

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



华中科技大学出版社  
<http://www.hustp.com>

# iOS 编程(第 4 版)

Christian Keur  
[美] Aaron Hillegass 著  
Joe Conway

丁道骏 译  
张召 吴春燕 审校

华中科技大学出版社  
中国·武汉

# 前言

## Introduction

要成为一名优秀的 iOS 程序员，需要攻克以下三道难题。

- 必须学会 Objective-C 语言。Objective-C 是 C 语言的扩展，小巧简单。读完本书的前 3 章，读者就能掌握 Objective-C 的基础知识。
- 必须掌握 Cocoa 的常用技术。其中包括内存管理、委托机制（delegation）、固化机制（archiving），以及如何正确使用视图控制器（view controller）。理解这些技术需要花些时间。本书前半部分会介绍这些内容。
- 必须掌握框架（framework）。读者的最终目标是了解 iOS 的所有框架，学会如何使用框架中的每一个类和方法。但这几乎是不可能完成的任务：iOS 有 3000 多个方法，200 多个类。而且随着 iOS 的版本升级，Apple 还会不断地加入新的类和新方法。本书将会介绍 iOS SDK 中的各个组成部分，但是不会太过深入。作者的目标是带领读者入门，使读者能够自行阅读并理解 Apple 的参考文档。

Big Nerd Ranch 公司采用本书作为“iOS 新手培训课程”的教材。这些内容经过了长期的检验，并且帮助很多人成为 iOS 程序员。真心希望本书也能给你带来帮助。

## 本书适合哪些读者

本书假设读者已经跃跃欲试，准备开发 iOS 应用，所以不会花费笔墨去证明 iPhone、iPad 和 iPod touch 是很棒的产品。本书假设读者了解 C 语言并对“面向对象编程”有一定的了解。如果不是，建议读者先阅读 Big Nerd Ranch 出版的《Objective-C 编程》（华中科技大学出版社）。

## 第 4 版有哪些更新

第 4 版使用的是 Xcode 5，并且所有的应用都运行于安装有 iOS 7 的设备或模拟器。

作者在第 4 版采用了更加现代化的语法编写 Objective-C 代码，使用属性、点语法、实例变量自动合成和字面量，并更多地使用 Block 对象。

第 4 版还加入了众多新章节，分别介绍第 3 版以来 Apple 为 iOS 新加入的特性，包括自动布局、动态字体、状态恢复、Block 动画和 `NSURLSession` 等。

除了以上这些明显的变化，作者还根据读者和学生提出的问题对本书进行了大量修订。可

以说与前一版本相比，新版页页有改进。

## 教学理念

本书将向读者传授 iOS 编程的基本概念。在阅读的同时，读者还要输入大量的代码，并创建一组应用。这样，完成本书的学习后，读者得到的将不仅是知识，还有经验。相信你我多少都有过被灌输知识的痛苦经历，所以本书转而使用“边做边学”的教学方法——概念与代码并重。

多年的 iOS 编程教学工作让作者了解到：

- 某些概念是学习 iOS 编程必须知道的。本书会集中介绍这部分内容。
- 学习能立即派上用处的概念，效果最佳。
- 知识与经验并重时，学习效果最佳。
- 实际操作很重要。本书会要求读者先输入代码，再理解含义。读者可能会觉得这种不明就里的模仿意义不大，但是“找出错误并修正代码”是学习编程的好方法。这种最基础的调试过程不仅不是累赘，反而能帮助读者彻底理解代码。这也是作者鼓励读者自己输入代码的原因。虽然可以直接下载例子代码，但是拷贝粘贴不是编程。本书对读者及读者的编程技能有更高的期望。

这种模仿对读者意味着什么？意味着读者要信任本书作者，而且要有耐心。本书会尽可能地将问题讲透，但读者有时只能相信作者的意见（如果读者对此有异议，请往下看——作者列出了若干解决方案，也许能有帮助）。遇到暂时不能理解的概念时不要气馁，因为本书不会将涉及某个概念的所有知识一次全部介绍完，这是有意为之。如果某个概念没有解释清楚，那么很可能在需要时再提供更详细的介绍。有些初看无法理解的概念，可能会在读者第一次（或第十二次）实际应用时突然变得清晰易懂。

每个人的学习方法不同。读者可能会喜欢本书这种“按需分步介绍概念”的方法，也可能不喜欢。如果是后者，这里提供若干建议。

- 先不要着急，等待在后续的章节中将问题讲透。
- 查阅索引，先阅读相关概念的详细介绍。
- 查阅 Apple 提供的在线文档。这些文档是非常重要的开发工具，需要多多练习使用。读者应该尽早地，也应尽可能多地使用在线文档。
- 如果在学习 Objective-C 或面向对象的编程概念时遇到困难（或是预感会有困难），那么建议读者先阅读 Big Nerd Ranch 的《Objective-C 编程》（华中科技大学出版社出版）。



## 如何使用本书

本书内容基于 Big Nerd Ranch 的培训课程，所以有其特定的阅读方法。

读者可以先设定一个合理的目标，例如“每天阅读一章”。然后在阅读时为自己找一个安静的场所，至少一个小时内不会被打断。关掉 Email、Twitter 客户端和聊天工具——读书无法多任务并行，必须集中精力。

在阅读的同时，读者还需要编写代码。根据读者的喜好，可以先通读整章。但是真正的学习从编写代码开始。要真正理解某个概念，需要编写程序并进行实际操作，尤其是调试（debug）程序的过程，会特别有帮助。

书中的部分项目需要使用额外的文件，例如第 1 章的 Quiz 应用需要一个图标文件。本书已经为读者准备好了这些文件，通过以下网址可以下载本书的项目代码和资源文件：  
<https://github.com/dingdaojun/iOSProgramming4ed>。

学习分两类。学习历史时，要做的只是在已经理解的知识架构上添加更多细节。我们将这类学习称为简单学习（easy learning）。学习历史的确需要花费很长的时间，但是难度不大。学习 iOS 编程则是困难学习（hard learning），会经常卡壳，尤其是在刚开始的时候。作者编写本书的目的是帮助读者越过难度陡增的学习曲线。下面提供两则建议，希望帮助读者轻松越过障碍。

- 找位懂 iOS 开发并且愿意回答读者提问的程序员。第一次尝试将应用安装至 iOS 设备时，如果缺少有经验的程序员的协助，就可能会遇到困难。
- 保证足够的睡眠。缺少睡眠将无法记住所学的知识。

## 本书是如何组织的

本书各章都会先介绍一个或多个 iOS 开发概念，然后给出具体的示例代码。每章末尾有练习，为读者提供编写代码的机会。建议读者至少完成部分的练习，以巩固所学，同时也提升自信。此外，多数章节最后会有一到两个“深入学习”部分，对之前介绍的内容做一些补充。

第 1 章介绍如何创建并安装一个很小的应用，以此带领读者入门 iOS 开发。读者将开始学习如何使用 Xcode 和 iOS 模拟器，以及创建项目和文件所需的全部步骤。第 1 章还介绍了 Model-View-Controller 及其在 iOS 开发中的作用。

第 2、3 章简单介绍 Objective-C 和内存管理。这两章没有创建 iOS 应用，而是构建并调试了一个名为 RandomItems 的工具，以帮助读者理解这些概念。

第 4、5 章将创建一个名为 Hypnosister 的应用，介绍视图和视图层次结构，并使用它们构建用户界面。

第 6、7 章通过 HypnoNerd 应用介绍如何使用视图控制器管理用户界面。读者将有机会练习如何使用视图和视图控制对象，并通过 UITabBar 对象切换屏幕。另外，读者还将获得大量与委托机制有关的经验（委托机制是一种重要的设计模式），并学习使用协议（protocol）、调

试器和本地通知。

第 8 章介绍本书最庞大的应用 **Homepwner**（顺便提一句，**Homepwner** 并没有拼错。读者可以通过 [www.urbandictionary.com](http://www.urbandictionary.com) 找到 **pwn** 的定义）。**Homepwner** 的功能是保留一份财产清单，以供发生灾难后核对。本书将花 14 章的篇幅来介绍 **Homepwner**。

第 8、9 和 19 章介绍如何使用并显示表格。读者将学到 **UITableView**、**UITableViewController** 和数据源机制；学会如何在表格视图中显示数据、如何让用户编辑表格及如何改善相关的界面。

第 10 章以第 6 章的导航机制为基础，介绍 **UINavigationController** 的用法，并为 **Homepwner** 增加一个垂直（drill-down）界面和一个导航条。

第 11 章介绍如何通过相机拍照，以及如何为 **Homepwner** 实现显示图片和保存图片的功能。读者将有机会使用 **NSDictionary** 和 **UIImagePickerController**。

第 12、13 章将暂时跳出 **Homepwner** 应用，创建一个名为 **TouchTracker** 的绘图应用，以介绍触摸事件。读者将学到如何为应用增加多点触摸功能，以及如何使用 **UIGestureRecognizer** 来响应特定的手势。此外，读者还将了解第一响应对象（first responder）和响应对象链（responder chain），并针对 **NSDictionary** 做更多练习。

第 14 章将向读者介绍如何使用调试仪表、Instruments 和静态分析器（static analyzer），并借此优化 **TouchTracker** 应用的性能。

第 15、16 章会将 **Homepwner** 改为通用应用（universal application），能在 iPhone 和 iPad 上全屏运行。同时，读者还会学习自动布局（auto layout），使 **Homepwner** 的界面可以适配不同大小的屏幕。

