSString*)searchText
{
    //如果没有授权, 则退出
    if (ABAddressBookGetAuthorizationStatus() != kABAuthorizationStatusAuthorized)
    {
        return ;
    }
    CFErrorRef error = NULL;
    ABAddressBookRef addressBook = ABAddressBookCreateWithOptions(NULL, &error);

    if([searchText length]==0)
    {
        //查询所有
        self.listContacts = CFBridgingRelease(ABAddressBookCopyArrayOfAllPeople
            (addressBook));
    } else {
        //条件查询
        CFStringRef cfSearchText = (CFStringRef)CFBridgingRetain(searchText);
        self.listContacts = CFBridgingRelease(ABAddressBookCopyPeopleWithName
            (addressBook, cfSearchText));
        CFRelease(cfSearchText);
    }
    [self.tableView reloadData];
    CFRelease(addressBook);
}
```

在该方法中, 我们使用ABAddressBookGetAuthorizationStatus()函数返回应用的授权状态, 其中kABAuthorizationStatusAuthorized常量代表用户已经授权。在没有授权的情况下, 该方法不进行任何处理。

ABAddressBookCopyArrayOfAllPeople函数用于查询所有数据,ABAddressBookCopyPeopleWithName函数根据条件查询,返回值是CFArrayRef类型,不能直接赋值给listContacts (NSArray*类型)属性,处理方式一般如下两种:

```
self.listContacts = (__bridge NSArray *)ABAddressBookCopyArrayOfAllPeople(addressBook);
```

或

```
self.listContacts = CFBridgingRelease(ABAddressBookCopyArrayOfAllPeople(addressBook));
```

使用第一种方式时,不会转让对象所有权,只是简单强制转化。而CFBridgingRelease函数实现的是Core Foundation类型到Foundation类型的转化并把对象所有权转让给ARC(自动引用计数),因此不需要释放属性listContacts对应的成员变量。类似的还有CFBridgingRetain函数,它实现的是Foundation类型到Core Foundation类型的转化,并把对象所有权转让给调用者,因此需要释放这个对象,相关代码如下:

```
CFStringRef cfSearchText = (CFStringRef)CFBridgingRetain(searchText);
self.listContacts = CFBridgingRelease(ABAddressBookCopyPeopleWithName
    (addressBook, cfSearchText));
CFRelease(cfSearchText);
```

第④行代码释放了addressBook对象。Core Foundation框架中的数据类型内存管理是不受ARC管理的,但是与Foundation框架的MRC管理类似,需要手动释放。CFRelease函数就相当于Foundation框架中的release(或autorelease)方法。

ViewController.m中的SearchBar查询相关方法如下:

```
#pragma mark --UISearchBarDelegate 协议方法
- (void)searchBarCancelButtonClicked:(UISearchBar *)searchBar
{
    // 查询所有
    [self filterContentForSearchText:@""];
}
#pragma mark - UISearchDisplayController Delegate Methods
// 当文本内容发生改变时,向表视图数据源发出重新加载消息
- (BOOL)searchDisplayController:(UISearchDisplayController *)controller
    shouldReloadTableForSearchString:(NSString *)searchString
{
    [self filterContentForSearchText:searchString];
    // 返回YES的情况下,表视图可以重新加载
    return YES;
}
```

11.2.2 读取单值属性

在一条联系人记录中,有很多属性,这些属性又有单值属性和多值属性之分。单值属性是只有一个值的属性,如姓氏和名字等,它们由下面的常量定义。

- ❑ **kABPersonFirstNameProperty**。名字。
- ❑ **kABPersonLastNameProperty**。姓氏。
- ❑ **kABPersonMiddleNameProperty**。中间名。
- ❑ **kABPersonPrefixProperty**。前缀。
- ❑ **kABPersonSuffixProperty**。后缀。
- ❑ **kABPersonNicknameProperty**。昵称。
- ❑ **kABPersonFirstNamePhoneticProperty**。名字汉语拼音或音标。
- ❑ **kABPersonLastNamePhoneticProperty**。姓氏汉语拼音或音标。
- ❑ **kABPersonMiddleNamePhoneticProperty**。中间名汉语拼音或音标。
- ❑ **kABPersonOrganizationProperty**。组织名。

❑ **kABPersonJobTitleProperty**。头衔。

❑ **kABPersonDepartmentProperty**。部门。

❑ **kABPersonNoteProperty**。备注。

读取记录属性的函数是ABRecordCopyValue，其定义如下：

```
CTypeRef ABRecordCopyValue (
    ABRecordRef record,
    ABPropertyID property
);
```

其中ABRecordRef参数是记录对象，ABPropertyID是属性ID，就是上面的常量kABPersonFirstNameProperty等。返回值类型是CTypeRef，它是Core Foundation类型的“泛型”，可以代表任何的Core Foundation类型。

```
ViewController.m中的tableView:cellForRowAtIndexPath:方法主要实现了访问单值属性：
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
    (NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    }

    ABRecordRef thisPerson = CFBridgingRetain([self.listContacts objectAtIndex:
        [indexPath row]]]; ①

    NSString *firstName = CFBridgingRelease(ABRecordCopyValue(thisPerson,
        kABPersonFirstNameProperty)); ②
    firstName = firstName != nil?firstName:@"";
    NSString *lastName = CFBridgingRelease(ABRecordCopyValue(thisPerson,
        kABPersonLastNameProperty));
    lastName = lastName != nil?lastName:@"";
    cell.textLabel.text = [NSString stringWithFormat:@"%@ %@", firstName, lastName];

    CFRelease(thisPerson); ③

    return cell;
}
```

第①行语句用于从NSArray*集合中取出一个元素，并将其转化为Core Foundation的ABRecordRef类型。第②行中的CFBridgingRelease(ABRecordCopyValue(thisPerson, kABPersonFirstNameProperty))语句将名字属性取出来，转化为NSString*类型。第③行代码用于释放ABRecordRef对象。

此外，为了把选中的联系人传递给详细界面，我们需要先获得选中记录的ID，然后把ID传递到详细界面。这个过程是在ViewController.m中的prepareForSegue:方法完成的，相关代码如下所示：

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    if ([segue.identifier isEqualToString:@"showDetail"]) {
        NSIndexPath *indexPath = [self.tableView indexPathForSelectedRow];

        ABRecordRef thisPerson = CFBridgingRetain([self.listContacts
            objectAtIndex:indexPath.row]);
        DetailViewController *detailViewController = [segue
            destinationViewController];

        ABRecordID personID = ABRecordGetRecordID(thisPerson); ①
        NSNumber *personIDAsNumber = [NSNumber numberWithInt:personID]; ②
        detailViewController.personIDAsNumber = personIDAsNumber; ③

        CFRelease(thisPerson); ④
    }
}
```


第①行代码调用函数ABRecordGetRecordID获取选中记录的ID，其中ID为ABRecordID类型。为了传递这个ID给DetailViewController视图控制器，我们为DetailViewController视图控制器定义了personIDAsNumber属性。第③行代码将ID赋给personIDAsNumber属性。DetailViewController.h的代码如下：

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

@interface DetailViewController : UITableViewController

@property (weak, nonatomic) IBOutlet UIImageView *imageView;

@property (weak, nonatomic) IBOutlet UILabel *lblName;
@property (weak, nonatomic) IBOutlet UILabel *lblMobile;

@property (weak, nonatomic) IBOutlet UILabel *lblIPhone;
@property (weak, nonatomic) IBOutlet UILabel *lblWorkEmail;
@property (weak, nonatomic) IBOutlet UILabel *lblHomeEmail;

@property (strong, nonatomic) NSNumber* personIDAsNumber;

@end
```

其中personIDAsNumber属性为NSNumber*类型。

11.2.3 读取多值属性

- 多值属性是包含多个值的集合类型，如电话号码、E-mail和URL等，它们主要由下面的常量定义。
- ❑ **kABPersonPhoneProperty**。电话号码属性，kABMultiStringPropertyType类型的多值属性。
 - ❑ **kABPersonEmailProperty**。E-mail属性，kABMultiStringPropertyType类型的多值属性。
 - ❑ **kABPersonURLProperty**。URL属性，kABMultiStringPropertyType类型的多值属性。
 - ❑ **kABPersonRelatedNamesProperty**。亲属关系人属性，kABMultiStringPropertyType类型的多值属性。
 - ❑ **kABPersonAddressProperty**。地址属性，kABMultiDictionaryPropertyType类型的多值属性。
 - ❑ **kABPersonInstantMessageProperty**。即时聊天属性，kABMultiDictionaryPropertyType类型的多值属性。
 - ❑ **kABPersonSocialProfileProperty**。社交账号属性，kABMultiDictionaryPropertyType类型的多值属性。

在多值属性中，包含了label（标签）、value（值）和ID等部分，其中标签和值都是可以重复的，而ID是不能重复的，如图11-5所示。

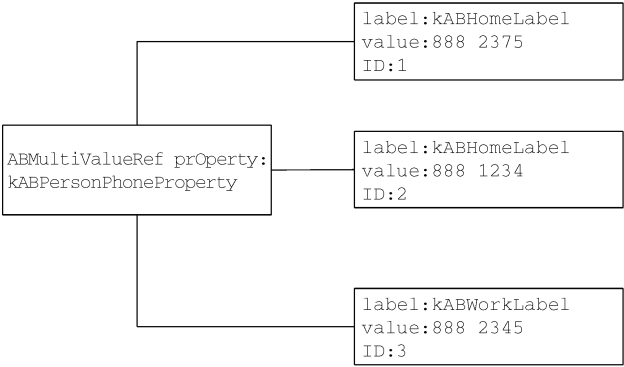


图11-5 多值属性中的标签、值和ID

多值属性访问方式与单值属性访问方式类似，都使用ABRecordCopyValue函数。不同的是，多值属性访问的返回值是ABMultiValueRef，然后使用ABMultiValueCopyArrayOfAllValues函数从ABMultiValueRef中获取CFArrayRef数组。ABMultiValueCopyArrayOfAllValues函数的定义如下：

```
CFArrayRef ABMultiValueCopyArrayOfAllValues (
    ABMultiValueRef multiValue
);
```

ABMultiValueCopyLabelAtIndex函数可以从ABMultiValueRef对象中返回标签，其定义如下：

```
CFStringRef ABMultiValueCopyLabelAtIndex (
    ABMultiValueRef multiValue,
    CFIndex index
);
```

其中参数multiValue是ABMultiValueRef对象，index是查找标签的索引。

ABMultiValueGetIdentifierAtIndex函数可以从ABMultiValueRef对象中返回ID，其定义如下：

```
ABMultiValueIdentifier ABMultiValueGetIdentifierAtIndex (
    ABMultiValueRef multiValue,
    CFIndex index
);
```

在DetailViewController.m文件的viewDidLoad方法中，取得E-mail多值属性，其代码如下：

```
ABRecordID personID = [self.personIDAsNumber intValue];
NSErrorRef error = NULL;
ABAddressBookRef addressBook = ABAddressBookCreateWithOptions(NULL, &error);
ABRecordRef person = ABAddressBookGetPersonWithRecordID(addressBook, personID);
.....
ABMultiValueRef emailsProperty = ABRecordCopyValue(person, kABPersonEmailProperty);
NSArray* emailsArray = CFBridgingRelease(ABMultiValueCopyArrayOfAllValues(emailsProperty));
for(int index = 0; index < [emailsArray count]; index++){
    NSString *email = [emailsArray objectAtIndex:index];
    NSString *emailLabel = CFBridgingRelease(ABMultiValueCopyLabelAtIndex (emailsProperty, index));

    if ([emailLabel isEqualToString:(NSString*)kABWorkLabel]) {
        [self.lblWorkEmail setText:email];
    } else if ([emailLabel isEqualToString:(NSString*)kABHomeLabel]) {
        [self.lblHomeEmail setText:email];
    } else {
        NSLog(@"%@: %@", @"其他Email", email);
    }
}
CFRelease(emailsProperty);
```

其中ABMultiValueCopyArrayOfAllValues(emailsProperty)语句用于从emailsProperty属性中取出数组集合。kABWorkLabel和kABHomeLabel都是E-mail多值属性的标签。kABWorkLabel是工作E-mail标签，kABHomeLabel是家庭E-mail标签。另外，还有kABOtherLabel，它也是E-mail标签。最后，需要释放emailsProperty。

在DetailViewController.m的viewDidLoad方法中，取得电话号码多值属性的代码如下：

```
ABMultiValueRef phoneNumberProperty = ABRecordCopyValue(person,
    kABPersonPhoneProperty);
NSArray* phoneNumberArray = CFBridgingRelease(ABMultiValueCopyArrayOfAllValues
    (phoneNumberProperty));
for(int index = 0; index < [phoneNumberArray count]; index++){
    NSString *phoneNumber = [phoneNumberArray objectAtIndex:index];
    NSString *phoneNumberLabel = CFBridgingRelease(ABMultiValueCopyLabelAtIndex
    (phoneNumberProperty, index));

    if ([phoneNumberLabel isEqualToString:(NSString*)kABPersonPhoneMobileLabel]) {
        [self.lblMobile setText:phoneNumber];
    }
}
```

```

    } else if ([phoneNumberLabel isEqualToString:(NSString*)kABPersonPhoneIPhoneLabel]) {
        [self.lblIPhone setText:phoneNumber];
    } else {
        NSLog(@"%@: %@", @"其他电话", phoneNumber);
    }
}
CFRelease(phoneNumberProperty);

```

在上述代码中，`kABPersonPhoneMobileLabel`和`kABPersonPhoneIPhoneLabel`都是电话号码属性的标签，其中前者是移动电话号码标签，后者是iPhone电话号码标签。此外，还有下面几个电话标签。

- ❑ `kABPersonPhoneMainLabel`。主要电话号码标签。
- ❑ `kABPersonPhoneHomeFAXLabel`。家庭传真电话号码标签。
- ❑ `kABPersonPhoneWorkFAXLabel`。工作传真电话号码标签。
- ❑ `kABPersonPhonePagerLabel`。寻呼机号码标签。

11.2.4 读取图片属性

通讯录中的联系人可以有一张照片。读取联系人照片的相关函数有`ABPersonCopyImageData`和`ABPersonHasImageData`等。`ABPersonCopyImageData`可以获取联系人照片，它的定义如下：

```

CFDataRef ABPersonCopyImageData (
    ABRecordRef person
);

```

它的返回类型是`CFDataRef`，与之对应的Foundation框架类型是`NSData*`。

`ABPersonHasImageData`函数用于判断联系人是否有照片，它的定义如下：

```

bool ABPersonHasImageData (
    ABRecordRef person
);

```

在`DetailViewController.m`的`viewDidLoad`方法中，取得联系人照片的代码如下：

```

if (ABPersonHasImageData(person)) {
    NSData *photoData = CFBridgingRelease(ABPersonCopyImageData(person));
    if(photoData){
        [self.imageView setImage:[UIImage imageDataWithBytes:photoData]];
    }
}

```

`ABPersonCopyImageData`取出的是`CFDataRef`类型，将其转化为`NSData*`，再使用`UIImage`的构造方法`imageWithData:`构建`UIImage`对象，然后再把`UIImage`对象赋值给`imageView`图片控件。

11.3 写入联系人信息

写入联系人信息到通讯录数据库时，涉及创建联系人、修改联系人和删除联系人等操作，下面简要介绍一下这些操作。

图11-6用于创建联系人信息。在首页点击导航栏右边的+按钮，弹出添加模态视图，添加完成之后，点击Save按钮保存并返回首页。如果点击Cancel按钮，则不保存。

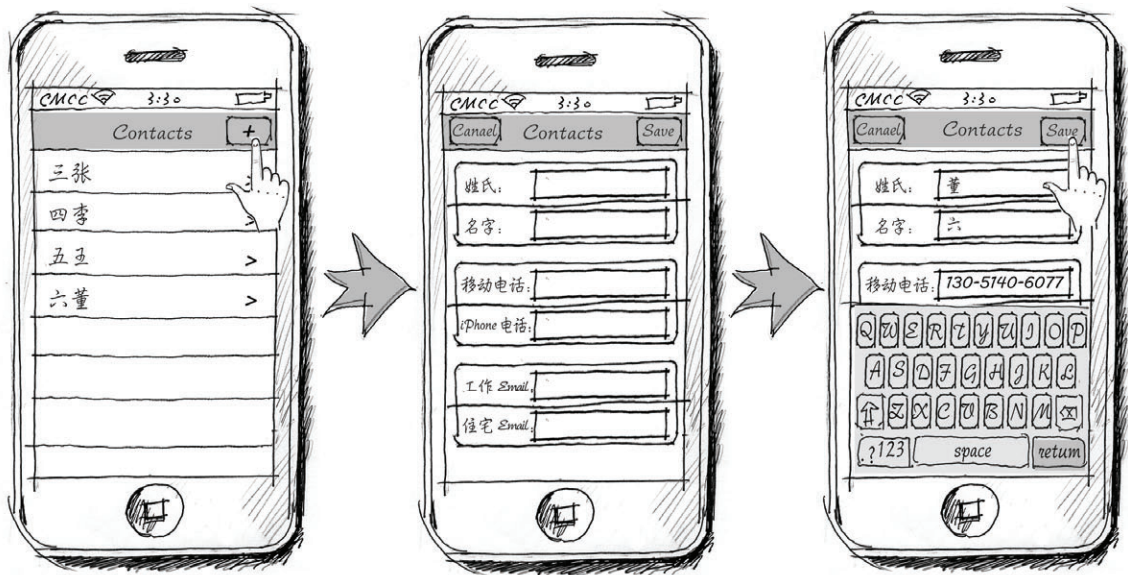


图11-6 创建联系人信息

图11-7用于修改联系人信息。在首页点击某个联系人单元格，导航到该联系人详细信息界面，在这个界面中我们可以修改电话号码和E-mail，但不能修改姓名。修改完成之后，点击Save按钮保存并导航回首页。

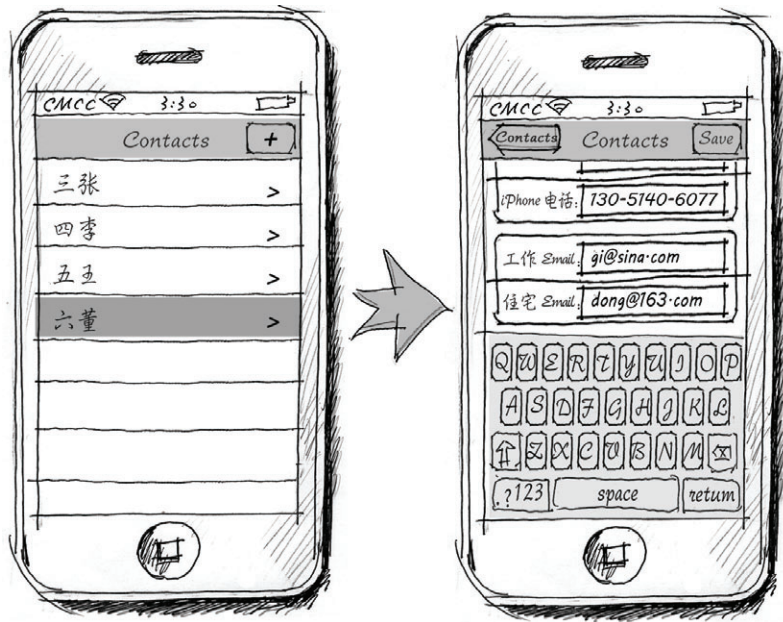


图11-7 修改联系人信息

图11-8用于删除联系人信息。在首页点击某个联系人单元格，导航到该联系人详细信息界面，在这个界面中点击“删除联系人”按钮，就可以删除该联系人并导航回首页。

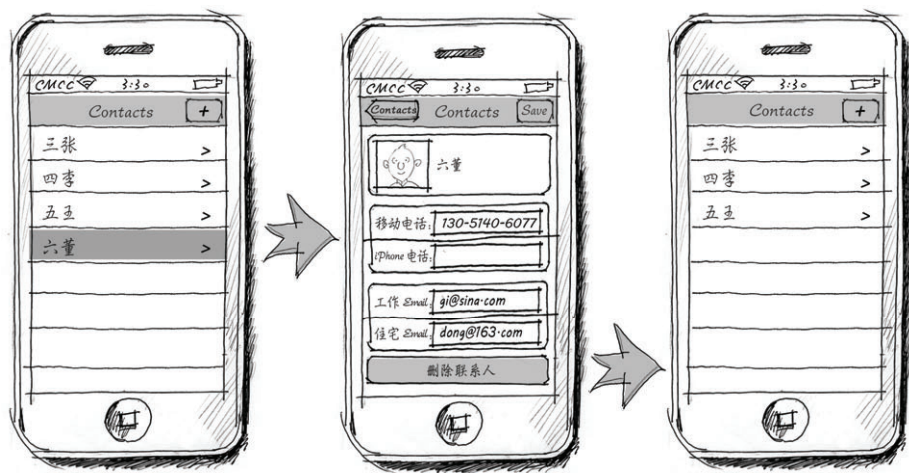


图11-8 删除联系人信息

11.3.1 创建联系人

如图11-9所示，创建联系人基本上都会经历这5个步骤，其中第二个步骤（设置记录中的各个属性值）相对复杂。

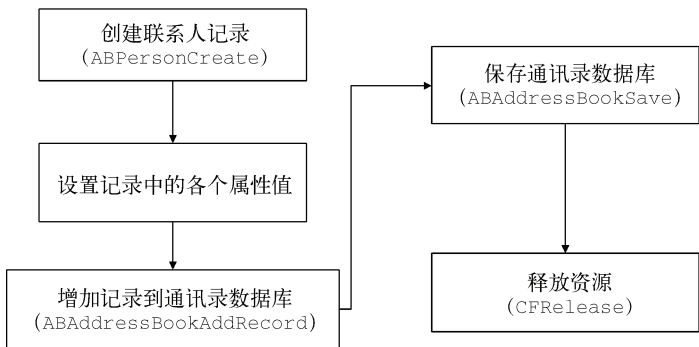


图11-9 创建联系人流程图

一般情况下，在编程过程中主要会用到如下6个函数。

- ❑ **ABPersonCreate**。创建联系人记录。图11-9的第一个步骤会用到这个函数。
- ❑ **ABRecordSetValue**。设定联系人记录中的属性，包括了多值属性和单值属性。图11-9的第二个步骤会用到这个函数。
- ❑ **ABMultiValueCreateMutable**。创建可变多值类型（ABMutableMultiValueRef）。图11-9的第二个步骤会用到这个函数。
- ❑ **ABMultiValueAddValueAndLabel**。按照标签设置多值属性的值，图11-9的第二个步骤会用到这个函数。
- ❑ **ABAddressBookAddRecord**。增加记录到通讯录数据库。图11-9的第三个步骤会用到这个函数。
- ❑ **ABAddressBookSave**。保存未保存的变更到通讯录数据库。图11-9的第四个步骤会用到这个函数，这个函数也适用于删除和修改联系人的情况。

下面我们看看添加界面中添加按钮的触发方法，即AddViewController.m的saveClick:方法：


```

- (IBAction)saveClick:(id)sender {

    CFErrorRef error = NULL;
    ABAddressBookRef addressBook = ABAddressBookCreateWithOptions(NULL, &error);

    ABRecordRef person = ABPersonCreate();

    //保存姓名
    ABRecordSetValue(person, kABPersonFirstNameProperty,
        (__bridge CFTypeRef)self.txtFirstName.text, &error); ①
    ABRecordSetValue(person, kABPersonLastNameProperty,
        (__bridge CFTypeRef)self.txtLastName.text, &error);

    //设置电话号码
    ABMutableMultiValueRef multi = ABMultiValueCreateMutable
        (kABMultiStringPropertyType); ②
    ABMultiValueAddValueAndLabel(multi, (__bridge CFTypeRef)self.txtMobile.text,
        kABPersonPhoneMobileLabel, NULL);
    ABMultiValueAddValueAndLabel(multi, (__bridge CFTypeRef)self.txtIPhone.text,
        kABPersonPhoneIPhoneLabel, NULL);

    //添加电话号码到记录
    ABRecordSetValue(person, kABPersonPhoneProperty, multi, &error);
    CFRelease(multi);

    //设置E-mail属性
    multi = ABMultiValueCreateMutable(kABMultiStringPropertyType);
    ABMultiValueAddValueAndLabel(multi, (__bridge CFTypeRef)self.txtHomeEmail.text,
        kABHomeLabel, NULL);
    ABMultiValueAddValueAndLabel(multi, (__bridge CFTypeRef)self.txtWorkEmail.text,
        kABWorkLabel, NULL);
    //添加E-mail到记录
    ABRecordSetValue(person, kABPersonEmailProperty, multi, &error);
    CFRelease(multi);

    //增加记录到数据库
    ABAddressBookAddRecord(addressBook, person, &error);
    //保存到数据库
    ABAddressBookSave(addressBook, &error);
    CFRelease(person);
    CFRelease(addressBook);

    [self dismissViewControllerAnimated:YES completion:nil];
}

```

在上述代码中，第①行代码使用了ABRecordSetValue函数，该函数的定义如下：

```

bool ABRecordSetValue (
    ABRecordRef record,
    ABPropertyID property,
    CFTypeRef value,
    CFErrorRef *error
);

```

此外，还使用了(__bridge CFTypeRef)self.txtFirstName.text语句将NSString*转化为CFTypeRef类型。

在第②行代码中，ABMultiValueCreateMutable(kABMultiStringPropertyType)语句用于创建可变的 多值类型（即ABMutableMultiValueRef类型）对象。ABMutableMultiValueRef是ABMultiValueRef的可 变类型，在添加和修改的情况下使用的多值类型都应该是可变类型。

ABMultiValueAddValueAndLabel函数的定义如下：

```

bool ABMultiValueAddValueAndLabel (
    ABMutableMultiValueRef multiValue,

```



```

    CTypeRef value,
    CFStringRef label,
    ABMultiValueIdentifier *outIdentifier
);

```

其中参数outIdentifier是指定ID，如果忽略可以传递NULL。ABAddressBookAddRecord函数的定义如下：

```

bool ABAddressBookAddRecord (
    ABAddressBookRef addressBook,
    ABRecordRef record,
    CFErrorRef *error
);

```

ABAddressBookSave函数的定义如下：

```

bool ABAddressBookSave (
    ABAddressBookRef addressBook,
    CFErrorRef *error
);

```

11.3.2 修改联系人

如图11-10所示，修改联系人的过程一般会经历4个步骤。修改联系人与创建联系人非常类似，差别在于前者比后者少了“增加记录到通讯录数据库”这一步，这是因为修改之前该联系人记录已经存在，这条联系人记录是使用ABAddressBookGetPersonWithRecordID函数通过ID从数据库中获取的，而不是通过ABPersonCreate函数新创建的。

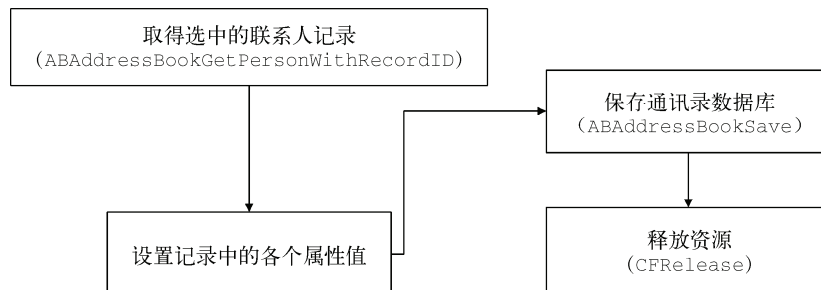


图11-10 修改联系人的流程图

这个过程中用到的函数与创建联系人时也大体一致，不同的是修改联系人时使用的是ABAddressBookGetPersonWithRecordID函数而不会使用ABPersonCreate和ABAddressBookAddRecord函数。

下面我们看看修改界面中“保存”按钮的代码，即DetailViewController.m中的saveClick:方法：

```

- (IBAction)saveClick:(id)sender {

    ABRecordID personID = [self.personIDasNumber intValue];
    CFErrorRef error = NULL;
    ABAddressBookRef addressBook = ABAddressBookCreateWithOptions(NULL, &error);

    ABRecordRef person = ABAddressBookGetPersonWithRecordID(addressBook, personID);

    //设置电话号码
    ABMutableMultiValueRef multi = ABMultiValueCreateMutable(kABMultiStringPropertyType);
    ABMultiValueAddValueAndLabel(multi, (__bridge CTypeRef)self.txtMobile.text,
        kABPersonPhoneMobileLabel, NULL);
    ABMultiValueAddValueAndLabel(multi, (__bridge CTypeRef)self.txtIPhone.text,
        kABPersonPhoneIPhoneLabel, NULL);

    //添加电话号码到数据库

```

```

ABRecordSetValue(person, kABPersonPhoneProperty, multi, &error);
CFRelease(multi);

//设置E-mail属性
multi = ABMultiValueCreateMutable(kABMultiStringPropertyType);
ABMultiValueAddValueAndLabel(multi, (__bridge CTypeRef)self.txtHomeEmail.text,
    kABHomeLabel, NULL);
ABMultiValueAddValueAndLabel(multi, (__bridge CTypeRef)self.txtWorkEmail.text,
    kABWorkLabel, NULL);
//添加E-mail到数据库
ABRecordSetValue(person, kABPersonEmailProperty, multi, &error);
CFRelease(multi);

//保存到数据库
ABAddressBookSave(addressBook, &error);
CFRelease(addressBook);

//导航回根视图控制器ViewController
[self.navigationController popToRootViewControllerAnimated:YES];
}

```

其中ABAddressBookGetPersonWithRecordID函数通过ID获得联系人记录，这个ID是我们从首页传递过来的。该函数的定义如下：

```

ABRecordRef ABAddressBookGetPersonWithRecordID (
    ABAddressBookRef addressBook,
    ABRecordID recordID
);

```

其中参数addressBook是通讯录对象，recordID是记录的ID。

由于person对象是ABAddressBookGetPersonWithRecordID函数获取的，它的所有权不在当前视图控制器（DetailViewController）中，因此不能使用CFRelease(person)释放person，这是与创建联系人不同的地方。

11.3.3 删除联系人

与添加联系人和修改联系人相比，删除联系人比较简单，它的流程如图11-11所示，一般会经历4个步骤。首先通过ABAddressBookGetPersonWithRecordID函数获得选中的联系人记录，然后调用ABAddressBookRemoveRecord函数删除联系人记录，接着将这个变化保存到通讯录数据库，最后释放资源。

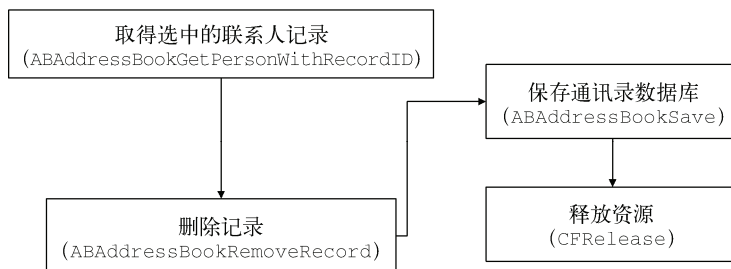


图11-11 删除联系人流程图

下面我们看看修改界面中删除按钮触发的方法，即DetailViewController.m中的deleteClick:方法：

```

- (IBAction)deleteClick:(id)sender {
    ABRecordID personID = [self.personIDAsNumber intValue];
    CFErrorRef error = NULL;
    ABAddressBookRef addressBook = ABAddressBookCreateWithOptions(NULL, &error);

```

```

ABRecordRef person = ABAddressBookGetPersonWithRecordID(addressBook, personID);

//删除记录
ABAddressBookRemoveRecord(addressBook, person, &error);
//保存到数据库
ABAddressBookSave(addressBook, &error);
CFRelease(addressBook);

//导航回根视图控制器ViewController
[self.navigationController popToRootViewControllerAnimated:YES];
}

```

ABAddressBookRemoveRecord函数用于删除联系人记录，它的定义如下：

```

bool ABAddressBookRemoveRecord (
    ABAddressBookRef addressBook,
    ABRecordRef record,
    CFErrorRef *error
);

```

其中addressBook参数是通讯录对象，record参数是要删除的联系人记录。

另外，[self.navigationController popToRootViewControllerAnimated:YES]语句可以导航到首页（即当前界面的根视图），其效果与点击导航栏左侧的返回按钮是一样的。

11.4 高级 API

高级API是在AddressBookUI框架中定义的，它为我们访问通讯录数据提供了UI界面。该框架提供了4个视图控制器及其对应的委托协议。

11.4.1 选择联系人

选择联系人时，需要使用ABPeoplePickerNavigationController控制器，它是选取联系人的导航控制器。如图11-12所示，呈现模态PeoplePicker视图，该视图是ABPeoplePickerNavigationController控制器提供的，可以查询联系人。

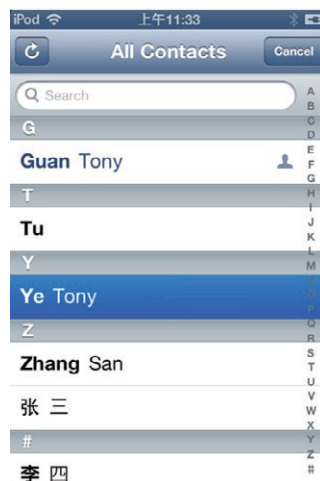


图11-12 PeoplePicker视图

ABPeoplePickerNavigationController对应的委托协议是ABPeoplePickerNavigationController-Delegate，它定义了3个必须实现的方法。

- ❑ **peoplePickerNavigationController:shouldContinueAfterSelectingPerson:**。选中联系人时调用，如果该方法返回YES，则显示选中联系人的详细视图（如图11-13所示），如果返回NO，则不做任何动作。
- ❑ **peoplePickerNavigationController:shouldContinueAfterSelectingPerson:property:identifier:**。如果上面的方法返回YES，则进入如图11-13所示的联系人详细视图后，选择联系人属性时会调用该方法。该方法返回YES，则调用该属性的默认动作。例如，选择的是E-mail属性，则调用iOS内置的E-mail程序发送E-mail，返回NO，则不做任何动作。
- ❑ **peoplePickerNavigationControllerDidCancel:**。点击PeoplePicker视图中的Cancel按钮时调用。



图11-13 联系人详细视图

下面我们通过案例介绍如何使用ABPeoplePickerNavigationController从通讯录中选择联系人。如图11-14所示，当用户点击首页导航栏右侧的+按钮，则呈现模态视图PeoplePicker，选择完成后回到首页并把选择的用户添加到表视图中。

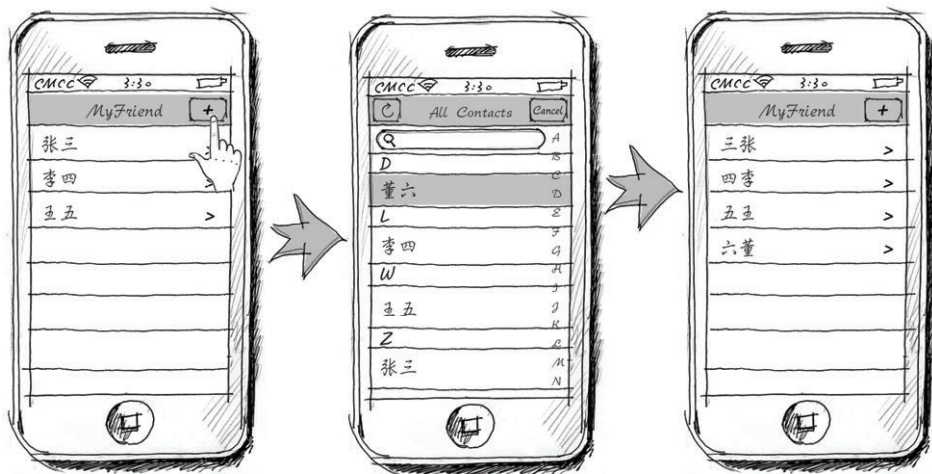


图11-14 选择联系人

在MyFriend工程中，ViewController.h的代码如下：

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

@interface ViewController : UITableViewController
    <ABPeoplePickerNavigationControllerDelegate>

    //保存联系人姓名数组属性
@property(n nonatomic, retain) NSMutableArray *contactNames;
    //保存联系人ID数组属性
@property(n nonatomic, retain) NSMutableArray *contactIDs;

    //选择联系人
- (IBAction)selectContacts:(id)sender;

@end
```

在MyFriend工程中，ViewController.m中selectContacts:方法的代码如下：

```
- (IBAction)selectContacts:(id)sender
{
    ABPeoplePickerNavigationController *peoplePicker =
        [[ABPeoplePickerNavigationController alloc] init];
    peoplePicker.peoplePickerDelegate = self;
    [self presentViewController:peoplePicker animated:YES completion:nil];
}
```

在该方法中实例化ABPeoplePickerNavigationController，需要使用peoplePicker.peoplePickerDelegate = self把当前视图控制器分配给peoplePickerDelegate属性，再使用presentViewController:animated:completion:方法模态呈现PeoplePicker视图。

ABPeoplePickerNavigationController还有一个displayedProperties属性，该属性用于设置联系人详细视图中的显示属性列表，相关代码如下：

```
peoplePicker.displayedProperties = @[ [NSNumber numberWithInt:kABPersonEmailProperty],
    [NSNumber numberWithInt:kABPersonPhoneProperty] ];
```

把要显示的属性常量kABPersonEmailProperty和kABPersonPhoneProperty封装成为NSNumber*对象，再把这些对象放入到NSArray集合中。

在MyFriend工程中，ViewController.m中实现的ABPeoplePickerNavigationControllerDelegate委托方法的代码如下：

```
- (void)peoplePickerNavigationControllerDidCancel:
    (ABPeoplePickerNavigationController *)peoplePicker
{
    [peoplePicker dismissViewControllerAnimated:YES completion:nil];
}

- (BOOL)peoplePickerNavigationController:(ABPeoplePickerNavigationController *)
    peoplePicker shouldContinueAfterSelectingPerson:(ABRecordRef)person ①
{
    NSString *name = CFBridgingRelease(ABRecordCopyCompositeName(person)); ②

    [self.contactNames addObject:name];
    [self.contactIDs addObject:[NSNumber numberWithInt:ABRecordGetRecordID(person)]];

    [peoplePicker dismissViewControllerAnimated:YES completion:nil];

    //取得最后一个放入contactIDs中的ID，也就是刚刚选择的ID
    NSIndexPath *path = [NSIndexPath indexPathForRow:self.contactIDs.count-1
        inSection:0];
```

```

//选中联系人，插入到首页表视图中
NSArray *array = [NSArray arrayWithObject:path];
[self.tableView insertRowsAtIndexPaths:array
                      withRowAnimation:UITableViewRowAnimationRight];
return NO;
}

- (BOOL)peoplePickerNavigationController:(ABPeoplePickerNavigationController *)peoplePicker
shouldContinueAfterSelectingPerson:(ABRecordRef)person
property:(ABPropertyID)property identifier:(ABMultiValueIdentifier)identifier
{
    return NO;
}

```

在第①行代码中，我们使用了`peoplePickerNavigationController:shouldContinue AfterSelectingPerson:`方法。在第②行代码中，我们使用了`ABRecordCopyCompositeName(person)`函数获取联系人的名字，该函数能够返回一个符合用户习惯的姓名格式，例如姓“张”、名“三”的联系人的名字，在中文环境下是“张三”，在英文环境下是“三张”。

第③行代码用于在表视图中以动画形式添加几个行，其中`array`是要插入行的`NSIndexPath`集合，`UITableViewRowAnimationRight`常量在添加时从右边进入，删除时从右边消失。该常量是在`UITableViewRowAnimation`枚举类型中定义的。

11.4.2 显示和修改联系人

显示和修改联系人时，我们会使用`ABPersonViewController`控制器，其中可以显示和编辑联系人，如图11-15所示。

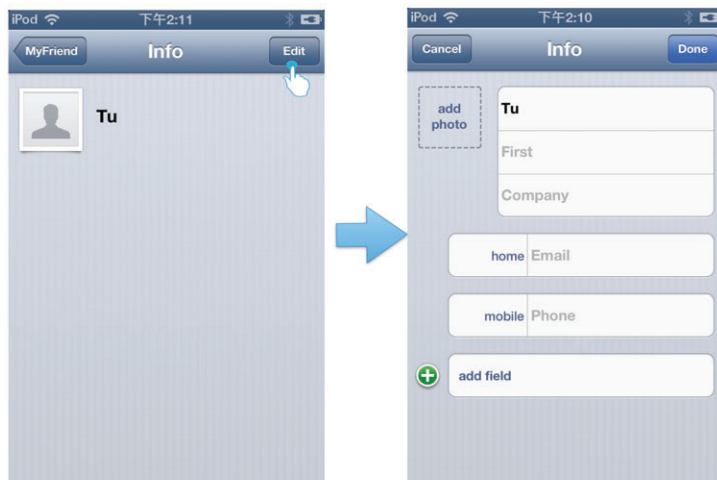


图11-15 显示和修改联系人

`ABPersonViewController`控制器必须在一个导航控制器（`UINavigationController`）内管理。我们使用下面的代码实现`ABPersonViewController`控制器`pvc`：

```
[self.navigationController pushViewController:pvc animated:YES];
```

`ABPersonViewController`对应的委托协议是`ABPersonViewControllerDelegate`，它定义了一个必须实现的方法`personViewController:shouldPerformDefaultActionForPerson:property:identifier:`。该方法

在选择联系人属性时调用, 返回YES时则调用该属性的默认动作, 例如选择的是E-mail属性, 则调用iOS内置的E-mail程序发送E-mail。如果该方法返回NO, 则不做任何动作。

现在我们将上一节的案例进一步完善。点击首页中的某个联系人, 进入该联系人详细信息界面, 从中可以显示和修改联系人, 如图11-16所示。

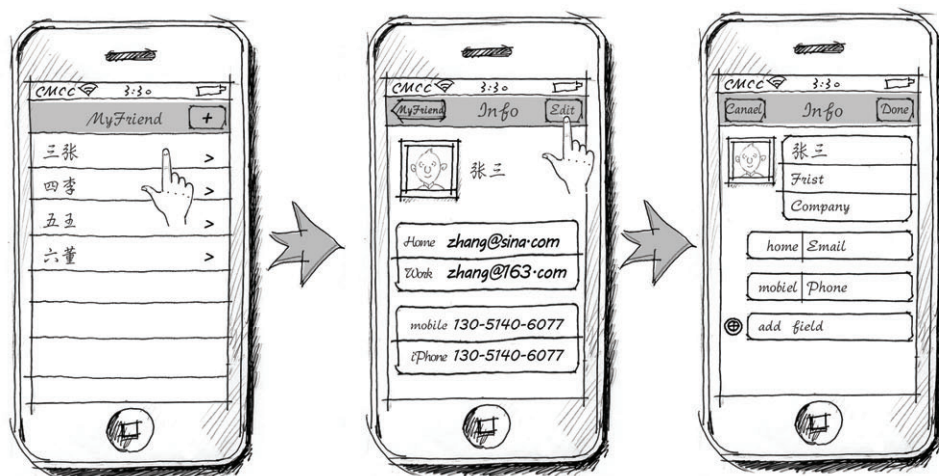


图11-16 显示和修改联系人

在MyFriend工程的ViewController.m文件中, 显示联系人的代码如下:

```

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)
indexPath {

    CFErrorRef error = NULL;
    ABAddressBookRef addressBook = ABAddressBookCreateWithOptions(NULL, &error);

    ABRecordRef person = ABAddressBookGetPersonWithRecordID(addressBook,
        [[self.contactIDs objectAtIndex:indexPath.row] intValue]);           ①

    ABPersonViewController *pvc = [[ABPersonViewController alloc] init];    ②
    pvc.personViewDelegate = self;    ③
    pvc.displayedPerson = person;    ④
    pvc.allowsEditing = YES;    ⑤
    pvc.allowsActions = NO;    ⑥

    pvc.displayedProperties = @[NSNumber numberWithInt:kABPersonEmailProperty],
        [NSNumber numberWithInt:kABPersonPhoneProperty]];    ⑦

    [self.navigationController pushViewController:pvc animated:YES];    ⑧

    CFRelease(addressBook);    ⑨
}

```

在第①行代码中, 我们使用ABAddressBookGetPersonWithRecordID函数取出联系人记录。第②行代码实例化ABPersonViewController控制器, 然后设定控制器的属性, 其中各个属性的含义如下所示。

- ❑ **personViewDelegate**属性。设定保存ABPersonViewControllerDelegate实现对象。
- ❑ **displayedPerson**属性。设定要显示的联系人记录对象。
- ❑ **allowsEditing**属性。设定联系人视图是否可以编辑。如果该属性为YES, 则会在视图导航栏右边出现Edit按钮 (如图11-17所示); 如果该属性为NO, 则不会出现该按钮, 如图11-18所示。



图11-17 allowsEditing = YES



图11-18 allowsEditing = NO

❑ **allowsActions**属性。设定是否可以显示动作按钮，如发送E-mail和FaceTime调用等。如果该属性为YES，则在视图下方显示动作按钮，如图11-19所示；如果该属性为NO，则在视图下方不会显示动作按钮，如图11-20所示。

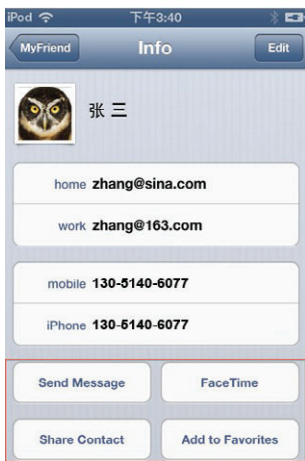


图11-19 allowsActions = YES

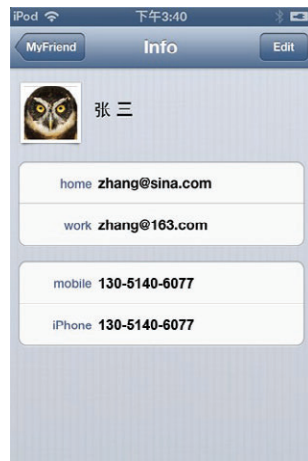


图11-20 allowsActions = NO

❑ **displayedProperties**属性。设定视图中显示的联系人属性。

在MyFriend工程的ViewController.m中，实现ABPersonViewControllerDelegate委托方法的代码如下：

```
- (BOOL)personViewController:(ABPersonViewController *)personViewController
    shouldPerformDefaultActionForPerson:(ABRecordRef)person
    property:(ABPropertyID)property identifier:(ABMultiValueIdentifier)identifierForValue
{
    return YES;
}
```

11.4.3 创建联系人

创建联系人时，可以使用ABNewPersonViewController和ABUnknownPersonViewController控制器，其

中ABNewPersonViewController用于创建新联系人，如图11-21所示。



图11-21 NewPerson视图

ABUnknownPersonViewController用于呈现记录部分信息，这些信息可以用于创建新联系人，或添加到已经存在的联系人。图11-22所示的是UnknownPerson视图，最下面的两个按钮Create New Contact和Add to Existing Contact可以把视图中这些属性的内容添加到新联系人或现有联系人中。

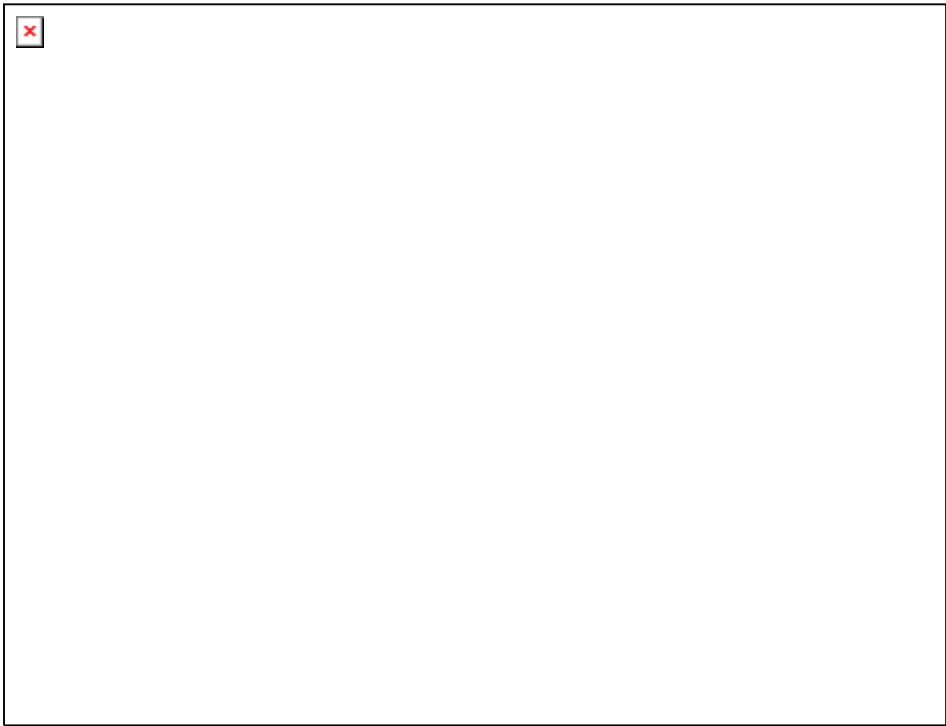


图11-22 UnknownPerson视图

在使用时这两个视图控制器还是有很多差别的。它们与显示和修改联系人的ABPersonViewController控制器类似，必须在一个导航控制器(UINavigationController)内管理。但是ABNewPersonViewController采用模态视图导航方式，呈现时使用presentViewController:animated:completion:方法，关闭时使用dismissViewControllerAnimated:completion:方法。而ABUnknownPersonViewController采用导航堆栈管理，显示时导航控制器发出pushViewController:animated:消息，关闭时导航控制器发出popViewControllerAnimated:消息。

ABNewPersonViewController对应的委托协议是ABNewPersonViewControllerDelegate，它定义了一个必须实现方法newPersonViewController:didCompleteWithNewPerson:。该方法在用户点击Done或Cancel按钮时调用，点击Done按钮会将新建联系人保存到通讯录数据库中，点击Cancel按钮则不保存。

ABUnknownPersonViewController对应的委托协议是ABUnknownPersonViewControllerDelegate，它也定义了一个必须实现的方法unknownPersonViewController:didResolveToPerson:，该方法在创建新联系人或添加到已经存在的联系人时调用。

下面我们通过案例介绍如何使用这两个控制器创建联系人。如图11-23所示，当用户点击首页的ABNewPersonViewController按钮时，进入NewPerson视图；点击ABUnknownPersonViewController按钮时，进入UnknownPerson视图。

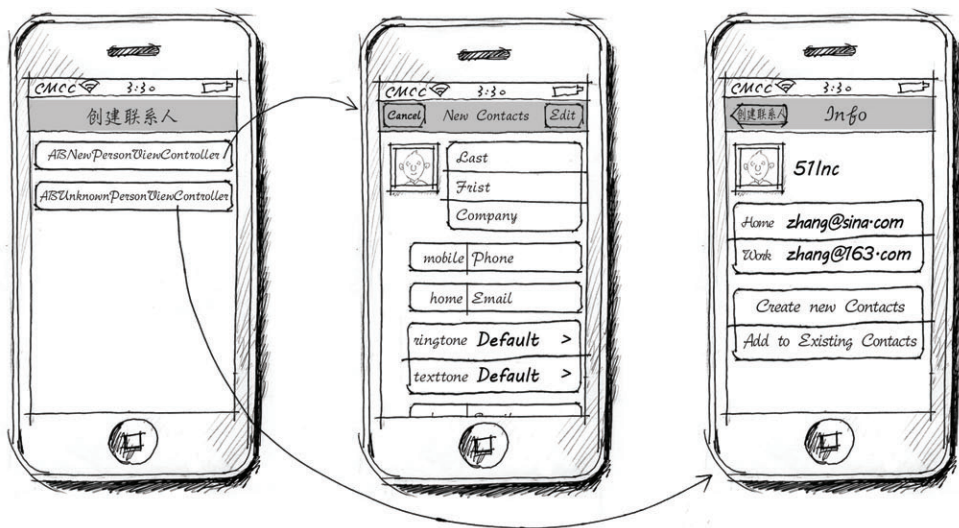


图11-23 创建联系人案例

在Create_Contacts工程中，ViewController.h的代码如下：

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

@interface ViewController : UITableViewController
    <ABNewPersonViewControllerDelegate, ABUnknownPersonViewControllerDelegate>

    // 点击ABNewPersonViewController按钮
    - (IBAction)newClick:(id)sender;

    // 点击ABUnknownPersonViewController按钮
    - (IBAction)unknowClick:(id)sender;

@end
```

在Create_Contacts工程中，ViewController.m的newClick:方法的代码如下：

```
- (IBAction)newClick:(id)sender {

    ABNewPersonViewController *viewController = [[ABNewPersonViewController alloc] init];
    viewController.newPersonViewDelegate = self;
    UINavigationController *newNavigationController =
        [[UINavigationController alloc] initWithRootViewController:viewController];

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

在该方法中，我们实例化了ABNewPersonViewController，并以模态方式呈现该视图。

在ViewController.m中，ABNewPersonViewControllerDelegate委托方法的实现代码如下：

```
- (void)newPersonViewController:(ABNewPersonViewController *)newPersonView
    didFinishWithNewPerson:(ABRecordRef)person
{
    [newPersonView dismissViewControllerAnimated:YES completion:nil];
}
```

在该方法中，我们关闭了模态视图控制器ABNewPersonViewController。在ViewController.m中，unknowClick:方法的实现代码如下：

```
- (IBAction)unknowClick:(id)sender {

    ABRecordRef person = ABPersonCreate();

    //保存姓名
    ABRecordSetValue(person, kABPersonFirstNameProperty, @"51inc", NULL);

    //设置E-mail属性
    ABMutableMultiValueRef multi = ABMultiValueCreateMutable(kABMultiStringPropertyType);
    ABMultiValueAddValueAndLabel(multi, @"eorient@sina.com", kABHomeLabel, NULL);
    ABMultiValueAddValueAndLabel(multi, @"eorient@163.com", kABWorkLabel, NULL);
    //添加E-mail到联系人记录
    ABRecordSetValue(person, kABPersonEmailProperty, multi, NULL);
    CFRelease(multi);

    ABUnknownPersonViewController *viewController =
        [[ABUnknownPersonViewController alloc] init];
    viewController.unknownPersonViewDelegate = self;
    viewController.displayedPerson = person;
    viewController.allowsAddingToAddressBook = YES;

    CFRelease(person);

    [self.navigationController pushViewController:viewController animated:YES];
}
```

在该方法中，第①行代码实例化了ABUnknownPersonViewController控制器，并设定了它需要显示的一些属性值，其中包括了单值属性和多值属性。allowsAddingToAddressBook设定是否允许把视图中的这些属性内容添加到新联系人或已有联系人中：如果其值为YES，则允许添加，如图11-24所示；如果其值为NO，则不允许添加，如图11-25所示。

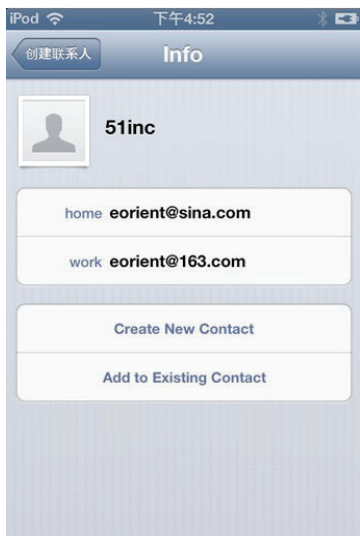


图11-24 allowsAddingToAddressBook=YES



图11-25 allowsAddingToAddressBook= NO

pushViewController:animated: 方法用于显示视图控制器 ABUnknownPersonViewController。在 ViewController.m 中, ABUnknownPersonViewControllerDelegate 委托方法实现代码如下:

```
- (void)unknownPersonViewController:(ABUnknownPersonViewController *)
    unknownCardViewController didResolveToPerson:(ABRecordRef)person
{
    [self.navigationController popViewControllerAnimated:YES];
}
```

在该方法中, 我们调用 popViewControllerAnimated: 方法关闭视图控制器 ABUnknownPersonViewController。

11.5 小结

在 iOS 开发中, 很多需求都涉及通讯录的访问。本章首先介绍了访问通讯录所需要的框架, 然后介绍了如何使用 AddressBook 框架读取联系人信息, 具体包括联系人记录、单值属性、多值属性和图片属性的读取。接下来, 介绍了如何使用该框架将联系人信息写入数据库, 具体包括联系人的创建、修改和删除。最后, 介绍了如何使用高级 API 实现选择联系人、显示和修改联系人以及创建联系人的操作。

Part 2

第二部分

网 络 篇

本 部 分 内 容

- 第 12 章 访问 Web Service
- 第 13 章 定位服务与地图应用

如果数据不在本地，而放在远程服务器上，那么如何取得这些数据呢？服务器能给我们提供一些服务，这些服务大多都基于HTTP协议。HTTP协议基于请求和应答，在需要的时候建立连接提供服务，在不需要的时候断开连接。

本章中，我们将向大家介绍如何使用Web Service以及数据交换格式。

12.1 概述

Web Service技术通过Web协议提供服务，保证不同平台的应用服务可以相互操作，为客户端程序提供不同的服务。类似Web Service的服务不断问世，如Java的RMI（Remote Method Invocation，远程方法调用）、Java EE的EJB（Enterprise JavaBean，企业级JavaBean）、CORBA（Common Object Request Broker Architecture，公共对象请求代理体系结构）和微软的DCOM（Distributed Component Object Model，分布式组件对象模型）等。

目前，3种主流的Web Service实现方案有REST^①、SOAP^②和XML-RPC^③。XML-RPC和SOAP都是比较复杂的技术，XML-RPC是SOAP的前身。与复杂的SOAP和XML-RPC相比，REST风格的Web Service更加简洁，越来越多的Web Service开始采用REST风格设计和实现。例如，亚马逊已经提供了REST风格的Web Service进行图书查找，雅虎提供的Web Service也是REST风格的。本书会重点介绍REST Web Service。

SOAP Web Service数据交换格式是固定的，而REST Web Service数据交换格式是我们自定义的，使用比较方便。在介绍REST Web Service之前，我们先介绍数据交换格式。

12.2 数据交换格式

数据交换格式就像两个人在聊天一样，采用彼此都能听得懂的语言，你来我往，其中的语言就相当于通信中的数据交换格式。有时候，为了防止聊天被人偷听，可以采用暗语。同理，计算机程序之间也可以通过数据加密技术防止“偷听”。

数据交换格式主要分为纯文本格式、XML格式和JSON格式，其中纯文本格式是一种简单的、无格式的数据交换方式。

例如，为了告诉别人一些事情，我会写下如图12-1所示的留言条。

① REST（Representational State Transfer，表征状态转移）是Roy Fielding博士在2000年他的博士论文中提出的一种软件架构风格。
——引自维基百科<http://zh.wikipedia.org/zh-cn/REST>

② SOAP（Simple Object Access Protocol，简单对象访问协议）是交换数据的一种协议规范，用在计算机网络Web服务（Web service）中，交换带结构的信息。——引自维基百科<http://zh.wikipedia.org/wiki/SOAP>

③ XML-RPC是一个远程过程调用（远端程序呼叫）（Remote Procedure Call，RPC）的分布式计算协议，通过XML封装调用函数，并使用HTTP协议作为传送机制。——引自维基百科<http://zh.wikipedia.org/wiki/XML-RPC>

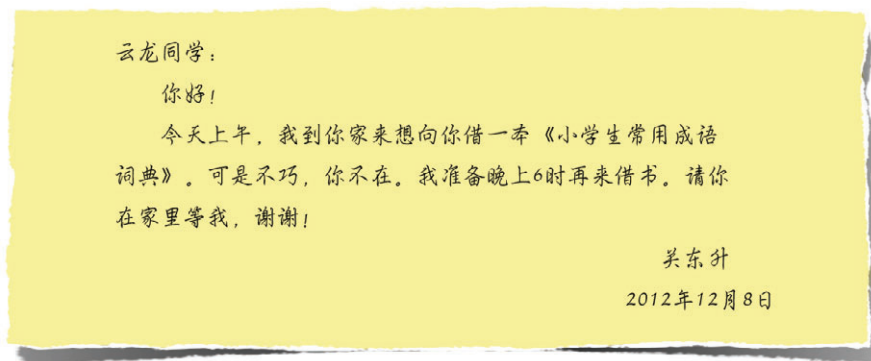


图12-1 留言条

留言条有一定的格式，共有4部分——称谓、内容、落款和时间，如图12-2所示。

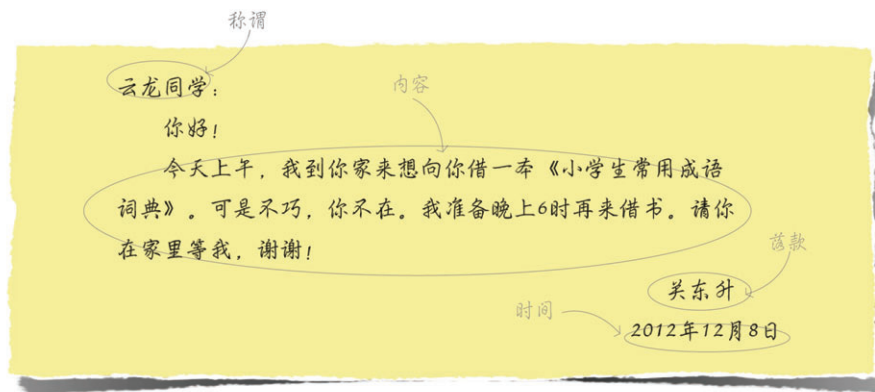


图12-2 留言条格式

如果我们用纯文本格式描述留言条，可以按照如下的形式：

"云龙同学","你好！\n今天上午，我到你家来想向你借一本《小学生常用成语词典》。可是不巧，你不在。我准备晚上6时再来借书。请你在家里等我，谢谢！","关东升","2012年12月08日"

留言条中的4部分数据按照顺序存放，各个部分之间用逗号分割。数据量小的时候，可以采用这种格式。但是随着数据量的增加，问题也会暴露出来，我们可能搞乱它们的顺序，如果各个数据部分能有描述信息就好了。而XML格式和JSON格式可以带有描述信息，它们叫做“自描述的”结构化文档。

将上面的留言条写成XML格式，具体如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>云龙同学</to>
  <content>你好！\n今天上午，我到你家来想向你借一本《小学生常用成语词典》。
    可是不巧，你不在。我准备晚上6时再来借书。请你在家里等我，谢谢！</content>
  <from>关东升</from>
  <date>2012年12月08日</date>
</note>
```

我们看到位于尖括号中的内容（<to>...</to>等）就是描述数据的标识，在XML中称为“标签”。

将上面的留言条写成JSON格式，具体如下：

```
{to:"云龙同学",content:"你好！\n今天上午，我到你家来想向你借一本《小学生常用成语词典》。可是不巧，
  你不在。我准备晚上6时再来借书。请你在家里等我，谢谢！",from:"关东升",date:"2012年12月08日"}
```

数据放置在大括号{}之中,每个数据项目之前都有一个描述名字(如to等),描述名字和数据项目之间用冒号(:)分开。

可以发现,一般来讲,JSON所用的字节数要比XML少,这也是很多人喜欢采用JSON格式的主要原因,因此JSON也被称为“轻量级”的数据交换格式。接下来,我们将重点介绍XML和JSON数据交换格式。

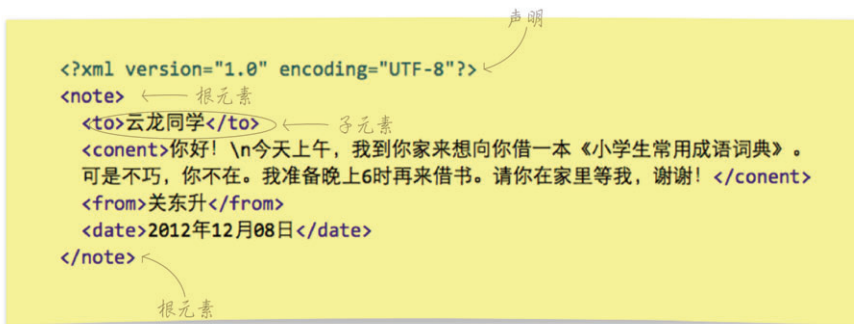
12.2.1 XML文档结构

XML是一种自描述的数据交换格式。虽然XML数据交换格式不如JSON“轻便”,但也是非常重要的数据交换格式,多年来一直用于各种计算机语言中,是老牌的、经典的、灵活的数据交换方式。

在读写XML文档之前,我们需要了解XML文档结构。前面提到的留言条XML文档,它由开始标签<flag>和结束标签</flag>组成,它们就像括号一样,把数据项括起来。这样不难看出,标签<to></to>之间是“称谓”,标签<content></content>之间是“内容”,标签<from></from>之间是“落款”,标签<date></date>之间是“日期”。

XML文档结构要遵守一定的格式规范。XML虽然在形式上与HTML很相似,但是它有着严格的语法规则。只有严格按照规范编写的XML文档才是有效的文档,也称为“格式良好”的XML文档。XML文档的基本架构可以分为下面几部分。

- **声明。**在图12-3中,<?xml version="1.0" encoding="UTF-8"?>就是XML文件的声明,它定义了XML文件的版本和使用的字符集,这里为1.0版,使用中文UTF-8字符。
- **根元素。**在图12-3中,note是XML文件的根元素,<note>是根元素的开始标签,</note>是根元素的结束标签。根元素只有一个,开始标签和结束标签必须一致。
- **子元素。**在图12-3中,to、content、from和date是根元素note的子元素。所有元素都要有结束标签,开始标签和结束标签必须一致。如果开始标签和结束标签之间没有内容,可以写成<from/>,这称为“空标签”。



```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>云龙同学</to>
  <content>你好! \n今天上午,我到你家来想向你借一本《小学生常用成语词典》。
  可是不巧,你不在。我准备晚上6时再来借书。请你在家里等我,谢谢!</content>
  <from>关东升</from>
  <date>2012年12月08日</date>
</note>
```

图12-3 XML文档结构

- **属性。**图12-4是具有属性的XML文档,而留言条的XML文档中没有属性。它定义在开始标签中。在开始标签<Note id="1">中,id="1"是Note元素的一个属性,id是属性名,1是属性值,其中属性值必须放置在双引号或单引号之间。一个元素不能有多个相同名字的属性。
- **命名空间。**用于为XML文档提供名字唯一的元素和属性。例如,在一个学籍信息的XML文档中,需要引用到教师和学生,他们都有一个子元素id,这时直接引用id元素会造成名称冲突,但是如果将两个id元素放到不同的命名空间中就会解决这个问题。图12-5中以xmlns:开头的内容都属于命名空间。
- **限定名。**它是由命名空间引出的概念,定义了元素和属性的合法标识符。限定名通常在XML文档中用作特定元素或属性引用。图12-5中的标签<soap:Body>就是合法的限定名,前缀soap是由命名空间定义的。

```
<?xml version="1.0" encoding="UTF-8"?>
<Notes>
  <Note id="1">属性
    <CDate>2012-12-21</CDate>
    <Content>早上8点钟到公司</Content>
    <UserID>tony</UserID>
  </Note>
  <Note id="2">
    <CDate>2012-12-22</CDate>
    <Content>发布iOSBook1</Content>
    <UserID>tony</UserID>
  </Note>
</Notes>
```

图12-4 有属性的XML文档

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <queryResponse xmlns="http://tempuri.org/">
      <queryResult>
        <Note>
          <UserID>string</UserID>
          <CDate>string</CDate>
          <Content>string</Content>
          <ID>int</ID>
        </Note>
        <Note>
          <UserID>string</UserID>
          <CDate>string</CDate>
          <Content>string</Content>
          <ID>int</ID>
        </Note>
      </queryResult>
    </queryResponse>
  </soap:Body>
</soap:Envelope>
```

图12-5 命名空间和限定名的XML文档

12.2.2 解析XML文档

XML文档操作有“读”与“写”，读入XML文档并分析的过程称为“解析”。事实上，在使用XML开发的过程中，“解析”XML文档占很大的比重。

解析XML文档时，目前有两种流行的模式：SAX和DOM。SAX是一种基于事件驱动的解析模式。解析XML文档时，程序从上到下读取XML文档，如果遇到开始标签、结束标签和属性等，就会触发相应的事件。但是这种解析XML文件的方式有一个弊端，那就是只能读取XML文档，不能写入XML文档，它的优点是解析速度快。iOS重点推荐使用SAX模式解析。

DOM模式将XML文档作为一棵树状结构进行分析，获取节点的内容以及相关属性，或是新增、删除和修改节点的内容。XML解析器在加载XML文件以后，DOM模式将XML文件的元素视为一个树状结构的节点，一次性读入到内存中。如果文档比较大，解析速度就会变慢。但是在DOM模式中，有一点是SAX无法取代的，那就是DOM能够修改XML文档。

iOS SDK提供了两个XML框架，具体如下所示。

- ❑ NSXML。它是基于Objective-C语言的SAX解析框架，是iOS SDK默认的XML解析框架，不支持DOM模式。
- ❑ libxml2。它（<http://xmlsoft.org/>）是基于C语言的XML解析器，被苹果整合在iOS SDK中，支持SAX和DOM模式。

此外，在iOS中解析XML时，还有很多第三方框架可以采用，具体如下所示。

- ❑ TBXML。它是轻量级的DOM模式解析库，不支持XML文档验证和XPath，只能读取XML文档，不能写XML文档，但是解析XML是最快的。
- ❑ TouchXML。它是基于DOM模式的解析库。与TBXML类似，只能读取XML文档，不能写XML文档。
- ❑ KissXML。它是基于DOM模式的解析库，基于TouchXML，主要的不同是可以写入XML文档。
- ❑ TinyXML。它是基于C++语言的DOM模式解析库，可以读写XML文档，不支持XPath。
- ❑ GDataXML。它是基于DOM模式的解析库，由Google开发，可以读写XML文档，支持XPath查询。

从解析性能方面来看，NSXML和TBXML都非常优秀，但是使用NSXML编程时有点麻烦，而TBXML就简单多了。

下面我们通过一个实例介绍使用NSXML和TBXML框架解析XML的过程。现在有一个记录“我的备忘录”信息的Notes.xml文件，它的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<Notes>
  <Note id="1">
    <CDate>2012-12-21</CDate>
    <Content>8点钟到公司</Content>
    <UserID>tony</UserID>
  </Note>
  <Note id="2">
    <CDate>2012-12-22</CDate>
    <Content>发布iOSBook1</Content>
    <UserID>tony</UserID>
  </Note>
  <Note id="3">
    <CDate>2012-12-23</CDate>
    <Content>发布iOSBook2</Content>
    <UserID>tony</UserID>
  </Note>
  <Note id="4">
    <CDate>2012-12-24</CDate>
    <Content>发布iOSBook3</Content>
    <UserID>tony</UserID>
  </Note>
  <Note id="5">
    <CDate>2012-12-25</CDate>
    <Content>发布2016奥运会应用iPhone版本</Content>
    <UserID>tony</UserID>
  </Note>
  <Note id="6">
    <CDate>2012-12-26</CDate>
    <Content>发布2016奥运会应用iPad版本</Content>
    <UserID>tony</UserID>
  </Note>
</Notes>
```

文档中的根元素是Notes，其中有很多子元素Note，而每个Note元素都有一个id属性（表示“备忘录”的序号），以及CDate（表示“备忘录”的日期）、Content（表示“备忘录”的内容）和UserID（表示“备忘录”的创建人ID）这3个子元素。

下面我们以MyNotes（备忘录）应用作为案例，使用不涉及分层架构设计的简单版本，只使用其中的表示层代码，并且只考虑iPhone版本。应用运行界面如图12-6所示，其中数据来源于本地资源文件中的Notes.xml文件。我们需要使用NSXMLParser框架解析XML文档，并将数据放置于界面表视图中。

1. 使用NSXML

NSXML是iOS SDK自带的，也是苹果默认的解析框架，它采用SAX模式解析，是SAX解析模式的代表。NSXML框架的核心是NSXMLParser和它的委托协议NSXMLParserDelegate，其中主要的解析工作是在NSXMLParserDelegate实现类中完成的。委托中定义了很多回调方法，在SAX解析器从上到下遍历XML文档的过程中，遇到开始标签、结束标签、文档开始、文档结束和字符串时就会触发这些方法。这些方法有很多，下面我们列出5个常用的方法。

- ❑ **parserDidStartDocument:**。在文档开始的时候触发。
- ❑ **parser:didStartElement:namespaceURI:qualifiedName:attributes:**。遇到一个开始标签时触发，其中namespaceURI部分是命名空间，qualifiedName是限定名，attributes是字典类型的属性集合。
- ❑ **parser:foundCharacters:**。遇到字符串时触发。
- ❑ **parser:didEndElement:namespaceURI:qualifiedName:**。遇到结束标签时触发。
- ❑ **parserDidEndDocument:**。在文档结束时触发。

上面这5个方法都是按照解析文档的顺序触发的，理解它们的先后顺序很重要。下面我们再通过图12-7所示的UML时序图来了解它们的触发顺序。



图12-6 MyNotes应用

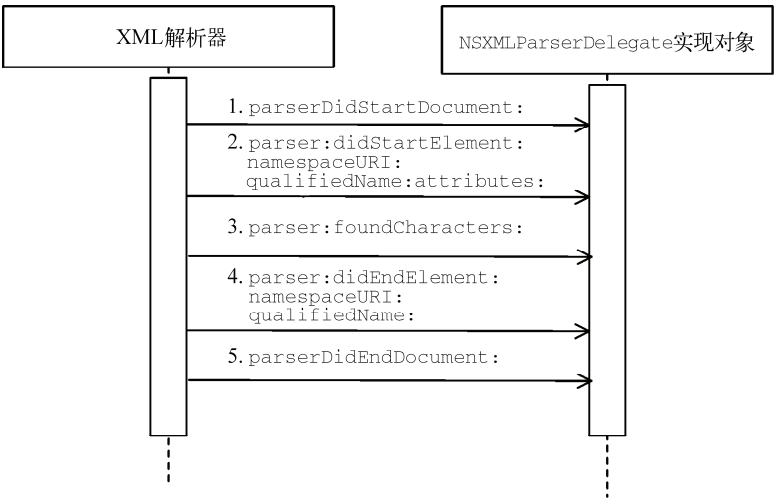


图12-7 UML时序图

就同一个元素而言，触发顺序是按照图12-7所示进行的。在整个解析过程中，它们的触发次数是：1方法和5方法为一对，都只触发一次；2方法和4方法为一对，触发多次；3方法在2和4之间触发，也会多次触发。触发的字符包括换行符和回车符等特殊字符，编程时要注意。

表示层的代码我不想过多修改，所以我编写了一个专门的解析类NotesXMLParser。NotesXMLParser.h文件的代码如下：

```
@interface NotesXMLParser : NSObject <NSXMLParserDelegate>

//解析出的数据，内部是字典类型
@property (strong, nonatomic) NSMutableArray *notes;
//当前标签的名字
@property (strong, nonatomic) NSString *currentTagName;

//开始解析
- (void)start;

@end
```

NotesXMLParser类实现了NSXMLParserDelegate协议，还定义了currentTagName属性，其目的是在图12-7所示的2到4方法执行期间，临时存储正在解析的元素名。在3方法（parser:foundCharacters:）触发时，能够知道目前解析器处于哪个元素之中。

在NotesXMLParser.m中，start方法的代码如下：

```
- (void)start
{
    NSString* path = [[NSBundle mainBundle] pathForResource:@"Notes" ofType:@"xml"];

    NSURL *url = [NSURL fileURLWithPath:path];
    //开始解析XML
    NSXMLParser *parser = [[NSXMLParser alloc] initWithContentsOfURL:url];
    parser.delegate = self;
    [parser parse];
}
```

NSXMLParser是解析类，它有3个构造方法，具体如下所示。

❑ **initWithContentsOfURL:**。可以使用URL对象创建解析对象。本例中采用该方法先从资源文件中加载对象，获得URL对象，再使用URL对象构建解析对象。

❑ **initWithData:**。可以使用NSData创建解析对象。

❑ **initWithStream:**。可以使用IO流对象创建解析对象。

解析对象创建好后，需要指定委托属性delegate为self，然后发送parse消息，开始解析文档。

在NotesXMLParser.m中，parserDidStartDocument:方法的代码如下：

```
// 文档开始的时候触发
- (void)parserDidStartDocument:(NSXMLParser *)parser
{
    _notes = [NSMutableArray new];
}
```

由于parserDidStartDocument:方法只在解析开始时触发一次，因此我们可以在这个方法中初始化解析过程中用到的一些成员变量。

在NotesXMLParser.m中，parser:parseErrorOccurred:方法的代码如下：

```
// 文档出错的时候触发
- (void)parser:(NSXMLParser *)parser parseErrorOccurred:(NSError *)parseError
{
    NSLog(@"%@", parseError);
}
```

出错方法在前面没有介绍，这主要是因为该方法一般在调试阶段使用，实际发布时意义不大。更不要对用户使用UIAlertView提示，用户会被这些专业的错误信息吓坏的。

在NotesXMLParser.m中，parser:didStartElement:namespaceURI:qualifiedName:attributes:方法的代码如下：

```
// 遇到开始标签时触发
- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName
    namespaceURI:(NSString *)namespaceURI
    qualifiedName:(NSString *)qualifiedName
    attributes:(NSDictionary *)attributeDict
{
    _currentTagName = elementName;
    if ([_currentTagName isEqualToString:@"Note"]) {
        NSString *_id = [attributeDict objectForKey:@"id"];
        NSMutableDictionary *dict = [NSMutableDictionary new];
        [dict setObject:_id forKey:@"id"];
        [_notes addObject:dict];
    }
}
```

该方法中需要把elementName参数赋值给成员变量_currentTagName，其中elementName参数是正在解析的元素名字。如果元素名字为Note，取出它的属性id。属性是从attributeDict参数中传递过来的，它是一个字典类型，其中键的名字就是属性名字，值是属性的值。如第①行代码所示，从字典中取出id属性。在第②行代码中，我们实例化一个可变字典对象，用来存放解析出来的Note元素数据。成功解析之后，字典中应该有4对数据，即id、CDate、Content和UserID。第③行代码把id放入可变字典中的。第④行代码把可变字典放入到可变量组集合_notes变量中。

在NotesXMLParser.m中，parser:foundCharacters:方法的代码如下：

```
// 遇到字符串时触发
- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string
{
    // 替换回车符和空格
}
```

```

string = [string stringByTrimmingCharactersInSet:[NSCharacterSet
    whitespaceAndNewlineCharacterSet]];
if ([string isEqualToString:@""]) {
    return;
}
NSMutableDictionary *dict = [_notes lastObject];

if ([_currentTagName isEqualToString:@"CDate"] && dict) {
    [dict setObject:string forKey:@"CDate"];
}

if ([_currentTagName isEqualToString:@"Content"] && dict) {
    [dict setObject:string forKey:@"Content"];
}

if ([_currentTagName isEqualToString:@"UserID"] && dict) {
    [dict setObject:string forKey:@"UserID"];
}
}

```

①
②
③
④

该方法主要用于解析元素文本内容。由于换行符和回车符等特殊字符也会触发该方法，因此在第①行用来过滤掉换行符和回车符，其中stringByTrimmingCharactersInSet:方法是剔除字符方法，[NSCharacterSet whitespaceAndNewlineCharacterSet]指定字符集为换行符和回车符。

在NotesXMLParser.m中，parser:didEndElement:namespaceURI:qualifiedName:方法的代码如下：

```

//遇到结束标签时触发
- (void)parser:(NSXMLParser *)parser didEndElement:(NSString *)elementName
    namespaceURI:(NSString *)namespaceURI
    qualifiedName:(NSString *)qName
{
    self.currentTagName = nil;
}

```

该方法主要清理刚刚解析完成的元素产生的应用，以便不影响接下来的解析。

在NotesXMLParser.m中，parserDidEndDocument:方法的代码如下：

```

//文档结束时触发
- (void)parserDidEndDocument:(NSXMLParser *)parser
{
    [[NSNotificationCenter defaultCenter] postNotificationName:
        @"reloadViewNotification" object:self.notes userInfo:nil];
    self.notes = nil;
}

```

进入该方法就意味着解析完成，需要清理一些成员变量，同时要将数据返回给表示层（表视图控制器）。这里，我们使用通知机制将数据通过广播通知投送回表示层。

在表示层，需要修改的主要是MasterViewController类，其中需要修改的代码如下：

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    self.detailViewController = (DetailViewController *)
        [[self.splitViewController.viewControllers lastObject] topViewController];

    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(reloadView:)
        name:@"reloadViewNotification"
        object:nil];

    NotesXMLParser *parser = [NotesXMLParser new];
    //开始解析
}

```

①
②

```

        [parser start];
    }

#pragma mark - 处理通知

- (void)reloadView: (NSNotification*)notification
{
    NSMutableArray *resList = [notification object];
    self.listData = resList;
    [self.tableView reloadData];
}

```

主要添加的代码用粗体表示，其中第①行代码用于注册一个通知，这样MasterViewController才能在解析完成后接收到投送回来的通知。一旦投送成功，就会触发第④行的reloadView:方法，在该方法中取出数据并重新加载表视图。有关表视图其他方法的实现，我们就不再介绍了。

2. 使用TBXML

使用TBXML解析XML文档时，采用的是DOM模式。通过上面的比较可以发现，它是非常好的解析框架。下面我们介绍一下TBXML框架的用法。TBXML的下载地址是<https://github.com/71squared/TBXML>，技术支持网站是http://www.tbxml.co.uk/TBXML/TBXML_Free.html。

下载完成并解压后，需要将TBXML-Headers和TBXML-Code文件夹添加到工程中，编译时会有很多不支持ARC的编译错误，错误信息如下：

```

<工程目录>/TBXML/TBXML-Code/TBXML+HTTP.m:21:13: error: 'autorelease' is unavailable:
    not available in automatic reference counting mode
    return [request autorelease];
    ^

<工程目录>/TBXML/TBXML-Code/TBXML+HTTP.m:21:13: error:
    ARC forbids explicit message send of 'autorelease'
    return [request autorelease];
    ^~~~~~

<工程目录>/TBXML/TBXML-Code/TBXML+HTTP.m:47:6: error:
    'release' is unavailable: not available in automatic reference counting mode
    [params release];
    ^

<工程目录>/TBXML/TBXML-Code/TBXML+HTTP.m:47:6: error:
    ARC forbids explicit message send of 'release'
    [params release];
    ^~~~~~

<工程目录>/TBXML/TBXML-Code/TBXML+HTTP.m:48:13: error:
    'autorelease' is unavailable: not available in automatic reference counting mode
    return [request autorelease];
    ^

<工程目录>/TBXML/TBXML-Code/TBXML+HTTP.m:48:13: error:
    ARC forbids explicit message send of 'autorelease'
    return [request autorelease];
    ^~~~~~

```

我们需要修改TBXML的源代码。在TBXML中，有一个ARC的开关，需要在工程头文件MyNotes-Prefix.pch中添加宏定义：

```
#define ARC_ENABLED
```

由于TBXML依赖于libz.dylib库，我们还需要在工程Framework中添加这个库，添加完成之后再进行编译就可以了。

我们再看一下代码实现部分。先创建一个NotesTBXMLParser类来解析XML文档。NotesTBXMLParser.h中的代码如下：

```

#import "TBXML.h"

@interface NotesTBXMLParser : NSObject
//解析出的数据内部是字典类型
@property (strong, nonatomic) NSMutableArray *notes;
//开始解析方法
-(void)start;
@end

```

使用TBXML时，需要引入头文件TBXML.h。notes属性用来存放解析的数据。在NotesTBXMLParser.m中，start方法的代码如下：

```

-(void)start
{
    _notes = [NSMutableArray new];

    TBXML* tbxml = [[TBXML alloc] initWithXMLFile:@"Notes.xml" error:nil];           ①
    TBXMLElement * root = tbxml.rootXMLElement;                                   ②

    if (root) {
        TBXMLElement * noteElement = [TBXML childElementNamed:@"Note"
            parentElement:root];                                                    ③
        while ( noteElement != nil) {
            NSMutableDictionary *dict = [NSMutableDictionary new];

            TBXMLElement *CDateElement = [TBXML
                childElementNamed:@"CDate" parentElement:noteElement];
            if ( CDateElement != nil) {
                NSString *CDate = [TBXML textForElement:CDateElement];             ④
                [dict setValue:CDate forKey:@"CDate"];

            }

            TBXMLElement *ContentElement = [TBXML
                childElementNamed:@"Content" parentElement:noteElement];
            if ( ContentElement != nil) {
                NSString *Content = [TBXML textForElement:ContentElement];
                [dict setValue:Content forKey:@"Content"];

            }

            TBXMLElement *UserIDElement = [TBXML
                childElementNamed:@"UserID" parentElement:noteElement];
            if ( UserIDElement != nil) {
                NSString *UserID = [TBXML textForElement:UserIDElement];
                [dict setValue:UserID forKey:@"UserID"];

            }

            //获得ID属性
            NSString *_id = [TBXML valueOfAttributeNamed:@"id"
                forElement:noteElement error:nil];                                  ⑤
            [dict setValue:_id forKey:@"id"];

            [_notes addObject:dict];

            noteElement = [TBXML nextSiblingNamed:@"Note"
                searchFromElement:noteElement];                                     ⑥
        }
    }
    NSLog(@"解析完成...");

    [[NSNotificationCenter defaultCenter] postNotificationName:
        @"reloadViewNotification" object:self.notes userInfo:nil];
    self.notes = nil;
}

```

与NSXML不同，TBXML解析采用DOM模式，不需要事件驱动，使用起来比较简单。第①行代码使用构造方法initWithXMLFile:error:创建一个TBXML对象，这个构造方法是从文件中构造TBXML对象的。TBXML提

供了丰富的构造方法，有类构造方法，也有实例构造方法。下面是它的两个实例构造方法。

❑ **initWithXMLString:error:**。通过XML字符串构造TBXML对象。

❑ **initWithXMLData:error:**。通过NSData数据构造TBXML对象，这个方法非常适合于在网络通信下解析。

第②行代码用于获得文档的根元素对象。由于没有提供XPath支持，解析文档需要从根元素开始，这个过程有点像“剥洋葱皮”。第③行代码中的[TBXML childElementNamed:@"Note" parentElement:root]查找root元素下面的Note元素。在Notes.xml文档中，Note元素应该有很多，但是childElementNamed:parentElement:方法只是返回第一个Note元素，那么如何循环得到其他的Note元素呢？第⑥行代码就是获得同层下一个Note元素的方法，Sibling意为“兄弟”元素，即非父子关系的同层元素。

第④行代码[TBXML textForElement:CDateElement]取得元素的文本内容，这就相当于“剥洋葱皮”已经剥到了“洋葱心”，这个方法就是取出这个“洋葱心”。第⑤行代码[TBXML valueOfAttributeNamed:@"id" forElement:noteElement error:nil]是id属性值。

在视图控制器MasterViewController中引入NotesTBXMLParser.h文件，并修改viewDidLoad方法，就可以运行了：

```
- (void)viewDidLoad
{
    .....

    NotesXMLParser *parser = [NotesXMLParser new];

    NotesTBXMLParser *parser = [NotesTBXMLParser new];
    //开始解析
    [parser start];
}
```

12.2.3 JSON文档结构

JSON (JavaScript Object Notation) 是一种轻量级的数据交换格式。所谓轻量级，是与XML文档结构相比而言的，描述项目的字符少，所以描述相同数据所需的字符个数要少，那么传输速度就会提高，而流量却会减少。

如果留言条采用JSON描述，可以设计成下面的样子：

```
{ "to": "云龙同学",
  "content": "你好! \n今天上午, 我到你家来想向你借一本《小学生常用成语词典》。可是不巧, 你不在。我准备晚上6时再来借书。请你在家里等我, 谢谢!",
  "from": "关东升",
  "date": "2012年12月08日" }
```

由于Web和移动平台开发对流量的要求是要尽可能少，对速度的要求是要尽可能快，而轻量级的数据交换格式JSON就成为理想的数据交换格式。

构成JSON文档的两种结构为对象和数组。对象是“名称-值”对集合，它类似于Objective-C中的字典类型，而数组是一连串元素的集合。

对象是一个无序的“名称/值”对集合，一个对象以{（左括号）开始，}（右括号）结束。每个“名称”后跟一个:（冒号），“名称-值”对之间使用,（逗号）分隔。JSON对象的语法表如图12-8所示。

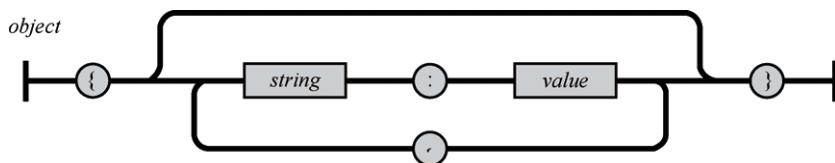


图12-8 JSON对象的语法表

下面是一个JSON对象的例子：

```
{
  "name": "a.htm",
  "size": 345,
  "saved": true
}
```

数组是值的有序集合，以[（左中括号）开始，]（右中括号）结束，值之间使用,（逗号）分隔。JSON数组的语法表如图12-9所示。

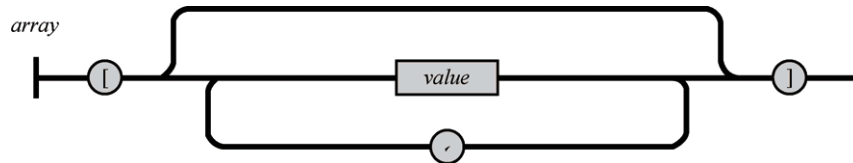


图12-9 JSON数组的语法表

下面是一个JSON数组的例子：

```
["text", "html", "css"]
```

在数组中，值可以是双引号括起来的字符串、数值、true、false、null、对象或者数组，而且这些结构可以嵌套。数组中值的JSON语法结构如图12-10所示。

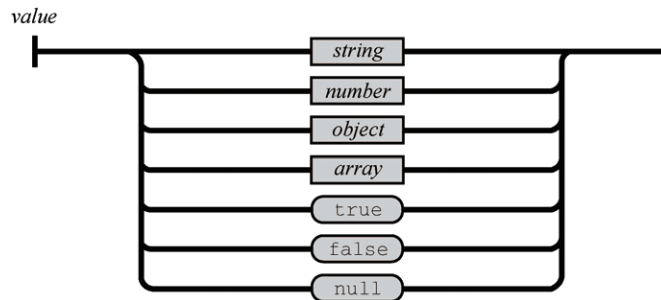


图12-10 JSON值的语法结构图

12.2.4 JSON数据解码

把数据从JSON文档中读取处理的过程称为“解码”过程，即解析和读取过程。由于JSON技术比较成熟，在iOS平台上，也会有很多框架可以进行JSON的编码/解码，具体如下所示。

- ❑ SBJson。它是比较老的JSON编码/解码框架，原名是json-framework。这个框架现在更新仍然很频繁，支持ARC，源码下载地址为<https://github.com/stig/json-framework>。
- ❑ TouchJSON。它也是比较老的JSON编码/解码框架，支持ARC和MRC，源码下载地址为<https://github.com/TouchCode/TouchJSON>。
- ❑ YAJL。它是比较优秀的JSON框架，基于SBJson进行了优化，底层API使用C编写，上层API使用Objective-C编写，使用者可以有多种不同的选择。它不支持ARC，源码下载地址为<http://lloyd.github.com/yajl/>。
- ❑ JSONKit。它是更为优秀的JSON框架，它的代码很小，但是解码速度很快，不支持ARC，源码下载地址为<https://github.com/johnezang/JSONKit>。
- ❑ NextiveJson。它也是非常优秀的JSON框架，与JSONKit的性能差不多，但是在开源社区中没有JSONKit知名度高，不支持ARC，源码下载地址为<https://github.com/nextive/NextiveJson>。

❑ **NSJSONSerialization**。它是iOS 5之后苹果提供的API,是目前非常优秀的JSON编码/解码框架,支持ARC。iOS 5之后的SDK就已经包含这个框架了,不需要额外安装和配置。如果你的应用要兼容iOS 5之前的版本,这个框架不能使用。

如果考虑兼容iOS 5之前的版本,NextiveJson和JSONKit都是不错的选择,它们都不支持ARC,使用起来有点麻烦,需要安装和配置到工程环境中去。如果使用iOS 5之后的版本,NSJSONSerialization应该是首选的。

下面我们通过一个案例MyNotes学习一下NSJSONSerialization的用法。这里重新设计数据结构为JSON格式,其中备忘录信息Notes.json文件的内容如下:

```
{ "ResultCode":0, "Record":[
    { "ID": "1", "CDate": "2012-12-23", "Content": "发布iOSBook0", "UserID": "tony" },
    { "ID": "2", "CDate": "2012-12-24", "Content": "发布iOSBook1", "UserID": "tony" },
    { "ID": "3", "CDate": "2012-12-25", "Content": "发布iOSBook2", "UserID": "tony" },
    { "ID": "4", "CDate": "2012-12-26", "Content": "发布iOSBook3", "UserID": "tony" },
    { "ID": "5", "CDate": "2012-12-27", "Content": "发布iOSBook4", "UserID": "tony" } ] }
```

注意 在iOS平台中,对于JSON文档的结构要求比较严格,每个JSON数据项目的“名称”必须使用双引号括起来,不能使用单引号或没有引号。在下面的代码文档中,“名称”省略了双引号,该文档在iOS平台解析时会出现异常,而在Java等其他平台就没有这些限制,也不会出现异常:

```
{ResultCode:0,Record:[
    {ID:'1',CDate:'2012-12-23',Content:'发布iOSBook0',UserID:'tony'},
    {ID:'2',CDate:'2012-12-24',Content:'发布iOSBook1',UserID:'tony'}}]
```

事实上,NSJSONSerialization使用起来更为简单,只要能确定你的项目使用了iOS 5 SDK就可以了。修改视图控制器MasterViewController的viewDidLoad方法,具体代码如下:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    self.detailViewController = (DetailViewController *)
        [[self.splitViewController.viewControllers lastObject] topViewController];

    [[NSNotificationCenter defaultCenter] addObserver:self
                                                selector:@selector(reloadView:)
                                                name:@"reloadViewNotification"
                                                object:nil];

    NSString* path = [[NSBundle mainBundle] pathForResource:
        @"Notes" ofType:@"json"];
    NSData *jsonData = [[NSData alloc] initWithContentsOfFile:path];

    NSError *error;
    id jsonObj = [NSJSONSerialization JSONObjectWithData:jsonData
                                                         options:NSJSONReadingMutableContainers
                                                         error:&error];
    if (!jsonObj || error) {
        NSLog(@"JSON解码失败");
    }
    self.listData = [jsonObj objectForKey:@"Record"];
}
```

在第①处代码中,我们使用NSJSONSerialization的类方法JSONObjectWithData:options:error:进行解码,其中options参数指定了解析JSON的模式。该参数是枚举类型NSJSONReadingOptions定义的,共有如下3个常量。

❑ **NSJSONReadingMutableContainers**。指定解析返回的是可变的数组或字典。如果以后需要修改结果,这个常量是合适的选择。

❑ **NSJSONReadingMutableLeaves**。指定叶节点是可变字符串。

❑ **NSJSONReadingAllowFragments**。指定顶级节点可以不是数组或字典。

此外，NSJSONSerialization还提供了JSON编码的方法：`dataWithJSONObject:options:error:`和`writeJSONObject:toStream:options:error:`。JSON编码方法的用法与解码方法非常类似，这里我们就不再介绍了，在后面章节中遇到时再介绍。

12.3 REST Web Service

REST被翻译为“表征状态转移”，听起来很抽象，“表征”指客户端看到的页面，页面的跳转就是状态的转移，客户端通过请求URI获得要显示的页面。通常，REST使用HTTP、URI、XML以及HTML这些现有的协议和标准。

REST Web Service是一个使用HTTP并遵循REST原则的Web Service，使用URI来定位资源。与Web Service的数据交互格式使用JSON和XML等。Web Service所支持的HTTP请求方法包括POST、GET、PUT或DELETE等。

注意 REST只是一种设计风格，不是设计规范，更不是行业标准，因此它的设计很灵活。REST Web Service的概念也很宽泛，可以泛指采用HTTP和HTTPS等传输协议并通过URI定位资源的Web Service。数据交互格式通常是JSON或XML格式。

REST Web Service基于HTTP，因此我们先介绍一些HTTP和HTTPS。

12.3.1 HTTP和HTTPS协议

Web Service应用层采用的是HTTP和HTTPS等传输协议，因此我们有必要介绍一下HTTP和HTTPS协议。

1. HTTP协议

HTTP是Hypertext Transfer Protocol的缩写，即超文本传输协议。网络中使用的基本协议是TCP/IP协议，目前广泛采用的HTTP、HTTPS、FTP、Archie和Gopher等是建立在TCP/IP协议之上的应用层协议，不同的协议对应着不同的应用。

Web Service使用的主要协议是HTTP协议，即超文本传输协议。HTTP是一个属于应用层的面向对象的协议，其简捷、快速的方式适用于分布式超文本信息的传输。它于1990年提出，经过多年的使用与发展，得到不断完善和扩展。HTTP协议支持C/S网络结构，是无连接协议，即每一次请求时建立连接，服务器处理完客户端的请求后，应答给客户端然后断开连接，不会一直占用网络资源。

HTTP/1.1协议共定义了8种请求方法：OPTIONS、HEAD、GET、POST、PUT、DELETE、TRACE和CONNECT。作为Web服务器，必须实现GET和HEAD方法，其他方法都是可选的。

GET方法是向指定的资源发出请求，发送的信息“显式”地跟在URL后面。GET方法应该只用在读取数据，例如静态图片等。GET方法有点像使用明信片给别人写信，“信内容”写在外面，接触到的人都可以看到，因此是不安全的。

POST方法是向指定资源提交数据，请求服务器进行处理，例如提交表单或者上传文件等。数据被包含在请求体中。POST方法像是把“信内容”装入信封中，接触到的人都看不到，因此是安全的。

2. HTTPS协议

HTTPS是Hypertext Transfer Protocol Secure，即超文本传输安全协议，是超文本传输协议和SSL的组合，用以提供加密通信及对网络服务器身份的鉴定。

简单地说，HTTPS是HTTP的升级版，与HTTPS的区别是：HTTPS使用https://代替http://，HTTPS使用端口443，而HTTP使用端口80来与TCP/IP进行通信。SSL使用40位关键字作为RC4流加密算法，这对于商业信息的加密是合适的。HTTPS和SSL支持使用X.509数字认证，如果需要的话，用户可以确认发送者是谁。

12.3.2 同步GET请求方法

iOS SDK为HTTP请求提供了同步和异步请求这两种不同的API，而且可以使用GET或POST等请求方法，这里我们先了解其中最为简单的同步GET请求方法。

为了学习这些API的用法，我们还是选择MyNotes应用案例，这次数据来源是Notes.xml（或Notes.json）文件。

我们首先实现查询业务。查询业务请求可以在主视图控制器MasterViewController类中实现，其中MasterViewController.h中的代码如下：

```
#import <UIKit/UIKit.h>
#import "NSString+URLEncoding.h"
#import "NSNumber+Message.h"

@interface MasterViewController : UITableViewController

@property (strong, nonatomic) DetailViewController *detailViewController;
// 保存数据列表
@property (nonatomic, strong) NSMutableArray* listData;

// 重新加载表视图
- (void)reloadView:(NSDictionary*)res;

// 开始请求Web Service
- (void)startRequest;

@end
```

引入头文件NSString+URLEncoding.h是在程序中需要对URL进行编码处理。引入头文件NSNumber+Message.h是将服务器返回的消息代码转换为用户能看懂的消息。MasterViewController.m中的主要代码如下：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.navigationItem.leftBarButtonItem = self.editButtonItem;
    self.detailViewController = (DetailViewController *)
        [[self.splitViewController.viewControllers lastObject] topViewController];
    [self startRequest];
}
①

#pragma mark - Table View
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection:
(NSInteger)section {
    return self.listData.count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        @"Cell" forIndexPath:indexPath];
    NSMutableDictionary* dict = self.listData[indexPath.row];
    cell.textLabel.text = [dict objectForKey:@"Content"];
    cell.detailTextLabel.text = [dict objectForKey:@"CDate"];
    return cell;
}
```

其中第①行代码调用自己的startRequest方法实现请求Web Service。在MasterViewController.m中，startRequest方法的代码如下：

```

/*
 * 开始请求Web Service
 */
-(void)startRequest
{
    NSString *strURL = [[NSString alloc]
        initWithFormat:@"http://iosbook1.com/service/mynotes/webservice.php?email=
        %@&type=%@&action=%@", @"<你的iosbook1.com用户邮箱>", @"JSON", @"query"]; ①

    NSURL *url = [NSURL URLWithString:[strURL URLEncodedString]]; ②

    NSURLRequest *request = [[NSURLRequest alloc] initWithURL:url]; ③

    NSData *data = [NSURLConnection sendSynchronousRequest:request
        returningResponse:nil error:nil]; ④

    NSLog(@"请求完成...");
    NSDictionary *resDict = [NSJSONSerialization JSONObjectWithData:
        data options:NSJSONReadingAllowFragments error:nil];
    [self reloadDataWithDict:resDict]; ⑤
}

```

在上述代码中，第①行代码指定请求的URL，这时URL所指向的Web Service由本书服务器http://www.iOSBook1.com提供，请求的参数全部暴露在URL后面，这是GET请求方法的典型特征。为了能够正确地请求数据，需要开发人员提供合适的参数，具体如下所示。

- ❑ **email**。它是http://iOSBook1.com网站的注册用户邮箱。如果用户使用这些Web Service，首先需要到这个网站注册成为会员，然后提供自己的注册邮箱。
- ❑ **type**。它是数据交互类型。Web Service提供了3种方式的数据：JSON、XML和SOAP。
- ❑ **action**。它指定调用Web Service的一些方法，这些方法有add、remove、modify和query，分别代表插入、删除、修改和查询处理。

第②行代码使用上面的strURL字符串创建NSURL对象，其参数是[strURL URLEncodedString]，strURL字符串又调用了URLEncodedString方法将字符串转换为URL字符串。在网上传输的时候，URL中不能有中文等特殊字符，比如特殊字符“<”必须进行URL编码才能传输，“<”字符的URL编码是“%3C”。URLEncodedString方法是我们自己编写的分类NSString (URLEncoding)，引入NSString+URLEncoding.h的目的就是为了使用这个分类。

第③行代码用于创建NSURLRequest对象，这是HTTP请求对象，它的构造方法是initWithURL:。此外，它还有一个更加复杂的构造方法initWithURL:cachePolicy:timeoutInterval:，使用该构造方法创建NSURLRequest对象的代码如下所示：

```

NSURLRequest *request = [[NSURLRequest alloc] initWithURL:url cachePolicy:
    NSURLRequestReloadIgnoringLocalAndRemoteCacheData timeoutInterval:5.0];

```

其中cachePolicy部分用于设置缓存策略，timeoutInterval部分用于设置请求超时时间，以秒为单位。缓存策略有7种，具体如下所示。

- ❑ **NSURLRequestUseProtocolCachePolicy**。NSURLRequest默认的缓存策略，由协议定义。
- ❑ **NSURLRequestReloadIgnoringCacheData**。忽略缓存直接从原始地址请求。
- ❑ **NSURLRequestReloadIgnoringLocalCacheData**。等同NSURLRequestReloadIgnoringCacheData情况。
- ❑ **NSURLRequestReturnCacheDataElseLoad**。只有在没有缓存数据时从原始地址请求。
- ❑ **NSURLRequestReturnCacheDataDontLoad**。只使用缓存数据，如果不存在缓存，就会抛出异常，请求失败，用于离线模式。

❑ **NSURLRequestReloadIgnoringLocalAndRemoteCacheData**。忽略本地和远程的缓存数据，直接从原始地址请求。

❑ **NSURLRequestReloadRevalidatingCacheData**。验证本地缓存数据与远程数据是否相同，如果相同，则使用本地缓存数据，否则请求远程数据。

第④行代码使用NSURLConnection的sendSynchronousRequest:returningResponse:error:方法进行请求，该方法是异步方法，返回值是NSData类型的数据。同步方法，就是请求过程中线程堵塞到这里，直到Web Service返回应答为止。因此，同步方法的用户体验不好，为了改善用户体验，在iOS 5中增加了可以在其他线程中请求的同步方法sendAsynchronousRequest:queue:completionHandler:，相关代码如下：

```
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
[NSURLConnection sendAsynchronousRequest:request
 queue:queue
 completionHandler:^(NSURLResponse* response, NSData* data, NSError* error) {

    NSLog(@"请求完成...");
    NSDictionary *resDict = [NSJSONSerialization JSONObjectWithData:data
                             options:NSJSONReadingAllowFragments error:nil];
    [self reloadView:resDict];
}];
```

在上述代码中，queue:部分是NSOperationQueue类型的参数，用于指定一个线程对象。completionHandler:部分是执行完成之后调用的代码块。运行程序我们会发现，表视图界面先出来，过一会数据被请求回来填入到表视图中，这个效果事实上已经实现了异步请求。

第⑤行代码在请求完成时调用，用于重新加载表视图中的数据。reloadView:方法的代码如下：

```
//重新加载表视图
- (void)reloadView:(NSDictionary*)res
{
    NSNumber *resultCodeObj = [res objectForKey:@"ResultCode"];
    if ([resultCodeObj integerValue] >= 0)
    {
        self.listData = [res objectForKey:@"Record"];
        [self.tableView reloadData];
    } else {
        NSString *errorStr = [resultCodeObj errorMessage];
        UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"错误信息"
                                                                message:errorStr
                                                                delegate:nil
                                                                cancelButtonTitle:@"OK"
                                                                otherButtonTitles: nil];
        [alertView show];
    }
}
```

从服务器返回的JSON格式数据有两种情况，一种是成功返回，相关代码如下：

```
{"ResultCode":0,"Record":[{"ID":1,"CDate":"2012-12-23","Content":
    "这只是一条测试数据，http://iosbook1.com注册。"}]}
```

另一种是失败返回，相关代码如下：

```
{"ResultCode":-1}
```

其中ResultCode数据项说明调用Web Service的结果。为了减少网络传输，我们只传递消息代码，不传递消息内容。上述代码中的第①行[resultCodeObj errorMessage]也是我们自定义的分类NSNumber (Message)中的方法，它实现了从“代号”到“消息”的转换，引入头文件NSNumber+Message.h就是为了使用这个分类。分类NSNumber (Message)的实现代码如下：


```

@implementation NSNumber (Message)

-(NSString *)errorMessage
{
    NSString *errorStr = @"";
    switch ([self integerValue]) {
        case -7:
            errorStr = @"没有数据。";
            break;
        case -6:
            errorStr = @"日期没有输入。";
            break;
        case -5:
            errorStr = @"内容没有输入。";
            break;
        case -4:
            errorStr = @"ID没有输入。";
            break;
        case -3:
            errorStr = @"数据访问失败。";
            break;
        case -2:
            errorStr = @"您的账号最多能插入10条数据。";
            break;
        case -1:
            errorStr = @"用户不存在, 请到http://iosshare.cn注册。";
            break;
        default:
            break;
    }

    return errorStr;
}

@end

```

分类中的代码很简单, 我们就不再解释了。注意, 如果返回的结果代码小于0, 说明操作失败了。

此外, 我们在前面还提到了一个分类NSString (URLEncoding), 它的作用是编码和解码URL, 它的代码如下:

```

@interface NSString (URLEncoding)

-(NSString *)URLEncodedString;
-(NSString *)URLDecodedString;

@end

@implementation NSString (URLEncoding)

- (NSString *)URLEncodedString
{
    NSString *result = (NSString *)
    CFBridgingRelease(CFURLCreateStringByAddingPercentEscapes
        (kCFAllocatorDefault,
        (CFStringRef)self,
        NULL,
        CFSTR("+$,#[ ]"),
        kCFStringEncodingUTF8));
    return result;
}

- (NSString*)URLDecodedString
{
    NSString *result = (NSString *)
    CFBridgingRelease(CFURLCreateStringByReplacingPercentEscapesUsingEncoding
        (kCFAllocatorDefault,
        (CFStringRef)self,

```

①

②

③

④

```

        CFSTR(""),
        kCFStringEncodingUTF8));
    return result;
}
@end

```

⑤

第①行代码中的CFURLCreateStringByAddingPercentEscapes函数是Core Foundation框架提供的C函数, 可以把内容转换成为URL编码。第②行的参数指定了本身为非法URL字符但不进行编码的字符集合, 例如“、!、*、(、)”和”等符号。第③行参数是本身为合法URL字符但需要进行编码的字符集合。

第④行代码中的CFURLCreateStringByReplacingPercentEscapesUsingEncoding函数是Core Foundation框架提供的C函数, 它与上面的CFURLCreateStringByAddingPercentEscape函数截然相反, 是进行URL解码的。第⑤行的参数指定不进行解码的字符集。

Foundation框架也提供了基于Objective-C进行URL编码和解码的方法。与CFURLCreateStringByAddingPercentEscape函数对应的NSString方法是stringByAddingPercentEscapesUsingEncoding, 与CFURLCreateStringByReplacingPercentEscapesUsingEncoding函数对应的NSString方法是stringByReplacingPercentEscapesUsingEncoding:。由于这些方法不能自定义是否要编码和解码的字符集, 因此没有上面的函数灵活。

12.3.3 异步GET请求方法

同步请求的用户体验不是很好, 因此很多情况下我们会采用异步调用。iOS SDK也提供了异步请求的方法, 而异步请求会使用NSURLConnection委托协议NSURLConnectionDelegate。在请求的不同阶段, 会回调委托对象的方法。NSURLConnectionDelegate协议的方法有如下几个。

❑ **connection:didReceiveData:**。请求成功, 建立连接, 开始接收数据。如果数据量很多, 它会被多次调用。

❑ **connection:didFailWithError:**。加载数据出现异常。

❑ **connectionDidFinishLoading:**。成功完成数据加载, 在connection:didReceiveData方法之后执行。使用异步请求的主视图控制器为MasterViewController。MasterViewController.h中的代码如下:

```

#import <UIKit/UIKit.h>
#import "NSString+URLEncoding.h"
#import "NSNumber+Message.h"

@interface MasterViewController : UITableViewController <NSURLConnectionDelegate>

@property (strong, nonatomic) DetailViewController *detailViewController;
//保存数据列表
@property (nonatomic, strong) NSMutableArray* listData;

//接收从服务器返回的数据
@property (strong, nonatomic) NSMutableData *datas;

//重新加载表视图
-(void)reloadView:(NSDictionary*)res;

//开始请求Web Service
-(void)startRequest;

@end

```

在MasterViewController的定义中, 我们实现了NSURLConnectionDelegate协议。datas属性用来存放从服务器返回的数据, 将其定义为可变类型, 是为了在服务器加载数据的过程中, 数据能不断地追加到这个属性中。MasterViewController.m中的代码如下:

```

/*
 * 开始请求Web Service
 */
-(void)startRequest {

    NSString *strURL = [[NSString alloc]
        initWithFormat:@"http://iosbook1.com/service/mynotes/
        webservice.php?email=%@&type=%@&action=%@",
        @"<你的iosbook1.com用户邮箱>", @"JSON", @"query"];

    NSURL *url = [NSURL URLWithString:[strURL URLEncodedString]];

    NSURLRequest *request = [[NSURLRequest alloc] initWithURL:url];

    NSURLConnection *connection = [[NSURLConnection alloc]
        initWithRequest:request
        delegate:self];

    if (connection) {
        _datas = [NSMutableData new];
    }
}

#pragma mark- NSURLConnection回调方法
-(void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data {    ①
    [_datas appendData:data];
}

-(void) connection:(NSURLConnection *)connection didFailWithError:
    (NSError *)error {

    NSLog(@"%@", [error localizedDescription]);
}

-(void) connectionDidFinishLoading: (NSURLConnection*) connection {    ②
    NSLog(@"请求完成...");
    NSDictionary* dict = [NSJSONSerialization JSONObjectWithData:
        _datas options:NSJSONReadingAllowFragments error:nil];
    [self reloadDataWithDict:dict];
}

```

在第①行的connection:didReceiveData:方法中,我们通过[_datas appendData:data]语句不断地接收服务器端返回的数据。如果加载成功,就回调第②行的connectionDidFinishLoading:方法,这也意味着这次请求结束,这时_datas中的数据是完整的。在这里,我们把数据发送回表示层的视图控制器。

从前两节来看,GET方法与同步请求还是异步请求无关,POST方法也是一样的。因此,在以后的学习中,如果没有特殊情况,我们只介绍异步请求方式。

12.3.4 POST请求方式

在这一节中,我们将介绍POST异步请求方式。使用POST请求的关键是使用NSMutableURLRequest类替代NSURLRequest类。

这里我们把MyNotes应用变成POST方法,此时MasterViewController.m中startRequest方法的代码如下:

```

-(void)startRequest
{

    NSString *strURL = [[NSString alloc] initWithFormat:
        @"http://iosbook1.com/service/mynotes/webservice.php"];    ①

    NSURL *url = [NSURL URLWithString:[strURL URLEncodedString]];

    NSString *post = [NSString stringWithFormat:@"email=%@&type=%@&action=%@",

```

```
        @"<你的iosbook1.com用户邮箱>",@"JSON",@"query"];           ②
NSData *postData = [postDataUsingEncoding:NSUTF8StringEncoding];      ③

NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:url]; ④
[request setHTTPMethod:@"POST"];                                       ⑤
[request setHTTPBody:postData];                                         ⑥

NSURLConnection *connection = [[NSURLConnection alloc]
    initWithRequest:request delegate:self];

if (connection) {
    _datas = [NSMutableData new];
}
}
```

第①行代码用于创建一个URL字符串，在这个URL字符串后面没有参数（即没有?号后面的内容）。请求参数放到请求体中，如第⑥行代码所示，其中postData就是请求参数，是NSData类型。参数字符串是在第②行创建的，最后的参数字符串如下所示：

```
email=<你的iosbook1.com用户邮箱>&type=JSON&action=query
```

第③行将参数字符串转换成NSData类型，编码一定要采用UTF-8。

第④行用于创建可变的请求对象NSMutableURLRequest。因为它是可变对象，所以会有一些set方法。第⑤行用于设置请求方法，这里不能使用GET方法请求。

12.3.5 调用REST Web Service的插入、修改和删除方法

在前面几节中，我们介绍的都是调用REST Web Service的查询方法。在这一节中，我们将介绍MyNotes应用其他的功能实现，包括插入、修改和删除备忘录。

这3个操作调用的Web Service与查询一样，都是http://iosbook1.com/service/mynotes/webservice.php。采用的是HTTP请求方法，建议使用POST方法。这是因为GET请求的是静态资源，数据传输过程也不安全，而POST主要请求动态资源。与查询类似，每个方法调用都要传递很多参数，它们之间的关系见表12-1。

表12-1 方法调用与参数关系

调用方法	type参数	action参数	id参数	date参数	content参数	email参数
add	需要	需要	不需要	需要	需要	需要
modify	需要	需要	需要	需要	需要	需要
remove	需要	需要	需要	不需要	不需要	不需要

表12-1中各个参数的说明如下。

- ❑ **type**。同“查询”调用，是数据交互类型。
- ❑ **action**。同“查询”调用，指定调用Web Service的哪些方法。
- ❑ **id**。备忘录信息中的主键，隐藏在界面之后。当删除和修改时，需要把它传给Web Service。
- ❑ **date**。备忘录信息中的日期字段数据。
- ❑ **content**。备忘录信息中的内容字段数据。
- ❑ **email**。备忘录信息中的用户邮箱字段，通过它可以查询与当前邮箱关联的用户数据。

1. 插入方法调用

插入方法调用主要是在插入视图控制器AddViewController中实现的，具体实现过程与查询业务非常类似。下面我们先看看AddViewController.h文件中的代码：

```

#import "NSString+URLEncoding.h"
#import "NSNumber+Message.h"

@interface AddViewController : UIViewController
    <UITextViewDelegate,NSURLConnectionDelegate> ①

@property (weak, nonatomic) IBOutlet UITextView *textView;

- (IBAction)onclickDone:(id)sender;
- (IBAction)onclickSave:(id)sender;

//接收从服务器返回数据
@property (strong, nonatomic) NSMutableData *datas; ②

//开始请求Web Service
- (void)startRequest; ③

@end

```

我们在原来代码的基础上添加了一些代码。在第①行代码中，我们添加了实现NSURLConnectionDelegate委托协议的声明。在第②行代码中，我们定义了可变NSData类型的datas属性，用来接收服务器返回的数据。第③行是请求服务方法。

AddViewController.m中的主要代码如下：

```

- (void)startRequest
{
    //准备参数
    NSDate *date = [NSDate new];
    NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
    [dateFormatter setDateFormat:@"yyyy-MM-dd"];
    NSString *dateStr = [dateFormatter stringFromDate:date];
    //设置参数
    NSString *post = [NSString stringWithFormat:@"email=%@&type=%@&action=%@&date=%@&content=%@",
        @"<你的iosbook1.com用户邮箱>", @"JSON", @"add", dateStr, _textView.text];

    NSString *strURL = [NSString alloc] initWithFormat:@"http://iosbook1.com/service/mynotes/webservice.php"];
    NSURL *url = [NSURL URLWithString:[strURL URLEncodedString]];
    NSData *postData = [post dataUsingEncoding:NSUTF8StringEncoding];

    NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:url];
    [request setHTTPMethod:@"POST"];
    [request setHTTPBody:postData];

    NSURLConnection *connection = [NSURLConnection alloc]
        initWithRequest:request delegate:self];

    if (connection) {
        _datas = [NSMutableData new];
    }
}

#pragma mark- NSURLConnection回调方法
- (void)connection:(NSURLConnection *)connection didReceiveData:
    (NSData *)data {
    [_datas appendData:data];
}

- (void) connection:(NSURLConnection *)connection didFailWithError: (NSError *)error {
    NSLog(@"%@", [error localizedDescription]);
}

- (void) connectionDidFinishLoading: (NSURLConnection*) connection {
    NSLog(@"请求完成...");
    NSDictionary* dict = [NSJSONSerialization JSONObjectWithData:_datas
        options:NSJSONReadingAllowFragments error:nil];
}

```

```

NSString *message = @"操作成功。";
NSNumber *resultCodeObj = [dict objectForKey:@"ResultCode"];
if ([resultCodeObj integerValue] < 0) {
    message = [resultCodeObj errorMessage];
}
UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"提示信息"
                                                    message:message
                                                    delegate:nil
                                                    cancelButtonTitle:@"OK"
                                                    otherButtonTitles:nil];

[alertView show];
}

```

插入方法的调用关键是请求服务的URL，其中一定要将action设定为add方法的参数，其他的参考表12-1。插入成功后，返回到主视图界面。要想看到刚才添加的数据，需要调用请求查询方法[self startRequest]语句，从viewDidLoad移动到viewWillAppear:方法中，而viewWillAppear:方法在每次显示界面的时候都会调用，其代码如下所示：

```

- (void)viewDidLoad
{
    .....
    {self startRequest};
}

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:YES];

    [self startRequest];
}

```

在这个方法中，实现查询调用会增加网络请求次数，有大量数据返回的情况下，显示界面会比较慢，我们将在后面的内容中介绍相应的解决方法。

2. 修改方法调用

修改方法调用与插入方法调用非常相似，请求过程是一样的，差别在于调用Web Service的参数。我们重点看看它的参数部分。修改方法是在视图控制器DetailViewController中实现的，其中startRequest方法的代码如下：

```

- (void)startRequest
{
    // 准备参数
    NSDate *date = [NSDate new];
    NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
    [dateFormatter setDateFormat:@"yyyy-MM-dd"];
    NSString *dateStr = [dateFormatter stringFromDate:date];
    // 设置参数
    NSString *post = [NSString stringWithFormat:@"email=%@&type=%@&action=%@&date=%@&content=%@&id=%@",
        @"<你的iosbook1.com用户邮箱>",
        @"JSON", @"modify", dateStr, _textView.text, _detailItem];
    ①

    NSString *strURL = [NSString alloc] initWithFormat:
        @"http://iosbook1.com/service/mynotes/webbservice.php"];

    NSURL *url = [NSURL URLWithString:[strURL URLEncodedString]];

    NSData *postData = [post dataUsingEncoding:NSUTF8StringEncoding];

    NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:url];
}

```



```

[request setHTTPMethod:@"POST"];
[request setHTTPBody:postData];

NSURLConnection *connection = [[NSURLConnection alloc]
    initWithRequest:request delegate:self];

if (connection) {
    _datas = [NSMutableArray new];
}
}

```

在上述代码中，第①行代码调用修改方法的参数设置，它需要提供表12-1所述的全部参数，其中id参数是从前一个视图控制器传递来的，日期是取得当前的系统时间，只有内容是从界面的TextView控件中取出来的。

3. 删除方法调用

删除方法调用与前面两个方法调用的过程也非常相似，差别在于调用Web Service的参数。但麻烦的是删除方法调用与查询方法调用是在同一个视图控制器MasterViewController中完成的，我们需要做一些判断。MasterViewController.h文件中的代码修改如下：

```

#define ACTION_QUERY      0           // 查询操作
#define ACTION_REMOVE     1           // 删除操作
#define ACTION_ADD        2           // 添加操作
#define ACTION_MOD        3           // 修改操作

@class DetailViewController;

@interface MasterViewController : UITableViewController <NSURLConnectionDelegate>
{
    int action; //请求动作标识
    int deleteRowId; //删除行号
}
.....
@end

```

为了提高程序的可读性，我们在头文件中定义了4个宏，对应于4种不同的调用。此外，又声明了一个成员变量action，用来记录当前的请求动作。将viewWillAppear:方法修改如下：

```

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:YES];
    action = ACTION_QUERY;
    [self startRequest];
}

```

在上述代码中，我们添加了action = ACTION_QUERY语句来表示当前调用操作是查询。

删除方法调用是在表视图数据源的tableView:commitEditingStyle:forRowAtIndexPath:方法中实现的，其代码如下：

```

- (void)tableView:(UITableView *)tableView commitEditingStyle:
    (UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (editingStyle == UITableViewCellEditingStyleDelete) {
        // 删除数据
        action = ACTION_REMOVE;
        deleteRowId = indexPath.row;
        [self startRequest];
    } else if (editingStyle == UITableViewCellEditingStyleInsert) {
    }
}

```

其中粗体部分是删除方法调用，action = ACTION_REMOVE语句表示当前调用操作是删除。请求Web Service的startRequest方法也需要做一些修改，该方法主要用于判断请求动作标识，其代码如下：

```

-(void)startRequest
{
    NSString *strURL = [[NSString alloc] initWithFormat:
        @"http://iosbook1.com/service/mynotes/webservice.php"];

    NSURL *url = [NSURL URLWithString:[strURL URLEncodedString]];

    NSString *post;
    if (action == ACTION_QUERY) { // 查询处理
        post = [NSString stringWithFormat:@"email=%@&type=%@&action=%@",
            @"<你的iosbook1.com用户邮箱>", @"JSON", @"query"];
    } else if (action == ACTION_REMOVE) { // 删除处理
        NSMutableDictionary* dict = self.listData[deleteRowId];
        post = [NSString stringWithFormat:@"email=%@&type=%@&action=%@&id=%@",
            @"<你的iosbook1.com用户邮箱>", @"JSON", @"remove", [dict objectForKey:@"ID"]];
    }

    NSData *postData = [post dataUsingEncoding:NSUTF8StringEncoding];

    NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:url];
    [request setHTTPMethod:@"POST"];
    [request setHTTPBody:postData];

    NSURLConnection *connection = [[NSURLConnection alloc]
        initWithRequest:request delegate:self];

    if (connection) {
        _datas = [NSMutableData new];
    }
}

```

我们根据请求动作标识判断是哪一个调用，然后设置不同的请求参数。由于请求成功加载完成时，它们都会回调 connectionDidFinishLoading: 方法，因此也需要在该方法中判断请求动作标识。connectionDidFinishLoading: 方法的代码如下：

```

-(void) connectionDidFinishLoading: (NSURLConnection*) connection {
    NSLog(@"请求完成...");

    NSDictionary* dict = [NSJSONSerialization JSONObjectWithData:
        _datas options:NSJSONReadingAllowFragments error:nil];

    if (action == ACTION_QUERY) { // 查询处理
        [self reloadDataWithDict:dict];
    } else if (action == ACTION_REMOVE) { // 删除处理
        NSString *message = @"操作成功。";
        NSNumber *resultCodeObj = [dict objectForKey:@"ResultCode"];

        if ([resultCodeObj integerValue] < 0) {
            message = [resultCodeObj errorMessage];
        }
        UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"提示信息"
            message:message
            delegate:nil
            cancelButtonTitle:@"OK"
            otherButtonTitles: nil];

        [alertView show];
        // 重新查询
        action = ACTION_QUERY;
        [self startRequest];
    }
}

```

①

②

注意第①行代码和第②行代码，它们的作用是在删除完成之后又重新进行了一次查询。

12.4 使用 ASIHTTPRequest 框架

ASIHTTPRequest框架是基于Objective-C的优秀的第三方HTTP框架，支持Mac OS X和iOS下的HTTP开发，其技术支持网站是<http://allseeing-i.com/ASIHTTPRequest/>。该框架具有如下优点：

- ❑ 支持将下载数据放在内存或本地文件中
- ❑ 容易访问请求和应答HTTP头
- ❑ 支持cookie
- ❑ 支持gzip请求或应答
- ❑ 支持缓存
- ❑ 支持同步或异步请求
- ❑ 支持HTTPS

但ASIHTTPRequest框架不支持ARC。如果在ARC下开发，配置起来有点麻烦，本章介绍的案例就是在ARC下开发的。

12.4.1 安装和配置ASIHTTPRequest框架

首先，从<https://github.com/pokeb/asi-http-request/tree>下载ASIHTTPRequest框架，然后打开asi-http-request目录，选择图12-11所示的文件，然后将其添加到我们的iOS工程中。

然后，还需要为工程添加一些支持的类库或框架，具体如下所示：

- ❑ CFNetwork.framework
- ❑ SystemConfiguration.framework
- ❑ MobileCoreServices.framework
- ❑ CoreGraphics.framework
- ❑ libz.dylib

添加过程如图12-12所示。选择工程的TARGETS（编号①所示），再选择Build Phases（编号②所示），接着选择编号③，然后从弹出界面中选择上面的框架或类库，最后点击编号④所示的Add按钮添加即可。

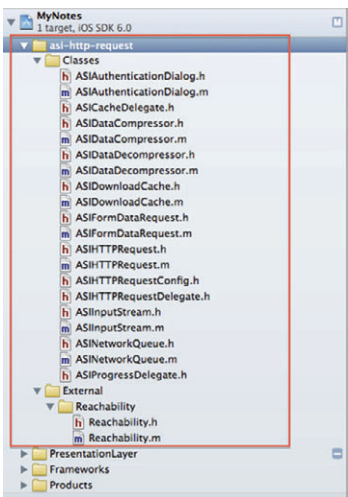


图12-11 在iOS工程中添加ASIHTTPRequest框架

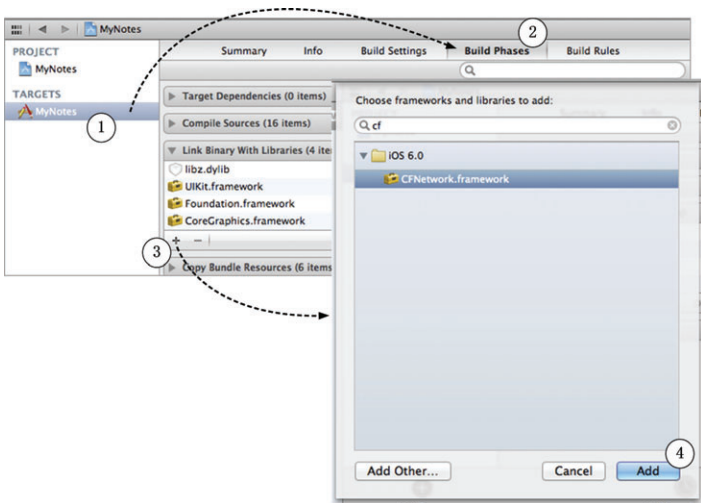


图12-12 添加框架和类库到工程

此时编译一下，看是否有错误。如果我们采用ARC管理内存的话，就会有些小麻烦，具体见图12-13所示的编译错误。

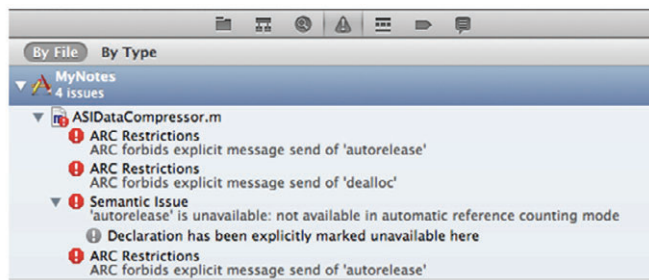


图12-13 编译错误

不难看出，这些编译错误是ARC不支持的错误。这是由于ASIHTTPRequest框架本身不支持ARC技术，它的源代码中使用了MRC（手动管理引用计数）。解决方法是将ASIHTTPRequest框架中的这些源程序文件设置为不采用ARC编译，编译参数是为-fno-objc-arc，如图12-14所示，选择工程的TARGETS（编号①所示），再选择Build Phases（编号②所示），然后再选择ASIHTTPRequest中的文件并双击，在弹出的对话框中输入-fno-objc-arc。

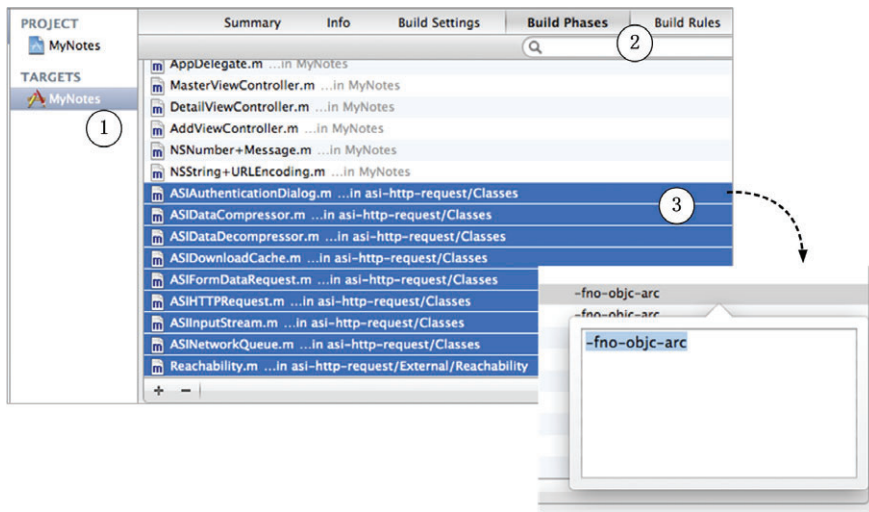


图12-14 设置编译参数-fno-objc-no

12.4.2 同步请求

在ASIHTTPRequest框架中，与HTTP请求相关的类有ASIHTTPRequest和ASIFormDataRequest，其中最常用的是ASIHTTPRequest。ASIFormDataRequest是ASIHTTPRequest的子类，可以发送与HTML类似的表单数据，也可以上传数据，默认采用POST请求方式。当然，也可以采用其他HTTP请求方式。它们都可以进行异步或同步请求。下面我们从最简单的GET同步请求开始介绍。

