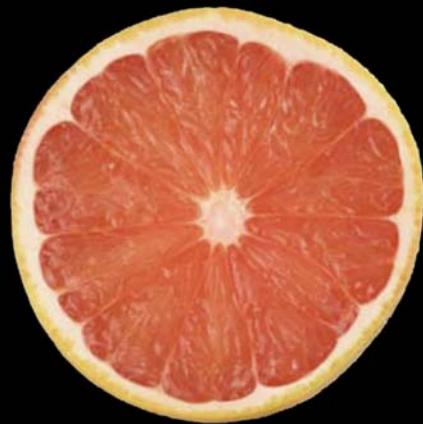


TURING 图灵程序设计丛书 移动开发系列

Apress*

Amazon
计算机
榜首图书

- 创造销售奇迹的最新经典著作
- 全面深入探索iPhone开发的无限可能
- 从这里，抢先拥抱软件开发的未来



Beginning iPhone Development
Exploring the iPhone SDK

iPhone开发基础教程

[美] Dave Mark
Jeff LaMarche 著
漆振 解巧云 孙文磊 等译

人民邮电出版社
POSTS & TELECOM PRESS

本书赞誉

- Amazon 计算机榜首图书
- 创造销售奇迹的最新经典著作
- 全面深入探索 iPhone 开发的无限可能
- 从这里，抢先拥抱软件开发的未来
- 国内最大、最专业iPhone开发社区[CocoaChina](#)强烈推荐

媒体评论

“Dave Mark一直是Mac编程图书作者中的佼佼者，而他现在又无可争议地成为了iPhone开发图书的王牌作者！本书是iPhone开发的权威指南，任何有意开始iPhone开发的人都应该阅读这本宝贵的参考指南。”

——Brian Greenstone（Pangea软件公司的总裁兼CEO）

“Trism游戏让我在2个月内收入25万美元，然后有无数人问我怎么开发iPhone应用，现在答案出现了！Dave和Jeff的书深入浅出、循序渐进而且示例丰富，堪称完美。它已经成了我的必备参考书，需要不时查阅。强烈推荐！”

——Steve Demeter（《连线》杂志“2008最佳iPhone应用”Trism游戏开发者）

内容简介

Apple公司的iPhone已经开创了移动平台新纪元！它与App Store的绝配也为全世界的程序员提供了一个施展才华的全新大舞台。只要有新奇的创意，你完全有可能像开发iShoot游戏的Ethan Nicholas（日收入2万多美元）和开发Trism游戏的Steve Demeter（月收入超过10万美元）那样，仅凭单枪匹马就赢得全球市场，成功创业，改变自己的人生。

本书由业界名家撰写，英文原版问世以后迅速登上Amazon计算机图书排行榜榜首并持续至今，总排名一度达到20名左右，创造了销售奇迹。而且，本书获得了读者的一致好评，已经被奉为经典。书中从到Apple网站注册账号，下载和安装免费iPhone SDK开始，清晰透彻地讲述了创建iPhone应用程序的全过程。在探讨基本概念和各个关键特性（iPhone界面元素、数据保存、SQLite、Quartz和OpenGL ES、手势支持、本地化、Core Location等）时，提供了丰富的实例。更难得的是，本书始终强调iPhone开发中的各种最佳实践，即使是有经验的开发人员，也会因此受益匪浅。

开卷阅读本书，进入iPhone开发的神奇世界吧，它将让你热血沸腾！

作者介绍

Dave Mark 深受爱戴的Apple技术开发专家，具有多年开发经验。他是许多Mac平台畅销书的作者，包括*Learn C on the Mac*、*Macintosh Programming Primer*系列以及*Ultimate Mac Programming*。可以通过www.davemark.com与他联系。

Jeff LaMarche 资深Apple平台专家，拥有多年企业级开发经验。他是*MacTech Magazine*和Apple公司开发人员网的专栏作家。

译者序

iPhone自从发布的那一天起就成为科技界的焦点。但一直为人诟病的是，其本身并不支持开源软件开发，也一直没有推出相应的第三方开发工具。而iPhone SDK的发布无疑解决了这一难题。开发人员可以使用iPhone SDK轻易地为iPhone和iPod Touch创建应用程序。其简单易学的操作方式和强大的功能为开发人员带来了超强的能力。

本书提供了关于iPhone SDK和iPhone开发的全面信息，对Objective-C编程语言、Xcode和Interface Builder开发工具进行了深入浅出的介绍，同时对iPhone开发的基本流程、原理和原则进行了详细和通俗的讲解。本书采用理论与实践相结合的方式，指导读者创建一系列应用程序，让读者能在实践中理解iPhone应用程序的运行方式和构建方式，掌握具体的iPhone特性，学会如何控制这些特性或与之交互。

全书共18章，分为3个部分。前4章介绍iPhone开发中的相关基本概念和开发人员所需的必备知识，并通过示例演示了一些标准的iPhone用户界面控件。第5章至第16章深入介绍如何开发各种高级iPhone特性，其中包括自动旋转、工具栏控制器、表视图、分层列表、应用程序设置、数据管理、绘图、手势输入、Core Location、加速计以及照相机和相片库。最后两章介绍如何将iPhone应用程序翻译为其他语言，从而让更多的用户接受并使用它，以及看完本书之后应该努力的方向。

本书覆盖面广、结构清晰，是一本有关iPhone开发的全新入门指南。它面向具备基本Objective-C知识的iPhone初、高级开发用户，不论你是经验丰富的开发人员，还是初涉编程领域的新手，都可以从本书中得到有用的信息。书中的示例通用性高，特别适合读者参考使用，这使本书成为广大读者的首选。

iPhone SDK是一个新兴的软件开发平台，但目前国内关于iPhone开发的资料非常有限。相信本书的出版可以为iPhone开发的发展起到推波助澜的作用。

本书由漆振、谢巧云、孙文磊等翻译，在翻译过程中得到了欧阳宇、盛海艳、杨越和张波的帮助，在此一并致谢。由于译者的知识水平有限，加之时间也比较仓促，文中难免会出现一些疏漏，恳请广大读者给予批评指正。

译者
2009年1月

前　　言

“从我开始使用Mac以来，我还没有看到过让我如此激动的编程平台。”最近我们经常听到这样的感言，坦白地说，我也有同感。iPhone是一种让人激动不已的出色技术，它将功能和乐趣完美地融合在一起。而程序员使用这种技术可以完成的工作也让人很激动！

这个世界的大门刚刚打开。花些时间浏览App Store，你会情不自禁地感动振奋。如果你并不负责设计自己的iPhone应用程序，那么为iPhone开发提供咨询也具有无限的商机。每个人好像都想把他们的产品导入该平台。我们的电话已经响个不停了。

如果你已经研究了几个月，偶尔访问一下我们的网站（<http://iphonedevbook.com>），并和我们打个招呼吧。请告诉我们有关你的项目的信息，我们很乐意倾听你的诉说。

Dave和Jeff

目录

第 1 章 欢迎来到 iPhone 的世界	1
1.1 关于本书	1
1.2 必要条件	1
1.3 必备知识	3
1.4 编写 iPhone 应用程序有何不同	4
1.4.1 只有一个正在运行的应用程序	4
1.4.2 只有一个窗口	4
1.4.3 受限访问	4
1.4.4 有限的响应时间	4
1.4.5 有限的屏幕大小	5
1.4.6 有限的系统资源	5
1.4.7 缺少 Cocoa 工具	5
1.4.8 新属性	5
1.4.9 与众不同的方法	6
1.5 本书内容	6
1.6 准备开始吧	7
第 2 章 创建基本项目	8
2.1 在 Xcode 中设置项目	8
2.2 Interface Builder 简介	12
2.2.1 nib 文件的构成	14
2.2.2 在视图中添加标签	15
2.3 iPhone 美化	17
2.4 小结	20
第 3 章 处理基本交互	21
3.1 模型—视图—控制器范型	21
3.2 创建项目	22
3.3 创建视图控制器	22
3.3.1 输出口	23
3.3.2 操作	23
3.3.3 将操作和输出口添加到视图控制器	24
3.3.4 将操作和输出口添加到实现文件	26
3.4 使用应用程序委托	30
3.5 编辑 MainWindow.xib	32
3.6 编辑 Button_FunViewController.xib	33
3.6.1 在 Interface Builder 中创建视图	33
3.6.2 连接所有元素	35
3.6.3 测试	37
3.7 小结	38
第 4 章 更丰富的用户界面	39

4.1	满是控件的屏幕	39
4.2	活动、静态和被动控件	41
4.3	创建应用程序	41
4.3.1	导入图像	41
4.3.2	实现图像视图和文本字段	42
4.3.3	添加图像视图	43
4.3.4	添加文本字段	46
4.3.5	设置第二个文本字段的属性	49
4.3.6	连接输出口	49
4.4	构建和运行	49
4.4.1	完成输入后关闭键盘	50
4.4.2	通过触摸背景关闭键盘	51
4.5	实现滑块和标签	52
4.5.1	确定输出口	52
4.5.2	确定操作	52
4.5.3	添加输出口和操作	52
4.5.4	添加滑块和标签	53
4.5.5	连接操作和输出口	54
4.6	实现开关和分段控件	55
4.6.1	确定输出口	55
4.6.2	确定操作	55
4.6.3	添加开关和分段控件	57
4.6.4	连接输出口	58
4.7	实现按钮、操作表和警报	59
4.7.1	将输出口及操作添加到控制器头文件	59
4.7.2	在 Interface Builder 中添加按钮	60
4.7.3	实现按钮的操作方法	60
4.8	显示操作表	61
4.9	美化按钮	63
4.9.1	viewDidLoad 方法	64
4.9.2	控件状态	65
4.9.3	可拉伸图像	65
4.10	小结	65
第 5 章	自动旋转和自动调整大小	67
5.1	使用自动调整属性处理旋转	68
5.1.1	指定旋转支持	68
5.1.2	使用自动调整属性设计界面	70
5.1.3	自动调整属性	70
5.1.4	设置按钮的自动调整属性	72
5.2	在旋转时重构视图	73
5.2.1	声明和连接输出口	74

5.2.2 在旋转时移动按钮	74
5.3 切换视图	77
5.3.1 确定输出口	78
5.3.2 确定动作	78
5.3.3 声明动作和输出口	79
5.3.4 设计两个视图	79
5.3.5 实现交换和动作	80
5.3.6 链接 Core Graphics 框架	83
5.4 小结	85
第6章 多视图应用程序	86
6.1 View Switcher 应用程序	88
6.2 多视图应用程序的体系结构	88
6.2.1 多视图控制器也是视图控制器	89
6.2.2 内容视图剖析	89
6.3 构建 View Switcher	89
6.3.1 创建视图控制器和 nib 文件	90
6.3.2 修改应用程序委托	92
6.3.3 SwitchViewController.h	93
6.3.4 修改 MainWindow.xib	93
6.3.5 编写 SwitchViewController.m	96
6.3.6 实现内容视图	99
6.4 制作转换动画	101
6.5 重构	103
6.6 小结	105
第7章 标签栏与选取器	106
7.1 Pickers 应用程序	106
7.2 委托和数据源	108
7.3 建立工具栏框架	108
7.3.1 创建文件	108
7.3.2 设置内容视图 nib	109
7.3.3 添加根视图控制器	109
7.4 实现日期选取器	113
7.5 实现单个组件选取器	116
7.5.1 声明输出口和操作	116
7.5.2 构建视图	116
7.5.3 将控制器实现为数据源和委托	117
7.6 实现多组件选取器	121
7.6.1 声明输出口和操作	121
7.6.2 构建视图	122
7.6.3 实现控制器	122
7.7 实现独立组件	125

7.8	使用自定义选取器创建简单游戏	132
7.8.1	编写控制器头文件	132
7.8.2	构建视图	133
7.8.3	添加图像资源	133
7.8.4	实现控制器	133
7.8.5	spin 方法	136
7.8.6	viewDidLoad 方法	137
7.8.7	最后的细节	139
7.8.8	链接 Audio Toolbox 框架	142
7.9	小结	143
第 8 章 表视图简介		144
8.1	表视图基础	144
8.2	实现一个简单的表	147
8.2.1	设计视图	147
8.2.2	编写控制器	148
8.3	添加一个图像	151
8.4	附加配置	151
8.4.1	设置缩进级别	152
8.4.2	处理行的选择	152
8.4.3	更改字体大小和行高	153
8.4.4	委托还能做什么？	155
8.5	定制表视图单元	155
8.5.1	单元应用程序	155
8.5.2	向表视图单元添加子视图	155
8.5.3	使用 UITableView 的自定义子类	159
8.6	分组分区和索引分区	163
8.6.1	构建视图	163
8.6.2	导入数据	163
8.6.3	实现控制器	164
8.6.4	添加索引	167
8.7	实现搜索栏	168
8.7.1	重新考虑设计	168
8.7.2	深层可变副本	168
8.7.3	更新控制器头文件	170
8.7.4	修改视图	171
8.7.5	修改控制器实现	172
8.8	小结	180
第 9 章 导航控制器和表视图		181
9.1	导航控制器	181
9.1.1	栈的性质	181
9.1.2	控制器栈	182

9.2	由 6 个部分组成的分层应用程序：Nav	182
9.3	构建 Nav 应用程序的骨架	184
9.3.1	创建根视图控制器	185
9.3.2	设置导航控制器	185
9.4	第 1 个子控制器：展示按钮视图	191
9.5	第 2 个子控制器：校验表	198
9.6	第 3 个子控制器：表行上的控件	202
9.7	第 4 个子控制器：可移动的行	207
9.7.1	编辑模式	208
9.7.2	创建一个新的二级控制器	208
9.8	第 5 个子控制器：可删除的行	213
9.9	第 6 个子控制器：可编辑的详细窗格	218
9.9.1	创建数据模型对象	219
9.9.2	创建控制器	221
9.9.3	创建详细视图控制器	224
9.10	更多内容	238
9.11	小结	240
第 10 章 应用程序设置和用户默认设置		241
10.1	了解设置束	241
10.2	AppSettings 应用程序	242
10.3	创建项目	243
10.4	使用设置束	245
10.4.1	在项目中添加设置束	245
10.4.2	设置属性列表	246
10.4.3	添加文本字段设置	247
10.4.4	添加安全文本字段设置	249
10.4.5	添加多值字段	249
10.4.6	添加拨动开关设置	250
10.4.7	添加滑块设置	251
10.4.8	添加子设置视图	252
10.5	读取应用程序中的设置	253
10.6	更改应用程序中的默认设置	257
10.7	小结	259
第 11 章 基本数据持久性		260
11.1	应用程序的沙盒	260
11.1.1	获取 Documents 目录	261
11.1.2	获取 tmp 目录	262
11.2	文件保存策略	262
11.2.1	单个文件持久性	262
11.2.2	多个文件持久性	262
11.3	持久保存应用程序数据	263

11.4	持久性应用程序	264
11.4.1	创建持久性项目	264
11.4.2	设计持久性应用程序视图	265
11.4.3	编辑持久性类	265
11.4.4	对模型对象进行归档	269
11.4.5	实现 NSCopying	270
11.5	归档应用程序	272
11.5.1	实现 FourLines 类	272
11.5.2	实现 PersistenceViewController 类	273
11.6	使用 iPhone 的嵌入式 SQLite3	276
11.7	小结	284
第 12 章 使用 Quartz 和 OpenGL 绘图		285
12.1	图形世界的两个视图	285
12.2	本章的绘图应用程序	286
12.3	Quartz 绘图方法	286
12.3.1	Quartz 2D 的图形上下文	286
12.3.2	坐标系	287
12.3.3	指定颜色	287
12.3.4	在上下文中绘制图像	289
12.3.5	绘制形状：多边形、直线和曲线	289
12.3.6	Quartz 2D 工具示例：模式、梯度、虚线模式	289
12.4	构建 QuartzFun 应用程序	290
12.4.1	创建随机颜色	291
12.4.2	定义应用程序常量	291
12.4.3	实现 QuartzFunView 框架	292
12.4.4	向视图控制器中添加输出口和操作	294
12.4.5	更新 QuartzFunViewController.xib	297
12.4.6	绘制直线	298
12.4.7	绘制矩形和椭圆形	299
12.4.8	绘制图像	301
12.5	一些 OpenGL ES 基础知识	306
12.6	小结	316
第 13 章 轻击、触摸和手势		317
13.1	多触摸术语	317
13.2	响应者链	318
13.3	多触摸体系结构	319
13.4	触摸浏览器应用程序	320
13.5	Swipe 应用程序	324
13.6	实现多个轻扫	327
13.7	检测多次轻击	329
13.8	检测捏合操作	333

13.9	自己定义手势	336
13.10	小结	339
第 14 章 我在哪里？使用 Core Location 定位功能		340
14.1	位置管理器	340
14.1.1	设置所需的精度	341
14.1.2	设置距离筛选器	341
14.1.3	启动位置管理器	341
14.1.4	更明智地使用位置管理器	341
14.2	位置管理器委托	342
14.2.1	获取位置更新	342
14.2.2	使用 CLLocation 获取纬度和经度	342
14.2.3	错误通知	343
14.3	尝试使用 Core Location	344
14.3.1	更新位置管理器	347
14.3.2	确定移动距离	348
14.4	小结	349
第 15 章 加速计		350
15.1	加速计物理学	350
15.2	访问加速计	351
15.2.1	UIAcceleration	351
15.2.2	实现 accelerometer:didAccelerate:方法	353
15.3	摇动与击碎	354
15.3.1	用于击碎的代码	355
15.3.2	加载模拟文件	358
15.3.3	完好如初——复原触摸	359
15.4	滚弹珠程序	359
15.4.1	实现 Ball View 控制器	360
15.4.2	编写 Ball View	361
15.4.3	计算小球运动	364
15.5	小结	366
第 16 章 iPhone 照相机和照片库		367
16.1	使用图像选取器和 UIImagePickerController	367
16.2	实现图像选取器控制器委托	368
16.3	实际测试照相机和库	370
16.3.1	设计界面	370
16.3.2	实现照相机视图控制器	371
16.4	小结	374
第 17 章 应用程序本地化		375
17.1	本地化体系结构	375
17.2	使用字符串文件	376
17.3	现实中的 iPhone：本地化应用程序	378

17.3.1	查看当前区域设置	381
17.3.2	测试 LocalizeMe	381
17.3.3	本地化 nib 文件	382
17.3.4	查看本地化的项目结构	383
17.3.5	本地化图像	385
17.3.6	本地化应用程序图标	386
17.3.7	生成和本地化字符串文件	386
17.4	小结	388
第 18 章 未来之路		390
18.1	答案揭晓	390
18.1.1	苹果公司的文档	390
18.1.2	邮件列表	391
18.1.3	论坛	391
18.1.4	网站	391
18.1.5	博客	391
18.1.6	如果仍未解决问题	392
18.2	再会	392

致 谢

没有我们善良、能干又聪明的家人、朋友和同伴的支持，本书是不可能完成的。首先，感谢Terry和Deneen对我们的宽容，他为我们专心写书提供了非常好的环境。这个项目耗费了相当长的时间，但是你们从未抱怨过。我们很幸运！

本书的完成还要归功于Apress工作人员的合作。他们不仅是本书的出版者，还是我们不可多得的朋友。Clay Andres策划了本书，并将我们带到了Apress。Dominic Shakeshaft始终带着微笑处理我们的抱怨，并总能找到合适的解决方法，使本书完成得更好。Laura Esterman是一位亲切的项目经理，她是我们实现每一个目标的动力。她让本书的编写能够有条不紊地进行，并为我们指明了正确的方向。Heather Lang是个极为出色的文字编辑，很荣幸能和你一起工作，请下一次依然做我们的文字编辑！Grace Wong和生产团队把零碎的稿件整合成书，Kari Brooks-Copony提供了出色的版式设计。Kelly Winquist用我们的Word文档印刷出非常精美的页面。Pete Aylward征集宣传素材，策划推出一系列营销活动。我们对Apress的所有工作人员表示由衷的感谢！

特别感谢我们优秀的技术审稿人Mark Dalrymple。除了提供具有独到见解的反馈之外，Mark还测试了本书中的所有代码，帮助我们保证了本书的正确性。感谢您！

最后，感谢我们的孩子，他们在父亲努力工作时表现出非常好的耐心。本书是送给你们的：Maddie、Gwynnie、Ian、Kai、Daniel、Kelley和Ryan。

第1章

欢迎来到 iPhone 的世界



你想编写iPhone应用程序？iPhone可能在今后很长一段时间内都是最有趣的新兴平台。毫无疑问，它是迄今为止最新颖的移动平台，特别是现在，苹果公司还提供了一组精美的、具有良好文档的工具来支持iPhone应用程序的开发。

1.1 关于本书

本书将带你走上创建iPhone应用程序的大道。我们的目标是让你通过初步学习，理解iPhone应用程序的运行方式和构建方式。在阅读过程中，你将创建一系列小型应用程序，每个应用程序都会突出特定的iPhone特性，展示如何控制这些特性或与其交互。如果将本书中的基本知识与你自己的创造力相结合，同时借助苹果公司大量翔实的文档，你将具备创建专业级iPhone应用程序所需的一切条件。

1.2 必要条件

在开始编写iPhone软件之前，需要做一些准备工作。对于初学者，需要一台运行Leopard (OS X 10.5.3或更高版本)的基于Intel的Macintosh计算机。2006年之后上市的任何Macintosh计算机(不管是笔记本还是台式机)应该都符合要求。

无需使用具备顶级配置的计算机，MacBook或Mac Mini就能够出色地完成任务。但是，对于较早且运行速度较慢的计算机型号，进行RAM升级能够获得较大的性能提升。

你还需要注册成为iPhone开发人员。只有完成了这一步，苹果公司才允许下载iPhone SDK(软件开发工具包)。

要进行注册，请访问<http://developer.apple.com/iphone/> (中文网站为<http://www.apple.com.cn/developer/iphone/>)，如图1-1所示)，该页面应该与图1-1中显示的页面类似。页面中提供了最新且功能最强大的iPhone SDK的下载链接。单击该链接将进入包含3个选项的注册页面。

最简单(而且免费)的选项是单击Download the Free SDK按钮。页面将提示输入Apple ID。使用你的Apple ID登录。如果还没有Apple ID，请单击Create Apple ID按钮，创建一个Apple ID，然后再登录。登录之后，将进入iPhone开发主页面。其中不仅有SDK的下载链接，还提供了各类文档、视频和示例代码等的链接，所有这些资源都能帮你进行iPhone应用程序开发。