第 17 章介绍如何实现自动转屏（autorotation），以及如何使用模式视图控制器和 iPad 特有的 **UIPopoverController**。

第 18 章将介绍多种保存/读取数据的机制。值得注意的是，**Homepwner** 应用会通过 **NSCoding** 协议固化（archive）数据。本章还介绍了切换应用状态（例如激活状态、后台运行状态和悬停状态）的各个过程。

第 20 章将升级 **Homepwner** 应用，支持动态字体（dynamic type），根据用户的偏好显示不同大小的字体。

第 21 章会再次跳出 **Homepwner** 应用，通过创建一个名为 **Nerdfeed** 的应用，介绍如何实现 Web 服务通信。**Nerdfeed** 会通过 **NSURLSession** 和 **NSJSONSerialization** 从指定的服务器获取并解析 JSON 数据。JSON 是目前最流行的数据传输格式。此外，**Nerdfeed** 还会用 **UIWebView** 对象来显示相应的 Web 页面。

第 22 章将介绍 **UISplitViewController**，并为 **Nerdfeed** 添加 **UISplitViewController** 对象，以适应 iPad 更大的屏幕尺寸。

第 23 章介绍 Core Data。**Homepwner** 应用将改用 **NSManagedObjectContext** 对象来保存

/读取数据。

第 24 章将为 **Homepwner** 启用状态恢复，无论用户何时重新回到应用，都会看到自己最后一次看到的界面。

第 25 章将介绍与国际化、本地化有关的概念和技术。读者将学习如何使用 **NSLocale**、字符串对照表（strings table）和 **NSBundle**，完成 **Homepwner** 应用的本地化。

第 26 章将介绍如何通过 **NSUserDefaults** 来永久保存用户偏好设置（user preferences）。本章将彻底完成 **Homepwner** 应用的开发。

第 27 章将介绍 **Block** 动画，并为 **HypnoNerd** 应用添加动画效果。读者将学习几种简单又有趣的动画效果，包括 iOS 7 中新引入的弹簧动画。

第 28 章将介绍 **storyboard**，一项能够帮助读者构建应用的 iOS 新特性。读者将学习如何使用 **storyboard** 文件来构建应用，以及 **storyboard** 的优缺点。

## 代码风格

本书包含大量代码，作者希望这些代码及其背后的设计思路能具备参考价值。作者在编写这些代码时，已经尽可能地符合 Cocoa 编程习惯。尽管如此，其中的某些部分还是会和 Apple 的示例代码，或者其他书籍中的代码有一定的差别。虽然读者目前可能还无法理解这些差别的具体含义，但是在正文开始前，还是有必要将这些问题先列出来。

- 本书采用编写代码或 **XIB** 文件的方式来创建视图控制器。还有一种方式是 **storyboard** 文件，本书第 28 章会介绍 **storyboard** 文件及其用法。但是作者自己在 **Big Nerd Ranch** 开发 iOS 应用时，几乎没有用过 **storyboard** 文件。
- 本书的绝大部分项目都创建自最简单的 **Xcode** 模板：**empty application**（空应用）。这里没有采用其他模板，是因为这些模板会自动生成额外代码，不利于初学者理解应用的工作原理。

## 版式说明

为了方便读者阅读，本书会对某些特定的内容使用专门的字体。其中，类名、方法名和函数名会以粗体、等宽的字体显示。类名的首字母会用大写，方法名的首字母会用小写。正文中出现的类名、方法名和函数名将采用加粗的 **Bitstream Vera Sar** 字体表示，例如，“在 **BNRRexViewController** 类的 **loadView** 方法里，使用 **NSLog** 函数输出结果。”

正文中出现的变量、常量和类型将采用 **Bitstream Vera Sar** 字体表示，但是不加粗。例如，“将变量 **fido** 定义为 **float** 类型，并赋予初始值 **M\_PI**。”

应用程序和系统菜单选项将采用 **Helvetica** 字体表示，例如，“打开 **Xcode**，从 **File** 菜单选择 **New Project...**”。

书中的所有示例代码都将采用 Bitstream Vera Sar 字体表示。需要读者输入的代码会加粗显示。例如下面这段代码，除了第一行和最后一行外（这两行是已经存在的代码，列出来是为了让读者知道应该在哪里加入新的代码），其余部分都需要读者自行输入。

```
@interface BNRQuizViewController ()  
  
@property (nonatomic, weak) IBOutlet UILabel *questionLabel;  
@property (nonatomic, weak) IBOutlet UILabel *answerLabel;  
  
@end
```

## 开发所需的硬件与软件

开发 iOS 应用要使用（也只能使用）基于 Intel 芯片的 Mac 计算机。读者需要下载由 Apple 提供的 iOS SDK，其中包括 Xcode（Apple 的集成开发环境）、iOS 模拟器，以及其他开发工具。

读者还需要注册加入 Apple 的 iOS 开发者计划，费用为每年 99 美元。原因有以下三个。

- 注册后可以免费下载最新的开发工具。
- 任何应用必需先经过数字签名，然后才能在设备上运行。因为只有注册会员才能签名应用，所以，如果读者需要在设备上测试应用，就需要注册。
- 只有会员才能将应用提交至 App Store。

如果读者打算花时间读完本书的全部内容，那么注册加入 iOS 开发者计划是值得的。注册网站：<http://developer.apple.com/programs/ios/>。

进行 iOS 开发需要准备哪些设备？本书前半部分所涉及的多数应用都是针对 iPhone 的，但是也能在 iPad 上运行。在 iPad 上运行 iPhone 应用时，系统只会以 iPhone 的屏幕尺寸显示。这样虽然无法充分利用 iPad 的大尺寸屏幕，但是在没有 iPhone 的情况下也是可以的。本书的前几章会集中介绍 iOS SDK 的基础部分，这些内容和设备无关。稍后的章节会介绍若干只针对 iPad 的内容，以及如何让应用能够在所有的 iOS 设备上全屏运行。

读者是否已经准备好了呢？下面开始正文。

# 目录

## Table of Contents

前言 .....	xiii
本书适合哪些读者 .....	xiii
第 4 版有哪些更新 .....	xiii
教学理念 .....	xiv
如何使用本书 .....	xv
本书是如何组织的 .....	xv
代码风格 .....	xvii
版式说明 .....	xvii
开发所需的硬件与软件 .....	xviii
第 1 章  第一个简单的 iOS 应用 .....	1
1.1 创建 Xcode 项目 .....	2
1.2 模型-视图-控制器 .....	4
1.3 设计 Quiz .....	5
1.4 创建视图控制器 .....	6
1.5 创建界面 .....	8
1.6 创建关联 .....	14
1.7 创建模型对象 .....	18
1.8 大功告成 .....	21
1.9 在模拟器上运行应用 .....	22
1.10 安装应用 .....	23
1.11 应用图标 .....	25
1.12 启动图片 .....	27
第 2 章  Objective-C .....	29
2.1 对象 .....	29
2.2 使用对象 .....	30
2.3 编写命令行工具 RandomItems .....	33
2.4 创建 Objective-C 类的子类 .....	38
2.5 深入学习 NSArray 与 NSMutableArray .....	58
2.6 异常与未知选择器 .....	60

2.7 练习 .....	62
2.8 初级练习：查找问题 .....	62
2.9 中级练习：另一个初始化方法 .....	62
2.10 高级练习：另一个类 .....	63
2.11 关于深入学习部分 .....	63
2.12 深入学习：如何为类命名 .....	63
2.13 深入学习：#import 和 @import .....	64
<b>第 3 章 通过 ARC 管理内存 .....</b>	<b>65</b>
3.1 栈 .....	65
3.2 堆 .....	66
3.3 指针变量与对象所有权 .....	66
3.4 强引用与弱引用 .....	70
3.5 属性 .....	74
3.6 深入学习：属性合成 .....	81
3.7 深入学习：Autorelease 池与 ARC 历史 .....	83
<b>第 4 章 视图与视图层次结构 .....</b>	<b>85</b>
4.1 视图基础 .....	86
4.2 视图层次结构 .....	86
4.3 创建 UIView 子类 .....	88
4.4 在 drawRect:方法中自定义绘图 .....	94
4.5 关于开发者文档 .....	105
4.6 初级练习：绘制图像 .....	106
4.7 深入学习：Core Graphics .....	106
4.8 高级练习：阴影和渐变 .....	108
<b>第 5 章 视图：重绘与 UIScrollView .....</b>	<b>111</b>
5.1 运行循环和重绘视图 .....	112
5.2 类扩展 .....	114
5.3 使用 UIScrollView .....	115
<b>第 6 章 视图控制器 .....</b>	<b>119</b>
6.1 创建 UIViewController 子类 .....	120
6.2 另一个视图控制器 .....	123
6.3 UITabBarController .....	130
6.4 视图控制器的初始化方法 .....	134
6.5 添加本地通知 .....	135
6.6 加载和显示视图 .....	136

