
Big nerd ranch
iOS Programming 4th Edition
iOS编程（第4版）
THE BIG NERD RANCH GUIDE
荣获Jolt生产力大奖
[美] Christian Keur Aaron Hillegass Joe Conway 著
丁道骏 译 张召 吴春燕 审校
華中科技大學出版社
http://www.hustp.com

版权信息

书名:《ios编程(第4版)》

原书名:iOS Progarmming, 4th Edition

ISBN:978-7-5609-9790-2

作者:[美] Christian Keur / [美] Aaron Hillegass / [美] Joe Conway

译者:丁道骏 / 张召 / 吴春燕

出版社:华中科技大学出版社

作者简介

Christian Keur是Big Nerd Ranch的高级讲师和软件工程师，负责编写Big Nerd Ranch的“iOS新手培训课程”教材。该教材广受好评，是本书的原型。Christian毕业于美国佐治亚理工学院计算机科学系，目前居住在亚特兰大。

Aaron Hillegass是Big Nerd Ranch的创始人之一，曾就职于NeXT公司和Apple公司，他拥有近20年的Objective-C、Cocoa、iOS开发与教学经验。Aaron与他人合著了《Mac OS X编程》和《Objectibe-C编程》。

Joe Conway曾参与编写了“iOS新手培训课程”教材。他最近创办了stable/kernel 公司，开发高质量的移动应用。

第1章 第一个简单的iOS应用

本章介绍如何编写一个简单的iOS应用——Quiz，功能为：在视图中显示一个问题，用户点击视图下方的按钮，可以显示相应的答案，用户点击上方的按钮，则会显示一个新问题（见图1-1）。


Carrier
1:45 PM
From what is cognac made?
Show Question
???
Show Answer

图1-1 第一个应用：Quiz

编写iOS应用时，读者必须先处理以下两个基本问题。

·如何创建并设置对象（例如，在某处放置一个按钮并将其标题设置为“显示问题”）？

·如何处理用户交互（例如，在用户按下某个按钮时执行某段代码）？

本书会用大量的篇幅回答这两个问题。

在阅读第1章时，请读者尽量走完整个流程，但不用试图搞懂每一个细节。模仿是一种有效的学习方式。可以通过模仿学会说话，也可以通过模仿学习iOS编程。等读者熟悉开发环境后，可以再尝试自行开发应用。现在，还请读者跟着本章照做，后续章节会详细介绍细节。

1.1 创建Xcode项目

运行Xcode，在File菜单中选择New→Project...。

Xcode会显示新的工作空间(workspace)窗口，同时工具栏(toolbar)处会弹出下拉窗口(sheet)。选择位于下拉窗口左侧iOS栏下的Application(见图1-2)，右侧有若干应用模板可供选择。请读者选择Empty Application(空应用)。

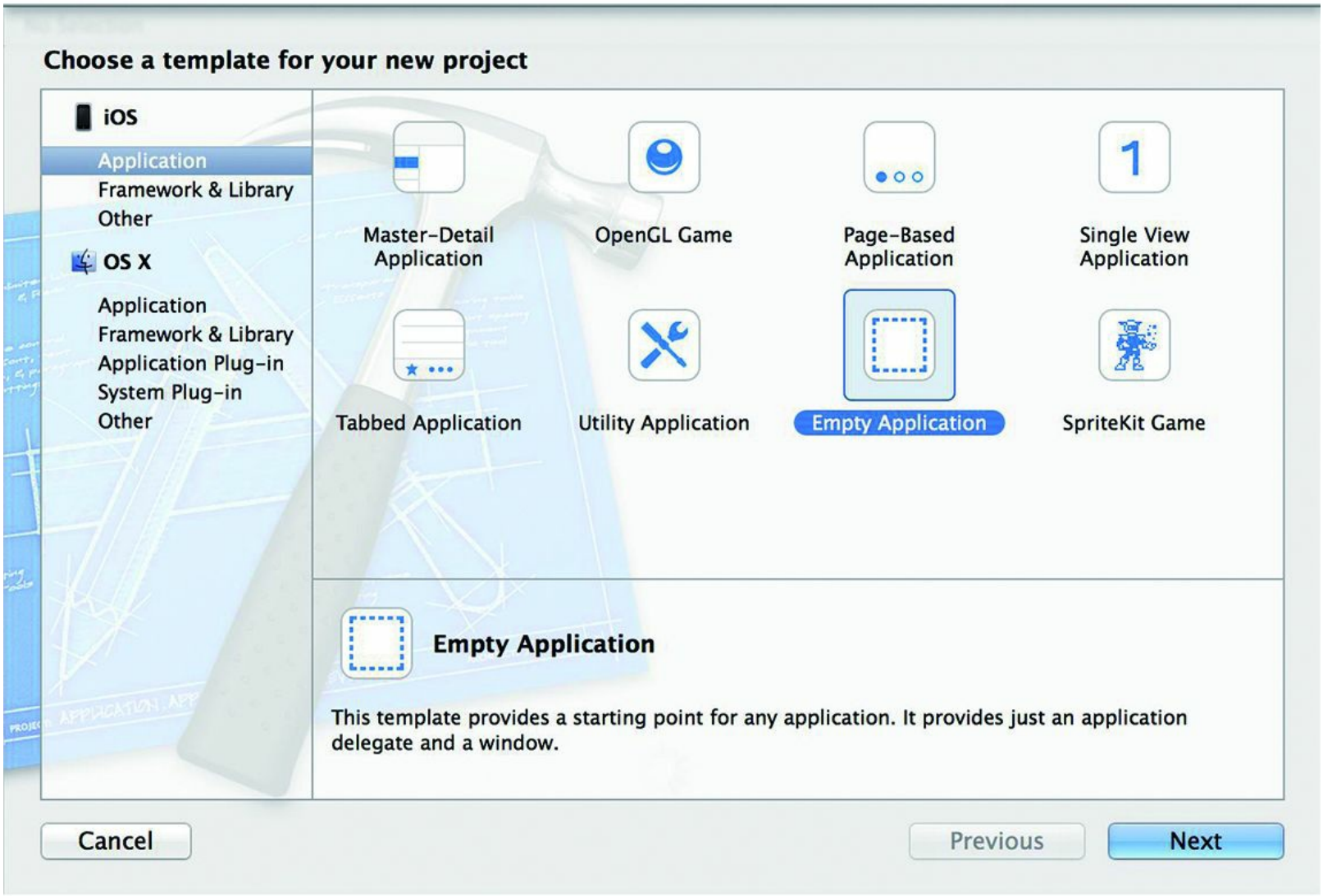


图1-2 创建新项目

空应用模板几乎没有多余的代码，而其他模板会生成很多通用代码。这些代码虽然能帮助开发应用，但是对于初学者，弊大于利。

本书中的项目都是使用Xcode 5.0.2创建的。Apple公司未来发布新版Xcode时，这些模板的名称可能有改动。读者在选择模板时，如果没有找到空应用模板，则可以选择一种看上去最简单的模板，例如Single View Application(单视图应用)。还可以访问本书原作者提供的论坛：http://forums.bignerdranch.com，以获取帮助。

单击Next按钮，在新出现的界面中，将Quiz填入Product Name文本框(见图1-3)。Organization Name和Company Identifier文本框也是必填的，读者可以分别填入Big Nerd Ranch和

com.bignerdranch，也可以填入自己的公司名称和公司的反向域名，如com.yourcompanynamehere。

将BNR填入Class Prefix文本框，在标题为Devices的弹出式菜单中选择iPhone。确保标题为Use Core Data的选择框未被选中。

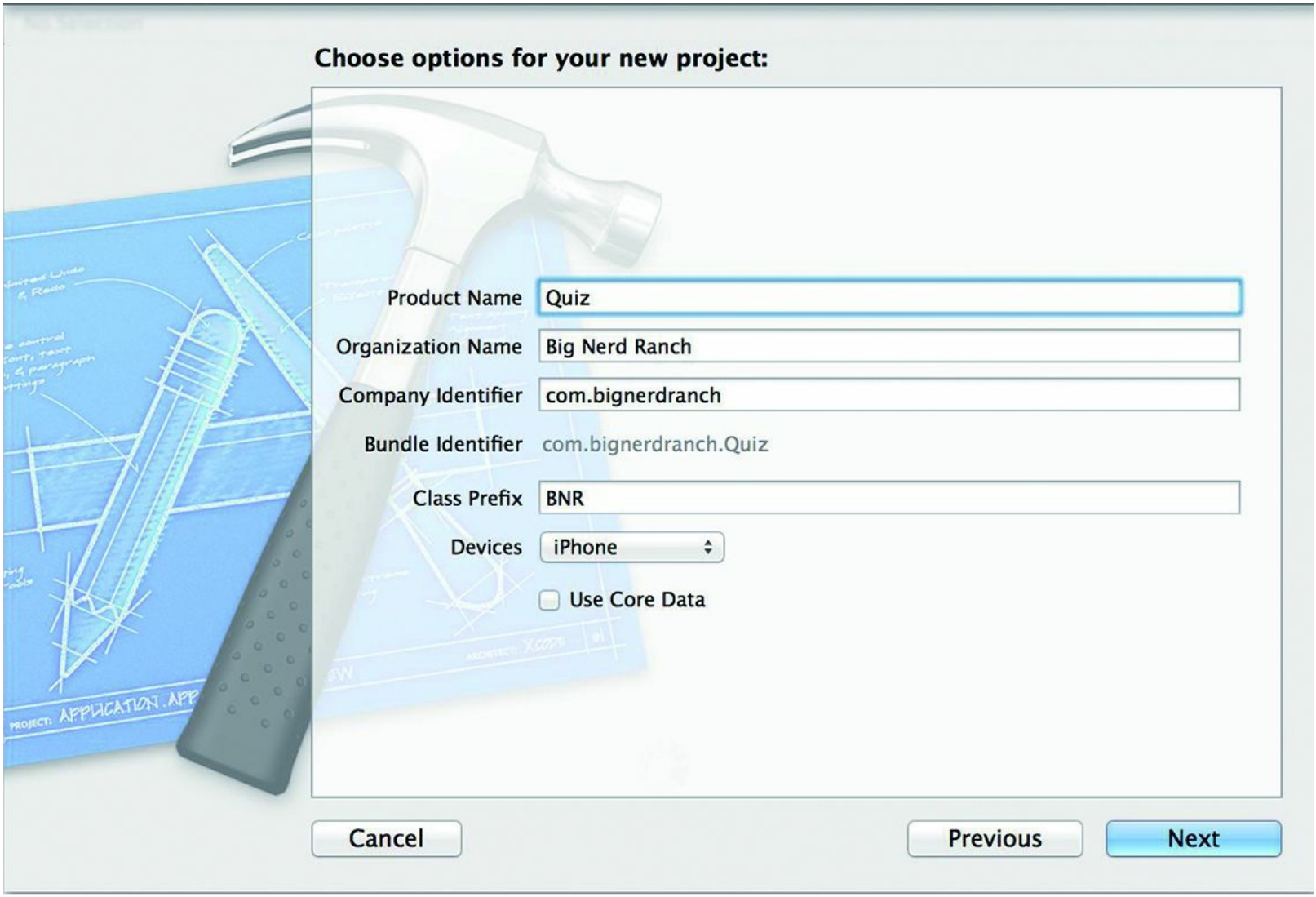


图1-3 设置新项目

虽然之前将Quiz项目的设备类型设置为了iPhone，但是生成的应用也能在iPad上运行。在iPad上，Quiz会在iPhone屏幕大小的窗口中运行，但不能充分利用iPad的大屏幕。对于一个用于学习的示例应用，这不是大问题。本书前半部分的应用都会使用基于iPhone设备的模板，并将重心放在学习iOS SDK的基础知识上。无论是哪种iOS设备，这些内容都是相同的。后面会介绍一些iPad独有的特性，以及如何编写在iPhone和iPad这两种设备上都能全屏运行的原生应用。

单击Next按钮后，Xcode会显示最后一个界面，提示读者保存项目。请准备好保存本书所有代码的目录，然后将Quiz项目保存在该目录下。本书不会介绍选择框Create local git repository for this project的作用，勾选或取消都可以。单击Create按钮，Quiz项目就创建好了。

项目创建完毕后，Xcode会显示工作空间窗口（见图1-4）。

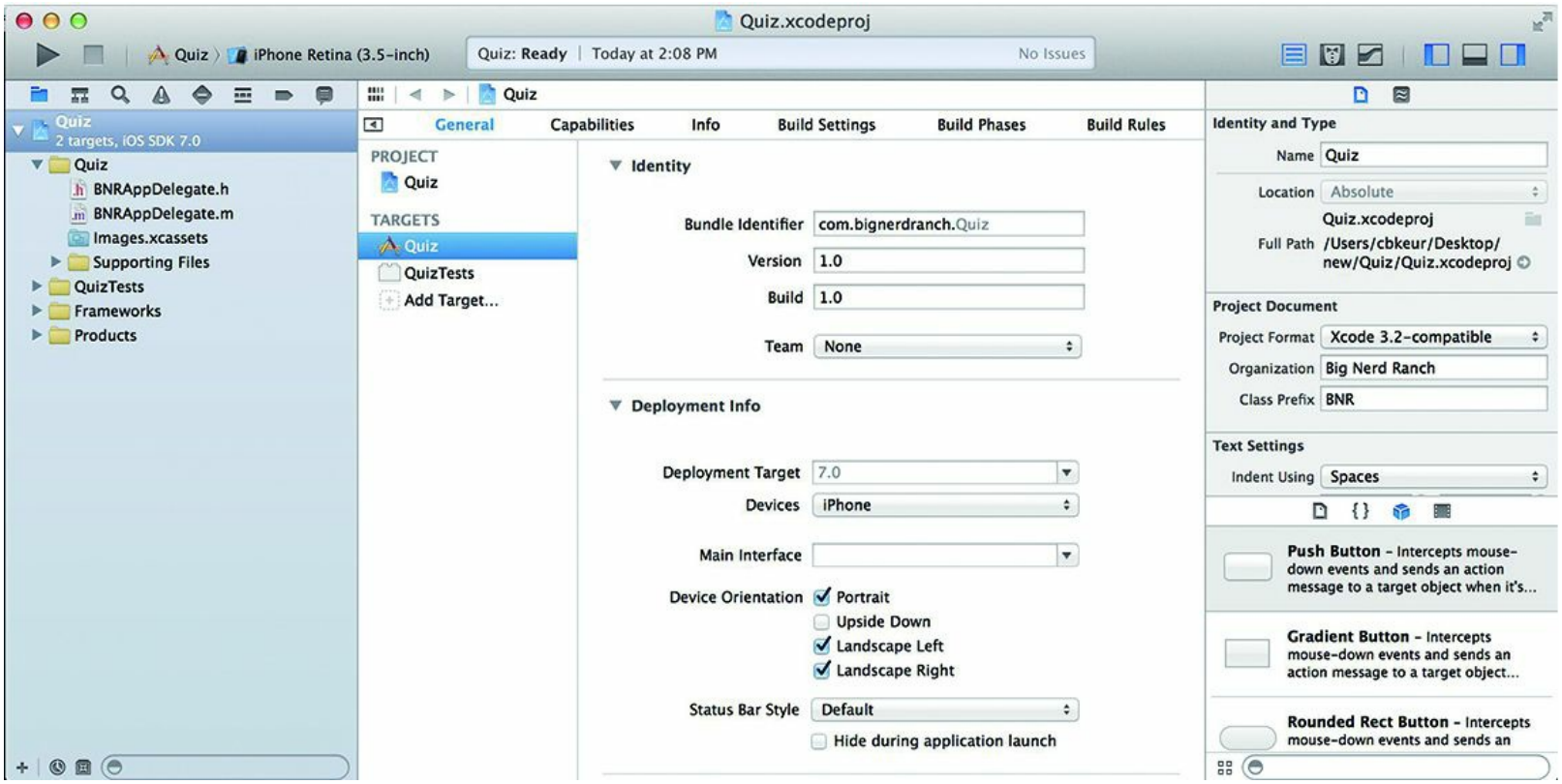


图1-4 Xcode工作空间窗口

位于工作空间窗口左侧的是导航面板区域(navigator area)，负责显示各种不同的导航面板。这些导航面板能分别显示项目的某些特定部分。单击导航面板选择条(位于导航面板区域上方)中的某个图标，可以选择相应的导航面板。

在Quiz项目工作空间中，当前选中的导航面板应该是项目导航面板(project navigator)，项目导航面板的作用是显示项目中的文件(见图1-5)。读者可以尝试选中任意一个文件，文件会在导航面板区域右边的编辑区域(editor area)中打开。

项目导航面板中的文件可以按目录分组，以帮助整理项目。Xcode模板已经为Quiz项目创建了若干组。读者可以随意修改组名或增加新的组。项目导航面板中的组只用来整理文件，与文件系统无关。

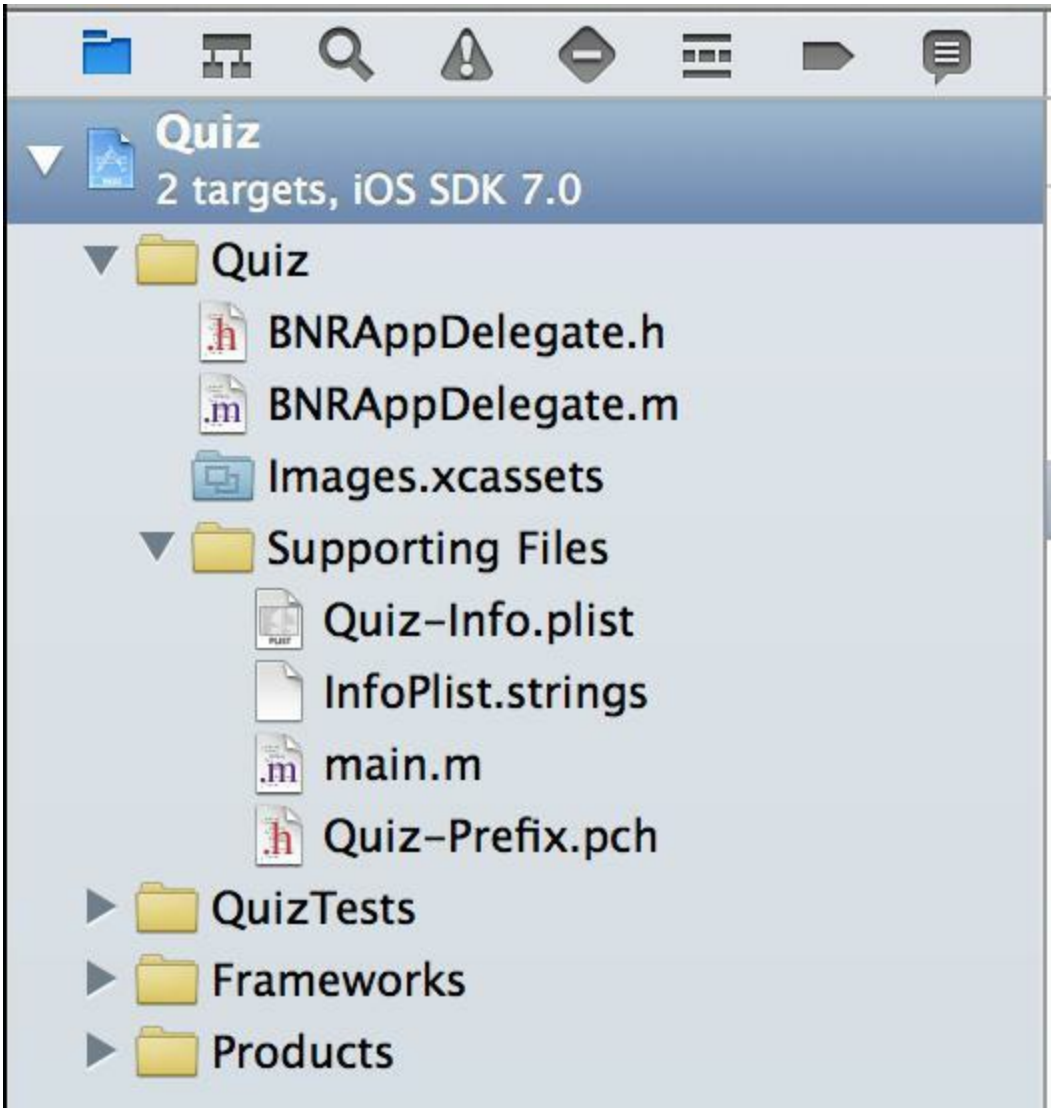


图1-5 项目导航面板列出的Quiz项目中的文件

在项目导航面板中，找到名为BNRAppDelegate.h和BNRAppDelegate.m的两个文件。它们被称为BNRAppDelegate的类(class)文件，是空应用模板自动创建的。

一个类(class)表示一种对象(object)。iOS开发是面向对象的，每个iOS应用都可以看成是由一系列协同工作的对象构成的。Quiz应用启动时，系统将创建一个BNRAppDelegate对象。BNRAppDelegate对象有时也称为BNRAppDelegate类的一个实例(instance)。

读者将在第2章中学习更多关于类和对象的知识。现在，还请读者关注iOS设计和开发的基本理论，继续完成本章的Quiz应用。

1.2 模型-视图-控制器

模型-视图-控制器(Model-View-Controller)，简称MVC，是iOS开发中频繁使用的一种设计模式。其含义是，应用创建的任何一个对象，其类型必定是模型对象、视图对象或控制器对象三种类型中的一种。

•视图对象是用户可以看见的对象，例如按钮、文本框、滑动条。视图对象用来构建用户界面，在Quiz应用中，显示问题和答案的标签以及标签下方的按钮都是视图对象。

•模型对象负责存储数据，与用户界面无关。Quiz应用中的模型对象是两个包含字符串对象的数组：questions数组和answers数组。

•通常情况下，模型对象表示真实世界中与用户相关的事物。例如，读者要为一家保险公司开发应用，那么很可能会设计一个InsurancePolicy(保险协议)类的模型对象。

•控制器对象扮演“管家”的角色，它用于控制视图对象为用户呈现的内容，以及负责确保视图对象和模型对象的数据保持一致。

一般来说，控制器用来回答：然后会发生什么？例如，用户从列表中选择了一项之后，控制器负责呈现接下来应该看到的内容。

图1-6显示的是应用响应用户操作的流程，例如用户点击了应用界面上的一个按钮。

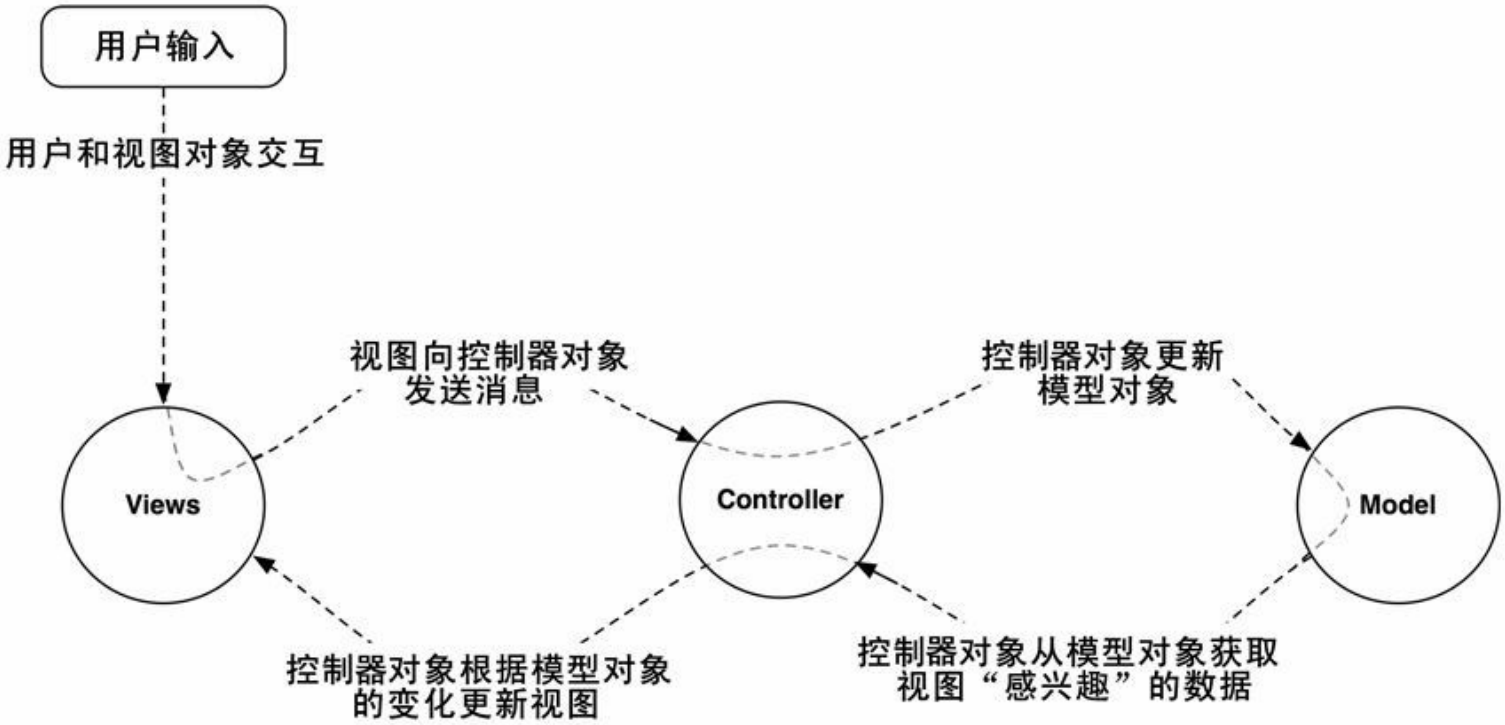


图1-6 MVC设计模式

请读者注意，模型对象和视图对象之间没有直接产生联系，而是由控制器对象负责彼此间的消息发送和数据传递。

1.3 设计Quiz

读者将使用MVC设计模式开发Quiz应用。以下列出了开发中需要使用的对象：

- 四个视图对象：UILabel和UIButton的对象各两个。
- 两个控制器对象：BNRAppDelegate和BNRQuizViewController的对象各一个。
- 两个模型对象：NSArray的对象两个。

图1-7显示的是Quiz应用的对象图，图中勾勒出了上述对象和相互关系。

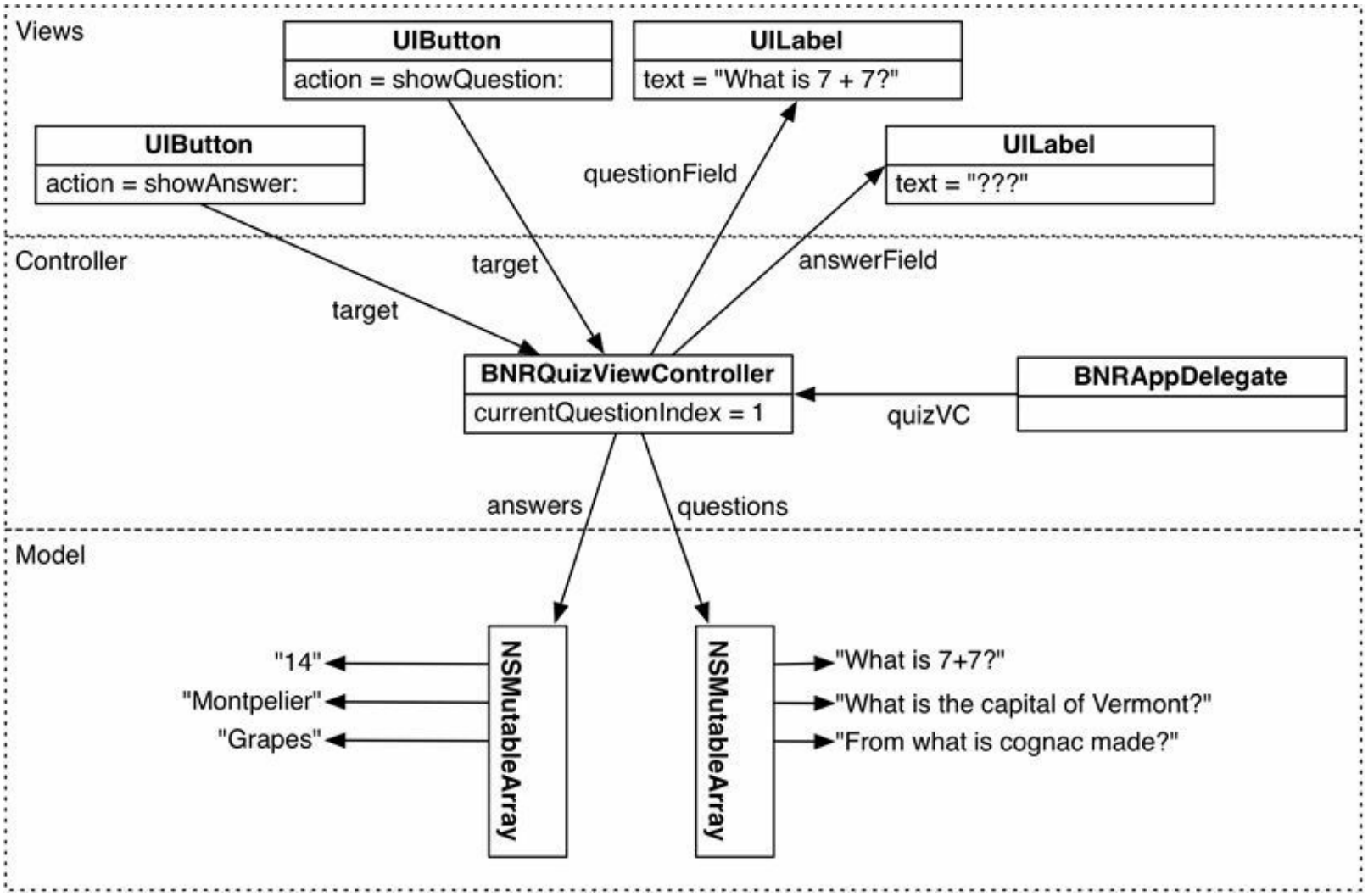


图1-7 Quiz应用的对象图

图1-7中展示了完成后的Quiz应用是如何工作的。例如，当用户按下Show Question按钮时，会触发BNRQuizViewController对象的一个方法（method）。方法与非面向对象语言中的函数（function）类似，都是一系列需要执行的命令。这个方法会从questions数组里取出一道新题目，然后通过位于视图上方的标签将题目显示出来。

读者现在可能还无法完全看懂这幅对象图，没关系，到本章结尾再回来看这幅图时，就能深刻理解Quiz应用的工作原理了。

现在请读者跟着本章一步步开发Quiz应用。第一步是创建控制器对象，应用的核心控制器——BNRQuizViewController。

1.4 创建视图控制器

空应用模板已经创建好了BNRAppDelegate类文件，现在请读者创建BNRQuizViewController类文件。我们将在第2章和第6章介绍更多关于类和视图控制器的知识。现在，还请读者跟着本章照做。

在File菜单中选择New→File…Xcode会弹出选择文件模板的下拉窗口。选择位于下拉窗口左侧iOS栏下的Cocoa Touch，再选择右侧的Objective-C class（Objective-C 类）并单击Next按钮（见图1-8）。

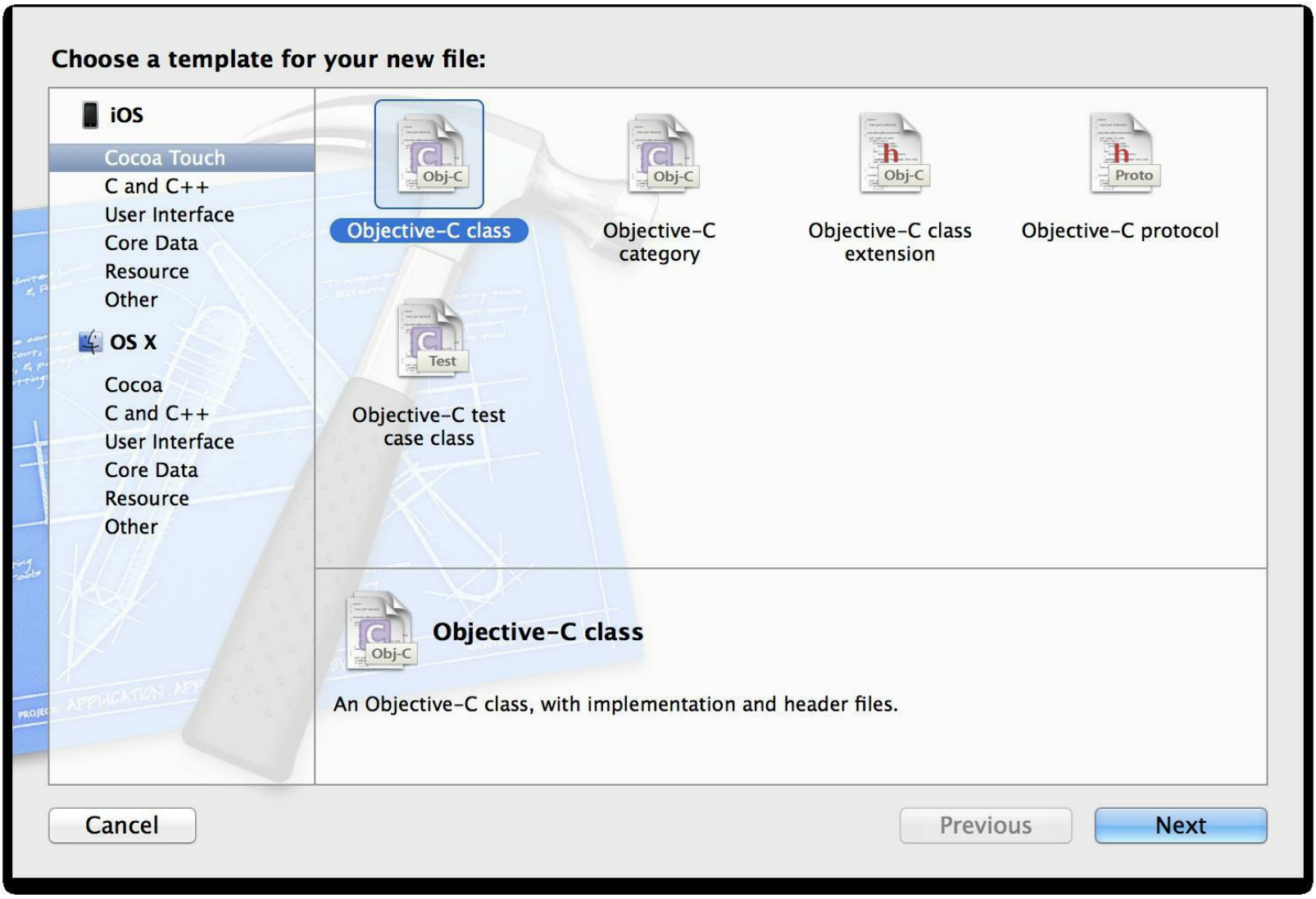


图1-8 创建Objective-C 类

在新出现的窗口中，将BNRQuizViewController填入Class文本框，然后点击Subclass of文本框的下拉按钮，在弹出式菜单中选择UIViewController。勾选标题为With XIB for user interface的选择框（见图1-9）。

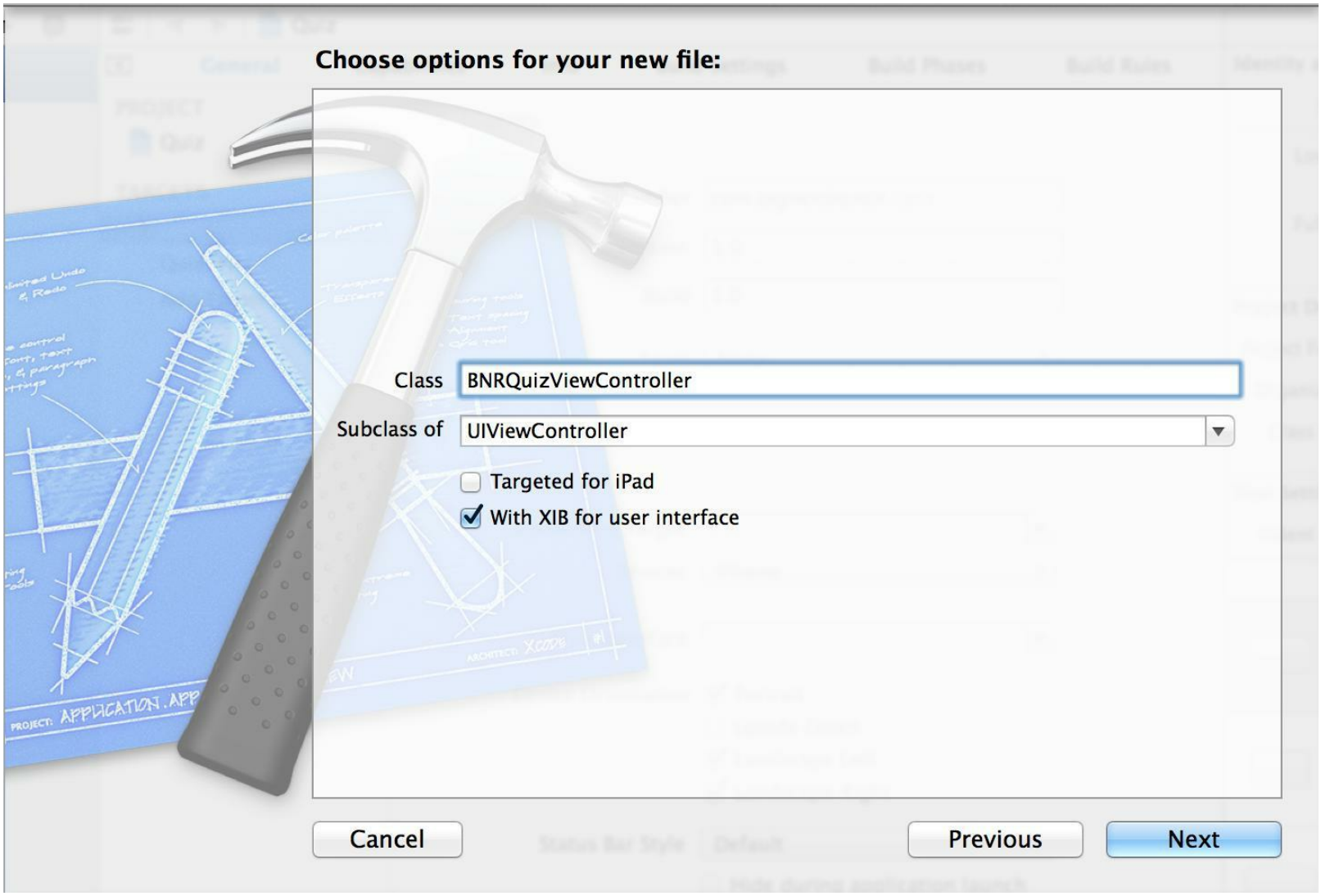


图1-9 创建视图控制器

单击Next按钮，Xcode会提示读者为BNRQuizViewController类文件选择保存位置。为项目创建一个新的类时，通常应该将类文件保存在对应的项目文件夹中，Xcode会默认选中当前项目文件夹，也可以点击标题为Group的弹出式菜单，菜单中显示了与项目导航面板中对应的组，读者可以选择想要保存类文件的组。因为组只用来整理文件，而Quiz项目又很简单，所以全部使用默认设置就可以了。

请读者确保选中了Targets中Quiz前的选择框（见图1-10）。选中后，Xcode在构建项目时会一起编译BNRQuizViewController类文件。最后单击Create按钮。

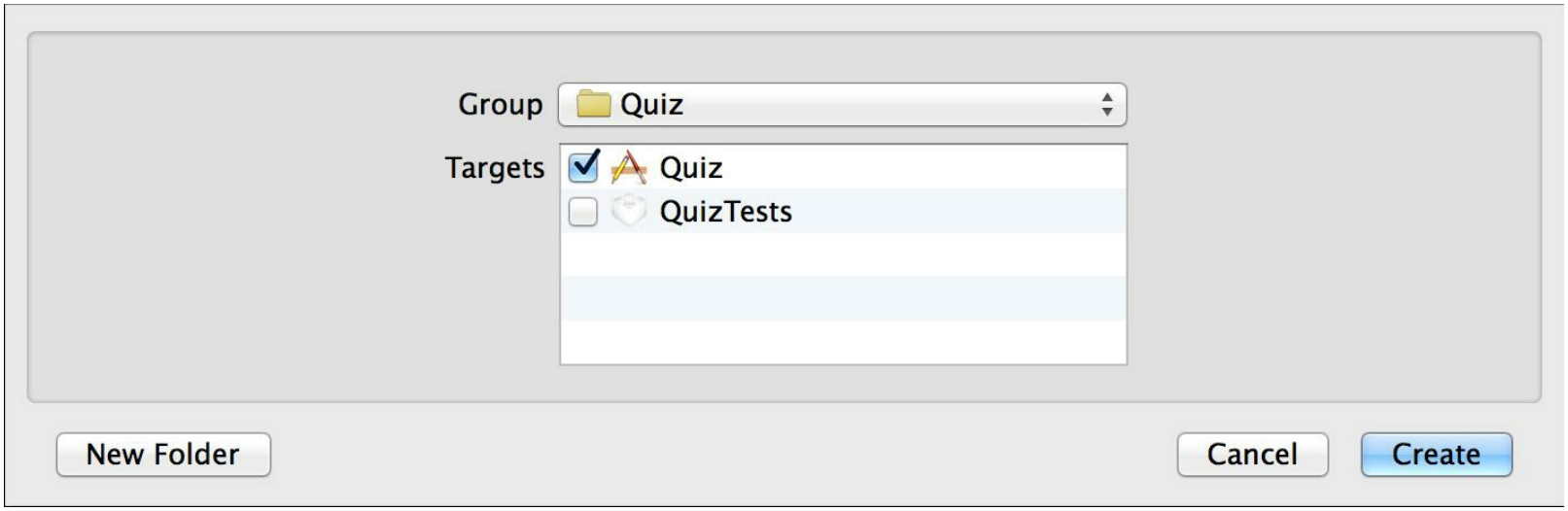


图1-10 选中Targets中的Quiz

1.5 创建界面

在项目导航面板中找到BNRQuizViewController类文件。读者在创建这个类时勾选了With XIB for user interface，因此Xcode自动生成了第三个文件：BNRQuizView- Controller.xib。请读者选择BNRQuizViewController.xib。

Xcode会使用Interface Builder（界面创建工具）打开XIB（发音“zib”）文件。Interface Builder是一种可视化编辑器，可以用拖动对象的方式创建图形用户界面。XIB的全称就是XML Interface Builder。

很多其他平台的GUI创建工具，需要先描述应用外观，然后单击某个按钮，生成大量代码。Interface Builder则不同，它是一种对象编辑器：使用时需要先创建并设置对象，例如按钮和标签，然后将对象保存在固化文件里。这种固化文件就是XIB文件。

Interface Builder将编辑区域分为两部分：左侧是dock，右侧是画布。

dock可以使用两种方式展示XIB文件中的对象：图标视图（icon view）和大纲视图（outline view）。图标视图能为屏幕腾出更多的空间，但是出于学习目的，使用大纲视图查看对象更加方便。

如果读者的dock处于图标视图状态，请点击icon / outline视图切换按钮将dock切换到大纲视图状态，按钮在画布左下角的位置（见图1-11）。

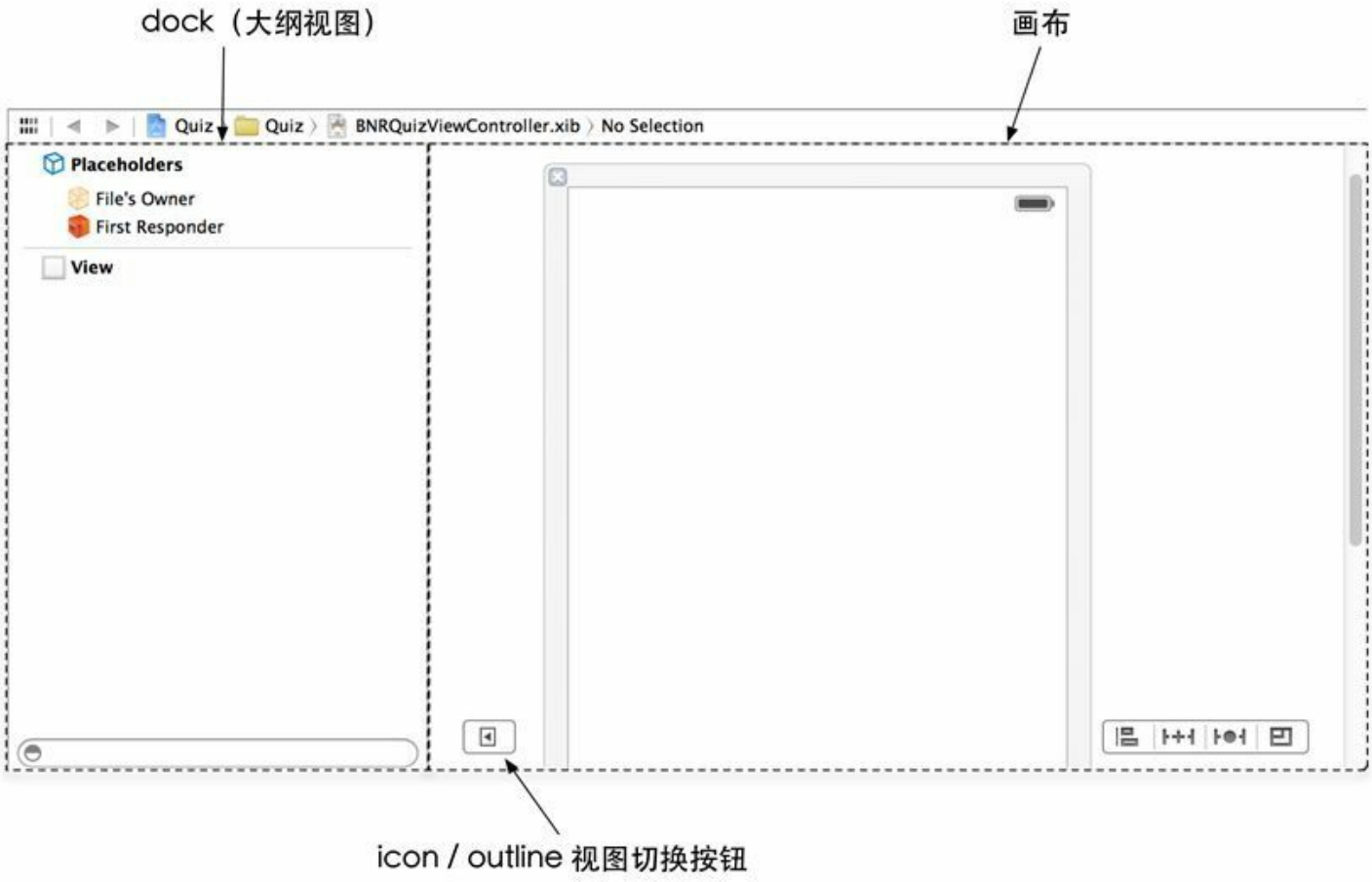


图1-11 使用Interface Builder编辑XIB文件

大纲视图中的BNRQuizViewController.xib包含三个对象：Placeholders模块中的两个占位符对象（placeholder）和一个View对象。请读者先忽略占位符对象，我们稍后再介绍它。

View对象是一个UIView类的对象，读者可以在画布上看到它。它是用户界面的基石，可以容纳其他视图对象。而且，读者在画布上看到的和应用正式运行时的外观完全相同。

单击大纲视图中的View对象，它会显示在画布上。拖曳移动视图，移动视图不会修改对象自身，只会重新组织画布。单击视图左上角的x按钮可以关闭视图。同样，这样做不会删除视图，只是将其从画布中移除。再次选中大纲视图中的View对象，可将其加回画布。

现在Quiz应用的用户界面只有一个空白的视图对象，还需要添加两个标签和两个按钮（见图1-12）。

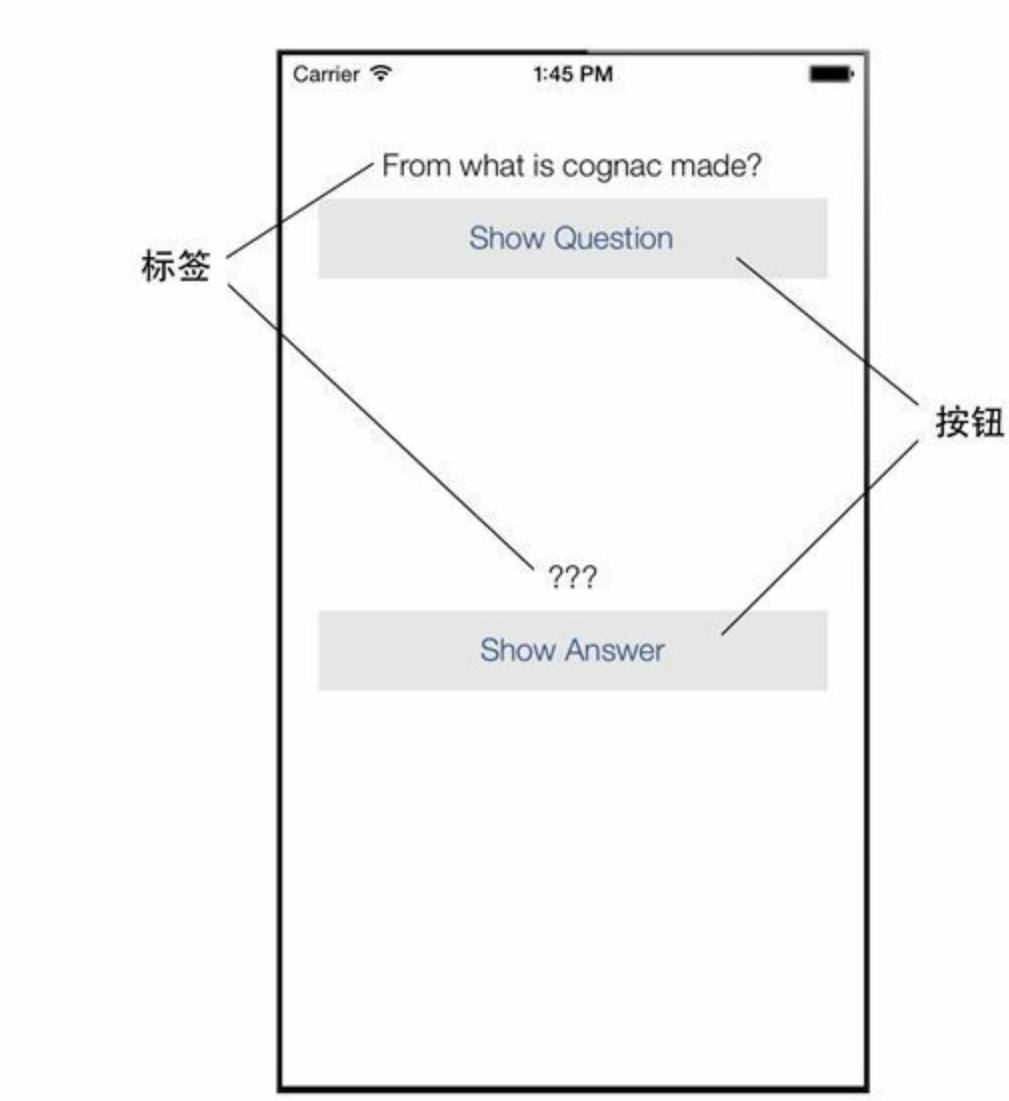


图1-12 需要的标签和按钮

创建视图对象

要加入这些视图对象，需要打开工具区域（utilities area）的对象库（object library）。

工具区域位于编辑器区域的右侧，分上下两部分：检视面板（inspector）和库面板（library）。上方的检视面板负责显示编辑器区域当前选中的文件或对象的各种设置。下方的库面板则会列出可以加入文件或项目的物件（例如对象或代码）。拖曳这两个区域间的分隔条，可以改变其相对的尺寸。

这两个面板的顶部都有一选择条，可以用来选择各种不同类型的面板和库（见图1-13）。在库面板选择条中，单击图标，可以打开对象库面板。

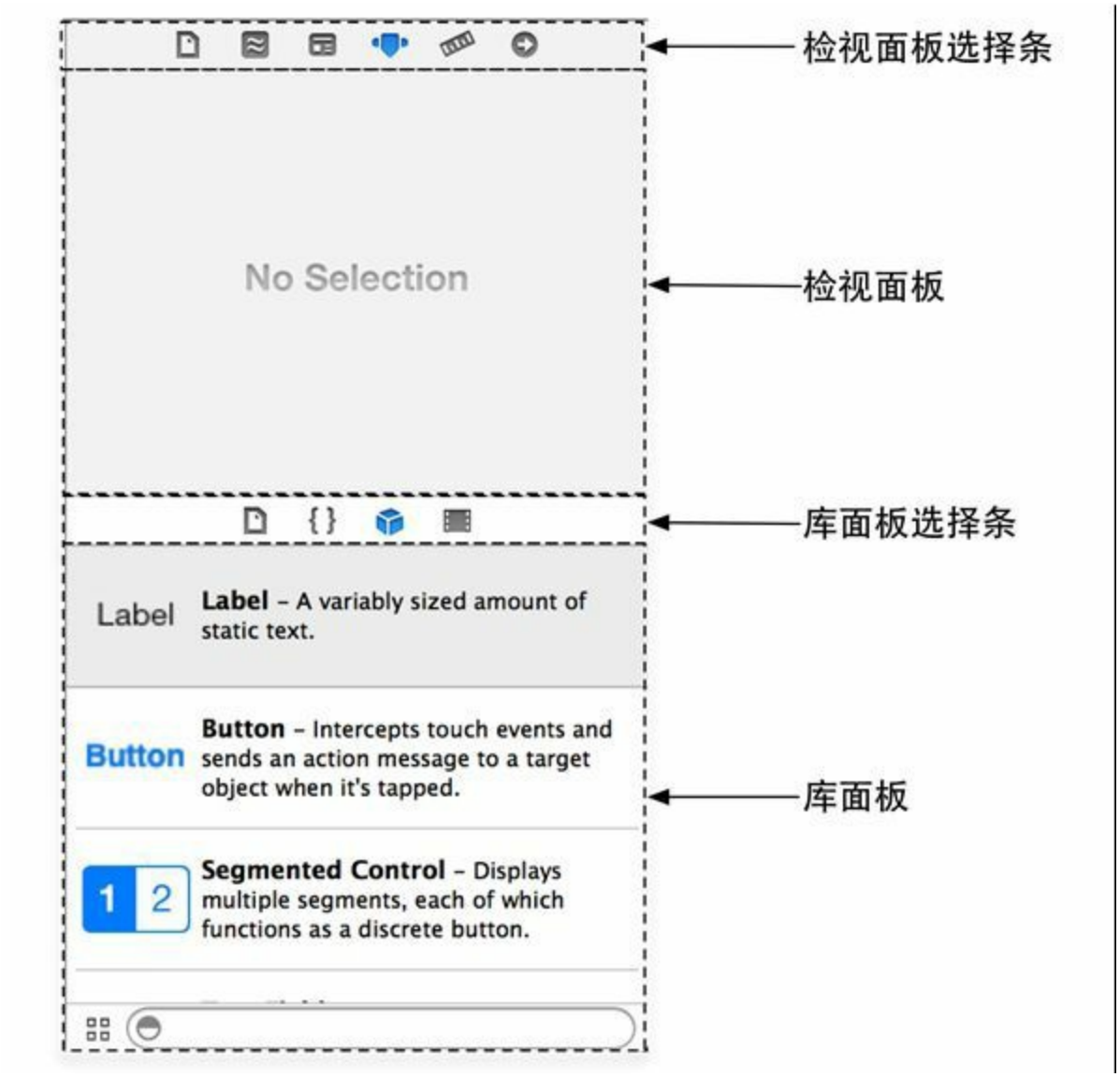


图1-13 Xcode工具区域

对象库中的对象主要用于构建用户界面，它们可以被拖曳到XIB文件中。

请读者在对象库中找到Label对象（应该可以在列表顶部找到Label对象，如果没有，可以往下滚动列表，或者使用面板底部的查询框），然后选中该对象并拖曳至（画布上的）视图对象上，再将这个Label对象放在视图正中间靠近顶部的位置。拖曳第二个Label对象，放在视图正中间靠近底部的位置。

接下来请找到Button对象，拖曳两个Button对象至视图对象，并分别放在两个Label对象的下方。

现在读者已经为BNRQuizViewController.xib添加了四个新的视图对象，在编辑区域的大纲视图中可以看到它们。

设置视图对象

视图对象已经全部创建好了，接下来应该设置对象的属性。部分属性，例如大小、位置和文本可以在画布中直接编辑。其他属性则必须通过属性检视面板（attributes inspector）来编辑，我们很快就会介绍它。

请读者修改以上四个对象的大小，使其宽度能够横跨窗口的大部分区域（在画布或大纲视图中选中对象，然后拖曳它的角或边，就可以修改对象的大小，如图1-14所示）。

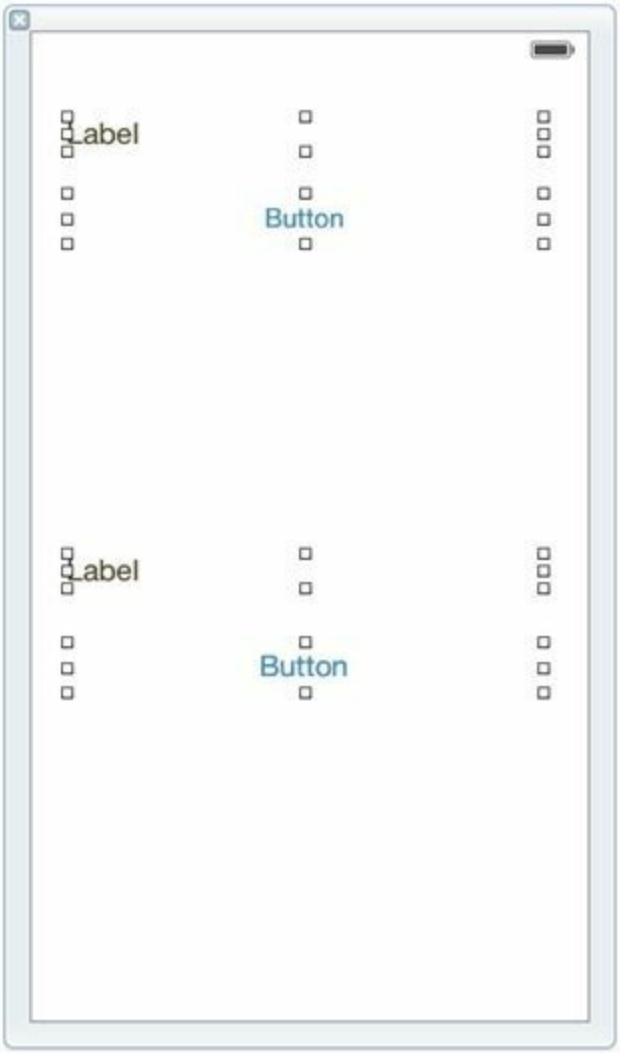


图1-14 修改标签和按钮的大小

双击画布中的某个Button对象，就可编辑该对象的标题。将上方Button对象的标题改为Show Question（显示问题），下方的改为Show Answer（显示答案）。使用同样的方法可编辑Label对象的文字。删除上方Label对象的文字（之后用于显示问题），下方的设置为？？？。这时的界面如图1-15所示。

图1-15 设置标签和按钮的文本

如果标签中的文本处于居中对齐状态，则界面会更加美观。设置标签的文本对齐方式必须通过属性检视面板来编辑。

选中位于视图下方的Label对象，单击检视面板选择条中的图标，打开属性面板。

在属性面板中找到标题为Alignment的分段控件（segmented control）。选择中间的那个选项（居中对齐），如图1-16所示。

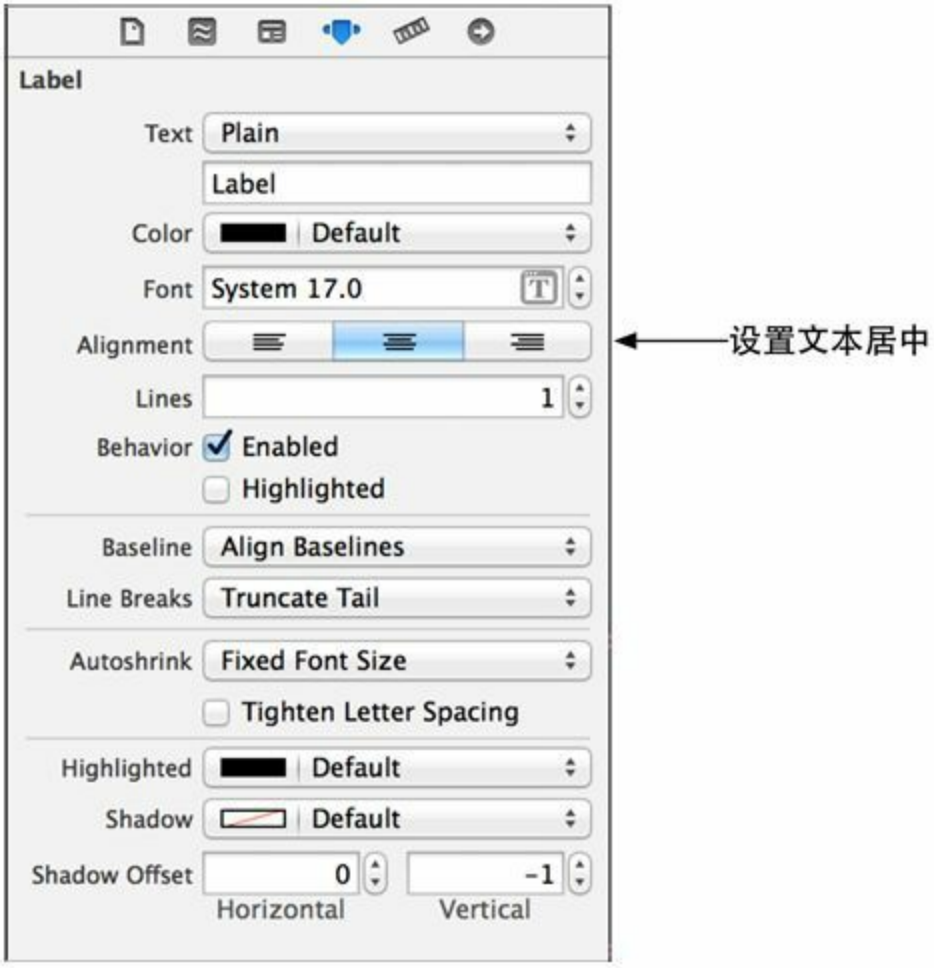


图1-16 设置label的文本对齐方式为居中对齐

修改后请读者观察画布中的变化，视图下方Label的文字？？？会改为居中对齐。下面将视图上方的Label也设置为居中对齐（该对象目前还没有文字，但是运行时会有）。

为了让用户知道按钮的点击区域，可以改变按钮的背景颜色。首先请在画布中选择Show Question按钮。

在属性检视面板中向下滚动，会看到View标题下方的属性。在Background标签右侧，点击颜色面板（白色方块，上面有一条红色的斜线）可以打开颜色拾取器。请读者为按钮选择一种满意的颜色。

接下来选择Show Answer按钮，但是这次点击颜色面板右侧有上下箭头的弹出式菜单。菜单中会显示系统默认颜色和最近使用的颜色。请在菜单中选择与Show Question按钮相同的颜色。

NIB文件

此时读者可能会好奇，应用运行时是如何使用这些对象的呢？

构建项目时，所有XIB文件都会被编译为NIB文件（NIB文件体积更小，更容易解析），然后Xcode会将NIB文件拷贝至应用的程序包（bundle）中。程序包其实就是目录，其中包含应用的可执行文件和其会用到的所有资源文件。

应用在运行时刻会从程序包中按需载入NIB文件并激活文件中的对象。Quiz应用只有一个NIB文件，并且会在应用启动时载入，复杂的应用会有很多NIB文件。读者将在第6章中学习更多关于应用如何加载NIB文件的知识。

现在，应用的用户界面看起来很不错，但是还没有任何功能。接下来请读者将视图对象和控制器对象——BNRQuizViewController关联起来，这样应用就可以响应用户操作了。

1.6 创建关联

通过关联(connection)，一个对象可以知道另一个对象在内存中的位置，从而使这两个对象可以协同工作。在Interface Builder中可以创建两种关联：插座变量(outlets)和动作(actions)。插座变量是一种指向对象的指针(将在第2章中介绍指针)；动作是一种方法，这种方法在视图对象和用户发生交互时会被调用，例如点击按钮、拖曳滑动条、滚动选择器等。

现在请读者离开可视化的Interface Builder，开始尝试编写一些代码。首先，创建插座变量指向两个UILabel对象。

声明插座变量

在项目导航面板中选择BNRQuizViewController.m文件，编辑区域会自动从Interface Builder切换到Xcode代码编辑器。

在BNRQuizViewController.m文件中，删除@implementation和@end之间所有应用模板自动生成的代码，这时文件看起来应该是这样：

```
#import "BNRQuizViewController.h"

@interface BNRQuizViewController ()

@end

@implementation BNRQuizViewController

@end
```

现在添加以下代码。读者可能看不懂这些代码，不用担心，请先输入代码。

```
#import "BNRQuizViewController.h"

@interface BNRQuizViewController ()

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

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

@end

@implementation BNRQuizViewController

@end
```

请读者注意粗体的代码。本书会以加粗的字体显示需要读者输入的代码。其他非粗体的代

码是为了提示读者应该插入新代码的位置。

新添加的代码中声明了两个属性(properties)。读者将在第3章中学习属性。现在请关注第一行代码的后半部分。

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

粗体代码为BNRQuizViewController对象声明了一个插座变量，名叫questionLabel，它可以指向一个UILabel对象。IBOutlet关键字告诉Xcode之后会使用Interface Builder关联该插座变量。

设置插座变量

在项目导航面板中选择BNRQuizViewController.xib，Xcode会重新打开Interface Builder。

下面将插座变量questionLabel指向视图上方的UILabel对象。

在dock中找到Placeholders模块中的File's Owner对象。占位符对象(placeholder)在程序运行时会表示其他对象。对于File's Owner对象来说，它表示BNRQuizViewController对象。BNRQuizViewController对象负责管理BNRQuizViewController.xib文件中定义的对象。右击(或者按住Control-单击)File's Owner，打开关联面板(见图1-17)。按住questionLabel右侧的圆圈，然后拖曳至视图上方的标签，当标签处于高亮状态时松开鼠标左键。这样插座变量就设置好了。

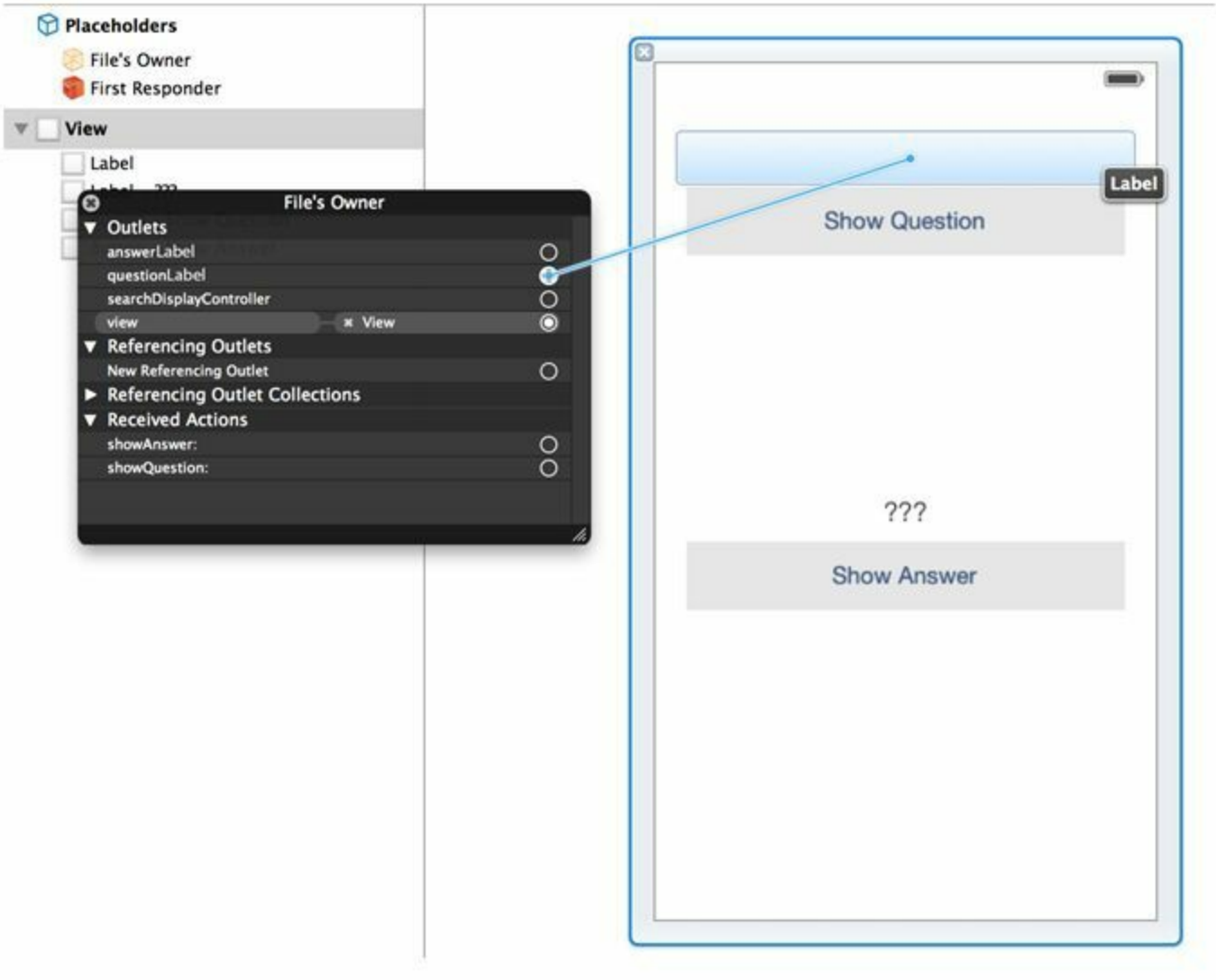


图1-17 设置questionLabel

（如果读者的关联面板没有显示questionLabel，请打开BNRQuizViewController.m文件仔细检查是否输错了代码。）

现在当应用载入NIB文件时，BNRQuizViewController对象的questionLabel插座变量会自动指向位于视图上方的UILabel对象。通过这个关联，BNRQuizViewController对象就能在应用运行时管理该标签显示的问题。

请读者使用同样的方式创建另一个关联。按住answerLabel右侧的圆圈，将其拖曳至下方的标签（见图1-18）。

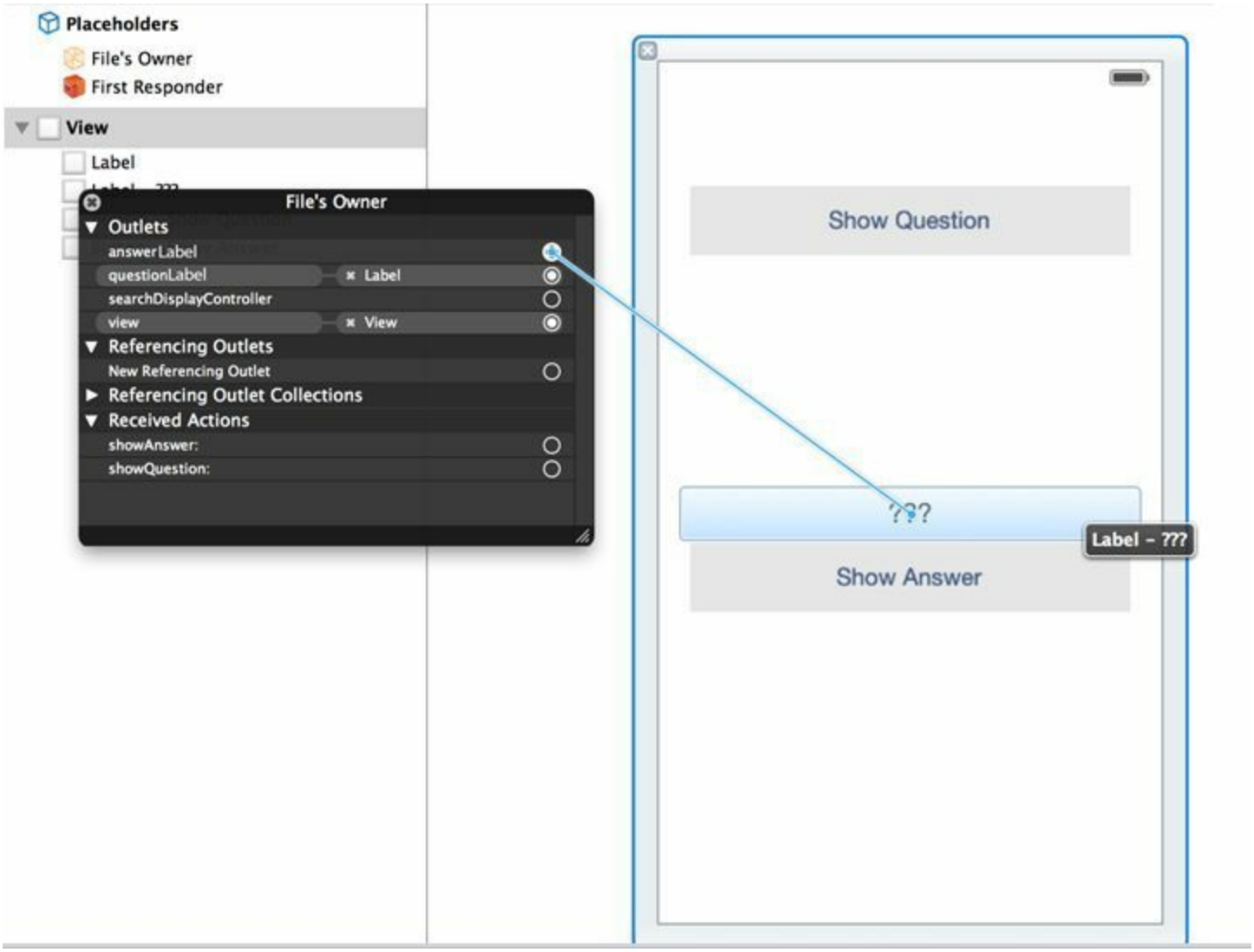


图1-18 设置answerLabel

注意，这里的起点是“拥有插座变量的对象”，终点是“插座变量需要指向的对象”。

所有的插座变量都已经设置好了，下一步是关联两个按钮，让它们可以响应用户点击。

点击UIButton对象时，该按钮可以向某个对象发送指定的消息(message)。这里接收消息的对象称为目标(target)。消息本身称为动作(action)，并且消息的名称就是点击按钮时应该触发的方法名称。

在Quiz应用中，两个按钮的目标都是BNRQuizViewController对象，但是每个按钮的动作不一样，首先请读者定义两个动作方法：showQuestion:和showAnswer:。

声明动作方法

请读者打开BNRQuizViewController.m文件，在@implementation和@end之间加入以下代码。

```
@implementation BNRQuizViewController
```

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

{

}

- (IBAction)showAnswer:(id)sender

{

}

@end
```

方法的具体实现代码将在关联动作之后添加。IBAction关键字告诉Xcode之后会使用Interface Builder关联该动作。

设置目标和动作

要设置某个对象的目标，可以按住Control，拖曳该对象至相应的目标，然后松开鼠标左键，目标就设置好了。这时Xcode会弹出新菜单，提示读者选择动作。

首先请读者设置Show Answer按钮的目标是BNRQuizViewController对象，动作是showQuestion:。

重新打开BNRQuizViewController.xib，在画布中选择Show Question按钮，按住Control并拖曳（或者右键拖曳）至File's Owner。当Xcode高亮显示File's Owner时，松开鼠标左键，选择弹出菜单中的showQuestion:，如图1-19所示。

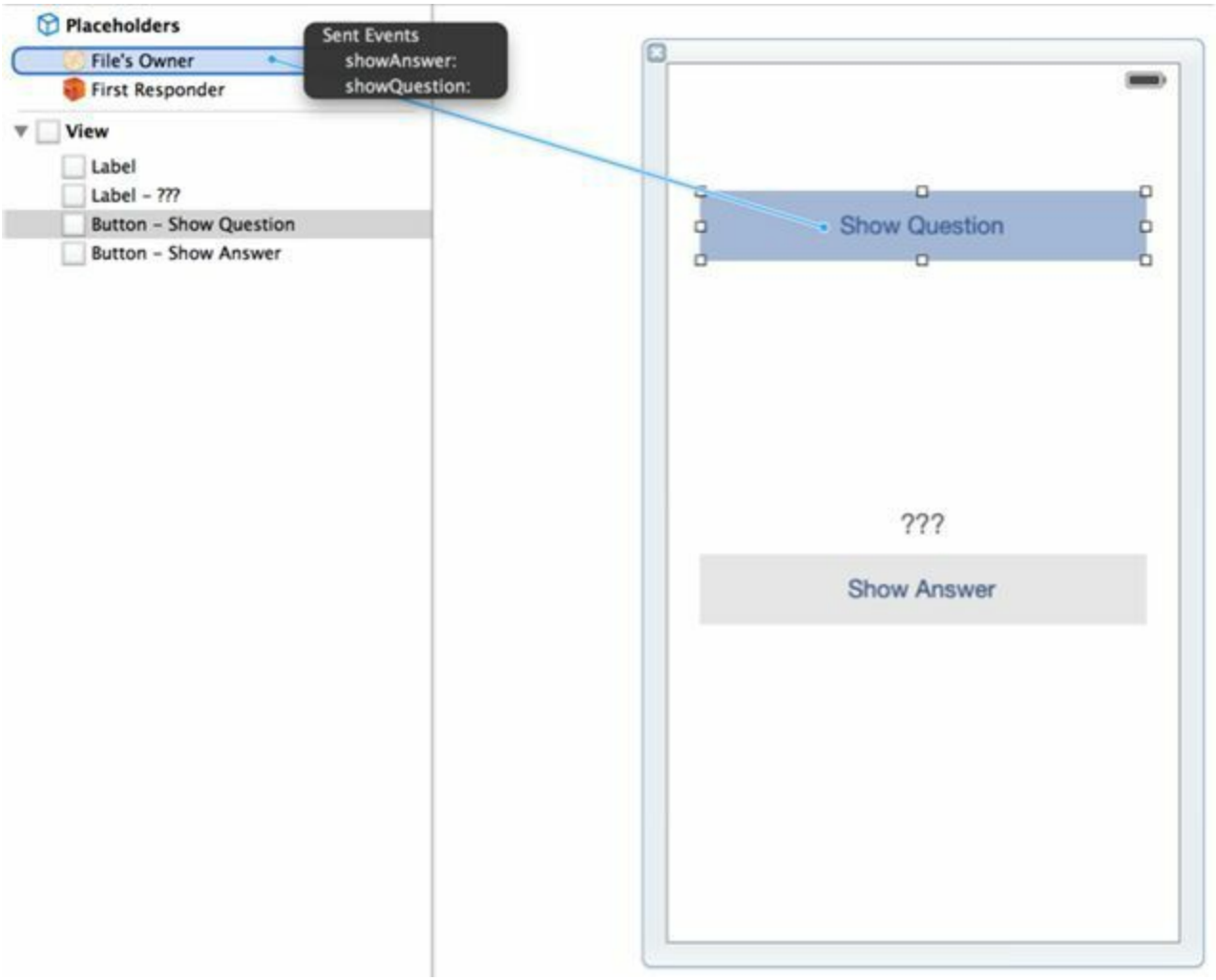


图1-19 设置Show Question按钮的目标/动作

下面设置Show Answer按钮的目标和动作。选中Show Answer按钮，按住Control并拖曳至File's Owner，选择弹出菜单中的showAnswer:。

Quiz中的关联

现在BNRQuizViewController对象和视图对象之间一共有五个关联。BNRQuizViewController对象的两个指针——answerLabel和questionLabel指向相应的UILabel对象；视图上的两个UIButton对象，其目标都是BNRQuizViewController对象；项目模板创建的一个额外的关联，即名为view的插座变量，指向作为应用背景的UIView对象。

读者可以通过关联检视面板（connections inspector）查看这些关联。选中大纲视图中的File's Owner，在检视面板中选择图标，打开关联检视面板（见图1-20）。

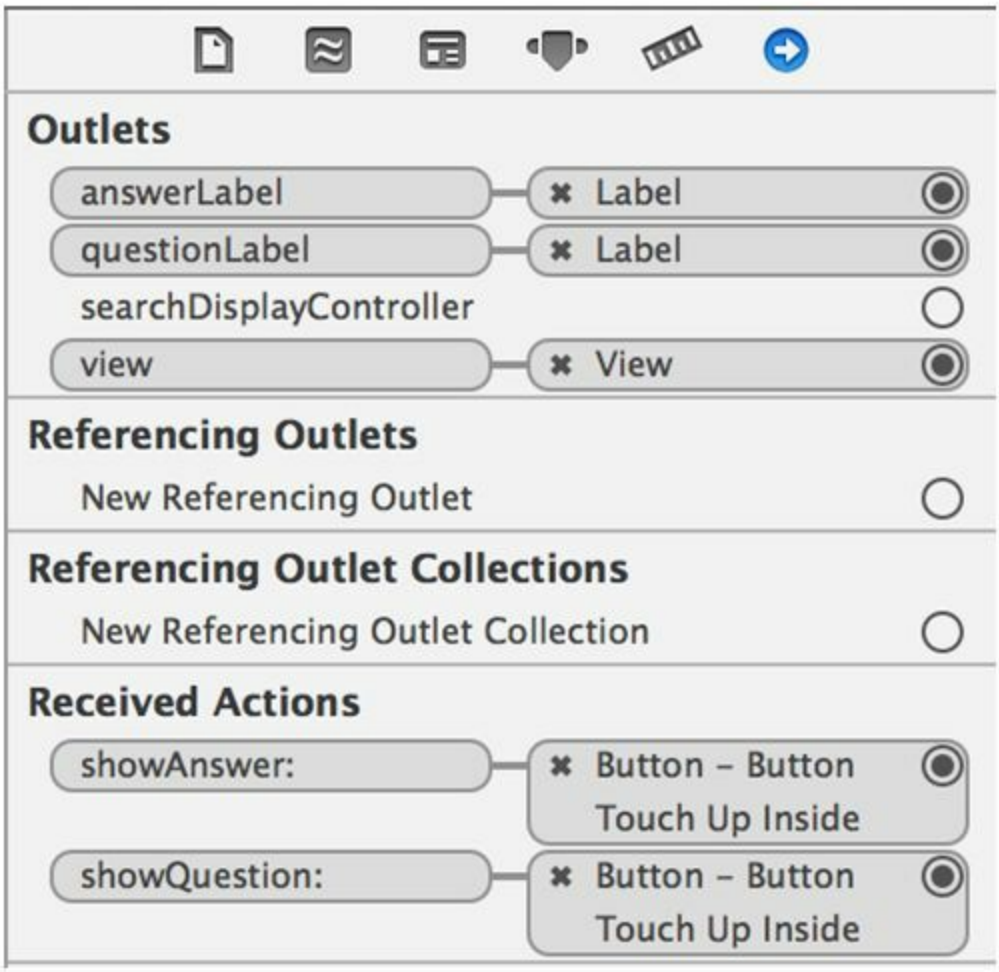


图1-20 通过检视面板查看关联

以上完成了Quiz应用的XIB文件。读者创建并设置了应用所需的视图对象，并为视图对象和控制器对象创建了所有必需的关联。下面开始创建模型对象。

1.7 创建模型对象

视图对象构成了用户界面，开发者通常在Interface Builder中创建、设置和关联视图对象。而模型对象则是通过编写代码创建的。

在项目导航面板中选择BNRQuizViewController.m文件。添加以下代码，声明一个整型变量和两个数组对象：

```
@interface BNRQuizViewController ()

@property (nonatomic) int currentQuestionIndex;

@property (nonatomic, copy) NSArray *questions;

@property (nonatomic, copy) NSArray *answers;

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

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

@end

@implementation BNRQuizViewController

- (IBAction)showQuestion:(id)sender

{

}

- (IBAction)showAnswer:(id)sender

{

}

@end
```

两个数组用于存储一系列问题和答案，而整型变量用于跟踪用户正在回答的问题。

为了确保用户在看到应用界面时，数组已经存储了所需的问题和答案，必须在BNRQuizViewController对象创建完毕之后立即创建数组。

BNRQuizViewController对象创建完毕之后会收到消息：initWithNibName:bundle:。请读者在BNRQuizViewController.m文件中实现initWithNibName: bundle:方法。

```
...
```

```
@property (nonatomic, weak) IBOutlet UILabel *answerLabel;
@end
@implementation BNRQuizViewController
- (instancetype)initWithNibName:(NSString *)nibNameOrNil
bundle:(NSBundle *)nibBundleOrNil
{
// 调用父类实现的初始化方法
self = [super initWithNibName:nibNameOrNil
bundle:nibBundleOrNil];
if (self) {
// 创建两个数组对象，存储所需的问题和答案
// 同时，将questions和answers分别指向问题数组和答案数组
self.questions = @[@"From what is cognac made?",
@"What is 7+7?",
@"What is the capital of Vermont?"];
self.answers = @[@"Grapes",
@"14",
@"Montpelier"];
}
 // 返回新对象的地址
return self;
}
...
@end
```

（现在读者不用理解代码，第2章将介绍更多关于Objective-C语言的知识。）

使用代码补全功能

读者在阅读本书的过程中，需要输入大量的代码。Xcode会在读者输入代码时，自动填入部分代码。例如，输入initWithNibName:bundle:的前几个字符，Xcode会显示相应的提示并将其列为备选项。按回车键，就可以将这个备选项写入代码，也可以在Xcode显示的下拉列表中选择其他的备选项。

如果选中的备选项对应的是带实参的方法，那么Xcode会在实参的位置放置占位符。（请读者注意，这里的占位符和之前XIB文件中介绍的占位符对象完全不是一回事。）

占位符不是真正的代码，必须将其替换成实际的代码。因为占位符的名称通常会和实参的名称相同，所以初学者很容易产生混淆：代码看上去完全正确，但是编译时会出错。

图1-21显示的是读者在输入上述代码时，可能会碰到的两个占位符。

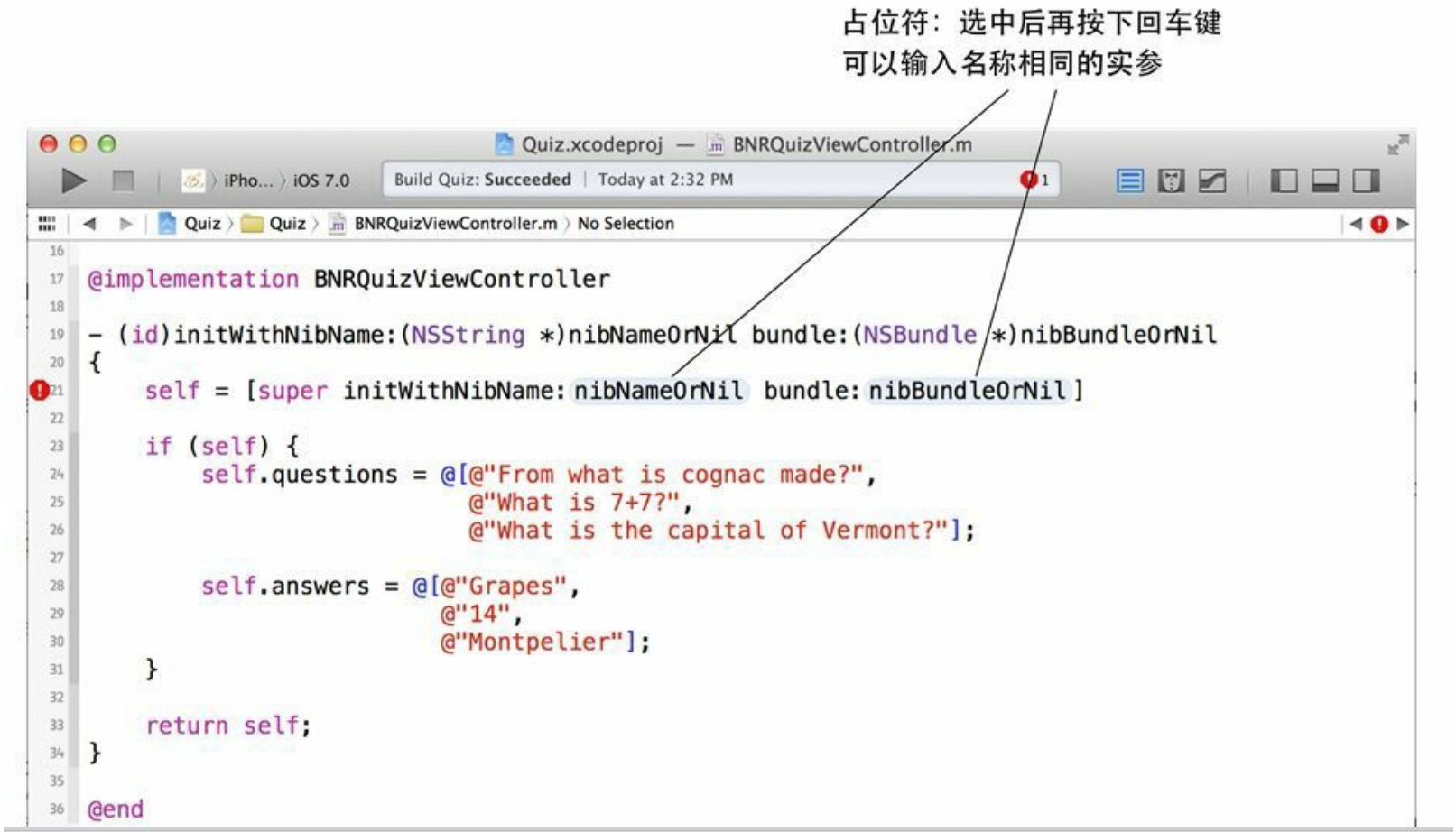


图1-21 代码补全时的占位符示例和错误示例

在图1-21中，initWithNibName:bundle:的第一行实现中的nibNameOrNil和nibBundleOrNil都是占位符。Xcode在显示占位符时，会将带特殊颜色的圆角矩形作为其背景，所以很容易辨认。要修正这类错误，需要选择占位符，输入真正的实参。这样，之前的圆角矩形背景就会消

失，代码也能通过编译。

读者在这里可以直接使用占位符的名称作为实参名称。只要选中占位符再按下回车键就可以了，Xcode会自动填入相同的实参名称。

在使用代码补全功能时，Xcode经常会给出和实际需要不同的代码但又很类似的备选项。Cocoa Touch所使用的命名约定（naming convention）会导致不同的方法、类型和变量拥有类似的名称。因此，请读者一定要注意检查，不要不经确认就选择Xcode给出的第一个备选项。

1.8 大功告成

现在读者已经完成了视图对象和控制器对象的创建、设置和关联，也在模型对象中存储了问题和答案。剩下的两项工作是：

•在BNRQuizViewController中实现showQuestion:和showAnswer:动作方法。

•在BNRAppDelegate中创建和显示BNRQuizViewController对象。

实现动作方法

在BNRQuizViewController.m文件中为showQuestion:和showAnswer:添加以下代码：

```
...
// 返回新对象的地址
return self;
}
- (IBAction)showQuestion:(id)sender
{
// 进入下一个问题
self.currentQuestionIndex++;
// 是否已经回答完了所有问题？
if (self.currentQuestionIndex == [self.questions count]) {
// 回到第一个问题
self.currentQuestionIndex = 0;
}
// 根据正在回答的问题序号从数组中取出问题字符串
NSString *question = self.questions[self.currentQuestionIndex];
// 将问题字符串显示在标签上
self.questionLabel.text = question;
```

```
    // 重置答案字符串
    self.answerLabel.text = @"???";
    }
    - (IBAction)showAnswer:(id)sender
    {
    // 当前问题的答案是什么？
    NSString *answer = self.answers[self.currentQuestionIndex];
    // 在答案标签上显示相应的答案
    self.answerLabel.text = answer;
    }
    @end
```

在屏幕上显示视图控制器

如果现在运行Quiz应用，读者将只能看到一个空白的屏幕，无法看到在BNRQuizViewController.xib文件中创建的用户界面。为了在屏幕上显示用户界面，必须将视图控制器和应用中的另一个控制器关联——BNRAppDelegate。

使用Xcode开发iOS应用时，所有应用模板都会自动帮读者创建一个应用程序委托（app delegate）。应用程序委托是每一个iOS应用都必须具备的启动入口。

应用程序委托负责管理应用的UIWindow对象。UIWindow对象表示应用唯一的主窗口。为了在屏幕上显示BNRQuizViewController，需要将它设置为UIWindow对象的根视图控制器（root view controller）。

ios应用启动完毕后，为了向用户显示界面，系统会做一些额外工作。在用户看到应用界面之前，应用程序委托会收到一条消息：application:didFinishLaunchingWithOptions:，可以在这条消息中添加应用的初始化代码。初始化代码通常用来确保在用户看到界面时，应用已经处于正确的设置。

在项目导航面板中选择BNRAppDelegate.m文件。在application:didFinish-LaunchingWithOptions:方法中创建BNRQuizViewController对象，并将它设置为UIWindow对象的根视图控制器。请读者添加以下代码：

```
    #import "BNRAppDelegate.h"
```

```
#import "BNRQuizViewController.h"

- (BOOL)application:(UIApplication *)application

didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

{

self.window = [[UIWindow alloc] initWithFrame:

[[UIScreen mainScreen] bounds]];

// 在这里添加应用启动后的初始化代码

BNRQuizViewController *quizVC = [[BNRQuizViewController alloc] init];

self.window.rootViewController = quizVC;

self.window.backgroundColor = [UIColor whiteColor];

[self.window makeKeyAndVisible];

return YES;

}
```

现在，应用启动完毕后会创建BNRQuizViewController对象，并通过init方法初始化该对象，加载由BNRQuizViewController.xib文件编译而来的NIB文件（init方法是通过initWithNibName:bundle:方法加载NIB文件的，本书6.4节会介绍此过程），然后将它设置为UIWindow对象的根视图控制器。

Quiz应用已经全部开发好了，读者可以体验一下自己开发的应用了。

1.9 在模拟器上运行应用

本节将在模拟器上运行Quiz应用，之后读者会学习如何在真实的设备上运行应用。首先在Xcode工具栏上找到标题为Scheme的弹出式菜单（见图1-22）。

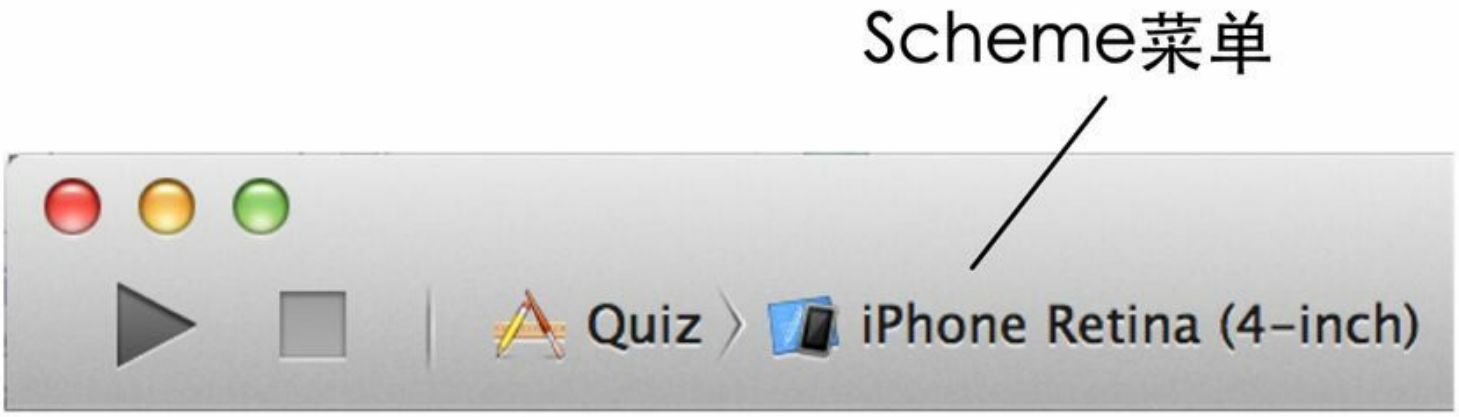


图1-22 选择iPhone Retina(4-inch)

如果Scheme菜单显示的是iPhone Retina（4-inch），表示应用将在模拟器上运行。如果显示的类似于Christian's iPhone（某某的iPhone），请读者点击菜单选择iPhone Retina（4-inch）。

本书使用iPhone Retina（4-inch），它和iPhone Retina（3.5-inch）的唯一区别是屏幕高度。如果选择在3.5英寸模拟器上运行Quiz应用，部分界面可能会被遮住。读者将在第15章中学习如何自动适配不同设备的屏幕尺寸，包括iPhone和iPad。

现在请单击工具栏上那个iTunes风格的播放按钮。这样Xcode就开始构建（编译）并运行Quiz应用了。“构建并运行”是一项很常用的功能，键盘快捷键是Command-R。记住并使用快捷键会方便很多。

构建项目发生错误或警告，可以单击导航面板选择条中的图标，在问题导航面板中查看出现的问题（见图1-23）。

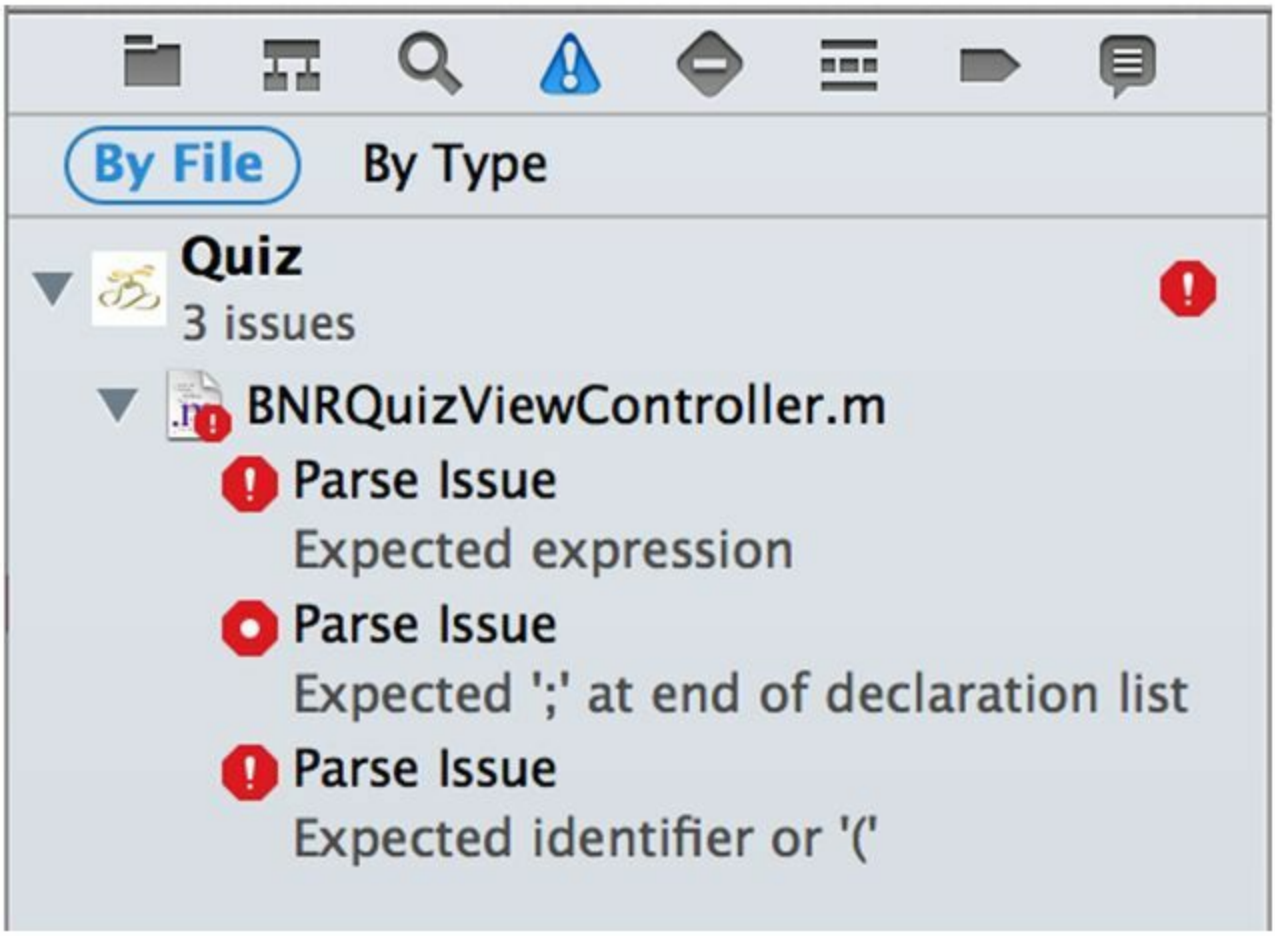


图1-23 列出错误和警告信息的问题导航面板

单击问题导航面板列出的条目，可以打开相应的源文件，并定位至产生问题的行。请读者参照本书中的代码，找到并修正所有的错误（例如输入错误），然后重新构建项目。重复这个过程，直到编译通过为止。

编译通过后，Xcode会在iOS模拟器中运行Quiz应用。第一次打开模拟器可能比较慢。

现在请读者体验自己开发的应用。按下Show Question按钮，视图上方的标签会显示一道新题目；按下Show Answer按钮会显示正确答案。如果Quiz应用不能正确工作，请检查BNRQuizViewController.xib中的关联。

1.10 安装应用

至此，读者已经编写并在模拟器上运行了自己的第一个应用。下将该应用装入设备。

读者要先从Apple公司得到一份开发者证书，才能将应用装入开发用的设备。已注册的iOS开发者（需支付一定的费用）都可以得到由Apple公司签发的开发者证书。Xcode会使用该证书为代码“签名”，使之能在设备上运行。没有有效的证书，应用无法在设备上运行。

Apple公司的Developer Program Portal网站（http://developer.apple.com）列有获得有效证书所需的所有说明和资源。设置流程的界面会经常发生变化，所以在此不做详细介绍。读者可以参考本书网站的一份详尽指南：http://www.bignerdranch.com/iOS_device_provisioning。

下面列出上述过程中的4个重要概念，以便读者能对其有一个更深入的了解。

Developer Certificate	这份证书文件会通过 Keychain Access（钥匙串访问）程序加入读者当前使用的钥匙串。Xcode 会使用这份证书为代码签名
App ID	应用程序标识（application identifier）是一串能够在 App Store 中唯一标识应用的字符串。应用程序标识通常为这种形式：com.bignerdranch.AwesomeApp，其中应用名称跟在公司名称后面 provisioning profile 中的应用程序标识必须和应用的程序包标识（bundle identifier）匹配。针对开发的 profile，App ID 可以包含通配符（wildcard character），匹配任意的程序包标识。要查看 Quiz 应用的程序包标识，可以选中位于项目导航面板顶部的 Quiz 条目，选中 TARGETS 中的 Quiz，最后选择 General 面板
Device ID（UDID，设备标识）	每个 iOS 设备都有一个唯一的标识
Provisioning Profile	要在开发设备和计算机上保存 provisioning profile 文件。该文件对应这些设置：一份开发者证书、一个应用标识和一组设备标识（只有和这些标识匹配的设备才能安装应用）。Provisioning Profile 文件的后缀名是.mobileprovision

Xcode在将应用安装至设备时，会通过计算机上的某个provisioning profile获得合适的证书，并用这份证书为应用的二进制文件签名。接着，开发设备的UDID会和provisioning profile中的某个UDID匹配，应用程序标识会和程序包标识匹配。最后，Xcode会将签名后的二进制文件传入开发设备，经由设备上的同一个provisioning profile确认并最终启动。

运行Xcode，将开发设备（iPhone、iPod touch或iPad）接入计算机。Xcode会自动打开Organizer窗口，也可以随时在Window→Organizer菜单中打开。选中Organizer窗口顶部的Devices项，可以列出所有的provisioning信息。

要在设备上运行Quiz应用，必须要求Xcode将应用装入设备，而不是模拟器。单击工具栏上的弹出式菜单Scheme，选择列表中的iOS Device（见图1-24）。如果没有iOS Device这项，就寻找类似Christian's iPhone（某某的iPhone）这样的选项。

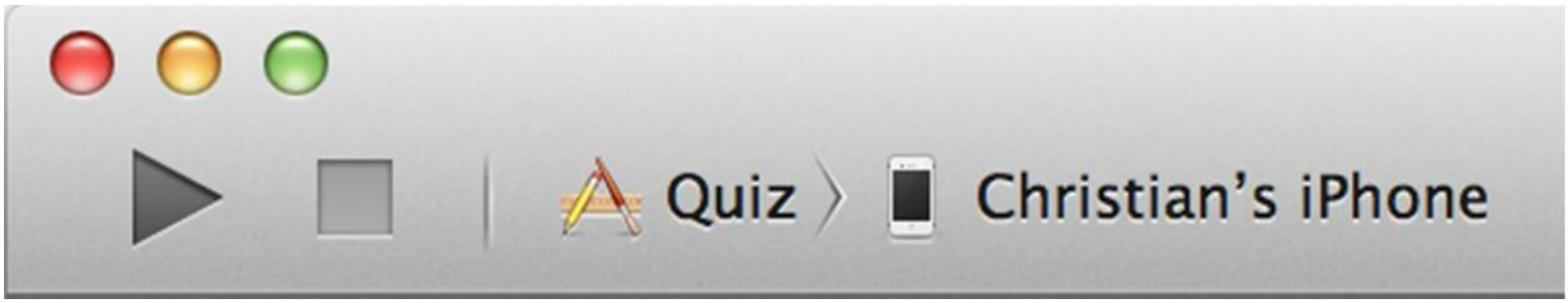


图1-24 选择设备

构建并运行应用（Command-R），稍后应用就会出现在设备上。

1.11 应用图标

等Xcode运行Quiz应用(装入设备或者使用模拟器)后，打开设备的主屏幕(Home screen)，可以看到Quiz应用的图标：一块白板。下面为Quiz应用设置一个更好的图标。

应用图标(application icon)是一张图片，用于在主屏幕上指代应用。不同的设备对图标的尺寸要求也不同，具体的要求如表1-1所示。

表1-1 不同设备的应用图标尺寸

设　　备	应用图标尺寸
iPhone/iPod touch（iOS 7）	120 像素×120 像素(@2x)
iPhone/iPod touch（iOS 6 及之前版本）	57 像素×57 像素 114 像素×114 像素(@2x)
iPad（iOS 7 及之前版本）	72 像素×72 像素 144 像素×144 像素(@2x)

提交给App Store的应用需要针对每一种(可以运行该应用的)设备类型提供一个符合尺寸要求的图标。例如，如果读者计划只支持运行iOS 7及更高版本的iPhone和iPod touch，那么只需要提供一个图标就可以了(尺寸请参考表1-1)。但是，如果要开发一个支持iOS 6及更高版本的通用应用，就必须提供五种尺寸的图标，分别是两个iPad图标和三个iPhone/iPod touch图标。

本书已经为Quiz应用准备好了图标文件(大小为120像素×120像素)。读者可以从http://www.bignerdranch.com/solutions/iOSProgramming4ed.zip下载该图标(该文件还包含其他章节所需的资源)。解压iOSProgramming4ed.zip，在解压后的文件夹里找到Resources目录下的Icon@2x.png。

下面要将这个图标作为资源(resource)加入应用程序包。应用中的文件通常可以分为代码和资源两类。程序本身由代码构成(例如BNRQuizViewController.h和BNRQuizViewController.m)。资源则是图片和声音这类应用运行时会用到的文件。XIB文件在应用运行时被编译为NIB文件并载入，也属于资源。

选中项目导航面板中的Images.xcassets条目。再选中位于资源列表左边的AppIcon(见图1-25)。

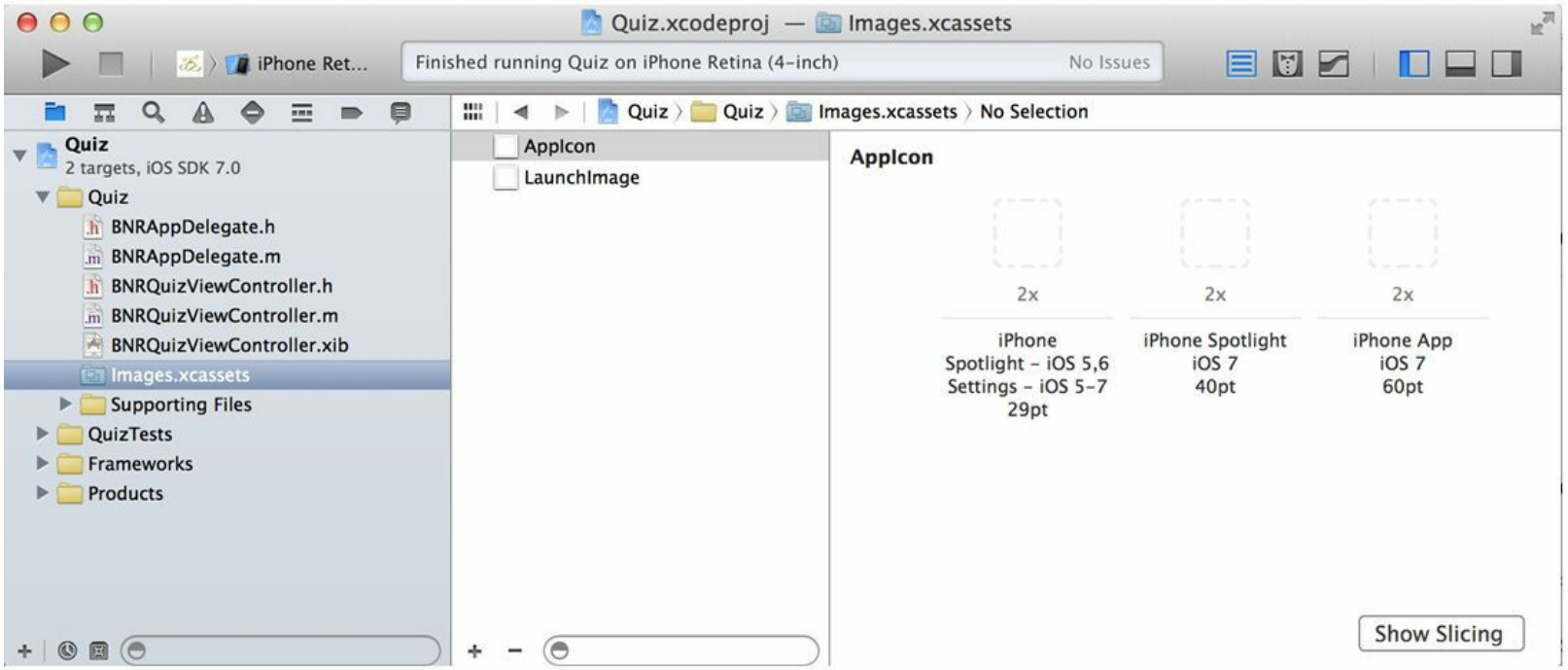


图1-25 打开资源目录

这个面板叫做资源目录(Asset Catalog)，读者可以在这里管理项目需要用到的所有图片。

将Icon@2x.png从Finder拖曳至AppIcon区域的设置块上(见图1-26)。Xcode会将文件拷贝至存放Quiz项目的目录，并在资源目录中加入相应的引用(references)。要确认Xcode是否正确地复制了图标文件，可以Control-单击资源目录中的图标文件，然后选择弹出菜单中的Show in Finder。

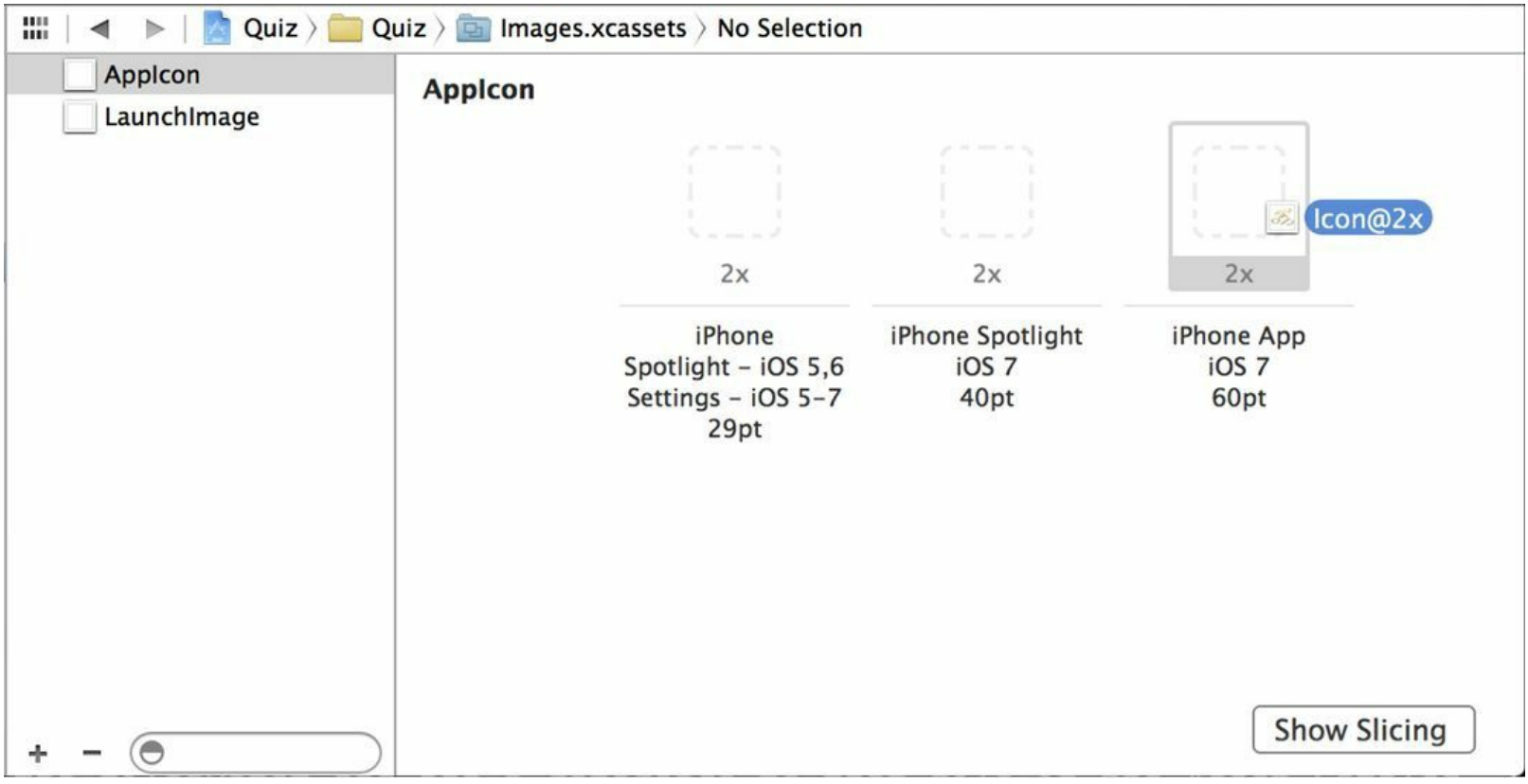


图1-26 在资源目录中添加应用图标

构建并运行应用。退出应用后，可以在主屏幕看到带有BNR标记的Quiz图标。

如果无法看到添加后的图标，请删除设备或模拟器上的应用后重新运行应用。在设备上，可以像删除其他应用一样删除Quiz，在模拟器上还可以用更简单的方法——还原模拟器。打开模拟器，在菜单中选择iOS Simulator→Reset Content and Settings…（iOS模拟器→还原内容和设置…）。这样会将模拟器还原到默认设置并删除所有应用。再次运行应用就会看到新图标了。

1.12 启动图片

启动图片(launch image)是另一个可以在资源目录中管理的应用选项。系统在载入应用时，会先显示应用的启动图片(如果没有设置启动图片，那么系统会在载入应用时显示黑屏)。iOS中的启动图片有其特定的作用:向用户传达“应用正在启动”的信息，并描绘应用启动后的用户交互界面。因此，好的启动图片应该是应用的空白(content-less)截图。例如，在系统自带的时钟(Clock)应用的启动图片中，底部有四个选项卡，全部处于未选中状态。当应用启动完成后，用户上一次选择的选项卡会被选中，同时界面也显示出来(注意，系统会在应用完成启动后替换掉启动图片。启动图片不会成为应用的背景图片)。

存放应用图标文件的Resources目录下还有两张启动图片:Default@2x.png和Default-568h@2x.png。打开资源目录，选择LaunchImage，和添加应用图标的方法相同，将两张启动图片拖放到相应的位置。

构建并运行应用。当系统启动应用时，应该可以短暂地看到启动图片。

为什么本书提供了两张启动图片？启动图片必须和运行应用的设备屏幕尺寸相同，因此读者需要分别准备3.5英寸和4英寸Retina屏幕的启动图片。注意，如果应用需要支持运行iOS 6及更低版本的iPhone和iPod touch，还需要添加一张3.5英寸非Retina屏幕的启动图片。表1-2列出了不同类型的设备所需的图片尺寸。

表1-2 不同设备的启动图片尺寸

设　　备	启动图片尺寸（竖屏 / 横屏）
iPhone/iPod touch（不支持 Retina 显示屏）	(320×480)像素 /(480×320)像素
iPhone/iPod touch（支持 Retina 显示屏，3.5 英寸）	(640×960)像素 /(960×640)像素
iPhone/iPod touch（支持 Retina 显示屏，4 英寸）	(640×1136)像素 /(1136×640)像素
iPad（不支持 Retina 显示屏）	(768×1024)像素 /(1024×768)像素
iPad（支持 Retina 显示屏）	(1536×2048)像素 /(2048×1536)像素

(注意，表1-2列出的是设备的屏幕分辨率。启动应用时，实际的系统状态条会浮动在启动图片上。)恭喜读者已经成功开发并安装了自己的第一个应用。下面开始深入介绍各项相关知识。

第2章 Objective-C

开发iOS应用需要使用Objective-C语言和Cocoa Touch框架。Objective-C源自C语言，是C语言的扩展。Cocoa Touch框架则是一个Objective-C类的集合。本书假定读者略懂C语言和面向对象编程。如果读者对C语言和面向对象编程尚有疑问，推荐先阅读《Objective-C编程》（华中科技大学出版社，2012）。

本章将向读者介绍Objective-C的基础知识，并开发一款名为RandomItems的命令行工具。因为后续章节会用到本章所创建的BNRItem类，所以即使读者熟悉Objective-C，也建议通读本章，完成RandomItems项目。

2.1 对象

假设读者需要通过某种方式在程序中描述一场聚会。该聚会有若干特有的属性，例如聚会名称、日期和一份受邀者清单。此外，还需要让“聚会”做些事情，例如向所有受邀者发送一封提醒电邮、打印名牌或取消聚会。

如果使用C语言，则可以定义一个结构（structure），用于保存描述聚会的数据。这个结构会有数据成员，和聚会的属性一一对应。每个数据成员都有名称和类型。创建聚会时，可以通过malloc函数分配一块足够大的内存，存放相应的结构。

如果使用Objective-C语言，则要使用类（class）而不是结构来展现一场聚会。类就像是制造对象的饼干模子。通过Party类可以创建特定的对象，这些对象都是Party类的实例（instance）。每个Party对象都能为某一个特定的聚会保存数据（见图2-1）。

所有的对象，包括Party对象，都是内存中的一块数据。对象通过实例变量（instance variable）保存属性的值。（本书有时会将实例变量简称为“ivars”。）在Objective-C语言中，实例变量的变量名之前通常会加上一个下画线。因此，可以为Party对象定义以下实例变量：_name（名称）、_date（日期）和_budget（预算）。

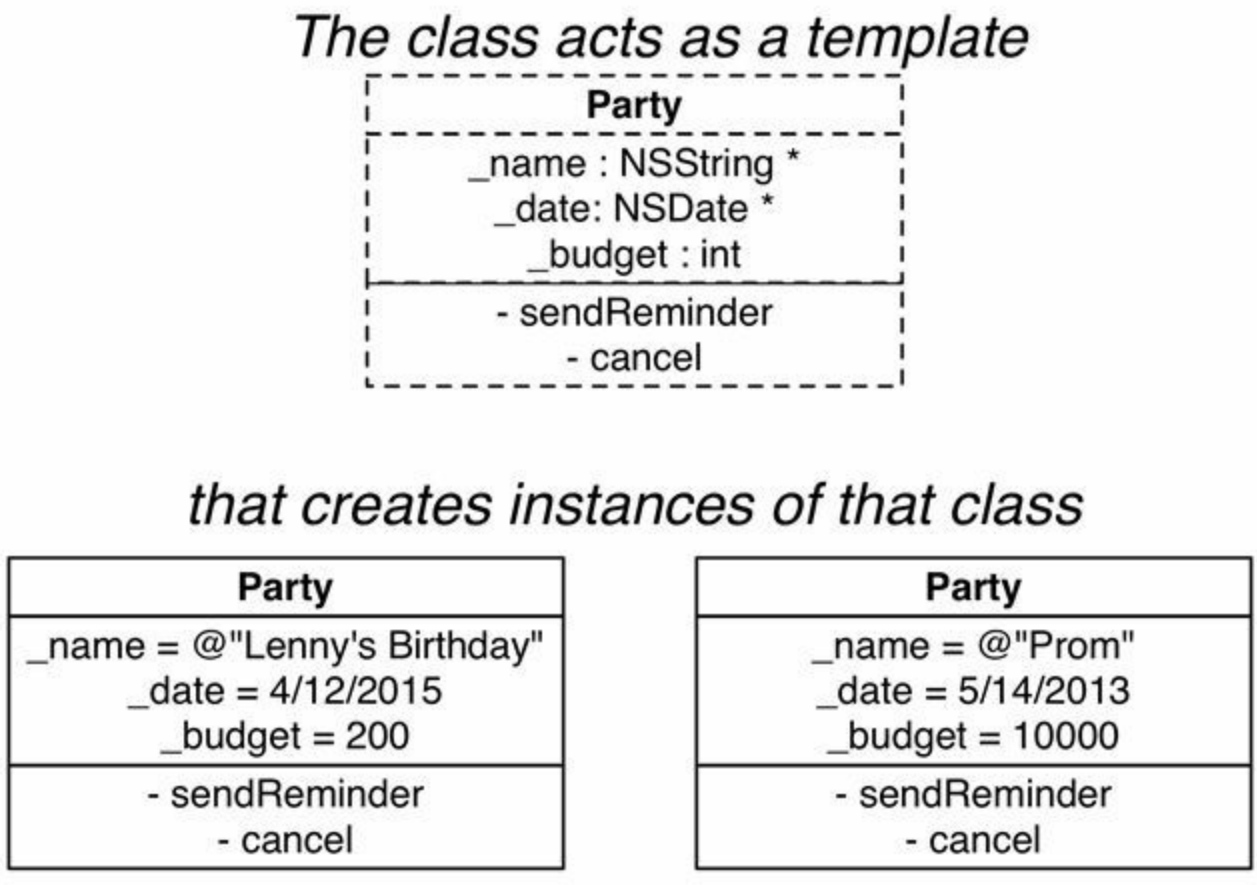


图2-1 Party类及其对象

C结构是一块内存，对象也是一块内存。C结构有数据成员，每个数据成员有名称和类型。与之类似，对象有实例变量，每个实例变量也有名称和类型。

C结构和Objective-C类之间有一个重要差别:类有方法(method)。方法和函数类似,也有名称、返回类型和一组期望传入的参数。此外,方法还可以访问对象的实例变量。要调用某个对象的方法,可以向该对象发送相应的消息(message)。

2.2 使用对象

要使用某个类的对象，必须先得到一个指向该对象的变量（variable）。这类“指针变量”保存的是对象在内存中的地址，而不是对象自身（所以是“指向”某个对象）。下面为一个“指向对象的变量”的声明示例：

```
Party*partyInstance;
```

这段声明代码只创建了一个指针，并没有创建任何Party对象，仅仅是声明了一个可以指向某个Party对象的指针变量，变量名是partyInstance。请注意变量名之前没有加下画线，所以它不是实例变量。

创建对象

对象是有生命周期的：首先被创建出来，然后接收消息，最后在不需要时被释放。

向某个类发送alloc消息，可以创建该类的对象。类在收到alloc消息后，会在内存中创建对象（在堆上创建，和调用malloc函数的效果相同），并返回指向新对象的指针，这时程序就可以将这个指针保存在某个变量中：

```
Party*partyInstace=[Party alloc];
```

上面这行代码创建了一个指向Party对象的指针。程序得到指向某个对象的指针后，就可以向该对象发送消息。对新创建的对象，必须先向其发送一个初始化消息（initialization message）。虽然向类发送alloc消息能够创建对象，但是在完成初始化之前，新创建的对象还无法正常工作。

```
Party *partyInstance = [Party alloc];

[partyInstance init];
```

因为任何一个对象都必须在创建并且初始化后才能使用，所以上述两个消息应该写在一行代码里，其代码如下：

```
Party *partyInstance = [[Party alloc] init];
```

这种将两个消息合写在一行代码中的做法称为嵌套消息发送（nested message send）。程序会先执行最里面那个方括号中的代码，所以Party类会先收到alloc消息。接着，alloc方法会返回指向新创建对象的指针。最后，未初始化的对象会收到init消息，返回初始化后的对象指针，并将指针保存在变量中。

发送消息

一旦某个对象完成了初始化，就可以向其发送更多其他的消息。

下面进一步介绍消息发送语法的组成结构。首先，消息必须写在一对方括号中。方括号中的消息包含如下三个部分。

接收方（receiver）	指针，指向执行方法的对象
选择器（selector）	需要执行方法的方法名
实参（arguments）	以变量形式传给方法的数值

接收方(receiver)指针，指向执行方法的对象选择器(selector)需要执行方法的方法名实参(arguments)以变量形式传给方法的数值

以Party类为例，向Party对象发送addAttendee:消息，可以添加参加聚会的客人：

[partyInstance addAttendee:somePerson];

向partyInstance(接收方)发送addAttendee:消息会触发addAttendee:方法(取决于选择器)，并传入somePerson(实参)。

addAttendee消息只有一个参数，但是Objective-C语言中的方法可以有很多实参，或者没有实参，例如init消息就没有实参。

收到邀请的客人要回复是否参加，并告诉主人会带来的食物。因此，Party对象还要一个名为addAttendee:withDish:的方法。这个方法有两个实参："客人"和他准备带来的"食物"。每个实参都会和选择器中的相应标签配对，每个标签都会以冒号结尾。所有的标签合在一起构成选择器(见图2-2)。

图2-2 消息发送代码的各个组成部分

标签和参数必须配对的语法是Objective-C的一项重要特性。在其他语言中，上面这行代码可能会写成：

partyInstance.addAttendeeWithDish(somePerson, deviledEggs);

在这些语言中，传入函数的各个数值分别对应哪个参数并不明显。在Objective-C中，每个数值都会和相应的标签配对，代码如下：

```
[partyInstance addAttendee:somePerson withDish:deviledEggs];
```

读者可能要花些时间来习惯这种语法，一旦熟悉后，就会发现这种在选择器中插入实参的语法能更容易读懂代码。这里要记住，每一组方括号只对应一条需要发送的消息。虽然这里的addAttendee:withDish:有两个标签，但仍只是一条消息，发送这条消息只会触发一个方法。

在Objective-C中，方法的唯一性取决于方法名。因此，即使参数类型或返回类型不同，一个类也不能有两个名称相同的方法。但是不同的方法可包含某个相同的标签，前提是这些方法的名称并不完全相同。以Party类为例，它有addAttendee:和addAttendee:withDish:两个方法。这是两个不同的方法，不共享任何代码。

此外，还要注意消息和方法之间的区别：方法是指一块可以执行的代码，而消息是指要求类或对象执行某个方法的动作。此外，消息的名称和将要执行的方法的名称一定是相同的。

释放对象

将指向对象的变量设置为nil，可以要求程序释放该对象，代码如下：

```
partyInstance = nil;
```

这行代码会释放partyInstance变量所指向的对象（实际情况会更复杂，第3章会详细介绍内存管理方面的知识）。

nil是值为0的指针（对应C语言中的NULL，Java语言中的null）。一个值为nil的指针通常代表其没有指向任何对象。仍以Party对象为例，举办聚会需要场地（venue）。当主办方正在决定应该在何处举办聚会时，代表场地的实例变量venue的值将是nil。通过判断指针变量的值是否为nil，可以实现相应的逻辑处理，代码如下：

```
if (venue == nil) {

    [organizer remindToFindVenueForParty];

}
```

Objective-C程序员通常会使用更短小精炼的语法来判断某个指针是否为nil，代码如下：

```
if (!venue) {

    [organizer remindToFindVenueForParty];

}
```

因为！运算符的意思是“不”，所以if（！venue）的意思是“如果venue的值为空”，或者如果venue的值是nil，那么这条表达式的运算结果将为真（true）。

Objective-C允许向某个值为nil的变量发送消息，且不会发生任何事情。在其他语言中，向空指针（值为0的指针）发送消息是非法的，因此，通常要先检查指针是否为空，代码如下：

```
//是否为nil？

if(venue){

    [venue sendConfirmation];

}
```

在Objective-C中，因为程序会忽略发送给nil的消息，所以无需做这样的检查，直接发送消息即可，代码如下：

```
[venue sendConfirmation];
```

如果venue的值是nil，那么向其发送sendConfirmation消息不会有任何结果（反之，如果某个应用应该完成某项功能，但实际没做任何事情，那么问题很可能出在某个指针变量上——原本应该指向某个对象的指针，实际的值却是nil）。

理论知识就学习到这里，现在请读者跟着本章完成一个新项目——RandomItems。

2.3 编写命令行工具PandomItems

RandomItems不是iOS应用，而是命令行工具。命令行工具不用开发复杂的用户界面，所以能集中精力学习Objective-C。本章和第3章的重点是学习Objective-C，第4章再开发iOS应用。

运行Xcode，选择File→New→Project…在新出现的窗口左侧选择OS X下的Application，然后选择右侧面板上方的Command Line Tool（命令行工具），如图2-3所示。单击Next按钮。

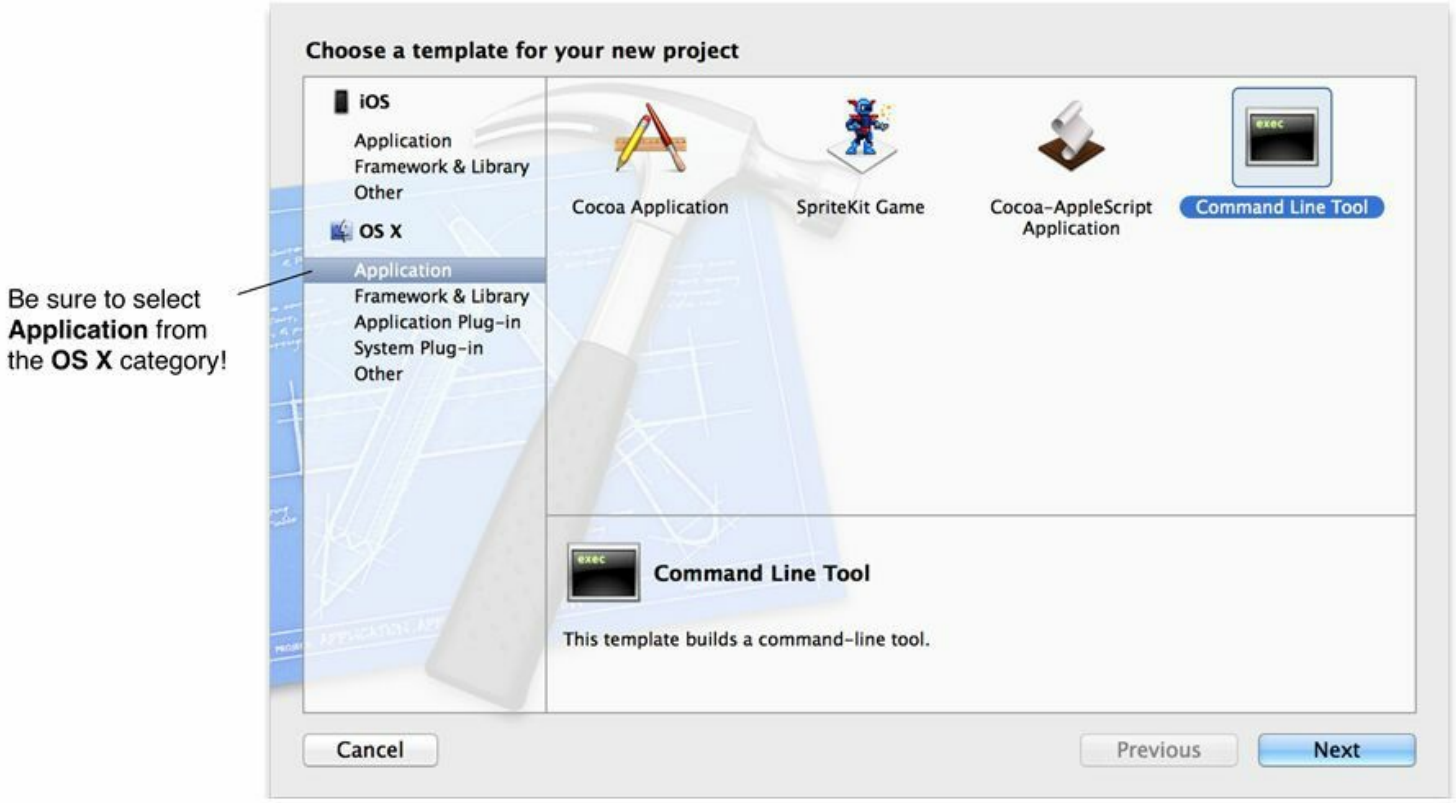


图2-3 创建一个命令行工具项目

在新出现的面板中，将项目命名为RandomItems，项目类型选择Foundation（见图2-4）。

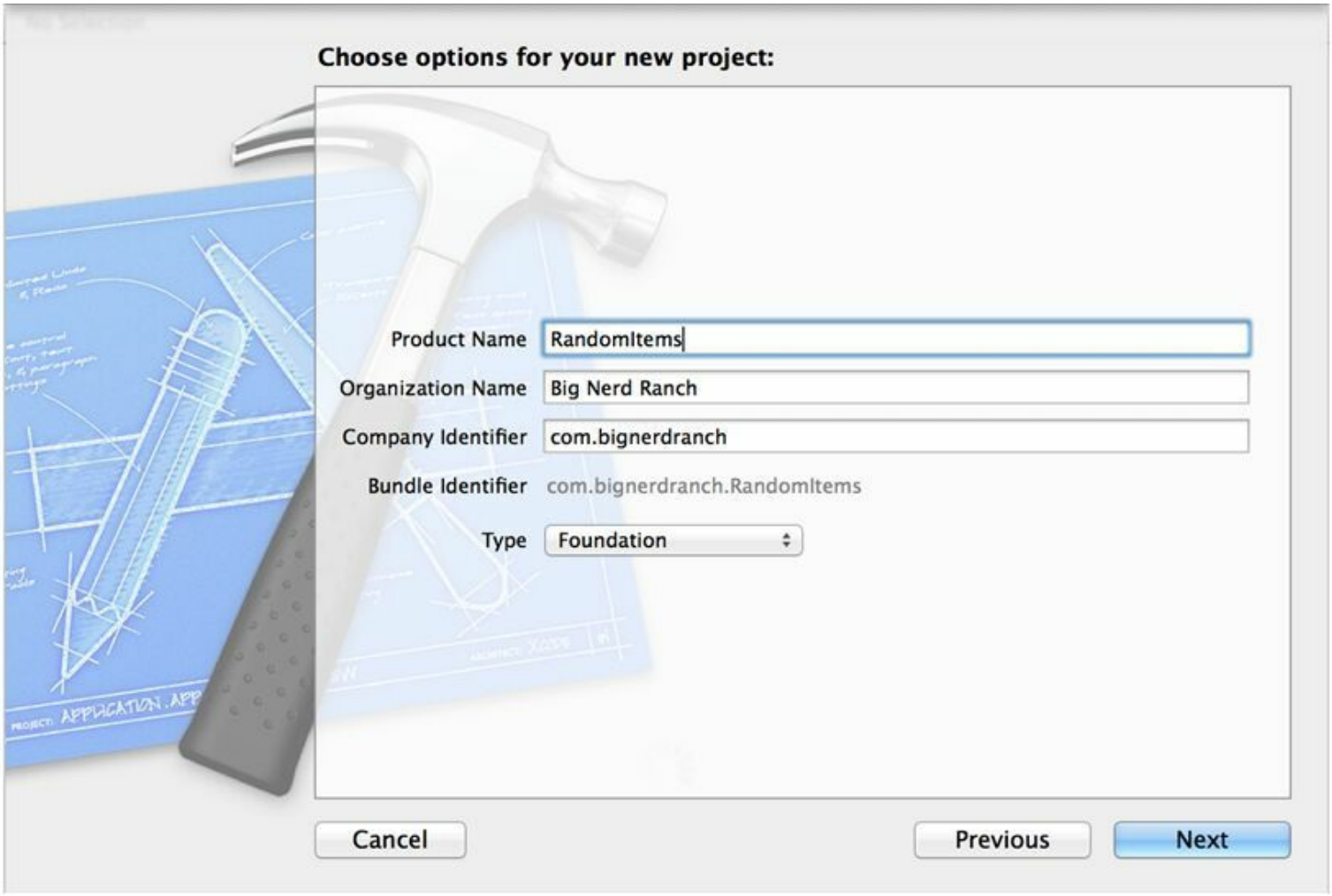


图2-4 为项目命名

单击Next按钮，Xcode会提示保存项目。保存好项目，本书之后的项目还会用到RandomItems的部分代码。

RondomItems的第一个版本将创建一个包含4个字符串的数组。数组包含一组按序排列的对象，可以通过索引存取。其他语言可能会将类似的对象称为list（队列）或vector（向量）。数组中第一个对象的索引都是0。

创建数组后，将遍历数组打印所有字符串。Xcode控制台中的输出会类似图2-5所示。

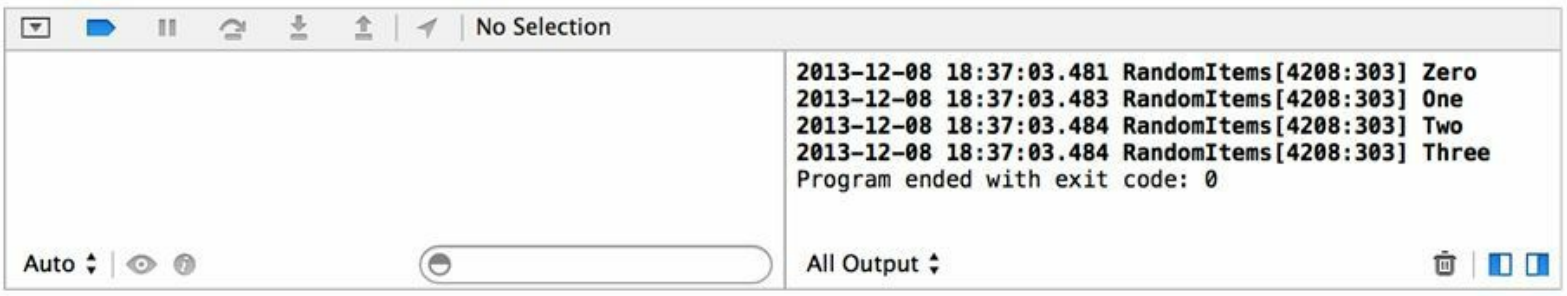


图2-5 控制台的输出

RandomItems中有5个对象：1个NSMutableArray对象，4个NSString对象，如图2-6所示。

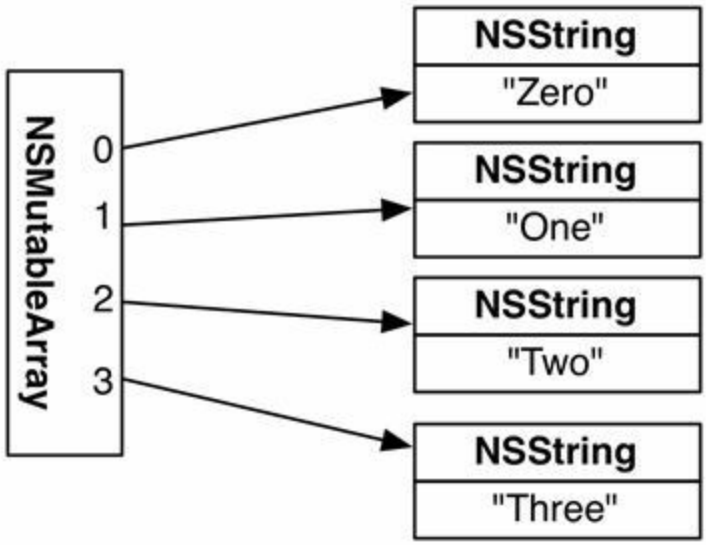


图2-6 NSMutableArray对象包含指向NSString对象的指针

在Objective-C中，数组所包含的“对象”并不是对象自身，而只是指向对象的指针。当程序将某个对象加入数组时，数组会保存该对象在内存中的地址。

现在请关注NSMutableArray和NSString。NSMutableArray是NSArray的子类。Objective-C中的类是以层次结构（hierarchy）的形式存在的。除了整个层级结构的根类NSObject外，每个类都有一个且只有一个父类（superclass）并继承其父类的行为。

图2-7展示了NSMutableArray和NSString的一些方法和层级结构。

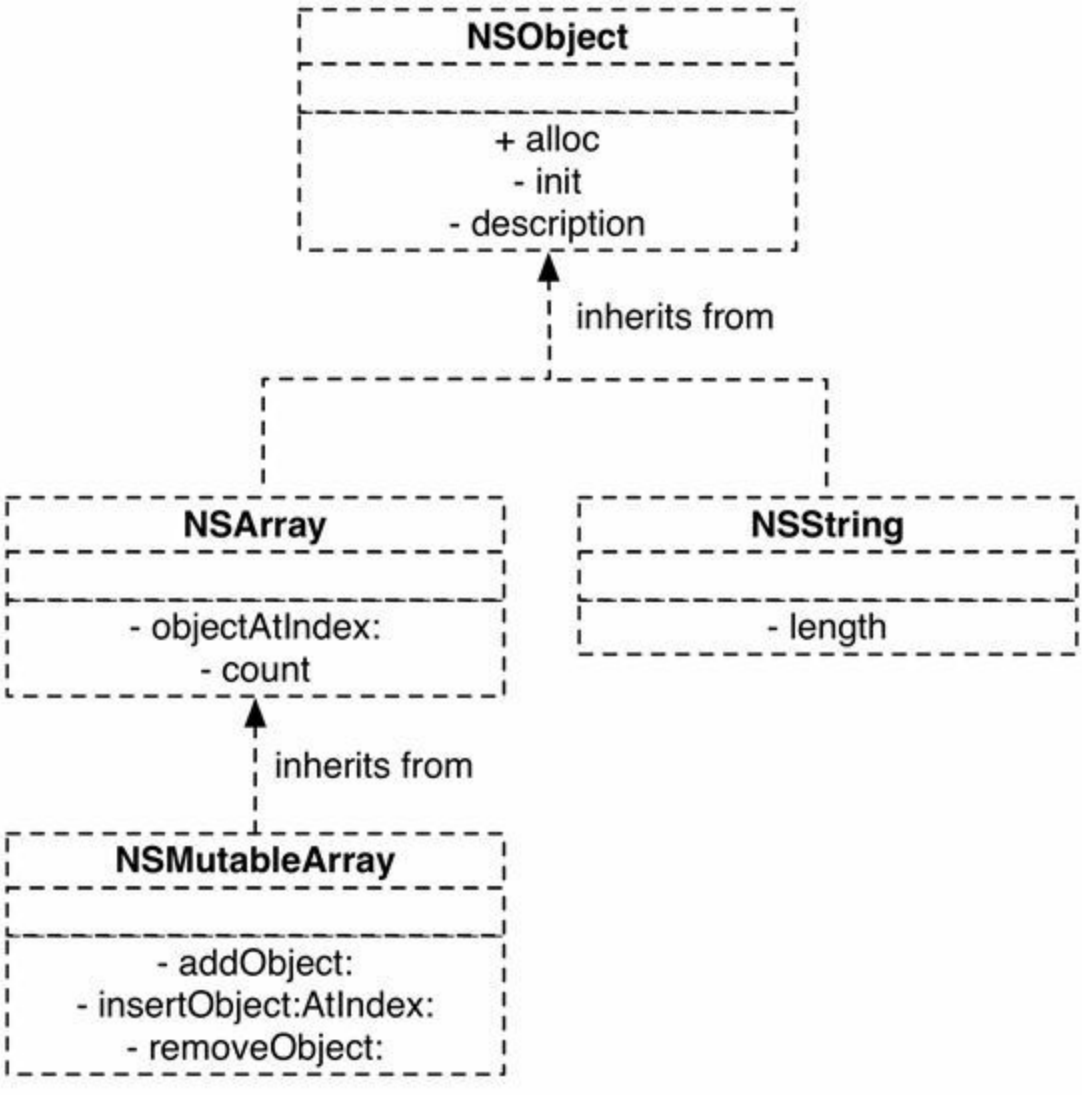


图2-7 部分类的层级结构

NSObject作为位于层次结构顶部的父类，其职责是实现Cocoa Touch框架中所有对象的基本行为。所有的类都会继承NSObject中的方法和实例变量。NSObject实现了很多方法，其中两个方法是alloc和init（见图2-7）。因此所有的类都可以使用alloc和init方法创建自己的对象。

子类可以通过添加方法和实例变量扩充其继承自父类的行为：

•NSString扩充NSObject的行为，可以存储和处理字符串，也添加了很多方法，如length，可以返回一个字符串的长度。

•NSArray扩充NSObject的行为，可以按索引存取对象（objectAtIndex:），也可以获取存储的对象数量（count）。

•NSMutableArray扩充NSArray的行为，可以动态增加和删除对象。

创建数组并填充字符串

现在开始在代码中使用这些类。在项目导航面板中选择main.m文件，Xcode会在编辑区域打

开该文件。读者可以看到，该文件中的部分代码是由Xcode自动生成的，其中的main函数是C（或Objective-C）程序的入口点（entry point）。

删除用NSLog打印“Hello, World!”的那行代码，添加新代码，创建NSMutableArray对象并添加4个字符串，最后释放NSMutableArray对象：

```
#import

int main(int argc,const char*argv[])

{

@autoreleasepool{

// 在这里输入代码

NSLog(@"Hello, World!");

//创建一个NSMutableArray对象，并用items变量保存该对象的地址

NSMutableArray*items=[[NSMutableArray alloc]init];

//向items所指向的NSMutableArray对象发送addObject:消息

//每次传入一个字符串

[items addObject:@"One"];

[items addObject:@"Two"];

[items addObject:@"Three"];

//继续向同一个对象发送消息，这次是insertObject:atIndex;

[items insertObject:@"Zero" atIndex:0];

//释放items所指向的NSMutableArray对象

items = nil;

}

return 0;

}
```

加入数组的是NSString对象，可以通过在字符串前添加一个“@”前缀来创建一个NSString对象：

NSString *myString = @"Hello, World!";

遍历数组

现在items数组中有4个NSString对象。接下来，遍历数组中的每一个对象并将结果输出至控制台。

可以使用for循环：

```
for (int i = 0; i < [items count]; i++) {

NSString *item = [items objectAtIndex:i];

NSLog(@"%@", item);

}
```

因为数组的索引是从0开始的，所以计数器i的初始值为0。数组的最后一个索引是对象数量减去1，因此计数器的终值是[items count] - 1（因为计数器是整型，为方便起见写成i < [items count]，而不是i <= [items count] - 1）。在循环体中，向数组发送objectAtIndex:消息，根据当前索引获取NSString对象，再输出至控制台。

数组对象所包含的对象个数是一个非常重要的信息。这是因为在获取数组对象所包含的对象时，如果使用的索引大于或等于数组对象所包含对象的个数，程序就会抛出异常（本章结尾处会介绍更多有关异常的知识）。

这段代码当然可以正常工作，但是Objective-C提供了一种更好的遍历数组的语法，称为快速枚举（fast enumeration）。快速枚举比传统的for循环简洁很多，出错概率更低，而且经过编译器的优化，通常比for循环更快。

在main.m中，添加以下代码，使用快速枚举遍历items数组：

```
int main (int argc, const char * argv[])

{

@autoreleasepool {

//创建一个NSMutableArray对象，并用items变量保存该对象的地址

NSMutableArray *items = [[NSMutableArray alloc] init];

//向items所指向的NSMutableArray对象发送addObject:消息

//每次传入一个字符串
```

```
[items addObject:@"One"];

[items addObject:@"Two"];

[items addObject:@"Three"];

//继续向同一个对象发送消息，这次是insertObject:atIndex:

[items insertObject:@"Zero" atIndex:0];

//遍历items数组中的每一个item

for (NSString *item in items) {

//打印对象信息

NSLog(@"%@", item);

}

//释放items所指向的NSMutableArray对象

items=nil;

}

returnO;

}
```

快速枚举有一个限制：如果需要在循环体中添加或删除对象，就不能使用快速枚举，否则程序会抛出异常。这时只能设置计数器并使用普通的for循环。

构建并运行应用（Command-R），Xcode会在窗口底部显示一个新面板，称为调试区域（debug area）。显示程序输出结果的控制台位于调试区域右侧（见图2-8）。

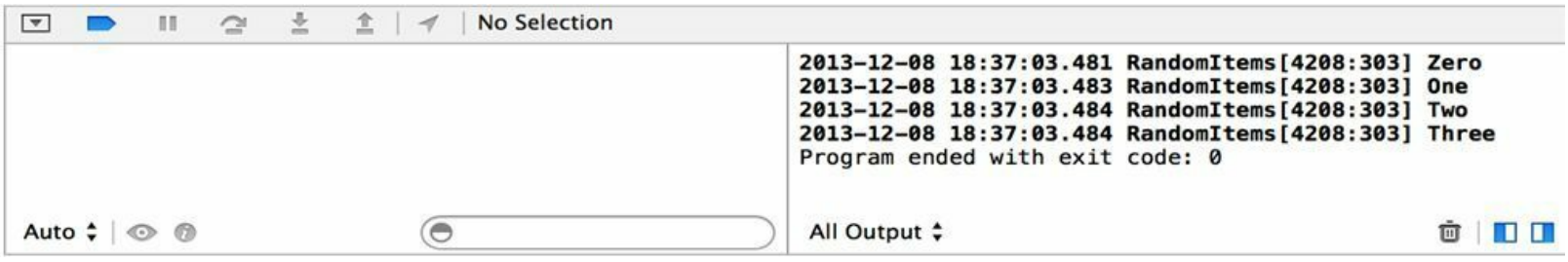


图2-8 调试区域和控制台

如果需要改变这些面板的大小，可以拖曳调试区域和其下面板的边框。实际上，工作空间中的所有区域都可以通过拖曳边框改变大小。

读者已经完成了RandomItems的第一个版本，在开发下一个版本之前，先学习NSLog函数和格式字符串。

格式字符串

NSLog函数可以将某个指定的字符串输出至Xcode的控制台。此外，NSLog的实参个数并不确定，其中的第一个实参是必需的，且必须是NSString对象。这个实参称为格式字符串（format string）。

格式字符串可以包含文字和多个转换说明（token）。格式字符串中的转换说明（也称为格式规格）必须以百分号（%）为前缀。除了传入NSLog函数的第一个实参，每个额外传入的实参都会替换掉格式字符串中的一个转换说明。

转换说明会指定和其相对应的实参的类型。代码如下：

```
int a = 1;

float b = 2.5;

char c = 'A';

NSLog(@"Integer: %d Float: %f Char: %c", a, b, c);
```

上例的控制台输出应该为：

```
Integer: 1 Float: 2.5 Char: A
```

Objective-C的格式字符串基本和C语言相同。但是Objective-C支持一种额外的转换说明：%@，对应的实参类型是指向任何一种对象的指针。

程序在处理格式字符串时，如果遇到%@，则不会将其直接替换为相应位置的实参。程序会先向相应位置的实参发送description消息，得到description方法所返回的NSString对象，然后使用得到的NSString对象替换%@。

因为程序会向%@所对应的实参发送消息，所以这些实参必须是对象。请读者回顾图2-7，可以看到NSObject实现了description方法，因此所有的Objective-C对象也都实现了该方法并可以对应%@。

2.4 创建Objective-C类的子类

本节要创建一个名为BNRItem的NSObject子类（见图2-9）。

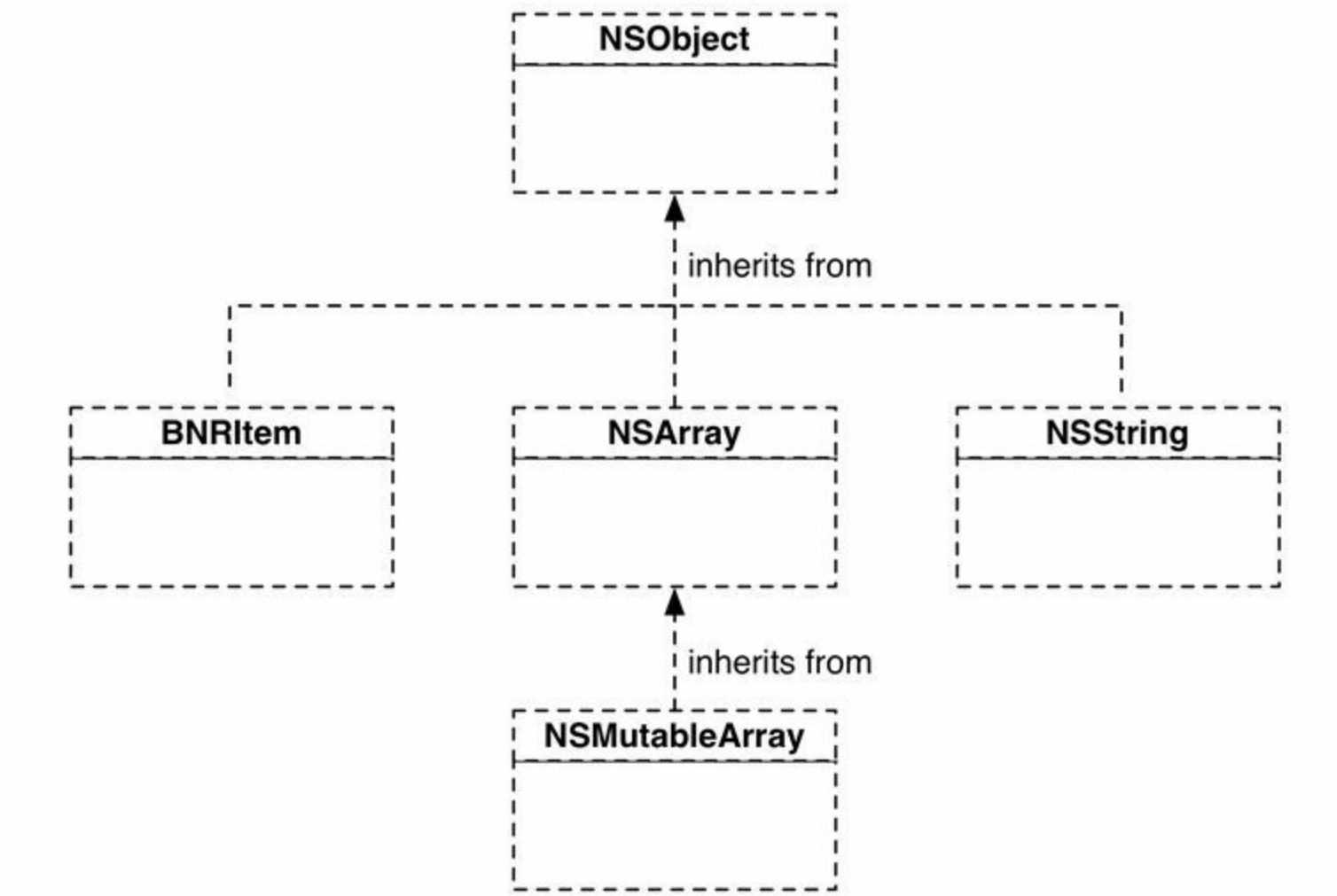


图2-9　BNRItem类的层级结构

BNRItem对象表示某人在真实世界拥有的一件物品，例如笔记本电脑、自行车、背包等。在模型-视图-控制器设计模式中，BNRItem属于模型类，BNRItem对象用来存储私人物品信息。

创建BNRItem类之后，将使用BNRItem代替NSString，在items数组中存储BNRItem对象（见图2-10）。

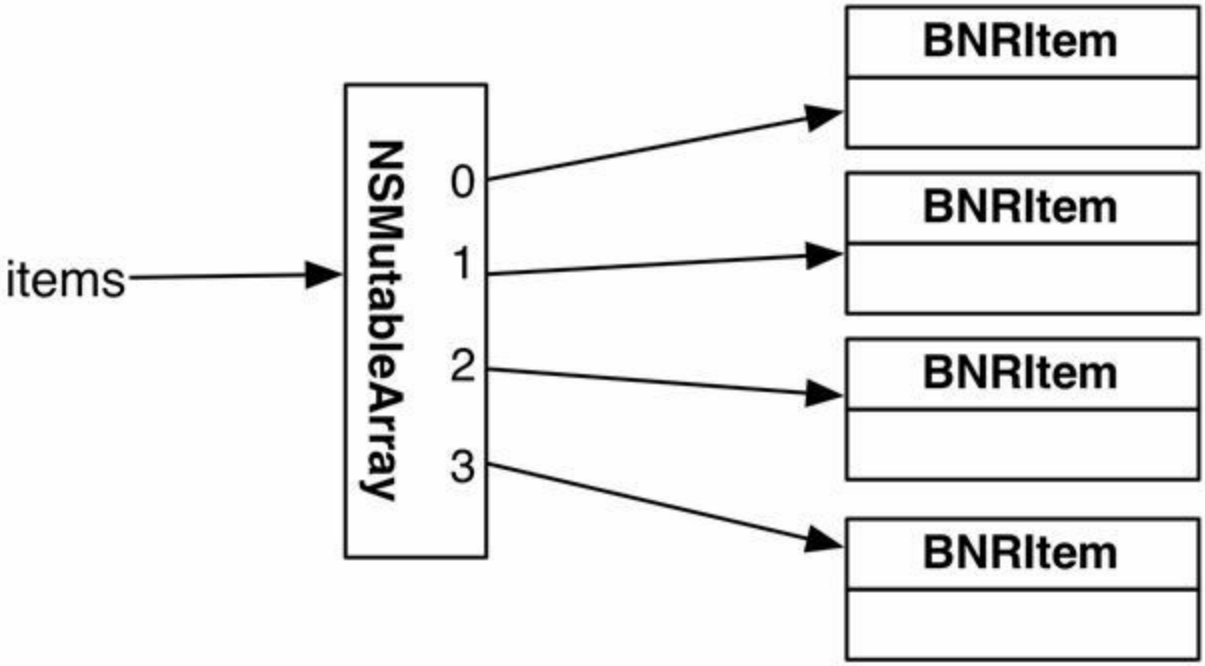


图2-10　使用BNRItem代替NSString

在本书后续章节中，读者会在开发一个复杂的iOS应用时重用BNRItem。

创建NSObject子类

用Xcode创建新类的方法为：选择菜单File→New→File…，出现新的面板后，选择面板左侧OS X部分下的Cocoa，然后选择面板右侧的Objective-C class并单击Next按钮（见图2-11）。

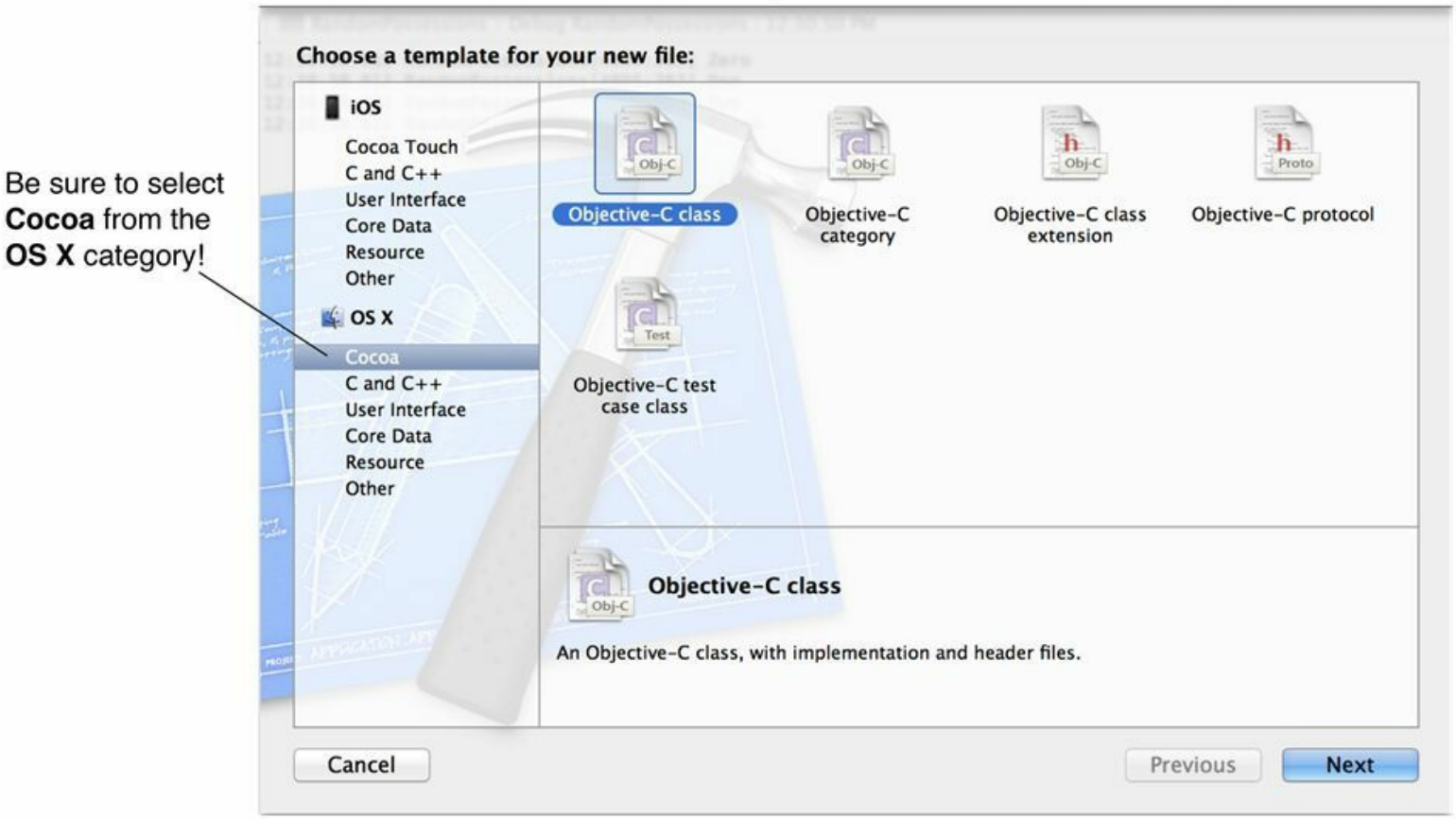


图2-11 创建新类

在新出现的面板中，将新类命名为BNRItem(标题为Class的文本框)，并将其父类设置为NSObject(标题为Subclass of的文本框)，然后单击Next按钮(见图2-12)。

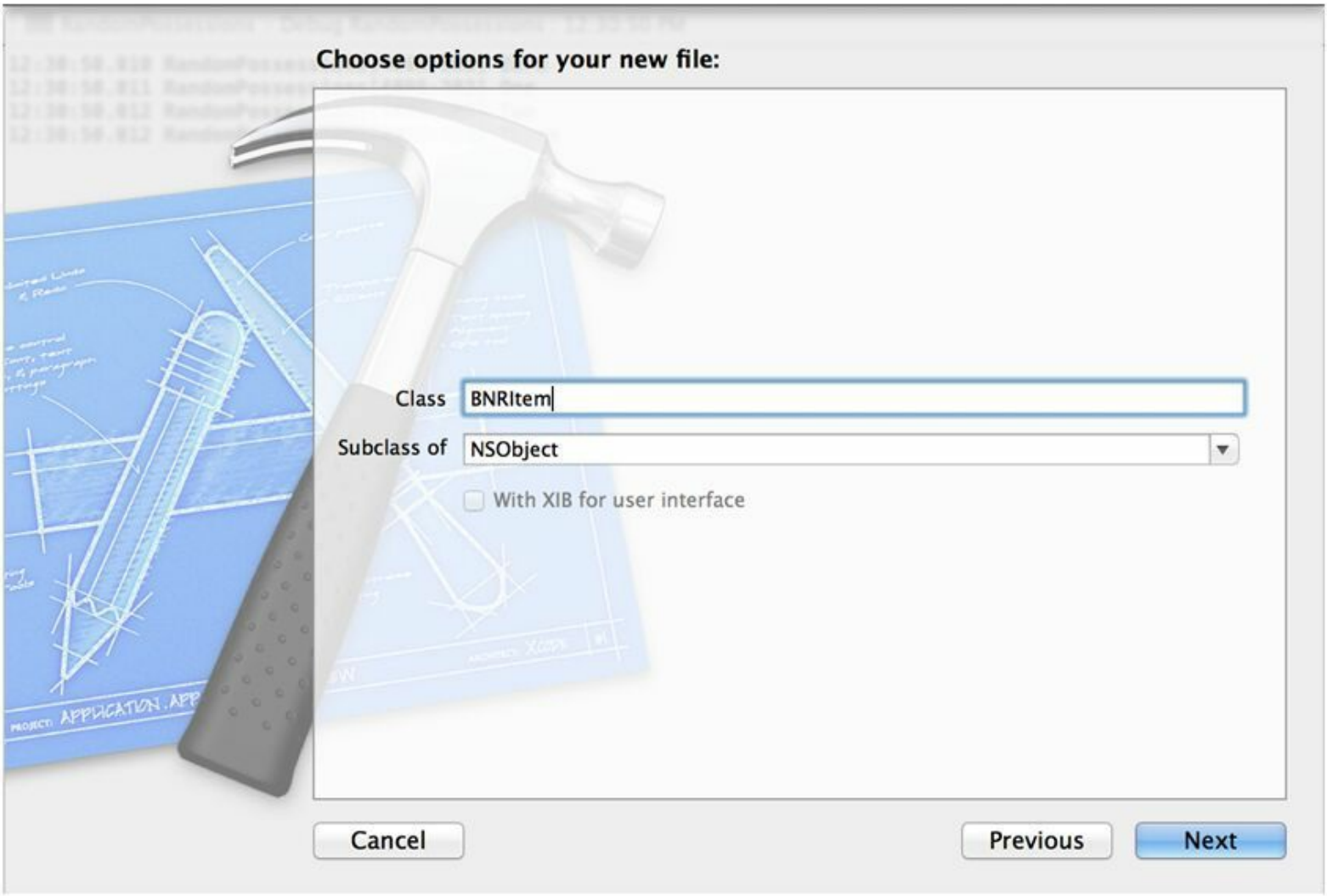


图2-12 选择父类

Xcode会显示一个下拉窗口，提示读者为类文件选择保存位置，全部使用默认设置就可以了。确保选中了Targets中RandomItems前的选择框。单击Create按钮。

在项目导航面板中，找到BNRItem类文件——BNRItem.h和BNRItem.m:

•BNRItem.h是头文件(header file)，也称为接口文件(interface file)，负责声明类的类名、类的父类、每个类的对象都会拥有的实例变量及该类实现的全部方法。

•BNRItem.m是实现文件(implementation file)，包含BNRItem类所实现的方法的全部代码。

每个Objective-C类都有两个这样的文件。读者可以将头文件看成某个类的“用户手册”，将实现文件看成“工程细节”，后者决定类实际会怎样工作。

在项目导航面板中选择BNRItem.h，Xcode会在编辑器区域显示该文件。BNRItem.h目前的代码如下:

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

@interface BNRItem : NSObject
```

@end

要在Objective-C中声明类，需要使用@interface指令，后跟类名，接着为冒号，冒号后面为父类的类名。Objective-C只允许单继承，所有的类都只能有一个父类：

@interface ClassName : SuperclassName

以上这段代码中的@end指令代表BNRItem类的声明至此结束。

请注意前缀@。Objective-C保留了C语言的关键字，并增加了若干Objective-C特有的关键字，新增加的关键字都用前缀@加以区分。

实例变量

真实世界中的物品会有名称、序列号、价值和创建日期。可以将它们设置为BNRItem的实例变量。

声明类的实例变量时，需要将相应的声明写在花括号里，并紧跟在类声明的后面。在BNRItem.h中，为BNRItem类添加一对花括号和四个实例变量：

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

@interface BNRItem : NSObject

{

NSString *_itemName;

NSString *_serialNumber;

int _valueInDollars;

NSDate *_dateCreated;

}

@end
```

这样，每个BNRItem对象都会有四个空位（spot），其中一个用于存放int整数，另外三个用于存放指向对象的指针，分别指向两个NSString对象和一个NSDate对象（请读者记住，*代表相应的变量是指针）。图2-13是一个BNRItem对象的例子，其实例变量都已经赋值。

图2-13一共显示了四个对象：一个BNRItem对象、两个NSString对象和一个NSDate对象。这里的每个对象都是独立的，和其他对象没有关联。BNRItem对象的三个实例变量是指向三个对象的指针，并没有直接保存这些对象。

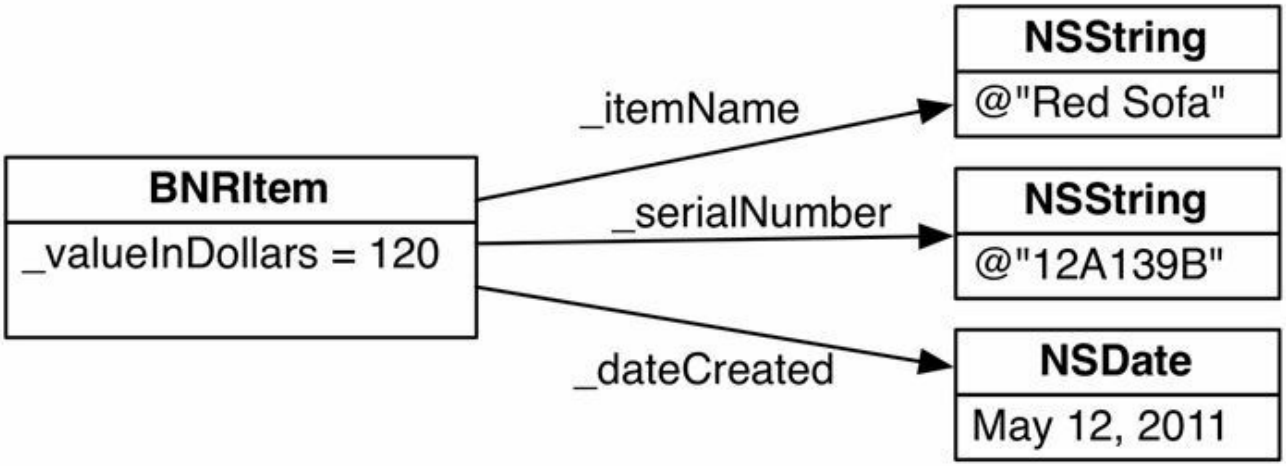


图2-13　一个BNRItem对象

例如，每个BNRItem对象都有一个名为_itemName的实例变量（指针类型）。图2-13中的BNRItem对象，其_itemName指向一个NSString对象，该对象的内容是Red Sofa（红色沙发）。但是这个BNRItem对象并不是保存Red Sofa字符串，而是将该字符串在内存中的地址赋给_itemName。读者可以将这种关系视为BNRItem对象将Red Sofa字符串命名为_itemName。

实例变量_valueInDollars的情况则不同。它不是指向其他对象的指针，而只是一个int类型的变量。对象会直接保存非指针类型的实例变量。指针的概念不容易理解，第3章会详细介绍对象、指针和实例变量。此外，还会使用对象图（见图2-13）来阐明对象和“指向对象的指针”之间的差别。

存取实例变量

为BNRItem对象添加实例变量后，还要能够存取这些变量的方法。在面向对象的编程语言中，这类存取实例变量的方法称为存取方法（accessor method），即存方法和取方法。如果没有存取方法，就无法访问对象的实例变量。

在BNRItem.h中，为BNRItem对象的实例变量声明存取方法。实例变量_valueInDollars、_itemName和_serialNumber需要存方法和取方法，而实例变量_dateCreated是只读的（read-only），不能修改，因此只需要取方法。

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

@interface BNRItem : NSObject

{

NSString *_itemName;

NSString *_serialNumber;
```

```
    int _valueInDollars;
    NSDate *_dateCreated;
    }
    - (void)setItemName:(NSString *)str;
    - (NSString *)itemName;
    - (void)setSerialNumber:(NSString *)str;
    - (NSString *)serialNumber;
    - (void)setValueInDollars:(int)v;
    - (int)valueInDollars;
    - (NSDate *)dateCreated;
    @end
```

在Objective-C中，存方法的命名规则为英文单词set加上要修改的实例变量的变量名（首字母大写）。以itemName为例，其存方法的方法名是setItemName:。在其他语言中，取方法的方法名通常会是getItemName。但在Objective-C中，取方法的方法名就是实例变量的变量名。Cocoa Touch库中的部分代码会假定读者所编写的类也遵守这样的约定。因此，有着良好代码风格的Cocoa Touch程序员都会遵守这个约定。

（有Objective-C经验的读者请注意，第3章会介绍属性（@property）。）

接下来打开BNRItem的实现文件，BNRItem.m。

任何一个类的实现文件，都必须在其顶部导入自己的头文件。类的实现需要知道相应的类是如何声明的。导入（#import）和C语言中的包含（#include）作用相同，差别是#import可以确保不会重复导入同一个文件。

位于导入语句下面的是实现程序段（implementation block）。实现程序段从@implementation指令开始，后跟要实现的类的类名。实现文件中的所有方法定义都要写在实现程序段里。实现程序段以@end指令结束。

在BNRItem.m中，删除@implementation和@end之间由项目模板加入的代码，然后为BNRItem.h中声明的实例变量实现存取方法。

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

```
- (void)setItemName:(NSString *)str
{
_itemName = str;
}
- (NSString *)itemName
{
return _itemName;
}
- (void)setSerialNumber:(NSString *)str
{
_serialNumber = str;
}
- (NSString *)serialNumber
{
return _serialNumber;
}
- (void)setValueInDollars:(int)v
{
_valueInDollars = v;
}
- (int)valueInDollars
{
return _valueInDollars;
}
```

```
- (NSDate *)dateCreated
{
return _dateCreated;
}
@end
```

在以上这段代码中，存方法将传入的参数赋给了实例变量；取方法则返回实例变量的值。

现在，如果Xcode提示有错误，请读者检查代码并修复（大小写错误或者漏掉分号等）。

下面开始测试新创建的类和存取方法。首先在main.m中导入BNRItem的头文件。

```
#import <Foundation/Foundation.h>
#import "BNRItem.h"
int main (int argc, const char * argv[])
{
...
```

读者可能会问为什么导入BNRItem.h而不导入NSMutableArray.h？这是因为NSMutableArray包含在Foundation框架中，只要导入Foundation/Foundation.h，就会同时导入NSMutableArray。BNRItem则位于仅包含自己一个类的类文件中，必须在main.m中明确导入。否则，编译器无法确定BNRItem类是否存在并会提示错误。

接下来创建一个新的BNRItem对象，然后将该对象的实例变量输出至控制台，代码如下：

```
int main (int argc, const char * argv[])
{
@autoreleasepool {
    NSMutableArray *items = [[NSMutableArray alloc] init];
    [items addObject:@"One"];
    [items addObject:@"Two"];
    [items addObject:@"Three"];
    [items insertObject:@"Zero" atIndex:0];
```

```
        // 遍历items数组中的每一个item
        for (NSString *item in items) {
            // 打印对象信息
            NSLog(@"%@", item);
        }
        BNRItem *item = [[BNRItem alloc] init];
        NSLog(@"%@ %@ %@ %d", [item itemName], [item dateCreated],
                        [item serialNumber], [item valueInDollars]);
        items = nil;
    }
    return 0;
}
```

构建并运行应用。在输出的末端，可以发现一行包含三个(null)和一个0的字符串。粗体代码首先创建了一个新的BNRItem对象，而输出结果就是该对象的实例变量的值(见图2-14)。

```
2013-12-08 18:43:11.237 RandomItems[4239:303] Zero
2013-12-08 18:43:11.239 RandomItems[4239:303] One
2013-12-08 18:43:11.239 RandomItems[4239:303] Two
2013-12-08 18:43:11.240 RandomItems[4239:303] Three
2013-12-08 18:43:11.240 RandomItems[4239:303] (null) (null) (null) 0
Program ended with exit code: 0
```

图2-14　实例变量的值

当某个对象被创建出来后，其所有的实例变量都会被设为默认值：如果实例变量是指向对象的指针，那么相应的指针会指向nil。如果实例变量是int这样的基本类型，那么其数值会是0。

要为新创建的BNRItem对象设置更有意义的数据，需要创建一些新对象，然后将这些对象作为实参传给该对象的存方法。

在main.m中加入以下代码：

```
// 请读者注意，这里省略了部分相邻代码
```

```
...
BNRItem *item = [[BNRItem alloc] init];
// 创建一个新的NSString对象"Red Sofa"，并传给BNRItem对象
[item setItemName:@"Red Sofa"];
// 创建一个新的NSString对象"A1B2C"，并传给BNRItem对象
[item setSerialNumber:@"A1B2C"];
// 将数值100传给BNRItem对象，赋给valueInDollars
[item setValueInDollars:100];
NSLog(@"%@ %@ %@ %d", [item itemName], [item dateCreated],
[item serialNumber], [item valueInDollars]);
...
```

构建并运行应用，能在控制台看到所有的实例变量输出（见图2-15），但是_dateCreated的值仍然是（null）。后面的章节中会学习如何在创建BNRItem对象时给_dateCreated赋值。

```
2013-12-08 18:44:56.551 RandomItems[4254:303] Zero
2013-12-08 18:44:56.553 RandomItems[4254:303] One
2013-12-08 18:44:56.553 RandomItems[4254:303] Two
2013-12-08 18:44:56.554 RandomItems[4254:303] Three
2013-12-08 18:44:56.554 RandomItems[4254:303] Red Sofa (null) A1B2C 100
Program ended with exit code: 0
```

图2-15　给实例变量赋值

使用点语法

之前的代码是通过发送消息来存取实例变量的：

```
BNRItem *item = [[BNRItem alloc] init];
// 发送消息为_valueInDollars实例变量赋值
[item setValueInDollars:5];
```

```
// 发送消息获取_valueInDollars的值

int value = [item valueInDollars];
```

另一种方法是使用点语法(dot syntax)，也叫做点符号(dot notation)。以下代码使用点语法存取实例变量：

```
BNRItem *item = [[BNRItem alloc] init];

// 使用点语法为_valueInDollars实例变量赋值

item.valueInDollars = 5;

// 使用点语法获取_valueInDollars的值

int value = item.valueInDollars;
```

语法格式为：消息接受者(item)后面加上一个“。”，再加上实例变量的名字(去掉变量名之前的下画线，如_valueInDollars改为valueInDollars)。

请注意，点语法在存和取方法中的用法相同(item.valueInDollars)。区别是：如果点语法用在赋值号左边，就表示存方法，用在右边则代表取方法。

点语法和存取方法在应用运行时没有区别。两种语法编译后的代码也一样，无论点是语法还是存取方法都会调用之前实现的valueInDollars和setValueInDollars:方法。

相对于调用存取方法，越来越多的Objective-C程序员更倾向于使用点语法，点语法的可读性更好，特别是在有多层嵌套消息的情况下。Apple的官方代码坚持使用点语法存取实例变量，因此本书也会这样做。

在main.m中修改代码，使用点语法存取实例变量：

```
...

BNRItem *item = [[BNRItem alloc] init];

// 创建一个新的NSString对象"Red Sofa"，并传给BNRItem对象

~~[item setItemName:@"Red Sofa"];~~

item.itemName = @"Red Sofa";

// 创建一个新的NSString对象"A1B2C"，并传给BNRItem对象

~~[item setSerialNumber:@"A1B2C"];~~

item.serialNumber = @"A1B2C";
```

```
// 将数值100传给BNRItem对象，赋给valueInDollars
[item setValueInDollars:100];
item.valueInDollars = 100;
NSLog(@"%@ %@ %@ %d", [item itemName], [item dateCreated],
[item serialNumber], [item valueInDollars]);
NSLog(@"%@ %@ %@ %d", item.itemName, item.dateCreated,
item.serialNumber, item.valueInDollars);
...
```

类方法和实例方法

Objective-C中的方法分为实例方法和类方法两种。类方法（class method）的作用通常是创建对象，或者获取类的某些全局属性。类方法不会作用在对象上，也不能存取实例变量。实例方法（instance method）则用来操作类的对象（对象有时也称为类的一个实例），例如，存取方法都是实例方法，用来设置和获取对象的实例变量。

调用实例方法时，需要向类的对象发送消息，而调用类方法时，则向类自身发送消息。

例如，在创建一个BNRItem对象时，首先向BNRItem类发送alloc（类方法）消息，然后向使用alloc方法创建的对象发送init（实例方法）消息。

前文介绍过的description即是一个实例方法。在下一节中，读者将为BNRItem实现description方法，返回一个描述BNRItem对象的字符串。在后面的章节中还将实现一个类方法，用来创建有随机数据的BNRItem对象。

覆盖方法

子类可以覆盖（override）父类的方法。以description为例，向某个NSObject对象发送description消息时，可以得到一个NSString对象。这个NSString对象会包含当前对象的类名和其在内存中的地址信息，例如：

```
<BNRQuizViewController:0x4b222a0>
```

任何一个NSObject的子类都可以覆盖description方法，使返回的字符串能更好地描述子类的对象。例如，NSString覆盖description，以返回NSString对象自身。NSArray覆盖description，以

返回数组对象所包含的所有对象的描述字符串。

因为BNRItem是NSObject的子类(NSObject是最初声明description方法的类)，所以在BNRItem类中重新实现description方法，就是在覆盖NSObject的description方法。

在BNRItem.m中要覆盖description方法，可以将新的代码写在@implementation和@end之间的任意位置(除了现有方法的花括号内)中实现，代码如下：

```
- (NSString *)description

{

NSString *descriptionString =

[[NSString alloc] initWithFormat:@"%@ (%@): Worth $%d, recorded on %@",

                self.itemName,

                self.serialNumber,

                self.valueInDollars,

                self.dateCreated];

return descriptionString;

}
```

请注意代码中没有直接传入实例变量的名称(例如_itemName)，而是调用了存取方法(使用点语法)。使用存取方法访问实例变量是良好的编程习惯，即使是访问对象自身的实例变量，也应该使用存取方法。访问实例变量时，如果使用了存取方法，系统可以在操作实例变量时附加一些额外操作，后面的章节中会介绍到。

覆盖description方法后，向某个BNRItem对象发送description消息就会得到一个NSString对象，相对于内存地址，该字符串可以更好地描述BNRItem对象。

在main.m中删除之前通过NSLog输出BNRItem对象的实例变量的那行代码，改用覆盖后的description方法，代码如下：

```
...

item.valueInDollars = 100;

~~NSLog(@"%@ %@ %@ %d", item.itemName, item.dateCreated,~~

~~item.serialNumber, item.valueInDollars);~~

// 程序会先调用相应实参的description方法，
```

// 然后用返回的字符串替换%@

NSLog(@"%@", item);

items = nil;

构建并运行应用，在控制台中查看输出结果（见图2-16）。

```
2013-12-08 18:44:56.551 RandomItems[4254:303] Zero
2013-12-08 18:44:56.553 RandomItems[4254:303] One
2013-12-08 18:44:56.553 RandomItems[4254:303] Two
2013-12-08 18:44:56.554 RandomItems[4254:303] Three
2013-12-08 18:44:56.554 RandomItems[4254:303] Red Sofa (null) A1B2C 100
Program ended with exit code: 0
```

图2-16　打印BNRItem对象的描述字符串

如果不是要覆盖父类的方法，而是要创建全新的实例方法，又该怎样做？要创建新的实例方法，需要先在相应的头文件中声明新的方法，然后在对应的实现文件中定义该方法。下面开始创建两个新的实例方法，用于初始化BNRItem对象。

初始化方法

BNRItem类目前还只能使用从NSObject类继承而来的init方法初始化对象。本节将创建两个新的实例方法用于初始化BNRItem对象，这种用于初始化类的对象的方法称为初始化方法（initialization method，或initializer）。

在BNRItem.h中声明两个初始化方法，代码如下：

NSDate *_dateCreated;

}

- (instancetype)initWithItemName:(NSString *)name

valueInDollars:(int)value

　　serialNumber:(NSString *)sNumber;

- (instancetype)initWithItemName:(NSString *)name;

- (void)setItemName:(NSString *)str;

（稍后会学习instancetype。）

每个初始化方法的方法名都会以英文单词init开头。初始化方法的这种命名模式只是一种约定，不会使其有别于其他实例方法。但是，Objective-C中的命名约定很重要，应该严格遵守（这点特别重要，忽视Objective-C的命名约定会产生问题，其严重程度将远超大部分初学者的预期）。

初始化方法类似init，但是会带参数，用于初始化当前的对象。为了应对各种不同的初始化需要，很多类会提供一种以上的初始化方法。例如，第一个初始化方法有三个参数，分别用来设置BNRItem对象的名称、价值和序列号——必须知道这些信息才能使用该初始化方法。如果只知道BNRItem对象的名称，就使用第二个初始化方法。

指定初始化方法

任何一个类，无论有多少个初始化方法，都必须选定其中的一个作为指定初始化（designated initializer）方法。指定初始化方法要确保对象的每一个实例变量都处在一个有效的状态。有效（valid）一词有很多不同的意思，这里是指向初始化后的对象发送消息时，输出结果是可预期的，并且不会有“坏事”发生。

指定初始化方法的参数通常会和最重要的、最常用的实例变量相对应。以BNRItem类为例，它有四个实例变量，但是只有其中三个是可写的，因此BNRItem类的指定初始化方法应该有三个实参，并为_dateCreated赋值。打开BNRItem.h，在指定初始化方法前添加注释：

```
    NSDate *_dateCreated;

    }

    // BNRItem类的指定初始化方法

    - (instancetype)initWithItemName:(NSString *)name

                        valueInDollars:(int)value

    serialNumber:(NSString *)sNumber;

    - (instancetype)initWithItemName:(NSString *)name;

    - (void)setItemName:(NSString *)str;
```

instancetype

两个初始化方法的返回类型都是instancetype。该关键字表示方法的返回类型和调用方法的

对象类型相同。init方法的返回类型都声明为instancetype。

为什么不将返回类型声明为BNRItem *？问题在于，BNRItem的子类会继承其全部方法，其中包括初始化方法和其返回类型。如果BNRItem的子类对象收到该初始化消息，那么返回的会是什么类型的对象？答案是相应子类的对象，而不是BNRItem对象。读者可能会想："这个问题容易解决，在子类中覆盖初始化方法并修改返回类型即可"。但是，在Objective-C中，一个对象不能同时拥有两个选择器相同、但是返回类型（或者参数类型）不同的方法。

为了避免这个问题，可以声明init方法的返回类型和调用方法的对象类型相同，这样就保证了对象初始化后仍然是正确的类型。

id

在Objective-C引入instancetype关键字之前，初始化方法的返回类型都是id（读音"eye-dee"）。id的定义是"指向任意对象的指针"（id和C语言的void* 类似）。使用Xcode创建类文件时，模板代码中仍然使用id作为初始化方法的返回类型，我们希望Apple能尽快更新模板代码。

instancetype只能用来表示方法返回类型，但是id还可以用来表示变量和方法参数的类型。如果程序运行时无法确定一个对象的类型，就可以将该对象声明为id。

```
id objectOfUnknownType;
```

可以使用id快速遍历存储不同类型对象的数组：

```
for (id item in items) {
NSLog(@"%@", item);
}
```

请注意，因为id的定义是"指向任意对象的指针"，所以不能在变量名或参数名前再加"*"。

实现BNRItem类的指定初始化方法

在BNRItem.m中为BNRItem类实现指定初始化方法，代码如下：

```
@implementation BNRItem
- (instancetype)initWithItemName:(NSString *)name
        valueInDollars:(int)value
            serialNumber:(NSString *)sNumber
```

```
{
// 调用父类的指定初始化方法
self = [super init];
// 父类的指定初始化方法是否成功创建了父类对象？
if (self) {
    // 为实例变量设定初始值
    _itemName = name;
    _serialNumber = sNumber;
    _valueInDollars = value;
    // 设置_dateCreated的值为系统当前时间
    _dateCreated = [[NSDate alloc] init];
}
// 返回初始化后的对象的新地址
return self;
}
```

在以上代码中，请注意实例变量_dateCreated的值被设置为指向NSDate对象的指针，它表示系统当前时间。

接下来请看方法实现中的第一行代码。当编写指定初始化方法时，首先要做的事情是通过super关键字，调用父类的指定初始化方法。最后要做的事情是通过self关键字，返回一个指针，指向成功初始化后的对象。因此，要理解初始化方法就要先了解self和super。

self

self存在于方法中，是一个隐式(implicit)局部变量。编写方法时不需要声明self，并且程序会自动为self赋值，指向收到消息的对象自身(大多数面向对象的语言也有这个概念，有些将其称为this，而不是self)。通常情况下，self会用来向对象自己发送消息，代码如下：

```
- (void)chickenDance
```

```
{
[self pretendHandsAreBeaks];
[self flapWings];
[self shakeTailFeathers];
}
```

初始化方法的最后一行代码必须返回初始化后的对象。这样，调用者才能将该对象赋给指针变量。

```
return self;
```

super

在覆盖父类的某个方法时，往往需要保留该方法在父类中的实现，然后在其基础上扩充子类的实现。为了能更方便地完成这项任务，Objective-C提供了一个名为super的关键字：

```
- (void)someMethod
{
[super someMethod];
[self doMoreStuff];
}
```

super是如何工作的？通常情况下，当某个对象收到消息时，系统会先从这个对象的类开始，查询和消息名相同的方法名。如果没找到，则会在这个对象的父类中继续查找。该查询过程会沿着继承路径向上，直到找到相应的方法名为止（如果直到层次结构的顶端也没能找到合适的方法，程序就会抛出异常）。

向super发消息，其实是向self发消息，但是要求系统在查找方法时跳过当前对象的类，从父类开始查询。以BNRItem的指定初始化方法为例，向super发送init消息会调用NSObject的init。

确认父类的初始化结果

在调用父类的指定初始化方法之后，代码检查父类的初始化结果。如果初始化方法不能正确完成对象的初始化，就会返回nil。因此，在调用父类的初始化方法后，应该先将得到的返回值赋给self变量，然后确认该变量是不是nil，如果不是nil，再进行下一步的初始化工作。

初始化方法中的实例变量

现在请注意初始化方法的核心代码：初始化实例变量。之前本书提到，不要直接访问实例变量，而是使用存取方法。但是在初始化方法中相反，应该直接访问实例变量。

初始化方法在执行时，无法确定新创建对象的实例变量是否已经处于正确设置。实例变量可能具有不正确的值，也可能没有被正确分配，如果这时调用存取方法访问实例变量，则很可能会引起错误。本书在初始化方法中直接访问实例变量，不调用存取方法。

一些非常优秀的Objective-C程序员坚持在初始化方法中也使用存取方法。他们认为，如果存取方法中包含对实例变量的不安全操作，那么应该将这些复杂的代码提取出来，放到其他方法中，存取方法应该保持简单，只用来设置和获取实例变量。两种说法都有道理，但是本书选择在初始化方法中直接访问实例变量。

其他初始化方法与初始化方法链

接下来实现BNRItem的第二个初始化方法。实现initWithItemName:方法时，不需要将指定初始化方法中的代码搬过来再重写一遍。它只需要调用指定初始化方法，将得到的实参作为_itemName传入，而其他实参则使用某个默认值传入。

在BNRItem.m中实现initWithItemName:方法：

```
- (instancetype)initWithItemName:(NSString *)name

{

return [self initWithItemName:name

                    valueInDollars:0

serialNumber:@""];

}
```

BNRItem还有第三个初始化方法——init，继承自父类NSObject。如果向某个BNRItem对象发送init消息，程序就不会调用之前创建的指定初始化方法。因此必须覆盖BNRItem类的init方法，将其和指定初始化方法“串联”起来。

覆盖BNRItem.m中的init方法，调用initWithItemName:方法，并使用默认值设置BNRItem对象的名称，代码如下：

```
- (instancetype)init
```

```
{
return [self initWithItemName:@"Item"];
}
```

现在，如果给BNRItem对象发送init消息，则会调用initWithItemName:方法，传入默认名称。而initWithItemName:方法又会调用initWithItemName:valueInDollars: serialNumber:方法并传入默认值和序列号。

图2-17列出了BNRItem类的多个初始化方法之间的关系。其中白色为指定初始化方法，灰色为其他初始化方法。

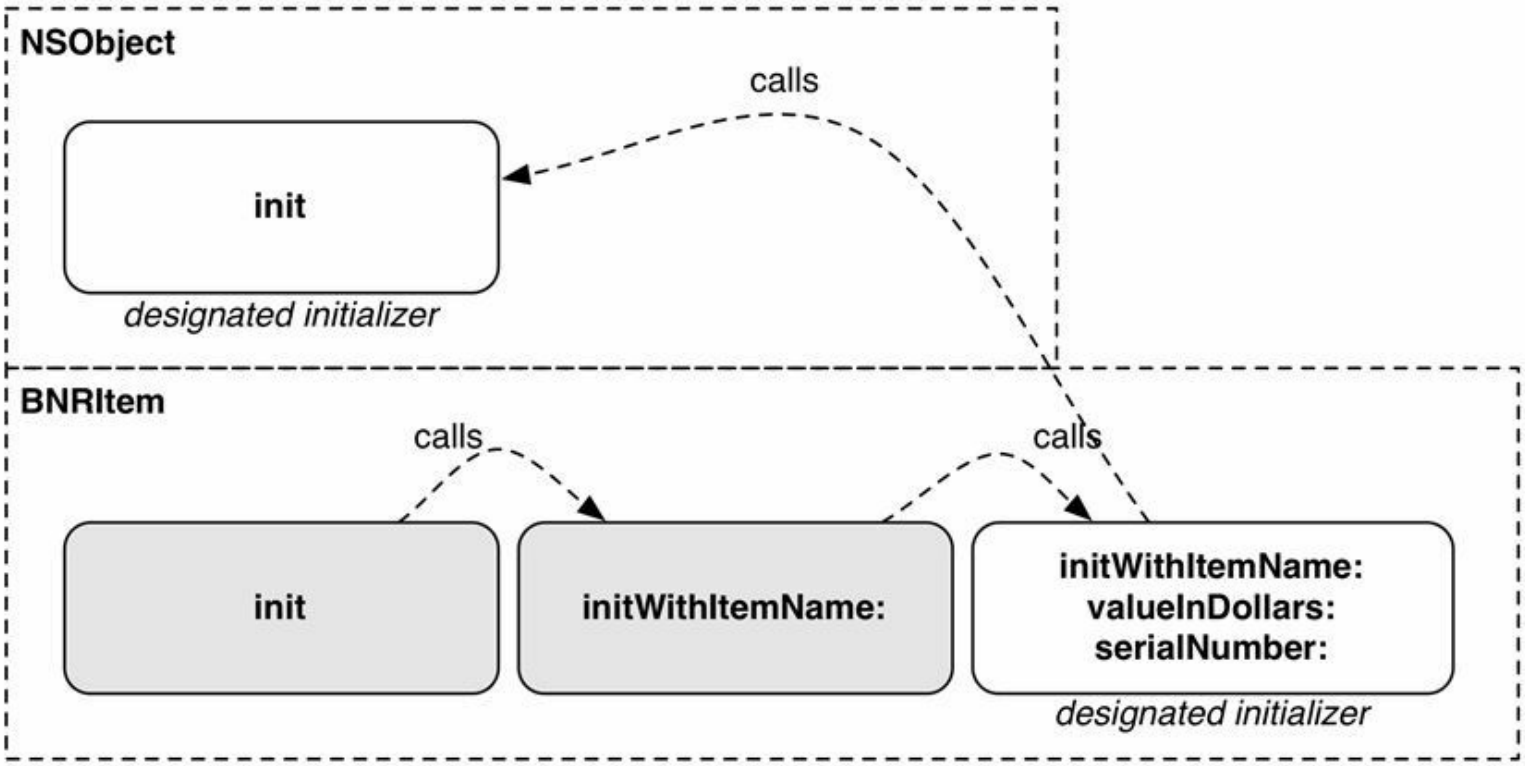


图2-17　初始化方法链

串联(chain)使用初始化方法的机制可以减少错误，也更容易维护代码。在创建类时，需要先确定指定初始化方法，然后只在指定初始化方法中编写初始化的核心代码，其他初始化方法只需要调用指定初始化方法(直接或间接)并传入默认值即可。

下面为初始化方法总结若干简单的规则。

•类会继承父类所有的初始化方法，也可以为类加入任意数量的初始化方法。

•每个类都要选定一个指定初始化方法。

•在执行其他初始化工作之前，必须先用指定初始化方法调用父类的指定初始化方法(直接或间接)。

•其他初始化方法要调用指定初始化方法(直接或间接)。

•如果某个类所声明的指定初始化方法与其父类的不同,就必须覆盖父类的指定初始化方法并调用新的指定初始化方法(直接或间接)。

使用初始化方法

现在可以使用BNRItem的指定初始化方法设置实例变量。

在main.m中找到创建并初始化BNRItem对象,并为其实例变量赋值的那几行代码。删除这几行代码,替换成如下所示的一行代码(创建BNRItem对象,然后调用指定初始化方法设置实例变量)。

```
...
// 遍历items数组中的每一个item
for (NSString *item in items) {
// 打印对象信息
NSLog(@"%@", item);
}
~~BNRItem *item = [[BNRItem alloc] init];~~
~~item.itemName = @"Red Sofa";~~
~~item.serialNumber = @"A1B2C";~~
~~item.valueInDollars = 100;~~
BNRItem *item = [[BNRItem alloc] initWithItemName:@"Red Sofa"
    valueInDollars:100
        serialNumber:@"A1B2C"];
NSLog(@"%@", item);
...
```

构建并运行应用,控制台会输出BNRItem对象的描述信息,且相应的数据应该和传入指定初始化方法的实参一致。

接下来验证另外两个初始化方法是否能正常工作。在main.m中使用initWithItemName:和init各创建一个BNRItem对象。

```
...
BNRItem *item = [[BNRItem alloc] initWithItemName:@"Red Sofa"
    valueInDollars:100
serialNumber:@"A1B2C"];
NSLog(@"%@", item);
BNRItem *itemWithName = [[BNRItem alloc]initWithItemName:@"Blue Sofa"];
NSLog(@"%@", itemWithName);
BNRItem *itemWithNoName = [[BNRItem alloc] init];
NSLog(@"%@", itemWithNoName);
items = nil;
}
return 0;
}
```

构建并运行应用，在控制台中检查BNRItem的初始化方法链是否正常工作（见图2-18）。

```
2013-12-08 18:55:18.620 RandomItems[4302:303] Zero
2013-12-08 18:55:18.622 RandomItems[4302:303] One
2013-12-08 18:55:18.623 RandomItems[4302:303] Two
2013-12-08 18:55:18.623 RandomItems[4302:303] Three
2013-12-08 18:55:18.628 RandomItems[4302:303] Red Sofa (A1B2C): Worth $100, recorded on 2013-12-08 23:55:18 +0000
2013-12-08 18:55:18.629 RandomItems[4302:303] Blue Sofa (): Worth $0, recorded on 2013-12-08 23:55:18 +0000
2013-12-08 18:55:18.629 RandomItems[4302:303] Item (): Worth $0, recorded on 2013-12-08 23:55:18 +0000
Program ended with exit code: 0
```

图2-18　三个初始化方法的输出结果

还剩最后一项工作——编写一个类方法创建有随机数据的BNRItem对象。

类方法

从语法上看，类方法的声明和实例方法的声明不同，差别在于第一个字符。在返回类型的

前面，实例方法使用的是字符-，而类方法使用的是字符+。

在BNRItem.h中声明一个类方法，用来创建有随机数据的BNRItem对象，代码如下：

```
@interface BNRItem : NSObject

{

NSString *_itemName;

NSString *_serialNumber;

int _valueInDollars;

NSDate *_dateCreated;

}

+ (instancetype)randomItem;

- (instancetype)initWithItemName:(NSString *)name

  valueInDollars:(int)value

serialNumber:(NSString *)sNumber;
```

注意头文件中的声明顺序。实例变量声明应该写在最前面，然后是类方法，接下来是初始化方法，最后是其他方法。这种排列顺序是一种约定，可以使头文件更容易阅读。

在BNRItem.m中实现randomItem方法，创建、配置并返回一个BNRItem对象（再次提醒读者，新加入的代码要写在@implementation和@end之间），代码如下：

```
+ (instancetype)randomItem

{

// 创建不可变数组对象，包含三个形容词

NSArray *randomAdjectiveList = @[@"Fluffy", @"Rusty", @"Shiny"];

// 创建不可变数组对象，包含三个名词

NSArray *randomNounList = @[@"Bear", @"Spork", @"Mac"];

// 根据数组对象所含对象的个数，得到随机索引

// 注意：运算符%是模运算符，运算后得到的是余数
```

```
// 因此adjectiveIndex是一个0到2(包括2)的随机数
NSInteger adjectiveIndex = arc4random() % [randomAdjectiveList count];
NSInteger nounIndex = arc4random() % [randomNounList count];
// 注意，类型为NSInteger的变量不是对象，
// NSInteger是一种针对unsigned long(无符号长整数)的类型定义
NSString *randomName = [NSString stringWithFormat:@"%@ %@",
    [randomAdjectiveList objectAtIndex:adjectiveIndex],
    [randomNounList objectAtIndex:nounIndex]];
int randomValue = arc4random() % 100;
NSString *randomSerialNumber = [NSString stringWithFormat:@"%c%c%c%c%c",
            '0' + arc4random() % 10,
            'A' + arc4random() % 26,
            '0' + arc4random() % 10,
            'A' + arc4random() % 26,
            '0' + arc4random() % 10];
BNRItem *newItem = [[self alloc] initWithItemName:randomName
     valueInDollars:randomValue
serialNumber:randomSerialNumber];
return newItem;
}
```

首先，方法体的前两行代码创建了两个不可变数组，分别是randomAdjectiveList和randomNounList。请注意创建数组的语法——“@”符号后面加上一对方括号，数组中的对象写在方括号里，用逗号隔开。(以上两个数组中的对象全部是NSString对象。)这是一种创建NSArray对象的简洁语法。请注意，这种语法只能创建不可变数组，如果要使用可变数组，则不能使用这种语法。

创建形容词和名词数组之后，randomItem方法会根据一个随机的形容词和一个随机的名词创建出一个字符串。此外，还会创建出一个随机的整数和另一个根据随机数和随机字符创建

的字符串。

最后，randomItem方法会创建一个BNRItem对象，调用新对象的指定初始化方法并输入之前随机创建的对象和整数。

randomItem方法还使用NSString的类方法stringWithFormat:。调用stringWithFormat:方法时，要将相应的消息直接发送给NSString类。stringWithFormat:方法会根据传入的参数返回相应的NSString对象。在Objective-C中，如果某个类方法的返回类型是这个类的对象（例如stringWithFormat:和randomPossession），就可以将这种类方法称为便捷方法（convenience method）。

这里要注意randomItem是如何使用self的。因为它是类方法，所以self是指BNRItem类自身而不是某个对象。在便捷方法中，应该使用self，而不是直接使用类的类名。这样，子类也能使用父类的便捷方法，不至于发生错误。以BNRItem为例，如果创建一个BNRItem类的子类BNRToxicWasteItem，就可以向这个子类发送randomItem消息：

测试**BNRItem**类

现在开始编写本章最终版本的RandomItems。打开main.m，只保留创建和销毁items数组的代码，删除其他代码。然后向数组中添加10个有随机内容的BNRItem对象，最后输出至控制台（见图2-19）。

```
2013-10-29 18:17:42.880 RandomItems[64653:303] Rusty Spork (8Q2U8): Worth $73, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.880 RandomItems[64653:303] Shiny Spork (5Y2V3): Worth $40, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.881 RandomItems[64653:303] Rusty Spork (2F9Z7): Worth $40, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.881 RandomItems[64653:303] Rusty Bear (8G5V6): Worth $99, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.881 RandomItems[64653:303] Shiny Spork (3P9B1): Worth $10, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.882 RandomItems[64653:303] Rusty Mac (6R5C1): Worth $93, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.882 RandomItems[64653:303] Fluffy Spork (3E4O0): Worth $1, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.882 RandomItems[64653:303] Fluffy Mac (3A6T4): Worth $30, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.883 RandomItems[64653:303] Shiny Spork (8S3I1): Worth $77, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.883 RandomItems[64653:303] Rusty Spork (4F6F9): Worth $65, recorded on 2013-10-29 22:17:42 +0000
Program ended with exit code: 0
```

图2-19　有随机内容的BNRItem对象

```
int main (int argc, const char * argv[])

{

@autoreleasepool {

    NSMutableArray *items = [[NSMutableArray alloc] init];

    [items addObject:@"One"];

    [items addObject:@"Two"];
```

```
    [items addObject:@"Three"];
    [items insertObject:@"Zero" atIndex:0];
    // 遍历items数组中的每一个item
    for (NSString *item in items) {
        // 打印对象信息
        NSLog(@"%@", item);
    }
    BNRItem *item = [[BNRItem alloc] initWithItemName:@"Red Sofa"
        valueInDollars:100
        serialNumber:@"A1B2C"];
    NSLog(@"%@", item);
    BNRItem *itemWithName = [[BNRItem alloc] initWithItemName:@"Blue Sofa"];
    NSLog(@"%@", itemWithName);
    BNRItem *itemWithNoName = [[BNRItem alloc] init];
    NSLog(@"%@", itemWithNoName);
    for (int i = 0; i < 10; i++) {
        BNRItem *item = [BNRItem randomItem];
        [items addObject:item];
    }
    for (BNRItem *item in items) {
        NSLog(@"%@", item);
    }
    items = nil;
}
```

```
  return 0;

    }
```

请注意代码中第一条循环语句没有使用快速遍历语法，因为循环语句是在向数组中添加对象，而不是遍历。

构建并运行应用，检查控制台中的输出。

2.5 深入学习NSArray与NSMutableArray

开发iOS应用时经常要用到数组，现在开始深入学习数组的相关知识。

Objective-C中的数组可以存储不同类型的对象，虽然items数组目前只存储BNRItem对象，但是也可以存储NSDate对象或其他对象。这一点和很多强类型（strongly-typed）语言不同，这些语言的数组只能保存一种类型的对象。

数组对象只能保存指向Objective-C对象的指针，所以不能将基本类型（primitive）的变量或C结构加入数组对象。如果要将基本类型的变量和C结构加入数组，可以先将它们“包装”成Objective-C对象，例如NSNumber、NSValue和NSData。

注意，不能将nil加入数组对象。如果要将“空洞”加入数组对象，就必须使用NSNull对象。NSNull对象的作用就是代表nil，所以可以用来解决这类问题，代码如下：

```
[items addObject:[NSNull null]];
```

访问数组中的对象时，可以向数组对象发送objectAtIndex:消息，它会返回指定索引的对象，但是这种语法非常繁琐，还有一种更简洁的下标语法：

```
NSString *foo = items[0];
```

这行代码与发送objectAtIndex:消息的效果是相同的：

```
NSString *foo = [items objectAtIndex:0];
```

下面在BNRItem.m中使用下标语法重新实现randomItem方法。

```
+ (instancetype)randomItem
{
    ...
    ~~NSString *randomName = [NSString stringWithFormat:@"%@ %@",~~
        ~~[randomAdjectiveList objectAtIndex:adjectiveIndex],~~
        ~~[randomNounList objectAtIndex:nounIndex]];~~
    NSString *randomName = [NSString stringWithFormat:@"%@ %@",
        randomAdjectiveList[adjectiveIndex],
        randomNounList[nounIndex]];
    int randomValue = arc4random() % 100;
```

...

return newItem;

}

构建并运行应用，检查控制台中的输出结果是否和之前的结果相同。

方括号的嵌套层数越多，代码的可读性就越差。因为方括号的作用可能不同，容易产生混淆：有的是发送消息，有的是存取方法，有的是访问数组中的对象。坚持使用点语法和下标语法可以清晰地突出消息发送代码，也可以避免代码过于冗长。

与点语法和存取方法的关系相同，下标语法和objectAtIndex:消息编译后的结果也是一样的，编译器会自动将下标语法转换为objectAtIndex:消息。

在NSMutableArray中，可以使用下标语法向数组中添加和修改对象。

NSMutableArray *items = [[NSMutableArray alloc] init];

items[0] = @"A"; // Add @"A"

items[1] = @"B"; // Add @"B"

items[0] = @"C"; // Replace @"A" with @"C"

这几行代码等价于向items发送insertObject:atIndex:和replaceObject- AtIndex:withObject:消息。

2.6 异常与未知选择器

在运行时，当某个对象收到消息后，会根据创建该对象的类，执行和相应消息相匹配的方法。这种特性和多数编译语言不同，在这些编译语言中，需要执行的方法在编译时就决定了。

Objective-C对象都有一个名为isa的实例变量。对象可以通过自己的isa知道自身的类型。类在创建了一个对象后，会为新创建的对象的isa实例变量赋值，将其指回自己，即创建该对象的类（见图2-20）。这里之所以将这个实例变量命名为isa，是因为通过该实例变量可以知道某个对象“是”哪个类的实例（英文is a的意思“是一个”），虽然在开发应用时很少直接使用isa，但是Objective-C的很多特性都源自isa实例变量。

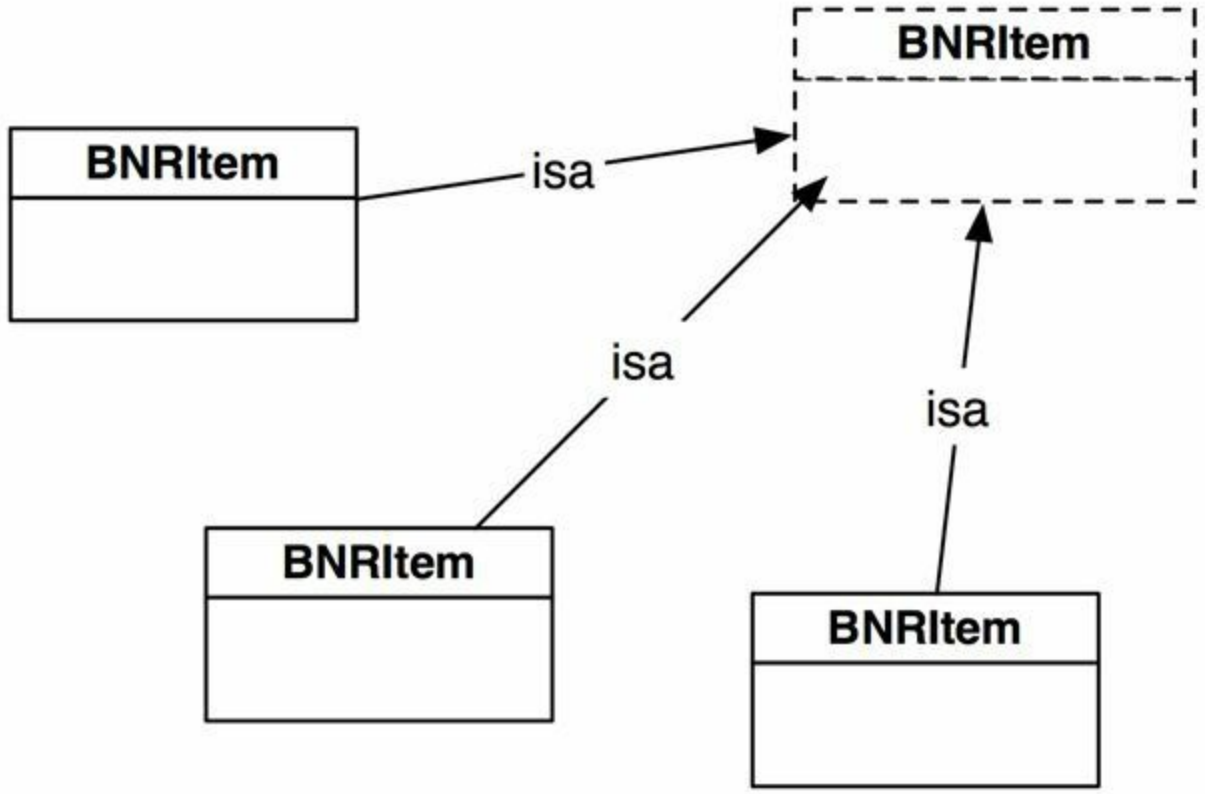


图2-20　isa实例变量

对象只能响应（respond）类（isa指向的类）中具有相应实现方法的消息。而对象的类型又只能在运行时才能确定，因此Xcode无法在编译时（即构建应用时）判断某个对象是否能响应特定的消息。如果Xcode判断应用会向某个对象发送其无法响应的消息，就会显示相应的警告信息，但是代码仍然能够编译通过。

出于某些原因（原因很多），如果应用最后还是向某个对象发送了其无法响应的消息，那么程序就会抛出异常（exception）。异常也称为运行时错误（run-time error），这是因为异常只会在应用运行时才会出现。和运行时错误相对应的是编译时错误，编译时错误只会在构建应用或编译代码时出现。

下面要在RandomItems中制造一处异常，让读者有机会进一步熟悉异常。

打开main.m，使用NSArray的lastObject方法取得数组中的最后一个BNRItem对象，然后发送一条该对象无法处理的消息：

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

#import "BNRItem.h"

int main (int argc, const char * argv[])

{

@autoreleasepool {

    NSMutableArray *items = [[NSMutableArray alloc] init];

    for (int i = 0; i < 10; i++) {

        BNRItem *item = [BNRItem randomItem];

        [items addObject:item];

    }

    id lastObj = [items lastObject];

    // lastObj是BNRItem对象，无法处理count消息

    [lastObj count];

    for (BNRItem *item in items) {

        NSLog(@"%@", item);

    }

    items = nil;

}

return 0;

