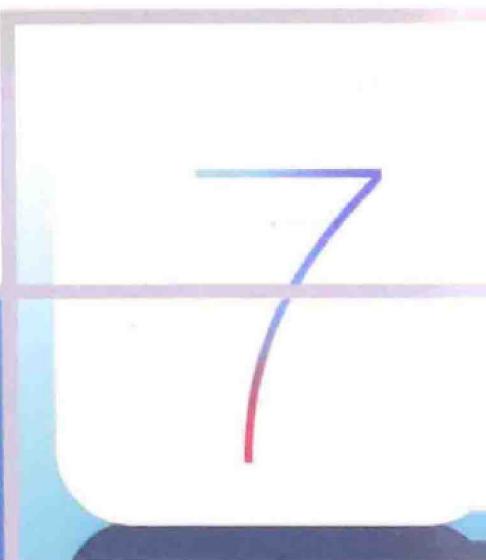
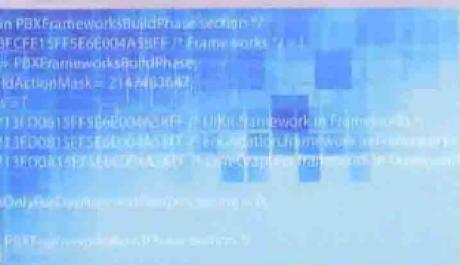




[美] Rob Napier Mugunth Kumar 著
美团移动 译

iOS编程实战

- iOS开发进阶首选
- 深入挖掘iOS高级特性与开发技巧
- 挑战编程极限，打造非凡应用



人民邮电出版社
POSTS & TELECOM PRESS



iOS 7 Programming: Pushing the Limits

“翻看本书目录就知道，这是奋战在项目一线的开发人员急需的一本书。”

——T. Casto

“iOS开发过程中最容易出问题的Zombies、GCD、KVO等主题在本书中都有介绍，对于具备一定基础而又想迫切提高开发水平的程序员来说，这是一本相当给力的书。”

——Jaime Moreno

“看过这本书的上一版，因此这一版出来后我毫不犹豫就将它放进了购物车。本书是iOS开发进阶的基石。我将本书介绍给了公司内一些年轻的iOS开发者，他们读完成后给出一致好评。”

——京东读者对上一版的评价

本书深入介绍iOS 7新特性和新功能，涵盖iOS 7大部分新增特性，包括新的后台操作、Core Bluetooth、UIKit 动力学以及TextKit。另外还介绍了如何处理新的扁平化UI，并新增了一章你可能不知道的“小技巧”。如果读者熟练掌握C和C++，读完本书即可创建性能优异的iPhone、iPad和iPod touch应用。

本书主要内容包括：

- ▶ iOS 7新特性和新功能概览；
- ▶ 深入解析多任务、多平台、安全服务、应用内购买、自动布局等高级主题；
- ▶ 全面介绍REST、高级GCD、本地化和国际化、Core Bluetooth；
- ▶ 细致讲解UIKit动力学、自定义过渡及其他内容。

本书助你充分利用iOS 7新特性，挑战编程极限，打造非凡应用。



WILEY

Copies of this book sold without a Wiley sticker
on the cover are unauthorized and illegal.

图灵社区：iTuring.cn

热线：(010)51095186转600

分类建议 计算机/移动开发/iOS

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-36803-4



ISBN 978-7-115-36803-4

定价：79.00元

iOS编程实战

[美] Rob Napier Mugunth Kumar 著
美团移动 译



人民邮电出版社
北京

图书在版编目（C I P）数据

iOS编程实战 / (美) 纳皮尔 (Napier, R.) , (美)
库玛 (Kumar, M.) 著 ; 美团移动译. -- 北京 : 人民邮
电出版社, 2014. 9
(图灵程序设计丛书)
ISBN 978-7-115-36803-4

I. ①i… II. ①纳… ②库… ③美… III. ①移动终
端—应用程序—程序设计 IV. ①TN929. 53

中国版本图书馆CIP数据核字(2014)第191743号

内 容 提 要

本书是最受开发者喜爱的 iOS 进阶图书。它包含大量代码示例，主线是围绕如何设计、编写和维护优秀的 iOS 应用。开发者可从本书学到大量关于设计模式、编写可重用代码以及语法与新框架的知识。

相对上一版，新版进行了大幅修订，新增 6 章阐述 iOS 7 新特性，并对大部分内容进行了更新，涵盖了 iOS 7 大部分新增特性，包括新的后台操作（第 11 章）、Core Bluetooth（第 13 章）、UIKit 动力学（第 19 章）以及 TextKit（第 21 章）。我们提供了如何处理新的扁平化 UI 的指南（第 2 章），还新增了一章开发者不太常见但相当实用的“小技巧”（第 3 章）。

本书适合 iOS 移动开发人员。

-
- ◆ 著 [美] Rob Napier Mugunth Kumar
 - 译 美团移动
 - 责任编辑 朱 巍
 - 责任印制 焦志炜
 - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
 - 邮编 100164 电子邮件 315@ptpress.com.cn
 - 网址 <http://www.ptpress.com.cn>
 - 三河市海波印务有限公司印刷
 - ◆ 开本：800×1000 1/16
 - 印张：25.5
 - 字数：603千字 2014年9月第1版
 - 印数：1-3 500册 2014年9月河北第1次印刷
 - 著作权合同登记号 图字：01-2014-1133号
-

定价：79.00元

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

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

前　　言

从某种程度上说，iOS 7 是从 SDK 随着 iPhone OS 2 发布到现在 iOS 发生过的最大的变化。人们在新闻和博客中讨论新的扁平化用户界面的各个方面，及其对应用开发者和用户的意义。可以说，从没有一次 iOS 的升级会使得这么多的开发者重新设计 UI。

但是从另外的角度看，iOS 7 几乎可以从 iOS 6 无缝升级。比起 iOS 4 在多任务上的变化，iOS 7 只需要对应用做很小的改动，尤其是开发者使用标准 UI 或者完全自定义 UI 的情况。对于这两种极端情况，UI 的变化要么是自动完成的，要么压根儿跟开发者没关系。

不过，对所有的开发者来说，iOS 还是带来了变化。有很多管理后台操作的方法，但是后台运行的规则甚至比以前更严格了。UIKit 动力学意味着更灵活的动画，不过实现起来不简单。TextKit 为文本布局带来了令人难以置信的特性，也伴随着令人发疯的限制和 bug。iOS 7 是大杂烩，既有美好也有挫败。不过你得学习 iOS 7，因为用户很快就会升级。

如果你准备好了去探索最新的苹果系统，准备好了挑战应用的极限，那么本书会助你一臂之力。

读者对象

这并不是一本入门书。其他一些书会教你 Objective-C 并一步步指导你学习 Interface Builder。不过本书假定你已经拥有一些 iOS 开发经验。可能是自学的，或者上过培训班，没准已经有一个应用即将完工只是没有上架而已。对于此类读者，如果你打算学习更深入的内容、最佳实践以及作者源自真实工程的开发经验，那你就找对书了。

这本书并不是示例的简单堆砌，它包含大量代码，不过主线还是围绕如何设计、编写和维护优秀的 iOS 应用。本书不仅会教你怎么做，并且会剖析这样做的原因。你会学到很多关于设计模式、编写可重用代码以及语法与新框架的知识。

本书内容

iOS 平台总是向前发展，本书也一样。书中大部分示例需要至少 iOS 6 才能运行，有些需要 iOS 7。所有示例都启用了自动引用计数（ARC）、自动属性合成和对象字面量。除了很少几处外，本书不会讨论向后兼容。如果你的代码过于庞大，必须要向后兼容，你可能知道如何处理。本书主旨是通过最好的特性来创造最佳应用。

本书专注于 iPhone 5、iPad 3 和更新的型号。大部分主题对其他 iOS 设备也适用。第 15 章讲了如何处理平台间的差异。

新版内容

本版涵盖了 iOS 7 大部分的新增特性，包括新的后台操作（第 11 章）、Core Bluetooth（第 13 章）、UIKit 动力学（第 19 章）以及 TextKit（第 21 章）。我们提供了如何处理新的扁平化 UI 的指南（第 2 章），还新增了一章你可能不知道的“小技巧”（第 3 章）。

本书专注于 iOS 7 中最有价值的信息。前几版的有些章节被移到了网站上 (iosptl.com)。读者可以在那里找到关于常见的 Objective-C 实践、定位服务、错误处理等内容的章节。

本书结构

iOS 提供了非常丰富的工具，既有 UIKit 这样的高层框架，也有 Core Text 这样的低层工具。有时候，同一个目标可以通过多种方式来达成。作为开发人员，如何找到最合适工具呢？

本书既考虑了日常开发需求，也考虑了特定的用途，能够帮你作出正确的选择。学完本书，你会明白每个框架存在的价值、框架之间的相互关系，以及什么时候选用哪一个框架。最终，你会知道哪个框架最适合解决哪一类问题。

本书分四部分，从最常用的工具一直讲到最强大的工具。这一版新增的章会在前面用“新增”字样标识，而经过大范围更新的章会用“更新”字样标识。

第一部分：全新功能

如果你熟悉 iOS 6，这部分可以让你快速了解 iOS 7 的新特性。

- (新增) 第 1 章 “全新的系统” —— iOS 7 增加了大量新特性，本章将带你快速概览有哪些内容。
- (新增) 第 2 章 “世界是平的：新的 UI 范式” —— iOS 7 对 iOS 应用的外观和行为做了巨大的改变。本章将介绍迁移所需的新模式和设计语言。

第二部分：充分利用日常工具

作为一名 iOS 开发人员，你应该掌握很多常用工具，比如通知、表视图和动画图层。不过要想发挥它们的全部潜力，就要熟悉它们。在这一部分，我们将学到 Cocoa 开发的最佳实践。

- (新增) 第 3 章 “你可能不知道的” —— 即使你是一位有经验的开发者，你可能并不熟悉 Cocoa 的一些小特性和技巧。本章介绍作者根据多年 iOS 开发经验总结的最佳实践，以及 Cocoa 一些不那么常见的方面。
- (更新) 第 4 章 “故事板及自定义切换效果” —— 故事板仍然会使一些熟悉 nib 文件的开发人员感到费解。你在这里将会学到如何使用故事板来提升应用。
- (更新) 第 5 章 “掌握集合视图” —— 集合视图正在逐步替代表视图，成为开发人员偏爱的布局控件。即使对于表格类布局，集合视图也提供了极大的灵活性，要想开发出迷人的应用，你应该理解这一点。本章会教你如何掌握这一重要工具。
- (新增) 第 6 章 “使用自动布局” —— 如果 WWDC 2013 有什么核心的信息，那么一定是：使用自动布局。会议期间几乎所有的 UIKit 会话都在反复强调这一点。由于 Xcode 4 中诸多 Interface

Builder 问题，开发者可能尽量避免使用自动布局。Xcode 5 极大地改进了对自动布局的支持。不管是喜欢使用约束，还是渴望回归 springs & struts，你都不该错过最新的自动布局。

- 第 7 章“更完善的自定义绘图”——很多新开发者都对自定义绘图退避三舍，但它却是快速创建美观用户界面的关键。这一章将探究 UIKit 和 Core Graphics 中有关绘图的功能，告诉大家怎么才能做到既快又美。
- 第 8 章“Core Animation”——iOS 设备对动画的支持是无与伦比的。借助强大的 GPU 和高度优化的 Core Animation，你可以创建直观又吸引人的界面。在这一章中，我们会介绍一些基础知识以及动画的原理。
- (更新) 第 9 章“多任务”——多任务是许多应用程序的重要部分，这一章将介绍如何同时使用操作和 GCD 执行多任务。

第三部分：选择工具

- 第 10 章“创建 (Core) Foundation 框架”——说到 iOS 中最强大的框架，你能想到的 Core 框架可能会有 Core Graphics、Core Animation、Core Text，但它们都是基于 Core Foundation 框架的。在这一章中，我们学习如何使用 Core Foundation 数据类型，以便充分利用 iOS 提供的功能。
- (更新) 第 11 章“幕后制作：后台处理”——iOS 7 的后台处理又灵活了很多，但是，要想充分利用这些新变化，你得遵循一些新规则。本章带你深入学习新的 NSURLSession，以及如何最好地实现状态恢复。
- 第 12 章“使用 REST 服务”——基于 REST 的服务是现代应用程序的核心，这一章将教会你在 iOS 中最好地实现它们。
- (新增) 第 13 章“充分利用蓝牙设备”——苹果一直在加强 iOS 与其他设备创建 ad hoc 网络的能力。这使得开发全新的应用成为可能：从更好的游戏到微定位服务，再到更方便的文件共享。加快创新，投入这个全新的市场吧！
- 第 14 章“通过安全服务巩固系统安全”——用户安全和保护隐私永远是第一位的。这一章会介绍如何通过钥匙串、证书和密码保护应用和用户数据不会被盗用。
- (更新) 第 15 章“在多个苹果平台和设备及 64 位体系结构上运行应用”——iOS 家族人丁兴旺，不仅有了 iPod touch、iPhone、iPad、Apple TV，而且新机型仍会不断涌现。目前还无法一次编写随处运行。为了保证应用在任何平台上都表现卓越，本章将讨论如何基于硬件和平台调整应用。
- 第 16 章“国际化和本地化”——虽然你现在可能只想关注某个国家的市场，但让应用明天能够顺利走向世界也只需做一点点工作。本章会告诉你如何不影响当前开发，又能减少未来的麻烦和成本。
- 第 17 章“调试”——要是每个应用第一次就能完美运行该有多好。幸好，Xcode 和 LLDB 提供了很多能帮助你抓住狡猾 bug 的工具。你会学到很多高级的内容，了解实际开发中如何处理错误。
- (更新) 第 18 章“性能调优”——高性能可以让应用脱颖而出。优化 CPU 和内存性能非常重要，不过你也需要优化电池以及网络使用。苹果公司提供了 Instruments 这个强大的工具来解决这些问题。你会学到如何使用 Instruments 来找到瓶颈，以及如何在找到问题后改善性能。