---

6.7	与视图控制器及其视图进行交互 .....	138
6.8	初级练习：增加一个标签项 .....	139
6.9	中级练习：控制逻辑 .....	139
6.10	深入学习：键值编码 .....	139
6.11	深入学习：Retina 显示屏 .....	140
<b>第 7 章</b>	<b>委托与文本输入 .....</b>	<b>143</b>
7.1	文本框（UITextField） .....	143
7.2	委托 .....	146
7.3	协议 .....	148
7.4	向屏幕中添加 UILabel 对象 .....	150
7.5	运动效果 .....	152
7.6	使用调试器 .....	153
7.7	深入学习：main()与 UIApplication .....	157
7.8	中级练习：捏合-缩放 .....	157
<b>第 8 章</b>	<b>UITableView 与 UITableViewController .....</b>	<b>159</b>
8.1	编写 Homepwner 应用 .....	159
8.2	UITableViewController .....	160
8.3	UITableView 数据源 .....	164
8.4	UITableViewCell 对象 .....	170
8.5	代码片段库 .....	175
8.6	初级练习：表格段 .....	178
8.7	中级练习：固定行 .....	178
8.8	高级练习：修改 UITableView 对象的外观 .....	178
<b>第 9 章</b>	<b>编辑 UITableView .....</b>	<b>179</b>
9.1	编辑模式 .....	179
9.2	增加行 .....	185
9.3	删除行 .....	187
9.4	移动行 .....	188
9.5	初级练习：更改“删除”按钮的标题 .....	190
9.6	中级练习：禁止移动某个表格行 .....	190
9.7	高级练习：彻底禁止移动某个表格行 .....	190
<b>第 10 章</b>	<b>UINavigationController .....</b>	<b>191</b>
10.1	UINavigationController 对象 .....	192
10.2	额外的视图控制器 .....	196
10.3	UINavigationController 的导航功能 .....	201

10.4	UINavigationController .....	205
10.5	初级练习：显示数字键盘 .....	210
10.6	中级练习：关闭数字键盘 .....	210
10.7	高级练习：压入更多视图控制器 .....	210
<b>第 11 章</b>	<b>相机 .....</b>	<b>211</b>
11.1	通过 UIImageView 对象显示照片 .....	212
11.2	通过 UIImagePickerController 拍摄照片 .....	216
11.3	创建 BNRImageStore.....	220
11.4	NSDictionary.....	222
11.5	创建并使用键 .....	225
11.6	使用 BNRImageStore.....	227
11.7	关闭键盘 .....	228
11.8	初级练习：编辑照片 .....	230
11.9	中级练习：删除照片 .....	230
11.10	高级练习：Camera Overlay .....	230
11.11	深入学习：导航实现文件.....	230
11.12	深入学习：摄像 .....	233
<b>第 12 章</b>	<b>触摸事件与 UIResponder .....</b>	<b>235</b>
12.1	触摸事件 .....	235
12.2	创建 TouchTracker 应用 .....	237
12.3	实现 BNRDrawView，完成绘图功能 .....	239
12.4	处理触摸事件并创建线条对象 .....	240
12.5	初级练习：保存与读取 .....	245
12.6	中级练习：颜色 .....	245
12.7	高级练习：圆圈 .....	246
12.8	深入学习：响应对象链 .....	246
12.9	深入学习：UIControl.....	247
<b>第 13 章</b>	<b>UIGestureRecognizer 与 UIMenuController .....</b>	<b>249</b>
13.1	UIGestureRecognizer 子类 .....	250
13.2	用 UITapGestureRecognizer 对象识别“按下”手势 .....	250
13.3	同时添加多种触摸手势 .....	252
13.4	UIMenuController .....	254
13.5	UILongPressGestureRecognizer.....	256
13.6	UIPanGestureRecognizer 以及同时识别多个手势 .....	257
13.7	深入学习：UIMenuController 与 UIResponderStandardEditActions.....	260



---

13.8	深入学习：再谈 <code>UIGestureRecognizer</code> .....	261
13.9	中级练习：修正错误 .....	262
13.10	高级练习：速度与宽度 .....	262
13.11	高级练习：颜色 .....	262
第 14 章	调试工具.....	263
14.1	仪表.....	263
14.2	Instruments .....	265
14.3	静态分析器 .....	275
14.4	项目、目标和构建设置 .....	277
第 15 章	自动布局入门.....	283
15.1	通用化 <code>Homepwner</code> .....	283
15.2	自动布局系统 .....	285
15.3	在 <code>Interface Builder</code> 中添加约束 .....	289
15.4	调试约束问题 .....	298
15.5	初级练习：打造完美界面 .....	306
15.6	中级练习：通用化 <code>Quiz</code> .....	307
15.7	深入学习：使用 <code>_autolayoutTrace</code> 方法调试约束问题 .....	307
15.8	深入学习：使用多个 <code>XIB</code> 文件 .....	308
第 16 章	在代码中使用自动布局 .....	309
16.1	视觉化格式语言 .....	310
16.2	创建约束 .....	311
16.3	添加约束 .....	312
16.4	固有内容大小 .....	315
16.5	另一种添加方式 .....	316
16.6	深入学习： <code>NSAutoresizingMaskLayoutConstraint</code> .....	318
第 17 章	自动转屏， <code>UIPopoverController</code> 与模态视图控制器 .....	321
17.1	自动转屏 .....	321
17.2	自动转屏通告机制 .....	324
17.3	<code>UIPopoverController</code> .....	326
17.4	更多的模态视图控制器 .....	329
17.5	线程安全的单例 .....	337
17.6	初级练习：为另一个类添加线程安全的单例 .....	339
17.7	高级练习： <code>UIPopoverController</code> 对象的外观 .....	339
17.8	深入学习：位掩码 .....	339
17.9	深入学习：视图控制器之间的关系 .....	340

第 18 章 保存、读取与应用状态 .....	345
18.1 固化 .....	345
18.2 应用沙盒 .....	348
18.3 NSKeyedArchiver 与 NSKeyedUnarchiver .....	350
18.4 应用状态与状态切换 .....	353
18.5 通过 NSData 将数据写入文件 .....	356
18.6 NotificationCenter 和内存过低警告 .....	358
18.7 模型-视图-控制器-存储设计模式 .....	361
18.8 初级练习: PNG .....	361
18.9 深入学习: 应用的状态切换 .....	362
18.10 深入学习: 文件系统的读取和写入 .....	363
18.11 深入学习: 应用程序包 .....	366
第 19 章 创建 UITableViewCell 子类 .....	369
19.1 创建 BNRItemCell .....	369
19.2 处理图片 .....	377
19.3 由 UITableViewCell 对象转发动作消息 .....	380
19.4 捕获变量 .....	385
19.5 初级练习: 设置颜色 .....	386
19.6 高级练习: 缩放 .....	387
19.7 深入练习: UICollectionView .....	387
第 20 章 动态字体 .....	389
20.1 使用用户首选字体 .....	390
20.2 响应用户首选字体的改变 .....	392
20.3 修改自动布局约束 .....	393
20.4 确定用户首选字体大小 .....	395
20.5 修改 BNRItemCell .....	397
第 21 章 Web 服务与 UIWebView .....	403
21.1 Web 服务 .....	404
21.2 UIWebView .....	414
21.3 认证信息 .....	416
25.4 中级练习: 加强 UIWebView .....	418
21.5 高级练习: 课程预告 .....	418
21.6 深入学习: HTTP 请求主体 .....	419
第 23 章 Core Data .....	431
23.1 对象-关系映射 .....	431

---

23.2	用 Core Data 重写 BNRItemStore 的数据保存功能.....	432
23.3	再谈 SQL .....	450
23.4	Faults .....	451
23.5	各种存取机制的优缺点 .....	453
23.6	初级练习: Asset 的 iPad 界面 .....	454
23.7	中级练习: 增加 BNRAsetType 对象 .....	454
23.8	高级练习: 显示某种类型的 BNRItem 对象 .....	454
<b>第 24 章</b>	<b>状态恢复 .....</b>	<b>455</b>
24.1	状态恢复的工作原理 .....	455
24.2	启用状态恢复 .....	456
24.3	恢复标识和恢复类 .....	457
24.4	状态恢复与应用生命周期 .....	459
24.5	恢复视图控制器 .....	461
24.6	编码状态数据 .....	464
24.7	保存视图状态 .....	465
24.8	中级练习: 为另一个应用启用状态恢复 .....	467
24.9	深入学习: 设置快照 .....	467
<b>第 25 章</b>	<b>本地化 .....</b>	<b>469</b>
25.1	通过 NSNumberFormatter 实施国际化 .....	470
25.2	资源的本地化 .....	473
25.3	NSLocalizedString()与字符串对照表 .....	477
25.4	初级练习: 再添加一套本地化资源 .....	480
25.5	深入学习: NSBundle 在国际化过程中的作用 .....	480
25.6	深入学习: 不通过基础国际化对 XIB 文件实施本地化 .....	481
<b>第 26 章</b>	<b>NSUserDefaults .....</b>	<b>483</b>
26.1	NSUserDefaults .....	483
26.2	设置束 .....	487
<b>第 27 章</b>	<b>控制动画 .....</b>	<b>491</b>
27.1	基础动画 .....	491
27.2	关键帧动画 .....	494
27.3	在动画完成后执行特定的代码 .....	496
27.4	弹簧动画 .....	497
27.5	中级练习: 提升 Quiz 的用户体验 .....	498
<b>第 28 章</b>	<b>UIStoryboard .....</b>	<b>499</b>
28.1	创建 Storyboard 文件 .....	499

28.2	Storyboard 文件中的 UITableViewController .....	503
28.3	Segue .....	506
28.4	改变颜色 .....	512
28.5	传递数据 .....	513
28.6	Storyboards 的优缺点 .....	520
28.7	深入学习：状态恢复 .....	521
第 29 章	后记 .....	523
29.1	接下来做什么 .....	523
29.2	结束语 .....	524
第 22 章	UISplitViewController .....	421
22.1	在 Nerdfeed 中使用 UISplitViewController .....	422
22.2	在竖排模式下显示主视图控制器 .....	425
22.3	将 Nerdfeed 改为通用应用 .....	428
索引	.....	525