图1-1 苹果公司的iPhone开发中心网站

iPhone SDK中包含的一个最重要的元素是Xcode，它是苹果公司的IDE（集成开发环境）。Xcode提供了各种实用工具，用于创建和调试源代码，编译应用程序以及调优应用程序性能。学习完本书，你将会迷恋上Xcode！

这个免费的SDK还包含一个仿真器，它支持在Mac上运行大多数iPhone程序。这对于学习如何编写iPhone程序极其有用。但是，免费选项不支持将应用程序下载到实际的iPhone（或iPod Touch）中。此外，它也不支持在苹果公司的iPhone App Store上分发应用程序。要实现这些功能，需要使用另外两个下载选项，它们不是免费的。

说明 仿真器不支持依赖于硬件的特性，比如iPhone的加速计或摄像功能。要支持这些特性，需要使用其他选项。

标准版程序的价格为99美元。它提供了全面的开发工具、资源和技术支持，支持通过苹果公司的App Store分发应用程序，并且最重要的是，支持在iPhone上（而不仅是仿真器上）测试和调试代码。

企业版程序的价格为299美元，可供企业开发专用的、内部的iPhone和iPod Touch应用程序。

有关这两种程序的详细信息，请访问<http://developer.apple.com/iphone/program/>。

由于iPhone是一种始终连网的移动设备，并且使用的是其他公司的无线基础设施，因此苹果公司对iPhone开发人员的限制比对Mac开发人员多得多，Mac开发人员无需经过苹果公司的审查或批准就能够编写和分发程序。

苹果公司添加这些限制，更多的是为了尽量避免分发恶意或效率低下的程序，因为这类程序可能降低共享网络的性能。开发iPhone应用程序似乎麻烦不少，但苹果公司在简化开发过程方面付出了巨大努力。还应该提及的是，99美元的价格比微软公司的软件开发IDE——Visual Studio的价格低得多。

另外，很明显，你还需要一部iPhone。虽然大部分代码都可以通过iPhone仿真器进行测试，但并非所有程序都是如此。一些应用程序需要在实际的iPhone上进行全面测试，然后才能分发给公众。

说明 如果要注册标准版或企业版程序，你应该立即注册。批准过程可能需要一些时间，并且通过批准之后才能在iPhone或iPod Touch上运行应用程序。但是不必担心，前几章中的所有项目以及本书中的大多数应用程序，都可以在iPhone仿真器上运行。

1.3 必备知识

学习本书应该具备一定的编程知识。你应该理解面向对象编程的基础知识，例如，了解对象、循环和变量的含义，还应该熟悉Objective-C编程语言。SDK中的Cocoa Touch是本书使用的主要工具，它使用的是Objective-C 2.0编程语言，但是如果不懂Objective-C语言的新增特性也没有关系。我们将重点介绍要使用的2.0语言特性，并解释其工作原理和使用它的原因。

你还应该熟悉iPhone本身。就像在任何其他平台中编写应用程序一样，你需要熟悉iPhone的各种特性，并了解iPhone界面以及iPhone程序的外观。

还不熟悉Objective-C？

如果你从未使用Objective-C编写过程序，那么以下资源有助于你了解该语言。

首先，阅读由Mac编程专家Mark Dalrymple和Scott Knaster撰写的*Learn Objective-C on the Mac^①*一书，该书浅显易懂，是学习Objective-C基础知识的优秀图书：

<http://www.apress.com/book/view/9781430218159>

① 中文版即将由人民邮电出版社出版。——编者注

接下来，访问Apple iPhone开发中心网站并下载*The Objective-C 2.0 Programming Language*一书的电子版，该书深入浅出地介绍了Objective-C 2.0的方方面面，是一本优秀的参考指南：

<http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/ObjectiveC>

注意，需要登录才能访问此文档。

1.4 编写iPhone应用程序有何不同

如果从未使用过Cocoa或它的前期产品NextSTEP，那么你可能会发现Cocoa Touch（用于编写iPhone应用程序的应用程序框架）稍显另类。它与其他常用应用程序框架（如用于构建.NET或Java应用程序的框架）之间存在一些基本差异。你起初可能会有点不知所措，但不必担心，只要勤加练习，就可以掌握其中的规律。

如果你具备使用Cocoa或NextSTEP编程的经验，则会发现iPhone SDK中有许多熟悉的身影。其中的许多类都是从用于Mac OS X开发的程序版本中原样借鉴过来的，一些类即便存在不同，它们也遵循相同的基本原则，并使用类似的设计模式。但是，Cocoa和Cocoa Touch之间却存在一些差异。

无论你的知识背景如何，都需要谨记iPhone开发与桌面应用程序开发之间的重要差异。

1.4.1 只有一个正在运行的应用程序

除了操作系统之外，任何时候iPhone上都只能运行一个应用程序。随着iPhone内存的增大、处理器的增强，这一点在未来可能会发生变化。但是在目前，在执行代码时，你的应用程序将是唯一正在运行的程序。若你的应用程序不是用户正在交互中的，那么它不会起作用。

1.4.2 只有一个窗口

在桌面及笔记本操作系统中，多个程序可以同时运行，并且可以分别创建和控制多个窗口。而iPhone则有所不同，它只允许应用程序操作一个“窗口”。应用程序与用户的所有交互都在这个窗口中完成，而且这个窗口大小就是iPhone屏幕的大小，是固定的。

1.4.3 受限访问

计算机上的程序可以访问启动它们的用户能够访问的任何内容，而iPhone则严格限制了应用程序的权限。你只能在iPhone为应用程序创建的文件系统中读写文件。此区域称为应用程序的沙盒，应用程序在其中存储文档、首选项等需要存储的数据。

应用程序还存在其他方面的限制。举例来说，你不能访问iPhone上端口号较小的网络端口，或者执行台式计算机中需要根用户或管理员权限才能执行的操作。

1.4.4 有限的响应时间

由于其使用方式特殊，iPhone及其应用程序需要具备较快的响应时间。启动应用程序之后，

需要打开应用程序，载入首选项和数据，并尽快在屏幕上显示主视图，这一切要在几秒之内发生。只要应用程序在运行，就可以从其下方拖出一个菜单条。如果用户按主页（home）按钮，iPhone 就会返回主页，并且用户需要快速保存一切内容并退出。如果未在5秒之内保存并放弃控制，则应用程序进程将被终止，无论用户是否已经完成保存。

因此，你在设计iPhone应用程序时需要注意这一点，以确保用户退出时不会丢失数据。

1.4.5 有限的屏幕大小

iPhone的屏幕显示效果非常出色，从它推出直到现在，它一直是消费者设备上分辨率最高的屏幕。但是，iPhone的显示屏并不大，你施展的空间要比现代计算机小很多，仅有 480×320 像素。而在撰写本书时，苹果公司最便宜的iMac支持 1680×1050 像素，最便宜的笔记本电脑MacBook支持 1280×800 像素。而苹果公司最大的显示器，30英寸的Cinema Display，支持 2560×1600 像素。

1.4.6 有限的系统资源

阅读本书的任何资深程序员可能都会对128 MB内存、4 GB存储空间的机器嗤之以鼻，因为其资源实在是非常有限，但这种机器却是真实存在的。或许，开发iPhone应用程序与在内存为48 KB的机器上编写复杂的电子表格应用程序不属于同一级别，二者之间没有可比性，但由于iPhone的图形属性和它的功能，所以其内存不足是非常容易出现的。目前上市的iPhone具备128 MB物理内存，当然这还会随时间不断增长。内存的一部分用于屏幕缓冲和其他一些系统进程。通常，大约一半内存将留给应用程序使用。

虽然64 MB对于这样的小型计算机可能已经足够了，但谈到iPhone的内存时还有另一个因素需要考虑：现代计算机操作系统，如Mac OS X，会将一部分未使用的内存块写到磁盘的交换文件中。这样，当应用程序请求的内存超过计算机实际可用的内存时，它仍然可以运行。但是，iPhone OS并不会将易失性内存（如应用程序数据）写到交换文件中。因此，应用程序可用的内存量将受到电话中未使用物理内存量的限制。

Cocoa Touch提供了一种内置机制，可以将内存不足的情况通知给应用程序。出现这种情况时，应用程序必须释放不需要的内存，甚至可能被强制退出。

1.4.7 缺少 Cocoa 工具

如果你在接触iPhone之前有过Cocoa方面的经验，那么你过去习惯使用的一些工具在iPhone中已经不可用了。iPhone SDK不支持Core Data或Cocoa Binding。我们之前已经说过，Cocoa Touch使用的是Objective-C 2.0，但该语言中的一个关键特性在iPhone中并不可用：Cocoa Touch不支持垃圾收集。

1.4.8 新属性

前面已经说过，Cocoa Touch缺少Cocoa的一些功能，但iPhone SDK中也有一些新功能是Cocoa所没有的，或者至少不是在任何Mac上都可用的。iPhone SDK为应用程序提供了一种定位方法，

即使用Core Location确定电话的当前地理坐标。iPhone还提供了一个内置的摄像和照片库，并且SDK允许应用程序访问这两者。iPhone还提供了一个内置的加速计，用于检测iPhone的持有和移动方式。

1.4.9 与众不同的方法

iPhone没有键盘和鼠标，这意味着它与用户的交互方式与通用的计算机截然不同。所幸的是，大多数交互都不需要你来处理。举例来说，如果在应用程序中添加一个文本字段，则iPhone知道在用户单击该字段时调用键盘，而不需要编写任何额外的代码。

1.5 本书内容

下面是本书其余章节的简要概述。

第2章：讲述如何使用Xcode和Interface Builder创建一个简单的界面，并在iPhone屏幕上添加一些文本。

第3章：开始实现与用户的交互，构建一个简单的应用程序，用于在运行时根据用户按下的按钮动态更新显示的文本。

第4章：以第3章为基础，介绍其他一些iPhone标准用户界面控件。我们还将介绍如何使用警告框和表提醒用户做出决策，或者通知用户发生了一些异常事件。

第5章：了解自动旋转机制，该机制允许在纵向或横向模式下使用iPhone应用程序。

第6章：介绍更多高级用户界面，并阐述如何创建多视图界面。我们将更改在运行时为用户显示的视图，以创建更加复杂的用户界面。

第7章：介绍如何实现工具栏控制器，它是一个标准的iPhone用户界面。

第8章：介绍表视图。表视图是向用户提供数据列表的主要方法，并且是基于分层导航的应用程序的基础。

第9章：介绍如何实现分层列表，它是最常用的iPhone应用程序界面之一，你可以通过它查看更多或更详细的数据。

第10章：介绍如何实现应用程序设置，iPhone中的这种机制允许用户设置他们的应用程序级首选项。

第11章：介绍iPhone上的数据管理。我们将讨论如何创建用于保存应用程序数据的对象，以及如何将这些数据持久存储到iPhone的文件系统和嵌入式数据库SQLite中。

第12章：绘图是人们的普遍爱好，这一章介绍如何实现一些自定义绘图，这需要使用Quartz和OpenGL ES中的基本绘图函数。

第13章：iPhone的多点触摸屏幕可以接受用户的各种手势输入。这一章讲述如何检测基本的手势，如双指捏合和单指滑动，还将介绍定义新手势的过程，并讨论新手势的适用情况。

第14章：iPhone可以通过Core Location确定其纬度和经度。这一章将编写利用Core Location计算iPhone的物理位置的代码，并在各种应用中使用该信息。

第15章：介绍如何与iPhone加速计交互，iPhone通过加速计确定其持有方式。我们将讨论应用程序如何通过该信息完成一些有趣的任务。

第16章：每个iPhone都有自己的摄像设备和图片库，这两者都可供应用程序使用。这一章介绍如何使用它们。

第17章：iPhone现已遍及70多个国家，这一章介绍以何种方式编写应用程序能方便地把所有部分翻译为其他语言，从而发掘应用程序的潜在用户。

第18章：至此，你已经掌握了iPhone应用程序的基本构建方法。但接下来再应向何处去呢？这一章将探索掌握iPhone SDK的后续步骤。

1.6 准备开始吧

iPhone是一款全新的、令人难以置信的计算平台，是轻松开发的利器。编写iPhone应用程序将成为一种全新的体验，这种体验与之前你使用过的任何平台都不同。所有看似熟悉的功能都具有其独特的一面，但随着深入体会本书中的代码，你将能把这些概念紧密联系起来并融会贯通。

应该谨记，本书中的练习并不是一份检验清单，似乎完成这些练习之后，你就自动具有了iPhone开发专家的资格。在继续下一个项目之前，确保已经理解了这些概念和原理。不要害怕修改代码。多多尝试并观察结果是在Cocoa Touch环境中克服编码困难的最佳方法。

如果你已经安装了iPhone SDK，请继续阅读本书。如果还没有，请立即安装iPhone SDK。然后开始iPhone之旅！

下一章将构建一个基于分层导航的应用程序，它类似于iPhone随带的电子邮件应用程序。通过这个应用程序，用户可以访问数据嵌套列表和编辑数据。不过，在此之前，需要先掌握表视图的基本概念。这正是本章将要介绍的内容。

表视图是用于向用户显示数据的一种最常见的机制。它们是高度可配置的对象，可以被配置为用户所需的任何形式。电子邮件使用表视图显示账户、文件夹和消息的列表，但是表视图并不仅限于显示文本数据。还可以在YouTube、Settings和iPod应用程序中使用表视图，尽管这些应用程序具有十分不同的外观（参见图8-1）。



图8-1 虽然看起来各自不同，但Settings、iPod和YouTube应用程序都使用表视图来显示数据

8.1 表视图基础

表用于显示数据列表。数据列表中的每项都由行表示。iPhone表没有限制行的数量，其数量仅受可用存储空间的限制。iPhone表可以只有一列。

表视图是显示表数据的视图对象，它是UITableView类的一个实例。表中的每个可见行都由UITableViewCell类实现。因此，表视图是显示表中可见部分的对象，表视图单元负责显示表中的一行（参见图8-2）。



图8-2 每个表视图都是UITableView的一个实例，每个可见行都是UITableViewCell的一个实例

8

表视图并不负责存储表中的数据。它们只存储足够绘制当前可见行的数据。表视图从遵循UITableviewDelegate协议的对象获取配置数据，从遵循UITableviewDataSource协议的对象获得行数据。本章稍后介绍的示例应用程序将展示如何实现这些操作。

如前所述，所有的表都只有1列。但是，图8-1右边所示的YouTube应用程序外观确实至少拥有2列，如果数一数图标，甚至会发现原来有3列。不过情况并非如此。表中的每一行都由一个UITableViewCell表示，可以使用一个图像、一些文本和一个可选的辅助图标来配置每个UITableViewCell对象。其中辅助图标是指位于右边的一个小图标，下一章将对它进行详细的介绍。

如果需要的话，可以在一个单元中放置更多的数据。可以通过两种基本方法来完成此操作。一种方法是向UITableViewCell添加子视图；另一种方法是通过子类化UITableViewCell。你可以按照任何喜欢的方式展示表视图单元，也可以添加任何想要的子视图。这样看来，单列限制并不像开始听起来那样可怕。如果这些让你感到迷惑，别担心，本章稍后将介绍这方面的技巧。

分组表和索引表

表视图分为两种基本样式。第一种是分组表。分组表中的每个组都由嵌入在圆角矩形中的多个行组成，如图8-3最左边的图片所示。注意，一个分组表可以只包含一个组。

另一种类型是索引表（在某些地方又称为无格式表）。索引表是默认的样式。任何没有圆角矩形属性的表都是索引表视图。

如果数据源提供了必要的信息，通过表视图，用户可以使用右侧的索引来导航列表。图8-3显示了一个分组表、一个不带索引的索引表（无格式表）和一个带有索引的索引表。



图8-3 相同的表视图分别显示为分组表（左）、不带索引的索引表（通常指无格式表，中间）、和一个带有索引的索引表（右）

表中的每个部分被称为数据源中的分区（section）。在分组表中，每个分组都是一个分区（参见图8-4）。在索引表中，数据的每个索引分组都是一个分区。例如，在图8-3所示的索引表中，以“A”开头的所有名称都是一个分区，以“B”开头的那些名称则是另一个分区，依此类推。



图8-4 分组表中的分区和行显而易见，不过所有的表都支持分区和行

分区主要有两个作用。在分组表中，每个分区表示一个组。在索引表中，每个分区对应一个索引条目。因此，如果你希望显示一个按字母顺序列出索引且每个字母作为一个索引条目的列表，那么你将拥有26个分区，每个分区包含以特定字母开头的所有值。

注意 从技术上来说，可以创建带有索引的分组表。即便如此，也不应该为分组表视图提供索引。*iPhone Human Interface Guideline (iPhone人性化界面指南)* 这本书中明确指出分组表不应该提供索引。

在本章中，我们将创建这两种类型的表。

8.2 实现一个简单的表

下面通过一个最简单的示例来了解表视图的工作原理。本示例将显示一个文本值列表。

在Xcode中创建一个新项目。对于本章来说，我们将使用基于视图的应用程序模板。选择这一项，然后将项目命名为Simple Table。

8.2.1 设计视图

8

展开Resources和Classes文件夹。这是一个极为简单的应用程序，它不需要任何输出口或操作，双击Simple_TableViewController.xib，在Interface Builder中打开该文件。View窗口应该已经打开，因此，只需要在库中找到Table View（参见图8-5），并将它拖到View窗口中即可。

表视图会自动将其高度和宽度调整为View窗口的高度和宽度（参见图8-6）。这正是我们所希望的。将表视图设计为占据屏幕的整个宽度，以及除应用程序导航栏、工具栏或标签栏之外的高度。

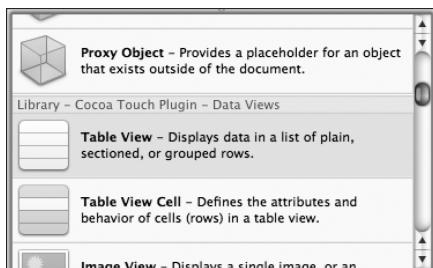


图8-5 库中的表视图



图8-6 添加表视图之后的View窗口

将表视图拖到View窗口上之后，它应该仍然处于选中状态，如果不是，那么通过单击操作来选定它，然后按下 $\text{⌘} \text{2}$ 打开连接检查器。你会注意到，表视图的前两个可用连接和选取器视图的前两个连接一样，都是数据源和委托。从每个连接旁边的圆圈拖到File's Owner图标。这样一来，控制器类就成为此表的数据源和委托。完成上述操作之后，保存并关闭，然后返回到Xcode。

8.2.2 编写控制器

下面是控制器类的头文件。单击Simple_TableViewController.h，并添加以下代码：

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

@interface Simple_TableViewController : UIViewController
    <UITableViewDelegate, UITableViewDataSource>
{
    NSArray *listData;
}
@property (nonatomic, retain) NSArray *listData;
@end
```

上述代码的作用是让类遵从两个协议，类需要使用这两个协议来充当表视图的委托和数据源，然后声明一个数组用于放置将要显示的数据。

现在切换到Simple_TableViewController.m，添加更多的代码：

```
#import "Simple_TableViewController.h"

@implementation Simple_TableViewController
@synthesize listData;
#pragma mark Table View Controller Methods
- (void)viewDidLoad {
    NSArray *array = [[NSArray alloc] initWithObjects:@"Sleepy", @"Sneezy",
                      @"Bashful", @"Happy", @"Doc", @"Grumpy", @"Dopey", @"Thorin",
                      @"Dorin", @"Nori", @"Ori", @"Balin", @"Dwalin", @"Fili", @"Kili",
                      @"Oin", @"Gloin", @"Bifur", @"Bofur", @"Bombur", nil];
    self.listData = array;
    [array release];
    [super viewDidLoad];
}
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}
- (void)dealloc {
    [listData release];
    [super dealloc];
}
#pragma mark -
#pragma mark Table View Data Source Methods
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
```

```

{
    return [self.listData count];
}
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *SimpleTableIdentifier = @"SimpleTableIdentifier";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        SimpleTableIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
            reuseIdentifier: SimpleTableIdentifier] autorelease];
    }

    NSUInteger row = [indexPath row];
    cell.textLabel.text = [listData objectAtIndex:row];
    return cell;
}
@end

```

8

我们为控制器添加了3个方法。你可能对第一个方法`viewDidLoad`感到很熟悉，因为我们前面使用过类似的方法。此方法只创建了一个要传递给表的数据的数组。在实际应用程序中，此数组很可能来自于另一个源，比如文本文件、属性列表或URL。

继续往下看，你会看到我们添加了两个数据源方法。第一个方法是`tableView:numberOfRowsInSection:`，表使用它来查看指定分区中有多少行。正如你所希望的，默认的分区数量为1，此方法用于返回组成列表的表分区中的行数。只需返回数组中数组项的数量即可。

下一个方法可能需要一些解释，让我们更仔细地看一下此方法：

```

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{

```

当表视图需要绘制其中一行时，则会调用此方法。你会注意到此方法的第二个参数是一个`NSIndexPath`实例。表视图正是使用此机制把分区和行绑定到一个对象中的。要从`NSIndexPath`中获得一行或一个分区，只需要调用行方法或分区方法就可以了，这两个方法都返回一个`int`值。

第一个参数`tableView`是对发起请求的表的引用。通过它，我们可以创建充当多个表的数据源的类。

下面，声明一个静态字符串实例。

```
static NSString *SimpleTableIdentifier = @"SimpleTableIdentifier";
```

此字符串充当表示某种表单元的键。在此表中，我们将只使用一种单元，因此定义一种标识符就可以了。表视图在iPhone的小屏幕上一次只能显示几行，但是表自身能够保存相当多的数据。记住，表中的每一行都由一个`UITableViewCell`实例表示，该实例是`UIView`的一个子类，这就意味着每一行都能拥有子视图。对于大型表来说，如果视图为表中的每一行都分配一个表视图单元，

不管该行当前是否正被显示，这都将带来大量开销。幸好表并不是这样工作的。

相反，因滚动操作离开屏幕的一些表视图单元，将被放置在一个可以被重用的单元序列中。如果系统运行比较慢，表视图就从序列中删除这些单元，以释放存储空间，不过，只要有可用的存储空间，表视图就会重新获取这些单元，以便以后再次使用它们。

当一个表视图单元滚出屏幕时，另一个表视图单元就会从另一边滚动到屏幕上。如果滚动到屏幕上的新行重新使用从屏幕上滚动下来的其中一个单元，系统就会避免与不断创建和释放那些视图相关的开销。要充分利用此机制，我们需要让表视图给定一个出列单元，该出列单元正是我们需要的类型。注意，我们现在正在使用前面声明的**NSString**标识符。实际上，我们需要一个**SimpleTableIdentifier**类型的可重用单元：