第四部分：超越极限

这一部分是全书最精彩的内容。你已经学到了基础知识，掌握了基本技能。现在该使用高级工具来超越极限了。这一部分将带你深入地了解 iOS。

- (新增) 第 19 章“近乎物理效果：UIKit 动力学”——苹果一直致力于让动态的动画界面更容易实现。UIKit 动力学是其最新杰作，为 UIKit 带来了“类物理效果”的引擎。这个工具很强大，同时要用好也很难。本章学习如何使用。
- (新增) 第 20 章“魔幻的自定义过渡”——WWDC 2013 最绚丽的演示程序就是关于自定义过渡效果的。忘了“推入”，来学习如何创建动态和交互式的过度效果吧。
- (更新) 第 21 章“精妙的文本布局”——iOS 7 以文本为中心的 UI 需要在字体处理和文本布局的细节上投入大量精力。TextKit 带来了很多新特性，从动态字体到排除路径，还带来了 bug 和令人抓狂的限制。无论怎么处理文本，首先你得掌握属性化字符串。本章介绍这些强大的数据结构的方方面面，以及如何用 TextKit 充分利用这些数据结构。
- 第 22 章“Cocoa 的大招：键值编码和观察”——苹果的许多强大框架都是依靠 KVO(Key-Value Observing) 来维护性能和灵活性的。你会学到如何利用灵活性和 KVO 的速度，以及让它如此透明的诀窍。
- (新增) 第 23 章“超越队列：GCD 高级功能”——分派队列是非常强大的工具，已经成了很多应用的重要组成部分。但是除了队列以外，GCD 还有别的东西。本章介绍信号量、分派组还有非常强大的分派数据和分配 IO 这些工具。
- 第 24 章“深度解析 Objective-C”——这一章致力于揭开 Objective-C 背后的秘密，包括如何使用 Objective-C 运行时直接动态地修改类和方法、如何通过 Objective-C 函数调用 C 方法，以及如何通过系统来扩展程序。

以上各章可以跳读，除了需要 Core Foundation 数据对象（特别是 Core Graphics、Core Animation）的几章，其他章都是相互独立的。关于 Core Foundation 的内容，最终会归总到第 10 章“创建 Core Foundation 框架”。

阅读条件

本书所有示例都是用 Mac OS X 10.8 上的 Xcode 5 以及 iOS 7 开发的。你需要一个苹果开发人员账户来访问大部分工具和文档，并且需要一个开发人员许可证来运行 iOS 设备上的应用程序。对此，请参考<http://developer.apple.com/programs/ios>并注册账号。

本书中大部分示例可以在 Xcode 5 的 iOS 模拟器中运行。使用 iOS 模拟器就不需要苹果开发人员许可证了。

苹果文档

苹果公司在自己的网站上和 Xcode 中提供了大量文档。这些文档的 URL 地址变动很频繁而且非常长。本书会使用标题而不是 URL 来引用这些文档。如果想在 Xcode 中寻找文档，请按下 Cmd-Option-? 快捷键或点击 Help→Documentation and API Reference。并在 Documentation Organizer 窗口中点击搜索

图标，输入文档的标题，并从搜索结果中选择文档。可以参考图 0-1 中搜索 Coding Guidelines for Cocoa 的示例。

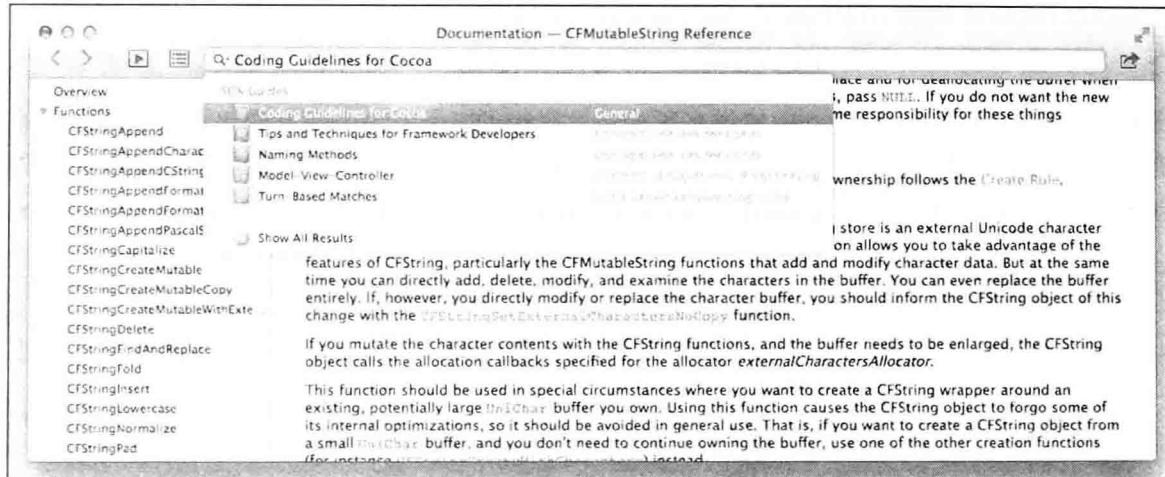


图 0-1 搜索 Coding Guidelines for Cocoa

如果想在苹果官方网站查找文档，可以访问 <http://developer.apple.com>，点击 Member Center，并登录。选择 iOS Dev Center，并在搜索框中输入文档的标题。

在线文档与 Xcode 文档是相同的。你可能会接收到 iOS 和 Mac 两个平台的结果，请阅读 iOS 版。很多 iOS 文档是 Mac 版的副本，偶尔会包含 iOS 不支持的函数调用或常量。本书会告诉你哪些功能在 iOS 上能用。

源代码

在学习本书示例的时候，可以手工输入代码，也可以使用本书附带的源代码文件。本书所有的源代码可以在 <https://github.com/iosptl/ios7ptl> 或 www.wiley.com/go/ptl/ios7programming 上下载得到^①。举个例子，下载之后，在第 21 章文件夹 SimpleLayout 工程的 CoreTextLabel.m 文件中可以看到如下代码：

CoreTextLabel.m (SimpleLayout)

```
- (id)initWithFrame:(CGRect)frame {
    if ((self = [super initWithFrame:frame])) {
        CGAffineTransform transform = CGAffineTransformMakeScale(1, -1);
        CGAffineTransformTranslate(transform,
                                0, -self.bounds.size.height);
        self.transform = transform;
        self.backgroundColor = [UIColor whiteColor];
    }
    return self;
}
```

^① 也可以访问图灵社区本书页面 (<http://www.ituring.com.cn/book/1340>) 下载。——编者注

本书中有些代码片段并不完整，其目的只是为了辅助上下文说明问题。要想查看完整代码，可在本书同步网站下载（www.wiley.com/go/ptl/ios7programming）。

勘误

虽然我们已经尽了最大努力，但错误在所难免。有些可能是因为内容有变化，有些可能是输入错误，有些可能是我们理解上有偏差。为了确保代码与时俱进，请参考<https://github.com/iosptl/ios7ptl>上的最新版本，以及博客中的相关文章。任何问题都可以发送给 robnapier@gmail.com 或 contact@mk.sg。

目 录

第一部分 全新功能

第 1 章 全新的系统	2
1.1 新的 UI	2
1.2 UIKit Dynamics 和 Motion Effects	3
1.3 自定义过渡效果	4
1.4 新的多任务模式	4
1.5 Text Kit	5
1.6 动态字体	5
1.7 MapKit 集成	5
1.8 SpriteKit	5
1.9 LLVM 5	5
1.10 Xcode 5	7
1.10.1 nib 文件格式的变化	7
1.10.2 源代码控制集成	7
1.10.3 自动配置	7
1.10.4 对调试导航面板的改进	8
1.10.5 文档浏览器	8
1.10.6 Asset Catalog	9
1.10.7 测试导航面板	9
1.10.8 持续集成	10
1.10.9 Auto Layout 改进	10
1.10.10 iOS 模拟器	10
1.11 其他	10
1.12 小结	11
1.13 扩展阅读	11
第 2 章 世界是平的：新的 UI 范式	12
2.1 清晰、依从和层次	12
2.2 动画、动画、动画	12
2.2.1 UIKit Dynamics	13

2.2.2 UIMotionEffect	13
2.3 着色	14
2.4 用半透明实现层次和上下文	14
2.5 动态字体	15
2.6 自定义过渡效果	16
2.7 把应用过渡（迁移）到 iOS 7	17
2.7.1 UIKit 变化	17
2.7.2 自定义设计	17
2.7.3 支持 iOS 6	17
2.8 小结	20
2.9 扩展阅读	20

第二部分 充分利用日常工具

第 3 章 你可能不知道的	22
3.1 命名最佳实践	22
3.1.1 自动变量	22
3.1.2 方法	22
3.2 属性和实例变量最佳实践	23
3.3 分类	24
3.4 关联引用	26
3.5 弱引用容器	27
3.6 NSCache	28
3.7 NSURLComponents	28
3.8 CFStringTransform	29
3.9instancetype	30
3.10 Base64 和百分号编码	31
3.11 -[NSArray firstObject]	31
3.12 小结	31
3.13 扩展阅读	32

第 4 章 故事板及自定义切换效果	33	6.5 扩展阅读	69
4.1 初识故事板	33		
4.1.1 实例化故事板	34		
4.1.2 加载故事板中的视图控制器	34		
4.1.3 联线	34		
4.1.4 使用故事板来实现表视图	36		
4.2 自定义切换效果	37		
4.2.1 优点	39		
4.2.2 白璧微瑕——合并冲突	39		
4.3 小结	39		
4.4 扩展阅读	39		
第 5 章 掌握集合视图	41		
5.1 集合视图	41		
5.1.1 类与协议	41		
5.1.2 示例	42		
5.2 用集合视图自定义布局实现高级定制	49		
5.2.1 石工布局	50		
5.2.2 封面浏览布局	55		
5.3 小结	56		
5.4 扩展阅读	56		
第 6 章 使用自动布局	58		
6.1 Xcode 4 的自动布局	58		
6.2 了解自动布局	59		
6.3 Xcode 5 中自动布局的新特性	59		
6.3.1 在 Xcode 5 中使用自动布局	61		
6.3.2 固有尺寸	62		
6.3.3 固有尺寸和本地化	63		
6.3.4 设计时和运行时布局	63		
6.3.5 自动更新边框	64		
6.3.6 顶部和底部布局引导	65		
6.3.7 辅助编辑器中的布局预览	65		
6.3.8 在设计时调试自动布局	65		
6.3.9 在自动布局中使用滚动视图	66		
6.3.10 使用自动布局和边框	66		
6.3.11 可视格式化语言	66		
6.3.12 可视格式化语言的缺点	67		
6.3.13 调试布局错误	68		
6.4 小结	69		
第 7 章 更完善的自定义绘图	71		
7.1 iOS 的不同绘图系统	71		
7.2 UIKit 和视图绘图周期	72		
7.3 视图绘制与视图布局	73		
7.4 自定义视图绘制	74		
7.4.1 通过 UIKit 绘图	74		
7.4.2 路径	75		
7.4.3 理解坐标系	77		
7.4.4 重新调整大小以及内容模式	79		
7.4.5 变形	80		
7.4.6 通过 Core Graphics 进行绘制	82		
7.4.7 混用 UIKit 与 Core Graphics	85		
7.4.8 管理图形上下文	85		
7.5 优化 UIView 绘制	87		
7.5.1 避免绘图	87		
7.5.2 缓存与后台绘制	88		
7.5.3 自定义绘图与预渲染	88		
7.5.4 像素对齐与模糊文本	89		
7.5.5 透明、不透明与隐藏	90		
7.6 小结	90		
7.7 扩展阅读	91		
第 8 章 Core Animation	92		
8.1 视图动画	92		
8.2 管理用户交互	94		
8.3 图层绘制	94		
8.3.1 直接设置内容	96		
8.3.2 实现 display 方法	97		
8.3.3 自定义绘图	97		
8.3.4 在自己的上下文中绘图	99		
8.4 移动对象	99		
8.4.1 隐式动画	100		
8.4.2 显式动画	101		
8.4.3 模型与表示	101		
8.4.4 关于定时	103		
8.5 三维动画	105		
8.6 美化图层	108		
8.7 用动作实现自动动画	108		

8.8 为自定义属性添加动画.....	110	第 11 章 幕后制作：后台处理	140
8.9 Core Animation 与线程.....	111	11.1 后台运行最佳实践：能力越大责任 越大.....	140
8.10 小结	111	11.2 iOS 7 中后台运行的重要变化	142
8.11 扩展阅读.....	111	11.3 用 NSURLSession 访问网络	142
第 9 章 多任务	113	11.3.1 会话配置	143
9.1 多任务和运行循环简介.....	113	11.3.2 任务	143
9.2 以操作为中心的多任务开发	114	11.3.3 后台传输	144
9.3 用 GCD 实现多任务	118	11.4 周期性拉取和自适应多任务	146
9.3.1 分派队列简介	119	11.5 后台唤醒.....	146
9.3.2 用分派屏障创建同步点	120	11.6 状态恢复系统	147
9.3.3 分派组	121	11.6.1 测试状态恢复系统	147
9.4 小结	121	11.6.2 选择性加入	148
9.5 扩展阅读.....	121	11.6.3 应用启动过程的变化	149
第三部分 选择工具			
第 10 章 创建 (Core) Foundation 框架	124	11.6.4 状态恢复标识符	149
10.1 Core Foundation 类型	124	11.6.5 状态编码器与状态解码器.....	149
10.2 命名和内存管理	125	11.6.6 表视图和集合视图	153
10.3 分配器	126	11.7 小结	154
10.4 内省	126	11.8 扩展阅读.....	154
10.5 字符串和数据	127	第 12 章 使用 REST 服务	156
10.5.1 常量字符串	127	12.1 REST 简介	157
10.5.2 创建字符串	128	12.2 选择数据交换格式	157
10.5.3 转换为 C 字符串	129	12.2.1 在 iOS 中解析 XML	157
10.5.4 其他字符串操作符	131	12.2.2 在 iOS 中解析 JSON	158
10.5.5 字符串的支持存储	131	12.2.3 XML 与 JSON	159
10.5.6 CFData	132	12.2.4 模型版本化	160
10.6 容器类型	132	12.3 假想的 Web 服务	160
10.6.1 CFArray	133	12.4 重要提醒	161
10.6.2 CFDictionary	133	12.5 RESTfulEngine 架构 (iHotelApp 示例代码)	161
10.6.3 CFSet 和 CFBag	134	12.5.1 NSURLConnection 与第三 方框架	161
10.6.4 其他容器类型	134	12.5.2 创建 RESTfulEngine	162
10.6.5 回调函数	134	12.5.3 使用访问令牌对 API 调用 进行认证	165
10.7 自由桥接	136	12.5.4 在 RESTfulEngine.m 中覆 盖相关方法以添加自定义 认证头部	165
10.8 小结	139		
10.9 扩展阅读.....	139		