1. 实现GET同步请求

实现GET同步请求时，我们使用最基本的请求类ASIHTTPRequest就可以了。下面我们还是以MyNotes应用为例来介绍，只考虑查询功能的实现。修改主视图控制器MasterViewController.m中的startRequest方法，具体如下：

```

- (void)startRequest
{
    NSString *strURL = [[NSString alloc] initWithFormat:
        @"http://iosbook1.com/service/mynotes/webservice.php?
        email=%@&type=%@&action=%@",
        @"", @"JSON", @"query"];

    NSURL *url = [NSURL URLWithString:[strURL URLEncodedString]];

    ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];           ①
    [request startSynchronous];                                             ②
    NSLog(@"请求完成...");
    NSError *error = [request error];                                       ③

    if (!error) {
        //NSString *response = [request responseString];

        NSData *data = [request responseData];                             ④
        NSDictionary *resDict = [NSJSONSerialization JSONObjectWithData:data
            options:NSJSONReadingAllowFragments error:nil];
        [self reloadDataWithDict:resDict];
    }
}

```

使用ASIHTTPRequest类时，需要引入头文件ASIHTTPRequest.h。第①行代码用于创建ASIHTTPRequest请求对象，第②行代码调用请求对象的startSynchronous，这是同步请求方法。如果跟踪程序的运行，程序会被堵塞，直到应答回来之后再继续运行，这也就是“同步”的含义。第③行代码调用请求对象的error方法返回一个错误对象，如果这个错误对象非空，说明这个请求过程出现了错误。第④行代码调用请求对象的responseData方法，获得从服务器端应答回来的结果，它的返回值为NSData类型，这种类型非常适合NSJSONSerialization解析。如果想返回字符串，可以调用请求对象的responseString方法。

2. 实现POST同步请求

发送POST方法的请求时，无论同步请求还是异步请求，都将使用ASIFormDataRequest类，它们稍微有些不同。本节中，我们只介绍同步请求下使用ASIFormDataRequest类来发送POST方法。修改主视图控制器MasterViewController.m中的startRequest方法，具体如下：

```

- (void)startRequest
{
    NSString *strURL = @"http://iosbook1.com/service/mynotes/webservice.php";           ①

    NSURL *url = [NSURL URLWithString:[strURL URLEncodedString]];

    ASIFormDataRequest *request = [ASIFormDataRequest requestWithURL:url];             ②
    [request setPostValue:@" " forKey:@"email"];                                     ③
    [request setPostValue:@"JSON" forKey:@"type"];                                 ④
    [request setPostValue:@"query" forKey:@"action"];                             ⑤

    [request startSynchronous];                                                       ⑥
    NSLog(@"请求完成...");

    NSError *error = [request error];

    if (!error) {
        //NSString *response = [request responseString];

        NSData *data = [request responseData];
        NSDictionary *resDict = [NSJSONSerialization JSONObjectWithData:data
            options:NSJSONReadingAllowFragments error:nil];
        [self reloadDataWithDict:resDict];
    }
}

```

使用ASIFormDataRequest时,需要引入头文件ASIFormDataRequest.h。从第①行代码的URL后面看,它没有跟参数,不再是GET方法了。第②行代码是ASIFormDataRequest请求对象。第③行代码至第⑤行代码用于设置POST方法的参数,注意“名字”和“值”的顺序不要颠倒了。第⑥行代码调用startSynchronous方法发送同步请求。

如果我们想发送除了GET和POST以外的请求方法,可以使用[request setRequestMethod:@"PUT"]语句设置,其中PUT是请求方法。

12.4.3 异步请求

我们运行一下上一节的代码,如果网很慢,查询的时候会一直黑屏,直到请求结束界面才出现结果,这样用户体验很不好。因此,同步请求一般只是在某个子线程中使用,而不在主线程中使用。异步请求的用户体验要比同步请求好,因此一般情况下异步请求用得更多。在等待过程中,在状态栏上会出现网络等待指示器的经典旋转小图标,如图12-15所示,这可以使用ASIHTTPRequest异步请求来实现,不用自己额外编写代码。



图12-15 显示网络等待指示器

ASIHTTPRequest和ASIFormDataRequest两个请求类都可以发送异步请求,而后者继承了前者的异步请求方法,所以我们重点介绍ASIHTTPRequest的异步请求。异步请求后的处理是通过回调委托对象的方法requestFinished:或requestFailed:实现的。修改主视图控制器MasterViewController.m中的startRequest方法,具体如下:

```

- (void)startRequest
{
    NSString *strURL = [NSString alloc] initWithFormat:
        @"http://iosbook1.com/service/mynotes/webservice.php?email=
        %@" type=%@" action=%@",
        @"<你的iosbook1.com用户邮箱>", @"JSON", @"query"];

    NSURL *url = [NSURL URLWithString:[strURL URLEncodedString]];

    ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
    [request setDelegate:self];
    [request startAsynchronous];
}

- (void)requestFinished:(ASIHTTPRequest *)request
{
    NSData *data = [request responseData];
    NSDictionary *resDict = [NSJSONSerialization JSONObjectWithData:data
        options:NSJSONReadingAllowFragments error:nil];
    [self reloadDataWithDict:resDict];
}

- (void)requestFailed:(ASIHTTPRequest *)request
{
    NSError *error = [request error];
    NSLog(@"%@", [error localizedDescription]);
}

```

在上述代码中,第①行代码设置委托对象为self,然后通过第②行代码发起异步请求,服务器端返回成功,则回调第③行的requestFinished:方法,失败则回调第④行的requestFailed:方法,它们的参数都是ASIHTTPRequest类型。这两个方法是默认的回调方法,我们也可以自定义回调方法。因此上面的代码也可以改为如下形式:

```

- (void)startRequest
{
    .....
}

```



```

        [request setDidFinishSelector:@selector(requestSuccess:)];
        [request setDidFailSelector:@selector(requestError:)];

        [request startAsynchronous];
    }
- (void)requestSuccess:(ASIHTTPRequest *)request
{
    NSData *data = [request responseData];
    NSDictionary *resDict = [NSJSONSerialization JSONObjectWithData:data
                             options:NSJSONReadingAllowFragments error:nil];
    [self reloadDataWithDict:resDict];
}
- (void)requestError:(ASIHTTPRequest *)request
{
    NSError *error = [request error];
    NSLog(@"%@", [error localizedDescription]);
}

```

我们通过请求对象的setDidFinishSelector:方法指定要回调成功方法，setDidFailSelector:方法指定要回调失败方法。

在异步请求中，为了使代码更加整洁，可以使用代码块并在代码块中指定回调方法。主视图控制器MasterViewController.m中的startRequest方法如下：

```

- (void)startRequest
{
    NSString *strURL = [[NSString alloc] initWithFormat:
        @"http://iosbook1.com/service/mynotes/webservice.php?email=
        %@"&type=%@"&action=%@",
        @"<你的iosbook1.com用户邮箱>", @"JSON", @"query"];

    NSURL *url = [NSURL URLWithString:[strURL URLEncodedString]];

    __weak ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];           ①
    [request setCompletionBlock:^(
        NSData *data = [request responseData];                                   ②
        NSDictionary *resDict = [NSJSONSerialization JSONObjectWithData:data
                                options:NSJSONReadingAllowFragments error:nil];    ③
        [self reloadDataWithDict:resDict];

    )];

    [request setFailedBlock:^(
        NSError *error = [request error];                                         ④
        NSLog(@"%@", [error localizedDescription]);

    )];
    [request startAsynchronous];
}

```

第②行代码中的setCompletionBlock:^(...)和第④行代码中的setFailedBlock:^(...)调用的就是代码块，这两个方法分别在成功完成和失败时回调。使用代码块有一些小麻烦，它可能会造成循环保持（retain cycle）问题。循环保持是一个内存问题，假设A对象保持了B对象，B对象也保持了A对象，A和B将无法释放。为了解决代码块循环保持问题，我们在第①行代码声明ASIHTTPRequest对象之前使用__weak关键字，它的意思是ASIHTTPRequest对象是弱引用的，不进行保持处理，这种解决方式适合于iOS 5之后的ARC内存管理方式。如果是MRC内存管理方式，需要在ASIHTTPRequest对象之前使用__block关键字。

12.4.4 使用请求队列

为了实现多线程并发请求网络能力，ASIHTTPRequest被设计成NSOperation的子类。ASINetworkQueue

被设计成为NSOperationQueue的子类，它们的类图如12-16所示。

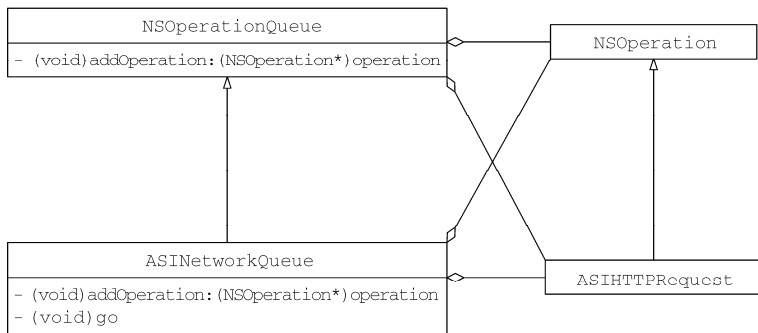


图12-16 ASI队列类图

从图12-16可见，除了继承关系外，还有组成关系。NSOperationQueue队列可以由NSOperation或ASIHTTPRequest组成，而ASINetworkQueue也可以由NSOperation或ASIHTTPRequest组成。NSOperation和NSOperationQueue是苹果Foundation框架提供的，ASIHTTPRequest和ASINetworkQueue是ASI框架提供的。

如果NSOperationQueue是线程管理器，NSOperation就相当于一个线程，它们被添加到NSOperationQueue队列中有序地执行。ASINetworkQueue和ASIHTTPRequest也有这样的概念，只是ASINetworkQueue线程管理器能够提供更多与网络相关的服务，如获得下载进度等。在本节中，我们将重点介绍ASINetworkQueue管理请求队列。

下面通过一个例子介绍一下请求队列的用法。我们设计了如图12-17所示的应用，当用户点击GO按钮时，从服务器同时下载两张图片并将其显示在界面中。

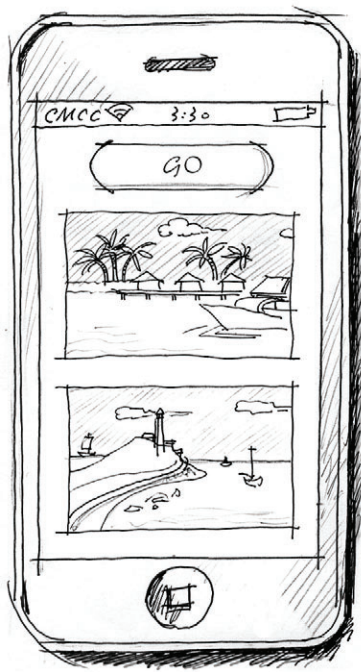


图12-17 设计原型图

UI部分的设计步骤就不再介绍了。下面我们直接看看主视图控制器，ViewController.h中的代码如下：

```
#import "ASIHTTPRequest.h"
#import "ASINetworkQueue.h"
#import "NSNumber+Message.h"
#import "NSString+URLEncoding.h"

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UIImageView *imageView1;
@property (weak, nonatomic) IBOutlet UIImageView *imageView2;
@property (strong) ASINetworkQueue *networkQueue;

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

需要引入ASI框架的两个头文件ASIHTTPRequest.h和ASINetworkQueue.h，其中imageView1和imageView2是与界面对应的两个图片视图控件。此外，还定义了ASINetworkQueue类型的networkQueue属性。

下面我们直接看看主视图控制器ViewController.m中GO按钮的调用方法，具体如下：

```
- (IBAction)onClick:(id)sender {

    if (!_networkQueue) {
        _networkQueue = [[ASINetworkQueue alloc] init];
    }

    //停止以前的队列
    [_networkQueue cancelAllOperations];

    //创建ASI队列
    [_networkQueue setDelegate:self];
    [_networkQueue setRequestDidFinishSelector:@selector(requestFinished:)];
    [_networkQueue setRequestDidFailSelector:@selector(requestFailed:)];
    [_networkQueue setQueueDidFinishSelector:@selector(queueFinished:)];

    for (int i=1; i<3; i++) {
        NSString *strURL = [[NSString alloc]
            initWithFormat:@"http://iosbook1.com/service/download.php?email=%@&FileName=
            test%i.jpg", @"<你的iosbook1.com用户邮箱>", i];
        NSURL *url = [NSURL URLWithString:[strURL URLEncodedString]];
        ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
        request.tag = i;
        [_networkQueue addOperation:request];

        [_networkQueue go];
    }
}
```

在上述代码中，第①行代码创建ASINetworkQueue对象，第②行代码[_networkQueue cancelAllOperations]用于停止以前的队列。第③行代码设置请求成功的回调方法，第④行代码设置请求的失败回调方法，第⑤行代码设置队列完成（所有请求完成）回调的方法。由于我们有两个请求，无论成功还是失败，它们都回调相同的方法，如何区分它们呢？可以借助请求对象的tag属性来区分，它是一个整数属性，如第⑥行代码所示。另外，如果想传递更多的数据给请求对象，可以使用属性userInfo，这是一个字典类型的属性，我们可以放置很多数据到这个字典中。第⑦行代码将请求对象放入到队列中，第⑧行代码执行队列。

我们再看看它们的回调方法，相关代码如下：

```
- (void)requestFinished:(ASIHTTPRequest *)request
{
    NSData *data = [request responseData];
    NSError *error;
    NSDictionary *resDict = [NSJSONSerialization JSONObjectWithData:data
        options:NSJSONReadingAllowFragments error:&error];
}
```

```

if (!resDict) {
    UIImage *img = [UIImage imageWithData:data];
    if (request.tag == 1) {
        _imageView1.image = img;
    } else {
        _imageView2.image = img;
    }
} else {
    NSNumber *resultCodeObj = [resDict objectForKey:@"ResultCode"];
    NSString *errorStr = [resultCodeObj errorMessage];
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"错误信息"
                                                         message:errorStr
                                                         delegate:nil
                                                         cancelButtonTitle:@"OK"
                                                         otherButtonTitles:nil];
    [alertView show];
}
if ([_networkQueue requestsCount] == 0) {
    [self setNetworkQueue:nil];
}
NSLog(@"请求成功");
}
- (void)requestFailed:(ASIHTTPRequest *)request
{
    NSError *error = [request error];
    NSLog(@"%@",[error localizedDescription]);
    if ([_networkQueue requestsCount] == 0) {
        [self setNetworkQueue:nil];
    }
    NSLog(@"请求失败");
}
- (void)queueFinished:(ASIHTTPRequest *)request
{
    if ([_networkQueue requestsCount] == 0) {
        [self setNetworkQueue:nil];
    }
    NSLog(@"队列完成");
}

```

在上述代码中，requestFinished:方法是成功请求对象的回调方法，因此有两个请求对象会被调用两次。在第①行代码中，我们根据GO按钮点击事件设定的请求对象的tag属性，判断是哪个请求对象的回调，进而显示不同的图片视图。第②行代码中的[_networkQueue requestsCount]可以判断队列中请求对象的个数。

12.4.5 上传数据

数据上传是通过ASIFormDataRequest类实现的。前面提到过，ASIFormDataRequest相当于HTML的表单，通过Submit按钮提交给服务器。因此，ASIFormDataRequest请求对象的作用相当于提交表单数据，默认采用的是POST请求方法。

下面我们通过一个实例介绍一下如何使用ASIFormDataRequest实现上传。我们设计了如图12-18所示的应用，当用户点击GO按钮时，从本地上传图片，然后再下载这张图片并将其显示在界面中。

有关UI部分的设计步骤就不再介绍了，我们直接看看主视图控制器ViewController.h中的代码，具体如下：

```

- (IBAction)onClick:(id)sender {

    NSString *strURL1 = @"http://iosbook1.com/service/upload.php ";
    NSURL *url1 = [NSURL URLWithString:[strURL1 URLEncodedString]];

    ASIFormDataRequest *request = [ASIFormDataRequest requestWithURL:url1];
    [request setPostValue:@"<你的iosbook1.com用户邮箱>"
                     forKey:@"email"];
}

```

```

NSString *path = [[NSBundle mainBundle] pathForResource:@"test1"
                ofType:@"jpg"];
[request setFile:path forKey:@"file"];
[request setDelegate:self];
[request setDidFinishSelector:@selector(requestSuccess:)];
[request setDidFailSelector:@selector(requestError:)];
[request startAsynchronous];
}

```

②



图12-18 上传应用的设计原型图

从上面的代码可见，提交数据时，我们使用ASIFormDataRequest请求对象和setPostValue等方法。上传数据使用第②行代码，其中setFile方法用于设置文件的路径，forKey方法用于设置键名字，这个键名字相当于HTML表单中的上传控件>，name是与forKey对应的。其他两个回调方法的代码如下：

```

- (void)requestSuccess:(ASIHTTPRequest *)request
{
    NSData *data = [request responseData];
    NSError *error;
    NSDictionary *resDict = [NSJSONSerialization JSONObjectWithData:data
                        options:NSJSONReadingAllowFragments error:&error];
    if (!resDict) {
        // 查看图片
        [self seeImage];
    } else {
        NSNumber *resultCodeObj = [resDict objectForKey:@"ResultCode"];

        NSString *errorStr = [resultCodeObj errorMessage];
        UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"错误信息"
                        message:errorStr
                        delegate:nil
                        cancelButtonTitle:@"OK"
                        otherButtonTitles:nil];

        [alertView show];
    }
}

```

①

```

}
- (void)requestError: (ASIHTTPRequest *)request
{
    NSError *error = [request error];
    NSLog(@"%@", [error localizedDescription]);
}

```

在请求成功方法requestSuccess:中,第①行代码实现上传成功后,把刚刚上传的图片下载下来。seeImage方法的代码如下:

```

- (void)seeImage {
    NSString *strURL = [[NSString alloc]
        initWithFormat:@"http://iosbook1.com/service/download.php?email=
        %@"&FileName=1.jpg",
        @"<你的iosbook1.com用户邮箱>"];
    NSURL *url = [NSURL URLWithString:[strURL URLEncodedString]];
    __weak ASIHTTPRequest *request = [ASIHTTPRequest requestWithURL:url];
    [request setCompletionBlock:^(
        NSData *data = [request responseData];
        UIImage *img = [UIImage imageWithData:data];
        _imageView1.image = img;
    )];
    [request setFailedBlock:^(
        NSError *error = [request error];
        NSLog(@"%@", [error localizedDescription]);
    )];
    [request startAsynchronous];
}

```

在上述代码中,第①行代码提供下载文件的URL,其中下载的图片名是1.jpg,这是一个固定的名字。请求下载图片使用ASIHTTPRequest异步请求方式来实现。

12.5 反馈网络信息改善用户体验

在使用有网络服务的应用时,用户希望看到应用运行的进度、网络状态等反馈信息。作为网络应用的开发者,我们必须注意这些问题,以便给用户提供良好的用户体验。

12.5.1 iOS 6 表视图刷新控件的使用

首先,看看MyNotes应用中的一个缺陷。在显示备忘录数据的表视图界面中,数据请求放在viewWillAppear:方法中,相关代码可以查看源代码“12.3.5 MyNotes(插入、删除、修改)”。viewWillAppear:方法的代码如下:

```

- (void)viewWillAppear: (BOOL) animated
{
    [super viewWillAppear:YES];
    action = ACTION_QUERY;
    [self startRequest];
}

```

在上述代码中,[self startRequest]语句用于网络请求。这条语句放在viewWillAppear:方法中的最大问题是,每次显示这个界面时,都会发起网络请求。这样设计的目的是修改、删除和插入完成后,我们想重新刷新一下界面看到更新的数据。但是这种设计的副作用是进入详细信息界面后,从图12-19中②号图片的“备忘录”按钮再回到①号界面时,也会发起网络请求,如果数据量很大,界面就会比较“卡”,这是没有必要的。

有没有一种方法在用户需要的时候由自己刷新呢?在主界面视图(①号图)的导航栏中,我们已经有两个按钮——Edit和+按钮,不能再放置按钮了。现在有一种交互方式,向下拉动表视图可以触发刷新动作,而iOS 6提供了这种刷新控件。



图12-19 查看MyNotes应用的详细信息

图12-20所示的是iOS 6中的下拉刷新，有点像是在拉“胶皮糖”，当“胶皮糖”拉断后，会出现等待指示器。



图12-20 iOS下拉刷新

在iOS 6中，我们在UITableViewController中添加了refreshControl属性，这个属性保持了UIRefreshControl的一个对象指针。UIRefreshControl就是iOS 6为表视图实现下拉刷新而提供的。目前，UIRefreshControl类只能用于表视图界面，而不能用于其他视图。通过下拉刷新布局等问题可以不必考虑，UITableViewController会将其自动放置于表视图中。

在主视图控制器MasterViewController.m中删除viewWillAppear:方法的代码，具体如下：

```
-(void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:YES];
    action = ACTION_QUERY;
    [self startRequest];
}
```

修改主视图控制器MasterViewController.m中的viewDidLoad:方法，注意加粗部分：

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

```

self.navigationItem.leftBarButtonItem = self.editButtonItem;

self.detailViewController = (DetailViewController *)
    [[self.splitViewController.viewControllers lastObject] topViewController];

//查询请求数据
action = ACTION_QUERY;
[self startRequest];

//初始化UIRefreshControl
UIRefreshControl *rc = [[UIRefreshControl alloc] init];
rc.attributedTitle = [[NSAttributedString alloc] initWithString:@"下拉刷新"];
[rc addTarget:self action:@selector(refreshTableView)
    forControlEvents:UIControlEventValueChanged];
self.refreshControl = rc;
    ①
    ②
    ③
    ④
}

```

在上述代码中,第①行代码用于构造UIRefreshControl对象,第②行代码用于设置它的attributedTitle属性,该属性用于为下拉控件显示标题文本。第③行代码为刷新控件添加UIControlEventValueChanged事件处理机制,其中refreshTableView是UIControlEventValueChanged事件的处理方法。第④行代码用于设置视图的refreshControl属性,这里把刚刚创建的UIRefreshControl对象赋值给该属性。refreshTableView方法的代码如下:

```

-(void) refreshTableView
{
    if (self.refreshControl.refreshing) {
        self.refreshControl.attributedTitle = [[NSAttributedString
            alloc] initWithString:@"加载中..."];
        // 查询请求数据
        action = ACTION_QUERY;
        [self startRequest];
    }
}
    ①
    ②
    ③

```

在上述代码中,第①行代码通过控件的refreshing属性判断控件是否还处于刷新状态。刷新状态的图标是我们常见的等待指示器,而显示的文字“加载中...”,是通过第②行代码设置控件的attributedTitle属性实现的。接下来,应该是网络请求操作,如第③行代码所示。

由于异步请求成功返回之后,需要回调reloadView:方法,在这个方法中我们需要停止刷新控件。修改主视图控制器MasterViewController.m中的reloadView:方法,注意加粗部分:

```

-(void)reloadView:(NSDictionary*)res
{
    [self.refreshControl endRefreshing];
    self.refreshControl.attributedTitle = [[NSAttributedString alloc] initWithString:
        @"下拉刷新"];
    NSNumber *resultCodeObj = [res objectForKey:@"ResultCode"];
    if ([resultCodeObj integerValue] >=0)
    {
        self.listData = [res objectForKey:@"Record"];
        [self.tableView reloadData];
    } else {
        NSString *errorStr = [resultCodeObj errorMessage];
        UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"错误信息"
            message:errorStr
            delegate:nil
            cancelButtonTitle:@"OK"
            otherButtonTitles: nil];
        [alertView show];
    }
}
    ①
    ②

```

其中第①行代码调用刷新控件的endRefreshing方法停止刷新。第②行代码设置显示文本为“下拉刷新”。此时我们就将刷新控件添加到表视图的主视图控制器中了，大家可以测试一下看看这种用户体验是不是很好呢！

12.5.2 使用等待指示器控件

当应用请求网络资源时，请求的数据要过一会才能返回，这段时间我们可以给用户看点东西，消除他们的心理等待时间，给用户更好的体验。这种控件叫做等待指示器。iOS提供了两种等待指示器：等待指示器控件（UIActivityIndicatorView）和网络等待指示器。在这一节中，我们先介绍等待指示器控件。

等待指示器控件用起来比较灵活。从技术角度说，等待指示器控件可以放置在视图中。当然，从设计规范上讲，等待指示器控件应该放置在工具栏、导航栏以及弹出的对话框中，请求结束应该消失。

那么，在MyNotes应用的主视图界面中，等待指示器控件应该放在哪里呢？如图12-20所示，MyNotes应用有一个导航栏，只能将等待指示器控件放在图12-21所示的3个位置。



图12-21 等待指示器控件放置的位置

由于这个应用左右两个位置已经被Edit和+按钮占据了，所以等待指示器控件只能放置在中间（②号位置）。下面我们将等待指示器控件添加到MyNotes应用中。修改主视图控制器MasterViewController.m中的viewDidLoad方法，注意加粗部分：

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

    self.navigationItem.leftBarButtonItem = self.editButtonItem;
    self.detailViewController = (DetailViewController *)
        [[self.splitViewController.viewControllers lastObject] topViewController];

    //查询请求数据
    action = ACTION_QUERY;
    [self startRequest];
    [self showActivityIndicatorViewInNavigationBar];
}
```

①

```
//初始化UIRefreshControl
UIRefreshControl *rc = [[UIRefreshControl alloc] init];
rc.attributedTitle = [[NSAttributedString alloc] initWithString:@"下拉刷新"];
[rc addTarget:self action:@selector(refreshTableView)
  forControlEvents:UIControlEventValueChanged];
self.refreshControl = rc;
}
```

在上述代码中，第①行代码调用showActivityIndicatorInViewInNavigationItem方法，它是我们自己定义的方法，其代码如下：

```
//在导航栏中显示等待对话框
-(void) showActivityIndicatorInViewInNavigationItem
{
    UIActivityIndicatorView *aiview = [[UIActivityIndicatorView alloc]
        initWithActivityIndicatorStyle:UIActivityIndicatorViewStyleWhite];           ①
    self.navigationItem.titleView = aiview;                                         ②
    [aiview startAnimating];                                                         ③
    self.navigationItem.prompt = @"数据加载中...";                                  ④
}
```

在上述代码中，第①行代码用于创建等待指示器控件。我们也可以通过在Interface Builder中使用故事板或xib文件创建等待指示器控件，并设置它的属性。在第①行代码中，我们设置等待指示器的样式是UIActivityIndicatorViewStyleWhite。第②行代码设置导航栏项目的titleView属性，该属性的位置如图12-22所示。注意，设置titleView属性之后，就不能显示title属性了。title属性如图12-23所示，它与titleView属性的位置是重合的、互斥的。导航栏项目的prompt属性常用于提示用户，如第④行代码所示，在数据加载时会提示“数据加载中...”。

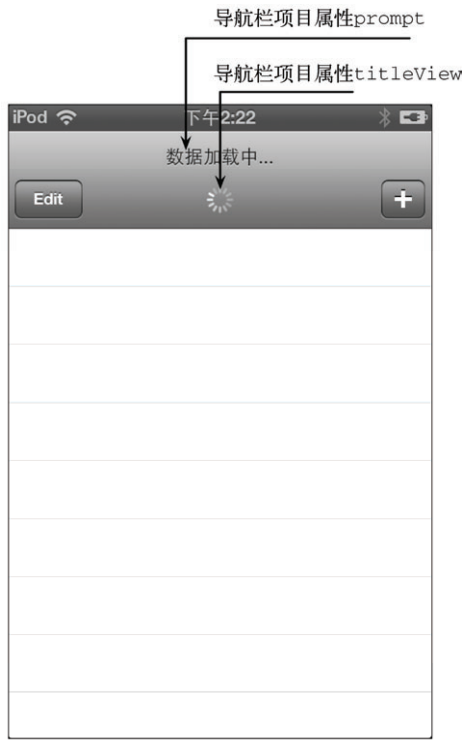


图12-22 导航栏项目的prompt和titleView属性



图12-23 导航栏项目的title属性

等待指示器控件有4个重要的方法和属性，具体如下所示。

❑ **startAnimating**方法。用于开始动画，即旋转起来，如第③行代码所示。

❑ **stopAnimating**方法。用于停止动画，即停止旋转。

❑ **isAnimating**方法。判断是否在旋转。

❑ **hidesWhenStopped**属性。它是布尔值，用于设置控件停止时是否隐藏。

当接收请求时，应该停止等待指示器的“旋转”，这需要在reloadView:方法中停止，其代码如下：

```
//重新加载表视图
-(void)reloadView:(NSDictionary*)res
{
    if (self.refreshControl) {
        [self.refreshControl endRefreshing];
        self.refreshControl.attributedTitle = [[NSAttributedString alloc] initWithString:@"下拉刷新"];
    }

    //停止等待指示器，恢复导航栏
    self.navigationItem.titleView = nil;
    self.navigationItem.prompt = nil;

    .....
}
```

停止指示器控件本应该调用stopAnimating方法，但放在导航栏项目中的等待指示器控件与其有所不同。更重要的是，要移除这个控件，让原来的title内容显示出来，可以使用self.navigationItem.titleView = nil这条语句。

12.5.3 使用网络等待指示器

在介绍ASIHTTPRequest框架时，我们已提到过网络等待指示器，它在状态栏中以经典旋转小图标的形式出现。

它使用UIApplication类的networkActivityIndicatorVisible属性（它是布尔值）设置。由于UIApplication采用单例设计模式，所以可以在程序的任何地方使用[UIApplication sharedApplication]方法调用获得的UIApplication对象。

由于使用ASIHTTPRequest框架时内置了网络等待指示器的功能，所以只要有网络请求，都会出现网络等待指示器图标。如果我们不使用其他框架，而直接使用NSURLRequest时，需要自己添加代码。下面我们为MyNotes应用添加网络等待指示器。修改主视图控制器MasterViewController.m中的startRequest方法，注意加粗部分：

```
-(void)startRequest {

    NSString *strURL = [[NSString alloc] initWithFormat:
        @"http://iosbook1.com/service/mynotes/websevice.php"];

    NSURL *url = [NSURL URLWithString:[strURL URLEncodedString]];

    NSString *post;
    if (action == ACTION_QUERY) { //查询处理
        post = [NSString stringWithFormat:@"email=%@&type=%@&action=%@",
            @"", @"JSON", @"query"];
    } else if (action == ACTION_REMOVE) { //删除处理

        NSIndexPath *indexPath = [self.tableView indexPathForSelectedRow];
        NSMutableDictionary* dict = self.listData[indexPath.row];
        post = [NSString stringWithFormat:@"email=%@&type=%@&action=%@&id=%@",
            @"", @"JSON", @"remove",
            [dict objectForKey:@"ID"]];
    }
}
```

```

NSData *postData = [post dataUsingEncoding:NSUTF8StringEncoding];

NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:url];
[request setHTTPMethod:@"POST"];
[request setHTTPBody:postData];

[UIApplication sharedApplication].networkActivityIndicatorVisible = YES;           ①

NSURLConnection *connection = [[NSURLConnection alloc]
                                initWithRequest:request delegate:self];

if (connection) {
    _datas = [NSMutableData new];
}
}

```

在上述代码中，第①行代码设置networkActivityIndicatorVisible属性为YES，即会在状态栏中显示网络等待指示器图标。另外，我们需要在请求结束时停止，这需要在reloadView:方法中添加代码，具体如下：

```

// 重新加载表视图
- (void)reloadView: (NSDictionary*)res
{
    if (self.refreshControl) {
        [self.refreshControl endRefreshing];
        self.refreshControl.attributedTitle = [[NSAttributedString alloc]
                                                initWithString:@"下拉刷新"];
    }
    [UIApplication sharedApplication].networkActivityIndicatorVisible = NO;           ①
    .....
}

```

其中第①行代码用于停止网络等待指示器，并且其图标会在状态栏中消失。

12.6 小结

通过对本章的学习，我们了解了数据交换格式，其中XML和JSON是主要的方式。此外，我们还重点介绍了Web Service的访问，其中REST Web Service是学习的重点。最后，我们还在访问Web Service的过程中介绍了ASIHTTPRequest框架。

在这个时代，我们已经越来越离不开手机了。手机上越来越多的应用是基于地图的，而大部分地图又使用了定位服务。手机的优势就是能够到处移动，我们往往需要知道自己所在的位置，然后再查询自己周围的饭店、影院以及交通路线等。查找自己的位置时，可以使用GPS等方式提供的定位服务。找到饭店、影院和交通路线等信息时，再通过地图标注出来。

在iOS中，定位服务与地图应用是完全不同的两套API，但是它们结合很紧密，因此把它们放在一章中介绍给大家。

13.1 定位服务

iOS设备提供了3种不同的途径进行定位，具体如下所示。

- ❑ **Wi-Fi**。通过Wi-Fi路由器的地理位置信息查询，比较省电。iPhone、iPod touch和iPad都可以采用这种方式定位。
- ❑ **蜂窝式移动电话基站**。通过移动运营商基站定位。只有iPhone、3G版本的iPod touch和iPad可以采用这种方式定位。
- ❑ **GPS卫星**。通过GPS卫星位置定位，这种方式最为准确，但是耗电量大，不能遮挡。iPhone、iPod touch和iPad都可以采用这种方式定位。

在对定位服务编程时，iOS不像Android系统可以指定采用哪种途径进行定位。iOS的API把底层这些细节屏蔽掉了，开发人员和用户并不知道现在设备是采用哪种方式进行定位的，iOS系统会根据设备的情况和周围的环境采用一套最佳的解决方案。也就是说，如果能够接收GPS信息，那么设备优先采用GPS定位，否则采用Wi-Fi或蜂窝基站定位。在Wi-Fi和蜂窝基站之间，优先使用Wi-Fi，如果无法连接Wi-Fi才使用蜂窝基站定位。

提示 GPS（全球定位系统）是20世纪70年代由美国陆、海、空三军联合研制的新一代空间卫星导航定位系统，其主要目的是为陆、海、空三大领域提供实时、全天候和全球性的导航服务，并用于情报收集、核爆监测和应急通信等一些军事目的。经过20余年的研究实验，耗资300亿美元，到1994年3月，全球覆盖率高达98%的24颗GPS卫星已布设完成。到2012年10月26日，中国也成功发射16颗北斗导航卫星。这些导航卫星分为军用频道和民用频道，前者是加密的，定位精度极高，而後者的定位精度要低一些。

总体来说，GPS定位的优点是准确、覆盖面广，缺点是不能被遮挡（例如在建筑物里面收不到GPS卫星信号）、GPS开启后比较耗电。蜂窝基站不仅误差比较大，而且会耗费用户流量。而Wi-Fi定位是最经济实惠的。

13.1.1 定位服务编程

在iOS 6中，定位服务没有太大的变化，主要使用Core Location框架，定位时主要使用CLLocationManager、

CLLocationManagerDelegate和CLLocation这3个类，下面简要介绍一下它们。

- ❑ **CLLocationManager**。用于定位服务管理类，它能够给我们提供位置信息和高度信息，也可以监控设备进入或离开某个区域，还可以获得设备的运行方向等。
- ❑ **CLLocationManagerDelegate**。它是CLLocationManager类的委托协议。
- ❑ **CLLocation**。该类封装了位置和高度信息。

在定位服务的应用中，第一次请求位置信息时，系统会提示用户是否允许开启定位服务。如图13-1所示，用户所在的位置是比较私密的信息，应用获取这些信息时，用户是有知情权和否定权的。如果应用在用户不知情的情况下获得其位置信息，这在某些国家是违法的。



图13-1 允许开启定位服务

如果用户“不允许”，定位服务就无法获得位置信息了。如果想改变这些设置，可以在系统“设置”应用中开启或关闭，如图13-2所示。

如图13-2右图所示，我们可以关闭所有的定位服务，此时只需关闭最上面的“定位服务”开关控件就可以了。当然，也可以关闭或开启下面的具体应用。



图13-2 开启或关闭定位服务

如图13-3所示，在应用启动进入界面时，会获得位置信息，并显示在对应的文本框中。如果设备位置发生变化，也会重新获取位置信息，并更新对应的文本框。

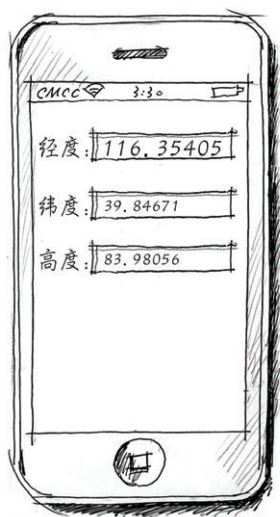


图13-3 定位服务案例

首先，为工程引入Core Location框架，具体步骤是选择工程中的TARGETS→WhereAml→Build Phases→Link Binary With Libraries，选择右下角的+按钮，打开“选择要添加的框架和库”对话框，如图13-4所示。

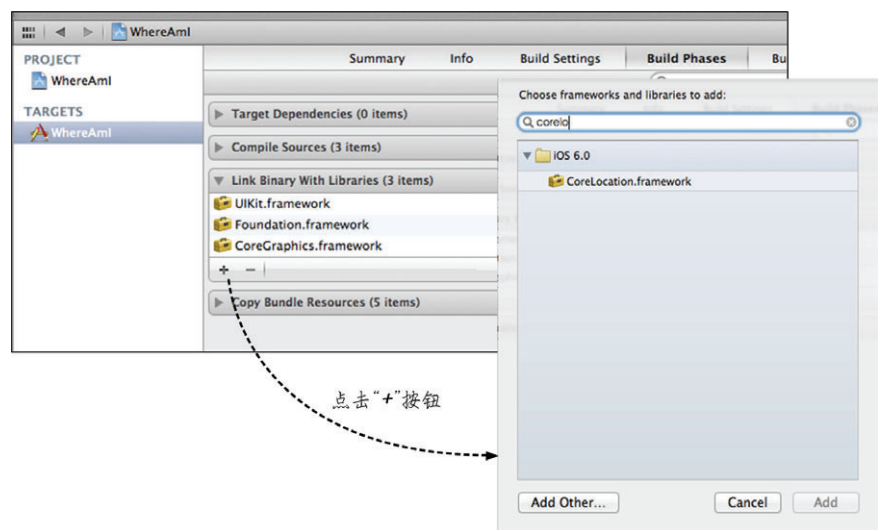


图13-4 添加框架到工程

在该对话框中选择CoreLocation.framework，点击Add按钮完成添加。下面我们直接看看实现代码，其中主要代码是在视图控制器ViewController中编写的。ViewController.h中的代码如下：

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

@interface ViewController : UIViewController <CLLocationManagerDelegate>

// 经度
@property (weak, nonatomic) IBOutlet UITextField *txtLng;
```

```

//纬度
@property (weak, nonatomic) IBOutlet UITextField *txtLat;
//高度
@property (weak, nonatomic) IBOutlet UITextField *txtAlt;

@property (nonatomic, strong) CLLocationManager *locationManager;

@end

```

在上述代码中,我们首先引入了CoreLocation/CoreLocation.h和CoreLocation/CLLocationManagerDelegate.h这两个头文件,然后在定义ViewController时声明了CLLocationManagerDelegate协议。此外,我们还定义了CLLocationManager *locationManager属性。

在ViewController.m中, viewDidLoad方法的代码如下:

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    //初始化定位服务管理对象
    _locationManager = [[CLLocationManager alloc] init];
    _locationManager.delegate = self;
    _locationManager.desiredAccuracy = kCLLocationAccuracyBest;           ①
    _locationManager.distanceFilter = 1000.0f;                          ②
}

```

在上述代码中,我们主要对CLLocationManager的成员变量_locationManager进行了初始化。首先,使用[[CLLocationManager alloc] init]语句实例化CLLocationManager对象,然后使用_locationManager.delegate = self语句设置定位服务委托为self。第①行代码设置desiredAccuracy属性,它是一个非常重要的属性,其取值有6个常量,具体如下所示。

- ❑ **kCLLocationAccuracyNearestTenMeters**。精确到10米。
- ❑ **kCLLocationAccuracyHundredMeters**。精确到100米。
- ❑ **kCLLocationAccuracyKilometer**。精确到1000米。
- ❑ **kCLLocationAccuracyThreeKilometers**。精确到3000米。
- ❑ **kCLLocationAccuracyBest**。设备使用电池供电时最高的精度。
- ❑ **kCLLocationAccuracyBestForNavigation**。导航情况下最高的精度,一般有外接电源时才能使用。

精度越高,请求获得位置信息的时间就越短,这就意味着设备越耗电,因此一个应用应该选择适合它的精度。如果你的应用是一个车载导航应用, kCLLocationAccuracyBestForNavigation是比较好的选择,你可以使用汽车上的电瓶为设备供电。如果你的应用是为徒步旅行者提供的导航应用, kCLLocationAccuracyHundredMeters是一个不错的选择。

第②行代码设置distanceFilter属性,它是距离过滤器,定义了设备移动后获得位置信息的最小距离,单位是米,本例设置为1000米。

初始化CLLocationManager类后,需要使用startUpdatingLocation方法开始定位服务,该方法定义在ViewController.m的viewWillAppear:方法中。viewWillAppear:方法的代码如下:

```

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    //开始定位
    [_locationManager startUpdatingLocation];
}

```

调用startUpdatingLocation方法时,就会开启定位服务。根据设定的条件,它不断请求回调新的位置信息。因此,开启这个方法一定要慎重,在视图控制器的声明周期方法viewWillAppear:中使用这个方法是最合适的。与开启服务对应的方法是stopUpdatingLocation方法,它是在视图控制器的viewWillDisappear:方

法中调用的。viewWillDisappear:方法的代码如下:

```
- (void)viewWillDisappear:(BOOL)animated
{
    [super viewWillDisappear:animated];
    //停止定位
    [_locationManager stopUpdatingLocation];
}
```

这个方法在视图消失(应用退到后台)时调用,能够保证最及时地关闭定位服务。在iOS 6中,请求有所变化,定位服务应用退入后台后,可以延迟更新位置信息,这可以通过allowDeferredLocationUpdatesUntilTraveled:timeout:方法实现。要关闭延迟更新,可以使用disallowDeferredLocationUpdates方法实现。此外,在iOS 6中,新增了pausesLocationUpdatesAutomatically属性,它能设定自动暂停位置更新,而把定位服务的开启和暂停管理权交给系统,这样会更加合理和简单。

一旦定位服务开启,并设置好CLLocationManager委托属性delegate后,当用户设备移动到到达过滤距离时,就会回调委托方法。与定位服务有关的方法有如下两个。

❑ **locationManager:didUpdateLocations:**。定位成功。这是iOS 6中新增的方法,替代了之前的locationManager:didUpdateToLocation:fromLocation:方法。

❑ **locationManager:didFailWithError:**。定位失败。

实现CLLocationManager委托的代码如下:

```
#pragma mark Core Location委托方法用于实现位置的更新
- (void)locationManager:(CLLocationManager *)manager didUpdateLocations:
    (NSArray *)locations
{
    CLLocation * currLocation = [locations lastObject];
    _txtLat.text = [NSString stringWithFormat:@"%3.5f",
        currLocation.coordinate.latitude];
    _txtLng.text = [NSString stringWithFormat:@"%3.5f",
        currLocation.coordinate.longitude];
    _txtAlt.text = [NSString stringWithFormat:@"%3.5f",currLocation.altitude];

}

- (void)locationManager:(CLLocationManager *)manager didFailWithError:(NSError *)error
{
    NSLog(@"error: %@",error);
}
```

在locationManager:didUpdateLocations:方法中,参数locations是位置变化的集合,它按照时间变化的顺序存放。如果想获得当前设备的位置,可以使用第①行中的[locations lastObject]语句获得集合中的最后一个元素,它就是设备的当前位置了。从集合中返回的对象类型是CLLocation,CLLocation封装了位置、高度等信息。在上面的代码中,我们使用了它的两个属性altitude和coordinate,其中前者是高度值,后者是封装经度和纬度的结构体CLLocationCoordinate2D。CLLocationCoordinate2D的定义如下:

```
typedef struct {
    CLLocationDegrees latitude;        // 纬度
    CLLocationDegrees longitude;      // 经度
} CLLocationCoordinate2D;
```

其中latitude为纬度信息,longitude为经度信息,它们都是CLLocationDegrees类型。CLLocationDegrees是使用typedef定义的double类型。

第②行代码中的currLocation.coordinate.latitude表达式用于获得设备当前的纬度,第③行代码中的currLocation.coordinate.longitude表达式用于获得设备当前的经度,而第④行代码中的currLocation.altitude表达式用于获得高度。

13.1.2 地理信息反编码

在上一节的案例中，我们知道了经度和纬度，那又能怎么样呢？一般人很难知道这些数字（比如(114.15818, 22.28468)）代表的是什么地方，地理信息反编码就是用来解决这个问题的，它通过地理坐标返回某个地点的相关文字描述信息。这些描述信息封装在CLPlacemark类中，我们把这个类叫做“地标”类。“地标”类有很多属性，下面是其中主要的几个文字描述相关属性。

- ❑ **addressDictionary**。地址信息的字典，包含一些键值对，其中的键在Address Book framework（地址簿框架）中定义。
- ❑ **ISOcountryCode**。ISO国家代号。
- ❑ **Country**。国家信息。
- ❑ **postalCode**。邮政编码。
- ❑ **administrativeArea**。行政区域信息。
- ❑ **subAdministrativeArea**。行政区域附加信息。
- ❑ **locality**。指定城市信息。
- ❑ **subLocality**。指定城市信息附加信息。
- ❑ **thoroughfare**。指定街道级别信息。
- ❑ **subThoroughfare**。指定街道级别的附加信息。

地理信息反编码使用CLGeocoder类实现，这个类能够实现地理坐标与地理文字描述信息之间的转换。CLGeocoder类中进行地理信息反编码的方法是：

```
- (void)reverseGeocodeLocation:(CLLocation *)location completionHandler:
    (CLGeocodeCompletionHandler)completionHandler
```

其中参数location是要定位的地理位置对象，completionHandler指定了一个代码块，用于地理信息反编码之后的回调。

下面我们通过图13-5所示的案例介绍一下地理信息反编码，这个案例在上一节案例的基础上添加了地理信息反编码功能。在界面上单击“地理信息反编码”按钮，反编码的结果就会显示在下面的TextView控件中。

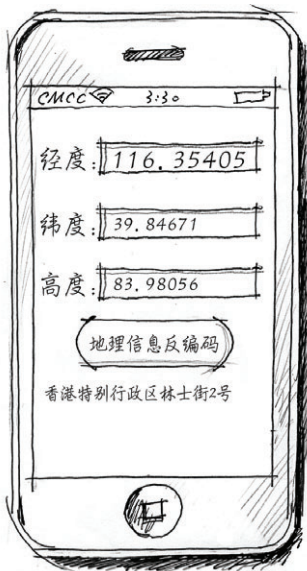


图13-5 地理信息反编码案例

UI设计部分我们不再介绍。下面我们直接看看实现代码，其中主要是在视图控制器ViewController中添加按钮事件处理方法reverseGeocode:，该方法的代码如下：

```

- (IBAction)reverseGeocode:(id)sender {

    CLGeocoder *geocoder = [[CLGeocoder alloc] init];

    [geocoder reverseGeocodeLocation:_currLocation
     completionHandler:^(NSArray *placemarks, NSError *error) {           ①

        if ([placemarks count] > 0) {                                       ②

            CLPlacemark *placemark = placemarks[0];                       ③

            NSDictionary *addressDictionary = placemark.addressDictionary;  ④

            NSString *address = [addressDictionary objectForKey:(NSString *)
                                kABPersonAddressStreetKey];                ⑤
            address = address == nil ? @"": address;

            NSString *state = [addressDictionary objectForKey:(NSString *)
                               kABPersonAddressStateKey];                  ⑥
            state = state == nil ? @"": state;

            NSString *city = [addressDictionary objectForKey:(NSString *)
                              kABPersonAddressCityKey];                    ⑦
            city = city == nil ? @"": city;

            _txtView.text = [NSString stringWithFormat:@"% %@ \n %@ \n %@",state,
                                address,city];

        }

    }];
}

```

在上述代码中，第①行代码调用reverseGeocodeLocation:completionHandler:方法进行地理信息编码，其中_currLocation是CLLocation类型的成员变量，它是在委托方法locationManager: didUpdateLocations:中获得的。在回调代码completionHandler:^(NSArray *placemarks, NSError *error)中，参数placemarks是反编码成功的地标集合。事实上，一个地理坐标点并不是完全意义上的几个圆点，它泛指一个范围，因此在这个范围中，有可能有多种不同的描述信息，这些信息被放在地标集合中。另一个参数error描述了出错信息，如果error非空，则说明反编码失败。

第②行代码判断地标集合大于0，即反编码成功并且有返回的描述信息情况。第③行代码从地标集合中取出一个地标对象，如果用户需要查看所有的地标信息，循环遍历出来就可以了。第④行代码从地标对象取出addressDictionary属性，它返回字典类型对象。第⑤行代码使用kABPersonAddressStreetKey键从字典中取出地标的街道信息。第⑥行代码使用kABPersonAddressStateKey键从字典中取出地标的所在州、省等信息。第⑦行代码使用kABPersonAddressCityKey键从字典中取出地标的所在市等信息。这些键是AddressBook框架中定义的常量，因此我们需要为工程添加AddressBook.framework，具体步骤参考图13-4。此外，还要在ViewController.h中添加头文件引入代码：

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

13.1.3 地理信息编码查询

地理信息编码查询与反编码刚好相反，它通过地理信息的文字描述查询出相关的地理坐标，这种查询结果也是一个集合。例如，如果我们去查询“城南”，会找出很多地点，因为这个关键词太普遍了，这种情况下我们可以指定区域范围查询。地理信息编码查询也采用CLGeocoder类，其中有关地理信息编码的方法有如下几个。

- ❑ **geocodeAddressDictionary:completionHandler:**。通过指定一个地址信息字典对象参数进行查询。
- ❑ **geocodeAddressString:completionHandler:**。通过指定一个地址字符串参数进行查询。
- ❑ **geocodeAddressString:inRegion:completionHandler:**。通过指定地址字符串和查询的范围作为参数进行查询，其中inRegion部分的参数用于指定查询范围，它是CLRegion类型。

下面我们通过图13-6所示的案例介绍一下地理信息编码查询，在“输入查询地点关键字”文本框中输入关键字，多个关键字可以用逗号分隔。单击“地理信息编码查询”按钮查询出结果后，取出第一个地址显示在界面中。

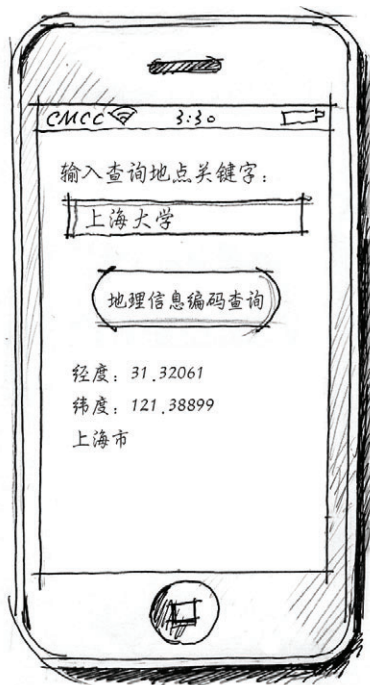


图13-6 地理信息编码查询案例

我们直接看看实现代码，其中主要的代码是视图控制器ViewController中添加按钮事件处理的方法geocodeQuery::

```

- (IBAction)geocodeQuery:(id)sender {

    if (_txtQueryKey.text == nil || [_txtQueryKey.text length] == 0) {
        return;
    }

    CLGeocoder *geocoder = [[CLGeocoder alloc] init];
    [geocoder geocodeAddressString:_txtQueryKey.text completionHandler:^(
        NSArray *placemarks, NSError *error) {                                ①
        NSLog(@"查询记录数: %i", [placemarks count]);
        if ([placemarks count] > 0) {
            CLPlacemark* placemark = [placemarks objectAtIndex:0];
            CLLocationCoordinate2D coordinate = placemark.location.coordinate;    ②
            NSString* strCoordinate = [NSString stringWithFormat:@"经度:%3.5f\n纬度:%3.5f", coordinate.latitude, coordinate.longitude];
            NSDictionary *addressDictionary = placemark.addressDictionary;

            NSString *address = [addressDictionary
                                objectForKey:(NSString *)kABPersonAddressStreetKey];
            address = address == nil ? @"": address;
        }
    }];
}

```

```

NSString *state = [addressDictionary
    objectForKey:(NSString *)kABPersonAddressStateKey];
state = state == nil ? @"": state;

NSString *city = [addressDictionary
    objectForKey:(NSString *)kABPersonAddressCityKey];
city = city == nil ? @"": city;

_txtView.text = [NSString stringWithFormat:@"%@@ \n %@ \n%@ \n%@",
    strCoordinate,state, address,city];

//关闭键盘
[_txtQueryKey resignFirstResponder];
}
}];
}

```

在上述代码中，第①行代码调用`geocodeAddressString:completionHandler:`方法进行地理信息编码查询，这里没有指定查询范围。查询的返回结果`placemarks`参数是`NSArray`类型集合，取出集合中第一个地标对象。在第②行代码中，我们使用`placemark.location.coordinate`属性获得地标的经纬度信息。

这样我们就可以进行查询了。如果想指定查询范围，可以使用`geocodeAddressString:inRegion:completionHandler:`方法，相关代码如下：

```

CLRegion * region = [[CLRegion alloc] initWithCenter:_currLocation.coordinate
    radius:1000.0f
    identifier:@"GeocodeRegion"];
CLGeocoder *geocoder = [[CLGeocoder alloc] init];
[geocoder geocodeAddressString:_txtQueryKey.text inRegion:region
    completionHandler:^(NSArray *placemarks, NSError *error) {
    .....
}];

```

在上述代码中，第①行代码用于构造一个`CLRegion`对象，它封装了一个地理区域类，其构造方法为`initWithCenter:radius:identifier:`，其中`initWithCenter`参数指定区域中心点，类型是`CLLocationCoordinate2D`，`radius`参数指定区域半径的单位为米，`identifier`参数为区域指定一个标识，保证在你的应用中是唯一的，这个参数不能为空。第②行代码执行查询。

13.1.4 关于定位服务的测试

一般情况下，定位服务应用的测试和运行有两个选择：模拟器和设备。原则上，我们先通过模拟器，然后再使用设备测试。由于定位服务的特点，使用设备测试时我们需要到现场进行测试，所以有的时候有一定的局限性。因此，使用模拟器测试有的时候是不可替代的。

在Xcode早期版本中，模拟器是不能模拟位置信息的变化，请求获取位置信息只是固定苹果公司总部地址。而现在的Xcode版本预先设置了几个地址，我们可以模拟改变位置。如果想让模拟器一开始运行的时候就能够获得模拟数据，可以在启动参数中设置。首先，在Xcode工具的左上角编辑应用的Scheme，如图13-7所示。

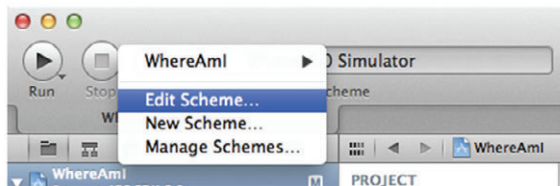


图13-7 编辑Scheme

选择Edit Scheme菜单后,弹出如图13-8所示的对话框,从中选择Run WhereAmI.app→Options,在Core Location项目中选中Allow Location Simulation复选框,然后在下面的Default Location下拉框中选择你感兴趣的城市。

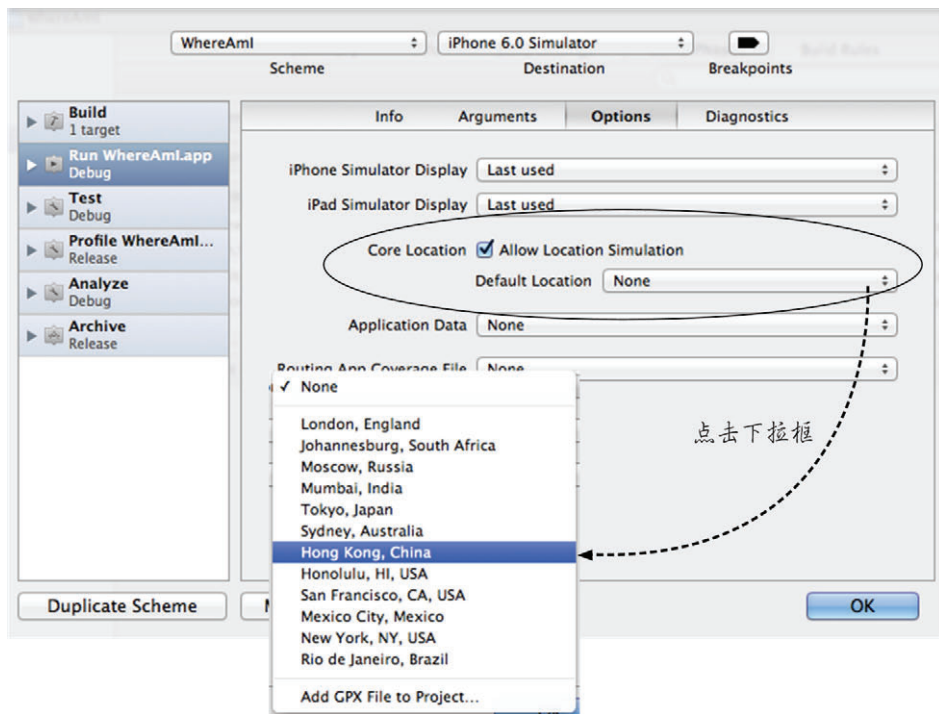


图13-8 设置启动参数

这样应用启动时,就会模拟定位到你选择的城市了。如果列表中没有我们需要的地点,可以使用最下面的Add GPX File to Project菜单项为工程添加一个GPX^①文件。下面是GPX文件的内容:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<gpx xmlns="http://www.topografix.com/GPX/1/1"
creator="MyGeoPosition.com" version="1.1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.topografix.com/GPX/1/1
http://www.topografix.com/GPX/1/1/gpx.xsd">
<wpt lat="40.002240" lon="116.323328">
<name>中国北京 东城区北京站东街北京 邮政编码: 100005</name>
<src>MyGeoPosition.com</src>
<link>http://mygeoposition.com</link>
</wpt>
</gpx>
```

<wpt>标签中的lat属性设置纬度,lon属性设置经度。自己手写这个文件还是比较麻烦的,我一般使用www.mygeoposition.com网站提供的GPX工具(如图13-9所示)工具生成。这个网站免费提供地理信息编码和反编码、生成KML和GPX文件等服务。

① GPX (GPS eXchange Format, GPS交换格式) 是一个XML格式,是为应用软件设计的通用GPS数据格式。——引自于维基百科
<http://zh.wikipedia.org/wiki/Gpx>



图13-9 使用www.mygeoposition.com提供GPX工具


如果我们想找到“清华大学”的地理坐标，可以在地图上找到地点，然后点击鼠标，会出现一个描述该地点经纬度的“气泡”，我们再点击GPX标签，就会生成如图13-10所示的GPX文件，可以直接把这些内容复制出来，也可以点击“下载kml文件”按钮，将GPX文件下载到本地。

提示 在图13-10中选择GPX标签后，下面的按钮标签是“下载kml文件”，这应该是本站开发者的笔误，我猜测应该是“下载gpx文件”才对，但是这不影响我们使用它下载GPX文件。



图13-10 生成GPX文件

得到GPX文件后，可以通过图13-8所示的Add GPX File to Project菜单项将它添加到Xcode工程中，此时在菜单中就会出现GPX文件了。如图13-11所示，如果我添加的文件名是test.gpx，则在菜单中出现test菜单项，选择test即可使用这个模拟坐标数据了。

如果在应用启动参数中没有设置初始的模拟位置数据，我们还可以在运行之后设置。在调试工具栏中选择模拟定位按钮，即可选择模拟位置，如图13-12所示。

Xcode中的模拟器还提供了连续位置变化测试能力。如果想开发导航应用，这个功能对我们有很大的帮助。此外，模拟器有几个固定的模式，可以发出连续变化的位置数据。打开模拟菜单的“调试”→“位置”，可以发

现共有7个菜单项，如图13-13所示。

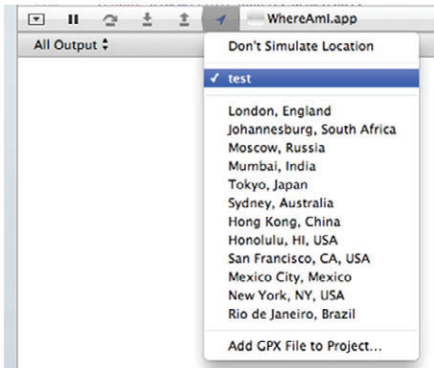


图13-11 添加GPX文件

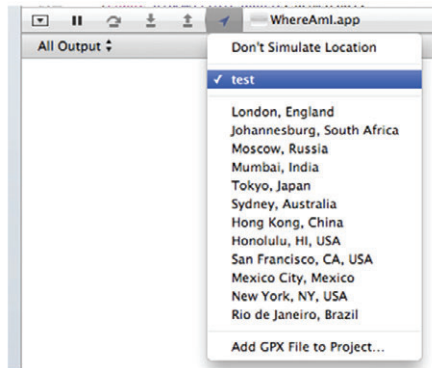


图13-12 设置模拟定位数据



图13-13 模拟器位置菜单

其中后面3个都能发出连续的位置变化数据，它们的起始点从苹果公司总部开始，按照一个固定的线路运动，这三者的区别是City Bicycle Ride是最慢的，City Run要快一些，Freeway Drive最快。

13.2 使用 iOS 6 苹果地图

在iOS 6中，苹果自己的地图代替了谷歌地图，但是API编程接口没有太大的变化，所以开发人员不需要再学习很多新东西就能开发地图应用。因此，本节介绍的内容也同样适用于在iOS 5上开发地图应用。

在iOS应用程序中，我们使用Map Kit API开发地图应用，其核心是MKMapView类。下面我们从显示地图、添加标注和跟踪用户位置变化这3个方面来简要介绍一下Map Kit API。

13.2.1 显示地图

在Map Kit API中，显示地图的视图是MKMapView，它的委托协议是MKMapViewDelegate。使用Map Kit API时，需要导入MapKit框架。

下面我们通过如图13-14所示的案例介绍一下Map Kit API的用法，在“输入查询地点关键字”文本框中输入关键字，点击“查询”按钮，先进行地理信息编码查询，获得地标信息后，会在地图上标注出来。



图13-14 iOS地图应用案例

首先，请参考图13-4的操作添加框架MapKit.framework，然后在工程中打开MainStoryboard.storyboard的Interface Builder设计，从对象库中拖曳Map View到设计界面中，如图13-15所示。

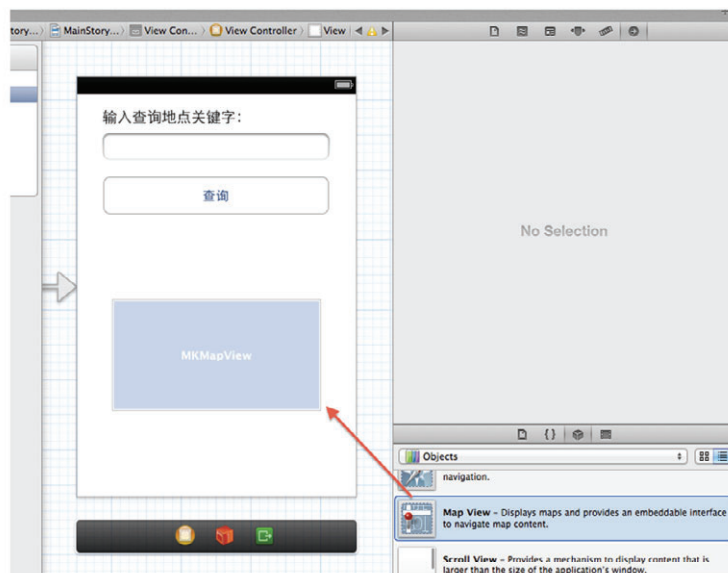


图13-15 在Interface Builder中设计Map View

调整Map View的位置和大小，使其尽可能充满界面下面的空白部分，然后为Map View定义输出口。下面我们看看主视图控制器ViewController.h中的代码：

```
#import <UIKit/UIKit.h>
#import <MapKit/MapKit.h>
#import "MapLocation.h"

@interface ViewController : UIViewController <MKMapViewDelegate>

@property (weak, nonatomic) IBOutlet UITextField *txtQueryKey;

@property (weak, nonatomic) IBOutlet MKMapView *mapView;

- (IBAction)geocodeQuery:(id)sender;

@end
```

由于使用Map Kit API，需要引入头文件。头文件MapLocation.h是我们自己定义的用于描述地图标注点的。在定义ViewController时，还需要声明MKMapViewDelegate协议。txtQueryKey属性是查询关键字的文本框，mapView属性是MKMapView类型，与界面对应。点击“查询”按钮时，触发geocodeQuery:方法，该方法处理查询并在地图上做标注。

下面我们看看ViewController.m中viewDidLoad方法的代码：

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

    _mapView.mapType = MKMapTypeStandard;
    _mapView.delegate = self;
}
```

在viewDidLoad方法中，我们设置了地图的类型，其中共有3种类型，具体如下所示。

- ❑ **MKMapTypeStandard**。标注地图类型，如图13-16所示。
- ❑ **MKMapTypeSatellite**。卫星地图类型。如图13-17所示，在卫星地图中没有街道名称等信息。
- ❑ **MKMapTypeHybrid**。混合地图类型。如图13-18所示，混合地图是在卫星地图上标注出街道等信息。



图13-16 标注地图

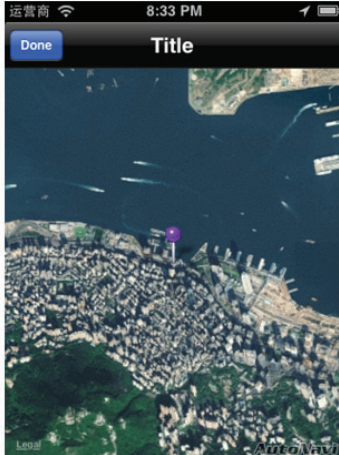


图13-17 卫星地图

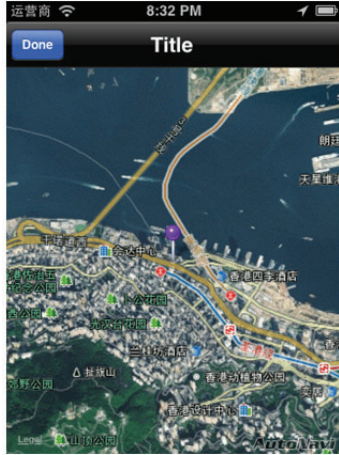


图13-18 混合地图

在viewDidLoad方法中，`_mapView.delegate = self`这行语句用于当前视图控制器赋值给地图视图的delegate属性，这样地图视图在需要的时候就会回调ViewController，如果失败，就回调下面的失败方法：

```
- (void)mapViewDidFailLoadingMap:(MKMapView *)theMapView withError:(NSError *)error {
    NSLog(@"error : %@",[error description]);
}
```

13.2.2 添加标注

如果要在地图视图上添加标注点，需要两个步骤：第一步是触发添加动作，第二步是实现地图委托方法mapView:viewForAnnotation:。

1. 触发添加动作

我们通过“查询”按钮触发添加标注动作，相关代码如下：

```
- (IBAction)geocodeQuery:(id)sender {
    if (_txtQueryKey.text == nil || [_txtQueryKey.text length] == 0) {
        return;
    }

    CLGeocoder *geocoder = [[CLGeocoder alloc] init];
    [geocoder geocodeAddressString:_txtQueryKey.text completionHandler:
     ^(NSArray *placemarks, NSError *error) {

        NSLog(@"查询记录数: %i",[placemarks count]);

        if ([placemarks count] > 0) {
            [_mapView removeAnnotations:_mapView.annotations];
        }

        for (int i = 0; i < [placemarks count]; i++) {

            CLPlacemark* placemark = placemarks[i];
            ①
        }
    }];
}
```


2. 实现地图委托方法mapView:viewForAnnotation:

MKMapViewDelegate委托协议方法mapView:viewForAnnotation:的代码如下:

```

- (MKAnnotationView *) mapView:(MKMapView *)theMapView
  viewForAnnotation:(id <MKAnnotation>) annotation {
    ①

    MKPinAnnotationView *annotationView = (MKPinAnnotationView *)[_mapView
      dequeueReusableAnnotationViewWithIdentifier:@"PIN_ANNOTATION"];
    ②
    if(annotationView == nil) {
    ③
      annotationView = [[MKPinAnnotationView alloc] initWithAnnotation:annotation
        reuseIdentifier:@"PIN_ANNOTATION"];
    ④
    }

    annotationView.pinColor = MKPinAnnotationColorPurple;
    ⑤
    annotationView.animatesDrop = YES;
    ⑥
    annotationView.canShowCallout = YES;
    ⑦

    return annotationView;
}

```

在上述代码中,第①行代码所示的委托方法mapView:viewForAnnotation:在地图视图添加标注时回调。给地图视图添加标注的方法是[mapView addAnnotation:annotation],其中annotation是地图标注对象。

第②~④行代码用于获得地图标注对象MKPinAnnotationView,其中采用了可重用MKPinAnnotationView对象设计。可重用对象为了节约内存,尽可能使用已有对象,减少实例化对象。首先,在第②行代码中,我们使用dequeueReusableAnnotationViewWithIdentifier:方法通过一个可重用标识符PIN_ANNOTATION获得MKPinAnnotationView对象,如果这个对象不存在(第③行代码判断是否存在),则需要使用initWithAnnotation:reuseIdentifier:构造方法创建,其中reuseIdentifier参数是可重用标识符。

第⑤行代码设置大头针标注视图的颜色为紫色。此外,该颜色还可以设置成MKPinAnnotationColorRed(红色)和MKPinAnnotationColorGreen(绿色)。

第⑥行代码说明设置标注视图时,是否以动画效果的形式显示在地图上。第⑦行代码用于在标注点上显示一些附加信息。如果canShowCallout为YES,点击“大头针”头时,会出现一个气泡(如图13-19所示),而气泡中的文字信息封装在MapLocation对象中,其中第一行文字(大一点的文字)保存在title属性中,而第二行文字(小一点的文字)保存在subtitle属性中。



图13-19 canShowCallout设置为YES的情况

在委托方法的最后,返回annotationView标注点视图对象。

最后,我们看看自定义标注类MapLocation。MapLocation.h中的代码如下:

```

#import <MapKit/MapKit.h>

@interface MapLocation : NSObject<MKAnnotation>

@property (nonatomic, readwrite) CLLocationCoordinate2D coordinate;
//街道信息属性

```

```

@property (nonatomic, copy) NSString *streetAddress;
//城市信息属性
@property (nonatomic, copy) NSString *city;
//州、省、市信息
@property (nonatomic, copy) NSString *state;
//邮编
@property (nonatomic, copy) NSString *zip;
//地理坐标
@property (nonatomic, readwrite) CLLocationCoordinate2D coordinate;

@end

```

首先，需要引入头文件，这是因为MKAnnotation协议是包含在该头文件中的。地图标注点类必须实现MKAnnotation协议。MKAnnotation协议需要实现如下所示的两个方法。

- ❑ - (NSString *)title。标注点上的主标题。
- ❑ - (NSString *)subtitle。标注点上的副标题。

MapLocation.m中的代码如下：

```

#import "MapLocation.h"
@implementation MapLocation

- (NSString *)title {
    return @"您的位置!";
}

- (NSString *)subtitle {
    NSMutableString *ret = [NSMutableString new];
    if (_state)
        [ret appendString:_state];
    if (_city)
        [ret appendString:_city];
    if (_city && _state)
        [ret appendString:@" ", "];
    if (_streetAddress && (_city || _state || _zip))
        [ret appendString:@" " • "];
    if (_streetAddress)
        [ret appendString:_streetAddress];
    if (_zip)
        [ret appendFormat:@"", %@, _zip];
    return ret;
}

@end

```

在subtitle方法中，我们将它的属性拼接成字符串返回。这里，我们可以根据自己的需要和习惯拼接在这个字符串的前后。

13.2.3 跟踪用户位置变化

MapKit提供了跟踪用户位置和方向变化的API，这样我们就不用自己编写定位服务代码了。开启地图视图的showsUserLocation属性，并设置方法setUserTrackingMode:就可以了，参考代码如下：

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    if ([CLLocationManager locationServicesEnabled])
    {
        _mapView.mapType = MKMapTypeStandard;
        _mapView.delegate = self;
        _mapView.showsUserLocation = YES;
        [_mapView setUserTrackingMode:MKUserTrackingModeFollow animated:YES];
    }
}

```


其中`_mapView.showsUserLocation = YES`允许跟踪显示用户位置信息。在iOS设备中，显示用户位置的方式是一个发亮的蓝色小圆点，如图13-20所示。

`[_mapView setUserTrackingMode:MKUserTrackingModeFollow animated:YES]`语句设置用户跟踪模式。用户跟踪模式有如下3种。

- ❑ **MKUserTrackingModeNone**。没有用户跟踪模式。
- ❑ **MKUserTrackingModeFollow**。可以跟踪用户的位置变化。
- ❑ **MKUserTrackingModeFollowWithHeading**。可以跟踪用户的位置和方向变化。

然后，我们还需要实现地图视图委托方法`mapView:didUpdateUserLocation:`，它的代码如下：

```
- (void)mapView:(MKMapView *)mapView didUpdateUserLocation:(MKUserLocation *)userLocation
{
    _mapView.centerCoordinate = userLocation.location.coordinate;
}
```

该委托方法在定位服务更新完成用户位置时回调，我们在该方法中重新调整地图的中心点为当前用户的中心点。

这几行代码就可以跟踪用户位置的变化了，这里可以使用模拟器测试一下，运行结果如图13-21所示，其中会有一个小圆圈，它按照我们设定的速度运行。



图13-20 显示用户位置

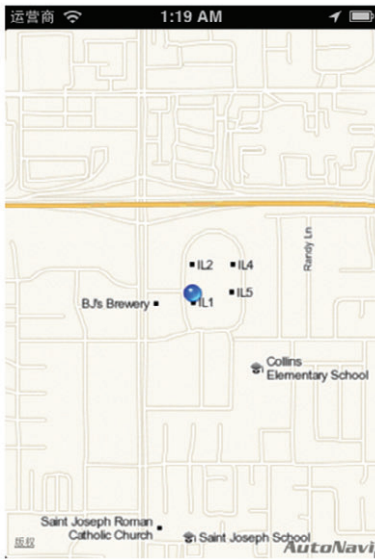


图13-21 跟踪用户位置变化

13.3 使用程序外地图

有时候，我们只使用地图的一些基本功能，如查看位置、导航、参考路线等功能，此时我们不必自己设计地图界面，也不需要面对复杂的API，只需在应用中调用程序外地图即可。调用程序外的地图时，有两个选择：iOS 6苹果地图和谷歌Web地图。

13.3.1 调用iOS 6 苹果地图

iOS设备中都带有一个地图应用，图13-22为iPod touch屏幕中的“地图”，我们可以使用它完成地图相关的大

部分工作。事实上，在iOS 6中，我们可以在自己的应用程序中调用它，并且可以给它传递一些参数进行初始化显示。

在图13-23所示的界面中输入查询地点关键字，然后进行地理信息编码查询，结果会调入自带的苹果地图，如图13-24所示，界面中的标注点是我们传递给它的。我们可以像使用自带地图应用一样进行查询线路、设置地图类型等操作。



图13-22 自带地图应用



图13-23 查询关键字



图13-24 显示标注点

在iOS 6中实现这个功能，需要使用Map Kit中的MKPlacemark和MKMapItem这两个类。因此，我们还需要在工程中添加MapKit.framework。主要代码如下：

```

- (IBAction)geocodeQuery:(id)sender {
    if (_txtQueryKey.text == nil || [_txtQueryKey.text length] == 0) {
        return;
    }

    CLGeocoder *geocoder = [[CLGeocoder alloc] init];
    [geocoder geocodeAddressString:_txtQueryKey.text completionHandler:^(NSArray
        *placemarks, NSError *error) {
        NSLog(@"查询记录数: %i", [placemarks count]);
        if ([placemarks count] > 0) {
            CLPlacemark* placemark = placemarks[0];

            CLLocationCoordinate2D coordinate = placemark.location.coordinate;
            NSDictionary* address = placemark.addressDictionary;
            MKPlacemark *place = [[MKPlacemark alloc]
                initWithCoordinate:coordinate addressDictionary:address];

            MKMapItem *mapItem = [[MKMapItem alloc] initWithPlacemark:place];
            [mapItem openInMapsWithOptions:nil];

            //关闭键盘
            [_txtQueryKey resignFirstResponder];
        }
    }];
}

```

在上述代码中，第③行代码用于实例化MKPlacemark对象。注意，MKPlacemark与CLPlacemark不同，前者是地图上的地标类，后者是定位使用的地标类。第③行代码使用了构造方法initWithCoordinate:

addressDictionary:，其中initWithCoordinate:部分指定地理坐标点，该地理坐标点通过第①行 placemark.location.coordinate语句取得；addressDictionary:部分设置该地点信息，它的参数可以通过第②语句placemark.addressDictionary获得。

第④行代码用于实例化MKMapItem对象。MKMapItem类封装了地图上一个点的信息类。我们把需要在地图上显示的点封装到MKMapItem对象中。initWithPlacemark:是它的构造方法，它的参数是MKPlacemark类型。

第⑤行代码调用iOS自带的苹果地图应用。openInMapsWithLaunchOptions:方法是MKMapItem类的实例方法，其参数是NSDictionary类型，这个参数可以控制显示地图的初始化信息，它包含一些键：

- ❑ **MKLaunchOptionsDirectionsModeKey**。设定路线模式，它有两个值MKLaunchOptionsDirectionsModeDriving（驾车路线）和MKLaunchOptionsDirectionsModeWalking（步行路线）。
- ❑ **MKLaunchOptionsMapTypeKey**。设定地图类型。
- ❑ **MKLaunchOptionsMapCenterKey**。设定地图中心点。
- ❑ **MKLaunchOptionsMapSpanKey**。设置地图跨度。
- ❑ **MKLaunchOptionsShowsTrafficKey**。设置显示交通状况。

例如，我们可以使用下面的代码在地图上设置行车路线：

```
NSDictionary* options = [[NSDictionary alloc] initWithObjectsAndKeys:
    MKLaunchOptionsDirectionsModeDriving, MKLaunchOptionsDirectionsModeKey, nil];

MKMapItem *mapItem = [[MKMapItem alloc] initWithPlacemark:place];
[mapItem openInMapsWithLaunchOptions:options];
```

设置行车路线后，会在地图上标注出路线，如图13-25所示。默认情况下，起点是当前位置，这个位置通过定位服务获得。终点是我们查询的地点，即place变量所包含的信息。

如果有多个点需要标注，我们可以使用MKMapItem的类方法：

```
+ (BOOL)openMapsWithItems:(NSArray *)mapItems launchOptions:(NSDictionary *)launchOptions
```

其中参数mapItems是标注点的集合，launchOptions是启动参数。如果想使用这个方法，则上面的例子可以修改如下：

```
CLGeocoder *geocoder = [[CLGeocoder alloc] init];
[geocoder geocodeAddressString:_txtQueryKey.text completionHandler:^(NSArray
    *placemarks, NSError *error) {
    NSLog(@"查询记录数: %i", [placemarks count]);

    NSMutableArray* array = [NSMutableArray new];

    for (int i = 0; i < [placemarks count]; i++) {
        CLPlacemark* placemark = placemarks[i];

        CLLocationCoordinate2D coordinate = placemark.location.coordinate;
        NSDictionary* address = placemark.addressDictionary;

        MKPlacemark *place = [[MKPlacemark alloc]
            initWithCoordinate:coordinate addressDictionary:address];

        MKMapItem *mapItem = [[MKMapItem alloc] initWithPlacemark:place];

        [array addObject:mapItem];

        //关闭键盘
        [_txtQueryKey resignFirstResponder];
    }
}
```

```

    }

    if ([array count] > 0) {
        [MKMapItem openMapsWithItems:array launchOptions:nil];
    }
}];

```

③

与传递单个标注点不同的是，我们需要进行循环遍历，然后把地图地标对象MKMapItem放到集合中，如第②代码所示。第③行代码用于在循环完成后调用[MKMapItem openMapsWithItems:array launchOptions:nil]启动自带苹果地图。运行结果如图13-26所示，可以发现，传递的两个地标点都标注处理了。



图13-25 标注行车路线



图13-26 多个标注点

13.3.2 调用谷歌Web地图

此外，我们也可以借助谷歌的Web地图API开发地图应用，但这里所涉及的技术都是Web技术，而非本地技术。谷歌提供的地图查询的URL形如：<http://maps.google.com/maps?q=参数>

修改上面的案例，使用谷歌提供的Web地图，相关代码如下：

```

- (IBAction)geocodeQuery:(id)sender {
    if (_txtQueryKey.text == nil || [_txtQueryKey.text length] == 0) {
        return;
    }

    CLGeocoder *geocoder = [[CLGeocoder alloc] init];
    [geocoder geocodeAddressString:_txtQueryKey.text completionHandler:^(NSArray
        *placemarks, NSError *error) {
        NSLog(@"查询记录数: %i", [placemarks count]);
        if ([placemarks count] > 0) {
            CLPlacemark* placemark = placemarks[0];

            CLLocationCoordinate2D coordinate = placemark.location.coordinate;

            NSString *urlString = [NSString stringWithFormat:
                @"http://maps.google.com/maps?q=%f,%f",
                coordinate.latitude,

```

```

        coordinate.longitude];                                ①

        NSURL *url = [NSURL URLWithString:urlString];        ②

        [[UIApplication sharedApplication] openURL:url];      ③

        // 关闭键盘
        [_txtQueryKey resignFirstResponder];

    }

    }
};
}

```

上述代码的调用关键是①、②、③这几行代码。第①行代码设置url访问字符串，其中q后面是参数，其中%f和%f分别代码纬度和经度。第②行代码用于构造一个URL对象。第③行代码使用内置浏览器打开这个URL。该案例的运行结果如图13-27所示。



图13-27 谷歌Web地图

13.4 小结

通过对本章的学习，我们可以了解iOS中的定位服务技术，包括地理信息编码和反编码查询。此外，我们还介绍了iOS 6苹果地图的使用，包括显示地图、添加标注以及跟踪用户位置变化等。最后，我们介绍了程序外地图的使用，包括调用iOS 6苹果地图和谷歌Web地图。

Part 3

第三部分

进阶篇

本 部 分 内 容

- 第 14 章 iOS 中的商业模式
- 第 15 章 找出程序中的 bug——调试
- 第 16 章 基于测试驱动的 iOS 开发
- 第 17 章 让你的程序“飞”起来——性能优化
- 第 18 章 管理好你的程序代码——代码版本控制
- 第 19 章 把你的应用放到 App Store 上

苹果公司于2008年7月推出软件应用市场App Store。App Store作为一个为iOS设备提供应用软件下载的平台，将开发者与用户有机地连接起来。苹果公司规范化地管理这个市场，可以很好地保护开发者的利益。另外，苹果公司为开发者提供了十分详尽的开发资料和文档，在利润上苹果公司与开发者采用3:7的分成方式。在种种因素的作用下，很多开发者投入到iOS的开发中，他们希望有一天凭借优秀的创意和设计创造出属于自己的一片天空。

在本章中，我们主要讲解如何在App Store中淘自己的第一桶金，以及盈利相关的一些技巧。

14.1 收费策略

App Store是一个巨大的应用市场，“买家”来自全球各地。在这个平等、稳定的市场里，开发者作为“卖家”，唯一的奋斗目标就是创造利润。那么如何将利益最大化，如何能够脱颖而出呢？接下来，我们就从以下几个方面来探讨。

- ❑ iOS如何赚钱
- ❑ 避免定价策略误区
- ❑ 免费软件的艺术
- ❑ 在适当的时间、适当的位置植入广告
- ❑ 尝试不同的盈利模式

14.1.1 iOS 如何赚钱

作为App Store淘金者，如何赚钱是开发者最关心的问题。下面我们先介绍几种盈利方式。

- ❑ **产品定价。**应用和游戏开发好后，在上传App Store之前，你需要为自己的产品定价，0.99美元、1.99美元或者更多，这个定价由你决定。之后，用户在App Store下载你的产品，下载一次收费一次。《愤怒的小鸟》、《水果忍者》、《植物大战僵尸》等优秀的产品，依靠下载收费为他们带来了相当可观的收入。
- ❑ **植入广告。**可以在你的产品中放入一些第三方广告，第三方广告联盟根据产品中广告展示次数和点击次数支付给你相关的费用。苹果自身带有iAd广告，做得比较好的还有被谷歌收购的AdMob。产品是否适合植入广告取决于产品的粘稠度，如果粘稠度不高，则当用户打开你的产品时，点几下就关掉了，这样广告就失去了价值。
- ❑ **应用内购买。**这种盈利方式很值得研究，它通过提供增值服务来刺激用户再次消费。应用内购买是目前很多应用（尤其是游戏类应用）最常用的手段，盈利效果非常明显。一般有购买虚拟道具、关卡解锁、延长服务时间、扩充存储空间、升级用户权限、提升隐私级别等方式。现在很多应用采用免费模式，然后在产品中鼓励、刺激用户进行消费。

14.1.2 避免定价策略误区

获得最大的赢利点是我们的最终追求。有些开发者会把产品开发周期、开发成本甚至是主观上对产品的信心作为定价的依据，感觉这款产品可玩性强，定价就高一些，否则就低点。鉴于销售额等于单价×销售量，而产品价格会直接影响产品的销售情况，上述的定价方式难免有些盲目。

单价是开发者决定的，销售量是用户决定的，但是单价的高低会直接影响销售量，获得最大化的利润要从单价和销售量之间寻找一个平衡点。针对某一具体应用的定价，我们至少要对它的潜在用户群、竞争对手进行详细的调研和分析。

14.1.3 免费软件的艺术

免费应用真的免费么？如果想做一款免费的应用，它一定要做得比收费的应用还要好，才会有更大的竞争力。一般用户会认为收费的应用优于免费的应用，但是如果一款免费的应用比一些同类收费的应用更有趣、更实用、用户体验更好，那我为什么还要选择收费的应用呢。免费应用的好评率会大大提高其知名度，这就是免费应用最好的宣传。想从用户手中赚钱，就要先给用户甜头。首先占有广大用户，使用户深深受到免费应用的吸引，这个时候用户已经不知不觉地上了“瘾”，当用户玩劲正酣的时候，总有些功能满足不了他的需求，这时他会发现原来这款应用是有增值服务的，应用内购买的虚拟道具等出现了，同时应用系统本身会不断地鼓励用户购买这些服务。这就是免费软件的艺术，预先取之，必先予之。

上述免费应用的特征一般都是高收入应用的共同特点，也是成为高收入应用的基本要素。要成为顶级高收入应用，还需要一套非常精巧而完善的收费机制。

14.1.4 在适当的时间、适当的地点植入广告

在应用中植入广告是一个很不错的盈利模式。现在iOS设备的用户量相当大，但由于一些消费观念等问题，我们不愿意付费下载应用，更希望使用免费的应用或者破解的应用。这些因素使很多开发者开始转变营销思路，有的进军海外市场，有的使用应用内购买模式，还有的通过投入广告营利。针对这么大的用户量，如果广告投放适当，也会给开发者带来一笔很可观的收益。那么，广告该如何投放呢？

首先，需要分析一下我们的应用是否适合投放广告。这要从产品的可玩性、粘稠度等性能来考虑，如果用户使用时间不长，使用频率也不是很高的话，那么对这类应用投放广告就没什么意义。投放广告的应用一定要可玩性很强，粘度很高，非常吸引用户使用。

对免费应用投放广告，对于开发者来说是一个盈利模式，但是对于用户来说，即使你的应用可以免费下载，他们仍然比较反感广告，会担心广告影响使用或造成不便。开发者在设计时要把这些情况考虑在内，把广告的负面影响降到最低，让用户能接受带广告的应用。

广告出现的位置很重要，一般放到应用的启动和结束的位置，这样对用户的使用不会造成影响。如果是使用性的应用，可以将广告放到顶部或者底部的位置；如果是游戏，可以将广告放到不影响游戏操作、不遮挡视线的位置，或者放到虚拟道具介绍、关卡转换等位置中。广告窗口的选择也很关键，广告窗口的颜色、色调要与整体应用色调一致，如果反差很大，也会造成用户反感。

14.1.5 尝试不同的盈利模式

上面提到，在App Store上面发布产品有很多盈利方式，那么究竟选择哪种方式呢？免费+广告？还是免费+应用内购买？或者是收费+应用内购买？其实，最好的方法是把这些方式都试一下，看哪个盈利模式最适合你的应用。不要单纯地以自己的主观想法去选择盈利方式，实践才是检验真理的唯一标准。

要应用保持良好的收益，经常做一些分析是很有必要的。知己知彼，百战不殆。从产品自身角度、市场角度、

客户群体、竞争对手、合作伙伴、用户价值等方面分析，不断地完善我们的应用。

14.2 使用苹果 iAd 广告

iAd广告是苹果官方广告平台，我们可以在iOS平台上放置iAd广告。对于在iAd中投放广告的商家有很高的门槛限制，不过相对而言对开发者的回报率比较好，但遗憾的是，iAd广告支持的国家很少。

提示 在本书成书之前，苹果只在11个国家开通了iAd广告，它们是美国、加拿大、澳大利亚、新西兰、日本、德国、英国、意大利、法国、西班牙和墨西哥。其他国家的用户看不到应用上的广告。

下面我们从技术角度来实现这些策略。本节先介绍苹果iAd广告API以及如何将iAd广告添加到你的应用中。iAd广告有横幅广告和插页广告，它们的API完全不同，应用场景也完全不同。

14.2.1 横幅广告

横幅广告像“条幅”一样挂在屏幕上，在屏幕中某一位置占有部分空间。当点击横幅广告时，导航到另外的一个应用或者弹出模态窗口以呈现广告的细节。点击关闭广告按钮，可以回到原始的屏幕。

无论横屏还是竖屏的情况，横幅广告在不同iOS设备中的尺寸都是固定的，具体如下所示。

- ❑ 在iPhone和iPod touch横屏的情况下，横幅广告的尺寸是480×32。
- ❑ 在iPhone和iPod touch竖屏的情况下，横幅广告的尺寸是320×50。
- ❑ 在iPad横屏的情况下，横幅广告的尺寸是1024×66。
- ❑ 在iPad竖屏的情况下，横幅广告的尺寸是768×66。

提示 在iPhone 5中，情况有些特殊，iPhone 5的屏幕尺寸是320×568点，而不是iPhone 4的320×480点，因此在iOS 6（iPhone 5的最低版本是iOS 6）中，横幅广告应该是自适应宽度，在设计时要把这个问题考虑在内。

横幅广告栏是由ADBannerView类定义的，它是iOS对象库中的标准控件。要把它添加到屏幕中，可以通过Interface Builder设计器将其拖曳到设计窗口，如图14-1所示。

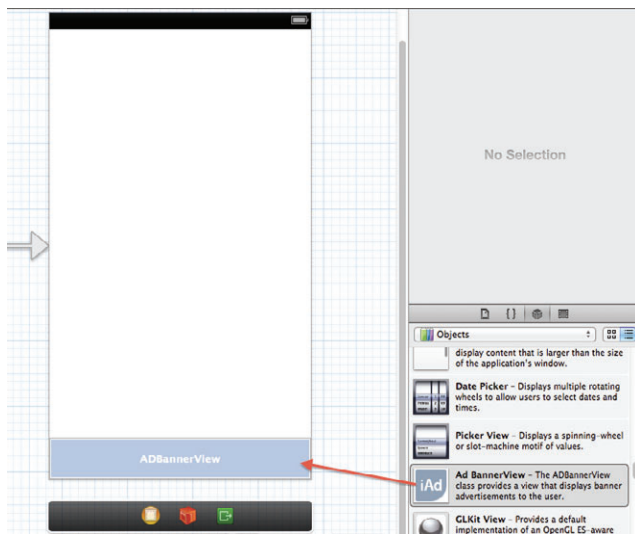


图14-1 从Interface Builder设计器中添加横幅广告栏

直接从对象库中拖曳的这种方式在横屏和竖屏切换时不够灵活，在应对iPhone 5横屏情况时也不方便，因此，我们推荐采用编码方式动态设定ADBannerView的位置。无论采用哪一种方式，在使用iAd之前，都需要引入iAd框架。如图14-2所示，在框架库中选择iAd.framework并将其添加到当前工程中。

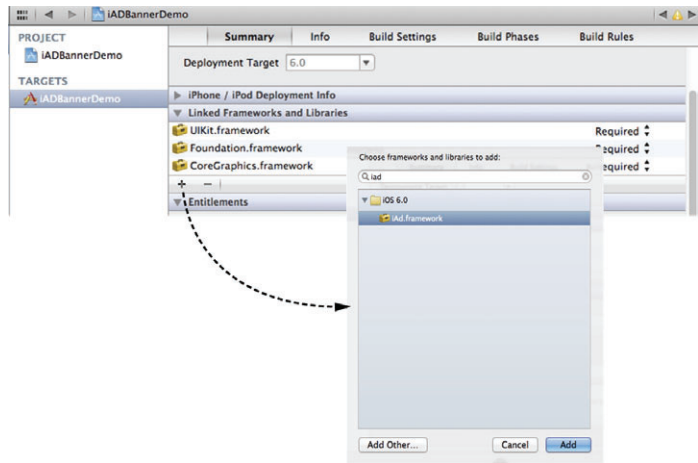


图14-2 添加iAd框架

iAd框架也是插页广告必须使用的框架。下面我们看一个横幅广告栏案例——iADBannerDemo，它使用分屏控件（UIPageControl）展示4张图片，我们在屏幕的顶部放置ADBannerView，在屏幕旋转时ADBannerView的大小也会相应地变化。图14-3为ADBannerView竖屏的情况，图14-4为ADBannerView横屏的情况。

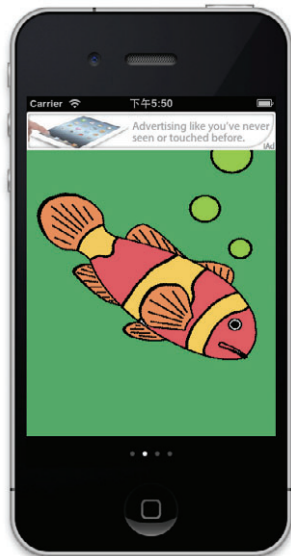


图14-3 ADBannerView竖屏的情况

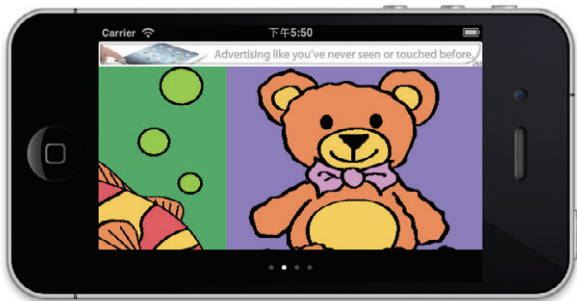


图14-4 ADBannerView横屏的情况

下面为iADBannerDemo实现代码的ViewController.h部分：

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

@interface ViewController : UIViewController <UIScrollViewDelegate,
```

```

ADBannerViewDelegate>

@property (strong, nonatomic) UIImageView *page1;
@property (strong, nonatomic) UIImageView *page2;
@property (strong, nonatomic) UIImageView *page3;
@property (strong, nonatomic) UIImageView *page4;

@property (weak, nonatomic) IBOutlet UIScrollView *scrollView;
@property (weak, nonatomic) IBOutlet UIPageControl *pageControl;
@property (strong, nonatomic) ADBannerView *bannerView;

- (IBAction)changePage:(id)sender;

@end

```

首先，我们引入了iAd/iAd.h头文件。此外，还要实现ADBannerViewDelegate协议，该协议规定了ADBannerView加载成功和加载失败的方法，以及响应动作开始和结束等方法。我们还定义了ADBannerView *的属性bannerView，注意属性参数strong即强引用类型。

下面我们再看看ViewController.m中视图控制器初始化相关的代码：

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.pageControl.numberOfPages = 4;

    self.scrollView.frame = self.view.frame;

    self.scrollView.delegate = self;

    self.scrollView.contentSize = CGSizeMake(4 * 320.0f, 420.0f);

    self.page1 = [[UIImageView alloc] initWithImage:[UIImage
        imageNamed:@"animal-1.png"]];
    self.page1.frame = CGRectMake(0.0f, 0.0f, 320.0f, 420.0f);
    [self.scrollView addSubview:self.page1];

    self.page2 = [[UIImageView alloc] initWithImage:[UIImage
        imageNamed:@"animal-2.png"]];
    self.page2.frame = CGRectMake(320.0f, 0.0f, 320.0f, 420.0f);
    [self.scrollView addSubview:self.page2];

    self.page3 = [[UIImageView alloc] initWithImage:[UIImage
        imageNamed:@"animal-3.png"]];
    self.page3.frame = CGRectMake(320.0f * 2, 0.0f, 320.0f, 420.0f);
    [self.scrollView addSubview:self.page3];

    self.page4 = [[UIImageView alloc] initWithImage:[UIImage
        imageNamed:@"animal-4.png"]];
    self.page4.frame = CGRectMake(320.0f * 3, 0.0f, 320.0f, 420.0f);
    [self.scrollView addSubview:self.page4];

    //实例化ADBannerView时，我们采用构造方法initWithAdType:
    if ([ADBannerView instancesRespondToSelector:@selector(initWithAdType:)]) {
        self.bannerView = [[ADBannerView alloc] initWithAdType:ADAdTypeBanner];
        //设定iOS 6广告栏自动调整宽度
        self.bannerView.autoresizingMask = UIViewAutoresizingFlexibleWidth;
    } else {
        self.bannerView = [[ADBannerView alloc] init];
    }
    self.bannerView.delegate = self;

    [self.view addSubview:self.bannerView aboveSubview:self.scrollView];
}

```

```

    }

    - (void)viewDidLayoutSubviews
    {
        [self layoutAnimated:[UIView areAnimationsEnabled]];
    }

    - (void)layoutAnimated:(BOOL)animated
    {
        //在iOS 6中, 广告栏是自动调整的
        //为了支持iOS 5.0及以下版本, 还需要设定currentContentSizeIdentifier
        #if __IPHONE_OS_VERSION_MIN_REQUIRED < __IPHONE_6_0 ④
            CGRect contentFrame = self.view.bounds;
            if (contentFrame.size.width < contentFrame.size.height) {
                self.bannerView.currentContentSizeIdentifier =
                    ADBannerContentSizeIdentifierPortrait;
            } else {
                self.bannerView.currentContentSizeIdentifier =
                    ADBannerContentSizeIdentifierLandscape;
            }
        #endif

        CGRect bannerFrame = _bannerView.frame;
        if (self.bannerView.bannerLoaded) { ⑤
            //设置在屏幕上边
            bannerFrame.origin.y = 0;
        } else {
            //设置在屏幕之外, 使之隐藏起来
            bannerFrame.origin.y -= self.bannerView.frame.size.height;
        }

        [UIView animateWithDuration:animated ? 0.25 : 0.0 animations:^( ⑥
            self.bannerView.frame = bannerFrame;
        )];
    }

    #if __IPHONE_OS_VERSION_MIN_REQUIRED < __IPHONE_6_0 ⑦
    - (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
        interfaceOrientation
    {
        return YES;
    }
    #endif

    - (NSUInteger)supportedInterfaceOrientations ⑧
    {
        return UIInterfaceOrientationMaskAll;
    }

```

在上述代码中, viewDidLoad方法是视图控制器加载时调用的方法。在该方法中, 我们需要实例化ADBannerView。在iOS 6中, 它的实例化方式有所不同。为了使应用能同时适用于iOS 6之前的版本, 我们做如下处理:

```

    if ([ADBannerView instancesRespondToSelector:@selector(initWithAdType:)]) {
        self.bannerView = [[ADBannerView alloc] initWithAdType:ADAdTypeBanner]; ①
        //设定iOS 6广告栏自动调整宽度
        self.bannerView.autoresizingMask = UIViewAutoresizingFlexibleWidth; ②
    } else {
        self.bannerView = [[ADBannerView alloc] init]; ③
    }

```

其中采用发送ADBannerView类的instancesRespondToSelector:消息判断ADBannerView类是否有实例方法initWithAdType, 如果有, 则说明这是iOS 6的SDK。在iOS 6中, 实例化ADBannerView的代码如第①行代码所示。第②行代码设置广告栏自动调整宽度。第③行代码是iOS 6之前的实例化方式。

`viewDidLoadLayoutSubviews`方法重写`UIViewController`类,该方法会在重新加载子视图时调用。在该方法中,我们调用了`layoutAnimated:`方法。`layoutAnimated:`方法是我们自定义的方法,主要为了处理视图的重新加载、屏幕初始化和屏幕的旋转等。

第④行代码采用了预编译指令`#if __IPHONE_OS_VERSION_MIN_REQUIRED < __IPHONE_6_0`判断当前iOS版本是否小于6.0,因为iOS 6之后不再支持`ADBannerView`的`currentContentSizeIdentifier`属性了。类似的问题还有第⑦行的`shouldAutorotateToInterfaceOrientation:`方法,iOS 6之后也不再支持该方法,而用`supportedInterfaceOrientations`方法来取代它。

第⑤行代码表示成功加载`ADBannerView`后,要在屏幕中显示`ADBannerView`,需要设定它的`y`坐标为0,这样`ADBannerView`会出现在屏幕顶部,否则需要隐藏起来。

第⑤行代码用于计算`ADBannerView`的位置,第⑥行代码才重新改变它的位置,并且以动画效果的形式显示。下面我们再看看实现代码`ViewController.m`中有关`ADBannerViewDelegate`委托的实现代码:

```
- (void)bannerViewDidLoadAd: (ADBannerView *)banner
{
    NSLog(@"广告加载成功");
    [self layoutAnimated:YES];
}

- (void)bannerView: (ADBannerView *)banner didFailToReceiveAdWithError: (NSError *)error
{
    NSLog(@"广告加载失败");
    [self layoutAnimated:YES];
}

- (void)bannerViewActionDidFinish: (ADBannerView *)banner
{
    NSLog(@"广告关闭");
    [self layoutAnimated:YES];
}
```

`bannerViewDidLoadAd:`和`bannerView:didFailToReceiveAdWithError:`方法会被反复回调。因此,第一次出现“广告加载失败”也没有关系,可以等到下一次调用。

运行一下看看效果,加载失败时不显示横幅广告,加载成功时会看到如图14-3和图14-4所示的广告横幅,点击广告栏,启动广告详细内容界面,如图14-5所示。

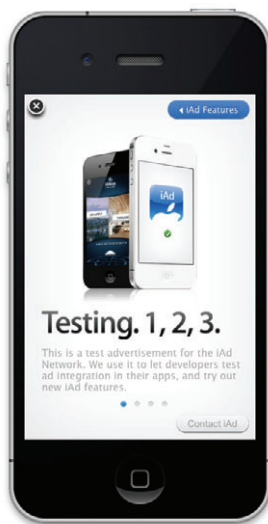


图14-5 启动广告详细内容界面

提示 图14-3和图14-4的横幅广告栏以及图14-5所示的广告界面都是苹果给我们的测试条幅和界面，并不是真实的内容，我们只需要关注横幅广告栏是否会出现、它的位置，以及是否能够进入广告详细界面即可。在实际发布时，应用必须审核通过上线后才能看到真正的内容。

14.2.2 插页广告

与横幅广告不同，插页广告可以占用屏幕的全部空间。为了不引起用户反感，苹果对插页广告的使用有一些限制和指导意见。插页广告只能用于iPad设备，只能在“内容显示”和“过渡画面”这两个场景中使用。

1. 内容显示场景

这个场景是指插页广告与其他应用的内容一起呈现，主要在杂志等平铺页面导航模式下使用。如图14-6所示，原本只有3个画面，我们可以在它们之间添加插页广告。

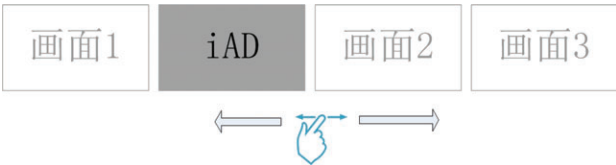


图14-6 内容显示的插页广告

在这种方式中，广告画面看上去很像应用的一部分，不过它的内容是广告。插页广告画面以非模态方式呈现。比如，在看一部电影时，我们看到主人公拿着一瓶国窖1573在喝，这种模式的插页广告搞不好就会引起用户的极大反感。

我们通过案例（iAdFullScreen1Demo）介绍这种场景下插页广告的实现，案例设计原型图如图14-7所示。原本有3个图片视图放置在ScrollView中，通过左右滑屏来浏览它们，下面还有一个分屏控件来控制这些视图。当插页广告加载成功时，会添加一个视图，这样4个视图就并排放入ScrollView中了。当点击插页广告时，会弹出模态对话框，从中可以查看详细的广告内容。



图14-7 内容显示插页广告案例

下面为iAdFullScreen1Demo实现代码的ViewController.h部分：

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

```

#define S_WIDTH 1024           //屏幕宽度
#define S_HEIGHT 768          //屏幕高度

@interface ViewController : UIViewController <UIScrollViewDelegate, ADInterstitialAdDelegate>

@property (strong, nonatomic) NSMutableArray *pageList;

@property (weak, nonatomic) IBOutlet UIScrollView *scrollView;
@property (weak, nonatomic) IBOutlet UIPageControl *pageControl;
@property (strong, nonatomic) ADInterstitialAd *interstitial;

- (IBAction)changePage:(id)sender;

@end

```

在iOS中, 插页广告类是ADInterstitialAd, 使用这个类时, 也需要引入iAd/iAd.h头文件, 还需要实现ADInterstitialAdDelegate协议, 该协议规定了ADInterstitialAd加载成功和加载失败的方法, 以及响应动作开始和结束等方法。此外, 我们还定义了ADInterstitialAd *的属性interstitial, 注意属性参数strong是强引用类型。此外, 属性pageList用来装载要浏览的视图对象。

下面我们再看看ViewController.m中视图控制器初始化相关的主要代码:

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.interstitial = [[ADInterstitialAd alloc] init];           ①
    self.interstitial.delegate = self;                             ②

    self.scrollView.delegate = self;

    self.pageList = [[NSMutableArray alloc] init];
    [self.pageList addObject:[UIImageView alloc] initWithImage:[UIImage
        imageNamed:@"apple-1.jpg"]]];
    [self.pageList addObject:[UIImageView alloc] initWithImage:[UIImage
        imageNamed:@"apple-2.jpg"]]];
    [self.pageList addObject:[UIImageView alloc] initWithImage:[UIImage
        imageNamed:@"apple-3.jpg"]]];

    [self reloadPage];                                           ③
}

//重新加载画面中的视图
- (void)reloadPage
{
    //内容视图高度
    int contentHeight = self.scrollView.frame.size.height;

    self.pageControl.numberOfPages = [self.pageList count];      ④

    self.scrollView.contentSize = CGSizeMake([self.pageList count] * S_WIDTH,
        contentHeight);                                           ⑤

    for (int i = 0; i < [self.pageList count]; i++) {            ⑥
        UIView *view = self.pageList[i];
        [view removeFromSuperview];
        view.frame = CGRectMake(S_WIDTH * i, 0.0f, S_WIDTH, contentHeight);
        [self.scrollView addSubview:view];
    }
}

```

在viewDidLoad方法中, 第①行代码实例化了ADInterstitialAd, 并赋值给属性interstitial。第②行代码把当前视图控制器对象分配给ADInterstitialAd委托对象。

第③行代码调用自定义的reloadPage方法。由于ADInterstitialAd广告内容的插入视图数由原来的3个变成了4个，我们要使用reloadPage方法将这个新的视图添加到ScrollView中去。在reloadPage方法中，第④行代码重新计算分屏控件的总页数，第⑤行代码重新调整ScrollView的内容视图大小，第⑥行代码中的for循环将集合中的视图添加到ScrollView上。为了防止重复添加，应该先从ScrollView移除一下再添加为好。

下面我们来看ViewController.m中有关ADInterstitialAdDelegate委托协议的实现代码：

```
// 每次加载一个新广告时，都会调用该方法
- (void)interstitialAdDidLoad:(ADInterstitialAd *)interstitialAd
{
    NSLog(@"广告加载成功");

    UIView* interstitialContainer = [[UIView alloc] initWithFrame:CGRectMake(0,0,320,50)]; ①
    [self.pageList insertObject:interstitialContainer
                    atIndex:self.pageControl.currentPage]; ②
    [self reloadPage]; ③
    [self.interstitial presentInView:interstitialContainer]; ④
}

// 卸载ADInterstitialAd对象时调用该方法
- (void)interstitialAdDidUnload:(ADInterstitialAd *)interstitialAd
{
    NSLog(@"广告卸载");
    for (UIView* item in self.pageList) { ⑤
        if ([item isKindOfClass:[UIView class]]) {
            [item removeFromSuperview]; ⑥
            [self.pageList removeObject:item]; ⑦
            [self reloadPage]; ⑧
            break;
        }
    }
}

// 执行一个广告动作时回调，返回YES表示可以执行动作，返回NO表示不可以执行
- (void)interstitialAd:(ADInterstitialAd *)interstitialAd didFailWithError:
    (NSError *)error
{
    NSLog(@"广告加载失败");
}

// 执行一个广告动作时回调，返回YES表示可以执行动作，返回NO表示不可以执行
- (BOOL)interstitialAdActionShouldBegin:(ADInterstitialAd *)interstitialAd
    willLeaveApplication:(BOOL)willLeave
{
    return YES;
}

// 用户关闭模态广告窗口时，回调该方法
- (void)interstitialAdActionDidFinish:(ADInterstitialAd *)interstitialAd
{
    NSLog(@"广告关闭");
}
```

这5个方法都是ADInterstitialAdDelegate委托协议的方法，其中interstitialAdDidLoad:方法在新的广告加载成功时调用。全部广告必须要在一个视图控制器当中，所以第①行代码实例化UIView*类型的interstitialContainer对象，其中interstitialContainer是插页广告的容器。第②行代码把interstitialContainer视图添加到pageList集合中，然后在第③行代码中重新加载所有视图。第④行代码在interstitialContainer中呈现插页广告interstitial，presentInView:方法是ADInterstitialAd类中的实例方法，用于呈现插页广告ADInterstitialAd。

interstitialAdDidUnload:方法在ADInterstitialAd对象卸载时调用。ADInterstitialAd对象呈现一定

时间后会主动卸载，因此，在该方法中需要注意的是我们不但要从ScrollView中移除原来的插页广告容器interstitialContainer，还要把它从pageList集合中移除掉，第⑤~⑧行代码的for循环实现了这个目的。

interstitialAd:didFailWithError:方法在广告加载失败时调用。在广告加载失败的情况下，应用会在一定时间后尝试重新加载，因此看到日志加载失败时不用惊慌。

interstitialAdActionShouldBegin:willLeaveApplication:方法在广告执行一个动作时，询问是否可以执行，如果返回YES，表示可以继续执行动作，返回NO，表示不能执行。

interstitialAdActionDidFinish:方法在广告动作执行完毕时回调。

运行一下看看插页广告能否呈现，有时加载比较慢，有时还会有加载错误，如果要触发卸载方法，需要等待更长的时间。

2. 过渡画面场景

过渡画面场景是在两个画面转移（或跳转）时出现插页广告，广告画面以模态方式弹出，关闭之后可以回到原来的画面。如图14-8所示，在画面1跳转到画面2时，会弹出模态iAd插页广告。



图14-8 过渡画面的插页广告

这种广告就像我们看电影时突然暂停屏幕，出现请休息一下，播放15分钟的广告后再回来继续。这种广告也非常容易引起用户反感，我们要慎用。一般在游戏结束、音乐播放完成后插入插页广告为好。

下面我们通过案例（iAdFullScreen2Demo）介绍一下这种场景下插页广告的实现。案例设计原型图如图14-9所示，在点击“切换图片”按钮时，不是马上进入第二个画面，而是先弹出广告的模态对话框，关闭该对话框之后才显示第二个画面。



图14-9 过渡画面插页广告案例

下面为iAdFullScreen2Demo实现代码的ViewController.h部分：

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

```

@interface ViewController : UIViewController <ADInterstitialAdDelegate>

@property (strong, nonatomic) UIImageView *page1;
@property (strong, nonatomic) UIImageView *page2;

@property (strong, nonatomic) ADInterstitialAd *interstitial;

- (IBAction)switchView:(id)sender;

@end

```

下面我们在看看ViewController.m中视图控制器viewDidLoad方法的代码：

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.page1 = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"apple-1.jpg"]];
    self.page1.frame = CGRectMake(0.0f, 0.0f, 1024.0f, 704.0f);
    [self.view addSubview:self.page1];

    self.page2 = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"apple-2.jpg"]];
    self.page2.frame = CGRectMake(0.0f, 0.0f, 1024.0f, 704.0f);

    self.interstitial = [[ADInterstitialAd alloc] init];
    self.interstitial.delegate = self;
}

```

下面是点击“切换图片”按钮时调用的switchView:方法的代码：

```

- (IBAction)switchView:(id)sender {
    [UIView animateWithDuration:1.0 animations:^(
        if ([self.page1 removeFromSuperview]) {
            [self.page1 removeFromSuperview];
            [self.view addSubview:self.page2];
        } else {
            [self.page2 removeFromSuperview];
            [self.view addSubview:self.page1];
        }
    ) completion:^(BOOL finished) {
        if (self.interstitial.loaded) {
            [self.interstitial presentFromViewController:self];
        }
    }];
}

```

①

②

在上述代码中，UIView的animateWithDuration:animations:completion:方法实现了视图动画切换。animations部分和completion部分定义的都是代码块：前者是在播放动画时调用，其中实现了将两个视图添加和移除的工作；后者在完成动画时调用，其中实现了以模式方式呈现插页广告，第①行代码判断广告是否加载成功，如果加载成功，则通过第②行语句presentFromViewController:方法模式呈现。

下面我们看看ViewController.m中有关ADInterstitialAdDelegate委托的实现代码：

```

//ADInterstitialAd对象卸载时调用该方法
- (void)interstitialAdDidUnload:(ADInterstitialAd *)interstitialAd
{
    NSLog(@"广告卸载");
    self.interstitial.delegate = nil;
    self.interstitial = [[ADInterstitialAd alloc] init];
    self.interstitial.delegate = self;
}

//一个新的广告被加载每次都会调用该方法
- (void)interstitialAdDidLoad:(ADInterstitialAd *)interstitialAd
{
    NSLog(@"广告加载成功");
}

```

```

}

//获得广告内容出错时调用该方法
- (void)interstitialAd: (ADInterstitialAd *)interstitialAd didFailWithError:
    (NSError *)error
{
    NSLog(@"广告加载失败");
}

//执行一个广告动作执行调用, 返回YES则可以执行动作, 返回NO则不可以执行
- (BOOL)interstitialAdActionShouldBegin: (ADInterstitialAd *)interstitialAd
    willLeaveApplication: (BOOL)willLeave
{
    return YES;
}

//用户关闭模态广告窗口时回调该方法
- (void)interstitialAdActionDidFinish: (ADInterstitialAd *)interstitialAd
{
    NSLog(@"广告关闭");
    self.interstitial.delegate = nil;
    self.interstitial = [[ADInterstitialAd alloc] init];
    self.interstitial.delegate = self;
}

```

由于我们想一直循环加载广告, 无论是关闭模态广告后还是广告卸载后, 都需要重新创建ADInterstitialAd对象, 然后发起广告请求。因此, 我们会看到interstitialAdActionDidFinish:和interstitialAdDidUnload:方法中都有下面的几行代码:

```

self.interstitial.delegate = nil;
self.interstitial = [[ADInterstitialAd alloc] init];
self.interstitial.delegate = self;

```

运行一下看看在切换视图时能否呈现广告, 如果加载成功, 则可以看到广告。注意观察在加载失败后是否能在一段时间后再次加载, 在关闭模态广告后过一段时间是否能够加载, 还有广告一直放置一段时间后能否卸载, 卸载后一段时间内是否能加载。

14.2.3 查看你的收入

如果应用发布成功, 则可以通过iTunes Connect(<https://itunesconnect.apple.com/WebObjects/iTunesConnect.woa>)网站查看应用的下载量和广告收入情况。登录时需要使用开发者账号, 登录成功后看到的主要界面如图14-10所示。

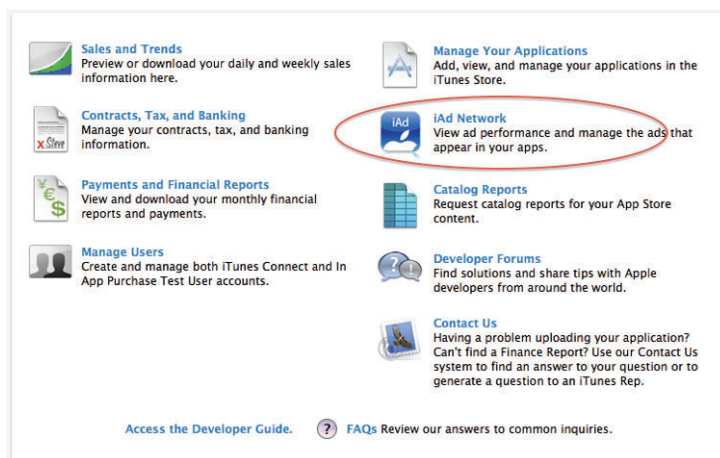


图14-10 iTunes Connect登录成功界面

点击iAd Network超链接，进入iAd网络页面。首先，页面最上方是Summary，如图14-11所示，在这部分中我们可以整体查看所有应用的广告收入情况。

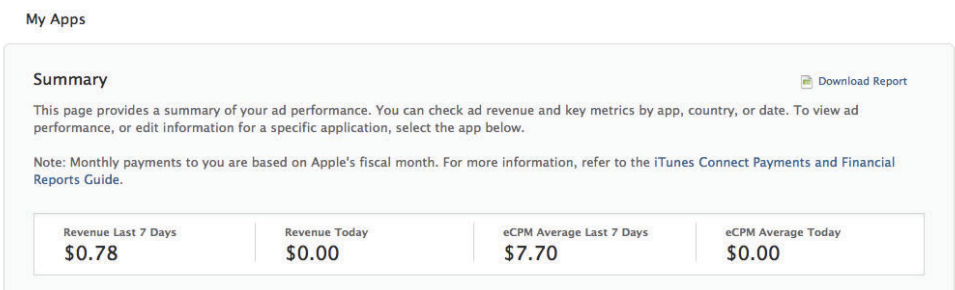


图14-11 iAd网络Summary部分

图14-12和图14-13是iAd网络Performance的图表和表格，它们以不同的形式展示你的账户下应用广告的收入。在图14-13中，Ad Stauts（广告状态）一栏如果是绿色的Live Ads，则说明该应用iAd网络已经激活。

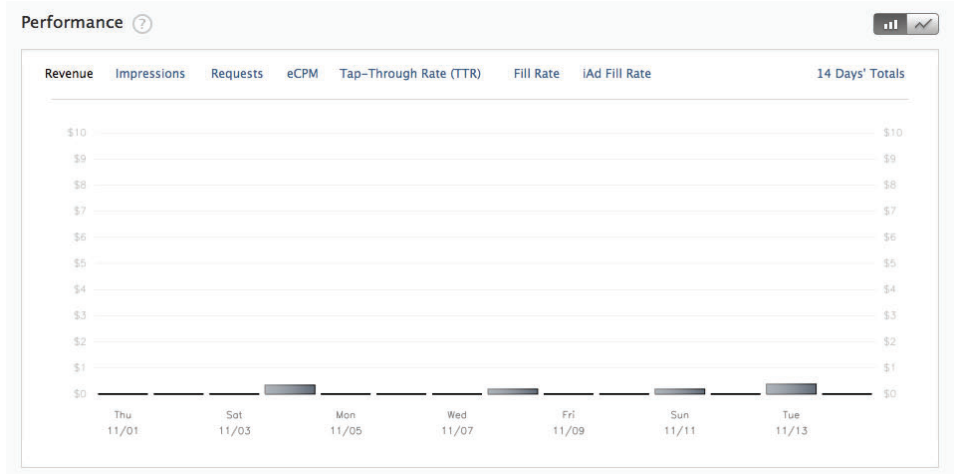


图14-12 iAd网络Performance部分图表界面

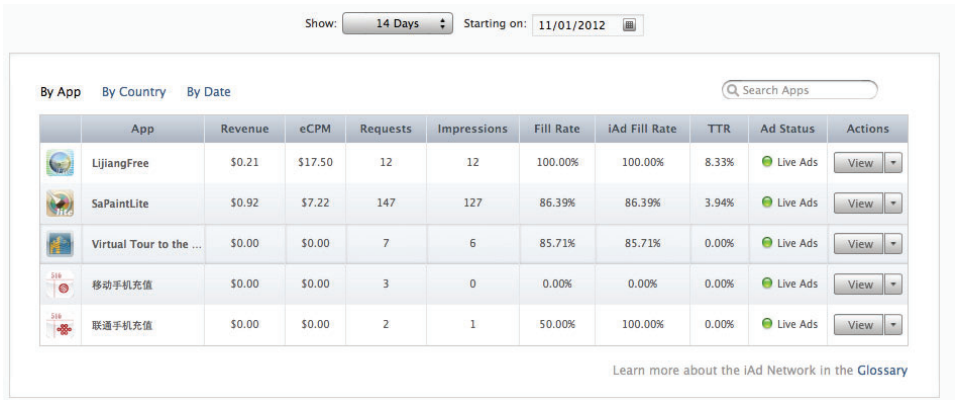


图14-13 iAd网络Performance部分表格界面

下面我们解释一下图14-13中的相关名词。

- ❑ Revenue。收入。
- ❑ eCPM。每一千次展示可以获得的广告收入。
- ❑ Requests。请求次数。
- ❑ Impressions。浏览次数。
- ❑ Fill Rate。填充率，Impressions / Requests的结果。这个比例过低，说明效果不好。
- ❑ iAd Fill Rate。iAd广告填充率。
- ❑ TTR。点击率，这个指标很能说明广告有效性。广告的收入分为展示收入和点击收入，而点击收入要比展示收入高。

这些图表一方面可以帮助我们查看收入情况，另一方面可以帮助我们调整收费策略。例如，如果填充率比较低，可能是广告在应用中展示的时间太短，需要设定计时器延迟它的展示时间。再如，点击率比较低，可能是广告放的位置有问题。

14.3 使用谷歌 AdMob 广告

在一些无法显示iAd广告的国家，使用谷歌的AdMob广告是一个非常不错的选择。

14.3.1 注册 AdMob 账号和管理应用

要我们的应用添加AdMob广告，首先需要注册AdMob账号。AdMob主页是<https://www.google.com/ads/admob/>，登录页面的网址是<https://www.admob.com/login/>，如图14-14所示。登录时，可以使用早期的AdMob账号，也可以使用谷歌账号。在登录页面中点击Need to register?或Create an account now后，可以注册谷歌账号。

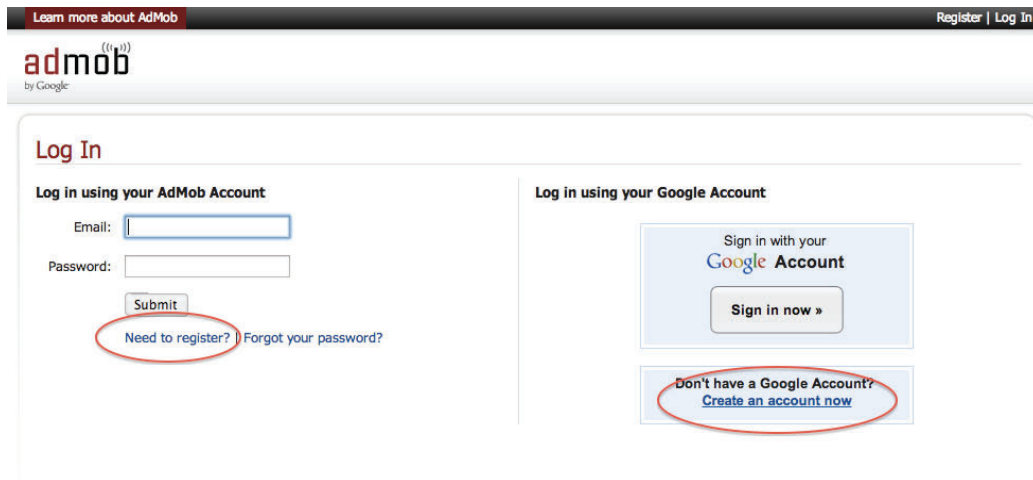
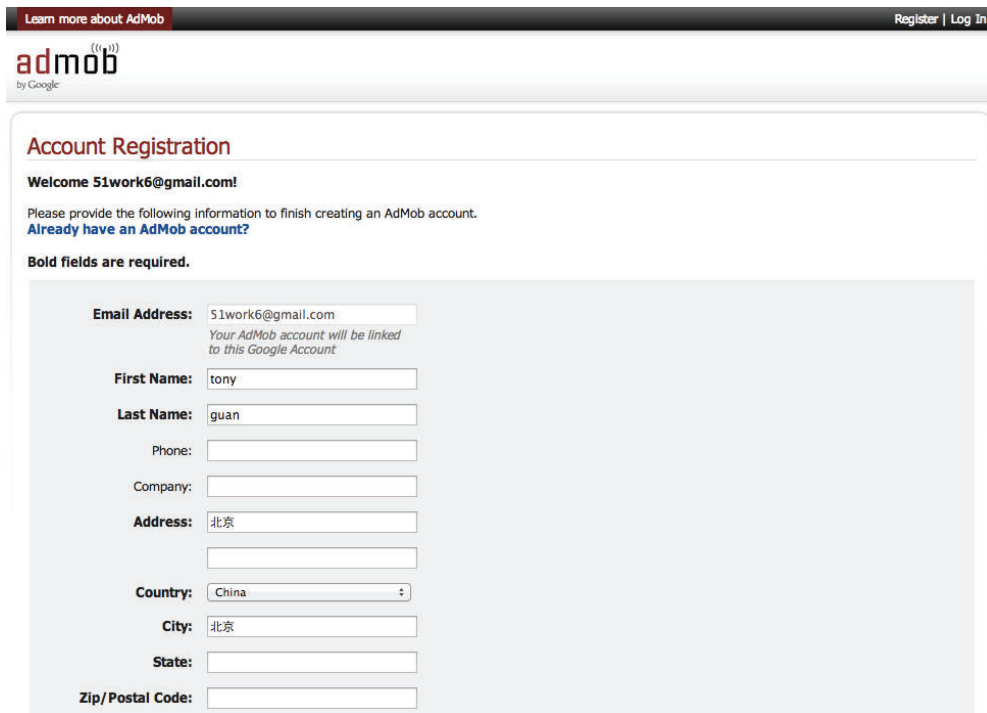


图14-14 AdMob登录页面

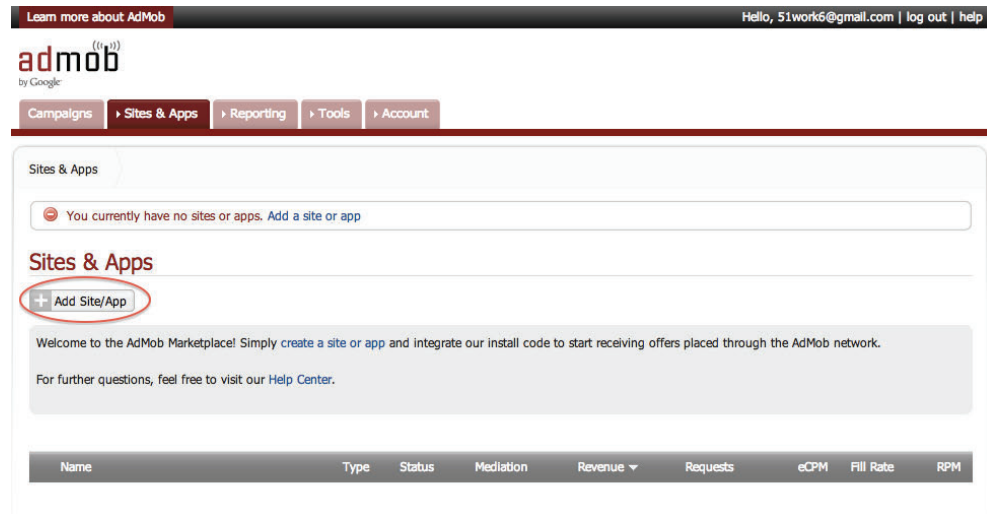
如果是第一次登录，还需要为AdMob填写完整的信息，如图14-15所示。填写必要而完整的信息后，提交信息就可以了。



The image shows the AdMob Account Registration page. At the top, there is a navigation bar with "Learn more about AdMob" and "Register | Log in". The AdMob logo is prominently displayed. The main heading is "Account Registration". Below it, a welcome message says "Welcome 51work6@gmail.com!". A note asks the user to provide information to finish creating an AdMob account, with a link for those who already have an account. A note states "Bold fields are required." The registration form includes fields for Email Address (51work6@gmail.com), First Name (tony), Last Name (guan), Phone, Company, Address (北京), Country (China), City (北京), State, and Zip/Postal Code. A note indicates that the AdMob account will be linked to the Google Account.

图14-15 AdMob注册页面

登录成功之后，我们就可以管理应用。选择Sites & Apps标签，进入应用管理页面，如图14-16所示，点击Add Site/App按钮可以添加应用。



The image shows the AdMob Sites & Apps management page. The top navigation bar includes "Learn more about AdMob" and "Hello, 51work6@gmail.com | log out | help". The AdMob logo is present. The main navigation bar has tabs for Campaigns, Sites & Apps, Reporting, Tools, and Account. The "Sites & Apps" tab is selected. A message states "You currently have no sites or apps. Add a site or app". Below this, the "Sites & Apps" section has a button labeled "+ Add Site/App" which is circled in red. A welcome message to the AdMob Marketplace is displayed, along with a link to the Help Center. At the bottom, there is a table header with columns: Name, Type, Status, Mediation, Revenue, Requests, eCPM, Fill Rate, and RPM.

图14-16 添加应用

如果账号中有关付费的信息不完整，则会进入付费账号信息页面，如图14-17所示。这个信息非常重要，谷歌根据该信息为我们付款。

Learn more about AdMob Hello, 51work5@gmail.com | log out | help

admob
by Google

Campaigns Sites & Apps Reporting Tools Account

Account Payment Details

Please use only English characters to fill out account payment details.

Payment Details

Before you create a site please complete the technical contact details and payment preferences.

Tax Information :

AdMob is required to collect tax information from all its publishers - you are responsible for keeping it up to date. You can change your tax information at any time.

Country: China

Account Type: Please Select

Business Name:
Must match name on tax return

Local Tax ID:

Payment Details :

☒ Pay via ACH/Wire ☐ Pay via PayPal (Is PayPal supported in my country?)

图14-17 添加付费账号信息

如果付费账号信息完整，页面会直接跳转到图14-18所示的页面。目前，AdMob提供的应用类型有Android、iPad、iPhone和Windows Phone 7移动应用。

Learn more about AdMob Hello, si92@sina.com | log out | help

admob
by Google

Campaigns Sites & Apps Reporting Tools Account

Sites & Apps Add Site/App

Add Site/App

Site Info Get Site Code

Select a site or app type

Android App iPad App iPhone App Windows Phone 7 App

Publisher Help

Select your site type and complete the information below to register your site or app and retrieve the appropriate Publisher Code on the subsequent page.

Each type of site or app has a specific version of Publisher Code required to integrate with the AdMob marketplace.

Details

App name:

App Store URL:

Category: Select a category

App description:

图14-18 选择添加的应用类型

在输入信息中, App name非常重要, 必须添加, App Store URL不是必须填写的信息, App description是必须填写的信息。需要注意的是, 选择不同的类型, 详细信息中的内容也有所不同。输入完成后, 点击Continue按钮继续, 跳转到如图14-19所示的界面。

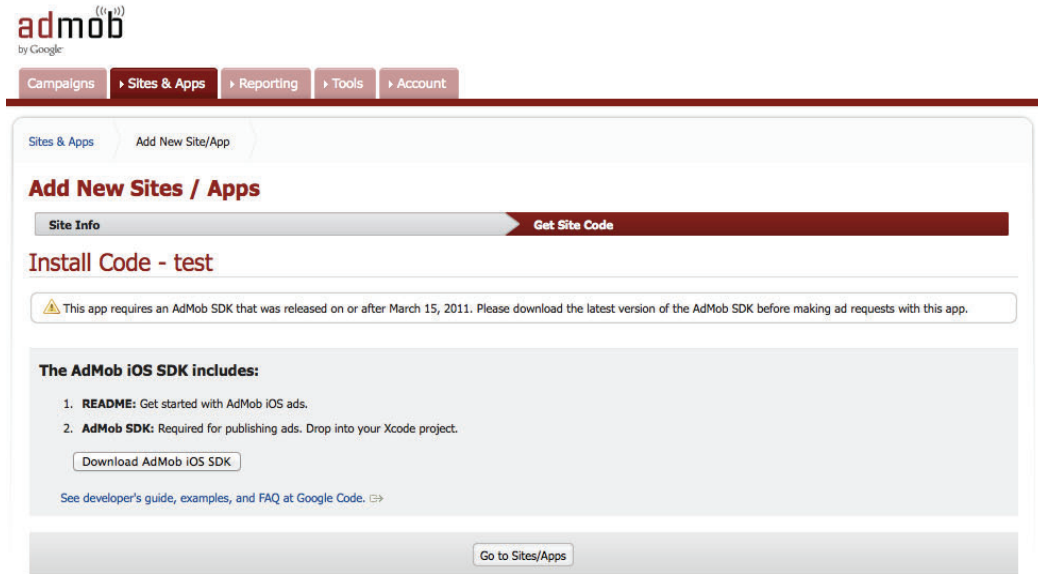


图14-19 成功添加应用界面

在图14-19所示的成功界面中, 可以下载AdMob iOS SDK, 也可以查看帮助获得实例代码。点击Go to Sites/Apps按钮, 页面跳转回到图14-20所示的应用管理页面。

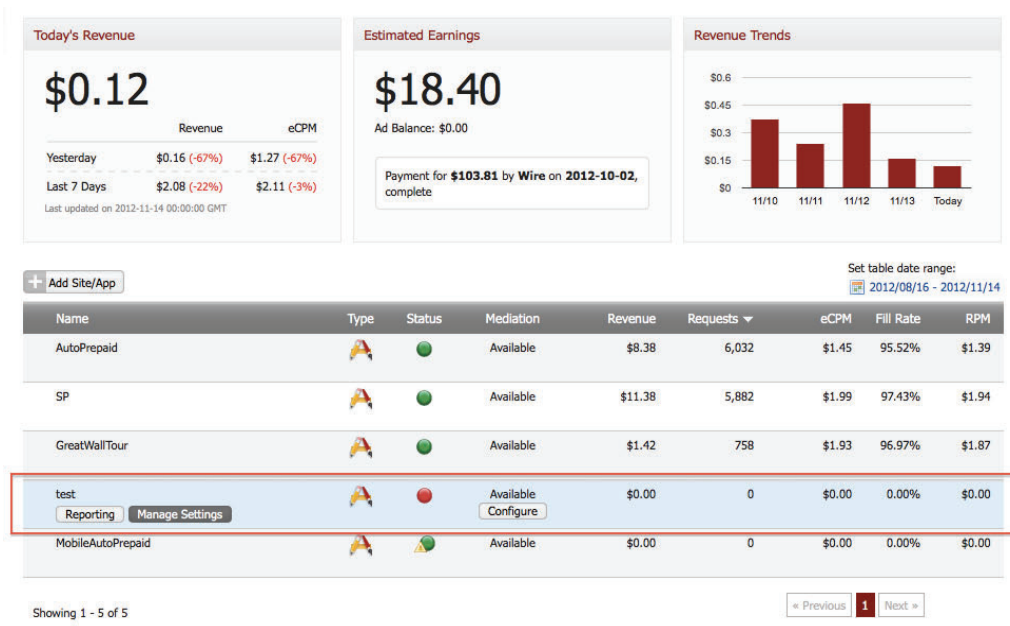


图14-20 应用管理页面

test是我们刚刚添加的。当把鼠标放在test行时，下面会出现Reporting和Manage Settings按钮，选择Manage Settings按钮可以重新配置该应用，如图14-21所示。

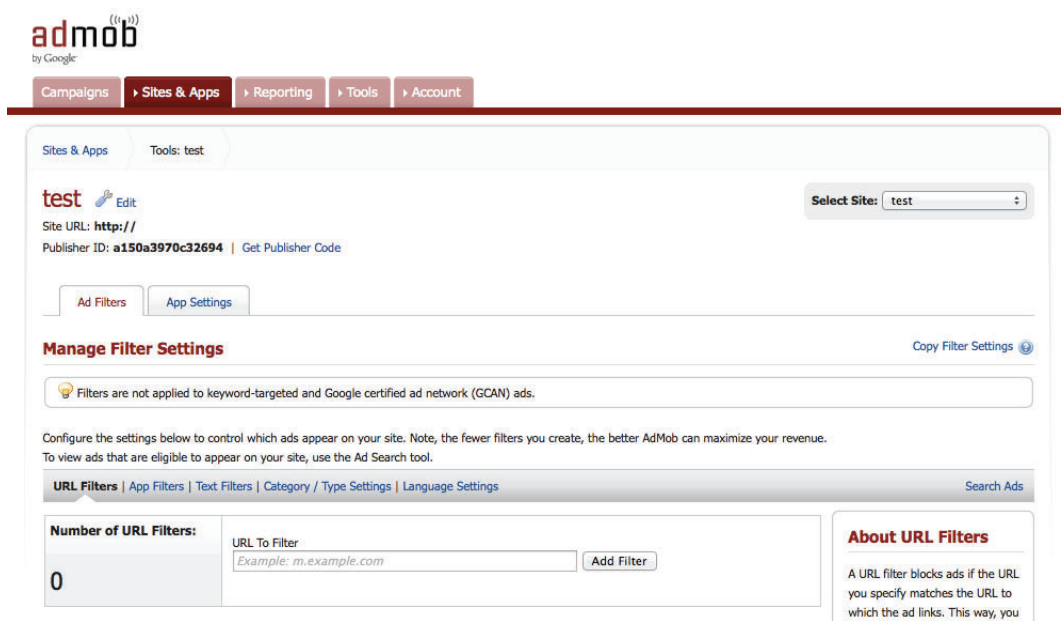


图14-21 管理应用信息页面

Publisher ID（发布者ID）是非常重要的信息，在我们的程序代码中通过它请求广告，我们需要把它硬编码到应用中。

14.3.2 下载谷歌 AdMob Ads SDK 和示例代码

我们可以在图14-19所示的页面中点击Download AdMob iOS SDK按钮下载谷歌AdMob Ads SDK，也可以直接在浏览器中输入<https://developers.google.com/mobile-ads-sdk/download>进入下载页面，然后点击GoogleAdMobAdsSdkiOS.zip下载。目前的SDK版本是6.2.1，如果是iOS 6，则使用IDFA或IFA（Identifier for Advertisers，广告主识别码）进行广告跟踪，iOS 5以及之前的版本使用UUID（Universally Unique Identifier，通用唯一识别码）进行广告跟踪。广告主根据跟踪的结果分析用户消费习惯，从而推送他关心的广告信息。

提示 出于安全考虑，在iOS 6中，苹果禁止访问设备的UUID，这样广告主就无法跟踪用户，而苹果推出了替代技术IDFA。IDFA是临时的标识，广告主可以利用它来跟踪广告，用户可以通过设置“应用”→“通用”→“关于本机”→“广告”，在设备上设置禁止或开启IDFA访问，如图14-22所示。

广告提供了AdMob的示例代码，大家可以到<https://code.google.com/p/google-mobile-dev/downloads/list>上下载。针对于iOS平台的有BannerExample_iOS_2.6.zip和InterstitialExample_iOS_2.5.zip压缩文件，其中BannerExample_iOS示例用于介绍横幅广告，InterstitialExample_iOS示例用于介绍插页广告。

使用示例代码时，代码工程中并不包含AdMob的静态库和.h文件，需要我们自己导入到工程中。如图14-23所示，Google AdMob Ads SDK处于没有导入的状态。



图14-22 设置广告跟踪

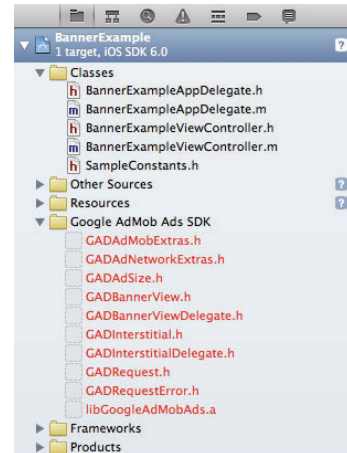


图14-23 AdMob静态库和.h文件没有导入到工程

在工程中删除Google AdMob Ads SDK组，然后重新导入。选择AdMob解压目录中的9个.h文件和一个静态库文件libGoogleAdMobAds.a，如图14-24所示。

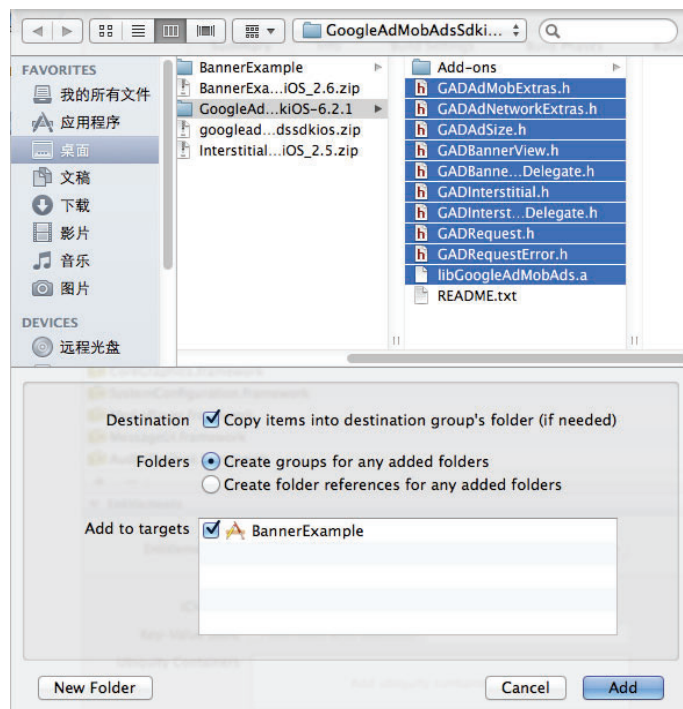


图14-24 导入AdMob静态库和.h文件到工程

导入完成之后，可以编译一下，试一试能否编译通过。编译成功后，如果还不能运行，需要修改SampleConstants.h文件。在这个文件中，只有一条语句，它定义了一个宏kSampleAdUnitID，是AdMob中添加的应用的发布者ID，将其修改为自己的发布者ID才能运行：