 }
```

构建并运行应用，应用能编译通过并运行，但是会马上崩溃。查看控制台的输出，能找到如下的错误提示：

```
2014-01-19 12:23:47.990 RandomItems[10288:707] ***
```

Terminating app due to uncaught exception 'NSInvalidArgumentException', reason:

'-[BNRItem count]: unrecognized selector sent to instance 0x100117280'

这里显示的是异常的描述信息。如何解读这些信息？首先，描述信息的起始部分会列出日期、时间和应用的名称。这部分信息都可以忽略，重点是“***”之后的内容，这部分内容会显示程序发生了异常，并列出相应的原因。

在异常的描述信息中，产生异常的原因是最重要的部分。通过RandomItems发出的异常信息可以知道，这里产生异常的原因是程序向某个对象发送了一个未知选择器（unrecognized selector）。前文介绍过，选择器和消息有关。换句话说，RandomItems向某个对象发送了一条消息，但是收到消息的对象并没有实现相应的方法。

在这段异常的描述信息中，还可以找到消息的接收方和消息名，即某个BNRItem对象收到了count消息。这些信息能够帮助除错。位于行首的“-”代表接收方是对象，如果是“+”则代表接收方是类。

通过以上这个例子，读者应该可以了解两个开发要点：首先，当应用停止响应（halt）或崩溃时，应先检查控制台的输出。运行时发生的错误和编译时发现的错误一样重要。其次，要记住未知选择器的意思是某个对象收到了其没有实现的消息。未知选择器是经常会遇到的错误，牢记这个要点能帮助读者快速地诊断这类问题。

有些编程语言会通过try和catch程序段来处理异常。虽然Objective-C也支持这种特性，但是在开发应用时并不常用。通常情况下，异常源自程序员所犯的错误，所以应该在代码中修正，而不是在运行时处理。

在main.m中删除为了制造异常所加入的代码：

```
for (int i = 0; i < 10; i++) {

    BNRItem *item = [BNRItem randomItem];

    [items addObject:item];

    }

    id lastObj = [items lastObject];

    // lastObj是BNRItem对象，无法处理count消息

    [lastObj count];

    for (BNRItem *item in items) {

        NSLog(@"%@", item);

    }
```

2.7 练习

很多章节结尾处会提供至少一道习题，让读者能够有机会进一步巩固并验证在相关章节中学到的知识。建议读者尽可能多地完成这些练习，巩固之前学到的知识，并从“通过书本学习iOS开发”过渡到“独立进行iOS开发”。

这些练习分为以下三个难度等级。

•初级。初级练习会要求读者重复完成相关章节中已经完成过的任务，只是细节稍有不同。这些练习可以帮助读者巩固在相关章节中学到的知识，并让读者有机会在没有示例代码的情况下，自己输入类似的代码。多练习才能熟能生巧。

•中级。中级练习会要求读者做更多的调研工作，花更多的时间来思考如何完成任务。虽然读者要使用之前没有接触过的方法、类和属性，但是要完成的任务仍然和在相关章节中完成过的那些类似。

•高级。高级练习会难一点，可能需要读者花更多时间才能完成。这些练习要求读者在彻底理解相关章节所介绍的知识的基础上，独立思考并解决问题。完成这些练习能帮助读者做好准备，迎接实际工作中的iOS开发挑战。

2.8 初级练习：查找问题

试为RandomItems制造一处错误：要求NSMutableArray对象items返回第11个对象。运行应用并观察其抛出的异常。

2.9 中级练习：另一个初始化方法

试为BNRItem再创建一个初始化方法。这个初始化方法不是BNRItem的指定初始化方法。新方法要有两个NSString类型的参数，分别针对实例变量itemName和实例变量serialNumber。

2.10 高级练习：另一个类

试创建一个BNRItem子类并将其命名为BNRContainer。BNRContainer对象包含一个名为subitems的数组对象，其中都是BNRItem对象。BNRContainer的实例方法description能返回BNRContainer对象的名称、价值（其下所有的BNRItem对象的价值总和，再加上BNRContainer对象自身的价值）和其包含的所有BNRItem对象。如果编写正确，BNRContainer对象也能包含其他BNRContainer对象，并且其description方法能够正确返回价值总和，以及所有包含的BNRItem对象。

2.11 关于深入学习部分

除了练习，很多章节还会包含一个或多个“深入学习”部分。这些部分会针对当前章节的内容做更深入的介绍，或者提供更多的信息。深入学习部分所介绍的知识不是必须掌握的，但是这些内容对学习iOS开发很有帮助。

2.12 深入学习：如何为类命名

开发RandomItems这样的小规模应用时，只要创建少量的类就能完成任务。但是当应用的规模逐渐扩大，实现越来越复杂时，要创建的类会越来越多。当项目规模达到一定程度时，很可能产生问题：两个不同的类拥有相同的名称。当两个不同的类拥有相同的名称时，编译器将无法判断应该使用哪一个。这类问题称为名字空间冲突（namespace collision）。

其他编程语言是通过引入名字空间（namespace）机制来解决这类问题的。读者可以将名字空间想象成组，不同的类可以归在不同的组中。在这些编程语言中使用类时，需要同时给出类名和其所属的名字空间。

Objective-C没有提供名字空间机制。为了能更好地区分类，需要为类名增加前缀（长度为2至3个字符）。以RandomItems为例，BNRItem类的类名就有BNR前缀，而不是单纯的Item。

有着良好编程风格的程序员都会为类名加上前缀。这些前缀通常会和当前开发的应用有关，或者和代码所属的代码库有关。举一个假设的例子，如果读者正在开发一个名为“MovieViewer”的应用，就可以将所有和这个项目有关的类的类名都加上前缀MOV。对于会在多个项目中共享的类，其前缀通常会和程序员的名字（例如CBK）、公司的名称（例如BNR）或类库（例如一个专门提供地图功能的库，可以使用MAP前缀）有关。

Apple所提供的类也有前缀。这些类都是通过框架组织的，并且每个框架都有自己的前缀。例如UILabel类是由UIKit框架提供的，NSArray类和NSString类是由Foundation框架提供的（NS代表NeXTSTEP，这些类最初都是为NeXTSTEP系统开发的）。

读者在编写类时，应该使用3个字符的前缀，避免和Apple提供的类产生冲突——它们都是2个字符的前缀。即使现在使用只有2个字符的前缀没有出错，也应该尽量改为使用3个字符的前缀，以减少与Apple未来发布的类产生冲突的可能。

2.13 深入学习：#import和@import

和Objective-C刚刚诞生时相比，现在的Objective-C中，类的数量已经非常庞大，有必要将类按照功能纳入框架中管理。在之前的代码中，通常会在头文件中导入Foundation框架：

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

代码会导入Foundation框架中的所有头文件：

```
#import <Foundation/NSArray.h>

#import <Foundation/NSAutoreleasePool.h>

#import <Foundation/NSBundle.h>

#import <Foundation/NSByteOrder.h>

#import <Foundation/NSCalendar.h>

#import <Foundation/NSCharacterSet.h>

...
```

这样，文件中就不用再明确导入Foundation框架中的任何类了。编译器会使用C语言的预处理器将所有需要的头文件拷贝到文件中。

直接导入整个框架的方式开始可以满足要求，但是随着框架中的类以及项目中需要使用的框架越来越多，编译器也会花费越来越长的时间处理大量重复的头文件。为了解决这个问题，Xcode为所有项目都添加了一个预编译头文件（precompiled header file），第一次编译项目时，预编译头文件中列出的文件会被编译并缓存，编译器会重复使用缓存结果快速编译项目中的其他文件。之前创建的RandomItems项目有一个RandomItems-Prefix.pch文件，编译器会预编译该文件中列出的Foundation框架：

```
#ifdef __OBJC__

#import <Foundation/Foundation.h>

#endif
```

在预编译头文件中，仍然需要明确导入Foundation框架。

这种方式在一段时期内很好地满足了Objective-C程序员对编译速度的要求，但是Apple近期发现在项目中维护。pch文件低效耗时，因此继续优化编译器并引入了@import指令：

```
@import Foundation;
```

这行代码告诉编译器需要使用Foundation框架，之后编译器会优化预编译头文件和缓存编

译结果的过程。同时，文件中不用再明确引用框架——编译器会根据@import自动导入相应的框架。

目前只有Apple提供的框架可以使用@import。如果需要导入自己编写的类和框架，只能使用#import。

在写作本书时，Xcode 5.0.2的模板文件仍然使用#import，但是可以肯定，未来@import的使用将会越来越多。

第3章 通过ARC管理内存

本章将介绍iOS的内存管理机制，以及其背后的自动引用计数（automatic reference counting, ARC）特性。下面先介绍若干与应用内存有关的基本概念。

3.1 栈

当程序执行某个方法(或函数)时，会从内存中名为栈(stack)的区域分配一块内存空间，这块内存空间称为帧(frame)。帧负责保存程序在方法内声明的变量的值。在方法内声明的变量称为局部变量(local variable)。

当某个应用启动并运行main函数时，它的帧会被保存在栈的底部。当main调用另一个方法(或函数)时，这个方法(或函数)的帧会压入栈的顶部。被调用的方法还可以再调用其他方法，依此类推，最终会在栈中形成一个塔状的帧序列。当被调用的方法(或函数)结束时，程序会将其帧从栈顶“弹出”并释放。如果同一个方法再次被调用，则应用会创建一个全新的帧，并将其压入栈的顶部。

以RandomItems为例，它的main函数会调用BNRItem的randomItem方法，randomItem又会调用alloc方法。调用这些方法后的栈的状态如图3-1所示。需要注意的是，当程序在执行main函数并调用其他方法时，会在栈中保留main函数的帧。这是因为程序还没有完成main函数的整个执行过程。

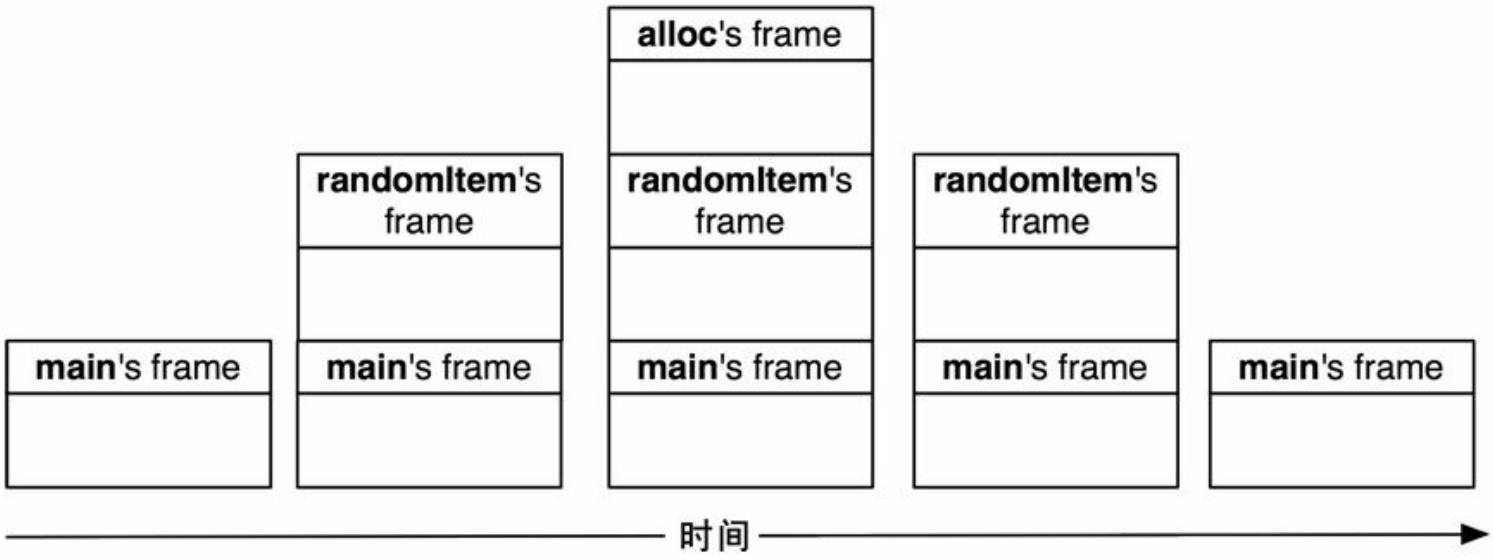


图3-1　调用RandomItems的main函数时栈的变化过程

在第2章完成的代码中，RandomItems会在main函数中循环调用BNRItem的randomItem方法。当应用每次调用BNRItem的randomItem方法时，RandomItems的栈就会随着相应帧的推入和弹出，先变大再变小。

3.2 堆

堆(heap)是指内存中的另一块区域，和栈是分开的。为这两类内存区域分别取名堆和栈，是为了能够形象地描述这两个概念。栈会按后进先出的规则保存一组帧，而堆则包含了大量无序的活动对象，需要通过指针来保存这些对象在堆中的地址。

当应用向某个类发送alloc消息时，系统会从堆中分配出一块内存，其大小足够存放相应对象的全部实例变量。以BNRItem对象为例，BNRItem对象包含5个实例变量，其中4个为指针变量(isa、_itemName、_serialNumber和_dateCreated)，另一个为int变量(_valueInDollars)。因此，系统会为1个int变量和4个指针变量分配内存，其中指针变量保存其他对象在堆中的地址。

ios应用在启动和运行时会持续创建需要的对象，如果堆的空间是无限的，则可以随意创建所需的对象，并且在应用运行期间不用释放。

但是可供应用支配的堆空间是有限的，而且iOS设备的内存也非常有限。因此，当应用不再需要某些对象时，就要将其释放掉。这是非常重要的一步，因为释放对象后，可以将其占用的内存归还给堆，使之能够被重新使用。另外，也要绝对避免释放应用正在使用的对象。

ARC和内存管理

幸运的是，编写iOS应用时，并不需要记录对象是否应该保留或释放，而只需要通过ARC管理内存，也就是自动引用计数。本书中所有应用都会使用ARC。在Apple引入ARC之前，应用只能通过手动引用计数(manual reference counting)来管理内存。本章结尾部分会对手动引用计数做更多的介绍。

大多数情况下，可以依靠ARC来自动地完成应用的内存管理工作。但是，为了能够在需要的时候手动处理某些特殊的内存管理问题，理解其背后的工作原理也很重要。所以下面要进一步介绍“对象所有权”概念。

3.3 指针变量与对象所有权

指针变量暗含了对其所指向的对象的所有权(ownership)。

•当某个方法(或函数)有一个指向某个对象的局部变量时，可以称该变量拥有(own)该变量所指向的对象。

•当某个对象有一个指向其他对象的实例变量时，可以称该对象拥有该实例变量所指向的对象。

请读者回想RandomItems应用，应用在main函数中创建一个NSMutableArray对象，然后创建10个BNRItem对象并加入该数组。图3-2显示的是RandomItems中的对象及指向这些对象的指针。

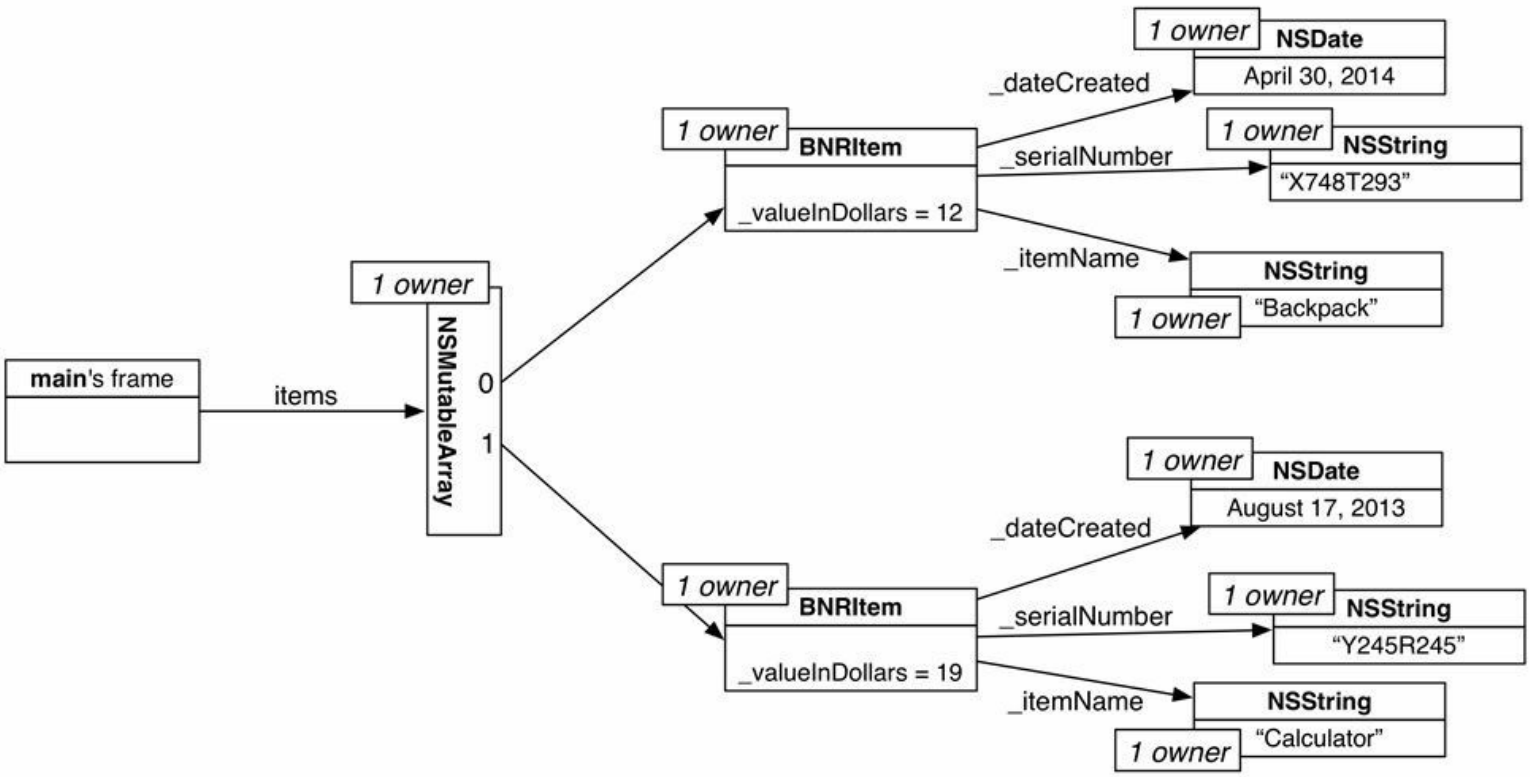


图3-2　RandomItems对象图(图中有两个BNRItem对象)

因为指向NSMutableArray对象的items指针是main函数的局部变量，所以main函数拥有相应的NSMutableArray对象。

NSMutableArray对象拥有其包含的所有BNRItem对象。对NSMutableArray这种collection类，其对象保存的是指向对象的指针，而不是对象自身。这些指针暗含了以下所有权:数组对象拥有其包含的所有对象。

最后，每一个BNRItem对象都拥有其实例变量所指向的对象。

对象所有权概念可以帮助我们决定是否应该释放某个对象并回收该对象占有的内存。

•如果某个对象没有拥有者，就应该将其释放掉。没有拥有者的对象是孤立的，程序无法向其发送消息。保留这样的对象只会浪费宝贵的内存空间，导致内存泄露（memory leak）问题。

•如果某个对象有一个或多个拥有者，就必须保留不能释放。如果释放了某个对象，但是其他对象或方法仍然有指向该对象的指针（准确地说，是指向该对象被释放前的地址），那么向该指针指向的对象发送消息就会使应用崩溃。释放正在使用的对象的错误称为过早释放。指向不存在的对象的指针称为空指针（dangling pointer）或者空引用（dangling reference）。

哪些情况会使对象失去拥有者

下列情况会使对象失去拥有者：

•当程序修改某个指向特定对象的变量并将其指向另一个对象时。

•当程序将某个指向特定对象的变量设置为nil时。

•当程序释放对象的某个拥有者时。

•当从collection类中（例如数组）删除对象时。

下面逐个介绍这4种情况。

修改指针变量

以BNRItem为例，假设有某个BNRItem对象，其实例变量_itemName所指向的NSString对象是@“Rusty Spork”（生锈的叉勺）。如果有人将这个叉勺抛光，去除锈迹，那么“生锈的叉勺”就变成“闪闪发光的叉勺”（Shiny Spork）。这时，就需要将_itemName指向一个不同的NSString对象（见图3-3）。

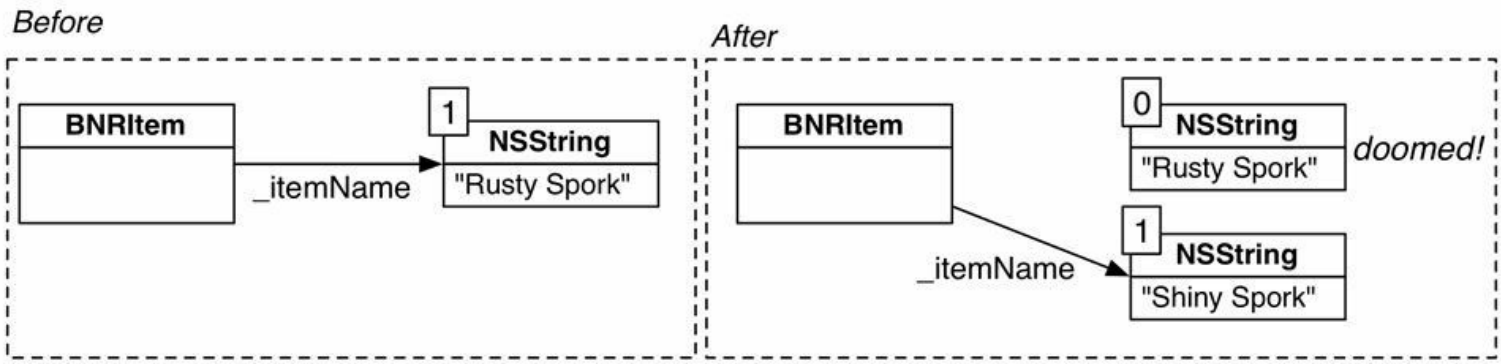


图3-3　修改指针变量所指向的对象

当应用修改_itemName的值，将其从指向@“Rusty Spork”的地址改为指向@“Shiny Spork”的

地址时，@“Rusty Spork”就会失去一个拥有者。如果此时@“Rusty Spork”已经没有其他拥有者了，它就应该被释放。

将指针变量的值设为nil

如果某个指针变量的值是nil，就代表这个指针没有指向任何对象。以BNRItem为例，假设有一个代表某台电视机的BNRItem对象，当这台电视机的序列号被刮去时，就应该将相应BNRItem对象的实例变量_serialNumber设为nil。而这个_serialNumber曾经指向的NSString对象将失去一个拥有者。

对象的拥有者被释放

对象的拥有者被释放时，就会失去一个拥有者，所以释放一个对象可能会造成其他对象失去拥有者。

方法或函数可以通过局部变量成为对象的拥有者。当程序执行完某个方法，将其帧弹出栈时，就会释放该方法的所有局部变量。在这些变量中，如果有指向其他对象的指针变量，那么这些对象就会失去一个拥有者。

从collection对象中删除对象

collection对象会拥有其包含的对象。当程序将某个对象移出可修改的collection对象（例如NSMutableArray对象）时，这个被移出的对象就会失去一个拥有者，代码如下：

```
[items removeObject,item];  //item所指向的对象会失去一个拥有者
```

需要注意的是，当某个对象失去一个拥有者时，程序不一定会释放这个对象。只要还有另一个指向该对象的指针，程序就会保留这个对象。但是当某个对象失去最后一个拥有者时，就一定会被释放。

所有权链（Ownership chains）

因为对象可以拥有其他对象，后者也可以再拥有别的对象，所以释放一个对象可能会产生连锁反应，导致多个对象失去拥有者，进而释放对象并归还内存。

RondomItems中就有一个现成的例子，请读者先回顾RandomItems的对象图（见图3-2）。

在main.m中，RandomItems会先输出包含多个BNRItem对象的items数组，然后将items变量设

置为nil。当程序将items变量设置为nil时，会导致其指向的NSMutableArray对象失去唯一的拥有者，并因此被释放。

但这只是开始。当程序释放NSMutableArray对象时，也会释放其包含的所有指向BNRItem对象的指针。一旦程序释放这些指针变量，相应的BNRItem对象将失去最后一个拥有者，并被程序释放。

程序在释放BNRItem对象时，也会释放其实例变量。在这些实例变量中，如果是指向其他对象的指针变量，这些对象就会失去一个拥有者。因为这些对象只有BNRItem对象一个拥有者，所以也会被释放。

下面为RandomItems加入测试代码，输出上述释放过程。NSObject实现了一个名为dealloc的方法。当某个对象即将被释放时，程序会调用该对象的dealloc方法。通过覆盖BNRItem的dealloc方法，可以在程序释放BNRItem对象时向控制台输出一行提示。

打开RandomItems项目，选中BNRItem.m，然后覆盖dealloc，代码如下：

```
- (void)dealloc

{

NSLog(@"Destroyed: %@", self);

}
```

在main.m中加入下面这行代码：

```
NSLog(@"Setting items to nil...");

items = nil;
```

构建并运行应用，和之前一样，控制台应该会输出BNRItem数组的描述信息。新增加的提示内容，首先是“items将被设置为nil”（Setting items to nil…），然后是针对每个BNRItem对象的“某对象已经被释放”（Destroyed: %@）。

最后，程序会释放所有的对象，只剩下main函数。程序仅仅是将items设置为nil，就完成了这些自动的内存清理和回收工作。使用ARC的好处可见一斑。

3.4 强引用与弱引用

之前介绍过，只要指针变量指向了某个对象，那么相应的对象就会多一个拥有者，并且不会被程序释放。这种指针特性(attribute)称为强引用(strong reference)。

程序也可以选择让指针变量不影响其指向对象的拥有者个数。这种不会改变对象拥有者个数的指针特性称为弱引用(weak reference)。

弱引用非常适合解决一种称为强引用循环(strong reference cycle，有时也称为保留循环)的内存管理问题。当两个或两个以上的对象相互之间有强引用特性的指针关联时，就会产生强引用循环。强引用循环会导致内存泄露。当两个对象互相拥有时，将无法通过ARC机制来释放。即使应用中的其他对象都释放了针对这两个对象的所有权，这两个对象及其拥有的所有对象也无法被释放。

因此在编写应用时，程序员必须自己做一些额外的工作，才能帮助ARC解决强引用循环所导致的内存泄露问题。解决强引用循环的途径是将某个指针的特性设置为弱引用。

下面为RandomItems加入一处强引用循环，借以向读者介绍如何解决此类问题。首先，让BNRItem对象能够保存另外一个BNRItem对象(这样就能表现背包和钱包这样的物品关系)。此外，BNRItem对象会有一个指针实例变量，指回包含该对象的BNRItem对象。

在BNRItem.h中，增加两个实例变量和相应的存取方法，代码如下：

```
@interface BNRItem : NSObject

{

NSString *_itemName;

NSString *_serialNumber;

int _valueInDollars;

NSDate *_dateCreated;

BNRItem *_containedItem;

BNRItem *_container;

}

+ (instancetype)randomItem;

- (instancetype)initWithItemName:(NSString *)name

        valueInDollars:(int)value
```

```
      serialNumber:(NSString *)sNumber;
- (instancetype)initWithItemName:(NSString *)name;
- (void)setContainedItem:(BNRItem *)item;
- (BNRItem *)containedItem;
- (void)setContainer:(BNRItem *)item;
- (BNRItem *)container;
```

在BNRItem.m中实现新加入的存取方法，代码如下：

```
- (void)setContainedItem:(BNRItem *)item
{
_containedItem = item;
// 将item加入容纳它的BNRItem对象时，
// 会将它的container实例变量指向容纳它的对象。
item.container = self;
}
- (BNRItem *)containedItem
{
return _containedItem;
}
- (void)setContainer:(BNRItem *)item
{
_container = item;
}
- (BNRItem *)container
{
```

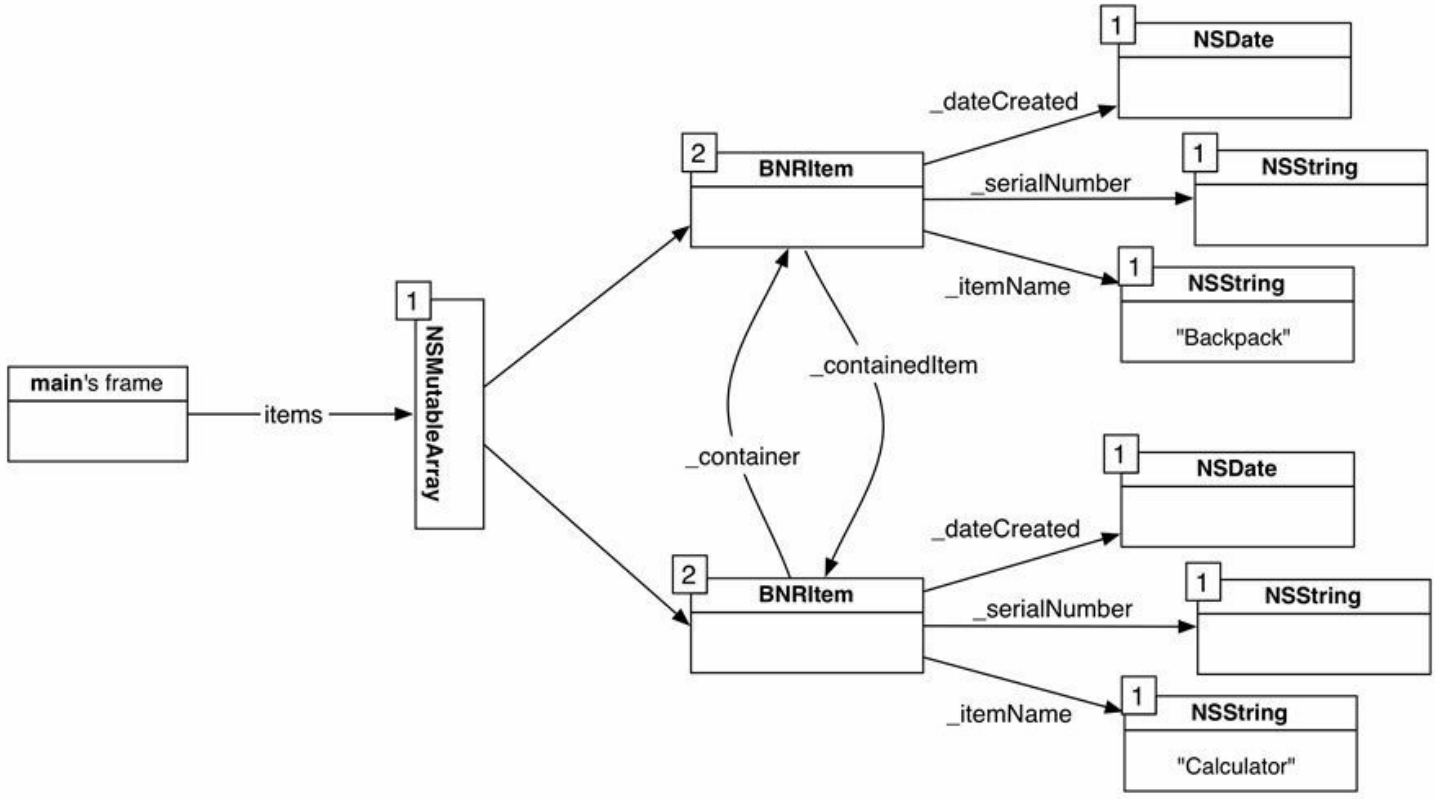
```
return _container;
}
```

在main.m中，首先删除创建并枚举多个随机BNRItem对象的代码。然后创建两个新的BNRItem对象并加入items数组。最后将这两个对象用指针关联起来，代码如下：

```
int main (int argc, const char * argv[])
{
@autoreleasepool {
    NSMutableArray *items = [[NSMutableArray alloc] init];
    for (int i = 0; i < 10; i++) {
        BNRItem *item = [BNRItem randomItem];
        [items addObject:item];
    }
    BNRItem *backpack = [[BNRItem alloc] initWithItemName:@"Backpack"];
    [items addObject:backpack];
    BNRItem *calculator = [[BNRItem alloc] initWithItemName:@"Calculator"];
    [items addObject:calculator];
    backpack.containedItem = calculator;
    backpack = nil;
    calculator = nil;
    for (BNRItem *item in items)
        NSLog(@"%@", item);
    NSLog(@"Setting items to nil...");
    items = nil;
}
return 0;
```

}

修改后的RandomItems的对象图如图3-4所示。



>

图3-4　存在强引用循环问题的RondomItems

构建并运行应用，检查控制台输出，会发现应用并没有输出释放这些对象的提示信息。

RondomItems无法正确地释放对象是由强引用循环引起的：backpack变量指向的对象和calculator变量指向的对象都有强引用特性的指针，并指向彼此。图3-5显示的是在应用将items设置为nil后，没有被释放并继续占用内存的所有对象。

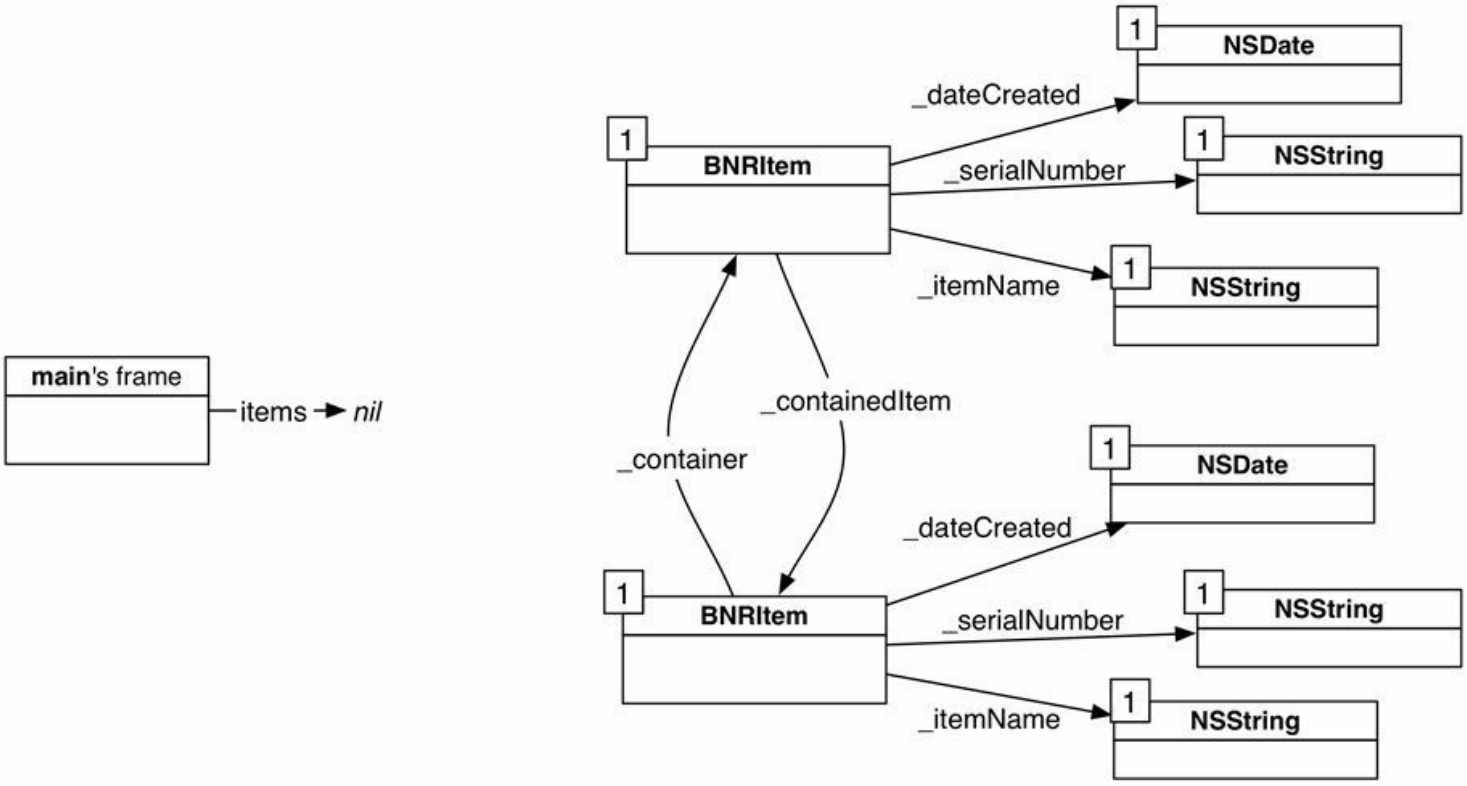


图3-5　对象没有被释放，造成内存泄露

在RandomItems将items设置为nil后，除了新创建的两个对象自身外，应用的其余部分（例如这段代码中的main函数）都将无法使用这两个对象，并且这两个无法被使用的对象会一直占用应用的内存。此外，因为应用不会释放这两个对象，所以，如果这些对象的实例变量还指向了其他对象，那么这些被指向的对象也不会被释放。

要解决RandomItems的强引用循环问题，需要将新创建的两个BNRItem对象之间的某个指针改为弱引用特性。在决定将哪个指针改为弱引用前，可以先为存在强引用循环问题的多个对象决定相应的父-子关系（parent-child relationship）。确定父-子关系后，就可以让父对象拥有子对象，并确保子对象不会拥有父对象。以RandomItems的强引用循环为例，backpack是父对象，calculator是子对象。根据以上规则，可以将backpack指向calculator（_containedItem）的指针保留为强引用特性，并将calculator指向backpack（_container）的指针改为弱引用特性。

使用__weak关键字，可以将某个变量声明为具有弱引用特性。在BNRItem.h中，为实例变量container增加弱引用特性，代码如下：

```
__weak BNRItem *_container;
```

构建并运行应用，修改后的RandomItems能正确地释放新创建的两个BNRItem对象。

大部分强引用循环问题都可以为其确定一个父-子关系。通常情况下，父对象应该使用具有强引用特性的指针，指向子对象。而子对象则应该使用具有弱引用特性的指针，指回父对象。这样就可以避免强引用循环问题。

如果某个子对象具有一个强引用特性的指针，指向父对象的父对象，一样也会导致强引用循环。所以上述规则也适用于：如果某个子对象需要有一个指针，指向父对象的父对象（或者

是父对象的父对象的父对象，等等），那么该指针必须具有弱引用特性。

Xcode的Leaks工具可以帮助我们找出强引用循环问题。第14章会介绍如何使用Leaks工具。

具有弱引用特性的指针指向的对象被释放后，指针会自动设置为nil。以RandomItems为例，当应用释放backpack后，会自动地将calculator的实例变量_container设置为nil。这就是弱引用的好处：如果_container没有被设置为nil，backpack对象被释放后会留下一个空指针，访问该指针就会引起程序崩溃。

修正强引用循环问题后，RandomItems的新对象图如图3-6所示。注意，图中代表指针变量_container的箭头已经改为虚线。虚线代表具有弱引用特性的指针。强引用特性的指针变量仍然用实线表示。

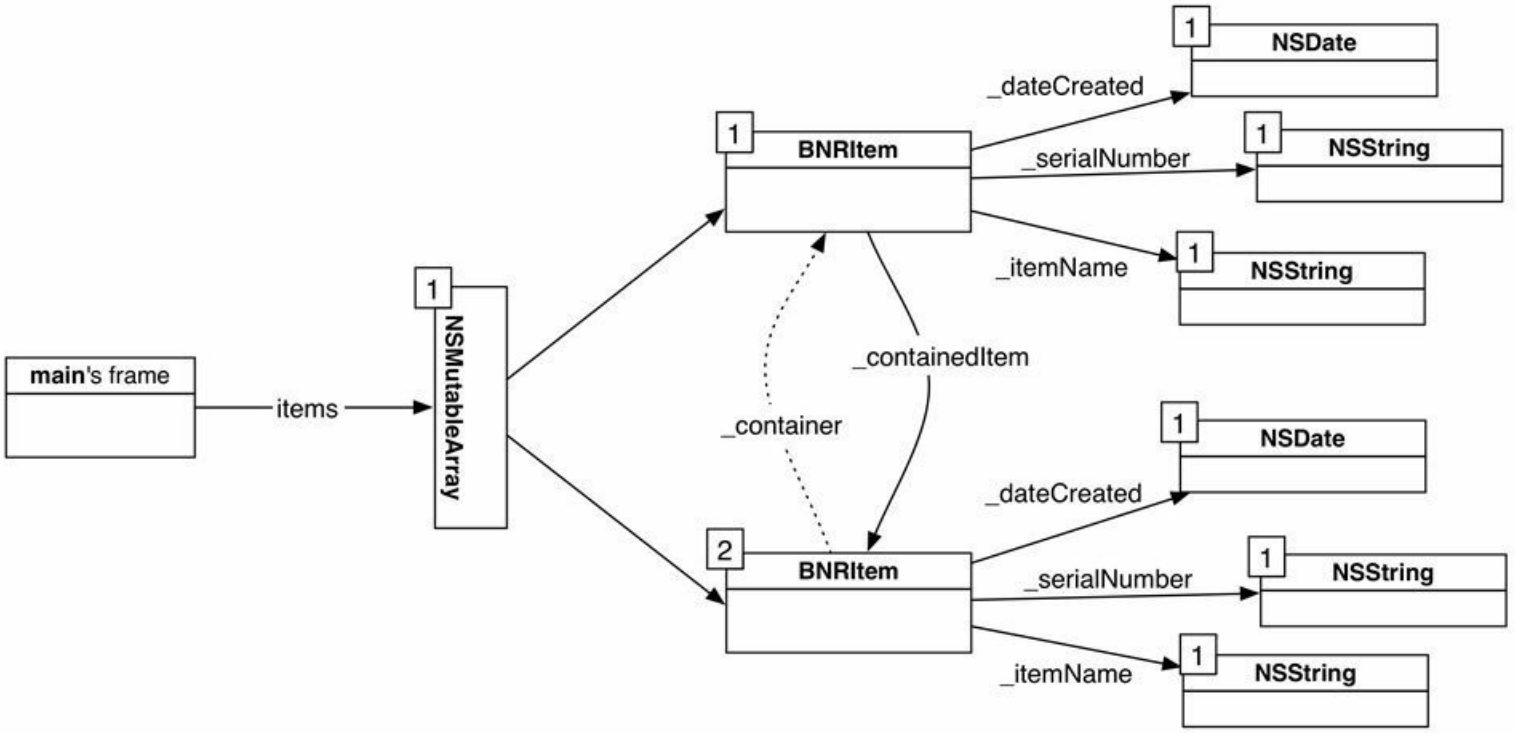


图3-6　解决强引用循环问题后的RondomItems

3.5 属性

之前在编写BNRItem时，需要为每一个实例变量声明并实现一对存取方法。下面介绍如何通过属性来简化这个过程。通过属性，也可以为类声明实例变量并实现相应的存取方法，而且更简便。使用属性后，可以大幅减少所需编写的代码量，并且写出的类文件也更容易理解。

声明属性

下面先给出一则属性声明的示例：

@property NSString *itemName;

声明一个属性，等于隐含地为相应名称的实例变量声明一对存取方法。请看表3-1，左边是没有使用属性的类，右边则使用了属性，但是两个类是完全等价的。

表3-1　使用和不使用属性的两个等价类

	不使用属性	使用属性

BNRThing.h	@interface BNRThing : NSObject
	{
	NSString *_name;
	}
	- (void)setName:(NSString *)n;
	- (NSString *)name;
	@end
		@interface BNRThing : NSObject
		
		@property NSString *name;
		
		@end

BNRThing.m	@implementation BNRThing
	
	- (void)setName:(NSString *)n
	{
	_name = n;
	}
	
	- (NSString *)name
	{
	return _name;
	}
	
	@end
		@implementation BNRThing
		@end

表3-1中的两个类都具有一个实例变量_name和一对存取方法，左边的类中需要将声明和实现都明确地写出来，而右边只需要声明一个属性就可以达到相同的效果。

下面修改BNRItem类，使用属性替换实例变量和存取方法。

打开BNRItem.h，删除所有实例变量和存取方法声明，改为相应的属性，代码如下：

```
@interface BNRItem : NSObject
{
NSString *_itemName;
NSString *_serialNumber;
int _valueInDollars;
NSDate *_dateCreated;
BNRItem *_containedItem;
__weak BNRItem *_container;
}
@property BNRItem *containedItem;
@property BNRItem *container;
@property NSString *itemName;
@property NSString *serialNumber;
@property int valueInDollars;
@property NSDate *dateCreated;
+ (instancetype)randomItem;
- (instancetype)initWithItemName:(NSString *)name
        valueInDollars:(int)value
          serialNumber:(NSString *)sNumber;
- (instancetype)initWithItemName:(NSString *)name;
- (void)setItemName:(NSString *)str;
- (NSString *)itemName;
```

~~- (void)setSerialNumber:(NSString *)str;~~

~~- (NSString *)serialNumber;~~

~~- (void)setValueInDollars:(int)v;~~

~~- (int)valueInDollars;~~

~~- (NSDate *)dateCreated;~~

~~- (void)setContainedItem:(BNRItem *)item;~~

~~- (BNRItem *)containedItem;~~

~~- (void)setContainer:(BNRItem *)item;~~

~~- (BNRItem *)container;~~

@end

现在的BNRItem.h可读性更好，代码如下：

```
@interface BNRItem : NSObject
+ (instancetype)randomItem;
- (instancetype)initWithItemName:(NSString *)name
        valueInDollars:(int)value
         serialNumber:(NSString *)sNumber;
- (instancetype)initWithItemName:(NSString *)name;
@property BNRItem *containedItem;
@property BNRItem *container;
@property NSString *itemName;
@property NSString *serialNumber;
@property int valueInDollars;
@property NSDate *dateCreated;
@end
```

请注意属性的名字是实例变量的名字去掉下画线，编译器根据属性生成实例变量时会自动在变量名前加上下画线。

如果声明了一个名为itemName的属性，编译器会自动生成实例变量_itemName、取方法itemName和存方法setItemName:。(请注意实例变量和存取方法的声明不会出现在文件中，编译器会在编译时自动加入这些代码)因此程序能像之前一样正常工作。

声明属性还可以为相应的存取方法生成代码。在BNRItem.m中，删除之前实现的存取方法。

```
- (void)setItemName:(NSString *)str
{
_itemName = str;
}
- (NSString *)itemName
{
return _itemName;
}
- (void)setSerialNumber:(NSString *)str
{
_serialNumber = str;
}
- (NSString *)serialNumber
{
return _serialNumber;
}
- (void)setValueInDollars:(int)p
{
_valueInDollars = p;
}
```

```
- (int)valueInDollars
{
return _valueInDollars;
}
- (NSDate *)dateCreated
{
return _dateCreated;
}
- (void)setContainedItem:(BNRItem *)item
{
_containedItem = item;
item.container = self;
}
- (BNRItem *)containedItem
{
return _containedItem;
}
- (void)setContainer:(BNRItem *)item
{
_container = item;
}
- (BNRItem *)container
{
return _container;
```

```
}
```

读者可能会注意到setContainedItem:方法。该方法除了设置_containedItem实例变量外，还设置了传入的BNRItem对象的_container实例变量。之后读者会学习如何自定义存取方法。接下来学习有关属性的基本知识。

属性的特性

任何属性都可以有一组特性(attribute)，用于描述相应存取方法的行为。这些特性需要写在小括号里，并跟在@property指令后面，示例如下：

```
@property (nonatomic, readwrite, strong) NSString *itemName;
```

任何属性都有三个特性，每个特性都有多种不同的可选类型。在这些可选类型中，有一种是默认的。如果属性的某个特性使用默认类型，就可以在声明该属性时忽略这项特性。

多线程特性

多线程特性(Multi-threading attribute)有两种可选类型：nonatomic和atomic。(多线程超出了本书的讨论范围，这里只需要知道有这两个类型即可。)大多数Objective-C程序员会将这个特性设置为nonatomic，Big Nerd Ranch也是，Apple也是。本书代码中的所有属性都会使用nonatomic。

因为nonatomic不是默认类型，所以在声明属性时，必须明确地写出nonatomic。

打开BNRItem.h，为所有属性添加nonatomic特性，代码如下：

```
@interface BNRItem : NSObject

+ (instancetype)randomItem;

- (instancetype)initWithItemName:(NSString *)name

        valueInDollars:(int)value

         serialNumber:(NSString *)sNumber;

- (instancetype)initWithItemName:(NSString *)name;

@property (nonatomic) BNRItem *containedItem;

@property (nonatomic) BNRItem *container;
```

```
@property (nonatomic) NSString *itemName;

@property (nonatomic) NSString *serialNumber;

@property (nonatomic) int valueInDollars;

@property (nonatomic) NSDate *dateCreated;

@end
```

读/写特性

读/写特性(Read/write attribute)也有两种可选类型:readwrite和readonly。编译器会为具有readwrite特性的属性生成存方法和取方法,如果是readonly类型,则只会生成取方法。第二个特性的默认类型是readwrite。BNRItem中的属性除了dateCreated是readonly类型,其他的都是readwrite类型。

在BNRItem.h中,将dateCreated声明为readonly的属性,要求编译器只为相应的实例变量生成取方法,代码如下:

```
@property (nonatomic, readonly) NSDate *dateCreated;
```

内存管理特性

内存管理特性(Memory management attribute)有四种可选类型:strong、weak、copy和unsafe_unretained。这些类型决定相应的实例变量将如何引用对象。

对于不指向任何对象的属性(例如int valueInDollars),不需要做内存管理,这时应该选用unsafe_unretained,它表示存取方法会直接为实例变量赋值。Apple引入ARC之前曾经使用assign表示这种类型。

unsafe_unretained中的"unsafe(不安全)"可能会误导读者。该类型的"不安全"是相对于弱引用而言的。与弱引用不同,unsafe_unretained类型的指针指向的对象被销毁时,指针不会自动设置为nil,而是成为空指针,因此不安全。但是当处理非对象属性(non-object)时,是不会出现空指针问题的。

unsafe_unretained是非对象属性的默认值,所以valueInDollars属性不用明确写出该类型。

对于指向Objective-C对象的属性,四种类型都有可能。默认是strong类型,但是通常程序员会明确写出strong。(Apple曾经修改过默认值,未来仍然可能有改动。)

在BNRItem.m中,为属性添加内存管理特性,其中containedItem和dateCreated属性设置为

strong, container属性设置为weak:

@property (nonatomic, strong) BNRItem *containedItem;

@property (nonatomic, weak) BNRItem *container;

@property (nonatomic) NSString *itemName;

@property (nonatomic) NSString *serialNumber;

@property (nonatomic) int valueInDollars;

@property (nonatomic, readonly, strong) NSDate *dateCreated;

将container属性设置为weak是为了避免强引用循环，之前的代码演示过这个问题。

现在itemName和serialNumber属性还没有添加内存管理特性。它们是指向NSString对象的属性。通常情况下，当某个属性是指向其他对象的指针，而且该对象的类有可修改的子类（例如NSString/NSMutableString或NSArray/NSMutableArray）时，应该将该属性的内存管理特性设置为copy。

在BNRItem.m中，将itemName和serialNumber属性的内存管理特性设置为copy:

@property (nonatomic, strong) BNRItem *containedItem;

@property (nonatomic, weak) BNRItem *container;

@property (nonatomic, copy) NSString *itemName;

@property (nonatomic, copy) NSString *serialNumber;

@property (nonatomic) int valueInDollars;

@property (nonatomic, readonly, strong) NSDate *dateCreated;

改用copy特性后，itemName属性的存方法可能类似于以下代码：

- (void)setItemName:(NSString *)itemName

{

_itemName = [itemName copy];

}

和前一个版本的setItemName:方法不同，这段代码没有直接将传入的itemName赋给实例变

量_itemName，而是先向itemName发送了copy消息。该对象的copy方法会返回一个新的NSString对象（不是NSMutableString对象），并且其拥有的数据会和收到copy消息的那个对象相同。接着，新版本的setItemName:方法会为实例变量_itemName赋值，指向新的NSString对象。

这样做的原因是，如果属性指向的对象的类有可修改的子类，那么该属性可能会指向可修改的子类对象，同时，该对象可能会被其他拥有者修改。因此，最好先复制该对象，然后再将属性指向复制后的对象。

以BNRItem为例，假设有某个初始化后的BNRItem对象，其实例变量_itemName指向一个NSMutableString对象，代码如下：

```
NSMutableString *mutableString = [[NSMutableString alloc] init];

BNRItem *item = [[BNRItem alloc] initWithItemName:mutableString

        valueInDollars:5

          serialNumber:@"4F2W7"]];
```

这段代码是有效的，因为凡是可以使用NSString对象的地方，也可以使用NSMutableString对象（NSMutableString是NSString的子类）。真正的问题是程序可能在BNRItem对象不知情的情况下修改mutableString变量所指向的NSMutableString对象。

当读者可以掌控某个应用的全部代码时，自然可以确保mutableString变量所指向的NSMutableString对象不会被意外地修改。但是，当其他程序员也会使用读者编写的类时，就要做最坏的打算，编写具有“防御性”的代码。

对于这段代码来说，需要加入的防御措施是将itemName属性的内存管理特性设置为copy。

至于所有权，copy方法返回的是拥有强引用特性的指针，而收到copy消息的NSString对象不会发生任何变化：该对象不会获得也不会失去拥有者，其数据也不会发生任何变化。

只有可变对象应该设置为copy，而复制不可变对象会浪费内存空间——不可变对象不会发生上述问题，因为任何对象都无法修改它们。为了避免不必要的复制，向不可变对象发送copy消息时，会返回一个指向自己（仍然是不可变的）的指针。

自定义属性的存取方法

默认情况下，属性自动添加的存取方法非常简单，类似以下代码：

```
- (void)setContainedItem:(BNRItem *)item

{

_containedItem = item;
```

```
}

- (BNRItem *)containedItem

{

return _containedItem;

}
```

属性自动添加的存取方法大部分是可以直接使用的。但是对于containedItem属性，默认的存方法无法满足要求，setContainedItem:方法需要完成额外的工作：设置传入对象的container属性。

可以在实现文件中编写自定义的存方法，覆盖默认的方法。在BNRItem.m中，加回之前实现的setContainedItem:方法：

```
- (void)setContainedItem:(BNRItem *)containedItem

{

_containedItem = containedItem;

self.containedItem.container = self;

}
```

编译器看到自定义的setContainedItem:方法之后，不会再为containedItem属性创建默认的存方法。但是仍然会创建默认的取方法containedItem。

请注意，如果既覆盖了存方法，也覆盖了取方法（或者为只读属性覆盖了取方法），那么编译器就不会再自动创建相应的实例变量了。如果需要实例变量，就必须明确声明。

如果默认的存取方法无法满足要求，必须手动为相应的实例变量实现自定义的存取方法。

现在构建并运行应用，BNRItem的运行结果应该和之前的完全相同。

3.6 深入学习：属性合成

本章介绍过，属性会自动生成存取方法，也会自动声明和创建实例变量。实际上，我们还可以自定义生成过程，得到符合实际需要的实例变量和存取方法。

在头文件中声明属性时，只会生成存取方法的声明。为了让属性生成实例变量并实现存取方法，该属性必须被合成（synthesized）。通常情况下，编译器会自动合成属性并生成默认的实例变量和存取方法。如果需要自定义属性的合成方式，可以在实现文件中使用@synthesize指令：

```
@implementation Person

// 创建存取方法，方法名是age和setAge:，

// 同时创建实例变量_age

@synthesize age = _age;

// 其他方法

@end
```

以上代码与编译器自动合成的效果相同。赋值号左边的age表示需要创建存取方法，方法名是age和setAge:。右边的_age表示需要创建实例变量，变量名为_age。

也可以不写变量名，这样实例变量的变量名会和方法名相同：

```
@synthesize age;

// 和以下语句效果相同：

@synthesize age = age;
```

有时我们不希望属性自动生成实例变量和存取方法。例如，有一个Person类，类中有三个属性：spouse（配偶）、lastName（姓氏）和lastNameOfSpouse（配偶的姓氏）。

```
@interface Person : NSObject

@property (nonatomic, strong) Person *spouse;

@property (nonatomic, copy) NSString *lastName;

@property (nonatomic, copy) NSString *lastNameOfSpouse;

@end
```

在这个例子中，spouse和lastName属性需要生成实例变量，因为配偶和姓氏是个人基本信

息。而lastNameOfSpouse属性则不用生成实例变量，因为可以通过spouse的lastName属性知道配偶的姓氏，所以不需要在两个Person对象中都生成实例变量，既浪费内存也容易出错。我们可以为lastNameOfSpouse属性自定义存取方法：

```
@implementation Person

- (void)setLastNameOfSpouse:(NSString *)lastNameOfSpouse

{

self.spouse.lastName = lastNameOfSpouse;

}

- (NSString *)lastNameOfSpouse

{

return self.spouse.lastName;

}

@end
```

由于同时覆盖了存方法和取方法，编译器不会再为lastNameOfSpouse自动生成实例变量，和期望效果一致。

3.7 深入学习：Autorelease池与ARC历史

在Objective-C支持ARC之前，只能用手动引用计数（manual reference counting）来管理内存。使用手动引用计数时，必须向某个对象发送特定的消息，才能改变该对象的拥有者个数。

[anObject release]; // anObject会失去一个拥有者

[anObject retain]; // anObject会得到一个拥有者

由此带来的问题：如果在修改某个指针变量的值，将其指向另一个对象前，忘记向该指针之前所指向的对象发送release消息，就一定会导致内存泄露。另一方面，向某个尚未收到retain消息的对象发送release消息，就会导致提前释放问题。使用手动引用计数管理内存时，程序员需要花费大量的时间来调试并修正这类问题。在大规模的项目中，因为手动引用计数而引发的问题会更复杂。

在只能使用手动引用计数的“黑暗时期”，Apple协助开发了一款名为Clang静态分析器（Clang static analyzer）的开源项目，并将其整合进了Xcode。简单地说，静态分析器可以分析代码并报告其能够发现的错误（第14章会对静态分析器做更多的介绍）。这些错误包括内存泄露和过早释放。有良好编程习惯的程序员会用静态分析器分析自己编写的代码，检查这类问题并做出相应的修正。

静态分析器的效果非常出色，以至于最后Apple考虑使用静态分析器来为代码自动插入所有的retain和release调用，并最终导致了ARC的诞生。ARC大大简化了内存管理工作。

使用手动引用计数管理内存时，还需要理解自动释放池（autorelease pool）。当对象收到autorelease消息时，某个自动释放池会成为该对象的临时拥有者。这样就可以解决这类问题：某个方法创建了一个新的对象，但是创建方又不需要成为该对象的拥有者。为了能返回新创建的对象，同时避免提前释放问题，就可以向新创建的对象发送autorelease消息。便捷方法借助自动释放池，将新创建的对象返回给调用方，又不产生内存管理问题。便捷方法的代码示例如下：

```
+ (BNRItem *)someItem

{

BNRItem *item = [[[BNRItem alloc] init] autorelease];

return item;

}
```

程序必须向对象发送release消息，才能减少相应对象的所有方个数。因此，针对someItem返回的BNRItem对象，调用方必须清楚自己的所有权。但是这种所有权关系并不是一目了然的，代码如下：

```
BNRItem *item = [BNRItem someItem]; // item变量是否拥有someItem方法返回的对象？
```

NSString *string = [item itemName]; // string变量是否拥有itemName方法的返回对象？

因此，如果某个对象不是通过alloc方法或copy方法创建的，就很有可能在返回给调用方前收到过autorelease消息。调用方要根据具体的情况，保留该对象的所有权。否则，在自动释放池被销毁时，程序就会释放这个对象。

使用ARC管理内存时，编译器会自动处理上述任务（在某些情况下，还能优化代码，彻底去除这部分代码）。此外，@autoreleasepool指令后面跟一对花括号，可以创建一个自动释放池。在@autoreleasepool的花括号中，如果某个方法返回一个新创建的对象，而该方法的方法名不包含alloc和init，那么这个新创建的对象通常会被放入相应的自动释放池。在应用执行完某个@autoreleasepool中的程序段后，该自动释放池中的所有对象都会失去一个拥有者，代码如下：

@autoreleasepool {

// 从someItem方法得到一个BNRItem对象，该方法的方法名没有包含alloc或copy

BNRItem *item = [BNRItem someItem];

} // 自动释放池被销毁，item变量所指向的对象会被释放

ios应用会自动创建一个默认的自动释放池，所以读者无须关心这个问题，但是了解@autoreleasepool的作用有益无害。

第四章 视图与视图层次结构

本章将介绍视图与视图层次结构的概念，并编写一个Hypnosister应用。应用只有一个界面，绘制了一系列同心圆（见图4-1）。

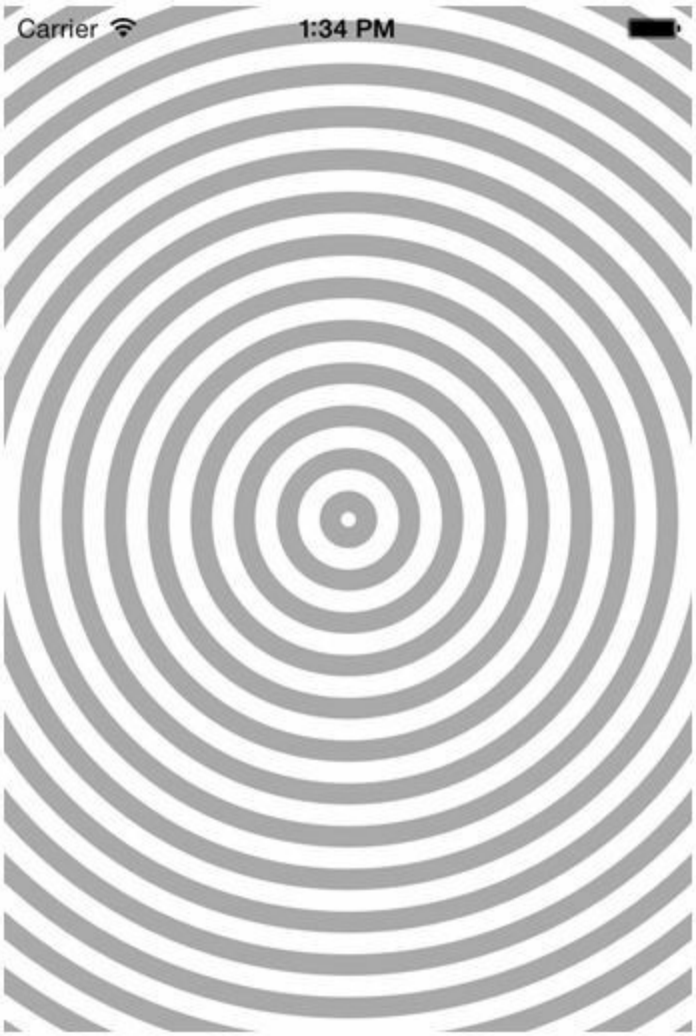


图4-1 Hypnosister

运行Xcode，选择File→New→Project...（键盘快捷键是Command-Shift-R）。在新出现的窗口中，选择左侧iOS部分的Application，然后选择右侧的Empty Application（空应用），单击Next按钮。

在文本框Product Name中填入Hypnosister，Class Prefix中填入BNR，在Devices弹出式菜单中选择iPhone。确保标题为Use Core Data的选择框未被选中（见图4-2）。

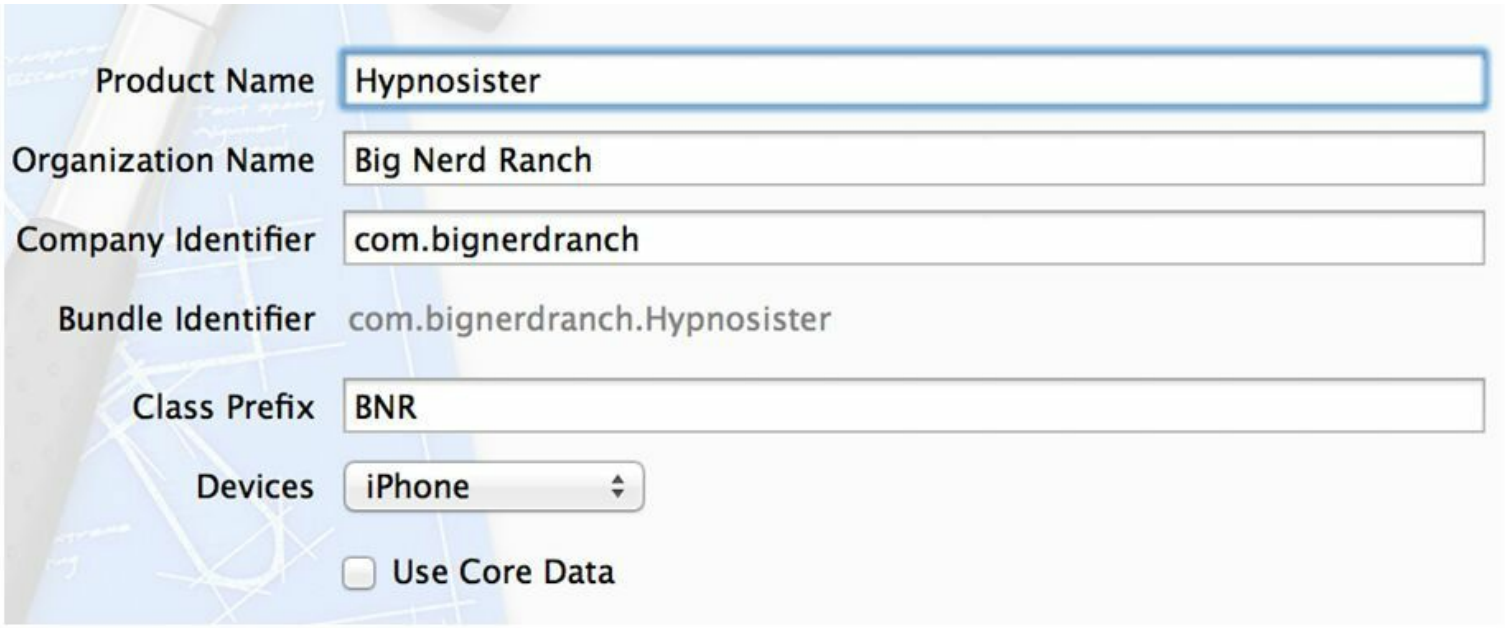


图4-2 设置Hypnosister项目

Hypnosister应用没有涉及用户交互，这样可以集中精力学习视图绘制的工作原理。首先介绍有关视图和视图层次结构的基本理论。

4.1 视图基础

- 视图是UIView对象，或是UIView子类对象。
- 视图知道如何绘制自己。
- 视图可以处理事件，例如触摸(touch)。
- 视图会按层次结构排列，位于视图层次结构顶端的是应用窗口。

第1章创建的Quiz应用包括4个视图对象：2个UIButton对象和2个UILabel对象。在Quiz应用中，视图是在Interface Builder中创建的，而Hypnosister应用将通过编写代码来创建和设置视图。

4.2 视图层次结构

任何一个应用都有且只有一个UIWindow对象。UIWindow对象就像一个容器，负责包含应用中的所有视图。应用需要在启动时创建并设置UIWindow对象，然后为其添加其他视图。

加入窗口的视图会成为该窗口的子视图（subview）。窗口的子视图还可以有自己的子视图，从而构成一个以UIWindow对象为根视图的视图层次结构（见图4-3）。

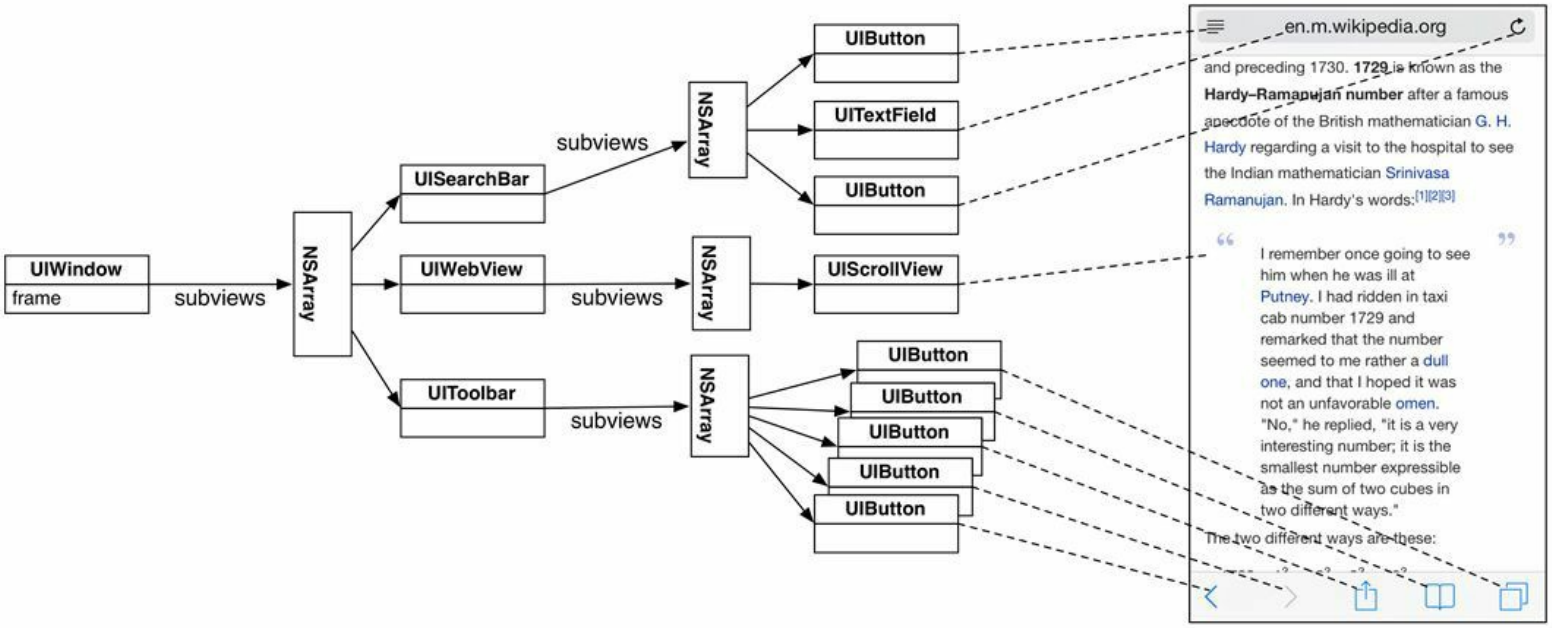


图4-3 视图层次结构构成应用的界面

视图层次结构形成之后，系统会将其绘制到屏幕上，绘制过程可以分为两步：

•层次结构中的每个视图（包括UIWindow对象）分别绘制自己。视图会将自己绘制到图层（layer）上，每个UIView对象都有一个layer属性，指向一个CALayer类的对象。读者可以将图层看成是一个位图图像（bitmap image）。

•所有视图的图层组合成一幅图像，绘制到屏幕上。

图4-4使用计算器应用的视图层次结构展示了上述绘制步骤。

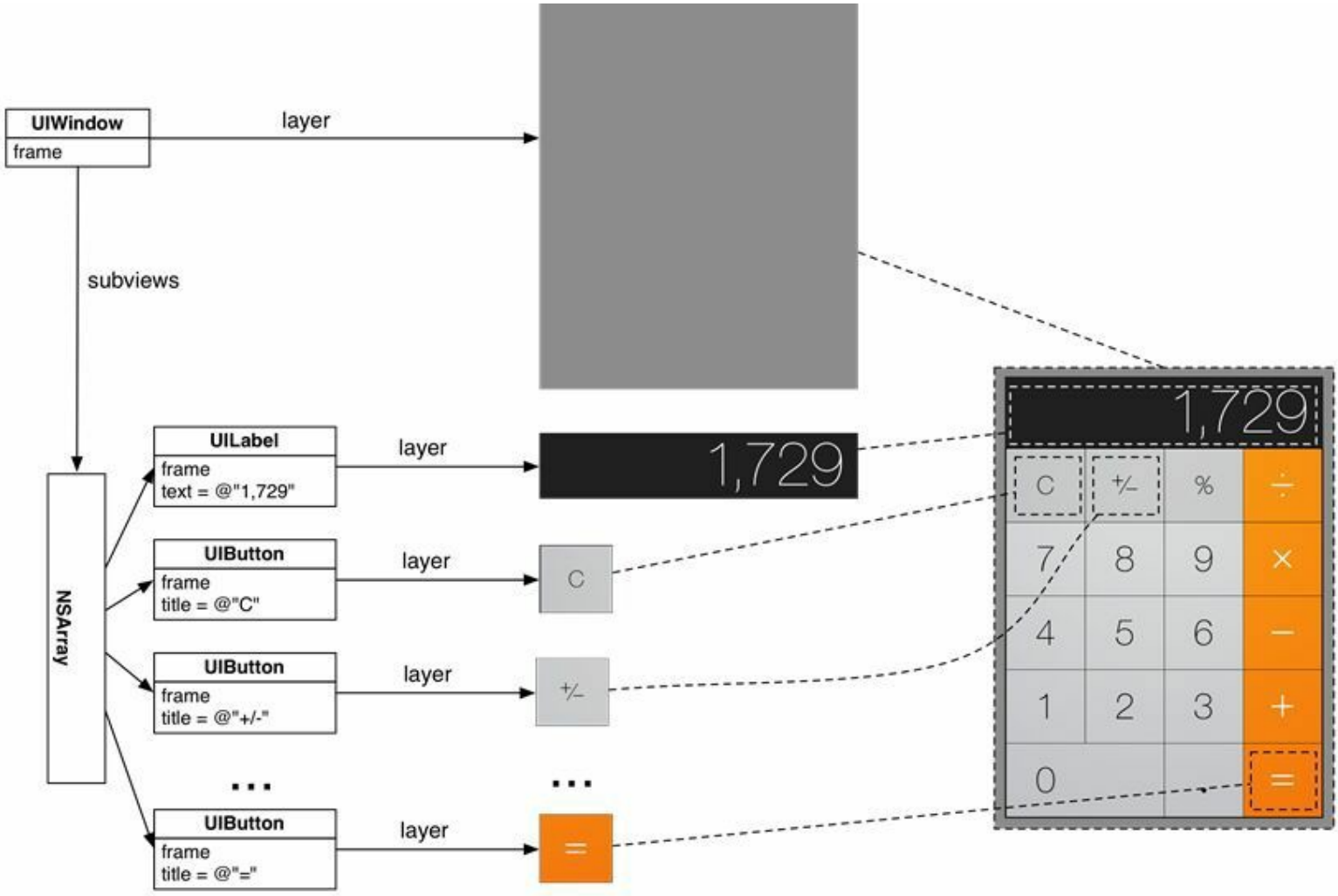


图4-4 视图首先分别绘制自己，然后组合起来绘制到屏幕上

部分视图类（例如UIButton和UILabel）已经实现了绘图功能。例如，在Quiz应用中，只需要创建UILabel对象并设置文本内容，UILabel对象就可以自动完成绘制文本的过程。

但是Apple并没有提供可以自动绘制同心圆的视图对象。因此，Hypnosister应用将创建一个UIView的子类，并编写自定义的绘图代码。

4.3 创建UIView子类

首先创建一个UIView子类，选择菜单File→New→File…（键盘快捷键是Command-N），再选择iOS下的Cocoa Touch，然后选择Objective-C class（见图4-5）。

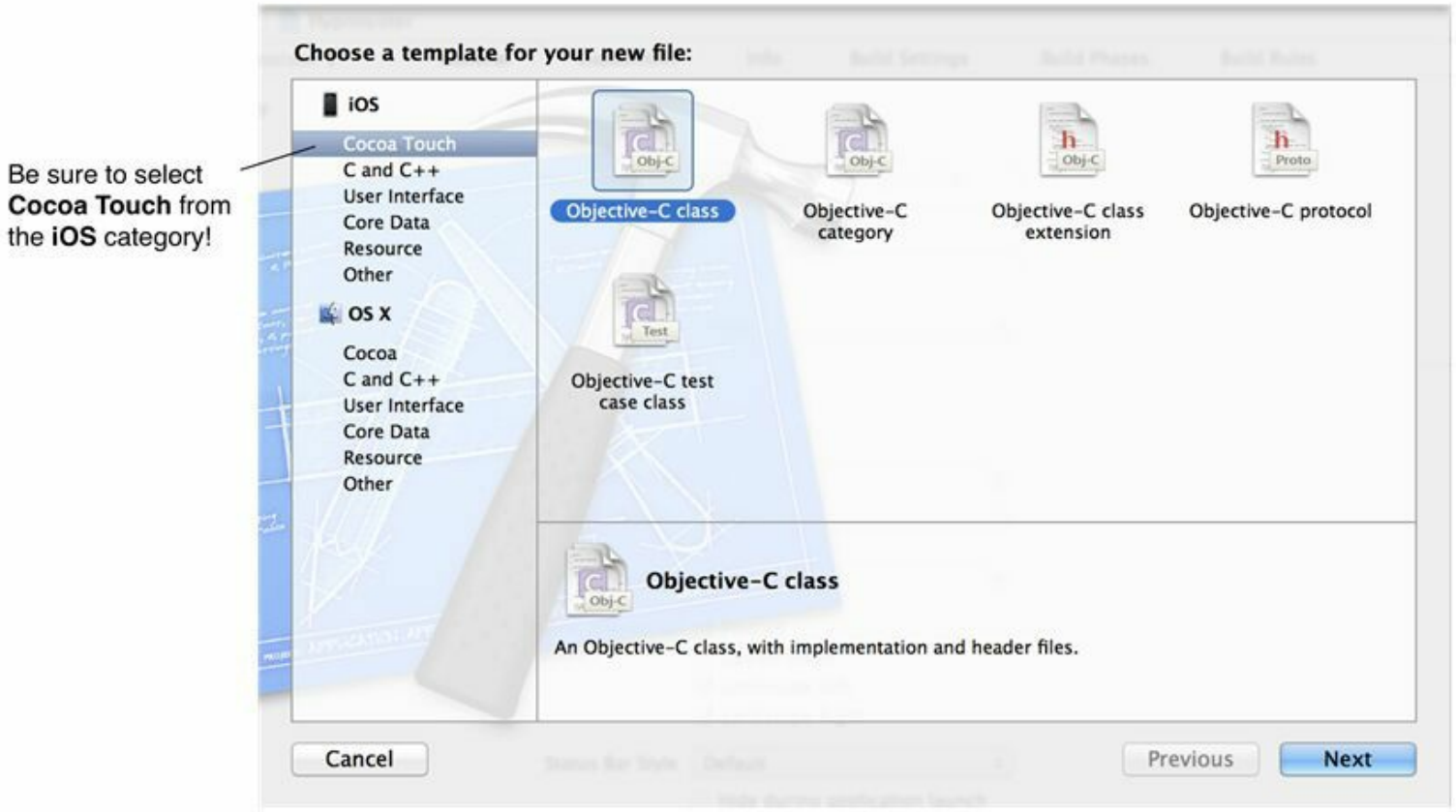


图4-5 创建新类

点击Next按钮，在Class文本框中输入BNRHypnosisView，在Subclass of下拉菜单中选择UIView（见图4-6）。

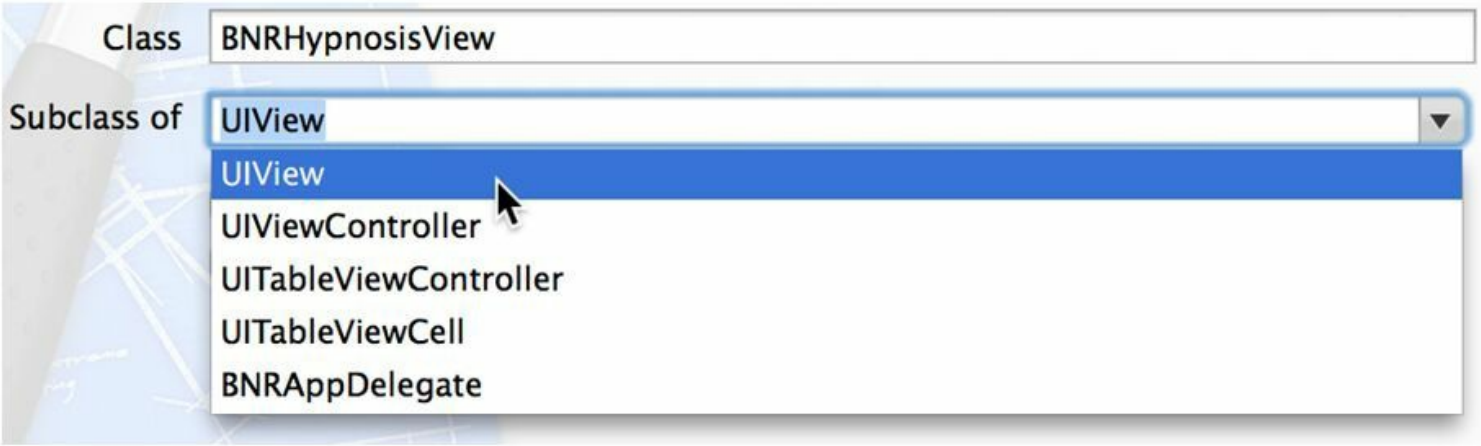


图4-6 设置新类的父类是UIView

单击Next按钮，确保选中了Targets中Hypnosister前的选择框。单击Create按钮。

在编写绘制同心圆的代码之前，首先需要了解如何通过编写代码创建和显示视图对象。为简单起见，本节不会在BNRHypnosisView中绘制同心圆，而是绘制一个红色背景的矩形，如图4-7所示。

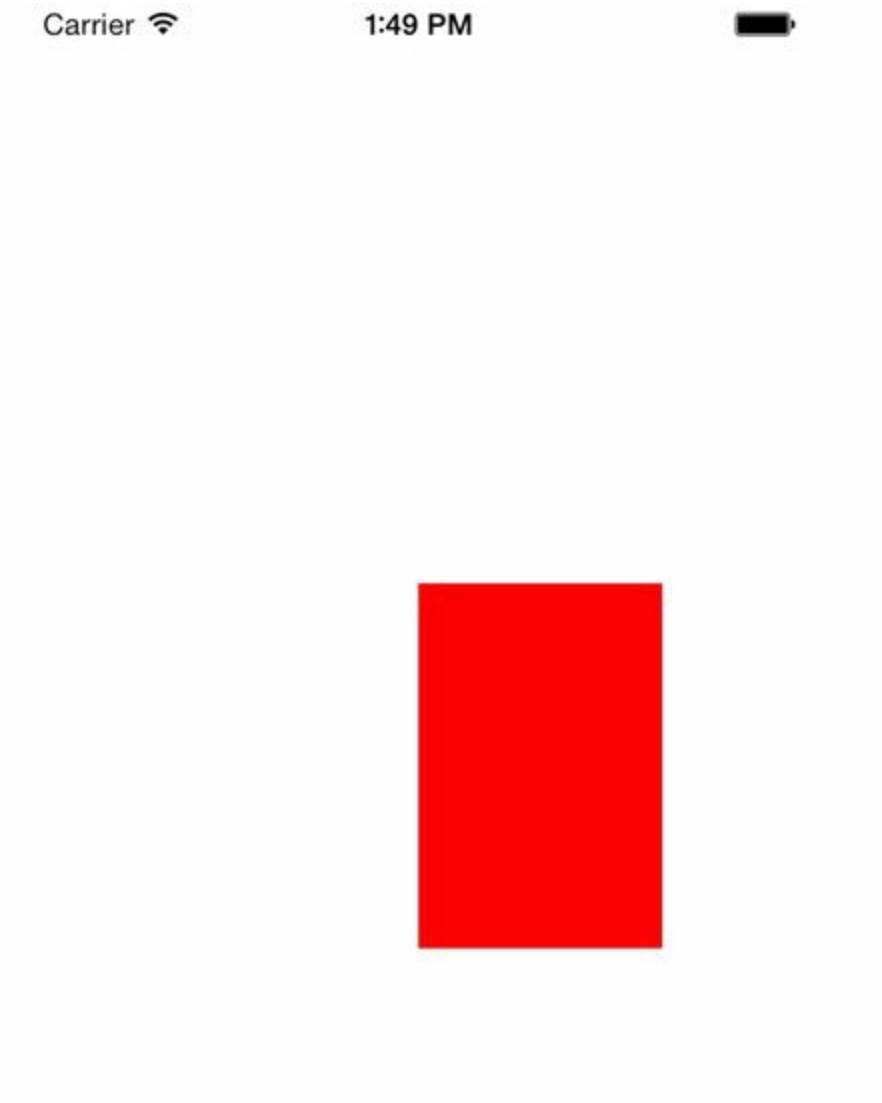


图4-7 BNRHypnosisView的第一个版本

视图及其frame属性

打开BNRHypnosisView.m文件。UIView子类的模板会自动生成两个方法，第一个是initWithFrame:，该方法是UIView的指定初始化方法，带有一个CGRect结构类型的参数，该参数会被赋给UIView的frame属性。

@property(nonatomic)CGRect frame;

视图的frame属性保存的是视图的大小和相对于父视图的位置。

CGRect结构包含另外两个结构：origin和size。origin的类型是CGPoint结构，该结构包含两个float类型的成员：x和y。size的类型是CGSize结构，该结构也包含两个float类型的成员：width和

height(见图4-8)。

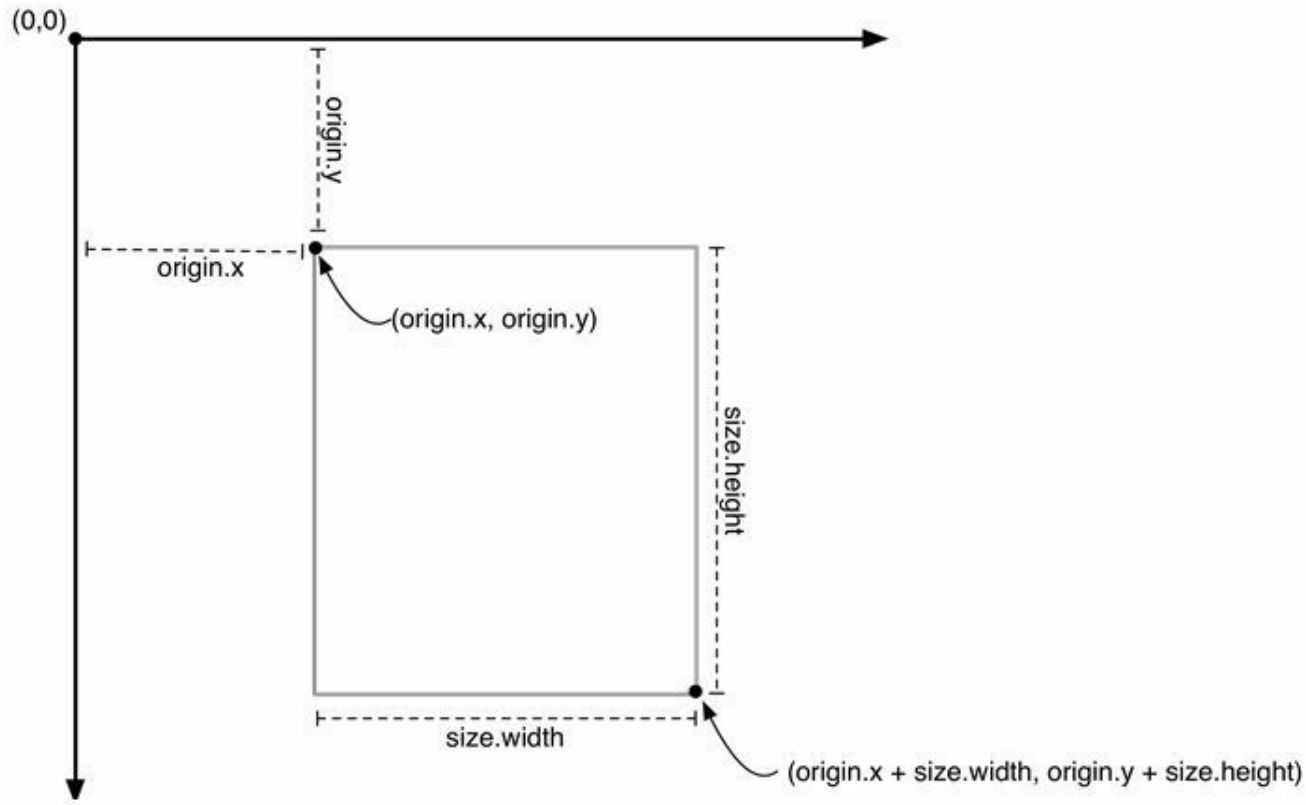


图4-8 CGRect

图由frame可知，视图对象的形状一定是矩形。

打开BNRAppDelegate.m，在文件顶部导入BNRHypnosisView类的头文件。

```
#import "BNRAppDelegate.h"

#import "BNRHypnosisView.h"

@implementation BNRAppDelegate
```

在BNRAppDelegate.m中找到application:didFinishLaunchingWithOptions:方法，在创建UIWindow对象的代码之后创建一个CGRect结构，然后使用该结构创建一个BNRHypnosisView对象，并设置backgroundColor属性为红色。最后，将BNRHypnosisView对象加入UIWindow对象，使其成为窗口的子视图。

```
- (BOOL)application:(UIApplication *)application

didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

{

self.window = [[UIWindow alloc] initWithFrame:
```

```
                [[UIScreen mainScreen] bounds]];

    CGRect firstFrame = CGRectMake(160, 240, 100, 150);

    BNRHypnosisView *firstView = [[BNRHypnosisView alloc]

                                  initWithFrame:firstFrame];

    firstView.backgroundColor = [UIColor redColor];

    [self.window addSubview:firstView];

    self.window.backgroundColor = [UIColor whiteColor];

    [self.window makeKeyAndVisible];

    return YES;

    }
```

结构不是Objective-C对象，因此不能向CGRect发送消息。可以使用CGRectMake函数创建一个CGRect。该函数需要传入origin.x、origin.y、size.width和size.height的值作为参数。CGRect占用的内存比大部分对象都小，因此不用传入指针——initWithFrame:方法中需要传入的参数类型是CGRect，而不是CGRect *。

为了设置backgroundColor属性，需要使用UIColor的类方法：redColor。这是一个简便方法，它返回一个表示红色的UIColor对象。UIColor中定义了一系列对应于常见颜色的简便方法，例如blueColor、blackColor和clearColor。

构建并运行应用，红色的矩形就是BNRHypnosisView对象。因为BNRHypnosisView对象的frame结构中的origin是(160, 240)，所以相对于UIWindow对象(BNRHypnosisView对象的父视图)的左上角，BNRHypnosisView对象的左上角会位于右侧160点、下方240点的位置。此外，因为这个frame结构中的size是(100, 150)，所以BNRHypnosisView对象的宽度为100点，高度为150点(见图4-9)。

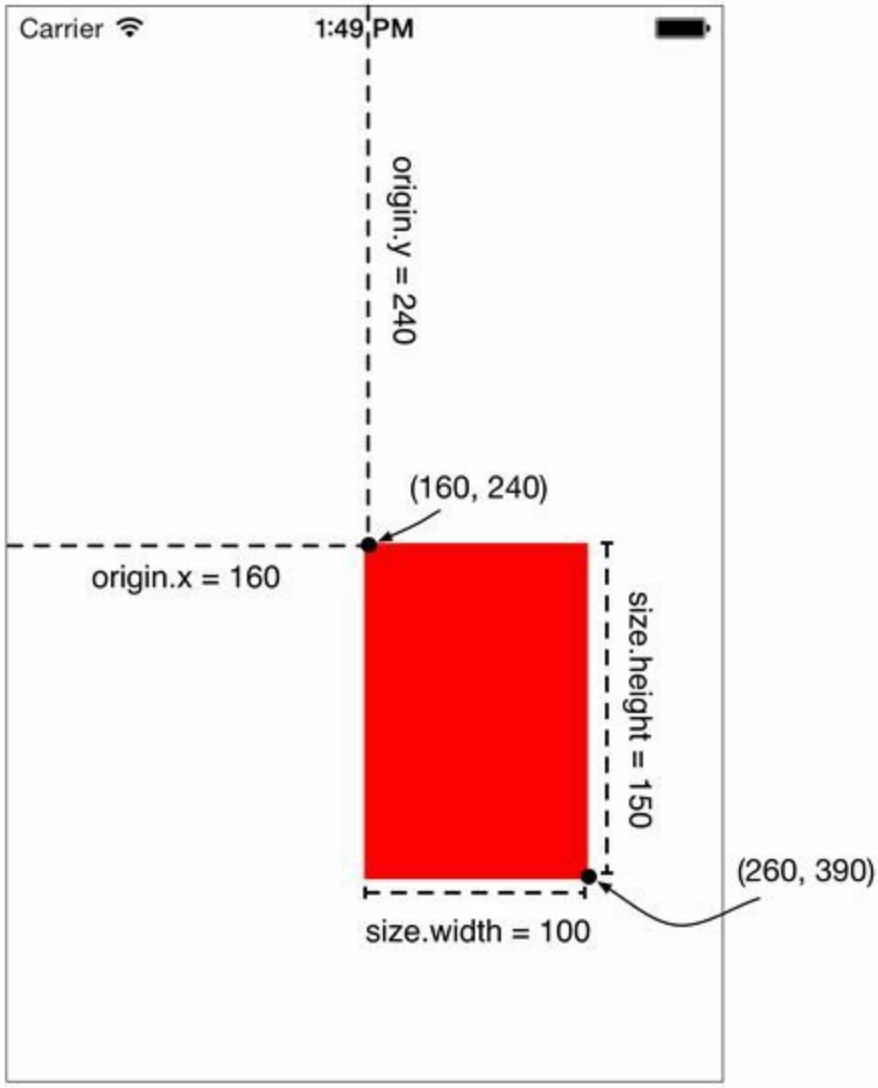


图4-9 向Hypnosister应用中添加一个BNRHypnosisView对象

请注意这些值的单位是点(points)，不是像素(pixels)。如果单位是像素，那么视图在Retina和非Retina显示屏上的大小无法保持一致。点的大小与设备分辨率相关，取决于屏幕以多少像素显示一个点：在Retina显示屏上，一个点是两个像素高度、两个像素宽度；非Retina显示屏则是一个像素高度、一个像素宽度。如果打印到纸上，一英寸是72点。为了保持应用界面在不同分辨率的屏幕上看起来一致，大小、位置、直线、曲线等都使用点作为单位。

在Xcode控制台中，请注意输出的警告信息："Application windows are expected to have a root view controller at the end of application launch.(应用启动完毕之后，需要为窗口设置一个根视图控制器。)"视图控制器可以用来管理应用中的多个视图对象，大部分iOS应用都有一个或多个视图控制器。Hypnosister是一个非常简单的应用，不需要视图控制器，所以读者可以忽略这行警告信息。第6章会介绍更多关于视图控制器的知识。

现在请看应用中已经创建好的视图层次结构(见图4-10)。

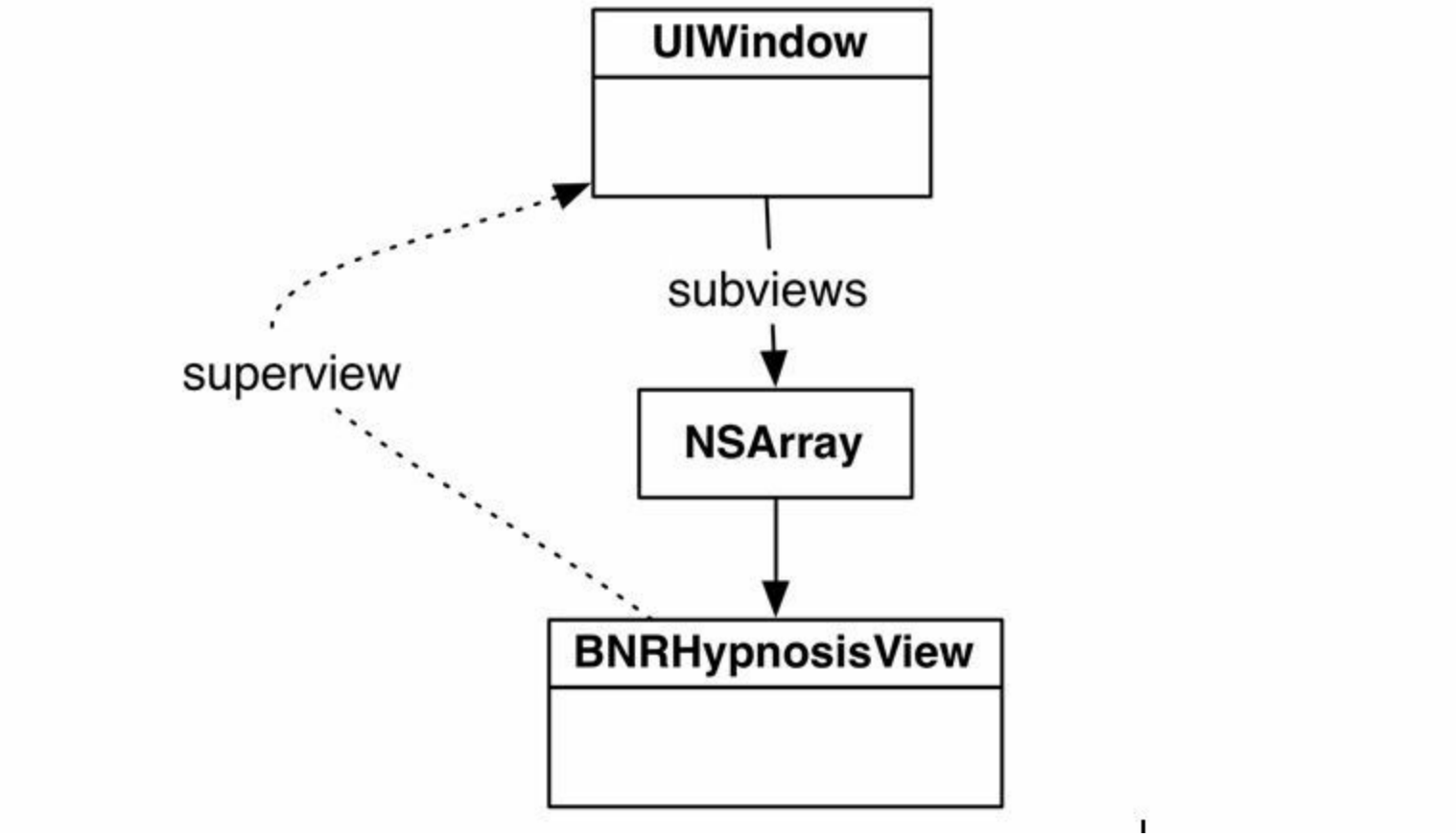


图4-10 UIWindow对象有一个子视图——BNRHypnosisView

每个UIView对象都有一个superview属性。将一个视图作为子视图加入另一个视图时，会自动创建相应的反向关联。在本例中，BNRHypnosisView对象的superview属性指向应用的UIWindow对象。（为了避免强引用循环，superview属性是弱引用。）

接下来尝试向视图层次结构中再加入一个视图。打开BNRAppDelegate.m，再创建一个BNRHypnosisView对象并设置不同的大小、位置和背景颜色：

```
...
[self.window addSubview:firstView];
CGRect secondFrame = CGRectMake(20, 30, 50, 50);
BNRHypnosisView *secondView = [[BNRHypnosisView alloc]
                               initWithFrame:secondFrame];
secondView.backgroundColor = [UIColor blueColor];
[self.window addSubview:secondView];
self.window.backgroundColor = [UIColor whiteColor];
...
```

再次构建并运行应用，除了之前添加的红色矩形外，窗口的左上角还有一个蓝色矩形。新的视图层次结构如图4-11所示。

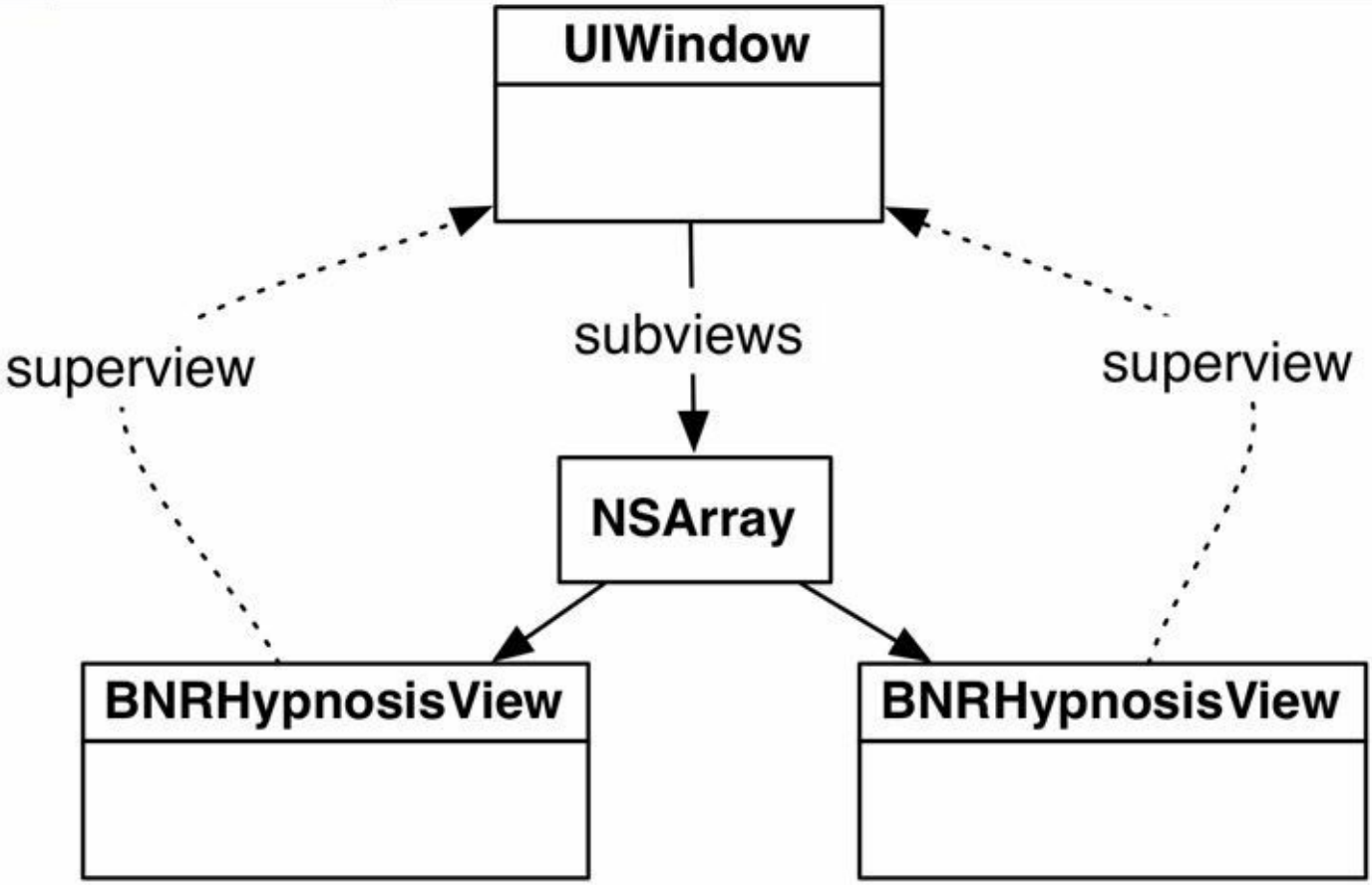


图4-11 UIWindow包含两个子视图

视图层次结构还可以包含两层以上的嵌套。在BNRAppDelegate.m中将第二个BNRHypnosisView对象加入第一个BNRHypnosisView对象，而不是UIWindow对象（见图4-12）。

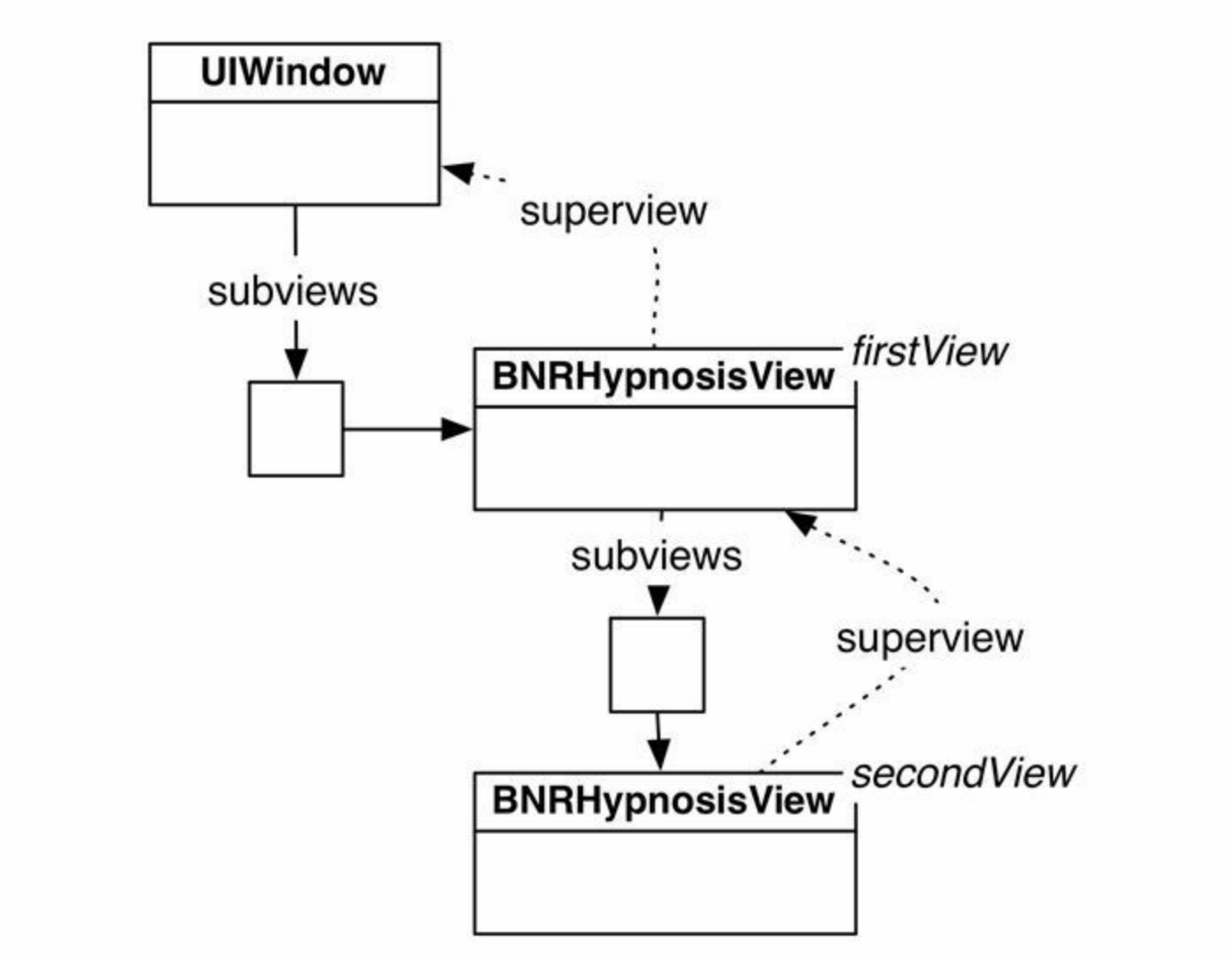


图4-12 secondView是firstView的子视图

代码如下:

```
...
BNR HypnosisView *secondView = [[BNRHypnosisView alloc]
                                initWithFrame:secondFrame];
secondView.backgroundColor = [UIColor blueColor];
[self.window addSubview:secondView];
[firstView addSubview:secondView];
...
```

构建并运行应用，请注意secondView在屏幕上的位置发生了变化。视图的frame所代表的位置是相对于其父视图的，所以secondView的左上角将以firstView的左上角位置为起点，偏移(20, 30)点。新的视图层次结构如图4-13所示。

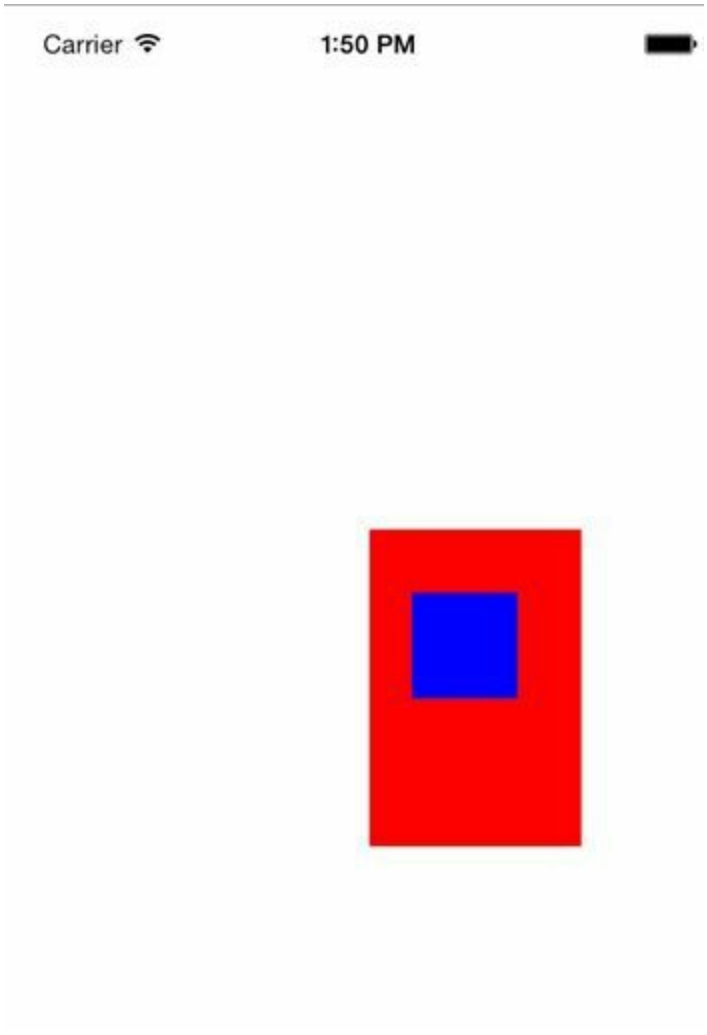


图4-13 Hypnosister应用新的视图层次结构

（如果读者觉得蓝色的BNRHypnosisView对象看起来比之前大，这是由于该视图位于更小的区域而产生的一种错觉。它的大小其实没有改变。）现在读者已经理解了视图层次结构，在进行下一步工作之前，请先删除secondView对象。

```
...
[self.window addSubview:firstView];
CGRect secondFrame = CGRectMake(20, 30, 50, 50);
BNRHypnosisView *secondView = [[BNRHypnosisView alloc]
                    initWithFrame:secondFrame];
secondView.backgroundColor = [UIColor blueColor];
[view addSubview:secondView];
self.window.backgroundColor = [UIColor whiteColor];
...
```

4.4 在drawRect:方法中自定义绘图

前面编写了一个名为BNRHypnosisView的UIView子类，创建了两个BNRHypnosisView对象，设置了它们的frame属性和backgroundColor属性，并将这两个对象加入了Hypnosister的视图层次结构。本节开始学习如何在drawRect:方法中为BNRHypnosisView编写自定义的绘图代码。

视图根据drawRect:方法将自己绘制到图层上。UIView的子类可以覆盖drawRect:，完成自定义的绘图任务。例如，UIButton的drawRect:方法默认会在frame表示的矩形区域中心画出一行浅蓝色的文字。

覆盖drawRect:后首先应该获取视图从UIView继承而来的bounds属性，该属性定义了一个矩形范围，表示视图的绘制区域。

视图在绘制自己时，会参考一个坐标系，bounds表示的矩形位于自己的坐标系，而frame表示的矩形位于父视图的坐标系，但是两个矩形的大小是相同的。

读者可能会疑惑，在表示的矩形大小相同的情况下，为什么除了frame属性之外，还要定义bounds属性？

frame和bounds表示的矩形用法不同。frame用于确定与视图层次结构中其他视图的相对位置，从而将自己的图层与其他视图的图层正确组合成屏幕上的图像。而bounds属性用于确定绘制区域，避免将自己绘制到图层边界之外（见图4-14）。

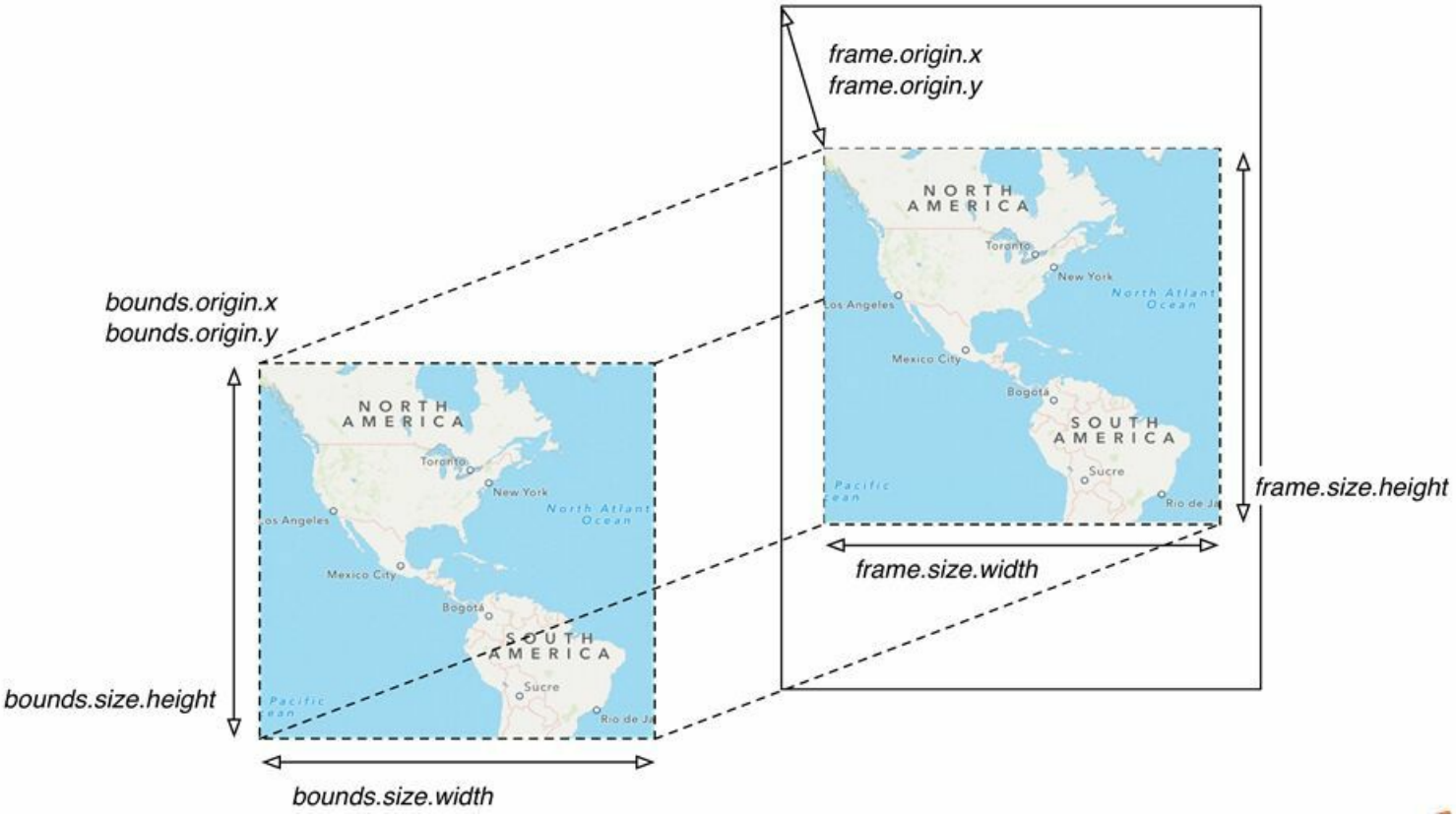


图4-14 bounds和frame

在BNRAppDelegate.m文件中，将UIWindow对象的bounds属性赋给firstView的frame属性，这样可以让firstView充满屏幕。

```
- (BOOL)application:(UIApplication *)application

didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

{

self.window = [[UIWindow alloc] initWithFrame:

        [[UIScreen mainScreen] bounds]];

// 在这里添加应用启动后的初始化代码

~~CGRect firstFrame = CGRectMake(160, 240, 100, 150);~~

CGRect firstFrame = self.window.bounds;

BNRHypnosisView *firstView = [[BNRHypnosisView alloc]

                initWithFrame:firstFrame];

[self.window addSubview:firstView];

self.window.backgroundColor = [UIColor whiteColor];

[self.window makeKeyAndVisible];

return YES;

}
```

构建并运行应用，屏幕上会显示一个充满屏幕的红色视图。

绘制圆形

接下来在BNRHypnosisView.m的drawRect:方法中添加绘图代码，画出一个尽可能大的圆形，但是不超过视图的绘制区域。

首先需要根据视图的bounds属性找到绘制区域的中心点，代码如下：

```
- (void)drawRect:(CGRect)rect

{
```

CGRect bounds = self.bounds;

// 根据bounds计算中心点

CGPoint center;

center.x = bounds.origin.x + bounds.size.width / 2.0;

center.y = bounds.origin.y + bounds.size.height / 2.0;

}

然后比较视图的宽和高，将较小值的二分之一设置为圆形的半径（使用较小值可以保证无论设备处于横握还是竖握模式都能画出不超过绘制区域的圆形），代码如下：

- (void)drawRect:(CGRect)rect

{

CGRect bounds = self.bounds;

// 根据bounds计算中心点

CGPoint center;

center.x = bounds.origin.x + bounds.size.width / 2.0;

center.y = bounds.origin.y + bounds.size.height / 2.0;

// 根据视图宽和高中的较小值计算圆形的半径

float radius = (MIN(bounds.size.width, bounds.size.height) / 2.0);

}

UIBezierPath

现在可以开始使用UIBezierPath类绘制圆形了。UIBezierPath用来绘制直线或曲线，从而组成各种形状，例如圆形。

首先需要创建一个UIBezierPath对象，代码如下：

- (void)drawRect:(CGRect)rect

{

```
...

// 根据视图宽和高中的较小值计算圆形的半径

float radius = (MIN(bounds.size.width, bounds.size.height) / 2.0);

UIBezierPath *path = [[UIBezierPath alloc] init];

}
```

接下来定义UIBezierPath对象需要绘制的路径。如何定义一个可以绘制圆形的路径？解决问题的最佳方式就是查看Apple提供的开发者文档。文档中详细介绍了UIBezierPath的使用方法。

使用开发者文档

在Xcode菜单中选择Help→Documentation and API Reference（文档和API参考手册）。也可以使用键盘快捷键Option-Command-?（其实还需要按住Shift，以便选择“？”）。

（查看文档时，Xcode可能会从Apple服务器上获取最新内容，有时会要求读者输入Apple ID和密码。）

这时Xcode会打开文档浏览器，在搜索框中输入UIBezierPath，然后在搜索结果中选择UIBezierPath Class Reference（UIBezierPath参考手册），如图4-15所示。

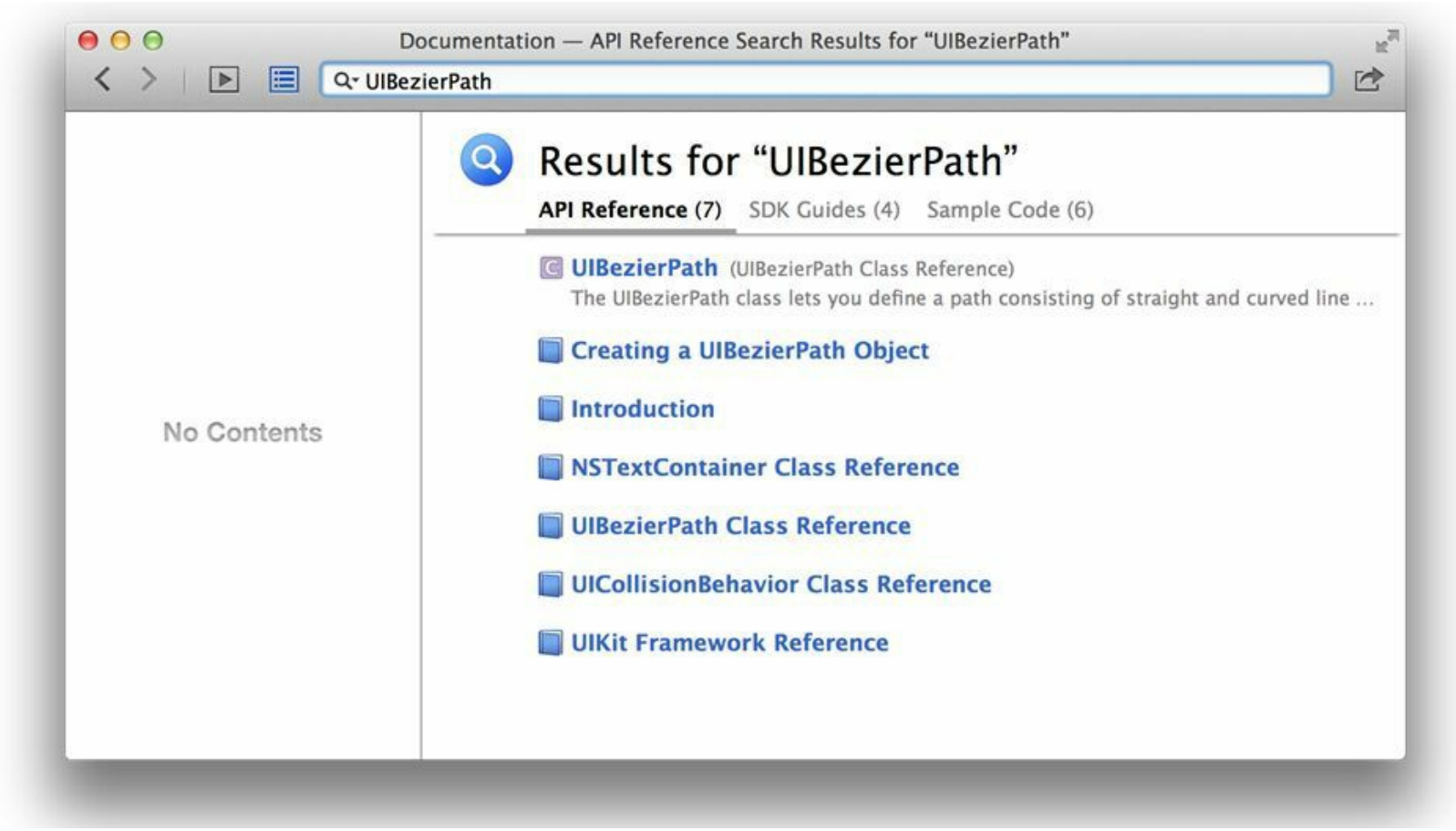


图4-15 在文档中搜索UIBezierPath

UIBezierPath参考手册打开时会显示类的概览页面，但是关于定义绘制路径的内容位于Task部分。Task部分位于参考手册左边的内容列表中，（如果没有看到内容列表，请点击浏览器左上角的按钮。）可以根据需要实现的功能在Task部分查找对应的方法。

第一个Task是Creating a UIBezierPath Object（创建UIBezierPath对象），之前的代码中已经完成了。第二个是Constructing a Path（构建路径），选择该Task，查看UIBezierPath中与之相关的一系列方法（见图4-16）。

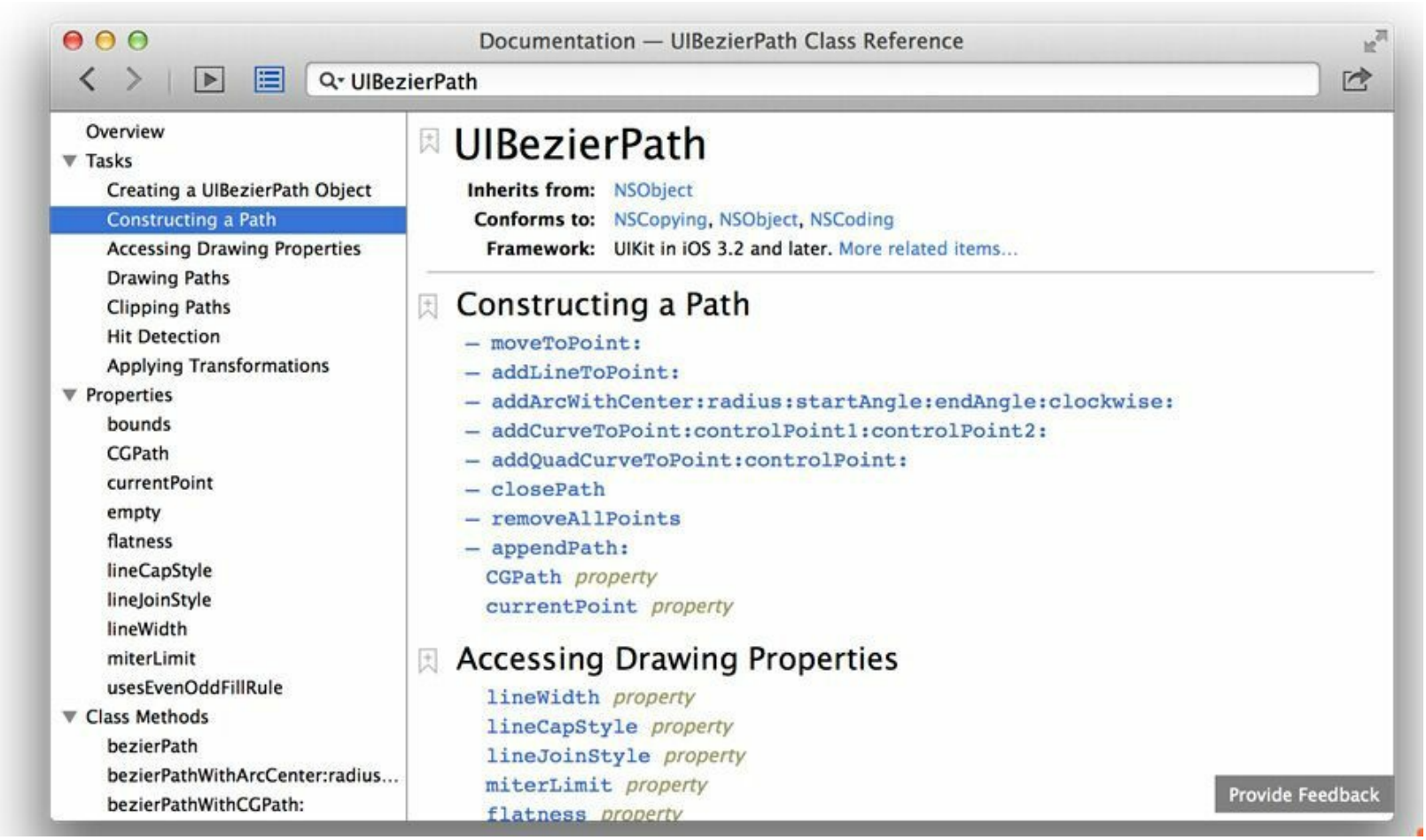


图4-16 与构建路径有关的方法

请注意addArcWithCenter:radius:startAngle:endAngle:clockwise:方法，可以根据角度和半径定义弧形路径的方法定义圆形路径。点击该方法查看参数使用说明，之前已经计算出了圆形的圆心和半径，圆形的起止角度分别是0和M_PI * 2（startAngle和endAngle参数值的单位是弧度）。

（如果读者对弧度和角度的转换有疑问，可以先简单地传入这两个值，或者在方法文档中找到Discussion部分，点击Figure 1链接，查看圆形中有关弧度的示意图。）

最后，由于我们绘制的是一个整圆，顺时针还是逆时针都是可以的，因此为clockwise传入YES或NO。

在BNRHypnosisView.m中，向UIBezierPath对象发送消息，定义绘制路径：

```
- (void)drawRect:(CGRect)rect

{

CGRect bounds = self.bounds;

// 根据bounds计算中心点

CGPoint center;
```

```
center.x = bounds.origin.x + bounds.size.width / 2.0;

center.y = bounds.origin.y + bounds.size.height / 2.0;

// 根据视图宽和高中的较小值计算圆形的半径

float radius = (MIN(bounds.size.width, bounds.size.height) / 2.0);

UIBezierPath *path = [[UIBezierPath alloc] init];

// 以中心点为圆心、radius的值为半径，定义一个0到M_PI * 2.0弧度的路径(整圆)

[path addArcWithCenter:center

radius:radius

        startAngle:0.0

endAngle:M_PI * 2.0

    clockwise:YES];

}
```

路径已经定义好了，但是只定义路径不会进行实际的绘制。回到UIBezierPath参考手册，在Task中选择Drawing Paths（绘制路径）。在可以绘制路径的方法中，选择stroke。（其他绘制方法会为图形填充颜色或者需要其他参数，例如CGBlendMode。）

现在向UIBezierPath对象发送消息，绘制之前定义的路径，代码如下：

```
- (void)drawRect:(CGRect)rect

{

...

UIBezierPath *path = [[UIBezierPath alloc] init];

// 以中心点为圆心、radius的值为半径，定义一个0到M_PI * 2.0弧度的路径(整圆)

[path addArcWithCenter:center

radius:radius

        startAngle:0.0

endAngle:M_PI * 2.0
```

```
        clockwise:YES];

    // 绘制路径！

    [path stroke];

    }
```

构建并运行应用，可以看到一个由黑色细线构成的圆形，其直径和屏幕宽度相同（如果设备处于横握状态，那么圆形的直径和屏幕高度相同），如图4-17所示。

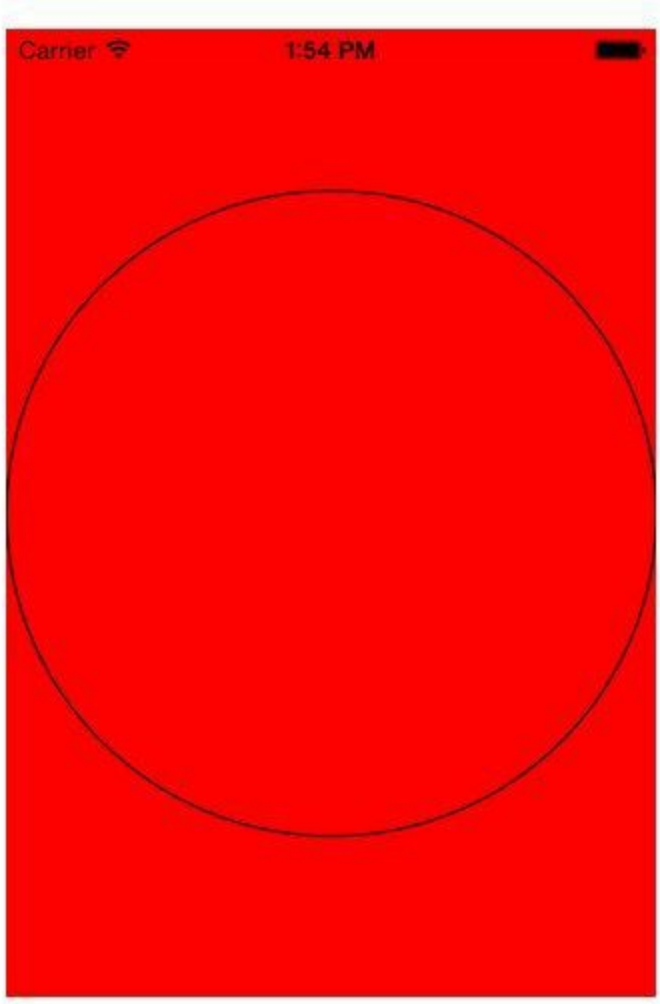


图4-17 BNRHypnosisView中画出了一个圆形

根据Hypnosister应用最初的设计方案，构成圆形的轮廓线应该比现在更粗，同时颜色是浅灰色而不是黑色。

为了改变线条的粗细和颜色，请回到UIBezierPath参考手册，在内容列表中找到Properties（属性）部分。在属性列表中找到lineWidth属性，该属性是CGFloat类型，默认值为1.0。

在BNRHypnosisView.m中，设置线条的宽度为10点，代码如下：

```
- (void)drawRect:(CGRect)rect
```

```
{
...
// 以中心点为圆心、radius的值为半径，定义一个0到M_PI * 2.0弧度的路径(整圆)
[path addArcWithCenter:center
radius:radius
        startAngle:0.0
endAngle:M_PI * 2.0
    clockwise:YES];
// 设置线条宽度为10点
path.lineWidth = 10;
// 绘制路径！
[path stroke];
}
```

构建并运行应用，现在圆形的轮廓线应该比之前更粗。

UIBezierPath中没有设置线条颜色的属性，但是UIBezierPath参考手册中的Overview部分介绍了设置线条颜色的方法。点击内容列表中的Overview，在第五段（随着Apple修订文档可能会有变动）中有一句话："You set the stroke and fill color using the UIColor class.（请您使用UIColor类设置线条颜色和填充颜色。）"

可以点击UIColor链接进入UIColor参考手册，在UIColor的Task部分，选择Drawing Operations，查看与绘制操作相关的方法。可以使用set或setStroke方法设置线条颜色，请读者选择用法更加明确的setStroke方法。

setStroke方法是一个实例方法，因此需要创建一个UIColor对象。前面提到过，UIColor有一系列对应于常见颜色的简便方法，可以在UIColor参考手册中的Class Methods部分找到这些方法，其中包括lightGrayColor。

现在可以设置线条颜色了，在BNRHypnosisView.m中，创建一个表示浅灰色的UIColor对象并向其发送setStroke消息，这样就可以画出一个浅灰色轮廓的圆形。

```
- (void)drawRect:(CGRect)rect
{
```

```
...
// 设置线条宽度为10点
path.lineWidth = 10;
// 设置绘制颜色为浅灰色
[[UIColor lightGrayColor] setStroke];
// 绘制路径！
[path stroke];
}
```

构建并运行应用，可以看到圆形的轮廓线已经变成浅灰色了。

读者可能已经注意到，视图的backgroundColor属性不会受drawRect:中代码的影响，通常应该将重写了drawRect:方法的视图的背景颜色设置为透明（对应于clearColor），这样可以让视图只显示drawRect:方法中绘制的内容。

在BNRAppDelegate.m中，删除设置视图背景颜色的代码：

```
BNRHypnosisView *firstView = [[BNRHypnosisView alloc]
                              initWithFrame:firstFrame];
~~firstView.backgroundColor = [UIColor redColor];~~
[self.window addSubview:view];
```

然后，在BNRHypnosisView.m的initWithFrame:方法中，设置BNRHypnosisView对象的背景颜色为透明：

```
- (instancetype)initWithFrame:(CGRect)frame
{
self = [super initWithFrame:frame];
if (self) {
    // 设置BNRHypnosisView对象的背景颜色为透明
    self.backgroundColor = [UIColor clearColor];
```

```
}

return self;

}
```

构建并运行应用，可以看到一个透明背景的圆形，如图4-18所示。

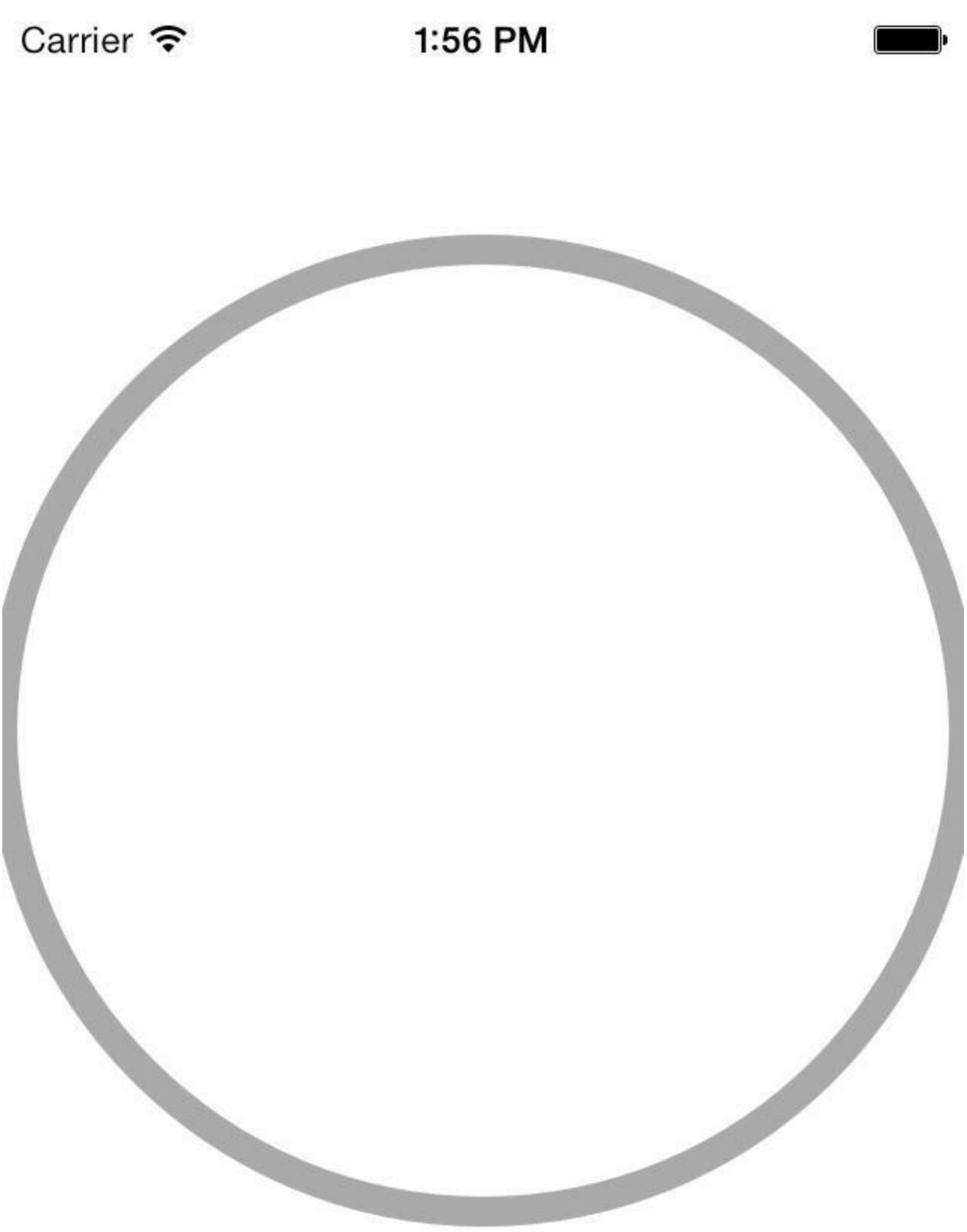


图4-18 透明背景的圆形

绘制同心圆

在BNRHypnosisView中绘制多个同心圆有两个方法，第一个方法是创建多个UIBezierPath对象，每个对象代表一个圆形；第二个方法是使用一个UIBezierPath对象绘制多个圆形，为每个圆形定义一个绘制路径。显然，第二个方法更好，只创建一个UIBezierPath对象可以减少内存占用。

首先需要确定最外层圆形的直径，然后从这个直径递减，确定其他圆形的直径。请注意直径必须大于零。

这里使用视图的对角线作为最外层圆形的直径，使最外层圆形成为视图的外接圆——只能在视图的四个角上看到该圆形的浅灰色轮廓。

在BNRHypnosisView.m中，删除绘制单个圆形的代码，改为绘制多个同心圆：

```
- (void)drawRect:(CGRect)rect

{

CGRect bounds = self.bounds;

// 根据bounds计算中心点

CGPoint center;

center.x = bounds.origin.x + bounds.size.width / 2.0;

center.y = bounds.origin.y + bounds.size.height / 2.0;

// 根据视图宽和高中的较小值计算圆形的半径

float radius = (MIN(bounds.size.width, bounds.size.height) / 2.0);

// 使最外层圆形成为视图的外接圆

float maxRadius = hypot(bounds.size.width, bounds.size.height) / 2.0;

UIBezierPath *path = [[UIBezierPath alloc] init];

// 以中心点为圆心，radius的值为半径，

 // 定义一个0到M_PI * 2.0弧度的路径(整圆)

[path addArcWithCenter:center
```

```
        radius:radius
    startAngle:0.0
        endAngle:M_PI * 2.0
    clockwise:YES];
for (float currentRadius = maxRadius; currentRadius > 0;
currentRadius -= 20) {
[path addArcWithCenter:center
radius:currentRadius // 半径为currentRadius！
            startAngle:0.0
            endAngle:M_PI * 2.0
            >clockwise:YES];
}
// 设置线条宽度为10点
path.lineWidth = 10;
// 设置绘制颜色为浅灰色
[[UIColor lightGrayColor] setStroke];
// 绘制路径！
[path stroke];
}
```

构建并运行应用，虽然BNRHypnosisView画出了一系列同心圆，但是屏幕右边多出了一条奇怪的直线，如图4-19所示。

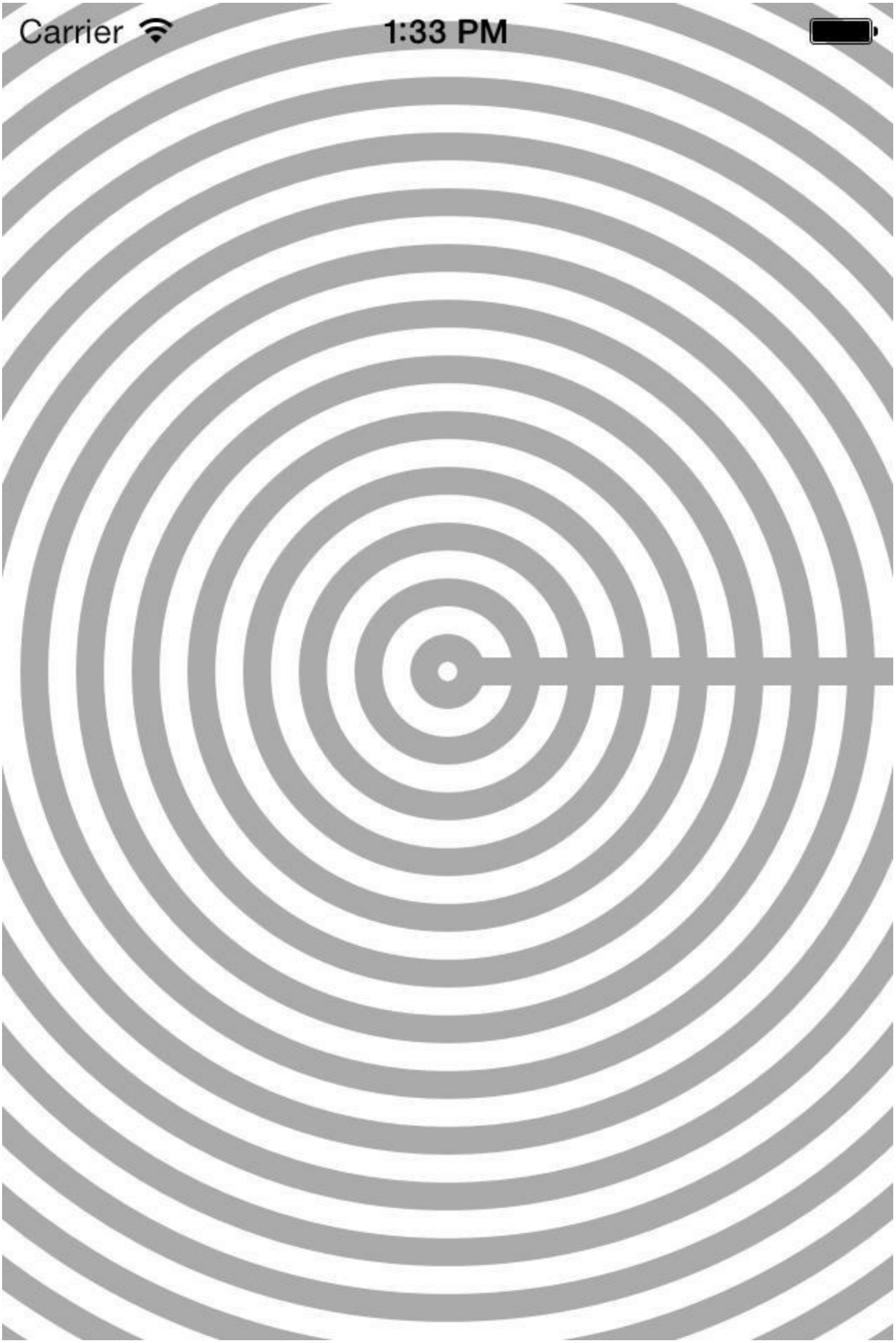


图4-19 BNRHypnosisView绘制了不正确的同心圆

这是因为单个UIBezierPath对象将多个路径（每个路径可以画出一个圆形）连接起来，形成了一个完整的路径。可以将UIBezierPath对象想象为一支在纸上画画的铅笔——在绘制完某个圆后去绘制另一个圆时，铅笔并没有抬起，仍然停留在纸上，因此会继续留下笔迹。正确的做法是每次绘制新的圆形之前，必须抬起笔。

在drawRect:方法的for循环中，当绘制下一个圆形时，抬起笔并移动到正确的位置：

```
- (void)drawRect:(CGRect)rect

{

CGRect bounds = self.bounds;

// 根据bounds计算中心点

CGPoint center;

center.x = bounds.origin.x + bounds.size.width / 2.0;

center.y = bounds.origin.y + bounds.size.height / 2.0;

// 使最外层圆形成为视图的外接圆

float maxRadius = hypot(bounds.size.width, bounds.size.height) / 2.0;

UIBezierPath *path = [[UIBezierPath alloc] init];

for (float currentRadius = maxRadius; currentRadius > 0; currentRadius -= 20) {

[path moveToPoint:CGPointMake(center.x + currentRadius, center.y)];

[path addArcWithCenter:center

                radius:currentRadius // 半径为currentRadius！

                startAngle:0.0

                endAngle:M_PI * 2.0

                clockwise:YES];

}

// 设置线条宽度为10点

path.lineWidth = 10.0;

// 绘制路径
```

```
[path stroke];

}
```

构建并运行应用，现在绘制的同心圆完全正确，如图4-20所示。

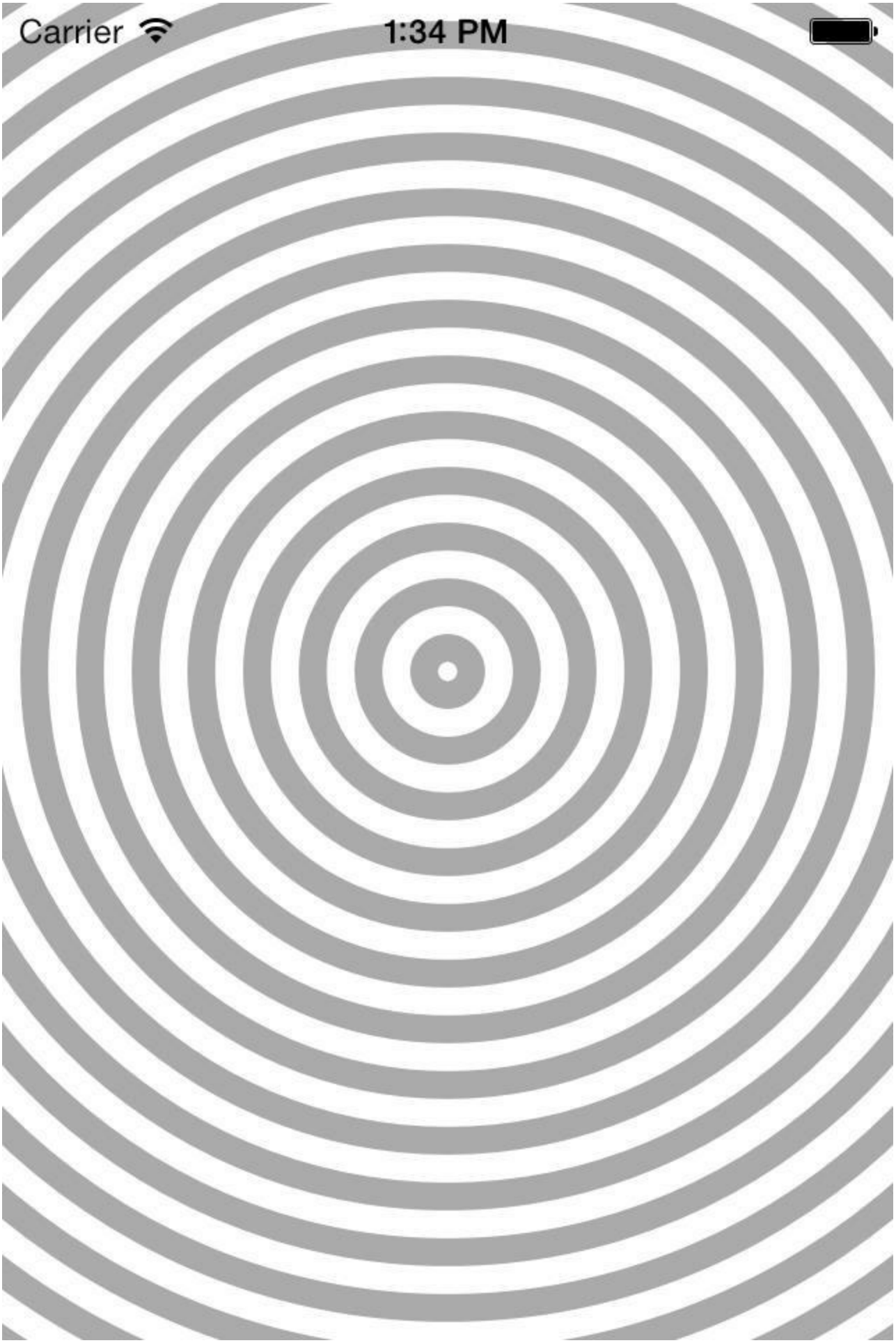


图4-20 绘制同心圆

Hypnosister只使用了UIBezierPath的少数功能，读者可以查看UIBezierPath参考手册完成本章后面的练习，学习使用圆弧、直线和曲线组合起来画出更多有创意的图案。

4.5 关于开发者文档

API参考手册中包含对类的使用方法的说明，开发者经常在文档中查阅该手册。但是，文档中除了API参考手册之外还包括以下内容：

SDK Guides（SDK 开发指南）	根据主题组织，而不是类或者方法，介绍了有关 Objective-C 和 iOS 开发的高级主题
Sample Code（示例代码）	完整的示例项目，演示了类的 Apple 官方使用方法

使用Apple的开发者文档是每一位iOS程序员的必修课。读者在阅读本书的过程中遇到新的类和方法时，请耐心查阅参考文档，看看除了书中介绍的之外还有哪些功能，也可以在iOS Developer Library（iOS开发者文档库）的网站中浏览SDK指南或下载感兴趣的示例代码：developer.apple.com/library。

4.6 初级练习：绘制图像

从文件系统中加载一个图像文件，并将其绘制到同心圆上，如图4-21所示。

图4-21 绘制图像

首先需要找到一个图像文件，最好是部分透明的PNG格式的图像文件。（本书源代码压缩包中有一个logo.png文件正好可以使用。）然后将该文件拖入Xcode项目中。

从该文件创建一个UIImage对象只需要一行代码：

```
UIImage *logoImage = [UIImage imageNamed:@"logo.png"];
```

在drawRect:方法中将图像绘制到视图上也只需要一行代码：

```
[logoImage drawInRect:someRect];
```

4.7 深入学习：Core Graphics

UIImage、UIBezierPath和NSString都提供了至少一种用于在drawRect:中绘图的方法，这些绘图方法会在drawRect:执行时分别将图像、图形和文本绘制到视图的图层上。

使用这些方法进行绘图既简单又方便，但是绘制方法内部封装了很多复杂的绘图代码。

无论绘制JPEG、PDF还是视图的图层，都是由Core Graphics框架完成的。本章使用的UIBezierPath，其实是将Core Graphics代码封装在一系列方法中，以方便开发者调用，降低了绘图难度。为了真正了解绘图的过程与原理，必须深入学习Core Graphics是如何工作的。

Core Hraphics是一套提供2D绘图功能的C语言API，使用C结构和C函数模拟了一套面向对象的编程机制，并没有Objective-C对象和方法。Core Graphics中最重要的“对象”是图形上下文（graphics context），图形上下文是CGContextRef的“对象”，负责存储绘画状态（例如画笔颜色和线条粗细）和绘制内容所处的内存空间。

视图的drawRect:方法在执行之前，系统首先为视图的图层创建一个图形上下文，然后为绘画状态设置一些默认参数。drawRect:方法开始执行时，随着图形上下文不断执行绘图操作，图层上的内容也会随之改变。drawRect:执行完毕后，系统会将图层与其他图层一起组合成完整的图像并显示在屏幕上。

参与绘图操作的类都定义了改变绘画状态和执行绘图操作的方法，这些方法其实调用了对应的Core Graphics函数。例如，向UIColor对象发送setStroke消息时，会调用Core Graphics中的CGContextSetRGBStrokeColor函数改变当前图形上下文中的画笔颜色。

以下两段代码的效果是相同的：

```
[[UIColor colorWithRed:1.0 green:0.0 blue:1.0 alpha:1.0] setStroke];

UIBezierPath *path = [UIBezierPath bezierPath];

[path moveToPoint:a];

[path addLineToPoint:b];

[path stroke];
```

直接使用Core Graphics函数完成相同的绘图操作：

```
CGContextSetRGBStrokeColor(currentContext, 1, 0, 0, 1);

CGMutablePathRef path = CGPathCreateMutable();

CGPathMoveToPoint(path, NULL, a.x, a.y);

CGPathAddLineToPoint(path, NULL, b.x, b.y);
```

```
CGContextAddPath(currentContext, path);

CGContextStrokePath(currentContext);

CGPathRelease(path);
```

像CGContextSetRGBStrokeColor这类在图形上下文中执行绘图操作的函数，第一个参数需要传入指向图形上下文的指针。可以在drawRect:方法中调用UIGraphics- GetCurrentContext函数获取当前图形上下文。视图的当前图形上下文是在drawRect:方法执行之前创建的。

```
- (void)drawRect:(CGRect)rect

{

CGContextRef currentContext = UIGraphicsGetCurrentContext();

CGContextSetRGBStrokeColor(currentContext, 1, 0, 0, 1);

CGMutablePathRef path = CGPathCreateMutable();

CGPathMoveToPoint(path, NULL, a.x, a.y);

CGPathAddLineToPoint(path, NULL, b.x, b.y);

CGContextAddPath(currentContext, path);

CGContextStrokePath(currentContext);

CGPathRelease(path);

CGContextSetStrokeColorWithColor(currentContext, color);

}
```

UIBezierPath和UIColor的所有绘图功能都可以通过直接调用Core Graphics函数完成。实际上，UIBezierPath和UIColor在Core Graphics中有对应的C结构：CGMutablePathRef 和 CGColorRef。通常情况下，使用Objective-C类更加方便。

但是，有些功能只能使用Core Graphics完成，例如绘制渐变。Core Graphics中的结构和函数都具有CG前缀，如果遇到无法使用Objective-C类完成的绘图功能，可以在文档中查阅带有CG前缀的结构和函数，直接使用Core Graphics。

或许读者会感到疑惑，为什么很多Core Graphics类型都带有Ref后缀。带有Ref后缀的类型是Core Graphics中用来模拟面向对象机制的C结构。Core Graphics“对象”与Objective-C对象都是在堆上分配内存，因此创建一个Core Graphics“对象”时，同样会返回一个指向对象内存地址的指针。

使用这种分配方式的C结构都有一个用来表示结构指针(结构名后加一个“*”)的类型定义(type definition)。例如，CGColor结构(不会直接使用的类型)有一个表示CGColor *的类型定义——CGColorRef(应该使用的类型)。使用这种类型定义是为了区分指针变量，方便开发者判断指针变量是指向C结构还是可以接收消息的Objective-C对象。

相反，部分类型没有结构指针，因此类型名称不带Ref后缀。例如CGRect和CGPoint。这些类型的数据结构简单，可以直接在栈上分配，因此不需要使用结构指针。

带有Ref后缀的类型的对象可能具有指向其他Core Graphics“对象”的强引用指针，并成为这些“对象”的拥有者。但是ARC无法识别这类强引用和“对象”所有权，必须在使用完之后手动释放。规则是，如果使用名称中带有create或者copy的函数创建了一个Core Graphics“对象”，就必须调用对应的Release函数并传入该对象指针。

最后，Mac开发中也可以使用Core Graphics，使用该框架编写的代码在Mac和iOS平台上都能运行，例如开源项目core-plot。

4.8 高级练习：阴影和渐变

到目前为止，还无法使用Objective-C类绘制阴影和渐变，只能使用Core Graphics。

绘制阴影之前，需要将阴影效果添加到一个图形上下文中，之后在该图形上下文中绘制的所有不透明图像都会带有阴影效果。可以为阴影设置偏移量（使用CGSize表示）和模糊指数（使用CGFloat表示）。以下是为图形上下文添加阴影效果的方法声明：

```
void CGContextSetShadow (
CGContextRef context,
CGSize offset,
CGFloat blur);
```

还有一个方法可以设置一个颜色参数，为简单起见，这里使用默认阴影颜色。

请注意没有删除阴影效果的函数。因此需要在添加阴影效果之前保存绘图状态，之后再恢复没有阴影效果的状态。类似于以下代码：

```
CGContextSaveGState(currentContext);
CGContextSetShadow(currentContext, CGSizeMake(4,7), 3);
// 在这里绘制的图像会带有阴影效果
CGContextRestoreGState(currentContext);
// 在这里绘制的图像没有阴影效果
```

练习的第一部分是为上一个练习中绘制的图像添加阴影效果，如图4-22所示。

图4-22 绘制阴影

渐变用来在图形中填充一系列平滑过渡的颜色。可以在CGGradientRef中设置需要的颜色和渐变的方式(线性渐变和径向渐变)。类似于以下代码:

```
CGFloat locations[2] = { 0.0, 1.0 };

CGFloat components[8] = { 1.0, 0.0, 0.0, 1.0, // 起始颜色为红色
```

```
                    1.0, 1.0, 0.0, 1.0 }; // 终止颜色为黄色

    CGColorSpaceRef colorspace = CGColorSpaceCreateDeviceRGB();

    CGGradientRef gradient = CGGradientCreateWithColorComponents(colorspace,

                                components,

                                locations,2);

    CGPoint startPoint = ...;

    CGPoint endPoint = ...;

    CGContextDrawLinearGradient(currentContext, gradient,

                startPoint, endPoint, 0);

    CGGradientRelease(gradient);

    CGColorSpaceRelease(colorspace);
```

CGContextDrawLinearGradient函数的最后一个参数用来设置起始位置(startPoint)和终止位置(endPoint)以外的绘制区域的颜色填充方式。如果需要使用起始颜色填充起始位置之前的绘制区域，可以将最后一个参数设置为kCGGradientDraws- BeforeStartLocation。相反，如果需要使用终止颜色填充终止位置之后的绘制区域，可以设置为kCGGradientDrawsAfterEndLocation。如果渐变起止位置以外的绘制区域都需要填充，可以使用位运算符将两个选项连接起来：

```
    CGContextDrawLinearGradient(currentContext, gradient, startPoint, endPoint,

    kCGGradientDrawsBeforeStartLocation | kCGGradientDrawsAfterEndLocation);
```

请注意，与填充颜色不同，无法使用渐变填充路径——渐变会直接填满整个图形上下文。因此，如果需要将渐变应用在指定路径范围以内，必须使用剪切路径(clipping path)裁剪图形上下文。同时，与绘制阴影时的情况类似，没有函数可以删除剪切路径，同样需要在使用剪切路径之前保存绘图状态，填充渐变之后再恢复绘图状态。

以下代码将一个UIBezierPath对象myPath设置为当前图形上下文currentContext的剪切路径：

```
    CGContextSaveGState(currentContext);

    [myPath addClip];

    // 在这里为myPath填充渐变

    CGContextRestoreGState(currentContext);
```

练习的第二部分是绘制一个带有渐变效果的三角形。起始颜色（底部）是黄色，终止颜色（顶部）是绿色，如图4-23所示。

图4-23 绘制带有渐变效果的三角形

第5章视图：重绘与UIScrollView

本章学习视图的重绘机制并继续开发Hypnosister应用，当用户触摸BNRHypnosisView时，圆形的颜色会改变。为了改变圆形的颜色，BNRHypnosisView需要重新绘制自己。在后面的章节中，还会将一个UIScrollView对象添加到Hypnosister的视图层次结构中。

首先要在BNRHypnosisView中声明一个属性，用来表示圆形的颜色。之前的项目一直是在头文件中声明属性的，其实属性也可以在类扩展（class extensions）中声明。

打开BNRHypnosisView.m文件，在文件顶部添加以下代码：

```
#import "BNRHypnosisView.h"

@interface BNRHypnosisView ()

@property (strong, nonatomic) UIColor *circleColor;

@end

@implementation BNRHypnosisView
```

加入的三行代码称为BNRHypnosisView的类扩展。类扩展中声明了一个circleColor属性——为什么要将该属性声明在类扩展中而不是头文件中？原因会在完成改变圆形颜色的功能之后介绍，现在只要将circleColor看成是BNRHypnosisView的一个普通属性即可。

在BNRHypnosisView.m的initWithFrame:方法中，为circleColor属性设置默认颜色，代码如下：

```
- (instancetype)initWithFrame:(CGRect)frame

{

self = [super initWithFrame:frame];

if (self) {

    self.backgroundColor = [UIColor clearColor];

    self.circleColor = [UIColor lightGrayColor];

}

return self;

}
```

在drawRect:方法中修改设置线条颜色的代码，使用circleColor作为线条颜色：

```
// 设置线条宽度为10点

path.lineWidth = 10;

[[UIColor lightGrayColor] setStroke];

[self.circleColor setStroke];

// 绘制路径！

[path stroke];
```

构建并运行应用，BNRHypnosisView对象绘制的圆形颜色应该与之前的相同。下一步是编写视图被触摸时改变圆形颜色的代码。

当用户触摸视图时，视图会收到touchesBegan:withEvent:消息，该消息用来处理触摸事件。第12章会详细介绍触摸事件与事件处理机制，现在只需要直接覆盖touchesBegan: withEvent:方法即可，这样就可以在BNRHypnosisView被触摸后改变circleColor属性所表示的颜色。

在BNRHypnosisView.m中覆盖touchesBegan:withEvent:，首先向控制台打印一条信息，然后创建一个随机生成的UIColor对象，并赋给circleColor属性。

```
// BNRHypnosisView被触摸时会收到该消息

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event

{

NSLog(@"%@ was touched", self);

// 获取三个0到1之间的数字

float red = (arc4random() % 100) / 100.0;

float green = (arc4random() % 100) / 100.0;

float blue = (arc4random() % 100) / 100.0;

UIColor *randomColor = [UIColor colorWithRed:red

                                       green:green

                                       blue:blue

                                       alpha:1.0];
```

```
    self.circleColor = randomColor;

    }
```

构建并运行应用，触摸BNRHypnosisView上的任意位置，控制台会打印视图被触摸的消息，但是圆形的颜色没有改变——BNRHypnosisView没有重新绘制自己。接下来会介绍视图没有重绘的原因以及如何解决这个问题。

5.1 运行循环和重绘视图

ios应用启动时会开始一个运行循环(run loop)。运行循环的工作是监听事件，例如触摸。当事件发生时，运行循环会为相应的事件找到合适的处理方法。这些处理方法会调用其他方法，而这些方法又会调用更多其他方法，依此类推。只有当这些方法都执行完毕时，控制权才会再次回到运行循环。

当应用将控制权交回给运行循环时，运行循环首先会检查是否有等待重绘的视图(即在当前循环收到过setNeedsDisplay消息的视图)，然后向所有等待重绘的视图发送drawRect:消息，最后视图层次结构中所有视图的图层再次组合成一幅完整的图像并绘制到屏幕上。

图5-1是以用户在文本框中输入文字为例，展示视图如何在运行循环中重新绘制自己。

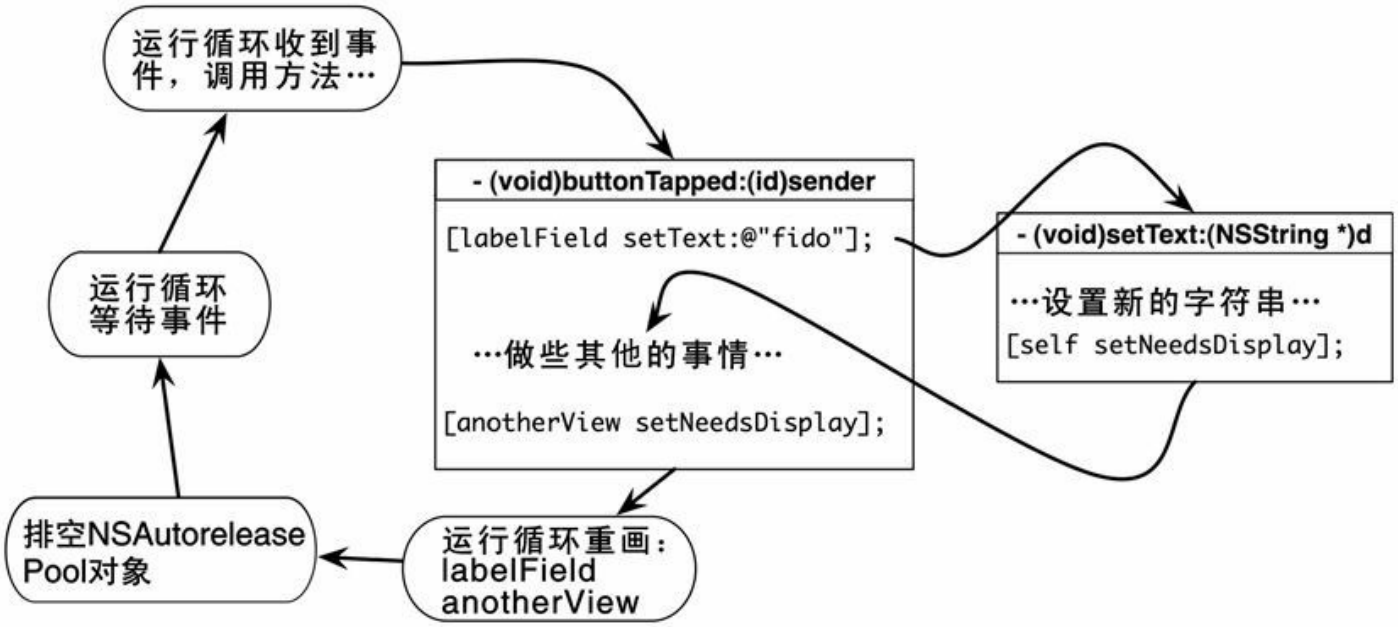


图5-1 视图在运行循环中重新绘制自己

ios做了两方面优化来保证用户界面的流畅性——不重绘显示的内容没有改变的视图；在每次事件处理周期(event handling cycle)中只发送一次drawRect:消息。在事件处理周期中，视图的属性可能会发生多次改变，如果视图在每次属性改变时都重绘自己，就会减慢界面的响应速度。相反，iOS会在运行循环的最后阶段集中处理所有需要重绘的视图，尤其是对于属性发生多次改变的视图，在每次事件处理周期中只重绘一次。

在Hypnosister应用中，首先，通过控制台的输出信息可以知道，BNRHypnosisView的触摸处理方法已经正确捕获了触摸事件。其次，虽然运行循环在touchesBegan:withEvent:执行完成后再次获得了控制权，但是并没有向BNRHypnosisView发送drawRect:消息。

为了标记视图要重绘，必须向其发送setNeedsDisplay消息。iOS SDK中提供的视图对象会自动在显示的内容发生改变时向自身发送setNeedsDisplay消息，以UILabel对象为例，在某个UILabel对象收到setText:消息后，就会将自己标记为要重绘(因为UILabel对象所显示的文字内容变了，所以必须将自己重新绘制到图层上)。而对自定义的UIView子类，必须手动向其发送setNeedsDisplay消息，如Hypnosister应用中的BNRHypnosisView。

在BNRHypnosisView.m中，为circleColor属性实现自定义的存方法，当circleColor改变时，向视图发送setNeedsDisplay消息。

```
- (void)setCircleColor:(UIColor *)circleColor

{

_circleColor = circleColor;

[self setNeedsDisplay];

}
```

再次构建并运行应用，现在可以通过触摸视图改变圆形颜色了。

（还有一种优化方法：只重绘视图的某一区域。可以通过向视图发送setNeedsDisplayInRect:消息标记视图的某一区域要重绘。当视图收到drawRect:消息时，setNeedsDisplayInRect:会将CGRect类型的参数传递给drawRect:，重绘视图的指定区域。但是，随着应用的视图层次结构越来越复杂，计算正确的重绘区域也会越来越困难。通常情况下，不需要手动指定视图的重绘区域，除非绘图代码显着降低了应用性能。）

5.2 类扩展

现在请读者思考，为什么要将circleColor属性声明在BNRHypnosisView的类扩展中？将属性声明在头文件中和类扩展中有什么区别？

在第2章中介绍过，头文件是一个类的“用户手册”，其他类可以通过头文件知道该类的功能和使用方法。使用头文件的目的是向其他类公开该类声明的属性和方法，也就是说，头文件中声明的属性和方法对其他类是可见的（visible）。

但是，并不是每一个属性或方法都要向其他类公开。只会在类的内部使用的属性和方法应当声明在类扩展中。circleColor属性只会被BNRHypnosisView使用，其他类不需要使用该属性，因此它应该被声明在类扩展中。

在类扩展中声明类的内部属性和方法是良好的编程习惯，这样做可以保持头文件的精简，避免内部实现细节的暴露，保证头文件中全部是其他类确实需要使用的属性和方法，从而让其他开发者更容易理解如何使用该类。

在语法上，类扩展的声明方法与头文件类似，需要使用@interface指令，后跟类名，接着为一对空括号。声明属性和方法之后，需要使用@end代表类扩展的声明至此结束。通常应将该类扩展写在实现文件顶部，位于@implementation之前：

```
#import "BNRHypnosisView.h"

@interface BNRHypnosisView ()

@property (strong, nonatomic) UIColor *circleColor;

@end

@implementation BNRHypnosisView
```

子类同样无法访问父类在类扩展中声明的属性和方法。例如，BNRHypnosisView的子类无法访问circleColor属性。

有时需要让其他开发者了解类的某些内部属性和方法，以便更好地理解类的工作原理和使用方法。可以在另一个文件中声明类扩展，并将该文件导入类的实现文件中。

本书将使用类扩展声明其他类不需要访问的属性和方法，从而隐藏内部实现细节。

5.3 使用UIScrollView

本节将为Hypnosister应用添加一个UIScrollView对象，使其成为应用窗口的子视图，然后再将BNRHypnosisView作为子视图加入UIScrollView对象（见图5-2）。

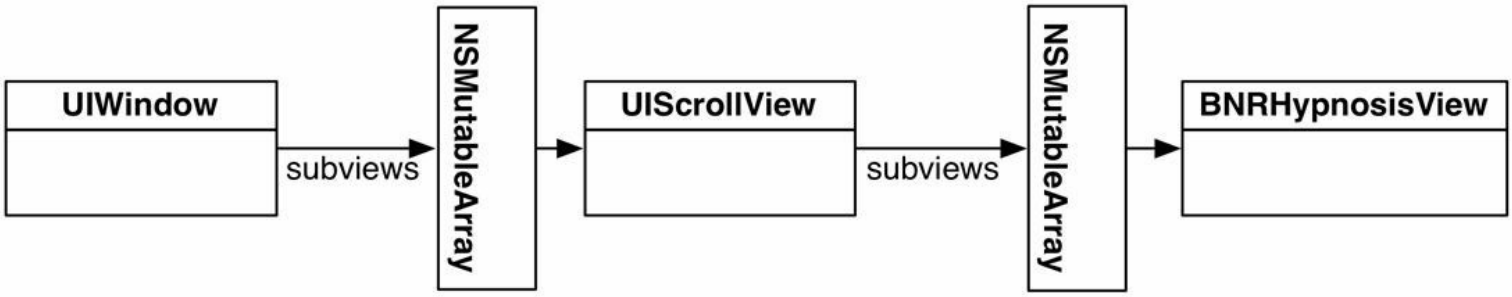


图5-2 加入UIScrollView之后的视图层次结构

通常情况下，UIScrollView对象适用于那些尺寸大于屏幕的视图。当某个视图是UIScrollView对象的子视图时，该对象会画出该视图的某块区域（形状为矩形）。当用户按住这块矩形区域并移动手指（即拖动，pan）时，UIScrollView对象会改变该矩形所显示的子视图区域。读者可以将UIScrollView对象看成是镜头，而其子视图是拍摄的景观。这里移动的是“镜头”，而不是“景观”（见图5-3）。UIScrollView对象的尺寸就是这个“镜头”的尺寸，而其能够拍摄的范围是由其属性contentSize决定的。通常情况下，contentSize的数值就是子视图的尺寸。

UIScrollView是UIView的子类，同样可以使用initWithFrame:消息初始化，还可以将其作为子视图添加到其他视图中。

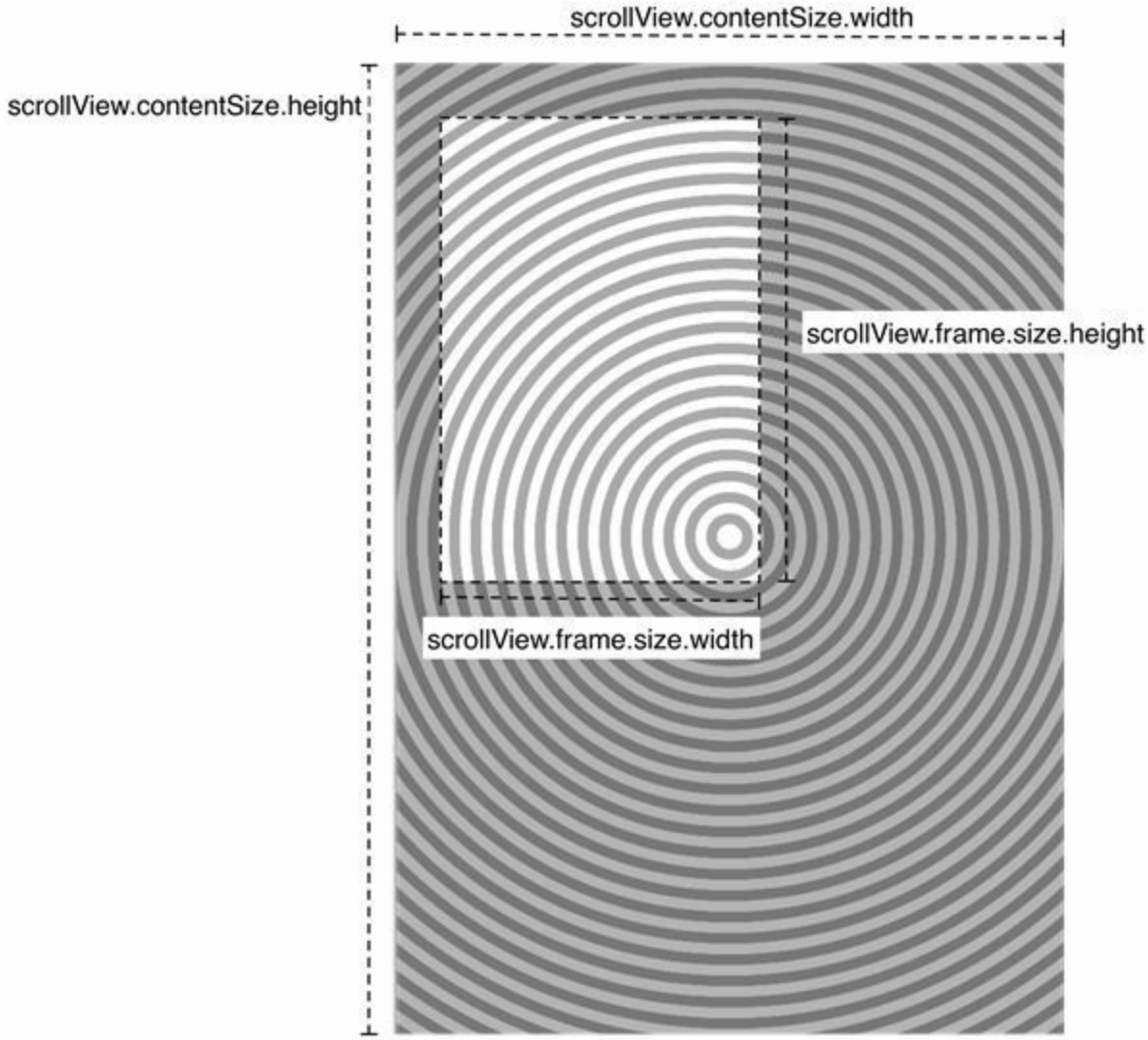


图5-3 UIScrollView对象及其contentSize

在BNRAppDelegate.m中，创建一个有着超大尺寸的BNRHypnosisView对象并将其加入一个UIScrollView对象，然后将这个UIScrollView对象加入窗口，代码如下：

```
- (BOOL)application:(UIApplication *)application

didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

{

self.window = [[UIWindow alloc] initWithFrame:

        [[UIScreen mainScreen] bounds]];

// 在这里添加应用启动后的初始化代码

 CGRect firstFrame = self.window.bounds;

BNRHypnosisView *firstView = [[BNRHypnosisView alloc]
```

```
initWithFrame:firstFrame];

[self.window addSubview:firstView];

// 创建两个CGRect结构分别作为UIScrollView对象和BNRHypnosisView对象的frame

CGRect screenRect = self.window.bounds;

CGRect bigRect = screenRect;

bigRect.size.width *= 2.0;

bigRect.size.height *= 2.0;

// 创建一个UIScrollView对象，将其尺寸设置为窗口大小

UIScrollView *scrollView = [[UIScrollView alloc] initWithFrame:screenRect];

[self.window addSubview:scrollView];

// 创建一个有着超大尺寸的BNRHypnosisView对象并将其加入UIScrollView对象

BNRHypnosisView *hypnosisView = [[BNRHypnosisView alloc]

                                initWithFrame:bigRect];

[scrollView addSubview:hypnosisView];

// 告诉UIScrollView对象"取景"范围有多大

scrollView.contentSize = bigRect.size;

self.window.backgroundColor = [UIColor whiteColor];
```

构建并运行应用，可以上、下、左、右拖动来查看超大尺寸的BNRHypnosisView对象的其余部分（见图5-4）。

图5-4 BNRHypnosisView的右上角

移动BNRHypnosisView对象的同时，圆形的颜色也会发生变化。这是由于移动视图时也会产生触摸事件，因此运行循环会将触摸事件发送给UIScrollView和BNRHypnosisView。第13章会介绍如何检测和处理“tap（点击）”手势，用来区别触摸手势和移动手势。

使用UIScrollView还可以实现“Pinch-to-zoom（捏合缩放）”功能。虽然只需要添加几行代码就可以实现该功能，但是这些代码涉及第7章中的技术。为Hypnosister添加捏合缩放功能是第7章的练习。

拖动与分页

UIScrollView对象还可以滑动显示所有加入UIScrollView对象的子视图。

在BNRAppDelegate.m中，将BNRHypnosisView对象的尺寸改回与屏幕的尺寸相同，然后再创建一个BNRHypnosisView对象，将其尺寸也设置为与屏幕的尺寸相同并加入UIScrollView对象。此外，还要将UIScrollView对象的contentSize的宽度设置为屏幕宽度的2倍，高度不变，代码如下：

```
// 创建两个CGRect结构分别作为UIScrollView对象和BNRHypnosisView对象的frame
```

```
CGRect screenRect = self.window.bounds;
CGRect bigRect = screenRect;
bigRect.size.width *= 2.0;
bigRect.size.height *= 2.0;
// 创建一个UIScrollView对象，将其尺寸设置为窗口大小
UIScrollView *scrollView = [[UIScrollView alloc] initWithFrame:screenRect];
[self.window addSubview:scrollView];
// 创建一个有着超大尺寸的BNRHypnosisView对象并将其加入UIScrollView对象
BNRHypnosisView *hypnosisView = [[BNRHypnosisView alloc]
                                 initWithFrame:bigRect];
// 创建一个大小与屏幕相同的BNRHypnosisView对象并将其加入UIScrollView对象
BNRHypnosisView *hypnosisView = [[BNRHypnosisView alloc]
                                 initWithFrame:screenRect];
[scrollView addSubview:hypnosisView];
// 创建第二个大小与屏幕相同的BNRHypnosisView对象并放置在第一个BNRHypnosisView
// 对象的右侧，使其刚好移出屏幕外
screenRect.origin.x += screenRect.size.width;
BNRHypnosisView *anotherView = [[BNRHypnosisView alloc]
initWithFrame:screenRect];
[scrollView addSubview:anotherView];
// 告诉UIScrollView对象"取景"范围有多大
scrollView.contentSize = bigRect.size;
```

构建并运行应用，按从左向右的方向拖动屏幕，应该能先后看到两个BNRHypnosisView对象。请注意，UIScrollView对象有可能同时显示两个BNRHypnosisView对象的部分区域，即两个BNRHypnosisView对象的连接部分（见图5-5）。

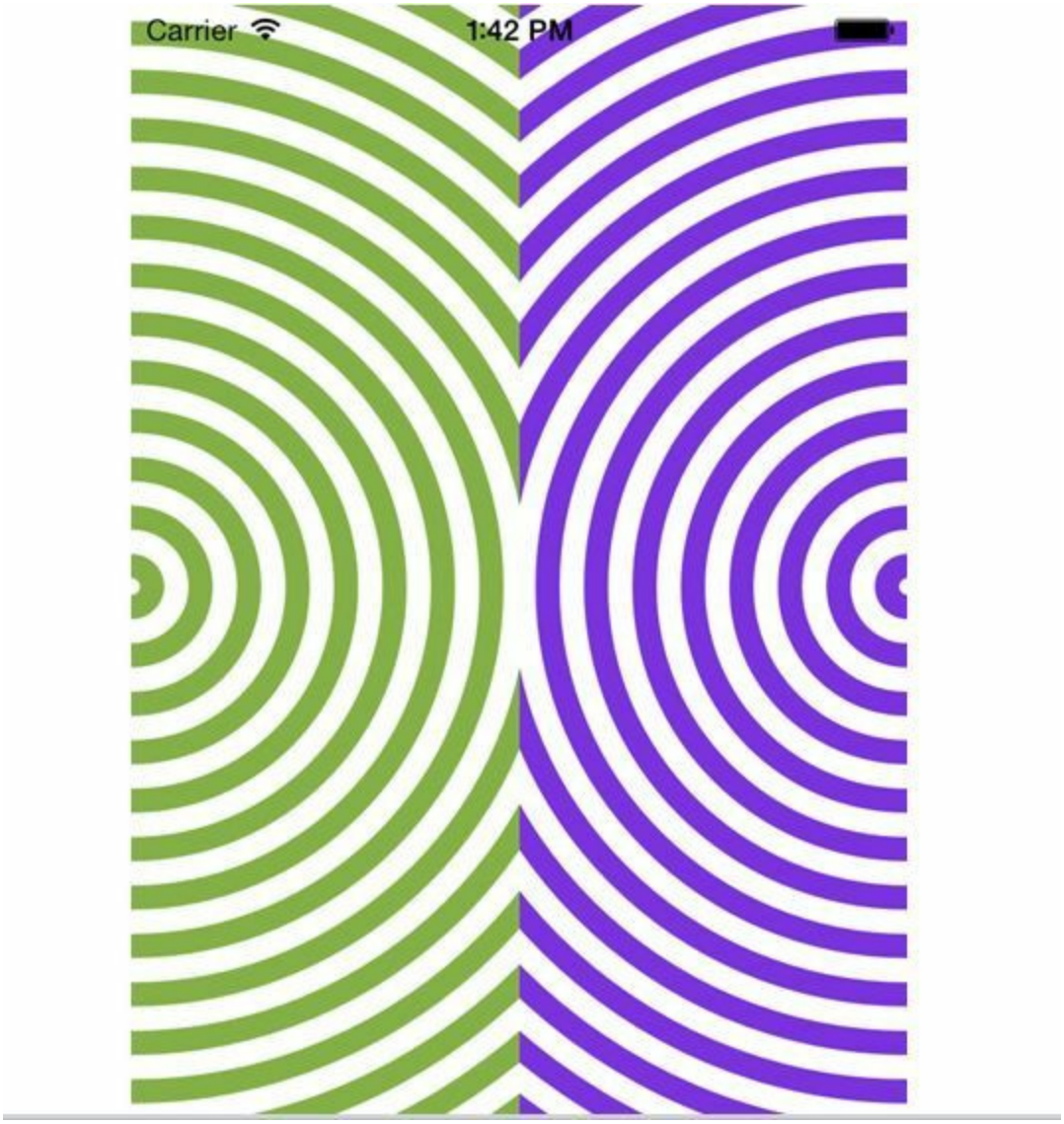


图5-5 UIScrollView显示两个BNRHypnosisView的连接部分

某些情况下，这种行为是符合预期的。但是有时也要让UIScrollView对象的“镜头”的边和其显示的某个视图的边对齐。为此，要将UIScrollView对象的pagingEnabled设置为YES。在BNRAppDelegate.m中将scrollView的pagingEnabled设置为YES，代码如下：

```
UIScrollView *scrollView = [[UIScrollView alloc] initWithFrame:screenRect];

[scrollView setPagingEnabled:YES];

[[self window] addSubview:scrollView];
```

构建并运行应用，拖动屏幕直到能看到两个BNRHypnosisView对象的连接部分，然后松开手指。UIScrollView对象应该会自动将“镜头”切换到其中一个BNRHypnosisView对象上。UIScrollView对象的分页实现原理是：UIScrollView对象会根据其bounds的尺寸，将contentSize分割为尺寸相同的多个区域。拖动结束后，UIScrollView实例会自动滚动并只显示其中的一个区域。

第6章 视图控制器

第5章创建了一个视图层次结构(一个UIScrollView和它的两个子视图)并将其添加到屏幕上。第5章使用的方式是将UIScrollView作为子视图添加到应用窗口中，但是更常见的做法是使用视图控制器。

视图控制器是UIViewController类或其子类的对象。每个视图控制器都负责管理一个视图层次结构，包括创建视图层次结构中的视图并处理相关用户事件，以及将整个视图层次结构添加到应用窗口。

本章将创建一个名为HypnoNerd的应用。在HypnoNerd中，用户可在两个视图层次结构之间自由切换——第一个视图层次结构用于催眠自己，第二个用于设置催眠提醒时间(见图6-1)。

图6-1 HypnoNerd的两个视图层次结构

为了实现切换功能，需要为HypnoNerd应用创建两个UIViewController子类：BNRHypnosisViewController和BNRReminderViewController，并使用一个名为UITabBarController的类在这两个视图控制器之间切换。

使用Empty Application模板创建新项目，并将项目命名为HypnoNerd，然后根据图6-2填写设置选项。

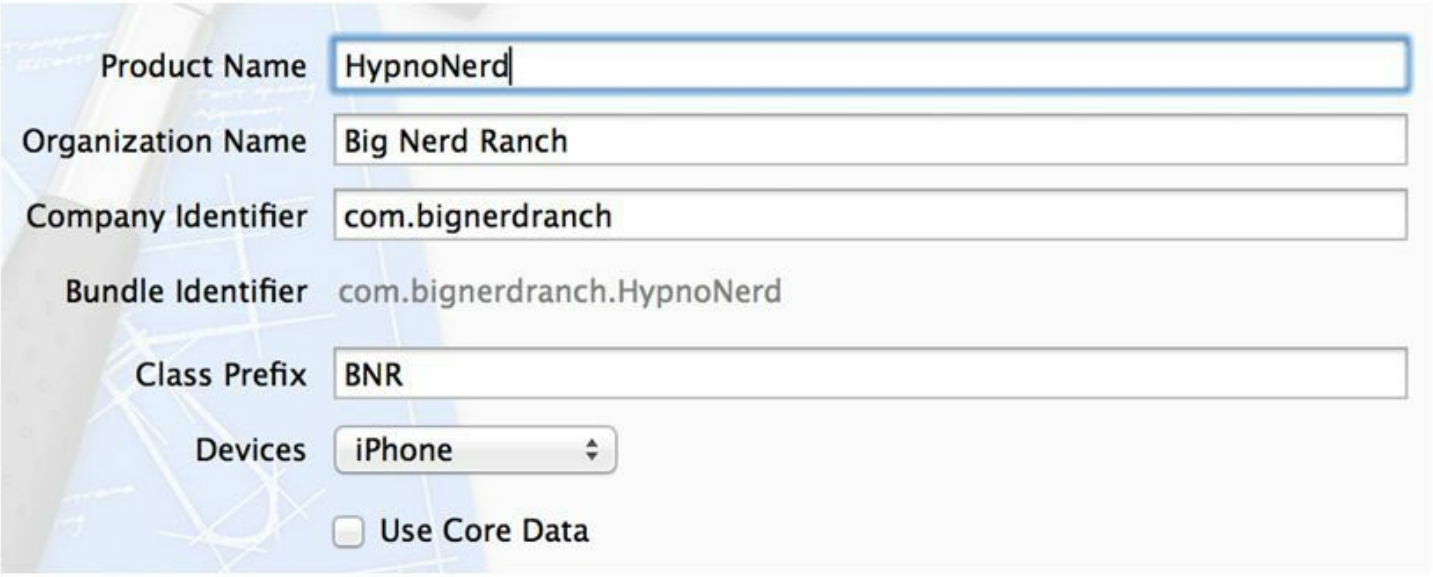


图6-2 创建新项目

HypnoNerd应用需要使用BNRHypnosisView（编写Hypnosister时创建的UIView子类），在Finder中找到Hypnosister的项目目录，并将目录中的BNRHypnosisView.h和BNRHypnosisView.m拖曳至HypnoNerd的项目导航面板。

在Xcode弹出的下拉窗口中选中Copy items into destination group's folder（if needed）（拷贝文件或目录至目标组的目录（如果需要）），单击Finish按钮（见图6-3）。

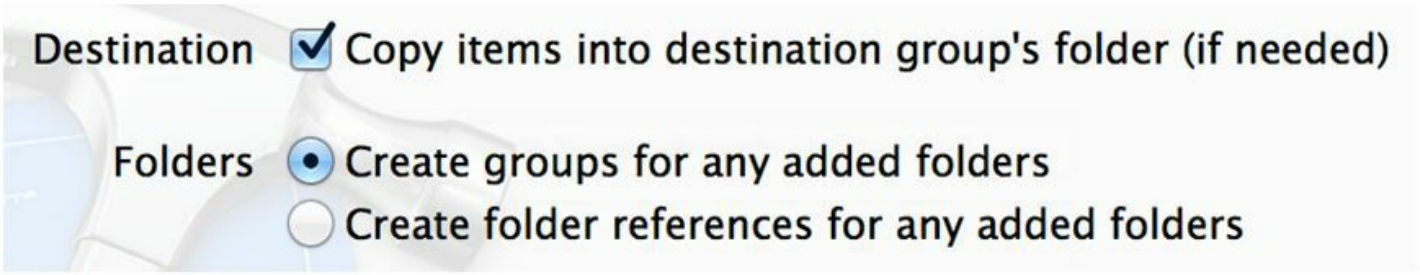


图6-3 将文件拷贝到HypnoNerd项目

Xcode会先拷贝BNRHypnosisView.h和BNRHypnosisView.m，然后将拷贝后得到的两个文件保存在HypnoNerd的项目目录下，最后将这两个文件加入HypnoNerd项目。

6.1 创建UIViewController子类

从File菜单中选择New→File…，然后选中窗口左侧iOS部分的Cocoa Touch，再选中窗口右侧的Objective-C class，最后单击Next按钮。

在新出现的面板中，在Class文本框中输入BNRHypnosisViewController，在Subclass of下拉菜单中选择NSObject，单击Next按钮（见图6-4）。Xcode会提示创建文件，单击Create按钮。

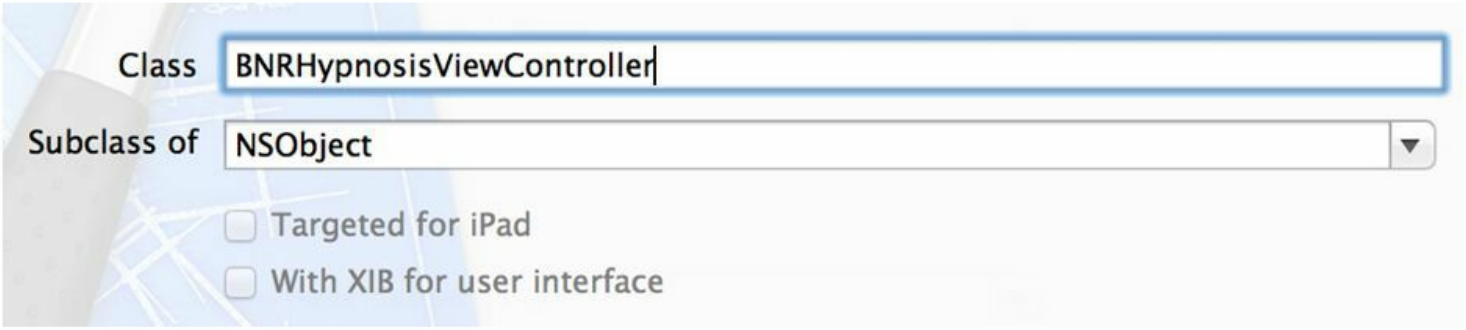


图6-4 创建BNRHypnosisViewController

读者可能会问，为什么要将BNRHypnosisViewController的父类设置为NSObject，而不是UIViewController？选择NSObject会生成最简单的Xcode文件模板，从最简单的模板开始编写代码有利于读者理解应用的工作原理。

打开BNRHypnosisViewController.h，将BNRHypnosisViewController的父类修改为UIViewController，修改后的代码如下：

```
~~@interface BNRHypnosisViewController : NSObject~~

@interface BNRHypnosisViewController : UIViewController

@end
```

UIViewController的view属性

BNRHypnosisViewController从UIViewController中继承了一个重要属性：

```
@property (nonatomic, strong) UIView *view;
```

view属性指向一个UIView对象。之前介绍过，UIViewController对象可以管理一个视图层次结构，view就是这个视图层次结构的根视图，当程序将view作为子视图加入窗口时，也会加入UIViewController对象所管理的整个视图层次结构（见图6-5）。

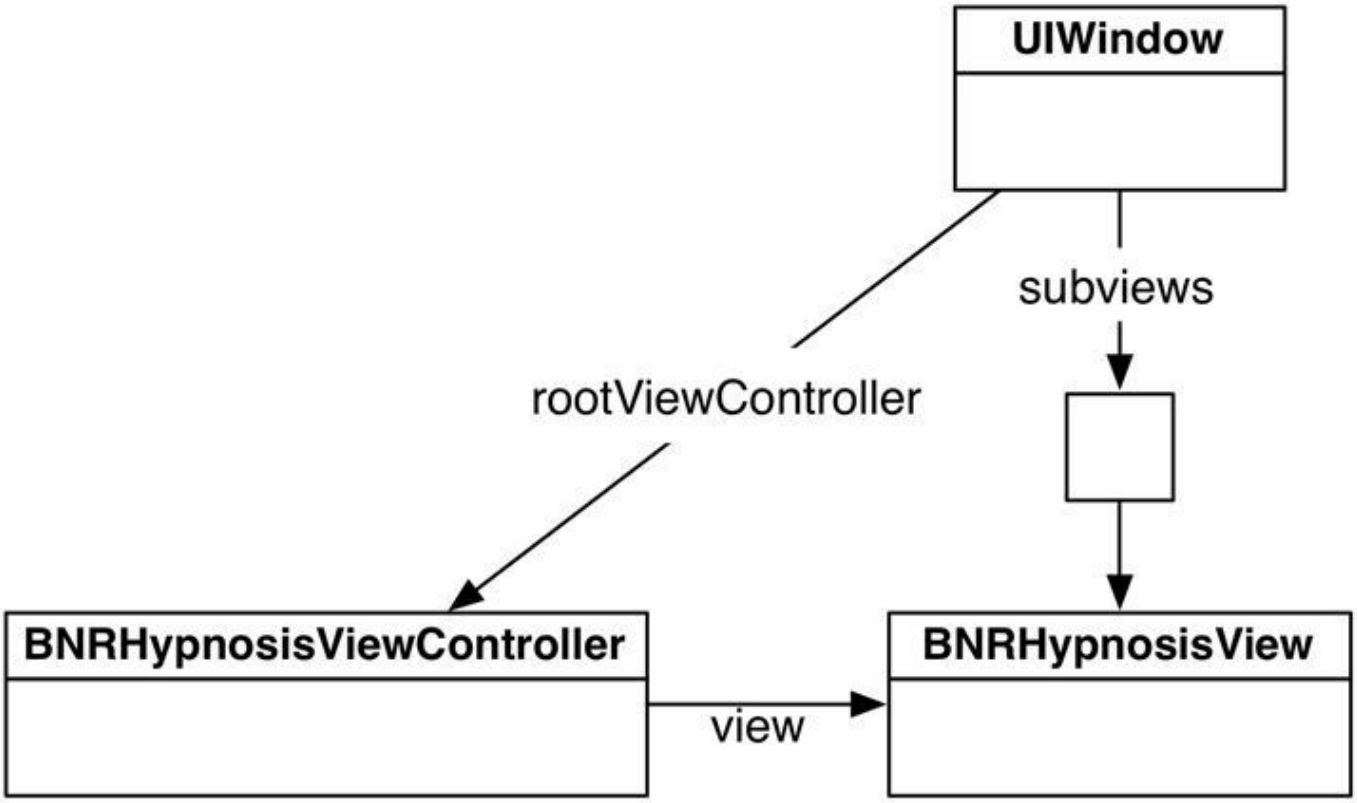


图6-5 HypnoNerd的对象图

视图控制器不会在其被创建出来的那一刻马上创建并载入相应的视图。只有当应用需要将某个视图控制器的视图显示到屏幕上时，相应的视图控制器才会创建其视图。这种延迟加载(lazy loading)视图的做法能提高内存的使用效率。

视图控制器可以通过两种方式创建视图层次结构。

•代码方式：覆盖UIViewController中的loadView方法。

•文件方式：使用Interface Builder创建一个NIB文件，然后加入所需的视图层次结构，最后视图控制器会在运行时加载由该NIB文件编译而成的XIB文件。(在第1章中介绍过，应用在运行时会按需载入XIB文件并激活文件中的视图。)

因为BNRHypnosisViewController的视图层次结构中只包含一个视图，所以接下来将通过代码方式来创建视图。

通过代码方式创建视图

在BNRHypnosisViewController.m顶部导入BNRHypnosisView的头文件，然后覆盖loadView方法，创建一个大小与屏幕相同的BNRHypnosisView对象并将其赋给视图控制器的view属性：

```
#import "BNRHypnosisViewController.h"
```

```
#import "BNRHypnosisView.h"

@implementation BNRHypnosisViewController

- (void)loadView

{

// 创建一个BNRHypnosisView对象

BNRHypnosisView *backgroundView = [[BNRHypnosisView alloc] init];

// 将BNRHypnosisView对象赋给视图控制器的view属性

self.view = backgroundView;

}

@end
```

视图控制器刚被创建时，其view属性会被初始化为nil。之后，当应用需要将该视图控制器的视图显示到屏幕上时，如果view属性是nil，就会自动调用loadView方法。

接下来将BNRHypnosisViewController对象的视图层次结构加入应用窗口，并将其显示在屏幕上。

设置根视图控制器

为了将视图控制器的视图层次结构加入应用窗口，UIWindow对象提供了一个方法：setRootViewController:。当程序将某个视图控制器设置为UIWindow对象的rootViewController时，UIWindow对象会将该视图控制器的view作为子视图加入窗口。此外，还会自动调整view的大小，将其设置为与窗口的大小相同。

首先在BNRAppDelegate.m文件顶部导入BNRHypnosisViewController.h，然后创建一个BNRHypnosisViewController对象，最后将其设置为UIWindow对象的rootViewController，代码如下：

```
#import "BNRAppDelegate.h"

#import "BNRHypnosisViewController.h"

@implementation BNRAppDelegate

- (BOOL)application:(UIApplication *)application
```

```
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

{

self.window = [[UIWindow alloc] initWithFrame:

        [[UIScreen mainScreen] bounds]];

// 在这里添加应用启动后的初始化代码

BNRHypnosisViewController *hvc = [[BNRHypnosisViewController alloc] init];

self.window.rootViewController = hvc;

self.window.backgroundColor = [UIColor whiteColor];

[self.window makeKeyAndVisible];

return YES;

}
```

rootViewController的view需要在应用启动完毕后就显示，所以UIWindow对象会在设置完rootViewController后立刻加载其view。

假设读者需要自己实现setRootViewController:，那么可以根据第4章所学到的知识，代码示例如下：

```
- (void)setRootViewController:(UIViewController *)viewController

{

// 获取根视图控制器的视图

UIView *rootView = viewController.view;

// 根据UIWindow对象的bounds，为视图创建相应的frame

CGRect viewFrame = self.bounds;

rootView.frame = viewFrame;

// 将视图作为子视图加入UIWindow对象

[self addSubview:rootView];

// 将viewController赋给实例变量_rootViewController
```

```
    _rootViewController = viewController;

    }
```

方法中的第一行代码需要使用根视图控制器的view，由于此时根视图控制器刚刚被创建，其view是nil，因此需要向其发送loadView消息创建view。

构建并运行应用，HypnoNerd的界面应该与Hypnosister的类似，但是HypnoNerd是使用视图控制器来显示BNRHypnosisView的，而不是直接将BNRHypnosisView添加到UIWindow。虽然使用视图控制器增加了复杂性，但是在本章结尾就能看到这样做的灵活性和可扩展性。

6.2 另一个视图控制器

本节将创建第二个视图控制器——BNRReminderViewController。该视图控制器的界面用于让用户选择催眠时间(见图6-6)，然后在该时间提醒用户。请注意，即使HypnoNerd此时不再运行，用户也应该能收到提醒。

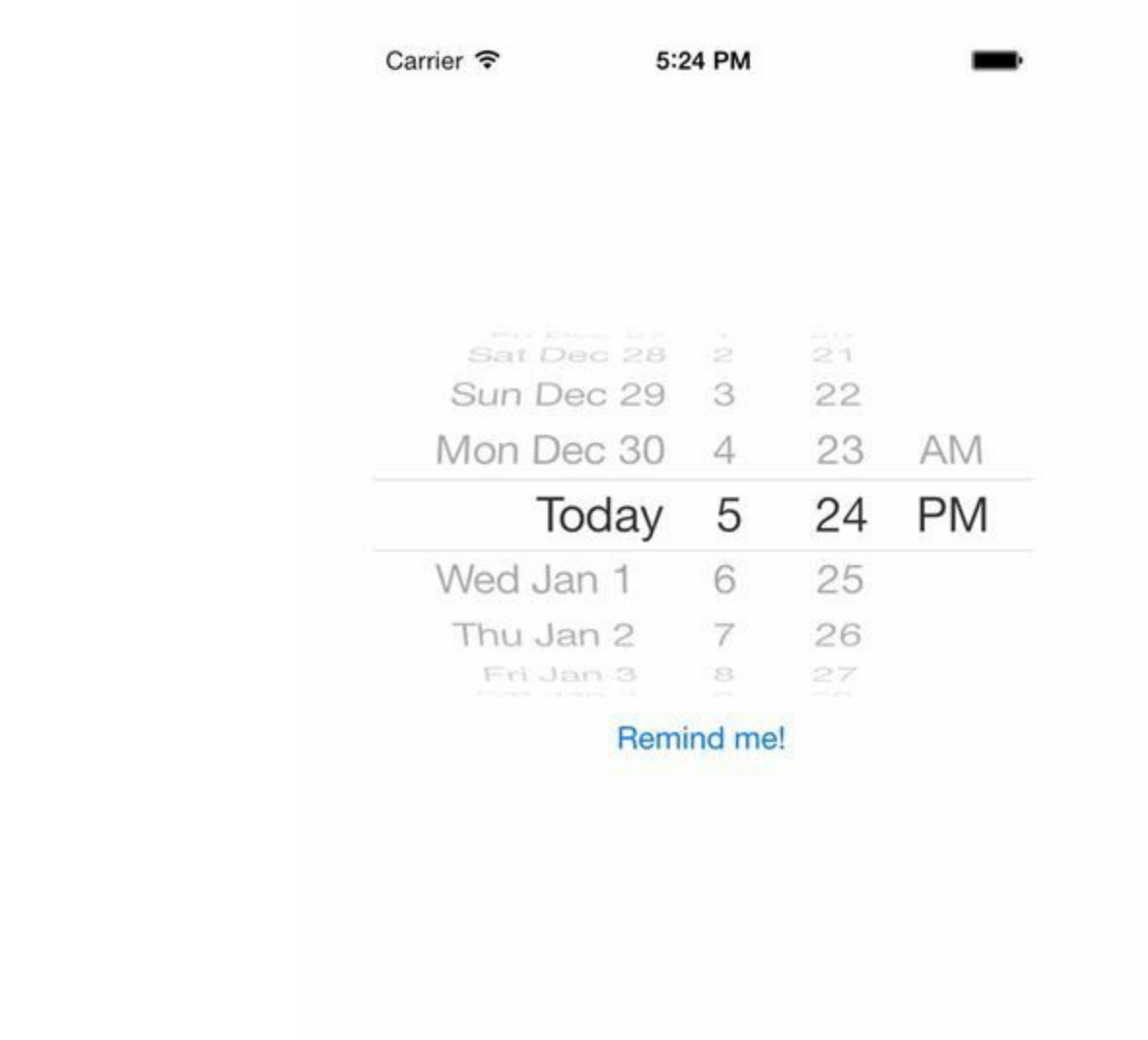


图6-6 BNRReminderViewCobtroller

使用Xcode创建一个新的Objective-C类(键盘快捷键是Command-N)，选择NSObject作为父类，并将其命名为BNRReminderViewController。

在BNRReminderViewController.h中，将BNRReminderViewController的父类改为UIViewController，代码如下：

~~@interface BNRReminderViewController : NSObject~~

@interface BNRReminderViewController : UIViewController

BNRReminderViewController的view是一个全屏幕尺寸的UIView对象，并包含两个子视图：一个UIDatePicker对象和一个UIButton对象(见图6-7)。

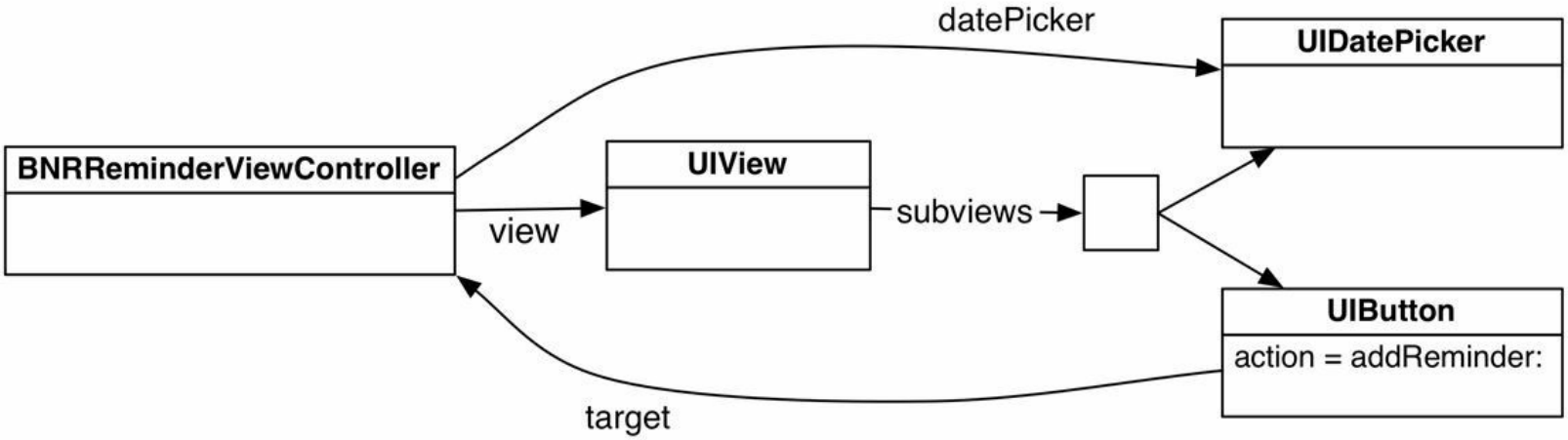


图6-7 BNRReminderViewController的视图层次结构

因此，需要为BNRReminderViewController添加一个datePicker属性，指向一个UIDatePicker对象。同时，需要将BNRReminderViewController设置为UIButton对象的目标，还需要编写一个addReminder:方法并设置为UIButton对象的动作。

BNRReminderViewController对象的view会有两个子视图。当某个视图控制器的view拥有子视图时，使用Interface Builder创建视图层次结构会方便很多。

在Interface Builder中创建视图

首先打开BNRReminderViewController.m，添加一个类扩展，声明一个datePicker属性，然后创建一个addReminder:方法，向控制台输出datePicker的日期。

```
#import "BNRReminderViewController.h"

@interface BNRReminderViewController ()

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

@end

@implementation BNRReminderViewController

- (IBAction)addReminder:(id)sender

{

NSDate *date = self.datePicker.date;

NSLog(@"Setting a reminder for %@", date);
```

```
}

@end
```

第1章中介绍过，IBOutlet和IBAction关键字告诉Xcode，这些属性或方法之后会在Interface Builder中关联。

接下来需要创建一个XIB文件，从File菜单中选择New→File…，然后选中iOS部分的User Interface，再选中Empty（见图6-8），最后单击Next按钮。

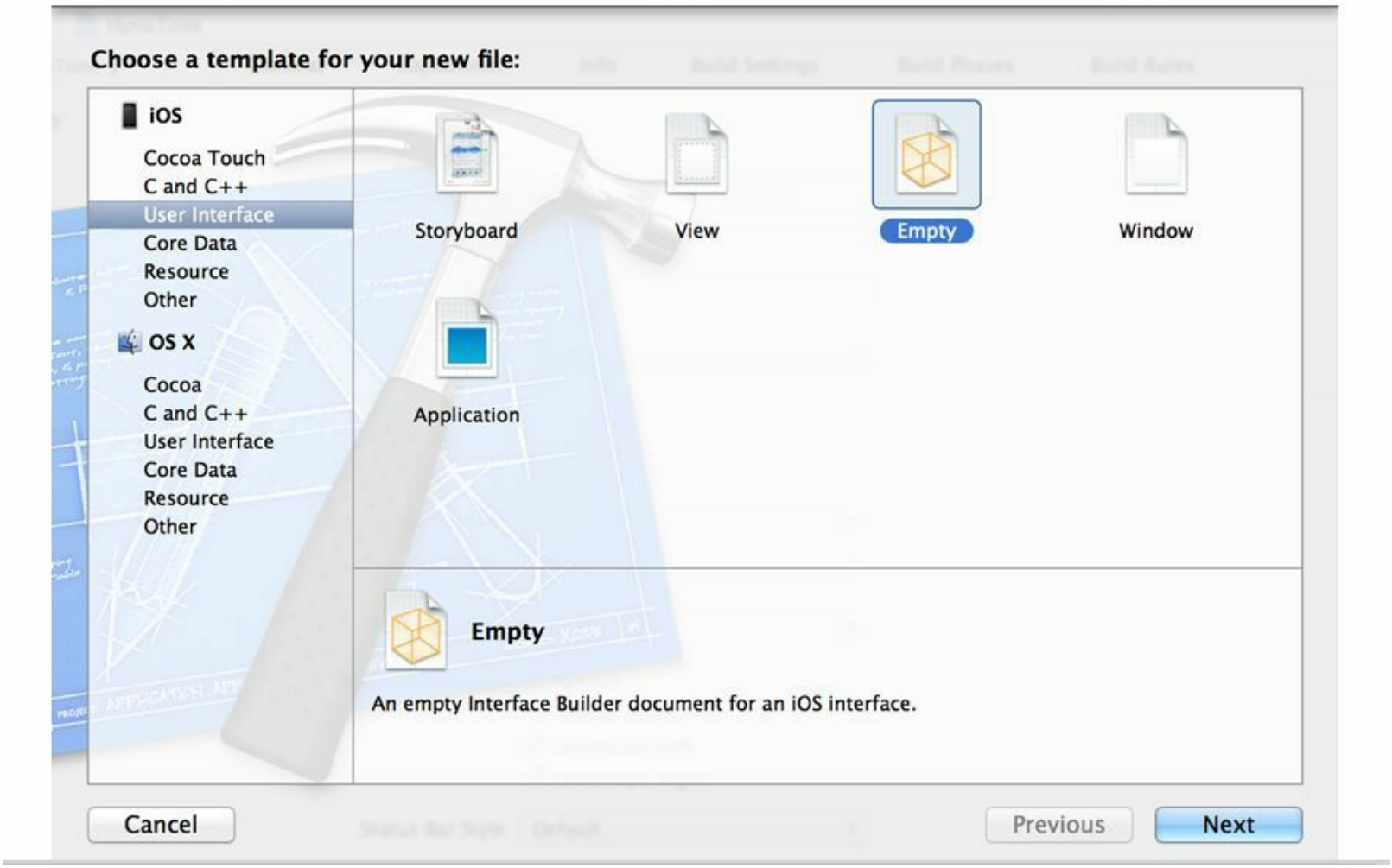


图6-8 创建空的XIB文件

在新出现的面板中，选择Device Family下拉菜单中的iPhone，单击Next按钮。

将新文件命名为BNRReminderViewController.xib并保存。（请读者务必按照本书给出的文件名创建文件。书中很多文件名都是根据iOS SDK的各种约定取的，以便代码能够正常工作。）

选中项目导航面板中的BNRReminderViewController.xib，该文件会在Interface Builder中打开。

创建视图对象

从对象库面板(位于Xcode右下方)拖曳一个UIView对象至画布。默认情况下，该对象的大小和屏幕大小相同，不需要手动调整其大小。

然后再拖曳一个UIButton对象和一个UIDatePicker对象至UIView对象，并设置其大小和位置，再双击UIButton对象修改按钮标题，如图6-9所示。

图6-9 BNRReminderViewController的XIB文件

在画布左边的大纲视图中可以看见已经创建好的视图层次结构：View是根视图，其中包含两个子视图，分别是Picker和Button(见图6-10)。

图6-10 BNRReminderViewController.xib中的视图层次结构

加载NIB文件

当视图控制器从NIB文件中创建视图层次结构时，不需要覆盖loadView方法，默认的loadView方法会自动处理NIB文件中包含的视图层次结构。

接下来在UIViewController的指定初始化方法中为BNRReminderViewController设置需要加载的NIB文件：