12.5.5 取消请求	166	13.5 使用蓝牙设备	192
12.5.6 请求响应	166	13.5.1 通过扫描寻找服务	192
12.5.7 对 JSON 数据进行键值编码	167	13.5.2 连接设备	194
12.5.8 列表页面的 JSON 对象与详细页面的 JSON 对象	169	13.5.3 直接获取外围设备	194
12.5.9 嵌套 JSON 对象	169	13.5.4 发现服务	194
12.5.10 少即是多	171	13.5.5 发现特性	195
12.5.11 错误处理	171	13.6 创建自己的外围设备	197
12.5.12 本地化	173	13.6.1 广播服务	197
12.5.13 使用分类处理其他格式	173	13.6.2 常见场景	200
12.5.14 在 iOS 中提升性能的小技巧	174	13.7 在后台运行	200
12.6 缓存	174	13.7.1 后台模式	200
12.7 需要离线支持的原因	174	13.7.2 电量考虑	200
12.8 缓存策略	175	13.7.3 状态保存和恢复	200
12.8.1 存储缓存	175	13.8 小结	201
12.8.2 缓存版本和失效	178	13.9 扩展阅读	201
12.9 数据模型缓存	178		
12.10 缓存版本控制	182		
12.11 创建内存缓存	183		
12.11.1 为 AppCache 设计内存缓存	184		
12.11.2 处理内存警告	185		
12.11.3 处理结束和进入后台通知	186		
12.12 创建 URL 缓存	186		
12.12.1 过期模型	187		
12.12.2 验证模型	187		
12.12.3 示例	187		
12.12.4 用 URL 缓存来缓存图片	188		
12.13 小结	188		
12.14 扩展阅读	188		
第 13 章 充分利用蓝牙设备	190		
13.1 蓝牙历史	190		
13.2 为什么选择低功耗蓝牙	191		
13.3 蓝牙 SDK	191		
13.3.1 服务器	191		
13.3.2 客户端	191		
13.4 类和协议	191		
13.5 使用蓝牙设备	192		
13.5.1 通过扫描寻找服务	192		
13.5.2 连接设备	194		
13.5.3 直接获取外围设备	194		
13.5.4 发现服务	194		
13.5.5 发现特性	195		
13.6 创建自己的外围设备	197		
13.6.1 广播服务	197		
13.6.2 常见场景	200		
13.7 在后台运行	200		
13.7.1 后台模式	200		
13.7.2 电量考虑	200		
13.7.3 状态保存和恢复	200		
13.8 小结	201		
13.9 扩展阅读	201		
第 14 章 通过安全服务巩固系统安全	203		
14.1 理解 iOS 沙盒	203		
14.2 保证网络通信的安全	204		
14.2.1 证书工作原理	205		
14.2.2 检验证书的有效性	207		
14.2.3 判断证书的可信度	210		
14.3 使用文件保护	211		
14.4 使用钥匙串	213		
14.5 使用加密	216		
14.5.1 AES 概要	217		
14.5.2 使用 PBKDF2 将密码转换成密钥	217		
14.5.3 AES 模式和填充	219		
14.5.4 初始化向量	220		
14.5.5 使用 HMAC 进行认证	221		
14.5.6 错误的密码	222		
14.5.7 组合使用加密和压缩	222		
14.6 小结	222		
14.7 扩展阅读	222		
第 15 章 在多个苹果平台和设备及 64 位体系结构上运行应用	224		
15.1 开发多平台应用	225		
15.1.1 可配置的目标设置：Base SDK 和 Deployment Target	225		

15.1.2 支持多个 SDK 时的注意事项：框架、类和方法	225	第 17 章 调试	252
15.1.3 检查框架、类和方法的可用性	227	17.1 LLDB	252
15.2 检测设备的功能	228	17.2 使用 LLDB 进行调试	252
15.2.1 检测设备及判断功能	228	17.2.1 dSYM 文件	253
15.2.2 检测硬件和传感器	229	17.2.2 符号化	254
15.3 应用内发送 Email 和短信	233	17.3 断点	255
15.4 支持新的 4 英寸设备族系	233	17.4 观察点	258
15.4.1 Cocoa 自动布局	235	17.5 LLDB 控制台	259
15.4.2 代码中固化屏幕尺寸	235	17.6 NSZombieEnabled 标志	262
15.4.3 iPhone 5s 和新的 64 位指令集	235	17.7 不同的崩溃类型	263
15.5 向 iOS 7 迁移	236	17.7.1 EXC_BAD_ACCESS	263
15.5.1 自动布局	236	17.7.2 SIGSEGV	263
15.5.2 支持 iOS 6	236	17.7.3 SIGBUS	264
15.5.3 应用图标	237	17.7.4 SIGTRAP	264
15.5.4 无边界按钮	237	17.7.5 EXC_ARITHMETIC	264
15.5.5 着色	237	17.7.6 SIGILL	264
15.5.6 图片更新	237	17.7.7 SIGABRT	264
15.6 向 64 位体系结构迁移	238	17.7.8 看门狗超时	265
15.6.1 数据溢出	238	17.7.9 自定义错误信号处理程序	265
15.6.2 序列化数据	239	17.8 断言	265
15.6.3 针对 64 位体系结构的条件编译	239	17.9 异常	267
15.7 UIRuntimeInterfaceRequired	239	17.10 收集崩溃报告	268
15.8 小结	240	17.11 第三方崩溃报告服务	269
15.9 扩展阅读	240	17.12 小结	270
第 16 章 国际化和本地化	242	17.13 扩展阅读	270
16.1 什么是本地化	242	第 18 章 性能调优	272
16.2 本地化字符串	243	18.1 性能思维模式	272
16.3 对未本地化的字符串进行审查	244	18.1.1 指导方针一：产品是为了取悦用户才存在的	272
16.4 格式化数字和日期	245	18.1.2 指导方针二：设备是为了方便用户而存在的	272
16.5 nib 文件和 Base Internationalization	248	18.1.3 指导方针三：做到极致	272
16.6 本地化复杂字符串	248	18.1.4 指导方针四：用户的感知才是实际的	273
16.7 小结	250	18.1.5 指导方针五：关注能带来大收益的方面	273
16.8 扩展阅读	251	18.2 欢迎走入 Instruments 的世界	273
18.3 查找内存问题	275	18.3 查找内存问题	275

18.4	查找 CPU 问题	279
18.4.1	Accelerate 框架	282
18.4.2	GLKit	283
18.4.3	编译器优化	283
18.4.4	链接器优化	284
18.5	绘图性能	284
18.6	优化磁盘访问和网络访问	286
18.7	小结	286
18.8	扩展阅读	286

第四部分 超越极限

第 19 章 近乎物理效果：UIKit 动力学		290
19.1	动画类、行为和动力项	290
19.2	UIKit “物理”	291
19.3	内置行为	292
19.3.1	迅速移动	292
19.3.2	附着	292
19.3.3	推力	293
19.3.4	重力	294
19.3.5	碰撞	294
19.3.6	动力项	295
19.4	行为层次结构	295
19.5	自定义操作	295
19.6	实战：一个“撕开”视图	296
19.6.1	拖拽视图	296
19.6.2	撕开该视图	297
19.6.3	添加额外效果	300
19.7	多个动力学动画类	301
19.8	与 UICollectionView 交互	302
19.9	小结	305
19.10	扩展阅读	305
第 20 章 魔幻的自定义过渡		306
20.1	iOS 7 中的自定义过渡	306
20.2	过渡协调器	307
20.3	集合视图和布局过渡	308
20.4	使用故事板和自定义联线的自定义视图控制器过渡	308
20.5	自定义视图控制器过渡：iOS 7 风格	308

20.6	使用 iOS 7 SDK 的交互式自定义过渡	310
20.7	小结	312
20.8	扩展阅读	313
第 21 章 精妙的文本布局		314
21.1	理解富文本	314
21.1.1	字符与字形	314
21.1.2	理解字体	316
21.1.3	段落样式	316
21.2	属性化字符串	317
21.2.1	用字体描述符选择字体	318
21.2.2	设置段落样式	319
21.2.3	HTML	319
21.2.4	简化属性化字符串的使用	320
21.3	动态字体	321
21.4	Text Kit	322
21.4.1	Text Kit 的组件	323
21.4.2	多容器布局	324
21.4.3	排除路径	325
21.4.4	继承文本容器	326
21.4.5	继承文本存储	327
21.4.6	继承布局管理器	331
21.4.7	针对字形的布局	334
21.5	Core Text	337
21.5.1	用 CTFramesetter 进行简单的布局	337
21.5.2	为非连续路径创建框架	338
21.5.3	排版器、文本行、连续文本和字形	340
21.6	小结	340
21.7	扩展阅读	341
第 22 章 Cocoa 的大招：键值编码和观察		342
22.1	键值编码	342
22.1.1	用 KVC 赋值	344
22.1.2	用键路径遍历属性	345
22.1.3	KVC 和容器类	345
22.1.4	KVC 和字典	347
22.1.5	KVC 和非对象	347

22.1.6 用 KVC 实现高阶消息传递	347	23.8 扩展阅读	367
22.1.7 容器操作符	347		
22.2 键值观察	348	第 24 章 深度解析 Objective-C	368
22.2.1 KVO 和容器类	350	24.1 理解类和对象	368
22.2.2 KVO 是如何实现的	351	24.2 使用方法和属性	370
22.3 KVO 的权衡	351	24.3 使用方法签名和调用	372
22.4 小结	352	24.4 消息传递如何工作	378
22.5 扩展阅读	353	24.4.1 动态实现	378
第 23 章 超越队列：GCD 高级功能	354	24.4.2 快速转发	380
23.1 信号量	354	24.4.3 普通转发	383
23.2 分派源	356	24.4.4 转发失败	384
23.3 定时器源	358	24.4.5 各种版本的 <code>objc_msgSend</code>	384
23.4 单次分派	358	24.5 方法混写	385
23.5 队列关联数据	359	24.6 ISA 混写	387
23.6 分派数据和分派 I/O	362	24.7 方法混写与 ISA 混写	388
23.7 小结	367	24.8 小结	389
		24.9 扩展阅读	389

Part 1

第一部分

全新功能

本部分内容

- 第1章 全新的系统
- 第2章 世界是平的：新的UI范式

全新的系统

iOS 7 为苹果设备带来了全新的用户界面，但是 iOS 7 在用户界面上的变化不仅仅是表面的。就像本书一样，iOS 7 是给用户和开发者双方都带来好处的一次大升级。本章我们从新的 UI 及其对应用的影响开始讨论，然后是各种 SDK 的附加功能，还有新的 IDE 和编译器 Xcode 5/LLVM 5。

iOS 7 也为 SDK 带来了大量全新的核心特性，能帮助应用脱颖而出。iOS 7 增加了两种新的技术，分别是 UIKit Dynamics 和 UIMotionEffects，它们能帮你实现用 Quartz Core 不好实现的动画。过去只有用故事板才能实现视图之间的自定义过渡效果，现在已经可以用在任意两个视图控制器之间的过渡上了。内置的日历和照片应用就是很好的例子，它们利用自定义过渡效果来完美地为用户提供其在应用中所处的位置。Text Kit 则是基于 Objective-C 的加强版 Core Text，且易于使用，可以说是最重要也最有意思的新增特性。SDK 还有其他的新增特性，举几个例子，Sprite Kit、Dynamic Text、更紧密的 MapKit 集成、针对所有应用的真正的多任务、更好的低功耗蓝牙支持。

新特性还不止这些，随着新的 SDK 特性而来的还有 Xcode 5，它是 iOS 7 的新 IDE，用 ARC 编译器完全重写过。Xcode 5 速度更快，不易崩溃，新的 IDE 还引入了新编译器 LLVM 5。

苹果再一次对平台做了巨大的改动，这是颠覆性的，会让本已拥挤不堪的 App Store 中的大部分应用都过时。对于很多应用来说，App Store 又成了一个可以闯出一番天地的市场，作为 iOS 开发者，大家的未来就如 2008 年 iOS SDK 刚发布时那么好。

现在我们深入理解一下这些新特性，从新的 UI 开始。

1.1 新的 UI

iOS 7 的 UI 改变巨大，不只是表面的。iOS 7 去除了 UI 元素的拟物效果，强化了真实世界的物理特性，比如运动模拟。

iOS 7 主要关注以下几个主题：依从、清晰和层次。内容要比外表重要。如今应用要尽量用全屏显示内容，过去的“镀层效果”变成了半透明，并且会以柔化、模糊的效果显示后面的内容。所有的工具栏、导航栏和状态栏也都变成了半透明状态。除了半透明，它们还会模糊并染上背后 UI 的颜色，从而给用户一种层次的感觉，如图 1-1 所示。



图 1-1 时钟应用的计时器选项卡周围的区域模糊显示了背后的黑色表盘

空间上的层次不只表现在工具栏、导航栏和状态栏的半透明效果，而是贯穿整个 UI。多任务栏的 home 键就是 iOS 7 的 UI 如何展现空间层次的绝佳例子。双击 home 键会看到应用的截屏和应用的图标，不过这不重要，仔细看这里的动画效果：整个屏幕拉远后显示了其他所有正在运行的应用。点击某个应用图标又会拉近到那个应用。日历应用也有动画来提供空间层次的感觉。从年视图进入月视图再到日视图，该过程使用了精妙的过渡效果来强化空间层次的感觉。点击某个月份进入月视图，再在月视图上点击年份就会拉远。类似地，点击应用图标启动应用会拉近，不像之前的 iOS 版本那样从屏幕中央拉近，而是从应用图标的中心拉近。关闭应用则会产生相反的效果，会缩回图标而不是屏幕中央，以便用户知道当前的状态。

新的通知中心、控制中心，甚至还有某些模态视图控制器（比如 iPod 的专辑列表视图控制器）也用模糊效果来显示背后的东西。显示背后的东西能让用户知道自己在哪儿。

对于开发者来说，这些新变化中最有意思的是，在大部分情况下，如果使用 UIKit 来构建应用，就不需要做额外的工作让这些特性生效。在第 2 章中你会更加了解 UIKit 的新特性以及如何为 iOS 7 设计应用。

1.2 UIKit Dynamics 和 Motion Effects

iOS 7 的 SDK 增加了能让开发者为用户界面加入运动拟真的类。任何符合 `UIDynamicItem` 协议的对象都能加上运动拟真效果。从 iOS 7 开始，所有的 `UIView` 都符合这个协议，所以应用中的任何 `UIView` 的子类（包括 `UIControl`）都能加上运动拟真效果。