```
#define kSampleAdUnitID @"INSERT_YOUR_ADMOB_ID_HERE" //例如: @"a14df1974738141"
```

14.3.3 添加 AdMob 横幅广告

AdMob横幅广告与iAd横幅广告的概念一样，出现在屏幕中某一位置，占有部分空间，点击广告进入广告详细信息页面。

AdMob也规定了不同的横幅广告尺寸，如表14-1所述。

表14-1 AdMob横幅广告尺寸

尺 寸	适用范围	说 明
320×50	iPhone、iPod touch和iPad	标准横幅广告，使用常量kGADAdSizeBanner定义
300×250	iPad	IAB（Interactive Advertising Bureau，美国互动广告局）规定的矩形广告尺寸，使用常量kGADAdSizeMediumRectangle定义
468×60	iPad	IAB规定的全尺寸横幅广告尺寸，使用常量kGADAdSizeFullBanner定义
728×90	iPad	IAB规定的页首横幅尺寸，使用常量kGADAdSizeLeaderboard定义

横幅广告栏由GADBannerView类定义。要把GADBannerView添加到我们工程中，只能采用编码方式动态设定GADBannerView的位置。使用AdMob时，需要引入一些框架：StoreKit、AudioToolbox、MessageUI、SystemConfiguration、CoreGraphics和AdSupport等。

引入框架后，还需要设置编译参数Other Linker Flags，把它的Debug和Release参数都设置为-ObjC，如图14-25所示。

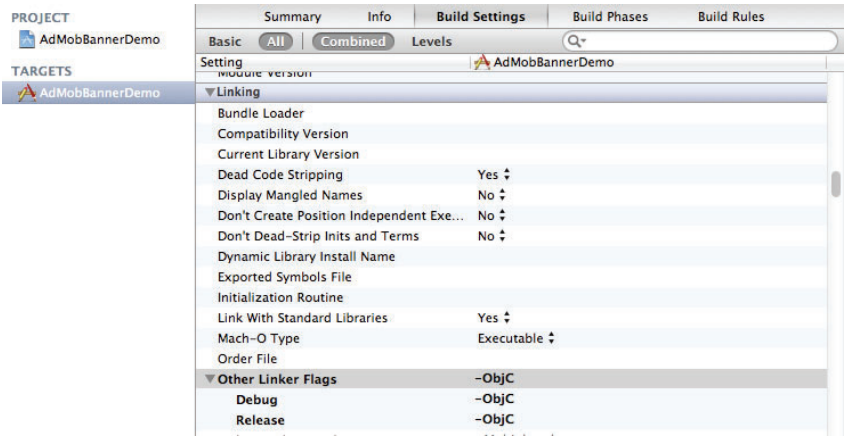


图14-25 设置编译参数Other Linker Flags

下面我们将14.2.1节的iAd横幅广告栏案例换成AdMob。下面的代码是案例工程AdMobBannerDemo中的ViewController.h部分：

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

#define kSampleAdUnitID @"a14df1974738141"

@interface ViewController : UIViewController <UIScrollViewDelegate,
    GADBannerViewDelegate>

@property (strong, nonatomic) UIImageView *page1;
@property (strong, nonatomic) UIImageView *page2;
@property (strong, nonatomic) UIImageView *page3;
@property (strong, nonatomic) UIImageView *page4;
```

```

@property (weak, nonatomic) IBOutlet UIScrollView *scrollView;
@property (weak, nonatomic) IBOutlet UIPageControl *pageControl;
@property (strong, nonatomic) GADBannerView *bannerView;

- (IBAction)changePage:(id)sender;

- (GADRequest *)createRequest;

@end

```

其中需要引入GADBannerView.h头文件，还要实现GADBannerViewDelegate协议，该协议规定了GADBannerView接收成功和接收失败的方法。我们还定义了GADBannerView *的属性bannerView。注意，属性参数strong是强引用类型。

下面我们再看看实现代码ViewController.m中视图控制器初始化的相关代码：

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.pageControl.numberOfPages = 4;

    self.scrollView.frame = self.view.frame;

    self.scrollView.delegate = self;

    self.scrollView.contentSize = CGSizeMake(4 * 320.0f, 420.0f);

    self.page1 = [[UIImageView alloc] initWithImage:[UIImage
        imageNamed:@"animal-1.png"]];
    self.page1.frame = CGRectMake(0.0f, 0.0f, 320.0f, 420.0f);
    [self.scrollView addSubview:self.page1];

    self.page2 = [[UIImageView alloc] initWithImage:[UIImage
        imageNamed:@"animal-2.png"]];
    self.page2.frame = CGRectMake(320.0f, 0.0f, 320.0f, 420.0f);
    [self.scrollView addSubview:self.page2];

    self.page3 = [[UIImageView alloc] initWithImage:[UIImage
        imageNamed:@"animal-3.png"]];
    self.page3.frame = CGRectMake(320.0f * 2, 0.0f, 320.0f, 420.0f);
    [self.scrollView addSubview:self.page3];

    self.page4 = [[UIImageView alloc] initWithImage:[UIImage
        imageNamed:@"animal-4.png"]];
    self.page4.frame = CGRectMake(320.0f * 3, 0.0f, 320.0f, 420.0f);
    [self.scrollView addSubview:self.page4];

    //设定广告栏屏幕尺寸，实例化GADBannerView
    self.bannerView = [[GADBannerView alloc] initWithAdSize:kGADAdSizeBanner]; ①

    //设置应用发布者ID
    self.bannerView.adUnitID = kSampleAdUnitID; ②
    //设置委托
    self.bannerView.delegate = self;
    //设置广告栏的根视图控制器
    [self.bannerView setRootViewController:self];

    [self.view addSubview:self.bannerView];
    //请求加载广告
    [self.bannerView loadRequest:[self createRequest]]; ③
}
//创建广告请求

```

```

- (GADRequest *)createRequest {                                ④
    GADRequest *request = [GADRequest request];                ⑤
#ifdef DEBUG
    //设置测试设备, 防止测试阶段的无效请求
    request.testDevices = [NSArray arrayWithObjects:
        @"5F6A1B6D-6283-4BDF-A368-1EAA17F5DE78", nil];        ⑥
#endif
    return request;
}

- (void)viewDidLoadSubviews                                  ⑦
{
    CGRect contentFrame = self.view.bounds;
    if (contentFrame.size.width < contentFrame.size.height) {    ⑧
        //竖屏情况下设置广告栏的位置
        self.bannerView.center = CGPointMake(self.view.center.x,
            kGADAdSizeBanner.size.height / 2);                    ⑨
    } else {
        //横屏情况下设置广告栏的位置
        self.bannerView.center = CGPointMake(contentFrame.size.width / 2 ,
            kGADAdSizeBanner.size.height / 2);                    ⑩
    }
}
}

```

在上面的代码中, viewDidLoad方法在视图控制器加载时调用。在这个方法中, 我们需要实例化GADBannerView。如第①行代码所示, 我们使用构造方法initWithAdSize:参数kGADAdSizeBanner描述320×50大小的横幅广告栏常量。GADBannerView还有一个构造方法initWithAdSize:(GADAdSize)size origin:(CGPoint)origin, 该方法的origin:部分用于指定广告栏的位置。

在第②行代码中, GADBannerView的adUnitID属性表示应用发布者ID。第③行代码发送loadRequest:消息请求广告, 其中又调用自定义方法createRequest创建请求。

第④行代码定义了createRequest方法, 其返回值类型是GADRequest *。在该方法中, 第⑤行代码中的[GADRequest request]发出广告请求, 其中request方法是类方法。在AdMob中, 广告请求比较简单。第⑥行的request.testDevices语句被放置在条件编译语句#ifdef DEBUG...#endif中, 它只有在调试模式下起作用, 指定测试模拟器或设备ID。谷歌不希望接收测试阶段的广告请求。使用测试模式还有一个优点, 那就是当广告无法请求获得时, 谷歌能够保证提供测试广告返回给我们的设备端。5F6A1B6D-6283-4BDF-A368-1EAA17F5DE78是我当前模拟器的测试设备ID。

第⑦行代码中的viewDidLoadSubviews方法在重新加载子视图时调用。在该方法中, 第⑨行和第⑩行代码用于设置广告栏的位置, kGADAdSizeBanner.size.height可以取得kGADAdSizeBanner常量的高度(即50点)。如果想获得宽度, 可以使用kGADAdSizeBanner.size.width。

提示 测试设备ID在iOS 6之前获得的是UUID, 而iOS 6之后是IDFA, 获得UUID的方法是[[UIDevice currentDevice] uniqueIdentifier], 获得IDFA的方法是[[ASIdentifierManager sharedManager] advertisingIdentifier] UUIDString]。我们可以在应用程序启动时调用AppDelegate的application:didFinishLaunchingWithOptions:方法。在该方法中, 我们添加如下粗体显示的代码, 这样我们就可以在启动时获得测试设备ID, 然后再把它设置到request.testDevices:

```

#import "AppDelegate.h"
#import <AdSupport/ASIdentifierManager.h
@implementation AppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
    (NSDictionary *)launchOptions {
#ifdef DEBUG

```

```

//在iOS 6中, 输出IDFA; 如果iOS版本号小于6, 则输出UDID
if (NSClassFromString(@"ASIdentifierManager")) {
    NSLog(@"GoogleAdMobAdsSDK ID for testing: %@",
        [[[ASIdentifierManager sharedManager]
        advertisingIdentifier] UUIDString]);
} else {
    NSLog(@"GoogleAdMobAdsSDK ID for testing: %@",
        [[UIDevice currentDevice] uniqueIdentifier]);
} #endif
return YES;
}
.....
@end

```

下面我们再看看ViewController.m中有关GADBannerViewDelegate委托的实现代码:

```

//广告接收成功
- (void)adViewDidReceiveAd:(GADBannerView *)adView {
    NSLog(@"广告接收成功");
}
//广告接收失败
- (void)adView:(GADBannerView *)view didFailToReceiveAdWithError:(GADRequestError *)error {
    NSLog(@"广告接收失败 %@", [error localizedFailureReason]);
    //重新请求加载广告
    [self.bannerView loadRequest:[self createRequest]];
}

```

adViewDidReceiveAd:方法在广告接收成功时回调, 我们可以在该方法中开启一个计时器, 控制广告栏的显示方式和时间。

adView:didFailToReceiveAdWithError:方法在广告接收失败时回调。我们可以在广告接收失败的情况下重新发送请求, 也可以选择加载别的广告。一个应用可以根据不同的情况、不同的地点、不同的时段动态选择不同的广告展示, 这样才能做出一个好的应用。

运行一下看看效果, 广告请求成功后会看到如图14-26所示的广告横幅, 点击广告栏即可启动广告详细内容画面。

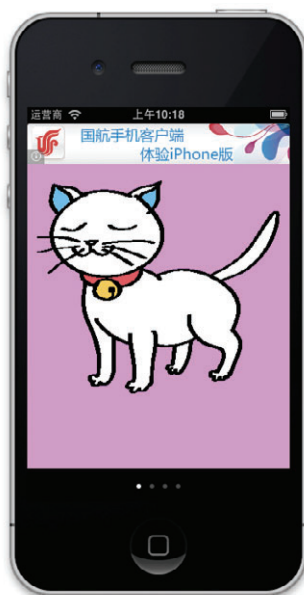


图14-26 AdMob横幅广告画面

14.3.4 添加 AdMob 插页广告

AdMob插页广告与苹果iAd插页广告比较相似，都是全屏显示，它的应用场景与iAd稍有不同。在应用启动、视频前贴片或游戏关卡加载时显示广告，我们把这种场景称为“启动场景”，这与iAd的“内容显示场景”类似。还有一种是在视频播放结束或游戏结束时显示的，我们称之为“结束场景”。

1. 启动场景

应用启动、视频前贴片或游戏关卡加载时，会弹出模态全屏广告对话框，点击全屏广告左上角的关闭按钮，可以关闭该对话框，如图14-27所示。没有广告时，直接从①界面（启动界面）进入③界面（应用主界面），有广告时，则在它们之间插入插页广告。

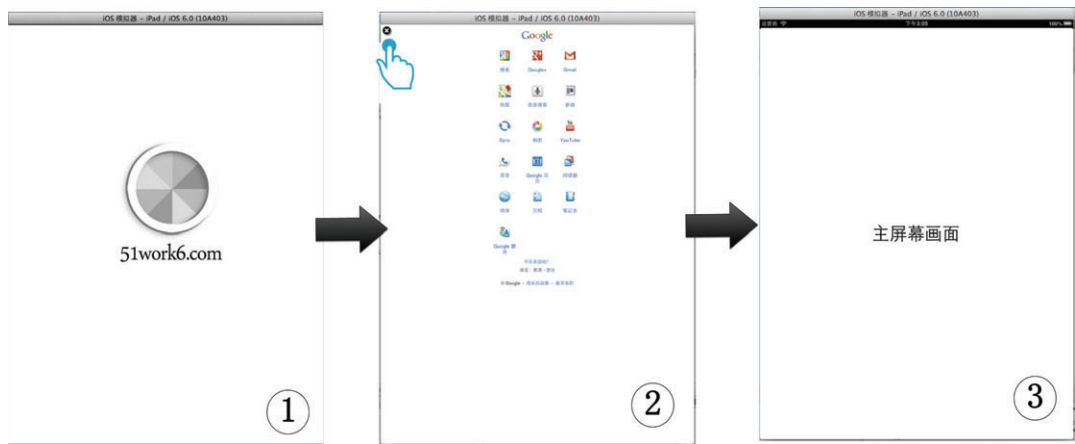


图14-27 启动场景的插页广告

下面我们通过案例AdMobFullScreen1Demo介绍一下这种场景下插页广告的实现。创建工程后，需要将AdMob需要的.h文件和静态库文件导入到工程中，并添加所需的框架，具体请参考14.3.2节的内容。

下面的代码是AdMobFullScreen1Demo中应用程序委托对象的AppDelegate.h部分：

```
#import <UIKit/UIKit.h>
#import "GADInterstitial.h"
#import <AdSupport/ASIdentifierManager.h>

#define INTERSTITIAL_AD_UNIT_ID @"a14df1974738141"

@interface AppDelegate : UIResponder <UIApplicationDelegate,GADInterstitialDelegate>

@property (strong, nonatomic) UIWindow *window;

@property(nonatomic, strong) GADInterstitial *splashInterstitial;

- (GADRequest *)createRequest;

@end
```

这里我们引入头文件GADInterstitial.h是为了使用AdMob插页广告类GADInterstitial，引入头文件AdSupport/ASIdentifierManager.h是为了使用ASIdentifierManager类,该类是iOS 6 SDK添加的,用于获得IDFA。在定义应用程序委托对象AppDelegate时，声明实现GADInterstitialDelegate委托协议，该协议规定了GADInterstitial生命周期事件。此外，我们还定义了GADInterstitialAd *的属性splashInterstitial，注意属性参数strong是强引用类型。

下面我们再看看AppDelegate.m中的主要代码：


```

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
(NSDictionary *)launchOptions
{
#ifdef DEBUG
    //在iOS 6中输出IDFA; 如果iOS版本号小于6, 则输出UDID
    if (NSClassFromString(@"ASIdentifierManager")) {
        NSLog(@"GoogleAdMobAdsSDK ID for testing: %@",
            [[[ASIdentifierManager sharedManager]
            advertisingIdentifier] UUIDString]);
    } else {
        NSLog(@"GoogleAdMobAdsSDK ID for testing: %@",
            [[UIDevice currentDevice] uniqueIdentifier]);
    }
#endif

    self.splashInterstitial = [[GADInterstitial alloc] init];           ①

    self.splashInterstitial.adUnitID = INTERSTITIAL_AD_UNIT_ID;       ②
    self.splashInterstitial.delegate = self;                           ③

    [self.splashInterstitial loadAndDisplayRequest:[self createRequest]
        usingWindow:self.window
        initialImage:[UIImage imageNamed:@"Splash.png"]];           ④

    return YES;
}

#pragma mark GADRequest generation
//创建广告请求
- (GADRequest *)createRequest {
    GADRequest *request = [GADRequest request];

#ifdef DEBUG
    //设置测试设备, 防止测试阶段的无效请求
    request.testDevices = [NSArray arrayWithObjects:
        @"5F6A1B6D-6283-4BDF-A368-1EAA17F5DE78", nil];
#endif

    return request;
}

#pragma mark GADInterstitialDelegate实现
- (void)interstitial:(GADInterstitial *)interstitial didFailToReceiveAdWithError:
(GADRequestError *)error {
    NSLog(@"广告接收失败 %@", [error localizedDescription]);
}

```

在上述代码中, application:didFinishLaunchingWithOptions:在应用程序启动时调用。第①行代码实例化GADInterstitial对象, 第②行代码设置GADInterstitial对象的adUnitID属性, 该属性指定应用发布者ID。在第④行代码中, GADInterstitial对象调用loadAndDisplayRequest:usingWindow:initialImage:方法加载并显示全屏广告界面, 请求开始后界面一直显示的是initialImage:部分指定的初始化界面, 直到请求成功或失败。请求成功后, 会模态呈现全屏广告。在该方法中, 请求对象是用createRequest方法获得的。

interstitial:didFailToReceiveAdWithError:方法在接收广告失败时调用, 它是GADInterstitial-Delegate委托协议定义的方法之一。

在这种场景下, 需要在应用程序委托对象中完成代码编写, 而不需要主视图控制器ViewController部分。可以运行一下, 看看效果。

2. 结束场景

该场景是在视频播放结束或游戏结束时显示广告, 它需要有一个触发条件, 满足条件时弹出模态全屏广告对话框, 如图14-28所示。



图14-28 结束场景的插页广告

下面我们通过案例 AdMobFullScreen2Demo 介绍一下这种场景下插页广告的实现。下面是 AdMobFullScreen2Demo 实现代码的 ViewController.h 部分：

```
#import <UIKit/UIKit.h>
#import "GADInterstitial.h"
#import <AdSupport/ASIdentifierManager.h>
#define INTERSTITIAL_AD_UNIT_ID @"a14df1974738141"
@interface ViewController : UIViewController <GADInterstitialDelegate>
{
    NSTimer *timer;          //用于模拟控制游戏进度
}
//进度条
@property (weak, nonatomic) IBOutlet UIProgressView *progressView;
//插页广告GADInterstitial对象属性
@property (nonatomic, strong) GADInterstitial *interstitial;
//界面中的按钮
@property (weak, nonatomic) IBOutlet UIButton *startButton;
//创建广告请求对象
- (GADRequest *)createRequest;
//更新进度条
- (void)update;
//按钮事件
- (IBAction)start:(id)sender;
@end
```

下面我们再看看实现代码 ViewController.m 中按钮事件的 start: 方法和更新进度条的 update 方法的代码：

```
- (IBAction)start:(id)sender {
    self.startButton.enabled = NO;

    timer = [NSTimer scheduledTimerWithTimeInterval:1.0
                                                target:self
                                                selector:@selector(update)
                                                userInfo:nil
                                                repeats:YES];
}

- (void)update
{
    self.progressView.progress += 0.1;
}
```

```

        if (self.progressView.progress == 1.0) {
            //游戏结束
            NSLog(@"游戏结束");
            [timer invalidate];
            timer = nil;

            //初始化广告
            self.interstitial = [[GADInterstitial alloc] init];

            self.interstitial.adUnitID = INTERSTITIAL_AD_UNIT_ID;
            self.interstitial.delegate = self;
            [self.interstitial loadRequest:[self createRequest]];
        }
    }

```

在start:方法中,第①行代码开始一个NSTimer计划执行,每隔0.1秒调用一次update方法。在update方法中,第②行代码用于判断游戏是否结束(当然这是模拟),NSTimer使用完,就需要使用[timer invalidate]语句停止计划执行。第③行代码用于实例化GADInterstitial对象。第⑥行代码通过调用createRequest获得请求对象发出广告请求。

createRequest方法的代码如下:

```

- (GADRequest *)createRequest {
    GADRequest *request = [GADRequest request];

#ifdef DEBUG
    //设置测试设备,防止测试阶段的无效请求
    request.testDevices = [NSArray arrayWithObjects: @"5F6A1B6D-6283-4BDF-A368-1EAA17F5DE78", nil];
#endif

    return request;
}

```

下面我们再看看ViewController.m中有关GADInterstitialDelegate委托的实现代码:

```

#pragma mark GADInterstitialDelegate实现
- (void)interstitial:(GADInterstitial *)interstitial
didFailToReceiveAdWithError:(GADRequestError *)error {
    NSLog(@"广告接收失败 %@", [error localizedDescription]);
}

- (void)interstitialDidReceiveAd:(GADInterstitial *)interstitial {
    NSLog(@"广告接收成功");
    [self.interstitial presentFromRootViewController:self];
    self.startButton.enabled = YES;
    self.progressView.progress = 0.0;
}

```

在接收成功的interstitialDidReceiveAd:方法中,需要使用[self.interstitial presentFromRootViewController:self]语句模态呈现广告对话框,GADInterstitial对象的presentFromRootViewController:方法需要在成功请求回来后再调用。运行一下,看看是否能呈现广告。

14.3.5 为广告提交用户和位置信息

如果广告主能够获得用户信息或位置信息,那么展示给用户的广告会更有针对性。出于对用户隐私的尊重,谷歌要求只能指定以下信息:用户性别、用户生日和位置等。

修改AdMobBannerDemo工程中ViewController.m的createRequest代码,具体如下:

```

- (GADRequest *)createRequest {
    GADRequest *request = [GADRequest request];

    //设定性别
    request.gender = kGADGenderMale;
}

```

```

//设置出生年月日
[request setBirthdayWithMonth:12 day:7 year:1973];           ②
//设置位置信息
[request setLocationWithLatitude:39.904667                  ③
                      longitude:116.408198
                      accuracy:1000];

//设置位置描述信息
[request setLocationWithDescription:@"美食城"];             ④

#ifdef DEBUG
//设置测试设备,防止测试阶段的无效请求
request.testDevices = [NSArray arrayWithObjects:
                      @"5F6A1B6D-6283-4BDF-A368-1EAA17F5DE78", nil];

#endif

return request;
}

```

在上述代码中,第①行代码用于设定用户的性别,其中kGADGenderMale是男性常量,类似还有kGADGenderMale女性常量和kGADGenderUnknown未知常量。第②行代码用于设置用户的出生信息。

第③行代码用于设置用户位置信息,其中setLocationWithLatitude:部分用于设置纬度,如39.904667,longitude:部分是经度,如116.408198,这个经纬度是北京地理坐标,accuracy:部分是水平精度。如果我们得不到经纬度地理坐标,也可以采用地理信息描述方式查询位置,第④行的setLocationWithDescription:方法就可以实现这个目的。

14.3.6 搜索广告

AdMob还提供了一些插件来编写更加准确的广告应用,搜索广告就是其中一个。搜索广告用于访问Google AdWords广告资源,帮助用户查找他们所需的内容。

搜索广告使用GADSearchBannerView替代GADBannerView,使用GADSearchBannerView替代GADBannerView。

使用搜索广告,首先导入Add-ons目录中的Search、DoubleClick和Mediation子目录。注意导入采用组,如图14-29所示,在Folders中选中Create groups for any added folders单选按钮,这可以使原来目录中的子目录变成工程中的“组”。

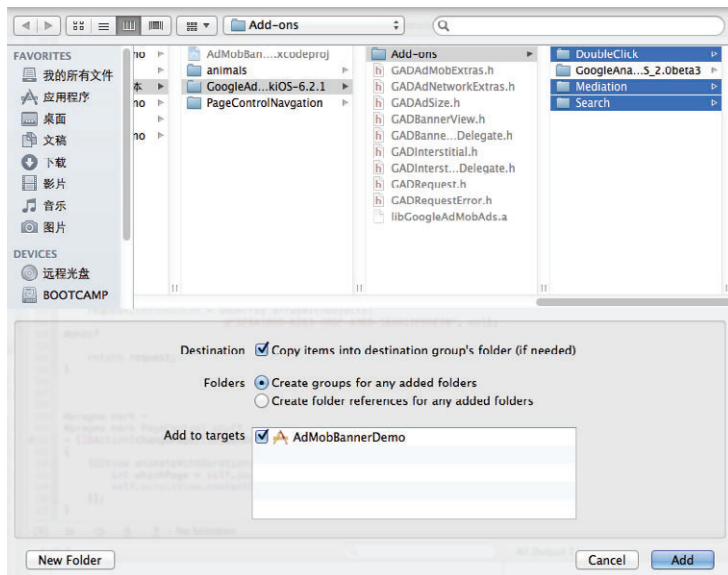


图14-29 导入Add-ons目录

下面我们再看看代码部分，将AdMobBannerDemo工程中ViewController.h的代码修改如下：

```
#import <UIKit/UIKit.h>
#import "GADSearchBannerView.h" ①
#import "GADSearchRequest.h" ②

#define kSampleAdUnitID @"a14df1974738141"

@interface ViewController : UIViewController <UIScrollViewDelegate, GADBannerViewDelegate>

@property (strong, nonatomic) UIImageView *page1;
@property (strong, nonatomic) UIImageView *page2;
@property (strong, nonatomic) UIImageView *page3;
@property (strong, nonatomic) UIImageView *page4;

@property (weak, nonatomic) IBOutlet UIScrollView *scrollView;
@property (weak, nonatomic) IBOutlet UIPageControl *pageControl;
@property (strong, nonatomic) GADSearchBannerView *bannerView; ③

- (IBAction)changePage:(id)sender;

- (GADRequest *)createRequest;

@end
```

在上述代码中，我们引入的GADSearchBannerView.h和GADSearchRequest.h文件与原来相比有些差别。我们需要注意第③行中属性bannerView的类型为GADSearchBannerView *而不是GADBannerView *。

ViewController.m中主要改变的代码如下：

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

    ...

    //设定广告栏屏幕尺寸，实例化GADSearchBannerView
    self.bannerView = [[GADSearchBannerView alloc] initWithAdSize:kGADAdSizeBanner]; ①
    //设置应用发布者ID
    self.bannerView.adUnitID = kSampleAdUnitID;
    //设置委托
    self.bannerView.delegate = self;
    //设置广告栏的根视图控制器
    [self.bannerView setRootViewController:self];

    [self.view addSubview:self.bannerView];
    //请求加载广告
    [self.bannerView loadRequest:[self createRequest]];
}

//创建广告请求
- (GADRequest *)createRequest {

    GADSearchRequest *adRequest = [[GADSearchRequest alloc] init]; ②

    [adRequest setQuery:@"学习英语"]; ③

    GADRequest *request = [adRequest request]; ④

#ifdef DEBUG
    //设置测试设备，防止测试阶段的无效请求
    request.testDevices = [NSArray arrayWithObjects:
        @"5F6A1B6D-6283-4BDF-A368-1EAA17F5DE78", nil];
#endif
}
```

```
#endif  
  
    return request;  
}
```

在viewDidLoad方法中，第①行代码实例化GADSearchBannerView对象，而不是GADBannerView对象。

在createRequest方法中，第②行代码实例化GADSearchRequest请求对象，我们可以通过这个对象的setQuery:方法设置搜索关键词。第④行代码用于获得GADRequest对象，这是通过GADSearchRequest的类方法request返回的。

下面我们看看运行效果，如图14-30所示，我们搜索到一个有关“学习英语”的广告。

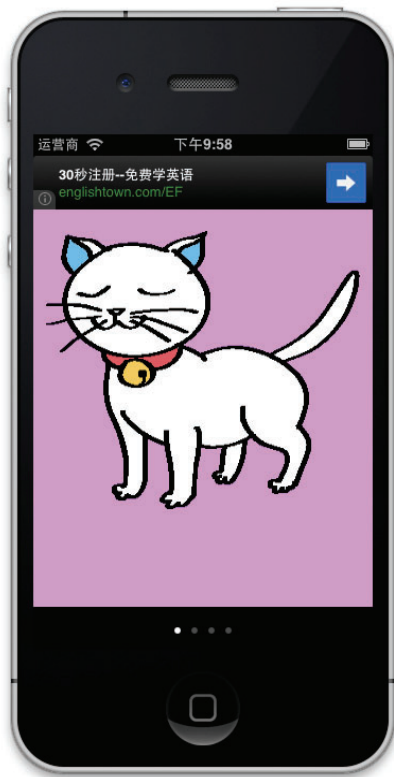


图14-30 搜索广告运行结果

14.3.7 查看你的收入

说了这么多，查看收入可能是我们最关心的事情。登录成功后，选择“站点和应用程序”标签进入，可以管理我们的应用，当然也可以查看应用的收入情况，如图14-31所示。

“当日收入”是当天的收入情况；“估算收入”是一个时间段的收入情况，这个收入可能需要扣除谷歌认为的一些无效展示和点击；“盈利趋势”是一个柱状图表，可以很清楚地看到收入走向。

图14-31最下面的内容可以管理应用，查看应用的收入点击情况。广告投放率是iAd的填充率，如果广告投放率低，说明广告展示时间太短，或者点击得太少。RPM是每千次交互活动的收入，如果只有广告收入，则 $RPM = eCPM \times \text{广告投放率}$ 。



图14-31 查看AdMob收入

14.4 应用内购买

应用内购买（In-App Purchase）是另外一种收费策略，很适合游戏、杂志期刊类的应用。这种收费策略有点像买水果时的先尝后买，游戏的前几个关卡免费，当你觉得有意思需要用到后面的关卡时则需要付费。一般购买内容包括内容类型、功能扩展、服务和订阅等。

14.4.1 概述

苹果为应用内购买提供了开发API——Store Kit，它为开发人员省去了很多麻烦，也为应用发布者提供了便利的收费途径和运营方式。

苹果在App Store上提供的应用内购买产品类型有如下3种。

- ❑ 消耗型。产品购买之后即被消费，再次购买该产品时还需要支付，只能应用于当前设备。
- ❑ 非消耗型。该产品一旦购买可以一直使用，而且可以在与该用户账号关联的多个设备上使用。App Store会保留用户的购买记录。
- ❑ 订阅型。订阅类产品在订阅周期内如同非消费型购买一样，在订阅期过后如消费型购买一样。作为开发者，需要确保用户订阅的内容在其iTunes同步的设备上都有效，可以在程序内部加入自己的订阅计划更新机制。苹果期望订阅类产品可以通过外部服务器交付。另外，订阅类产品可以在与该用户账号关联的多个设备上使用。订阅类产品又可以细分为：自动再生订阅类、非自动再生订阅类和免费订阅类。

事实上，Store Kit只是帮助收集付款信息，其他的工作需要开发人员自己设计和实现，如应用上的UI布局和产品交付模式等。产品交付模式非常重要，它与产品的类型有很大的关系，不同产品交付模式的管理也是不同的。苹果提供如下两种交付模式供开发者参考。

- ❑ 内置产品类型。需要交付的产品已经在程序内部，通常用于一些功能的锁定，这些功能原本是在程序中，但是需要购买这些功能才能解锁，开发人员需要记录这些购买记录，并且能够备份和恢复这些信息。它的优点是能很快交付产品给客户。大多数的内置产品应用为非消耗型产品。这种模式是我们本书重点介绍的模式。
- ❑ 服务器产品类型。在这种模式下，需要开发商或运行商提供另外的服务器，将要交付的内容、服务和订阅的产品更新到服务器上。应用程序与服务器和App Store交互获取信息。这种方式非常灵活，但是投入比较大，适合于订阅、内容和服务类产品。

下面我们通过如图14-32所示的案例来介绍一下应用内购买的管理、程序编写和测试，应用从App Store上请求要购买的产品列表，点击每一行后面的“购买”按钮，可以购买该产品。

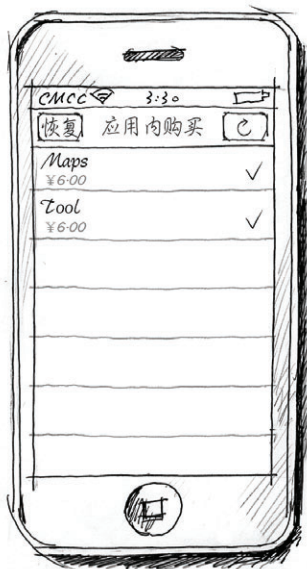


图14-32 应用内购买案例设计原型草图

14.4.2 测试环境搭建

搭建应用内购买测试环境比较麻烦,需要跨两个工具管理:iTunes Connect和iOS开发中心的配置门户网站(iOS Provisioning Portal)。iTunes Connect的网址为[https://itunesconnect.apple.com/ WebObjects/iTunesConnect.woa](https://itunesconnect.apple.com/WebObjects/iTunesConnect.woa)，我们在14.2.3节中介绍过，它可以帮助我们发布应用、管理应用、查看收入、管理iAd广告和用户等。iOS开发中心的配置门户网站可以帮助我们管理测试设备、证书和App ID等信息，其网址为<https://developer.apple.com/ios/manage/overview/index.action>。

使用这些工具的测试环境搭建流程图如图14-33所示。

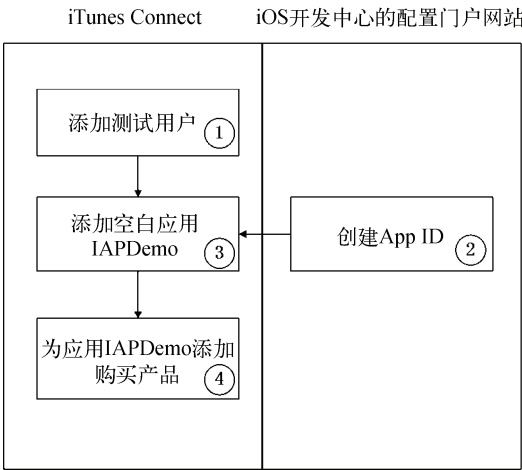


图14-33 应用内购买测试环境搭建流程图

1. 添加测试用户

测试具有应用内购买功能的应用需要使用iTunes Connect的测试用户，不能是iTunes Connect的真实用户，也不能是App Store的真实用户。当管理员成功登录iTunes Connect后，会看到如图14-34所示的界面。

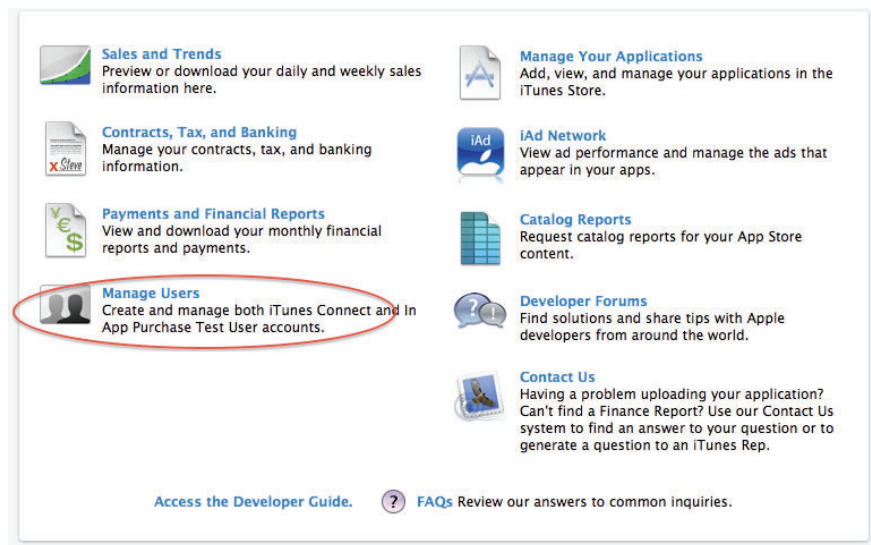


图14-34 成功登录iTunes Connect

点击Manager Users进入用户管理界面，如图14-35所示，我们看到用户类型有iTunes Connect User和Test User，这里我们选择Test User。

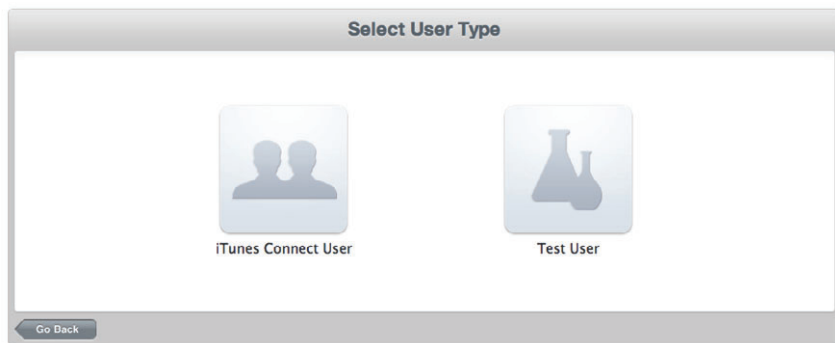


图14-35 选择用户类型

然后进入创建测试用户界面，详细的创建过程不再介绍。

2. 创建App ID

为了调试和发布的需要，每个应用都有唯一的App ID。App ID是在iOS开发中心的配置门户网站创建的，具体的创建过程可参见15.5.3节。如图14-36所示，创建App ID的包标识符（Bundle Identifier）是com.51work6.IAPDemo。

包标识符非常重要，它必须与该工程的包标识符一致。使用Xcode打开工程，可以发现Bundle Identifier也是com.51work6.IAPDemo，如图14-37所示。

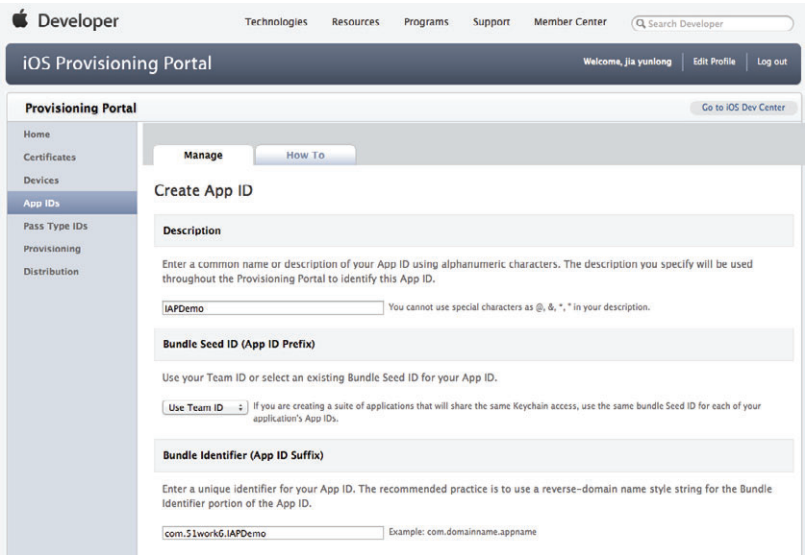


图14-36 创建App ID

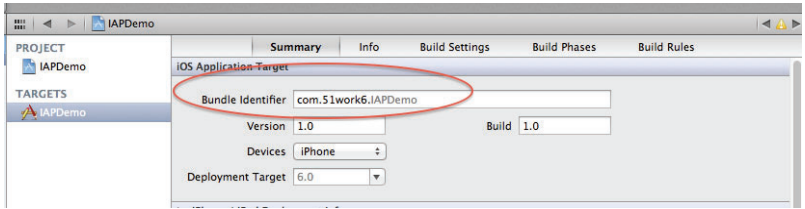


图14-37 在Xcode中查看包标识符

3. 添加空白应用IAPDemo

我们在做别的应用测试时不需要使用iTunes Connect，更不需要使用它添加空白的应用，这个操作一般是在应用发布时（相关内容可参见19.2节）使用。

图14-38是我们要创建的应用信息，注意其中的Bundle ID是我们在iOS开发中心配置门户网站创建的App ID。

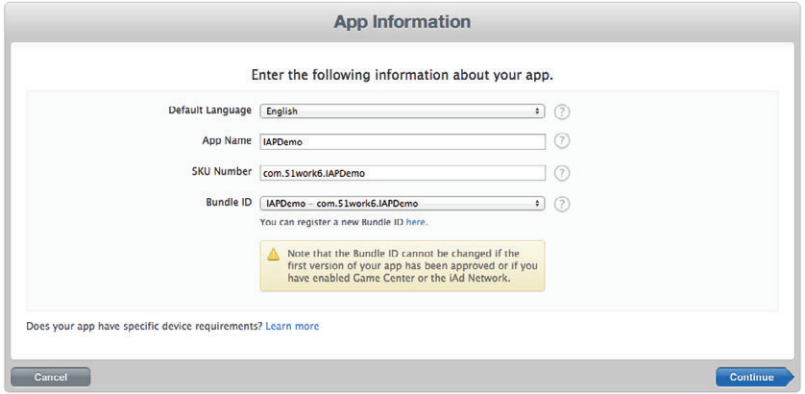


图14-38 在iTunes Connect中添加应用

4. 为IAPDemo应用添加购买产品

此外，我们还需要在iTunes Connect中为应用IAPDemo添加购买产品。在iTunes Connect中选择Manage Your Applications→com.51work6.IAPDemo，进入图14-39所示的管理界面。

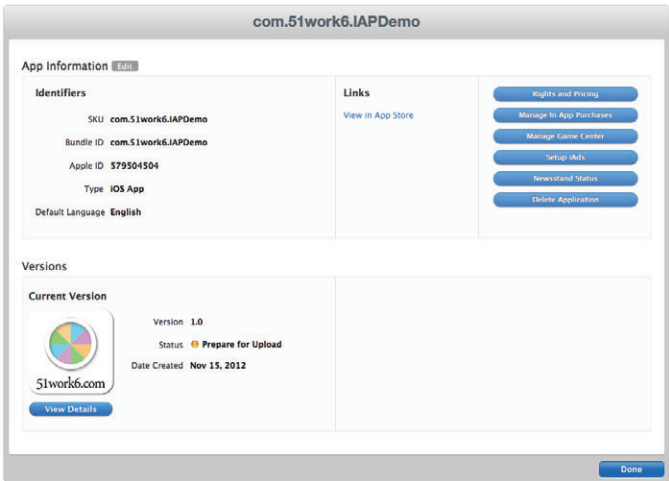


图14-39 在iTunes Connect中添加购买产品

然后点击右边的Manage In App Purchases按钮，进入图14-40所示的应用内购买产品管理界面。

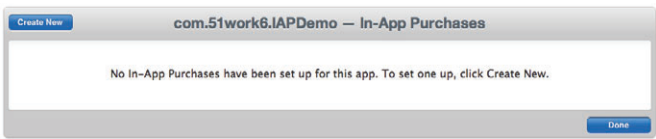


图14-40 应用内购买产品管理界面

接着点击左上角的Create New按钮，进行图14-41所示的选择产品类型界面。

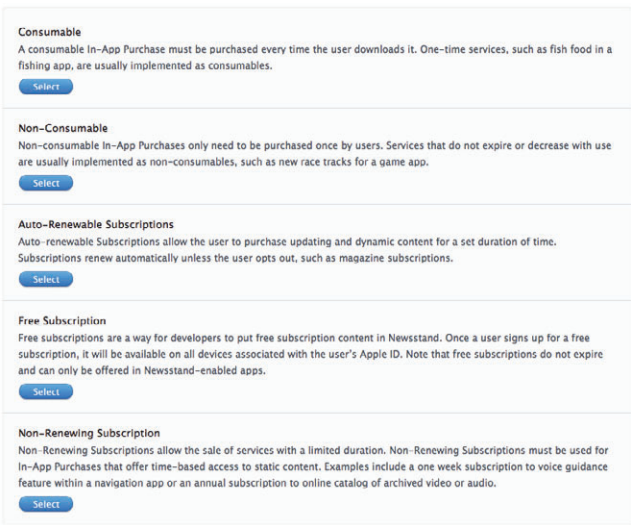


图14-41 选择应用内购买产品的类型

这里我们选择的是Non-Consumable（非消耗型产品），然后进入应用内购买信息输入界面，如图14-42所示。

在图14-42中，Reference Name是显示在iTunes Connect里的名字，这个名字在应用里是不可见的。Product ID是Product identifier（产品标识符），它具有唯一性，因此建议采用“包标识符+产品名”，我们这里输入的是com.51work6.IAPDemo.elves。将Cleared for Sale设定为YES状态时，这些产品就可以购买了。Price Tier是产品的价格。

com.51work6.IAPDemo — In-App Purchases

In-App Purchase Summary

Enter a reference name and a product ID for this In-App Purchase. You must also add at least one language, along with a display name and a description in that language.

Reference Name:

elves

?

Product ID:

com.51work6.IAPDemo.elves

?

Pricing and Availability

Enter the pricing and availability details for this In-App Purchase below.

Cleared for Sale

Yes

No

Price Tier

Tier 1

?

View Pricing Matrix

Price Tier 1

App Store	Customer Price	Your Proceeds
U.S.*	US \$0.99	US \$0.70

图14-42 应用内购买信息输入界面

把当前界面滚动到底部，会看到如图14-43所示的应用内购买产品详细界面。

In-App Purchase Details

Language

Details for this In-App Purchase are shown below. You must provide at least one language at all times.

Add Language

Language	Display Name	Description
Click Add Language to get started.		

Hosting Content with Apple

Select if you want Apple to host your In-App Purchase content. If you select yes, you must upload your content to Apple before sending the In-App Purchase for review.

Hosting Content with Apple

Yes

No

Review Notes (Optional)

Additional information about your In-App Purchase that can help us with our review, such as test accounts that can be used (including user names, passwords and so on). Review notes cannot exceed 4000 bytes.

Screenshot for Review

Before you submit your In-App Purchase for review, you must upload a screenshot. This screenshot will be for review purposes only. It will not be displayed on the App Store. Screenshots must be at least 640x920 pixels and at least 72 DPI.

Choose File

图14-43 添加应用内购买产品详细界面

图灵社区会员 叶清泉(qqyadf@126.com) 专享 尊重版权

点击Add Language按钮，弹出如图14-44所示的对话框，在这里可以输入客户端要显示的相关信息，其中Language是要显示的语言，Display Name是该语言下显示的产品名，Description是产品的描述。当然，我们可以根据需要添加多种语言。



图14-44 添加语言对话框

在图14-43所示的界面中单击Choose File按钮，可以上传产品预览图片，它并不会显示在客户端，只是审核时使用。但这里必须上传图片，并且图片的大小也有要求。输入完成后保存，此时回到图14-45所示的界面。可以发现，现在我们已经有了4个应用内购买产品了。

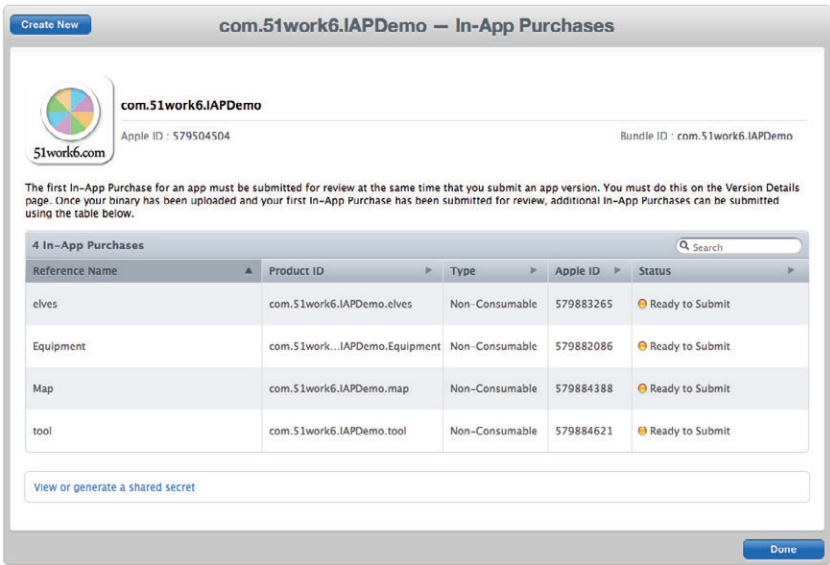


图14-45 添加产品完成界面

14.4.3 在程序中实现应用内购买

在程序中实现应用内购买需要4个步骤，下面我们就来介绍每一步的实现过程。

1. 创建工程和初始化处理

我们需要为应用创建一个工程，这里将其命名为IAPDemo。设置工程的Bundle Identifier为com.51work6.IAPDemo。然后，为工程添加必要的框架StoreKit.framework，它是应用内购买要求的必须框架。

打开故事板文件，添加一个导航控制器和表视图控制器，按照图14-46所示摆放控件，并连接输出和动作事件。

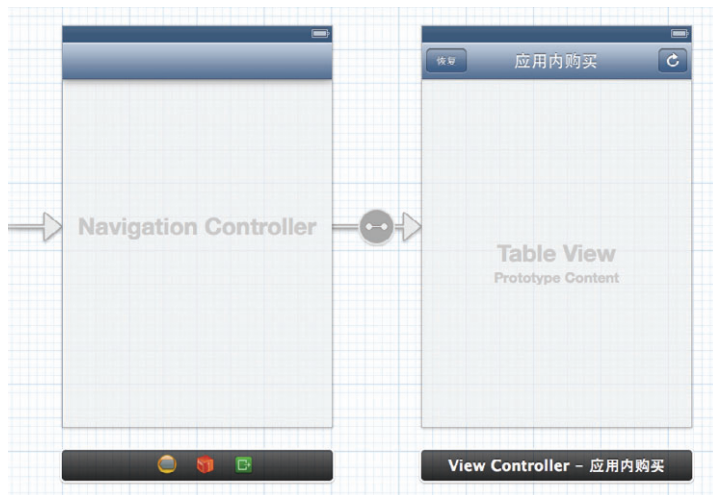


图14-46 IAPDemo设计画面

下面我们看看程序代码，其中ViewController.h中的代码如下：

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

@interface ViewController : UITableViewController
    <SKProductsRequestDelegate, SKPaymentTransactionObserver> ①

//点击刷新按钮
- (IBAction) request:(id)sender;
//点击恢复按钮
- (IBAction) restore:(id)sender;

//刷新按钮属性
@property (weak, nonatomic) IBOutlet UIBarButtonItem *refreshButton; ②
//恢复按钮属性
@property (weak, nonatomic) IBOutlet UIBarButtonItem *restoreButton; ③
//产品列表
@property (nonatomic, strong) NSArray* skProducts;
//数字格式
@property (nonatomic, strong) NSNumberFormatter * priceFormatter;
//产品标识集合
@property (nonatomic, strong) NSSet * productIdentifiers;

@end
```

在ViewController声明中，我们实现了SKProductsRequestDelegate和SKPaymentTransaction-Observer协议，其中前者是与SKProductsRequest对象对应的委托协议，后者是交易观察者协议，定义了观察交易结果的方法。

在第②行代码中，我们定义了刷新按钮的输出口属性refreshButton，需要在程序中改变它的状态。点击“刷新”按钮时，应用与iTunes Connect进行通信。在没有返回结果的情况下，不希望用户再点击“刷新”按钮。在程序中，我们把refreshButton属性设置为不可用状态，当结果返回之后再把refreshButton设置为可用状态。第③行的restoreButton属性也是这样设计的。

下面我们看看ViewController.m中viewDidLoad方法的代码，具体如下：

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

```

//设置数字格式
self.priceFormatter = [[NSNumberFormatter alloc] init];           ①
[self.priceFormatter setFormatterBehavior:NSNumberFormatterBehavior10_4]; ②
[self.priceFormatter setNumberStyle:NSNumberFormatterCurrencyStyle]; ③

//从ProductIdentifiers.plist文件读取应用内产品标识
NSString* path = [[NSBundle mainBundle] pathForResource:@"ProductIdentifiers"
ofType:@"plist"]; ④
NSArray* array = [[NSArray alloc] initWithContentsOfFile:path]; ⑤
//从NSArray转化为NSSet
self.productIdentifiers = [[NSSet alloc] initWithArray:array]; ⑥

//添加self作为交易观察者对象
[[SKPaymentQueue defaultQueue] addTransactionObserver:self]; ⑦
}

```

在viewDidLoad方法中，第①~③行代码设置数字格式属性priceFormatter，我们需要将从App Store请求返回的产品金额本地格式化显示。第①行代码实例化NSNumberFormatter。第②行代码定义数字格式化行为，常量NSNumberFormatterBehavior10_4是指从Mac OS X 10.4之后采用的。第③行代码设置数字格式，其中常量NSNumberFormatterCurrencyStyle指定为当前样式。

第④~⑥行代码从ProductIdentifiers.plist文件中读取应用内产品标识，苹果建议产品标识不要硬编码到程序中。但是从文件中读取的类型是NSArray类型，而productIdentifiers属性是NSSet类型，需要从NSArray类型转换为NSSet类型，NSSet的构造方法initWithArray:可以实现该目的。

第⑦行代码将当前视图控制器对象作为交易观察者对象。当前视图控制器对象实现了交易观察者协议，即当前视图控制器对象现在的身份就是交易观察者了，但是需要addTransactionObserver:方法设定到付款队列中。

2. 获得产品信息

要获得产品信息，可以通过SKProductsRequest对象实现。获得产品信息请求完成回调productsRequest:didReceiveResponse:方法，然后在这个方法中更新UI刷新表视图。

下面我们看看ViewController.m中request:方法的代码：

```

- (IBAction)request:(id)sender {

    //检查设备是否在家长控制中禁止应用内购买
    if ([SKPaymentQueue canMakePayments]) {           ①
        //没有设置可以请求应用内购买信息
        SKProductsRequest *request= [[SKProductsRequest alloc]
initWithProductIdentifiers:self.productIdentifiers]; ②
        request.delegate = self;                       ③
        [request start];                                ④

        self.navigationItem.prompt = @"刷新中...";      ⑤
        self.refreshButton.enabled = NO;               ⑥
        self.restoreButton.enabled = NO;               ⑦

    } else {
        //有设置的情况下
        UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"访问限制"
message:@"您不能进行应用内购买！"
delegate:nil
cancelButtonTitle:@"Ok"
otherButtonTitles: nil];
        [alertView show];
    }
}

```

request:方法在点击“刷新”按钮时触发,其中第①行代码通过[SKPaymentQueue canMakePayments]语句判断是否已经在家长控制中禁止应用内购买项目。选择“设置”→“通用”→“访问限制”,可以进行家长控制禁止设置,如图14-47所示。



图14-47 家长控制设置

第②行代码的作用是实例化SKProductsRequest对象。构造方法initWithProductIdentifiers:使用产品标识符集合作为参数来创建SKProductsRequest对象,其中产品标识符是我们在iTunes Connect中设定的。第③行代码将SKProductsRequest对象的delegate属性设为self,这需要当前视图控制器实现SKProductsRequestDelegate协议。实现SKProductsRequestDelegate协议时,productsRequest:didReceiveResponse:是必须要实现的方法。第④行代码的作用是开始发出请求。

下面我们看看程序代码ViewController.m中实现SKProductsRequestDelegate协议的productsRequest:didReceiveResponse:方法:

```
- (void)productsRequest:(SKProductsRequest *)request didReceiveResponse:
    (SKProductsResponse *)response
{
    NSLog(@"加载应用内购买产品...");

    self.navigationItem.prompt = nil;
    self.refreshButton.enabled = YES;
    self.restoreButton.enabled = YES;

    self.skProducts = response.products;
    for (SKProduct * skProduct in self.skProducts) {
        NSLog(@"找到产品: %@ %@ %0.2f",
              skProduct.productIdentifier,
              skProduct.localizedTitle,
              skProduct.price.floatValue);
    }

    [self.tableView reloadData];
}
```

该方法在请求产品返回后回调,其中response参数是返回的应答对象,它的products属性返回skProduct对象的集合。skProduct对象描述产品信息,其productIdentifier属性是产品标识符,localizedTitle属性是产品的名字,price属性是产品的价格。

在ViewController.m中,UITableViewDataSource协议的tableView:cellForRowAtIndexPath:方法的代码如下:

```

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleSubtitle
            reuseIdentifier:CellIdentifier];
    }

    int row = [indexPath row];
    SKProduct * product = self.skProducts[row]; ①

    cell.textLabel.text = product.localizedTitle; ②

    [self.priceFormatter setLocale:product.priceLocale]; ③
    cell.detailTextLabel.text = [self.priceFormatter
        stringFromNumber:product.price]; ④

    //从应用设置文件中读取购买信息
    BOOL productPurchased = [[NSUserDefaults standardUserDefaults]
        boolForKey:product.productId]; ⑤
    if (productPurchased) {
        cell.accessoryType = UITableViewCellAccessoryCheckmark;
        cell.accessoryView = nil;
    } else {
        UIImage *buttonUpImage = [UIImage imageNamed:@"button_up.png"];
        UIImage *buttonDownImage = [UIImage imageNamed:@"button_down.png"]; ⑥

        UIButton *button = [UIButton buttonWithType:UIButtonTypeCustom];
        button.frame = CGRectMake(0.0f, 0.0f, buttonUpImage.size.width,
            buttonUpImage.size.height);
        [button setBackgroundImage:buttonUpImage forState:UIControlStateNormal];
        [button setBackgroundImage:buttonDownImage
            forState:UIControlStateHighlighted];
        [button setTitle:@"购买" forState:UIControlStateNormal];
        button.tag = indexPath.row; ⑦
        [button addTarget:self action:@selector(buttonTapped:)
            forControlEvents:UIControlEventTouchUpInside]; ⑧

        cell.accessoryView = button;
    }
    return cell;
}

```

该方法为表视图单元格提供数据。第①行代码从产品集合中取出SKProduct对象。第②行代码从SKProduct对象中取出localizedTitle属性内容，并将其设置到单元格主标题（textLabel）上。第③行代码设置属性priceFormatter的数字格式的本地化信息，product.priceLocale属性是SKProduct对象的本地化信息。第④行代码按照本地化设定格式化产品价格，然后将内容设置到单元格的副标题（detailTextLabel）中。第⑤行代码从应用设置文件中判断该产品是否已经被购买，每次购买产品后，本地的设置文件会记录下购买信息。

第⑥~⑧行代码用于实例化UIButton对象，并设置按钮。第⑦行代码将单元格行号赋值给按钮的tag属性，第⑧行代码设置按钮的触摸事件关联到buttonTapped:方法，即触摸按钮时会调用buttonTapped:方法。

ViewController.m中buttonTapped:方法的代码如下：

```

- (void)buttonTapped:(id)sender {

    UIButton *buyButton = (UIButton *)sender;
    //通过按钮的tag属性获得被点击按钮的索引，使用索引从数组中取出SKProduct对象
    SKProduct *product = self.skProducts[buyButton.tag]; ①
    //获得产品的付款对象

```

```

SKPayment * payment = [SKPayment paymentWithProduct:product];           ②
//把付款对象添加到付款队列中
[[SKPaymentQueue defaultQueue] addPayment:payment];                     ③
}

```

该方法是在用户触摸某个单元格后的按钮时触发的方法。在第①行代码中，按钮的tag属性是选中单元格的索引，通过这个索引从集合中取出SKProduct对象。第②行代码使用paymentWithProduct:方法创建SKPayment对象，SKPayment是付款对象。第③行代码将付款对象添加到付款队列中。

3. 处理交易结果

接收交易是通过实现SKPaymentTransactionObserver协议的观察者完成的，这个观察者在在本例中就是当前的视图控制器。调用下面的代码可以使观察者对象开始监听交易事件结果：

```
[[SKPaymentQueue defaultQueue] addTransactionObserver:self];
```

下面的代码是ViewController.m中实现SKPaymentTransactionObserver协议的代码：

```

- (void)paymentQueue:(SKPaymentQueue *)queue updatedTransactions:(NSArray *)transactions
{
    for (SKPaymentTransaction * transaction in transactions) {
        switch (transaction.transactionState)
        {
            case SKPaymentTransactionStatePurchased: //交易完成
                [self completeTransaction:transaction];
                break;
            case SKPaymentTransactionStateFailed: //交易失败
                [self failedTransaction:transaction];
                break;
            case SKPaymentTransactionStateRestored: //交易恢复
                [self restoreTransaction:transaction];
            default:
                break;
        }
    }
}

```

该方法在交易结果更新时被调用，其中transactions参数是更新的交易（SKPaymentTransaction）对象的集合，我们通过循环遍历交易集合逐一处理交易。交易状态（transactionState）有3种：交易完成（SKPaymentTransactionStatePurchased）、交易失败（SKPaymentTransactionStateFailed）和交易恢复（SKPaymentTransactionStateRestored）。根据不同状态，我们可以调用不同的方法。

这些响应处理交易的方法如下：

```

//交易完成
- (void)completeTransaction:(SKPaymentTransaction *)transaction {
    NSLog(@"交易完成...");
    [self provideContentForProductIdentifier:transaction.payment.productIdentifier]; ①
    //把交易从付款队列中移除
    [[SKPaymentQueue defaultQueue] finishTransaction:transaction];                ②
}

//交易恢复
- (void)restoreTransaction:(SKPaymentTransaction *)transaction {
    NSLog(@"交易恢复...");

    self.navigationItem.prompt = nil;
    self.refreshButton.enabled = YES;
    self.restoreButton.enabled = YES;

    [self provideContentForProductIdentifier:transaction.originalTransaction.
        payment.productIdentifier]; ③
    [[SKPaymentQueue defaultQueue] finishTransaction:transaction];                ④
}

```



```

    }

    // 交易失败
    - (void)failedTransaction:(SKPaymentTransaction *)transaction {

        NSLog(@"交易失败...");
        if (transaction.error.code != SKErrorPaymentCancelled)
        {
            NSLog(@"交易失败: %@", transaction.error.localizedDescription);
        }

        [[SKPaymentQueue defaultQueue] finishTransaction: transaction];
    }

    // 购买成功
    - (void)provideContentForProductIdentifier:(NSString *)productIdentifier {
        [[NSUserDefaults standardUserDefaults] setBool:YES forKey:productIdentifier];
        [[NSUserDefaults standardUserDefaults] synchronize];
        [self.tableView reloadData];
    }

```

在上述代码中，completeTransaction:是处理交易成功的方法，其中第①行代码传递产品标识符调用provideContentForProductIdentifier:方法处理交易，第②行代码[[SKPaymentQueue defaultQueue] finishTransaction:transaction]语句结束交易，并把交易从付款队列中移除。无论什么状态下的交易，都应该调用第②行语句结束交易。

restoreTransaction:是恢复交易的方法，其中第③行代码传递上一次交易的产品标识符调用provideContentForProductIdentifier:方法处理交易，transaction.originalTransaction.payment.productIdentifier语句用来获得上次交易产品的标识符，originalTransaction属性是上次交易对象。

failedTransaction:是处理交易失败的方法。在该方法中，输出取消以外的交易失败信息。注意在用户选择取消操作时，也会调用该方法，因此通过代码if (transaction.error.code != SKErrorPaymentCancelled)代码判断这非取消的情况。

provideContentForProductIdentifier:是我们自己编写的方法，用于处理交易成功和恢复交易这两种情况。在该方法中，我们主要将交易中的产品记录到本地的应用设置文件中。

4. 恢复交易

由于多种原因，用户需要恢复交易。我们在视图的导航栏左边放置了一个恢复按钮，当用户点击该按钮时，会触发restore:方法，它的代码如下：

```

- (IBAction)restore:(id)sender {
    self.navigationItem.prompt = @"恢复中...";
    self.refreshButton.enabled = NO;
    self.restoreButton.enabled = NO;
    [[SKPaymentQueue defaultQueue] restoreCompletedTransactions];
}

```

恢复交易主要通过[[SKPaymentQueue defaultQueue] restoreCompletedTransactions]语句实现的。此外，恢复时，也会触发SKPaymentTransactionObserver观察者的paymentQueue:updatedTransactions:方法。

14.4.4 测试应用内购买

在测试应用之前，我们还需要设置一下应用的环境。下面我们修改一下本地资源文件ProductIdentifiers.plist，修改后的内容如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"

```

```
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
  <string>com.51work6.IAPDemo.map</string>
  <string>com.51work6.IAPDemo.tool</string>
  <string>com.51work6.IAPDemo.elves</string>
  <string>com.51work6.IAPDemo.Equipment</string>
</array>
</plist>
```

修改这个资源文件的目的是告诉我们的应用程序需要查找哪些产品，其中每一个配置项都是产品标识符。然后我们就可以运行了。运行IAPDemo应用，我们会看到如图14-48所示的界面。

点击刷新按钮开始请求数据，如图14-49所示。在请求过程中，导航栏中的两个按钮都不能点击，请求返回之后，两个按钮才可以使用。

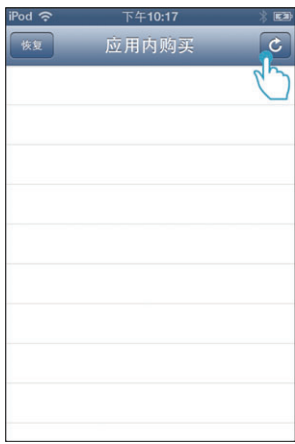


图14-48 运行IAPDemo应用的界面



图14-49 点击刷新按钮后IAPDemo界面

选择要购买的产品，然后点击“购买”按钮，稍等几秒钟，会弹出询问是否购买该产品对话框，如图14-50所示。大家不要担心，购买是在沙箱中进行的，不会发生真正的购买行为。

点击“购买”按钮，将弹出如图14-51所示的输入购买账号对话框。需要注意的是，我们在沙箱中模拟购买时，购买账号必须是测试账号，即我们在iTunes Connect中创建的测试用户。



图14-50 确认购买对话框



图14-51 输入测试账号

非测试账号是不能测试的。如果已经在设备中设置了其他的账号登录，请进入“设置”应用→“iTunes Store和App Store”中注销账号，然后再重新购买。

如果购买成功，会出现如图14-52所示的画面，其中成功购买产品的单元格后面的“购买”按钮，变成了选中符号✓。

点击“恢复”按钮的过程与刷新按钮基本类似，其中也要输入购买账号。恢复成功后，会刷新表视图，如图14-53所示。



图14-52 测试成功



图14-53 恢复成功

14.5 小结

通过对本章的学习，我们了解了iOS中的商业模式，其中收费策略值得广大读者借鉴。此外，我们还介绍了植入广告和应用内购买的API，其中植入广告包括苹果自己的iAd和谷歌的AdMob广告。

编码过程中出现bug^①在所难免，有时在找出这些bug上耗费的精力和时间不比重新创建一个工程少多少。调试可以帮我们找出程序中的bug，熟练掌握各种调试工具能够帮助我们快速找出程序中的bug，提高开发效率。

15.1 Xcode 调试工具

Xcode提供了强大的代码编辑、性能分析和调试功能，我们应该熟练掌握这些功能。为了便于学习，我们把10.4节的案例拿到本章中使用，它的代码是分层的，每层都处于不同的工程中。

15.1.1 定位编译错误

使用Xcode工具开发时，很容易出现定位编译错误，而大部分编译错误都会在编写代码时提示给程序员，一小部分编译错误等到编译时才能够显示出来。如图15-1所示，出现错误的位置会显示红色圆形感叹号❗。

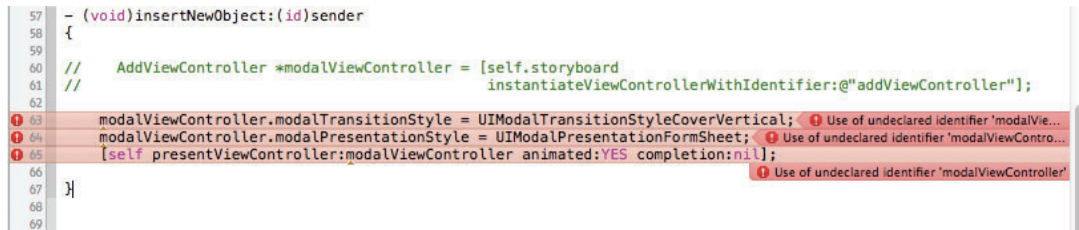


图15-1 Xcode中显示编译错误

提示 程序错误分为编译错误和逻辑错误，其中前者是在程序编译时暴露出来的错误，我们通过Xcode可以定位到这些错误，编译器还会给出错误原因提示；后者是程序运行结果与我们期望的不一致，这些错误可以通过调试和测试找出。

编译Xcode工程时，可以使用Ctrl+B快捷键完成，也可以通过Product→Build菜单项来完成。除了显示编译错误，Xcode还可以显示警告。每一个警告我们都不应该忽略，它可能引起应用运行时的崩溃。在Xcode中显示警告，会显示黄色三角形感叹号⚠️，如图15-2所示。

① bug是指程序中的缺陷和漏洞。在本书中，bug也包括了错误和异常。

在导航面板中，左侧显示的是每次的操作，右边是对应操作的日志。图15-4所示是进行的编译操作，在日志中会显示正常消息和问题消息，正常消息是绿色圆形对号✔，问题消息包括了警告和错误，它们的图标与上一节介绍的一样。点击每一行结尾的显示列表图标☰，会列出该项目的详细信息，如图15-5所示。

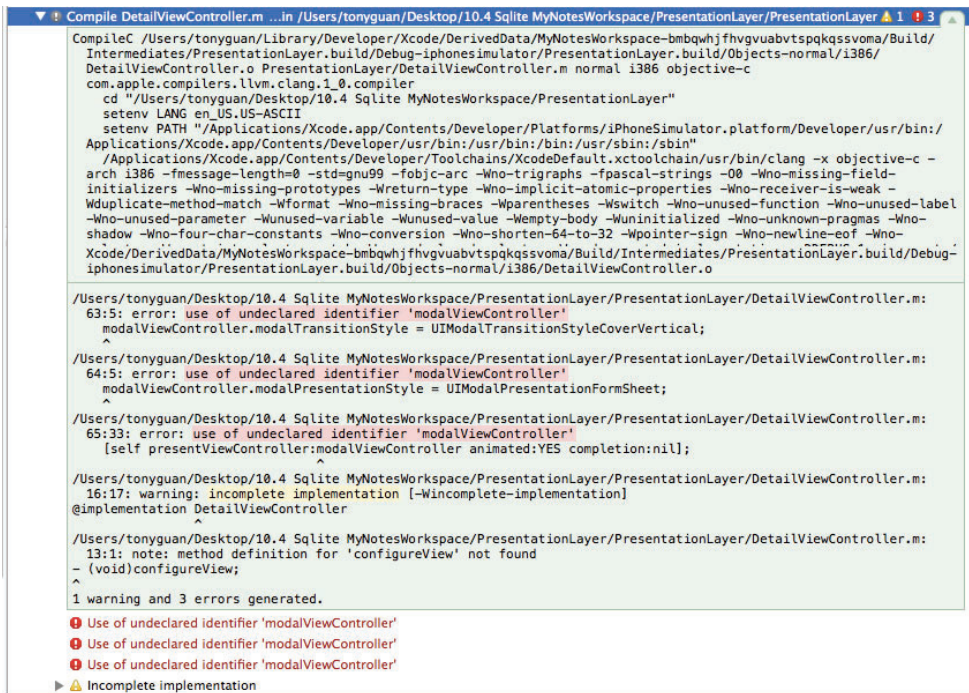


图15-5 显示日志项目详细信息

图15-5显示了编写DetailViewController.m日志的过程，告诉用户编写时采用了哪些命令、成功还是失败、失败的原因等。

日志分析很有用。例如，我们在上传应用到App Store时，需要找到编译之后的.app文件，此时我们可以打开Touch命令日志项目，如图15-6所示。

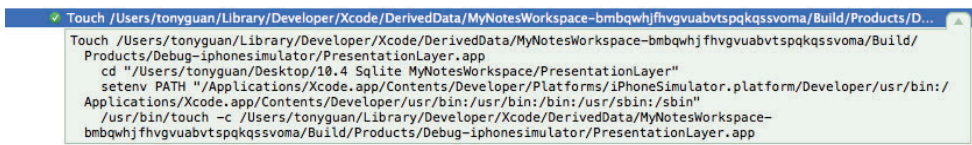


图15-6 Touch命令日志项目

从图中可以看到生成的PresentationLayer.app文件被复制到下面的目录中：`/Users/tonyguan/Library/Developer/Xcode/DerivedData/MyNotesWorkspace-bmbqwhjfhvgvuabvtspqkqssvoma/Build/Products/Debug-iphonesimulator/`。

15.1.3 设置和查看断点

第一次运行编写成功的程序时，往往会出现始料未及的结果。为了找出原因，我们需要在程序中设置断点进行调试。断点指在条件满足的情况下程序会挂起在那里，我们可以在这里查看变量、单步运行等操作内容。断点可以分为以下3种类型。

- ❑ 文件行断点。执行到特定文件某一行时触发。
- ❑ 符号断点。调用某一个函数或方法时触发，程序挂起在函数或方法的第一行。
- ❑ 异常断点。产生异常时触发，分为Objective-C异常断点和C++异常断点。Objective-C异常断点会在遇到Objective-C异常时触发，C++异常断点在遇到C++异常时触发。

1. 文件行断点设置

设置文件行断点很简单，直接点击该文件中的行号即可。图15-7在MasterViewController.m文件中设置第34行作为断点。

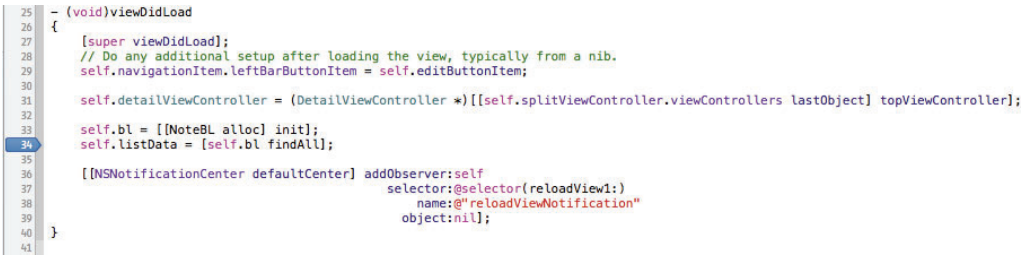


图15-7 设置文件行断点

断点可以删除、禁止使用和编辑。在断点上点击鼠标右键，弹出的快捷菜单如图15-8所示。当选择Disable Breakpoint菜单项时，会禁止使用断点，这时断点处于灰色状态。选择Delete Breakpoint菜单项时，可以删除断点。此外，也可以拖曳断点离开行号列，当出现一个小云朵时释放鼠标，这样也可以删除断点。



图15-8 断点管理

选择Edit Breakpoint菜单项，会弹出断点编辑对话框，如图15-9所示。在断点编辑对话框中，我们可以为断点设定触发条件和忽略次数，并添加动作。

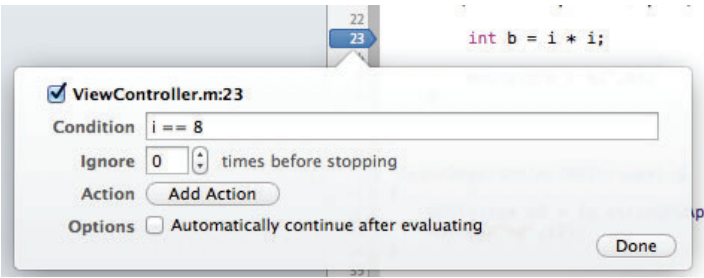


图15-9 编辑断点


如果我们有下面这段程序代码，我只是想看看`i==8`是什么情况，可以在第23行（`int b = i * i;`）中设置断点。默认情况下，循环体10次都触发断点，我们需要在图15-9的Condition中设置`i==8`：

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

    for(int i =0 ; i < 10 ;i++) {
        int b = i * i;
        NSLog(@"b = %i",b);
    }
}
```

这样当程序运行到`i==8`时，程序就会挂起。我们还可以Ignore中设置忽略次数，同样上面的需求可以设置Ignore为8，也可以达到同样的效果。

2. 符号断点设置

设置符号断点与设置文件行断点不同，需要点击导航面板中的  按钮打开断点导航面板，如图15-10所示。在断点导航面板中，可以看到所有的断点。

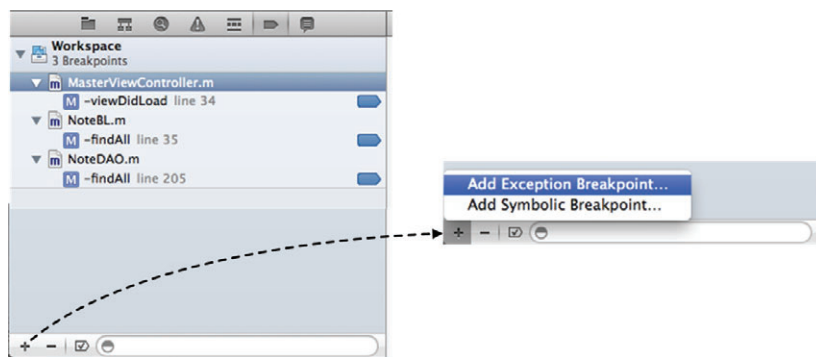


图15-10 断点导航面板

点击图15-10左下角的+按钮，会弹出如图15-10右图所示的菜单，其中有两项——Add Symbolic Breakpoint和Add Exception Breakpoint，前者可以创建符号断点，后者可以创建异常断点。这里我们选择Add Symbolic Breakpoint菜单项，此时可以弹出创建符号断点对话框，如图15-11所示。

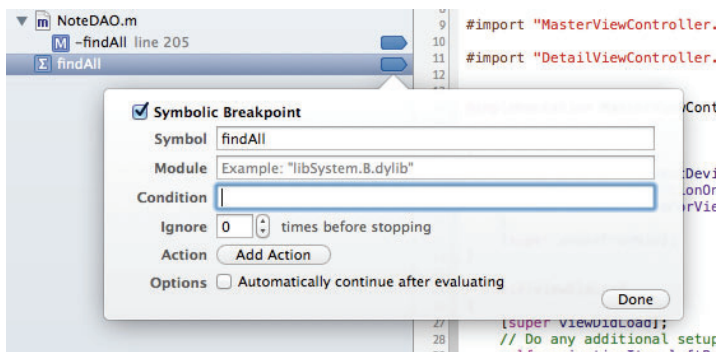


图15-11 创建符号断点对话框

在Symbol中输入findAll方法。NoteBL和NoteDAO类中都有这个方法，因此这两个findAll方法都会触发断点。图15-12和图15-13所示的断点挂起在findAll方法的第一行。

```

31 //查询所用数据方法
32 -(NSMutableArray*) findAll
33 {
34     NoteDAO *dao = [NoteDAO sharedManager];
35     return [dao findAll];
36 }
37 @end
38

```

图15-12 断点挂起在NoteBL.m中findAll方法的第一行

```

162 //查询所有数据方法
163 -(NSMutableArray*) findAll
164 {
165     NSString *path = [self applicationDocumentsDirectoryFile];
166     NSMutableArray *listData = [[NSMutableArray alloc] init];
167     if (sqlite3_open([path UTF8String], &db) != SQLITE_OK) {
168         sqlite3_close(db);
169         NSAssert(NO, @"数据库打开失败。");
170     } else {
171         // ...
172     }
173 }
174

```

图15-13 断点挂起在NoteDAO.m中findAll方法的第一行

3. 异常断点设置

在图15-10中点击左下角的+按钮，从弹出菜单中选择Add Exception Breakpoint，此时会弹出创建异常断点对话框，如图15-14所示。

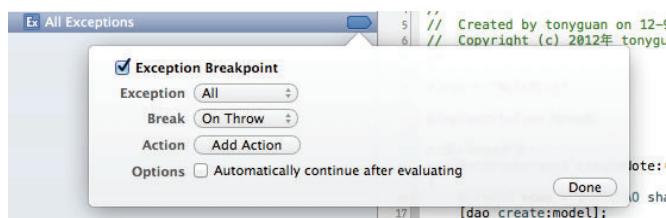


图15-14 设置异常断点

在Exception项中，可以选择Objective-C或者C++异常断点，Break项可以设定On Throw还是On Catch，即断点是在抛出时触发还是在捕获时触发。

为了测试，我们可以修改一下MasterViewController.m中的viewDidLoad方法，制造一个运行期异常：

```

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    self.detailViewController = (DetailViewController *)
        [[self.splitViewController.viewControllers lastObject] topViewController];

    self.bl = [[NoteBL alloc] init];
    self.listData = [self.bl findAll];

    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(reloadView:)
                                             name:@"reloadViewNotification"
                                             object:nil];
}

```

原来的语句是：

```

[[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(reloadView:)
                                             name:@"reloadViewNotification"
                                             object:nil];

```

修改为：

```

[[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(reloadView1:)
                                             name:@"reloadViewNotification"
                                             object:nil];

```

也就是reloadView1:方法是不存在的，当点击保存操作时，这里就会抛出异常。默认情况下，没有捕获和设置断点，程序会直接跳到main.m中的main函数里面了，如图15-15所示。

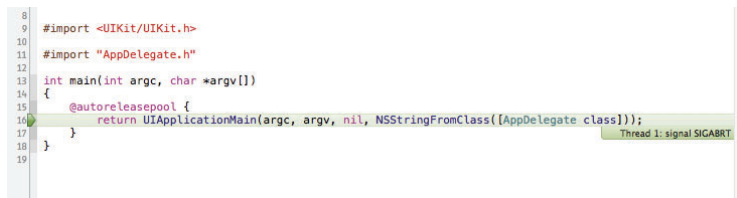


图15-15 异常的默认处理

如果我们设置了异常断点,程序会挂起在AddViewController.m中onClickSave:方法的第50行,如图15-16所示。

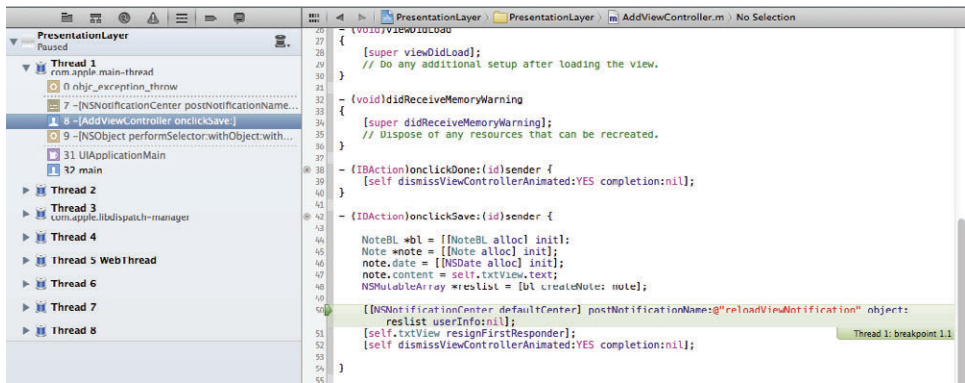


图15-16 设置了异常断点处理

默认情况下,异常会在输出窗口中输出。本例的输出结果如下:

```

2012-10-15 21:43:08.382 PresentationLayer[4293:c07] -[MasterViewController reloadView1]:
unrecognized selector sent to instance 0x7541270

```

15.1.4 调试工具栏

在Xcode编辑区下方,有一个调试窗口,分为调试工具栏、窗口显示按钮、变量查看窗口和输出窗口等几个部分,如图15-17所示。

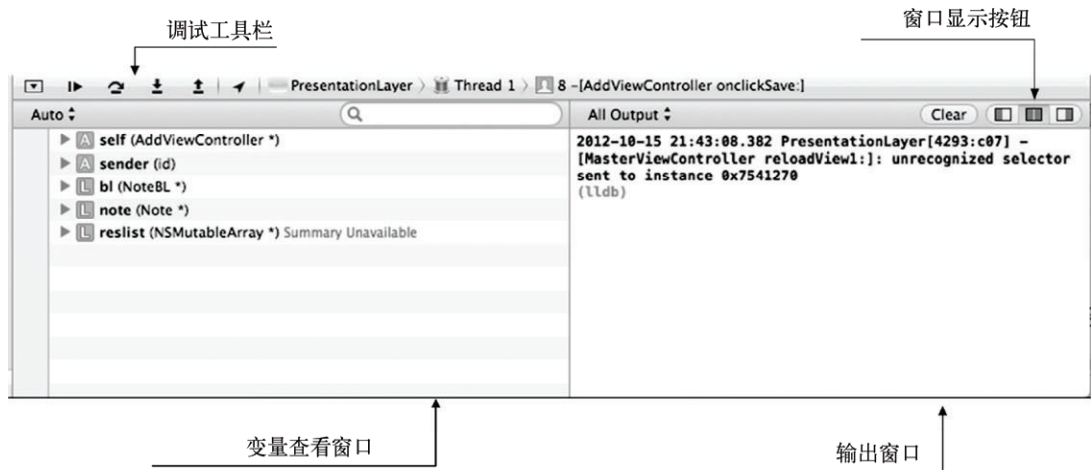


图15-17 调试窗口

图15-18是调试工具栏，其中隐藏按钮可以隐藏或者显示调试窗口；模拟位置按钮可以向模拟器设备发送虚拟的位置坐标，它应用于位置服务应用中的测试；使用跳转栏，可以跳转到具体工程下某个类的方法中，能够跟踪程序的运行过程。



图15-18 调试工具栏

在执行控制按钮中，有4个是我们在调试过程中最为常用的，如图15-19所示。在断点挂起之后，点击继续执行按钮可以继续执行。单步跳过按钮是单步执行，遇到方法和函数时不进入。单步进入方法则进入到方法或者函数里。单步跳出在进入到方法或函数里面，想跳回到原来调用它的地方时使用。

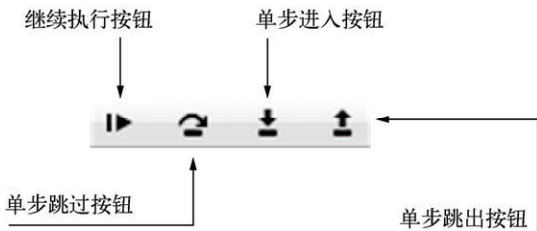


图15-19 执行控制按钮

例如，当程序挂起在viewDidLoad方法的语句self.listData = [self.bl findAll]时，如图15-20所示，点击继续执行按钮，程序就会接着运行，直到运行到下一个断点，或者结束。如果点击单步跳过按钮，则执行到第36行。如果点击单步进入按钮，则进入NoteBL.m的findAll方法。

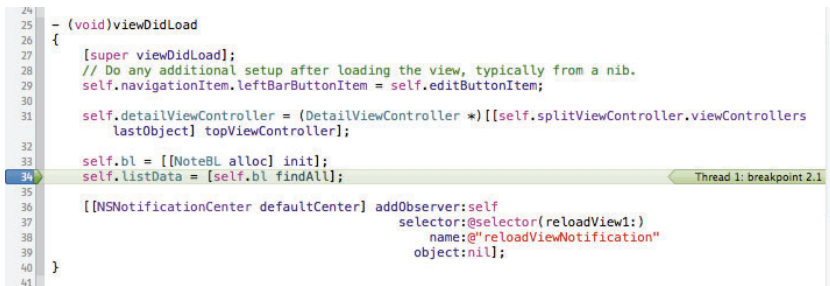


图15-20 程序挂起

15.1.5 输出窗口

使用窗口显示按钮，可以控制同时显示左右两个窗口（变量查看窗口和输出窗口）或者只显示其中一个窗口。输出窗口有3个选择：All Output、Debugger Output和Target Output，如图15-21所示。

在程序调试时，可以在Debugger Output窗口中执行编译器的调试命令。如图15-22所示，p命令是计算基本数据类型表达式，po命令是计算对象类型的表达式，其他命令我们将在后面介绍。

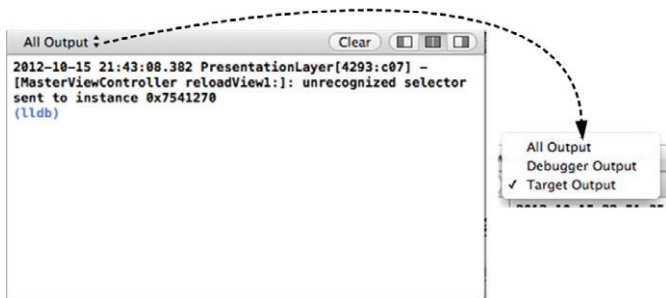


图15-21 输出窗口

```
(lldb) p 9+9
(int) $4 = 18
(lldb) po note
(Note *) $5 = 0x07166510 <Note: 0x7166510>
(lldb)
```

图15-22 Debugger Output窗口

在Target Output窗口中，可以显示程序出错和异常等信息，以及通过一些函数（如NSLog和NSAssert函数）输出的信息，如图15-23所示。

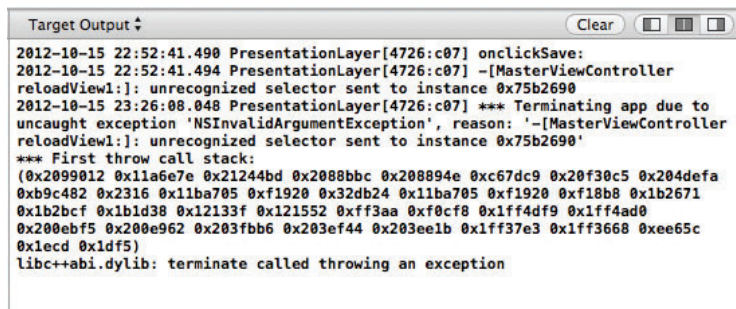


图15-23 Target Output窗口

在图15-23中，有NLog函数的输出信息onlickSave:，其他信息是都是异常输出信息。

15.1.6 变量查看窗口

变量查看窗口位于调试窗口的左侧，用于查看变量和寄存器内容。通过点击窗口左上角的小三角，可以选择查看变量的范围——Auto、Local Variables和All Variables, Registers, Globals and Statics，如图15-24所示，其中各个项的含义如下所示。

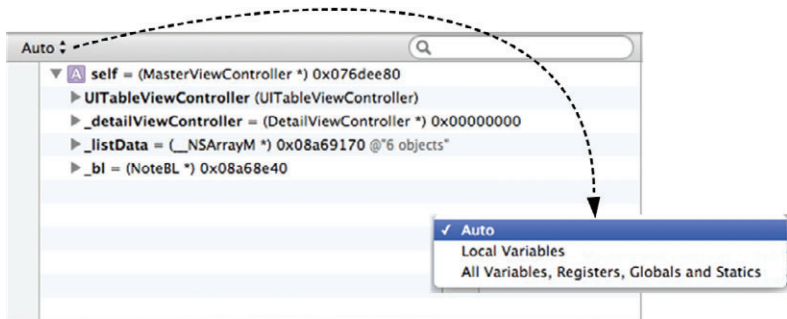


图15-24 变量查看窗口

- ❑ Auto。查看经常使用的变量。
- ❑ Local Variables。查看本地变量。
- ❑ Variables, Registers, Globals and Statics。查看全部变量，包括寄存器和全局变量等，如图15-25所示，其中图标A是自动变量、S是静态变量、R是寄存器、L是本地变量。

在变量查看窗口中选择变量后，点击鼠标右键，会弹出一个快捷菜单，从中可以进行一些操作，下面我们先演示一下如何编辑变量。如图15-26所示，选择变量*i* = (int) 10，点击鼠标右键，从弹出的快捷菜单中选择Edit Value菜单项，此时该行代码进入修改状态，我们可以在其中输入相关值。

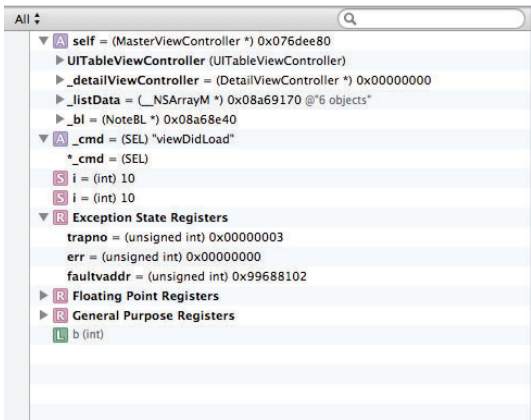


图15-25 变量查看窗口的全部变量选项

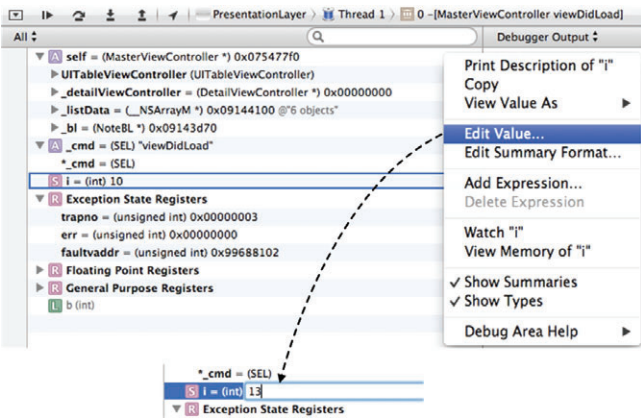


图15-26 修改变量

此外，我们也可以通过快捷菜单打印变量。如图15-27所示，右击*listData*变量（它是NSArrayM*类型的对象），从弹出的快捷菜单中选择Print Description of "*_listData*"菜单项，此时在变量输出窗口打印输出变量。

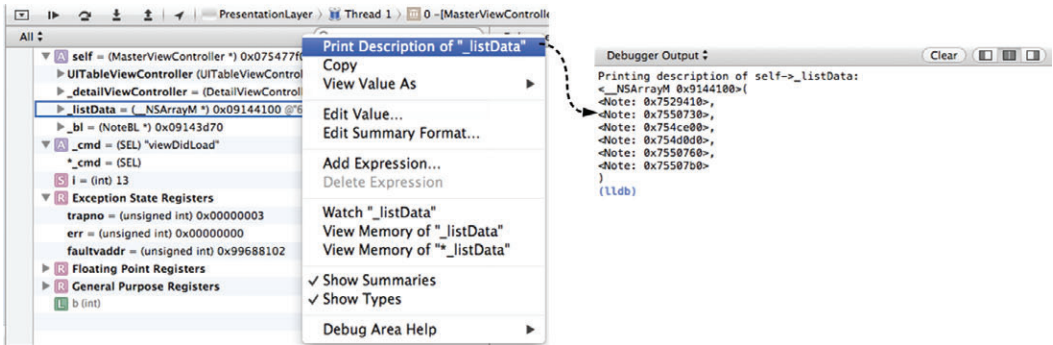


图15-27 打印变量

15.1.7 查看线程

iOS和Mac OS X是支持多线程的。作为开发工具的Xcode，要能够查看线程的情况。在Xcode中，有两种方式可以查看线程，一个是在跳转栏中选择线程下拉列表，如图15-28所示。选择某个线程后，会显示一个代码运行的堆栈。

选择堆栈中的方法，此时编辑窗口会进入该方法中，如果该方法没有源代码，将显示汇编语言。

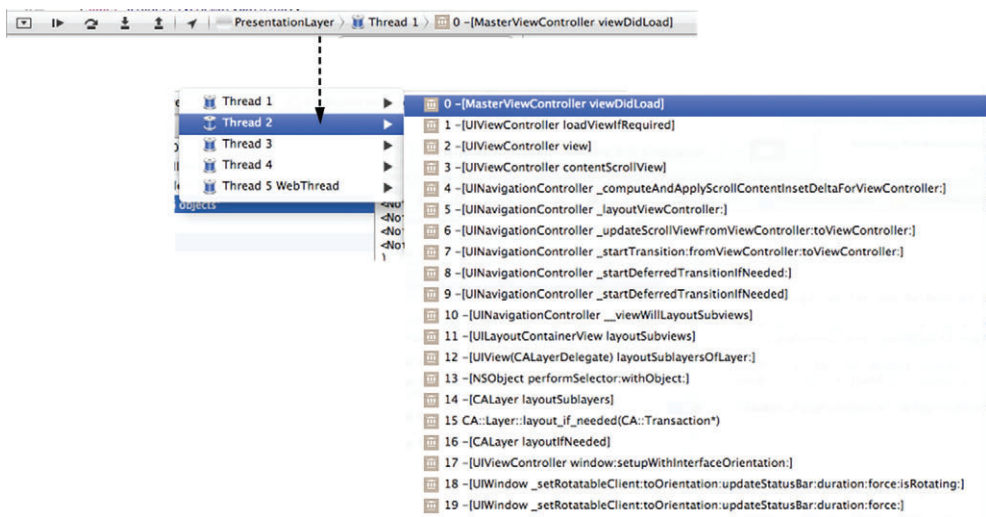



图15-28 在跳转栏中查看线程

另一种查看线程的方法是在导航栏面板中查看。如图15-29所示，点击“显示调试导航面板”按钮, 借助这个面板可以查看线程情况。该面板只显示大概的调用堆栈，没有上一种查看方式反应的情况详细。

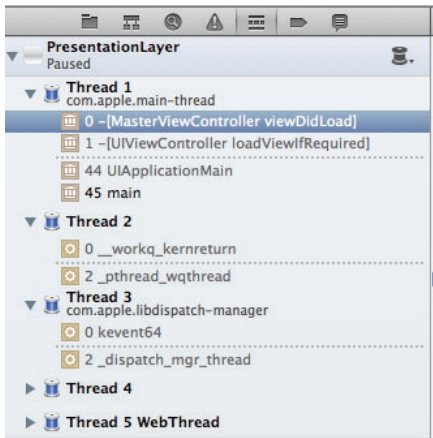


图15-29 在导航面板中查看线程

15.2 日志与断言输出

在程序运行过程中，可以输出一些信息到输出窗口，其中有些信息是应用遇到错误或异常时由系统抛出的错误堆栈信息。当然，也可以根据需要自己输出或抛出异常。根据这些信息，我们可以分析程序出了什么问题以及程序的运行情况。

15.2.1 使用NSLog函数

NSLog是Foundation框架提供的Objective-C日志输出函数，与标准C中的printf函数类似，可以格式化输出。对于不同的数据类型，NSLog函数中格式化字符串是不同的，如表15-1所示。

表15-1 NSLog函数中的格式化字符串

类 型	实 例	NSLog中的格式化字符串
char	'a', '\n'	%c
short int	-10	%hi, %hx, %ho
unsigned short int	9	%hu, %hx, %ho
int	17, -99, 0xFFAE, 0878	%i, %x, %o
unsigned int	17u, 101U, 0xFFu	%u, %x, %o
long int	17L, -2998, 0xffffL	%li, %lx, %lo
unsigned long int	17UL, -100ul, 0xffffUL	%li, %lx, %lo
long long int	0xe5e5e5e5LL, 50011	%lli, %llx, %llo
unsigned long long int	17ull, 0xffefULL	%llu, %llx, %llo
float	12.3f, 3.1e-5f	%f, %e, %g
double	12.34, 3.1e-5	%f, %e, %g
long double	12.34l, 3.1e-5l	%Lf, %Le, %Lg
id	nil	%p, %@
对象指针类型 (NSArray*)	"<Note: 0x7697220>", "<Note: 0x769ad70>"	%@

与NSLog函数类似的还有NSLogv函数，它可以将输出重新定向到文件中。

15.2.2 使用NSAssert宏

NSLog函数是无条件输出，即程序运行到该语句，就会输出结果。如果想有条件输出结果，可以使用NSAssert宏。注意，NSAssert并不是函数，它的定义如下：

```
#define NSAssert(condition, desc, ...)
```

其中第一个参数condition是布尔表达式，第二个参数desc是描述信息，参数后面的...是格式化desc描述信息的。如果condition为NO，则输出desc描述信息，并抛出异常NSInternalInconsistencyException；如果condition为YES，则不输出信息。

在MasterViewController.m的viewDidLoad方法中，我们也使用了NSAssert代码，具体如下：

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

    for(; i < 10 ;i++) {

        NSAssert(i >= 0 && i < 9, @"i = %i变量超出了范围。", i);
        int b = i * i;
        NSLog(@"b = %i",b);
    }

    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    self.detailViewController = (DetailViewController *)
        [[self.splitViewController.viewControllers lastObject] topViewController];

    self.bl = [[NoteBL alloc] init];
    self.listData = [self.bl findAll];

    NSLog(@"%@",_listData);

    [[NSNotificationCenter defaultCenter] addObserver:self
```

```

selector:@selector(reloadView:)
    name:@"reloadViewNotification"
    object:nil];
}

```

当变量*i*等于9时，条件 (*i* >= 0 && *i* < 9) 为NO，NSAssert宏会抛出异常，输出窗口输出的内容如图15-30所示。输出信息有两条，一条是输出NSAssert宏的位置为MasterViewController.m中的33行；另一条是异常堆栈信息。

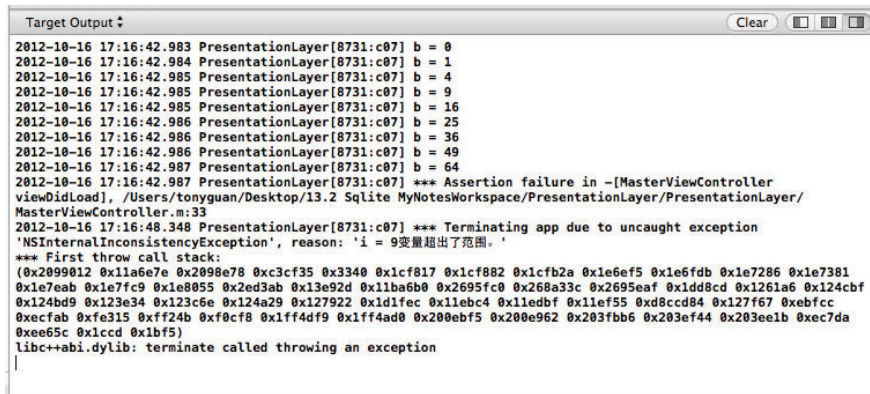


图15-30 NSAssert输出信息

与NSAssert类似的宏还有NSAssert1、NSAssert2、NSAssert3、NSAssert4和NSAssert5，其中宏后面的数字代表格式化描述信息参数的个数，它们的定义如下：

```

#define NSAssert1(condition, desc, arg1)
#define NSAssert2(condition, desc, arg1, arg2)
#define NSAssert3(condition, desc, arg1, arg2, arg3)
#define NSAssert4(condition, desc, arg1, arg2, arg3, arg4)
#define NSAssert5(condition, desc, arg1, arg2, arg3, arg4, arg5)

```

因此语句：

```
NSAssert(i >= 0 && i < 9, @"i = %i变量超出了范围。", i)
```

等价于：

```
NSAssert1(i >= 0 && i < 9, @"i = %i变量超出了范围。", i)
```

语句：

```
NSAssert(i >= 0 && i < 9, @"i = %i变量超出了范围,结果是%i。", i, i*i)
```

等价于：

```
NSAssert2(i >= 0 && i < 9, @"i = %i变量超出了范围,结果是%i。", i, i*i)
```

依次类推，NSAssert5可以提供5个格式化参数。

15.2.3 移除NSLog和NSAssert

我们使用NSLog和NSAssert的目的是为了调试，并在调试阶段输出一些信息，但是在调试结束、应用发布后，如果还使用NSLog和NSAssert输出信息，那样会影响性能。事实上，这个工作量是比较大的，而且刚刚移除掉时，你会发现又要进行调试，然后再把NSLog和NSAssert加入到程序代码中，很麻烦！因此，需要设定两套不同的编译参数环境，我们把这个环境称为Scheme。Xcode中的Scheme是一些Target的集合，它们配置不同的编译参数，也可能包括了一些可运行的测试集合。Xcode的Scheme位于Xcode的左上角，如图15-31所示，点击PresentationLayer会弹出Scheme菜单，从中可以编辑、新建和管理Scheme。

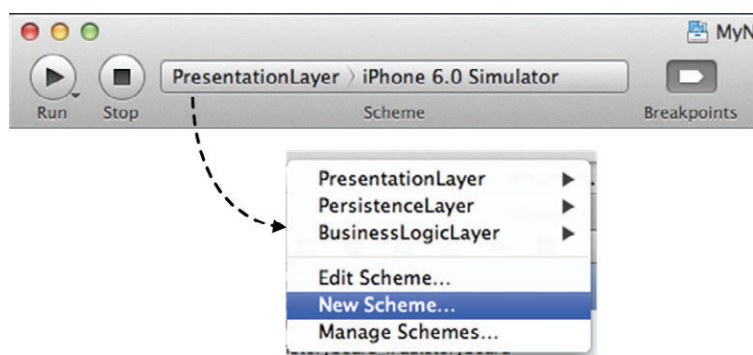


图15-31 Xcode中的Scheme

从图15-31中选择New Scheme菜单项，接着会弹出一个对话框，从中选择Target为PresentationLayer，如图15-32所示。

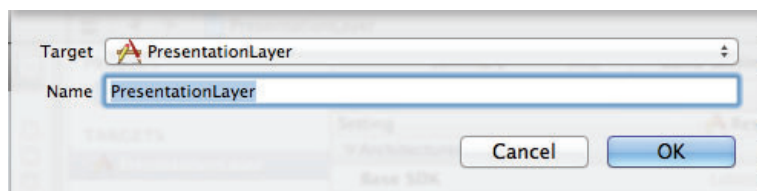


图15-32 选择Target为PresentationLayer

然后点击OK按钮就创建成功了，这样刚创建好的Scheme为PresentationLayer2。此时再选择Edit Scheme菜单项编辑PresentationLayer2。如图15-33所示，选择Info标签，将Build Configuration（编译配置）修改为Release。这里说明一下编译配置各个项的含义：Debug是为调试编译而配置的，Release是为发布编译而配置的。

接下来，我们需要为不同的Scheme配置不同的参数。由于移除NSLog和NSAssert的方式不同，所以下面我们分别介绍一下它们。

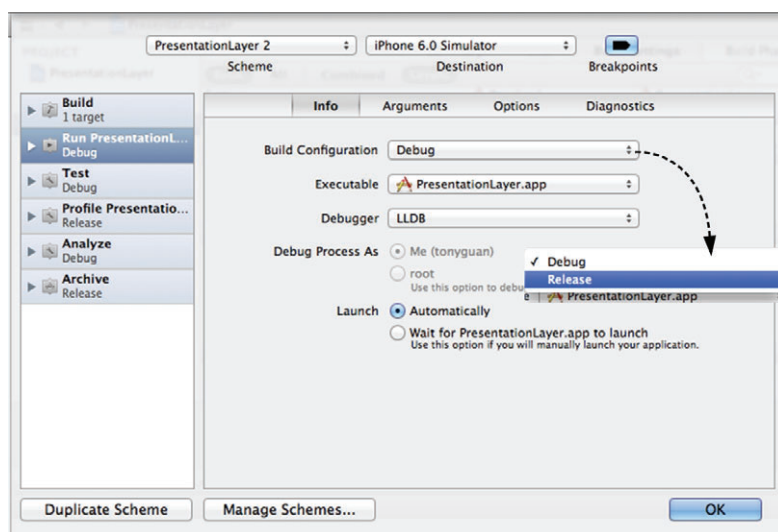


图15-33 选择编译配置为Release

1. 移除NSAssert

移除NSAssert比较简单，我们需要在TARGETS中选择Build Settings，找到Preprocessor Macros（预处理宏）项目，配置它的Release为NS_BLOCK_ASSERTIONS，具体操作步骤为：双击Release后面的空白处，此时会弹出对话框，点击对话框中的+添加NS_BLOCK_ASSERTIONS即可，如图15-34所示。

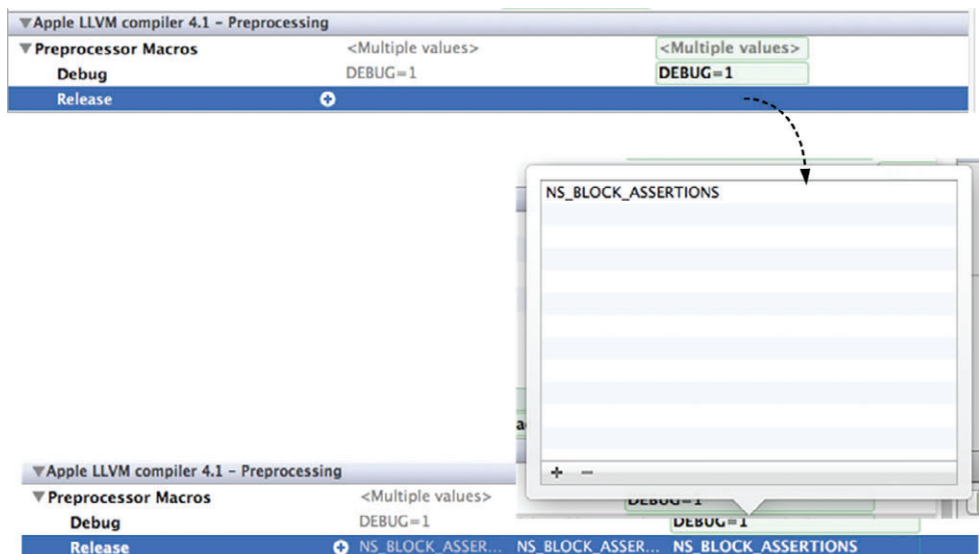


图15-34 配置为Release为NS_BLOCK_ASSERTIONS

NS_BLOCK_ASSERTIONS是Foundation框架中定义好的预处理宏，如果在编译环境中设置NS_BLOCK_ASSERTIONS，在编译的时候NSAssert宏将被移除，我们可以分别运行Scheme中的PresentationLayer→iPhone 6.0 Simulator和PresentationLayer2→iPhone 6.0 Simulator测试一下。比较测试结果，可以发现执行PresentationLayer时会抛出异常，而PresentationLayer2可以执行通过。

MyNotesWorkspace包含了3个工程，PresentationLayer工程中的Target配置了NS_BLOCK_ASSERTION。其他两个工程的配置可以参照上面的方法，这里不再赘述。

2. 移除NSLog

移除NSLog要比移除NSAssert复杂一些，需要修改程序代码。思路是重新定义一个宏替代NSLog，这个宏是有条件编译的。为了能够在工程所有源代码中使用这个宏，需要在<工程名>-Prefix.pch文件中定义这个宏。这个文件引入的.h文件和定义的宏作用于全部工程中的源代码模块，这样可以省去在每个.h文件中定义宏。打开PresentationLayer工程中的PresentationLayer-Prefix.pch，添加定义新的日志宏，内容如下：

```
#ifdef DEBUG
#   define DLog(...) NSLog(__VA_ARGS__)
#else
#   define DLog(...)
#endif
```

编译器在编译的时候判断是否定义DEBUG宏，如果定义了，则使用DLog替代NSLog。#ifdef...#else...#endif是条件编译语句，是在编译的时候编译器进行判断，新的日志宏是DLog。NSLog(__VA_ARGS__)中的__VA_ARGS__参数也是一个宏，它是一个可以提供可变参数的宏。

与移除NSAssert类似，需要在TARGETS中选择Build Settings，找到Preprocessor Macros项目，配置它的Debug为DEBUG或（DEBUG=1），如图15-35所示。

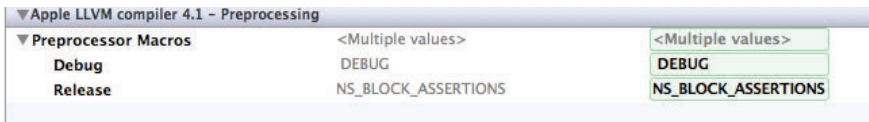


图15-35 配置为Debug为DEBUG

我们可以分别运行 Scheme 中的 PresentationLayer→iPhone 6.0 Simulator 和 PresentationLayer2→ iPhone 6.0 Simulator 测试一下，可以发现，PresentationLayer 执行时会有日志输出，而 PresentationLayer2 则没有。

为了在其他工程中也能移除 NSLog，需要同样的方法配置其他两个工程，配置过程同上，不再赘述。

15.3 LLDB 调试工具

我们在前面用过 LLDB 调试工具，它在输出窗口中使用 `p` 和 `po` 命令输出表达式的内容。`p` 和 `po` 就是调试工具的命令，调试工具的编译器相对独立于 Xcode。我们进行 Objective-C 程序开发时，用过 3 种编译器——GCC、LLVM^① 和 Apple LLVM，其中 GCC 是比较古老的编译器，现在我们主要使用 LLVM GCC 和 Apple LLVM。GCC 的调试工具是 GDB，是 GCC Debug 工具的缩写，LLVM GCC 和 Apple LLVM 的调试工具是 LLDB（或 `lldb`）。GDB 和 LLDB 命令有一些差别，本书只介绍 LLDB 命令。LLDB 命令很多，本节选取了几个常用的命令。

为了便于学习，我们还是把 10.4 节的案例拿出来做一些修改。如何进入 LLDB 进行调试呢？进入 LLDB 调试工具的一种方式是从终端进入，另外一种是从 Xcode 进入。Xcode 工具我们比较熟悉，这里主要介绍这种方式。具体做法很简单，就是在程序中设置断点，当程序挂起时，在输出窗口中选择 Debugger Output，这时输出窗口有 `(lldb)` 命令提示符，这就进入了 LLDB 调试工具了。

15.3.1 断点命令

关于断点命令，我们主要介绍设置断点、查看断点、删除断点、单步进入、单步跳出和继续运行等。

1. 设置断点

设置断点时，可以使用命令 `breakpoint set`，该命令可以设置文件行断点和符号断点，并且它们都有简略写法。

为了在 PresentationLayer 的 MasterViewController.m 中第 42 行设置断点（这是一种文件行断点），我们可以使用如下命令：

```
(lldb) breakpoint set --file MasterViewController.m --line 42
(lldb) br s -f MasterViewController.m -l 12
(lldb) b MasterViewController.m:12
```

这 3 条命令都可以实现同样的效果，后两个是简略写法。我们测试一下命令，发现程序执行到第 42 行后挂起了，如图 15-36 所示。

如果我们要在所有的 `findAll` 方法调用时挂起（这属于符号断点设置），可以使用如下命令：

```
(lldb) breakpoint set --selector findAll
(lldb) br s -S findAll
```

这两条命令都可以实现同样的效果，后者是简略写法。这个命令很简单，大家可以自己测试一下。

^① LLVM 是 Low Level Virtual Machine（低级虚拟机）的缩写。

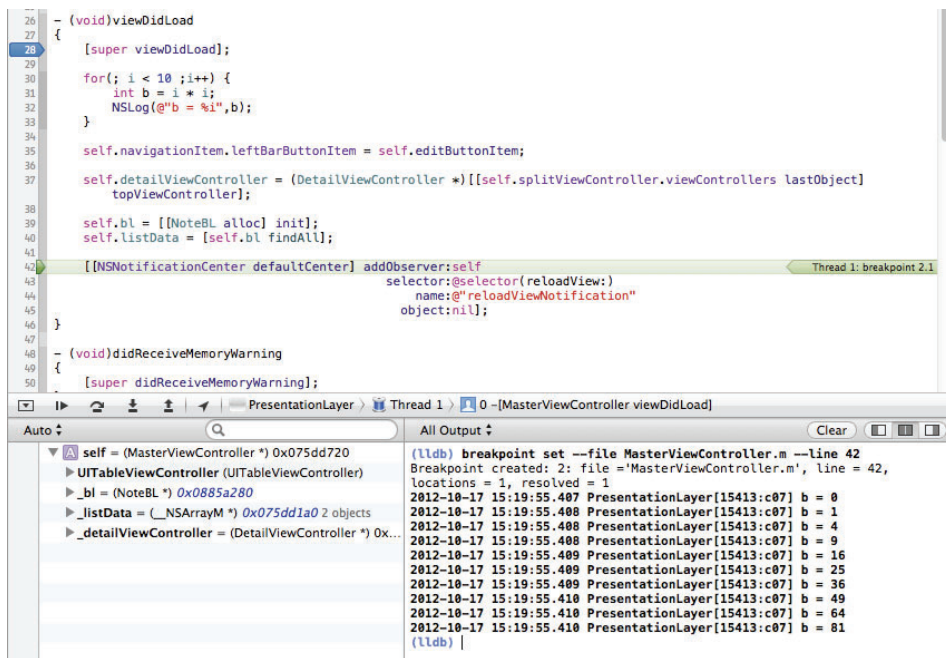


图15-36 测试breakpoint set命令

2. 查看断点

断点设置太多了之后，就需要查看一下断点设置情况，此时可以使用的命令如下：

```
(lldb) breakpoint list
(lldb) br 1
```

这两条命令都可以实现同样的效果，后者是简略写法。测试这个命令的结果如图15-37所示，其中有3个断点，每个断点前面有标号，1号断点通过Xcode常规方式设置，2号断点使用**b MasterViewController.m:12**命令设置，3号断点使用**br s -S findAll**命令设置。每个断点后面都有一些关于本断点的描述。

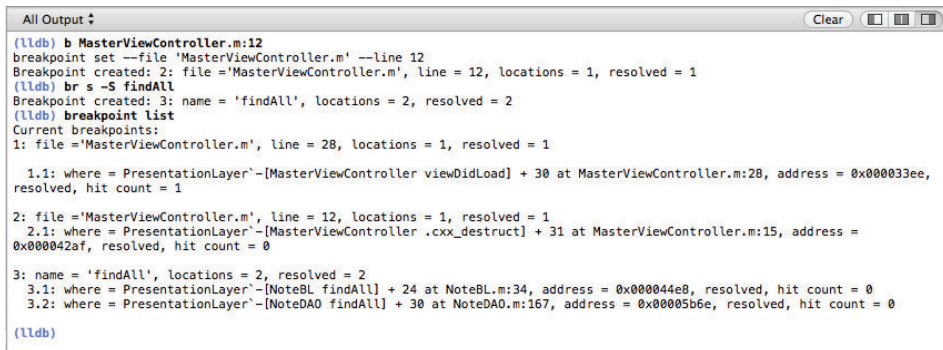


图15-37 查看断点命令

注意 使用breakpoint set设置的断点不能显示在Xcode工具的断点导航面板中，也不能通过Xcode来管理。只有通过Xcode常规方式设置的断点才可以显示在断点导航面板中。

3. 删除断点

breakpoint set 设置的断点，需要使用如下命令删除：

```
(lldb) breakpoint delete 断点编号
(lldb) br del 断点编号
```

这两条命令都可以实现同样的效果，后者是简略写法。这个命令很简单，大家可以自己测试一下。

4. 单步进入

有时候，我们需要在挂起状态下单步执行，单步执行分为单步进入和单步跳出。单步进入就是能够进入到方法或者函数中，该命令有基于源代码级别和基于指令级别两种类型，源代码级别是在源代码进行单步执行，指令级别是在汇编指令中单步执行。这里我们只介绍源代码级别单步进入，它的命令如下：

```
(lldb) thread step-in
(lldb) step
(lldb) s
```

这3条命令都可以实现同样的效果，后两个是简略写法。第一条命令中有thread，这说明单步进入命令是在同一个线程中单步运行，不涉及跨线程问题。

5. 单步跳过

单步跳过与单步进入类似，它遇到方法或函数时不会进入，而是直接跳过。该命令也有基于源代码级别和基于指令级别两种类型。源代码级别的单步跳过命令如下：

```
(lldb) thread step-over
(lldb) next
(lldb) n
```

这3条命令都可以实现同样的效果，后两个是简略写法。

6. 继续运行

继续运行与单步跳过和单步进入类似。执行该命令时，程序会运行到下一个断点或结束。该命令也有基于源代码级别和基于指令级别两种类型。源代码级别的单步跳过命令如下：

```
(lldb) thread continue
(lldb) continue
(lldb) c
```

这3条命令都可以实现同样的效果，后两个是简略写法。

15.3.2 观察点命令

在程序中对某个要观察的变量设置一个观察点，当这个变量变化时程序就会挂起。观察点与断点很相似，只是触发条件不同。下面我们分别介绍设置观察点、查看观察点和删除观察点这3个命令。

1. 设置观察点

在PresentationLayer中，我们先看看MasterViewController.m中viewDidLoad方法的代码：

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

    for(; i < 10 ;i++) {
        int b = i * i;
        NSLog(@"b = %i",b);
    }
    .....
}
```

如果将循环体变量b设置为观察点，相关命令如下：

```
(lldb) watchpoint set variable b
(lldb) wa s v b
```

这两条命令都可以实现同样的效果，后者是简略写法。为变量设置观察点时，变量不能超出它的作用域，变量b的作用域是for循环体，否则命令会出现下面的错误：

```
error: no variable or instance variable named 'b' found in this frame
```

2. 查看观察点

设置的观察点较多时，我们需要查看一下观察点设置情况，使用的命令如下：

```
(lldb) watchpoint list
(lldb) watch 1
```

这两条命令都可以实现同样的效果，后者是简略写法。测试这个命令的结果如图15-38所示，可以看到这里只有一个观察点，编号为1。

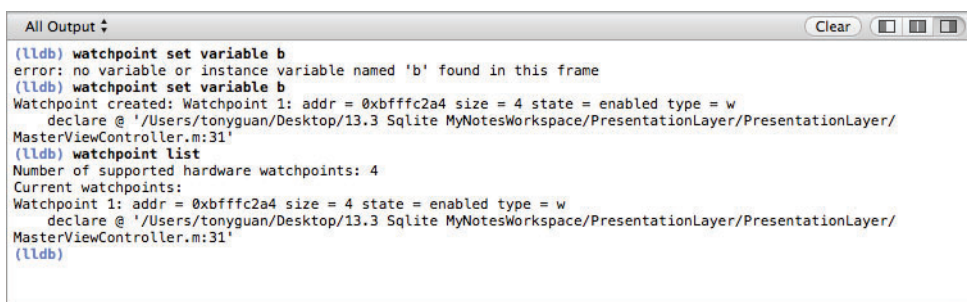


图15-38 查看观察点命令

3. 删除观察点

使用watchpoint set设置的观察点可以使用如下命令删除：