```
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:  
    SimpleTableIdentifier];
```

现在，表视图中可能没有任何多余的单元了，我们来检查这些**cell**，看一下它是否为零(**nil**)。如果是，则使用上面提到的标识符字符串手动创建一个新的表视图单元。从某种程度上来说，我们将不可避免地重复使用此处创建的单元，因此需要确保它具有相同的类型。

```
if (cell == nil) {  
    cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero  
        reuseIdentifier: SimpleTableIdentifier] autorelease];  
}
```

现在，我们拥有了一个可以返回到表视图的表视图单元。下面所有要做的就是把需要显示的信息放在该表视图单元中。在表的一行内显示文本是很常见的任务，因此表视图单元提供了一个名称为**text**的属性，我们可以设置此属性以显示字符串。对于这种情况，我们需要从**listData**数组中获取正确的字符串，然后使用它设置表视图单元的**text**属性。

要完成上述操作，需要知道表视图需要显示哪些行。可以从**indexPath**变量获取该信息，如下所示：

```
NSUInteger row = [indexPath row];
```

我们使用这个值从数组获取正确的字符串，将它分配给单元的**text**属性，然后返回该单元。

```
cell.text = [listData objectAtIndex:row];  
return cell;  
}
```

情况并不那么糟糕，对吗？下面编译并运行应用程序。等一下，是不是有地方弄错了？出现了一个链接错误？你认为我们应该怎么做呢？

如果你被难住了，那么给你一个提示：我们使用了一个名为**CGRrectZero**的常量，它是Core Graphics框架的一部分。默认情况下，Core Graphics框架没有连接到项目。如果你忘记如何将它连接进来的话，可以参见第5章，其中逐步地介绍了此过程。解决这个问题后，就可以成功完成编译了。运行应用程序，你将看到显示在表视图中的数组值（参见图8-7）。

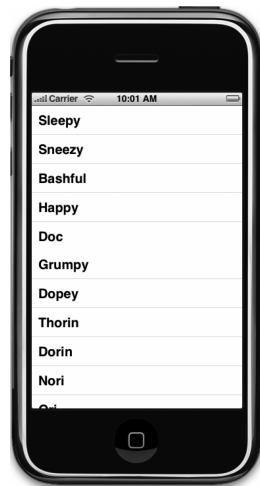


图8-7 简单的表应用程序

8.3 添加一个图像

要是可以向每一行添加一个图像就好了。我们需要创建一个UITableViewController子类来添加图像吗？不用。实际上，如果能够让图像位于每一行的左侧就不需要这么做了。默认的表视图单元会把这个情况处理好。下面我们来看一看。

在项目归档文件的08 Simple Table文件夹中，找到名为star.png的文件，然后把它添加到项目的Resources文件夹中。star.png是为此项目准备的一个小图标。

下面看看代码部分。在Simple_TableViewController.m文件中，在tableView:cellForRowAtIndexPath:方法中添加以下代码：

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *SimpleTableIdentifier = @"SimpleTableIdentifier";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        SimpleTableIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
            reuseIdentifier: SimpleTableIdentifier] autorelease];
    }

    NSUInteger row = [indexPath row];
    cell.textLabel.text = [listData objectAtIndex:row];
    UIImage *image = [UIImage imageNamed:@"star.png"];
    cell.imageView.image = image;
    return cell;
}
```

8

好，这就完成了。刚才我们把单元的image属性设置为想要显示的任何图像。现在，如果编译并运行应用程序，出现的列表每一行的左侧都有一个漂亮的小星星图标（参见图8-8）。当然，如果愿意的话，可以为表中的每一行设置一个不同的图像。或者，费些工夫，为所有行分别应用不同的图标。

8.4 附加配置

你会注意到我们使用控制器作为此表视图的数据源和委托，不过到现在为止，还没有真正实现UITableViewDelegate的任何方法。与选取器视图不同，较简单的表视图不需要委托代替它们完成一些功能。数据源提供了绘制表所需要的所有数据。委托只是用于配置表视图的外观并处理某些用户交互。现在，让我们看一

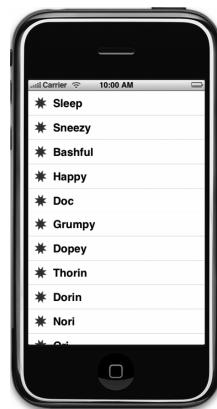


图8-8 使用单元的image属性，为每个表视图单元添加一个图像

以下几个配置选项。下一章将更详细地介绍此内容。

8.4.1 设置缩进级别

可以使用委托指定缩进某些行。在Simple_TableViewController.m文件中，在代码中的`@end` declaration上方添加以下方法：

```
#pragma mark -
#pragma mark Table Delegate Methods

- (NSInteger)tableView:(UITableView *)tableView
    indentationLevelForRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSUInteger row = [indexPath row];
    return row;
}
```

此方法把每一行的缩进级别设置为其行号，所以第0行的缩进级别为0，第1行的缩进级别为1，依此类推。缩进级别是一个整数，它会告诉表视图把一行向右移动一点。缩进级别的数量越大，行向右缩进得就越多。例如，可以使用这项技术来表示一行从属于另一行，就好像在电子邮件中表示子文件夹一样。

再次运行应用程序，可以看到每一行都在上一行的基础上向右移动了一些距离（参见图8-9）。

8.4.2 处理行的选择

表的委托可以使用两个方法确定用户是否选择了特定的行。一个方法在一行被突出显示之前调用，并且可以用于阻止选中此行，甚至改变被选中的行。让我们来实现这个方法，并指定第一行是不能被选中的。将以下方法添加到Simple_TableViewController.m的尾部，位于在`@end`声明之前：

```
- (NSIndexPath *)tableView:(UITableView *)tableView
    willSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSUInteger row = [indexPath row];
    if (row == 0)
        return nil;

    return indexPath;
}
```

这个方法获取传递过来的`indexPath`，它用于表示哪项将被选中。我们的代码着眼于哪一行将被选中。如果这一行是第一行，其索引将始终为零，那么它将返回`nil`，表示实际上没有行被

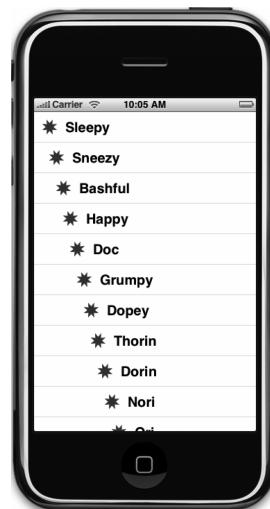


图8-9 表中的每一行都比上一行具有更高的缩进级别

选中。否则，它返回`indexPath`，表示选择可以继续进行。

在编译和运行应用程序之前，我们还要实现委托方法，在一行被选中之后调用该方法，通常它也是实际处理选择的地方。用户选中一行时，可以在这里做任何操作。在下一章中，我们将使用此方法处理更深入的问题。本章将只使用此方法抛出一个警告以显示某一行被选中了。将下面的方法添加到`Simple_TableViewController.m`的尾部，位于`@end`声明之前。

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSUInteger row = [indexPath row];
    NSString *rowValue = [listData objectAtIndex:row];

    NSString *message = [[NSString alloc] initWithFormat:
        @"You selected %@", rowValue];
    UIAlertView *alert = [[UIAlertView alloc]
        initWithTitle:@"Row Selected!"
        message:message
        delegate:nil
        cancelButtonTitle:@"Yes I Did"
        otherButtonTitles:nil];
    [alert show];

    [message release];
    [alert release];
}
```

添加此方法之后，编译并运行应用程序。看一下你是否能够选中第一行（应该不能），然后选择其他行。被选中的行应该会突出显示，然后弹出警告通知你选择的是哪一行（参见图8-10）。

注意，你还可以在传递回`indexPath`之前修改索引路径，这将导致不同的行和/或分区被选中。你不会经常这样做，因为没有什么理由需要更改用户的选。在大多数情况下，使用此方法时将返回`indexPath`或`nil`，以允许或禁止某个选择。

8.4.3 更改字体大小和行高

假设我们希望更改表视图中使用的字体大小。在大多数情况下，不应该覆盖默认的字体，那是用户所希望看到的。不过有时候我们有合适的原因这样做。在`tableView:cellForRowAtIndexPath:`方法中添加下面的代码行，然后编译并运行：



图8-10 在本示例中，第一行是不可选的。当选中其他任意行时会显示一个警告。此功能是使用委托方法完成的

```

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *SimpleTableIdentifier = @"SimpleTableIdentifier";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        SimpleTableIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
            reuseIdentifier:SimpleTableIdentifier] autorelease];
    }

    NSUInteger row = [indexPath row];
    cell.textLabel.text = [listData objectAtIndex:row];
    cell.textLabel.font = [UIFont boldSystemFontOfSize:80];
    UIImage *image = [UIImage imageNamed:@"star.png"];
    cell.imageView.image = image;
    return cell;
}

```

现在运行应用程序，列表中的值会变得很大，但是它们的大小并不适合行（参见图8-11）。好，现在靠表视图委托来援救吧！表视图委托可以指定表中的行高。实际上，如果需要的话，它可以为每一行指定唯一值。下面向控制器类中添加此方法，代码位于@end之前。

```

#pragma mark -
#pragma mark Table View Delegate Methods
- (CGFloat)tableView:(UITableView *)tableView
heightForRowAtIndexPath:(NSIndexPath *)indexPath
{
    return 180;
}

```

在上面的代码中，我们使表视图把所有行高都设置为行180像素。编译并运行应用程序，现在表中的行应该高多了（参见图8-12）。



图8-11 大字体效果不错！不过如果我们
能看见所有内容就更好了



图8-12 使用委托更改行大小。太大了吗？
可能吧

8.4.4 委托还能做什么？

委托还能处理更多任务，下一章在介绍分层数据时会用到这些任务中的大多数。要了解更多内容，请使用文档浏览器查看UITextViewDelegate协议，然后看一下还有什么可用的其他方法。

8.5 定制表视图单元

你可以直接为表视图做许多事情，不过一般来说，你会希望以不受UITableViewCell直接支持的方式格式化每一行中的数据。对于这种情况，可以采用两种基本方法。一种方法是向UITableViewCell添加子视图，另一种方法是创建一个UITableViewCell子类。下面我们来看一下这两种方法。

8.5.1 单元应用程序

要展示如何使用自定义单元，我们将创建一个新的应用程序，使用另一个表视图，然后向用户显示两行信息（参见图8-13）。应用程序将显示一系列常见计算机模型的名称和颜色。通过向表视图单元添加子视图，我们将在同一个表单元中显示这两组信息。

8.5.2 向表视图单元添加子视图

默认的表视图单元只显示一行文本。即使你试图通过指定一个包含回车符的字符串让单元显示多行，它也会删除回车符，并在下一个单元的行中显示数据。现在我们要创建一个项目，向单元添加子视图以摆脱这种束缚，这样就可以在每个单元中显示两行数据。

使用基于视图的应用程序模板创建一个新的Xcode项目，将它命名为Cells。双击CellsViewController.xib，添加一个TableView，然后像我们在上一章所做的，把委托和数据源设置为File's Owner。保存并回到Xcode。如果需要，可以参考8.6.1节的内容。

1. 修改控制器头文件

单击CellsViewController.h，添加以下代码：

```
#import <UIKit/UIKit.h>
#define kNameValueTag      1
#define kColorValueTag     2

@interface CellsViewController : UIViewController
<UITableViewDataSource, UITableViewDelegate>
{
    NSArray      *computers;
```

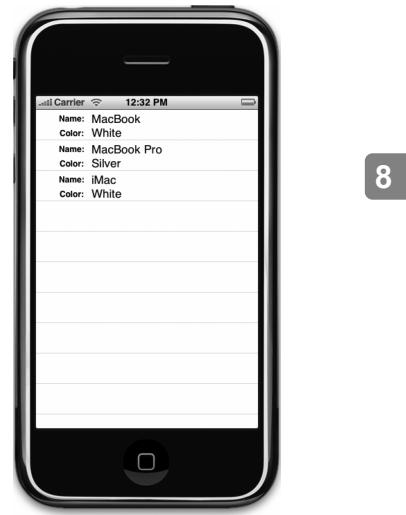


图8-13 向表视图单元添加子视图，可以让一个单元支持多行数据

```

}
@property (nonatomic, retain) NSArray *computers;
@end

```

你首先会注意到这里我们定义了两个常量。我们稍后将使用这两个常量为将要添加到表视图单元中的子视图分配标记（tag）。下面将向一个单元中添加4个子视图，其中2个需要在每一行进行更改。为此，需要在使用特定的行数据更新单元时，通过某种机制检索单元中的两个字段。如果为需要再次使用的每个标签设置唯一的标记值，那么将能够从表视图单元检索它们并设置它们的值。

2. 实现控制器代码

在控制器中，我们需要设置要用到的一些数据，然后通过实现表数据源方法将这些数据反馈给表。单击CellsViewController.m，然后添加以下代码：

```

#import "CellsViewController.h"

@implementation CellsViewController
@synthesize computers;
- (void)viewDidLoad {

    NSDictionary *row1 = [[NSDictionary alloc] initWithObjectsAndKeys:
        @"MacBook", @"Name", @"White", @"Color", nil];
    NSDictionary *row2 = [[NSDictionary alloc] initWithObjectsAndKeys:
        @"MacBook Pro", @"Name", @"Silver", @"Color", nil];
    NSDictionary *row3 = [[NSDictionary alloc] initWithObjectsAndKeys:
        @"iMac", @"Name", @"White", @"Color", nil];

    NSArray *array = [[NSArray alloc] initWithObjects:row1, row2,
        row3, nil];
    self.computers = array;

    [row1 release];
    [row2 release];
    [row3 release];
    [array release];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

```

```
- (void)dealloc {
    [computers release];
    [super dealloc];
}

#pragma mark -
#pragma mark Table Data Source Methods
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    return [self.computers count];
}
-(UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellTableIdentifier = @"CellTableIdentifier ";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        CellTableIdentifier];
    if (cell == nil) {
        CGRect cellFrame = CGRectMake(0, 0, 300, 65);
        cell = [[[UITableViewCell alloc] initWithFrame: cellFrame
            reuseIdentifier: CellTableIdentifier] autorelease];

        CGRect nameLabelRect = CGRectMake(0, 5, 70, 15);
        UILabel *nameLabel = [[UILabel alloc] initWithFrame: nameLabelRect];
        nameLabel.textAlignment = UITextAlignmentRight;
        nameLabel.text = @"Name:";
        nameLabel.font = [UIFont boldSystemFontOfSize:12];
        [cell.contentView addSubview: nameLabel];
        [nameLabel release];

        CGRect colorLabelRect = CGRectMake(0, 26, 70, 15);
        UILabel *colorLabel = [[UILabel alloc] initWithFrame:
            colorLabelRect];
        colorLabel.textAlignment = UITextAlignmentRight;
        colorLabel.text = @"Color:";
        colorLabel.font = [UIFont boldSystemFontOfSize:12];
        [cell.contentView addSubview: colorLabel];
        [colorLabel release];

        CGRect nameValueRect = CGRectMake(80, 5, 200, 15);
        UILabel *nameValue = [[UILabel alloc] initWithFrame:
            nameValueRect];
        nameValue.tag = kNameValueTag;
        [cell.contentView addSubview:nameValue];
        [nameValue release];

        CGRect colorValueRect = CGRectMake(80, 25, 200, 15);
        UILabel *colorValue = [[UILabel alloc] initWithFrame:
```

```

        colorValueRect];
colorValue.tag = kColorValueTag;
[cell.contentView addSubview:colorValue];
[colorValue release];

}

NSUInteger row = [indexPath row];
NSDictionary *rowData = [self.computers objectAtIndex:row];
UILabel *name = (UILabel *)[cell.contentView viewWithTag:
    kNameValueTag];
name.text = [rowData objectForKey:@"Name"];

UILabel *color = (UILabel *)[cell.contentView viewWithTag:
    kColorValueTag];
color.text = [rowData objectForKey:@"Color"];
return cell;
}
@end

```

在这里，`viewDidLoad`方法创建了一系列字典。每个字典都包含表中一行的名称和颜色信息。某一行中的名称在字典的`Name`键下，颜色在`color`键下。我们把所有的字典放到了同一个数组里，这就是此表的数据。

让我们着重看一下`tableView:cellForRowAtIndexPath:`，此方法中真正添加了新内容。代码的前两行像前面介绍过的一样，先创建一个标识符，然后，如果表提供了出列的表视图单元，则要求表将该单元退出队列。

如果表没有任何可以重用的单元，就必须创建一个新的单元。创建新单元时，还需要创建和添加将要用到的子视图，以实现每单元两行的表。让我们更仔细地研究一下代码。首先，创建一个单元，其实这跟前面一样，除非不使用表视图进行计算，而是手动指定单元的大小。

```

CGRect cellFrame = CGRectMake(0, 0, 300, 65);
cell = [[[UITableViewCell alloc] initWithFrame: cellFrame
reuseIdentifier: CellTableIdentifier] autorelease];

```

接下来需要创建4个`UILabel`，并把它们添加到表视图单元。表视图单元已经有了一个名为`contentView`的`UIView`子视图，用于对它的所有子视图进行分组，就像第4章中我们对`UIView`中的两个开关进行分组一样。因此，我们不用把标签作为子视图直接添加到表视图单元，而是添加到`contentView`。

```
[cell.contentView addSubview:colorValue];
```

其中两个标签包含静态文本。`NameLabel`标签包含`Name`:文本，`colorLabel`标签包含`color`:文本。这些是不需要更改的静态标签。另两个标签用于显示指定行的数据。记住，稍后我们需要检索这些字段，所以要为这两个字段分配值。例如，把常量`kNameValueTag`分配给`nameValue`的`tag`字段：

```
nameValue.tag = kNameValueTag;
```

稍后，我们将使用该标记从单元中检索正确的标签。

创建新单元之后，使用传入的indexPath参数确定表正在请求单元的哪一行，然后使用该行的值为请求的行获取正确的字典。记住，该字典有两个键/值对，一个是名称，一个是颜色。

```
NSUInteger row = [indexPath row];
NSDictionary *rowData = [self.computers objectAtIndex:row];
```

还记得我们前面设置的标记吗？在这里，我们使用那些标记检索需要设置值的标签。

```
UILabel *name = (UILabel *)[cell.contentView viewWithTag:kNameValueTag];
```

检索到标签以后，只需将标签文本设置为从表示此行的字典里获取的一个值。

```
name.text = [rowData objectForKey:@"Name"];
```

编译并运行应用程序，你会得到带有两行数据的行，如图8-13所示。向表视图添加视图比单独使用标准的表视图单元具有更大的灵活性，不过，通过编程创建、定位和添加所有的子视图是一项单调乏味的工作。如果我们能在Interface Builder中设计表视图单元就好了，不是吗？

8.5.3 使用 UITableViewCell 的自定义子类

我们很幸运，可以使用Interface Builder设计表视图单元。我们将使用Interface Builder重新创建与刚才使用代码构建的界面相同的两行界面。要达到此目的，可以创建一个UITableViewCell子类和一个包含表视图单元的新nib文件。然后，当我们需要一个表视图单元来表示一行时，不是向标准的表视图单元添加子视图，而是从nib文件加载子类，并使用将添加的两个输出口来设置名称和颜色。有道理吗？让我们开始付诸行动吧。

在Xcode中右键单击（或Ctrl+单击）Classes文件夹，从出现的**Add**子菜单中选择**New File...**，或者按下⌘N键。新建文件向导出现后，从左侧窗格选择Cocoa Touch Classes，然后从右上窗格选择UITableViewCell子类。单击Next按钮，将新文件命名为CustomCell.m，并确保选中了Also Create“CustomCell.h”复选框。

创建文件之后，在Xcode中右键单击Resources文件夹，再次选择**Add→New File...**。这一次，在新建文件向导的左侧窗格中单击User Interfaces，在右上窗格中选择Empty XIB。当提示输入名称时，输入CustomCell.xib。

1. 创建UITableViewCell子类

创建所有必要的新文件之后，下面让我们继续创建UITableViewCell的新子类。

我们将在子类中使用输出口，这会简化对需要在每一行更改的值的设置。我们可以再次使用标记(tag)，这完全避免了创建子类，而使用这种方式，代码会更加简明并容易阅读，因为我们可以仅通过设置属性来设置每一行单元上的标签，如下所示：

```
cell.nameLabel = @"Foo";
```

单击CustomCell.h，添加以下代码：

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

```

@interface CustomCell : UITableViewCell {
    IBOutlet UILabel *nameLabel;
    IBOutlet UILabel *colorLabel;
}
@property (nonatomic, retain) UILabel *nameLabel;
@property (nonatomic, retain) UILabel *colorLabel;
@end

这就是我们需要添加的所有内容，下面切换到CustomCell.m，并添加两行代码：

#import "CustomCell.h"

@implementation CustomCell
@synthesize nameLabel;
@synthesize colorLabel;
- (id)initWithFrame:(CGRect)frame
{
    reuseIdentifier:(NSString *)reuseIdentifier {
        if (self = [super initWithFrame:frame
                                         reuseIdentifier:reuseIdentifier]) {
            // Initialization code
        }
        return self;
    }
- (void)setSelected:(BOOL)selected animated:(BOOL)animated {
    [super setSelected:selected animated:animated];
    // Configure the view for the selected state
}