在 iOS 7 之前要实现像运动拟真一样的物理特性也是可能的，需要用到 `QuartzCore.framework` 里的动画方法。但是 iOS 7 的 UIKit 中的新 API 让这个任务变得非常简单。在 iOS 7 中实现拟真运动要比使用 `QuartzCore.framework` 容易，因为前者采用声明式写法指定 UIKit 的动力学行为，把动力

学行为“附着”到 `UIView` (或其子类) 上即可, 而后者本质上是命令式写法。

为视图添加 `UIDynamicAnimator` 并为该视图的子视图附着以下动力学行为之一——`UIAttachmentBehavior`、`UICollisionBehavior`、`UIGravityBehavior`、`UIPushBehavior`、`UISnapBehavior`, 即可实现运动拟真。这些行为类暴露的属性可以用来自定义行为以及微调行为, 动力学动画类会搞定剩下的事情。之前说过, 实际编码过程更多是声明式的, 而不是命令式的。只要告诉 `UIDynamicAnimator` 你所需要的, 它就会帮你乖乖干活。

除了 `UIView` 以外, `UICollectionViewLayoutAttributes` 也符合 `UIDynamicItem` 协议。这意味着开发者也能把动力学行为附着到集合视图单元上。一个很好的例子是内置的信息应用, 上下滚动消息对话时能看到聊天气泡之间有逼真的碰撞运动效果。信息应用已经不用表格视图了, 而是用集合视图和附着在其单元之上的动力学行为。

`UIMotionEffect` 是 iOS 7 中增加的又一个类, 它能帮助开发者为用户界面加上运动拟真。使用 `UIKit Dynamics` 能基于程序实现的物理效果实现运动拟真。用 `UIMotionEffect` 则能基于设备的运动实现运动拟真。凡是能够通过 `CAAnimation` 实现动画的都能用 `UIMotionEffect` 实现。`CAAnimation` 的动画是时间的函数, 而 `UIMotionEffect` 的动画则是设备运动的函数。

第 19 章会进一步介绍 `UIKit Dynamics` 和 `UIMotionEffect`。第 5 章则讲到了如何用 `UIKit Dynamics` 来在集合视图上实现类似于信息应用的效果。

1.3 自定义过渡效果

在 iOS 7 之前, 自定义过渡效果只能在用故事板创建用户界面时才能使用。通常的做法是创建一个 `UIStoryboardSegue` 的子类, 覆盖 `perform` 方法来实现过渡。iOS 7 把这个概念带到了一个新的高度, 允许任何用 `pushViewController:animated:` 方法推入的视图控制器都能使用自定义过渡效果。这通过实现 `UIViewControllerAnimatedTransitioning` 协议就可以做到。

在第 20 章中, 我们将学习如何用传统方法 `UIStoryboardSegue` 和新方法 `UIViewControllerAnimatedTransitioning` 创建过渡效果。

1.4 新的多任务模式

从 iOS 4 到 iOS 6, 只有某些类别的应用可以在后台运行, 比如需要及时得知用户位置变动的基于地理位置的应用, 或者需要串流并在后台播放音乐的音乐播放器应用。但是不同于桌面版本, 应用并不是真正在后台运行, 而是将自己注册为周期性启动以便在后台获取内容。iOS 7 为 `UIBackgroundMode` 键引入了一个新值 `fetch` 来支持这种模式。

还有一种有意思的新东西是远程通知 (`remote-notification`) 的后台模式, 能让应用在接收到远程通知 (推送通知) 时下载或获取新数据。

第 9 章会详细讨论新的多任务模式。

此外, `bluetooth-central` 和 `bluetooth-peripheral` 后台模式现在支持一种新技术, 叫做状态保存和恢复, 它跟 `UIKit` 中的同名技术类似。蓝牙相关的多任务模式会在第 13 章讲到。

1.5 Text Kit

Text Kit 是基于 Core Text 构建的快速现代 Unicode 文本布局引擎。Core Text 严重依赖于 `CoreFoundation.framework`, 而 Text Kit 则更现代, 使用的是 `Foundation.framework` 中的类。这也意味着与 Core Text 不同, Text Kit 对 ARC (自动引用计数) 比较友好。在 iOS 7 之前, 大部分渲染文本的 UIKit 组件 (比如说 `UILabel`、`UITextView` 和 `UITextField`) 都用到了 WebKit。苹果重写了这些类, 并在 Text Kit 的基础上构建 WebKit, 这意味着 Text Kit 成了 UIKit 中的一等公民, 开发者能够而且也应该使用 Text Kit 做所有的文本渲染。Text Kit 以三个类 (`NSTextStorage`、`NSLayoutManager` 和 `NSTextContainer`) 为中心, 还有对 `NSAttributedString` 的扩展。Text Kit 本身没有独立的框架, 相关的类和扩展方法添加到了 `UIKit.framework` 和 `Foundation.framework` 中。你会在第 21 章了解到 Text Kit 的相关知识。

1.6 动态字体

在 iOS 7 中, 用户可以设置文本大小, 几乎所有的内置应用都会遵守这个设置, 并且会基于用户的偏好改变文本大小。注意增大文本不总是会让字体的磅数变大。如果设置的文本大小可能会让渲染的小磅数文本难以辨认, iOS 7 会自动加粗文本。在应用中加入动态字体的支持, 就能根据 iOS 的设置应用中的设置来动态改变文本大小。iOS 7 的动态字体有一个缺点: 不支持自定义字体。你会在第 21 章了解到动态字体的相关知识。

1.7 MapKit 集成

iOS 7 更加紧密地集成了苹果地图, 现在 `MKMapView` 对象可以显示 3D 地图了, 如果你是地图供应商, 同样可以用 `MKMapView` 控件, 不过可以提供自己的地图块来替换内置的地图块。此外, 地图供应商还能用 `MKDrections` 在不退出应用的情况下在地图上显示路线。

1.8 SpriteKit

SpriteKit 是苹果对 `cocos2d` 和 `box2d` 的回应。SpriteKit 提供了干净的 OpenGL 封装, 跟 `QuartzCore.framework` 非常类似。`QuartzCore.framework` 主要用来做 UI 动画, 而 SpriteKit 则更多是服务于模拟 3D (也就是 2.5D) 的游戏制作。本书不会讲述 SpriteKit。

1.9 LLVM 5

iOS 7 的另一个重大变化是 Xcode 5 和新的 LLVM 5 编译器。LLVM 5 提供了用户可见的特性和性能特性。LLVM 5 提供了生成 `armv7s` 和 `arm64` 指令集的完整支持, 这意味着只要用 LLVM 5 重新编译应用, 应用就可以在 iPhone 5s 和最新的 iPad Air 上运行得更快。

在 C++这边, LLVM 5 现在完全支持 C++11 标准, 还用新的 `libc++` 库代替了旧的 `gnuc++` 库。

在 Objective-C 这边, LLVM 5 增加了新的编译器警告, 默认打开了几项已有的警告。比如说, 编

译器会提醒无用代码——那些没有任何地方会调用的代码。

可以说，最大的变化是把枚举量变成了一等公民。枚举量是类型，现在如果试图把一种枚举量转换为另外一种，LLVM 5 会发出警告。

LLVM 还能检测到正在使用的选择器是否在当前作用域中进行了声明，如果没有，LLVM 5 会发出警告。

对Objective-C语言的加强

今年 Objective-C 语言有两个主要的加强，那就是模块和自动引用计数的变化。

1. 模块

苹果引入了模块，用来代替#import 语句。C/C++已经用了三四十年的#import 语句，在引入第三方库或框架时很好用。过去几年来，C/C++应用程序在变大，#import 语句开始出现复杂的编译问题。每个.m 文件都可能因为几行 import 语句就引入几千行代码，而且在这些 import 语句中，有些会在很多文件中重复出现，比如说#import <UIKit/UIKit.h> 和#import <Foundation/Foundation.h>。

问题在于每次修改一个文件，编译器都需要编译那个文件，连带这个文件引入的代码。很明显这会让编译时间变成 $O(S \times H)$ ，这里 S 是源代码文件数，而 H 是引入的头文件数。为了避免这个问题，一些有经验的开发者曾经尝试在头文件中写前向声明而在实现文件中用 import 语句，然后把常用的头文件加入预编译头文件中。这种方法在一定程度上解决了该问题，减少了一些编译时间。但是有时候在预编译头文件中加入不那么常用的头文件会造成命名空间污染，而一个写得很差的框架（定义类和宏时没有用前缀）可能会被另外一个写得很差的框架里的类覆盖。

为了解决这些问题，苹果引入了模块。模块在语义上把框架和 import 语句封装到了代码中，而不是把框架的内容复制粘贴到代码中。

模块可以把编译复杂度从 $O(S \times H)$ 降低到 $O(S+H)$ ，这意味着无论导入一个模块多少次，都只会编译一次。也就是说 S 个源文件和 H 个头文件会编译 $(S+H)$ 次而不是 $(S \times H)$ 次。事实上，H 个头文件会预编译到动态库 dylib 中，并且会自动链接，这样就能把编译时间从 $O(S \times H)$ 降低到 $O(S)$ 。

只有最新的 Xcode 5 和 LLVM 5 编译器支持模块，更重要的是，只有最新的 iOS 7 SDK 才支持模块。如果你的工程必须支持 iOS 6，那就只能再等等。使用模块导入内置库时，编译速度会有很大提升，尤其是大型工程。所有用 Xcode 5 创建的新工程都默认使用模块。用 Xcode 5 打开一个已有工程时，可以在 Build Settings 选项卡下打开模块支持，如图 1-2 所示。

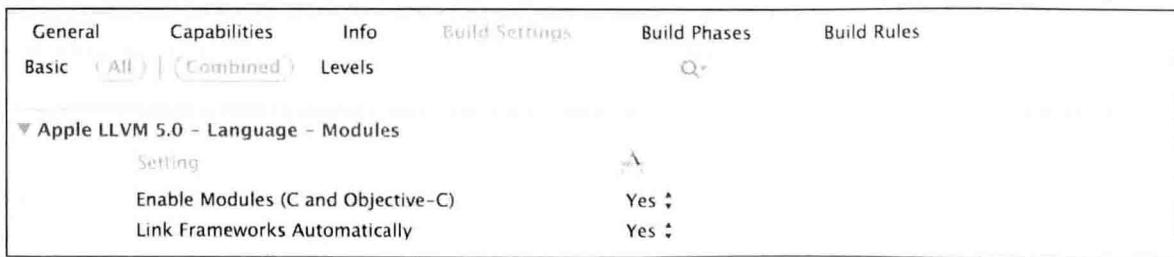


图 1-2 Xcode 5 中 LLVM 5 的模块支持

模块还省去了在工程中显式地链接框架的麻烦，以前需要在工程设置的 Build Phases 选项卡中添

加框架才可以做到。不过这个版本的 Xcode/LLVM 只对内置库提供模块支持。这一点可能会在不久的将来改变（就是 iOS 8）。

2. ARC 的改进

LLVM 5 有了更好的循环引用检测，而且也能预测一个弱变量会不会在被块捕获之前变为空。不过更有意思的是 ARC 和 Core Foundation 类的整合。在 LLVM 前面的版本中，编译器规定必须用桥接转换，即使对象的所有权没有变化。有了 LLVM 5，这种情况会变成自动的。这意味着如果没有所有权变化，就无需再在 Core Foundation 对象和对应的 Foundation 对象之间用桥接转换了。

第 20 章会详细解释 Core Foundation 和 ARC 的改进。

1.10 Xcode 5

Xcode 4 是苹果很有野心的项目。2011 年，苹果完全抛弃了 Xcode 3 时代独立的 Xcode 和 Interface Builder 这个已经存在了近 10 年的组件，把编辑 Interface Builder 文件的功能集成到了 Xcode 中。不过，Xcode 4 运行缓慢，还经常崩溃，在一些 WWDC 的讲座上也发生过这类情况。

Xcode 4 性能差的主要原因是它使用旧的 GCC 编译器，而且用了垃圾回收（而不是 ARC）。Xcode 5 在面向用户上的变化很少，但是从头开始的。它是用 LLVM 5 编译的，用 ARC 重写过，这意味着它的内存使用更高效，在的机器上运行更快。

1.10.1 nib文件格式的变化

Xcode 5 的一个主要变化是引入了一种更易读的新 nib 文件格式。格式仍然是 XML，但是对人类更友好。这意味着作为开发者，解决 XIB 文件中的合并冲突会容易得多。

不过新文件格式和旧版 Xcode 不兼容，但是大部分情况下这都不会是问题。确保团队中所有开发者都用最新的 Xcode。Xcode 4.6 无法打开新的文件格式。

1.10.2 源代码控制集成

Xcode 5 现在支持合并和切换 Git 分支。Xcode 5 还能在不离开 IDE 的情况下从远程仓库推送和拉取提交。这意味着开发者可以在 Xcode 以外花费更少的时间（无论是终端还是第三方的 Git 客户端），而把更多的精力放在 Xcode 中。

1.10.3 自动配置

Xcode 4 已经集成了 iOS 开发者账号，允许开发者自动创建和下载授权文件。Xcode 5 把这种集成带到了更高的层次上，开发者可以使用 Apple ID 登录 Xcode。Xcode 5 会自动获取开发者合作的团队，并把它们连接到 Xcode。开发者甚至可以把一个团队指定给一个项目，Xcode 会自动在开发者网站上创建必需的授权文件。

Xcode 5 新增了一种特性叫 capabilities，能让开发者在 Xcode 中设置高级的苹果特性，比如打开应用内购买、iCloud 或者 Game Center。Xcode 现在会根据新的 App ID 自动下载必需的授权文件，如果有必要的话甚至会更新 Info.plist 文件。

1.10.4 对调试导航面板的改进

Xcode 的调试导航面板现在可以在 Xcode 中就显示实时的应用程序内存、CPU 使用以及电量消耗。不再需要频繁地启动 Instruments 来测量应用的性能。调试导航面板还允许开发者从 Xcode 中启动以 CPU 或内存检测工具运行的 Instruments。图 1-3 和图 1-4 显示了 Xcode 5 中的 CPU 和内存报告功能。