## 第1章

# 第一个简单的 iOS 应用

## A Simple iOS Application

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）。



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

### Creating an Xcode Project

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）。

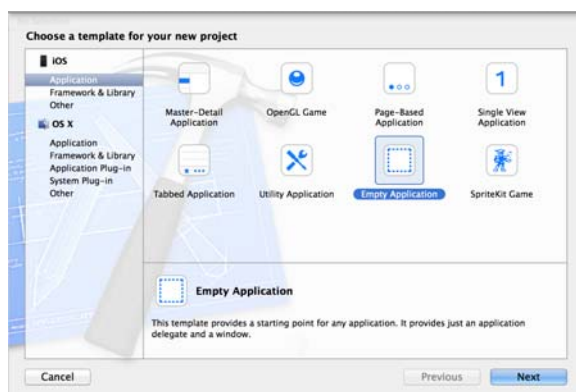


图 1-2 创建新项目

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。

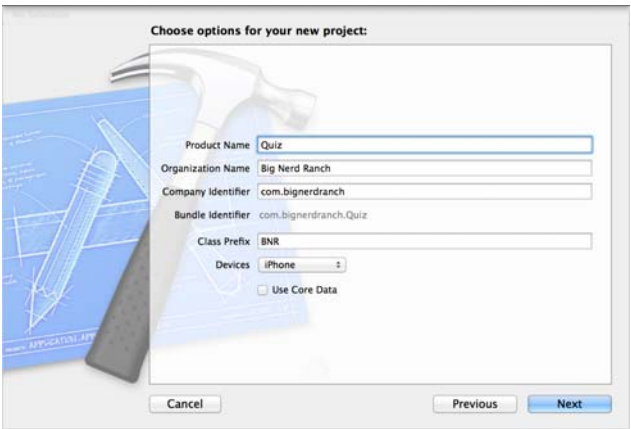


图 1-3 设置新项目

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）。

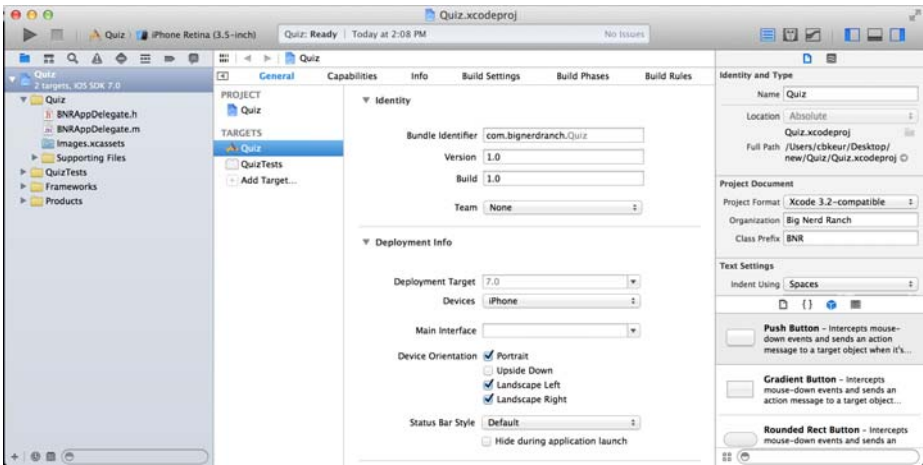


图 1-4 Xcode 工作空间窗口

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

### Model-View-Controller

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。

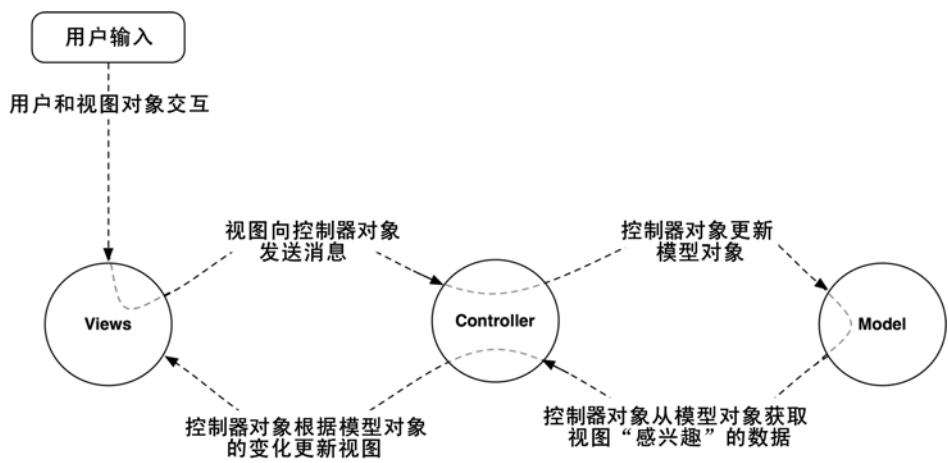


图 1-6 MVC 设计模式

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

### Designing 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。

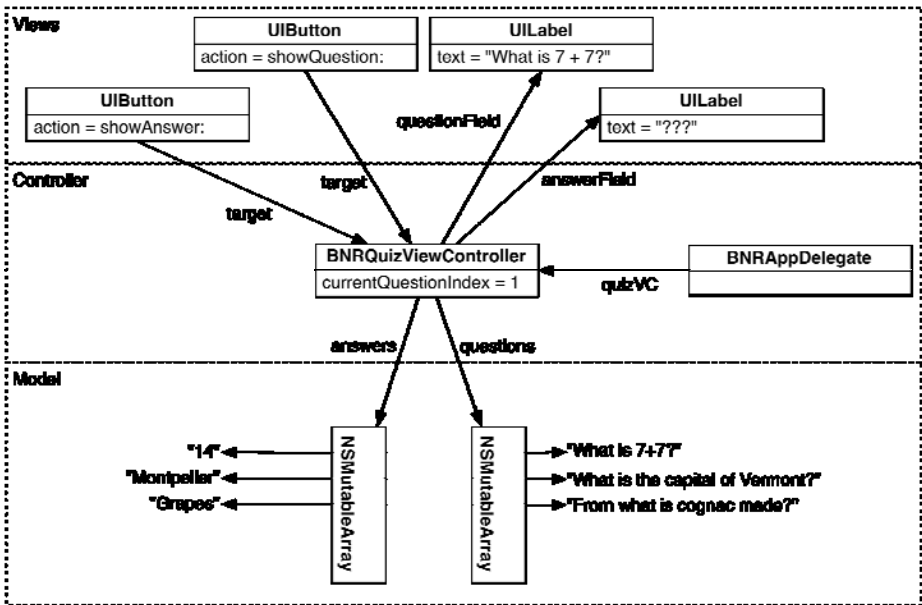


图 1-7 Quiz 应用的对象图

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

### Creating a View Controller

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）。

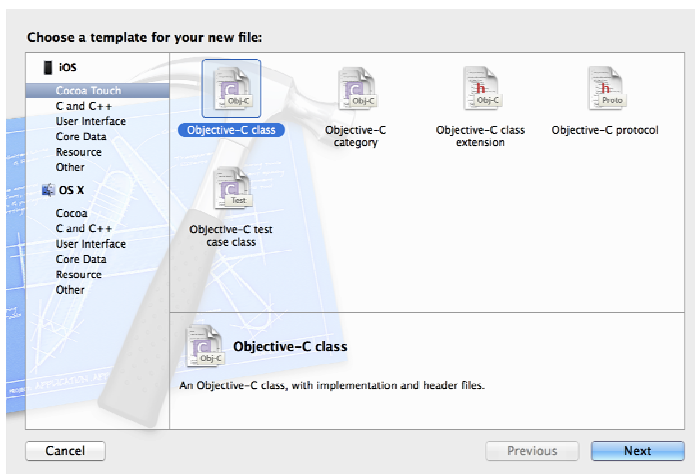


图 1-8 创建 Objective-C 类

在新出现的窗口中，将 **BNRQuizViewController** 填入 **Class** 文本框，然后单击 **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）。

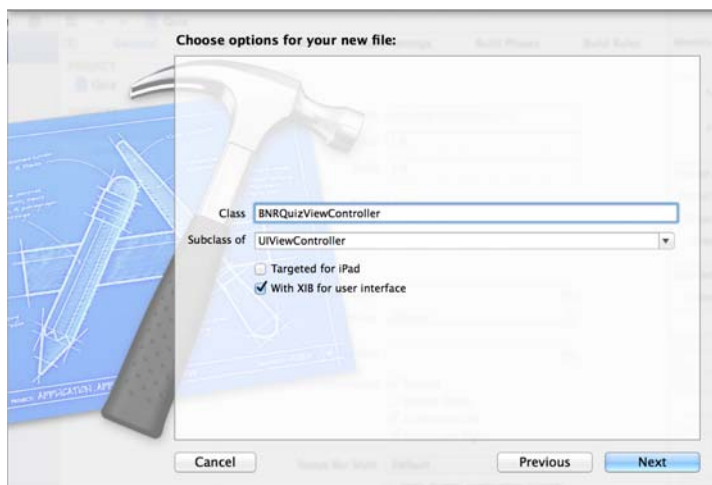


图 1-9 创建视图控制器

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。

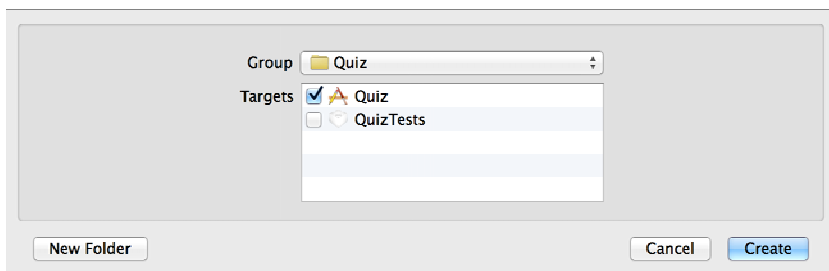


图 1-10 选中 Targets 中的 Quiz

## 1.5 创建界面

### Building an Interface

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：**BNRQuizViewController.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）。