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

```

保存上面的代码，并准备好自定义子类。

2. 在Interface Builder中设计表视图单元

下一步，双击CustomCell.xib，在Interface Builder中打开文件。这个主窗口中只有两个图标：File's owner和FirstResponder。在库中找到表视图单元（参见图8-14），然后把它拖动到主窗口中。

确保选中了表视图单元，然后按下⌘4，打开标识检查器。将类从UITableViewCell改为CustomCell。

然后，按下⌘3，打开大小检查器，将表视图单元的高度从44改为65。这会让我们有更多的活动空间。

最后，按下⌘1，打开属性检查器。其中第一个字段是

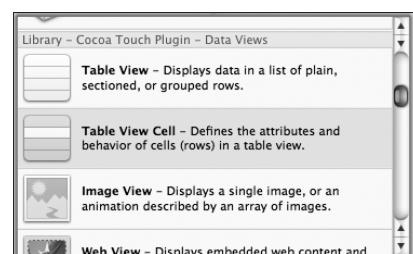


图8-14 库中的表视图单元

`Identifier`, 它是我们在代码中使用过的可重用的标识符。如果记不起来这一内容, 请查阅本章前面的内容并找到 `SimpleTableIdentifier`。将 `Identifier` 设置为 `CustomCellIdentifier`。下一步, 找到名称为 `Accessory` 的弹出按钮, 把 `Detail Disclosure` 改为 `None` (参见图8-15)。扩展图标会在单元中占用一些空间, 但是我们希望整个单元空间都归自己所用。下一章将详细地讨论扩展图标。

记住, 即使 `UITableViewCell` 是 `UIView` 的子类, 它仍然使用内容视图对子视图进行保存和分组。双击 Custom Cell 图标, 将打开一个新的窗口, 你会发现一个标记为 Content View 的灰色虚线圆角矩形 (参见图8-16)。

Interface Builder 通过这种方式告诉你应该添加一些内容, 因此在库中找到 View 并把它拖到 Custom Cell 窗口中。

发布视图之后会发现它的大小和窗口大小不一致, 还要进行调整。选中新的视图, 打开大小检查器。通过将 x 设置为 0, y 设置为 0, w 设置为 320, h 设置为 65, 把 View 的大小和位置改为符合 Custom Cell 窗口的大小。

现在所有项都设置完成了。我们有了一个画布, 可以使用它在 Interface Builder 中设计表视图单元。下面就开始吧。

从库中拖动 4 个标签到 Custom Cell 窗口, 按照图8-17所示进行布局和重命名。要将 `Name:` 和 `Color:` 设置为 黑体, 选中它们并按下 `⌘B`。下一步, 选中右上方的标签, 增加其宽度, 将其右边缘拖到右边的蓝线。按同样方式更改右下位置的标签。这样做的目的是让名称和颜色数据拥有更大的显示空间。

现在, 按下 Control 键, 并将 Custom Cell 图标拖到视图右上位置的标签, 为它指定 `nameLabel` 输出口。然后, 按下 Control 键, 把 Custom Cell 图标再次拖到右下位置的标签, 为它指定 `colorLabel` 输出口。

你可能很奇怪为什么我们没有做任何和 File's Owner 图标有关的事情。原因是根本不需要。我们使用这个表单元显示数据, 不过与用户的所有交互都是通过表视图来完成的, 因此它不需要自己的控制器类。我们实际上只是使用 nib 作为一种模板, 以便可视地设计表单元。

保存 nib 并将其关闭, 然后返回 Xcode。

3. 使用新的表视图单元

要使用我们设计的单元, 必须对 `CellsViewController.m` 中的 `tableView:cellForRowAtIndexPath` 方法做一些大的改动。删除当前的 `tableView:cellForRowAtIndexPath` 方法, 用下面

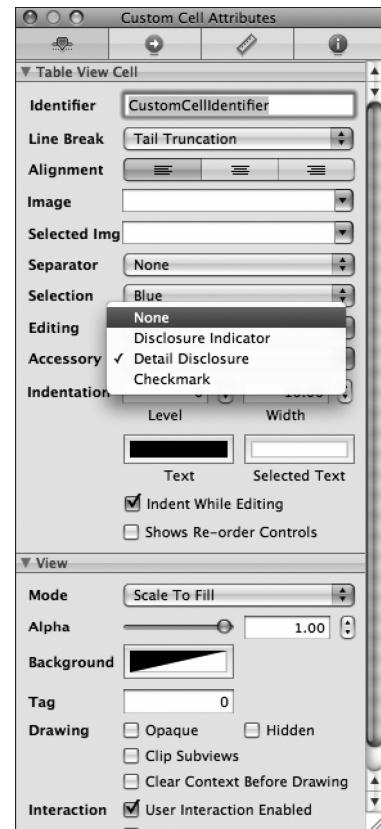


图8-15 关闭扩展图标

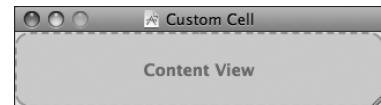


图8-16 表视图单元的窗口



图8-17 表视图单元的设计

的新版本代替：

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CustomCellIdentifier = @"CustomCellIdentifier";

    CustomCell *cell = (CustomCell *)[tableView
        dequeueReusableCellWithIdentifier:CustomCellIdentifier];
    if (cell == nil)
    {
        NSArray *nib = [[NSBundle mainBundle] loadNibNamed:@"CustomCe"
            owner:self options:nil];
        cell = [nib objectAtIndex:1];
    }
    NSUInteger row = [indexPath row];
    NSDictionary *rowData = [self.computers objectAtIndex:row];
    cell.colorLabel.text = [rowData objectForKey:@"Color"];
    cell.nameLabel.text = [rowData objectForKey:@"Name"];
    return cell;
}
```

更改CellsViewController.m时，在接近顶部的位置添加此行：

```
#import "CustomCell.h"
```

由于我们在nib文件中设计了表视图单元，因此，如果没有可重用的单元，只需从nib文件中加载即可。我们在`objectAtIndex:`调用中使用索引值1而不是0，因为对象0是文件的所有者，它并不是我们想要的。First Responder不是由`loadNibName:owner:options:`返回的，因此表视图单元的索引值为1。

下面还要添加另一项内容。由于更改了表视图单元默认的高度值，我们必须告知表视图这一事实；否则，表视图单元不会留下足够的显示空间。为此，在CellsViewController.m中添加以下委托方法，将其添加到`@end`之前：

```
- (CGFloat)tableView:(UITableView *)tableView
    heightForRowAtIndexPath:(NSIndexPath *)indexPath
{
    return
}
```

然后，我们不能从单元获得这个值，因为此委托方法可能在单元存在之前被调用，因此我们必须硬编码这个值。把这个常量定义添加到CustomCell.h的顶部，然后删除那些不再需要的`tag`常量。

```
#define kTableViewRowHeight 66
#define kNameValueTag 1
#define kColorValueTag 2
```

这就好了。编译并运行程序。现在，两行的表单元都是基于Interface Builder设计技术的。

8.6 分组分区和索引分区

下一个项目将探讨表的另一个基本内容。仍然使用一个表视图（没有分层），不过我们将把数据分为几个分区。再次使用基于视图的应用程序模板创建一个新的Xcode项目，这一次将它命名为Sections。

8.6.1 构建视图

打开Classes和Resources文件夹，在Interface Biulder中双击SectionsViewController.xib，打开文件。与之前一样，把表视图拖到View窗口中。然后按下⌘2，将数据源和委托连接到File's Owner图标。

下一步，确保选中表视图，按下⌘1，打开属性检查器。把表视图的Style从Indexed改为Grouped（参见图8-18）。这里可以提示一下，我们在本章开头讨论过索引类型和分组类型之间的差别。保存并返回Xcode。

8.6.2 导入数据

要完成此项目需要大量的数据。我们提供了另一个属性列表，为你节省几个小时敲键盘的时间。从本书随附的项目归档文件的08 Sections文件夹中找到名为sortednames.plist文件，把它添加到项目的Resource文件夹中。

完成添加以后，单击sortednames.plist，看一下它到底是什么（参见图8-19）。它是包含字典的一个属性列表，其中字母表中的每个字母都有一个条目。每个字母下面是以该字母开头的名称列表。

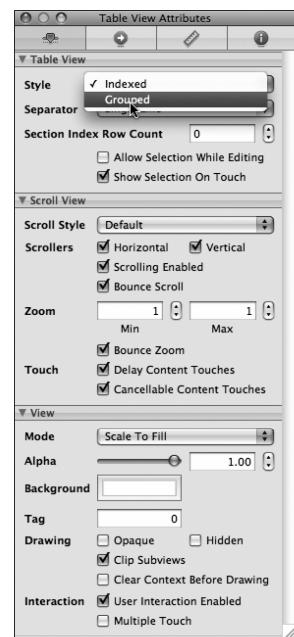


图8-18 表视图的属性检查器

sortednames.plist		
	Type	Value
Root	Dictionary	(26 items)
A	Array	(245 items)
B	Array	(23 items)
C	Array	(141 items)
D	Array	(117 items)
E	Array	(92 items)
F	Array	(27 items)
G	Array	(64 items)
H	Array	(51 items)
I	Array	(35 items)
J	Array	(206 items)
K	Array	(159 items)
L	Array	(108 items)
M	Array	(169 items)
N	Array	(51 items)
O	Array	(13 items)
P	Array	(39 items)
Q	Array	(7 items)
R	Array	(104 items)
S	Array	(112 items)
T	Array	(80 items)
U	Array	(2 items)
V	Array	(20 items)
W	Array	(17 items)
X	Array	(5 items)
Y	Array	(19 items)
Z	Array	(24 items)
Item 1	String	Zachariah
Item 2	String	Zachary
Item 3	String	Zachery
Item 4	String	Zack
Item 5	String	Zackary
Item 6	String	Zackery

图8-19 sortednames.plist属性列表文件

我们将使用这个属性列表中的数据填充表视图，并为每个字母创建一个分区。

8.6.3 实现控制器

单击SectionsViewController.h文件，添加NSDictionary和NSArray实例变量及其相应的属性声明。字典将保存所有数据。数组将保存以字母顺序排序的分区。还需要让类遵循UITableViewDataSource和UITableViewDelegate协议：

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

@interface SectionsViewController : UIViewController
<UITableViewDataSource, UITableViewDelegate>
{
    NSDictionary *names;
    NSArray      *keys;
}
```

```
@property (nonatomic, retain) NSDictionary *names;
@property (nonatomic, retain) NSArray *keys;
@end
```

现在，切换到SectionsViewController.m，并添加以下代码：

```
#import "SectionsViewController.h"

@implementation SectionsViewController
@synthesize names;
@synthesize keys;
#pragma mark -
#pragma mark UIViewController Methods

- (void)viewDidLoad {
    NSString *path = [[NSBundle mainBundle] pathForResource:@"sortednames"
        ofType:@"plist"];
    NSDictionary *dict = [[NSDictionary alloc]
        initWithContentsOfFile:path];
    self.names = dict;
    [dict release];

    NSArray *array = [[names allKeys] sortedArrayUsingSelector:
        @selector(compare:)];
    self.keys = array;
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
```

```
- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [names release];
    [keys release];
    [super dealloc];
}

#pragma mark -
#pragma mark Table View Data Source Methods
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return [keys count];
}
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    NSString *key = [keys objectAtIndex:section];
    NSArray *nameSection = [names objectForKey:key];
    return [nameSection count];
}
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSUInteger section = [indexPath section];
    NSUInteger row = [indexPath row];

    NSString *key = [keys objectAtIndex:section];
    NSArray *nameSection = [names objectForKey:key];

    static NSString *SectionsTableIdentifier = @"SectionsTableIdentifier";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        SectionsTableIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
            reuseIdentifier: SectionsTableIdentifier ] autorelease];
    }

    cell.textLabel = [nameSection objectAtIndex:row];
    return cell;
}
- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section
{
```

```

NSString *key = [keys objectAtIndex:section];
return key;
}
@end

```

上面的大部分代码和我们以前看到的没有多大区别。在`viewDidLoad`方法中，从添加到项目的属性列表中创建了一个`NSDictionary`实例，并为它指定名称`names`。然后，获取字典中的所有键，按照字典中字母表的顺序对键值进行排序，将会得到一个有序的`NSArray`。记住，`NSDictionary`使用字母表中的字母作为它的键，因此这个数组将拥有26个字母，按照顺序从A到Z。我们将通过这个数组来了解分区。

下面看一下数据源方法。我们添加的第一个方法指定了分区的数量。上次没有实现此方法是因为默认设置1已经很合适了。这一次，我们需要告诉表视图字典中的每个键都有一个分区。

```

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return [keys count];
}

```

第二个方法用于计算特定分区中的行数。上一次，只有一个分区，所以只返回数组中拥有的行数。这次，我们将以分区为单位返回行数。可以检索对应于正被考虑的分区的数组，并从该数组返回行的数量，从而达到目的。

```

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    NSString *key = [keys objectAtIndex:section];
    NSArray *nameSection = [names objectForKey:key];
    return [nameSection count];
}

```

在`tableView:cellForRowIndexPath:`方法中，必须从索引路径获取分区和行，用它们来确定要使用哪一个值。分区会告诉我们从名字字典中取出哪个数组，然后可以使用行指出要使用该数组中的哪个值。方法中的其他内容基本上和Simple Table应用程序代码中的相同。

可以通过`tableView:titleForHeaderInSection`方法为每个分区指定一个可选的标题值，然后只返回这一组的字母就可以了。

```

- (NSString *)tableView:(UITableView *)tableView
    titleForHeaderInSection:(NSInteger)section
{
    NSString *key = [keys objectAtIndex:section];
    return key;
}

```

最后，由于在`SectionViewController.m`中引用了`CGRectZero`，我们需要连接到Core Graphics框架。单击Groups & Files窗格中的Frameworks文件夹，从Project菜单中选择Add to Project。如果你忘记了如何导航到框架，请返回到第5章的末尾部分，该部分详细介绍了框架的位置。

现在可以编译并运行项目，然后慢慢地欣赏它了。记住，我们已经把表的样式改为Grouped

了，因此最终拥有一个带有26个分区的分组表，如图8-20所示。

作为比较，让我们把表视图再次改为索引风格，然后看一下带有多个分区的索引表视图是什么样子。在Interface Builder中双击SectionViewController.xib，打开该文件。选中表视图，使用属性检查器将视图改为Indexed。保存并返回Xcode，编译并运行，得到的是相同的数据和不一样的外观（参见图8-21）。

8.6.4 添加索引

当前表的一个问题是行数太多了。此列表中有2000个名称，要找到Zacharian或Zebedian，你的手指会非常累，更别说Zojirishu了。

这个问题的一个解决方案是，在表视图的右侧添加一个索引。既然我们已经把表视图样式改成了索引类型，要添加一个索引相对来说也很容易。在SectionsViewController.m文件尾部，@end前面添加如下方法：

```
- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)tableView
{
    return keys;
}
```

这就完成了。在这个方法中，委托请求一个值数组在索引中显示。必须使表视图中的多个分区使用索引，而且此数组中的条目必须对应于这些分区。返回的数组拥有的条目数必须与拥有的分区数相同，且值必须对应于适当的分区。也就是说，此数组中的第一项带给用户的是第一个分区，即分区0。

再次编译并运行，你将得到一个很好的索引（参见图8-22）。

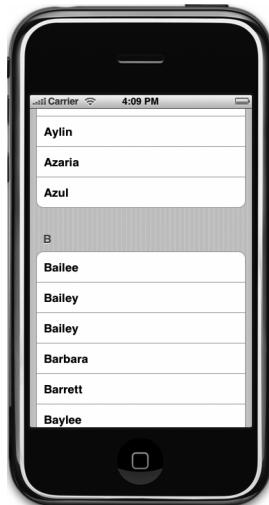


图8-20 有多个分区的分组表

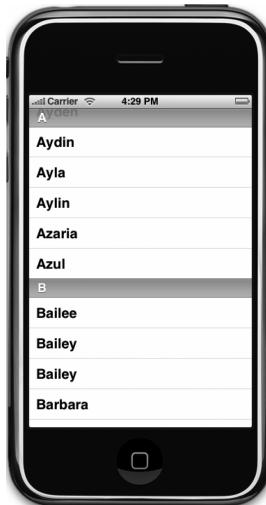


图8-21 带有分区的索引表视图



图8-22 带有一个索引的索引表视图

8.7 实现搜索栏

索引是很有用的，即便如此，此处我们仍然有非常多的名称。例如，如果要查看名称Arabella是否存在于列表中，使用索引之后仍然需要拖动滚动条。如果能够通过指定搜索项简化该列表就好了，对不对？这样对用户更友好。那只是一点额外的工作，并不是特别糟糕。我们将实现一个标准的iPhone搜索栏，如图8-23所示。

8.7.1 重新考虑设计

在我们开始着手做之前，需要考虑一下应用程序是如何工作的。当前，我们拥有一个包含多个数组的字典，其中字母表中的每个字母都占有一个数组。该字典是不可改变的，这就意味着不能从字典添加或删除值，它包含的数组也是如此。当用户取消搜索或者更改搜索项时，还必须能够返回到源数据集。

我们能做的就是创建两个新的字典：一个包含完整数据集的不可改变的字典、一个可以从中删除行的可变的字典副本。委托和数据源将从可变字典进行读取，当搜索标准更改或者取消搜索时，可以从不可改变的字典刷新可变字典。听起来像个好计划，下面就开始吧。

注意 下一个项目有些复杂，阅读得太快让你感到有些不知所措。如果某些概念让你感到头疼，请查阅Mark Dalrymple和Scott Knaster所著的*Learn Object-C* (Apress, 2009)，阅读此书中与类别和易变性有关的内容。

8.7.2 深层可变副本

现在存在一个问题，`NSDictionary`遵循`NSMutableCopying`协议，该协议返回一个`NSMutableDictionary`，但是这个方法创建的是浅副本。也就是说，调用`mutableCopy`方法时，它将创建一个新的`NSMutableDictionary`对象，该对象拥有源字典所拥有的所有对象。它们并不是副本，而是相同的实际对象。如果我们所处理的字典仅存储字符串，这会很好，因为从副本中删除一个值不会对源字典产生任何影响。然而，由于字典中存有数组，如果我们从副本的数组中删除对象，这些对象也将从源字典的数组中被删除，因为副本和源都指向相同的对象。

要解决这一问题，需要为存有数组的字典创建一个深层可变副本。这并不难，不过我们应该把这项功能放在哪儿呢？

如果说“在类别中”，那么好极了，你的想法是正确的。如果还没有想到，别担心，习惯这种语言需要花费一定的时间。通过类别可以向现有对象添加附加方法，而不需要子类化这些对象。类别经常会被Objective-C的新手忽略，因为这些特性是大多数其他语言所没有的。

通过类别，我们可以向`NSDictionary`添加一个方法来实现深层副本，返回的`NSMutable-`



图8-23 带有搜索栏的应用程序

Dictionary拥有相同的数据，但不包含相同的实际对象。

在项目窗口中，选择Classes文件夹，然后按下⌘N创建一个新文件。向导出现之后，从最底端的左侧选择Other。然而，类别没有文件模板，我们要创建几个空文件来保存它。选择Empty File图标，将第一个文件命名为NSDictionary-MutableDeepCopy.h。重复此过程，将第二个文件命名为NSDictionary-MutableDeepCopy.m。

提示 创建类别所需的两个文件的一个更快的方法就是，选择**NSObject**子类模板，然后删除文件内容。此选项将同时提供头文件和实现文件，为你省了一步。

将下面的代码添加到NSDictionary-MutableDeepCopy.h中：

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

```
@interface NSDictionary(MutableDeepCopy)
- (NSMutableDictionary *)mutableDeepCopy;
@end
```

切换到NSDictionary-MutableDeepCopy.m，并添加以下实现：

```
#import "NSDictionary-MutableDeepCopy.h"
```

```
@implementation NSDictionary (MutableDeepCopy)
- (NSMutableDictionary *) mutableDeepCopy
{
    NSMutableDictionary *ret = [NSMutableDictionary dictionaryWithCapacity:
        [self count]];
    NSArray *keys = [self allKeys];
    for (id key in keys)
    {
        id oneValue = [self valueForKey:key];
        id oneCopy = nil;

        if ([oneValue respondsToSelector:@selector(mutableDeepCopy)])
            oneCopy = [oneValue mutableDeepCopy];
        else if ([oneValue respondsToSelector:@selector(mutableCopy)])
            oneCopy = [oneValue mutableCopy];
        if (oneCopy == nil)
            oneCopy = [oneValue copy];
        [ret setValue:oneCopy forKey:key];
    }
    return ret;
}
@end
```

此方法创建一个新的可变字典，然后在源字典中的所有键中进行迭代，为它遇到的每个数组创建可变副本。由于此方法将表现得好像它是**NSDictionary**的一部分，所以对**self**的任何引用都是对调用到此方法上的字典的一个引用。此方法首先尝试创建深层可变副本，如果对象没有响应**mutableDeepCopy**消息，那么它将尝试创建可变副本，如果对象没有响应**mutableCopy**消息，它就

回过头再创建常规副本，以确保对字典中包含的所有对象都创建了副本。通过这种方式，如果我们有一个包含字典（或支持深层可变副本的其他对象）的字典，则也将对所包含的内容创建深层副本。

对于大多数读者来说，这可能是第一次在Objective-C中看到如下语法：

```
for (id key in keys)
```

Objective-C 2.0还有一种新特性，叫做快速枚举。快速枚举是NSEnumerator的语言级替代，《Learn Objective C》一书对它进行了介绍。可以通过快速枚举对集合（如NSArray）进行迭代，从而避免了创建附加对象的麻烦。

所有已交付的Cocoa集合类，包括NSDictionary、NSArray和NSSet，都支持快速枚举。而且可以在任何需要的时候使用此语法对集合进行迭代。它将确保你获得效率最高的循环。

你可能注意到了，我们正在使用一个便利类方法创建ret字典，虽然我们以前曾警告过不要使用不必要的便利方法。使用便利方法的问题是，由便利方法返回的对象被放到自动释放池，并至少要等待当前事件循环结束，即使它的保留计数下跌到零。很显然，这里的问题是内存的低使用率。

在这种情况下，当方法结束后返回值时，必须在该方法上调用autorelease，以确保它依然为调用此方法的代码而存在。因为不管怎么样，它也要到自动释放池中，我们可以利用便利方法，省去了键入一行或两行代码的工作。

如果在其他类中包含NSDictionary-MutableDeepCopy.h头文件，我们将能够在任何喜欢的NSDictionary对象上调用mutableDeepCopy。现在让我们使用这一功能。

8.7.3 更新控制器头文件

下一步，我们需要向控制器头文件添加一些输出口。表视图需要一个输出口。目前为止，我们还没有用到数据源方法之外指向表视图的指针，现在就需要一个，因为我们需要根据搜索的结果通知表重新加载它自己。

我们还需要一个指向搜索栏的输出口，它是一个用于搜索的控件。除了这两个输出口之外，还将需要一个附加字典。现有的字典和数组都是不可改变的对象，我们需要把它们改为相应的可变版本，这样，NSArray就变成了NSMutableArray，NSDictionary就变成了NSMutableDictionary。

在控制器中不再需要任何新的操作方法，不过我们需要几个新的方法。目前，仅对它们进行声明，输入代码之后再对它们进行详细的讨论。

还需要让类遵循UISearchBarDelegate协议。除了充当表视图的委托之外，我们还需要让它充当搜索栏的委托。

在SectionsViewController.h文件中做如下更改：

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

@interface SectionsViewController : UIViewController
<UITableViewDataSource, UITableViewDelegate, UISearchBarDelegate>
```

```

{
    IBOutlet      UITableView *table;
    IBOutlet      UISearchBar *search;
    NSDictionary *allNames;
    NSMutableDictionary *names;
    NSMutableArray  *keys;
    NSDictionary  *names;
    NSArray       *keys;

}

@property (nonatomic, retain) NSDictionary *names;
@property (nonatomic, retain) NSArray *keys;
@property (nonatomic, retain) UITableView *table;
@property (nonatomic, retain) UISearchBar *search;
@property (nonatomic, retain) NSDictionary *allNames;
@property (nonatomic, retain) NSMutableDictionary *names;
@property (nonatomic, retain) NSMutableArray *keys;
- (void)resetSearch;
- (void)handleSearchForTerm:(NSString *)searchTerm;
@end