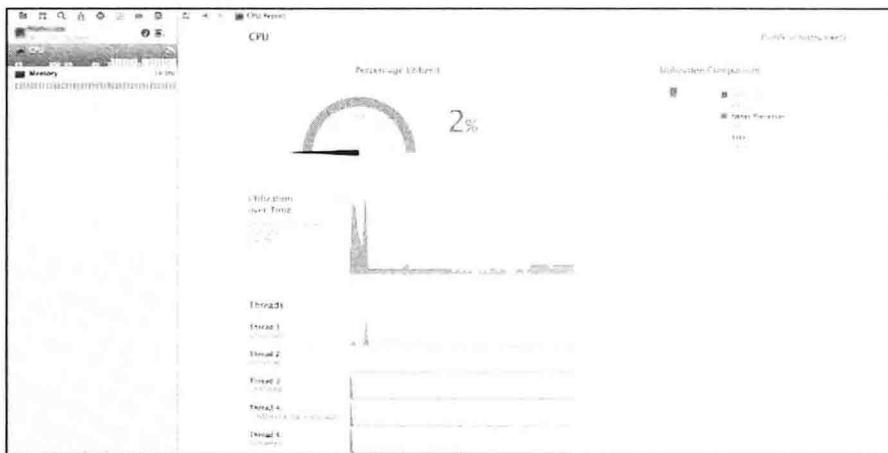


图 1-3 Xcode 5 的调试导航面板中显示的当前运行应用的 CPU 使用

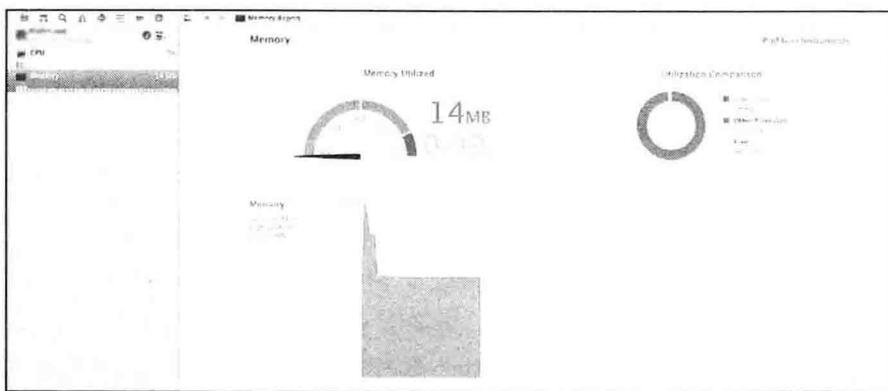


图 1-4 Xcode 5 的调试导航面板中显示的当前运行应用的内存使用

1.10.5 文档浏览器

Xcode 5 有了全新的支持选项卡浏览的文档浏览器，开发者可以为文档页添加书签（不幸的是文档书签不支持 iCloud 同步）。目录现在是在文档内容外面用一个单独的选项卡显示的。

Dash 是一个第三方应用程序，要比 Xcode 5 的文档浏览器好很多，也快很多。它在 Mac App Store 中是免费的，强烈推荐读者下载。Dash 能自动找到 Xcode 安装的文档，也能提供第三方文档，比如 cocos2d。

1.10.6 Asset Catalog

Asset Catalog 提供了一种新的方式来为应用中的图片分组。Asset Catalog 包含图片集（应用中用到的图片和资源）、应用图标和启动图。这些启动图、应用图标和图片集会基于设计它们所针对的设备来分组。

Asset Catalog 是一个 Xcode 的特性，所以就算应用需要支持 iOS 6 也能使用。在 iOS 6 上，Xcode 会确保 `UIImage` 的 `imageNamed:` 方法会返回 catalog 中的正确图片。在 iOS 7 上，Xcode 5 会把 Asset Catalog 编译成运行时二进制文件 (.car 文件)，能减少应用下载的时间。

Asset Catalog 还提供了一种创建可拉伸图片的方法，能让开发者在图片中指定可拉伸的区域。开发者要做的只是在 Asset Catalog 中选择一张图片，在图片顶部点击 Start Slicing 按钮，调整切片区域来指定可拉伸的区域。可以选择只垂直拉伸、只水平拉伸或者两个方向都拉伸。

不过，用 Asset Catalog 创建可拉伸区域（见图 1-5）只能在以 iOS 7 及后续版本为部署目标的工程中使用。

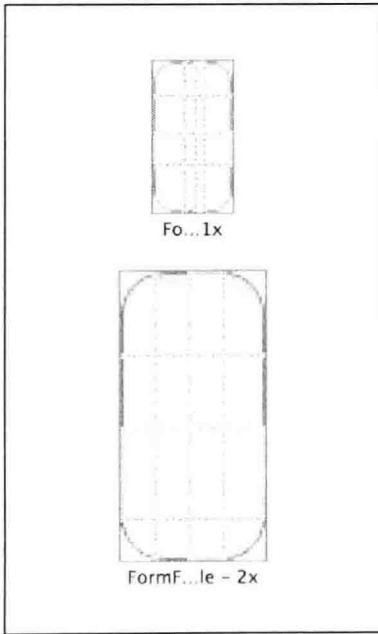


图 1-5 在 Asset Catalog 的图片集中创建可拉伸区域

1.10.7 测试导航面板

Xcode 5 让编写单元测试用例成了一等公民。在 Xcode 5 中创建的所有新工程都会自动包含新的测试框架 `XCTest.framework`，用来代替 `OCUnit` (`SenTestingKit.framework`)。Xcode 5 中还是带 `SenTestingKit.framework`，Xcode 5 有一个选项可以自动把 `OCUnit` 的测试用例转换为 `XCTest`。这么做好处在于可以方便地用 `xcodebuild` 这样的命令行工具做测试。除了前一个版本的 Xcode 中已有的 7 种导航面板，Xcode 还增加了一个新的导航面板叫测试导航面板。测试导航面板可以显示开发者编写的测试用例的执行情况。

可以创建一种新断点，用来在测试失败的时候停止程序执行，如图 1-6 所示。这种断点可以在断点导航面板中添加。

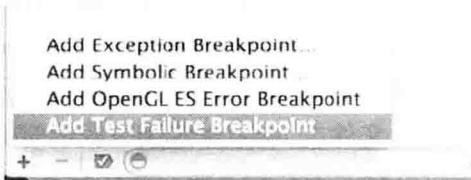


图 1-6 添加测试失败断点

1.10.8 持续集成

Xcode 5 通过 Xcode 服务（在 OS X Mavericks 服务器上运行）支持持续集成。开发者现在可以创建自动执行“每次提交都运行静态分析”或者“在每晚 12 点编译产品”等操作的机器人。

这类机器人运行在 OS X Mavericks 服务器上，开发者在客户端的测试导航面板中就能看到结果。

Xcode 5 没有原生的 Jenkins 支持，Jenkins 是普遍使用的持续集成工具。开发者仍然需要手动捣鼓命令行、shell 脚本，甚至是提交后的钩子。Xcode 5 也不支持 OS X Mountain Lion 服务器的持续集成。不过，用 Xcode 5 和 OS X Mavericks 有个小小的好处，诸如断点测试等代码问题会出现在客户端的测试导航面板中。

1.10.9 Auto Layout改进

Xcode 5 改进了开发者使用 Auto Layout 的工作流。最大的变化是 Xcode 不会自动添加 Auto Layout 约束条件。开发者必须亲自动手让 Xcode 为我们添加约束条件，开发者也能删除 Xcode 添加的约束条件。如果删除的约束条件会导致布局歧义，Xcode 会发出警告。如果不能同时满足几个约束条件，Xcode 也会发出警告。Auto Layout 和 Xcode 5 的变化无法用一节内容描述清楚，所以我们为此添加了一整章的内容，也就是第 6 章。

1.10.10 iOS模拟器

现在 iOS 模拟器支持 iCloud 了，而且在 iOS 模拟器中运行的应用可以访问 iCloud 的数据了。不过 iOS 模拟器不再支持低功耗蓝牙的模拟。（关于蓝牙的支持详见第 13 章。）

1.11 其他

除了 LLVM 编译器、Xcode IDE 这些新东西，iOS 7 也加强了 UIActivityViewController 来

支持用 AirDrop 与附近的设备共享数据。一种称为 Multipeer Connectivity 的类似的技术，能让开发者在不需要热点的情况下就连接附近的设备并发送任意数据。Multipeer Connectivity 还能在附近设备用点对点 WiFi 网络的方式连接着别的 WiFi 热点的情况下与其连接。

1.12 小结

呼！好多变化。前面说过 iOS 7 是 iOS 诞生以来最大的升级，对于用户和开发者来说都是大事。

iOS 的升级速率一直要比竞争对手快很多。在写作本书时，根据各大网站的统计，iOS 7 已经占据了市场上 70% 以上的设备，而 iOS 6 只占不到 20%。事实上，iOS 7 在发布的第一周就被 30% 以上的设备接纳了。

如果在 App Store 上已经有了应用，那可以考虑把它当做只针对 iOS 7 的新应用重写，然后和旧版应用一起卖。当大量用户升级到 iOS 7 之后，就可以把旧版应用从 App Store 下架了。对最终用户来说，有很多杀手级的特性，比如全新的用户界面、iTunes Radio、Air Drop 以及更强的多任务，都会加快他们升级系统。所以，为什么还要等呢？让我们从下一章全新的 UI 范例开始 iOS 7 之旅吧。

1.13 扩展阅读

1. 苹果文档

下面的文档位于 iOS Developer Library (<https://developer.apple.com/library/ios/navigation/index.html>) 中，通过 Xcode Documentation and API Reference 也可以找到。

- *What's New in iOS 7*
- *What's New in Xcode*
- *Xcode Continuous Integration Guide*
- *iOS 7 UI Transition Guide* (TP40013174 1.0.43)
- *Assets Catalog Help*

2. WWDC 讲座

下面的讲座视频可以在 developer.apple.com 找到。

- WWDC 2013 “Session 400: What's new in Xcode 5”
- WWDC 2013 “Session 402: What is new in the LLVM compiler”
- WWDC 2013 “Session 403: From Zero to App Store using Xcode 5”
- WWDC 2013 “Session 406: Taking control of Auto Layout in Xcode 5”
- WWDC 2013 “Session 409: Testing in Xcode 5”
- WWDC 2013 “Session 412: Continuous integration with Xcode 5”

世界是平的：新的 UI 范式

iOS 7 给苹果设备带来了全新的用户界面 (UI)。iOS 7 在 UI 上的变化是自其诞生以来最大的。iOS 7 专注于三个重要的特点：清晰、依从和层次。理解这三个特点很重要，因为这有助于设计跟原生的系统内置应用一样的应用。

本章将介绍 iOS 7 引入的一些重要变化以及如何让应用使用这些新特性。前半章展示开发者应该了解的重要的 UI 范式，利用这些范式可以使应用更上一层楼。后半章展示如何把已有的 iOS 6 应用迁移到 iOS 7 上，并且需要的话，也可以保持向后兼容性。

2.1 清晰、依从和层次

清晰 (clarity) 很简单，就是对用户来说表达的意思是清晰的。大部分 iOS 用户会抓住一些碎片时间使用应用，他们会打开 Facebook，查看通知或者新闻订阅，发布状态更新或照片，然后关闭应用。至少在 iPhone 上，很少有应用是用户长时间使用的。事实上统计表明，对于大部分应用，用户每次只会用 80 秒左右。这意味着开发者必须在极短的时间内把信息传递给用户，因此应该把最重要的信息清晰地展示在屏幕上。举个例子，如果开发天气应用，温度和天气状况是用户最感兴趣的两块信息，要把 UI 设计得足够简洁，让用户不需要在视野范围内寻找就能直接看到。

依从 (deference) 是说操作系统不会提供和应用的 UI 竞争的 UI，相反，它依从于应用及其内容。这意味着每个应用都会有独一无二的外观和体验。在 iOS 7 之前，应用的 UI 是用户看到的在系统提供的导航栏和工具栏内部的东西，镀层效果、不规则边框和渐变色被依从于其背后内容的半透明栏取代。操作系统的默认应用依从于内容，而 UI 在大部分情况下只起辅助作用。

为 iOS 7 设计的应用应该具有层次 (depth) 的概念。控件和内容应该处于不同的层，从 home screen 开始就是这样，视差效果给人感觉图标是浮在最上面的一层上，模态化的警告框也有类似效果。通知中心和控制中心也有层次，它们用的是模糊和半透明而不是视差效果。

2.2 动画、动画、动画

使用 iOS 7 后开发者发现的第一件事可能就是应用的视觉拟真（拟物）没有了。阴影很不明显，几乎看不出来，抛光按钮完全消失了。真实世界的拟物元素，比如备忘录 (iPhone) 和通讯录 (iPad) 的人造革，也都不见了。

最重要的是，应用程序窗口默认就是全屏的，状态栏现在是半透明的，浮在应用程序窗口的上方。

注意，直到 iOS 6，应用程序的全屏和状态栏的半透明一直都只是可选的。在 iOS 6 中，必须把导航栏的 `translucent` 属性设置为 YES，并且把视图控制器的 `wantsFullScreenLayout` 设置为 YES 才行。在 iOS 7 中所有的窗口默认都是这么设置的。

尽管视觉拟真没有了，iOS 7 强化了运动拟真。运动拟真的强化从锁屏开始。试着拖动摄像头按钮把锁屏界面“提起来”，然后从屏幕中间“掉下来”，你会看到锁屏界面会在碰到屏幕底部时弹起来。现在再试着把锁屏界面往屏幕底部推一下来“撞击”，你会看到弹得更厉害了，就跟真实世界的物体撞到表面一样。所以，拟物并没有完全从 iOS 7 中消失，之前版本的 iOS 强调视觉拟物，而 iOS 7 更加重视物理拟物。

iOS 7 仍然是拟物的，只不过不是强调视觉拟物而是物理拟物，屏幕上的物体遵从物理定律，其行为类似于真实世界的物体。

2.2.1 UIKit Dynamics

在应用中实现物理特性很简单，iOS 7 引入了一个新类 `UIDynamicAnimator`，用来模拟真实世界的物理定律。开发者可以创建一个 `UIDynamicAnimator` 对象并将其添加到视图上，这个视图通常是视图控制器的根视图，在 UIKit Dynamics 上下文中通常也被称为引用视图，引用视图的子视图现在就会基于其关联的行为遵从物理定律。

行为可以通过声明定义并添加到视图。事实上，任何遵从 `UIDynamicItem` 协议的对象都可以添加行为。`UIView` 及其子类（包括 `UIButton`）遵从这个协议，这意味着屏幕上可见的任何东西几乎都可以关联一种行为。

这里还有一件有意思的事情，`UICollectionViewLayoutAttributes` 遵从 `UIDynamicItem` 协议，这样可以让集合视图的元素有动力学行为。默认的信息应用就是一个例子。