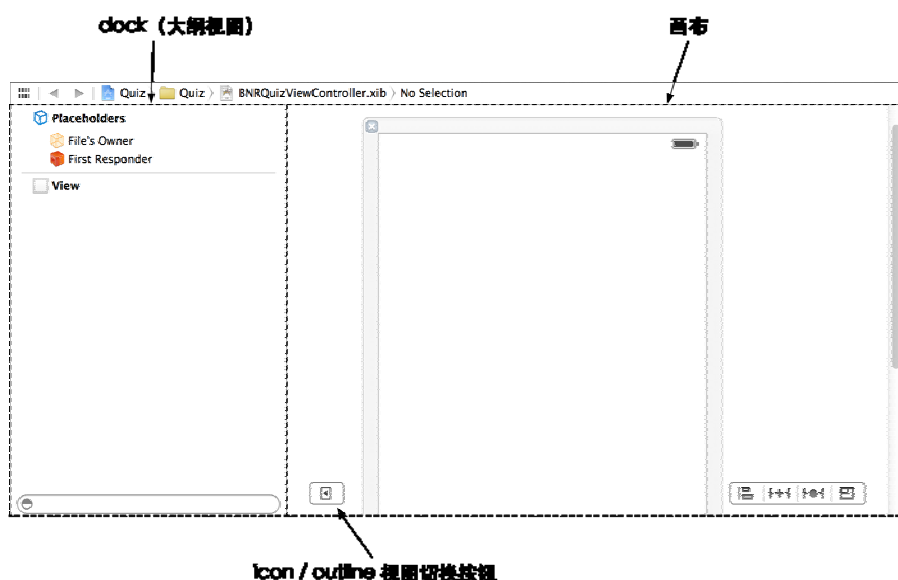


图 1-11 使用 Interface Builder 编辑 XIB 文件

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）。

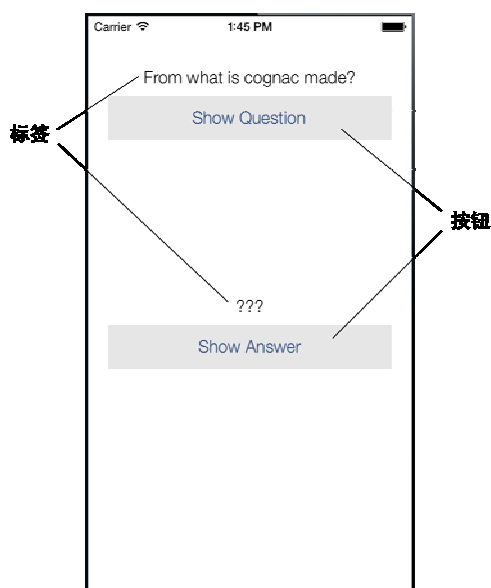



图 1-12 需要的标签和按钮

## 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**。

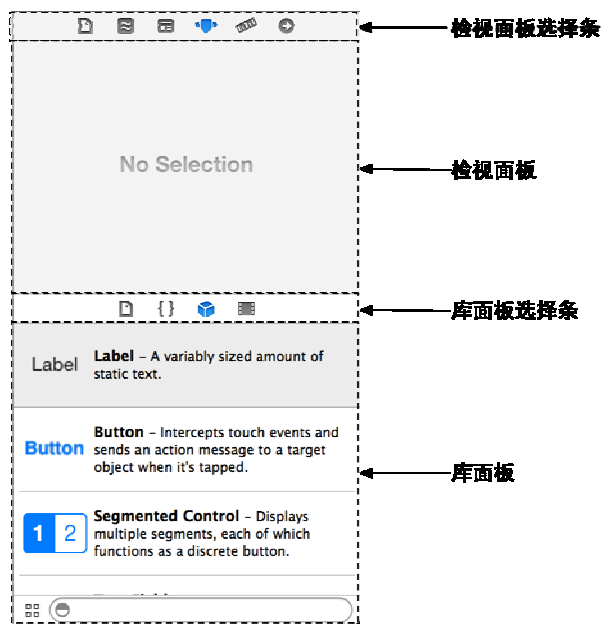


图 1-13 Xcode 工具区域

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）。

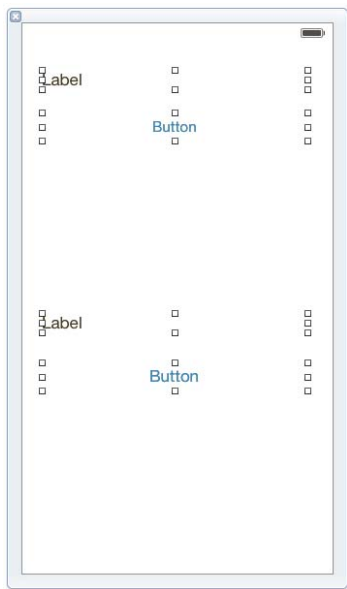


图 1-14 修改标签和按钮的大小


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。

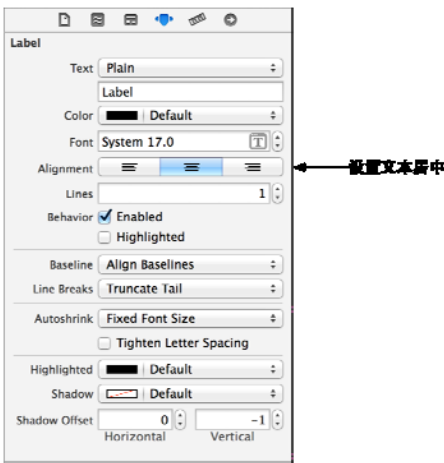


图 1-16 设置 label 的文本对齐方式为居中对齐

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

---

### Making Connections

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。

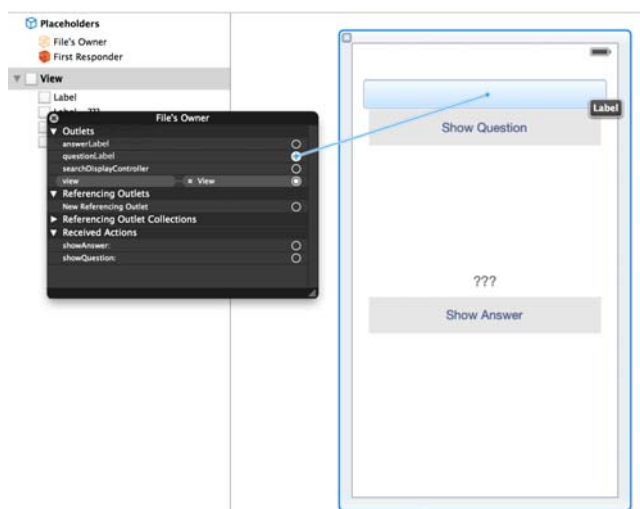


图 1-17 设置 questionLabel

（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）。

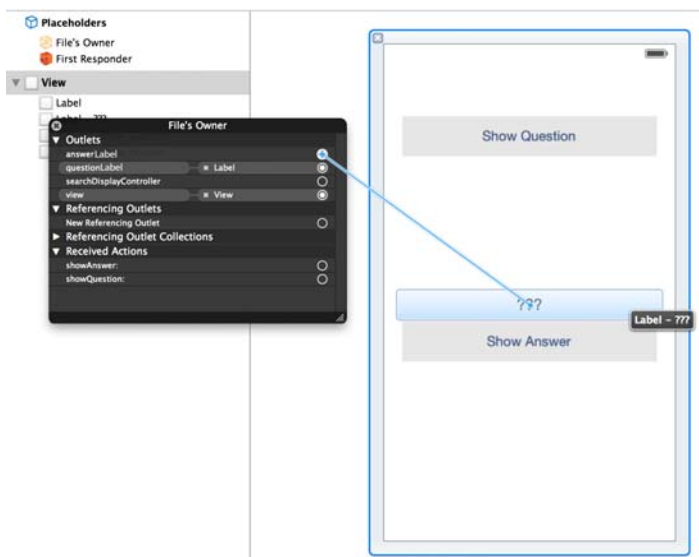


图 1-18 设置 `answerLabel`

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。

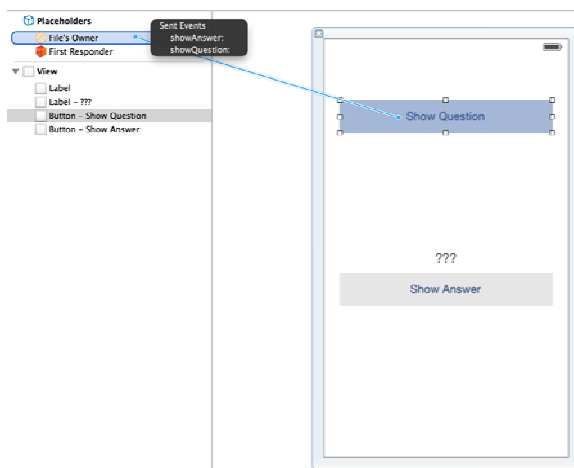



图 1-19 设置 Show Question 按钮的目标/动作

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）。

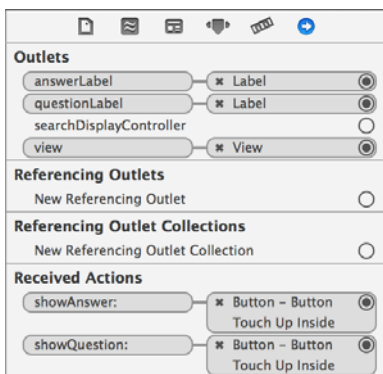


图 1-20 通过检视面板查看关联

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

### Creating Model Objects

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 `nibNameOrNil` 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