```

这就是我们所做的更改。`table`输出口将指向表视图；`search`输出口将指向搜索栏；`allNames`字典将存有所有数据集；`names`字典将存有那些与当前搜索标准匹配的数据集；`keys`将存有索引值和分区名称。如果清楚地理解了这些内容，下面让我们在Interface Builder中修改视图。

8

8.7.4 修改视图

在Interface Builder中双击SectionsViewController. xib，打开该文件。打开之后，选中表视图，然后使用顶部的调整大小的工具将视图缩短一些，为顶部的搜索栏腾出一些空间。不要担心如何精确地进行调整，你在一秒钟之内就会调整到最好。提示一下，如果你不知道如何显示调整大小工具，可以将nib主窗口改为列表模式，然后双击Table View，这会选中Table View并显示调整大小的工具。

下一步，从库中选择Search Bar（参见图8-24），然后把它添加到视图的顶部。

调整搜索栏和表视图，使它们看起来好看一些。表视图的顶部应该正好挨着搜索栏的底部，这两个控件应该正好占据视图的全部空间，如图8-25所示。

现在，按下Control键并将File's Owner图标拖到表视图，然后选中表输出口。同样对搜索栏重复上述操作，然后选中搜索输出。单击搜索栏，按下⌘键，打开属性检查器，如图8-26所示。

选中Shows Cancel Button复选框。在Placeholder字段中键入search，搜索字段的右侧会出现一个Cancel按钮，用户可以使用此按钮取消搜索。搜索字段中将出现以灰色字母显示的占位符文本search。

按下⌘2，转到连接检查器，从delegate连接拖到File's Owner图标，以告知搜索栏视图控制器也是搜索栏的委托。

这就是所有我们所需要的，因此一定要进行保存。下面让我们回到Xcode。

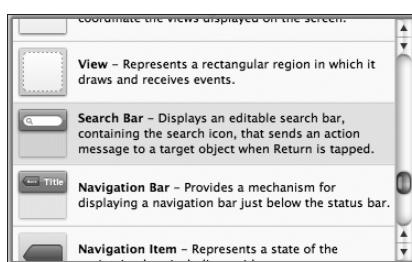


图8-24 库中的Search Bar



图8-25 带有表视图和搜索栏的新视图

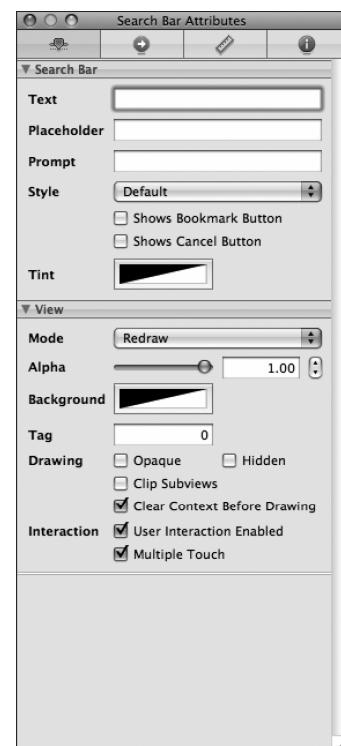


图8-26 搜索栏的属性检查器

8.7.5 修改控制器实现

对搜索栏进行的更改非常大。对SectionsViewController.m做如下更改，然后快速返回，查看这些更改。

```
#import "SectionsViewController.h"
#import "NSDictionary-MutableDeepCopy.h"

@implementation SectionsViewController
@synthesize names;
@synthesize keys;
@synthesize table;
@synthesize search;
@synthesize allNames;
#pragma mark -
#pragma mark Custom Methods
- (void)resetSearch
{
    self.names = [self.allNames mutableDeepCopy];
    NSMutableArray *keyArray = [[NSMutableArray alloc] init];
}
```

```

[keyArray addObjectFromArray:[[self.allNames allKeys]
    sortedArrayUsingSelector:@selector(compare:)]];
self.keys = keyArray;
[keyArray release];
}
- (void)handleSearchForTerm:(NSString *)searchTerm
{
    NSMutableArray *sectionsToRemove = [[NSMutableArray alloc] init];
    [self resetSearch];

    for (NSString *key in self.keys) {
        NSMutableArray *array = [names valueForKey:key];
        NSMutableArray *toRemove = [[NSMutableArray alloc] init];
        for (NSString *name in array) {
            if ([name rangeOfString:searchTerm
                options:NSCaseInsensitiveSearch].location == NSNotFound)
                [toRemove addObject:name];
        }

        if ([array count] == [toRemove count])
            [sectionsToRemove addObject:key];
        [array removeObjectsInArray:toRemove];
        [toRemove release];
    }
    [self.keys removeObjectsInArray:sectionsToRemove];
    [sectionsToRemove release];
    [table reloadData];
}

#pragma mark -
#pragma mark UIViewController Methods

- (void)viewDidLoad {
    NSString *path = [[NSBundle mainBundle] pathForResource:@"sortednames"
        ofType:@"plist"];
    NSDictionary *dict = [[NSDictionary alloc]
        initWithContentsOfFile:path];
    self.names = dict;
    self.allNames = dict;

    [dict release];

    NSArray *array = [[names allKeys] sortedArrayUsingSelector:
        @selector(compare:)];
    self.keys = array;

    [self resetSearch];
    search.autocapitalizationType = UITextAutocapitalizationTypeNone;
    search.autocorrectionType = UITextAutocorrectionTypeNo;
}

```

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [table release];
    [search release];
    [allNames release];
    [keys release];
    [names release];
    [super dealloc];
}
#pragma mark -
#pragma mark Table View Data Source Methods
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return [keys count];
    return ([keys count] > 0) ? [keys count] : 1;
}
- (NSInteger)tableView:(UITableView *)aTableView
    numberOfRowsInSection:(NSInteger)section
{
    if ([keys count] == 0)
        return 0;
    NSString *key = [keys objectAtIndex:section];
    NSArray *nameSection = [names objectForKey:key];
    return [nameSection count];
}
- (UITableViewCell *)tableView:(UITableView *)aTableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSUInteger section = [indexPath section];
    NSUInteger row = [indexPath row];

    NSString *key = [keys objectAtIndex:section];
    NSArray *nameSection = [names objectForKey:key];

    static NSString *sectionsTableIdentifier = @"sectionsTableIdentifier";
}
```

```
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:  
    sectionsTableIdentifier];  
if (cell == nil) {  
    cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero  
        reuseIdentifier: sectionsTableIdentifier] autorelease];  
}  
  
cell.textLabel.text = [nameSection objectAtIndex:row];  
return cell;  
}  
- (NSString *)tableView:(UITableView *)tableView  
    titleForHeaderInSection:(NSInteger)section  
{  
    if ([keys count] == 0)  
        return @"";  
  
    NSString *key = [keys objectAtIndex:section];  
    return key;  
}  
- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)tableView  
{  
    return keys;  
}  
#pragma mark -  
#pragma mark Table View Delegate Methods  
- (NSIndexPath *)tableView:(UITableView *)tableView  
    willSelectRowAtIndexPath:(NSIndexPath *)indexPath  
{  
    [search resignFirstResponder];  
    return indexPath;  
}  
#pragma mark -  
#pragma mark Search Bar Delegate Methods  
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar  
{  
    NSString *searchTerm = [searchBar text];  
    [self handleSearchForTerm:searchTerm];  
}  
  
- (void)searchBar:(UISearchBar *)searchBar  
    textDidChange:(NSString *)searchTerm  
{  
    if ([searchTerm length] == 0)  
    {  
        [self resetSearch];  
        [table reloadData];  
        return;  
    }  
}
```

```

        [self handleSearchForTerm:searchTerm];

    }

- (void)searchBarCancelButtonClicked:(UISearchBar *)searchBar
{
    search.text = @"";
    [self resetSearch];
    [table reloadData];
    [searchBar resignFirstResponder];
}

@end

```

1. 从allNames复制数据

键入上述所有代码之后，你还跟得上进度吗？让我们停下来看一看刚才的代码。先看一下添加的两个新方法。这是第一个方法：

```

- (void)resetSearch
{
    self.names = [self.allNames mutableDeepCopy];
    NSMutableArray *keyArray = [[NSMutableArray alloc] init];
    [keyArray addObjectsFromArray:[[self.allNames allKeys]
        sortedArrayUsingSelector:@selector(compare:)]];
    self.keys = keyArray;
    [keyArray release];
}

```

取消搜索或更改搜索条件时将调用此方法。它所做的的是创建`allNames`的可变副本，将它赋值为`names`，然后刷新`keys`数组，使它包含字母表中的所有字母。我们必须刷新`keys`数组，因为如果某搜索排除了某分区中的所有值，那么还需要除去这一分区。否则，屏幕将被标题和空的分区充满，这看起来很糟糕。我们也不希望为不存在的内容提供索引，因此根据搜索短语挑选名称时，还需要去除空的分区。

2. 实现搜索

另一个方法实现了实际的搜索：

```

- (void)handleSearchForTerm:(NSString *)searchTerm
{
    NSMutableArray *sectionsToRemove = [[NSMutableArray alloc] init];
    [self resetSearch];

    for (NSString *key in self.keys)
    {

        NSMutableArray *array = [names valueForKey:key];
        NSMutableArray *toRemove = [[NSMutableArray alloc] init];
        for (NSString *name in array) {
            if ([name rangeOfString:searchTerm
                options:NSCaseInsensitiveSearch].location == NSNotFound)
                [toRemove addObject:name];
        }
    }
}

```

```

if ([array count] == [toRemove count])
    [sectionsToRemove addObject:key];
[array removeObjectsInArray:toRemove];
[toRemove release];
}
[self.keys removeObjectInArray:sectionsToRemove];
[sectionsToRemove release];
[table reloadData];
}

```

虽然我们将在搜索栏委托方法中进行搜索，但仍然要把`handleSearchForTerm:`拖到自己的方法中，因为我们将需要在两个不同的委托方法中使用相同的功能。通过在`handleSearchForTerm:`方法中嵌入搜索，我们将功能固定在一个位置，这样便于维护，然后只在需要时调用这个新的方法即可。

这是此部分真正有趣的地方，让我们把这个方法分为几小段来看一下。首先，创建一个数组，它将存有我们找到的空分区。后面使用此数组删除这些空分区，因为在集合中进行迭代时，从该集合中删除对象是不安全的。由于我们正使用快速枚举，这样做将引发异常。因此，我们不能在键中进行迭代时删除键，就把将要删除的分区存储在一个数组中，在完成所有枚举之后一次删除所有的对象。分配数组之后，重置搜索：

```

NSMutableArray *sectionsToRemove = [[NSMutableArray alloc] init];
[self resetSearch];

```

下一步，枚举新存储的`keys`数组中的所有键。

```

for (NSString *key in self.keys)
{

```

每次进行循环时，获取对应于当前键的名称数组，并创建存有需要从`names`数组中删除的值的数组。记住，我们删除的是名称和分区，因此需要知道哪些键是空的，以及哪些名称与搜索条件不匹配。

```

NSMutableArray *array = [names valueForKey:key];
NSMutableArray *toRemove = [[NSMutableArray alloc] init];

```

接下来，在当前数组中对所有名称进行迭代。因此，如果当前正处理键“A”，那么此循环将迭代所有以“A”开头的名称。

```

for (NSString *name in array)
{

```

此循环使用了一个返回字符串中子字符串位置的`NSString`方法。通过指定`NSCaseInsensitiveSearch`选项，表明我们不关心搜索短语的大小写。也就是说，“A”相当于“a”。此方法返回的值是一个`NSRange`结构，它带有两个成员：`location`和`length`。如果没有找到搜索短语，`location`将被设置为`NSNotFound`，因此只要检查有没有`NSNotFound`就可以了。如果返回的`NSRange`包含`NSNotFound`，就把名称添加到将要删除的对象数组中。

```

if ([name rangeOfString:searchTerm
options:NSCaseInsensitiveSearch].location == NSNotFound)

```

```
[toRemove addObject:name];
}
```

对给定字母的所有名称迭代完成之后，检查将要删除的名称数组的长度是不是和名称数组的长度相同，如果相同，则这个分区就是空的，我们将把它添加到键的数组中，以备将来删除。

```
if ([array count] == [toRemove count])
    [sectionsToRemove addObject:key];
```

下一步，从此分区的数组中删除不匹配的名称，然后释放用于存储名称的数组。尽可能地避免在这样的循环中使用便利方法很重要，因为它们会在每次循环中将一些东西放入自动释放池。然而，直到完成循环，自动释放池才会充满。

```
[array removeObjectsInArray:toRemove];
[toRemove release];
}
```

最后，删除空分区，释放用于存储空分区的数组，并告知表重新加载数据。

```
[self.keys removeObjectFromArray:sectionsToRemove];
[sectionsToRemove release];
[table reloadData];
}
```

3. 修改viewDidLoad

在viewDidLoad中，我们进行了一些更改。首先，把属性列表加载到allNames字典而不是names字典，删除加载keys数组的代码，因为现在使用resetSearch方法完成加载。然后调用resetSearch方法，它填充names可变字典和keys数组。下一步，在搜索栏上发起两个调用，以进行在Interface Builder中不能设置的一些配置，因为受影响的设置在属性检查器中不可用：

```
search.autocapitalizationType = UITextAutocapitalizationTypeNone;
search.autocorrectionType = UITextAutocorrectionTypeNo;
```

由于搜索是不区分大小写的，因此不需要把用户键入到搜索栏的内容变为大写。我们还不希望使用自动校正，因为许多搜索短语可能是名称的一部分，我们不希望搜索对它们进行自动校正。

4. 修改数据源方法

如果跳转到数据源方法，你会发现我们对该方法做了一些微小的修改。因为names字典和keys数组依然用于提供数据源，这些方法基本上和以前相同。我们必须说明这样一个事实：表视图始终拥有分区的基本部分，而且搜索可能把所有名称排除在所有分区之外。因此，我们添加了一些代码来检查删除了所有分区的情况，在那种情况下，我们为表视图提供了一个没有行且只有一个空白名称的分区。这避免了所有问题，不会给用户任何错误的反馈。

5. 添加表视图委托方法

在数据源方法下面添加一个委托方法。如果用户在使用搜索栏时单击一行，我们希望键盘不再起作用。这是通过实现

```
- (NSIndexPath *)tableView:(UITableView *)tableView
    willSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [search resignFirstResponder];
    return indexPath;
}
```

6. 添加搜索栏委托方法

搜索栏有许多在其委托上调用的方法。当用户点击键盘上的返回按钮或搜索键时，将调用**searchBarSearchButtonClicked:**。此方法从搜索栏获取搜索短语，并调用我们的搜索方法，这个搜索方法将删除**names**中不匹配的名称和**keys**中的空分区。

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
    NSString *searchTerm = [searchBar text];
    [self handleSearchForTerm:searchTerm];
}
```

每次使用搜索栏的时候都应该实现**searchBarSearchButtonClicked:**方法。除此之外，我们还实现了另一个搜索栏委托方法，不过实现这个方法需要谨慎一些。这个方法用于实现现场搜索(*live search*)。每次更改搜索短语时，不管用户是否选中了搜索按钮或点击了返回，我们都重新进行搜索。这种行为具有非常高的用户友好性，因为用户在键入时可以看见结果的改变。如果用户在键入第3个字符后减少了足够长的列表，那么他们可以停止键入，然后选择想要的行。

实现现场搜索会减弱应用程序的性能，尤其是在显示图像或拥有复杂数据模型的时候。在这种情况下，如果只有2000个字符串，没有图像或扩展图标，程序会运行得非常好，即使是在第一代iPhone或iPod Touch上。

不要以为仿真器中的高性能可以转换为你的设备上的高性能。如果你打算实现这样的现场搜索，就需要在实际硬件上做大量测试，以确保应用程序保持响应。拿不定主意的时候不要使用它。或许用户很乐意敲击搜索按钮呢。

现在，你已经获得了足够的警告，下面的代码将处理一个现场搜索。实现搜索栏委托方法**searchBar:textDidChange:**的代码为如下所示：

```
- (void)searchBar:(UISearchBar *)searchBar
    textDidChange:(NSString *)searchTerm
{
    if ([searchTerm length] == 0)
    {
        [self resetSearch];
        [table reloadData];
        return;
    }
    [self handleSearchForTerm:searchTerm];
}
```

注意，这里我们查找一个空的字符串。如果字符串为空，那么所有名称都将与它匹配，因此我们只要重置搜索并重新加载数据就可以了，而不需要枚举所有名称。

最后，我们实现一个方法，当用户在搜索栏上单击Cancel按钮时，我们能得到通知。

```
- (void)searchBarCancelButtonClicked:(UISearchBar *)searchBar
{
    search.text = @"";
    [self resetSearch];
    [table reloadData];
    [searchBar resignFirstResponder];
}
```

当用户单击Cancel时，程序会将搜索短语设置为空字符串，然后重置搜索，并重新加载数据以显示所有名称。此外，还需要让搜索栏生成第一响应者状态，这样键盘就不再起作用，以便于用户重新处理表视图。

8.8 小结

感觉怎么样？这一章的内容极其重要，你已经学习了很多！你应该对平面表（flat table）的工作方式有了非常好的理解，并了解了如何自定义表和表视图单元，以及如何配置表视图。你还了解了如何实现搜索栏，在任何呈现大量数据的iPhone应用程序中，这都是一个至关重要的工具。要确保真正理解了本章中的所有内容，因为本章是后面章节的基础。

下一章将继续介绍表视图，你将学习如何使用表视图呈现分层数据。你将了解如何创建内容视图，以便用户能够编辑表视图中选中的数据，以及如何在表中呈现检验表，在表的行中嵌入控件和删除行。

使用 Quartz 和 OpenGL 绘图

12

到目前为止，本书中的所有应用程序都是通过UIKit框架中的视图和控件来构造的。借助这些常备组件，我们可以执行许多操作，并且可以构造各式各样的应用程序界面。但是，如果不能高瞻远瞩，某些应用程序会无法完全实现。例如，有时应用程序需要能够进行自定义绘图。幸而，我们可以依靠两个不同的库来满足我们的绘图需要。一个库是Quartz 2D，它是Core Graphics框架的一部分；另一个库是OpenGL ES，它是跨平台的图形库。OpenGL ES是跨平台图形库OpenGL的简化版。OpenGL ES是OpenGL的一个子集，OpenGL ES是专门为iPhone之类的嵌入式系统（因此缩写为字母“ES”）设计的。本章将介绍这两个功能强大的图形环境。我们将在这两种环境中构建示例应用程序，并尝试了解什么时候使用哪个环境。

12.1 图形世界的两个视图

尽管Quartz和OpenGL有许多共性，但它们之间存在明显差别。Quartz是一组函数、数据类型以及对象，专门设计用于直接在内存中对视图或图像进行绘制。

Quartz将正在绘制的视图或图像视为一个虚拟的画布，并遵循所谓的**绘画者模型**。这只是一个奇特的方式，之所以这么说，是因为应用绘图命令的方式很大程度上与将颜料应用于画布的方式相同。如果绘画者将整个画布涂为红色，然后将画布的下半部分涂为蓝色，那么画布将变为一半红色、一半蓝色或紫色。如果颜料是不透明的，应该为蓝色，如果颜料是半透明的，应该为紫色。

Quartz的虚拟画布采用相同的工作方式。如果将整个视图涂为红色，然后将视图下半部分涂为蓝色，你将拥有一个一半红色、一半蓝色或紫色的视图，这取决于第二个绘图操作是完全不透明的还是部分透明的。每个绘图操作都将被应用于画布，并且处于之前所有绘图操作之上。

另一方面，OpenGL ES以状态机的形式实现。这个概念可能有点不好理解，因为不能将其归结为一个简单的比喻，如在虚拟画布上绘画。OpenGL ES不允许执行直接影响视图、窗口或图像的操作，它维护一个虚拟的三维世界。当向这个世界中添加对象时，OpenGL会跟踪所有对象的状态。虽然OpenGL ES没有提供虚拟画布，但是却提供了一个进入其世界的虚拟窗口。可以向该世界中添加对象并定义虚拟窗口相对于该世界的位置。然后，OpenGL根据配置方式以及各种对象彼此相对的位置绘制视图，并通过该窗口呈现给用户。这个概念有点儿抽象，因此如果你感到困惑，也不必担心。本章稍后将通过示例详细说明这个概念。

Quartz相对比较容易使用。它提供了各种直线、形状以及图像绘制函数。尽管易于使用，但Quartz 2D仅限于二维绘图。尽管许多Quartz函数会在绘图时利用硬件加速，但无法保证在Quartz中执行的任何操作都得到了加速。

尽管OpenGL非常复杂，并且概念上也比较难理解，但是它的强大性是毫无疑问的。它同时提供了二维和三维绘图工具。它经过专门设计，目的是为了充分利用硬件加速。由于它可以跟踪虚拟世界的状态，因此还非常适合用于编写游戏和其他复杂的、图形密集的程序。

12.2 本章的绘图应用程序

下一个应用程序是一个简单的绘图程序（参见图12-1）。我们将分别使用Quartz 2D和OpenGL ES来构建该应用程序，因此你会真正感受到它们之间的差别。

该应用程序的特点是顶部和底部各有一个工具栏，每个工具栏都有一个分段控件。顶部的控件用于更改图形颜色，底部的控件用于更改要绘制的形状。当用户触击和拖动对象时，程序将用所选颜色绘制所选形状。为了最大程度地降低应用程序的复杂性，一次只绘制一种形状。