```
- (instancetype)initWithNibName:(NSString *)nibName
                         bundle:(NSBundle *)nibBundle;
```

该方法包含两个参数，分别用于指定NIB文件的文件名及其所在的程序包。

在BNRAppDelegate.m文件顶部导入BNRReminderViewController.h，然后创建一个BNRReminderViewController对象，再将其设置为应用窗口的根视图控制器，代码如下：

```
#import "BNRReminderViewController.h"

@implementation BNRAppDelegate

- (BOOL)application:(UIApplication *)application
```

```
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

{

self.window = [[UIWindow alloc] initWithFrame:

        [[UIScreen mainScreen] bounds]];

// 在这里添加应用启动后的初始化代码

// 这段代码不会使用hvc，所以Xcode会显示一处相应的警告，先忽略

BNRHypnosisViewController *hvc = [[BNRHypnosisViewController alloc] init];

// 获取指向NSBundle对象的指针，该NSBundle对象代表应用的主程序包

NSBundle *appBundle = [NSBundle mainBundle];

// 告诉初始化方法在appBundle中查找BNRReminderViewController.xib文件

BNRReminderViewController *rvc = [[BNRReminderViewController alloc]

initWithNibName:@"BNRReminderViewController"

        bundle:appBundle];

self.window.rootViewController = rvc;

self.window.backgroundColor = [UIColor whiteColor];

[self.window makeKeyAndVisible];

return YES;

}
```

向NSBundle发送mainBundle消息可以得到应用的主程序包。主程序包对应于文件系统中项目的根目录，包含代码文件和资源文件（例如NIB文件和图片），初始化方法需要的BNRReminderViewController.xib文件也包含在主程序包中。

至目前为止，视图层次结构中的所有视图对象都已经创建和设置好了，视图控制器的初始化方法可以正确加载NIB文件了，视图控制器也成为UIWindow的根视图控制器并将视图层次结构加入到应用窗口中了。但是，如果现在构建并运行应用，则应用会崩溃，请注意控制台中输出的异常信息：

```
'-[UIViewController _loadViewFromNibNamed:bundle:] loaded the

"BNRReminderViewController" nib but the view outlet was not set.'
```

当NIB文件被加载时，会创建文件中的视图对象，但是这些视图对象在应用运行时并没有与BNRReminderViewController关联起来，包括BNRReminderViewController的view属性。当该视图控制器需要将view添加到应用窗口时，view是nil，因此应用崩溃并输出了异常信息。

那么如何将XIB文件中创建的视图对象在运行时与视图控制器进行关联呢？这时需要使用File's Owner对象。

关联File'Owner

File's Owner对象是一个占位符对象（placeholder），它是XIB文件特意留下的一个“空洞”。当某个视图控制器将XIB文件加载为NIB文件时，首先会创建XIB文件中的所有视图对象，然后会将自己填入相应的File's Owner空洞，并建立之前在Interface Builder中设置的关联（见图6-11）。

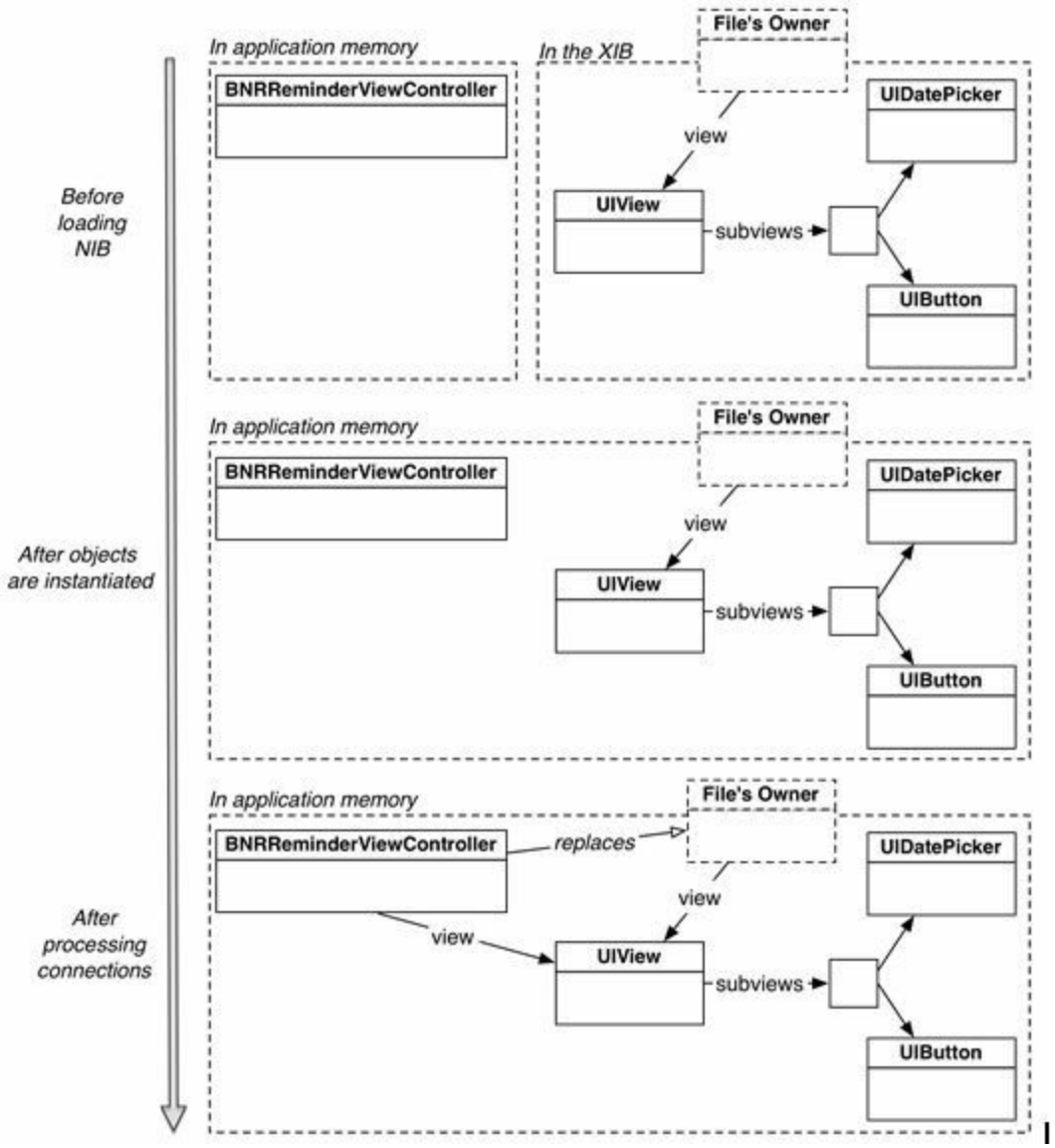


图6-11 NIB文件加载过程的时间轴

因此，如果要在运行时关联加载NIB文件的对象，可在XIB文件中关联File's Owner。首先需

设置BNRReminderViewController.xib文件中的File's Owner是BNRReminderViewController。

重新打开BNRReminderViewController.xib，选中大纲视图中的File's Owner，单击位于检视面板区域上方的按钮，打开标识检视面板（identity inspector）。将Class文本框中的内容（File's Owner表示的类）改为BNRReminderViewController（见图6-12）。

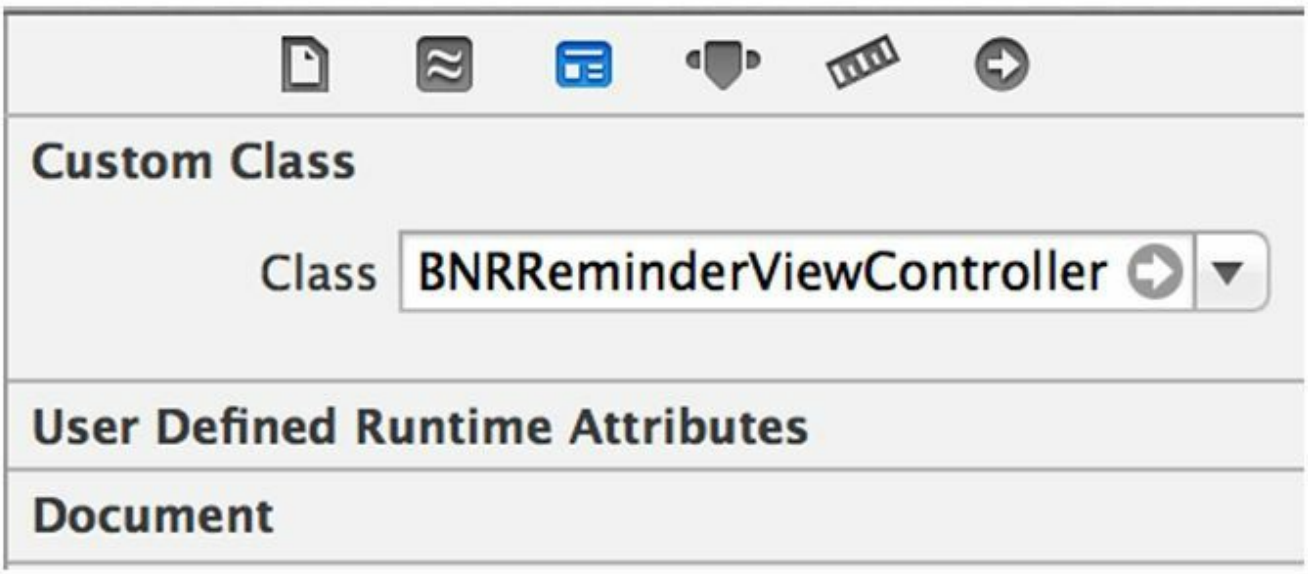


图6-12 针对File's Owner的标识检视面板

接下来在XIB文件中添加所需的关联，首先是视图控制器的view属性，在大纲视图中按住Control并单击File's Owner，Xcode会显示关联面板，列出所有可用的关联，在插座变量（outlets）部分有一个view。将view插座变量与画布中的UIView对象关联起来。这样，当BNRReminderViewController对象载入该XIB文件时，其view属性就能自动指向画布中的UIView对象（见图6-13）。

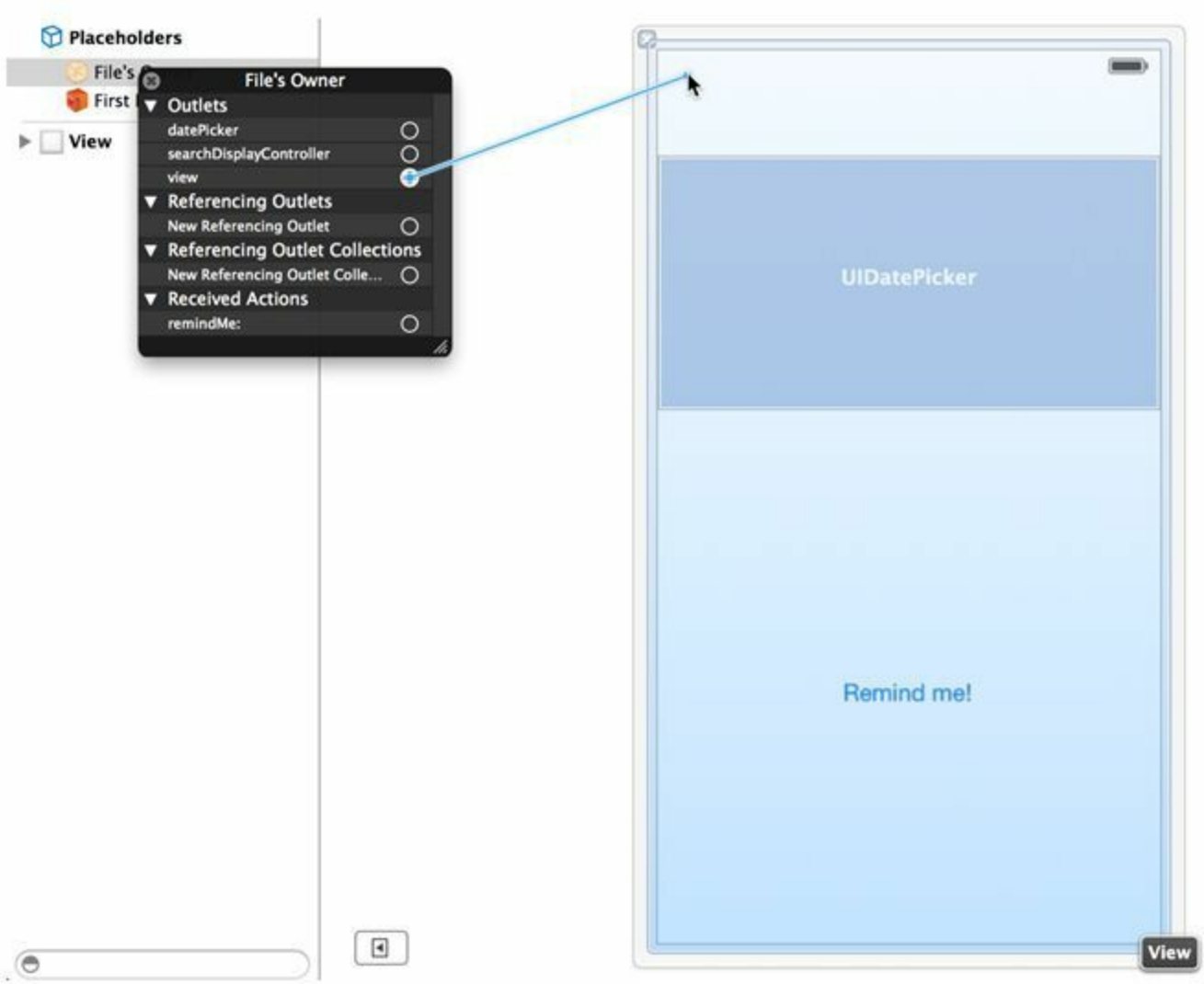


图6-13 关联view插座变量

现在，BNRReminderViewController对象在运行时可以加载view了，构建并运行应用，BNRReminderViewController会加载XIB文件中创建的UIView对象，应用也不会再崩溃了。

使用同样的方法关联其余插座变量，首先将插座变量datePicker关联至UIDatePicker对象，然后将UIButton对象关联至File's Owner并选择弹出式菜单中的addReminder:，如图6-14所示。

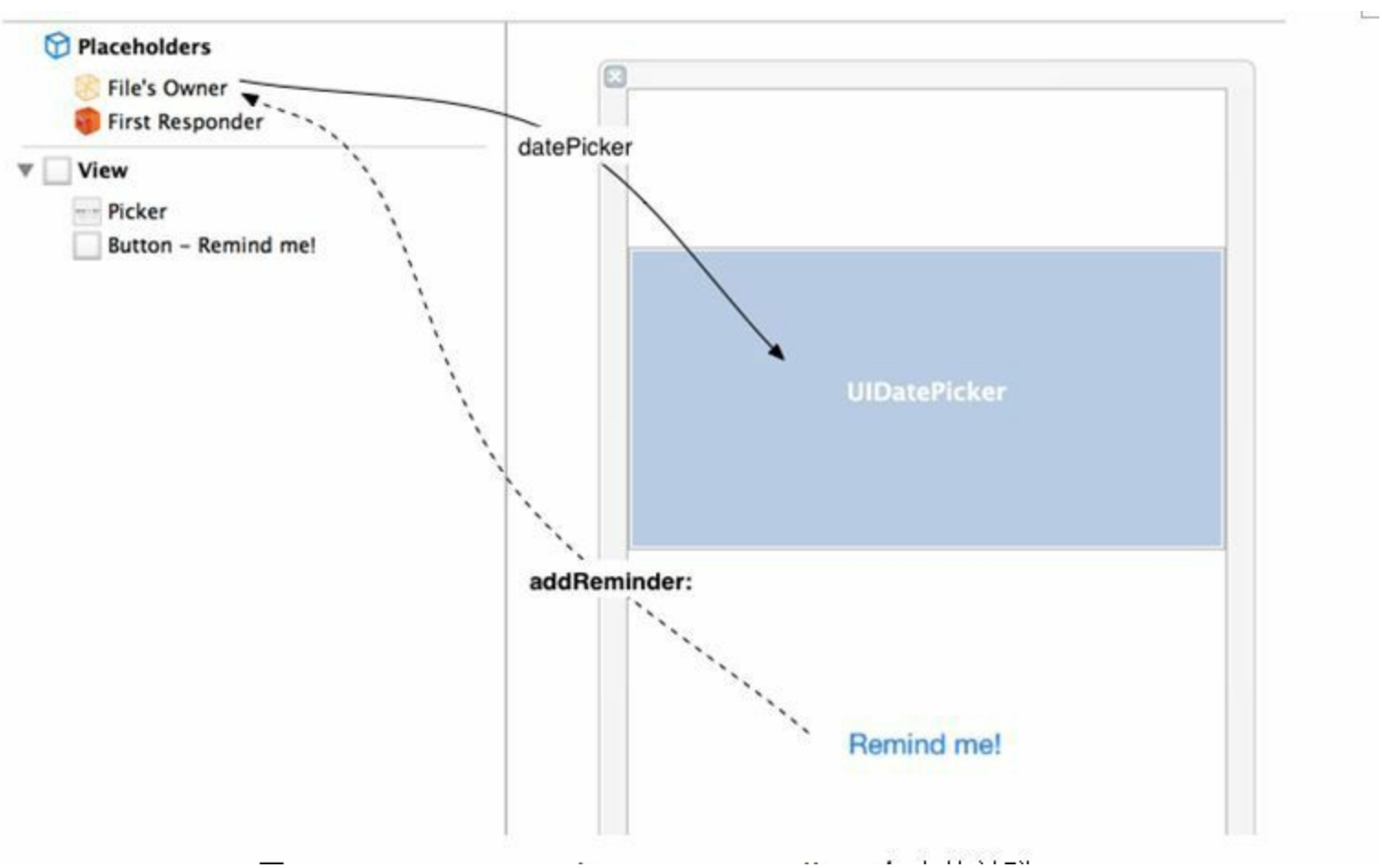


图6-14 BNRReminderViewController.xib中的关联

构建并运行应用，选择一个时间，然后点击Remind Me按钮，并检查控制台输出的时间是否与选择的提醒时间相同。之后的章节中会使用本地通知功能完成addReminder:方法。

之前的代码中将BNRReminderViewController的datePicker插座变量声明为弱引用。将插座变量声明为弱引用是一种编程约定。当系统的可用内存偏少时，视图控制器会自动释放其视图并在之后需要显示时再创建。因此，视图控制器应该使用弱引用特性的插座变量指向view的子视图，以便在释放view时同时释放view的所有子视图，从而避免内存泄漏。

6.3 UITabBarController

当应用为了响应用户的操作，需要将当前显示的视图切换为另一个时，使用视图控制器的好处就越加明显。本书将介绍多种显示视图控制器的途径。本节先介绍如何通过一个UITabBarController对象，使应用能够在BNRHypnosisViewController对象和BNRReminderViewController对象之间自由地切换。

UITabBarController对象可以保存一组视图控制器。此外，UITabBarController对象还会在屏幕底部显示一个标签栏(tab bar)，标签栏会有多个标签项(tab item)，分别对应UITabBarController对象所保存的每一个视图控制器。单击某个标签项，UITabBarController对象就会显示该标签项所对应的视图控制器的视图。

在BNRAppDelegate.m中创建一个UITabBarController对象，将之前创建的两个视图控制器加入该对象，最后将UITabBarController对象设置为UIWindow对象的rootViewController，代码如下：

```
- (BOOL)application:(UIApplication *)application

didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

{

self.window = [[UIWindow alloc] initWithFrame:

        [[UIScreen mainScreen] bounds]];

// 在这里添加应用启动后的初始化代码

BNRHypnosisViewController *hvc = [[BNRHypnosisViewController alloc] init];

// 获取指向NSBundle对象的指针，该对象代表应用的主程序包

NSBundle *appBundle = [NSBundle mainBundle];

// 告诉初始化方法在appBundle中查找BNRReminderViewController.xib文件

BNRReminderViewController *rvc = [[BNRReminderViewController alloc]

initWithNibName:@"BNRReminderViewController"

        bundle:appBundle];

UITabBarController *tabBarController = [[UITabBarController alloc] init];

tabBarController.viewControllers = @[hvc, rvc];

~~self.window.rootViewController = rvc;~~
```

```
    self.window.rootViewController = tabBarController;

    self.window.backgroundColor = [UIColor whiteColor];

    [self.window makeKeyAndVisible];

    return YES;

    }
```

构建并运行应用，底部的标签栏中其实有两个标签项，点击标签栏的左边和右边可以在两个不同的视图控制器之间切换，下一节中会为标签项设置标题和图标，可以让用户清楚地知道每个标签项的功能。

UITabBarController也是UIViewController的子类，也有一个名为view的属性。UITabBarController对象的view指向一个包含两个子视图的UIView对象，分别是标签栏和当前选中的视图控制器的视图（见图6-15）。

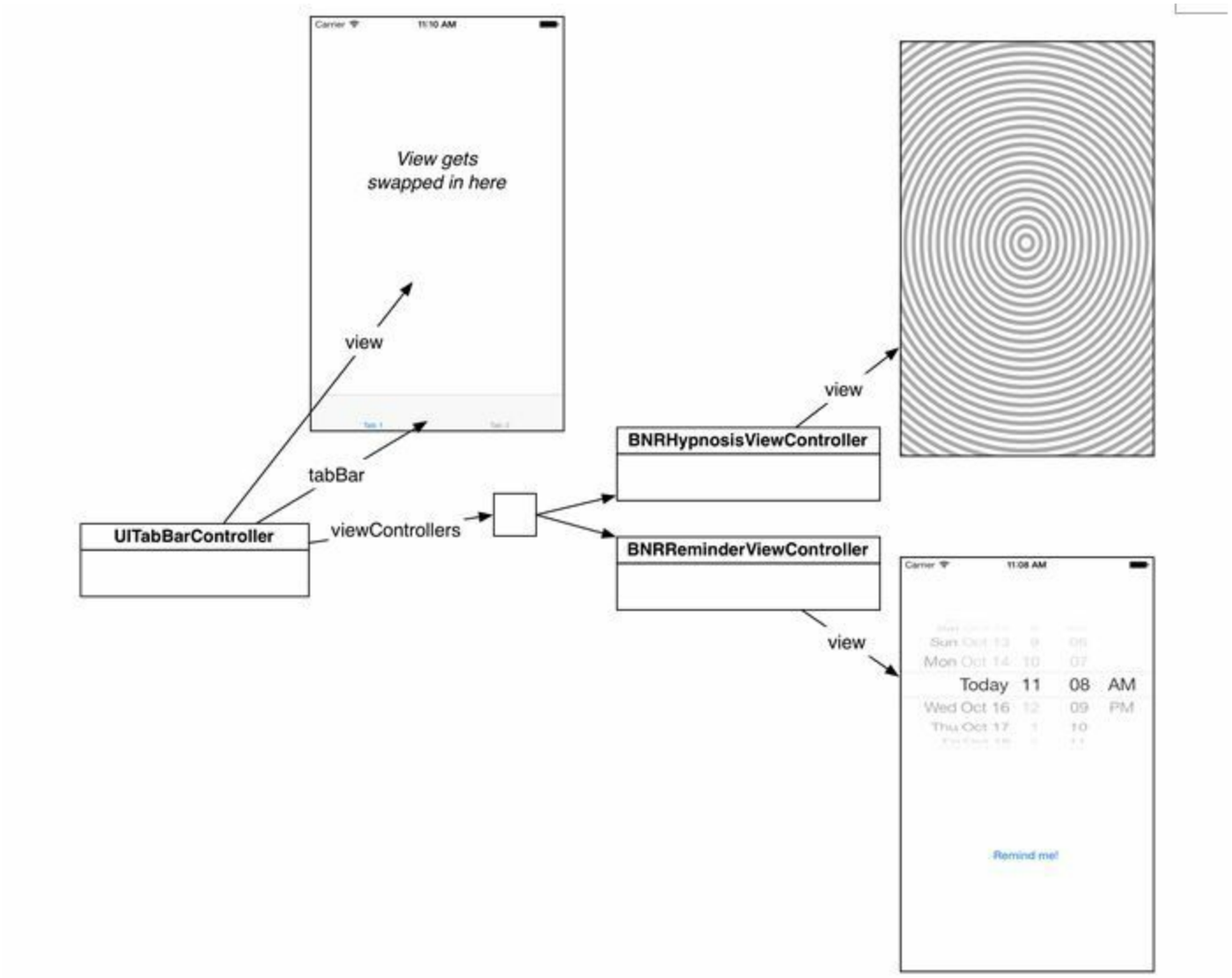


图6-15 UITabBarController的对象图

设置标签项

标签栏上的每一个标签项都可以显示标题和图片，具体数据需要由视图控制器的tabBarItem属性提供。当UITabBarController对象加入一个视图控制器时，就会为标签栏增加一个标签项，并根据新加入的视图控制器的tabBarItem属性设置该标签项的标题和图片。以iPhone自带的电话应用为例，UITabBarController对象和其包含的视图控制器的关系如图6-16所示。

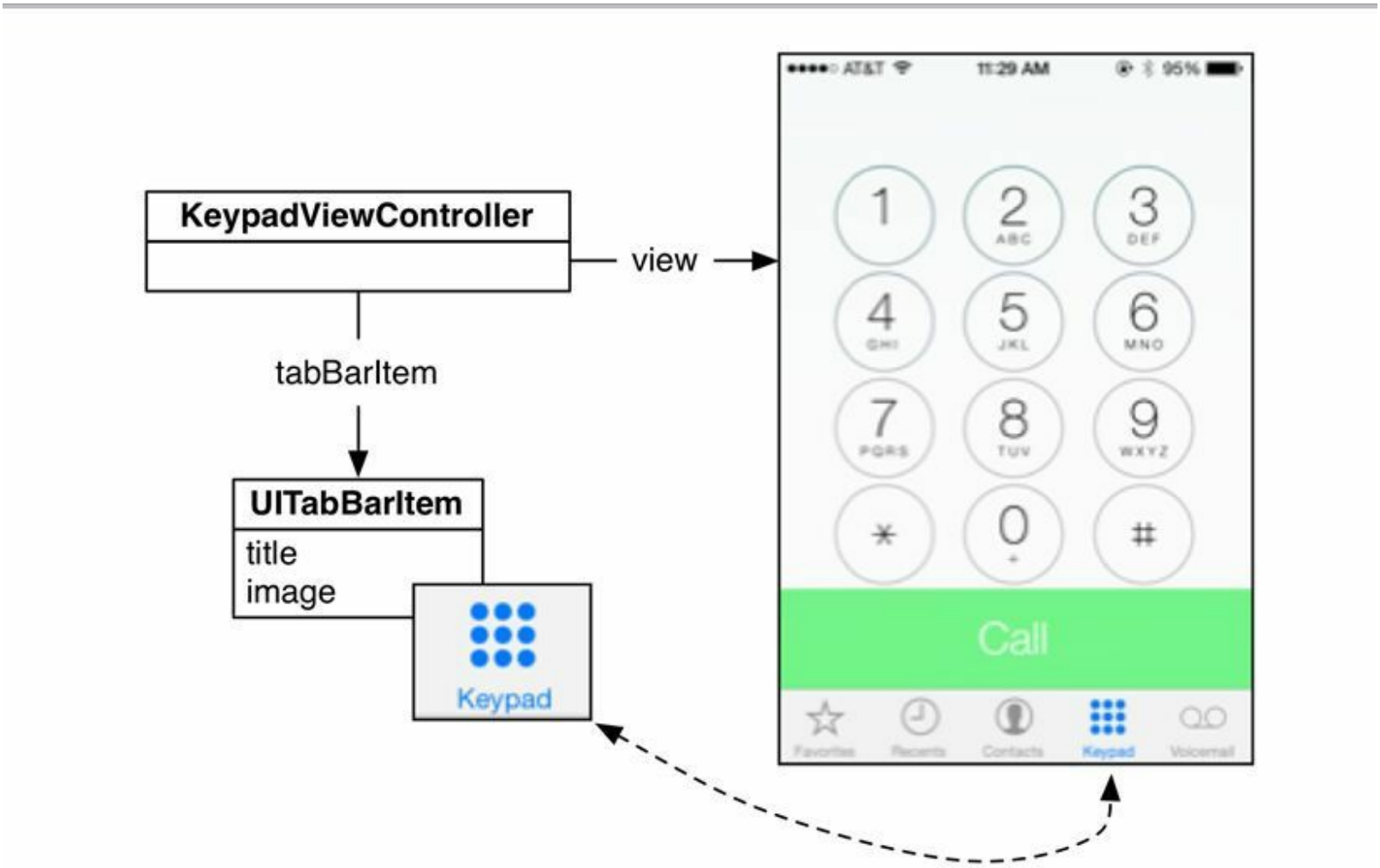


图6-16 UITabBarItem示例

首先，读者需要将标签项的图像文件加入项目中。在项目导航面板打开Images.xcassets，然后打开之前从http://www.bignerdranch.com/solutions/iOSProgramming4ed.zip下载的压缩包，将压缩包中Resources目录下的Hypno.png、Time.png、Hypno@2x.png和Time@2x.png拖曳至资源目录左边的图像文件列表。

接下来打开BNRHypnosisViewController.m，覆盖UIViewController的指定初始化方法initWithNibName:bundle:，设置BNRHypnosisViewController的标签项：

```
- (instancetype)initWithNibName:(NSString *)nibNameOrNil

        bundle:(NSBundle *)nibBundleOrNil
```

```
{
self = [super initWithNibName:nibNameOrNil
                bundle:nibBundleOrNil];
if (self) {
// 设置标签项的标题
self.tabBarItem.title = @"Hypnotize";
// 从图像文件创建一个UIImage对象
// 在Retina显示屏上会加载Hypno@2x.png, 而不是Hypno.png
UIImage *i = [UIImage imageNamed:@"Hypno.png"];
// 将UIImage对象赋给标签项的image属性
self.tabBarItem.image = i;
}
return self;
}
```

打开BNRReminderViewController.m, 重复上述代码:

```
- (instancetype)initWithNibName:(NSString *)nibNameOrNil
                bundle:(NSBundle *)nibBundleOrNil
{
self = [super initWithNibName:nibNameOrNil
            bundle:nibBundleOrNil];
if (self) {
    // 获取tabBarItem属性所指向的UITabBarItem对象
    UITabBarItem *tbi = self.tabBarItem;
    // 设置UITabBarItem对象的标题
```

```
        tbi.title = @"Reminder";
        // 设置UITabBarItem对象的图像
        UIImage *i = [UIImage imageNamed:@"Time.png"];
        tbi.image = i;
    }
    return self;
    }
```

构建并运行应用，可以看到标签栏上有两个带有标题和图标的标签项（见图6-17），用户可以通过标题和图标清楚地知道每个标签项的功能。

图6-17 带标题和图标的标签项

6.4 视图控制器的初始化方法

之前在创建BNRHypnosisViewController的标签项时覆盖了initWithNibName: bundle:，但是，在BNRAppDelegate.m中是使用init方法来初始化BNRHypnosis- ViewController对象的。这是由于initWithNibName:bundle:是UIViewController的指定初始化方法，向视图控制器发送init消息会调用initWithNibName:bundle:方法并为两个参数都传入nil，因此使用init初始化BNRHypnosisViewController对象也可以正确设置视图控制器的标签项。

BNRHypnosisViewController在创建view属性时没有加载NIB文件，因此不需要指定NIB文件的文件名。那么，如果向一个需要使用NIB文件的视图控制器发送init消息，会发生什么情况？下面就来编写代码测试。

在BNRAppDelegate.m中，修改代码，使用init初始化BNRReminderViewController：

```
BNRHypnosisViewController *hvc = [[BNRHypnosisViewController alloc] init];

// 获取指向NSBundle对象的指针，该对象代表应用的主程序包

NSBundle *appBundle = [NSBundle mainBundle];

// 告诉初始化方法在appBundle中查找BNRReminderViewController.xib文件

BNRReminderViewController *rvc = [[BNRReminderViewController alloc]

    initWithNibName:@"BNRReminderViewController"

        bundle:appBundle];

BNRReminderViewController *rvc = [[BNRReminderViewController alloc] init];

UITabBarController *tabBarController = [[UITabBarController alloc] init];
```

构建并运行应用，运行结果会和之前的完全一样。当init调用initWithNibName:bundle:时，虽然为两个参数都传入了nil，但是UIViewController对象仍然会在应用程序包中查找和当前UIViewController子类同名的XIB文件。因此，当BNRReminderViewController需要创建view时，仍然会载入应用程序包中的BNRReminderViewController.xib。

所以之前的章节中建议读者为UIViewController子类和该子类需要载入的NIB文件取相同的名称。这样当视图控制器需要加载视图时，会自动载入正确的XIB文件。

6.5 添加本地通知

接下来使用本地通知(local notification)实现催眠提醒功能。本地通知用于向用户提示一条消息——即使应用没有运行，用户也可以收到本地通知。

应用还可以通过后台服务器实现推送通知(push notification)。有关推送通知的技术细节请参考Apple的Local and Push Notification Programming Guide(本地通知和推送通知编程指南)。

实现本地通知非常简单，首先需要创建一个UILocalNotification对象并设置其显示内容和提醒时间，然后调用UIApplication单例对象的scheduleLocalNotification:方法注册该通知就可以了。

实现addReminder:方法，代码如下：

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

{

NSDate *date = self.datePicker.date;

NSLog(@"Setting a reminder for %@", date);

UILocalNotification *note = [[UILocalNotification alloc] init];

note.alertBody = @"Hypnotize me!";

note.fireDate = date;

[[UIApplication sharedApplication] scheduleLocalNotification:note];

}
```

构建并运行应用，首先选中Reminder标签项；然后选择一个距离现在较近的提醒时间，以便更快收到催眠提醒；最后点击Remind Me按钮。为了看到本地通知，必须关闭HypnoNerd应用，请点击设备底部的Home键或者选择模拟器菜单Hardware→Home(硬件→首页)。当之前选择的提醒时间到了之后，屏幕顶部会出现一个通知栏(见图6-18)，点击通知栏就可以启动HypnoNerd应用。

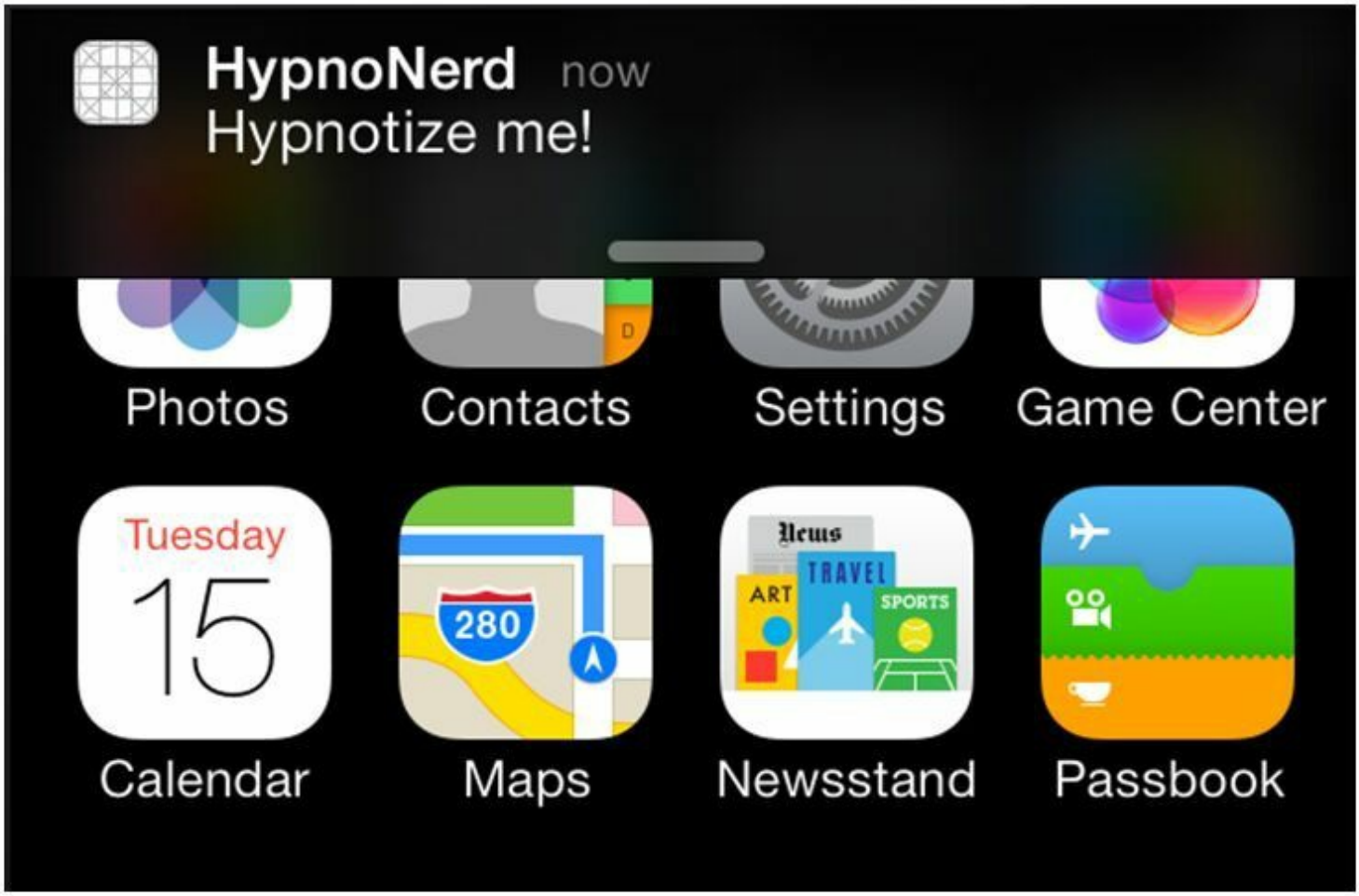


图6-18 本地通知

可能读者已经注意到了一个问题：用户可以选择一个过去的时间。之后的章节会设置datePicker，只允许用户选择未来的时间。

6.6 加载和显示视图

现在HypnoNerd应用中有两个视图控制器，这种情况下之前介绍的视图延迟加载机制显得尤为重要。当应用启动后，标签栏会默认显示第一个视图控制器的视图，这时第二个视图控制器的视图不需要显示，只有当用户点击了第二个视图控制器的标签项，才显示相应的视图。

可以使用视图控制器的viewDidLoad方法检查视图控制器的视图是否已经加载，每个UIViewController对象都实现了viewDidLoad方法，该方法会在载入视图后立刻被调用。

在BNRHypnosisViewController.m中覆盖viewDidLoad方法，向控制台输出一条提示信息，代码如下：

```
- (void)viewDidLoad

{

// 必须调用父类的viewDidLoad

[super viewDidLoad];

NSLog(@"BNRHypnosisViewController loaded its view.");

}
```

同样，在BNRReminderViewController.m中覆盖viewDidLoad方法，代码如下：

```
- (void)viewDidLoad

{

[super viewDidLoad];

NSLog(@"BNRReminderViewController loaded its view.");

}
```

构建并运行应用，控制台应该会输出BNRHypnosisViewController对象载入其视图的提示信息。单击对应BNRReminderViewController对象的标签项，控制台才会输出该对象载入其视图的提示信息。现在两个视图控制器的视图都已加载完毕，继续切换标签项不会再次触发视图控制器的viewDidLoad方法，所以控制台不会再有任何输出（请读者尝试并检查结果）。

为了实现视图延迟加载，在initWithNibName:bundle:中不应该访问view或view的任何子视图。凡是和view或view的子视图有关的初始化代码，都应该在viewDidLoad方法中实现，避免加载不需要在屏幕上显示的视图。

访问视图

通常情况下，在用户看到XIB文件中创建的视图之前需要对它们做一些额外的初始化工作。但是，关于视图的初始化代码不能写在视图控制器的初始化方法中——此时视图控制器并未加载NIB文件，所有指向视图的属性都是nil。如果向这些属性发送消息，虽然编译时不会报错，但是运行时无法对这些属性做任何操作。

那么，应该在哪个方法中访问XIB文件中的视图呢？主要包括两个方法，可以根据实际需要选择。第一个方法是用于确认视图已经加载的viewDidLoad，该方法会在视图控制器加载完NIB文件之后被调用，此时视图控制器中所有视图属性都已经指向了正确的视图对象。第二个方法是viewWillAppear:，该方法会在视图控制器的view添加到应用窗口之前被调用。

两个方法的区别是，如果只需要在应用启动后设置一次视图对象，就选择viewDidLoad；如果用户每次看到视图控制器的view时都需要对其进行设置，则选择viewWillAppear:。

BNRReminderViewController的view有一个子视图datePicker，在用户看到该子视图之前需要对其做额外设置。目前，用户可以选择一个已经过去的时间作为提醒时间，因此需要设置datePicker，只允许用户选择一个距离现在至少60秒以后的时间。

因为用户每次看到BNRReminderViewController的view时都更新datePicker，所以应该覆盖viewWillAppear:。打开BNRReminderViewController.m，在viewWillAppear:中设置datePicker的minimumDate属性（可供选择的最小时间）。

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

{

[super viewWillAppear:animated];

self.datePicker.minimumDate = [NSDate dateWithTimeIntervalSinceNow:60];

}
```

构建并运行应用，选择Reminder标签项，现在用户只能选择一个未来时间了。

相反，如果覆盖的是viewDidLoad而不是viewWillAppear:，则BNRReminderView- Controller只会在用户第一次看到datePicker时设置minimumDate，之后在应用运行的过程中，minimumDate属性会保持不变。随着应用运行时间越来越长，用户很快就能够选择已经过去的时间。

viewWillAppear方法中的animated参数用于设置是否使用视图显示或消失的过渡动画。UITabBarController不会显示过渡动画，第10章中将会介绍UINavigation- Controller，它会在视图控制器被推入和推出屏幕时使用过渡动画。

6.7 与视图控制器及其视图进行交互

现在请看视图控制器的生命周期方法(lifecycle method)，其中一部分方法读者之前遇到过：

•application:didFinishLaunchingWithOptions在该方法中设置和初始化应用窗口的根视图控制器。该方法只会在应用启动完毕后调用一次，之后如果从其他应用切换回本应用，则该方法不会再次被调用。如果关闭应用后台进程(双击Home键可以打开后台进程栏)并重新启动应用，该方法才会再次被调用。

•initWithNibName:bundle该方法是UIViewController的指定初始化方法，创建视图控制器时，就会调用该方法。请注意，某些情况下，需要在同一个应用中创建多个相同的UIViewController子类对象，每次创建一个该类的对象时，都会调用一次该类的initWithNibName:bundle:方法。

•loadView可以覆盖该方法，使用代码方式设置视图控制器的view属性。

•viewDidLoad可以覆盖该方法，设置使用NIB文件创建的视图对象。该方法会在视图控制器加载完视图后被调用。

•viewWillAppear可以覆盖该方法，设置使用NIB文件创建的视图对象。该方法和viewDidAppear:会在每次视图控制器的view显示在屏幕上时被调用；相反，viewWillDisappear:和viewDidDisappear:方法会在每次视图控制器的view从屏幕上消失时被调用。因此，如果打开HypnoNerd应用并在Hypnosis和Reminder两个标签项之间来回切换，那么BNRReminderViewController的viewDidLoad方法只会被调用一次，而viewWillAppear:方法会被调用很多次。

6.8 初级练习：增加一个标签项

使用第1章创建的Quiz项目为UITabBarController对象添加第三个标签项，让用户可以回答问题并查看答案。

6.9 中级练习：控制逻辑

将一个UISegmentedControl对象（分段控件）加入BNRHypnosisViewController对象的视图，UISegmentedControl对象要包含三个分段，分别对应红、绿和蓝三种颜色。当用户单击某个分段时，将BNRHypnosisView中圆形的轮廓改为相应的颜色。注意：在练习前先拷贝项目，然后在拷贝后的项目中修改代码。

6.10 深入学习：键值编码

当NIB文件被载入之后，其中的插座变量是通过键值编码（Key-value coding，KVC）来设置的。键值编码是通过一系列定义在NSObject中的方法实现的，使用这些方法可以通过属性的名称存取属性的值，以下是其中的两种方法：

```
- (id)valueForKey:(NSString *)k;

- (void)setValue:(id)v forKey:(NSString *)k;
```

valueForKey和setValue:forKey:是通用的属性存取方法，可以获取或设置任意对象的属性。例如，selectedObj对象有一个fido属性，那么可以使用如下代码获取或设置fido属性：

```
id currentFido = [selectedObj valueForKey:@"fido"];

[selectedObject setValue:userChoice forKey:@"fido"];
```

如果selectedObj中定义了fido属性的存取方法，那么valueForKey:和setValue:forKey:就会调用相应的存取方法；如果没有存取方法，则会查找名为_fido或fido的实例变量并直接设置或返回实例变量的值；如果既没有相应的存取方法，也没有_fido或fido实例变量，就会抛出异常。

NIB文件在加载时会使用setValue:forKey:设置插座变量。例如，如果需要为某个对象设置一个名为rex的插座变量，那么该对象必须定义了setRex:方法或者具有_rex实例变量（或rex实例变量），否则在加载NIB文件时会抛出异常，类似以下提示：

```
[<BNRSunsetViewController 0x68c0740> setValue:forUndefinedKey:]:

this class is not key value coding-compliant for the key rex.'
```

如果读者看到了这类异常提示，则很有可能是因为在Interface Builder中关联好插座变量之后又更改了该插座变量的名称。

由此本节传达出一条最重要的编程法则：请务必遵守存取方法命名规范。其目的不仅仅是方便其他开发者阅读代码，系统也具有一套依赖于命名规范的工作机制，如果不遵守命名规范，很有可能会发生意外错误。

下面来看一个例子。某个视图控制器定义了clock插座变量，指向一个表示时钟的视图；同时，它还作为一个按钮的目标，为其定义了动作方法setClock:。该方法用于获取网络最新时间并更新时钟视图，方法声明如下：

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

这样就会产生一个奇怪的问题：当NIB文件被加载时，该方法会立即执行，同时也无法正确设置clock插座变量。原因是系统会使用setClock:动作方法设置clock——系统会将setClock:视为clock的存方法。因此需要重新命名该动作方法，例如updateClock:。

如果不遵守命名规范，就可能在这类奇怪的问题上浪费大量时间。

6.11 深入学习：Retina显示屏

随着iPhone 4的发布，Apple公司推出了针对iPhone和iPod Touch的Retina显示屏。Retina显示屏拥有很高的分辨率——其中4英寸屏幕是640像素×1136像素，3.5英寸屏幕是640像素×960像素（之前的屏幕分辨率是320像素×480像素）。下面介绍如何让图像在两种设备上都有最好的显示效果。

对于矢量图像，不用做任何处理就能在两种设备上有最好的显示效果。以BNRHypnosisView的drawRect:和其绘制的文字为例，只需要编写同一段代码，就能让视图以当前设备允许的最大分辨率渲染图像。如果应用是通过Core Graphics函数绘图的，那么画出的图像在不同的设备上会有不同的显示效果。Core Graphics以点为单位描述线、曲线和文字等。对于非Retina显示屏，1个点是1个像素。对于Retina显示屏，1个点是2像素×2像素（见图6-19）。

图6-19 不同分辨率下的绘图效果

鉴于以上这些差别，如果位图图片（例如JPEG或PNG文件）不适合当前设备的屏幕类型，显示效果就会大打折扣。例如，假设某个应用有一个25像素×25像素的小图片，并且要画出的图片尺寸是25点×25点（注意，这里的单位是点）。在Retina显示屏上显示该图片时，系统必须拉伸该图片才能覆盖50像素×50像素的区域。这时系统会用一种称为抗锯齿（anti-aliasing）的平均算法，使放大后的图片看上去更圆滑。处理后的图片虽然圆滑，但是看上去较模糊（见图6-20）。

图6-20 图片放大后变模糊了

也不能使用更大的图片：为非Retina屏幕缩小图片时，平均算法一样会产生问题。唯一的解决方案是在应用程序包里放入两套图：一套针对非Retina显示屏，像素分辨率和相应的屏幕点数相同；另一套针对Retina显示屏，像素尺寸比前一套大1倍。

好在不需要编写任何额外的代码，应用就能根据当前的设备载入相应版本的图片。开发者唯一需要做的就是为高分辨率的图片文件加上后缀名@2x。当应用使用UIImage的imageNamed:载入图片时，imageNamed:会在程序包里查找并获取适合特定设备的文件。

第7章 委托与文本输入

本章介绍委托(delegation)和UITextField的使用。委托是Cocoa Touch中的一种常见设计模式，而UITextField是常用的文本输入控件。此外，还会介绍如何使用Xcode调试器，找出并修正代码中的问题。

本章将继续完善HypnoNerd应用，用户可以在界面中输入并显示催眠信息，如图7-1所示。

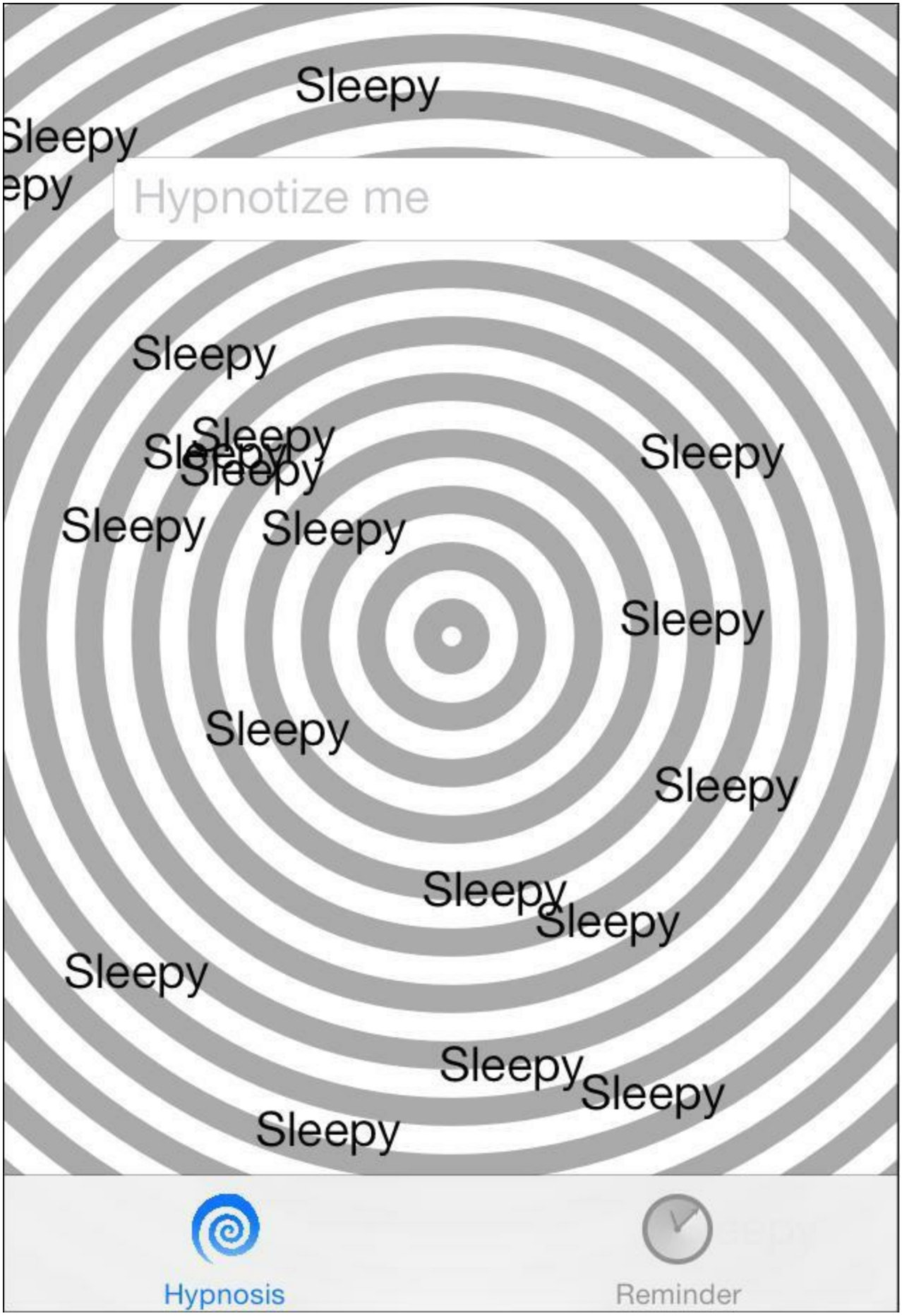


图7-1 完成后的HypnoNerd

7.1 文本框（UITextField）

首先打开第6章创建的HypnoNerd应用。

之前介绍过UILabel，它可以用来在界面中显示文本，但是用户无法选择或编辑UILabel中的文本。相反，UITextField可以接受用户输入的文本，例如在登录界面中，如果需要让用户输入用户名和密码，就可以使用UITextField。

打开BNRHypnosisViewController.m，修改loadView方法，向view中添加一个UITextField对象：

```
- (void)loadView

{

CGRect frame = [UIScreen mainScreen].bounds;

BNRHypnosisView *backgroundView =

[[BNRHypnosisView alloc] initWithFrame:frame];

CGRect textFieldRect = CGRectMake(40, 70, 240, 30);

UITextField *textField = [[UITextField alloc] initWithFrame:textFieldRect];

// 设置UITextField对象的边框样式，便于查看它在屏幕上的位置

textField.borderStyle = UITextBorderStyleRoundedRect;

[backgroundView addSubview:textField];

self.view = backgroundView;

}
```

构建并运行应用，Hypnotize标签页会显示一个文本框，该文本框就是刚才添加的UITextField对象。点击文本框，这时屏幕底部会弹出键盘，用于向文本框中输入文字。为了理解UITextField对象对点击事件的响应过程，下面需要介绍第一响应者（first responder）的概念。

UIResponder

UIResponder是UIKit框架中的一个抽象类。之前章节介绍过它的几个子类：

- UIView

- UIViewController
- UIApplication

UIResponder定义了一系列方法，用于接收和处理用户事件，例如触摸事件、运动事件（如摇晃设备）和功能控制事件（如编辑文本或播放音乐）等。UIResponder的子类会覆盖这些方法，实现自己的事件响应代码。

在以上事件中，触摸事件显然应该由被触摸的视图负责处理。系统会将触摸事件直接发送给被触摸的视图，第5章介绍过触摸事件的处理方法。

其他类型的事件则会由第一响应者负责处理，UIWindow有一个firstResponder属性指向第一响应者。例如，当用户点击UITextfield对象时，UITextfield对象就会成为第一响应者。UIWindow会将firstResponder指向该对象，之后，如果应用接收到运动事件和功能控制事件，都会发送给UITextfield对象（见图7-2）。

图7-2 第一响应者

当某个UITextField对象或UITextView对象成为第一响应者时，屏幕会弹出键盘。除了用户点击之外，还可以在代码中向UITextField对象发送becomeFirstResponder消息，使其成为第一响应者。相反，如果要关闭键盘，则可以向UITextField对象发送resignFirstResponder消息，且要求该对象放弃第一响应者状态。一旦第一响应者不是UITextField对象，键盘就会消失。

实际上，大部分视图都不需要成为第一响应者。例如UISlider对象，该对象只处理触摸事件（用户拖曳滑块），而不会接受其他类型的事件，因此它不需要成为第一响应者。

设置UITextField的键盘

UITextField对象有一系列属性，用于设置弹出的键盘。下面就修改这些属性，为UITextField对象添加占位符文本，并修改键盘的换行键类型。

```
- (void)loadView

{
```

```
CGRect frame = [UIScreen mainScreen].bounds;

BNRHypnosisView *backgroundView =

[[BNRHypnosisView alloc] initWithFrame:frame];

CGRect textFieldRect = CGRectMake(40, 70, 240, 30);

UITextField *textField = [[UITextField alloc] initWithFrame:textFieldRect];

// 设置UITextField对象的边框样式，便于查看它在屏幕上的位置

textField.borderStyle = UITextBorderStyleRoundedRect;

textField.placeholder = @"Hypnotize me";

textField.returnKeyType = UIReturnKeyDone;

[backgroundView addSubview:textField];

self.view = backgroundView;

}
```

构建并运行应用，现在UITextField对象中有一行占位符文本Hypnotize me，当用户在UITextField对象中输入文字时，占位符文本就会消失。同时，换行键不再显示默认的Return，而是Done（见图7-3）。

图7-3 UIReturnKeyDone类型的键盘

但是，如果读者点击Done键，会发现应用没有任何反应。实际上，修改换行键的类型只是改变了换行键的外观，如果需要实现用户点击换行键后的功能，必须编写相应代码。在编写代码

之前，再介绍UITextField对象中另外几个有用的属性：

autocapitalizationType	设置 UITextField 对象的自动大写功能，包括 none（关闭自动大写）、words（单词）、sentences（句子）、all characters（所有字母）四种类型
autocorrectionType	启用 / 禁用（设置为 YES / NO，下同）UITextField 对象的拼写建议功能，可以提示用户输错的单词并提供修改建议
enablesReturnKeyAutomatically	启用 / 禁用 UITextField 对象的换行键自动监测功能。如果将该属性设置为 YES，UITextField 对象会自动监测用户输入，并根据是否输入了文字启用 / 禁用换行键
keyboardType	设置 UITextField 对象弹出的键盘类型，例如 ASCII Capable（ASCII 标准键盘）、E-mail Address（电子邮件地址）、Number Pad（数字键盘）和 URL（网址）
secureTextEntry	启用 / 禁用 UITextField 对象的安全输入功能。如果将该属性设置为 YES，UITextField 对象会以圆点代替用户输入的文字，类似于通常见到的密码文本框

7.2 委托

之前章节介绍过目标-动作(Target-Action)设计模式。目标-动作是UIKit中常用的设计模式之一，第1章在编写Quiz应用时，曾经针对UIButton对象使用过这种设计模式。目标-动作的工作方式为：当某个特定的事件发生时(例如按下按钮)，发生事件的一方会向指定的目标对象发送一个之前设定好的动作消息。

在目标-动作中，针对不同的事件，需要创建不同的动作消息。UIButton对象的事件比较简单，通常只需要处理点击事件；相反，像UITextField这类事件复杂的对象，Apple使用委托设计模式。UITextField对象具有一个委托属性，通过为UITextField对象设置委托，UITextField对象会在发生事件时向委托发送相应的消息，由委托处理该事件。例如，对于编辑UITextField对象文本内容的事件，有以下两个对应的委托方法：

```
- (void)textFieldDidEndEditing:(UITextField *)textField;

- (void)textFieldDidBeginEditing:(UITextField *)textField;
```

还有一类带有返回值的委托方法，用于从委托中查询需要的信息，例如，

```
- (BOOL)textFieldShouldEndEditing:(UITextField *)textField;

- (BOOL)textFieldShouldBeginEditing:(UITextField *)textField;

- (BOOL)textFieldShouldClear:(UITextField *)textField;

- (BOOL)textFieldShouldReturn:(UITextField *)textField;
```

注意，在委托方法中，通常应该将对象自身作为第一个参数。多个对象可能具有相同的委托，当委托收到消息时，需要根据该参数判断发送该消息的对象。例如，如果某个视图控制器中包含多个UITextField对象，它们的委托都是该视图控制器，那么视图控制器就需要根据textField参数获取相应的UITextField对象并执行不同的操作。

下面就将UITextField对象所位于的视图控制器——BNRHypnosisViewController设置为它的委托，并实现textFieldShouldReturn:委托方法，当用户点击Done按钮时，UITextField对象就会调用该方法。

打开BNRHypnosisViewController.m，修改loadView方法，将UITextField对象的委托属性设置为BNRHypnosisViewController自身。

```
- (void)loadView

{

CGRect frame = [UIScreen mainScreen].bounds;

BNRHypnosisView *backgroundView =
```

```
    [[BNRHypnosisView alloc] initWithFrame:frame];

    CGRect textFieldRect = CGRectMake(40, 70, 240, 30);

    UITextField *textField = [[UITextField alloc] initWithFrame:textFieldRect];

    // 设置UITextField对象的边框样式，便于查看它在屏幕上的位置

    textField.borderStyle = UITextBorderStyleRoundedRect;

    textField.placeholder = @"Hypnotize me";

    textField.returnKeyType = UIReturnKeyDone;

    // 这里Xcode会提示一处警告信息，下一节将介绍原因并消除该警告

    textField.delegate = self;

    [backgroundView addSubview:textField];

    self.view = backgroundView;

    }
```

textFieldShouldReturn:只有一个参数，就是用户点击换行键的相应UITextField对象。目前，应用只会向控制台输出UITextField对象的文本内容。

接下来在BNRHypnosisViewController.m中实现textFieldShouldReturn:方法。建议读者从书中复制方法声明，确保方法与UITextField对象的委托方法完全一致，否则UITextField对象不会调用该方法。

```
    - (BOOL)textFieldShouldReturn:(UITextField *)textField

    {

    NSLog(@"%@", textField.text);

    return YES;

    }
```

构建并运行应用，在UITextField对象中输入一些文字，然后点击Done，这时控制台中会输出这些文字。

请注意，BNRHypnosisViewController不需要实现UITextField对象的所有委托方法，UITextField对象会在运行时检查委托是否实现了某个方法，如果没有实现，UITextField对象就不会调用该方法。

7.3 协议

凡是支持委托的对象，其背后都有一个相应的协议（如果读者之前接触过Java或C#，这些语言中的“协议”称为“接口”），声明可以向该对象的委托对象发送的消息。委托对象需要根据这个协议为其“感兴趣”的事件实现相应的方法。如果一个类实现了某个协议中规定的方法，就称这个类遵守（conform）该协议。

针对UITextField的委托对象，相应的协议示例代码如下：

```
@protocol UITextFieldDelegate <NSObject>

@optional

- (BOOL)textFieldShouldBeginEditing:(UITextField *)textField;

- (void)textFieldDidBeginEditing:(UITextField *)textField;

- (BOOL)textFieldShouldEndEditing:(UITextField *)textField;

- (void)textFieldDidEndEditing:(UITextField *)textField;

- (BOOL)textField:(UITextField *)textField

    shouldChangeCharactersInRange:(NSRange)range

        replacementString:(NSString *)string;

- (BOOL)textFieldShouldClear:(UITextField *)textField;

- (BOOL)textFieldShouldReturn:(UITextField *)textField;

@end
```

声明协议的语法是，使用@protocol指令开头，后跟协议的名称（例如UITextFieldDelegate）。尖括号里的NSObject是指NSObject协议，其作用是声明UITextFieldDelegate包含NSObject协议的全部方法。接着声明新协议特有的方法，最后使用@end指令结束。

协议不是类，只是一组方法声明。不能为协议创建对象，或者添加实例变量。协议自身不实现方法，需要由遵守相应协议的类来实现。

UITextFieldDelegate协议来自iOS SDK，Xcode文档会详细介绍iOS SDK中的协议，列出协议中的所有方法。此外，读者也可以编写自己的协议，第22章会作相应的介绍。

协议所声明的方法可以是必需的（required）或是可选的（optional）。协议方法默认都是必需的。使用@optional指令，可以将写在该指令之后的方法全部声明为可选的。以之前列出的UITextFieldDelegate协议为例，可以发现该协议的所有方法都是可选的。委托协议中的方法通常都是可选的。

发送方在发送可选方法前，会先向其委托发送另一个名为respondsToSelector:的消息。所有Objective-C对象都从NSObject继承了respondsToSelector:方法，该方法能在运行时检查对象是否实现了指定的方法。@selector()指令可以将选择器(selector)转换成数值，以便将其作为参数进行传递。例如，UITextField可以实现如下方法：

```
- (void)clearButtonTapped

{

// textFieldShouldClear:是可选方法，需要先检查委托是否实现了该方法

SEL clearSelector = @selector(textFieldShouldClear:);

if ([self.delegate respondsToSelector:clearSelector]) {

    if ([self.delegate textFieldShouldClear:self]) {

        self.text = @"";

    }

}

}
```

如果某个协议方法是必需的，那么发送方可以直接向其委托对象发送相应的消息，不用检查委托对象是否实现了该方法。这意味着，如果委托对象没有实现相应的方法，应用就会抛出未知选择器(unrecognized selector)异常，导致应用崩溃。

为了防止发生此类问题，编译器会检查某个类是否实现了相关协议的必需方法。要让编译器能够执行此类检查，必须将相应的类声明为遵守指定的协议，其语法格式为：在头文件或类扩展的@interface指令末尾，将类所遵守的协议以逗号分隔的列表形式写在尖括号里。

在BNRHypnosisViewController.m的类扩展中，将BNRHypnosisViewController声明为遵守UITextFieldDelegate协议，代码如下：

```
@interface BNRHypnosisViewController()<UITextFieldDelegate>

@end
```

再次构建应用，因为已将BNRHypnosisViewController声明为了遵守UITextFieldDelegate协议，所以编译器不会再针对“textField.delegate = self”这行代码发出警告。此外，如果要为BNRHypnosisViewController实现UITextFieldDelegate协议中的其他方法，那么在输入这些方法的方法名时，Xcode会自动提示完整的方法名或提供备选列表(auto-completed功能)。

读者可能会注意到，iOS SDK中很多类都具有委托，而几乎所有的委托都是弱引用属性。这是为了避免对象及其委托之间产生强引用循环(见图7-4)。例如，BNRHypnosisViewController

是UITextfield对象的委托，而且UITextfield对象是BNRHypnosisViewController的强引用属性，如果UITextfield对象再对其委托保持强引用，就会在两者之间产生强引用循环，很可能造成内存泄露。

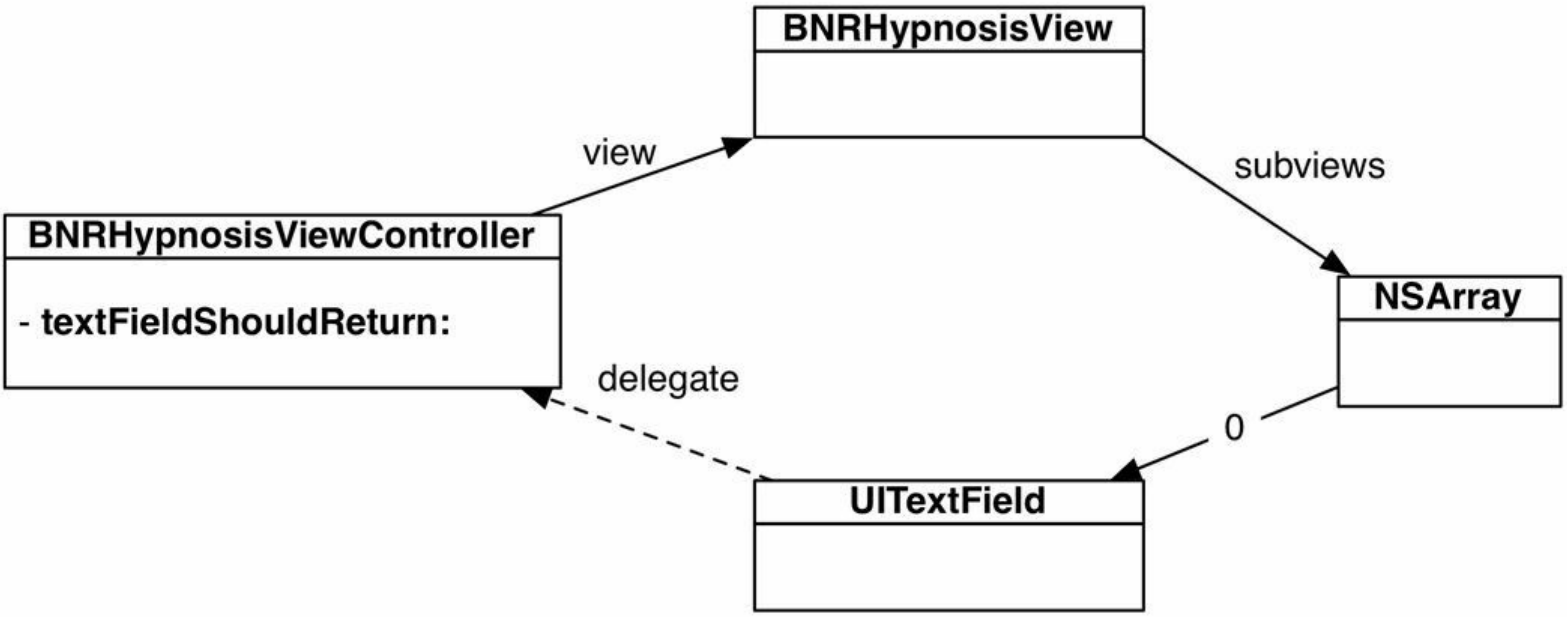


图7-4 避免强引用循环

7.4 向屏幕中添加UILabel对象

为了增强HypnoNerd的催眠效果，本节将添加一些出现在屏幕随机位置的UILabel对象。

在BNRHypnosisViewController.m中添加一个新方法，在屏幕随机位置绘制20个UILabel对象。同时，该方法有一个NSString类型的参数，表示UILabel对象显示的文字。

```
- (void)drawHypnoticMessage:(NSString *)message
{
for (int i = 0; i < 20; i++) {
    UILabel *messageLabel = [[UILabel alloc] init];
    // 设置UILabel对象的文字和颜色
    messageLabel.backgroundColor = [UIColor clearColor];
    messageLabel.textColor = [UIColor whiteColor];
    messageLabel.text = message;
    // 根据需要显示的文字调整UILabel对象的大小
    [messageLabel sizeToFit];
    // 获取随机x坐标,
    // 使UILabel对象的宽度不超出BNRHypnosisViewController的view宽度
    int width = (int)(self.view.bounds.size.width –
        messageLabel.bounds.size.width);
    int x = arc4random() % width;
    // 获取随机y坐标,
    // 使UILabel对象的高度不超出BNRHypnosisViewController的view高度
    int height = (int)(self.view.bounds.size.height –
        messageLabel.bounds.size.height);
    int y = arc4random() % height;
```

```
        // 设UILabel对象的frame
        CGRect frame = messageLabel.frame;
        frame.origin = CGPointMake(x, y);
        messageLabel.frame = frame;
        // 将UILabel对象添加到BNRHypnosisViewController的view中
        [self.view addSubview:messageLabel];
        }
    }
```

接下来修改textFieldShouldReturn:，将UITextField对象的文本内容作为message参数。调用drawHypnoticMessage:；再清空文本内容并调用resignFirstResponder关闭键盘。

```
    - (BOOL)textFieldShouldReturn:(UITextField *)textField
    {
    NSLog(@"%@", textField.text);
    [self drawHypnoticMessage:textField.text];
    textField.text = @"";
    [textField resignFirstResponder];
    return YES;
    }
```

构建并运行应用，在UITextField对象中输入一些文字，然后点击Done，这时输入的文字会随机出现在屏幕中的各个位置。

7.5 运动效果

ios设备内嵌了许多功能强大的传感器，例如加速传感器，磁场传感器和三轴陀螺仪等。应用可以通过这些感应器了解设备的速度、方向和角度，并实现有用的功能。例如，应用可以根据设备的方向自动将界面调整为横排模式或竖排模式。从iOS 7开始，Apple引入了一些新API，可以轻松为应用添加一种通过感应器实现的视差(parallax)效果。

读者可以想象自己坐在一辆飞驰的车中，这时向车窗外望去，会发现远处景物的倒退速度比近处的要慢很多。这是大脑对空间和速度差异产生的一种错觉，称为视差。在iOS 7中，视差效果随处可见，例如，在主屏幕中，如果稍微倾斜设备，可以发现主屏幕中的图标会随着倾斜方向相对于壁纸移动；在iOS系统其他部分和应用中，包括红色的通知标识、音量调节指示器和警告视图等，都带有一定程度的视差效果。

应用可以通过UIInterpolatingMotionEffect类实现相同的效果，只需要创建一个UIInterpolatingMotionEffect对象，设置其方向(垂直或水平)、键路径(key path，需要使用视差效果的属性)和相对最小/最大值(视差的范围)，再将其添加到某个视图上，该视图就能获得相应的视差效果。

在BNRHypnosisViewController.m中，修改drawHypnoticMessage:方法，为UILabel对象分别添加水平方向和垂直方向的视差效果，使UILabel对象的中心点坐标在每个方向上最多移动25点。

```
[self.view addSubview:messageLabel];

UIInterpolatingMotionEffect *motionEffect;

motionEffect = [[UIInterpolatingMotionEffect alloc]

initWithKeyPath:@"center.x"

        type:UIInterpolatingMotionEffectTypeTiltAlongHorizontalAxis];

motionEffect.minimumRelativeValue = @(-25);

motionEffect.maximumRelativeValue = @(25);

[messageLabel addMotionEffect:motionEffect];

motionEffect = [[UIInterpolatingMotionEffect alloc]

initWithKeyPath:@"center.y"

    type:UIInterpolatingMotionEffectTypeTiltAlongVerticalAxis];

motionEffect.minimumRelativeValue = @(-25);

motionEffect.maximumRelativeValue = @(25);
```

```
    [messageLabel addMotionEffect:motionEffect];

}
```

为了测试运动效果，必须在真实iOS设备中运行应用。选择一个设备，构建并运行应用，向屏幕中添加一些UILabel对象，然后稍微倾斜屏幕，可以注意到UILabel对象会随着倾斜方向移动，具有逼真的视差效果。

7.6 使用调试器

使用Xcode启动应用时，调试器会被附着在应用上。调试器会监视应用当前的状态，例如应用当前正在执行的方法和该方法能够访问的变量数值。调试器能够帮助我们了解应用当前正在做什么，进而找到并修正错误。

使用断点

使用调试器的途径之一是设置断点。为某行代码设置断点后，应用会在运行该行代码前暂停执行。接着可以让应用逐行运行余下的代码。当应用没有按预期工作时，可以通过断点和逐行运行找出问题。

在项目导航面板中选中BNRHypnosisView.m（注意，不要错选为BNRHypnosisViewController.m），找到initWithFrame:中设置circleColor属性的那行代码。单击这行代码左侧的空白区域（位于编辑器区域左侧的浅色长条，见图7-5），Xcode就会在该行添加一个蓝色的指示器，表示为该行代码设置了断点。

```
18
19  - (id)initWithFrame:(CGRect)frame
20  {
21      self = [super initWithFrame:frame];
22      if (self) {
23          self.backgroundColor = [UIColor clearColor];
24          self.circleColor = [UIColor lightGrayColor];
25      }
26      return self;
27  }
```

图7-5 断点示例

构建并运行应用，当应用执行到设置了断点的那一行代码时，就会暂停执行。新出现的绿色指示器（和断点位于同一行）会显示应用当前的运行位置。

这时可以进一步查看HypnoNerd的运行状况。单击导航面板区域中的图标，打开调试导航面板，面板中会显示应用在断点处的栈跟踪信息。当应用暂停时，凡是其栈帧在栈中的方法和函数都会出现在栈跟踪信息中。通过调试导航面板底部的滑动条，可以展开或收起栈跟踪信息。将滑块拖曳至最右端，可以显示栈跟踪信息中的全部方法（见图7-6）。

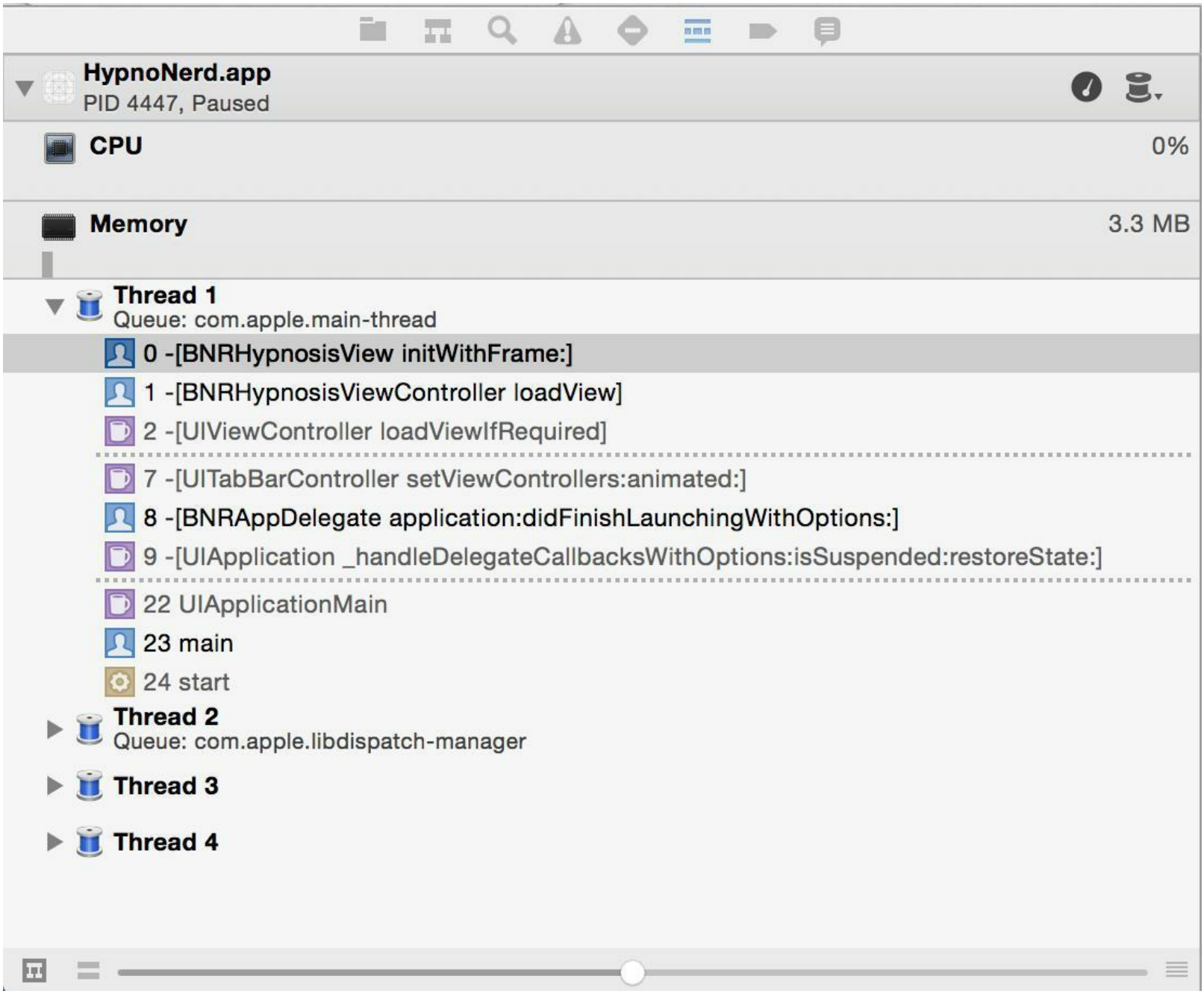


图7-6 调试导航面板

栈跟踪信息顶部会显示断点处的方法，位于该方法下方的是调用该方法的方法，依此类推。之前读者自己实现的两个方法呈黑色，系统库实现的方法呈灰色。

选中位于栈顶的方法。位于编辑器区域下方，控制台左侧的是变量视图。变量视图会针对当前选中的方法（BNRHypnosisView的initWithFrame:），显示其作用域内的变量和这些变量的当前数值（见图7-7）。

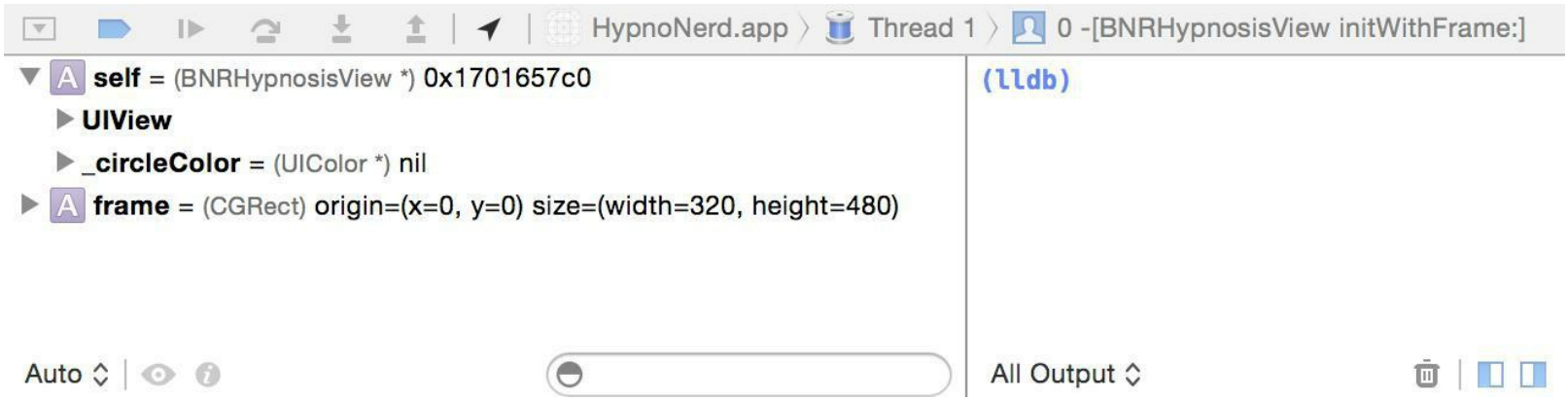


图7-7 带变量视图的调试区域

（如果Xcode没有显示变量视图，则可以先找到位于控制台右下角的控件，然后单击左边的按钮，打开变量视图。）

变量视图在显示指针变量时，会显示指针变量所指向对象的内存地址，其中，self已经有了内存地址。在断点所位于的方法中，self是BNRHypnosisView对象，有内存地址意味着在执行到断点的那行代码之前，程序已经创建并初始化了该对象。

接下来点击self旁边的三角形按钮。self下方的第一个条目是self的父类。BNRHypnosisView的父类是UIView。单击UIView边上的三角形按钮，可以显示self继承自父类的变量。

除继承自UIView的变量之外，BNRHypnosisView还有一个_circleColor变量。由于断点设置在为该变量赋值的那行代码，因此HypnoNerd还没有执行该行代码，_circleColor变量的值是nil（0x0）。

单步执行代码

调试器除了能暂停应用的执行外，还可以一行一行地单步执行代码，并显示应用执行每行代码后的状态。通过调试工具条（debugger bar）中的多个按钮，可以控制应用的运行。调试工具条位于编辑器区域和调试区域之间（见图7-8）。

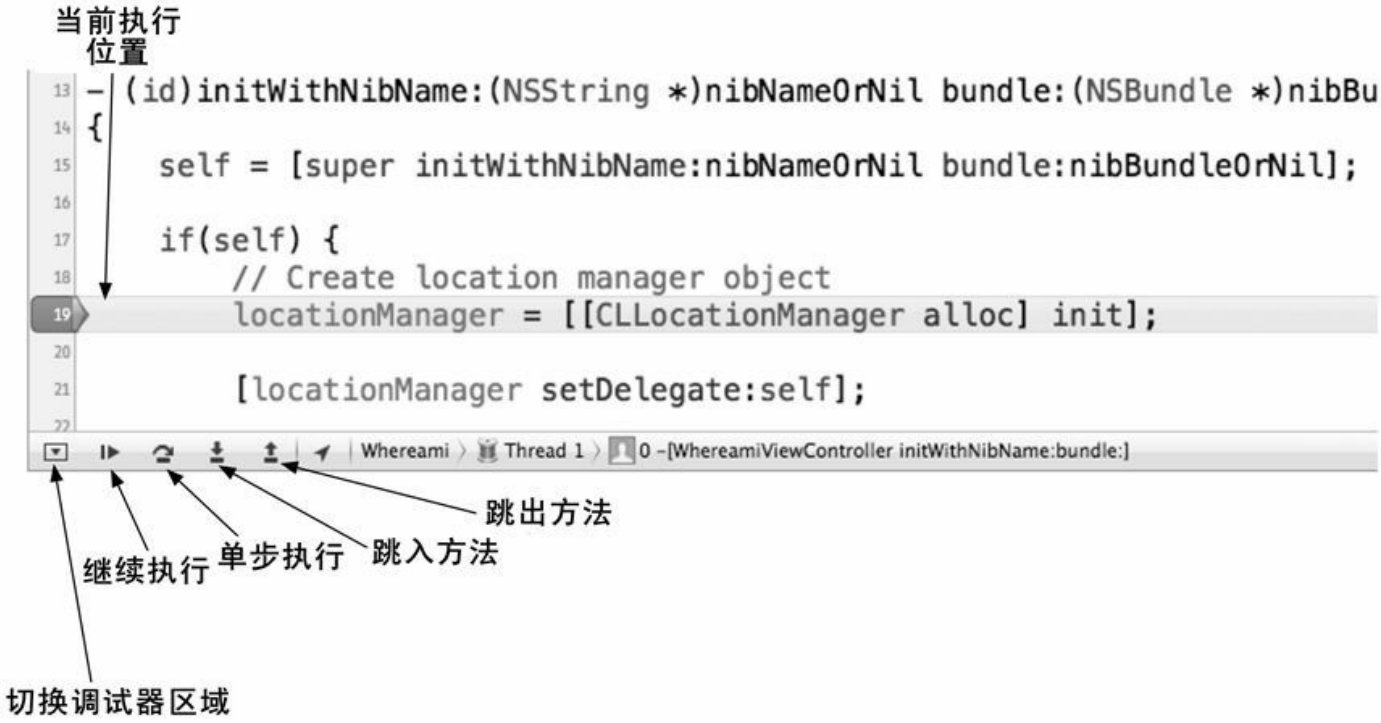


图7-8 调试工具条

单击单步执行(step over line)按钮，Xcode只会执行当前指向的那行代码，即为_circleColor变量赋值。绿色指示器会移动至下一行。变量视图所显示的_circleColor会变成内存中的一个有效地址，表示程序已经创建并初始化了_circleColor。

接下来可以继续单步执行代码并查看执行情况，也可以单击继续执行(continue executing)按钮，恢复代码的正常运行。此外，还可以单击跳入方法(step into method)按钮，跳入某个方法。跳入某个方法时，调试器会进入断点代码所调用的方法。跳入方法后，一样可以单步执行代码。

类似还有跳出方法(step out method)按钮，跳出某个方法时，调试器会进入调用断点代码的方法。请读者点击跳出方法按钮，就可以发现调试器会进入BNRHypnosisViewController.m的loadView方法。

删除断点

如果需要在调试后正常运行应用，可以通过以下几种方式删除断点：将蓝色断点符号拖曳出代码标志区；右击蓝色断点符号，选择Delete Breakpoint(删除断点)；单击导航面板选择条中的图标，显示断点导航面板并列出项目中的所有断点，然后选择相应的断点并删除。

新程序员有时会忘记曾经为项目设置过断点。当应用在断点处暂停运行时，看上去会和应用崩溃非常类似。当读者找不出应用崩溃的原因时，可以在断点导航面板中检查断点，也许“崩溃”其实是由某个被遗忘的断点造成的。

设置异常断点

以上介绍了如何为某一行代码设置断点。此外，还可以让调试器为导致应用崩溃或引发异常的那行代码自动设置断点。

为此，需要为调试器增加一个针对所有异常的断点。打开断点导航面板，单击面板底部的“+”按钮，选择Add Exception Breakpoint…（添加异常断点），如图7-9所示。

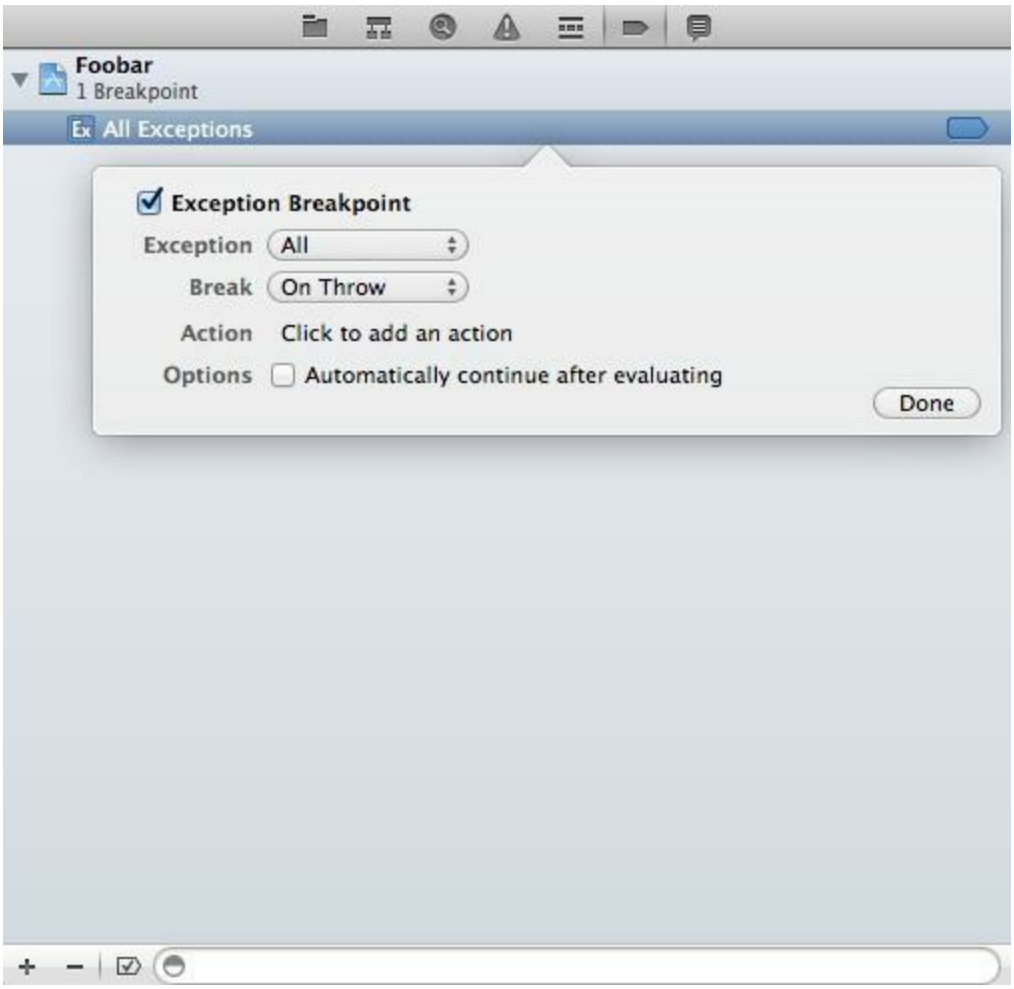


图7-9 添加异常断点

如果读者遇到了应用崩溃问题，并且无法找到错误原因时，就可以通过添加异常断点定位有问题的代码。

7.7 深入学习：main()与UIApplication

用C语言编写的程序，其执行入口都是main()。用Objective-C语言编写的程序也是这样，下面对iOS应用中的main()做一个简单的介绍。

在HypnoNerd的项目导航面板中选择main.m，应该能在编辑器区域看到以下代码：

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

这段代码中的UIApplicationMain函数会创建一个UIApplication对象。每个iOS应用都有且只有一个UIApplication对象，该对象的作用是维护运行循环。一旦程序创建了某个UIApplication对象，该对象的运行循环就会一直循环下去，main()的执行也会因此阻塞。

此外，UIApplicationMain函数还会创建某个指定类的对象，并将其设置为UIApplication对象的delegate。该对象的类是由UIApplicationMain函数的最后一个实参指定的，该实参的类型是NSString对象，代表的是某个类的类名。所以在以上这段代码中，UIApplicationMain会创建一个BNRAppDelegate对象，并将其设置为UIApplication对象的delegate。

在应用启动运行循环并开始接收事件前，UIApplication对象会向其委托发送一个特定的消息，使应用能有机会完成相应的初始化工作。这个消息的名称是application:didFinishLaunchingWithOptions:。之前已经在BNRAppDelegate.m中实现了匹配该消息的方法，并在该方法中创建了UIWindow对象和多个视图控制对象。

每个iOS应用都有一个main()，完成的都是相同的任务。如果读者仍有疑问，可以打开第1章编写的Quiz项目，然后查看其main.m文件。

7.8 中级练习：捏合-缩放

为第5章创建的Hypnosister应用添加捏合-缩放功能。

首先需要为UIScrollView对象设置委托：

•BNRAppDelegate需要遵守UIScrollViewDelegate协议。

•在application:didFinishLaunchingWithOptions:方法中，将BNRAppDelegate对象自身设置为UIScrollView对象的委托。

为了实现UIScrollView对象的委托方法，需要为BNRAppDelegate添加一个属性，指向BNRHypnosisView对象。请读者在BNRAppDelegate.m的类扩展中添加一个BNRHypnosisView属性，然后修改其余代码，将BNRHypnosisView实例变量改为属性。

接下来需要设置UIScrollView对象：将BNRHypnosisView对象作为子视图添加到该对象中；将其pagingEnabled属性设置为NO；设置contentSize属性（决定缩放区域大小）。还可以根据需要设置其他属性，请读者在UIScrollView的类参考手册中查找感兴趣的属性。

最后，实现UIScrollView对象的委托方法viewForZoomingInScrollView:，返回BNRHypnosisView对象。

如果读者遇到了问题，可以在UIScrollView的类参考手册中查找相关信息，还可以查看UIScrollViewDelegate协议中的各个委托方法说明。

提示：在模拟器中，为了模拟双指捏合手势，可以在按住Option键的同时拖曳鼠标。

第8章UITableView与UITableViewController

很多iOS应用会在界面中使用某种列表控件：用户可以选中、删除或重排列表中的条目。这些控件其实都是UITableView对象，可以用来显示一组对象，例如，用户地址簿中的一组人名，或者App Store中最畅销的一组应用。

UITableView对象虽然只能显示一列数据，但是没有行数限制。图8-1显示的是若干UITableView对象示例。

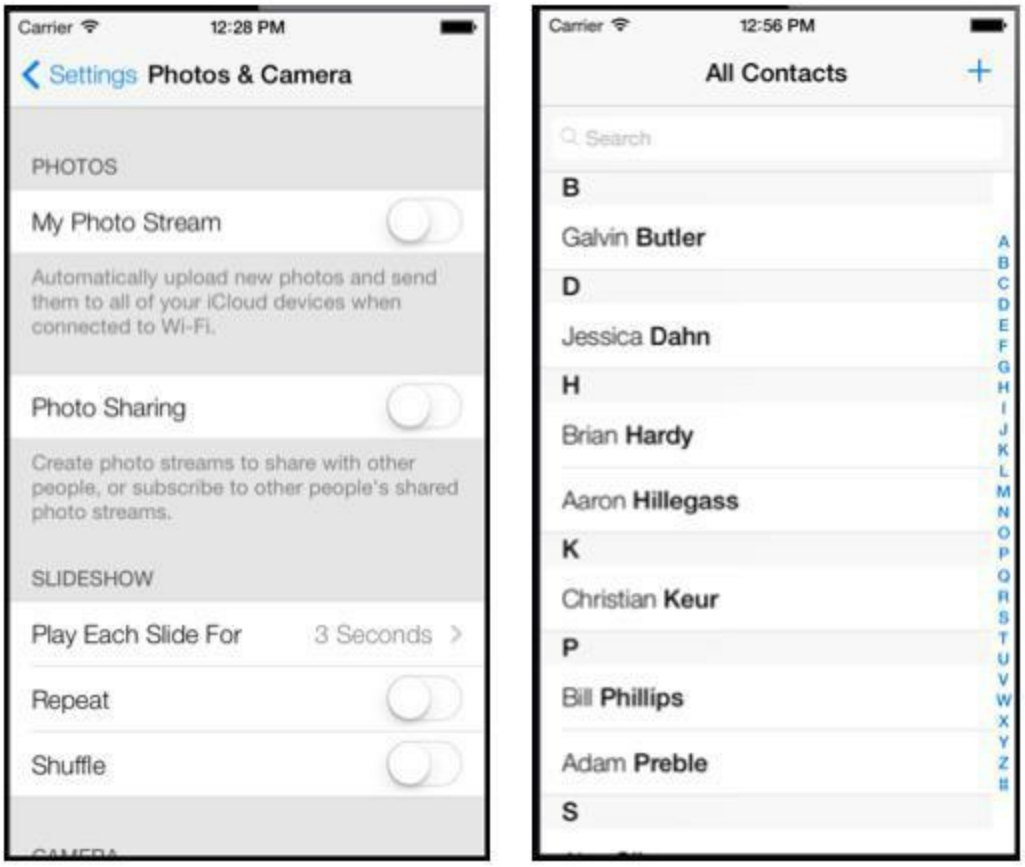


图8-1 UITableView对象示例

8.1 编写Homepwner应用

本章要开发一个名为Homepwner的应用，用来管理财产清单。如果碰上火灾（或者其他灾难），就可以向保险公司提供这份清单（顺便提一句，Homepwner一词没有拼写错误。如果读者想要了解pwn的含义，请访问http://www.urbandictionary.com）。

和本章之前介绍的iOS项目不同，Homepwner不是小项目。随着后面9章内容的不断深入，Homepwner会逐步进化为一个复杂的有实际用处的应用。到本章结束时，Homepwner将通过一个UITableView对象显示一组BNRItem对象（见图8-2）。

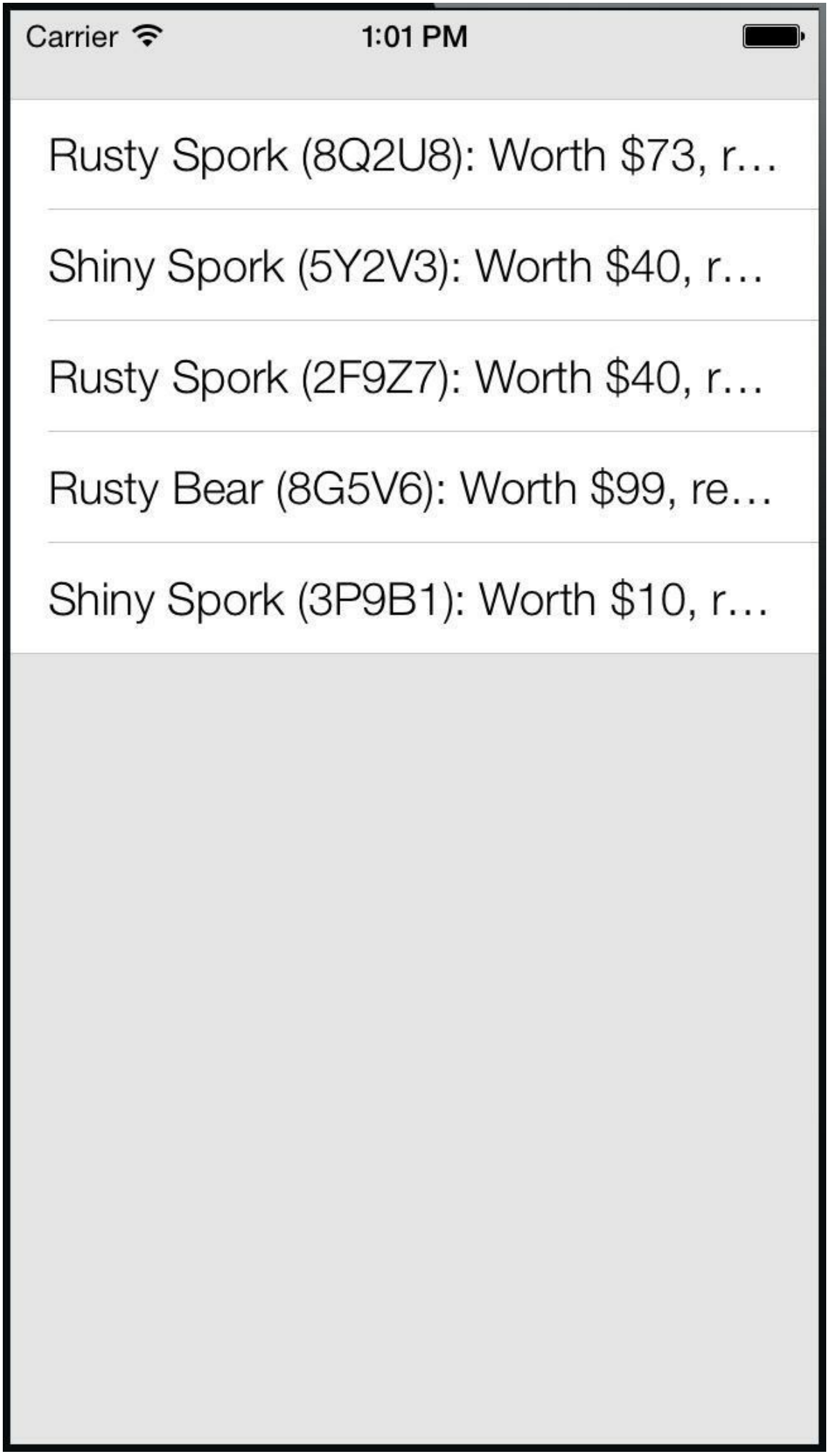

Carrier
1:01 PM
Rusty Spork (8Q2U8): Worth $73, r…
Shiny Spork (5Y2V3): Worth $40, r…
Rusty Spork (2F9Z7): Worth $40, r…
Rusty Bear (8G5V6): Worth $99, re…
Shiny Spork (3P9B1): Worth $10, r…

图8-2 Homepwner：第一阶段

创建一个新的iPhone应用并选择Empty Application模板，根据图8-3填写相应的设置信息。

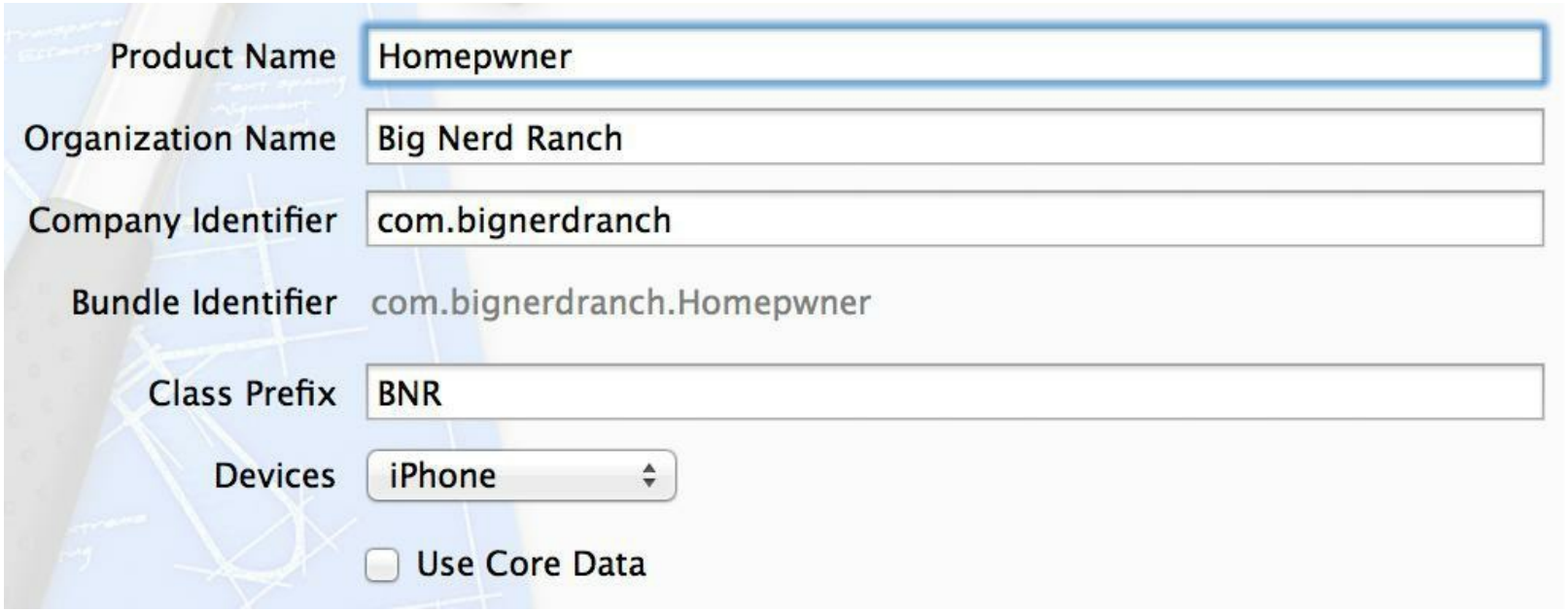


图8-3 设置Homepwner

8.2 UITableViewController

UITableView是视图。第1章中介绍过模型-视图-控制器(Model-View-Controller)，它是iOS开发者必须遵守的一种设计模式。其含义是，应用创建的任何一个对象，其类型必定是以下三种类型中的一种。

•模型：负责存储数据，与用户界面无关。

•视图：负责显示界面，与模型对象无关。

•控制器：负责确保视图对象和模型对象的数据保持一致。

一般来说，控制器用来控制应用的流程，例如，在删除数据之前必须提示用户："Really delete this item?(确定要删除该条数据吗？)"

因此，作为视图对象的UITableView不应该负责处理应用的逻辑或数据。当在应用中使用UITableView对象时，必须考虑如何搭配其他的对象，与UITableView对象一起工作：

•通常情况下，要通过某个视图控制对象来创建和释放UITableView对象，并负责显示或隐藏视图。

•UITableView对象要有数据源才能正常工作。UITableView对象会向数据源查询要显示的行数、显示表格行所需的数据和其他所需的数据。没有数据源的UITableView对象只是空壳。凡是遵守UITableViewDataSource协议的Objective-C对象，都可以成为UITableView对象的数据源(即dataSource属性所指向的对象)。

•通常情况下，要为UITableView对象设置委托对象，以便能在该对象发生特定事件时做出相应的处理。凡是遵守UITableViewDelegate协议的对象，都可以成为UITableView对象的委托对象。

UITableViewController对象可以扮演以上全部角色，包括视图控制对象、数据源和委托对象。

UITableViewController是UIViewController的子类，所以也有view属性。UITableViewController对象的view属性指向一个UITableView对象，并且这个UITableView对象由UITableViewController对象负责设置和显示。UITableViewController对象会在创建UITableView对象后，为这个UITableView对象的dataSource和delegate赋值，并指向自己(见图8-4)。

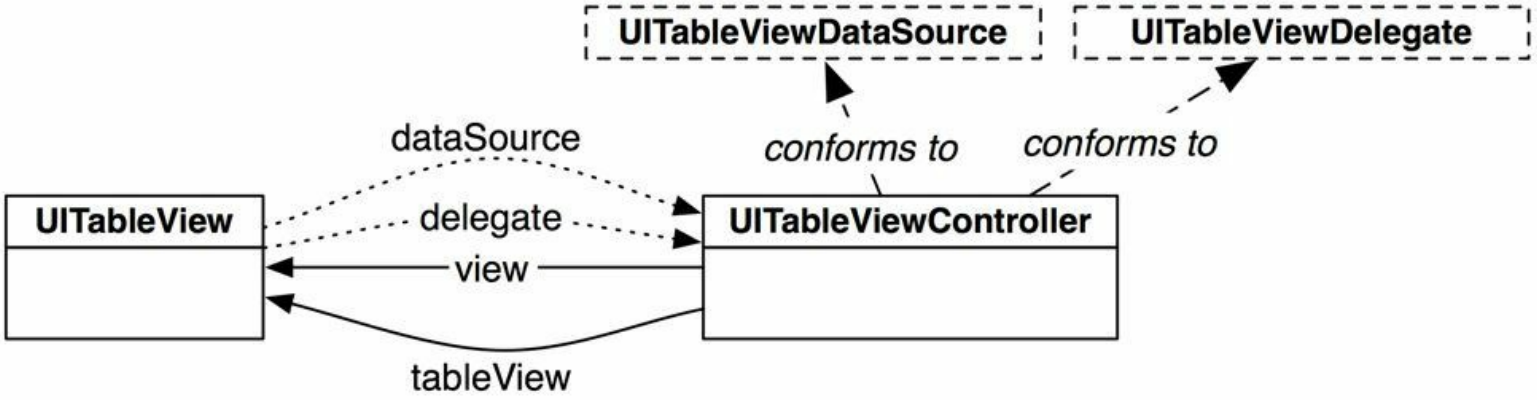


图8-4 UITableViewController和UITableView之间的关系

创建UITableViewController子类

下面要为Homepwner编写一个UITableViewController子类。这次要使用NSObject模板。选择File菜单中的New菜单项，然后选择File…选中窗口左侧iOS部分的Cocoa Touch，然后选中窗口右侧的Objective-C class，最后单击Next按钮。在新出现的面板中，在Class文本框中输入BNRItemsViewController，在Subclass of下拉菜单中选择NSObject，单击Next按钮。Xcode会提示创建文件，单击Create按钮。

打开BNRItemsViewController.h，修改BNRItemsViewController的父类，代码如下：

~~#import <Foundation/Foundation.h>~~

~~@interface BNRItemsViewController : NSObject~~

#import <UIKit/UIKit.h>

@interface BNRItemsViewController : UITableViewController

UITableViewController的指定初始化方法是initWithStyle:。调用initWithStyle:时要传入一个类型为UITableViewStyle的常数，该常数决定了UITableView对象的风格。目前可以使用的UITableViewStyle常量有两个，即UITableViewStylePlain和UITableViewStyleGrouped。两种风格的外观在iOS 6中差异很大，但是在iOS 7中非常相似。

现在将UITableViewController的指定初始化方法改为init:，为此需要遵守以下两条规则：

•在新的指定初始化方法中调用父类的指定初始化方法。

•覆盖父类的指定初始化方法，调用新的指定初始化方法。

在BNRItemsViewController.m中实现以下两个初始化方法。

```
#import "BNRItemsViewController.h"

@implementation BNRItemsViewController

- (instancetype)init

{

// 调用父类的指定初始化方法

self = [super initWithStyle:UITableViewStylePlain];

return self;

}

- (instancetype)initWithStyle:(UITableViewStyle)style

{

return [self init];

}
```

实现以上两个初始化方法后，可以确保无论向新创建的BNRItemsViewController对象发送哪一个初始化方法，初始化后的对象都会使用UITableViewStylePlain风格。

打开BNRAppDelegate.m，在application:didFinishLaunchingWithOptions:中创建一个BNRItemsViewController对象，并将其设置为UIWindow的rootViewController。此外，还要在BNRAppDelegate.m顶部导入BNRItemsViewController的头文件，代码如下：

```
#import "BNRItemsViewController.h"

@implementation BNRAppDelegate

- (BOOL)application:(UIApplication *)application

didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

{

self.window = [[UIWindow alloc] initWithFrame:

                [[UIScreen mainScreen] bounds]];

// 在这里添加应用启动后的初始代码

// 创建BNRItemsViewController对象
```

```
    BNRItemsViewController *itemsViewController =
        [[BNRItemsViewController alloc] init];
    // 将BNRItemsViewController的表视图加入窗口层次结构
    self.window.rootViewController = itemsViewController;
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
    }
```

构建并运行应用，虽然只能看到一个空白屏幕，但是屏幕上确实存在一个空的UITableView对象（见图8-5）。BNRItemsViewController作为UITableViewController的子类，继承了view方法。view方法会调用loadView方法，如果视图不存在，则loadView方法会创建并载入一个空的视图。因为UITableViewController对象的视图类型是UITableView，所以向UITableViewController对象发送view消息会得到一个空的UITableView对象。

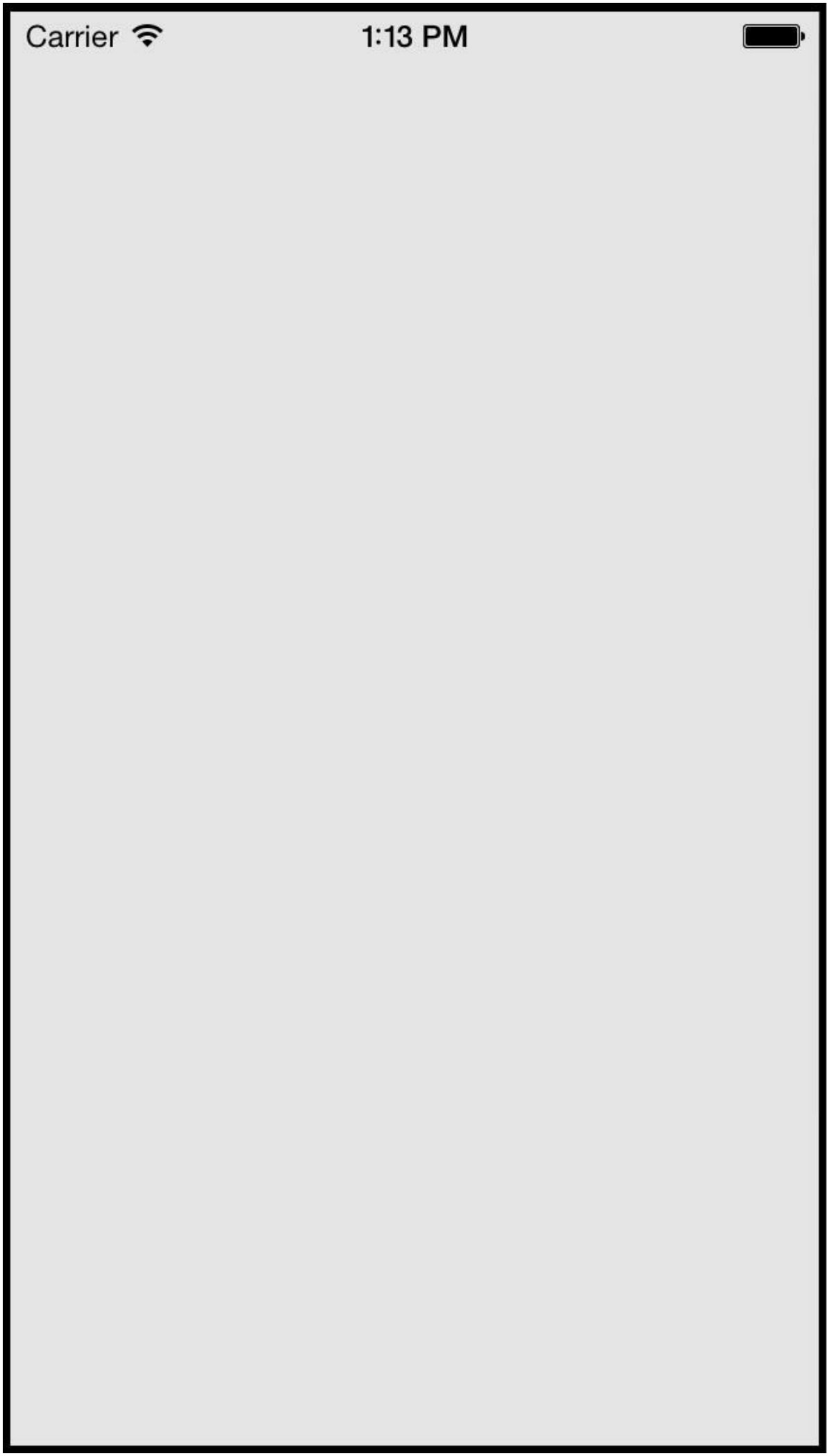

Carrier
1:13 PM

图8-5 空的UITableView对象

下面要为UITableView对象设置内容。这里借用第2章编写的BNRItem类，并将表格行和BNRItem对象对应起来，一行对应一个BNRItem对象。在Finder中找到BNRItem类的头文件和实现文件（BNRItem.h和BNRItem.m），并将这两个文件拖曳至Homepwner的项目导航面板。

在Xcode弹出的下拉窗口，选中Copy items into destination group's folder（拷贝文件或目录至目标组的目录），单击Finish按钮。Xcode会先将BNRItem.h和BNRItem.m拷贝至Homepwner的项目目录，然后将拷贝后的新文件加入Homepwner项目。

本章中不需要使用BNRItem的containedItem和container属性（它们只是用来演示强引用循环），因此，在BNRItem.h文件中删除以下代码：

```
@property (nonatomic, strong) BNRItem *containedItem;

@property (nonatomic, weak) BNRItem *container;
```

同时，在BNRItem.m中删除setContainedItem:方法：

```
- (void)setContainedItem:(BNRItem *)i

{

_containedItem = i;

// 将item加入容纳它的BNRItem对象时，

// 会将它的container实例变量指向容纳它的对象

self.containedItem.container = self;

}
```

8.3 UITableView数据源

在Cocoa Touch中，为UITableView对象设置表格行的流程与面向过程的编程模式不同。如果是面向过程的编程模式，就要“告诉”UITableView对象应该显示什么内容。在Cocoa Touch中，UITableView对象会自己查询另一个对象以获得需要显示的内容，这个对象就是UITableView对象的数据源，也就是dataSource属性所指向的对象。以BNRItemsViewController对象的UITableView对象为例，UITableView对象的数据源就是BNRItemsViewController对象自己。所以下面要为BNRItemsViewController对象添加相应的属性和方法，使其能够保存多个BNRItem对象。

第2章中的RandomItems应用使用了一个NSMutableArray对象来保存多个BNRItem对象。Homepwner也要使用相同的方式，但是要稍作修改：将用于保存BNRItem对象的NSMutableArray对象抽象为BNRItemStore对象（见图8-6）。这里为什么使用的是NSMutableArray而不是NSArray？因为BNRItemStore对象同样需要负责保存和加载BNRItem对象。

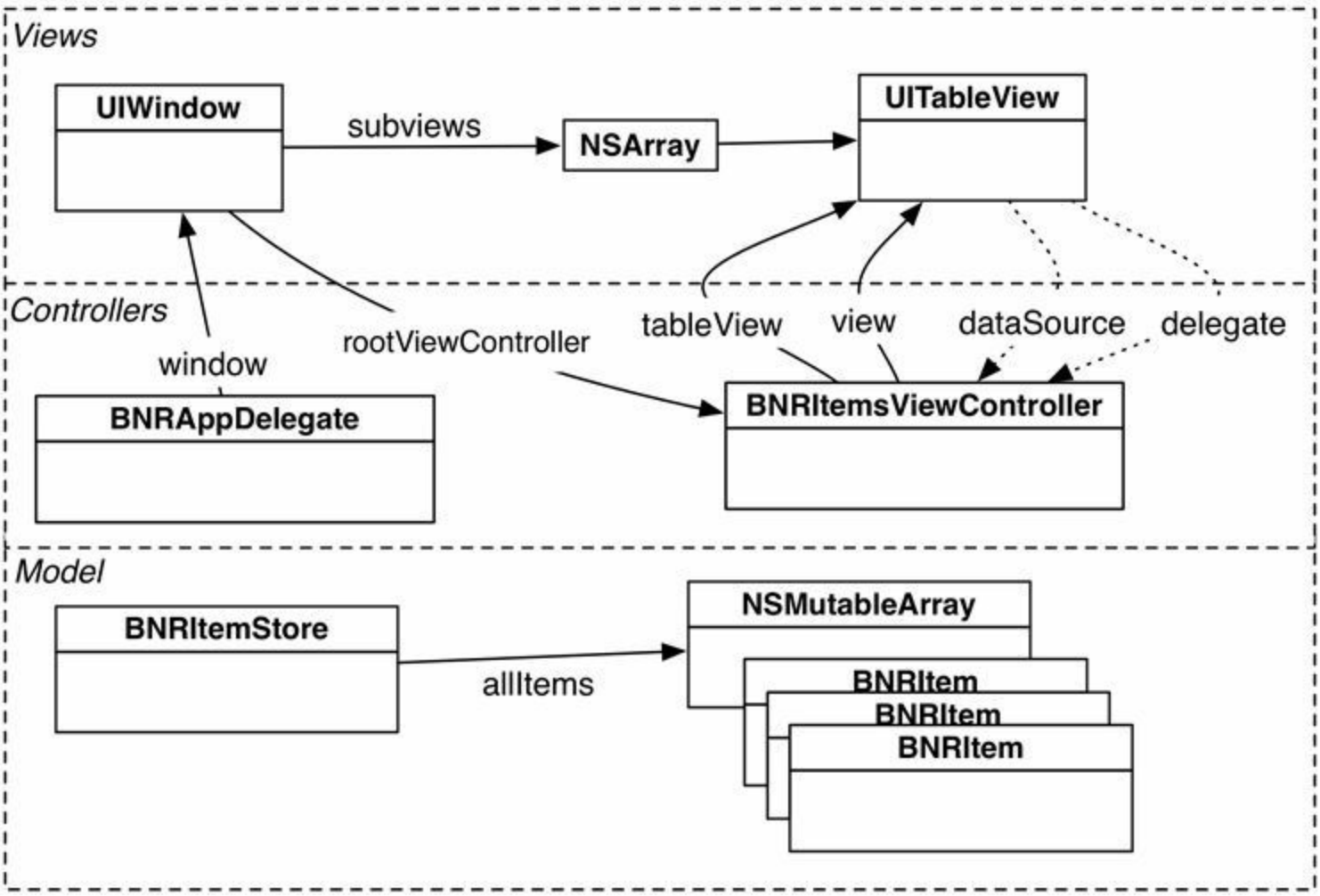


图8-6 Homepwner对象图

当某个对象需要访问所有的BNRItem时，可以通过BNRItemStore获取包含所有BNRItem的NSMutableArray。之后的章节还会为BNRItemStore添加操作数组的功能，例如添加、删除和排序。此外，BNRItemStore还会负责将BNRItem存入文件，或者从文件重新载入。

创建BNRItemStore

选择File菜单中的New菜单项，然后选择File…创建一个新的NSObject子类并将其命名为BNRItemStore。

BNRItemStore对象是一个单例。也就是说，每个应用只会有一个这种类型的对象。如果应用尝试创建另一个对象，BNRItemStore类就会返回已经存在的那个对象。当某个程序要在很多不同的代码段中使用同一个对象时，将这个对象设置为单例会很方便，只要向该对象的类发送特定的方法，就可以得到相同的对象。

在BNRItemStore.h中声明sharedStore类方法，代码如下：

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

@interface BNRItemStore : NSObject

// 注意，这是一个类方法，前缀是+，不是-

+ (instancetype)sharedStore;

@end
```

在BNRItemStore类收到sharedStore消息后，会检查是否已经创建BNRItemStore的单例对象。如果已经创建，就返回已有的对象，否则先创建再返回。

在BNRItemStore.m中实现sharedStore，同时编写一个抛出异常的init方法和私有指定初始化方法initPrivate。

```
@implementation BNRItemStore

+ (instancetype)sharedStore

{

static BNRItemStore *sharedStore = nil;

// 判断是否需要创建一个sharedStore对象

if (!sharedStore) {

    sharedStore = [[self alloc] initPrivate];

}

return sharedStore;
```

```
}
// 如果调用[[BNRItemStore alloc] init]，就提示应该使用[BNRItemStore sharedStore]
- (instancetype)init
{
@throw [NSException exceptionWithName:@"Singleton"
            reason:@"Use +[BNRItemStore sharedStore]"
              userInfo:nil];
return nil;
}
// 这是真正的(私有的)初始化方法
- (instancetype)initPrivate
{
self = [super init];
return self;
}
```

这段代码将sharedStore指针声明为了静态变量(static variable)。当某个定义了静态变量的方法返回时，程序不会释放相应的变量。静态变量和全局变量(global variable)一样，并不是保存在栈中的。

sharedStore变量的初始值是nil。当程序第一次执行sharedStore方法时，会创建一个BNRItemStore对象，并将新创建的对象的地址赋给sharedStore变量。当程序再次执行sharedStore方法时，无论是第几次，sharedStore变量仍然会指向最初的那个BNRItemStore对象。因为指向BNRItemStore对象的sharedStore变量是强引用，且程序永远不会释放该变量，所以sharedStore变量所指向的BNRItemStore对象也永远不会被释放。

BNRItemStore需要创建一个新的BNRItem对象时会向BNRItemStore对象发送消息，收到消息的BNRItemStore对象会创建BNRItem对象并将其保存到一个BNRItem数组中，之后BNRItemsViewController可以通过该数组获取所有BNRItem对象，并使用这些对象填充自己的表视图。

在BNRItemStore.h中声明一个方法和一个属性，分别用于创建和保存BNRItem对象。

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

@class BNRItem;

@interface BNRItemStore : NSObject

@property (nonatomic, readonly) NSArray *allItems;

+ (instancetype)sharedStore;

- (BNRItem *)createItem;

@end
```

这段代码使用了@class指令。该指令的作用是告诉编译器，某处代码定义了一个名为BNRItem的类。当某个文件只需要使用BNRItem类的声明，无须知道具体的实现细节时，就可以使用该指令。使用该指令后，不用在BNRItemStore.h中导入BNRItem.h，就能将createItem方法的返回类型声明为指向BNRItem对象的指针。当某个类的头文件发生变化时，对那些通过@class指令声明该类的其他文件，编译器可以不用重新编译，这样就可以大幅节省编译时间。

在另一些文件中，程序会向BNRItem类或BNRItem对象发送消息。对这些文件，就必须导入BNRItem的头文件，使编译器知道所有的实现细节。在BNRItemStore.m顶部导入BNRItem.h，以便之后向BNRItem对象发送消息，代码如下：

```
#import "BNRItemStore.h"

#import "BNRItem.h"
```

请注意，Homepwner将使用BNRItemStore管理BNRItem数组——包括添加、删除和排序。因此，除BNRItemStore之外的类不应该对BNRItem数组做这些操作。在BNRItemStore内部，需要将BNRItem数组定义为可变数组。而对于其他类来说，BNRItem数组则是不可变数组。这是一种常见的设计模式，用于设置内部数据的访问权限：某个对象中有一种可修改的数据，但是除该对象之外，其他对象只能访问该数据而不能修改它。例如，在之前的代码中，allItems属性被声明为NSArray类型（不可变数组），并将其设置为readonly。这样，其他类既无法将一个新数组赋给allItems，也无法修改allItems。

接下来在BNRItemStore.m的类扩展中声明一个可变数组。

```
#import "BNRItem.h"

@interface BNRItemStore ()

@property (nonatomic) NSMutableArray *privateItems;

@end

@implementation BNRItemStore
```

然后实现initPrivate方法，初始化privateItems属性。同时还需要覆盖allItems的取方法，返回privateItems。

```
- (instancetype)initPrivate

{

self = [super init];

if (self) {

    _privateItems = [[NSMutableArray alloc] init];

}

return self;

}

- (NSArray *)allItems

{

return self.privateItems;

}
```

allItems方法的返回值是NSArray类型，但是方法体中返回的是NSMutableArray类型的对象，这种写法是正确的，因为NSMutableArray是NSArray的子类。读者可以将NSMutableArray看成是一种特殊的NSArray，它具有NSArray的所有功能。（请注意，如果allItems的类型是NSMutableArray，而privateItems的类型是NSArray，那么这种写法就是错误的，因为NSArray没有NSMutableArray中关于修改数组的功能。）

这种写法可能会引起一个问题：虽然头文件中将allItems的类型声明为NSArray，但是其他对象调用BNRItemStore的allItems方法时，得到的一定是一个NSMutableArray对象——Objective-C对象知道自己的类型，无论是属性声明还是返回值类型声明都不会修改对象类型。

使用像BNRItemStore这样的类时，应该遵循其头文件中的声明使用类的属性和方法，例如，在BNRItemStore头文件中，因为allItems属性的类型是NSArray，所以应该将其作为NSArray类型的对象使用。如果将allItems转换为NSMutableArray类型并修改其内容，就违反了BNRItemStore头文件中的声明。可以通过覆盖allItems方法避免其他类修改allItems：在allItems方法中使用copy方法返回privateItems属性的不可变副本（对应于copy方法，还有一个mutableCopy方法可以返回相应的可变副本），类似于以下代码：

```
- (NSArray *)allItems

{
```

```
return [self.privateItems copy];

}
```

以上代码没有使用黑体标注的原因是，建议读者不要编写这类代码。其实只要遵循编程约定（这里的约定是：遵循头文件中的声明），就不会出现这类问题。

在BNRItemStore.m中，按照之前介绍的方式实现createItem方法：

```
- (BNRItem *)createItem

{

BNRItem *item = [BNRItem randomItem];

[self.privateItems addObject:item];

return item;

}
```

现在请读者回顾第3章中有关属性合成的知识。BNRItemStore.h将allItems声明为只读属性，而BNRItemStore.m又覆盖了allItems的取方法，因此编译器不会为allItems生成取方法和实例变量_allItems。

实现数据源方法

在BNRItemsViewController.m顶部导入BNRItemStore.h和BNRItem.h。然后更新指定初始化方法，创建5个随机的BNRItem对象并加入BNRItemStore对象，代码如下：

```
#import "BNRItemsViewController.h"

#import "BNRItemStore.h"

#import "BNRItem.h"

@implementation BNRItemsViewController

- (instancetype)init

{

// 调用父类的指定初始化方法

self = [super initWithStyle:UITableViewStylePlain];
```

```
    if (self) {
        for (int i = 0; i < 5; i++) {
            [[BNRItemStore sharedStore] createItem];
        }
    }
    return self;
}
```

将若干BNRItem对象加入BNRItemStore对象后，下面要让BNRItemsViewController对象将这些BNRItemStore对象转变成UITableView对象可以显示的表格行。当某个UITableView对象要获取显示的数据时，会向其数据源发送一组特定的消息。这些消息都是在UITableViewDataSource协议中声明的。

选择Help菜单中的Documentation and API Reference，搜索UITableViewDataSource协议的参考文档，然后选中左侧面板中的Tasks（见图8-7）。

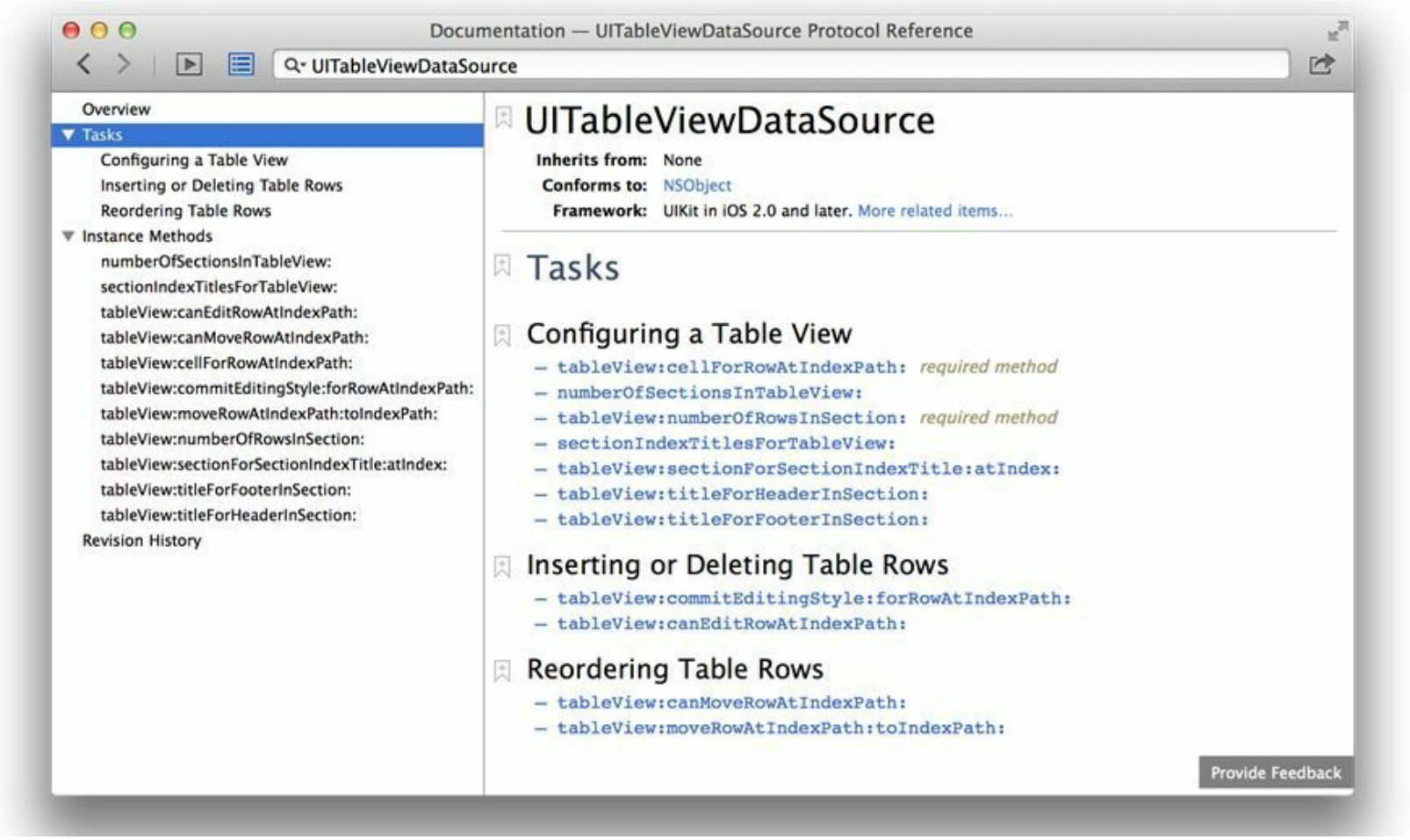


图8-7 UITableViewDataSource协议的参考文档

第一个task是Configuring a Table View（配置表视图），其中包含了一系列相关方法，请注意这些方法中有两个被标记为了required method（必需方法）。要让BNRItemsViewController遵守UITableViewDataSource协议，就必须为BNRItemsViewController实现tableView:numberOfRowsInSection:和tableView:cellForRowAtIndexPath:这两个必需方法。UITableView对象可以通过数据源对象的这两个方法获得应该显示的行数及显示各行所需的视图。

当某个UITableView对象要显示表格内容时，会向自己的数据源（dataSource属性所指向的对象）发送一系列消息，其中包括必需方法和可选方法。tableView: numberOfRowsInSection:（必需方法）会返回一个整型值，代表UITableView对象显示的行数。对于Homepwner中的UITableView对象来说，BNRItemStore中的每个BNRItem对象都应该对应一个表格行。

在BNRItemsViewController.m中实现tableView:numberOfRowsInSection:方法：

```
- (NSInteger)tableView:(UITableView *)tableView

numberOfRowsInSection:(NSInteger)section

{

return [[[BNRItemStore sharedStore] allItems] count];

}
```

请注意该方法的返回值是一个NSInteger类型的整数。从Apple开始支持64位应用之后，整型值在32位应用中应该是一个32位的整数，而在64位应用中应该是一个64位的整数。因此Apple使NSInteger（有符号整型）和NSUInteger（无符号整型）在32位和64位应用中表示不同位数的整数。为了使应用适配32位和64位设备，请读者使用以上类型代替int。

传入tableView:numberOfRowsInSection:方法的section参数起什么作用？UITableView对象可以分段显示数据，每个表格段（section）包含一组独立的行。以通讯录（Contacts）应用为例，所有以字母“D”开头的名字都会被归在一个表格段中。UITableView对象默认只有一个表格段。本章中的Homepwner应用也只会使用一个表格段。一旦读者理解了UITableView对象的工作原理，就能很容易地实现多个表格段。这也是本章结尾处的第一个练习。

UITableViewDataSource协议中的另外一个必须实现的方法是tableView:cellForRowAtIndexPath:。在实现该方法前，需要先介绍另一个类：UITableViewCell。

8.4 UITableViewCell对象

表视图所显示的每一行都是一个独立的视图，这些视图是UITableViewCell对象。本节将学习创建UITableViewCell对象并使用UITableViewCell对象填充表视图。在第19章中还会学习创建自定义的UITableViewCell子类。

UITableViewCell对象有一个子视图：contentView（见图8-8）。contentView也包含很多子视图，它的子视图构成UITableViewCell对象的主要外观。此外，UITableViewCell对象还可以显示一个辅助指示视图（accessory indicator）。辅助指示视图的作用是显示一个指定的图标，用于向用户提示UITableViewCell对象可以执行的动作。这些图标包括勾选标记、展开图标或中间有v形图案的蓝色圆点。为UITableViewCell对象设置不同的辅助指示视图类型，可以改变图标的外观。辅助指示视图的默认类型是UITableViewCellAccessoryNone，它也是本章将使用的类型。第19章会再次介绍辅助指示视图（读者也可自行参阅UITableViewCell的参考文档，以了解更多详细信息）。

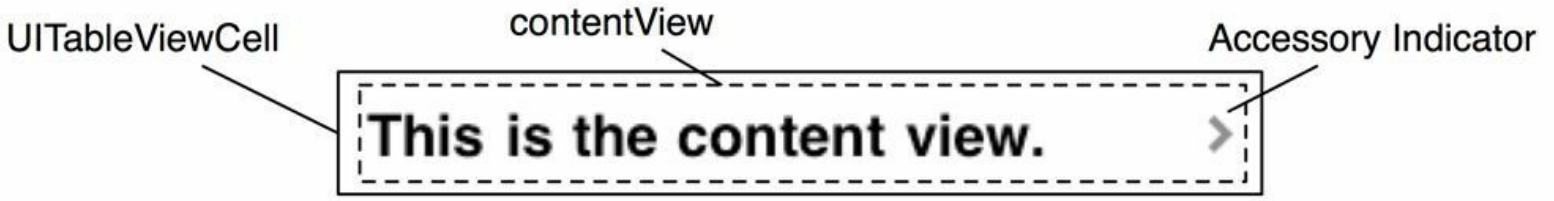


图8-8 UITableViewCell布局

负责显示UITableViewCell对象所代表的数据，是contentView所包含的三个子视图。其中的两个视图是UILabel对象，分别为textLabel属性和detailTextLabel属性所指向的对象。第三个是UIImageView对象，即imageView属性所指向的对象（见图8-9）。本章只用到了textLabel属性。

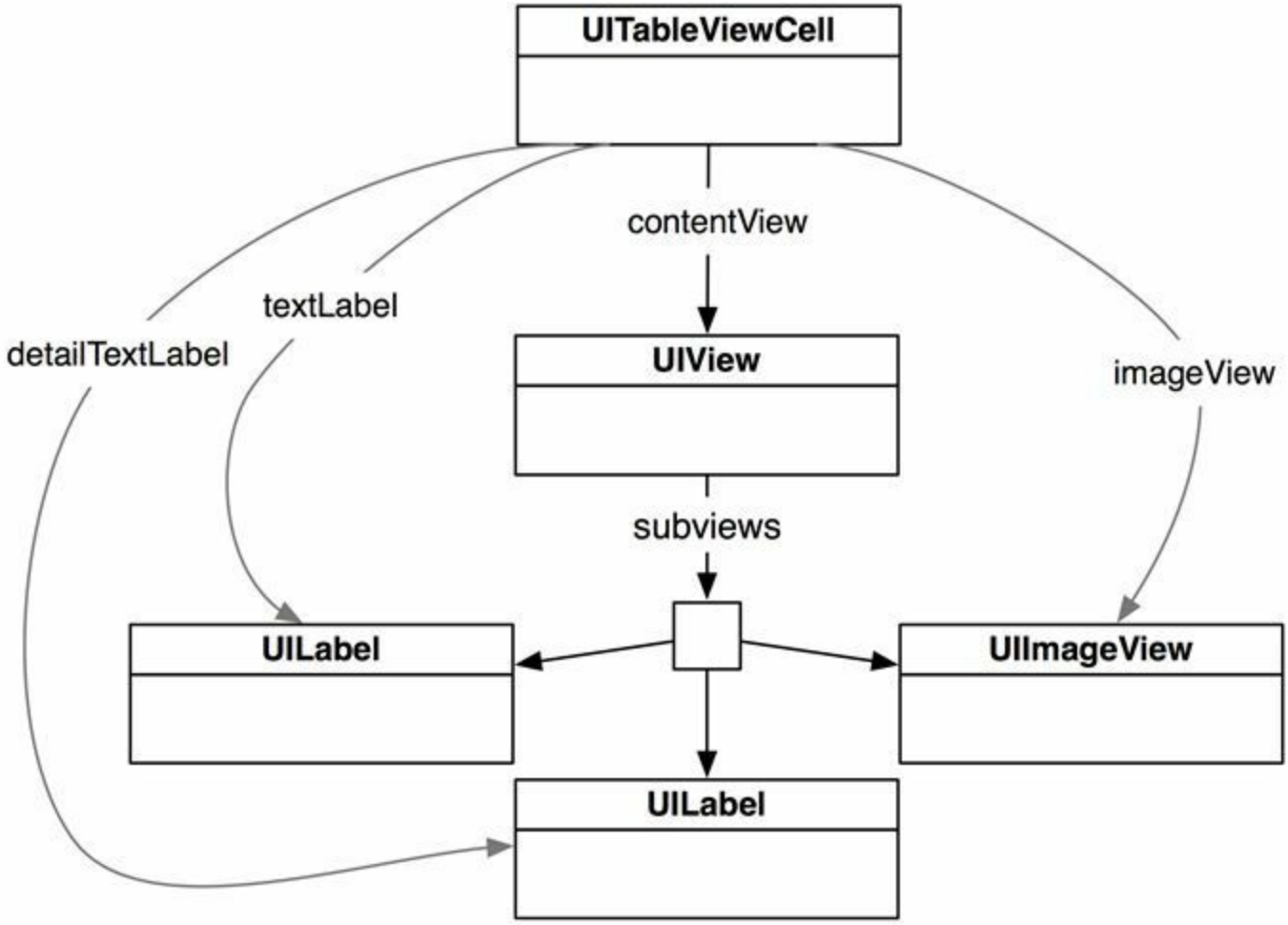


图8-9 UITableViewCell的视图层次结构

此外，在创建UITableViewCell对象时，可以选择一种风格（UITableViewCellStyle）。这种风格决定UITableViewCell对象会显示上述三个子视图中的哪几个，以及这些视图在contentView中的位置。图8-10列出了所有的UITableViewCellStyle常量和相应的外观示例。

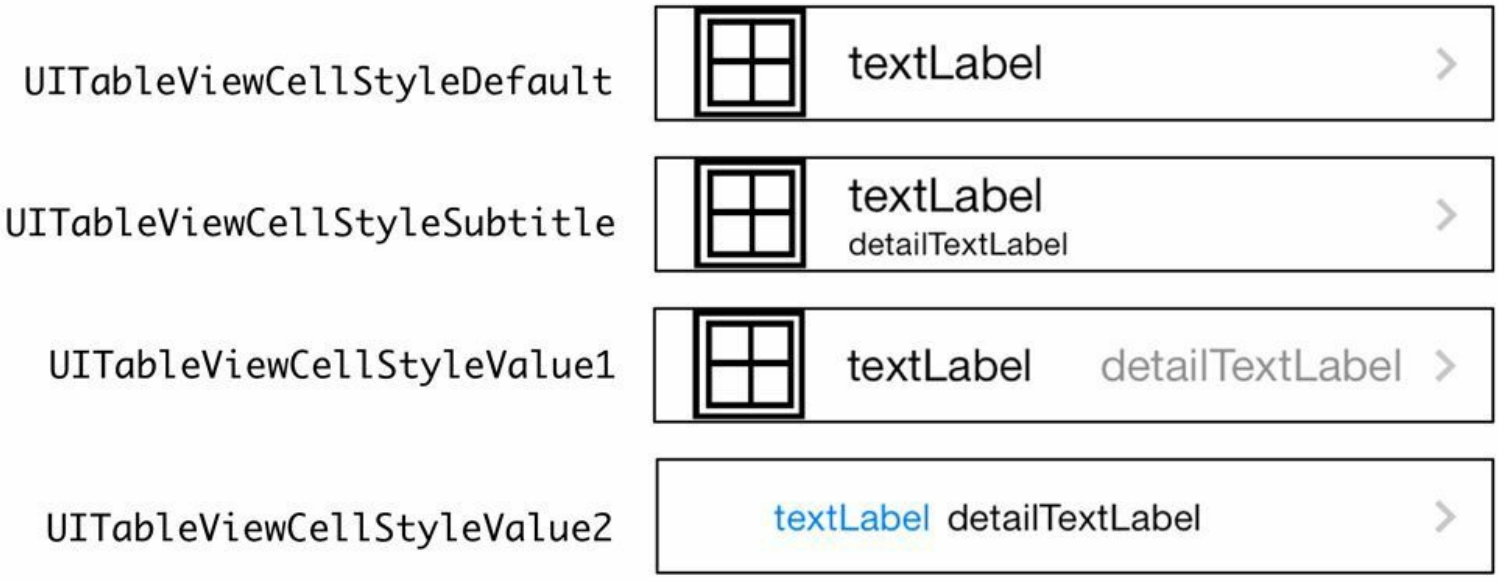


图8-10 UITableViewCellStyle常量

创建并获取UITableViewCell对象

下面介绍如何通过UITableViewCell对象的textLabel属性显示BNRItem对象的描述信息。要完成这项任务，需要实现UITableViewDataSource协议的第二个必需方法——tableView:cellForRowAtIndexPath:。对于BNRItemsViewController类，tableView:cellForRowAtIndexPath:需要完成这些任务：创建一个UITableViewCell对象，获取UITableViewCell对象所代表的BNRItem对象，向BNRItem对象发送description消息并得到描述信息，将得到的描述信息赋给UITableViewCell对象的textLabel属性，最后返回UITableViewCell对象（见图8-11）。

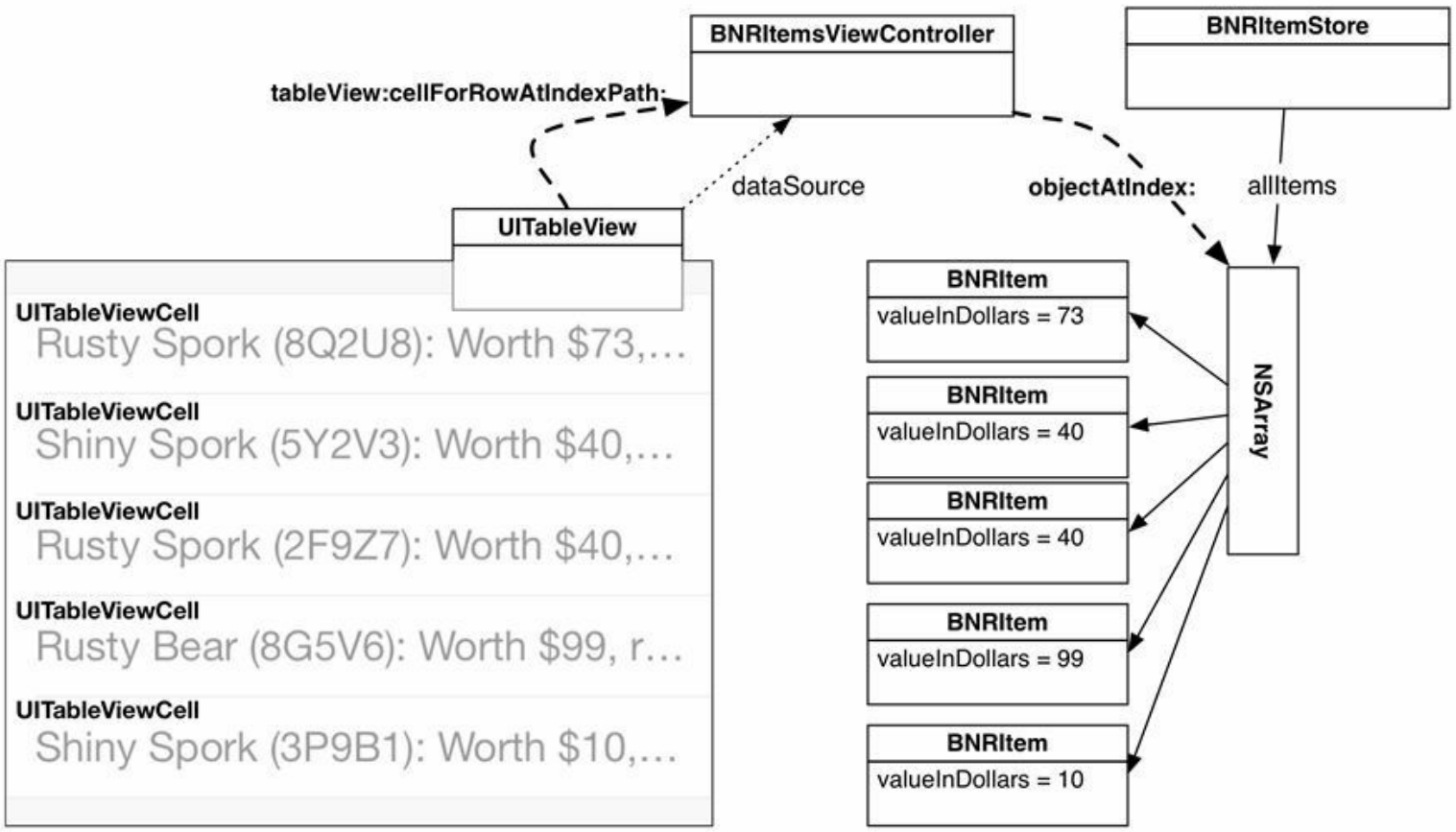


图8-11 获取UITableViewCell对象的流程

如何确定BNRItem对象和UITableViewCell对象的对应关系？传入tableView:cellForRowAtIndexPath:的第二个实参是一个NSIndexPath对象。该对象包含两个属性：section（段）和row（行）。当UITableView对象向其数据源发送tableView: cellForRowAtIndexPath:消息时，其目的是获取用于显示第section个表格段、第row行数据的UITableViewCell对象。Homepwner只显示一个表格段，所以其UITableView对象只要用到row。

在BNRItemsViewController.m中实现tableView:cellForRowAtIndexPath:，让第n行显示allItems数组中的第n个BNRItem对象，代码如下：

```
- (UITableViewCell *)tableView:(UITableView *)tableView
```

```
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
// 创建UITableViewCell对象，风格使用默认的UITableViewCellStyleDefault
UITableViewCell *cell =
    [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
                                reuseIdentifier:@"UITableViewCell"];
// 获取allItems的第n个BNRItem对象，
// 然后将该BNRItem对象的描述信息赋给UITableViewCell对象的textLabel
// 这里的n是该UITableViewCell对象所对应的表格行索引
NSArray *items = [[BNRItemStore sharedStore] allItems];
BNRItem *item = items[indexPath.row];
cell.textLabel.text = [item description];
return cell;
}
```

构建并运行应用，Homepwner应该会显示一个UITableView对象，其中包含一组随机的BNRItem对象。

现在请读者回顾第3章的RandomItems。之前在RandomItems中创建了BNRItem类及其对象（属于MVC中的模型对象），并在控制台中打印了BNRItem对象的数据。

Homepwner则重用了BNRItem类，在不改动BNRItem的情况下，只需使用不同的视图对象和控制器对象就可以通过另一种完全不同的方式展示BNRItem的数据，这就是MVC设计模式的典型示例。如果按照MVC设计模式设计类的功能，以后就很容易将类用于其他应用中。

重用UITableViewCell对象

ios设备内存有限。如果某个UITableView对象要显示大量的记录，并且要针对每条记录创建相应的UITableViewCell对象，就会很快耗尽iOS设备的内存资源。

为了解决该问题，需要重用UITableViewCell对象（见图8-12）。当用户滚动UITableView对象时，部分UITableViewCell对象会移出窗口。UITableView对象会将移出窗口的UITableViewCell

对象放入UITableViewCell对象池，等待重用。当UITableView对象要求数据源返回某个UITableViewCell对象时，数据源可以先查看对象池。如果有未使用的UITableViewCell对象，就可以用新的数据配置这个UITableViewCell对象，然后将其返回给UITableView对象，从而避免创建新对象。

这里还有一个问题：因为有时需要创建UITableViewCell的子类，用于实现特定的外观或特性，所以UITableView对象可能会拥有不同类型的UITableViewCell对象。如果UITableViewCell对象池中的对象创建自不同的子类，那么UITableView对象就有可能得到错误类型的UITableViewCell对象。鉴于上述原因，必须确保UITableView对象能够得到指定类型的UITableViewCell对象，这样才能确定返回的对象会拥有哪些属性和方法。

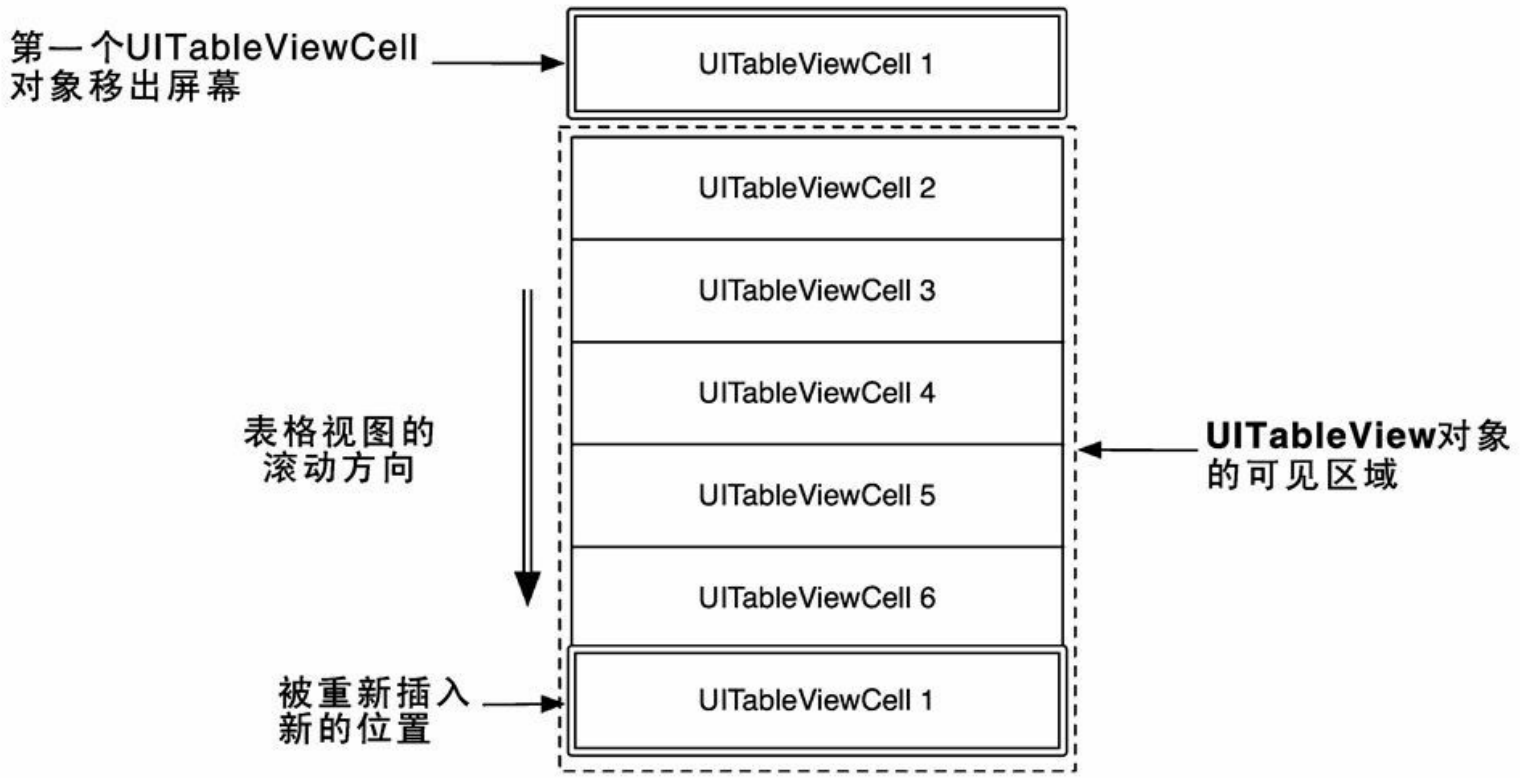


图8-12 重用UITableViewCell对象

从UITableViewCell对象池获取对象时，无须关心取回的是否是某个特定的对象。因为无论取回的是哪个对象，都要重新设置数据。真正要关心的是取回的对象是否是某个特定的类型。每个UITableViewCell对象都有一个类型为NSString的reuseIdentifier属性。当数据源向UITableView对象获取可重用的UITableViewCell对象时，可传入一个字符串并要求UITableView对象返回相应的UITableViewCell对象，这些UITableViewCell对象的reuseIdentifier属性必须和传入的字符串相同。按照约定，应该将UITableViewCell或者UITableViewCell子类的类名用作reuseIdentifier。

在BNRItemsViewController.m中更新tableView:cellForRowAtIndexPath:，以便能够重用UITableViewCell对象，代码如下：

```
- (UITableViewCell *)tableView:(UITableView *)tableView

cellForRowAtIndexPath:(NSIndexPath *)indexPath
```

```
{
~~UITableViewCell *cell =~~
  ~~[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault~~
~~reuseIdentifier:@"UITableViewCell"];~~
// 创建或重用UITableViewCell对象
UITableViewCell *cell =
    [tableView dequeueReusableCellWithIdentifier:@"UITableViewCell"
            forIndexPath:indexPath];
NSArray *items = [[BNRItemStore sharedStore] allItems];
BNRItem *item = items[indexPath.row];
cell.textLabel.text = [item description];
return cell;
}
```

之前的代码都是手动创建UITableViewCell对象的。为了重用UITableViewCell对象，必须将创建的过程交由系统管理——需要告诉表视图，如果对象池中没有UITableViewCell对象，应该初始化哪种类型UITableViewCell对象。

在BNRItemsViewController.m中覆盖viewDidLoad方法，向表视图注册应该使用的UITableViewCell。

```
- (void)viewDidLoad
{
[super viewDidLoad];
[self.tableView registerClass:[UITableViewCell class]
forCellReuseIdentifier:@"UITableViewCell"];
}
```

重用UITableViewCell对象，意味着UITableView对象只需要创建少量的UITableViewCell对象，从而减少内存的占用量，提升用户界面的流畅性。构建并运行应用，Homepwner的运行结果应该与之前相同。

8.5 代码片段库

读者可能已经注意到，如果在某个Objective-C的实现文件中输入init，Xcode就会自动列出一组备选项，其中包括init方法。选择后，Xcode就会在当前位置加入一段init方法的默认实现代码。

Xcode自动加入的这段代码源自代码片段库（code snippet library）。打开工具区域，点击库面板选择条（library selector）中的图标就可以打开代码片段库面板（见图8-13）。此外，也可以使用快捷键Command-Control-Option-2，直接打开工具区域和代码片段库面板（替换快捷键中的数字，可以打开其他的库面板）。

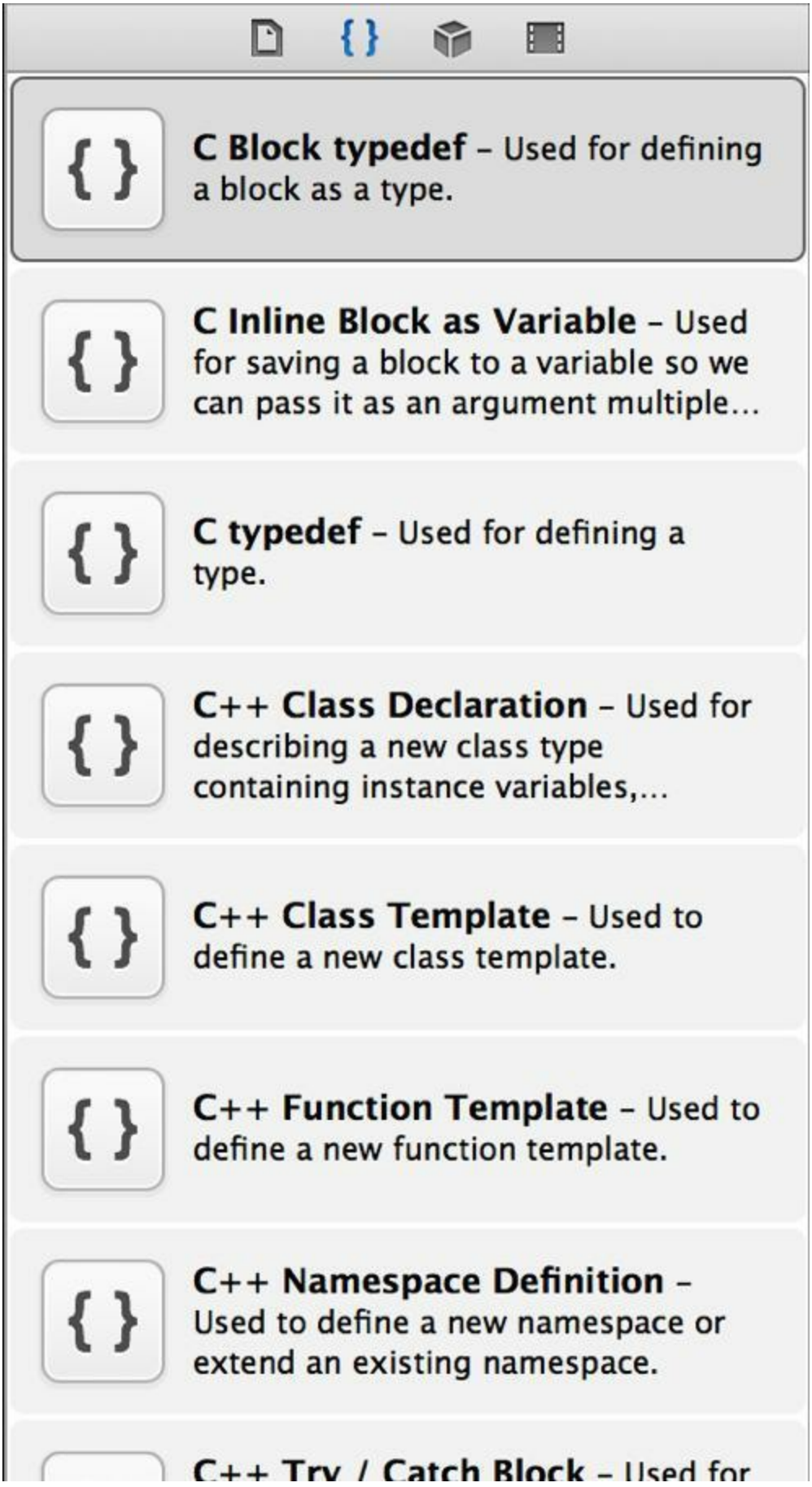


图8-13 代码片段库面板

库面板会列出很多代码片段（见图8-13）。单击其中的一个，然后鼠标稍作悬停，Xcode就会弹出设置窗口并显示相应代码片段的详细信息。单击窗口中的Edit按钮（见图8-14）。

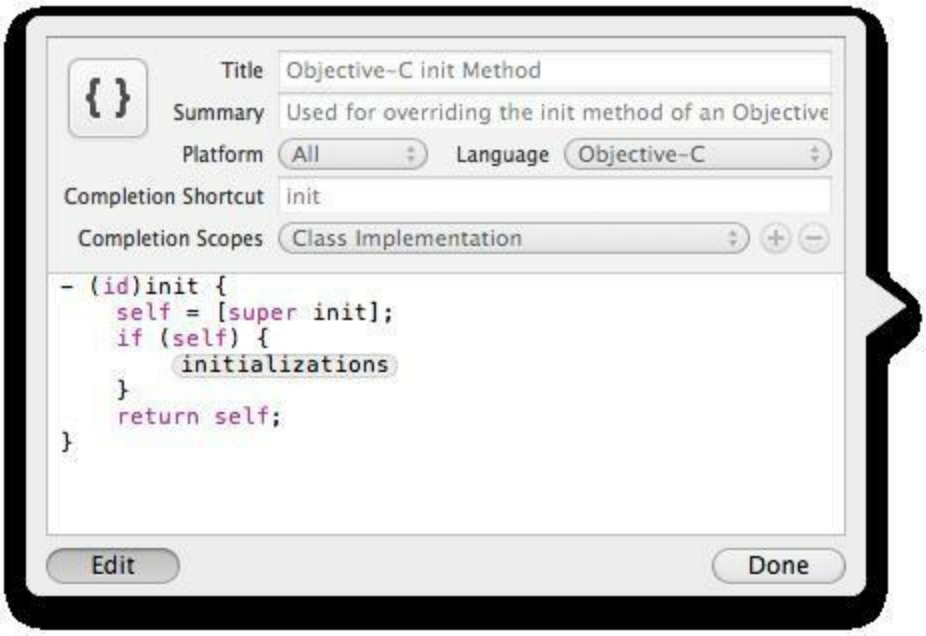


图8-14 代码片段编辑窗口

通过窗口中的Completion Shortcut输入框，可以查看或修改代码片段的触发字符串（在源代码文件中输入该字符串后，会触发Xcode自动加入相应的代码片段）。由图8-14所显示的编辑窗口可知：只有在Objective-C代码中（Language是Objective-C）并且是在类的实现部分（Completion Scopes是Class Implementation）输入init，Xcode才会自动输入init方法的代码片段。

在代码片段库中，虽然Xcode自带的代码片段是只读的，不能编辑，但是读者可创建自定义的代码片段。找到BNRItemsViewController.m中的tableView:numberOfRowsInSection:，选中整个方法，代码如下：

```
- (NSInteger)tableView:(UITableView *)tableView

numberOfRowsInSection:(NSInteger)section

{

return [[[BNRItemStore sharedStore] allItems] count];

}
```

将选中的代码段拖曳至代码片段库，Xcode会将这段代码加入代码片段库并弹出编辑窗口，以便读者能够做进一步的设置。

新加入的代码片段有一个缺陷：返回语句是针对Homepwner应用编写的，两者耦合在了一起。如果返回值是某种代码补全占位符（code completion placeholder），就可以很容易地在该位置输入代码。这样，当开发其他应用时，也能很方便地使用这段代码片断。在编辑窗口中，将之前的代码片段改为以下内容：

```
- (NSInteger)tableView:(UITableView *)tableView

numberOfRowsInSection:(NSInteger)section
```

{

return <#number of rows#>;

}

然后根据图8-15填写编辑窗口中的其他输入框，最后单击Done按钮。

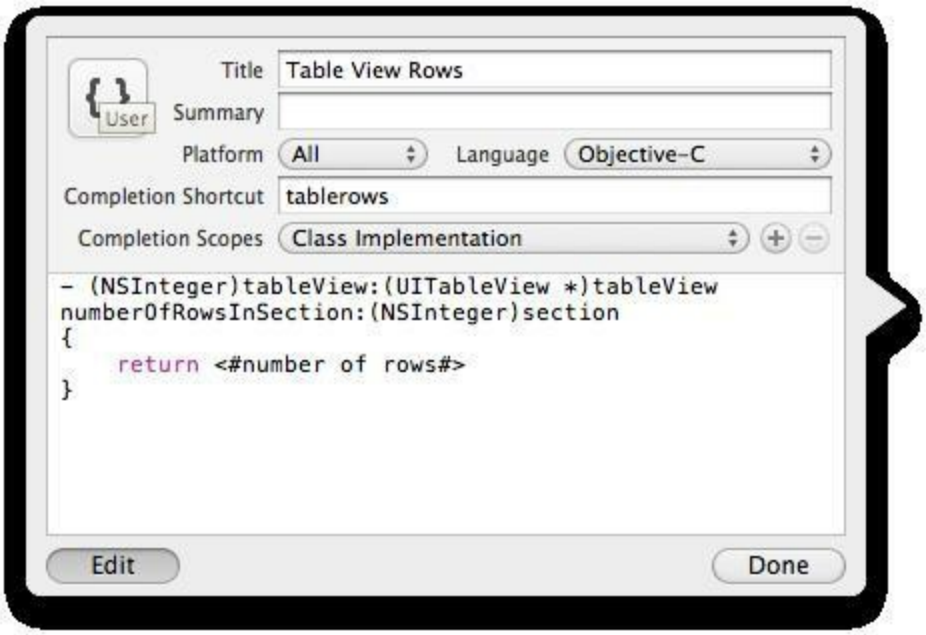


图8-15 创建新的代码片段

打开BNRItemsViewController.m，输入tablerows，Xcode会提示可以自动加入之前新增加的那个代码片段。回车后，Xcode会自动加入代码并选中number of rows占位符。（如果有多个占位符，可以使用快捷键Control-/跳到下一个占位符。）

因为之前已经定义了tableView:numberOfRowsInSection:，所以请读者删除这段Xcode自动加入的代码。

8.6 初级练习：表格段

编写代码，使Homepwner中的UITableView对象能够显示两个表格段：一段显示价值大于50美元的BNRItem对象，另一段显示余下的BNRItem对象。因为后续章节还会使用Homepwner项目，所以请读者在练习前先拷贝项目，然后在拷贝后的项目中修改代码。

8.7 中级练习:固定行

编写代码,使Homepwner中的UITableView对象能够在表格末端显示一个标题为No more items!的UITableViewCell对象。无论BNRItemStore包含0个还是多个BNRItem对象,UITableView对象也要显示该UITableViewCell对象。

8.8 高级练习：修改UITableView对象的外观

完成中级练习后，将UITableView对象中的表格行高度改为60点。但是有一个例外，对在中级练习中加入的那个始终出现在表格末端的表格行，要保留其高度不变，仍旧是44点。最后，更新UITableView对象的背景，显示一幅图像（为了使背景图像可以正确显示，请读者参考第1章中的表1-2，根据自己的设备类型准备一个相应大小的图像文件）。

第9章 编辑UITableView

第8章完成的Homepwner应用可以通过UITableView对象显示一组BNRItem对象。下面要为Homepwner添加新的功能，使UITableView可以响应用户操作，包括添加、删除和移动表格行。图9-1显示的是本章最终完成的Homepwner。

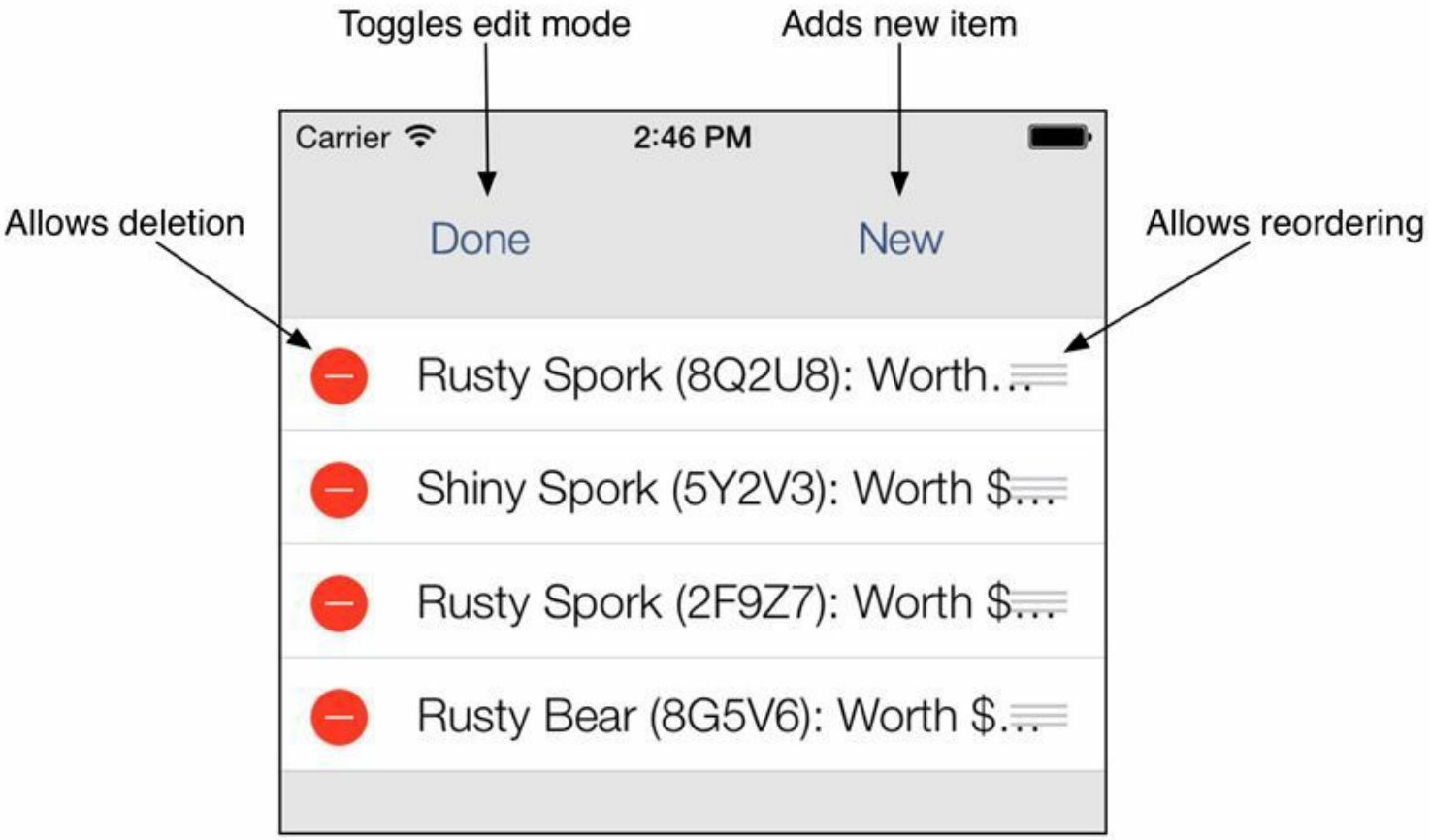


图9-1 编辑模式下的Homepwner

9.1 编辑模式

UITableView有一个名为editing的属性，如果将editing属性设置为YES，UITableView就会进入编辑模式。在编辑模式下，用户可以管理UITableView中的表格行，例如之前提到的添加、删除和移动等操作。但是编辑模式没有提供修改行的内容的功能。

首先需要更新界面，使用户可以将UITableView对象设置为编辑模式。本章是为UITableView对象的表头视图（header view）增加一个按钮，然后通过点击按钮使UITableView对象进入或退出编辑模式。表头视图是指UITableView对象可以在其表格上方显示的特定视图，适合放置针对某个表格段或整张表格的标题和控件。表头视图可以是任意的UIView对象。

表头视图有两种，分别针对表格段和表格。类似地，还有表尾视图（footer view），也具有表格段和表格两种（见图9-2）。

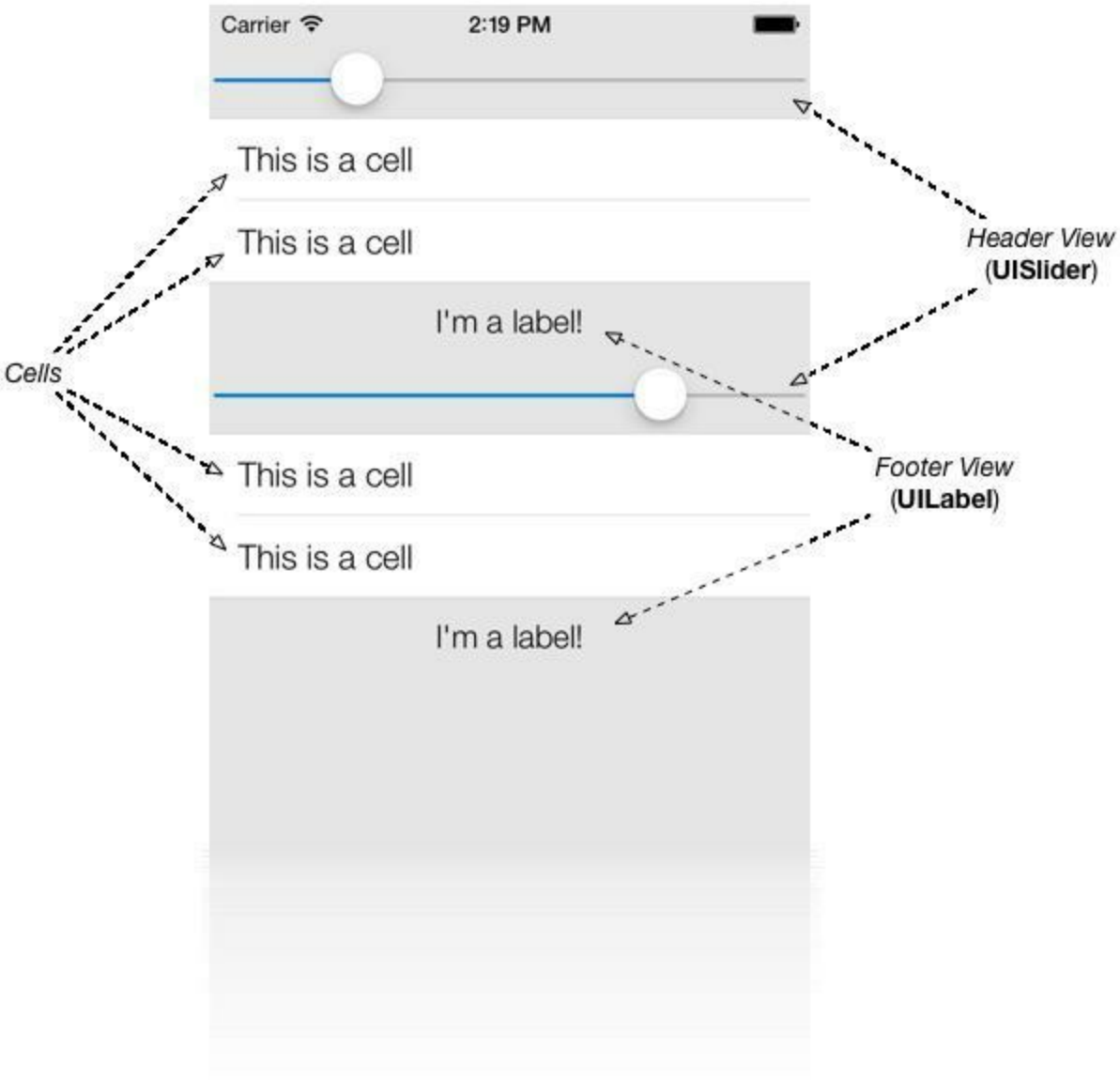


图9-2 针对表格段的表头视图和表尾视图

接下来创建一个针对表格的表头视图。这个表头视图包含两个UIButton对象，其中一个负责切换UITableView对象的编辑模式，另一个负责创建新的BNRItem对象并加入UITableView对

象。可以使用代码创建这个表头视图及其包含的子视图，但是本章将使用XIB文件创建它们。之后，BNRItemsViewController对象在需要显示表头视图时会加载相应的XIB文件。

首先需要编写一些代码。重新打开第8章中的Homepwner.xcodeproj，在BNRItemsViewController.m中添加BNRItemsViewController的类扩展，然后声明一个插座变量并添加两个新方法，代码如下：

```
@interface BNRItemsViewController ()

@property (nonatomic, strong) IBOutlet UIView *headerView;

@end

@implementation BNRItemsViewController

// 这里省略了其他方法

- (IBAction)addNewItem:(id)sender

{

}

- (IBAction)toggleEditingMode:(id)sender

{

}
```

载入XIB文件后，headerView会指向XIB文件中的顶层对象，并且是强引用。指向顶层对象的插座变量必须声明为强引用；相反，当插座变量指向顶层对象所拥有的对象（例如顶层对象的子视图）时，应该使用弱引用。

下面创建一个新的XIB文件。和本书之前创建的XIB文件不同，这个XIB文件和视图控制器的视图无关（BNRItemsViewController作为UITableViewController的子类，可以自行创建其视图）。通常情况下，可以用XIB文件来创建某个视图控制器的视图。但是，也可以在XIB文件中随意创建多个视图并设置层级结构和布局，然后在运行应用时按需载入。

选择File菜单中的New菜单项，然后选择File…选中窗口左侧iOS部分的User Interface，然后选中窗口右侧的Empty模板，最后单击Next按钮（见图9-3）。

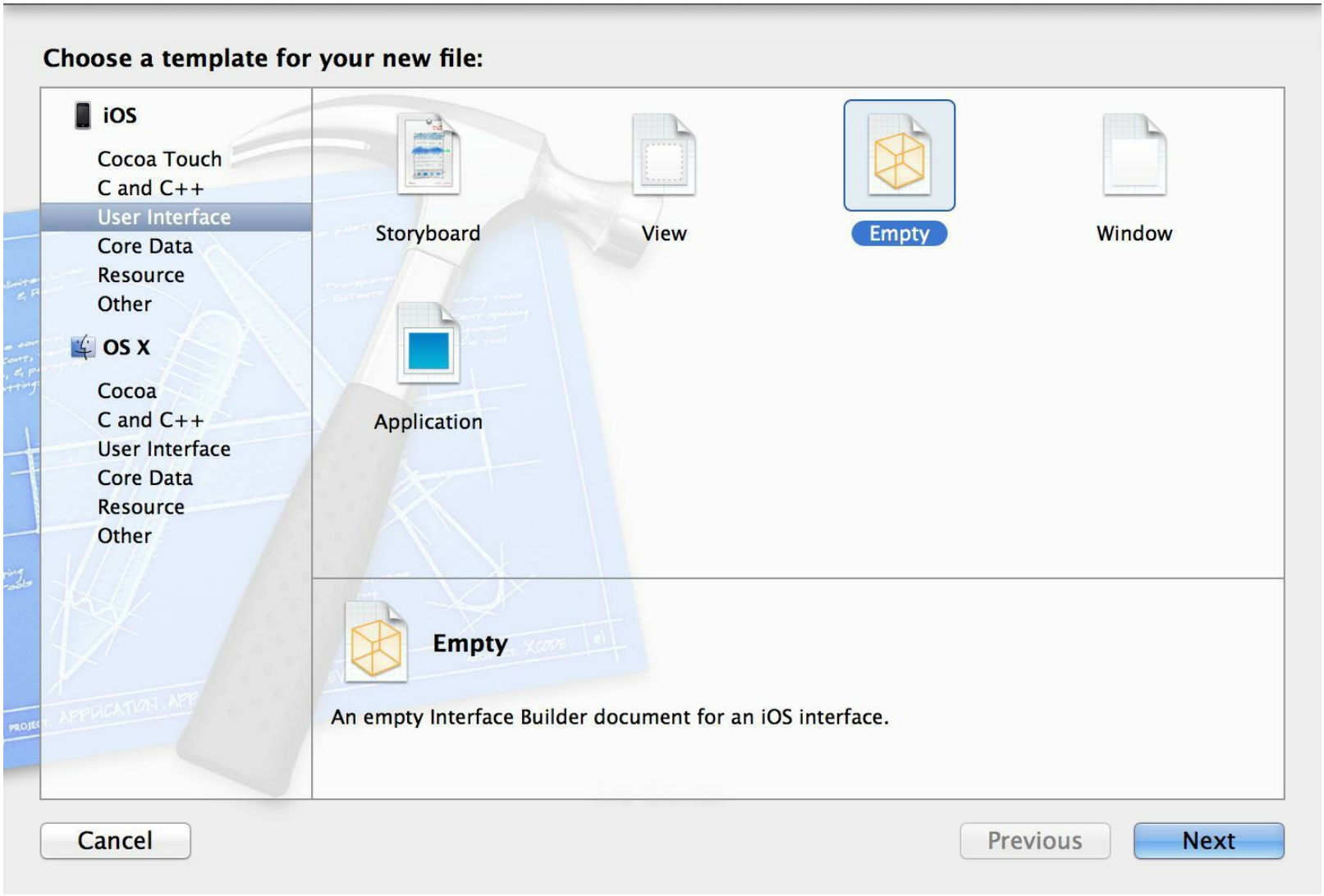


图9-3 创建新的XIB文件

在新出现的面板中选择Device Family下拉菜单中的iPhone，单击Next按钮。Xcode会提示保存文件，将文件名设置为HeaderView.xib，单击Save按钮。

选中HeaderView.xib中的File's Owner，打开标识检视面板，将Class文本框中的UIViewController修改为BNRItemsViewController（见图9-4）。

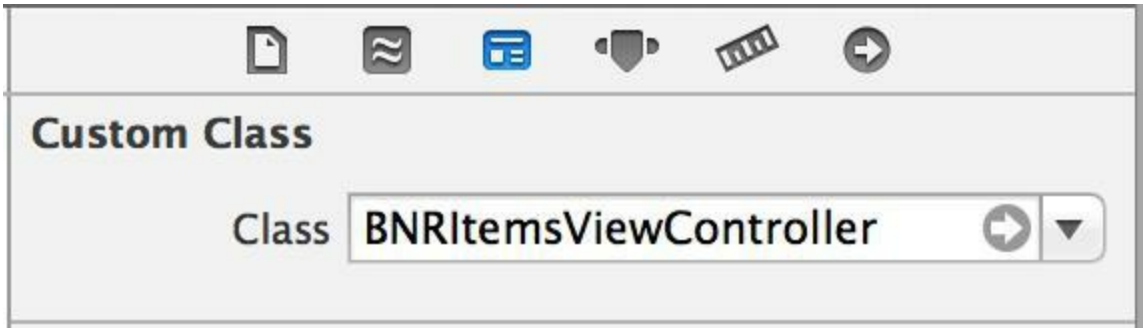


图9-4 修改File'Owner

先拖曳一个UIView对象至画布，然后再拖曳两个UIButton对象至这个UIView对象。现在需要调整UIView对象的大小，但是读者会发现UIView对象的大小被锁定了，无法调整，因此需要解除大小锁定。选中UIView对象并打开属性检视面板，在Simulated Metrics部分中点击标题为

Size的选项列表，选择None（见图9-5）。

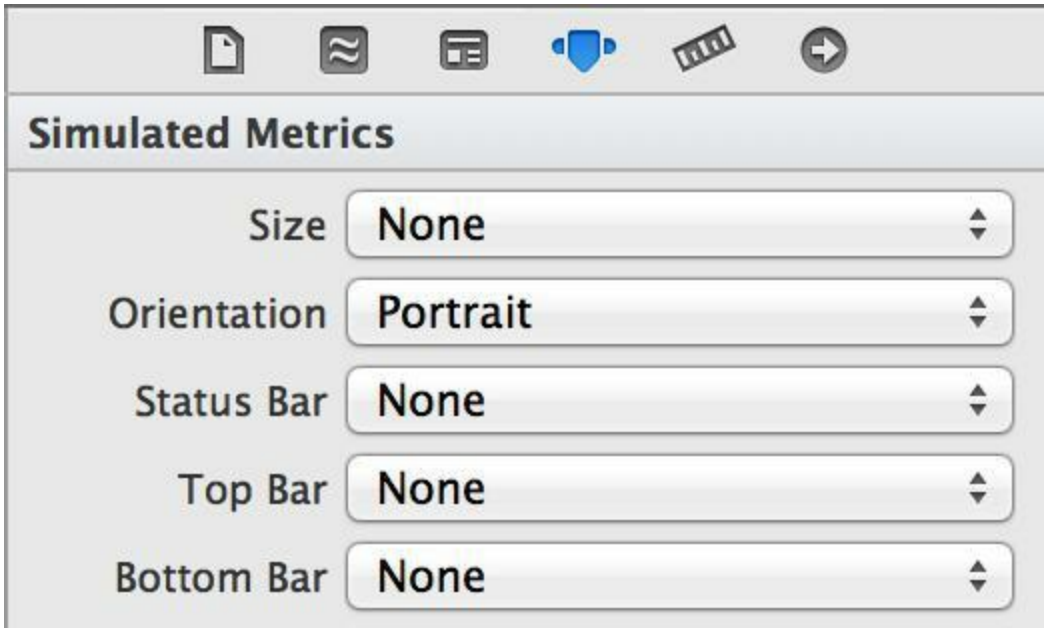


图9-5 解除视图大小锁定

现在可以调整视图大小了，请按图9-6所示调整UIView对象的大小并创建相应的关联。

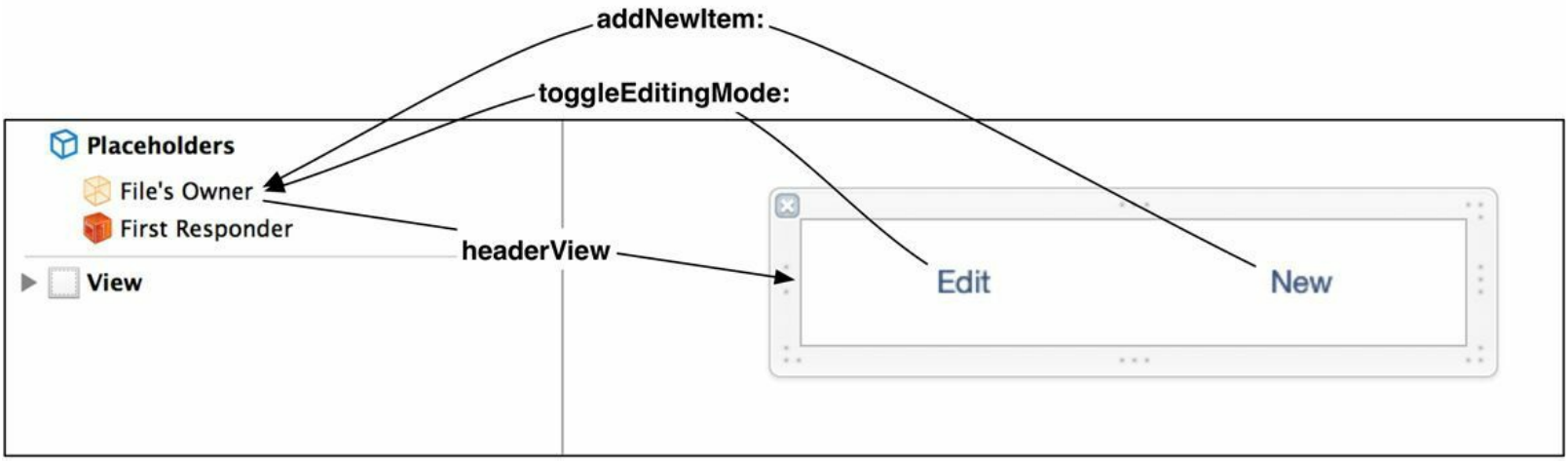


图9-6 HeaderView.xib的布局

还要将UIView对象的背景颜色修改为全透明颜色。具体做法为：选中之前加入的UIView对象并打开属性检视面板。单击标题为Background的颜色的选项列表，选择Clear Color（见图9-7）。

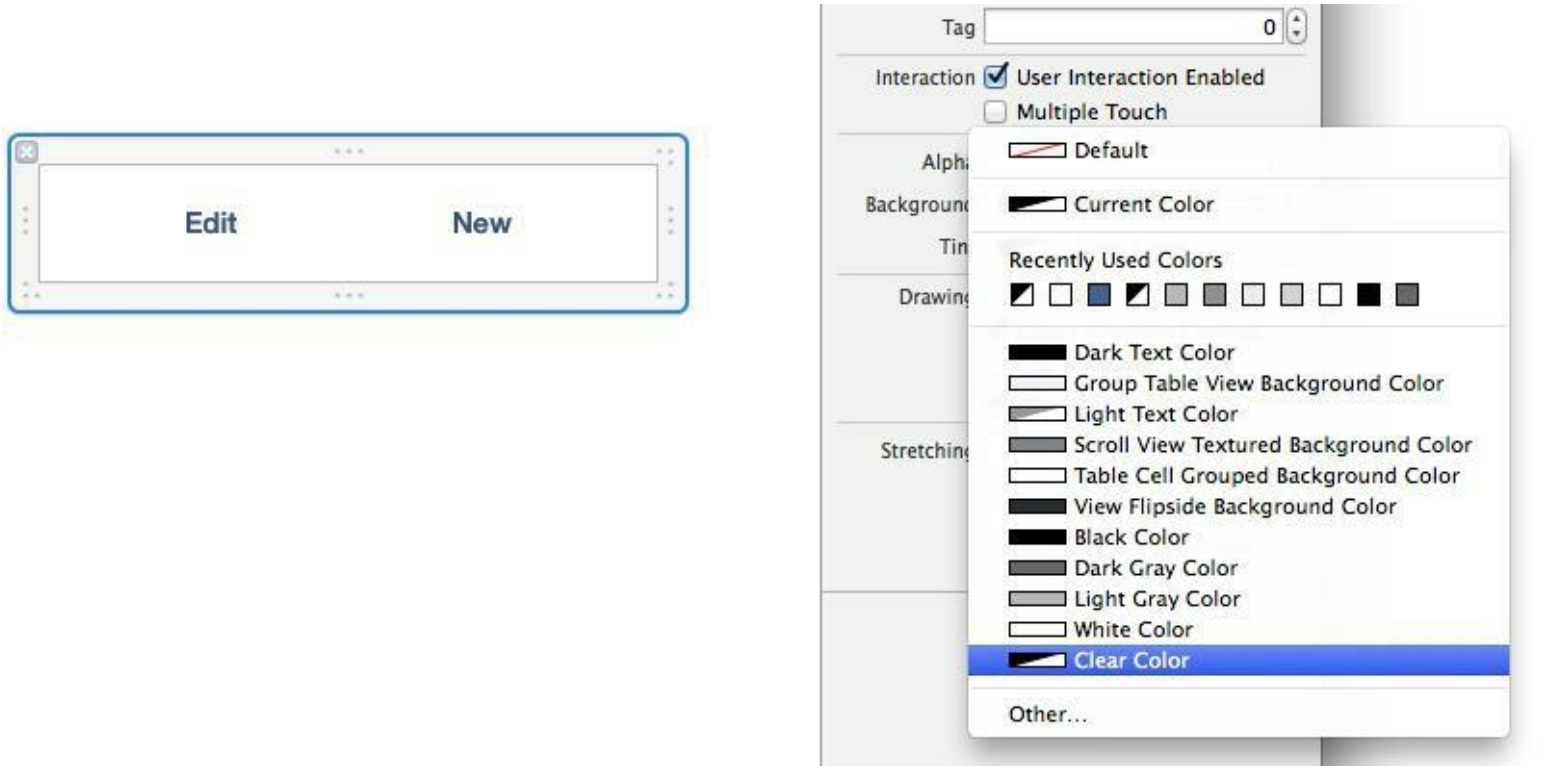


图9-7 将背景色设置为Clear Color

本书之前所创建的XIB文件，都是通过UIViewController类的默认实现自动载入的。以第6章的BNRReminderViewController为例，因为它是UIViewController的子类，所以BNRReminderViewController对象会在需要显示其视图时，自动载入BNRReminderViewController.xib。但是对于HeaderView.xib，则需要编写特定的代码，让BNRItemsViewController对象能够“手动”载入该XIB文件。

使用NSBundle类可以载入指定的XIB文件。该类是“应用程序包”和“应用程序包所包含的可执行文件”之间的接口。通过该类，应用可以访问某个程序包中的文件。向该类发送mainBundle消息可以得到指向主NSBundle对象的指针，该对象是应用在启动时创建的。

得到主NSBundle对象后，就可以要求其载入应用程序包中的某个XIB文件。在BNRItemsViewController.m中实现headerView方法，代码如下：

```
- (UIView *)headerView

{

// 如果还没有载入headerView...

if (!_headerView) {

    // 载入HeaderView.xib

    [[NSBundle mainBundle] loadNibNamed:@"HeaderView"

                                  owner:self
```

```
        options:nil];

    }

    return _headerView;

    }
```

该方法使用了一种名为延迟实例化(Lazy Instantiation)的设计模式:只会在真正需要使用某个对象时再创建它。在某些情况下,这种设计模式可以显着减少内存占用。

调用loadNibNamed:owner:options:时,需要传入XIB文件的文件名。文件名不需要包含后缀,NSBundle会自行判断并处理。此外,这段代码将self作为owner实参(拥有者)传给了NSBundle对象,目的是当BNRItemsViewController对象将XIB文件加载为NIB文件时,使用BNRItemsViewController对象自身替换占位符对象File's Owner。

BNRItemsViewController对象会在第一次收到headerView消息时载入HeaderView.xib,然后为插座变量headerView赋值,并将其指向HeaderView.xib中的顶层UIView对象。当用户按下这个顶层UIView对象中的任何一个按钮时,BNRItemsViewController对象都会收到指定的动作消息。

加载了headerView后,还需要将其设置为UITableView对象的表头视图。在BNRItemsViewController.m的viewDidLoad方法中,添加以下代码:

```
    - (void)viewDidLoad

    {

    [super viewDidLoad];

    [self.tableView registerClass:[UITableViewCell class]

        forCellReuseIdentifier:@"UITableViewCell"];

    UIView *header = self.headerView;

    [self.tableView setTableHeaderView:header];

    }
```

构建并运行应用,UITableView对象会在列表上方显示两个按钮。

XIB文件不仅可以用来创建视图控制器的视图(例如BNRReminderView- Controller.xib),也可以用来固化其他的视图层次结构(例如HeaderView.xib)。实际上,任何对象都可以通过向主NSBundle对象发送loadNibNamed:owner:options:消息手动载入XIB文件。

前面介绍过,UIViewController默认已经实现了通过XIB文件载入视图的功能。其实现原理

和headerView方法的相同，代码也相似。唯一的差别是，UIViewController对象会在载入XIB文件后，将插座变量view关联至XIB文件中指定的UIView对象。UIViewController的loadView方法的代码示例如下（这里列出的代码仅是举例，和UIViewController的loadView方法的实际代码不同）：

```
- (void)loadView
{
// NIB文件位于哪个bundle中？
// 是否为initWithNibName:bundle:方法传了bundle参数？
NSBundle *bundle = [self nibBundle];
if (!bundle) {
    // 使用默认bundle
    bundle = [NSBundle mainBundle];
}
// NIB文件的名称是什么？
// 是否为initWithNibName:bundle:方法传了NIB文件名参数？
NSString *nibName = [self nibName];
if (!nibName) {
    // 使用默认NIB文件名
    nibName = NSStringFromClass([self class]);
}
// 尝试在bundle中查找默认NIB文件
NSString *nibPath = [bundle pathForResource:nibName
    ofType:@"nib"];
// 该NIB文件是否存在？
if (nibPath) {
    // 加载NIB文件（同时还会设置view插座变量）
```

```
        [bundle loadNibNamed:nibName owner:self options:nil];
    } else {
        // 如果没有NIB文件，就创建一个空白视图
        self.view = [[UIView alloc] init];
    }
    }
```

接下来实现toggleEditingMode:。虽然可以直接通过设置UITableView对象的editing属性来切换编辑模式，但是UITableViewController也有一个从UIViewController继承而来的editing属性。当某个UITableViewController对象的editing属性发生变化时，UITableViewController对象会同步修改其UITableView对象的editing属性。

向某个UIViewController对象或UIViewController子类对象发送setEditing: animated:消息，可以设置该对象的editing属性（UITableViewController覆盖了UIViewController的setEditing:animated:方法）。在BNRItemsViewController.m中实现toggleEditingMode:，代码如下：

```
    - (IBAction)toggleEditingMode:(id)sender
    {
    // 如果当前的视图控制对象已经处在编辑模式…
    if (self.isEditing) {
        // 修改按钮文字，提示用户当前的表格状态
        [sender setTitle:@"Edit" forState:UIControlStateNormal];
        // 关闭编辑模式
        [self setEditing:NO animated:YES];
    } else {
        // 修改按钮文字，提示用户当前的表格状态
        [sender setTitle:@"Done" forState:UIControlStateNormal];
        // 开启编辑模式
        [self setEditing:YES animated:YES];
    }
```

}

构建并运行应用，按下Edit按钮，UITableView对象会开启编辑模式（见图9-8）。

图9-8 编辑模式下的UITableView对象

9.2 增加行

通常有两种方式可以在应用运行时为UITableView对象增加行。

•在表视图上方放置添加按钮。如果数据的字段较多，需要显示一个用于输入的详细视图，就可以使用这种方式。例如，在iOS自带的通讯录（Contacts）应用中，点击添加按钮可以进入添加新联系人的详细视图并输入该联系人的信息。

•在UITableViewCell对象左边显示一个绿色加号按钮。在为数据添加一个新字段时可以使用这种方式。例如，在联系人应用中需要为联系人添加生日信息，可以在编辑模式中点击“add birthday（添加生日）”左边的绿色加号按钮。

本节采用的则是另一种方式：在headerView中放置一个标题为New的按钮。当用户按下这个按钮时，就为UITableView对象添加一新行。在BNRItemsViewController.m中实现addNewItem:，代码如下：

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

{

// 创建NSIndexPath对象，代表的位置是：第一个表格段，最后一个表格行

NSInteger lastRow = [self.tableView numberOfRowsInSection:0];

NSIndexPath *indexPath = [NSIndexPath indexPathForRow:lastRow inSection:0];

//将新行插入UITableView对象

[self.tableView insertRowsAtIndexPaths:@[indexPath]

 withRowAnimation:UITableViewRowAnimationTop];

}
```

创建NSIndexPath对象，代表的位置是：第一个表格段，最后一个表格行将新行插入UITableView对象构建并运行应用。按下New按钮，Homepwner会崩溃。查看控制台输出，可以发现崩溃原因是应用抛出了内部冲突异常（internal inconsistency exception）。

这是因为任何一个UITableView对象都要从其数据源获取需要显示的数据，数据源决定UITableView对象需要显示的行数。addNewItem:方法为UITableView对象增加了一新行，使行数从原来的5行增加到了6行。接着，当UITableView对象再次访问其数据源，获取应该显示的行数时，BNRItemsViewController对象会查询BNRItemStore对象并返回allItems数组所包含的指针个数。因为allItems数组没有发生变化，仍旧是5个，所以UITableView对象会由于两者的个数不同而抛出内部冲突异常。

添加表格行时，必须确保UITableView对象当前显示的行数与数据源提供的行数相同。为

此，必须在添加新行前，创建一个新的BNRItem对象并加入BNRItemStore。

在BNRItemsViewController.m的addNewItem:方法中添加以下代码：

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

{

NSInteger lastRow = [self.tableView numberOfRowsInSection:0];

// 创建新的BNRItem对象并将其加入BNRItemStore对象

BNRItem *newItem = [[BNRItemStore sharedStore] createItem];

// 获取新创建的对象在allItems数组中的索引

NSInteger lastRow = [[[BNRItemStore sharedStore] allItems] indexOfObject:newItem];

NSIndexPath *indexPath = [NSIndexPath indexPathForRow:lastRow inSection:0];

//将新行插入UITableView对象

[self.tableView insertRowsAtIndexPaths:@[indexPath]

 withRowAnimation:UITableViewRowAnimationTop];

}
```

构建并运行应用。按下New按钮，UITableView对象会以动画的形式将新创建的行移入表格的末端。这里需要重点指出，视图的责任是将模型对象中的数据呈现给用户，只更新视图而不更新模型对象就会发生数据不一致的错误。

这段代码调用了BNRItemsViewController对象的tableView方法以获取表视图。BNRItemsViewController对象的tableView方法继承自UITableViewController，会返回UITableViewController对象的表视图。虽然也可以向UITableViewController对象发送view消息，并且得到同一个对象，但是因为tableView方法的返回类型是指向UITableView对象的指针，所以可以将其返回的指针直接赋给UITableView类型的指针变量，以便向视图发送UITableView特有的消息（例如insertRowsAtIndexPaths: withRowAnimation:），避免Xcode发出警告信息。

因为可以通过New按钮来为UITableView对象增加新行，所以在BNRItemsView- Controller.m的init方法中删除使用BNRItemStore创建5个随机BNRItem对象的代码：

```
- (instancetype)init

{

// 调用父类的指定初始化方法
```

```
    self = [super initWithStyle:UITableViewStylePlain];

    if (self) {

        for (int i = 0; i < 5; i++) {

            [[BNRItemStore sharedStore] createItem];

        }

    }

    return self;

    }
```

构建并运行应用。UITableView对象不会显示任何内容。按下New按钮，可以为UITableView对象增加新行。

9.3 删除行

在编辑模式下，UITableViewCell对象可能会显示中间有个减号的红色圆圈（见图9-8）。这个红色圆圈是删除控件（deletion control），按下删除控件可以删除其所属的那个表格行。但是Homepwner中的删除控件不会执行任何操作（请读者自己尝试）。这是因为UITableView对象在删除某一个表格行前，会先向数据源发送一条特定的消息，得到确认后才会有实际的操作。

要删除Homepwner中的某个表格行（UITableViewCell对象），必须执行两步：①从UITableView对象删除指定的UITableViewCell对象。②找到和需要删除的UITableViewCell对象对应的BNRItem对象，也将其从BNRItemStore中删除。为了完成第二步，BNRItemStore必须实现新的方法，用于移除指定的BNRItem对象。在BNRItemStore.h中声明新方法removeItem:，代码如下：

```
@interface BNRItemStore : NSObject

+ (BNRItemStore *)sharedStore;

@property (nonatomic, strong, readonly) NSArray *allItems;

- (BNRItem *)createItem;

- (void)removeItem:(BNRItem *)item;

@end
```

在BNRItemStore.m中实现removeItem:，代码如下：

```
- (void)removeItem:(BNRItem *)item

{

[self.privateItems removeObjectIdenticalTo:item];

}
```

removeItem方法调用了NSMutableArray的removeObjectIdenticalTo:。除了removeObjectIdenticalTo:，还可以使用removeObject:。这两个方法的差别是，removeObject:会枚举数组，向每一个对象发送isEqual:消息。isEqual:的作用是判断当前对象和传入对象所包含的数据是否相等（返回YES或NO）。不同的类可以根据自身情况覆盖isEqual:并实现相应的逻辑。以BNRItem为例，当两个BNRItem对象的valueInDollars相等时，可以认为这两个对象是相等的。

removeObjectIdenticalTo:方法不会比较对象所包含的数据，只会比较指向对象的指针。因此，该方法只会移除数组所保存的那些和传入对象指针完全相同的指针。虽然本章没有覆盖BNRItem的isEqual:来实现特殊的比较逻辑，但是将来可能会。因此，removeItem:方法应该使用removeObjectIdenticalTo:删除指定的BNRItem:对象。

接下来为BNRItemsViewController实现tableView:commitEditingStyle: forRowAtIndexPath:，该方法是UITableViewDataSource协议所声明的方法之一（UITableView对象会向BNRItemsViewController对象发送这个消息。注意：UITableView对象的数据源是BNRItemsViewController对象，而不是负责保存数据的BNRItemStore对象）。

UITableView对象在向其数据源发送tableView:commitEditingStyle: forRowAtIndexPath:消息时，会传入三个实参。第一个实参是发送该消息的UITableView对象。第二个实参是UITableViewCellEditingStyle类型的常数（删除表格行时，传入的是UITableViewCellEditingStyleDelete）。第三个实参是一个NSIndexPath对象，其中包含相应表格行所在的表格段索引和行索引。

在BNRItemsViewController.m中实现tableView:commitEditingStyle: forRowAtIndexPath:，先从BNRItemStore对象中删除相应的BNRItem对象，然后向UITableView对象发送deleteRowsAtIndexPaths:withRowAnimation:消息，删除表格视图中的相应表格行，代码如下：

```
- (void)tableView:(UITableView *)tableView

commitEditingStyle:(UITableViewCellEditingStyle)editingStyle

forRowAtIndexPath:(NSIndexPath *)indexPath

{

// 如果UITableView对象请求确认的是删除操作……

if (editingStyle == UITableViewCellEditingStyleDelete)

{

    NSArray *items = [[BNRItemStore sharedStore] allItems];

    BNRItem *item = items[indexPath.row];

    [[BNRItemStore sharedStore] removeItem:item];

    // 还要删除表格视图中的相应表格行（带动画效果）

    [tableView deleteRowsAtIndexPaths:@[indexPath]

withRowAnimation:UITableViewRowAnimationFade];

}

}
```

构建并运行应用。先加入若干新行，然后删除其中的某个表格行，Homepwner的UITableView对象能正确地执行上述操作。最后请读者尝试“swipe-to-delete（滑动删除）”功能，

在表格行中向左滑动，应该可以看见一个红色的删除按钮，点击删除按钮也可以删除表格行。

9.4 移动行

要改变UITableView对象所显示的行的排列位置，需要为数据源实现另一个源自UITableViewDataSource协议的方法：tableView:moveRowAtIndexPath:。

9.3节已介绍了删除表格行时需要向UITableView对象发送deleteRowsAtIndexPaths:withRowAnimation:消息，以执行删除操作。但是移动表格行不需要，UITableView对象会自动执行移动操作，然后向数据源发送tableView:moveRowAtIndexPath:toIndexPath:消息，报告相关的移动信息。因此只需要为数据源实现tableView:moveRowAtIndexPath: toIndexPath:方法，然后根据传入的移动信息更新数据即可。

在为数据源实现tableView:moveRowAtIndexPath:toIndexPath:前，需要先为BNRItemStore增加一个方法，使BNRItemStore对象能够改变某个BNRItems对象在allItems数组中的位置。在BNRItemStore.h中声明方法，代码如下：

```
- (void)moveItemAtIndex:(NSUInteger)fromIndex
 toIndex:(NSUInteger)toIndex;
```

在BNRItemStore.m中实现该方法，代码如下：

```
- (void)moveItemAtIndex:(NSUInteger)fromIndex
            toIndex:(NSUInteger)toIndex
{
if (fromIndex == toIndex) {
    return;
}
// 得到要移动的对象的指针，以便稍后能将其插入新的位置
BNRItem *item = self.privateItems[fromIndex];
// 将item从allItems数组中移除
[self.privateItems removeObjectAtIndex:fromIndex];
//根据新的索引位置，将item插回allItems数组
[self.privateItems insertObject:item atIndex:toIndex];
}
```

接下来，在BNRItemsViewController.m中实现tableView:moveRowAtIndexPath: toIndexPath:，更新BNRItemStore对象，代码如下：

```
- (void)tableView:(UITableView *)tableView

moveRowAtIndexPath:(NSIndexPath *)sourceIndexPath

 toIndexPath:(NSIndexPath *)destinationIndexPath

{

[[BNRItemStore sharedStore] moveItemAtIndex:sourceIndexPath.row

            toIndex:destinationIndexPath.row];

