TURING 图灵程序设计丛书 移动开发系列

Apress

Objective-C
最新学习指南

- 最受欢迎的Objective-C培训教材全新升级版
- 权威解读Objective-C和Cocoa特性
- iPhone、iPad、Mac开发必备



Learn Objective-C on the Mac
For OS X and iOS (2nd Edition)

Objective-C基础教程

（第2版）

Scott Knaster
[美] Waqar Malik 著
Mark Dalrymple
周庆成 译

人民邮电出版社
POSTS & TELECOM PRESS

“读过Scott Knaster书的人都知道，他的书从来不会让人失望，这本也不例外。它全面介绍了Objective-C语言的基础知识和最新特性，语言简明扼要，行文风趣幽默。”

——Bob Boonstra

“我是Scott Knaster的老读者了，25年来他撰写了很多畅销的Mac编程书，这本书一如既往，十分地棒！”

——John A. Vink

Learn Objective-C on the Mac For OS X and iOS (2nd Edition)

Objective-C基础教程（第2版）

Objective-C是一门面向对象的通用而强大的高级编程语言。它有着优雅的编程环境，并发扬了C语言的优秀特性，是苹果的iOS和OS X操作系统的主要编程语言。

本书全面系统地讲述了Objective-C的基础知识和面向对象编程的重要概念，结合实例介绍了Cocoa工具包的优秀特性和框架，以及继承、复合、对象初始化、类别、协议、内存管理和源文件组织等重要编程技术，教你如何针对iOS或OS X用户界面编写出优秀的应用程序。另外，本书第2版新增的主要内容有：

- Objecitve-C的最新特性——代码块、ARC、类扩展；
- 新增工具Clang静态分析器和GCD；
- 如何使用UIKit框架开发精致的iOS应用程序；
- 如何使用最新版本的Xcode。

无论你是初次接触Objective-C和Cocoa，还是已有丰富的C语言、C++或者Java编程经验，本书都能让你轻松过渡并熟练掌握Objective-C！

Apress

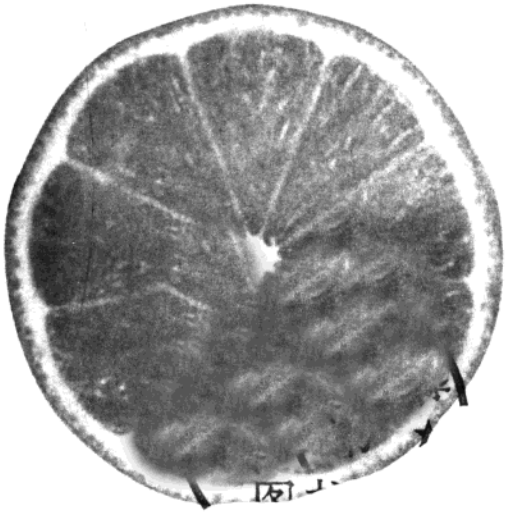
图灵社区：www.ituring.com.cn
新浪微博：@图灵教育　@图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议　计算机/程序设计/移动开发

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-31458-1
9 787115 314581 >
ISBN 978-7-115-31458-1
定价：59.00元

TURING 图灵程序设计丛书 移动开发系列



Learn Objective-C on the Mac
For OS X and iOS (2nd Edition)

Objective-C基础教程

(第2版)

Scott Knaster
[美] Waqar Malik 著
Mark Dalrymple
周庆成 译

人民邮电出版社

图书在版编目（CIP）数据

Objective-C基础教程 : 第2版 / (美) 克纳斯特 (Knaster,S.), (美) 马利克 (Malik,W.), (美) 达尔林普尔 (Dalrymple,M.) 著 ; 周庆成译. -- 北京 : 人民邮电出版社, 2013.5
(图灵程序设计丛书)
书名原文: Learn Objective-C on the Mac: For OS X and iOS, Second Edition
ISBN 978-7-115-31458-1

Ⅰ. ①0… Ⅱ. ①克… ②马… ③达… ④周… Ⅲ. ①C语言－程序设计－教材 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2013)第069008号

内 容 提 要

Objective-C 是扩展 C 的面向对象编程语言，也是 iPhone 开发用到的主要语言。本书结合理论知识与示例程序，全面而系统地介绍了 Objective-C 编程的相关内容，包括 Objective-C 在 C 的基础上引入的特性、Cocoa 工具包的功能及框架，以及继承、复合、源文件组织等众多重要的面向对象编程技术。附录中还介绍了如何从其他语言过渡到 Objective-C。

本书适合各类开发人员阅读。

图灵程序设计丛书

Objective-C基础教程（第2版）

◆ 著 [美] Scott Knaster Waqar Malik Mark Dalrymple
译 周庆成
责任编辑 卢秀丽
执行编辑 张 霞

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
三河市海波印务有限公司印刷

◆ 开本：800×1000 1/16
印张：20.5
字数：483千字 2013年5月第1版
印数：1－5 000册 2013年5月河北第1次印刷

著作权合同登记号 图字：01-2012-5557 号

ISBN 978-7-115-31458-1

定价：59.00元

读者服务热线：(010)51095186 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版权声明

Original English language edition, entitled *Learn Objective-C on the Mac: For OS X and iOS, Second Edition* by Scott Knaster, Waqar Malik, Mark Dalrymple, published by Apress, 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705 USA.

Copyright © 2012 by Scott Knaster, Waqar Malik, Mark Dalrymple. Simplified Chinese-language edition copyright © 2013 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由Apress L.P.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

译 者 序

Objective-C是一种通用、高级、面向对象的编程语言。它扩展了标准的ANSI C编程语言，将Smalltalk式的消息传递机制加入到ANSI C中。它是苹果的OS X和iOS操作系统及其相关API、Cocoa和Cocoa Touch的主要编程语言。

Objective-C最初源于NeXTSTEP操作系统，之后在OS X和iOS中继承了下来。目前主要支持的编译器有GCC和Clang，其中Clang被应用于Xcode 4.0中。

20世纪80年代初，Brad Cox与Tom Love以SmallTalk-80语言为基础发明了Objective-C。Objective-C在C语言的基础上添加了扩展，成为了能够创建和操作对象的一门新的程序设计语言。

1988年，苹果前CEO乔布斯的NeXT Computer公司获得了Objective-C语言的授权，并开发出了Objective-C的语言库和一个名为NEXTSTEP的开发环境。1992年，自由软件基金会的GNU开发环境增加了对Objective-C的支持。1994年，NeXT Computer公司和Sun Microsystem联合发布了一个针对NEXTSTEP系统的标准典范，名为OPENSTEP，在自由软件基金会的实现名称为GNUstep。1996年12月20日，苹果公司宣布收购NeXT Software公司，NEXTSTEP/OPENSTEP环境成为苹果操作系统主要发行版本OS X的基础。这个版本的开发环境被苹果公司称为Cocoa。在2006年7月苹果全球开发者会议中，Apple发布了Objective-C 2.0，增加了垃圾收集（只支持OS X系统）、属性、快速枚举等语法功能，改进了运行时性能，并添加了对64位系统的支持。2007年10月发布的Mac OS X v10.5中包含了Objective-C 2.0的编译器。

Objective-C作为面向对象的语言，其出现时间比C++还要早。随着OS X系统与iOS平台的不断发展，越来越多的移动开发者开始学习这门语言，因而其市场份额不断增大，排名也一度超越C++和C#等主流语言。在2011年与2012年，Objective-C连续两次赢得了TIOBE的年度编程语言大奖。在未来的日子里，Objective-C依然还有很大的上升空间。

本书全面介绍了Objective-C语言的基础知识。正文共有20章及一个附录，内容涵盖了面向对象编程的基础知识，Objective-C特有的继承、复合、内存管理、ARC自动引用计数、对象初始化、协议、键/值编码等特性，以及如何使用Xcode配合Cocoa或Cocoa Touch进行应用开发。附录中详细讲述了使用其他语言的编程人员在学习Objective-C时应注意的事项。本书内容精彩丰富，采用了示例程序与理论知识相结合的方式，相关的示例程序代码文件可以在Apress官网或图灵社区上免费下载得到。

本书内容由浅入深、结构清晰、步骤明确、简单易学、寓教于乐，特别适合没有面向对象编程基础的初学者学习，也同样适合于Objective-C程序员阅读和参考之用，而即使是其他语言的开

发人员，通过阅读本书也可以很快掌握Objective-C。相信你通过本书的学习，能够轻松驾驭Objective-C语言。祝你开发的应用程序能早日在App Store上架。

在翻译本书的过程中，我得到了许多人的帮助。首先感谢图灵社区的朱巍老师给予我这一次机会，感谢编辑与校对本书的所有编辑，你们帮我解决了很多问题，才保证了这本书的质量。其次感谢购买本书的读者，你们的支持使这本书的出版更有意义。最后衷心感谢上一版的译者们，你们在国内iOS开发刚起步时克服各种困难翻译出版了本书的第一版，为国内的移动开发者们提供了便利。在翻译本书第二版时，译者采纳了读者针对上一版的错误提出的建议，并尽可能地加以改善，但仍难免有疏漏之处，希望读者能一如既往地为本书勘误。



前　言

每当采用新平台时，程序员都会面临一个艰巨的挑战，那就是熟悉编程语言、开发工具、设计模式和新环境的标准软件库。

一般来说，程序员写代码时还面临着尽快交付软件的压力，因此很容易继续采用原有系统上所用的方法。这经常导致代码无法真正适应新平台，或者重复了已有的功能，还会给后期维护带来不少麻烦。

在理想状态下，新程序员可以向有新平台开发经验的同事请教，请他们提供指导，指引方向。遗憾的是，iOS平台发展得太快，很少能有这样的指导者。

如果你身边没有指导者，那么有没有别的方法呢？

本书的三位作者是苹果开发者技术服务组织的元老，他们都为新近使用苹果技术的软件工程师解答过无数的问题，并且一直在向他们传授良好的开发习惯。这些经验都体现在本书之中，可为读者答疑解惑，让大家对苹果开发平台不仅知其然，也知其所以然。

例如，第3章在介绍基础概念时采用了循序渐进、有条不紊的方式，而不是一古脑儿地列出陌生的类、方法和技术，让你通过练习自己去理清消化。

本书是你学习苹果公司的iOS以及OS X开发平台核心语言的最佳指导。

John C. Randolph

致　　谢

这本书是我们写的，但如果只有我们三位作者，本书根本不可能付梓出版。诚然，我们输入了文字，编写了代码，但若没有出色的出版团队，本书可能不过是一篇又臭又长的博文而已。

非常感谢Brent Dubi，他不厌其烦地给身处各地的我们发邮件，打电话，协调工作。感谢Nick Waynik为我们提供技术支持，这项任务真的很艰巨，因为苹果公司为Xcode和Cocoa添加了很多强大复杂的功能。还要感谢Gwenan Spearing，他帮我们斟字酌句，润色语言，使每句话都清晰简明。此外，还要感谢我们能干的文字编辑Heather Lang，他曾编辑本书的第一版，这次又挺身而出，帮助我们写出既符合语法又生动形象的文字。这些人都极大地提高了本书的质量，也使本书的内容更加精彩。