### Pulling it all Together

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（`AppDelegate`）。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:didFinishLaunchingWithOptions:` 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

### Running on the Simulator

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）。

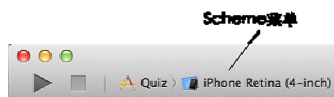



图 1-22 选择 iPhone Retina(4-inch)

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）。

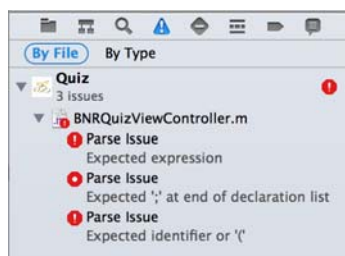


图 1-23 列出错误和警告信息的问题导航面板

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

### Deploying an Application

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。下面将该应用装入设备。

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](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 <b>Keychain Access</b> (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	<p>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:<b>com.bignerdranch.AwesomeApp</b>,其中应用名称跟在公司名称后面</p> <p>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 <b>General</b> 面板</p>
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 <b>.mobileprovision</b>

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

## Application Icons

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)。

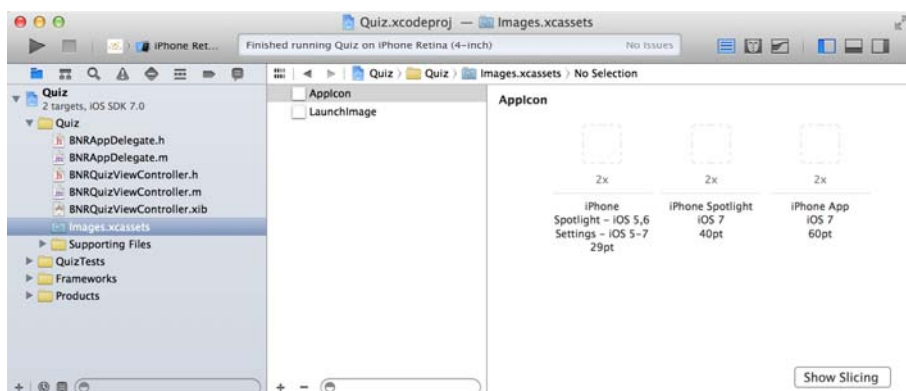


图 1-25 打开资源目录

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**。

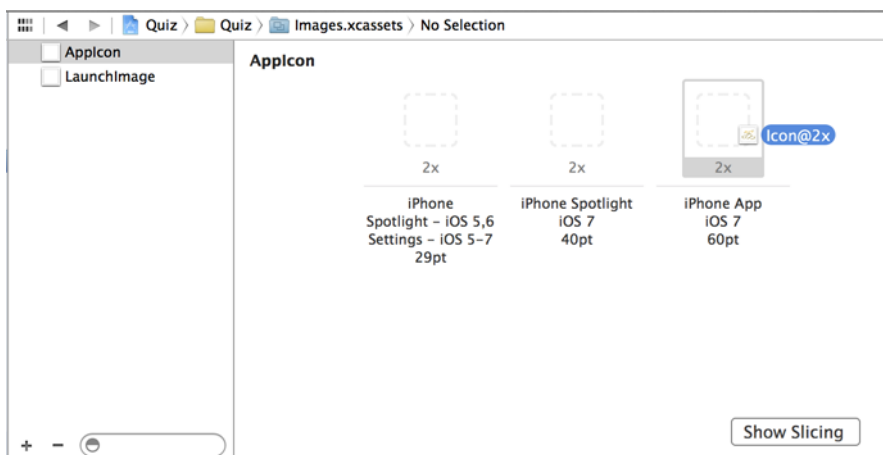


图 1-26 在资源目录中添加应用图标

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

### Launch Images

**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

## 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

### Objects

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划线。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）。

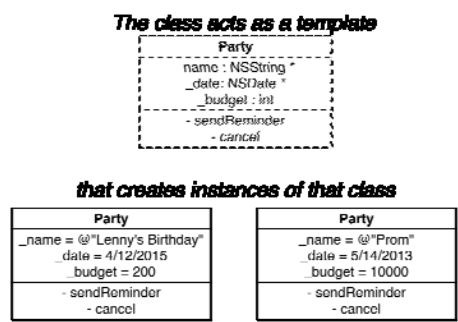


图 2-1 Party 类及其对象

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

### Using Objects

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划线，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 *partyInstance = [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

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）。

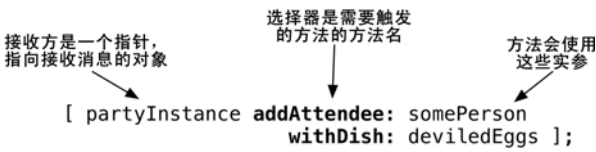


图 2-2 消息发送代码的各个组成部分

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：

```
// venue 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 RandomItems

### Beginning RandomItems

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。

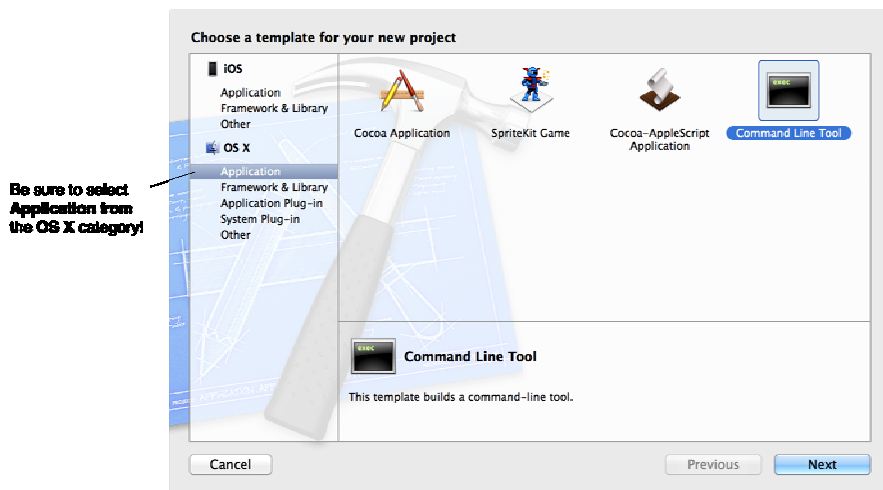


图 2-3 创建一个命令行工具项目

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）。

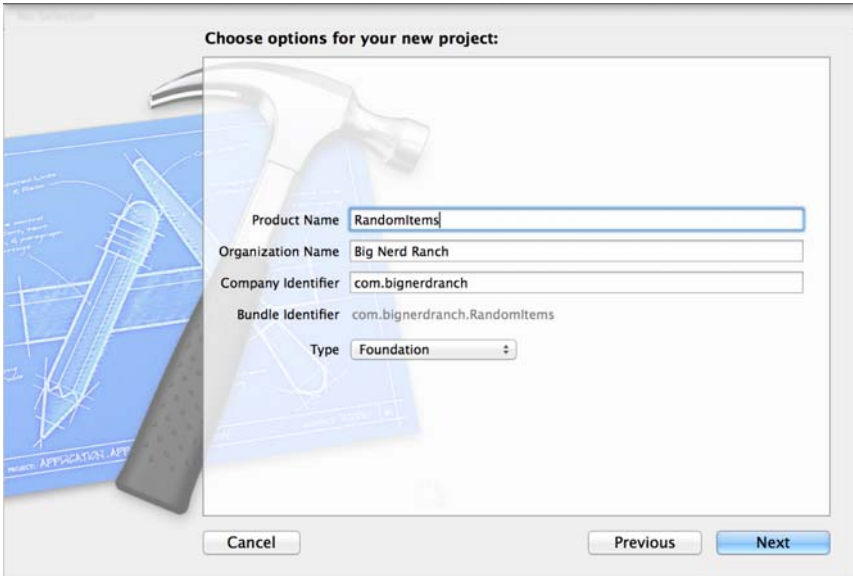


图 2-4 为项目命名

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。

**RandomItems** 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。

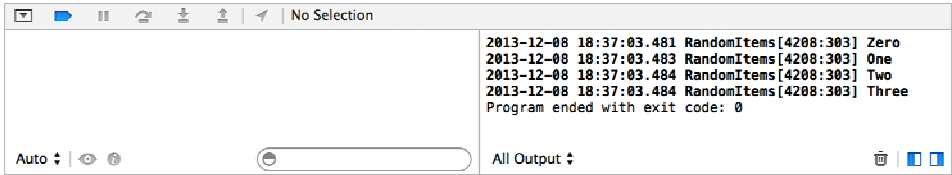


图 2-5 控制台的输出

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

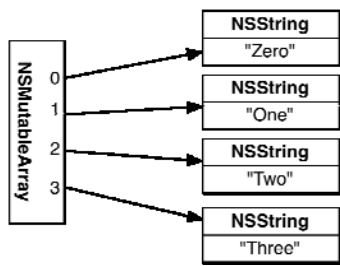


图 2-6 NSMutableArray 对象包含指向 NSString 对象的指针

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。

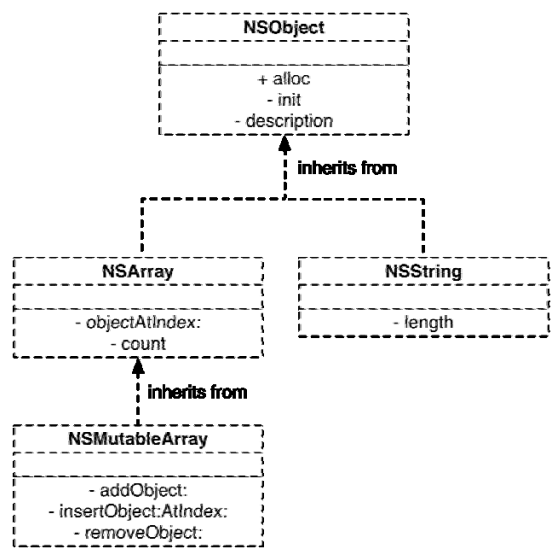


图 2-7 部分类的层级结构

**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 <Foundation/Foundation.h>

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;
    }
    return 0;
}
```