}
```

构建并运行应用。当UITableView对象处于编辑模式时，会在每个表格行的右侧显示一个换位控件（有三条横线的控件）。按住换位控件后拖动，可以将相应的表格行移动至新的位置（见图9-9）。

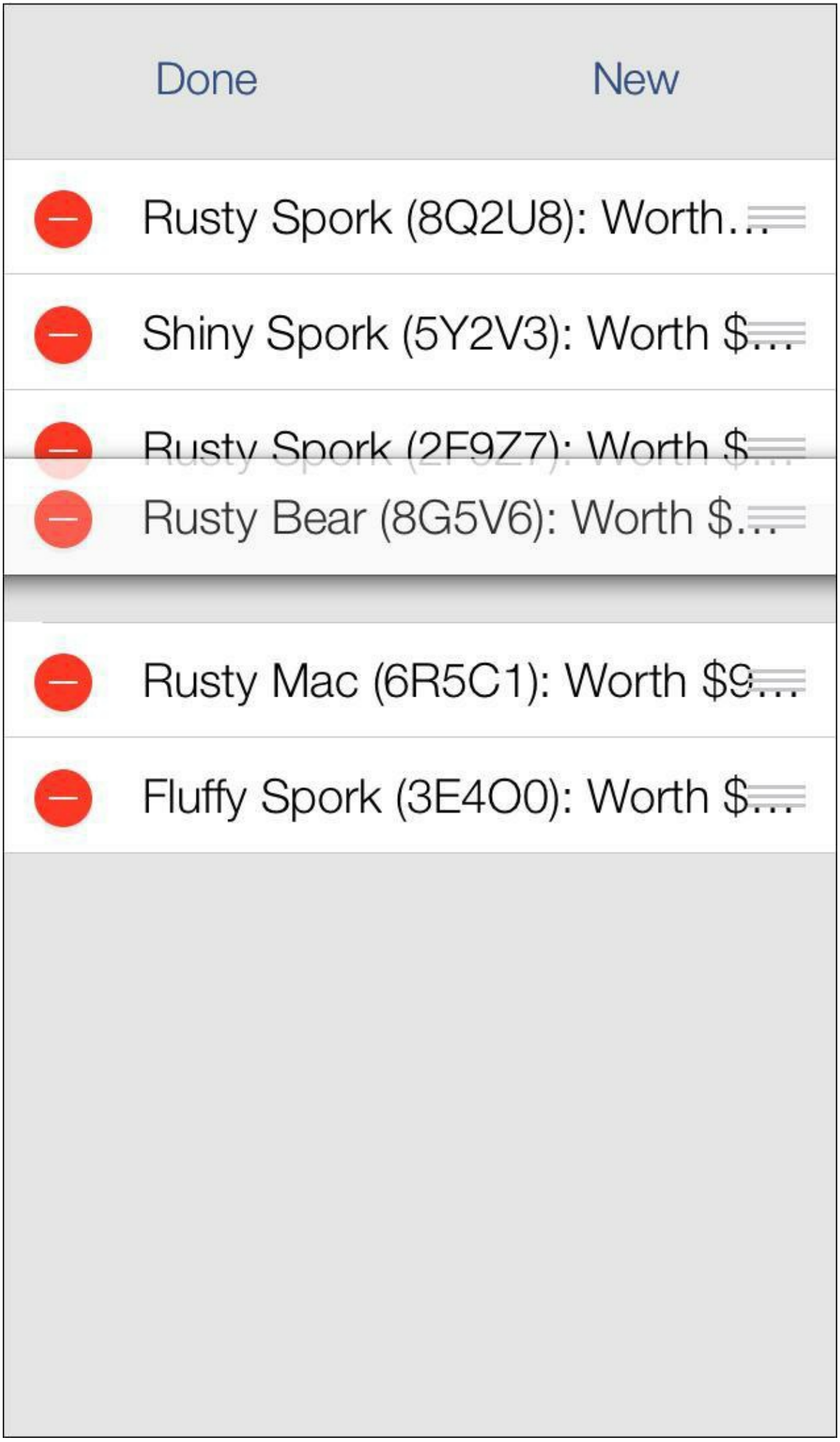

Done
New
Rusty Spork (8Q2U8): Worth…
Shiny Spork (5Y2V3): Worth $…
Rusty Spork (2F9Z7): Worth $…
Rusty Bear (8G5V6): Worth $…
Rusty Mac (6R5C1): Worth $9…
Fluffy Spork (3E4O0): Worth $…

图9-9 移动中的某个表格行

UITableView对象的数据源只需要实现tableView:moveRowAtIndexPath: toIndexPath:，相应的UITableView对象就会显示换位控件。借助Objective-C的语言特性，UITableView对象可以在运行时检查其数据源是否实现了tableView: moveRowAtIndexPath:toIndexPath:。如果已实现，UITableView对象就会显示换位控件，反之则不会。

9.5 初级练习：更改“删除”按钮的标题

删除UITableView对象中的某个表格行时，相应的UITableViewCell对象会在其右侧显示一个标题为“Delete”的按钮。请读者将该按钮的标题改为“Remove”。

9.6 中级练习：禁止移动某个表格行

编写代码，使Homepwner中的UITableView对象能够始终在最后一行显示一个标题为No more items!的UITableViewCell对象（与第8章的练习类似），然后禁止移动该表格行。

9.7 高级练习：彻底禁止移动某个表格行

完成中级练习后，读者可能会发现一个问题：虽然不能移动No more items!，但是可以将其他行移动至No more items!的下方。更新代码，解决上述问题，使No more items!能够始终位于表格视图的底部。

第10章 UINavigationController

第6章介绍了UITabBarController对象，通过使用该对象，用户可以切换不同的屏幕。当要切换的各个屏幕之间没有相互依存的关系时，该对象可以很好地完成任务。但是当多个屏幕互有关联时，就要使用另一种视图控制器。

以iOS自带的Setting(设置)应用为例，设置应用拥有多个互有关联的窗口(见图10-1)，其中包括一组设定选项(例如“声音”)，针对每个设定选项的详细设置视图及针对每个详细设置选择视图。这类界面称为垂直界面(drill-down interface)。

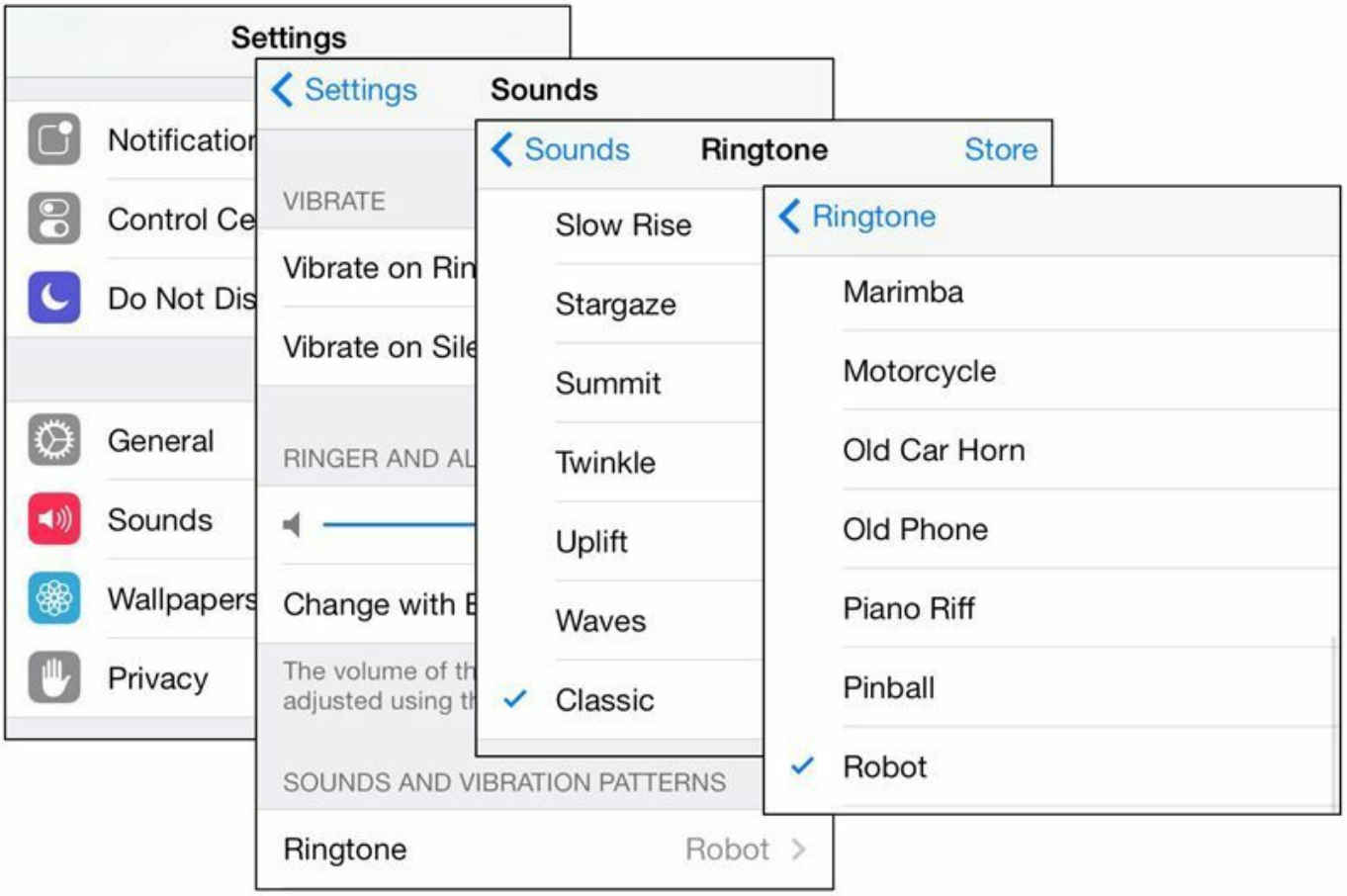


图10-1 设置应用的垂直界面

本章介绍如何通过使用UINavigationController对象为Homepwner加入垂直界面，使用户能够查看并编辑BNRItem对象的详细信息(见图10-2)。

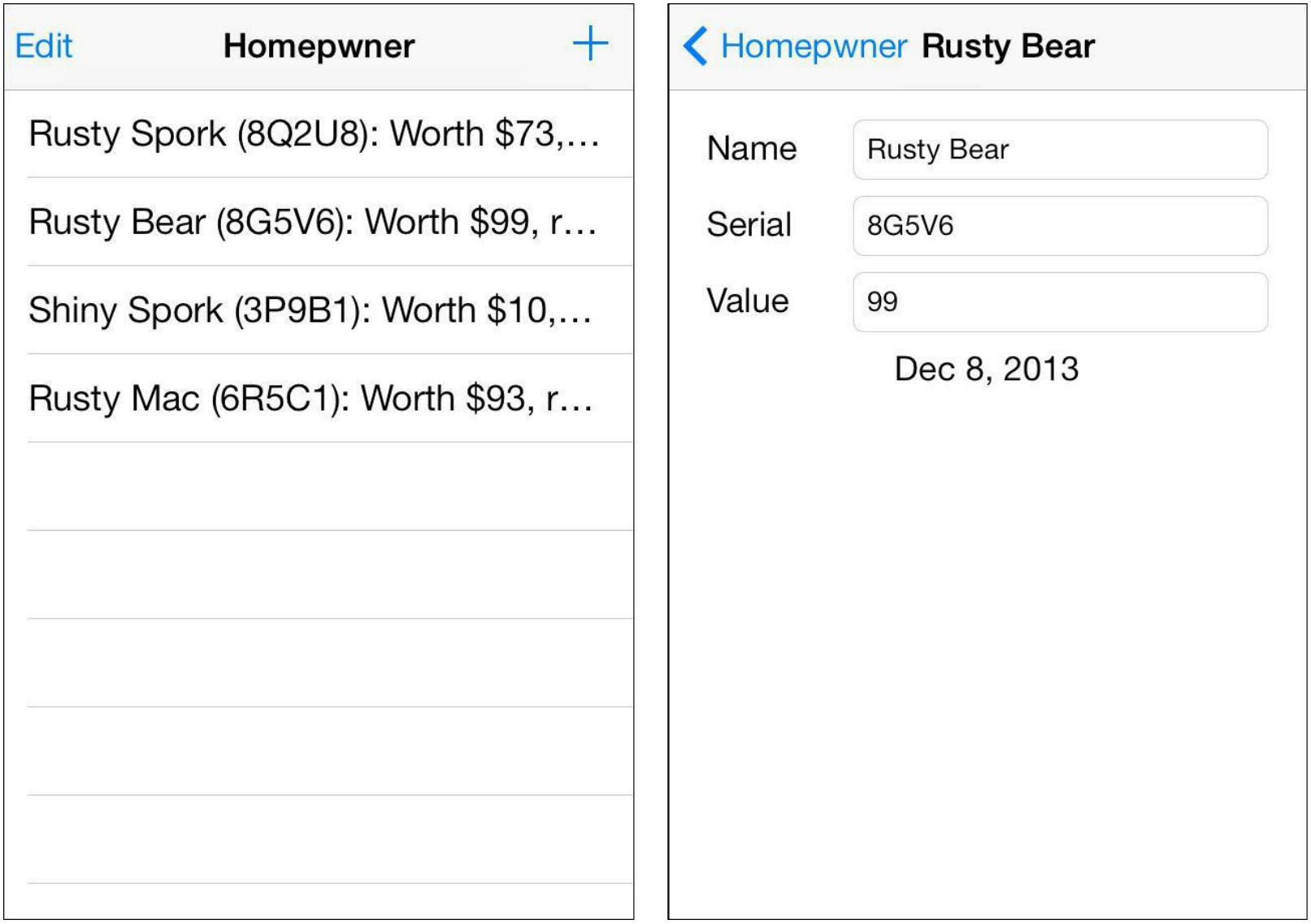


图10-2 使用UINavigationController对象的Homepwner

10.1 UINavigationController对象

当某个应用通过UINavigationController对象显示多个屏幕的信息时，相应的UINavigationController对象会以栈的形式保存所有屏幕的信息。这里的栈是一个数组对象，保存的都是UIViewController对象。一个UIViewController对象的视图对应一个屏幕。只有位于栈顶的UIViewController对象，其视图才是可见的。

初始化UINavigationController对象时，需要传入一个UIViewController对象。这个UIViewController对象将成为UINavigationController对象的根视图控制器（root view controller），且根视图控制器将永远位于栈底。应用可以在运行时向UINavigationController的栈压入更多的视图控制器。

将某个视图控制器压入UINavigationController对象的栈时，新加入的视图控制器的视图会从窗口右侧推入。出栈时，UINavigationController对象会移出位于栈顶的视图控制器，其视图会向窗口右侧推出，然后用户会看见仅次于栈顶位置的视图控制器的视图。

图10-3显示的是一个包含两个视图控制器的UINavigationController对象，其中一个是根视图控制器，另一个是位于栈顶的其他视图控制器。因为后一个视图控制器位于栈顶，所以用户可以看到该视图控制器的视图。

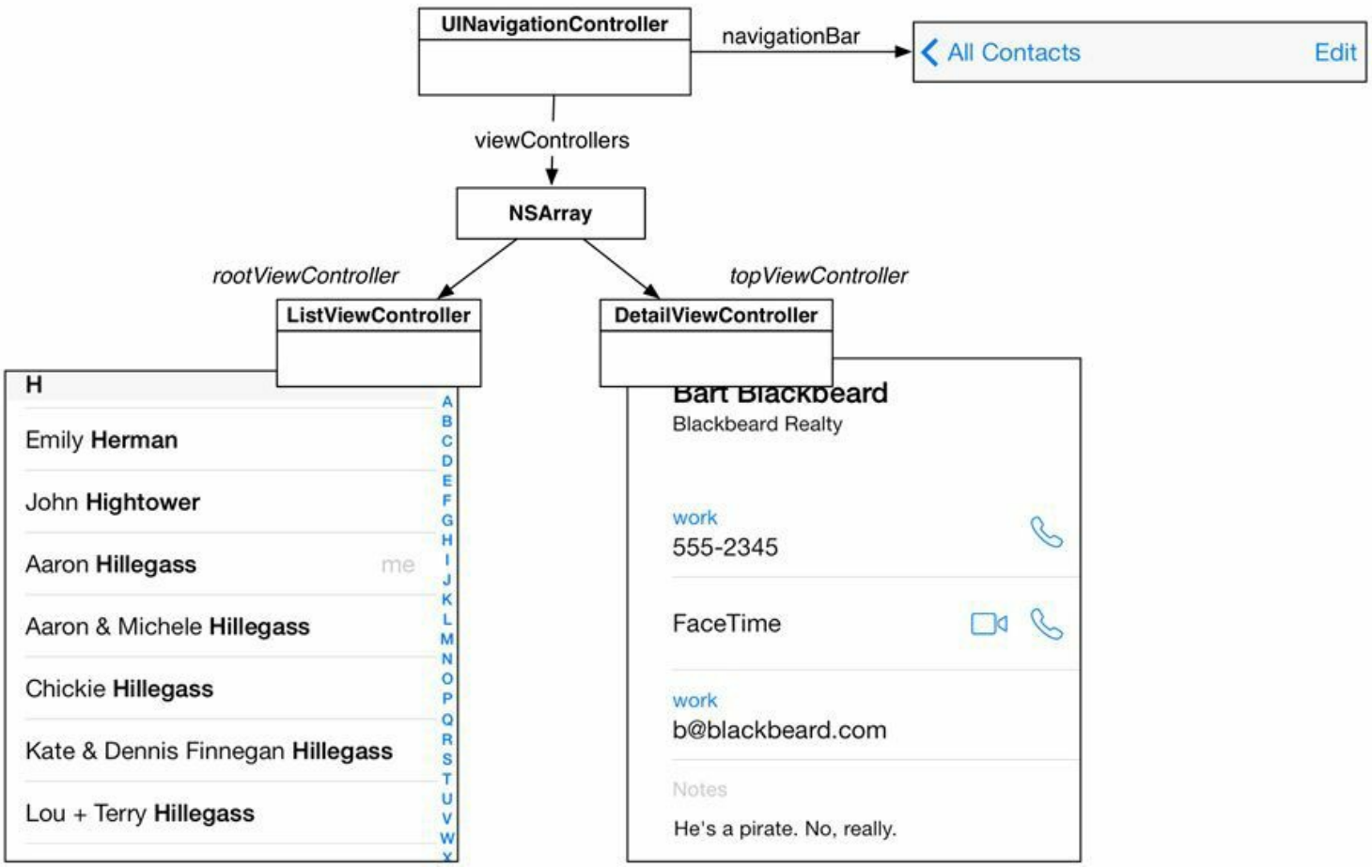


图10-3 某个UINavigationController对象的栈

类似于UITabBarController对象，UINavigationController对象有一个名为viewControllers的属性，指向一个负责保存视图控制器的数组对象。在这个数组对象中，根视图控制器是第一个对

象。当应用将某个视图控制器压入栈后，UINavigationController对象会将新加入的对象加在viewControllers数组的末尾。因此，该数组中的最后一个视图控制器会位于栈的顶部。UINavigationController对象的topViewController属性是一个指针，指向当前位于栈顶的视图控制器。

UINavigationController是UIViewController的子类，所以UINavigationController对象也有自己的视图。该对象的视图有两个子视图：一个是UINavigationBar对象，另一个是topViewController的视图（见图10-4）。和其他视图控制器一样，可以将UINavigationController对象设置为UIWindow对象的rootViewController，从而将该对象的视图作为子视图加入窗口。

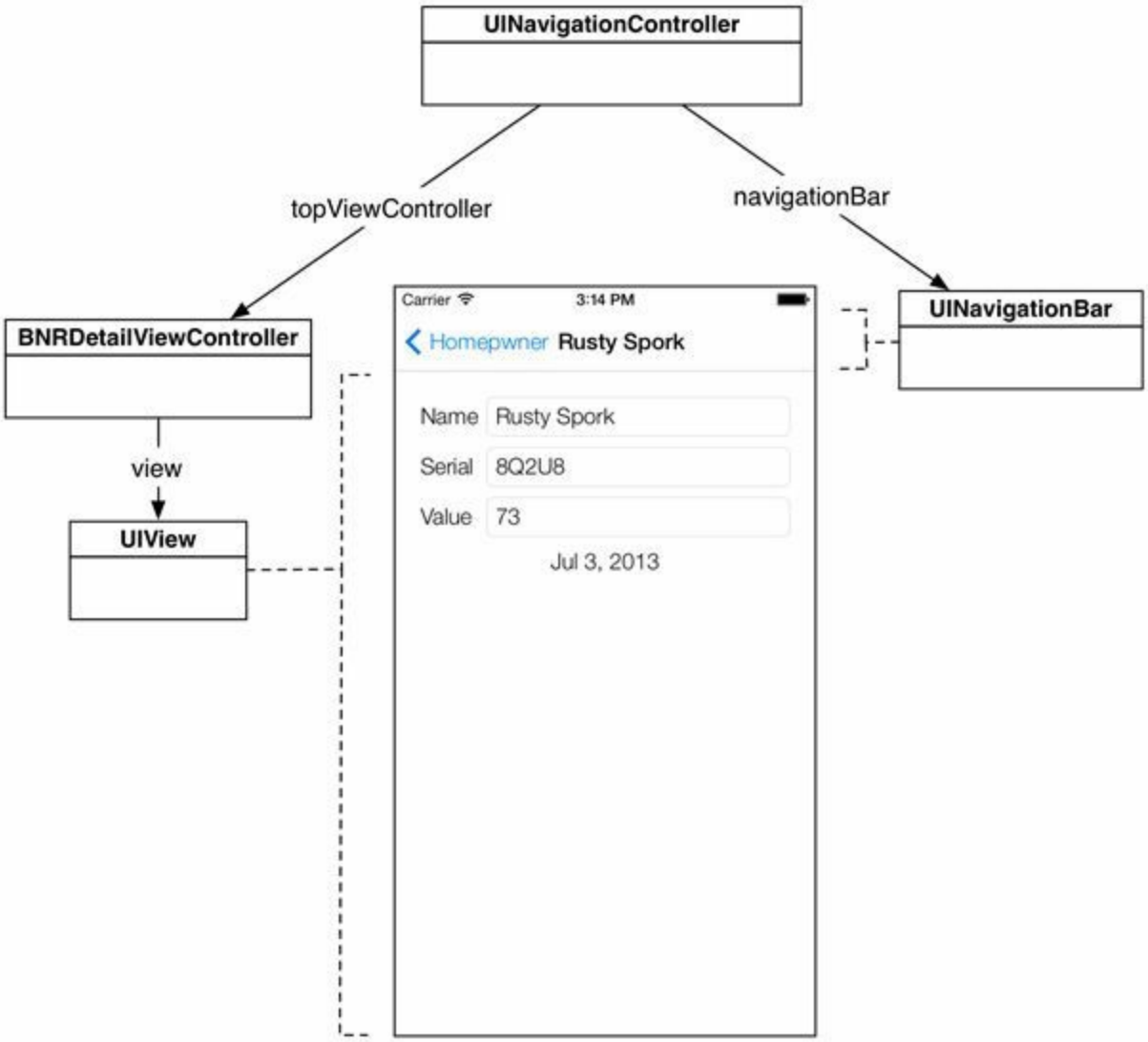


图10-4 UINavigationController对象的视图示例

本章首先为Homepwner添加一个UINavigationController对象，然后将BNRItems-ViewController对象设置为UINavigationController对象的rootViewController。此外，还要创建一个UIViewController子类。当用户选中UITableView对象的某个表格行时，Homepwner要创建该子类的对象并将新创建的对象压入UINavigationController栈，使相应的视图能够推入窗口。用户通过该视图控制器，可以查看并修改选中的BNRItem对象的属性。更新后的Homepwner应用对象图如图10-5所示。

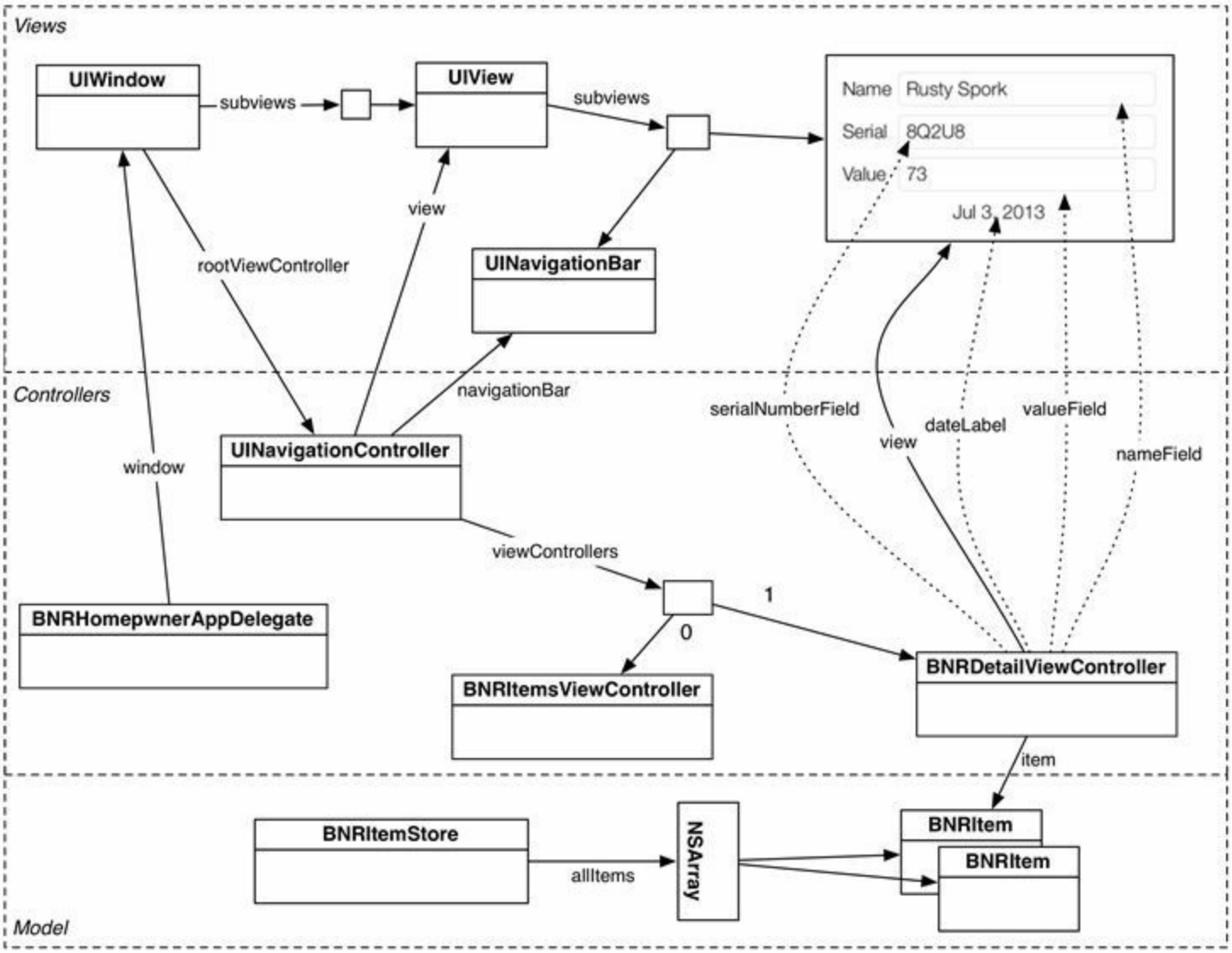


图10-5 Homepwner应用对象图

图10-5列出了较多的对象，可见Homepwner已经初具规模。通过视图控制器和UINavigationController对象，能够很方便地处理此类复杂的对象图。编写iOS应用时，要将各个UIViewController对象视为独立的个体，这点很重要。余下的大量工作可以由Cocoa Touch来完成。

接下来为Homepwner添加一个UINavigationController对象。打开Homepwner项目并在项目导航面板中选中BNRAppDelegate.m。UINavigationController对象的使用方法很简单，只需要为其设置一个根视图控制器，然后将它的视图添加到屏幕上即可。

在application:didFinishLaunchingWithOptions:中先创建一个BNRItemsViewController对象，然后创建一个UINavigationController对象并将之前创建的BNRItemsViewController对象设置为根视图控制器，最后将UINavigationController对象设置为UIWindow对象的根视图控制器，代码如下：

- (BOOL)application:(UIApplication *)application

didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

```
{
self.window = [[UIWindow alloc]
initWithFrame:UIScreen.mainScreen.bounds]];
// 在这里添加应用启动后的初始化代码
BNRItemsViewController *itemsViewController =
[[BNRItemsViewController alloc] init];
// 创建UINavigationController对象
// 该对象的栈只包含itemsViewController
UINavigationController *navController = [[UINavigationController alloc]
initWithRootViewController:itemsViewController];
self.window.rootViewController = itemsViewController;
// 将UINavigationController对象设置为UIWindow对象的根视图控制器,
// 这样就可以将UINavigationController对象的视图添加到屏幕上
self.window.rootViewController = navController;
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}
```

构建并运行应用。除了屏幕顶部新出现的UINavigationBar对象(见图10-6), Homepwner看上去应该和之前的相同。BNRItemsViewController对象的视图大小会发生变化以适应带UINavigationBar对象的屏幕。这项工作是由UINavigationController完成的。

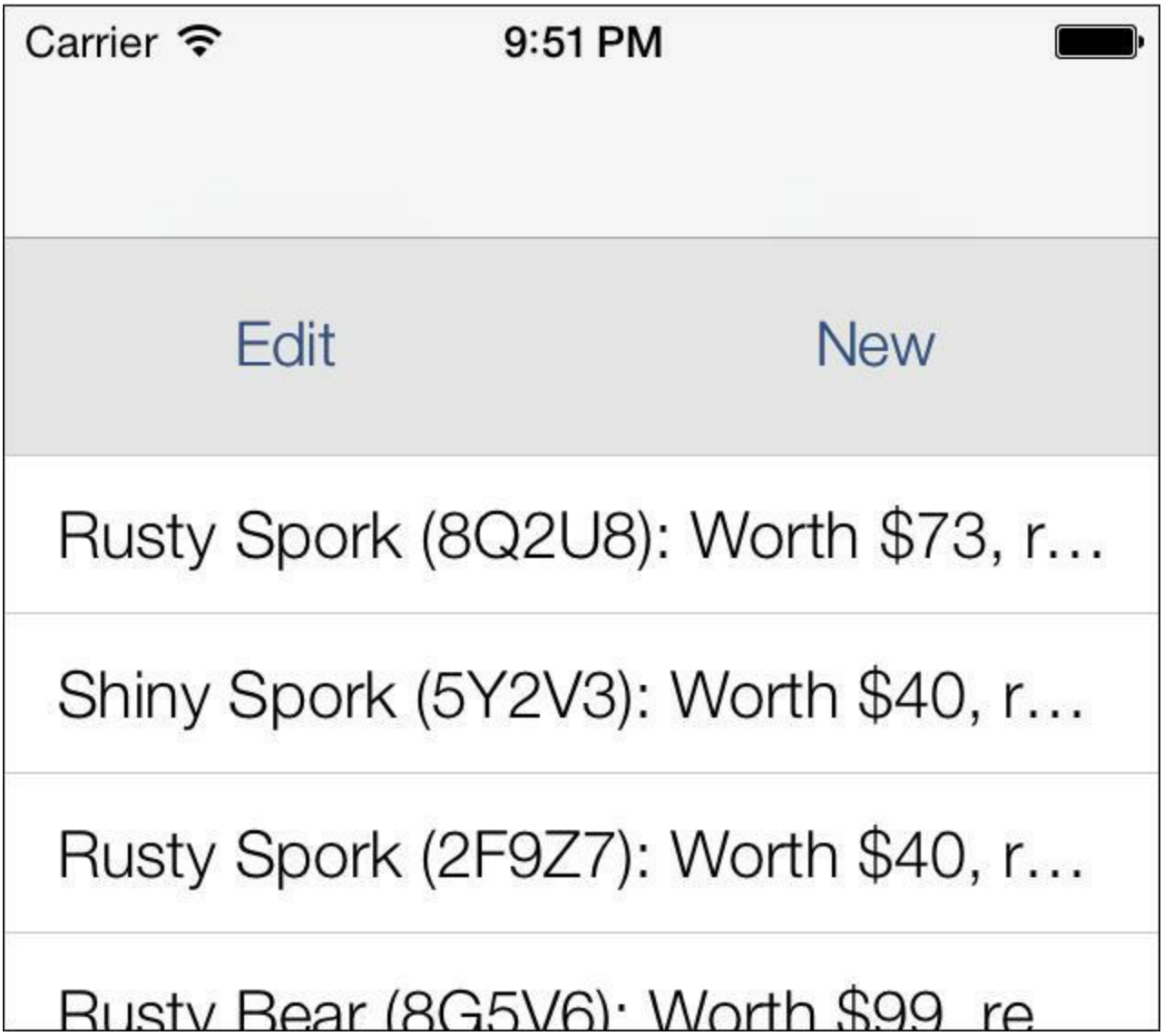


图10-6 Homepwner会在屏幕顶部显示一个空白的导航条

10.2 额外的视图控制器

要让Homepwner的UINavigationController对象真正发挥作用，还要将另一个视图控制器压入UINavigationController对象的栈。选择File菜单中的New菜单项，然后选择File…选中窗口左侧iOS部分的Cocoa Touch，然后选中窗口右侧的Objective-C class，最后单击Next按钮。在新出现的面板中，在Class文本框中输入BNRDetailViewController，在Subclass of下拉菜单中选择UIViewController，勾选选择框With XIB for user interface，单击Next按钮（见图10-7）。Xcode会提示保存文件，单击Save按钮。

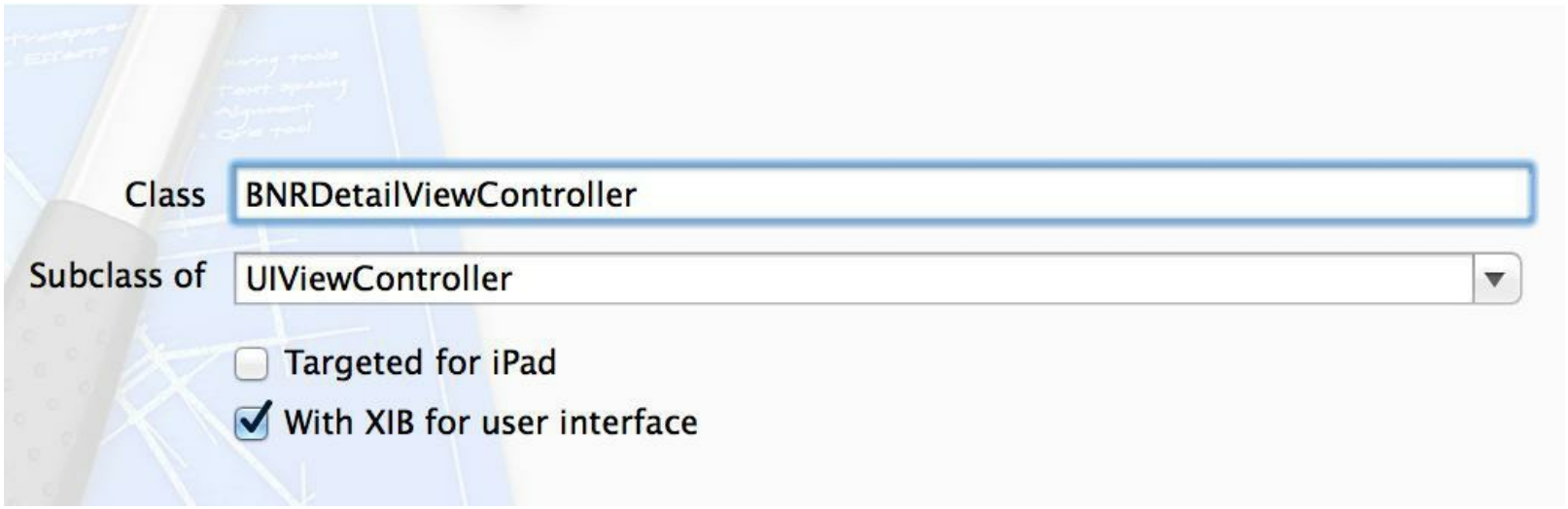


图10-7 创建UIViewController子类及相应的XIB文件

在BNRDetailViewController.m中删除@implementation指令和@end指令间的所有代码，修改后的文件内容如下：

```
#import “BNRDetailViewController.h”

@interface BNRDetailViewController（）

@end

@implementation BNRDetailViewController

@end
```

下面要为Homepwner添加这些功能：当用户点击UITableView对象中的某个表格行时，Homepwner要显示一个新的屏幕。针对选中的BNRItem对象的每一个属性，这个新的屏幕要显示一个可以编辑的文本框。负责显示BNRItem对象信息的视图应该交由BNRDetail-ViewController控制。

为了显示BNRItem对象的四个属性，需要在BNRDetailViewController中为每一个属性创建一个视图并声明相应的插座变量。下面来添加这些插座变量并在XIB文件中创建相应的关联。

本章之前的例子在创建关联时，需要执行独立三步：①在头文件中声明插座变量。②在XIB文件中设置界面。③在XIB文件中创建关联。Xcode提供了一种快捷途径，可以一次完成这三

步。

首先，在项目导航面板中选中BNRDetailViewController.xib。然后，在项目导航面板中按住Option并单击BNRDetailViewController.m，Xcode会在辅助编辑器（assistant editor）中显示该文件，并且之前打开的BNRDetailViewController.xib仍然可见。（找到位于工作空间顶部的Editor控件，单击中间的按钮可以打开/关闭辅助编辑器。显示辅助编辑器的快捷键是Command-Option-回车，返回标准编辑器的快捷键是Command-回车。）

接下来要打开对象库面板，以便将要使用的子视图加入顶层视图。找到位于工作空间右上角的View控件，单击右边的按钮可以打开工具区域（也可以使用快捷键Command-Option-0）。

这时的Xcode窗口会被各种面板占据，下面要暂时腾出一点空间。首先单击View控件左侧的按钮（快捷键是Command-0），隐藏导航面板。然后单击位于编辑区域左下角的切换按钮，将dock由大纲视图切换为图标视图。调整布局后的工作空间如图10-8所示。

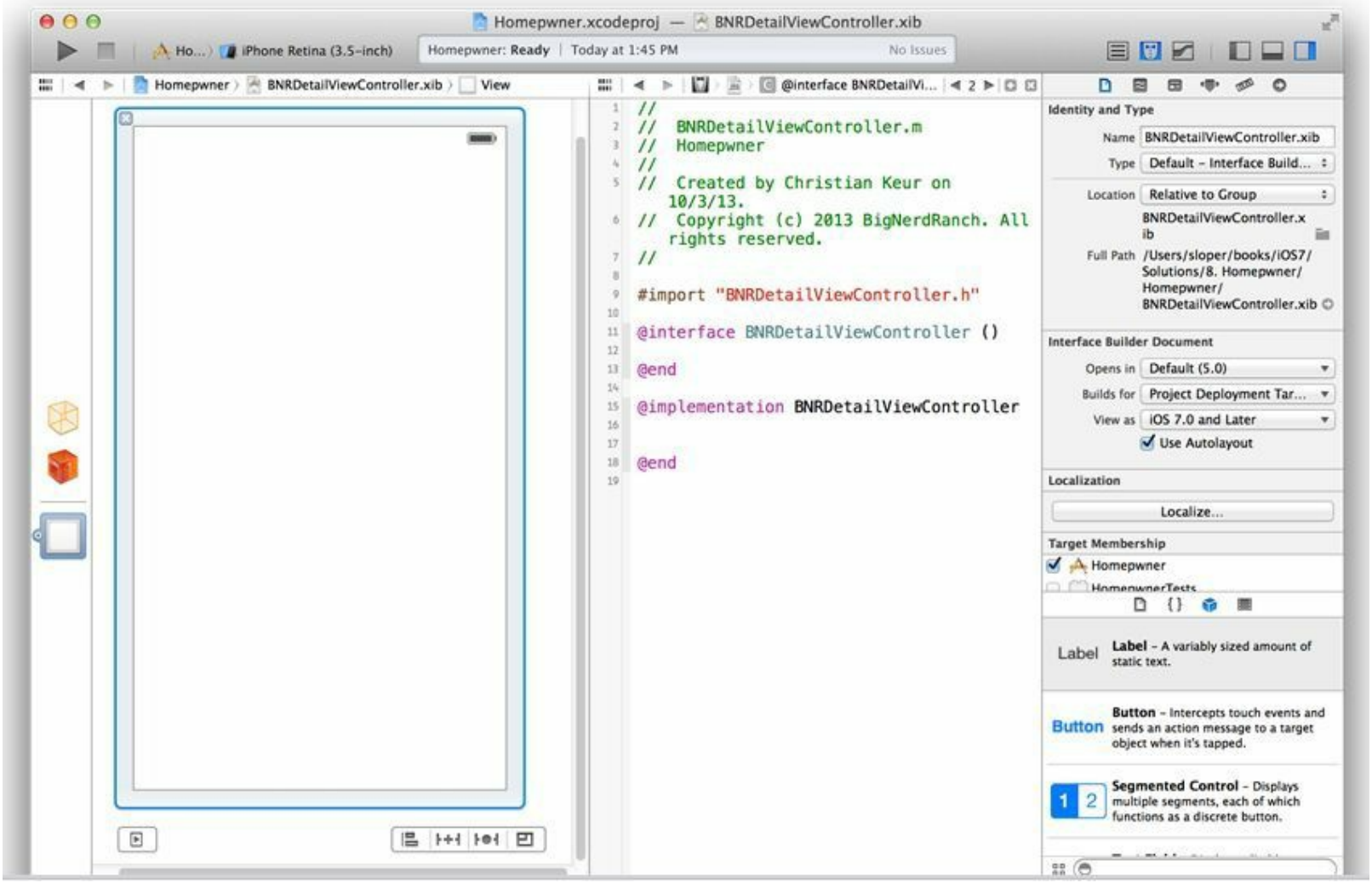


图10-8 调整布局后的工作空间

将四个UILabel对象和三个UITextField对象拖曳至画布区域中的顶层视图，然后根据图10-9进行设置。

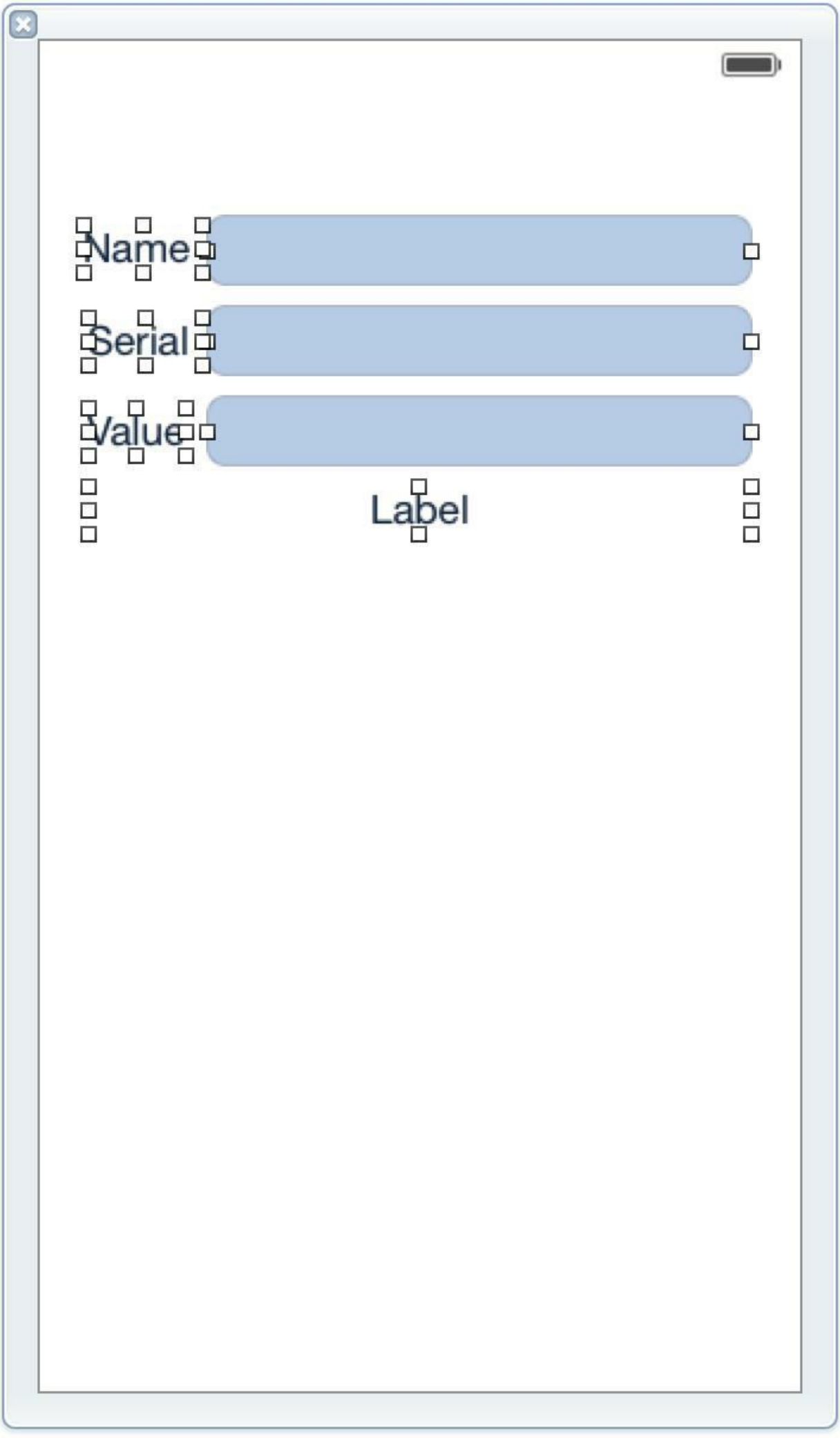

Name
Serial
Value
Label

图10-9 设置后的BNRDetailViewController.xib

请注意，不要将这些子视图放置在view的最顶端。在视图控制器中，view会衬于UINavigationBar的下方，导致UINavigationBar会遮挡view最顶端的内容（对于UITabBar也是同样的情况，因此也不要将子视图放置在view的最底端）。为了方便开发者设置用户界面，Interface Builder为最顶层视图提供了Simulated Metrics，用来预览用户界面的各种外观设置效果，例如顶部带有导航栏或底部带有标签栏的效果。此外，Simulated Metrics中还包括大小、方向、状态栏等设置选项，请读者自行尝试调整这些选项。

首先选中最顶层视图，然后打开属性检视面板，面板顶部显示了Simulated Metrics的一组设置选项，请读者将Top Bar设置为Translucent Navigation Bar（见图10-10）。

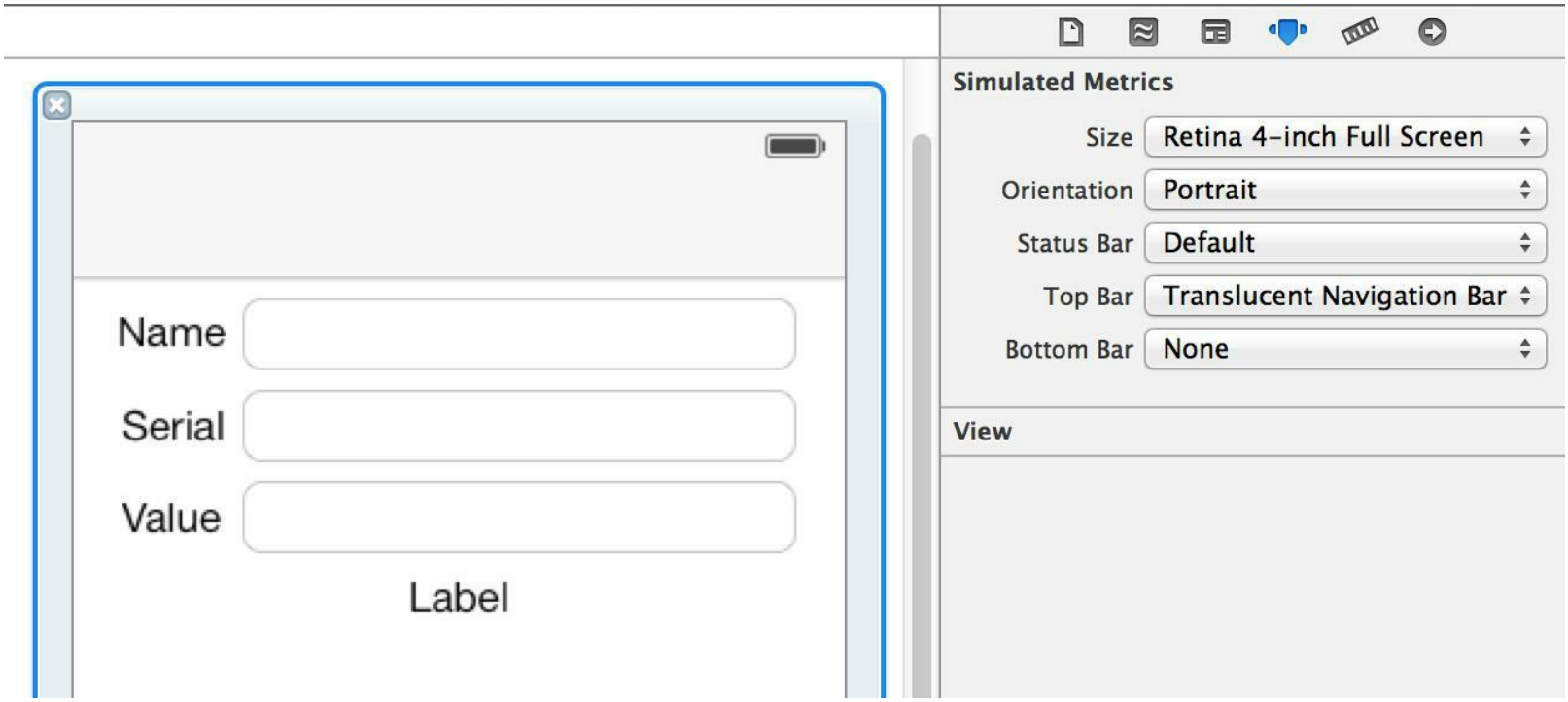


图10-10 Simulated Metrics

接下来将三个UITextField对象和位于底部的UILabel对象关联至BNRDetailView- Controller的相应插座变量。在BNRDetailViewController.xib中按住Control，将位于Name标签右侧的UITextField对象拖曳至BNRDetailViewController.m的类扩展中，如图10-11所示。

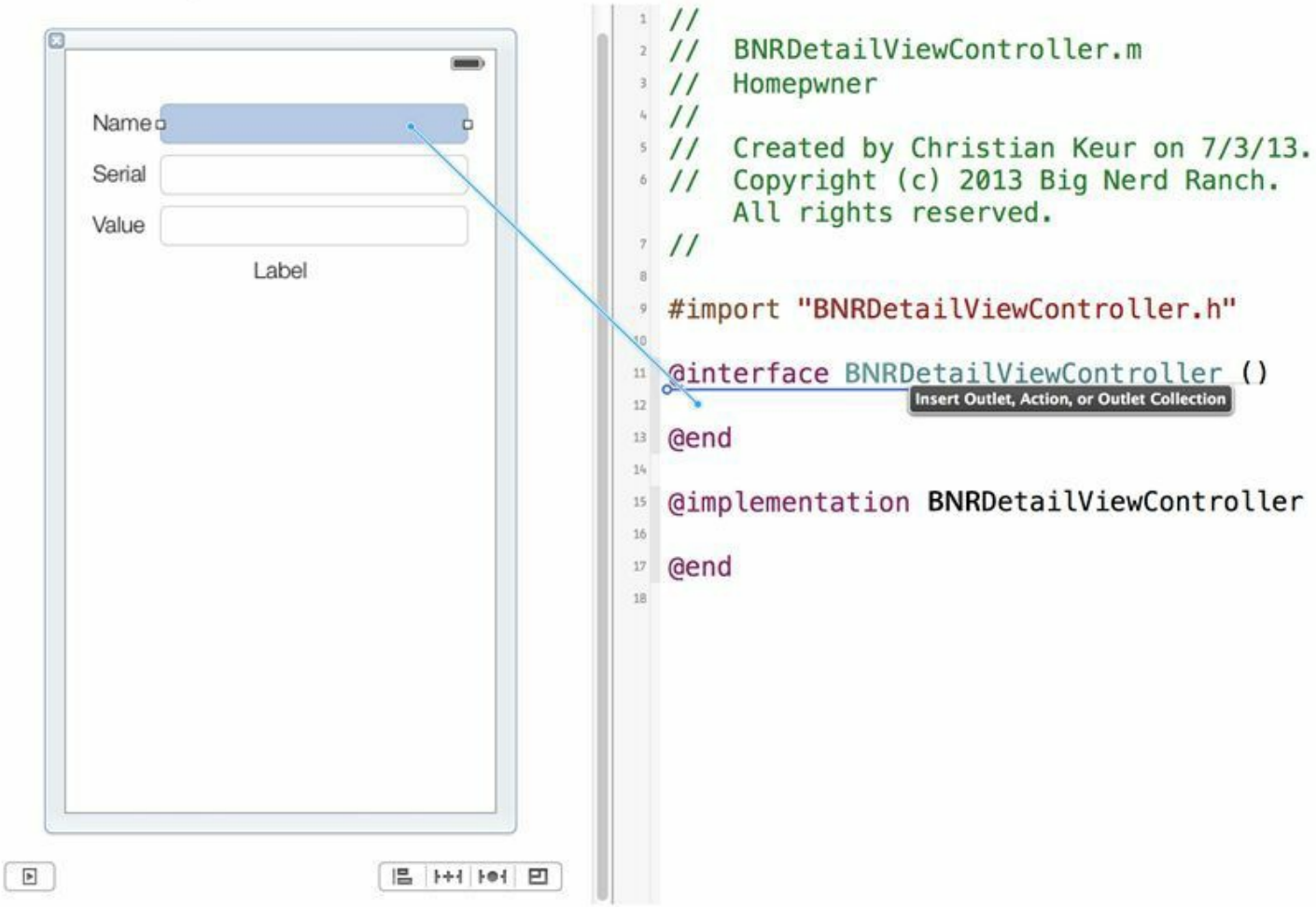


图10-11 从XIB拖曳至源代码

拖曳至类扩展中后松开鼠标按键，Xcode会弹出设置窗口。在Name文本框中输入nameField，选择Storage下拉菜单中的Weak，单击Connect按钮（见图10-12）。

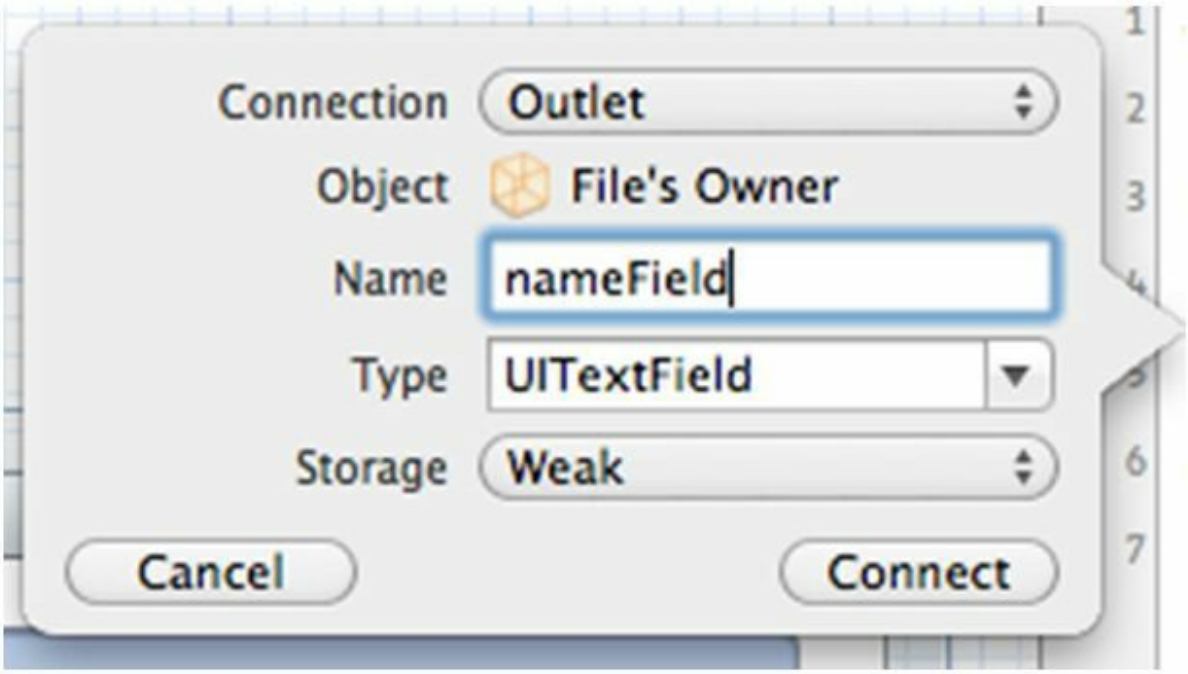


图10-12 自动生成插座变量并创建关联

上述拖曳操作会为BNRDetailViewController创建一个带IBOutlet前缀的属性，类型为UITextField，名称为nameField。因为nameField属性指向的不是XIB文件中的顶层对象，所以要将引用类型设置为Weak。

上述拖曳操作还会将XIB文件的File's Owner的插座变量nameField关联至相应的UITextField对象。要验证上述关联是否存在，可以按住Control并单击File's Owner，Xcode会显示关联面板并列出File's Owner的所有关联。将光标悬停在面板的nameField关联上，Xcode会高亮显示位于该关联另一端的UITextField对象。通过上述快捷途径，只需一步就能完成插座变量的创建和关联。

重复上述步骤，创建其他三个插座变量，对象名称如图10-13所示。

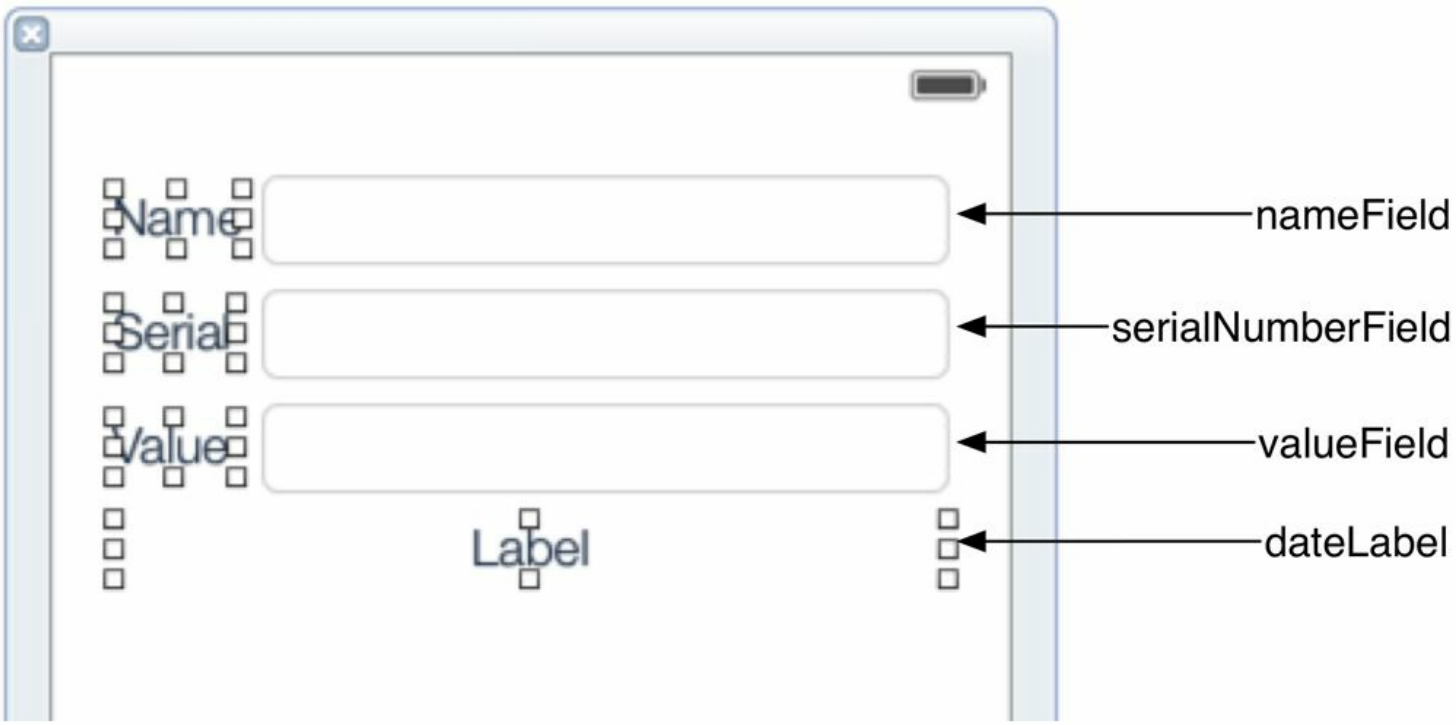


图10-13 关联图

创建完所有的关联后，BNRDetailViewController.h的代码如下：

```
#import “BNRDetailViewController.h”

@interface BNRDetailViewController（）

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

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

@property（weak, nonatomic）IBOutlet UITextField *valueField;
```

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

@end

@implementation BNRDetailViewController

@end
```

如果完成后的BNRDetailViewController.h和本书列出的代码不同，那么很有可能是因为没有正确地创建关联，可以通过以下三步进行修正：首先，重复之前介绍过的拖曳过程，再次创建关联，直到类扩展中的代码和上面列出的代码相同。然后，删除之前由Xcode创建的错误代码（例如方法声明和实例变量声明）。最后，检查XIB文件是否包含错误的关联。打开BNRDetailViewController.xib，按住Control并单击File's Owner，Xcode会显示关联面板。如果某个关联有问题，Xcode会在该关联的最右端显示一个感叹号图标。找出有问题的关联，然后通过关联左边的x按钮来删除已经建立的关联。

设置XIB文件时，要确保其中的关联都是正确的，这点非常重要。产生错误关联的常见原因为：读者修改了某个插座变量的变量名，但是没有更新XIB文件中的相应关联；或者读者彻底删除了某个插座变量，但是没有删除XIB文件中的相应关联。无论导致错误关联的原因是什么，当应用载入包含错误关联的XIB文件时，都有可能崩溃。

下面要在BNRDetailViewController.xib中创建剩余的关联。针对XIB文件中的每一个UITextField对象，将其delegate属性关联至File's Owner（按住Control，从UITextField对象拖曳至File's Owner，然后选择弹出菜单中的delegate）。

Homepwner项目已经初具规模，包含了较多的源代码文件，读者可能经常要在编辑器区域中切换显示这些文件。使用Xcode的标签（tab）功能，可以快速地在多个常用文件之间进行切换。双击项目导航面板中的某个文件，Xcode会在一个新的标签中打开该文件。此外，也可以使用快捷键Command-T来打开一个空白标签。使用快捷键Command-Shift-}可以选中下一个标签，使用快捷键Command-Shift-{可以选中上一个标签。打开Xcode的Preferences（偏好设置）窗口，选择Key Bindings标签，可以看到用于管理项目的其他快捷键。

10.3 UINavigationController的导航功能

前面已经为Homepwner加入了一个UINavigationController对象，并创建了BNRItemsViewController和BNRDetailViewController这两个UIViewController子类。下面将这些对象组合在一起以实现其功能：当用户点击BNRItemsViewController对象的表格视图中的某一行时，Homepwner会创建一个BNRDetailViewController对象，然后将该对象的视图推入窗口，显示当前选中的BNRItem对象的各项属性。

将视图控制器压入栈

要完成上述任务，就必须创建BNRDetailViewController对象。问题是该在何处创建该对象？本章之前的应用都会在application:didFinishLaunchingWithOptions:中创建所有的视图控制器。例如，第6章在application:didFinishLaunchingWithOptions:中创建了两个视图控制器并将它们加入了UITabBarController对象的viewControllers数组。

但是，在使用UINavigationController时，不能简单地将所有可能用到的视图控制器都压入UINavigationController对象的栈。该对象的viewControllers数组是动态的：一开始只有一个根视图控制器，应用需要根据情况推入新的视图控制器。因此，Homepwner需要某个对象（除UINavigationController对象外）来负责创建BNRDetailViewController对象，并将新创建的对象压入UINavigationController对象的栈。

这个负责创建BNRDetailViewController对象的对象需要满足两个条件：首先，该对象要知道在什么时候将BNRDetailViewController对象压入栈。其次，该对象需要拥有一个指向UINavigationController对象的指针，以便向UINavigationController对象发送pushViewController:animated:消息。

BNRItemsViewController对象满足上述两个条件。首先，因为该对象是UITableView对象的委托对象，所以，当用户点击UITableView对象的某个表格行时，BNRItemsViewController对象会收到tableView:didSelectRowAtIndexPath:消息。其次，凡是加入了某个UINavigationController对象的栈的视图控制器，都可以向自己发送navigationController消息，以得到指向该对象的指针。因为应用会将BNRItemsViewController对象设置为UINavigationController对象的根视图控制器，所以BNRItemsViewController对象会一直留在UINavigationController对象的栈中，从而使BNRItemsViewController对象总能得到指向相应UINavigationController对象的指针。

因此，应该由BNRItemsViewController对象负责创建BNRDetailViewController对象并将其加入UINavigationController对象的栈。在BNRItemsViewController.h顶部导入BNRDetailViewController的头文件。

```
#import “BNRDetailViewController.h”

@interface BNRItemsViewController : UITableViewController
```

当用户点击UITableView对象中的某个表格行时，该对象会向其委托对象发送

tableView:didSelectRowAtIndexPath:消息并传入选中的行的索引信息。在BNRItemsViewController.m中实现tableView:didSelectRowAtIndexPath:，创建BNRDetailViewController对象，然后将新创建的对象压入UINavigationController对象的栈，代码如下：

```
@implementation BNRItemsViewController

- (void)tableView:(UITableView *)tableView

didSelectRowAtIndexPath:(NSIndexPath *)indexPath

{

BNRDetailViewController *detailViewController =

[[BNRDetailViewController alloc] init];

// 将新创建的BNRDetailViewController对象压入UINavigationController对象的栈

[self.navigationController pushViewController:detailViewController

animated:YES];

}
```

构建并运行应用，添加一新行并选中之。Homepwner应该会以动画的形式，将BNRDetailViewController对象的视图从屏幕右侧推入。此外，位于窗口顶部的UINavigationBar对象会在其左端显示一个标题为“Back(返回)”的按钮。点击这个按钮，UINavigationController对象会弹出位于栈顶的BNRDetailViewController对象，退回至BNRItemsViewController对象。

因为UINavigationController对象的栈是一个数组，会拥有其包含的所有视图控制器，所以当应用执行完tableView:didSelectRowAtIndexPath:方法后，该对象将成为新创建的BNRDetailViewController对象的唯一拥有者。一旦UINavigationController对象将某个BNRDetailViewController对象弹出栈，弹出的这个对象就会立刻被释放。当用户再次选中某个表格行时，应用会创建新的BNRDetailViewController对象。

使用UINavigationController对象时，经常会由当前处于栈顶的视图控制器来负责压入另一个视图控制器，这是常见的使用模式。也就是说，UINavigationController对象的根视图控制器会负责创建并压入下一个视图控制器，而下一个视图控制器会负责创建并压入再下一个视图控制器，依此类推。有些应用的视图控制器可以根据用户的输入创建并压入不同类型的视图控制器。以iOS自带的照片(Photos)应用为例，照片应用会根据用户选中的媒体类型将视频视图控制器或图片视图控制器压入UINavigationController对象的栈。

只能用于iPad的UISplitViewController类，使用的是另一种模式。iPad拥有更大的屏幕，可以同时显示垂直界面中的两个视图控制器，从而不需要将后者压入栈。第22章会对UISplitViewController作更多的介绍。

视图控制器之间的数据传递

下面为UITextField对象设置显示内容。为此，需要通过某种途径将选中的BNRItem对象从BNRItemsViewController对象传入BNRDetailViewController对象。

首先为BNRDetailViewController添加一个属性，用来保存指定的BNRItem对象。当用户点击UITableView对象中的某个表格行时，BNRItemsViewController对象应该将选中的BNRItem对象传给即将被压入栈的BNRDetailViewController对象。得到BNRItem对象后，BNRDetailViewController对象就可以针对相应的BNRItem属性设置所有的UITextField对象。当用户修改了某个UITextField对象后，BNRDetailViewController对象也会将新的值赋给相应的BNRItem对象属性。

在BNRDetailViewController.h中，先在文件顶部用@class指令来前置声明BNRItem，然后为BNRDetailViewController类添加一个属性，代码如下：

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

@class BNRItem;

@interface BNRDetailViewController : UIViewController

@property (nonatomic, strong) BNRItem *item;

@end
```

在BNRDetailViewController.m中，导入BNRItem的头文件：

```
#import "BNRItem.h"
```

当应用要显示BNRDetailViewController对象的视图时，该对象需要根据item的各个属性设置其视图的子视图。在BNRDetailViewController.m中覆盖viewWillAppear:，将BNRItem对象的各个属性赋给相应的UITextField对象，代码如下：

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

{

[super viewWillAppear:animated];

BNRItem *item = self.item;

self.nameField.text = item.itemName;

self.serialNumberField.text = item.serialNumber;
```

```
self.valueField.text =
[NSString stringWithFormat:@"%d", item.valueInDollars];
// 创建NSDateFormatter对象，用于将NSDate对象转换成简单的日期字符串
static NSDateFormatter *dateFormatter = nil;
if（！dateFormatter）{
dateFormatter = [[NSDateFormatter alloc] init];
dateFormatter.dateStyle = NSDateFormatterMediumStyle;
dateFormatter.timeStyle = NSDateFormatterNoStyle;
}
// 将转换后得到的日期字符串设置为dateLabel的标题
self.dateLabel.text = [dateFormatter stringFromDate:item.dateCreated];
}
```

接下来在BNRItemsViewController.m的tableView:didSelectRowAtIndexPath:方法中添加以下代码，以便在BNRDetailViewController对象收到viewWillAppear:消息前将其item属性设置为相应的BNRItem对象。

```
-（void）tableView:（UITableView *）tableView
didSelectRowAtIndexPath:（NSIndexPath *）indexPath
{
BNRDetailViewController *detailViewController =
[[BNRDetailViewController alloc] init];
NSArray *items = [[BNRItemStore sharedStore] allItems];
BNRItem *selectedItem = items[indexPath.row];
// 将选中的BNRItem对象赋给DetailViewController对象
detailViewController.item = selectedItem;
[self.navigationController pushViewController:detailViewController
```

animated:YES];

}

当要在多个视图控制器之间传递数据时，很多刚接触iOS开发的初学者可能会无从下手。Homepwner使用了一种简单高效的解决方案：由根视图控制器保存所有的数据，然后将数据的子集传给下一个视图控制器。

构建并运行应用，添加一新的表格行并选中。Homepwner会显示BNRDetailViewController对象的视图，其中包含选中的BNRItem对象的详细信息。虽然用户可以编辑这些信息，但是当屏幕返回至BNRItemsViewController对象后，UITableView对象不会根据用户做出的修改来更新显示内容。要解决这个问题，需要编写代码，根据用户的输入来修改BNRItem对象的相应属性。

视图的显示和消失

当UINavigationController对象切换视图时，其包含的两个UIViewController对象会分别收到viewWillDisappear:消息和viewWillAppear:消息。即将出栈的UIViewController对象会收到viewWillDisappear:消息，即将入栈的UIViewController对象会收到viewWillAppear:消息。

当某个BNRDetailViewController对象退出栈时，应该将各个UITextField对象的值赋给BNRItem对象的相应属性。覆盖viewWillDisappear:和viewWillAppear:时，必须先调用父类的实现，以完成必要的工作。在BNRDetailViewController.m中实现viewWillDisappear:，代码如下：

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

{

[super viewWillDisappear:animated];

// 取消当前的第一响应对象

[self.view endEditing:YES];

// 将修改“保存”至BNRItem对象

BNRItem *item = self.item;

item.itemName = self.nameField.text;

item.serialNumber = self.serialNumberField.text;

item.valueInDollars = [self.valueField.text intValue];

}
```

这段代码使用了UIView的endEditing:方法。当某个视图收到endEditing:消息时，如果该视图（或者其下的任何子视图）是当前的第一响应对象，就会取消自己的第一响应对象状态，而且虚拟键盘也会关闭（传入的参数代表是否需要强制取消第一响应对象状态。某些第一响应对象可能会拒绝退出状态，传入YES可以强制其退出）。

更新代码后，当用户点击UINavigationBar对象上的Back（返回）按钮时，BNRDetailViewController对象就会更新相应的BNRItem对象。当BNRItemsView- Controller对象的视图再次出现在屏幕上时，它就会收到viewWillAppear:消息。这时应该刷新UITableView对象，使用户能够立刻看到更新后的数据。在BNRItemsViewController.m中覆盖viewWillAppear:，代码如下：

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

{

[super viewWillAppear:animated];

[self.tableView reloadData];

}
```

构建并运行应用。现在应该可以在BNRItemsViewController对象和BNRDetail-ViewController对象之间来回切换并修改数据，UITableView对象会及时刷新，以显示更新后的BNRItem对象。

10.4 UINavigationBar

Homepwner的UINavigationBar对象目前没有显示任何内容。下面先让UINavigationBar对象针对栈顶的UIViewController对象显示一个具有描述性的标题。

UIViewController对象有一个名为navigationItem的属性，类型为UINavigationItem。和UINavigationBar不同，UINavigationItem不是UIView的子类，不能在屏幕上显示。UINavigationItem对象的作用是为UINavigationBar对象提供绘图所需的内容。当某个UIViewController对象成为UINavigationController的栈顶对象时，UINavigationBar对象就会访问该UIViewController对象的navigationItem，获取和界面显示有关的内容，如图10-14所示。

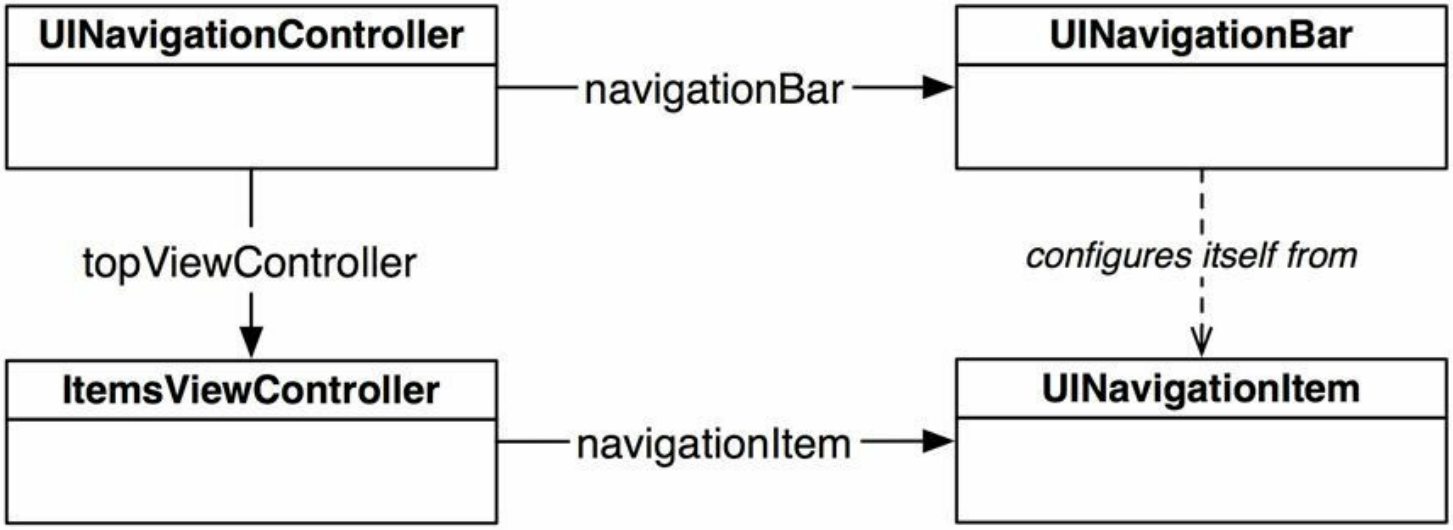


图10-14 UINavigationItem

UINavigationItem对象默认是“空”的。要让UINavigationBar对象能够显示最基本的信息，可以为UINavigationItem对象设置一个简单的标题（title属性）。当应用将某个UIViewController对象移至UINavigationController对象的栈顶时，UINavigationBar对象就会访问UIViewController对象的navigationItem属性，查看相应的title属性是否指向有效的NSString对象。如果是，就会在UINavigationBar对象的正中显示该字符串（见图10-15）。

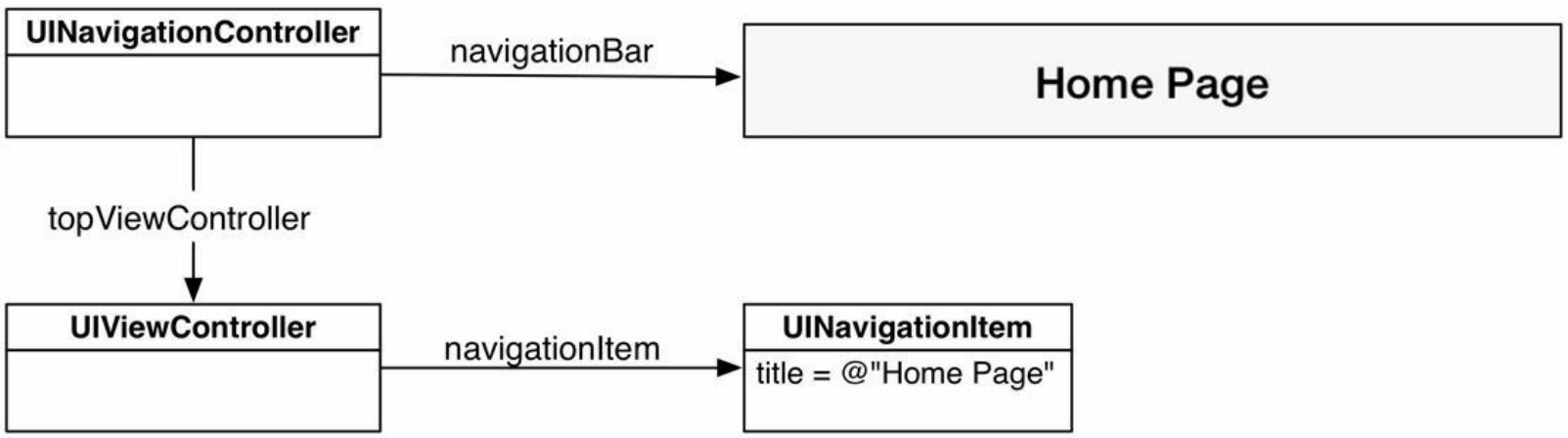


图10-15 带标题的UINavigationItem对象

修改BNRItemsViewController.m中的init方法，将navigationItem的title属性设置为字符串Homepwner，代码如下：

```
- (instancetype)init

{

self = [super initWithStyle:UITableViewStylePlain];

if (self) {

UINavigationItem *navItem = self.navigationItem;

navItem.title = @“Homepwner”;

}

return self;

}
```

构建并运行应用。UINavigationBar对象会显示标题Homepwner。添加一个新的表格行并选中，UINavigationBar对象的标题会消失，所以下面要为BNRDetailViewController对象也设置一个“导航栏标题”，使用的字符串是选中的BNRItem对象的name属性。和BNRItemsViewController不同的是，不能在BNRDetailViewController的init方法中设置标题，因为这时的item属性还没有赋值，是nil。

正确的做法是在设置BNRDetailViewController对象的item属性时，设置相应的导航栏标题。在BNRDetailViewController.m中实现setItem:，替换掉编译器为item属性合成的存方法，代码如下：

```
- (void)setItem:(BNRItem *)item

{

_item = item;

self.navigationItem.title = _item.itemName;

}
```

构建并运行应用，添加一个新的表格行并选中。UINavigationBar对象会显示选中的BNRItem对象的name属性。

UINavigationItem对象除了可以设置标题字符串(title属性)外，还可以设置若干其他的界面

属性，如图10-16所示。这些可以自定义的属性包括：leftBarButtonItem、rightBarButtonItem和titleView。其中leftBarButtonItem和rightBarButtonItem都是指向UIBarButtonItem对象的指针，该对象包含显示某种特殊按钮所需的信息，这种按钮只能在UINavigationBar对象或UIToolbar对象中使用。

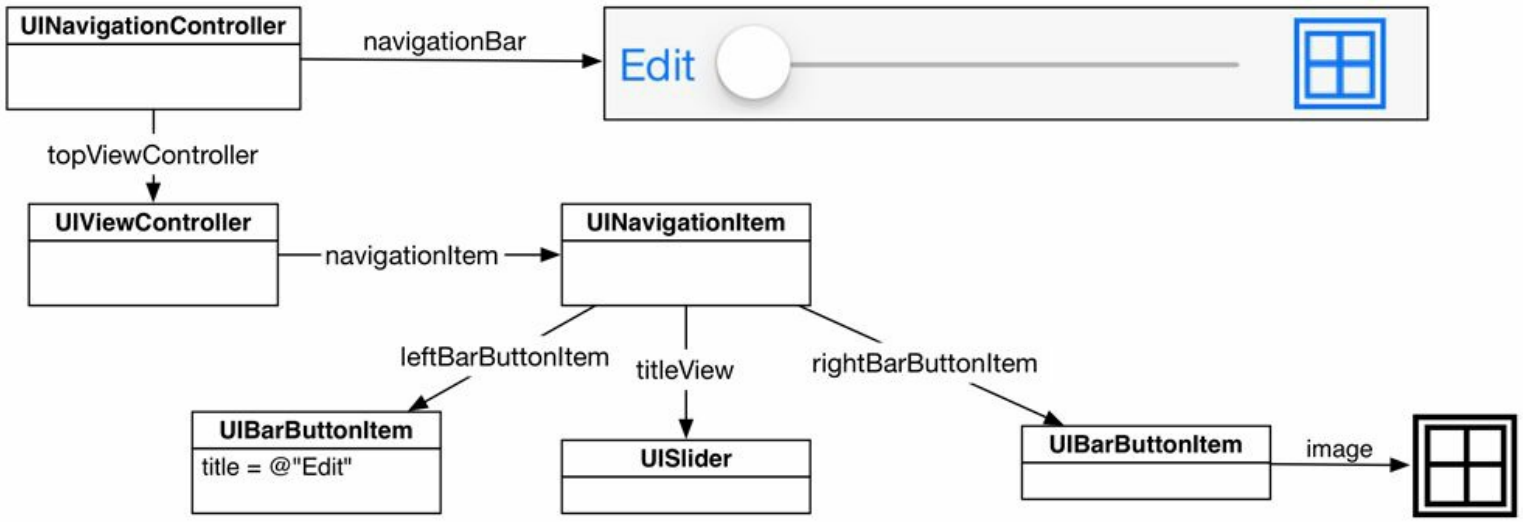


图10-16 设置了各种界面属性的UINavigationItem对象

类似于UINavigationItem，UIBarButtonItem也不是UIView的子类。UINavigationItem对象封装了一些设置信息，以便UINavigationBar在运行时可以正确显示该对象。同样，UIBarButtonItem对象封装了关于如何在UINavigationBar显示单个按钮的信息，UINavigationBar对象会根据UIBarButtonItem对象中的信息显示相应的按钮。（UIToolbar与UINavigationBar一样，也是通过UIBarButtonItem对象来显示按钮的。）

除了leftBarButtonItem和rightBarButtonItem，UIBarButtonItem对象的第三个界面属性是titleView。UINavigationBar对象包含两种标题显示模式。第一种模式是前面介绍过的：显示一个简单的字符串。第二种模式是在UINavigationBar对象正中显示一个视图。两种模式不能共存。如果需要在UINavigationBar对象正中显示复杂的界面，就可以先创建拥有自定义视图（例如UIButton对象、UISlider对象、UIImageView对象甚至MKMapView对象）的视图控制器，然后为该对象的titleView赋值，并指向这个自定义视图。以图10-16中的UINavigationItem对象为例，该对象的titleView属性指向的就是一个自定义视图。通常情况下，使用第一种模式就可以了，本章使用的也是这种模式。

下面要让UINavigationBar对象显示一个按钮：当BNRItemsViewController对象位于栈顶时，在UINavigationBar对象的右端显示一个按钮。当用户点击这个按钮时，为UITableView对象添加一个新的表格行。

为了能在用户点击UINavigationBar对象中的按钮时触发指定的方法，必须为相应的UIBarButtonItem对象设置目标-动作对，其工作机制类似UIControl的目标-动作对。通过某个UIBarButtonItem对象的初始化方法和相应的存方法，都可以为该对象设置目标-动作对。

在BNRItemsViewController.m的init方法中，创建UIBarButtonItem对象并设置目标对象和动作方法，代码如下：

```
- (instancetype)init
{
self = [super initWithStyle:UITableViewStylePlain];
if (self) {
UINavigationItem *navItem = self.navigationItem;
navItem.title = @"Homepwner";
// 创建新的UIBarButtonItem对象
// 将其目标对象设置为当前对象，将其动作方法设置为addNewItem:
UIBarButtonItem *bbi = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
target:self
action:@selector(addNewItem:)];
// 为UINavigationItem对象的rightBarButtonItem属性赋值，
// 指向新创建的UIBarButtonItem对象
navItem.rightBarButtonItem = bbi;
}
return self;
}
```

设置动作方法时，需要传入的参数类型是SEL。前文已经介绍过，SEL是指向选择器的指针，而选择器代表方法的整个方法名（包括所有的冒号）。@selector()不关心方法的返回类型、参数类型及参数名。

构建并运行应用。点击+按钮，UITableView对象会添加一个新的表格行（除了本节使用的initWithBarButtonSystemItem:target:action:，UIBarButtonItem还有多个其他初始化方法。详细内容请读者自行查阅文档）。

下面再为UINavigationBar对象添加一个按钮，用于替换掉表头视图中的Edit按钮。编辑BNRItemsViewController.m中的init方法，代码如下：

```
- (instancetype)init
```

```
{
self = [super initWithStyle:UITableViewStylePlain];
if (self) {
UINavigationItem *navItem = self.navigationItem;
navItem.title = @"Homepwner";
// 创建新的UIBarButtonItem对象
// 将其目标对象设置为当前对象，将其动作方法设置为addNewItem:
UIBarButtonItem *bbi = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
target:self
action:@selector(addNewItem:)];
// 为UINavigationItem对象的rightBarButtonItem属性赋值，
// 指向新创建的UIBarButtonItem对象
navItem.rightBarButtonItem = bbi;
navItem.leftBarButtonItem = self.editButtonItem;
}
return self;
}
```

只需编写一行代码，向BNRItemsViewController对象发送editButtonItem消息，就能得到可以加入UINavigationBar对象的Edit(编辑)按钮。构建并运行应用，点击"编辑"按钮，UITableView对象会进入编辑模式。UIViewController对象有一个名为editButtonItem的属性，当该对象收到editButtonItem消息后，如果editButtonItem属性的值是nil，就会创建一个标题为"编辑"的UIBarButtonItem对象。此外，editButtonItem方法所返回的UIBarButtonItem对象默认已经设置好了目标-动作对。当用户点击对应的按钮时，包含该对象的UIViewController对象就会收到setEditing:animated:消息。

为Homepwner的UINavigationBar对象添加"＋"按钮和"编辑"按钮后，可以删除之前加入的那些和表头视图有关的代码。删除BNRItemsViewController.m中的以下代码：

```
- (UIView *)headerView
```

~~{~~

~~if (! _headerView) {~~

~~[[NSBundle mainBundle] loadNibNamed:@"HeaderView"~~

~~owner:self options:nil];~~

~~}~~

~~return _headerView;~~

~~}~~

~~- (IBAction)toggleEditingMode:(id)sender~~

~~{~~

~~if (self.isEditing) {~~

~~[sender setTitle:@"Edit" forState:UIControlStateNormal];~~

~~[self setEditing:NO animated:YES];~~

~~} else {~~

~~[sender setTitle:@"Done" forState:UIControlStateNormal];~~

~~[self setEditing:YES animated:YES];~~

~~}~~

~~}~~

最后请读者删除不再需要使用的headerView属性和HeaderView.xib文件。

构建并运行应用。Homepwner不会再显示表头视图，取而代之的是带两个按钮的UINavigationBar对象（见图10-17）。

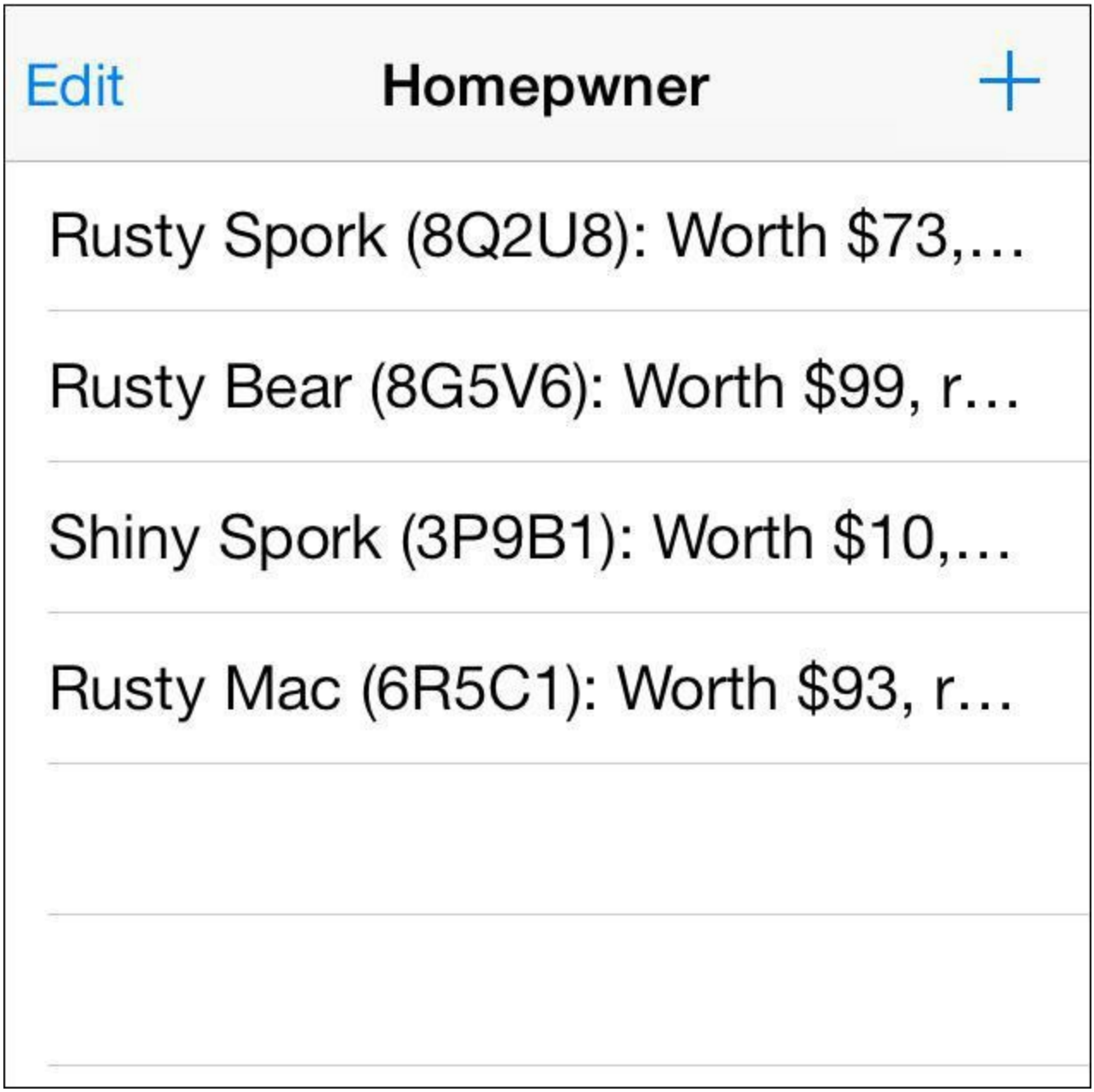


图10-17 使用UINavigationBar的Homepwner

10.5 初级练习：显示数字键盘

在BNRDetailViewController对象的视图中，其中一个UITextField对象负责显示BNRItem对象的valueInDollars属性。当该对象成为第一响应对象时，屏幕会显示一个类型为QWERTY的键盘。因为需要输入的是数字，所以最好将键盘类型改为数字键盘（Number Pad）。提示：可以通过属性检视面板直接在XIB文件中设置UITextField对象的键盘类型。

10.6 中级练习：关闭数字键盘

完成初级练习后，读者可能会发现数字键盘没有换行（return）键。为Homepwner加入某种界面元素，可以让用户关闭数字键盘。

10.7 高级练习：压入更多视图控制器

目前BNRItem的dateCreated属性是只读的，无法从对象外部进行修改。为了实现修改dateCreated属性的功能，可以为BNRDetailViewController对象的视图添加一个标题为Change Date的UIButton对象，并将该对象摆放在dateLabel的下方。当用户点击该按钮时，需要将一个新的视图控制器压入UINavigationController对象的栈。这个新加入的视图控制器的视图要包含一个UIDatePicker对象，可以让用户修改当前选中的BNRItem对象的dateCreated属性。

第11章 相机

本章要为Homepwner添加照片功能（见图11-1）。具体任务是显示一个UIImagePickerController对象，使用户能够为BNRItem对象拍照并保存。拍摄的照片会和相应的BNRItem对象建立关联，当用户进入某个BNRItem对象的详细视图时，可以看见之前为该对象拍摄的照片。

Homepwner **Rusty Spork**

Name: Coffee Cup

Serial: 8Q2U8

Value: 7

Dec 9, 2013

图11-1 具有照片功能的Homepwner

照片的文件可能很大，最好与BNRItem对象的其他数据分开保存。本章将创建第二个用于存储数据的类BNRImageStore，负责保存BNRItem对象的照片。BNRImageStore可以按需获取并缓存照片，还可以在设备内存过低时清空缓存中的照片。

11.1 通过UIImageView对象显示照片

首先要将照片赋给BNRDetailViewController对象，才能在该对象的视图中显示。要在视图上显示照片，一种简单的途径是使用UIImageView对象。

打开Homepwner.xcodeproj，点击项目导航面板中的BNRDetailViewController. xib。在对象库面板中找到UIImageView对象，然后拖曳至XIB文件中的顶层视图，放在UILabel对象的下方。接下来调整UIImageView对象的大小，在水平方向基本充满屏幕，但是底部要留有一定的空间，用于之后添加UIToolbar对象（见图11-2）。

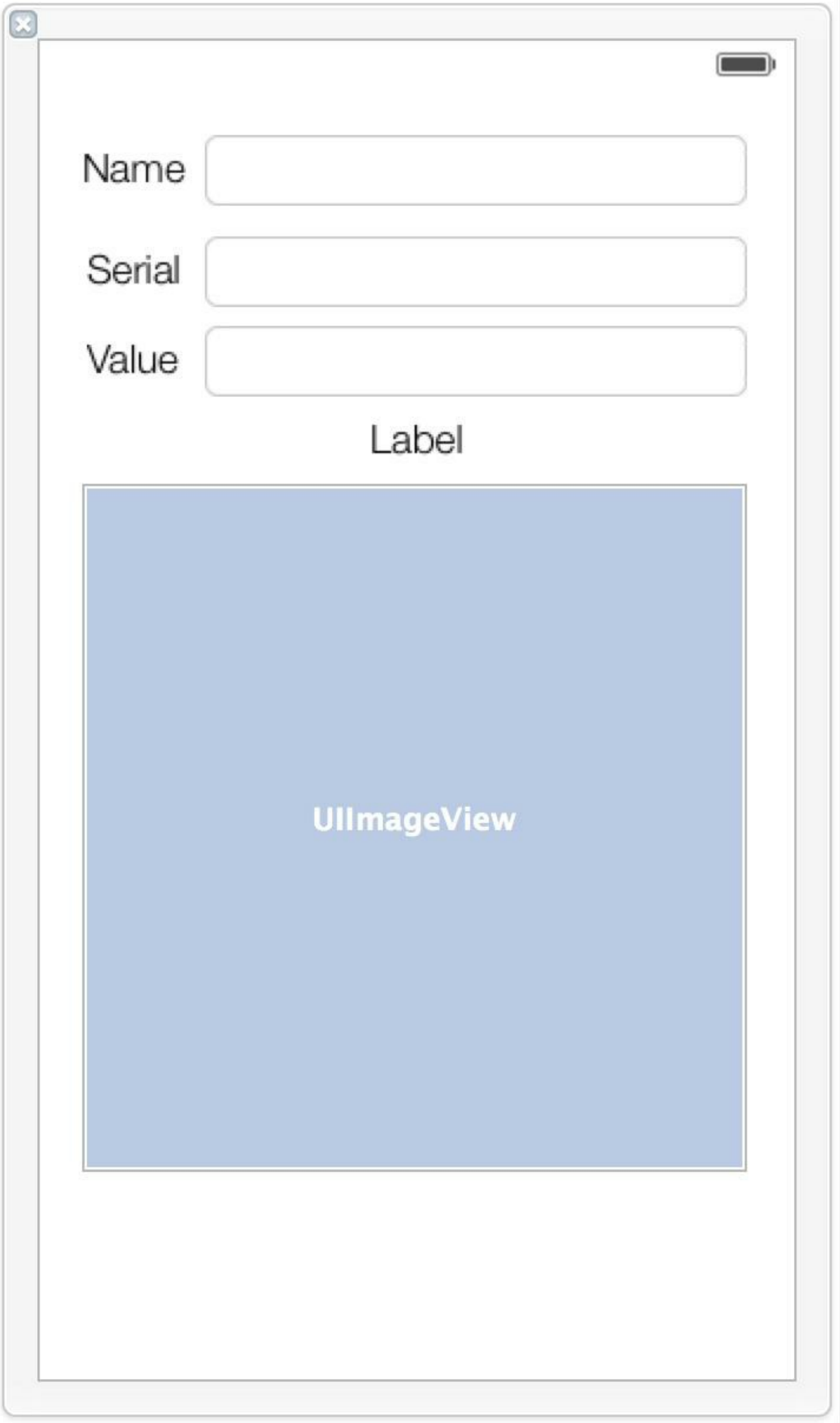

Name
Serial
Value
Label
UIImageView

图11-2 BNRDetailViewController对象视图中的UIImageView对象

UIImageView对象会根据其contentMode属性显示一张指定的图片。contentMode属性的作用是确定图片在frame内的显示位置和缩放模式。contentMode的默认值是UIViewContentModeScaleToFill。当contentMode的值是UIViewContentMode- ScaleToFill时，UIImageView对象会在显示图片时缩放图片的大小，使其能够填满整个视图空间，但是可能会改变图片的宽高比。如果使用其默认值，UIImageView对象为了能在正方形的区域中显示由相机拍摄的大尺寸照片，就要改变照片的宽高比。为了获得最佳显示效果，要修改UIImageView对象的contentMode，要求其根据宽高比缩小照片。

选中UIImageView对象并打开属性检视面板，找到标题为Mode的下拉列表并将其值改为Aspect Fit，如图11-3所示。在Aspect Fit模式下，UIImageView对象会在显示图片时按宽高比缩放图片，使其能够填满整个视图。

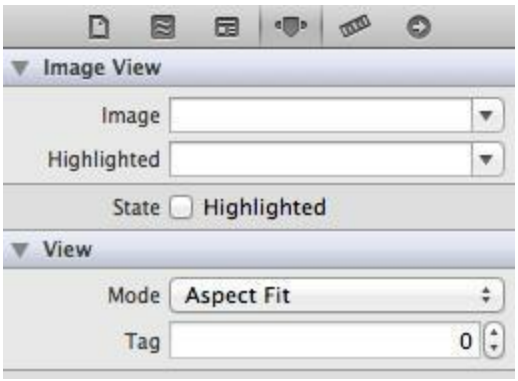


图11-3 将UIImageView对象的Mode改为Aspect Fit

在辅助编辑器中打开BNRDetailViewController.m（按住Option键并点击项目导航面板中的BNRDetailViewController.m）。按住Control键，将UIImageView对象拖曳至BNRDetailViewController.m的类扩展中。Xcode会弹出设置窗口，在Name中输入imageView，Storage选择Weak，点击Connect按钮。

完成上述操作后，BNRDetailViewController的类扩展应该如下所示：

```
@interface BNRDetailViewController ()

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

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

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

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

@property (weak, nonatomic) IBOutlet UIImageView *imageView;

@end
```

添加相机按钮

下面为Homepwner添加一个相机按钮，用于启动拍照过程。为此，需要先创建一个UIToolbar对象，然后将该对象放置在BNRDetailViewController对象的视图底部，最后将按钮放置在UIToolbar对象上（虽然可以将这个按钮直接放在UINavigationBar对象上，但是因为稍后还要在该对象上放置另外一个按钮，所以改用UIToolbar对象）。

在BNRDetailViewController.xib中，拖曳一个UIToolbar对象至顶层视图的底部。

UIToolbar对象的工作方式和UINavigationBar非常相似，同样可以加入UIBarButtonItem对象。区别是UINavigationBar只能在左右两端分别放置一个UIBarButtonItem对象，而UIToolbar对象可以有一组UIBarButtonItem对象。只要屏幕能够容纳，UIToolbar对象自身并没有限制可以存放的UIBarButtonItem对象个数。

加入XIB的UIToolbar对象默认自带一个UIBarButtonItem对象。选中该对象，打开属性检视面板并在Identifier下拉菜单中选择Camera，相应的按钮会显示一个相机图标（见图11-4）。

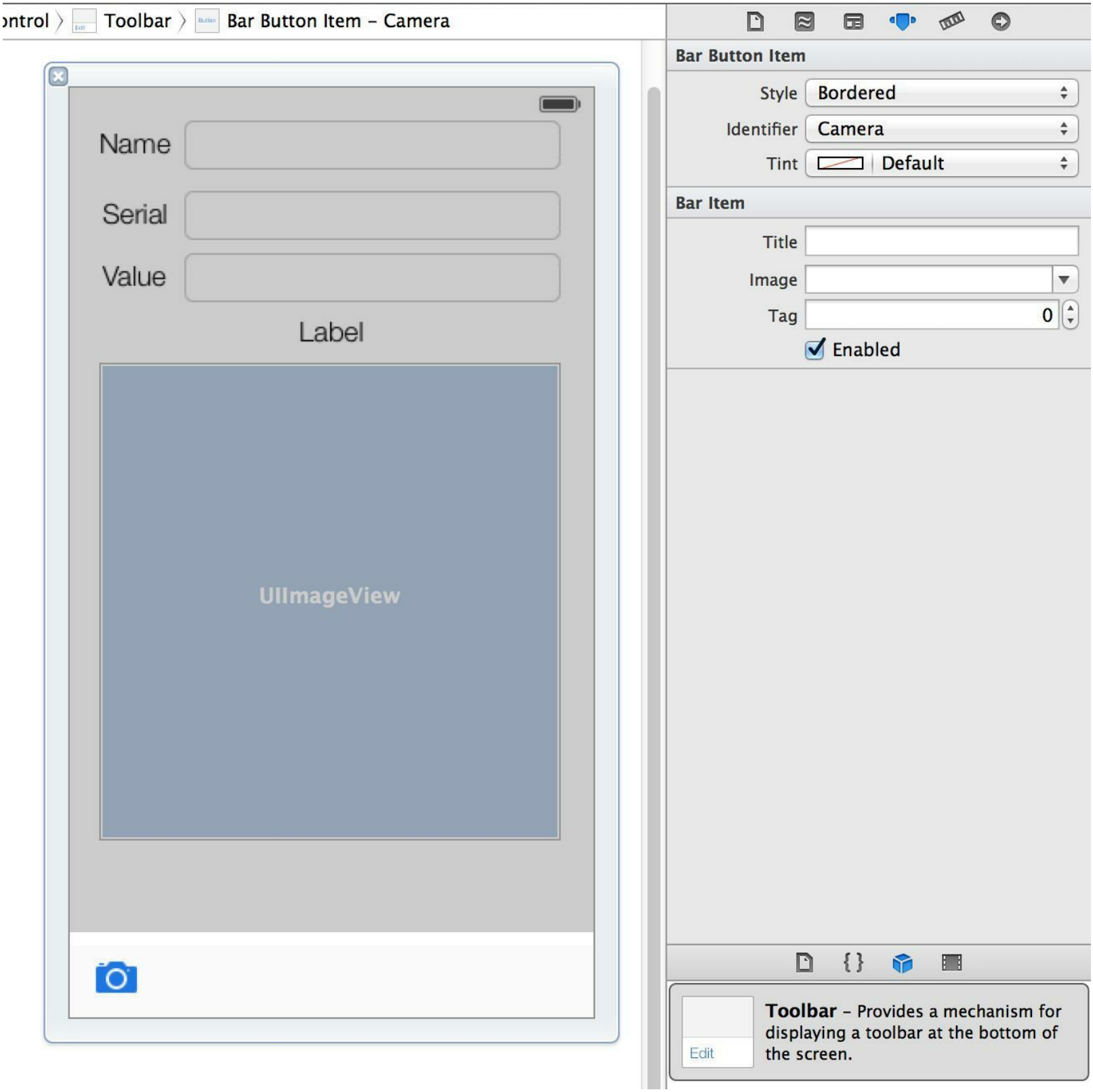


图11-4 带有一个UIBarButtonItem对象的UIToolbar

下面需要为相机按钮设置目标和动作。本章之前的例子都是通过以下两步来创建动作关联的：首先在代码中声明动作方法；然后在XIB文件中创建相应的关联。与插座变量类似，动作方法也可以通过一步直接创建并关联。

按住Option键并点击项目导航面板中的BNRDetailViewController.m，在辅助编辑器中打开该文件。

在BNRDetailViewController.xib中，选中相机按钮，按住Control键，将其拖曳至

BNRDetailViewController.m的方法实现区域（见图11-5）。

图11-5 通过XIB创建并关联动作方法

松开鼠标，Xcode会显示一个设置窗口。选择Connection下拉菜单中的Action，在Name输入框中输入takePicture:，点击Connect按钮（见图11-6）。

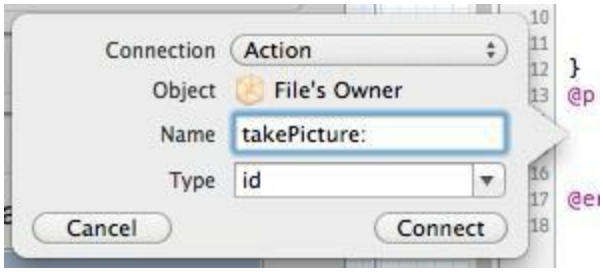


图11-6 创建动作方法并建立关联

完成上述步骤后，Xcode会在BNRDetailViewController.m中添加一个名为takePicture:的空方法，并将UIBarButtonItem对象的目标设置为BNRDetailViewController对象，动作方法设置为takePicture:，空的takePicture:代码如下：

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

{

}
```

此外，Xcode会在代码中提示动作方法的关联状态。注意takePicture:方法左侧的行号区域，

这里有一个圆圈图标(见图11-7)。如果这个圆圈是实心的，表示XIB文件已经包含针对该动作方法的关联；如果是空心的，则表示尚未创建关联。

图11-7 源代码编辑器所显示的关联状态

本章稍后还需要引用UIToolbar对象，下面为该对象设置插座变量。选中UIToolbar对象(注意不要选成了UIToolbar对象上的相机按钮)。然后按住Control键，将UIToolbar对象拖曳到BNRDetailViewController.m的类扩展中。在弹出窗口中，将插座变量命名为toolbar，Storage选择Weak，最后点击Connect按钮。

现在BNRDetailViewController.m类扩展中的代码应该如下所示：

```
@interface BNRDetailViewController ()

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

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

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

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

@property (weak, nonatomic) IBOutlet UIImageView *imageView;

@property (weak, nonatomic) IBOutlet UIToolbar *toolbar;

@end
```

和插座变量关联一样，也要保证XIB文件中动作方法的关联都是正确的。如果在BNRDetailViewController.xib中发现任何错误的动作方法关联(查看关联检视面板，找出带感叹号图标的关联)，就必须删除重建。

11.2 通过UIImagePickerController拍摄照片

下面在takePicture:中创建并显示UIImagePickerController对象。创建该对象时，必须为新创建的对象设置sourceType属性和delegate属性。

设置UIImagePickerController对象的源

设置sourceType属性时必须使用特定的常量，这些常量表示UIImagePicker- Controller对象获取照片的“源”。目前有以下三种可以使用的常量。

•UIImagePickerControllerSourceTypeCamera：用于用户拍摄一张新照片。

•UIImagePickerControllerSourceTypePhotoLibrary：用于显示界面，让用户选择相册，然后从选中的相册中选择一张照片。

•UIImagePickerControllerSourceTypeSavedPhotosAlbum：用于让用户从最近拍摄的照片里选择一张照片。

图11-8列出了三种常量的使用效果。

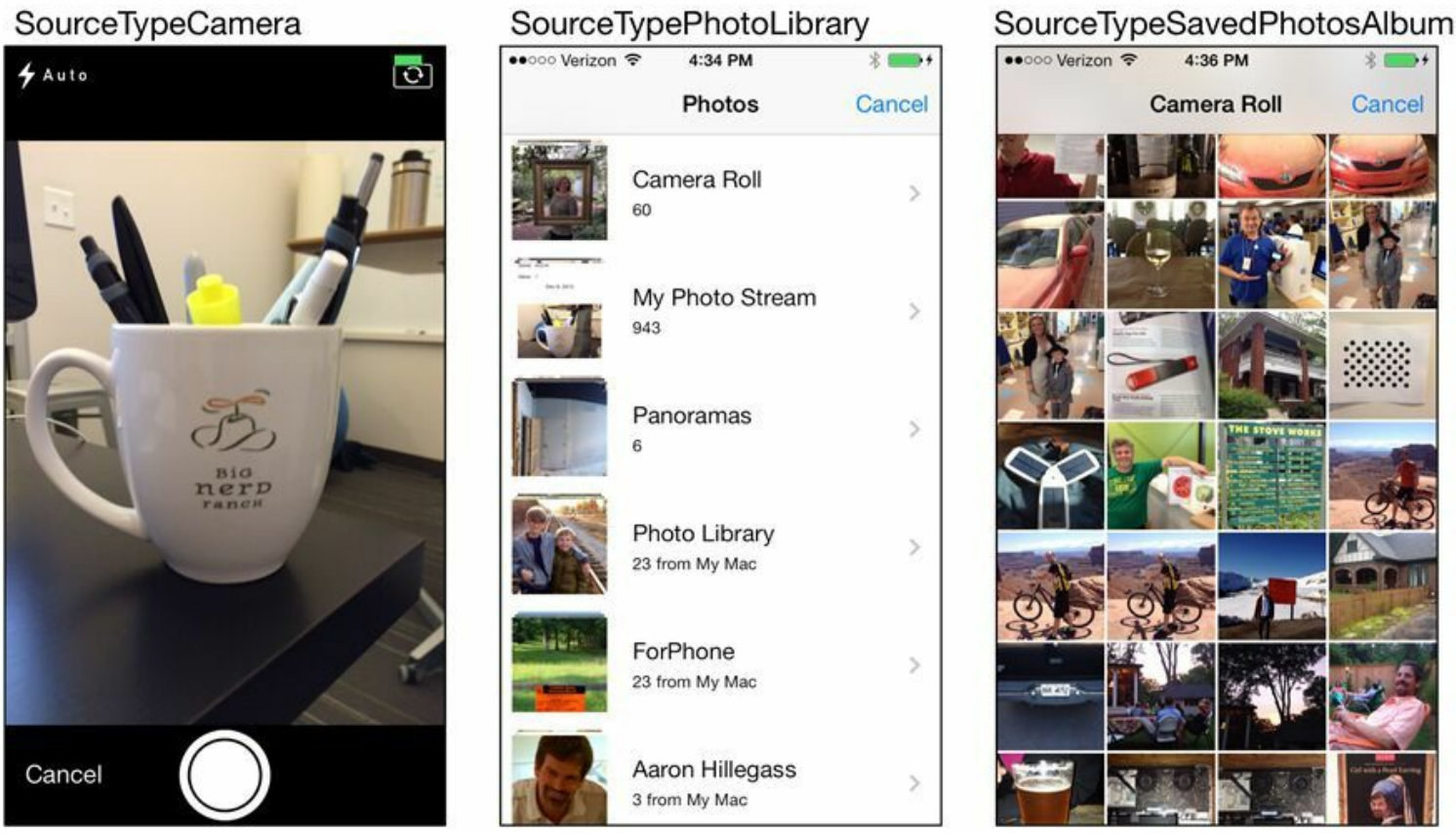


图11-8 三种常量的使用效果

对于没有相机的设备，第一种选取类型UIImagePickerControllerSourceTypeCamera是无效的。所以在使用第一种变量前，需要向UIImagePickerController类发送isSourceTypeAvailable:消息，检查设备是否支持相机。发送该消息时，需要传入待检查的选取类型常量。

```
+ (BOOL)isSourceTypeAvailable:(UIImagePickerControllerSourceType)sourceType;
```

isSourceTypeAvailable:会返回一个布尔类型的值，表示设备是否支持传入的选取类型。

下面创建一个UIImagePickerController对象，然后设置其sourceType属性。在BNRDetailViewController.m的takePicture:方法中添加以下代码：

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

{

UIImagePickerController *imagePicker =

[[UIImagePickerController alloc] init];

// 如果设备支持相机，就使用拍照模式

// 否则让用户从照片库中选择照片

if ([UIImagePickerController

isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera]) {

imagePicker.sourceType = UIImagePickerControllerSourceTypeCamera;

} else {

imagePicker.sourceType =

UIImagePickerControllerSourceTypePhotoLibrary;

}

}
```

设置UIImagePickerController对象的委托

除了sourceType属性，还需要为UIImagePickerController对象设置委托，也就是delegate属性。当用户从UIImagePickerController对象中选择了一张照片后，委托会收到imagePickerController:didFinishPickingMediaWithInfo:消息（如果用户取消了选择，则委托会收到imagePickerControllerDidCancel:消息）。

UIImagePickerController对象的委托通常应该设置为需要获取照片的对象，因此这里应该设置为BNRDetailViewController对象（需要在详细界面显示用户选择的照片）。在BNRDetailViewController.m的类扩展中，将BNRDetailViewController声明为遵守UINavigationControllerDelegate和UIImagePickerControllerDelegate协议（UIImagePickerController是UINavigationController的子类，所以UIImagePicker- Controller的委托也要遵守UINavigationControllerDelegate协议），代码如下：

```
@interface BNRDetailViewController ()

<UINavigationControllerDelegate, UIImagePickerControllerDelegate>
```

接下来在takePicture:中将UIImagePickerController对象的委托设置为BNRDetailViewController对象自身。

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

{

UIImagePickerController *imagePicker =

[[UIImagePickerController alloc] init];

// 如果设备支持相机，就使用拍照模式

// 否则让用户从照片库中选择照片

if ([UIImagePickerController

isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera]) {

imagePicker.sourceType = UIImagePickerControllerSourceTypeCamera;

} else {

imagePicker.sourceType = UIImagePickerControllerSourceTypePhotoLibrary;

}

imagePicker.delegate = self;

}
```

以模态的形式显示UIImagePickerController对象

为UIImagePickerController对象设置了源类型和委托之后，就可以在屏幕中显示该对象。和

之前显示的UIViewController子类对象不同，该对象必须以模态的(modal)形式显示。以模态的形式显示的视图控制器，其视图会占据整个屏幕，直到关闭。

要以模态的形式显示某个视图控制器，需要向窗口当前显示的UIViewController对象发送presentViewController:animated:completion:，并为第一个参数传入需要显示的视图控制器。同时，如果为第二个参数animated:传入YES，相应的视图控制器的视图会从屏幕底部滑入(第17章会深入介绍模态视图控制器与第三个参数)。

在BNRDetailViewController.m中的takePicture:方法结尾处添加以下代码，以模态形式显示UIImagePickerController对象。

```
imagePicker.delegate = self;

// 以模态的形式显示UIImagePickerController对象

[self presentViewController:imagePicker animated:YES completion:nil];

}
```

构建并运行应用。选中某个BNRItem对象，Homepwner会显示BNRDetailViewController对象的视图。按下UIToolbar对象上的相机按钮，Homepwner会显示UIImagePicker- Controller对象的界面(见图11-9)。读者可以拍摄一张照片(如果设备没有相机，则可以选择一张现有的照片)。


Auto
Big
nerd
ranch
Cancel

图11-9 UIImagePickerController对象的界面

（如果读者是针对模拟器构建并运行应用的，那么模拟器上的相册默认是没有图片的。解决方法是启动模拟器中的Safari浏览器，访问某个带图片的网页，按住某张图片不放直到出现动作菜单，然后选择存储图像（Save Image）。这样，Safari会将选中的图片存入模拟器的照片库。但是模拟器终究只是模拟器，如果UIImagePickerController对象无法显示之前保存的图片，那么更换图片再试几次。）

保存照片

选择了一张照片以后，UIImagePickerController对象就会自动关闭，返回BNRDetailViewController界面。但是，如何在代码中获取选择的照片呢？答案是之前介绍过的委托方法——imagePickerController:didFinishPickingMediaWithInfo:。

在BNRDetailViewController.m中实现imagePickerController:didFinishPicking-MediaWithInfo:，将选择的照片放入之前创建的UIImageView对象中，然后关闭UIImagePickerController对象，代码如下：

```
- (void)imagePickerController:(UIImagePickerController *)picker

didFinishPickingMediaWithInfo:(NSDictionary *)info

{

// 通过info字典获取选择的照片

UIImage *image = info[UIImagePickerControllerOriginalImage];

// 将照片放入UIImageView对象

self.imageView.image = image;

// 关闭UIImagePickerController对象

[self dismissViewControllerAnimated:YES completion:nil];

}
```

构建并运行应用。拍摄或选择一张照片，UIImagePickerController对象会自动关闭，返回BNRDetailViewController界面，这时界面会显示之前拍摄或选择的照片。

虽然目前Homepwner还可以正常工作，但假设之后用户添加了几百个BNRItem对象，而且每个BNRItem对象都带有一张高清照片，就必须采用新的方案存储和显示照片。为了避免一次性向内存中放入大量的照片，可以先将所有照片存储在磁盘中，只有当用户进入某个BNRItem对

象的详细界面时，才能将该对象的照片从磁盘读取到内存中。同时，如果用户浏览了多个BNRItem对象，应用由于内存中加载了大量BNRItem对象的照片而收到内存过低警告，就必须清空内存中不再使用的照片，释放占用的内存空间。下面就通过创建BNRImageStore类实现上述方案。

11.3 创建BNRImageStore

BNRImageStore对象将负责保存用户所拍的所有照片。创建一个名为BNRImageStore的NSObject子类，然后在BNRImageStore.h中添加以下代码：

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

@interface BNRImageStore : NSObject

+ (instancetype)sharedStore;

- (void)setImage:(UIImage *)image forKey:(NSString *)key;

-(UIImage *)imageForKey:(NSString *)key;

- (void)deleteImageForKey:(NSString *)key;

@end
```

接下来在BNRImageStore.m的类扩展中声明一个属性，用于存储照片。

```
@interface BNRImageStore ()

@property (nonatomic, strong) NSMutableDictionary *dictionary;

@end

@implementation BNRImageStore
```

和BNRItemStore类似，BNRImageStore也是单例。在BNRImageStore.m中加入以下代码，确保BNRImageStore的单例状态。

```
@implementation BNRImageStore

+ (instancetype)sharedStore

{

static BNRImageStore *sharedStore = nil;

if (!sharedStore) {

sharedStore = [[self alloc] initPrivate];

}

return sharedStore;
```

```
}
// 不允许直接调用init方法
- (instancetype)init
{
@throw [NSException exceptionWithName:@"Singleton"
reason:@"Use +[BNRImageStore sharedStore]"
userInfo:nil];
return nil;
}
// 私有初始化方法
- (instancetype)initPrivate
{
self = [super init];
if (self) {
_dictionary = [[NSMutableDictionary alloc] init];
}
return self;
}
```

然后实现其他三个新方法，代码如下：

```
- (void)setImage:(UIImage *)image forKey:(NSString *)key
{
[self.dictionary setObject:image forKey:key];
}
- (UIImage *)imageForKey:(NSString *)key
```

```
{
return [self.dictionary objectForKey:key];
}
- (void)deleteImageForKey:(NSString *)key
{
if (!key) {
return;
}
[self.dictionary removeObjectForKey:key];
}
```

11.4 NSDictionary

BNRImageStore的属性dictionary是一个指向NSMutableDictionary对象（字典对象）的指针。和数组对象类似，字典对象也是collection对象，也有不可修改的版本（NSDictionary）和可修改的版本（NSMutableDictionary）。数组对象与字典对象的差别是，数组对象包含一组有序的指向对象的指针，可以通过整数索引存取。例如，直接获取数组的第n个元素的代码如下：

```
// 将某个对象插入数组顶部

[someArray insertObject:someObject atIndex:0];

// 获取刚才插入的对象

someObject = [someArray objectAtIndex:0];
```

字典对象中的指针不是有序排列的，需要通过键（key）来存取指针，不能使用索引。键其实也是对象，而且最常用的是NSString对象。

```
// 将某个对象加入NSMutableDictioanry对象，对应的键是“MyKey”

[someDictionary setObject:someObject forKey:@“MyKey”];

// 取回刚才加入的对象

someObject = [someDictionary objectForKey:@“MyKey”];
```

字典对象是由键-值对（key-value pair）组成的。这里的键是某个不可修改的对象（通常是NSString对象），用来存取另一个与之对应的对象，也就是值。在其他语言中，这里的字典对象称为哈希图（hash map）或哈希表（hash table）。

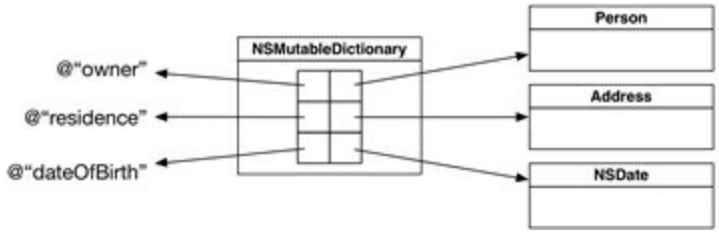


图11-10 NSDictionary对象图

NSDictionary非常有用，其中最常见的用法是可变数据结构（flexible data structures）和查询表（lookup tables）。

首先介绍可变数据结构。为了在代码中描述一个模型对象，常见的做法是创建一个NSObject的子类，然后添加模型对象的相关属性。例如，对于一个表示“人”的模型对象来说，可以创建一个名为Person的NSObject子类，然后添加姓名、年龄和其他需要的属性。类似地，NSDictionary也可以用来描述模型对象。还是以“人”为例，NSDictionary中可以针对姓名、年龄和其他需要的属性保存相应的键-值对。

使用NSDictionary与NSObject子类Person的区别是，Person要求事先明确定义好“人”的各项属性，并且之后无法添加新的属性，也无法删除或修改现有属性。相反，如果使用NSDictionary，“人”的数据就只是一系列键-值对，操作起来非常简单，例如，为某个人添加“地址”时，只需要为@“address”键设置表示地址的字符串就可以了。

当然，并不是所有的模型对象都可以通过NSDictionary来描述。大部分模型对象具有严格的定义和特定的数据处理方式，不适合采用简单的键-值对管理数据。相反，如果模型对象根据不同的配置选项具有不同的数据结构，就应该使用NSDictionary。例如UIImagePickerController的委托方法imagePickerController:didFinishPicking- MediaWithInfo:，其第二个参数就是一个通过NSDictionary描述的模型对象，根据UIImagePickerController的相关配置，该对象可能包含照片或视频（本章第11.12节会介绍如何使用UIImagePickerController录制视频），以及相关元数据信息。

再介绍NSDictionary的另一个常见用法：查询表。读者在刚开始学习编程时，可能会写出类似如下代码：

```
- (void)changeCharacterClass:(id)sender

{

NSString *enteredText = textField.text;

CharacterClass *cc = nil;

if ([enteredText isEqualToString:@“Warrior”]) {

cc = knight;

} else if ([enteredText isEqualToString:@“Mage”]) {

cc = wizard;

} else if ([enteredText isEqualToString:@“Thief”]) {

cc = rogue;

}

character.characterClass = cc;

}
```

当读者需要编写包含大量if-else或switch语句的代码时，通常应该考虑替换为NSDictionary。NSDictionary可以事先在两组对象之间建立一对一的映射关系。例如，上述代码中的if-else语句可以替换为一个NSDictionary对象：

```
NSMutableDictionary *lookup = [[NSMutableDictionary alloc] init];
```

[lookup setObject:knight forKey:@“Warrior”];

[lookup setObject:wizard forKey:@“Mage”];

[lookup setObject:rogue forKey:@“Thief”];

有了lookup查询表，changeCharacterClass:方法就可以简化为：

-（void）changeCharacterClass:（id）sender

{

character.characterClass = [lookup objectForKey:textField.text];

}

使用NSDictionary查询表的另一个优点：不需要在方法中硬编码所有数据（角色类型）；相反，可以将数据保存在文件系统或远程服务器中，甚至可以由用户动态添加或修改。

BNRImageStore将使用NSDictionary查询表存储照片。BNRImageStore会为每一张照片生成唯一的键，之后可以通过键查找对应的照片。

使用字典对象时，键不能重复。在将某个键-值对加入字典对象时，如果字典对象已经保存了拥有相同的键的值，那么旧的值会被替换掉。如果要用一个键来保存多个对象，则可以先将这些对象存入数组对象，然后将这个数组对象作为值存入字典对象。

与NSArray类似，NSDictionary提供了用于创建对象的简洁语法。请读者回忆NSArray的简洁语法并注意与NSDictionary的区别：NSArray是通过“@[]”创建的，NSDictionary则是通过“@{}”创建的。

使用简洁语法创建NSDictionary对象时，每一个键值对之间需要使用逗号“，”隔开，而键与值之间则使用冒号“：”隔开，例如，

NSDictionary *dictionary = @{@“key”: object, @“anotherKey”: anotherObject};

另外，NSDictionary也可以使用NSArray中的下标语法，只需要将“[]”中的序号换成键，就可以读取该键所对应的值，例如，

id object = dictionary[@“key”];

// 与以下代码效果相同

id object = [dictionary objectForKey:@“key”];

而NSMutableDictionary还可以通过下标语法设置键所对应的值：

dictionary[@“key”] = object;

// 与以下代码效果相同

[dictionary setObject:object forKey:@“key”];

下面更新BNRImageStore，使用下标语法存取dictionary中的UIImage对象：

- (void)setImage:(UIImage *)image forKey:(NSString *)key

{

~~[self.dictionary setObject:image forKey:key];~~

self.dictionary[key] = image;

}

- (UIImage *)imageForKey:(NSString *)key

{

~~return [self.dictionary objectForKey:key];~~

return self.dictionary[key];

}

字典对象的内存管理和数组对象类似。当字典对象加入某个对象后，会成为该对象的拥有方。当字典对象移除某个对象后，会放弃该对象的拥有权。

11.5 创建并使用键

将照片加入BNRImageStore对象时，需要针对不同的照片使用不同的键，然后将这个键赋给相应的BNRItem对象。当BNRDetailViewController对象要从BNRImageStore对象载入照片时，需要先从BNRItem对象得到照片的键，然后通过BNRImageStore对象查找相应的照片。在BNRItem.h中为BNRItem类添加itemKey属性，用来保存照片的键，代码如下：

```
@property (nonatomic, readonly, strong) NSDate *dateCreated;

@property (nonatomic, copy) NSString *itemKey;
```

照片的键不能重复，否则无法通过BNRImageStore对象中的字典对象准确地保存对象。有很多种途径可以生成无重复的字符串(unique string)，本节将使用Cocoa Touch提供的一种机制，这种机制可以生成唯一标识(UUID，也称为GUID)。每一个NSUUID类的对象都表示一个唯一的UUID。UUID是基于当前时间、计数器(counter)和硬件标识(通常为无线网卡的MAC地址)等数据计算生成的。如果使用字符串表示UUID，则示例如下：

```
4A73B5D2-A6F4-4B40-9F82-EA1E34C1DC04
```

在BNRDetailViewController.m顶部导入BNRImageStore.h，代码如下：

```
#import “BNRDetailViewController.h”

#import “BNRItem.h”

#import “BNRImageStore.h”
```

打开BNRItem.m，修改指定初始化方法，生成一个UUID并设置为itemKey：

```
- (instancetype)initWithItemName:(NSString *)name

valueInDollars:(int)value

serialNumber:(NSString *)sNumber

{

// 调用父类的指定初始化方法

self = [super init];

// 父类的指定初始化方法是否成功创建了父类对象？

if (self) {

// 为对象变量设定初始值
```

```
    _itemName = name;
    _serialNumber = sNumber;
    _valueInDollars = value;
    // 设置_dateCreated的值为系统当前时间
    _dateCreated = [[NSDate alloc] init];
    // 创建一个NSUUID对象，然后获取其NSString类型的值
    NSUUID *uuid = [[NSUUID alloc] init];
    NSString *key = [uuid UUIDString];
    _itemKey = key;
    }
    // 返回初始化后的对象的新地址
    return self;
    }
```

然后修改BNRDetailViewController.m中的imagePickerController:didFinish-PickingMediaWithInfo:，将UIImage对象存入BNRImageStore对象：

```
    -（void）imagePickerController:（UIImagePickerController *）picker
    didFinishPickingMediaWithInfo:（NSDictionary *）info
    {
    UIImage *image = info[UIImagePickerControllerOriginalImage];
    // 以itemKey为键，将照片存入BNRImageStore对象
    [[BNRImageStore sharedStore] setImage:image
    forKey:self.item.itemKey];
    imageView.image = image;
    [self dismissViewControllerAnimated:YES completion:nil];
    }
```

每当BNRDetailViewController获取到UIImage对象之后，都会将其存入BNRImageStore对象。BNRImageStore对象与BNRItem对象都保存了UIImage对象的键，随时可以通过键找到需要的UIImage对象。

类似地，在用户删除了某个BNRItem对象后，需要同时在BNRImageStore对象中删除对应的UIImage对象。在BNRDetailViewController.m顶部导入BNRImageStore.h，然后修改removeItem:方法，代码如下：

```
#import “BNRImageStore.h”

@implementation BNRItemStore

-（void）removeItem:（BNRItem *）item

{

NSString *key = item.itemKey;

[[BNRImageStore sharedStore] deleteImageForKey:key];

[self.privateItems removeObjectIdenticalTo:item];

}
```

读者也许会问，为什么不直接为BNRItem添加一个属性指向UIImage对象呢？这样不就可以直接处理UIImage对象了吗？虽然目前确实可以这么做，但是当本书第18章升级Homepwner，将BNRItem对象与UIImage对象存入文件系统时，就会遇到问题。

系统在创建了一个UIImage对象后，会将其保存在某块内存区域中，该内存区域有一个地址，称为内存地址。如果没有将UIImage对象从内存移动至文件系统，那么UIImage对象的内存地址将保持不变，确实可以通过添加一个UIImage属性，访问该属性指向的内存地址，找到对应的UIImage对象；但是，如果将UIImage对象存入文件系统，当应用重新启动时（第18章会详细介绍应用的生命周期），就需要将UIImage对象从文件系统载入内存，这时UIImage对象的内存地址会发生变化，无法使用之前的内存地址找到该对象。相反，如果使用键关联UIImage对象，BNRImageStore会先根据键在文件系统中找到对应的UIImage对象，然后将其载入内存，返回新的UIImage对象指针。因此，使用键可以正确地持久化保存UIImage对象。

11.6 使用BNRImageStore

当Homepwner需要显示BNRDetailViewController对象的视图时(见图11-11)，该对象需要通过当前选中的BNRItem对象的itemKey属性，从BNRImageStore对象得到相应的照片，然后将该照片放置在UIImageView对象上。

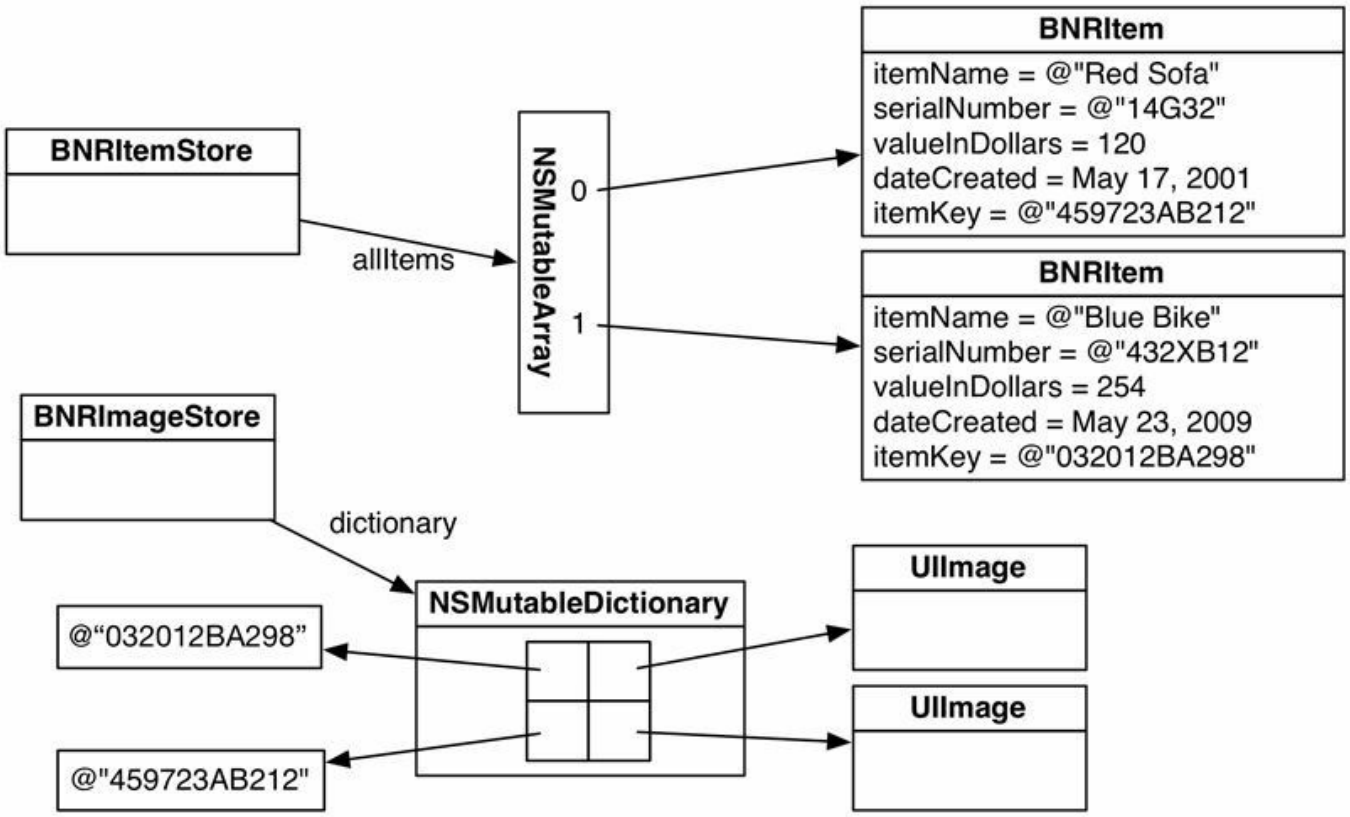


图11-11 缓存

Homepwner会在两种情况下显示BNRDetailViewController对象的视图:①当用户选中BNRItemsViewController对象的UITableView对象的某行时。②当用户关闭UIImagePickerController对象时。无论哪种情况，BNRDetailViewController对象都应该在UIImageView对象上显示当前选中的BNRItem对象的照片。将以下代码加入BNRDetailViewController.m中的viewWillAppear:。

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

{

[super viewWillAppear:animated];

self.nameField.text = item.itemName;

self.serialNumberField.text = item.serialNumber;

self.valueField.text = [NSString stringWithFormat:@“%d”,
```

```
    item.valueInDollars];

    static NSDateFormatter *dateFormatter = nil;

    if（！dateFormatter）{

    dateFormatter = [[NSDateFormatter alloc] init];

    dateFormatter.dateStyle = NSDateFormatterMediumStyle;

    dateFormatter.timeStyle = NSDateFormatterNoStyle;

    }

    self.dateLabel.text = [dateFormatter stringFromDate:item.dateCreated];

    NSString *itemKey = self.item.itemKey;

    // 根据itemKey，从BNRImageStore对象获取照片

    UIImage *imageToDisplay =

    [[BNRImageStore sharedStore] imageForKey:itemKey];

    // 将得到的照片赋给UIImageView对象

    self.imageView.image = imageToDisplay;

    }
```

如果针对指定的键，BNRImageStore对象没有包含相应的照片（或者选中的BNRItem对象的itemKey属性是nil），那么UIImageView对象的对象变量image的值会是nil。当image属性是nil时，UIImageView对象不会有任何显示。

构建并运行应用，添加一个新的BNRItem对象，然后进入其详细界面。点击工具栏上的相机按钮，为该对象拍摄一张照片，Homepwner应该能正确地显示拍摄的照片。

11.7 关闭键盘

Homepwner在显示BNRDetailViewController对象的视图时，系统显示的键盘会挡住子视图imageView，使用户无法看到完整的照片。下面为BNRDetailViewController实现委托方法textFieldShouldReturn:，使用户能通过按下“换行”键来取消UITextField对象的第一响应状态而关闭键盘（这也是为什么之前会为UITextField对象的插座变量delegate创建关联）。首先，在BNRDetailViewController.h中将BNRDetailViewController声明为遵守UITextFieldDelegate协议，代码如下：

```
@interface BNRDetailViewController ()

<UINavigationControllerDelegate, UIImagePickerControllerDelegate,

UITextFieldDelegate>
```

然后在BNRDetailViewController.m中实现textFieldShouldReturn:，代码如下：

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField

{

[textField resignFirstResponder];

return YES;

}
```

为了让Homepwner提供更佳的用户体验，需要在用户轻按BNRDetailViewController对象的视图的其他区域时，也能关闭键盘。向顶层视图发送endEditing:消息会使UITextField对象（顶层视图的子视图）取消第一响应状态而关闭键盘。下面要解决的问题是如何在用户按下视图时发送指定的消息。

前文曾经介绍过如何为UIButton这类对象设置目标-动作对，使其能够在被按下时，向指定的目标对象发送指定的动作消息。UIButton的这种目标-动作特性继承自父类UIControl。下面要将BNRDetailViewController对象的视图从UIView对象改为UIControl对象，使其能够处理触摸事件。

选中BNRDetailViewController.xib中的顶层视图。打开标识检视面板，将Class文本框中的UIView修改为UIControl（见图11-12）。

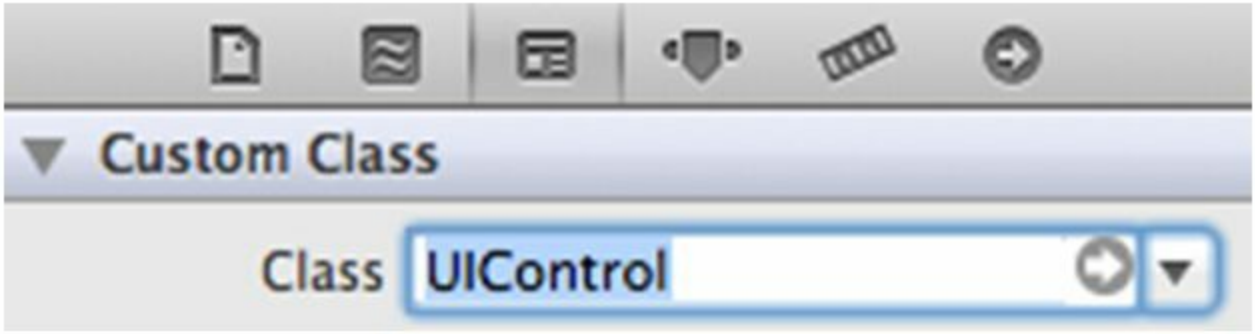


图11-12 修改BNRDetailViewController对象的视图的类

在辅助编辑器中打开BNRDetailViewController.m。按住Control键，从顶层视图开始（已经改为了UIControl）拖曳至BNRDetailViewController.m的方法实现区域。出现弹出窗口后，选择Connection下拉菜单中的Action。注意，这里的窗口和“创建并关联UIBarButtonItem时出现的窗口”略有不同。UIBarButtonItem对象是简化版本的UIControl，只能在被按下时向目标对象发送动作消息。而UIControl对象能够处理多种不同的事件，并发送相应的动作消息。

因为UIControl能够处理多种类型的事件，所以必须为要触发的动作消息设置合适的事件类型。对BNRDetailViewController对象的视图，因为要在用户按下视图时发送指定的动作消息，所以应该将事件类型设置为UIControlEventTouchUpInside（当手指在UIControl对象的bounds区域内离开屏幕时触发）。根据图11-13进行设置，然后点击Connect按钮。

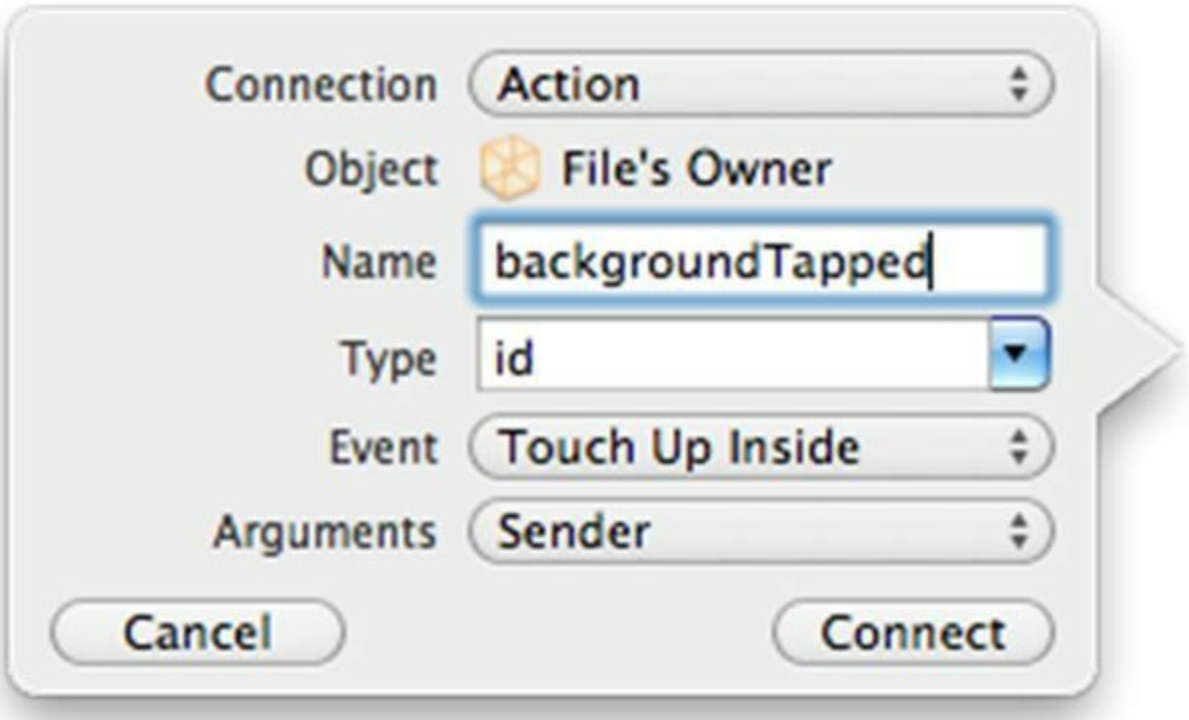


图11-13 为UIControl对象设置动作关联

Xcode会在BNRDetailViewController.m中创建相应的空方法backgroundTapped:。将以下代码

加入backgroundTapped:。

```
- (IBAction)backgroundTapped:(id)sender
{
[self.view endEditing:YES];
}
```

构建并运行应用，测试两种关闭键盘的途径。

11.8 初级练习：编辑照片

UIImagePickerController自带了一套照片编辑界面，可以用来编辑当前选中的照片。修改代码，使用户能够编辑照片，并且在BNRDetailViewController对象的视图中显示修改后的照片。

11.9 中级练习：删除照片

添加一个按钮，用于清除BNRItem对象的照片。

11.10 高级练习：Camera Overlay

UIImagePickerController有一个名为cameraOverlayView的属性。修改代码，使UIImagePickerController对象能够在取景视图的正中显示一个十字。

11.11 深入学习：导航实现文件

UIViewController的头文件和实现文件中可能有很多方法，为了在阅读或编写代码时快速找到需要的方法，Xcode在代码编辑器中设置了一个跳转栏，其位置如图11-14所示。

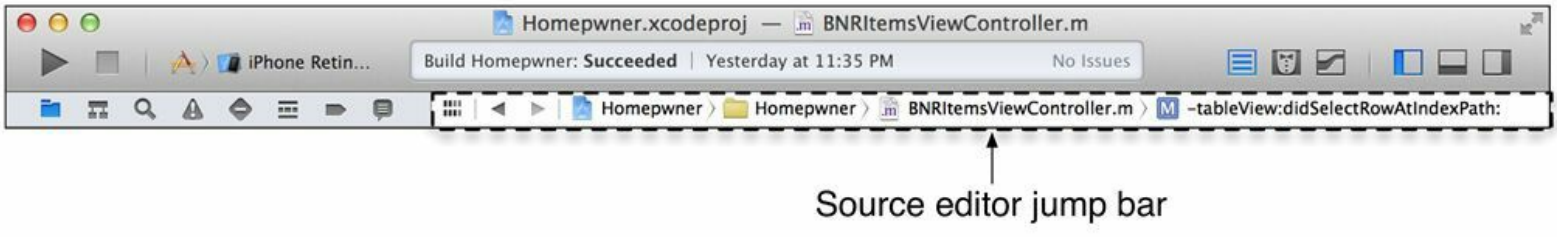


图11-14 代码编辑器中的跳转栏

跳转栏会显示完整的导航路径：项目？文件组？文件？方法（见图11-15），其中，“方法”表示文件中光标所在的方法。

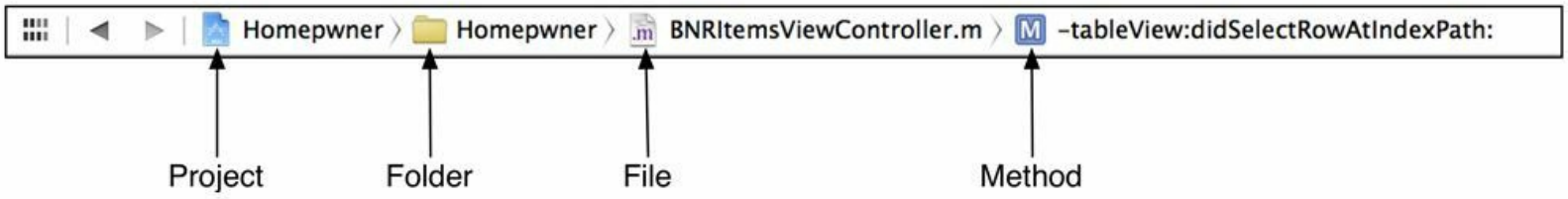


图11-15 导航路径

导航路径完全对应于项目层次结构，点击任何一级路径，Xcode都会弹出项目中的同级层次结构，可以非常方便地导航到项目中的任意部分。例如，点击文件路径，Xcode就会弹出同级文件列表（见图11-16）。

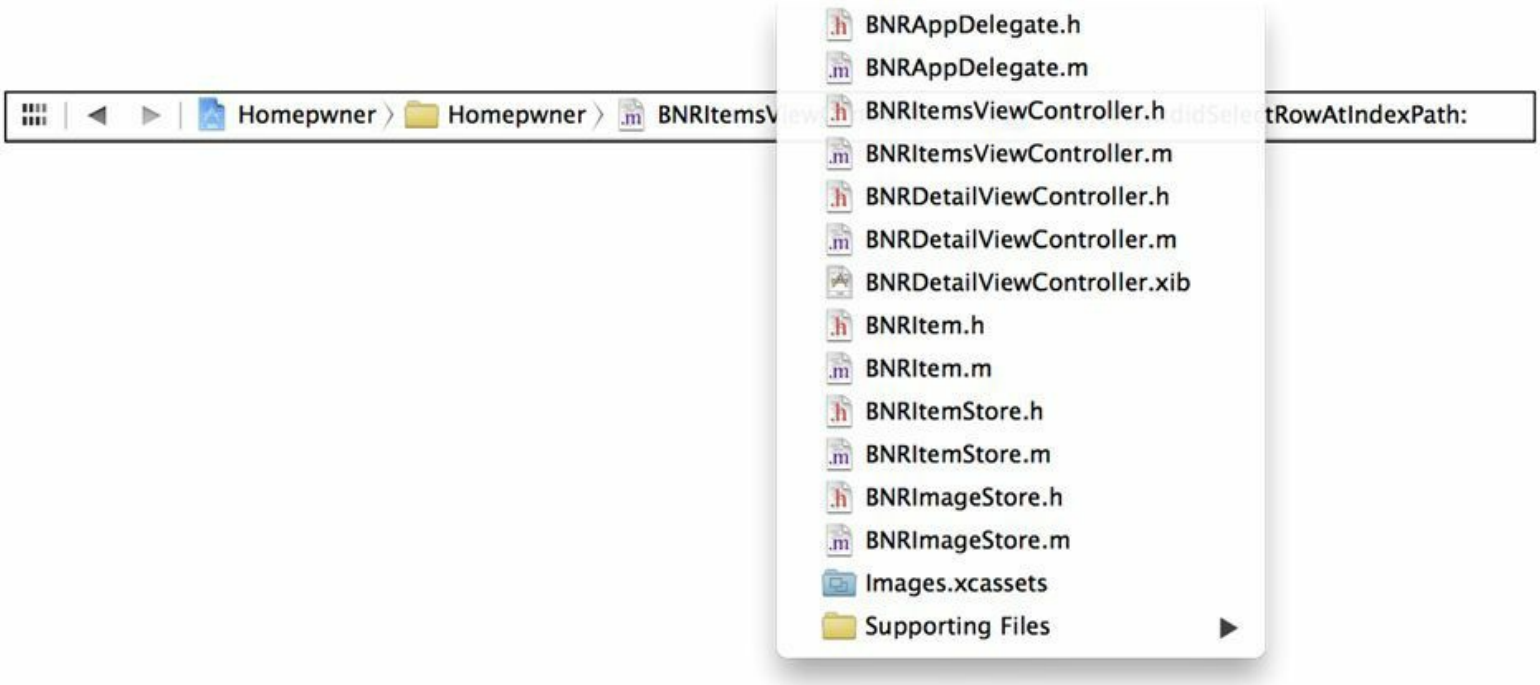


图11-16 同级文件列表

下面介绍跳转栏最常用也是最重要的一项功能：在实现文件中导航。点击方法路径，Xcode会弹出当前文件中的所有方法列表。这时可以输入关键词，在方法列表中搜索需要的方法；还可以使用上下方向键选择列表中的方法，然后按下Enter键跳转到选择的方法。图11-17显示了在BNRItemsViewController.m中搜索“indexpath”的方法列表。

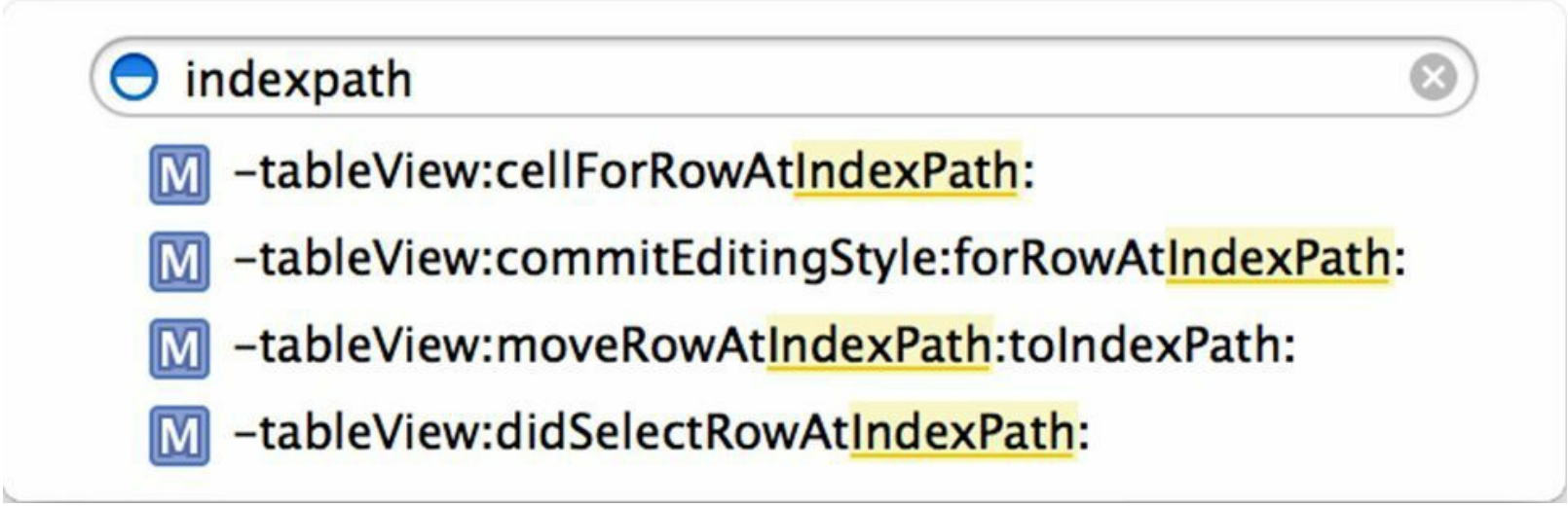


图11-17 搜索“indexpath”的方法列表

#pragma mark

随着类中的方法越来越多，在冗长的方法列表中查找需要的方法也会越来越困难。优秀的iOS开发者的一个习惯是使用#pragma mark预处理指令将方法按照功能分组，写在文件中的特定区域，这样，方法列表中就会分组显示各项功能的方法，查找起来非常方便。

```
#pragma mark - View life cycle

- (void)viewDidLoad {…}

- (void)viewWillAppear:(BOOL)animated {…}

#pragma mark - Actions

- (void)addNewItem:(id)sender {…}
```

#pragma mark不会对代码本身起任何作用，但是Xcode会根据#pragma mark组织文件中的方法。图11-18显示了在BNRItemsViewController.m中添加#pragma mark后的方法列表。

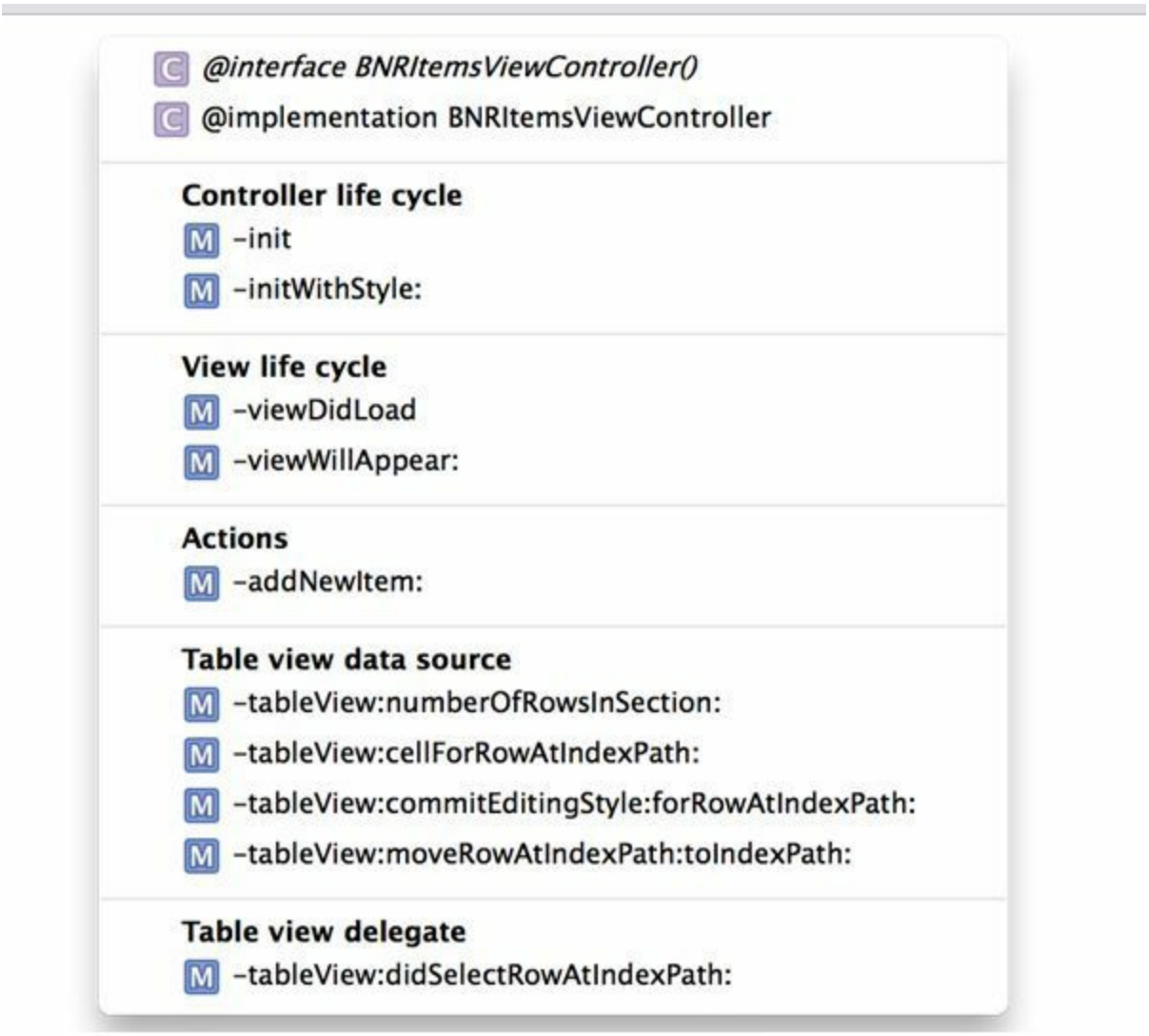


图11-18 代码编辑器中的跳转栏

可以通过#pragma mark为一组方法添加分割线和功能说明:

```
// 添加分割线

#pragma mark -

// 添加功能说明

#pragma mark My Awesome Methods

// 同时添加分割线和功能说明

#pragma mark - My Awesome Methods
```

经常使用#pragma mark, 可以使代码更加清晰有条理。一旦之后需要重新审查代码, 就会感受到事先使用了#pragma mark的好处。久而久之, 读者也会养成使用#pragma mark分组方法的好习惯, 成为优秀的iOS开发者。

11.12 深入学习：摄像

理解了如何通过UIImagePickerController实现拍照功能，实现摄像功能也会很容易。前文曾经提过，它的sourceType属性决定了照片的来源是相机、照片库还是最近拍摄的照片。它还有一个名为mediaTypes的属性，该属性是一个数组对象，包含媒体类型标识（NSString对象）。mediaTypes属性的作用是限制用户选择媒体类型。

UIImagePickerController对象可以选择的媒体类型有两种，分别为静态照片和视频。mediaTypes数组默认只包含常量字符串kUTTypeImage。因此，如果不修改该对象的mediaTypes属性，那么用户只能使用相机拍摄静态照片，而照片库和最近拍摄的照片中也只会显示静态照片。

添加摄像（或者选取已有视频）功能很简单，只需将常量字符串kUTTypeMovie加入mediaTypes数组即可。但是有些设备不支持摄像功能。和类方法isSourceTypeAvailable:（检查设备是否有相机）相似，availableMediaTypesForSourceType:方法可以检查相机是否能拍摄视频。以下代码可以将UIImagePickerController对象设置为既能摄像又能拍照。

```
UIImagePickerController *ipc = [[UIImagePickerController alloc] init];

NSArray *availableTypes = [UIImagePickerController

availableMediaTypesForSourceType:UIImagePickerControllerSourceTypeCamera];

ipc.mediaTypes = availableTypes;

ipc.sourceType = UIImagePickerControllerSourceTypeCamera;

ipc.delegate = self;
```

加入摄像功能的UIImagePickerController界面会多出一个开关，用户使用这个开关，可以在照相模式和摄像模式之间切换。如果用户选择摄像模式，就需要在UIImagePickerController的委托方法imagePickerController: didFinishPickingMediaWithInfo:中处理摄像结果。

处理静态照片时，传入imagePickerController:didFinishPickingMediaWithInfo:的info参数（NSDictionary类型）会包含一个UIImage对象，以对应整张照片。但是针对拍摄的视频，Cocoa Touch没有提供相应的“UIVideo类”（移动设备内存有限，一次载入整个视频不现实）。因此，UIImagePickerController对象会将拍摄的视频存入临时目录。当用户结束摄像时，该对象的委托对象会收到imagePickerController: didFinishPickingMediaWithInfo:消息，并且传入的info参数会包含视频的文件路径。获取路径的代码如下：

```
-（void）imagePickerController:（UIImagePickerController *）picker

didFinishPickingMediaWithInfo:（NSDictionary *）info

{
```

```
NSURL *mediaURL = info[UIImagePickerControllerMediaURL];

}
```

第18章会详细介绍文件系统，读者目前需要知道的是，将视屏文件留在临时目录里是“不安全”的，应该将文件移至其他目录，代码如下：

```
- (void)imagePickerController:(UIImagePickerController *)picker

didFinishPickingMediaWithInfo:(NSDictionary *)info

{

NSURL *mediaURL = info[UIImagePickerControllerMediaURL];

if (mediaURL) {

// 确定设备是否支持视频

if (UIVideoAtPathIsCompatibleWithSavedPhotosAlbum([mediaURL path])) {

// 将视频存入相册

UISaveVideoAtPathToSavedPhotosAlbum([mediaURL path], nil, nil, nil);

// 删除临时目录下的视频

[[NSFileManager defaultManager] removeItemAtPath:[mediaURL path]

error:nil];

}

}

}
```

最后还有一点需要额外说明：如何限制用户只能拍摄视频（或者只能选择现有的视频）？限制用户只能选择照片很简单（保留mediaTypes的默认值即可）。允许用户在照片和视频之间做一个选择也很简单（将mediaTypes设置为availableMediaTypesForSourceType:的返回值即可）。但是，如果要限制用户只使用视频，就会复杂一些。首先必须确定设备是否支持视频，然后设置mediaTypes属性指向一个只包含视频标识的数组对象。

```
NSArray *availableTypes =

[UIImagePickerController availableMediaTypesForSourceType:

UIImagePickerControllerSourceTypeCamera];
```

if ([availableTypes containsObject:(__bridge NSString *)kUTTypeMovie]) {

[ipc setMediaTypes:@[(__bridge NSString *)kUTTypeMovie]];

}

为什么要将kUTTypeMovie转成NSString？这是因为常量kUTTypeMovie的类型是CFStringRef，其定义如下：

const CFStringRef kUTTypeVideo;

编译这段代码，编译器会报告无法找到kUTTypeMovie的定义。这是因为kUTTypeMovie和kUTTypeImage都是在MobileCoreServices框架中声明的，所以必须先导入该框架：

@import MobileCoreServices;

第12章 触摸事件与UIResponder

本书接下来的三章将创建一个名为TouchTracker的应用，深入学习触摸事件和手势处理，还会教读者如何调试应用程序。

本章将创建一个“画板”视图，用户可以在该视图上触摸并绘制线条（见图12-1）。借助多点触摸，用户可以同时画多根线条。

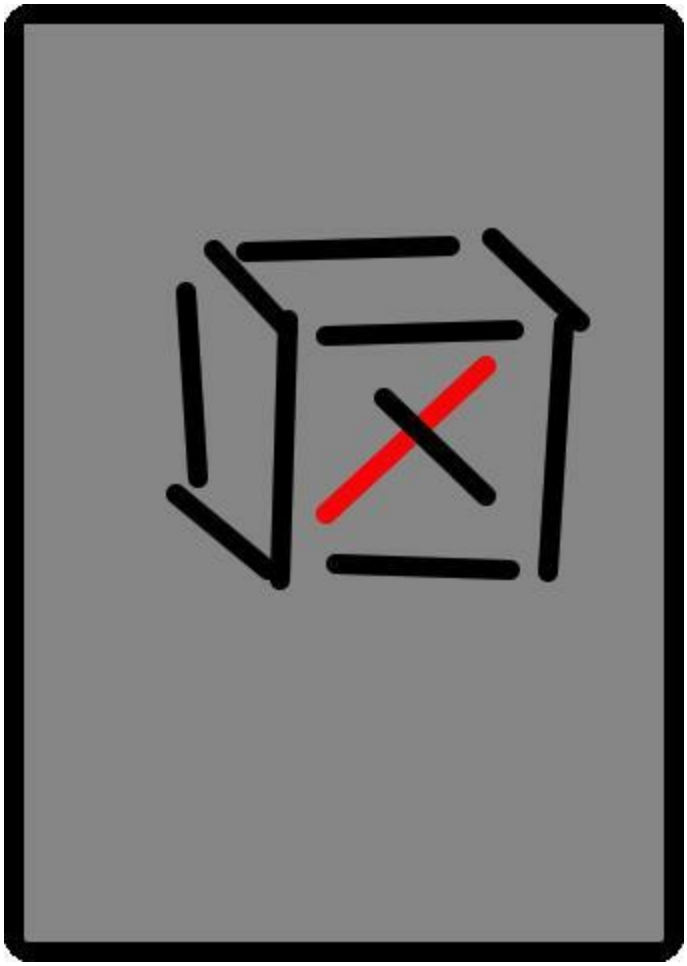


图12-1 画板应用

12.1 触摸事件

因为UIView是UIResponder的子类，所以覆盖以下四个方法就可以处理四种不同的触摸事件：

•一根手指或多根手指触摸屏幕。

- (void)touchesBegan:(NSSet *)touches

withEvent:(UIEvent *)event;

•一根手指或多根手指在屏幕上移动(随着手指的移动，相关的对象会持续发送该消息)。

- (void)touchesMoved:(NSSet *)touches

withEvent:(UIEvent *)event;

•一根手指或多根手指离开屏幕。

- (void)touchesEnded:(NSSet *)touches

withEvent:(UIEvent *)event;

•在触摸操作正常结束前，某个系统事件(例如有电话进来)打断了触摸过程。

- (void)touchesCancelled:(NSSet *)touches

withEvent:(UIEvent *)event;

当系统检测到手指触摸屏幕的事件后，就会创建UITouch对象(一根手指的触摸事件对应一个UITouch对象)。发生触摸事件的UIView对象会收到touchesBegan:withEvent:消息，系统传入的第一个实参touches(NSSet对象)会包含所有相关的UITouch对象。

当手指在屏幕上移动时，系统会更新相应的UITouch对象，为其重新设置对应的手指在屏幕上的位置。最初发生触摸事件的那个UIView对象会收到touchesMoved:withEvent:消息，系统传入的第一个实参touches(NSSet对象)会包含所有相关的UITouch对象，而且这些UITouch对象都是最初发生触摸事件时创建的。

当手指离开屏幕时，系统会最后一次更新相应的UITouch对象，为其重新设置对应的手指在屏幕上的位置。接着，最初发生该触摸事件的视图会收到touchesEnded:withEvent:消息。当收到该消息的视图执行完touchesEnded:withEvent:后，系统就会释放和当前事件有关的UITouch对象。

下面对UITouch对象和事件响应方法的工作机制做一个归纳。

•一个UITouch对象对应屏幕上的一根手指。只要手指没有离开屏幕，相应的UITouch对象就会一直存在。这些UITouch对象都会保存对应的手指在屏幕上的当前位置。

•在触摸事件的持续过程中，无论发生什么，最初发生触摸事件的那个视图都会在各个阶段收到相应的触摸事件消息。即使手指在移动时离开了这个视图的frame区域，系统还是会向该视图发送touchesMoved:withEvent:和touchesEnded:withEvent:消息。也就是说，当某个视图发生触摸事件后，该视图将永远“拥有”当时创建的所有UITouch对象。

•读者自己编写的代码不需要也不应该保留任何UITouch对象。当某个UITouch对象的状态发生变化时，系统会向指定的对象发送特定的事件消息，并传入发生变化的UITouch对象。

当应用发生某个触摸事件后（例如触摸开始、手指移动、触摸结束），系统都会将该事件添加至一个由UIApplication单例管理的事件队列。通常情况下，很少会出现满队列的情况，所以UIApplication会立刻分发队列中的事件。分发某个触摸事件时，UIApplication会向“拥有”该事件的视图发送特定的UIResponder消息（如果读者在执行触摸操作时感觉应用的反应迟缓，就很有可能是因为应用的某个方法占用了大量CPU时间，导致队列堵塞。第14章会介绍如何查出产生这类问题的原因）。

当多根手指在同一个视图、同一个时刻执行相同的触摸动作时，UIApplication会用单个消息、一次分发所有相关的UITouch对象。UIApplication在发送特定的UIResponder消息时，会传入一个NSSet对象，该对象将包含所有相关的UITouch对象（一个UITouch对象对应一根手指）。但是，因为UIApplication对“同一个时刻”的判断很严格，所以通常情况下，哪怕一组事件都是在很短的一段时间内发生的，UIApplication也会发送多个UIResponder消息，分批发送UITouch对象。

12.2 创建TouchTracker应用

下面创建一个名为TouchTracker的应用。通过Xcode的Empty Application模板，创建一个新的iPhone项目并将其命名为TouchTracker。Class Prefix保持与之前项目相同，填入BNR（见图12-2）。

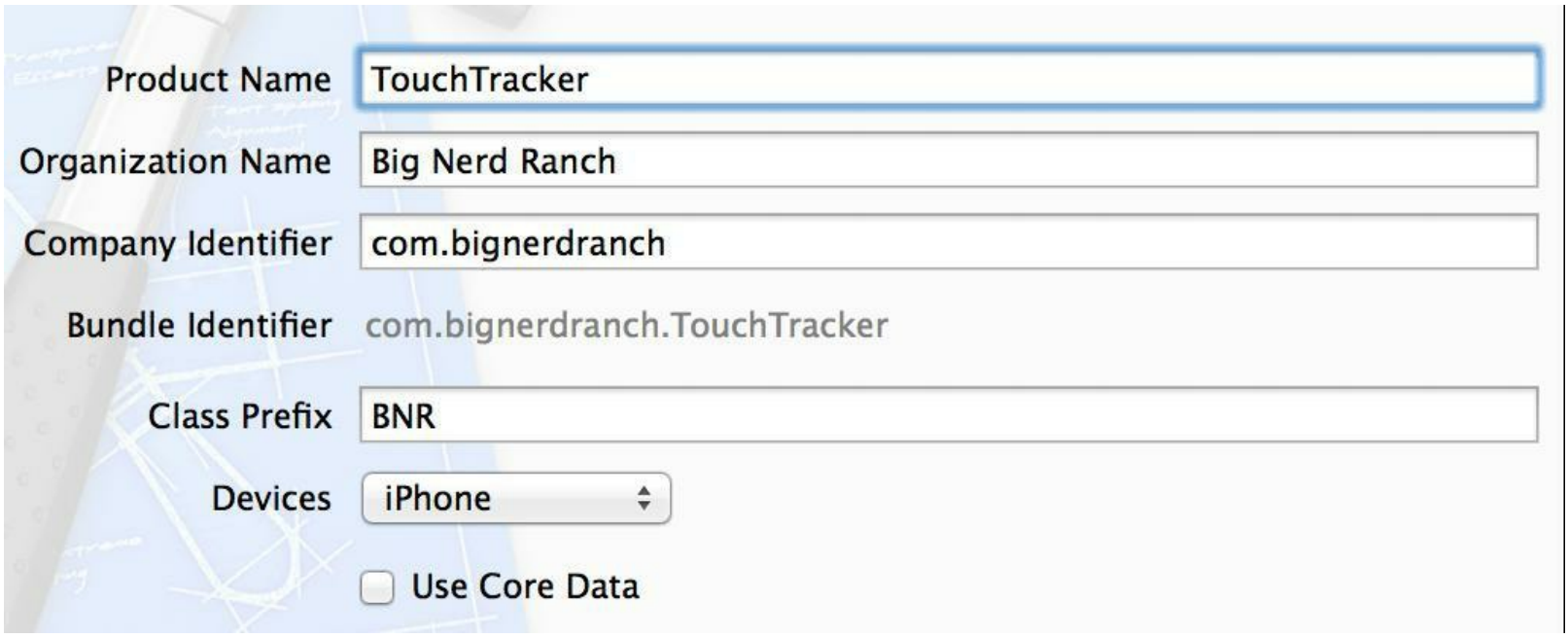


图12-2 创建TouchTracker

首先，TouchTracker需要一个能够描述线条的模型对象。创建一个新的NSObject子类并将其命名为BNRLine。在BNRLine.h中，声明两个CGPoint类型的属性，代码如下：

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

@interface BNRLine : NSObject

@property (nonatomic) CGPoint begin;

@property (nonatomic) CGPoint end;

@end
```

接着，创建一个新的NSObject子类并将其命名为BNRDrawView。在BNRDrawView.h中，将BNRDrawView的父类改为UIView，代码如下：

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

~~@interface BNRDrawView : NSObject~~

@interface BNRDrawView : UIView

@end
```

下面创建一个UIViewController子类，用于管理BNRDrawView对象。创建一个新的NSObject子类并将其命名为BNRDrawViewController。在BNRDrawViewController.h中，将BNRDrawViewController的父类改为UIViewController，代码如下：

```
@interface BNRDrawViewController : NSObject

@interface BNRDrawViewController : UIViewController
```

在BNRDrawViewController.m中，先导入BNRDrawView.h，然后覆盖loadView方法，创建一个BNRDrawView对象并将其赋给BNRDrawViewController对象的view属性，代码如下：

```
#import “BNRDrawViewController.h”

#import “BNRDrawView.h”

@implementation BNRDrawViewController

- (void)loadView

{

self.view = [BNRDrawView alloc] initWithFrame:CGRectZero];

}

@end
```

在BNRAppDelegate.m中，先导入BNRDrawViewController.h，然后创建一个BNRDrawViewController对象，并将新创建的对象设置为UIWindow的rootView- Controller，代码如下：

```
#import “BNRAppDelegate.h”

#import “BNRDrawViewController.h”

@implementation BNRAppDelegate

- (BOOL)application:(UIApplication *)application

didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

{

self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];

// 在此处加入自定义代码

BNRDrawViewController *dvc = [[BNRDrawViewController alloc] init];
```

```
self.window.rootViewController = dvc;
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}
```

图12-3是目前TouchTracker的对象图。

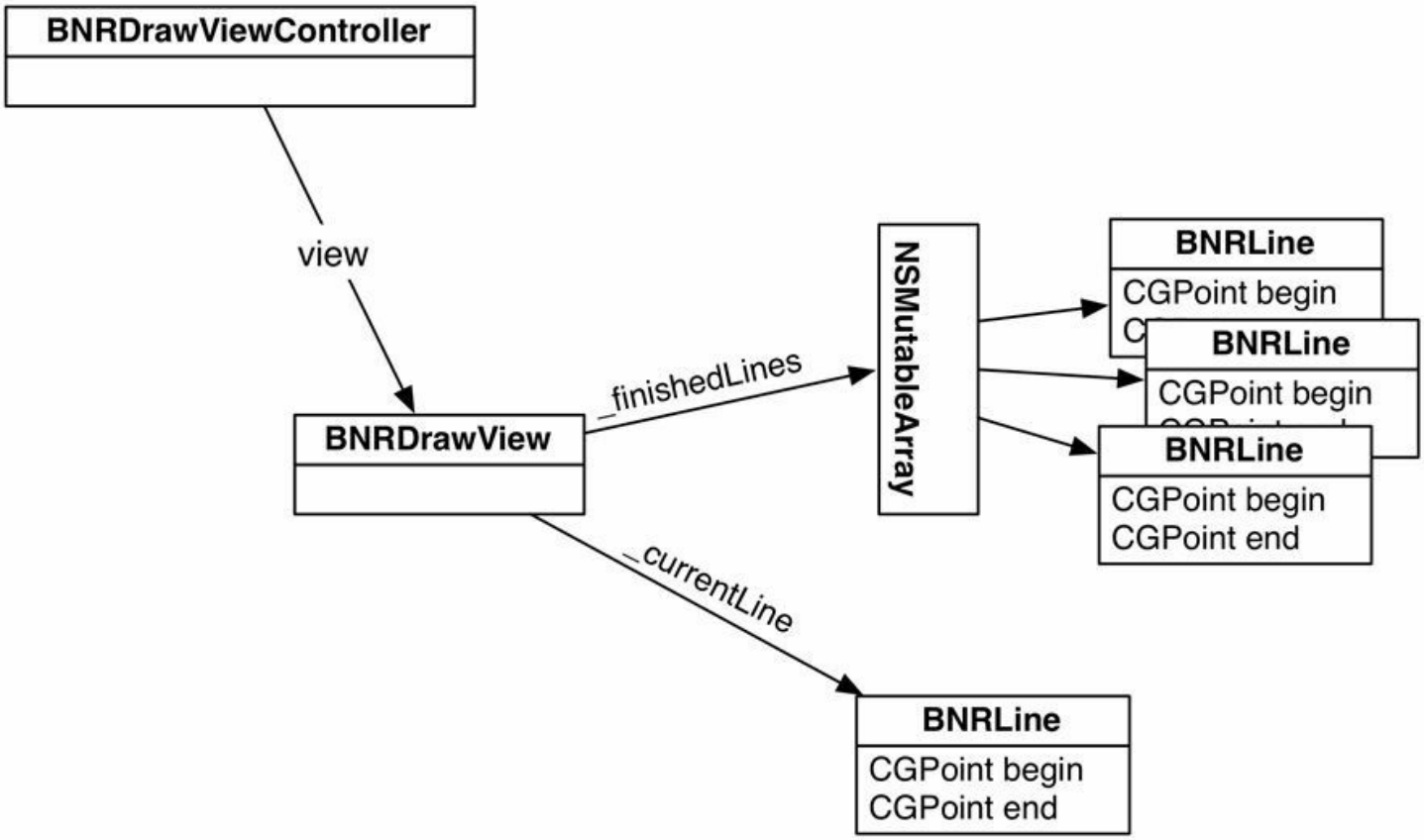


图12-3 TouchTracker对象图

12.3 实现BNRDrawView，完成绘图功能

BNRDrawView对象需要管理正在绘制的线条和绘制完成的线条。在BNRDrawView.m中，先导入BNRLine.h；然后在类扩展中添加两个属性，用于保存这两种线条；最后实现initWithFrame:，代码如下：

```
#import “BNRDrawView.h”

#import “BNRLine.h”

@interface BNRDrawView（）

@property（nonatomic, strong）BNRLine *currentLine;

@property（nonatomic, strong）NSMutableArray *finishedLines;

@end

@implementation BNRDrawView

-（instancetype）initWithFrame:（CGRect）r

{

self = [super initWithFrame:r];

if（self）{

self.finishedLines = [[NSMutableArray alloc] init];

self.backgroundColor = [UIColor grayColor];

}

return self;

}
```

为了测试创建线条的代码是否正确，需要在BNRDrawView中编写绘制线条的代码。在BNRDrawView.m中实现drawRect:方法，代码如下：

```
-（void）strokeLine:（BNRLine *）line

{

UIBezierPath *bp = [UIBezierPath bezierPath];
```

```
bp.lineWidth = 10;
bp.lineCapStyle = kCGLineCapRound;
[bp moveToPoint:line.begin];
[bp addLineToPoint:line.end];
[bp stroke];
}
-（void）drawRect:（CGRect）rect
{
// 用黑色绘制已经完成的线条
[[UIColor blackColor] set];
for（BNRLine *line in self.finishedLines）{
[self strokeLine:line];
}
if（self.currentLine）{
// 用红色绘制正在画的线条
[[UIColor redColor] set];
[self strokeLine:self.currentLine];
}
}
```

12.4 处理触摸事件并创建线条对象

因为通过两个点可以定义一条直线，所以BNRLine对象需要用begin属性和end属性来保存这两个点。当触摸事件开始时，BNRDrawView对象需要创建一个BNRLine对象，并将begin和end都设置为触摸发生时的手指位置。当触摸事件继续时（手指在屏幕上移动），BNRDrawView对象要将end设置为手指的当前位置。当触摸结束时，这个BNRLine对象就能代表完成后的线条。

在BNRDrawView.m中实现touchesBegan:withEvent:方法，创建BNRLine对象：

```
- (void)touchesBegan:(NSSet *)touches
withEvent:(UIEvent *)event
{
UITouch *t = [touches anyObject];
// 根据触摸位置创建BNRLine对象
CGPoint location = [t locationInView:self];
self.currentLine = [[BNRLine alloc] init];
self.currentLine.begin = location;
self.currentLine.end = location;
[self setNeedsDisplay];
}
```

接下来实现touchesMoved:withEvent:方法，获取currentLine的终点：

```
- (void)touchesMoved:(NSSet *)touches
withEvent:(UIEvent *)event
{
UITouch *t = [touches anyObject];
CGPoint location = [t locationInView:self];
self.currentLine.end = location;
[self setNeedsDisplay];
```

```
}
```

最后实现touchesEnded:withEvent:方法，将currentLine加入finishedLines：

```
- (void)touchesEnded:(NSSet *)touches
withEvent:(UIEvent *)event
{
[self.finishedLines addObject:self.currentLine];
self.currentLine = nil;
[self setNeedsDisplay];
}
```

构建并运行应用，在屏幕上绘制线条。可以发现，当手指正在屏幕上绘制时，绘制的线条是红色的；当手指离开屏幕时，线条的颜色会变为黑色。

处理多点触摸

读者可能已经注意到，如果在使用一根手指绘制的同时使用别的手指触摸屏幕，并不会同时画出多根线条。接下来将更新BNRDrawView，使TouchTracker可以处理多点触摸。

默认情况下，视图在同一时刻只能接受一个触摸事件。如果一根手指已经触发了touchesBegan:withEvent:方法，那么在手指离开屏幕之前（触发touchesEnded: withEvent:方法之前），其他触摸事件都会被忽略——对于BNRDrawView来说，“忽略”是指touchesBegan:withEvent:或其他UIResponder消息都不会再发送给BNRDrawView。

为了使BNRDrawView同时接受多个触摸事件，需要在BNRDrawView.m中添加以下代码：

```
- (instancetype)initWithFrame:(CGRect)r
{
self = [super initWithFrame:r];
if (self) {
self.finishedLines = [[NSMutableArray alloc] init];
self.backgroundColor = [UIColor grayColor];
```

```
    self.multipleTouchEnabled = YES;

    }

    return self;

    }
```

现在当多根手指在屏幕上触摸、移动、离开时，BNRDrawView都将收到相应的UIResponder消息。但是现有代码并不能正确处理这些消息：现有代码在同一时刻只能处理一个触摸消息（只能画出一根线条）。

之前实现的触摸方法中，代码向NSSet类型的touches发送了anyObject消息——在只能接受单点触摸的视图中，touches在同一时刻只会包含一个触摸事件，因此anyObject可以正确返回唯一的触摸事件。但是在可以接受多点触摸的视图中，touches在同一时刻可能包含一个或多个触摸事件，必须修改现有代码，依次处理所有触摸事件。

目前，代码中只有一个currentLine属性用于保存正在绘制的直线。读者可能会考虑为BNRDrawView添加多个属性保存多条直线，例如currentLine1和currentLine2，但是这种方法很难处理众多BNRLine属性与触摸事件之间的对应关系。假设用户用三根手指同时触摸屏幕并因此创建了三个BNRLine对象，当其中一根手指移动时就难以知道应该更新哪个BNRLine属性。

更好的解决方案是使用NSMutableDictionary对象来保存正在绘制的线条：发生触摸事件时，BNRDrawView可以根据传入的UITouch对象创建BNRLine对象并将两者关联存入NSMutableDictionary（其实并不能直接使用UITouch对象作为NSMutableDictionary的键，后面会介绍如何使用UITouch对象的内存地址存取BNRLine对象）。当BNRDrawView再次收到触摸事件时，可以根据传入的UITouch对象在NSMutableDictionary中找到并更新相应的BNRLine对象。

在BNRDrawView.m中，添加一个NSMutableDictionary类型的属性，名为linesIn- Progress，用于代替currentLine。然后在initWithFrame:中初始化linesInProgress：

```
    @interface BNRDrawView（）

    @property (nonatomic, strong) BNRLine *currentLine;

    @property（nonatomic, strong）NSMutableDictionary *linesInProgress;

    @property（nonatomic, strong）NSMutableArray *finishedLines;

    @end

    @implementation BNRDrawView

    -（instancetype）initWithFrame:（CGRect）r

    {
```

```
self = [super initWithFrame:r];
if (self) {
self.linesInProgress = [[NSMutableDictionary alloc] init];
self.finishedLines = [[NSMutableArray alloc] init];
self.backgroundColor = [UIColor grayColor];
self.multipleTouchEnabled = YES;
}
return self;
}
```

现在更新UIResponder方法，将所有正在绘制的线条加入linesInProgress。在BNRDrawView.m中更新touchesBegan:withEvent: :

```
- (void)touchesBegan:(NSSet *)touches
withEvent:(UIEvent *)event
{
// 向控制台输出日志，查看触摸事件发生顺序
NSLog(@"%@", NSStringFromSelector(_cmd));
for (UITouch *t in touches) {
CGPoint location = [t locationInView:self];
BNRLine *line = [[BNRLine alloc] init];
line.begin = location;
line.end = location;
NSValue *key = [NSValue valueWithNonretainedObject:t];
self.linesInProgress[key] = line;
}
~~UITouch *t = [touches anyObject];~~
```

~~CGPoint location = [t locationInView:self];~~

~~self.currentLine = [[BNRLine alloc] init];~~

~~self.currentLine.begin = location;~~

~~self.currentLine.end = location;~~

[self setNeedsDisplay];

}

以上代码使用快速枚举将所有已经开始的触摸事件加入linesInProgress——同时请读者注意，加入linesInProgress之前，需要使用valueWithNonretainedObject:方法将UITouch对象的内存地址封装为NSValue对象，作为BNRLine对象的键。使用内存地址分辨UITouch对象的原因是，在触摸事件开始、移动、结束的整个过程中，其内存地址是不会改变的，内存地址相同的UITouch对象一定是同一个对象。现在TouchTracker的对象图如图12-4所示。

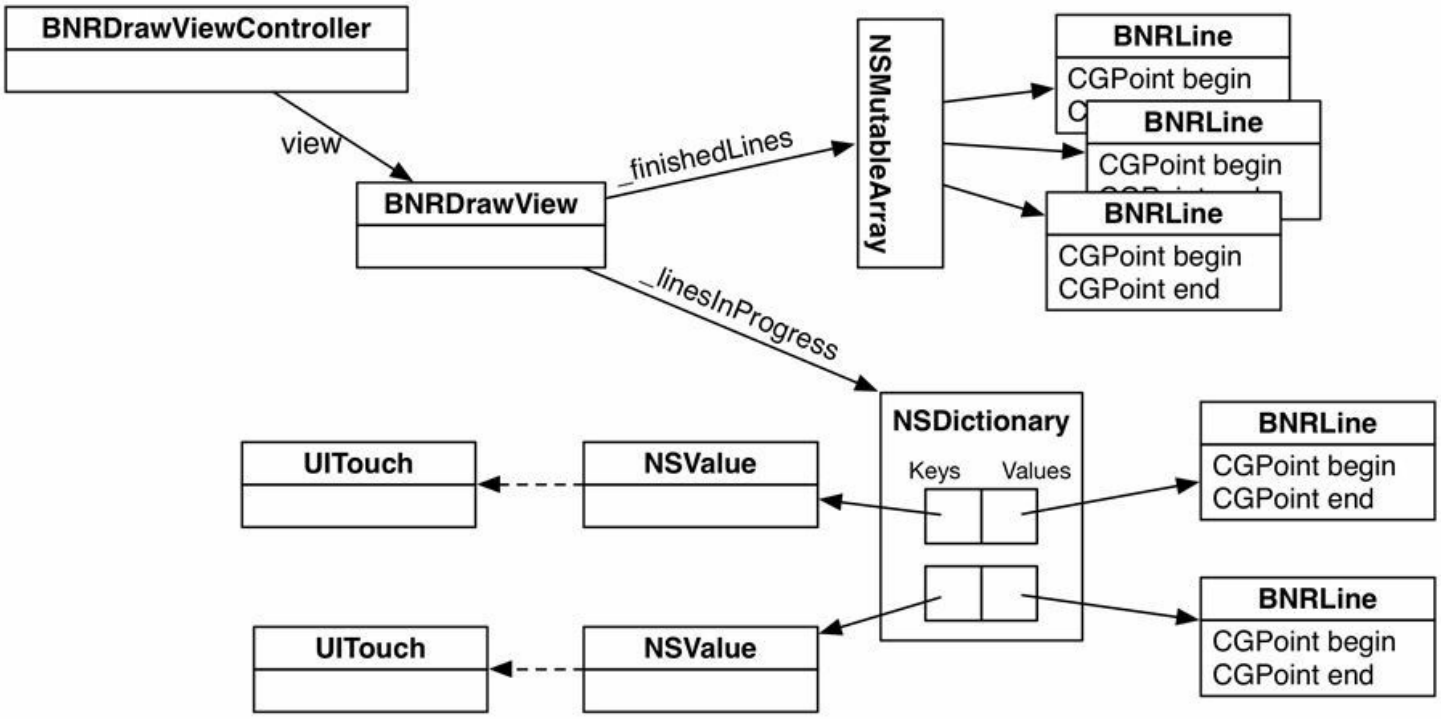


图12-4 TouchTracker对象图

继续更新其余UIResponder方法，在touchesMoved:withEvent:中根据UITouch对象查找对应的线条：

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event

{

// 向控制台输出日志，查看触摸事件发生的顺序

NSLog(@"%@", NSStringFromSelector(_cmd));

for（UITouch *t in touches）{

NSValue *key = [NSValue valueWithNonretainedObject:t];

BNRLine *line = self.linesInProgress[key];

line.end = [t locationInView:self];

}

~~UITouch *t = [touches anyObject];~~

~~CGPoint location = [t locationInView:self];~~

~~self.currentLine.end = location;~~

[self setNeedsDisplay];

}

然后在touchesEnded:withEvent:中将所有绘制完成的线条移动到_finishedLines数组中：

-（void）touchesEnded:（NSSet *）touches withEvent:（UIEvent *）event

{

// 向控制台输出日志，查看触摸事件发生的顺序

NSLog(@"%@", NSStringFromSelector(_cmd));

for（UITouch *t in touches）{

NSValue *key = [NSValue valueWithNonretainedObject:t];

BNRLine *line = self.linesInProgress[key];

[self.finishedLines addObject:line];

[self.linesInProgress removeObjectForKey:key];

}

~~[self.finishedLines addObject:self.currentLine];~~

~~self.currentLine = nil;~~

```
[self setNeedsDisplay];

}
```

最后更新drawRect:方法，绘制linesInProgress中的所有线条：

```
// 用黑色绘制已经完成的线条

[[UIColor blackColor] set];

for（BNRLine *line in self.finishedLines）{

[self strokeLine:line];

}

// 用红色绘制正在画的线条

[[UIColor redColor] set];

for（NSValue *key in self.linesInProgress）{

[self strokeLine:self.linesInProgress[key]];

}

~~if (self.currentLine) {~~

~~// 用红色绘制正在画的线条~~

~~[[UIColor redColor] set];~~

~~[self strokeLine:self.currentLine];~~

~~}~~

}
```

构建并运行应用，同时使用多个手指在TouchTracker中绘制线条，检查运行结果（在模拟器中按住Option并拖曳可以模拟多点触摸）。

读者可能会问，为什么UITouch对象自身不能用作NSMutableDictionary的键？这是由于NSDictionary及其子类NSMutableDictionary的键必须遵守NSCopying协议——键必须可以复制（可以响应copy消息）。UITouch并不遵守NSCopying协议，因为每一个触摸事件都是唯一的，不应该被复制。相反，NSValue遵守NSCopying协议，同一个UITouch对象会在触摸过程中创建包含相同内存地址的NSValue对象。

读者还应该知道，当视图收到touchesMoved:withEvent:消息时，touches中只会包含正在移动

的UITouch对象。也就是说，如果使用三个手指同时触摸视图，但是只移动其中一个手指，其他两个手指保持不动，那么touches中只会包含一个UITouch对象。

最后还需要处理触摸取消事件。如果系统中断了应用，触摸事件就会被取消（例如iPhone接到电话）。这时应该将应用恢复到触摸事件发生前的状态，对于TouchTracker来说，需要清除所有正在绘制的线条。

在BNRDrawView.m中实现touchesCancelled:withEvent:方法：

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

{

// 向控制台输出日志，查看触摸事件发生的顺序

NSLog(@"%@", NSStringFromSelector(_cmd));

for (UITouch *t in touches) {

NSValue *key = [NSValue valueWithNonretainedObject:t];

[self.linesInProgress removeObjectForKey:key];

}

[self setNeedsDisplay];

}
```

12.5 初级练习：保存与读取

更新代码，使TouchTracker能够在停止运行时保存所有的BNRLine对象，重新启动时再读取并恢复BNRLine对象。

12.6 中级练习：颜色

将BNRLine对象加入_finishedLines时，根据线条的角度为BNRLine对象设置不同的绘图颜色。

12.7 高级练习：圆圈

为TouchTracker添加新功能，使用户可以用两根手指画出圆圈。画圈时，两根手指分别代表圆外矩形框的两个对角点（提示：实现画圈功能时，可以用两个独立的NSMutableDictioary对象来管理UITouch对象，这样容易很多）。

12.8 深入学习：响应对象链

第7章已简单介绍过UIResponder和第一响应对象。UIResponder对象可以接收触摸事件，而UIView是典型的UIResponder子类。除了UIView，还有很多其他的UIResponder子类，其中包括UIViewController、UIApplication和UIWindow。UIViewController不是视图对象，既不能触摸也无法显示，为什么也是UIResponder子类？这是因为虽然不能向UIViewController对象直接发送触摸事件，但是该对象能够通过响应对象链接收事件。

UIResponder对象拥有一个名为nextResponder的指针，相关的UIResponder对象可以通过该指针组成一个响应对象链（见图12-5）。当UIView对象属于某个UIViewController对象时，其nextResponder指针就会指向包含该视图的UIViewController对象。当UIView对象不属于任何UIViewController对象时，其nextResponder指针就会指向该视图的父视图。UIViewController对象的nextResponder通常会指向其视图的父视图。最顶层的父视图是UIWindow对象，而UIWindow对象的nextResponder指向的是UIApplication单例。

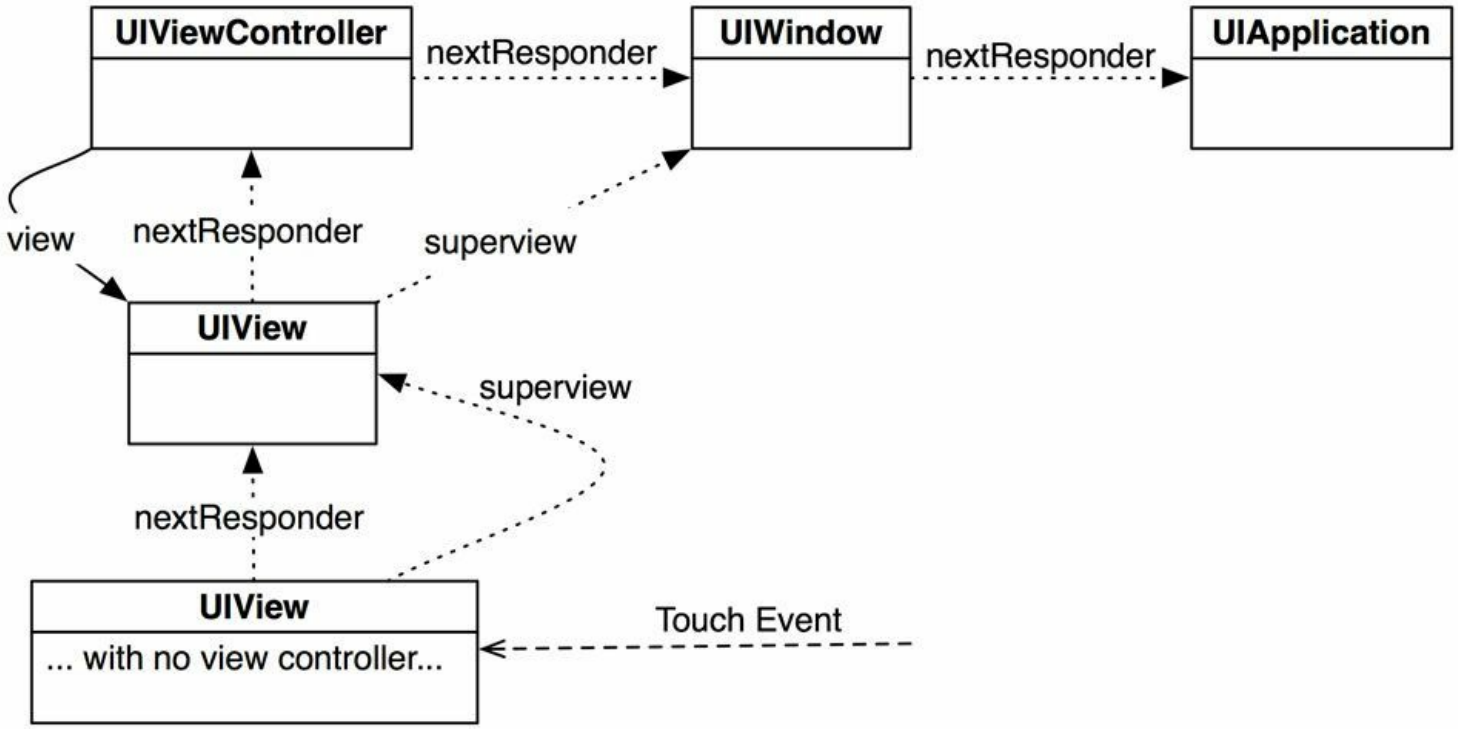


图12-5 响应对象链

如果UIResponder对象没有处理传给它的事件，会发生什么？该对象会将未处理的消息转发给自己的nextResponder。这也是touchesBegan:withEvent:这类方法的默认实现。因此，如果没有为某个UIResponder对象覆盖特定的事件处理方法，那么该对象的nextResponder会尝试处理相应的触摸事件。最终，该事件会传递给UIApplication（响应对象链的最后一个对象），如果UIApplication也无法对其处理，系统就会丢弃该事件。

除了由UIResponder对象向nextResponder转发消息，也可以直接向nextResponder发送消息。假设有一个正在跟踪触摸事件的视图，当该视图发生连按事件时，需要由该视图的nextResponder来处理这个事件。相应的代码如下：

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
UITouch *touch = [touches anyObject];
if (touch.tapCount == 2) {
[[self nextResponder] touchesBegan:touches withEvent:event];
return;
}
……继续处理非连按触摸事件
}
```

12.9 深入学习：UIControl

UIControl是部分Cocoa Touch类的父类，例如UIButton和UISlider。前面已经介绍过如何为这类UIControl对象设置目标对象和动作方法。学习完触摸事件与UIResponder的相关知识后，本节将进一步介绍UIControl是如何覆盖前面所介绍的那些UIResponder方法的。

对于UIControl对象，每个可能触发的控件事件（control event）都有一个对应的常量。以UIButton对象为例，该对象的常用控件事件是UIControlEventTouchUpInside。如果某个目标对象是针对UIControlEventTouchUpInside注册的，那么只有当用户触摸了这个UIControl对象，并且手指是在该对象的frame区域内离开屏幕时，目标对象才会收到指定的动作消息。因此，可以将控件事件UIControlEventTouchUpInside视为按下操作。

对于UIButton对象，除了UIControlEventTouchUpInside，还可以针对其他事件注册动作消息。例如，假设要完成以下任务：无论用户的手指是在frame区域内离开屏幕，还是在frame区域外离开屏幕，都要触发指定的方法。为了完成这项任务，可以同时注册两个控件事件，示例代码如下：

```
[rButton addTarget:tempController

action:@selector（resetTemperature:）

forControlEvents:UIControlEventTouchUpInside

| UIControlEventTouchUpOutside];
```

下面列出UIControl处理UIControlEventTouchUpInside的示例代码。

```
// 和实际代码不同，实际代码会更复杂一点！

- （void）touchesEnded:（NSSet *）touches withEvent:（UIEvent *）event

{

// 指针指向已经结束的触摸对象

UITouch *touch = [touches anyObject];

// 触摸结束时的位置（使用当前UIControl对象的坐标系）

CGPoint touchLocation = [touch locationInView:self];

// 结束时的位置是否在视图的bounds区域内？

if（CGRectContainsPoint（self.bounds, touchLocation））

{
```

```
// 向注册了UIControlEventTouchUpInside事件的所有目标对象发送动作消息

[self sendActionsForControlEvents:UIControlEventTouchUpInside];

} else {

// 触摸事件是在bounds区域外结束的,

// 所以要向注册了UIControlEventTouchUpOutside事件的所有目标对象发送动作消息

[self sendActionsForControlEvents:UIControlEventTouchUpOutside];

}

}
```

那么UIControl对象是如何将这些动作消息发送给相应的目标对象的？在上面这段touchesEnded:withEvent:方法的末尾，UIControl对象会向自己发送sendActions-ForControlEvents:消息。该消息会遍历UIControl对象的所有目标-动作对，根据传入的控件事件类型进行查找，然后向匹配的目标对象发送对应的动作消息。

但是，UIControl对象绝对不是直接向目标对象发送消息，而是要通过UIApplication转发。为什么UIControl对象不能直接向目标对象发送动作消息？这是因为在UIControl对象所拥有的目标-动作对中，目标对象可以是nil。UIApplication在转发源自UIControl对象的消息时，会先判断目标对象是不是nil。如果是nil，UIApplication就会先找出UIWindow对象的第一响应对象，然后向第一响应对象发送相应的动作消息。

第13章 UIGestureRecognizer与UIMenuController

第12章介绍了如何通过覆盖UIResponder的事件方法来直接处理触摸事件。编写iOS应用时，可能需要识别某组触摸事件是否符合特定的手势（gesture）规则，例如“张开或合拢两个手指”（pinch）或滑动（swipe）。对这类常见的手势，可以用特定的UIGestureRecognizer对象来识别，而不用自己编写相关的代码。

UIGestureRecognizer对象会截取本应由视图处理的触摸事件。当某个UIGestureRecognizer对象识别出特定的手势后，就会向指定的对象发送指定的消息。iOS SDK默认提供若干种UIGestureRecognizer对象。本章将更新TouchTracker，借助由iOS SDK提供的三种UIGestureRecognizer对象，用户可以选择、移动并删除线条（见图13-1）。此外，本章后半部分还会介绍另一个很有用的Objective-C类：UIMenuController。

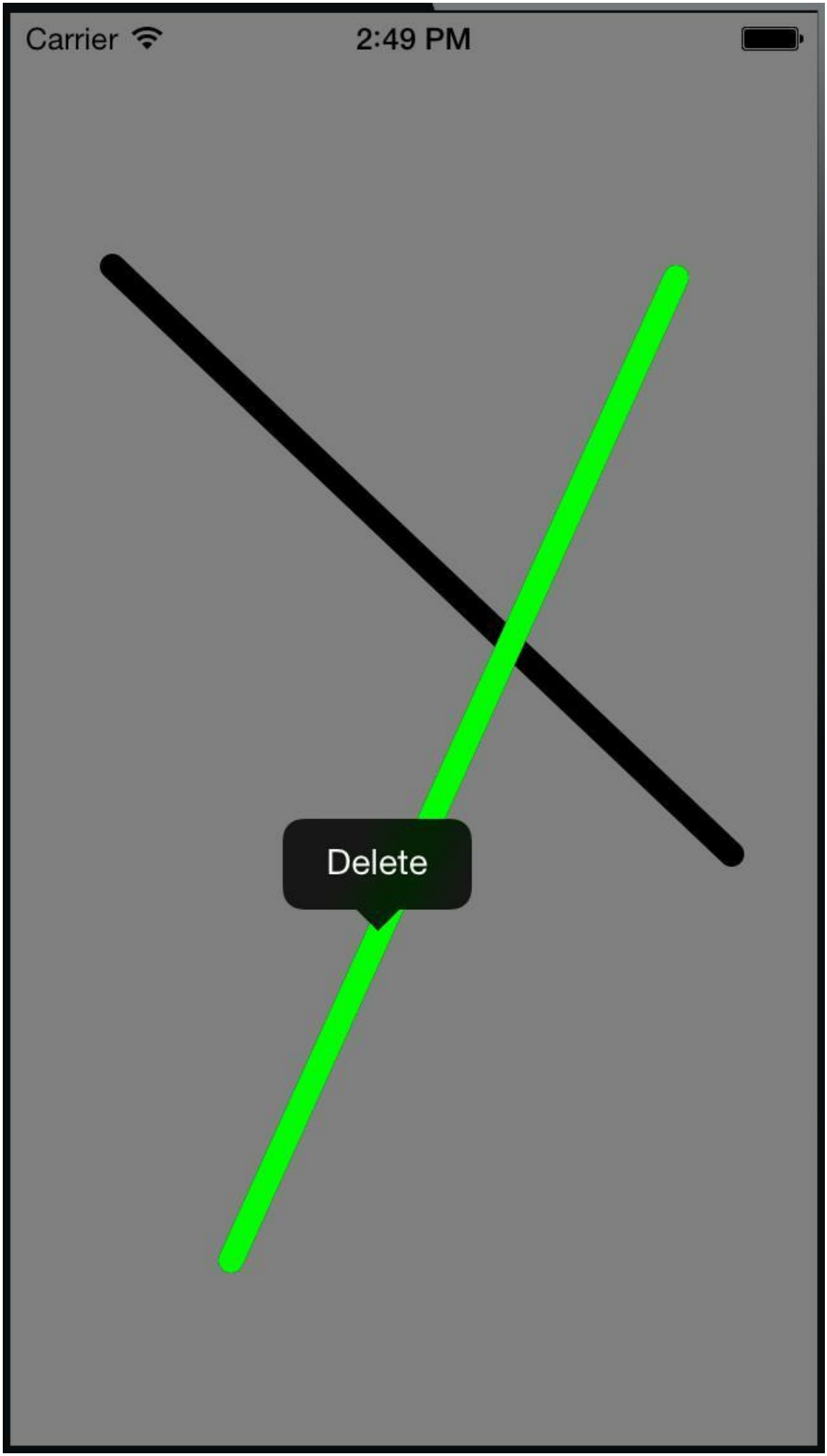

Carrier
2:49 PM
Delete

图13-1 完成后的TouchTracker

13.1 UIGestureRecognizer子类

在为应用添加手势识别功能时，需要针对特定的手势创建相应的UIGestureRecognizer子类对象，而不是直接使用UIGestureRecognizer对象。iOS SDK提供了多种能够处理不同手势的UIGestureRecognizer子类。

使用UIGestureRecognizer子类对象时，除了要设置目标-动作对，还要将该子类对象“附着”在某个视图上。当该子类对象根据当前附着的视图所发生的触摸事件识别出相应的手势时，就会向指定的目标对象发送指定的动作消息。由UIGestureRecognizer对象发出的动作消息都会遵守以下规范：

```
- (void)action:(UIGestureRecognizer *)gestureRecognizer;
```

UIGestureRecognizer对象在识别手势时，会截取本应由其附着的视图自行处理的触摸事件（见图13-2）。因此，附着了UIGestureRecognizer对象的视图可能不会收到常规的UIResponder消息，例如touchesBegan:withEvent:。

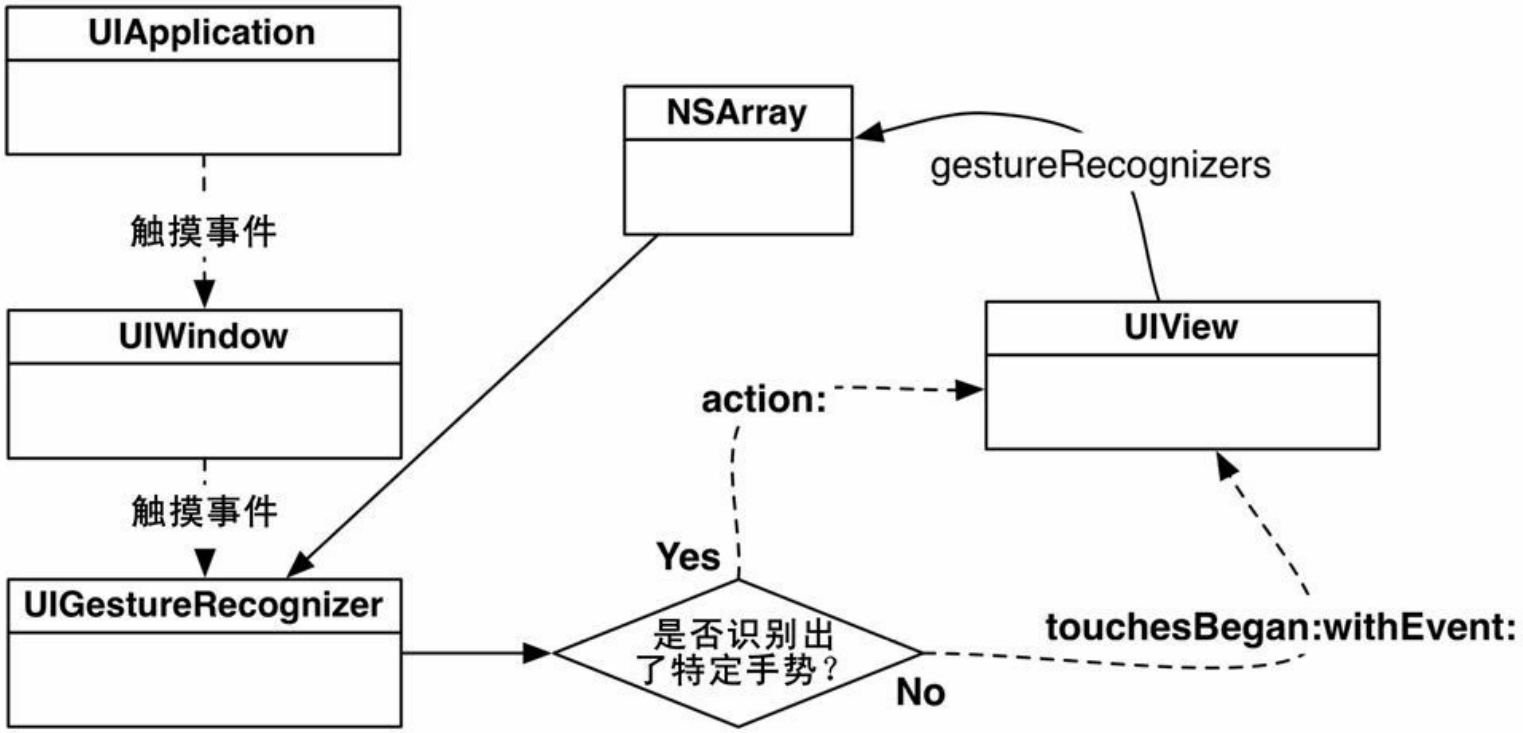


图13-2 UIGestureRecognizer对象会截取UITouch对象

13.2 用UITapGestureRecognizer对象识别“按下”手势

下面为TouchTracker添加第一个UIGestureRecognizer子类对象：UITapGesture- Recognizer，当用户双击屏幕时，会清除屏幕上的所有线条。打开在第12章中完成的TouchTracker.xcodeproj。

在BNRDrawView.m的initWithFrame:中创建一个UITapGestureRecognizer对象，设置点击次数为2，使BNRDrawView可以识别双击手势：

```
- (instancetype)initWithFrame:(CGRect)r

{

self = [super initWithFrame:r];

if (self) {

self.linesInProgress = [[NSMutableDictionary alloc] init];

self.finishedLines = [[NSMutableArray alloc] init];

self.backgroundColor = [UIColor grayColor];

self.multipleTouchEnabled = YES;

UITapGestureRecognizer *doubleTapRecognizer =

[[UITapGestureRecognizer alloc] initWithTarget:self

action:@selector(doubleTap:)];

doubleTapRecognizer.numberOfTapsRequired = 2;

[self addGestureRecognizer:doubleTapRecognizer];

}

return self;

}
```

当用户在BNRDrawView上双击时，BNRDrawView会收到doubleTap:消息，在BNRDrawView.m中实现doubleTap:：

```
- (void)doubleTap:(UIGestureRecognizer *)gr

{
```

```
NSLog(@"Recognized Double Tap");

[self.linesInProgress removeAllObjects];

[self.finishedLines removeAllObjects];

[self setNeedsDisplay];

}
```

双击事件不需要处理消息中传入的UIGestureRecognizer参数，但是本章稍后在处理其他几种手势的动作消息时，会介绍如何从UIGestureRecognizer参数中获取相关信息。

构建并运行应用，在屏幕上画几根线条，然后双击屏幕，所画的线条应该会全部清除。

读者可能会注意到，双击屏幕时，BNRDrawView会在手指双击的位置画出一个红色圆点（在模拟器上看起来更明显）。这个圆点是由BNRDrawView画出的一根很短的线条（当用户点击屏幕时，BNRDrawView会收到touchesBegan:withEvent:消息）。检查控制台中的日志信息，可以看见触摸事件发生的顺序：

```
touchesBegan:withEvent:

Recognized Double Tap

touchesCancelled:withEvent:
```

由于UIGestureRecognizer对象会通过截取触摸事件来识别手势，因此，在UIGestureRecognizer对象识别出手势之前，UIView会收到所有UIResponder消息。对于UITapGestureRecognizer来说，在识别出点击手势（在屏幕中的一小块区域触摸并迅速抬起手指）之前，UIView会收到touchesBegan:withEvent:消息；在识别出点击手势之后，UITapGestureRecognizer会自行处理相关触摸事件，由这些触摸事件所引发的UIResponder消息将不会再发送给UIView。直到UITapGestureRecognizer检测出点击手势已经结束，UIView才会重新收到UIResponder消息（touchesCancelled:withEvent:）。

为了去掉红色圆点，需要在识别出点击手势之前避免向UIView发送touchesBegan:withEvent:消息。在BNRDrawView.m中，修改initWithFrame:方法，代码如下：

```
UITapGestureRecognizer *doubleTapRecognizer =

[[UITapGestureRecognizer alloc] initWithTarget:self

action:@selector(doubleTap:)];

doubleTapRecognizer.numberOfTapsRequired = 2;

doubleTapRecognizer.delaysTouchesBegan = YES;
```

```
    [self addGestureRecognizer:doubleTapRecognizer];

    }

    return self;

    }
```

构建并运行应用，绘制一些线条，然后双击屏幕清除所有线条，这次应该不会再出现红色圆点了。

13.3 同时添加多种触摸手势

接下来为BNRDrawView添加单击手势，让用户可以选择屏幕上的线条（后面章节会添加删除选中线条的功能）。这次仍然使用UITapGestureRecognizer对象，但是点击次数需要改为1。（UITapGestureRecognizer的numberOfTapsRequired属性默认值就是1，不需要在代码中设置。）

在BNRDrawView.m中，修改initWithFrame:方法，代码如下：

```
[self addGestureRecognizer:doubleTapRecognizer];

UITapGestureRecognizer *tapRecognizer =

[[UITapGestureRecognizer alloc] initWithTarget:self

action:@selector(tap:)];

tapRecognizer.delaysTouchesBegan = YES;

[self addGestureRecognizer:tapRecognizer];

}

return self;

}
```

然后实现tap:方法，当UITapGestureRecognizer识别出单击手势时，会向控制台输出一条日志信息。

```
- (void)tap:(UIGestureRecognizer *)gr

{

NSLog(@"Recognized tap");

}
```

构建并运行应用。可以发现，点击一次可以正确识别出单击手势，控制台会输出tap:中的日志信息；但是，如果点击两次，BNRDrawView无法区分单击手势和双击手势，tap:和doubleTap:都会执行。

如果需要为视图添加多种手势，就需要考虑这些手势之间的关系。双击手势包含两次单击，为了避免UITapGestureRecognizer将双击事件分拆为两个单击事件，可以设置UITapGestureRecognizer在单击后暂时不进行识别，直到确定不是双击手势后再识别为单击手势。

在initWithFrame:中添加以下代码：

```
UITapGestureRecognizer *tapRecognizer =

[[UITapGestureRecognizer alloc] initWithTarget:self

action:@selector(tap:)];

tapRecognizer.delaysTouchesBegan = YES;

[tapRecognizer requireGestureRecognizerToFail:doubleTapRecognizer];

[self addGestureRecognizer:tapRecognizer];
```

构建并运行应用，单击屏幕，UITapGestureRecognizer会稍作停顿，确定是单击手势后再执行tap:方法；而双击屏幕不会再执行tap:方法了。

现在为BNRDrawView添加单击选择线条功能。首先在BNRDrawView.m的类扩展中添加一个属性，用于保存选中的线条，代码如下：

```
@interface BNRDrawView ()

@property (nonatomic, strong) NSMutableDictionary *linesInProgress;

@property (nonatomic, strong) NSMutableArray *finishedLines;

@property (nonatomic, weak) BNRLine *selectedLine;

@end
```

（请读者注意，以上代码将selectedLine设置为弱引用，原因是：首先，finishedLines数组会保存selectedLine，是强引用；其次，如果用户清除所有线条，finishedLines会移除selectedLine，这时BNRDrawView会自动将selectedLine设置为nil。）

下面在drawRect:中添加代码，用绿色绘制选中的线条：

```
[[UIColor redColor] set];

for (NSValue *key in self.linesInProgress) {

[self strokeLine:self.linesInProgress[key]];

}

if (self.selectedLine) {

[[UIColor greenColor] set];

[self strokeLine:self.selectedLine];
```

```
}
```

然后在BNRDrawView.m中实现lineAtPoint:，根据传入的位置找出距离最近的那个BNRLine对象，代码如下：

```
-（BNRLine *）lineAtPoint:（CGPoint）p
{
// 找出离p最近的BNRLine对象
for（BNRLine *l in self.finishedLines）{
CGPoint start = l.begin;
CGPoint end = l.end;
// 检查线条的若干点
for（float t = 0.0; t <= 1.0; t += 0.05）{
float x = start.x + t *（end.x - start.x）;
float y = start.y + t *（end.y - start.y）;
// 如果线条的某个点和p的距离在20点以内，就返回相应的BNRLine对象
if（hypot（x - p.x, y - p.y）< 20.0）{
return l;
}
}
}
// 如果没能找到符合条件的线条，就返回nil，代表不选择任何线条
return nil;
}
```

（要找到距离某个点最近的线条，还有更好的算法，以上的lineAtPoint:只是一个简化的实现。）

运行TouchTracker时，需要向lineAtPoint:传入手势点击的位置。通过UIGestureRecognizer对象，可以很容易地获取该信息。UIGestureRecognizer对象有一个名为locationInView:的方法，该

方法会根据传入的UIView对象的坐标系返回手势发生时的位置信息。

在BNRDrawView.m的tap:中先调用UIGestureRecognizer对象的locationInView:，然后将得到的位置信息作为实参传给lineAtPoint:，最后将返回的BNRLine对象赋给selectedLine。

```
- (void)tap:(UIGestureRecognizer *)gr

{

NSLog(@"Recognized tap");

CGPoint point = [gr locationInView:self];

self.setSelectedLine = [self lineAtPoint:point];

[self setNeedsDisplay];

}
```

构建并运行应用。先画一些线条，然后点击其中的某根线条，TouchTracker应该会用绿色重绘这根线条。

13.4 UIMenuController

下面要为TouchTracker增加另一项功能：当用户选中某根线条时，TouchTracker要在用户手指按下的位置显示一个菜单。这个菜单要为用户提供一个删除选项。iOS SDK提供了一个名为UIMenuController的类，可以用来显示这类菜单。UIMenuController对象可以包含一组UIMenuItem对象（菜单项），并能在现有的视图上显示这些UIMenuItem对象。每个UIMenuItem对象都有自己的标题（会在菜单中显示）和动作方法（UIMenuItem对象会向第一响应对象发送动作消息）（见图13-3）。