在此，Waqar还要感谢他的孩子Adam和Mishal，以及他美丽的妻子Irrum，给予他充足的时间写作本书。

准备工作

如果想要下载源代码或提交勘误，请访问Apress网站上的本书页面：www.apress.com/9781430241881。

为了更好地利用本书，建议你阅读时身边放台电脑，再准备一杯喜欢的饮料也是蛮不错的，不过要小心离电脑远一点哦，打翻了就悲剧了，因为修理费用会让你郁闷一整天的。

差不多就是这些了，愿本书能为你带来无限乐趣。

目　录

第 1 章　启程……1

1.1　预备知识……1
1.2　历史……1
1.3　内容简介……2
1.4　准备工作……3
1.5　小结……5

第 2 章　对 C 的扩展……6

2.1　最简单的 Objective-C 程序……6
2.2　解构 Hello Objective-C 程序……10
2.2.1　#import 语句……10
2.2.2　框架……11
2.2.3　NSLog()和@"字符串"……11
2.3　布尔类型……14
2.4　小结……18

第 3 章　面向对象编程的基础知识……19

3.1　间接……19
3.1.1　变量与间接……20
3.1.2　使用文件名的间接……22
3.2　在面向对象编程中使用间接……28
3.2.1　过程式编程……28
3.2.2　实现面向对象编程……34
3.3　有关术语……37
3.4　Objective-C 语言中的 OOP……38
3.4.1　@interface 部分……38
3.4.2　@implementation 部分……41
3.4.3　实例化对象……43
3.4.4　扩展 Shapes-Object 程序……45
3.5　小结……47

第 4 章　继承……48

4.1　为何使用继承……48
4.2　继承的语法格式……51
4.3　继承的工作机制……53
4.3.1　方法调度……54
4.3.2　实例变量……55
4.4　重写方法……57
4.5　小结……59

第 5 章　复合……61

5.1　什么是复合……61
5.2　自定义 NSLog()……62
5.3　存取方法……65
5.3.1　设置 engine 属性的存取方法……66
5.3.2　设置 tires 属性的存取方法……67
5.3.3　Car 类代码的其他变化……68
5.4　扩展 CarParts 程序……69
5.5　复合还是继承……70
5.6　小结……71

第 6 章　源文件组织……72

6.1　拆分接口和实现……72
6.2　拆分 Car 程序……75
6.3　使用跨文件依赖关系……77
6.3.1　重新编译须知……78
6.3.2　让汽车跑一会儿……79

6.3.3 导入和继承 ····· 81
6.4 小结 ····· 82
第 7 章 深入了解 Xcode ····· 84
7.1 窗口布局一览 ····· 84
7.2 改变公司名称 ····· 85
7.3 使用编辑器的技巧 ····· 86
7.4 在 Xcode 的帮助下编写代码 ····· 87
7.4.1 首行缩进（美观排版） ····· 88
7.4.2 代码自动完成 ····· 88
7.4.3 括号配对 ····· 90
7.4.4 批量编辑 ····· 91
7.4.5 代码导航 ····· 94
7.4.6 集中精力 ····· 96
7.4.7 使用导航条 ····· 97
7.4.8 获取信息 ····· 98
7.5 调试 ····· 101
7.5.1 暴力测试 ····· 101
7.5.2 Xcode 的调试器 ····· 101
7.5.3 精巧的调试符号 ····· 101
7.5.4 开始调试 ····· 101
7.5.5 检查程序 ····· 104
7.6 备忘表 ····· 105
7.7 小结 ····· 106
第 8 章 Foundation Kit 介绍 ····· 107
8.1 稳固的 Foundation ····· 107
8.2 使用项目样本代码 ····· 107
8.3 一些有用的数据类型 ····· 108
8.3.1 范围 ····· 108
8.3.2 几何数据类型 ····· 109
8.4 字符串 ····· 109
8.4.1 创建字符串 ····· 110
8.4.2 类方法 ····· 110
8.4.3 关于大小 ····· 111
8.4.4 字符串比较 ····· 111
8.4.5 不区分大小写的比较 ····· 112
8.4.6 字符串内是否还包含别的字符串 ····· 113
8.4.7 可变性 ····· 114
8.5 集合大家族 ····· 115
8.5.1 NSArray ····· 115
8.5.2 可变数组 ····· 119
8.5.3 枚举 ····· 120
8.5.4 快速枚举 ····· 121
8.5.5 NSDictionary ····· 122
8.5.6 请不要乱来 ····· 124
8.6 其他数值 ····· 124
8.6.1 NSNumber ····· 124
8.6.2 NSValue ····· 125
8.6.3 NSNull ····· 126
8.7 示例：查找文件 ····· 126
8.8 小结 ····· 130
第 9 章 内存管理 ····· 131
9.1 对象生命周期 ····· 131
9.1.1 引用计数 ····· 132
9.1.2 对象所有权 ····· 134
9.1.3 访问方法中的保留和释放 ····· 134
9.1.4 自动释放 ····· 136
9.1.5 所有对象放入池中 ····· 136
9.1.6 自动释放池的销毁时间 ····· 137
9.1.7 自动释放池的工作流程 ····· 138
9.2 Cocoa 的内存管理规则 ····· 140
9.2.1 临时对象 ····· 141
9.2.2 拥有对象 ····· 141
9.2.3 垃圾回收 ····· 143
9.2.4 自动引用计数 ····· 144
9.3 异常 ····· 154
9.3.1 与异常有关的关键字 ····· 155
9.3.2 捕捉不同类型的异常 ····· 156
9.3.3 抛出异常 ····· 156
9.3.4 异常也需要内存管理 ····· 157

9.3.5 异常和自动释放池……158
9.4 小结……159
第 10 章 对象初始化……160
10.1 分配对象……160
10.1.1 初始化对象……160
10.1.2 编写初始化方法……161
10.1.3 初始化时要做些什么……162
10.2 便利初始化函数……163
10.3 更多部件改进……164
10.3.1 Tire 类的初始化……165
10.3.2 更新 main()函数……166
10.3.3 清理 Car 类……168
10.4 Car 类的内存清理（垃圾回收方式和 ARC 方式）……171
10.5 指定初始化函数……172
10.5.1 子类化问题……173
10.5.2 Tire 类的初始化函数改进后的版本……175
10.5.3 添加 AllWeatherRadial 类的初始化函数……175
10.6 初始化函数规则……176
10.7 小结……176
第 11 章 属性……177
11.1 使用属性值……177
11.1.1 简化接口代码……178
11.1.2 简化实现代码……179
11.1.3 点表达式的妙用……182
11.2 属性扩展……183
11.2.1 名称的使用……186
11.2.2 只读属性……188
11.2.3 自己动手有时更好……189
11.2.4 特性不是万能的……189
11.3 小结……189
第 12 章 类别……191
12.1 创建类别……191
12.1.1 开始创建类别……192
12.1.2 @interface 部分……193
12.1.3 @implementation 部分……194
12.1.4 类别的缺陷……195
12.1.5 类别的优势……196
12.1.6 类扩展……196
12.2 利用类别分散实现代码……197
12.3 通过类别创建前向引用……200
12.4 非正式协议和委托类别……201
12.4.1 ITunesFinder 项目……202
12.4.2 委托和类别……204
12.4.3 响应选择器……205
12.4.4 选择器的其他应用……206
12.5 小结……206
第 13 章 协议……207
13.1 正式协议……207
13.1.1 声明协议……207
13.1.2 采用协议……208
13.1.3 实现协议……209
13.2 复制……209
13.2.1 复制 Engine……210
13.2.2 复制 Tire……211
13.2.3 复制 Car……212
13.2.4 协议和数据类型……215
13.3 Objective-C 2.0 的新特性……215
13.4 委托方法……216
13.5 小结……218
第 14 章 代码块和并发性……219
14.1 代码块……219
14.1.1 代码块和函数指针……219
14.1.2 Objective-C 变量……223
14.2 并发性……224
14.2.1 同步……224
14.2.2 队列也要内存管理……227
14.2.3 操作队列……229

14.3 小结 …… 231
第 15 章 AppKit 简介 …… 232
15.1 构建项目 …… 232
15.2 创建委托文件的@interface 部分 …… 234
15.3 Interface Builder …… 235
15.4 设计用户界面 …… 236
15.5 创建连接 …… 239
15.5.1 连接输出口（IBOutlet） …… 239
15.5.2 连接操作（IBAction） …… 240
15.6 应用程序委托的实现 …… 242
15.7 小结 …… 244
第 16 章 UIKit 简介 …… 245
16.1 视图控制器 …… 249
16.2 小结 …… 263
第 17 章 文件加载与保存 …… 264
17.1 属性列表 …… 264
17.1.1 NSDate …… 264
17.1.2 NSData …… 265
17.1.3 写入和读取属性列表 …… 266
17.1.4 修改对象类型 …… 267
17.2 编码对象 …… 268
17.3 小结 …… 273
第 18 章 键/值编码 …… 274
18.1 入门项目 …… 274
18.2 KVC 简介 …… 276
18.3 键路径 …… 277
18.4 整体操作 …… 278
18.4.1 休息一下 …… 279
18.4.2 快速运算 …… 282
18.5 批处理 …… 284
18.6 nil 仍然可用 …… 285
18.7 处理未定义的键 …… 286
18.8 小结 …… 287
第 19 章 使用静态分析器 …… 288
19.1 静态工作 …… 288
19.1.1 开始分析 …… 288
19.1.2 协助分析器 …… 292
19.1.3 了解更多 …… 293
19.2 小结 …… 295
第 20 章 NSPredicate …… 296
20.1 创建谓词 …… 296
20.2 计算谓词 …… 297
20.3 数组过滤器 …… 298
20.4 格式说明符 …… 299
20.5 运算符 …… 300
20.5.1 比较和逻辑运算符 …… 300
20.5.2 数组运算符 …… 301
20.6 有 SELF 就足够了 …… 302
20.7 字符串运算符 …… 304
20.8 LIKE 运算符 …… 304
20.9 结语 …… 305
附录 从其他语言转向 Objective-C …… 306
索引 …… 314

第1章 启 程

欢迎阅读本书！本书旨在教你学会Objective-C编程语言的基础知识。Objective-C语言是C语言的一个扩展集，几乎OS X或iOS平台上的所有应用程序都是用该语言开发的。

本书除了介绍Objective-C语言，还会介绍苹果公司为其提供的工具包Cocoa（针对OS X系统）和Cocoa Touch（针对iOS系统）。它们都是用Objective-C语言编写的，里面分别包含了OS X和iOS系统的所有用户界面元素和其他所有相关内容。学会了Objective-C之后，你就可以用Cocoa来开发功能完备的项目，还可以深入阅读其他相关书籍，比如*Learn Cocoa on the Mac*（Apress，2010）和*Beginning iOS 5 Development*[①]（Apress，2011）。

本章会介绍一些在学习Objective-C语言之前需要了解的基本知识，还将介绍Objective-C语言的一些历史，并简要介绍其他各章的内容。

1.1 预备知识

读者在阅读本书之前，应对C语言或类似的编程语言（比如C++或Java）有一定的了解。无论是哪一种语言，都应该熟悉它的基本原理，理解什么是变量、方法和函数，知道怎样使用条件和循环语句来控制程序流方向。本书将重点介绍Objective-C在其基础语言C上添加的新特性，以及苹果公司Cocoa工具包的一些优秀功能。