2.2.2 UIMotionEffect

物理拟物还没完。主屏幕上的视差效果，警告框好像浮动在视图上方的效果，Safari 新的切换选项卡 UI 在倾斜设备时会显示“更多”内容，这些都是真实行为的模拟。用 `UIMotionEffect` 很容易就能为应用添加这类特性。iOS 7 中的一个新类 `UIInterpolatingMotionEffect`，能让开发者不费吹灰之力就添加这类效果。可以把 `UIMotionEffect` 理解为跟 `CAAnimation` 类似。`CAAnimation` 会对其所属的层做动画，`UIMotionEffect` 会对其所属的视图做动画。`CAAnimation` 的动画是时间的函数，`UIMotionEffect` 的动画则是设备动作的函数。我们会在第 19 章详细介绍 UIKit Dynamics 和 `UIMotionEffect`。

`CAAnimation` 动画是时间的函数，`UIMotionEffect` 是设备动作的函数。

2.3 着色

`UIView` 有了一个新属性，叫做 `tintColor`，可以给应用着色。所有作为其他视图的子视图存在的 UI 元素都会在未设置 `tintColor` 的情况下使用父视图的 `tintColor`。这意味着给应用程序的窗口设置 `tintColor` 就可以得到全局的着色。

这里要注意的很重要的一点是当模态化视图出现时，iOS 7 会将其背后所有的 UI 元素变模糊。如果开发者有一个自定义视图，并且用了父视图的 `tintColor` 来做一些自定义渲染，就要覆盖 `tintColorDidChange` 方法来更新变化。

也可以在 Xcode 的故事板编辑器的 File inspector 面板中设置 `tintColor`。

2.4 用半透明实现层次和上下文

UIKit Dynamics 和 UIMotionEffect 能帮助用户理解应用中的空间深度。iOS 7 中还有一个强化空间深度概念的特性是在大部分模态化窗口上统一使用了模糊和半透明效果。控制中心和通知中心用一种精细的方式模糊了背景，以便用户能知道背后是什么。

iOS 7 通过直接读取显存来实现这种效果。遗憾的是，iOS 7 没有提供 SDK 来让开发者轻松实现这种效果，可能是出于安全上的考虑。比如说，无法这么做：

```
self.view.blurRadius = 50.0f;
self.view.saturationDelta = 2.0f;
```

不过，有一个 WWDC 讲座中的示例代码有 `UIImage` 的分类，`UIImage+ImageEffects`（本书的示例代码中有）允许实现这种效果，尽管这种方法很难用也不直观。但是你想超越极限，这也是你阅读本书的原因，不是吗？所以，在自己的应用中实现这种效果吧。

现在要展示如何创建一个看起来像通知中心或者控制中心背景的图层。首先创建一个单视图应用，挑一张漂亮的背景图加上，再添加一个按钮。接下来写一个按钮动作处理方法，弹出一个会将后面的背景模糊的视图。

捷径（或者说投机取巧）是创建一个不可见的 `UIToolbar` 并使用它的图层，而不是创建新图层。不过这种技巧很容易失效，所以不推荐。更好的办法是用 iOS 7 新的 `UIScreen` 接口截屏，切割截屏图片，为其添加模糊效果，并把模糊的图片作为弹出视图的背景。

来看看代码：

用 `UIImage+ImageEffects` 创建模糊的弹出层（`SCTViewController.m`）

```
// 创建图层
self.layer = [CALayer layer];
self.layer.frame = CGRectMake(80, 100, 160, 160);
[self.view.layer addSublayer:self.layer];

// 截屏
float scale = [UIScreen mainScreen].scale;
```

```

UIGraphicsBeginImageContextWithOptions(self.view.frame.size, YES, scale);
[self.view drawViewHierarchyInRect:self.view.frame afterScreenUpdates:NO];
__block UIImage *image = UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();

// 裁剪截图
CGImageRef imageRef = CGImageCreateWithImageInRect(image.CGImage,
    CGRectMake(self.layer.frame.origin.x * scale,
              self.layer.frame.origin.y * scale,
              self.layer.frame.size.width * scale,
              self.layer.frame.size.height * scale));
image = [UIImage imageWithCGImage:imageRef];

// 添加效果
image = [image applyBlurWithRadius:50.0f
                           tintColor:
[UIColor colorWithRed:0 green:1 blue:0 alpha:0.1]
                           saturationDeltaFactor:2
                           maskImage:nil];

// 放到新建的图层上
self.layer.contents = (__bridge id)(image.CGImage);

```

applyBlurWithRadius:tintColor:saturationDeltaFactor:maskImage:方法位于 `UIImage+ImageEffects` 分类中，包括分类在内的完整代码可以从本书网站上下载。

图 2-1 显示了代码运行的效果。

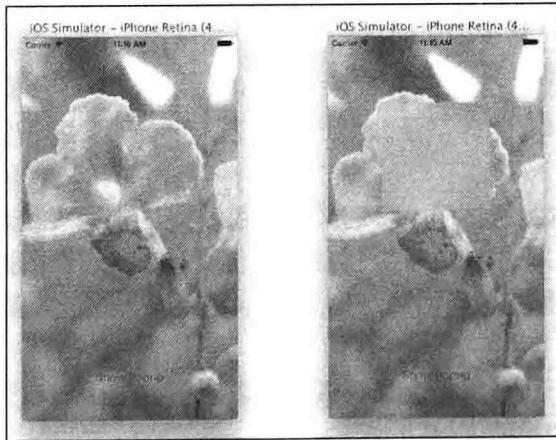


图 2-1 应用在 iOS 7 中运行时弹出视图前后的截屏

2.5 动态字体

在 iOS 7 中，用户可以设置文字大小。内置的邮件、日历以及其他大部分应用都会按照这个设置动态改变文字大小。注意增大文字大小并不总是能让字体的磅数变大。当文字大小设置的值可能会让小号字体无法辨认时，iOS 7 会自动加粗文字。为应用增加动态字体支持后，还可以在设置应用中动态改变文字大小。不过，iOS 7 的动态字体有一个缺点：在写作本章时，还不支持自定义字体。第 21 章会介绍动态字体。

2.6 自定义过渡效果

iOS 7 的另一个新特性是苹果在大部分内置应用中尽量减少了屏幕切换的使用。打开日历应用，点击一个日期，日期视图以自定义的过渡动画效果从月视图中出现。类似地，创建新事件时，改变事件的开始和结束时间也是用动画实现自定义过渡。注意，这个动画在之前版本的 iOS 中是导航控制器的默认推入页面操作动画。还有一个例子是照片应用中的照片选项卡，万年不变的相机胶卷被一个使用自定义过渡在年度视图、精选视图、时刻视图和单张照片视图之间切换的集合视图替代了。iOS 7 新的 UI 范式强调让用户知道自己在哪里，而不是让他们迷失在推入的无数视图控制器中。大部分情况下，这些动画都用自定义过渡实现，如图 2-2 所示。



图 2-2 日历应用的屏幕，显示使用自定义过渡的屏幕

为 iOS 7 设计应用时，要认真考虑使用自定义过渡是否对于让用户知道自己在哪儿有所帮助。iOS 7 SDK 增加了接口，能毫无困难地实现这些动画。

自定义过渡类型

iOS 7 的 SDK 支持两类自定义过渡：自定义视图控制器过渡和交互式视图控制器过渡。自定义视图控制器过渡之前也能实现，要用到故事板和自定义联线。交互式视图控制器过渡则让用户利用手势（通常是拖动）控制过渡的过程（从开始到结束）。因此当用户拖动或快速滑动手指时，会从一个视图控制器过渡到另一个。

当过渡是时间的函数时，通常是自定义视图控制器过渡，是手势识别器的参数或类似事件的函数时，通常是交互式视图控制器过渡。

比如，可以认为导航控制器的推入过渡（就像 iOS 6 那样）是一种自定义视图控制器过渡，而 `UIPageViewController` 则是交互式视图控制器过渡。使用 `UIPageViewController` 在视图之间翻页时，过渡跟时间无关，页面的过渡跟着手指的移动走，所以这是交互式视图控制器过渡。而 `UINavigationController` 过渡（在 iOS 6 中）在一段时间内发生，所以这类过渡是自定义视图控制器过渡。

iOS 7 SDK 允许开发者自定义几乎任何类型的过渡：视图控制器的 presentation 和 dismissal、`UINavigationController` 的推入和弹出过渡、`UITabBarController` 的过渡（`UITabBarController` 默认不对视图控制器做动画），甚至是集合视图的布局变化过渡。

自定义视图控制器过渡比交互式视图控制器过渡更易使用。本章会展示如何创建自定义视图控制器过渡。交互式视图控制器过渡相对较难使用，第 20 章会介绍这个主题。

2.7 把应用过渡（迁移）到 iOS 7

到目前为止，本章介绍了 iOS 7 中引入的主要 UI 范式并实现了其中一种：模糊效果。在本节中，我们会展示如何把应用从 iOS 6 过渡到 iOS 7。

你已经知道了，iOS 7 的 UI 和之前的版本大不相同。因此，你得了解大量的 API 变化。可能的话，完全放弃对 iOS 6 的支持，因为 iOS 7 的升级率要比 iOS 6 高得多。不过，如果业务需要支持 iOS 6，继续阅读下一节，我们会展示如何在不牺牲 iOS 6 的情况下支持 iOS 7。

2.7.1 UIKit变化

在 iOS 7 中，几乎所有的 UI 元素都发生了变化。按钮没有边框，开关和分段控件变小了，滑块和进度条变细了。如果使用自动布局，大部分变化都不会对你有什么影响。如果没有使用自动布局，那么是时候把 nib 文件升级到自动布局了。如果不使用自动布局，那么在 3.5 寸和 4 寸设备上支持 iOS 6 和 iOS 7 就需要写大量的布局代码。

如果你之前一直不想使用自动布局，那么现在必须开始学习了。Xcode 5 中的自动布局要比 Xcode 4.x 中的好用得多，本书第 6 章会更为详细地讲解自动布局。

2.7.2 自定义设计

一般来说大部分应用都会用自定义设计图来做应用皮肤，开发者需要仔细考虑如何设计。原因是 iOS 7 的设计是非常“平”的，视觉拟物削弱了很多。这么做是因为在 iPhone 4 之前，几乎所有的屏幕（手机/PC/Mac）像素密度都介于 70 ppi ~ 160 ppi（每英寸像素数）。iPhone 4 的视网膜屏将像素密度增加到了 320 ppi。这个像素密度接近健康人眼所能接受的最高程度，再多就是人眼接受不到的多余数据了。这也是印刷业采用 300 ppi 的原因。

印刷业不需要模拟抛光发亮的按钮、不规则边框、渐变或者艺术字体。你什么时候见过广告牌或者杂志封面的图上加上人造抛光或者发亮的效果？没有。为什么？因为不需要。（有些杂志有光泽，但那是真的光泽。）事实上，软件设计师利用抛光、发亮按钮来让他们的设计在（当时的）低分辨率（70 ppi ~ 160 ppi）屏幕上看起来比较好看。

而 iPhone 4 屏幕的像素密度达到了接近高印刷质量的杂志的程度，为 UI 添加光泽和发光效果就没有必要了。这就是 Twitterrific 5、LetterPress 和 Clear（代办事项应用）甚至在苹果之前就大量减少了界面效果的原因。Windows Phone 7 界面从第一天开始就没有抛光、发亮或者阴影。

今天，除了 iPad mini，苹果在 2011 年和 2012 年间发布的几乎所有 iOS 设备都采用了视网膜屏。扁平化设计是未来。给 UI 设计光泽在 iOS 7 中看起来就过时了。

如果用了抛光或是发亮按钮，你就得重新设计 UI 了。问问自己，这个 UI 在杂志上打印出来会是什么样子的。你得像印刷业的设计师那样考虑问题，而不是软件业的设计师。

2.7.3 支持iOS 6

写向后兼容 iOS 6 的代码要比以前难，因为 iOS 7 是其诞生以来的一次大升级。如果你的应用需要支持 iOS 6，首先让 iOS 6 的应用跟 iOS 7 的外观类似，这意味着并非要在 iOS 7 中用扁平按钮却在

iOS 6 中用抛光按钮，而是把 iOS 6 的按钮也变扁平。首先改变 iOS 6 版本的设计，使其看起来像 iOS 7 版本。对结果满意后，再添加 iOS 7 独有的特性。

1. 应用图标

iOS 7 使用的图标大小和 iOS 6 不同，对于 iPhone 应用是 120×120，对于 iPad 应用是 152×152。你得为应用更换这些略大的图标，避免使用抛光或者发亮，“Icon already includes gloss effects (UIPrerenderedIcon)” 这个设置在 iOS 7 下不起作用，可能最终会废弃。

2. 启动图

启动图现在应该是 480 点高（对于 iPhone 5 来说是 568）。iOS 7 中启动图会渲染在状态栏下面，如果你用的启动图太短，屏幕底部会出现黑边。

3. 状态栏

iOS 7 中的状态栏是半透明的，会将其背后的内容模糊。在 iOS 6 版本的应用中，应该用抛光不那么多的设计图（甚至完全没有光泽）确保状态栏是扁平的。在 iOS 7 中，视图控制器会延伸到全屏，位于状态栏下面。在 iOS 6 版本的应用中，可以把状态栏的样式改为 `UIStatusBarStyleBlack-Translucent` 来模拟这种行为。

4. 导航栏

iOS 7 中的导航栏默认也是半透明的，而且没有阴影，底部边界倒是有一条细线。可以考虑为 iOS 6 版本的应用添加类似的效果来取代阴影。

在 iOS 6 中，设置 `tintColor` 会改变导航栏的颜色，要在 iOS 7 中得到同样的效果，需要设置 `barTintColor`。导航栏上的返回按钮也变了，以前是带边框的按钮，现在则是箭头和文字。记住，就算是默认按钮也不带边框了，而且 `UIButtonTypeRoundedRect` 也废弃了。在 iOS 6 版本的应用中，可以用导航栏的外观代理协议（`appearance proxy protocol`）来自定义返回按钮，并为返回按钮设置类似于 iOS 7 版本的图片。

导航栏的背景图通常是 320×44，有时候开发者可能会用略高的带阴影的导航栏。在 iOS 7 中，导航栏的背景图延伸到了状态栏下方，而不是像 iOS 6 那样在视图控制器上方。避免使用高于 44 点的导航栏背景图。你可能得重新设计背景图并且考虑用别的办法添加阴影。iOS 6 为 `UINavigationBar` 引入了 `shadowImage` 属性，可以用这个属性设置导航栏阴影。

5. 工具栏

工具栏也是半透明的，不过更重要的是工具栏上的按钮没有边框。如果按钮不多于 3 个，可以考虑用文本按钮而不是图片按钮。比如 iOS 7 的音乐应用中的正在播放页面用文本按钮表示重复、随机和创建（如图 2-3 所示）。