图13-3 UIMenuController对象示例

每个iOS应用只有一个UIMenuController对象。当应用要显示该对象时，要先为它设置一组UIMenuItem对象，然后设置显示位置（矩形区域），最后将其设置为可见。在BNRDrawView.m的tap:中完成上述过程，使TouchTracker能够在用户按下线条时，在按下的位置显示菜单。此外，还要在tap:中实现隐藏菜单的功能：当手指按下的位置附近没有线条时，TouchTracker要取消当前选中的线条并隐藏UIMenuController对象，代码如下：

```
- (void)tap:(UIGestureRecognizer *)gr

{

NSLog(@"Recognized tap");

CGPoint point = [gr locationInView:self];

self.selectedLine = [self lineAtPoint:point];

if (self.selectedLine) {

// 使视图成为UIMenuItem动作消息的目标

[self becomeFirstResponder];

// 获取UIMenuController对象

UIMenuController *menu = [UIMenuController sharedMenuController];
```

```
// 创建一个新的标题为“Delete”的UIMenuItem对象
UIMenuItem *deleteItem = [[UIMenuItem alloc]
initWithTitle:@“Delete” action:@selector(deleteLine:)];
menu.menuItems = @[deleteItem];
// 先为UIMenuController对象设置显示区域，然后将其设置为可见
[menu setTargetRect:CGRectMake(point.x, point.y, 2, 2) inView:self];
[menu setMenuVisible:YES animated:YES];
} else {
// 如果没有选中的线条，就隐藏UIMenuController对象
[[UIMenuController sharedMenuController] setMenuVisible:NO
animated:YES];
}
[self setNeedsDisplay];
}
```

要显示UIMenuController对象，还要满足一个条件：显示UIMenuController对象的UIView对象必须是当前UIWindow对象的第一响应对象。这也是为什么在tap:的起始部分会向BNRDrawView对象发送becomeFirstResponder消息。如果要将某个自定义的UIView子类对象设置为第一响应对象，就必须覆盖该对象的canBecomeFirstResponder方法。

在BNRDrawView.m中覆盖canBecomeFirstResponder方法并返回YES，代码如下：

```
-（BOOL）canBecomeFirstResponder
{
return YES;
}
```

构建并运行应用，选中某根线条，但是TouchTracker并不会按预期显示UIMenuController对象。这是因为该对象在显示前会枚举所有的UIMenuItem对象，检查第一响应对象是否实现了指定的动作方法。如果没有实现该方法，UIMenuController对象就不会显示相应的UIMenuItem对象；如果所有UIMenuItem对象的动作方法都没有实现，应用就不会显示UIMenuController对象。

在BNRDrawView.m中实现deleteLine:，显示UIMenuController对象和Delete菜单项，代码如下：

```
- (void)deleteLine:(id)sender

{

// 从已经完成的线条中删除选中的线条

[self.finishedLines removeObject:self.selectedLine];

// 重画整个视图

[self setNeedsDisplay];

}
```

构建并运行应用。画一根线条，按下并选中该线条，选择菜单中的Delete菜单项，TouchTracker应该会删除这根线条。

13.5 UILongPressGestureRecognizer

下面介绍另外两个UIGestureRecognizer子类:UILongPressGestureRecognizer和UIPanGestureRecognizer。当用户按住某根线条(长按)不放时,TouchTracker应该选中该线条。如果某根线条通过上述手势选中,就应该允许用户拖移(pan)这根线条至新位置。

本节先介绍UILongPressGestureRecognizer。在BNRDrawView.m的initWithFrame:中先创建一个UILongPressGestureRecognizer对象,然后将该对象附着在BNRDrawView对象上,代码如下:

```
[self addGestureRecognizer:tapRecognizer];

UILongPressGestureRecognizer *pressRecognizer

= [[UILongPressGestureRecognizer alloc]

initWithTarget:self action:@selector(longPress:)];

[self addGestureRecognizer:pressRecognizer];
```

当用户按住BNRDrawView对象不放时,附着在该对象上的UILongPressGesture- Recognizer对象就会向其发送longPress:消息。UILongPressGestureRecognizer对象默认会将持续时间超过0.5秒的触摸事件识别为长按手势。设置UILongPressGesture- Recognizer对象的minimumPressDuration属性可以修改这个时间。

“点击屏幕”是一种简单的手势,只有一个事件。UITapGestureRecognizer对象一旦识别出按下手势,就会触发按下事件并结束该手势的识别过程。“长按屏幕”则不同,UILongPressGestureRecognizer对象在识别出长按手势后,会持续跟踪该手势并在不同的阶段分别触发三种不同的事件。

例如,当用户触摸某个UIView对象后,UILongPressGestureRecognizer对象会将这个触摸事件识别为“可能会发生”的长按手势。为了能够准确地识别长按手势,UILongPressGestureRecognizer对象必须等待后续事件,根据触摸时间做下一步的判断。

当触摸的时间足够长时,UILongPressGestureRecognizer对象就会将其识别为长按手势并触发“长按开始”事件。当用户的手指离开屏幕时,该对象会触发“长按结束”事件。

当某个UIGestureRecognizer子类对象触发特定的事件后,其state属性会发生变化。以UILongPressGestureRecognizer对象为例,和上述三种事件相对应的state属性分别为:UIGestureRecognizerStatePossible、UIGestureRecognizerStateBegan和UIGestureRecognizerStateEnded。

当某个UIGestureRecognizer子类对象的state属性发生变化时(除了切换至UIGestureRecognizerStatePossible的情况),该对象就会向其目标对象发送指定的动作消息。所以当某个长按手势开始和结束时,相应的UILongPressGestureRecognizer对象都会向其目标对象发送同一个消息。和该消息匹配的方法可以通过UIGestureRecognizer对象的state属性来判断当前的事件类型,然后根据不同的事件类型执行不同的代码。

longPress:的实现逻辑为：当BNRDrawView对象收到longPress:时，如果事件类型是“开始”，就应该根据手势发生时的位置选中距离最近的那根线条。这样用户就可以通过长按屏幕选中某根线条。如果事件类型是“结束”，就应该取消当前选中的线条。

在BNRDrawView.m中实现longPress:，代码如下：

```
- (void)longPress:(UIGestureRecognizer *)gr
{
if (gr.state == UIGestureRecognizerStateBegan) {
CGPoint point = [gr locationInView:self];
self.selectedLine = [self lineAtPoint:point];
if (self.selectedLine) {
[self.linesInProgress removeAllObjects];
}
} else if (gr.state == UIGestureRecognizerStateEnded) {
self.selectedLine = nil;
}
[self setNeedsDisplay];
}
```

构建并运行应用。画一根线条，按住线条不放。TouchTracker应该会用绿色重画这根线条。

13.6 UIPanGestureRecognizer以及同时识别多个手势

当用户按住某根线条不放时，TouchTracker应该允许用户通过移动手指来拖曳选中的线条。这类手势称为拖动(pan)，可以用UIPanGestureRecognizer对象来识别。

通常情况下，UIGestureRecognizer对象不会将其处理过的触摸事件再交给其他对象来处理。一旦某个UIGestureRecognizer子类对象识别出了相应的手势，就会“吃掉”所有相关的触摸事件，导致其他UIGestureRecognizer对象没有机会再处理这些触摸事件。对TouchTracker，这种特性会导致BNRDrawView对象无法处理拖动手势，这是因为整个拖动手势都是在长按手势中发生的。要解决这个问题，需要让UILongPressGestureRecognizer对象和UIPanGestureRecognizer对象能够同时识别手势。

在BNRDrawView.m的类扩展中将BNRDrawView声明为遵守UIGestureRecognizer- Delegate协议。然后声明一个类型为UIPanGestureRecognizer的属性，代码如下：

```
@interface BNRDrawView () <UIGestureRecognizerDelegate>

@property (nonatomic, strong) UIPanGestureRecognizer *moveRecognizer;

@property (nonatomic, strong) NSMutableDictionary *linesInProgress;

@property (nonatomic, strong) NSMutableArray *finishedLines;

@property (nonatomic, weak) BNRLine *selectedLine;

@end
```

更新BNRDrawView.m中的initWithFrame:，创建一个UIPanGestureRecognizer对象，设置属性，然后将该对象附着在BNRDrawView对象上，代码如下：

```
[self addGestureRecognizer:pressRecognizer];

self.moveRecognizer = [[UIPanGestureRecognizer alloc] initWithTarget:self

action:@selector(moveLine:)];

self.moveRecognizer.delegate = self;

self.moveRecognizer.cancelsTouchesInView = NO;

[self addGestureRecognizer:self.moveRecognizer];
```

UIGestureRecognizerDelegate协议声明了很多方法，目前BNRDrawView只需要用到其中的一个：gestureRecognizer:shouldRecognizeSimultaneouslyWith- GestureRecognizer:。当某个UIGestureRecognizer子类对象识别出特定的手势后，如果发现其他的UIGestureRecognizer子类对象也识别出了特定的手势，就会向其委托对象发送gestureRecognizer:shouldRecognizeSimultaneouslyWithGestureRecognizer:消息。如果相应的方法

返回YES，那么当前的UIGestureRecognizer子类对象就会和其他UIGestureRecognizer子类对象共享UITouch对象。

在BNRDrawView.m中，如果消息的发送方是_moveRecognizer，就返回YES；否则返回NO，代码如下：

```
- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer

shouldRecognizeSimultaneouslyWithGestureRecognizer:(UIGestureRecognizer *)other

{

if (gestureRecognizer == self.moveRecognizer) {

return YES;

}

return NO;

}
```

完成上述修改后，当用户长按某根线条不放时，UIPanGestureRecognizer对象也能收到相关的UITouch对象，从而可以跟踪用户的手指移动。当用户的手指开始移动时，UIPanGestureRecognizer对象的状态也会切换至“开始”。如果UILongPress-GestureRecognizer对象和UIPanGestureRecognizer对象不能同时识别手势，那么当用户的手指开始在屏幕上移动时，UILongPressGestureRecognizer对象的状态还是会切换至“开始”，但是UIPanGestureRecognizer对象的状态不会发生变化，也不会向其目标对象发送动作消息。

除了之前介绍的三种状态，UIPanGestureRecognizer对象还有一种变化后（changed）状态。当手指开始移动时，UIPanGestureRecognizer对象会进入“开始”状态，并向其目标对象发送指定的动作消息。当手指在屏幕上移动时，UIPanGestureRecognizer对象的状态会切换至“变化后”状态，并持续地向其目标对象发送指定的动作消息。最后，当手指离开屏幕时，UIPanGestureRecognizer对象的状态会切换至“结束”状态，并向其目标对象最后一次发送指定的动作消息。

下面要实现UIPanGestureRecognizer对象的动作方法moveLine:。在moveLine:中，要调用UIPanGestureRecognizer对象的translationInView:方法。该方法会根据传入的UIView对象的坐标系，以CGPoint结构的形式返回手指的拖动距离。当拖动手势开始时，拖动距离是0点（x和y都是0）。拖动的过程中，拖动距离会不断发生变化（将手指移动至窗口的最右端，x的值就会很高。将手指移回手势的起始位置，拖动距离就会变回0点）。

在BNRDrawView.m中实现moveLine:。因为调用该方法的将是UIPanGestureRecognizer对象，所以可以将moveLine:的传入实参类型声明为UIPanGestureRecognizer对象，代码如下：

```
- (void)moveLine:(UIPanGestureRecognizer *)gr
```

```
{
// 如果没有选中的线条就直接返回
if（！self.selectedLine）{
return;
}
// 如果UIPanGestureRecognizer对象处于“变化后”状态
if（gr.state == UIGestureRecognizerStateChanged）{
// 获取手指的拖移距离
CGPoint translation = [gr translationInView:self];
// 将拖移距离加至选中的线条的起点和终点
CGPoint begin = self.selectedLine.begin;
CGPoint end = self.selectedLine.end;
begin.x += translation.x;
begin.y += translation.y;
end.x += translation.x;
end.y += translation.y;
// 为选中的线条设置新的起点和终点
self.selectedLine.begin = begin;
self.selectedLine.end = end;
// 重画视图
[self setNeedsDisplay];
}
}
```

构建并运行应用。画一根线条，按住这根线条不放并开始拖移。读者会发现当前选中的线条位置并不能和手指的位置保持一致。这是因为moveLine:会持续累加当前选中的线条的起点

和终点。如果UIPanGestureRecognizer对象可以增量地报告拖移距离（以上次调用moveLine:时的位置为起点），就能解决上述问题。UIPanGestureRecognizer有一个名为setTranslation:的方法，调用该方法并传入CGPointZero，就能将手指的当前位置设置为拖移手势的起始位置。因此，只需要在UIPanGestureRecognizer对象报告位置变化时，向其发送setTranslation:消息并传入CGPointZero，就能使该对象增量地报告拖移距离。

在BNRDrawView.m的moveLine:底部，加入下面这行代码。

```
[self setNeedsDisplay];

[gr setTranslation:CGPointZero inView:self];

}

}
```

构建并运行应用，画一根线条，按住这根线条不放并开始拖移。当前选中的线条位置应该会和手指的位置保持一致。

在BNRDrawView.m的initWithFrame:中，TouchTracker还设置了UIPanGesture- Recognizer对象的cancelsTouchesInView属性。该属性的默认值是YES，当某个UIGestureRecognizer对象的cancelsTouchesInView属性为YES时，这个对象会在识别出特定的手势时，“吃掉”所有和该手势有关的UITouch对象。这样，该对象所依附的UIView对象将不会收到之前介绍过的那些UIResponder消息，例如touchesBegan:withEvent:。

通常情况下，上述特性是符合开发预期的，但是也有例外。以TouchTracker为例，如果UIPanGestureRecognizer对象在识别出拖移手势时吃掉了所有相关的UITouch对象，那么BNRDrawView对象将没有机会处理这些UITouch对象，也就无法创建相应的BNRLine对象。

当某个UIGestureRecognizer对象的cancelsTouchesInView属性为NO时，这个对象所依附的UIView对象仍然会收到相应的UIResponder消息，从而有机会处理相关的UITouch对象。读者可以尝试注释掉那行设置cancelsTouchesInView属性的代码，检验效果。

13.7 深入学习：UIMenuController与UIResponderStandardEditActions

UIMenuController对象拥有一组默认的UIMenuItem对象，专门负责为用户显示“编辑”菜单。以UITextField或UITextText为例，当用户按下上述两种对象时，UIMenuController对象就会显示这类UIMenuItem对象（例如剪切、拷贝和粘贴等）。这类UIMenuItem对象都有固定的动作消息。以剪切菜单项为例，当用户选中“剪切”项后，相应的UIMenuItem对象就会向显示该菜单项的UIView对象发送cut:消息。

如果某个UIResponder子类需要实现特定的“编辑”功能，就可以实现相应的方法。以UITextField为例，为了能够剪切当前选中的文字，UITextField实现了cut:方法。UIResponderStandardEditActions协议声明了所有这类“编辑”方法。

如果某个UIView子类实现了UIResponderStandardEditActions协议中的方法，那么当这个子类对象显示UIMenuController对象时，就会出现和这些方法相对应的菜单项。这是因为该对象会在显示前枚举所有的“编辑”菜单项，然后根据其动作消息向视图发送canPerformAction:withSender:消息。如果视图实现了指定的方法，该消息就会返回YES，否则返回NO。UIMenuController对象会根据canPerformAction:withSender:的返回结果判断是否应该显示相应的菜单项。

如果读者要实现某个UIResponderStandardEditActions协议中的方法，又不希望UIMenuController对象显示相应的菜单项，可以覆盖canPerformAction:withSender:，然后针对特定的方法返回NO，代码如下：

```
- (BOOL)canPerformAction:(SEL)action withSender:(id)sender

{

if (action == @selector(copy:))

return NO;

// 父类的实现会根据目标对象是否实现了特定的动作方法，返回YES或NO

return [super canPerformAction:action withSender:sender];

}
```

13.8 深入学习：再谈UIGestureRecognizer

本章只对UIGestureRecognizer做了一个简单的介绍，关于它的属性和委托方法还没有进行介绍。此外，iOS SDK还提供了若干其他的UIGestureRecognizer子类，读者也可以创建自己的UIGestureRecognizer子类。本节将对UIGestureRecognizer再做一个概要介绍，以帮助读者了解“UIGestureRecognizer可以做什么”。

当某个UIGestureRecognizer子类对象附着在UIView对象上时，这个子类对象会自动处理和触摸事件有关的UIResponder方法（例如touchesBegan:withEvent:）。UIGestureRecognizer子类对象都是很“贪婪的”，凡是附着了该子类对象的UIView对象，通常都不会再收到和触摸事件有关的消息，或者会延迟收到这类消息。修改UIGestureRecognizer对象的某些属性（例如delaysTouchesBegan、delaysTouches- Ended和cancelsTouchesInView），可以改变这种行为。如果要进一步修改UIGestureRecognizer对象处理触摸事件的逻辑，还可以为其实现特定的委托方法。

某些情况下，一个UIView对象可能会附着多个UIGestureRecognizer子类对象，并且其中的两个UIGestureRecognizer子类对象所识别的都是类似的手势。在这种情况下，通过调用UIGestureRecognizer的requireGestureRecognizerToFail:，可以将这两个UIGestureRecognizer子类对象“串”起来：只有当某个UIGestureRecognizer子类对象识别不出手势时，另一个子类对象才会开始识别。

掌握UIGestureRecognizer的关键是根据不同的UIGestureRecognizer子类，理解其各种状态的含义。总的来说，UIGestureRecognizer子类对象可以有以下几种状态：

·UIGestureRecognizerStatePossible·UIGestureRecognizerStateFailed

·UIGestureRecognizerStateBegan·UIGestureRecognizerStateCancelled

·UIGestureRecognizerStateChanged·UIGestureRecognizerStateRecognized

·UIGestureRecognizerStateEnded

多数情况下，UIGestureRecognizer子类对象会处于“可能”（UIGestureRecognizer- StatePossible）状态。当某个UIGestureRecognizer子类对象识别出特定的手势时，其状态会切换至“开始”（UIGestureRecognizerStateBegan）状态。如果某个UIGestureRecognizer子类对象的手势是可持续的（例如拖移），其状态就会切换至“变化后”（UIGestureRecognizer- StateChanged）状态，而且会保持该状态直到手势结束。在“变化后”状态中，只要手势发生了变化，UIGestureRecognizer子类对象就会向目标对象发送指定的动作消息。当手势结束时，UIGestureRecognizer子类对象会切换至“结束”（UIGestureRecognizerStateEnded）状态。

不是所有的UIGestureRecognizer子类对象都会有“开始”“变化后”和“结束”这三种状态。对某些识别“不连续”手势（例如按下）的UIGestureRecognizer子类，就只有一个“已识别”（UIGestureRecognizerStateRecognized）状态（UIGesture- RecognizerStateRecognized和UIGestureRecognizerStateEnded的值是相同的）。

UIGestureRecognizer子类对象是可以“取消的”（例如有电话进来），也可能会失败（例如根据

当前的触摸事件不足以识别特定的手势）。当某个UIGestureRecognizer子类对象的状态发生变化时，该对象会向目标对象发送指定的动作消息。相应的动作方法可以根据UIGestureRecognizer子类对象的state属性执行不同的逻辑。

iOS SDK提供了多个UIGestureRecognizer子类，如UIPinchGestureRecognizer、UISwipeGestureRecognizer和UIRotationGestureRecognizer。这些子类对象都有自己的额外属性，可以用来调整其识别行为，具体的细节请读者参考相关文档。

如果iOS SDK提供的UIGestureRecognizer子类不能满足需要，读者还可以创建自己的UIGestureRecognizer子类。具体的细节请读者参考UIGestureRecognizer文档的“Subclassing Notes”（如何创建UIGestureRecognizer子类）部分。

13.9 中级练习：修正错误

TouchTracker其实有一个Bug：当用户按下并选中某根线条后，TouchTracker会显示相应的菜单。如果在没有关闭该菜单的情况下就开始画新的线条，TouchTracker会在画新线条的同时拖移之前选中的线条。试修正这个错误。

13.10 高级练习：速度与宽度

试通过UIPanGestureRecognizer对象记录画线时的速度，然后根据这个速度修改线条的宽度。不要假设画线速度的最小值和最大值，而应该先向控制台输出速度信息，获取可靠的数据。

13.11 高级练习：颜色

为TouchTracker增加三根手指的向上滑动手势（swipe）。当TouchTracker识别出该手势后，需要显示一个能够设置颜色的面板。一旦用户选择了某种颜色，之后就要用这种颜色画出所有线条。显示颜色面板时，TouchTracker不应该创建额外的线条。如果无法做到这点，那么至少应该在识别出三根手指的滑动手势后，删除不应该创建的BNRLine对象。

第14章 调试工具

第7章介绍过如何使用调试器找到并修正代码中的问题。本章继续介绍若干由iOS SDK提供的调试工具，指导读者如何在开发过程中使用这些工具。

14.1 仪表

Xcode 5引入了调试仪表(debug gauges)，通过仪表可以直观地看出应用的CPU和内存占用量。

在iOS设备中构建并运行TouchTracker(分析应用性能时应该使用真实设备，用户不会在模拟器上打开应用)，然后在导航面板选择条中点击，打开调试导航面板(debug navigator)，如图14-1所示。

图14-1 仪表

当应用处于运行状态时，调试导航面板会以柱状图显示CPU和内存占用量，并随着应用运行实时更新占用量数据。请读者注意，调试导航面板中的数据是根据运行应用的硬件计算出来的，苹果电脑的CPU比iOS设备更快，如果在iOS模拟器上运行应用，调试导航面板中显示的CPU占用量可能会非常低，不利于分析性能问题。

点击CPU可以在编辑区域打开CPU占用量报告(CPU Report)，如图14-2所示。

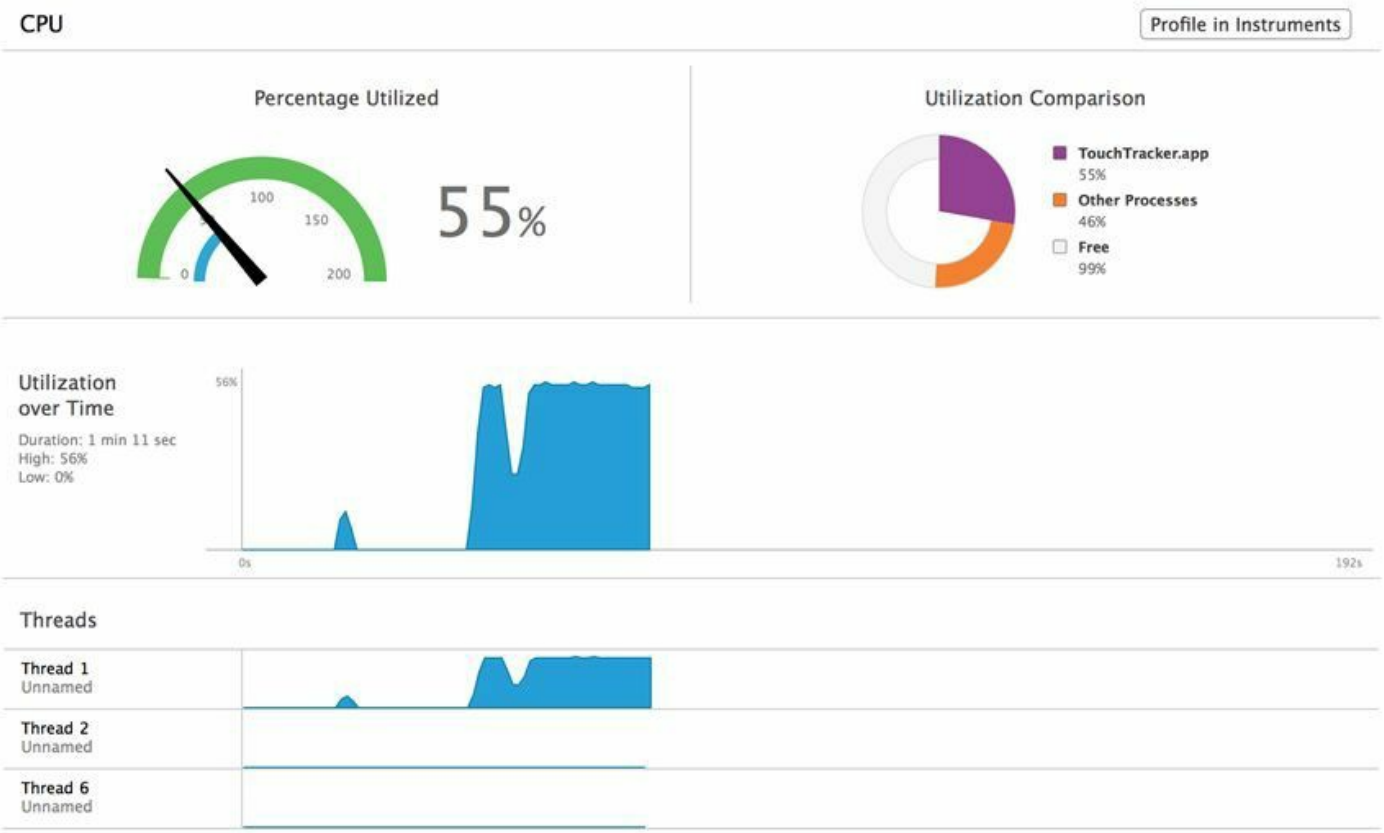


图14-2 CPU占用量报告

占用量报告包括以下四部分：

•占用百分比（Percentage Utilized）

根据运行应用设备的CPU核心数显示CPU占用比。例如，单核设备中最大CPU占用量是100%，而双核设备中是200%，所以应用在单核设备中的CPU占用量通常比双核设备更高。如果应用处于空闲状态（进入后台或没有操作），CPU占用量会显示0%。

•占用环比（Utilization Comparison）

使用环形图比较应用与系统中其他进程（Process）的CPU占用百分比。如果应用的CPU占用百分比不高，但是运行不流畅，那么可以从该环形图中查看Other Processes的占用百分比。如果设备中后台运行的其他应用太多，就可能导致Other Processes的占用百分比很高，应用运行速度变慢。

•时间-占用比（Utilization over Time）

使用折线图显示应用CPU占用量随运行时间变化的过程，以及占用量在运行时间内的峰值和谷值。

•线程（Threads）

显示应用中每一个线程的时间-占用比。本书不会深入介绍多线程编程，如果读者遇到了多线程问题，可以查看各个线程的CPU占用比，并针对异常线程着重分析。

接下来用手指在TouchTracker中绘制线条，手指不要离开屏幕，持续绘制大量线条，可以发现TouchTracker的CPU占用比迅速升高——手指在屏幕上移动时，BNRDrawView会不断收到touchesMoved:withEvent:消息并调用drawRect:重绘自己。绘制的线条越多，drawRect:的工作量就越大，CPU占用比就越高。

回到调试导航面板，点击Memory，打开内存占用量报告（Memory Report），如图14-3所示。

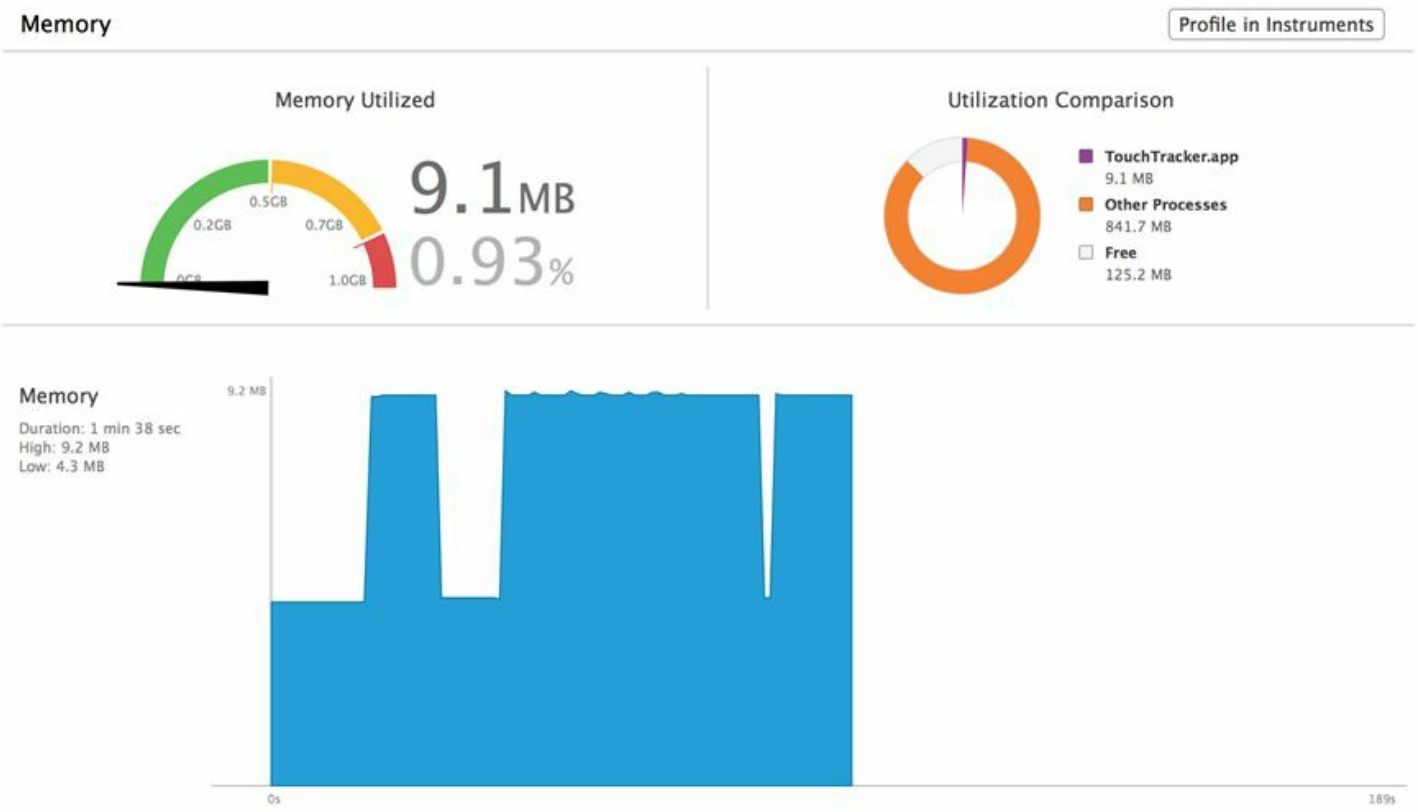


图14-3 内存占用量报告

和CPU占用量报告类似，内存占用量报告直观地显示了应用内存占用的情况，但是Memory部分（位于最下方）显示的折线图可能会让读者产生误解：应用似乎始终占用了100%的内存。实际上，图中将运行时间内的占用峰值作为100%显示，因此图中y轴的最大值与左侧High栏中显示的数值相同，都是9.2 MB。

无论开发何种类型的应用，都应该尽可能降低CPU和内存占用量，提高应用的流畅度和用户体验。建议读者在编写代码的过程中随时观察调试仪表，尽早发现应用的性能问题，如果发现异常情况，需要及时查找代码中的问题，尝试编写性能更好的代码。

14.2 Instruments

从仪表和占用量报告中可以简要分析出应用的性能，但是，如果应用的CPU和内存占用量过高，需要从代码中查找性能问题，则可以使用Instruments，它提供了比仪表和占用量报告更详细的数据分析。

Instruments是一种与Xcode紧密集成的调试工具，可以实时监控并统计应用运行时的各项数据，生成详细的分析报告。它由若干组件组成，这些组件检查的事项包括：应用创建了哪些对象、每一个方法和函数的CPU占用量、文件I/O和网络I/O等。通过使用这些不同的组件，可以找出程序中的性能瓶颈，发现代码中的问题。

Allocations组件

Allocations组件可以列出应用创建过的全部对象，以及这些对象所占用的内存大小。

当监视某个应用时，Allocations组件会对这个应用进行性能分析（profiling）。虽然可以在模拟器上对某个应用进行性能分析，但是在真实的设备上进行可以得到更精确的数据。

要对当前打开的项目执行性能分析，可以按住位于工作空间左上角的Run按钮不放，然后在新出现的弹出窗口中选择Profile（见图14-4）。

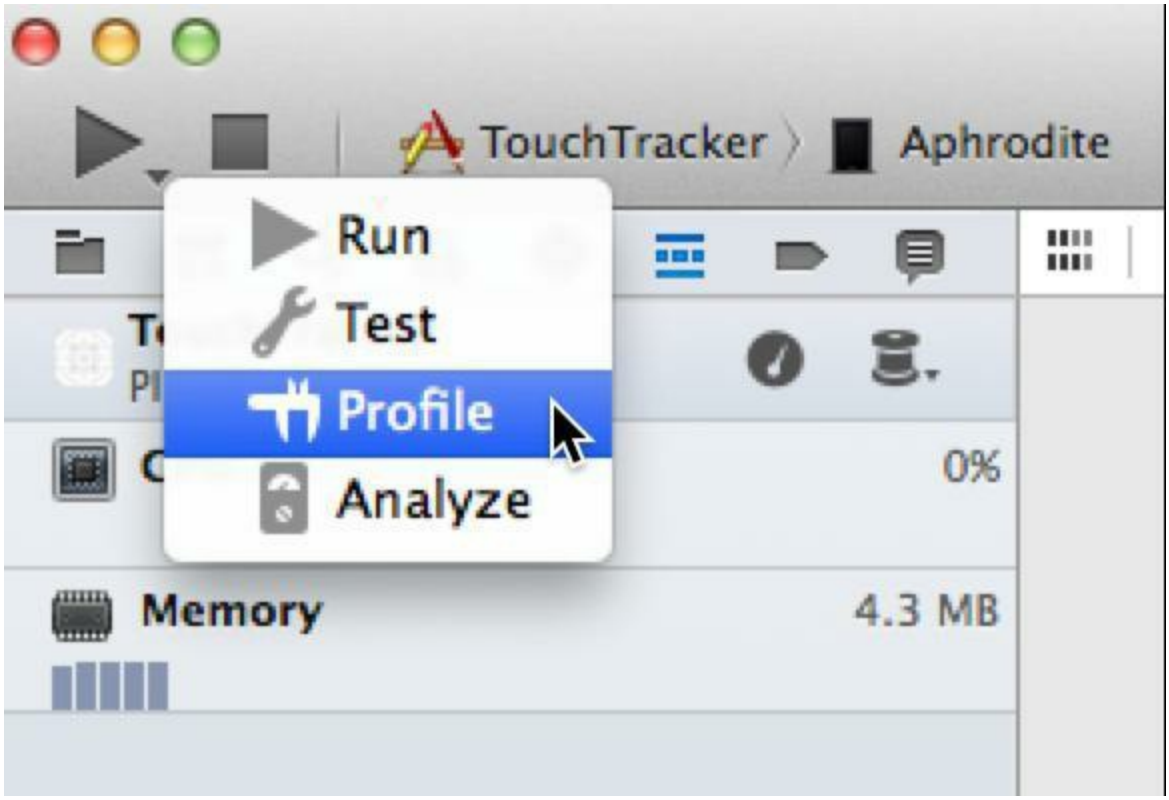


图14-4 执行性能分析

Xcode会启动Instruments。Instruments会显示一个下拉窗口并列出所有可用的组件（图中只显示了8个组件，读者向下滚动可以看见其余组件）。选择Allocations并单击Profile按钮（见图14-

5)。

图14-5 选择组件

单击Profile按钮后，Instruments会启动TouchTracker并打开Instruments的主窗口（见图14-6）。当读者第一次看到Instruments的主窗口界面时，可能会感觉比较复杂。但是和Xcode的工作空间一样，多使用几次便可熟悉。首先，打开主窗口的全部区域，确保可以看到所有的信息：找到位于窗口顶部的View控件，按下全部三个按钮，打开相应的三个主区域（见图14-6）。

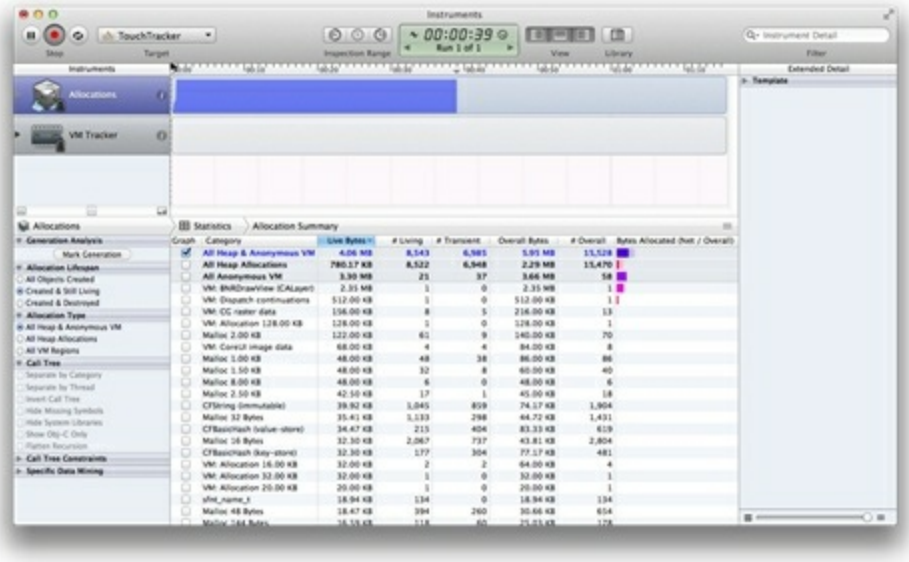


图14-6 Allocations组件

Allocations组件会显示一张表格，列出应用执行过的所有内存分配。因为数据很多，所以要先过滤，只列出由我们自己编写的代码创建的对象。首先，在BNRDrawView对象上画若干根线条。然后，在窗口右上角的Category查询框中输入BNRLine。

Allocations组件会过滤Object Summary表格所显示的条目，只列出和BNRLine有关的内存分配，即已创建的BNRLine对象（见图14-7）。

Graph	Category	Live Bytes	# Living	# Transient	Overall Bytes	# Overall	Bytes Allocated (Net / Overall)
☐	BNRLine	128 Bytes	4	0	128 Bytes	4	

图14-7 已创建的BNRLine对象

Living列会显示某种对象的现存个数。Live Bytes列会显示这些现存对象占用了多少内存。# Overall列会显示应用运行至今共创建了多少个某种类型的对象（其中包括已经释放的）。

根据Allocations组件列出的表格可知，现存的BNRLine对象个数和总共创建的BNRLine对象个数暂时是相同的。连按屏幕，清除所有的线条。这时Allocations组件的表格不会再显示任何BNRLine对象。这是因为Allocations组件默认只会显示现存的对象。如果要显示所有创建过的对象，则可以找到位于左侧面板的Allocation Lifespan区域，然后勾选All Objects Created（见图14-8）。

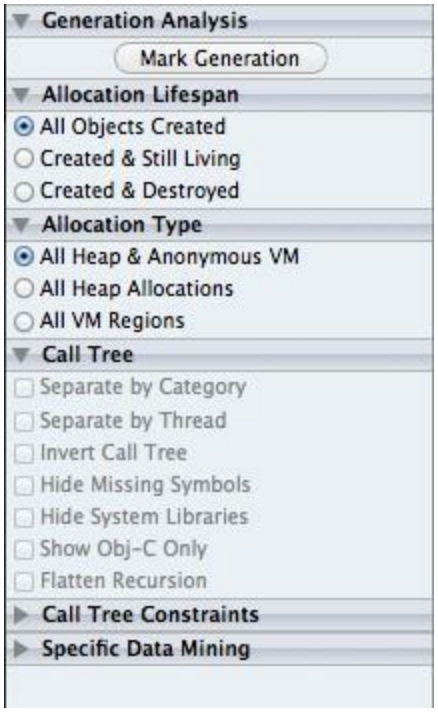


图14-8 Allocations组件选项

在Allocations组件列出的表格中，选中某个BNRLine表格项。该组件会在该表格项的Category列中显示一个箭头按钮。单击这个按钮，该组件会显示该项内存分配的详细信息（见图14-9）。

Statistics › Allocation Summary › BNRLine

#	Address	Category	Timestamp	Live	Size	Responsible Library	Responsible Caller
0	0x145ec5c0	BNRLine	02:57.539.569	•	32 Bytes	TouchTracker	-[BNRDrawView touchesBe...
1	0x145cf0c0	BNRLine	02:58.552.178	•	32 Bytes	TouchTracker	-[BNRDrawView touchesBe...
2	0x1468a070	BNRLine	02:59.152.489	•	32 Bytes	TouchTracker	-[BNRDrawView touchesBe...
3	0x145cf0a0	BNRLine	03:00.954.103	•	32 Bytes	TouchTracker	-[BNRDrawView touchesBe...

图14-9 BNRLine对象内存分配一览

表格中的每一行都代表应用创建过的一个BNRLine对象。选中其中的某一行，Allocations组件会在Extended Detail区域（扩展详细区域）显示相应的栈跟踪（stack trace）信息。扩展详细区

域位于Instruments窗口的右侧(见图14-10)。通过栈跟踪信息，可以知道当前选中的BNRLine对象是由哪一行代码创建的。栈跟踪信息中的灰色条目源自系统库的调用，黑色条目源自我们自己编写的代码。找到位置最靠近表格顶部的源自我们自己的代码(-[BNRDrawView touchesBegan:withEvent:])，然后双击相应的条目。

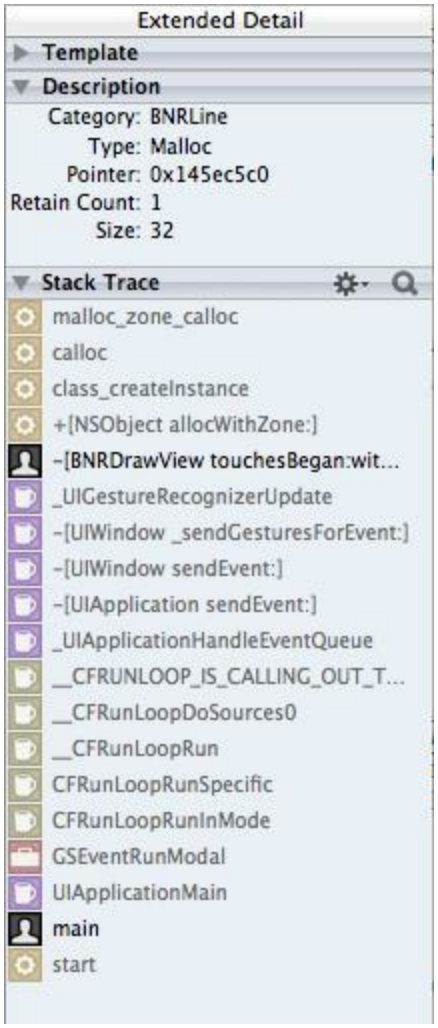


图14-10 栈跟踪信息

双击栈跟踪信息中的某项条目后，Allocations组件会显示相应的代码，并隐藏之前显示BNRLine对象的表格(见图14-11)。Allocations组件会在若干行代码的右侧显示一个百分比数字，这个数字代表某行代码所分配的内存大小占当前方法所分配的内存总量的比例。例如，图14-11中的0.2%代表新创建的BNRLine对象占整个touchesBegan:withEvent:所分配内存的比例。

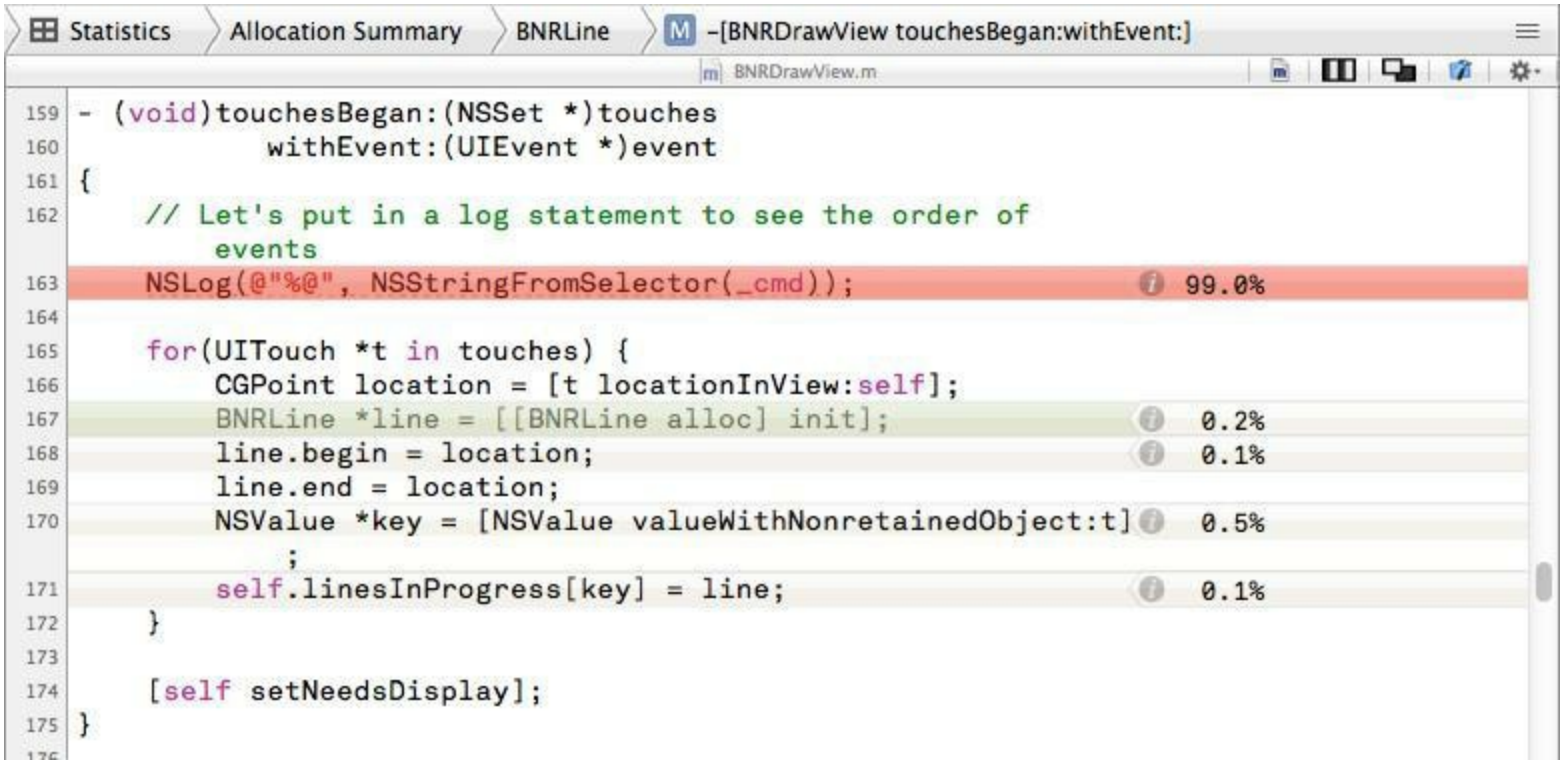


图14-11 Instruments组件所显示的源代码

Allocations组件会在代码区域上方显示一个导航条，列出之前查看过的窗口记录（见图14-12）。单击代表某个窗口的按钮，Allocations组件会再次显示相应的窗口。

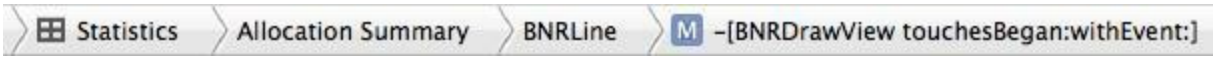


图14-12 位于代码区域上方的导航条

单击导航条中的BNRLine按钮，Allocations组件会重新显示过滤后的对象列表（只显示BNRLine对象）。单击表格中的某个BNRLine对象，然后单击该行的箭头按钮。Allocations组件会显示选中的BNRLine对象的“生存记录”，其中包括两个“事件”，即TouchTracker创建这个BNRLine对象的时刻及TouchTracker释放这个BNRLine对象的时刻。选中某个事件项，Allocations组件会在扩展详细区域显示相应的栈跟踪信息。

Generation Analysis

接下来介绍Allocations组件的Generation Analysis（阶段分析，也称为Heapshot Analysis）功能。首先清空查询框，不过滤任何结果。然后找到位于Instruments窗口左侧的Generation Analysis区域，单击Mark Generation（划分阶段），Instruments会在表格中显示Generation A（阶段A）；单击Generation A旁边的小三角按钮，Instrument会展开Generation A，列出该阶段发生的所有内存分配。再画若干线条，单击Mark Generation，Instruments会在表格中显示另一个阶段Generation B（阶段B）；单击Generation B边上的三角按钮（见图14-13）。

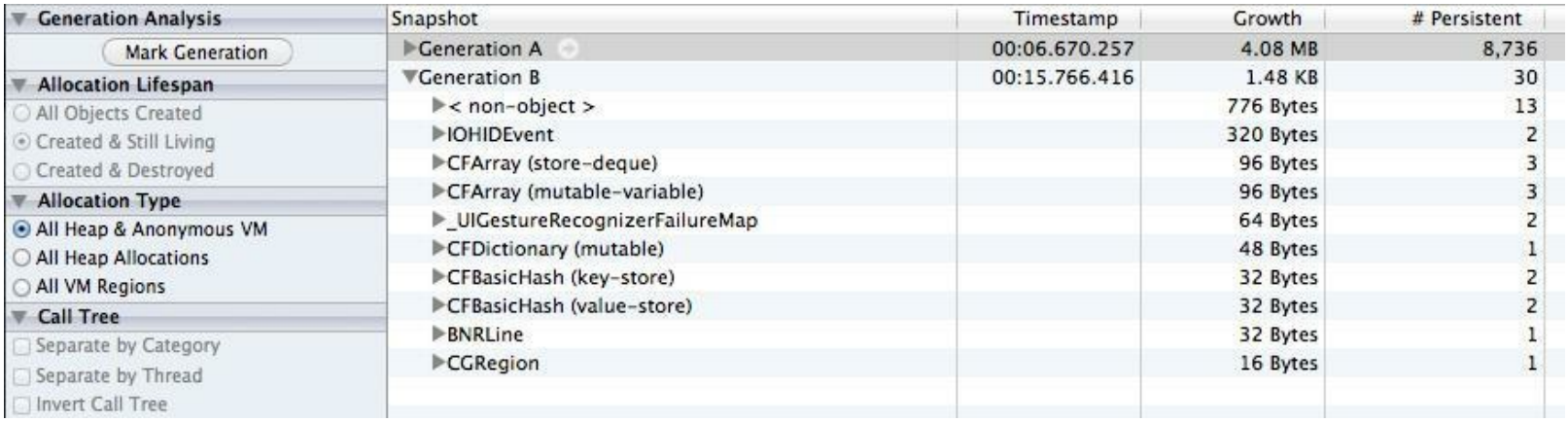


图14-13 Generation Analysis

Generation B会列出Generation A后发生的所有内存分配，其中包括刚创建的BNRLine对象，以及当时为了处理其他任务而创建的若干对象。Allocations组件没有限制可以添加的Generation数量。通过使用Generation，可以很方便地找出应用针对某个特定事件而创建的对象。在TouchTracker中连按屏幕，清除线条，Generation B中的相关对象应该会消失。

要让Instruments重新显示Object Summary表格，可以先找到表格上方的导航条，单击标题为Generations的按钮，然后在弹出菜单中选择Statistics。

Generation Analysis还可以用来跟踪内存占用趋势，查找内存泄漏问题。只要重复应用活动周期，并将每一个活动周期划分为一个阶段（Mark Generation），就可以监控应用中内存分配和回收的过程。例如，在TouchTracker中，首先用手指绘制几根线条，然后双击屏幕清除所有线条，这时点击Mark Generation按钮；继续绘制几根线条，双击清除，再次点击Mark Generation按钮……请读者重复以上操作并观察结果。

理想情况下，两个阶段之间所有分配的内存都应该被回收，因为第一阶段创建的对象在第二阶段开始时会被释放，但实际上两个阶段之间还存在系统框架创建的对象——读者只需要关注自己创建的对象。对于TouchTracker来说，如果两个阶段之间存在未被释放的BNRLine对象，说明代码中存在内存泄漏。

Time Profiler组件

Time Profiler组件提供了应用运行时的详细CPU占用量统计数据。为了介绍该组件，需要在BNRDrawView.m中的drawRect:末尾处加入以下代码，大幅提高CPU占用量：

```
float f = 0.0;

for（int i = 0; i < 1000000; i++）{

f = f + sin(sin(sin(time(NULL) + i)));

}
```

NSLog(@"f = %f", f);

构建应用并执行性能分析。当Instruments列出所有可选的组件时，选择Time Profiler（见图14-14）。等Instruments启动完TouchTracker并显示主窗口后，按下View控件中的全部按钮（按下后的按钮会显示为蓝色），打开所有的三个区域。

图14-14 Time Profiler组件

在TouchTracker中用手指在屏幕上滑动，这样BNRDrawView会不断收到touchesMoved:withEvent:消息并调用drawRect:重绘自己，导致无用的sin函数反复执行。

这时Time Profiler只显示了应用中各个线程的消耗时间，接下来需要具体了解各个方法和函数的执行时间。单击暂停按钮（位于Stop按钮左侧），然后在左侧面板中勾选标题为Invert Call Tree的选择框。现在查看表格中的内容，表格中的每一行都代表一个函数调用或方法调用。表格左列（Running Time）显示的是应用执行该函数所花费的时间（以毫秒为单位，还会显示占用全部运行时间的百分比，见图14-15）。通过这个表格，读者可以大致了解应用是如何分配执行时间的。

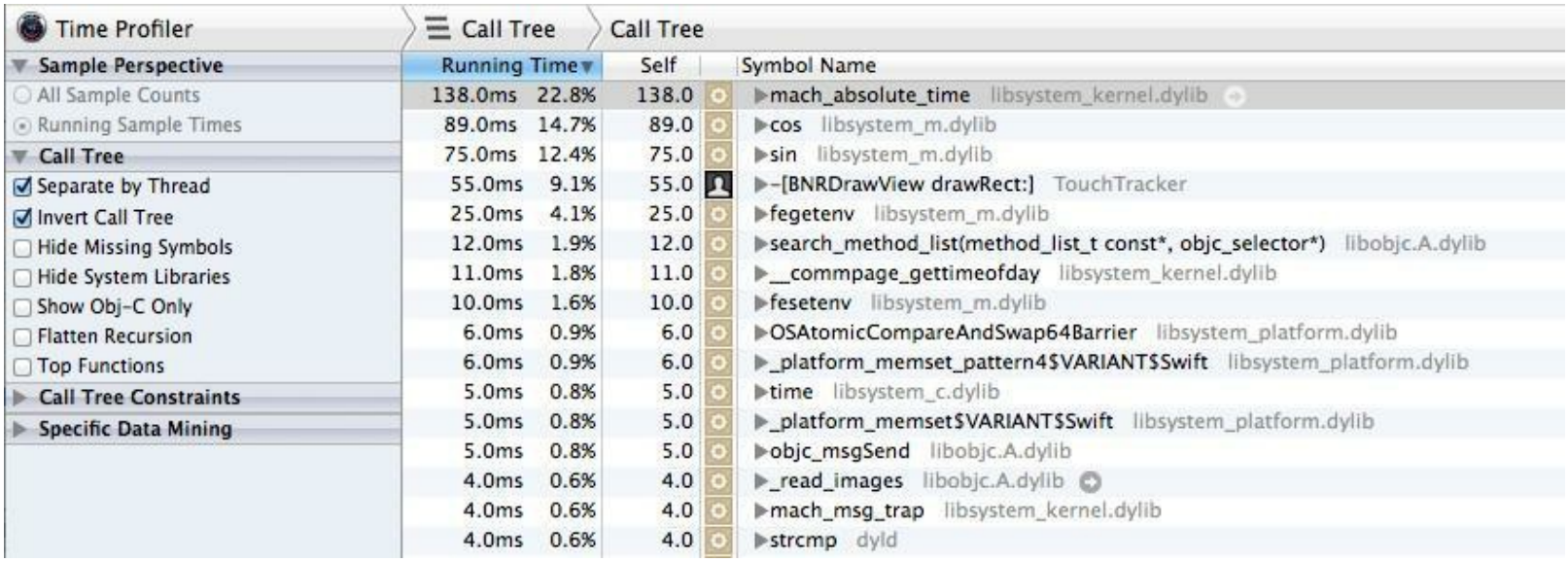


图14-15 Time Profiler组件显示的统计结果

要判断某个应用是否有效率问题，并没有一成不变的硬性规则，不能认为“如果应用在某个特定的函数上消耗了百分之X的CPU时间，就一定有问题”。正确的做法是从用户的角度进行测试，如果发现应用反应迟缓，就用Time Profiler来查找问题。以TouchTracker为例，为BNRDrawView加入无用的sin函数后，读者应该会在画线时察觉到反应迟缓的现象。

在TouchTracker中画线时，BNRDrawView对象会收到touchesMoved:withEvent:消息和drawRect:消息。通过Time Profiler组件，可以查看应用在这两个方法上花了多少时间，并能和其他的方法进行比较。如果应用在某个方法上花费了过多的时间，那么这个方法就可能有问题。

需要注意的是，有些任务本身就是很消耗时间的。以TouchTracker为例，用户每次移动手指就要刷新整个屏幕，这本身就是很消耗时间的操作。如果因此影响了用户体验，就应该想办法减少刷新屏幕的次数。例如，无论收到多少次触摸事件，都每隔1/10秒刷新一次屏幕。

Time Profiler组件会显示应用调用过的几乎所有的函数和方法。通过筛选显示结果，可以要求该组件只列出特定部分的代码。例如，mach_msg_trap()函数有时会出现在表格的顶部。这是因为主线程会在等待输入时调用该函数。因为应用将大量的时间花在mach_msg_trap()函数上是没有问题的，所以可以要求Time Profiler将这部分时间剔除出统计结果。

找到位于Instruments窗口右上角的查询框，输入mach_msg_trap()，然后选中表格中的mach_msg_trap()条目。找到位于窗口左侧的Specific Data Mining区域，单击下方的Symbol按钮。该区域中的表格会显示mach_msg_trap()函数，并在相应的表格项右侧显示一个标题为Charge的弹出按钮。单击Charge并将其改为Prune。清空搜索框中的文字，Time Profiler组件会更新表格并忽略mach_msg_trap()所消耗的时间（见图14-16）。要将mach_msg_trap()所消耗的时间加回总时间，可以选中Specific Data Mining表格中的mach_msg_trap()，然后单击Restore按钮。

图14-16 忽略mach_msg_trap所消耗的时间

除了忽略指定函数或方法所消耗的时间，还可以通过其他途径过滤Time Profiler组件的统计结果。这些途径包括：只显示Objective-C调用（showing only Objective-C calls）、隐藏系统库调用（hiding system libraries）和将调用时间计入调用方（charging calls to callers）。前两种途径很容易理解，下面详细介绍“将调用时间计入调用方”。选中统计结果中的mach_ absolute_time()（或者其他以mach_absolute_time开头的方法），然后单击Symbol按钮。Time Profiler会将该函数移出统计结果，然后将其加入Specific Data Mining表格并设置为Charge（计算调用时间）状态，这意味着应用花在该函数上的时间都会被计入相应的调用方。

这时，Time Profiler会将统计结果中的mach_absolute_time()替换成调用方gettimeofday()。如

果重复上述步骤，再计算(charge)gettimeofday()，则Time Profiler会将gettimeofday()替换成调用方time()。继续计算time()，Time Profiler会将time()替换成调用方drawRect:。因为Time Profiler组件会将time()、gettimeofday()和mach_absolute_time()的调用时间都计入drawRect:，所以drawRect:会出现在统计表格的顶部。

某些常规函数会占用大量的CPU时间。多数情况下，这些函数调用都是正常的，也是无法避免的。以objc_msgSend()函数为例，它是Objective-C消息发送机制的主要派发函数。当某个应用在向对象发送大量消息时，objc_msgSend()可能会出现在CPU统计列表的顶部，一般情况下这是正常的。但是，如果应用花在消息派发上的时间多于相应消息所触发的方法的实际工作时间，而且应用的性能不佳，就有问题。

举个实际的例子。如果将向量、点和矩形都封装成类，就需要为这些类实现方法，用来加上、减去和乘以其他同类对象。此外还要实现存取方法，用于获取和设置实例变量。当应用通过这些对象执行绘图任务时，即使是很简单的任务(例如创建两个向量并相加)，也要向对象发送大量的消息。对这种情况，更好的解决方案是用C结构来描述这类数据类型，这样应用就可以直接存取内存(这也是为什么CGRect和CGPoint是C结构，而不是Objective-C类)。

最后，在drawRect:中删除之前添加的提高CPU占用量的代码。

Leaks组件

本节介绍Instruments的另一个很有用的组件：Leaks组件。虽然ARC降低了应用发生内存泄露的可能，但无法解决强引用循环问题。Leaks组件能帮助读者找出应用中的强引用循环问题。

首先，故意制造一处强引用循环，在BNRLine对象中添加一个属性，指向包含BNRLine对象的数组对象。在BNRLine.h中声明一个新属性，代码如下：

```
@property (nonatomic, strong) NSMutableArray *containingArray;
```

更新BNRDrawView.m中的touchesEnded:withEvent:，为每个BNRLine对象设置containingArray：

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

{

// 从字典中删除UITouch对象

for (UITouch *t in touches) {

NSValue *key = [NSValue valueWithNonretainedObject:t];

BNRLine *line = self.linesInProgress[key];
```

```
    [self.finishedLines addObject:line];

    [self.linesInProgress removeObjectForKey:key];

    line.containingArray = self.finishedLines;

    }

    // 重新绘制

    [self setNeedsDisplay];

    }
```

最后更新BNRDrawView.m中的doubleTap:方法，注释掉清空self.finishedLines的那行代码，改为创建一个新的NSMutableArray对象。

```
    - (void)doubleTap:(UIGestureRecognizer *)gr

    {

    NSLog(@"Recognized Double Tap");

    [self.linesInProgress removeAllObjects];

    // [self.finishedLines removeAllObjects];

    self.finishedLines = [[NSMutableArray alloc] init];

    [self setNeedsDisplay];

    }
```

构建应用并执行性能分析。当Instruments列出所有可选的组件时，选择Leaks。

先画出若干根线条，然后连按屏幕，清除所有的线条。选中位于窗口左侧上方的Leaks表格项，稍等几秒，Leaks组件就会在统计表格中显示三个表格项，分别代表一个NSMutableArray对象、若干BNRLine对象和一个大小为16个字节的内存块（Malloc 16 Bytes）。这些表格项代表的都是内存泄露。

找到表格上方的导航条，单击标题为Leaks的按钮，然后在弹出菜单中选择Cycles & Roots（见图14-17）。这时Leaks组件会以图形的形式显示强引用循环：一个NSMutableArray对象（self.finishedLines）拥有其包含的每个BNRLine对象，每个BNRLine对象也都有一个指回该对象的引用。

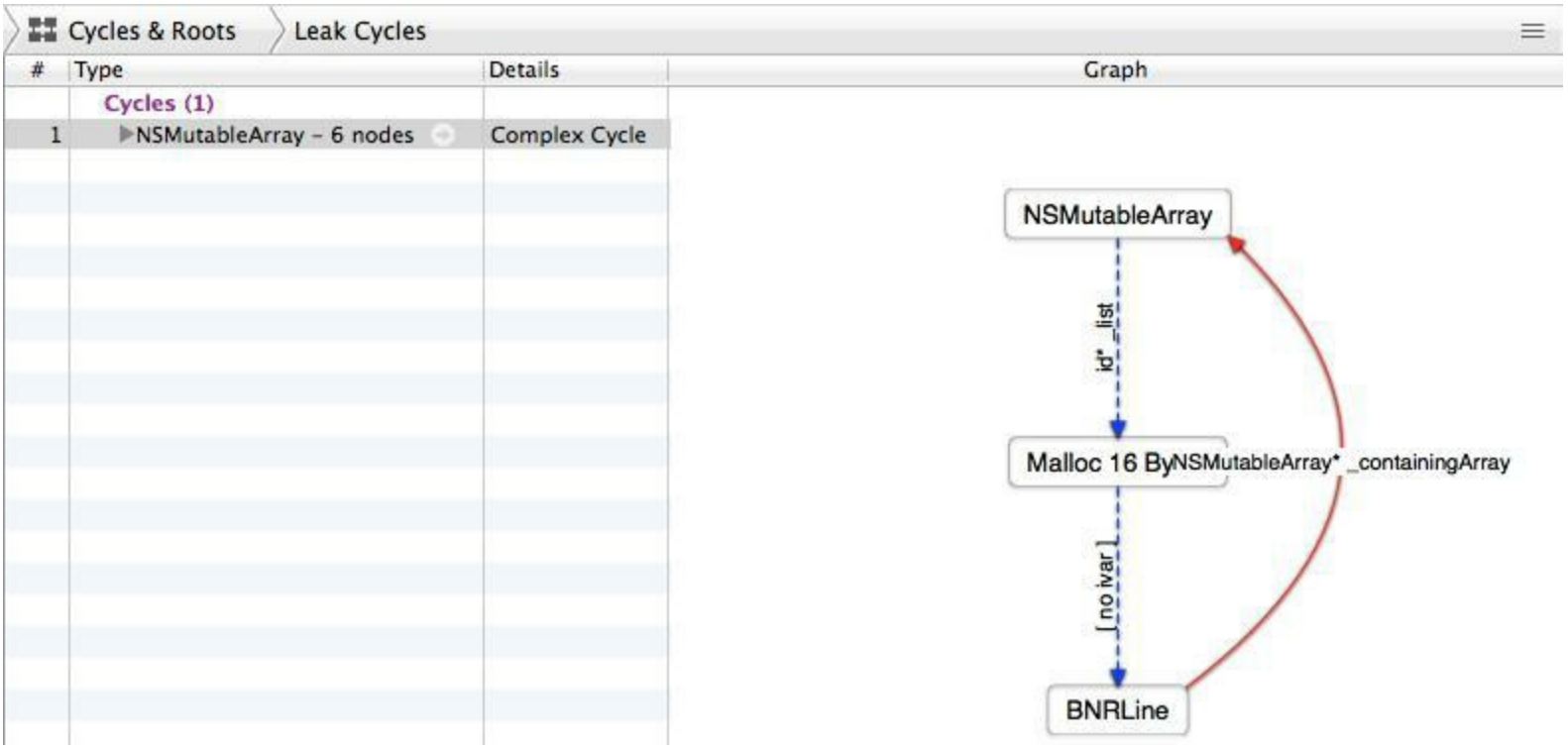


图14-17 图形化的强引用循环对象图

要修正上述强引用循环问题，只需要将containingArray属性声明为弱引用，或者删除之前添加的代码，还原touchesEnded:withEvent:和doubleTap:。

通过上述介绍，读者应该对Instruments有一个大致的了解。熟能生巧，以后还要靠读者自己多使用、多尝试。假设读者打算在Instruments上花费大量的开发时间，请注意：如果应用没有性能问题，就不要太在意Instruments的输出结果。Instruments是诊断工具，是用来诊断现有问题的，而不是找出新的问题。先写出干净的能够工作的代码，当应用的性能不佳时，再通过Instruments查找并修正问题。

14.3 静态分析器

Instruments可以在应用运行时发现问题，而Xcode的静态分析器不用运行应用就可以分析(analyze)代码。静态分析器可以根据预置的经验数据猜测执行代码后可能发生的状况，并报告潜在的问题。

当检查代码时，静态分析器会枚举每条可能的代码路径(code path)，并分开检查所有的函数和方法。一段代码可以有很多控制语句(if、for、switch等)，这些语句的输入条件决定应用会执行哪部分的代码。代码路径是指在控制语句的作用下，代码可能执行的流程。例如，拥有一条if语句的方法就有两条代码路径：条件失败是一条，条件成功是另一条。

目前静态分析器检测TouchTracker时不会报告任何问题。下面在BNRDrawView.m中加入一段有问题的代码：

```
- (int)numberOfLines

{

int count;

// 计数前先查看相应的指针是否为nil

if (self.linesInProgress && self.finishedLines)

count = [self.linesInProgress count] + [self.finishedLines count];

return count;

}
```

运行静态分析器的方法和分析应用相同：按住工作空间左上角的Run按钮不放，这次在弹出窗口中选择Analyze。此外，也可以使用键盘快捷键Command-Shift-B运行静态分析器。

Xcode会在问题面板中显示分析结果。分析TouchTracker后，静态分析器会报告在numberOfLines方法的返回点报告一处逻辑错误(Logic error)。静态分析器认为numberOfLines方法中的某条代码路径会导致该方法返回未定义的值或垃圾值。也就是说，该方法可能会返回尚未赋值的count(见图14-18)。

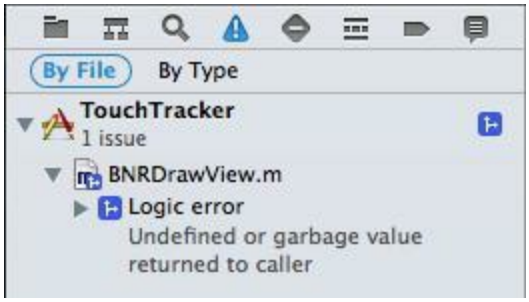


图14-18 静态分析结果

静态分析器还可以显示出上述结果的分析过程。单击位于分析结果左侧的小三角按钮，展开详细信息。单击位于小三角按钮下方的条目，Xcode会在编辑器区域显示对应的代码。此外，Xcode还会用箭头指明产生问题的那条代码路径（见图14-19）。如果Xcode没有在代码左侧的空白区域显示行号，可以通过以下途径打开相应的设置：打开预置窗口（选择Xcode菜单中的Preferences菜单项），选中Text Editing标签并勾上Show Line Numbers（显示行号）。

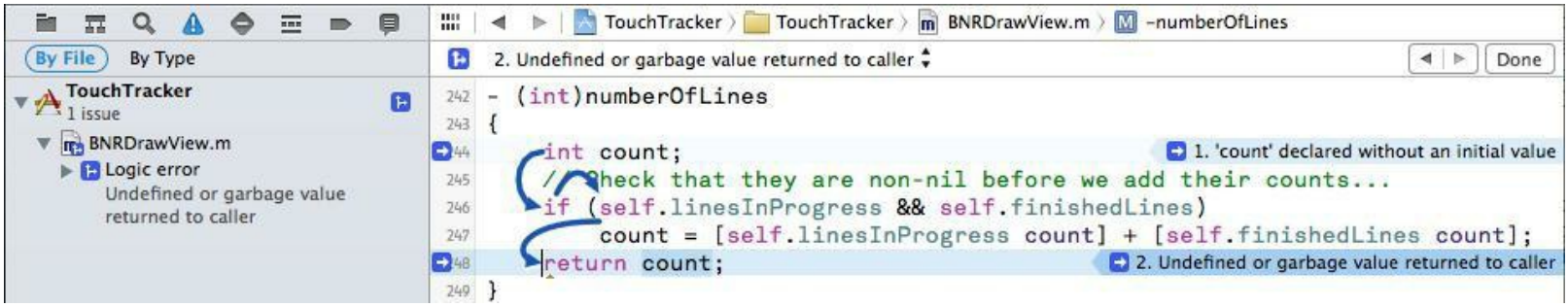


图14-19 展开后的分析结果

针对上述逻辑错误，静态分析器显示的代码路径如下。

（1）定义了count变量但是没有为其赋值。

（2）if语句的条件判断失败，导致count没有得到赋值。

（3）numberOfLines方法没有为count赋值就返回了该变量。

要解决上述问题，可以在定义count时就将其赋为0。

```
- (int)numberOfLines

{

~~int count;~~

int count = 0;

// 计数前先查看相应的指针是否为nil

if (linesInProcess && completeLines)

count = [linesInProcess count] + [completeLines count];

return count;

}
```

再使用静态分析器分析TouchTracker，Xcode应该不会再报告任何问题。

静态分析器能够找出多种类型的问题（经常分析代码是个好习惯）。读者不应该回避静态分析器报告的错误提示。相反，花些时间展开分析报告并弄清楚具体的原因，会对应用的开发及

提高读者的编程能力有帮助。

14.4 项目、目标和构建设置

Xcode项目中都包含扩展名为.xcodeproj的文件，称为Xcode项目文件。文件中包含了项目设置、对项目中其他文件的引用（例如源代码、图片、框架、库等）和文件组的排列方式。TouchTracker的项目文件是TouchTracker.xcodeproj。

每个Xcode项目又具有一个或多个目标（target）。构建并运行应用时，实际上并不是运行了Xcode项目，而是运行了一个目标。目标会根据项目中的文件和项目设置构建并生成一个特定的产品（product），最常见的产品就是应用，但也可能是库或者单元测试集（unit test bundle）。

创建新项目时，Xcode会根据读者选择的项目模板自动创建一个目标。例如，创建TouchTracker时，读者选择的是iOS应用模板（iOS application template），因此Xcode会创建名为TouchTracker的iOS应用目标（iOS application target）。

要查看目标的详细信息，可以单击项目导航面板顶部的TouchTracker项目文件，然后在编辑器区域找到用于显示/隐藏项目和目标列表的开关按钮，点击按钮显示TouchTracker的项目和目标列表，如图14-20所示。

图14-20 TouchTracker的项目和目标列表

所有的目标都会包含构建设置（build settings）。构建设置的作用是描述构建目标的方式。项目也会包含构建设置，项目的构建设置将成为其下所有目标的默认构建设置。下面介绍TouchTracker的项目构建设置。在项目和目标列表中选择TouchTracker项目，然后单击编辑器区域顶部的Build Settings标签（见图14-21）。Xcode会显示项目一级的构建设置，项目的目标会将这些设置作为默认值继承下来。通过编辑器区域右上角的搜索框，可以查找特定的设置。在查询框中输入Base SDK，Xcode会过滤显示内容并列出匹配的设置（Base SDK的作用是指定构建应用时使用的iOS SDK版本，请读者选择最新版本的iOS SDK）。

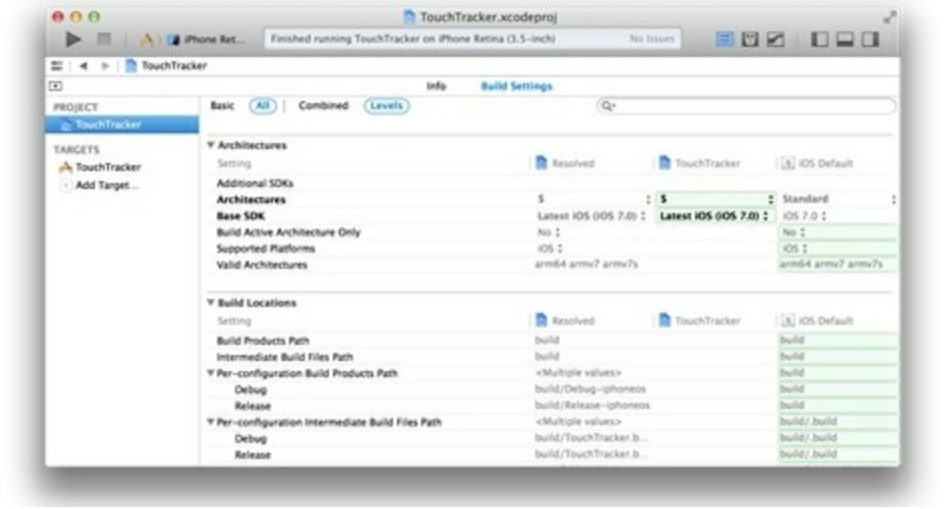


图14-21 TouchTracker的项目构建设置

在项目和目标列表中，先选择TouchTracker目标，然后选择Build Settings标签。Xcode会显示目标一级的构建设置，在设置列表上方找到并单击Levels（见图14-22）。

图14-22 TouchTracker的目标构建设置

Levels模式可以同时显示三种不同等级下的构建设置，这三种等级分别为：OS、project（项目）和target（目标）。最右侧的表格列会显示iOS Default（iOS默认）设置，即项目一级的默认设置，可以在项目一级的设置中覆盖。位于iOS Default左侧的列显示的是项目一级的设置，再往左显示的是当前目标的设置。Resolved列显示的是实际使用的设置，通常与位于表格最左侧的指定设置相同。单击不同等级中的表格单元，可以为相应等级的设置赋值。

读者还可以尝试搜索static analyzer。搜索结果中的第一个设置选项是Analyze During 'Build'。该选项默认为No，如果将其设置为Yes，Xcode会在构建应用时启动静态分析器。静态分析会增加项目的构建时间，但同时也可以帮助读者及时发现代码中的问题。

构建配置

目标和项目还可以包含多个构建配置（build configurations），构建配置的作用是包含一组指定的构建设置。创建项目时，Xcode会创建两个构建配置：debug和release。对应debug配置的构

建设置可以帮助调试应用，而对应release配置的构建设置会打开优化选项，加快应用的执行速度。

下面查看TouchTracker的构建配置。选中TouchTracker项目，然后选中顶部的Info标签（见图14-23）。

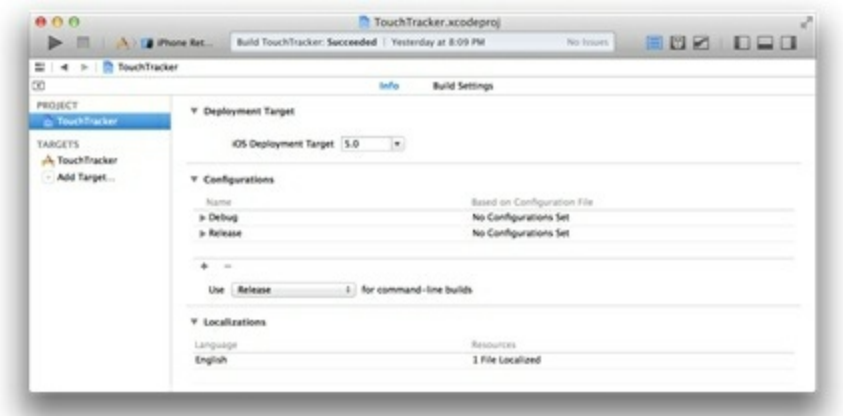


图14-23 构建配置列表

Configurations区域中的表格会列出可以在项目和目标中使用的构建配置。通过表格下方的按钮可以增加或删除构建配置。

修改构建设置

下面做一个练习，修改目标的构建设置：Preprocessor Macros（预处理宏），以巩固之前介绍的内容。预处理宏的作用之一是允许编译器根据条件编译不同的代码段。构建应用时，预处理宏会有已定义和未定义两种状态。只有当某个预处理宏是已定义时，编译器才会编译包含在该预处理指令中的代码。Xcode会在Preprocessor Macros条目中列出当前构建配置中所有已定义的预处理宏。

首先在项目和目标列表中选中TouchTracker目标，然后在Build Settings面板中查找Preprocessor Macros。在Preprocessor Macros条目中，双击Debug配置行的第二列（值）。在新出现的表格中，增加一个条目：VIEW_DEBUG（见图14-24）。

将VIEW_DEBUG加入Debug配置中的Preprocessor Macros后，当Xcode通过debug配置构建TouchTracker目标时，预处理宏VIEW_DEBUG就会处于已定义状态。

下面为TouchTracker增加一些调试代码，并且要求Xcode只有在使用debug配置构建目标时，才能编译这些代码。UIView类有一个名为recursiveDescription的未公开方法，该方法可以向控制台输出应用的整个视图层次结构。但是Apple规定，凡是在App Store中发布的应用一律不能使用未公开的方法。因此，只有当VIEW_DEBUG是已定义的时候，才能调用该方法。

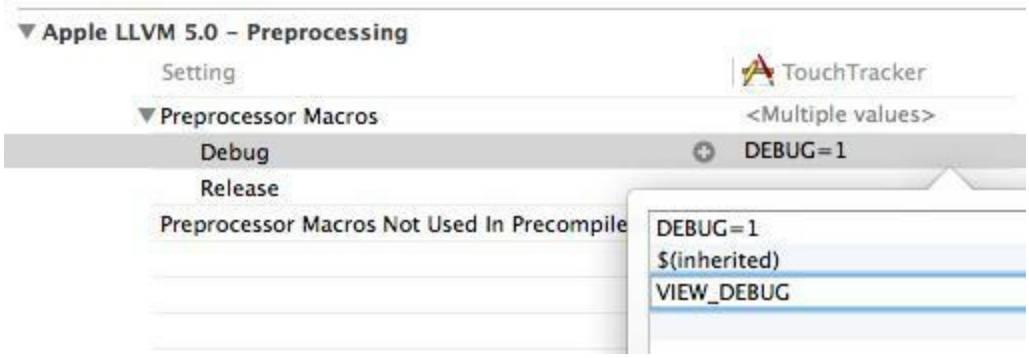


图14-24 修改构建设置

在BNRAppDelegate.m的application:didFinishLaunchingWithOptions:中加入以下代码：

```
[self.window makeKeyAndVisible];
#ifdef VIEW_DEBUG
NSLog(@"%@", [self.window
performSelector:@selector(recursiveDescription)]);
#endif
return YES;
}
```

这段代码会向UIWindow对象发送recursiveDescription消息（因为recursiveDescription是未公开的方法，所以必须用performSelector:来发送消息才能避免Xcode发出警告信息）。recursiveDescription会向控制台输出当前视图（这里是UIWindow对象）的描述信息，然后是其包含的所有子视图的描述信息，接下来是子视图的子视图，依此类推。无论使用哪种配置构建目标，都可以保留这段代码。这是因为在release配置中，VIEW_DEBUG是未定义的，所以在为App Store构建应用时，Xcode不会编译调用recursiveDescription方法的这行代码。

构建并运行应用，TouchTracker应该会向控制台输出当前窗口的整个视图层次结构。

第15章 自动布局入门

本章将通用化(universalize)Homepwner应用，使应用在iPhone和iPad上都能原生(natively)运行。之后还会调整Homepwner的详细界面并介绍自动布局系统(Auto Layout System)，根据设备类型显示相应的界面布局。

15.1 通用化Homepwner

目前Homepwner可以在iPad模拟器上运行，但它是以iPhone模式运行的，如图15-1所示。

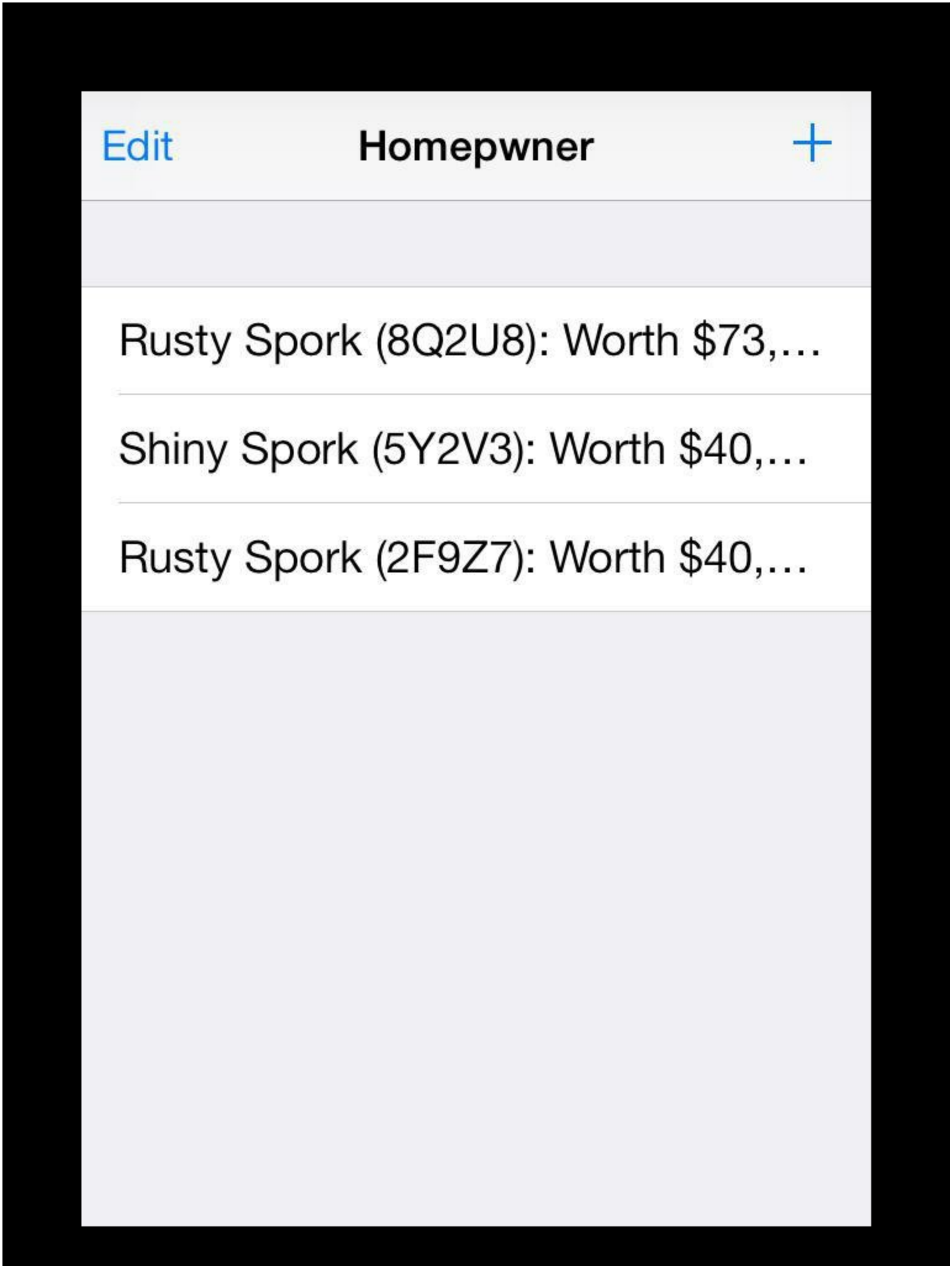

Edit
Homepwner
Rusty Spork (8Q2U8): Worth $73,...
Shiny Spork (5Y2V3): Worth $40,...
Rusty Spork (2F9Z7): Worth $40,...

图15-1 在iPad模拟器上运行的iPhone应用

iPad用户不会喜欢这种“iPhone放大版”的应用。为了让iPad用户感觉Homepwner是专为iPad设计的应用，需要将Homepwner升级为通用应用（universal application）。通用应用是可以在iPhone和iPad上原生运行的应用。

重新打开Homepwner。在项目导航面板中，选择Homepwner项目文件（位于文件列表的顶部）。然后在项目和目标列表中选择Homepwner目标，最后点击General标签。General标签中可以编辑目标的部分属性。

找到Deployment Info（部署信息）部分，点击标题为Device的弹出菜单，将其由iPhone改为Universal（如图15-2）。

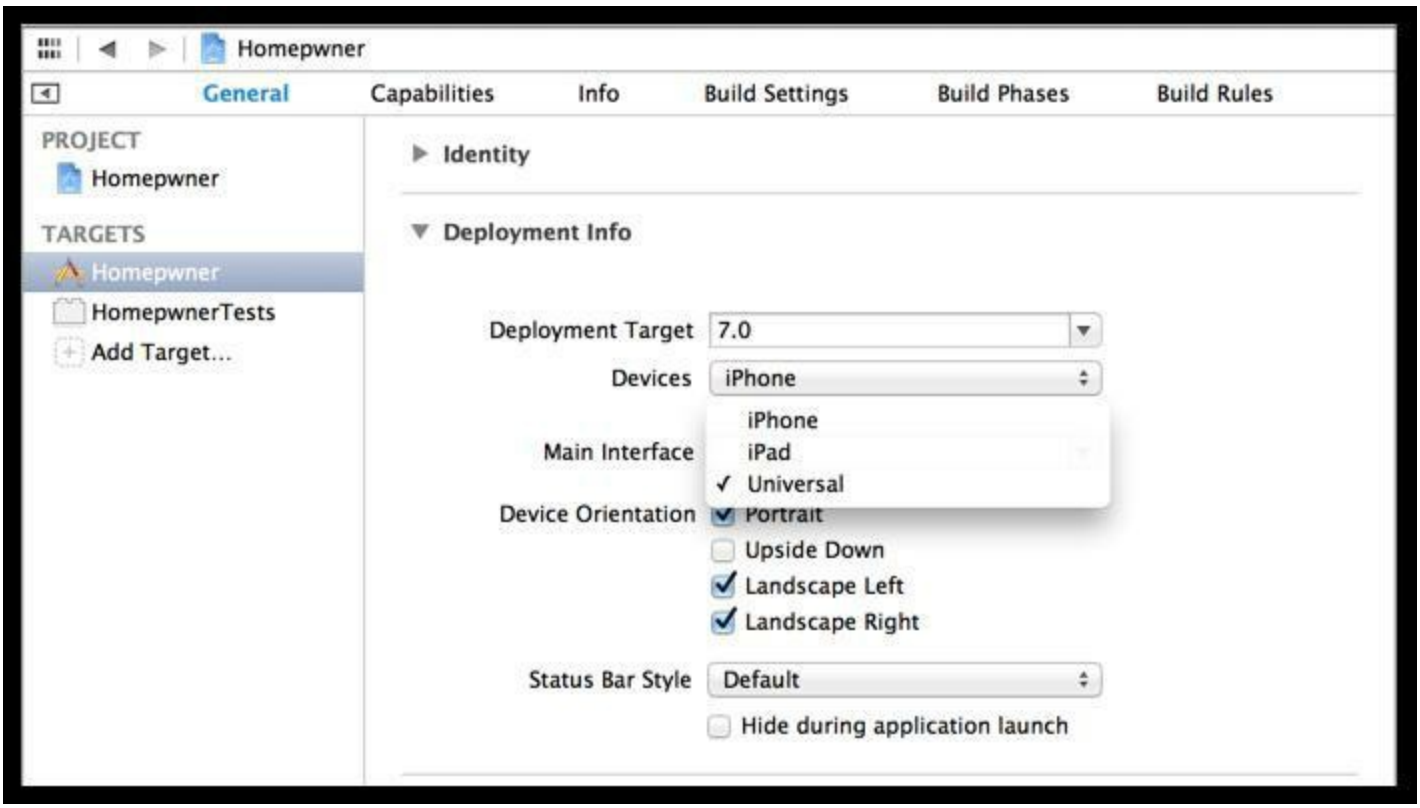


图15-2 通用化Homepwner

现在Homepwner已经是一个通用应用了。从Scheme菜单中选择任意一种iPad模拟器，然后构建并运行应用，检查Homepwner是否可以在iPad上原生运行（见图15-3）。可以发现，Homepwner的表视图和单元格的宽度都和iPad屏幕宽度相同，充分利用了iPad的大屏幕，效果比之前好了很多。

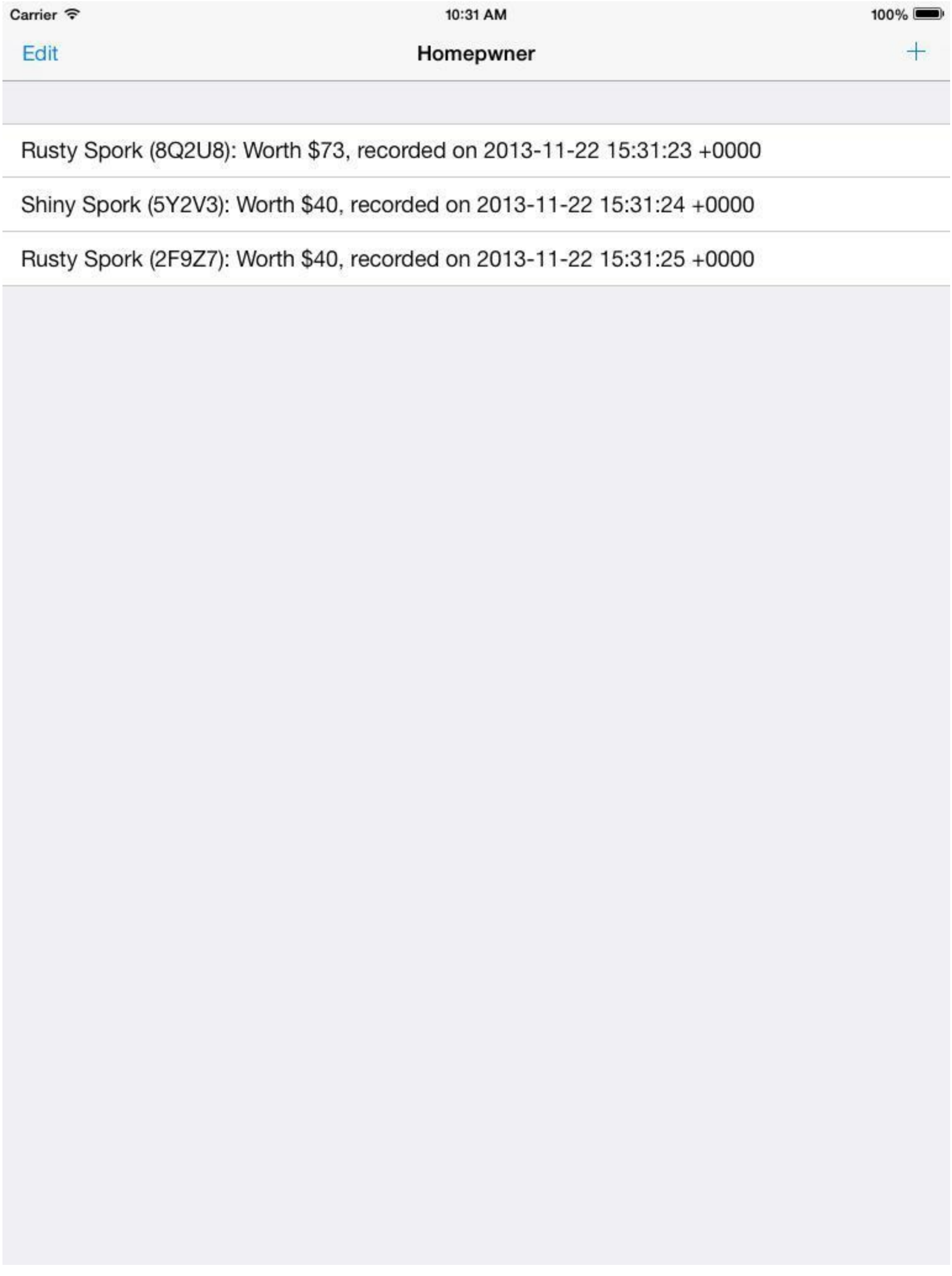

Carrier
10:31 AM
100%
Edit
Homepwner
Rusty Spork (8Q2U8): Worth $73, recorded on 2013-11-22 15:31:23 +0000
Shiny Spork (5Y2V3): Worth $40, recorded on 2013-11-22 15:31:24 +0000
Rusty Spork (2F9Z7): Worth $40, recorded on 2013-11-22 15:31:25 +0000

图15-3 在iPad模拟器上运行的通用应用

现在添加一个BNRItem对象，然后点击该对象进入详细界面。如图15-4所示，详细界面没有在iPad上自动放大，需要进一步调整。

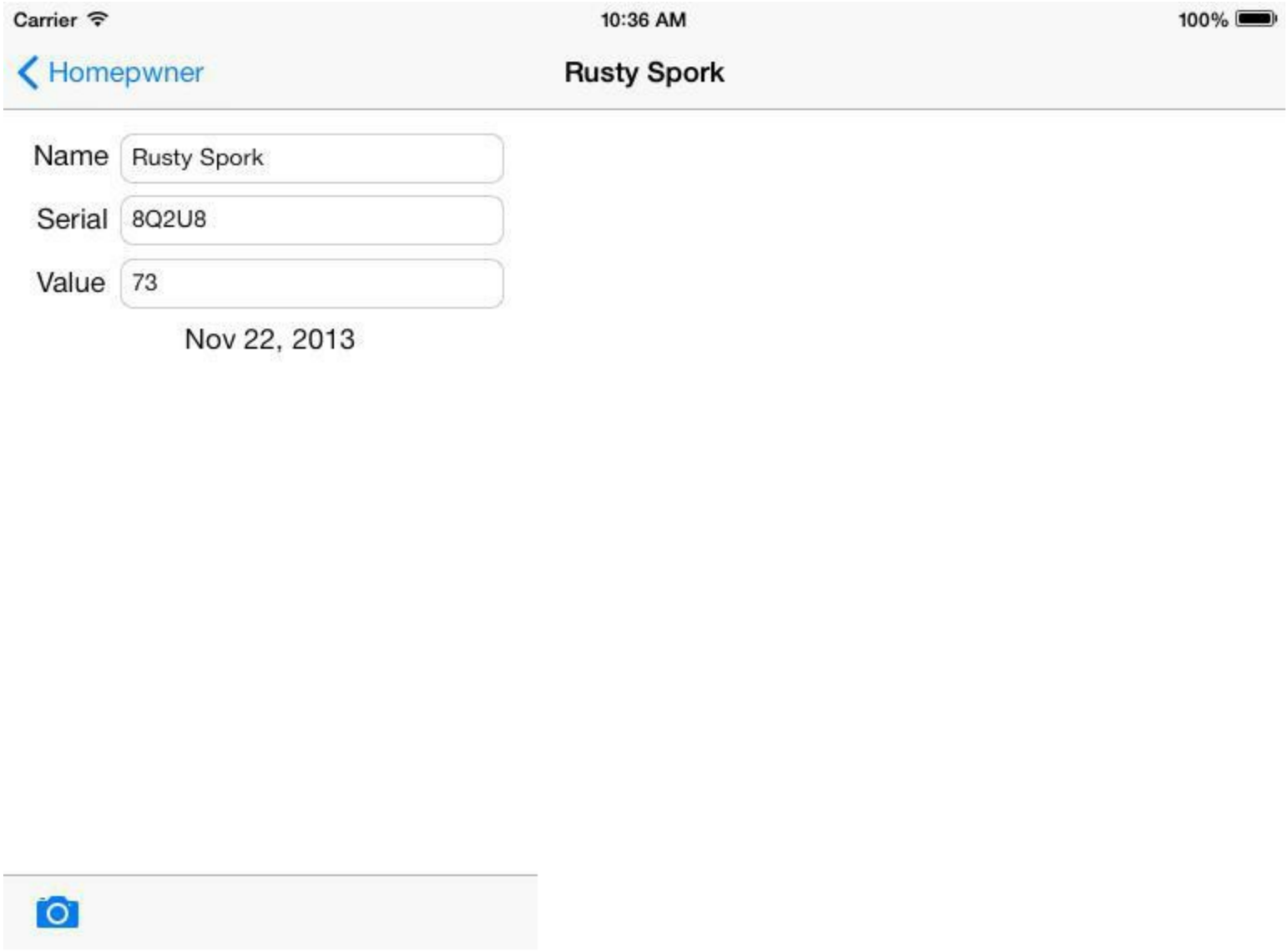

Carrier
10:36 AM
100%
Homepwner
Rusty Spork
Name
Rusty Spork
Serial
8Q2U8
Value
73
Nov 22, 2013

图15-4 详细界面没有自动放大

表视图可以自动调整大小以适配iPad屏幕，但是自定义的详细界面并不知道在iPad上如何显示。下面就通过自动布局系统告诉详细界面在不同设备中的显示方式。

15.2 自动布局系统

第4章介绍过，视图的frame属性定义了视图的大小和相对于父视图的位置。之前的章节中，无论是编写代码还是在Interface Builder中设置，frame都是基于绝对坐标（absolute coordinates）的。绝对坐标降低了界面布局的灵活性，它要求开发者事先知道屏幕的具体尺寸，无法自动适配不同尺寸的屏幕。

相反，自动布局描述的是视图之间的相对位置关系，它可以在应用运行时根据设备的屏幕尺寸动态改变各个视图的frame。

表15-1列出了不同设备的屏幕尺寸。（请读者注意，对于自动布局系统来说，Retina屏幕的尺寸与非Retina屏幕的尺寸是“相同”的，因为视图的坐标单位使用的是点而不是像素，虽然Retina屏幕的像素数目是非Retina屏幕的4倍，但是两者的点数目是相同的。）

表15-1 设备的屏幕尺寸

设　　备	宽度×高度（点）
iPhone/iPod touch（4S 及更早版本）	320×480
iPhone/iPod touch（5 及更高版本）	320×568
iPad（所有版本）	768×1024

对齐矩形和布局属性

自动布局系统为视图定义了一个新的矩形区域——对齐矩形（alignment rectangle），该矩形有若干布局属性（见图15-5）。

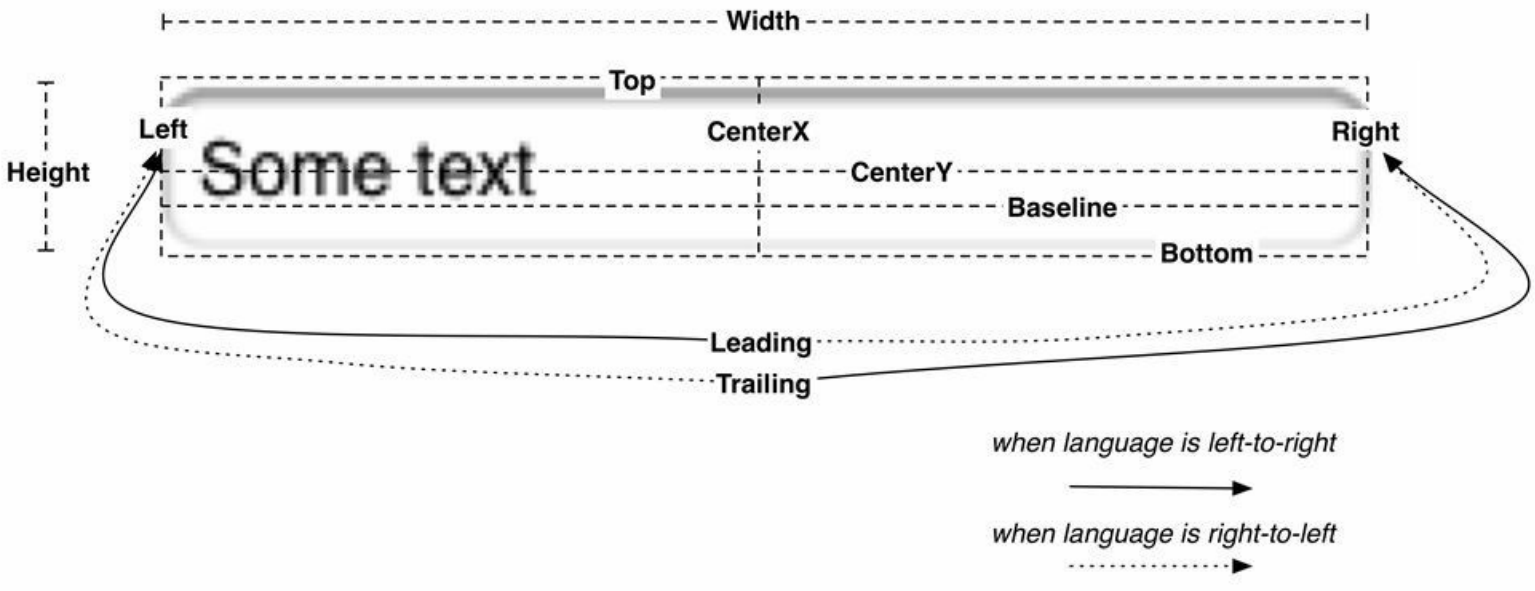


图15-5 布局属性定义了视图的对齐矩形

宽/高（Width / Height）	定义对齐矩形的大小
顶边/底边/左边/右边 （Top / Bottom / Left / Right）	定义对齐矩形的四条边与视图层次结构中另一个对齐矩形的距离
中心点坐标（CenterX / CenterY）	定义对齐矩形中心点的位置
基准线（Baseline）	对于大部分视图来说，基准线与底边相同，但是也有例外，例如，UITextField的基准线是其显示的文本区域的底边。这样可以避免下行字母（descenders，指底部超出基准线的字母，例如“g”和“p”）被紧贴在UITextField下方的视图遮盖
起始位置/终止位置 （Leading / Trailing）	只用于负责显示文本的视图，例如UITextField和UILabel。如果设备当前设置的语言是从左向右阅读的（例如英语），那么对齐矩形的左边就是起始位置，而右边是终止位置；相反，如果设备语言是从右向左阅读的（例如阿拉伯语），那么右边是起始位置，左边是终止位置

XIB文件默认会使用自动布局系统，并为包含的所有视图都定义一个对齐矩形。但是，对于复杂的界面布局，例如Homepwner的详细界面，默认的显示效果通常并不好，需要进一步设置各个视图的布局方式。

对于开发者来说，由于事先不知道屏幕尺寸，因此无法直接为视图定义对齐矩形。相反，只需要为视图设置一系列约束（constraints），系统就可以确定所有布局属性的值，并根据布局属性为视图定义对齐矩形。

约束

约束既可以定义布局属性的具体值，也可以定义布局属性之间的关系。例如，可以为视图添加如下约束：“视图的高度始终是44点”“两个视图之间的垂直距离是8点”“这些视图的宽度始终保持相等”等。

虽然视图的布局属性很多，但是并不需要为每一个布局属性都添加一个约束。自动布局系统会根据约束和布局属性之间的关系计算出每一个布局属性的值，例如，如果为视图定义了左边和宽度约束，那么系统会自动计算出右边：左边＋宽度＝右边。

如果系统在检查了所有约束之后，仍然无法确定某些布局属性的值，就会提示“缺少约束（Missing Constraints）”或“有歧义的布局（Ambiguous Layout）”。如果在这种情况下构建应用，Xcode并不会报错，但是在应用运行时，系统将无法正确对界面布局。本章稍后会介绍如何调试这类布局错误。

下面就来练习为BNRDetailViewController中的UIToolbar对象添加约束，首先需要考虑UIToolbar对象在不同屏幕中的布局方式：

•UIToolbar对象应该始终位于屏幕底部。

•UIToolbar对象的宽度应该和屏幕宽度保持相等。

•UIToolbar对象的高度始终是44点。（Apple建议的UIToolbar标准高度。）

为了将期望的布局方式转换为约束，需要引入最近相邻视图（nearest neighbor）的概念。最近相邻视图通常是指视图在某个方向上距离最近的兄弟视图（视图层次结构中的同级视图），但是，如果视图在该方向上没有兄弟视图，那么最近相邻视图就是其父视图（见图15-6）。

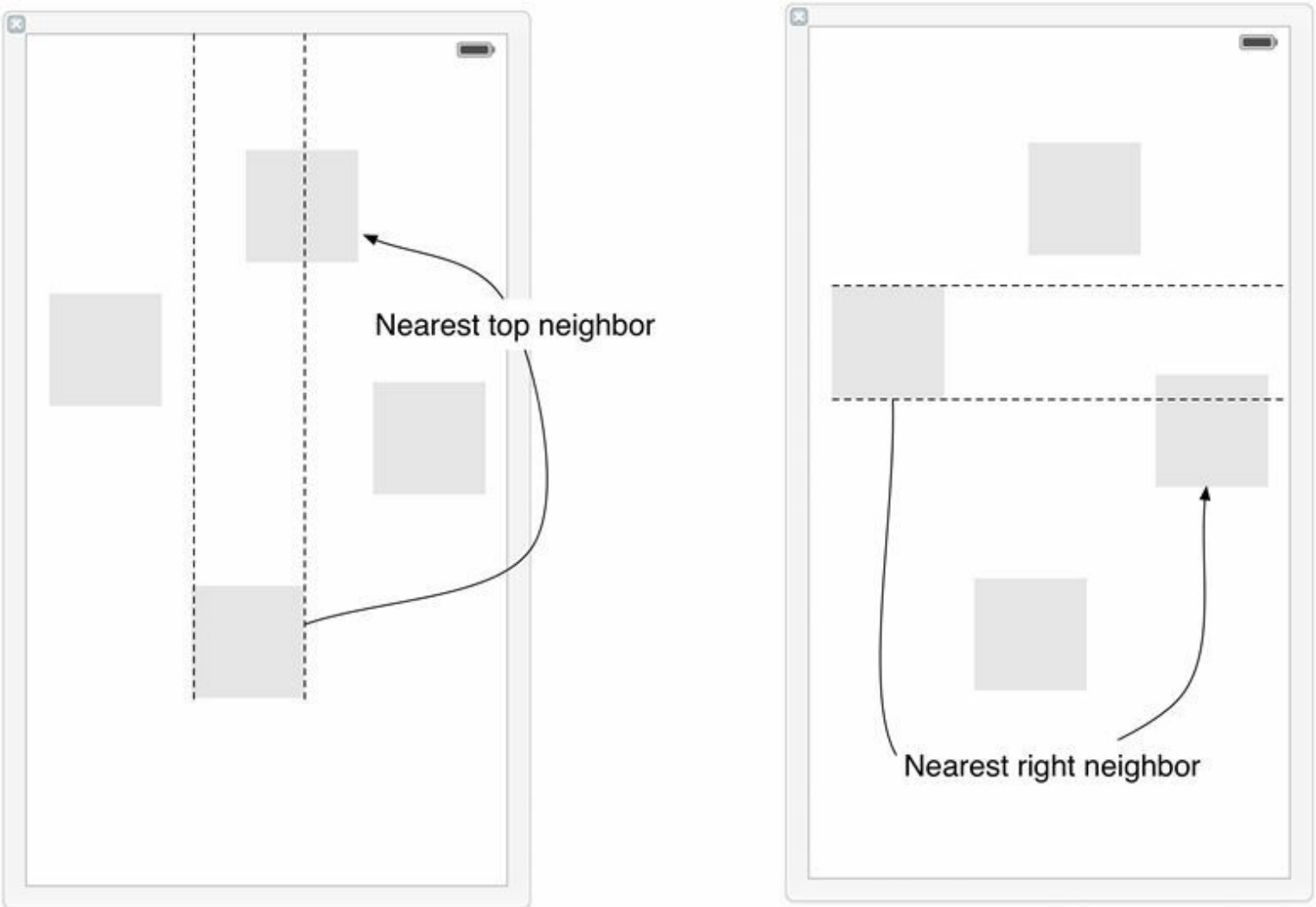


图15-6 最近相邻视图

根据最近相邻视图的概念，可以将UIToolbar对象的布局方式转换为以下约束：

（1）UIToolbar对象的底边与其最近相邻视图（BNRDetailViewController的view，也是UIToolbar对象的父视图）的距离始终是0点。

（2）UIToolbar对象的左边与其最近相邻视图的距离始终是0点。

(3)UIToolbar对象的右边与其最近相邻视图的距离始终是0点。

(4)UIToolbar对象的高度始终是44点。

读者可能会发现，以上约束并没有包括UIToolbar对象的宽度和顶边。实际上，第2条和第3条约束已经限定了UIToolbar对象的宽度与屏幕宽度始终保持相等，而第1条和第4条约束则限定了UIToolbar对象的顶边与其最近相邻视图的距离始终是屏幕高度减去44点。

现在可以为UIToolbar对象添加约束了，与添加视图的过程类似，可以通过Interface Builder或编写代码为视图添加约束。Apple建议开发者尽可能使用Interface Builder，本章也将介绍如何在Interface Builder中添加约束。如果读者需要通过代码创建视图，可以参考本书第16章如何通过代码为视图添加约束。

15.3 在Interface Builder中添加约束

首先打开BNRDetailViewController.xib文件，选中画布中的UIImageView对象并将其删除。第16章会使用代码重新创建该对象并为其添加约束。

然后选中UIToolbar对象，并找到画布右下角的自动布局约束菜单（Auto Layout Constraint Menu），如图15-7所示。

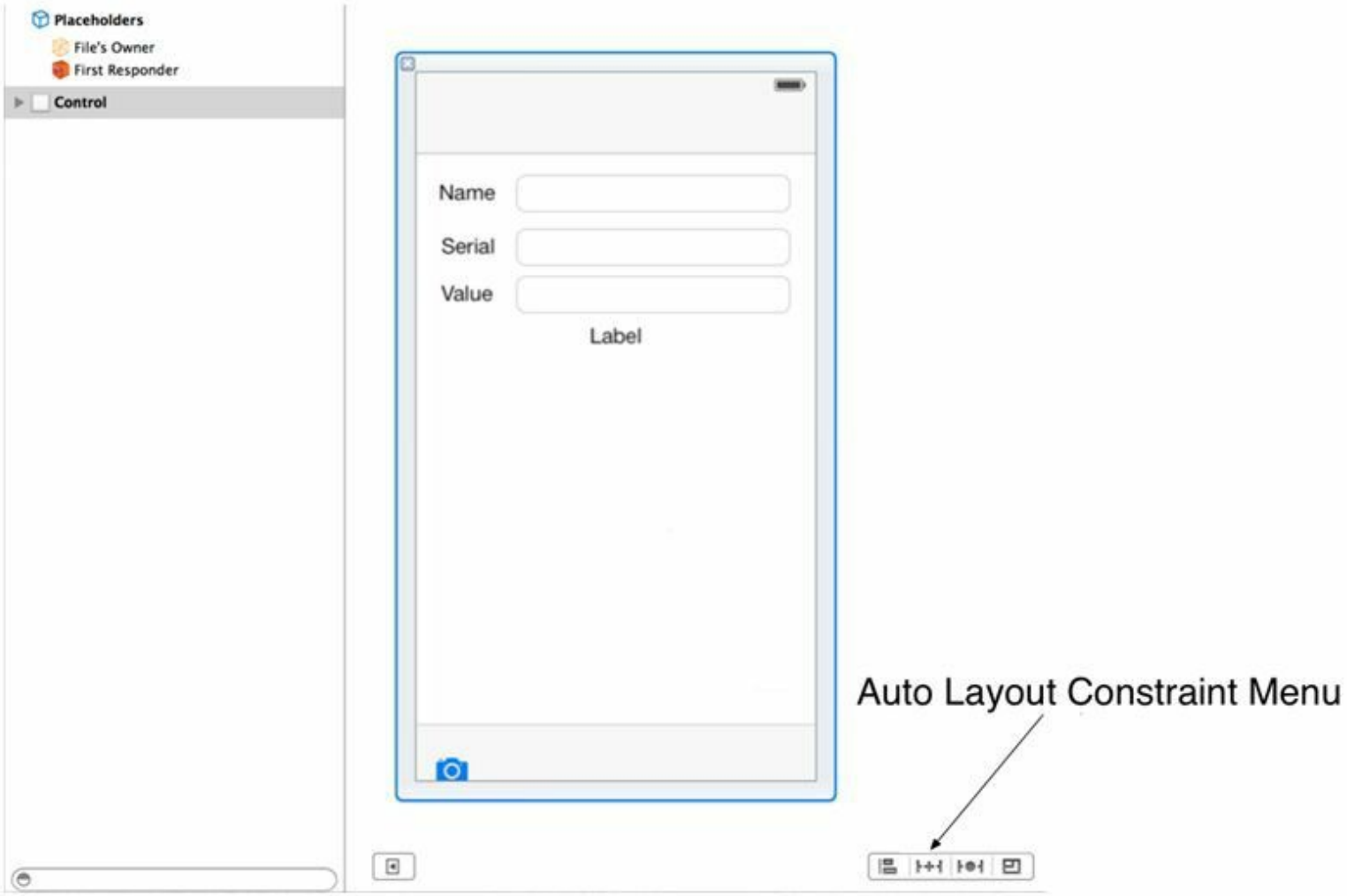


图15-7 自动布局约束菜单

点击图标（左边第二个），显示Pin菜单。Pin菜单显示了UIToolbar对象的大小和位置，可以在菜单中为UIToolbar对象添加需要的约束（见图15-8）。

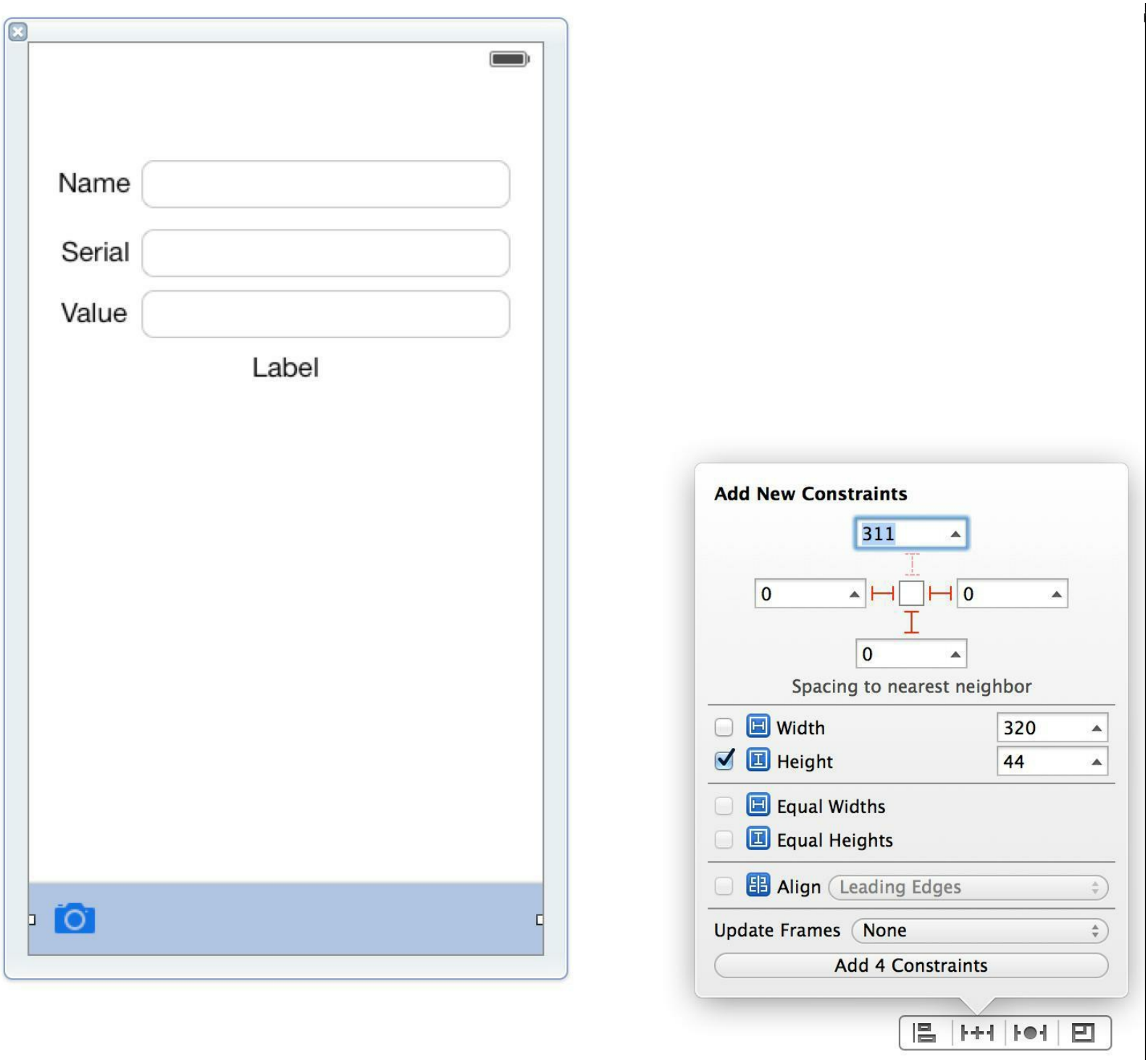


图15-8 为UIToolbar对象添加四个约束

Pin菜单的顶部显示了UIToolbar对象在四个方向上与最近相邻视图的距离，如前所述，这里需要设置左边、右边、底边与最近相邻视图的距离都是0点。UIToolbar对象在左边、右边、底边方向上都没有兄弟视图，因此这三个方向上的最近相邻视图都是BNRDetailViewController的view。

点击输入框和中间小矩形之间的橘红色虚线，虚线会变成实线。这样就可以将输入框中的值添加为视图的约束。

在Pin菜单的中间位置找到UIToolbar对象的高度（Height），可以发现Height的默认值就是44点，直接勾选Height旁边的选择框，为UIToolbar对象添加高度约束。现在Pin菜单底部的按钮会显示“Add 4 Constraints（添加四个约束）”，点击该按钮完成添加约束。

在iPad模拟器上构建并运行应用，创建一个BNRItem对象，然后点击该对象进入详细界面。UIToolbar对象会显示在屏幕底部，而且宽度与屏幕宽度相同，如图15-9所示。

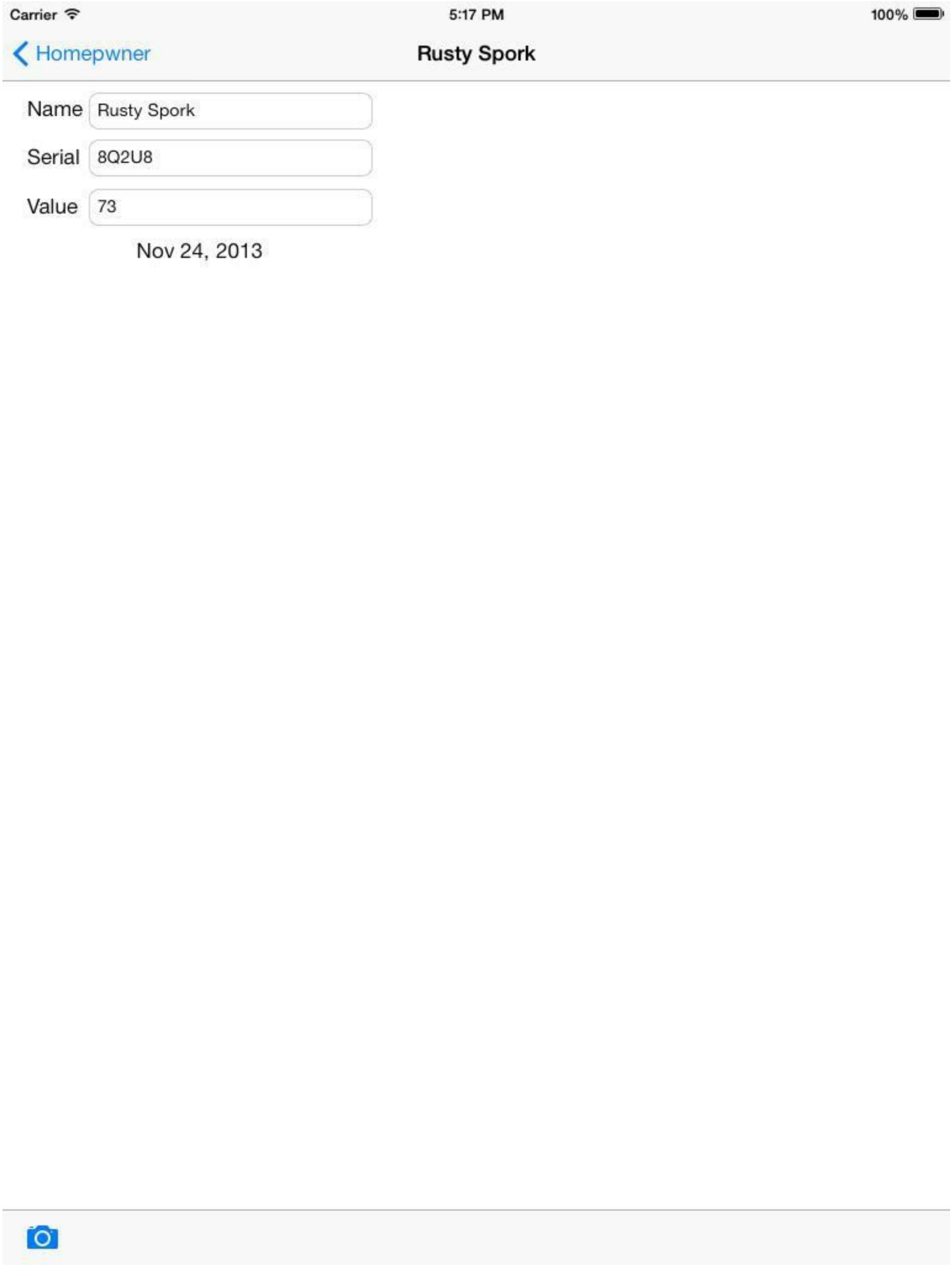

Carrier
5:17 PM
100%
Homepwner
Rusty Spork
Name
Rusty Spork
Serial
8Q2U8
Value
73
Nov 24, 2013

图15-9 布局正确的UIToolbar对象

可以在画布左侧的Dock中看到为UIToolbar对象添加的约束。找到Constraints并点击左侧三角形的展开按钮，可以发现，其中只有三个约束。第四个约束是限定UIToolbar对象的高度，它包含在UIToolbar对象中。点击Toolbar，找到Toolbar下方的Constraints，展开就可以看到第四个约束（见图15-10）。

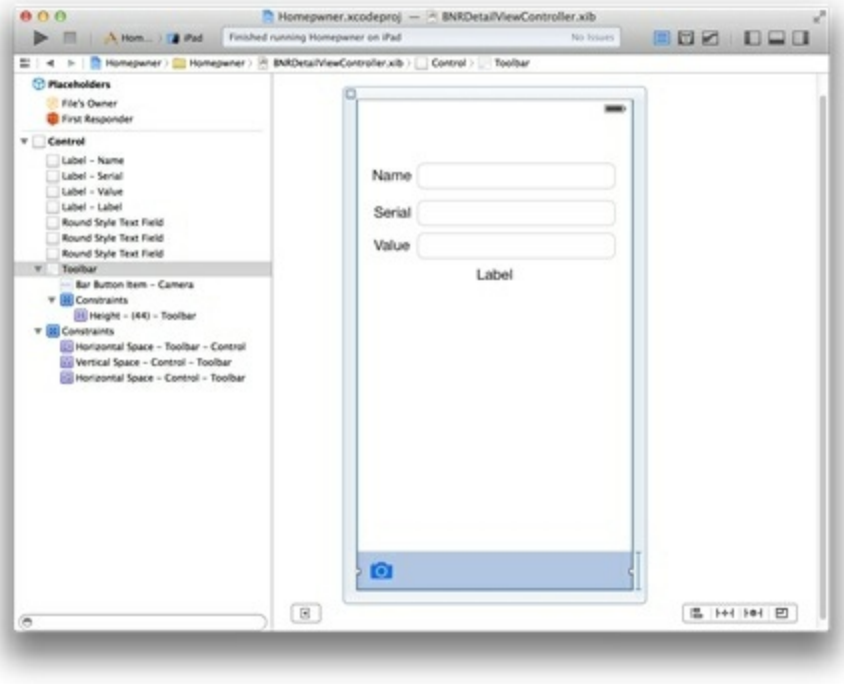


图15-10 布局正确的UIToolbar对象

读者可能会问，这四个约束都是加在UIToolbar对象上的，为什么只有第四个约束包含在UIToolbar对象中？这是因为，前三个约束涉及与父视图布局属性的关系。如果约束同时作用于某个视图及其父视图，那么Interface Builder会将约束加在父视图上。所以，前三个约束实际上是加在UIToolbar对象的父视图Control上的，也就是BNRDetailViewController的view（请读者回忆第11章，为了可以通过点击视图关闭键盘，第11章将该视图的类由UIView改为了UIControl）。相反，第四个约束用于限定UIToolbar对象的高度，只作用于UIToolbar对象，所以Interface Builder会将约束加在UIToolbar对象上。

如果使用Interface Builder，约束会自动添加到恰当的视图上。而代码中创建和添加约束的步骤与使用Interface Builder的不同，第16章会介绍在代码中如何判定约束应该添加到哪个视图上。

在画布中，约束表现为蓝色的直线。例如，如果在Dock中选中某个约束，画布中就会显示相应的蓝色直线表示该约束处于选中状态；如果在画布中选中某个视图，则会显示所有表示该视图约束的蓝色直线。（有时这条直线位于视图边缘，不容易发现。）

删除约束也非常简单，只需要在Dock中选中某个约束，或者在画布中选中表示某个约束的蓝色直线，然后按下删除（Delete）键即可。请读者尝试删除UIToolbar对象的高度约束，然后在Pin菜单中重新为UIToolbar对象添加高度约束。

添加更多约束

下面为Name标签添加约束。目前Name标签在iPad上显示效果很好，但是，由于本书第25章会为Homepwner添加多语言支持，而第27章会让Homepwner动态调整字体，因此仍然需要为Name标签添加约束。

如果不考虑语言、字体和屏幕尺寸，Name标签应该始终处于屏幕左上角而且大小始终保持不变。在画布中选中Name标签，然后点击Pin菜单。

在Pin菜单顶部选择左边和顶边，Name标签在这两个方向上的最近相邻视图是其父视图（BNRDetailViewController的view）。然后，勾选Width和Height旁边的选择框，限定Name标签的大小是固定值。

现在Pin菜单应该如图15-11所示。请注意，约束的值是根据视图在画布上的位置计算出来的，读者的Name标签位置可能与图中不同，所以约束的值也可能不同。如果读者的Pin菜单中输入框的值与图中不一致，不需要将其修改成图中的值，否则Name标签的约束与其画布上的位置不一致，会出现Misplaced Views（视图位置错误）警告。本章稍后会介绍如何处理Misplaced Views警告。

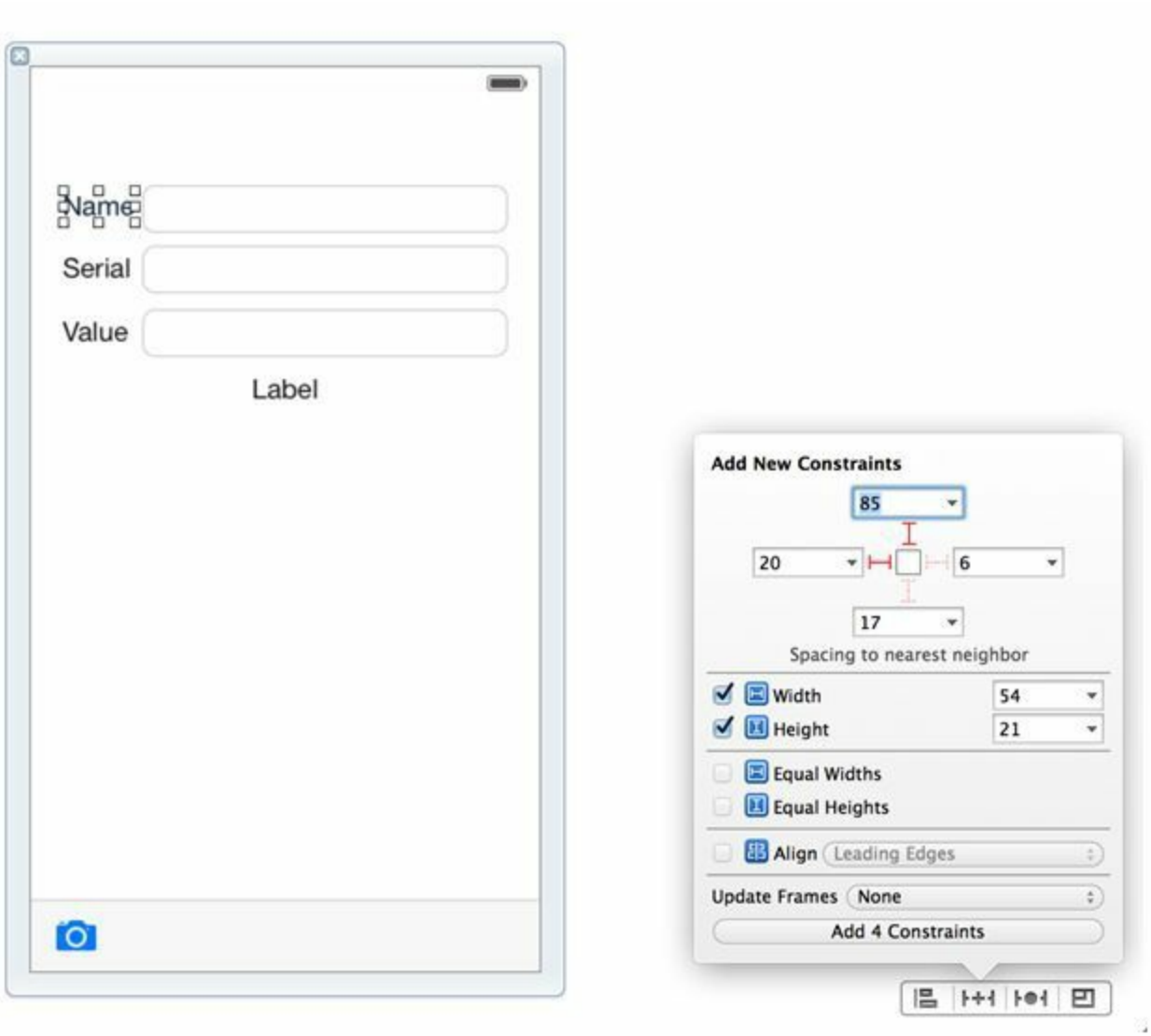


图15-11 为Name标签添加约束

在Pin菜单底部点击Add 4 Constraints，为Name标签添加约束。

现在考虑Name标签右边的文本框。如果不考虑屏幕尺寸，文本框应该位于Name标签右侧，并且填充大部分屏幕。

选中文本框，然后打开Pin菜单。在菜单顶部选择左边和右边，添加这两个方向上的约束，使文本框的左边与Name标签保持当前距离，右边与屏幕保持20点的距离。这样文本框的宽度会从Name标签右侧延伸到距离屏幕右侧20点的位置。

现在文本框的约束出现了问题。画布中表示文本框约束的直线在正常情况下是蓝色的，但是现在变成了橘红色。这表示文本框缺少约束，自动布局系统无法根据当前约束确定所有布局属性的值，也就无法为文本框定义对齐矩形。

为了知道问题的详细信息，可以在Dock中点击Control旁边的红色图标进入约束问题列表（见图15-12）。

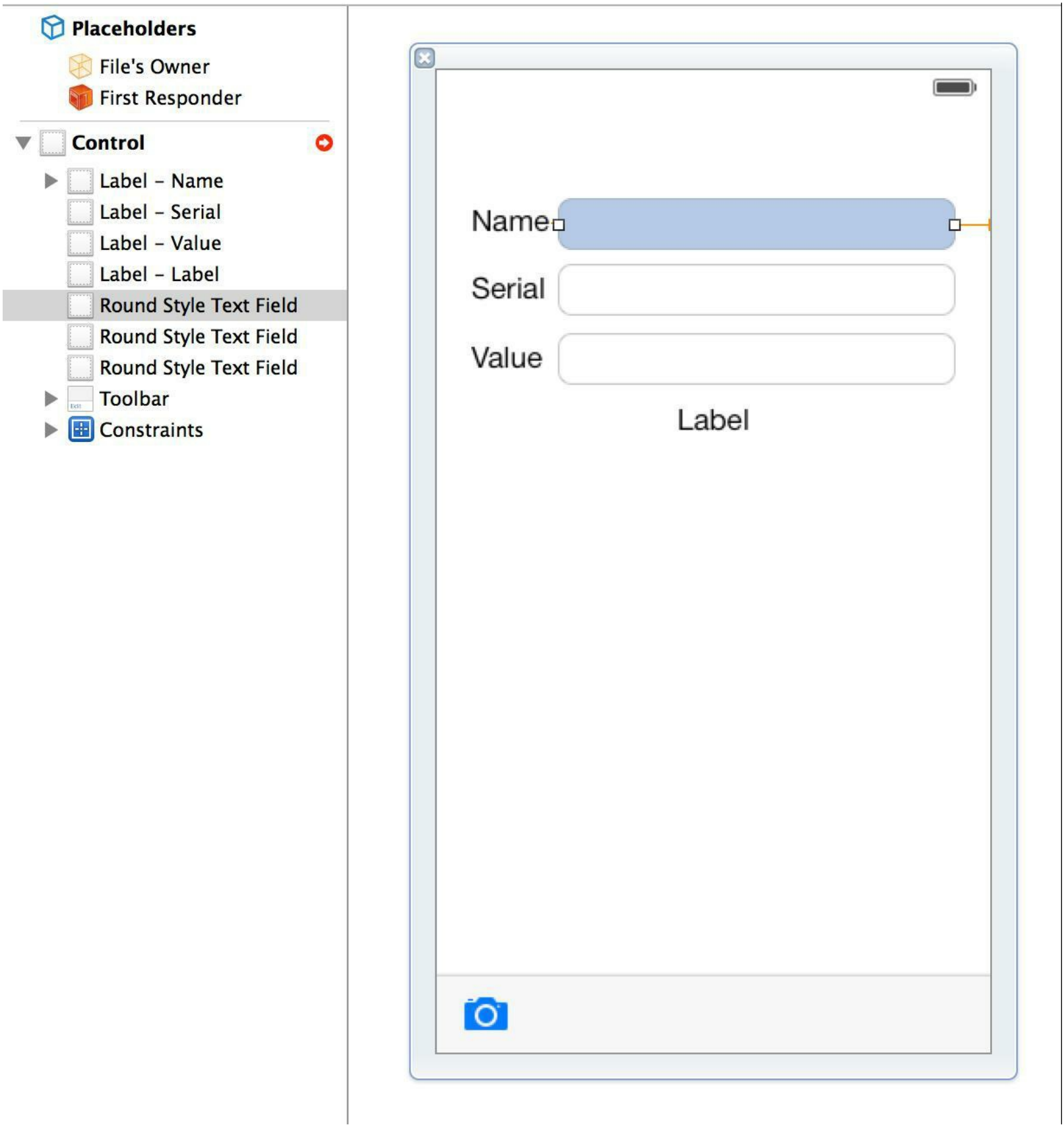


图15-12 缺少约束的文本框

列表中的问题会按照类型分组，文本框的约束问题属于Missing Constraints（缺少约束）。根据问题描述可以知道，目前文本框缺少Y轴（垂直）方向上的位置约束（Need Constraints for: Y position）。解决方法是，打开Pin菜单，在菜单顶部选择顶边，限制文本框的顶边与其父视图的距离始终保持不变，然后点击Add 1 Constraint添加约束。

在Interface Builder中，还可以将多个视图按照某个布局属性对齐。因此上述问题还有另一种更好的解决方案：将文本框与Name标签沿基准线对齐。这样当用户输入文字时，文本框中的文字会与Name标签中的文字位于同一基准线上，看起来会非常整齐。

在画布中选中文本框，然后按住Shift键不放，选中Name标签。这样可以同时选中文本框和Name标签。接下来在约束菜单中点击图标，显示Align（对齐）菜单，再勾选标题为Baselines的选择框，最后点击Add 1 Constraint添加约束（见图15-13）。

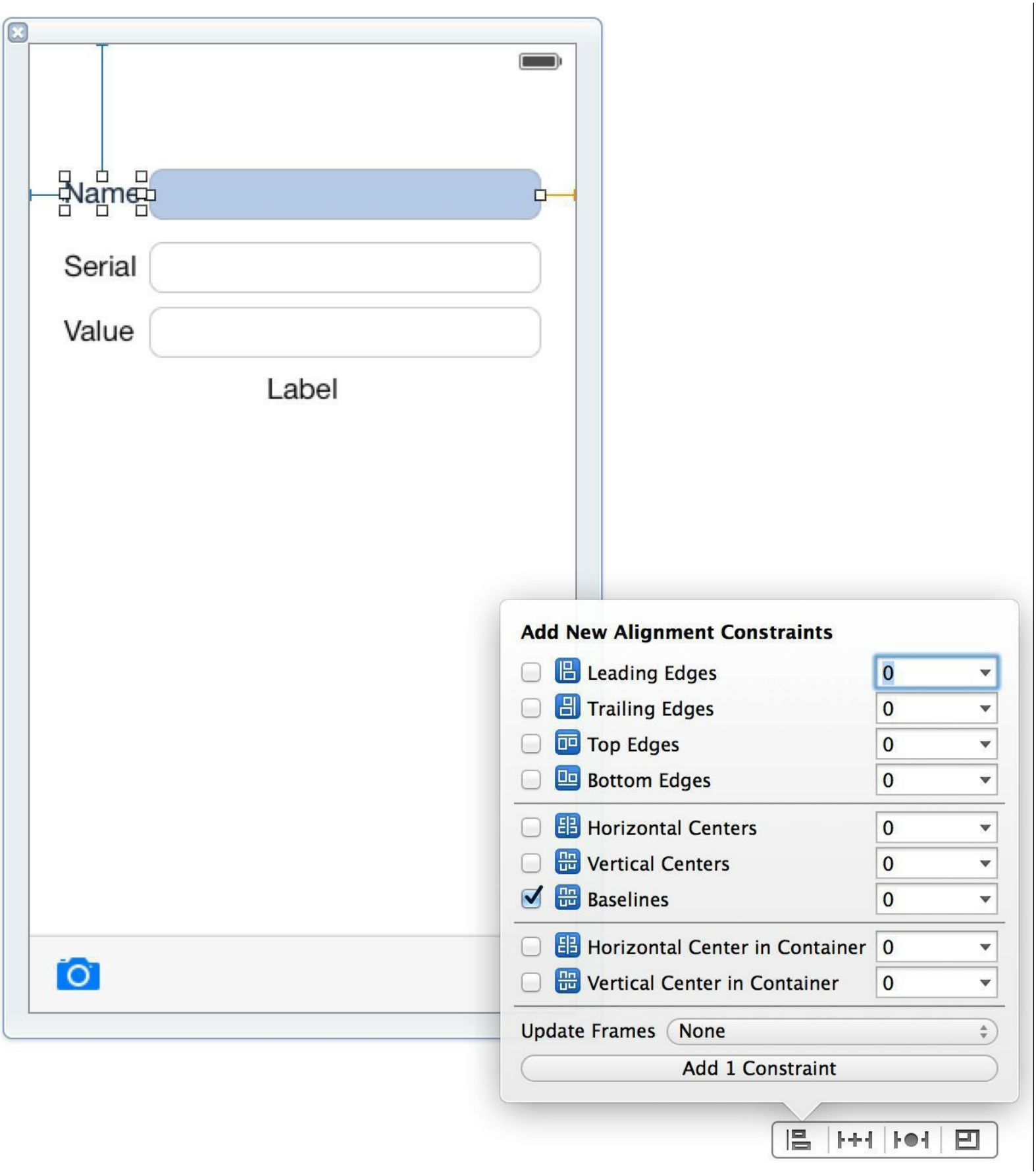


图15-13 让Name标签和文本框按基准线对齐

现在文本框中表示约束的直线会由橘红色变为蓝色，同时Control旁边的红色图标也消失了——自动布局系统已经可以根据文本框的约束确定文本框的大小和位置了。

除缺少约束之外，还有另外两类约束问题：约束冲突和视图位置错误。本章稍后会介绍如何

调试这些约束问题。

在iPad上构建并运行应用，选中某个BNRItem对象进入详细界面。可以看到，顶部的文本框位于Name标签右边，并且填充了大部分屏幕（见图15-14）。

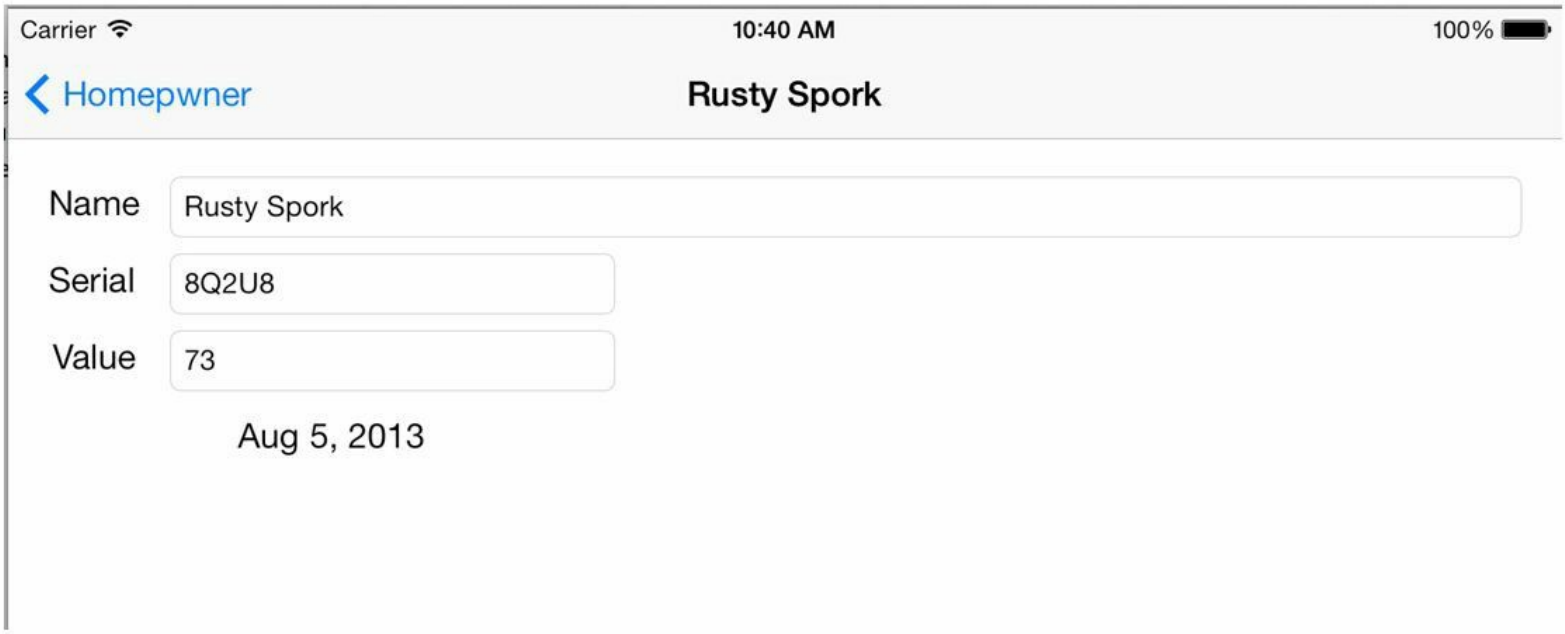


图15-14 文本框填充了大部分屏幕

为其余视图添加约束

下面继续为其余视图添加约束。首先使用约束菜单为Serial标签添加以下约束：

- 顶边与Name标签保持当前距离。
- 左边与Name标签的左边对齐。
- 宽度和高度保持当前值不变。

接下来介绍一种新方法：与设置插座变量和动作方法类似，通过在画布中拖曳为视图添加约束。在画布中选中Serial标签，按住Control键，然后将其拖曳到Name标签并释放鼠标。这时Interface Builder会弹出一个约束列表，如图15-15所示。（列表中显示的约束与拖曳方向、源视图（from view）和目标视图（to view）有关。Interface Builder会自动判断可以为视图添加哪些约束。）

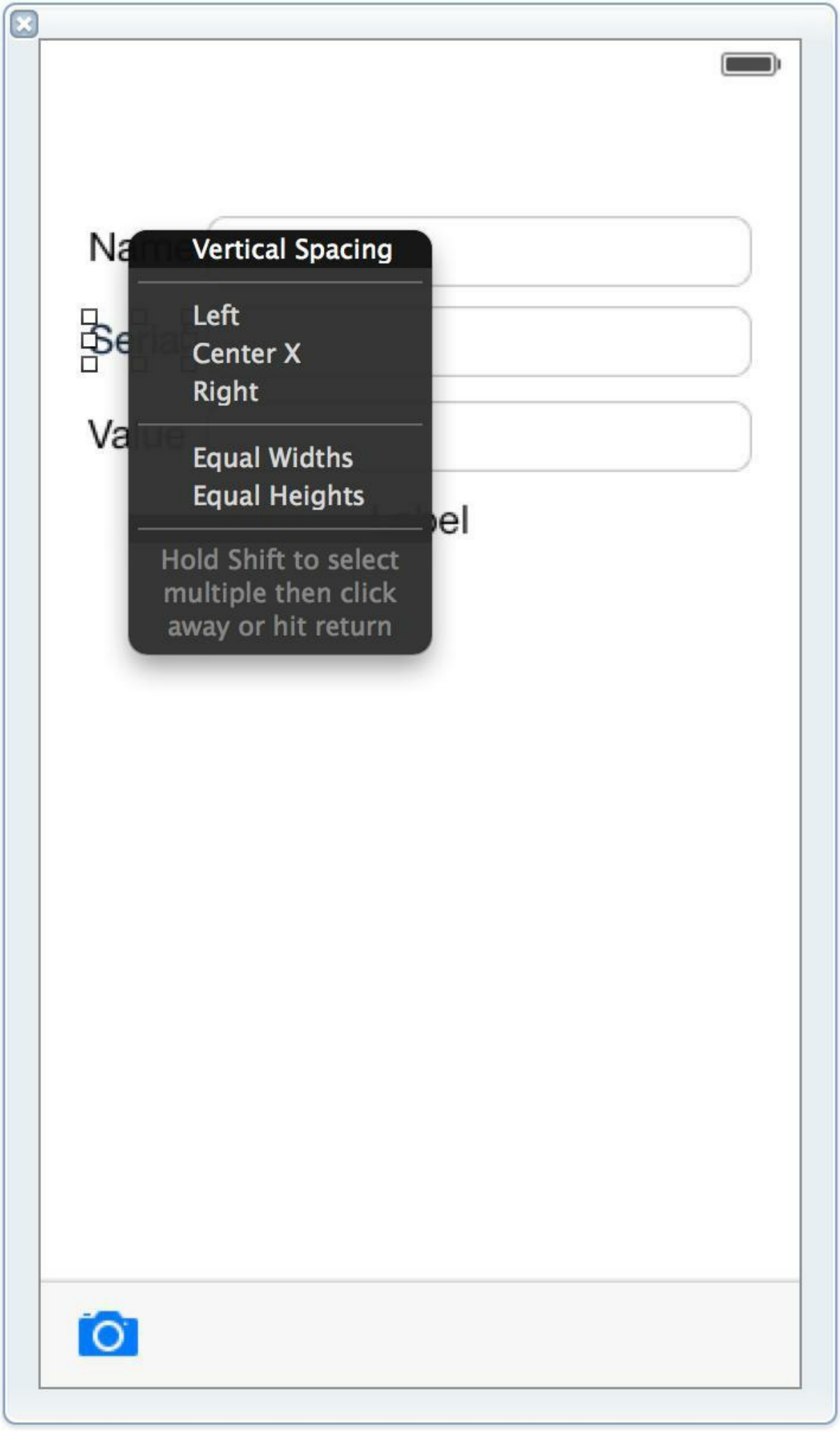

Vertical Spacing
Left
Center X
Right
Equal Widths
Equal Heights
Hold Shift to select multiple then click away or hit return

图15-15 通过拖曳为视图添加约束

在约束列表中选择Vertical Spacing(垂直距离)，使Serial标签和Name标签的垂直距离保持当前值不变——等同于在Pin菜单中选择顶边。

再次将其拖曳到Name标签，这次选择Left(左边)——等同于在Align菜单中选择左对齐。

最后需要固定Serial标签的宽度和高度。因为宽度和高度约束只对Serial标签自身有影响，所以需要将Serial标签拖曳到自身。按住Control键，将Serial标签沿倾斜方向拖曳到自身，然后按住Shift键，在约束列表中同时选择Width(宽度)和Height(高度)。(如果读者没有同时看见Width和Height选项，是因为没有沿倾斜方向拖曳。如果沿垂直方向拖曳，则不会出现Width;相反，如果沿水平方向拖曳，则不会出现Height。)

现在为Serial标签右边的文本框添加以下约束:

•左边与Serial标签保持当前距离。

•基准线与Serial标签的基准线对齐。

•右边与父视图保持当前距离。

选中文本框并将其拖曳到Serial标签，在约束列表中同时选择Horizontal Spacing(水平距离)和Baseline(基准线)。再将其向右拖曳到父视图，选择Trailing Space to Container(右边与父视图保持当前距离)。

请读者通过约束菜单或拖曳为其余三个视图添加约束。

Value标签:

•顶边与Serial标签保持当前距离。

•左边与Serial标签的左边对齐。

•宽度和高度保持当前值不变。

Value标签右侧的文本框:

•左边与Value标签保持当前距离。

•基准线与Value标签的基准线对齐。

•右边与父视图保持当前距离。

date标签:

•顶边与Value标签右侧的文本框保持当前距离。

•左边和右边与父视图保持当前距离。

•高度保持当前值不变。

在iPad上构建并运行应用。可以发现，添加约束后，BNRDetailViewController的界面效果看起来好多了。

优先级

每个约束都具有优先级（priority level），如果多个约束之间有冲突，自动布局系统会根据优先级决定使用哪些约束。优先级的取值范围是1到1000，默认值是1000，表示约束是必需（required constraint）的。因此，之前添加的约束都是必需的，优先级都是1000，如果这些约束之间存在冲突，优先级无法帮助自动布局系统解决约束冲突。自动布局系统在这种情况下会提示约束问题，下一节就来介绍如何调试约束问题。

15.4 调试约束问题

现在BNRDetailViewController.xib中的所有视图都添加了约束，自动布局系统也为所有视图定义了对齐矩形，因此BNRDetailViewController的view在iPhone和iPad中的显示效果都很好。

在添加大量约束的过程中，很容易产生问题，例如缺少约束、约束冲突和视图位置错误。自动布局系统提供了若干用于调试约束问题的工具，下面就依次介绍这些工具以及如何使用这些工具调试约束问题。

有歧义的布局

有歧义的布局(ambiguous layout)是指自动布局系统无法根据当前约束确定视图的所有布局属性，该问题通常是由于视图缺少约束。

目前BNRDetailViewController.xib中没有缺少约束的视图，下面就来演示这类问题。首先在UIControl中的date标签下方添加两个标签，分别位于屏幕左侧和右侧。然后打开属性检视面板，将两个标签的背景颜色改为浅灰色，以便查看它们的frame，界面应该类似图15-16。


Name
Serial
Value
Label
Label
Label

图15-16 添加两个新标签

接下来为两个标签添加一些约束。按住Shift键，同时选中两个标签，再打开Pin菜单，在菜单顶部选择顶边、左边和右边。最后点击Add 5 Constraints添加约束（见图15-17）。

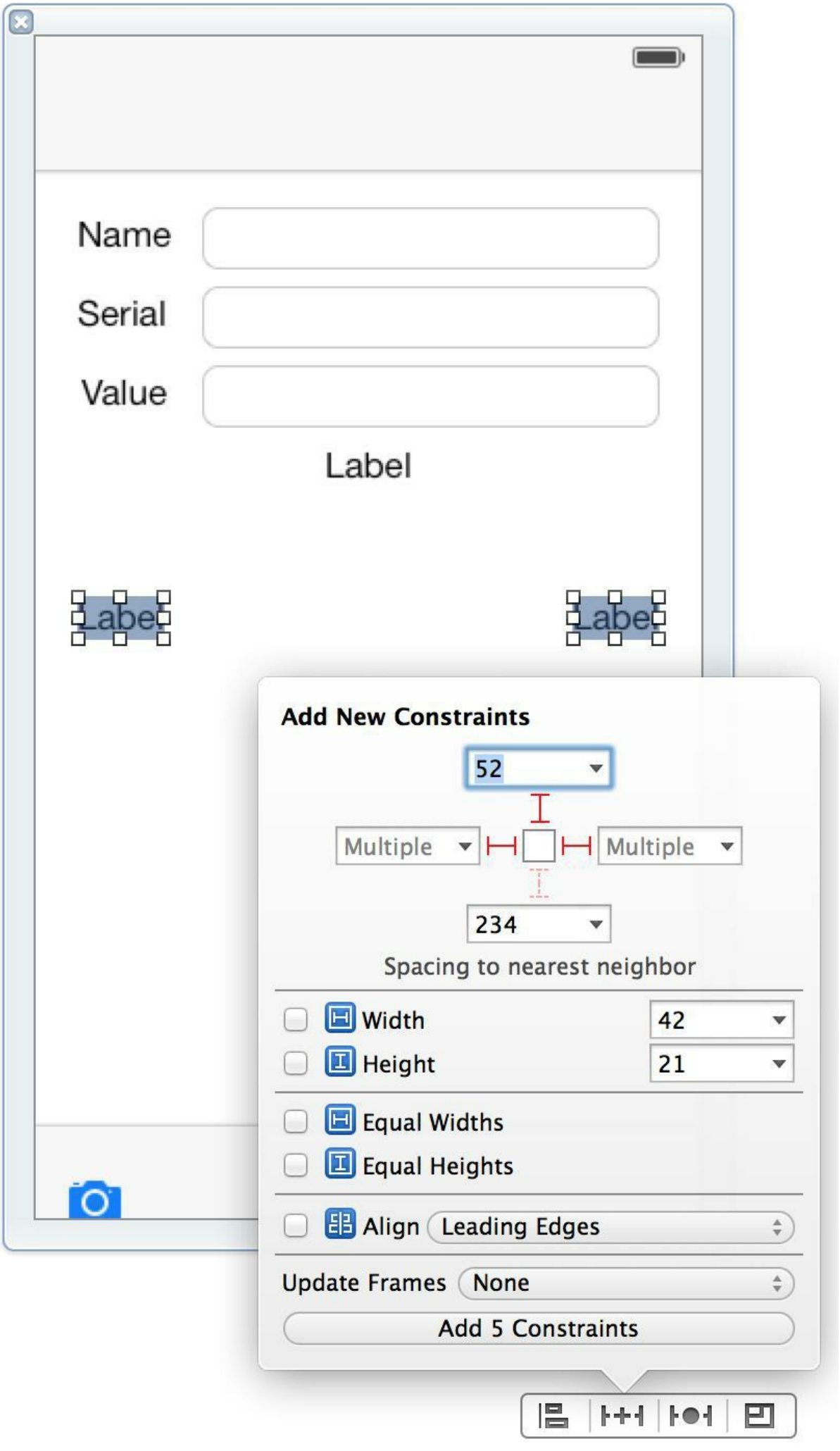

Name
Serial
Value
Label
Label
Label
Add New Constraints
52
Multiple
Multiple
234
Spacing to nearest neighbor
Width
42
Height
21
Equal Widths
Equal Heights
Align
Leading Edges
Update Frames
None
Add 5 Constraints

图15-17 同时为两个标签添加约束

读者可能会问，为两个标签各添加了三个方向上的约束，为什么按钮提示添加5个约束（Add 5 Constraints），而不是6个呢？这是因为，左侧标签右边方向的约束和右侧标签左边方向的约束是同一个约束，Interface Builder会自动识别这类重复约束并只会添加一次。

如果现在构建并运行应用，BNRDetailViewController的view在iPhone中的显示效果很好，但是在iPad中，其中一个标签会比另一个标签更宽，如图15-18所示。


Carrier
12:55 PM
100%
Homepwner
Rusty Spork
Name
Rusty Spork
Serial
8Q2U8
Value
73
Nov 23, 2013
Label
Label

图15-18 在iPad中，其中一个标签比另一个更宽

这两个标签目前缺少约束，自动布局系统无法确定它们的所有布局属性，只能在应用运行时根据其他约束推测某些布局属性的值，因此BNRDetailViewController的view在iPad上没有按照期望的方式正确布局。下面介绍如何使用UIView的两个方法：hasAmbiguousLayout和exerciseAmbiguousLayout来调试这类问题。

打开BNRDetailViewController.m，覆盖viewDidLayoutSubviews方法，检查其view中是否存在有歧义布局的子视图。

```
- (void)viewDidLayoutSubviews

{

for (UIView *subview in self.view.subviews) {

if([subview hasAmbiguousLayout])

NSLog(@"AMBIGUOUS: %@", subview);

}

}
```

当UIViewController的view首次出现在屏幕上或frame发生变化（例如旋转设备）时，UIViewController就会收到viewDidLayoutSubviews消息。在iPad模拟器上构建并运行应用，进入BNRDetailViewController，然后查看控制台中的输出。可以看见，两个标签存在有歧义的布局。

为了进一步知道视图缺少哪种约束，可以查看自动布局系统推测的另一种布局方式。在BNRDetailViewController.m中修改backgroundTapped:方法，向有歧义布局的子视图发送exerciseAmbiguityInLayout消息。

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

{

[self.view endEditing:YES];

for (UIView *subview in self.view.subviews) {

if ([subview hasAmbiguousLayout]) {

[subview exerciseAmbiguityInLayout];

}
```

```
    }

    }
```

构建并运行应用，进入BNRDetailViewController，然后点击视图任意位置，这时会发现宽度较窄的标签会变宽，而较宽的标签会变窄。如果再次点击，两个标签的宽度就会恢复原状（见图15-19）。


Carrier
12:56 PM
100%
Homepwner
Rusty Spork
Name
Rusty Spork
Serial
8Q2U8
Value
73
Nov 23, 2013
Label
Label

图15-19 自动布局系统推测的另一种布局方式

由于两个标签的宽度都没有添加约束，因此自动布局系统提供了两种布局效果，点击视图就会在这两种效果之间切换。为了消除歧义，需要为其中一个标签添加宽度约束。只要确定了一个标签的宽度，自动布局系统就能同时确定另一个标签的宽度。这里有一种很好的解决方案：让两个标签的宽度保持相等。

打开BNRDetailViewController.xib，按住Control键，将其中一个标签拖曳到另一个标签，然后选择Equal Widths（宽度相等）。在iPad上构建并运行应用，会发现两个标签宽度相等，控制台中也不会输出有歧义布局的子视图信息了。点击背景，界面也不会再发生变化。

现在界面已经没有约束错误了，自动布局系统为所有视图都定义了对齐矩形，界面只有唯一的布局效果。

在BNRDetailViewController.xib中，选中并删除用于演示约束问题的两个标签。

请读者记住，exerciseAmbiguityInLayout方法仅仅是用来调试约束问题的工具，用来查看自动布局系统在有歧义布局情况下的各种布局效果——发布应用时，不要使用该方法。

在BNRDetailViewController.m中，删除backgroundTapped:中调用exercise- AmbiguityInLayout的代码以及viewDidLayoutSubviews。

```
- (void)viewDidLayoutSubviews
{
    for (UIView *subview in self.view.subviews) {
        if([subview hasAmbiguousLayout])
            NSLog(@"AMBIGUOUS: %@", subview);
    }
}
- (IBAction)backgroundTapped:(id)sender
{
[self.view endEditing:YES];
for (UIView *subview in self.view.subviews) {
  if ([subview hasAmbiguousLayout]) {
      [subview exerciseAmbiguityInLayout];
```

```
    }
}
}
```

无法满足的约束

如果为视图添加了不必要的约束，就可能造成多个约束之间发生冲突，自动布局系统无法同时满足这些约束。下面在BNRDetailViewController中演示这类问题。

打开BNRDetailViewController.xib，选**中**date标签，**然后在属性检视面板中将背景颜色改为浅灰色，以便查看其**frame。**接下来在**Pin**菜单中限定**标签**的宽度始终为当前值。**

在iPhone**中构建并运行应用，没有**问题。**再切换到**iPad，**界面看起来也没有**问题，**但是控制台会输出类似以下内容**:

```
Unable to simultaneously satisfy constraints.

Probably at least one of the constraints in the following list is one you don't want.

Try this:（1）look at each constraint and try to figure out which you don't expect;

        （2）find the code that added the unwanted constraint or constraints and fix it.

（Note: If you're seeing NSAutoresizingMaskLayoutConstraints that you don't understand,

refer to the documentation for the UIView property

translatesAutoresizingMaskIntoConstraints）

(

“<NSLayoutConstraint:0xa333da0 H:[UILabel:0xa333ca0（280）]>”,

“<NSLayoutConstraint:0xa394500 H:[UILabel:0xa333ca0]-（20）-|

（Names: '|':UIControl:0xa38cd80 ）>”,

“<NSLayoutConstraint:0xa394530 H:|-（20）-[UILabel:0xa333ca0]

（Names: '|':UIControl:0xa38cd80 ）>”,

“<NSAutoresizingMaskLayoutConstraint:0xa3a1a70 h=-&-

v=-&- UIControl:0xa38cd80.width ==

_UIParallaxDimmingView:0xa37b140.width>”,
```

“<NSAutoresizingMaskLayoutConstraint:0xa3a21d0 h=--&

v=--& H:[_UIParallaxDimmingView:0xa37b140(768)]>”

)

Will attempt to recover by breaking constraint

<NSLayoutConstraint:0xa333da0 H:[UILabel:0xa333ca0(280)]>

Break on objc_exception_throw to catch this in the debugger.

The methods in the UIConstraintBasedLayoutDebugging category on UIView listed

in <UIKit/UIView.h> may also be helpful.

首先，Xcode提示“unable to simultaneously satisfy constraints（无法同时满足所有约束）”并给出了两条建议：

（1）look at each constraint and try to figure out which you don't expect;（在界面中查看是否添加了不需要的约束；）

（2）find the code that added the unwanted constraint or constraints and fix it.（查看代码中是否添加了不需要的约束并删除相应代码。）

接下来是与问题相关的所有约束，最后Xcode会报告忽略了哪些约束，以便解决冲突。在这里，Xcode忽略了date标签的宽度约束。

下面请读者思考约束冲突的原因。之前为date标签添加了左边和右边两个方向上的约束，现在又限定其宽度为固定值，因此产生了冲突。解决方法是删除左边约束、右边约束和宽度约束中的任意一个。

在BNRDetailViewController.xib中删除date标签的宽度约束，然后设置其背景颜色为透明（Clear Color）。

视图位置错误

如果视图在XIB文件中的frame与自身约束不一致，就会发生视图位置错误。视图位置错误是指视图在运行时的frame与其在画布中的frame不同。下面仍然在BNRDetailViewController中演示这类问题。

选中date标签并将其向下拖曳一定的距离，这时界面应该类似图15-20。

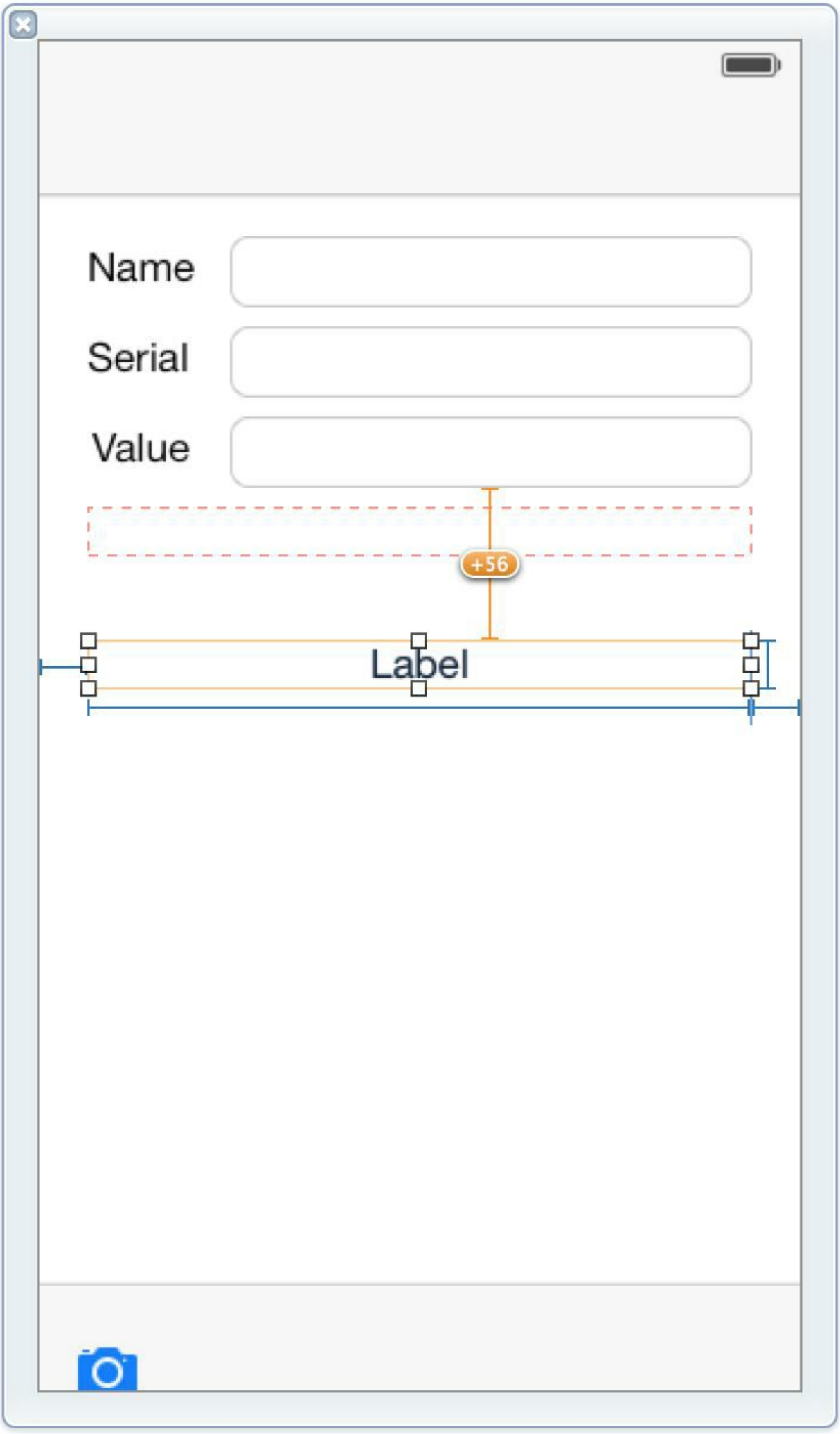

Name
Serial
Value
+56
Label

图15-20 位置错误的视图

标签原来的位置会出现一个橘红色虚线边框的矩形，这是标签在运行时的frame。自动布局系统在运行时会根据约束将标签移动到该矩形的位置，而不是刚才拖曳后的位置。

这类错误有两种解决方案，取决于视图在画布上的frame是否符合要求。如果要求视图在运行时的frame与画布上的frame相同，就修改视图的约束，匹配当前frame；反之，就修改视图的大小或位置匹配当前约束。对于date标签，画布上的frame不符合要求，需要移动date标签匹配当前约束。

选中date标签，然后在画布右下角的约束菜单中点击图标，显示Resolve Auto Layout Issues（解决自动布局问题）菜单，如图15-21所示。

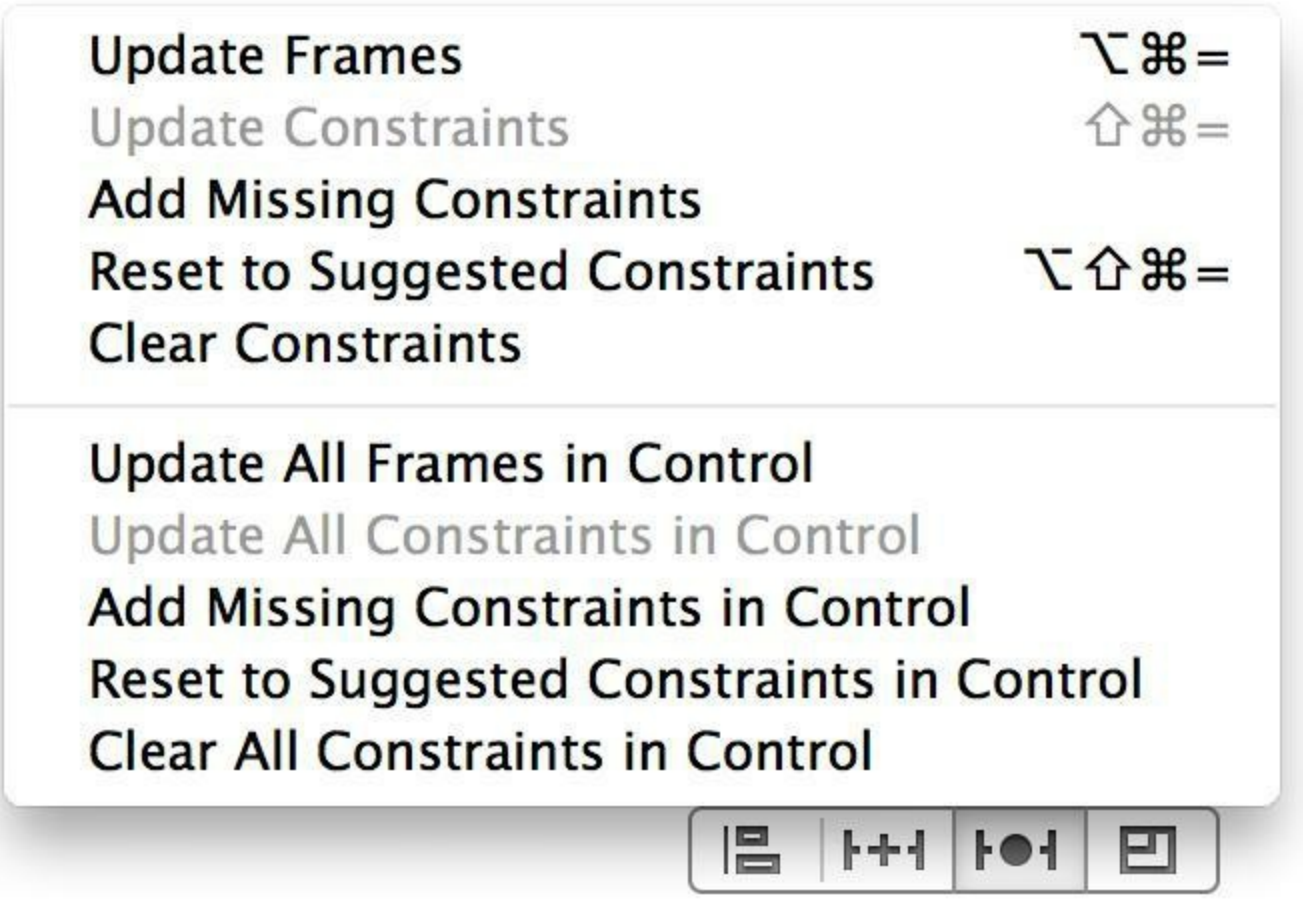


图15-21 Resolve Auto Layout Issues菜单

在菜单顶部选择Update Frames（更新frame属性），date标签会恢复原来位置，与当前约束相匹配。

相反，如果需要修改约束匹配当前frame，就选择Update Constraints（更新约束）。

下半部分菜单与上半部分的功能基本相同，只是上半部分菜单仅仅会操作选中的视图，而下半部分菜单会操作界面上的所有视图。

15.5 初级练习：打造完美界面

首先打开Resolve Auto Layout Issues菜单，选择Clear All Constraints in Control（清除Control视图中的所有约束），然后在iPad模拟器上构建并运行应用，进入详细界面，重新回顾界面在iPad上的布局问题（见图15-4），最后加回需要的约束。

在添加约束的过程中，读者可以尝试不同的添加方式（约束菜单或拖曳），还可以尝试一次添加多个约束，一次为多个视图添加约束。如果Interface Builder显示了警告或错误，尝试使用之前介绍的方式调试布局问题，确保界面可以完美显示。

15.6 中级练习：通用化Quiz

首先将Quiz改为通用应用，不要添加任何约束，直接在iPad模拟器上构建并运行应用。这时界面应该类似图15-22。

What is 7+7?

Show Question

14

Show Answer

图15-22 Resolve Auto Layout Issues菜单

接下来请读者思考界面在iPad上应该如何布局，然后打开BNRQuizView- Controller.xib添加需要的约束，实现期望的布局方式。

15.7 深入学习：使用_autolayoutTrace方法调试约束问题

本章之前通过遍历并依次发送hasAmbiguousLayout消息的方式来查找有歧义布局的视图。如果这些视图都是视图控制器view的一级子视图，那么遍历一次就可以找出全部视图，但是，如果这些视图中又包含复杂的视图层次结构，就应该使用另一种方法。

UIWindow有一个名为_autolayoutTrace的私有实例方法，该方法返回一个表示UIWindow中整个视图层次结构的字符串。对于有歧义布局的视图，_autolayoutTrace会使用AMBIGUOUS LAYOUT（有歧义的布局）标记出来。

使用该方法的最好方式是在显示视图的代码（如视图控制器的viewWillAppear:方法）中设置一个断点，当程序在断点处停下来之后，在控制台中输入以下代码，然后按下Enter键：

```
(lldb) po [[UIWindow keyWindow] _autolayoutTrace]
```

如果应用界面与期望的布局方式不一致，同时也无法确定问题原因，就可以使用该方法找出有歧义布局的视图。

15.8 深入学习：使用多个XIB文件

如果一个视图控制器需要针对iPhone和iPad显示完全不同的界面，则可以创建两个独立的XIB文件，让视图控制器在运行时根据设备类型加载相应的XIB文件。第6章中介绍过XIB文件的命名规则，针对iPhone和iPad的XIB文件需要在类名后加上对应的后缀：

BNRDetailViewController~iphone.xib

BNRDetailViewController~ipad.xib

现在，BNRDetailViewController可以在运行时根据设备类型自动加载后缀为~iphone或~ipad的XIB文件。

请注意，这种方案并不能替代自动布局系统。读者仍然需要在两个XIB文件中为视图添加约束。多个XIB文件只能解决设备屏幕尺寸问题，而自动布局系统还可以解决设备语言、偏好字体大小和设备方向问题。

第16章 在代码中使用自动布局

本章将介绍如何在代码中使用自动布局系统。第15章提到过，Apple建议开发者尽可能使用Interface Builder添加约束。但是，如果视图是在代码中创建的，就要通过代码添加约束。

下面就通过代码重新创建UIImageView对象，然后在BNRDetailViewController的viewDidLoad中为该对象添加约束。

本书第6章通过覆盖loadView方法，在代码中为UIViewController创建视图。通常，如果是创建整个视图层次结构及所有视图约束，就覆盖loadView方法；如果只是向通过NIB文件创建的视图层次结构中添加一个视图或约束，就覆盖viewDidLoad。

在BNRDetailViewController.m中覆盖viewDidLoad方法，创建一个UIImageView对象，并赋给imageView属性，代码如下：

```
- (void)viewDidLoad

{

[super viewDidLoad];

UIImageView *iv = [[UIImageView alloc] initWithImage:nil];

// 设置UIImageView对象的内容缩放模式

iv.contentMode = UIViewContentModeScaleAspectFit;

// 告诉自动布局系统不要将自动缩放掩码转换为约束

iv.translatesAutoresizingMaskIntoConstraints = NO;

// 将UIImageView对象添加到view上

[self.view addSubview:iv];

// 将UIImageView对象赋给imageView属性

self.imageView = iv;

}
```

设置translatesAutoresizingMaskIntoConstraints为NO的那行代码是为了解决约束与旧布局方式的冲突。在Apple引入自动布局系统之前，iOS一直根据自动缩放掩码（autoresizing masks）缩放视图，以适配不同大小的屏幕。

每一个视图对象都有自动缩放掩码，默认情况下，视图会将自动缩放掩码转换为对应的约

束，这类约束经常会与手动添加的约束产生冲突。因此，必须手动禁用这类自动转换（本章最后一节会详细介绍自动布局系统与自动缩放掩码）。

下面请读者思考imageView的布局方式。首先，imageView的宽度应该与屏幕宽度相同；其次，该对象应该与位于其上方的dateLabel的距离保持标准间距8点；同样，该对象应该与位于其下方的toolbar的距离也保持8点。下面将以上布局方式翻译成约束：

- 左边、右边与父视图的距离都是0点。
- 顶边与dateLabel的距离是8点。
- 底边与toolbar的距离是8点。

Apple建议使用一种特殊的语法在代码中创建约束，称为视觉化格式语言（visual format language，VFL）。下一节就会介绍如何使用视觉化格式语言为imageView添加约束。但是，并不是所有的约束都可以通过视觉化格式语言描述，添加这类约束需要通过另一种方式，本章最后会介绍这种方式。

16.1 视觉化格式语言

视觉化格式语言定义了一系列使用字符串描述约束的象形语法，而这类字符串称为视觉化格式字符串（visual format string）。一个视觉化格式字符串无法同时描述不同方向上的约束，但是可以描述一个方向上的多个约束。因此，为imageView添加约束时，需要使用两个视觉化格式字符串，分别用于描述垂直和水平两个方向上的约束。

首先是描述水平间距的视觉化格式字符串：

@“H:|-0-[imageView]-0-|”

“H:”表示约束的方向是水平（horizontal）；视图需要写在方括号中（“[]”）；“|”表示其父视图。以上字符串描述的约束是：imageView的左边和右边与父视图的距离都是0点。

在视觉化格式语言中，0及其连接符可以省略不写，例如，上述字符串也可以写成：

@“H:|[imageView]|”

与水平约束类似，imageView的垂直约束可以写成：

@“V:[dateLabel]-8-[imageView]-8-[toolbar]”

注意，为了将垂直方向上的约束写在水平排列的字符串中，视觉化格式语言规定，在垂直方向上，字符串的左边表示顶边、右边表示底边。因此，以上字符串描述的约束是：imageView的顶边与dateLabel的距离是8点，其底边与toolbar的距离也是8点。

下面介绍使用视觉化格式字符串描述更复杂的约束。假设有两个UIImageView对象，需要在水平方向添加以下约束：

•两个UIImageView对象的水平间距始终保持10点。

•左边的UIImageView对象与父视图的左边距始终保持20点。

•右边的UIImageView对象与父视图的右边距始终保持20点。

相应的视觉化格式字符串如下：

@“H:|-20-[imageViewLeft]-10-[imageViewRight]-20-|”

如果要限定视图尺寸，则可以在视图后面添加一个括号，然后在括号中填入等号和需要限定的数值。注意，如果是水平约束，则该数值表示宽度；如果是垂直约束则表示高度。例如，以下视觉化格式字符串会限定someView的高度是50点。

@“V:[someView(==50)]”

16.2 创建约束

在代码中，约束是NSLayoutConstraint类的对象，添加约束时，需要先创建NSLayoutConstraint对象，再将其添加到视图对象中。在XIB文件中创建并添加约束只需要一步就可以完成，但是在代码中，创建和添加约束需要分为两个不同的步骤。

NSLayoutConstraint提供了一个类方法，可以根据视觉化格式字符串创建约束：

+（NSArray *）constraintsWithVisualFormat:（NSString *）format

options:（NSLayoutFormatOptions）opts

metrics:（NSDictionary *）metrics

views:（NSDictionary *）views

该方法返回一个NSLayoutConstraint对象数组——视觉化格式字符串可能同时创建多个约束。其中，第一个参数是视觉化格式字符串，第二、三个参数可以忽略，通常可以分别传入0和nil。第四个参数是视图名称字典，字典中包含视觉化格式字符串中需要使用的视图对象名称。例如，在下列的视觉化格式字符串中，包含imageView、dateLabel和toolbar三个视图对象名称，请注意，它们仅仅是一个字符串形式的名称而已。

@“H:|-[imageView]-|”

@“V:[dateLabel]-[imageView]-[toolbar]”

为了让自动布局系统知道这些名称所表示的具体视图对象，就需要通过视图名称字典将名称与视图对象关联起来。例如，字符串@“imageView”表示的是self.imageView。

在BNRDetailViewController.m中修改viewDidLoad方法，创建视图名称字典：

[self.view addSubview:iv];

self.imageView = iv;

NSDictionary *nameMap = @{ @“imageView” : self.imageView,

@“dateLabel” : self.dateLabel,

@“toolbar” : self.toolbar };

}

根据Apple命名规范，应该使用属性的名称或实例变量的名称作为视图对象的键。视觉化格式字符串将使用字典中的键表示相应的视图对象。

接下来为imageView添加水平和垂直两个方向上的约束，代码如下：

```
- (void)viewDidLoad
{
[super viewDidLoad];
…
NSDictionary *nameMap = @{ @“imageView” : self.imageView,
@“dateLabel” : self.dateLabel,
@“toolbar” : self.toolbar };
// imageView的左边和右边与父视图的距离都是0点
NSArray *horizontalConstraints =
[NSLayoutConstraint constraintsWithVisualFormat:
@“H:|-0-[imageView]-0-|”
options:0
metrics:nil
views:nameMap];
// imageView的顶边与dateLabel的距离是8点，底边与toolbar的距离也是8点
NSArray *verticalConstraints =
[NSLayoutConstraint constraintsWithVisualFormat:
@“V:[dateLabel]-[imageView]-[toolbar]”
options:0
metrics:nil
views:nameMap];
}
```

16.3 添加约束

现在代码中已经创建了两个NSLayoutConstraint对象数组，下面需要将它们添加到某个视图中。UIView有一个可以同时添加多个约束的实例方法：

- (void)addConstraints:(NSArray *)constraints

那么，如何判断约束应该添加到哪个视图中呢？以下是判定法则：

•如果约束同时对多个父视图相同的视图起作用（例如图16-1中的约束“A”），那么约束应该添加到它们的父视图中。

•如果约束只对某个视图自身起作用（例如约束“B”），那么约束应该添加到该视图中。

•如果约束同时对多个父视图不同的视图起作用（例如图16-1中的约束“C”），但是这些视图在层次结构中有共同的祖先视图，那么约束应该添加到它们最近一级的祖先视图中。

•如果约束同时对某个视图及其父视图起作用（例如约束“D”），那么约束应该添加到其父视图中。

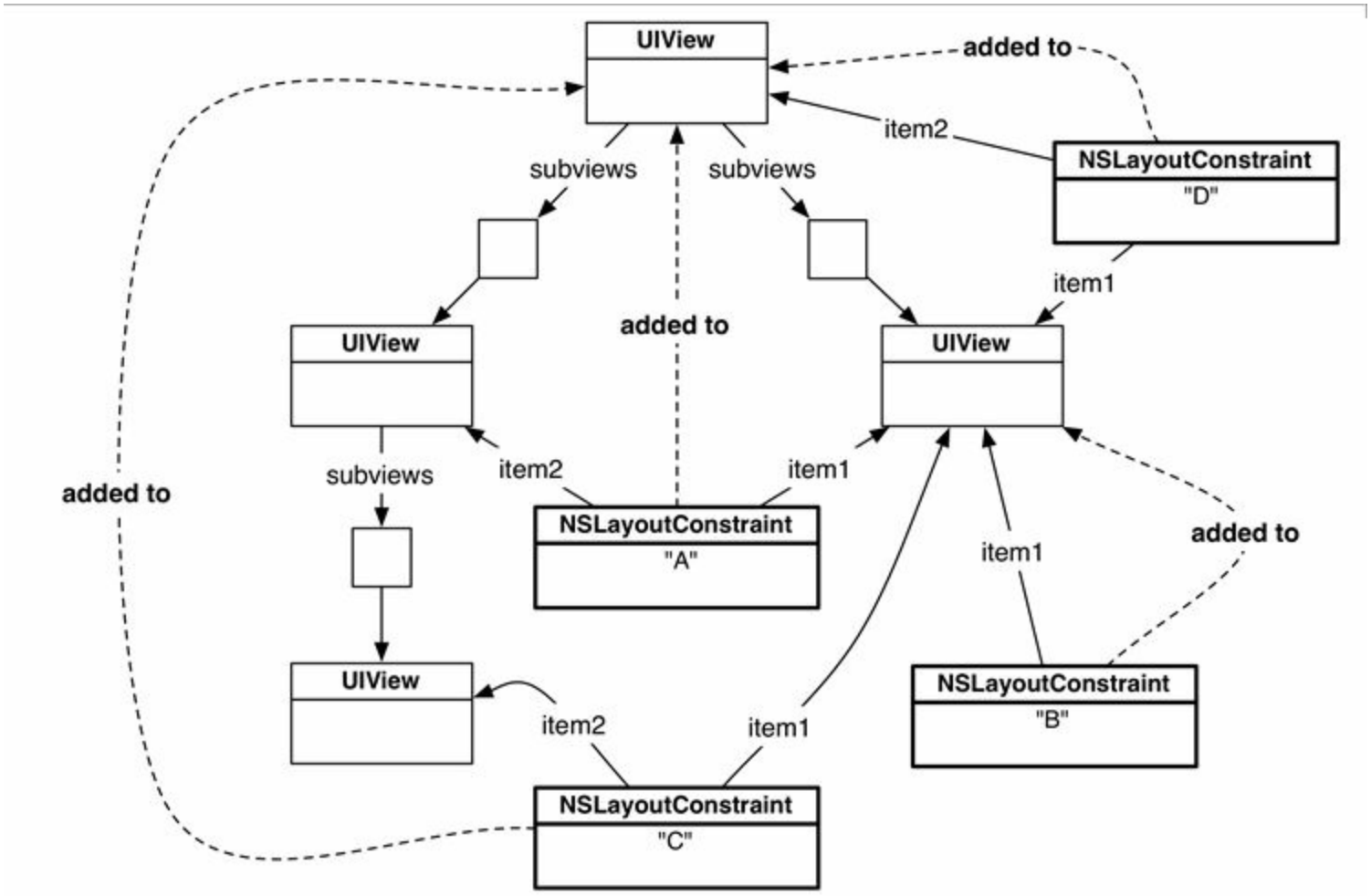


图16-1 视图层次结构与约束之间的关系

在imageView的水平方向上，约束同时对imageView及其父视图起作用，因此约束应该添加

到imageView的父视图——BNRDetailViewController的view中。

而在垂直方向上，约束同时对imageView、dateLabel和toolbar起作用，这三个视图的父视图都是BNRDetailViewController的view，因此约束应添加到它们的父视图中。

在BNRDetailViewController.m中修改viewDidLoad方法，将两个NSLayout- Constraint对象数组添加到BNRDetailViewController的view中，代码如下：

```
…

NSArray *verticalConstraints =

[NSLayoutConstraint constraintsWithVisualFormat:

@“V:[dateLabel]-[imageView]-[toolbar]”

options:0

metrics:nil

views:nameMap];

[self.view addConstraints:horizontalConstraints];

[self.view addConstraints:verticalConstraints];

}
```

构建并运行应用。创建一个BNRItem对象，然后为其设置一张图片。这时，根据所设置的图片大小，详细界面看起来可能正常，也可能有问题。如果图片比较小，那么详细界面可能类似于图16-2。

Carrier 6:56 PM 100%

Homepwner

Rusty Spork

Name

Serial

Value 73

Dec 2, 2013

图16-2 选择小尺寸图片导致的布局问题

为了理解这种情况发生的原因，首先要介绍固有内容大小（intrinsic content size）的概念。

（请注意，目前自动布局系统存在一个bug，可能导致valueField的约束出现问题。对于基准线约束，如之前添加的使valueField与valueLabel沿基准线对齐，有时并不起作用。如果读者的valueField有问题，请删除其基准线约束，然后添加一个垂直方向的约束。）

16.4 固有内容大小

固有内容大小的含义：视图要显示的实际内容区域大小。例如，UILabel的固有内容大小是由需要显示的文字数量决定的。而UIImageView的固有内容大小则是所显示图片的尺寸。

自动布局系统会根据固有内容大小为视图添加相应的约束，与其他约束不同，这类约束有两个优先级属性，分别是内容放大优先级（content hugging priority）和内容缩小优先级（content compression resistance priority）。

优先级属性在水平方向和垂直方向可以有不同的数值。因此，可以为视图的宽和高设置不同的优先级，每个视图都有四个优先级数值。

首先介绍如何在Interface Builder中查看并修改这些优先级。重新打开BNRDetailView-Controller.xib，按住Shift，在画布中同时选中三个UITextField对象。然后点击图标，打开大小检视面板（size inspector），找到Content Hugging Priority和Content Compression Resistance Priority部分（见图16-3）。

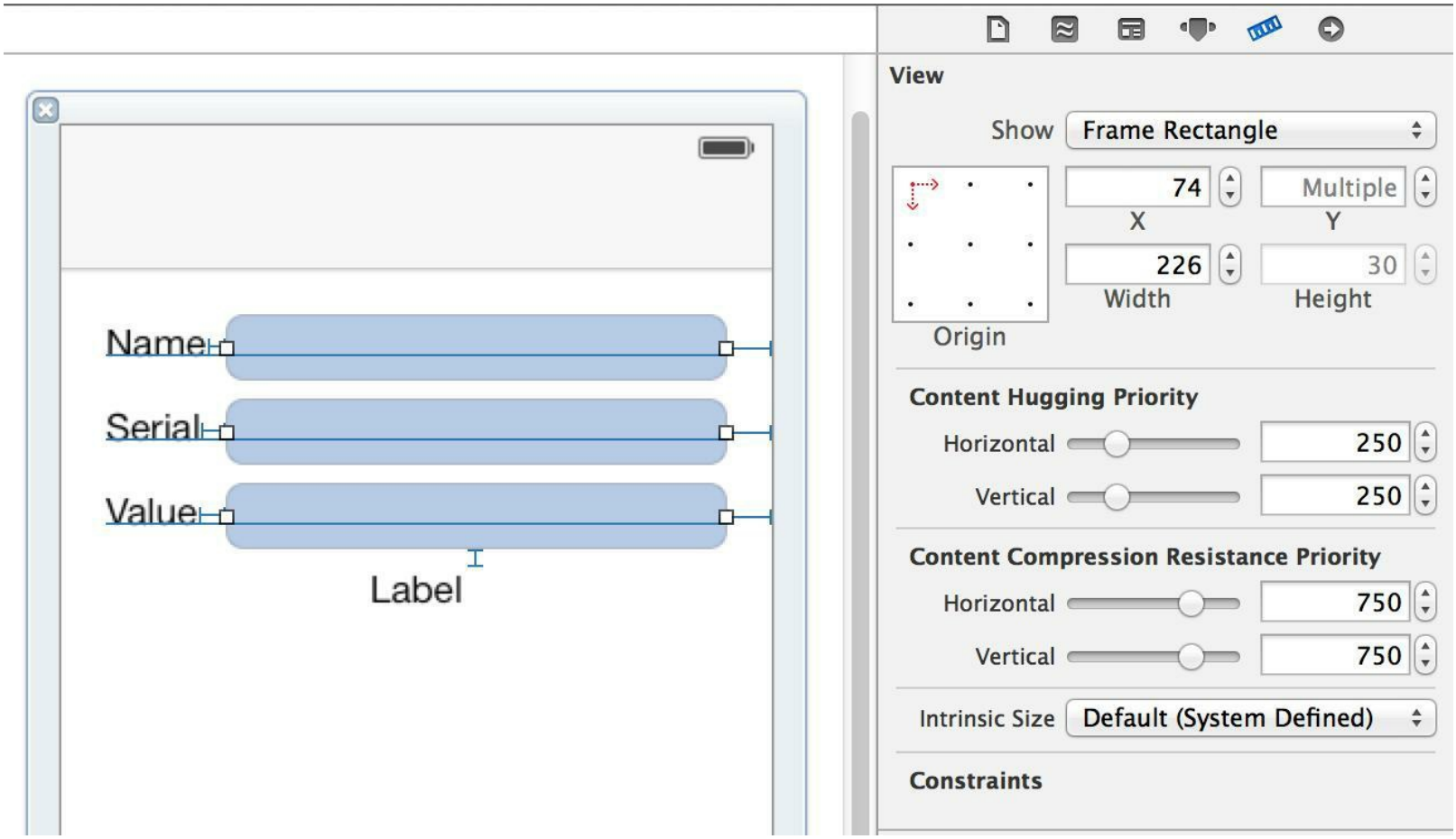


图16-3 在Interface Builder中查看优先级

可以看见，目前这些优先级的数值并不是1000，其中Content Hugging Priority的Vertical是250，而imageView的是251。因此，如果用户选择了一张小尺寸图片，自动布局系统会增加UITextField对象的高度，使高度超出UITextField对象的固有内容大小。为了解决该问题，需要将imageView垂直方向的优先级设置为比其他视图低的数值。

下面介绍如何通过代码修改视图的优先级属性。打开BNRDetailViewController.m，修改viewDidLoad方法，降低imageView垂直方向的优先级，代码如下：

```
- (void)viewDidLoad

{

[super viewDidLoad];

UIImageView *iv = [[UIImageView alloc] initWithImage:nil];

// 设置UIImageView对象的内容缩放模式

iv.contentMode = UIViewContentModeScaleAspectFit;

// 告诉自动布局系统不要将自动缩放掩码转换为约束

iv.translatesAutoresizingMaskIntoConstraints = NO;

// 将UIImageView对象添加到view上

[self.view addSubview:iv];

// 将UIImageView对象赋给imageView属性

self.imageView = iv;

// 将imageView垂直方向的优先级设置为比其他视图低的数值

[self.imageView setContentHuggingPriority:200

forAxis:UILayoutConstraintAxisVertical];

[self.imageView setContentCompressionResistancePriority:700

forAxis:UILayoutConstraintAxisVertical];

}
```

构建并运行应用，选择一张小尺寸图片，这次自动布局系统不会再修改UITextField对象的高度。

16.5 另一种添加方式

虽然视觉化格式语言可以形象地描述大部分约束，但是，如果某个约束是根据另一个约束计算而来的，就无法使用视觉化格式语言。例如，假设需要将dateLabel的高度设置为nameLabel高度的两倍，就必须使用NSLayoutConstraint的另一个工厂方法：

+（id）constraintWithItem:（id）view1

attribute:（NSLayoutAttribute）attr1

relatedBy:（NSLayoutRelation）relation

toItem:（id）view2

attribute:（NSLayoutAttribute）attr2

multiplier:（CGFloat）multiplier

constant:（CGFloat）c

与视觉化格式语言不同，该方法只会创建一个约束，该约束用于限定两个布局属性之间的关系。在该方法的参数中，multiplier是乘数，constant是常量，用于计算view1布局属性的值，稍后会给出计算表达式。

现在请读者打开NSLayoutConstraint头文件，找到NSLayoutAttribute枚举，其中包括所有布局属性常量：

•NSLayoutAttributeLeft

•NSLayoutAttributeRight

•NSLayoutAttributeTop

•NSLayoutAttributeBottom

•NSLayoutAttributeWidth

•NSLayoutAttributeHeight

•NSLayoutAttributeBaseline

•NSLayoutAttributeCenterX

•NSLayoutAttributeCenterY

•NSLayoutAttributeLeading

•NSLayoutAttributeTrailing

下面演示如何使用该方法：如果需要将imageView的宽度设置为自身高度的1.5倍，可以编写如下代码（读者不要将这段代码添加到项目中，否则会与其他约束产生冲突）：

```
NSLayoutConstraint *aspectConstraint =

[NSLayoutConstraint constraintWithItem:self.imageView

attribute:NSLayoutAttributeWidth

relatedBy:NSLayoutRelationEqual

toItem:self.imageView

attribute:NSLayoutAttributeHeight

multiplier:1.5

constant:0.0];
```

为了理解该方法所描述的约束，图16-4是该方法中各个参数的计算表达式，自动布局系统会根据该表达式为第一个参数所表示的视图计算布局属性的值，再创建相应的NSLayoutConstraint对象。

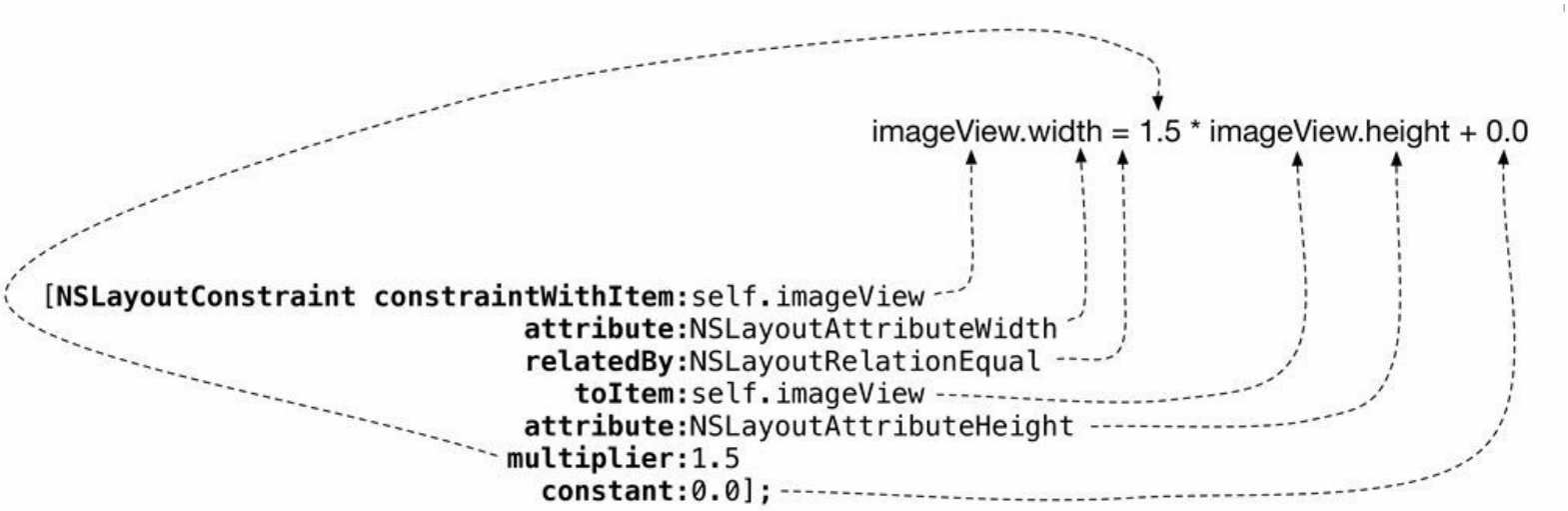


图16-4 NSLayoutConstraint表达式

由于该方法只创建了一个约束，因此需要使用UIView添加单个约束的实例方法：

```
- (void)addConstraint:(NSLayoutConstraint *)constraint
```

之前介绍过判断约束应该添加到哪个视图中的四条判定法则，而该方法符合第二条法则。该方法创建的约束只对第一个参数所表示的视图（这里是imageView）起作用，因此约束应该添加到该视图中。

对于aspectConstraint, 应该将其添加到imageView中:

```
[self.imageView addConstraint:aspectConstraint];
```

16.6 深入学习：NSAutoresizingMaskLayoutConstraint

在Apple引入自动布局系统之前，iOS应用一直使用自动缩放掩码管理视图布局，视图只具有相对于父视图的自动缩放掩码，无法设置与兄弟视图之间的布局关系。

默认情况下，视图会将自动缩放掩码转换为对应的约束，这类约束经常会与手动添加的约束产生冲突，下面就演示这类冲突。

在viewDidLoad中，注释掉禁用转换自动缩放掩码的代码：

```
// 设置UIImageView对象的内容缩放模式

iv.contentMode = UIViewContentModeScaleAspectFit;

// 告诉自动布局系统不要将自动缩放掩码转换为约束

// iv.translatesAutoresizingMaskIntoConstraints = NO;

// 将UIImageView对象添加到view上

[self.view addSubview:iv];
```

现在UIImageView对象会将自动缩放掩码转换为约束。构建并运行应用，然后进入详细界面。这时控制台会输出布局问题和解决建议。

```
Unable to simultaneously satisfy constraints.

Probably at least one of the constraints in the following list is one you don't

want. Try this:（1）look at each constraint and try to figure out which you don't

expect;（2）find the code that added the unwanted constraint or constraints and

fix it.（Note: If you're seeing NSAutoresizingMaskLayoutConstraints that you don't understand, refer to the documentation for the UIView property

translatesAutoresizingMaskIntoConstraints）

（

“”,

“<NSLayoutConstraint:0x9153ee0

H:|-（20）-[UILabel:0x9149f00]（Names: '|':UIControl:0x91496e0 ）>”,

“<NSLayoutConstraint:0x9153fa0
```

UILabel:0x9149970.leading == UILabel:0x9149f00.leading>",

"<NSLayoutConstraint:0x91540c0

UILabel:0x914a1e0.leading == UILabel:0x9149970.leading>",

"<NSLayoutConstraint:0x9154420

H:[UITextField:0x914fe20]-(20)-|

(Names: '|':UIControl:0x91496e0)>",

"<NSLayoutConstraint:0x9154450

H:[UILabel:0x914a1e0]-(12)-[UITextField:0x914fe20]>",

"<NSLayoutConstraint:0x912f5a0 (Names: '|':UIControl:0x91496e0)>",

H:|-(NSSpace(20))-[UIImageView:0x91524d0

"<NSLayoutConstraint:0x91452a0 (Names: '|':UIControl:0x91496e0)>",

H:[UIImageView:0x91524d0]-(NSSpace(20))-|

"<NSAutoresizingMaskLayoutConstraint:0x905f130

h=--& v=--& UIImageView:0x91524d0.midX ==>"

)

Will attempt to recover by breaking constraint

<NSLayoutConstraint:0x914a2e0 H:[UILabel:0x914a1e0(42)]>

第15章介绍过，以上输出表明多个约束之间发生了冲突，Xcode无法同时满足所有约束。其中，与问题相关的所有约束都会以固定格式依次列举出来，例如，

<NSLayoutConstraint:0x9153fa0

UILabel:0x9149970.leading == UILabel:0x9149f00.leading>

以上格式首先是约束对象的类名和内存地址，然后是约束的作用，在本例中，位于0x9153fa0的约束限定0x9153fa0和0x9149970两个UILabel对象左对齐。

可以注意到，按照以上格式，除最后一个约束外，其余约束的类名都是NSLayoutConstraint。最后一个约束是由UIImageView对象的自动缩放掩码转换而来的，所以类名是NSAutoresizingMaskLayoutConstraint。

最后是Xcode所忽略的约束。可以看到，Xcode忽略的是之前手动为UILabel对象添加的宽度约束，而不是NSAutoresizingMaskLayoutConstraint对象，所以导致界面布局与期望不同。

现在读者可以理解设置translatesAutoresizingMaskIntoConstraints属性为NO的原因了，其作用就是避免自动布局系统生成与其他约束产生冲突的NSAutoresizingMask- LayoutConstraint对象。

最后请读者恢复viewDidLoad中的代码：

```
// 设置UIImageView对象的内容缩放模式

iv.contentMode = UIViewContentModeScaleAspectFit;

// 告诉自动布局系统不要将自动缩放掩码转换为约束

iv.translatesAutoresizingMaskIntoConstraints = NO;

// 将UIImageView对象添加到view上

[self.view addSubview:iv];
```

第17章 自动转屏，UIPopoverController与模态视图控制器

前两章介绍了自动布局系统，使Homepwner能够根据设备屏幕的尺寸自动调整界面布局。例如，Homepwner会将UIToolbar对象始终放置在屏幕底部并保持其宽度与屏幕宽度相同。

当设计一款通用应用时，除了界面布局外，还要考虑如何充分利用各类设备的特性，满足用户在使用不同设备时的需求。同样的功能，在小屏幕设备（iPhone和iPod Touch）和大屏幕设备（iPad）中很可能有着完全不同的设计方案。

•本章将对Homepwner做出四个方面的调整，针对不同设备设计用户体验更好的功能实现方式：

•仅在iPad中，当用户倒置设备时，Homepwner可以转动界面。

•仅在iPad中，当用户点击相机按钮时，Homepwner会通过UIPopoverController显示UIImagePickerController。

•仅在iPad中，当用户创建新的BNRItem对象时，Homepwner会以模态形式显示详细界面。

•仅在iPhone中，当设备处于横排方向时，Homepwner会禁用详细界面中的相机按钮。

为了实现以上设计方案，本章会介绍如何判断设备类型并针对不同设备编写特定代码。此外，还会介绍自动转屏、UIPopoverController和模态视图控制器。

17.1 自动转屏

iOS应用具有两个意义不同的“方向”，分别是设备方向（device orientation）和界面方向（interface orientation）。

设备方向指的是设备的物理方向，包括正的竖排方向（right-side up）、倒置方向（upside down）、左旋转方向（rotated left）、右旋转方向（rotated right）、正面朝上和背面朝上。可以通过UIDevice类的orientation属性获取设备方向。

而界面方向是指用户所看到的应用界面的方向。以下是所有可能的界面方向：

UIInterfaceOrientationPortrait竖排方向，主屏幕按钮位于屏幕下方

UIInterfaceOrientationPortraitUpsideDown竖排方向，主屏幕按钮位于屏幕上方

UIInterfaceOrientationLandscapeLeft横排方向，主屏幕按钮位于屏幕右侧

UIInterfaceOrientationLandscapeRight横排方向，主屏幕按钮位于屏幕左侧

当应用的界面方向发生变化时，UIWindow对象会根据新的界面方向调整自身大小并旋转其视图层次结构。同时，视图层次结构中的所有视图对象都会根据相关约束重新调整自身布局。

打开Homepwner.xcodeproj，在iPad模拟器上构建并运行应用。

当Homepwner在iPad模拟器上运行时，可以模拟转屏操作。选中某一个BNRItem对象，进入详细界面，然后在模拟器的Hardware菜单中选择Rotate Left。这时，模拟器窗口以及应用界面会向左旋转，由于事先已经为视图添加了约束，因此Homepwner在任何方向都可以很好地显示。

当设备方向发生改变时，应用会收到新的方向信息，并根据相关设置决定是否根据新的设备方向调整界面方向。

再次点击Rotate Left选项，此时模拟器会处于倒置方向，但是界面方向并没有随之改变——原因：Homepwner不支持将界面方向设置为UIInterfaceOrientationPortraitUpsideDown。第15章在项目属性页面将Homepwner改为通用应用，本章将在相同的页面修改Homepwner所支持的界面方向。

选择Target信息中的General标签项，在展开的Deployment Info中选择iPad，可以看见Device Orientation后面的四个选项中除第二行的Upside Down外，Portrait、Landscape Left和 Landscape Right 都已选中。请读者选中Upside Down（见图17-1）。

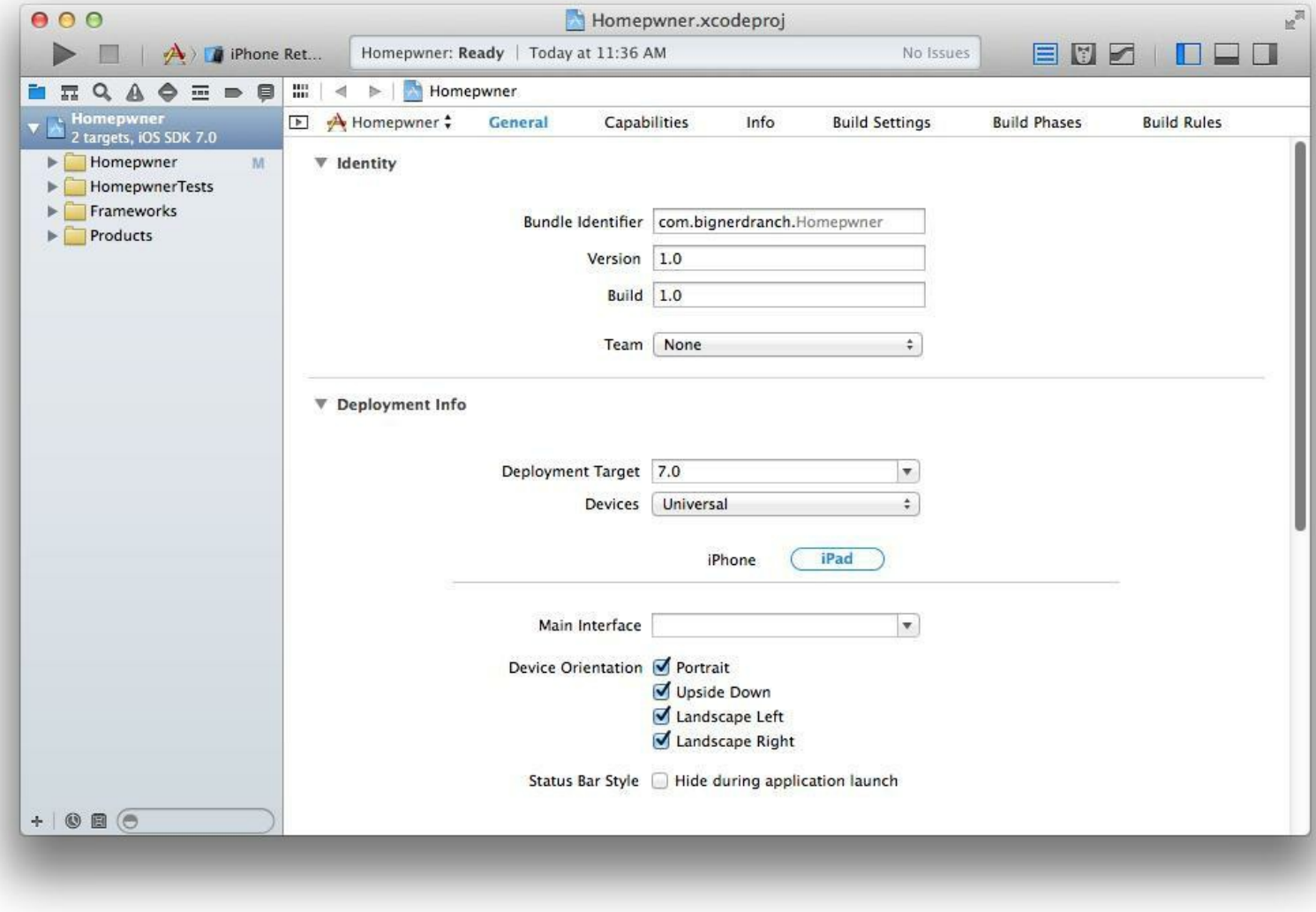


图17-1 修改Homepwner，支持Upside Down界面方向

构建并在iPad模拟器上运行应用，在Hardware菜单中将模拟器向同一个方向旋转两次。现在Homepwner可以支持倒置的界面方向了。通常，iPad应用需要支持所有界面方向，而iPhone应用需要支持除Upside Down以外的界面方向。在Deployment Info中选择iPhone，可以看到应用在iPhone或iPod Touch上不支持Upside Down。

有时可能需要使应用只支持某些特定的界面方向。例如，许多游戏应用只支持两个横排方向；许多iPhone应用只支持正的竖排方向。勾选或取消Device Orientation中的选项可以选择合适的界面方向。

实际上，不但可以设置应用的界面方向，还可以为应用中的每个UIViewController单独设置界面方向。（在Homepwner中，主要包括以模态形式显示的BNRDetaileViewController和其余包含在UINavigationController中的UIViewController。）每个UIViewController都可以在supportedInterfaceOrientations方法中返回自身支持的界面方向。当界面方向发生变化时，应用的rootViewController以及应用自身（应用的Info.plist中的Supported Interface Orientations部分）都必须支持新的界面方向，应用才能旋转至该方向。

默认情况下，UIViewController对象在iPad中支持所有界面方向，在iPhone中则支持除Upside Down以外的界面方向。如果需要修改默认支持方向，则必须在相应的UIViewController中覆盖supportedInterfaceOrientations方法，该方法的默认实现类似于以下代码：

```
- (NSUInteger)supportedInterfaceOrientations
{
if ([UIDevice currentDevice].userInterfaceIdiom
== UIUserInterfaceIdiomPad) {
return UIInterfaceOrientationMaskAll;
} else {
return UIInterfaceOrientationMaskAllButUpsideDown;
}
}
```

以上代码会根据应用当前运行的设备改变支持的界面方向。首先通过UIDevice的类方法currentDevice获取表示当前设备的对象，然后根据该对象的userInterfaceIdiom属性判断当前运行的设备是iPad还是iPhone。截至本书写作期间，该属性的值可能是UIUserInterfaceIdiomPhone或UIUserInterfaceIdiomPad。最后根据设备类型决定应用支持的界面方向。

假设应用的根视图控制器只支持两个横排方向，则可以在supportedInterface- Orientations方法中添加以下代码：

```
- (NSUInteger)supportedInterfaceOrientations
{
// 对于所有设备，只支持左横排方向和右横排方向
return UIInterfaceOrientationMaskLandscapeLeft
| UIInterfaceOrientationMaskLandscapeRight;
}
```

（如果不了解按位或“|”运算符，可以参考本章17.8节）。

在许多应用中，UINavigationController和UITabViewController是出现频率最高的两种视图控制器。UINavigationController使用继承自UIViewController的supportedInterfaceOrientations。如果需要由UINavigationController当前显示的视图控制器来决定所支持的界面方向，则可以创建UINavigationController的子类并覆盖supportedInterfaceOrientations方法，代码如下：

```
@implementation MyNavigationController
- (NSUInteger)supportedInterfaceOrientations
```

```
{

return self.topViewController.supportedInterfaceOrientations;

}

@end
```