对于不具备C语言基础的Objective-C学习者，可以先看看本书的附录或阅读*Learn C on the Mac*（Apress，2009）后，再来学习本书的内容。

1.2 历史

Cocoa和Objective-C是苹果公司OS X和iOS操作系统的核心。虽然OS X（尤其是iOS）出现的时间相对较晚，但Objective-C和Cocoa的推出则已有时日。早在20世纪80年代初，Brad Cox为了融合流行的、可移植的C语言和优雅的Smalltalk语言的优势，就设计出了Objective-C语言。1985年，史蒂夫·乔布斯创立了NeXT公司，致力于创建功能强大且经济实惠的工作站。NeXT

① 中文版《iOS 5 基础教程》已由人民邮电出版社出版。——译者注

公司选择Unix作为操作系统并创建了NextSTEP（使用Objective-C语言开发的一款功能强大的用户界面工具包）。虽说它很有特点，并拥有了少量忠实的拥趸，但是在商业上却并没有获得成功。

苹果公司在1996年收购了NeXT（或者也可以说是NeXT收购了苹果公司[①]）之后，NextSTEP更名为Cocoa，并得到了Macintosh开发人员的广泛认可。苹果公司的开发工具（包括Cocoa）都是免费提供的，只要具备一定的编程经验和基本的Objective-C知识，以及强烈的求知欲，任何程序员都可以使用这些工具。

有人可能会问："既然Objective-C和Cocoa都是在20世纪80年代（那还是Alf和A-Team流行的时代，更不用提爷爷辈的Unix了）发明的，难道它们现在还没有过时吗？"当然没有！Objective-C和Cocoa是由一群优秀的编程人员耗费数年时间完成的，而且他们从未停止过更新与改进。经过多年发展，Objective-C和Cocoa已经演化成了一个美观精致且功能强大的工具集。近几年，iOS已经成为了最热门的开发平台，而Objective-C则是为其开发优秀应用程序的不二之选。因此，从NeXT最早采用至今已有二十多年，Objective-C的魅力依然不减当年。

1.3　内容简介

Objective-C是以C语言为基础的一个扩展集，它添加了一些微妙但意义重大的新特性。如果你接触过C++或Java编程语言，那么一定会惊叹Objective-C代码竟然如此简洁。本书其他章节将会详细介绍Objective-C在C语言基础上所添加的新特性。

- 第2章主要介绍Objective-C语言引入的基本特性。
- 第3章介绍面向对象编程的基础知识。
- 第4章介绍如何创建继承其父类特性的子类。
- 第5章讨论相关对象之间协同工作的技巧。
- 第6章演示创建程序源文件的实际策略。
- 第7章介绍Xcode的使用诀窍和强大功能，以帮助你提高编程效率。
- 第8章暂时告别Objective-C，转而介绍Cocoa的重要框架Foundation Kit，为你展现Cocoa的优秀特性。
- 第9章详细介绍Cocoa应用程序中的内存管理操作。
- 第10章讨论对象初始化的神奇。
- 第11章介绍了Objective-C语言中点表示法的实际作用，以及轻松访问对象的方法。
- 第12章详细描述了Objective-C语言中的一个非常出色的特性：类别（Category）。你可以通过它为现有的类（即便不是你所写的）添加自己的方法。
- 第13章介绍了Objective-C的一种继承方式——协议（Protocol），它允许类文件实现打包的特性集。

① 当时苹果公司已濒临绝境，在收购NeXT后，乔布斯成为了苹果的CEO，开始大刀阔斧地进行改革。——编者注

- 第14章展示了如何使用Objective-C的最新特性“程序块”（Block，它能够包含数据和代码）来增强函数的功能。
- 第15章介绍如何用AppKit框架来开发精致的OS X应用程序。
- 第16章与第15章类似，只不过它介绍的是iOS应用程序的基础框架UIKit。
- 第17章将展示如何保存和检索数据。
- 第18章讲解了如何使用“键–值编码”方法来间接处理数据。
- 第19章介绍了如何利用Xcode中强大的工具来查找程序员平常会出现的错误。
- 最后，第20章介绍如何分解数据。

如果你之前使用的是Java或C++等其他编程语言，或是Windows或Linux等其他平台，那么可以先阅读一下本书的附录，其中指出了学习Objective-C所需要克服的一些思维障碍。

1.4 准备工作

Xcode是苹果提供的用来创建iOS和OS X应用程序的开发环境。Mac电脑上并没有预装Xcode，不过只要你的苹果电脑运行的是OS X 10.7以上的系统，就可以轻松地免费下载并安装。

踏上OS X和iOS开发漫长奇妙之旅的第一步就是确保在电脑上安装了Xcode。没有安装的读者可以从Mac App Store上下载安装。方法是点击Dock栏上的App Store图标（如图1-1所示），也可以在用户的应用程序文件夹中打开App Store。

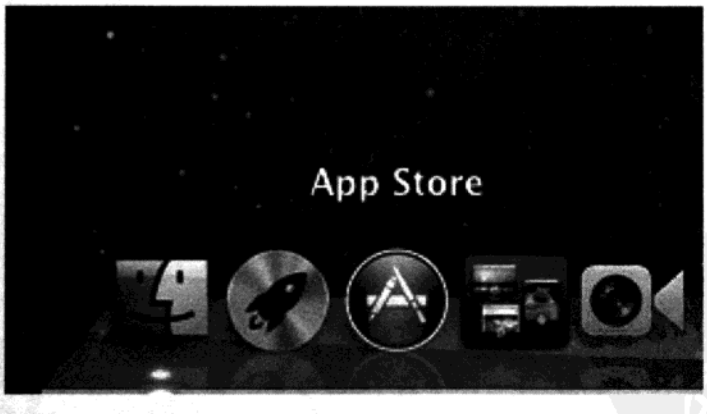


图1-1 Dock栏上的App Store图标

然后在Mac App Store应用程序右上角的搜索框中输入Xcode进行搜索（如图1-2所示）。

或者你也可以点击顶部的“类别”按钮并选择“软件开发工具”选项，此时你将在顶端某个位置看到Xcode（如图1-3所示）。点击Xcode图标便会跳转到它的下载页面了（如图1-4所示）。

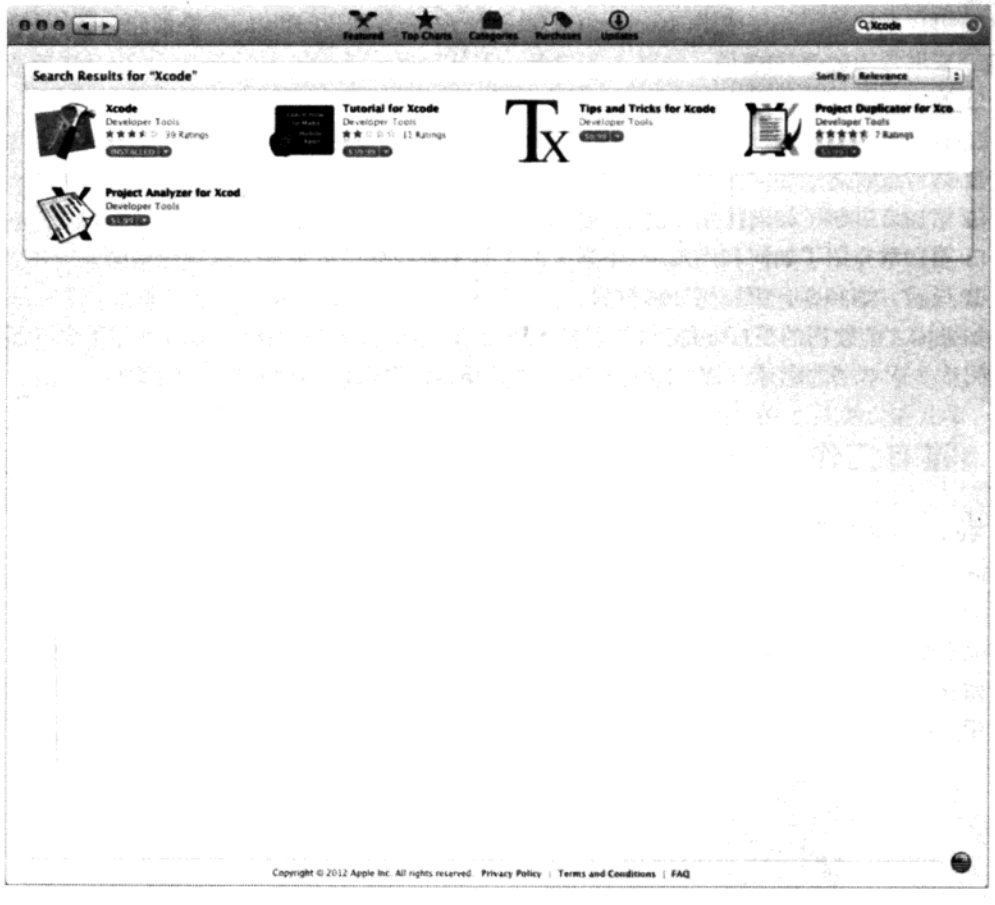


图1-2　在Mac App Store程序中搜索Xcode

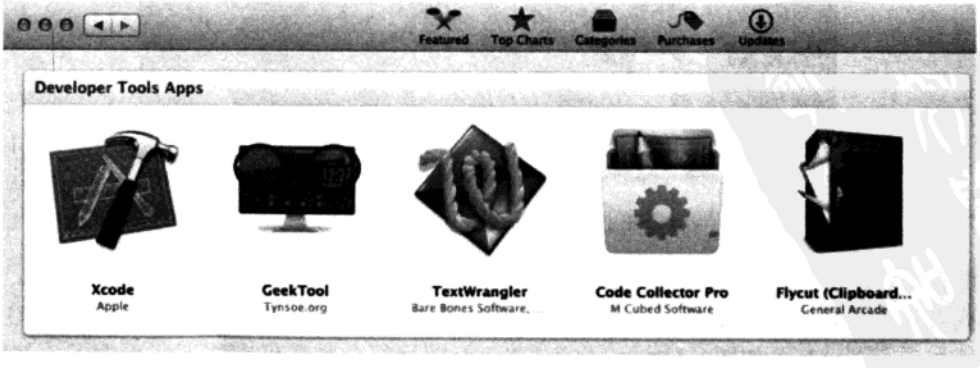


图1-3　软件开发工具



图1-4 Mac App Store中Xcode的下载页面

请点击“免费”按钮，然后点击“安装App”按钮。接下来App Store便会把Xcode[①]安装到你的应用程序文件夹中。

现在你可以开始Objective-C学习之旅了。祝你好运！我们会与你一起开始这场旅途，至少会陪你踏上第一段旅途。

1.5 小结

OS X和iOS程序都是用Objective-C语言编写的，它所使用的技术可以追溯到20世纪80年代，如今这些技术已经演化成一个功能强大的工具集。本书假设你已对C语言或其他一般编程语言有了一定的了解。