12.3 Quartz绘图方法

使用Quartz绘制图形时，通常会向绘制图形的视图中添加绘图代码。例如，可能会创建`UIView`的子类，并向该类的`drawRect:`方法中添加Quartz函数调用。`drawRect:`方法是`UIView`类定义的一部分，并且每次需要重绘视图时都会调用该方法。如果在`drawRect:`中插入Quartz代码，则会先调用该代码，然后重绘视图。

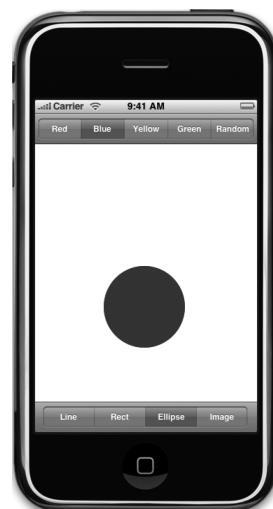


图12-1 本章中的简单绘图应用程序

12.3.1 Quartz 2D的图形上下文

在Quartz 2D中，和其他Core Graphics中一样，绘图是在图形上下文中进行的，通常，只称为上下文。每个视图都有相关联的上下文。要在某个视图中绘图时，你将检索当前上下文，使用此上下文进行各种Quartz图形调用，并且让此上下文负责将图形呈现到视图上。

这行代码将检索当前上下文：

```
CGContextRef context = UIGraphicsGetCurrentContext();
```

说明 我们使用Core Graphics C函数，而不是使用Objective-C对象来绘图。Core Graphics的API是基于C的，因此在本章的此部分中编写的大多数代码将由C函数调用组成。

定义图形上下文之后，可以将该上下文传递给各种Core Graphics函数来进行绘图。例如，以下代码将在上下文中绘制一条2像素宽的直线：

```
CGContextSetLineWidth(context, 2.0);
CGContextSetStrokeColorWithColor(context, [UIColor redColor].CGColor);
CGContextMoveToPoint(context, 100.0f, 100.0f);
CGContextAddLineToPoint(context, 200.0f, 200.0f);
CGContextStrokePath(context);
```

第一个调用指定任何绘制都应该创建一条2像素宽的直线。然后，我们指定笔划颜色应该为红色。在Core Graphics中，有两种颜色与绘图操作关联：笔划颜色和填充颜色。笔划颜色用于绘制直线以及形状的轮廓，填充颜色用于填充形状。

上下文具有一种与绘制直线关联的不可见的“画笔”。调用`CGContextMoveToPoint()`时，会将这个不可见的画笔移动到新位置，而不是实际绘制任何内容。这表示我们指定要绘制的直线从位置(100, 100)处开始（参见12.3.2节有关位置的说明）。下一个函数实际上绘制了一条从当前画笔位置到指定位置（该位置将成为新的画笔位置）的直线。在Core Graphics中绘图时，我们没有绘制任何实际可见内容。我们创建了形状、直线或某些其他对象，但它们不包含颜色或者任何使其可见的内容。就像用不可见的墨水在书写一样。在执行某些操作使其可见之前，我们看不到直线。因此，下一步是告知Quartz使用`CGContextStrokePath()`绘制直线。该函数将使用之前我们设置的线宽和笔划颜色对此直线进行涂色并使其可见。

12.3.2 坐标系

在上面的代码块中，我们将一对浮点数作为参数传递给`CGContextMoveToPoint()`和`CGContextLineToPoint()`。这些浮点数表示在Core Graphics坐标系中的位置。此坐标系中的位置由其x和y坐标表示，我们通常用(x, y)来表示。上下文左上角为(0, 0)。向下移动时，y增加。向右移动时，x增加。

在最后一个代码片段中，我们绘制了一条从(100, 100)到(200, 200)的对角线，绘制的直线类似于图12-2所示的直线。

12

在iPhone绘图时需经常使用的一个概念就是坐标系，它借鉴了许多图形库的绘图机制以及传统的几何学。例如，在OpenGL ES中，(0, 0)位于左下角，当y坐标增加时，你将移向上文或视图的顶部，如图12-3所示。使用OpenGL时，必须将位置从视图坐标系转换为OpenGL坐标系。这非常容易，在本章稍后的部分中，你将了解如何使用OpenGL。

若要在坐标系中指定一个点，某些Quartz函数需要使用两个浮点数作为参数。其他Quartz函数要求该点嵌入在`CGPoint`中，`CGPoint`是一个包含两个浮点值（即x和y）的`struct`。若要描述视图或其他对象的大小，Quartz将使用`CGSize`。`CGSize`也是一个拥有两个浮点值（即width和height）的`struct`。Quartz还声明一个名为`CGRect`的数据类型，它用于在坐标系中定义矩形。`CGRect`包含两个元素，一个是名为`origin`的`CGPoint`，它确定矩形的左上角，另一个是名为`size`的`CGSize`，它确定矩形的宽度（width）和高度（height）。

12.3.3 指定颜色

颜色是绘图的一个重要因素，因此理解颜色在iPhone上的运行原理是非常重要的。UIKit为此

提供了一个Objective-C类：`UIColor`。你不能在Core Graphic调用中直接使用`UIColor`对象，但可以像我们之前在以下代码片段中所做的一样，使用它的`CGColor`属性从`UIColor`实例中检索`CGColor`引用（Core Graphic函数要求这样做）：

```
CGContextSetStrokeColorWithColor(context, [UIColor redColor].CGColor);
```

我们使用`redColor`便利方法创建了一个`UIColor`实例，然后检索它的`CGColor`属性，并将该属性传递给函数。

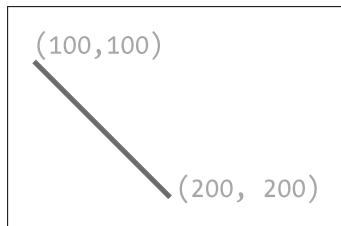


图12-2 在视图坐标系中绘制一条直线

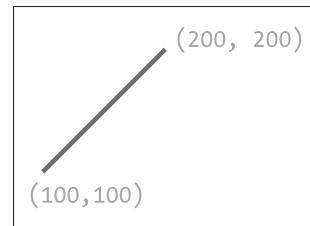


图12-3 在许多图形库（包括OpenGL）中，从(100, 100)到(200, 200)绘制一条直线应该与此类似，而不是与图12-2中的直线类似

1. iPhone显示的颜色理论

在现代计算机图形中，通常用4个要素（即红色、绿色、蓝色和透明度）表示颜色。在Quartz 2D中，这些值都是`CGFloat`类型（与iPhone上的`float`相同）类型，并且只能在0和1中取值。前3个要素很容易理解，因为它们表示加法三原色或RGB颜色模型（参见图12-4）。以不同比例组合这3种颜色可以产生不同的颜色。如果以相同的比例将这3种级别的原色放到一起，出现在你眼前的结果将是白色或某种灰度，具体情况取决于所混合原色的饱和度。以不同比例组合这3种原色，你可以获得一系列不同的颜色，称为色域。

你可能在小学学习过，原色包括红色、黄色和蓝色。这些原色（称为历史减法三原色或RYB颜色模型）在现代颜色理论中很少应用，几乎从来没有在计算机图形中使用。RYB颜色模型的色域是非常有限的，并且该模型不容易进行数学定义。你可能从来没有怀疑过小学的美术教师，但他们的理解至少不适用于计算机图形环境。之后我们会重复地讲“原色包括红色、绿色和蓝色”。

2. 比所看到的颜色还多

除了红色、绿色和蓝色之外，Quartz 2D（以及OpenGL ES）还有另一个组件：即alpha，它表示颜色的透明程度。当在一种颜色的上面绘制另一种颜色时，Alpha用于确定绘制的最终颜色。如果alpha为1.0，则绘制的颜色为100%不透明，它的下面的任何颜色都无法看清楚。如果它的值为任何小于1.0的值，则它下面的颜色将能够透过它显示出来，最后获得混合的颜色。当使用alpha组件时，有时颜色模型称为RGBA颜色模型，但是从技术上讲，alpha实际上并不是颜色的一部分；它只是定义绘制时颜色与其他颜色的交互方式。

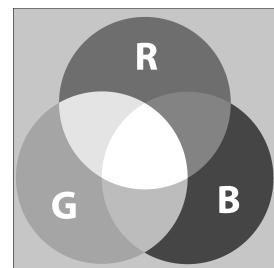


图12-4 组成RGB颜色模型的加法三原色的简单表示

尽管在计算机图形中最常用的是RGB模型，但是它不是唯一的颜色模型。其他一些模型也得到了使用，包括色调、饱和度、值（HSV）；色调、饱和度、亮度（HSL）；蓝绿色、洋红色、黄色、黑色（CMYK），它们用于实现四色打印；灰度级。此外，不同版本的RGB颜色空间使这一切变得更加复杂。所幸，对于大多数操作来说，我们不必担心所使用的颜色模型。我们只需从UIColor对象中传递CGColor，Core Graphics即可处理任何所需的转换。在使用OpenGL ES时，记住由于OpenGL ES需要采用RGBA来指定颜色，因此Quartz支持其他颜色模型，这一点非常重要。

UIColor提供了许多便利方法，可以返回初始化为特定颜色的UIColor对象。在上一个代码示例中，我们使用redColor方法来获取初始化为红色的颜色。幸运的是，这些便利方法创建的UIColor实例都使用RGBA颜色模型。

如果你需要对颜色进行更多控制，但不使用便利方法，则可以通过指定所有这4个组件来创建一种颜色。下面是一个示例：

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

12.3.4 在上下文中绘制图像

使用Quartz 2D，可以在上下文中直接绘制图像。这是Objective-C类（UIImage）的另一个示例，你可以使用此类作为操作Core Graphics数据结构（CGImage）的备用选项。此UIImage类包含将图像绘制到当前上下文中的方法。你需要确定此图像出现在上下文中的位置，方法是：指定一个CGPoint来确定图像的左上角或者指定一个CGRect来框住图像，并根据需要调整图像大小使其适合该框。可以在当前上下文中绘制一个UIImage，如下所示：

```
CGPoint drawPoint = CGPointMake(100.0f, 100.0f);
[image drawAtPoint:drawPoint];
```

12.3.5 绘制形状：多边形、直线和曲线

Quartz 2D提供了许多函数，这些函数简化了复杂形状创建。若要绘制一个矩形或一个多边形，实际上你不必计算角度，绘制直线或者根本不必进行任何数学计算。你只需调用一个Quartz函数即可实现该操作。例如，绘制椭圆形的方法是，定义它所适合的矩形并且让Core Graphics执行以下任务：

```
CGRect theRect = CGRectMake(0,0,100,100);
CGContextAddEllipseInRect(context, theRect);
CGContextDrawPath(context, kCGPathFillStroke);
```

对于矩形也是类似的方法。此外，还有许多方法用于创建更为复杂的形状（如弧形和Bezier路径）。若要了解有关Quartz中弧形和Bezier路径的详细信息，请查看http://developer.apple.com/documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/dq_intro/chapter_1_section_1.html上iPhone Dev Center中或Xcode联机文档中的*Quartz 2D Programming Guide*（Quartz 2D编程指南）。

12.3.6 Quartz 2D 工具示例：模式、梯度、虚线模式

Quartz 2D不像OpenGL那么昂贵，却提供了许多吸引人的工具，尽管这些工具中的许多工具

不在本书的讨论范围之内，但你应该知道它们的存在。例如，Quartz 2D支持用梯度填充多边形，而不仅仅是用纯色，并且不仅仅支持实线，而且还支持虚线模式。浏览图12-5中截取自苹果公司QuartzDemo示例代码的屏幕截图，了解Quartz 2D的实际操作示例。

现在你已经基本了解了Quartz 2D的工作原理以及它的功能，让我们尝试使用它吧。

12.4 构建QuartzFun应用程序

在Xcode中，使用基于视图的应用程序模板创建一个新项目，并将其命名为QuartzFun。创建项目之后，展开Classes和Resources文件夹，单击Classes文件夹，以便我们可以添加类。这个模板已经为我们提供了一个应用程序委托和一个视图控制器。我们将在视图中执行自定义绘图，因此需要创建一个UIView子类。在该子类中，我们将通过覆盖drawRect:方法进行绘图。创建一个新的Cocoa Touch Classes文件，并选择UIView subclass模板。将该文件命名为QuartzFunView.m，并确保创建了头文件。

与之前一样，我们将定义一些常量，但这次定义的常量是多个类所需要的，并且不是特定于某个类的。我们将只为常量创建头文件，因此通过以下访问创建一个新文件：从Other栏中选择Empty File模板，并将其命名为Constants.h。

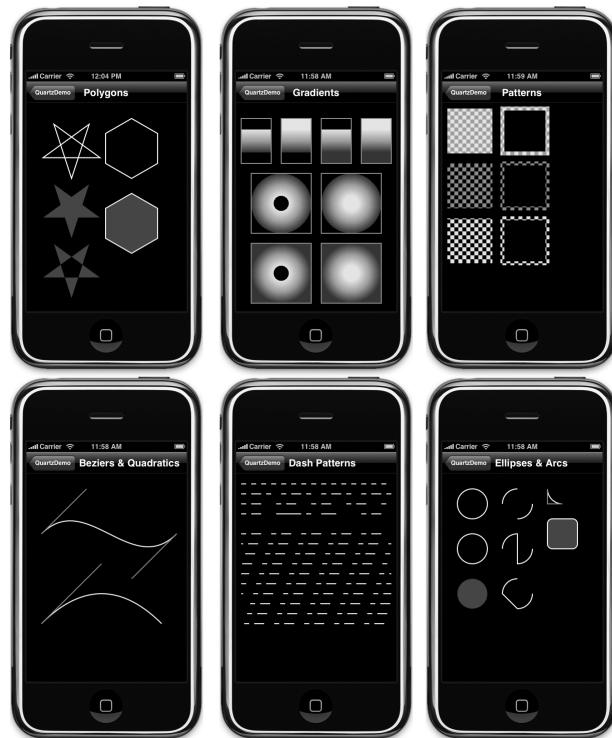


图12-5 一些Quartz 2D示例，来自于苹果公司提供的Quartz Demo示例项目

我们还需要创建两个文件。查看图12-1，你可以看到我们提供了一个选择随机颜色的选项。但UIColor没有提供返回随机颜色的方法，因此我们必须编写代码来执行该操作。当然，我们将该代码放置到控制器类中，但是由于我们是了解Objective-C的程序员，因此将该代码放置到UIColor上的某个类别中。使用Empty File模板创建两个文件，将它们分别命名为UIColor-Random.h和UIColor-Random.m。或者，使用NSObject subclass模板创建UIColor-Random.m，并让该模板为你自动创建UIColor-Random.h；然后删除这两个文件的内容。

12.4.1 创建随机颜色

让我们首先处理此类别。在UIColor-Random.h中，添加以下代码：

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

@interface UIColor(Random)
+(UIColor *)randomColor;
@end
```

现在，切换到UIColor-Random.m并添加以下内容：

```
#import "UIColor-Random.h"

@implementation UIColor(Random)
+(UIColor *)randomColor
{
    static BOOL seeded = NO;
    if (!seeded) {
        seeded = YES;
        srand(time(NULL));
    }
    CGFloat red = (CGFloat)random()/(CGFloat)RAND_MAX;
    CGFloat blue = (CGFloat)random()/(CGFloat)RAND_MAX;
    CGFloat green = (CGFloat)random()/(CGFloat)RAND_MAX;
    return [UIColor colorWithRed:red green:green blue:blue alpha:1.0f];
}
@end
```

这非常简单。我们声明一个静态变量，该变量告诉我们该方法是否是第一次调用。应用程序运行期间第一次调用该方法时，我们将运行随机数字生成器。在此处执行此操作意味着：我们不必依赖应用程序在其他地方执行该操作，因此，我们可以在其他iPhone项目中重用此类别。

在运行随机数字生成器之后，生成三个随机的CGFloat，其值介于0.0和1.0之间，使用这3个值来创建新的颜色。我们将alpha设置为1.0，以便所有生成的颜色都是不透明的。

12.4.2 定义应用程序常量

我们将使用分段控制器为用户可以选择的每个选项定义常量。单击Constants.h并添加以下内容：

```

typedef enum {
    kLineShape = 0,
    kRectShape,
    kEllipseShape,
    kImageShape
} ShapeType;

typedef enum {
    kRedColorTab = 0,
    kBlueColorTab,
    kYellowColorTab,
    kGreenColorTab,
    kRandomColorTab
} ColorTabIndex;
#define degreesToRadian(x) (3.14159265358979323846 * x / 180.0)

```

为了使代码更具有可读性，我们使用**typedef**声明了两个枚举类型。

12.4.3 实现QuartzFunView框架

由于我们将在**UIView**的某个子类中进行绘图，因此让我们用其所需的所有内容设置该类，但进行绘图的实际代码除外，我们稍后将添加这些代码。单击QuartzFunView.h并进行以下更改：

```

#import <UIKit/UIKit.h>
#import "Constants.h"

@interface QuartzFunView : UIView {
    CGPoint      firstTouch;
    CGPoint      lastTouch;
    UIColor     *currentColor;
    ShapeType    shapeType;
    UIImage     *drawImage;
    BOOL         useRandomColor;
}
@property CGPoint firstTouch;
@property CGPoint lastTouch;
@property (nonatomic, retain) UIColor *currentColor;
@property ShapeType shapeType;
@property (nonatomic, retain) UIImage *drawImage;
@property BOOL useRandomColor;
@end

```

我们做的第一件事情就是导入刚才创建的Constants.h头文件，这样便可以使用枚举。然后，声明实例变量。前两个变量将跟踪用户拖过屏幕的手指。我们将用户第一次触摸屏幕的位置存储在**firstTouch**中，将拖动时手指的位置以及拖动结束时手指的位置存储在**lastTouch**中。我们的绘图代码将使用这两个变量来确定在哪里绘制请求的形状。

接下来，我们定义某种颜色来存放用户的颜色选择，并定义一个**shapeType**以跟踪用户想绘制的形状。然后，定义一个**UIImage**属性，该属性存放用户选择底部工具栏中最右侧项目时在屏幕上绘制的图像（参见图12-6）。我们定义的最后一个属性是**Boolean**，它用于跟踪用户是否请

求随机颜色。

切换到QuartzFunView.m并进行以下更改：

```
#import "QuartzFunView.h"
#import "UIColor-Random.h"
@implementation QuartzFunView
@synthesize firstTouch;
@synthesize lastTouch;
@synthesize currentColor;
@synthesize shapeType;
@synthesize drawImage;
@synthesize useRandomColor;

- (id)initWithCoder:(NSCoder*)coder
{
    if (( self = [super initWithCoder:coder] ) ) {
        self.currentColor = [UIColor redColor];
        self.useRandomColor = NO;
        if (drawImage == nil)
            self.drawImage = [UIImage imageNamed:@"iphone.png"];
    }
    return self;
}
- (void)drawRect:(CGRect)rect {
    // Drawing code
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    if (useRandomColor)
        self.currentColor = [UIColor randomColor];
    UITouch *touch = [touches anyObject];
    firstTouch = [touch locationInView:self];
    lastTouch = [touch locationInView:self];
    [self setNeedsDisplay];
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    lastTouch = [touch locationInView:self];
    [self setNeedsDisplay];
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    lastTouch = [touch locationInView:self];
    [self setNeedsDisplay];
}
```

```

- (void)dealloc {
    [currentColor release];
    [drawImage release];
    [super dealloc];
}

```

@end

由于该视图是从nib中加载的，因此我们首先实现`initWithCoder:`。请记住nib中的对象实例将存储为归档对象，这与我们在上一章将对象归档和加载到磁盘所使用的机制完全相同。因此，从nib中加载对象实例时，`init:`或`initWithFrame:`都不会调用。而是使用`initWithCoder:`，因此任何初始化代码都需要在这里添加。若要将初始颜色值设置为红色，则将`useRandomColor`初始化为NO，并加载我们将要绘制的图像文件。你不必完全理解此处的其他代码。我们将在第13章详细讨论使用`touchesBegan:withEvent:`、`touchesMoved:withEvent:`和`touchesEnded:withEvent:`方法的技巧和具体细节。简单地说，可以覆盖这3个`UIView`方法以确定用户触摸iPhone屏幕的位置。

当用户手指第一次触摸屏幕时会调用`touchesBegan:withEvent:`。在该方法中，如果用户已经使用之前添加到`UIColor`中的新`randomColor`方法选择了某个随机颜色，则我们需要更改此颜色。之后，我们存储当前位置，这样便可以知道用户第一次触摸屏幕的位置，并指出：需要通过在`self`上调用`setNeedsDisplay`来重新绘制视图。

当用户在屏幕上拖动手指时会连续调用接下来的`touchesMoved:withEvent:`方法。此处，我们所要做的就是将新位置存储在`lastTouch`中，并指出需要重新绘制该屏幕。

当用户将手指从屏幕上抬起时会调用最后一个方法，即`touchesEnded:withEvent:`。就像在`touchesMoved:withEvent:`方法中一样，我们所要做的就是将最后一个位置存储在`lastTouch`变量中，并指出需要重新绘制该视图。

如果你还没有全面了解这3个方法在触摸过程中所执行的操作，请不用担心，我们将在后续章节中更详细地介绍这些方法。

完成应用程序骨架并运行之后，我们将重新回顾这些方法。`drawRect:`方法就是此应用程序的主体部分，目前仅包含一条注释，因为我们尚未编写该方法。我们首先需要完成应用程序设置，然后再添加绘图代码。

12.4.4 向视图控制器中添加输出口和操作

如果参考图12-1，你会看到我们的界面包括两个分段控制器，一个位于屏幕顶部，另一个位于屏幕底部。使用顶部的控制器，用户可以选择颜色，但该分段控制器仅应用于底部4个选项中的3个选项，因此我们需要一个到顶部分段控制器的输出口，以便当它不起作用时将它隐藏。我



图12-6 当在屏幕上绘制`UIImage`时，颜色控件消失

们还需要两个方法，一个方法是在选择新颜色时调用，另一个方法是在选择新形状时调用。

单击QuartzFunViewController.h并进行以下更改：

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

@interface QuartzFunViewController : UIViewController {
    IBOutlet UISegmentedControl *colorControl;
}

@property (nonatomic, retain) UISegmentedControl *colorControl;
- (IBAction)changeColor:(id)sender;
- (IBAction)changeShape:(id)sender;
@end
```