UITabViewController则会依次检查每个标签项所支持的界面方向，然后在自身的supportedInterfaceOrientations方法中返回所有标签项都支持的界面方向。

17.2 自动转屏通告机制

当设备方向发生变化时，可能需要在视图控制器中执行一些操作。例如，Homepwner应用目前存在一个问题：当应用在iPhone中运行，并且界面处于横排方向时，BNRDetailViewController中的UIImageView对象会显得非常小，如果限制用户只能在竖排方向拍摄和浏览照片，用户体验会更好——可以隐藏UIImageView对象并禁用相机按钮。

首先，为了禁用相机按钮，需要创建一个指向相机按钮的插座变量。打开BNRDetailView-Controller.m，同时，在辅助编辑器中打开BNRDetailViewController.xib（在项目导航面板中按住Option键并单击BNRDetailViewController.xib）。

现在，按住Control键，将UIToolbar中的相机按钮拖曳到BNRDetailViewController.m的类扩展中，创建弱引用的插座变量cameraButton。Xcode会自动生成如下代码：

```
@property (weak, nonatomic) IBOutlet UIBarButtonItem *cameraButton;
```

接下来的任务是：当应用在iPhone中运行，并且界面处于横排方向时，隐藏UIImageView对象并禁用相机按钮。

如果需要在界面方向发生变化时执行某些操作，则可以在UIViewController中覆盖willAnimateRotationToInterfaceOrientation:duration:方法。在界面方向发生改变后，UIViewController会收到willAnimateRotationToInterfaceOrientation: duration:消息，消息的第一个参数是新的界面方向。

在BNRDetailViewController.m中创建一个新方法：prepareViewsForOrientation:，如果设备类型是iPhone，同时界面方向处于横排方向，就隐藏UIImageView对象并禁用相机按钮。接下来，当BNRDetailViewController的视图出现在屏幕上，以及界面方向发生变化时，调用此方法：

```
- (void)prepareViewsForOrientation:(UIInterfaceOrientation)orientation

{

// 如果是iPad，则不执行任何操作

if ([UIDevice currentDevice].userInterfaceIdiom

== UIUserInterfaceIdiomPad) {

return;

}

// 判断设备是否处于横排方向

if (UIInterfaceOrientationIsLandscape(orientation)) {

self.imageView.hidden = YES;
```

```
self.cameraButton.enabled = NO;
} else {
self.imageView.hidden = NO;
self.cameraButton.enabled = YES;
}
}
- (void)willAnimateRotationToInterfaceOrientation:
(UIInterfaceOrientation)toInterfaceOrientation
duration:(NSTimeInterval)duration
{
[self prepareViewsForOrientation:toInterfaceOrientation];
}
- (void)viewWillAppear:(BOOL)animated
{
[super viewWillAppear:animated];
UIInterfaceOrientation io =
[[UIApplication sharedApplication] statusBarOrientation];
[self prepareViewsForOrientation:io];
…
```

在iPhone模拟器上构建并运行应用。在BNRDetaileViewController中为BNRItem添加一张图片，然后将iPhone模拟器旋转至横排方向。可以发现添加的图片消失了，同时相机按钮也会变为灰色，无法点击。接下来将iPhone模拟器旋转至竖排方向，此时消失的图片会再次显示在屏幕上，相机按钮也可以点击了。如果在iPad上运行应用，图片会一直显示，相机按钮也始终可以点击。

如果在该方法中编写修改视图属性的代码（例如修改视图的frame或通过hidden显示/隐藏视图），则方法会自动为属性的变化过程添加动画效果（例如由大变小或淡入淡出）。duration参数表示动画的持续时间。如果在应用转屏时不需要对视图做处理，或者不需要使用属性变换的动画效果，也可以将该方法替换为willRotateToInterfaceOrientation: duration:。两个方法将会在

转屏时同时调用，参数的含义也都是相同的。但是，willRotateToInterfaceOrientation:duration:方法中不会自动添加任何动画效果。

此外，如果需要在转屏完成之后执行一些操作，则可以在UIViewController中覆盖didRotateFromInterfaceOrientation:方法。该方法的参数是发生转屏之前的界面方向。最后，UIViewController对象有一个interfaceOrientation属性，表示该对象的当前界面方向。

本节编写了针对iPhone的特定代码，下一节将为另一种设备添加新功能——如果应用运行在iPad中，则会通过UIPopoverController显示UIImagePickerController。

17.3 UIPopoverController

为了有效利用充足的屏幕空间，Apple针对iPad应用提供了UIPopoverController。在iPhone应用中，如果要让用户做一个选择（例如在照片库中选择一张照片），或者针对屏幕上的某些元素显示相关摘要信息（例如向用户解释某一项数据的具体含义），则可能会使用模态视图控制器或警告视图；而在iPad应用中，这些场景更适合使用UIPopoverController。对于Homepwner应用，当用户在详细界面点击相机按钮时，就可以使用UIPopover- Controller来显示UIImagePickerController。

UIPopoverController能够在一个带边框的窗口中显示一个指定的视图控制器的视图。此外，这个窗口会“悬浮”在其他界面的前面。创建UIPopoverController对象时需要设置contentViewController属性，指向需要显示的视图控制器。

本节要完成的任务为：当用户按下相机按钮时，通过UIPopoverController对象来显示UIImagePickerController对象（见图17-2）。

Homepwner

Item

Name: Item

Serial:

Value: 0

Dec 20, 2013

图17-2 UIPopoverController

在BNRDetailViewController.m的类扩展中，将BNRDetailViewController声明为遵守UIPopoverControllerDelegate协议，代码如下：

```
@interface BNRDetailViewController（）
<UINavigationControllerDelegate, UIImagePickerControllerDelegate,
UITextFieldDelegate, UIPopoverControllerDelegate>
```

再添加一个属性，用于保存UIPopoverController对象，代码如下：

```
@interface BNRDetailViewController（）
<UINavigationControllerDelegate, UIImagePickerControllerDelegate,
UITextFieldDelegate, UIPopoverControllerDelegate>
@property（strong, nonatomic）UIPopoverController *imagePickerPopover;
@property（weak, nonatomic）IBOutlet UITextField *nameField;
```

然后在takePicture:方法的末端添加以下代码：

```
imagePicker.delegate = self;
[self presentViewController:imagePicker animated:YES completion:nil];
// 显示UIImagePickerController对象
// 创建UIPopoverController对象前先检查当前设备是否是iPad
if（[UIDevice currentDevice].userInterfaceIdiom == UIUserInterfaceIdiomPad）{
// 创建UIPopoverController对象，用于显示UIImagePickerController对象
self.imagePickerPopover = [[UIPopoverController alloc]
initWithContentViewController:imagePicker];
self.imagePickerPopover.delegate = self;
// 显示UIPopoverController对象，
// sender指向的是代表相机按钮的UIBarButtonItem对象
[self.imagePickerPopover
```

```
presentPopoverFromBarButtonItem:sender

permittedArrowDirections:UIPopoverArrowDirectionAny

animated:YES];

} else {

[self presentViewController:imagePicker animated:YES completion:nil];

}

}
```

这段代码在创建UIPopoverController对象前，要先检查设备的类型，这点很重要。只有当iOS应用是在iPad系设备上运行的时候，才能创建UIPopoverController对象，否则应用会抛出异常。

针对iPad模拟器或iPad构建并运行应用。选中UITableView对象中的某一行，显示BNRDetailViewController对象的视图并按下相机按钮。Homepwner会显示一个UIPopoverController对象并显示UIImagePickerController对象的视图。从UIImagePicke-Controller对象所显示的照片中挑选一张，Homepwner会关闭UIPopoverController对象，并在BNRDetailViewController对象的视图中显示选中的照片。

触摸屏幕的其他区域可以关闭UIPopoverController对象。如果某个UIPopoverController对象是通过这种方式关闭的，那么该对象会向自己的委托对象发送popoverControllerDidDismissPopover:消息。在BNRDetailViewController.m中实现popoverControllerDidDismissPopover:，代码如下：

```
-（void）popoverControllerDidDismissPopover:

（UIPopoverController *）popoverController

{

NSLog（@“User dismissed popover”）；

self.imagePickerPopover = nil;

}
```

为了放弃针对UIPopoverController对象的拥有权，这段代码在popoverController-DidDismissPopover:中将imagePickerPopover赋为了nil。每当用户按下拍照按钮时，Homepwner都会创建一个新的UIPopoverController对象。

当用户在UIImagePickerController对象中选择一张图片后，也要关闭UIPopoverController对象。在BNRDetailViewController.m中的imagePickerController: didFinishPickingMediaWithInfo:末

尾关闭UIPopoverController对象，代码如下：

```
self.imageView.image = image;

[self dismissViewControllerAnimated:YES completion:nil];

// 判断UIPopoverController对象是否存在

if (self.imagePickerPopover) {

// 关闭UIPopoverController对象

[self.imagePickerPopover dismissPopoverAnimated:YES];

self.imagePickerPopover = nil;

} else {

// 关闭以模态形式显示的UIImagePickerController对象

[self dismissViewControllerAnimated:YES completion:nil];

}

}
```

要“主动地”关闭UIPopoverController对象，可以向需要关闭的UIPopoverController对象发送dismissPopoverAnimated:消息。如果UIPopoverController对象是通过这种方式关闭的，就不会向其委托对象发送popoverControllerDidDismissPopover:消息。为了应对这种情况，还需要在关闭UIPopoverController对象之后将其设置为nil。

以上代码有一个小问题需要修正。在用户按下拍照按钮打开UIPopoverController对象后，如果再次按下拍照按钮，Homepwner就会崩溃。这是因为当用户再次按下按钮时，触发的takePicture:会创建新的UIPopoverController对象，并将新创建的对象赋给imagePickerPopover，从而导致当前可见的UIPopoverController对象失去其最后的一个拥有方并被释放。要解决这个问题，只要在创建并显示新的UIPopoverController对象前，检查imagePickerPopover是否指向有效的UIPopoverController对象。如果是有效的，并且该对象的视图是可见的，就关闭这个对象，而不是重新创建新的，代码如下：

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

{

if ([self.imagePickerPopover isPopoverVisible]) {

// 如果imagePickerPopover指向的是有效的UIPopoverController对象，
```

```
// 并且该对象的视图是可见的，就关闭这个对象，并将其设置为nil

[imagePickerPopover dismissPopoverAnimated:YES];

imagePickerPopover = nil;

return;

}

UIImagePickerController *imagePicker =

[[UIImagePickerController alloc] init];
```

构建并运行应用。按下拍照按钮并显示UIPopoverController对象，然后再次按下拍照按钮，之前显示的UIPopoverController对象就会消失。

17.4 更多的模态视图控制器

本节将指导读者更新Homepwner，使其能够在添加BNRItem对象时以模态的形式显示一个BNRDetailViewController对象，供用户创建新的BNRItem对象（见图17-3）。当用户选中某个已存在的BNRItem对象时，Homepwner的处理不变，还是会将该对象压入UINavigationController对象的栈。


Edit
Homepwner
Cancel
Item
Done
Name
Item
Serial
Value
0
Dec 20, 2013

图17-3 添加BNRItem对象

为了实现BNRDetailViewController的两种使用模式，需要创建新的指定初始化方法initForNewItem:。initForNewItem:需要根据传入的使用模式（创建新的BNRItem对象，或者显示已存的BNRItem对象）设置相应的界面。

在BNRDetailViewController.h中声明initForNewItem:，代码如下：

```
- (instancetype)initForNewItem:(BOOL)isNew;

@property (nonatomic, strong) BNRItem *item;
```

当Homepwner使用BNRDetailViewController对象创建新的BNRItem对象时，需要在相应的UINavigationBar对象两端分别显示Done（完成）按钮和Cancel（取消）按钮。在BNRDetailViewController.m中实现initForNewItem:，代码如下：

```
- (instancetype)initForNewItem:(BOOL)isNew
{
self = [super initWithNibName:nil bundle:nil];
if (self) {
if (isNew) {
UIBarButtonItem *doneItem = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemDone
target:self
action:@selector(save:)];
self.navigationItem.rightBarButtonItem = doneItem;
UIBarButtonItem *cancelItem = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemCancel
target:self
action:@selector(cancel:)];
self.navigationItem.leftBarButtonItem = cancelItem;
}
```

```
}

return self;

}
```

在之前的代码中，当某个类要弃用父类的指定初始化方法并改用新的指定初始化方法时，都会覆盖父类的指定初始化方法并调用新的那个。本节要修改的BNRDetail- ViewController类不同，为了禁止使用父类的指定初始化方法，覆盖后的initWithNibName:bundle:方法不仅不应执行任何代码，而且要抛出异常，提示“禁止使用该初始化方法”。

在BNRDetailViewController.m中覆盖父类的指定初始化方法，代码如下：

```
- (instancetype)initWithNibName:(NSString *)nibNameOrNil

bundle:(NSBundle *)nibBundleOrNil

{

@throw [NSException exceptionWithName:@“Wrong initializer”

reason:@“Use initForNewItem:”

userInfo:nil];

return nil;

}
```

覆盖后的initWithNibName:bundle:会创建并抛出一个NSException对象。新创建的NSException对象包含名称（name属性）和原因（reason属性）。抛出异常后，应用会终止运行，并向控制台输出相应的错误信息。

为了测试initWithNibName:bundle:是否会抛出异常，需要先找到调用该方法的代码，即BNRItemsViewController的tableView:didSelectRowAtIndexPath:。在tableView:didSelectRowAtIndexPath:中，BNRItemsViewController对象会创建一个BNRDetailViewController对象，然后向新创建的对象发送init消息，init方法又会调用initWithNibName:bundle:。因此，当用户选中UITableView对象中的某一行时，就会导致应用抛出名称为Wrong initializer（调用错误的初始化方法）的异常。

构建并运行应用，（Xcode会警告找不到save:和cancel:方法，可以忽略这些警告。）选中UITableView对象中的某一行，Homepwner会终止运行并向控制台输出相应的异常信息，其中包括异常的名称和原因。

为了修正这个错误，也为了能够正确地初始化BNRDetailViewController对象，下面要修改BNRItemsViewController.m中的tableView:didSelectRowAtIndexPath:，使用新的指定初始化方法，代码如下：

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
BNRDetailViewController *detailViewController =
[[BNRDetailViewController alloc] init];
BNRDetailViewController *detailViewController =
[[BNRDetailViewController alloc] initForNewItem:NO];
NSArray *items = [[BNRItemStore sharedStore] allItems];
```

再次构建并运行应用，选中UITableView对象中的某一行，Homepwner应该不会再崩溃。

下面要添加新的代码，使Homepwner能够在用户添加新的BNRItem对象时显示BNRDetailViewController对象。

修改BNRItemsViewController.m中的addNewItem:方法，先创建一个新的BNRDetailViewController对象，然后创建一个新的UINavigationController对象，并将之前创建的BNRDetailViewController对象设置为该对象的根视图控制器，最后用模态形式显示该对象，代码如下：

```
- (IBAction)addNewItem:(id)sender
{
BNRItem *newItem = [[BNRItemStore sharedStore] createItem];
NSInteger lastRow =
[[[BNRItemStore sharedStore] allItems] indexOfObject:newItem];
NSIndexPath *indexPath = [NSIndexPath indexPathForRow:lastRow inSection:0];
[self.tableView insertRowsAtIndexPaths:@[indexPath]
withRowAnimation:UITableViewRowAnimationTop];
BNRDetailViewController *detailViewController =
[[BNRDetailViewController alloc] initForNewItem:YES];
detailViewController.item = newItem;
```

UINavigationController *navController = [[UINavigationController alloc]

initWithRootViewController:detailViewController];

[self presentViewController:navController animated:YES completion:nil];

}

构建并运行应用，按下+按钮，BNRDetailViewController对象的视图应该会从窗口底部滑入，相应的UINavigationBar对象会显示Done按钮和Cancel按钮（因为没有实现相应的动作方法，所以按下这两个按钮都会导致应用抛出异常）。

注意，即使BNRDetailViewController对象不会再导航到其他视图控制器，也需要将其先压入导航控制器栈中再推入，这样就不用为了显示标题以及放置Done按钮和Cancel按钮而单独添加一个导航栏（UINavigationBar对象）。

关闭模态视图控制器

要关闭某个以模态形式显示的视图控制器，必须向负责显示该对象的视图控制器发送dismissViewControllerAnimated:completion:。本章之前就是这样处理UIImagePicker- Controller对象的：BNRDetailViewController对象会负责以模态形式显示一个UIImagePickerController对象，当需要关闭UIImagePickerController对象时，也会由BNRDetailViewController对象负责关闭该对象。

添加BNRItem对象时的情况有些特别。添加该对象时，Homepwner会以模态的形式显示BNRDetailViewController对象。此外，相应的UINavigationBar对象会显示Done和Cancel两个按钮，当用户按下这两个按钮时，Homepwner会关闭BNRDetailViewController对象。这里的问题是，这两个按钮会将动作消息发送给BNRDetailViewController对象，但是负责关闭这个对象的是BNRItemsViewController对象。因此，BNRItemsView- Controller对象要通过某种途径得到以模态形式显示自己的那个视图控制器的指针，然后向该对象发送dismissViewControllerAnimated:completion:消息，从而关闭自己。

UIViewController对象有一个名为presentingViewController的属性，当某个UIViewController对象以模态形式显示时，该属性会指向显示该对象的那个UIViewController对象。因此，BNRDetailViewController对象可以通过presentingView- Controller属性得到所需的指针，然后向该指针指向的视图控制器发送dismissView- ControllerAnimated:completion:消息。在BNRDetailViewController.m中实现Done按钮的动作方法，代码如下：

- (void)save:(id)sender

{

[self.presentingViewController dismissViewControllerAnimated:YES

```
completion:nil];

}
```

Cancel按钮的动作方法会复杂一些。在BNRItemsViewController对象的视图中，当用户按下+按钮后，Homepwner会创建一个新的BNRItem对象，然后将该对象加入BNRItemStore对象。接着，Homepwner才会创建并显示BNRDetailViewController对象，供用户设置新创建的BNRItem对象。所以，当用户按下Cancel按钮时，Homepwner要从BNRItemStore对象移除之前创建的BNRItem对象。先在BNRDetailViewController.m顶部导入BNRItemStore.h，代码如下：

```
#import “BNRDetailViewController.h”

#import “BNRItem.h”

#import “BNRImageStore.h”

#import “BNRItemStore.h”

@implementation BNRDetailViewController
```

然后在BNRDetailViewController.m中实现Cancel按钮的动作方法，代码如下：

```
- (void)cancel:(id)sender

{

// 如果用户按下了Cancel按钮，就从BNRItemStore对象移除新创建的BNRItem对象

[[BNRItemStore sharedStore] removeItem:self.item];

[self.presentingViewController dismissViewControllerAnimated:YES

completion:nil];

}
```

构建并运行应用，添加新的BNRItem对象并按下Cancel按钮。BNRDetailViewController对象的视图应该会滑出窗口，且UITableView对象也不会加入新行。再添加一个BNRItem对象并按下Done按钮。BNRDetailViewController视图一样会滑出窗口，但是UITableView对象这次会加入新行。

最后还有一点需要向读者说明。前文提到过，负责以模态形式显示BNRDetailViewController对象的是BNRItemsViewController对象。其实这是为了方便读者理解而给出的简化的不准确的解释，实际的关系更复杂。BNRDetailViewController对象的presentingViewController属性指向的并不是BNRItemsViewController对象，而是包含该对象的UINavigation- Controller对象。这是由Cocoa Touch的内部实现机制决定的。这也是为什么当以模态形式显示BNRDetailViewController对象时，该对象的视图会遮住UINavigationBar对象。

如果读者要实现的只是关闭模态视图控制器，就不用关心presentingViewController属性指向的究竟是哪个对象。只需要向该属性指向的对象发送dismissViewControllerAnimated:completion:消息即可。本章会在结尾处详细解释与此有关的视图控制器关系。

视图控制器的模态样式

在iPhone或iPod touch中以模态形式显示视图控制器时，视图控制器的视图会占据整个窗口。对iPhone系设备，这是默认的样式，也是唯一的选择。对iPad则有两个额外的选项：表单样式（form sheet）和页单样式（page sheet）。修改视图控制器的modalPresentationStyle属性，可以改变其在模态形式下的外观。该属性的类型是UIModalPresentationStyle，其值可以是UIModalPresentationFormSheet或UIModalPresentationPageSheet两个常量中的一个。

在表单样式下，模态视图控制器的视图会出现在iPad屏幕中部的矩形区域，并且其下层的视图（即parentViewController的视图）会变暗（见图17-4）。


Edit
Homepwner
Cancel
Item
Done
Name
Item
Serial
Value
0
Dec 20, 2013

图17-4 表单样式示例

在页单样式下，当设备处于竖排方向时，视图的显示效果和其在全屏样式下的相同。当设备处于横排方向时，视图的宽度和其在竖排方向时的相同。此外，在视图的左右两侧，位于其下层的视图（即parentViewController的视图）会变暗。

更新BNRItemsViewController.m中的addNewItem:，修改将要以模态形式显示的UINavigationController对象的外观样式，代码如下：

```
UINavigationController *navController = [[UINavigationController alloc]

initWithRootViewController:detailViewController];

navController.modalPresentationStyle = UIModalPresentationFormSheet;

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

值得注意的是，这段代码修改的是UINavigationController对象的modalPresentation- Style，而不是BNRDetailViewController对象的。这是因为以模态形式显示的视图控制器是UINavigationController对象。

针对iPad模拟器或iPad构建并运行应用。按下+按钮，添加新的BNRItem对象，相应的视图控制器应该会滑入窗口。填入若干BNRItem对象的详细信息并按下Done按钮。UITableView对象会再次出现，但是并没有显示新的BNRItem对象。为什么？

修改模态样式前，UINavigationController对象会占据整个屏幕，并彻底遮住BNRItemsViewController对象的视图，即导致该视图“消失”。关闭模态视图控制器后，BNRItemsViewController对象的视图会再次出现，因此该对象会收到viewWillAppear:消息和viewDidAppear:消息，从而有机会重新刷新UITableView对象，使其显示的内容和BNRItemStore对象所保存的数据保持一致。

修改模态样式后，UINavigationController对象不会占据整个屏幕，BNRItemsViewController对象的视图也不会消失。关闭模态视图控制器后，BNRItemsViewController对象不会收到viewWillAppear:消息和viewDidAppear:消息，因此也没有机会刷新UITableView对象。

为了解决上述问题，必须另找机会来刷新UITableView对象。先准备要加入的代码，在BNRItemsViewController对象的某个方法中刷新UITableView对象，只需编写一行简单的代码，如下：

```
[self.tableview reloadData];
```

为了能够在模态视图控制器被关闭后立刻执行这行代码，可以使用视图控制器的dismissViewControllerAnimated:completion:方法。

类型为Block对象的completion实参

前文中的代码在调用dismissViewControllerAnimated:completion:方法和presentViewController:animated:completion:方法时，传入的最后一个实参都是nil。以dismissViewControllerAnimated:completion:为例，它的最后一个实参声明如下。

```
- (void)dismissViewControllerAnimated:(BOOL)flag

completion:(void (^)(void))completion;
```

completion实参的类型是“Block对象”。通过dismissViewControllerAnimated: completion:的这个实参，可以解决之前提到的问题。在使用该实参前，要先向读者介绍Block对象。Block对象的概念和语法并不容易掌握，所以本节只进行简单介绍。后续章节在涉及Block对象时，会穿插讲解Block对象在各类场景下的具体用法。

Block对象封装了一段用于延时执行的代码，因此，可以将刷新UITableView对象的那行代码放入一个Block对象，然后将指向该对象的指针作为实参传入dismissViewController-Animated:completion:。当相关的视图控制器被关闭后，Homepwner就会执行Block对象中的这段代码。

在BNRDetailViewController.h中为BNRDetailViewController添加一个属性，用于保存指向Block对象的指针。

```
@property (nonatomic, copy) void (^dismissBlock)(void);
```

这行代码的作用是声明BNRDetailViewController拥有一个名为dismissBlock的属性，该属性是一个指向Block对象的变量。类似C函数，Block对象也有返回值和一组实参，必须在声明Block对象时一并列出。以dismissBlock属性所指向的Block对象为例，其返回值是void，并且没有任何实参。

为了访问需要刷新的UITableView对象，必须在BNRItemsViewController的实例方法中创建相应的Block对象。这是因为只有BNRItemsViewController对象才能够通过tableView属性访问其UITableView对象。

修改BNRItemsViewController.m中的addNewItem:，创建一个负责刷新UITableView对象的Block对象，并将该对象赋给BNRDetailViewController对象，代码如下：

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

{

// 创建BNRItem对象，然后将新创建的对象加入BNRItemStore对象

BNRItem *newItem = [[BNRItemStore sharedStore] createItem];

BNRDetailViewController *detailViewController =

[[BNRDetailViewController alloc] initForNewItem:YES];
```

```
detailViewController.item = newItem;

detailViewController.dismissBlock = ^{

[self.tableView reloadData];

};

UINavigationController *navController = [[UINavigationController alloc]

initWithRootViewController:detailViewController];
```

当用户按下+按钮添加一个BNRItem对象时，BNRItemsViewController对象会创建一个Block对象并将其指针赋给BNRDetailViewController对象的dismissBlock属性。该Block对象所包含的代码会刷新BNRItemsViewController对象的UITableView对象。

当要关闭以模态形式显示的BNRDetailViewController对象时，可以将dismissBlock属性所指向的Block对象传给dismissViewControllerAnimated: completion:。修改BNRDetailViewController.m中的save:和cancel:，在调用dismissViewControllerAnimated:completion:时将dimissBlock属性作为最后一个实参传入，代码如下：

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

{

[self.presentingViewController dismissViewControllerAnimated:YES

completion:nil];

[self.presentingViewController dismissViewControllerAnimated:YES

completion:self.dismissBlock];

}

- (IBAction)cancel:(id)sender

{

[[BNRItemStore sharedStore] removeItem:self.item];

[self.presentingViewController dismissViewControllerAnimated:YES

completion:nil];

[self.presentingViewController dismissViewControllerAnimated:YES
```

completion:self.dismissBlock];

}

构建并运行应用。按下+按钮，添加一个BNRItem对象，然后按下Done按钮。UITableView对象应该会显示新创建的BNRItem对象。

读者目前不用太在意Block对象的语法和使用方法，从第19章起会详细介绍Block对象。

以模态形式显示视图控制器时的动画效果

前文介绍了如何修改视图控制器的模态样式。当应用以模态形式显示某个视图控制器时，会附带特定的动画效果，这个动画效果也是可以修改的。类似模态样式，视图控制器有一个名为modalTransitionStyle的属性，该属性的类型是UIModalTransitionStyle，其值必须是相关的预定义常量之一。如果使用默认的动画效果，则视图控制器的视图会从屏幕底部滑入。其他可以使用的动画效果有淡入（fade in）、翻转（flip in）和模拟书页卷角（page curl）。

以下是这些动画效果所对应的常量：

UIModalTransitionStyleCoverVertical从底部滑入

UIModalTransitionStyleCrossDissolve淡入

UIModalTransitionStyleFlipHorizontal以3D效果翻转

UIModalTransitionStylePartialCurl模拟书页卷角

17.5 线程安全的单例

到目前为止，本书开发的都是单线程应用（single-threaded app）。单线程应用无法充分利用多核设备（从iPhone 4S起，世界上大部分iOS设备都属于多核设备）的CPU资源：在同一时间，单线程应用只能使用CPU的一个核，也只能执行一个函数。相反，多线程应用（multi-threaded app）可以同时在不同的CPU核上执行多个函数。

本书第11章是通过以下代码创建单例的：

```
+（instancetype）sharedStore
{
static BNRImageStore *sharedStore = nil;
if（！sharedStore）{
sharedStore = [[self alloc] initPrivate];
}
return sharedStore;
}
// 不允许直接调用init方法
-（instancetype）init
{
@throw [NSException exceptionWithName:@“Singleton”
reason:@“Use +[BNRImageStore sharedStore]”
userInfo:nil];
return nil;
}
// 私有初始化方法
-（instancetype）initPrivate
{
```

```
self = [super init];

if (self) {

_dictionary = [[NSMutableDictionary alloc] init];

}

return self;

}
```

以上代码可以在单线程应用中正确创建单例，但是在多线程应用中，以上代码可能会创建多个BNRImageStore对象。同时，某个线程还可能会访问其他线程中没有正确初始化的BNRImageStore对象。

为了确保在多线程应用中只创建一次对象，可以使用dispatch_once()创建线程安全的单例(thread-safe singleton)。

打开BNRImageStore.m，修改sharedStore方法，为BNRImageStore创建线程安全的单例：

```
+ (instancetype)sharedStore

{

static BNRImageStore *sharedStore = nil;

~~if (! sharedStore) {~~

~~sharedStore = [[self alloc] initPrivate];~~

~~}~~

static dispatch_once_t onceToken;

dispatch_once(&onceToken, ^{

sharedStore = [[self alloc] initPrivate];

});

return sharedStore;

}
```

构建并运行应用，虽然运行结果不会有任何变化，但是现在sharedStore可以在多核设备中正确返回唯一的BNRImageStore对象。

17.6 初级练习：为另一个类添加线程安全的单例

模仿17.5节，使用dispatch_once()为BNRItemStore添加线程安全的单例。

17.7 高级练习：UIPopoverController对象的外观

UIPopoverController对象的外观是可以修改的。更新代码，修改负责显示UIImagePickerController对象的UIPopoverController对象的外观（提示：可以使用UIPopoverController的popoverBackgroundViewClass属性）。

17.8 深入学习：位掩码

本章之前介绍了supportedInterfaceOrientations方法，其返回值决定了视图控制器可以支持的所有界面方向。虽然supportedInterfaceOrientations方法仅仅返回单一整形值，但是这个整形值仍然可以囊括四种界面方向中的任意一种或几种。如何使用单一值描述多种可能值？答案是位掩码（bitmasks）。

使用位掩码可以避免为保存每一种可能值而创建大量的属性。例如，在不使用位掩码的情况下，如果要表示视图控制器所支持的界面方向，可能会添加下列四个属性：

```
@property (nonatomic, assign) BOOL canRotateToLandscapeLeft;

@property (nonatomic, assign) BOOL canRotateToLandscapeRight;

@property (nonatomic, assign) BOOL canRotateToPortrait;

@property (nonatomic, assign) BOOL canRotateToPortraitUpsideDown;
```

界面方向只有四种，需要添加四个属性，而iOS中还有大量的类具有存在更多可能值的选项。例如UIView，如果打开其头文件，可以看见其动画选项UIViewAnimationOptions具有数十种可能值（本书第27章会介绍动画）。相反，使用位掩码，只需要单一整形值就可以存储任意数量的可能值。

位掩码会将各个值表示为一系列开关，其原理是：计算机是使用二进制格式存储数据的，二进制数据是由一系列0和1组成的。下面列出的是若干以10为基数的数（十进制是我们使用的数字进制）和相应的以2为基数的数（二进制是计算机使用的数字进制）。

110 = 000000012

210 = 000000102

1610 = 000100002

2710 = 000110112

3410 = 001000102

二进制数字中的列称为二进制位（bit）。可以将二进制位想象成开关，1代表“开”，0代表“关”。这样就可以将int类型的整数（至少会占用32个二进制位）想成是一组开关。数字的每个位代表一个“开关”：1代表“开（真）”，0代表“关（假）”。这等于是将很多布尔值塞进了一个整数。

在以上数字中，1、2和16都是2的n次幂（分别是0次幂、1次幂和4次幂）——只有一个二进制位是1（其余都是0）。而27和34不是2的n次幂，其二进制数值中有多个1。可以用这类数值为2次幂的数字来代表位掩码中的“开关”，每个开关称为一个掩码。

例如，以下是所有界面方向的位掩码：

UIInterfaceOrientationMaskPortrait = 210 = 000000102

UIInterfaceOrientationMaskPortraitUpsideDown = 410 = 000001002

UIInterfaceOrientationMaskLandscapeRight = 810 = 000010002

UIInterfaceOrientationMaskLandscapeLeft = 1610 = 000100002

用按位或(|)运算符可以“打开”位掩码中的某个“开关”。执行按位或运算时需要两个数字，只要这两个数字位于同一列上的值有一个是1，那么生成的数字的相应列的值也会是1。将某个数字和一个值为2n的数字进行按位或运算时，第n位的“开关”就会打开。例如，按位或1和16，会得到如下结果：

00000010（210，UIInterfaceOrientationMaskPortrait）

|00010000（1610，UIInterfaceOrientationMaskLandscapeLeft）

00010010（1810，同时包含UIInterfaceOrientationMaskPortrait

和UIInterfaceOrientationMaskLandscapeLeft）

和按位或运算符相对的是按位与(&)运算符。执行按位与运算时需要两个数字，只有当这两个数字位于同一列上的值都是1时，生成的数字的相应列的值才会是1。

00010010（1810，Portrait和LandscapeLeft）

&00010000（1610，LandscapeLeft）

00010000（1610，YES）

00010010（1810，Portrait和LandscapeLeft）

&00000100（410，UpsideDown）

00000000（010，NO）

因为非0代表真(0代表假)，所以可以用按位与运算符来检查“开关”是打开的还是关闭的。可以使用以下代码检查视图控制器所支持的界面方向：

if ([viewController supportedInterfaceOrientations]

```
& UIInterfaceOrientationMaskLandscapeLeft）

{

// 支持左横排方向

}
```

17.9 深入学习：视图控制器之间的关系

要理解应用会在什么位置，以及会以怎样的形式显示某个视图控制器，就有必要搞清楚视图控制器之间的关系。视图控制器之间的关系可以分为两类：父-子关系和显示-被显示关系。下面分别针对这两种类型进行详细介绍。

父-子关系

当使用视图控制器容器（view controller container）时，就会产生拥有父-子关系的视图控制器。UINavigationController对象、UITabBarController对象和UISplitViewController对象（第22章会介绍）都是视图控制器容器。这些容器的共性是都有一个类型为数组对象的viewControllers属性，用于保存一组视图控制器。

无论哪种视图控制器容器，都是UIViewController的子类对象，因此也有一个名为view的属性。视图控制器容器的特性是：容器对象会将viewControllers中的视图作为子视图加入自己的视图。此外，容器对象通常都会有自己的默认外观。以UINavigation- Controller对象为例，它会在其视图顶部显示一个UINavigationBar对象，然后在余下的空间中显示topViewController的视图。

处在同一个父-子关系下的视图控制器形成一个族系（family）。以UINavigationController为例，某个UINavigationController对象和其下的viewControllers都属于同一个族系。一个族系可以有多个级别，例如，一个UITabBarController对象可以包含一个UINavigationController对象，而这个UINavigationController对象又可以包含一个UIViewController对象。在这种情况下，这三个视图控制器都处在同一个族系中（见图17-5）。任何容器对象都可以通过viewControllers访问其子对象，而子对象也可以通过UIViewController对象的四个特定属性来访问其容器对象。

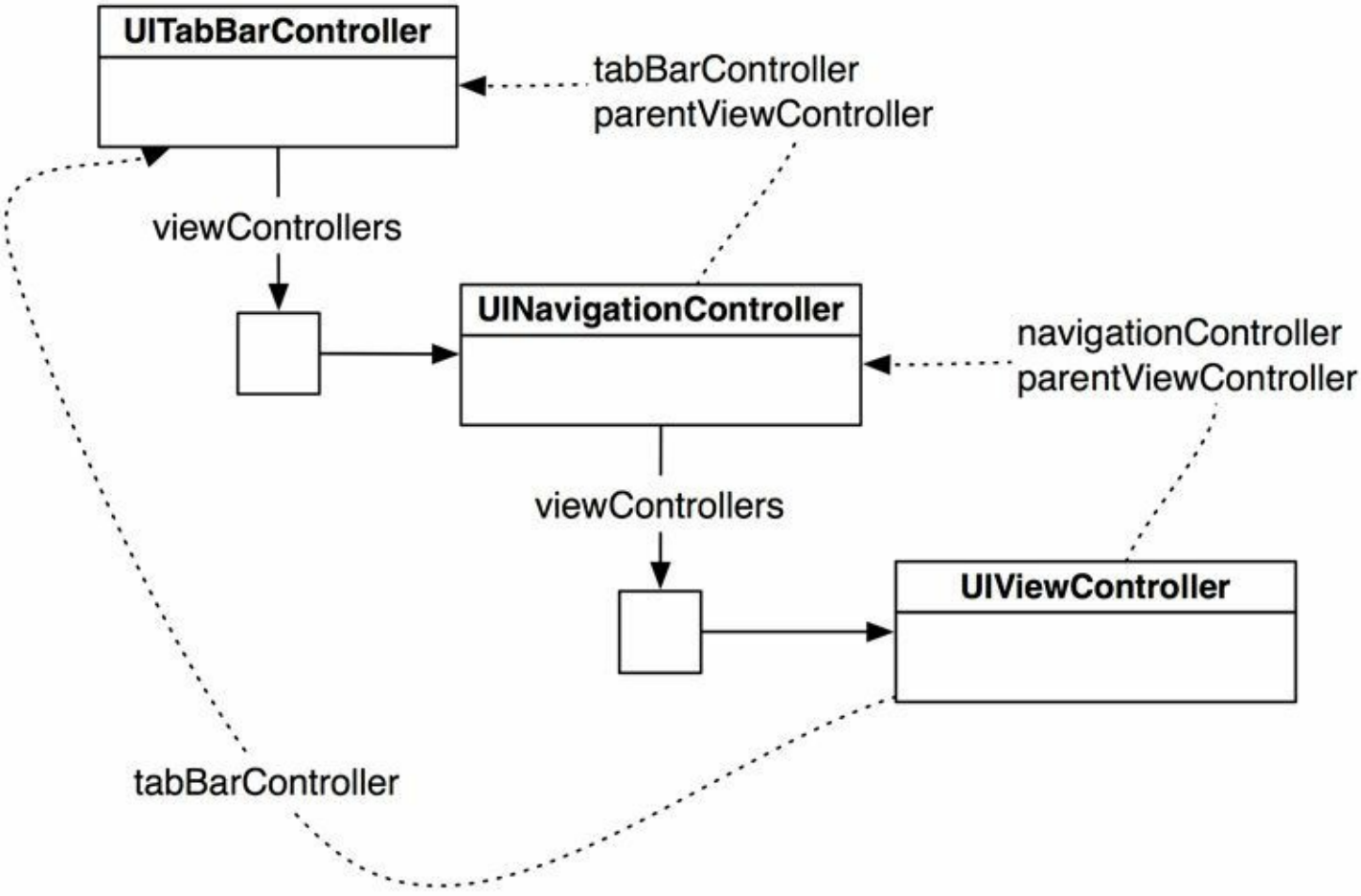


图17-5 视图控制器族系示例

先介绍这四个特定属性的前三个，这三个属性分别是：navigationController、tabBarController和splitViewController。当某个视图控制器收到navigationController消息、tabBarController消息或splitViewController消息后，就会沿着族系向上查找，直到找到类型匹配的视图控制器容器。如果没有找到，相应的方法就会返回nil。

然后介绍第四个属性。UIViewController有一个名为parentViewController的属性。该属性会指向族系中“最近”的那个容器对象。因此，根据族系的形成方式，parentViewController可能会指向UINavigationController对象、UITabBarController对象或UISplitViewController对象。

显示-被显示关系

视图控制器的另一类关系是显示-被显示关系（presenting-presenter relationship）。当某个视图控制器以模态形式显示另一个视图控制器时，就会产生拥有这种关系的视图控制器。当某个视图控制器（A）以模态形式显示另一个视图控制器（B）时，B的视图会覆盖A的视图。这和之前介绍的父-子关系不同，父-子关系中的子视图只会在容器对象的视图内显示。此外，任何一个UIViewController对象都可以以模态的形式显示另一个视图控制器。

通过UIViewController对象的presentingViewController属性和presentedViewController属性，可以得到指向相应视图控制器的指针。当某个视图控制器（A）以模态形式显示另一个视图控

制器(B)时，A的presentedViewController会指向B，而B的presentingViewController属性会指向A(见图17-6)。

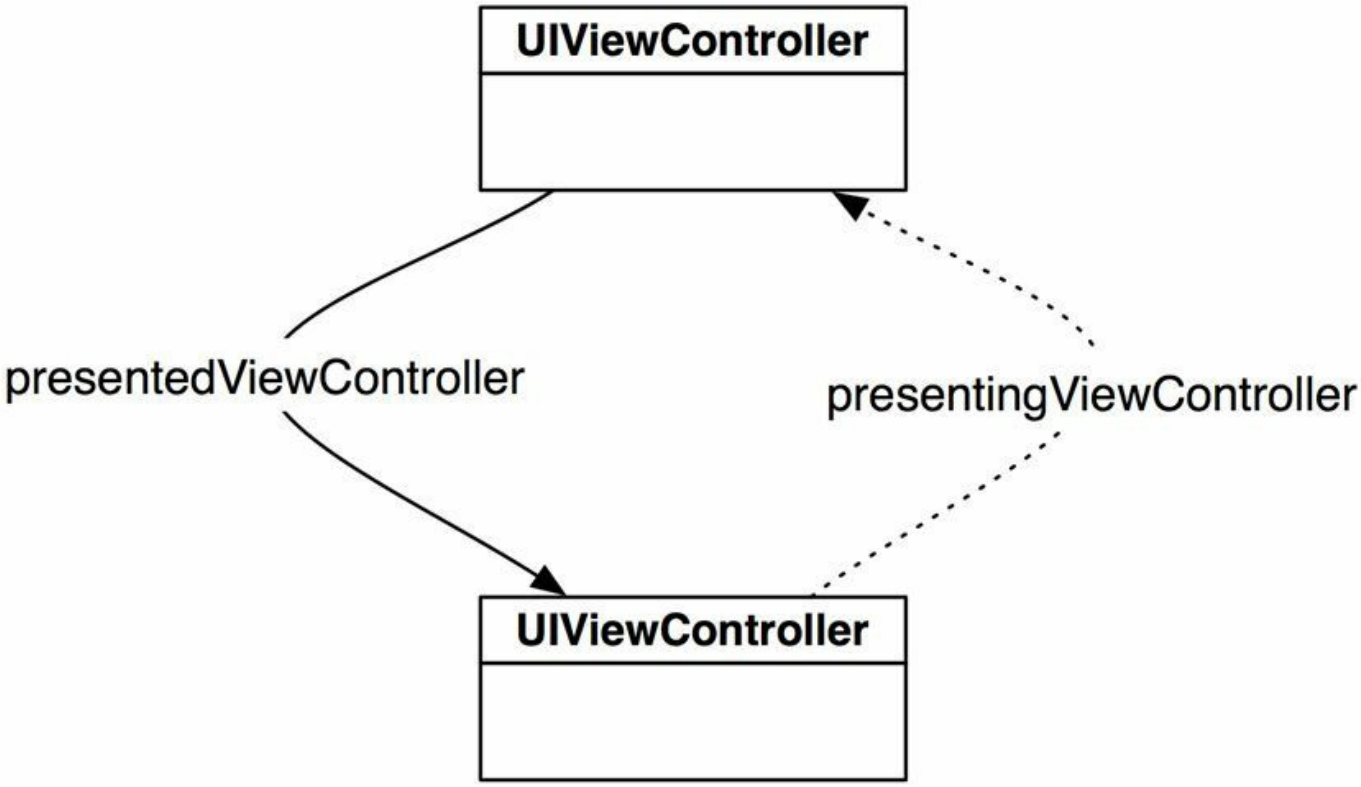


图17-6 显示-被显示关系

显示-被显示关系中的族系

在显示-被显示关系中，位于关系两头的视图控制器不会处于同一个族系中。被显示的视图控制器会有自己的族系，这个族系可以只有一个UIViewController对象，也可以由多个UIViewController对象组成。

UIViewController中的某些属性是和族系有关的(例如presentedViewController和navigationController)，理解各种族系之间的差别可以帮助读者理解这些属性的工作原理。以图17-7中的视图控制器为例，图中有两个族系，分别包含多个视图控制器。此外，该图还显示了和视图控制器有关的各个属性。

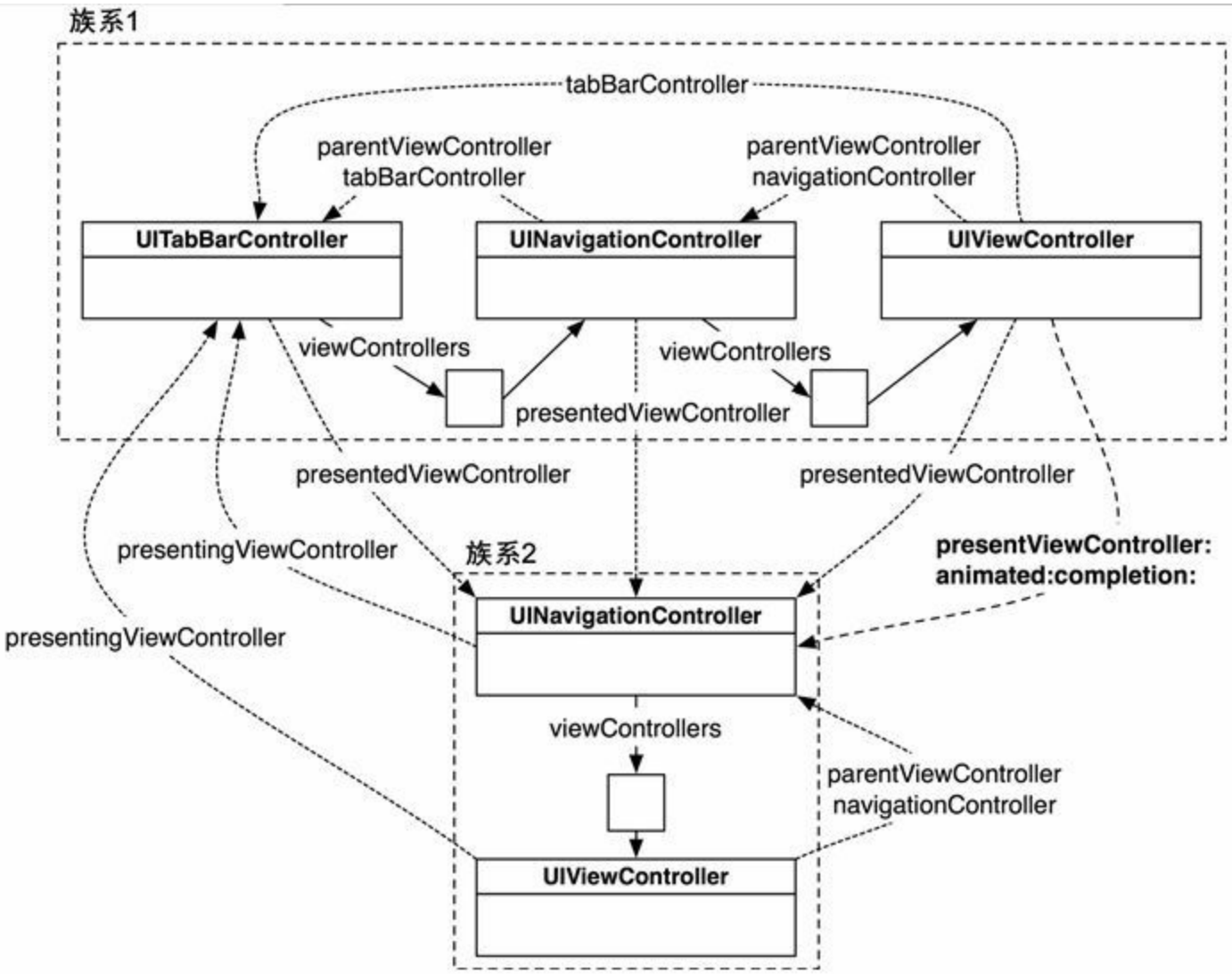


图17-7 视图控制器的层次结构示例

从图17-7中可以看出，凡是针对父-子关系的属性，其指向的对象都会在当前族系的范围内。因此，向族系2中的视图控制器发送tabBarController消息，并不会返回族系1中的UITabBarController对象，而会返回nil。同理，向族系2中的视图控制器发送navigationController消息，返回的是族系2中的UINavigationController对象，而不是族系1中的。

不同族系中的视图控制器的关系稍微复杂一些。当应用以模态形式显示某个视图控制器时，负责显示该视图控制器的将是相关族系中的顶部视图控制器（或者可以说“最老”的那个）。以图17-7中的对象为例，族系2中的视图控制器的presentingViewController属性指向的都是UITabBarController对象。无论应用是向族系1中的哪个视图控制器发送presentViewController:animated:completion:消息，负责显示的对象始终是UITabBarController对象。

这也解释了为什么当Homepwner以模态形式显示BNRDetailViewController对象时，该对象的视图会遮住UINavigationBar对象，而不会将BNRDetailViewController对象压入UINavigationController对象的栈。虽然Homepwner向BNRItemsViewController对象发送的是presentViewController:animated:completion:消息，但是实际负责以模态形式显示BNRDetailViewController对象的是位于族系顶部的UINavigationController对象。BNRDetailViewController对象的视图会被放置在UINavigationController对象的视图上方，从而

导致遮住UINavigationBar对象。

对某个族系中的所有视图控制器，其presentingViewController属性和presentedViewController属性都是有效的，并且总会指向另一个族系中的顶部对象。

通过编写代码，可以改变这种“顶部对象负责以模态形式显示其他视图控制器”的行为（只能在iPad中使用），这样就可以限定视图的显示位置。以BNRDetailViewController对象为例，可以要求包含了该对象的UINavigationController对象显示在UITableView对象上，从而避免覆盖UINavigationBar。

为此，UIViewController提供了definesPresentationContext属性，其默认值是NO。当某个视图控制器的definesPresentationContext是NO时，会将“显示权（presentation off）”传递给父视图控制器，并沿着族系依次向上传递，直到最顶层视图控制器。也就是说，最后会由最顶层视图控制器负责显示新的视图控制器；相反，在传递过程中，如果某个视图控制器的definesPresentationContext是YES，该视图控制器就不会再将“显示权”传递给父视图控制器，而是由自己负责显示新的视图控制器（见图17-8）。此外，对后面这种情况，必须将需要显示的视图控制器的modalPresentationStyle属性设置为UIModalPresentationCurrentContext。

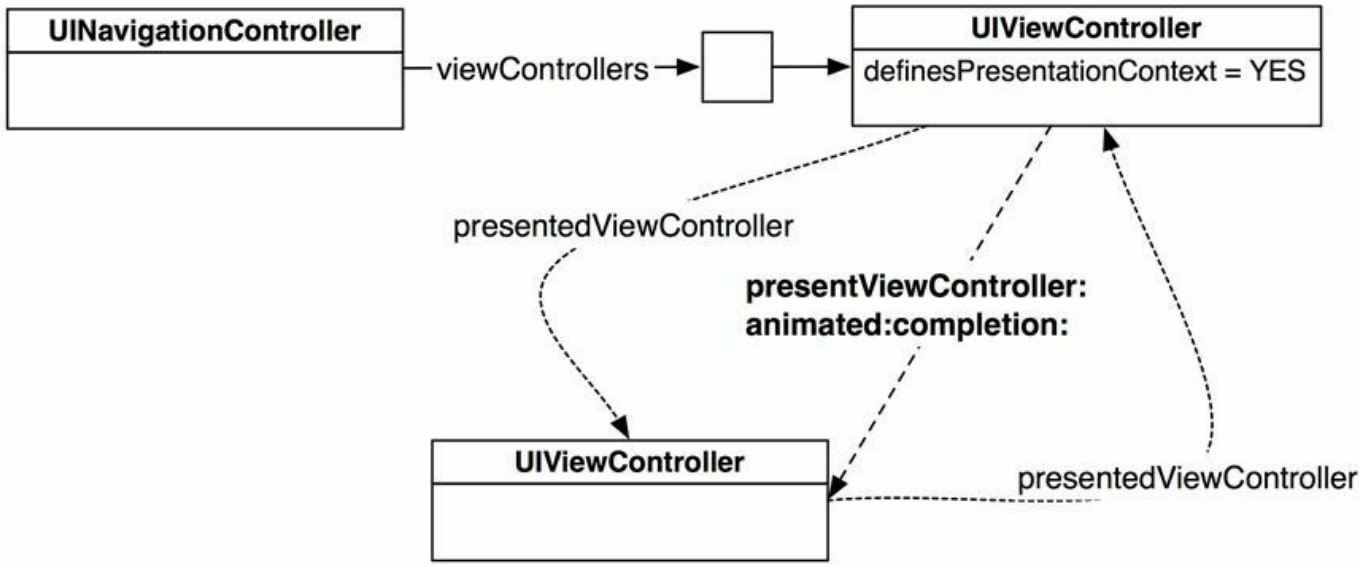


图17-8 definesPresentationContext属性为YES的示例

修改BNRItemsViewController.m中的addNewItem:，针对上述内容做一个测试，代码如下：

```
UINavigationController *navController = [[UINavigationController alloc]

initWithRootViewController:detailViewController];

navController.modalPresentationStyle = UIModalPresentationFormSheet;

navController.modalPresentationStyle = UIModalPresentationCurrentContext;

self.definesPresentationContext = YES;
```

```
    navController.modalTransitionStyle =

    UIModalTransitionStyleFlipHorizontal;

    [self presentViewController:navController animated:YES completion:nil];

    }
```

针对iPad模拟器或iPad构建并运行应用。按下+按钮，BNRDetailViewController对象的视图应该不会覆盖UINavigationBar对象。以上修改只是为了测试，请读者在开始第18章之前先还原代码。

第18章 保存、读取与应用状态

iOS SDK提供了多种保存和读取数据的机制，本章将介绍其中最常用的几种。此外，还会针对iOS的文件系统介绍和文件存取有关的基本概念。

18.1 固化

对大多数iOS应用，可以将其功能总结为：提供一套界面，帮助用户管理特定的数据。在这一过程中，不同类型的对象要各司其职：模型对象负责保存数据，视图对象负责显示数据，控制器对象负责在模型对象与视图对象之间同步数据。因此，当某个应用要保存和读取数据时，通常要完成的任务是保存和读取相应的模型对象。

对Homepwner，用户可以管理的模型对象是BNRItem对象。目前Homepwner不能保存BNRItem对象，所以，当用户重新运行Homepwner时，之前创建的BNRItem对象都会消失。本章将介绍如何通过固化来保存和读取BNRItem对象。

固化是由iOS SDK提供的一种保存和读取对象的机制，使用非常广泛。当应用固化某个对象时，会将该对象的所有属性存入指定的文件。当应用解固（unarchive）某个对象时，会从指定的文件读取相应的数据，然后根据数据还原对象。

为了能够固化或解固某个对象，相应对象的类必须遵守NSCoding协议，并且实现两个必需方法：encodeWithCoder:和initWithCoder:，代码如下：

```
@protocol NSCoding

- (void)encodeWithCoder:(NSCoder *)aCoder;

- (instancetype)initWithCoder:(NSCoder *)aDecoder;

@end
```

打开Homepwner.xcodeproj，在BNRItem.h中将BNRItem声明为遵守NSCoding协议，代码如下：

```
@interface BNRItem : NSObject <NSCoding>
```

下面为BNRItem实现NSCoding协议的两个必需方法。先实现encodeWithCoder:，它有一个类型为NSCoder的参数，BNRItem的encodeWithCoder:方法要将所有的属性都编码至该参数。在固化过程中，NSCoder会将BNRItem转换为键-值对形式的数据并写入指定的文件。

在BNRItem.m中实现encodeWithCoder:，将BNRItem中所有属性的名称和值加入NSCoder对象，代码如下：

```
- (void)encodeWithCoder:(NSCoder *)aCoder

{

[aCoder encodeObject:self.itemName forKey:@“itemName”];

[aCoder encodeObject:self.serialNumber forKey:@“serialNumber”];

[aCoder encodeObject:self.dateCreated forKey:@“dateCreated”];
```

```
[aCoder encodeObject:self.itemKey forKey:@“itemKey”];

[aCoder encodeInt:self.valueInDollars forKey:@“valueInDollars”];

}
```

在这段代码中，凡是指向对象的指针都会用encodeObject:forKey:编码，而valueInDollars是用encodeInt:forKey:编码的。请读者查阅NSCoder的文档了解编码的数据类型。无论编码哪种类型的数据，必须有相应的键才能存入NSCoder对象。这个键是字符串，负责标识相应的属性。按照约定，编码某个属性时要使用的键就是该属性的名称。

当应用需要编码某个对象时（encodeObject:forKey:中的第一个参数），会向该对象发送encodeWithCoder:消息。收到该消息的对象需要编码自己的属性，所以也会向这些属性发送encodeWithCoder:消息（见图18-1）。因此，对象的编码过程是一个递归过程：编码中的对象会再编码其他对象。

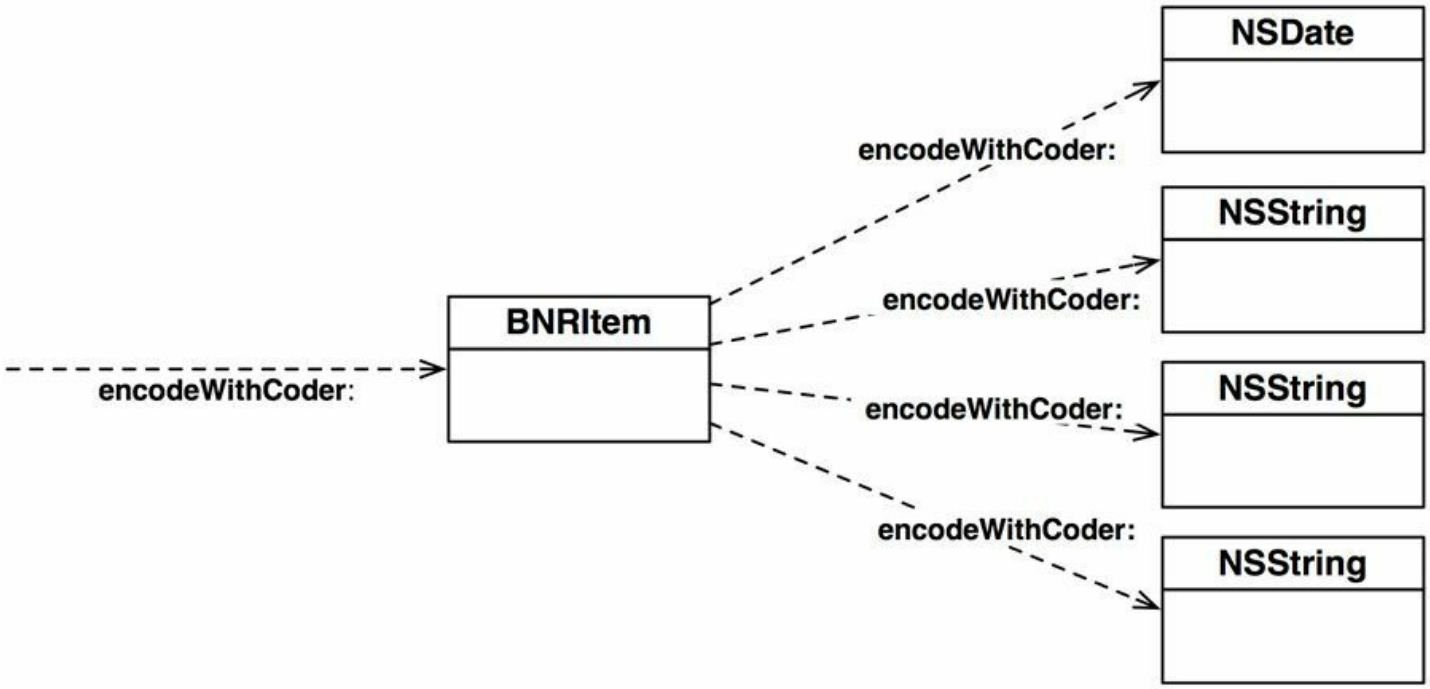


图18-1 编码BNRItem对象

为了能够编码BNRItem对象，BNRItem的所有属性也必需遵守NSCoding协议（除了valueInDollars）。现在请读者在开发文档中查看类参考手册，检查NSString和NSDate是否遵守NSCoding协议。除了打开文档浏览器外，还有一种更快捷的方式可以直接在代码中查看类参考手册。

在BNRItem.m中，按住Option键，点击代码中任意一个NSString，Xcode会弹出一个提示框，显示类的简要说明、头文件链接和参考手册链接（见图18-2）。

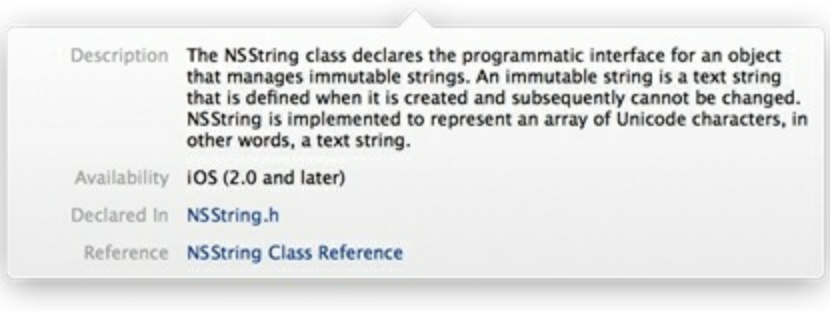


图18-2 按住Option键，点击NSString

点击NSString Class Reference，在参考手册顶部可以看到NSString遵守的所有协议，其中并没有NSCoding协议，但是包含一个NSSecureCoding协议，点击NSSecureCoding，可以发现该协议遵守NSCoding，因此NSString也遵守NSCoding。

使用同样的方法查看NSDate的类参考手册，可以发现NSDate也遵守NSCoding。

按住Option键，点击方法、类型定义、协议等都可以打开相应的开发文档，读者可以使用这种方式在编写代码的过程中快速查阅需要的信息，以了解它们的使用方法。

现在继续讨论编码。编码BNRItem对象时，需要针对每个属性指定相应的键。当Homepwner从文件读取相应的数据并重新创建BNRItem对象时，会根据键来设置属性。当应用需要根据编码后的数据初始化某个对象时，会向该对象发送initWithCoder:消息。initWithCoder:应该还原之前通过encodeWithCoder:编码的所有对象，然后将这些对象赋给相应的属性。在BNRItem.m中实现initWithCoder:，代码如下：

```
- (instancetype)initWithCoder:(NSCoder *)aDecoder

{

self = [super init];

if (self) {

_itemName = [aDecoder decodeObjectForKey:@“itemName”];

_serialNumber = [aDecoder decodeObjectForKey:@“serialNumber”];

_dateCreated = [aDecoder decodeObjectForKey:@“dateCreated”];

_itemKey = [aDecoder decodeObjectForKey:@“itemKey”];

_valueInDollars = [aDecoder decodeIntForKey:@“valueInDollars”];

}

return self;
```

```
}
```

initWithCoder:也有一个类型为NSCoder的参数，和encodeWithCoder:不同，该参数的作用是为初始化BNRItem对象提供数据。这段代码通过向NSCoder对象发送decodeObjectForKey:重新设置相应的属性。valueInDollars是一个例外，它是整数类型的属性，需要使用decodeIntForKey:创建。

第2章介绍过初始化链和指定初始化方法。initWithCoder:是一个特例，和第2章介绍的这些初始化方法无关。BNRItem需要保留原有的指定初始化方法，initWithCoder:也不会调用指定初始化方法。

XIB文件也是基于固化机制的。当读者在Xcode中将某个视图拖曳至画布时，Xcode会创建相应的对象。保存XIB文件时，Xcode会将这些视图固化至指定的文件（UIView遵守NSCoding协议）。当应用需要载入XIB文件时，就会解固XIB文件中的视图。和普通的固化文件相比，XIB文件会略有差别，但是两者保存和载入的流程大致相同。

修改后的BNRItem对象遵守NSCoding协议，可以通过固化机制来保存和读取它。构建应用，确保没有语法错误。下一个需要解决的问题是，应该将BNRItem对象保存在哪里？

18.2 应用沙盒

每个iOS应用都有自己专属的应用沙盒。应用沙盒就是文件系统中的目录，但是iOS系统会将每个应用的沙盒目录与文件系统的其他部分隔离（见图18-3）。应用必须“待”在自己的沙盒里，并只能访问自己的沙盒。

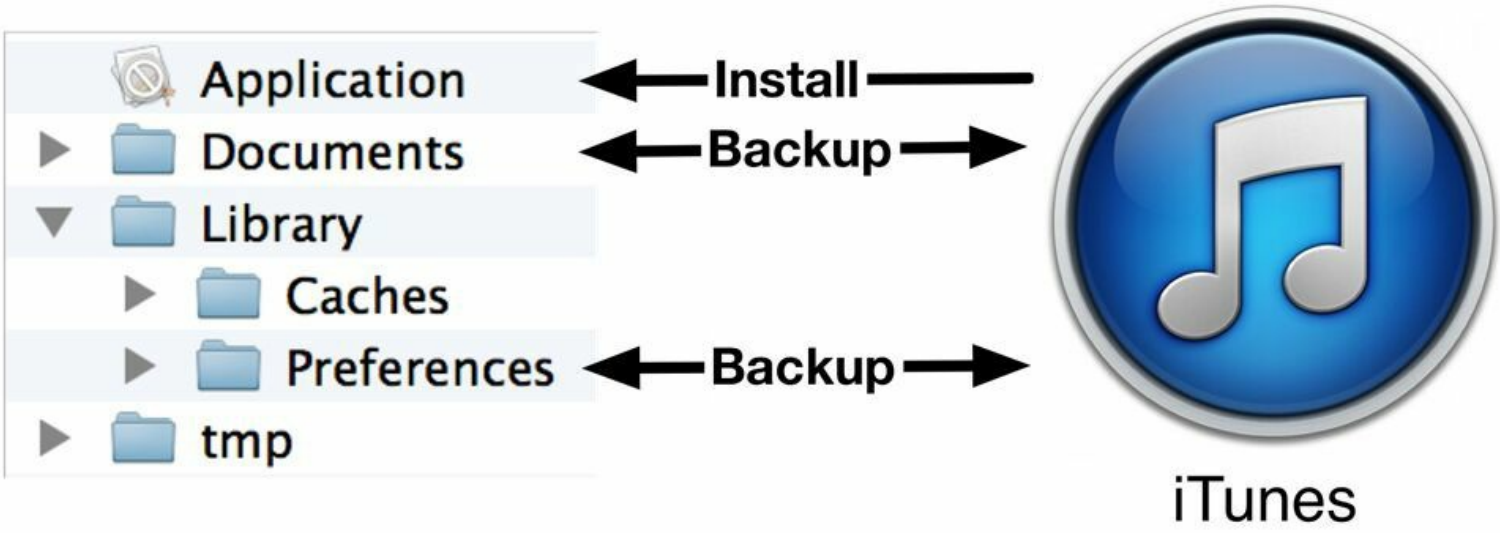


图18-3 应用沙盒

应用沙盒会包含以下多个目录：

应用程序包（application bundle）	包含应用可执行文件和所有资源文件，例如NIB文件和图像文件。它是只读目录
Documents/	存放应用运行时生成的并且需要保留的数据。iTunes或iCloud会在同步设备时备份该目录。当设备发生故障时，可以从iTunes或iCloud恢复该目录中的文件。例如，Homepwner应用可将用户所拥有的物品信息保存在Documents/中
Library/Caches/	存放应用运行时生成的需要保留的数据。与Documents/目录不同的是，iTunes或iCloud不会在同步设备时备份该目录。不备份缓存数据的主要原因是相关数据的体积可能会很大，从而延长同步设备所需的时间。如果数据源是在别处（例如Web服务器），就可以将得到的数据保存在Library/Caches/目录。当用户需要恢复设备时，相关的应用只需要从数据源（例如Web服务器）再次获取数据即可
Library/Preferences/	存放所有的偏好设置，iOS的设置（Settings）应用也会在该目录中查找应用的设置信息。使用NSUserDefaults类（第26章会介绍），可以通过Library/Preferences目录中的某个特定文件以键-值对的形式保存数据。iTunes或iCloud会在同步设备时备份该目录
tmp/	存放应用运行时所需的临时数据。当某个应用没有运行时，iOS系统可能会清除该应用的tmp/目录下的文件，但是为了节约用户设备空间，不能依赖这种自动清除机制，而是当应用不再需要使用tmp/目录中的文件时，就及时手动删除这些文件。iTunes或iCloud不会在同步设备时备份tmp/目录。通过NSTemporaryDirectory函数可以得到应用沙盒中的tmp/目录的全路径

应用程序包（application bundle）包含应用可执行文件和所有资源文件，例如NIB文件和图像文件。它是只读目录

Documents/存放应用运行时生成的并且需要保留的数据。iTunes或iCloud会在同步设备时备份该目录。当设备发生故障时，可以从iTunes或iCloud恢复该目录中的文件。例如，Homepwner应用可将用户所拥有的物品信息保存在Documents/中

Library/Caches/存放应用运行时生成的需要保留的数据。与Documents/目录不同的是，iTunes或iCloud不会在同步设备时备份该目录。不备份缓存数据的主要原因是相关数据的体积可能会很大，从而延长同步设备所需的时间。如果数据源是在别处（例如Web服务器），就可以将得到的数据保存在Library/Caches/目录。当用户需要恢复设备时，相关的应用只需要从数据源（例如Web服务器）再次获取数据即可

Library/Preferences/存放所有的偏好设置，iOS的设置（Settings）应用也会在该目录中查找应用的设置信息。使用NSUserDefaults类（第26章会介绍），可以通过Library/Preferences目录中的某个特定文件以键-值对的形式保存数据。iTunes或iCloud会在同步设备时备份该目录

tmp/存放应用运行时所需的临时数据。当某个应用没有运行时，iOS系统可能会清除该应用的tmp/目录下的文件，但是为了节约用户设备空间，不能依赖这种自动清除机制，而是当应用不再需要使用tmp/目录中的文件时，就及时手动删除这些文件。iTunes或iCloud不会在同步设备时备份tmp/目录。通过NSTemporaryDirectory函数可以得到应用沙盒中的tmp/目录的全路径

获取文件路径

下面要为Homepwner增加保存和读取BNRItem对象的功能，具体要求是：将所有的BNRItem对象保存至Documents目录中的某个文件，并由BNRItemStore对象负责该文件的写入与读取。为此，BNRItemStore对象需要获取相应文件的全路径。

在BNRItemStore.m中编写一个新方法，实现上述功能：

```
- (NSString *)itemArchivePath
{
// 注意第一个参数是NSDocumentDirectory而不是NSDocumentationDirectory
NSArray *documentDirectories =
NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
// 从documentDirectories数组获取第一个，也是唯一文档目录路径
NSString *documentDirectory = [documentDirectories firstObject];
return [documentDirectory
stringByAppendingPathComponent:@"items.archive"];
}
```

通过C函数NSSearchPathForDirectoriesInDomains可以得到沙盒中的某种目录的全路径。该函数有三个实参，其中后两个实参需要传入固定的值（该函数源自Mac OS X，而在Mac OS X中，可以为后两个实参传入其他值）。第一个实参是NSSearchPathDirectory类型的常量，负责指定目录的类型。例如，传入NSCachesDirectory可以得到沙盒中的Caches目录的路径。

将某个NSSearchPathDirectory常量（例如NSDocumentDirectory）作为关键词查找文档，就能找到其他的常量。需要注意的是，这些常量不区分iOS或Mac OS X，其中的部分常量不能在iOS应用中使用。

NSSearchPathForDirectoriesInDomains函数的返回值是NSArray对象，包含的都是NSString对象。为什么该函数的返回值不是一个NSString对象？这是因为对Mac OS X，可能会有多个目录匹配某组指定的查询条件。但是在iOS上，一种目录类型只会有一个匹配的目录。所以上面这段代码会获取数组的第一个也是唯一的一个NSString对象，然后在该字符串的后面追加固化文件的文件名，并最终得到保存BNRItem对象的文件路径。

18.3 NSKeyedArchiver与NSKeyedUnarchiver

18.2节修改了BNRItem类，能使Homepwner固化BNRItem对象。此外，还介绍了保存数据所需的目录。最后要解决的两个问题是：①如何保存或读取数据？②何时保存或读取数据？下面先解决“保存”问题。Homepwner应该在退出（exit）时，通过NSKeyedArchiver类保存BNRItem对象。

在BNRItemStore.h中声明一个新方法，代码如下：

```
- (BOOL)saveChanges;
```

在BNRItemStore.m中实现该方法，向NSKeyedArchiver类发送archiveRootObject: toFile:消息，代码如下：

```
- (BOOL)saveChanges

{

NSString *path = [self itemArchivePath];

// 如果固化成功就返回YES

return [NSKeyedArchiver archiveRootObject:self.privateItems

toFile:path];

}
```

这段代码中的archiveRootObject:toFile:会将privateItems中的所有BNRItem对象都保存至路径为itemArchivePath的文件。代码本身很简单，其工作原理如下：

•archiveRootObject:toFile:会先创建一个NSKeyedArchiver对象。（NSKeyed- Archiver是抽象类NSCoder的具体实现子类。）

•然后，archiveRootObject:toFile:会向privateItems发送encodeWithCoder:消息，并传入NSKeyedArchiver对象作为第一个参数。

•privateItems的encodeWithCoder:方法会向其包含的所有BNRItem对象发送encodeWithCoder:消息，并传入同一个NSKeyedArchiver对象。这些BNRItem对象都会将其属性编码至同一个NSKeyedArchiver对象（见图18-4）。

•当所有的对象都完成编码后，NSKeyedArchiver对象就会将数据写入指定的文件。

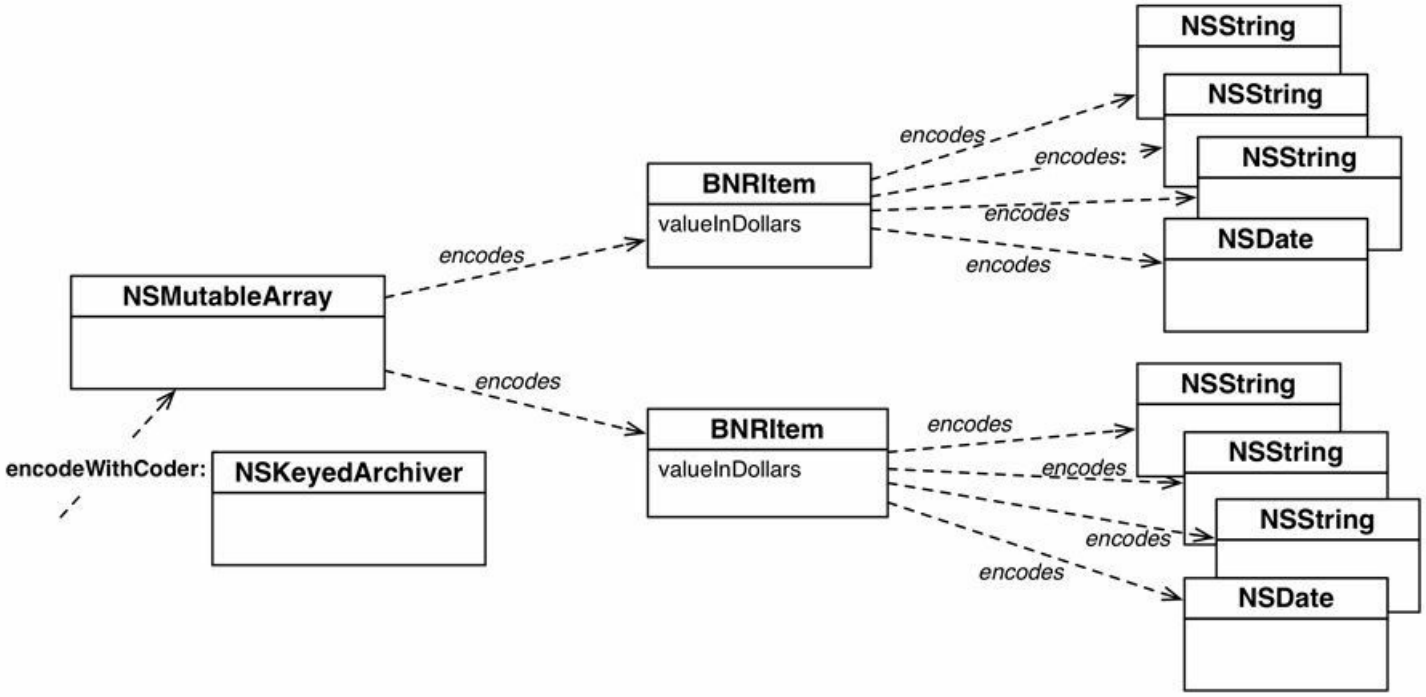


图18-4 固化privateItems中包含的BNRItem对象

在用户按下设备的主屏幕按钮后，BNRAppDelegate对象会收到application-DidEnterBackground:消息。Homepwner应该在applicationDidEnterBackground:中向BNRItemStore对象发送saveChanges消息。

在BNRAppDelegate.m顶部导入BNRItemStore.h，然后实现applicationDid- EnterBackground:，保存所有的BNRItem对象，代码如下：

```
#import “BNRItemStore.h”

@implementation HomepwnerAppDelegate

-（void）applicationDidEnterBackground:（UIApplication *）application

{

BOOL success = [[BNRItemStore sharedStore] saveChanges];

if（success）{

NSLog（@“Saved all of the BNRItems”）;

} else {

NSLog（@“Could not save any of the BNRItems”）;

}

}
```

（Xcode在创建Homepwner项目时，可能已经通过模板为BNRAppDelegate实现了applicationDidEnterBackground:。如果BNRAppDelegate.m已经有了一个名为applicationDidEnterBackground:的方法，就应该在已有方法的尾部增加上述代码，而不是重复加入同一个方法。）

针对模拟器构建并运行应用。增加若干BNRItem对象，然后按下主屏幕按钮，退出至主屏幕。Homepwner应该会在控制台输出相应的提示信息，表示已经保存了全部BNRItem对象。

虽然Homepwner现在还不能读取之前保存的文件并还原BNRItem对象，但是可以验证应用是否将某些数据保存至了指定的文件。在Finder中，使用快捷键Command-Shift-G，然后在文本框中输入~/Library/ApplicationSupport/iPhoneSimulator，最后单击“前往”（Go）按钮。Finder会打开指定的目录，模拟器会在该目录中保存所有已安装的应用和相应的程序包。

根据构建并运行应用时所指定的iOS版本（这里以iOS 7.0为例），打开相应的目录（7.0）。打开Applications目录，应该可以看到安装至模拟器的全部应用。因为这些目录的目录名并没有包含应用的名称，所以只能逐个打开查找，直到找到包含Homepwner应用的那个。

在包含Homepwner的目录中，打开Documents目录（见图18-5）。Finder应该会显示一个名为items.archive的文件。建议读者为iPhone Simulator目录创建一个“快捷方式”（alias），方便将来再次访问。

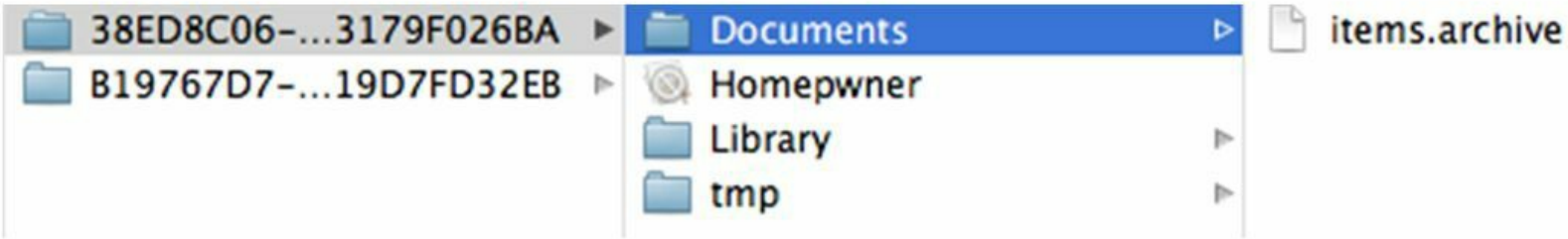


图18-5 Homepwner的沙盒

下面继续为Homepwner添加功能，使之能够读取前面保存的文件。为了能在Homepwner启动时载入之前保存的全部BNRItem对象，需要在创建BNRItemStore对象时使用NSKeyedUnarchiver类。在BNRItemStore.m中，将以下代码加入initPrivate。

```
- (instancetype)initPrivate

{

self = [super init];

if (self) {

_privateItems = [[NSMutableArray alloc] init];

NSString *path = [self itemArchivePath];
```

```
_privateItems = [NSKeyedUnarchiver unarchiveObjectWithFile:path];

// 如果之前没有保存过privateItems，就创建一个新的

if（！_privateItems）{

_privateItems = [[NSMutableArray alloc] init];

}

}

return self;

}
```

这段代码中的unarchiveObjectWithFile:类方法会创建一个NSKeyedUnarchiver对象，然后根据指定的路径载入固化文件。接着，NSKeyedUnarchiver类会查看固化文件中的根对象，然后根据根对象的类型创建相应的对象。Homepwner在创建固化文件时，使用的根对象是NSMutableArray对象，所以解固时的根对象也是NSMutableArray（如果根对象是BNRItem对象，那么unarchiveObjectWithFile:也会返回BNRItem对象）。

创建完NSMutableArray对象后，NSKeyedUnarchiver的unarchiveObjectWithFile:方法会向新创建的NSMutableArray对象发送initWithCoder:消息，并将NSKeyedUnarchiver对象作为实参传入。NSMutableArray对象会通过NSKeyedUnarchiver对象解码相关的对象（BNRItem对象），向所有解固后的对象发送initWithCoder:消息，传入同一个NSKeyedUnarchiver对象。

构建并运行应用，增加若干BNRItem对象，然后终止Homepwner。再次运行Homepwner，应该会看到之前创建的BNRItem对象。测试Homepwner的保存与读取功能时要注意：如果是通过单击Xcode的Stop按钮来终止Homepwner的，那么Homepwner将没有机会保存BNRItem对象。所以读者必须先按下主屏幕按钮，然后再单击Xcode的Stop按钮。

之前因为无法保存BNRItem对象，所以Homepwner创建的都是随机的BNRItem对象，以方便测试。更新后的Homepwner已经支持保存并读取BNRItem对象，所以没有必要再创建随机的BNRItem对象。更新BNRItemStore.m中的createItem方法，修改为创建空的BNRItem对象，代码如下：

```
-（BNRItem *）createItem

{

~~BNRItem *item = [BNRItem randomItem];~~

BNRItem *item = [[BNRItem alloc] init];

[self.privateItems addObject:item];
```

```
return item;

}
```

18.4 应用状态与状态切换

为了保存BNRItem对象，Homepwner会在进入后台运行状态(background state)时进行固化操作。本节将介绍iOS应用可以拥有的各种状态、导致应用切换状态的原因，以及如何在应用发生状态切换时执行指定的代码。图18-6是这些信息的总览图。

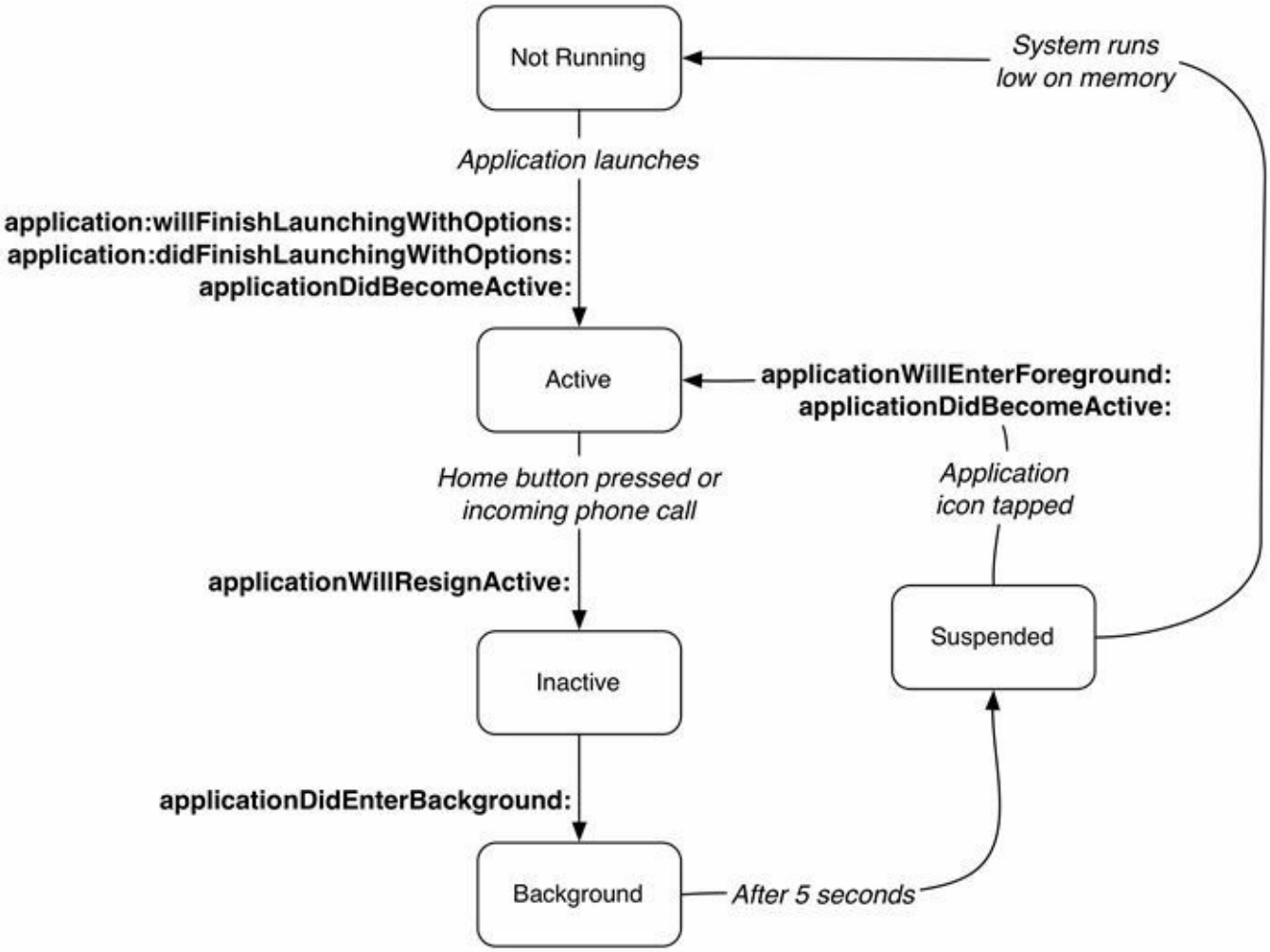


图18-6 iOS应用的多种状态

当应用没有运行时，会处在未运行状态(not running state)，不会执行任何代码，也不会占用RAM。

当应用启动后，会进入激活状态(active state)，可以显示界面、接收事件并处理事件。

当应用处在激活状态时，可能会被某个系统事件打断，临时进入未激活状态(inactive state)。这类系统事件包括收到短消息、收到推送、来电或闹钟到点等。发生系统事件时，iOS会显示相应的提示界面并遮住当前应用的部分界面。当应用处于未激活状态时，其大部分界面是可见的(iOS显示的提示界面只会遮住部分窗口)，也可以执行代码，但是不会接收事件。通常情况下，应用只会在未激活状态停留很短的时间。按下位于iOS设备顶部的锁定按钮，当前处于激活状态的应用会切换至未激活状态，并且会保留未激活状态，直到设备解锁。

当用户按下主屏幕按钮(Home button)时，或者通过某种途径切换至另一个应用时，当前运

行的应用会从激活状态切换为后台运行状态(background state)(实际上，应用会先从激活状态切换为未激活状态，停留极短的时间，然后再进入后台运行状态)。处于后台运行状态的应用仍然可以执行代码，但是其界面不再可见，也不能接收事件。默认情况下，进入后台运行状态的应用有大约10秒的时间，然后会进入挂起状态(suspended state)。读者在开发应用时，不能依赖这个不确定的时间差，而是应该尽快保存用户数据并释放系统资源。

处于挂起状态的应用不能执行代码，其界面也不可见，并且会释放在挂起状态下无须使用的所有资源。挂起的应用就像是进行了“低温干燥”的处理，可以在用户再次启动时快速解冻。表18-1列出了上述应用状态的主要特性。

表18-1 应用的各种状态

状　　态	界面是否可见	是否能接收事件	是否能执行代码
未运行状态	否	否	否
激活状态	是	是	是
未激活状态	大部分	否	是
后台运行状态	否	否	是
挂起状态	否	否	否

连按设备的主屏幕按钮进入多任务界面(multitasking display)，iOS会显示所有处于后台运行状态和挂起状态的应用(还会显示最近运行过的但已经被终止的应用)，如图18-7所示。

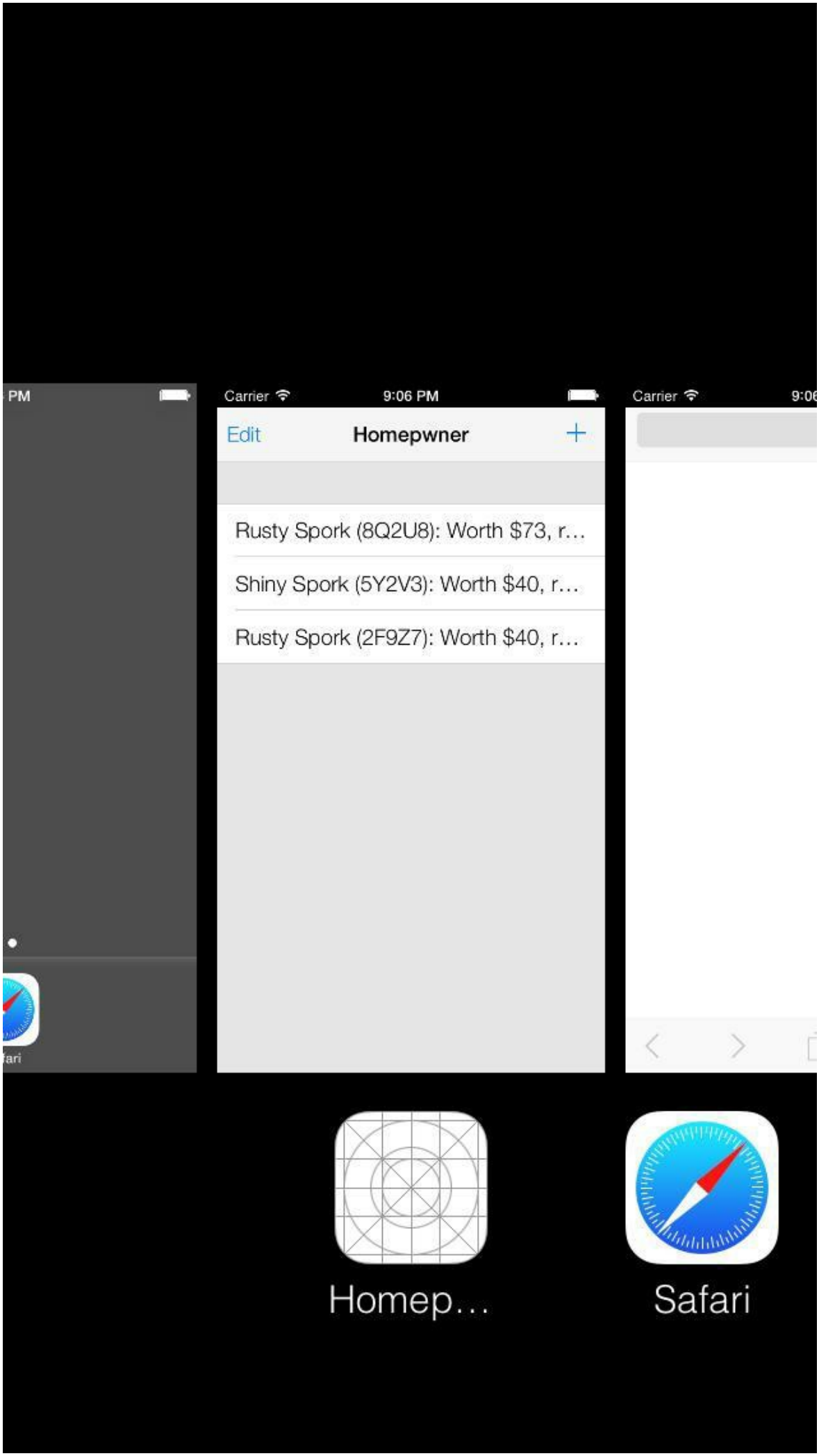

PM
Carrier
9:06 PM
Edit
Homepwner
Rusty Spork (8Q2U8): Worth $73, r…
Shiny Spork (5Y2V3): Worth $40, r…
Rusty Spork (2F9Z7): Worth $40, r…
Carrier
9:06
Homep…
Safari

图18-7 多任务界面中所有处于后台运行状态或挂起状态的应用

当iOS系统认为当前可用的内存过低时，会根据需要终止处于挂起状态的应用。当系统有足够多的空余内存时，处于挂起状态的应用可以一直保留该状态。当处于挂起状态的应用即将被系统终止时，就不会收到相应的通告，系统会直接将其从内存中移除（终止后的应用，其图标可能还会留在多任务界面中，按下图标会重新启动应用）。

当应用状态发生变化时，UIApplication对象的委托对象会收到相应的消息。以下列出的是在UIApplicationDelegate协议中声明的部分消息，这些消息都和应用状态发生变化有关（图18-6描述了这些状态的转化过程）。

- (BOOL)application:(UIApplication *)app

didFinishLaunchingWithOptions:(NSDictionary *)options

- (void)applicationDidBecomeActive:(UIApplication *)app;

- (void)applicationWillResignActive:(UIApplication *)app;

- (void)applicationDidEnterBackground:(UIApplication *)app;

- (void)applicationWillEnterForeground:(UIApplication *)app;

为应用委托对象实现上述方法，就能在应用状态发生变化时执行指定的代码。当应用切换至后台运行状态时，应该保存修改过的数据及应用的各种状态（这里的状态不是指应用状态）。这是因为在应用进入挂起状态前，后台运行状态应用能够执行代码的最后一个状态。一旦应用进入挂起状态，就随时有可能会被iOS系统终止。

18.5 通过NSData将数据写入文件

Homepwner虽然可以固化所有的BNRItem对象，但是并不会保存相关的图片。下面扩充BNRImageStore，实现功能：加入图片时将图片保存为文件；需要时再从文件载入图片。

BNRItem对象的图片应该保存至Documents目录。保存文件时，可以将BNRItem对象的itemKey属性（用户拍摄或选取图片时生成）作为图片文件的文件名。

打开BNRImageStore.m，在类扩展中加入以下方法声明。

```
- (NSString *)imagePathForKey:(NSString *)key;
```

在BNRImageStore.m中实现imagePathForKey:，根据传入的键创建相应的文件路径。

```
- (NSString *)imagePathForKey:(NSString *)key
{
NSArray *documentDirectories =
NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask,
YES);
NSString *documentDirectory = [documentDirectories firstObject];
return [documentDirectory stringByAppendingPathComponent:key];
}
```

为了保存图片，需要先将图片的数据按JPEG格式提取出来，然后拷贝至内存中的某块缓冲区。Foundation框架提供的NSData类可以创建、维护和释放内存缓冲区，所以无须使用malloc这类C函数。NSData对象可以保存一定字节数的二进制数据，下面通过NSData来保存图片数据。

修改BNRImageStore.m中的setImage:forKey:，获取图片路径并保存图片，代码如下：

```
- (void)setImage:(UIImage *)image forKey:(NSString *)key
{
self.dictionary[key] = image;
// 获取保存图片的全路径
NSString *imagePath = [self imagePathForKey:key];
```

```
// 从图片提取JPEG格式的数据

NSData *data = UIImageJPEGRepresentation(image, 0.5);

// 将JPEG格式的数据写入文件

[data writeToFile:imagePath atomically:YES];

}
```

UIImageJPEGRepresentation函数有两个实参，一个是UIImage对象，另一个是浮点数变量，代表压缩质量。压缩质量的值必须在0到1之间，1代表最高质量（不压缩）。该函数会返回一个NSData对象。

向某个NSData对象发送writeToFile:atomically:消息，可以将该对象中的数据写入指定的文件。writeToFile:atomically:的第一个实参负责指定文件路径。第二个实参atomically是一个布尔值，当atomically为YES时，NSData对象会先将数据写入某个临时文件，然后等写入操作成功后再将文件移至第一个实参所指定的路径，并覆盖已有的文件。这样，即使应用在写入文件的过程中崩溃，也不会损坏现有的数据。

需要注意的是，这种将数据写入文件的方式不是固化。虽然NSData对象自身也可以固化，但writeToFile:atomically:的工作原理是将NSData对象中的数据逐字节复制到文件中。

修改BNRImageStore.m中的deleteImageForKey:，在移除指定的UIImage对象后，删除相应的图片文件。

```
- (void)deleteImageForKey:(NSString *)key

{

if (!key) {

return;

}

[self.dictionary removeObjectForKey:key];

NSString *imagePath = [self imagePathForKey:key];

[[NSFileManager defaultManager] removeItemAtPath:imagePath

error:nil];

}
```

修改后的BNRImageStore对象会将所有的图片保存在各自的文件中。当ImageStore对象要返

回某个BNRItem对象的图片时，就需要读取相应的文件。通过UIImage的类方法imageWithContentsOfFile:可以从指定的文件载入图片。

修改BNRImageStore.m中的imageForKey:，使BNRImageStore对象能够通过文件创建图片（如果BNRImageStore对象已经包含指定的图片，就直接返回该图片）。

```
- (UIImage *)imageForKey:(NSString *)key

{

~~return self.dictionary[key];~~

// 先尝试通过字典对象获取图片

UIImage *result = self.dictionary[key];

if (！result) {

NSString *imagePath = [self imagePathForKey:key];

// 通过文件创建UIImage对象

result = [UIImage imageWithContentsOfFile:imagePath];

// 如果能够通过文件创建图片，就将其放入缓存

if (result) {

self.dictionary[key] = result;

}

else {

NSLog(@“Error: unable to find %@”, [self imagePathForKey:key]);

}

}

return result;

}
```

构建并运行应用，为某个BNRItem对象拍摄一张照片，然后按下Home键。再次启动Homepwner，选中之前设置了照片的BNRItem对象，BNRDetailViewController对象的视图应该会显示所有已存的详细信息，包括之前拍摄的照片。

Homepwner会在用户拍摄照片后立刻将相应的图片存入文件，但是对于BNRItem对象中的其他详细信息，则会在应用进入后台运行状态时再保存。之所以要尽快地保存图片，是因为图片的尺寸很大，会占用太多的存储空间。

18.6 NSNotificationCenter和内存过低警告

当iOS设备内存不足时，系统会向运行中的应用发送一条内存过低警告通知。应用收到该通知后，应该立刻释放当前不需要使用的资源以及后期可以重新创建的对象。视图控制器在收到该通知的同时还会收到didReceiveMemoryWarning消息。

除了视图控制器，其他对象中也可能存在需要及时释放的数据。BNRImageStore对象就是如此：它可以释放当前没有显示的UIImage对象，等到需要时再从文件系统载入。

为了使视图控制器以外的对象也可以在内存不足时释放数据，必须使用通知中心（notification center）。每一个iOS应用中都有一个NSNotificationCenter对象，读者可以将其视作智能的通知栏，对象可以将自己注册为某个通知的观察者（observer），例如："如果有人找到了我丢失的狗，请立刻通知我。"当另一个对象发送了一条通知："我找到了一条狗。"通知中心就会将这条通知发送给对应的观察者。

内存过低警告通知的名称是UIApplicationDidReceiveMemoryWarning- Notification。凡是需要自己处理内存过低警告的对象，都可以注册并接收这个通知。编辑BNRImageStore.m中的initPrivate方法，将BNRImageStore对象注册为UIApplicationDidReceiveMemoryWarningNotification通知的观察者，代码如下：

```
- (instancetype)initPrivate

{

self = [super init];

if (self) {

_dictionary = [[NSMutableDictionary alloc] init];

NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];

[nc addObserver:self

selector:@selector(clearCache:)

name:UIApplicationDidReceiveMemoryWarningNotification

object:nil];

}

return self;

}
```

现在BNRImageStore对象已经注册为通知中心的观察者了（见图18-8）。

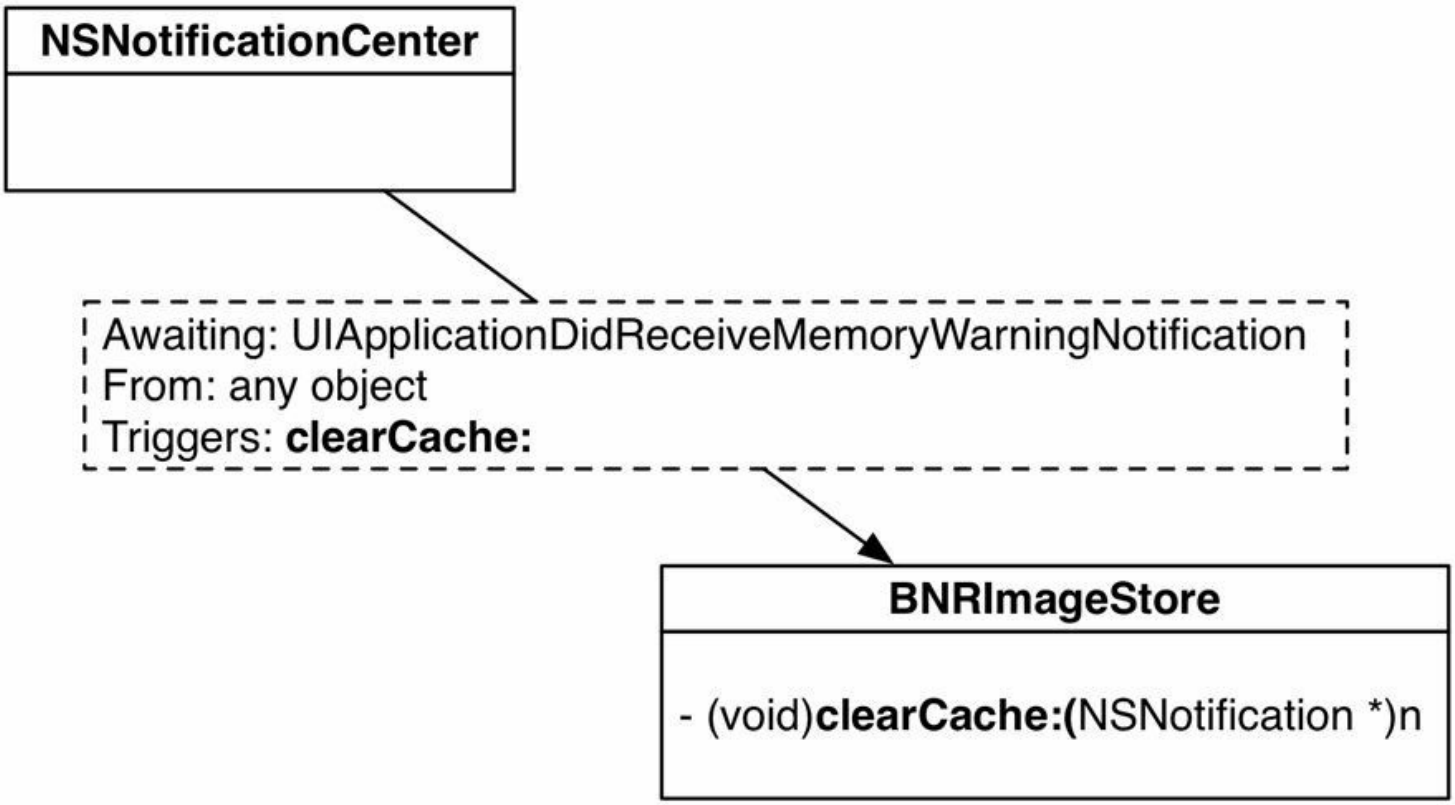


图18-8 注册为通知中心的观察者

当系统发出内存过低警告通知时，BNRImageStore对象会收到UIApplicationDid-ReceiveMemoryWarningNotification通知，并调用clearCache:方法清除缓存（见图18-9）。

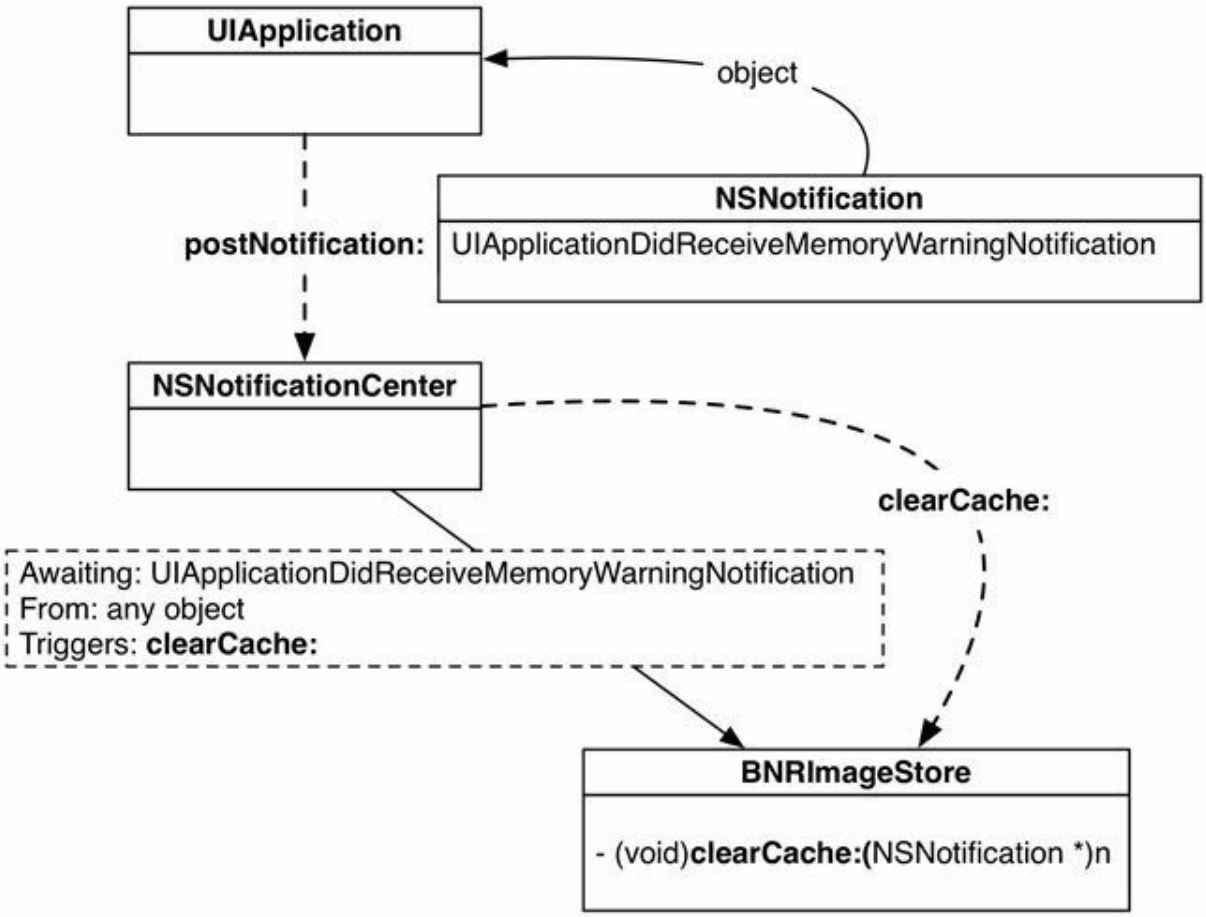


图18-9 接收通知

在BNRImageStore.m中实现clearCache:，移除BNRImageStore对象所包含的所有UIImage对象。

```
- (void)clearCache:(NSNotification *)note

{

NSLog(@"flushing %d images out of the cache", [self.dictionary count]);

[self.dictionary removeAllObjects];

}
```

当字典对象移除某个对象时，被移除对象的拥有方个数会减1。因此，当BNRImageStore对象移除其包含的所有UIImage对象时，这些UIImage对象的拥有方个数都会减1。减1后，没有拥有方的UIImage对象会被释放，并在需要时由BNRImageStore对象根据相应的文件再次创建。有些UIImage对象可能还会有其他拥有方，这些UIImage对象不会被释放。例如，BNRDetailViewController对象的imageView会保留某个UIImage对象，那么该对象不会被释放。当imageView放弃之前保留的UIImage对象时（例如UINavigationController对象弹出了BNRDetailViewController对象，或者应用将新的UIImage对象赋给了imageView），相应的UIImage对象才会被释放。

在模拟器中构建并运行应用，加入一些图片，然后在Hardware（硬件）菜单中选择Simulate Memory Warning（模拟内存警告），控制台中会输出日志信息，提示缓存图片已经被清除。

深入学习NSNotificationCenter

通知与委托、目标-动作相同，也是回调的一种形式。它们的区别是，委托和目标-动作需要对象直接向自己的委托或目标发送消息，而通知使用一位中介者：NSNotificationCenter。

Objective-C使用NSNotification表示通知，每个NSNotification对象都具有名称（NSNotificationCenter根据该名称检索此通知的所有观察者）、来源对象（发布该通知的对象）和可选的userInfo字典（来源对象需要告诉观察者的额外信息）。例如，当系统状态栏的frame发生变化后，UIApplication会发布名为UIApplicationDidChangeStatus- BarFrameNotification的通知，并带有一个存储状态栏的新frame的userInfo字典。如果代码中收到了该通知，则可以使用以下方式获取状态栏的新frame：

```
- (void)statusBarMovedOrResized:(NSNotification *)note

{

NSDictionary *userInfo = [note userInfo];
```

```
NSValue *wrappedRect = userInfo[UIApplicationStatusBarFrameUserInfoKey];

CGRect newFrame = [wrappedRect CGRectValue];

……使用newFrame……

}
```

需要注意的是，如果需要向NSDictionary中存储CGRect结构，必须先将其包装为NSValue对象——NSDictionary中只能存储Objective-C对象。

那么，如何了解userInfo字典中存储了哪些内容呢？类参考手册中会详细说明类发布的通知，大部分通知的说明都是："This notification does not contain a userInfo dictionary.（该通知不会附带userInfo字典。）"对于带有userInfo字典的通知，文档中会列出字典中的所有键及其对应值的含义。

addObserver:selector:name:object:方法的最后一个参数通常是nil——假设观察的通知名称为"Fire！"，传入nil表示接受所有对象发布的"Fire！"通知。如果为该参数传入某个对象，那么观察者只会接受该对象发布的"Fire！"通知，其他对象发布的"Fire！"通知则会被忽略。

通知允许多个对象将回调函数注册到相同的事件上。同一个通知可以有许多观察者，当通知发布后，所有观察者都会执行之前注册的回调函数（同时执行，没有先后顺序）。如果多个对象需要监听同一个事件，则通知是最好的解决方案。例如，许多对象都需要在设备方向发生变化时执行一系列操作，因此该事件发生时系统会发布名为UIDeviceOrientationDid-ChangeNotification的通知。

最后需要说明的是，NSNotification和NSNotificationCenter与用户看到的推送通知（push notification）或本地通知（local notification）无关，NSNotificationCenter只用来处理应用内部的对象通信。

18.7 模型-视图-控制器-存储设计模式

本章扩充了BNRItemStore对象，它能够将其包含的全部BNRItem对象存入文件，并根据之前保存的文件重建BNRItem对象。Homepwner中的控制对象会向BNRItemStore对象查询各自需要的模型对象，并且不会关心这些模型对象的出处。Homepwner中的控制对象只会使用模型对象，所有和创建模型对象有关的工作都将由BNRItemStore对象来完成。

标准的模型-视图-控制器设计模式要求控制对象负责模型对象的保存和读取。但在实际情况中，这样做的效果并不是很好。控制对象的主要任务是处理模型对象和视图对象之间的交互，如果还要负责实现所有的存取细节，则可能会“不堪重负”。为此，可以将模型对象的存取逻辑移入另一类对象：存储对象。

通过存储对象所提供的方法，控制对象可以获取或保存模型对象。保存和读取模型对象的实现细节全部由存储对象负责。以本章中的Homepwner为例，存储对象（BNRItemStore对象）会通过某个指定的文件来创建和保存BNRItem对象。除了文件，存储对象也可以使用数据库、Web服务或其他途径来为控制对象创建模型对象。

除了能够简化控制器类，这种设计模式的另一个好处是：不用修改控制对象或应用的其他部分，就能修改存储对象的工作方式。这种修改可以很简单，例如修改数据的目录结构；也可以很复杂，例如修改数据的格式。因此，无论某个应用有多少个需要存取数据的控制对象，都只需要修改相应的存储对象即可。

本书之前介绍过模型-视图-控制器，本章将这种设计模式拓展为模型-视图-控制器-存储（Model-View-Controller-Store）。

18.8 初级练习 : PNG

将图片的数据按PNG的格式提取，然后写入文件。

18.9 深入学习：应用的状态切换

下面通过编写一些简单的测试代码，帮助读者理解应用的各种状态切换。

本书之前介绍过隐式变量self，self指向执行当前方法的对象。除了self，还有一个名为_cmd的隐式变量，它是当前方法的选择器。NSStringFromSelector函数可以根据指定的选择器生成相应的字符串。

在BNRAppDelegate.m中，实现和应用状态切换有关的委托方法，并向控制台输出当前方法的方法名。需要加入的方法共有四个（在加入方法前，先检查Xcode是否已经在BNRAppDelegate.m中创建了相应的方法），代码如下：

```
- (void)applicationWillResignActive:(UIApplication *)application

{

NSLog(@"%@", NSStringFromSelector(_cmd));

}

- (void)applicationWillEnterForeground:(UIApplication *)application

{

NSLog(@"%@", NSStringFromSelector(_cmd));

}

- (void)applicationDidBecomeActive:(UIApplication *)application

{

NSLog(@"%@", NSStringFromSelector(_cmd));

}

- (void)applicationWillTerminate:(UIApplication *)application

{

NSLog(@"%@", NSStringFromSelector(_cmd));

}
```

在application:didFinishLaunchingWithOptions:起始处和applicationDid- EnterBackground:起始处加入以下代码。

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
NSLog(@"%@", NSStringFromSelector(_cmd));
…
}
- (void)applicationDidEnterBackground:(UIApplication *)application
{
NSLog(@"%@", NSStringFromSelector(_cmd));
[[BNRItemStore sharedStore] saveChanges];
}
```

构建并运行应用。控制台应该会先输出application:didFinishLaunching- WithOptions:，然后输出applicationDidBecomeActive:。

按下主屏幕按钮，通过控制台的输出可知，Homepwner会先进入未激活状态，然后马上进入后台运行状态。按下主屏幕上的Homepwner图标或者在多任务界面选择Homepwner，重新启动应用。通过控制台的输出可知，Homepwner会先进入前台工作状态，然后进入激活状态。

按下主屏幕按钮，再次退出应用。连按主屏幕按钮，打开多任务界面，将Homepwner界面的缩略图向上滑出屏幕，系统会立刻终止Homepwner，但是Homepwner的应用委托对象不会收到任何消息。

18.10 深入学习：文件系统的读取和写入

除了固化和NSData，还有其他一些途径可以实现数据的存取。下面介绍其中几种主要途径（Core Data会在第23章介绍）。

编写iOS应用时，也可以使用C语言库的标准I/O函数，代码如下：

```
FILE *inFile = fopen（“textfile”, “rt”）;

char *buffer = malloc（someSize）;

fread（buffer, byteCount, 1, inFile）;

FILE *outFile = fopen（“binaryfile”, “w”）;

fwrite（buffer, byteCount, 1, outFile）;
```

但是在编写iOS应用时，标准I/O函数并不常用。这是因为还有其他的途径可以读/写二进制数据或文本数据，而且更方便。如果要读/写二进制数据，则可以使用NSData。如果要读/写文本数据，则可以分别使用NSString的实例方法initWithContentsOfFile:和writeToFile:atomically:encoding:error:，代码如下：

```
// 用于保存NSError对象地址的局部指针变量

NSError *err;

NSString *someString = @“Text Data”;

BOOL success = [someString writeToFile:@“/some/path/file”

atomically:YES

encoding:NSUTF8StringEncoding

error:&err];

if（！success）{

NSLog（@“Error writing file: %@”, [err localizedDescription]）;

}

NSString *myEssay

= [[NSString alloc] initWithContentsOfFile:@“/some/path/file”

encoding:NSUTF8StringEncoding
```

```
error:&err];

if（！myEssay）{

NSLog(@“Error reading file: %@”, [err localizedDescription]）;

}
```

先介绍这段代码中的NSError。应用在执行方法时，可能会因为各种原因而导致失败。例如，将数据写入文件时会因为路径无效而失败，也可能会因为没有足够的权限而失败。NSError对象的作用是保存失败的原因。向NSError对象发送localizedDescription消息，可以得到相应错误的描述信息。因为这些信息是给人看的(human-readable)，所以可以向用户显示或通过控制台输出。

在这段代码中，获取NSError对象的语法看上去有些奇怪。通常情况下，只有当某个方法在执行代码时发生了错误，才有必要创建NSError对象。也就是说，负责创建NSError对象的应该是被调用的方法，而不是调用方。为了能让调用方得到被调用的方法所创建的NSError对象，需要将指针变量的地址(&err)作为实参传给相应的方法。假设有某个方法(方法A)，该方法的某个实参可以返回一个NSError对象，那么在调用方法A前，需要先创建一个类型为NSError *的指针变量(局域变量)。注意，这里不需要创建NSError对象，因为这是方法A的任务。然后将这个局部变量的地址(&err)传给可能会出错的方法A。如果方法A在执行代码时发生了错误，就会创建一个NSError对象，并将新创建的对象的地址赋给传入的指针。如果调用方不关心NSError对象，则可以传入nil。

如果要向用户显示错误，通常可以使用UIAlertView对象(见图18-10)。

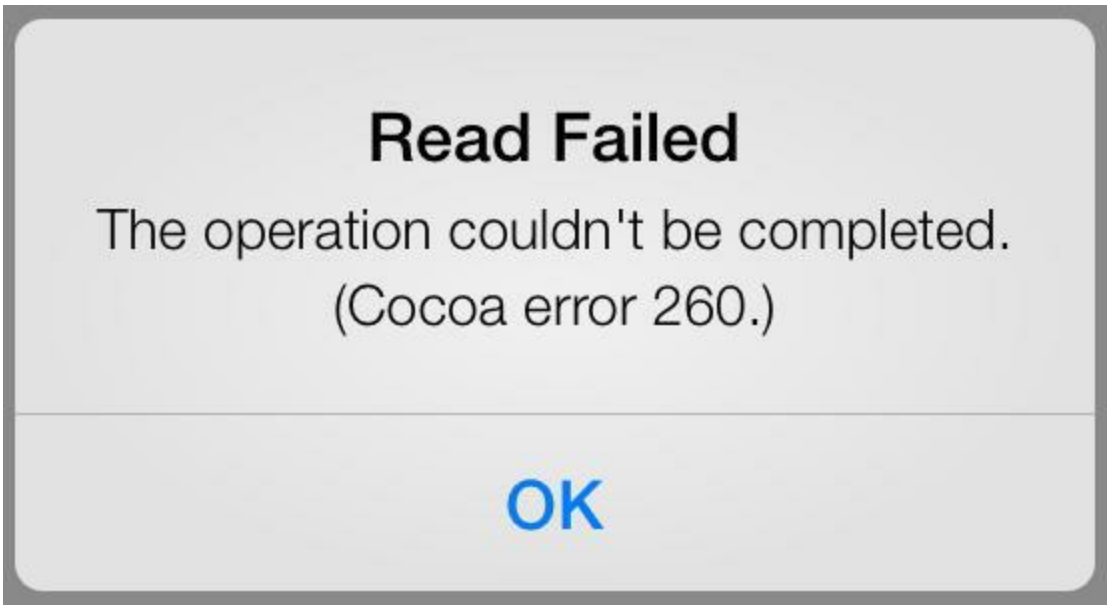


图18-10 UIAlertView示例

可以使用如下代码创建并显示UIAlertView：

```
NSString *myEssay
```

= [[NSString alloc] initWithContentsOfFile:@“/some/path/file”

encoding:NSUTF8StringEncoding

error:&err];

if（！myEssay）{

UIAlertView *a

= [[UIAlertView alloc] initWithTitle:@“Read Failed”

message:[err localizedDescription]

delegate:nil

cancelButtonTitle:@“OK”

otherButtonTitles:nil];

[a show];

}

很多语言将所有非预期（unexpected）错误作为异常抛出，但是Objective-C的异常只用来处理程序错误。当异常抛出时，详细信息都封装在NSException对象中。这些信息主要用来帮助程序员调试代码，例如“试图在只有两个对象的数组中访问第七个对象。”NSException中还包括方法调用栈信息，指明了抛出异常的代码位置。

NSException和NSError的使用场景不同。如果需要指出程序员的编码错误，则应该使用NSException。例如，一个方法只能接受奇数作为参数，但是程序员在调用该方法时传入了偶数，这时应该抛出异常，以方便程序员解决代码错误。相反，对于预期（expected）错误，如用户错误和设备环境错误，应该使用NSError。例如，一个方法需要读取用户照片，但是没有访问用户相册的权限，这时应该向方法调用者返回一个NSError对象，指出不能执行本次操作的原因。

现在继续讨论文件读/写。和NSString类似，NSDictionary和NSArray也有writeToFile:和initWithContentsOfFile:。只有当collection对象包含可序列化（property list serializable）对象时，才能通过writeToFile:这类方法将数据存入文件。可序列化对象包括NSString、NSNumber、NSDate、NSData、NSArray和NSDictionary。NSArray对象或NSDictionary对象这类文件的写入方法生成的都是XML格式的property list文件，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">
<array>
<dict>
<key>firstName</key>
<string>Christian</string>
<key>lastName</key>
<string>Keur</string>
</dict>
<dict>
<key>firstName</key>
<string>Joe</string>
<key>lastName</key>
<string>Conway</string>
</dict>
<dict>
<key>firstName</key>
<string>Aaron</string>
<key>lastName</key>
<string>Hillegass</string>
</dict>
</array>
</plist>
```

几乎所有的操作系统都能读取XML格式的property list文件，所以使用这种格式存储数据会很方便。很多Web服务程序会采用这种格式的文件作为输入和输出。读/写property list文件的代码如下：

```
NSMutableDictionary *d = [NSMutableDictionary dictionary];
d[@“String”] = @“A string”;
[d writeToFile:@“/some/path/file” atomically:YES];
NSMutableDictionary *anotherD = [[NSMutableDictionary alloc]
initWithContentsOfFile:@“/some/path/file”];
```

18.11 深入学习：应用程序包

Xcode在构建iOS应用时，需要完成的主要工作是创建应用程序包（application bundle）。应用程序包会包含应用的可执行文件和执行应用所需的全部资源。这些资源包括XIB文件、图片和音频文件等，即应用在运行时需要使用的所有文件。将某个文件加入项目时，Xcode会自动判断是否应该将该文件加入应用程序包。

通过以下步骤，可以查看会被加入Homepwner程序包的文件。选中位于项目导航面板顶部的Homepwner条目，选中右侧面板中的Homepwner目标，选择Build Phases面板。展开Copy Bundle Resources（拷贝程序包资源）列表，可以看到一组文件。Xcode会在构建项目时，将这组文件加入应用程序包。

Build Phases面板中的每一个列表，都是Xcode在构建项目时需要经历的阶段。Copy Bundle Resources阶段的任务是将指定的文件（资源文件）全部拷贝至应用程序包。

将某个应用装入模拟器后，可以通过文件系统查看相应的应用程序包（见图18-11）。在Finder中打开~/Library/Application Support/iPhoneSimulator/（版本号）/Applications目录。该目录下的子目录都是已安装应用的沙盒。打开某个子目录，可以看到一个应用沙盒应该包含的目录，其中包括：应用程序包、Documents目录、tmp目录和Library目录。右键单击或Command-单击应用程序包，然后选择上下文菜单中的Show Package Contents（显示包内容）。

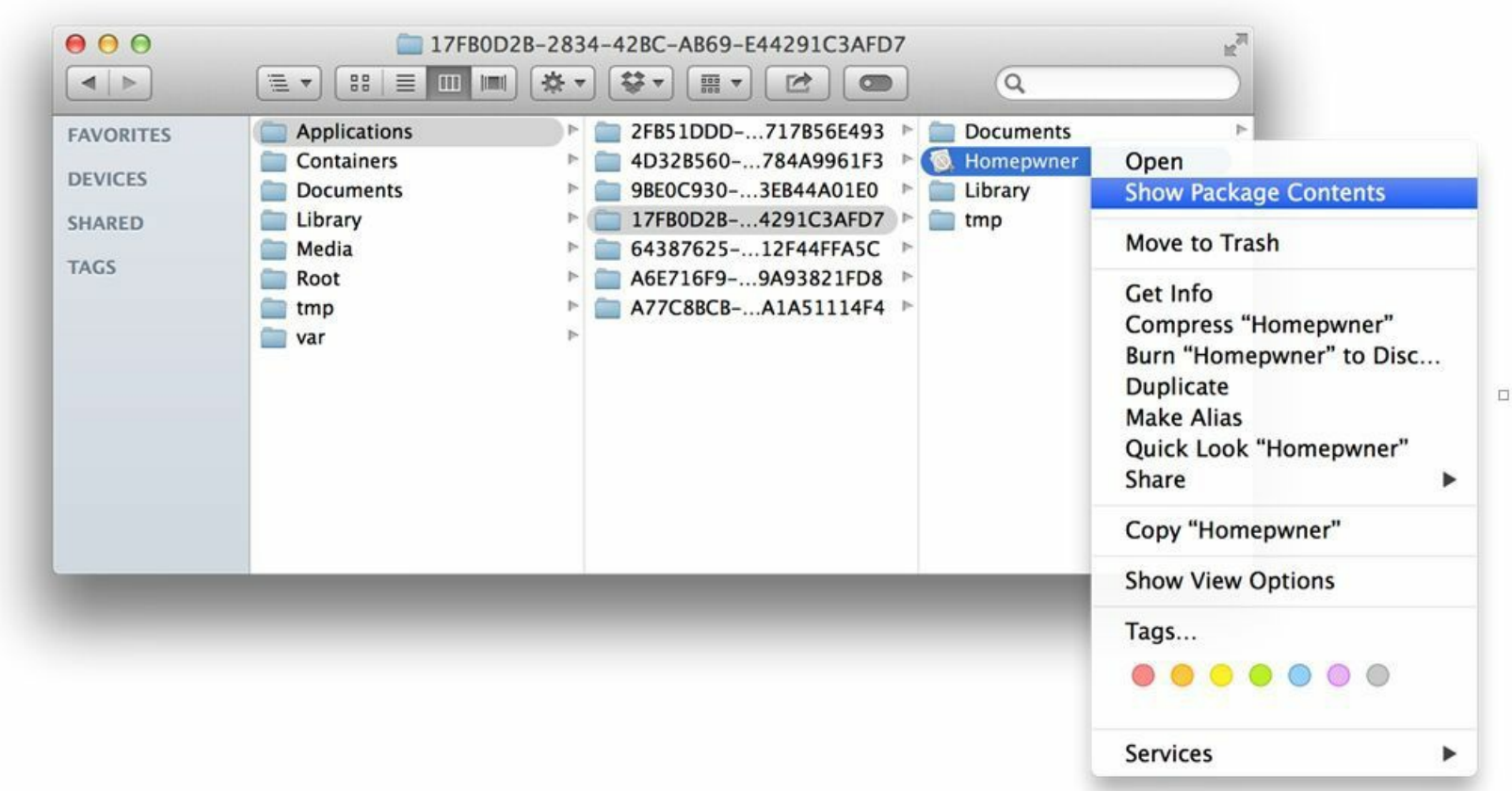


图18-11 查看应用程序包

Finder会打开一个新窗口并显示包内的内容（见图18-12）。当用户通过App Store下载某个应用后，相应的程序包会被拷贝至用户的设备。

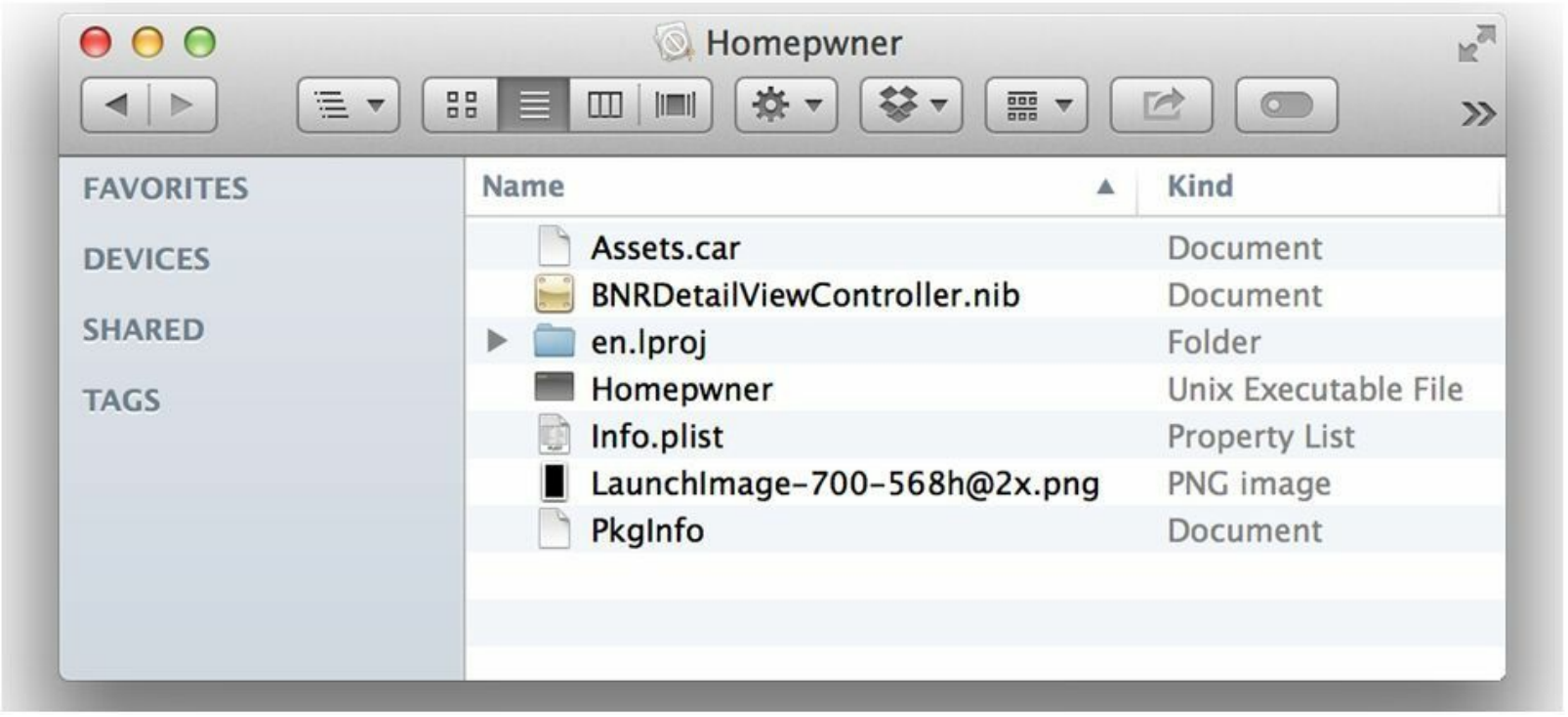


图18-12 应用程序包

iOS应用可以在运行时载入应用程序包中的文件。要获得应用程序包中的某个文件的全路径，需要先得到代表应用程序包的NSBundle对象，然后通过该对象得到某个文件的全路径，代码如下：

```
// 获取代表应用程序包的NSBundle对象

NSBundle *applicationBundle = [NSBundle mainBundle];

// 通过NSBundle对象，获得包内名为myImage.png的文件的全路径

NSString *path = [applicationBundle pathForResource:@“myImage”

ofType:@“png”];
```

调用pathForResource:ofType:后，如果应用程序包没有包含指定的文件，则它会返回nil。如果文件存在，则它会返回该文件的全路径。

应用程序包中的文件都是只读的，不能修改。此外，也不能在运行时向应用程序包添加文件。应用程序包中的文件通常包含：按钮图片、界面音效或初始化模板等。后续章节为了能在运行时载入这类资源，还会再次使用pathForResource:ofType:方法。

第19章 创建UITableViewCell子类

之前章节已经介绍过，UITableView对象显示的是一组UITableViewCell对象。对大多数应用来说，UITableViewCell自带的textLabel、detailTextLabel和imageView就够用了。当读者需要某个UITableViewCell对象显示更多的细节，或者拥有不同的布局时，就需要创建自定义的UITableViewCell子类。

本章将创建一个名为BNRItemCell的UITableViewCell子类，以便能够完整显示BNRItem的信息。完成后的BNRItemCell可以显示BNRItem的名称、价值（美元）、序列号和相应图片的缩略图（见图19-1）。

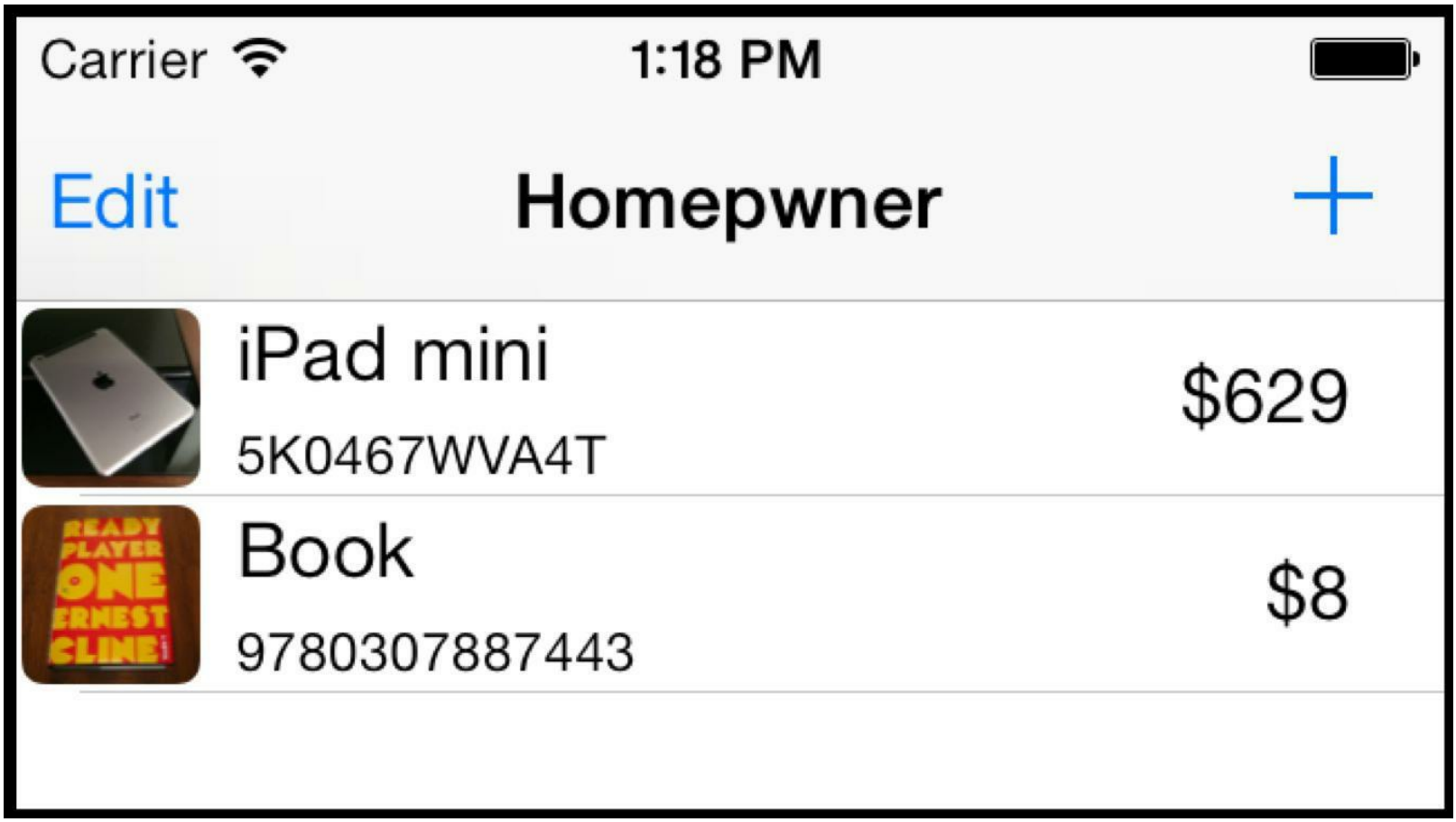


图19-1 改用BNRItemCell后的Homepwner

19.1 创建BNRItemCell

UITableViewCell是UIView的子类。创建UIView子类时，定制界面的方法是覆盖drawRect:，但是在创建UITableViewCell子类时，定制界面的方法是向UITableViewCell加入子视图。不过，并不是直接将子视图加入UITableViewCell，而是加入UITableViewCell的另一个子视图：contentView。

contentView起容器的作用，用于存放其他子视图。这些子视图构成UITableViewCell的布局（见图19-2）。要改变UITableViewCell子类的外观，需要修改contentView所包含的子视图。例如，可以创建UITextField、UILabel和UIButton等视图对象，并将它们加入contentView。

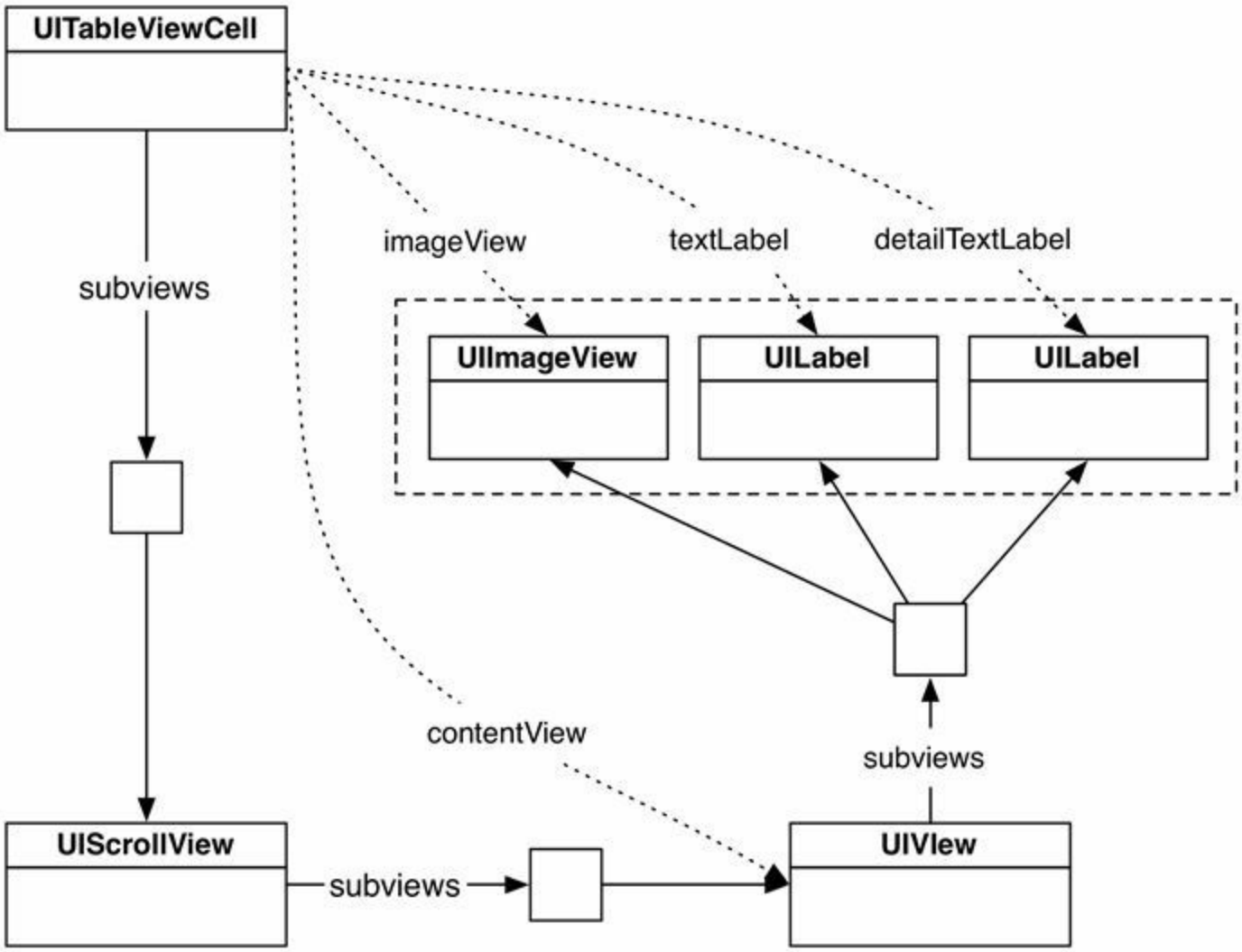


图19-2 UITableViewCell的视图层次结构

必须将子视图加入contentView而不是UITableViewCell对象自身的原因是，UITableViewCell对象会根据外部条件改变contentView的大小。例如，当UITableView对象进入编辑模式时，UITableViewCell对象会改变contentView的大小，为编辑控件（例如删除控件和移位控件）留出位置（见图19-3）。如果直接将子视图加入UITableViewCell对象，编辑控件就会遮住这些子视图。进入编辑模式时，UITableViewCell对象不会改变大小（UITableViewCell对象的宽度必须和UITableView对象的宽度相等），但是其包含的contentView会改变大小。

读者可能已经注意到视图层次结构中的UIScrollView对象，当UITableView对象进入编辑模式时，UITableViewCell对象会将contentView移动到左侧，这个过程需要借助UIScrollView对象。同样，在UITableViewCell对象中从右向左滑动显示删除控件时，也需要借助UIScrollView对象。实际上，contentView是UIScrollView对象的一个子视图。

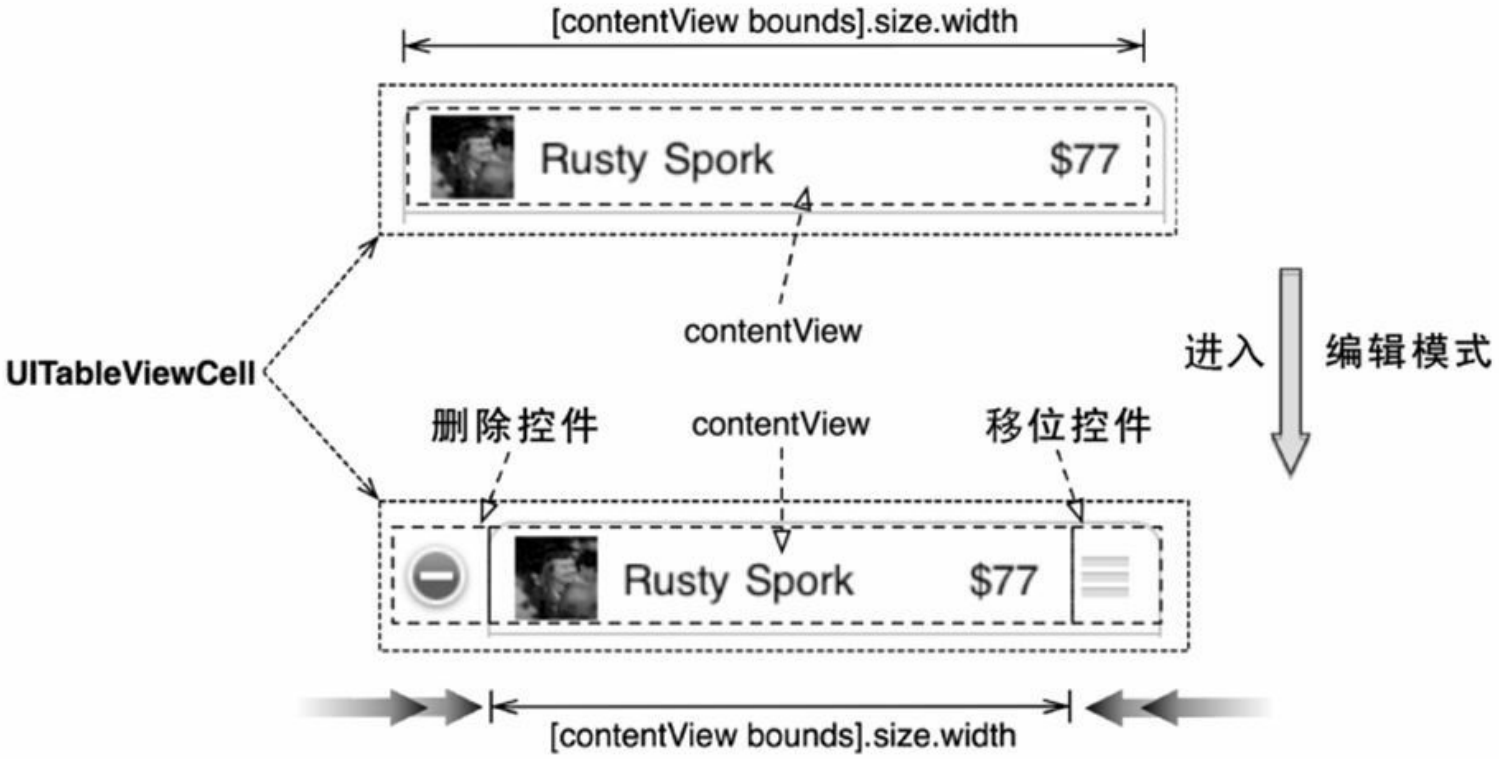


图19-3 UITableViewCell对象的布局（标准模式和编辑模式）

打开Homepwner.xcodeproj。创建一个新的NSObject子类，并将其命名为BNRItemCell。在BNRItemCell.h中，将BNRItemCell的父类修改为UITableViewCell，代码如下：

~~@interface BNRItemCell : NSObject~~

@interface BNRItemCell : UITableViewCell

创建UITableViewCell子类的界面

创建UITableViewCell子类界面的最简单方法就是使用XIB文件。用空应用模板创建一个新的XIB文件并将其命名为BNRItemCell.xib（这里的Device Family无关紧要，使用默认值即可）。

新创建的XIB文件固化了一个BNRItemCell对象，当UITableView需要一个新的BNRItemCell对象时，UITableView会从这个XIB文件中解固BNRItemCell对象。

打开BNRItemCell.xib，从对象库面板中拖曳一个UITableViewCell对象至画布（请注意，是选择UITableViewCell，而不是UITableView或UITableViewController）。

在大纲视图中选中Table View Cell，再打开标识检视面板，在标题为Class的文本框中填入BNRItemCell（见图19-4）。

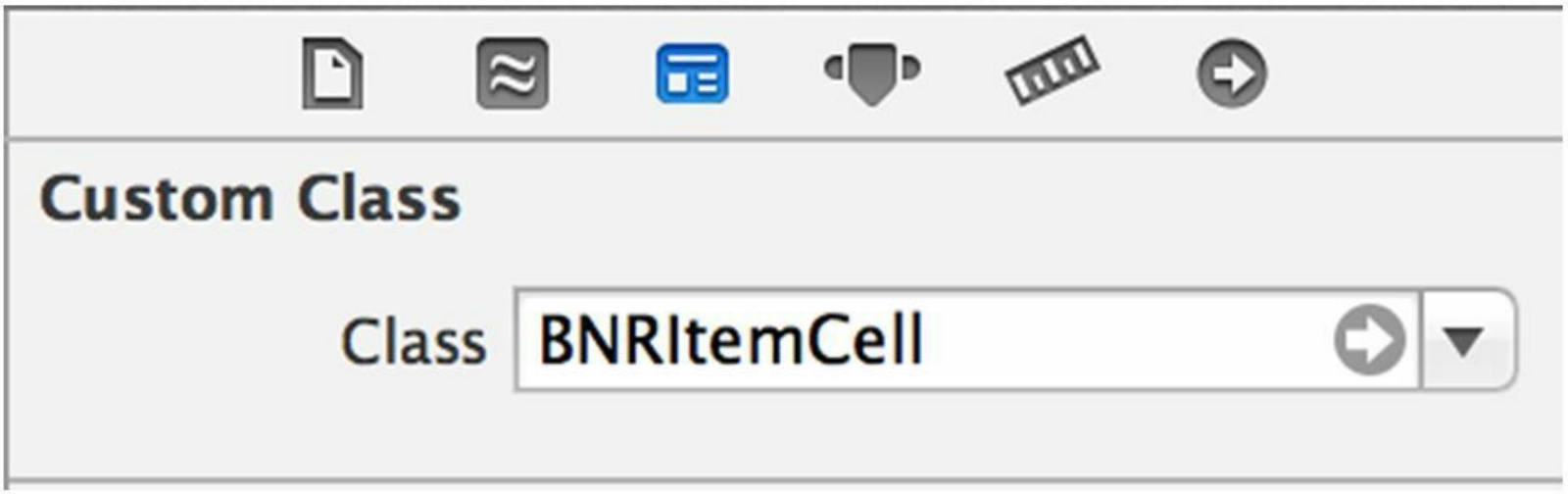


图19-4 修改Class

每个BNRItemCell对象需要显示三个文本标签和一张图片，所以需要拖曳三个UILabel对象和一个UIImageView对象至BNRItemCell。根据图19-5设置视图的大小和位置。请注意，最底部的UILabel对象需要使用较小的字体，文字颜色为深灰色。

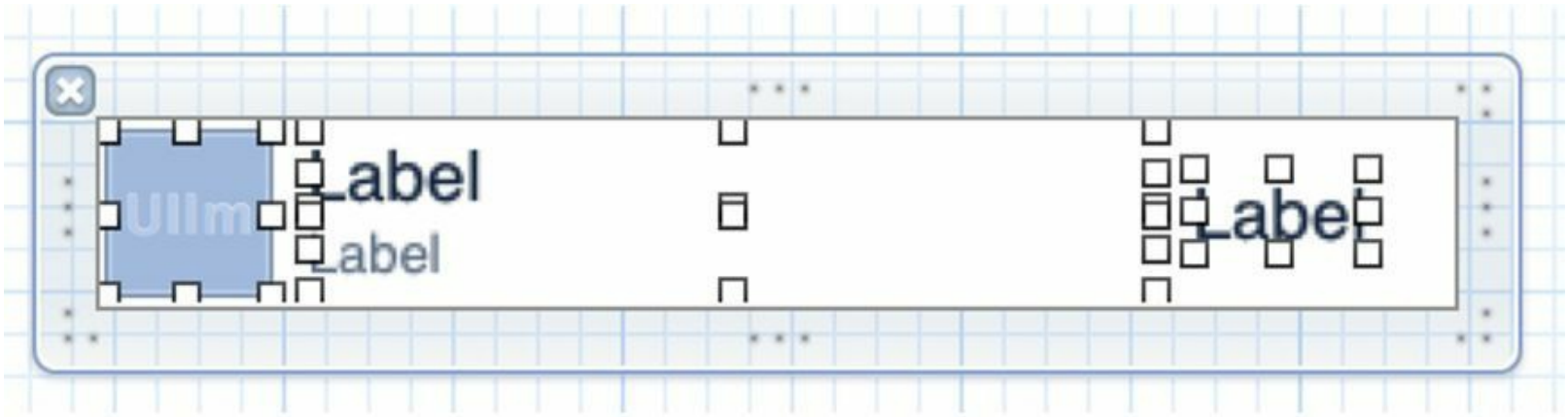


图19-5 BNRItemCell的布局

为BNRItemCell创建并关联插座变量

为了使BNRItemsViewController可以在tableView:cellForRowAtIndexPath:中设置BNRItemCell的界面内容，必须在BNRItemCell中添加相应的插座变量，用来关联三个UILabel对象和UIImageView对象。下面就使用拖曳的方式来创建并关联插座变量。

打开BNRItemCell.xib，然后按住Option并单击BNRItemCell.h，在辅助视图打开BNRItemCell.h。

按住Control，依次将新添加的子视图拖曳至BNRItemCell.h的方法声明区域，再根据图19-6为各个插座变量命名并设置关联特性（注意Connection、Storage和Object）。

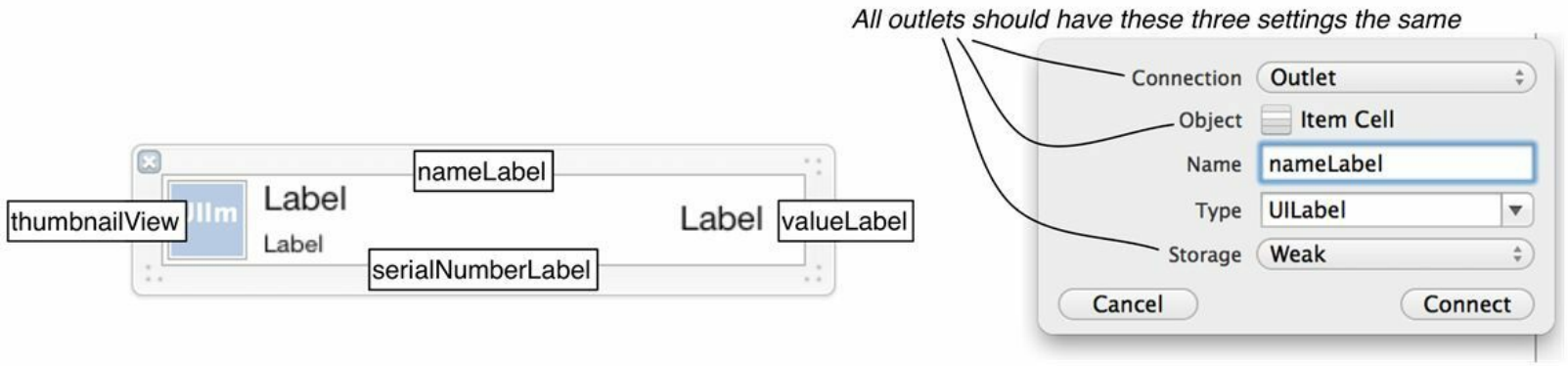


图19-6 BNRItemCell中的关联

现在请读者仔细检查BNRItemCell.h中的代码：

```
@interface BNRItemCell : UITableViewCell

@property (weak, nonatomic) IBOutlet UIImageView *thumbnailView;

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

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

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

@end
```

UITableViewCell的XIB文件不会使用File's Owner，所以不用为其设置类名，也不用为其创建任何关联。与UIViewController的XIB文件不同，UITableViewCell的XIB文件在解固时，不需要使用某个对象代替File's Owner，也不需要将其中的固化对象关联到File's Owner。为了理解两种XIB文件的区别，首先需要知道UITableView加载UITableViewCell的过程。

使用BNRItemCell

下面在BNRItemsViewController的tableView:cellForRowAtIndexPath:中为UITableView对象创建BNRItemCell对象。

首先在BNRItemsViewController.h顶部导入BNRItemCell.h：

```
#import “BNRItemCell.h”
```

在之前章节中，为了让UITableView在需要使用UITableViewCell时创建相应类的对象，需要向UITableView注册UITableViewCell的类；本章通过NIB文件加载UITableViewCell，则需要注册相应的NIB文件。

在BNRItemsViewController.m的viewDidLoad中注册BNRItemCell.xib，并将重用标识设置为BNRItemCell，代码如下：

```
- (void)viewDidLoad

{

[super viewDidLoad];

[self.tableView registerClass:[UITableViewCell class]

forCellReuseIdentifier:@"UITableViewCell"];

// 创建UINib对象，该对象代表包含了BNRItemCell的NIB文件

UINib *nib = [UINib nibWithNibName:@"BNRItemCell" bundle:nil];

// 通过UINib对象注册相应的NIB文件

[self.tableView registerNib:nib

forCellReuseIdentifier:@"BNRItemCell"];

}
```

注册NIB文件的原理非常简单，仅仅是将UINib对象以"BNRItemCell"作为键保存到NSDictionary中。UINib对象包含所有保存在其XIB文件中的数据，当UITableView对象需要使用UITableViewCell对象时，就会使用相应的UINib对象创建新的UITableViewCell对象。

在UITableView对象中注册了包含BNRItemCell.xib的UINib对象之后，UITableView对象就可以通过"BNRItemCell"键找到并加载BNRItemCell对象。

在BNRItemsViewController.m中修改tableView: cellForRowAtIndexPath:，代码如下：

```
- (UITableViewCell *)tableView:(UITableView *)tableView

cellForRowAtIndexPath:(NSIndexPath *)indexPath

{

UITableViewCell *cell =

[tableView dequeueReusableCellWithIdentifier:@"UITableViewCell"

forIndexPath:indexPath];

// 获取BNRItemCell对象，返回的可能是现有的对象，也可能是新创建的对象
```

```
BNRItemCell *cell =

[tableView dequeueReusableCellWithIdentifier:@“BNRItemCell”];

forIndexPath:indexPath];

NSArray *items = [[BNRItemStore sharedStore] allItems];

BNRItem *item = items[indexPath.row];

cell.textLabel.text = item.description;

// 根据BNRItem对象设置BNRItemCell对象

cell.nameLabel.text = item.itemName;

cell.serialNumberLabel.text = item.serialNumber;

cell.valueLabel.text =

[NSString stringWithFormat:@“$%d”, item.valueInDollars];

return cell;

}
```

首先，由于创建了UITableViewCell子类，因此需要修改重用标识；其次，当设置BNRItemCell对象时，需要将BNRItem对象的各个属性赋给对应UILabel对象的text属性（本章稍后再处理UIImageView对象）。

构建并运行应用，创建一个新的BNRItem对象。可以发现，BNRItemCell对象已经可以正确加载了，不过显示效果并不好：目前所有子视图都没有添加约束。下面就在XIB文件中为BNRItemCell对象的各个子视图添加约束。

为**BNRItemCell**添加约束

BNRItemsViewController的UITableView对象会根据设备屏幕尺寸自动调整大小。当UITableView对象的宽度改变时，其中的所有UITableViewCell对象也会自动改变宽度，保持与UITableView对象的宽度相等。因此，必须为之前添加的子视图添加约束，以便自动适配UITableViewCell对象的当前宽度。（UITableViewCell对象的高度通常不会改变。只有手动设置UITableView对象的rowHeight属性或编写tableView:heightForRow- AtIndexPath:方法才会修改UITableViewCell对象的高度。）

下面是各个子视图需要的约束：

1.UIImageView对象的尺寸始终为(40×40)像素，在contentView中垂直居中，而左边紧贴contentView。

2.nameLabel和serialNumberLabel保持与UIImageView对象的左边距相等，并且始终保持画布当前距离。同时，它们的宽度延伸到valueLabel左边，填充大部分屏幕，而高度保持当前值。

3.valueLabel在contentView中垂直居中，左边和右边分别与两个UILabel对象和contentView保持画布当前距离。

首先选中UIImageView对象，然后打开Pin菜单，固定UIImageView对象的高度和宽度(也可以选中UIImageView对象并按住Control键将其沿倾斜方向拖曳至自身)。

接下来让UIImageView对象在contentView中垂直居中。打开Align菜单，选择Vertical Center in Container(在父视图中垂直居中)。如果这里使用拖曳的方式，请注意不要将UIImageView对象拖曳到了另一个子视图。避免该问题的方法是，将UIImageView对象拖曳至大纲视图中的Content View(见图19-7)。

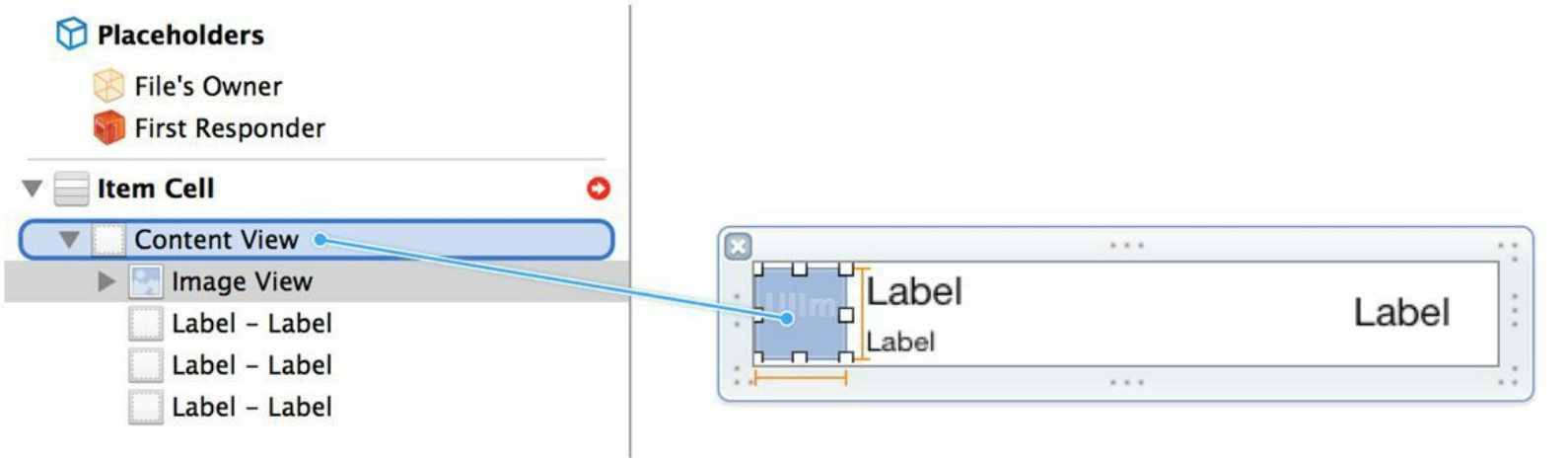


图19-7 拖曳至大纲视图

下面同时为所有子视图设置水平方向的约束。按住Shift键，同时选中4个子视图，再打开Pin菜单，在菜单顶部选择左边和右边，然后点击Add 6 Constraints添加约束。

现在UIImageView对象已经具备了所有需要的约束，如果在画布中选中UIImageView对象，可以看见UIImageView对象周围蓝色的约束直线。(如果UIImageView对象的约束直线不是蓝色的，也不用担心，本章稍后会介绍如何修复该问题。)完成后的UIImageView对象约束应该如图19-8所示。

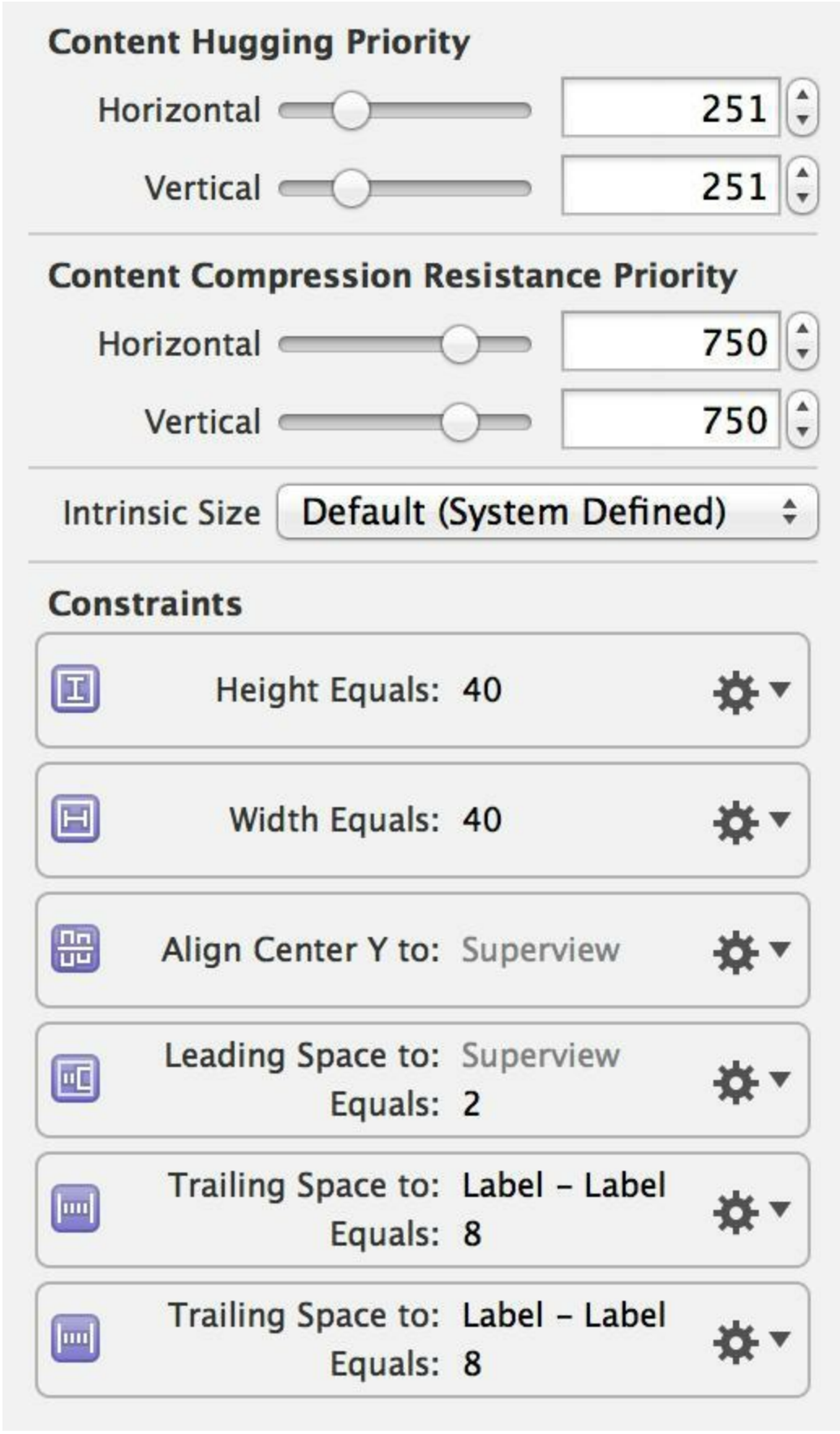


图19-8 UIImageView对象的约束

继续为nameLabel和serialNumberLabel添加约束。同时选中两个UILabel对象，然后打开Pin菜单，在菜单顶部选择顶边和底边，再勾选Height，最后点击Add 5 Constraints添加约束。可以发现，两个UILabel对象的约束直线也变为蓝色了。但是，如果现在改变UITableViewCell对象的高度，就会产生约束冲突。目前，垂直方向的所有约束关系(Relation)都是相等(Equal)。以下是

垂直方向约束的视觉化格式字符串：

V:|-1-[nameLabel(==21)]-5-[serialNumberLabel(==15)]-1-|

由此可知，目前垂直方向的总高度为1+21+5+15+1=43，与contentView的高度相同。如果改变UITableViewCell的高度，就无法同时满足垂直方向的所有约束（读者可以尝试在大小检视面板中修改UITableViewCell的高度）。为了解决该问题，需要将其中一个约束的关系修改为大于或等于（Greater Than or Equal）。

下面修改nameLabel底边与serialNumberLabel顶边距离的约束。该约束在画布上的蓝色直线很短，很难选中；相反，可以选中nameLabel，然后打开大小检视面板，面板中列出了nameLabel的所有约束。找到底边与serialNumberLabel顶边距离的约束，然后点击右侧的齿轮图标，在弹出的菜单中选择Select and Edit…（选中并编辑…）。这时Xcode会自动切换到该约束的属性检视面板，在顶部的Relation下拉菜单中，选择Greater Than or Equal。现在两个UILabel对象的约束应该类似于图19-9。

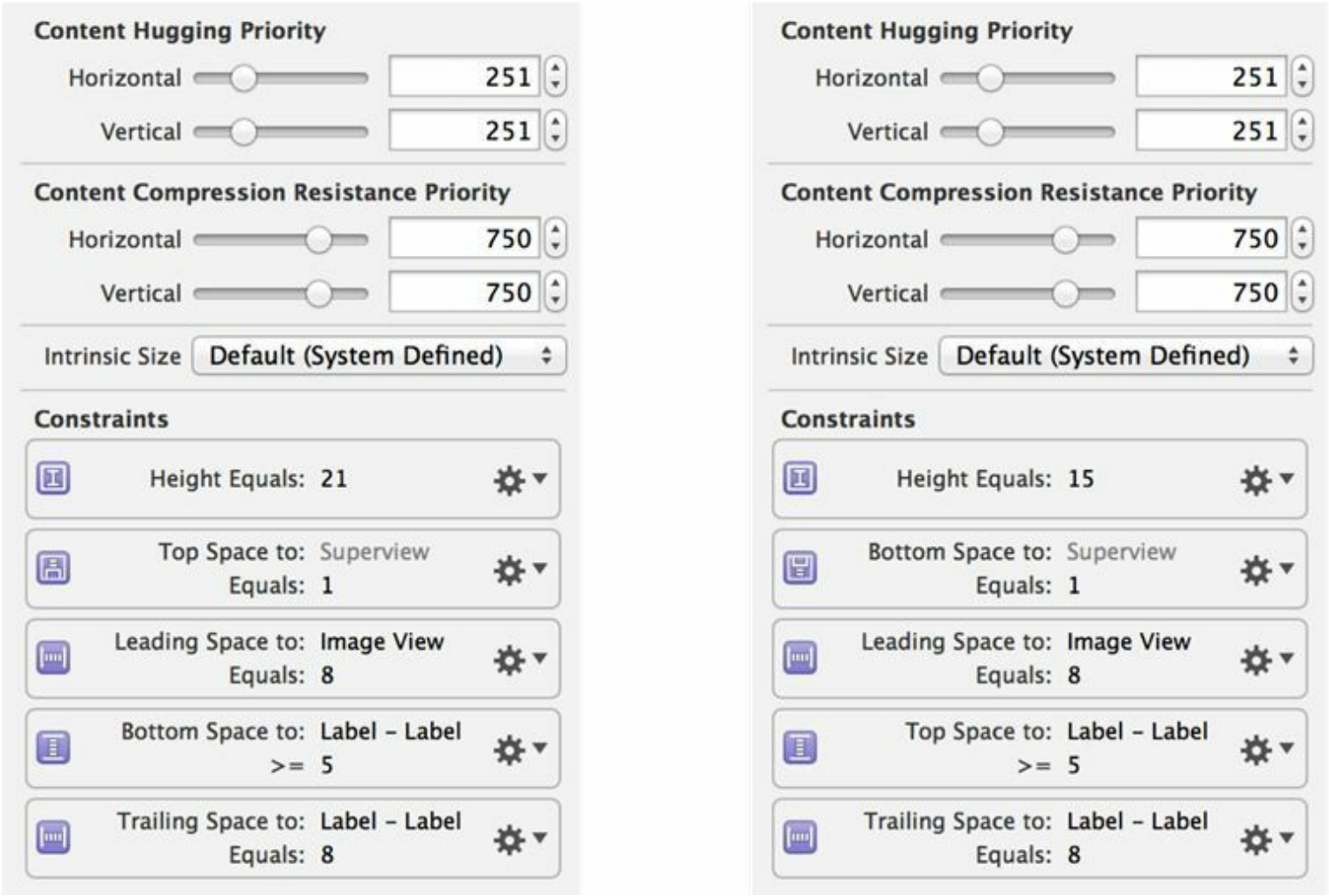


图19-9 nameLabel和serialNumberLabel的约束

接下来选中valueLabel，点击Align菜单，勾选Vertical Center in Container（在父视图中垂直居中），最后点击Add 1 Constraint添加约束。

现在还有一个问题：目前三个UILabel对象都没有限定宽度，因此自动布局系统会根据固有内容大小设置它们的宽度。但是，目前三个UILabel对象的内容放大优先级是相同的，如果三个

UILabel对象显示的文字都比较少(三个UILabel对象占据的屏幕宽度大于固有内容大小的宽度),自动布局系统就无法判断应该拉伸哪个UILabel对象以满足当前约束。

为了解决该问题,需要使valueLabel的内容放大优先级高于其他两个UILabel对象。读者可能会问:为什么不直接限定valueLabel的宽度呢?如果直接限定宽度,当valueLabel需要显示的文字比较多时,超出宽度的文字就会被截断而无法显示;相反,调整内容放大优先级后,valueLabel的宽度将根据需要显示的文字自动调整,以便显示所有文字(但是,如果文字过多,自动布局系统也会截断文字以满足当前约束)。

选中valueLabel,打开大小检视面板,将Horizontal Content Hugging Priority设置为1000。这时valueLabel的约束应该类似于图19-10。

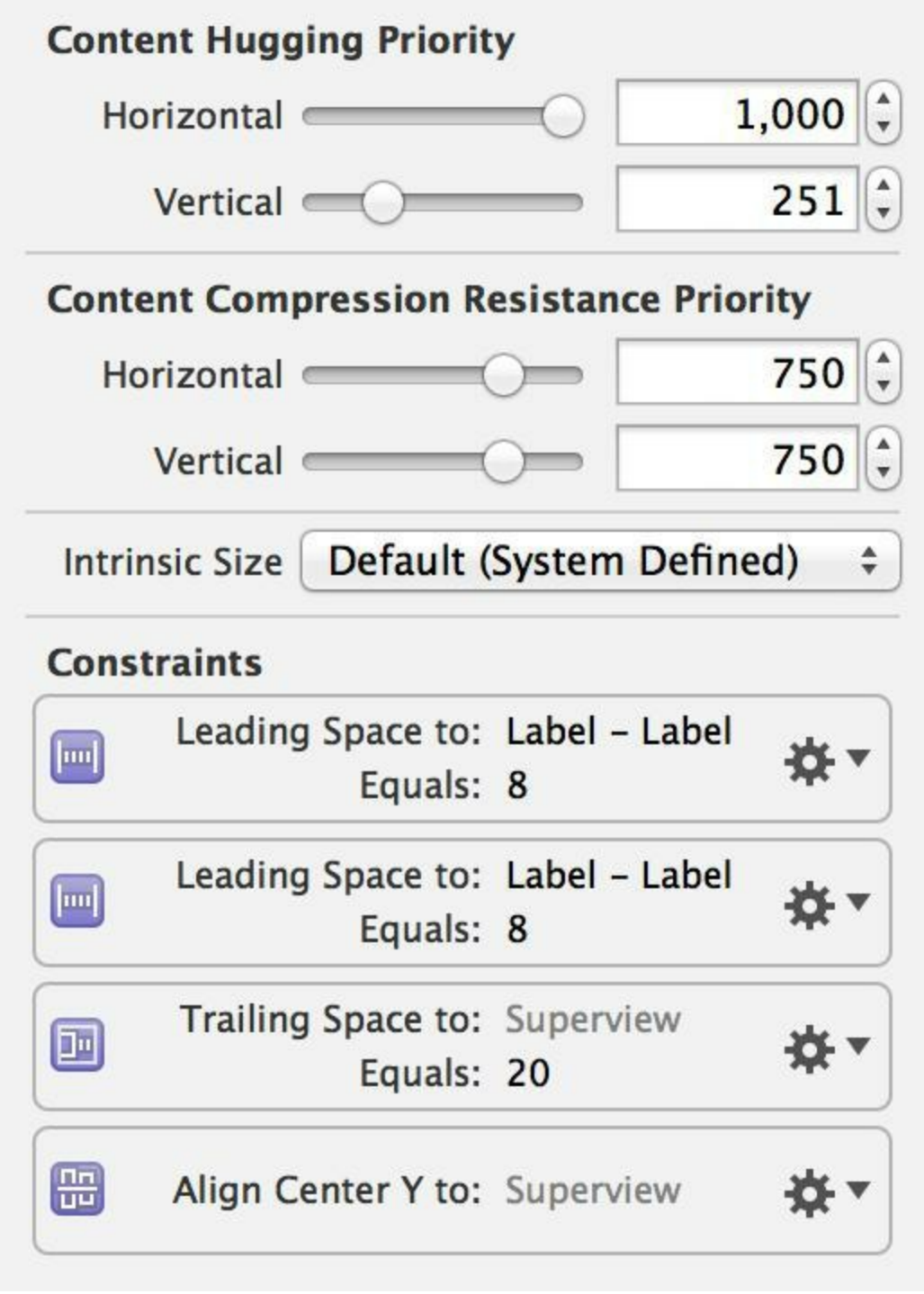


图19-10 valueLabel的约束

现在请读者检查画布中的约束直线，如果仍然有橘红色直线，可以打开Resolve Auto Layout Issues菜单，然后选择Update All Frames in Item Cell。

虽然添加约束的步骤较多，但是目前contentView的子视图会根据UITableViewCell对象的尺寸自动调整大小和布局，以适配不同的屏幕和方向。

19.2 处理图片

现在为BNRItemCell中的UIImageView对象设置图片。要在BNRItemCell中显示BNRItem对象的图片，可以先根据BNRItem对象从BNRImageStore中获取相应的图片，然后直接赋给UIImageView对象。但是因为图片的尺寸很大，所以这样做需要读取大量的数据，还要由UIImageView负责调整图片的大小，这些额外的工作都会影响应用的性能。另一种更合理的解决方案是创建并使用图片的缩略图（thumbnail）。

iOS SDK提供了多种创建缩略图的途径，其中之一是根据原图在屏外上下文（offscreen context）中画出按比率缩小后的版本，然后从上下文取出新创建的图片。下面通过这种途径为BNRItem对象的图片创建缩略图。首先要为BNRItem添加一个UIImage属性，指向缩略图；然后要将这些缩略图保存起来，以便在应用再次启动时重新载入。

第11章通过BNRImageStore对象存取全尺寸的图片，每张图片都有自己的独立文件；相反，缩略图的文件体积很小，可以直接将其和BNRItem对象的其他属性一起固化。

打开BNRItem.h，为BNRItem对象的缩略图声明一个新属性，此外还要声明一个新方法，根据全尺寸图片设置缩略图，代码如下：

```
@property (nonatomic, copy) NSString *imageKey;

@property (nonatomic, strong) UIImage *thumbnail;

- (void)setThumbnailFromImage:(UIImage *)image;

@end
```

当用户为某个BNRItem对象拍摄或选取了图片后，BNRItem对象会获得相应的全尺寸图片。然后，BNRItem对象要为全尺寸图片生成缩略图，并赋给thumbnail属性。

完成上述过程的方法是setThumbnailFromImage:。它根据传入的全尺寸图片，在屏外上下文中创建图片的小尺寸版本，并将其赋给thumbnail。

为了能够创建屏外上下文，并通过上下文创建图片，iOS特别提供了一套函数。UIGraphicsBeginImageContext函数可以创建屏外图形上下文（offscreen image context）。需要传入该函数的实参有：图形上下文的宽和高（CGSize结构）、缩放倍数（scaling factor）和图片是否透明（布尔值）。调用该函数后，新创建的CGContext结构将成为当前图形上下文。

要在上下文中绘图，需要使用Core Graphics。具体做法和实现UIView子类的drawRect:类似。调用UIGraphicsGetImageFromCurrentImageContext函数可以从上下文得到一个UIImage对象，即绘制的图片。

通过图形上下文得到UIImage对象后，必须调用UIGraphicsEndImageContext函数，清理相应的上下文。

在BNRItem.m中实现以下方法，通过屏外上下文创建缩略图。

```
- (void)setThumbnailFromImage:(UIImage *)image
{
CGSize origImageSize = image.size;
// 缩略图的大小
CGRect newRect = CGRectMake(0, 0, 40, 40);
// 确定缩放倍数并保持宽高比不变
float ratio = MAX(newRect.size.width / origImageSize.width,
newRect.size.height / origImageSize.height);
// 根据当前设备的屏幕scaling factor创建透明的位图上下文
UIGraphicsBeginImageContextWithOptions(newRect.size, NO, 0.0);
// 创建表示圆角矩形的UIBezierPath对象
UIBezierPath *path = [UIBezierPath bezierPathWithRoundedRect:newRect
cornerRadius:5.0];
// 根据UIBezierPath对象裁剪图形上下文
[path addClip];
// 让图片在缩略图绘制范围内居中
CGRect projectRect;
projectRect.size.width = ratio * origImageSize.width;
projectRect.size.height = ratio * origImageSize.height;
projectRect.origin.x = (newRect.size.width - projectRect.size.width) / 2.0;
projectRect.origin.y = (newRect.size.height - projectRect.size.height) / 2.0;
// 在上下文中绘制图片
[image drawInRect:projectRect];
// 通过图形上下文得到UIImage对象，并将其赋给thumbnail属性
```

```
UIImage *smallImage = UIGraphicsGetImageFromCurrentImageContext();

self.thumbnail = smallImage;

// 清理图形上下文

UIGraphicsEndImageContext();

}
```

修改BNRDetailViewController.m中的imagePickerController:didFinishPicking-MediaWithInfo:，加入以下代码，在获取全尺寸图片后创建缩略图。

```
- (void)imagePickerController:(UIImagePickerController *)picker

didFinishPickingMediaWithInfo:(NSDictionary *)info

{

UIImage *image = info [UIImagePickerControllerOriginalImage];

[self.item setThumbnailFromImage:image];
```

为BNRItem添加thumbnail属性后，就可以在BNRItemsViewController中使用该属性。更新BNRItemsViewController.m中的tableView:cellForRowAtIndexPath:，代码如下：

```
cell.valueLabel.text =

[NSString stringWithFormat:@"$%d", item.valueInDollars];

cell.thumbnailView.image = item.thumbnail;

return cell;

}
```

构建并运行应用。为某个BNRItem对象拍摄照片，当UITableView对象再次出现时，应该会显示该对象的缩略图、名称和价值（对已经存在的BNRItem对象需要重新拍摄照片）。

最后还要将缩略图固化到文件。在BNRItem.m的initWithCoder:中加入以下代码：

```
- (id)initWithCoder:(NSCoder *)aDecoder

{

self = [super init];

if (self) {
```

```
_itemName = [aDecoder decodeObjectForKey:@“itemName”];
_serialNumber = [aDecoder decodeObjectForKey:@“serialNumber”];
_dateCreated = [aDecoder decodeObjectForKey:@“dateCreated”];
_itemKey = [aDecoder decodeObjectForKey:@“itemKey”];
_thumbnail = [aDecoder decodeObjectForKey:@“thumbnail”];
_valueInDollars = [aDecoder decodeIntForKey:@“valueInDollars”]
}
return self;
}
-（void）encodeWithCoder:（NSCoder *）aCoder
{
[aCoder encodeObject:self.itemName forKey:@“itemName”];
[aCoder encodeObject:self.serialNumber forKey:@“serialNumber”];
[aCoder encodeObject:self.dateCreated forKey:@“dateCreated”];
[aCoder encodeObject:self.itemKey forKey:@“itemKey”];
[aCoder encodeObject:self.thumbnail forKey:@“thumbnail”];
[aCoder encodeInt:self.valueInDollars forKey:@“valueInDollars”];
}
```

构建并运行应用。为若干BNRItem对象拍摄照片，然后终止应用并重新启动，应该能看到之前拍摄的照片的缩略图。

19.3 由UITableViewCell对象转发动作消息

开发iOS应用时，可能需要将UIControl对象（或UIControl子类对象，例如UIButton对象）加入某个UITableViewCell对象。以BNRItemCell为例，假设要添加如下功能：点击某个BNRItemCell中的缩略图时，显示相应BNRItem的全尺寸的图片。为了完成上述功能，本节将在UIImageView对象上添加一个透明的UIButton对象。此外，如果在iPad中点击UIButton对象，Homepwner需要使用UIPopoverController显示全尺寸图片。

打开BNRItemCell.m，添加一个空的动作方法，用于显示全尺寸图片：

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

{

}
```

打开BNRItemCell.xib，拖曳一个UIButton对象至UIImageView对象，再删除UIButton对象的标题。同时选中UIImageView对象和UIButton对象，然后打开Align菜单，勾选Leading Edges、Trailing Edges、Top Edges和Bottom Edges四个选项。接下来在标题为Update Frames的下拉菜单中选择Items of New Constraints（匹配新约束），最后单击Add 4 Constraints添加约束（见图19-11）。

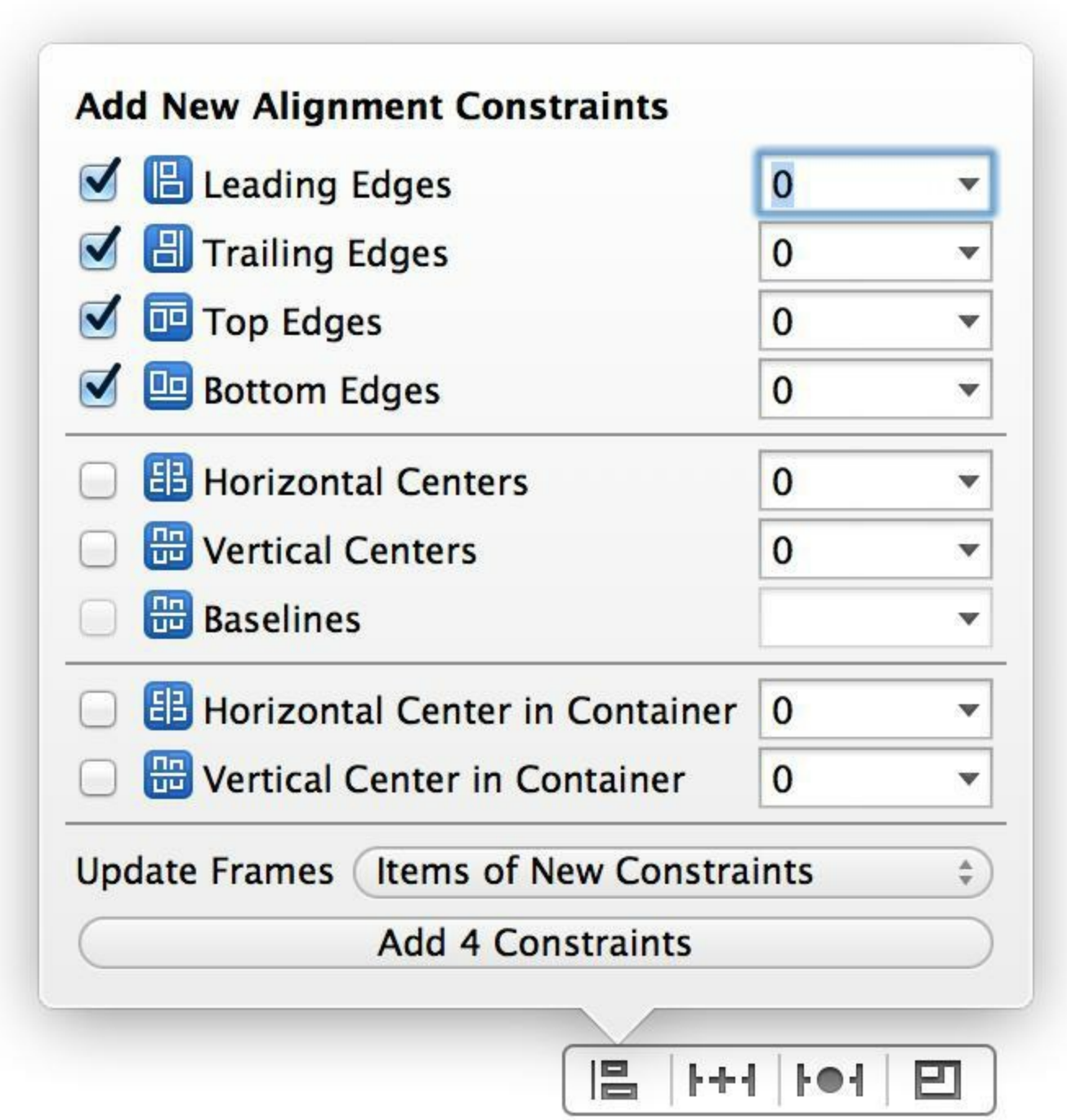


图19-11 UIButton对象的约束

最后需要将UIButton对象的动作方法设置为showImage:。在辅助编辑器中打开BNRItemCell.xib和BNRItemCell.m，然后按住Control，将UIButton对象拖曳至showImage:（见图19-12）。

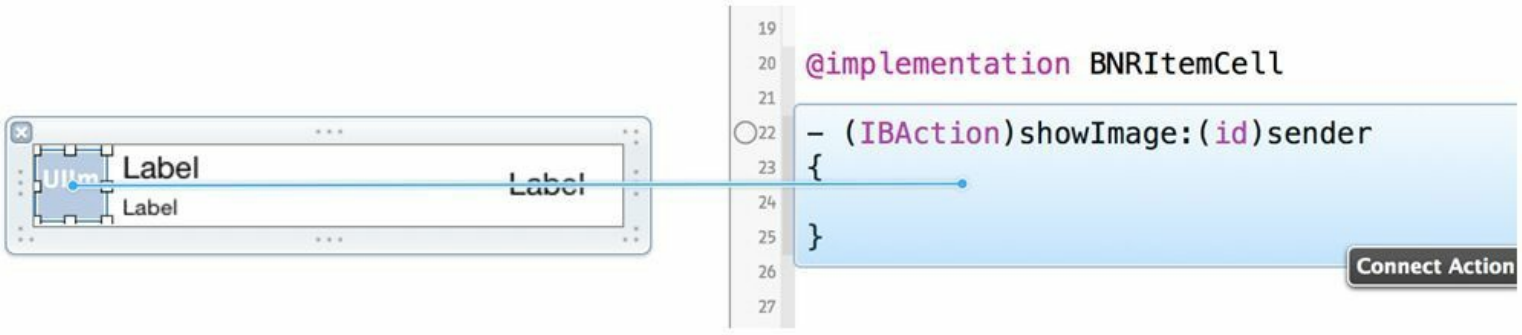


图19-12 设置UIButton对象的动作方法

现在，点击按钮就可以将showImage:消息发送给BNRItemCell对象。下面需要实现showImage:——这里有一个问题：UIButton对象会将动作消息发送给相应的BNRItemCell对象，但是BNRItemCell对象不是控制器，无法访问全尺寸图片，甚至无法访问其当前显示的BNRItem对象。

虽然可以为BNRItemCell添加一个属性，指向其当前显示的BNRItem对象，但是BNRItemCell对象是视图对象，不应该直接访问模型对象，也不应该负责显示视图控制器（例如UIPopoverController）。

更好的解决方案是：通过BNRItemsViewController为BNRItemCell添加一个Block对象，当用户点击UIButton对象时，在Block对象中显示全尺寸图片。

为BNRItemCell添加Block对象

本书第17章曾简要介绍过Block对象，本章将深入学习Block对象。

打开BNRItemCell.h，添加一个Block属性，代码如下：

```
@interface BNRItemCell : UITableViewCell

@property (nonatomic, weak) IBOutlet UIImageView *thumbnailView;

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

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

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

@property (nonatomic, copy) void (^actionBlock)(void);

@end
```

读者可能会觉得语法有些奇怪，Block对象类似函数，有名称、参数和返回值，图19-13列出了Block对象的语法格式。

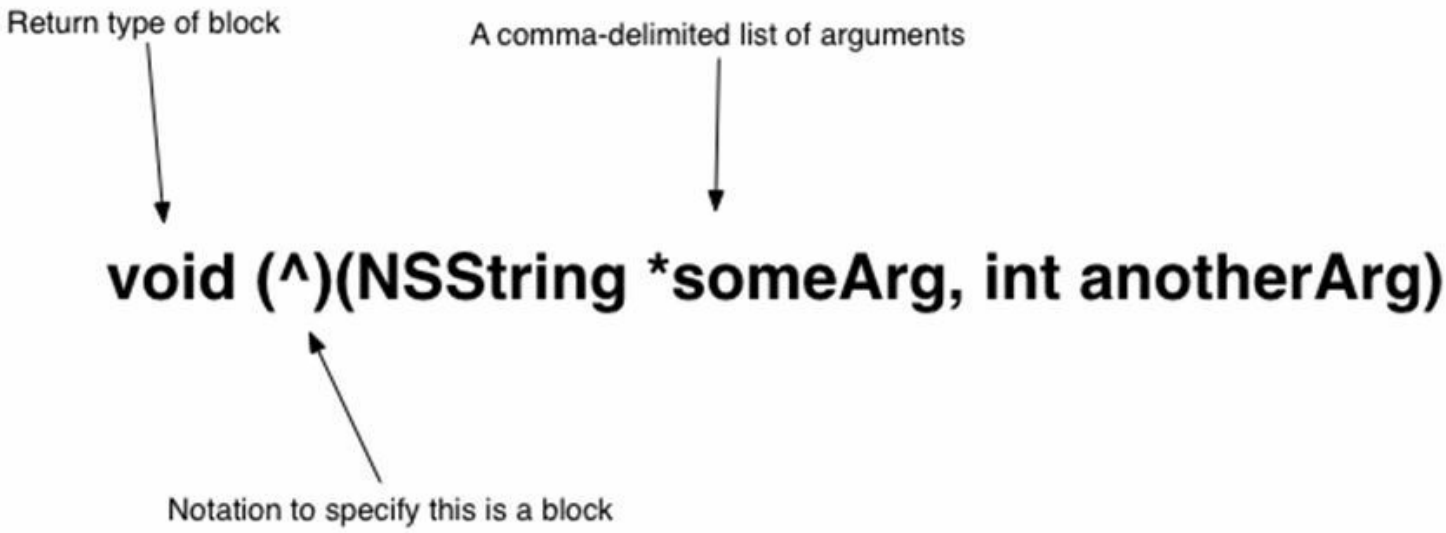


图19-13 Block对象的语法

请注意，actionBlock被声明为copy。系统对Block对象和其他对象的内存管理方式不同，Block对象是在栈中创建的，而其他对象是在堆中创建的。这意味着，即使应用针对新创建的Block对象保留了强引用类型的指针，一旦创建该对象的方法返回，那么与方法内部的其他局部变量相同，新创建的Block对象也会被立即释放。为了在声明Block对象的方法返回后仍然保留该对象，必须向其发送copy消息。拷贝某个Block对象时，应用会在堆中创建该对象的备份。这样，即使应用释放了当前方法的栈，堆中的Block对象也不会被释放。

在BNRItemCell.m的showImage:中调用Block对象，代码如下：

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

{

// 调用Block对象之前要检查Block对象是否存在

if (self.actionBlock) {

self.actionBlock();

}

}
```

下面更新BNRItemsViewController.m中的tableView: cellForRowAtIndex- Path:，向控制台输出传入的NSIndexPath对象，测试是否可以正确执行Block对象：

```
- (UITableViewCell *)tableView:(UITableView *)tableView

cellForRowAtIndexPath:(NSIndexPath *)indexPath

{
```

```
BNRItem *item = [[BNRItemStore sharedStore] allItems][indexPath.row];

// 获取BNRItemCell对象，返回的可能是现有的对象，也可能是新创建的对象

BNRItemCell *cell =

[tableView dequeueReusableCellWithIdentifier:@“BNRItemCell”

forIndexPath:indexPath];

// 根据BNRItem对象设置BNRItemCell对象

cell.nameLabel.text = item.itemName;

cell.serialNumberLabel.text = item.serialNumber;

cell.valueLabel.text =

[NSString stringWithFormat:@“$%i”, item.valueInDollars];

cell.thumbnailView.image = item.thumbnail;

cell.actionBlock = ^{

NSLog(@“Going to show image for %@”, item);

};

return cell;

}
```

构建并运行应用。点击某个缩略图（准确地说，是位于UIImageView对象上的透明UIButton对象），应该能在控制台看到相应的输出信息。

通过UIPopoverController显示图片

下面在BNRItemsViewController中修改BNRItemCell的actionBlock，根据UIButton对象所在的BNRItemCell对象，获取相应的BNRItem对象，然后在UIPopoverController中显示该BNRItem对象的图片。

要在UIPopoverController中显示图片，需要先准备好一个能够显示图片的UIViewController对象。使用Objective-C class文件模板创建一个名为BNRImage- ViewController的UIViewController子类，请注意不要勾选Also create XIB file。

BNRImageViewController只有一个视图，下面将通过代码创建视图。在BNRImageViewController.m中实现loadView，代码如下：

```
- (void)loadView

{

UIImageView *imageView = [[UIImageView alloc] init];

imageView.contentMode = UIViewContentModeScaleAspectFit;

self.view = imageView;

}
```

这里不需要为UIImageView对象添加任何约束。BNRImageViewController显示在UIPopoverController中，UIPopoverController会自动将UIImageView对象（BNRImageViewController的view）的大小调整为与自身一致。

下面在BNRImageViewController.h中添加一个属性，用来保存需要显示的UIImage对象：

```
@interface BNRImageViewController : UIViewController

@property (nonatomic, strong) UIImage *image;

@end
```

创建BNRImageViewController对象后，要将相应的UIImage对象赋给image属性。在BNRImageViewController.m中实现viewWillAppear:，将image显示到UIImageView中：

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

{

[super viewWillAppear:animated];

// 必须将view转换为UIImageView对象，以便向其发送setImage:消息

UIImageView *imageView = (UIImageView *)self.view;

imageView.image = self.image;

}
```

现在可以完成actionBlock了。在BNRItemsViewController.m中，首先添加一个属性，用于保存UIPopoverController，然后将BNRItemsViewController声明为遵守UIPopoverControllerDelegate协议，代码如下：

```
@interface BNRItemsViewController () <UIPopoverControllerDelegate>
@property (nonatomic, strong) UIPopoverController *imagePopover;
@end
```

接下来在BNRItemsViewController.m顶部导入需要的头文件：

```
#import "BNRImageStore.h"
#import "BNRImageViewController.h"
```

最后在actionBlock中显示UIPopoverController：

```
cell.actionBlock = ^{
NSLog(@"Going to show the image for %@", item);
if ([UIDevice currentDevice] userInterfaceIdiom == UIUserInterfaceIdiomPad) {
NSString *itemKey = item.itemKey;
// 如果BNRItem对象没有图片，就直接返回
UIImage *img = [[BNRImageStore sharedStore] imageForKey:imageKey];
if (!img)
return;
}
// 根据UITableView对象的坐标系获取UIImageView对象的位置和大小
// 注意：这里也许会出现警告信息，下面很快就会讨论到
CGRect rect = [self.viewconvertRect:cell.thumbnailView.bounds
fromView:cell.thumbnailView];
// 创建BNRImageViewController对象并为image属性赋值
BNRImageViewController *ivc = [[BNRImageViewController alloc] init];
ivc.image = img;
// 根据UIImageView对象的位置和大小
```

```
// 显示一个大小为600x600点的UIPopoverController对象

self.imagePopover = [[UIPopoverController alloc]

initWithContentViewController:ivc];

self.imagePopover.delegate = self;

self.imagePopover.PopoverContentSize = CGSizeMake(600, 600)];

[self.imagePopover presentPopoverFromRect:rect

inView:self.view

permittedArrowDirections:UIPopoverArrowDirectionAny

animated:YES];

}

};
```

最后，在BNRItemsViewController.m中实现popoverControllerDidDismissPopover:，当用户关闭UIPopoverController时，将UIPopoverController设置为nil：

```
-(void)popoverControllerDidDismissPopover:

(UIPopoverController *)popoverController

{

self.imagePopover = nil;

}
```

在iPad模拟器中构建并运行应用。点击某个BNRItemCell对象中的缩略图，Homepwner应该会弹出一个UIPopoverController对象，并显示相应BNRItem对象的全尺寸图片。点击其他区域可以关闭UIPopoverController对象。

19.4 捕获变量

Block对象可以使用其封闭作用域(enclosing scope)内的所有变量。对声明了某个Block对象的方法，该方法的作用域就是这个Block对象的封闭作用域。因此，这个Block对象可以访问该方法的所有局部变量、传入该方法的实参以及所属对象的实例变量。在上述代码中，BNRItem对象(item)和BNRItemCell对象(cell)都是actionBlock封闭作用域中的捕获变量。

如果捕获变量是Objective-C对象，那么Block对象对捕获变量具有强引用。如果捕获变量也对Block对象具有强引用，就会导致强引用循环。之前在actionBlock中创建rect时，Xcode会提示警告信息："Capturing 'cell' strongly in this block is likely to lead to a strong reference cycle(在Block对象中捕获'cell'很可能导致强引用循环)"——BNRItemCell对actionBlock具有强引用，actionBlock对BNRItemCell也具有强引用，如图19-14所示。

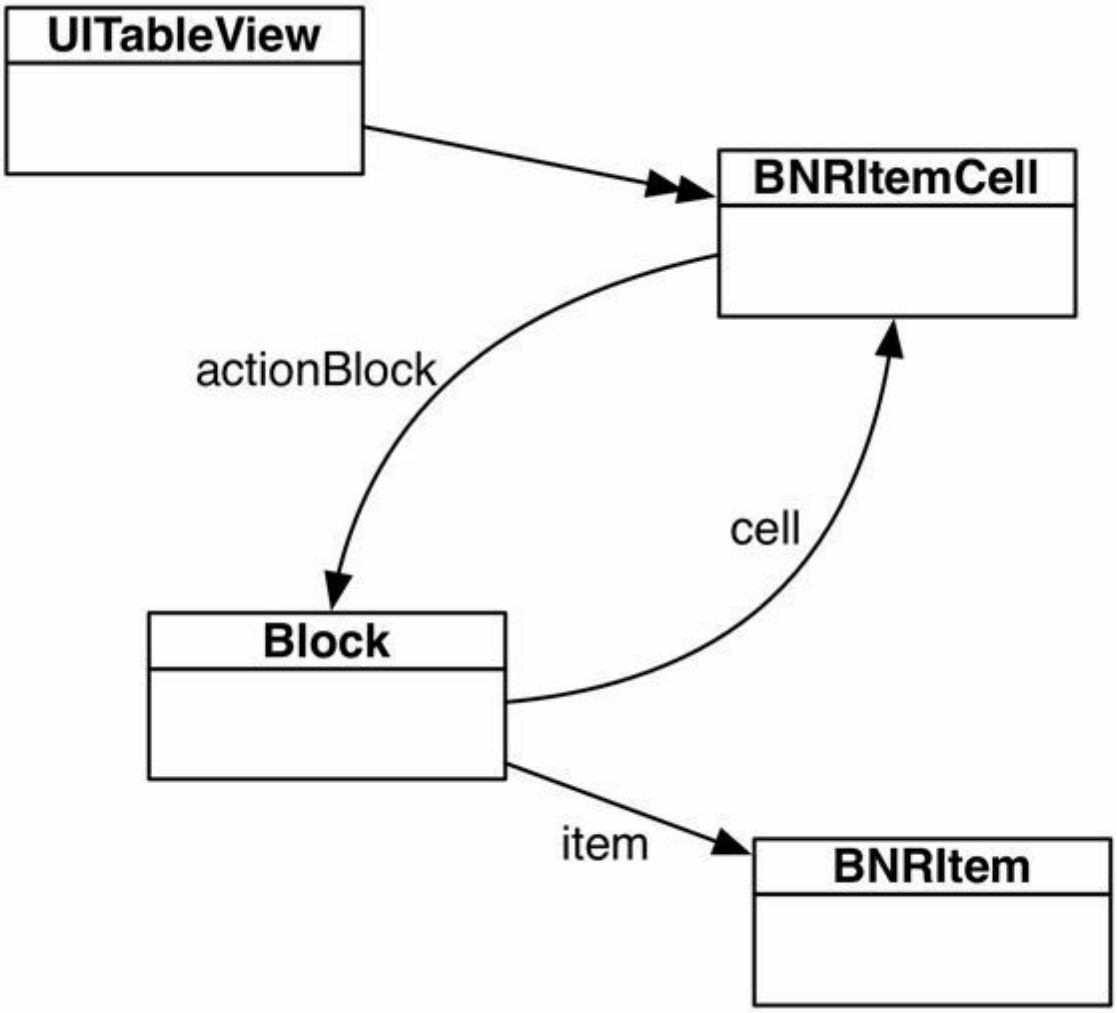


图19-14 cell和Block对象之间存在强引用循环

解决问题的方法是：将actionBlock对cell的引用改为弱引用。

在BNRItemsViewController.m中修改actionBlock中的代码，使其对cell具有弱引用，代码如下：

```
__weak *weakCell = cell;
```

```
cell.actionBlock = ^{
NSLog(@"Going to show the image for %@", item);
BNRItemCell *strongCell = weakCell;
if ([UIDevice currentDevice] userInterfaceIdiom == UIUserInterfaceIdiomPad) {
NSString *itemKey = item.itemKey;
// 如果BNRItem对象没有图片，就直接返回
UIImage *img = [[BNRImageStore sharedStore] imageForKey:imageKey];
if (!img)
return;
}
// 根据UITableView对象的坐标系获取UIImageView对象的位置和大小
// 注意：这里也许会出现警告信息，下面很快就会讨论到
CGRect rect = [self.viewconvertRect:cell.thumbnailView.bounds
fromView:cell.thumbnailView];
CGRect rect = [self.viewconvertRect:strongCell.thumbnailView.bounds
fromView:strongCell.thumbnailView];
// 创建BNRImageViewController对象并为image属性赋值
BNRImageViewController *ivc = [[BNRImageViewController alloc] init];
ivc.image = img;
// 根据UIImageView对象的位置和大小
// 显示一个大小为600x600点的UIPopoverController对象
self.imagePopover = [[UIPopoverController alloc]
initWithContentViewController:ivc];
self.imagePopover.delegate = self;
```

```
    self.imagePopover.PopoverContentSize = CGSizeMake(600, 600)];

    [self.imagePopover presentPopoverFromRect:rect

    inView:self.view

    permittedArrowDirections:UIPopoverArrowDirectionAny

    animated:YES];

    }

    };
```

在Block对象执行过程中，必须保证Block对象始终可以访问cell。因此，以上代码在actionBlock内部创建了strongCell，以保持对cell的强引用。这与Block对象对捕获变量的强引用不同，strongCell只是在Block对象执行过程中对cell保持强引用。

构建并运行应用，运行结果与之前相同，但是现在应用不会由于强引用循环导致内存泄露。

a

19.5 初级练习：设置颜色

如果BNRItem对象的价值大于50美元，就将valueLabel的文字颜色设置为绿色；反之，如果小于50美元，就设置为红色。

19.6 高级练习：缩放

BNRImageViewController对象应该能够居中显示图片并支持缩放功能。在BNRImageViewController.m中实现上述两项功能。

19.7 深入练习：UICollectionView

UICollectionView与UITableView非常相似：

•UICollectionView是UIScrollView的子类。

•与UITableViewCell类似，UICollectionView对象显示一组UICollection- ViewCell或其子类。

•UICollectionView具有数据源，负责提供UICollectionViewCell。

•UICollectionView具有委托，可以在委托方法中处理相关回调事件，例如选择了某一个UICollectionViewCell。

•UICollectionViewController与UITableViewController类似，UICollection- ViewController也是UIViewController的子类，其view是UICollectionView。

UICollectionView与UITableView的区别是，UITableView只能显示一列UITableViewCell，在大屏幕设备（如iPad）中有很大的局限性。UICollectionView则可以将UICollectionViewCell按任意方式布局，其中最常见的是网格布局（见图19-15）。

Homepwner

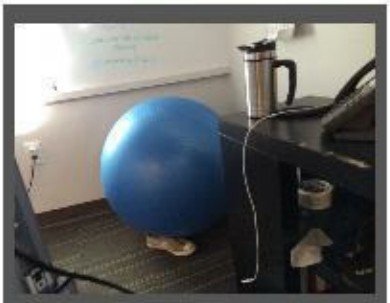
Mouse

Matt

Headphones

?