图 2-3 iOS 7 中控制按钮的截图

6. 视图控制器

在 iOS 7 中，所有视图控制器都是全屏布局的。可以在 iOS 7 中支持新布局，在 iOS 6 中保留旧布局，但是应用看起来会有点过时。要拥抱变化，在 iOS 6 版本中用全屏布局。将视图控制器的 `wants-`

`FullscreenLayout` 属性设置成 YES 就可以在 iOS 6 中实现这一点。注意，这个属性在 iOS 7 中废弃了，而且在 iOS 7 中将这个属性设置成 NO 的行为是未定义的。

在 iOS 7 中也可以使用不透明的条栏，这可以通过 Interface Builder 的选项控制视图在半透明/不透明条栏下方的样子，如图 2-4 所示。

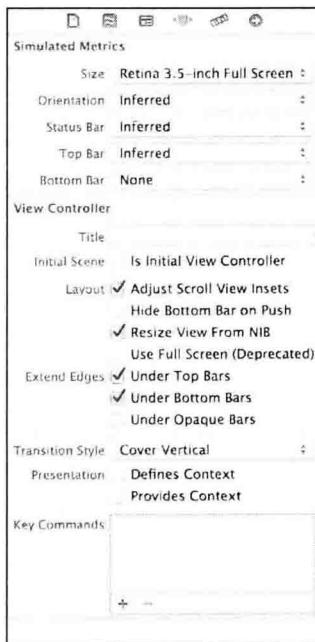


图 2-4 Interface Builder 中将视图延伸到半透明条栏下方的选项

Interface Builder 的 nib 文件中还有 Top Layout Guide 和 Bottom Layout Guide（只有打开自动布局才有），也可以用这些引导点设置约束条件。所以开发者可以让按钮总是离 Top Layout Guide 50 个点。

7. 表格视图控制器

表格视图控制器，尤其是分组表格视图样式，不再有内嵌效果了。分割线变成了内嵌的，从图片开始，到屏幕边缘结束。单元分割线变细了，颜色也变浅了。

section 头颜色也变浅了，而且是纯色，不再是渐变色了。在 iOS 7 版本的应用中可以给 `sectionIndexBackgroundColor` 设置值来改变颜色。在 iOS 6 中，只能在 `tableView:viewForHeaderInSection:` 委托方法中返回外观和 iOS 7 版本类似的视图。

还有一个重要的变化是选择样式。iOS 6 提供了两种样式：蓝色和灰色。用内置样式时，文本的前景色会从黑色变成白色。在 iOS 7 中不会这样，高亮色只是一种浅灰色，覆盖 `UITableViewCell` 子类的 `setSelected:animated:` 和 `setHighlighted:animated:` 可以模拟这种行为。

默认的滑动删除手势以前是从左向右滑，现在是从右向左滑。如果开发者正考虑在表格视图单元中用从右向左滑的手势来显示菜单或做一个动作，就得考虑用别的方法来实现了。

8. 拖动手势

iOS 7 默认在所有应用中使用两种拖动手势，第一种是 `UIScreenEdgePanGestureRecognizer`，

在导航控制器中，从屏幕边缘滑动可以让用户回到上一个视图。这个行为是默认的。如果你在使用像 Facebook 的旧版应用那样的汉堡菜单（俗称侧滑菜单或者侧滑面板），就得考虑关掉显示菜单的手势。事实上，随着 iOS 7 的发布，Facebook 完全抛弃了汉堡菜单，采用了选项卡栏。第二种手势是从屏幕底部拖动的手势，可以显示控制中心。如果你的应用使用了类似的手势来调用某个功能，就得重新设计界面了。

9. 警告框和操作列表

警告框和操作列表总是使用系统的默认样式，除非开发者创建自己的。如果有自定义的警告框，可以考虑加入 `UIMotionEffect` 以便让其看起来像是浮在视图控制器上方。在 iOS 6 中建立层次概念要靠阴影，而在 iOS 7 中则用运动效果。在 iOS 6 中模拟运动效果很难。除非愿意花费时间和精力，否则就在 iOS 6 中保留 UI 元素的阴影吧。

2.8 小结

本章介绍了新的 UI 范式，还有新设计背后的深层次原因，并且深入讲解了 iOS 7 中不同的 UI 相关的技术。最后，我们介绍了如何将应用迁移到 iOS 7，并且（可选地）维持向后兼容性。要做到写出在 iOS 6 和 iOS 7 上都能工作的代码并且外观在两个系统上看起来都很好是很难的（至少是相对而言）。如果业务允许你投入时间和资源，那就做吧。否则，把精力集中在 iOS 7 每个可能的功能上，让应用脱颖而出。

iOS 7 为开发者提供了在 App Store 大获成功的机遇，举几个例子：iOS 7 专用的待办事项应用，iOS 7 专用的 Twitter 客户端，iOS 7 专用的日历应用。理解了 iOS 7 的 UI 范式，是时候大干一把了。

2.9 扩展阅读

1. 苹果文档

下面的文档位于 iOS Developer Library (<https://developer.apple.com/library/ios/navigation/index.html>) 中，通过 Xcode Documentation and API Reference 也可以找到。

- *What's New in iOS 7*

2. WWDC 讲座

- WWDC 2013, “Session 226: Implementing Engaging UI on iOS”
- WWDC 2013, “Session 218: Custom Transitions Using View Controllers”
- WWDC 2013, “Session 201: Building User Interfaces for iOS 7”
- WWDC 2013, “Session 208: What's New in iOS User Interface Design”
- WWDC 2013, “Session 225: Best Practices for Great iOS UI Design”

第二部分

充分利用日常工具

本部分内容

- 第3章 你可能不知道的
- 第4章 故事板及自定义切换效果
- 第5章 掌握集合视图
- 第6章 使用自动布局
- 第7章 更完善的自定义绘图
- 第8章 Core Animation
- 第9章 多任务

你可能不知道的

如果你在阅读本书，说明你可能已经掌握好 iOS 的基础了，但是有一些小功能和实践，很多开发者可能经过多年的开发还不熟悉。本章将介绍一部分技巧和窍门，它们都足够重要，但是又比较小，不足以独立成章，还有一些最佳实践，能让代码更健壮、更易维护。

3.1 命名最佳实践

纵观整个 iOS，命名约定非常重要。在下面几节中，你将了解如何正确命名不同的东西，以及命名如此重要的原因。

3.1.1 自动变量

Cocoa 是动态类型语言，开发者很容易就会对手头的对象类型感到迷糊。容器（数组、字典等）没有关联的类型，因此开发者经常会不小心写出下面这样的代码：

```
NSArray *dates = @[@"1/1/2000"];
NSDate *firstDate = [dates firstObject];
```

这段代码能编译，而且不会有警告，但是在使用 `firstDate` 时，很可能就会因为未知选择器异常而崩溃。这个错误的根源在于把一个字符串数组叫做 `dates`。这个数组应该叫 `dateStrings`，否则就应该容纳 `NSDate` 对象。这类谨慎的命名可以减少很多让人头疼的问题。

3.1.2 方法

方法的名字应该清晰地表明它接受的参数类型和返回值的类型。比如，下面这个方法就会让人迷惑：

```
- (void)add; //会让人迷惑
```

看起来 `add` 应该有一个参数，但实际上没有。它只是加上某个默认对象？

下面这样命名就清晰很多：

```
- (void)addEmptyRecord;
- (void)addRecord:(Record *)record;
```

很明显 `addRecord:` 接受一个 `Record` 参数，如果可能产生误解，对象的类型应该和名字匹配。比如，下面这个例子就是常见的错误：

```
- (void)setURL:(NSString *)URL; //错误
```

这里有错误是因为命名为 `setURL:` 的方法应该接受 `NSURL` 作为参数，而不是 `NSString`。如果你需要字符串，就应该明确指出：

```
- (void)setURLString:(NSString *)string;  
- (void)setURL:(NSURL *)URL;
```

不要滥用这个规则，对于某些类型很显然的变量就不要添加类型信息了。属性命名为 `name` 比 `nameString` 好，只要你的系统中没有 `Name` 类就会让读代码的人困惑。

方法命名也有跟内存管理和键值编码（KVC，在第 22 章详细讨论）相关的规则。尽管自动引用计数（ARC）的出现使得一些规则不那么重要了，但是命名不正确的方法在 ARC 代码和非 ARC 代码（包括苹果框架中的非 ARC 代码）交互时可能会产生很难查的 bug。

方法的命名应该用驼峰命名法，第一个字母小写。

如果方法用 `alloc`、`new`、`copy` 或者 `mutableCopy` 开头，那么调用者拥有返回的对象（也就是说，对象的引用计数会净增加 1，调用者必须负责释放）。有一个属性命名为 `newRecord` 就可能导致问题，改名为 `nextRecord` 或类似的东西。

`get` 开头的方法应该返回一个值的引用，比如：

```
- (void)getPerson:(Person **)person;
```

不要用 `get` 前缀作为属性访问方法的一部分，`name` 属性的获取应该是 `-name`。

3.2 属性和实例变量最佳实践

属性应该表示对象的状态，获取方法不应该有外部副作用（可以有缓存等内部副作用，但是对于调用者来说应该不可见），一般来说，获取方法的调用应该高效，当然也不应该阻塞调用者。

避免直接访问实例变量，要用存取器，一会儿我会讲到例外情况，但是首先讲讲为什么要用存取器。

在有 ARC 之前，造成 bug 的一大常见因素就是直接访问实例变量。开发者会忘记正确保留或释放实例变量，于是程序会产生内存泄漏或崩溃。因为 ARC 会自动管理保留和释放，有些开发者会认为这条规则不重要了，但是还有其他原因要用存取器。

- **键值观察**——使用存取器最重要的原因可能是属性会被观察，如果不用存取器，开发者可能需要在每次修改属性的实例变量时调用 `willChangeValueForKey:` 和 `didChangeValueForKey:`。存取器会自动调用这些方法。
- **副作用**——开发者自身或者某个子类会在设置方法中引入副作用，可能是发出通知或在 `NSUndoManager` 中注册了事件，除非必要，不要绕开这些副作用。类似地，开发者或者子类可能会在获取方法中增加缓存，而直接访问实例变量会绕开缓存。
- **惰性初始化**——如果属性是惰性初始化的，必须用存取器使其正确释放。
- **锁**——如果为属性引入锁来管理多线程代码，直接访问实例变量会破坏锁并可能使程序崩溃。
- **一致性**——有的人可能会说，只有我们在确定基于前面提到的原因而需要用存取器时才用存取器。但是这样会让代码难以维护。更好的做法是怀疑每次实例变量的直接访问并解释原因，而不是要一直记得哪些实例变量需要用存取器哪些不需要。这样会让代码易于阅读、评审和维护。在 Objective-C 中，存取器（尤其是自动合成的存取器）经过了高度优化，带来的好处对得起那点开销。

话虽这么说，还是有些地方不应该用存取器。

- 在存取器内部。显然，在存取器内部不能用存取器本身。一般来说，也不要在获取方法中使用设置方法（有些情况下会发生死循环）。存取器应该访问自己的实例变量。
- `dealloc`。ARC 极大减少了需要写 `dealloc` 的地方，不过有时候还是会用到。最好不要在 `dealloc` 中调用外部对象。对象可能处于不一致的状态，而且观察者收到属性变化的多个通知也可能会迷惑，而实际上真实含义是对象正在被销毁。
- 初始化。类似于 `dealloc`，在初始化过程中，对象可能处于不一致状态，在这个阶段一般不应该触发通知或者有别的副作用。这里通常也是初始化只读变量的地方，比如 `NSMutableArray`。这样能避免把属性声明为 `readwrite`，从而只有开发者本人能进行初始化。

Objective-C 对存取器做了高度优化，为可维护性和灵活性提供了重要的特性。通过存取器访问属性，即便这个属性是开发者自己的，这可以作为一条一般规则。

3.3 分类

分类允许在运行时为已有的类添加方法。任何类，就算是苹果提供的 Cocoa 类，也可以用分类扩展，而且这些方法对于类的所有对象都可用。声明分类很简单，看起来就像声明类接口，只是分类的名字在括号中：

```
@interface NSMutableString (PTLCapitalize)
- (void)ptl_capitalize;
@end
```

`PTLCapitalize` 是分类的名字，注意，这里没有声明实例变量。分类无法声明实例变量，也无法合成属性（其实本质是一样的）。3.4 节会讲到如何添加分类数据。分类可以声明属性是因为属性声明只是方法声明的一种方式，只是不能合成属性，因为那样会创建实例变量。`PTLCapitalize` 分类不需要 `ptl_capitalize` 方法在任何地方有实际的实现。如果没有实现 `ptl_capitalize`，而调用者试图调用它的话，系统会抛出异常。这里编译器不会保护你。如果要实现的话，按照惯例看起来可能是这样的：

```
@implementation NSMutableString (PTLCapitalize)
- (void)ptl_capitalize {
    [self setString:[self capitalizedString]];
}
@end
```

说“按照惯例”是因为这个方法不一定要在分类实现中定义，分类实现也不一定要用和分类接口相同的名字。不过，如果你提供了一个名为 `PTLCapitalize` 的`@implementation` 语法块，就必须实现名为 `PTLCapitalize` 的`@interface` 语法块中的方法。

从技术上讲，分类可以覆盖方法，但是这么做很危险，我们不推荐。如果两个分类实现了同样的方法，实际用哪一个是未定义的。如果一个类因为维护的原因被分成了分类，开发者编写的覆盖方法会变成未定义行为，这是一类追踪起来会让人发疯的 bug。此外，用这个特性会让代码难以理解。分类覆盖方法也无法调用原方法。要调试的话，推荐方法混写，第 24 章会讲到。

为了避免碰撞，分类方法前面应该加上前缀，后面跟上下划线，就像 `ptl_capitalize` 例子中那样。Cocoa 一般不会这样嵌入下划线，但是在这种情况下，这么做要比其他写法清晰。

分类的一个不错的用法是为已有的类提供实用方法。要做到这一点，推荐用原类名 + 扩展名来命名头文件和实现文件。比方说，可以在 `NSDate` 上创建一个 `PTLExtensions`：

NSDate+PTLExtensions.h

```
@interface NSDate (PTLExtensions)
- (NSTimeInterval)ptl_timeIntervalUntilNow;
@end
```

NSDate+PTLExtensions.m

```
@implementation NSDate (PTLExtensions)
- (NSTimeInterval)ptl_timeIntervalUntilNow {
    return -[self timeIntervalSinceNow];
}
@end
```