希望你能从本书中获得乐趣！

① 作者在写这本书时用的是4.2之后的版本，而译者在翻译此书时，Xcode最新的版本是4.4.1，二者差别不大，读者无需担心兼容问题。——译者注

第2章 对C的扩展

虽然Objective-C只是在C语言的基础上添加了一些新特性，但它非常好用！本章将指导你构建第一个（当然还会有第二个）Objective-C程序，同时会讲解那些关键的新特性。

2.1 最简单的 Objective-C 程序

你可能见过经典程序Hello World的C语言版本，该程序会输出Hello, world!或类似的简短语句。Hello World通常是C语言编程初学者要学习的第一个程序。我们现在将继承此传统，编写一个名为Hello Objective-C的类似程序。

构建Hello Objective-C程序

我们假设你在阅读本书的时候已经安装了苹果公司的Xcode开发包。如果你尚未安装Xcode，或者之前从未使用过此程序，可以参阅Dave Mark所写的*Learn C on the Mac*（Apress,2008），该书第2章会指导你如何获取、安装并使用Xcode来编写程序。

本节将讲解使用Xcode创建你的第一个Objective-C项目的全过程。如果你对Xcode已经很熟悉了，可以直接跳过这部分内容，我们不会介意的。在继续下一步之前，请确定解压了本书项目归档文件（可以从Apress网站的Source Code/Download页面下载[①]）中名为9781430241881的归档文件。该项目位于02.01-Hello Objective-C文件夹中[②]。

要创建项目，首先要启动Xcode。你可以在/Developer/Applications[③]目录下找到Xcode应用程序。为了便于访问，我们把Xcode图标放在了Dock快捷工具栏中。你也可以这么做。

Xcode启动完毕后，你将会看到欢迎界面，如图2-1所示。你可以在窗口左边选择接下来要做什么，也可以从右边列表选择打开某个最近编辑过的项目（当然，如果你是第一次启动Xcode，那就看不到任何最近编辑过的项目了）。假如你看不到欢迎界面，点击Window菜单下的Welcome to Xcode选项或使用Command+Shift+1快捷键就可以显示它。

---

① 本书代码也可以从图灵网站www.ituring.com.cn本书页面免费下载。——译者注

② 归档文件中作者把文件夹名称误写成Objectivr，请读者注意。——译者注

③ Xcode从4.3版本开始更改位置在“应用程序”文件夹中。——译者注

在欢迎界面中点击Create a new Xcode project选项（如图2-1所示），或者点击File➤New➤New Project。Xcode将在列表中显示它支持创建的各种项目类型。请不要受其他项目类型的影响，直接选中窗口左边Mac OS X下的Application，在右边选择Command Line Tool（命令行工具）图标，如图2-2所示。接下来点击Next按钮。



图2-1 Xcode欢迎界面

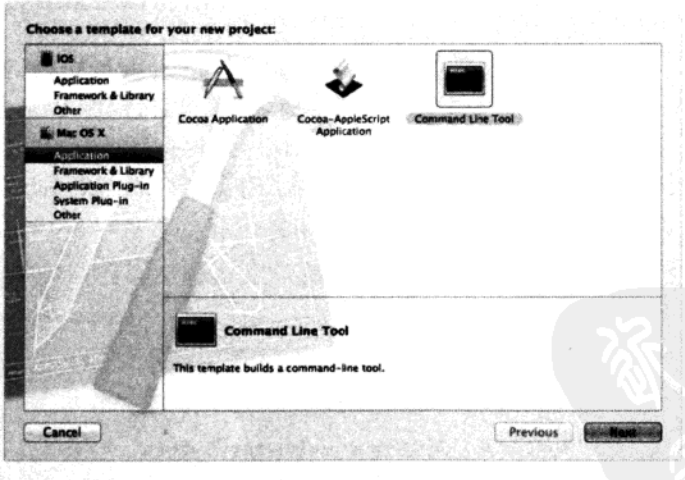


图2-2 创建一个新的命令行工具

在接下来的界面中（如图2-3所示），你需要对你的新项目进行设置。请在Product Name文本框中输入Hello Objective-C这个经典短语，然后在Company Identifier文本框中输入你的公司或网站地址的DNS反向格式，比如com.mywebsite，目前你只需要输入com.thinkofsomethingclever就可以了。

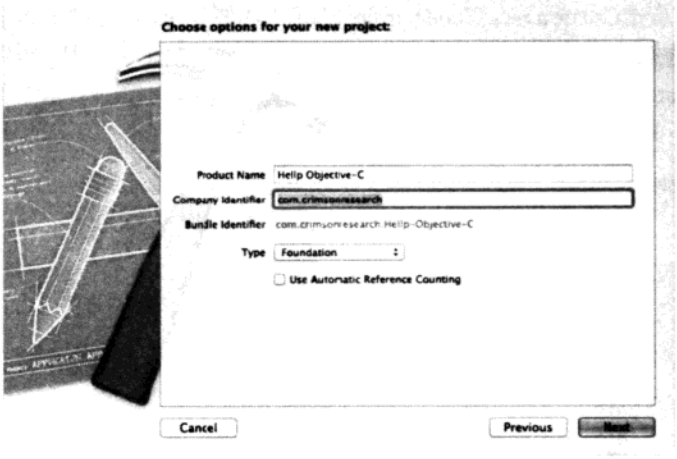


图2-3　设置你的项目选项

这个界面把最重要的选项留在了最后，它就是你想要创建的命令行工具类型。请确保选择了Foundation。完成之后你的界面看起来应该如图2-3所示。接下来点击Next按钮。

这时Xcode会向下弹出一个面板，询问你项目的存储位置（如图2-4所示）。我们把它放入了一个专用的项目目录中以便管理，当然你可以把它放在任何地方。

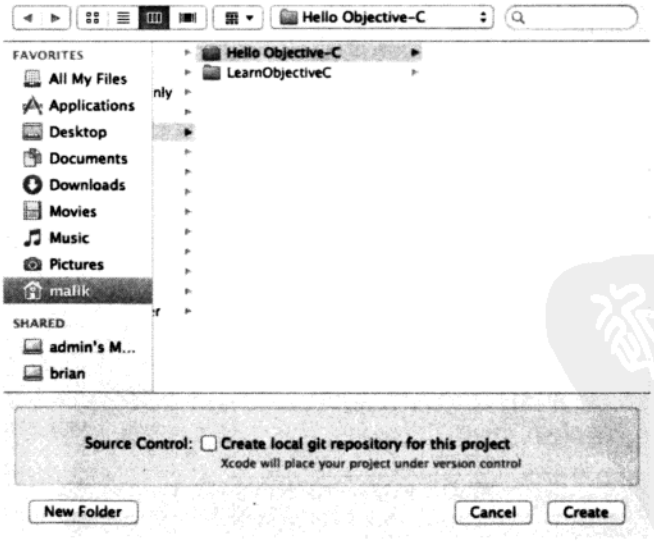


图2-4　为新的foundation工具命名

单击Save按钮之后，Xcode会显示它的主窗口，即项目窗口（如图2-5所示）。此窗口展示了项目的各组成部分以及编辑窗格。main.m就是包含了Hello Objective-C程序代码的源文件。



图2-5 Xcode的主窗口

Xcode体贴地在main.m中为每个新项目都准备了样本代码。我们可以让Hello Objective-C应用程序比Xcode提供的样本代码更简单一些。删除Hello Objective-C.m中的所有内容，并替换为以下代码：

```
#import <Foundation/Foundation.h>
int main (int argc, const char *argv[])
{
 NSLog (@"Hello, Objective-C!");

 return (0);
} // main
```

如果你目前还不能理解所有这些代码，请不必担心。我们稍后将详细地逐行分析此程序。

源代码只有转换成可以运行的程序之后才有意义。点击左上角的Run按钮或按下Command+R，就会生成并运行程序。只要没有任何语法错误，Xcode就会编译并链接你的程序，然后再运行它。选择View➤Debug Area➤Activate Console的菜单选项或按下Command+Shift+C快捷键，就会打开Xcode控制台窗口，其中会显示程序的输出结果，如图2-6所示。

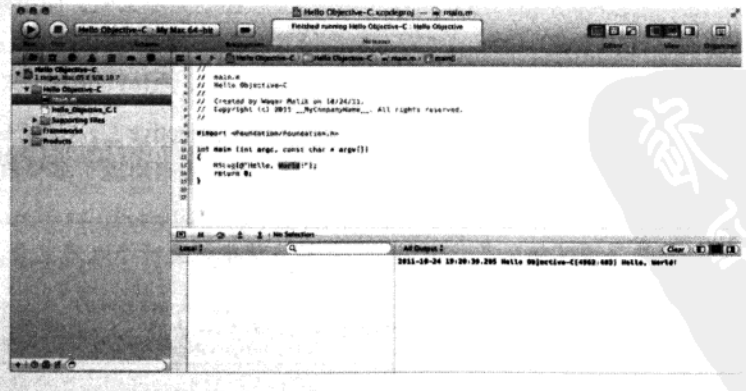


图2-6 运行Hello Objective-C程序

现在你已经完成了你的第一个Objective-C程序。恭喜你！接下来我们要剖析它的工作方式。

2.2　解构 Hello Objective-C 程序

我们在这里重新列出main.m文件的内容：

```
#import <Foundation/Foundation.h>
int main (int argc, const char *argv[])
{
  NSLog (@"Hello, Objective-C!");
  return (0);
} // main
```

Xcode通过.m扩展名来表示文件使用的是Objective-C代码，应由Objective-C编译器处理。而C编译器处理.c文件，C++编译器处理.cpp文件。在Xcode中，所有这些编译工作默认由LLVM处理，这个编译器能够理解C语言的全部3个变体。

如果你了解普通的C语言，那么你应该很熟悉main文件中包含的这两行代码：`main()`的声明语句和结尾的`return(0)`语句。请记住Objective-C本质上就是C语言，它用来声明`main()`和返回数值的语法与C语言是一样的。代码中余下几行看起来和普通的C语言有一些细微的差别，比如说这个陌生的`#import`是什么？想知道答案的话，请继续阅读吧！

> **说明**　Objective-C刚诞生的时候，扩展名.m代表message，它指的是Objective-C的一个主要特性，后面的章节将会介绍。现在我们称之为“.m文件”。

2.2.1　#import语句

与C语言一样，Objective-C使用头文件来包含结构体、符号常量和函数原型等元素的声明。在C语言中我们用`#include`语句来通知编译器查询头文件中相应的定义代码。在Objective-C中我们也可以使用`#include`来达到同样的目的，不过你可能永远都不会那么做，而是像下面这样使用`#import`语句：

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

`#import`是由Xcode使用的编译器提供的，Xcode在你编译Objective-C、C和C++程序时都会使用它。`#import`可保证头文件只被包含一次，无论此命令在该文件中出现了多少次。

> **说明**　在C语言中，程序员通常使用基于`#ifdef`命令的方案来避免一个文件包含另一个文件而后者又包含前者的情况。
>
> 而在Objective-C中，程序员使用`#import`命令来实现这个功能。

这里的`#import <Foundation/Foundation.h>`语句告诉编译器查找Foundation框架中的Foundation.h头文件。

2.2.2 框架

什么是框架？很高兴你问了这个问题。框架是一种把头文件、库、图片、声音等内容聚集在一个独立单元中的集合体。苹果公司将Cocoa、Carbon、QuickTime和OpenGL等技术作为框架集来提供。Cocoa的组成部分有Foundation和Application Kit（也称为AppKit）框架。此外还有一套支持型框架，包含了Core Animation和Core Image，它们为Cocoa增添了许多精彩的功能。