```
(lldb) watchpoint delete 观察点编号
(lldb) watch del 观察点编号
```

这两条命令都可以实现同样的效果，后者是简略写法。这个命令很简单，大家可以自己测试一下。

15.3.3 查看变量和计算表达式命令

使用调试工具时，少不了查看变量和计算表达式，下面我们介绍几个有关的命令，包括frame、target、expr和print等。

1. 查看本地变量

使用如下命令可以查看当前堆栈帧^①的所有本地变量：

```
(lldb) frame variable
(lldb) fr v
```

这两条命令都可以实现同样的效果，后者是简略写法。我们测试一下命令，得到的结果如图15-39所示。可以发现，当程序执行到第31行后挂起，然后在输出窗口中看到的是frame variable。

如果我们只想看看某个具体的变量，可以使用下面的命令：

```
(lldb) frame variable bar
(lldb) fr v bar
(lldb) p bar
```

^① 堆栈帧表示对当前线程调用堆栈的一个函数调用。

其中bar是变量名。这3条命令都可以实现同样的效果，第二条语句是第一条语句的简略写法，第三条语句p bar是print bar的缩写，其中print命令用于打印和计算表达式。

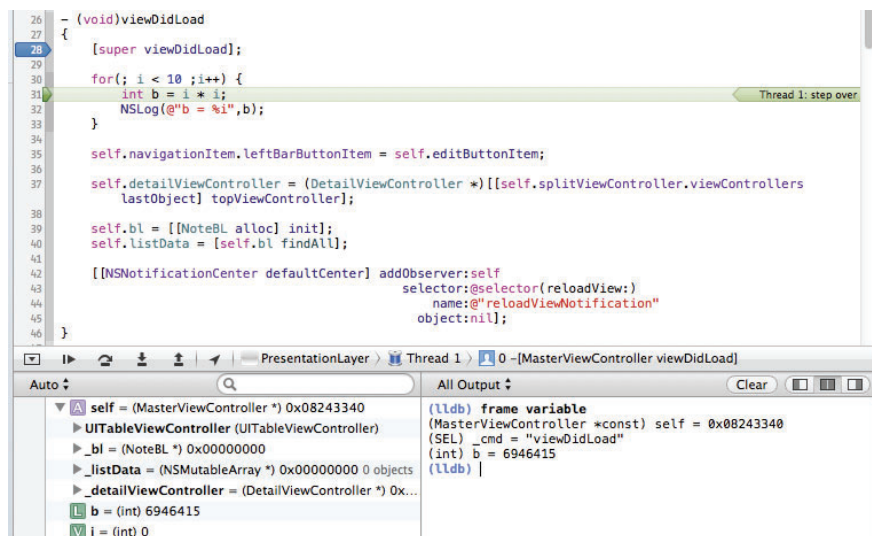


图15-39 测试frame命令

2. 查看全局变量

要查看全局变量，可以使用下面的命令：

```
(lldb) target variable
(lldb) ta v
```

这两条命令都可以实现同样的效果，后者是简略写法。我们测试一下这个命令，运行结果如图15-40所示。可以发现，程序执行到第31行后挂起，在输出窗口输入的是target variable。



图15-40 测试target命令

如果只是想看看某个具体变量，可以使用下面的命令：

```
(lldb) target variable baz
(lldb) ta v baz
```

其中baz是变量名。这两条命令都可以实现同样的效果，第二条语句是第一条语句的简略写法。

3. 计算基本数据类型表达式

计算表达式在调试时是很重要的，这里我们先看看基本数据类型构成的表达式的计算。为了计算程序运行到MasterViewController.m中第32行时表达式*i * i*的结果，我们可以在第32行设置断点，如图15-41所示。

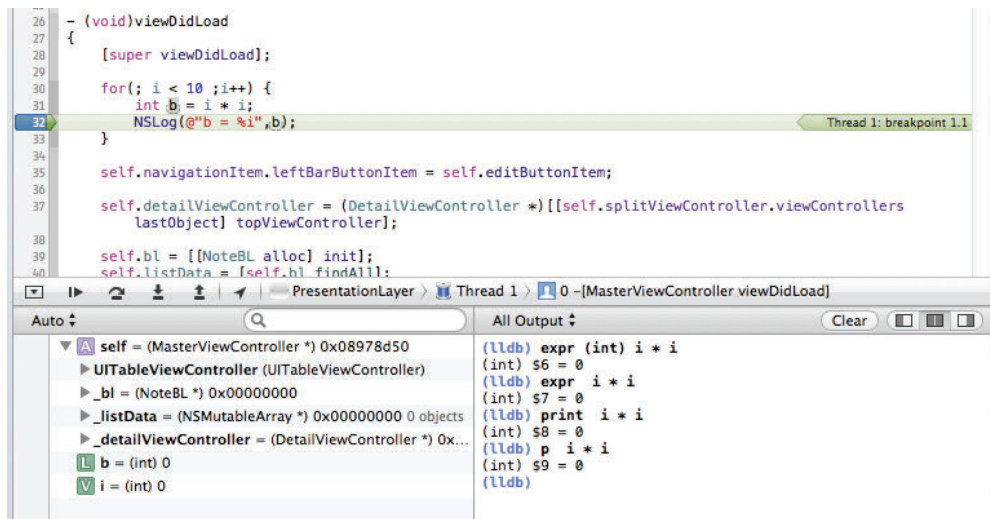


图15-41 计算基本数据类型表达式

为了计算*i * i*表达式，我们可以使用下面这几个命令：

```
(lldb) expr (int) i * i
(lldb) expr i * i
(lldb) print i * i
(lldb) p i * i
```

expr和print都可以用来计算表达式，并且print还可以省略为p。

4. 计算对象数据类型表达式

如果我们想计算程序运行到MasterViewController.m中第42行时listData数组中第一个元素的date属性，可以在第42行设置断点，如图15-42所示。

为了计算listData属性中第一个元素Note对象的date属性表达式，我们可以使用下面的几个命令：

```
(lldb) expr -o -- ((Note*)self.listData[0]).date
(NSDate *) $15 = 0x0881f990 2012-10-15 03:15:14 +0000
(lldb) po ((Note*)self.listData[0]).date
(NSDate *) $16 = 0x0881f990 2012-10-15 03:15:14 +0000
```

表达式中的self.listData[0]是取得NSArray数组的第一个元素，为了使其是一个合法的表达式，需要使用强制类型转换((Note*)self.listData[0])，然后再调用date属性。\$15 = 0x0881f990 2012-10-15 03:15:14 +0000用于将表达式计算的结果保存在\$15变量中，它的地址是0x0881f990，内容是2012-10-15 03:15:14 +0000。

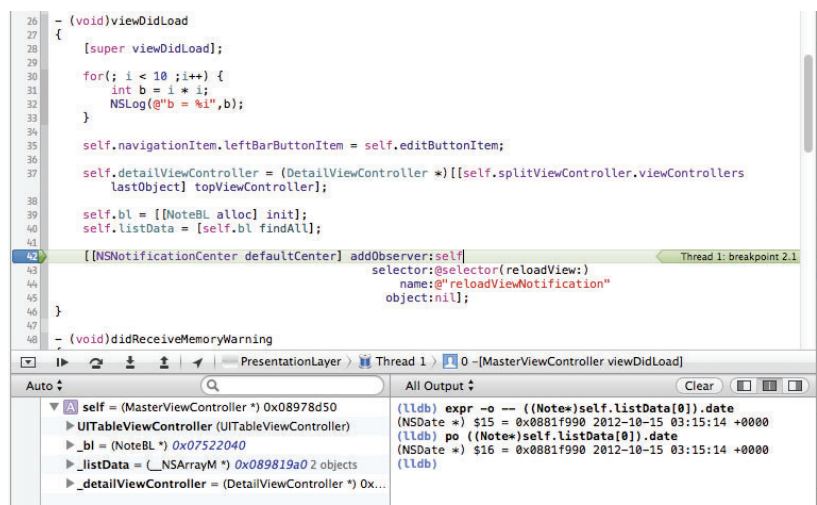


图15-42 计算对象数据类型的表达式

15.4 异常堆栈报告分析

在使用Xcode工具的开发过程中，面对运行异常，很多初学者往往毫无头绪，不知道如何跟踪异常堆栈、如何分析异常堆栈报告。本节中，我们将介绍如何跟踪异常堆栈和分析堆栈报告。

提示 异常堆栈是程序抛出异常之前，对象之间方法（或函数）调用的“路径”，它是程序运行的“黑匣子”。异常堆栈的输出内容包括很多信息，如调用顺序、方法所属框架（或库）、方法所属类、方法（或函数）地址等。

15.4.1 跟踪异常堆栈

默认情况下，使用Xcode工具进行开发时，产生异常时会有信息输出，也会有异常堆栈输出，不过它们都是晦涩难懂的内存地址，我们需要单独添加程序代码来处理这些信息。

为了学习这些知识点，我们使用10.4节的案例。在PresentationLayer中，修改一下MasterViewController.m中的viewDidLoad方法，制造一个运行期异常：

```
(void)viewDidLoad
{
    [super viewDidLoad];
    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    self.detailViewController = (DetailViewController *)
        [[self.splitViewController.viewControllers lastObject] topViewController];

    self.bl = [[NoteBL alloc] init];
    self.listData = [self.bl findAll];

    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(reloadView:)
        name:@"reloadViewNotification"
        object:nil];
}
```

原来的语句是：

```
[[NSNotificationCenter defaultCenter] addObserver:self
                                         selector:@selector(reloadView:)
                                         name:@"reloadViewNotification"
                                         object:nil];
```

修改为：

```
[[NSNotificationCenter defaultCenter] addObserver:self
                                         selector:@selector(reloadView1:)
                                         name:@"reloadViewNotification"
                                         object:nil];
```

也就是reloadView1:方法是不存在的。当点击添加功能中的保存操作时，程序运行到这里就会抛出异常。默认情况下，运行产生异常时，输出到输出窗口的内容如下：

```
2012-10-17 21:33:49.557 PresentationLayer[2424:c07] -[MasterViewController
reloadView1:]: unrecognized selector sent to instance 0xd44cad0
2012-10-17 21:33:49.558 PresentationLayer[2424:c07] *** Terminating app
due to uncaught exception 'NSInvalidArgumentException', reason:
'-[MasterViewController reloadView1:]: unrecognized selector sent to
instance 0xd44cad0'
*** First throw call stack:
(0x2099012 0x11a6e7e 0x21244bd 0x2088bbc 0x208894e 0xc67dc9 0x20f30c5
0x204defa 0xb9c482 0x2325 0x11ba705 0xf1920 0x32db24 0x11ba705
0xf1920 0xf18b8 0x1b2671 0x1b2bcf 0x1b1d38 0x12133f 0x121552 0xff3aa
0xf0cf8 0x1ff4df9 0x1ff4ad0 0x200ebf5 0x200e962 0x203fbb6 0x203ef44
0x203eelb 0x1ff37e3 0x1ff3668 0xee65c 0x1eed 0x1e15)
libc++abi.dylib: terminate called throwing an exception
```

这些日志由系统输出，共两条日志，第一条是异常描述信息，表示不能识别MasterViewController中的reloadView1方法。第二条是异常堆栈信息，除了第一行外，其他的都是十六进制数据，根本不知道是什么意思。异常堆栈信息对于我们基本上没有用，或许“超人”才需要它们。

将十六进制数据翻译成我们能够理解的异常堆栈信息的过程叫做symbolicate（符号化）。Xcode提供了symbolicatecrash工具来符号化，但使用起来比较麻烦。一般情况下，我们采用在程序中添加一小段代码实现符号化。修改一下PresentationLayer中的main.m代码：

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

int main(int argc, char *argv[])
{
    @try {
        @autoreleasepool {
            return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));
        }
    } @catch (NSEException *exception) {
        NSLog ( @"Stack Trace: %@", [exception callStackSymbols ] );
    }
}
```

我们知道，main.m是程序的逻辑入口点。事实上，所有的异常最后都要抛到这里，如果这里不处理，就会由系统采用默认的处理方式。我们在进行异常捕获，然后处理输出异常堆栈信息，其中的exception是异常对象，callStackSymbols方法可以处理异常堆栈。我们再次运行抛出异常，输出窗口的代码如下：

```
2012-10-17 22:12:05.501 PresentationLayer[2494:c07] -[MasterViewController
reloadView1:]: unrecognized selector sent to instance 0x76632c0
2012-10-17 22:12:05.506 PresentationLayer[2494:c07] Stack Trace: (
0   CoreFoundation                0x0209902e __exceptionPreprocess + 206
1   libobjc.A.dylib                0x011a6e7e objc_exception_throw + 44
2   CoreFoundation                0x021244bd -[NSObject(NSObject)
```

```

    doesNotRecognizeSelector:] + 253
3  CoreFoundation 0x02088bbc ____forwarding__ + 588
4  CoreFoundation 0x0208894e __CF_forwarding_prep_0 + 14
5  Foundation      0x00c67dc9 __57-[NSNotificationCenter
    addObserver:selector:name:object:]_block_
    invoke_0 + 40

6  CoreFoundation 0x020f30c5 ____CFXNotificationPost
    _block_invoke_0 + 85
7  CoreFoundation 0x0204defa __CFXNotificationPost + 2122
8  Foundation      0x00b9c482 -[NSNotificationCenter
    postNotificationName:object:userInfo:]
    + 98

9  PresentationLayer 0x00002225 -[AddViewController
    onclickSave:] + 501
10 libobjc.A.dylib 0x011ba705 -[NSObject performSelector:
    withObject:withObject:] + 77
11 UIKit           0x000f1920 -[UIApplication sendAction:
    to:from:forEvent:] + 96
12 UIKit           0x0032db24 -[UIBarButtonItem(UIInternal)
    _sendAction:withEvent:] + 139
13 libobjc.A.dylib 0x011ba705 -[NSObject performSelector:
    withObject:withObject:] + 77
14 UIKit           0x000f1920 -[UIApplication sendAction:
    to:from:forEvent:] + 96
15 UIKit           0x000f18b8 -[UIApplication sendAction:
    toTarget:fromSender:forEvent:] + 61
16 UIKit           0x001b2671 -[UIControl sendAction:
    to:forEvent:] + 66
17 UIKit           0x001b2bcf -[UIControl(Internal)
    _sendActionsForEvents:withEvent:] + 578
18 UIKit           0x001b1d38 -[UIControl touchesEnded:
    withEvent:] + 546
19 UIKit           0x0012133f -[UIWindow
    _sendTouchesForEvent:] + 846
20 UIKit           0x00121552 -[UIWindow sendEvent:] + 273
21 UIKit           0x000ff3aa -[UIApplication sendEvent:]
    + 436
22 UIKit           0x000f0cf8 _UIApplicationHandleEvent
    + 9874
23 GraphicsServices 0x01ff4df9 _PurpleEventCallback + 339
24 GraphicsServices 0x01ff4ad0 PurpleEventCallback + 46
25 CoreFoundation  0x0200ebf5 __CFRunLoop_IS_CALLING_OUT_TO__
    A_SOURCE1_PERFORM_FUNCTION__ + 53
26 CoreFoundation  0x0200e962 __CFRunLoopDoSource1 + 146
27 CoreFoundation  0x0203fbb6 __CFRunLoopRun + 2118
28 CoreFoundation  0x0203ef44 CFRunLoopRunSpecific + 276
29 CoreFoundation  0x0203ee1b CFRunLoopRunInMode + 123
30 GraphicsServices 0x01ff37e3 GSEventRunModal + 88
31 GraphicsServices 0x01ff3668 GSEventRun + 104
32 UIKit           0x000ee65c UIApplicationMain + 1211
33 PresentationLayer 0x00001d2d main + 157
34 PresentationLayer 0x00001c45 start + 53
)

```

这就是异常堆栈的信息了，这个内容比系统默认的信息要容易理解得多。

15.4.2 分析堆栈报告

下面我们介绍一下如何分析堆栈报告，它里面的内容是什么含义。

一条堆栈信息由5部分构成，如图15-43所示。

❑ 第①部分：堆栈输出序号，序号越大表示越早被调用。

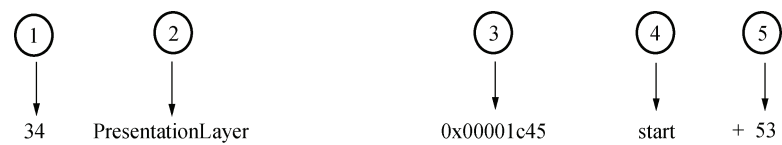


图15-43 堆栈信息的构成

- ❑ 第②部分：调用方法（或函数）所属的框架（或库），图中所示的PresentationLayer是我们自己编写的表示层工程。
- ❑ 第③部分：调用方法（或函数）的内存地址，这个信息对我们帮助不是很大。
- ❑ 第④部分：调用方法（或函数）名，这个信息对我们很重要。
- ❑ 第⑤部分：调用方法（或函数）编译之后的代码偏移量，这个信息很多人误认为是行号，对我们基本没有帮助。

此外，堆栈信息是要从下往上看，程序运行的过程是从下面的方法（或函数）调用项目的方法（或函数）。

例如，下面语句是PresentationLayer中的main函数调用UIKit中的 UIApplicationMain方法：

```
32 UIKit                                0x000ee65c UIApplicationMain + 1211
33 PresentationLayer                    0x00001d2d main + 157
```

在程序运行过程中，堆栈信息可能很长，我们不需要每一行都去看，只需关注我们自己的工程（或库）就可以了。这是因为首先我们要假定别人提供给我们的框架（或库）是正确的，先看自己的工程（或库）中的方法（或函数），找到那条调用语句看看是不是有问题。

这就是堆栈报告分析过程了。当然是否能够找出问题，还会因人而异，有调试经验的人可能很快就能找出问题。

15.5 在 iOS 设备上调试

所有的应用在发布之前一定要在iOS设备（真机）上调试，本节我们就来简要介绍一下。为了保证开发者和苹果的自身利益，防止非授权用户和设备使用，苹果对在iOS设备上调试的应用有着严格的限制，同时还需要一套复杂的操作。操作流程如图15-44所示，下面分别介绍一下该流程中的各环节。

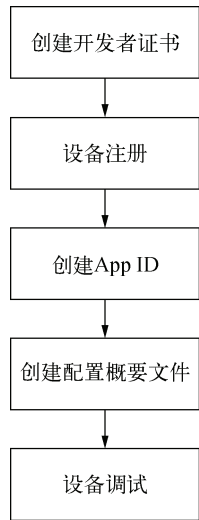


图15-44 设备调试流程

提示 应用在设备上调试和在App Store上发布都必须使用苹果开发者账号（App ID）。申请苹果开发者账号的个人开发者，所需的花费为99美元/年。

15.5.1 创建开发者证书

要想在iOS设备上调试应用程序，必须具有开发者证书。每个开发人员一次仅允许使用一个开发者证书。证书的管理可以登录iOS开发中心的配置门户网站（网址为https://developer.apple.com/ios/manage/overview/index.action）。登录该网站时，需要苹果的iOS开发者账号，登录成功后的界面如图15-45所示。

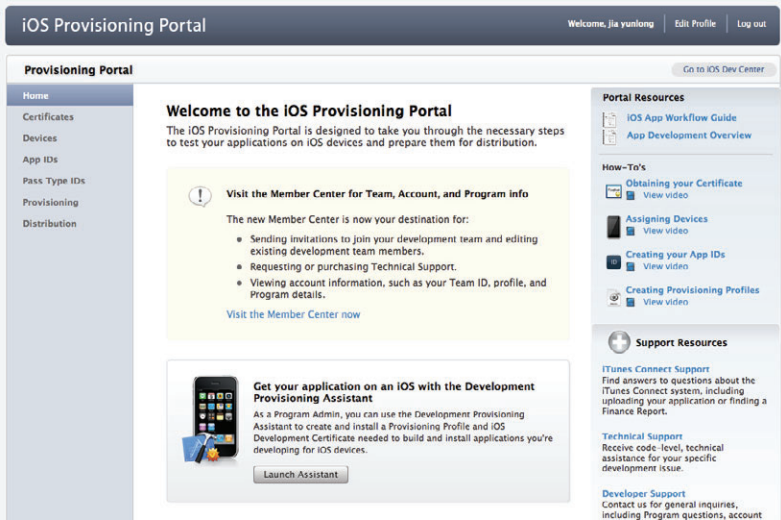


图15-45 登录iOS配置门户网站

点击左边的Certificates（证书）导航菜单，得到的证书管理界面如图15-46所示，在此处下载证书和删除证书。如果没有证书，则界面如图15-47所示。

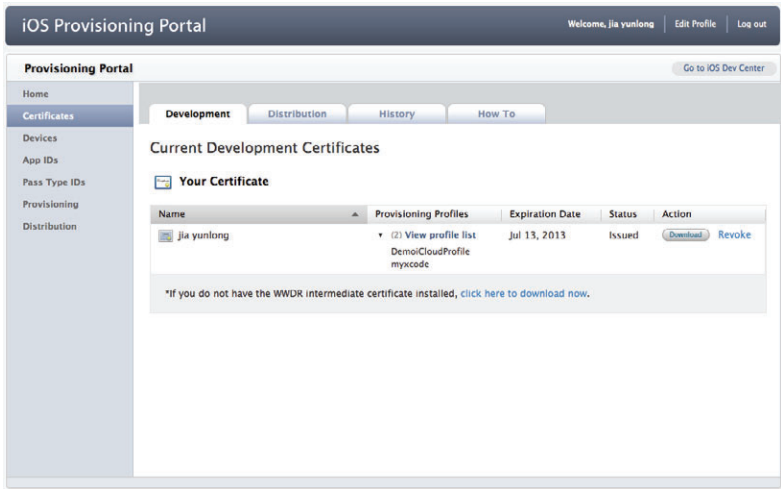


图15-46 证书管理界面

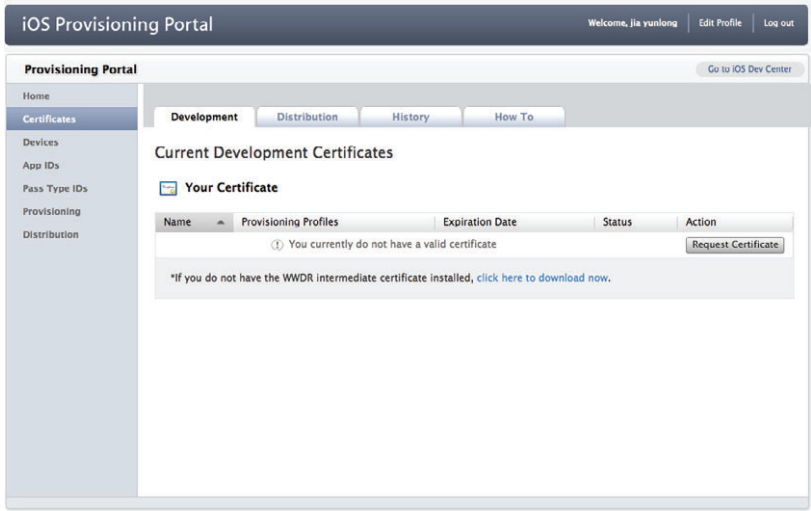


图15-47 无证书界面

如果没有证书，则需要创建证书。创建过程分成两步：

- ❑ 生成证书签名公钥；
- ❑ 提交证书公钥文件到配置门户网站。

1. 生成证书签名公钥

在安装有Mac OS X操作系统的苹果电脑中打开“应用程序”→“实用工具”→“钥匙串访问”，得到的界面如图15-48所示。

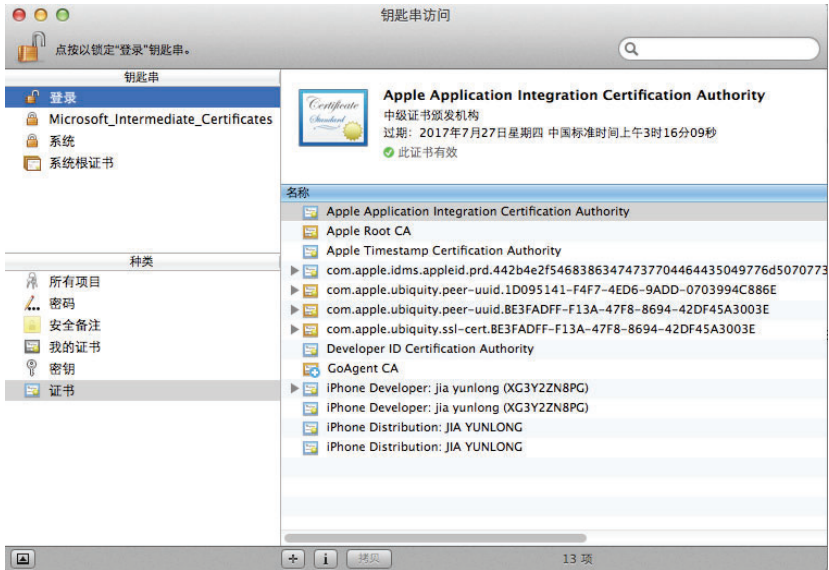


图15-48 钥匙串访问工具

选择“钥匙串访问”→“证书助理”→“从证书颁发机构请求证书”菜单项，此时弹出的对话框如图15-49所示，在“用户电子邮件地址”中输入eorient@sina.com，在“常用名称”中输入“eorient”，然后在“请求是”中选择“存储到磁盘”单选按钮。

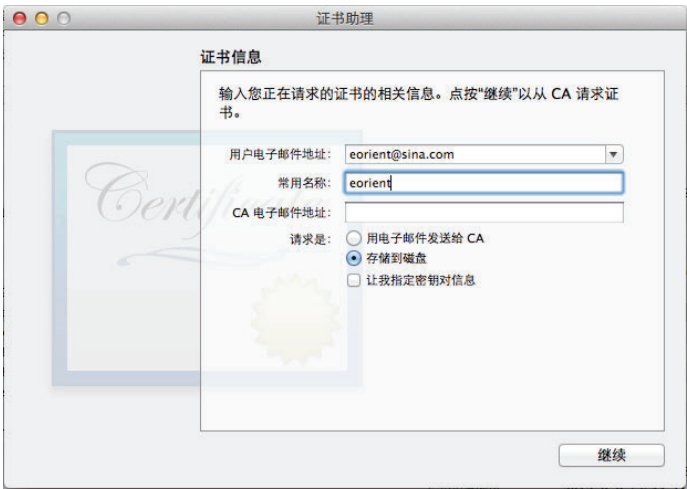


图15-49 证书助理信息

在图15-49所示的页面中输入信息后，点击“继续”按钮，会弹出如图15-50所示的证书签名公钥文件存储对话框，在这里我们可以修改文件名和存储位置。

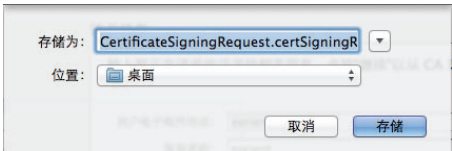


图15-50 证书签名公钥文件存储对话框

如果默认不修改，则点击“存储”按钮存储公钥文件，此时会在桌面上生成CertificateSigningRequest.certSigningRequest公钥文件。

2. 提交证书公钥文件到配置门户网站

生成CertificateSigningRequest.certSigningRequest公钥文件后，重新回到配置门户网站提交证书公钥文件。在图15-47所示的页面中请求开发者证书，具体操作为点击Request Certificate按钮，打开如图15-51所示的界面，从中选择Development标签。

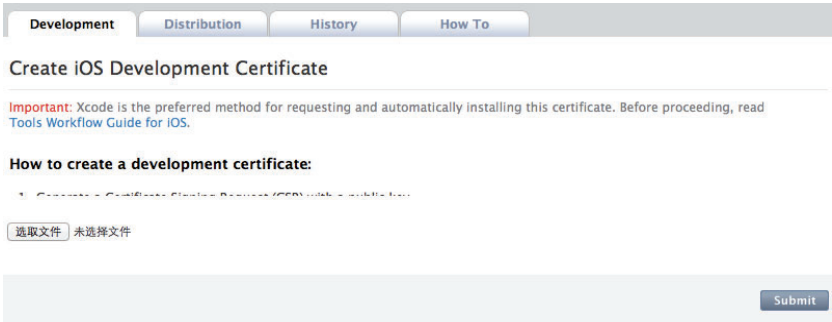


图15-51 Development标签页

在页面的最下面找到“选取文件”按钮，选取桌面上的CertificateSigningRequest.certSigningRequest文件，然后点击Submit按钮，添加完成后的界面如图15-52所示。

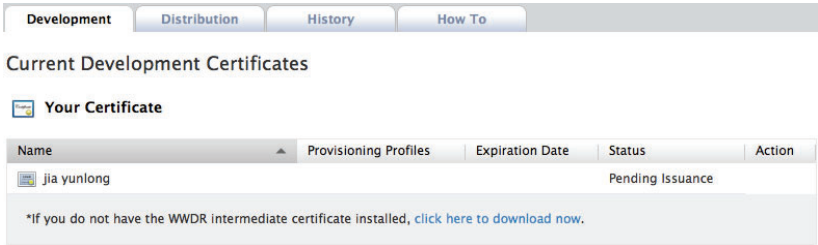


图15-52 提交证书签名公钥文件

图15-52只是一个过渡界面，稍等一会儿重新加载后，我们会看到如图15-53所示的界面。

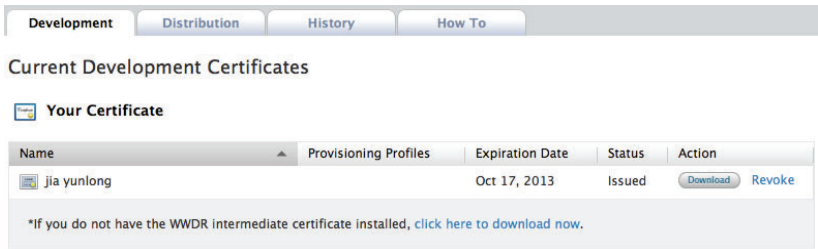


图15-53 提交成功

这样我们就成功创建了开发者证书。

15.5.2 设备注册

为了控制iOS设备的非法使用，苹果要求为调试的iOS设备进行注册。注册过程也是在配置门户网站完成的。点击左边的Devices导航菜单，如图15-54所示。

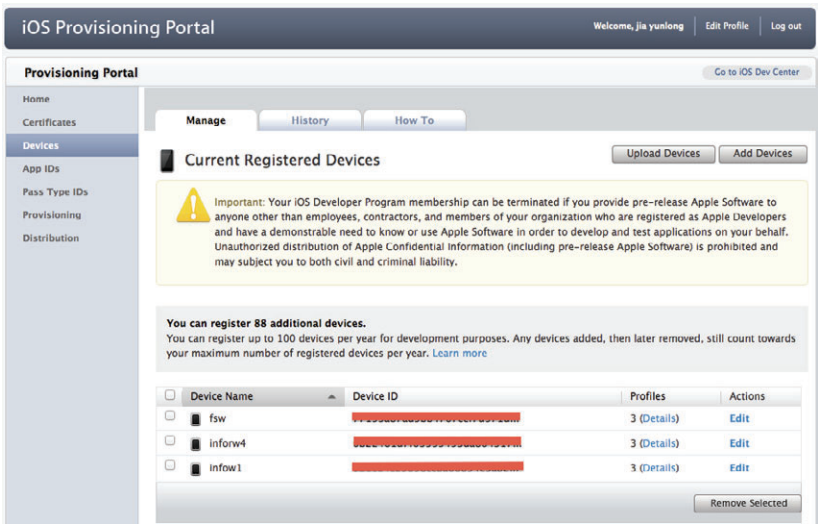


图15-54 设备注册

接着点击Add Devices按钮，此时会打开如图15-55所示的界面，从中输入设备名（Device Name）和设备ID（Device ID）。

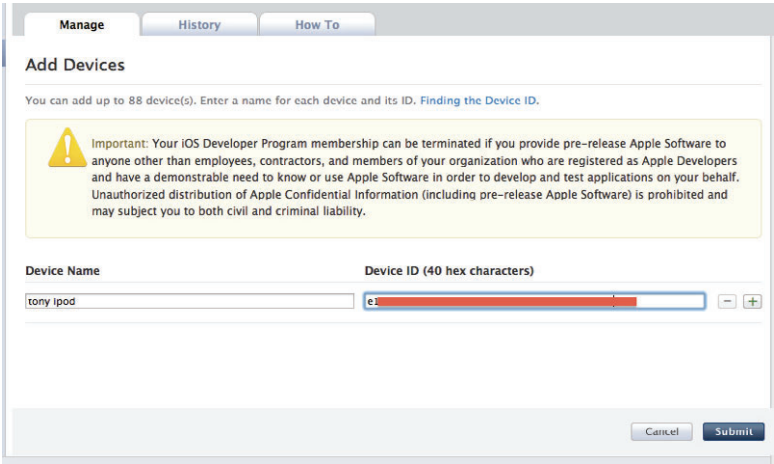


图15-55 添加设备

提示 把设备连接到iTunes中获得设备ID。如图15-56所示，设备信息默认显示的是序列号。点击序列号，它就会变成标识符（UDID）显示，这个标识符就是设备ID。



图15-56 获得设备ID

在图15-55所示的界面中输入完信息后，点击Submit按钮。添加设备后的界面如图15-57所示，其中tony ipod是我们刚才添加的设备。

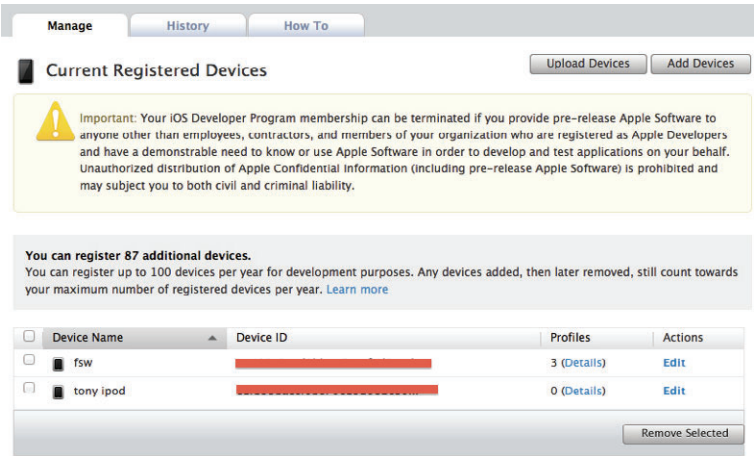


图15-57 注册完成的设备

15.5.3 创建App ID

设备注册成功后，还需要为应用创建App ID，该过程也是在配置门户网站完成的。点击左边的App IDs导航菜单，得到的界面如图15-58所示。

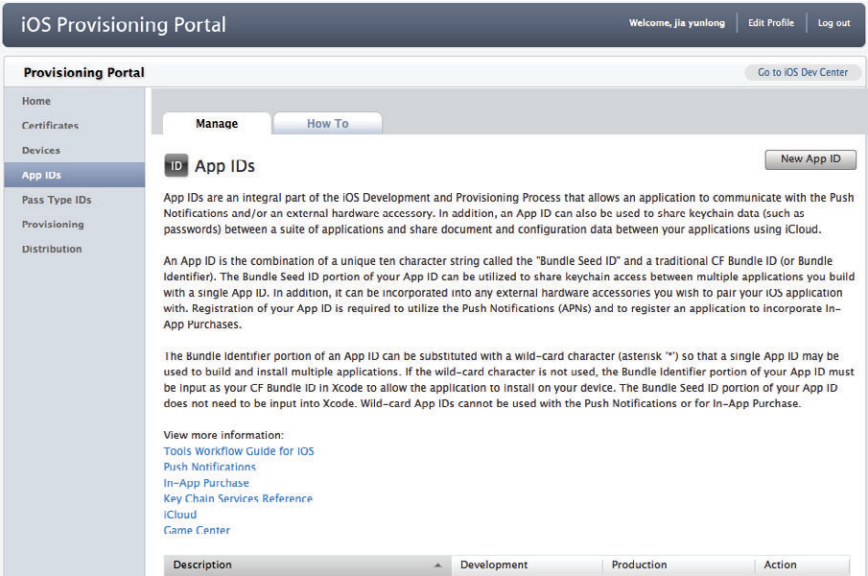


图15-58 创建App ID

接着点击New App ID按钮，此时将打开如图15-59所示的界面，其中有3个输入项，下面我们简要介绍一下。

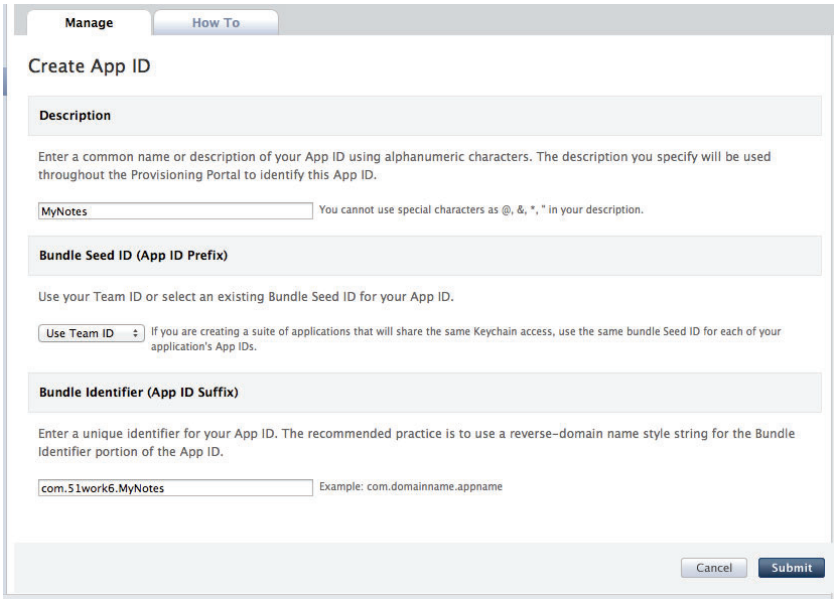


图15-59 创建App ID的详细页面

❑ Description。描述，可以输入一些描述应用的信息。

- ❑ Bundle Seed ID(App ID Prefix)。应用包种子ID，它作为应用的前缀，所描述的应用共享了相同的公钥。
- ❑ Bundle Identifier(App ID Suffix)。包标识符，它作为应用的后缀，苹果推荐使用域名反写。本例中输入的是com.51work6.MyNotes，与图15-60所示的应用程序TARGETS中设定的包标识符保持一致就可以了。

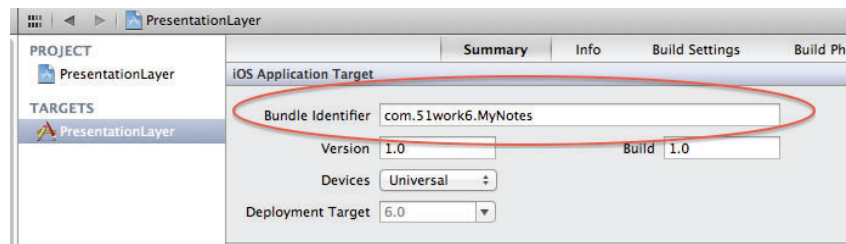


图15-60 应用程序TARGETS中设定的包标识符

注意 默认情况下，在PresentationLayer工程中，TARGETS的包标识符是com.51work6.PresentationLayer，我们需要修改 PresentationLayer 工程中 PresentationLayer-Info.plist 文件中的 Bundle identifier 项目，由 com.51work6.\${PRODUCT_NAME:rfc1034identifier} 修改为 com.51work6.MyNotes，如图15-61所示。

Key	Type	Value
Information Property List	Dictionary	(17 items)
Localization native development region	String	en
Bundle display name	String	\$(PRODUCT_NAME)
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	com.51work6.MyNotes
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL

图15-61 修改包标识符

在图15-59所示的界面中完成相应的信息输入后，点击Submit按钮提交信息，此时会跳转到创建App ID页面。在下面的App ID列表中，会发现我们刚刚创建的MyNotes，如图15-62所示。

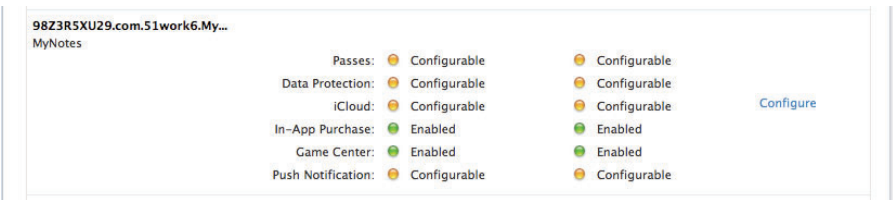


图15-62 创建完成的App ID

15.5.4 创建配置概要文件

配置概要文件是应用在设备上编译时使用的，分为开发配置概要文件和发布配置概要文件，分别用于开发（调试）和发布。管理配置概要文件的界面如图15-63所示，通过左边的Provisioning导航菜单进入，其中Development标签用于管理开发配置概要文件，Distribution标签用于管理发布配置概要文件。本节简要介绍一下创建开发配置概要文件的过程，发布配置概要文件的创建与此类似。

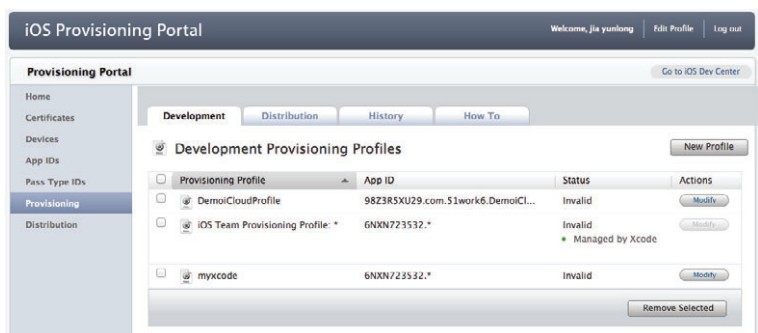


图15-63 管理配置概要文件界面

在图15-63所示的界面中点击New Profile按钮，进入创建配置概要文件界面，如图15-64所示。在Profile Name中输入MyNotes Profile，Certificates（证书）选择jia yunlong，这是我们的证书名，App ID选择MyNotes。Devices项目是这个配置概要文件所支持的设备，下面是注册的设备，这里勾选用于调试的设备。

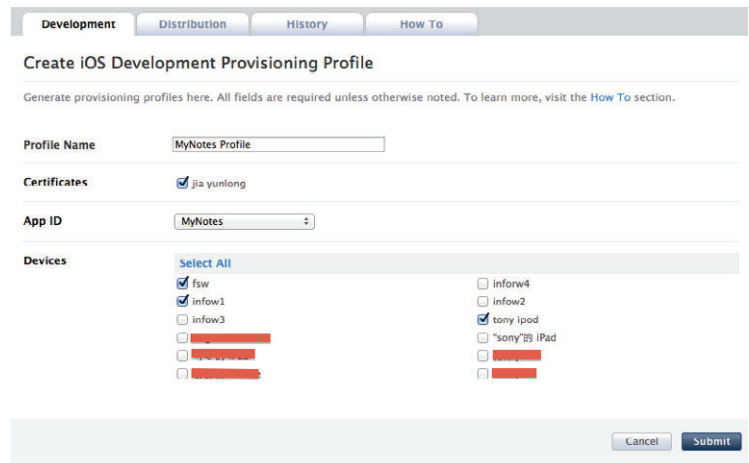


图15-64 创建配置概要文件界面

点击Submit按钮提交，此时界面跳转回管理配置概要文件界面。稍等一会儿刷新一下界面，得到的界面如图15-65所示，此时配置概要文件创建完毕。点击MyNotes Profile行后面的Download按钮，可以下载配置概要文件，点击Edit超链接可以进行界面修改。

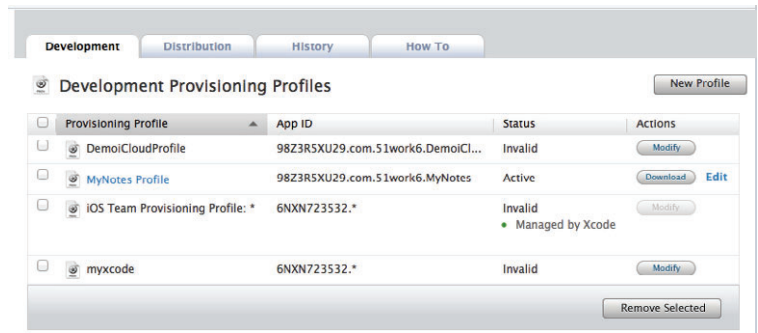


图15-65 创建完配置概要文件的界面

15.5.5 设备调试

为了能够实现设备调试，我们需要下载配置概要文件。下载成功后，双击该文件，进入Xcode设备管理工具，如图15-66所示，在这里可以管理这些配置概要文件。

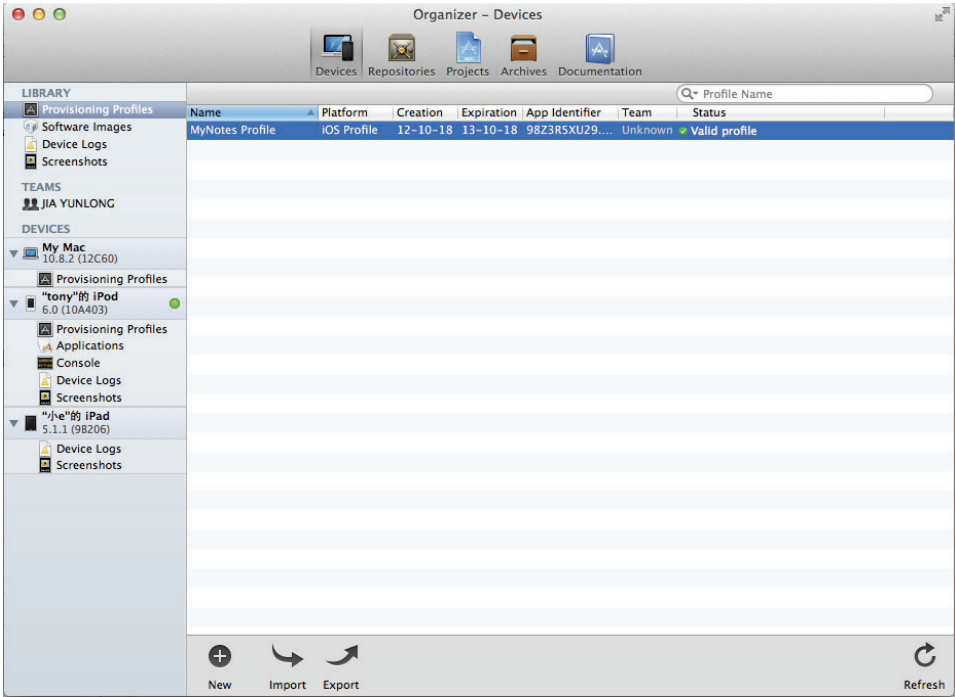


图15-66 Xcode设备工具

打开工作空间MyNotesWorkspace.xcworkspace，选择PresentationLayer工程中TARGETS下的PresentationLayer，依次点击Build Settings→Code Signing→Code Signing Identity，如图15-67所示，把Debug和Release的代码签名标识选择为MyNotes Profile。

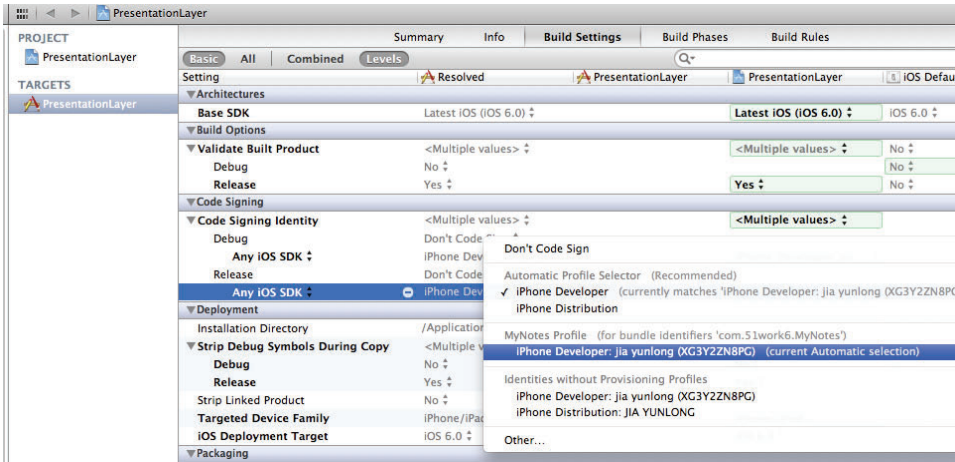


图15-67 选择代码签名标识

然后选择Scheme为PresentationLayer的“tony”的iPod”，如图15-68所示。接着编译工程，如果正常，则会编译成功。

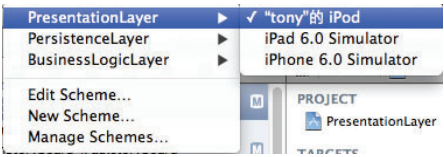


图15-68 选择Scheme

编译成功后，点击Xcode中的运行按钮，就可以在设备上运行应用了。

15.6 Xcode 设备管理工具

关于Xcode设备管理工具，我们在上一节中也用到了，在下载完配置概要文件后，双击该文件后进入的就是这个界面，但我们默认使用LIBRARY菜单下的Provisioning Profiles子菜单中打开。选择Window→Organizer→Devices菜单项，进入的界面如图15-69所示，此时选择“tony”的iPod”设备，右边会显示该设备的相关信息。



图15-69 “tony” 的iPod设备信息

在左边的设备菜单下面，有几个子菜单：配置概要文件（Provisioning Profiles）、应用（Applications）、控制台（Console）、设备日志（Device Logs）和屏幕快照（Screenshots）。在LIBRARY菜单下，也有配置概要文件（Provisioning Profiles）、设备日志（Device Logs）和屏幕快照（Screenshots）子菜单，它们包含了所有设备中的配置概要文件、设备日志和屏幕快照。

15.6.1 管理设备配置概要文件

使用Xcode设备上的管理工具，可以针对不同iOS设备来管理它们的配置概要文件。如图15-70所示，选择“tony”的iPod”中的设备配置概要文件，可以管理该设备的配置概要文件。

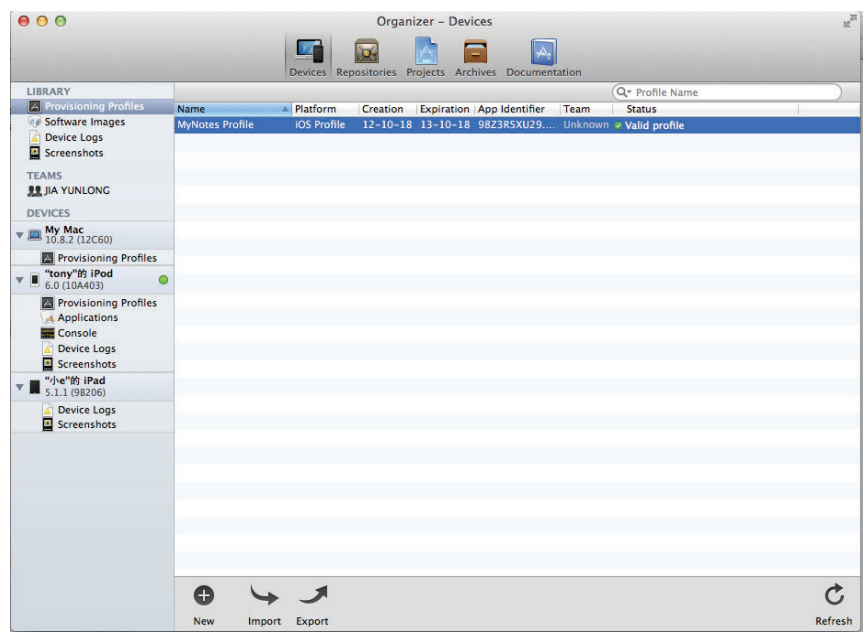


图15-70 管理特定设备的配置概要文件

15.6.2 查看设备上的应用程序

选择“‘tony’的iPod”中的Applications菜单，右侧会出现设备上调试过应用，如图15-71所示。目前，我们的设备上只有一个应用PresentationLayer 1.0。

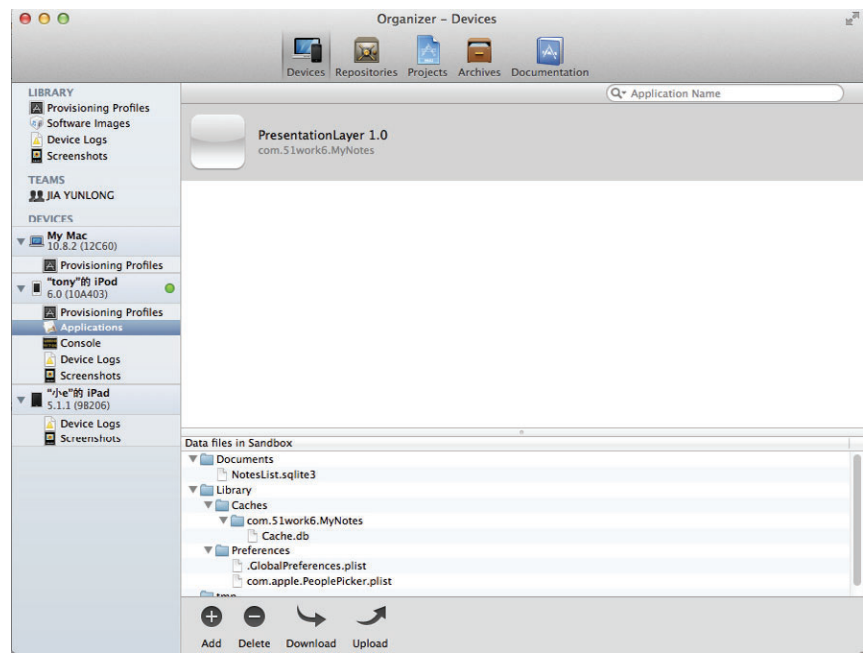
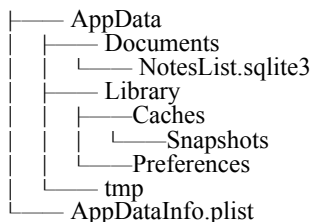


图15-71 设备上调试的应用

在图15-71的右下半部分区域中,是与设备上应用相对应的沙箱目录,我们可以在这里维护其中的数据。点击Download按钮,可以将设备应用沙箱目录中的数据下载到本地电脑。下载的数据类似于com.51work6.MyNotes 2012-10-18 16.27.45.186.xcappdata,其中xcappdata是包文件。选中该文件并右击,将会显示包内容,具体如下:



其中NotesList.sqlite3是应用中的SQLite数据文件。我们可以在终端窗口中使用sqlite3命令,查看NotesList.sqlite3中的数据。下面是查询Note表中数据的SQL语句:

```

$ sqlite3 NotesList.sqlite3
SQLite version 3.7.12 2012-04-03 19:43:07
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .tables
Note
sqlite> select * from Note;
2012-10-18 17:05:39|ab c
2012-10-18 17:13:07|x y z
sqlite>
  
```

当然,我们可以通过SQL语句修改数据,然后将其上传到设备中。如果我们使用如下的SQL语句插入两个数据:

```

sqlite> insert into Note (cdate,content) values ('2012-10-28 10:13:07','我的iOS');
sqlite> insert into Note (cdate,content) values ('2012-10-20 1:13:07','我的Java');
sqlite>
  
```

然后在图15-71所示的界面点击Upload按钮上传刚才修改的包文件,再重新运行设备上的PresentationLayer,得到的运行结果如图15-72所示。可以发现,在设备的界面中显示了我们刚刚插入的数据。

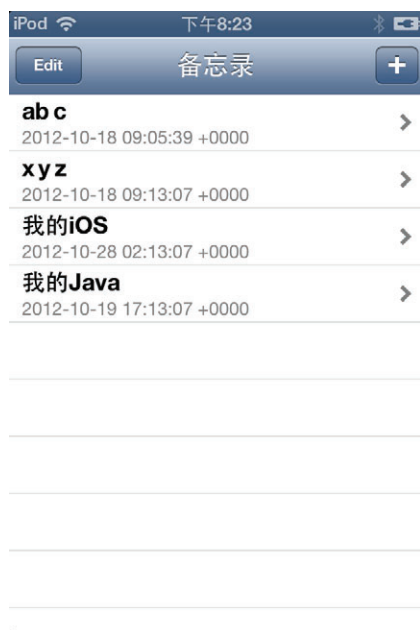


图15-72 插入的数据

15.6.3 设备控制台

选择“‘tony’的iPod”中的Console菜单，右侧界面会出现控制台上输出的信息，如图15-73所示。

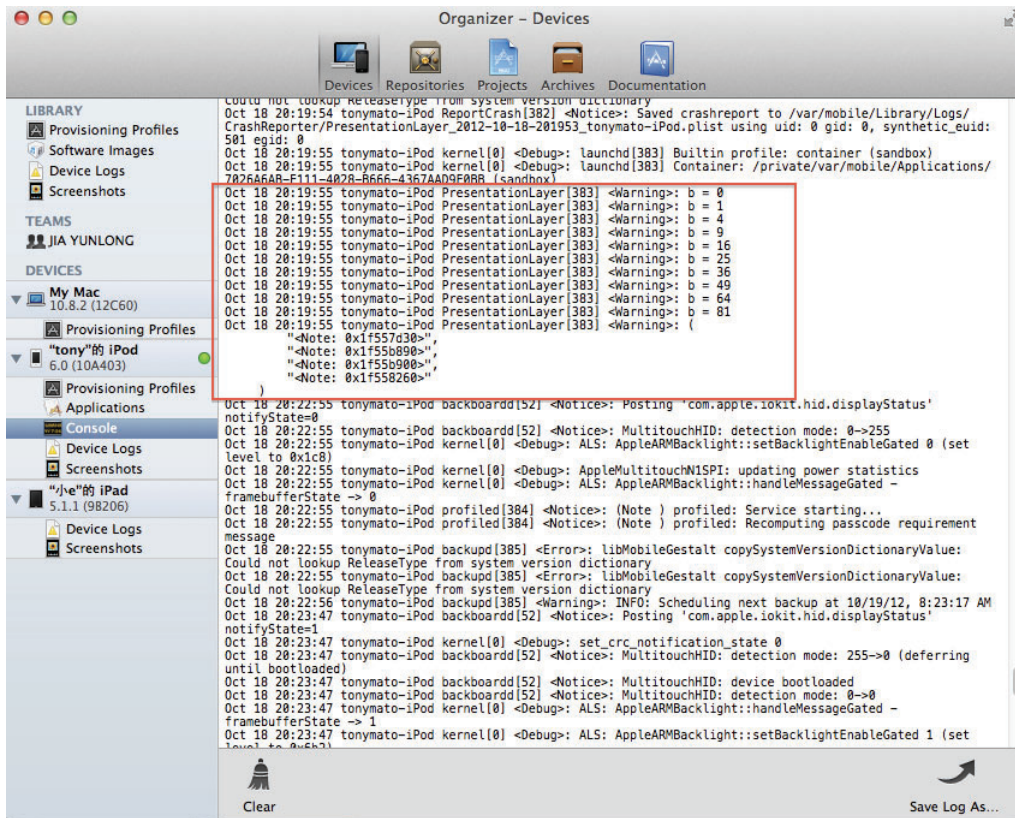


图15-73 控制台输出信息

右侧界面中的信息是由设备上的多个应用输出的，方框部分的信息是由PresentationLayer应用输出的。点击下面的Clear按钮，可以清除控制台输出的信息。点击Save Log AS...按钮，可以将控制台输出信息保存到文本文件中。

15.6.4 设备日志

选择“‘tony’的iPod”中的Device Logs项，右侧界面就会出现设备日志信息，如图15-74所示。

右边的设备日志信息部分分成左右两栏。左栏是日志标题，包括进程、类型和日志时间，其中的5条日志都是PresentationLayer应用产生的，类型是Crash代表应用崩溃（即产生异常）的输出日志。点击某个日志信息，右栏会显示详细的日志描述信息，其中包括跟踪异常堆栈的信息输出。

点击下面的Import按钮，可以导入日志信息。点击Export按钮，可以导出日志信息。Re-Symbolicate按钮用于重新符号化日志信息。由于设备输出的日志信息都是十六进制数据，如图15-75所示，此时点击Re-Symbolicate按钮，可以符号化日志信息。

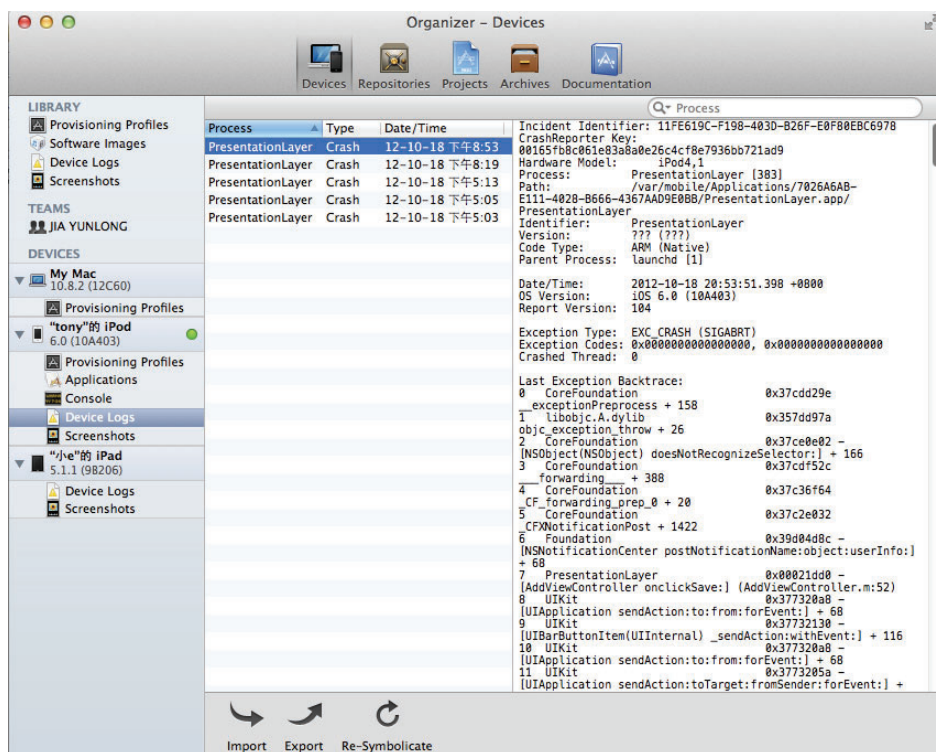


图15-74 设备日志信息

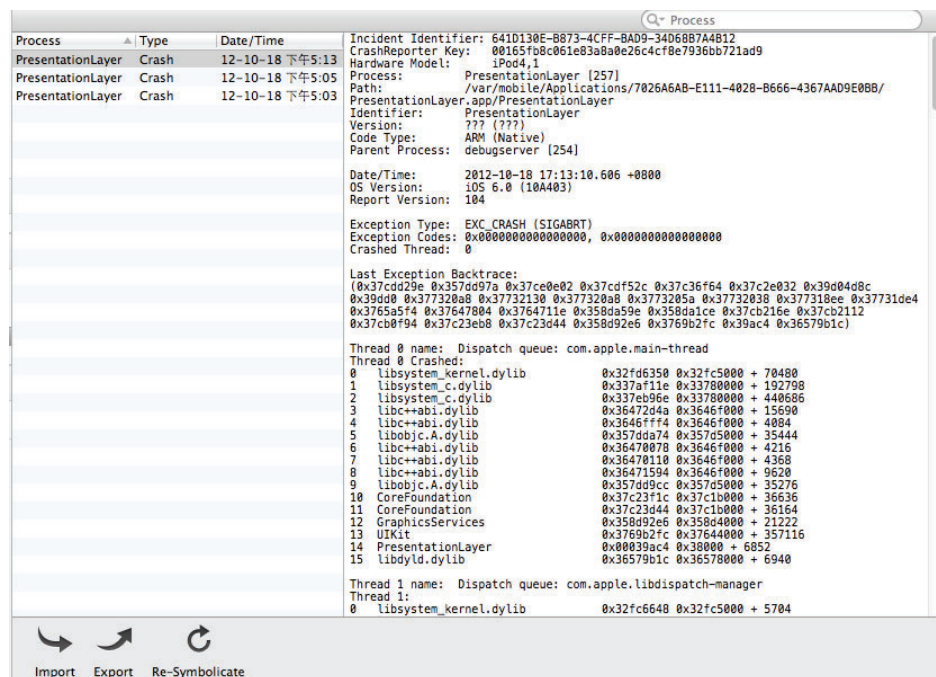


图15-75 未符号化的设备日志信息

关于符号化跟踪异常堆栈的问题，我们在15.4.1节中介绍过。通过在main.m的main函数中添加try-catch语句的方式实现此操作，这是在没有设备的情况下采用的方式。如果有设备，建议采用本节介绍的方式来跟踪异常堆栈，它的好处在于第一不用修改代码，其次设备日志输出信息更加详细。其中的一段日志信息如下：

```

0   CoreFoundation          0x37cdd29e __exceptionPreprocess + 158
1   libobjc.A.dylib         0x357dd97a objc_exception_throw + 26
2   CoreFoundation          0x37ce0e02 -[NSObject(NSObject)
    doesNotRecognizeSelector:] + 166
3   CoreFoundation          0x37cdf52c ____forwarding____ + 388
4   CoreFoundation          0x37c36f64 _CF_forwarding_prep_0 + 20
5   CoreFoundation          0x37c2e032 _CFXNotificationPost + 1422
6   Foundation              0x39d04d8c -[NSNotificationCenter postNotificationName:
    object:userInfo:] + 68
7   PresentationLayer       0x00021dd0 -[AddViewController onclickSave:]
    (AddViewController.m:52)
8   UIKit                   0x377320a8 -[UIApplication sendAction:to:
    from:forEvent:] + 68
9   UIKit                   0x37732130 -[UIBarButtonItem(UIInternal)
    _sendAction:withEvent:] + 116
10  UIKit                   0x377320a8 -[UIApplication sendAction:to:
    from:forEvent:] + 68
11  UIKit                   0x3773205a -[UIApplication sendAction:toTarget:
    fromSender:forEvent:] + 26
12  UIKit                   0x37732038 -[UIControl sendAction:to:
    forEvent:] + 40
13  UIKit                   0x377318ee -[UIControl(Internal) _sendActionsForEvents:
    withEvent:] + 498
14  UIKit                   0x37731de4 -[UIControl touchesEnded:withEvent:]
    + 484
15  UIKit                   0x3765a5f4 -[UIWindow _sendTouchesForEvent:]
    + 520
16  UIKit                   0x37647804 -[UIApplication sendEvent:] + 376
17  UIKit                   0x3764711e _UIApplicationHandleEvent + 6150
18  GraphicsServices        0x358da59e _PurpleEventCallback + 586
19  GraphicsServices        0x358da1ce PurpleEventCallback + 30
20  CoreFoundation          0x37cb216e __CFRUNLOOP_IS_CALLING_OUT_
    TO_A_SOURCE1_PERFORM_FUNCTION__ + 30
21  CoreFoundation          0x37cb2112 __CFRunLoopDoSource1 + 134
22  CoreFoundation          0x37cb0f94 __CFRunLoopRun + 1380
23  CoreFoundation          0x37c23eb8 CFRunLoopRunSpecific + 352
24  CoreFoundation          0x37c23d44 CFRunLoopRunInMode + 100
25  GraphicsServices        0x358d92e6 GSEventRunModal + 70
26  UIKit                   0x3769b2fc UIApplicationMain + 1116
27  PresentationLayer       0x00021ac4 main (main.m:16)
28  libdyld.dylib           0x36579b1c start + 0

```

注意看，第7行后面有个括号(AddViewController.m:52)，这说明调用发生在AddViewController.m中的第52行。我们可以和15.4.1节中第9行的代码比较一下：

```

9   PresentationLayer       0x00002225 -[AddViewController
    onclickSave:] + 501

```

我们发现后者没有行号，只有+501这个没有意义的偏移量。

15.7 小结

在本章中，我们介绍了iOS中都有哪些调试工具并重点介绍了几个常用的工具，具体包括日志与断言的输出、LLDB调试工具、异常堆栈报告分析；接下来讲解了如何在真机上调试应用，涉及开发者证书的创建、设备的注册、App ID的创建、配置概要文件的创建和设备调试；最后分析了Xcode设备管理工具的使用。

本章内容十分重要，希望大家能够好好消化这部分的知识。

为了找出程序中的错误与缺陷，需要对程序进行测试。测试也是保证产品质量、安全性和完整性的重要手段，更是软件开发生命周期的重要阶段。测试按照阶段划分为：单元测试、集成测试、系统测试和回归测试。单元测试是对软件组成单元进行测试，其目的是检验软件基本组成单位的正确性，其测试对象是软件设计的最小单位——模块。

单元测试是一种白盒测试。白盒测试是一种细粒度的测试，具体到方法、函数和异常测试，因此是由能够看懂编程语言、了解程序结构的程序员发起的。为了验证程序的正确性，程序员需要编写测试程序，按照测试用例测试程序是否能够有预期的结果。

提示 测试用例是一组条件和场景，根据它来确定程序的正确性。

软件工程方法强调，没有进行单元测试并给出单元测试报告的程序是不能提交给测试团队进行其他测试的，更不能发布应用程序，否则将是一场灾难！它使我们投入更多的时间，浪费更多的资源。

单元测试也是软件工程方法学的一个重要手段。近年来，有一种很流行的软件工程方法学——极限编程（eXtreme Programming，缩写为XP），它是一种敏捷软件开发方法，其实践核心是测试驱动开发（Test-driven development，缩写为TDD），或称为测试先行（Test First）。本章中，我们就围绕测试驱动这个主题介绍基于测试驱动开发的iOS软件开发。

16.1 测试驱动的软件开发概述

单元测试是测试驱动的核心，可以通过手动测试或者自动测试来完成，但自动测试需要使用测试控件。本节中，我们就来介绍单元测试、测试驱动的概念以及iOS单元测试框架。

16.1.1 测试驱动的软件开发流程

传统的开发流程如图16-1所示，先是程序编码，然后设计单元测试用例，编写单元测试程序，进行单元测试，最后出具单元测试报告。如果测试没有通过，要根据测试修改程序代码，然后再重新走单元测试流程。

而极限编程的测试驱动的软件开发流程如图16-2所示，先是设计单元测试用例程序，编写单元测试程序，然后编写程序代码和进行单元测试，最后出具单元测试报告。如果测试没有通过，根据测试修改程序代码，然后重新走单元测试流程。如果通过测试，再设计其他的单元测试用例。

在测试驱动开发流程中，各个阶段都是一个可逆的反复迭代过程。用例的设计可以先是功能说明书中的一个功能，然后针对该功能进行测试驱动的开发流程，再编写其他的功能。

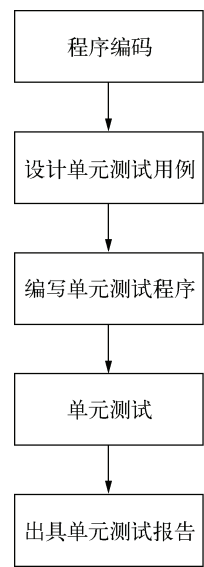


图16-1 传统的开发流程

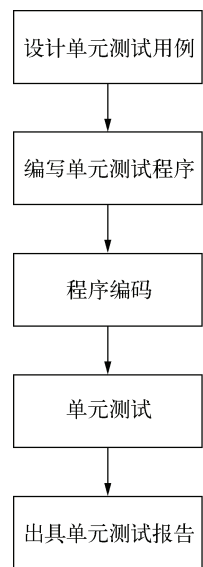


图16-2 测试驱动开发流程

比较这两种方式，我们可以发现测试驱动开发的优势很明显，它能够及时地发现程序中的问题，从而少犯错误，减少资源浪费。

16.1.2 测试驱动的软件开发案例

下面我们通过案例介绍测试驱动的软件开发流程。该案例是一个iPhone版的计算个人月工资和奖金所得税的简易工具。根据最新的个人所得税法，月工资和奖金所得税分成7个级别，如表16-1所示，起征点为3500元，计算公式为：

月应纳个人所得税税额=月应纳税所得额×适用税率－速算扣除数
其中，月应纳税所得额=工资和奖金－三险一金－起征点。在本案例中，三险一金我们设为0，起征点是3500。

表16-1 个人所得税的7个级别

月应纳税所得额	适用税率	速算扣除数（元）
月应纳税额不超过1500元	3%	0
月应纳税额在1500元至4500元之间	10%	105
月应纳税额在4500元至9000元之间	20%	555
月应纳税额在9000元至35000元之间	25%	1005
月应纳税额在35000元至55000元之间	30%	2755
月应纳税额在55000元至80000元之间	35%	5505
月应纳税额超过80000元	45%	13505

应用的设计原型草图如图16-3所示，用户可以在文本框中输入月收入总额，点击“计算”按钮，即可计算出月应纳个人所得税税额，并将其显示在下面的标签中。

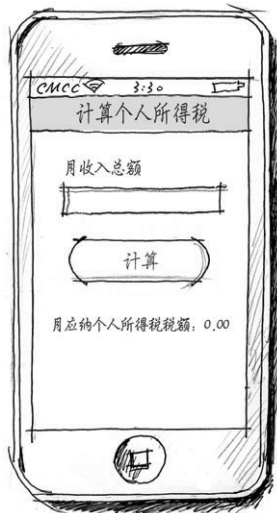


图16-3 计算个人所得税应用的设计原型草图

根据图16-2所示的测试驱动开发流程，我们介绍一下上述案例的开发过程。

1. 设计单元测试用例

测试用例就是一组条件。根据需求，我们设计了7个测试用例（如表16-2所示）。在输入条件中，我们采用常见值和边界值作为测试数据进行测试，从而来增加测试用例的测试覆盖率。

表16-2 个人所得税应用的单元测试用例

测试用例	输入条件月收入总额(元)	输出结果月应纳个人所得税税额（元）	说 明
1	5 000	45	测试月应纳税额不超过1500元
2	8 000	345	测试月应纳税额在1 500元至4 500元之间
3	12 500	1 245	测试月应纳税额在4 500元至9 000元之间
4	38 500	7 745	测试月应纳税额在9 000元至35 000元之间
5	58 500	13 745	测试月应纳税额在35 000元至55 000元之间
6	83 500	22 495	测试月应纳税额在55 000元至80 000元之间
7	103 500	31 495	测试月应纳税额超过80 000元

2. 编写单元测试程序

有了测试用例后，我们就可以编写测试程序了。不要着急，我们还要做一些前期准备工作，包括创建工程、基本的UI布局、动作事件处理和定义输出口等，这些工作的细节我们就不再介绍了。完成之后的工程是PITax，视图控制器ViewController.h的程序代码如下：

```
//
//ViewController.h
//PITax

#import <UIKit/UIKit.h>

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UITextField *txtRevenue;

@property (weak, nonatomic) IBOutlet UILabel *lblTax;
```

```

- (IBAction)onClick:(id)sender;

//计算个人所得税
-(NSString*) calculate:(NSString*)revenue;

@end

```

其中txtRevenue是“月收入总额”文本框的属性，lblTax是“月应纳个人所得税税额”标签的属性。onClick:是响应“计算”按钮动作事件的方法。calculate:是计算个人所得税的方法，其中revenue参数是输入条件——月收入总额，该方法的返回值是输出结果——月应纳个人所得税税额，这个方法我们需要重点测试。视图控制器ViewController.m的程序代码如下：

```

@implementation ViewController

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

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

- (IBAction)onClick:(id)sender {
    //关闭键盘
    [self.txtRevenue resignFirstResponder];
    self.lblTax.text =[self calculate:self.txtRevenue.text];
}

//计算个人所得税
-(NSString*) calculate:(NSString*)revenue {

    return nil;
}

@end

```

onClick:方法实现了放弃第一响应者来关闭键盘和调用calculate:方法计算个人所得税的功能。calculate:方法没有实际代码，暂时返回nil。

这些基本的准备工作完成后，我们看看如何编写单元测试代码。基于测试驱动的单元测试是迭代式的，因此先编写测试用例1。首先我们需要在工程中再创建一个新的目标用于测试，选择File→New→Target...菜单项，此时将打开如图16-4所示的对话框，在Product Name中输入PITaxTest。

创建完成后，请删除PITaxTest组中的AppDelegate.m和AppDelegate.h文件，并在PITaxTest组中创建PITaxTest类。PITaxTest.h中的代码如下：

```

#import "ViewController.h"

@interface PITaxTest : NSObject

@property (nonatomic,strong) ViewController *viewController;

- (void)setUp;

- (void)tearDown;

//用例1: 测试月应纳税额不超过1500元
- (void)testCalculateLevel1;

@end

```

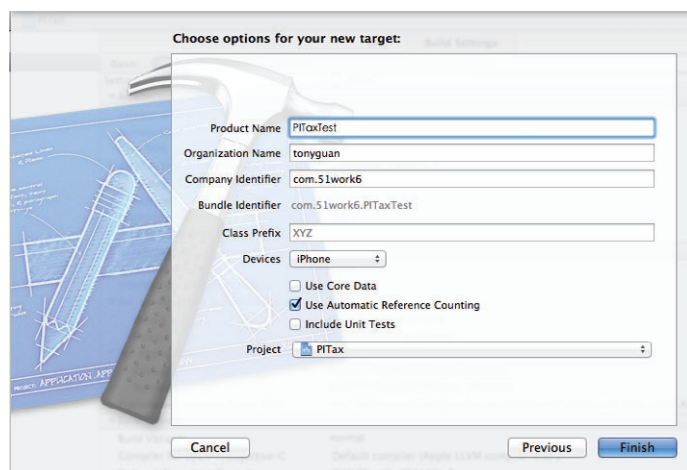


图16-4 创建测试用目标

属性viewController是要测试的ViewController视图控制器类型，注意属性参数是strong。setUp方法是初始化测试资源时使用的方法。tearDown与setUp相反，是释放资源时使用的方法。testCalculateLevel1方法就是测试方法，这个方法一般是test开头。如果使用了测试框架进行测试，以test开头是必需的，这是因为测试框架根据这个条件判断哪些是测试方法，哪些是普通方法。在测试过程中，这些以test开头的测试方法将被调用。一般而言，一个测试方法对应一个测试用例。这里我们先编写了一个测试用例，测试月应纳税额不超过1500元。

PITaxTest.m中的代码如下：

```
@implementation PITaxTest

- (void)setUp
{
    self.viewController = [[ViewController alloc] init];
}

- (void)tearDown
{
    self.viewController = nil;
}

//用例1: 测试月应纳税额不超过1500元
- (void)testCalculateLevel1
{
    double dbRevenue = 5000;
    NSString *strRevenue = [NSString stringWithFormat:@"%f",dbRevenue];
    NSString* strTax=[self.viewController calculate:strRevenue];

    if ([strTax doubleValue] == 45) {
        NSLog(@"testCalculateLevel1测试通过。");
    } else {
        NSLog(@"testCalculateLevel1测试失败, 期望值是: 45 实际值是: %@", strTax);
    }
}

@end
```

在上述代码中，我们在setUp方法中初始化测试类中的viewController属性，又在tearDown方法中将viewController属性设置为nil，从而释放资源。由于采用了ARC管理内存，我们不必在tearDown方法中release成员变量了。

testCalculateLevel1方法是测试方法，输入条件是5000，计算结果保存在变量strTax中。将这个变量转

换为double类型后,如果它等于45,则说明我们编写的程序在第一个测试用例下测试通过了,否则测试失败。在单元测试中,这个过程叫“断言”。当然,我们在这种情况下运行测试程序,断言一定是失败的,因为我们没有编写被测试程序。

接下来,我们需要修改PITaxTest\Supporting Files组中的main.m代码:

```
#import "PITaxTest.h"

int main(int argc, char *argv[])
{

    PITaxTest *tester = [[PITaxTest alloc] init];


    [tester setUp];

    [tester testCalculateLevel1];

    [tester tearDown];

}
```

上述代码是PITaxTest Target的程序入口,它调用测试类PITaxTest进行测试。

为了能够编译和运行测试程序,我们还需要选中PITax组中的ViewController.m文件,打开其文件检查器,如图16-5所示,找到Target Membership并勾选PITaxTest项,这样在PITaxTest编译时也会编译ViewController.m文件,否则编译PITaxTest时会有编译错误。

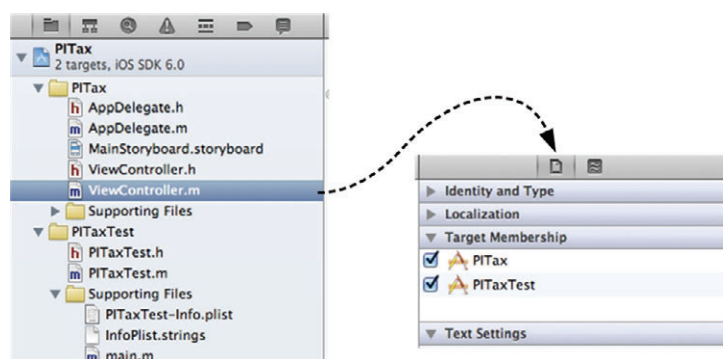


图16-5 文件检查器

3. 程序编码

测试程序编写完成后,再编写对应的被测试程序。注意我们不需要编写所有的代码,只需要编写测试用例1对应的程序代码即可。修改PITax组中ViewController.m的calculate:方法,其代码如下:

```
-(NSString*) calculate:(NSString*)revenue {

    //月应纳个人所得税税额
    double tax = 0.0f;

    if ([revenue length] > 0) {

        double dbRevenue = [revenue doubleValue];

        //月应纳税所得额
        double dbTaxRevenue = dbRevenue - 3500;

        //月应纳税所得额不超过1500元
        if (dbTaxRevenue <= 1500) {
            tax = dbTaxRevenue * 0.03;
        }
    }
}
```

```

        //月应纳税所得额在1500元至4500元之间
    }
}

NSString* strTax = [NSString stringWithFormat:@"%%.2f", tax<0?0:tax];
return strTax;
}

```

4. 单元测试

在用例1下的测试程序代码和被测试程序代码都编写完成后，我们需要进行测试，此时直接运行PITaxTest就可以了。选择Scheme中PITaxTest中的iPhone 6.0 Simulator，然后点击Run按钮运行，输出结果如下：

```
2012-10-23 22:47:36.228 PITaxTest[1435:c07] testCalculateLevel1测试通过。
```

5. 出具单元测试报告

测试报告一般会有测试成功、失败、异常或警告等几种信息。好的测试报告还会有测试一个方法所用的时间，测试失败时程序在哪里出了问题，期望的结果与实际结果的差别等。下面的信息是我们输出的，看起来信息比较少，如果想获得更多的信息，就需要使用测试框架了：

```
2012-10-23 22:49:37.598 PITaxTest[1456:c07] testCalculateLevel1测试失败，期望值是：45 实际值是：0.00。
```

综上所述，测试驱动的软件开发是一个迭代的螺旋上升的开发过程。本例中，我们还没有实现其他的6个用例测试和程序编写，请读者自己实现，也可以下载本书的配套代码参考实现。

16.1.3 iOS单元测试框架

原则上，是否使用测试框架都不会影响单元测试的结果，但是“工欲善其事，必先利其器”，使用单元测试框架更便于我们测试和分析结果。

主要的iOS单元测试框架有以下几种。

- ❑ **OCUnit**。它是开源测试框架，与Xcode工具集成在一起使用非常方便。测试报告以文本形式输出到输出窗口中。
- ❑ **GHUnit**。它是开源测试框架，可以将测试报告以应用的形式可视化输出到设备或模拟器上，也可以以文本的形式输出到输出窗口中。使用GHUnit，可以测试用OCUnit编写的测试用例。
- ❑ **OCMock**。它是开源测试框架，主要为测试提供mock对象（伪对象）。

在接下来的内容中，我们会展开介绍这3个框架。

16.2 使用 OCUnit 测试框架

我们在上一节中实现了一个基于测试驱动的软件开发案例，但是其中的单元测试程序没有采用任何框架，使用起来比较烦琐。本节中，我们将介绍iOS单元测试框架有哪些以及OCUnit测试框架的用法。

16.2.1 添加OCUnit到工程中

添加OCUnit到工程中，有两种方法：一种是在创建工程时添加，选中Include Unit Tests复选框；另一种是在现有工程中添加Cocoa Touch Unit Testing Bundle Target来实现。下面我们详细介绍这两种添加过程。

1. 创建工程时添加OCUnit

该种方式添加的单元测试属于应用测试。在创建一个工程时，如果采用Single View Application模板，则勾选Include Unit Tests复选框即可在工程中添加OCUnit框架，如图16-6所示。

添加OCUnit之后的工程如图16-7所示，此时在导航面板中会多一个PITaxTests组，其中包含PITaxTests测试类。在右边的TARGETS栏中多了一个PITaxTests Target。

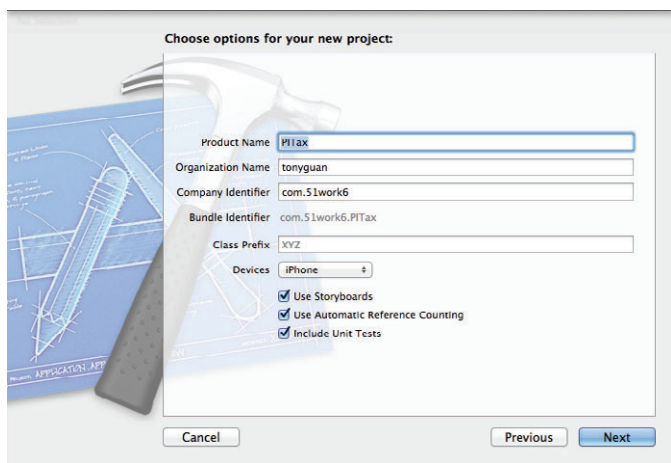


图16-6 创建工程时添加OCUnit

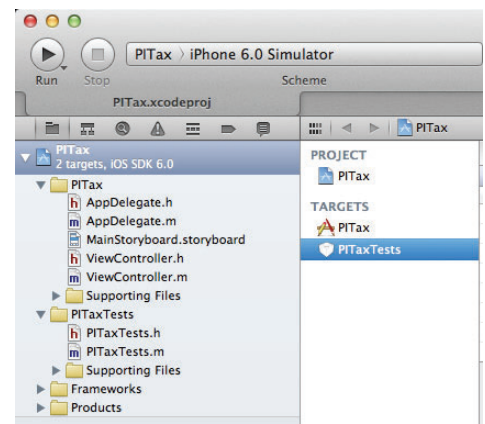


图16-7 添加OCUnit之后的工程

但是打开Scheme列表时，还是只有一个PITax，这是我们需要关注的。若要运行测试程序，可以选择Product→Test菜单，或者点击工具栏中的Test按钮（下拉Run按钮选择），或使用快捷键command+u等几种方式。如果打开Frameworks组，会发现其中添加了SenTestingKit.framework，这就是OCUnit框架。

2. 在现有工程中添加Target实现

使用这种方式添加的单元测试属于逻辑测试。在一个现有的工程中，选择File→New→Target...菜单项，此时打开的界面如图16-8所示，从中选择iOS→Other中的Cocoa Touch Unit Testing Bundle模板。

点击Next按钮，接着在Product Name文本框中输入LogicTest。添加OCUnit之后的工程如图16-9所示，此时在导航面板中多出了LogicTest组，该组中包含LogicTest测试类。在右边的Target栏中，多了一个LogicTest Target。

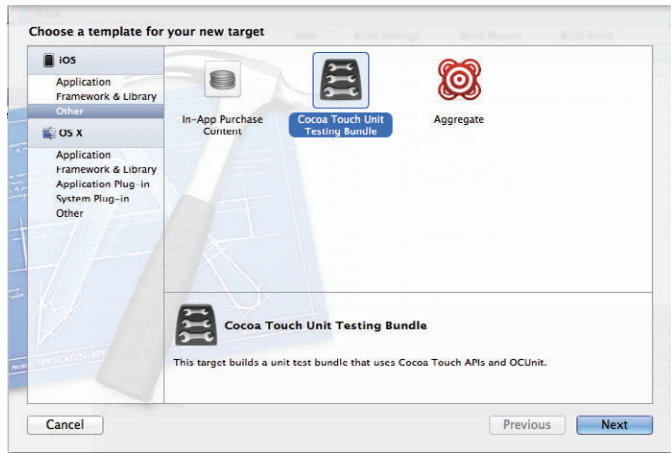


图16-8 给现有工程添加Target

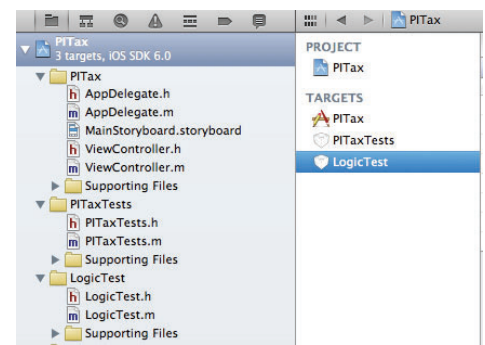


图16-9 添加OCUnit之后的工程

与上一种添加方式不同的是，此时在Scheme列表中会添加一个LogicTest，这是我们需要关注的，也是应用单元测试和逻辑单元测试的另一个不同之处。要运行它，需要选择Scheme中LogicTest的iPhone 6.0 Simulator（或iPad 6.0 Simulator），但是不能选择iOS Device，因为逻辑单元测试只能在模拟器中运行。

无论哪种方式，默认生成的测试类基本都是一样的。下面的代码是默认生成的LogicTest测试类中的LogicTest.h和LogicTest.m文件：

```
//
//LogicTest.h
//
#import <SenTestingKit/SenTestingKit.h>

@interface LogicTest : SenTestCase

@end
//
//LogicTest.m
//
#import "LogicTest.h"

@implementation LogicTest

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

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

- (void)testExample
{
    STFail(@"Unit tests are not implemented yet in LogicTest");
}

@end
```

作为OCUnit测试类，LogicTest需要引入SenTestingKit/SenTestingKit.h头文件，并继承SenTestCase父类。testExample方法是一般的测试方法，其中的STFail是OCUnit框架定义的一个宏，是无条件断言失败，实际使用时要修改这个方法中的代码。

在.m文件中，需要重新调用setUp和tearDown方法。与16.1节一样，setUp方法是初始化方法，tearDown方法是释放资源的方法，它们在每次调用测试方法之前和之后调用。因此，在测试类运行的生命周期中，这两个方法可能多次运行，它们的时序图如图16-10所示。

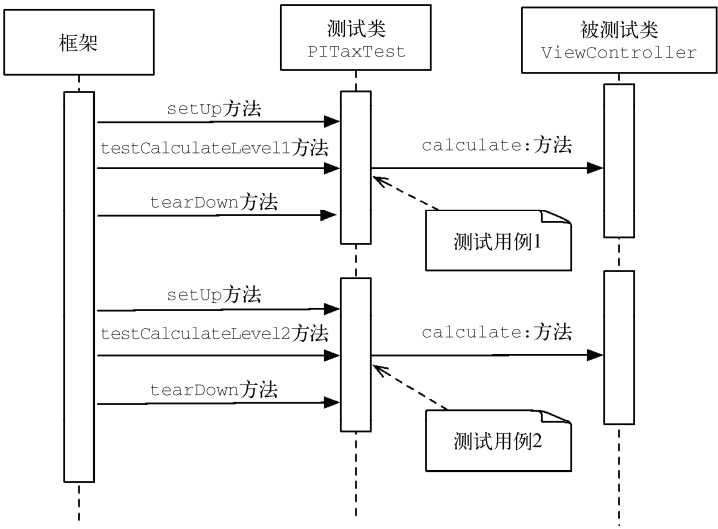


图16-10 测试类运行时序图

16.2.2 应用测试和逻辑测试

在上一节中添加OCUnit到工程时，我们提到过应用测试和逻辑测试这两个概念。它们并非OCUnit中的概念，而是单元测试中的概念。应用测试是对整个应用程序进行的测试，设计测试用例时要考虑到运行环境等因素，例如在测试Java EE时需要考虑Web容器和EJB容器等环境问题。而逻辑测试则是轻量级的，只测试某个业务逻辑对象的方法或算法的正确性。

我们通过下面的几个方面了解它们的不同。

- ❑ **创建方法。**逻辑测试通过添加Target创建，应用测试通过创建工程时选择Include Unit Tests复选框创建。
- ❑ **运行支持。**逻辑测试只能在模拟器上运行，不能在设备上运行；应用测试可以在模拟器和设备上同时运行。
- ❑ **测试加载过程。**逻辑测试加载时间短，只需要加载被测试类；而应用测试加载时间长，需要加载整个应用。
- ❑ **Scheme个数。**逻辑测试类会生成一个Scheme；应用测试类没有Scheme。

16.2.3 编写OCUnit测试方法

每一个单元测试用例对应于测试类中的一个方法，因此测试类分为逻辑测试类和应用测试类。在设计测试用例时，逻辑测试和应用测试也是不同的。编写OCUnit测试方法时，也分逻辑测试和应用测试。下面我们还是通过计算个人所得税应用介绍它们的编写过程，而被测试类ViewController的编写过程不再介绍。

1. 逻辑测试方法

逻辑测试应该测试这个应用的业务逻辑，即测试ViewController类中的calculate:方法，其测试用例的规划和设计与16.1.2小节一样，具体可以参考表16-2。

LogicTest.h的代码如下：

```
#import <SenTestingKit/SenTestingKit.h>
#import "ViewController.h"

@interface LogicTest : SenTestCase

@property (nonatomic, strong) ViewController *viewController;

@end
```

在上述代码中，我们定义了viewController属性，并且该属性的参数设置为strong。

LogicTest.m的代码如下：

```
#import "LogicTest.h"

@implementation LogicTest

- (void)setUp
{
    [super setUp];
    self.viewController = [[ViewController alloc] init];
}

- (void)tearDown
{
    self.viewController = nil;
    [super tearDown];
}

// 用例1: 测试月应纳税额不超过1500元
- (void)testCalculateLevel1
{
```

```

        double dbRevenue = 5000;
        NSString *strRevenue = [NSString stringWithFormat:@"%f",dbRevenue];
        NSString* strTax =[self.viewController calculate:strRevenue];
        STAssertTrue([strTax doubleValue] == 45, @"期望值是: 45 实际值是: %@", strTax);
    }
    //用例2: 测试月应纳税额在1500元至4500元之间
    - (void)testCalculateLevel2
    {
        double dbRevenue = 8000;
        NSString *strRevenue = [NSString stringWithFormat:@"%f",dbRevenue];
        NSString* strTax =[self.viewController calculate:strRevenue];
        STAssertTrue([strTax doubleValue] == 345, @"期望值是: 345 实际值是: %@", strTax);
    }
    //用例3: 测试月应纳税额在4500元至9000元之间
    - (void)testCalculateLevel3
    {
        double dbRevenue = 12500;
        NSString *strRevenue = [NSString stringWithFormat:@"%f",dbRevenue];
        NSString* strTax =[self.viewController calculate:strRevenue];
        STAssertTrue([strTax doubleValue] == 1245, @"期望值是: 1245 实际值是: %@", strTax);
    }
    //用例4: 测试月应纳税额在9000元至35000元之间
    - (void)testCalculateLevel4
    {
        double dbRevenue = 38500;
        NSString *strRevenue = [NSString stringWithFormat:@"%f",dbRevenue];
        NSString* strTax =[self.viewController calculate:strRevenue];
        STAssertTrue([strTax doubleValue] == 7745, @"期望值是: 7745 实际值是: %@", strTax);
    }
    //用例5: 测试月应纳税额在35000元至55000元之间
    - (void)testCalculateLevel5
    {
        double dbRevenue = 58500;
        NSString *strRevenue = [NSString stringWithFormat:@"%f",dbRevenue];
        NSString* strTax =[self.viewController calculate:strRevenue];
        STAssertTrue([strTax doubleValue] == 13745, @"期望值是: 13745 实际值是: %@", strTax);
    }
    //用例6: 测试月应纳税额在55000元至80000元之间
    - (void)testCalculateLevel6
    {
        double dbRevenue = 83500;
        NSString *strRevenue = [NSString stringWithFormat:@"%f",dbRevenue];
        NSString* strTax =[self.viewController calculate:strRevenue];
        STAssertTrue([strTax doubleValue] == 22495, @"期望值是: 22495 实际值是: %@", strTax);
    }
    //用例7: 测试月应纳税额超过80000元
    - (void)testCalculateLevel7
    {
        double dbRevenue = 103500;
        NSString *strRevenue = [NSString stringWithFormat:@"%f",dbRevenue];
        NSString* strTax =[self.viewController calculate:strRevenue];
        STAssertTrue([strTax doubleValue] == 31495, @"期望值是: 31495 实际值是: %@", strTax);
    }
    @end

```

在上述代码中，我们在setUp方法中初始化viewController，在tearDown方法中释放viewController属性。测试方法testCalculateLevel1至 testCalculateLevel7对应测试用例1至测试用例7，测试方法中的STAssertTrue是OCUnit框架中定义的断言宏。在OCUnit框架中，与断言有关的宏如表16-3所示。

表16-3 OCUit框架定义的断言宏

框 架	说 明
STAssertEqualObjects	当两个对象不相等或者是其中一个对象为nil时，断言失败
STAssertEquals	当参数1不等于参数2时断言失败，用于C中的基本数据测试
STAssertNil	当参数不是nil时，断言失败
STAssertNotNil	当参数是nil时，断言失败
STAssertTrue	当表达式为false时，断言失败
STAssertFalse	当表达式为true时，断言失败
STAssertThrows	如果表达式没有抛出异常，则断言失败
STAssertNoThrow	如果表达式抛出异常，则断言失败

2. 应用测试方法

应用测试主要用于测试应用程序的一些功能，这些功能具体到点击一个按钮触发一个事件，因此，它主要测试表示层。下面我们先看看视图控制器ViewController.m中有哪些方法需要测试，然后再来设计测试用例：

```
@implementation ViewController

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

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

- (IBAction)onClick:(id)sender {
    // 关闭键盘
    [self.txtRevenue resignFirstResponder];

    self.lblTax.text = [self calculate:self.txtRevenue.text];
}

// 计算个人所得税
- (NSString*) calculate:(NSString*)revenue {
    .....
}

@end
```

viewDidLoad和didReceiveMemoryWarning这两个方法是否需要测试，要看其中是够有一些自己编写的代码。就目前而言，我们不需要测试它们。onClick:方法是响应用户点击“计算”按钮的方法，它需要测试。calculate:方法是业务逻辑方法，我们在逻辑测试中测试过了，是否需要再测试呢？一般情况下，应该只在逻辑测试中测试就可以了。但是如果该方法需要外部环境（依赖其他类或需要特殊运行环境等），逻辑测试无法提供，此时需要进行应用测试。这是因为应用测试能够在设备上运行，它能够提供一个实际的、真实的测试环境。

下面我们实现onClick:方法的应用测试。我们要模拟点击按钮事件的处理，它的输入条件是通过文本框控件输入的，输出结果是通过标签控件展示的。设计测试用例时，选取常见值和边界值作为输入值，文本框的键盘限制为数字键盘，如图16-11所示。



图16-11 数字键盘

输入验证不需要考虑太多，只需要考虑空的情况即可。这里我们设计了6个用例，如表16-4所述。

表16-4 `onClick:` 方法的应用测试用例

测试用例	输入条件月收入总额(元)	输出结果月应纳个人所得税税额 (元)	说 明
1	空白	0.00	测试不输入直接点击“计算”按钮
2	8000	345.00	测试整数
3	8000.59	345.12	测试小数
4	08000.59	345.12	测试有前导0的数据
5	40000.50.56	8195.15	测试输入两个小数点
6	40000.50..56	8195.15	测试连在一起的两个小数点

我们看看应用测试类 `AppIlicationTest.h` 的代码：

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

#import "AppDelegate.h"
#import "ViewController.h"

@interface AppIlicationTest : SenTestCase

@property (nonatomic, strong) ViewController *viewController;

@end
```

应用测试类 `AppIlicationTest.m` 中的 `setUp` 和 `tearDown` 方法的代码如下：

```
- (void) setUp
{
    [super setUp];
    AppDelegate *appDelegate = [[UIApplication sharedApplication] delegate];
    UIWindow *window = [appDelegate window];

    UINavigationController *navController = (UINavigationController*)window.rootViewController;
    self.viewController = (ViewController*)navController.topViewController;
}

- (void) tearDown
{
}
```

```

        self.viewController = nil;
        [super tearDown];
    }

```

我们在setUp方法中需要初始化viewController属性。viewController代表的是一个视图控制器，它是iOS系统通过故事板文件创建的，而不能简单地通过下面的语句实例化：

```
self.viewController = [[ViewController alloc] init];
```

我们可以通过应用程序委托对象AppDelegate获得window对象。每个window对象可以使用属性rootViewController取得它的一个根视图控制器，本例中的根视图控制器是UINavigationController，而不是ViewController。此外，我们还需要使用UINavigationController的topViewController属性取得ViewController对象。

应用测试类ApplicationTest.m中测试方法的代码如下：

```

//测试不输入直接点击“计算”按钮的情况
- (void)testOnClickInputBlank
{
    STAssertNotNil(self.viewController, @"ViewController没有赋值。");
    //设定输入值
    self.viewController.txtRevenue.text = @"";
    //调用onClick测试
    [self.viewController onClick:nil];
    //取得输出结果
    NSString* strTax = self.viewController.lblTax.text;
    //断言
    STAssertEqualObjects(strTax, @"0.00", @"期望值是: 0.00 实际值是: %@", strTax);
}

//测试整数
- (void)testOnClickInputIntegerNumber
{
    STAssertNotNil(self.viewController, @"ViewController没有赋值。");
    //设定输入值
    self.viewController.txtRevenue.text = @"8000";
    //调用onClick测试
    [self.viewController onClick:nil];
    //取得输出结果
    NSString* strTax = self.viewController.lblTax.text;
    //断言
    STAssertEqualObjects(strTax, @"345.00", @"期望值是: 345.00 实际值是: %@", strTax);
}

//测试小数
- (void)testOnClickInputOneDot
{
    STAssertNotNil(self.viewController, @"ViewController没有赋值。");
    //设定输入值
    self.viewController.txtRevenue.text = @"8000.59";
    //调用onClick测试
    [self.viewController onClick:nil];
    //取得输出结果
    NSString* strTax = self.viewController.lblTax.text;
    //断言
    STAssertEqualObjects(strTax, @"345.12", @"期望值是: 345.12 实际值是: %@", strTax);
}

//测试输入两个小数点的情况
- (void)testOnClickInputTwoDot
{
    STAssertNotNil(self.viewController, @"ViewController没有赋值。");
    //设定输入值
    self.viewController.txtRevenue.text = @"40000.50.56";
    //调用onClick测试
    [self.viewController onClick:nil];
    //取得输出结果
    NSString* strTax = self.viewController.lblTax.text;

```

```

//断言
STAssertEqualObjects(strTax, @"8195.15", @"期望值是: 8195.15 实际值是: %@", strTax);
}
//测试有前导0的数据
- (void)testOnClickInputPrefixZero
{
    STAssertNotNil(self.viewController, @"ViewController没有赋值。");
    //设定输入值
    self.viewController.txtRevenue.text = @"08000.59";
    //调用onClick测试
    [self.viewController onClick:nil];
    //取得输出结果
    NSString* strTax = self.viewController.lblTax.text;
    //断言
    STAssertEqualObjects(strTax, @"345.12", @"期望值是: 345.12 实际值是: %@", strTax);
}
//测试连在一起的两个小数点的情况
- (void)testOnClickInputLinkDot
{
    STAssertNotNil(self.viewController, @"ViewController没有赋值。");
    //设定输入值
    self.viewController.txtRevenue.text = @"40000.50..56";
    //调用onClick测试
    [self.viewController onClick:nil];
    //取得输出结果
    NSString* strTax = self.viewController.lblTax.text;
    //断言
    STAssertEqualObjects(strTax, @"8195.15", @"期望值是: 8195.15 实际值是: %@", strTax);
}

```

这些测试方法都是非常类似的。首先,需要使用STAssertNotNil宏判断一下self.viewController是否为nil,然后使用self.viewController.txtRevenue.text设置文本框值。真正运行的时候,我们是通过文本框控件输入的。[self.viewController onClick:nil]是测试的核心目的,不需要使用参数,传递nil就可以了。输出结果是从lblTax标签控件中取得的。最后,我们使用STAssertEqualObjects断言宏。

16.2.4 分析测试报告

到目前为止,我们还没有看看测试的运行结果。这些测试结果就是测试报告,它会比我们自己编写测试类输出更多的信息,为我们提供参考和分析。应用测试报告和逻辑测试报告是没有区别的,这里我们只介绍其中一个就可以了。

测试程序运行后,无论是用例测试成功还是失败,都会输出信息,这些信息可以在3个地方看到:错误警告导航面板、日志导航面板和输出窗口。

1. 错误警告导航面板

在错误警告导航面板中看到的是用例测试失败信息,用例测试成功信息不会在这里输出。错误警告导航面板如图16-12所示,点击其中的信息,可以直接定位到断言失败语句中。

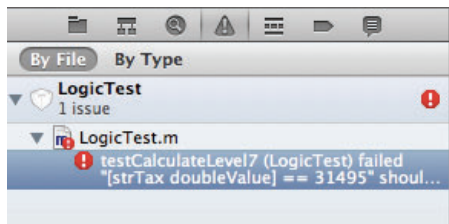


图16-12 错误警告导航面板

在图16-12中，显示的警告信息如下：

```
testCalculateLevel7 (LogicTest) failed: "[strTax doubleValue] == 31495" should be true.
期望值是: 31495 实际值是: 31500.00
```

这些信息告诉我们testCalculateLevel7测试方法断言失败，期望值和实际值是什么。

2. 日志导航面板

在日志导航面板中，我们可以看到更多的信息，包括输出成功和失败的信息，如图16-13所示。

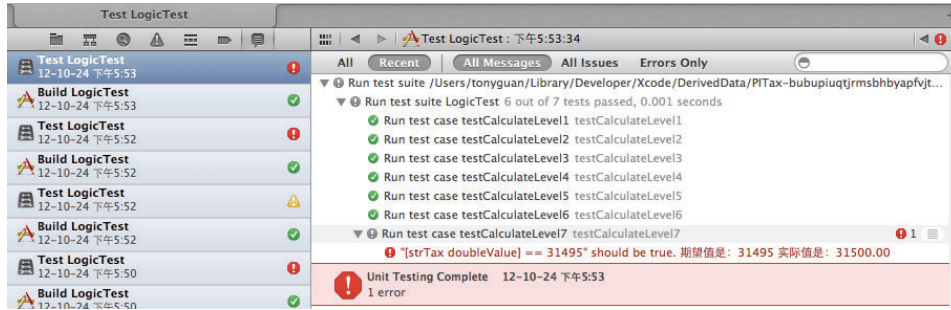


图16-13 日志导航面板

在图16-13中，显示的信息如下：

```
Test Suite 'LogicTest' started at 2012-10-24 09:53:34 +0000
Test Case '-[LogicTest testCalculateLevel1]' started.
Test Case '-[LogicTest testCalculateLevel1]' passed (0.000 seconds).
Test Case '-[LogicTest testCalculateLevel2]' started.
Test Case '-[LogicTest testCalculateLevel2]' passed (0.000 seconds).
Test Case '-[LogicTest testCalculateLevel3]' started.
Test Case '-[LogicTest testCalculateLevel3]' passed (0.000 seconds).
Test Case '-[LogicTest testCalculateLevel4]' started.
Test Case '-[LogicTest testCalculateLevel4]' passed (0.000 seconds).
Test Case '-[LogicTest testCalculateLevel5]' started.
Test Case '-[LogicTest testCalculateLevel5]' passed (0.000 seconds).
Test Case '-[LogicTest testCalculateLevel6]' started.
Test Case '-[LogicTest testCalculateLevel6]' passed (0.000 seconds).
Test Case '-[LogicTest testCalculateLevel7]' started.
/.../LogicTest.m:104: error: -[LogicTest testCalculateLevel7] : "[strTax
doubleValue] == 31495" should be true. 期望值是: 31495 实际值是: 31500.00
Test Case '-[LogicTest testCalculateLevel7]' failed (0.000 seconds).
Test Suite 'LogicTest' finished at 2012-10-24 09:53:34 +0000.
Executed 7 tests, with 1 failure (0 unexpected) in 0.001 (0.002) seconds
```

我们会看到日志信息开头有Test Suite...和Test Case...，其中Test Suite...是测试用例集合，就是一个测试类。这里有两个Test Suite...信息输出，即一个是开始，一个是结束。Test Case...是测试用例的意思。此外，日志信息中还包括了运行每一测试用例的时间以及测试测试用例集合的时间。在Test Case...中，passed说明测试通过。Executed...日志信息是对于上面测试结果的总结。

3. 输出窗口

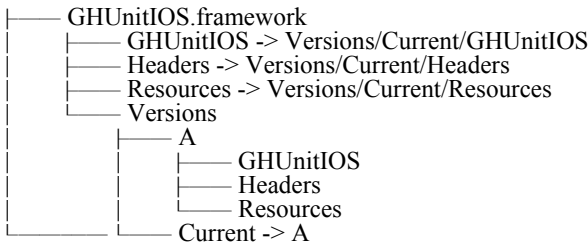
在输出窗口中看到的 information 基本上与日志导航面板看到的信息是相同的，这里就不再详述。

16.3 使用 GHUnit 测试框架

GHUnit测试框架是一个基于Objective-C的测试框架，支持Mac OS X 10.5和iOS 3.0以上版本。它也是其他一些框架的基础框架，比如OCUnit和GTM (Google Toolkit Mac)。一般用于逻辑测试，不用于应用测试。它能够在设备和模拟器上测试，还支持异步测试。使用GHUnit测试框架后，输出的日志信息更加详细。

16.3.1 添加GHUnit到工程

由于GHUnit是第三方框架, 需要从<https://github.com/gabriel/gh-unit/downloads>上下载。下载后得到的是GHUnit 框架包GHUnitIOS-<发布版本号>.zip源程序, 将其解压缩后, 得到的目录结构和内容如下:



将GHUnit框架添加到PITax工程有些复杂, 下面我们把它分成如下6个步骤进行介绍。

1. 添加一个Target

如果我们需要把GHUnit框架添加到PITax工程中, 则需要添加一个Target, 此时选择的不是Cocoa Touch Unit Testing Bundle模板, 而是Empty Application模板。添加完成后的工程如图16-14所示。

2. 添加GHUnitIOS.framework到工程

选择Frameworks组并右击, 从弹出的快捷菜单中选择“文件”菜单, 找到解压后的GHUnitIOS.framework目录, 如图16-15所示, 在Add to targets列表框中只选中GHUnitTest复选框即可。

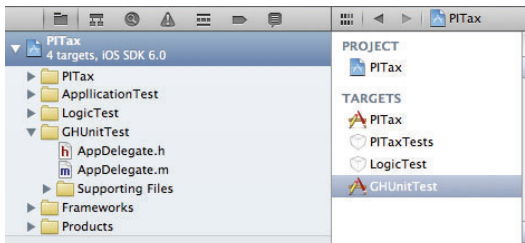


图16-14 添加一个空的应用程序Target

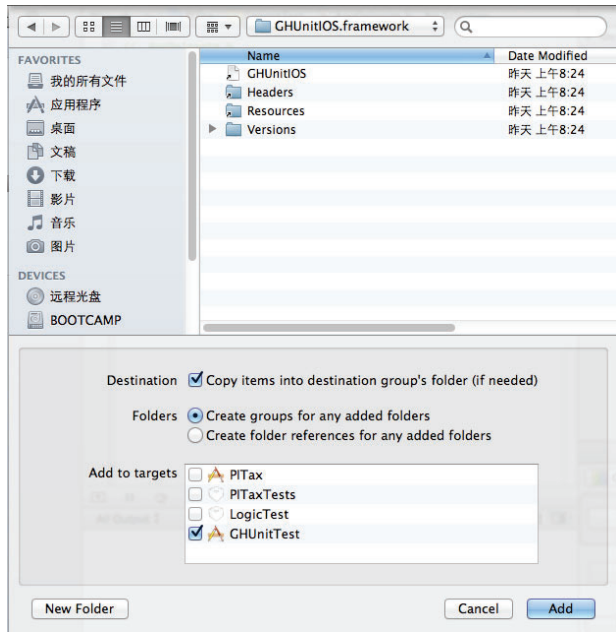


图16-15 添加GHUnitIOS.framework到工程

如果添加成功, 则可以在Frameworks组下面看到GHUnitIOS.framework。此外, 由于GHUnitIOS依赖于QuartzCore.framework框架, 我们还需要把QuartzCore.framework也添加到工程中来。

3. 设置编译参数Other Linker Flags

将GHUnitIOS.framework添加到工程后, 需要修改编译参数Other Linker Flags, 具体操作为: 选择TARGETS 中的GHUnitTest→Build Settings→Other Linker Flags, 将其修改为-ObjC -all_load, 如图16-16所示。

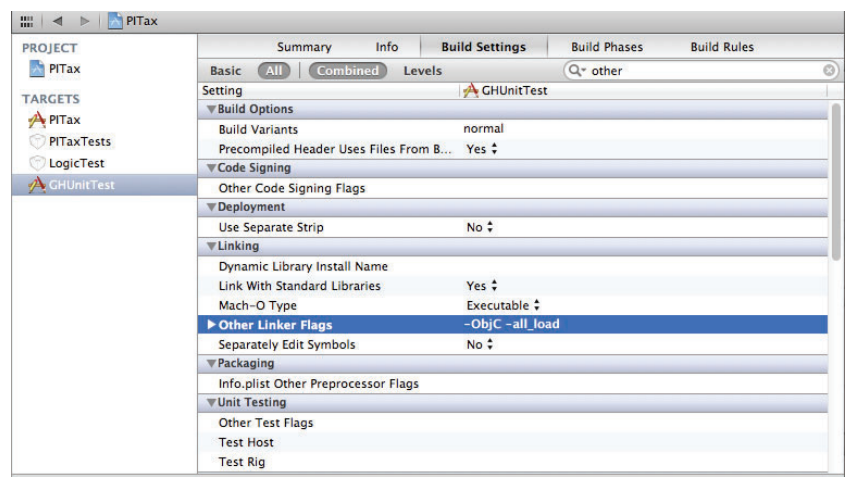


图16-16 设置编译参数Other Linker Flags

4. 修改编译参数Framework Search Paths

在编译的时候，要能找到GHUnitIOS.framework搜索路径，此时需要修改编译参数Framework Search Paths，按照图16-17所示的顺序添加路径。

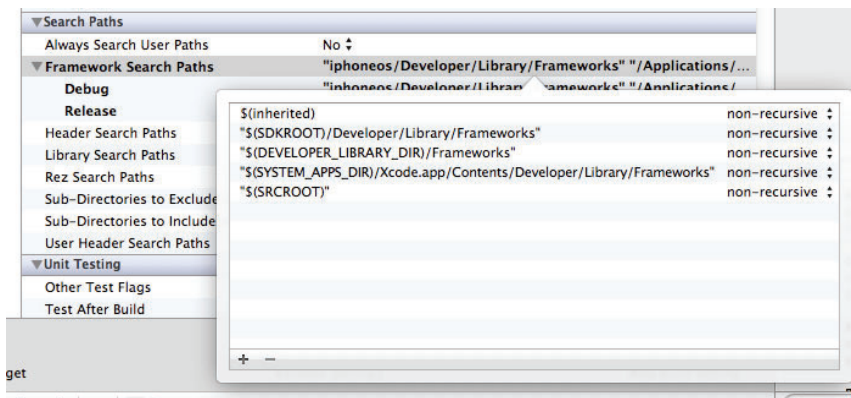


图16-17 修改编译参数Framework Search Paths

5. 修改程序代码

在代码部分中，我们需要删除AppDelegate.h和AppDelegate.m文件，然后修改GHUnitTest中的main.m文件，具体如下：

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

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

在上述代码中，我们使用语句UIApplicationMain(argc, argv, nil, @"GHUnitIOSAppDelegate")替换了语句UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class])),这是因为我们不需要程序运行AppDelegate对象，而要运行由框架提供的GHUnitIOSAppDelegate对象。

6. 设置Target Membership

由于测试类和被测试类之间有依赖关系，所有我们需要设置Target Membership。首先设置PITax中的ViewController.m文件，然后打开其文件检查器，按照图16-18所示设置它的Target Membership。ViewController是被测试类，它要能够编译到所有的Target中去。

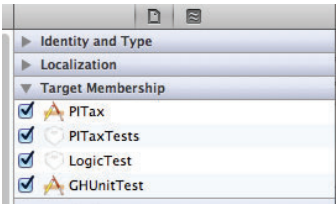


图16-18 在ViewController.m文件中设置Target Membership

如果我们需要使用应用测试，则需要按照图16-19所示设置ApplicationTest中ApplicationTest.m文件的Target Membership。这里没有选中GHUnitTest Target的原因是，GHUnitTest测试不用进行应用测试。

如果要进行逻辑测试，则需要按照图16-20所示设置LogicTest中LogicTest.m文件的Target Membership。

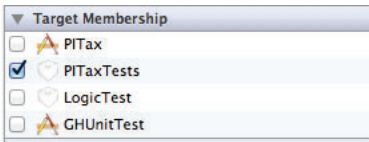


图16-19 在ApplicationTest.m文件中设置Target Membership



图16-20 在LogicTest.m文件中设置Target Membership

GHUnitIOS.framework需要按照图16-21所示设置Target Membership。

SenTestingKit.framework需要按照图16-22所示设置Target Membership。这里必须设置GHUnitTest Target，这是因为GHUnit框架中有时会用到OCUnit框架中的类。



图16-21 为GHUnitIOS.framework设置Target Membership

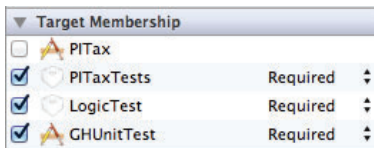


图16-22 SenTestingKit.framework设置Target Membership

16.3.2 编写GHUnit测试用例

使用GHUnit框架测试还有另一个好处，那就是可以直接测试。使用GHUnit框架编写的测试类可以是下面3个类的子类。

- ❑ **GHTTestCase**。GHUnit框架测试类。
- ❑ **SenTestCase**。OCUnit框架测试类。
- ❑ **GTMTestCase**。Google Toolkit Mac测试类。

使用SenTestCase编写测试用例在前面已经介绍过，这里不再介绍。使用GTMTestCase编写测试用例，则超出了本书的介绍范围。在这一节中，我们主要介绍使用GHTTestCase编写的测试用例。

在http://gabriel.github.com/gh-unit/docs/appledoc_include/guide_testing.html网页中，提供了编写GHTTestCase测试用例类的模板代码，具体如下：

```

@interface MyTest : GHTestCase {}
@end

@implementation MyTest

//Run before each test method
- (void)setUp { }

//Run after each test method
- (void)tearDown { }

//Run before the tests are run for this class
- (void)setUpClass { }

//Run before the tests are run for this class
- (void)tearDownClass { }

//Tests are prefixed by 'test' and contain no arguments and no return value
- (void)testA {
    GHTestLog(@"Log with a test with the GHTestLog(...) for test specific logging.");
}

//Another test; Tests are run in lexical order
- (void)testB { }

//Override any exceptions; By default exceptions are raised, causing a test failure
- (void)failWithException:(NSEException *)exception { }

@end

```

这里将.h文件中的类声明也放在.m文件中，我们的测试用例类也不使用.h文件。与SenTestCase不同的有两个特殊方法：setUpClass和tearDownClass。setUpClass是在所有的用例测试之前调用，tearDownClass是所有的用例测试完成后调用。测试过程中的时序图如图16-23所示。

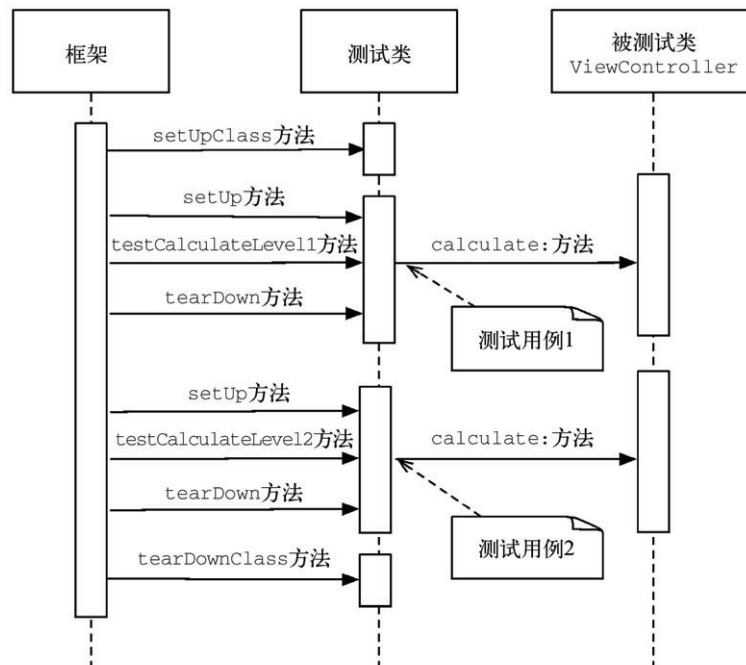


图16-23 测试类运行时序图

可以看到，setUpClass和tearDownClass在整个测试过程中只执行一次，而setUp和tearDown的执行次数与测试方法的多少有关。

创建一个测试类MyGHTests，然后删除它的MyGHTests.h文件，将其代码修改如下：

```
#import <GHUnitIOS/GHUnit.h>
@interface MyGHTests : GHTestCase { }
@end

@implementation MyGHTests

- (void)testEqualStrings {

    GHTestLog(@"比较两个字符串是否相等。");
    NSString *string1 = @"abc";
    //断言string1不是nil
    GHAssertNotNil(string1, nil);
    NSString *string2 = @"ABC";
    //断言对象相等
    GHAssertEqualObjects(string1, string2, @"string1 应该等于: %@", string2);
}

@end
```

这里我们只定义了一个测试方法testEqualStrings。在这个测试方法中，GHTestLog宏是输出日志的；GHAssertNotNil宏类似于STAssertNotNil宏，断言对象不是nil；GHAssertEqualObjects宏类似于STAssertEqualObjects宏，断言两个对象是否相同，这里的例子很显然会失败，这是我们有意而为之，为了查看断言失败日志。

16.3.3 分析测试报告

测试报告的输出形式包括文本和图形界面两种形式。在iOS平台中，图形界面输出就是一个iOS应用，如图16-24所示。



图16-24 iOS平台中GHUnit测试框架的图形界面输出

在图16-24中，左图是以表视图展示输出结果列表，GHUnit会动态检查有哪些测试类，然后进行测试，其中表视图LogicTest描述的是OCUnit测试类LogicTest的测试结果，表视图MyGHTests描述的是GHUnit测试类MyGHTests的测试结果。点击导航栏右边的Run按钮，可以重新测试。

在表视图中，黑色文字输出的测试用例代表断言成功，红色文字输出的测试用例代表断言失败。如果点击断言失败单元格，可以进入详细描述画面（见图16-24右图），其中会输出异常跟踪堆栈，这些堆栈信息对于我们分析错误和缺陷非常有用。点击导航栏右边的Re-run按钮，可以重新测试。

此外，日志信息也可以输出到输出窗口中。进入表视图界面时，它的输出信息如下：

```
Test Suite 'Tests' started.
Starting LogicTest/testCalculateLevel1
    OK (0.000s)

Starting LogicTest/testCalculateLevel2
    OK (0.000s)

Starting LogicTest/testCalculateLevel3
    OK (0.000s)

Starting LogicTest/testCalculateLevel4
    OK (0.000s)

Starting LogicTest/testCalculateLevel5
    OK (0.000s)

Starting LogicTest/testCalculateLevel6
    OK (0.000s)

Starting LogicTest/testCalculateLevel7
    FAIL (0.000s)

Starting MyGHTests/testEqualStrings
MyGHTests/testEqualStrings <GHTest: 0x9cb6fe0>: 比较两个字符串是否相等。
2012-10-25 09:06:37.915 GHUnitTest[885:3f4b]
    Name: GHTestFailureException
    File: /Users/tonyguan/Desktop/16.3 PITax/GHUnitTest/MyGHTests.m
    Line: 27
    Reason: 'abc' should be equal to 'ABC'. string1 应该等于: ABC.