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）。

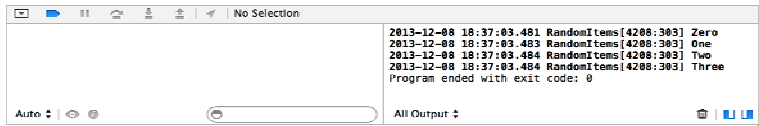


图 2-8 调试区域和控制台

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

### Subclassing an Objective-C Class

本节要创建一个名为 `BNRItem` 的 `NSObject` 子类（见图 2-9）。

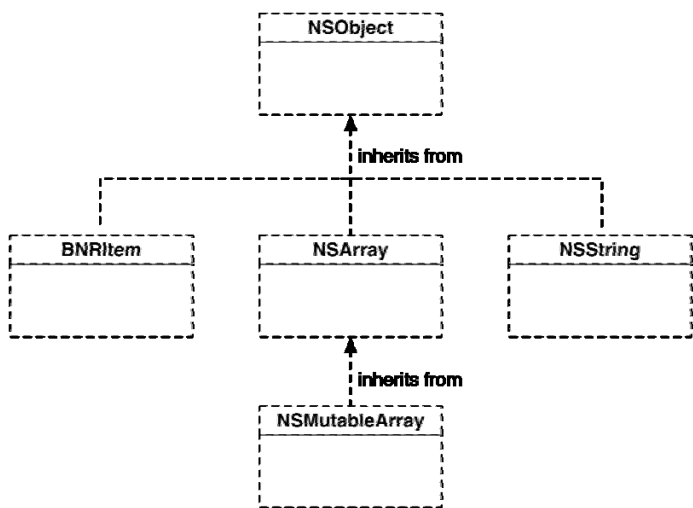


图 2-9 BNRItem 类的层级结构

**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）。

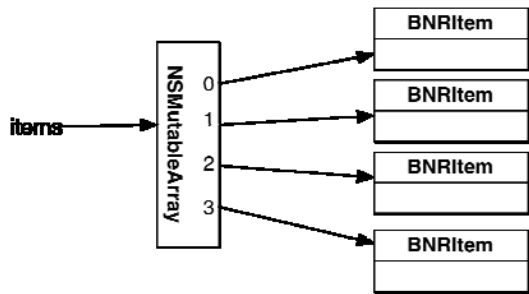


图 2-10 使用 **BNRItem** 代替 **NSString**

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）。

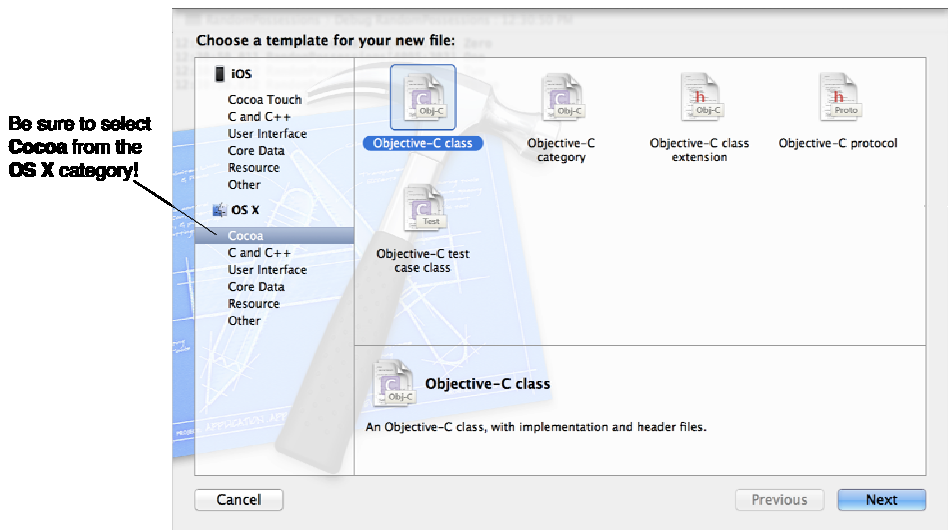


图 2-11 创建新类

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）。

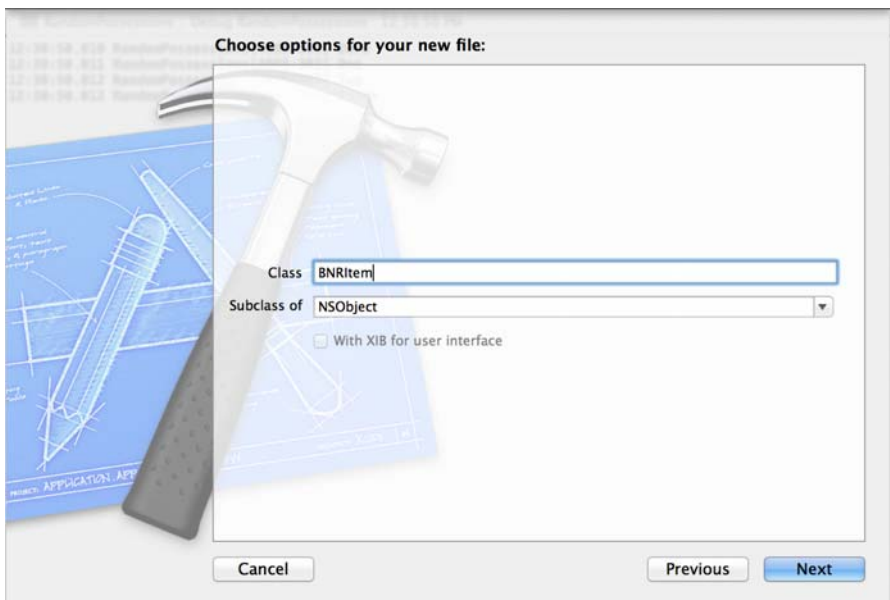


图 2-12 选择父类

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。

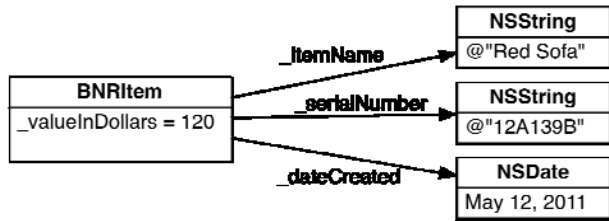


图 2-13 一个 **BNRItem** 对象

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 BNRIItem : 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 **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 **BNRIItem** 的实现文件, **BNRIItem.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 **BNRIItem.m** 中, 删除 **@implementation** 和 **@end** 之间由项目模板加入的代码, 然后为 **BNRIItem.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，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 setName:@"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划线, 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 setName:@"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** 的关键词：

```
- (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。

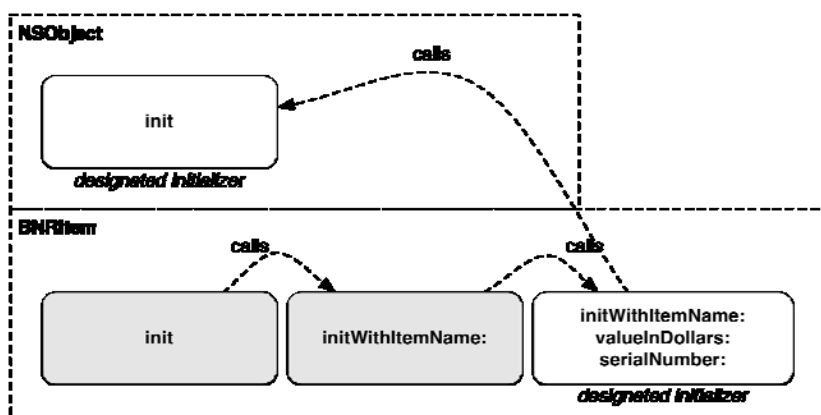


图 2-17 初始化方法链

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:@"%s@ %s",
                            [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：