Helium



Exercise ball

Stool

Office Chair

图19-15 使用UICollectionView的Homepwner

UICollectionView是如何布局UICollectionViewCell的？UICollectionView含有一个布局对象，负责控制每一个UICollectionViewCell的属性，包括位置和大小。UICollectionView的布局对象继承自一个名为UICollectionViewLayout的抽象类。如果需要将UICollectionViewCell按网格布局，则可以使用系统提供的UICollectionView- FlowLayout。但是，如果需要实现其他的布局方式，就必须创建UICollectionViewLayout的自定义子类。

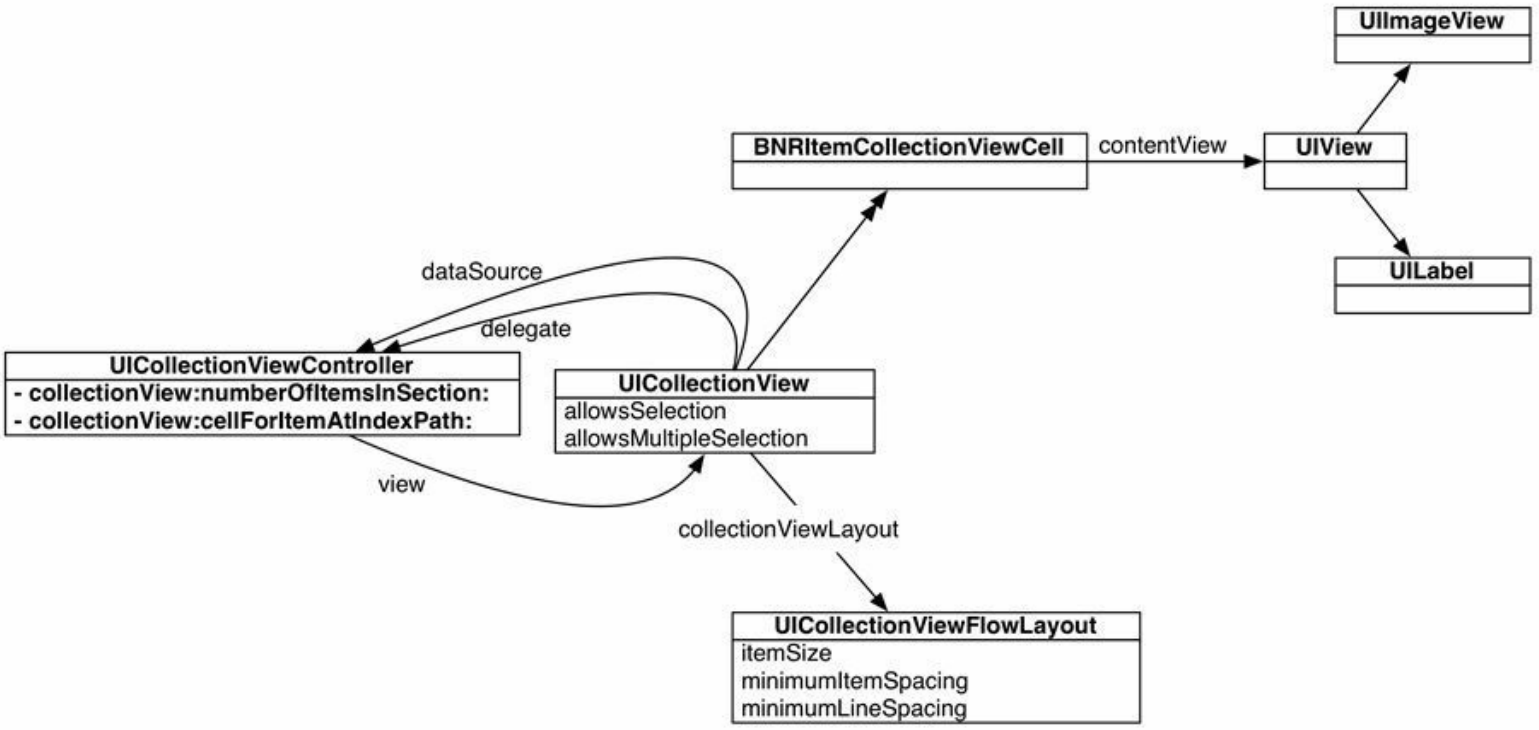


图19-16 UICollectionView对象图示例

本书之前章节并没有子类化UITableViewCell，而是直接使用UITableViewCell。相反，UICollectionViewCell通常并不能满足需求——UICollectionViewCell也有contentView，但是与UITableViewCell不同，UICollectionViewCell的contentView在默认情况下没有任何子视图。因此，如果需要使用UICollectionView，通常还需要创建一个UICollectionViewCell子类。

了解了以上的内容后，读者就可以尝试使用UICollectionView了。最后，UICollectionViewCell也具有背景视图和选中状态下的背景视图（当UICollectionViewCell处于选中状态时，该视图会覆盖在背景视图上方）。

第20章 动态字体

在制作应用的界面时，想同时满足所有用户的需求并不容易。有些用户喜欢紧凑的界面，以便同时看到更多的信息；另外一些用户喜欢宽松的界面，以便清楚地看到每一项信息；还有一些用户视力不好，需要将界面放大至能看清的程度。因此，应用应该针对不同的用户需求提供不同的显示效果。

动态字体是iOS 7引入的一项新技术，提供了7种不同大小的字体。用户可以在系统的设置(Settings)应用中选择自己喜欢的字体——支持动态字体的应用会根据用户选择的字体(称为用户首选字体)调整应用中显示的文字大小。本章将升级Homepwner，支持动态字体。图20-1是Homepwner最大字体和最小字体的显示效果对比。

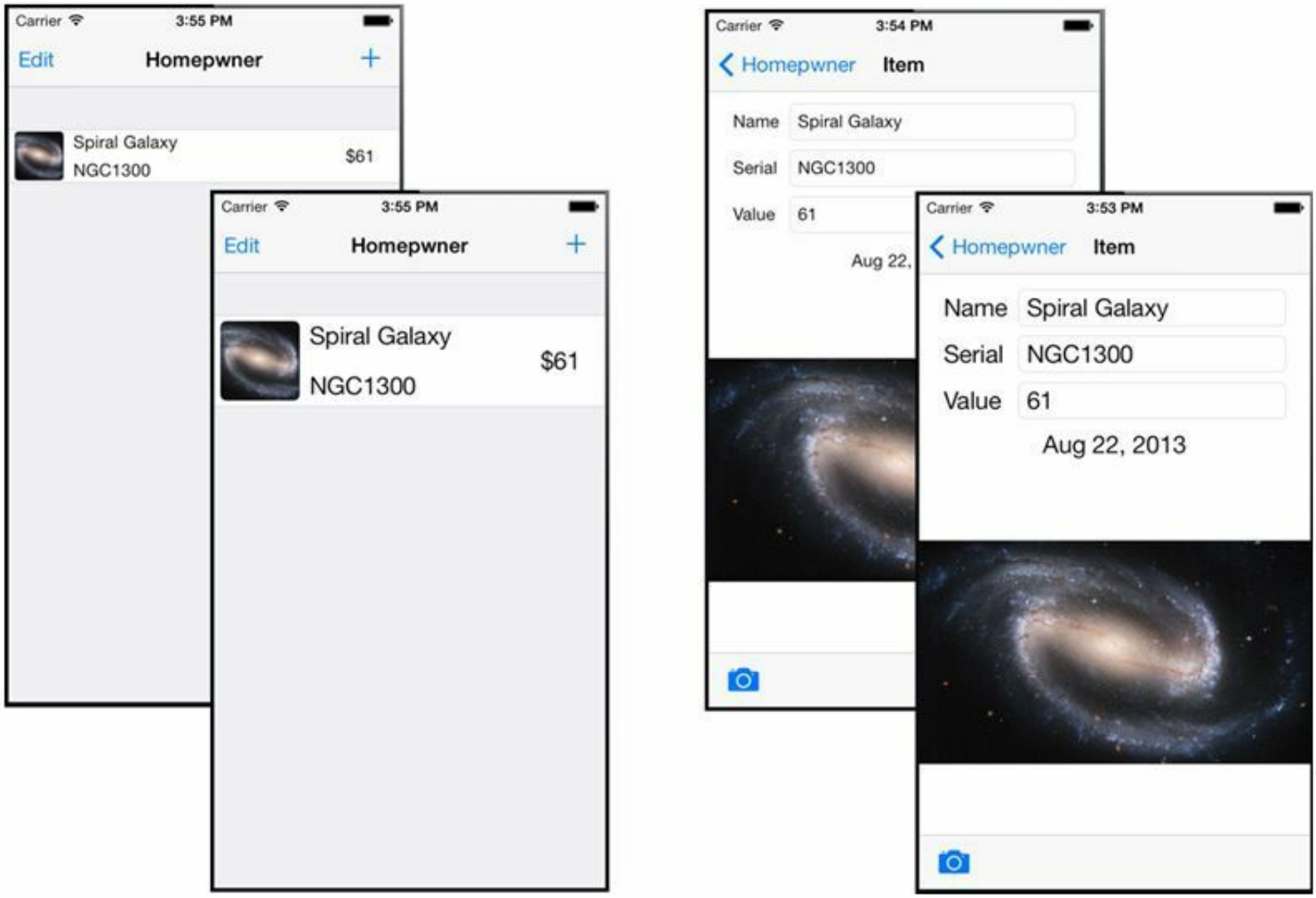


图20-1 支持动态字体的Homepwner

动态字体的核心概念是文本样式(text style)。当应用需要使用某种文本样式时，系统会根据用户首选字体决定该样式所对应的实际字体大小。图20-2显示了6种不同的文本样式。

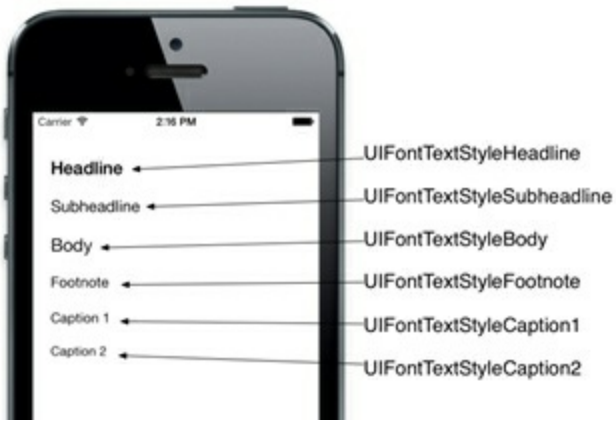


图20-2 6种不同的文本样式

20.1 使用用户首选字体

动态字体的使用方法非常简单，首先根据文本样式创建一个UIFont对象，然后将其设置为文本控件的字体就可以了（例如UILabel和UITextField的font属性）。下面就更新BNRDetailViewController，在代码中使用动态字体。

由于本章会设置界面中所有UILabel对象和UITextField对象的font属性，而其中三个UILabel对象没有对应的插座变量，因此需要在BNRDetailViewController.m的类扩展中为三个UILabel对象设置插座变量，代码如下：

```
@interface BNRDetailViewController ()

@property (nonatomic, strong) UIPopoverController *imagePickerPopover;

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

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

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

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

@property (weak, nonatomic) IBOutlet UIImageView *imageView;

@property (weak, nonatomic) IBOutlet UIToolbar *toolbar;

@property (weak, nonatomic) IBOutlet UIBarButtonItem *cameraButton;

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

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

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

@end
```

然后添加一个方法，为正文（UIFontTextStyleBody）的UIFont对象创建文本样式，再赋给所有UILabel对象和UITextField对象的font属性，代码如下：

```
- (void)updateFonts

{

UIFont *font = [UIFont preferredFontForTextStyle:UIFontTextStyleBody];

self.nameLabel.font = font;
```

```
    self.serialNumberLabel.font = font;

    self.valueLabel.font = font;

    self.dateLabel.font = font;

    self.nameField.font = font;

    self.serialNumberField.font = font;

    self.valueField.font = font;

    }
```

接下来在viewWillAppear:中调用updateFonts方法，当UILabel对象将要出现在屏幕上时，修改它们的字体：

```
    self.imageView.image = imageToDisplay;

    [self updateFonts];

    }
```

preferredFontForTextStyle:方法会根据用户首选字体和传入的文本样式返回对应的UIFont对象。构建并运行应用，这时界面还不会发生变化。

下面修改用户首选字体。按下iOS设备的Home键（如果是iOS模拟器，选择硬件→首页），然后打开设置应用，选择通用（General）→文字大小（Text Size），将滑块拖曳到最左端，设置用户首选字体为最小值（见图20-3）。

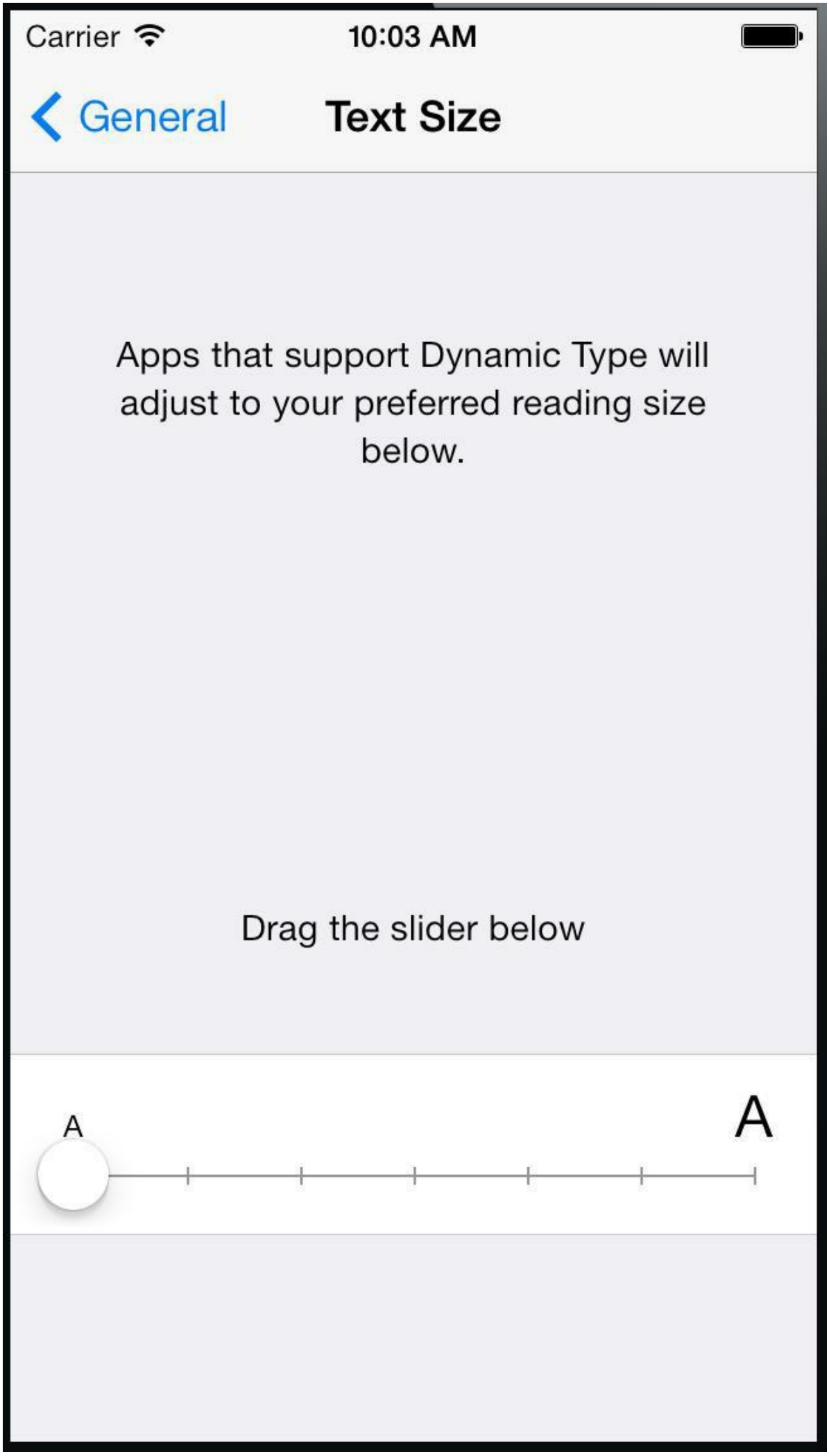

Carrier
10:03 AM
General
Text Size
Apps that support Dynamic Type will adjust to your preferred reading size below.
Drag the slider below
A
A

图20-3 修改用户首选字体大小

现在回到Homepwner的BNRDetailViewController界面，可以发现，界面中的文本控件并没有根据用户首选字体调整文字大小。这是因为，当应用从后台返回到前台运行时，BNRDetailViewController没有再次收到viewWillAppear:消息，所以没有及时更新文本控件的字体。为了解决该问题，必须设法观察并响应用户首选字体的改变。

20.2 响应用户首选字体的改变

用户修改了首选字体大小后，应用会收到UIContentSizeCategoryDidChangeNotification通知，可以注册该通知响应用户首选字体的改变，例如更新界面中文本控件的字体。

打开BNRDetailViewController.m，在initForNewItem:中将BNRDetailViewController对象注册为UIContentSizeCategoryDidChangeNotification的观察者，然后在dealloc中移除观察者，代码如下：

```
    self.navigationItem.leftBarButtonItem = cancelItem;

    }

    // 注意不要将注册通知的代码写在if (isNew) {}中

    NSNotificationCenter *defaultCenter =

    [NSNotificationCenter defaultCenter];

    [defaultCenter addObserver:self

    selector:@selector(updateFonts)

    name:UIContentSizeCategoryDidChangeNotification

    object:nil];

    }

    return self;

    }

    - (void)dealloc

    {

    NSNotificationCenter *defaultCenter =

    [NSNotificationCenter defaultCenter];

    [defaultCenter removeObserver:self];

    }
```

注意，用于响应UIContentSizeCategoryDidChangeNotification的方法是之前实现的updateFonts，viewWillAppear:中也会调用该方法。再次构建并运行应用，首先在设置应用中修

改首选字体大小，然后返回应用，这次所有文本控件都会根据用户首选字体调整文字大小。

因为现在文本控件的文字大小会动态变化，所以下一节将调整文本控件的约束，支持动态字体。

20.3 修改自动布局约束

第15章限定了BNRDetailViewController界面中四个UILabel对象的宽度和高度，如果应用支持动态字体，自动布局系统将无法修改它们的宽度和高度。如果用户选择了一个较小的字体，UILabel对象可能会留下大片空白区域；相反，如果选择了一个较大的字体，UILabel对象可能无法显示全部文字。为了解决该问题，需要让自动布局系统根据UILabel对象的intrinsicContentSize属性动态设置UILabel对象的frame。

打开BNRDetailViewController.xib，在画布中依次选中四个UILabel对象并删除它们的宽度和高度约束。这时Interface Builder会提示视图位置错误，打开Resolve Auto Layout Issues菜单，选择Update All Frames in Control。

现在请读者修改除底部的dateLabel外的任意一个UILabel对象的文字（可以多输入一些文字，超出UILabel对象的当前宽度）。这时读者会注意到，三个UITextField对象将无法对齐。为了解决UITextField对象的布局问题，首先需要理解自动布局系统是如何设置视图frame的。

复习内容放大优先级与内容缩小优先级

第16章介绍过，所有视图都具有intrinsicContentSize属性，表示视图的固有内容大小，自动布局系统会根据固有内容大小自动为视图添加宽度和高度约束。如果需要让自动布局系统在必要时基于固有内容大小放大视图尺寸，则可以为视图添加一个优先级比视图的内容放大优先级（Content Hugging Priority）高的约束；相反，如果需要让自动布局系统在必要时基于固有内容大小缩小视图尺寸，则可以为视图添加一个优先级比视图的内容缩小优先级（Content Compression Resistance Priority）高的约束。

首先检查UILabel对象和UITextField对象在水平方向上的约束，如果使用视觉化格式语言描述该约束，以nameLabel和nameField为例，视觉化格式字符串如下所示：

H:|-8-[nameLabel]-8-[nameField]-8-|

从视觉化格式字符串中可以看出，目前并没有为nameLabel和nameField添加与宽度相关的约束，因此自动布局系统会在必要时调整nameLabel和nameField的宽度：对于内容放大优先级较高的视图，自动布局系统会根据固有内容大小设置它的宽度；对于内容放大优先级较低的视图，自动布局系统则会拉伸它的宽度，以满足当前约束。如果比较UILabel对象和UITextField对象，会发现UILabel对象的内容放大优先级是251，而UITextField对象的则是250。由于UILabel对象的内容放大优先级更高，因此UILabel对象会保持固有内容大小的宽度，而UITextField对象则会根据当前约束拉伸宽度。

为了对齐所有UITextField对象，可以使位于UITextField对象左侧的三个UILabel对象保持宽度相同。读者可能认为之前删除的宽度约束可以使三个UILabel对象保持宽度相同了，但实际上之前删除的宽度约束与现在要添加的约束有本质区别：之前的宽度约束是分别添加在各个UILabel对象上的，三个约束之间互相独立，没有任何关系，仅仅是限定的数值相同；而现在要同时为三个UILabel对象添加约束，无论其中哪个UILabel对象的宽度发生变化，其他两个

UILabel对象也都会随之发生变化，始终保持宽度相等。

同时选中位于UITextField对象左侧的三个UILabel对象，然后打开Pin菜单，选择Equal Widths（宽度相等），并在Update Frames下拉菜单中选择All Frames in Container，最后点击Add 2 Constraints添加约束。这时界面应该类似于图20-4。

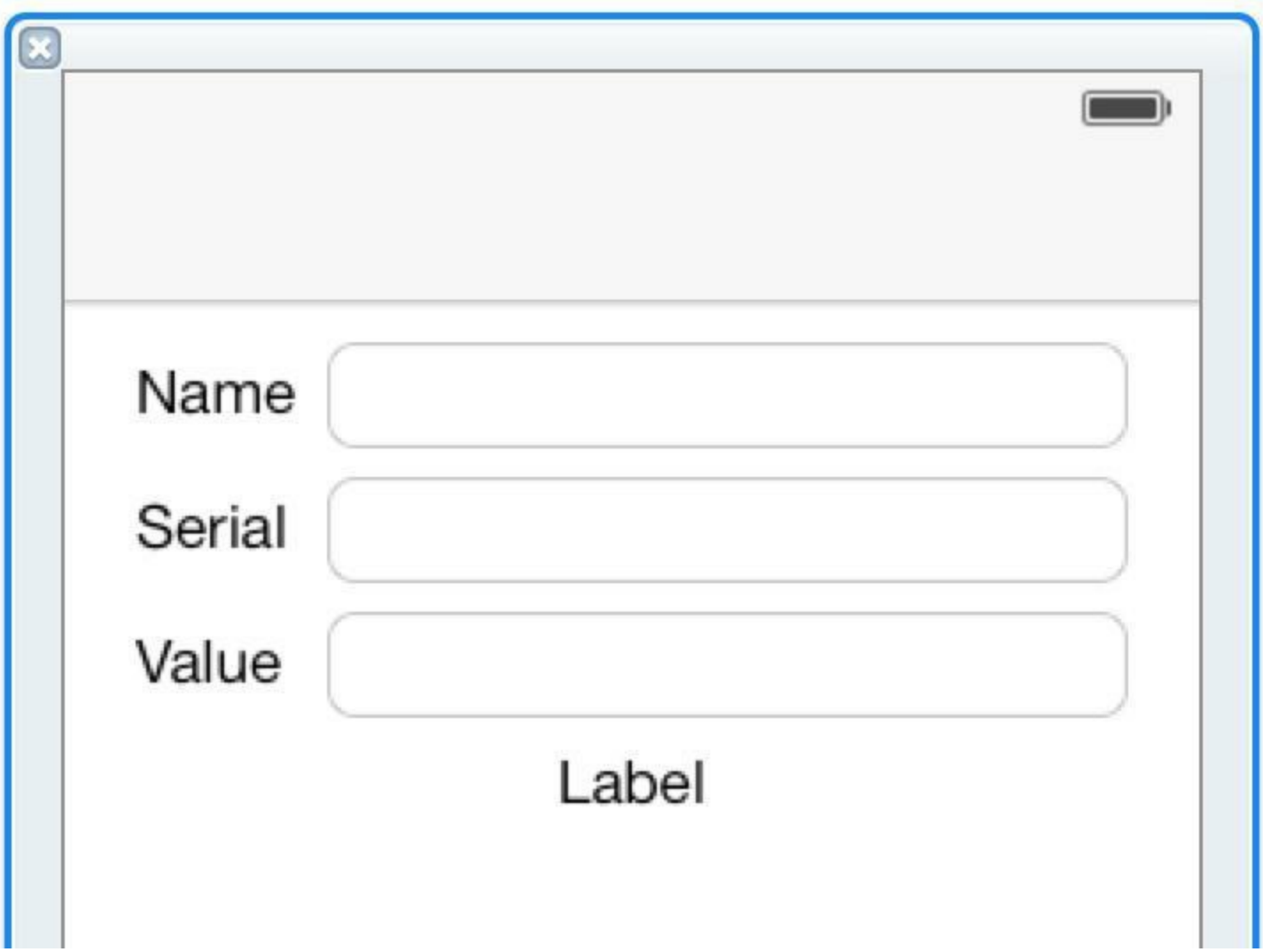


图20-4 为三个UILabel对象添加宽度相等约束

虽然步骤很简单，但是读者可能会对背后的原理产生疑惑。下面就具体解释以上步骤的原理。首先，UILabel对象的内容放大优先级是251，高于UITextField对象的250，因此自动布局系统会根据固有内容大小设置UILabel对象的宽度；其次，因为刚才为三个UILabel对象添加了宽度相等约束，所以目前三个UILabel对象各具有两个与宽度相关的约束：宽度相等约束（优先级为1000，属于必需约束）与自动布局系统根据固有内容大小添加宽度约束。

再看约束对视图布局的影响。为了满足当前约束，现在只有文字内容最长的UILabel对象会保持固有内容大小，而另外两个UILabel对象则会拉伸宽度，与最长的UILabel对象保持相等，这是由于宽度相等约束的优先级（1000）高于内容放大优先级（251）。

理解上述原理非常重要。目前UILabel对象和UITextField对象可以根据显示的文字内容自动调整布局，而文字内容在很多情况下都会发生变化，最常见的就是动态字体以及针对多种语言实施本地化（第25章会介绍本地化）。

现在BNRDetailViewController可以很好地支持动态字体了，在设置应用中修改用户首选字体，使用不同的字体大小测试BNRDetailViewController界面的显示效果。读者会注意到，界面中的文字会自动缩放，界面布局也会自动调整。

20.4 确定用户首选字体大小

接下来在BNRItemsViewController中使用动态字体，为此需要做出两个方面的修改：根据用户首选字体动态改变UITableView的行高；与BNRDetailViewController类似，修改BNRItemCell，动态改变contentView中文本控件的文字大小。

首先修改UITableView的行高。如果用户选取了较大的字体，那么行高也要相应增加，避免文字过于拥挤。通过UIApplication单例的preferredContentSizeCategory属性就可以确定当前用户首选字体大小，该属性返回表示字体大小的字符串常量，以下是所有字体大小常量：

- UIContentSizeCategoryExtraSmall
- UIContentSizeCategorySmall
- UIContentSizeCategoryMedium
- UIContentSizeCategoryLarge（默认值）
- UIContentSizeCategoryExtraLarge
- UIContentSizeCategoryExtraExtraLarge
- UIContentSizeCategoryExtraExtraExtraLarge

打开BNRItemsViewController.m，创建一个方法，根据用户首选字体大小修改UITableView的行高，然后在viewWillAppear:中调用该方法，代码如下：

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

{

[super viewWillAppear:animated];

~~[self.tableView reloadData];~~

[self updateTableViewForDynamicTypeSize];

}

- (void)updateTableViewForDynamicTypeSize

{

static NSDictionary *cellHeightDictionary;

if (!cellHeightDictionary) {
```

```
cellHeightDictionary = @{

UIContentSizeCategoryExtraSmall : @44,

UIContentSizeCategorySmall : @44,

UIContentSizeCategoryMedium : @44,

UIContentSizeCategoryLarge : @44,

UIContentSizeCategoryExtraLarge : @55,

UIContentSizeCategoryExtraExtraLarge : @65,

UIContentSizeCategoryExtraExtraExtraLarge : @75 };

}

NSString *userSize =

[[UIApplication sharedApplication] preferredContentSizeCategory];

NSNumber *cellHeight = cellHeightDictionary[userSize];

[self.tableView setRowHeight:cellHeight.floatValue];

[self.tableView reloadData];

}
```

构建并运行应用，首先在设置应用中修改用户首选字体，然后重新启动应用，可以发现，UITableView的行高会根据首选字体大小自动调整。（如果没有重新启动应用而是直接返回应用，就需要先进入BNRDetailViewController，再返回BNRItemsViewController，以便BNRItemsViewController可以收到viewWillAppear:消息。）

与BNRDetailViewController相同，BNRItemsViewController也需要注册为UIContentSizeCategoryDidChangeNotification的观察者，观察并响应用户首选字体的改变。

在BNRItemsViewController.m的init方法中将BNRItemsViewController注册为UIContentSizeCategoryDidChangeNotification的观察者，然后在dealloc中移除观察者。注意，响应该通知的方法是刚才实现的updateTableViewForDynamicTypeSize。

```
self.navigationItem.rightBarButtonItem = bbi;

self.navigationItem.leftBarButtonItem = [self editButtonItem];

NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
```

```
[nc addObserver:self
selector:@selector(updateTableViewForDynamicTypeSize)
name:UIContentSizeCategoryDidChangeNotification
object:nil];
}
return self;
}
- (void)dealloc
{
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
[nc removeObserver:self];
}
```

构建并运行应用，这次一旦修改了用户首选字体，UITableView就会立刻根据首选字体大小改变行高。

20.5 修改BNRItemCell

现在UITableView的行高已经可以根据首选字体大小动态改变了，但是BNRItemCell中UILabel对象的文字大小还不会发生任何变化。本节将修改BNRItemCell，支持动态字体。首先查看BNRItemCell各个子视图的现有约束：UIImageView对象位于contentView左边，并保持垂直居中；nameLabel始终位于顶部，serialNumberLabel始终位于底部；valueLabel位于contentView右边，也保持垂直居中（见图20-5）。

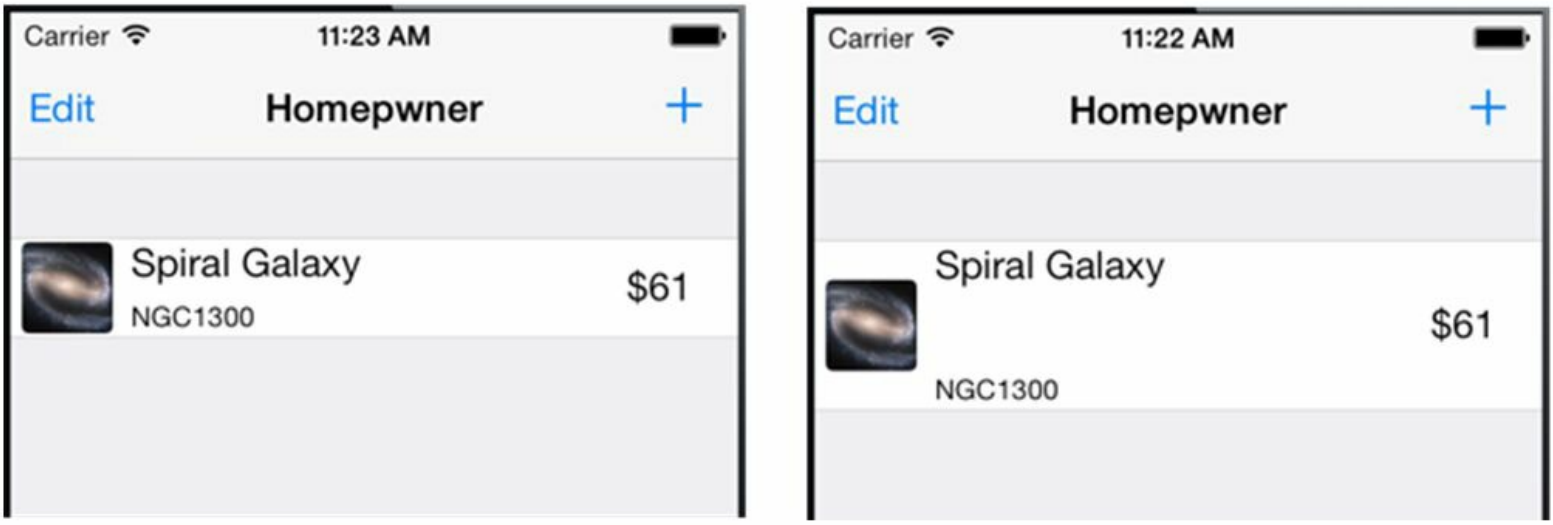


图20-5 BNRItemCell各个子视图的现有约束

下面根据用户首选字体设置UILabel对象的font属性，读者可以参考之前在BNRDetailViewController或BNRItemsViewController中添加的代码。

打开BNRItemCell.m，添加下列方法：

```
- (void)updateInterfaceForDynamicTypeSize

{

UIFont *font = [UIFont preferredFontForTextStyle:UIFontTextStyleBody];

self.nameLabel.font = font;

self.serialNumberLabel.font = font;

self.valueLabel.font = font;

}

- (void)awakeFromNib

{
```

```
[self updateInterfaceForDynamicTypeSize];

NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];

[nc addObserver:self

selector:@selector(updateInterfaceForDynamicTypeSize)

name:UIContentSizeCategoryDidChangeNotification

object:nil];

}

- (void)dealloc

{

NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];

[nc removeObserver:self];

}
```

以上代码只有awakeFromNib与之前有所区别。当NIB文件解档某个对象后，该对象就会收到awakeFromNib消息，可以在awakeFromNib中设置该对象，或者执行其他需要的初始化操作。例如，以上代码在awakeFromNib中设置了UILabel对象的font属性，并将BNRItemCell对象注册为UIContentSizeCategoryDidChangeNotification的观察者。构建并运行应用，现在BNRItemCell可以根据用户首选字体改变界面中的文字大小了。

接下来需要解决BNRItemCell中的一些自动布局问题。

与之前BNRDetailViewController中的问题类似，第19章限定了nameLabel和serialNumberLabel的高度，自动布局系统将无法修改两个UILabel对象的高度。

打开BNRItemCell.xib，删除两个UILabel对象的高度约束，这时Interface Builder会提示视图位置错误。打开Resolve Auto Layout Issues菜单，选择Update All Frames in Homepwner Item Cell。构建并运行应用，现在UILabel对象会根据用户首选字体自动调整高度。

最后还剩下UIImageView对象。下面将修改UIImageView对象的约束，使UIImageView对象可以根据用户首选字体自动调整大小。

为约束添加插座变量

如果需要修改某个视图的大小或位置，无论是修改为固定值还是修改与其他视图之间的相

对关系，都应该修改视图的约束，而不是硬编码视图的frame；否则，当界面需要重新布局时，自动布局系统仍然会根据之前的约束修改视图的frame，从而覆盖硬编码的frame。

为了在运行时根据用户首选字体动态修改UIImageView对象的宽度和高度约束，需要为两个约束添加相应的插座变量。约束是NSLayoutConstraint类的对象，添加插座变量的过程与其他视图对象是相同的。

在BNRItemCell.m的类扩展中，添加并关联宽度和高度约束的插座变量，代码如下：

```
@interface BNRItemCell（）

@property（nonatomic, weak）IBOutlet NSLayoutConstraint

*imageViewHeightConstraint;

@property（nonatomic, weak）IBOutlet NSLayoutConstraint

*imageViewWidthConstraint;

@end
```

现在可以在代码中动态修改UIImageView对象的两个约束了。在BNRItemCell.m中，修改updateInterfaceForDynamicTypeSize，获取用户首选字体大小，然后修改宽度和高度约束的限定值（设置NSLayoutConstraint对象的constant属性）。

```
-（void）updateInterfaceForDynamicTypeSize

{

UIFont *font = [UIFont preferredFontForTextStyle:UIFontTextStyleBody];

self.nameLabel.font = font;

self.serialNumberLabel.font = font;

self.valueLabel.font = font;

static NSDictionary *imageSizeDictionary;

if（！imageSizeDictionary）{

imageSizeDictionary = @{

UIContentSizeCategoryExtraSmall : @40,

UIContentSizeCategorySmall : @40,

UIContentSizeCategoryMedium : @40,
```

```
    UIContentSizeCategoryLarge : @40,
    UIContentSizeCategoryExtraLarge : @45,
    UIContentSizeCategoryExtraExtraLarge : @55,
    UIContentSizeCategoryExtraExtraExtraLarge : @65
    };
    }
    NSString *userSize =
    [[UIApplication sharedApplication] preferredContentSizeCategory];
    NSNumber *imageSize = imageSizeDictionary[userSize];
    self.imageViewHeightConstraint.constant = imageSize.floatValue;
    self.imageViewWidthConstraint.constant = imageSize.floatValue;
    }
```

构建并运行应用，修改用户首选字体，这时UIImageView对象会根据用户首选字体自动调整大小。同时，由于之前限定了nameLabel和serialNumberLabel左边与UIImageView对象右边的距离，因此当UIImageView对象的大小发生变化时，nameLabel和serialNumberLabel也会自动调整位置，保持与UIImageView对象的间距。如果之前限定的是两个UILabel对象左边与父视图的距离，那么UIImageView对象就可能遮住两个UILabel对象，或者留下大片空白。

现在BNRItemCell可以很好地支持动态字体了，但是还需要做最后一处修改。

占位符约束

以上代码同时修改了UIImageView对象的宽度和高度约束，虽然这样做没有问题，但是还有一种更好的解决方案：只修改UIImageView对象的高度约束，然后添加另一个约束，限定UIImageView对象的宽度与高度相同。该约束无法在Interface Builder中添加，必须通过代码创建。回到BNRItemCell.m，在awakeFromNib中为UIImageView对象（thumbnailView属性）添加该约束，代码如下：

```
    - (void)awakeFromNib
    {
    [self updateInterfaceForDynamicTypeSize];
```

```
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
[nc addObserver:self
selector:@selector(updateInterfaceForDynamicTypeSize)
name:UIContentSizeCategoryDidChangeNotification
object:nil];
NSLayoutConstraint *constraint =
[NSLayoutConstraint constraintWithItem:self.thumbnailView
attribute:NSLayoutAttributeHeight
relatedBy:NSLayoutRelationEqual
toItem:self.thumbnailView
attribute:NSLayoutAttributeWidth
multiplier:1
constant:0];
[self.thumbnailView addConstraint:constraint];
}
```

现在可以删除imageViewWidthConstraint属性和相关代码:

```
@interface BNRItemCell ()
@property (nonatomic, weak) IBOutlet NSLayoutConstraint
*imageViewHeightConstraint;
~~@property (nonatomic, weak) IBOutlet NSLayoutConstraint~~
~~*imageViewWidthConstraint;~~
@end
@implementation
- (void)updateInterfaceForDynamicTypeSize
```

```
{
// 这里省略了其他代码……
NSNumber *imageSize = imageSizeDictionary[userSize];
self.imageViewHeightConstraint.constant = imageSize.floatValue;
self.imageViewWidthConstraint.constant = imageSize.floatValue;
}
```

打开BNRItemCell.xib，删除UIImageView对象与imageViewWidthConstraint的关联。

最后还有一个问题：UIImageView对象现在有两个宽度约束。除了以上代码创建的之外，XIB文件中也添加过宽度约束。如果在运行时UIImageView对象的宽度发生变化，就会造成约束冲突。

为了解决该问题，需要删除Interface Builder中添加的宽度约束，但是这么做会导致Interface Builder提示视图位置错误或者布局有歧义。因此，更好的方法是将宽度约束设置为占位符约束（placeholder constraint）。自动布局系统会在构建时移除占位符约束，占位符约束并不会真正对视图起作用。

在BNRItemCell.xib中选中UIImageView对象的宽度约束，然后打开属性检视面板，选中标题为Placeholder的选择框。Placeholder选择框后面的描述也解释了“Placeholder”的含义：Remove at build time（在构建时移除），如图20-6所示。构建并运行应用，界面不会有任何变化，但是现在UIImageView对象的宽度约束是根据高度约束定义的。

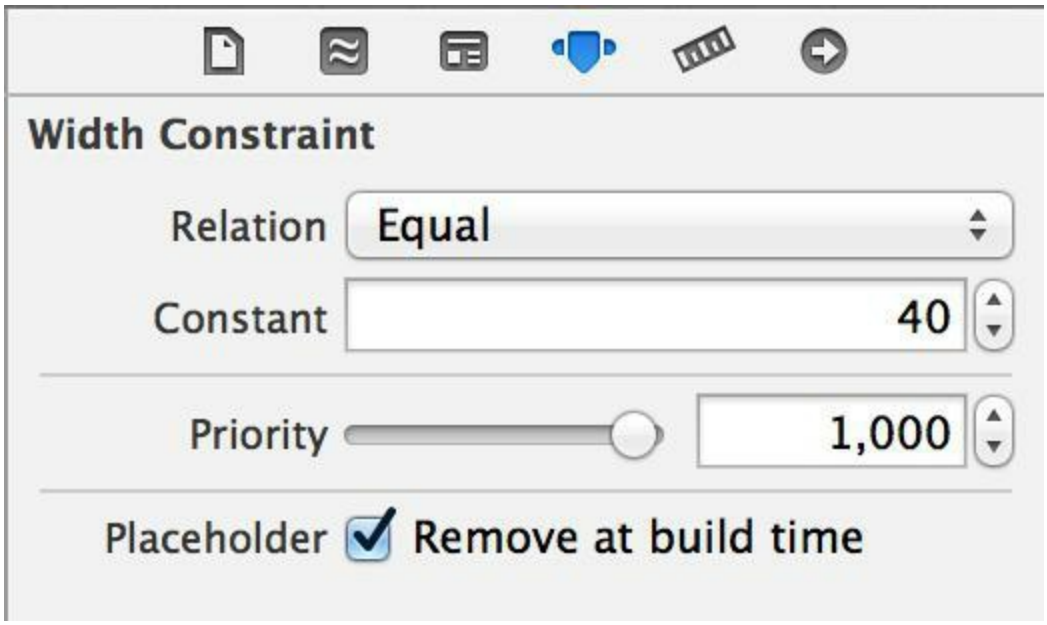


图20-6 占位符约束

第21章 Web服务与UIWebView

本书接下来的两章将开发一款名为Nerdfeed的应用，介绍有关Web服务和UISplitViewController的内容。

Nerdfeed的作用是读取并通过UITableView显示Big Nerd Ranch提供的在线课程，选择某项课程会打开相应课程的网页。图21-1显示的是完成后的Nerdfeed。本章将完成Nerdfeed的基础部分。

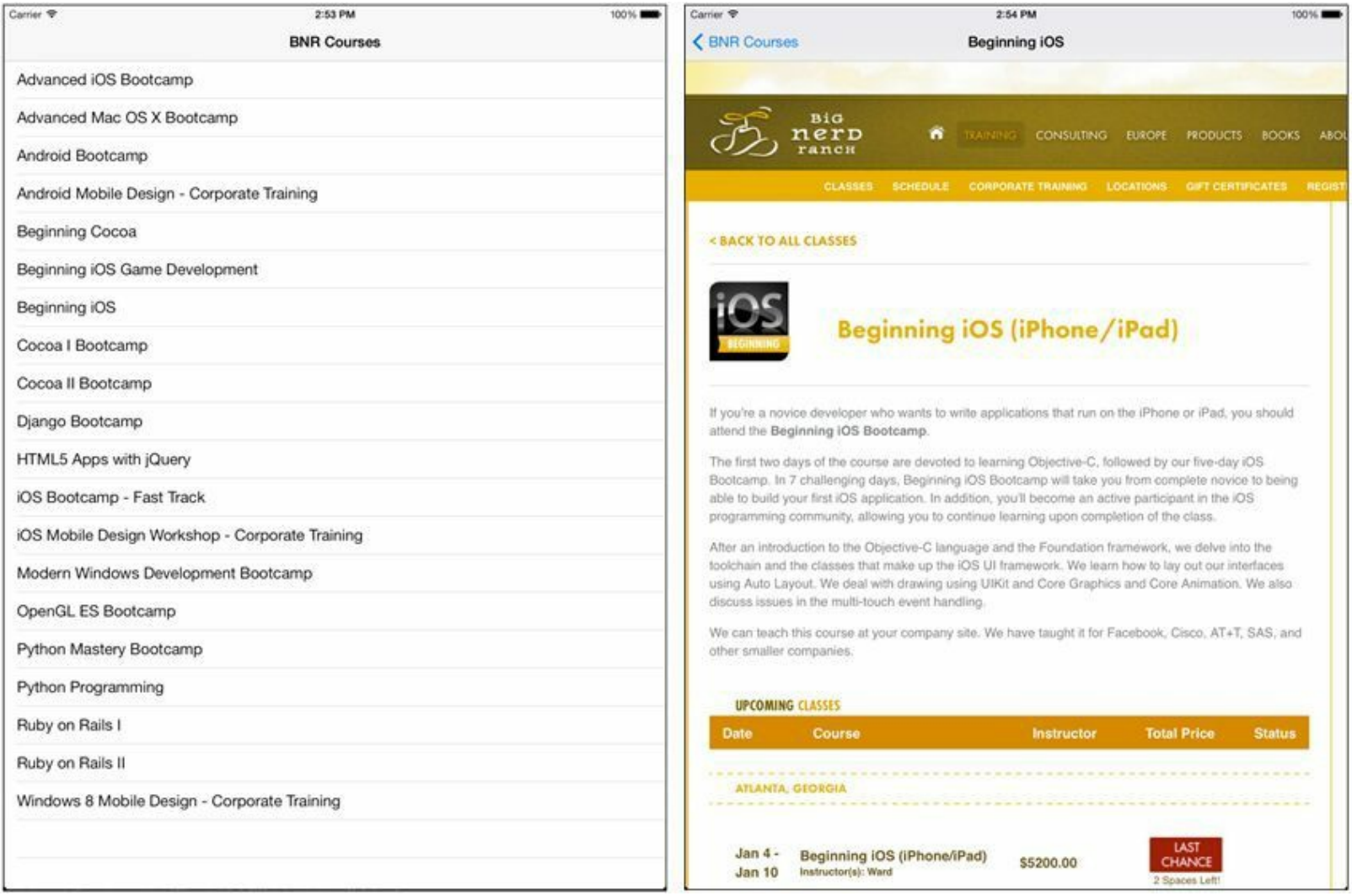


图21-1 Nerdfeed

要实现图21-1所示的Nerdfeed，需要完成两项任务：①连接指定的Web服务并获取数据，然后根据得到的数据创建模型对象。②用UIWebView显示Web内容。图21-2显示的是Nerdfeed的对象图。

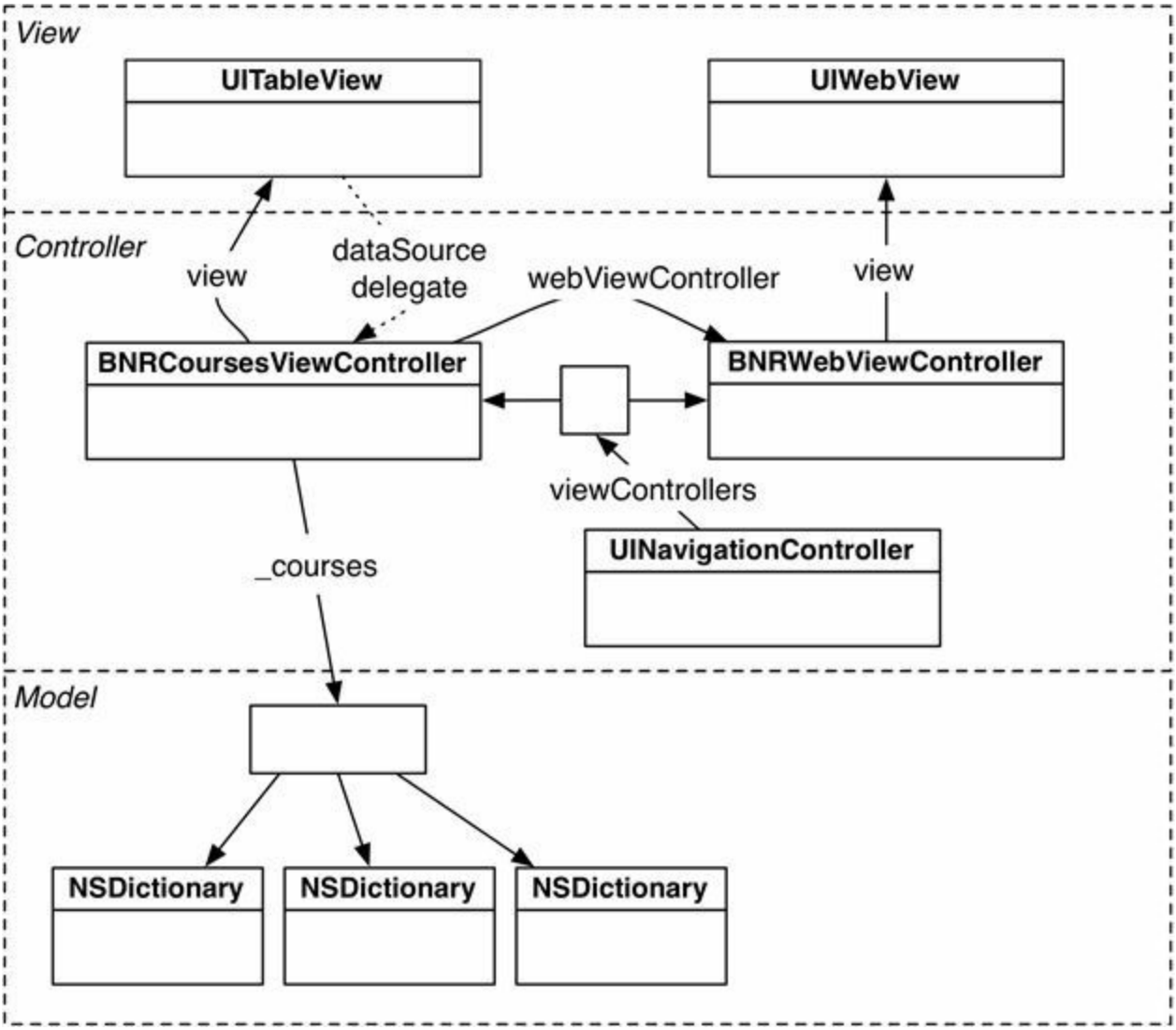


图21-2 Nerdfeed对象图

21.1 Web服务

网页浏览器能通过HTTP协议和指定的Web服务器进行通信。以最简单的交互过程为例，浏览器向服务器发出请求，要求获取指定URL的内容。作为应答，服务器会返回相应的页面（通常是HTML文件和图片）。然后浏览器对这些返回的数据进行排版与显示。

对更复杂的交互过程，浏览器可以在发出的请求中包含其他参数，例如表单数据（form data）。服务器会处理这些参数并返回定制的或动态生成的网页。

Web浏览器的使用很广泛，而且经历了很长一段的发展时期。因此，与HTTP有关的技术都已经相当成熟：HTTP数据可以轻松通过多数的防火墙，Web服务器有很高的安全性并且执行效率很高。此外，用于开发Web应用的工具也越来越容易使用。

读者可以编写iOS客户端应用，通过HTTP协议与运行Web服务的服务器进行通信。这个服务器端称为Web服务。通过HTTP协议，iOS客户端可以与Web服务相互发出请求和给予响应。

HTTP协议没有规定传输数据的格式，所以客户端和服务端可以相互传输复杂的数据。这些传输数据的格式通常是JSON（JavaScript Object Notation）或XML。如果读者可以管理服务器，就可以随意选用数据传输格式。否则，就只能根据服务器支持的格式编写应用。

Nerdfeed的任务是向位于http://bookapi.bignerdranch.com的courses服务发起请求。本例中，courses服务将返回Big Nerd Ranch的最新在线课程，数据格式为JSON。

编写Nerdfeed应用

使用Empty Application模板创建一个iPad应用（选择Devices列表中的iPad）并将其命名为Nerdfeed，如图21-3所示（如果读者没有iPad，也可以使用iPad模拟器）。

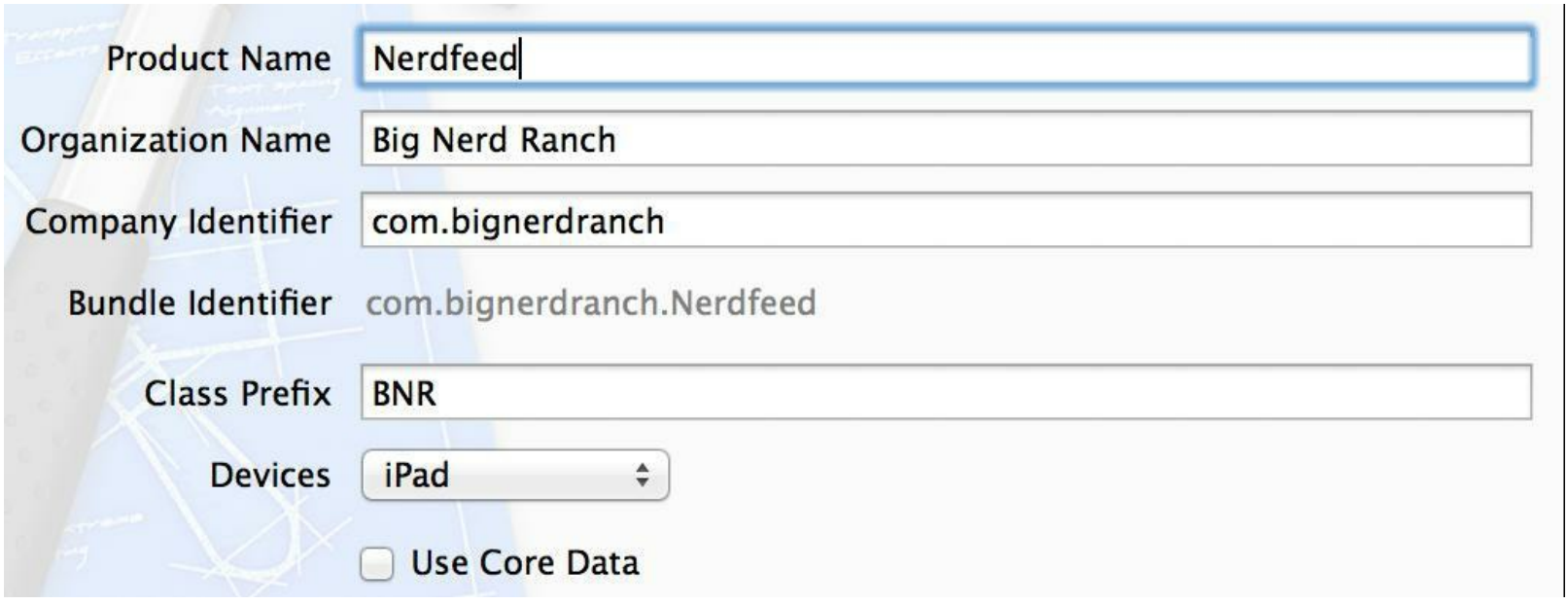


图21-3 通过Empty Application模板创建iPad应用

在为Nerdfeed实现Web服务功能前，需要先构建一个简单的用户界面。创建一个新的NSObject子类并将其命名为BNRCoursesViewController。在BNRCoursesView- Controller.h中，将父类修改为UITableViewController，代码如下：

~~@interface BNRCoursesViewController : NSObject~~

@interface BNRCoursesViewController : UITableViewController

在BNRCoursesViewController.m中实现空的数据源方法，能使项目可以成功构建，代码如下：

- (NSInteger)tableView:(UITableView *)tableView

numberOfRowsInSection:(NSInteger)section

{

return 0;

}

- (UITableViewCell *)tableView:(UITableView *)tableView

cellForRowAtIndexPath:(NSIndexPath *)indexPath

{

return nil;

}

在BNRAppDelegate.m中，先创建一个BNRCoursesViewController对象，然后使用该对象创建一个UINavigationController对象，最后将UINavigationController对象设置为UIWindow的rootViewController。代码如下：

#import “BNRAppDelegate.h”

#import “BNRCoursesViewController.h”

@implementation BNRAppDelegate

- (BOOL)application:(UIApplication *)application

didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

{

self.window =

```
    [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    BNRCoursesViewController *cvc = [[BNRCoursesViewController alloc]
    initWithStyle:UITableViewStylePlain];
    UINavigationController *masterNav =
    [[UINavigationController alloc] initWithRootViewController:cvc];
    self.window.rootViewController = masterNav;
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
    }
```

构建并运行应用，Nerdfeed应该会显示一个UINavigationBar对象和一个空的UITableView对象。

NSURL、NSURLRequest、NSURLSession和NSURLSessionTask

要从Web服务器获取数据，Nerdfeed需要使用NSURL、NSURLRequest、NSURLSession- Task和NSURLSession四个类（见图21-4）。

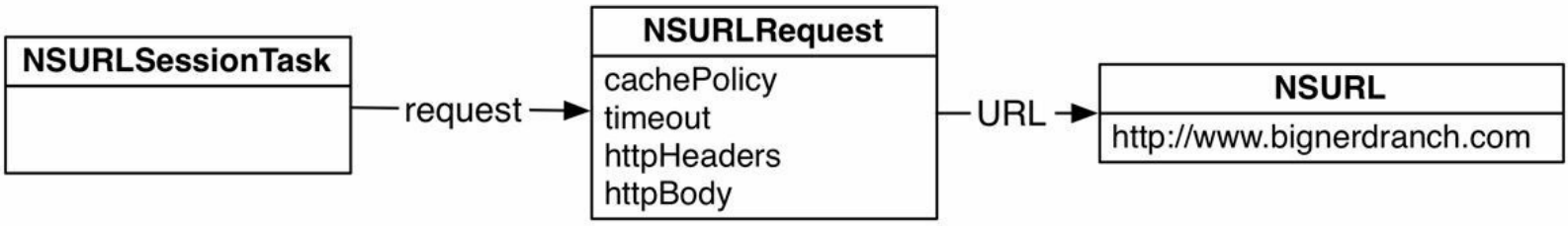


图21-4 NSURL、NSURLRequest、NSURLSessionTask关系图

在与Web服务器进行通信的过程中，这些类各自扮演了重要的角色。

•NSURL对象负责以URL的格式保存Web应用的位置。对大多数Web服务，URL将包含基地址（base address）、Web应用名和需要传送的参数。

•NSURLRequest对象负责保存需要传送给Web服务器的全部数据，这些数据包括：一个NSURL对象、缓存方案（caching policy）、等待Web服务器响应的最长时间和需要通过HTTP协议传送的额外信息（NSMutableURLRequest是NSURLRequest的可变子类）。

•每一个NSURLSessionTask对象都表示一个NSURLRequest的生命周期。NSURLSessionTask可以跟踪NSURLRequest的状态，还可以对NSURLRequest执行取消、暂停和继续操作。NSURLSessionTask有多种不同功能的子类，包括NSURLSession- DataTask，NSURLSessionUploadTask和NSURLSessionDownloadTask。

•NSURLSession对象可以看作是一个生产NSURLSessionTask对象的工厂。可以设置其生产出的NSURLSessionTask对象的通用属性，例如请求头的内容（参见本章21.6节）、是否允许在蜂窝网络下发送请求等。NSURLSession对象还有一个功能强大的委托，可以跟踪NSURLSessionTask对象的状态、处理服务器的认证要求等。

构建URL与发送请求

不同的Web服务，请求格式也不同。Web服务没有一成不变的规则，读者需要查阅相关Web服务的文档，然后根据指定的格式构建请求。客户端发送的数据必须符合服务器的要求，才能得到相应的响应。

根据Big Nerd Ranch在线课程的Web服务要求，相应的URL示例如下：

<http://bookapi.bignerdranch.com/courses.json>

上述URL的基地址是bookapi.bignerdranch.com，Web服务是courses，服务器响应的数据格式是json。

Web服务请求根据实际需要会有多种构成格式。以上Web服务请求实际上访问了服务器端的courses路径（读者可以将基地址后的“/”类比为Mac文件系统中的目录路径），这类格式很常见——服务器端根据功能将Web服务分配到不同的路径，然后要求客户端以路径作为参数请求不同的Web服务。因此，以上Web服务请求的含义是：“请求所有在线课程信息，并以JSON格式返回”。

还有一种很常见的Web服务请求构成格式：

<http://baseURL.com/serviceName?argumentX=valueX&argumentY=valueY>

例如，如果要获取指定年份和月份的在线课程，那么可以构建如下Web服务请求：

http://bookapi.bignerdranch.com/courses.json?year=2014&month=11

URL字符串必须是URL安全的（URL-safe）。例如，在URL中，不允许出现空格字符和双引号，必须使用转义序列（escape-sequence）来替换这些字符：

NSString *search = @“Play some \”Abba\“”;

NSString *escaped =

[search stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding];

// 转义后的字符串是“Play%20some%20%22Abba%22”

之后，如果需要对字符串解除转义，可以使用NSString的实例方法stringByRemovingPercentEncoding。

-（NSString *）stringByRemovingPercentEncoding

Big Nerd Ranch的服务器处理完来自客户端的请求后，会返回包含在线课程列表的JSON数据。BNRCoursesViewController对象（发送Web服务请求的对象）的UITableView对象需要显示各个在线课程的标题。

NSURLSession

NSURLSession有两种不同的含义：第一种含义指NSURLSession类；第二种含义指一组用于处理网络请求的API。为了区别这两种含义，本书会在第一种含义下直接使用“NSURLSession”，而在第二种含义下则使用“NSURLSession API”。

在BNRCoursesViewController.m的类扩展中添加一个属性，用于保存NSURLSession对象：

```
@interface BNRCoursesViewController（）

@property（nonatomic）NSURLSession *session;

@end
```

接下来覆盖BNRCoursesViewController.m的initWithStyle:，创建NSURLSession对象：

```
-（instancetype）initWithStyle:（UITableViewStyle）style

{

self = [super initWithStyle:style];

if（self）{

self.navigationItem.title = @“BNR Courses”;

NSURLSessionConfiguration *config =

[NSURLSessionConfiguration defaultSessionConfiguration];

_session = [NSURLSession sessionWithConfiguration:config

delegate:nil
```

```
delegateQueue:nil];

}

return self;

}
```

用于创建NSURLSession对象的工厂方法sessionWithConfiguration有三个参数：NSURLSessionConfiguration对象、委托和委托队列，本例中全部使用默认值即可。其中，第一个参数需要创建一个默认的NSURLSessionConfiguration对象，而第二个和第三个参数只需要传入nil就可以获得相应的默认值。

下面在BNRCoursesViewController.m中实现fetchFeed方法：首先创建一个NSURLRequest对象（用于连接bookapi.bignerdranch.com并查询在线课程列表），然后使用NSURLSession对象创建一个NSURLSessionDataTask对象，将NSURLRequest对象发送给服务器。代码如下：

```
- (void)fetchFeed

{

NSString *requestString =

@"http://bookapi.bignerdranch.com/courses.json";

NSURL *url = [NSURL URLWithString:requestString];

NSURLRequest *req = [NSURLRequest requestWithURL:url];

NSURLSessionDataTask *dataTask =

[self.session dataTaskWithRequest:req

completionHandler:

^(NSData *data, NSURLResponse *response, NSError *error) {

NSString *json = [[NSString alloc] initWithData:data

encoding:NSUTF8StringEncoding];

NSLog(@"%@", json);

}];

[dataTask resume];

}
```

创建NSURLRequest对象的过程非常简单，首先创建NSURL对象，然后使用NSURL对象创建NSURLRequest对象。

由于Nerdfeed只会发送单个简单请求，因此代码使用sharedSession方法获取使用默认配置的NSURLSession单例就可以了。实际上，如果应用中的请求种类较多，参数复杂，那么一个应用中也可能存在多个NSURLSession对象。

现在，只要为NSURLSession对象提供一个NSURLRequest对象和一个Block对象（completionHandler），NSURLSession对象就会创建一个NSURLSessionTask对象。由于Nerdfeed是从Web服务中请求数据，所以应该使用NSURLSessionTask的子类——NSURLSessionDataTask。NSURLSessionTask在刚创建的时候都处于暂停状态，所以需要手动调用resume（继续）方法，让NSURLSessionTask开始向Web服务发送请求。另外，completionHandler中暂时只向控制台中输出返回的JSON数据，下一节会介绍如何处理JSON数据。

Nerdfeed应该在创建BNRCoursesViewController对象后就发起网络请求。修改BNRCoursesViewController.m的initWithStyle:，调用fetchFeed方法：

```
- (instancetype)initWithStyle:(UITableViewStyle)style

{

self = [super initWithStyle:style];

if (self) {

self.navigationItem.title = @“BNR Courses”;

NSURLSessionConfiguration *config =

[NSURLSessionConfiguration defaultSessionConfiguration];

_session = [NSURLSession sessionWithConfiguration:config

delegate:nil

delegateQueue:nil];

[self fetchFeed];

}

return self;

}
```

构建并运行应用。确保iOS设备可以访问网络（如果是iOS模拟器，确保Mac可以访问网络），稍等片刻，应用会从Web服务获取JSON数据，然后将其转换为字符串并输出到控制台（如果控

制台中没有任何输出，请检查Web服务的URL是否正确）。

JSON数据

读者可能会觉得控制台中输出的JSON数据非常复杂，但其实JSON的语法非常简单易懂。JSON只包含有限的几种基础对象，用于表示来自服务器的模型对象，例如数组、字典、字符串和数字。数组可能包含多个字符串、数字、字典或其他数组；而字典则可能包含一到多个键-值对，其中键是字符串，而值可能是字符串、数字、数组或其他字典。每个JSON文件都是由这些基础对象嵌套组合而成的。

以下是一个非常简单的JSON文件：

```
{
“name” : “Christian”,
“friends” : [“Aaron”, “Mikey”],
“job” : {
“company” : “Big Nerd Ranch”,
“title” : “Senior Nerd”
}
}
```

JSON中的字典通过花括号（{和}）表示，花括号内是字典的键-值对。可以看出，以上JSON文件以左花括号开始，右花括号结束，意味着JSON文件的顶层对象（top-level object）是一个字典，该字典包含三个键-值对，分别是name、friends和job。

字符串通过引号中的文本表示。在JSON字典中，字符串既可以作为键，也可以作为值。顶层字典name键的值是字符串“Christian”。

JSON中的数组通过中括号（[和]）表示。数组可以包含其他JSON对象。顶层字典friends键的值是一个字符串数组，包括Aaron和Mikey。

字典中还可以包含其他字典。顶层字典job键的值是另一个字典，包含两个键-值对，分别是company和title。

Nerdfeed将解析服务器返回的JSON数据，封装为在线课程模型对象，并将模型对象存储到courses属性中，如图21-5所示。

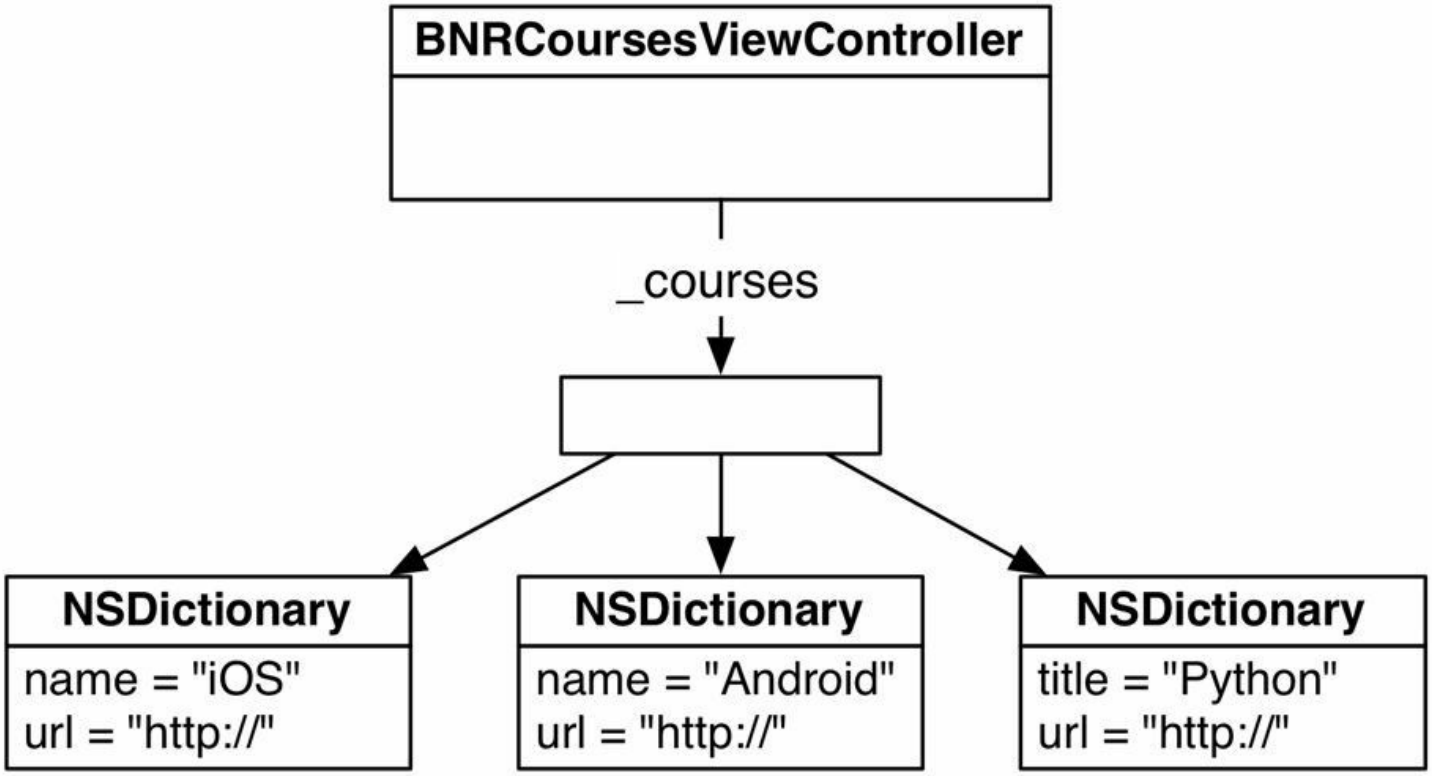


图21-5 在线课程模型对象图

解析JSON数据

Apple提供了专门用于解析JSON数据的类:NSJSONSerialization。NSJSON- Serialization可以将JSON数据中的对象转换为对应的Objective-C对象——字典会转换为NSDictionary对象;数组会转换为NSArray对象;字符串会转换为NSString对象;数字会转换为NSNumber对象。

打开BNRCoursesViewController.m, 修改NSURLSessionDataTask的completion- Handler, 使用NSJSONSerialization将JSON数据转换为Objective-C对象:

```
^(NSData *data, NSURLResponse *response, NSError *error) {

~~NSString *json = [[NSString alloc] initWithData:data~~

~~encoding:NSUTF8StringEncoding];~~

~~NSLog(@"%@", json);~~

NSDictionary *jsonObject =

[NSJSONSerialization JSONObjectWithData:data

options:0

error:nil];
```

```
NSLog(@"%@", jsonObject);

}
```

构建并运行应用，稍等片刻后检查控制台。这次控制台会输出解析后的JSON数据——NSLog对象可以格式化输出NSDictionary和NSArray，而jsonObject是NSDictionary对象，有一个NSString键"courses"，值是一个NSArray对象，包括所有在线课程信息。

当NSURLSessionDataTask完成网络请求之后，就可以使用NSJSONSerialization解析Web服务返回的JSON数据。图21-6显示了解析后的数据格式。

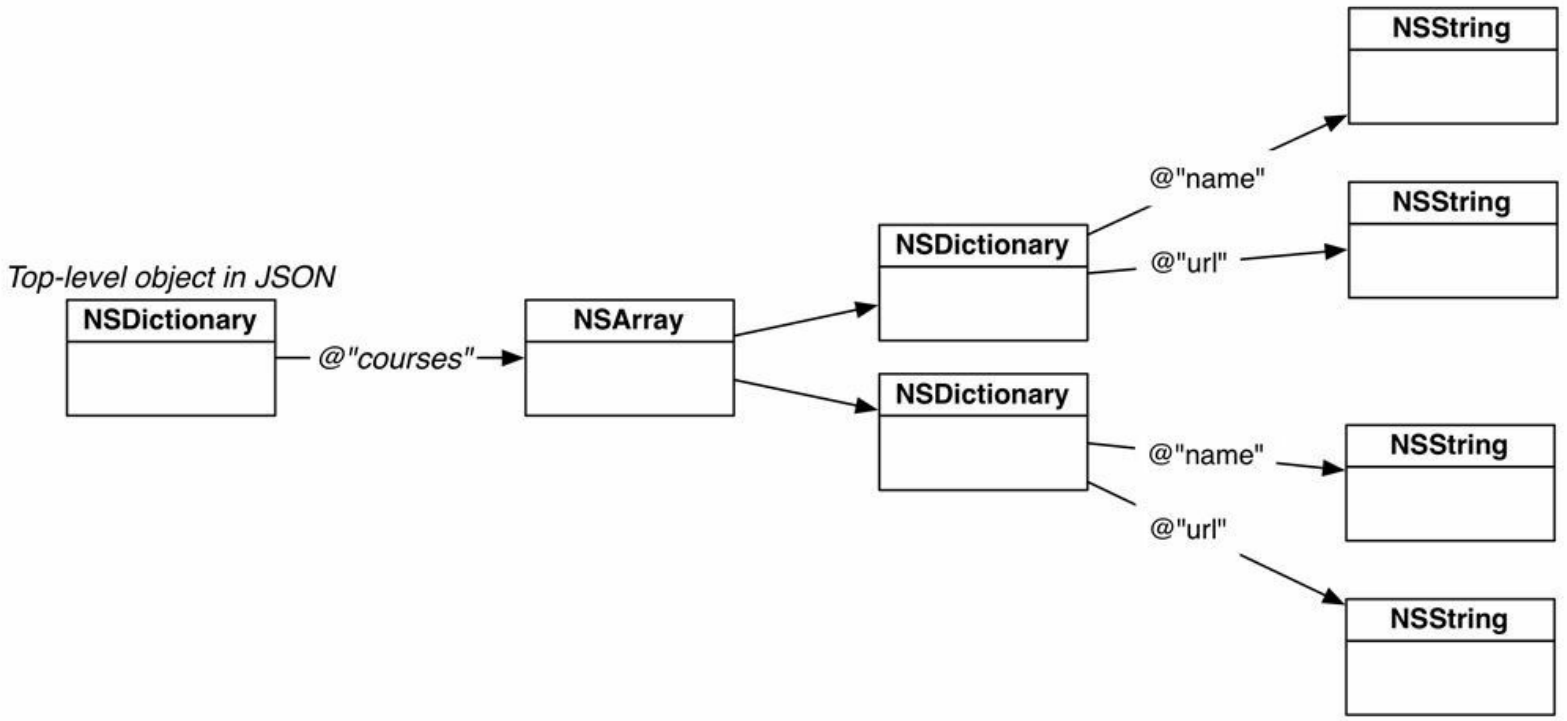


图21-6 解析后的JSON数据

在BNRCoursesViewController.m的类扩展中添加一个属性，用于保存在线课程数组。数组中的每一个元素都是一个NSDictionary对象，表示一项课程的详细信息。代码如下：

```
@interface BNRCoursesViewController ()

@property (nonatomic) NSURLSession *session;

@property (nonatomic, copy) NSArray *courses;

@end
```

然后修改NSURLSessionDataTask的completionHandler，代码如下：

```
^(NSData *data, NSURLResponse *response, NSError *error) {

NSDictionary *jsonObject =
```

```
[NSJSONSerialization JSONObjectWithData:data

options:0

error:nil];

NSLog(@"%@", jsonObject);

self.courses = jsonObject[@"courses"];

NSLog(@"%@", self.courses);

}
```

接下来修改UITableView的数据源方法，将各项课程的标题(title)显示在相应的UITableViewCell中。最后还需要覆盖viewDidLoad，向UITableView注册UITableView- Cell。代码如下：

```
- (void)viewDidLoad

{

[super viewDidLoad];

[self.tableView registerClass:[UITableViewCell class]

forCellReuseIdentifier:@"UITableViewCell"];

}

- (NSInteger)tableView:(UITableView *)tableView

numberOfRowsInSection:(NSInteger)section

{

return 0;

return self.courses.count;

}

- (UITableViewCell *)tableView:(UITableView *)tableView

cellForRowAtIndexPath:(NSIndexPath *)indexPath

{
```

~~return nil;~~

UITableViewCell *cell =

[tableView dequeueReusableCellWithIdentifier:@“UITableViewCell”

forIndexPath:indexPath];

NSDictionary *course = self.courses[indexPath.row];

cell.textLabel.text = course[@“title”];

return cell;

}

主线程

随着iPhone 4S中A5处理器的问世，具备多核处理器的iOS设备已经成为主流。多核处理器可以同时运行多段代码，每段代码都在一个独立的线程（thread）上运行，互不干扰。这种特征称为并发性（concurrency）。到目前为止，本书的代码都是在主线程（main thread）中运行的。主线程有时也称为用户界面线程（UI thread），这是由于修改用户界面的代码必须在主线程中运行。

当Web服务请求成功后，BNRCoursesViewController需要重新加载UITableView对象的数据（调用UITableView对象的reloadData方法）。默认情况下，NSURLSession- DataTask是在后台线程中执行completionHandler的，为了让reloadData方法在主线程中运行，可以使用dispatch_async函数。

在BNRCoursesViewController.m中，修改NSURLSessionDataTask的completion- Handler，在主线程中重新加载UITableView对象的数据：

^（NSData *data, NSURLResponse *response, NSError *error）{

NSDictionary *jsonObject =

[NSJSONSerialization JSONObjectWithData:data

options:0

error:nil];

self.courses = jsonObject[@“courses”];

NSLog（@“%@”, self.courses）;

```
    dispatch_async(dispatch_get_main_queue(), ^{

    [self.tableView reloadData];

    });

    }
```

构建并运行应用。当Web服务请求成功后，应该可以在UITableView对象中看到Big Nerd Ranch的在线课程列表。

21.2 UIWebView

之前介绍过，课程的详细信息是一个NSDictionary对象，其中含有“url”键，值是一个URL字符串，表示对应的课程网址。下面要让用户在选中某项在线课程后，可以浏览该课程的网页。读者可能会想到让Nerdfeed启动Safari浏览器，但是如果用户在查看了网页之后需要继续使用Nerdfeed，就必须再从Safari浏览器切换回Nerdfeed。其实还有更好的解决方案——Apple提供了UIWebView，可以不用切换至Safari就能在应用内直接打开网页。

UIWebView对象可以显示指定的网页内容。实际上，iOS设备和模拟器中的Safari浏览器也是通过UIWebView对象来显示网页内容的。本节将创建一个新的视图控制器，它的view是一个UIWebView对象。当用户在UITableView对象中选中某项在线课程时，Nerdfeed要将这个视图控制器压入UINavigationController栈，然后要求UIWebView对象载入相应的课程网页。

创建一个新的NSObject子类并将其命名为BNRWebViewController。在BNRWebViewController.h中声明一个新属性URL，然后将BNRWebViewController的父类改为UIViewController，代码如下：

```
@interface BNRWebViewController : NSObject

@interface BNRWebViewController : UIViewController

@property (nonatomic) NSURL *URL;

@end
```

在BNRWebViewController.m中，加入以下代码：

```
@implementation BNRWebViewController

- (void)loadView

{

UIWebView *webView = [[UIWebView alloc] init];

webView.scalesPageToFit = YES;

self.view = webView;

}

- (void)setURL:(NSURL *)URL

{

_URL = URL;
```

```
if (_URL) {

NSURLRequest *req = [NSURLRequest requestWithURL:_URL];

[(UIWebView *)self.view loadRequest:req];

}

}

@end
```

在BNRCoursesViewController.h中，声明一个新属性，指向BNRWebViewController对象。代码如下：

```
@class BNRWebViewController;

@interface BNRCoursesViewController : UITableViewController

@property (nonatomic) BNRWebViewController *webViewController;

@end
```

在BNRAppDelegate.m中，导入BNRWebViewController.h，创建BNRWebView- Controller对象并将其赋给BNRCoursesViewController对象的webViewController属性。代码如下：

```
#import "BNRWebViewController.h"

@implementation BNRAppDelegate

- (BOOL)application:(UIApplication *)application

didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

{

self.window =

[[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];

BNRCoursesViewController *cvc = [[BNRCoursesViewController alloc]

initWithStyle:UITableViewStylePlain];

UINavigationController *masterNav =

[[UINavigationController alloc] initWithRootViewController:cvc];
```

```
BNRWebViewController *wvc = [[BNRWebViewController alloc] init];

cvc.webViewController = wvc;

self.window.rootViewController = masterNav;

self.window.backgroundColor = [UIColor whiteColor];

[self.window makeKeyAndVisible];

return YES;

}
```

（注意，之前的项目都是由UINavigationController对象的当前视图控制器负责创建下一个需要入栈的视图控制对象。而本章的Nerdfeed则是由BNRAppDelegate创建BNRWebViewController对象。这样做的目的是为第22章做准备：第22章的Nerdfeed在iPad中将改用UISplitViewController显示视图控制对象。）

在BNRCoursesViewController.m中，导入BNRWebViewController.h，然后实现tableView:didSelectRowAtIndexPath:。当用户点击UITableView对象中的某个UITableViewCell后，Nerdfeed会创建BNRWebViewController对象并将其压入UINavigationController栈。代码如下：

```
#import “BNRWebViewController.h”

@implementation BNRCoursesViewController

-（void）tableView:（UITableView *）tableView

didSelectRowAtIndexPath:（NSIndexPath *）indexPath

{

NSDictionary *course = self.courses[indexPath.row];

NSURL *URL = [NSURL URLWithString:course[@“url”]];

self.webViewController.title = course[@“title”];

self.webViewController.URL = URL;

[self.navigationController pushViewController:self.webViewController

animated:YES];

}
```

构建并运行应用。点击UITableView对象中的某项在线课程，Nerdfeed应该会打开新的视图控制器并显示相应课程的网页。

21.3 认证信息

Web服务可以在返回HTTP响应时附带认证要求(authentication challenge)，其作用是询问“HTTP请求的发起方是谁？”这时发起方应该提供相应的用户名和密码(即认证信息)，只有当认证通过后，Web服务才会返回真正的HTTP响应。

当应用收到认证要求时，NSURLSession的委托会收到URLSession:task:didReceiveChallenge:completionHandler:消息，可以在该消息中发送用户名和密码，完成认证。

打开BNRCoursesViewController.m，更新fetchFeed方法，访问安全度更高的Big Nerd Ranch在线课程Web服务(注意将http改为https)，代码如下：

```
- (void)fetchFeed

{

~~NSString *requestString =~~

~~@“http://bookapi.bignerdranch.com/courses.json”;~~

NSString *requestString =

@“https://bookapi.bignerdranch.com/private/courses.json”;

NSURL *url = [NSURL URLWithString:requestString];

NSURLRequest *req = [NSURLRequest requestWithURL:url];
```

下面需要在初始化BNRCoursesViewController时为NSURLSession设置委托。更新initWithStyle:方法，代码如下：

```
- (instancetype)initWithStyle:(UITableViewStyle)style

{

self = [super initWithStyle:style];

if (self) {

self.navigationItem.title = @“BNR Courses”;

NSURLSessionConfiguration *config =

[NSURLSessionConfiguration defaultSessionConfiguration];

~~_session = [NSURLSession sessionWithConfiguration:config~~
```

~~delegate:nil~~

~~delegateQueue:nil];~~

_session = [NSURLSession sessionWithConfiguration:config

delegate:self

delegateQueue:nil];

[self fetchFeed];

}

return self;

}

然后在BNRCoursesViewController.m的类扩展中，使BNRCoursesViewController遵守NSURLSessionDataDelegate协议：

@interface BNRCoursesViewController ()<NSURLSessionDataDelegate>

@property (nonatomic) NSURLSession *session;

@property (nonatomic, copy) NSArray *courses;

@end

构建并运行应用，Web服务会返回一条错误消息：未认证的访问请求（unauthorized access）。因此，Web服务不会返回任何数据，UITableView对象中也没有显示任何在线课程信息。

为了通过Web服务认证，需要实现NSURLSession收到认证要求的委托方法。该方法会提供一个Block对象，可以将认证信息传入这个Block对象，完成认证。

在BNRCoursesViewController.m中，实现URLSession:task:didReceiveChallenge:completionHandler:方法，处理Web服务的认证要求：

-（void）URLSession:（NSURLSession *）session task:（NSURLSessionTask *）task

didReceiveChallenge:（NSURLAuthenticationChallenge *）challenge

completionHandler:

（void（^）（NSURLSessionAuthChallengeDisposition,

NSURLCredential *））completionHandler

```
{

NSURLCredential *cred = [NSURLCredential

credentialWithUser:@“BigNerdRanch”

password:@“AchieveNerdvana”

persistence:NSURLCredentialPersistenceForSession];

completionHandler(NSURLSessionAuthChallengeUseCredential, cred);

}
```

completionHandler有两个参数。第一个参数是认证类型，由于NSURLCredential对象中提供了用户名和密码，因此类型是NSURLSessionAuthChallengeUseCredential。第二个参数是认证信息，即NSURLCredential对象。以上代码使用NSURLCredential的工厂方法credentialWithUser:password:persistence:创建NSURLCredential对象，该方法需要提供用户名、密码和认证信息的有效期限（有效期限是一个枚举值）。

构建并运行应用，运行结果与之前的相同。但是现在应用访问的是安全度更高的Big Nerd Ranch在线课程Web服务。

25.4 中级练习：加强UIWebView

每个UIWebView对象都可以保存一份独立的浏览历史。向某个UIWebView对象发送goBack消息和goForward消息，可以根据浏览历史打开之前浏览过的网页。试创建一个UIToolbar对象并将其加入BNRWebViewController对象的视图层次结构。为新创建的UIToolbar对象增加后退按钮和前进按钮，当用户按下相应的按钮时，UIWebView对象应该可以来回打开浏览历史中的网页。额外练习：通过UIWebView对象的另外两个属性，设置后退按钮和前进按钮的有效状态（enable）与无效状态（disable）。

21.5 高级练习：课程预告

在线课程Web服务除了提供课程的常规信息之外，还会提供相应的课程预告信息，包括课程起止日期、导师信息和上课地点。创建一个新的UITableViewCell子类，显示课程标题和最近一次课程的上课时间。（注意：并非所有课程都会提供课程预告信息。）

21.6 深入学习：HTTP请求主体

NSURLSessionTask会使用HTTP协议来和Web服务进行通信，发送和接收的数据必须符合HTTP规范。图21-7显示的是本章所编写的Nerdfeed向服务器发送的实际数据示例。

NSURLRequest可以提供多个方法，用来设置HTTP请求的各个方面。NSURLRequest对象会根据具体的设置，生成符合规范的发送数据。

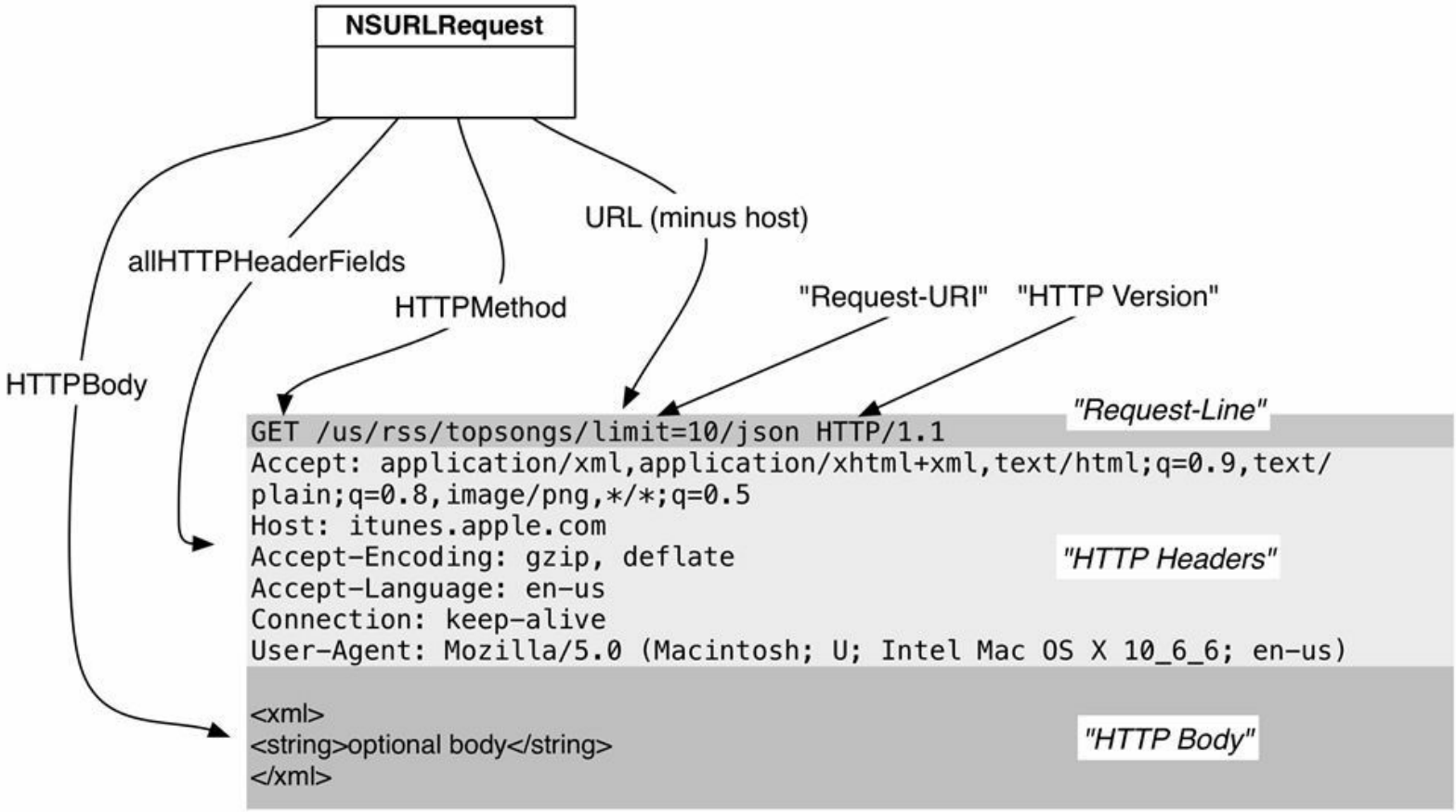


图21-7 HTTP请求格式

HTTP请求分为请求行、HTTP头（header）和HTTP主体（body）三个部分。其中的HTTP主体是可选的。请求行（在Apple的API中称为状态行）是HTTP请求的第一行数据，其作用是告诉服务器客户端想做什么。对本例中的请求，客户端想要通过GET方法得到位于/courses.json的资源（此外，该请求还指明了当前使用的HTTP规范的版本号）。

这里的GET命令是一种HTTP方法。虽然HTTP规范定义了多种HTTP方法，最常见的是GET和POST。NSURLRequest默认会使用GET方法，GET方法代表客户端想要从服务器得到某些信息。而信息所在的位置称为Request-URI（/courses.json）。

在互联网早期，Request-URI通常是某个文件在服务器上的路径。例如，请求http://www.website.com/index.html会返回文件index.html，然后由浏览器在其窗口中显示该文件。现在，也可以用Request-URI来表示服务器实现的某个Web服务。例如，本章的Nerdfeed会向courses服务发送请求，提供相应的参数并得到服务器返回的JSON数据。一样是通过GET方法获取信息，但是服务器能够根据传入的参数执行更多的任务。

除了从服务器获取信息，还可以向服务器发送信息。例如，很多Web服务支持上传照片。某个客户端应用可以通过服务请求将照片数据传给服务器。向服务器发送数据时，需要使用HTTP的POST方法。POST方法代表客户端想要向服务器发送某些信息，并将这些信息包含在（可选的）HTTP主体中。HTTP请求的主体是传送给服务器的数据，这些数据通常是XML格式、JSON格式或Base-64编码后的数据。

如果某个HTTP请求包含主体，就必须包含Content-Length头。NSURLRequest会计算主体的大小并自动添加Content-Length头。代码如下：

```
NSURL *someURL = [NSURL URLWithString:@“http://www.photos.com/upload”];

UIImage *image = [self profilePicture];

NSData *data = UIImagePNGRepresentation(image);

NSMutableURLRequest *req =

[NSMutableURLRequest requestWithURL:someURL

cachePolicy:NSURLRequestReloadIgnoringCacheData

timeoutInterval:90];

// 加入HTTP主体数据，NSMutableURLRequest对象会自动加入相应的Content-Length头

req.HTTPBody = data;

// 这行代码会修改请求行中的HTTP方法

req.HTTPMethod = @“POST”;

// 也可以通过代码“手动”设置Content-Length

[req setValue:[NSString stringWithFormat:@“%d”, data.length]

forHTTPHeaderField:@“Content-Length”];
```

第22章 UISplitViewController

iPhone和iPod touch的屏幕空间有限。因此，当应用需要显示垂直(drill-down)界面时，可以通过UINavigationController在列表视图和详细视图之间切换。

iPad有足够大的空间，可以通过Cocoa Touch提供的UISplitViewController类，同时显示这两类视图。UISplitViewController会以主-从(master-detail)的关系来显示两个视图控制器。主视图控制器会占据屏幕左侧的狭长区域，从视图控制器会占据余下的空间。UISplitViewController只能用于iPad。

本章将指导读者改写Nerdfeed，使其能够在iPad上运行，并使用UISplitViewController显示相应的视图控制器(见图22-1)。此外，还要将Nerdfeed修改为通用应用，当其在iPhone上运行时，仍然使用UINavigationController。

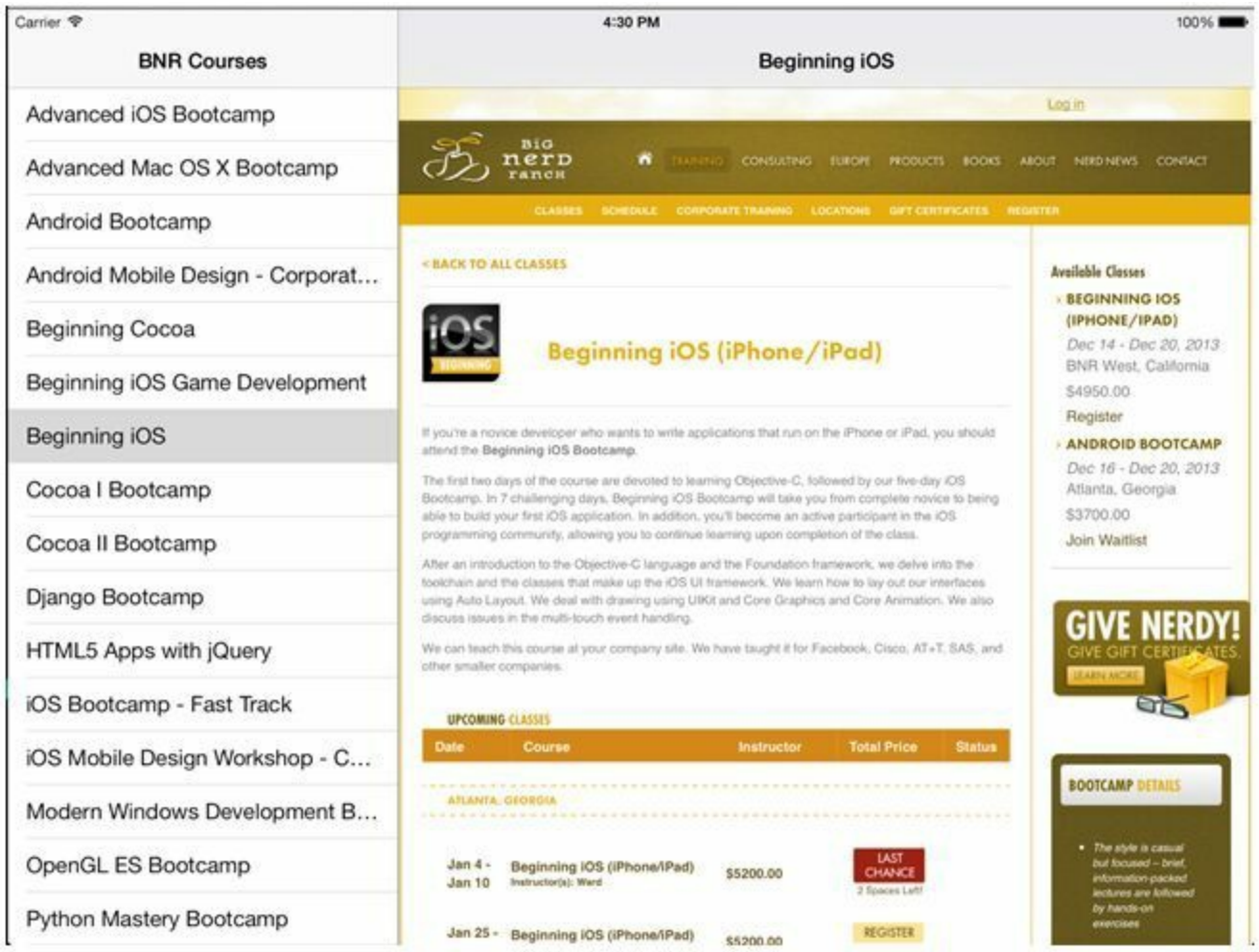


图22-1 使用UISplitViewController的Nerdfeed

22.1 在Nerdfeed中使用UISplitViewController

前面已介绍过UINavigationController和UITabBarController。和这些视图控制器类似，当初始化UISplitViewController时，也需要传入一组视图控制器。不同的是，UISplitViewController对象只能包含两个视图控制器：一个主视图控制器和一个从视图控制器。某个视图控制器是主还是从，是由该对象在数组中的位置决定的。数组中的第一项是主视图控制器，第二项是从视图控制器。

用Xcode打开Nerdfeed.xcodeproj，然后在项目导航面板中选中BNRAppDelegate.m。

更新application:didFinishLaunchingWithOptions:，检查当前运行的设备是否是iPad，是就创建UISplitViewController对象。iPhone不支持UISplitViewController，如果当前的设备是iPhone，创建UISplitViewController对象则会导致应用抛出异常。代码如下：

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
self.window
= [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
BNRCoursesViewController *lvc = [[BNRCoursesViewController alloc]
initWithStyle:UITableViewStylePlain];
UINavigationController *masterNav =
[[UINavigationController alloc] initWithRootViewController:lvc];
BNRWebViewController *wvc = [[BNRWebViewController alloc] init];
lvc.webViewController = wvc;
~~self.window.rootViewController = masterNav;~~
// 检查当前设备是否是iPad
if ([[UIDevice currentDevice] userInterfaceIdiom] ==
UIUserInterfaceIdiomPad) {
// 必须将webViewController包含在导航视图控制器中，稍后会解释原因
UINavigationController *detailNav =
```

```
[[UINavigationController alloc] initWithRootViewController:wvc];

UISplitViewController *svc = [[UISplitViewController alloc] init];

// 将从视图控制器设置为UISplitViewController对象的委托对象

// 稍后会用到(暂时忽略Xcode对这行代码发出的警告信息)

svc.delegate = wvc;

svc.viewControllers = @[masterNav, detailNav];

// 将UISplitViewController对象设置为UIWindow对象的根视图控制器

self.window.rootViewController = svc;

} else {

// 对于非iPad设备，仍然使用导航视图控制器

self.window.rootViewController = masterNav;

}

self.window.backgroundColor = [UIColor whiteColor];

[self.window makeKeyAndVisible];

return YES;

}
```

这段代码将创建UISplitViewController对象的代码放入了if子句，这是为将Nerdfeed修改为通用应用做准备。基于同样的原因，读者现在应该明白为什么要在UIApplication的委托中创建BNRWebViewController对象，而不是使用常见的模式(由UINavigationController对象的当前视图控制器创建下一个需要压入UINavigationController栈的对象)。创建UISplitViewController对象时，必需设置主、从视图控制器，缺一不可。图22-2显示的是Nerdfeed的UISplitViewController对象图。

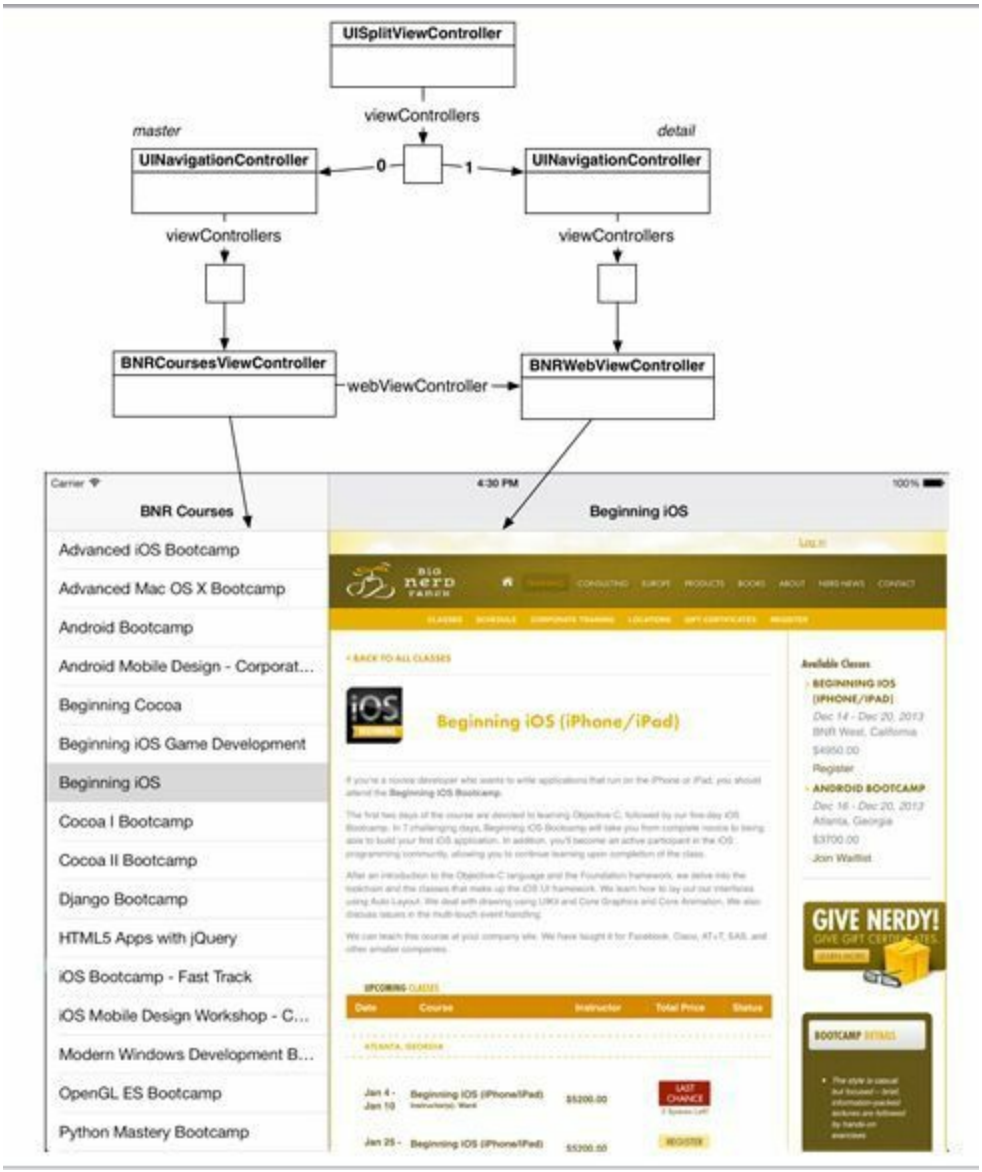


图22-2 UISplitViewController对象图

构建并运行应用，如果设备处于竖排模式，应用界面将没有任何内容；将设备旋转至横排模式，就可以在屏幕上看到两个视图控制器。下一节就会解决这个问题，让Nerdfeed在任何方向都可以正确显示。

如果现在按下UITableView对象中的某个表格行，Nerdfeed并不会在右侧面板（从面板）处显示BNRWebViewController对象，而是会将BNRWebViewController对象压入位于左侧面板（主面板）的UINavigationController栈，以替换掉BNRCoursesViewController对象。要解决该问题，需要在用户按下某个表格行时，检查BNRCoursesViewController对象是否属于某个UISplitViewController对象。如果是，就要执行不同的逻辑。

向某个UIViewController对象发送splitViewController消息，可以得到一个指针，并指向该对象所属的UISplitViewController对象。如果UIViewController对象不属于任何UISplitViewController对象，那么splitViewController方法会返回nil（见图22-3）。实际情况要稍微复杂一点，如果某个视图控制器A是某个UISplitViewController对象B的主/从对象之一，对象A的splitViewController方法就会返回对象B。如果某个视图控制器C不是对象B的主/从对象之一，但是对象A包含对象C，那么对象C的splitViewController方法也会返回对象B（本章的BNRCoursesViewController和BNRWebViewController都属于后一种情况）。

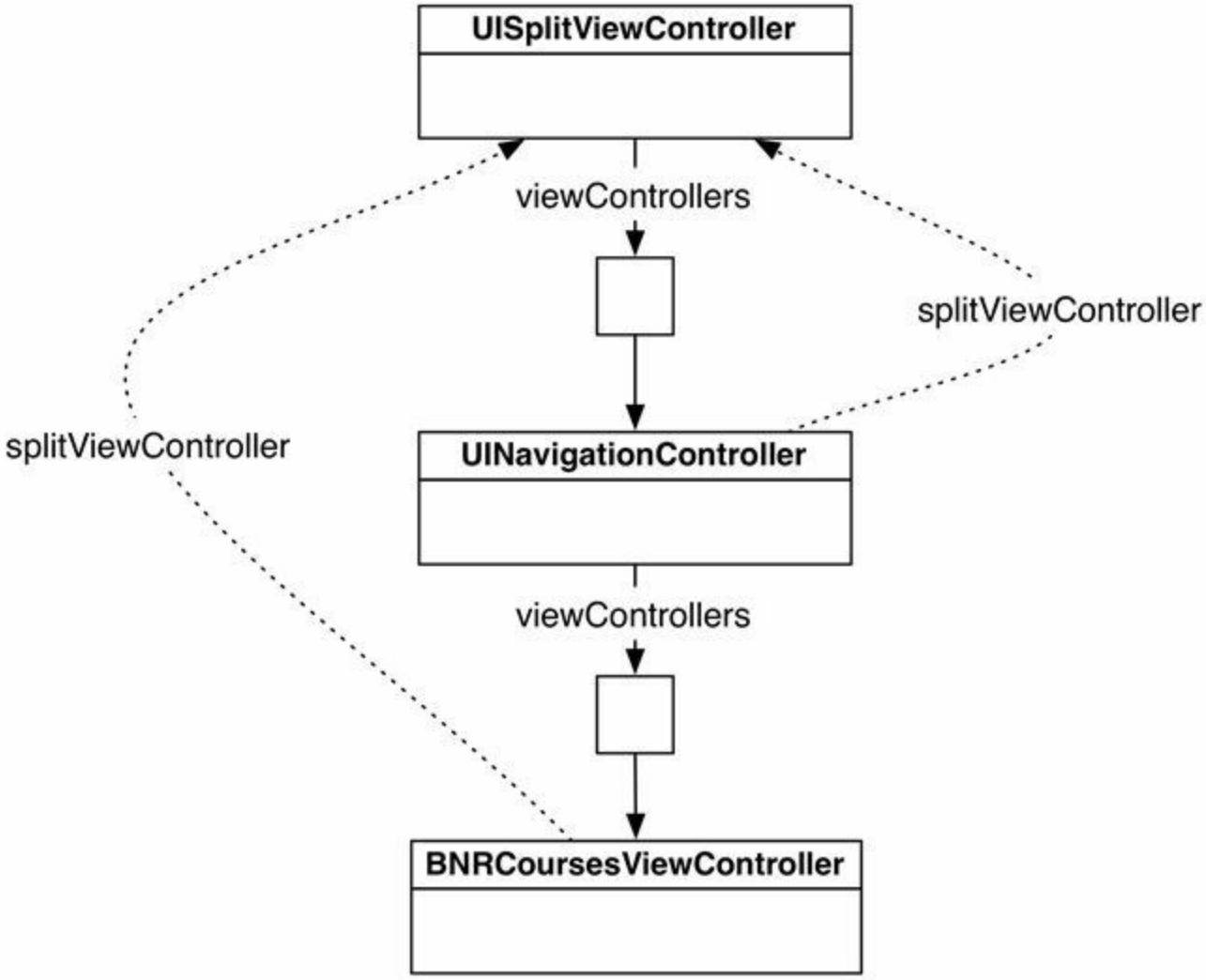


图22-3 UIViewController的splitViewController属性

更新BNRCoursesViewController.m中的tableView:didSelectRowAtIndexPath:，在该方法顶部，在将BNRWebViewController对象压入UINavigationController栈前，检查当前的BNRCoursesViewController对象是否属于某个UISplitViewController对象。代码如下：

```
-（void）tableView:（UITableView *）tableView
didSelectRowAtIndexPath:（NSIndexPath *）indexPath
{
NSDictionary *course = self.courses[indexPath.row];
NSURL *URL = [NSURL URLWithString:course[@“url”]];
self.webViewController.title = course[@“title”];
self.webViewController.URL = URL;
[self.navigationController pushViewController:self.webViewController
```

~~animated:YES];~~

if（！self.splitViewController）{

[self.navigationController pushViewController:self.webViewController

animated:YES];

}

}

如果BNRCoursesViewController对象不属于任何UISplitViewController对象，就可以认为当前的设备不是iPad，应该将BNRWebViewController对象压入UINavigation- Controller栈。如果BNRCoursesViewController对象属于某个UISplitViewController对象，就应该由这个UISplitViewController对象负责显示BNRWebViewController对象。

构建并运行应用，转动设备至横屏模式，按下UITableView对象中的某个表格行，Nerdfeed应该会在右侧面板载入相应的课程页面。

22.2 在竖排模式下显示主视图控制器

在竖排模式下，UISplitViewController对象不会显示主视图控制器。最好能通过某种途径、不用转动设备就能显示主视图控制器，进而可以通过UITableView对象选择课程。UISplitViewController对此有内建的处理机制：应用可以通过其委托方法得到一个UIBarButtonItem对象。按下这个按钮，应用会显示一个包含相应主视图控制器的UIPopoverController对象（见图22-4）。

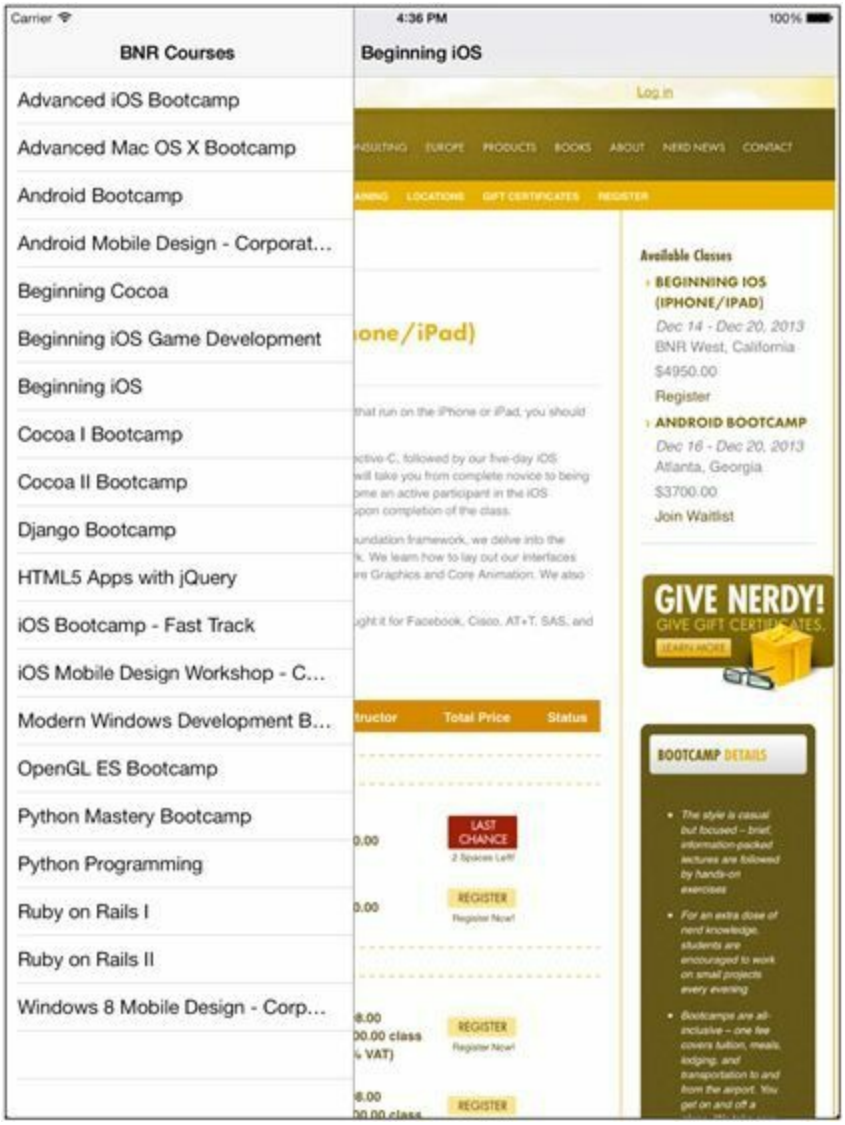


图22-4 UIPopoverController对象中的主视图控制器

当Nerdfeed将某个视图控制器设置为UISplitViewController的从视图控制器时，会将该视图控制器设置为UISplitViewController的委托。当设备转动至竖屏模式时，这个视图控制器会收到相应的委托消息，并得到一个指向特定UIBarButtonItem对象的指针。

在BNRWebViewController.h中，将BNRWebViewController声明为遵守UISplitViewControllerDelegate协议，代码如下：

```
@interface BNRWebViewController : UIViewController

<UISplitViewControllerDelegate>
```

构建并运行应用，Nerdfeed的运行结果应该和之前的一样，且Xcode不会再显示警告信息。

在BNRWebViewController.m中实现以下委托方法，将传入的UIBarButtonItem对象加入BNRWebViewController对象的navigationItem，代码如下：

```
- (void)splitViewController:(UISplitViewController *)svc

willHideViewController:(UIViewController *)aViewController

withBarButtonItem:(UIBarButtonItem *)barButtonItem

forPopoverController:(UIPopoverController *)pc

{

// 如果某个UIBarButtonItem对象没有标题，该对象就不会有任何显示

barButtonItem.title = @“Courses”;

// 将传入的UIBarButtonItem对象设置为navigationItem的左侧按钮

self.navigationItem.leftBarButtonItem = barButtonItem;

}
```

这段代码为UIBarButtonItem对象设置了标题。如果某个UIBarButtonItem对象没有标题，该对象就不会有任何显示（如果主视图控制器的navigationItem的title属性不是空字符串，UINavigationController对象就会自动将该字符串作为标题赋给相应的UIBarButtonItem对象）。

构建并运行应用。旋转设备至竖屏模式，Nerdfeed应该会在导航条左侧显示一个UIBarButtonItem对象。按下这个按钮，Nerdfeed应该会显示包含主视图控制器的UIPopoverController对象。

之前要求读者必须将从视图控制器放置在UINavigationController中，目的是能在导航条上显示上述UIBarButtonItem对象。虽然无须通过UINavigationController就能将某个视图控制器加入UISplitViewController对象，但是使用UINavigationController可以很方便地将UIBarButtonItem对象加入导航条。否则还需要先创建UINavigationBar或UIToolbar，加入UIBarButtonItem对象，然后将新创建的UINavigationBar或UIToolbar设置为BNRWebViewController视图的子视图。

Courses按钮还有一个问题：当用户将设备转回横屏模式时，按钮不会消失。为了解决该问题，委托对象需要实现另一个UISplitViewControllerDelegate方法。在BNRWeb-ViewController.m中实现该方法。代码如下：

```
- (void)splitViewController:(UISplitViewController *)svc

willShowViewController:(UIViewController *)aViewController
```

```
invalidatingBarButtonItem:(UIBarButtonItem *)barButtonItem

{

// 移除navigationItem的左侧按钮

// 确认该按钮是需要删除的那个(虽然根据现有的代码可以确定不会有错)

if (barButtonItem == self.navigationItem.leftBarButtonItem) {

self.navigationItem.leftBarButtonItem = nil;

}

}
```

构建并运行应用。转动设备，在竖屏模式和横屏模式之间切换，Nerdfeed应该可以正确显示和隐藏Courses按钮。

22.3 将Nerdfeed改为通用应用

刚开始创建Nerdfeed时，我们的目标是开发一款只针对iPad的应用。下面要将Nerdfeed改为通用应用。选中位于项目导航面板顶部的Nerdfeed条目，在编辑器区域选中Nerdfeed目标，然后单击General标签，在编辑区域找到标题为Devices的弹出菜单，选择Universal（见图22-5）。

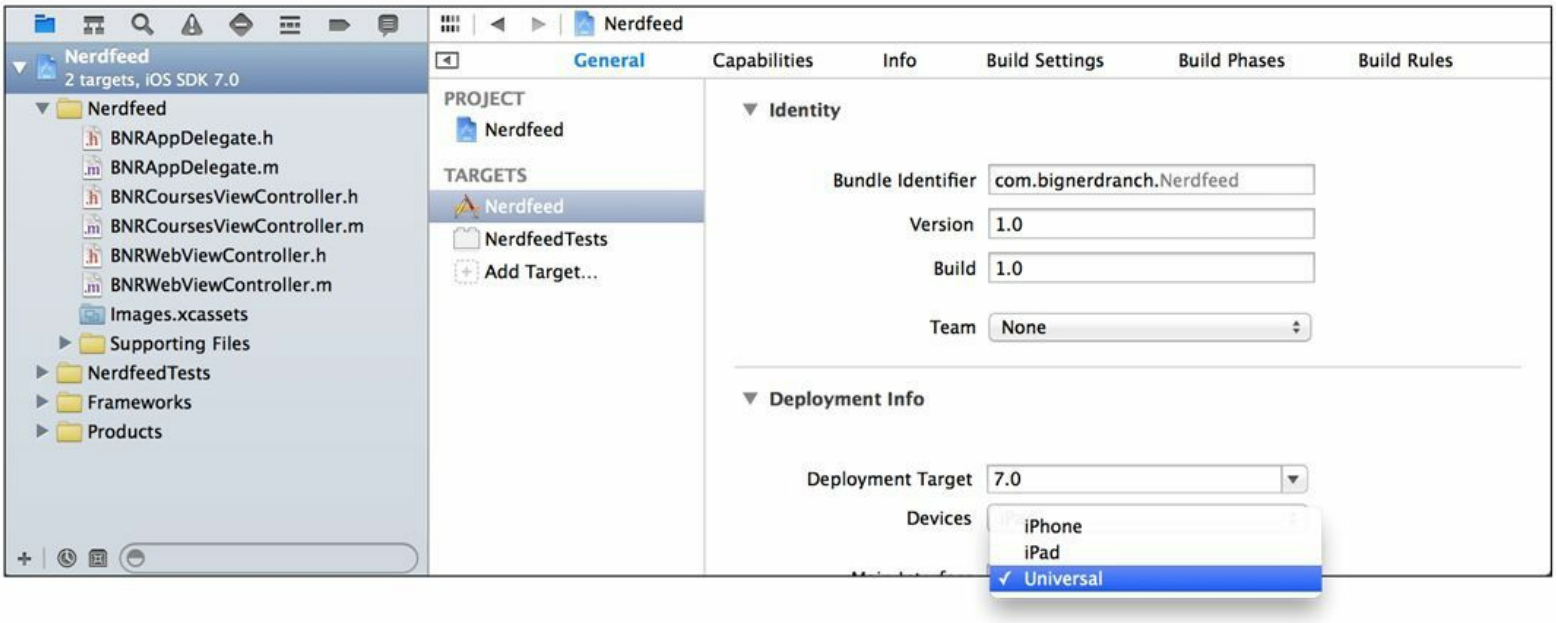


图22-5 将Nerdfeed改为通用应用

现在Nerdfeed已经是通用应用了，可以在iPhone中正确运行。针对不同类型的模拟器构建并运行应用，检查运行结果是否正确（见图22-6）。

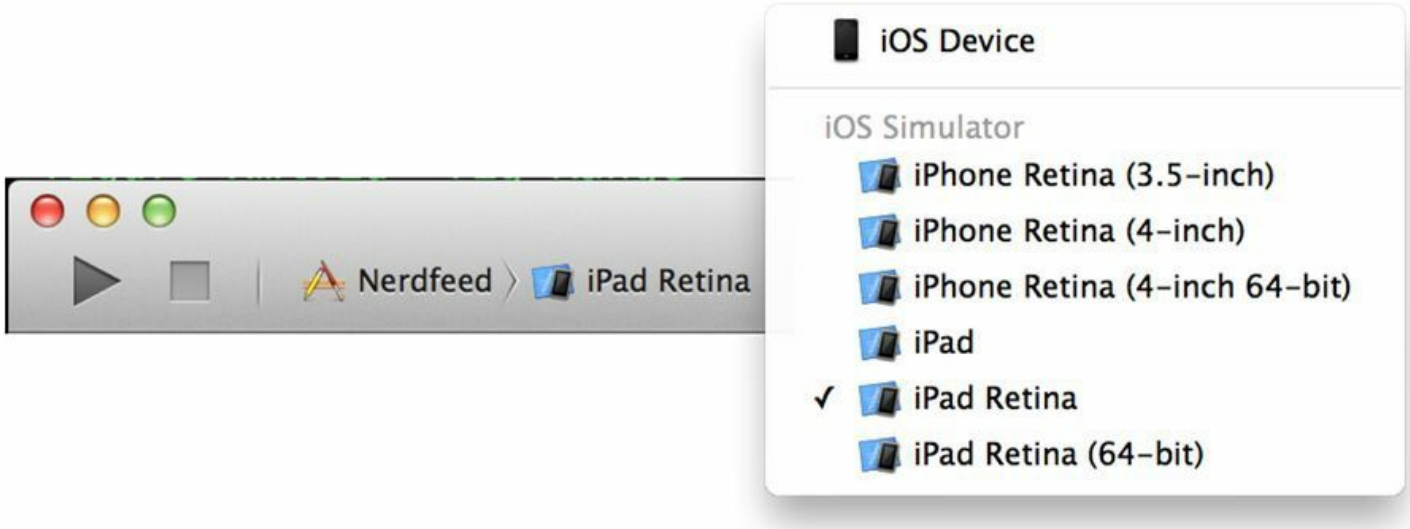


图22-6 改用iPhone模拟器

Nerdfeed的通用化过程很简单，原因有两个（记住这两点会对读者将来自己编写应用有帮助）。

•本章在编写Nerdfeed时，就顾及了若干类在不同设备上的差异性。例如，考虑到UISplitViewController不能在iPhone或iPod touch上使用，就针对这些设备提供了第二套界面。通常情况下，读者在使用iOS SDK提供的类时，应该参阅相关文档的讨论部分。这部分文档会说明能否在特定的设备上使用相应的类和方法。

•Nerdfeed是一个相对简单的应用。越早开始考虑通用化的问题，实施时就会越容易。随着应用的开发，会有越来越多的细节和通用化过程产生牵连。编写代码时就把通用化的问题考虑进去，会比将来基于现有代码修改容易得多。定位实现细节是件很困难的事情，有可能引发新的问题并破坏当前能够正常工作的代码。

第23章 Core Data

iOS应用可以通过多种机制保存和读取数据。在决定使用哪种机制前，第一个要考虑的问题通常是“本地还是远程？”如果要将数据保存在远程服务器中，那么通常会选用Web服务。如果要将数据保存在当前设备的文件系统中，那么通常会选用固化或Core Data。

目前，Homepwner为了将BNRItem对象存入文件系统，使用的是固化机制。使用固化的最大缺点是数据必须“整存整取”：要访问固化文件中的任何数据，必须先解固整个文件；要保存数据的任何改动，必须重写整个文件。Core Data没有这样的缺点。Core Data可以只读取已存对象中的一小部分。如果取出的对象发生了变化，也只要更新相应部分的文件。如果某个应用要在文件系统和RAM之间传送大量模型对象，那么Core Data的这种增量读取、更新、删除和插入的特性可以大幅提高性能。

23.1 对象-关系映射

Core Data框架提供的是对象-关系映射(object-relational mapping)功能。也就是说，Core Data可以将Objective-C对象转化成数据，并能将这些数据保存在SQLite数据库文件中。此外，Core Data也可以将保存后的数据还原成Objective-C对象。SQLite是一种关系数据库，可以通过单个文件保存所有的数据(从技术上讲，SQLite是一套管理数据库文件的代码库。本书中的SQLite一词，不仅指该代码库，还指相应的数据库文件)。需要注意的是，与Oracle、MySQL或SQL Server不同，SQLite不是一套全功能的数据库服务器，没有自己的服务进程，所以也没有客户端(client)可以通过网络连接SQLite。

借助Core Data，读者无须了解SQL就能读取和保存关系数据库中的数据。即便是这样，还是应该对关系数据库的工作原理有一定的了解。本章将介绍若干关系数据库的基本知识，并用Core Data重写BNRItemStore的数据保存功能。

23.2 用Core Data重写BNRItemStore的数据保存功能

目前，Homepwner是通过固化存取数据的。当模型对象的数量较少时（比如少于1000个），使用固化不会有问题。但是随着模型对象的增多，固化的整存整取特性会产生效率问题，所以需要改用一种能够增量存取、增量更新数据的机制。Core Data可以胜任这项任务。

模型文件

关系数据库有一个名为表格（table）的概念。一张表格代表一种类型的事物，可以是人、信用卡购物记录或房地产列表。每张表格可以有很多列，用于保存相应事物的特定信息。例如，代表人的表格可以有不同的列，分别代表姓名、生日和身高。此外，表中的一行代表一个人。

关系数据库的这种组织关系可以很好地“翻译”至Objective-C。表格对应Objective-C类；表格的列对应类的属性；表格的行对应类的对象。Core Data的作用就是在关系模型和对象模型之间来回移动数据（见图23-1）。

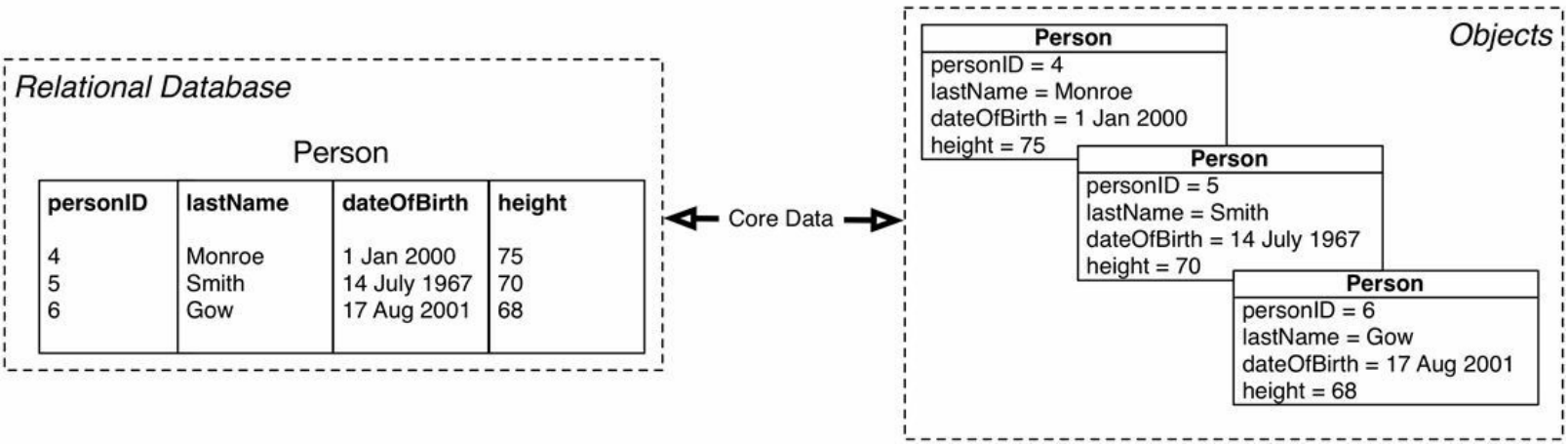


图23-1 Core Data的作用

针对上述概念，Core Data使用的是另一套术语。在Core Data中，表格/类称为实体（entity），列/属性称为实体属性（attribute）。使用Core Data的模型文件可以描述特定的实体和相应的实体属性。下面用模型文件来描述BNRItem实体，以及相应的实体属性，例如itemName、serialNumber和valueInDollars。

打开Homepwner.xcodeproj。选择File菜单中的New菜单项，然后选择New File…选中窗口左侧iOS部分的Core Data，然后选中窗口右侧的Data Model并单击Next按钮，最后将文件命名为Homepwner（后缀名使用默认的），如图23-2所示。

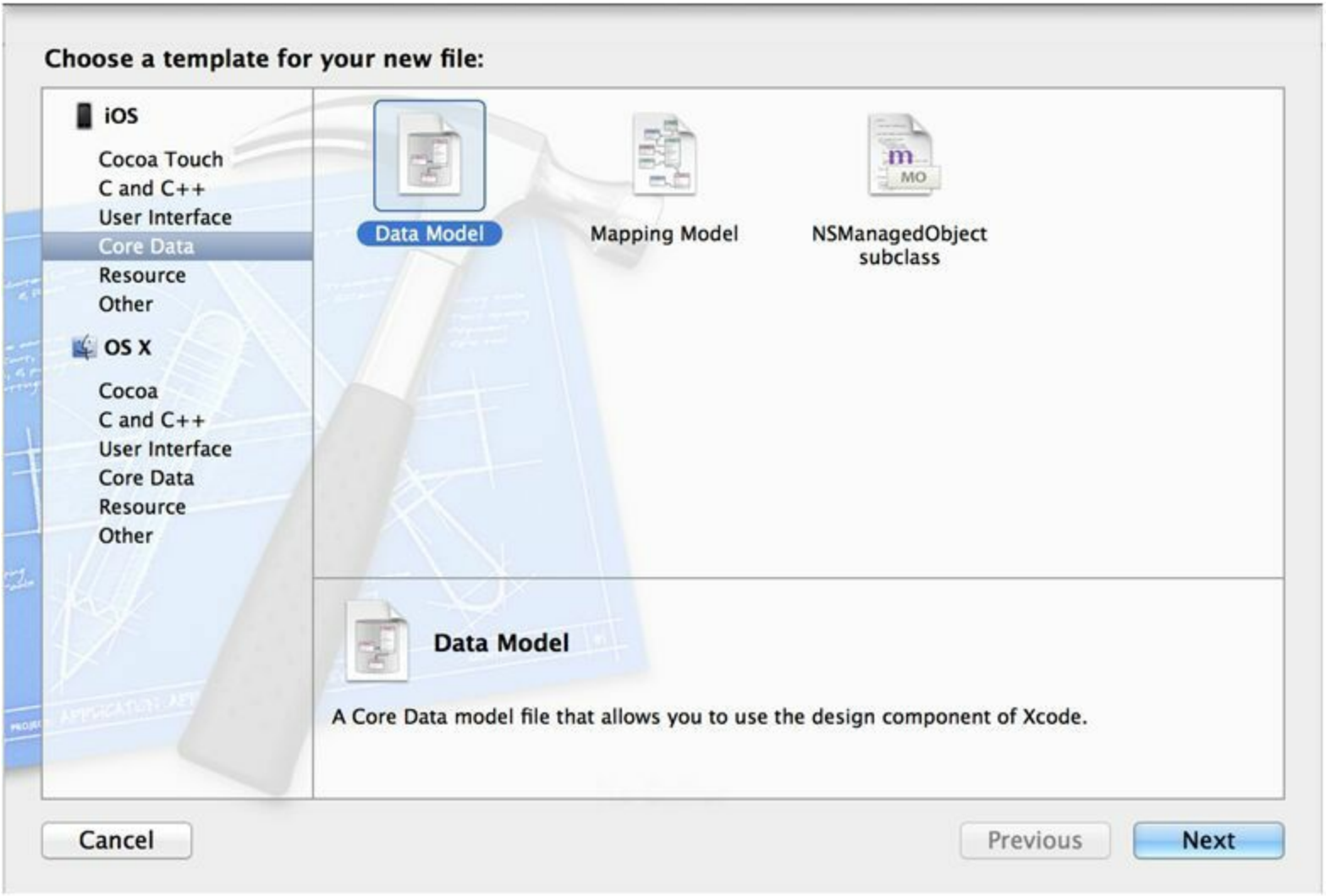


图23-2 创建模型文件

Xcode会将新创建的Homepwner.xcdatamodeld文件加入项目。在项目导航面板中选中该文件，Xcode会在编辑器区域显示相应的用户界面，通过该界面可编辑Core Data模型文件。

单击位于编辑器区域左下角的Add Entity（增加实体）按钮，Xcode会为模型文件增加一个新的实体，并将该实体加入左侧列表的ENTITIES区域。双击新增加的实体，将实体名改为BNRItem（见图23-3）。

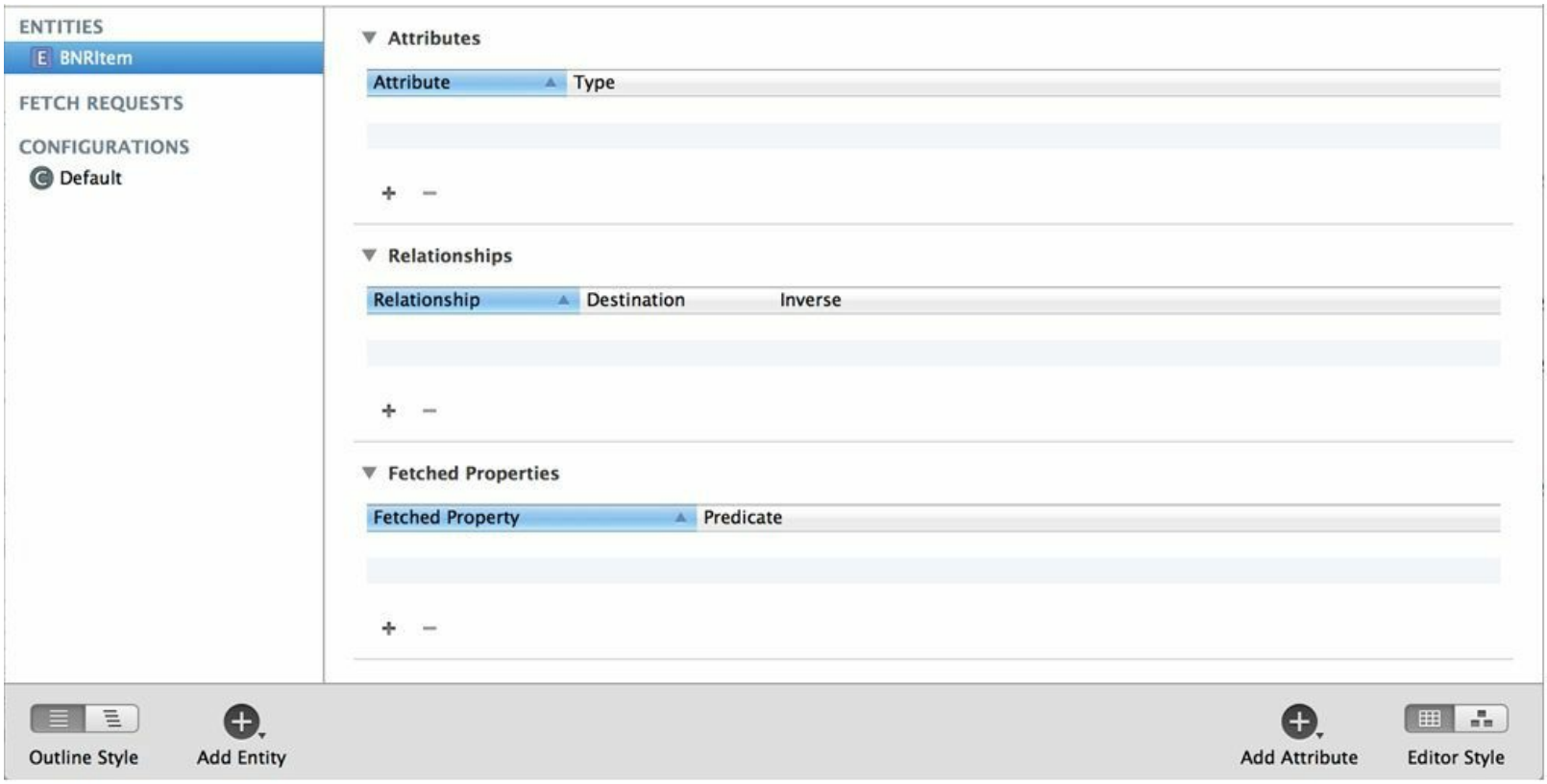


图23-3 创建BNRItem实体

下面为BNRItem实体增加实体属性，这些实体属性将成为BNRItem类的属性。单击Attributes区域下方的＋按钮可以加入新的实体属性。根据以下列表，增加实体属性并设置相应的Attribute（实体属性名）和Type（实体属性类型）。

•itemName, String（字符串）。

•serialNumber, String（字符串）。

•valueInDollars, Integer 32（32位整数）。

•dateCreated, Date（日期）。

•itemKey, String（字符串）。

•thumbnail, Transformable（可变类型）。thumbnail的类型是UIImage，实体属性没有与之对应的类型。稍后就会介绍Transformable。

下面为BNRItem实体再增加一个实体属性。在Homepwner中，用户可以通过移动UITableView对象中的UITableViewCell对象，重新排列相应BNRItem对象的顺序。使用固化保存包含BNRItem对象的数组时，默认就能保留排列顺序。但是在使用Core Data时，因为关系数据库的表格默认不会保留行的排列位置，所以要针对每个BNRItem对象保存一个额外的位置信息。然后在获取某一组行时，根据这个实体属性（表格列）来排序。此外，也可以根据其他实体属性来排序（例如，获取所有的BNREmployee对象时可以根据lastName排序）。

为了保留BNRItem对象在UITableView对象中的位置，下面创建一个名为orderingValue的实体属性，用来记录BNRItem对象在UITableView对象中的位置。当要获取一组BNRItem对象时，

可以要求Core Data(数据库)按实体属性orderingValue进行排序;当BNRItem对象在UITableView对象中的位置发生变化时,需要更新相应对象的orderingValue。下面创建该实体属性:名称是orderingValue,类型是Double。

Core Data只能存储有限的几种数据类型,并不能存储UIImage对象。因此只能将thumbnail声明为Core Data可以存储的transformable类型,Core Data会在存储或恢复transformable类型的实体属性时首先将其转换为NSData,然后再存入文件系统或恢复为Objective-C对象。为了向Core Data描述转换过程,需要创建NSValueTransformer的子类。

创建一个NSValueTransformer的子类BNRImageTransformer,打开BNRImage- Transformer.m,覆盖以下方法,实现UIImage和NSData对象之间的转换:

```
@implementation BNRImageTransformer

+ (Class)transformedValueClass

{

return [NSData class];

}

- (id)transformedValue:(id)value

{

if (！value) {

return nil;

}

if ([value isKindOfClass:[NSData class]]) {

return value;

}

return UIImagePNGRepresentation(value);

}

- (id)reverseTransformedValue:(id)value

{

return [UIImage imageWithData:value];
```

```
}

@end
```

BNRImageTransformer的实现代码很容易理解。transformedValueClass是类方法，用于声明transformedValue:方法的返回类型（BNRImageTransformer的转换结果类型）；Core Data在存储transformable类型的实体属性时会调用transformedValue:方法，将其转换为可以存储的类型。对于thumbnail，transformedValue:的参数类型是UIImage，返回值类型是NSData；Core Data在恢复thumbnail时会调用reverseTransformedValue:，根据文件系统中存储的NSData创建UIImage对象。有了BNRImageTransformer后，Core Data还要知道如何使用BNRImageTransformer来处理thumbnail。

打开Homepwner.xcdatamodeld，选择BNRItem实体。选中Attributes区域中的thumbnail，单击检视选择面板中的按钮，打开数据模型检视面板（data model inspector），在第二个标题为Name的文本框中填入BNRImageTransformer（见图23-4）。

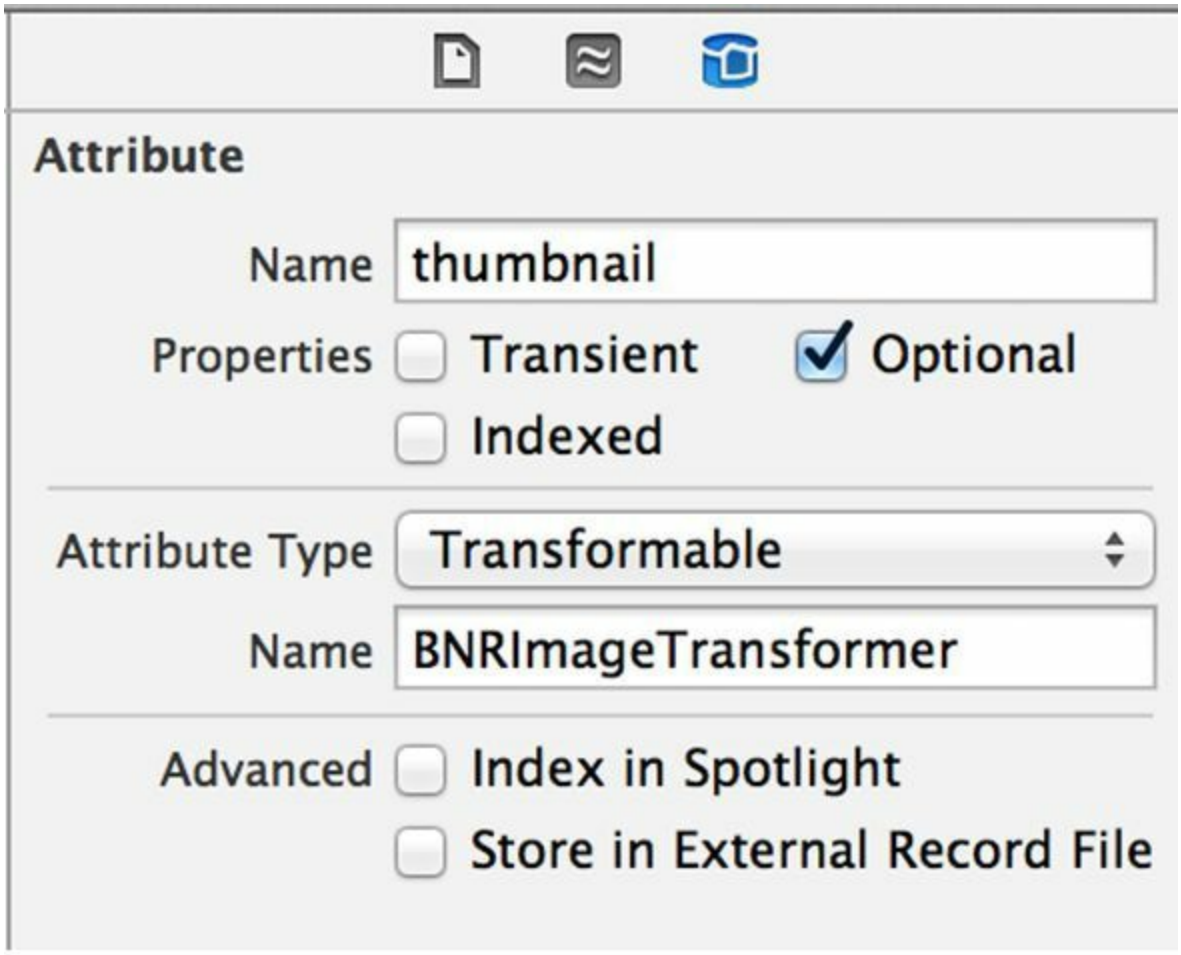


图23-4 为BNRItem实体增加并设置实体属性

完成上述操作后，就可以通过新创建的模型文件存取BNRItem对象了，但这还不够。使用Core Data的另一个好处是能够在实体之间建立关联。下面要增加一个名为BNRAssetType的实体，用来描述BNRItem对象的分类。例如，可以将一幅画归在艺术类。BNRAssetType和BNRItem一样，应该在同一个模型文件中创建，和BNRAssetType相对应的表格行也会在运行时被映射为相应的Objective-C对象。

在Homepwner.xcdatamodeld中增加一个名为BNRAssetType的实体，然后为这个新增加的实体创建一个名为label的实体属性，类型为String，用来代表分类的名称（见图23-5）。

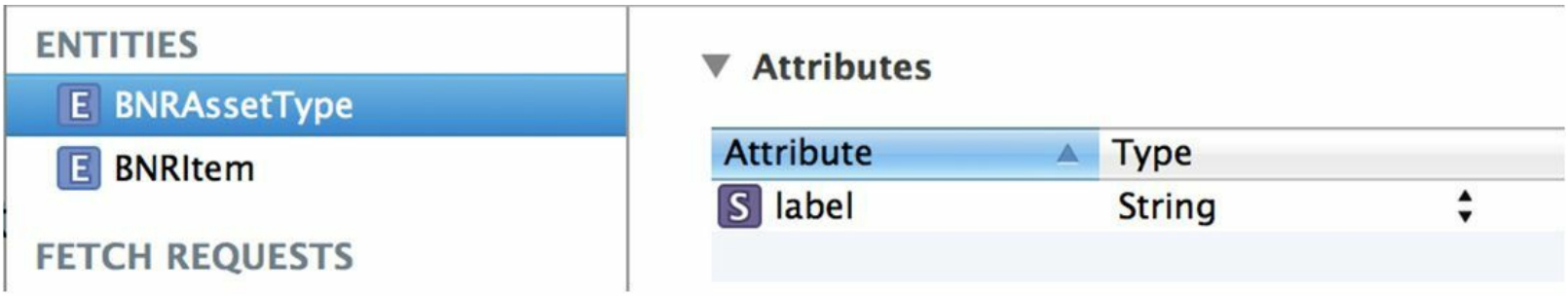


图23-5 创建BNRAssetType实体

下面为BNRAssetType和BNRItem建立关系（relationship）（Core Data会将实体间的关系表示为对象之间的指针）。Core Data支持两类关系：一对一（to-one）关系和一对多（to-many）关系。当某个实体拥有某种一对一关系时，该实体的对象会有一个指针指向位于相应关系另一端的实体对象。以BNRItem为例，它会有一个指向BNRAssetType实体的一对一关系，因此BNRItem对象会拥有一个指向BNRAssetType对象的指针。

当某个实体拥有某种一对多关系时，该实体的对象会拥有一个指向NSSet对象的指针，该对象会包含和相应一对多关系有关的实体对象。以BNRAssetType为例，因为可以有多个BNRItem对象属于同一个类型（BNRAssetType），所以BNRAssetType实体应该有一个指向BNRItem实体的一对多关系。

为BNRItem实体和BNRAssetType实体创建上述关系后，就可以向某个BNRAssetType对象查询并返回一组属于该类型的BNRItem对象。此外，也可以向某个BNRItem对象查询其所属的类型（见图23-6）。

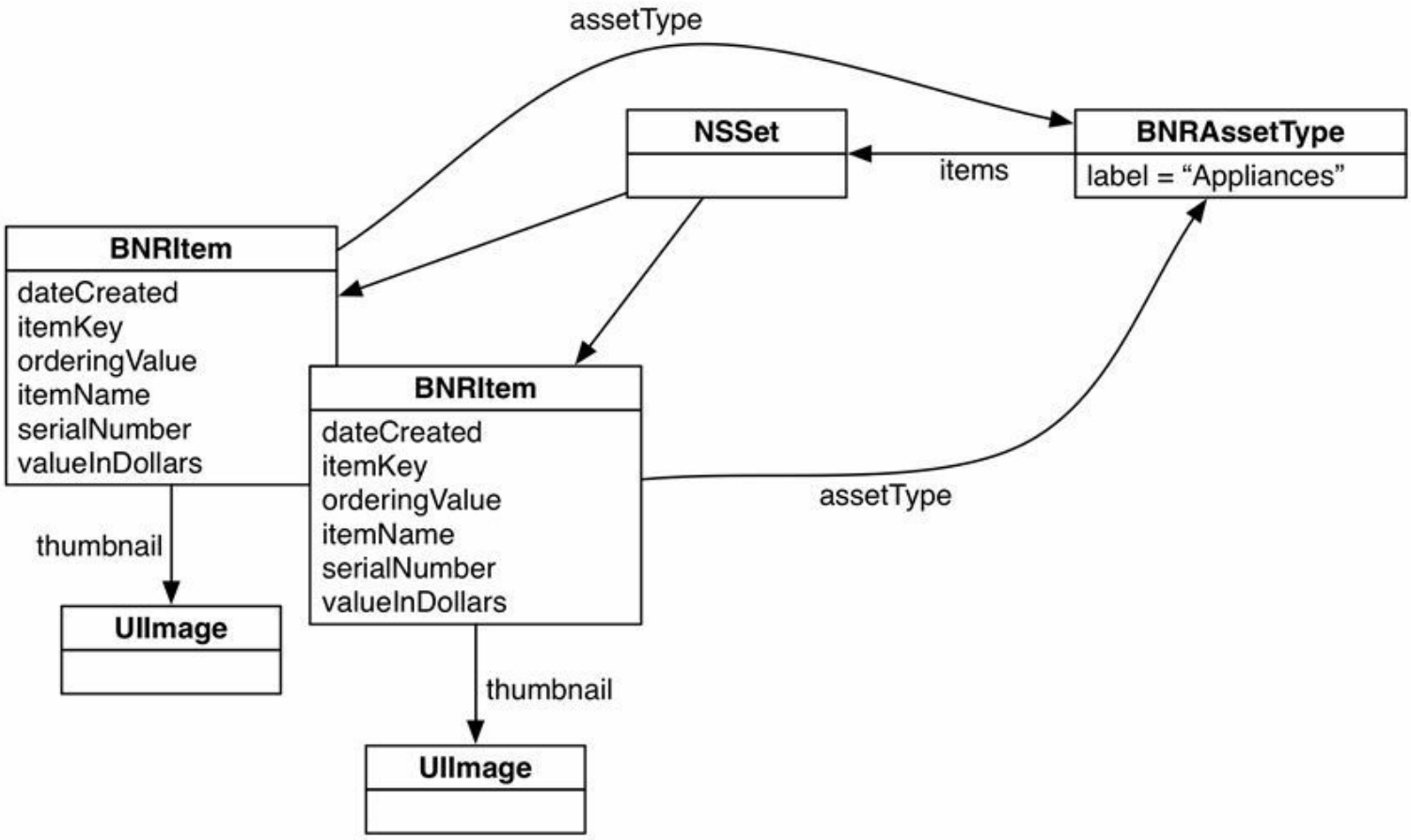


图23-6 Homepwner中的实体关系

下面要在模型文件中增加上述关系。选中BNRAssetType实体，单击Relationships（关系）列表下方的＋按钮。在Relationship列中，将关系命名为items。然后在Destination（目标）列中选择BNRItem。最后在数据模型检视面板中点击标题为Type的下拉菜单，由To One改为To Many（见图23-7）。

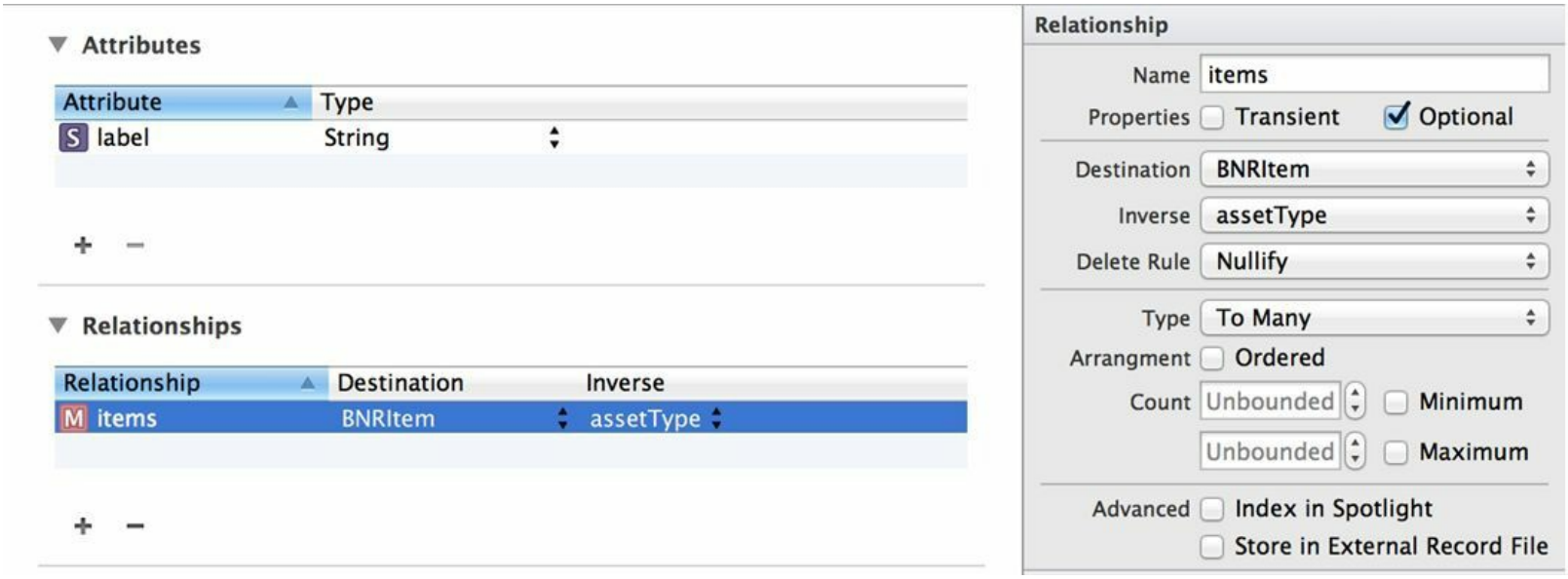


图23-7 创建items关系

选中BNRItem实体，增加一个名为assetType的关系，然后在Destination列中选择BNRAssetType。在Inverse（反向）列中选择items（见图23-8）。

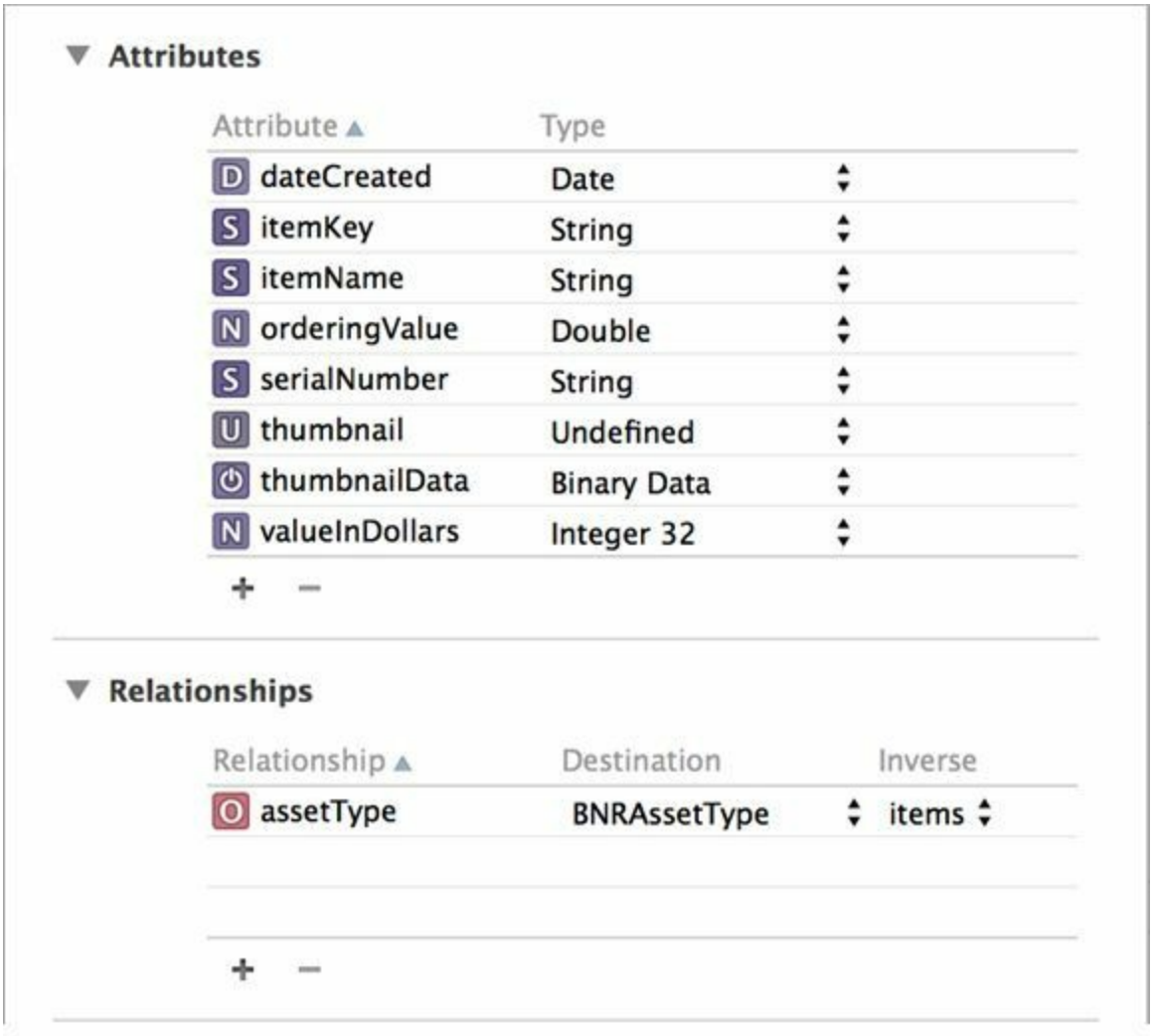


图23-8 创建assetType关系

NSManagedObject与NSManagedObject子类

通过Core Data取回(fetch)的对象，默认情况下都是NSManagedObject对象。NSManagedObject是NSObject的子类，也是Core Data的重要组成部分。NSManagedObject对象的工作模式有点类似字典对象：它会根据相应实体的每一个property(属性或关系)，保存一个键-值对。

NSManagedObject对象不仅仅是数据容器。除了保存数据，还可以通过创建NSManagedObject子类让自定义的NSManagedObject对象完成其他任务。当对应某个实体的NSManagedObject对象要执行自定义的任务时，必须先创建相应的NSManagedObject子类，然后在模型文件中修改这个实体，将代表该实体的类从默认的NSManagedObject改为新创建的NSManagedObject子类。

选中BNRItem实体，打开数据模型检视面板，将Class文本框的内容修改为BNRItem，如图23-9所示。这样，当Homepwner通过Core Data取回BNRItem实体时，相应的对象类型将是BNRItem(BNRAssetType实体的类型仍是NSManagedObject)。

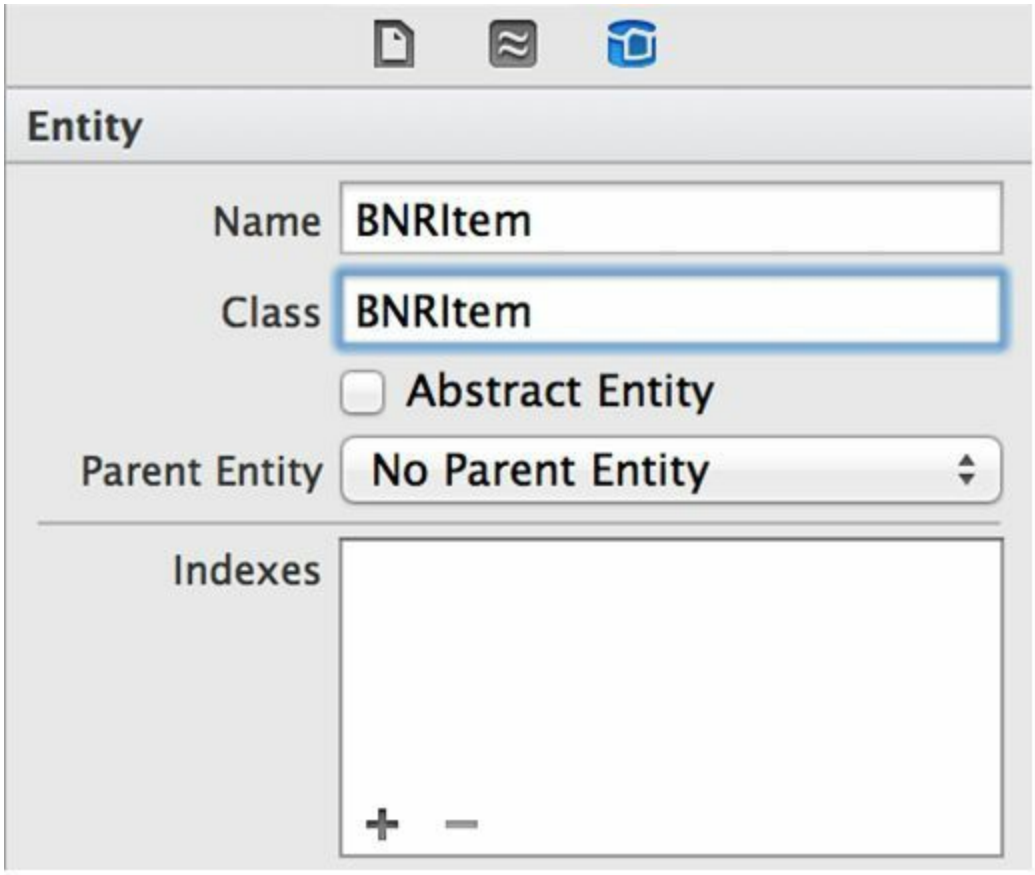


图23-9 修改实体的类

在为BNRItem实体创建NSManagedObject子类时，需要先解决一个问题：Homepwner项目已经有了一个名为BNRItem的类，并且这个类不是继承自NSManagedObject。如果直接修改现有的BNRItem类，则需要做很多改动才能将其父类改为NSManagedObject。最简单的解决方案是移除现有的BNRItem类，然后通过Xcode自动生成针对BNRItem实体的BNRItem类，最后将之前在BNRItem类中实现的方法拷贝至新的BNRItem类。

在Finder中，将BNRItem.h和BNRItem.m拖曳至桌面留作备份。然后在项目导航面板中移除这两个文件（在Finder中移动这两个文件后，Xcode会因为找不到相应的文件而在项目导航面板中将这两个文件显示为红色）。

选中Homepwner.xcdatamodeld，选中BNRItem实体。选择File菜单中的New菜单项，然后选择New File…选中窗口左侧iOS部分的Core Data，然后选中窗口右侧的NSManagedObject subclass，单击Next按钮。Homepwner数据模型文件之前的选择框应该已经选中，如果没有选中，请选中并单击Next按钮。在下一个窗口，确保选中了BNRItem实体并再次单击Next按钮，最后单击Save按钮创建NSManagedObject子类。

Xcode会生成两个新文件：BNRItem.h和BNRItem.m。在BNRItem.h中，将thumbnail属性的类型改为UIImage指针，并加入两个曾在BNRItem类中实现过的方法。默认情况下，Xcode会将属性声明为对象，因此int类型的属性被声明为NSNumber。将orderingValue改为double，valueInDollars改为int。代码如下：

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

@import CoreData;
```

```
@interface BNRItem : NSManagedObject

@property (nonatomic, strong) NSDate * dateCreated;

@property (nonatomic, strong) NSString * itemKey;

@property (nonatomic, strong) NSString * itemName;

@property (nonatomic, strong) NSNumber * orderingValue;

@property (nonatomic) double orderingValue;

@property (nonatomic, strong) NSString * serialNumber;

@property (nonatomic, strong) id thumbnail;

@property (nonatomic, strong) UIImage *thumbnail;

@property (nonatomic, strong) NSData * thumbnailData;

@property (nonatomic, strong) NSNumber * valueInDollars;

@property (nonatomic) int valueInDollars;

@property (nonatomic, strong) NSManagedObject *assetType;

- (void)setThumbnailFromImage:(UIImage *)image;

@end
```

（Xcode可能会将strong属性生成为retain。两者的含义是相同的，在引入ARC之前，强引用属性使用retain表示，引入之后才改为使用strong。Xcode中用于生成NSManagedObject subclass的工具可能没有同步更新，因此在生成的代码中仍然使用旧的表示方法）。

将setThumbnailFromImage:从之前的BNRItem.m拷贝至新的BNRItem.m，代码如下：

```
- (void)setThumbnailFromImage:(UIImage *)image

{

CGSize origImageSize = image.size;

CGRect newRect = CGRectMake(0, 0, 40, 40);

float ratio = MAX(newRect.size.width / origImageSize.width,

newRect.size.height / origImageSize.height);
```

```
UIGraphicsBeginImageContextWithOptions(newRect.size, NO, 0.0);
UIBezierPath *path = [UIBezierPath bezierPathWithRoundedRect:newRect
cornerRadius:5.0];
[path addClip];
CGRect projectRect;
projectRect.size.width = ratio * origImageSize.width;
projectRect.size.height = ratio * origImageSize.height;
projectRect.origin.x = (newRect.size.width - projectRect.size.width) / 2.0;
projectRect.origin.y = (newRect.size.height - projectRect.size.height) / 2.0;
[image drawInRect:projectRect];
UIImage *smallImage = UIGraphicsGetImageFromCurrentImageContext();
self.thumbnail = smallImage;
UIGraphicsEndImageContext();
}
```

首次启动Homepwner时，肯定不会有已存的BNRItem对象和BNRAssetType对象。当用户创建新的BNRItem对象时，Homepwner会将新增加的对象加入数据库。当这些对象被加入数据库时，都会收到awakeFromInsert消息。所以，应该在BNRItem对象的awakeFromInsert方法中设置dateCreated和itemKey属性。在BNRItem.m中实现awakeFromInsert方法，代码如下：

```
-(void)awakeFromInsert
{
[super awakeFromInsert];
self.dateCreated = [NSDate date];
// 创建NSUUID对象，获取其UUID字符串
NSUUID *uuid = [[NSUUID alloc] init];
NSString *key = [uuid UUIDString];
self.itemKey = key;
```

}

这段代码的功能和前面BNRItem的指定初始化方法相同。构建应用，检查语法错误，先不要运行。

更新BNRItemStore

Core Data框架中的NSManagedObjectContext负责应用和数据库之间的交互工作。通过NSManagedObjectContext对象所使用的NSPersistentStoreCoordinator对象，可以指定文件路径并打开相应的SQLite数据库。NSPersistentStoreCoordinator对象需要配合某个模型文件才能工作（NSManagedObjectModel对象可以代表模型文件）。在Homepwner中，需要由BNRItemStore对象来使用上述的多个Core Data对象，以完成数据的存取工作。这些对象之间的关系如图23-10所示。

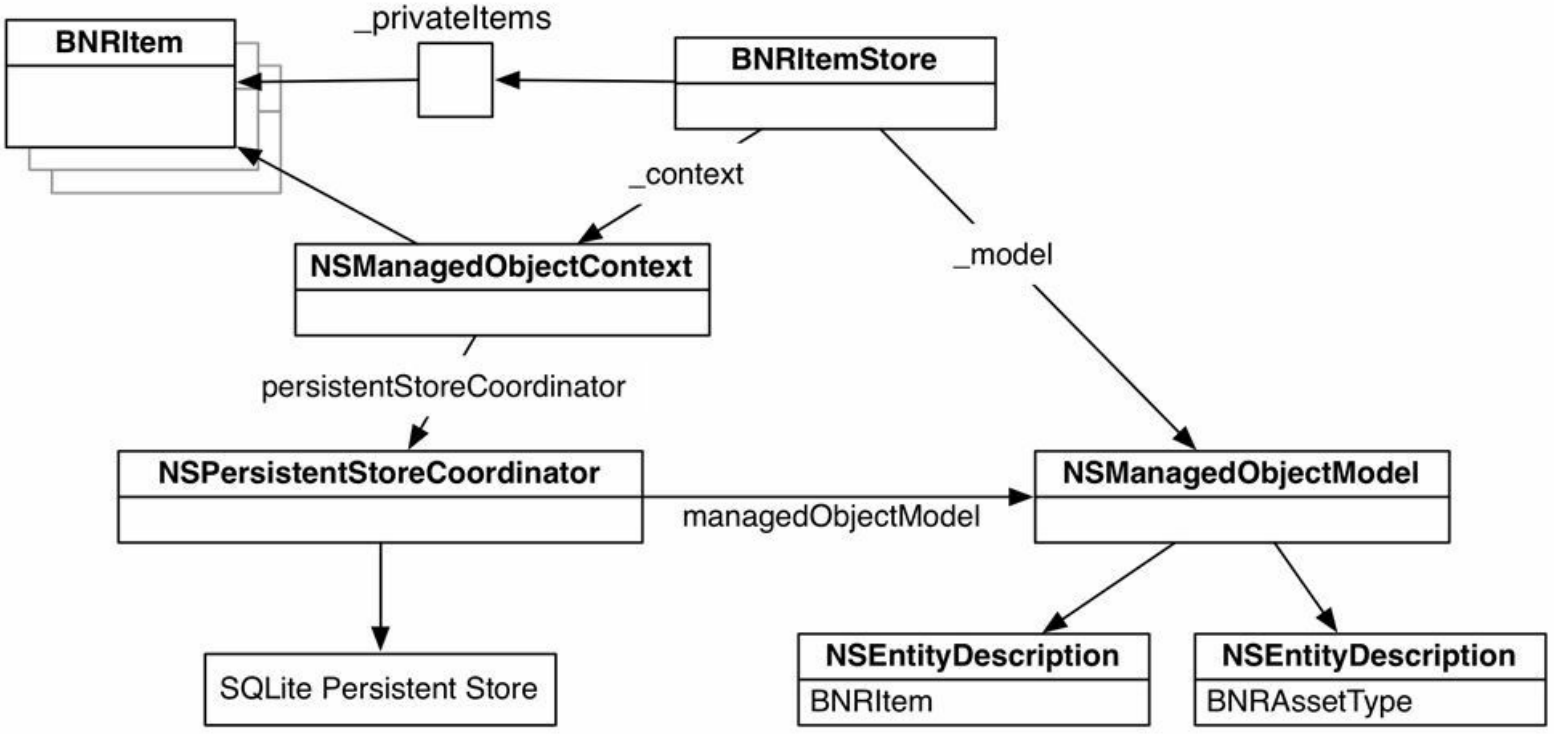


图23-10 BNRItemStore与NSManagedObjectContext

在BNRItemStore.m中导入Core Data框架并在类扩展中声明三个属性，代码如下：

```
@import CoreData;

@interface BNRItemStore ()

@property (nonatomic) NSMutableArray *privateItems;

@property (nonatomic, strong) NSMutableArray *allAssetTypes;
```

@property (nonatomic, strong) NSManagedObjectContext *context;

@property (nonatomic, strong) NSManagedObjectModel *model;

然后修改itemArchivePath方法，返回不同的路径，Core Data能够将数据保存至另一个文件，代码如下：

- (NSString *)itemArchivePath

{

NSArray *documentDirectories =

NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,

NSUserDomainMask,

YES);

// 从documentDirectories数组获取文档目录路径(该数组只包含一个对象)

NSString *documentDirectory = [documentDirectories firstObject];

~~return [documentDirectory stringByAppendingPathComponent:@"items.archive"];~~

return [documentDirectory stringByAppendingPathComponent:@"store.data"];

}

初始化BNRItemStore对象时，需要创建并设置相应的NSManagedObjectContext对象和NSPersistentStoreCoordinator对象。NSPersistentStoreCoordinator对象需要知道两件事情：实体信息(包括实体属性和关系)和存取数据的路径。为此，要先创建一个NSManagedObjectModel对象，保存源自Homepwner.xcdatamodeld的实体信息。然后用新创建的NSManagedObjectModel对象初始化NSPersistentStoreCoordinator对象。创建并设置NSPersistentStoreCoordinator对象后，就可以创建一个NSManagedObject- Context对象，并传入之前创建的NSPersistentStoreCoordinator对象(由该对象负责文件的存取)。

更新BNRItemStore.m中的initPrivate方法，代码如下：

- (instancetype)initPrivate

{

self = [super init];

if (self) {

~~NSString *path = self.itemArchivePath;~~

~~_privateItems = [NSKeyedUnarchiver unarchiveObjectWithFile:path];~~

~~if（！_privateItems）{~~

~~_privateItems = [[NSMutableArray alloc] init];~~

}

// 读取Homepwner.xcdatamodeld

_model = [NSManagedObjectModel mergedModelFromBundles:nil];

NSPersistentStoreCoordinator *psc =

[[NSPersistentStoreCoordinator alloc] initWithManagedObjectModel:_model];

// 设置SQLite文件路径

NSString *path = self.itemArchivePath;

NSURL *storeURL = [NSURL fileURLWithPath:path];

NSError *error = nil;

if（！[psc addPersistentStoreWithType:NSSQLiteStoreType

configuration:nil

URL:storeURL

options:nil

error:&error]）{

@throw [NSException exceptionWithName:@“OpenFailure”

reason:[error localizedDescription]

userInfo:nil];

}

// 创建NSManagedObjectContext对象

_context = [[NSManagedObjectContext alloc] init];

_context.persistentStoreCoordinator = psc;

```
}

return self;

}
```

修改前的BNRItemStore对象会通过固化将整个BNRItem数组写入文件，而修改后的BNRItemStore对象可以向NSManagedObjectContext对象发送save:消息，NSManagedObjectContext对象会根据上次保存后发生的变化更新store.data中的相应记录。修改BNRItemStore.m中的saveChanges方法，代码如下：

```
-（BOOL）saveChanges

{

~~NSString *path = [self itemArchivePath];~~

~~return [NSKeyedArchiver archiveRootObject:allItems~~

~~toFile:[self itemArchivePath]];~~

NSError *error;

BOOL successful = [self.context save:&error];

if（！successful）{

NSLog(@“Error saving: %@”, [error localizedDescription]）;

}

return successful;

}
```

saveChanges方法会在应用进入后台运行状态时被调用。

NSFetchRequest与NSPredicate

当Homepwner第一次使用BNRItem对象时，会一次性地取出store.data中的所有BNRItem对象。要通过NSManagedObjectContext对象得到这些BNRItem对象，就必须先设置一个NSFetchRequest对象，然后执行该对象。执行NSFetchRequest对象后，可以得到一组与指定的参数相匹配的对象。

执行NSFetchRequest对象前，需要先设置相应的实体描述。实体描述的作用是定义所要获取

的对象的实体。以Homepwner为例，要取回BNRItem对象，就要将实体描述定义为BNRItem实体。此外，还可以为NSFetchRequest对象设置排序描述对象（sort descriptors），用于指定返回对象的排列次序。排序描述对象拥有一个键（和某个实体属性对应）和一个布尔值（代表次序是升序还是降序）。下面让Core Data根据BNRItem实体的orderingValue属性，按升序排列返回的BNRItem对象。在BNRItemStore.m中，定义一个新方法loadAllItems，创建并设置NSFetchRequest对象，然后执行该NSFetchRequest对象，最后将返回的结果存入privateItems，代码如下：

```
- (void)loadAllItems

{

if (！self.privateItems) {

NSFetchRequest *request = [[NSFetchRequest alloc] init];

NSEntityDescription *e = [NSEntityDescription entityForName:@"BNRItem"

inManagedObjectContext:self.context];

request.entity = e;

NSSortDescriptor *sd = [NSSortDescriptor

sortDescriptorWithKey:@"orderingValue"

ascending:YES];

request.sortDescriptors = @[sd];

NSError *error;

NSArray *result = [self.context executeFetchRequest:request

error:&error];

if (！result) {

[NSException raise:@"Fetch failed"

format:@"Reason: %@", [error localizedDescription]];

}

self.privateItems = [[NSMutableArray alloc] initWithArray:result];

}

}
```

同时，在BNRItemStore.m的initPrivate方法末端，向BNRItemStore对象发送loadAllItems消息，代码如下：

```
_context.persistentStoreCoordinator = psc;

[self loadAllItems];

}

return self;

}
```

构建应用，检查是否有语法错误。

这样，Homepwner会一次性地取回所有的BNRItem实体对象。这是相对简单的情况，如果某个应用的数据庞大，就应该只获取需要使用的实体对象。为NSFetchRequest对象设置特定的NSPredicate对象，可以使Core Data只返回符合条件的对象。

一个NSPredicate对象可以包含一个“条件”，其结果可以是真或假。例如，如果只要获取价值大于50元的BNRItem对象，就可以创建一个NSPredicate对象并将该对象加入相应的NSFetchRequest对象，代码如下：

```
NSPredicate *p = [NSPredicate predicateWithFormat:@“valueInDollars > 50”];

[request setPredicate:p];
```

NSPredicate对象的格式字符串可以很长、很复杂。Apple的开发文档《Predicate Programming Guide》对此有详细的介绍。

此外，还可以用NSPredicate对象过滤数组中的对象。因此，即使是通过Core Data获取的allItems数组，一样可以使用NSPredicate对象再次过滤，代码如下：

```
NSArray *expensiveStuff = [allItems filteredArrayUsingPredicate:p];
```

添加和删除BNRItem对象

以上完成了保存和载入功能，下面实现添加和删除功能。改用Core Data后，就不能再用alloc方法和init方法来创建BNRItem对象，而应该通过NSManagedObjectContext对象插入一个针对BNRItem实体的新对象，并得到相应的BNRItem对象。修改BNRItemStore.m中的createItem方法，代码如下：

```
-（BNRItem *）createItem

{
```

~~BNRItem *item = [[BNRItem alloc] init];~~

```
double order;
if ([self.allItems count] == 0) {
order = 1.0;
} else {
order = [[self.privateItems lastObject] orderingValue] + 1.0;
}
NSLog(@"Adding after %d items, order = %.2f",
[self.privateItems count], order);
BNRItem *item =
[NSEntityDescription insertNewObjectForEntityForName:@"BNRItem"
inManagedObjectContext:self.context];
item.orderingValue = order;
[self.privateItems addObject:item];
return item;
}
```

当用户删除某个BNRItem对象后，需要通知NSManagedObjectContext对象从数据库删除相应的数据。将以下代码加入BNRItemStore.m中的removeItem:。

```
- (void)removeItem:(BNRItem *)item
{
NSString *key = item.itemKey;
[[BNRImageStore sharedStore] deleteImageForKey:key];
[self.context deleteObject:item];
[self.privateItems removeObjectIdenticalTo:item];
}
```

排列BNRItem对象

下面要为BNRItemStore实现BNRItem对象的排序功能。因为Core Data在使用关系数据库保存数据时，默认不会保留行的排列位置，所以当某个BNRItem对象在UITableView对象中的位置发生变化时，就必须更新该对象的orderingValue属性。

如果orderingValue是整数类型，那么实现排序功能会有点复杂。当某个BNRItem对象移动至新位置时，其他的BNRItem对象的orderingValue属性也要跟着变化。这也是为什么之前将orderingValue的类型声明为了double。当orderingValue的类型为double时，只需要找出位于插入位置之前和之后的BNRItem对象，将两个对象的orderingValues属性相加并除以2，就可以得到移动后的新orderingValue。修改BNRItemStore.m中的movePossessionAtIndex:toIndex:，加入排序功能，代码如下：

```
- (void)moveItemAtIndex:(NSUInteger)fromIndex

toIndex:(NSUInteger)toIndex

{

if (fromIndex == toIndex) {

return;

}

BNRItem *item = self.privateItems[fromIndex];

[self.privateItems removeObjectAtIndex:fromIndex];

[self.privateItems insertObject:item atIndex:toIndex];

// 为移动的BNRItem对象计算新的orderValue

double lowerBound = 0.0;

// 在数组中，该对象之前是否有其他对象？

if (toIndex > 0) {

lowerBound = [self.privateItems[(toIndex - 1)] orderingValue];

} else {

lowerBound = [self.privateItems[1] orderingValue] - 2.0;
```

```
}

double upperBound = 0.0;

// 在数组中，该对象之后是否有其他对象？

if (toIndex < [self.privateItems count] - 1) {

upperBound = [self.privateItems[(toIndex + 1)] orderingValue];

} else {

upperBound = [self.privateItems[(toIndex - 1)] orderingValue] + 2.0;

}

double newOrderValue = (lowerBound + upperBound) / 2.0;

NSLog(@"moving to order %f", newOrderValue);

item.orderingValue = newOrderValue;

}
```

构建并运行应用。虽然Homepwner的功能没有发生变化，但是内部已经改用Core Data实现数据的存取。

为Homepwner增加BNRAssetType功能

除了BNRItem实体，Homepwner的模型文件还描述了一个名为BNRAssetType的实体。BNRItem实体和BNRAssetType实体之间是一对一的关系。Homepwner需要提供某种途径，使用户可以为某个BNRItem对象设置BNRAssetType对象。此外，还要扩充BNRItemStore，使BNRItemStore对象能够获取BNRAssetType对象（创建此对象的任务将作为练习留给读者完成）。

在BNRItemStore.h中声明一个新方法，代码如下：

```
- (NSArray *)allAssetTypes;
```

在BNRItemStore.m中实现该方法，当Homepwner首次运行时（这时的BNRItemStore对象不会包含任何BNRAssetType对象），需要创建三种默认类型，代码如下：

```
- (NSArray *)allAssetTypes

{
```

```
if（！_allAssetTypes）{
NSFetchRequest *request = [[NSFetchRequest alloc] init];
NSEntityDescription *e =
[NSEntityDescription entityForName:@“BNRAssetType”
inManagedObjectContext:self.context];
request.entity = e;
NSError *error = nil;
NSArray *result = [self.context executeFetchRequest:request
error:&error];
if（！result）{
[NSException raise:@“Fetch failed”
format:@“Reason: %@”, [error localizedDescription]];
}
_allAssetTypes = [result mutableCopy];
}
// 第一次运行？
if（[_allAssetTypes count] == 0）{
NSManagedObject *type;
type = [NSEntityDescription
insertNewObjectForEntityForName:@“BNRAssetType”
inManagedObjectContext:self.context];
[type setValue:@“Furniture” forKey:@“label”];
[_allAssetTypes addObject:type];
type = [NSEntityDescription
```

```
    insertNewObjectForEntityForName:@“BNRAssetType”

    inManagedObjectContext:self.context];

    [type setValue:@“Jewelry” forKey:@“label”];

    [_allAssetTypes addObject:type];

    type = [NSEntityDescription

    insertNewObjectForEntityForName:@“BNRAssetType”

    inManagedObjectContext:self.context];

    [type setValue:@“Electronics” forKey:@“label”];

    [_allAssetTypes addObject:type];

    }

    return _allAssetTypes;

    }
```

下面要修改用户界面，使用户能够在BNRDetailViewController对象的视图中查看某个BNRItem对象的BNRAssetType对象，并进行修改（见图23-11）。

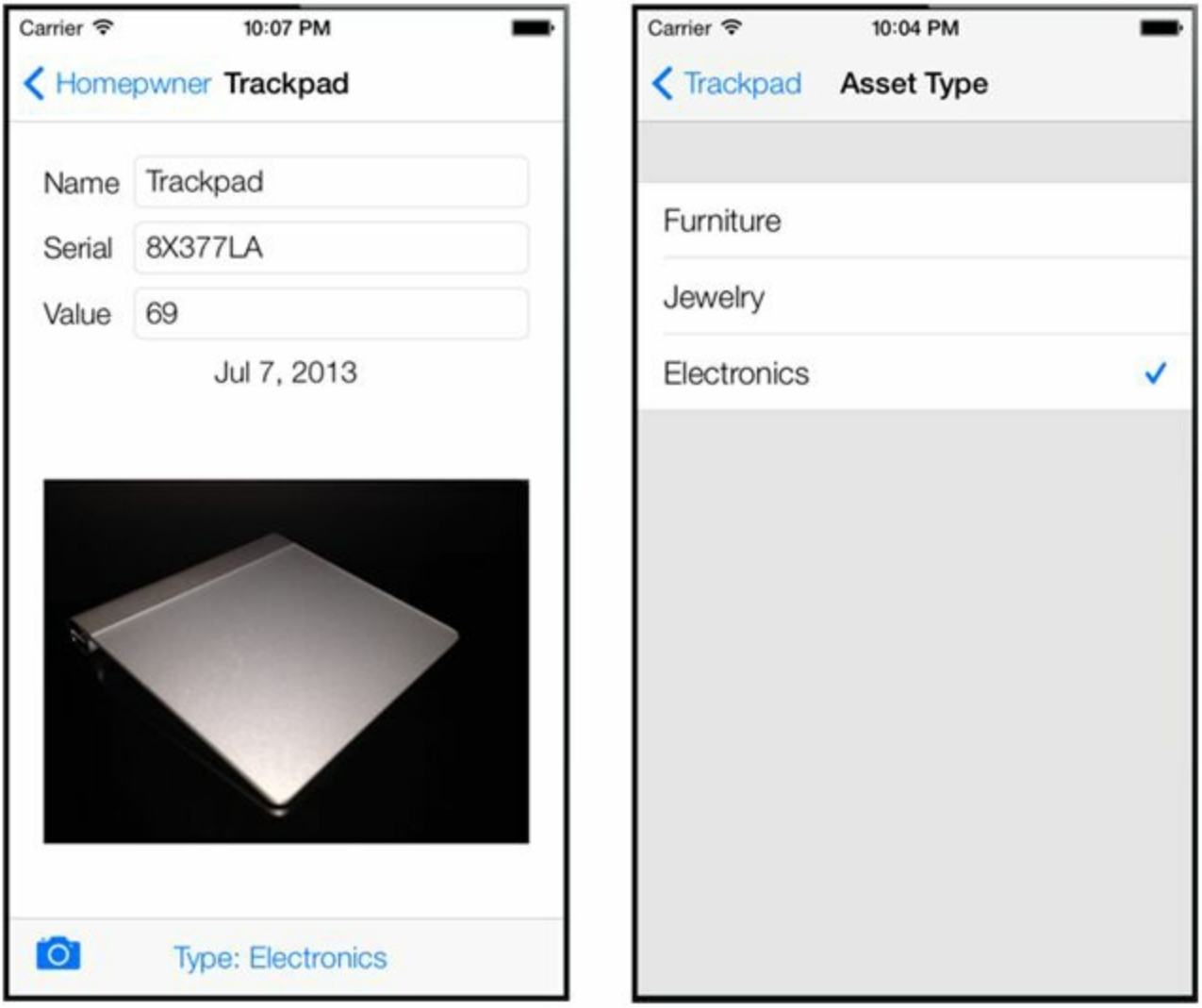


图23-11 查看和修改BNRAssetType对象的界面

使用Objective-C class模板创建新文件，父类选择NSObject，类名使用BNRAssetType-ViewController。

在BNRAssetTypeViewController.h中，先前置声明BNRItem类，然后将BNRAssetTypeViewController的父类改为UITableViewController，最后为BNRAssetTypeViewController声明一个类型为BNRItem的item属性。

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

@class BNRItem;

@interface BNRAssetTypeViewController : NSObject

@interface BNRAssetTypeViewController : UITableViewController

@property (nonatomic, strong) BNRItem *item;

@end
```

BNRAssetTypeViewController对象的作用是显示一组可供用户选择的BNRAssetType对象。按下BNRDetailViewController中的指定按钮，Homepwner应该会显示BNRAssetTypeViewController对象。在BNRAssetTypeViewController.m中实现数据源方法并导入相应的头文件（前文已经介绍过如何实现这类代码）：

```
#import “BNRAssetTypeViewController.h”

#import “BNRItemStore.h”

#import “BNRItem.h”

@implementation BNRAssetTypeViewController

- (instancetype)init

{

return [super initWithStyle:UITableViewStylePlain];

}

- (instancetype)initWithStyle:(UITableViewStyle)style

{

return [self init];

}

- (void)viewDidLoad

{

[super viewDidLoad];

[self.tableView registerClass:[UITableViewCell class]

forCellReuseIdentifier:@“UITableViewCell”];

}

- (NSInteger)tableView:(UITableView *)tableView

numberOfRowsInSection:(NSInteger)section

{

return [[[BNRItemStore sharedStore] allAssetTypes] count];
```

```
}
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
UITableViewCell *cell =
[tableView dequeueReusableCellWithIdentifier:@"UITableViewCell"
forIndexPath:indexPath];
NSArray *allAssets = [[BNRItemStore sharedStore] allAssetTypes];
NSManagedObject *assetType = allAssets[indexPath.row];
// 通过键-值编码(key-value coding)得到BNRAssetType对象的label属性
NSString *assetLabel = [assetType valueForKey:@"label"];
[cell.textLabel.text = assetLabel;
// 为当前选中的对象加上勾选标记
if (assetType == self.item.assetType) {
cell.accessoryType = UITableViewCellAccessoryCheckmark;
} else {
cell.accessoryType = UITableViewCellAccessoryNone;
}
return cell;
}
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexPath];
```

```
    cell.accessoryType = UITableViewCellAccessoryCheckmark;

    NSArray *allAssets = [[BNRItemStore sharedStore] allAssetTypes];

    NSManagedObject *assetType = allAssets[indexPath.row];

    self.item.assetType = assetType;

    [self.navigationController popViewControllerAnimated:YES];

    }

    @end
```

打开BNRDetailViewController.xib，向UIToolbar上拖曳一个UIBarButtonItem，然后按住Control将其拖曳到BNRDetailViewController.m的类扩展中，创建名为assetTypeButton插座变量。接下来使用同样的方法为assetTypeButton创建动作方法，命名为showAssetTypePicker:。

现在BNRDetailViewController.m中应该已经声明了以下属性和方法：

```
    @property（weak, nonatomic）IBOutlet UIBarButtonItem *assetTypeButton;

    @end

    @implementation BNRDetailViewController

    // 省略其他方法

    -（IBAction）showAssetTypePicker:（id）sender

    {

    }

    @end
```

在BNRDetailViewController.m顶部导入BNRAssetTypeViewController.h，代码如下：

```
    #import “BNRDetailViewController.h”

    #import “BNRAssetTypeViewController.h”
```

在BNRDetailViewController.m中实现showAssetTypePicker:，代码如下：

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

    {
```

```
[self.view endEditing:YES];

BNRAssetTypeViewController *avc = [[BNRAssetTypeViewController

alloc] init]; avc.item = self.item;

[self.navigationController pushViewController:avc

animated:YES];

}
```

最后更新UIBarButtonItem对象的标题，显示指定的BNRItem对象的类型。将以下代码加入BNRDetailViewController.m中的viewWillAppear:。

```
if (self.itemKey) {

// 根据itemKey从BNRImageStore对象获取相应的图片

UIImage *imageToDisplay = [[BNRImageStore sharedStore]

imageForKey:self.itemKey];

// 将得到的图片赋给UIImageView对象

self.imageView.image = imageToDisplay;

} else {

// 清空UIImageView对象

self.imageView.image = nil;

}

NSString *typeLabel = [self.item.assetType valueForKey:@“label”];

if (！typeLabel) {

typeLabel = @“None”;

}

self.assetTypeButton.title = [NSString stringWithFormat:@“Type: %@”, typeLabel];

[self updateFonts];

}
```

构建并运行应用。选中某个BNRItem对象后，应该能设置该对象的BNRAssetType对象。

23.3 再谈SQL

前面已介绍了如何通过Core Data来使用SQLite。如果读者想知道Core Data具体执行了哪些SQL命令，则可以通过为应用设置一个特定的命令行参数，要求Core Data将所有的SQLite调用输出至控制台。以Homepwner为例，设置该命令行参数的流程为：选择Product菜单中的Edit Scheme…，选择左侧列表中的Run Homepwner.app，然后选择右侧的Arguments（参数）标签。增加-com.apple.CoreData.SQLDebug和1两个参数（见图23-12）。

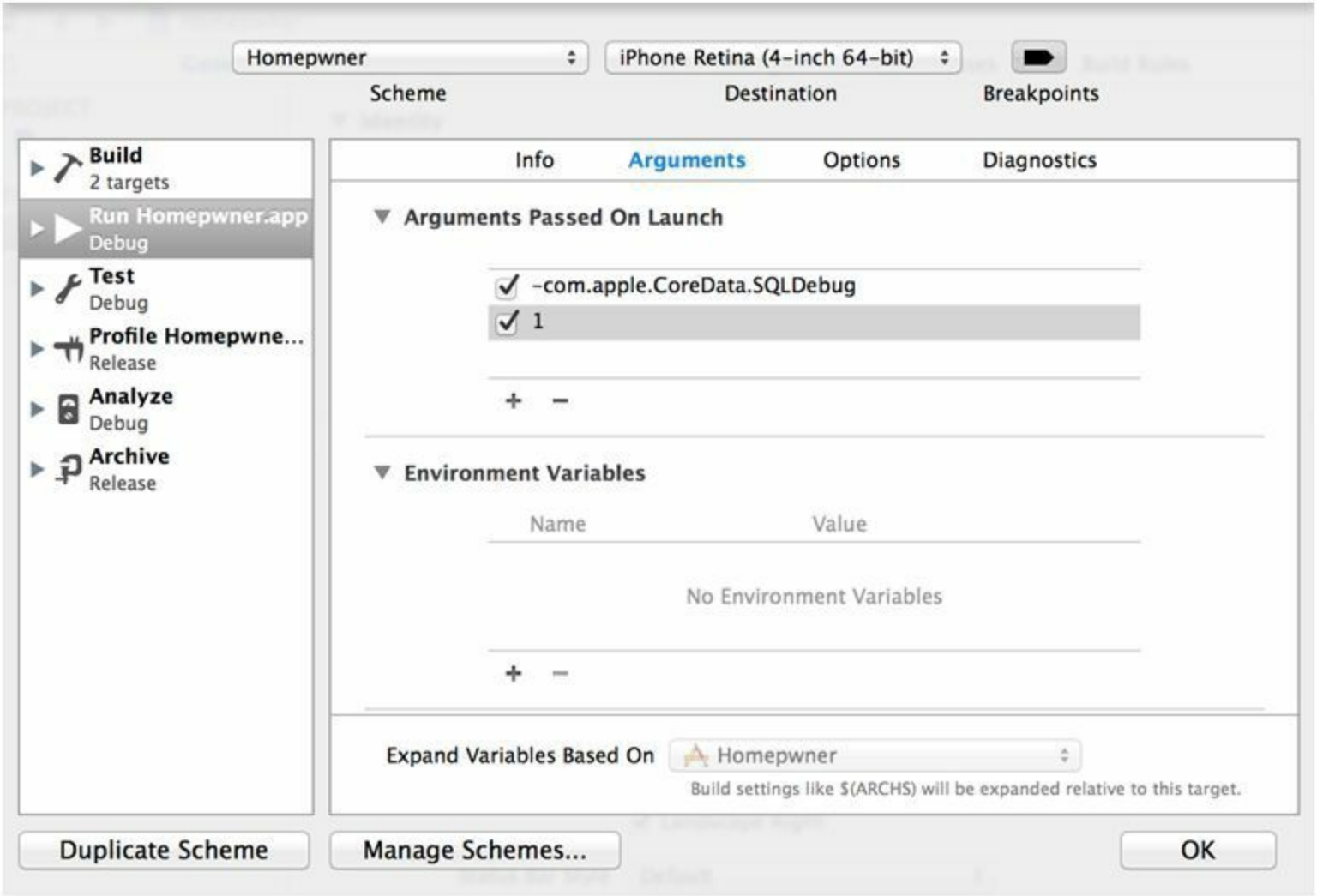


图23-12 打开Core Data的SQL日志输出开关

构建并运行应用。打开调试区域和控制台，以便查看SQL输出。增加若干BNRItem对象，然后随意浏览并查看这些对象。读者应该会在控制台看到相应的输出信息。

23.4 Faults

Core Data在创建实体对象时，不会根据实体中的关系立刻获取相应的对象。也就是说，在获取带关系的NSManagedObject对象时，Core Data会使用一种轻量级的占位符对象——faults（触发对象）代替关系另一端的对象，直到确实需要访问重量级的真实对象。

faults分为一对多faults（代替NSSet对象）和一对一faults（代替NSManagedObject对象）。以BNRItem为例，当Homepwner应用通过Core Data取回BNRItem对象时，Core Data不会立刻创建这些BNRItem对象的BNRAssetType对象，而是创建faults并代替BNRAssetType对象（见图23-13）。当Homepwner需要使用BNRAssetType对象时，Core Data才会创建相应的对象。

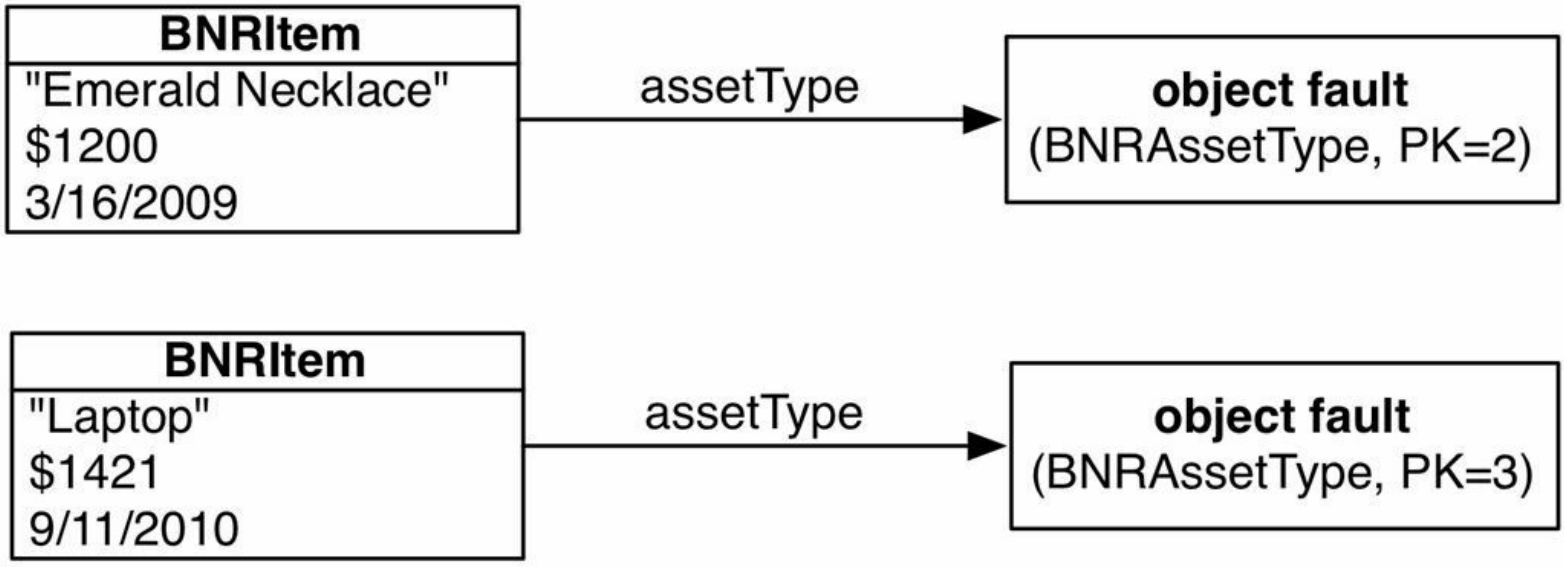


图23-13 替换faults前

faults知道自己所代表的实体和主键（primary key）。以BNRItem为例，当应用向某个代表BNRAssetType对象的faults查询类型名称（label属性）时，Core Data所执行的SQL语句如下：

```
SELECT t0.Z_PK, t0.Z_OPT, t0.ZLABEL FROM ZBNRASSETTYPE t0 WHERE t0.Z_PK = 2
```

字段名的前缀Z_及OPT都属于Core Data的实现细节，本书不做讨论。Core Data在替换faults时，会将包含真实数据的NSManagedObject对象放置在完全相同的内存位置上（见图23-14）。

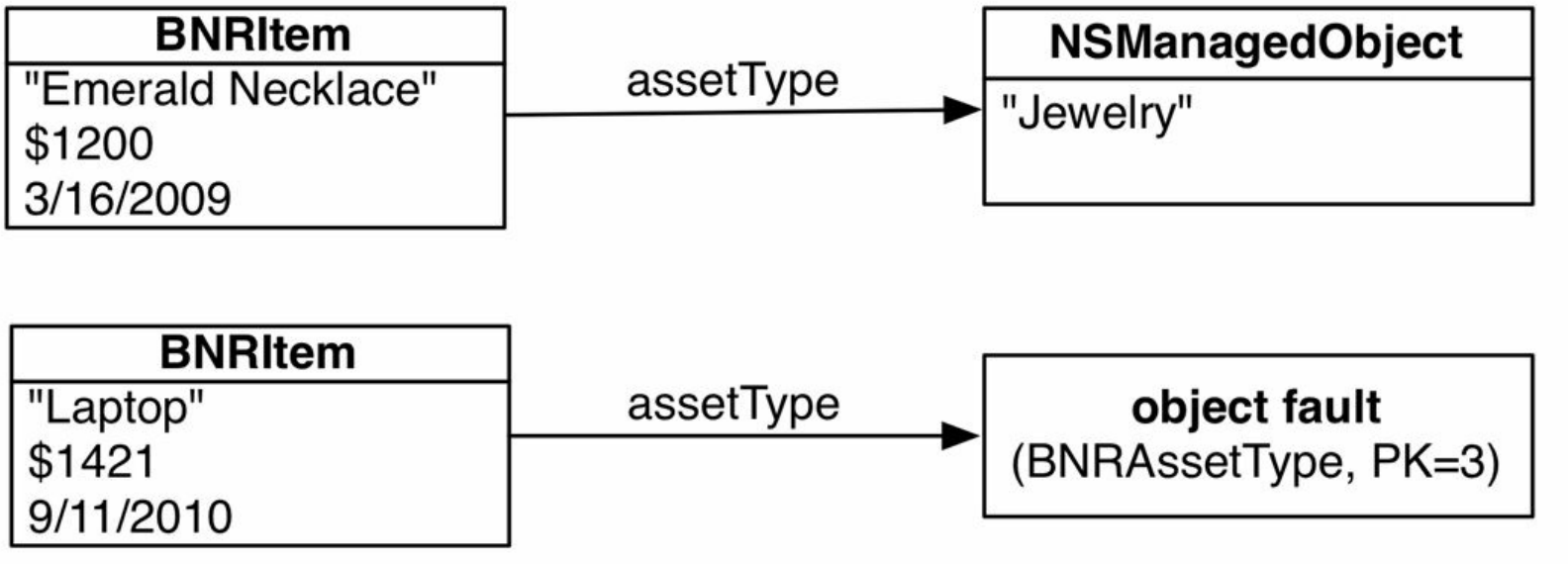


图23-14 替换faults后

这种“延迟获取”机制使Core Data更容易使用，效率也更高。

下面介绍一对多类型的faults。还是以Homepwner为例，假设它要以另一种模式工作：先显示一组BNRAssetType对象供用户选择，然后根据选中的BNRAssetType对象获取相应的BNRItem对象，最后显示这些BNRItem对象。如何通过Core Data完成这项任务？对Core Data第一次取回的BNRAssetType对象，都会有一个一对多类型的faults，代替包含BNRItem对象的NSSet对象（见图23-15）。

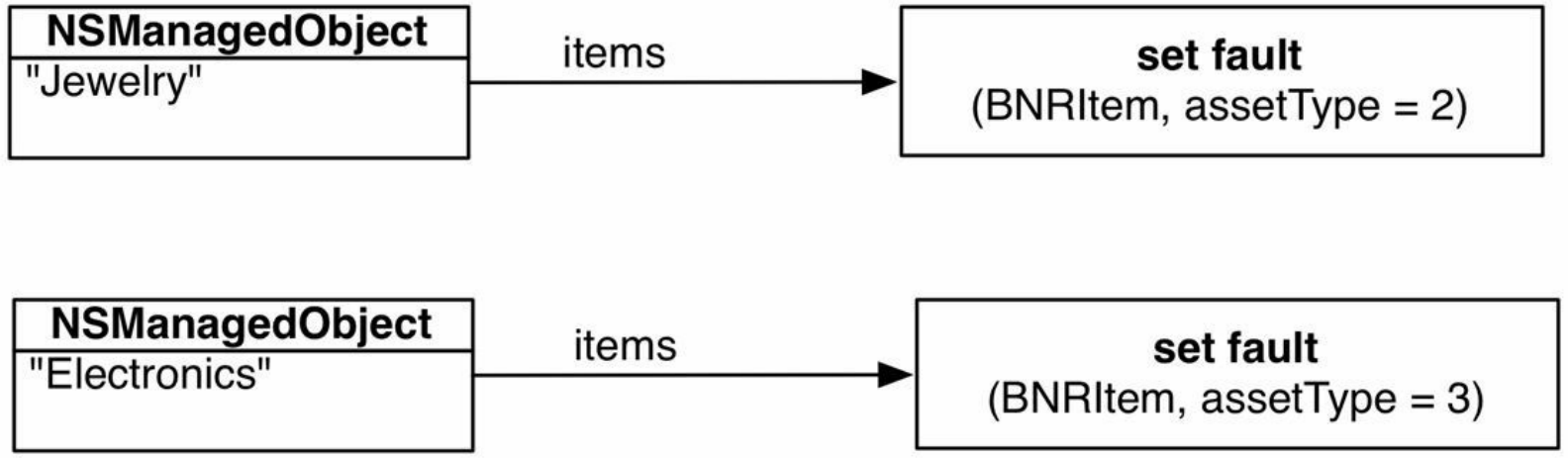


图23-15 一对多faults

当一对多faults收到特定的消息，要求返回BNRItem对象时，Core Data就会取回相应的对象，并用一个NSSet对象替换掉这个faults（见图23-16）。

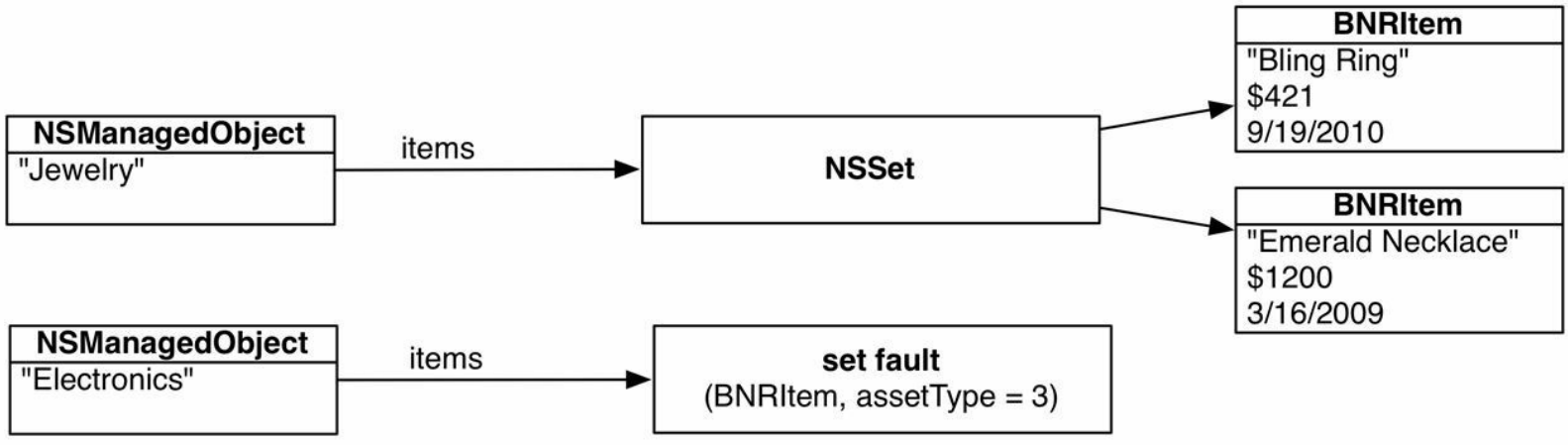


图23-16 替换一对多faults后

Core Data是一套功能强大、使用灵活的数据存取框架，本章只针对其主要功能做了一个简单的介绍，详细信息请参阅Apple提供的开发文档《Core Data Programming Guide》。本章没有涉及的Core Data内容包括：

lNSFetchRequest提供了一套功能强大的机制，可以用来指定需要获取的数据。本章只使用了它的一小部分功能，读者应该对其有一个更深入的了解。此外，还应该了解这些和其有关的类：NSPredicate、NSSortOrdering、NSExpressionDescription和NSExpression。最后，还可以在模型文件中创建NSFetchRequest模板。

生成属性（fetched property）。生成属性有点类似一对多关系，也有点类似NSFetchRequest对象。通常可以在模型文件中设置生成属性。

随着应用的更新和升级，需要经常修改数据模型。修改数据模型不是一个简单的问题，这也是为什么Apple编写了《Data Model Versioning and Data Migration Programming Guide》一整本书来介绍这个问题。

Core Data支持数据验证，可以在数据转成NSManagedObject对象时进行验证，也可以在保存NSManagedObject对象时进行验证。

一个NSManagedObjectContext对象可以同时配合多个NSPersistentStoreCoordinator对象使用，也可以将模型分割成不同的配置，并为每个配置分配不同的NSPersistentStoreCoordinator对象。在实体对象之间，如果NSPersistentStore- Coordinator对象不同，就不能建立关系（但是可以通过生成属性达到类似的效果）。

23.5 各种存取机制的优缺点

iOS应用可以通过多种途径保存和读取数据。读者要考虑哪种途径更适合自己的应用。表23-1列出了几种途径的优缺点，以帮助读者做决定。

表23-1 存取机制优缺点

技术	优　　点	缺　　点
固化	支持有序的一对多关系（对应 NSArray 对象，而不是 NSSet 对象）。容易处理版本差异	一次性读入全部对象（不支持 faults）。不支持增量更新
Web 服务	能很容易地在不同的设备和应用间共享数据	需要服务器支持，需要网络连接
Core Data	默认会使用延迟获取机制。支持增量更新	不容易处理版本间的差异（通过 NSModelMapping 对象可以解决该问题）。默认不支持有序的一对多关系，需要自己实现

23.6 初级练习：Asset的iPad界面

在iPad中，用UIPopoverController对象显示BNRAssetTypeViewController对象。

23.7 中级练习：增加BNRAssetType对象

为BNRAssetTypeViewController的navigationItem增加一个按钮，使用户能够增加新的BNRAssetType对象。

23.8 高级练习：显示某种类型的BNRItem对象

修改BNRAssetTypeViewController，为UITableView对象增加一个表格段，并在该表格段中显示所有属于指定类型的BNRItem对象。

第24章 状态恢复

本书第18章介绍过，应用具有状态周期，如果应用进入后台运行状态，同时设备内存过低，系统就可能终止该应用，并释放应用所占用的内存。为了保证流畅的用户体验，应该在用户离开应用时保存状态，即使应用被系统终止，仍然要让用户产生应用没有关闭过的感觉。

为了保存和恢复应用状态，必须为应用启用状态恢复（state restoration）。状态恢复的工作原理与归档类似，当应用进入后台运行时，系统会保存应用的视图控制器层次结构。当应用被系统终止后，系统会在用户下一次启动应用时恢复层次结构中的所有视图控制器。（如果系统没有终止应用，那么应用仍然处于内存中，不需要恢复状态。）

本章将为Homepwner应用启用状态恢复。首先演示如果没有启用状态恢复，Homepwner会有哪些问题。

打开Homepwner.xcodeproj，构建并运行应用。创建一个新的BNRItem对象，然后进入其详细界面（BNRDetailViewController）。下面要模拟系统终止应用，并触发状态恢复的过程。在iOS模拟器中，点击Home键（键盘快捷键是Command-Shift-H），让应用进入后台运行。现在模拟系统关闭应用的过程——回到Xcode，点击Stop按钮（Command-.），再重新运行应用。

当应用再次启动时，屏幕会短暂地显示BNRDetailViewController，然后立即切换到BNRItemsViewController。这是由于当应用进入后台运行时，系统会生成当前界面的快照（snapshot）。当应用再次启动时，系统会将之前的快照作为应用的启动图片，直到应用启动成功后，系统才会移除快照。

如果应用没有实现状态恢复功能，用户会短暂地看见应用之前的状态，然后立即恢复到初始状态。因此，有必要为Homepwner添加状态恢复功能，提升用户体验。

24.1 状态恢复的工作原理

读者可以将一个运行中的应用想象为一棵由众多视图控制器和视图组成的树，其中树的根是应用的根视图控制器。例如，HypnoNerd应用的“树”类似于图24-1。

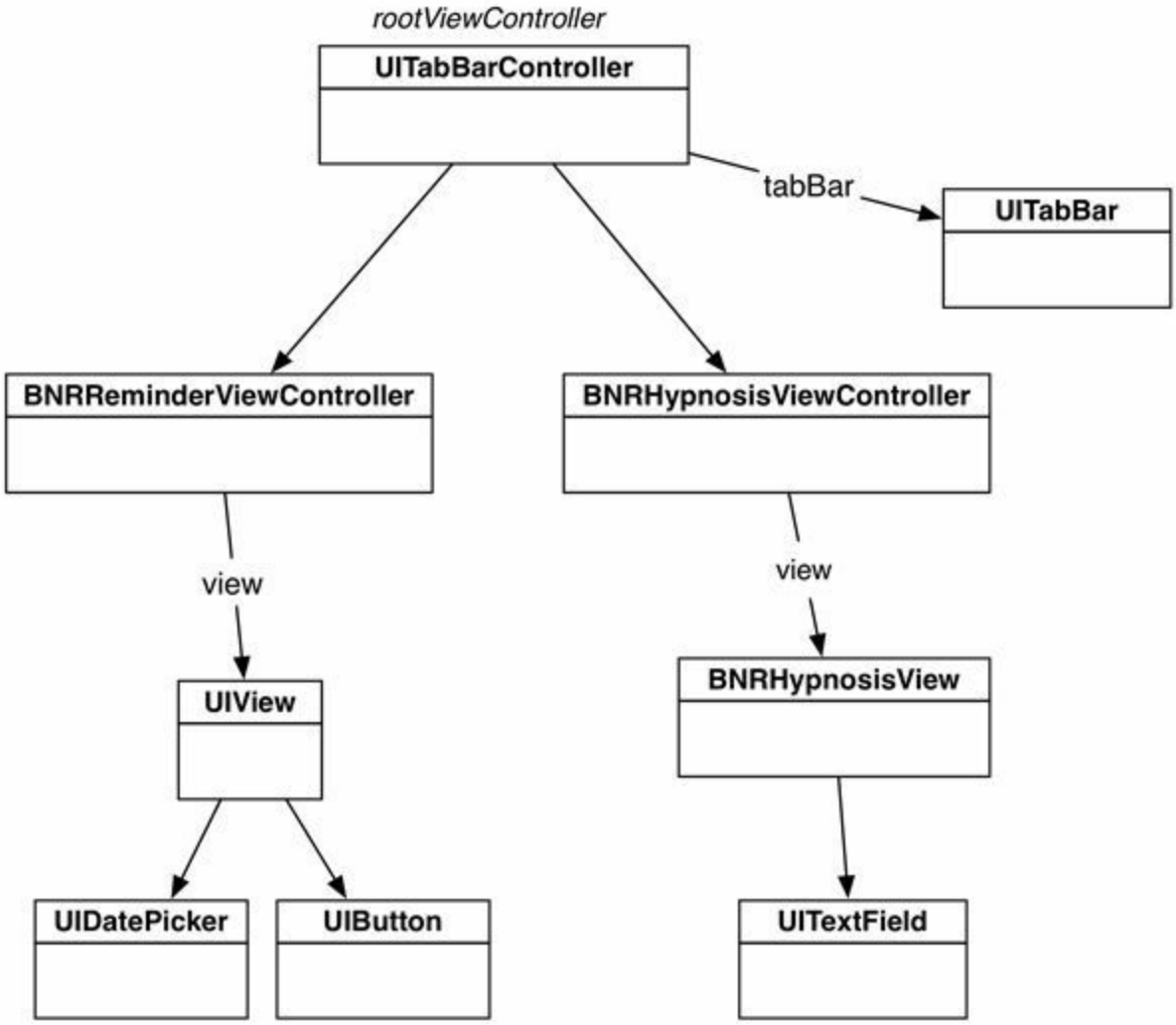


图24-1 应用树

如果应用启用了状态恢复，系统在终止应用之前会遍历应用树中的每一个节点，记录每个节点的状态信息，例如，对象的唯一标识、类和需要保存的状态数据。在终止应用之后，系统会将这些信息存储到文件系统中。

其中，对象的唯一标识又称为对象的恢复标识（restoration identifier），通常与对象的类名相同；类称为恢复类（restoration class），通常与该对象的isa指针指向的类相同；状态数据则保存了对象状态信息，例如，UITabBarController的状态数据包括当前选中的是哪一个标签项。

当用户重新启动应用后，系统会读取之前保存的状态信息，重新创建应用树，依次恢复树中的每一个节点：

- 系统通过节点的恢复类为该节点创建一个新的视图控制器。

•将一组恢复标识赋给新节点:包括该节点的恢复标识及其所有祖先节点的恢复标识。数组中第一个标识是根节点的恢复标识,最后一个标识是新节点的恢复标识。

•将对应的状态数据赋给新节点。状态数据保存在第18章介绍的NSCoder对象中。

24.2 启用状态恢复

状态恢复在默认情况下是禁用的，如果需要启用，必须手动在应用程序委托中设置。

打开BNRAppDelegate.m，实现保存和恢复应用状态的两个委托方法，代码如下：

```
@implementation BNRAppDelegate
- (BOOL)application:(UIApplication *)application
shouldSaveApplicationState:(NSCoder *)coder
{
return YES;
}
- (BOOL)application:(UIApplication *)application
shouldRestoreApplicationState:(NSCoder *)coder
{
return YES;
}
```

这样，当应用进入后台运行时，系统会保存应用状态；而当应用恢复到前台运行或重新启动时，系统会恢复应用状态。为了理解系统恢复应用状态的过程，下一节将详细介绍恢复标识和恢复类。

24.3 恢复标识和恢复类

系统在保存应用状态时，首先会查询UIWindow对象rootViewController的restorationIdentifier（恢复标识）属性，如果rootViewController具有restorationIdentifier属性，那么系统会保存其状态；然后，系统会遍历rootViewController的子视图控制器，并保存它们的状态；接下来继续遍历这些视图控制器的子视图控制器……直到层次结构的最后一级。如果其中某个视图控制器没有restorationIdentifier属性，那么系统不会保存该视图控制器及其所有子孙视图控制器的状态，无论子孙视图控制器是否具有restorationIdentifier属性。

例如，在图24-2中，应用有两个用灰色标识的视图控制器，它们没有恢复标识，因此系统无法恢复其状态，也无法恢复其子视图控制器的状态。

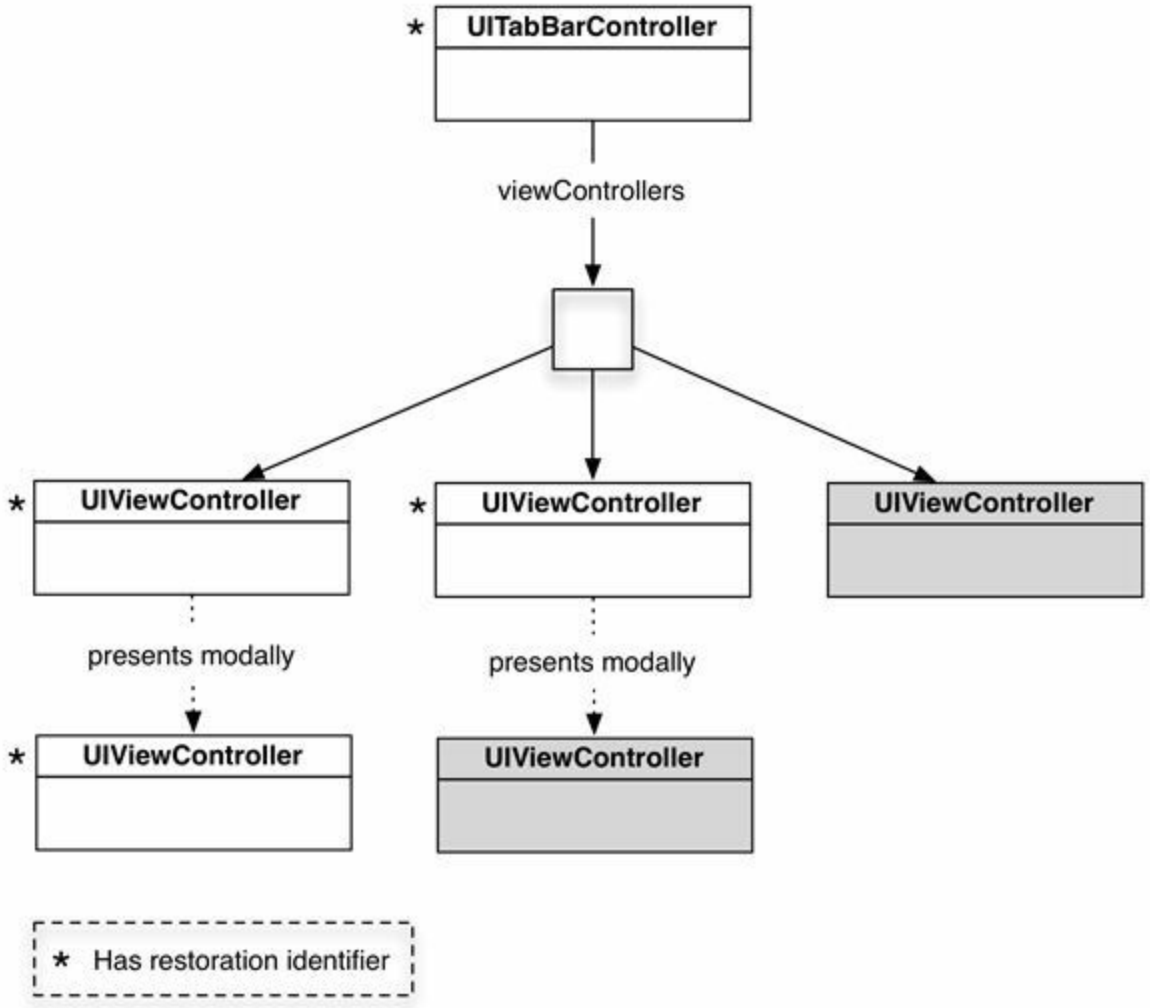


图24-2 恢复标识

根据Apple命名规范，通常应该将类名作为类的恢复标识。打开BNRAppDelegate.m，在application:didFinishLaunchingWithOptions:中为UINavigationController对象设置恢复标识：

- (BOOL)application:(UIApplication *)application

didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