(
  0  CoreFoundation          0x01ccc02e __exceptionPreprocess + 206
  1  libobjc.A.dylib         0x01109e7e objc_exception_throw + 44
  2  CoreFoundation          0x01d54fb1 -[NSException raise] + 17
  3  GHUnitTest               0x0000f495 -[GHTestCase failWithException:] + 33
  4  GHUnitTest               0x00004c63 -[MyGHTests testEqualStrings] + 2323
  5  libobjc.A.dylib         0x0111d663 -[NSObject performSelector:] + 62
  6  GHUnitTest               0x0000abc5 +[GHTesting runTestWithTarget:
    selector:exception:interval:reraiseExceptions:]
    + 450
  7  GHUnitTest               0x000067ae -[GHTest run:] + 278
  8  GHUnitTest               0x00008f67 -[GHTestGroup _run:] + 715
  9  GHUnitTest               0x00009291 -[GHTestGroup run:] + 130
 10  GHUnitTest               0x00008f67 -[GHTestGroup _run:] + 715
 11  GHUnitTest               0x00009291 -[GHTestGroup run:] + 130
 12  GHUnitTest               0x0000b778 -[GHTestRunner runTests] + 257
 13  GHUnitTest               0x0000b8e8 -[GHTestRunner _runInBackground] + 79
 14  Foundation               0x00b550d5 -[NSThread main] + 76
 15  Foundation               0x00b55034 __NSThread__main__ + 1304
 16  libsystem_c.dylib        0x99681557 _pthread_start + 344
 17  libsystem_c.dylib        0x9966bcee thread_start + 34
)
FAIL (0.000s)

Test Suite 'Tests' finished.
```

```
Executed 8 of 8 tests, with 2 failures in 0.001 seconds (0 disabled).
```

```
Failed tests:
  LogicTest/testCalculateLevel7
  MyGHTests/testEqualStrings
```

在输出结果中，“比较两个字符串是否相等”是通过GHTestLog宏输出的。点击断言失败单元格，进入详细描述界面，在输入窗口中也会输出更加详细的信息：

```
2012-10-25 09:15:32.335 GHUnitTest[885:c07] LogicTest/testCalculateLevel7 ✖ 0.00s

Name: SenTestFailureException
File: Unknown
Line: Unknown
Reason: "[strTax doubleValue] == 31495" should be true. 期望值是: 31495 实际值是: 31500.00

(
  0 CoreFoundation 0x01ccc02e __exceptionPreprocess + 206
  1 libobjc.A.dylib 0x01109e7e objc_exception_throw + 44
  2 CoreFoundation 0x01d54fb1 -[NSException raise] + 17
  3 SenTestingKit 0x20103d8c -[SenTestCase raiseException:] + 33
  4 libobjc.A.dylib 0x0111d6b0 -[NSObject performSelector:
    withObject:] + 70
  5 SenTestingKit 0x20103dc7 -[SenTestCase failWithException:] + 54
  6 GHUnitTest 0x0000423a -[LogicTest testCalculateLevel7] + 554
  7 libobjc.A.dylib 0x0111d663 -[NSObject performSelector:] + 62
  8 GHUnitTest 0x0000abc5 +[GHTesting runTestWithTarget:
    selector:exception:interval:reraiseExceptions:]
    + 450
  9 GHUnitTest 0x000067ae -[GHTest run:] + 278
  10 GHUnitTest 0x00008f67 -[GHTestGroup _run:] + 715
  11 GHUnitTest 0x00009291 -[GHTestGroup run:] + 130
  12 GHUnitTest 0x00008f67 -[GHTestGroup _run:] + 715
  13 GHUnitTest 0x00009291 -[GHTestGroup run:] + 130
  14 GHUnitTest 0x0000b778 -[GHTestRunner runTests] + 257
  15 GHUnitTest 0x0000b8e8 -[GHTestRunner _runInBackground] + 79
  16 Foundation 0x00b550d5 -[NSThread main] + 76
  17 Foundation 0x00b55034 __NSThread__main__ + 1304
  18 libsystem_c.dylib 0x99681557 _pthread_start + 344
  19 libsystem_c.dylib 0x9966bcee thread_start + 34
)
```

从上面的日志输出结果看，GHUnit要比OCUnit更加详细、方便。

16.4 使用伪对象

设计测试用例时，要考虑到运行环境等因素。测试用例依赖于外部一些对象，这些对象只能在特定的运行环境下才能提供，而我们在单元测试时还没有这些对象，如在测试Java EE时需要Web容器和EJB容器。或者是一些对象过于复杂，不方便提供，如测试iOS的表示层中的视图控制器和视图等对象时，它们都有很多状态需要初始化，直接使用也不是很方便。这些情况下可以采用伪对象来模拟真实对象。

什么时候需要使用伪对象，伪对象提出者Tim Mackinnon给了如下一些建议。

- ❑ 真实对象具有不可确定的行为（产生不可预测的结果，如股票的行情）。
- ❑ 真实对象很难被创建（比如具体的Web容器）。
- ❑ 真实对象的某些行为很难触发（比如网络错误）。
- ❑ 真实情况下程序的运行速度很慢。
- ❑ 真实对象有用户界面。
- ❑ 测试需要询问真实对象是如何被调用的（比如测试可能需要验证某个回调函数是否被调用了）。
- ❑ 真实对象实际上并不存在（当需要和其他开发小组或者新的硬件系统打交道的时候，这是一个普遍的问题）。

16.4.1 添加OCMock到工程

OCMock提供了Objective-C的伪对象框架。要添加OCMock到工程中，首先在<http://ocmock.org>网站下载新版本文件ocmock-<版本号>.dmg。打开该文件，其内容如下：

```

├── OSX
├── Source
├── iOS
│   ├── OCMock
│   └── libOCMock.a

```

在这个文件中，共有3个目录，其中OSX是给Mac OS X测试使用的，Source是源程序代码，iOS是给iOS测试使用的。下面的libOCMock.a是编译之后的静态库文件，OCMock目录是对应的头文件。如果想参考一些实例，可以从GitHub的<https://github.com/erikdoe/ocmock>下载，然后解压Examples目录下关于iOS的实例代码。

下面我们分几个步骤介绍一下如何将OCMock添加到PITax工程中。

1. 添加libOCMock.a静态库到工程

把打开的.dmg文件中，把iOS目录复制到PITax工程下面，并改名为OCMockiOS，如图16-25所示。

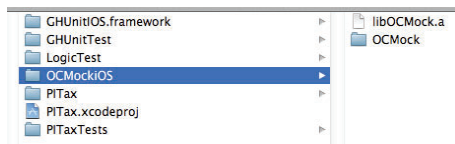


图16-25 复制iOS目录到PITax工程下

打开PITax工程，选择TARGETS中的LogicTest(我们要进行逻辑测试)，接着打开Build Phases→Link Binary With Libraries，点击下面的+按钮，此时从弹出的对话框中点击Add Other按钮，在工程目录下面找到libOCMock.a文件添加即可，如图16-26所示。

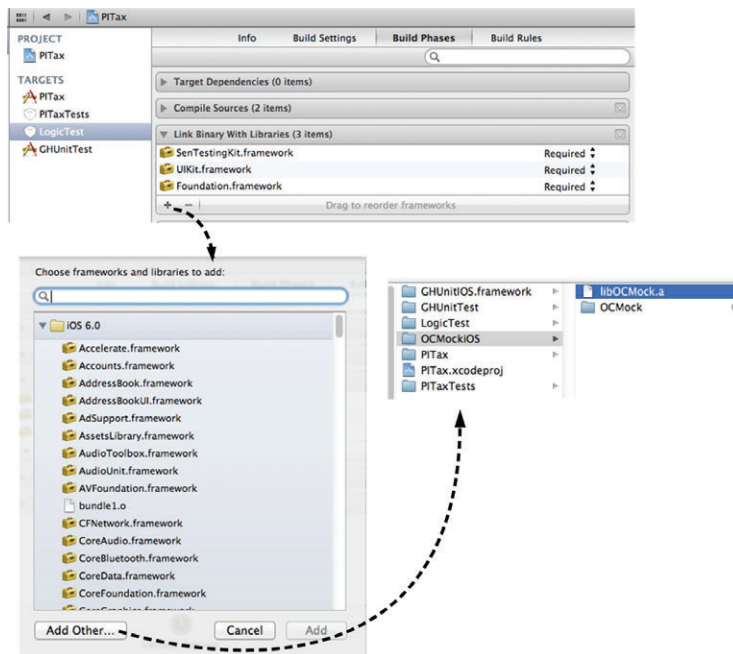


图16-26 添加libOCMock.a文件

2. 设置Target Membership

为使用OCMock对象的Target设置Target Membership。在刚才添加libOCMock.a时，我们已经选择了LogicTest Target，如果我们还想在其他的Target中使用OCMock对象，则需要选中libOCMock.a文件，打开其文件检查器，设置它的Target Membership，如图16-27所示。

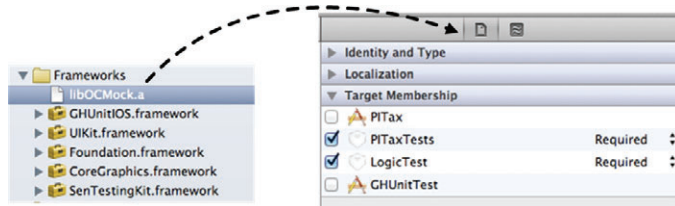


图16-27 为libOCMock.a文件设置Target Membership

3. 设置编译参数Other Linker Flags

选择TARGETS中的LogicTest→Build Settings→Other Linker Flags，将其修改为-ObjC -all_load，如图16-28所示。

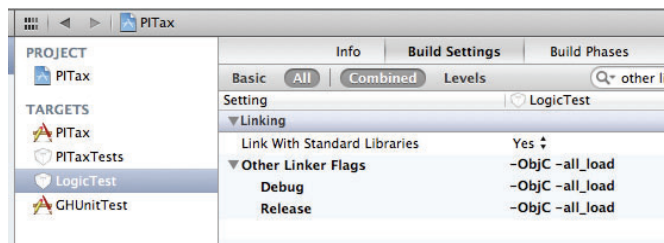


图16-28 设置编译参数Other Linker Flags

4. 设置编译参数Header Search Paths

由于在编译的时候需要提供搜索头文件路径，所以需要修改编译参数Header Search Paths。选择TARGETS中的LogicTest→Build Settings→Header Search Paths，设置其路径为"\${PROJECT_DIR}/OCMockiOS"，而且递归的，如图16-29所示。

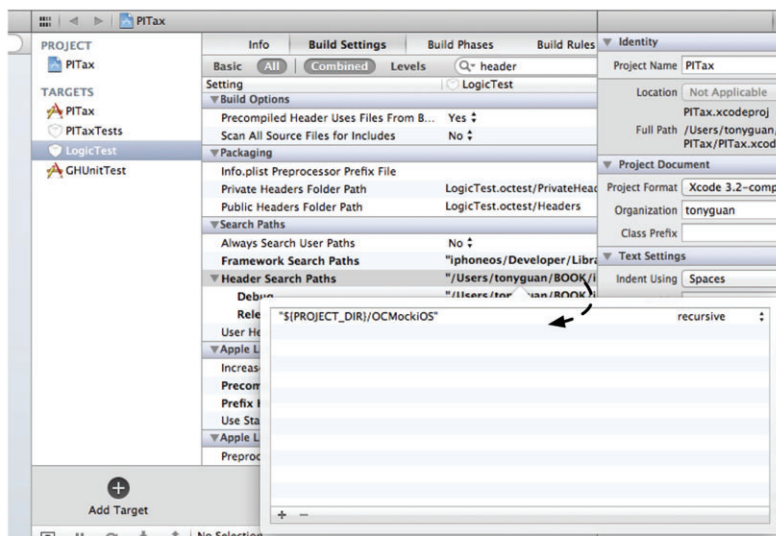


图16-29 设置编译参数Header Search Paths

16.4.2 使用OCMock对象

有了OCMock对象之后，我们原来有些不能做的测试就可以做了。例如我们原来无法对表示层中的UI事件、属性设置等进行逻辑测试，现在可以了；还有我们现在可以在没有数据库的情况下测试数据业务逻辑层。

使用伪对象的一般流程如图16-30所示。

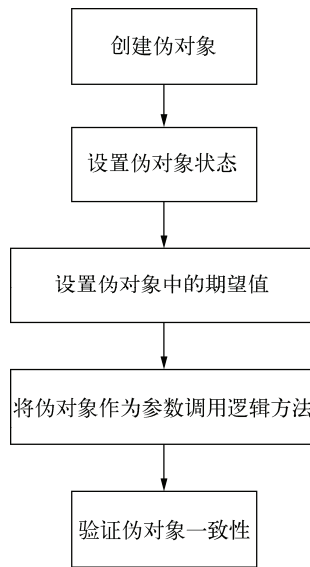


图16-30 使用伪对象的一般流程

下面我们设计一个PITax项目表示层UI测试用例来熟悉一下OCMock的用法。如果我们想测试PITax项目中的“计算”按钮的点击事件方法onClick:，该方法的代码如下：

```

- (IBAction)onClick:(id)sender {
    //关闭键盘
    [self.txtRevenue resignFirstResponder];

    self.lblTax.text =[self calculate:self.txtRevenue.text];
}
  
```

设计用例的输入值为5000，输出45.00。除了关注输入输出结果，更重要的是还要关注文本框txtRevenue是否发出resignFirstResponder消息，以证明键盘是否关闭；标签lblTax是否发出setText:消息，以证明标签lblTax会显示45.00。

我们需要在LogicTest中添加UI测试类UILogicTest。UILogicTest.h文件中的代码如下：

```

#import <SenTestingKit/SenTestingKit.h>
#import <OCMock/OCMock.h>
#import "ViewController.h"

@interface UILogicTest : SenTestCase
{
    id txtField;
    id label;
    ViewController *controller;
}

@property (nonatomic, strong) id txtField;
@property (nonatomic, strong) id label;
@property (nonatomic, strong) ViewController *controller;
  
```

```

- (void)testCalculateWithOnClick;

@end

```

在上述代码中，我们定义了3个变量，其中txtField是文本框伪对象，label是标签伪对象，controller是ViewController真实对象。controller不应该定义为伪对象，这是因为我们只关注文本框和标签。

在UILogicTest.m文件中，setUp的代码如下：

```

- (void)setUp
{
    [super setUp];

    txtField = (id)[OCMockObject mockForClass:[UITextField class]];
    label = (id)[OCMockObject mockForClass:[UILabel class]];
    controller = [[ViewController alloc] init];

    //设置视图控制器中的控件
    //设置文本字段控件属性
    [controller setValue:txtField forKey:@"txtRevenue"];
    //设置标签控件属性
    [controller setValue:label forKey:@"lblTax"];
}

```

在setUp方法中，我们初始化了伪对象txtField和label，以及真实对象controller。[OCMockObject mockForClass:[UITextField class]]语句是UITextField类型的伪对象。创建完伪对象后，需要把伪对象txtField和label设置到ViewController中。在真实世界中，ViewController是实例化时由iOS系统加载MainStoryboard.storyboard故事板文件时创建的，在加载过程中也创建了txtField和label，并且设定它们到ViewController的对应属性中。在测试情况下，我们需要自己完成。[controller setValue:txtField forKey:@"txtRevenue"]用于将txtField伪对象设置给ViewController的txtRevenue属性，[controller setValue:label forKey:@"lblTax"]用于将label伪对象设置给ViewController的lblTax属性。

在UILogicTest.m文件中，testCalculateWithOnClick测试方法的代码如下：

```

- (void)testCalculateWithOnClick
{
    [[[txtField stub] andReturn:@"5000"] text];
    [[[txtField expect] resignFirstResponder];

    [[[label expect] setText:@"45.00"];

    //测试按钮点击事件
    [controller onClick:OCMOCK_ANY];
}

```

在测试方法中，需要设置伪对象的状态，这里主要是txtField伪对象。[[[txtField stub] andReturn:@"5000"] text]用于设置txtField伪对象的text属性为5000，就相当于我们在文本框中输入了5000。[[[txtField expect] resignFirstResponder]用于设置期望文本框伪对象发出resignFirstResponder消息。[[[label expect] setText:@"45.00"]用于设置期望标签label伪对象的text属性被设置为45.00字符串。[controller onClick:OCMOCK_ANY]用于调用逻辑方法，其中OCMOCK_ANY是一个宏，代表任何对象，这是因为onClick:这个方法的参数传递什么都无所谓，给一个nil也可以。

在UILogicTest.m文件中，tearDown方法的代码如下：

```

- (void)tearDown
{
    [txtField verify];
    [label verify];
}

```

```
self.txtField = nil;
self.label = nil;
self.controller = nil;
[super tearDown];
}
```

tearDown方法要验证伪对象的一致性，即我们设置的期望是否与实现一致。[txtField verify]就用于验证伪对象txtField的一致性。验证完成之后，需要释放资源。

16.5 iOS 单元测试最佳实践

我们在前面学习了iOS单元测试框架——OCUnit、GHUnit和OCMock，我们也能够编写想要的测试程序，但是如何将它们有机地结合在一起使用呢？本节我们将讨论这个问题。

16.5.1 iOS单元测试策略

在7.5.2节讨论的分层架构设计中，不同实现模式也影响了测试框架的选择和测试用例的设计。

除信息系统层外，其他的3层我们需要进行单元测试。单元测试分为逻辑测试和应用测试，在分层架构设计中，业务逻辑层和数据持久层只能进行逻辑测试。表示层比较复杂，两种测试都可以，测试用例由设计和采用哪种技术而定。在PITax应用中，点击“计算”按钮的测试用例可以进行应用测试，可以使用OCMock进行逻辑测试。下面我们总结这3个框架与应用分层架构之间的关系，如表16-5所述。

表16-5 框架与应用分层架构

测试框架	表 示 层	业务逻辑层	数据持久层
OCUnit	逻辑测试、应用测试	逻辑测试	逻辑测试
GHUnit	逻辑测试	逻辑测试	逻辑测试
OCMock	逻辑测试提供伪对象	逻辑测试提供伪对象	逻辑测试提供伪对象

在这3个测试框架中，OCUnit和GHUnit是可以互相替代的，GHUnit要比OCUnit更为先进，但应用测试一般采用OCUnit，而不采用GHUnit。OCMock只提供伪对象，并不提供单元测试需要的用例、断言和日志输出等。严格意义上讲，OCMock算不上一个框架。

下面我们通过第10章中的案例MyNotes来介绍分层架构下的单元测试。在分层架构下，工程之间有一定的依赖关系，上层依赖于下层，所以我们按照自下而上的顺序介绍。

16.5.2 测试数据持久层

数据持久层不太容易测试，因为它依赖于数据库。方法测试的顺序也会对测试结果有影响，测试用例的设计要考虑到这些问题。对于该层的测试，我们采用GHUnit框架，一方面是由于GHUnit更加先进，另一方面是OCUnit可以获得应用沙箱目录：

```
~/Library/Application Support/iPhone Simulator/6.0/Documents/NotesList.sqlite3
```

这个目录是在模拟情况下获得的，很显然是不存在的。正常情况下，应该是：

```
~/Library/Application Support/iPhone Simulator/6.0/Applications/<应用ID>/Documents/NotesList.sqlite3
```

要想访问沙箱目录下的资源文件，不能采用OCUnit框架，这是因为OCUnit框架在逻辑测试情况下不能在设备上运行。

这里我们要测试的是MyNotesWorkspace，它的数据持久层是一个静态库工程，其环境配置方式与应用程序工

程有点区别。下面我们介绍一下添加GUnit到数据持久层（它是静态库工程PersistenceLayer），首先按照16.3.1节所述的步骤添加，创建的测试工程名为PersistenceLayerTests，其中要注意在Note.m和NoteDAO.m中这样设置Target Membership，如图16-31所示。

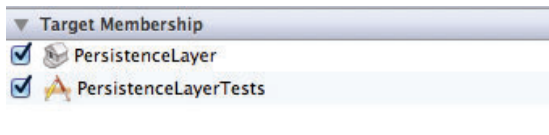


图16-31 在Note.m和NoteDAO.m中设置Target Membership

然后添加PersistenceLayerTests工程所需要的类库，本例中需要添加libsqlite3.dylib库。添加完成后的工程如图16-32所示。

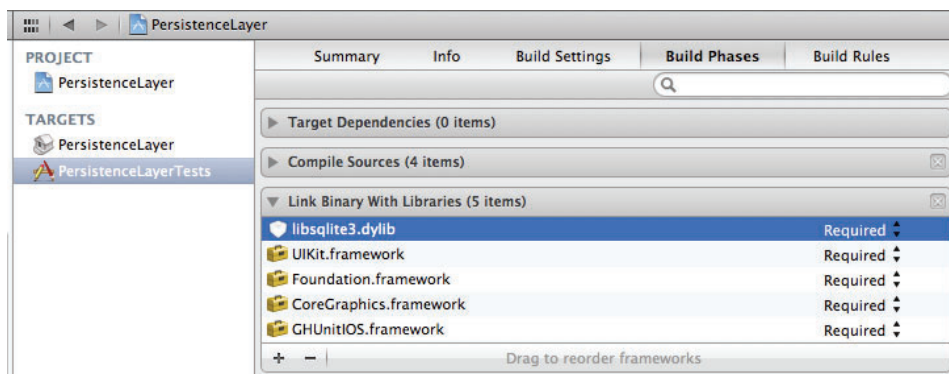


图16-32 添加libsqlite3.dylib库

接着设置Target Membership为PersistenceLayerTests，如图16-33所示。如果需要添加其他的库，其添加过程与libsqlite3.dylib一样。

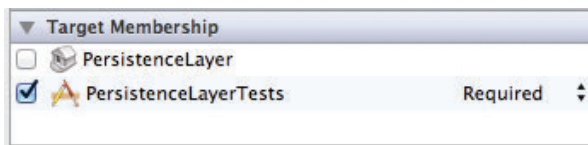


图16-33 为libsqlite3.dylib设置Target Membership

配置完成后，就可以编译测试工程了。如图16-34所示，选择Scheme为PersistenceLayerTests，然后编译之前最好先清除原来的编译文件（选择Product→Clean菜单项即可）。

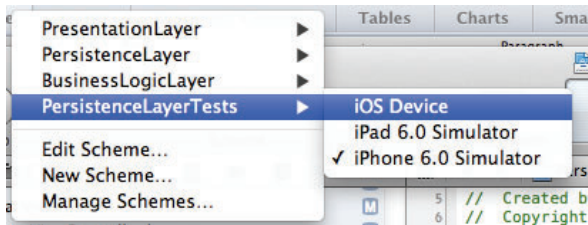


图16-34 选择Scheme为PersistenceLayerTests

添加完GJUnit框架后，我们开始设计测试类和测试用例。首先研究要测试的MyNotesWorkspace，它的数据持久层类图如图16-35所示。

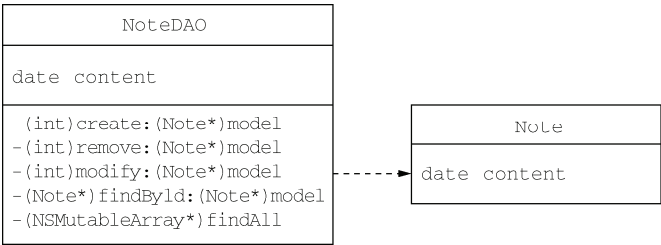


图16-35 数据持久层类图

从图中可以看到这层中有两个类，Note类是实体类，只有一些属性，这种类不需要进行测试。NoteDAO类中有5个方法，每个方法应该有多测试用例。但是由于本书篇幅有限，我们把这个测试用例的粒度放大一些，一个方法一个测试用例。因为方法之间也有依赖关系，方法测试的顺序也很重要。在Note表中没有数据的情况下，我安排的顺序是create→findAll→findById→modify→remove。数据持久层的测试用例如表16-6所示。

表16-6 数据持久层的测试用例

测试用例	测试方法	输入条件	输出结果
1	create	cdate=2000-06-0308:20:38 content= welcome www.51work6.com	无异常，返回值为0
2	findAll	无	查询记录数为1，第1条记录的cdate字段为2000-06-0308:20:38，content字段为welcome www.51work6.com
3	findById	cdate=2000-06-0308:20:38	查询结果非nil，记录的cdate字段为2000-06-0308:20:38,content字段为welcome www.51work6.com
4	modify	cdate=2000-06-0308:20:38content=www.51work6.com	无异常，返回值为0。通过findById查询该记录，查询结果非nil，记录的cdate字段为2000-06-0308:20:38，content字段为www.51work6.com
5	remove	cdate=2000-06-0308:20:38	无异常，返回值为0。通过findById查询该记录，查询结果nil，记录的cdate字段为2000-06-0308:20:38，content字段为www.51work6.com

按照表16-6所示的内容编写测试类，并且注意测试方法的顺序。测试类NoteDAOTests的头部声明代码如下：

```
#import <GJUnitIOS/GJUnit.h>
#import "NoteDAO.h"
#import "Note.h"

@interface NoteDAOTests: GHTestCase {

}

@property (nonatomic,strong) NSDateFormatter * dateFormatter;
@property (nonatomic,strong) NoteDAO * dao;

@property (nonatomic,strong) NSString* theContent;
@property (nonatomic,strong) NSDate* theDate;

@end
```

NoteDAOTests继承了GHTTestCase父类。它有4个属性，这几个属性所代表的变量在测试用例中很常用。在测试类中初始化和释放资源相关的方法的代码如下：

```
- (void)setUpClass {
    self.dateFormatter = [[NSDateFormatter alloc] init];
    [self.dateFormatter setDateFormat:@"%yyyy-MM-dd HH:mm:ss"];

    self.dao = [NoteDAO sharedManager];

    self.theContent = @"welcome www.51work6.com";
    self.theDate = [self.dateFormatter dateFromString:@"2000-06-03 08:20:38"];
}

- (void)tearDownClass {
    self.dateFormatter = nil;
    self.dao = nil;
}

- (void)setUp {}

- (void)tearDown {}
```

由于我们要初始化成员变量，并且在这个测试的生命周期中只需要一次，因此在setUpClass方法中实现初始化就可以了，对应的释放在tearDownClass方法中完成。这里我们没有使用setUp和tearDown这两个方法。

测试create方法的代码如下：

```
//测试插入备忘录的方法
- (void) testCreate
{
    //创建Note对象
    Note *note = [[Note alloc] init];
    note.date = self.theDate;
    note.content = self.theContent;

    int res = [self.dao create:note];
    //断言无异常，返回值为0
    GHAssertTrueNoThrow(res == 0, @"数据插入失败");
}
```

根据测试用例要求断言无异常，返回值为0。GHAssertTrueNoThrow(res == 0, @"数据插入失败")这条语句实现了这个断言。

测试findAll方法的代码如下：

```
//测试查询所有数据的方法
- (void) testFindAll
{
    NSArray* list = [self.dao findAll];
    //断言查询记录数为1
    GHAssertTrue([list count] == 1, @"查询记录数期望值为: 1 实际值为: %i",
        [list count]);

    Note* note = list[0];
    //断言cdate=2000-06-0308:20:38
    GHAssertEqualObjects(self.theDate, note.date, @"日期字段期望值为:
        %@ 实际值为: %@", self.theDate, note.date);
    //断言content= welcome www.51work6.com
    GHAssertEqualObjects(self.theContent, note.content, @"内容字段期望值为:
        %@ 实际值为: %@", self.theContent, note.content);
}
```

测试findById方法的代码如下：

```

//测试按照主键查询数据的方法
-(void) testFindById
{
    //创建Note对象
    Note *note = [[Note alloc] init];
    note.date = self.theDate;

    Note* resNote = [self.dao findById:note];
    //断言查询结果非nil
    GHAssertNotNil(resNote, @"查询记录为nil");
    //断言cdate=2000-06-0308:20:38
    GHAssertEqualObjects(note.date, resNote.date, @"日期字段期望值为:
        %@ 实际值为: %@", note.date, resNote.date);
    //断言content= welcome www.5lwork6.com
    GHAssertEqualObjects(self.theContent, resNote.content, @"内容字段期望值为:
        %@ 实际值为: %@", self.theContent, resNote.content);
}

```

测试modify方法的代码如下:

```

//测试修改备忘录的方法
-(void) testModify
{
    //创建Note对象
    Note *note = [[Note alloc] init];
    note.date = self.theDate;
    note.content = @"www.5lwork6.com";

    int res = [self.dao modify:note];
    //断言无异常, 返回值为0
    GHAssertTrueNoThrow(res == 0, @"数据修改失败");

    Note* resNote = [self.dao findById:note];
    //断言查询结果非nil
    GHAssertNotNil(resNote, @"查询记录为nil");
    //断言cdate=2000-06-0308:20:38
    GHAssertEqualObjects(note.date, resNote.date, @"日期字段期望值为:
        %@ 实际值为: %@", note.date, resNote.date);
    //断言content=www.5lwork6.com
    GHAssertEqualObjects(note.content, resNote.content, @"内容字段期望值为:
        %@ 实际值为: %@", note.content, resNote.content);
}

```

测试remove方法的代码如下:

```

//测试删除数据的方法
-(void) testRemove
{
    //创建Note对象
    Note *note = [[Note alloc] init];
    note.date = self.theDate;

    int res = [self.dao remove:note];
    //断言无异常, 返回值为0
    GHAssertTrueNoThrow(res == 0, @"数据修改失败");

    Note* resNote = [self.dao findById:note];
    //断言查询结果nil
    GHAssertNil(resNote, @"记录删除失败");
}

```

运行测试工程PersistenceLayerTests, 如果全部测试通过, 得到的结果如图16-36所示。



图16-36 测试数据持久层结果

16.5.3 测试业务逻辑层

业务逻辑层的测试与数据持久层的测试类似，我们也采用GJUnit框架。

MyNotesWorkspace的业务逻辑层也是一个静态库工程，它的环境配置参考数据持久层。此外，由于业务逻辑层工程BusinessLogicLayer依赖于持久层工程PersistenceLayer，所以需要设置编译参数User Header Search Paths和Library Search Paths。

User Header Search Paths用于指定用户提供类库的头文件搜索路径，具体设置方法为：选择TARGETS中的BusinessLogicLayerTests→Build Settings→User Header Search Paths，将其改为\$(BUILT_PRODUCTS_DIR)，如图16-37所示。

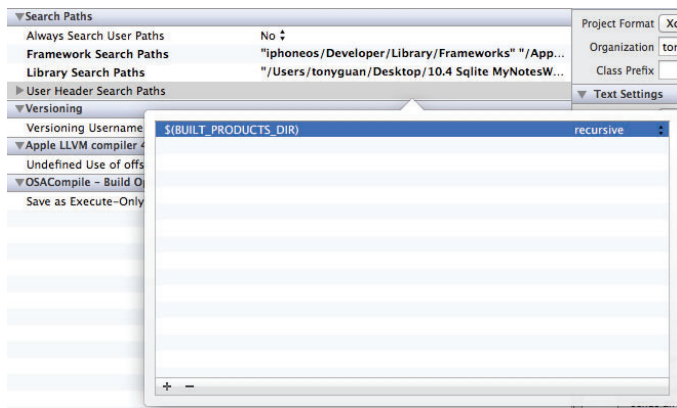


图16-37 设置编译参数User Header Search Paths

Library Search Paths用于设定类库搜索路径。这里我们需要清除原来的内容，因为它依赖的类库文件libPersistenceLayer.a在编译时会复制到BusinessLogicLayerTests编译目录下。选择TARGETS中的BusinessLogicLayerTests→Build Settings→Library Search Paths，删除其中内容。配置完成后，选择Scheme为BusinessLogicLayerTests，编译工程。

下面我们来设计测试类和测试用例。MyNotesWorkspace中的业务逻辑层类图如图16-38所示。

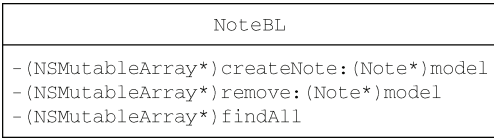


图16-38 业务逻辑层类图

从类图中可以看到，这一层中只有一个类NoteBL，它有3个方法，每个方法也应该有多个测试用例，我们把测试用例的粒度放大到一个方法一个测试用例。测试方法的顺序是createNote→findAll→remove。业务逻辑层的测试用例如表16-7所示。

表16-7 业务逻辑层的测试用例

测试用例	测试方法	输入条件	输出结果
1	createNote	cdate=2000-06-0308:20:38content=welcome www.51work6.com	查询记录数为1
2	findAll	无	查询记录数为1，第1条记录的 cdate= 2000-06-0308:20:38content= welcome www.51work6.com
3	remove	cdate=2000-06-0308:20:38	查询记录数为0

按照表16-7所示的内容编写测试类，注意测试方法的顺序。测试类NoteBLTests的头部声明代码如下：

```
#import <GUnitIOS/GUnit.h>
#import "NoteDAO.h"
#import "Note.h"
#import "NoteBL.h"

@interface NoteBLTests : GHTestCase {

}

@property (nonatomic,strong) NSDateFormatter * dateFormatter;
@property (nonatomic,strong) NoteBL * bl;

@property (nonatomic,strong) NSString* theContent;
@property (nonatomic,strong) NSDate* theDate;

@end
```

测试类NoteBLTests实现部分的声明代码如下：

```
@implementation NoteBLTests

- (void)setUpClass {
    self.dateFormatter = [[NSDateFormatter alloc] init];
    [self.dateFormatter setDateFormat:@"%yyyy-MM-dd HH:mm:ss"];

    self.bl = [[NoteBL alloc] init];

    self.theContent = @"welcome www.51work6.com";
    self.theDate = [self.dateFormatter dateFromString:@"2000-06-03 08:20:38"];
}

- (void)tearDownClass {
    self.dateFormatter = nil;
    self.bl = nil;
}

- (void)setUp { }

- (void)tearDown { }

//测试插入备忘录的方法
```

```

-(void) testCreateNote
{
    //创建Note对象
    Note *note = [[Note alloc] init];
    note.date = self.theDate;
    note.content = self.theContent;

    NSArray* list = [self.bl createNote:note];
    //断言查询记录数为1
    GHAssertTrue([list count] == 1, @"查询记录数期望值为: 1 实际值为: %i",
        [list count]);
}

//测试查询所有数据的方法
-(void) testFindAll
{
    NSArray* list = [self.bl findAll];
    //断言查询记录数为1
    GHAssertTrue([list count] == 1, @"查询记录数期望值为: 1 实际值为: %i",
        [list count]);

    Note* note = list[0];
    //断言cdate=2000-06-03 08:20:38
    GHAssertEqualObjects(self.theDate, note.date, @"日期字段期望值为:
        %@ 实际值为: %@", self.theDate, note.date);
    //断言content=welcome www.51work6.com
    GHAssertEqualObjects(self.theContent, note.content, @"内容字段期望值为:
        %@ 实际值为: %@", self.theContent, note.content);
}

//测试删除数据方法
-(void) testRemove
{
    //创建Note对象
    Note *note = [[Note alloc] init];
    note.date = self.theDate;

    NSArray* list = [self.bl remove:note];
    //断言查询记录数为0
    GHAssertTrue([list count] == 0, @"查询记录数期望值为: 0 实际值为: %i", [list count]);
}

@end

```

运行测试工程BusinessLogicLayerTests，如果全部测试通过，得到的结果如图16-39所示。



图16-39 测试业务逻辑层的结果

16.5.4 测试表示层

表示层的测试最为复杂，因为表示层有很多UI元素、动作事件和输出口问题等，并且UI控件之间很多情况下是强耦合的。在测试表示层时，其中会用到应用测试、逻辑测试和伪对象等技术。

下面我们先看看环境配置，我们需要分别配置逻辑测试的Target和应用测试的Target。

1. 逻辑测试的Target

逻辑测试的Target会用于GJUnit和OCMock，我们需要将GJUnit和OCMock添加到MyNotesWorkspace的PresentationLayer工程中。

GJUnit的添加过程请参考业务逻辑层，测试工程名为LogicTests，不同的是PresentationLayer工程依赖于两个库——libBusinessLogicLayer.a和libPersistenceLayer.a，在设置Target Membership时，libBusinessLogicLayer.a的设置如图16-40所示。

libPersistenceLayer.a的设置如图16-41所示，其中没有选中LogicTests Target，这是因为使用时libBusinessLogicLayer.a就会把libPersistenceLayer.a也一起复制到编译环境中。



图16-40 为libBusinessLogicLayer.a设置Target Membership



图16-41 为libPersistenceLayer.a设置Target Membership

OCMock框架的添加可参考16.4.1节。然后，修改编译参数User Header Search Paths，具体方法为：选择TARGETS中的 LogicTests→Build Settings→User Header Search Paths，将其改为\$(BUILT_ PRODUCTS_DIR)。

此时就将这两个框架添加到工程PresentationLayer中的LogicTests Target了。

2. 应用测试的Target

应用测试不能使用GJUnit，所以我们只需要添加OCMock框架到MyNotesWorkspace的PresentationLayer工程中即可。

我们先添加一个OCUnit的Cocoa Touch Unit Testing Bundle Target，然后在Product Name项中输入ApplicationTests。这种方式添加的测试属于逻辑测试，我们需要将其升级为应用测试。点击Scheme中的Manage Schemes...菜单项，此时弹出的对话框如图16-42所示，选中ApplicationTests复选框，点击左下角的-按钮删除它。

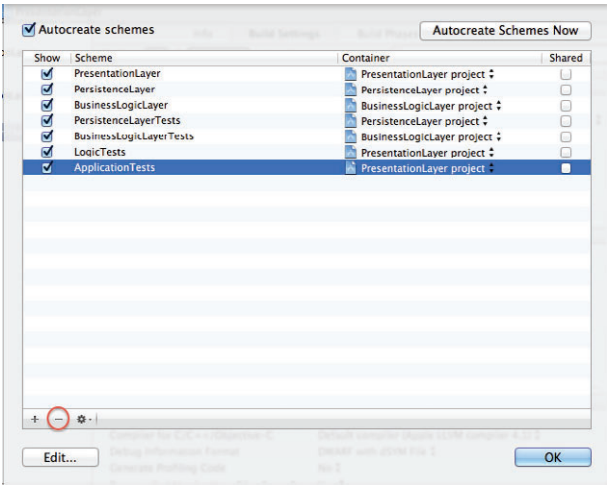


图16-42 Scheme管理对话框

接着修改编译参数 Bundle Loader。如图 16-43 所示，选择 TARGETS 中的 ApplicationTests→Build Settings→Bundle Loader，将其修改为 `$(BUILT_PRODUCTS_DIR)/PresentationLayer.app/PresentationLayer`，其中 PresentationLayer 是我们的工程名。

然后修改编译参数 Test Host。如图 16-44 所示，选择 TARGETS 中的 ApplicationTests→Build Settings→Test Host，将其修改为 `$(BUNDLE_LOADER)`。

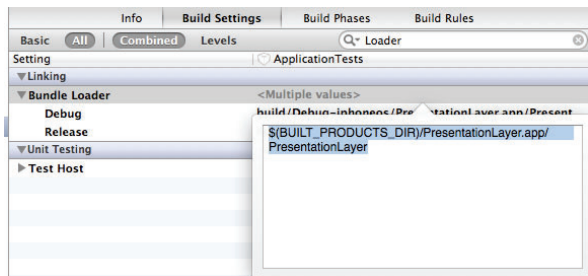


图16-43 修改编译参数Bundle Loader

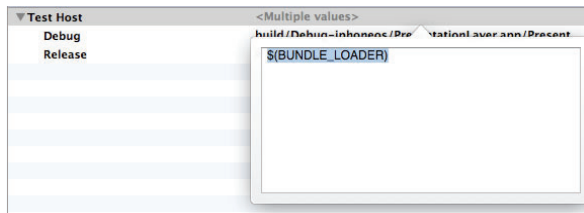


图16-44 修改编译参数Test Host

最后修改 PresentationLayer Scheme。在 Scheme 菜单中选择 Edit Scheme... 菜单项，此时弹出的对话框如图 16-45 所示，选择 Test 窗口，再点击左下角的 + 按钮，从弹出的对话框中选择 PresentationLayer→ApplicationTests 项。

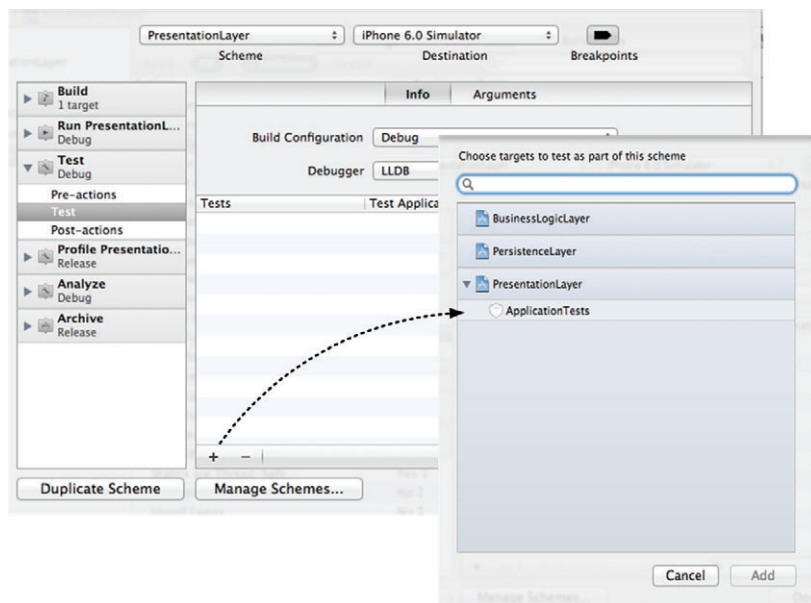


图16-45 修改PresentationLayer Scheme

这样我们就在 PresentationLayer 中添加了 ApplicationTests Target，并把 ApplicationTests 升级为应用测试 Target。选择 Product→Build For→Testing 菜单项即可编译 ApplicationTests，选择 Product→Test 菜单项即可运行 ApplicationTests。

然后，我们再把 OCMock 添加到 ApplicationTests Target 中，具体请参考 16.4.1 节。其中第一步（添加 libOCMock.a 静态库到工程）可以省略，因为在 LogicTests 已经添加了。然后，修改编译参数 User Header Search Paths，具体方法是选择 TARGETS 中的 ApplicationTests→Build Settings→User Header Search Paths，将其改为 `$(BUILT_PRODUCTS_DIR)`。

此时这两个框架就添加到工程 PresentationLayer 中的 ApplicationTests Target 中了。

下面我们来考虑一下测试用例，由于表示层测试用例很多，找一些有代表性的测试用例介绍一下。下面我们分别从应用测试和逻辑测试的测试用例进行介绍。

1. 应用测试

应用测试的测试用例依赖于应用程序本身，例如MasterViewController应用启动的初始画面的视图控制器，它是由应用加载故事板文件创建的MasterViewController实例，而AddViewController和DetailViewController视图控制器不是应用画面启动时创建的。在设计用例时，我们主要考虑测试MasterViewController类。MasterViewController类继承了表视图控制器，因此要对表视图控制器的委托方法进行测试。

下面我们看看ApplicationTests.h文件的代码，具体如下：

```
#import <SenTestingKit/SenTestingKit.h>
#import <OCMock/OCMock.h>
#import "MasterViewController.h"
#import "DetailViewController.h"
#import "Note.h"
#import "AppDelegate.h"

@interface ApplicationTests : SenTestCase

@property (nonatomic, strong) MasterViewController *masterViewController;

@end
```

ApplicationTests.m文件中setUp方法的代码如下：

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

    AppDelegate *appDelegate = [[UIApplication sharedApplication] delegate];
    UIWindow *window = [appDelegate window];

    UINavigationController *navController = (UINavigationController*)
        window.rootViewController;
    self.masterViewController = (MasterViewController*)
        navController.topViewController;

    //准备测试数据
    NSDateFormatter* dateFormatter = [[NSDateFormatter alloc] init];
    [dateFormatter setDateFormat:@"%yyyy-MM-dd HH:mm:ss"];
    //插入测试数据cdate=2000-06-03 08:20:38 , content=welcome www.51work6.com
    NSString *theContent = @"welcome www.51work6.com";
    NSDate* theDate = [dateFormatter dateFromString:@"%2000-06-03 08:20:38"];
    //创建Note对象
    Note *note = [[Note alloc] init];
    note.date = theDate;
    note.content = theContent;
    [[NoteDAO sharedManager] create:note];

    //插入测试数据cdate=2001-06-03 08:20:38 , content=www.51work6.com
    theContent = @"www.51work6.com";
    theDate = [dateFormatter dateFromString:@"%2001-06-03 08:20:38"];
    //创建Note对象
    note = [[Note alloc] init];
    note.date = theDate;
    note.content = theContent;
    [[NoteDAO sharedManager] create:note];
}
```

在上述代码中，我们首先需要从应用程序委托对象AppDelegate中取出masterViewController对象，然后插入两个测试数据。

ApplicationTests.m文件中tearDown方法的代码如下：

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

    //删除测试数据
    NSArray* list = [[NoteDAO sharedManager] findAll];
    for (Note* note in list) {
        [[NoteDAO sharedManager] remove:note];
    }
}
```

tearDown方法负责删除setUp方法中插入的两条临时测试数据。

ApplicationTests.m文件中测试方法testInitViewController的代码如下：

```
- (void)testInitViewController
{
    //断言MasterViewController非空
    XCTAssertNotNil(self.masterViewController, @"MasterViewController非空");
    //断言MasterViewController中的listData属性非空
    XCTAssertNotNil(self.masterViewController.listData,
        @"MasterViewController中listData属性非空");
    //断言MasterViewController中的bl属性非空
    XCTAssertNotNil(self.masterViewController.bl, @"MasterViewController中bl属性非空");
}
```

在上面的方法方法中，我们主要测试应用启动后，是否能正确创建MasterViewController及其属性listData和bl。listData是保存数据列表的属性，它在MasterViewController的viewDidLoad方法中初始化。bl是NoteBL*类型的对象，用于调用业务逻辑层方法，也是在viewDidLoad方法中初始化的。

ApplicationTests.m文件中测试方法testMasterViewControllerReturnsCorrectNumberOfRows方法的代码如下：

```
- (void)testMasterViewControllerReturnsCorrectNumberOfRows
{
    [self.masterViewController.tableView reloadData];
    //断言表视图返回的行数为2
    STAssertEquals(2, [self.masterViewController tableView:nil
        numberOfRowsInSection:0], @"应该返回的行数为2");
}
```

这个方法用于测试表视图控制器数据源委托方法tableView:numberOfRowsInSection:。由于数据库表Note中有两条数据，我们断言行数为2。

ApplicationTests.m文件中测试方法testMasterViewControllerTableViewCanEditRowAtIndexPath的代码如下：

```
- (void)testMasterViewControllerTableViewCanEditRowAtIndexPath
{
    //断言表视图返回的行数为2
    STAssertTrue([self.masterViewController tableView:nil
        canEditRowAtIndexPath:nil], @"应该返回True");
}
```

这个方法用于测试表视图控制器数据源委托方法tableView:canEditRowAtIndexPath:。

ApplicationTests.m文件中测试方法testMasterViewControllerReturnsTableViewCell的代码如下：

```
- (void)testMasterViewControllerReturnsTableViewCell
{
    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:0 inSection:0];

    UITableViewCell *cell = [self.masterViewController
        tableView:self.masterViewController.tableView cellForRowAtIndexPath:indexPath];
}
```

```

//断言单元格应该返回非空
STAssertNotNil(cell, @"单元格应该返回非空");
//断言标签为非空
STAssertEqualObjects(@"welcome www.51work6.com", cell.textLabel.text, @"标签为非空");
}

```

这个方法用于测试表视图控制器数据源委托方法tableView:cellForRowAtIndexPath:。

ApplicationTests.m文件中测试方法testPrepareForSegue的代码如下:

```

- (void)testPrepareForSegue
{
    id mockDetailViewController = [OCMockObject mockForClass:
    [DetailViewController class]];
    id mockStoryboardSegue = [OCMockObject mockForClass:
    [UIStoryboardSegue class]];

    [[[mockStoryboardSegue stub] andReturn:@"showDetail"] identifier];
    [[[mockStoryboardSegue stub] andReturn:mockDetailViewController]
    destinationViewController];

    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:0 inSection:0];
    Note *note = self.masterViewController.listData[indexPath.row];
    [[mockDetailViewController expect] setDetailItem:note];

    [self.masterViewController.tableView selectRowAtIndexPath:indexPath animated:NO
    scrollPosition:UITableViewScrollPositionNone];

    [self.masterViewController prepareForSegue:mockStoryboardSegue sender:
    OCMOCK_ANY];
    //验证
    [mockStoryboardSegue verify];
    [mockDetailViewController verify];
}

```

这个方法用于测试prepareForSegue:sender:方法。当用户点击单元格时,会触发prepareForSegue:sender:方法。由于触发这个方法时,界面会从主界面跳转到详细信息界面,为详细视图控制器传递数据过去,因此,测试时需要使用伪对象DetailViewController和UIStoryboardSegue。

2. 逻辑测试

在表示层的逻辑测试中,我们主要测试DetailViewController和AddViewController视图控制器。在DetailViewController控制器中测试setDetailItem方法,它是访问detailItem属性的方法。

DetailViewControllerTests是DetailViewController控制器测试类,它的代码如下:

```

#import <GHUnitIOS/GHUnit.h>
#import <OCMock/OCMock.h>
#import "DetailViewController.h"
#import "NoteDAO.h"
#import "Note.h"

@interface DetailViewControllerTests : GHTestCase {
}

@property (nonatomic,strong) NSDateFormatter * dateFormatter;

@property (nonatomic,strong) NSString* theContent;
@property (nonatomic,strong) NSDate* theDate;

@end

```

```

@implementation DetailViewControllerTests

- (void)setUpClass {
    self.dateFormatter = [[NSDateFormatter alloc] init];
    [self.dateFormatter setDateFormat:@"%yyyy-MM-dd HH:mm:ss"];

    self.theContent = @"welcome www.51work6.com";
    self.theDate = [self.dateFormatter dateFromString:@"2000-06-03 08:20:38"];
}

- (void)tearDownClass {
    self.dateFormatter = nil;
    self.theContent= nil;
    self.theDate= nil;
}

- (void)setUp { }
- (void)tearDown { }

//测试setDetailItem方法
- (void) testSetDetailItem
{

    DetailViewController *detailViewController = [[DetailViewController alloc] init];

    id mockDetailDescriptionLabel = [OCMockObject mockForClass:[UILabel class]];

    detailViewController.detailDescriptionLabel = mockDetailDescriptionLabel;

    [[mockDetailDescriptionLabel stub] andReturn:self.theContent] text];

    //创建Note对象
    Note *note = [[Note alloc] init];
    note.date = self.theDate;
    note.content = self.theContent;

    [[mockDetailDescriptionLabel expect] setText:self.theContent];

    [detailViewController setDetailItem:note];
    //验证
    [mockDetailDescriptionLabel verify];
}
@end

```

在 testSetDetailItem 方法中，我们采用 init 方法实例化 DetailViewController，因此它的属性 detailDescriptionLabel 没有设置，我们需要使用伪对象 Label 设置。在 setDetailItem: 的执行过程中，我们期望 mockDetailDescriptionLabel 被设置了 text 属性。

AddViewControllerTests 是 AddViewController 控制器的测试类，它的代码如下：

```

#import <GHUnitIOS/GHUnit.h>
#import <OCMock/OCMock.h>
#import "AddViewController.h"
#import "NoteDAO.h"
#import "Note.h"

@interface AddViewControllerTests : GHTestCase {
}

@property (nonatomic, strong) NSDateFormatter * dateFormatter;

@property (nonatomic, strong) NSString* theContent;
@property (nonatomic, strong) NSDate* theDate;

```



```

@end

@implementation AddViewControllerTests

- (void)setUpClass { }
- (void)tearDownClass {}
- (void)setUp {}
- (void)tearDown {}

//测试onclickSave方法
-(void) testOnClickSave
{
    AddViewController *addViewController = [[AddViewController alloc] init];

    id mockTextView = [OCMockObject mockForClass:[UITextView class]];
    addViewController.txtView = mockTextView;
    [[mockTextView stub] andReturn:@"welcome www.51work6.com"] text];
    [[mockTextView expect] resignFirstResponder];

    //测试通知
    id mock = [OCMockObject observerMock];
    [[NSNotificationCenter defaultCenter] addMockObserver:mock
                                           name:@"reloadViewNotification"
                                           object:nil];

    [[mock expect] notificationWithName:@"reloadViewNotification"
                                     object:OCMOCK_ANY userInfo:OCMOCK_ANY];

    [addViewController onclickSave:OCMOCK_ANY];

    //验证
    [mockTextView verify];
    [mock verify];

    [[NSNotificationCenter defaultCenter] removeObserver:mock];
}

-(void)testTextView
{
    AddViewController *addViewController = [[AddViewController alloc] init];
    id mockTextView = [OCMockObject mockForClass:[UITextView class]];

    [[mockTextView expect] resignFirstResponder];
    BOOL res = [addViewController textView:mockTextView
                      shouldChangeTextInRange:NSMakeRange(0,0) replacementText:@"\n"];

    GHAssertFalse(res, @"输入回车符号");

    [mockTextView verify];
}
@end

```

在上述代码中，testOnClickSave方法用于测试在添加界面中点击了Save按钮的情况，其中使用了UITextView的伪对象，并将这个伪对象分配给AddViewController的txtView属性。在该测试方法中，我们还测试是否发出了通知。通知的测试需要使用MockObserver，通过[OCMockObject observerMock]获得MockObserver对象，然后使用下面的语句注册通知：

```

[[NSNotificationCenter defaultCenter] addMockObserver:mock
                                           name:@"reloadViewNotification" object:nil];

```

下面的语句用于设置期望断言通知是否发出：

```

[[mock expect] notificationWithName:@"reloadViewNotification" object:OCMOCK_ANY

```

```
userInfo:OCMOCK_ANY];
```

通知完成的时候要注销通知，相关代码如下：

```
[[NSNotificationCenter defaultCenter] removeObserver:mock]
```

测试方法testTextView是测试UITextViewDelegate委托协议textView:shouldChangeTextInRange:replacementText:的方法。当在TextView中按回车键时，需要关闭键盘，因此我们需要使用UITextView伪对象，断言在输入回车时，返回值为NO。

16.6 小结

通过对本章的学习，我们了解了测试驱动的iOS开发，掌握了测试驱动开发流程，以及单元测试框架OCUnit、GHUnit和OCMock，其中OCUnit是Xcode自带的单元测试框架，GHUnit是第三方提供的测试框架，可以在模拟器和设备上测试，OCMock为我们提供了单元测试需要的伪对象。

让你的程序“飞”起来—— 性能优化

相对电脑而言，移动设备具有内存少、CPU速度慢等特点，因此iOS开发人员需要尽可能优化应用的性能。性能优化需要考虑的问题很多，本章就来介绍几个重要的优化方法。本章是非常难掌握的一章，当然也是非常重要的一章。

17.1 内存优化

Objective-C有3种内存管理方法，它们分别是MRR（Manual Retain Release，手动保持释放）、ARC（Automatic Reference Counting，自动引用计数）和GC（Garbage Collection，垃圾收集），下面我们分别介绍一下它们。

- ❑ MRR。也称为MRC（Manual Reference Counting，手动引用计数），就是由程序员自己负责管理对象生命周期，负责对象的创建和销毁。
- ❑ ARC。采用与MRR一样的内存引用计数管理方法，但不同的是，它在编译时会在合适的位置插入对象内存释放（如release、autorelease和retain等），程序员不用关心对象释放的问题。苹果推荐在新项目中使用ARC，但在iOS 5之前的系统中不能采用ARC。
- ❑ GC。在Objective-C 2.0之后，内存管理出现了类似于Java和C#的内存垃圾收集技术，但是垃圾收集与ARC完全不同，垃圾收集是后台有一个线程负责检查已经不再使用的对象，然后释放之。由于后台有一个线程一直运行，因此会严重影响性能，这也是Java和C#程序的运行速度无法超越C++的主要原因。GC技术不能应用于iOS开发，只能应用于Mac OS X开发。

从上面的介绍可知，iOS采用MRR和ARC这两种方式，ARC是苹果推荐的方式，MRR方式相对比较原始，对于程序员的能力要求很高，但是它很灵活、方便，很不容易驾驭好。17.1.1~17.1.3节主要基于MRR内存管理方式，17.1.4~17.1.5节中MRR和ARC这两种方式都是需要的。

17.1.1 内存泄漏问题的解决

内存泄漏指当一个对象或变量在使用完成后没有释放掉，这个对象一直占用着这部分内存，直到应用停止。如果这种对象过多，内存就会耗尽，其他应用就无法运行。这个问题在C++、C和Objective-C的MRR中是比较普遍的问题。

在Objective-C中，释放对象的内存时，可以发送release和autorelease消息，它们都是可以将引用计数减1。当引用计数为0时，release消息会使对象立刻释放，autorelease消息会将对象放入内存释放池中延迟释放。

我们看看“17.1.1 MemoryLeakSample”工程中ViewController的代码片段：

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

```

NSBundle *bundle = [NSBundle mainBundle];
NSString *plistPath = [bundle pathForResource:@"team"
                                             ofType:@"plist"];

// 获取属性列表文件中的所有数据
self.listTeams = [[NSArray alloc] initWithContentsOfFile:plistPath];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"CellIdentifier";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
                                         reuseIdentifier:CellIdentifier];
    }

    NSUInteger row = [indexPath row];
    NSDictionary *rowDict = [self.listTeams objectAtIndex:row];
    cell.textLabel.text = [rowDict objectForKey:@"name"];

    NSString *imagePath = [rowDict objectForKey:@"image"];
    imagePath = [imagePath stringByAppendingString:@".png"];
    cell.imageView.image = [UIImage imageNamed:imagePath];

    cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;

    return cell;
}

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSUInteger row = [indexPath row];
    NSDictionary *rowDict = [self.listTeams objectAtIndex:row];
    NSString *rowValue = [rowDict objectForKey:@"name"];

    NSString *message = [[NSString alloc] initWithFormat:@"您选择了%@队。", rowValue];
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"请选择球队"
                                                    message:message
                                                    delegate:self
                                                    cancelButtonTitle:@"Ok"
                                                    otherButtonTitles:nil];

    [alert show];

    [tableView deselectRowAtIndexPath:indexPath animated:YES];
}

```

大家看看,上面的这3个方法会有什么问题呢?如果代码基于ARC,这是没有问题的,但遗憾的这是基于MRR的,都存在内存泄漏的可能性。从理论上讲,内存泄漏是由对象或变量没有释放引起的,但实践证明并非所有的未释放对象或变量都会导致内存泄漏,这与硬件环境和操作系统环境有关,因此我们需要检测工具帮助我们找到这些“泄漏点”。

在Xcode中,共提供了两种工具帮助查找泄漏点:Analyze和Instruments。Analyze是静态分析工具。可以通过Product→Analyze菜单项启动,图17-1所示为静态分析之后的代码界面。Instruments是动态分析工具,它与Xcode集成在一起,可以在Xcode中通过Product→Profile菜单项启动。如图17-2所示,Instruments有很多跟踪模板可以动态分析和跟踪内存、CPU和文件系统。

```
16 - (void)viewDidLoad
17 {
18     [super viewDidLoad];
19
20     NSBundle *bundle = [NSBundle mainBundle];
21     NSString *plistPath = [bundle pathForResource:@"team"
22                           ofType:@"plist"];
23     //获取属性列表文件中的全部数据
24     self.listTeams = [[NSArray alloc] initWithContentsOfFile:plistPath];
25
26 }
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45 - (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
46 {
47     static NSString *CellIdentifier = @"CellIdentifier";
48     UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
49     if (cell == nil) {
50         cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier];
51     }
52
53     NSUInteger row = [indexPath row];
54     NSDictionary *rowDict = [self.listTeams objectAtIndex:row];
55     cell.textLabel.text = [rowDict objectForKey:@"name"];
56
57     NSString *imagePath = [rowDict objectForKey:@"image"];
58     imagePath = [imagePath stringByAppendingString:@".png"];
59     cell.imageView.image = [UIImage imageNamed:imagePath];
60
61     cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;
62
63     return cell;
64 }
65
66 #pragma mark - UITableViewDelegate 协议方法
67 - (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
68 {
69     NSUInteger row = [indexPath row];
70     NSDictionary *rowDict = [self.listTeams objectAtIndex:row];
71     NSString *rowValue = [rowDict objectForKey:@"name"];
72
73     NSString *message = [NSString alloc] initWithFormat:@"您选择了%@队。", rowValue];
74     UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"请选择球队"
75                                                         message:message
76                                                         delegate:self
77                                                         cancelButtonTitle:@"Ok"
78                                                         otherButtonTitles:nil];
79
80     [alert show];
81
82     [tableView deselectRowAtIndexPath:indexPath animated:YES];
83 }
84
85
86 @end
```

图17-1 使用Analyze静态分析之后的代码界面

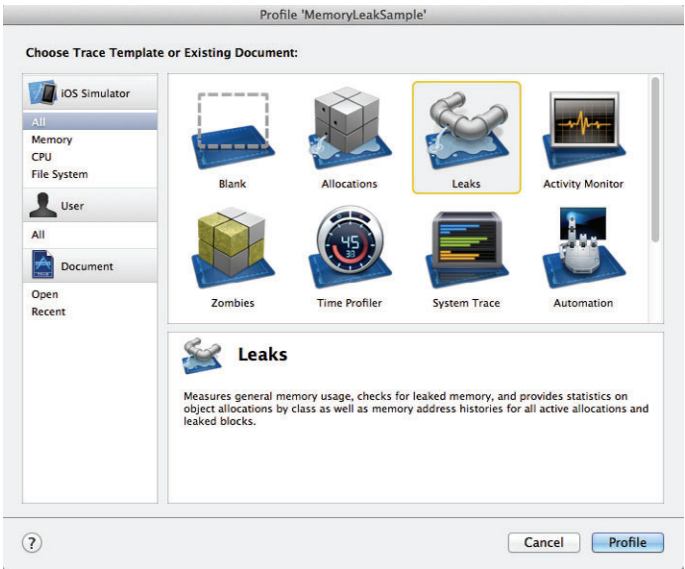


图17-2 Instruments分析工具

我们可以结合使用这两个工具查找泄漏点。先使用Analyze静态分析查找可疑泄漏点，再用Instruments动态分析中的Leaks和Allocations跟踪模板进行动态跟踪分析，确认这些点是否泄漏，或者是否有新的泄漏出现等。

在图17-1所示的Analyze静态分析结果中，凡是有🔍图标的行都是工具发现的疑似泄漏点。点击viewDidLoad方法中疑似泄漏点行末尾的🔍图标，会展开分析结果，具体如图17-3所示。



图17-3 viewDidLoad方法疑似泄漏点的展开结果

图17-3中的线表明了程序执行的路径。在这个路径中，第1处说明在第25行中，Objective-C对象的引用计数是1，说明在这里创建了一个Objective-C对象。第2处说明在第27行中引用计数为1，该对象没有释放，怀疑有泄漏。这样的说明已经很明显地告诉我们问题所在了，[[NSArray alloc] initWithContentsOfFile:plistPath]创建了一个对象，并赋值给listTeams属性所代表的成员变量，然而完成了赋值工作之后，创建的对象并没有显式地发送release和autorelease消息。将代码修改如下：

```
NSArray *array = [[NSArray alloc] initWithContentsOfFile:plistPath];
self.listTeams = array;
[array release];
```

点击tableView:cellForRowAtIndexPath:方法中疑似泄漏点行末尾的🔍图标，展开分析结果，如图17-4所示。

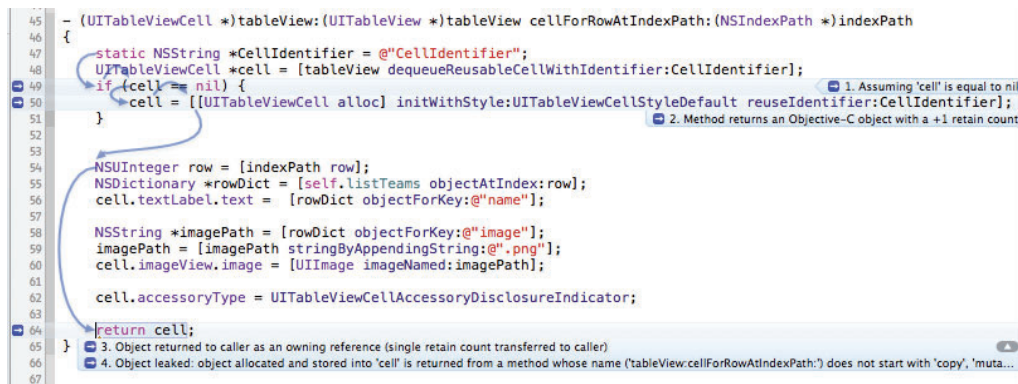


图17-4 tableView:cellForRowAtIndexPath:方法的疑似泄漏点展开结果

这主要说明UITableViewCell *类型的cell对象在第64行有可能存在泄漏。在表视图中，tableView:cellForRowAtIndexPath:方法用于实例化表视图单元格并设置数据，因此cell对象实例化后不能马上释放，而应该使用autorelease延迟释放。可以在创建cell对象时发送autorelease消息，将代码修改如下：

```
if (cell == nil) {
    cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
                                     reuseIdentifier:CellIdentifier] autorelease];
}
```

我们在看一下tableView:didSelectRowAtIndexPath:方法中的疑似泄漏点，共有两个。点击行末尾的🔍图标，展开分析结果，具体如图17-5和图17-6所示。



图17-5 tableView:didSelectRowAtIndexPath:方法疑似泄漏点1的展开结果



图17-6 tableView:didSelectRowAtIndexPath:方法疑似泄漏点2的展开结果

图17-5所示的是message对象创建之后没有释放，我们只需要在[alert show]之后添加[message release]语句代码就可以了。

在Objective-C中，实例化对象有如下两种方式：

```
NSString *message = [[NSString alloc] initWithFormat:@"%您选择了%@队。", rowValue]; ①
NSString *message = [NSString stringWithFormat:@"%您选择了%@队。", rowValue]; ②
```

第①行所示的以init开头的构造方法在alloc之后调用，我们将其称为“实例构造方法”。对于使用该方法创建的对象，其所有权是调用者，调用者需要对它的生命周期负责，具体说负责创建和释放。第②行所示的以string（去掉NS后类名）开头的方法，它通过类直接调用，我们将其称为“类级构造方法”。对于使用该方法创建的对象，其所有权非调用者所有，调用者无权释放它，否则就会因过度释放而“僵尸化”，这个问题我们会在下一节中介绍。

提示 采用alloc、new、copy和mutableCopy所创建的对象，所有权属于调用者，它的生命周期由调用者管理，调用者负责通过release或autorelease方法释放对象。

图17-6所示的是UIAlertView *类型的alert对象创建后没有释放，我们只需要在[alert show]之后添加[alert release]语句就可以了。修改之后的代码如下：

```
-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSUInteger row = [indexPath row];
    NSDictionary *rowDict = [self.listTeams objectAtIndex:row];
    NSString *rowValue = [rowDict objectForKey:@"name"];
    NSString *message = [[NSString alloc] initWithFormat:@"%您选择了%@队。", rowValue];
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"请选择球队" message:message delegate:self cancelButtonTitle:@"Ok" otherButtonTitles:nil];
    [alert show];
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

```

NSString *message = [[NSString alloc] initWithFormat:@"您选择了%@队。", rowValue];
UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"请选择球队"
                                                    message:message
                                                    delegate:self
                                                    cancelButtonTitle:@"Ok"
                                                    otherButtonTitles:nil];

[alert show];
[alert release];
[message release];
[tableView deselectRowAtIndexPath:indexPath animated:YES];
}

```

上面介绍的是使用Analyze静态分析查找可疑泄漏点。之所以称为“可疑泄漏点”，是因为这些点未必一定泄漏。确认这些点是否泄漏，还要通过Instruments动态分析工具中的Leaks和Allocations跟踪模板。Analyze静态分析只是一个理论上的预测过程。在Xcode中通过Product→Profile菜单项启动Instruments动态分析工具，接着选择Leaks模板，打开的界面如图17-7所示。

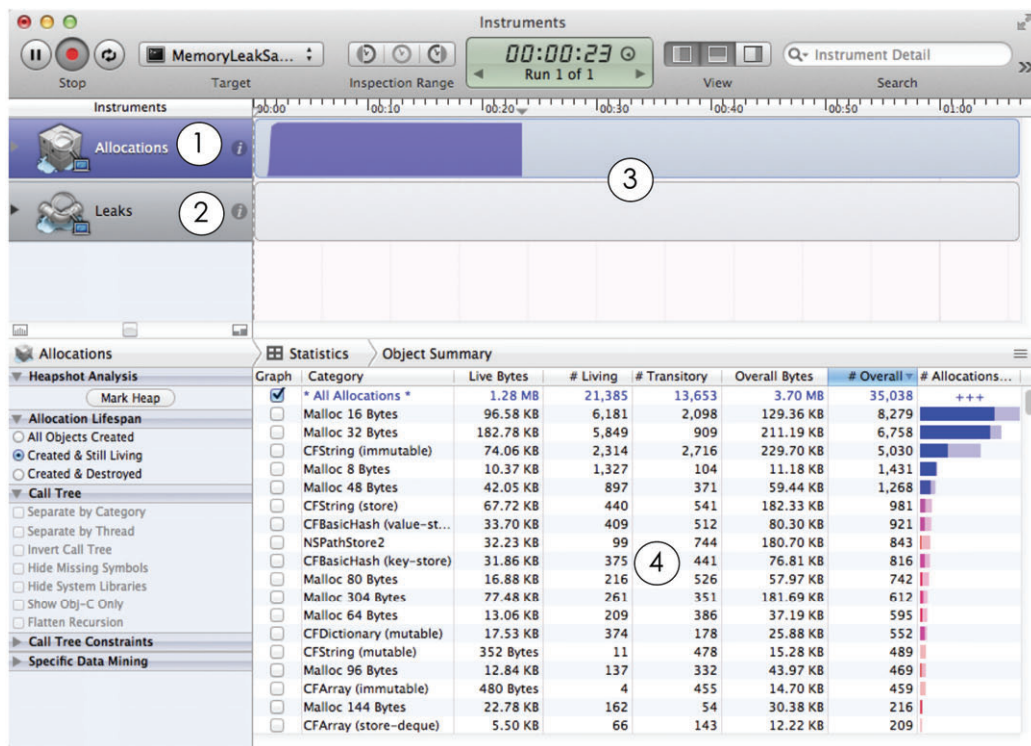


图17-7 Instruments的Leaks模板

在Instruments中，虽然选择了Leaks模板，但默认情况下也会添加Allocations模板。基本上凡是分析内存都会使用Allocations模板，它可以监控内存分布情况。选中Allocations模板（图中①区域），右边的③区域会显示随着时间的变化内存使用的折线图，同时在④区域会显示内存使用的详细信息，以及对象分配情况。点击Leaks模板（图中②区域），可以查看内存泄漏情况，如图17-8所示，如果在③区域有红线出现，则有内存泄漏，④区域则会显示泄漏的对象。

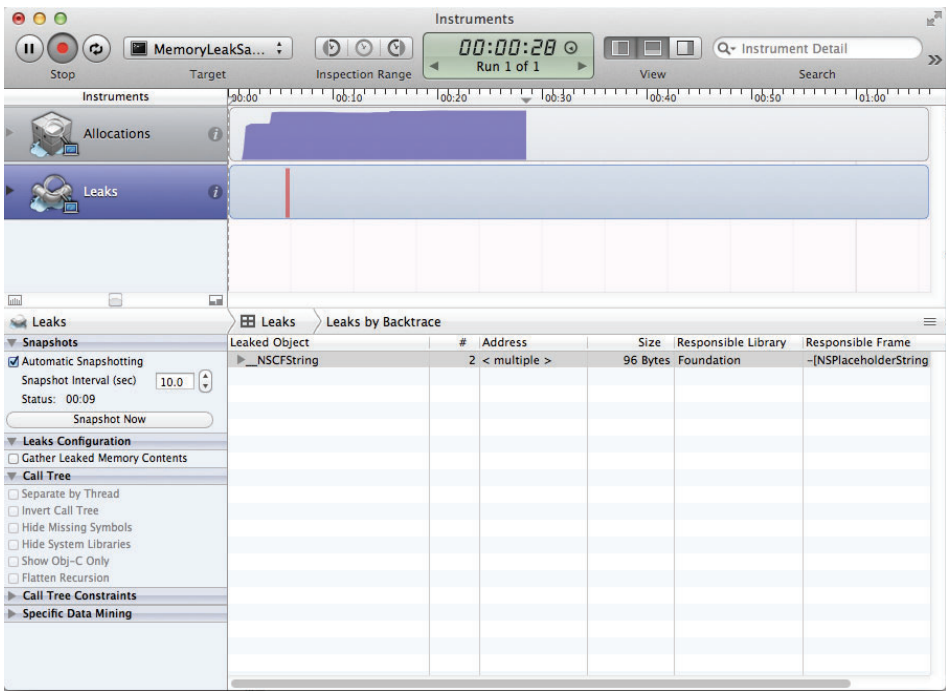


图17-8 Instruments检测到的内存泄漏

图17-8中出现的泄漏是在点击表视图中单元格测试tableView:didSelectRowAtIndexPath:方法时发生的，其中NSCFString类型的对象发生了泄漏，NSCFString类型在NSFoundation中是NSString *类型。点击泄漏对象前面的三角形，展开该对象，如图17-9所示。可以发现，里面有两个对象，可以看到它们的内存地址、占用字节、所属框架和响应方法等信息。

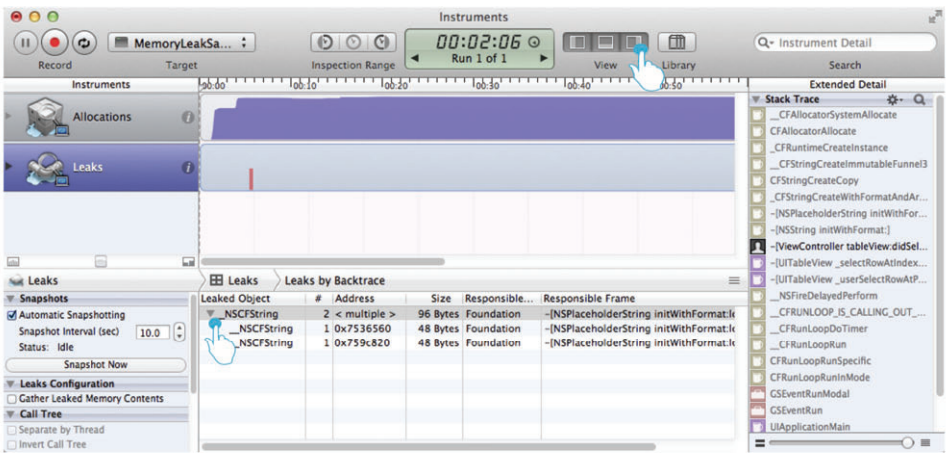




图17-9 查看泄漏的详细信息

点击View中的按钮，打开扩展详细视图，可以看到右边的跟踪堆栈信息，其中图标所示的条目是我们自己应用的代码，点击它即可进入程序代码，如图17-10所示。

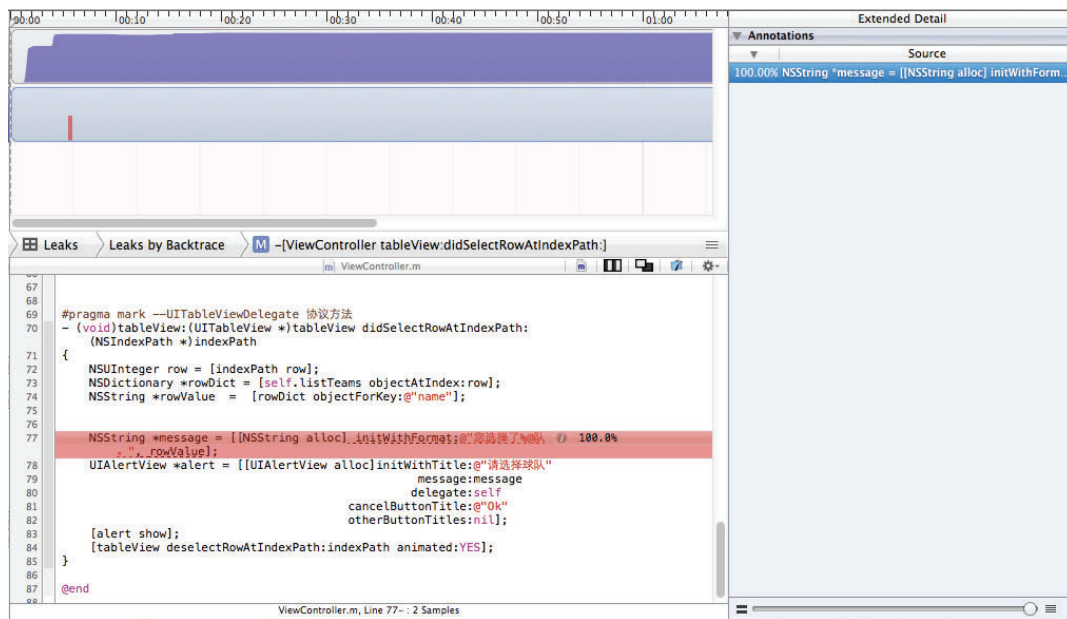


图17-10 查看泄漏点

图17-10所示的第77行代码并不是泄漏点，而是其中的NSString *类型的对象后来发生了泄漏，因此可以断定是message对象之后没有释放。我们将代码修改如下：

```
(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSUInteger row = [indexPath row];
    NSDictionary *rowDict = [self.listTeams objectAtIndex:row];
    NSString *rowValue = [rowDict objectForKey:@"name"];

    NSString *message = [[NSString alloc] initWithFormat:@"您选择了%@队。", rowValue];
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"请选择球队"
                                                    message:message
                                                    delegate:self
                                                    cancelButtonTitle:@"Ok"
                                                    otherButtonTitles:nil];

    [alert show];
    [message release];
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

这里我们添加了[message release]语句。很多人还会猜测alert对象（UIAlertView *类型的对象）会有泄漏，因此重新运行Instruments工具，反复点击单元格测试，但并未发现表示内存泄漏的红线！如图17-11所示，Instruments工具认为alert对象不释放不会引起内存泄漏。如果我们想进一步评估它对于内存的应用，可以看看Allocations模板的折线图。从图17-11中看到，每点击一次，总占用内存数都有所增加，这说明alert对象没有释放虽然不是很严重，但是也会增加占用内存，因此必须释放alert对象。

提示 有些情况下，对象没有释放是无法检测到的，反复测试内存占用也没有明显的增加，这时最好在配置比较低的设备上测试一下，如果问题依然，可以不用释放对象。但是从编程习惯上讲，我们应该释放该对象。

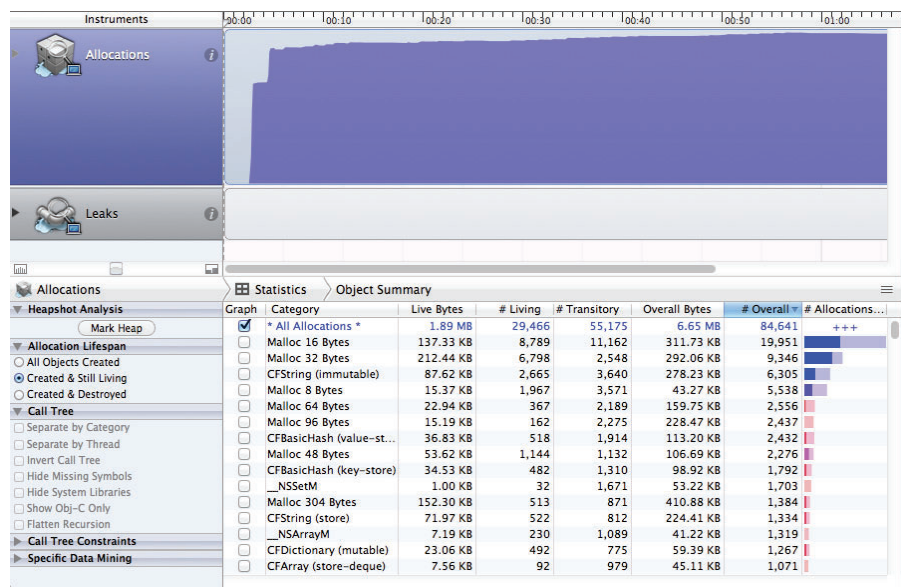


图17-11 检测alert对象泄漏

事实上，内存泄漏是极其复杂的问题，工具使用是一方面，经验是另一方面。提高经验，然后借助于工具才是解决内存泄漏的根本。

17.1.2 查找和解决僵尸对象

内存泄漏指一个对象或变量在使用完成后没有释放掉。如果我们走了另外一个极端情况，会是什么样呢？这就导致过度释放问题，从而使对象“僵尸化”，该对象则被称为僵尸对象。如果一个对象已经被释放过了，或者调用者没有这个对象的所有权而释放它，都会造成过度释放，产生僵尸对象。

对于很多人来说，僵尸对象或许听起来很恐怖、也很陌生，但是如果说起EXEC_BAD_ACCESS异常，可能大家并不陌生。如果应用的某个方法试图调用僵尸对象，则会崩溃（应用直接跳出），并抛出异常EXEC_BAD_ACCESS。

下面我们看看代码工程“17.1.2 ZombieSample”中ViewControllor的代码片段：

```
(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSUInteger row = [indexPath row];
    NSDictionary *rowDict = [self.listTeams objectAtIndex:row];
    NSString *rowValue = [rowDict objectForKey:@"name"];

    NSString *message = [[NSString alloc] initWithFormat:@"您选择了%@队。", rowValue];
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"请选择球队"
                                                       message:message
                                                       delegate:self
                                                       cancelButtonTitle:@"Ok"
                                                       otherButtonTitles:nil];

    [alert release];
    [message release];
    [alert show];
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

注意看上述代码中的粗体部分，你会发现什么问题吗？程序运行时，抛出EXEC_BAD_ACCESS异常。假设我们现在无法找到问题，可以使用Instruments工具的Zombies跟踪模板。按照图17-12所示选择Zombies模板，接着点击Profile按钮就可以进入了。



图17-12 Instruments的Zombies模板

接着点击 Allocations 中的 i 按钮，接着从弹出的对话框中设置 Target 项为 ZombieSample，然后在 Launch Configuration 中选中 Record reference counts（表示显示引用计数）和 Enable NSZombie detection（表示能够检测僵尸对象）复选框，如图17-13所示。

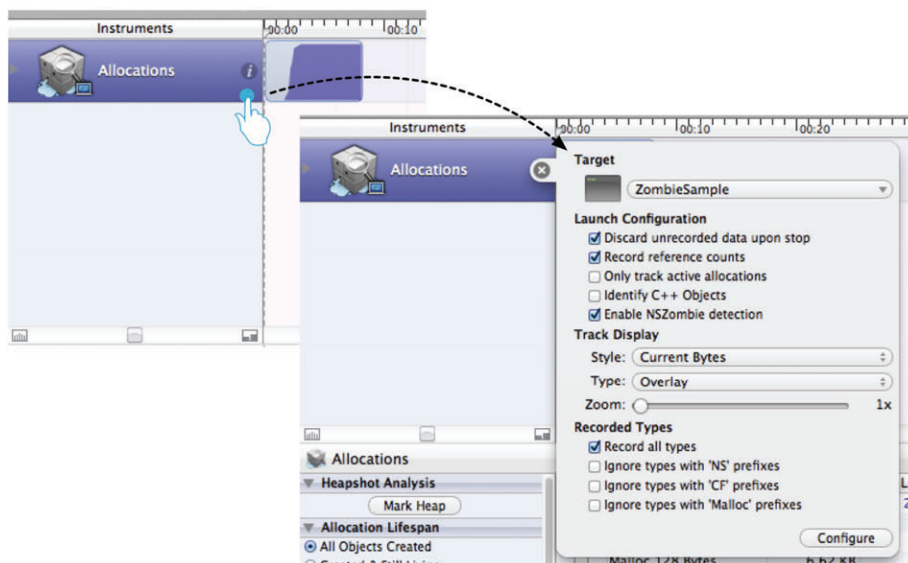


图17-13 配置Zombies模板

这样在程序运行时，如果发现僵尸对象，它就会弹出一个对话框，如图17-14所示，点击其中的→按钮，便会在屏幕下方显示僵尸对象的详细信息。

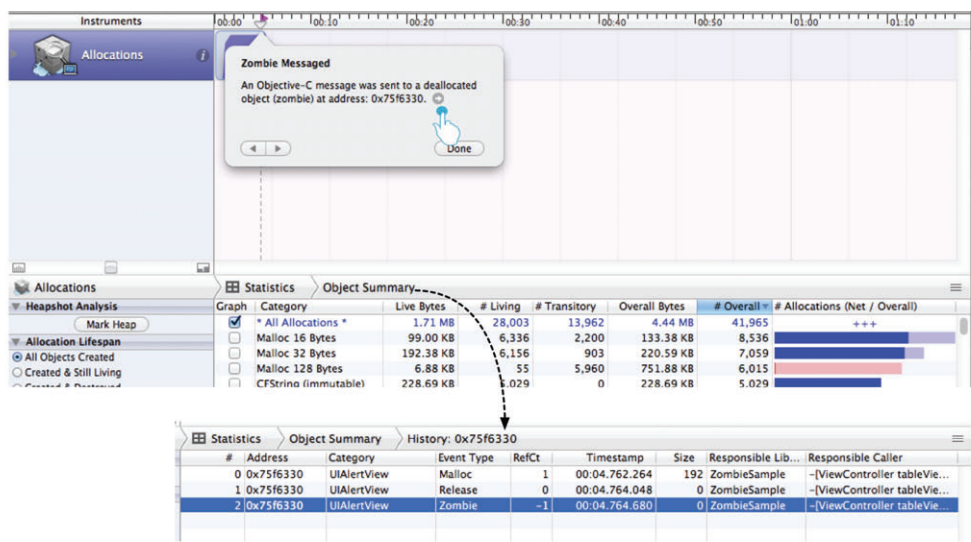


图17-14 僵尸对象详细信息

从图17-14可见，僵尸对象为UIAlertView类型。从上到下僵尸对象的引用计数变化是：1（创建）→0（释放）→-1（僵尸化）。点击View中的按钮，打开扩展详细视图，然后在右边的跟踪堆栈信息中点击条目进入我们的程序代码并定位到僵尸对象，如图17-15所示。

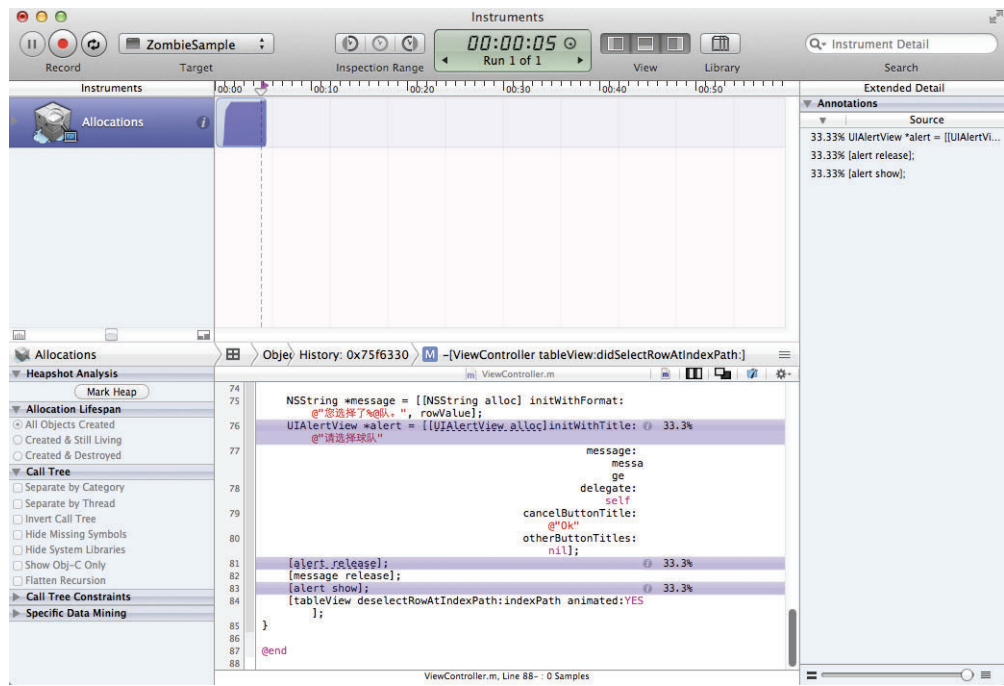


图17-15 僵尸对象定位

在图17-15中，3条高亮显示的代码会影响对象的引用计数，从中我们不难发现问题。就本例而言，我们需要将本节开头第②行代码[alert show]放在[alert release]语句之前调用就可以了。

17.1.3 autorelease的使用问题

在MRR中，释放对象通过release或autorelease消息实现，其中release消息会立刻使引用计数减一，autorelease消息会使对象放入内存释放池中延迟释放，对象的引用计数并不变化，而是向内存释放池中添加一条记录，直到池被销毁前通知池中的所有对象全部发送release消息才真正将引用计数减少。

由于使用autorelease消息会使对象延迟释放，所以除非必须，否则不要使用它释放对象。在iOS程序中，默认内存释放池的释放在程序结束。应用程序入口main.m文件的代码如下：

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

代码被包裹在@autoreleasepool {...}之间，这是池的作用范围，默认是整个应用。如果产生大量对象，采用autorelease释放也会导致内存泄漏。那么什么时候才必须使用autorelease呢？我们看看下面的代码：

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"CellIdentifier";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier] autorelease];
    }

    NSUInteger row = [indexPath row];
    NSDictionary *rowDict = [self.listTeams objectAtIndex:row];
    cell.textLabel.text = [rowDict objectForKey:@"name"];

    NSString *imagePath = [rowDict objectForKey:@"image"];
    imagePath = [imagePath stringByAppendingString:@".png"];
    cell.imageView.image = [UIImage imageNamed:imagePath];

    cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;

    return cell;
}
```

在上述代码中，cell对象不能马上释放，我们需要使用它设置表视图界面。autorelease一般用在为其他调用者提供对象的方法中，对象在该方法不能马上释放，而需要延迟释放。

此外，还有一种情况需要使用autorelease，即前面提到的“类级构造方法”：

```
NSString *message = [NSString stringWithFormat:@"您选择了%@队。", rowValue];
```

该对象的所有权虽然不是当前调用者，但它是由iOS系统通过发送autorelease消息放入到池中的。当然，这一切对于开发者都是不可见的，我们也要注意减少使用这样的语句。

17.1.4 响应内存警告

好的应用应该在系统内存警告的情况下释放一些可以重新创建的资源。在iOS中，我们可以在应用程序委托对象、视图控制器以及其他类中获得系统内存警告消息。

1. 应用程序委托对象

在应用程序委托对象中接收内存警告消息，需要重写applicationDidReceiveMemoryWarning:方法，具体可参考“17.1.4 RespondMemoryWarningSample”中AppDelegate的代码片段：

```
- (void)applicationDidReceiveMemoryWarning:(UIApplication *)application
{
    NSLog(@"AppDelegate中调用applicationDidReceiveMemoryWarning:");
}
```

2. 视图控制器

在视图控制器中接收内存警告消息，需要重写didReceiveMemoryWarning方法，具体可参考代码“17.1.4 RespondMemoryWarningSample”中ViewController的代码片段：

```
- (void)didReceiveMemoryWarning
{
    NSLog(@"ViewController中didReceiveMemoryWarning调用");
    [super didReceiveMemoryWarning];
    //释放成员变量
    [_listTeams release];
}
```

注意，释放资源代码应该放在[super didReceiveMemoryWarning]语句后面。

3. 其他类

在其他类中可以使用通知。在内存警告时，iOS系统会发出UIApplicationDidReceiveMemoryWarningNotification通知，凡是在通知中心注册了该通知的类都会接收到内存警告通知，具体可参考代码“17.1.4 RespondMemoryWarningSample”中ViewController的代码片段：

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

    NSBundle *bundle = [NSBundle mainBundle];
    NSString *plistPath = [bundle pathForResource:@"team"
                                                ofType:@"plist"];

    //获取属性列表文件中的全部数据
    NSArray *array = [[NSArray alloc] initWithContentsOfFile:plistPath];
    self.listTeams = array;
    [array release];

    //接收内存警告通知，调用handleMemoryWarning方法处理
    NSNotificationCenter *center = [NSNotificationCenter defaultCenter];
    [center addObserver:self
                selector:@selector(handleMemoryWarning)
                name:UIApplicationDidReceiveMemoryWarningNotification
                object:nil];
}

//处理内存警告
- (void) handleMemoryWarning
{
    NSLog(@"ViewController中handleMemoryWarning调用");
}
```

在上述代码中，我们在viewDidLoad方法中注册UIApplicationDidReceiveMemoryWarningNotification消息，接收到报警信息后调用handleMemoryWarning方法。这些代码完全可以写在其他类中，在ViewController中重写didReceiveMemoryWarning方法就可以了。本例只是示意性地介绍一下 UIApplicationDidReceiveMemoryWarningNotification报警消息。

内存警告在设备上并不经常出现，一般我们没有办法模拟，但模拟器上有一个功能可以模拟内存警告。启动模拟器，选择“硬件”→“模拟内存警告”模拟器菜单，这时我们会在输出窗口中看到内存警告发生了，具体如下所示：

```
2012-11-06 16:49:16.419 RespondMemoryWarningSample[38236:c07] Received memory warning.
2012-11-06 16:49:16.422 RespondMemoryWarningSample[38236:c07] AppDelegate中调用
applicationDidReceiveMemoryWarning:
2012-11-06 16:49:16.422 RespondMemoryWarningSample[38236:c07] ViewController中
handleMemoryWarning调用
2012-11-06 16:49:16.423 RespondMemoryWarningSample[38236:c07] ViewController中
didReceiveMemoryWarning调用
```

17.1.5 选择nib还是故事板

故事板是苹果在iOS 5之后推出的技术，本意是集成多个nib文件于一个故事板文件，管理起来方便。故事板还能反应控制器之间的导航关系，很多导航只需要连线就可以了，不需写代码，使用起来很方便。但是我告诫读者，从内存占用角度看，故事板不是一个好的技术。

为了比较，我们使用Xcode中的Master-Detail模板分别创建基于故事板的应用StoryboardDemo和基于nib的应用NibDemo，然后通过Instruments工具的Allocations模板分析ViewController视图控制器加载时内存占用方面的差别。图17-16是NibDemo工程的Allocations模板跟踪，图17-17是StoryboardDemo工程的Allocations模板跟踪。

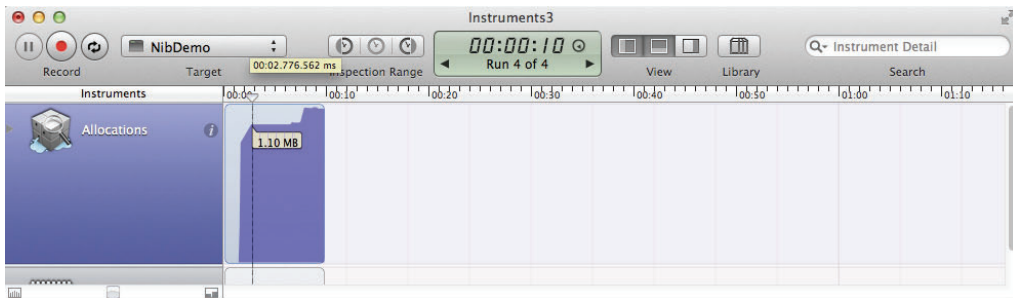


图17-16 NibDemo工程的Allocations模板跟踪

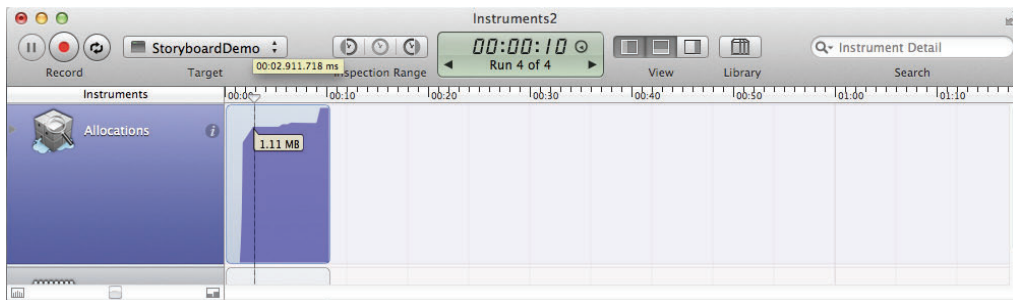


图17-17 StoryboardDemo工程的Allocations模板跟踪

如图17-16所示，界面启动用时00:02.776.562ms，内存占用1.10MB。如图17-17所示，界面启动用时00:02.911.718ms，内存占用1.11MB。NibDemo比StoryboardDemo界面启动时间要短，内存占用要少0.01MB，即约等于10KB。

默认情况下，工程中有一个故事板文件，它集成了应用中几乎所有的控制器。随着业务复杂度的增加，故事板的Interface Builder设计界面会变得杂乱无比，故事板文件会变得非常庞大。在加载故事板时，应用程序有些迟缓，内存占用也会增加。

事实上，nib仍然是比较好的技术，只不过不能表达界面之间的导航关系，界面导航要手工编写代码。

17.2 优化资源文件

从狭义上讲,资源文件是放置在应用程序本地与应用程序一起编译、打包和发布的非程序代码文件,如应用中用到的声音、视频、图片和文本文件等。从广义上讲,资源文件可以放置于任何地方,可以放置于本地,也可以放在云服务器中。

在iOS中,本地资源文件编译后放置于应用程序包文件中(即<应用名>.app文件)。如下代码用于访问如图17-18所示的team.plist本地资源文件:

```
NSBundle *bundle = [NSBundle mainBundle];
NSString *plistPath = [bundle pathForResource:@"team" ofType:@"plist"];
```

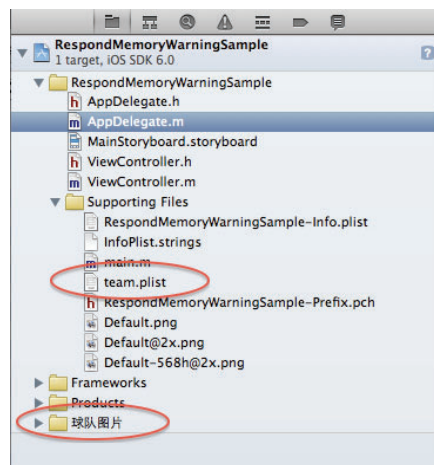


图17-18 资源文件

图17-18所示的“球队图片”组也放置了一些资源文件。添加资源文件的方法是通过右键添加文件到工程中。资源文件在使用的过程中需要优化,包括文件格式、文件类型、文件大小和文件结构等方面,使得它更适合于某个应用,“适合”两个字很重要。当然,优化方向很多,下面我们从图片格式优化、声音格式优化和.plist文件优化等方面介绍。

17.2.1 图片文件优化

图片文件优化包括文件格式和文件大小的优化。在移动设备中,支持的图片格式主要是PNG、GIF和JPEG格式,苹果推荐使用PNG格式。在Xcode中,集成了第三方PNG优化工具pngcrush^①,它可以在编译的时候对PNG格式文件进行优化和压缩,而我们只需要设定如图17-19所示的编译参数Compress PNG Files为YES就可以了。

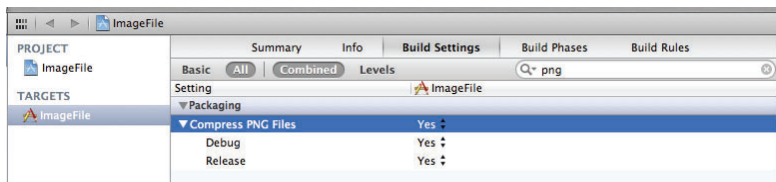


图17-19 设定编译参数Compress PNG Files

① PNG图形文件优化工具 (<http://pmt.sourceforge.net/pngcrush/>), 提供了基于微软Windows、UNIX和Linux命令行工具。

打开“17.2.1 ImageFile”工程中“测试图片”目录中的background（未优化）.png文件，在Finder中查看该文件的属性，它是一个320×480px、大小为317KB的PNG图片，如图17-20所示。

使用Xcode编译工程，在编译之后的目录中找到ImageFile.app包文件。打开包文件，查看目录中background（未优化）.png文件的属性，可以发现该文件是205KB的PNG图片了，如图17-21所示。

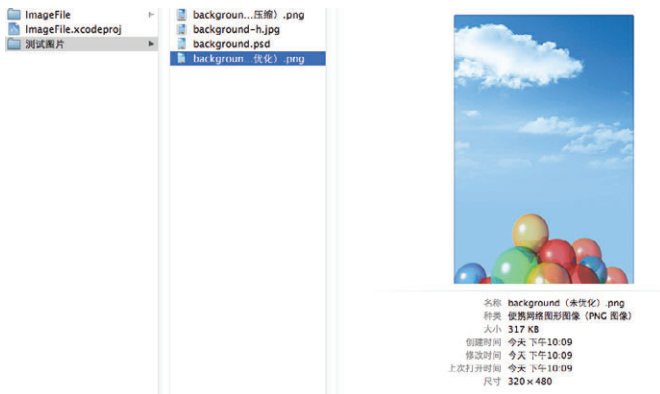


图17-20 未优化的PNG文件属性

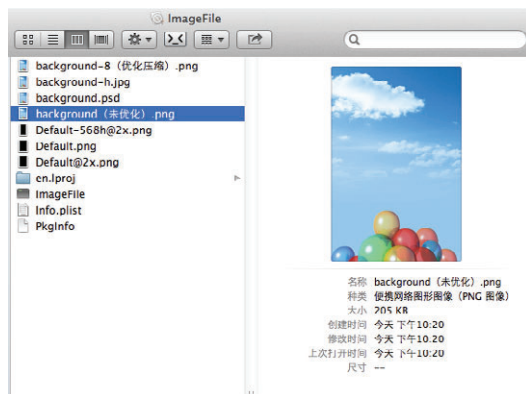


图17-21 优化的PNG文件属性

Xcode工具可以在编译时优化PNG图片，但是即便经过优化和压缩的PNG图片文件，也比JPEG图片文件大得多。打开“17.2.1 ImageFile”工程中“测试图片”目录中的background-8（优化压缩）.png文件和background-h.jpg文件，比较可以发现，前者是经过优化的质量最低的PNG-8（8位PNG格式）文件，其大小是61KB；后者是经过优化的质量最高的JPEG格式文件，其大小是22KB。在本例中，PNG比JPEG文件大3倍多。

如果是本地资源文件，这样的差别不是很大，但如果是分布在网络云服务器中的资源文件，应用在加载这些图片时，会从网络上下载到本地，这时候JPEG就很有优势了。

综上所述，如果在本地资源情况下，我们应该优先使用PNG格式文件，如果资源来源于网络，最好采用JPEG格式文件。

另外，图片是一种很特殊的资源文件。创建UIImage对象时，可以使用类级构造方法+ imageNamed:和实例构造方法-initWithContentsOfFile:。+ imageNamed:方法会在内存中建立缓存，这些缓存直到应用停止才清除。如果是贯穿整个应用的图片（如图标、logo等），推荐使用+ imageNamed:创建。如果是仅使用一次的图片，推荐使用下面的语句：

```
NSString *path = [[NSBundle mainBundle] pathForResource:@"animal-2" ofType:@"png"];
UIImage *image = [[UIImage alloc] initWithContentsOfFile:path];
.....
[image release]; //MRR情况下调用
```

17.2.2 音频文件优化

在讨论音频文件优化之前，我们先讨论一下音频文件格式。在iOS平台中，主要的音频文件格式有以下3种。

- ❑ **WAV文件。**WAV是一种由微软和IBM联合开发的用于音频数字存储的文件格式。WAV文件的格式灵活，可以存储多种类型的音频数据。由于文件较大，不太适合于移动设备这些存储容量小的设备。
- ❑ **MP3（MPEG Audio Layer 3）文件。**MP3利用MPEG Audio Layer 3技术，将数据以1：10甚至1：12的压缩率压缩成容量较小的文件。MP3是一种有损压缩格式，它尽可能地去掉人耳无法感觉的部分和不敏感的部分。这么高的压缩比率非常适合于移动设备这些存储容量小的设备，现在非常流行。
- ❑ **CAFF（Core Audio File Format）文件。**CAFF是苹果开发的专门用于Mac OS X和iOS系统的无压缩音频

格式，它被设计用来替换老的WAV格式。

❑ **AIFF（Audio Interchange File Format）文件。**AIFF是苹果开发的专门用于Mac OS X系统的专业的音频文件格式。AIFF的压缩格式是AIFF-C（或AIFC），将数据以4：1压缩率进行压缩，应用于Mac OS X和iOS系统。

音频文件优化包括了文件格式和文件大小的优化，但也要考虑到文件使用场景、采用的技术（OpenAL、AVAudioPlayer）等因素。在iOS应用中，使用本地音频资源文件的主要应用场景是背景音乐和音乐特效，下面我们在这两个方面介绍相关的优化技术。

1. 背景音乐优化

背景音乐会在应用中反复播放，它会一直驻留在内存中并耗费CPU，所以更合适比较小的文件，而压缩文件是不错的选择。压缩文件主要有AIFC和MP3两种格式，一般我们首选AIFC，因为这是苹果推荐的格式。但是我们获得的原始文件格式不一定是AIFC，这种情况下我们需要使用afconvert工具^①将其转换为AIFC格式。在终端中执行如下命令：

```
$ afconvert -f AIFC -d ima4 Fx08822_cast.wav
```

其中-f AIFC参数用于转换为AIFC格式，-d ima4参数指定解码方式，Fx08822_cast.wav是要转换的源文件。转换成功后，会在相同目录下生成Fx08822_cast.aifc文件。本例中的源文件Fx08822_cast.wav的大小是295KB，转换之后的Fx08822_cast.aifc文件的大小是82KB。当然，afconvert工具也可以转换MP3等其他压缩格式文件。如果我们同时有WAV文件，就应优先采用WAV文件。MP3本身是有损压缩，如果再经过afconvert转换，音频的质量会受到影响。

2. 音乐特效优化

音乐特效用于很多游戏中，如发射子弹、敌人被打死或按钮点击等发出的声音，这些声音都是比较短的。如果追求震撼的3D效果，可以采用苹果专用的无压缩CAFF格式文件，其他格式的文件尽量不要考虑。一般不要使用压缩音频文件，这主要是因为音乐特效通常采用OpenAL技术，它只接受无压缩的音频文件。另外，压缩音频文件都会造成音质的丢失。如果我们没有CAFF格式的文件，也可以使用afconvert工具将其转换为CAFF格式。在终端中执行如下命令：

```
$ afconvert -f caff -d LEI16 Fx08822_cast.wav
```

其中-f caff参数用于转换为CAFF格式，-d LEI16参数指定解码方式，Fx08822_cast.wav是要转换的源文件。默认音频的采样频率为22050Hz，如果想提高音频采样频率，可以通过如下命令：

```
$ afconvert -f caff -d LEI16@44100 Fx08822_cast.wav
```

其中-d LEI16@44100参数中的44100表示音频采用频率为44100Hz。

如果我们采用的资源文件不在本地，而是在分布在网络云服务器中，那么情况另当别论了，应用在加载这些音频文件时候，往往带宽是要考虑的问题，减小文件大小胜过于对音质的要求，这种情况下MP3格式是非常适合的。

综上所述，音频文件在使用本地资源情况下，应用于背景音乐时AIFC格式是首选，应用于音乐特效时CAFF格式是首选。如果是资源来源与网络，最好采用MP3格式文件。

17.3 延迟加载

延迟加载（lazy load）指一些对象不是在应用和视图等初始化时创建，而是在用到它的时候创建。当应用中有一些对象并不经常使用时，延迟加载可以提高程序性能。

^① 苹果在Mac OS X中提供的音频转换命令行工具，位于/usr/bin目录下。

17.3.1 资源文件的延迟加载

首先，我们要考虑的就是对资源文件的延迟加载。由于资源文件的访问涉及IO操作，这本身就会耗费一定的CPU时间，如果文件比较大而且加载时机又不合适，就会造成内存浪费。前面我们了解到资源文件包括图片、音频和文本文件等，无论是什么类型的文件，有些情况下采用延迟加载是很有必要的。

例如，我们有如图17-22所示的需求，可以使用分屏控件（UIPageControl）左右滑动屏幕来浏览这4张图片。



图17-22 图片延迟加载实例

下面是该实例的一种实现代码，详见PageControlNavigation工程中的ViewController.h文件：

```
@interface ViewController : UIViewController <UIScrollViewDelegate>

@property (strong, nonatomic) UIImageView *page1;
@property (strong, nonatomic) UIImageView *page2;
@property (strong, nonatomic) UIImageView *page3;
@property (strong, nonatomic) UIImageView *page4;

@property (weak, nonatomic) IBOutlet UIScrollView *scrollView;
@property (weak, nonatomic) IBOutlet UIPageControl *pageControl;

- (IBAction)changePage:(id)sender;

@end
```

PageControlNavigation工程中ViewController.m文件的代码如下：

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

    self.scrollView.contentSize = CGSizeMake(self.view.frame.size.width*4,
        self.scrollView.frame.size.height);
    self.scrollView.frame = self.view.frame;
    self.scrollView.delegate = self;

    self.page1 = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"animal-1.png"]];
    self.page1.frame = CGRectMake(0.0f, 0.0f, 320.0f, 420.0f);
    [self.scrollView addSubview:self.page1];

    self.page2 = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"animal-2.png"]];
    self.page2.frame = CGRectMake(320.0f, 0.0f, 320.0f, 420.0f);
    [self.scrollView addSubview:self.page2];
```

```

self.page3 = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"animal-3.png"]];
self.page3.frame = CGRectMake(320.0f * 2, 0.0f, 320.0f, 420.0f);
[self.scrollView addSubview:self.page3];

self.page4 = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"animal-4.png"]];
self.page4.frame = CGRectMake(320.0f * 3, 0.0f, 320.0f, 420.0f);
[self.scrollView addSubview:self.page4];

}

- (void) scrollViewDidScroll: (UIScrollView *) aScrollView
{
    CGPoint offset = aScrollView.contentOffset;
    self.pageControl.currentPage = offset.x / 320.0f;
}

#pragma mark -
#pragma mark PageControl stuff
- (IBAction)changePage:(id)sender
{
    [UIView animateWithDuration:0.3f animations:^(
        int whichPage = self.pageControl.currentPage;
        self.scrollView.contentOffset = CGPointMake(320.0f * whichPage, 0.0f);
    )];
}

```

这个程序代码我们就不用解释了。根据我们前面学习的知识，理解这些应该不是问题。大家关注的是4个UIImageView控件page1~page4什么时候实例化，什么时候加载图片。在上面的代码中，加载图片是在viewDidLoad方法中完成的，也就是在ViewController视图控制器加载时全部将4个图片文件加载了。但是有的时候用户不一定会浏览后面的图片，他可能只看到第一张或第二张，后面的两张没有去看，此时后面的两张图片仍然加载内存的话，会造成内存浪费。

下面我们重新修改了这个实例，具体可参考LazyLoadPageControlNavigation工程中的ViewController.m文件：

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.scrollView.contentSize = CGSizeMake(self.view.frame.size.width*4,
        self.scrollView.frame.size.height);
    self.scrollView.frame = self.view.frame;
    self.scrollView.delegate = self;

    self.page1 = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"animal-1.png"]]; ①
    self.page1.frame = CGRectMake(0.0f, 0.0f, 320.0f, 420.0f);
    [self.scrollView addSubview:self.page1];

}

- (void) scrollViewDidScroll: (UIScrollView *) aScrollView
{
    CGPoint offset = aScrollView.contentOffset;
    self.pageControl.currentPage = offset.x / 320.0f;

    [self loadImage:self.pageControl.currentPage + 1]; ②

}

#pragma mark -
#pragma mark PageControl stuff
- (IBAction)changePage:(id)sender
{
    [UIView animateWithDuration:0.3f animations:^(

```

```

        int whichPage = self.pageControl.currentPage;
        self.scrollView.contentOffset = CGPointMake(320.0f * whichPage, 0.0f);
        [self loadImage:self.pageControl.currentPage + 1];
    }];
}

-(void) loadImage:(int) nextPage
{
    if (nextPage == 1 && self.page2 == nil) {
        self.page2 = [[UIImageView alloc] initWithImage:[UIImage
            imageNamed:@"animal-2.png"]];
        self.page2.frame = CGRectMake(320.0f * nextPage, 0.0f, 320.0f, 420.0f);
        [self.scrollView addSubview:self.page2];
    }

    if (nextPage == 2 && self.page3 == nil) {
        self.page3 = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"animal-3.png"]];
        self.page3.frame = CGRectMake(320.0f * nextPage, 0.0f, 320.0f, 420.0f);
        [self.scrollView addSubview:self.page3];
    }

    if (nextPage == 3 && self.page4 == nil) {
        self.page4 = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"animal-4.png"]];
        self.page4.frame = CGRectMake(320.0f * nextPage, 0.0f, 320.0f, 420.0f);
        [self.scrollView addSubview:self.page4];
    }
}

```

由于ViewController视图控制器加载时，界面只需要显示第一个图片animal-1.png，因此在viewDidLoad方法中，第①行代码只加载 animal-1.png 图片并创建 page1 对象。当屏幕滑动时，触发 ScrollView 的 scrollViewDidScroll: 方法。在第②行代码中，我们调用加载下一张图片的方法 loadImage:。在分屏控件变化时，会触发 changePage: 方法。在第③行代码中，我们调用加载下一张图片的方法 loadImage:。loadImage: 是我们自己定义的加载下一张图片的方法，这里只需要考虑后面3张图片就可以了。在该方法中，如果装载图片的 UIImageView 控件 page1~page4 为 nil，则说明这个图片没有加载，需要加载和创建 UIImageView 对象。

在这两种实现方式中，LazyLoadPageControlNavigation 实现了延迟加载。很显然，LazyLoadPageControlNavigation 的延迟加载友好很多。那么，两者究竟有多大的差别，这是可以量化的。通过 Instruments 工具的 Allocations 模板，可以分析 ViewController 视图控制器加载时内存占用方面的差别。图17-23是无延迟加载实现案例的 Allocations 模板跟踪，图17-24是延迟加载实现案例的 Allocations 模板跟踪。

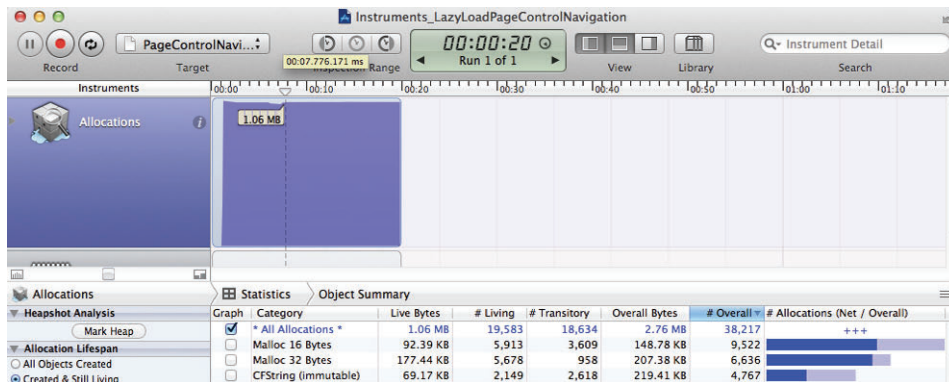


图17-23 无延迟加载实现案例的 Allocations 模板跟踪

如图 17-23 所示，界面启动用时 00:07.776.171ms，内存占用 1.06MB。如图 17-24 所示，界面启动用时 00:07.513.671ms，内存占用 1.04MB。可见，无延迟加载比有延迟加载界面启动时间要长，内存占用要多 0.02MB，即约等于 21KB。后面的 3 个图片文件 animal-2.png、animal-3.png 和 animal-4.png 的大小分别是 7KB、8KB 和 6KB，它们加在一起刚好是 21KB。

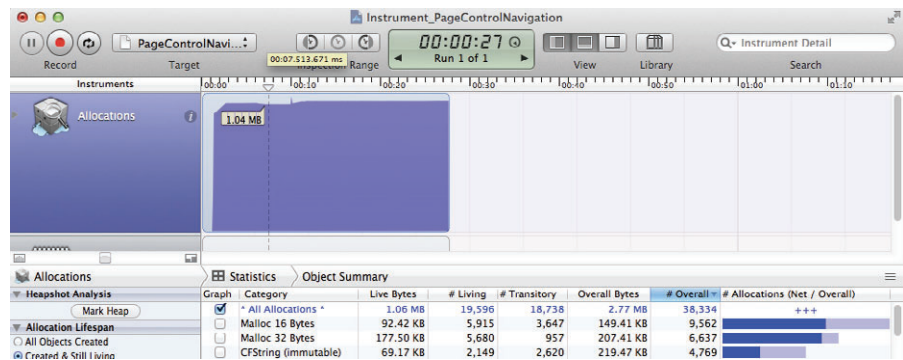


图17-24 延迟加载实现案例的Allocations模板跟踪

在上面的案例中可以发现，延迟加载的优势很明显。如果一定会访问到资源文件，则延迟加载这些资源文件时，内存占用方面就没有优势了，但是在界面加载速度方面还是有优势的。

17.3.2 故事板和nib文件的延迟加载

nib 和故事板也都属于资源文件，它是非常特殊的资源文件，应用不仅需要读取它，而且要根据里面描述的信息创建视图和子视图，以及它们的视图控制器等对象。创建这么多对象会耗费很多时间，占用很多内存，因此，它们的延迟加载问题非常重要。

1. 故事板文件延迟加载

默认情况下，创建基于故事板的应用时，只有一个故事板文件（MainStoryboard.storyboard）。这种情况下，故事板内部的视图控制器的创建和加载都是由Segue来控制的，Segue会帮助我们管理好这些控制器，包括延迟加载等问题。我们使用Xcode的Utility Application模板创建一个基于故事板的工程StoryboardLazyLoadDemo，研究故事板的延迟加载机理。创建的工程如图 17-25 所示。

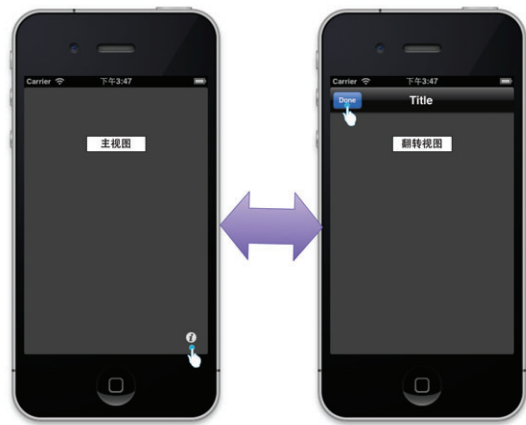


图17-25 使用Utility Application模板创建的工程

在主视图中点击i按钮时，MainViewController会延迟加载FlipsideViewController，然后弹出模态模式。实现代码是StoryboardLazyLoadDemo工程的MainViewController.m中的prepareForSegue:sender:方法，具体如下：

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    if ([[segue identifier] isEqualToString:@"showAlternate"]) {
        [[segue destinationViewController] setDelegate:self];
    }
}
```

这里我们根本看不到FlipsideViewController的影子，导航到FlipsideViewController是通过故事板文件中的Segue定义的，如图17-26所示。

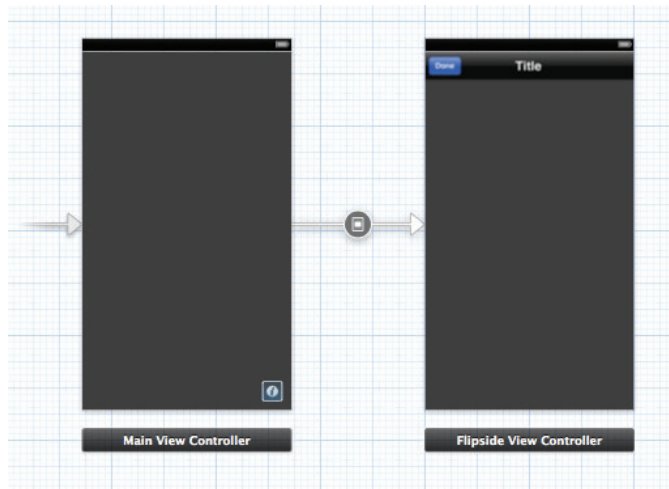


图17-26 模态视图的Segue

Segue定义的两个视图控制器的导航关系，也用来维护和管理下一个视图控制器的延迟加载时机，这种情况下我们无法“插手”视图控制器的延迟加载。但是一种情况除外，那就是使用了故事板，而控制器之间没有定义导航关系，没有定义Segue，如图17-27所示。

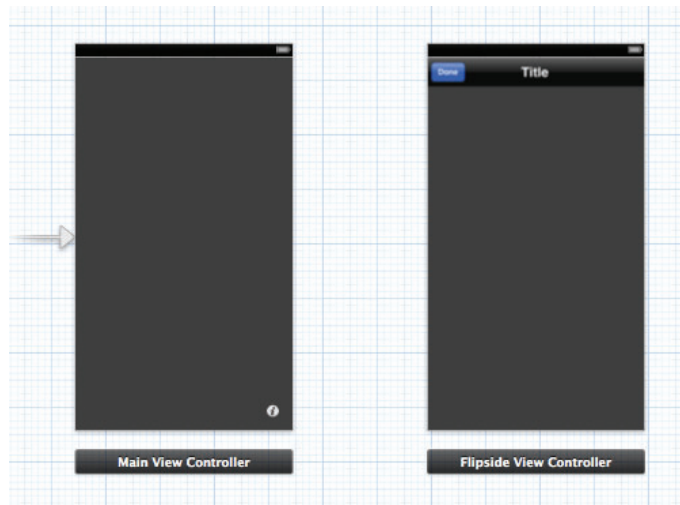


图17-27 没有定义Segue的故事板

这种情况下，添加showInfo:方法响应主视图的i按钮点击事件，具体代码可参考StoryboardLazyLoadDemo工程的MainViewController.m:

```
- (IBAction)showInfo:(id)sender
{
    //获得当前故事板对象
    UIStoryboard *mainStoryboard = [self storyboard];
    //从故事板文件中创建MainStoryboard故事板对象
    //UINavigationController *mainStoryboard = [UINavigationController storyboardWithName:@"MainStoryboard"
    bundle:nil];

    //通过名为flipsideViewController的Storyboard ID创建视图控制器对象
    FlipsideViewController* controller = [mainStoryboard
    instantiateViewControllerWithIdentifier:@"flipsideViewController"];

    controller.delegate = self;
    controller.modalTransitionStyle = UIModalTransitionStyleFlipHorizontal;
    [self presentViewController:controller animated:YES completion:nil];
}
```

在单一故事板文件中，[self storyboard]语句可以获得当前故事板对象。如果想在多故事板的情况下获得非当前故事板对象，可以通过[UINavigationController storyboardWithName:@"MainStoryboard" bundle:nil]语句创建。本例中不用使用该语句，使用它会多创建一个故事板对象，就会占用更多的内存。

2. nib文件延迟加载

相对于故事板而言，nib要灵活很多。nib文件有两种：一种是描述视图控制器的，另一种是描述视图的，它们的加载方式有所区别。无论是哪一种，分散管理的nib文件使我们通过编程方式访问它更加方便。我们同样使用Xcode的Utility Application模板创建一个基于nib的工程NibLazyLoadDemo。实现代码是MainViewController.m中的showInfo:方法，具体如下：

```
- (IBAction)showInfo:(id)sender
{
    FlipsideViewController *controller = [[FlipsideViewController alloc]
    initWithNibName:@"FlipsideViewController" bundle:nil];
    controller.delegate = self;
    controller.modalTransitionStyle = UIModalTransitionStyleFlipHorizontal;
    [self presentViewController:controller animated:YES completion:nil];
}
```

本例中的nib文件是视图控制器nib文件，我们可以使用视图控制器的initWithNibName:bundle:方法从nib文件中创建视图控制器对象。

下面的代码是从MyView的nib文件创建一个视图对象，这里的nib文件就是描述视图的：

```
NSArray* nibViews = [[NSBundle mainBundle] loadNibNamed:@"MyView"
    owner:self
    options:nil];
```

返回结果是NSArray集合。我们还需要遍历集合，通过isKindOfClass:判断目标视图类型，代码如下：

```
MyView *view;
for (id object in nibViews) {
    if ([object isKindOfClass:[ MyView class]])
        view = (MyView *)object;
}
```

有些情况下，故事板和nib会混合使用。在有故事板的工程中，有时候需要使用别人已经编写好的nib文件和对应类（视图或视图控制器）。当然，通过上面的两种方式也是可以的。

17.4 数据持久化的优化

在iOS中，数据持久化的载体主要有文件、SQLite数据库和Core Data。本节中，我们就从这几个方面入手讨论数据持久化的优化问题。

17.4.1 使用文件

文件是数据持久化的重要载体。文件优化可以包括很多方面，下面我们从文件访问、文件结构和文件大小这3个方面来介绍。

1. 文件访问优化

避免多次写入很少的数据，最好是当数据积攒到一定数量时一次写入。因为文件访问涉及IO操作，我们知道频繁的IO操作会影响性能，所以最好将文件读写访问从主线程中剥离出来，由一个子线程负责。另外，过于频繁地写入数据会影响设备中闪存的寿命。

文件的写入应该采用增量方式，每次只写入变化的部分，不要为改变几个字节写入整个文件。这样就要求不能采用简单的属性列表对象写入方式。这是一个很复杂的问题，文件内容的变化可以是追加、删除和修改。文件追加很容易实现，删除就比较麻烦了，需要找到要删除的数据，这样访问文件就采用随机访问方式了。修改与删除的问题是一样的。与其这么麻烦，不如采用别的持久化技术了。

2. 文件结构优化

文件要保存数据，它就应该结构化的。苹果中的.plist文件就是很好的结构化文件，其结构是层次模型的树形结构，层次的深浅会影响读取/写入的速度。在能够满足用户需求的情况下，要减少层次深度。下面是一个世界杯足球赛部分小组信息的属性列表文件team（5层次）.plist：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">
<array>
  <dict>
    <key>teamname</key>
    <string>A</string>
    <key>tearmlist</key>
    <array>
      <dict>
        <key>name</key>
        <string>南非</string>
        <key>image</key>
        <string>SouthAfrica</string>
      </dict>
      <dict>
        <key>name</key>
        <string>墨西哥</string>
        <key>image</key>
        <string>Mexico</string>
      </dict>
    </array>
  </dict>
  <dict>
    <key>teamname</key>
    <string>B</string>
    <key>tearmlist</key>
    <array>
      <dict>
        <key>name</key>
        <string>阿根廷</string>
        <key>image</key>
```

```
        <string>Argentina</string>
      </dict>
    <dict>
      <key>name</key>
      <string>尼日利亚</string>
      <key>image</key>
      <string>Nigeria</string>
    </dict>
  </array>
</dict>
</array>
</plist>
```

该文件有5个层次，具体如图17-28所示，其中第一层是数组类型集合；第二层是字典集合，其中描述了小组名和小组中的球队列表；第三层是数组类型集合，描述了小组中的球队列表；第四层是字典集合；第五层是字符串，描述了球队名和球队图标信息。

Key	Type	Value	
▼ Root	Array	(2 items)	①
▼ Item 0	Dictionary	(2 items)	②
tearmname	String	A	
▼ teamlist	Array	(2 items)	③
▼ Item 0	Dictionary	(2 items)	④
name	String	南非	⑤
image	String	SouthAfrica	
▼ Item 1	Dictionary	(2 items)	
name	String	墨西哥	
image	String	Mexico	
▼ Item 1	Dictionary	(2 items)	
tearmname	String	B	
▼ teamlist	Array	(2 items)	
▼ Item 0	Dictionary	(2 items)	
name	String	阿根廷	
image	String	Argentina	
▼ Item 1	Dictionary	(2 items)	
name	String	尼日利亚	
image	String	Nigeria	

图17-28 5个层次的team.plist文件

这个文件访问起来很不方便，遍历起来也很不方便，也很影响性能。我们重新设计了这个属性列表文件，其内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">
<array>
  <dict>
    <key>name</key>
    <string>A1-南非</string>
    <key>image</key>
    <string>SouthAfrica</string>
  </dict>
  <dict>
    <key>name</key>
    <string>A2-墨西哥</string>
    <key>image</key>
    <string>Mexico</string>
  </dict>
  <dict>
    <key>name</key>
```

```

        <string>B1-阿根廷</string>
        <key>image</key>
        <string>Argentina</string>
    </dict>
</dict>
    <key>name</key>
    <string>B2-尼日利亚</string>
    <key>image</key>
    <string>Nigeria</string>
</dict>
</array>
</plist>

```

此时这个文件有3个层次，其中第一层是数组类型集合，第二层是字典集合，第三层是字符串，描述了球队名和球队图标信息，如图17-29所示。

Key	Type	Value
▼ Root	Array	(4 items) ①
▼ Item 0	Dictionary	(2 items) ②
name	String	A1-南非 ③
image	String	SouthAfrica
▼ Item 1	Dictionary	(2 items)
name	String	A2-墨西哥
image	String	Mexico
▼ Item 2	Dictionary	(2 items)
name	String	B1-阿根廷
image	String	Argentina
▼ Item 3	Dictionary	(2 items)
name	String	B2-尼日利亚
image	String	Nigeria

图17-29 3层次的team.plist文件

与上面的5层次文件相比，3层次访问起来比较方便，性能会比较好。此外，在文件大小方面，3层次文件是647KB，5层次文件是893KB。

3. 文件大小优化

文件大小也是优化的一个重要指标。从上面的比较可以看到，调整文件结构可以减少文件大小。此外，我们也可以通过序列化.plist文件减少文件大小。Foundation框架提供了NSPropertyListSerialization类，它就是为此而设计的。NSPropertyListSerialization类中有2个常用方法，具体如下所示。

- ❑ **+ dataWithPropertyList:format:options:error:**。按照指定的格式和操作参数，序列化属性列表对象到NSData对象。
- ❑ **+ propertyListWithData:options:format:error:**。按照指定的格式和操作参数，从NSData对象反序列化到属性列表对象中。

为了介绍NSPropertyListSerialization类，我们修改一下10.2节的工作空间，数据原来保存在NotesList.plist属性列表文件中，现在我们换成序列化二进制文件NotesList.binary。下面我们修改数据持久层工程PersistenceLayer中的NoteDAO.m文件。首先，添加如下两个方法：

```

//从文件中读取数据到NSMutableArray
- (NSMutableArray*) readFromArray: (NSString*) path
{
    //从文件读取到NSMutableData
    NSMutableData *data = [[NSMutableData alloc] initWithContentsOfFile:path];
    //反序列化到属性列表对象 (NSMutableArray)
    NSMutableArray* array = [NSPropertyListSerialization propertyListWithData:data
        options:NSPropertyListMutableContainersAndLeaves format: NULL error:NULL]; ①
}

```

```

    return array;
}

//写入NSMutableArray数据到文件中
-(void) write:(NSMutableArray*)array toFilePath: (NSString*) path
{
    //把属性列表对象 (NSMutableArray) 序列化为NSData
    NSData * data = [NSPropertyListSerialization dataWithPropertyList:array format:
        NSPropertyListBinaryFormat_v1_0 options:NSPropertyListMutableContainersAndLeaves
        error:NULL];
    ②

    //写入到沙箱目录的序列化文件
    BOOL success = [data writeToFile:path atomically:YES];

    if (!success) {
        NSAssert(0, @"错误写入文件");
    }
}

```

在上述代码中，readFromArray:方法从文件中读取数据到NSMutableArray，流程是读取文件到NSData对象，然后再从NSData对象中反序列化处理属性列表对象。本例中的属性列表对象是NSMutableArray类型，其中第①行代码用于处理这一个过程。propertyListWithData后面的参数是反序列化的数据来源，它是NSData类型。options后面的参数是NSPropertyListMutableContainersAndLeaves，这个参数是枚举类型NSPropertyListMutabilityOptions中定义的常量，这几个常量的说明如下所示。

- ❑ **NSPropertyListImmutable**。属性列表包含不可变对象。
- ❑ **NSPropertyListMutableContainers**。属性列表父节点是可变类型，子节点是不可变类型。
- ❑ **NSPropertyListMutableContainersAndLeaves**。属性列表父节点和子节点都是可变类型。

由于在我们的属性列表文件中所有的数据项目都是可变的，所以使用了NSPropertyListMutableContainersAndLeaves常量。在第①行代码中，format后面的参数为NULL，说明格式是自动识别的。

提示 属性列表对象是与属性列表文件结构对应的，它可以NSData、NSString、NSArray和NSDictionary类型以及它们的可变类型。此外，还可以是NSDate和NSNumber类型。

writeToFile:方法把NSMutableArray数据序列化后写入到文件中，流程是先序列化NSMutableArray数据到NSData对象中，然后在把NSData对象写入到文件中。第②行代码就是完成序列化处理的，其中dataWithPropertyList后面的参数是要序列化的属性列表对象，format后面的参数是NSPropertyListFormat枚举类型。NSPropertyListFormat枚举类型包含的常量有如下几个。

- ❑ **NSPropertyListXMLFormat_v1_0**。指定属性列表文件格式是XML格式，仍然是纯文本类型，不会压缩文件。
- ❑ **NSPropertyListBinaryFormat_v1_0**。指定属性列表文件格式为二进制格式，文件是二进制类型，会压缩文件。
- ❑ **NSPropertyListOpenStepFormat**。指定属性列表文件格式为ASCII码格式，对于旧格式的属性列表文件，不支持写入操作。

本例中，我们设置的是NSPropertyListBinaryFormat_v1_0，大小减少了，加载速度提高了，这样就达到了优化的效果。关于本例中其他方法的改动，这里我们就不再介绍了。

17.4.2 使用SQLite数据库

当需要处理较大的数据集时，就不能采用文件了。因为文件不支持事务处理，这时候我们可以选择SQLite

数据库或Core Data。本节中，我们先从表结构、查询和插入（或删除）这几个方面介绍一下SQLite数据库方面的优化。

1. 表结构优化

SQLite是嵌入式关系型数据，它可以建立多表之间复杂的关系，但是如果放在iOS、Android等这些移动设备上时，我们需要考虑设备上本地表能建多少，表中字段有多少，表之间关系的复杂程度等问题。

在CPU处理能力低、内存少、存储空间少的情况下，我们不能在本地建立复杂表关系，表的个数不要超过5个，表中的字段数也不宜太多。移动设备中的数据不可能是企业级系统数据的全部，它只是企业级系统的补充和扩展。例如，在你的iPhone手机中不可能有全部的新浪微博用户信息，一方面是不安全，另一方面是数据量很大，最高配置的iPhone也不可能存放下这么多数据。这是我们在开发移动应用时始终要牢记的：移动设备在整个应用系统中的角色是什么？

2. 查询优化

查询是衡量数据库性能的重要指标之一。在查询方面可优化的有很多，例如建立索引、限制返回记录数和where条件子句等。

使用索引，能够提高查询的性能。具体哪些字段需要创建索引，这很关键。只有在表连接或where条件子句中使用字段时，才能提高查询性能。在INTEGER PRIMARY KEY字段上，一般不用建索引。如果表中的数据很少，则建索引的效果不大。

由于移动设备屏幕相对来说比较小，屏幕上能显示的数据不多，如果一次查询出的记录数超过屏幕能显示的行数，这就没有必要了，因为这样反而会占用更多的内存，耗费宝贵的CPU时间。因此，我们需要为查询添加返回记录数的限制。下面的语句是SQLite支持的写法：

```
SELECT * FROM Note Limit 10 Offset 5;
```

以上语句表示从Note表查询数据出来，其中10表示查询的最大记录数不超过10个，5表示偏移量，即跳过5行取10个。

在where条件子句的优化方面，就有更多优化方式了。比如，尽量不要使用Like模糊匹配查询，如果可能，则使用=号查询；尽量不要使用IN语句，可以使用=号和or替代。此外，在多个条件中，要把非文本的条件放在前面，文本条件放在后面，示例代码如下：

```
(salary > 5000000) AND (lastName LIKE 'Guan') 优于 (lastName LIKE 'Guan') AND (salary > 5000000)
```

这是因为非文本的条件判断比较快，如果不满足，就不再计算后面的条件表达式了。

3. 插入（或删除）优化

索引可以提供查询性能，但是对于插入和删除是有负面影响的。索引就像是书中的目录，插入和删除数据必然造成索引重排，所以创建索引要慎重。

在SQLite中，有一些PRAGMA指令可以改变数据库的行为。PRAGMA synchronous指令用于设置数据同步操作。同步是指在插入数据时，将数据同时保存到存储介质中。如果PRAGMA synchronous = OFF，则表示关闭了数据同步，不等待数据保存到存储介质就可继续执行插入操作，这在大量数据插入时可以大大提高速度。在Objective-C中，可以调用sqlite3_exec函数设置数据是否同步，相关语句如下：

```
sqlite3_open(DATABASE, &db);
sqlite3_exec(db, "PRAGMA synchronous = OFF", NULL, NULL, &err);
```

插入完成后，也可以重新设置PRAGMA synchronous = NORMAL或PRAGMA synchronous = FULL。

17.4.3 使用Core Data

Core Data是面向对象的ORM技术，苹果公司推荐使用。它提供了缓冲、延迟加载等技术，其性能比较好，但有时候我们会发现它的性能要比SQLite差，这主要与存储类型的设置有关。Core Data的存储类型有

NSSQLiteStoreType、NSBinaryStoreType和NSInMemoryStoreType，我们主要采用NSSQLiteStoreType类型，这样底层存储就采用了SQLite数据库，SQLite数据库的优点也能发挥出来。

使用Core Data时，还要考虑查询优化问题。它的查询是通过NSFetchRequest执行Predicate定义的逻辑查询条件实现的，在优化规则上与SQLite的where条件子句是一样的。此外，如果要查询返回记录数的限制，可以使用如下语句：

```
NSFetchRequest *request = [[NSFetchRequest alloc] init];
//限制一次提取记录数
[request setFetchLimit:10];
//限制提取记录偏移量
[request setFetchOffset:5];
```

这两条语句相当于SELECT * FROM Note Limit 10 Offset 5;。

此外，还可以设置pragma指令，相关语句如下：

```
NSMutableDictionary *pragmaOptions = [NSMutableDictionary dictionary];
[pragmaOptions setObject:@"OFF" forKey:@"synchronous"];
[pragmaOptions setObject:@"OFF" forKey:@"count_changes"];
[pragmaOptions setObject:@"MEMORY" forKey:@"journal_mode"];
[pragmaOptions setObject:@"MEMORY" forKey:@"temp_store"];