所做的更改一目了然。接下来，切换到QuartzFunViewController.m并进行以下更改：

```
#import "QuartzFunViewController.h"
#import "QuartzFunView.h"
#import "UIColor-Random.h"
#import "Constants.h"

@implementation QuartzFunViewController
@synthesize colorControl;

- (IBAction)changeColor:(id)sender {
    UISegmentedControl *control = sender;
    NSInteger index = [control selectedSegmentIndex];

    QuartzFunView *quartzView = (QuartzFunView *)self.view;

    switch (index) {
        case kRedColorTab:
            quartzView.currentColor = [UIColor redColor];
            quartzView.useRandomColor = NO;
            break;
        case kBlueColorTab:
            quartzView.currentColor = [UIColor blueColor];
            quartzView.useRandomColor = NO;
            break;
        case kYellowColorTab:
            quartzView.currentColor = [UIColor yellowColor];
            quartzView.useRandomColor = NO;
            break;
        case kGreenColorTab:
            quartzView.currentColor = [UIColor greenColor];
            quartzView.useRandomColor = NO;
            break;
        case kRandomColorTab:
            quartzView.useRandomColor = YES;
            break;
        default:
```

```

        break;
    }
}

- (IBAction)changeShape:(id)sender {
    UISegmentedControl *control = sender;
    [(QuartzFunView *)self.view setShapeType:[control
        selectedSegmentIndex]];

    if ([control selectedSegmentIndex] == kImageShape)
        colorControl.hidden = YES;
    else
        colorControl.hidden = NO;
}

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

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [colorControl release];
    [super dealloc];
}

@end

```

这段代码也非常简单。在`changeColor:`方法中，我们确定已选择的分段，并根据选择内容创建一个新颜色。我们将`view`转换为`QuartzFunView`。接下来，设置它的`currentColor`属性，以便它知道绘图所使用的颜色。但选择随机颜色时除外，如果选择随机颜色，我们只需将视图的`useRandomColor`属性设置为`YES`。由于所有绘图代码将包含在视图内部，因此我们不必对其他方法执行任何操作。

在`changeShape:`方法中，我们的做法类似。但是，由于不必创建对象，因此我们只需将视图的`shapeType`属性设置为来自`sender`的分段索引。还记得`ShapeType enum`吗？`enum`的4个元素与应用程序视图底部的4个工具栏分段相对应。我们将形状设置为与当前所选择的分段相同，并根据是否选择了`Image`分段来隐藏`colorControl`和取消隐藏`colorControl`。

12.4.5 更新 QuartzFunViewController.xib

开始绘图之前，我们需要向nib中添加分段控件，然后连接操作和输出口。双击QuartzFunViewController.xib，在Interface Builder中将其打开。第一件事是更改视图的类，因此在标签为QuartzFunViewController.xib的窗口中单击View图标，并按 $\text{⌘}4$ 打开身份检查器。将该类从UI View改为QuartzFun View。

接下来，在库中找到Navigation Bar。确保你控制的是Navigation Bar，而非Navigation Controller。我们只是希望该工具栏位于视图顶部。将Navigation Bar紧贴在视图窗口的顶部。

接下来，在库中找到Segmented Control，并将该控件拖动到Navigation Bar的顶部。放下该控件之后，它应该仍然为选中状态。捕捉此分段控件任何一侧的调整大小的点，调整它的大小，以便它占据导航栏的整个宽度。你不会看到任何蓝色引导线，但这种情况下，Interface Builder会限制该栏的最大大小，因此只需拖动它直到不再进一步展开为止。

按 $\text{⌘}1$ 调出属性检查器，并将分段数量从2更改为5。依次双击各分段，将它们的标签分别改为（从左到右）Red、Blue、Yellow、Green和Random。此时，你的View窗口应该类似于图12-7。

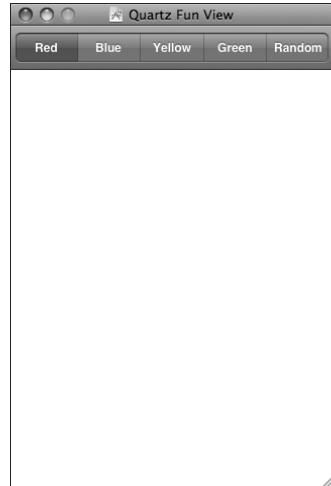


图12-7 完成后的导航栏

按住Control的同时，将File's Owner图标拖到分段控件上，并选择colorControl输出口。接下来，确保选中分段控件，并按 $\text{⌘}2$ 调出连接检查器。从Value Changed事件拖到File's Owner并选择changeColor:操作。

现在，在库中找到Toolbar，从中拖出一个工具栏并将其放置在窗口底部。库中的Toolbar上有一个我们并不需要的按钮，因此选择该按钮并按下Delete键。放置控件并删除按钮之后，选中另一个Segmented Control，并将其拖到工具栏上。

12

结果是，分段控件在工具栏中居中有点困难，因此我们将提供一点帮助。将Flexible Space Bar Button Item从库中拖到位于分段控件左侧的工具栏上。接下来，将另一个Flexible Space Bar Button Item拖到位于分段控件右侧的工具栏上。当我们调整该工具栏的大小时，这些项目将使分段控件位于工具栏的中心。单击分段控件以将其选中，并调整其大小以使它适合此工具栏，其中左右两侧各留有一点空间。

接下来，按 $\text{⌘}1$ 打开属性检查器，并将分段数从2更改为4。按顺序将4个分段的标题改为Line、Rect、Ellipse和Image。切换到连接检查器，并将Value Changed事件连接到File's Owner的changeShape:操作方法。保存并关闭nib，然后返回Xcode。

说明 你可能想知道为什么我们将一个导航栏放置在视图的顶部，将一个工具栏放置在视图的底部。根据苹果公司发布的“iPhone人机界面指南”，导航栏是经过专门设计的，它在屏

幕顶部看起来比较美观，而工具栏则是专门为底部而设计。如果你在Interface Builder的窗口中阅读了Toolbar和Navigation Bar的描述，你将会看到对此设计意图的说明。

编译并运行应用程序，确保一切正常。目前你还不能在屏幕上绘制形状，但分段控件可以工作，当在底部控件中轻击Image分段时，颜色控件会消失。一切都开始工作之后，我们进行绘图。

12.4.6 绘制直线

返回Xcode，编辑QuartzFunView.m，并用以下代码替换空的drawRect:方法：

```
- (void)drawRect:(CGRect)rect {
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGContextSetLineWidth(context, 2.0);
    CGContextSetStrokeColorWithColor(context, currentColor.CGColor);
    CGContextSetFillColorWithColor(context, currentColor.CGColor);

    switch (shapeType) {
        case kLineShape:
            CGContextMoveToPoint(context, firstTouch.x, firstTouch.y);
            CGContextAddLineToPoint(context, lastTouch.x, lastTouch.y);
            CGContextStrokePath(context);
            break;
        case kRectShape:
            break;
        case kEllipseShape:
            break;
        case kImageShape:
            break;
        default:
            break;
    }
}
```

首先，检索对当前上下文的引用，以便知道要绘图的位置：

```
CGContextRef context = UIGraphicsGetCurrentContext();
```

接下来，将线宽设置为2.0，这意味着我们画的任何直线都是2个像素宽：

```
CGContextSetLineWidth(context, 2.0);
```

随后，设置所画直线的颜色。由于UIColor有该方法所需的CGColor属性，因此我们使用currentColor实例变量的这个属性将正确的颜色传递给该函数：

```
CGContextSetStrokeColorWithColor(context, currentColor.CGColor);
```

我们使用switch跳转到每个形状类型的相应代码。我们将从处理kLineShape的代码开始，使其正常工作，然后依次为每个形状添加代码：

```
switch (shapeType) {
    case kLineShape:
```

要绘制直线，我们将不可见画笔移动到用户触摸的第一个位置。请记住，我们将该值存储在 `touchesBegan:` 方法中，以便它总是反映用户上次触摸或拖动时触摸的第一个点。

```
CGContextMoveToPoint(context, firstTouch.x, firstTouch.y);
```

接下来，我们绘制一条从该点到用户触摸的最后一个点的直线。如果用户的手指仍然与屏幕接摸，则 `lastTouch` 包含用户手指的当前位置。如果用户的手离开了屏幕，则 `lastTouch` 包含用户手指离开屏幕时的位置。

```
CGContextAddLineToPoint(context, lastTouch.x, lastTouch.y);
```

然后，画出这条路径。以下函数将画出一条直线，这是我们使用之前设置的颜色和宽度绘制而成的：

```
CGContextStrokePath(context);
```

随后，我们只需完成此 `switch` 语句就可以了，如下所示：

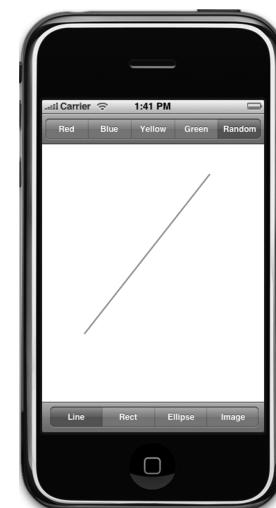
```
break;
case kRectShape:
    break;
case kEllipseShape:
    break;
case kImageShape:
    break;
default:
    break;
}
```

在构建和运行之前，还有最后一个步骤。由于 Quartz 2D 是 Core Graphics 的一部分，因此我们需要将 Core Graphics 框架添加到项目中。在项目窗口中，单击 Groups & Files 窗格中的 Frameworks，并添加此框架。如果你忘记了具体细节，请参考第 5 章。你将此书页折起角了，对吗？

此时，你应该能够进行编译和运行了。Rect、Ellipse 和 Shape 选项将不可用，但你应该能够很好地绘制直线（参见图 12-8）。

12.4.7 绘制矩形和椭圆形

让我们实现同时绘制矩形和椭圆形的代码，因为 Quartz 2D 基本上采用相同的方法实现这两个对象。对 `drawRect:` 方法进行以下更改：



12

图 12-8 应用程序中绘制直线的部分现在已经完成。在该图像中，我们使用随机颜色进行绘制

```

- (void)drawRect:(CGRect)rect {
    if (currentColor == nil)
        self.currentColor = [UIColor redColor];
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGContextSetLineWidth(context, 2.0);
    CGContextSetStrokeColorWithColor(context, currentColor.CGColor);

    CGContextSetFillColorWithColor(context, currentColor.CGColor);
    CGRect currentRect = CGRectMake(
        (firstTouch.x > lastTouch.x) ? lastTouch.x : firstTouch.x,
        (firstTouch.y > lastTouch.y) ? lastTouch.y : firstTouch.y,
        fabsf(firstTouch.x - lastTouch.x),
        fabsf(firstTouch.y - lastTouch.y));

    switch (shapeType) {
        case kLineShape:
            CGContextMoveToPoint(context, firstTouch.x, firstTouch.y);
            CGContextAddLineToPoint(context, lastTouch.x, lastTouch.y);
            CGContextStrokePath(context);
            break;
        case kRectShape:
            CGContextAddRect(context, currentRect);
            CGContextDrawPath(context, kCGPathFillStroke);
            break;
        case kEllipseShape:
            CGContextAddEllipseInRect(context, currentRect);
            CGContextDrawPath(context, kCGPathFillStroke);
            break;
        case kImageShape:
            break;
        default:
            break;
    }
}

```

由于我们希望将椭圆形和矩形涂上纯色，因此我们添加一个使用 `currentColor` 设置填充颜色的调用：

```
CGContextSetFillColorWithColor(context, currentColor.CGColor);
```

接下来，我们声明一个 `CGRect` 变量。我们将使用 `currentRect` 来存放由用户拖动描述的矩形。请记住，`CGRect` 包含两个成员：`size` 和 `origin`。通过 `CGRectMake()` 函数，我们可以通过指定 `x`、`y`、`width` 和 `height` 值来创建 `CGRect`，因此可用于绘制矩形。绘制矩形的代码乍看上去有点吓人，但实际上并没有那么复杂。用户可以向任何方向拖动，因此起点因拖动方向而异。我们使用两个点

中较小的x值和两个点中较小的y值来创建起点。然后通过获得两个x值和两个y值之差的绝对值来计算出大小：

```
CGRect currentRect = CGRectMake (
    (firstTouch.x > lastTouch.x) ? lastTouch.x : firstTouch.x,
    (firstTouch.y > lastTouch.y) ? lastTouch.y : firstTouch.y,
    fabsf(firstTouch.x - lastTouch.x),
    fabsf(firstTouch.y - lastTouch.y));
```

定义此矩形之后，绘制矩形或椭圆形就像调用两个函数一样轻松，一个函数是绘制矩形或在我们定义的CGRect中绘制椭圆形，另一个函数是绘画并填充它。

```
case kRectShape:
    CGContextAddRect(context, currentRect);
    CGContextDrawPath(context, kCGPathFillStroke);
    break;
case kEllipseShape:
    CGContextAddEllipseInRect(context, currentRect);
    CGContextDrawPath(context, kCGPathFillStroke);
    break;
```

编译并运行应用程序，并试用Rect和Ellipse工具，看看你有多喜欢它们。不要忘记不时更改颜色和试用随机颜色。

12.4.8 绘制图像

我们的最后一件事是绘制图像。12 QuartzFun文件夹中包含一个名为iphone.png的图像，你可以将该图像添加到Resources文件夹中，或者也可以添加你要使用的任何.png文件，只要记得将代码中的文件名改为所选图像即可。

向drawRect:方法中添加以下代码：

```
- (void)drawRect:(CGRect)rect {
    if (currentColor == nil)
        self.currentColor = [UIColor redColor];

    CGContextRef context = UIGraphicsGetCurrentContext();

    CGContextSetLineWidth(context, 2.0);
    CGContextSetStrokeColorWithColor(context, currentColor.CGColor);

    CGContextSetFillColorWithColor(context, currentColor.CGColor);
    CGRect currentRect;
    currentRect = CGRectMake (
        (firstTouch.x > lastTouch.x) ? lastTouch.x : firstTouch.x,
        (firstTouch.y > lastTouch.y) ? lastTouch.y : firstTouch.y,
        fabsf(firstTouch.x - lastTouch.x),
        fabsf(firstTouch.y - lastTouch.y));

    switch (shapeType) {
```

```

    case kLineShape:
        CGContextMoveToPoint(context, firstTouch.x, firstTouch.y);
        CGContextAddLineToPoint(context, lastTouch.x, lastTouch.y);
        CGContextStrokePath(context);
        break;
    case kRectShape:
        CGContextAddRect(context, currentRect);
        CGContextDrawPath(context, kCGPathFillStroke);
        break;
    case kEllipseShape:
        CGContextAddEllipseInRect(context, currentRect);
        CGContextDrawPath(context, kCGPathFillStroke);
        break;
    case kImageShape: {
        CGFloat horizontalOffset = drawImage.size.width / 2;
        CGFloat verticalOffset = drawImage.size.height / 2;
        CGPoint drawPoint = CGPointMake(lastTouch.x - horizontalOffset,
                                         lastTouch.y - verticalOffset);
        [drawImage drawAtPoint:drawPoint];
        break;
    }
    default:
        break;
}
}

```

提示 注意，在switch语句中，我们在case kImageShape:下的代码两侧添加了花括号。GCC在case语句之后的第一行中声明变量时遇到了问题。这些花括号是我们告诉GCC停止抱怨的一种方式。我们还在switch语句之前声明了horizontaloffset，该方法将相关代码放到一起了。

首先，我们计算该图像的中心，因为我们希望绘制的图像以用户上次触摸的点为中心。如果不进行调整，则会在用户手指的左上角绘制该图像，这也是一個有效的选项。然后通过从lastTouch中的x和y值中减去这些偏移量来生成一个新的CGPoint。

```

CGFloat horizontalOffset = drawImage.size.width / 2;
CGFloat verticalOffset = drawImage.size.height / 2;
CGPoint drawPoint = CGPointMake(lastTouch.x - horizontalOffset,
                               lastTouch.y - verticalOffset);

```

现在，我们通知图像绘制自身。此行代码将进行这项工作：

```
[drawImage drawAtPoint:drawPoint];
```

应用程序如预期执行，但我们应该考虑进行一些优化。在该应用程序中，你不会注意到速度减慢，但是在更复杂的应用程序（在速度较慢的处理器上运行）中，你会看到某些延迟。该问题由touchesMoved:和touchesEnded:方法中的QuartzFunView.m引起。这两个方法都包含下面这行

代码：

```
[self setNeedsDisplay];
```

很明显，这是我们告知视图重新绘制自身的方式。该代码正常工作，但它导致整个视图被擦除并重新绘制，即使只是非常微小的更改也是如此。当我们准备拖动新形状时，我们希望擦除该屏幕，但我们不希望在拖动形状时一秒钟清除屏幕好几次。

为避免在拖动期间多次强制重新绘制整个视图，我们可以使用`setNeedsDisplayInRect:`。`setNeedsDisplayInRect:`是一个`NSView`方法，该方法会将视图区域的一个矩形部分标记为需要重新显示。我们需要重新绘制的不仅仅是`firstTouch`和`lastTouch`之间的矩形，还有当前拖动所包围的任何屏幕部分。如果用户触摸屏幕，然后在屏幕上到处乱画，则只需重新绘制`firstTouch`和`lastTouch`之间的部分，将许多不需要的已绘制的内容留在屏幕上。

答案是跟踪受`CGRect`实例变量中的特定拖动影响的整个区域。在`touchesBegan:`中，我们将该实例变量重置为仅用户触摸的点。然后在`touchesMoved:`和`touchesEnded:`中，使用一个Core Graphics函数获取当前矩形和存储的矩形的并集，然后存储所得到的矩形。此外，还使用该函数指定需要重新绘制的视图部分。该方法为我们提供了受当前拖动影响的正在运行的全部区域。

我们立刻在`drawRect:`方法中计算当前矩形，以便绘制椭圆形和矩形形状。我们将该计算结果移动到新方法中，以便在所有3个位置中使用此新方法，而没有重复代码。准备好了吗？让我们开始吧。对`QuartzFunView.h`进行以下更改：

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

@interface QuartzFunView : UIView {
    CGPoint      firstTouch;
    CGPoint      lastTouch;
    UIColor     *currentColor;
    ShapeType    shapeType;
    UIImage     *drawImage;
    CGRect       redrawRect;
}

@property CGPoint firstTouch;
@property CGPoint lastTouch;
@property (nonatomic, retain) UIColor *currentColor;
@property ShapeType shapeType;
@property (nonatomic, retain) UIImage *drawImage;
@property (readonly) CGRect currentRect;
@property CGRect redrawRect;
@end
```

我们声明了一个名为`redrawRect`的`CGRect`，我们将使用它来跟踪需要重新绘制的区域。我们还声明了一个名为`currentRect`的只读属性，该属性将返回我们之前在`drawRect:`中计算的矩形。注意，这个属性没有底层实例变量。

切换到`QuartzFunView.m`并进行以下更改：

```
#import "QuartzFunView.h"

@implementation QuartzFunView
@synthesize firstTouch;
@synthesize lastTouch;
@synthesize currentColor;
@synthesize shapeType;
@synthesize drawImage;
@synthesize redrawRect;
@dynamic currentRect;
- (id)initWithFrame:(CGRect)frame {
    if (self = [super initWithFrame:frame]) {
    }
    return self;
}
- (CGRect)currentRect {
    return CGRectMake (
        (firstTouch.x > lastTouch.x) ? lastTouch.x : firstTouch.x,
        (firstTouch.y > lastTouch.y) ? lastTouch.y : firstTouch.y,
        fabsf(firstTouch.x - lastTouch.x),
        fabsf(firstTouch.y - lastTouch.y));
}
- (void)drawRect:(CGRect)rect {

    if (currentColor == nil)
        self.currentColor = [UIColor redColor];

    CGContextRef context = UIGraphicsGetCurrentContext();

    CGContextSetLineWidth(context, 2.0);
    CGContextSetStrokeColorWithColor(context, currentColor.CGColor);

    CGContextSetFillColorWithColor(context, currentColor.CGColor);
    CGRect currentRect = CGRectMake (
        (firstTouch.x > lastTouch.x) ? lastTouch.x + firstTouch.x,
        (firstTouch.y > lastTouch.y) ? lastTouch.y + firstTouch.y,
        fabsf(firstTouch.x - lastTouch.x),
        fabsf(firstTouch.y - lastTouch.y));

    switch (shapeType) {
        case kLineShape:
            CGContextMoveToPoint(context, firstTouch.x, firstTouch.y);
            CGContextAddLineToPoint(context, lastTouch.x, lastTouch.y);
            CGContextStrokePath(context);
            break;
        case kRectShape:
            CGContextAddRect(context, self.currentRect);
            CGContextDrawPath(context, kCGPathFillStroke);
            break;
    }
}
```

```
case kEllipseShape:  
    CGContextAddEllipseInRect(context, self.currentRect);  
    CGContextDrawPath(context, kCGPathFillStroke);  
    break;  
case kImageShape:  
    if (drawImage == nil)  
        self.drawImage = [UIImage imageNamed:@"iphone.png"];  
  
    CGFloat horizontalOffset = drawImage.size.width / 2;  
    CGFloat verticalOffset = drawImage.size.height / 2;  
    CGPoint drawPoint = CGPointMake(lastTouch.x - horizontalOffset,  
                                    lastTouch.y - verticalOffset);  
    [drawImage drawAtPoint:drawPoint];  
    break;  
default:  
    break;  
}  
}  
  
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event  
{  
    UITouch *touch = [touches anyObject];  
    firstTouch = [touch locationInView:self];  
    lastTouch = [touch locationInView:self];  
  
    if (shapeType == kImageShape) {  
        CGFloat horizontalOffset = drawImage.size.width / 2;  
        CGFloat verticalOffset = drawImage.size.height / 2;  
        redrawRect = CGRectMake(firstTouch.x - horizontalOffset,  
                               firstTouch.y - verticalOffset,  
                               drawImage.size.width, drawImage.size.height);  
    }  
    else  
        redrawRect = CGRectMake(firstTouch.x, firstTouch.y, 0, 0);  
    [self setNeedsDisplay];  
}  
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event  
{  
    UITouch *touch = [touches anyObject];  
    lastTouch = [touch locationInView:self];  
  