```
BNRToxicWasteItem *item = [BNRToxicWasteItem randomItem];
```

## 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 (8C5V6): 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 (3E400): 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

### More on NSArray and 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**、**NSNumber** 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:@"%s %s",
                           [randomAdjectiveList objectAtIndex:adjectiveIndex],
                           [randomNounList objectAtIndex:nounIndex]];

    NSString *randomName = [NSString stringWithFormat:@"%s %s",
                           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 `replaceObjectAtIndex:withObject:` 消息。

## 2.6 异常与未知选择器

### Exceptions and Unrecognized Selectors

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 `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。

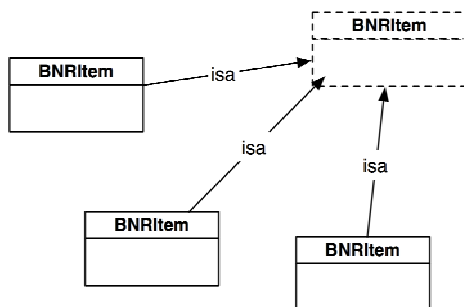


图 2-20 `isa` 实例变量

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，并列出了相应的原因。

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

### Challenges

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

### Bronze Challenge: Bug Finding

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

### Silver Challenge: Another\_INITIALIZER

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

### Gold Challenge: Another Class

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

### Are You More Curious?

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

### For the More Curious: Class Names

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。为了能更好地地区分，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

### For the More Curious: #import and @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

## Managing Memory with ARC

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

#### The Stack

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。

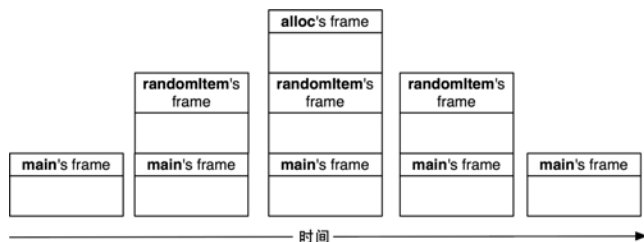


图 3-1 调用 **RandomItems** 的 **main** 函数时栈的变化过程

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

### The Heap

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将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

### Pointer Variables and Object Ownership

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。

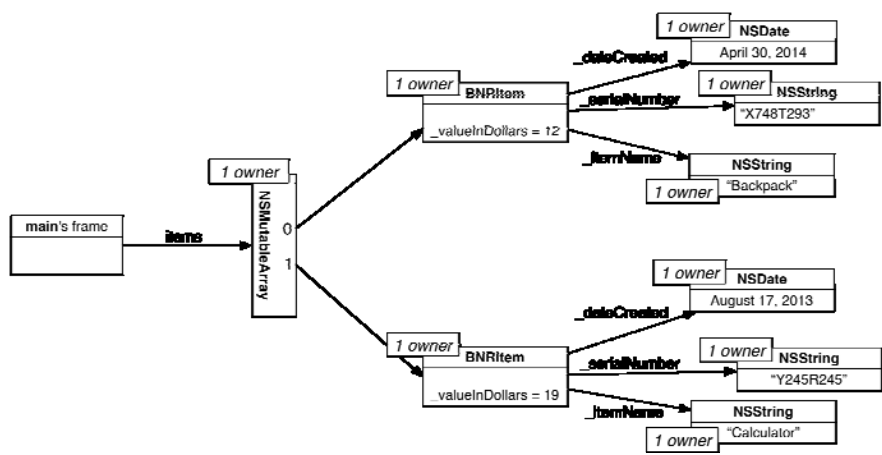


图 3-2 RandomItems 对象图（图中有两个 BNRItem 对象）

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）。

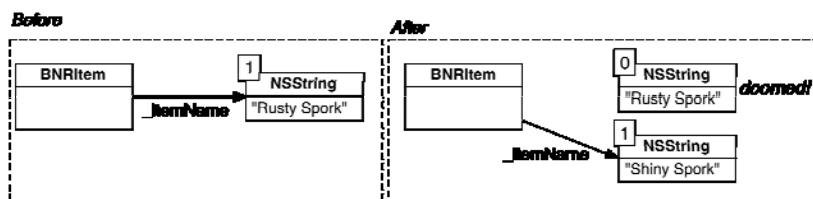


图 3-3 修改指针变量所指向的对象

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”` 已经没有任何其他拥有者了，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。

`RandomItems` 中就有个现成的例子，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，输出上述释放在过程。`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

### Strong and Weak References

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。

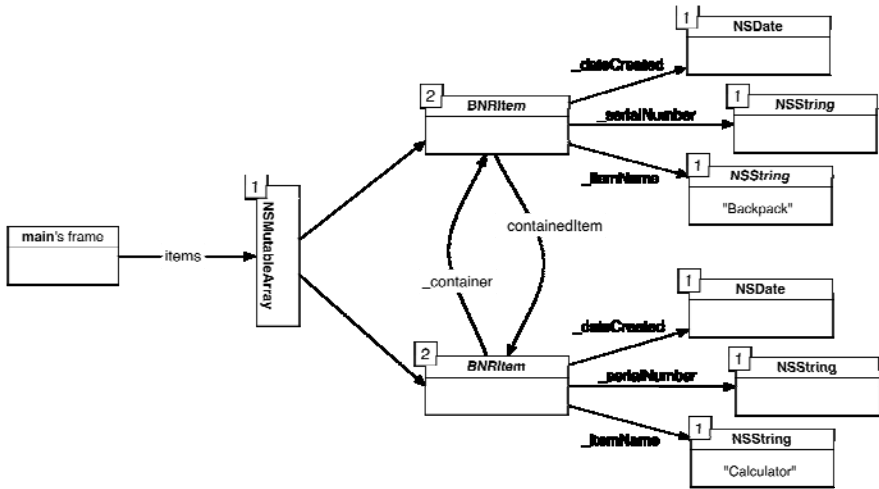


图 3-4 存在强引用循环问题的 RandomItems

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。

RandomItems 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。

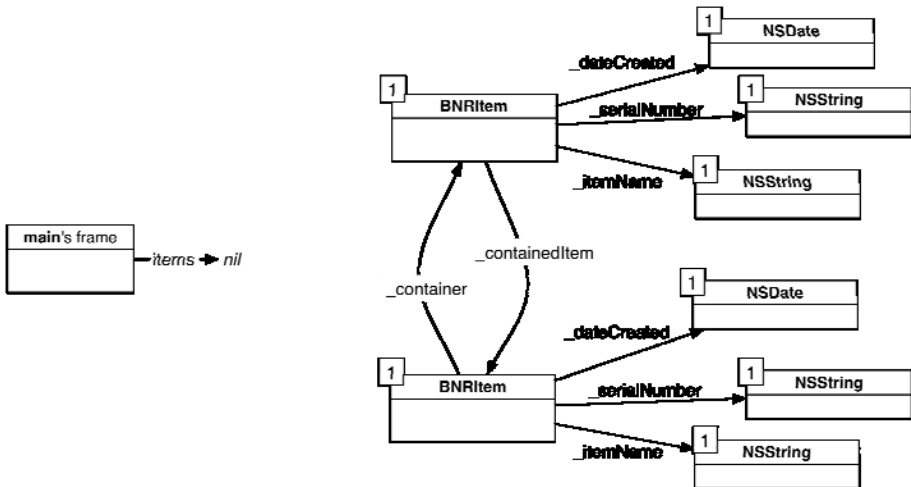


图 3-5 对象没有被释放，造成内存泄露

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。

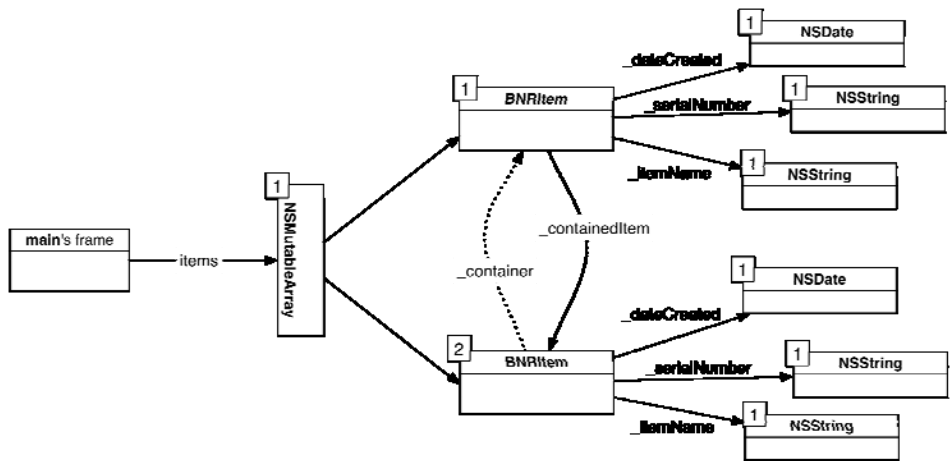


图 3-6 解决强引用循环问题后的 RandomItems

### 3.5 属性

#### Properties

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	<pre>@interface BNRThing : NSObject {     NSString *_name; } - (void)setName:(NSString *)n; - (NSString *)name; @end</pre>	<pre>@interface BNRThing : NSObject @property NSString *name; @end</pre>
BNRThing.m	<pre>@implementation BNRThing - (void)setName:(NSString *)n {     _name = n; }  - (NSString *)name {     return _name; }  @end</pre>	<pre>@implementation BNRThing @end</pre>

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划线，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划线。

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) BNRIItem *containedItem;
@property (nonatomic, weak) BNRIItem *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 `BNRIItem` 为例, 假设有某个初始化后的 `BNRIItem` 对象, 其实例变量 `_itemName` 指向一个 `NSMutableString` 对象, 代码如下:

```

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

### For the More Curious: Property Synthesis

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

### For the More Curious: Autorelease Pool and ARC History

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。