NSDictionary *storeOptions = [NSDictionary dictionaryWithObject:pragmaOptions
forKey:NSSQLitePragmasOption];
NSPersistentStore *store;
NSError *error = nil;
store = [psc addPersistentStoreWithType:NSSQLiteStoreType
configuration:nil
URL:url
options:storeOptions
error:&error];
```

在上述代码中，我们首先把这些pragma指令放置于NSMutableDictionary可变字典中，然后以NSSQLitePragmasOption为键再把指令设置到可变字典中。NSPersistentStore对象的addPersistentStoreWithType:configuration:URL:options:error:方法的options参数用于接收设置的NSDictionary对象。

为了方便分析Core Data的执行情况，我们可以使用Instruments工具中的Core Data跟踪模板，如图17-30所示。

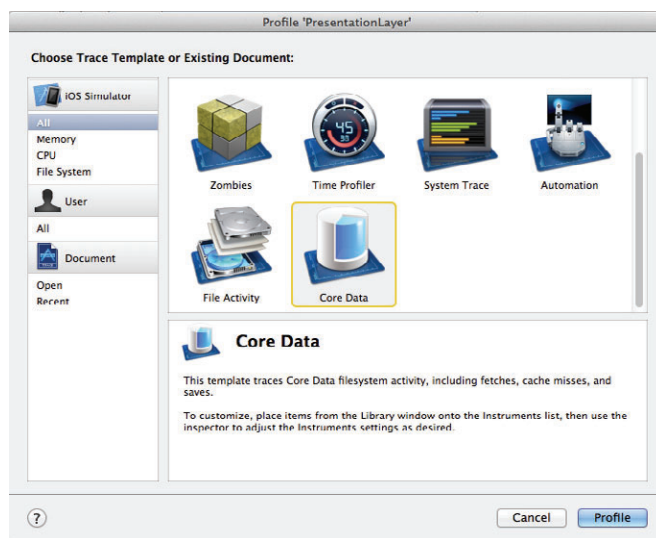


图17-30 选择Core Data跟踪模板

进入Core Data跟踪模板后，如图17-31所示，可以看到内部有3个跟踪项目：Core Data Fetches、Core Data Cache和Core Data Saves。

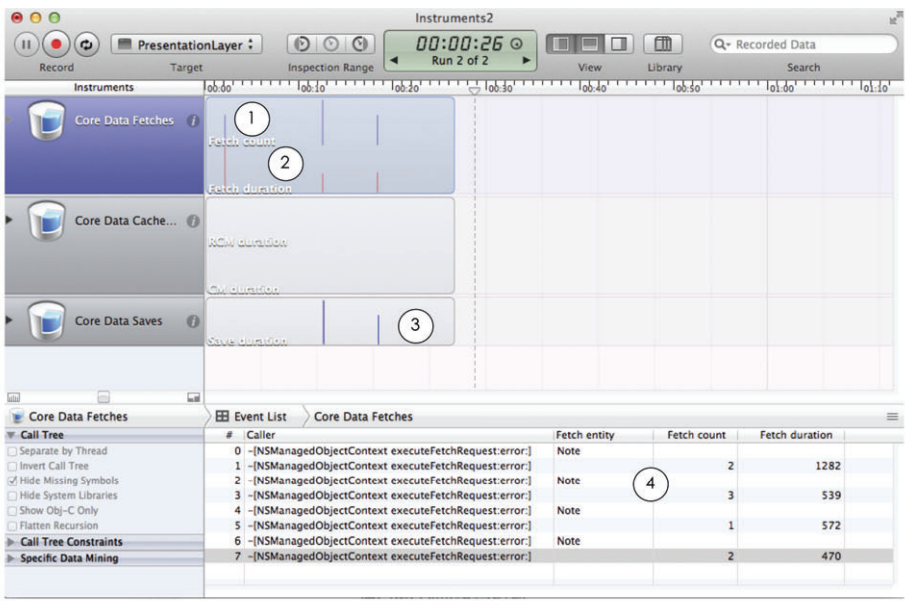


图17-31 Core Data跟踪模板

当我们执行查询、插入和删除操作时，在Core Data Fetches和Core Data Saves的跟踪项目右边会产生很多线。其中，①部分的蓝色线段为Fetch count（查询的记录数据）；②部分的红色线段为Fetch duration（执行查询的持续时间），将虚线拉到上面可以看到这些内容的具体数值；如果有数据要保存，③部分产生的蓝色线段为Save duration（保存持续时间）；④部分是更具体的信息，Fetch entity列是查询的实体类（Note），Fetch count列是查询的记录数，Fetch duration列是查询的执行时间。

17.5 可重用对象的使用

在iOS的一些视图中，它们的内部包含了子视图，当父视图显示区域发生变化时（如用手滑动屏幕），原来在屏幕中的子视图就会滑出到屏幕之外，而原来在屏幕之外的子视图就有机会进入屏幕中。如图17-32所示，当屏幕向上滑动时，Cupertino单元格滑出屏幕之外，Sherman Oaks单元格滑入到屏幕中。

这些操作会有什么问题吗？这在第4章讨论过。如果每次新单元格进入到屏幕中都去实例化一个新的单元格，必然增加内存开销。采用可重用对象设计可以不去实例化新的单元格，而是先使用可重用单元格标识到视图去找，如果找到则使用，没有则创建。

在iOS 6中，可以使用可重用对象的父视图有表视图、集合视图（UICollectionView）和地图视图（MKMapView）。由于地图的相关知识介绍超出本书的范围，我们就不再介绍相关内容了。

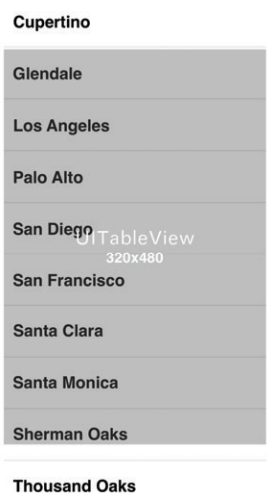


图17-32 表视图中的可重用单元格

17.5.1 表视图中的可重用对象

在iOS 6中,表视图中有两种子视图采用可重用对象设计,它们是表视图单元格 (UITableViewCell) 和表视图节头节脚视图 (UITableViewHeaderFooterView)。

1. 表视图单元格

表视图单元格的重用方法有两个: dequeueReusableCellWithIdentifier:方法和 dequeueReusableCellWithIdentifier:forIndexPath:方法。

通过dequeueReusableCellWithIdentifier:方法,可以用标识符从表视图中获得可重用单元格,模式代码如下:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"CellIdentifier";
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    }
    .....
    return cell;
}
```

要在表视图数据源的tableView:cellForRowAtIndexPath:方法中使用可重用单元格设计,首先通过dequeueReusableCellWithIdentifier:方法从表视图中找,如果cell为空,则需要使用initWithStyle:reuseIdentifier:构造方法创建。

dequeueReusableCellWithIdentifier:forIndexPath:方法是iOS 6之后提供的方法。与上一个方法比,该方法的签名多了forIndexPath:部分。它可以通过指定单元格位置获得可重用单元格,不需要判断,模式代码如下:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"CellIdentifier";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        CellIdentifier forIndexPath:indexPath];
    .....
    return cell;
}
```

这个方法的使用有一些限制,它只能应用于iOS故事板中,并且在故事板中设计表视图单元格后,指定表视图单元格为动态的,Identifier属性设置为CellIdentifier。图17-33设定了表视图单元格的属性。

2. 表视图节头节脚视图

UITableViewHeaderFooterView也是iOS 6中新加的内容,节头和节脚也会反复出现,它也需要可重用设计。使用表视图的dequeueReusableHeaderFooterViewWithIdentifier:方法获得UITableViewHeaderFooterView对象,如果没有可重用的UITableViewHeaderFooterView对象,则使用initWithReuseIdentifier:构造方法创建。其模式代码如下:

```
- (UIView *)tableView:(UITableView *)tableView viewForHeaderInSection:(NSInteger)section
{
    static NSString *headerReuseIdentifier =
        @"TableViewSectionHeaderViewIdentifier";

    UITableViewHeaderFooterView *sectionHeaderView = [tableView
        dequeueReusableCellWithIdentifier:headerReuseIdentifier];
```

```

if (!sectionHeaderView) {
    sectionHeaderView = [[UITableViewHeaderFooterView alloc]
        initWithReuseIdentifier:headerReuseIdentifier];
}

.....

return sectionHeaderView;
}

```

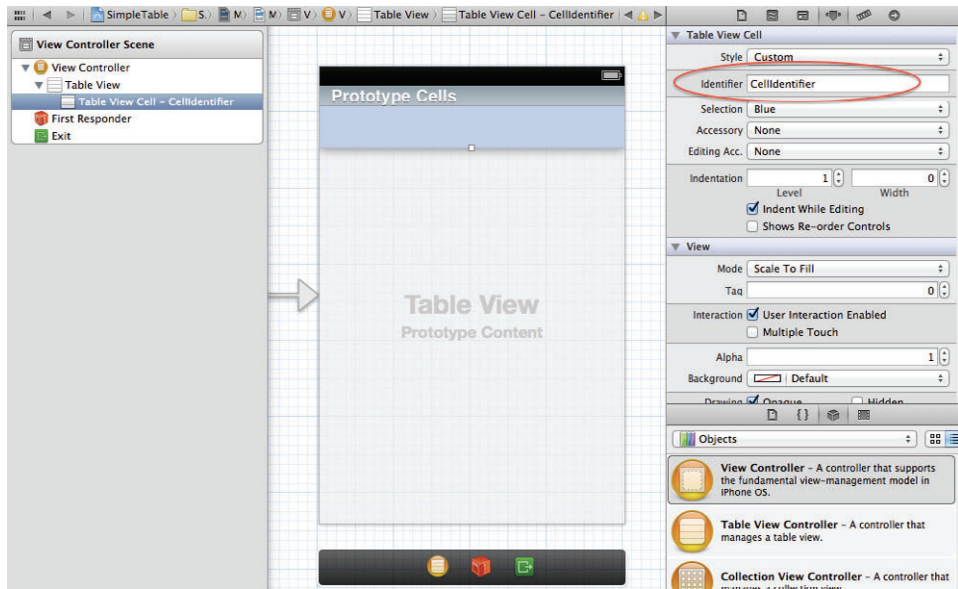


图17-33 设定表视图单元格的属性

需要在表视图委托协议UITableViewDelegate中的tableView:viewForHeaderInSection:方法中使用可重用对象设计。

17.5.2 集合视图中的可重用对象

集合视图在iOS 6中才可以使用。它也有两种子视图采用可重用对象设计，它们是单元格视图和补充视图，这两个视图都继承UICollectionViewReusableView，使用时需要自己编写相关代码。

1. 单元格视图

在集合视图中，我们可以使用UICollectionView的dequeueReusableCellWithIdentifier:forIndexPath:方法获得可重用的单元格，模式代码如下：

```

- (UICollectionViewCell *)collectionView:(UICollectionView *)
    collectionView cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
    Cell *cell = [collectionView dequeueReusableCellWithIdentifier:
        @"CellIdentifier" forIndexPath:indexPath];
    .....
    return cell;
}

```

在上述代码中，collectionView:cellForItemAtIndexPath:方法是集合视图数据源方法，其中Cell是我们自定义的继承自UICollectionViewReusableView的单元格类。使用dequeueReusableCellWithIdentifier:

时，需要使用故事板设计UI，并且需要将单元格的Identifier属性设置为CellIdentifier（如图17-34所示）。

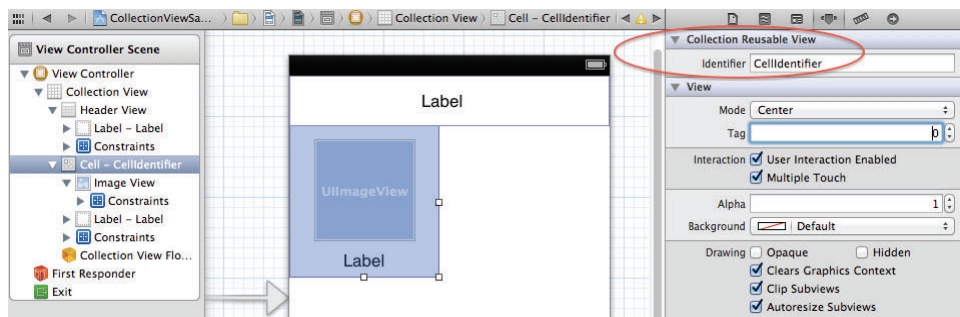


图17-34 设定集合视图单元格的属性

2. 补充视图

集合视图单元格可以使用 UICollectionView 的 dequeueReusableSupplementaryViewOfKind:withReuseIdentifier:forIndexPath:方法获得可重用的补充视图，模式代码如下：

```
- (UICollectionViewReusableView *)collectionView:(UICollectionView *)collectionView
viewForSupplementaryElementOfKind:(NSString *)kind atIndexPath: (NSIndexPath *)indexPath
{
    HeaderView *headerView = [collectionView
        dequeueReusableSupplementaryViewOfKind:UICollectionViewElementKindSectionHeader
        withReuseIdentifier:@"HeaderIdentifier" forIndexPath:indexPath];

    headerView.headerLabel.text = [self.eventDate objectAtIndex:indexPath.section];

    return headerView;
}
```

collectionView:viewForSupplementaryElementOfKind:atIndexPath:方法是集合视图的数据源方法，其中HeaderView是我们自定义的继承自UICollectionViewReusableView的补充视图类。使用 dequeueReusableSupplementaryViewOfKind:withReuseIdentifier:forIndexPath:时，需要使用故事板设计UI，并将补充视图的Identifier属性设置为HeaderIdentifier（如图17-35所示）。

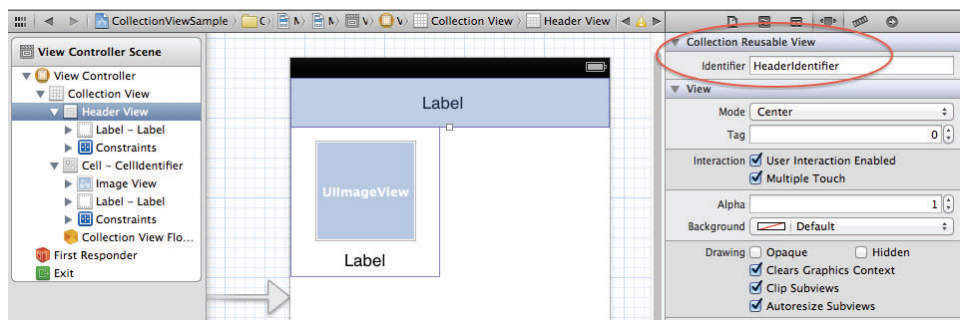


图17-35 设定补充视图的属性

17.5.3 地图视图中的可重用对象

在开发地图应用时，也有一个可重用对象MKPinAnnotationView，它是在地图上的一个标注。使用地图视图的dequeueReusableAnnotationViewWithIdentifier:方法，可以获得MKPinAnnotationView对象。如

果没有可重用的MKPinAnnotationView对象,则使用initWithAnnotation:reuseIdentifier:构造方法创建。其模式代码如下:

```

MKPinAnnotationView *annotationView
= (MKPinAnnotationView *)[_mapView
    dequeueReusableAnnotationViewWithIdentifier:@"PIN_ANNOTATION"];
if(annotationView == nil) {
    annotationView = [[MKPinAnnotationView alloc] initWithAnnotation:annotation
        reuseIdentifier:@"PIN_ANNOTATION"];
}

```

这段处理代码是地图视图中常用的处理方式,请大家牢记。

17.6 并发处理与多核 CPU

并发处理能够同时处理多个任务。在CPU单核时代,可以使用多线程技术进行并发处理。现在,iOS设备的CPU已经进入多核时代,从iPhone 4S和iPad 2之后开始采用A5双核CPU设计。异步设计方法可以充分地发挥多核优势。GCD(Grand Central Dispatch)是一种异步方法,是专为多核CPU而设计的并发处理技术。

本节中,我们就来介绍主线程阻塞问题以及GCD的相关内容。

17.6.1 主线程阻塞问题

主线程所做的事情应该是响应用户输入、事件处理、更新UI,而耗时的任务不要在主线程中处理。由于耗时任务使得主线程被阻塞了,不能响应用户的请求,这样应用的用户体验会很差。

下面我们先看一个例子。如图17-36所示,点击Load Image按钮,会从http://www.51work6.com/ios_book/animals/animal-3.png中加载图片。当我们点击Load Image按钮时,按钮会一直处于按下状态而不弹起,直到图片显示“完成”。这是因为主线程要进行耗时的处理(如进行网络通信、数据传输等任务),导致主线程不能响应用户的输入和请求,这就是我们要讨论的线程阻塞问题。

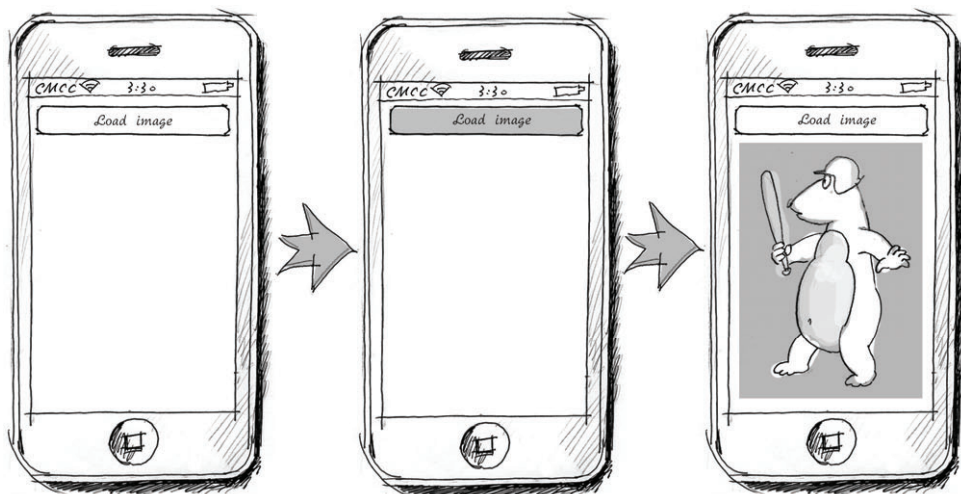


图17-36 主线程阻塞案例

下面我们看看代码部分。在BlockDemo工程中,ViewController.m中click:方法的代码如下:

```

- (IBAction)click:(id)sender {
    NSURL *imgkURL = [NSURL URLWithString:@"http://www.51work6.com/ios_book/animals/animal-3.png"];
}

```



```

NSData *imgData = [NSData dataWithContentsOfURL:imgkURL];
UIImage *img = [UIImage imageWithData:imgData];
self.imageView.image = img;

}

```

由于不能直接通过URL创建UIImage对象,因此先构建NSData对象,它是从URL请求回来的二进制数据对象,然后再用它来构建UIImage对象。最后,把UIImage对象赋值给UIImageView的image属性。

那么,如何解决阻塞主线程问题呢?那就是把这些执行比较耗时的阻塞线程的任务从主线程中移出到其他线程中处理。

17.6.2 选择NSThread还是GCD

解决主线程阻塞问题时,我们可以使用多线程。NSThread是Objective-C中的多线程类,但是麻烦的是我们需要管理线程,包括创建线程、线程间通信和销毁线程等。下面的代码是在ConcurrencyTest工程中使用NSThread创建线程,然后执行100次处理:

```

- (IBAction)testNSThread:(id)sender {
    [NSThread detachNewThreadSelector:@selector(calculationThreadEntry)
      toTarget:self withObject:nil];
}

- (void) calculationThreadEntry
{
    @autoreleasepool {
        NSUInteger counter = 0;
        while ([[NSThread currentThread] isCancelled] == NO)
        {
            [self doCalculation]; counter++;
            if (counter >= 100)
            {
                break;
            }
        }
    }
}

```

GCD是基于C语句级别的API,它提供了C函数。下面的代码是ConcurrencyTest工程中使用GCD创建管理线程,然后执行100次处理:

```

- (IBAction)testGCD:(id)sender {
    dispatch_queue_t queue = dispatch_get_global_queue
      (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    size_t numberOfIterations = 100;
    dispatch_async(queue, ^(void) { dispatch_apply(numberOfIterations,
      queue, ^(size_t iteration){
        [self doCalculation];
      });
    });
}

```

比较上面的两个例子,我们会发现NSThread类比GCD代码要烦琐,管理起来也不方便。GCD代码编写简单,还支持多核CPU处理。GCD是苹果重点推荐的并发技术,唯一的缺陷它是基于C语言的API。

在GCD中,有一个重要的概念,那就是派发队列(dispatch queue)。派发队列是一个对象,它可以接受任务,并将任务以先到先执行的顺序来执行。派发队列可以是并发的或串行的。并发队列可以执行多任务,串行队列同一时间只执行单一任务。在GCD中,有3种类型的派发队列。

❑ **串行队列。**串行队列通常用于同步访问一个特定的资源,每次只能执行一个任务。使用函数 `dispatch_queue_create`,可以创建串行队列。

- ❑ **并发队列。**也称为全局派发队列，可以并发地执行一个或多个任务。并发队列分为高、中、低3个优先级队列，中级为默认级别。可以使用调用`dispatch_get_global_queue`函数设定优先级来访问队列。在上面介绍的ConcurrencyTest工程中，`dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)`语句用于获得并发队列对象，其中`DISPATCH_QUEUE_PRIORITY_DEFAULT`是默认的优先级常量。
- ❑ **主队列。**它在应用程序的主线程中，用于更新UI。其他的两个队列不能更新UI。使用`dispatch_get_main_queue`函数，可以获得主队列。

我们把BlockDemo工程修改为GCD实现，具体代码如下：

```
- (IBAction)click:(id)sender {

    NSURL *imgkURL = [NSURL URLWithString:@"http://www.51work6.com/ios_book/animals/animal-3.png"];
    dispatch_queue_t downloadQueue = dispatch_queue_create("com.51work6.image.downloader",
        NULL);
    dispatch_async(downloadQueue, ^{
        NSData *imgData = [NSData dataWithContentsOfURL:imgkURL];
        UIImage *img = [UIImage imageWithData:imgData];
        dispatch_async(dispatch_get_main_queue(), ^{
            self.imageView.image = img;
        });
    });
}
```

在上述代码中，我们使用`dispatch_queue_create`函数创建串行队列，其中第一个参数`com.51work6.image.downloader`用于给队列指定一个名字，苹果推荐使用域名反写，这样便于查看日志信息。`dispatch_queue_t`是队列对象。`dispatch_async`函数异步执行任务，但是它在串行队列中仍然是同步执行的。在更新UI时，如`self.imageView.image = img`语句，`dispatch_async`函数必须在主线程中执行。我们使用语句`dispatch_async(dispatch_get_main_queue(), ^{...})`在主队列中更新UI，其中`^{...}`是块代码。

17.7 编译器和编译参数

不同的编译器对于编译之后的结果可以产生很大的影响。即便是相同的编译器，针对不同的平台和环境采用不同的编译参数，也会对结果产生很大的影响。下面我们从编译器、ARM架构编译平台以及编译参数Optimization Level等几个方面介绍优化。

17.7.1 GCC、LLVM GCC与Apple LLVM比较

Objective-C的编译器发展到现在，主要经历过3个阶段：GCC、LLVM GCC和Apple LLVM。

- ❑ **GCC (GNU Compiler Collection, GNU编译器套装)。**它是一套由GNU开发的编程语言编译器，也是Linux、Unix及Mac OS X操作系统的标准编译器。使用GCC，可以编译C、C++、Objective-C、Java和Pascal等语言。
 - ❑ **LLVM (Low Level Virtual Machine, 低级虚拟机)。**它提供了一套中立的中间代码和编译基础设施，并围绕这些设施提供了一套全新的编译策略，使得优化能够在编译、连接、运行环境执行。LLVM GCC是LLVM下编译C、C++和Objective-C的编译器。
 - ❑ **Apple LLVM。**它是苹果的LLVM编译器，2005年开始成为苹果官方支持的编译器。在WWDC (Worldwide Developers Conference, 苹果电脑全球研发者大会) 2010上，苹果公司报告LLVM编译器比GCC编译器快60%。在Xcode 4之后，苹果默认采用Apple LLVM编译器。
- 在Xcode中，我们可以选择编译器。如图17-37所示，在编译参数中可以选择Compiler for C/C++/Objective-C。

目前，我们使用的Xcode是4.5版本，默认的编译器是Apple LLVM compiler。此外，还可以选择LLVM GCC 4.2编译器，但是已经不推荐使用GCC了。

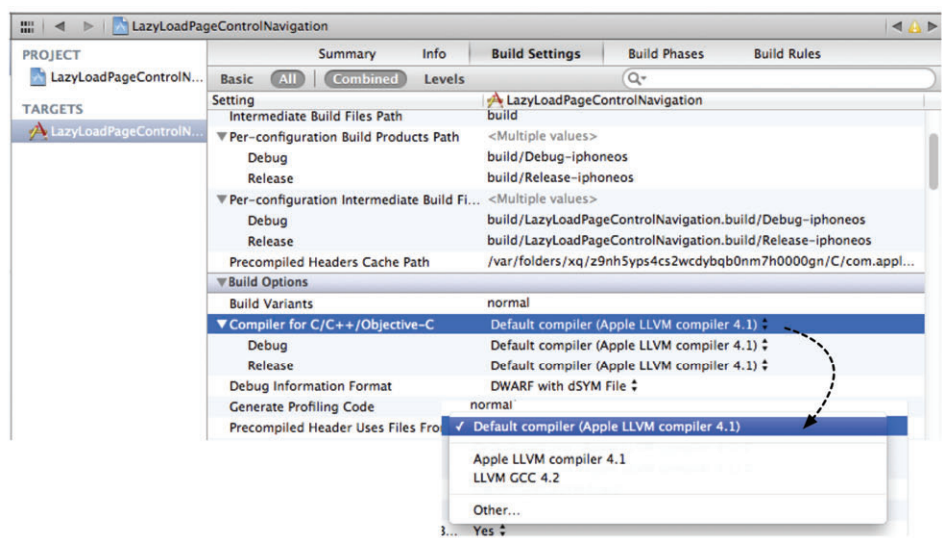


图17-37 Xcode 4.5编译器

我们对10个iOS应用程序分别采用3种不同的编译器进行测试，多次测试的平均值结果如图17-38所示，其中以GCC的运行时间作为标准值100，其他的是相对它的时间。从图表中可以看出，Apple LLVM运行时间最短。

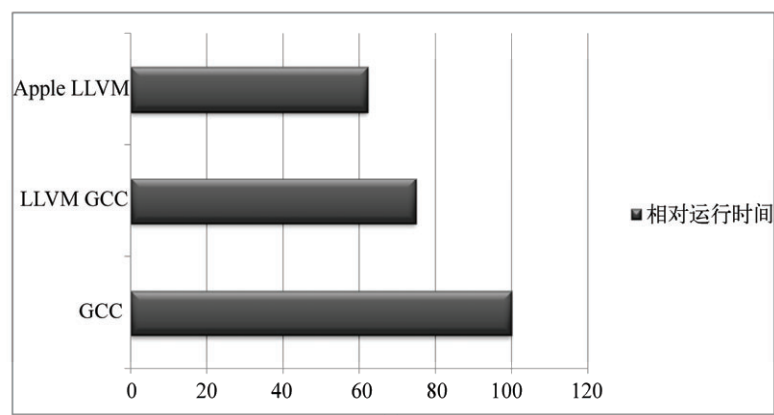


图17-38 3种编译器运行时间

17.7.2 ARM架构

在iOS中，CPU目前都采用ARM架构，分为ARMv6和ARMv7这两个版本。一些老的iOS设备（如iPhone 3G、第一代iPod touch和第二代iPod touch）的CPU采用ARMv6架构，而后来的iOS设备（如iPhone 3GS、第三代iPod touch和iPad之后）的CPU都采用ARMv7架构。选择不同的CPU架构也是一个优化手段。如图17-39所示，选择不同的编译Architectures，就意味着选择了不同的CPU架构。目前，Xcode 4.5默认支持的CPU采用ARMv7架构。图17-40是在Xcode旧版本中Architectures的设定。

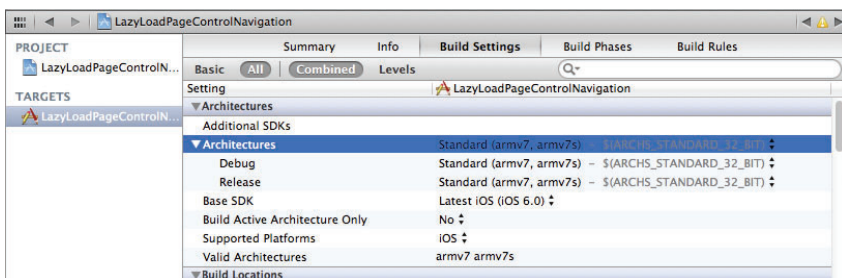


图17-39 Xcode 4.5中iOS平台的CPU架构

如果我们的应用确认只会在ARMv7下运行，那么当然要选择ARMv7。此外，我们要考虑应用也能兼容那些拿着老设备的用户，这里选择兼容ARMv6和ARMv7，如图17-40所示。

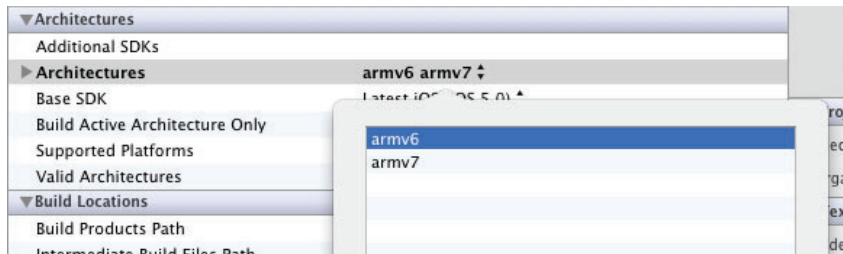


图17-40 兼容ARMv6和ARMv7

在进行浮点计算时，在ARMv6下建议禁用Thumb指令集^①。Thumb指令集有很多优点，但是在ARMv6下Thumb代码可访问的寄存器较少，缺乏条件指令，特别是它不能使用浮点硬件，如浮点加法、浮点减法和浮点乘法等。而ARMv7中包含Thumb-2指令集，它是Thumb指令的扩展集，增加了条件执行、可访问所有ARM寄存器、硬件浮点与NEON的32位Thumb指令。用Thumb-2缩减代码的代价几乎没有，因此ARMv7不需要禁用Thumb指令。我们需要在Xcode中选择Other C Flags编译参数，为ARMv6设置禁用Thumb的标识。图17-41是Xcode 4.5下的设置，其中ARMv7S architecture兼容ARMv6和ARMv7，这里设置它的标识为-mno-thumb。ARMv7不需要设置。

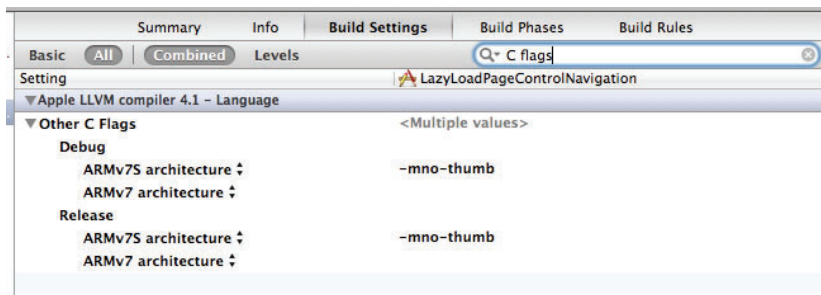


图17-41 在Xcode 4.5下设置Other C Flags编译参数

如果是Xcode 4.5之前的旧版本，其设置如图17-42所示。早期版本可以选择ARMv6，并设置ARMv6的标识为-mno-thumb。ARMv7不需要设置。

① Thumb指令集是ARM指令集的一个子集，其优点是缩小了代码尺寸，节约了内存、缓存以及内存带宽。

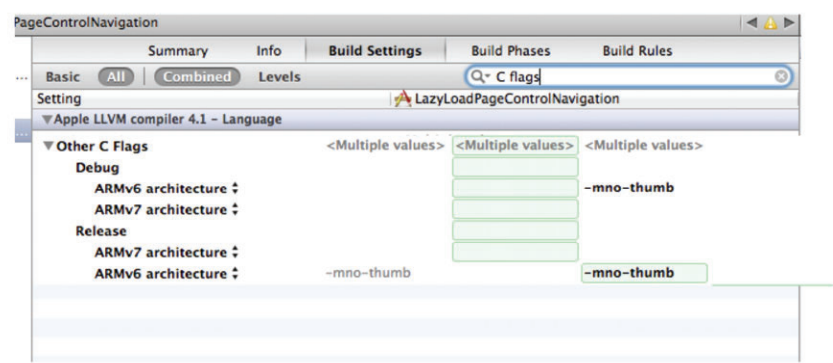


图17-42 在Xcode旧版本下设置Other C Flags编译参数

17.7.3 Optimization Level

在 Target 的编译参数 Optimization Level 中，默认情况下 Debug 设定为 None[-O0]，Release 设定为 Fastest,Smallest[-Os]，如图17-43所示。当然，我们可以改变这些选项，它们有5个级别：-O0、-O、-O1、-O2、-O3、-Os。

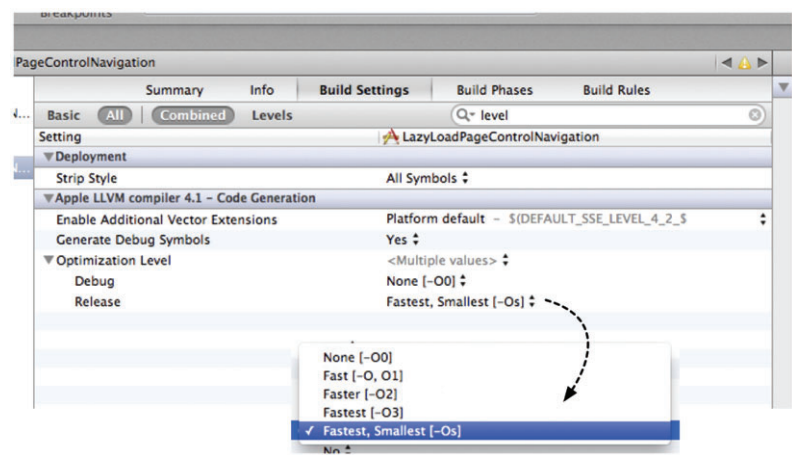


图17-43 在Xcode中设定Optimization Level编译参数

Optimization Level编译参数的作用是什么呢？它决定了程序在编译过程中的两个指标——编译速度和内存的占用，也决定了编译之后可执行结果的两个指标——速度和文件大小。如上所述，Optimization Level分为5个级别，具体为-O0、-O、-O1、-O2、-O3、-Os，这5个级别的含义如下所示。

- ❑ -O0。默认级别。不进行任何优化，直接将源代码编译到执行文件中，结果不进行任何重排，编译时间比较长。主要用于调试程序，可以进行设置断点、改变变量、计算表达式等调试工作。图17-43所示的Debug情况就是-O0级别。
- ❑ -O1（或-O）。最常用的优化级别，不考虑速度和文件大小权衡问题。与-O0级别相比，它生成的文件更小，可执行的速度更快，编译时间更少。
- ❑ -O2。在-O1级别基础上再进行优化，增加指令调度的优化。与-O1级别相比，它生成的文件大小没有变大，编译时间变长了，编译期间占用的内存更多了，但程序的运行速度有所提高。该级别是应用程序发布时的最理想级别，在增加文件大小的情况下提供了最大优化。

- -O3。在-O2和-O1级别上再进行优化，该级别可能会提高程序的运行速度，但是也会增加文件的大小。
- -Os。这种级别用于在有限的内存和磁盘空间下生成尽可能小的文件。由于使用了很好的缓存技术，它在某些情况下也会有很快的运行速度。该级别常用于发布iOS设备时，图17-43所示的就是Release为-Os级别的情况。

选择Optimization Level时，要权衡编译时间、编译内存占用、编译结果文件大小和执行速度等问题。一般情况下，-O0适合于调试，-Os级别是iOS设备上的应用发布最理想的选择。如果不满意这5个预定级别，用户可以自定义一个级别来编译。

17.8 小结

通过对本章的学习，我们了解了性能优化方法，其中包括内存优化、资源文件优化、延迟加载、持久化优化、使用可重用对象、多线程以及程序编译参数等。这些内容都是非常重要的，希望广大读者认真掌握。

18

管理好你的程序代码—— 代码版本控制

我有个朋友曾经做了一个时长3个月的项目，就在项目即将结束的时候他的硬盘坏了，数据全部丢失，而他之前从不做备份，结果可想而知。这件事对我的触动很大，我后来经常备份程序代码并将其备份在不同的电脑上。于是，有一段时间我每天下班的时候，都把程序代码备份到公司的服务器。随着时间的推移，备份到服务器上的数据越来越多，我很难快速查到想要的资料。在我与同事之间进行代码整合时，我们用U盘互相复制，由于版本不能及时更新造成了很多问题，其中必须要解决的问题有以下两点。

- ❑ **程序代码的备份。**便于查到历史版本，能够进行比较，知道修改过什么地方，是谁修改了这些代码等。

- ❑ **代码共享与整合。**能很容易得到团队其他人的代码，也能够很容易地把代码共享给其他成员。

解决这几个问题时，我们可以使用版本控制工具。版本控制工具是一种软件，开发人员要习惯使用版本控制工具，每日提交程序代码，提交的代码应该有清晰的注释，成员之间应该及时沟通。

版本控制工具很多，本章中我们为大家介绍Git代码版本控制工具。

18.1 概述

版本控制的重要性毋庸置疑，我们必须使用代码版本控制工具，每一个程序员和项目管理人员都必须深刻认识到这一点。

18.1.1 版本控制历史

版本控制的最早方式是将文件复制到文件服务器上，命名为“××年××月××日×××备份”目录，这在现在看来太原始了。作为软件工具，版本控制经历过两个阶段：集中管理模式和分布式管理模式。

- ❑ **集中管理模式。**这以一个服务器作为代码库，团队人员本地没有代码库只能与服务器进行交互。这种类型的版本控制工具有VSS（Visual Source Safe，微软开发的Microsoft Visual Studio套件中的软件之一）、CVS（Concurrent Versions System，并发版本系统）、SVN（Subversion）等，其中SVN是目前这种模式的佼佼者。

- ❑ **分布式管理模式。**这是更为先进的模式，不仅有一个中心代码库，而且每位团队人员本地也都有代码库，在不能上网的情况下也可以提交代码。该类型的版本控制工具有Git、Mercurial、Bazaar和Darcs。

Git是为了帮助管理Linux内核开发项目而开发的一个开源的版本控制工具。之前，BitKeeper工具是Linux内核开发人员使用的主要版本控制工具，它采用许可证管理版本，但Linus Torvalds^①觉得BitKeeper不适合Linux开源社区的工作，在2005年开始着手开发Git来替代BitKeeper。虽然Git最初是为了辅助Linux内核开发，但我们发现在很多其他软件项目也都可以使用Git。

① 著名的电脑程序员、黑客，Linux内核的发明人及该计划的合作者。

18.1.2 基本概念

版本控制工具涉及很多术语和概念，这里将常用的概念整理如下。

- ❑ **代码库 (repository)**。存放项目代码以及历史备份的地方。
- ❑ **分支 (branch)**。为了验证和实验一些想法、版本发布、缺陷修改等需要，建立一个开发主干之外的分支，这个分支被隔离在各自的开发线上。当改变一个分支中的文件时，这些更改不会出现在开发主干和其他分支中。
- ❑ **合并分支 (merging branch)**。完成某分支工作后，将该分支上的工作成果合并到主分支上。
- ❑ **签出 (check out)**。从代码库获得文件或目录，将其作为副本保存在工作目录下，此副本包含了指定代码库的最新版本。
- ❑ **提交 (commit)**。将工作目录中修改的文件或目录作为新版本复制回代码库。
- ❑ **冲突 (conflict)**。有时候提交文件或目录时可能会遇到冲突，当两个或多个开发人员更改文件中的一些相同行时，将发生冲突。
- ❑ **解决 (resolution)**。遇到冲突时，需要人为干预解决，这必须通过手动编辑该文件进行处理，必须有人逐行检查该文件，以接受一组更改并删除另一组更改。除非冲突解决，否则存在冲突的文件无法成功提交到代码库中。
- ❑ **索引 (index)**。Git工具特有的概念。在修改的文件提交到代码库之前被做出一个快照，这个快照被称为“索引”，它一般会暂时存储在一个临时存储区域中。

18.2 Git 代码版本控制

基于分布式管理模式的代码版本控制工具是现在的主流，而且Git还有成熟的代码托管服务GitHub网站。

18.2.1 服务器搭建

如果项目不需要与其他开发人员协同开发，我们就不需要Git服务器。与SVN和CVS不同，Git管理模式是分布的，如图18-1所示。每个开发者的本地电脑都有代码库，为了实现与他人协同开发，需要有个共享的代码库，我们把这个共享的代码库称为Git服务器。Git服务器比较特殊，它安装了一些通信协议并提供了一些安全认证，使授权客户端能够访问它的代码库。因此，任何能够提供通信协议和安全认证的代码库，都可以认为是服务器。图18-1所示的两个开发者本地代码库之间的虚线表示的就是这种情况。

为了能够与服务器进行通信，我们为服务器安装并选择通信协议。Git可以选择的协议有很多，比如本地协议、Git协议、HTTP(S)协议和SSH协议，其中本地协议主要用于本地文件系统的访问，Git协议只支持读远程库且无需身份认证，在读取远程库时它的效率最高，HTTP(S)协议和SSH协议都可以支持远程库读写、身份认证等操作。

SSH协议由于被广泛安装，因此Git通信中普遍采用SSH。我们知道SSH协议下每个用户都有一个账号，随着项目规模的扩大，开发者增加到上百人之后，服务器维护这些用户账号是非常麻烦的。Gitosis和Gitolite是基于SSH协议的Git服务器软件，它们能够使所有用户都使用同一个专用的SSH账号访问代码库，各个用户通过公钥认证的方式使用此专用SSH账号访问代码库，而用户在连接时使用的不同的公钥可以区分用户的身份。Gitosis和Gitolite除了具有SSH传统的优势外，还支持企业级授权和远程建库。

与Gitolite相比，Gitosis有很多信息需要配置，且难度很大。Gitolite是Gitosis的下一代版本，功能更加强大，配置比较简单，安装方法有多种形式，有一些比较简单，非常适合于初学者。本章重点介绍Gitolite安装、配置和管理等操作。

Linux对于SSH协议支持得很好，因此我们配置的Git服务器基于Linux的Ubuntu 11.10版本。在Ubuntu下搭建Git服务器的流程如图18-2所示。

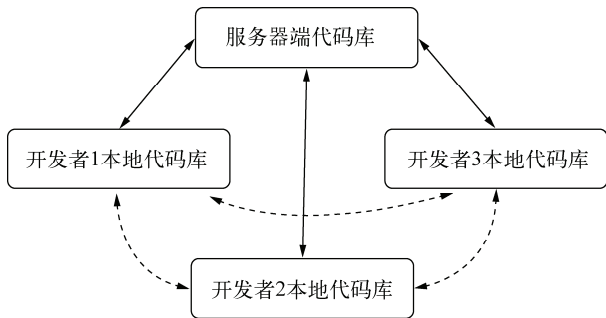


图18-1 Git分布管理模式

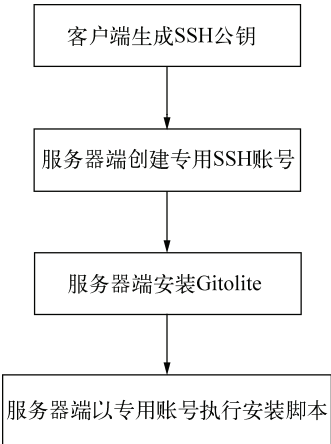


图18-2 搭建Git服务器的流程

1. 客户端生成SSH公钥

SSH采用公钥认证，通过ssh-keygen命令生成，在当前用户主目录下的.ssh目录下面生成两个私钥文件。

- ❑ id_rsa。私钥文件，基于RSA算法创建，该私钥文件要妥善保管，它被保存到客户端中。
- ❑ id_rsa.pub。公钥文件，与上面的私钥文件是一对，该文件可以公开，放置于服务器端。

采用公钥认证SSH登录，可以实现无口令登录远程服务器，即用公钥认证取代口令认证。这是我们生成SSH公钥的主要原因。我们在终端窗口中输入如下指令：

```
$ ssh-keygen

创建了自己的公钥/私钥对后，需要将服务器.ssh目录中的id_rsa.pub发送到服务器。如果我们想让当前客户端用户作为Git服务器管理者的话，在终端窗口执行如下指令：

$ cd ~/.ssh
$ scp id_rsa.pub tonyguan@192.168.1.108:/tmp/admin.pub
tonyguan@192.168.1.108's password:
id_rsa.pub                                     100% 420   0.4KB/s   00:00
```

其中tonyguan是服务器的用户，将本机上的公钥文件id_rsa.pub发送到服务器目录tmp，并重新命名为admin.pub。

提示 如果命令在执行过程中出现ssh: connect to host 192.168.1.108 port 22: Connection refused 错误，这很有可能是服务器端没有安装SSH协议。在服务器终端中输入sudo apt-get install openssh-server命令，可以安装SSH协议。

2. 服务器端创建专用SSH账号

我们需要为Git服务器访问代码库创建一个专用SSH账号，用户名为git，此时可以在服务器终端中输入如下指令：

```
$ sudo adduser git

输入密码和相关信息后，即可成功创建git应用。如果这个用户已经创建，确认它没有其他的用途后，可以删除git用户再创建，删除命令如下：

$ sudo userdel git
```

3. 服务器端安装Gitolite

上面的准备工作完成时，我们就可以在服务器端安装Gitolite软件了。Gitolite有几种安装方式，其中在Ubuntu下可以使用apt-get安装，具体指令如下：

```
$ sudo apt-get install gitolite
```

4. 服务器端以专用SSH账号执行安装脚本

这里我们的专用SSH账号是git，可以使用su切换到git用户。在服务器终端中输入如下指令，执行安装脚本：

```
$ gl-setup /tmp/admin.pub
The default settings in the rc file (/home/git/.gitolite.rc) are fine for most
people but if you wish to make any changes, you can do so now.

hit enter...
creating gitolite-admin...
Initialized empty Git repository in /home/git/repositories/gitolite-admin.git/
creating testing...
Initialized empty Git repository in /home/git/repositories/testing.git/
[master (root-commit) b9d6bb1] start
2 files changed, 6 insertions(+), 0 deletions(-)
create mode 100644 conf/gitolite.conf
create mode 100644 keydir/admin.pub
```

在这个过程中，hit enter...会进入vi文本编辑器，请修改\$REPO_UMASK = 0077为\$REPO_UMASK = 0027，然后保存并退出。脚本执行完成后，会在git主目录下创建repositories目录，在repositories目录下有gitolite-admin.git和testing.git目录。事实上，这两个子目录是两个Git代码库，其中gitolite-admin.git保存了Gitolite配置信息，它可以在客户端远程管理，testing.git代码库是为了测试而使用的。

18.2.2 Gitolite 服务器管理

Gitolite服务器可在本地管理，然后将结果推送给远程服务器，服务器会执行这些配置，从而实现管理的目的。首先，要在管理员客户端电脑上克隆服务器端gitolite-admin.git库，具体指令如下：

```
$ git clone git@192.168.1.108:gitolite-admin
```

提示 在Mac OS X下，可以到<https://code.google.com/p/git-osx-installer/>中下载.dmg文件并安装。在Ubuntu下，可以使用命令\$ sudo apt-get install git-core来安装。

进入gitolite-admin目录后，可以发现它的目录结构如下：

```
|— conf
|   |— gitolite.conf
|— keydir
|   |— admin.pub
```

该目录下面有conf和keydir两个目录，conf目录下面有gitolite.conf，keydir目录下面的admin.pub文件是之前管理员客户端传递给服务器的公钥文件。keydir目录可以存放多个用户的公钥文件，从而实现对其他用户的公钥认证。

进入gitolite-admin目录，使用vi等文本工具打开gitolite.conf文件，内容如下：

```
repo    gitolite-admin
RW+     =    admin

repo    testing
RW+     =    @all
```

在这个授权文件中，我们设置了gitolite-admin和testing这两个代码库。gitolite-admin只允许admin用户有读写

和强制更新的权限，RW为读写权限，+号为强制更新权限。testing设置测试代码库，设置为所有人都可以读写以及强制更新。

下面我们通过一个实际应用场景来介绍Gitolite的添加用户和授权修改实现过程。

我想为我们iOS开发团队创建一个组ios_team，用户有zhang、tony和zhao。服务器端的代码库是ios_repo，ios_team的组成员对代码库ios_repo有读写和强制更新的权限。

修改后的授权文件gitolite.conf的内容如下：

```
@ios_team=zhang tony zhao
repo      gitolite-admin
RW+       =      admin

repo      testing
RW+       =      @all

repo      ios_repo
RW+       =      @ios_team
```

保存后退出。然后我们需要从zhang、tony和zhao那里获得他们的公钥文件id_rsa.pub，分别将其命名为zhang.pub、tony.pub和zhao.pub，然后将其复制到gitolite-admin目录下的keydir目录中。keydir的目录结构如下：

```
keydir
├── admin.pub
├── tony.pub
├── zhang.pub
└── zhao.pub
```

然后在客户端终端进入gitolite-admin目录，输入如下命令：

```
$ git add .
$ git commit -m 'add user'
[master 6e4f3ad] add user
3 files changed, 3 insertions(+)
create mode 100644 keydir/tony.pub
create mode 100644 keydir/zhang.pub
create mode 100644 keydir/zhao.pub
```

这两个命令都是Git中的基本命令，git add .用于添加文件到本地代码库缓存，git commit用于提交缓存确定修改到本地代码库，-m参数是注释说明信息。现在的修改还只是在本地代码库，并没有同步到远程服务器上。要同步到远程服务器上，我们需要在客户端终端进入gitolite-admin目录，输入如下命令：

```
$ git push
Counting objects: 7, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 1.04 KiB, done.
Total 5 (delta 0), reused 0 (delta 0)
To git@192.168.1.108:gitolite-admin
76eabfb..6e4f3ad master -> master
```

git push命令用于推送本地代码库到远程服务器代码库。如果出现上面的运行结果，就修改完成了。如果我们打开服务器的repositories目录，此时会多一个ios_repo.git代码库，这就是我们新添加的代码库了。我们可以在zhao用户电脑上输入下面的指令测试一下：

```
$ git clone git@192.168.1.108:ios_repo
```

正确执行完成后，则克隆一个ios_repo代码库到zhao的本地电脑了。

提示 如果命令在执行过程中出现Agent admitted failure to sign using the key错误，需要在本机上执行\$ ssh-add ~/.ssh/id_rsa命令。

18.2.3 Git 常用命令

无论我们的项目是否需要与他人协同开发,都会用到Git的一些常用命令。我们在上一节中就用到了git add、git commit和git push命令。本节中,我们将介绍一些常用的Git命令,包括git help、git log、git init、git add、git rm、git commit和git status等命令。最后,还介绍了Git图形界面辅助工具gitk。

1. git help

第一个要掌握的是git help,通过它可以自己查找命令的帮助信息,此时在终端中执行如下命令即可:

```
$ git help <命令>
```

其中help后面是要查询的命令。

2. git log

该命令可以查看Git的日志信息。在终端中执行git log命令:

```
$ git log
```

执行结果为:

```
commit 2f027fbad790fa3e61ec9965a18415203f5e9683
Author: tonyguan <si92@sina.com>
Date:   Wed Oct 31 12:57:13 2012 +0800

    a

commit ac29dd648c7e78bfed5682742855683b5242c27f
Author: git on zhao-VirtualBox <git@zhao-VirtualBox>
Date:   Wed Oct 31 12:53:27 2012 +0800

    start
```

3. git init

该命令可以创建一个新的代码库,或者是初始化一个已存在的代码库。例如,我想在本地创建一个myrepo代码库,可以先使用mkdir创建这个目录,然后再执行git init。在终端中执行如下命令:

```
$ mkdir myrepo
$ cd myrepo
$ git init
Initialized empty Git repository in /Users/tonyguan/myrepo/.git/
```

使用mkdir创建myrepo目录(它只是一个普通的目录,并不是代码库)后,git init会在myrepo目录下生成一个隐藏的.git目录。

4. git add

该命令用来更新索引,记录下哪些文件有修改,或者添加了哪些文件。该命令并没有更新代码库,只有在提交的时候才将这些变化更新到代码库中。在终端中执行如下命令:

```
$ git add .
```

可以将当前工作目录和子目录下所有新添加和修改的文件添加到索引中。如果只想将某个文件添加到索引中,可以使用如下命令:

```
$ git add filename 或 $ git add *.txt
```

这里可以指定文件名,也可以使用通配符指定文件名。

5. git rm

该命令用于删除索引或代码库中的文件,然后通过提交命令将变化更新到代码库中。在终端中执行命令:

```
$ git rm filename 或 $ git rm *.txt
```


6. git commit

该命令用于更新缓存中的索引，但未被保存到代码库中的内容。在终端中执行命令：

```
$ git commit -m 'tony commit'
```

其中-m设定提交注释信息。

7. git status

该命令可以显示当前git的状态，包括哪些文件修改、删除和添加了，但是没有提交的信息。在终端中执行命令：

```
$ git status
```

会显示类似如下的内容：

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#   (commit or discard the untracked or modified content in submodules)
#
#   modified:   .DS_Store
#   modified:   HelloWorld (modified content, untracked content)
#
no changes added to commit (use "git add" and/or "git commit -a")
```

8. gitk

gitk并不是一个命令，而是一个图形工具，用来辅助管理Git，在终端中直接输入gitk就可以启动。图18-3所示的是gitk图形界面。

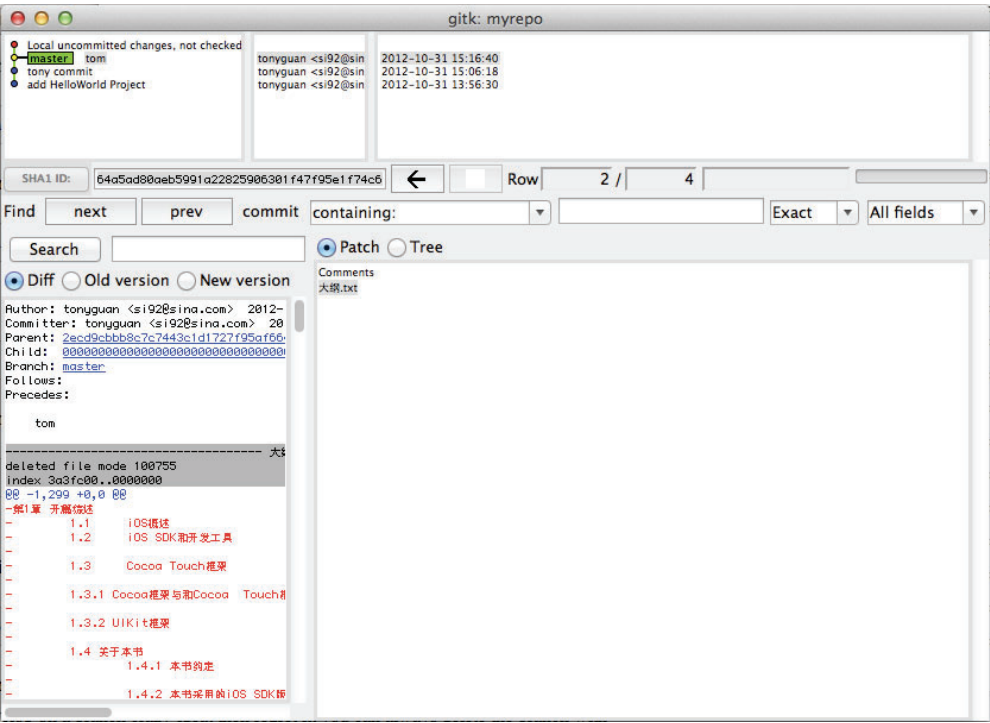


图18-3 gitk图形界面

在gitk工具中，可以查看日志、提交信息、注释、分支等信息，使用起来非常方便。

18.2.4 Git 分支

上一节介绍了Git中的常用命令，下面我们介绍Git分支。分支在版本控制中占有非常重要的地位。我们建立分支可能出于多种原因，但是无外乎基于验证一些想法、版本发布、bug修改等目的。

我们现在拿第2章的HelloWorld案例了解一下分支的用法，其中HelloWorld工程中视图控制器ViewController.m的代码如下：

```
//
//ViewController.m
//HelloWorld
//
//Created by tonyguan on 12-8-22.
//Copyright (c) 2012年 tonyguan. All rights reserved.
//

#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

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

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

@end
```

提示 我在本地创建了一个名为myrepo的代码库，把HelloWorld工程复制到myrepo代码库中，进入到myrepo下的HelloWorld目录添加并提交该工程。

1. 创建分支

创建分支时，使用命令`git branch <分支名>`。如果我们要创建一个testing分支，可以在终端中执行如下命令：

```
$ git branch testing
```

创建完成后，我们需要查看一下分支情况，此时可以在终端中执行如下命令：

```
$ git branch
* master
testing
```

其中*号的分支是当前分支，master是git默认创建的分支，testing是我们刚刚创建的分支。

2. 切换分支

在我们编辑工程文件时，需要在不同的分支间切换，此时可以在终端中执行如下命令：

```
$ git checkout testing
Switched to branch 'testing'
```

首先，我们要在master分支中修改ViewController.m文件，具体如下：

```
//
//ViewController.m
//HelloWorld
//
//Created by tonyguan on 12-8-22.
//Copyright (c) 2012年 tonyguan. All rights reserved.
//

#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    NSLog(@"Hello World.");
}

- (void)viewWillAppear
{
    NSLog(@"viewWillAppear call");
}

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

@end
```

在上述代码中，我们在viewDidLoad方法中添加输出Hello World.，并添加viewWillAppear方法。然后在终端中执行命令提交代码：

```
$ git add .
$ git commit -m 'master branch commit'
[master b62ea54] master branch commit
2 files changed, 7 insertions(+), 2 deletions(-)
rewrite HelloWorld.xcodeproj/project.xcworkspace/xcuserdata/
    tonyguan.xcuserdatad/UserInterfaceState.xcuserstate (83%)
```

切换到testing分支中，修改ViewController.m文件，具体如下：

```
//
//ViewController.m
//HelloWorld
//
//Created by tonyguan on 12-8-22.
//Copyright (c) 2012年 tonyguan. All rights reserved.
//

#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

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

```

        NSLog(@"世界你好.");
    }

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

@end

```

与master分支不同的是，在testing分支中，我们在viewDidLoad方法中添加输出“世界你好.”，而且也没有添加viewWillAppear方法。可见，master分支和testing分支在输出“世界你好.”还是输出"Hello World."是存在冲突的。然后在终端中执行命令提交代码：

```

$ git add .
$ git commit -m 'testing branch commit'
[testing c27a331] testing branch commit
2 files changed, 2 insertions(+), 2 deletions(-)
rewrite HelloWorld.xcodeproj/project.xcworkspace/xcuserdata/
    tonyguan.xcuserdatad/UserInterfaceState.xcuserstate (83%)

```

3. 合并分支

如果我们把testing分支合并到master分支，需要先切换回master分支，接着在终端中执行如下命令切换回master分支：

```

$ git checkout master
Switched to branch 'master'

```

合并分支可以使用git merge命令。然后在终端中执行如下合并命令：

```

$ git merge testing
error: Your local changes to the following files would be overwritten by merge:HelloWorld.
    xcodeproj/project.xcworkspace/xcuserdata/tonyguan.xcuserdatad/UserInterface
        State.xcuserstate
Please, commit your changes or stash them before you can merge.
Aborting

```

如果在提交和合并之前有一些修改和变化，会报出错误，此时我们需要重新提交或放弃修改这个文件。在终端中执行如下命令可以放弃修改该文件：

```

$ git checkout HelloWorld.xcodeproj

```

这里使用了git checkout命令放弃修改，在前面切换分支时我们也使用了该命令。

然后再使用git merge命令合并，会出现如下问题：

```

$ git merge testing
warning: Cannot merge binary files: HelloWorld.xcodeproj/project.xcworkspace/xcuserdata/
    tonyguan.xcuserdatad/UserInterfaceState.xcuserstate (HEAD vs. testing)

Auto-merging HelloWorld/ViewController.m
CONFLICT (content): Merge conflict in HelloWorld/ViewController.m
Auto-merging HelloWorld.xcodeproj/project.xcworkspace/xcuserdata/tonyguan.xcuserdatad/
    UserInterfaceState.xcuserstate
CONFLICT (content): Merge conflict in HelloWorld.xcodeproj/project.xcworkspace/xcuserdata/
    tonyguan.xcuserdatad/UserInterfaceState.xcuserstate
Automatic merge failed; fix conflicts and then commit the result.

```

问题的本质是master分支和testing分支在输出时有冲突。我们这个时候再看看程序代码ViewController.m：

```

//
//ViewController.m
//HelloWorld
//
//Created by tonyguan on 12-8-22.
//Copyright (c) 2012年 tonyguan. All rights reserved.

```

```
//

#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    <<<<<<< HEAD
    NSLog(@"Hello World.");
    =====
    NSLog(@"世界你好.");
    >>>>>>> testing
}

- (void)viewWillAppear
{
    NSLog(@"viewWillAppear call");
}

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

@end
```

其中viewWillAppear方法成功合并，而viewDidLoad方法中<<<<<<< HEAD ... >>>>>>>表示其中的内容有冲突。由于Git无法判断谁对谁错，所以需要人工解决。解决完成再次添加并提交，在终端中执行如下命令：

```
$ git commit -a -m 'testing branch merge commit'
[master 60eac40] testing branch merge commit
```

其中git commit -a -m相当于git add .和git commit -m的执行效果。

4. 删除分支

在分支合并完成且不再使用的情况下，可以使用git branch -d删除分支。要删除testing分支，可以在终端中执行如下命令：

```
$ git branch -d testing
Deleted branch testing (was c27a331).
```

也可以使用-D参数代替-d，其中-D表示强制删除分支。

如果我们想看看分支的情况，更好的方式是使用gitk辅助工具，我们的这次合并结果如图18-4所示。

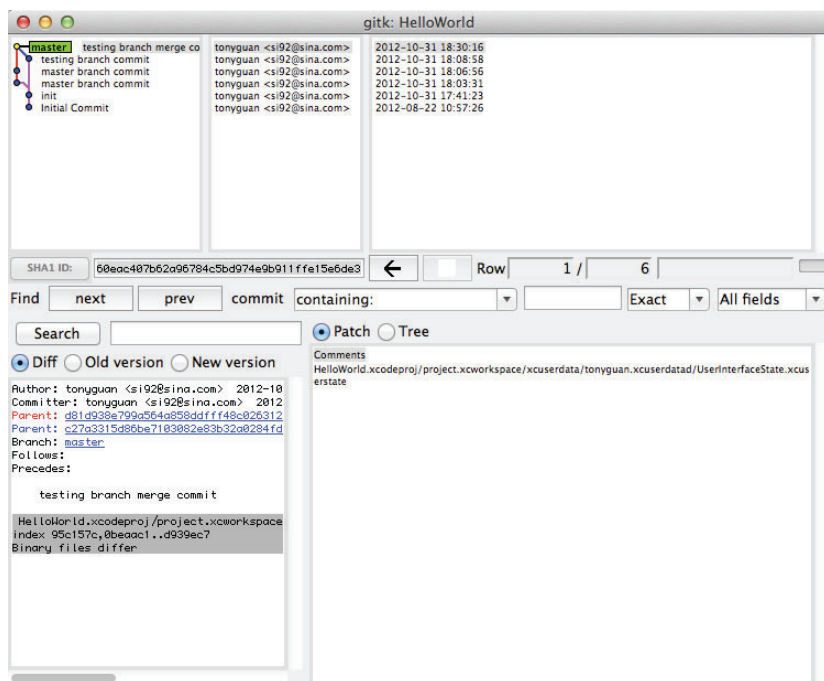


图18-4 使用gitk查看分支

18.2.5 Git 协同开发

如果我们的项目需要一个团队协同开发，就需要搭建Git服务器。协同开发涉及的常用命令有git clone、git fetch、git pull、git push和git remote add等。

下面我们通过解决实际工作中的一些问题来掌握这些命令的实现。如图18-5所示，开发者1和开发者2都共享一个服务器代码库myrepo，服务器环境都已经搭建完成。

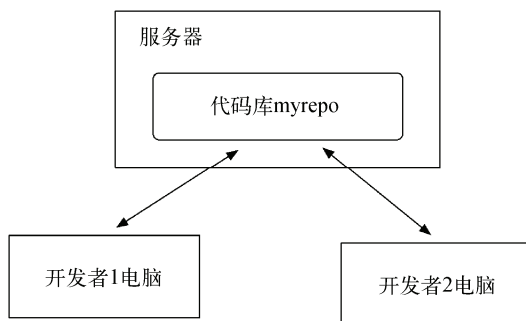


图18-5 Git协同开发

他们在工作中遇到的几个问题如下所示。

- ❑ 问题1。开发者1创建myrepo代码库和HelloWorld工程，然后提交到远程的服务器代码库。
- ❑ 问题2。开发者2克隆服务器代码库myrepo到本地代码库，对HelloWorld工程进行修改，并重新提交给服务器代码库。
- ❑ 问题3。当开发者2重新提交给服务器代码库时，开发者1如何获得本地代码库？

1. 问题1

开发者1的电脑上没有myrepo代码库，需要自己创建，而不是从服务器上克隆过来。首先，创建myrepo目录作为代码库，再将HelloWorld工程复制到myrepo代码库下面，它们的目录结构内容如下：

```
myrepo
├── HelloWorld
│   ├── HelloWorld
│   │   ├── AppDelegate.h
│   │   ├── AppDelegate.m
│   │   ├── HelloWorld-Info.plist
│   │   ├── HelloWorld-Prefix.pch
│   │   ├── ViewController.h
│   │   ├── ViewController.m
│   │   ├── en.lproj
│   │   │   ├── InfoPlist.strings
│   │   │   └── ViewController.xib
│   └── main.m
└── HelloWorld.xcodeproj
```

进入myrepo目录，会在终端中执行如下命令来初始化myrepo代码库：

```
$ git init
```

如果修改了代码，需要提交内容到本地代码库，此时可以在终端执行如下命令：

```
$ git add .
$ git commit -m 'tony commit'
[master 3609824] tony commit
13 files changed, 658 insertions(+)
create mode 100644 HelloWorld/.DS_Store
create mode 100644 HelloWorld/HelloWorld.xcodeproj/project.pbxproj
create mode 100644 HelloWorld/HelloWorld.xcodeproj/project.xcworkspace/
contents.xcworkspacedata
create mode 100644 HelloWorld/HelloWorld.xcodeproj/project.xcworkspace/
xcuserdata/tonyguan.xcuserdatad/UserInterfaceState.xcuserstate
create mode 100644 HelloWorld/HelloWorld.xcodeproj/xcuserdata/
tonyguan.xcuserdatad/xcodebugger/Breakpoints.xcbkptlist
create mode 100644 HelloWorld/HelloWorld.xcodeproj/xcuserdata/
tonyguan.xcuserdatad/xcschemes/HelloWorld.xcscheme
create mode 100644 HelloWorld/HelloWorld.xcodeproj/xcuserdata/
tonyguan.xcuserdatad/xcschemes/xcschememanagement.plist
create mode 100644 HelloWorld/HelloWorld/AppDelegate.h
create mode 100644 HelloWorld/HelloWorld/HelloWorld-Info.plist
create mode 100644 HelloWorld/HelloWorld/HelloWorld-Prefix.pch
create mode 100644 HelloWorld/HelloWorld/ViewController.h
create mode 100644 HelloWorld/HelloWorld/en.lproj/InfoPlist.strings
create mode 100644 HelloWorld/HelloWorld/en.lproj/ViewController.xib
```

然后将本地代码库推送到远程服务器，此时可以使用命令git remote add和git push，具体如下所示：

```
$ git remote add HW git@192.168.1.107:myrepo
$ git push HW master
Counting objects: 29, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (22/22), done.
Writing objects: 100% (26/26), 21.36 KiB, done.
Total 26 (delta 0), reused 0 (delta 0)
To git@192.168.1.107:myrepo
43184b3..3609824 master -> master
```

其中git remote add HW git@192.168.1.107:myrepo是为远程代码库一个名字HW，以便与远程代码库交互。git push HW master就是向刚才定义的远程代码库HW的master分支推送数据。

2. 问题2

开发者2的电脑上没有myrepo代码库，他不需要自己创建，而是从服务器上克隆过来，此时可以在终端中执

行如下命令：

```
$ git clone git@192.168.1.107:myrepo
```

然后他也对HelloWorld做了一些修改，那么如何推送他的数据到服务器代码库呢？事实上，这个过程与开发者1刚刚的推送方式是一样的，这里就不再介绍了。

3. 问题3

开发者1再次被告知他们程序有新的版本，他需要从服务器代码库中获取新的程序。使用git fetch命令，可以从服务器代码库获取数据，相关命令如下：

```
$ git fetch HW
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 3), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From 192.168.1.107:myrepo
 3609824..7680cce master    -> HW/master
```

这时打开修改的文件，发现没有变化，这是因为还需要使用git merge命令合并HW/master到本地master分支，相关代码如下：

```
$ git merge HW/master
Updating 3609824..7680cce
Fast-forward
 HelloWorld/HelloWorld/ViewController.m |    1 -
 1 file changed, 1 deletion(-)
```

再看看修改的文件是否发生了变化。合并过程中也可能发生冲突，需要人为解决这些冲突再合并，这可以通过Git提供的更加简便的命令git pull来实现。git pull命令是git fetch和git merge命令的一个组合，相关代码如下：

```
$ git pull HW master
From 192.168.1.107:myrepo
* branch          master      -> FETCH_HEAD
Updating 7680cce..26b89ea Fast-forward  .DS_Store          | Bin 0 -> 6148 bytes
.../UserInterfaceState.xcuserstate      | Bin 25632 -> 26270 bytes
 HelloWorld/HelloWorld/ViewController.m  |    2 +-
3 files changed, 1 insertion(+), 1 deletion(-)
create mode 100644 .DS_Store
```

其中HW是远程代码库名，master是要合并的本地分支。

18.2.6 Xcode 中 Git 的配置与使用

我们在前面介绍过很多Git命令都是在命令行下运行的。在命令行下管理Git有很多优点，这就不用多说了，但最大的缺点是要用户记住这些命令。Git图形界面还是很受用户欢迎的。作为集成开发环境工具Xcode，它也提供了一定的Git图形界面功能。但是要想在Xcode中使用Git管理工程代码，还要进行一些配置。

如果我们使用Xcode 4创建的一个iOS工程，在终端的命令行中提交代码时，可能会出现下面的部分信息：

```
create mode 100644 HelloWorld/HelloWorld.xcodeproj/project.xcworkspace/
contents.xcworkspacedata
create mode 100644 HelloWorld/HelloWorld.xcodeproj/project.xcworkspace/
xcuserdata/tonyguan.xcuserdatad/UserInterfaceState.xcuserstate
create mode 100644 HelloWorld/HelloWorld.xcodeproj/xcuserdata/
tonyguan.xcuserdatad/xcschemes/HelloWorld.xcscheme
create mode 100644 HelloWorld/HelloWorld.xcodeproj/xcuserdata/
tonyguan.xcuserdatad/xcschemes/xcschememanagement.plist
rewrite HelloWorld.xcodeproj/project.xcworkspace/xcuserdata/
tonyguan.xcuserdatad/UserInterfaceState.xcuserstate (83%)
```

事实上，能够列入代码版本控制的文件是有规定的，不能是编写的二进制文件、临时文件和用户特有的文件

等。下面是在Xcode 4中创建的HelloWorld工程的目录结果：

```

HelloWorld
├── HelloWorld
│   ├── AppDelegate.h
│   ├── AppDelegate.m
│   ├── HelloWorld-Info.plist
│   ├── HelloWorld-Prefix.pch
│   ├── ViewController.h
│   ├── ViewController.m
│   ├── en.lproj
│   │   ├── InfoPlist.strings
│   │   └── ViewController.xib
│   └── main.m
├── HelloWorld.xcodeproj
│   ├── project.pbxproj
│   ├── project.xcworkspace
│   │   ├── contents.xcworkspacedata
│   │   └── xcuserdata
│   │       ├── tonyguan.xcuserdatad
│   │       └── UserInterfaceState.xcuserstate
│   └── xcuserdata
│       ├── tonyguan.xcuserdatad
│       │   ├── xcdebugger
│       │   │   └── Breakpoints.xcbkptlist
│       │   └── xcschemes
│       │       ├── HelloWorld.xcscheme
│       │       └── xcschememanagement.plist

```

其中HelloWorld.xcodeproj属于包文件，它内部的很多东西是不能提交的，包括project.xcworkspace和xcuserdata，它们是与用户有关的。在Git中，有一个.gitignore配置文件，其中可以设置被忽略的文件。下面是一个.gitignore配置文件的内容：

```

# Exclude the build directory
build/*

# Exclude temp nibs and swap files
*~.nib
*.swp

# Exclude OS X folder attributes
.DS_Store

# Exclude user-specific XCode 3 and 4 files
*.model
*.modelv3
*.mode2v3
*.perspective
*.perspectivev3
*.pbxuser
*.xcworkspace
xcuserdata

```

在这个文件中，#号表示注释，可以使用正则表达式。此外，这个文件还考虑到了Xcode 3和Xcode 4的差别。这个文件创建后，应该放在什么地方呢？如果只考虑忽略一个特定的工程，.gitignore文件应该放在代码库目录下面，目录结构如下所示：

```

<代码库目录>
├── HelloWorld
│   ├── HelloWorld
│   │   ├── AppDelegate.h
│   │   ├── AppDelegate.m
│   │   ├── Default-568h@2x.png
│   │   └── Default.png

```


第一行的HelloWorld是工程目录，也是代码库的根目录，第二行的HelloWorld目录是存放源程序的目录。而我们以前的目录结构与此不同，具体如下所示：

```

1 myrepo
2   HelloWorld
3     HelloWorld
4       ...
5       ViewController.m
6       main.m
7     HelloWorld.xcodeproj
8   .git

```

第一行myrepo是代码库的根目录，第二行的HelloWorld是工程目录。对于这样的结构，一个代码库可以放置多个工程，是一对多的关系。而Xcode生成的方式是代码库就是工程目录，它们是一对一的关系。

如果我们还是采用一对多的关系，就不用创建工程时选中Create local git repository for this project复选框了。这就需要将现有的HelloWorld工程复制到myrepo目录，此时在终端中执行如下命令即可：

```

$ cd ~/myrepo
$ git init
Initialized empty Git repository in  ~/.git/

```

初始化完成之后，再添加并提交HelloWorld工程，这可以通过在终端中执行如下命令来完成：

```

$ git add .
$ git commit -m 'tony init'
[master (root-commit) 98d7e4a] tony init
10 files changed, 643 insertions(+)
create mode 100644 HelloWorld/HelloWorld.xcodeproj/project.pbxproj
create mode 100644 HelloWorld/HelloWorld/AppDelegate.h
create mode 100644 HelloWorld/HelloWorld/AppDelegate.m
create mode 100644 HelloWorld/HelloWorld/HelloWorld-Info.plist
create mode 100644 HelloWorld/HelloWorld/HelloWorld-Prefix.pch
create mode 100644 HelloWorld/HelloWorld/ViewController.h
create mode 100644 HelloWorld/HelloWorld/ViewController.m
create mode 100644 HelloWorld/HelloWorld/en.lproj/InfoPlist.strings
create mode 100644 HelloWorld/HelloWorld/en.lproj/ViewController.xib
create mode 100644 HelloWorld/HelloWorld/main.m

```

然后就可以在Xcode中修改这个工程了。修改并保存文件后，我们会看到在导航面板中文件的后面有一个M图标，如图18-7所示，这说明文件修改了但没有提交。

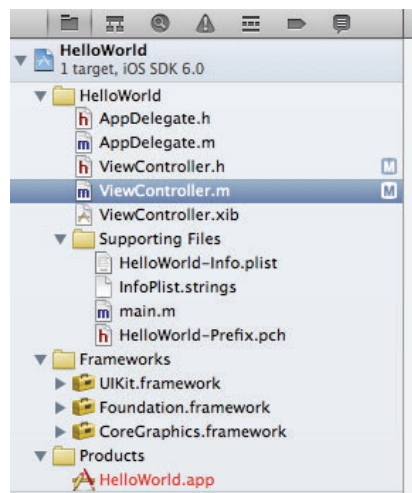


图18-7 文件修改了但没有提交的状态

如果只是想提交选中的文件，可以使用Source Control→Commit Selected Files...快捷菜单，其中Source Control菜单都是有关代码控制的。如果想提交全部的修改文件，可以使用File→Source Control→Commit...菜单项，此时弹出的对话框如图18-8所示。

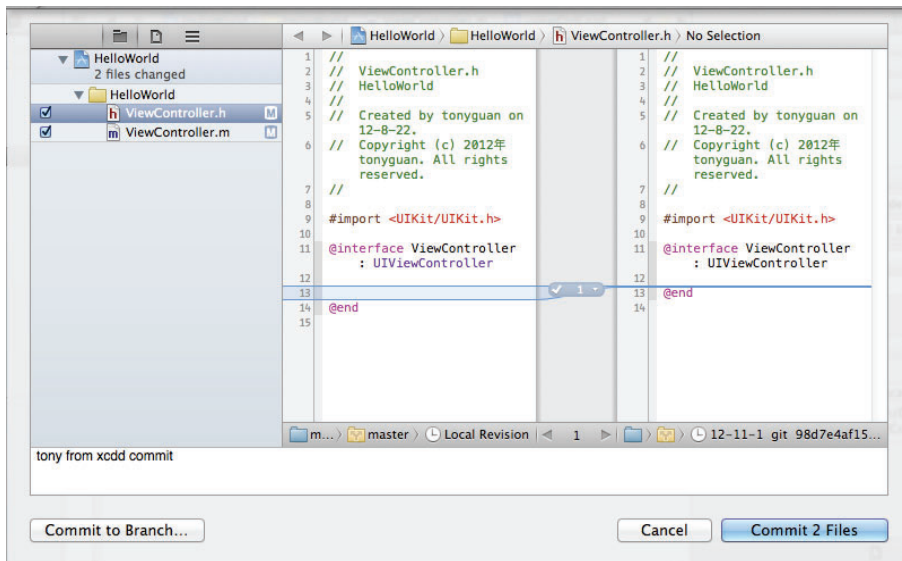


图18-8 提交文件

其中有两个代码窗口，左边是本地未提交版本，右边是代码库中的版本，这里可以比较看看修改了哪些内容。在下面输入框中添加注释，点击提交按钮就可以提交了。

2. 问题2

在Xcode中，可以通过File→Source Control→Push...菜单项将本地代码库推送给远程服务器代码库。但是如果是第一次访问，会出现如图18-9所示的对话框，表示没有可以推送的远程服务器代码库名。

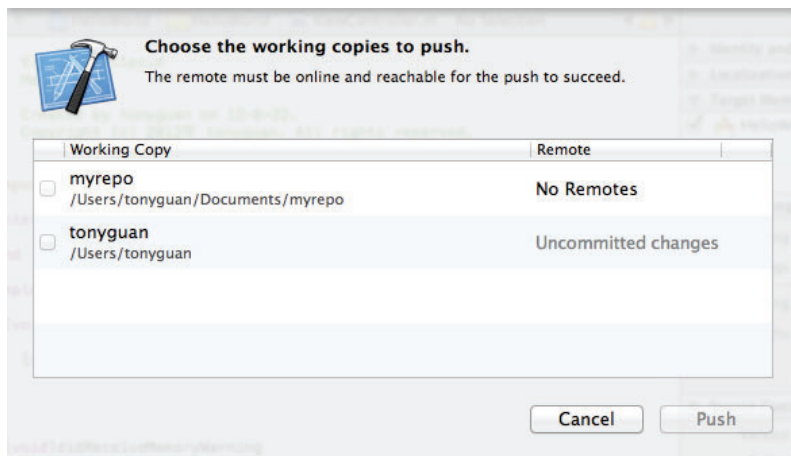


图18-9 推送远程服务器失败

此时我们需要建立远程服务器代码库名。在命令行中，我们是通过\$ git remote add hw git@192.168.1.108:myrepo创建的，其中hw就是这个名字。在Xcode中，可以通过Window→Organizer菜单项，在打开的界面

中选择Repositories→myrepo→Remotes，然后单击左下角的Add Remote按钮，此时会弹出Add a Remote对话框，再在Remote Name文本框中输入remote_repo，在Location文本框中输入git@192.168.1.108:myrepo，最后点击Create按钮创建这个名字，如图18-10所示。

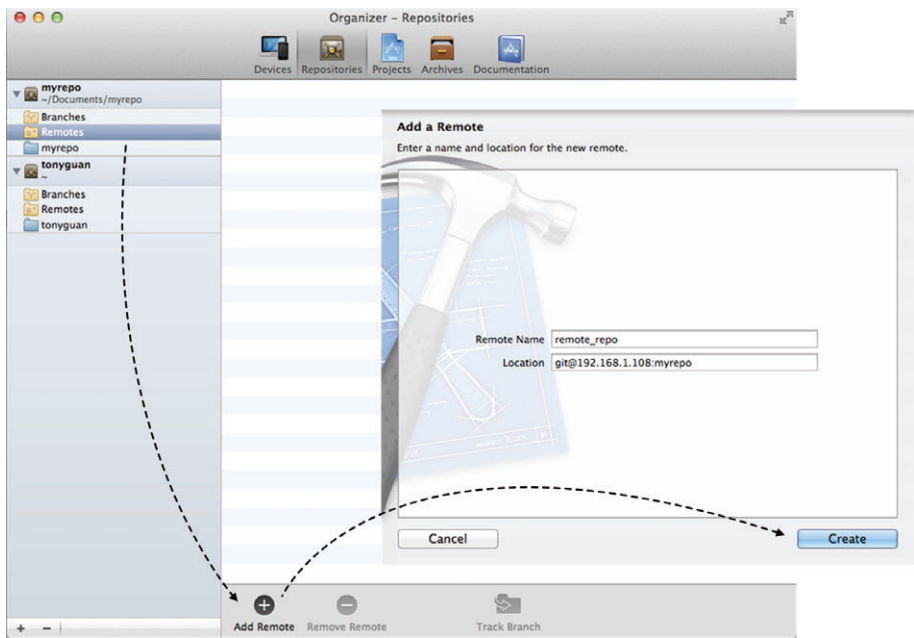


图18-10 创建远程服务器代码库名

创建完成后再重新推送，此时会弹出如图18-11所示的对话框，直接点击Push按钮推送即可。

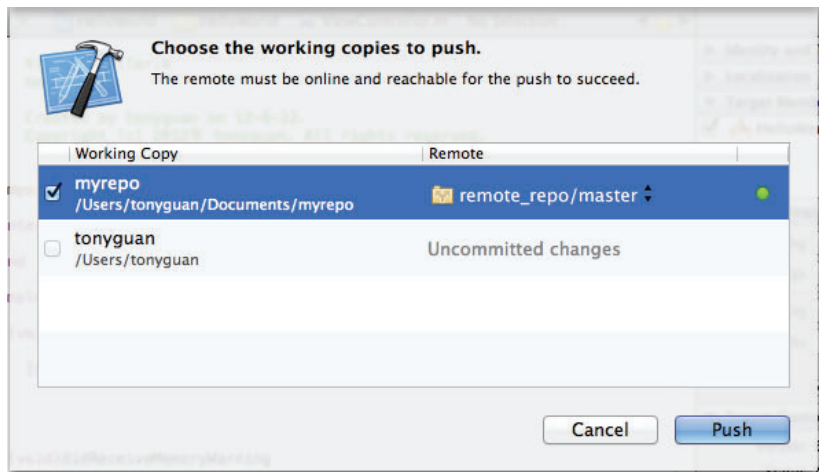


图18-11 推送远程服务器成功

3. 问题3

这个问题是从服务器代码库克隆到本地。首先,需要在Xcode中添加一个远程代码库。通过Window → Organizer菜单项进入Repositories界面，点击左下角的+按钮，从弹出列表中选择Add Repository项，此时将打开Add a Repository对话框，接着在Location文本框中输入git@192.168.1.108，在Type下拉列表中选择Git，Name项会自动添

加，如图18-12所示。如果Authentication required变为黄色小点，说明配置连接没有问题，然后点击Add按钮创建即可。

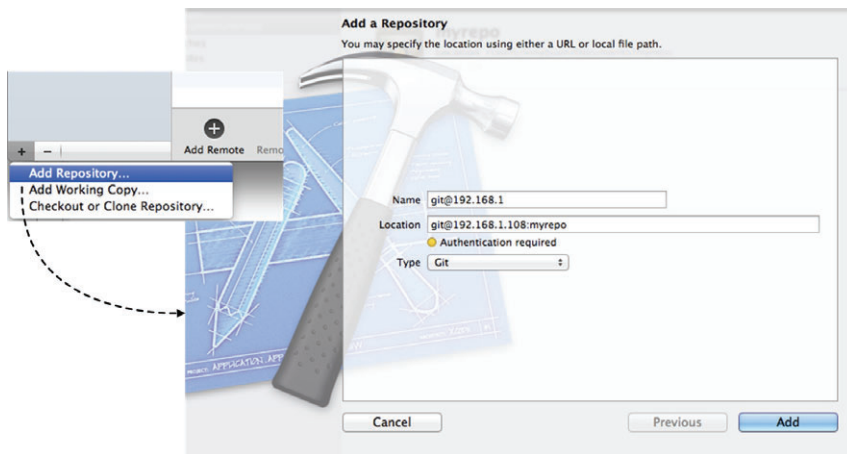


图18-12 添加远程代码库

如果创建成功，它就会出现在左边的代码库列表中，如图18-13所示，选择刚才创建的代码库，然后选择下面的Clone按钮，并选择本地的保存位置。

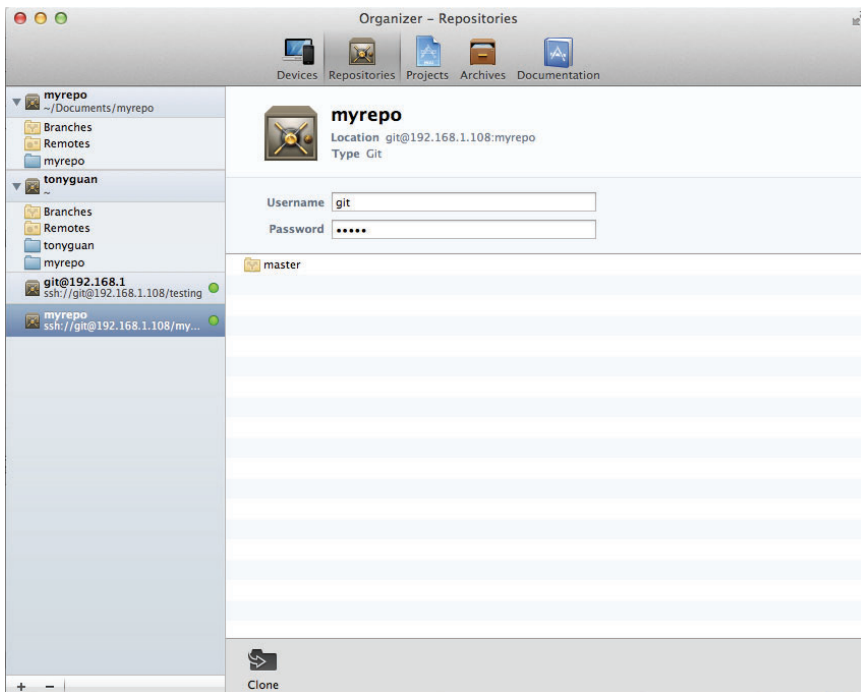


图18-13 克隆远程代码库

4. 问题4

如果服务器代码有新的版本，要将其库数据获取到本地，可以通过如下方法：选择File→Source Control→Pull...菜单项，此时弹出的对话框如图18-14所示，选择Choose按钮即可。

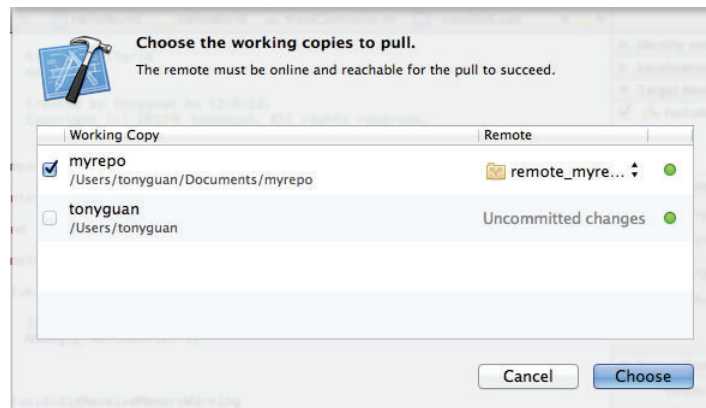


图18-14 获取远程代码库信息

如果这个过程中有冲突发生，会弹出如图18-15所示的对话框，从中可以看到它们的冲突点。

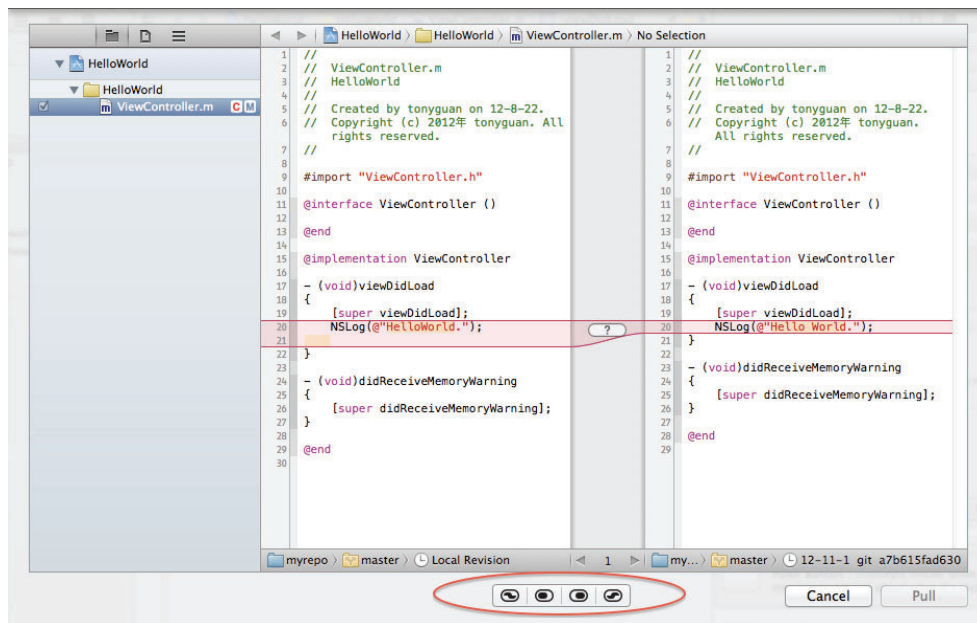


图18-15 代码冲突

在图18-15中，最下面的4个按钮用于合并和编辑冲突点。如果没有冲突，Pull就可以点击了，此时直接点击Pull按钮就可以了。

18.3 GitHub 代码托管服务

GitHub (<https://github.com>) 是全球最大的编程社区及代码托管网站，它可以提供基于Git版本控制系统的代码托管服务。GitHub同时提供商业账户和免费账户：免费账户不能创建私有项目；每个公有项目不能超过300MB的存储空间，但是300MB的空间限制并非硬性限制，如果不存在滥用情况的话，可以申请扩增托管空间。如果我们的团队成员分散在不同的地方，使用GitHub代码托管服务是一个不错的选择。

18.3.1 创建和配置 GitHub 账号

只有用户注册账号了，GitHub 才能提供服务。进入 <https://github.com/plans> 网址（如图 18-16 所示），创建免费账号、收费账号和收费组织。

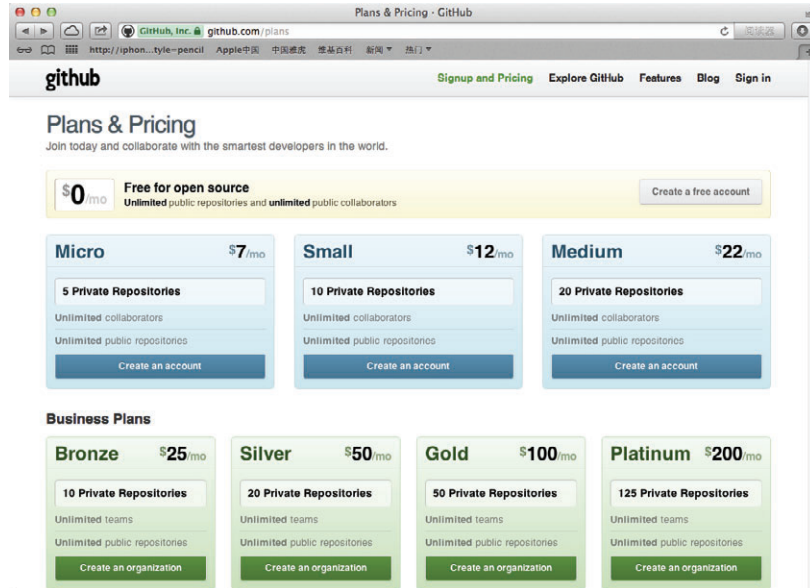


图18-16 创建账号

这里我们创建免费账号。点击 Create a free account 按钮，进入创建免费账号页面，输入账号、邮箱和密码，验证通过就可以创建了。进入网址 <https://github.com/login> 可以进行登录，这里可以使用刚才创建的账号测试一下。登录成功后的页面如图 18-17 所示。

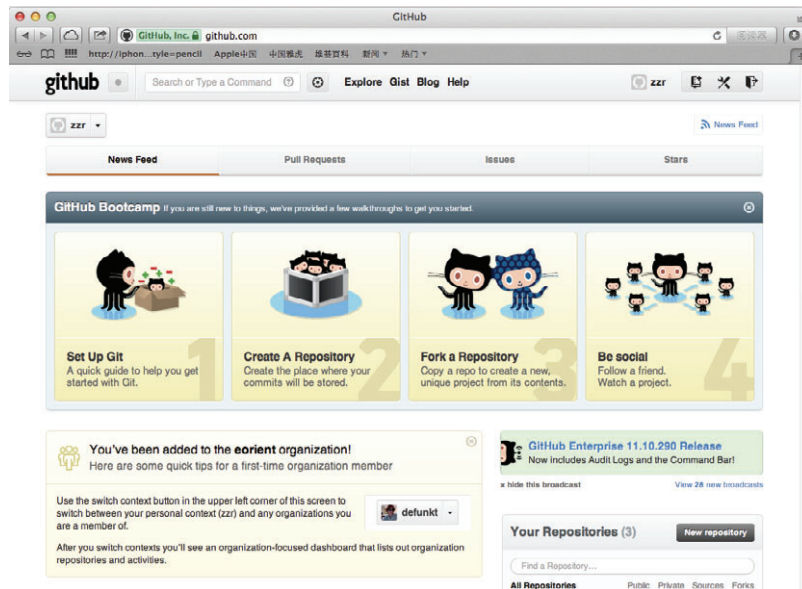


图18-17 登录成功页面

我们知道Git常用的协议是SSH协议,我们需要在本机上生成后,将公钥提供给GitHub网站。点击<你的账号>,然后点击Edit Your Profile按钮,从右边的列表框中选择SSH key项,再点击Add SSH key按钮,此时会进入如图18-18所示的页面。

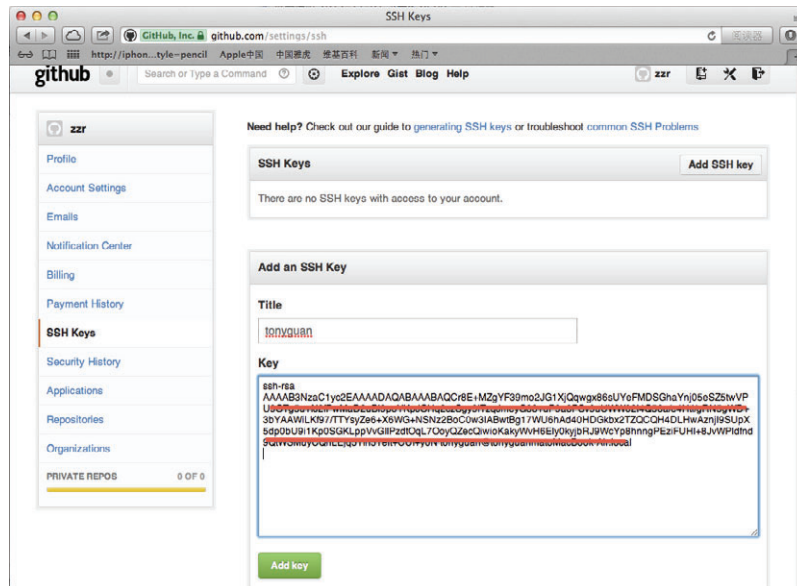


图18-18 添加SSH key页面

在Title中输入名字,这个可以随便命名。在Key中输入生成的公钥,用文本工具打开后将其复制到这里。然后点击Add key按钮提交内容,接着需要再次确认密码才能成功。图18-19为添加SSH key成功的页面。

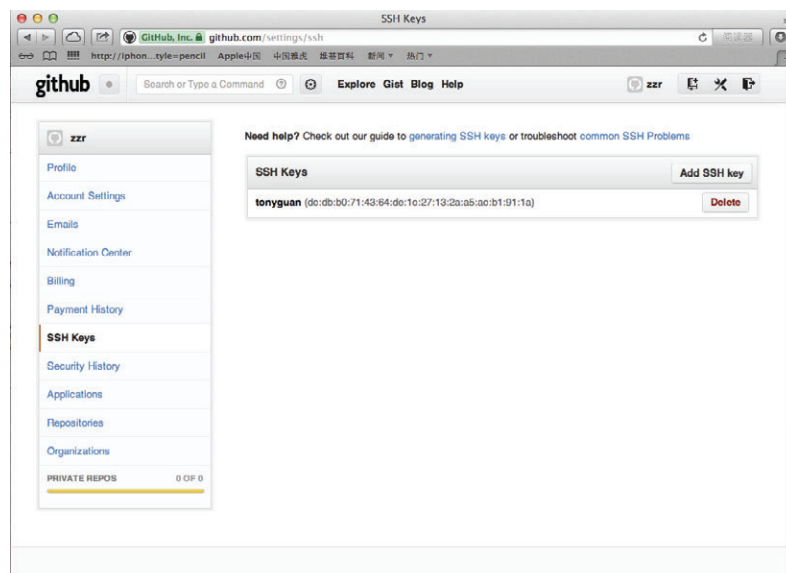


图18-19 添加SSH key成功的页面

这里可以提交多个key,用来管理同一用户在不同机器上的登录情况。

18.3.2 创建代码库

在GitHub中，代码库可以自己创建，也可以从别人那边派生（fork）过来。图18-20是代码库列表，其中zxr/Hello-Android是从名为tonyguan的用户那里派生过来的，其他的两个库是自己创建的。

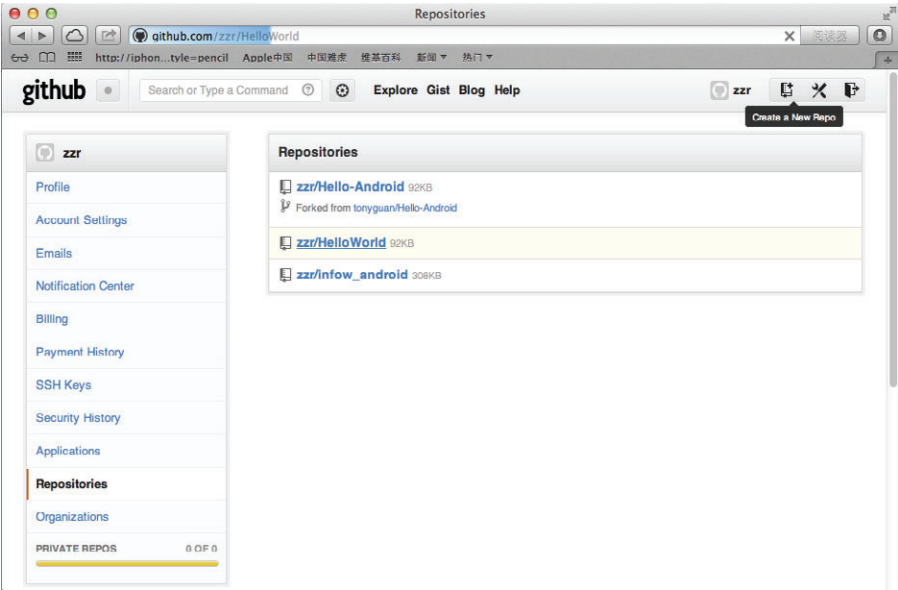


图18-20 代码库列表

如果要在GitHub中创建代码库，可以在登录成功页面中的右下角点击New repository按钮，如图18-21所示。

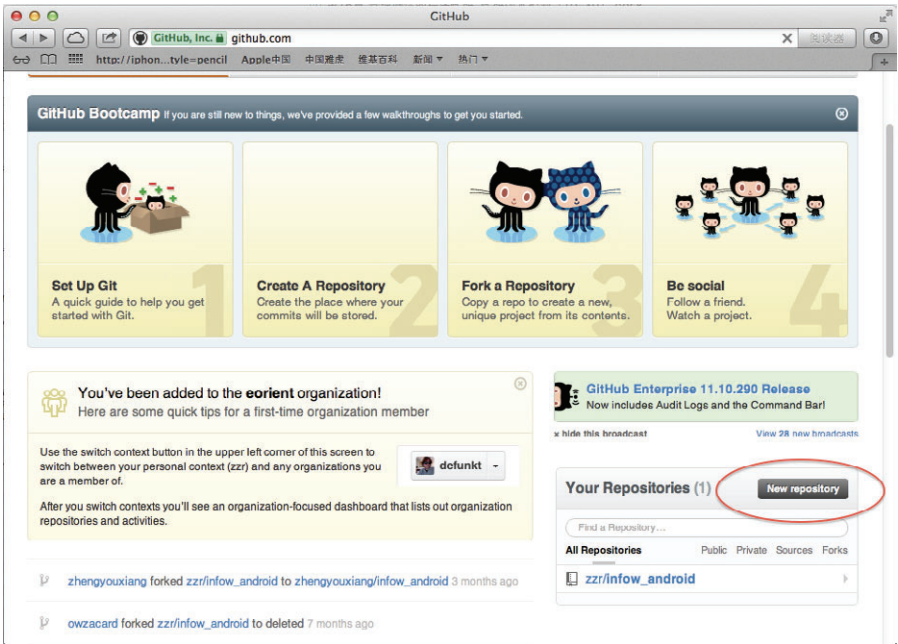


图18-21 创建代码库

此时进入如图18-22所示的创建代码库页面，在这里可以输入代码库的名字和描述信息。在这里，我们可以创建私有代码库或公有代码库。需要说明的是，只有付费账户才可以创建私有库。此外，我们还可以选择是否创建一个README文件来说明这个代码库。

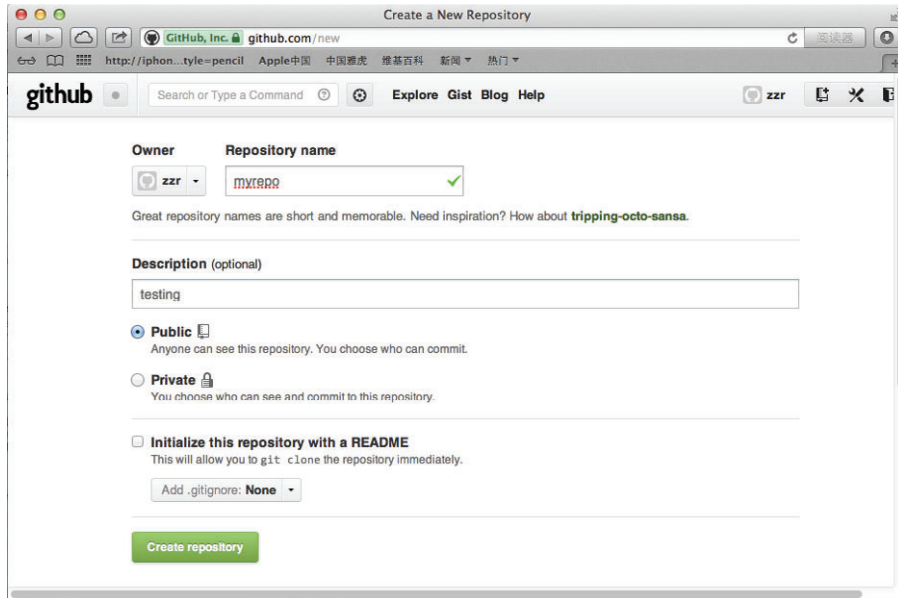


图18-22 创建代码库页面

此时点击Create repository按钮即可创建代码库，但此时代码库中还是空的，我们需要在本地电脑中推送一个Xcode项目到GitHub代码库。这里可以使用前面介绍的先创建myrepo目录，再使用git init命令初始化来推送，也可以从GitHub代码库克隆到本地。如果是克隆方式，则要在Xcode中添加代码库，具体请参考18.2.5 问题3解答部分。添加代码库的界面如图18-23所示。

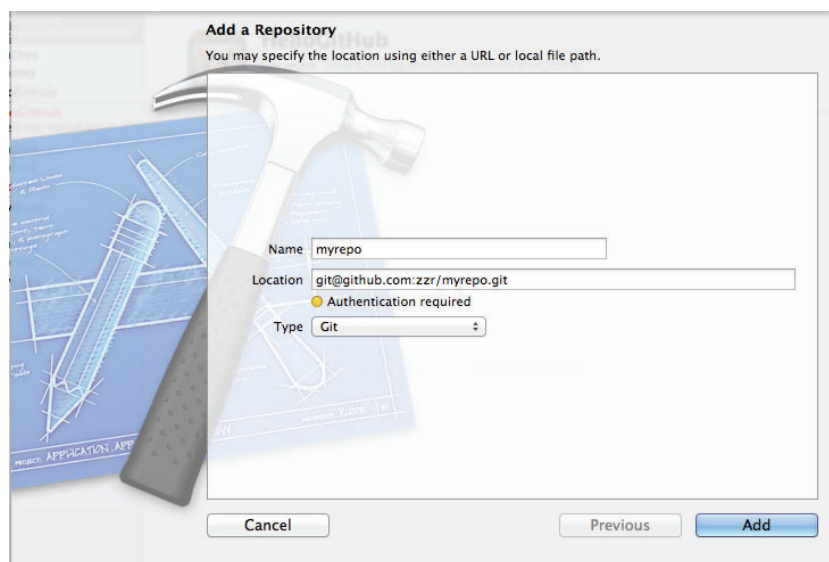


图18-23 添加GitHub代码库

选择刚刚创建的myrepo代码库克隆到本地。在myrepo目录中，创建新的工程HelloWorld或者复制现成的HelloWorld工程到myrepo目录，然后提交代码到本地库，再通过File→Source Control→Pull...菜单项推送到GitHub服务器，其中选择Remote为origin/master，如图18-24所示。

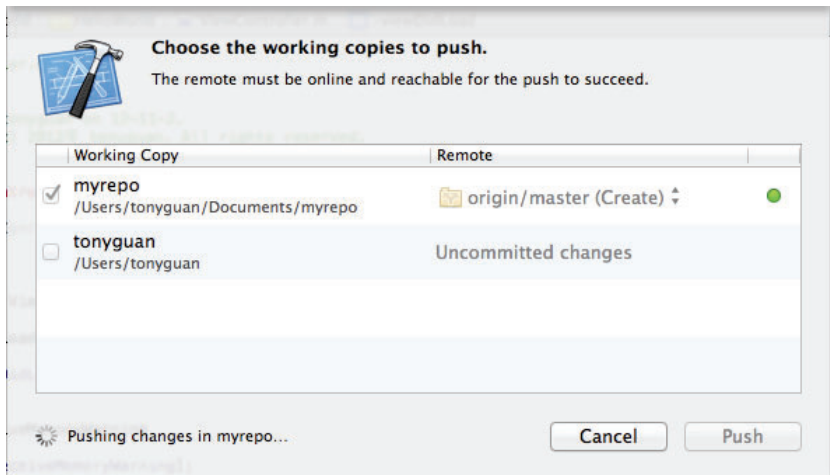


图18-24 选择GitHub代码库

若Push可以点击了，那就可以推送了，推送成功后可以GitHub中去查看。登录GitHub，进入zxr/myrepo代码库，选择Code标签，如图18-25所示，在这里看到我们推送的HelloWorld工程了，这说明推送成功了。

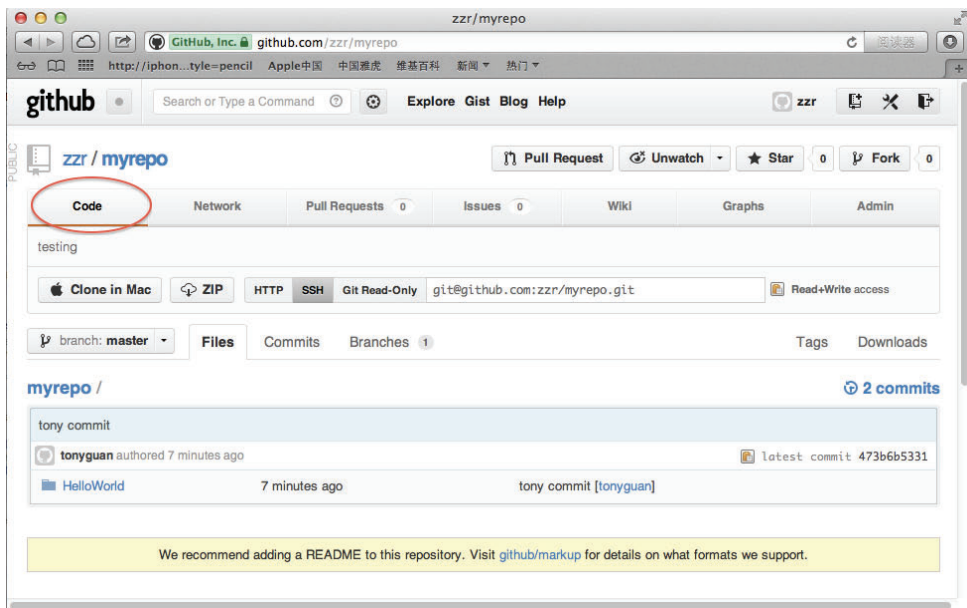


图18-25 查看myrepo代码库

有的时候我们需要删除代码库，此时可以点击如图18-25所示的Admin标签，进入如图18-26所示的代码库维护页面，接着将页面滚动到下面，此时会看到Delete this repository按钮，点击该按钮后再次确认即可完成该操作。这个操作破坏性比较大，大家操作时一定要谨慎。

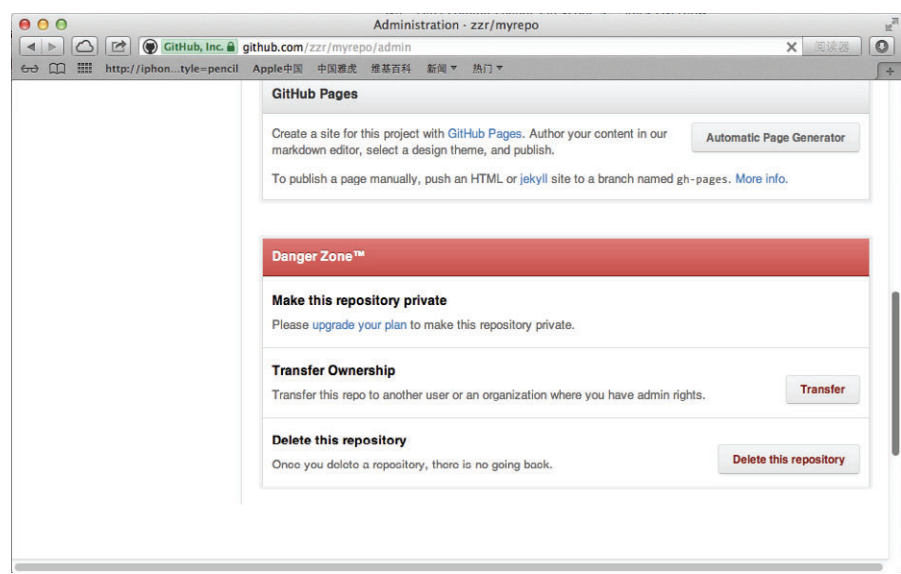


图18-26 删除代码库

18.3.3 派生代码库

获得代码库的最简单方式是从别人那里派生代码库。我们可以修改该代码库，然后提供回开发者。因此派生与Git中的分支很像，可以把它理解为代码库级别的分支。

如果我们想从tonyguan中派生ObserverPattern代码库，首先需要在GitHub中找到ObserverPattern代码库。GitHub搜索功能在网址https://github.com/search的页面中提供了，如图18-27所示。

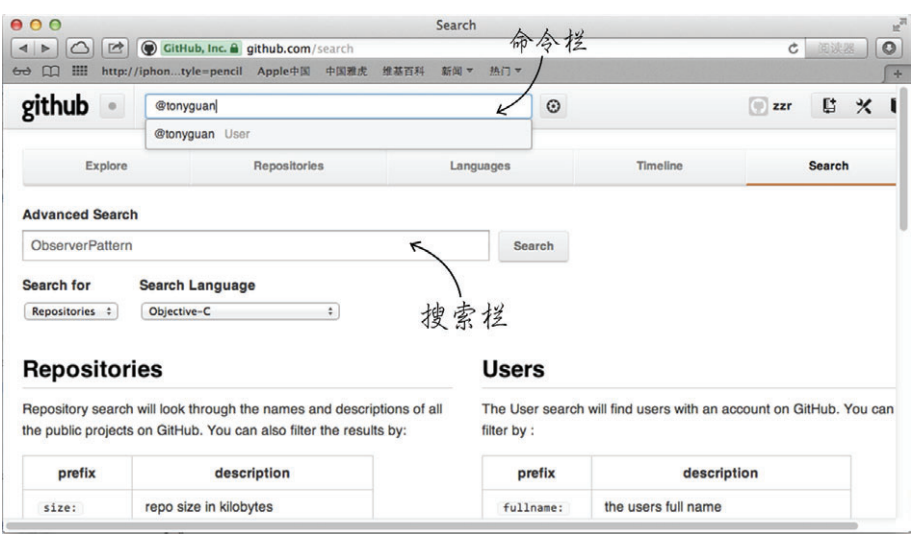


图18-27 GitHub中的执行命令和搜索页面

在这个网页中，在命令栏中输入命令并回车就可以执行命令了。在搜索栏中可以输入关键字。Search for可以选择搜索代码库、用户、代码或者全部信息，Search Language可以选择搜索特定语言。点击Search按钮即可进行搜索，得到的结果展示在下面，如图18-28所示。

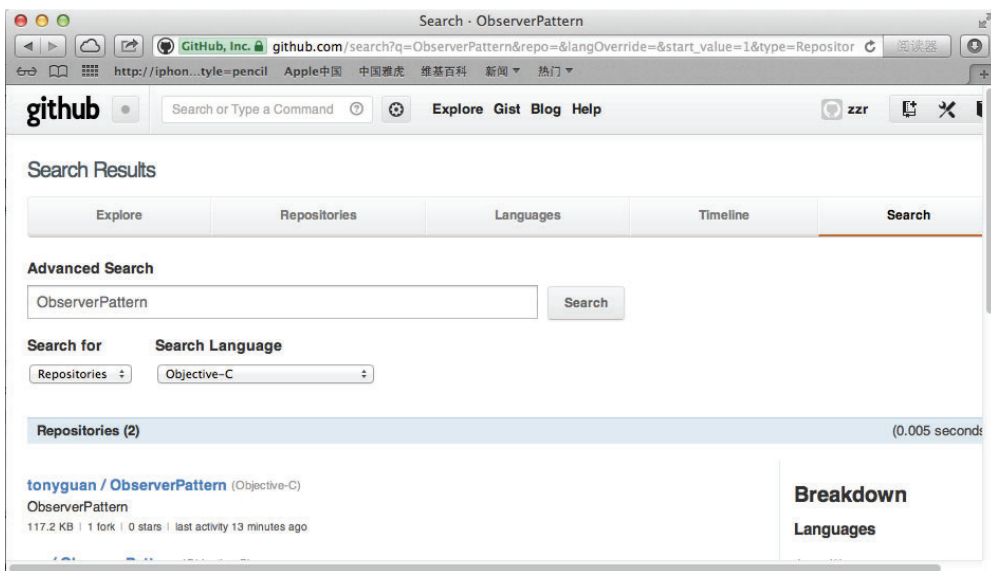


图18-28 搜索结果

点击tonyguan/ObserverPattern超链接，即可进入tonyguan账户下的ObserverPattern代码库，如图18-29所示。

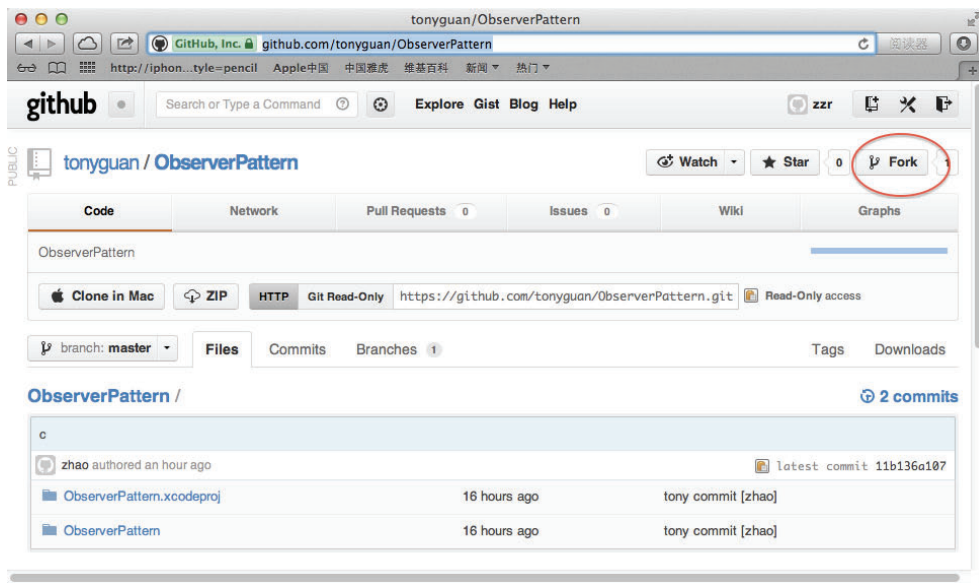


图18-29 tonyguan账户的ObserverPattern代码库

点击右上角的Fork按钮，此时会弹出一个确认对话框，确认之后就可以把ObserverPattern代码库派生到zzr账户下面了，如图18-30所示。

此时，zzr可以像使用自己创建的其他代码库一样使用这个库了。如果我们只是想参考别人的代码学习，这样派生过来后我们的工作就结束了。但如果我们不是旁观者而是参与者，需要对库中的项目修改，并推送到tonyguan/ObserverPattern代码库，这就涉及GitHub协同开发的问题了。

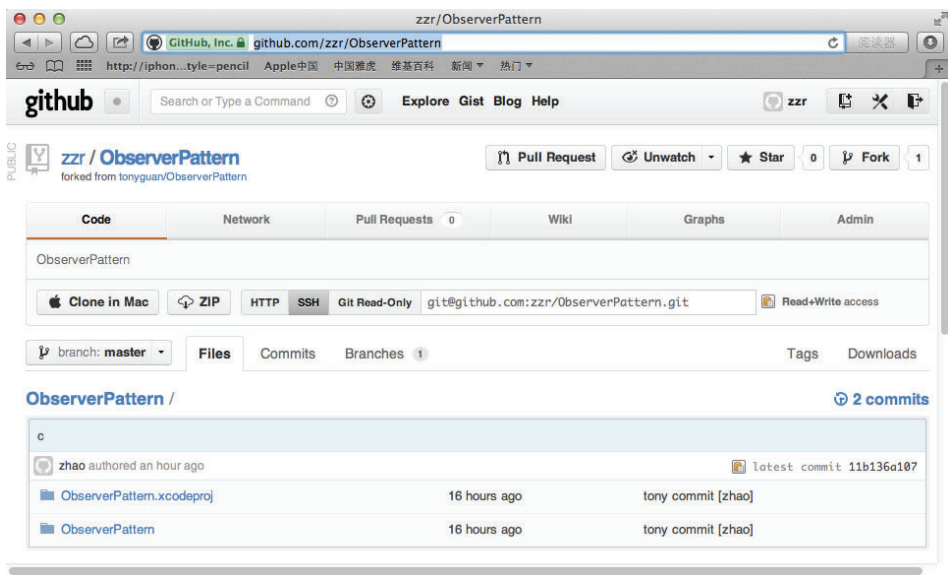


图18-30 zzr账户下的ObserverPattern代码库

18.3.4 使用 GitHub 协同开发

使用GitHub协同开发有两种模式：一种是比较传统的，其中代码库与开发者之间是一对多的关系模式；另一种是代码库与开发者之间是多对多的关系模式。

一对多的关系模式如图18-31所示，这里一个GitHub用户作为项目A代码库的管理员。这种模式很传统，适合于沟通密切的小团队开发，开发人员基本上不需要登录GitHub，只是使用GitHub作为一个远程服务器代码库存取代码。GitHub作为编程社区网站的优势没有体现出来，派生功能也没有用到。

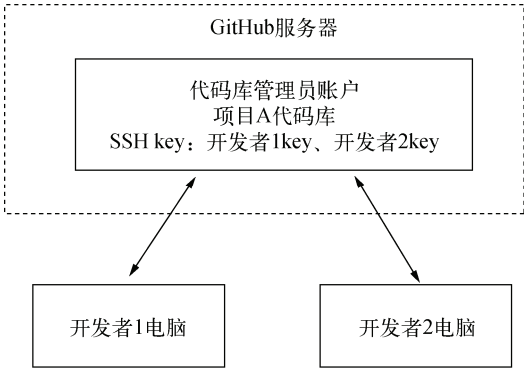


图18-31 一对多的关系模式

多对多的关系模式如图18-32所示，此时在GitHub服务器上存在多个A项目的代码库，但是只有一个是“主”的，由管理员账户维护。管理员也可能就是一个开发者，其他开发者有自己的账号，维护自己的A项目代码库，但是他们的项目代码库是从管理者那里派生过来的，修改完后需要推送回管理者的主代码库中。

这种模式很灵活，适用于复杂项目的大型团队，能够充分利用GitHub编程社区网站的功能，例如我们可以实现建立组织、互发邮件和通知等功能。这种模式主要使用GitHub的派生功能。

下面我们通过实例说明多对多的关系模式如何进行协同开发，具体流程如图18-33所示。

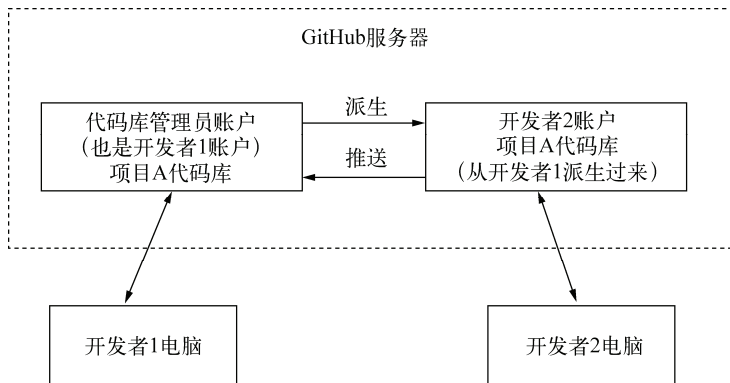


图18-32 多对多的关系模式

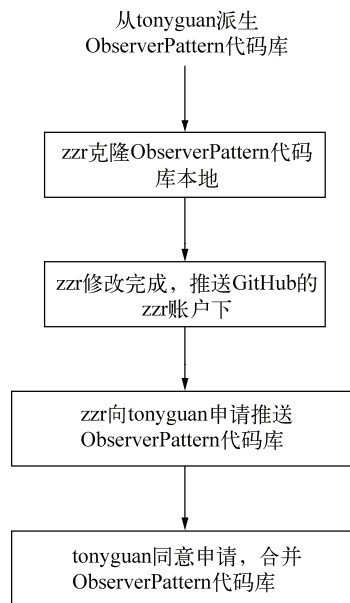


图18-33 多对多模式的工作流程图

在上一节中，zzr账户已经从tonyguan账户派生过来ObserverPattern代码库，所以这里从第二步开始介绍。要使zzr克隆ObserverPattern代码库到本地，可以在终端中执行如下命令：

```
$ git clone git@github.com:zzr/ObserverPattern.git
```

zzr修改了ObserverPattern中的ConcreteSubject.m文件，添加了注释“zzr修改”：

```
//zzr修改

#import "ConcreteSubject.h"