Foundation框架处理的是用户界面之下的那些层（Layer）的特性，比如数据结构和通信机制。本书中所有程序均以Foundation框架为基础。

> **说明** 学完本书后，如果想要成为Cocoa的权威专家，还需要精通Cocoa的Application Kit框架，它包含了Cocoa的许多高级特性：用户界面元素、打印、颜色和声音管理、AppleScript支持等。想要了解更多的信息，请参阅Jack Nutting、David Mark和Jeff LaMarche共同编著的*Learn Cocoa on the Mac*（Apress, 2010）。

每个框架都是一个重要的技术集合，通常包含数十个甚至上百个头文件。每个框架都有一个主头文件，它包含了框架内所有的头文件。通过在主头文件中使用`#import`，就可以访问框架内的所有功能。

Foundation框架的头文件占用了近1MB的磁盘存储空间，包含了14000多行代码，涵盖了100多个文件。只要使用`#import <Foundation/Foundation.h>`来包含主头文件，就能获得整个集合。也许你担心读取每个文件的文本会消耗编辑器很多时间，但是Xcode非常聪明：它使用预编译头文件（一种经过压缩的、摘要形式的头文件）来加快读取速度，通过`#import`导入这种文件时，加载速度会非常快。

如果你很好奇Foundation框架包含了哪些头文件，可以查看其Headers目录（/System/Library/Frameworks/Foundation.framework/Headers/）。只要不去移除或更改文件，仅仅是浏览，就不会对框架造成任何破坏。

2.2.3 NSLog()和@"字符串"

使用`#import`导入了Foundation框架的主头文件后，就可以开始使用Cocoa特性来编写代码了。Hello Objective-C中的第一行（也是唯一一行）实际代码使用了`NSLog()`函数，如下所示：

```
NSLog (@ "Hello, Objective-C!");
```

此代码可向控制台输出Hello, Objective-C!。如果你使用过C语言，那么一定遇到过`printf()`，`NSLog()`这个函数的作用和`printf()`很相似。

与`printf()`一样，`NSLog()`接受一个字符串作为其第一个参数，该字符串可以包含格式说明符

（比如%d），此函数会接受与格式说明符相匹配的其他参数。printf()在输出之前会把这些参数插入到第一个字符串参数中。

之前说过，Objective-C只是添加了一些新特性的C语言，因此你可以用printf()来代替NSLog()函数。但我们还是建议使用NSLog()，因为它添加了一些特性，比如时间戳、日期戳和自动附加换行符（'\n'）等。

1. 避免名称冲突的NS前缀

你可能会觉得NSLog()函数的命名方式有些奇怪。这里的NS到底是什么意思？事实上，Cocoa给其所有函数、常量和类型名称都添加了NS前缀。这个前缀告诉我们函数来自Cocoa而不是其他的工具包。

使用前缀能避免*名称冲突*（就是两个不同事物使用相同标识符时会引发的错误）。如果Cocoa将此函数命名为Log()，那么这个名称很可能会和某个不清楚情况的程序员创建的Log()函数发生冲突。当包含Log()的程序和包含同名函数的Cocoa一起构建时，Xcode会警告Log()被多次定义，并可能会产生糟糕的结果。

现在你已经知道了前缀的好处，但可能还是会奇怪为何前缀是NS而不是Cocoa。NS前缀的来历要追溯至此工具包还被称为NextSTEP的时候，当时它是NeXT Software公司（前NeXT公司，于1996年被苹果公司收购）的产品。苹果公司没有破坏为NextSTEP编写的代码的兼容性，继续使用NS前缀。由此可见，NS就像你我的阑尾一样，都属于历史遗存。

Cocoa已占用了NS前缀，所以很明显你不能再给你的任何变量和函数名称添加前缀NS，否则会让阅读代码的人产生混淆，他们会认为你创建的内容实际上属于Cocoa。况且，假如将来苹果公司为Cocoa添加一个函数，碰巧和你创建的函数名称相同，那么你的代码可能会出现问题。由于没有集中管理的前缀注册表，所以可以任意选择前缀。许多人使用他们的姓名首字母或公司名称作为前缀。为了使我们的例子更简单，本书中不会为代码使用前缀。

2. NSString:@是本体

让我们再来看看这行NSLog()语句：NSLog(@"Hello, Objective-C!");

你是否注意到了字符串前面的@符号？这可不是因为编辑疏忽而导致的排版错误。@符号是Objective-C在标准C语言基础上添加的特性之一。@符号意味着引号内的字符串应作为Cocoa的NSString元素来处理。

那么什么是NSString元素？去掉NS前缀，你会看到一个熟悉的术语：String（字符串）。字符串就是一串连续的可理解的字符，所以你一定能准确地猜到NSString就是Cocoa中的字符串。

NSString元素集成了大量的特性，在需要用字符串时，通过Cocoa可以随时使用它们。下面列出了NSString所支持的部分功能：

- 告诉你它的字符串长度是多少；
- 将自身与其他字符串进行比较；
- 将自身转换成整型值或浮点值。

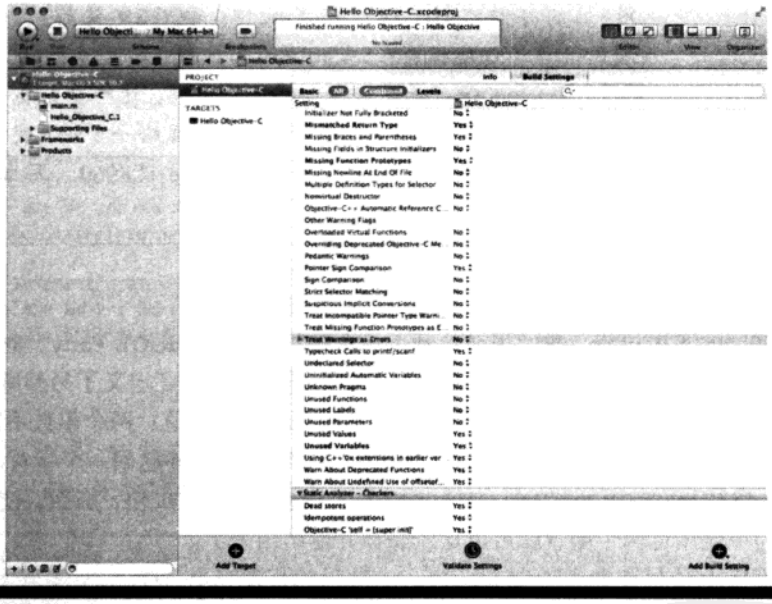
此外，它还有许多C语言风格的字符串无可比拟的功能。在第8章中将更多地用到并研究NSString元素。

注意字符串的格式

一个比较容易犯的错误就是将C语言风格的字符串（而不是NSString格式的@"字符串"元素）传递给了NSLog()。如果这样做的话，编译器会给出如下警告：

```
main.m:46: warning: passing arg 1 of 'NSLog' from incompatible pointer type
```

如果运行这个程序的话，它可能会崩溃。如果想要捕捉这样的问题，可以设置Xcode，让它总是将警报作为错误来处理。方法是首先选中Project Navigator（项目导航器）上的项目文件，然后再选中右边TARGETS文字下的Hello Objective-C选项，接下来再在编辑区的Build Settings选项卡下的搜索框中输入error，然后设置Treat Warnings as Errors的值为Yes（确保选中了顶部名为All的浮动按钮），如下图所示。



NSString的又一绝妙之处是，名称本身就是Cocoa的多个优秀特性之一。大多数Cocoa元素都以非常直接的方式命名，名称就可以反映出它们所能实现的功能。例如，NSArray存放数组，NSDateFormatter帮助你用不同的方式来设置时间格式，NSThread提供多线程编程工具，而NSSpeechSynthesizer能够让你听到语音。

接下来我们将继续讲解这个小程序。此程序的最后一行是返回语句，它可以终止main()函数的执行并结束这个程序：

```
return (0);
```

返回的值为0意味着这个程序成功地执行完了。它和C语言里返回语句的工作方式是一样的。

再次祝贺你！你刚刚已经编写、编译、运行并分析了你的第一个Objective-C程序。

2.3　布尔类型

许多编程语言都支持布尔（Boolean）类型，它指的是可以存储真值和假值的变量类型。Objective-C也不例外。

C语言拥有布尔数据类型bool，它具有true和false两个值。Objective-C也提供了一个相似的类型BOOL，它具有YES和NO两个值。顺便提一下，Objective-C的BOOL类型比C语言的bool类型早诞生了十多年。这两种不同的布尔类型可以在同一个程序中共存，但是如果你在编写Cocoa代码，那你只能使用BOOL。

> **说明**　Objective-C中的BOOL实际上是一种对带符号的字符类型（signed char）的类型定义（typedef），它使用8位的存储空间。通过#define指令把YES定义为1，NO定义为0。Objective-C并不会将BOOL作为仅能保存YES或NO值的真正布尔类型来处理。编译器仍将BOOL认作8位二进制数，YES和NO值只是在习惯上的一种理解。这样会引发一个小问题：如果不小心将一个大于1字节的整型值（比如short或int）赋给一个BOOL变量，那么只有低位字节会用作BOOL值。如果该低位字节刚好为0（比方说8960，写成十六进制为0x2300），BOOL值将会被认作是0，即NO值。

BOOL强大的实用功能

为了理解BOOL的实用功能，我们现在打开下一个项目：02.02-BOOL Party。此项目的作用是比较两个整数来判断它们是否相同。除了main()函数之外，此程序还定义了另外两个函数。第一个函数是areIntsDifferent()，它接受两个整型值，返回一个BOOL值：两个整数不同则返回YES，相同则返回NO。第二个函数是boolString()，它接受一个BOOL参数并返回一个字符串，参数是YES时返回@"YES"，参数是NO时返回@"NO"。如果想以可读的形式输出BOOL值，那么，使用这个函数会很方便。Main()中使用这两个函数就可以比较整数并输出结果了。

创建BOOL Party项目和生成Hello Objective-C项目的流程完全一样。

(1) 若Xcode尚未运行，先启动它。

(2) 选择Select File ➤ New ➤ New Project菜单选项。

(3) 选择左边Mac OS X下的Application，再在右边选中Command Line Tool。

(4) 点击Next按钮。

(5) 输入BOOL Party作为项目名称（Project Name），然后点击Next按钮。

(6) 点击Create按钮。

编辑main.m文件，使其如下所示：

```
#import <Foundation/Foundation.h>
// returns NO if the two integers have the same
// value, YES otherwise
BOOL areIntsDifferent (int thing1, int thing2)
{
```

```
  if (thing1 == thing2) {
   return (NO);
  } else {
  return (YES);
}

} // areIntsDifferent
// given a NO value, return the human-readable
// string "NO". Otherwise return "YES"

NSString *boolString (BOOL yesNo)
{
  if (yesNo == NO) {
   return (@"NO");
  } else {
   return (@"YES");
  }

} // boolString
int main (int argc, const char *argv[])
{
  BOOL areTheyDifferent;
  areTheyDifferent = areIntsDifferent (5, 5);
  NSLog (@"are %d and %d different? %@",
   5, 5, boolString(areTheyDifferent));

  areTheyDifferent = areIntsDifferent (23, 42);
  NSLog (@"are %d and %d different? %@",
   23, 42, boolString(areTheyDifferent));

  return (0);
} // main
```