```
{
self.window = [[UIWindow alloc]
initWithFrame:UIScreen.mainScreen.bounds]];
BNRItemsViewController *itemsViewController =
[[BNRItemsViewController alloc] init];
// 创建一个UINavigationController对象,
// 其根视图控制器是itemsViewController
UINavigationController *navController = [[UINavigationController alloc]
initWithRootViewController:itemsViewController];
// 将UINavigationController对象的类名设置为恢复标识
navController.restorationIdentifier =
NSStringFromClass([navController class]);
// 将UINavigationController对象设置为UIWindow的rootViewController
self.window.rootViewController = navController;
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}
```

现在UINavigationController对象已经有了恢复标识，系统会保存该对象的状态，同时，如果该对象的子孙视图控制器有恢复标识，系统也会保存它们的状态。

对于BNRItemsViewController和BNRDetailViewController，应该在指定初始化方法中设置其恢复标识。除此之外，还需要为两个视图控制器设置恢复类(restoration class)。系统在恢复某个对象的状态时，会要求其恢复类创建该对象。

打开BNRItemsViewController.m，修改init方法，设置恢复标识和恢复类：

```
- (instancetype)init
{
```

```
// 调用父类的指定初始化方法
self = [super initWithStyle:UITableViewStylePlain];
if (self) {
UINavigationItem *navItem = self.navigationItem;
navItem.title = @“Homepwner”;
self.restorationIdentifier = NSStringFromClass([self class]);
self.restorationClass = [self class];
```

对于BNRDetailViewController，则应该在initForNewItem:方法中设置：

```
- (instancetype)initForNewItem:(BOOL)isNew
{
self = [super initWithNibName:nil bundle:nil];
if (self) {
self.restorationIdentifier = NSStringFromClass([self class]);
self.restorationClass = [self class];
if (isNew) {
```

最后，还要为添加BNRItem对象时以模态形式推入的UINavigationController对象设置恢复标识。

重新打开BNRItemsViewController.m，修改addNewItem:方法，代码如下：

```
- (IBAction)addNewItem:(id)sender
{
// 通过BNRItemStore单例创建一个新的BNRItem对象
BNRItem *newItem = [[BNRItemStore sharedStore] createItem];
BNRDetailViewController *detailViewController =
[[BNRDetailViewController alloc] initForNewItem:YES];
detailViewController.item = newItem;
```

```
detailViewController.dismissBlock = ^{

[self.tableView reloadData];

};

UINavigationController *navController = [[UINavigationController alloc]

initWithRootViewController:detailViewController];

navController.restorationIdentifier =

NSStringFromClass([navController class]);
```

（请注意，以上代码并没有为两个UINavigationController对象设置恢复类，它们将由应用程序委托负责创建，下一节会介绍相关过程。）

现在，Homepwner中所有视图控制器都已经具有了恢复标识，当应用进入后台运行或被终止时，系统会保存所有视图控制器的状态。

24.4 状态恢复与应用生命周期

由于应用已经启用了状态恢复，因此应用的生命周期多出了一环——应用启动后首先会判断是否需要恢复应用状态，如图24-3所示。目前创建UIWindow对象和rootViewController的代码都位于application:didFinishLaunchingWithOptions:方法中，需要针对状态恢复修改代码。

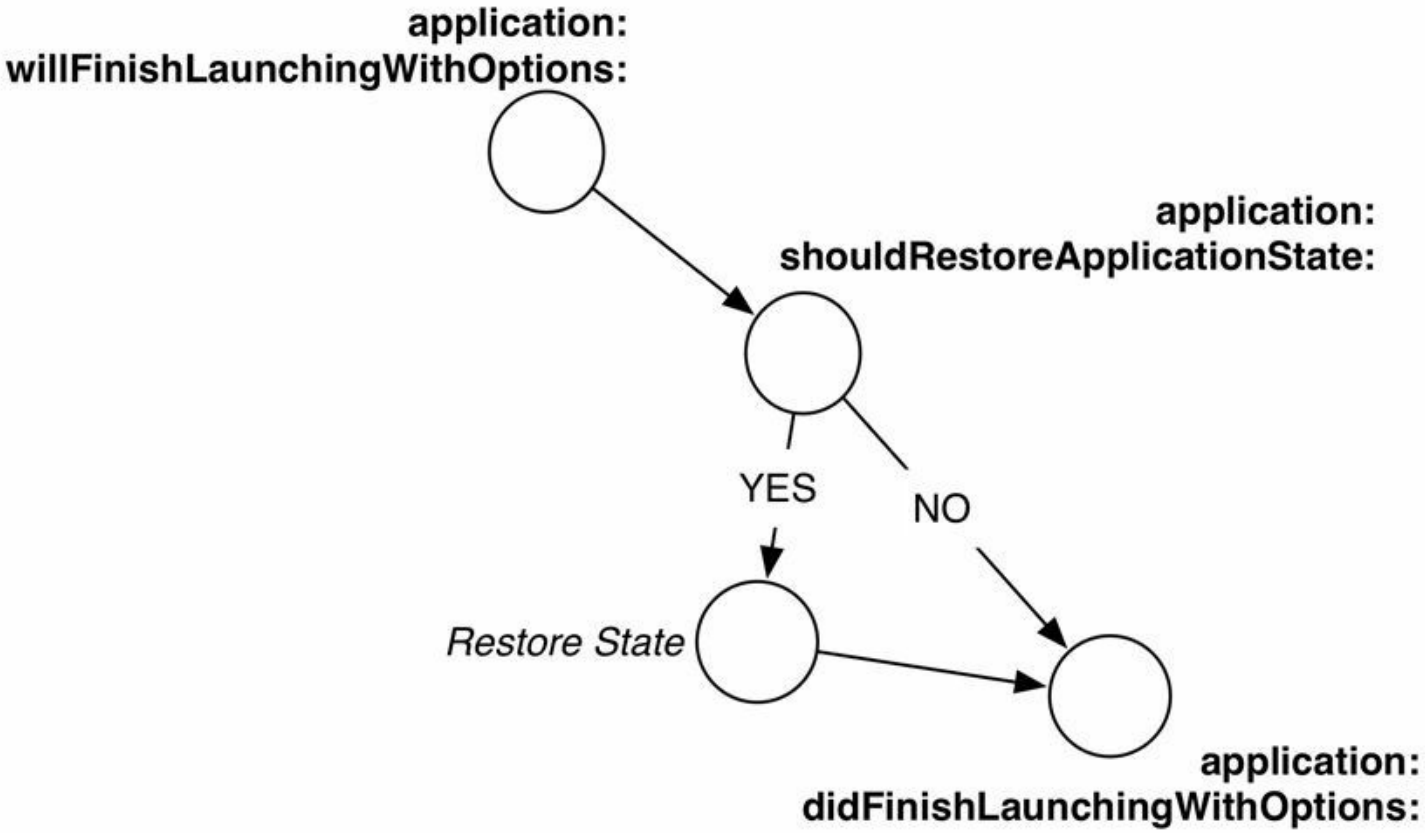


图24-3 启用状态恢复后的生命周期

application:willFinishLaunchingWithOptions:方法会在系统恢复应用状态之前调用。因此，需要在状态恢复触发之前执行的代码都应该写在该方法中，例如创建并设置UIWindow对象。

在BNRAppDelegate.m中覆盖该方法，创建并设置UIWindow对象，代码如下：

```
- (BOOL)application:(UIApplication *)application
willFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
self.window =
[[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
return YES;
```

}

然后更新application:didFinishLaunchingWithOptions:方法，如果应用没有触发状态恢复（例如，应用第一次启动时），就需要创建并设置应用的各个视图控制器。同时，创建并设置UIWindow对象的代码已经移动到了application: willFinishLaunching- WithOptions:中，因此要删除application: didFinishLaunchingWithOptions:中的相关代码：

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
~~self.window =~~
~~[[UIWindow alloc] initWithFrame:UIScreen.mainScreen.bounds]];~~
// 如果应用没有触发状态恢复，就创建并设置各个视图控制器
if (！self.window.rootViewController) {
BNRItemsViewController *itemsViewController =
[[BNRItemsViewController alloc] init];
// 创建一个UINavigationController对象，
// 其根视图控制器是itemsViewController
UINavigationController *navController = [[UINavigationController alloc]
initWithRootViewController:itemsViewController];
// 将UINavigationController对象的类名设置为恢复标识
navController.restorationIdentifier =
NSStringFromClass([navController class]);
// 将UINavigationController对象设置为UIWindow的rootViewController
self.window.rootViewController = navController;
}
~~self.window.backgroundColor = [UIColor whiteColor];~~
[self.window makeKeyAndVisible];
```

```
return YES;

}
```

24.5 恢复视图控制器

之前为Homepwner中的两个视图控制器设置了恢复类，恢复类将负责创建相应的视图控制器。在BNRItemsViewController.h中使BNRItemsViewController遵守UIViewController-Restoration协议，代码如下：

```
@interface BNRItemsViewController : UITableViewController

<UIViewControllerRestoration>

@end
```

然后打开BNRItemsViewController.m，实现UIViewControllerRestoration协议中唯一的方法，该方法返回一个新的视图控制器。代码如下：

```
+ (UIViewController *)viewControllerWithRestorationIdentifierPath:

(NSArray *)path coder:(NSCoder *)coder

{

return [[self alloc] init];

}
```

同样，BNRDetailViewController也需要遵守UIViewControllerRestoration协议，修改BNRDetailViewController.h，代码如下：

```
@interface BNRDetailViewController : UIViewController

<UIViewControllerRestoration>
```

下面有个问题：为了实现UIViewControllerRestoration协议的方法，需要使用initForNewItem:方法创建BNRDetailViewController对象，而initForNewItem:方法有一个BOOL类型的参数，如何知道是传入YES还是NO呢？答案是通过恢复标识路径（restoration identifier path）。

恢复标识路径其实是一个恢复标识数组，存储了该视图控制器及其所有祖先视图控制器的恢复标识。图24-4是Homepwner应用中各个视图控制器的恢复标识路径。

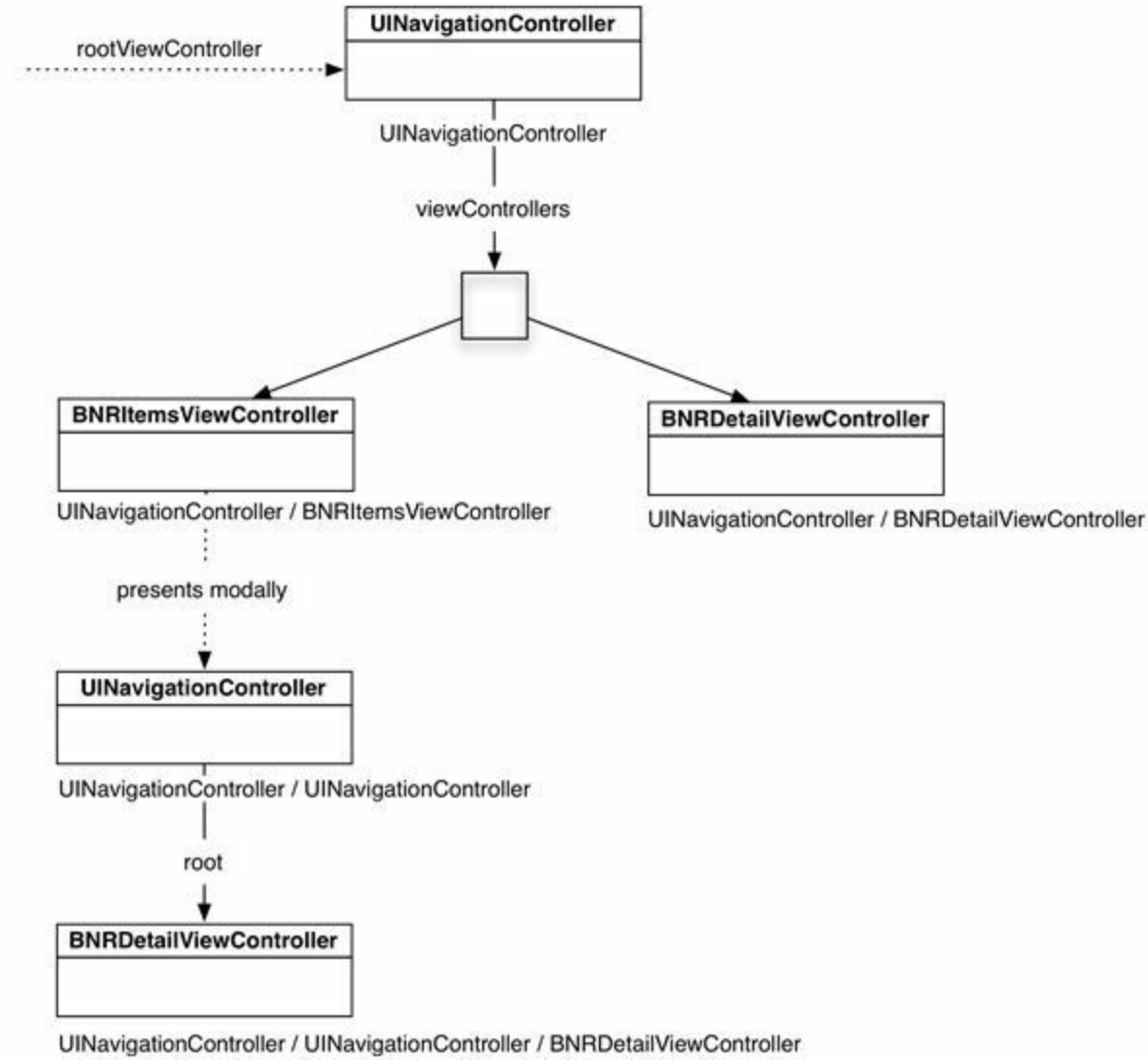


图24-4 恢复标识路径

图中大部分视图控制器的恢复标识路径都非常直观，唯独需要注意以模态形式推入的UINavigationController对象。

实际上，UINavigationController对象并不是由BNRItemsViewController对象推入的，而是BNRItemsViewController对象的父视图控制器——应用的根视图控制器（图24-4中顶层UINavigationController对象）。因此，该UINavigationController对象及其子视图控制器BNRDetailViewController的恢复标识路径中并不包括BNRItemsViewController。

由此可知，如果是查看已经存在的BNRItem对象，那么恢复标识数组的元素数量为2；如果是创建新的BNRItem对象，则元素数量为3。

打开BNRDetailViewController.m，实现UIViewControllerRestoration协议的方法，代码如下：

```
+ (UIViewController *)viewControllerWithRestorationIdentifierPath:

(NSArray *)path coder:(NSCoder *)coder

{
```

```
BOOL isNew = NO;

if ([path count] == 3) {

isNew = YES;

}

return [[self alloc] initForNewItem:isNew];

}
```

现在，BNRDetailViewController也可以正确恢复状态了，如果是创建新BNRItem对象，其导航栏会显示Cancel和Done两个按钮；否则只显示返回按钮。

接下来还需要为两个UINavigationController对象添加状态恢复功能，之前为它们设置了恢复标识，但是没有设置恢复类。如果某个需要恢复的对象没有恢复类，系统会要求应用程序委托创建该对象。打开BNRAppDelegate.m，实现UIApplicationDelegate协议中的application:viewControllerWithRestorationIdentifierPath:coder:方法，代码如下：

```
- (UIViewController *)application:(UIApplication *)application

viewControllerWithRestorationIdentifierPath:

(NSArray *)identifierComponents coder:(NSCoder *)coder

{

// 创建一个新的UINavigationController对象

UIViewController *vc = [[UINavigationController alloc] init];

// 恢复标识路径中的最后一个对象就是UINavigationController对象的恢复标识

vc.restorationIdentifier = [identifierComponents lastObject];

// 如果恢复标识路径中只有一个对象，

// 就将UINavigationController对象设置为UIWindow的rootViewController

if ([identifierComponents count] == 1) {

self.window.rootViewController = vc;

}

return vc;
```

```
}
```

构建并运行应用，创建一个新的BNRItem对象，进入其详细界面，然后触发状态恢复。这时Homepwner可以正确回到BNRDetailViewController界面，但是界面上并没有显示BNRItem对象的任何信息。虽然Homepwner可以正确保存视图控制器的层级结构，但是还没有保存相应的模型对象（BNRItem对象）。下一节将介绍如何保存并恢复BNRDetailView- Controller显示的BNRItem对象。

24.6 编码状态数据

为了保存状态信息，UIViewController需要持久化状态数据，持久化的方式与第18章介绍的归档(archiving)类似，也是使用NSCoder对象对数据进行编码(encoding)。下面就演示如何编码需要的状态数据。

打开BNRDetailViewController.m，编码当前BNRItem对象的itemKey属性，代码如下：

```
- (void)encodeRestorableStateWithCoder:(NSCoder *)coder

{

[coder encodeObject:self.item.itemKey

forKey:@"item.itemKey"];

[super encodeRestorableStateWithCoder:coder];

}
```

对应地，还需要实现解码(decoding)方法，通过itemKey在BNRItemStore中查找相应的BNRItem对象：

```
- (void)decodeRestorableStateWithCoder:(NSCoder *)coder

{

NSString *itemKey =

[coder decodeObjectForKey:@"item.itemKey"];

for (BNRItem *item in [[BNRItemStore sharedStore] allItems]) {

if ([itemKey isEqualToString:item.itemKey]) {

self.item = item;

break;

}

}

[super decodeRestorableStateWithCoder:coder];

}
```

构建并运行应用，进入某一个BNRItem对象的详细界面，然后触发状态恢复。这次BNRDetailViewController可以正确显示之前选择的BNRItem。

现在还有一个问题：BNRDetailViewController在恢复状态时会将BNRItem的各项属性分别赋给对应的UITextField对象，如果用户之前在UITextField对象中输入过新值，那么这些新值就会丢失。为了解决该问题，需要在用户离开应用时将新值赋给BNRItem的对应属性，并保存修改后的BNRItem。由于encodeRestorableStateWithCoder:会在应用进入后台运行状态时调用，因此修改并保存BNRItem的代码同样可以写在该方法中。

在BNRDetailViewController.m中修改编码方法，代码如下：

```
- (void)encodeRestorableStateWithCoder:(NSCoder *)coder

{

[coder encodeObject:self.item.itemKey

forKey:@“item.itemKey”];

// 保存UITextField对象中的文本

self.item.itemName = self.nameField.text;

self.item.serialNumber = self.serialNumberField.text;

self.item.valueInDollars = [self.valueField.text intValue];

// 保存修改

[[BNRItemStore sharedStore] saveChanges];

[super encodeRestorableStateWithCoder:coder];

}
```

现在BNRDetailViewController可以很好地保存和恢复自身状态。下一节将为BNRItemsViewController添加状态恢复功能。

24.7 保存视图状态

现在BNRItemsViewController中还存在以下问题：

•UITableView没有记录用户最后选中的BNRItem和滚动位置。

•BNRItemsViewController没有记录用户最后是位于正常模式还是编辑模式。每次重新启动应用后，BNRItemsViewController都会恢复到正常模式。

类似于UIViewController，同样可以为UIView及其子类设置恢复标识，并保存需要的状态信息。iOS SDK提供的部分UIView子类可以自动保存某些状态信息，例如UICollectionView、UIImageView、UIScrollView、UITableView、UITextField、UITextView和UIWebView。

Xcode文档中详细介绍了这些类会保存哪些状态信息。为了记录用户最后的滚动位置，可以使用UITableView自动保存的contentOffset属性，恢复UITableView的滚动位置。

先打开BNRItemsViewController.m，为UITableView对象设置恢复标识，代码如下：

```
- (void)viewDidLoad

{

[super viewDidLoad];

// 创建UINib对象，该对象代表包含了BNRItemCell的NIB文件

UINib *nib = [UINib nibWithNibName:@“BNRItemCell” bundle:nil];

// 通过UINib对象，注册相应的NIB文件

[self.tableView registerNib:nib

forCellReuseIdentifier:@“BNRItemCell”];

self.tableView.restorationIdentifier =

@“BNRItemsViewControllerTableView”;

}
```

构建并运行应用，多添加一些BNRItem对象，然后向下滚动UITableView对象。触发状态恢复，再重新启动应用。可以看见，UITableView对象可以正确恢复到之前滚动的位置。

接下来解决第二个问题，保存BNRItemsViewController的编辑状态。打开BNRItemsViewController.m，实现编码和解码方法，代码如下：

```
- (void)encodeRestorableStateWithCoder:(NSCoder *)coder
```

```
{
[coder encodeBool:self.isEditing forKey:@“TableViewIsEditing”];
[super encodeRestorableStateWithCoder:coder];
}
- (void)decodeRestorableStateWithCoder:(NSCoder *)coder
{
self.editing = [coder decodeBoolForKey:@“TableViewIsEditing”];
[super decodeRestorableStateWithCoder:coder];
}
```

下面介绍如何记录用户最后选中的BNRItem。UITableView会自动记录最后选中的UITableViewCell，但无法正确找到UITableViewCell所对应的BNRItem。也就是说，UITableView只会恢复视图对象（每一行的UITableViewCell），但无法自动为其关联正确的模型对象（每一行UITableViewCell所对应的BNRItem）。

在BNRItemsViewController.m的类扩展中，使BNRItemsViewController遵守UIDataSourceModelAssociation协议：

```
@interface BNRItemsViewController ()
<UIPopoverControllerDelegate, UIDataSourceModelAssociation>
@property (nonatomic, strong) UIPopoverController *imagePopover;
@end
```

UIDataSourceModelAssociation协议可以帮助系统在恢复视图对象时为其关联正确的模型对象。当系统保存视图状态时，会同时根据该视图对应模型的所在位置（NSIndexPath）保存一个唯一标识；之后，当恢复视图状态时，也会同时恢复该唯一标识，并找到模型之前的位置。这样，就可以将模型关联到对应位置的UITableViewCell。

在BNRItemsViewController.m中，实现UIDataSourceModelAssociation协议的modelIdentifierForElementAtIndexPath:方法，为选中的BNRItem对象设置唯一标识符（unique identifier），以便系统在恢复应用状态时可以正确找到相应的BNRItem对象。可以使用BNRItem对象的itemKey属性作为唯一标识符，代码如下：

```
- (NSString *)modelIdentifierForElementAtIndexPath:(NSIndexPath *)path
inView:(UIView *)view
```

```
{
NSString *identifier = nil;
if (path && view) {
// 为NSIndexPath参数所对应的BNRItem对象设置唯一标识符
BNRItem *item = [[BNRItemStore sharedStore] allItems][path.row];
identifier = item.itemKey;
}
return identifier;
}
```

接下来实现该协议的另一个对应方法：indexPathForElementWithModelIdentifier:，根据BNRItem对象的唯一标识符返回其所在的NSIndexPath，代码如下：

```
- (NSIndexPath *)indexPathForElementWithModelIdentifier:
(NSString *)identifier inView:(UIView *)view
{
NSIndexPath *indexPath = nil;
if (identifier && view) {
NSArray *items = [[BNRItemStore sharedStore] allItems];
for (BNRItem *item in items) {
if ([identifier isEqualToString:item.itemKey]) {
int row = [items indexOfObjectIdenticalTo:item];
indexPath = [NSIndexPath indexPathForRow:row inSection:0];
break;
}
}
}
```

```
    return indexPath;

}
```

构建并运行应用，触发状态恢复。Homepwner现在可以很好地保存和恢复整个应用的状态，提供了流畅的用户体验。

24.8 中级练习：为另一个应用启用状态恢复

按照本章介绍的步骤，为HypnoNerd应用启用状态恢复。提示：首先需要编码UITextField对象中的文本，其次需要编码UIDatePicker对象中的日期。稍微复杂一些的是BNRHypnosisView对象：为了保存和恢复BNRHypnosisView对象中的所有UILabel对象，需要为其设置恢复标识，再实现相应的编码和解码方法。

24.9 深入学习：设置快照

本章之前介绍过，当应用进入后台运行时，系统会生成当前界面的快照。但是，在某些情况下，系统生成的快照并不符合需求。用户在多任务界面以及下一次启动应用时都可以看到快照，如果应用展示了用户敏感数据（例如一个银行客户端，展示了用户的银行卡号和财务信息），就不能使用展示敏感数据的界面作为快照，否则，非机主本人进入多任务界面或启动应用时，也可以看到敏感数据。

除数据安全性外，还可以通过快照提示用户应用已经进入后台运行状态。例如系统自带的相机（Camera）应用，当应用进入后台运行状态时，应用会“冻结”镜头，并添加毛玻璃效果，避免用户认为应用仍然在等待拍摄。

为了手动设置快照，需要在系统生成快照之前修改界面内容，也就是应用进入未激活状态之前。第18章是通过实现UIApplicationDelegate协议方法观察应用生命周期变化的，而在视图控制器中，可以注册对应的通知，并在收到通知后的回调方法中修改界面内容。

例如，可以观察UIApplicationWillResignActiveNotification通知，生成需要的快照；同时观察UIApplicationDidBecomeActiveNotification通知，当用户返回应用时执行相应的操作（例如，要求用户重新登录，或者“解冻”镜头，让用户继续拍摄）。

```
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];

[nc addObserver:self

selector:@selector(applicationResigningActive:)

name:UIApplicationWillResignActiveNotification

object:nil];

[nc addObserver:self

selector:@selector(applicationBecameActive:)

name:UIApplicationDidBecomeActiveNotification

object:nil];

- (void)applicationResigningActive:(NSNotification *)note

{

// 修改界面内容，生成需要的快照

}

- (void)applicationBecameActive:(NSNotification *)note
```

{

// 返回应用时执行的操作

}

最后，如果不需要让用户在下一次启动应用时看到快照，可以向UIApplication单例发送ignoreSnapshotOnNextApplicationLaunch消息，忽略快照。例如，当应用无法连接网络，向用户提示错误信息时，就应该忽略快照，因为用户在返回应用时可能已经连接了网络。这时应用会使用启动图代替快照。

// 在保存应用状态的代码中添加下行代码

[[UIApplication mainApplication] ignoreSnapshotOnNextApplicationLaunch];

第25章 本地化

使用iOS系统的用户遍及全球各地。这些来自不同国家的用户，使用的语言也不同。通过国际化（internationalization）和本地化（localization）过程，可以确保这些用户都能正常地使用应用。国际化的作用是防止将本土文化信息（native cultural information）写死在应用里（所谓本土文化，是指语言、货币、日期格式、数字格式等）。

本地化的作用是根据用户设置的Language and Region Format（语言和区域格式），为应用提供适当的数据。读者可以在系统的Settings（设置）应用里找到这些设置：先选择General（通用），然后选择International（多语言环境），如图25-1所示。


General
International
Language
English
Voice Control
Keyboards
2
Region Format
United States
Calendar
Gregorian
Region Format Example
Saturday, January 5, 2013
12:34 AM
(408) 555-1212

图25-1 多语言环境设置

国际化和本地化的过程并不复杂。使用本地化API的应用甚至不需要重新编译，就能以其他的语言或区域发布。本章将对Homepwner的BNRDetailViewController实施本地化。顺便提一句，internationalization和localization都是较长的英文单词，所以有时会缩写为i18n和L10n。

25.1 通过NSNumberFormatter实施国际化

本节将使用NSNumberFormatter类为BNRItem的value属性实施数字格式和货币符号的国际化。

实际上，Homepwner中的部分字符串已经实施了国际化。启动Homepwner，添加一个新的BNRItem对象，在添加界面中，BNRDetailViewController中的dateLabel会根据当前区域设置格式化显示日期。在美国，日期的格式是：月 日，年。点击Cancel按钮退出添加界面。

接下来打开设置应用，将Region Format（区域格式）改为United Kingdom（英国）（英文界面：General？International？Region Format；中文界面：通用？多语言环境？区域格式）。然后返回Homepwner并再次添加一个新的BNRItem对象，这次dateLabel显示的日期格式是：日 月 年。由此可知，日期字符串已经国际化了。问题是，之前并没有刻意为该字符串实施国际化，Homepwner是如何做到这点的？

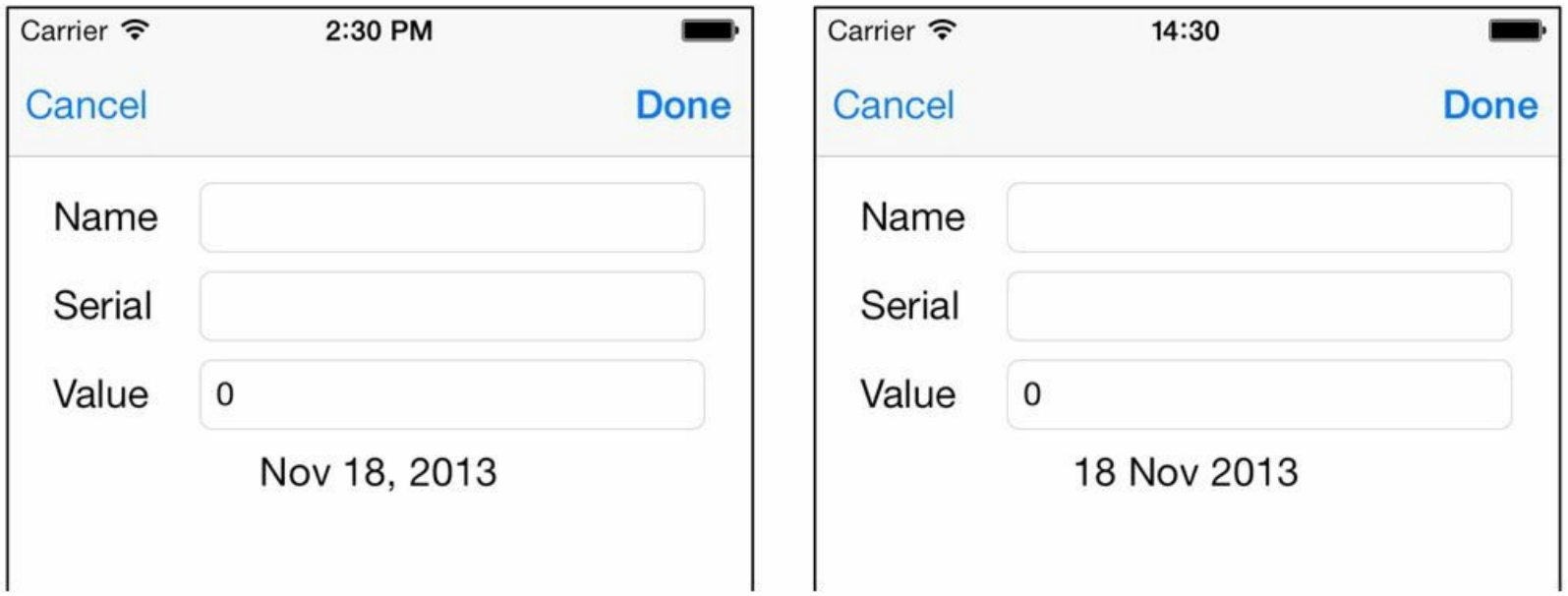


图25-2 日期格式：美国与英国

第10章通过NSDateFormatter对象生成了当前日期字符串，并赋给dateLabel的text属性。NSDateFormatter有一个名为locale的属性，默认会指向代表设备当前区域的NSLocale对象。当Homepwner通过NSDateFormatter对象生成日期字符串时，NSDateFormatter对象会先检查自己的locale属性，然后根据区域格式使用对应格式的字符串。因此，日期字符串从一开始就已经国际化了。

NSLocale对象表示某个区域的本土文化信息，其中包括如何显示符号、日期和小数，以及是否使用公制（metric system）等。用户通过设置应用，可以选择当前的区域（region），例如美国或英国。为什么Apple会在这里使用单词区域（region）而不是国家（country）？这是因为某些国家会有多个区域，各自有不同的设置（设置应用的Region Format（区域格式）表格列出了所有可以选择的区域）。

向NSLocale类发送currentLocale消息，可以得到一个NSLocale对象，该对象表示用户的当前区域设置。通过NSLocale对象，可以获取这类信息：“该区域的货币符号是什么？”或“该区域使

用的是公制吗？”

要获取此类信息，需要向NSLocale对象发送objectForKey:消息，并传入相应的NSLocale常量（读者可以在NSLocale的类参考手册中找到所有可用的常量）。代码如下：

```
NSLocale *locale = [NSLocale currentLocale];

BOOL isMetric = [[locale objectForKey:NSLocaleUsesMetricSystem] boolValue];

NSString *currencySymbol = [locale objectForKey:NSLocaleCurrencySymbol];
```

下面对BNRItemCell中显示的值实施国际化。打开Homepwner.xcodeproj。

NSLocale功能强大，但是直接使用会比较烦琐，因此Apple提供了更方便的类，例如之前使用的NSDateFormatter。类似的还有NSNumberFormatter，可以根据区域设置格式化显示数字。NSNumberFormatter提供了stringFromNumber:方法，用于返回当前区域格式的数字，如123, 456.789或123 456, 789等。

```
NSNumberFormatter *numberFormatter = [[NSNumberFormatter alloc] init];

NSString *numberAsString = [numberFormatter stringFromNumber:@123456.789];
```

NSNumberFormatter对象还可以格式化显示货币。如果将NSNumberFormatter对象的numberStyle属性设置为NSNumberFormatterCurrencyStyle，NSNumberFormatter对象不但会格式化数字，而且会添加当前区域格式的货币符号。（在某些国家，NSNumberFormatter在数字格式与货币格式下返回的字符串可能有很大差异，不仅仅是多了一个货币符号。）

```
NSNumberFormatter *currencyFormatter = [[NSNumberFormatter alloc] init];

currencyFormatter.numberStyle = NSNumberFormatterCurrencyStyle;

NSString *numberAsString = [currencyFormatter stringFromNumber:@123456.789];
```

在BNRItemsViewController.m的tableView:cellForRowAtIndexPath:方法中添加一个静态的NSNumberFormatter对象currencyFormatter，并将numberStyle设置为NSNumberFormatterCurrencyStyle。代码如下：

```
cell.serialNumberLabel.text = item.serialNumber;

// 创建一个静态NSNumberFormatter对象

static NSNumberFormatter *currencyFormatter = nil;

if (currencyFormatter == nil) {

currencyFormatter = [[NSNumberFormatter alloc] init];
```

```
currencyFormatter.numberStyle = NSNumberFormatterCurrencyStyle;

}

cell.valueLabel.text = [NSString stringWithFormat:@"$%d",

item.valueInDollars];
```

目前BNRItemCell使用"$%d"格式生成表示物品价值的字符串。这样会导致即使当前区域格式不是美国(United States)，物品价值仍然会显示美元符号。下面改用currencyFormatter生成当前区域格式的物品价值，代码如下：

```
// 为货币创建一个NSNumberFormatter

static NSNumberFormatter *currencyFormatter = nil;

if (currencyFormatter == nil) {

currencyFormatter = [[NSNumberFormatter alloc] init];

currencyFormatter.numberStyle = NSNumberFormatterCurrencyStyle;

}

~~cell.valueLabel.text = [NSString stringWithFormat:@"$%d",~~

~~item.valueInDollars];~~

cell.valueLabel.text = [currencyFormatter

stringFromNumber:@(item.valueInDollars)];

cell.thumbnailView.image = item.thumbnail;
```

构建并运行应用。如果读者从一开始就按照本书的步骤操作，那么现在BNRItemCell显示的物品价值格式应该是United Kingdom(英国)。接下来在设置应用中将区域格式重新改为United States(美国)，再返回Homepwner。

读者也许会以为BNRItemCell会恢复到美国格式，显示以美元($)为单位的物品价值，但是BNRItemCell并没有立刻更新物品价值的格式。这时需要重新加载UITableView对象的数据。可以先进入添加界面再取消添加，这样BNRItemsViewController会收到viewWillAppear:消息并调用UITableView对象的reloadData方法，更新BNRItemCell。这时BNRItemCell会正确地显示以美元为单位的物品价值(注意，这里仅仅是替换了符号，并没有进行英镑到美元的汇率转换)。

如果要及时响应当前区域设置的变化，可以向NSNotificationCenter注册当前区域设置发生变化的通知：NSCurrentLocaleDidChangeNotification。在BNRItemsView- Controller对象的init方法中，将BNRItemsViewController对象注册为该通知的观察者：

```
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
[nc addObserver:self
selector:@selector(updateTableViewForDynamicTypeSize)
name:UIContentSizeCategoryDidChangeNotification
object:nil];
// 注册当前区域设置发生变化的通知
[nc addObserver:self
selector:@selector(localeChanged:)
name:NSCurrentLocaleDidChangeNotification
object:nil];
}
return self;
```

然后添加localeChanged:方法，代码如下：

```
- (void)localeChanged:(NSNotification *)note
{
[self.tableView reloadData];
}
```

构建并运行应用。先在设置应用中修改当前的区域格式，再返回Homepwner。这次BNRItemCell会立刻更新物品价值的显示格式。

之前介绍过使用NSLocale对象获取当前区域设置的货币符号，读者可能会想到直接使用该符号拼接一个具体数值作为货币字符串。实际上，不同区域的货币格式可能不仅仅是货币符号不同，为了理解使用NSNumberFormatter的好处，请读者将区域格式改为德国(Germany)。从图25-3可以发现，除了货币符号之外，还有一些细节也发生了变化：①货币符号的位置(德国：货币符号位于数值之后；美国或英国：货币符号位于数值之前)。②空格(德国：数值与货币符号之间有一个空格；美国或英国：数值与货币符号之间没有空格)。③小数点符号(德国：逗号；美国或英国：点号)。④千位分隔符(德国：点号；美国或英国：逗号)。

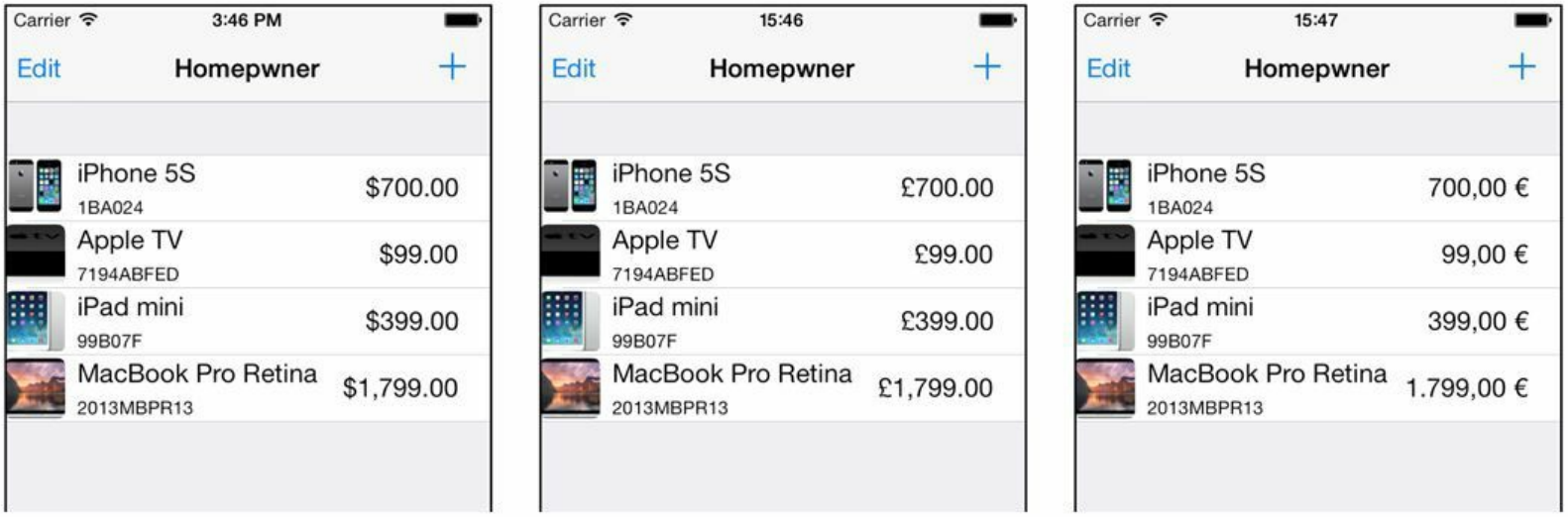


图25-3 数字格式：美国、英国和德国

25.2 资源的本地化

实施国际化时，需要通过NSLocale对象获取相应的区域信息。NSLocale只包含和区域有关的属性，如果需要“国际化”应用其他的资源，就需要实施本地化。本地化是指根据用户的区域或语言设置，创建相应的应用资源的过程。本地化通常意味着以下两件事情。

•针对不同的区域和语言，创建多个版本的资源，例如图片、声音和界面。

•创建并获取字符串对照表（strings table），将界面文字翻译成不同的语言。

任何资源文件，无论是图片文件还是XIB文件，都可以实施本地化。Xcode在针对某种语言本地化某个资源文件时，会复制一份该资源的拷贝，并将该拷贝加入应用程序包。此外，Xcode会根据语言将这些本地化后的资源文件存放在特定的目录中。这些目录的目录名就是对应的语言和区域的代码，后缀都是lproj。以美式英语为例，针对美式英语的语言代码是en_US：其中的en代表英语，US代表美国（如果资源文件无须区分区域，就可以去掉代码中的区域部分）。这类语言和区域的代码不是iOS特有的，而是一套跨平台的标准。

当应用通过NSBundle对象获取某个资源文件的路径时，该对象会根据指定的文件名先在程序包的根目录下查找匹配的文件。如果没有找到，NSBundle对象就会根据设备当前的语言设置在相应的lproj目录下继续查找。因此，只要完成资源文件的本地化，应用就能根据当前的语言设置自动载入匹配的文件。

但是，随着需要支持的语言数量越来越多，对各个语言的资源文件进行维护也会越来越困难。例如XIB文件，假设每一种语言所对应的lproj目录下都包含独立的XIB文件，如en.lproj/BNRDetailViewController.xib和es.lproj/BNRDetailViewController. xib等，那么，一旦修改了英文版的XIB（增加新的UILabel对象或UIButton对象），就要手动更新所有其他语言版本。

Xcode提供了一项称为基础国际化（Base Internationalization）的功能，可以轻松为XIB文件添加多语言支持。当项目启用基础国际化之后，Xcode会自动创建一个包含主XIB文件的Base.lproj目录。这样，项目中就不需要为每一种语言都创建一个独立的XIB文件，只要在主XIB文件的基础上创建包含对应语言字符串的Localizable.string文件就可以了。当然，如果创建Localizable.string文件无法满足要求，仍然可以创建独立的XIB文件——实际上这种情况并不常见，如果只是在替换了Localizable.string文件中的字符串后界面布局发生了问题，那么可以使用自动布局系统。

本节要对Homepwner的BNRDetailViewController.xib文件实施本地化，创建英语（English）和西班牙语（Spanish）的本地化资源。完成后的程序包会增加一个Base.lproj目录和两个对应语言的lproj目录。一般来说，首先应该为项目启用基础国际化，再添加其他语言的本地化资源。但是，截止本书写作期间，Xcode有一个bug：必须先对项目中的至少一个XIB文件实施了本地化，才可以启用基础国际化。

因此，下面首先对BNRDetailViewController.xib实施本地化。在项目导航面板中选择BNRDetailViewController.xib并打开工具区域。

点击检视面板选择条中的按钮，打开文件检视面板（file inspector）。找到面板中名为Localization的部分，单击Localize…按钮（见图25-4）。

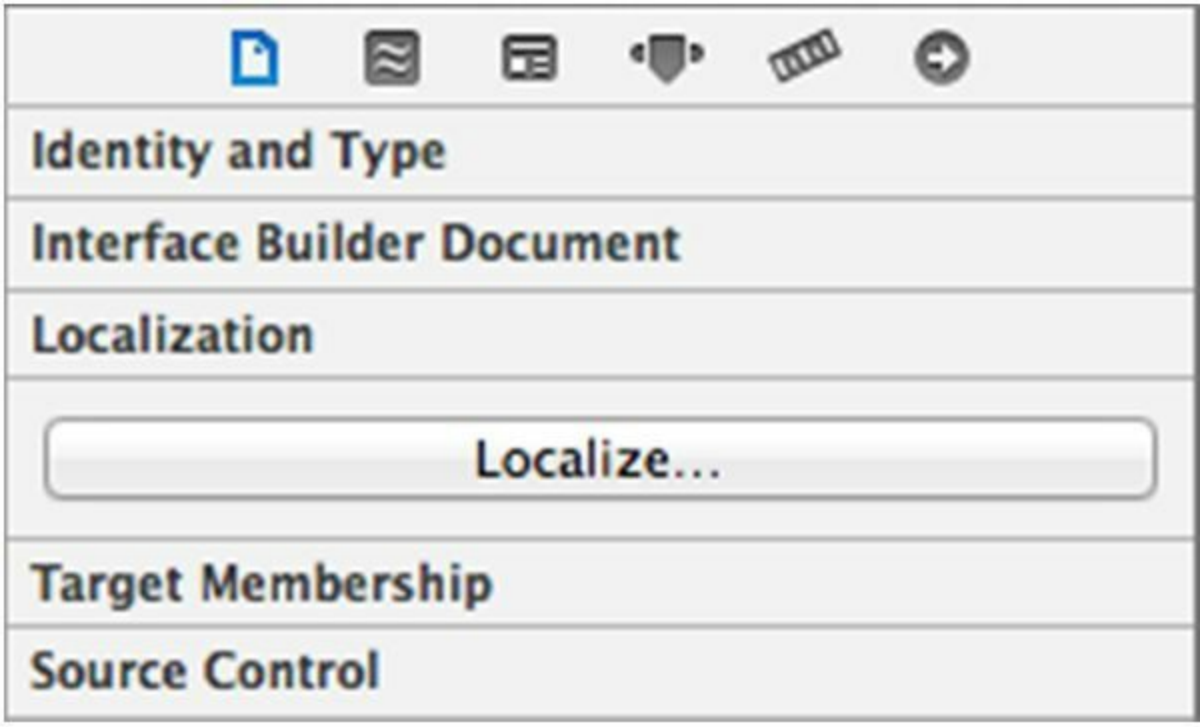


图25-4 开始对BNRDetailViewController.xib实施本地化

在弹出的下拉列表中选择English。Xcode会自动创建一个en.lproj目录，然后将BNRDetailViewController.xib文件移入该目录。

现在可以为项目启用基础国际化了。首先在项目导航面板顶部选择Homepwner项目文件，然后在编辑区域的项目和目标列表中选择Homepwner项目（注意不要选成了Homepwner目标），如图25-5所示。

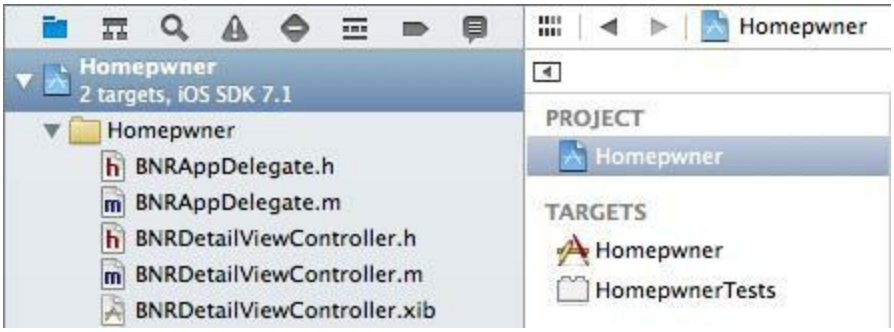


图25-5 选择项目信息

接下来打开Info标签页，在底部找到Localizations部分的Use Base Internationalization（启用基础国际化）选择框并将其选中。这时Xcode会弹出下拉窗口，提示选择需要实施基础国际化的资源文件（Resource File）与参考语言（Reference Language），文件列表中应该只有BNRDetailViewController.xib与English。直接点击Finish按钮。Xcode会创建一个Base.lproj目录，然后将BNRDetailViewController.xib文件移入该目录。

下面添加对西班牙语的支持。点击Localizations部分下方的+按钮，选择Spanish。Xcode会弹出与之前类似的下拉窗口，在文件列表中，读者可以不选择InfoPlist.strings，只需要选择BNRDetailViewController.xib。确保参考语言是Base，文件类型是Localizable Strings，然后点击Finish按钮。Xcode会创建一个es.lproj目录，并在该目录下生成一个

BNRDetailViewController.strings文件，该文件包含基础BNRDetailViewController. xib中的所有字符串。现在Localizations部分应该类似于图25-6。

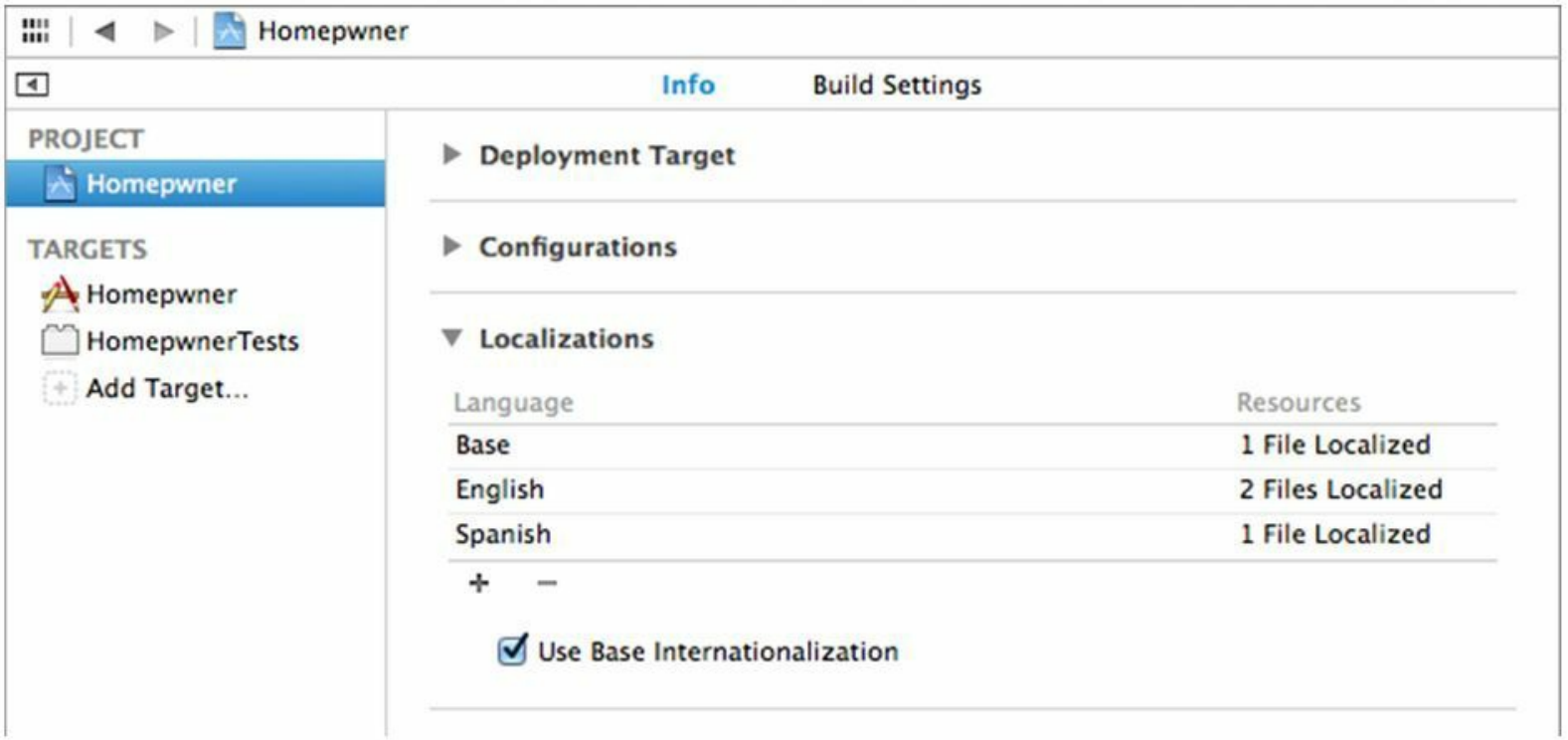


图25-6 本地化

在项目导航面板中，点击BNRDetailViewController.xib左边的三角形按钮（见图25-7）。Xcode已经将BNRDetailViewController.xib文件移动到了Base.lproj目录中，并在es.lproj目录中创建了BNRDetailViewController.strings文件。

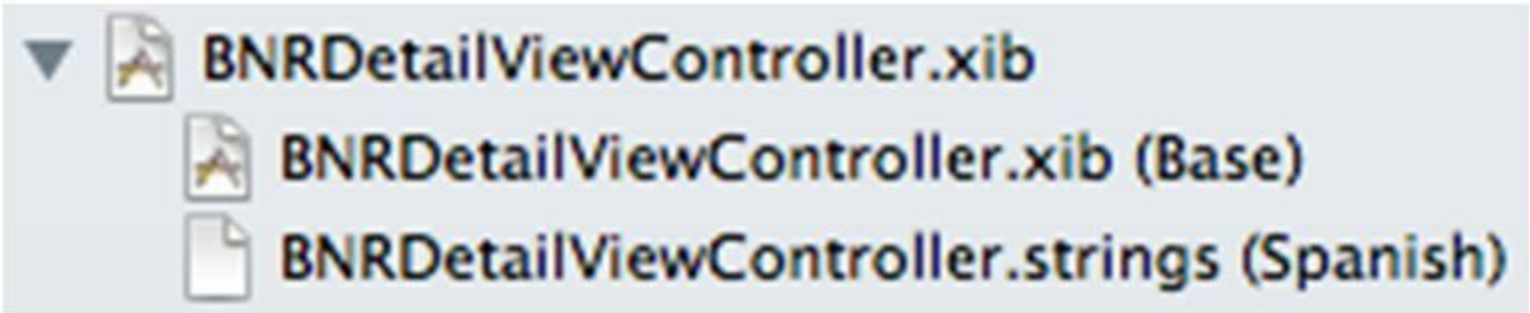


图25-7 项目导航面板所显示的已经本地化的XIB文件

选择BNRDetailViewController.xib（选择有或没有Base标识的都可以），文件检视面板应该类似于图25-8。

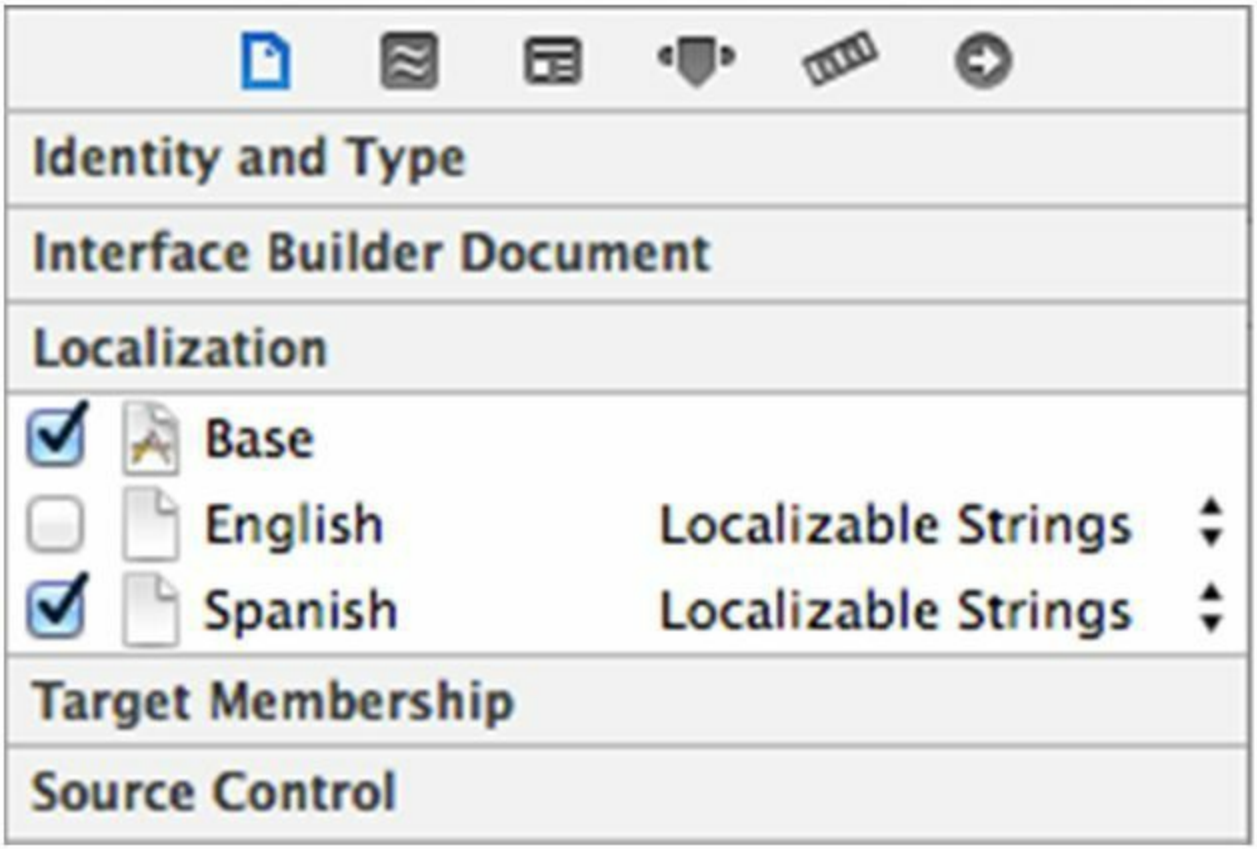


图25-8 对BNRDetailViewController.xib实施了本地化

在项目导航面板中选择西班牙语版本的BNRDetailViewController.strings，可以看到其中的文字仍然是英文。Xcode不会自动翻译文字，读者必须自己动手。

可以根据以下粗体文本内容编辑BNRDetailViewController.strings。其中的ObjectID和排列顺序可能会和读者生成的文件不同，可以用注释部分的text或title字段匹配翻译内容。

/* Class = “IBUILabel”; text = “Serial”; ObjectID = “JkL-nP-h3R”; */

“JkL-nP-h3R.text” = “Numéro de serie”;

/* Class = “IBUILabel”; text = “Label”; ObjectID = “Q5n-Bc-7IH”; */

“Q5n-Bc-7IH.text” = “Label”;

/* Class = “IBUILabel”; text = “Name”; ObjectID = “qzL-Fn-qch”; */

“qzL-Fn-qch.text” = “Nombre”;

/* Class = “IBUILabel”; text = “Value”; ObjectID = “rhE-7e-oTE”; */

“rhE-7e-oTE.text” = “Valor”;

/* Class = “IBUIBarButtonItem”; title = “Item”; ObjectID = “uNg-wM-Zcr”; */

“uNg-wM-Zcr.title” = “Item”;

因为BNRDetailViewController对象会在运行时动态设置text为Label的UILabel对象以及title为Item的UIBarButtonItem对象，所以这里没有翻译相应的字符串。保存BNRDetailViewController.strings。

以上完成了BNRDetailViewController.xib的本地化过程，下面做一个测试。在测试之前，要提醒读者留意Xcode的一个小问题：Xcode在构建应用时，有时会忽略某个资源文件的变化，从而导致生成的程序包没有更新相应的文件。要让Xcode彻底重新构建应用，需要先从设备或模拟器中删除应用（长按应用图标，当图标开始抖动时点击左上角的删除按钮），然后选择Product菜单中的Clean选项。这样可以强制Xcode彻底重新编译、构建并安装应用（如果还有问题，可以在按住Option键的同时打开Product菜单，然后选择Clean Build Folder…）。

在设置应用中，将语言修改为Español（英文界面：General⇒nternational⇒Language；中文界面：通用⇒多语言环境⇒语言）。重新启动Homepwner，选中某个BNRItem对象，应该能看到西班牙语版的界面。但是，界面中有些UILabel对象无法显示全部文字（见图25-9左）。

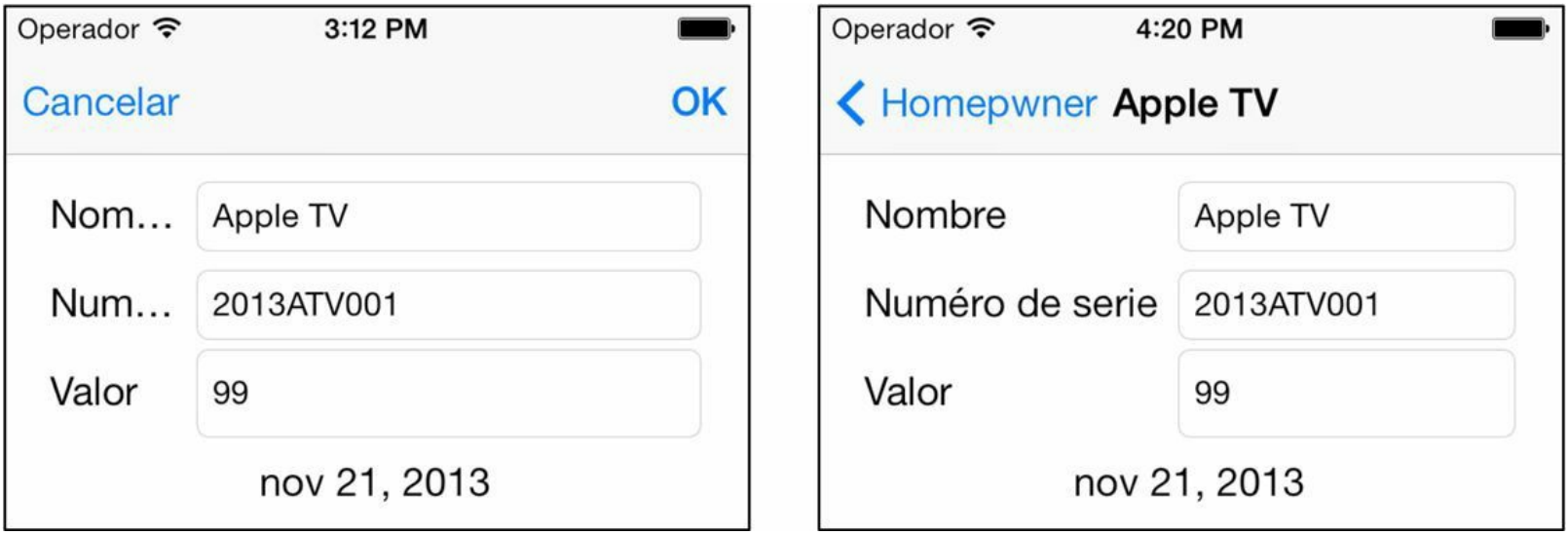


图25-9 西班牙语版的BNRDetailViewController.xib使用自动布局的前后对比

使用自动布局系统可以轻松解决这个问题。打开BNRDetailViewController.xib，先选中nameLabel，然后在工具区域中点击图标，打开大小检视面板，并找到nameLabel的宽度约束。目前该约束限定nameLabel的宽度是55点，无法针对较长的西班牙语单词自动调整宽度。点击宽度约束的齿轮按钮，在弹出菜单中选择Select and Edit…，在约束的编辑界面将Relation由Equal（等于）改为Greater Than or Equal（大于或等于）。

接下来修改serialNumberLabel和valueLabel的约束。为了使三个UITextfield对象的宽度保持一致，三个UILabel对象的宽度也必须保持一致。首先选中serialNumberLabel，删除其宽度约束。然后按住Control键，将其拖曳到nameLabel，在弹出菜单中选择Equal widths（宽度相等）。接下来使用相同的步骤，确保valueLabel的宽度与nameLabel的宽度保持相等。构建并运行应用，现在UITextfield对象会自动调整宽度，为前面的UILabel对象留出足够的显示空间，因此UILabel对象可以显示全部文字了（见图25-9右）。

25.3 NSLocalizedString()与字符串对照表

编写iOS应用时，往往需要动态地创建NSString对象，或者在界面中显示字符串常量。为了能显示这些字符串的多语言版本，需要创建一个字符串对照表(strings table)。字符串对照表其实就是文本格式的文件，其中包含一组键-值对。这里的键是需要支持多语言的字符串，值是相应的翻译。字符串对照表是一种需要加入程序包的资源文件，而且在编写iOS应用时，不需要编写复杂的代码就能得到字符串对照表中的数据。

代码中的字符串示例如下：

```
NSString *greeting = @“Hello!”
```

为了能够本地化代码中的字符串，需要用NSLocalizedString()宏(macro)替换字符串常量。

```
NSString *greeting =

NSLocalizedString(@“Hello!”, @“The greeting for the user”);
```

使用NSLocalizedString()宏时需要传入两个实参：第一个实参是键(必需)，第二个实参是注释(可选)。键是字符串对照表的查询值(lookup value)。运行时，NSLocalizedString()会先遍历程序包中的所有字符串对照表，找出和设备当前的语言设置相匹配的那个。然后在该文件中找出和相应的键匹配的字符串翻译。

Homepwner的导航条会显示字符串“Homepwner”，下面要本地化该字符串。更新BNRItemsViewController.m中的init方法，修改设置navigationItem标题的那行代码。

```
-(instancetype)init

{

// 调用父类的指定初始化方法

self = [super initWithStyle:UITableViewStylePlain];

if (self) {

UINavigationItem *navItem = [self navigationItem];

~~navItem.title = @“Homepwner”;~~

navItem.title = NSLocalizedString(@“Homepwner”, @“Name of application”);
```

还有两个视图控制器包含硬编码的字符串，需要实施国际化：BNRDetailViewController工具栏中用于显示资产类型的assetTypeButton标题和BNRAssetTypeViewController导航栏的标题。

在BNRDetailViewController.m中更新viewWillAppear:方法，代码如下：

```
NSString *typeLabel = [self.item.assetType valueForKey:@“label”];
if（！typeLabel）{
typeLabel = @“None”;
typeLabel = NSLocalizedString(@“None”, @“Type label None”）;
}
self.assetTypeButton.title =
[NSString stringWithFormat:@“Type: %@”, typeLabel];
self.assetTypeButton.title = [NSString stringWithFormat:
NSLocalizedString(@“Type: %@”, @“Asset type button”）， typeLabel];
[self updateFonts];
}
```

在BNRAssetTypeViewController.m中更新init方法，代码如下：

```
if（self）{
self.navigationItem.title = @“Asset Type”;
self.navigationItem.title =
NSLocalizedString(@“Asset Type”,
@“BNRAssetTypeViewController title”）;
}
return self;
}
```

用NSLocalizedString本地化某个实现文件后，就可以通过命令行程序genstrings自动生成相应的字符串对照表。

打开终端应用（Terminal.app）——也许读者之前从未使用过终端应用。终端应用其实就是一个Unix终端（OS X本质上是一个Unix操作系统），用于运行命令行工具。请读者将当前目录切换至BNRItemsViewController.m所在的目录。如果读者不熟悉Unix终端命令，可以按照以下步骤切换当前目录：首先在终端应用的窗口中输入cd，然后输入一个空格（注意此时不要按下回车键）：

cd

接下来打开Finder，前往包含BNRItemsViewController.m的目录，将该目录的图标（蓝色的文件夹）从Finder拖曳至终端应用。松开鼠标后，终端应用会自动填入该目录的文件系统路径。现在按下回车键。

这样，终端应用的当前目录就已经切换至BNRItemsViewController.m所在的目录。以作者的BNRItemsViewController.m目录为例，相应的终端命令如下：

cd /Users/aaron/Homepwner/Homepwner/

切换当前目录后，可以使用ls命令输出目录下的内容列表，请读者确认列表中是否包含BNRItemsViewController.m。

如果列表中确实包含BNRItemsViewController.m，就可以在终端应用中输入以下命令并按回车键，生成字符串对照表：

genstrings BNRItemsViewController.m

genstrings会在BNRItemsViewController.m所在的目录中创建一个名为Localizable. strings的文件。使用同样的方法，可以为另外两个视图控制器生成字符串。注意，由于Localizable.strings文件已经存在，因此需要将新的字符串附加到该文件，而不是创建新的Localizable.strings文件。在终端应用中输入以下命令（不要忘记添加-a选项），每行命令之后都需要按下回车键：

genstrings -a BNRDetailViewController.m

genstrings -a BNRAssetTypeViewController.m

现在Localizable.strings文件包含以上三个视图控制器中的国际化字符串。接下来将该文件从Finder中拖曳至项目导航面板并加入项目（或者右击项目文件，选择Add Files to “Homepwner” …菜单项，再选择Localizable.strings文件）。Xcode会在构建应用时将该文件复制到应用程序包中。

Xcode偶尔会无法正确显示字符串对照表。在项目导航面板中选中Localizable. strings，Xcode会在编辑器区域显示Localizable.strings。如果读者看到的是乱码，就要让Xcode以Unicode（UTF-16）编码重新解析（reinterpret）该文件。打开工具区域，选择文件检视面板，找到Text Settings（文本设置）区域，将标签为Text Encoding（文本编码）的弹出菜单设置为Unicode（UTF-16）（见图25-10）。Xcode会弹出对话框，并询问是reinterpret（重新解析）还是convert（转换），选择reinterpret。

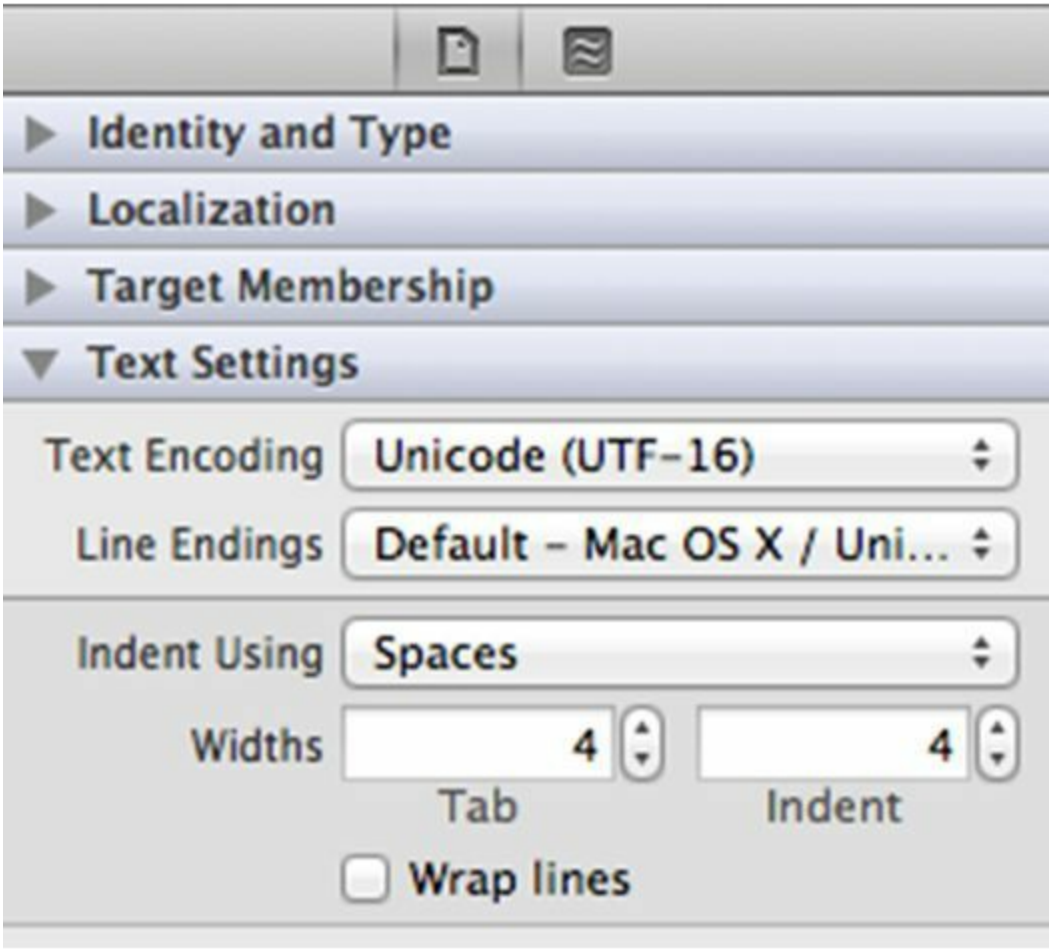


图25-10 修改文件编码

编辑器区域中的Localizable.strings应该显示如下内容：

/* Name of application */

“Homepwner” = “Homepwner”;

/* BNRAssetTypeViewController title */

“Asset Type” = “Asset Type”;

/* Type label None */

“None” = “None”;

/* Asset type button */

“Type: %@” = “Type: %@”;

字符串对照表中的注释源自传入NSLocalizedString的第二个实参。虽然第二个实参不会对NSLocalizedString起任何作用，但是这些注释能在稍后的翻译阶段起辅助提示的作用。

前面介绍了如何为XIB文件创建其他语言的版本，Localizable.strings也一样。在项目导航面板中选中Localizable.strings，点击工具区域的Localize…按钮，加入西班牙语版本的

Localizable.strings。选中新创建的Localizable.strings，Xcode会在编辑器区域显示该文件，位于编辑器区域左侧的字符串是键（即传入NSLocalizedString的第一个实参），位于右侧的是等待翻译的字符串（即NSLocalizedString的返回值）。将右侧的文字翻译成西班牙语（“n?”的输入方法是按下Option键的同时按下“n”），内容如下：

```
/* Name of application */

“Homepwner” = “Duen?o de casa”

/* AssetTypePicker title */

“Asset Type” = “Tipo de activo”;

/* Type label None */

“None” = “Nada”;

/* Asset type button */

“Type: %@” = “Tipo: %@”;
```

构建并运行应用，此时所有字符串都应该会以西班牙语显示。如果不是，则可以尝试删除应用，对项目执行清理（clean）操作并重新构建（或检查系统的语言设置）。

25.4 初级练习：再添加一套本地化资源

熟能生巧。为Homepwner增加一系列其他语言的本地化资源。如果在语言中遇到问题，则可以借助Google翻译。

25.5 深入学习：NSBundle在国际化过程中的作用

增加本地化资源的很多工作其实是由NSBundle完成的。以UIViewController为例，初始化某个UIViewController对象时需要传入两个实参：第一个实参是XIB文件的文件名，第二个实参是指向某个NSBundle对象的指针。通常会将nil作为第二个实参传入，UIViewController会将这个nil解释为应用的主程序包（main bundle）。主程序包其实就是应用程序包，包含应用的所有资源和可执行文件。Xcode在构建应用时，会将所有的lproj目录复制到主程序包。

视图控制对象在载入视图时，会通过之前指定的NSBundle对象查找相应的XIB文件。NSBundle对象会根据设备当前的语言设置在相应的lproj目录中查找指定的文件，并将找到的路径返回给视图控制对象，以供其载入。

通过NSBundle的实例方法pathForResource:ofType:可以在所有的lproj目录中查找指定类型的资源文件。如果需要得到应用程序包中的某个资源文件的路径，就可以向主程序包发送这个消息。以下代码演示了如何得到资源文件myImage.png的路径：

```
NSString *path = [[NSBundle mainBundle] pathForResource:@“myImage”

ofType:@“png”];
```

NSBundle对象会先检查程序包的根目录，查找是否有myImage.png文件。如果有，就返回该文件的全路径。如果没有，NSBundle对象就会根据设备的当前语言设置在相应的lproj目录下继续查找。如果还是没有找到，就返回nil。

这也是为什么在本地化某个文件后，必须删除已经安装的应用并清理项目。Xcode不会在重新安装应用时删除任何文件，因此，即使某个文件已经本地化了，只要是覆盖安装的程序包，其根目录仍然会保留之前尚未本地化的版本。在这种情况下，即使程序包里有新的lproj目录，NSBundle对象还是会先找到根目录下的文件并返回该文件的路径。

25.6 深入学习：不通过基础国际化对XIB文件实施本地化

Xcode在具备基础国际化功能之前，无法对各个字符串实施本地化。为了避免为每一种语言创建独立的XIB文件，当时iOS开发者使用的是命令行工具ibtool，具体流程为：首先，通过ibtool，从原版语言的XIB文件中抽取字符串；然后，翻译这些字符串；最后，根据翻译后的字符串创建新的XIB文件。

下面介绍ibtool的具体用法。首先打开终端应用，然后将当前目录切换到en.lproj所在的目录。以作者的系统为例，相应的终端命令如下：

```
cd /Users/aaron/Homepwner/Homepwner/en.lproj
```

下面要用ibtool抽取BNRDetailViewController.xib文件中的字符串。在终端中输入以下指令（必须在同一行中输入，这里分行是为了方便排版）：

```
ibtool --export-strings-file ~/Desktop/BNRDetailViewController.strings

BNRDetailViewController.xib
```

执行完这条命令后，ibtool会在桌面上创建一个BNRDetailViewController.strings文件，其中包含BNRDetailViewController.xib文件中的所有字符串。打开西班牙语版的BNRDetailViewController.strings文件，该文件的内容与之前通过基础国际化创建的本地化字符串列表基本一致。根据之前介绍的步骤，将列表中的字符串翻译成西班牙语。

下面要用ibtool创建一个新的西班牙语版的XIB文件。该文件将以英文版的BNRDetailView-Controller.xib为模板，并将其中的字符串都替换为BNRDetailViewController.strings中的值。为此，需要得到（项目目录下的）“英文版的XIB文件的路径”和“西班牙语版的XIB文件的路径”。请读者在Finder中打开这两个目录。

在终端应用中输入以下命令，在末尾的单词write后输入一个空格（但是不要按回车）。

```
ibtool --import-strings-file ~/Desktop/BNRDetailViewController.strings

--write
```

先将es.lproj中的BNRDetailViewController.xib拖曳至终端窗口，再将en. lproj中的BNRDetailViewController.xib拖曳至终端窗口。这时的命令行示例如下：

```
ibtool --import-strings-file ~/Desktop/BNRDetailViewController.strings

--write

/iphone/Homepwner/Homepwner/es.lproj/BNRDetailViewController.xib

/iphone/Homepwner/Homepwner/en.lproj/BNRDetailViewController.xib
```

这条命令的作用是基于en.lproj中的BNRDetailViewController.xib，创建新的

BNRDetailViewController.xib并存入es.lproj目录。然后用BNRDetailViewController. strings中的值替换新创建的BNRDetailViewController.xib中的字符串。

按下回车键(ibtool可能会向终端输出若干警告信息，可忽略)。

在项目导航面板中选中BNRDetailViewController.xib(Spanish)，可以看到相应的字符串已经本地化了西班牙语，如图25-11所示。最后，根据之前介绍的步骤，修改三个UILabel对象的约束，调整界面布局。

图25-11 西班牙语版的BNRDetailViewController.xib

第26章 NSUserDefaults

对于应用来说，每个用户都有自己特定的喜好与使用习惯。优秀应用会在用户使用过程中逐渐记录用户的使用习惯，让用户使用起来更加得心应手；或者，应用需要针对某些功能提供一组设置选项，让用户根据自己的喜好选择应用的使用方式。这类用户使用信息与应用设置统称为用户偏好设置（user preference）。那么，用户偏好设置应该存储到什么位置呢？每一个应用包中都有一个plist文件，可以方便地存储用户偏好设置。在代码中，可以通过NSUserDefaults类访问该plist文件。另外，还可以为应用创建一个设置束（settings bundle），在设置应用中注册用户偏好设置列表，用户可以在该列表中修改偏好设置。

本章将彻底完成Homepwner应用，添加读取和存储用户偏好设置的功能，还会为Homepwner应用创建一个设置束。

26.1 NSUserDefaults

NSUserDefaults提供了一个类方法standardUserDefaults，用于获取NSUserDefaults单例。NSUserDefaults单例以键-值对的形式存储了一系列用户偏好设置。其中，键是用户偏好设置的名称，值是对应的某类数据。与NSDictionary类似，可以通过objectForKey:和setObject:forKey:存取数据。

```
NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];

NSString *greeting = [defaults objectForKey:@“FavoriteGreeting”];

NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];

[defaults setObject:@“Hello” forKey:@“FavoriteGreeting”];
```

对于setObject:forKey:，NSUserDefaults会自动将对象存储到对应的plist文件中——显然，对象的类型必须是plist文件可以存储的类型，包括：NSArray、NSDictionary、NSString、NSData、NSDate和NSNumber。如果需要存储plist文件不支持的类型，可以先将其归档为NSData类型，再存入plist文件。

如果通过objectForKey:获取一项没有值的用户偏好设置，NSUserDefaults会返回该项的出厂设置，也称为默认设置。这类设置并没有存储在plist文件中，需要在每次应用启动时向NSUserDefaults注册。为了确保在代码中访问NSUserDefaults之前注册好出厂设置，可以在应用程序委托中覆盖类方法initialize，代码如下：

```
+（void）initialize

{

NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];

NSDictionary *factorySettings = @{@“FavoriteGreeting”: @“Hey!”,

@“HoursBetweenMothershipConnection : @2};

[defaults registerDefaults:factorySettings];

}
```

Objective-C运行环境（Objective-C runtime）会在创建某个类的第一个对象之前调用该类的initialize方法。

本节将为Homepwner添加一项用户偏好设置，可以在添加BNRItem对象时默认显示用户设置的名称和价值。

注册出厂设置

应用启动时，首先需要向NSUserDefaults注册出厂设置。通过NSUserDefaults对象存取数据时，使用的键必须是NSString对象。通常情况下，可以将这类字符串定义成静态变量。这样就不用每次使用写死的字符串，直接使用相应的变量即可。使用静态变量有两个好处，一是方便修改，二是可以避免输入错误（如果输错了变量名，编译器就会发出警告；但是，如果输错了字符串，则不会有任何提示）。

打开BNRAppDelegate.h，声明两个NSString类型的静态变量，代码如下：

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

extern NSString * const BNRNextItemValuePrefsKey;

extern NSString * const BNRNextItemNamePrefsKey;

@interface BNRAppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;

@end
```

再打开BNRAppDelegate.m，实现两个静态变量，然后将其作为键在initialize方法中注册出厂设置：

```
#import "BNRAppDelegate.h"

#import "BNRItemsViewController.h"

#import "BNRItemStore.h"

NSString * const BNRNextItemValuePrefsKey = @"NextItemValue";

NSString * const BNRNextItemNamePrefsKey = @"NextItemName";

@implementation BNRAppDelegate

+ (void)initialize

{

NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];

NSDictionary *factorySettings = @{BNRNextItemValuePrefsKey: @75,

BNRNextItemNamePrefsKey: @"Coffee Cup"};
```

```
[defaults registerDefaults:factorySettings];
}
```

读取用户偏好设置

下面在BNRItemStore.m的createItem方法中为BNRItem对象的属性设置默认值。注意，为了让编译器知道BNRNextItemValuePrefsKey常量，必须先在BNRItemStore.m顶部导入BNRAppDelegate.h。

```
-（BNRItem *）createItem
{
double order;
if（_allItems.count == 0）{
order = 1.0;
} else {
order = [[self.privateItems lastObject] orderingValue] + 1.0;
}
BNRItem *item =
[NSEntityDescription insertNewObjectForEntityForName:@”BNRItem“
inManagedObjectContext:self.context];
item.orderingValue = order;
NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
item.valueInDollars = [defaults integerForKey:BNRNextItemValuePrefsKey];
item.itemName = [defaults objectForKey:BNRNextItemNamePrefsKey];
// 查看NSUserDefaults中存储了哪些数据
NSLog（@”defaults = %@“, [defaults dictionaryRepresentation]）;
[self.privateItems addObject:item];
```

return item;

}

NSUserDefaults提供了若干简便方法，可以直接存取某些常用类型的值，例如以上代码中的integerForKey:。其他还有float、double、BOOL和NSURL等。以下两行代码的效果是相同的：

item.valueInDollars = [defaults integerForKey:BNRNextItemValuePrefsKey];

item.valueInDollars =

[[defaults objectForKey:BNRNextItemValuePrefsKey] intValue];

修改用户偏好设置

为应用提供修改用户偏好设置的界面很简单，可以在应用内创建相应的视图控制器，也可以为应用创建设置束，但是记录用户的使用习惯有一定的技巧，需要考虑应用特定的使用场景并推测用户的下一步操作。例如，假设用户设置了某个BNRItem对象的价值是100元，那么，用户可能将下一个BNRItem对象的价值也设置为100元。像这类用户使用信息就可以存入NSUserDefaults，方便用户再次使用。

打开BNRDetailViewController.m，导入BNRAppDelegate.h，然后修改viewWill- Disappear:，代码如下：

-（void）viewWillDisappear:（BOOL）animated

{

[super viewWillDisappear:animated];

[self.view endEditing:YES];

BNRItem *item = self.item;

item.itemName = self.nameField.text;

item.serialNumber = self.serialNumberField.text;

int newValue = [self.valueField.text intValue];

// 比较valueInDollars属性与valueField中的新值，判断是否有改动

if（newValue != item.valueInDollars）{

// 如果有改动，将新值赋给valueInDollars属性

```
    item.valueInDollars = newValue;

    // 将新值存入NSUserDefaults

    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];

    [defaults setInteger:newValue

    forKey:BNRNextItemValuePrefsKey];

    }

    item.valueInDollars = [self.valueField.text intValue];

    }
```

构建并运行应用，首先创建一个名为”Coffee Cup“、价值为75元的BNRItem对象。然后创建第二个BNRItem对象，这时对象的默认价值也为75元。

同时，控制台中会输出NSUserDefaults中存储的所有数据。其中，大部分数据的键都是NSGlobalDomain。NSGlobalDomain保存了整个设备的用户偏好设置，例如语言与地区。再看NextItemName，由于应用并没有设置NextItemName的值，因此NSUserDefaults会读取NextItemName的工厂设置。工厂设置的值存放在NSUserDefaults的注册域（registration domain）中。相反，由于现在已经设置了NextItemValue的值，因此NSUserDefaults会读取应用域（application domain）中的值。对于Homepwner来说，应用域的域名是com.bignerdranch.Homepwner。

默认情况下，应用域是空的，没有键也没有值。当应用第一次设置某项用户偏好设置的值时，相应的值会通过指定的键加入应用域。当通过NSUserDefaults获取某项用户偏好设置的值时，NSUserDefaults会先在应用域中查找，如果找到了值，NSUserDefaults就会返回这个值。如果没有找到，NSUserDefaults就会在注册域中查找并返回默认值。

另外，在用户偏好设置较多的应用中，可能需要提供一个”还原默认设置“的按钮，以便将所有用户偏好设置都恢复到默认值。可以通过NSUserDefaults的实例方法removeObjectForKey:删除某个键对应的用户偏好设置。

26.2 设置束

本节将为Homepwner应用创建一个设置束，用户可以在设置应用中找到Homepwner的用户偏好设置列表，并将NextItemName由”Coffee Cup“修改为其他常用值（见图26-1）。

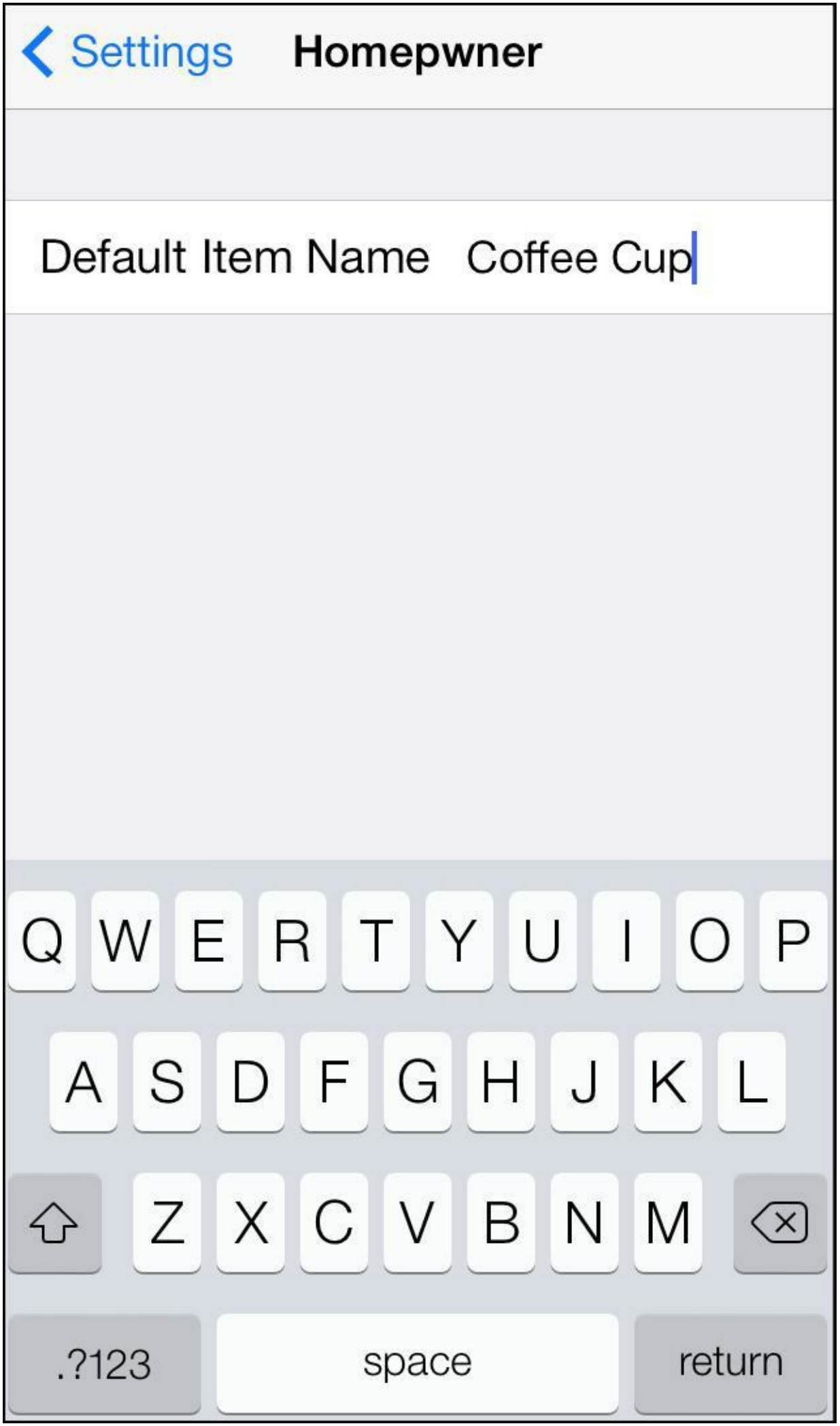

Settings
Homepwner
Default Item Name
Coffee Cup
Q W E R T Y U I O P
A S D F G H J K L
Z X C V B N M
.?123
space
return

图26-1 Homepwner的设置束

虽然可以通过设置束向设置应用注册用户偏好设置，但是，大多数第三方应用并没有采用这种方式。这是因为直接使用应用自身的界面会更方便。相反，如果使用设置应用，那么用户需要先回到主屏幕，启动设置应用，修改之后再重新回到应用。

如果读者确定要借助设置应用修改用户偏好设置，就需要先为应用创建一个设置束，”设置束“听起来很复杂，实际上它只是一个包含设置界面描述文件的目录。设置界面描述文件同样也是plist文件，设置应用通过该文件创建界面中的各项用户偏好设置，以及每项设置所使用的UI控件。除此之外，如果需要对应用实施本地化，还可以向设置束中添加包含本地化字符串（例如图26-1中的Default Item Name）的Root.strings文件。

下面在应用中创建一个新设置束。首先在Xcode的File菜单中选择New→File…，然后在iOS部分选择Resources，最后在右侧区域选择Settings Bundle（见图26-2）。

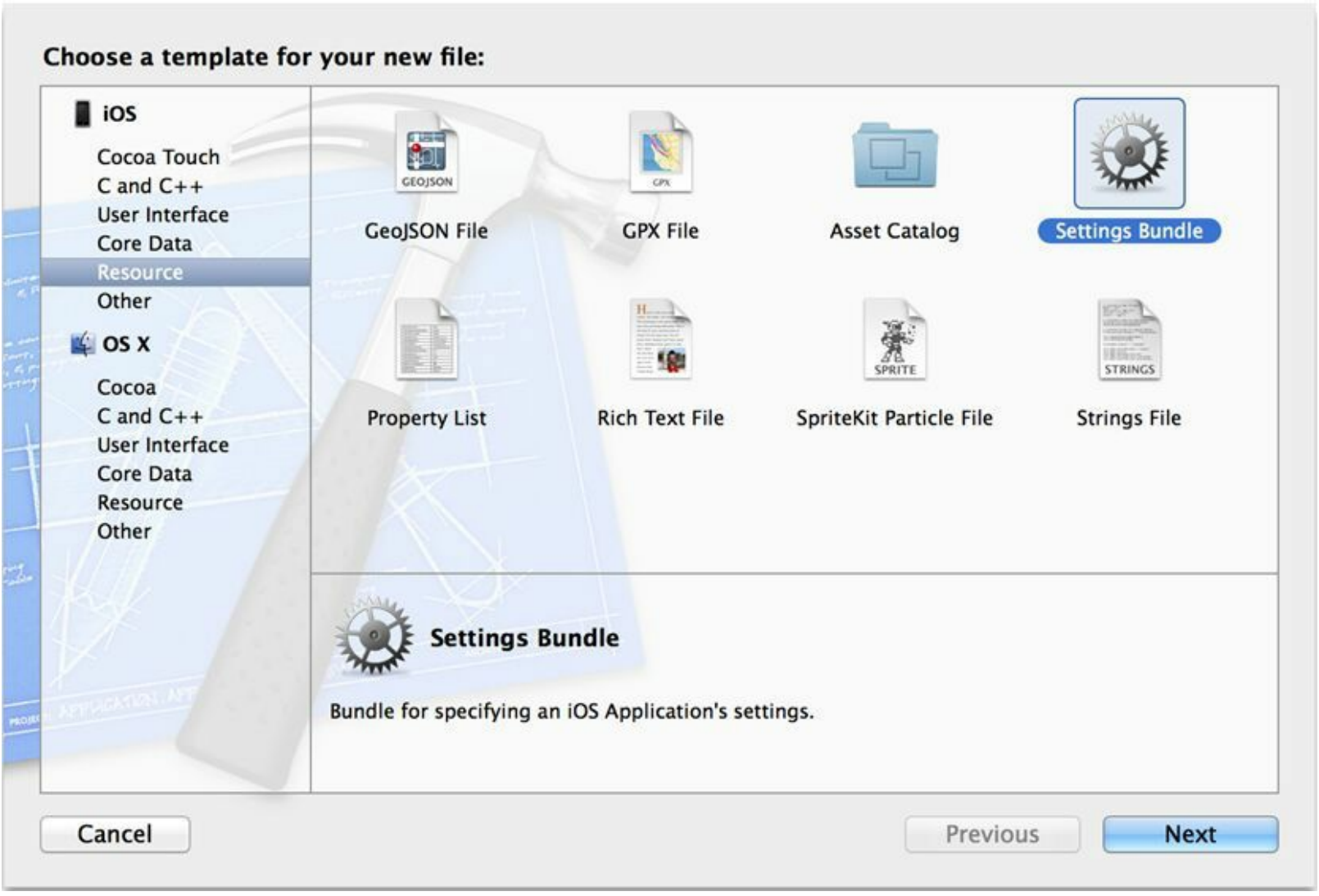


图26-2 Homepwner的设置束

在下一个面板中，使用默认文件名，点击create创建设置束。这时，项目中会出现一个Settings.bundle目录，目录中包含一个Root.plist文件和一个en.lproj子目录。首先介绍Root.plist文件。

编辑**Root.plist**

Root.plist描述了应用的设置界面，它包含一组字典，每一个字典必须含有一个Type（类型）键，表示一个特定的视图。它主要包括以下几种：

PSTextFieldSpecifier	文本框（带有标签）
PSToggleSwitchSpecifier	带有标签的开关（带有标签）
PSSliderSpecifier	不带有标签的滑动条（不带有标签）
PSRadioGroupSpecifier	一组单项选择列表，用户只能选择其中的一个
PSMultiValueSpecifier	进入子一级界面的单项选择列表，用户只能选择其中的一个
PSTitleValueSpecifier	不可编辑的文本（带有标签）
PSGroupSpecifier	用于为用户偏好设置列表分组
PSChildPaneSpecifier	用于创建子一级界面，类似于将子视图控制器压入导航控制器栈

PSChildPaneSpecifier用于创建子一级界面，类似于将子视图控制器压入导航控制器栈

请读者先阅读默认的Root.plist，然后构建并运行应用。当Homepwner启动后，切换到设置应用，找到并查看Homepwner的设置界面。

接下来返回Xcode，打开Root.plist，删除多余的用户偏好设置，只保留一个数组，该数组中只包含一个文本框。对文本框进行以下设置：

•设置Identifier为NextItemName，作为该项用户偏好设置的键。

•设置DefaultValue为Coffee Cup，如果NSUserDefaults中没有设置该项用户偏好设置，就使用该默认值。

•设置Title为NextItemName，用于在Root.strings文件中查找对应的本地化字符串。

这时Root.plist应该类似于图26-3。

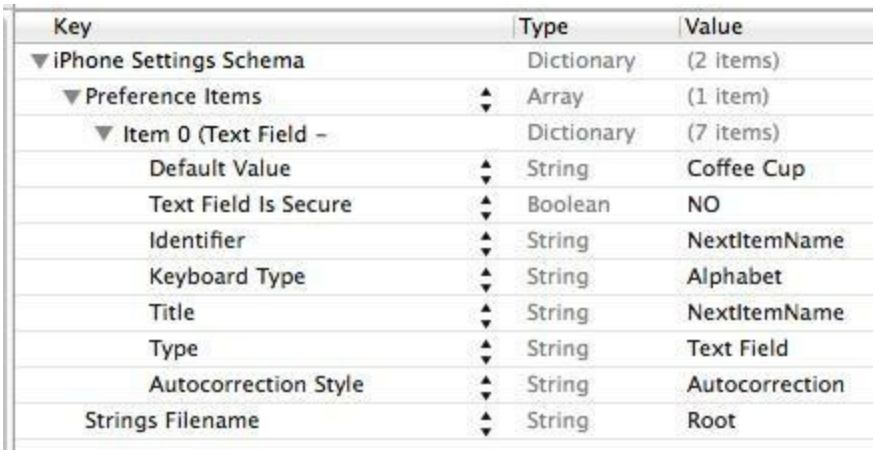


图26-3 Root.plist

实际上，修改Root.plist就是在不编写任何代码或创建视图的情况下对设置应用中的界面进行布局。设置应用会根据Root.plist自动为应用创建设置界面，包括创建视图控制器和所有视图。

如果读者在创建设置束时遇到了问题，则可以参考Apple提供的Settings Application Schema Reference（设置应用格式规范参考），其中列举了Root.plist所有支持的键-值对。

本地化Root.strings

设置束中有一个en.lproj目录，保存了设置界面需要使用的英文版字符串。读者可以将其中的键-值对修改成需要的文字，例如，可以为文本框设置标题：

”NextItemName“ = ”Default Item Name“;

其他名称也可通过以上格式进行修改。构建并运行应用，当用户创建一个新的BNRItem对象时，Homepwner会从设置应用中读取NextItemName的值作为该对象的默认名称。

最后，如果某项用户偏好设置发生了变化（应用内或设置应用中），应用就会收到NSUserDefaultsDidChangeNotification通知。如果某个对象需要及时响应用户偏好设置的变化，就可以将其注册为该通知的观察者。

第27章 控制动画

“animation(动画)”这个单词来源于拉丁语，意思是“the act of bringing to life.(为无生命的物体注入灵魂。)”在iOS开发中，动画能为应用注入灵魂。在界面交互过程中穿插适当的动画，可以赋予界面视觉线索，帮助用户了解应用的工作流程，从而营造有活力的用户体验。

本章将为HypnoNerd添加多种动画效果，介绍各种动画技术的使用方法。

27.1 基础动画

为应用添加适当的动画是提升用户体验的最佳方式。实际上，并不只是游戏类应用才需要动画，普通应用也可以通过动画提升界面的流畅性，例如，当界面元素出现或消失时，可以使用过渡动画减少突兀感；如果需要重点强调界面上的某个功能，或提示用户执行某项操作，则可以使用动画吸引用户的注意力；当用户执行完一项操作后，可以使用动画告诉用户执行结果和下一步操作。

在为HypnoNerd添加动画之前，首先需要查看iOS提供了哪些动画效果。打开Xcode文档，搜索UIView，然后在搜索结果中打开UIView类参考手册（UIView Class Reference），找到动画（Animations）部分。文档中列出了使用动画API的建议（文档建议iOS 4之后应该使用带有Block对象的动画API，本书也将使用这类API），以及UIView中可以添加动画效果的属性（见图27-1）。

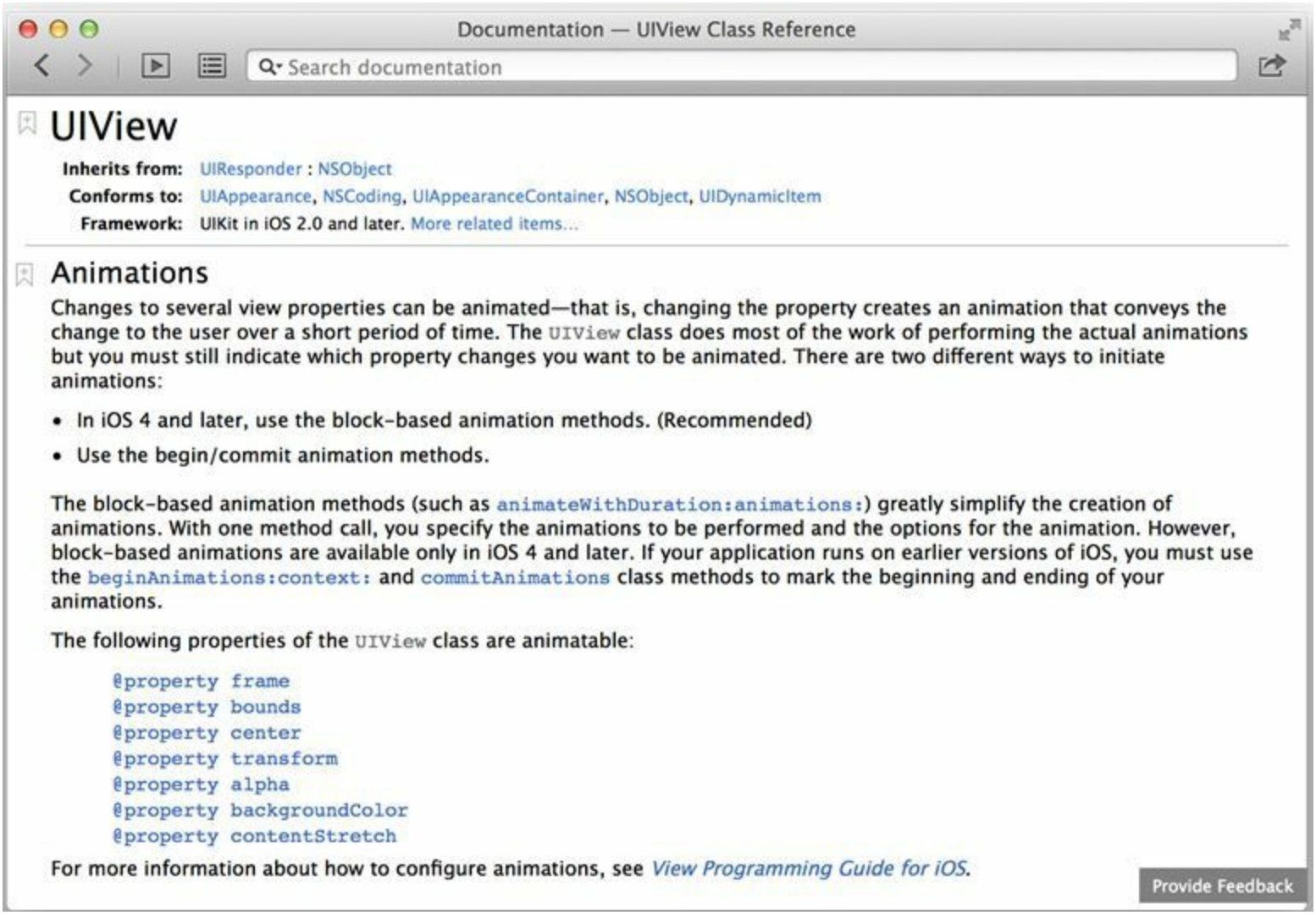


图27-1 UIView动画文档

读者在开始学习一项新技术时，查看文档是最好的学习方式。根据文档提供的信息，下面请读者尝试为HypnoNerd添加一些动画效果。首先添加基础动画（basic animation）。基础动画非常简单，只要提供UIView属性的起始值和终止值就可以了（见图27-2）。

图27-2 基础动画

打开HypnoNerd.xcodeproj，首先为UILabel对象添加透明度（alpha属性）的淡入动画效果。

打开BNRHypnosisViewController.m，修改drawHypnoticMessage:方法，代码如下：

```
[self.view addSubview:messageLabel];

// 设置messageLabel透明度的起始值

messageLabel.alpha = 0.0;

// 让messageLabel的透明度由0.0变为1.0

[UIView animateWithDuration:0.5 animations:^{

messageLabel.alpha = 1.0;

}];

UIInterpolatingMotionEffect *motionEffect =

[[UIInterpolatingMotionEffect alloc] initWithKeyPath:@“center.x”

type:UIInterpolatingMotionEffectTypeTiltAlongHorizontalAxis];
```

构建并运行应用。输入一些文字后点击Done，UILabel对象会有淡入的动画效果。显然，相比让UILabel对象突然出现在屏幕上，淡入动画更加流畅自然。

在带有Block对象的动画API中，最简单的就是animateWithDuration:animations:。该方法有两个参数，第一个参数是动画持续时间，第二个参数是用于执行动画的Block对象。该方法会遵循一个渐快-渐慢（ease-in/ease-out）的速度曲线，也就是说，动画开始时会逐渐加快，结束时会逐渐减慢。

时间函数

动画的速度曲线是由时间函数(timing function)控制的。animateWithDuration: animations:使用默认的渐快-渐慢函数。其他时间函数还包括线性函数(速度保持不变)、渐快函数(速度逐渐加快)和渐慢函数(速度逐渐减慢)等。

如果需要设置动画的时间函数，则可以使用animateWithDuration:delay:options: animations:completion:方法。options:参数中可以设置时间函数和其他一些选项(稍后将介绍可以设置的选项)。除此之外，该方法还可以使动画延迟一段时间执行(delay:)，以及在动画结束时执行特定的代码(completion:)。

在BNRHypnosisViewController.m中修改drawHypnoticMessage:，使用新的动画方法：

```
[self.view addSubview:messageLabel];

// 设置messageLabel透明度的起始值

messageLabel.alpha = 0.0;

// 让messageLabel的透明度由0.0变为1.0

[UIView animateWithDuration:0.5 animations:^{

messageLabel.alpha = 1.0;

}];

[UIView animateWithDuration:0.5

delay:0.0

options:UIViewAnimationOptionCurveEaseIn

animations:^{

messageLabel.alpha = 1.0;

}

completion:NULL];

UIInterpolatingMotionEffect *motionEffect =

[[UIInterpolatingMotionEffect alloc] initWithKeyPath:@“center.x”

type:UIInterpolatingMotionEffectTypeTiltAlongHorizontalAxis];
```

与之前不同，这次使用的是渐快函数(UIViewAnimationOptionCurveEaseIn)，动画结束时速度将不会逐渐减慢。options:中使用的是位掩码，可以使用按位或运算符(“|”)将多个选项值连接起来。以下是常用的选项值。

时间函数选项，控制动画的时间函数曲线

•UIViewAnimationOptionCurveEaseInOut

•UIViewAnimationOptionCurveEaseOut

•UIViewAnimationOptionCurveEaseIn

•UIViewAnimationOptionCurveLinear

UIViewAnimationOptionAllowUserInteraction

只有显式设置了该选项，视图在动画过程中才能响应用户事件。该选项通常用于重复动画，例如在抖动的视图中需要响应用户点击事件，以便执行删除操作（类似于在主屏幕中删除某个应用）。

UIViewAnimationOptionRepeat

让动画反复执行。该选项通常与UIViewAnimationOptionAutoreverse结合使用。

UIViewAnimationOptionAutoreverse

让UIView的属性从起始值变化到终止值后自动执行反转动画，再从终止值变化到起始值。

UIView类参考手册的Constants（常量）部分列举了所有可用的动画选项。本章稍后还会介绍更多选项。

27.2 关键帧动画

之前添加的动画都属于基础动画，只能将UIView的某个属性从一个值变化到另一个值。如果需要在多个值之间进行变化，就需要使用关键帧动画（keyframe animation）。关键帧动画可以包含任意一个关键帧（见图27-3），读者可将关键帧动画看成是若干个连续执行的基础动画。

图27-3 关键帧动画

关键帧动画的使用方法与基础动画类似，对应的UIView类方法是animateKeyframesWithDuration:delay:options:animations:completion:。同时，在animations:的Block对象中，还需要通过addKeyframeWithRelativeStartTime: relativeDuration:animations:方法依次添加每一个关键帧。

在BNRHypnosisViewController.m中修改drawHypnoticMessage:方法，将UILabel对象首先移动到屏幕中央，然后移动到一个随机位置。

```
[UIView animateWithDuration:0.5

delay:0.0

options:UIViewAnimationOptionCurveEaseIn

animations:^{

messageLabel.alpha = 1.0;

}

completion:NULL];
```

```
[UIView animateKeyframesWithDuration:1.0 delay:0.0 options:0 animations:^{

[UIView addKeyframeWithRelativeStartTime:0

relativeDuration:0.8 animations:^{

messageLabel.center = self.view.center;

}];

[UIView addKeyframeWithRelativeStartTime:0.8 relativeDuration:0.2

animations:^{

int x = arc4random() % width;

int y = arc4random() % height;

messageLabel.center = CGPointMake(x, y);

}];

} completion:NULL];

UIInterpolatingMotionEffect *motionEffect =

[[UIInterpolatingMotionEffect alloc] initWithKeyPath:@"center.x"

type:UIInterpolatingMotionEffectTypeTiltAlongHorizontalAxis];
```

animateKeyframesWithDuration:delay:options:animations:completion:中参数的使用方法与基础动画大致相同，只是基础动画的options:是UIView- AnimationOptions类型，而关键帧动画的options:是UIViewKeyframeAnimationOptions类型。另外，关键帧动画中的持续时间(duration)是整个动画的持续时间，也就是所有关键帧持续时间的总和。

addKeyframeWithRelativeStartTime:relativeDuration:animations:中的第一个参数是相对起始时间(relative start time)，表示该关键帧开始执行的时刻在整个动画持续时间中的百分比，取值范围是0到1。第二个参数是相对持续时间(relative duration)，表示该关键帧占整个动画持续时间的百分比，取值范围也是0到1。在本例中，第一个关键帧从0%开始(相对起始时间是0.0)，持续时间是整个动画持续时间的80%(相对持续时间是0.8)；第二个关键帧从80%开始(相对起始时间是0.8)，持续时间是整个动画持续时间的20%(相对持续时间是0.2)。

构建并运行应用。输入一些文字后点击Done，UILabel对象首先会移动到屏幕中央，然后会随机移动到另一个位置并停止。

27.3 在动画完成后执行特定的代码

编写iOS应用时，可能需要知道某个动画会在何时结束。这样就可以在动画结束时执行特定的代码，例如执行下一个动画，或者更新某个对象。怎样才能知道某个动画已经结束了？可以为completion:传入一个Block对象。

修改BNRHypnosisViewController.m，在动画完成后向控制台输出一条日志信息。

~~[UIView animateKeyframesWithDuration:1.0 delay:0.0 options:0 animations:^{~~

~~[UIView addKeyframeWithRelativeStartTime:0 relativeDuration:0.8 animations:^{~~

~~messageLabel.center = self.view.center;~~

~~}];~~

~~[UIView addKeyframeWithRelativeStartTime:0.8 relativeDuration:0.2 animations:^{~~

~~int x = arc4random() % width;~~

~~int y = arc4random() % height;~~

~~messageLabel.center = CGPointMake(x, y);~~

~~}];~~

~~} completion:NULL];~~

```
[UIView animateKeyframesWithDuration:1.0 delay:0.0 options:0 animations:^{
[UIView addKeyframeWithRelativeStartTime:0 relativeDuration:0.8 animations:^{
messageLabel.center = self.view.center;
}];
[UIView addKeyframeWithRelativeStartTime:0.8 relativeDuration:0.2 animations:^{
int x = arc4random() % width;
int y = arc4random() % height;
messageLabel.center = CGPointMake(x, y);
}];
} completion:^(BOOL finished) {
```

NSLog(@“Animation finished”);

}];

UIInterpolatingMotionEffect *motionEffect =

[[UIInterpolatingMotionEffect alloc] initWithKeyPath:@“center.x”

type:UIInterpolatingMotionEffectTypeTiltAlongHorizontalAxis];

构建并运行应用。在动画完成后，控制台就会输出动画执行完成的日志信息。

27.4 弹簧动画

iOS 7提供了一套功能强大的物理引擎，可以模拟真实物理世界的各类效果，其中一个有趣的效果就是弹簧效果。如果将描述弹簧效果的时间函数曲线应用到动画中，就形成iOS 7中常见的弹簧动画。Apple提供了一个轻松为视图添加弹簧动画的方法，下面就使用该方法，为UITextfield对象添加一个从屏幕顶部降落下来的弹簧动画，使它像弹簧一样来回跳动。

打开BNRHypnosisViewController.m，首先在类扩展中添加一个UITextField属性textField。然后修改loadView方法，调整UITextField对象的起始位置，移动到屏幕顶部以外，再将该对象赋给textField属性，代码如下：

```
@interface BNRHypnosisViewController () <UITextFieldDelegate>

@property (nonatomic, weak) UITextField *textField;

@end

@implementation BNRHypnosisViewController

// 这里省略其他方法

- (void)loadView

{

CGRect frame = [UIScreen mainScreen].bounds;

BNRHypnosisView *backgroundView =

[[BNRHypnosisView alloc] initWithFrame:frame];

~~CGRect textFieldRect = CGRectMake(40, 70, 240, 30);~~

CGRect textFieldRect = CGRectMake(40, -20, 240, 30);

UITextField *textField = [[UITextField alloc] initWithFrame:textFieldRect];

// 设置UITextField对象的边框样式，便于查看它在屏幕上的位置

textField.borderStyle = UITextBorderStyleRoundedRect;

[backgroundView addSubview:textField];

self.textField = textField;

self.view = backgroundView;
```

```
}

@end
```

UITextField对象的动画最好在视图刚出现在屏幕上时就开始执行，因此动画代码应该写在viewDidAppear:中。

在BNRHypnosisViewController.m中覆盖viewDidAppear:方法，使用弹簧动画将UITextField对象再移动到之前的位置，代码如下：

```
- (void)viewDidAppear:(BOOL)animated

{

[super viewDidAppear:animated];

[UIView animateWithDuration:2.0

delay:0.0

usingSpringWithDamping:0.25

initialSpringVelocity:0.0

options:0

animations:^{

CGRect frame = CGRectMake(40, 70, 240, 30);

self.textField.frame = frame;

}

completion:NULL];

}
```

用于添加弹簧动画的方法看起来参数较多，但实际上每个参数的含义都非常直观：

构建并运行应用。UITextField对象会从屏幕顶部降落下来，然后像弹簧一样来回跳动，最后停止在与之前相同的位置。

27.5 中级练习：提升Quiz的用户体验

为第1章创建的Quiz应用添加一些动画效果，提升Quiz的用户体验。

当Quiz显示一个新问题或者新答案时，应该让相应的UILabel对象从屏幕左侧飞入，并且透明度从0变为1。同时，旧问题或者旧答案对应的UILabel对象应该从屏幕右侧飞出，并且透明度从1变为0。

为了让动画看起来更逼真，可以尝试并找出效果最好的时间函数和动画执行时间。

第28章 UIStoryboard

前文已经介绍了如何通过独立的XIB文件为UIViewController子类构建用户界面，以及如何编写代码来创建相应的UIViewController子类对象。本章将介绍另一种为UIViewController子类构建用户界面并创建相应对象的途径：Storyboards。Storyboards是由iOS SDK提供的一项功能，可以通过某种类似XIB文件的文件，为所有的UIViewController子类对象构建用户界面，并创建相应的对象。此外，还可以将Storyboard文件中的UIViewController子类对象“连接”起来，设置相应的显示流程和显示模式。

Storyboards的作用是减少编写iOS应用时要编写的某类代码，其中包括：创建并设置UIViewController子类对象，以及UIViewController子类对象之间的交互。本章将通过一则例子应用，向读者介绍Storyboards的优点和缺点。

28.1 创建Storyboard文件

通过Empty Application模板创建一个新的iOS应用项目，并将该项目命名为Colorboard（见图28-1）。

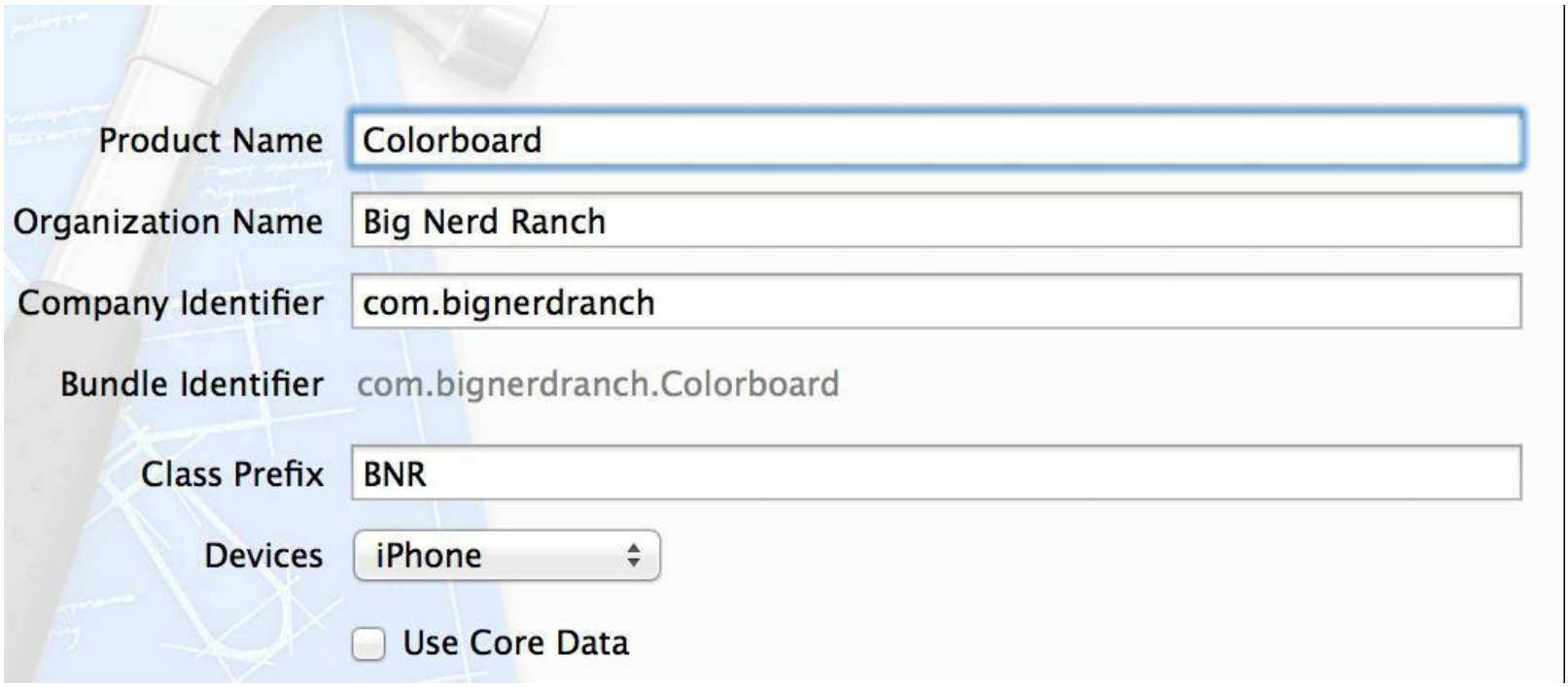


图28-1 创建Colorboard

选择New菜单中的New File…菜单项，在左侧的iOS区域选择User Interface，然后选择右侧的Storyboard模板，最后单击Next按钮（见图28-2）。

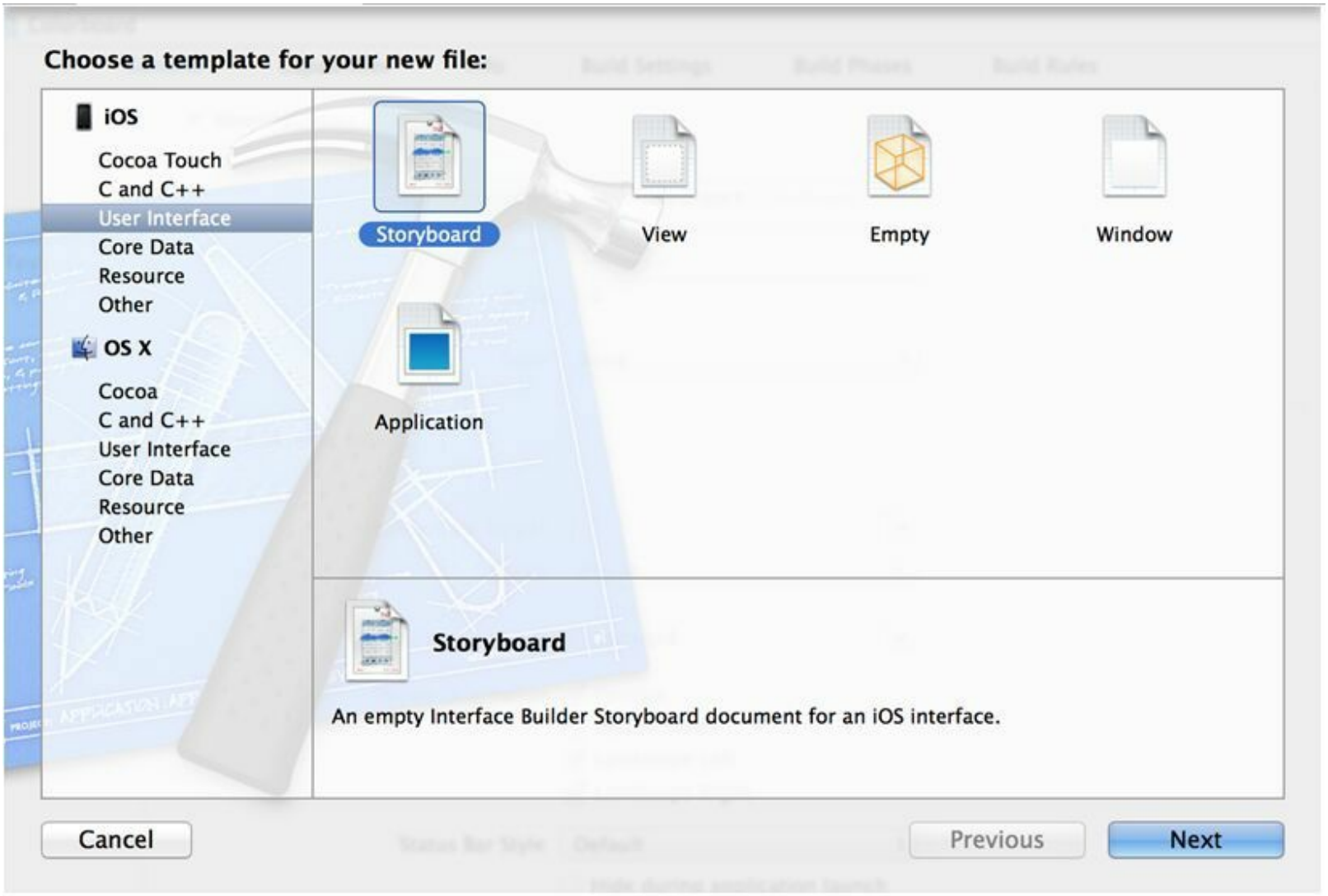


图28-2 创建storyboard文件

在新出现的面板中，找到标题为Device Family的下拉菜单，选择iPhone，单击Next按钮，然后将文件命名为Colorboard。

Xcode将创建一个名为Colorboard.storyboard的文件，并会在编辑器区域显示该文件。Storyboard文件和XIB文件类似，其不同点是，除了可以使用Storyboard文件构建界面，还可以使用Storyboard文件为多个视图控制器设置相互间的关系。Colorboard应用一共需要五个视图控制器，其中包括一个UINavigationController对象和一个UITableViewController对象。图28-3显示的是Colorboard的对象图。

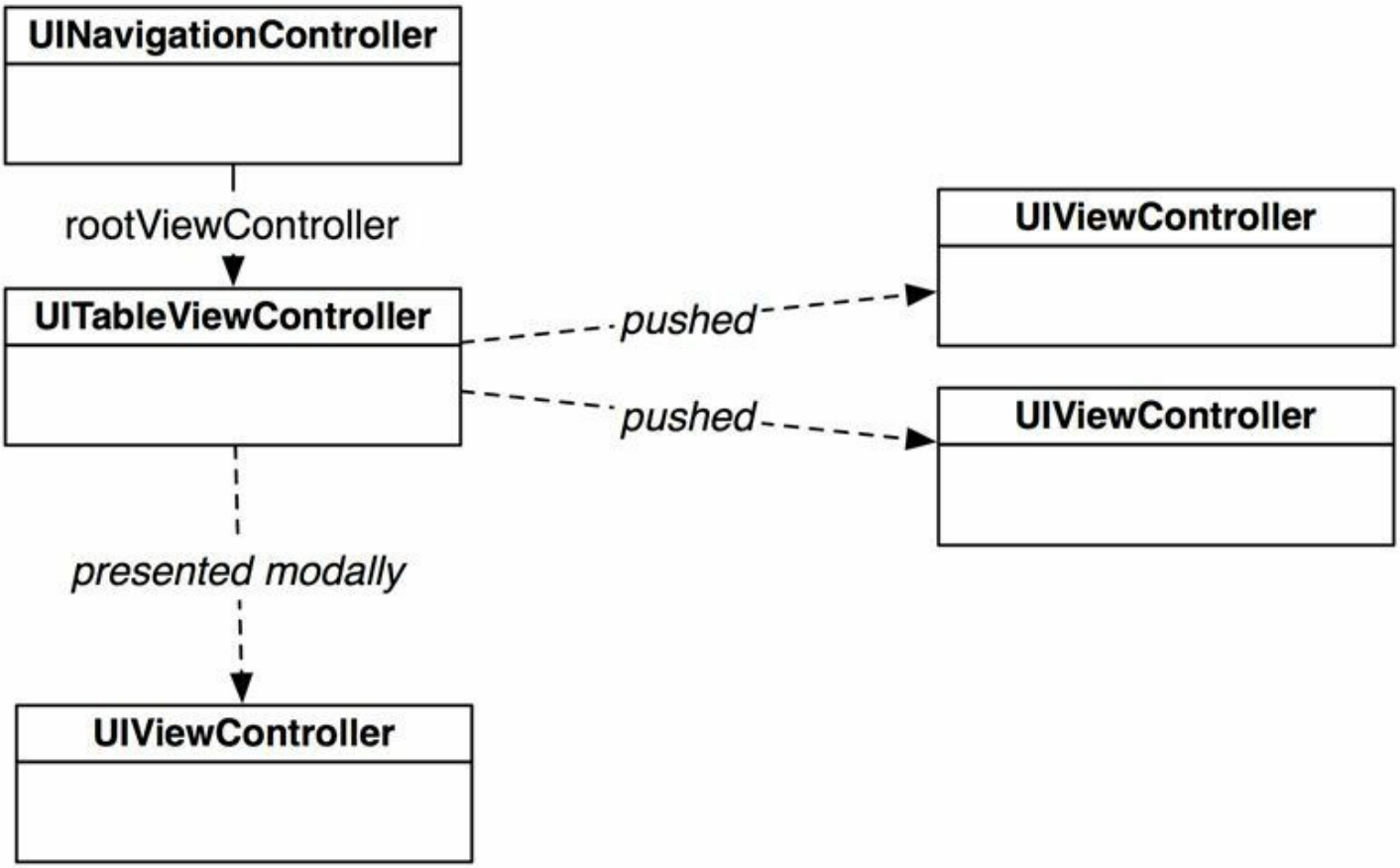


图28-3 Colorboard的对象图

通过Storyboard文件，不用编写任何代码，就可以构建出如图28-3所示的对象关系。

打开工具区域，显示对象库面板，拖曳一个UINavigationController对象至画布（见图28-4）。

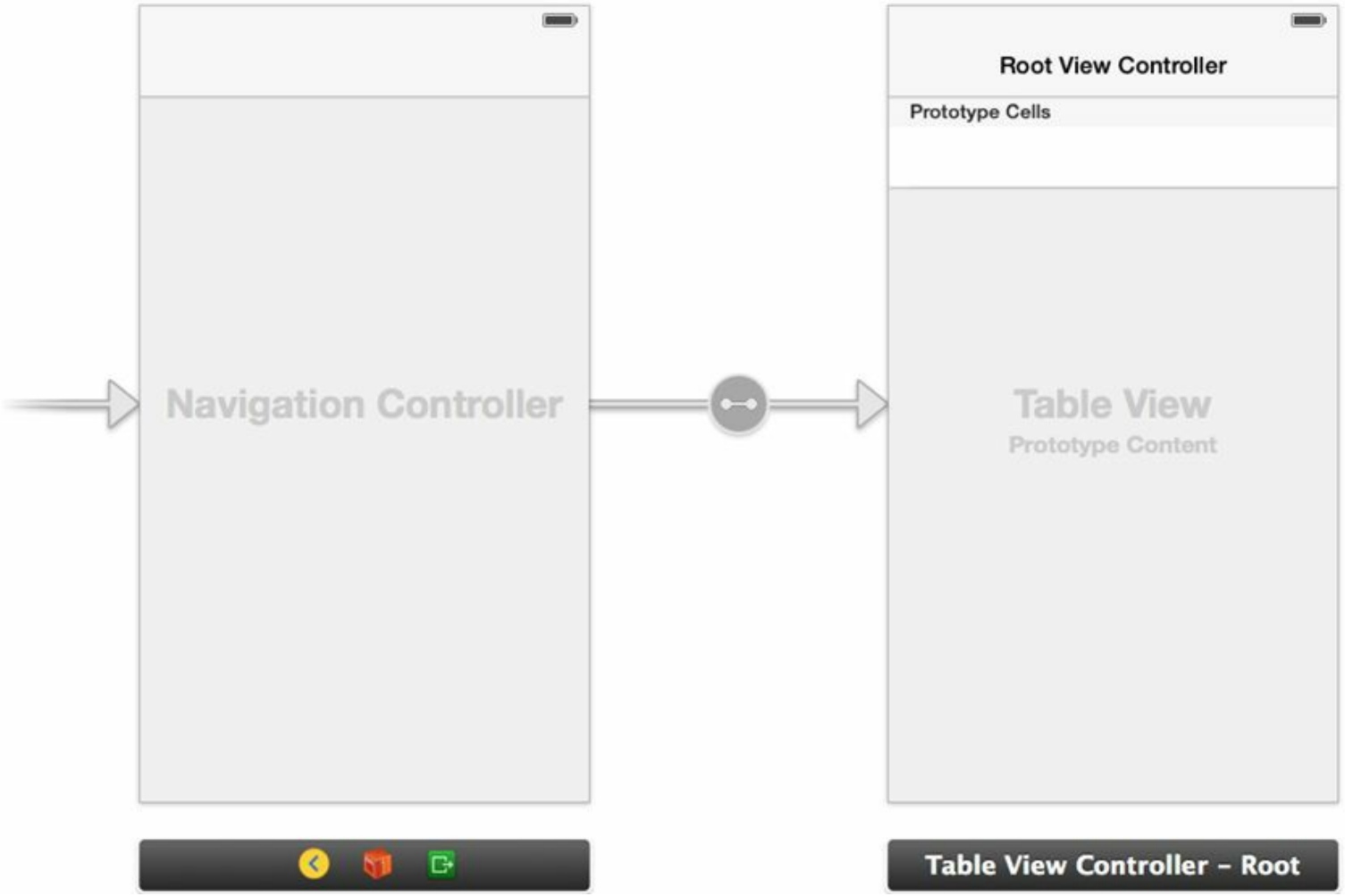


图28-4 Storyboard文件中的UINavigationController对象

将UINavigationController对象拖曳至Storyboard文件后，Xcode还会自动创建三个相关对象：一个UINavigationController对象的视图、一个UITableViewController对象和一个UITableViewController对象的视图。此外，Xcode还会将这个UITableViewController对象设置为UINavigationController对象的根视图控制器。

Xcode会用画布中的两个黑色工具条来分别代表UINavigationController对象和UIViewController对象。位于这两个工具条上方的是相应的视图，视图的设置方式和在XIB文件中设置视图的方式相同。如果要设置UIViewController对象，就应该先选中相应的黑色工具条。

在构建Storyboard文件前，先将Colorboard应用和新创建的Storyboard文件关联起来。选中位于项目导航面板顶部的Colorboard项目文件，然后在编辑区域选择Colorboard目标，再点击General标签。找到标题为Main Interface的文本框，输入Colorboard（见图28-5），或者选择下拉框中的Colorboard.storyboard文件。

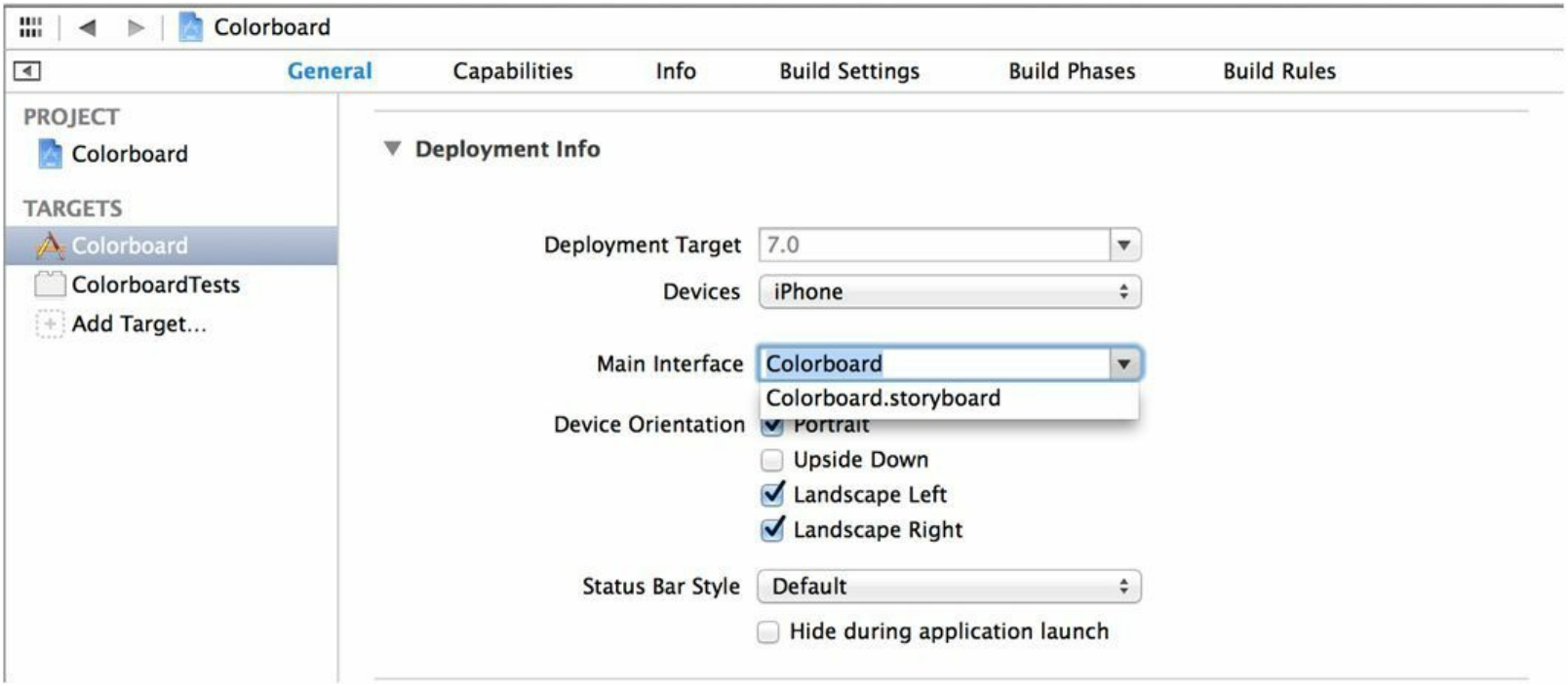


图28-5 设置主Storyboard文件

如果为某个应用设置了主Storyboard文件，该应用就会在启动时载入相应的Storyboard文件。应用除了会载入Storyboard文件和其包含的UIViewController对象，还会创建一个UIWindow对象，并将Storyboard文件中的初始视图控制器(initial view controller)设置为UIWindow对象的根视图控制器。Xcode在画布中显示某个Storyboard文件时，会在初始视图控制器的左侧显示一个灰色的箭头，指向该对象。

因为Storyboard文件会为应用提供UIWindow对象，所以应用委托方法中不需要再创建UIWindow对象了。

更新BNRAppDelegate.m中的application:didFinishLaunchingWithOptions:，删除创建UIWindow对象的那部分代码，如下：

```
- (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;
```

```
}
```

构建并运行应用，Colorboard应该会显示一个空白视图，以及一个标题为Root View Controller的UINavigationBar对象（见图28-6）。这些界面元素都源自之前创建的Storyboard文件，而且无须为此编写一行代码。

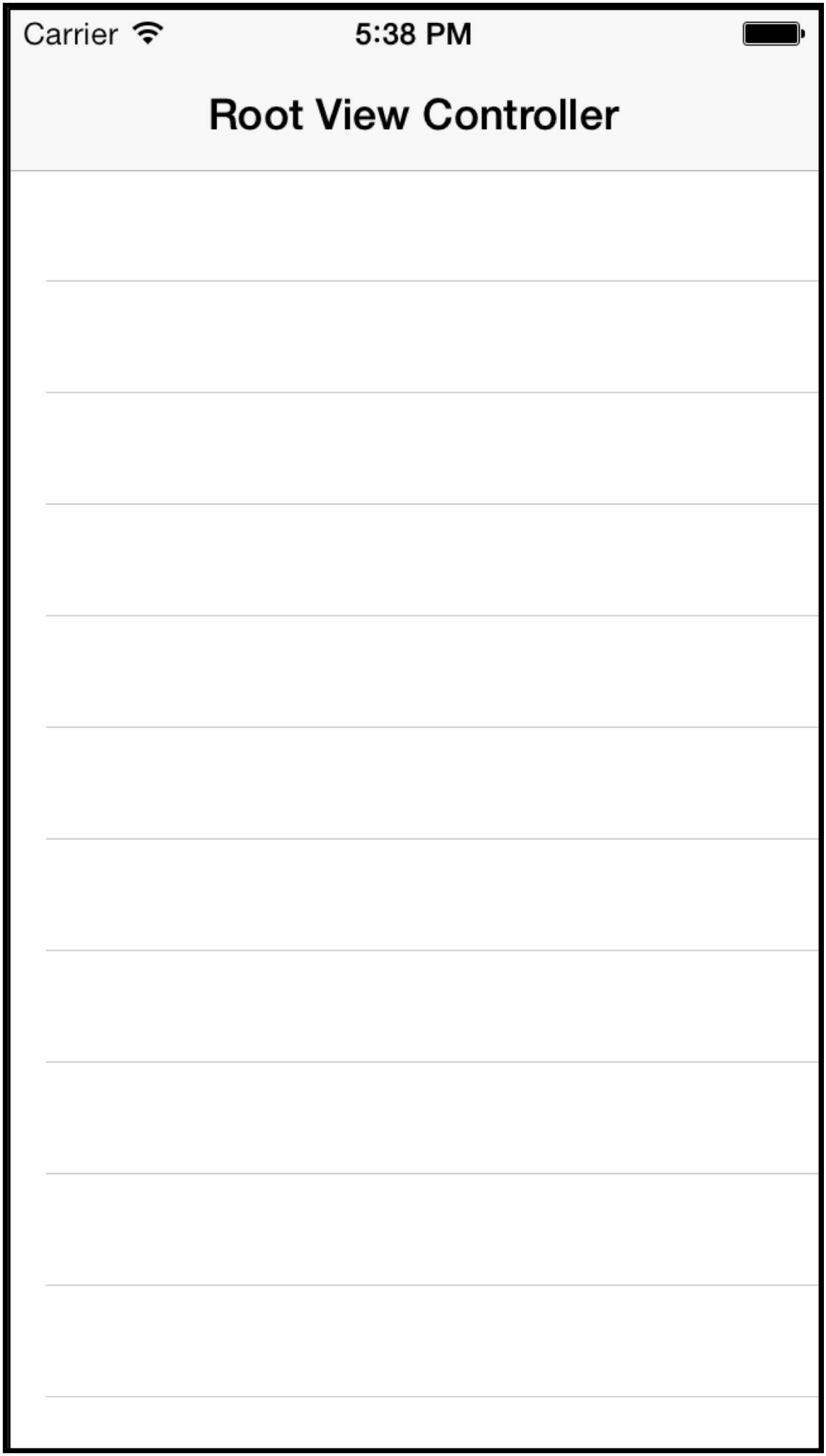

Carrier
5:38 PM
Root View Controller

图28-6 Colorboard的初始界面

28.2 Storyboard文件中的UITableViewController

编写iOS应用时，如果要使用UITableViewController对象，通常都要为其实现相应的数据源方法，针对不同的表格行返回相应的UITableViewCell对象。如果UITableView对象的显示内容是动态的（例如一组可能会被修改的BNRItem对象），通过实现数据源方法，就可以返回显示内容不同的UITableViewCell对象。但是，如果UITableView对象的显示内容是静态的，就可能需要编写大量的代码来创建所有的UITableViewCell对象。通过使用Storyboards，不用实现数据源方法，就能为某个UITableViewController对象设置静态的UITableViewCell对象。

下面要为Colorboard.storyboard增加一个UITableViewController对象，并为该对象设置静态的UITableViewCell对象。Apple经常会改变文件模板，所以，在Colorboard.storyboard中，UINavigationController对象的根视图控制器可能不是UITableViewController。出于学习目的，建议读者手动为UINavigationController对象添加根视图控制器。

如果storyboard自动为UINavigationController对象生成了根视图控制器，那么先删除其根视图控制器。在Colorboard.storyboard中，选中代表根视图控制器的那个黑色工具条，然后按下Delete键。这时的UINavigationController对象将不再拥有根视图控制器。

接下来从对象库面板中拖曳一个UITableViewController对象至画布，然后将其设置为UINavigationController对象的根视图控制器，步骤如下：按住Control键，从UINavigationController对象的view拖曳至UITableViewController对象的view，然后松开鼠标。这时Xcode会显示一个黑色的面板。选择面板中的root View（见图28-7）。

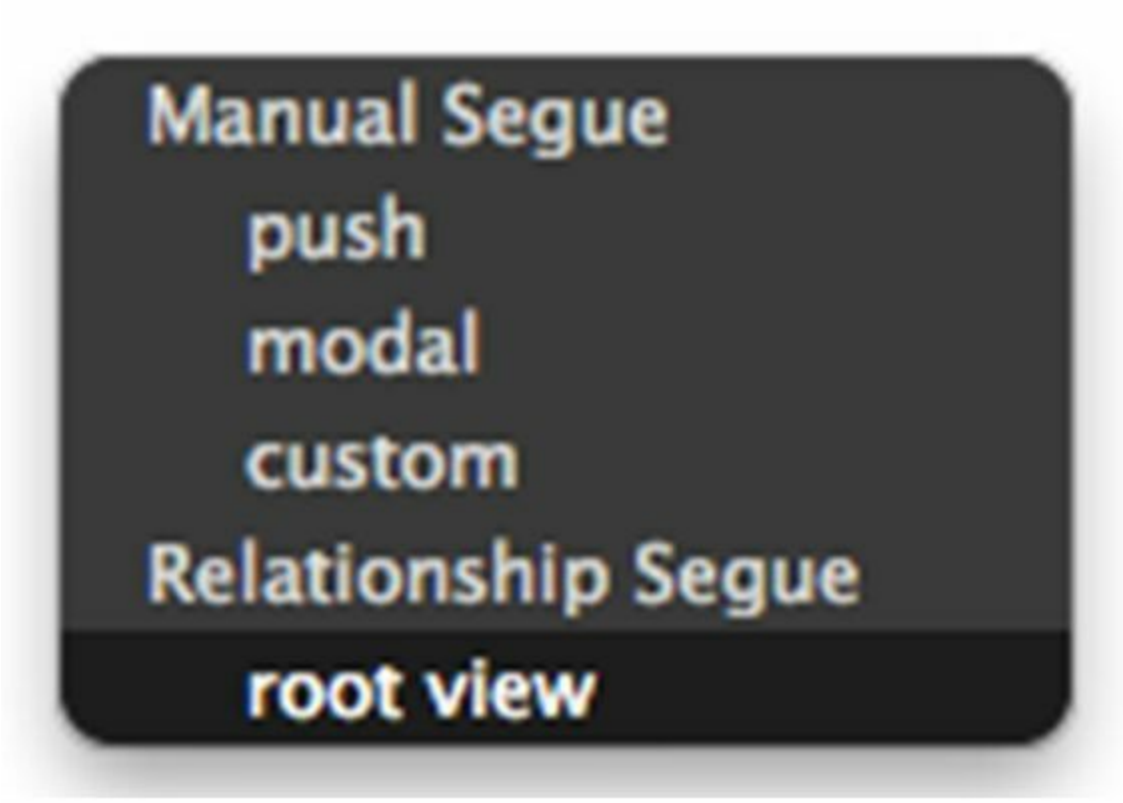


图28-7 设置关系

完成上述步骤后，Xcode会将UITableViewController对象设置为UINavigationController对象

的根视图控制器。Xcode会在这两个对象之间显示一个箭头，从UINavigationController对象指向UITableViewController对象。箭头中间的图标代表相应关系的类型（见图28-8）。

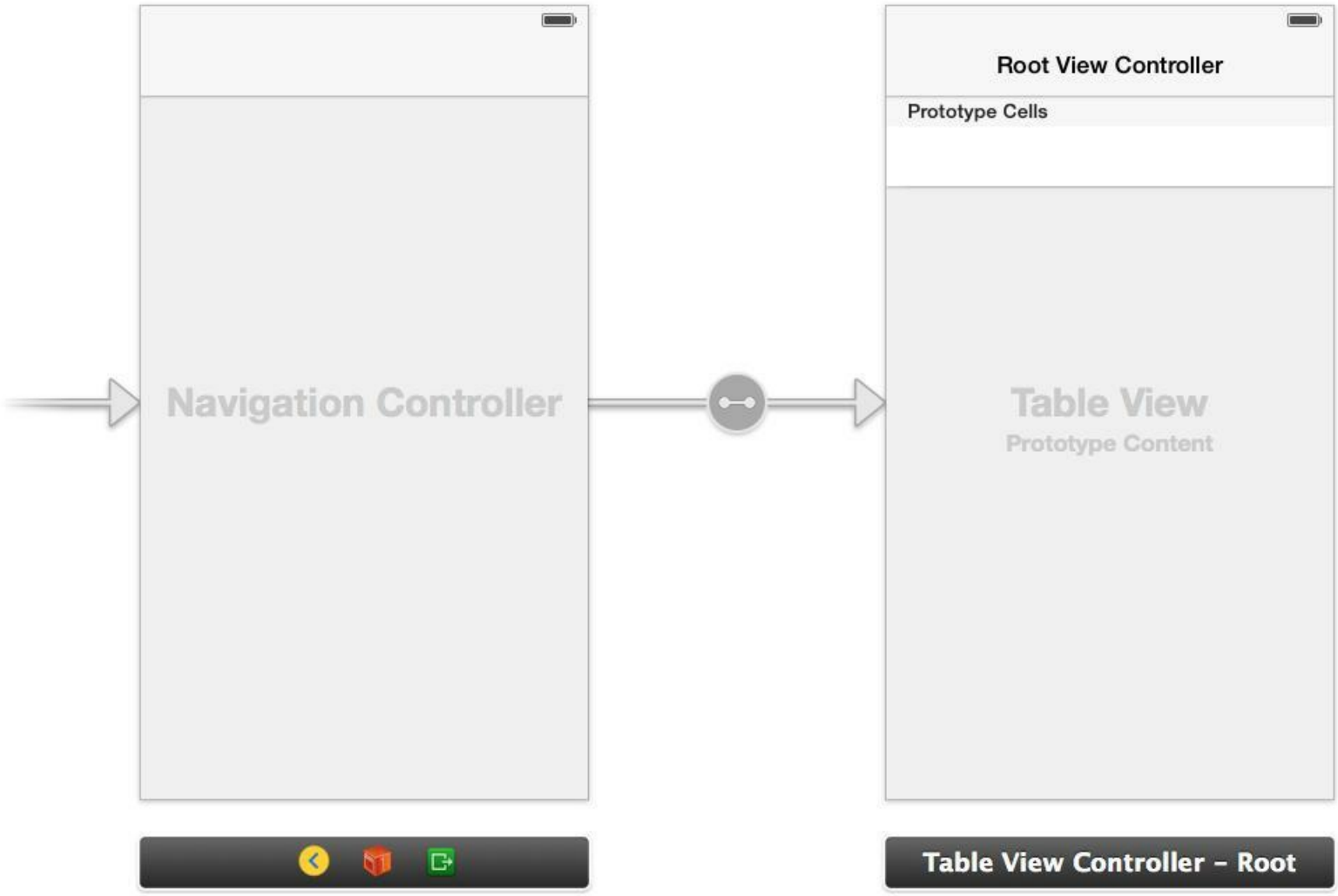


图28-8 设置UINavigationController和UITableViewController

（通过位于画布右下方的缩放按钮，可以修改画布的显示比例。如果缩小显示比例，就可以在画布的可视区域内看到更多的对象。对包含很多UIViewController对象的Storyboard文件，缩小显示比例会很有用。但是，在缩小的显示比例下，将无法在画布中选择UIViewController对象的view及其子视图。）

选中UITableViewController对象的UITableView对象，打开属性检视面板，找到标题为Content的下拉列表，然后选择Static Cells（见图28-9）。

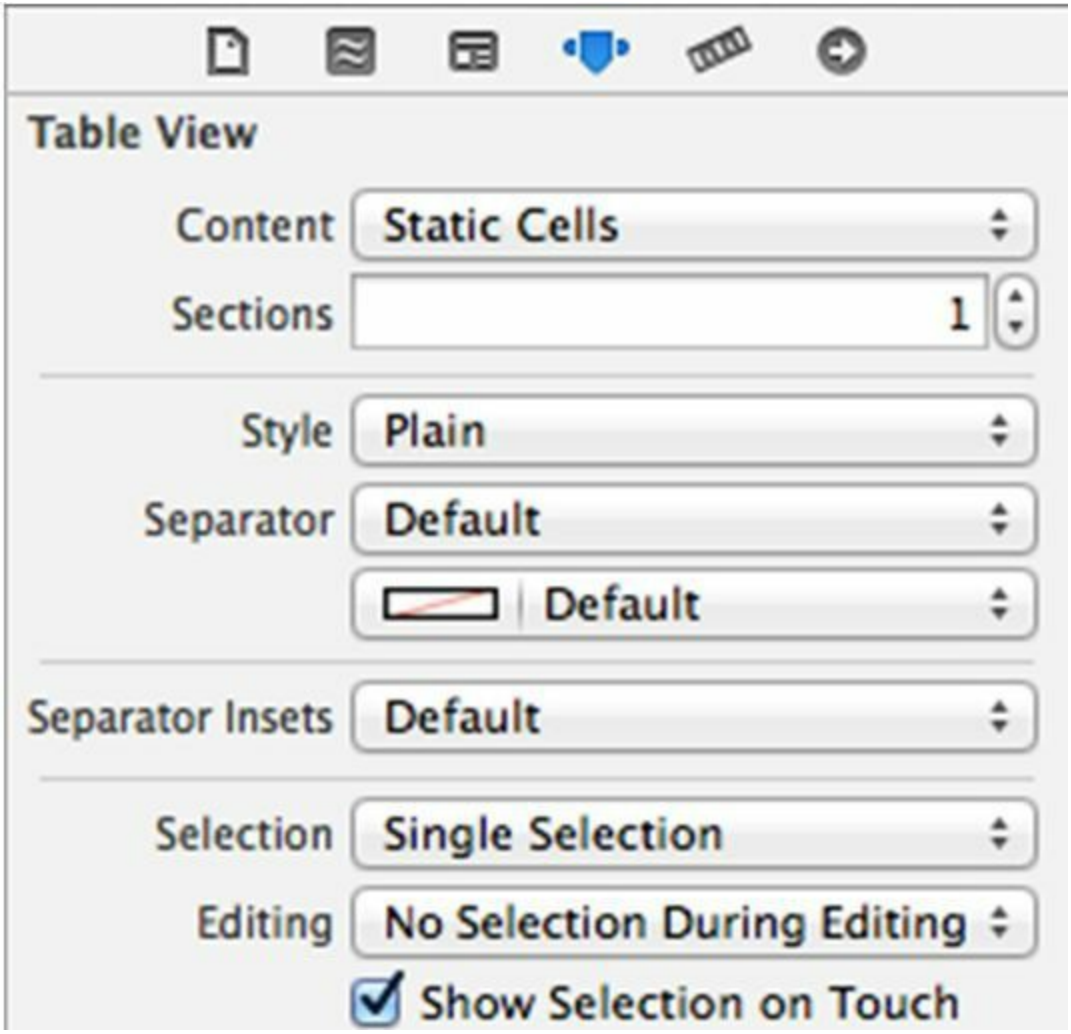


图28-9 Static Cells

Xcode会将三个UITableViewCell对象加至UITableView对象。可以分别选中这些UITableViewCell对象并进行设置。选中位于顶部的UITableViewCell对象，在属性检视面板中，找到标题为Style的下拉列表，然后选择Basic（见图28-10）。

完成上述修改后，当前选中的UITableViewCell对象的标题会变成Title。双击标题，将内容修改为Red（红色）。

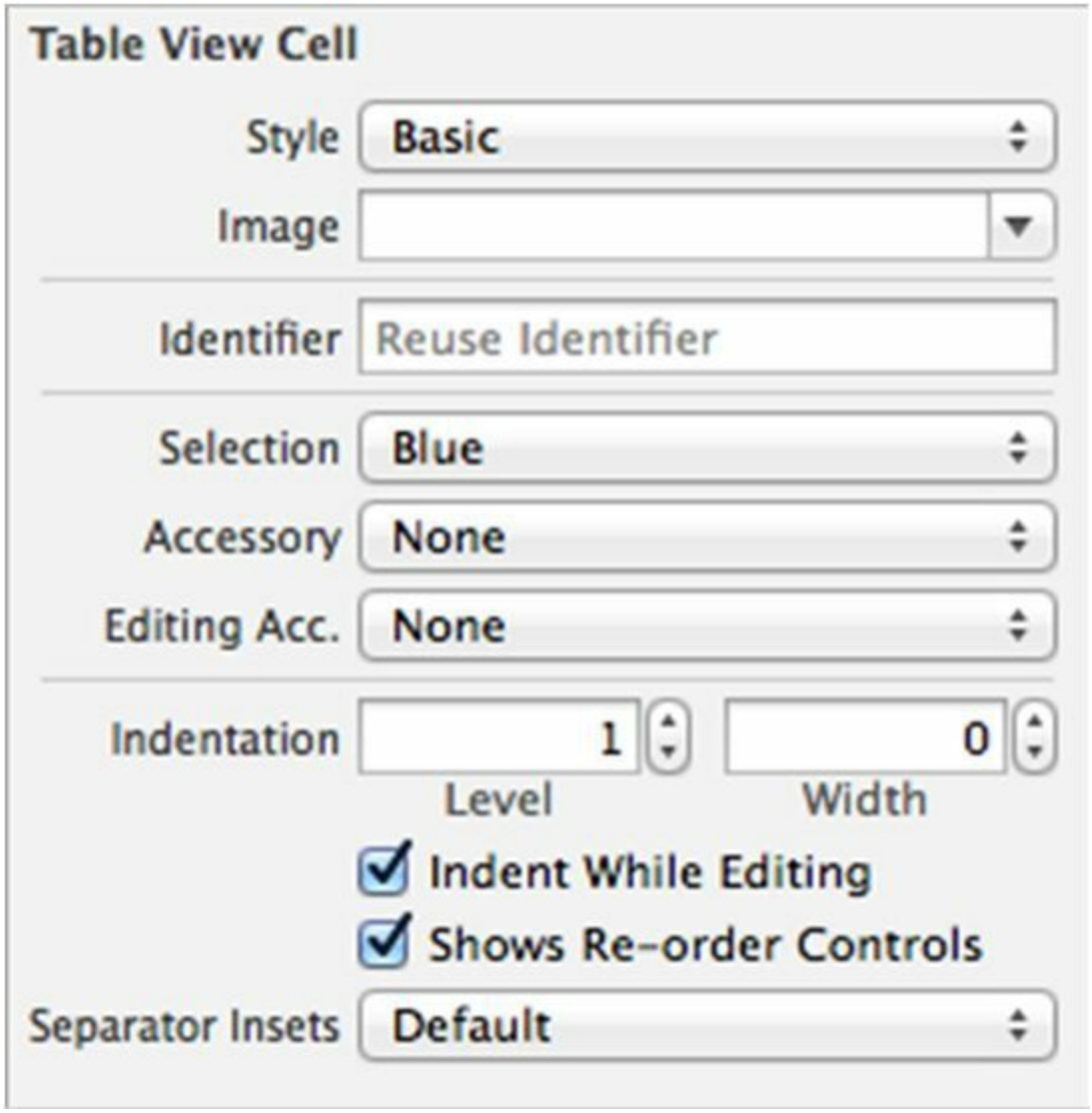


图28-10 Basic样式的UITableViewCell

针对第二个UITableViewCell对象，重复上述步骤，但是要将标题修改为Green(绿色)。最后删除第三个UITableViewCell对象(选中该对象，然后单击Delete)。

由于UITableViewController对象位于UINavigationController对象的视图控制器栈中，因此UITableView对象顶部有一个UINavigationBar对象。选中UINavigationBar对象，在属性检视面板中将其标题改为Colors。完成后的UITableView对象如图28-11所示。

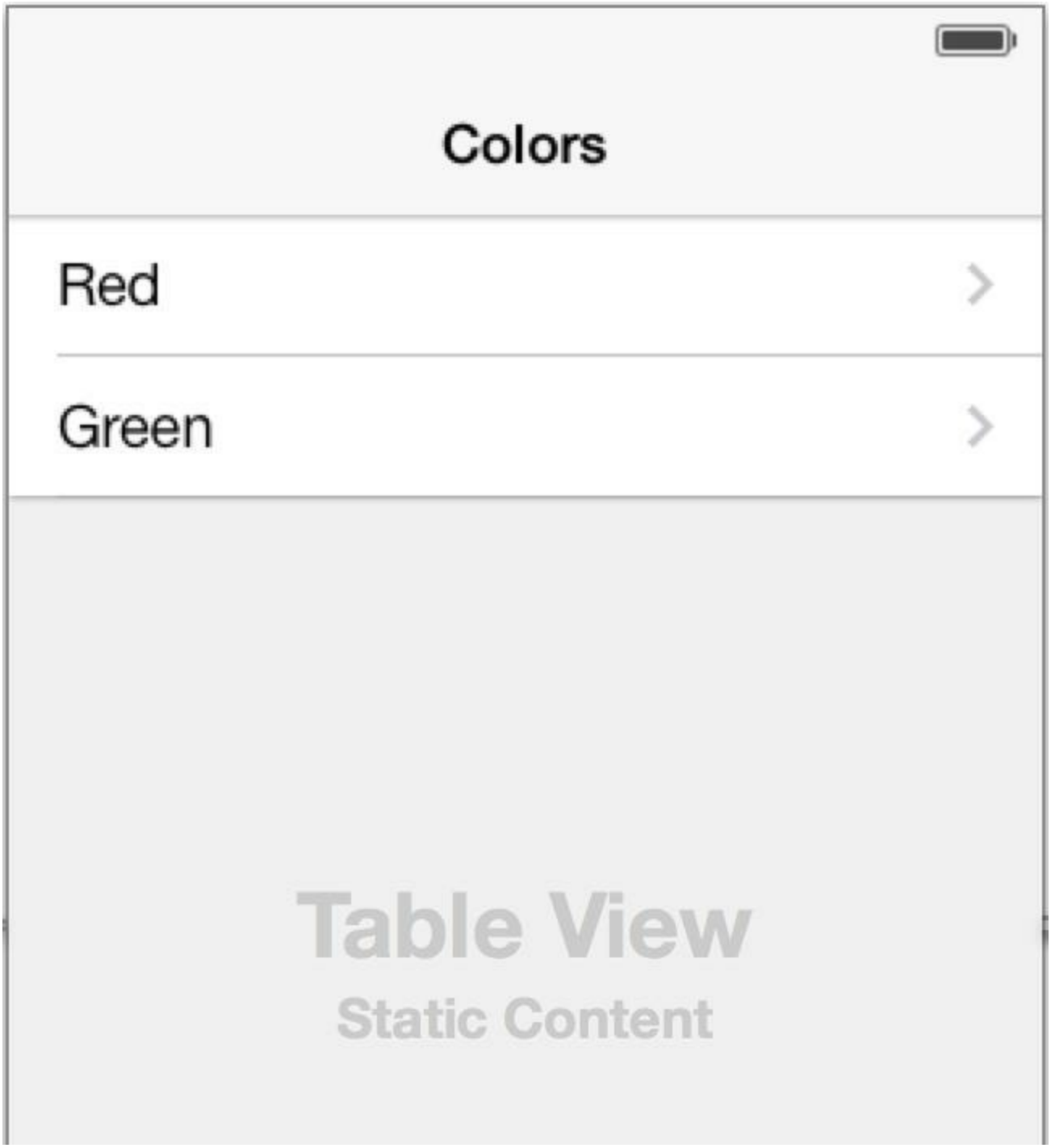


图28-11 完成后的UITableViewCell对象

构建并运行应用。Colorboard的界面应该和Storyboard文件中的界面完全一致：位于窗口顶部的是一个标题为Colors的UINavigationBar对象，位于该对象下方的是一个UITableView对象，其中有两个UITabelViewCell对象，标题分别为Red和Green。

由此可见，在Storyboard中，不需要实现任何数据源方法，就可以在UITableView对象中显示静态的UITableViewCell对象。

28.3 Segue

编写iOS应用时，通常需要实现某种导航界面，让用户能够在多个视图控制器之间切换。使用Storyboards时，可以通过UIStoryboardSegue对象来设置这类互动操作，一样不用编写任何代码。

当应用触发某个UIStoryboardSegue对象时，该对象会将另一个视图控制器移入屏幕。该对象的设置包含样式(style)、动作控件(action item)和标识(identifier)，其中的样式决定应用将怎样显示目标视图控制器(是推入UINavigationController栈还是以模态的形式显示)。动作控件是触发UIStoryboardSegue对象的视图对象，必须和触发的UIStoryboardSegue对象属于同一个Storyboard文件，例如UIButton对象或UIBarButtonItem对象等。标识用来获取相应的UIStoryboardSegue对象。当应用需要主动触发某个UIStoryboardSegue对象时(例如处理摇动事件，或者当触发动作的界面元素无法在Storyboard文件中设置时)，就可以通过标识获取需要的UIStoryboardSegue对象。

下面要为Colorboard.storyboard增加两个push(压入)样式的UIStoryboardSegue对象。push样式的UIStoryboardSegue对象会将目标视图控制器压入相应的UINavigationController栈。在增加UIStoryboardSegue对象前，需要先为Colorboard.storyboard创建并设置两个视图控制器，将其中一个视图控制器的背景颜色设置为红色，另一个设置为绿色。新增加的UIStoryboardSegue对象将位于UITableViewController对象和这两个视图控制器之间。这两个UIStoryboardSegue对象的动作控件都是UITableView对象所包含的UITableViewCell对象。按下某个UITableViewCell对象，Colorboard应该将相应的UIViewController对象压入UINavigationController栈。

从对象库面板中拖曳两个UIViewController对象至画布。选择UIViewController对象的视图，打开属性检视面板，然后将它们的背景颜色分别改为红色和绿色。

选中标题为Red的UITableViewCell对象，按住Control键并拖曳至红色的UIViewController对象，松开鼠标。Xcode会显示一个标题为Storyboard Segues的面板，列出当前UIStoryboardSegue对象可用的全部样式，选择push。

同样，将标题为Green的UITableViewCell对象拖曳至绿色的UIViewController对象，仍然选择push。完成后的画布如图28-12所示。

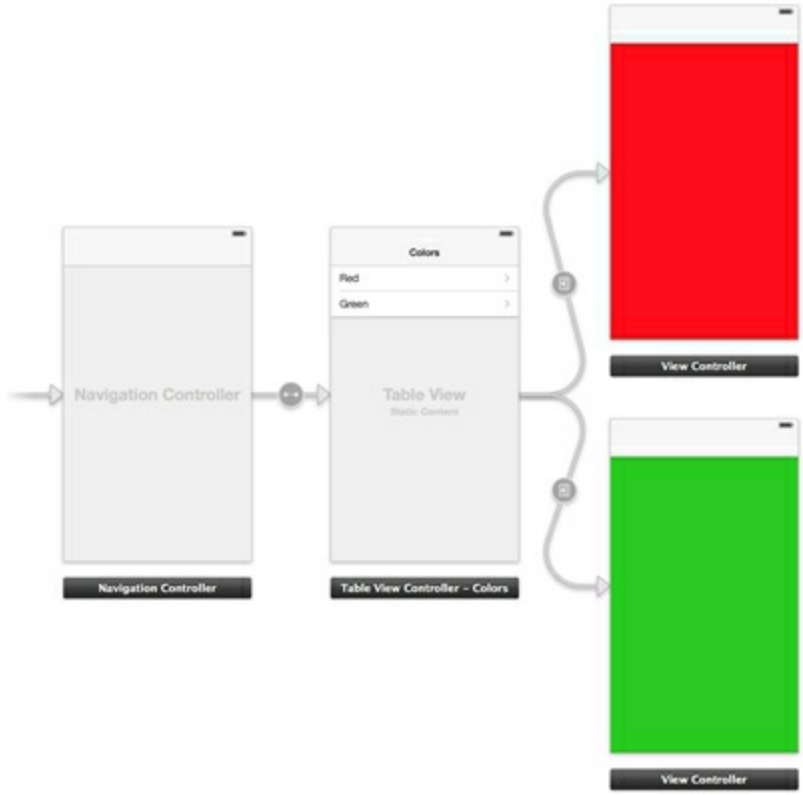


图28-12 设置两个UIStoryboardSegue对象

Xcode会在UITableViewController对象和两个UIViewController对象之间显示两个箭头，分别代表一个UIStoryboardSegue对象。箭头中间的图标代表相应UIStoryboardSegue对象的样式，图28-12中显示的都是push样式。

构建并运行应用，按下某个表格行，Colorboard应该会显示相应的UIViewController对象。点击Back按钮则可以退回至UITableViewController对象。至此，没有为Colorboard编写任何代码就实现了上述功能。

对push样式的UIStoryboardSegue对象，起始的UIViewController对象必须位于某个UINavigationController对象内，否则将无法正常工作。本例中的两个UIStoryboardSegue对象都符合上述条件。

下面再介绍另一种UIStoryboardSegue样式：modal（模态）样式。首先从对象库面板中拖曳一个UIViewController对象至画布，将其背景颜色设置为蓝色。下面为该对象创建一个UIStoryboardSegue对象，并将UIBarButtonItem对象设置为UIStoryboardSegue对象的动作控件。

在UITableViewController对象的视图顶部找到UINavigationBar对象，然后从对象库面板拖曳一个UIBarButtonItem对象至UINavigationBar对象的右侧。同时，打开属性检视面板，在Identifier下拉菜单中选择Add（“＋”）。接下来按住Control键，将UIBarButtonItem对象拖曳至新创建的UITableViewController对象，在黑色面板中选择modal。完成后的画布如图28-13所示（modal样式的UIStoryboardSegue对象和push样式的图标不同）。

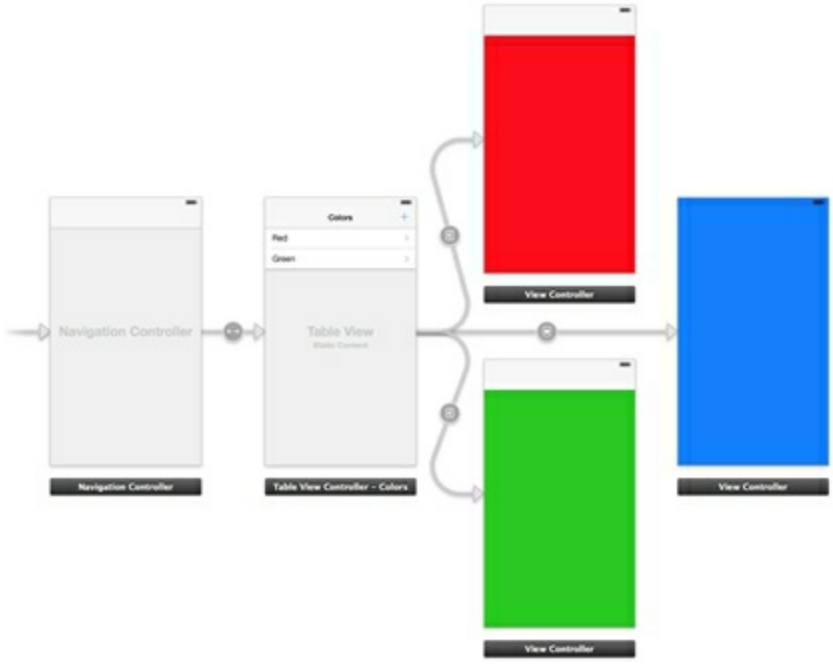


图28-13 modal样式的UIStoryboardSegue对象

构建并运行应用。点击UIBarButtonItem对象，蓝色背景的UIViewController对象应该会滑入窗口。唯一的缺点是无法关闭UIViewController对象。

下面需要在UIViewController对象的UINavigationBar对象中添加一个Done按钮，用于关闭UIViewController对象。但是，目前UIViewController对象并没有加入UINavigationController栈中，因此其顶部也没有UINavigationBar对象。

首先要为UIViewController对象添加一个UINavigationController对象。拖曳一个UINavigationController对象至画布，然后删除Storyboards自动添加的根视图控制器。

删除当前modal样式的UIStoryboardSegue对象，然后按住Control键，将“+”按钮拖曳至该UINavigationController对象，选择modal样式。同时，需要将UIViewController对象设置为该UINavigationController对象的根视图控制器（见图28-14）。

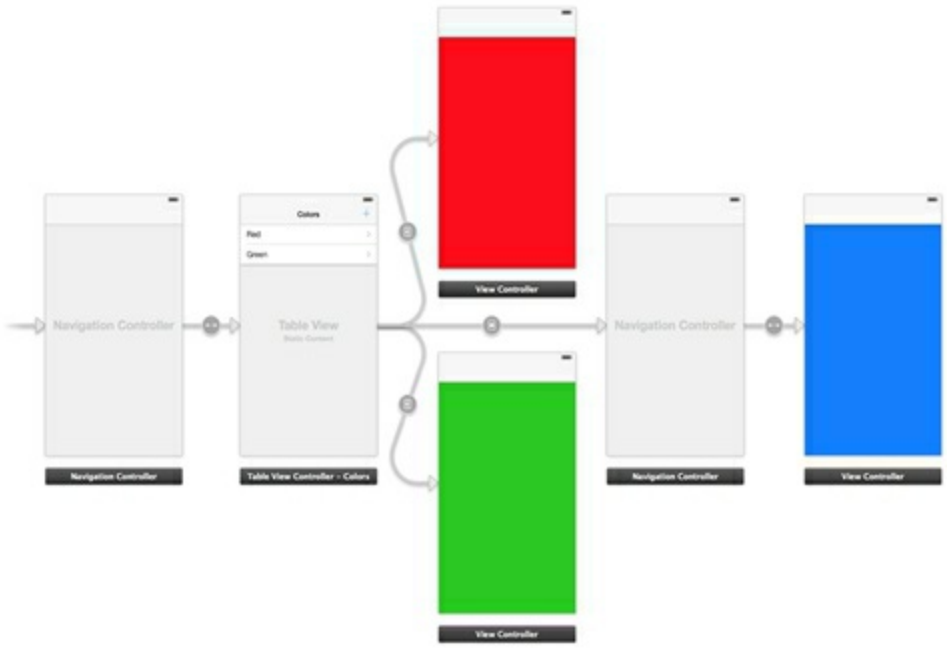


图28-14 添加UINavigationController对象

现在UIViewController对象已经加入到UINavigationController栈中，顶部有一个UINavigationBar对象。下面将一个UIBarButtonItem对象拖曳至UINavigationBar对象的右侧，然后打开属性检视面板，在Identifier下拉菜单中选择Done。这时UIViewController对象看起来应该类似于图28-15。

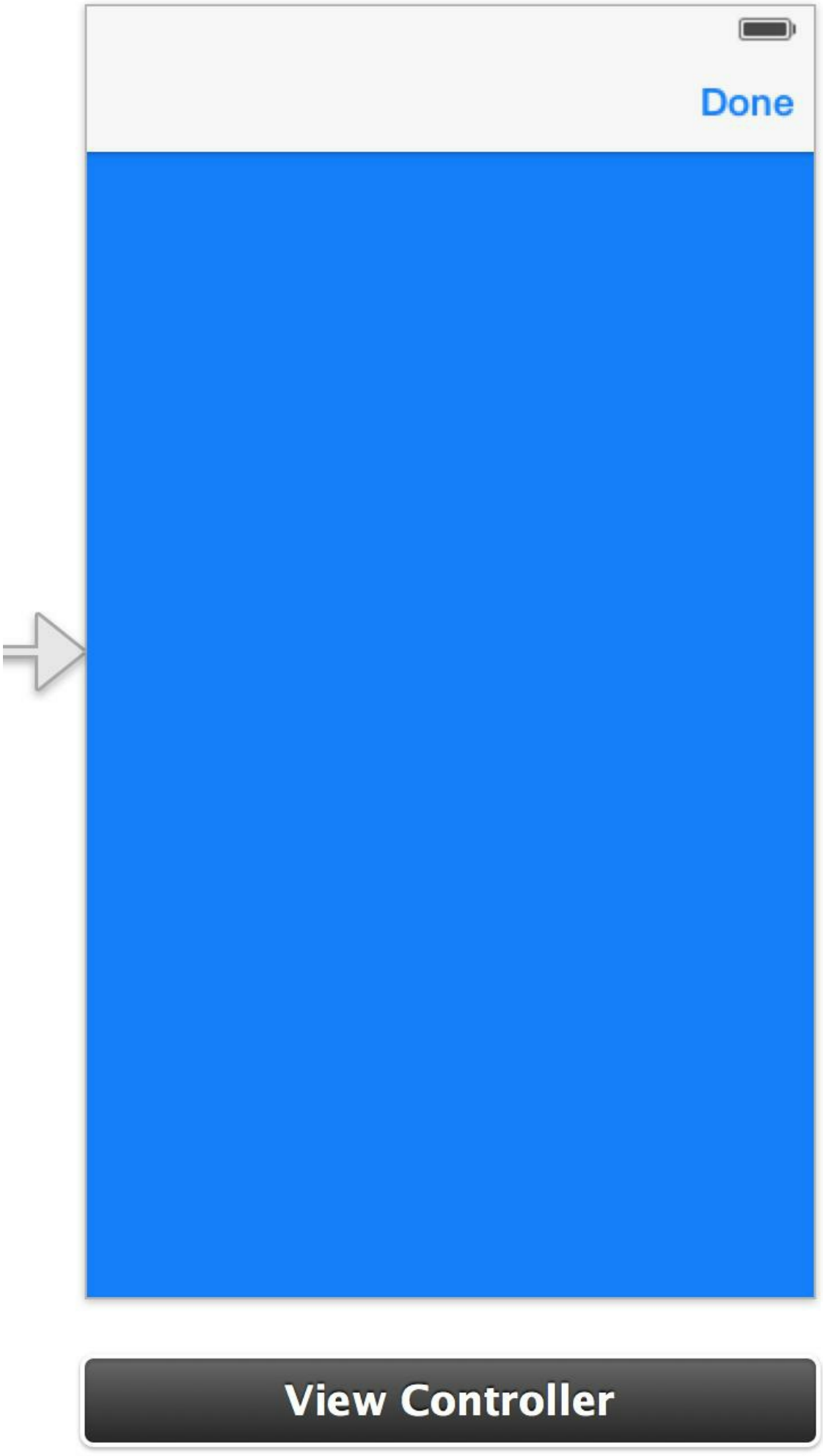

Done
View Controller

图28-15 Done按钮

接下来编写关闭UIViewController对象的方法，并将其与Done按钮关联。

在Colorboard.storyboard文件中，所有的视图控制器都是UIViewController对象，或者是iOS SDK自带的UIViewController子类对象（例如UITableViewController）。这些对象的类是无法修改的。如果需要修改Storyboard文件中的某个视图控制器，必须创建一个UIViewController自定义子类。同时，还需要在Storyboard文件中将视图控制器的类设置为自定义子类。

下面就为Colorboard创建一个新的UIViewController子类。创建一个NSObject子类，命名为BNRColorViewController。

在BNRColorViewController.h中，将父类修改为UIViewController，代码如下：

```
~~@interface BNRColorViewController : NSObject~~

@interface BNRColorViewController : UIViewController

@end
```

在BNRColorViewController.m中实现关闭自身的方法，代码如下：

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

{

[self.presentingViewController dismissViewControllerAnimated:YES

completion:nil];

}
```

重新打开Colorboard.storyboard，选中Colorboard以模态形式显示的蓝色UIViewController对象，然后打开标识检视面板，找到标题为Class的文本框，将UIViewController修改为BNRColorViewController（见图28-16）。

图28-16 将视图控制器的类修改为BNRColorViewController

注意这时的画布不能处于缩放状态，选中Done按钮，然后在黑色工具条中找到代表BNRColorViewController对象的那个图标。按住Control，将Done按钮拖曳至刚才找到的那个图标，松开鼠标。在新出现的面板中选择dismiss:方法（见图28-17）。

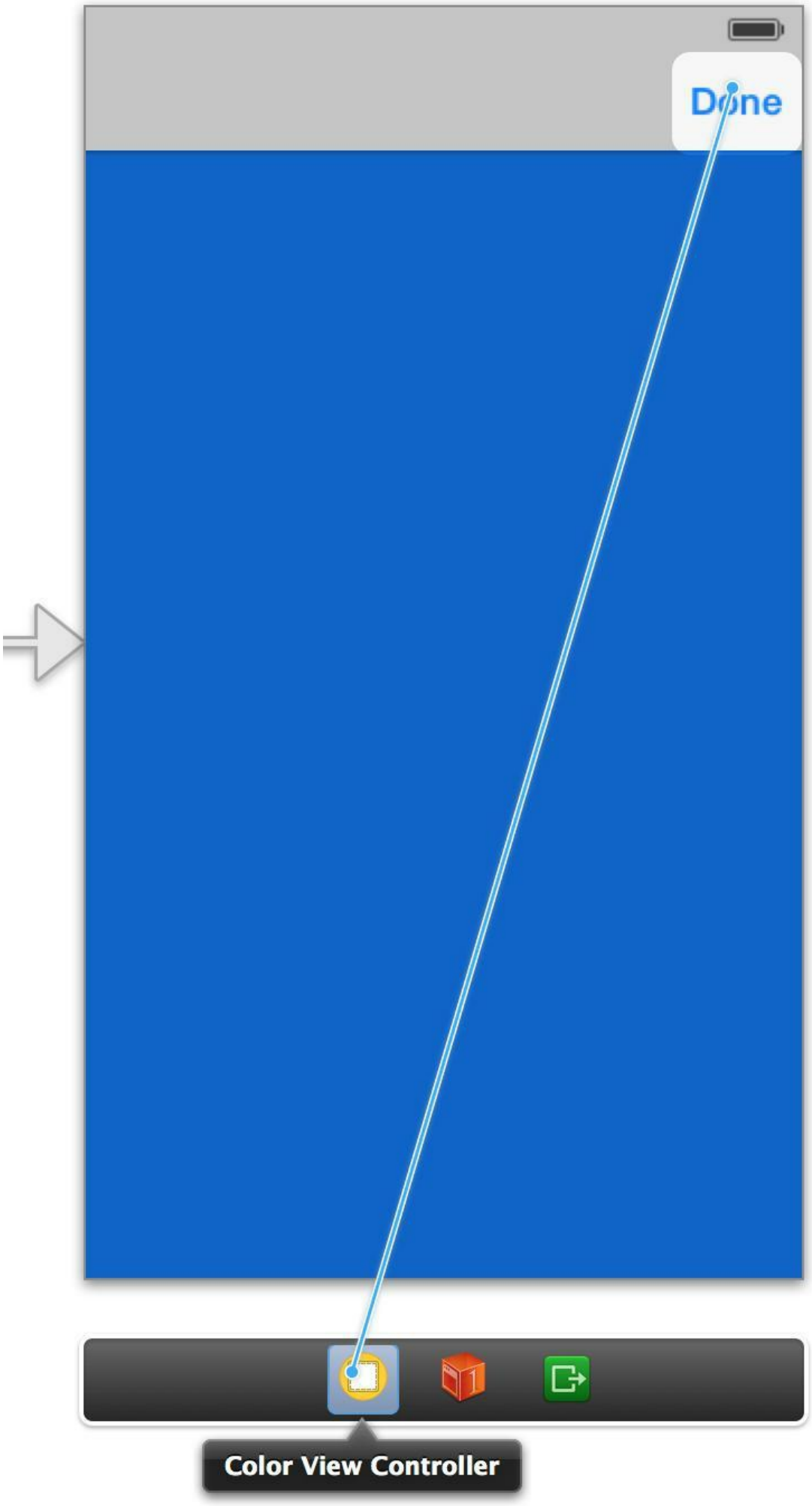

Done
Color View Controller

图28-17 在Storyboard文件中设置插座变量和动作方法

完成上述步骤后，当用户按下Done按钮时，BNRColorViewController对象就会收到dismiss:消息。构建并运行应用，进入BNRColorViewController对象，然后按下Done按钮。Colorboard应该会关闭BNRColorViewController对象。

28.4 改变颜色

下面将升级Colorboard应用，用户可以在Colorboard中选择一款颜色并将其收藏到颜色列表中。

返回Colorboard.storyboard，向BNRColorViewController中添加一个UITextField对象、三个UILabel对象和三个UISlider对象，界面看起来应该类似于图28-18。

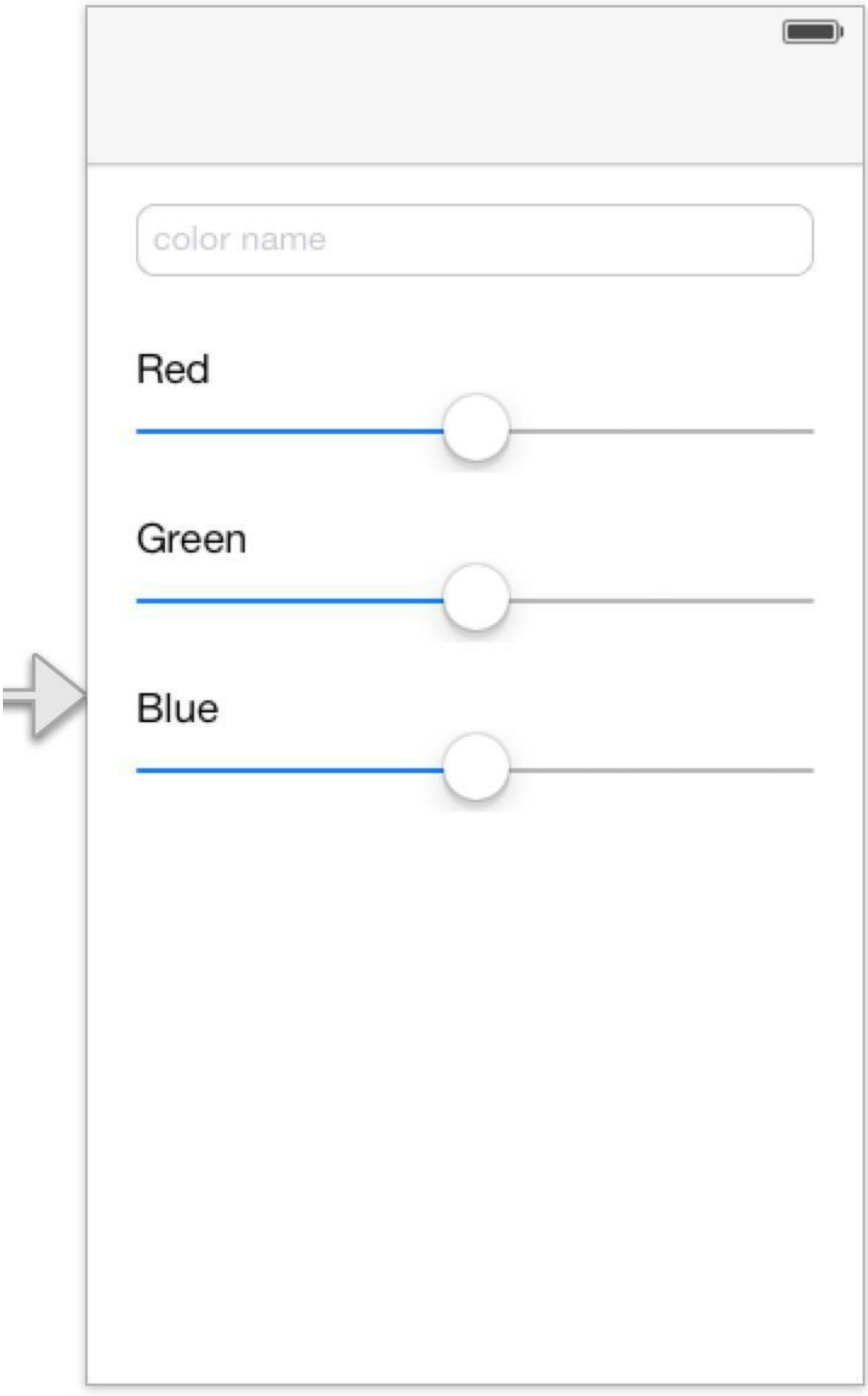

color name
Red
Green
Blue

图28-18 设置BNRColorViewController的view

下面根据UISlider对象中的色值修改BNRColorViewController的view的背景颜色。在BNRColorViewController.m的类扩展中，为UITextField对象和UISlider对象添加相应的插座变量：

```
@interface BNRColorViewController ()

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

@property (nonatomic, weak) IBOutlet UISlider *redSlider;

@property (nonatomic, weak) IBOutlet UISlider *greenSlider;

@property (nonatomic, weak) IBOutlet UISlider *blueSlider;

@end

@implementation
```

当用户拖曳任意一个UISlider对象时，都将触发相同的动作方法。在BNRColorViewController.m中添加该方法，代码如下：

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

{

float red = self.redSlider.value;

float green = self.greenSlider.value;

float blue = self.blueSlider.value;

UIColor *newColor = [UIColor colorWithRed:red

green:green

blue:blue

alpha:1.0];

self.view.backgroundColor = newColor;

}
```

接下来打开Colorboard.storyboard，然后按住Option键，在辅助编辑器中打开BNRColorViewController.m，首先关联视图与对应的插座变量，然后按住Control键，将三个

UISlider对象分别拖曳到Color View Controller，关联changeColor:动作方法。

构建并运行应用，拖曳任意一个UISlider对象，可以发现view的背景颜色会随之改变。

28.5 传递数据

本书第10章介绍过，编写应用时，经常需要在视图控制器之间传递数据。为了演示如何在Storyboards中传递数据，本节将在Colorboard中添加一个颜色收藏列表，用于保存BNRColorViewController中用户选择的颜色。

本节中的UITableView对象将不再使用Static Cells，而是使用Dynamic Prototypes（动态原型）。因此，接下来需要为UITableView对象实现一系列数据源方法。如果需要在数据源中返回不同类型的单元格，就可以使用动态原型。只要为不同类型的单元格分配唯一的重用标识，UITableView对象就可以根据数据源方法显示相应的单元格。

在Colorboard.storyboard中，删除红色和绿色的两个视图控制器，然后选中UITableView对象，打开属性检视面板。接下来点击标题为Content的下拉菜单，改为Dynamic Prototypes，最后在画布中删除第二个UITableViewCell对象。这时storyboard看起来应该类似于图28-19。

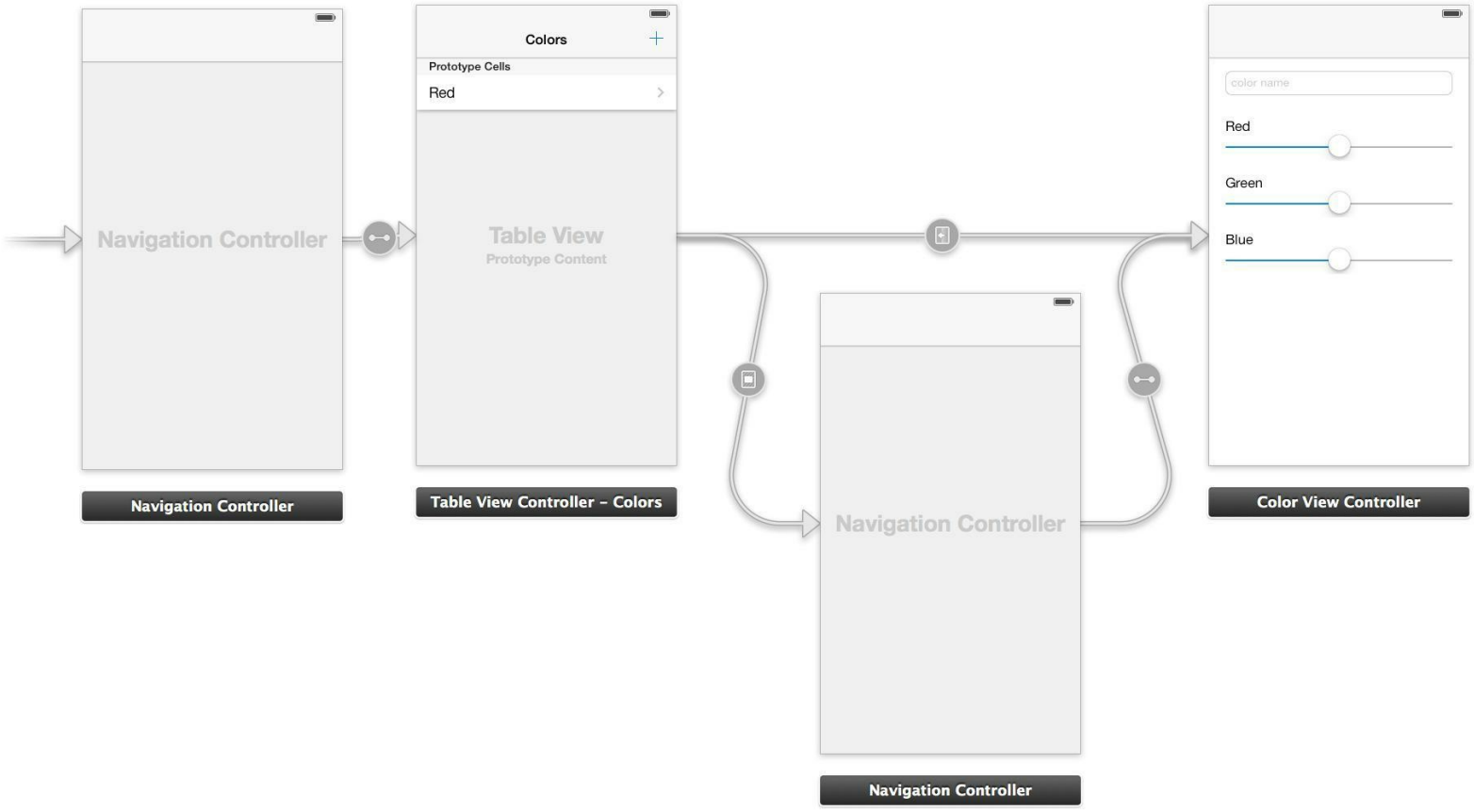


图28-19 动态原型

下面选中唯一的UITableViewCell对象，在Identifier文本框中输入UITableViewCell（见图28-20）。

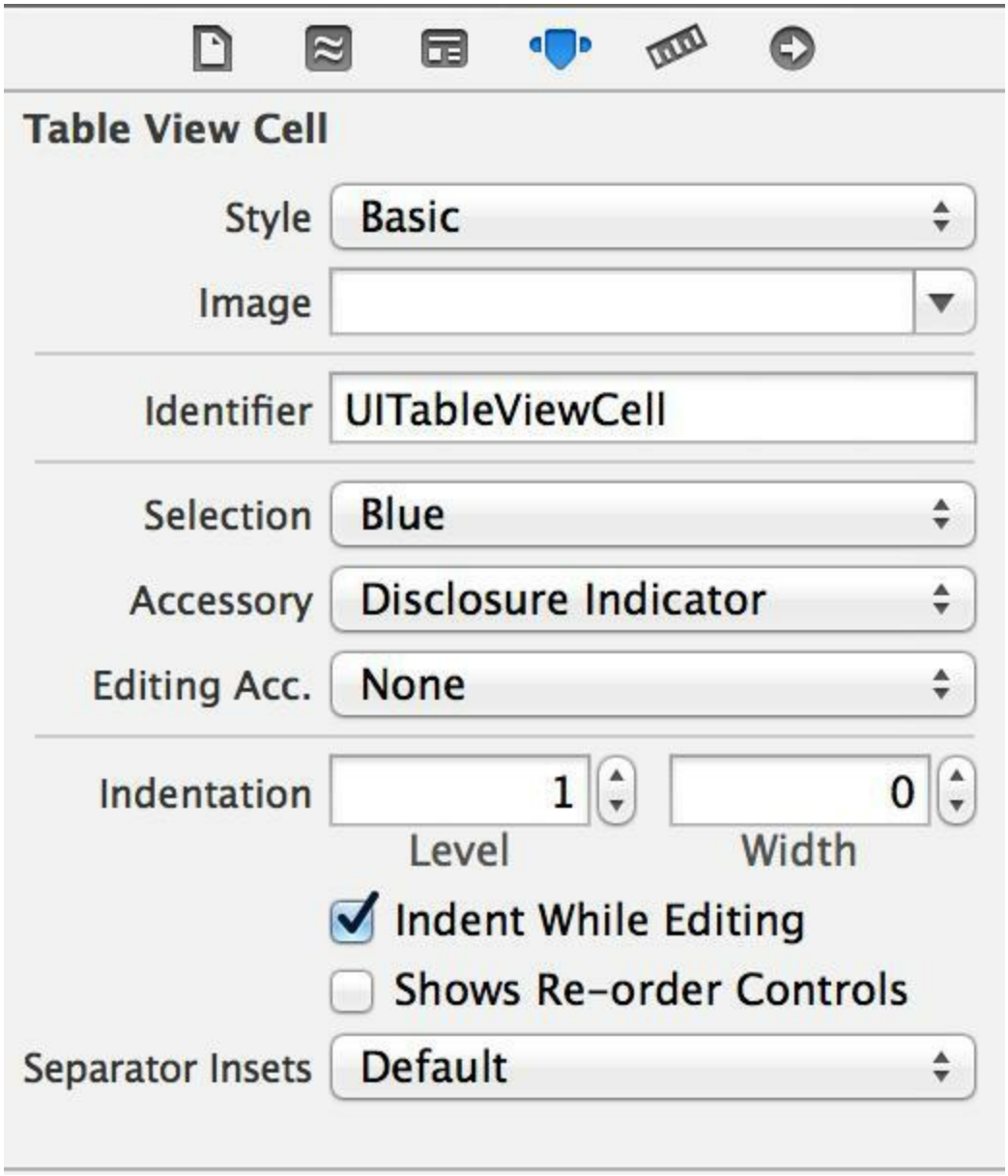


图28-20 设置UITableViewCell对象的重用标识

现在需要为Table View Controller创建对应的UITableViewController子类，以便为UITableView对象提供数据。创建一个新的NSObject子类，名为BNRPaletteView- Controller。

打开BNRPaletteViewController.h，将其父类修改为UITableViewController。

~~@interface BNRPaletteViewController : NSObject~~

@interface BNRPaletteViewController : UITableViewController

@end

再打开BNRPaletteViewController.m，导入BNRColorViewController.h，然后在类扩展中添加一个NSMutableArray属性。

#import “BNRPaletteViewController.h”

#import “BNRColorViewController.h”?

```
@interface BNRPaletteViewController（）
@property（nonatomic）NSMutableArray *colors;
@end
@implementation BNRPaletteViewController
```

接下来实现viewWillAppear:和UITableView对象的数据源方法：

```
-（void）viewWillAppear:（BOOL）animated
{
[super viewWillAppear:animated];
[self.tableView reloadData];
}
-（NSInteger）tableView:（UITableView *）tableView
numberOfRowsInSection:（NSInteger）section
{
return [self.colors count];
}
-（UITableViewCell *）tableView:（UITableView *）tableView
cellForRowAtIndexPath:（NSIndexPath *）indexPath
{
UITableViewCell *cell =
[tableView dequeueReusableCellWithIdentifier:@“UITableViewCell”
forIndexPath:indexPath];
return cell;
}
```

创建一个名为BNRColorDescription的NSObject子类，表示用户选择的颜色。

打开BNRColorDescription.h，添加一个UIColor属性和NSString属性，分别表示颜色和颜色名称：

```
@interface BNRColorDescription : NSObject

@property（nonatomic）UIColor *color;

@property（nonatomic, copy）NSString *name;

@end
```

再打开BNRColorDescription.m，覆盖init方法，设置两个属性的默认值：

```
@implementation BNRColorDescription

-（instancetype）init

{

self = [super init];

if（self）{

_color = [UIColor colorWithRed:0

green:0

blue:1

alpha:1];

_name = @“Blue”;

}

return self;

}

@end
```

为了测试代码是否可以正常工作，下面向BNRPaletteViewController的colors数组中添加一个BNRColorDescription对象。

在BNRPaletteViewController.m中引入BNRColorDescription.h，然后覆盖colors的取方法，添加一个BNRColorDescription对象。

```
#import “BNRPaletteViewController.h”
```

```
#import “BNRColorDescription.h”

@implementation BNRPaletteViewController

-（NSMutableArray *）colors

{

if（！_colors）{

_colors = [NSMutableArray array];

BNRColorDescription *cd = [[BNRColorDescription alloc] init];

[_colors addObject:cd];

}

return _colors;

}
```

同时，更新数据源方法，显示颜色名称：

```
-（UITableViewCell *）tableView:（UITableView *）tableView

cellForRowAtIndexPath:（NSIndexPath *）indexPath

{

UITableViewCell *cell =

[tableView dequeueReusableCellWithIdentifier:@“UITableViewCell”

forIndexPath:indexPath];

BNRColorDescription *color = self.colors[indexPath.row];

cell.textLabel.text = color.name;

return cell;

}
```

构建并运行应用。现在应用应该可以点击UITableViewCell对象进入BNRColorViewController界面，但是存在两个问题：首先，BNRColorViewController无法显示正确的颜色；其次，BNRColorViewController会同时显示Back（返回）按钮和Done按钮，正确做法是，只有在添加新颜色以模态形式进入BNRColorViewController时才显示Done按钮。为了解决

以上两个问题，需要在BNRPaletteViewController和BNRColorViewController之间传递需要的数据，包括BNRColorDescription对象以及该对象是否已经存在。

打开BNRColorViewController.h，在顶部导入BNRColorDescription.h，然后添加两个新属性：第一个用于判断编辑的是新颜色还是已经存在的颜色；第二个用于表示正在编辑的颜色：

```
#import “BNRColorDescription.h”

@interface BNRColorViewController : UIViewController

@property（nonatomic）BOOL existingColor;

@property（nonatomic）BNRColorDescription *colorDescription;

@end
```

当UIViewController对象触发UIStoryboardSegue对象时，它会收到prepareForSegue:sender:消息。prepareForSegue:sender:的两个参数分别是UIStoryboardSegue对象和动作控件。UIStoryboardSegue对象包含三个方面的信息：源视图控制器（source view controller）、目标视图控制器（destination view controller）和标识。为了区分不同的UIStoryboardSegue对象，下面为两个UIStoryboardSegue对象设置不同的标识。

重新打开Colorboard.storyboard，选中modal样式的UIStoryboardSegue对象，然后打开属性检视面板，在Identifier文本框中输入NewColor（新颜色）；同样，选中push样式的UIStoryboardSegue对象，设置其Identifier为ExistingColor（已经存在的颜色），如图28-21所示。

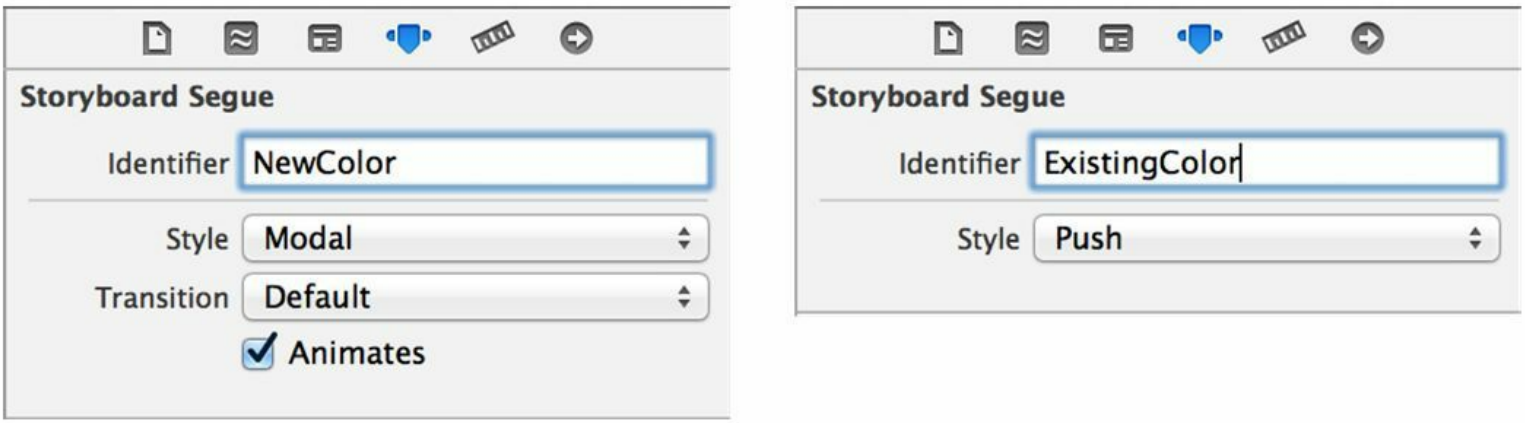


图28-21 为两个UIStoryboardSegue对象设置标识

现在，两个UIStoryboardSegue对象都具有标识，因此，可以在程序中根据标识判断用户是正在添加新颜色还是选择了已经存在的颜色，并使用不同的方式为BNRColorView- Controller对象传入BNRColorDescription对象。打开BNRPaletteViewController.m，实现prepareForSegue:sender:方法：

```
-（void）prepareForSegue:（UIStoryboardSegue *）segue sender:（id）sender

{
```

```
if ([segue.identifier isEqualToString:@“NewColor”]) {
// 如果是添加新颜色,
// 就创建一个BNRColorDescription对象并将其添加到colors数组中
BNRColorDescription *color = [[BNRColorDescription alloc] init];
[self.colors addObject:color];
// 通过UIStoryboardSegue对象
// 设置BNRColorViewController对象的颜色(colorDescription属性)
UINavigationController *nc =
(UINavigationController *)segue.destinationViewController;
BNRColorViewController *mvc =
(BNRColorViewController *)[nc topViewController];
mvc.colorDescription = color;
}
else if ([segue.identifier isEqualToString:@“ExistingColor”]) {
// 对于push样式的UIStoryboardSegue对象, sender是UITableViewCell对象
NSIndexPath *ip = [self.tableView indexPathForCell:sender];
BNRColorDescription *color = self.colors[ip.row];
// 设置BNRColorViewController对象的颜色,
// 同时设置其existingColor属性为YES(该颜色已经存在)
BNRColorViewController *cvc =
(BNRColorViewController *)segue.destinationViewController;
cvc.colorDescription = color;
cvc.existingColor = YES;
}
```

}

在prepareForSegue:sender:方法中，首先检查segue参数的identifier，确定触发的是哪个UIStoryboardSegue对象。如果用户点击了“＋”按钮，那么触发的segue是“NewColor”，需要创建一个新的BNRColorDescription对象并将其传给BNRColorView- Controller；如果用户选择了已经存在的颜色，那么触发的segue是“ExistingColor”，需要将用户选择的颜色传给BNRColorViewController（如果UIStoryboardSegue对象的动作控件是UITableViewCell对象，可以通过UITableViewCell对象知道用户选择的是哪个NSIndexPath对象）。

（请注意，“NewColor”的destinationViewController是UINavigationController对象，而“ExistingColor”的destinationViewController是BNRColorViewController对象。打开storyboard文件，可以看到，modal样式的UIStoryboardSegue对象指向一个UINavigationController对象，而push样式的则指向一个BNRColorViewController对象。push样式的UIStoryboardSegue对象直接将BNRColorViewController对象压入已经存在的UINavigationController栈中。）

下面还需要在BNRColorViewController中编写以下代码：如果用户选择的是已经存在的颜色，就不能显示Done按钮；需要根据colorDescription属性设置view的背景颜色、颜色名称和UISlider对象的滑块值；当BNRColorViewController的view从屏幕上消失时，需要保存用户选择的新颜色。

在BNRColorViewController.m中覆盖viewWillAppear:方法，如果颜色已经存在，就移除Done按钮：

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

{

[super viewWillAppear:animated];

// 如果颜色已经存在，就移除Done按钮

if (self.existingColor) {

self.navigationItem.rightBarButtonItem = nil;

}

}
```

接下来覆盖viewDidLoad方法，初始化view的背景颜色、颜色名称和UISlider对象的滑块值。

```
- (void)viewDidLoad

{

[super viewDidLoad];
```

```
UIColor *color = self.colorDescription.color;
// 从UIColor对象中取出RGB颜色分量
float red, green, blue;
[color getRed:&red
green:&green
blue:&blue
alpha:nil];
// 初始化UISlider对象的滑块值
self.redSlider.value = red;
self.greenSlider.value = green;
self.blueSlider.value = blue;
// 初始化view的背景颜色和颜色名称
self.view.backgroundColor = color;
self.textField.text = self.colorDescription.name;
}
```

最后，当view消失时保存颜色和颜色名称。

```
-（void）viewWillDisappear:（BOOL）animated
{
[super viewWillDisappear:animated];
self.colorDescription.name = self.textField.text;
self.colorDescription.color = self.view.backgroundColor;
}
```

构建并运行应用，Colorboard应该可以正确地显示并保存颜色。

28.6 Storyboards的优缺点

本章指导读者创建了一个Storyboard文件、创建并设置了若干视图控制器、为这些对象构建界面并创建了相应的UIStoryboardSegue对象。这些都是Storyboards的基础知识，虽然还可以为Storyboard文件加入一些其他类型的UIStoryboardSegue对象和UIViewController对象，但是基本原理都是一样的。使用Storyboards的好处是可以减少部分需要编写的代码。

例如，push样式的UIStoryboardSegue对象代替了下面这段代码：

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)ip
{
UIViewController *vc = [[UIViewController alloc] init];
[self.navigationController pushViewController:vc];
}
```

虽然Storyboards看上去很好用，但是本书作者认为Storyboards的缺点也不少。下面列举若干Storyboards的优缺点，其优点为：

- 可以很快地开发出原型，向客户或同事演示界面流程。
- 可以替代部分简单的代码。
- 能很方便地为UITableView对象创建静态的UITableViewCell对象。
- 自定义UITableViewCell对象时可以使用动态原型，不需要创建独立的XIB文件。
- 使用Xcode显示Storyboard文件时，看上去感觉很好。

其缺点为：

- 团队协作时，使用Storyboards会有困难。通常情况下，团队中的iOS程序员可以各自完成特定的UIViewController子类。但是，如果使用Storyboard文件来构建界面，那么所有人都必须使用同一个文件，不利于团队开发。
- 代码管理工具很难处理Storyboard文件。如果有两个人同时编辑一个Storyboard文件（这在团队开发中很常见），那么代码管理工具（例如Subversion和Git）就会报告发生冲突，需要人工解决。
- 会让简单的开发流程复杂化。假设要为某个应用编写一个UIViewController子类，然后为该应用的某个按钮实现动作方法，以模态的形式显示该子类的对象。如果直接通过编写代码来实现上述功能，就只需要创建UIViewController子类的对象（使用alloc方法和init方法），然后向

当前的视图控制器(self)发送presentViewController: animated:completion:即可。如果通过Storyboards来实现上述功能，就需要先打开Storyboard文件、拖曳一个UIViewController对象、在标识检视面板中将Class修改为新的UIViewController子类，最后创建并设置UIStoryboardSegue对象。

•使用Storyboards开发应用时，会降低开发的灵活性。当读者要编写代码来实现某些功能时，Storyboard文件不仅无法提供帮助，反而会阻碍实现自定义代码。也就是说，和“在Storyboard文件所提供的基础功能上增加高级功能”相比，“直接用代码来实现基础功能和高级功能”的工作量反而可能更少。

•Storyboards总是会创建新的UIViewController对象。每当应用执行一个UIStoryboardSegue对象，就会创建一个新的目标UIViewController对象。但是在某些情况下，可能需要保留并重用某个UIViewController对象，而不是在关闭该对象时就释放相应的对象。Storyboards不支持重用UIViewController对象。

总之，Storyboards能让简单的代码更简单，复杂的代码更复杂。本书的例子代码没有使用Storyboards，而且作者自己在开发iOS应用时，也没有使用Storyboards。这里已经列出了Storyboards的优缺点，读者在开发iOS应用时，需要根据具体的情况自行决定是否使用Storyboards。

28.7 深入学习：状态恢复

本书第24章介绍了在不使用Storyboards的情况下恢复应用状态，本章介绍如何通过storyboard文件实现状态恢复。

如果在Storyboards中使用状态恢复系统，可以避免编写大量样板代码（boilerplate code）。例如，视图控制器的恢复标识可以直接在Storyboard文件中设置。通常可以将视图控制器的Storyboard标识作为恢复标识，如图28-22所示。

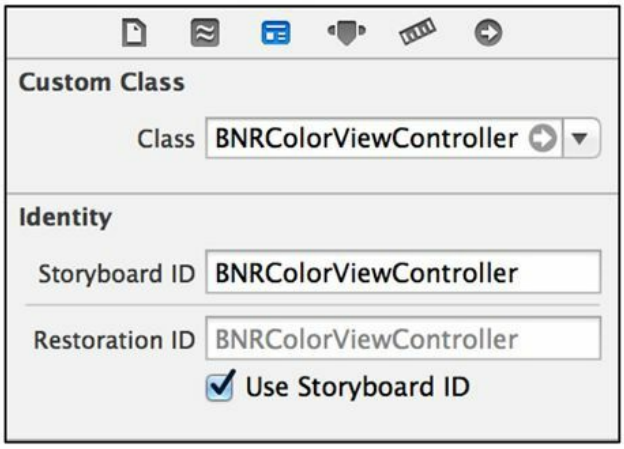


图28-22 Storyboard标识与恢复标识

首先，如果要启用状态恢复，同样要在应用委托中实现保存和恢复应用状态的两个委托方法：

```
@implementation BNRAppDelegate

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

接下来同样需要使UIViewController子类遵守UIViewControllerRestoration协议，并实现协议中的方法，返回相应的UIViewController子类对象。区别是，在Storyboards中，需要通过Storyboard文件实例化UIViewController子类，代码如下：

```
+（UIViewController *）
viewControllerWithRestorationIdentifierPath:（NSArray *）path
coder:（NSCoder *）coder
{
BNRColorViewController *vc = nil;
UIStoryboard *storyboard = [coder decodeObjectForKey:
UIStateRestorationViewControllerStoryboardKey];
if（storyboard）
{
vc =（BNRColorViewController *）[storyboard
instantiateViewControllerWithIdentifier:
@“BNRColorViewController”];
vc.restorationIdentifier = [identifierComponents lastObject];
vc.restorationClass = [BNRColorViewController class];
}
return vc;
}
```

NSCoder对象编码了包含视图控制器的Storyboard文件，可以通过UIStateRestorationViewControllerStoryboardKey对Storyboard文件解码，得到UIStoryboard对象。UIStoryboard提供了instantiateViewControllerWithIdentifier:方法，可以根据Storyboard标识创建相应的视图控制器。

除此之外，实现状态恢复的其余代码都是相同的。如果视图控制器要保存和恢复状态信息，同样要实现encodeRestorableStateWithCoder:和decodeRestorableStateWithCoder:方法。

第29章 后记

本书至此已近尾声。感谢读者，也恭喜读者阅读完了全书。下面有两则消息，一好一坏。

- 好消息：读者已经入门iOS开发。
- 坏消息：读者才刚入门iOS开发。

29.1 接下来做什么

读者可以尝试编写代码并从错误中获取经验，以及参阅一些繁杂庞大的开发文档，或者有机会的话向精通iOS开发的程序员求教。下面提供若干建议。

尽快开始编写应用。刚学到的知识如果不用，就会慢慢忘记。建议读者多做练习并扩充现有的知识。

深入。本书倾向广度而非深度。如果深入讲解，那么之前的每一章都可以独立成书。如果读者对某些章节特别感兴趣，想有更深入的了解，则可以自己尝试阅读Apple的相关文档并阅读一些博客文章（或StackOverflow问答）。

交流。很多城市会有iOS开发者聚会（iOS Developer Meetup，这里指美国），其中的演讲都很出色。此外，网上也有一些讨论组可以参与。如果读者正在开发项目，不妨找些人来帮忙，例如设计师、测试员及其他程序员。

犯错然后修正。如果读者不满意自己的代码，则可以推倒重来，从“失败”中汲取经验教训，并改用更好的设计架构。有人将这种修改过程称为重构（refactoring），读者会在重构过程中学到很多知识。

回馈。分享知识；礼貌地回答别人提出的“弱智”问题；公开一些源代码。

29.2 结束语

本书的三位作者都有Twitter账号：@aaronhillegass、@cbkeur和@joeconwaystk。

Big Nerd Ranch还会陆续推出其他开发书籍，敬请读者留意。此外，Big Nerd Ranch提供为期一周的程序员开发课程。最后，如果读者有编写应用的需求，Big Nerd Ranch也提供外包服务。详情请访问Big Nerd Ranch网站：http://www.bignerdranch.com/。

最后，感谢您，我们的读者。因为有你们，我们才能以写作、编程和教学为生。感谢您购买本书。