@implementation ConcreteSubject
.....
@end
```

然后将修改推送到自己的GitHub代码库，具体命令如下：

```
$git add .
$ git commit -m 'zzr commit'
[master 8873c7f] zzr commit
1 file changed, 0 insertions(+), 0 deletions(-)
$ git remote add r git@github.com:zzr/ObserverPattern.git
$ git push r master
Counting objects: 33, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (22/22), done.
Writing objects: 100% (25/25), 19.77 KiB, done.
Total 25 (delta 6), reused 0 (delta 0)
To git@github.com:zzr/ObserverPattern.git
11b136a..8873c7f master -> master
```

推送成功后，zzr需要在GitHub的ObserverPattern代码库中发送推送申请。进入ObserverPattern代码库，点击右上角Pull Request按钮，此时页面跳转到如图18-34所示的发送推送请求页面。

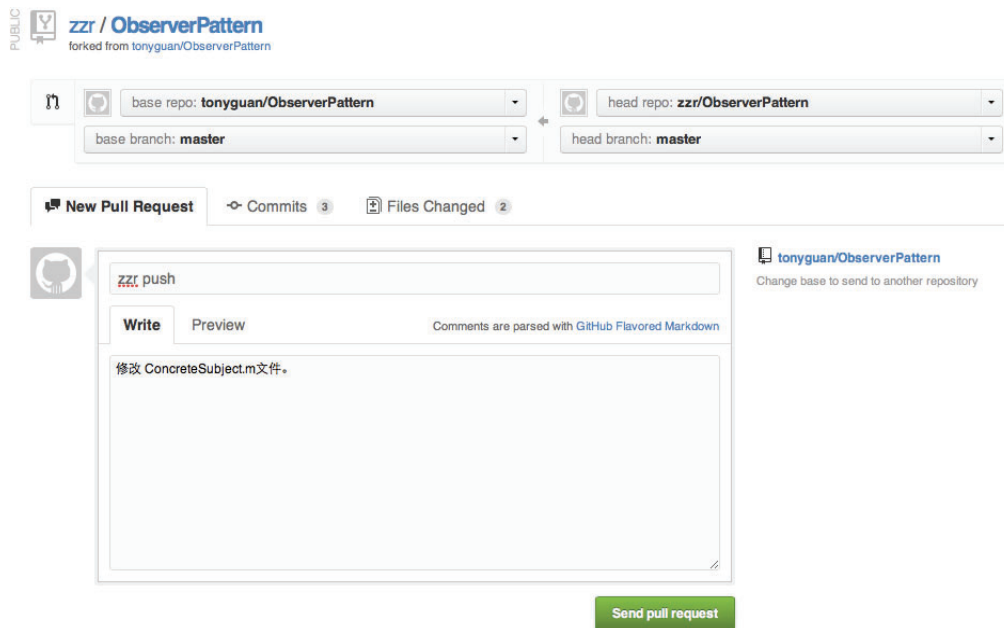


图18-34 发送推送请求页面

在该页面中，可以输入请求的标题和内容，这些信息很重要。填写完成后，点击Send pull request按钮发送请求，此时只需要等待结果。在问题没有解决之前，这个“讨论”是处于Open状态的，如图18-35所示。

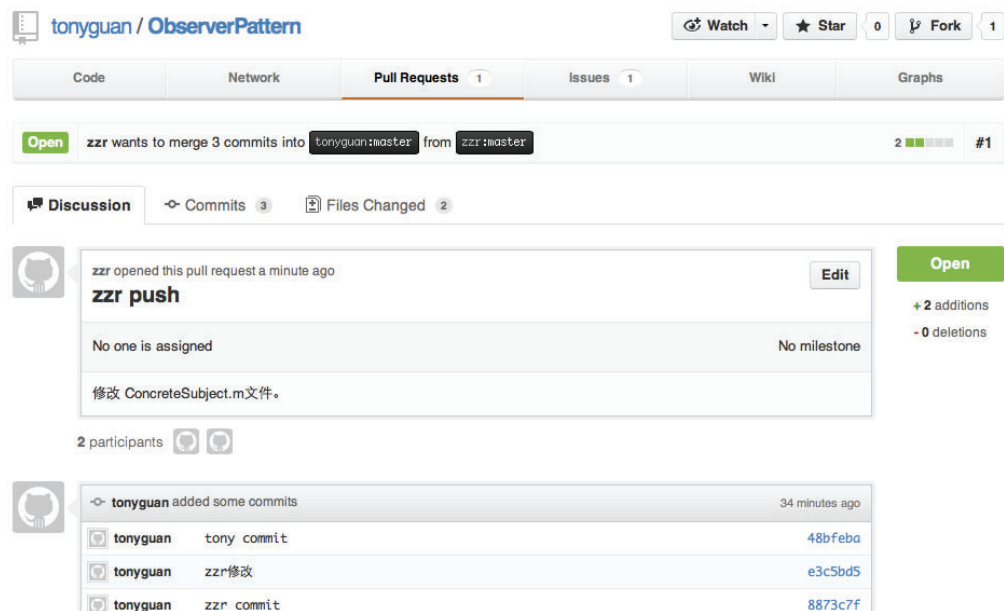


图18-35 发送推送请求的Open状态

我们再看看tonyguan账号，当他进入ObserverPattern代码库时，他发现Pull Requests和Issues为1，这说明有一个问题需要解决，并且有一个推送请求，如图18-36所示。

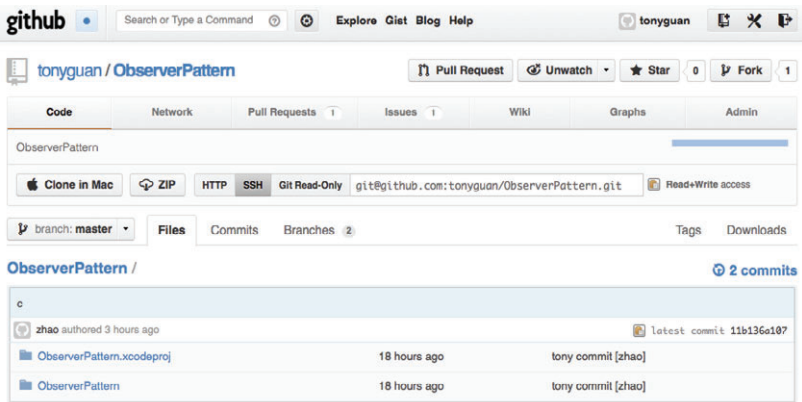


图18-36 推送请求信息

点击Pull Requests按钮，进入如图18-37所示的推送请求详细信息页面。

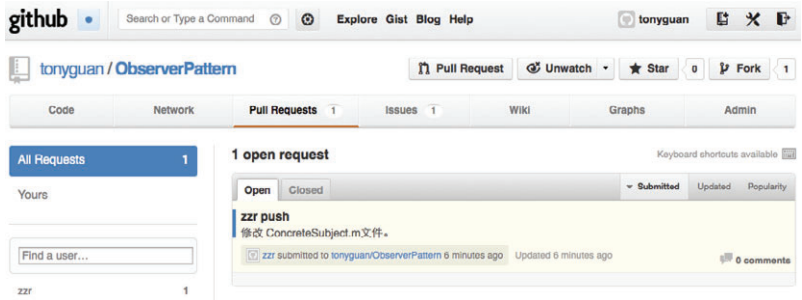


图18-37 推送请求详细信息页面

从中点击zzr push超链接，进入问题讨论页面，如图18-38所示。在这个页面中，可以查看这次推送的讨论、多次讨论的结果以及代码库的提交内容等信息。

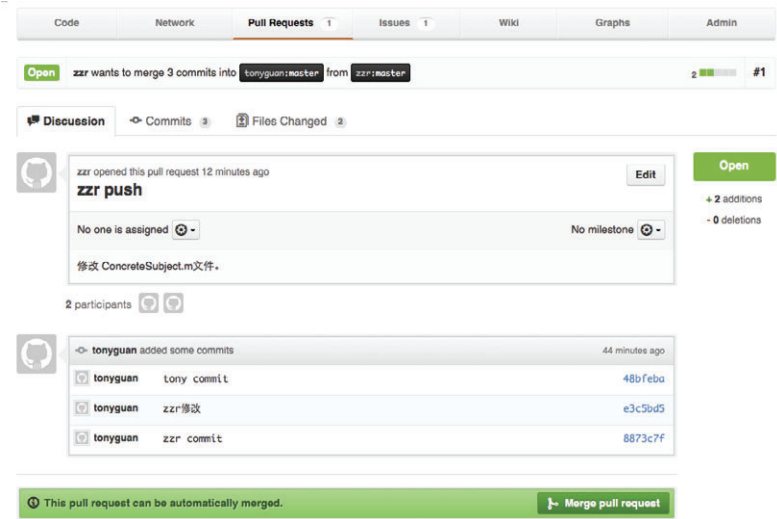


图18-38 问题讨论页面

如果tonyguan认为没有什么问题，就会接受zxr的推送请求，此时他需要合并这两个版本，直接点击Merge pull request按钮，即可进入如图18-39所示的确认页面。

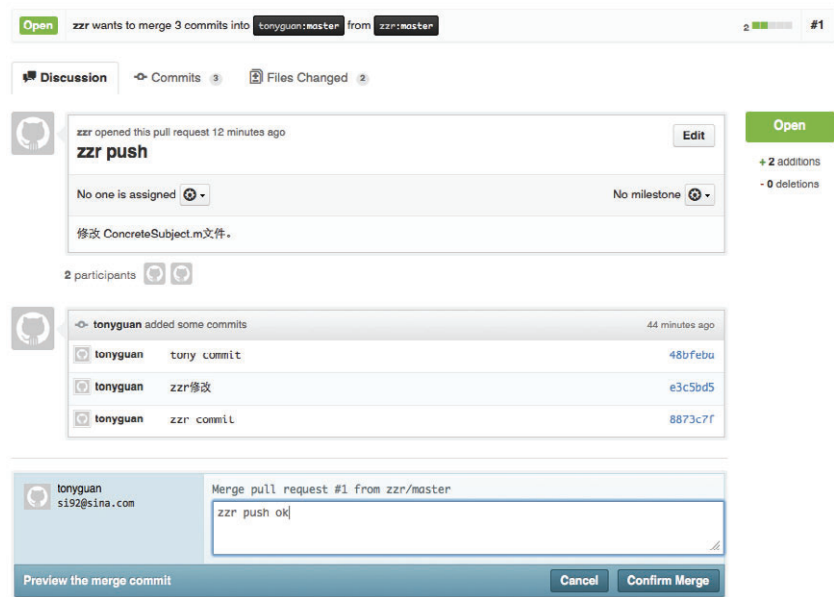


图18-39 合并确认页面

tonyguan会在确认页面输入他的合并意见或是说明，接着点击Confirm Merge按钮确认合并，如果成功，页面会跳转回问题讨论页面。这时候我们看到如图18-40所示的问题关闭页面，说明这个问题已经解决，不需要再讨论了，这个时候Pull Requests和Issues都为0了。

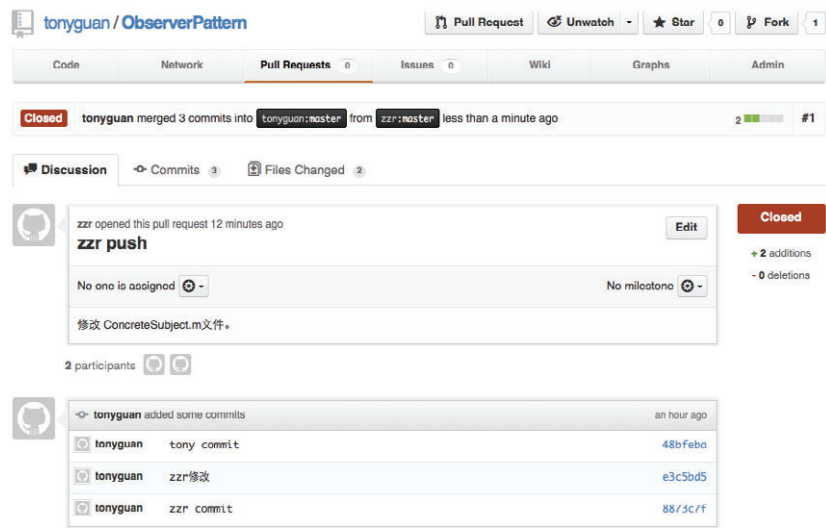



图18-40 问题关闭页面

如果我们回到zxr账户下看这个问题，结果也是关闭状态。这样zxr就将他修改的代码库成功推送给了tonyguan代码库。

18.3.5 管理组织

在协同开发的时候，管理员会让其他人员一起管理代码库，包括响应推送请求、合并推送等，这可以通过在 GitHub 中建立组织来实现。

在登录成功后，通过输入 `https://github.com/account/organizations/new`，或者点击右上角的  按钮，选择 Profile→Organizations 菜单项，进入如图 18-41 所示的组织管理页面。

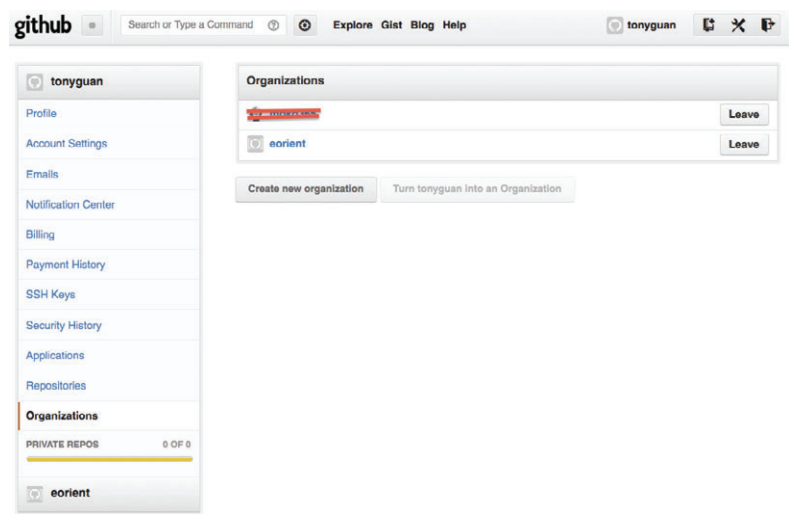


图18-41 组织管理页面

此时可以看到当前账号所属的组织，点击 Leave 按钮即可脱离该组织。点击 Create new organization 按钮，进入创建组织页面，如图 18-42 所示。

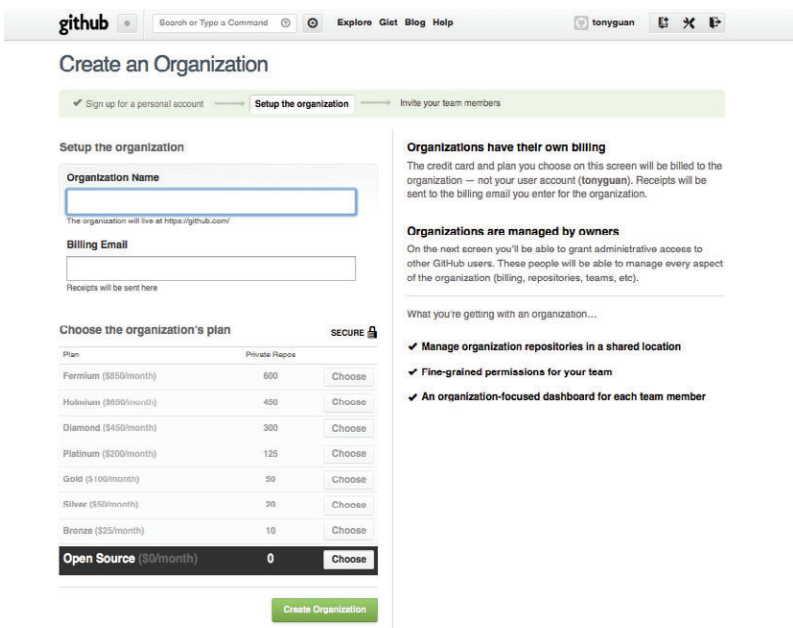


图18-42 创建组织页面

在图18-42中输入组织名和账单邮箱后, 点击Create Organization按钮, 即可进入添加组织成员页面。此时在添加成员文本框中输入GitHub账号, 会出现下拉列表框, 在其中选择要添加的成员, 点击后面的Add按钮邀请该成员进入组织, 如图18-43所示。

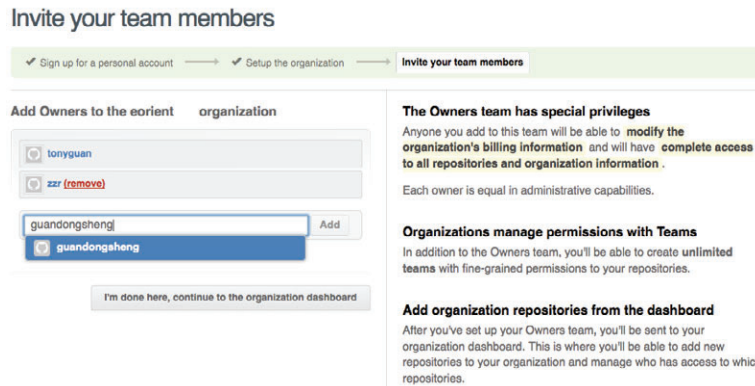


图18-43 添加组织成员页面

添加完组织成员后, 点击I'm done here, continue to the organization dashboard按钮, 此时创建组织的工作就完成了。

组织创建成功后, 组织的成员可以管理代码库。组织获得代码库有两种方法: 一种是自己创建, 过程可以参考一般账户的创建过程; 另一种是由成员转移过来。对于后者, 可以采用下面的方式来实现: 登录到管理的代码库, 滚动屏幕到页面底部, 接着点击Transfer按钮, 此时弹出的对话框如图18-44所示, 从中输入要转移的代码库和新的拥有者。

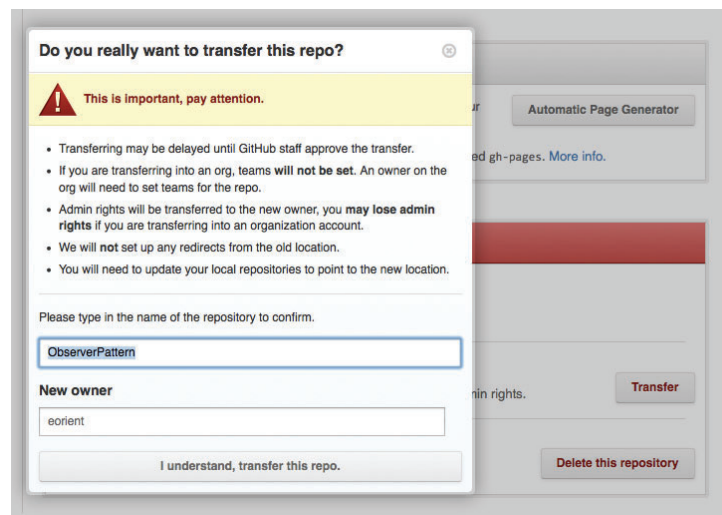


图18-44 转移代码库给组织

如果要转移给的组织是当前账户创建的, 就可以在组织中看到这个库了; 如果是其他的组织或成员, GitHub会有一个审核过程, 不会马上看到。

组织的其他成员在登录成功后, 点击页面的左下角Organizations即可选择将进入的组织, 如图18-45所示。选择某个组织后, 将看到如图18-46所示的页面, 其中可以看到该组织所管理的代码库列表。

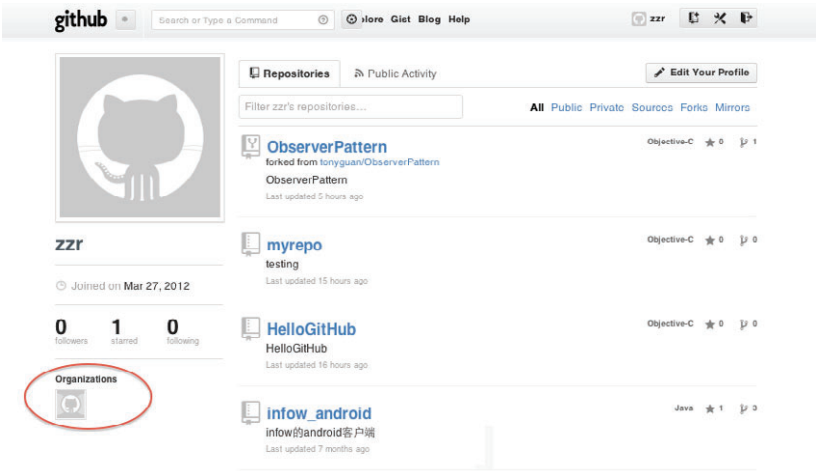


图18-45 其他成员进入组织

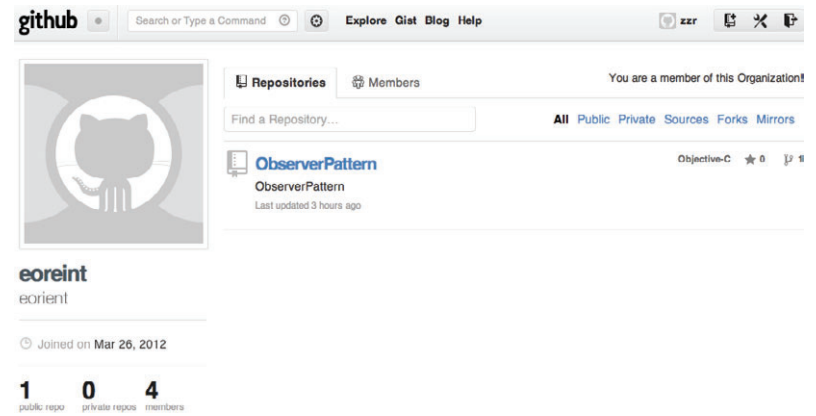


图18-46 组织中的代码库

点击其中的一个代码库即可进入其管理页面。图18-47是zzr账号进入eorient组织看到的ObserverPattern代码库管理页面，zzr不是该eorient组织的创建者，也不是ObserverPattern代码库的转移者。

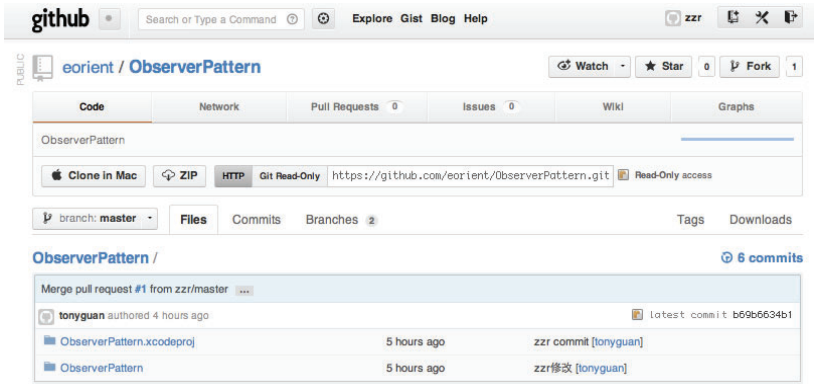


图18-47 zzr进入组织中的代码库

图18-48是tonyguan账号进入eorient组织看到的ObserverPattern代码库管理页面，其中tonyguan是eorient组织的创建者，也是ObserverPattern代码库的转移者。

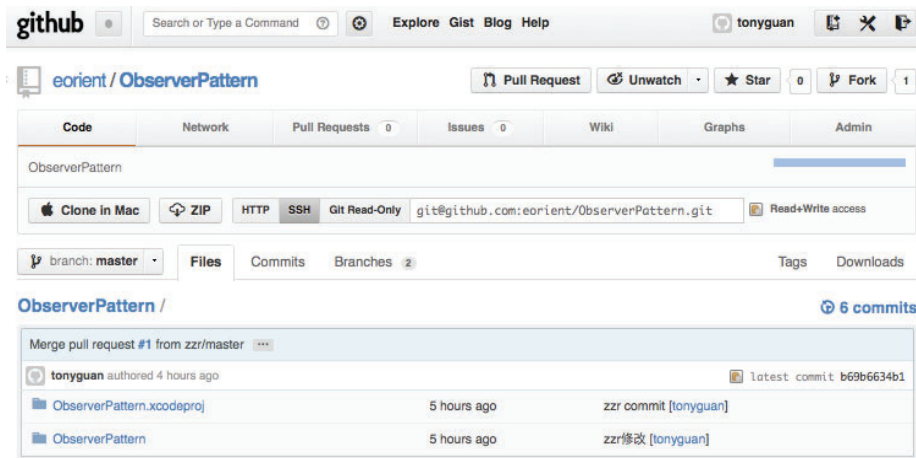


图18-48 tonyguan进入组织中的代码库

比较图18-47和图18-48，会发现什么问题呢？图18-48要比图18-47多一个Admin标签，Admin标签是管理代码库的入口，进入后可以重新命名、删除和转移代码库等。也就是说，zzr不能够对ObserverPattern代码库进行这些操作，这是因为zzr没有eorient组织的管理权限，这是我们需要关注的问题。虽然没有这些组织管理权限，但并不影响zzr响应ObserverPattern代码库的接受推送请求、合并推送等工作。

18.4 小结

通过对本章的学习，我们了解了如何使用Git进行代码版本控制，其中包括Git服务器的搭建、Git常用命令和协同开发，还介绍了如何配置和使用Git工具。GitHub是一个优秀的Git开发社区，使用GitHub代码托管服务是一个不错的选择。

当你阅读到本章的时候，恭喜你已经学习了本书的大部分知识，所具备的知识完全可以开发一个完整应用，并且在App Store上发布。但是在真正把应用发布到App Store之前，还需要“最后一公里”要走，这就是19.1节所要介绍的内容。应用全部开发妥当后，就可以在App Store上发布了，发布过程有点复杂，本章会介绍这个流程。发布完成后，我们就等苹果公司审核了。苹果公司对于发布在App Store上的应用审核非常严格，我们需要避免不通过的情况发生。

19.1 收官

在应用的开发过程中，我们往往只关注意程序本身和功能的实现，而不注重其“外表”，但是当应用要发布时，就需要为它添加启动界面和图标等。此外，还需要调整产品属性和进行编译操作等。

19.1.1 添加图标

用户第一眼看到的就是应用的图标。图标是我们的“着装”，给人很好的第一印象非常重要。“着装”应该大方得体，图标设计也是如此，但图标设计已经超出了本书的讨论范围，这里我们只介绍iOS图标的设计规格以及如何把图标添加到应用中去。

iOS应用使用的图标分为设备上使用的图标（见图19-1）和App Store上使用的图标（见图19-2），它们之间的区别只是尺寸大小的不同，如表19-1所述。



图19-1 设备上使用的图标

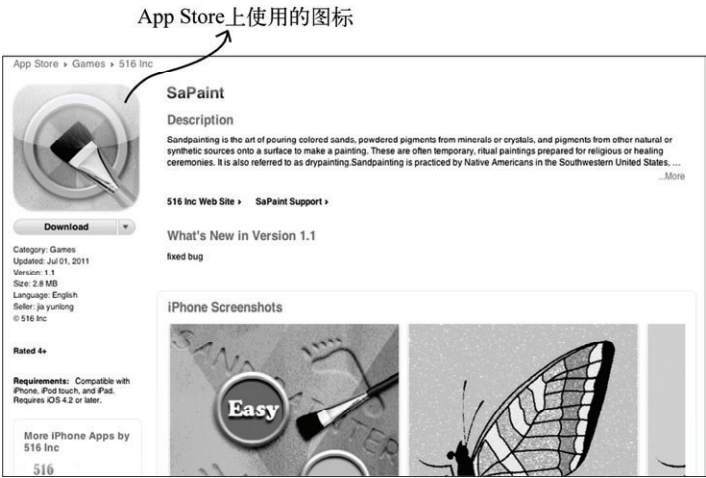


图19-2 App Store上使用的图标

表19-1 iOS应用使用的图标

	设备上使用的图标（单位为像素）	App Store上使用的图标（单位为像素）
iPhone 5和第5代iPod touch	114×114	1024×1024
高分辨率iPhone和iPod touch	114×114	1024×1024
iPhone和iPod touch	57×57	512×512
高分辨率iPad	144×144	1024×1024
iPad	72×72	512×512

App Store上图标的添加过程我们将在下一节介绍，本节我们主要介绍设备上图标的添加过程，主要有如下两种方法。

- ❑ 直接指定文件名的方式。设计图标时，iPhone和iPod touch版本需要两个，iPad版本需要两个，它们的默认命名是：在iPhone和iPod touch上，普通显示屏幕的文件名称为Icon.png，视网膜屏幕的文件名称为Icon@2x.png；在iPad上，普通显示屏幕的文件名称为Icon-72.png，视网膜屏幕的文件名称为Icon-72@2x.png。
然后在Xcode中通过快捷菜单中的Add File to...将这些图标文件添加到工程中就可以了，完成添加后的效果如图19-3所示。
- ❑ 通过工程的TARGETS来选择文件。使用Xcode打开工程，选择TARGETS→Summary→App Icons，然后右击No image specified项，此时从弹出的快捷菜单中选择Select File项即可，如图19-4所示。
这种方法虽然对于文件名称没有要求，但是文件的大小必须是57×57像素和114×114像素。在图19-4中，①号图标是57×57像素，是为普通iPhone和iPod touch准备的；②号图标是114×114像素，是为高分辨率iPhone和iPod touch以及iPhone 5和第5代iPod touch准备的。

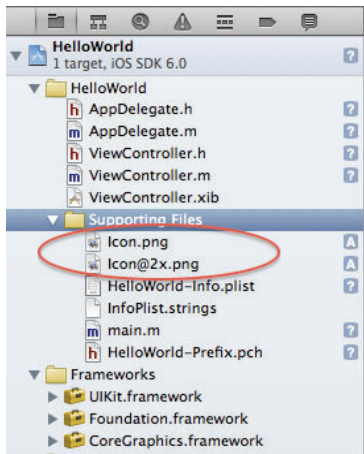


图19-3 添加图标文件到Xcode工程

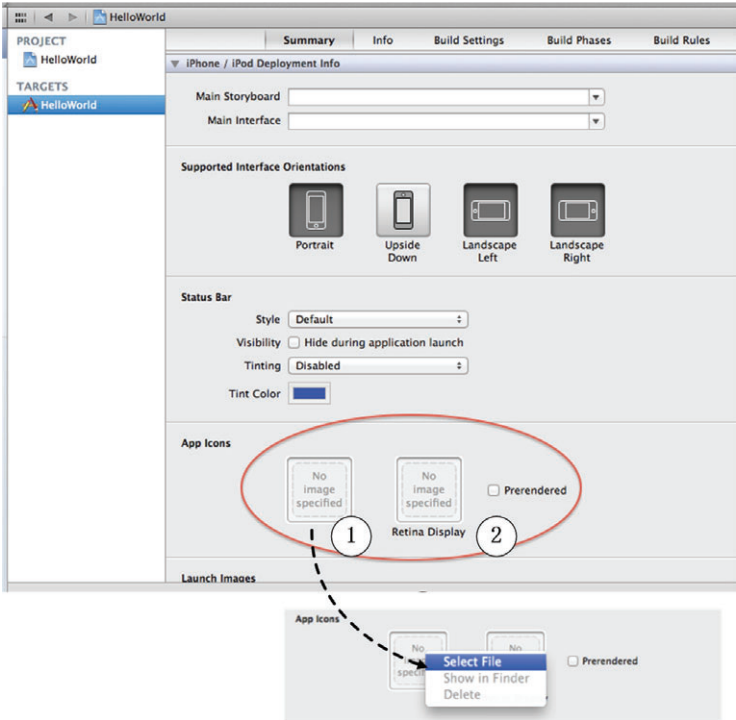


图19-4 选择图标文件到Xcode工程

19.1.2 添加启动界面

启动界面是应用启动与进入到第一个屏幕之间的界面。没有启动界面的应用进入第一个屏幕之前是黑屏，这会影响用户体验。虽然这在开发阶段没有什么影响，但是在应用发布前，还是需要添加启动界面的。

添加启动界面有很多学问，下面我们先看看两个应用的启动过程，如图19-5和图19-6所示。



图19-5 启动界面场景1

你认为这两个启动界面哪个更好呢？图19-5所示的应用进入第一个屏幕（④界面）时，要经过3个界面的变化，其中启动界面放有太大的logo，并且设计者还故意让其延迟了几秒钟，让用户看清楚这个logo。这样做或许很酷，也能宣传自己，但事实上用户既然能够下载你的应用，肯定知道你是谁了！如果第一次看，用户可能感觉很新奇，但是如果每天看会是什么感觉呢？

注意，图19-6所示的①界面不是真的已经进入应用了，而是一张图片，它与应用的第一屏幕非常相似，它能够使用户感觉到很快进入了应用，让用户没有感觉到等待，这才是以用户体验为中心的设计。

为了获得更好的用户体验，苹果对于iOS上的启动界面推荐采用图19-6所示的设计。在iOS中，苹果自带的应用都是这样设计的，比如自带的股票应用界面（见图19-7）。

图19-7中的①号图片是启动界面，与第一个屏幕（②号图片）非常类似，就是将动态部分去掉了，它给用户的感受是应用快速启动，几秒钟后，内容就会填充到屏幕中。



图19-6 启动界面场景2



图19-7 股票应用启动界面

在启动界面的规格上，苹果也给出了相应的尺寸要求，如表19-2所述。

表19-2 启动界面图片规格

设 备	图片大小	说 明
iPhone 5和第5代iPod touch	640×1136	包含状态栏，文件命名为Default-568h@2x.png
高分辨率iPhone和iPod touch	640×960	包含状态栏，文件命名为Default@2x.png
iPhone和iPod touch	320×480	包含状态栏，文件命名为Default.png
高分辨率iPad	1536×2008（竖屏） 2048×1496（横屏）	不包含状态栏，文件命名为Default-Portrait@2x~ipad.png（竖屏）或者Default-Landscape@2x~ipad.png（横屏）
iPad	768×1004（竖屏） 1024×748（横屏）	不包含状态栏，文件命名为Default-Portrait~ipad.png（竖屏）或者Default-Landscape~ipad.png（横屏）

从表19-2中可以看出，iPhone和iPod touch设备上的启动界面只有竖屏而不支持横屏，iPad设备支持横屏和竖屏。它们之间还有一个差别，那就是iPhone和iPod touch上的启动界面图片包含状态栏，因此，在iPhone和iPod touch设备上，启动界面上边会与状态栏重叠，重叠部分会被遮盖，如图19-8所示，其中①号图片是启动界面图片，②号图片是设备运行时看到的启动界面，虚线部分之上的内容被遮盖了。

而iPad设备上不包含状态栏这个高度，如图19-9所示，其中①号图片是启动界面图片，②号图片是设备运行时看到的启动界面，状态栏没有被启动界面遮盖。

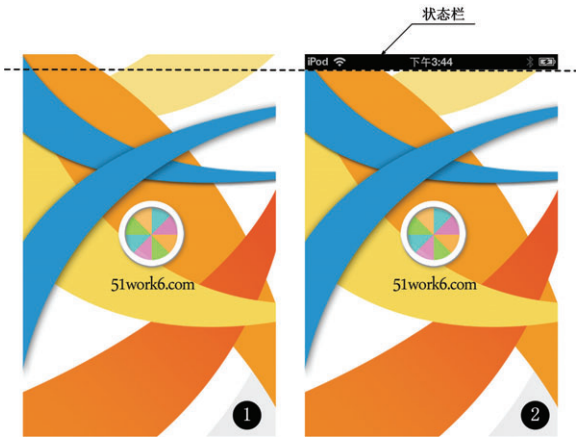


图19-8 iPhone和iPod touch设备上的启动界面

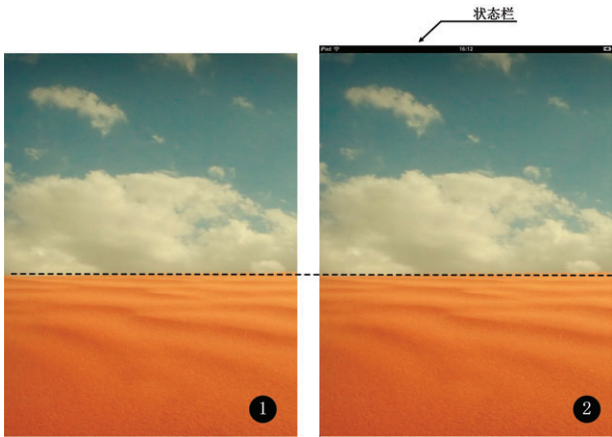


图19-9 iPad设备的启动界面

提示 状态栏的高度在不同的设备上是不同的，如表19-3所述。由于iOS高分辨率设备采用视网膜显示技术，它的一个点由4个像素组成，其中高度和宽度都是原来设备的2倍，因此在普通设备上为20点时，在高分辨率设备上就是40点。

表19-3 状态栏的高度

设 备	状态栏的高度
iPhone 5和第5代iPod touch	40像素或20点
高分辨率iPhone和iPod touch	40像素或20点
iPhone和iPod touch	20像素或20点
高分辨率iPad	40像素或20点
iPad	20像素或20点

与添加图标非常类似，添加启动界面也有两种方式：我们可以采用指定文件名的方式添加，此时文件的命名必须与表19-2所述的命名方式一致；如果我们觉得命名比较麻烦，也可以使用Xcode工具。图19-10演示了是在iPad设备上添加图片的过程，具体步骤为在Launch Images中选择其中一个并右击，然后从弹出的快捷菜单中选择Select File菜单项即可。如果图片右下角有感叹号，说明没有这个图片或其大小不符合规范。

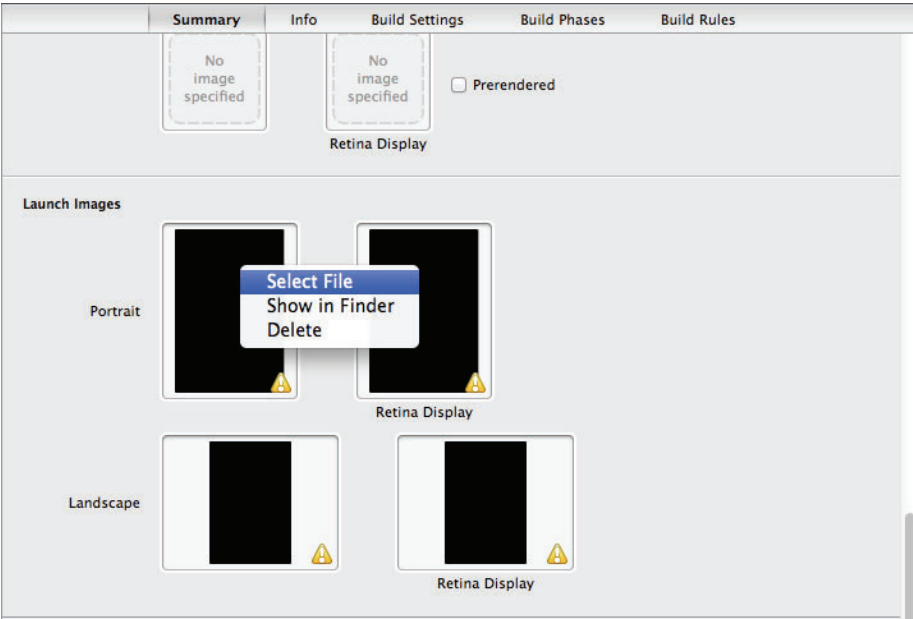


图19-10 在iPad上添加启动界面

19.1.3 调整Application Target属性

在编程过程中，有些产品的属性并不影响我们开发，即便这些属性设置不正确，一般也不会有什么影响。但是在产品发布时，正确地设置这些属性就很重要了。如果设置不正确，就会影响我们产品的发布。这些产品属性主要是TARGETS中的Application Target属性，如图19-11所示。

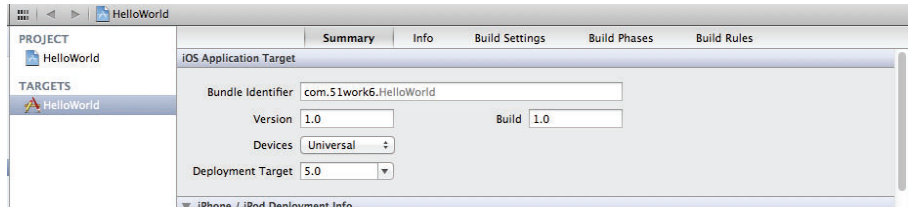


图19-11 Application Target属性

在这些属性中，主要包括Bundle identifier（包标识符）、Version（版本）和Deployment Target（部署目标）等，下面分别介绍它们的含义和重要性。

- ❑ 包标识符。包标识符在开发过程中对我们似乎没有什么影响，但是在发布时非常重要。事实上，我们已经在使用它了。在13.5.3节中创建App ID时就使用了包标识符，它是组成App ID的一部分。App ID在测试、发布和应用内购买等方面都会用到，因此包标识符也是非常重要的。包标识符的命名一般是“公司域名反写+应用产品名”。默认创建工程时，会创建图19-11中所示的包标识符com.51work6.HelloWorld，其中

HelloWorld是我们的工程和产品Target的名字。有时候，在发布应用时觉得这个名字不够好，想修改一下，但是在图19-11中我们只能修改前面的com.51work6，而不能修改后面的HelloWorld。为了修改这个名字，我们可以修改工程的名字。此外，还有另外一个比较简单直接的方法。所有的产品属性都在工程属性描述文件HelloWorld-Info.plist（工程名+Info.plist）中，因此我们可以直接修改，只是有些不太直观。打开HelloWorld-Info.plist，直接修改Bundle identifier为你想要的包标识符，如图19-12所示。

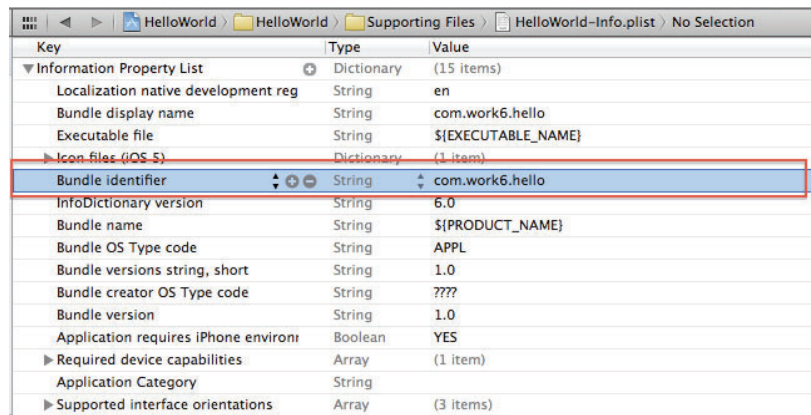


图19-12 修改包标识符属性

- ❑ 版本。应用的版本号看起来无关紧要，但是在发布时如果这里设定的版本号（图19-11中的Version）与iTunes Connect中设置的应用的版本号（图19-13中的Version）不一致，在打包上传到App Store上就会失败。关于如何在iTunes Connect中设置应用版本号，我们会在下一节中介绍。此外，这个版本号也会显示在App Store中。
- ❑ 部署目标。选择部署目标是开发应用之前就要考虑的问题，这关系到应用所能够支持的操作系统。如果考虑到支持老版本的操作系统（如低于4.3版本），就要将Deployment Target设为4.3，如图19-14所示。

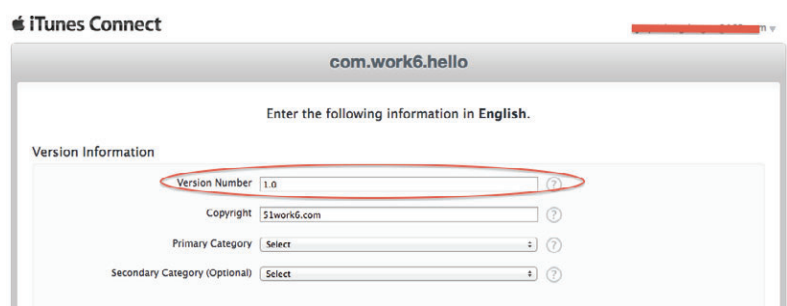


图19-13 iTunes Connect中设置的应用版本号

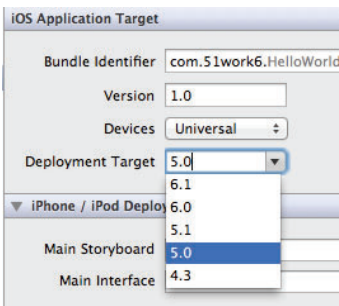


图19-14 选择操作系统版本

这样的修改也会出现新的问题，那就是iOS 5以后的新技术都无法使用，只能使用iOS 4.3的API。在产品升级的时候一定要谨慎，我们上传应用到App Store后，App Store会自动识别这个版本号。

19.1.4 为发布进行编译

从编写到发布应用会经历3个阶段：在模拟器上运行调试、在设备上运行调试和发布编译。苹果为了防止非法设备和非开发人员调试和发布应用，使用配置概要文件控制在设备上运行调试和发布编译阶段。配置概要文件分为两种——开发配置概要文件和发布配置概要文件，它们的创建非常相似（具体可参考15.5.4节）。完整的编译

发布流程如图19-15所示，其中创建开发者证书和创建App ID与调试阶段没有区别，具体请参考15.5节，下面介绍一下最后的两个步骤。

1. 创建发布配置概要文件

先登录iOS开发中心的配置门户网站，选择左边的导航菜单Provisioning，然后选择Distribution标签，如图19-16所示。

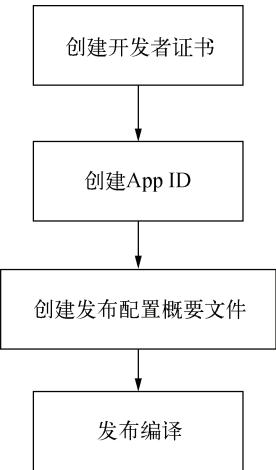


图19-15 发布编译流程

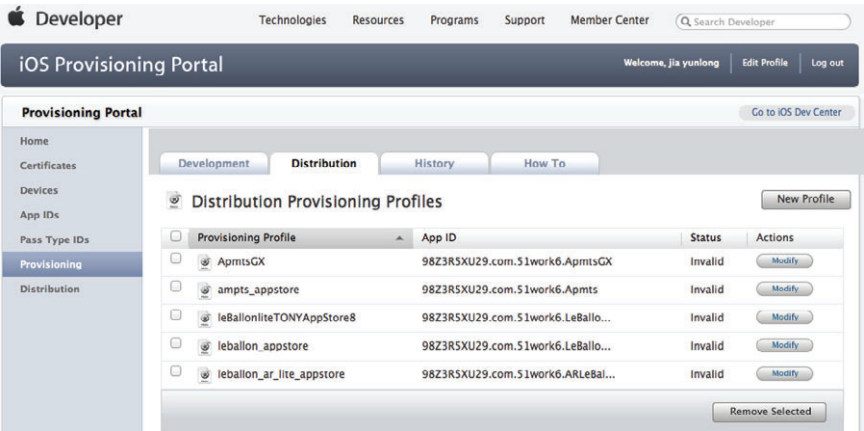


图19-16 管理发布配置概要文件

接着点击New Profile按钮，进入创建页面，如图19-17所示，其中Distribution Method有两个选项：App Store和Ad Hoc。Ad Hoc用于生成与设备关联的应用测试版本，供测试人员测试，因此它还属于测试。App Store是为发布创建的。

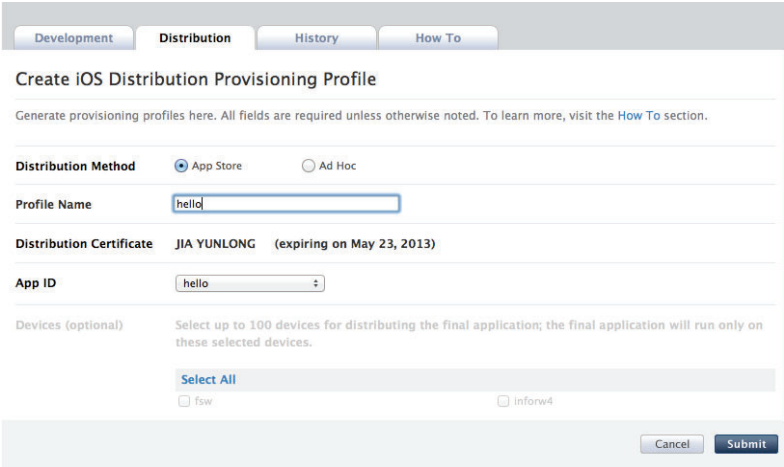


图19-17 创建发布配置概要文件

在Distribution Method中选择App Store单选按钮，在Profile Name文本框中输入hello，在App ID中选择我们创建的hello。输入完成后，点击Submit提交表单。成功后会跳转到如图19-16所示的页面，在这里我们会看到添加的hello。刷新页面，我们会看到hello处于活动状态，如图19-18所示，此时点击Download按钮，可以下载发布配置概要文件到本地。

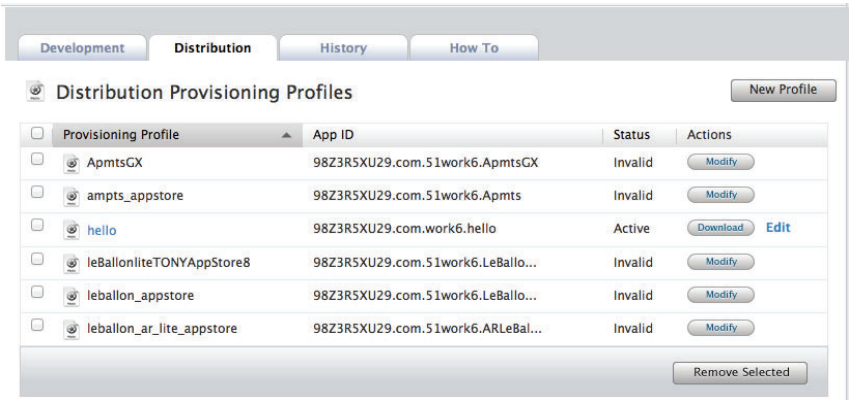


图19-18 下载发布配置概要文件

2. 发布编译

找到发布配置概要文件hello.mobileprovision，双击它，此时会进入Xcode设备管理工具，如图19-19所示。

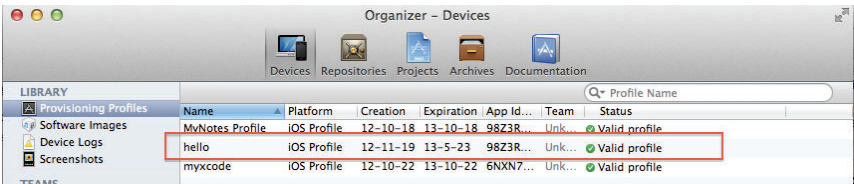


图19-19 打开发布配置概要文件

然后使用Xcode打开需要编译的工程或工作空间，选择工程的TARGETS，选择Build Settings→Code Signing→Code Signing Identity，选择Release的代码签名标识符为hello，如图19-20所示。

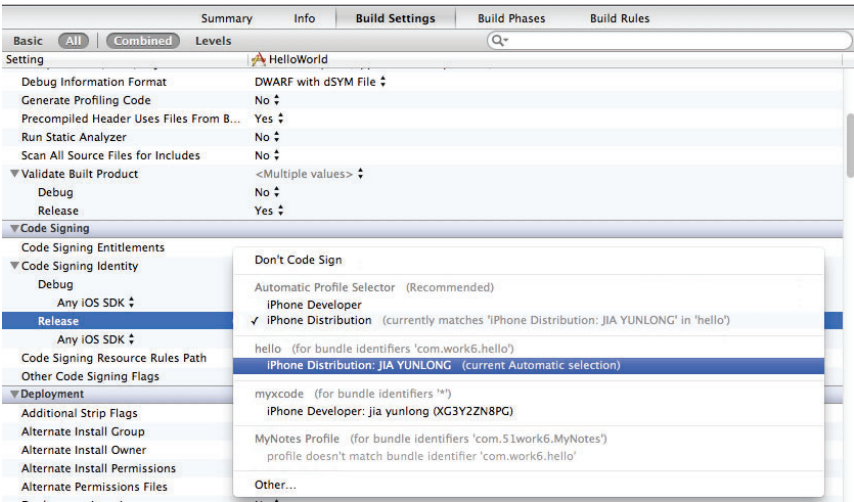


图19-20 选择代码签名标识符

接着选择工具栏中的Edit Scheme，打开编辑Scheme对话框，如图19-21所示，选择左下角的Duplicate Scheme按钮，复制一份新的Scheme为HelloWorld 2。

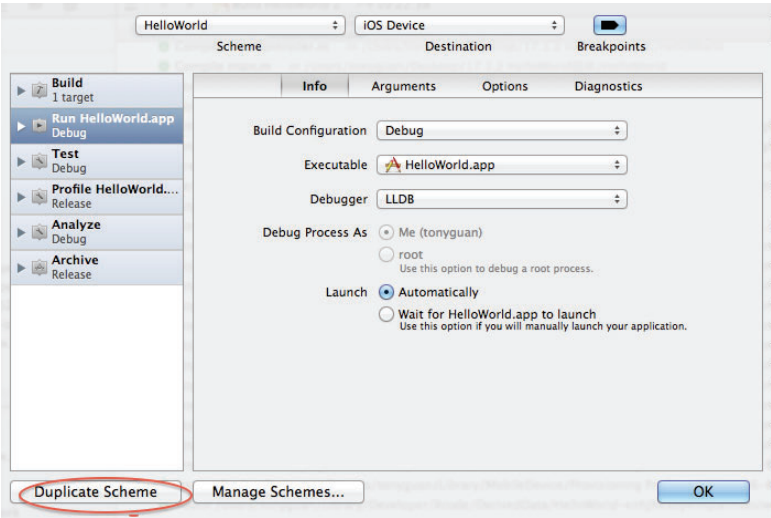


图19-21 复制Scheme

在左上角的Scheme选择下拉框中选择HelloWorld 2，然后在左边的列表中Run HelloWorld.app，在右边选择Info标签，在Build Configuration下拉框中选择Release，如图19-22所示。

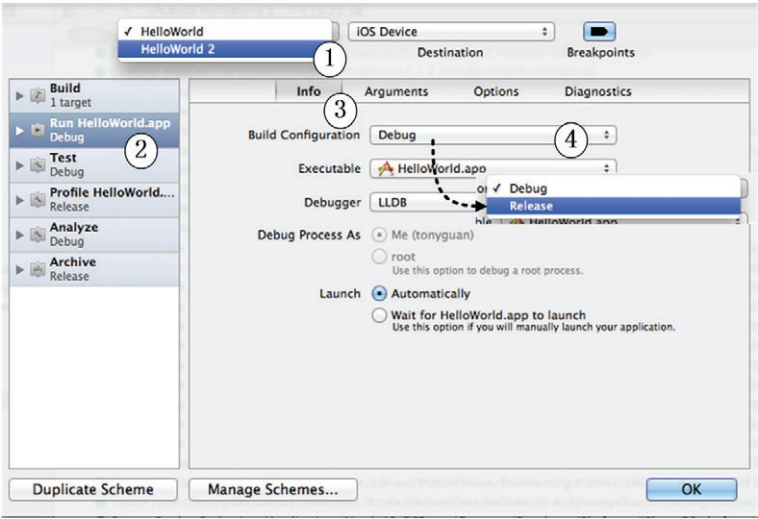


图19-22 配置HelloWorld 2

配置完成之后，选择HelloWorld 2中的iOS Device，如图19-23所示。再选择Product→Building for→Running菜单项，然后就可以编译了。

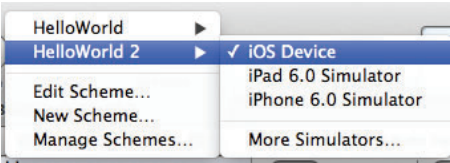


图19-23 选择HelloWorld 2中的iOS Device

如果编译结果有错误或警告，必须要解决，忽略警告往往也会导致发布失败。

在发布编译成功后，打开显示日志导航面板，我们会看到刚刚执行的Build HelloWorld 2已经成功了，如图19-24所示。

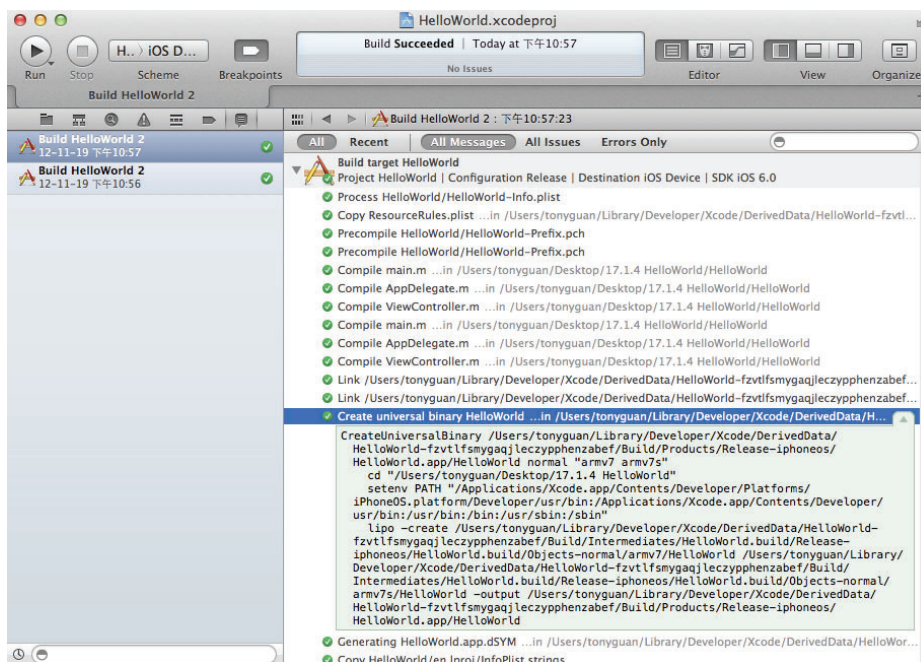


图19-24 发布编译成功

19.1.5 应用打包

在把应用上传到App Store之前，我们需要把编译的二进制文件和资源文件打成压缩包，压缩格式是ZIP。首先找到编译到什么地方，这个很重要也不太好找，我们可以看看图19-24所示的编译日志，找到其中的Create universal binary HelloWorld...的内容，然后展开内容，具体如下：

```
Create Universal Binary /Users/tonyguan/Library/Developer/Xcode/
DerivedData/HelloWorld-fzvtlfsmygaqjleczypphenzabef/Build/Products/
Release-iphonios/HelloWorld.app/HelloWorld normal "armv7 armv7s"
cd "/Users/tonyguan/Desktop/19.1.4 HelloWorld"
setenv PATH "/Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneOS.platform/Developer/usr/bin:/Applications/Xcode.app/
Contents/Developer/usr/bin:/usr/bin:/bin:/usr/sbin:/sbin"
lipo -create /Users/tonyguan/Library/Developer/Xcode/DerivedData/
HelloWorld-fzvtlfsmygaqjleczypphenzabef/Build/Intermediates/
HelloWorld.build/Release-iphonios/HelloWorld.build/Objects-
normal/armv7/HelloWorld /Users/tonyguan/Library/Developer/Xcode/
DerivedData/HelloWorld-fzvtlfsmygaqjleczypphenzabef/Build/
Intermediates/HelloWorld.build/Release-iphonios/
HelloWorld.build/Objects-normal/armv7s/HelloWorld -output /
Users/tonyguan/Library/Developer/Xcode/DerivedData/HelloWorld-
fzvtlfsmygaqjleczypphenzabef/Build/Products/Release-iphonios/
HelloWorld.app/HelloWorld
```

-output之后就是应用编译之后的位置了，其中/Users/tonyguan/Library/... /Products/Release-iphonios/是编译之后生成的目录，如图19-25所示，其中HelloWorld.app是包文件。

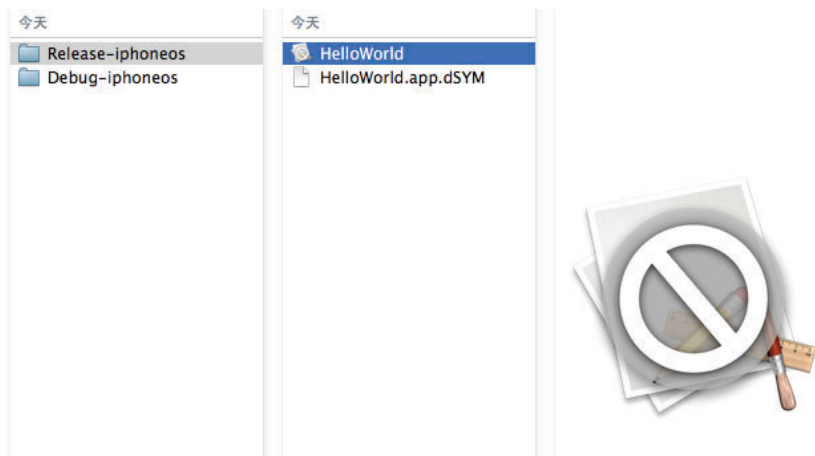


图19-25 编译之后生成的目录

使用快捷菜单,可以显示HelloWorld.app包文件的内容,如图19-26所示,其中HelloWorld文件是我们这个应用的二进制文件,其他的都是资源文件,包括图片、属性列表文件、nib and storyboard文件。nib是编译之后的xib文件,storyboard是编译之后的故事板文件等。

应用打包就是将HelloWorld.app包文件打包成HelloWorld.zip,具体操作是右击HelloWorld.app包文件,从弹出的快捷菜单中选择“压缩‘HelloWorld’”菜单项,如图19-27所示。这样就会在当前目录下生成HelloWorld.zip压缩文件了,请将这个文件保存好,我们会在后面用到。

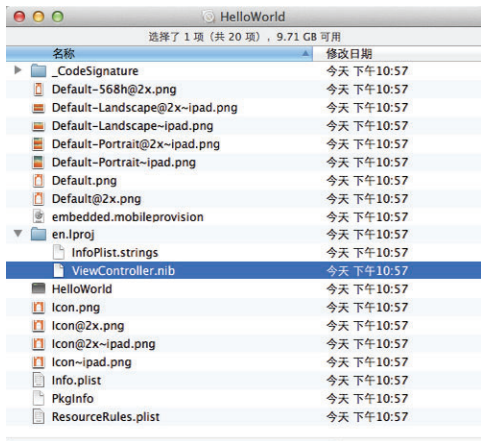


图19-26 HelloWorld.app包文件中内容

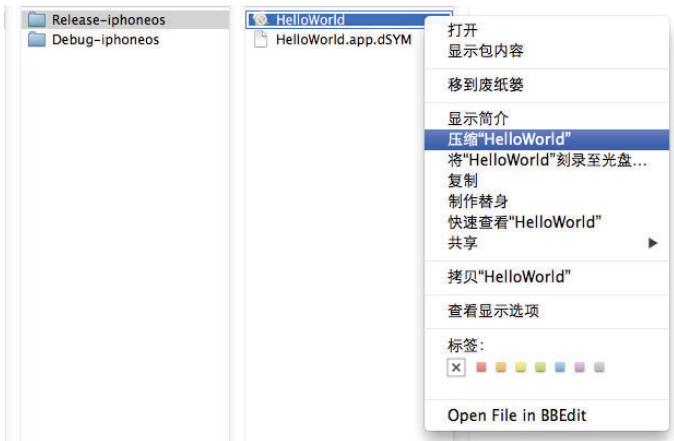


图19-27 压缩HelloWorld.app包文件

19.2 发布流程

程序打包后,就可以发布我们的应用了。发布应用在iTunes Connect中完成,发布完成后等待审核,审核通过后就可以在到App Store上销售了。详细的发布流程如图19-28所示。

其中第A步、第B步、第C步和第D步主要在iOS开发中心的配置门户网站中完成,这在本章前面已经介绍过了。这里我们介绍其他几个流程,其中主要的流程是在iTunes Connect中完成的,而上传应用使用Application Loader工具实现。

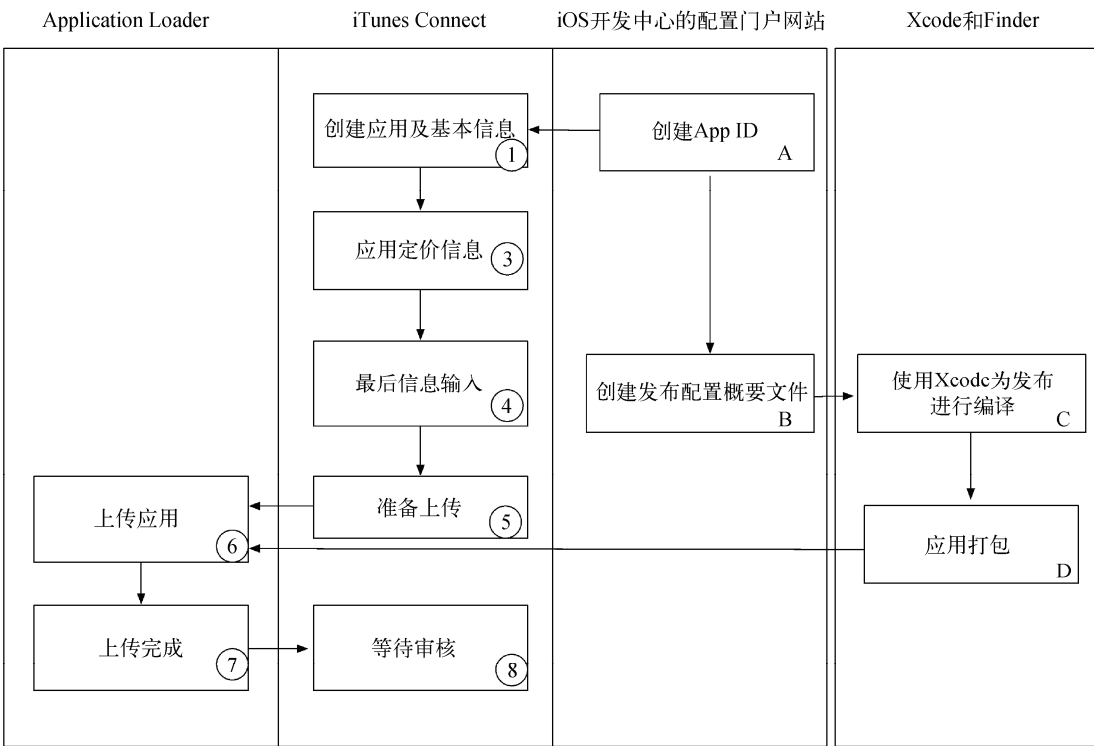


图19-28 发布流程图

19.2.1 创建应用及基本信息

通过网址<https://itunesconnect.apple.com/WebObjects/iTunesConnect.woa>打开iTunes Connect登录页面，使用苹果开发账号登录，登录成功后的iTunes Connect页面如图19-29所示。

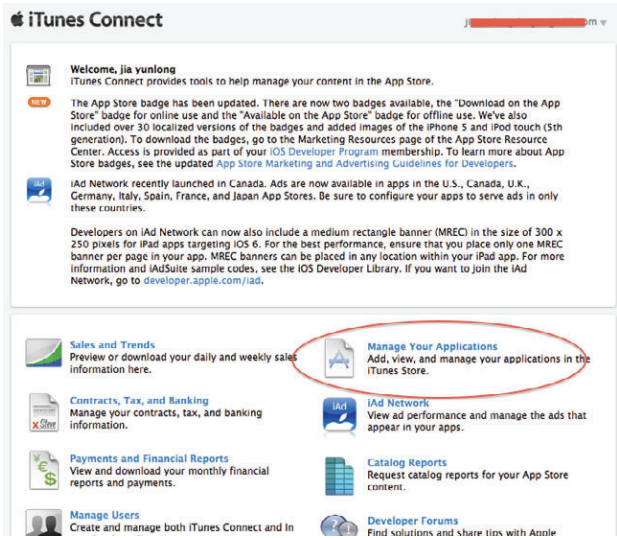


图19-29 iTunes Connect登录成功

点击Manage Your Applications图标，进入应用管理页面，如图19-30所示，在这里可以管理我们的应用，其中显示审核中的、未通过的以及已经上线的所有应用。



图19-30 应用管理页面

点击左上角的Add New App按钮，进入添加新应用页面，在这里可以输入应用的信息，如图19-31所示。

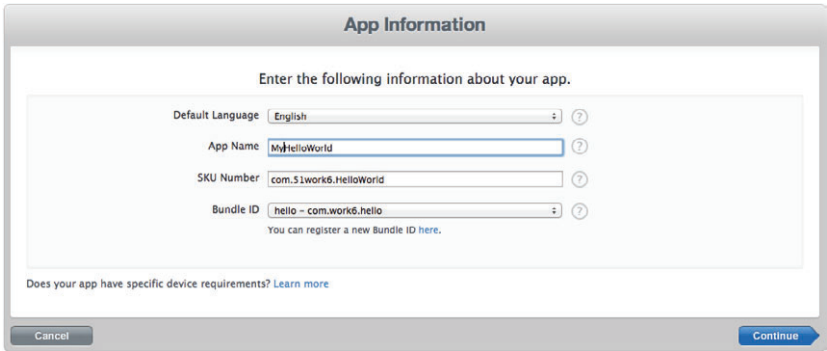


图19-31 添加新应用页面

在Default Language选择框中选择应用的默认语言。除了默认语言，我们还可以添加其他语言。在App Name文本框中输入应用的名称，这个名称是显示到App Store上面的名字，是不能重复的。在SKU Number文本框中输入应用的SKU号码。SKU是应用程序编号，具有唯一性，因此建议使用公司的“域名反写+应用名”，这里我们输入的是com.51work6.HelloWorld。在Bundle ID中输入应用包标识符，它是在iOS开发中心的配置门户网站创建App ID时生成的，如果配置门户网站中有就可以在下拉列表中找到。

19.2.2 应用定价信息

在图19-31中点击Continue按钮，进入选择发布日期和定价页面，如图19-32所示。

其中Availability Dates是应用可以使用的日期，Price Tier是应用的定价。这或许是我们最关心的了，定价只能选择不能输入，可以从Free~Tier87的88个收费档次选择，最高定价Tier87是6 498.00元。理论上，你可以定到这么高，是否能够卖得出就要看市场反馈了。这个定价很灵活，以后也可以修改。Discount for Educational Institutions表示为教育机构打折，Custom B2B App是自定义B2B应用，适用于批量购买的用户。



图19-32 选择发布日期和定价页面

19.2.3 最后信息输入

在图19-32中点击Continue按钮，将进入最后的信息输入页面，其中包含更加详细的部分，包括版本信息、元数据、应用审核信息、最终用户许可协议（EULA）以及上传应用图标和截图，下面我们分别介绍一下。

1. 版本信息

版本信息输入页面如图19-33所示。Version Number是应用的版本号，它必须与图19-11所示应用Target属性中的Version（应用版本号）一致，否则上传应用会失败。

Version Information

Version Number1.0

Copyright516 Inc.

Primary CategoryEducation

Secondary Category (Optional)Select

Rating

For each content description, choose the level of frequency that best describes your app.

App Rating Details

Apps must not contain any obscene, pornographic, offensive or defamatory content or materials of any kind (text, graphics, images, photographs, etc.), or other content or materials that in Apple's reasonable judgment may be found objectionable.

Apple Content Descriptions	None	Infrequent/Mild	Frequent/Intense
Cartoon or Fantasy Violence	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Realistic Violence	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Sexual Content or Nudity	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Profanity or Crude Humor	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Alcohol, Tobacco, or Drug Use or References	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Mature/Suggestive Themes	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Simulated Gambling	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Horror/Fear Themes	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Prolonged Graphic or Sadistic Realistic Violence	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Graphic Sexual Content and Nudity	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Graphic Sexual Content and Nudity	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

4+

App Rating

图19-33 版本信息输入页面

Copyright是版权信息，这里填上自己的版权信息就可以了。Primary Category用于选择应用的分类，也就是应用会发布到哪个频道，如果选择游戏，还要进行细化分类，因为游戏是App Store中最多的应用，所以分得比较细。

Secondary Category是第二分类。这两个分类选项可以根据自己的应用进行填写，要求不是特别严格。

然后设置Rating选项，这里主要根据应用中含有色情、暴力等内容的程度进行分级。不同的等级标识使用该应用的年龄段。同时，也会有一些国家根据这个评级高低来限制是否在本国销售。在这个选项中，开发者应该按应用的实际情况来填写，如果与所描述的内容不符，苹果会拒绝审核通过。

2. 元数据

元数据输入页面如图19-34所示。Description是应用描述信息，这个描述对应用很重要，将出现在App Store的应用介绍中。用户购买应用时，主要通过这段文字来了解我们的应用到底是做什么的，有什么用。因此，要认真、用心地准备这段文字，描述清楚应用的所有功能，体现出应用的特点、特色等，从而吸引用户来购买。

Metadata

Description	<input type="text" value="这是一个测试Demo"/>	?
Keywords	<input type="text" value="测试 Demo"/>	?
Support URL	<input type="text" value="http://www.51work6.com"/>	?
Marketing URL (Optional)	<input type="text" value="http://iosshare.cn"/>	?
Privacy Policy URL (Optional)	<input type="text" value="http://"/>	?

图19-34 元数据信息输入页面

Keywords是在App Store上查询该应用的关键词。Support URL里面需要填写应用技术支持的网址，Marketing URL里面填写应用营销的网址，主要是针对应用做进一步介绍。由于Description描述的文字和图片数是有限制的，可能不会把应用介绍得很详尽，所以我们可以自己创建一个网页，更详细地介绍我们的应用。Privacy Policy URL是填写隐私政策网址的地方，很多网站下面都有这个自己隐私政策的链接。

3. 应用审核信息

应用审核信息输入页面如图19-35所示，这里的信息主要是给苹果审核团队的工作人员看的。在Contact Information中填写开发者团队中负责与苹果审核小组联系的人员的信息，包括姓名、邮箱和电话号码。

App Review Information

Contact Information ?

First Name	<input type="text" value="tony"/>
Last Name	<input type="text" value="guan"/>
Email Address	<input type="text" value="eorient@sina.com"/>
Phone Number	<input type="text" value=""/>

Include your country code

Review Notes (Optional) ?

Demo Account Information (Optional) ?

Username	<input type="text"/>
Password	<input type="password"/>

图19-35 应用审核信息输入页面

在Review Notes中, 填写应用细节和一些特别的功能, 帮助审核人员快速了解该应用。在Demo Account Information中, 填写应用中的测试账号和密码, 提供给审核人员测试, 以便于更加顺畅通过审核。

4. 最终用户许可协议

最终用户许可协议输入页面如图19-36所示。最终用户许可协议只有用户同意后才能下载我们的应用。如果没有特别的, 建议不要添加。

EULA

If you want to provide your own End User License Agreement (EULA), [click here](#). If you provide a EULA, it must meet these [minimum terms](#). If you do not provide a EULA, the [standard EULA](#) will apply to your app.

EULA Text

Enter the text of your EULA above. It must meet these [minimum terms](#).

Select the countries in which your EULA applies. Select only the countries for which your EULA has been properly localized to meet local legal and language requirements. For all other countries, the [standard EULA](#) will apply.

Select All Deselect All

Albania	<input type="checkbox"/>	Dominica	<input type="checkbox"/>	Luxembourg	<input type="checkbox"/>	Senegal	<input type="checkbox"/>
Algeria	<input type="checkbox"/>	Dominican Republic	<input type="checkbox"/>	Macau	<input type="checkbox"/>	Seychelles	<input type="checkbox"/>

图19-36 最终用户许可协议输入页面

EULA Text是用户协议文本, 下面可以选择国家。

5. 上传应用图标和截图

上传应用图标和截图填写页面如图19-37所示, 这里可以上传应用的一些图片, 包括应用图标 (在App Store上使用的图标)、iPhone和iPod touch截图、iPhone 5和第5代iPod touch截图以及iPad的一些截图等。这里要注意所有图片尺寸的要求、格式要求以及DPI要求。随着系统升级, 苹果要求的内容也一直在变化, 详细内容可以参考苹果说明。

Uploads

Large App Icon ?

Choose File

3.5-Inch Retina Display Screenshots ?

Choose File

4-Inch Retina Display Screenshots ?

Choose File

iPad Screenshots ?

Choose File

Routing App Coverage File (Optional) ?

Choose File

Go Back

Save

图19-37 上传应用图标和截图页面

关于图标设计以及图片截取，我们一定要下点功夫。图标能够给用户带来第一印象的感受，所以一定要用心去设计。要让用户了解我们的应用，除了上面提到的描述外，另一个就是这里的截图了。截图往往比文字描述更形象、更具有说服力。应用可能有很多情景和功能，我们一定要挑选最具特色、最突出的功能截图。由于上传的截图不能超过5张，一定要把最好的图片放到前面，因为后面的图片需要向后滑动才能出现，这样才能吸引用户对我们的应用产生兴趣，考虑购买我们的应用。上传完成后，点击Save按钮，会进入如图19-38所示的最后信息输入成功页面。

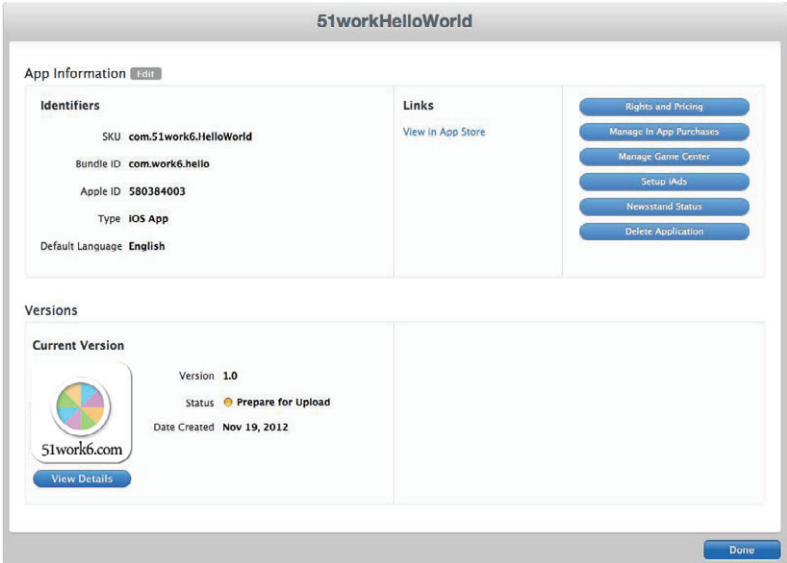


图19-38 最后信息输入成功页面

完成这些工作后，就已经在iTunes Connect中创建了一个应用，这时应用的状态是Prepare for Upload（准备上传）状态。在不同阶段，应用的状态是不同的，如等待上传、等待审核和等待销售等状态。

19.2.4 上传应用

现在就可以上传应用了。首先，在图19-38中点击左下角的View Details按钮，进入应用详细信息页面，如图19-39所示。



图19-39 应用详细信息页面

点击右上角的Ready to Upload Binary按钮，进入出口规定页面，如图19-40所示。



图19-40 出口规定页面

这里询问代码中是否有加密算法，美国出口法律规定禁止任何加密的软件流向国外，这里我们选择No即可。点击Save按钮，就可以上传应用了，此时我们的应用处于Waiting For Upload（等待上传）状态。

还记得19.1.5节生成的HelloWorld.zip文件吗？现在我们需要使用它了，使用Application Loader工具将其上传到App Store中。Application Loader工具是与Xcode工具一起被安装的，在Xcode 4.5中，它的位置是/Applications/Xcode.app/Contents/Applications/Application Loader.app。双击启动Application Loader，同意软件许可后，进入欢迎界面，如图19-41所示。

然后输入iTunes Connect账号和密码，点击Next按钮，进入图19-42所示的界面。



图19-41 Application Loader欢迎界面

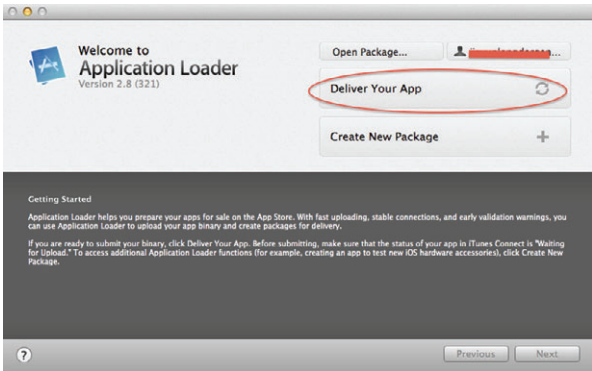


图19-42 Application Loader提交应用界面

接着点击Deliver Your App按钮，打开选择应用对话框，如图19-43所示。最后点击Next按钮，进入如图19-44所示的界面，在这里点击Choose按钮选择要上传的ZIP文件。

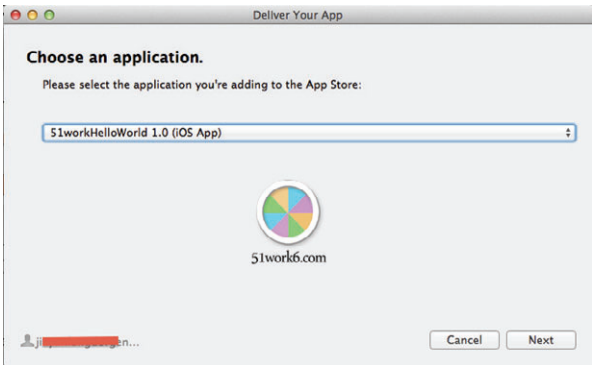


图19-43 选择应用界面

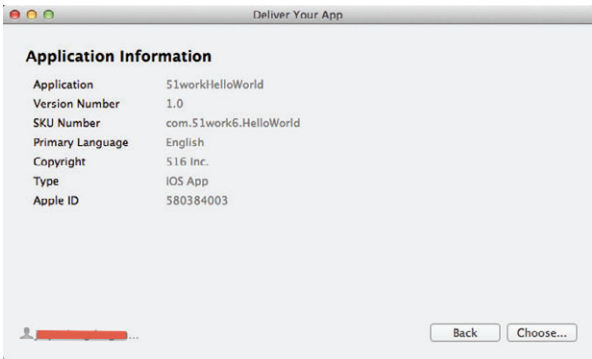


图19-44 选择上传的ZIP文件界面

选择完文件后，将打开如图19-45所示的界面，此时点击Send按钮就开始上传了。

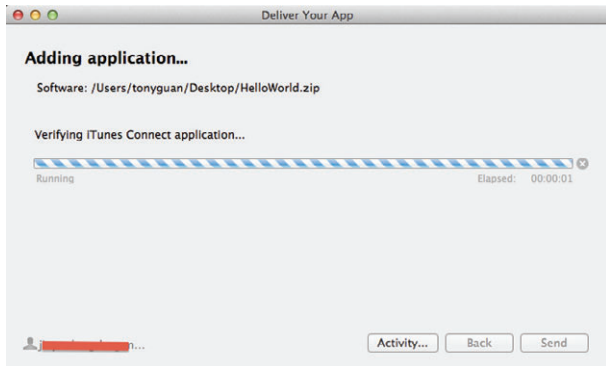


图19-45 开始上传

如果没有任何问题，接下来就是等待了。因为每天有很多程序要发布到App Store中，所以等待审核也要排队。

19.3 常见审核不通过的原因

App Store的审核是出了名的严格，相信大家也都略有耳闻。苹果官方提供了一份详细的审核指南，包括22大项、100多小项的拒绝上线条款，并且条款在不断增加中。此外，还包含一些模棱两可的条例，所以稍有“闪失”，应用就有可能被拒绝。但是有一点比较好，那就是每次遭到拒绝时，苹果会给出拒绝的理由，并指出你违反了审核指南的哪一条，开发者可以根据评审小组给的回复修改应用重新提交。下面我们讨论一下常见的被拒绝的原因。

1. 功能问题

在发布应用之前，我们一定要对产品进行认真的测试，如果在审核中出现了程序崩溃或者程序错误，无疑这是会被审核小组拒绝的。如果我们想发布一个演示版的程序，通过它给客户演示这也是不会被通过的。应用的功能与描述不相符，或者应用中含有欺诈虚假的功能，那么应用将被拒绝。比如在应用中有某个按钮，但是点击这个按钮时没有反应或者不能点击，这样的程序是不会通过的。

苹果不允许访问私有API，有浏览器的网络程序必须使用iOS WebKit框架和WebKit JavaScript。还有几点比较头痛的规则，那就是如果你的App没有什么显著的功能或者没有长久娱乐价值，也会被拒绝。如果你的应用已经在市场中已经存在了，在相关产品比较多时也可能被拒绝。

2. 用户界面问题

苹果审核指南规定开发者的应用必须遵守苹果《iOS用户界面指导原则》中解释的所有条款和条件，如果违反了这些设计原则，就会被拒绝上线，所以开发者在设计和开发产品之前一定要认真阅读《iOS用户界面指导原则》。这些原则中也渗透着苹果产品的一些理念，不仅是为了避免程序被拒绝而看，而且还为了让开发者们设计出更好的App。苹果不允许开发者更改自身按键的功能（包括声音按键以及静音按键），如果开发者使用了这些按键并利用它们做一些别的功能，将会被拒绝。

3. 商业问题

在要发布的应用中，首先不能侵犯苹果公司的商标及版权。简单地说，在应用中不能出现苹果的图标，不能使用苹果公司现在产品的类似名字为应用命名，涉及iPhone、iPad、iTunes等相关或者相近的名字都是不可以的。苹果认为这会误导用户，认为该应用是来自苹果公司的产品。误导用户认为该应用是受到苹果的肯定与认可的，也是不行的。

私自使用受保护的第三方材料（商标、版权、商业机密和其他私有内容），需要提供版权认可。如果你的应用涉及第三方版权的信息，开发者们要仔细考虑考虑了。由于有些开发者对版权法律意识比较淡薄，总会忽视这

一点，然而这一点是非常致命的。苹果处理这种被起诉的侵权应用，最轻的处罚是下架应用，有时需要将开发者账户里的钱转到起诉者账户。再严重的就是，起诉者将你告上法庭，除了自己账户中的钱被扣除外，还要另赔付起诉者相关费用。

4. 不当内容

一些不合适、不和谐的内容，苹果当然不会允许上架的。比如具有诽谤、人身攻击的应用，含有暴力倾向的应用，低俗、令人反感、厌恶的应用，赤裸裸的色情应用等。含有赌博性质的应用必须并且必须明确表示苹果不是发起者，也没有以任何方式参与活动。

5. 其他问题

关于宗教、文化或种族群体的应用或评论包含诽谤性、攻击性或自私性内容的应用不会被通过，使用第三方支付的应用会被拒绝，模仿iPod界面的应用将会被拒绝，怂恿用户造成设备损坏的应用会被拒绝。这里有个小故事，有一款应用，功能是比比谁将设备扔得高，最后算积分，这个应用始终没能上架，因为在测试应用的时候就摔坏了两部手机。此外，未获得用户同意便向用户发送推送通知，要求用户共享个人信息的应用都会被拒绝。

19.4 小结

通过对本章的学习，我们了解了如何在App Store上发布应用，发布之前需要处理哪些问题。发布者需要了解应用的发布流程，更应该熟悉应用审核不通过的一些常见原因，从而在开发时注意，以免等到审核时被拒绝，耽误了应用的上架时间。

Part 4

第四部分

实 战 篇

本 部 分 内 容

- 第 20 章 重构 MyNotes 应用——iOS 网络通信中的设计模式与架构设计
- 第 21 章 iOS 敏捷开发项目实战——2016 里约热内卢奥运会应用开发及 App Store 发布

重构MyNotes应用—— iOS网络通信中的设计 模式与架构设计

我们急于开发出能够满足用户需要的应用，往往只是关注应用“外表”的华丽，而忽略了应用“内部”的设计，这些“内部”设计就是这一章要介绍的架构设计。

好的架构设计可以提高开发效率，减少代码冗余，提高组件模块的可复用性等。好的架构设计是设计模式的有机结合，而不是设计模式的生硬堆砌。这有点像我看好莱坞大片的感觉，我关注的是影片本身的内容和艺术价值，而不是它的豪华演员阵容。

在本章中，我们将通过重构MyNotes应用来介绍网络通信架构设计及其涉及的设计模式——委托模式和观察者模式。

20.1 移动网络通信应用的分层架构设计

分层架构设计的目的是降低耦合度，提高应用的“可复用性”和“可扩展性”。图20-1是iOS平台网络通信应用的分层架构设计图。

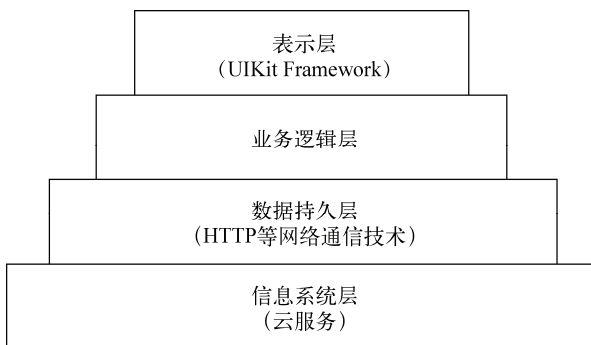


图20-1 网络通信应用的分层架构设计图

关于分层架构设计的相关内容，可以参考7.5节或者该书服务网站的相关主题。

本章要讨论的是网络应用的分层架构设计，因此数据来源是云服务而不是本地数据库。对应的数据持久层采用的是HTTP等网络通信技术，这些技术在第12章中已经介绍过了。

下面我们就分别详细介绍如何使用委托模式和观察者模式重构MyNotes应用。

20.2 基于委托模式实现

委托模式是Cocoa框架中几个常用的设计模式之一，详情可以参见3.2节或者该书服务网站<http://www.iosbook1.com>的相关主题。

20.2.1 网络通信与委托模式

委托模式类图如图20-2所示，它有两种角色：框架类和委托对象。框架类所做的是通用的与业务无关的处理工作，委托对象所做的是与业务相关的处理工作。委托对象需要实现委托协议。框架类一般都有一个委托对象的引用，在需要的时候，框架类会回调委托对象，并传递参数给委托对象。

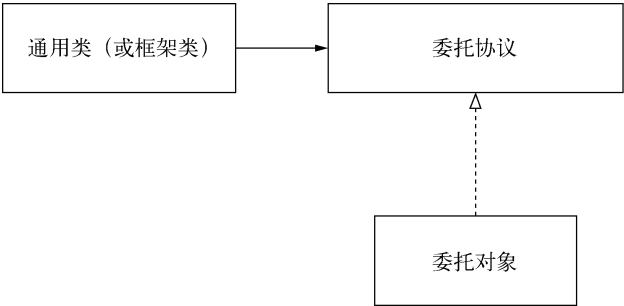


图20-2 委托模式类图

委托模式在iOS开发中应用极其广泛，下一节介绍的异步网络通信就是采用委托模式实现的。例如，iOS官方提供的 `NSURLConnection` 类和 `NSURLConnectionDelegate` 协议，以及 `ASIHTTPRequest` 框架中的 `ASIHTTPRequest` 类和 `ASIHTTPRequestDelegate` 协议，它们都采用了委托模式设计。其中作为框架类角色的 `NSURLConnection` 和 `ASIHTTPRequest`，委托协议有 `NSURLConnectionDelegate` 和 `ASIHTTPRequestDelegate` 协议对象。在请求完成或失败的情况下，框架类会调用委托对象，并传递参数给委托对象。

20.2.2 在异步网络通信中使用委托模式实现分层架构设计

由于同步网络请求的用户体验不好，因此异步网络请求是网络通信中采用的主要方式。我们需要将异步处理应用于如图20-1所示的分层设计框架中，下面简要介绍一下这个架构。

- ❑ **云服务层。**它就是分层架构中的信息系统层，是信息的来源，其数据来源于网络中的云服务。它与持久层的通信采用我们熟悉的JSON和XML格式。
- ❑ **持久层。**提供网络数据访问能力，它采用的是 `NSURLConnection` 或 `ASIHTTPRequest` 等框架，远程地、异步地调用云服务层，云服务层会将结果应答给持久层。为了能够回调业务逻辑层，它需要定义一个DAO委托协议（`DAODelegate`），而业务逻辑层需要实现DAO委托协议。
- ❑ **业务逻辑层。**它封装有一定业务处理功能的Objective-C类。为了能够回调表示层，它需要定义一个BL委托协议（`BLDelegate`），而表示层需要实现这个BL委托协议。相对业务逻辑层而言，持久层是框架类，业务逻辑层是持久层的委托对象。
- ❑ **表示层。**由UIKit Framework构成，它包括我们前面学习的视图、控制器、控件和事件处理等内容。它异步地调用业务逻辑层，业务逻辑层在结果返回之后回调表示层。相对表示层而言，业务逻辑层是框架类，表示层是业务逻辑层的委托对象。



图20-3 基于委托模式实现的分层架构设计图

20.2.3 类图

下面我们通过MyNotes应用介绍一下分层设计架构。从客户端角度看，我们只需要实现3层（表示层、业务逻辑层和持久层）即可，云服务层是服务器端要考虑的问题。下面我们分别介绍一下各层的设计类图。

1. 数据持久层

数据持久层的类图如图20-4所示。

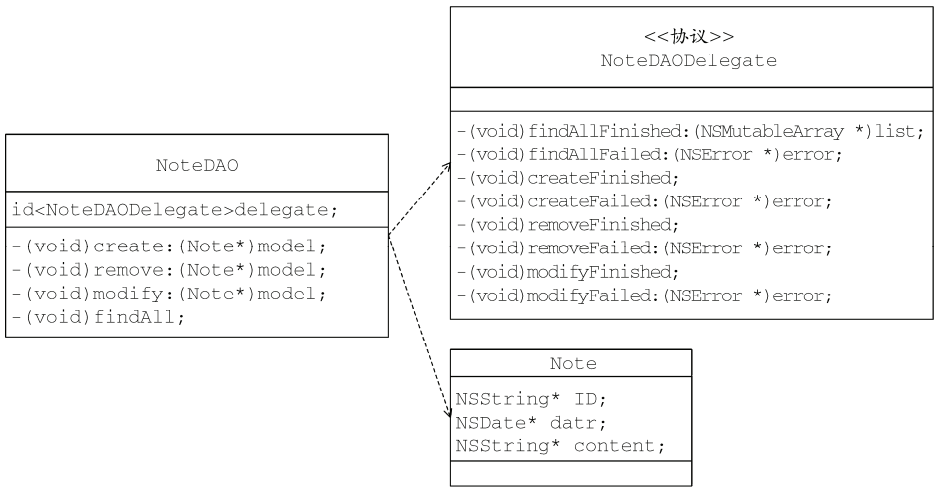


图20-4 数据持久层的类图

它由两个类和一个协议构成。Note类是实体类，实体类是应用中的“人”、“事”和“物”。NoteDAO类用于访问数据对象，它有create:、remove:、modify:和findAll:4种方法，对应于数据操作的插入、删除、修改和查询。

而NoteDAODelegate协议是委托协议，凡是异步调用持久层NoteDAO对象的其他层（如业务逻辑层）的对象都要实现该协议。这个对象就是NoteDAO委托对象，它往往是业务逻辑层的业务逻辑对象，在NoteDAO插入成功时，回调NoteDAO委托对象的findAllFinished:方法，并把查询结果回传给业务逻辑层；插入失败时，回调NoteDAO委托对象的findAllFailed:方法，并把一个错误对象回传给业务逻辑层。其他的方法还有插入成功的方法createFinished、删除成功的方法removeFinished、修改成功的方法modifFinished，这3个方法没有回传数据给表示层。而另外3个方法——插入失败的方法createFailed:、删除失败的方法removeFailed:和修改失败的方法modifyFailed:也会将一个错误对象回传给表示层。

2. 业务逻辑层

业务逻辑层的类图如图20-5所示。

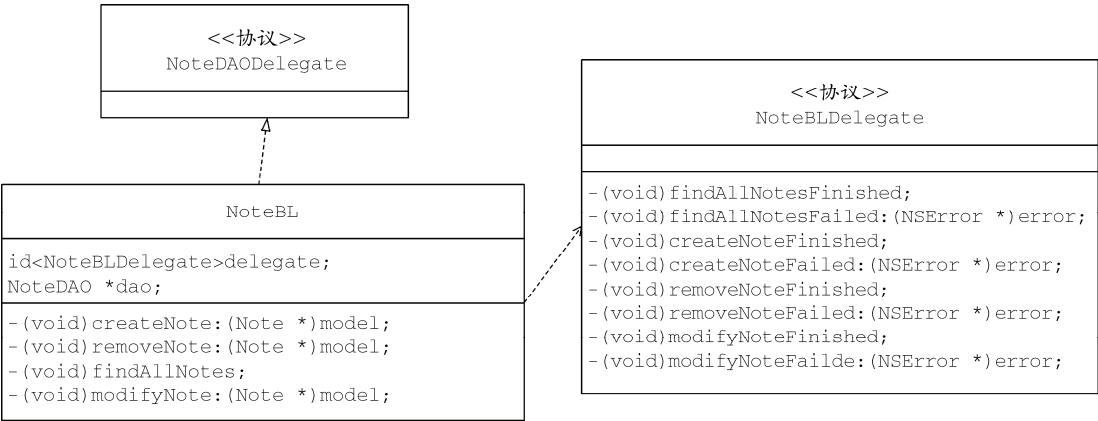


图20-5 业务逻辑层的类图

这个类图主要由NoteBL类和NoteBLDelegate协议构成，其中NoteBL类是业务逻辑类，它按照业务模块划分，其方法按照业务逻辑模块中的功能划分，每个功能对应一个公有方法。NoteBL是Note的维护模块逻辑对象，主要功能有插入备忘录、删除备忘录、修改备忘录和查询所有的备忘录。因此，它的方法有createNote:、removeNote:、modifyNote:和findAllNotes。此外，NoteBL要作为NoteDAO委托对象，需要实现NoteDAODelegate协议。

而NoteBLDelegate协议是委托协议，凡是异步调用业务逻辑层NoteBL对象的其他层（如表示层）的类都需要实现该协议。这个对象就是NoteBL委托对象，它是表示层的视图控制器对象，在NoteBL插入成功时候回调NoteBL委托对象的findAllNotesFinished:方法，并把查询结果回传表示层；插入失败情况下回调NoteBL委托对象的findAllNotesFailed:方法，并把一个错误对象回传给表示层。其他的方法还有插入成功的方法createNoteFinished、删除成功的方法removeNoteFinished和修改成功的方法modifyNoteFinished，这3个方法没有回传数据给表示层。而另外3个方法——插入失败的方法createNoteFailed:、删除失败的方法removeNoteFailed:和修改失败的方法modifyNoteFailed:也会将一个错误对象回传表示层。

3. 表示层

表示层的类图如图20-6所示。

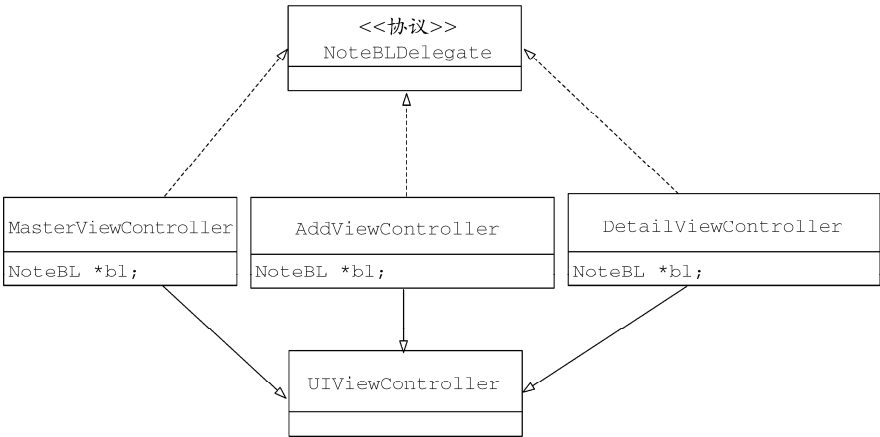


图20-6 表示层的类图

在该类图中，主要由主视图控制器类MasterViewController、添加视图控制器类AddViewController和详细视图控制器类DetailViewController构成，它们都实现了NoteBLDelegate协议。但具体实现哪些方法，根据视图控制器中的操作而定。例如，主视图控制器类MasterViewController有查询和删除操作，只需要实现findAllNotesFinished、findAllNotesFailed:、removeNoteFinished和removeNoteFailed:方法就可以了。

20.2.4 时序图

为了描述这些类的相互关系，我们介绍一下应用中的时序图。根据操作的不同，可以分为4个时序图，分别是：查询所有、插入、删除和修改。

1. 查询所有时序图

查询所有时序图如图20-7所示。

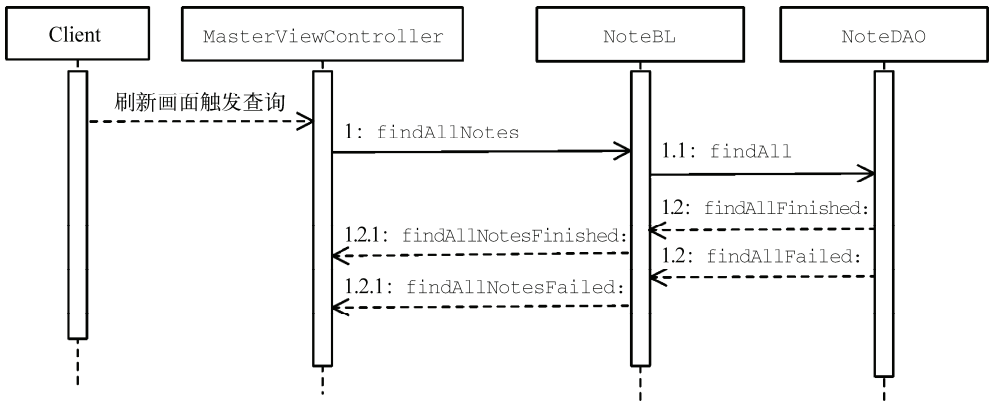


图20-7 查询所有时序图

查询所有时序图是用户在加载主视图或刷新主视图触发的查询操作。首先，MasterViewController异步调用NoteBL类中的findAllNotes方法，接着在NoteBL类中异步调用NoteDAO类的findAll方法。如果成功，我们会调用findAll方法，然后NoteDAO会回调NoteBL的findAllFinished:方法；如果失败，NoteDAO会回调NoteBL的findAllFailed:方法。在NoteBL中如果成功则回调MasterViewController的findAllNotesFinished:方法，更新表视图；失败则回调MasterViewController的findAllNotesFailed:方法，弹出警告对话框提示用户。

注意 findAllFinished:和findAllFailed:方法是二选一的，要么是调用findAllFinished:，要么调用findAllFailed:方法。同理，findAllNotesFinished:和findAllNotesFailed:方法也是二选一的，即要么调用findAllNotesFinished:方法，要么调用findAllNotesFailed:方法。

2. 插入操作时序图

插入操作时序图如图20-8所示。

插入操作时序图演示了用户进入添加界面后点击Save按钮后的处理过程。首先，在AddViewController的onclickSave:方法中异步调用NoteBL类中的createNote:方法，接着在NoteBL类中异步调用NoteDAO类的create:方法。如果成功，NoteDAO回调NoteBL的createFinished方法；如果失败，NoteDAO回调NoteBL的createFailed:方法。在NoteBL中，如果成功则回调AddViewController的createNoteFinished方法；失

败则回调AddViewController的createNoteFailed:方法，弹出警告对话框提示用户。

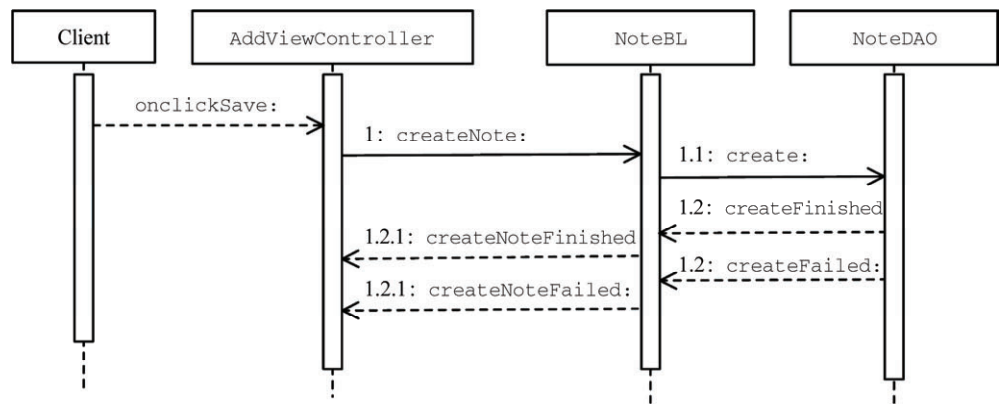


图20-8 插入操作时序图

注意 createFinished和createFailed:方法是二选一的，createNoteFinished和createNoteFailed:方法也是二选一的。

3. 删除操作时序图

删除操作是在主视图控制器完成的。当用户点击导航栏右边的Edit按钮后，得到的界面如图20-9所示，此时表视图处于编辑状态，我们点击某一条数据后面的Delete按钮即可将其删除。



图20-9 删除操作

删除操作时序图如图20-10所示。

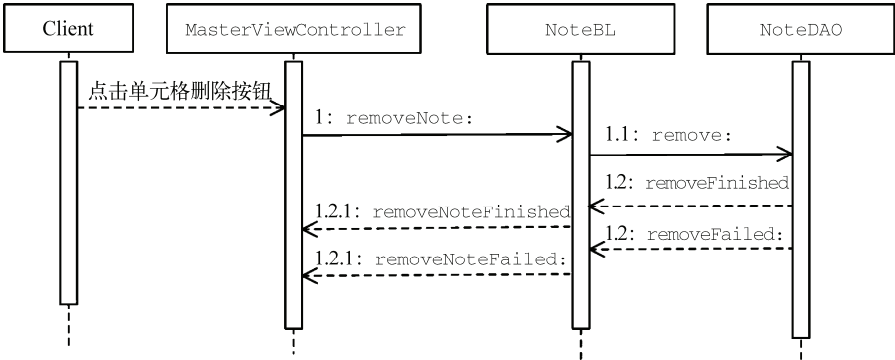


图20-10 删除操作时序图

删除操作时序图演示了在用户主视图界面点击Delete按钮后的处理过程。它是在MasterViewController中tableView:commitEditingStyle:forRowAtIndexPath:方法（实现表视图数据源的方法）中发起的，异步调用NoteBL类中的removeNote:方法，接着在NoteBL类中异步调用NoteDAO类的remove:方法。如果删除成功，NoteDAO回调NoteBL的removeFinished方法；如果删除失败，NoteDAO回调NoteBL的removeFailed:方法。在NoteBL中，如果成功则回调MasterViewController的removeNoteFinished方法；失败则回调MasterViewController的removeNoteFailed:方法，弹出警告对话框提示用户。

注意 removeFinished和removeFailed:方法是二选一的，removeNoteFinished和removeNoteFailed:方法也是二选一的。

4. 修改操作时序图

修改操作时序图如图20-11所示。

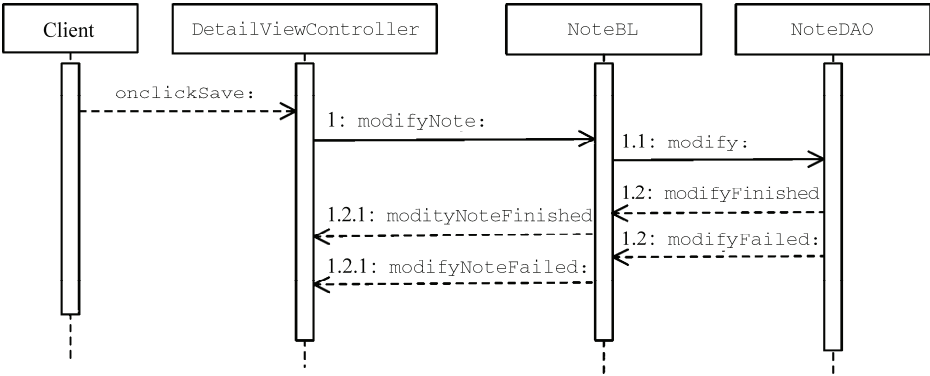


图20-11 修改操作时序图

修改操作时序图演示了用户在进入详细界面后点击Save按钮的处理过程。首先，DetailViewController的onclickSave:方法异步调用NoteBL类中的modifyNote:方法，接着在NoteBL类中调用NoteDAO类的modify:方法。如果成功，NoteDAO回调NoteBL的modifyFinished方法；如果失败，NoteDAO回调NoteBL的modifyFailed:方法。在NoteBL中如果成功则回调DetailViewController的modifyNoteFinished方法；失败则回调AddViewController的modifyNoteFailed:方法，弹出警告对话框提示用户。

注意 modifyFinished和modifyFailed:方法是二选一的, modifyNoteFinished和modifyNoteFailed:方法也是二选一的。

20.2.5 数据持久层的代码实现

原理我们已经介绍了很多,在接下来的3节中,我们介绍一下主要的代码实现部分,但UI部分的代码不再介绍。下面我们按照图20-3从下往上介绍。

在数据持久层中,共有3个类,其中主要的是NoteDAO类。下面我们看看NoteDAO.h部分的代码,具体如下:

```
#import "NoteDAODelegate.h"
#import "Note.h"
#import "ASIFormDataRequest.h"
#import "NSString+URLEncoding.h"
#import "NSNumber+Message.h"

#define BASE_URL @"http://iosbook1.com/service/mynotes/webservice.php"

@interface NoteDAO : NSObject

//保存数据列表
@property (nonatomic, strong) NSMutableArray* listData;

@property (weak, nonatomic) id <NoteDAODelegate> delegate; ①

//插入备忘录的方法
-(void) create:(Note*)model;
//删除备忘录的方法
-(void) remove:(Note*)model;
//修改备忘录的方法
-(void) modify:(Note*)model;
//查询所有数据的方法
-(void) findAll;

@end
```

其中第①行代码声明了delegate属性,它是用来保存委托对象弱引用的属性。id <NoteDAODelegate>说明必须是NoteDAODelegate协议的实现类。

在NoteDAO.m中,查询所有数据方法的代码如下:

```
//查询所有数据的方法
-(void) findAll
{
    NSURL *url = [NSURL URLWithString:[BASE_URL URLEncodedString]];

    __weak ASIFormDataRequest *request = [ASIFormDataRequest requestWithURL:url]; ①

    [request setPostValue:@"<你的iosbook1.com用户邮箱>" forKey:@"email"];
    [request setPostValue:@"JSON" forKey:@"type"];
    [request setPostValue:@"query" forKey:@"action"]; ②

    [request setCompletionBlock:^( ③

        NSData *data = [request responseData];
        NSDictionary *resDict = [NSJSONSerialization JSONObjectWithData:data
                                options:NSJSONReadingAllowFragments error:nil];
        NSNumber *resultCodeNumber = [resDict objectForKey:@"ResultCode"];
        if ([resultCodeNumber integerValue] >= 0) ④
        {
            NSMutableArray* listDict = [resDict objectForKey:@"Record"];
            NSMutableArray *listData = [NSMutableArray new];
            for (NSDictionary* dic in listDict) { ⑤
```



```

        Note *note = [Note new];
        note.ID = [dic objectForKey:@"ID"];
        note.date = [dic objectForKey:@"CDate"];
        note.content = [dic objectForKey:@"Content"];
        [listData addObject:note];
    }
    [self.delegate findAllFinished:listData];
} else {
    NSInteger resultCode = [resultCodeNumber integerValue];
    NSNumber *resultCodeNumber = [NSNumber numberWithInt:resultCode];
    NSString* message = [resultCodeNumber errorMessage];
    NSDictionary *userInfo = [NSDictionary dictionaryWithObject:message
        forKey:NSLocalizedDescriptionKey];
    NSError *err = [NSError errorWithDomain:@"DAO"
        code:resultCode userInfo:userInfo];
    [self.delegate findAllFailed:err];
}
}
[request setFailedBlock:^(
    NSError *error = [request error];
    [self.delegate findAllFailed:error];
)];
[request startAsynchronous];
}

```

在应用中，我们采用ASIHTTPRequest框架实现网络请求，而且在没有特殊情况下，都采用异步POST方法提交请求。为了使代码更简洁，我们采用了块代码方式。第①行代码定义ASIFormDataRequest对象。第①~②行代码是准备POST参数，第③行代码至第⑨行代码之前都是请求成功代码块，第⑨~⑩行是请求失败代码块，其中我们通过[self.delegate findAllFailed:err]语句回调委托对象的findAllFailed:方法，并不将错误对象返回给委托对象。

第④行代码从返回的结果取出ResultCode。当ResultCode大于等于0，则代表成功，如第⑤代码所示。第⑥行代码回调委托对象的findAllFinished:方法，并把数据listData返回给委托对象。

在ResultCode小于0的情况下，即从服务器返回失败的情况下，第⑦~⑧行代码创建一个NSError错误代码，因此服务器返回的是错误代码，通过分类NSNumber (Message)中的errorMessage方法获得错误消息。然后通过[self.delegate findAllFailed:err]语句回调委托对象的findAllFailed:方法，并把错误对象返回给委托对象。

除了查询所有方法，在NoteDAO中还有插入、删除和修改方法。这3个方法处理起来都非常类似，这里我们只介绍插入方法，其他的方法请大家自己下载代码查看。

NoteDAO.m中插入方法的代码如下：

```

- (void) create: (Note*) model
{
    NSURL *url = [NSURL URLWithString:[BASE_URL URLEncodedString]];
    __weak ASIFormDataRequest *request = [ASIFormDataRequest requestWithURL:url];

    [request setPostValue:@"<你的iosbook1.com用户邮箱>" forKey:@"email"];
    [request setPostValue:@"JSON" forKey:@"type"];
    [request setPostValue:@"add" forKey:@"action"];
    [request setPostValue:model.date forKey:@"date"];
    [request setPostValue:model.content forKey:@"content"];

    [request setCompletionBlock:^(
        NSData *data = [request responseData];
        NSDictionary *resDict = [NSJSONSerialization JSONObjectWithData:data
            options:NSJSONReadingAllowFragments error:nil];
        NSNumber *resultCodeNumber = [resDict objectForKey:@"ResultCode"];
        if ([resultCodeNumber integerValue] >= 0)
        {

```

```

        [self.delegate createFinished];
    } else {
        NSInteger resultCode = [resultCodeNumber integerValue];
        NSNumber *resultCodeNumber = [NSNumber numberWithInt:resultCode];
        NSString* message = [resultCodeNumber errorMessage];
        NSDictionary *userInfo = [NSDictionary dictionaryWithObject:message
            forKey:NSLocalizedDescriptionKey];
        NSError *err = [NSError errorWithDomain:@"DAO"
            code:resultCode userInfo:userInfo];

        [self.delegate createFailed:err];
    }
}];
[request setFailedBlock:^(
    NSError *error = [request error];
    [self.delegate createFailed:error];
)];
[request startAsynchronous];
}

```

比较一下插入方法与查询所有方法，代码基本上类似，差别就是POST参数和返回后回调的方法不同。另外的两个方法（删除和修改）也是比较类似的，处理步骤也比较类似，因此，我们可以采用模板方法设计模式^①对NoteDAO类进行优化，但这与本章重点介绍的分层框架设计关系不是很大，因此基于模板方法设计模式的优化我们就不再介绍了。第①行代码表示成功返回的情况下，回调委托对象的createFinished方法，而第②行代码和第③行代码表示失败返回的情况下，回调委托对象的createFailed:方法。

20.2.6 业务逻辑层的代码实现

在这个应用中，业务逻辑层中用到的类主要就是NoteBL类。下面我们看看NoteBL.h部分的代码，具体如下：

```

#import "Note.h"
#import "NoteDAO.h"
#import "NoteDAODelegate.h"
#import "NoteBLDelegate.h"

@interface NoteBL : NSObject <NoteDAODelegate>
@property (weak, nonatomic) id <NoteBLDelegate> delegate;
@property (strong, nonatomic) NoteDAO *dao;

//插入备忘录的方法
-(void) createNote:(Note*)model;
//删除备忘录的方法
-(void) removeNote:(Note*)model;
//查询所有数据的方法
-(void) findAllNotes;
//修改数据的方法
-(void) modifyNote:(Note*)model;

@end

```

在第①行代码中，我们声明NoteBL实现委托协议NoteDAODelegate；第②行代码声明的delegate属性用来保存委托对象弱引用，其中id <NoteBLDelegate>说明必须是NoteBLDelegate协议的实现类。第③行代码声明的dao属性用来保存NoteDAO对象强引用，这样可以防止在多个方法的使用过程中释放NoteDAO对象。

注意，我们在NoteDAO.h中也定义了delegate属性，它也是用来保存委托对象弱引用的。NoteDAO与NoteBL关系的比较复杂，如图20-12所示。NoteBL中的dao属性引用NoteDAO类，而NoteDAO类中的delegate引用到NoteBL类，它们构成一种循环引用关系。

① 详见<http://zh.wikipedia.org/wiki/模板方法>。

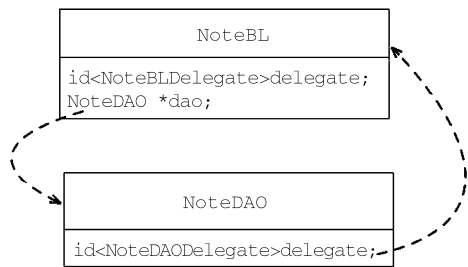


图20-12 循环引用关系

在这个循环引用关系中，它们之间不能都是强引用，否则会引起内存的循环保持问题。为了防止这类问题，我们在定义委托对象属性delegate时采用了弱引用方式。

在NoteBL.m中，我们只列出了主要的几个方法，具体如下所示：

```
-(id) init {
    self = [super init];
    if (self) {
        _dao = [NoteDAO new];
    }
    return self;
}
//插入备忘录的方法
-(void) createNote:(Note*)model {
    _dao.delegate = self;
    [_dao create:model];
}
//查询所有数据的方法
-(void) findAllNotes {
    _dao.delegate = self;
    [_dao findAll];
}
#pragma --mark NoteDAODelegate委托方法
//成功查询所有数据后调用的方法
- (void)findAllFinished:(NSMutableArray *)list {
    [_delegate findAllNotesFinished:list];
}
//查询所有数据失败后调用的方法
- (void)findAllFailed:(NSError *)error {
    [_delegate findAllNotesFailed:error];
}
//成功插入备忘录后调用的方法
- (void)createFinished {
    [_delegate createNoteFinished];
}
//插入备忘录失败后调用的方法
- (void)createFailed:(NSError *)error {
    [_delegate createNoteFailed:error];
}
```

在上述代码中，第①行是构造方法，在其中初始化成员变量_dao，它是NoteDao类型；第③行代码是插入方法；第④行代码设置_dao的委托对象为self，使得当前的业务逻辑类NoteBL成为NoteDao类的委托对象，从而建立了引用关系；第⑤行与第④行是同样目的。第⑦~⑧行是回调方法，其中均使用_delegate回调NoteBL委托对象的相关方法。

20.2.7 表示层的代码实现

表示层用到的类主要有3个——MasterViewController、AddViewController和DetailViewController，下面我们分别介绍一下它们。

1. MasterViewController类

MasterViewController.h部分的代码如下：

```

@interface MasterViewController : UITableViewController <NoteBLDelegate> ①

@property (strong, nonatomic) DetailViewController *detailViewController;
//保存数据列表
@property (nonatomic, strong) NSMutableArray* listData;
//删除数据索引
@property (nonatomic, assign) NSUInteger deletedIndex;
//删除数据
@property (nonatomic, strong) Note *deletedNote;
//bl对象
@property (nonatomic, strong) NoteBL *bl; ②

@end

```

其中第①行代码声明MasterViewController实现委托协议NoteBLDelegate，第②代码声明的bl属性用来保存NoteBL对象强引用。

在MasterViewController.m中，查询请求的主要代码如下：

```

- (void)viewDidLoad {
    .....
    _bl = [NoteBL new]; ①
    _bl.delegate = self; ②
    [_bl findAllNotes]; ③

    //初始化UIRefreshControl
    UIRefreshControl *rc = [[UIRefreshControl alloc] init]; ④
    rc.attributedTitle = [[NSAttributedString alloc] initWithString:@"下拉刷新"];
    [rc addTarget:self action:@selector(refreshTableView)
      forControlEvents:UIControlEventValueChanged];
    self.refreshControl = rc;
}
- (void) refreshTableView ⑤
{
    if (self.refreshControl.refreshing) {
        self.refreshControl.attributedTitle = [[NSAttributedString alloc]
        initWithString:@"加载中..."];
        _bl.delegate = self; ⑥
        [_bl findAllNotes]; ⑦
    }
}
#pragma mark - 处理通知
//成功查询所有数据后调用的方法
- (void)findAllNotesFinished:(NSMutableArray *)list { ⑧
    self.listData = list;
    [self.tableView reloadData];
    if (self.refreshControl) {
        [self.refreshControl endRefreshing];
        self.refreshControl.attributedTitle = [[NSAttributedString
        alloc] initWithString:@"下拉刷新"];
    }
}
//查询所有数据失败后调用的方法
- (void)findAllNotesFailed:(NSError*)error { ⑨
    NSString *errorStr = [error localizedDescription];
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"操作信息"
        message:errorStr
        delegate:nil
        cancelButtonTitle:@"OK"
        otherButtonTitles: nil];

    [alertView show];
    if (self.refreshControl) {
        [self.refreshControl endRefreshing];
    }
}

```

```

        self.refreshControl.attributedTitle = [[NSAttributedString
            alloc] initWithString:@"下拉刷新"];
    }
}

```

在viewDidLoad方法中，第①行代码声明了NoteBL对象，第②行代码设置_b1的委托对象为self，第③行代码调用业务逻辑层的findAllNotes方法。

第④行代码实例化刷新控件UIRefreshControl。在刷新表视图时，我们会调用第⑤行的refreshTableView方法。在refreshTableView方法中，我们也会调用业务逻辑层的findAllNotes方法，如第⑦行代码所示。

第⑧行代码为成功查询所有数据后调用的方法。在该方法中，我们将返回的数据绑定到表视图上。第⑨行代码为查询所有数据失败之后调用的方法，这里会弹出一个警告框给用户提示。

下面我们再看看删除相关的代码。删除操作也是在主视图控制器MasterViewController中实现的，具体代码如下：

```

- (void)tableView:(UITableView *)tableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
    forRowAtIndexPath:(NSIndexPath *)indexPath                                ①
{
    if (editingStyle == UITableViewCellEditingStyleDelete) {                    ②
        _deletedIndex = [indexPath row];
        _deletedNote = [_listData objectAtIndex:_deletedIndex];

        _bl.delegate = self;                                                    ③
        [_bl removeNote: _deletedNote];                                          ④

        [self.listData removeObject:_deletedNote];                              ⑤
        [tableView deleteRowsAtIndexPaths:@[indexPath]
            withRowAnimation:UITableViewRowAnimationFade];

    } else if (editingStyle == UITableViewCellEditingStyleInsert) {
    }
}
//成功删除备忘录后调用的方法
- (void)removeNoteFinished {
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"操作信息"
        message:@"删除成功。"
        delegate:nil
        cancelButtonTitle:@"OK"
        otherButtonTitles: nil];

    [alertView show];
}

//删除备忘录失败后调用的方法
- (void)removeNoteFailed:(NSError *)error {
    NSString *errorStr = [error localizedDescription];
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"操作信息"
        message:errorStr
        delegate:nil
        cancelButtonTitle:@"OK"
        otherButtonTitles: nil];

    [alertView show];

    [self.listData insertObject:_deletedNote atIndex:_deletedIndex];
    [self.tableView reloadData];
}

```

删除操作是在表视图数据源的tableView:commitEditingStyle:forRowAtIndexPath:方法中实现的。第②行代码判断当前的样式是否是删除样式UITableViewCellEditingStyleDelete。除了删除样式外，还有插入样式UITableViewCellEditingStyleInsert。第③行代码设置_b1的委托对象为self。第④行代码调用业务逻辑层对象进行删除处理。

2. AddViewController类

AddViewController.h部分的代码如下：

```
@interface AddViewController : UIViewController
    <UITextViewDelegate, NoteBLDelegate> ①

@property (weak, nonatomic) IBOutlet UITextView *textView;

//bl对象
@property (nonatomic, strong) NoteBL *bl; ②

- (IBAction)onclickDone:(id)sender;
- (IBAction)onclickSave:(id)sender;

@end
```

在上述代码中，第①行代码声明AddViewController实现委托协议NoteBLDelegate，第②代码声明的bl属性用来保存NoteBL对象强引用。

在AddViewController.m中，插入请求的主要代码如下：

```
- (IBAction)onclickSave:(id)sender { ①
    if (!_bl) {
        _bl = [NoteBL new];
        _bl.delegate = self;
    }
    Note *note = [Note new];
    note.date = [[NSDate alloc] init];
    note.content = _textView.text;
    [_bl createNote: note];
}
//成功插入备忘录后调用的方法
- (void)createNoteFinished { ②
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"操作信息"
                                                            message:@"插入成功。"
                                                            delegate:self
                                                            cancelButtonTitle:@"返回"
                                                            otherButtonTitles:@"继续", nil]; ③

    [alertView show];
}
//插入备忘录失败后调用的方法
- (void)createNoteFailed:(NSError *)error { ④

    NSString *errorStr = [error localizedDescription];
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"操作信息"
                                                            message:errorStr
                                                            delegate:self
                                                            cancelButtonTitle:@"返回"
                                                            otherButtonTitles:@"继续", nil]; ⑤

    [alertView show];
}
//响应对话框按钮事件
- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:
    (NSInteger)buttonIndex { ⑥
    if (buttonIndex == 0) { //选择返回按钮 ⑦
        [_textView resignFirstResponder];
        [self dismissViewControllerAnimated:YES completion:nil];
    }
}
```

当用户点击Save按钮时，会触发onclickSave:方法。在该方法中，我们实例化业务逻辑对象，设置它的委托对象为self，并异步调用createNote:方法实现数据的插入。如果插入成功，则回调第②行的createNoteFinished:方法。对于成功返回的处理方式，插入成功与查询成功不同，只需要给用户一个反馈信

息就可以了,具体如图20-13所示。如果用户点击“返回”按钮,则回到主视图,如果点击“继续”按钮,则留在本视图,继续添加内容。与只有一个按钮的AlertView不同,这种有两个按钮的AlertView需要设置委托为self。第③行代码在构造AlertView时设置了delegate:self参数,这样当用户选择AlertView中的按钮时,就会回调第⑥行的alertView:clickedButtonAtIndex:委托方法,其中buttonIndex参数可以判断是点击了哪个按钮的索引。在返回失败的情况下也做类似处理,如第⑤行代码所示。

3. DetailViewController类

DetailViewController.h部分的代码如下:

```
@interface DetailViewController : UIViewController
    <UISplitViewControllerDelegate, NoteBLDelegate> ①

@property (weak, nonatomic) IBOutlet UITextView *textView;
@property (strong, nonatomic) id detailItem;
//bl对象
@property (nonatomic, strong) NoteBL *bl; ②
//接收从服务器返回的数据
@property (strong, nonatomic) NSMutableData *datas;
- (IBAction)onclickSave:(id)sender;

@end
```

其中第①行代码声明DetailViewController实现委托协议NoteBLDelegate,第②代码声明的bl属性用来保存NoteBL对象强引用。

在DetailViewController.m中,修改请求的主要代码如下:

```
- (IBAction)onclickSave:(id)sender {
    if (!_bl) {
        _bl = [NoteBL new];
        _bl.delegate = self;
    }
    Note *note = _detailItem;
    note.date = [[NSDate alloc] init];
    note.content = _textView.text;
    [_bl modifyNote:note];

    [_textView resignFirstResponder];
}

//成功插入备忘录后回调的方法
- (void)modifyNoteFinished {
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"操作信息"
                                                            message:@"修改成功。"
                                                            delegate:self
                                                            cancelButtonTitle:@"返回"
                                                            otherButtonTitles:@"继续", nil];

    [alertView show];
}

//插入备忘录失败后回调的方法
- (void)modifyNoteFailed:(NSError *)error {
    NSString *errorStr = [error localizedDescription];
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"操作信息"
```



图20-13 信息提示框

```
message:errorStr
delegate:self
cancelButtonTitle:@"返回"
otherButtonTitles:@"继续", nil];

[alertView show];
}
//响应对话框按钮事件
- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:
(NSInteger)buttonIndex {
if (buttonIndex == 0) { //选择返回按钮
//返回
UINavigationController *navController =
(UINavigationController*)self.parentViewController;
[navController popViewControllerAnimated:YES];
}
}
```

上面的修改代码与插入代码非常相似，这里我们只把代码列出来，不再介绍了。

20.3 基于观察者模式的通知机制实现

观察者模式是Cocoa框架中几个常用的设计模式之一，详情可参考3.3节或者本书服务网站<http://www.iosbook1.com>的相关主题。

20.3.1 观察者模式的通知机制回顾

观察者模式的具体应用有两个——通知（notification）机制和KVO（Key-Value Observing，键值观察）机制，这里我们重点使用通知机制。通知机制与委托机制不同的地方是：通知是一对多的对象之间的通信，而委托是一对一的对象之间的通信。

如图20-14所示，在通知机制中，对某个通知感兴趣的所有对象都可以成为接收者。首先，这些对象需要向通知中心（NSNotificationCenter）发出addObserver:selector:name:object:消息进行注册，在投送对象投送通知给通知中心时，通知中心就会把通知广播给注册过的接收者。所有的接收者都不知道通知是谁投送的，更不关心它的细节。投送对象与接收者是一对多的关系。如果接收者不再关注通知，会给通知中心发出removeObserver:name:object:消息解除注册，以后不再接收通知。

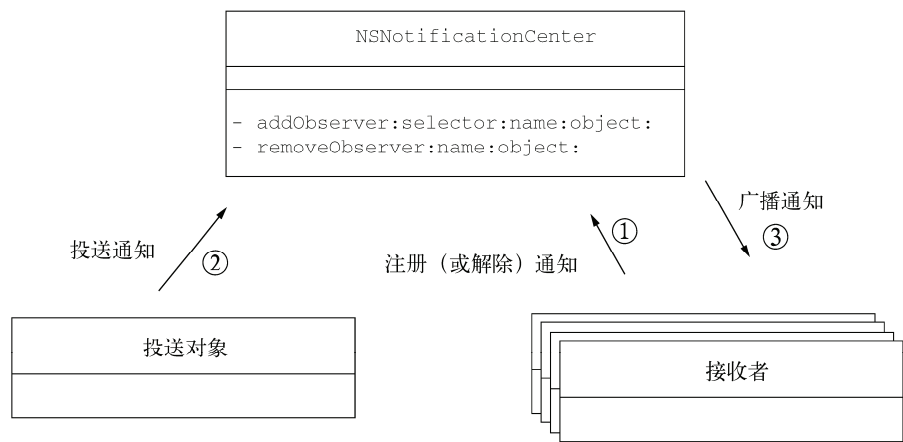


图20-14 通知机制图

20.3.2 异步网络通信中通知机制的分层架构设计

在上一节中，我们介绍了观察者设计模式下的通知机制。此外，通知机制也可以用于异步网络请求的参数传递，这里我们需要将异步处理应用于图20-15所示的分层设计框架中。

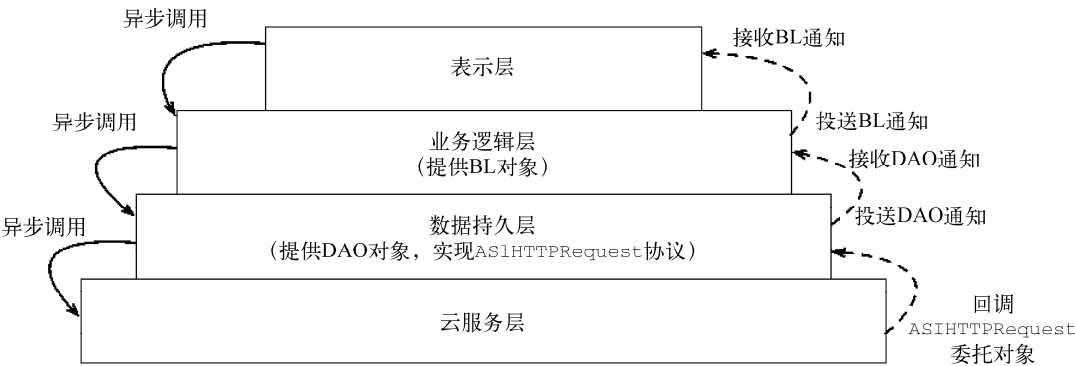


图20-15 基于通知机制的分层架构设计图

- ❑ 数据持久层。为了能够将数据返回给业务逻辑层，它需要投送一个DAO通知，而这个DAO通知必须是在业务逻辑层注册的。
- ❑ 业务逻辑层。为了接收持久层的通知，业务逻辑层需要注册并接收DAO通知。为了能够将数据返回给表示层，它需要投送一个BL通知。
- ❑ 表示层。为了接收业务逻辑层的通知，表示层需要注册并接收BL通知。

20.3.3 类图

在通知机制下MyNotes实现的分层设计架构中，也是只需要实现3层（表示层、业务逻辑层和数据持久层），下面我们分别介绍一下各层的设计类图。

1. 数据持久层

数据持久层的类图如图20-16所示。

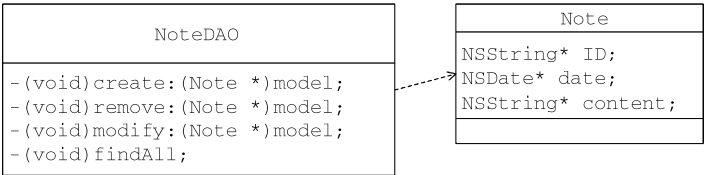


图20-16 数据持久层的类图

它由两个类构成，包括实体类Note和数据访问对象NoteDAO，其中NoteDAO有create:、remove:、modify:和findAll4个方法。

2. 业务逻辑层

业务逻辑层的类图如图20-17所示。

业务逻辑层只有一个NoteBL类，它有12个方法，其中前4个数据处理方法是插入、删除、修改和查询方法，而后8个方法是接收持久层通知的方法。每个数据处理方法都有两个接收方法与之对应，即一个成功，一个失败。例如，与数据插入方法createNote:对应的成功接收方法为createFinished，与它对应的失败接收方法为createFailed:。

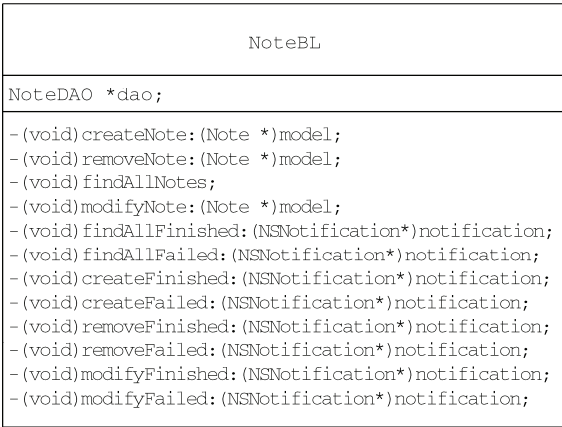


图20-17 业务逻辑层的类图

3. 表示层

表示层的类图如图20-18所示。

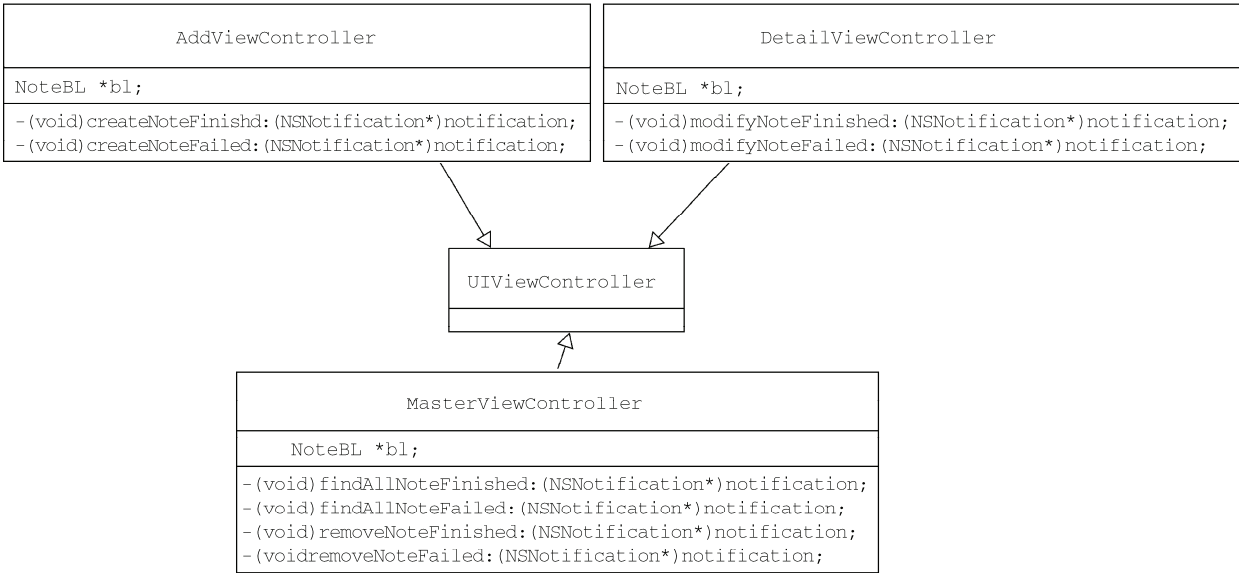


图20-18 表示层的类图

该类图主要由主视图控制器类MasterViewController、添加视图控制器类AddViewController和详细视图控制器类DetailViewController构成，其中每个类中都有几个接收业务逻辑层通知的方法。

20.3.4 时序图

为了能够描述这些类的动态行为，我们介绍一下应用中的时序图。根据操作的不同，可以划分为4个时序图，分别是查询所有、插入、删除和修改。

1. 查询所有时序图

查询所有时序图如图20-19所示。

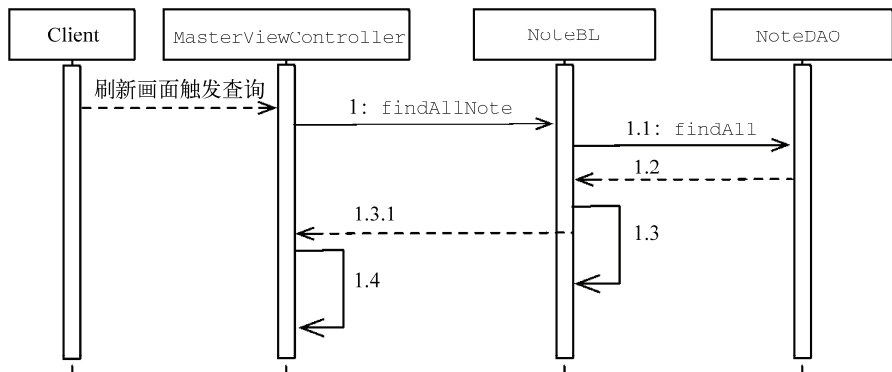


图20-19 查询所有时序图

查询所有时序图演示了用户在加载主视图或刷新主视图时触发的查询操作。首先，MasterViewController 异步调用 NoteBL 类中的 findAllNotes 方法，接着在 NoteBL 类中异步调用 NoteDAO 类的 findAll 方法，如果成功，NoteDAO 会投送 DaoFindAllFinishedNotification 通知，如图中的 1.2 消息所示。NoteBL 负责注册和观察 DaoFindAllFinishedNotification 通知，一旦 DaoFindAllFinishedNotification 通知被广播，它就会调用自身的查询成功方法 findAllFinished:，如图中的 1.3 所示消息。NoteBL 的 findAllFinished: 方法负责投送 BLFindAllFinishedNotification 成功通知给表示层，如图中 1.3.1 消息所示。表示层的 MasterViewController 负责注册和观察 BLFindAllFinishedNotification 通知，一旦有 BLFindAllFinishedNotification 通知被广播，它将调用自身的查询成功方法 findAllNotesFinished:，并在这个方法中更新表视图。

注意图中 1.2~1.4 的消息，除了成功返回，还有失败返回，它们返回的顺序是完全一样的，但是是二选一的。失败情况下，NoteDAO 投送 DaoFindAllFailedNotification 通知，NoteBL 接收到通知后，会触发自身的 findAllFailed: 方法。在 NoteBL 的 findAllFailed: 方法中，继续投送 BLFindAllFailedNotification 通知。MasterViewController 接收到 BLFindAllFailedNotification 通知后，会触发自身的 findAllNotesFailed:。在这个方法中，会给用户一些提示信息。

2. 插入操作时序图

插入操作时序图如图 20-20 所示。

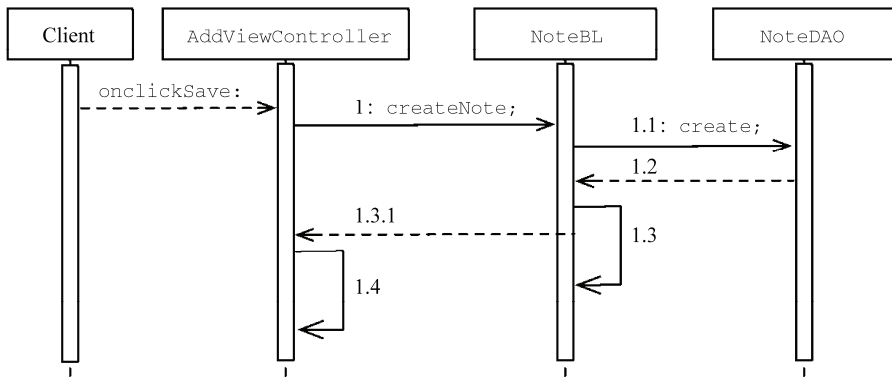


图20-20 插入操作时序图

插入操作时序图演示了用户进入添加界面后点击Save按钮的处理过程。首先，AddViewController的onclickSave:方法在异步调用NoteBL类中的createNote:方法，接着在NoteBL类中异步调用NoteDAO类的create:方法，返回结果成功或失败，都是按照图中1.2~1.4消息进行处理的。在插入操作时序图中所使用的通知如下。

- ❑ **DaoCreateFinishedNotification**。DAO插入数据成功通知，如图中1.2消息所示。
- ❑ **DaoCreateFailedNotification**。DAO插入数据失败通知，如图中1.2消息所示。
- ❑ **BLCreateNoteFinishedNotification**。BL插入数据成功通知，如图中1.3.1消息所示。
- ❑ **BLCreateNoteFailedNotification**。BL插入数据失败通知，如图中1.3.1消息所示。

还有图中1.3消息在插入数据成功的情况下，调用的是createFinished方法，失败的情况下调用的是createFailed:方法。图中1.4消息在插入数据成功的情况下调用的是createNoteFinished方法，失败的情况下调用的是createNoteFailed:方法。

3. 删除操作时序图

删除操作时序图如图20-21所示。

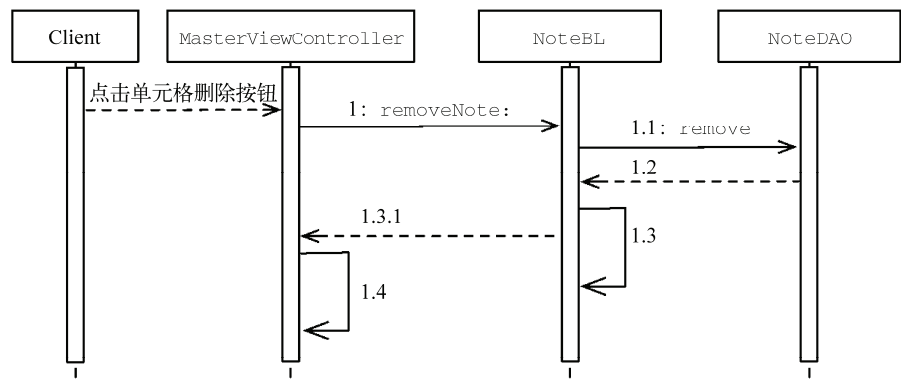


图20-21 删除操作时序图

删除操作时序图演示了用户再主视图界面点击删除按钮后的处理过程。在MasterViewController中的tableView:commitEditingStyle:forRowAtIndexPath:方法中，异步调用NoteBL类中的removeNote:方法，接着在NoteBL类中异步调用NoteDAO类的remove:方法。无论返回结果成功或者失败，都是按照图中1.2~1.4消息进行处理的。在删除操作时序图中所使用的通知如下。

- ❑ **DaoRemoveFinishedNotification**。DAO删除数据成功通知，如图中1.2消息所示。
- ❑ **DaoRemoveFailedNotification**。DAO删除数据失败通知，如图中1.2消息所示。
- ❑ **BLRemoveFinishedNotification**。BL删除数据成功通知，如图中1.3.1消息所示。
- ❑ **BLRemoveNoteFailedNotification**。BL删除数据失败通知，如图中1.3.1消息所示。

还有图中1.3消息在插入数据成功的情况下调用的是removeFinished方法，失败情况下调用的是removeFailed:方法。图中1.4消息在插入数据成功的情况下调用的是removeNoteFinished方法，失败的情况下调用的是removeNoteFailed:方法。

4. 修改操作时序图

修改操作时序图如图20-22所示。

修改操作时序图演示了用户在进入详细界面后，点击Save按钮后的处理过程。首先，我们在Detail-ViewController的onclickSave:方法中异步调用NoteBL类中的modifyNote:方法，接着在NoteBL类中异步

调用NoteDAO类的modify:方法。返回结果成功和失败，都是按照图中1.2~1.4消息，除了成功返回，这个过程与插入类似。在修改操作时序图中所使用的通知如下。

- ❑ **DaoModifyFinishedNotification**。DAO修改数据成功通知，如图中1.2消息所示。
- ❑ **DaoModifyFailedNotification**。DAO修改数据失败通知，如图中1.2消息所示。
- ❑ **BLModifyFinishedNotification**。BL修改数据成功通知，如图中1.3.1消息所示。
- ❑ **BLModifyNoteFailedNotification**。BL修改数据失败通知，如图中1.3.1消息所示。

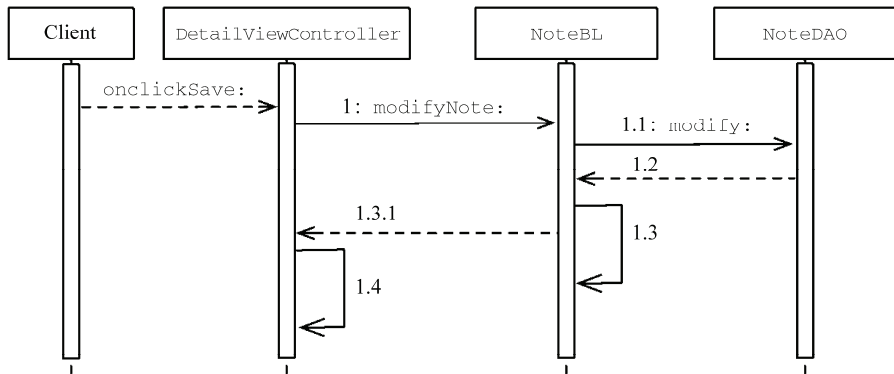


图20-22 修改操作时序图

还有图中1.3消息在插入数据成功的情况下调用的是modifyFinished方法，失败的情况下调用的是modifyFailed:方法。图中1.4消息在插入数据成功的情况下调用的是modifyNoteFinished方法，失败的情况下调用的是modifyNoteFailed:方法。

20.3.5 数据持久层的代码实现

在接下来的3节中，我们介绍一下主要的代码实现部分，但UI部分的代码我们不再介绍。我们按照图20-15所示从下往上介绍。

在数据持久层中有两个类，其中主要的是NoteDAO类。NoteDAO.h部分的代码如下：