构建并运行程序。需要选择View➤Debug Area➤Activate Console菜单选项或使用键盘快捷键Command+Shift+R来调出控制台窗口以便查看输出结果。你应该会看到类似下面的内容：

```
2012-01-20 16:47:09.528 02 BOOL Party[16991:10b] are 5 and 5 different? NO
2012-01-20 16:47:09.542 02 BOOL Party[16991:10b] are 23 and 42 different? YES
The Debugger has exited with status 0.
```

让我们再次将程序分解，逐个观察每个函数是如何执行的。

1. 第一个函数

我们要分析的第一个函数就是areIntDifferent()。

```
BOOL areIntsDifferent (int thing1, int thing2)
{
  if (thing1 == thing2) {
   return (NO);
  } else {
   return (YES);
  }

} // areIntsDifferent
```

areIntsDifferent()函数接受两个整型参数，返回一个BOOL值。它的语法和C语言很相似，你应该不会感到陌生。在这里，你会看到该函数将thing1和thing2进行比较。如果它们相同，就返回NO（即它们不是不同的）。如果它们不同，则返回YES。这非常直接明了，不是吗？

这样是无法获取布尔值的

经验丰富的C语言程序员也许会尝试将areIntsDifferent()函数的内容写成一行语句：

```
BOOL areIntsDifferent_faulty (int thing1, int thing2)
{
  return (thing1 - thing2);
} // areIntsDifferent_faulty
```

他们之所以这样做，是因为假定了非零值等于YES。但事实并非如此。确实，在C语言中此函数返回的值会是true或false，但在Objective-C中调用此函数返回的BOOL值只有YES或NO两个。程序员如果像下面这样使用此函数将会出错，因为23减5等于18。

```
if (areIntsDifferent_faulty(23, 5) == YES) {
// …. }
```

尽管这个函数的返回值在C语言中可能会等同于真值，但是在Objective-C中它不等于YES（YES的值以整型表示为1）。

记住，不要将BOOL值和YES直接进行比较。聪明过头的程序员有时会玩玩花样，写出类似areIntsDifferent_faulty这样有问题的函数。将前面的if语句改成如下所示可以解决此问题：

```
if (areIntsDifferent_faulty(5, 23)) {
// …. }
```

直接与NO比较一定是安全的，因为C语言中的假值就只有一个0。

2. 第二个函数

第二个函数boolString()的作用是将数值型的BOOL值映射为便于人们理解的字符串格式。

```
NSString *boolString (BOOL yesNo)
{
  if (yesNo == NO) {
   return (@"NO");
  } else {
   return (@"YES");
}

} // boolString
```

函数中间的if语句你应该很熟悉了吧。它只是比较了yesNo和常量NO，如果二者相同则返回@"NO"。否则，yesNo一定是真值，所以它返回@"YES"。

需要注意boolString()的返回值类型是一个指向NSString的指针。这意味着函数会返回一个Cocoa字符串，早在首次遇到NSLog()时我们就已经探讨过它了。如果观察返回语句，你会发现返回值前面有@符号，这明确地表示它们是NSString值。

main()是最后一个函数。声明main()的返回值类型和参数之后，有一个局部BOOL变量：

```
int main (int argc, const char *argv[])
{
  BOOL areTheyDifferent;
```

areTheyDifferent变量用来保存areIntsDifferent()返回的YES或NO值。我们可以直接把这个函数的BOOL返回值用于if语句，但添加这样一个变量可以使代码更易于阅读，这是没有坏处的。深度嵌套的解构经常令人困惑，也不易于理解，而且往往也是bug的藏身之所。

3. 比较

下面的两行代码使用areIntsDifferent()比较了一对整数，将返回值存储到了areTheyDifferent变量中，并通过NSLog()输出数字值和boolString()返回的易读的字符串：

```
areTheyDifferent = areIntsDifferent (5, 5);
NSLog (@"are %d and %d different? %@",
 5, 5, boolString(areTheyDifferent));
```

如前所见，NSLog()本质上就是具有Cocoa特色的printf()函数，它接受格式字符串，并将后续参数的值插入到这个格式说明符中。这里在对NSLog()的调用中，两个5将替换两个%d格式的占位符。

在我们提供给NSLog()的字符串的末尾，可以看到另一个@符号。这次表现为%@。它的含义是什么呢？它表示boolString()返回的是一个NSString指针。printf()不支持输出NSString，所以没有我们能够使用的格式说明符。NSLog()的编写者添加%@格式的说明符，是为了通知NSLog()接受适当的参数，将其作为NSString，并使用该字符串中的字符，将其发送到控制台中。

> **说明** 我们还没有正式介绍过对象，但现在可以简单透露一下：使用NSLog()输出任意对象的值时，都会使用%@格式来表示。在使用这个说明符时，对象会通过一个名为description的方法提供自己的NSLog()格式。NSString的description方法可简单输出字符串中的字符。

下面两行代码和刚才看过的代码非常相似：

```
areTheyDifferent = areIntsDifferent (23, 42);
NSLog (@"are %d and %d different? %@",
 23, 42, boolString(areTheyDifferent));
```

该函数对23和42两个整数进行了比较。由于这次它们是不同的，所以areIntsDifferent()返回YES。用户可以看到表示23和42不同的文本内容。

下面是最后一条返回语句，至此我们的BOOL Party程序就全部结束了。

```
  return (0);
} // main
```

在这个程序中，你学习了Objective-C的BOOL类型以及指示真和假的常量YES和NO。你可以像使用int和float等类型一样使用BOOL：将其用作变量、函数的参数和函数的返回值。

2.4　小结

在本章中，你第一次编写了两个Objective-C程序，这相当有趣吧！你还了解了Objective-C对C语言的一些扩展，比如#import能让编译器引入头文件（相同头文件只会引用一次）。你还学到了NSString字面量，也就是那些以@字符开头的字符串，例如@"hello"。你还使用了Cocoa提供的一个重要且通用的函数NSLog()，它可以将文本输出到控制台中。还使用了NSLog()专用的的格式说明符%@，它能够让你把NSString值插入到NSLog()的输出结果中。你还掌握了一个诀窍：只要在代码中发现了@符号，就知道现在看的是C语言的扩展Objective-C。最后你还学习了Objective-C的BOOL类型。

继续学习下一章的内容吧，我们将进入面向对象编程的奇妙世界。



第3章 面向对象编程的基础知识

只要你曾使用计算机进行过编程，不论资历深浅，都可能不止一次地听过“面向对象编程”这个术语。*面向对象编程*（Object-Oriented Programming）的首字母缩写为OOP，这是一种编程技术，最初是为了编写模拟程序而开发的。OOP很快就俘获了其他种类软件（比如涉及图形用户界面的软件）开发者的心。很快OOP就成为了业内一个非常重要的流行词。它被誉为具有魔力的银色子弹，可以使编程工作变得简单而愉悦。

当然，这种说法有些夸大其词。要精通OOP，仍然需要学习和练习。不过它确实能简化某些编程任务，甚至还能让编程变得更加有趣。本书将频繁讨论OOP，主要是因为Cocoa基于OOP概念，并且Objective-C是一种面向对象的语言。

那么到底什么是OOP呢？OOP是一种编程架构，可构建由多个对象组成的软件。对象就好比存在于计算机中的小零件，它们通过互相传递信息来完成工作。本章我们将关注OOP的基本概念，研究可实现OOP的编程风格，并讲解一些OOP特性背后的原理，最后会全面介绍OOP的机制。

> **说明** 与许多“新”技术一样，要追溯OOP的起源就要拨开历史的层层迷雾。它演变自20世纪60年代的Simula、70年代的Smalltalk、80年代的Clascal以及其他相关语言。C++、Java、Python和Objective-C等现代编程语言都从这些早期的语言中获得了灵感。

在研究OOP的过程中，你恐怕要做好迎接各种陌生技术术语的准备。OOP包含了很多听起来非常奇怪的术语，这会让人误以为它非常神秘并且难以理解，但事实并非如此。你甚至会怀疑计算机科学家们之所以创造了这些冗长夸张的词汇是为了向人们显摆他们有多聪明，其实他们不完全是这样。不过不必担心，我们会解释后面遇到的每一个术语。

在讨论OOP之前，先来看看OOP的一个关键概念：间接（indirection）。

3.1 间接

在编程界有一句老话，大意是这样的：“只要再多添加一层间接，计算机科学中就没有解决不了的问题。”*间接*这个词看似隐晦，其实意思非常简单——在代码中通过指针间接获取某个值，而不是直接获取。下面举一个现实生活中的例子：你可能不知道自己最喜欢的比萨饼店的电话号码，但你知道可以通过查阅电话簿来找到它。使用电话簿就是一种间接的形式。

间接也可以理解为让其他人代替自己做某件事。假设你有一箱书要还给住在城镇另一头的朋友Andrew，你知道你的邻居今晚要去拜访Andrew，你就不必亲自开车横跨城镇把书给他，你可以拜托友善的邻居送那箱书。这就是另一种间接：让他人代替你自己去完成工作。

在编程时，可以在多层之间使用间接，如编写一段代码来查询其他代码，并通过它继续访问另一层代码。你可能有过拨打技术支持热线的经历。你对客服说明了问题，他为你转接到能处理此问题的具体部门。该部门人员再将你转接到下一级能帮你解决问题的技术人员。如果你也和我们一样，发现自己拨打了错误的号码，那么你不得不转向另一部门寻求帮助。这种推诿就是间接的一种形式。幸运的是，计算机的耐心是无限的，为了找到答案能够接受多次差遣。

3.1.1　变量与间接

你可能会惊奇地发现自己已经在程序中使用过间接了。基本变量就是间接的一种实际应用。来看看下面这个输出数字1到5的小程序。在本书配套资源中的03.01 Count-1文件夹中可以找到这个程序。

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

int main (int argc, const char *argv[])
{
 NSLog (@"The numbers from 1 to 5:");

 for (int i = 1; i <= 5; i++) {
  NSLog (@"%d\n", i);
 }

 return (0);

} // main
```

Count-1程序中有一个运行了5次的for循环，它使用NSLog()来显示循环每次运行时的值。运行此程序之后，你将看到如下输出结果。

```
2012-01-21 11:52:01.940 03.01 Count-1[26429:903] The numbers from 1 to 5:
2012-01-21 11:52:01.943 03.01 Count-1[26429:903] 1
2012-01-21 11:52:01.944 03.01 Count-1[26429:903] 2
2012-01-21 11:52:01.944 03.01 Count-1[26429:903] 3
2012-01-21 11:52:01.945 03.01 Count-1[26429:903] 4
2012-01-21 11:52:01.947 03.01 Count-1[26429:903] 5
```

现在假设你想要更新这个程序，使其能输出从1到10的数字。那么你必须修改下列代码清单中以粗体显示的两处代码，然后重新构建这个程序（修改后的程序位于03.02 Count-2文件夹中）。

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

int main (int argc, const char * argv[])
{
 NSLog (@"The numbers from 1 to 10:");
```

```
 for (int i = 1; i <= 10; i++) {
  NSLog (@"%d\n", i);
 }

 return (0);

} // main
```