3

如果只有几个实用方法，把它们放在一个诸如名为 `PTLExtensions`（或者你的代码所用的任何前缀）的单个分类中比较方便。这样做可以很容易地把你最喜欢的扩展放在每个工程中。当然，这会造成代码膨胀，所以在决定要把多少代码扔进“实用”分类时要小心。Objective-C 无法像 C 或 C++ 那样高效地做无用代码删除。

+load

分类是在运行时附着到类上的。定义分类的库可能是动态加载的，所以分类可能会在很晚的时候被添加进来。（尽管开发者无法自己在 iOS 中写动态加载库，但是系统库（包括分类）是动态加载的。）Objective-C 提供了一个叫做 `+load` 的钩子方法，当分类第一次附着时会运行。就跟 `+initialize` 一样，开发者可以用这个方法实现分类相关的设置，比如初始化静态变量。无法在分类中安全地使用 `+initialize` 是因为主类可能已经实现了。如果多个分类都实现了 `+initialize`，哪个会运行是未定义的。

希望你能问出这个明显的问题：“如果分类不能用 `+initialize` 方法，因为这样可能跟其他分类冲突，那么多个分类都实现 `+load` 会怎么样？”这是 Objective-C 运行时少有的神奇部分。`+load` 方法是运行时的特殊情况，每个分类都可以实现，而且每个实现都会运行。运行顺序没有保证，你也不应该手动调用 `+load`。

无论分类是静态加载还是动态加载的，`+load` 都会被调用。调用发生在分类被添加到运行时环境中时，通常是在程序启动而 `main` 被调用前，不过也可能晚得多。

类也可以有自己的 `+load` 方法（不在分类中定义），而且在类被添加到运行时环境中时会运行。这种方法很少用到，除非动态添加类。

不需要像 `+initialize` 方法那样保护 `+load` 方法不被多次调用，系统只会给实际实现了 `+load` 的类发送 `+load` 消息，因此不会像 `+initialize` 那样收到来自子类的调用。每个 `+load` 方法只会被调用一次。不应该调用 `[super load]`。

3.4 关联引用

关联引用允许开发者为任何对象附着键值数据。这种能力有很多用法，一种常见用法是让分类为属性添加方法。

考虑 Person 类这个例子，假设你要用分类添加一个新属性，叫做 `emailAddress`。可能其他程序也用到了 Person，有时候需要电子邮箱地址，有时候不需要，分类就是很好的解决方案，可以避免在不需要的时候开销。或者 Person 不是你的，而维护者没有为你添加这个属性。不管哪种情况，你要怎么解决这个问题呢？首先，这里有基本的 Person 类：

```
@interface Person : NSObject
@property (nonatomic, readwrite, copy) NSString *name;
@end
```

```
@implementation Person
@end
```

现在在分类中用关联引用添加一个新属性 `emailAddress`：

```
#import <objc/runtime.h>
@interface Person (EmailAddress)
@property (nonatomic, readwrite, copy) NSString *emailAddress;
@end

@implementation Person (EmailAddress)

static char emailAddressKey;

- (NSString *)emailAddress {
    return objc_getAssociatedObject(self, &emailAddressKey);
}

- (void)setEmailAddress:(NSString *)emailAddress {
    objc_setAssociatedObject(self, &emailAddressKey,
                           emailAddress,
                           OBJC_ASSOCIATION_COPY);
}
@end
```

注意，关联引用基于键的内存地址，而不是值的。`emailAddressKey` 中存着什么并不重要，只要是唯一的不变的地址就可以。这也是一般会用未赋值的 `static char` 变量作为键的原因。

关联引用有良好的内存管理，能根据传递给 `objc_setAssociatedObject` 的参数正确处理 `copy`、`assign` 和 `retain` 等语义。当相关对象被销毁时关联引用会被释放。这个事实意味着可以用关联引用来追踪另一个对象何时被销毁。比如：

```
const char kWatcherKey;

@interface Watcher : NSObject
@end

#import <objc/runtime.h>
```

```

@implementation Watcher
- (void)dealloc {
    NSLog(@"HEY! The thing I was watching is going away!");
}
@end
...
NSObject *something = [NSObject new];
objc_setAssociatedObject(something, &kWatcherKey, [Watcher new],
                         OBJC_ASSOCIATION_RETAIN);

```

这种技术对调试很有用，不过也可以用做非调试目的，比如执行清理工作。

关联引用是给警告框或控件附着相关对象的好办法。比如说，你可以给警告框附着一个“表示对象”，如下代码所示。这段代码在本章的示例代码中有。

ViewController.m (AssocRef)

```

id interestingObject = ...;
UIAlertView *alert = [[UIAlertView alloc]
                      initWithTitle:@"Alert" message:nil
                      delegate:self
                      cancelButtonTitle:@"OK"
                      otherButtonTitles:nil];
objc_setAssociatedObject(alert, &kRepresentedObject,
                        interestingObject,
                        OBJC_ASSOCIATION_RETAIN_NONATOMIC);
[alert show];

```

现在，如果警告框被关闭，你就能知道原因了：

```

- (void)alertView:(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)buttonIndex {
    UIButton *sender = objc_getAssociatedObject(alertView,
                                                &kRepresentedObject);
    self.buttonLabel.text = [[sender titleLabel] text];
}

```

很多程序用调用者的实例变量处理这种任务，但是关联引用清晰得多，也简单得多。对熟悉 Mac 开发的人来说，这段代码类似于 `representedObject`，但是更灵活。

关联引用的一个局限(或者其他任何通过分类添加数据的方法)是无法集成 `encodeWithCoder:`，所以很难通过分类序列化对象。

3.5 弱引用容器

常见的 Cocoa 容器有 `NSArray`、`NSSet` 和 `NSDictionary`，对大部分使用场景来说都很好，但是在某些情况下并不适用。`NSArray` 和 `NSSet` 会保留保存其中的对象，`NSDictionary` 不光保留值，还要复制键。通常这些行为就是你所需要的，但是对某些问题来说这么做是和你对着干。幸好从 iOS 6 开始有了新的容器类：`NSMutableArray`、`NSTable` 和 `NSMutableDictionary`，在苹果文档中统称为指针容器类 (pointer collection class)，有时候配置为使用 `NSPointerFunctions` 类。

`NSMutableArray` 类似于 `NSArray`，`NSTable` 类似于 `NSSet`，`NSMutableDictionary` 类似于

`NSDictionary`。这些新容器类都可以配置为持有弱引用、非对象的指针或者其他罕见情形。`NSPointerArray`还有一个好处是可以存储 NULL 值，这也是 `NSArray` 的常见问题。

苹果关于指针容器类的文档通常会参考垃圾回收，因为这些类最初是为 10.5 的垃圾回收开发的。现在这些类兼容 ARC 弱引用。这一点在主类参考文档中通常写得没那么清楚，但是 `NSPointerFunctions` 类的参考文档中有说明。

指针容器类支持扩展，可以配置为使用 `NSPointerFunctions` 对象，但是大部分情况下将 `NSPointerFunctionsOptions` 标志位传递给 `-initWithOptions:` 会更简单。大部分情况下，比如 `+weakObjectsPointerArray`，有自己的快捷构造函数。

更多信息可以查阅对应的类参考文档，以及容器编程主题，还有 NSHipster 的“`NSTHashTable & NSMapTable`”这篇文章 (nshipster.com)。

3.6 NSCache

使用弱引用容器最常见的理由是实现缓存。但是很多时候可以用 Foundation 的缓存对象 `NSCache` 代替。多数情况下，其用法就跟 `NSDictionary` 一样，可以调用 `objectForKey:`、`setObject:forKey:` 和 `removeObjectForKey:`。

`NSCache` 的一些特性被低估了，比如其多线程安全性。开发者可以在任何线程上不加锁地修改 `NSCache`。`NSCache` 还被设计为能与符合 `<NSDiscardableContent>` 协议的对象整合。`<NSDiscardableContent>` 最常见的类型是 `NSPurgeableData`。通过调用 `beginContentAccess` 和 `endContentAccess`，开发者能控制何时丢弃对象是安全的。这不仅能在应用运行的时候提供自动缓存管理，甚至在应用暂停时也有用。通常，在内存吃紧且内存警告没有释放足够内存的情况下，iOS 开始杀掉暂停的后台应用，这时应用不会收到委托消息，会被杀掉。但如果用了 `NSPurgeableData`，iOS 会替你释放内存，即使应用处于暂停状态。

更多信息可以查阅 Xcode 文档中 `NSCache`、`<NSDiscardableContent>` 和 `NSPurgeableData` 参考文档。

3.7 NSURLComponents

有时候苹果会很低调地添加有趣的类。在 iOS 7 中，苹果添加了 `NSURLComponents`，它没有类参考文档。在 iOS 7 发布说明的“What’s New in iOS 7”一节中有提到，但是你得阅读 `NSURL.h` 才能看到其文档。

`NSURLComponents` 可以很方便地把 URL 分成几个部分，比如：

```
NSString *URLString =
@"http://en.wikipedia.org/wiki/Special:Search?search=ios";
NSURLComponents *components = [NSURLComponents
componentsWithString:URLString];
NSString *host = components.host;
```

也可以用 `NSURLComponents` 创建或修改 URL:

```
components.host = @"es.wikipedia.org";
NSURL *esURL = [components URL];
```

在 iOS 7 中, `NSURL.h` 添加了一些有用的分类来处理 URL。比方说, 可以用 `[NSCharacterSet URLPathAllowedCharacterSet]` 得到允许在路径中出现的字符集合。`NSURL.h` 还添加了 `[NSString stringByAddingPercentEncodingWithAllowedCharacters:]`, 允许开发者控制使用百分号编码的字符, 而以前只能用 Core Foundation 的 `CFURLCreateStringByReplacingPercentEscapes` 做这件事。

在 `NSURL.h` 中搜索 `_0` 可以找到所有的新方法及其文档。

3.8 CFStringTransform

`CFStringTransform` 是那种一旦你发现它, 就无法相信以前竟然不知道其存在的函数。它可以吧字符串变得容易标准化(normalization)、索引和搜索。比如说, 它可以使用 `kCFStringTransformStripCombiningMarks` 选项删除重音符号:

```
CFMutableStringRef string = CFStringCreateMutableCopy(NULL, 0,
                                                    CFSTR("Schläger"));
CFStringTransform(string, NULL, kCFStringTransformStripCombiningMarks, false);
... => string is now "Schlager"
CFRelease(string);
```

`CFStringTransform` 更为强大的功能是处理非拉丁书写系统, 比如阿拉伯文或中文。它能把很多书写系统转换为拉丁字母, 使得标准化变得很容易。比如, 可以像这样把汉字转换为拉丁字母:

```
CFMutableStringRef string = CFStringCreateMutableCopy(NULL, 0,
                                                    CFSTR("你好"));
CFStringTransform(string, NULL, kCFStringTransformToLatin, false);
... => string is now "ní hào"
CFStringTransform(string, NULL, kCFStringTransformStripCombiningMarks, false);
... => string is now "ni hao"
CFRelease(string);
```

注意这里用到的选项就是 `kCFStringTransformToLatin`, 不需要源语言。通过它可以转换几乎任何字符串, 而不需要知道其语言。`CFStringTransform` 还能把拉丁字母变成其他的书写系统, 比如阿拉伯文、韩文、希伯来文和泰文。

日文中的汉字

汉字总会被音译为普通话发音, 即使出现在其他书写系统中。对日文来说比较棘手, 因为日文字符串中可能出现汉字。比如“白い月”这个日文短语的三个字符中, 第一个和最后一个会被音译为普通话发音(bái 和 yuè), 而中间的字符会被音译为日语发音(i), 这样会产生无意义的字符串 bái i yuè。

尽管 `CFStringTransform` 能够处理平假名和片假名, 但是无法处理日文中的汉字。如果需要音译复杂的日文文本, 可以参考 00StevenG 的“`NSString Japanese`”(<https://github.com/00StevenG/NSString-Japanese>)一文, 它能处理汉字、罗马字还有平假名和片假名。

NSString-Japanese 基于 CFStringTokenizer，使用比较复杂，但是转换过程中能更加智能地处理语言。

更多信息见developer.apple.com上的 CFMutableString 和 CFStringTokenizer 的参考文档。

3.9instancetype

Objective-C 一直以来在子类继承上有些小问题。考虑下面的情形：

```
@interface Foo : NSObject
+ (Foo *)fooWithInt:(int)x;
@end

@interface SpecialFoo : Foo
@end

...
SpecialFoo *sf = [SpecialFoo fooWithInt:1];
```

这段代码会产生警告：Incompatible pointer types initializing ‘SpecialFoo *’ with an expression of type‘Foo *’。问题在于 fooWithInt 返回的是 Foo 对象，编译器不知道真正返回的其实是更具体的类 (SpecialFoo)，这是常见问题。考虑 [NSMutableArray array]，如果返回的是 NSArray，编译器不可能让你把它赋值给子类 (NSMutableArray) 而不产生警告。

这个问题有几种解决方案。首先，可以尝试像这样重载 fooWithInt:方法：

```
@interface SpecialFoo : Foo
+ (SpecialFoo *)fooWithInt:(int)x;
@end

@implementation SpecialFoo
+ (SpecialFoo *)fooWithInt:(int)x {
    return (SpecialFoo *)[super fooWithInt:x];
}
```

这种方法有用，但是不方便。只是为了增加类型转换，你得覆盖很多方法。也可以让调用者这样做类型转换：

```
SpecialFoo *sf = (SpecialFoo *)[SpecialFoo fooWithInt:1];
```

这种方法对 SpecialFoo 来说是方便了，但是对调用者却不方便。添加大量的类型转换也会让类型检查失效，从而更容易出错。

最常见的解决方案是让返回值变成 id：

```
@interface Foo : NSObject
+ (id)fooWithInt:(int)x;
@end

@interface SpecialFoo : Foo
@end

...
SpecialFoo *sf = [SpecialFoo fooWithInt:1];
```

有关此电子图书的说明

本人由于一些便利条件，可以帮您提供各种中文电子图书资料，且质量均为清晰的 PDF 图片格式，质量要高于网上大量传播的一些超星 PDG 的图书。方便阅读和携带。只要图书不是太新，文学、法律、计算机、人文、经济、医学、工业、学术等方面 的图书，我都可以帮您找到电子版本。所以，当你想要看什么图书时，可以联系我。我的 QQ 是：**89039855**，大家可以在 QQ 上联系我。

此 PDF 文件为本人亲自制作，请各位爱书之人尊重个人劳动，敬请您不要修改此 PDF 文件。因为这些图书都是有版权的，请各位怜惜电子图书资源，不要随意传播，否则，这些资源更难以得到。