```

#define BASE_URL @"http://iosbook1.com/service/mynotes/webservice.php"
@interface NoteDAO : NSObject

//保存数据列表
@property (nonatomic, strong) NSMutableArray* listData;

//插入备忘录的方法
-(void) create:(Note*)model;
//删除备忘录的方法
-(void) remove:(Note*)model;
//修改备忘录的方法
-(void) modify:(Note*)model;
//查询所有数据的方法
-(void) findAll;
@end

```

NoteDAO.m中查询所有数据的代码如下：

```

-(void) findAll {
    NSURL *url = [NSURL URLWithString:[BASE_URL URLEncodedString]];

    __weak ASIFormDataRequest *request = [ASIFormDataRequest requestWithURL:url];

```

```

[request setPostValue:@" " forKey:@"email"];
[request setPostValue:@"JSON" forKey:@"type"];
[request setPostValue:@"query" forKey:@"action"];

[request setCompletionBlock:^(

    NSData *data = [request responseData];
    NSDictionary *resDict = [NSJSONSerialization JSONObjectWithData:data
        options:NSJSONReadingAllowFragments error:nil];
    NSNumber *resultCodeNumber = [resDict objectForKey:@"ResultCode"];
    if ([resultCodeNumber integerValue] >=0)
    {
        NSMutableArray* listDict = [resDict objectForKey:@"Record"];
        NSMutableArray *listData = [NSMutableArray new];
        for (NSDictionary* dic in listDict) {
            Note *note = [Note new];
            note.ID = [dic objectForKey:@"ID"];
            note.date = [dic objectForKey:@"CDate"];
            note.content = [dic objectForKey:@"Content"];
            [listData addObject:note];
        }
        [[NSNotificationCenter defaultCenter] postNotificationName:
            DaoFindAllFinishedNotification object:listData];
    } else {
        NSInteger resultCode = [resultCodeNumber integerValue];
        NSNumber *resultCodeNumber = [NSNumber numberWithInt:resultCode];
        NSString* message = [resultCodeNumber errorMessage];
        NSDictionary *userInfo = [NSDictionary dictionaryWithObject:message
            forKey:NSLocalizedDescriptionKey];
        NSError *error = [NSError errorWithDomain:@"DAO"
            code:resultCode userInfo:userInfo];
        [[NSNotificationCenter defaultCenter]
            postNotificationName:DaoFindAllFailedNotification object:error];
    }
}];
[request setFailedBlock:^(
    NSError *error = [request error];
    [[NSNotificationCenter defaultCenter]
        postNotificationName:DaoFindAllFailedNotification object:error];
)];
[request startAsynchronous];
}

```

这里也是采用ASIHTTPRequest框架实现网络请求，其中的处理代码与基于委托模式的实现非常相似。第①行代码投送查询成功通知，通知中携带的object参数是我们查询返回的集合对象listData。第②行代码和第③行代码是投送查询失败通知，通知中携带的object参数是一个错误对象。

在NoteDAO中，插入、删除、修改方法与查询所有方法非常类似，我们不再介绍，大家可以自己下载代码查看。

20.3.6 业务逻辑层的代码实现

在我们这个应用中，业务逻辑层用到的类主要就是NoteBL类。NoteBL.m中查询所有方法的相关代码如下：

```

-(void) findAllNotes {
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(findAllFinished:)
        name:DaoFindAllFinishedNotification
        object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self

```

```

selector:@selector(findAllFailed:)
    name:DaoFindAllFailedNotification
    object:nil]; ②

[_dao findAll];
}
// 查询所有数据成功后调用的方法
- (void)findAllFinished:(NSNotification*)notification {
    NSMutableArray *resList = [notification object];
    [[NSNotificationCenter defaultCenter]
        postNotificationName:BLFindAllFinishedNotification object:resList]; ③
    [[NSNotificationCenter defaultCenter] removeObserver:self name:
        DaoFindAllFinishedNotification object:nil]; ④
    [[NSNotificationCenter defaultCenter] removeObserver:self name:
        DaoFindAllFailedNotification object:nil]; ⑤
}
// 查询所有数据失败后调用的方法
- (void)findAllFailed:(NSNotification*)notification {
    NSError *error = [notification object];
    [[NSNotificationCenter defaultCenter]
        postNotificationName:BLFindAllFailedNotification object:error]; ⑥
    [[NSNotificationCenter defaultCenter] removeObserver:self name:
        DaoFindAllFinishedNotification object:nil]; ⑦
    [[NSNotificationCenter defaultCenter] removeObserver:self name:
        DaoFindAllFailedNotification object:nil]; ⑧
}

```

findAllNotes:方法是由业务逻辑层异步调用的方法。在该方法中,需要注册两个查询通知,第①行代码是注册DaoFindAllFinishedNotification查询成功通知,第②行代码是注册DaoFindAllFailedNotification查询失败通知。

广播通知在使用时需要注册,在不再使用时需要解除,第④行和第⑦行代码用于解除DaoCreateFinishedNotification通知,第⑤行和第⑧行代码用于解除DaoFindAllFailedNotification通知。

在findAllFinished:方法中,第③行代码用于投送通知到表示层,然后再解除两个DAO查询通知。findAllFailed:方法也是类似的。

在NoteBL中,插入、删除、修改方法与查询所有方法非常类似,这里我们不再介绍,请大家自己下载代码查看。

20.3.7 表示层的代码实现

表示层用到的类主要有3个——MasterViewController、AddViewController和DetailViewController,下面我们分别介绍一下它们。

1. MasterViewController类

MasterViewController.m中查询请求的主要代码如下:

```

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    self.detailViewController = (DetailViewController *)
        [[self.splitViewController.viewControllers lastObject] topViewController];

    _bl = [NoteBL new];
    [_bl findAllNotes]; ①

    //初始化UIRefreshControl
    UIRefreshControl *rc = [[UIRefreshControl alloc] init];
    rc.attributedTitle = [[NSAttributedString alloc] initWithString:@"下拉刷新"];
    [rc addTarget:self action:@selector(refreshTableView)

```

```

        forControlEvents:UIControlEventsValueChanged];
self.refreshControl = rc;

[[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(findAllNotesFinished:)
                                             name:BLFindAllFinishedNotification
                                             object:nil];           ②
[[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(findAllNotesFailed:)
                                             name:BLFindAllFailedNotification
                                             object:nil];
[[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(removeNoteFinished:)
                                             name:BLRemoveFinishedNotification
                                             object:nil];
[[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(removeNoteFailed:)
                                             name:BLRemoveFailedNotification
                                             object:nil];           ③
}
- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    [[NSNotificationCenter defaultCenter] removeObserver:self];           ④
}
- (void) refreshTableView {
    if (self.refreshControl.refreshing) {
        self.refreshControl.attributedTitle = [[NSAttributedString
            alloc] initWithString:@"加载中..."];
        [_bl findAllNotes];           ⑤
    }
}
//成功查询所有数据后调用的方法
- (void)findAllNotesFinished:(NSNotification*)notification {           ⑥
    NSMutableArray *resList = [notification object];
    self.listData = resList;
    [self.tableView reloadData];
    if (self.refreshControl) {
        [self.refreshControl endRefreshing];
        self.refreshControl.attributedTitle = [[NSAttributedString
            alloc] initWithString:@"下拉刷新"];
    }
}
//查询所有数据失败后调用的方法
- (void)findAllNotesFailed:(NSNotification*)notification {
    NSError *error = [notification object];
    NSString *errorStr = [error localizedDescription];
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"操作信息"
                                                                message:errorStr
                                                                delegate:nil
                                                                cancelButtonTitle:@"OK"
                                                                otherButtonTitles: nil];

    [alertView show];
    if (self.refreshControl) {
        [self.refreshControl endRefreshing];
        self.refreshControl.attributedTitle = [[NSAttributedString
            alloc] initWithString:@"下拉刷新"];
    }
}
}

```

在主视图控制器中,第①行代码和第⑤行代码用于查询所有的异步调用,第①行在viewDidLoad方法中调用,即界面初始化时调用,第⑤行在表视图下拉刷新时调用。在viewDidLoad方法中,还注册了4个通知,如第②行代码~第③行代码所示。这些通知的注册是在viewDidLoad方法中进行的,解除也要在对应的方法中。在iOS 6中,viewDidUnLoad方法不再被使用,而是使用didReceiveMemoryWarning方法,这个方法也会在内存低报警时

调用，因此我们在这个方法中解除通知。第④行代码可以一次性解除所有已经注册的通知。

如果查询有了结果，主视图控制器会接收到通知，我们需要将这些通知中携带的参数提取出来。第⑥行的 [notification object] 语句使用通知的 object 方法提取参数。

MasterViewController.m 中删除请求的主要代码如下：

```
- (void)tableView:(UITableView *)tableView commitEditingStyle:
    (UITableViewCellEditingStyle)editingStyle
    forRowAtIndexPath:(NSIndexPath *)indexPath {
    if (editingStyle == UITableViewCellEditingStyleDelete) {
        _deletedIndex = [indexPath row];
        _deletedNote = [_listData objectAtIndex:_deletedIndex];

        [_bl removeNote: _deletedNote];

        [self.listData removeObject:_deletedNote];
        [tableView deleteRowsAtIndexPaths:@[indexPath]
            withRowAnimation:UITableViewRowAnimationFade];

    } else if (editingStyle == UITableViewCellEditingStyleInsert) {
    }
}
//成功删除备忘录后调用的方法
- (void)removeNoteFinished:(NSNotification*)notification
{
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"操作信息"
        message:@"删除成功。"
        delegate:nil
        cancelButtonTitle:@"OK"
        otherButtonTitles: nil];

    [alertView show];
}

//删除备忘录失败后调用的方法
- (void)removeNoteFailed:(NSNotification*)notification
{
    NSError *error = [notification object];
    NSString *errorStr = [error localizedDescription];
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"操作信息"
        message:errorStr
        delegate:nil
        cancelButtonTitle:@"OK"
        otherButtonTitles: nil];

    [alertView show];

    [self.listData insertObject:_deletedNote atIndex:_deletedIndex];
    [self.tableView reloadData];
}
```

2. AddViewController类

AddViewController.m 中插入请求的主要代码如下：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    _bl = [NoteBL new];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(createNoteFinished:)
        name:BLCreateNoteFinishedNotification
        object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(createNoteFailed:)
        name:BLCreateNoteFailedNotification
        object:nil];
}
```

```

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
- (IBAction)onclickSave:(id)sender {
    Note *note = [Note new];
    note.date = [[NSDate alloc] init];
    note.content = _textView.text;
    [_bl createNote: note];
}
//成功插入备忘录后调用的方法
- (void)createNoteFinished:(NSNotification*)notification {
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"操作信息"
                                                            message:@"插入成功。"
                                                            delegate:self
                                                            cancelButtonTitle:@"返回"
                                                            otherButtonTitles:@"继续", nil];

    [alertView show];
}
//插入备忘录失败后调用的方法
- (void)createNoteFailed:(NSNotification*)notification {
    NSError *error = [notification object];
    NSString *errorStr = [error localizedDescription];
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"操作信息"
                                                            message:errorStr
                                                            delegate:self
                                                            cancelButtonTitle:@"返回"
                                                            otherButtonTitles:@"继续", nil];

    [alertView show];
}
//响应对话框按钮事件
- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:
(NSInteger)buttonIndex {
    if (buttonIndex == 0) { //选择返回按钮
        [self dismissViewControllerAnimated:YES completion:nil];
    }
}

```

在viewDidLoad方法中，我们注册了BLCreateNoteFinishedNotification和BLCreateNoteFailedNotification通知。在didReceiveMemoryWarning方法中，[[NSNotificationCenter defaultCenter] removeObserver:self]解除所有的已注册通知。其他的代码我们前面都介绍过了，这里不再介绍。

3. DetailViewController类

DetailViewController.m中修改请求的主要代码如下：

```

- (void)viewDidLoad {
    .....
    _bl = [NoteBL new];
    [[NSNotificationCenter defaultCenter] addObserver:self
                                                selector:@selector(modifyNoteFinished:)
                                                name:BLCModifyFinishedNotification
                                                object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self
                                                selector:@selector(modifyNoteFailed:)
                                                name:BLCModifyFailedNotification
                                                object:nil];
}
- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
- (IBAction)onclickSave:(id)sender {
    Note *note = _detailItem;
    note.date = [[NSDate alloc] init];
}

```



```

        note.content = _textView.text;
        [_bl modifyNote:note];
    }
    //成功插入备忘录后调用的方法
    - (void)modifyNoteFinished:(NSNotification*)notification {
        UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"操作信息"
                                                                message:@"修改成功。"
                                                                delegate:self
                                                                cancelButtonTitle:@"返回"
                                                                otherButtonTitles:@"继续", nil];

        [alertView show];
    }
    //插入备忘录失败后调用的方法
    - (void)modifyNoteFailed:(NSNotification*)notification {
        NSError *error = [notification object];
        NSString *errorStr = [error localizedDescription];
        UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"操作信息"
                                                                message:errorStr
                                                                delegate:self
                                                                cancelButtonTitle:@"返回"
                                                                otherButtonTitles:@"继续", nil];

        [alertView show];
    }
    //响应对话框按钮事件
    - (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:
        (NSInteger)buttonIndex {
        if (buttonIndex == 0) { //选择返回按钮
            //返回
            UINavigationController *navController =
                (UINavigationController*)self.parentViewController;
            [navController popViewControllerAnimated:YES];
        }
    }

```

在viewDidLoad方法中，我们注册BLModifyFinishedNotification和BLModifyFailedNotification通知。在didReceiveMemoryWarning方法中，[[NSNotificationCenter defaultCenter] removeObserver:self]解除所有的已注册通知。

MyNotes应用中用到的通知都是自己定义的，通知机制中也有系统定义的一些通知，例如应用启动、终止等都会发出通知。

MyNotes应用中用到的通知定义在<工程名>-Prefix.pch文件（预编译头文件）中：

```

//定义DAO查询所有数据成功通知
#define DaoFindAllFinishedNotification @"DaoFindAllFinishedNotification"
//定义DAO查询所有数据失败通知
#define DaoFindAllFailedNotification @"DaoFindAllFailedNotification"
//定义DAO通过ID查询数据成功通知
#define DaoFindIdFinishedNotification @"DaoFindIdFinishedNotification"
//定义DAO通过ID查询数据失败通知
#define DaoFindIdFailedNotification @"DaoFindIdFailedNotification"
//定义DAO插入数据成功通知
#define DaoCreateFinishedNotification @"DaoCreateFinishedNotification"
//定义DAO插入数据失败通知
#define DaoCreateFailedNotification @"DaoCreateFailedNotification"
//定义DAO删除数据成功通知
#define DaoRemoveFinishedNotification @"DaoRemoveFinishedNotification"
//定义DAO删除数据失败通知
#define DaoRemoveFailedNotification @"DaoRemoveFailedNotification"
//定义DAO修改数据成功通知
#define DaoModifyFinishedNotification @"DaoModifyFinishedNotification"
//定义DAO修改数据失败通知
#define DaoModifyFailedNotification @"DaoModifyFailedNotification"
//定义BL查询所有数据成功通知
#define BLFindAllFinishedNotification @"BLFindAllFinishedNotification"

```

```
//定义BL查询所有数据失败通知
#define BLFindAllFailedNotification @"BLFindAllFailedNotification"
//定义BL插入数据成功通知
#define BLCreateNoteFinishedNotification @"BLCreateNoteFinishedNotification"
//定义BL插入数据失败通知
#define BLCreateNoteFailedNotification @"BLCreateNoteFailedNotification"
//定义BL删除数据成功通知
#define BLRemoveFinishedNotification @"BLRemoveFinishedNotification"
//定义BL删除数据失败通知
#define BLRemoveFailedNotification @"BLRemoveFailedNotification"
//定义BL修改数据成功通知
#define BLModifyFinishedNotification @"BLModifyFinishedNotification"
//定义BL修改数据失败通知
#define BLModifyFailedNotification @"BLModifyFailedNotification"
```

每一个工程都有一个<工程名>-Prefix.pch文件，它的作用域是整个工程。因此，在这个文件中定义的宏可以在整个工程中使用，但是不能跨越工程。因此，需要在每个工程预编译头文件，重复定义这些宏。

20.4 小结

通过重构MyNotes应用，我们把MyNotes应用的数据由原来的本地存储变成云存储。在这个过程中，我们介绍了移动网络通信应用中分层架构设计的必要性和重要性，还重点讲解了基于委托模式和观察者模式通知机制实现的分层架构设计。

iOS敏捷开发项目实战—— 2016里约热内卢奥运会应用 开发及App Store发布

这是本书的最后一章，也是本书的画龙点睛之笔。我想通过一个实际的应用使读者能够将本书前面讲过的知识点串联起来，了解iOS应用开发的一般流程，了解当下最为流行的开发方法学——敏捷开发。在开发过程中，我们会发现敏捷方法非常适合于iOS应用的开发。

21.1 应用分析与设计

本节中，我们从计划开发这个应用开始进行分析和设计，其中设计过程包括原型设计、数据库设计和架构设计。

21.1.1 应用概述

2012年伦敦奥运会结束时，我和智捷iOS课堂就在想开发一个介绍体育比赛的应用。比赛类应用是有时效性的，用户只会在比赛前使用，比赛结束后就没人使用了，而且比赛项目和日程表等信息会有一些变化，但是鉴于对体育的热爱，我们还是决定开发下一届奥林匹克运动会的应用。

目前，距离2016奥运会开幕时间还有3年的时间，官方并没有给出太多信息，而且比赛信息会随着时间的推进一点一点地发布，我们的应用也会随着这些信息的发布而更新版本。关于这个应用的发布情况，请读者关注智捷iOS课堂团队网站<http://www.51work6.com>，其源代码也会开源给大家，大家需要在法律许可的范围内使用。

我们首先介绍一下应用的需求。根据我们现在了解到的资料，能够在应用中提供的信息有举办城市、会徽、开幕时间、比赛项目和比赛日程的部分信息，其中比赛日程中的场地和会歌等信息还没有确定。

因此，我们整理了这个应用提供的一些功能：2016奥运会的一些基本信息、比赛项目、倒计时和比赛日程等。

21.1.2 需求分析

根据上面的功能描述，确定需求如下：

- ❑ 2016奥运会基本信息
- ❑ 比赛项目
- ❑ 倒计时
- ❑ 比赛日程
- ❑ 关于我们

这里我们采用用例分析方法描述用例图，如图21-1所示。

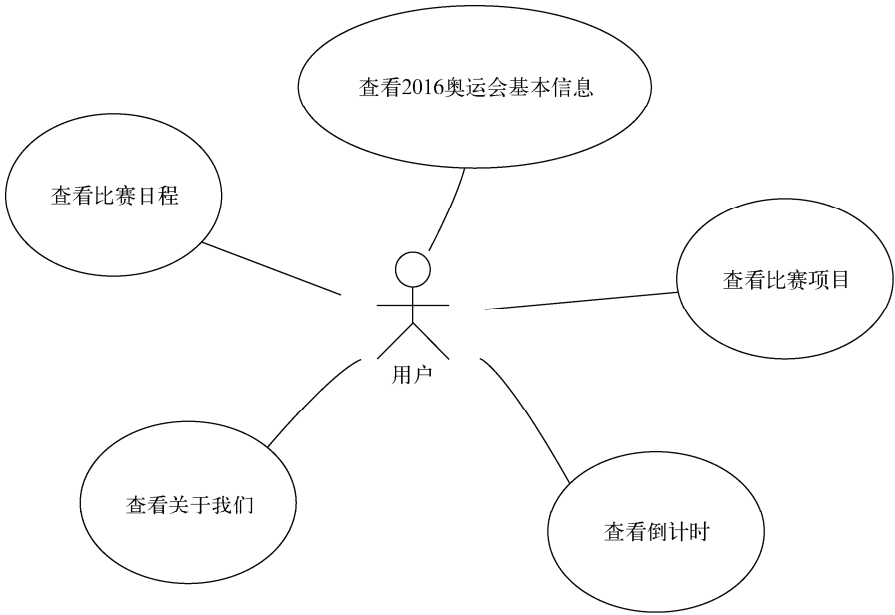


图21-1 2016奥运会应用用例图

21.1.3 原型设计

原型设计草图对于应用设计人员、开发人员、测试人员、UI设计人员以及用户都是非常重要的，该案例的原型如图21-2所示。

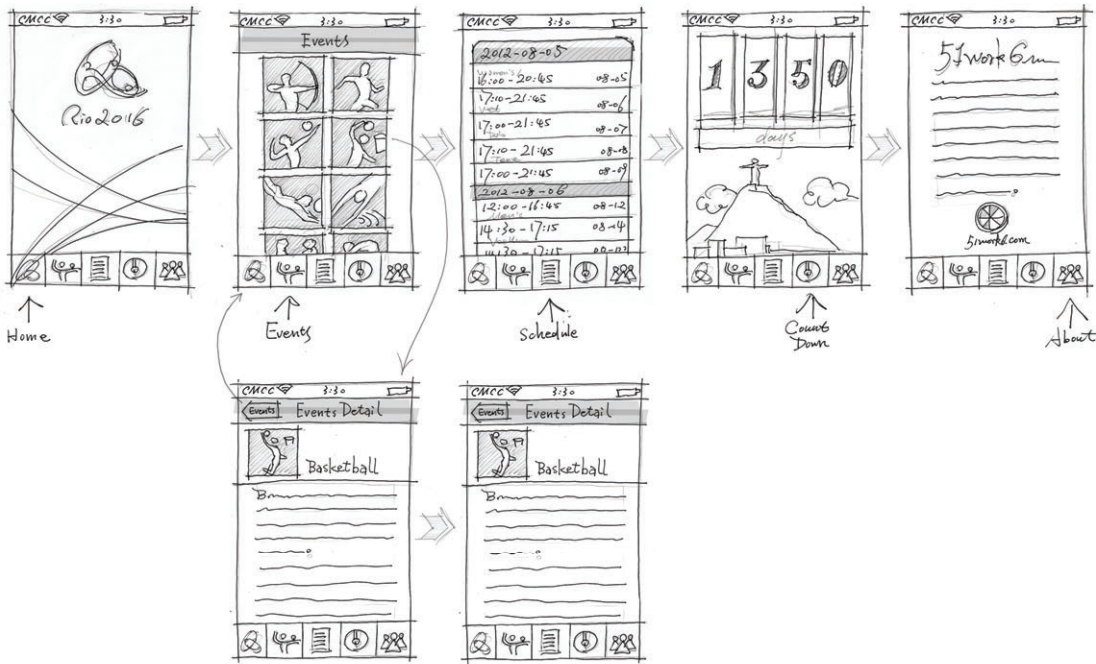


图21-2 原型设计图

21.1.4 数据库设计

从用例图可以了解到，这个应用都只是查看信息，没有信息插入、修改和删除等功能，那么这些信息放在哪里了呢？原本考虑放在一个文本文件中，但是文本文件的结构不好，管理起来也不方便，于是考虑到使用数据库，我们可以在第一次启动时将数据导入到iOS设备中的SQLite本地数据库。

在数据库概念设计阶段，我们首先需要找出应用中的实体，然后确定实体的属性以及实体关系。在数据库物理设计阶段，实体将演变成成为表，实体属性演变成成为字段。

应用中的实体有2016奥运会基本信息、比赛项目、倒计时信息和比赛日程，仔细分析后发现只有比赛项目和比赛日程，这是因为2016奥运会基本信息我们需要放到数据库中，倒计时信息是动态变化的，不需要放到数据库中。比赛项目和比赛日程是一对多的关系。最后的数据库物理数据模型如图21-3所示。

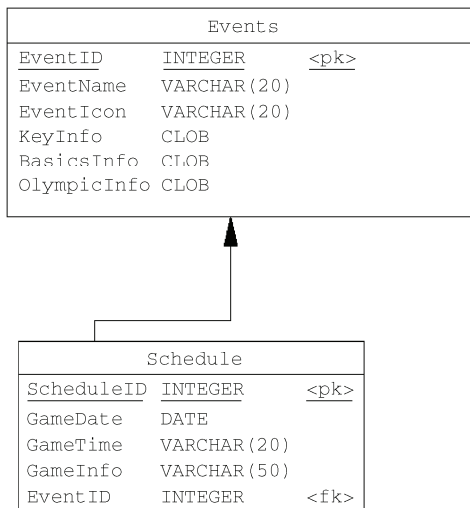


图21-3 数据库物理数据模型

该模型中的名词解释如下所示。

- ❑ **Events:** 比赛项目。
- ❑ **EventID:** 比赛项目编号，为主键，它是自增长整数类型。
- ❑ **EventName:** 比赛项目名。
- ❑ **EventIcon:** 比赛项目的图标名。
- ❑ **KeyInfo:** 比赛项目的关键信息。由于包含的字符很长，其数据类型采用大文本类型（CLOB）。
- ❑ **BasicsInfo:** 比赛项目基本信息，其数据类型采用大文本类型。
- ❑ **OlympicInfo:** 比赛项目奥运会历史信息，其数据类型采用大文本类型。
- ❑ **Schedule:** 比赛日程表。
- ❑ **ScheduleID:** 比赛日程ID，为主键，它是一个没有实际意义的自增长整型字段。由于其他字段都不太适合做主键，所以采用这种设计。
- ❑ **GameDate:** 比赛日期，日期类型。
- ❑ **GameTime:** 比赛时间，是在某个比赛日中的比赛时间段。
- ❑ **GameInfo:** 比赛描述，如男子单打和女子单打等。

两个表是通过EventID（比赛项目编号）关联在一起的。很多数据库建模工具都可以从物理数据模型生成DDL语句^①，使用这些语句可以很方便地创建和维护数据库中的表结构。

21.1.5 架构设计

为应用设计架构也是必需的，而且分层架构设计还可以提高项目管理水平、科学化任务分配以及实施敏捷开发。架构设计就是应用的“骨架”，搭建好“骨架”，我们再往里添砖加瓦。第7章介绍的低耦合分层设计结构基本上贯穿本书，也是我们推荐的移动应用架构。图21-4是2016奥运会应用分层架构设计图。

2016奥运会应用采用的分层架构设计是基于同一工作空间不同工程的分层实现模式。表示层是iOS应用工程，业务逻辑层和数据持久层是静态链接库工程。信息系统层是SQLite3数据库，没有采用Core Data技术，主要考虑从数据库设计的物理数据模型生成DDL语句非常方便。另外，应用中用到的数据很多，需要在应用启动时

① 数据定义语句，用于创建、删除和修改数据库对象，包括DROP、CREATE、ALTER、GRANT、REVOKE和TRUNCATE等语句。

插入到数据库中，这时使用insert语句批量插入数据性能会更好一些。

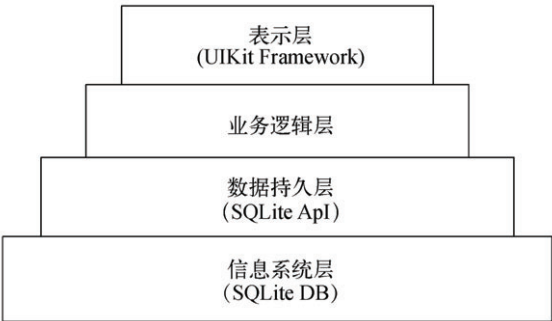


图21-4 2016奥运会应用分层架构设计图

21.2 iOS 敏捷开发

方法决定效率，好的开发方法可以大大地提高开发效率，敏捷开发是当下比较流行的软件开发方法学。面对激烈的市场竞争，iOS应用开发更需要敏捷。

21.2.1 敏捷开发宣言

“敏捷”（agile）源于2001年美国犹他州雪鸟滑雪圣地的一次聚会，这是聚会敏捷方法发起者和实践者的聚会。经过两天的讨论，通过一份简明扼要的《敏捷宣言》，概括了一套全新的软件开发价值观，从此宣告了敏捷开发运动的开始。

《敏捷宣言》的价值观如下所示。

- ❑ 人与人的交互重于过程和工具。
- ❑ 可用的软件重于求全责备的文档。
- ❑ 客户协作重于合同谈判。
- ❑ 随时应对变化重于遵循计划。

《敏捷宣言》背后还有12个原则，具体如下所示。

- ❑ 我们的最高目标是尽早和不断交付有价值的软件满足客户需要。
- ❑ 我们欢迎需求的变化，即使在开发后期。敏捷过程能够驾驭变化，保持客户的竞争优势。
- ❑ 经常交付可用的软件，从几星期到几个月，时间尺度越短越好。
- ❑ 项目过程中，业务人员与开发人员必须在一起工作。
- ❑ 要善于激励项目人员，给他们所需要的环境和支持，并相信他们能够完成任务。
- ❑ 在开发小组中最有效的沟通方法是面对面的交谈。
- ❑ 可用的软件是进度的主要衡量标准。
- ❑ 敏捷过程提倡可持续开发。出资人、开发人员和用户应该总是维持不变的节奏。
- ❑ 对技术的精益求精以及对设计的不断完善将提升敏捷性。
- ❑ 简单，尽可能减少工作量，也是一门艺术。
- ❑ 最佳的架构、需求和设计出自于自组织的团队。
- ❑ 团队要定期反省如何能够做到更有效，并相应地调整团队的行为。

《敏捷宣言》对软件开发行业具有非常重要的意义。

21.2.2 iOS 适合敏捷开发？

iOS 平台适合使用敏捷开发方法吗？当然适合。一般 iOS 平台的开发团队不会太大，起码在 iOS 设备端是这样的，一般是 2~3 人的团队。事实上，敏捷开发并不适合于大团队。随着团队规模的扩大，团队成员之间的沟通必然会出现一些问题。自组织的团队更不适合于大型团队。因此，2~3 人的小团队管理起来非常方便、灵活，敏捷开发实现起来也很容易。

此外，在 App Store 上发布 iOS 应用时，需要能够快速增量迭代，经常交付可用的版本，以快速占领市场，如果等到所有功能全部完成再发布，那样就已经失去了市场。让用户先用起来，再不断地完善，我们要开发的 2016 奥运会应用也是一样的，如果我们等到 2016 奥运会官方发布了全部的信息，我们再开发这个应用可能真要等到 2016 年了。它的场地信息、会歌可能需要等到开幕的前几个月才能定下来，我们可以先不包括这些信息，等到官方发布，我们再完善这些功能，我们应该提倡增量迭代发布，可能今天晚上奥运会会歌定下来了，那么明天早上就应该有一个包括播放会歌的版本发布。

测试驱动也是敏捷开发最重要的部分，相关内容在第 16 章中介绍过。iOS 平台现在有很多单元测试框架，有 Xcode 工具自带的 OCUnit 框架、开源测试框架 GHUnit 以及提供伪对象的 OCMock 框架，这些单元测试框架为测试驱动的软件开发提供了技术保证，只有敏捷开发指导下的测试驱动才能发挥这些测试框架的优势。

21.2.3 iOS 敏捷开发最佳实践

作为应对快速变化需求的一种软件开发方法，具体名称、理念、过程和术语都不尽相同。在具体实施上，敏捷开发又分为不同的几个“门派”，如 Scrum、XBreed、极限编程（XP，eXtreme Programming）和水晶方法等。而本书不打算介绍这些“门派”，只想取一些精华知识点应用于 iOS 平台，实现真正意义上的敏捷开发的最佳实践。根据多年 iOS 开发的经验，我总结了 iOS 应用敏捷开发的最佳实践，具体包括如下几项：

- ❑ 增量迭代
- ❑ 小型发布
- ❑ 测试驱动
- ❑ 科学分配任务

关于增量迭代、小型发布和测试驱动，我们在上一节中已经介绍过了，下面我们介绍科学分配任务的实现过程。

多年前，我在做项目经理的时候我的老板问我：一个 30 人月^①的项目是否是招聘到 30 人工作一个月就完成了呢？我的回答是基本上不可能，一方面是人越多越不好管理，另一方面是工作任务的分配是否合理。在实际开发工作中，我们可能都有这样的经历：一个人在写代码，其他人在背后看他“表演”，原因是别人需要他的代码，由于他的代码没有完成，或者是出现了问题，其他人等着他修改。这是由于任务分配是串行的，串行任务无法通过添加人员来提高开发效率。图 21-5 所示是串行分配任务的甘特图。

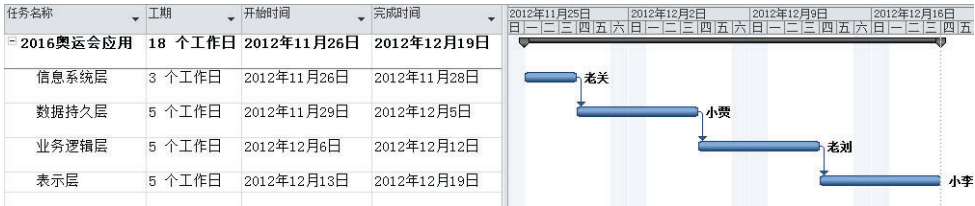


图 21-5 串行分配任务的甘特图

按照图 21-5 分配任务，4 个人需要 18 个工作日才能完成。但是有个问题，那就是在每个人工作的时候，其他的

① 人月是软件工程中衡量工作量的一种方法，通过人月数，评估工程开发成本，30 人月可以简单地理解为 10 人工作 3 月。

3个人都处于空闲状态。工作效率很低，开发周期比较长。图21-6是并行分配任务的甘特图。



图21-6 并行分配任务的甘特图

按照图21-6分配任务，4个人只需要8个工作日，开发周期缩短了10个工作日。可见，并行任务可以大大提供工作效率。当然，在实际的项目开发过程中，情况要复杂得多，例如开发人员的能力的差别，开发人员是否有其他的工作任务等。影响并行任务的除了人本身的因素外，还有任务本身的特点。在图21-6所示的任务中，数据持久层的开发任务依赖于信息系统层的完成，即老关没有创建好数据库，小贾就不能开发数据持久层，这种情况下我们串行处理。但有时我们可以通过一些技术手段，虚拟一些假的环境以便能够并行开发。

看过图21-6任务分配的人会问，为什么任务的分配是按照层划分，而不是按照业务模块划分的呢？按照业务模块划分多么方便啊，4个人每人一个模块，互相之间没有太多的依赖关系，这样开发速度很快。这种观点是错误的，表面看每个模块之间没有关系，但是模块之间的信息很多是共用的，开发之后他们会发现编码和测试更加困难。分层设计的另外一个好处是细化开发角色，提高开发效率，这样开发人员只关注于自己擅长的方面，构建专家级系统即可。

总之，作为项目管理者，应该尽可能根据现有人员的情况，将开发任务分成并行的子任务。注意任务之间的依赖关系，要合理分配。

经过了认真编排和详细计划，2016奥运会应用的工作任务如图21-7所示。

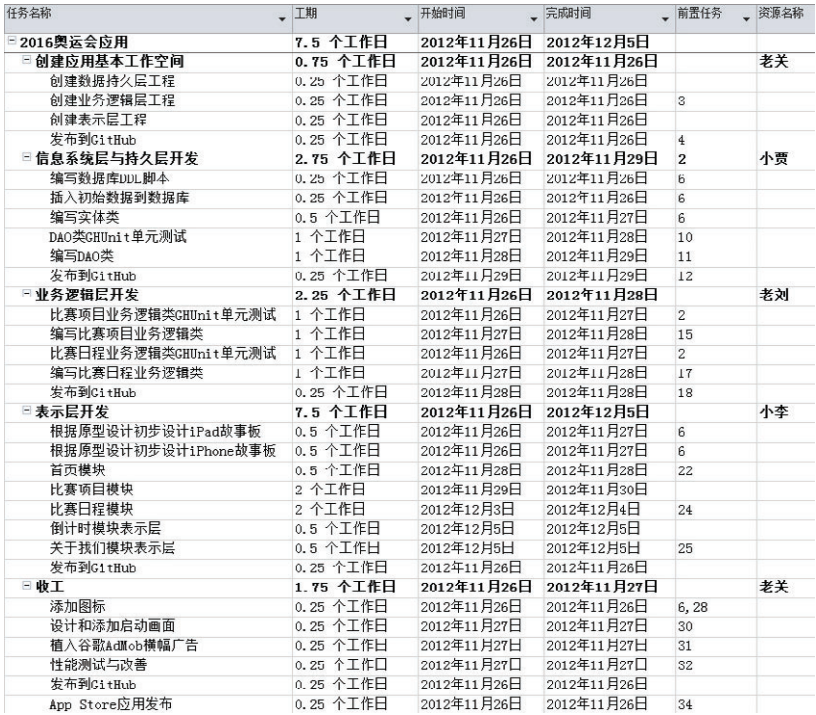


图21-7 2016奥运会应用任务分配甘特图

在接下来的内容中，我们将以这任务分配甘特图为主线，详细介绍这个应用的开发、测试和发布过程。由于有些任务是迭代的，重复的步骤我们不再介绍。另外，前面讲过的内容这里不再重复介绍了。

21.3 任务 1：创建应用基本工作空间

在开发项目之前，应该由一个人搭建开发环境，然后把环境复制给其他人使用。在这个应用中，我们采用了基于同一个工作空间的分层设计。任务1是由老关创建好工作空间，将源代码提交到GitHub中，然后再由其他的成员克隆到本地。在项目开发过程中，要求严格遵守使用GitHub进行源代码的版本控制。

首先，使用Xcode创建一个工作空间2016Olympics，具体步骤可参考7.5.4节。然后，分别创建其他3个工程。

1. 迭代1.1：创建数据持久层工程

在工作空间2016Olympics中添加一个静态链接库工程PersistenceLayer，具体步骤可参考7.5.4节。

2. 迭代1.2：创建业务逻辑层工程

在工作空间2016Olympics中添加一个静态链接库工程BusinessLogicLayer，具体步骤可参考7.5.4节。创建完工程后，需要配置PersistenceLayer工程的依赖关系。

3. 迭代1.3：创建表示层工程

在工作空间2016Olympics中添加应用程序工程，模板选择为Single View Application。在创建工程选项对话框中输入内容，如图21-8所示。

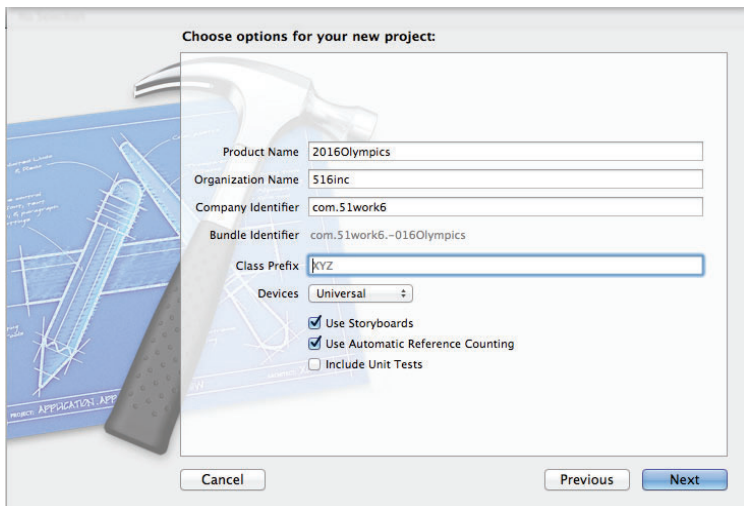


图21-8 新工程中的选项

这里Product Name（工程名）为2016Olympics，Organization Name（组织名）为516inc，Company Identifier（公司标识符）为com.51work6，Devices（选择设备）为Universal，这个选择可以使这个工程同时支持在iPad和iPhone（或iPod touch）设备上运行。这里选中Use Storyboards复选框和Use Automatic Reference Counting复选框。

创建完工程后，需要配置BusinessLogicLayer工程的依赖关系，具体步骤可参考7.5.4节。

4. 迭代1.4：发布到GitHub

老关完成了这些工作并编译通过后，需要将这个空白的工作空间分发给小李、老刘和小贾，让他们使用GitHub发布应用的第一个版本。注意，其他成员就不要自己再创建工作空间或任何工程了，只需要从GitHub上克隆代码，然后在自己的工程中添加自己的内容就可以了。

我们需要使用GitHub账号登录，并创建一个名为2016Olympics_iOS的代码库，如图21-9所示。

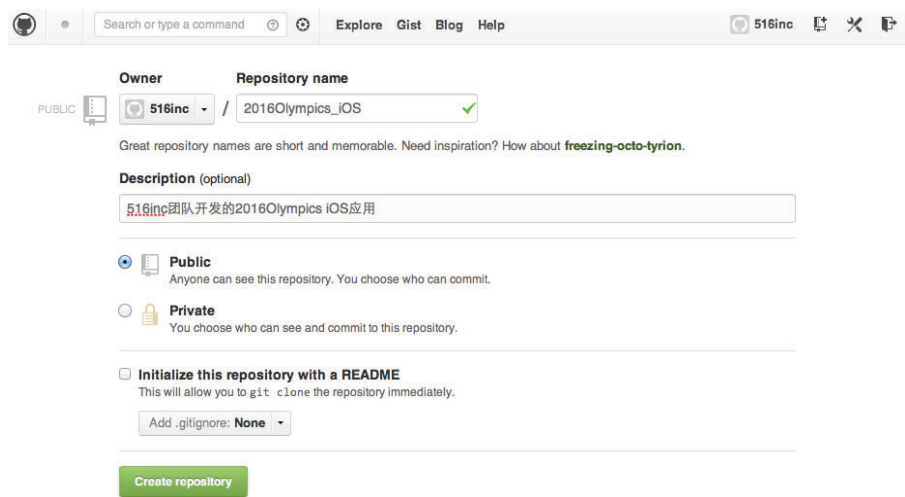


图21-9 创建代码库

创建完成后，需要将本地代码上传到GitHub服务器，具体步骤可参考18.2.4节。其他的成员使用`$git clone git@github.com:516inc/2016Olympics_iOS.git`克隆到本地。

21.4 任务 2：信息系统层与持久层开发

该任务由对数据库很熟悉、对SQLite API也非常熟悉的小贾负责完成，其他成员不用了解SQL语句、不需要知道SQLite API。

21.4.1 迭代 2.1：编写数据库DDL脚本

我们要按照图21-3所示的数据库物理数据模型来编写数据库DDL脚本。也有一些工具可以生成DDL脚本，然后把这个脚本放在数据库中执行就可以了。下面是编写的DDL脚本。

```
/*=====*/
/* DBMS name:      SQLite3 DB                               */
/* Created on:      2012/11/21 15:59:37                       */
/*=====*/

drop table if exists Events;

drop table if exists Schedule;

/*=====*/
/* Table: Events                                           */
/*=====*/
create table Events
(
    EventID          INTEGER      primary key autoincrement    not null,
    EventName        VARCHAR(20),
    EventIcon        VARCHAR(20),
    KeyInfo          CLOB,
    BasicsInfo      CLOB,
    OlympicInfo      CLOB
);

/*=====*/
/* Table: Schedule                                         */
/*=====*/
```

```

/*-----*/
create table Schedule
(
    ScheduleID          INTEGER    primary key autoincrement      not null,
    GameDate            DATE              not null,
    GameTime            VARCHAR(20)      not null,
    GameInfo            VARCHAR(50),
    EventID             INTEGER,
    constraint FK_SCHEDULE_REFERENCE_EVENTS foreign key (EventID) references Events (EventID)
);

```

21.4.2 迭代 2.2：插入初始数据到数据库

在应用中没有插入、删除和修改功能，那么原始的数据如何插入到数据库中呢？我们可以在应用启动并且建表完成后，使用insert语句插入到数据库中。我们在脚本文件中预先写好insert语句，类似于下面的语句：

```

insert into Events(EventName,EventIcon,KeyInfo,BasicsInfo,OlympicInfo)
values ('Athletics','athletics.gif','Athletics is...','There are four
main strands...','The ancient Olympic ...');
insert into Schedule (GameDate,GameTime,GameInfo,EventID) values
('2012-08-05','16:00 - 20:45','Women''s',28);

```

使用上面的语句，我们可以一起编写脚本文件create_load.sql，具体可参考工程中DBFile目录下的create_load.sql文件。

21.4.3 迭代 2.3：编写实体类

实体是应用中的人、事、物，在数据库设计的时候演变成为“表”，而在面向对象分析和设计时，实体演变成为“实体类”。因此，实体类与数据库的表有共同的渊源。图21-10是应用的实体类图，它看起来很像如图21-3所示的数据库物理数据模型。

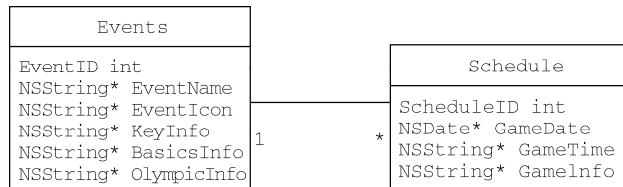


图21-10 实体类图

在实体类图中，实体比赛项目（Events）和比赛日程（Schedule）是一对多的关联关系。实体类Events的代码如下：

```

#import <Foundation/Foundation.h>

//比赛项目实体类
@interface Events : NSObject

//编号
@property(nonatomic, assign) NSUInteger EventID;
//项目名
@property(nonatomic, strong) NSString* EventName;
//项目图标
@property(nonatomic, strong) NSString* EventIcon;
//项目关键信息
@property(nonatomic, strong) NSString* KeyInfo;
//项目基本信息
@property(nonatomic, strong) NSString* BasicsInfo;
//项目奥运会历史信息

```



```

@property(nonatomic, strong) NSString* OlympicInfo;

@end

#import "Events.h"

@implementation Events

@end

```

实体类Schedule的代码如下：

```

#import <Foundation/Foundation.h>
#import "Events.h"

// 比赛日程表实体类
@interface Schedule : NSObject

// 编号
@property(nonatomic, assign) NSUInteger ScheduleID;
// 比赛日期
@property(nonatomic, strong) NSString* GameDate;
// 比赛时间
@property(nonatomic, strong) NSString* GameTime;
// 比赛描述
@property(nonatomic, strong) NSString* GameInfo;
// 比赛项目
@property(nonatomic, strong) Events* Event;

@end

#import "Schedule.h"

@implementation Schedule

@end

```

注意 看Event属性。在对象模型中，关联关系是通过在Schedule中定义属性Event关联在一起的。但是在Events实体类定义中，并没有与Schedule关联的属性，这说明这种关联关系是单向的。

21.4.4 迭代 2.4：DAO类GHUnit单元测试

编写完实体类后，我们看看DAO类的编写。由于这里采用的是测试驱动的开发方法，所以需要先进行DAO单元测试。即便是测试驱动，我们也需要先创建一些DAO类，这些类中的方法先不用具体实现。例如，EventsDAO的基本实现代码如下：

```

@implementation EventsDAO

// 插入数据的方法
-(int) create:(Events*)model{
    return 0;
}

// 删除数据的方法
-(int) remove:(Events*)model{
    return 0;
}

// 修改数据的方法
-(int) modify:(Events*)model{
    return 0;
}

// 查询所有数据的方法

```



```

- (NSMutableArray*) findAll{
    return nil;
}
//按照主键查询数据的方法
- (Events*) findById:(Events*)model{
    return nil;
}
@end

```

这里采用GJUnit单元测试框架，因此需要在PersistenceLayer工程中添加GJUnit单元测试框架，具体步骤可参考16.3节。在配置工程中，DBFile组中的create_load.sql和DBConfig.plist也需要设置Target Membership。如图21-11所示，选中GJUnitTest Target，把这两个文件编译到GJUnitTest执行文件中。

下面我们看看测试代码部分，EventsDAOTests测试类是用来测试EventsDAO的，它的代码如下：

```

#import <GJUnit/GJUnit.h>
#import "EventsDAO.h"
#import "Events.h"
#import "DBHelper.h"

@interface EventsDAOTests : GJUnitTestCase {}
@property (nonatomic, strong) EventsDAO * dao;
@property (nonatomic, strong) Events * theEvents;

@end

@implementation EventsDAOTests

- (void)setUpClass {
    //创建EventsDAO对象
    self.dao = [EventsDAO sharedManager];
    //创建Events对象
    self.theEvents = [[Events alloc] init];
    self.theEvents.EventName = @"test EventName";
    .....
}

- (void)tearDownClass {
    self.dao = nil;
}

- (void)setUp {}
- (void)tearDown {}

//测试插入数据的方法
- (void) test_1_Create
{
    int res = [self.dao create:self.theEvents];
    //断言无异常，返回值为0
    GJUnitAssertTrueNoThrow(res == 0, @"数据插入失败");
}

//测试按照主键查询数据的方法
- (void) test_2_FindById
{
    self.theEvents.EventID = 41;
    Events* resEvents = [self.dao findById:self.theEvents];
    //断言 查询结果非nil
    GJUnitAssertNotNil(resEvents, @" 查询记录为nil");
    //断言
    GJUnitAssertEqualObjects(self.theEvents.EventName , resEvents.EventName,
        @"比赛项目名测试失败");
    .....
}

//测试查询所有数据的方法
- (void) test_3_FindAll
{
    NSArray* list = [self.dao findAll];

```

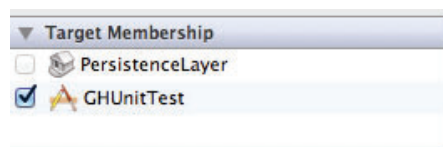


图21-11 设置Target Membership

```

//断言查询记录数为1
GHAAssertTrue([list count] == 41, @"查询记录数期望值为: 42 实际值为: %i",
[list count]);

Events* resEvents = list[40];
//断言
GHAAssertEqualObjects(self.theEvents.EventName, resEvents.EventName,
@"比赛项目名测试失败");
.....
}
//测试修改数据的方法
-(void) test_4_Modify
{
    self.theEvents.EventID = 41;
    self.theEvents.EventName = @"test modify EventName";

    int res = [self.dao modify:self.theEvents];
    //断言无异常, 返回值为0
    GHAAssertTrueNoThrow(res == 0, @"数据修改失败");

    Events* resEvents = [self.dao findById:self.theEvents];
    //断言查询结果非nil
    GHAAssertNotNil(resEvents, @"查询记录为nil");
    //断言
    GHAAssertEqualObjects(self.theEvents.EventName, resEvents.EventName, @"比赛项目名测试失败");
    .....
}
//测试删除数据的方法
-(void) test_5_Remove
{
    int res = [self.dao remove:self.theEvents];
    //断言无异常, 返回值为0
    GHAAssertTrueNoThrow(res == 0, @"数据修改失败");

    Events* resEvents = [self.dao findById:self.theEvents];
    //断言查询结果为nil
    GHAAssertNil(resEvents, @"记录删除失败");
}
@end

```

DBHelper类是一个辅助类, 帮助初始化数据库。我们先看看DBHelper.h的代码:

```

#import <Foundation/Foundation.h>
#import "sqlite3.h"

@interface DBHelper : NSObject
{
    sqlite3 *db;
}

//获得沙箱Documents目录下的全路径
+ (NSString *)applicationDocumentsDirectoryFile :(NSString *)fileName;

//初始化并加载数据
-(void) initDB;

//从数据库中获得当前数据库的版本号
- (int) dbVersionNubmer;

@end

```

在DBHelper中, 我们定义了数据库SQLite成员变量db以及3个方法, 其中initDB用来初始化数据库, dbVersionNubmer方法用来返回当前数据库的版本号。

在DBHelper.m中, initDB方法的代码如下:

```

//初始化并加载数据
-(void) initDB
{
    NSString* configTablePath = [[NSBundle mainBundle]
        pathForResource:@"DBConfig" ofType:@"plist"];
    NSDictionary* configTable = [[NSDictionary alloc]
        initWithContentsOfFile:configTablePath];
    //从配置文件获得数据版本号
    NSNumber *dbConfigVersion = [configTable objectForKey:@"DB_VERSION"];
    //从数据库DBVersionInfo表记录返回的数据库版本号
    int versionNubmer = [self dbVersionNubmer];

    //版本号不一致
    if ([dbConfigVersion intValue] != versionNubmer)
    {
        NSString* dbFilePath = [DBHelper
            applicationDocumentsDirectoryFile:DB_FILE_NAME];
        if (sqlite3_open([dbFilePath UTF8String], &db) != SQLITE_OK) {
            sqlite3_close(db);
            NSAssert(NO,@"数据库打开失败。");
        } else {
            //加载数据到业务表中
            NSLog(@"数据库升级...");
            char *err;
            NSString * createtablePath = [[NSBundle mainBundle]
                pathForResource:@"create_load" ofType:@"sql"];
            NSString* sql = [[NSString alloc] initWithContentsOfFile:
                createtablePath encoding:NSUTF8StringEncoding error:nil];
            if (sqlite3_exec(db,[sql UTF8String],NULL,NULL,&err) != SQLITE_OK) {
                NSLog(@"数据库升级失败的原因 : %@",[NSMutableString stringWithCString:
                    err encoding:NSUTF8StringEncoding]);
            }
            //把当前版本号写回到文件中
            NSString* usql = [[NSString alloc] initWithFormat: @"update DBVersionInfo
                set version_number = %i",[dbConfigVersion intValue]];
            if (sqlite3_exec(db,[usql UTF8String],NULL,NULL,&err) != SQLITE_OK) {
                NSLog(NO,@"更新DBVersionInfo数据失败。");
            }
            sqlite3_close(db);
        }
    }
}

```

该方法的主要目的是执行DLL脚本创建数据库，然后insert语句将数据插入到数据库。但是并不是每次都进行建表和插入数据的处理，这里通过一个数据库版本号来管理什么时候创建和插入。我们在数据库中建立一个表DBVersionInfo，它有一个字段version_number，用来记录当前数据库的版本号。如果进行了建表和插入数据的操作，则会更改这个版本号。第⑨行代码是更新数据库版本号的SQL语句。数据库记录的这个版本号要与应用程序资源文件DBConfig.plist中保存的版本号一致，如果不一致，就会执行创建表和插入数据处理。DBConfig.plist的内容如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://
    www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>DB_VERSION</key>
    <integer>8</integer>
</dict>
</plist>

```

没有发布新版本的时候，我们只需要修改一下这个属性列表文件中的DB_VERSION项让它累加即可，它相当于配置文件。

第⑧行代码是从create_load.sql脚本文件中读取SQL语句实现建表和插入数据，程序执行到这里的时候是最耗

时的。这个文件中的字符量很大，我们进行数据库版本的管理是有必要的。

ScheduleDAO的测试类ScheduleDAOTests与EventsDAOTests非常类似，这里我们就不再介绍了。运行测试用例，得到的结果如图21-12所示。

图21-12是所有测试用例测试通过的结果。但是在没有编写DAO中的具体方法之前，运行结果应该全部是红色的，随着编写完成和测试通过慢慢都会变成如图21-12所示的样子。这里的这个测试→编写→再测试是一个不断迭代螺旋上升的过程。



图21-12 运行数据持久层的测试用例

21.4.5 迭代 2.5：编写DAO类

编写DAO类是我们主要的工作。图21-13是应用的DAO类的类图。

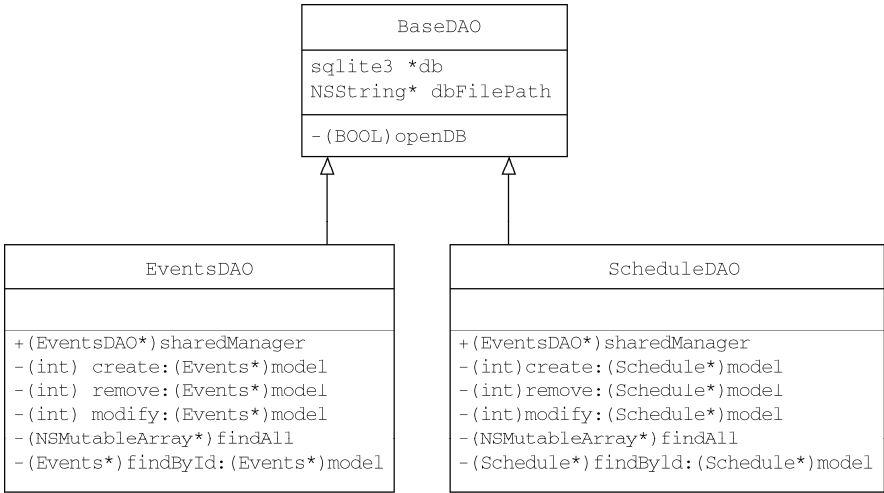


图21-13 DAO类的类图

EventsDAO和ScheduleDAO类都继承了BaseDAO类，采用单例设计模式设计，其中的sharedManager方法是获得DAO实例的方法。每一个DAO类都有5个方法，它们分别实现数据的插入、删除、修改和查询。

BaseDAO.h的代码如下：

```
#import "sqlite3.h"
#import "DBHelper.h"

@interface BaseDAO : NSObject
{
    sqlite3 *db;
}
//数据文件全路径
@property(nonatomic, strong) NSString* dbFilePath;

//打开SQLite数据库。如果返回true表示成功打开，返回false表示打开失败
-(BOOL)openDB;
@end
```

这里定义了SQLite成员变量db，这样子类EventsDAO和ScheduleDAO中就不再定义了。dbFilePath属性的定义也有这样的目的。openDB是打开数据库的方法，由于在DAO中多次使用，我们也在这里定义。此外，我们在BaseDAO中实现了openDB方法。

BaseDAO.m的代码如下：

```
#import "BaseDAO.h"
@implementation BaseDAO
- (id)init {
    self = [super init];
    if (self) {
        self.dbFilePath = [DBHelper applicationDocumentsDirectoryFile:DB_FILE_NAME];
        //初始化数据库
        DBHelper *dbhelper = [DBHelper new];
        [dbhelper initDB];
    }
    return self;
}
-(BOOL)openDB {
    if (sqlite3_open([self.dbFilePath UTF8String], &db) != SQLITE_OK) {
        sqlite3_close(db);
        NSLog(@"数据库打开失败。");
        return false;
    }
    return true;
}
@end
```

init方法是构造方法，这个方法在子类中通过[super init]方法调用。在该方法中，我们调用[dbhelper initDB]语句初始化数据库。openDB方法是打开数据库方法，如果打开失败，则关闭数据库并返回false，如果打开成功则返回true。

下面我们看看EventsDAO.m中的关键代码，其中插入数据的代码如下：

```
//插入数据方法
-(int) create:(Events*)model
{
    if ([self openDB]) {
        NSString *sqlStr = @"INSERT INTO Events (EventName,EventIcon,KeyInfo,BasicsInfo,
            OlympicInfo) VALUES (?, ?, ?, ?, ?)";
        sqlite3_stmt *statement;
        //预处理过程
        if (sqlite3_prepare_v2(db, [sqlStr UTF8String], -1, &statement, NULL) == SQLITE_OK) {

            //绑定参数开始
```

```

        sqlite3_bind_text(statement, 1, [model.EventName UTF8String], -1, NULL);
        sqlite3_bind_text(statement, 2, [model.EventIcon UTF8String], -1, NULL);
        sqlite3_bind_text(statement, 3, [model.KeyInfo UTF8String], -1, NULL);
        sqlite3_bind_text(statement, 4, [model.BasicsInfo UTF8String], -1, NULL);
        sqlite3_bind_text(statement, 5, [model.OlympicInfo UTF8String], -1, NULL);
        //执行插入
        if (sqlite3_step(statement) != SQLITE_DONE) {
            NSLog(@"插入数据失败。");
        }
    }
    sqlite3_finalize(statement);
    sqlite3_close(db);
}
return 0;
}

```

删除数据的代码如下：

```

//删除数据的方法
-(int) remove:(Events*)model
{
    if ([self openDB]) {

        //先删除子表（比赛日程表）相关数据
        NSString *sqlScheduleStr = [[NSString alloc] initWithFormat:@"DELETE from Schedule
                                where EventID=%i",model.EventID];
        char *err;

        //开启事务，立刻提交之前的事务
        sqlite3_exec(db,"BEGIN IMMEDIATE TRANSACTION",NULL,NULL,&err);

        if (sqlite3_exec(db,[sqlScheduleStr
                            UTF8String],NULL,NULL,&err) != SQLITE_OK) {
            //回滚事务
            sqlite3_exec(db,"ROLLBACK TRANSACTION",NULL,NULL,NULL);
        }

        //先删除主表（比赛项目）数据
        NSString *sqlEventsStr = [[NSString alloc] initWithFormat:
                                @"DELETE from Events where EventID =%i;",model.EventID];

        if (sqlite3_exec(db,[sqlEventsStr UTF8String],NULL,NULL,&err) !=
            SQLITE_OK) {
            //回滚事务
            sqlite3_exec(db,"ROLLBACK TRANSACTION",NULL,NULL,&err);
        }
        //提交事务
        sqlite3_exec(db,"COMMIT TRANSACTION",NULL,NULL,&err);

        sqlite3_close(db);
    }

    return 0;
}

```

这个删除方法非常特殊。由于比赛项目与比赛日程表之间有“主从”关系，当删除主表中的数据时，也要删除从表中的数据。比如我们在比赛项目表中删除了赛马比赛项目，那么在比赛日程表中当然也不能有赛马项目的比赛日程了，因此需要先删除从表（比赛日程表）中的相关数据，再删除主表（比赛项目表）中的数据。第①行代码是删除从表中数据的SQL语句，第③行是执行SQL语句，第⑤行代码是删除主表中数据的SQL语句，第⑥行是执行SQL语句。

这两条SQL语句的执行应该在一个事务里，它们具有原子性，要么全部成功，要么全部失败。第②行代码立刻提交之前的事务，并开启一个新事务。在执行删除操作失败的情况下，第④行代码和第⑦行代码要回滚事务。在成功删除的情况下，需要提交事务，如第⑧行代码所示。

查询所有数据方法findAll的代码如下：

```
// 查询所有数据方法
- (NSMutableArray*) findAll
{
    NSMutableArray *listData = [[NSMutableArray alloc] init];

    if ([self openDB]) {

        NSString *qsql = @"SELECT EventName,
            EventIcon, KeyInfo, BasicsInfo, OlympicInfo, EventID FROM Events";

        sqlite3_stmt *statement;
        // 预处理过程
        if (sqlite3_prepare_v2(db, [qsql UTF8String], -1, &statement,
            NULL) == SQLITE_OK) {

            // 执行
            while (sqlite3_step(statement) == SQLITE_ROW) {

                Events* events = [[Events alloc] init];

                char *cEventName = (char *) sqlite3_column_text(statement, 0);
                events.EventName = [[NSString alloc] initWithUTF8String: cEventName];

                char *cEventIcon = (char *) sqlite3_column_text(statement, 1);
                events.EventIcon = [[NSString alloc] initWithUTF8String: cEventIcon];

                char *cKeyInfo = (char *) sqlite3_column_text(statement, 2);
                events.KeyInfo = [[NSString alloc] initWithUTF8String: cKeyInfo];

                char *cBasicsInfo = (char *) sqlite3_column_text(statement, 3);
                events.BasicsInfo = [[NSString alloc]
                    initWithUTF8String: cBasicsInfo];

                char *cOlympicInfo = (char *) sqlite3_column_text(statement, 4);
                events.OlympicInfo = [[NSString alloc]
                    initWithUTF8String: cOlympicInfo];

                events.EventID = sqlite3_column_int(statement, 5);

                [listData addObject:events];

            }

            sqlite3_finalize(statement);
            sqlite3_close(db);

        }

        return listData;
    }
}
```

findById:和modify:我们就不再介绍了。ScheduleDAO类与EventsDAO类也非常相似，读者可以下载我们的源代码查看。

21.4.6 迭代 2.6：发布到GitHub

整个任务2由小贾负责完成，他需要将代码提交给GitHub服务器，此时需要在终端中执行下面的命令：

```
$ git add .
$ git commit -m 'jia commit'
$ git remote add hw git@github.com:516inc/2016Olympics_iOS.git
$ git push hw master
```

执行这些命令可以将本地代码提交到GitHub上，也可能会发生冲突。由于我们的任务分配是基于分层架构的，每个人的任务是独立的，相互之间没有交叉，所以如果不出纰漏的话，应该不会出现版本冲突。关于版本冲突的相关内容，可参考18.2.4节。

21.5 任务 3：业务逻辑层开发

业务逻辑层开发的成员需要熟悉应用的整个业务流程，他可以不了解SQL语句，也不需要知道SQLite API，而是直接调用持久层就可以了。该任务由对业务非常熟悉的老刘负责完成。

业务逻辑类的设计和划分是按照业务模块划分的，其中的方法与用例有关。由于与数据库交互的模块只有比赛项目和比赛日程，所以这个工程中只有比赛项目业务逻辑类和比赛日程业务逻辑类，其类图如图21-14所示。

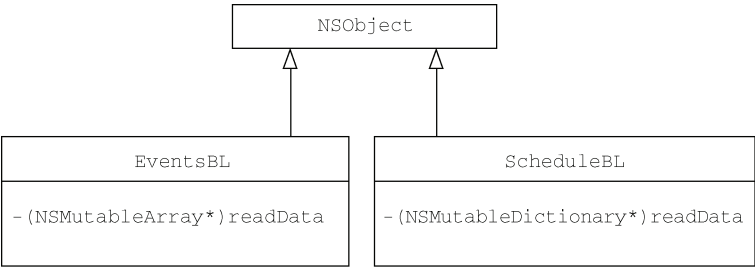


图21-14 业务逻辑层类图

由于目前应用中的业务处理都只是查询数据，然后返回被表示层，所以看起来比较简单。在一些复杂业务处理中，业务逻辑类的工作量应该还是比较大的。

21.5.1 迭代 3.1：比赛项目业务逻辑类GJUnit单元测试

业务逻辑层的编码和单元测试，都依赖于数据持久层。在图21-7所示的任务分配甘特图中，业务逻辑层和数据持久层是并行开发的。这需要开发者编写一些“假的”数据持久层DAO类，它们与真实的DAO类有着相同的接口，区别是“假的”DAO返回的数据在硬编码程序中。这样我们就可以测试业务逻辑层了。

比赛项目业务逻辑测试类EventsBLTests的代码如下：

```
#import <GJUnitiOS/GJUnit.h>
#import "EventsBL.h"
#import "Schedule.h"
#import "EventsDAO.h"

@interface EventsBLTests : GHTestCase {}

@property (nonatomic,strong) EventsBL * bl;
@property (nonatomic,strong) Events * theEvents;

@end

@implementation EventsBLTests

- (void)setUpClass {

    //创建EventsBL对象
```

```

        self.bl = [[EventsBL alloc] init];
        //创建Events对象
        self.theEvents = [[Events alloc] init];
        self.theEvents.EventName = @"test EventName";
        self.theEvents.EventIcon = @"test EventIcon";
        self.theEvents.KeyInfo = @"test KeyInfo";
        self.theEvents.BasicsInfo = @"test BasicsInfo";
        self.theEvents.OlympicInfo = @"test OlympicInfo";

        //插入测试数据
        EventsDAO *dao = [EventsDAO sharedManager];
        [dao create:self.theEvents];
    }

    - (void)tearDownClass {

        //删除测试数据
        self.theEvents.EventID = 41;
        EventsDAO *dao = [EventsDAO sharedManager];
        [dao remove:self.theEvents];

        self.bl = nil;
    }

    - (void)setUp {}

    - (void)tearDown { }

    //测试按照主键查询数据的方法
    - (void) testFindAll
    {
        NSArray* list = [self.bl readData];
        //断言查询记录数为1
        XCTAssertTrue([list count] == 41, @"查询记录数期望值为: 41 实际值为: %i", [list count]);

        Events* resEvents = list[40];
        //断言
        XCTAssertEqualObjects(self.theEvents.EventName, resEvents.EventName,
                               @"比赛项目名称测试失败");
        XCTAssertEqualObjects(self.theEvents.EventIcon, resEvents.EventIcon,
                               @"比赛项目图标测试失败");
        XCTAssertEqualObjects(self.theEvents.KeyInfo, resEvents.KeyInfo,
                               @"项目关键信息测试失败");
        XCTAssertEqualObjects(self.theEvents.BasicsInfo, resEvents.BasicsInfo,
                               @"项目基本信息测试失败");
        XCTAssertEqualObjects(self.theEvents.OlympicInfo, resEvents.OlympicInfo,
                               @"项目奥运会历史信息测试失败");
    }

@end

```

业务逻辑层的测试也需要create_load.sql和DBConfig.plist两个文件，也需要设置Target Membership编译属性。

21.5.2 迭代 3.2：编写比赛项目业务逻辑类

比赛项目业务逻辑类比较简单，其中EventsBL.h文件的代码如下：

```

#import <Foundation/Foundation.h>
#import "EventsDAO.h"
#import "Events.h"

@interface EventsBL : NSObject

//查询所用数据的方法
- (NSMutableArray*) readData;

@end

```

在上述代码中，我们定义了readData方法来从数据持久层查询数据。EventsBL.m文件的代码如下：

```
#import "EventsBL.h"

@implementation EventsBL

// 查询所用数据方法
-(NSMutableArray*) readData
{
    EventsDAO *dao = [EventsDAO sharedManager];

    NSMutableArray* list = [dao findAll];

    return list;
}

@end
```

21.5.3 迭代 3.3：比赛日程业务逻辑类 GHUnit 单元测试

比赛日程业务逻辑测试类ScheduleBLTests的代码如下：

```
#import <GHUnitIOS/GHUnit.h>
#import "ScheduleBL.h"
#import "Schedule.h"
#import "ScheduleDAO.h"

@interface ScheduleBLTests : GHTestCase {}

@property (nonatomic,strong) ScheduleBL * bl;
@property (nonatomic,strong) Schedule * theSchedule;

@end

@implementation ScheduleBLTests

- (void)setUpClass {

    //创建ScheduleBL对象
    self.bl = [ScheduleBL new];
    //创建Schedule对象
    self.theSchedule = [Schedule new];
    self.theSchedule.GameDate = @"test GameDate";
    self.theSchedule.GameTime = @"test GameTime";
    self.theSchedule.GameInfo = @"test GameInfo";
    Events *event = [Events new];
    event.EventName = @"Cycling Mountain Bike";
    event.EventID = 10;
    self.theSchedule.Event = event;

    //插入测试数据
    ScheduleDAO *dao = [ScheduleDAO sharedManager];
    [dao create:self.theSchedule];
}

- (void)tearDownClass {

    //删除测试数据
    self.theSchedule.ScheduleID = 502;
    ScheduleDAO *dao = [ScheduleDAO sharedManager];
    [dao remove:self.theSchedule];

    self.bl = nil;
}
```

①

```

- (void)setUp {}

- (void)tearDown {}

//测试按照主键查询数据的方法
-(void) testFindAll
{
    NSMutableDictionary* dict = [self.bl readData];

    NSArray *allkey = [dict allKeys];

    //断言查询记录数为1
    GHAssertTrue([allkey count] == 18, @"断言比赛天数");

    NSArray* schedules = [dict objectForKey:self.theSchedule.GameDate];
    Schedule* resSchedule = schedules[0];
    //断言
    GHAssertEqualObjects(self.theSchedule.GameDate ,resSchedule.GameDate,
        @"比赛日期测试失败");
    GHAssertEqualObjects(self.theSchedule.GameTime ,resSchedule.GameTime,
        @"比赛时间测试失败");
    GHAssertEqualObjects(self.theSchedule.GameInfo ,resSchedule.GameInfo,
        @"比赛描述测试失败");
    GHAssertEquals(self.theSchedule.Event.EventID ,resSchedule.Event.EventID,
        @"比赛项目ID测试失败");
    GHAssertEqualObjects(self.theSchedule.Event.EventName ,
        resSchedule.Event.EventName,@"比赛项目名测试失败");

}

@end

```

其中第①行代码用于实例化Schedule对象，相当于self.bl = [[Schedule alloc] init]语句。

21.5.4 迭代 3.4：编写比赛日程业务逻辑类

比赛日程业务逻辑类ScheduleBL相对复杂一点。ScheduleBL.h文件的代码如下：

```

#import <Foundation/Foundation.h>
#import "ScheduleDAO.h"
#import "Schedule.h"
#import "EventsDAO.h"
#import "Events.h"

@interface ScheduleBL : NSObject

// 查询所用数据方法
-(NSMutableDictionary*) readData;

@end

```

ScheduleBL.m文件的代码如下：

```

#import "EventsBL.h"
#import "ScheduleBL.h"
@implementation ScheduleBL

// 查询所用数据的方法
-(NSMutableDictionary*) readData
{
    ScheduleDAO *scheduleDAO = [ScheduleDAO sharedManager];

    NSMutableArray* schedules = [scheduleDAO findAll];
    NSMutableDictionary *resDict = [[NSMutableDictionary alloc] init];
    EventsDAO *eventsDAO = [EventsDAO sharedManager];
}

```

①
②

```
//延迟加载Events数据
for (Schedule *schedule in schedules) {
    Events *event = [eventsDAO findById:schedule.Event];
    schedule.Event = event;

    NSArray *allkey = [resDict allKeys];

    //把NSMutableArray结构转化为NSMutableDictionary结构
    if([allkey containsObject:schedule.GameDate]) {
        NSMutableArray* value = [resDict objectForKey:schedule.GameDate];
        [value addObject:schedule];
    } else {
        NSMutableArray* value = [[NSMutableArray alloc] init];
        [value addObject:schedule];
        [resDict setObject:value forKey:schedule.GameDate];
    }
}
return resDict;
}
@end
```

readData方法比较复杂。为了提高查询效率，在第①行查询出来的比赛日程数据并不包含比赛项目信息，而是在获取比赛项目信息数据时再进行加载。而我们的这个业务需要比赛日程和比赛项目信息，因此在第③行循环遍历schedules集合，重新使用[eventsDAO findById:schedule.Event]语句查询比赛项目对象，然后再把比赛信息对象赋值给比赛日程的Event属性（如第⑤行代码所示）。

在readData方法中，除了延迟加载，我们还有一个问题需要解决：把NSMutableArray结构数据转化为NSMutableDictionary结构数据。数据持久层返回的数据是NSMutableArray结构的，返回的部分数据如表21-1所述。

表21-1 NSMutableArray结构数据

ScheduleID	GameDate	GameTime	EventID
1	2016-08-05	16:00 - 20:45 Women's	28
2	2016-08-05	17:10 - 21:45 Women's	37
3	2016-08-05	17:00 - 21:45 Women's	36
4	2016-08-06	12:00 - 16:45 Men's	31
5	2016-08-06	14:30 - 17:15 Men's	30
6	2016-08-06	7:00 - 21:45 Men's	32
7	2016-08-07	9:00 - 21:45 Men's	35
8	2016-08-07	19:45 - 21:45 Men's	33

从表21-1中可见，GameDate（比赛日期）字段的很多记录是重复的，1、2、3记录是2016年8月5日的3场比赛，类似的还有4、5、6和7、8。从存储空间、访问效率和方便访问等几个方面考虑，我们需要的数据结果应该是NSMutableDictionary结构，如表21-2所述。

表21-2 NSMutableDictionary结构数据

ScheduleID	GameDate	GameTime	EventID
1	2016-08-05	16:00 - 20:45 Women's	28
2		17:10 - 21:45 Women's	37
3		17:00 - 21:45 Women's	36
4	2016-08-06	12:00 - 16:45 Men's	31
5		14:30 - 17:15 Men's	30
6		7:00 - 21:45 Men's	32
7	2016-08-07	9:00 - 21:45 Men's	35
8		19:45 - 21:45 Men's	33

如表21-2所述,返回的数据放在NSMutableDictionary中,它的键是GameDate,相同GameDate的数据被放在一个集合中。第⑥行代码判断NSMutableDictionary中是否有相同的GameDate,如果没有,则新建一个NSMutableArray集合,如第⑧行代码所示。把当前的比赛日程对象schedule添加到NSMutableArray集合,如第⑨行代码所示。第⑩行代码将GameDate作为键,把集合NSMutableArray数据放到字典NSMutableDictionary中。

如果有相同的GameDate,取出NSMutableArray集合,把当前的比赛日程对象schedule添加到集合NSMutableArray中。

21.5.5 迭代 3.5: 发布到GitHub

整个任务3由老刘负责完成,他需要将代码提交给GitHub服务器,此时在终端执行下面的命令即可:

```
$ git add .
$ git commit -m 'jia commit'
$ git remote add hw git@github.com:516inc/2016Olympics_iOS.git
$ git push hw master
```

如果出现版本冲突,请参考18.2.4节查看并解决。

21.6 任务 4: 表示层开发

从客观上讲,表示层开发的工作量是很大的,不仅有很多细节工作,而且还要做两套表示层,即iPhone和iPad。该任务是由对UIKit非常熟悉的小李负责完成的。

表示层实现的方式有两种:一种是在同一个工程中包含两套UI,就是故事板或者nib文件有两套,发布的时候是一个应用;另一种是对于两个工程(iPhone是一个工程,iPad是另外一个工程),会有两个不同的应用。这里我们采用的是第一种方式。

我们把不同的视图控制器和故事板文件放在不同组中。如图21-15所示是表示层的导航面板,我们在其中创建了iPad和iPhone组。

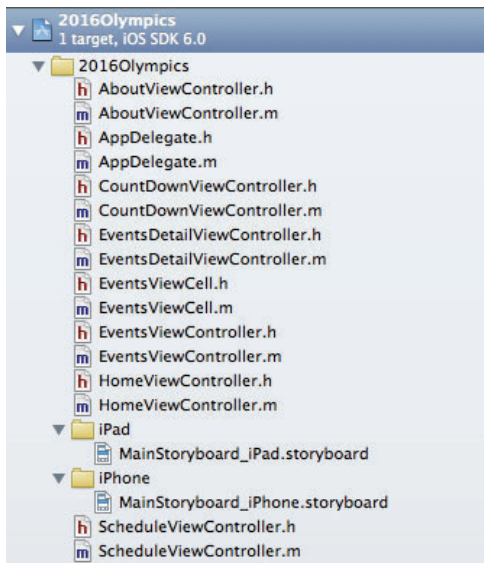


图21-15 表示层的导航面板

表21-3简要介绍了表示层的主要组件。

表21-3 表示层的主要组件

组 件 名	说 明
AppDelegate	应用程序委托对象
HomeController	首页模块视图控制器类
EventsViewController	比赛项目模块视图控制器类
EventsTableViewCell	比赛项目模块视图中使用的Collection视图中的单元格类
EventsDetailViewController	比赛项目详细模块视图控制器类
ScheduleViewController	比赛日程表模块视图控制器类
CountDownViewController	倒计时模块视图控制器类
AboutViewController	关于我们模块视图控制器类
MainStoryboard_iPad.storyboard	iPad版的故事板文件
MainStoryboard_iPhone.storyboard	iPhone版的故事板文件

21.6.1 迭代 4.1：根据原型设计初步设计iPad故事板

我们在设计应用原型的时候曾经绘制了原型设计图（见图21-2），它对于表示层开发非常重要。通过原型设计图，我们可以了解界面中有哪些UI元素以及应用模块中界面之间的跳转关系。所以，原型设计图可以帮助我们绘制故事板。苹果公司推出故事板技术的一个目的也是便于开发人员查看界面之间的跳转关系。

但是随着业务复杂程度的增加，故事板会变得越来越大，这也是故事板的一个缺点。图21-16是2016奥运会应用的iPad故事板。

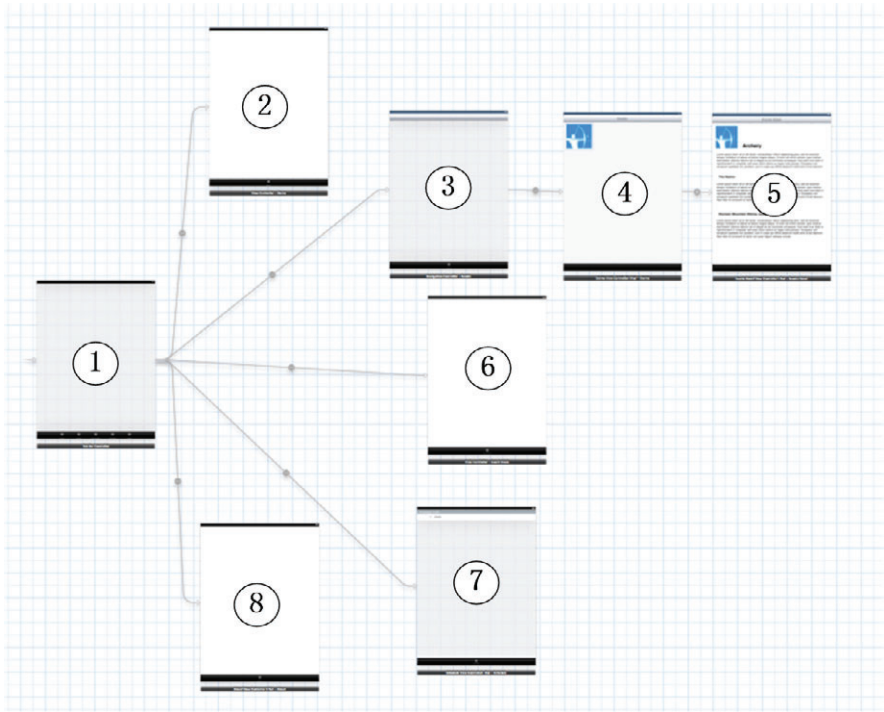


图21-16 2016奥运会应用的iPad故事板

从图21-16可见，每个单页面没法看清楚，于是我加上标号，通过标号给读者解释一下。

- ❑ ①：应用的根视图控制器，类型是UITabBarController。
- ❑ ②：首页的视图控制器，类型是UIViewController。
- ❑ ③：比赛项目模块的根视图控制器，类型是UINavigationController。
- ❑ ④：比赛项目模块的比赛项目Collection视图控制器，类型是UICollectionViewController。
- ❑ ⑤：比赛项目模块的比赛项目详细信息视图控制器，类型是UIViewController。
- ❑ ⑥：比赛日程表模块的视图控制器，类型是UITableViewController。
- ❑ ⑦：倒计时模块的视图控制器，类型是UIViewController。
- ❑ ⑧：关于我们模块的视图控制器，类型是UIViewController。

21.6.2 迭代 4.2：根据原型设计初步设计 iPhone 故事板

由于这个应用在iPhone和iPad上的功能基本一样，它们的设计原型图基本一样，所以iPhone版故事板和iPad版故事板也基本上一样。图21-17是iPhone版的故事板。

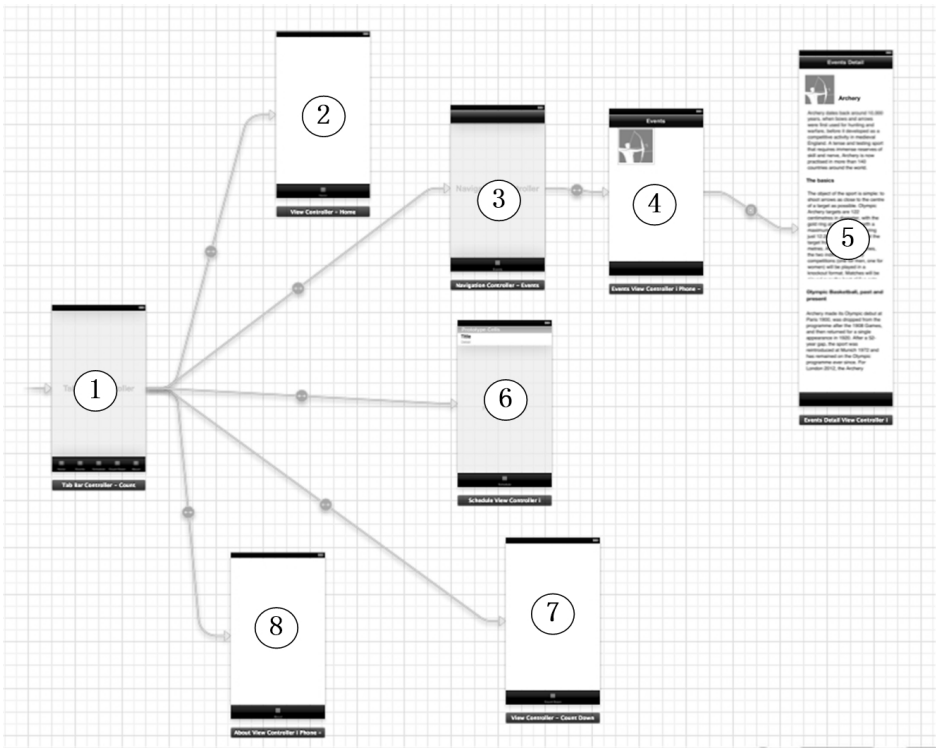


图21-17 2016奥运会应用的iPhone故事板

图中标号的解释与图21-16标号的解释是一样的。接下来，我们分别介绍各个模块的实现过程。

21.6.3 迭代 4.3：首页模块

首页模块只有一个界面，其中图21-18是iPad版的首页界面，图21-19是iPhone版的首页界面。视图控制器是HomeViewController类，它继承UIViewController类。按照如图21-18和图21-19所示的界面UI元素在iPad和iPhone故事板中进行设计。



图21-18 2016奥运会应用的iPad首页



图21-19 2016奥运会应用的iPhone首页

完成UI设计之后，在工程中添加HomeController类。首页只是展示一张图片，很简单，我们不用添加代码。

21.6.4 迭代 4.4：比赛项目模块

比赛项目模块有两个界面，图21-20是iPad版的Collection视图界面（左图）和详细视图界面（右图）。图21-21是iPhone版的Collection视图界面（左图）和详细视图界面（右图）。

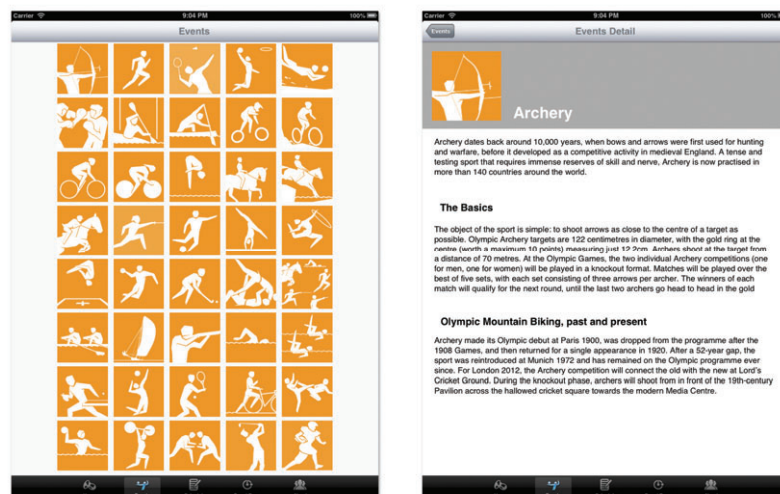


图21-20 iPad版比赛项目模块界面

按照图21-20和图21-21所示的界面在故事板中进行设计，这里我们重点介绍iPad版Collection视图界面的实现过程。用Interface Builder打开故事板文件中的Events视图，然后从对象库中拖曳Collection View Cell控件到设计界面中，再拖曳一个Image View控件到Collection View Cell控件中，如图21-22所示。



图21-21 iPhone版比赛项目模块界面

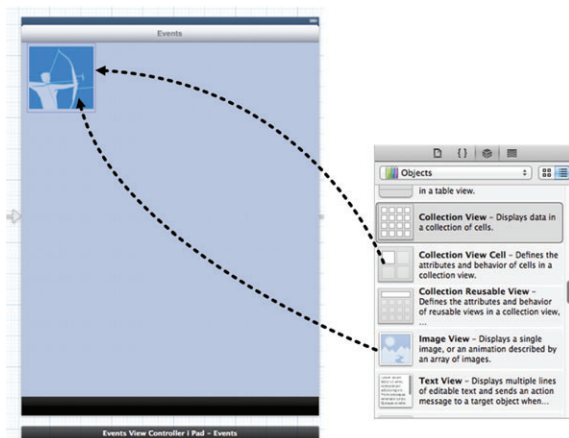





图21-22 iPad版比赛项目Collection视图设计界面

然后选中Collection View, 打开它的尺寸检查器, 如图21-23所示, 从中设置它的各个属性值。其中Cell Size 用于设置决定Collection View界面中一行有几个单元格。Section Insets可以调整行的上、下、左、右边距。通过调整这些参数, 可以调整Collection视图的界面布局。

选择Events View Cell, 打开其标识检查器, 如图21-24所示, 设置Custom Class→Class为EventsViewCell。再打开其属性检查器中的Collection Reusable View→Identifier, 输入Cell。

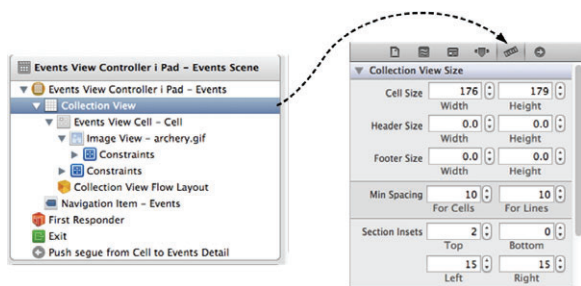


图21-23 Collection View尺寸属性

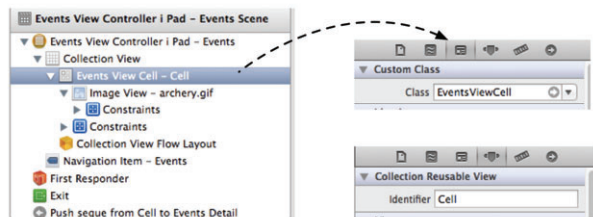


图21-24 Collection View Cell属性

单元格是我们自己定义的EventsViewCell类, 需要为里面的ImageView定义输出口。如图21-25所示, 在设计界面中选中ImageView, 然后用鼠标将其拖曳到右边代码窗口的imageView属性上。

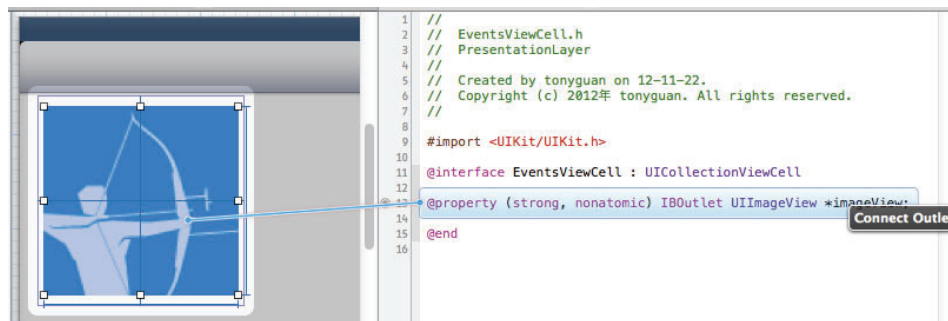


图21-25 为UIImageView定义输出口

下面我们再看看代码部分。在该模块中,EventsViewController类继承于UICollectionViewController类,EventsDetailViewController类继承于UIViewController类。EventsViewController.h文件的代码如下:

```
#import <UIKit/UIKit.h>
#import "EventsViewCell.h"
#import "EventsBL.h"
#import "Events.h"
#import "EventsDetailViewController.h"

@interface EventsViewController : UICollectionViewController
{
    //一行中的列数
    NSUInteger COL_COUNT;
}

@property (strong, nonatomic) NSArray * events;

@end
```

EventsViewController.m文件中初始化方法的代码如下:

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

    if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPhone) {
        //如果是iPhone设备,列数为2
        COL_COUNT = 2;
    } else {
        //如果是iPad设备,列数为5
        COL_COUNT = 5;
    }
}

- (void)viewWillAppear:(BOOL)animated
{
    if (self.events == nil || [self.events count] == 0) {
        EventsBL* bl = [[EventsBL alloc] init];
        //获取全部数据
        NSMutableArray *array = [bl readData];
        self.events = array;
        [self.collectionView reloadData];
    }
}
```

在viewDidLoad方法中,需要判断设备类型是iPhone还是iPad,然后设定每一行有几列。查询处理在viewWillAppear:方法中进行,而没有在viewDidLoad方法中处理,这是由于数据库的初始化是另外一个线程处理的,很有可能在用户进入界面时数据库还没有初始化完成,而viewDidLoad方法只有在界面初始化的时候才调用,因此调用viewDidLoad方法时很可能没有数据。

下面我们再看看数据源UICollectionViewDataSource的实现方法,主要的代码如下:

```
- (NSInteger)numberOfSectionsInCollectionView:(UICollectionView *)
collectionView
{
    return [self.events count] / COL_COUNT;
}
①

- (NSInteger)collectionView:(UICollectionView *)collectionView
numberOfItemsInSection:(NSInteger)section
{
    return COL_COUNT;
}
②

- (UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
```



```

        cellForItemAtIndexPath:(NSIndexPath *)indexPath                                ③
    {
        EventsTableViewCell *cell = [collectionView dequeueReusableCellWithReuseIdentifier:
            @"Cell" forIndexPath:indexPath];
        Events *event = [self.events objectAtIndex:(indexPath.section *
            COL_COUNT + indexPath.row)];
        cell.imageView.image = [UIImage imageNamed:event.EventIcon];
        return cell;
    }

```

第①行代码中的`numberOfSectionsInCollectionView:`方法用来指定Collection视图中节的个数,即有多少行。第②行代码中的`collectionView:numberOfItemsInSection:`方法用来指定Collection视图一节中有多少项,即一行中有多少单元格。第③行代码为每一个单元格提供显示数据。

当点击单元格时,界面会跳转到详细界面,它的处理代码如下:

```

- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    if ([[segue identifier] isEqualToString:@"showDetail"]) {

        NSIndexPath *indexPath = [[self.collectionView
            indexPathsForSelectedItems] objectAtIndex:0];
        Events *event = [self.events objectAtIndex:(indexPath.section *
            COL_COUNT + indexPath.row)];
        EventsDetailViewController_iPad *detailVC = [segue
            destinationViewController];
        detailVC.event = event;

    }
}

```

`prepareForSegue:segue:`方法在视图控制器中的Segue被执行的时候触发。进入详细界面,也会触发Segue,但是我们要判断Segue的`identifier`为`showDetail`。`showDetail`是在故事板中设置Segue的`identifier`属性。

EventsDetailViewController.h文件的代码如下:

```

#import <UIKit/UIKit.h>
#import "Events.h"

@interface EventsDetailViewController : UIViewController

@property(nonatomic, strong) Events *event;

@property (weak, nonatomic) IBOutlet UILabel *lblEventName;
@property (weak, nonatomic) IBOutlet UIImageView *imgEventIcon;
@property (weak, nonatomic) IBOutlet UITextView *txtViewKeyInfo;
@property (weak, nonatomic) IBOutlet UITextView *txtViewBasicsInfo;
@property (weak, nonatomic) IBOutlet UITextView *txtViewOlympicInfo;

@end

```

`event`属性是上一个界面传递过来的比赛项目数据。

EventsDetailViewController.m文件中初始化方法的代码如下:

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.imgEventIcon.image = [UIImage imageNamed:self.event.EventIcon];

    self.lblEventName.text = self.event.EventName;
    self.txtViewBasicsInfo.text = self.event.BasicsInfo;
    self.txtViewKeyInfo.text = self.event.KeyInfo;
    self.txtViewOlympicInfo.text = self.event.OlympicInfo;
}

```

在Collection视图中需要自定义单元格，EventsViewCell类是我们自定义的单元格类。EventsViewCell.h文件的代码如下：

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

@interface EventsViewCell : UICollectionViewCell

@property (strong, nonatomic) IBOutlet UIImageView *imageView;

@end
```

其中的自定义单元格类继承UICollectionViewCell类，这是UIView而非UIViewController的一个子类。此外，我们还定义了UIImageView的输出口属性imageView。根据实际情况，单元格中还可以包含其他的控件。此外，我们还需要在这里定义输出口。

EventsViewCell.m文件中的代码如下：

```
#import "EventsViewCell.h"

@implementation EventsViewCell

- (id)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {

    }
    return self;
}

@end
```

initWithFrame:方法是单元格构造方法，它也是UIView子类的默认构造方法。就我们这个应用而言，构造方法中没有其他的实现代码，基本上都是由Xcode模板生成的代码。

21.6.5 迭代 4.5：比赛日程模块

比赛日程模块有一个界面，其中图21-26是iPad版的界面，图21-27是iPhone版的界面。

按照图21-26和图21-27所示的界面UI元素在故事板中进行设计。两个版本界面使用的是表视图，但是它们的单元格样式是不同的，在设计时需要注意。

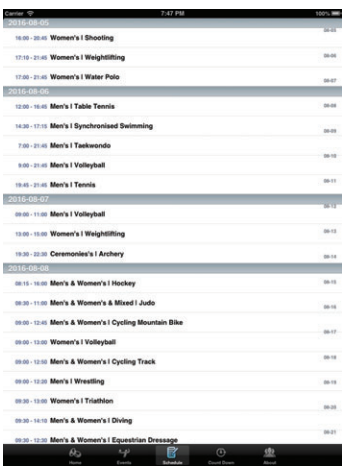


图21-26 iPad版的比赛日程界面

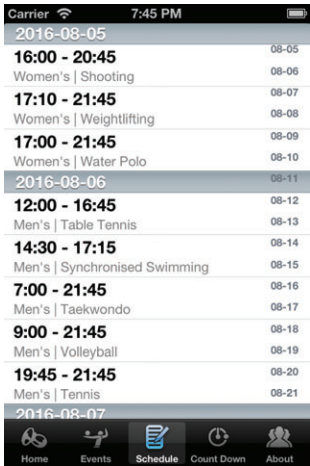


图21-27 iPhone版的比赛日程界面

下面我们看看代码部分。在该模块中，ScheduleViewController类继承自UITableViewController类。ScheduleViewController.h文件的代码如下：

```
#import <UIKit/UIKit.h>
#import "Schedule.h"
#import "ScheduleBL.h"
#import "EventsDetailViewController.h"

@interface ScheduleViewController : UITableViewController

//表视图使用的数据
@property (strong, nonatomic) NSDictionary * data;
//比赛日期列表
@property (strong, nonatomic) NSArray * arrayGameDateList;

@end
```

data属性包含了从数据返回的所有数据，arrayGameDateList属性是从data属性中提取出日期的集合。

ScheduleViewController.m文件中初始化方法的代码如下：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
}
- (void)viewWillAppear:(BOOL)animated
{
    if (self.data == nil || [self.data count] == 0) {
        ScheduleBL *bl = [ScheduleBL new];
        self.data = [bl readData];
        NSArray* keys = [self.data allKeys];
        //对key进行排序
        self.arrayGameDateList = [keys
            sortedArrayUsingSelector:@selector(compare:)];
    }
}
```

初始化查询处理也放在viewWillAppear:方法中，其原因与比赛项目模块是一样的。从业务逻辑层返回的数据是NSDictionary类型，放在属性data中，然后再从data属性取出所有的键，这些键事实上就是所有的比赛日期集合。在赋值给arrayGameDateList属性前要进行排序，因为NSDictionary结构中的数据是无序的。

ScheduleViewController.m文件中表视图数据源的实现方法如下：

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    NSArray* keys = [self.data allKeys];
    return [keys count];
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection:
(NSInteger)section
{
    //比赛日期
    NSString* strGameDate = [self.arrayGameDateList objectAtIndex:section];
    //比赛日期下的比赛日程表
    NSArray *schedules = [self.data objectForKey:strGameDate];
    return [schedules count];
}

- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section
{
    //比赛日期
    NSString* strGameDate = [self.arrayGameDateList objectAtIndex:section];
    return strGameDate;
}
```

```

- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    //比赛日期
    NSString* strGameDate = [self.arrayGameDateList
        objectAtIndex:indexPath.section];
    //比赛日期下的比赛日程表
    NSArray *schedules = [self.data objectForKey:strGameDate];
    Schedule *schedule = [schedules objectAtIndex:indexPath.row];

    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        CellIdentifier forIndexPath:indexPath];
    NSString* subtitle = [[NSString alloc] initWithFormat:@"%@" | %@",
        schedule.GameInfo, schedule.Event.EventName];
    cell.textLabel.text = schedule.GameTime;
    cell.detailTextLabel.text = subtitle;

    return cell;
}

- (NSArray *) sectionIndexTitlesForTableView: (UITableView *) tableView ①
{
    NSMutableArray *listTitles = [[NSMutableArray alloc] init];
    //2016-08-09 -> 08-09
    for (NSString *item in self.arrayGameDateList) {
        NSString *title = [item substringFromIndex:5]; ②
        [listTitles addObject:title];
    }
    return listTitles;
}

```

在第①行代码中, sectionIndexTitlesForTableView:用于为表视图添加索引, 返回一个索引的数组集合。但是如果我们直接返回arrayGameDateList属性, 显示表视图索引时是2016-08-09, 这样的索引标题太长了, 没有必要显示2016这个年份了。因此, 我们在第②行代码中进行遍历, 将集合截取为08-09格式。

21.6.6 迭代 4.6: 倒计时模块表示层

倒计时模块只有一个界面, 其中图21-28是iPad版的倒计时界面, 图21-29是iPhone版的倒计时界面。视图控制器是CountDownViewController类, 它继承UIViewController类。按照图21-28和图21-29所示的界面UI元素在iPad和iPhone的故事板中进行设计。



图21-28 iPad版的倒计时界面



图21-29 iPhone版的倒计时界面

完成UI设计之后，在工程中添加CountDownViewController类。CountDownViewController.h文件的代码如下：

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

@interface CountDownViewController : UIViewController

//显示倒计时牌个位数字Label
@property (weak, nonatomic) IBOutlet UILabel *lblOnesPlace;
//显示倒计时牌十位数字Label
@property (weak, nonatomic) IBOutlet UILabel *lblTensPlace;
//显示倒计时牌百位数字Label
@property (weak, nonatomic) IBOutlet UILabel *lblHundredsPlace;
//显示倒计时牌千位数字Label
@property (weak, nonatomic) IBOutlet UILabel *lblThousandsPlace;

@end
```

CountDownViewController.m中的代码如下：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    //创建NSDateComponents对象
    NSDateComponents *comps = [[NSDateComponents alloc] init];           ①
    //设置NSDateComponents中的日期
    [comps setDay:5];                                                     ②
    //设置NSDateComponents中的月份
    [comps setMonth:8];                                                   ③
    //设置NSDateComponents中的年份
    [comps setYear:2016];                                                 ④
    //创建日历对象
    NSCalendar *calender = [[NSCalendar alloc]
        initWithCalendarIdentifier:NSGregorianCalendar];                 ⑤
    //获得2016-8-5的NSDate日期对象
    NSDate *destinationDate = [calender dateFromComponents:comps];       ⑥
    //获得当前日期到2016-8-5时间的NSDateComponents对象
    NSDateComponents *components = [calender components:NSDayCalendarUnit fromDate:
        [NSDate date] toDate:destinationDate options:0];               ⑦
    //获得当前日期到2016-8-5相差的天数
    int days = [components day];                                           ⑧

    self.lblOnesPlace.text = [NSString stringWithFormat:@"%i", (days / 10)];
    self.lblTensPlace.text = [NSString stringWithFormat:@"%i", (days % 100 / 10)];
    self.lblHundredsPlace.text = [NSString stringWithFormat:@"%i", (days % 1000 / 100)];
    self.lblThousandsPlace.text = [NSString stringWithFormat:@"%i", (days % 10000 / 1000)];
}
```

第①行代码用于创建NSDateComponents对象。NSDateComponents类是一个封装了日期的可扩展组件，通过这个类可以计算出两个日期的年、月、日、时、分、秒的差别。第②行、第③行、第④行代码用于设置日期可扩展组件类的日、月、年。

第⑤行代码用于创建NSCalendar日历对象。NSCalendar封装了系统时间类，用于与时间有关的计算，其中initWithCalendarIdentifier:构造方法通过一个日历标识创建日历对象。在本应用中，使用的日历标识是NSGregorianCalendar，它代表格里历（世界上比较通用的日历）。此外，还有很多日历，如NSChineseCalendar、NSJapaneseCalendar和NSHebrewCalendar等。

第⑥行代码借助日历对象的dateFromComponents:方法从可扩展组件获得日期的NSDate对象。这个NSDate代表的时间是2016年8月5日，即2016里约热内卢奥运会的开幕时间。

第⑦行代码又创建了一个NSDateComponents对象。与第①行代码不同，它通过指定开始时间和结束时间来创建。在构造方法components:fromDate:toDate:中，components部分指定时间返回标志，这些标志还有：

- ❑ NSYearCalendarUnit
- ❑ NSMonthCalendarUnit
- ❑ NSDayCalendarUnit
- ❑ NSHourCalendarUnit
- ❑ NSMinuteCalendarUnit
- ❑ NSSecondCalendarUnit

本例是NSDayCalendarUnit，只计算相差的天数。如果对两个日期的年、月、日、时、分、秒的差别进行位运算，可以使用如下代码：

```
int units = NSYearCalendarUnit | NSMonthCalendarUnit | NSDayCalendarUnit
          | NSHourCalendarUnit | NSMinuteCalendarUnit | NSSecondCalendarUnit;
```

第⑧行代码用于计算相差的天数。类似地，[components year]用于获得相差的年数，[components month]用于获得相差的月数。

21.6.7 迭代 4.7：关于我们模块表示层

关于我们模块只有一个界面，其中图21-30是iPad版的关于我们界面，图21-31是iPhone版的关于我们界面。视图控制器是AboutViewController类，它继承UIViewController类。按照图21-30和图21-31所示的界面UI元素在iPad和iPhone故事板中进行设计。



图21-30 iPad版的关于我们界面



图21-31 iPhone版的关于我们界面

完成UI设计之后，在工程中添加AboutViewController类。由于关于我们界面比较简单，可以考虑添加广告。添加广告的过程我们会在下一节中介绍。

21.6.8 迭代 4.8：发布到GitHub

整个任务4由小李负责完成，他需要将代码提交给GitHub服务器，此时在终端执行下面的命令即可：

```
$ git add .
$ git commit -m 'jia commit'
$ git remote add hw git@github.com:516inc/2016Olympics_iOS.git
$ git push hw master
```

如果出现版本冲突，请参考18.2.4节查看并解决。

21.7 任务 5: 收工

程序编写到现在,基本上已经完成了,但还有最后一点工作需要完成,这些工作包括图标和启动界面的添加、广告添加、测试性能、提交代码到GitHub以及在App Store上发布应用。

21.7.1 迭代 5.1: 添加图标

图标是非常重要的,我们的UI设计师绘制好了一个图标,但是这个应用有iPad版本和iPhone版本,并且同时在一个工程中。因此,我们需要准备的图标有4个。

- ❑ iPhone和iPod touch普通分辨率设备。文件命名为Icon.png,分辨率为 57×57 。
- ❑ iPhone和iPod touch高分辨率设备。文件命名为Icon@2x.png,分辨率为 114×114 。
- ❑ iPad普通分辨率设备。文件命名为Icon-72.png,分辨率为 72×72 。
- ❑ iPad高分辨率设备。文件命名为Icon-72@2x.png,分辨率为 144×144 。

准备完成这些图标之后,请参考19.1.1节将图标添加到工程中。

21.7.2 迭代 5.2: 设计和添加启动界面

添加启动界面的原则是以用户为中心。这个应用的第一个屏幕是首页界面。由于我们的应用只考虑竖屏的情况,所以只需要准备5个图标即可。

- ❑ iPhone 5和第5代iPod touch设备。将文件命名为Default-568h@2x.png,分辨率为 1136×640 。
- ❑ iPhone和iPod touch高分辨率设备。将文件命名为Default@2x.png,分辨率为 960×640 。
- ❑ iPhone和iPod touch普通分辨率设备。将文件命名为Default.png,分辨率为 480×320 。
- ❑ iPad普通分辨率设备。将文件命名为Default-Portrait~ipad.png,分辨率为 1004×768 。
- ❑ iPad高分辨率设备。将文件命名为Default-Portrait~ipad@2x.png,分辨率为 2008×1536 。

需要注意的是,iPad的启动界面是不包含状态栏的(如图21-32所示),而iPhone和iPod touch是包含状态栏的(如图21-33所示)。

准备完这些启动界面之后,请参考19.1.2节将启动界面添加到工程中。



图21-32 2016奥运会的iPad版启动界面



图21-33 2016奥运会的iPhone版启动界面

21.7.3 迭代 5.3：植入谷歌AdMob横幅广告

这个应用是免费使用的，但我们还是希望能从这个应用中得到一些收入，此时可以在应用中植入广告。我们决定在关于我们这个模块中植入广告，因为这个模块不是主要功能模块，基本上不影响用户使用。在广告选择方面，我们选择了谷歌AdMob横幅广告，这主要考虑到苹果iAd广告支持的国家不包括中国。植入AdMob广告后的关于我们界面如图21-34所示。



图21-34 植入AdMob广告后的关于我们界面

要添加AdMob广告，首先需要将谷歌提供的AdMob库导入都工程中并配置好环境，具体步骤请参考14.3节。然后，修改2016Olympics工程的AboutViewController类。AboutViewController.h文件的代码如下：

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

#define kSampleAdUnitID @"a14df1974738141"

@interface AboutViewController : UIViewController <GADBannerViewDelegate>

@property (strong, nonatomic) GADBannerView *bannerView;

- (GADRequest *)createRequest;

@end
```

AboutViewController.m文件的主要代码如下：

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

    // 设定广告栏屏幕尺寸，实例化GADBannerView
    self.bannerView = [[GADBannerView alloc] initWithAdSize:kGADAdSizeBanner];
    // 设置应用发布者ID
    self.bannerView.adUnitID = kSampleAdUnitID;
    // 设置委托
    self.bannerView.delegate = self;
```

```

//设置广告栏的根视图控制器
[self.bannerView setRootViewController:self];
//竖屏情况下设置广告栏的位置
self.bannerView.center = CGPointMake(self.view.center.x,
    kGADAdSizeBanner.size.height / 2);
[self.view addSubview:self.bannerView];
//请求加载广告
[self.bannerView loadRequest:[self createRequest]];
}

//创建广告请求
- (GADRequest *)createRequest {
    GADRequest *request = [GADRequest request];
    return request;
}

#pragma mark GbannerViewViewDelegate实现

//广告接收成功
- (void)adViewDidReceiveAd:(GADBannerView *)adView {
    NSLog(@"广告接收成功");
}
//广告接收失败
- (void)adView:(GADBannerView *)view didFailToReceiveAdWithError:
    (GADRequestError *)error {
    NSLog(@"广告接收失败 %@", [error localizedFailureReason]);
    //重新请求加载广告
    [self.bannerView loadRequest:[self createRequest]];
}

```

21.7.4 迭代 5.4：性能测试与改善

到现在为止，2016奥运会应用的主要功能基本实现了，此时可以在设备上测试一下。在性能、UI布局 and 用户体验等诸多方面，设备可能与模拟器不同，而且设备之间也有比较大的不同。如果有条件，凡是支持iOS 6的所有型号的设备都应该拿来测试一下，这主要考虑用户设备的多样化。

我们对这个应用使用几个设备进行了测试，实际测试的结果是我们发现了问题。当第一次进入比赛项目或比赛日程表界面时，界面比较“卡”。经过分析发现，第一次查询使用数据库时，是由DAO对象创建数据库并添加初始化数据的，这个时间比较长。解决方法是把这些数据库的初始化放到子线程中处理，这个初始化只进行一次，除非数据库升级。因此，我们可以考虑在应用程序委托对象AppDelegate中进行初始化，主要是在启动界面加载时启动一个子线程，开始初始化数据库。在AppDelegate.m中添加的代码如下：

```

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    dispatch_queue_t queue = dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_async(queue, ^ {
        //加载并初始化数据库
        DBHelper *dbhelper = [DBHelper new];
        [dbhelper initDB];
    });
    return YES;
}

```

这里我们使用GCD方式异步启动子线程，[dbhelper initDB]语句用于初始化数据库。在这个方法中，我们会判断数据是否需要更新，如果需要才去更新。

21.7.5 迭代 5.5：发布到GitHub

整个任务5是由老关负责完成的，他需要将代码提交给GitHub服务器，此时在终端执行下面的命令即可：

```
$ git add .
$ git commit -m 'jia commit'
$ git remote add hw git@github.com:516inc/2016Olympics_iOS.git
$ git push hw master
```

如果出现版本冲突，请参考18.2.4节查看并解决。

21.7.6 迭代 5.6：在App Store上发布应用

我们还需要进行一些其他的测试（如综合测试和用户测试等）才能在App Store上发布应用，但这些与本书关系不是很大，这里就不再介绍了。

现在，我们可以在App Store上发布这个应用了，发布流程可参考19.2节。按照图19-28所示，发布流程分为8个步骤，下面我们按照这8个步骤介绍发布过程。

1. 创建App ID

创建App ID是在iOS开发中心的配置门户网站完成的，输入的内容如图21-35所示。

Create App ID

Description

Enter a common name or description of your App ID using alphanumeric characters. The description you specify will be used throughout the Provisioning Portal to identify this App ID.

2016Olympics You cannot use special characters as @, &, *, " in your description.

Bundle Seed ID (App ID Prefix)

Use your Team ID or select an existing Bundle Seed ID for your App ID.

Use Team ID If you are creating a suite of applications that will share the same Keychain access, use the same bundle Seed ID for each of your application's App IDs.

Bundle Identifier (App ID Suffix)

Enter a unique identifier for your App ID. The recommended practice is to use a reverse-domain name style string for the Bundle Identifier portion of the App ID.

com.51work6.-016Olympics Example: com.domainname.appname

Cancel Submit

图21-35 创建App ID

在Description输入框中输入2016Olympics，在Bundle Identifier (App ID Suffix)输入框中输入com.51work6.-016Olympics，这与工程中的Bundle Identifier一致。由于2016Olympics这个名字以数字开头，Xcode会将数字替换为-。

2. 创建应用及基本信息

创建应用及基本信息是在iTunes Connect中进行的。登录成功后，进入创建应用界面，从中输入应用的相关信息，如图21-36所示。

3. 应用定价信息

点击图21-36的Continue按钮后，进入应用定价信息页面，从中选择Price Tier为Free，如图21-37所示。

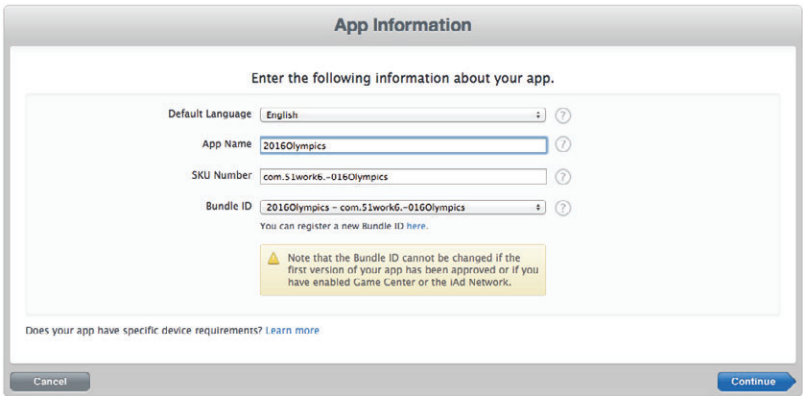


图21-36 创建应用及基本信息

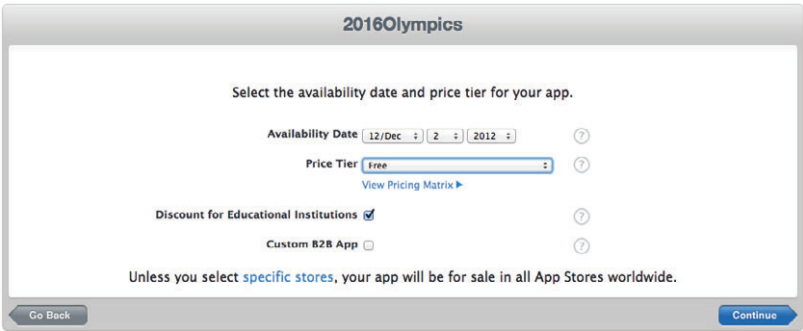


图21-37 应用定价信息

4. 最后信息输入

点击图21-37中的Continue按钮，进入最后信息输入页面，其中包括版本信息、评级信息、元数据、应用审核信息、最终用户许可协议以及上传应用图标和截图。这里要输入的信息很多，大家可以参考19.2.3节。输入完成信息之后，应用的状态是Prepare for Upload（准备上传）状态。

5. 准备上传

回到应用管理界面，点击2016Olympics应用，进入详细界面，如图21-38所示。点击右上角Ready to Upload Binary按钮，进入出口规定页面，从中选择No单选按钮，接着点击Save按钮，然后根据页面向导一步步完成操作，最后进入Waiting For Upload（等待上传）状态。



图21-38 2016Olympics应用详细界面

6. 发布编译

在上传之前要对应用进行发布编译，但这并不是简单的编译。需要先在iOS开发中心的配置门户网站创建发布配置概要文件，使用发布配置概要文件才能编译，具体步骤请参考19.1.4节。

7. 应用打包

编译完成之后，需要给编译之后的包文件进行打包，具体步骤请参考19.1.5节。打包之后的文件是2016Olympics.zip。

8. 上传应用

把压缩好的2016Olympics.zip文件通过Application Loader 工具上传到App Store，具体步骤请参考19.2.4节。上传成功后，我们会看到如图21-39所示的界面。

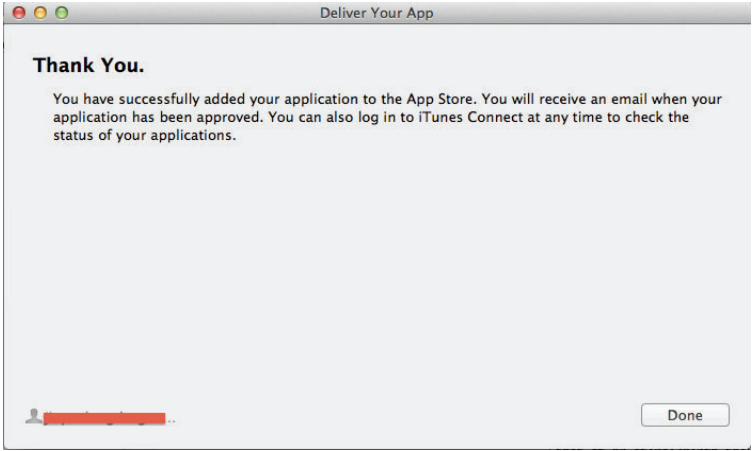


图21-39 上传成功

回到iTunes Connect中的应用管理，查看2016Olympics应用的详细信息，此时会看到应用处于Waiting For Review（等待审核）状态，如图21-40所示。

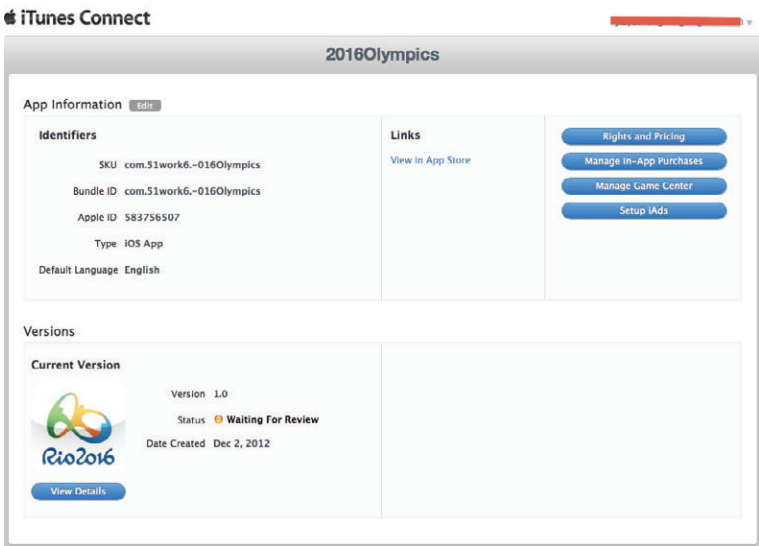


图21-40 等待审核

审核过程需要等待苹果团队进行。当这个应用的状态变成Ready for Sale（可以销售）了，就说明审核通过了。

21.8 小结

本章介绍了完整的iOS应用分析与设计、编程、测试和发布过程，这里采用的架构是分层设计的，采用的开发方法是敏捷开发方法。敏捷开发方法非常适合于iOS开发，希望广大读者能够认真学习。

图灵原创

- cocos2d-x 手机游戏开发:
跨 iOS、Android 和沃 Phone 平台
- 论道 HTML5
- Go 语言 · 云动力
- 推荐系统实践
- Unity 3D 游戏开发
- 大道至易：实践者的思想
- 思考的乐趣：Matrix67 数学笔记
- Node.js 开发指南
- Go 语言编程
- DBA 的思想天空——感悟 Oracle 数据库本质
- Kinect 人机交互开发实践
- 深入浅出 PhoneGap
- Cocos2d-x 高级开发教程：制作自己的《捕鱼达人》
- iOS 开发指南：从零基础到 App Store 上架
- 我是设计师
- Android 软件安全与逆向分析
- 腾云：云计算和大数据时代网络技术揭秘

若有写作意向，请联系我：

wangjh.turing@gmail.com

新浪微博 @ 图灵小花

在移动互联网飞速发展的今天，移动终端是兵家必争之地，无论你是做网络 / 通信还是做内容创意，无论你是设备厂商还是 App 开发者，都需要一本“平易近人”的书，陪你走在移动服务广阔的原野上。对于一位初学者而言，本书让你踏出 iOS 学习的第一步，从 iOS SDK 开始，到产品完善、上线（如发布到 App Store 上）都能一步到位，并最终成为 iOS 平台上的赢家。对于 Android 或 Windows 8 的开发者而言，本书完善的示例代码和分类，将非常有助于多平台对比开发和平行同步开发，从而协助大家在移动互联网时代能同时掌握多个主流平台。

高焕堂，亚太 Android 领域开发联盟总架构师

曾与关老师就智慧型手机与移动终端交换很多意见，并在后续与中国移动的合作案中，有幸与他一同参与，从中见识到关老师在移动终端开发与推广的热情。这是一本全面介绍 iOS 应用开发的图书，包括了 iOS 6 的相关内容，通俗易懂，深入浅出。对于 iOS 初学者以及需要提高的读者来说，这是一本优秀的学习参考书。本书不仅从理论出发，还提供了大量实战案例，相信对于移动应用开发者来说，这会是一本不可或缺的经典好书。

柯博文，美国硅谷 LoopTek 公司 CTO，《大富翁》游戏开发者

移动终端的发展日新月异，移动互联是未来趋势，本书结合大量案例讲述 iOS 应用开发技巧，便于读者掌握。尤其在进阶篇中，作者分享了自己在开发过程中的经验和心得。本书不仅从理论方面出发，还提供了大量实战案例。我们相信本书在你的 iOS 开发之路上会助你一臂之力。

智捷 iOS 课堂

关老师不仅是 51CTO.com 的专家博主，而且是 51CTO 关注移动开发的朋友们最为喜爱的一位博主。本书无疑是一本值得 iOS 初学者认真阅读的从入门到精通的教材。通过对本书的学习，你可以从一个对移动开发一窍不通的小白，逐步成长为一名 iOS 开发高手。当你看到自己的 App 上架时，你所得到的喜悦绝对会超出这本书自身的售价。

赵磊，51CTO 副总编



51CTO.com
技术成就梦想

图灵社区：www.ituring.com.cn

新浪微博：[@图灵教育](#) [@图灵社区](#)

反馈 / 投稿 / 推荐邮箱：contact@turingbook.com

热线：(010) 51095186 转 604

分类建议 计算机 / 移动开发 / iOS

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-32444-3



9 787115 324443 >

ISBN 978-7-115-32444-3

定价：99.00 元

欢迎加入 图灵社区

最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

最直接的读者交流平台

在图灵社区，你可以十分方便地写文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、审读、评选等多种活动，赢取积分和银子，积累个人声望。

ituring.com.cn