Count-2程序生成的输出结果为：

```
2012-01-21 12:03:13.433 03.02 Count-2[26507:903] The numbers from 1 to 10:
2012-01-21 12:03:13.435 03.02 Count-2[26507:903] 1
2012-01-21 12:03:13.436 03.02 Count-2[26507:903] 2
2012-01-21 12:03:13.436 03.02 Count-2[26507:903] 3
2012-01-21 12:03:13.437 03.02 Count-2[26507:903] 4
2012-01-21 12:03:13.437 03.02 Count-2[26507:903] 5
2012-01-21 12:03:13.438 03.02 Count-2[26507:903] 6
2012-01-21 12:03:13.438 03.02 Count-2[26507:903] 7
2012-01-21 12:03:13.438 03.02 Count-2[26507:903] 8
2012-01-21 12:03:13.439 03.02 Count-2[26507:903] 9
2012-01-21 12:03:13.439 03.02 Count-2[26507:903] 10
```

执行这样的简单修改显然不需要太多技巧，你可以通过简单的搜索替换操作来完成，而且只需要改变两处代码。然而在比较庞大的（比如有成千上万行代码的）程序中，执行搜索和替换就麻烦多了。即使只是将5替换成10，也必须要谨慎，这是因为代码中有些数字5是与循环次数无关的，所以不应该更改为10。

变量就是用来解决此类问题的。不需要在代码中直接修改循环的上限值（5或10），我们可以将这个数字放入某个变量中，通过添加一层间接来解决这个问题。使用了变量之后，就是告诉程序去“查看名为count的变量值，它会说明需要执行多少次循环”，而不是此前的“执行5次循环”。现在为更新过的程序命令为Count-3，内容如下所示。

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

int main (int argc, const char * argv[])
{
 int count = 5;

 NSLog (@"The numbers from 1 to %d:", count);

 for (int i = 1; i <= count; i++) {
  NSLog (@"%d\n", i);
 }

 return (0);

} // main
```

该程序的输出结果应当不难猜出：

```
2012-01-21 12:16:51.442 03.03 Count-3[26596:903] The numbers from 1 to 5:
2012-01-21 12:16:51.446 03.03 Count-3[26596:903] 1
2012-01-21 12:16:51.447 03.03 Count-3[26596:903] 2
2012-01-21 12:16:51.448 03.03 Count-3[26596:903] 3
2012-01-21 12:16:51.449 03.03 Count-3[26596:903] 4
2012-01-21 12:16:51.449 03.03 Count-3[26596:903] 5
```

说明　NSLog()时间戳和其他信息会占用大量空间，为简明起见，在以后的输出结果清单中我们将省略这些信息。

如果你想要输出从1到100的数字，只需要修改代码中的一处即可。

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

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

 int count = 100;

 NSLog (@"The numbers from 1 to %d:", count);

 for (int i = 1; i <= count; i++) {
  NSLog (@"%d\n", i);
 }

 return (0);

} // main
```

通过添加变量，代码变得比之前更简洁了，而且也更易于编辑了，尤其是在其他编程人员需要修改此代码时。他们不必为了修改循环次数而仔细查看程序中使用的每个数字5，以确定是否需要修改，而只需要修改count变量的值。

3.1.2　使用文件名的间接

文件是间接的另一个示例。先来看看位于03.04 Word-Length-1文件夹中的程序Word-Length-1，它可以输出多个单词及其长度。这一重要程序就是Web 2.0网页Length-o-words.com中使用的关键技术。下面是代码清单。

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

int main (int argc, const char * argv[])
{
 const char *words[4] = { "aardvark", "abacus",
        "allude", "zygote" };
 int wordCount = 4;
```

```
 for (int i = 0; i < wordCount; i++) {
  NSLog (@"%s is %lu characters long", words[i], strlen(words[i]));
 }

 return (0);

} // main
```

for循环决定了每次处理的是words数组中的哪一个单词。循环里面的NSLog()命令通过%s格式说明符输出单词。之所以使用%s，是因为words是C字符串数组，而不是@"NSString"对象的数组。%lu格式说明符取计算字符串长度的strlen()函数的整数值，并输出单词及其长度。

运行Word-Length-1程序之后，你将看到如下信息：

```
aardvark is 8 characters long
abacus is 6 characters long
allude is 6 characters long
zygote is 6 characters long
```

说明 我们省略了NSLog()添加到Word-Length-1程序输出结果中的时间戳和进程ID。

现在假设为Length-o-words.com投资的风险投资家希望你能使用另一组单词。他们审查过你的计划后决定替换为市场更为广阔的乡村音乐明星的名字。

因为我们将单词直接存储在程序中，所以必须编辑源文件，将原始单词替换为新的名字。在编辑的时候必须要注意标点符号，比如Joe Bob那行名字中的引号和输入项之间的逗号。下面是更新后的程序，可以在03.05 Word-Length-2文件夹中找到。

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

int main (int argc, const char * argv[])
{
 const char *words[4] = { "Joe-Bob \"Handyman\" Brown",
  "Jacksonville \"Sly\" Murphy",
  "Shinara Bain",
  "George \"Guitar\" Books" };
 int wordCount = 4;

 for (int i = 0; i < wordCount; i++) {
  NSLog (@"%s is %lu characters long", words[i], strlen(words[i]));
 }

 return (0);

} // main
```

因为我们很谨慎地修改了代码，所以程序仍然会如我们预期那样正常运行（注意空格和标点符号也会作为字符算入）。

```
Joe-Bob "Handyman" Brown is 24 characters long
Jacksonville "Sly" Murphy is 25 characters long
Shinara Bain is 12 characters long
George "Guitar" Books is 21 characters long
```

进行这样的修改需要大量的工作：每次都必须编辑Word-Length-2.m文件，确保没有录入错误，并且还要重新构建生成程序。如果程序是在网站上运行的，还必须重新测试和部署程序才能升级至Word-Length-2。

构造此程序的另一种方法就是将所有名字都移到代码之外的某个文本文件中，每行一个名字。没错，这就是间接。无需将名字直接放入源代码，而是让程序在其他地方查找这些名字。该程序从一个文本文件中读取一列名字，然后输出名字及其长度。这个新程序的项目文件位于03.06 Word-Length-3文件夹中，代码如下所示。

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

int main (int argc, const char * argv[])
{
 FILE *wordFile = fopen ("/tmp/words.txt", "r");
 char word[100];

 while (fgets(word, 100, wordFile))
  {
  // strip off the trailing \n
  word[strlen(word) - 1] ='\o';

  NSLog (@"%s is %lu characters long", word, strlen(word));
 }

 fclose (wordFile);

 return (0);

} // main
```

我们来浏览一下Word-Length-3程序，看看它是如何工作的。首先，fopen()函数打开words.txt文件以便读取内容，然后fgets()从文件中读取一行文本并将其放入字符数组word中。fgets()调用会保留每行之间用来断行的换行符，但我们并不需要它，因为如果留下它，换行符就会被计为单词中的一个字符。为了解决这个问题，我们将换行符替换为\0，这表示字符串的结束。最后我们使用熟悉的NSLog()输出单词及其长度。

> **说明**　注意一下fopen()使用的路径名称/tmp/words.txt。这意味着words.txt是/tmp目录中的一个文件，/tmp是Unix临时目录，在计算机重启时会被清空。可以使用/tmp来暂放会用到但不必一直保存的临时文件。在真实的程序中，应该将文件放在可以永久存放的位置，例如主目录下。

在运行程序之前，使用文本编辑器在/tmp目录中创建一个words.txt文件。在文件中输入以下名字。

```
Joe-Bob "Handyman" Brown
Jacksonville "Sly" Murphy
Shinara Bain
George "Guitar" Books
```

如果想要使用文本编辑器把文件保存到/tmp目录中，要先输入文件的内容，然后选择Save按钮，按下键盘上的斜杠键（/），并输入tmp，最后按回车键。

如果你愿意的话，也可以不输入名字，直接从03.06 Word-Length-3文件夹中将words.txt文件复制到/tmp中。如果想在Finder中看到/tmp，请点击Go➤Go to Folder按钮。

> **提示** 如果你使用的是我们此前完成的Word-Length-3项目，我们已经在里面使用了一些Xcode小技巧来帮助你把文件复制到/tmp中。看看你是否能发现我们是怎样做的。温馨提示：查看Group & Files窗格中的Targets区域。

运行Word-Length-3程序，输出结果和之前的一样。

```
Joe-Bob "Handyman" Brown is 24 characters long
Jacksonville "Sly" Murphy is 25 characters long
Shinara Bain is 12 characters long
George "Guitar" Books is 21 characters long
```

Word-Length-3是一个出色的间接示例。你不用直接在程序中输入名字，而是用命令查看/tmp/words.txt以获取单词。采用这种方案后，你可以随时更改单词集合，只需编辑文本文件，而不必修改程序。你可以尝试一下向words.txt文件中添加几个单词，然后重新运行程序。不必着急，慢慢来。

这个方法要更好一些，因为文本文件更易于编辑，而且远不像源代码那样易受修改的破坏。你可以让非编程人员使用文本编辑应用来编辑，让市场销售人员替你更新单词列表，这样你就有时间去处理更有兴趣的任务。

我们知道，在升级和改进程序方面人们总是有新想法。也许你的投资人认为计算烹调术语的长度是盈利的新途径。既然你的程序是从文件中获取数据的，你便可以随意修改单词而不必修改代码了。

尽管间接有很多优势，但Word-Length-3仍然相当脆弱，因为它必须使用单词文件的完整路径名，而文件处于不可靠的位置，因为如果计算机重启，/tmp/words.txt文件就会消失。同样，如果其他人使用你机器上的程序来处理他们自己的/tmp/words.txt文件，也可能会无意中覆盖你的文件。你可以每次都修改程序来使用不同路径的文件，但我们都深知这样做很麻烦，所以我们要使用另一种间接的技巧来进一步简化。

这次不再通过/tmp/word.txt路径获取单词，我们将修改这个程序，使其查看程序的第一个启动参数，以确定单词文件的位置。下面是Word-Length-4程序（可以在03.07 Word-Length-4文件夹中找到）。它使用命令行参数来指定文件名。对Word-Length-3所作的修改已突出显示。

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

int main (int argc, const char * argv[])
{
 if (argc == 1) {
  NSLog (@"you need to provide a file name");
  return (1);
 }

 FILE *wordFile = fopen (argv[1], "r");
 char word[100];
 while (fgets(word, 100, wordFile))
  {
  // strip off the trailing \n
  word[strlen(word) - 1] ='\o';

  NSLog (@"%s is %lu characters long", word, strlen(word));
 }

 fclose (wordFile);

 return (0);

} // main
```

用来处理文件的循环部分与Word-Length-3中的相同，但是建立循环的代码已做了更新和改进。If语句验证用户是否提供了路径名作为启动参数。代码通过main()的argc参数得知了启动参数的数量。因为程序名总是用作启动参数而传递，所以argc的值至少是1，也可能更大。如果用户不提供文件路径，那么argc的值将为1，也就是没有文件用来读取了，于是我们输出报错消息并终止程序运行。