    [self setNeedsDisplay];  
    if (shapeType == kImageShape) {  
        CGFloat horizontalOffset = drawImage.size.width / 2;  
        CGFloat verticalOffset = drawImage.size.height / 2;  
        redrawRect = CGRectUnion(redrawRect, CGRectMake(lastTouch.x -  
                                                       horizontalOffset, lastTouch.y - verticalOffset,  
                                                       drawImage.size.width, drawImage.size.height));  
    }  
}
```

```

else
    redrawRect = CGRectMakeUnion(redrawRect, self.currentRect);
redrawRect = CGRectMakeInset(redrawRect, -2.0, -2.0);
[self setNeedsDisplayInRect:redrawRect];
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    lastTouch = [touch locationInView:self];

    [self setNeedsDisplay];
    if (shapeType == kImageShape) {
        CGFloat horizontalOffset = drawImage.size.width / 2;
        CGFloat verticalOffset = drawImage.size.height / 2;
        redrawRect = CGRectMakeUnion(redrawRect,
                                     CGRectMakeMake(lastTouch.x - horizontalOffset,
                                                   lastTouch.y - verticalOffset, drawImage.size.width,
                                                   drawImage.size.height));
    }
    redrawRect = CGRectMakeUnion(redrawRect, self.currentRect);
    [self setNeedsDisplayInRect:redrawRect];
}
- (void)dealloc {
    [currentColor release];
    [drawImage release];
    [super dealloc];
}
@end

```

仅增加了几行代码，我们就减少了重新绘制视图所需的大量工作（不再需要擦除和重新绘制未受当前拖动影响的视图部分）。像这样妥善处理iPhone宝贵的处理器周期，可以在应用程序性能方面产生巨大差别，尤其是当应用程序变得更加复杂时。

12.5 一些 OpenGL ES 基础知识

如前所述，OpenGL ES和Quartz 2D采用完全不同的方法进行绘图。对OpenGL ES的详细介绍本身就是一本书，因此我们在此不对其进行讨论。我们使用OpenGL ES重新创建我们的Quartz 2D应用程序，只是为了让你对其有个基本了解，并且向你提供一些示例代码，你可以依据这些代码实现自己的OpenGL应用程序。

说明 准备在你自己的应用程序中添加OpenGL时，请顺便浏览一下<http://www.khronos.org/opengles/>，该网页是OpenGL ES标准组的主页。更好的做法是访问此页并搜索单词“tutorial”：<http://www.khronos.org/developers/resources/opengles/>。

让我们开始创建应用程序吧。

构建 GLFun 应用程序

在Xcode中，创建一个基于视图的新应用程序，并将其命名为GLFun。为了节省时间，将文件Constants.h、UIColor-Random.h、UIColor-Random.m和iphone.png从Quartz-Fun项目复制到这个新项目中。打开GLFunViewController.h并进行以下更改。你应该能够识别它们，因为它们与我们之前对QuartzFunViewController.h进行的更改相同：

```
#import <UIKit/UIKit.h>
#import "Constants.h"
@interface GLFunViewController : UIViewController {
    IBOutlet UISegmentedControl *colorControl;
}
@property (nonatomic, retain) UISegmentedControl *colorControl;
- (IBAction)changeColor:(id)sender;
- (IBAction)changeShape:(id)sender;
@end
```

切换到QuartzFunViewController.m并进行以下更改。你应该对它们非常熟悉：

```
#import "GLFunViewController.h"
#import "GLFunView.h"
#import "UIColor-Random.h"

@implementation GLFunViewController
@synthesize colorControl;
- (IBAction)changeColor:(id)sender {
    UISegmentedControl *control = sender;
    NSInteger index = [control selectedSegmentIndex];

    GLFunView *glView = (GLFunView *)self.view;

    switch (index) {
        case kRedColorTab:
            glView.currentColor = [UIColor redColor];
            glView.useRandomColor = NO;
            break;
        case kBlueColorTab:
            glView.currentColor = [UIColor blueColor];
            glView.useRandomColor = NO;
            break;
        case kYellowColorTab:
            glView.currentColor = [UIColor yellowColor];
            glView.useRandomColor = NO;
            break;
        case kGreenColorTab:
            glView.currentColor = [UIColor greenColor];
            glView.useRandomColor = NO;
            break;
        case kRandomColorTab:
            glView.useRandomColor = YES;
    }
}
```

```

        break;
    default:
        break;
    }
}

- (IBAction)changeShape:(id)sender {
    UISegmentedControl *control = sender;
    [(GLFunView *)self.view setShapeType:[control selectedSegmentIndex]];
    if ([control selectedSegmentIndex] == kImageShape)
        [colorControl setHidden:YES];
    else
        [colorControl setHidden:NO];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [colorControl release];
    [super dealloc];
}

@end

```

这与QuartzFunController.m之间的唯一差别是，我们引用一个名为`GLFunView`的视图，而不是名为`QuartzFunView`的视图。进行绘图的代码包含在`UIView`的子类中。由于这次我们进行绘图的方式完全不同，因此使用一个新类来包含绘图代码比较有意义。

继续操作之前，你需要在项目中添加几个文件。在12 GLFun文件夹中，你可以找到4个文件，名称分别为`Texture2D.h`、`Texture2D.m`、`OpenGL2DView.h`和`OpenGL2DView.m`。这些文件中的代码是由苹果公司编写的，它们使得在OpenGL ES中绘制图像更加容易。如果你愿意，可以在自己的程序中随意使用这些文件。

OpenGL ES本身并没有`sprite`或图像；它具有纹理，但纹理必须绘制到对象上。在OpenGL ES中绘制图像的方法是绘制一个多边形，然后将纹理映射到该多边形上，以便它与多边形的大小完全匹配。`Texture2D`将相对比较复杂的过程封装到一个易于使用的类中。

`OpenGL2DView`是一个`UIView`子类，它使用OpenGL进行绘图。我们设置此视图的目的是便

于在一一对的基础上映射OpenGL ES的坐标系和视图的坐标系。OpenGL ES是一个三维系统。OpenGL2Dview将OpenGL 3-D世界映射到2-D视图的像素。注意，尽管视图和OpenGL上下文之间是一对一的关系，但是y坐标仍然是翻转的，因此我们必须将y坐标从视图坐标系（y增加表示向下移动）转换为OpenGL坐标系（y增加表示向上移动）。

若要使用OpenGL2Dview类，首先要将其子类化，然后实现draw方法进行实际绘图。还可以在视图中实现所需的任何其他方法，如与触摸有关的方法。

使用UIview子类模板创建一个新文件，并将其命名为GLFunView.m，必须创建它的头文件。现在，你可以双击GLFunViewController.xib，然后设计其界面。这次我们不会指导你进行该过程，但是如果你不明白的话，可以参考12.4.5节，以便获得具体步骤。

完成之后，保存并返回到Xcode。单击GLFunView.h并进行以下更改：

```
#import <UIKit/UIKit.h>
#import "Constants.h"
#import "Texture2D.h"
#import "OpenGL2DView.h"

@interface GLFunView : UIView {
    @interface GLFunView : OpenGL2DView {
        CGPoint      firstTouch;
        CGPoint      lastTouch;
        UIColor     *currentColor;
        BOOL        useRandomColor;

        ShapeType    shapeType;

        Texture2D   *sprite;
    }
    @property CGPoint firstTouch;
    @property CGPoint lastTouch;
    @property (nonatomic, retain) UIColor *currentColor;
    @property BOOL useRandomColor;
    @property ShapeType shapeType;
    @property (nonatomic, retain) Texture2D *sprite;
    @end
}
```

此类与QuartzFunView.h非常相似，但它不使用UIImage来存放图像，我们使用Texture2D来简化将图像绘制到OpenGL ES上下文中的过程。我们还将超类从UIView改为OpenGL2Dview，以便视图支持使用OpenGL ES进行二维绘图。

切换到GLFunView.m并进行以下更改。

```
#import "GLFunView.h"

@implementation GLFunView
@synthesize firstTouch;
@synthesize lastTouch;
@synthesize currentColor;
@synthesize useRandomColor;
```

```
@synthesize shapeType;
@synthesize sprite;
- (id)initWithCoder:(NSCoder*)coder
{
    if (self = [super initWithCoder:coder]) {
        self.currentColor = [UIColor redColor];
        self.useRandomColor = NO;
    }
    return self;
}

- (void)draw
{
    glLoadIdentity();

    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    CGColorRef color = currentColor.CGColor;
    const CGFloat *components = CGColorGetComponents(color);
    CGFloat red = components[0];
    CGFloat green = components[1];
    CGFloat blue = components[2];

    glColor4f(red,green, blue, 1.0);

    switch (shapeType) {
        case kLineShape: {
            if (sprite){
                [sprite release];
                self.sprite = nil;
            }

            GLfloat vertices[4];

            // Convert coordinates
            vertices[0] = firstTouch.x;
            vertices[1] = self.frame.size.height - firstTouch.y;
            vertices[2] = lastTouch.x;
            vertices[3] = self.frame.size.height - lastTouch.y;
            glLineWidth(2.0);
            glVertexPointer (2, GL_FLOAT , 0, vertices);
            glDrawArrays (GL_LINES, 0, 2);

            break;
        }
        case kRectShape:{

            if (sprite){
                [sprite release];
                self.sprite = nil;
            }
        }
    }
}
```

```

    }

    // Calculate bounding rect and store in vertices
    GLfloat vertices[8];
    GLfloat minX = (firstTouch.x > lastTouch.x) ?
        lastTouch.x : firstTouch.x;
    GLfloat minY = (self.frame.size.height - firstTouch.y >
        self.frame.size.height - lastTouch.y) ?
        self.frame.size.height - lastTouch.y :
        self.frame.size.height - firstTouch.y;
    GLfloat maxX = (firstTouch.x > lastTouch.x) ?
        firstTouch.x : lastTouch.x;
    GLfloat maxY = (self.frame.size.height - firstTouch.y >
        self.frame.size.height - lastTouch.y) ?
        self.frame.size.height - firstTouch.y :
        self.frame.size.height - lastTouch.y;

    vertices[0] = maxX;
    vertices[1] = maxY;
    vertices[2] = minX;
    vertices[3] = maxY;
    vertices[4] = minX;
    vertices[5] = minY;
    vertices[6] = maxX;
    vertices[7] = minY;

    glVertexPointer (2, GL_FLOAT , 0, vertices);
    glDrawArrays (GL_TRIANGLE_FAN, 0, 4);
    break;
}

case kEllipseShape: {
    if (sprite){
        [sprite release];
        self.sprite = nil;
    }
    GLfloat vertices[720];
    GLfloat xradius = (firstTouch.x > lastTouch.x) ?
        (firstTouch.x - lastTouch.x)/2 :
        (lastTouch.x - firstTouch.x)/2;
    GLfloat yradius = (self.frame.size.height - firstTouch.y >
        self.frame.size.height - lastTouch.y) ?
        ((self.frame.size.height - firstTouch.y) -
            (self.frame.size.height - lastTouch.y))/2 :
        ((self.frame.size.height - lastTouch.y) -
            (self.frame.size.height - firstTouch.y))/2;
    for (int i = 0; i <= 720; i+=2)
    {
        GLfloat xOffset = (firstTouch.x > lastTouch.x) ?
            lastTouch.x + xradius
            : firstTouch.x + xradius;

```

```

        GLfloat yOffset = (self.frame.size.height - firstTouch.y >
            self.frame.size.height - lastTouch.y) ?
            self.frame.size.height - lastTouch.y + yradius :
            self.frame.size.height - firstTouch.y + yradius;
        vertices[i] = (cos(degreesToRadian(i))*xradius) + xOffset;
        vertices[i+1] = (sin(degreesToRadian(i))*yradius) +
            yOffset;

    }

    glVertexPointer (2, GL_FLOAT , 0, vertices);
    glDrawArrays (GL_TRIANGLE_FAN, 0, 360);
    break;

}

case kImageShape:
    if (sprite == nil) {
        self.sprite = [[Texture2D alloc] initWithImage: [UIImage
            imageNamed:@"iphone.png"]];
        glBindTexture(GL_TEXTURE_2D, sprite.name);
    }
    [sprite drawAtPoint:CGPointMake(lastTouch.x,
                                    self.frame.size.height - lastTouch.y)];
    break;
default:
    break;
}

glBindRenderbufferOES(GL_RENDERBUFFER_OES, viewRenderbuffer);
[context presentRenderbuffer:GL_RENDERBUFFER_OES];
}

- (void)dealloc {
    [currentColor release];
    [sprite release];
    [super dealloc];
}
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    if (useRandomColor)
        self.currentColor = [UIColor randomColor];

    UITouch* touch = [[event touchesForView:self] anyObject];
    firstTouch = [touch locationInView:self];
    lastTouch = [touch locationInView:self];
    [self draw];
}

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

```

```

{
    UITouch *touch = [touches anyObject];
    lastTouch = [touch locationInView:self];

    [self draw];
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    lastTouch = [touch locationInView:self];

    [self draw];
}
@end

```

不难看出，使用OpenGL或使用任何方法都不如使用Quartz 2D那么简洁直观。尽管它比Quartz更加强大，但是同时也更加复杂。有时OpenGL可能会令人畏缩。

由于该视图是从nib中加载的，因此我们添加了一个`initWithCoder:`方法，在该方法中，我们创建了一个`UIColor`并将其分配给`currentColor`。我们还将`useRandomColor`的默认值设置为`NO`。但是，我们尚未创建`Texture2D`对象。由于我们绘制的形状没有纹理，因此不需要加载纹理。如果加载纹理，OpenGL ES将在绘制多边形时尝试使用纹理。因此，我们需要采取一些步骤以确保在绘制其他形状时不加载纹理。处理此问题的首选方法是延迟加载纹理。

`initWithCoder:`方法后面是`draw`方法，你可以在该方法中看到两个库之间的实际差别。让我们看一看绘制直线的过程。下面就是在Quartz版本（我们已经删除了与绘图无关的代码）中绘制直线的方法：

```

CGContextRef context = UIGraphicsGetCurrentContext();
CGContextSetLineWidth(context, 2.0);
CGContextSetStrokeColorWithColor(context, currentColor.CGColor);
CGContextMoveToPoint(context, firstTouch.x, firstTouch.y);
CGContextAddLineToPoint(context, lastTouch.x, lastTouch.y);
CGContextStrokePath(context);

```

12

下面是在OpenGL中绘制相同的直线所需采取的步骤。首先，我们重置虚拟机世界以便删除任何旋转、转换或已经应用于它的其他变换：

```

glLoadIdentity();
接下来，我们将背景清除为白色：
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);

```

之后，我们必须通过分割`UIColor`并从中拖出各个RGB组件来设置OpenGL绘图颜色。所幸，我们不必担心`UIColor`使用的是哪种颜色模型。我们可以安全地假设它将使用RGBA颜色空间：

```

CGColorRef color = currentColor.CGColor;
const CGFloat *components = CGColorGetComponents(color);
CGFloat red = components[0];
CGFloat green = components[1];
CGFloat blue = components[2];
glColor4f(red, green, blue, 1.0);

```

若要绘制直线，我们需要两个顶点，这意味着我们需要一个包含4个元素的数组。前面讨论过，二维空间中的点由两个值（即x和y）表示。在Quartz中，我们使用一个**CGPoint struct**来存放这些点。在OpenGL中，点未嵌入到**struct**中。相反，我们用组成需要绘制的形状的所有点来填充数组。因此，若要在OpenGL ES中绘制一条从点(100, 150)到点(200, 250)的直线，我们将创建一个如下所示的顶点数组：

```

vertex[0] = 100;
vertex[1] = 150;
vertex[2] = 200;
vertex[3] = 250;

```

我们的数组格式为{x1, y1, x2, y2, x3, y3}。该方法中的下一段代码将两个**CGPoint**结构转换为顶点数组：

```

GLfloat vertices[4];
vertices[0] = firstTouch.x;
vertices[1] = self.frame.size.height - firstTouch.y;
vertices[2] = lastTouch.x;
vertices[3] = self.frame.size.height - lastTouch.y;

```

定义描述我们需要绘制的内容（在本例中为直线）的顶点数组之后，指定线宽，使用方法**glVertexPointer()**将该数组传递到OpenGL ES中，并通知OpenGL ES绘制数组：

```

glLineWidth(2.0);
glVertexPointer(2, GL_FLOAT, 0, vertices);
glDrawArrays(GL_LINES, 0, 2);

```

在OpenGL ES中完成绘图之后，我们必须告诉它渲染其缓冲器，并且告诉我们的视图上下文显示新渲染的缓冲器：

```

glBindRenderbufferOES(GL_RENDERBUFFER_OES, viewRenderbuffer);
[context presentRenderbuffer:GL_RENDERBUFFER_OES];

```

为了阐明在OpenGL中绘图的过程由3个步骤组成。首先，在上下文中绘图。其次，完成所有绘图之后，将上下文呈现到缓冲器中。第三，呈现渲染缓冲器，即当像素实际绘制到屏幕上时。

正如你所见，OpenGL示例比较长。当查看绘制椭圆的过程时，Quartz 2D和OpenGL ES之间的差别变得更加明显。OpenGL ES不知道如何绘制椭圆。OpenGL是OpenGL ES的老大哥甚至前辈，它有许多生成常见的二维和三维形状的便利函数，而这些便利函数只是从OpenGL ES分离出来的一部分功能，这使得OpenGL更加精简并且更加适合在嵌入式设备（如iPhone）中使用。因此，更多责任落在了开发人员的身上。

作为提示，下面是我们使用Quartz 2D绘制椭圆的方法：

```

CGContextRef context = UIGraphicsGetCurrentContext();
CGContextSetLineWidth(context, 2.0);
CGContextSetStrokeColorWithColor(context, currentColor.CGColor);
CGContextSetFillColorWithColor(context, currentColor.CGColor);
CGRect currentRect;
CGContextAddEllipseInRect(context, self.currentRect);
CGContextDrawPath(context, kCGPathFillStroke);

```

对于OpenGL ES版本，开始的步骤与之前相同，重置任何移动或旋转，将背景清除为白色以及基于currentColor设置绘图颜色。

```

glLoadIdentity();
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
CGColorRef color = currentColor.CGColor;
const CGFloat *components = CGColorGetComponents(color);
CGFloat red = components[0];
CGFloat green = components[1];
CGFloat blue = components[2];
	glColor4f(red, green, blue, 1.0);

```

由于OpenGL ES不知道如何绘制椭圆，因此我们必须自己绘制，这意味着需要面对复杂的几何理论。我们将定义一个顶点数组，该数组存放720个GLfloat，这将存放360个点的x和y位置，围绕圆一度一个。我们可以更改点数来提高或降低此圆的平滑度。这种方法在将适合iPhone屏幕的任何视图上看起来都不错，但如果你绘制的所有内容是比较小的圆，则严格来讲可能需要进行更多处理。

```
GLfloat vertices[720];
```

接下来，我们将根据存储在firstTouch和lastTouch中的两个点计算此椭圆的水平半径和垂直半径。

```

GLfloat xradius = (firstTouch.x > lastTouch.x) ?
    (firstTouch.x - lastTouch.x)/2 :
    (lastTouch.x - firstTouch.x)/2;
GLfloat yradius = (self.frame.size.height - firstTouch.y >
    self.frame.size.height - lastTouch.y) ?
    ((self.frame.size.height - firstTouch.y) -
     (self.frame.size.height - lastTouch.y))/2 :
    ((self.frame.size.height - lastTouch.y) -
     (self.frame.size.height - firstTouch.y))/2;

```

然后，我们将围绕圆进行循环，计算围绕圆的正确的点：

```

for (int i = 0; i <= 720; i+=2) {
    GLfloat xoffset = (firstTouch.x > lastTouch.x) ?
        lastTouch.x + xradius : firstTouch.x + xradius;
    GLfloat yoffset = (self.frame.size.height - firstTouch.y >
        self.frame.size.height - lastTouch.y) ?
        self.frame.size.height - lastTouch.y + yradius :
        self.frame.size.height - firstTouch.y + yradius;

```

```
    vertices[i] = (cos(degreesToRadian(i))*xradius) + xoffset;
    vertices[i+1] = (sin(degreesToRadian(i))*yradius) + yoffset;
}
```

最后，我们将顶点数组提供给OpenGL ES，通知OpenGL ES绘制并渲染它，然后通知上下文呈现新渲染的图像：

```
glVertexPointer(2, GL_FLOAT, 0, vertices);
glDrawArrays(GL_TRIANGLE_FAN, 0, 360);
glBindRenderbufferOES(GL_RENDERBUFFER_OES, viewRenderbuffer);
[context presentRenderbuffer:GL_RENDERBUFFER_OES];
```

我们不会审查矩形方法，因为它使用的基本技巧相同；我们定义一个顶点数组，该数组中的4个顶点用于定义矩形，然后我们渲染并呈现它。我们也不对绘制图像进行过多描述，因为苹果公司提供的可爱的*Texture2D*类，使得绘制一个小精灵就像在Quartz 2D中绘制一样容易。

在*draw*方法之后，我们拥有与之前版本相同的与触摸有关的方法。唯一差别是它不告知视图它需要显示，而是调用我们刚才定义的*draw*方法。不需要告知OpenGL ES将更新屏幕的哪些部分。它会计算出来并且利用硬件加速以最高效的方式绘制。但是，在编译和运行此程序之前，需要将这两个框架链接到你的项目。按照第5章有关添加Core Graphics框架的说明，然后选择*OpenGLES.framework*和*QuartzCore.framework*（而不是选择*CoreGraphics.framework*）。

现在，你已经看到了OpenGL ES入门的足够内容。如果你对在iPhone应用程序中使用OpenGL ES感兴趣，可以在<http://www.khronos.org/opengles/>上查找OpenGL ES规范以及与OpenGL ES相关的书籍、文档和论坛的链接。

提示 如果你想创建一个全屏的OpenGL ES应用程序，不必手动构建它。Xcode为你提供了一个实用的模板。该模板为你设置了屏幕和缓冲器，甚至将一些示例绘图和动画代码放置到类中，以便你可以看到放置你的代码的位置。想尝试吗？新建一个iPhone OS应用程序，然后选择OpenGL ES Application模板。

12.6 小结

在本章中，我们真的只抓到了iPhone绘图功能的一点皮毛。现在，你应该逐渐适应了Quartz 2D。并且，通过参考苹果公司的文档，你可以满足遇到的大多数绘图需求。你还应该对什么是OpenGL ES及其如何与iPhone的视图系统集成有个基本了解。

下一步做什么呢？你将学习如何在应用程序中添加手势支持。