如果细心的用户提供了文件路径，那么argc的值将大于1，然后我们可以通过查看argv数组来获知单词文件的路径。argv[1]保存着用户所提供的文件名（为了满足你的好奇心，我可以告诉你argv[0]是程序名）。

如果你在终端应用中运行此程序，那么在命令行中指定文件名会非常容易，如下所示。

```
$ ./Word-Length-4 /tmp/words.txt
Joe-Bob "Handyman" Brown is 24 characters long
Jacksonville "Sly" Murphy is 25 characters long
Shinara Bain is 12 characters long
George "Guitar" Books is 21 characters long
```

在Xcode中提供文件路径

如果你跟我们一样是在Xcode中编辑程序，那么在运行程序时提供文件路径会稍微复杂些。**启动参数**也称为**命令行参数**，在Xcode中设置它比在终端应用中要更困难一点。修改启动参数所需的步骤如下。

首先，在Xcode中点击Product ➤Edit Scheme菜单项并在弹出的面板中选择Arguments选项卡，如图3-1所示。

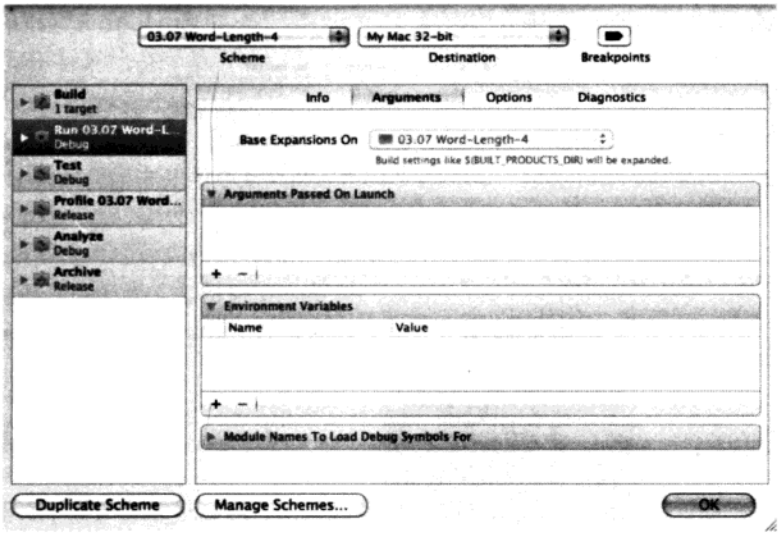


图3-1 Arguments选项卡

接下来，如图3-2所示，点击Arguments Passed On Launch标题下的加号按钮，然后输入启动参数——在本例中使用的是words.txt文件的路径。

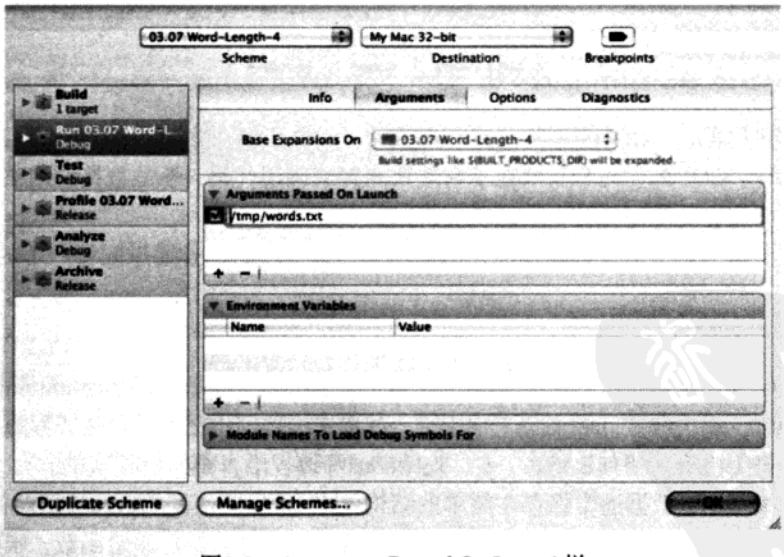


图3-2 Arguments Passed On Launch栏

现在Xcode会在你运行程序的时候将启动参数传递给Word-Length-4程序的argv数组。图3-3是该程序运行后所显示的结果。



图3-3　argv数组输出结果

为了好玩，你可以尝试使用/usr/share/dict/words路径来运行程序，此文件中有23万多个单词。你会惊讶于程序竟能处理这么庞大的数据！如果你不想再看到单词在Xcode的控制窗口内刷屏的话，可以点击停止按钮来终止程序运行。

因为参数是在运行时提供的，所以每个人都能使用你的程序得到他们想要的任一组单词的长度，即使是相当大的单词文件也没有问题。用户可以只修改数据而不必修改代码，这更加符合他们的习惯。这就是间接的本质：它告诉我们从哪里获得需要的数据。

3.2　在面向对象编程中使用间接

间接是OOP的核心。OOP使用间接来获取数据，就像我们在之前的例子中使用变量、文件和参数所做的那样。OOP真正的革命性在于它使用间接来调用代码。不是直接调用某个函数，而是间接调用。

只要理解了这一点，你就算掌握OOP的内涵了。其他一切都是通过间接产生的引申效果。

3.2.1　过程式编程

为了理解OOP的灵活性，我们先来看一下过程式编程（Procedual Programming），这样你就能明白OOP是为了解决哪些问题而创造出来的。过程式编程问世已久，它是编程的入门书籍和课程都会讲解的典型内容。诸如BASIC、C、Tcl和Perl等编程语言都是过程式的。

在过程式编程中，数据通常保存在简单的结构体中（例如C语言的struct元素）。还有一些较为复杂的数据结构，例如链表和树。当调用一个函数时，你将数据传递给函数，函数会处理这些数据。在过程式编程中经常会用到函数：你决定使用什么函数，然后调用它并传递其所需的数据。

1. 绘制几何体的形状

设想某个程序可以在屏幕上绘制各类几何体的形状。依赖强大的计算机可以让你实现这一设

想，你可以在03.08 Shapes-Procedural文件夹内找到该程序的源代码。为简明起见，Shapes-Procedural程序并不会真的在屏幕上绘制图形，而只是输出一些与该形状相关的文本信息。我们省略了绘图代码，以降低程序的复杂度，避免分散注意力，我们应集中精力编写一个可以以相同的方式处理多种元素的程序。

Shapes-Procedural程序中使用的是纯C语言和过程式编程方式。在代码的开始需要定义一些常量和一个结构体。

在Foundation头文件的导入命令（这个一定要有）之后，首先是通过枚举（enumeration）指定几种可以绘制的不同形状（圆形、矩形和椭圆形）。

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

typedef enum {
 kCircle,
 kRectangle,
 kEgg } ShapeType;
```

接下来的枚举定义了绘制形状时可用的几种颜色。

```
typedef enum {
 kRedColor,
 kGreenColor,
 kBlueColor } ShapeColor;
```

然后我们使用一个结构体来描述一个矩形，该矩形指定屏幕上的绘图区域。在本章的示例中，我们暂且不管该矩形用于其他形状的具体方式。

```
typedef struct {
 int x, y, width, height;
} ShapeRect;
```

最后我们用一个结构体将前面的所有内容结合起来，整体地描述一个形状。

```
typedef struct {
 ShapeType type;
 ShapeColor fillColor;
 ShapeRect bounds;
} Shape;
```

2. 接下来的工作

示例中接下来的main()函数声明了我们要绘制的形状的数组。声明完数组之后，数组中每个表示形状的结构体都通过分配空间来进行初始化。下列代码为我们设计了一个红色的圆形、一个绿色的矩形和一个蓝色的椭圆形。

```
int main (int argc, const char * argv[])
{
 Shape shapes[3];

 ShapeRect rect0 = { 0, 0, 10, 30 };
 shapes[0].type = kCircle;
 shapes[0].fillColor = kRedColor;
 shapes[0].bounds = rect0;
```

```
ShapeRect rect1 = { 30, 40, 50, 60 };
shapes[1].type = kRectangle;
shapes[1].fillColor = kGreenColor;
shapes[1].bounds = rect1;

ShapeRect rect2 = { 15, 18, 37, 29 };
shapes[2].type = kEgg;
shapes[2].fillColor = kBlueColor;
shapes[2].bounds = rect2;

drawShapes (shapes, 3);

return (0);

} // main
```

方便的C语言快捷操作

Shapes-Procedural程序的main()方法中的绘图区域是通过C语言中一个小技巧声明的：声明结构体变量时，你可以一次性初始化该结构体的所有元素。

```
ShapeRect rect0 = { 0, 0, 10, 30 };
```

结构体中的元素按照声明的顺序取值。回想一下，ShapeRect是这样声明的：

```
typedef struct {
int x, y, width, height; } ShapeRect;
```

这个初始化语句表示给rect0.x和rect0.y赋值为0，给rect0.width赋值为10，给rect0.height赋值为30。

使用这个技巧可以减少程序中需要输入的字符量，并且不会影响可读性。

初始化完shapes数组之后，main()调用了drawShapes()函数来绘制形状。

drawShapes()中的循环先检查数组中的每个Shape结构体，再通过switch语句查看结构体的type字段并调用适用该形状的绘制函数，传递绘图区域和颜色的参数。其代码如下所示。

```
void drawShapes (Shape shapes[], int count)
{
 for (int i = 0; i < count; i++) {

   switch (shapes[i].type) {

   case kCircle:
    drawCircle (shapes[i].bounds,
       shapes[i].fillColor);
    break;

   case kRectangle:
    drawRectangle (shapes[i].bounds,
         shapes[i].fillColor);
    break;
```

```
    case kEgg:
     drawEgg (shapes[i].bounds,
        shapes[i].fillColor);
     break;

   }
  }

} // drawShapes
```

下面是drawCircle()函数的代码，此函数输出矩形区域信息和传递给它的颜色。

```
void drawCircle (ShapeRect bounds, ShapeColor fillColor)
{
 NSLog (@"drawing a circle at (%d %d %d %d) in %@",
    bounds.x, bounds.y,
    bounds.width, bounds.height,
    colorName(fillColor));

} // drawCircle
```

NSLog()中调用的colorName()函数负责转换传入的颜色值，并返回NSString字面量，比如@"red"或@"blue"。

```
NSString *colorName (ShapeColor colorName)
{
 switch (colorName) {
  case kRedColor:
   return @"red";
   break;
 case kGreenColor:
   return @"green";
   break;
 case kBlueColor:
   return @"blue";
   break;
 }

 return @"no clue";

} // colorName
```

其他绘图函数几乎和drawCircle相同，只不过绘制出的是矩形和椭圆形。

下面是Shape-Procedural程序的输出结果（省略了NSLog()添加的时间戳和其他信息）。

```
drawing a circle at (0 0 10 30) in red
drawing a rectangle at (30 40 50 60) in green
drawing an egg at (15 18 37 29) in blue
```

这一切看起来十分简单明了，对不对？在使用过程式编程时，要花时间连接数据和用来处理该数据的函数。必须注意，要为各种数据类型使用正确的函数。例如，你必须调用drawRectangle()来绘制kRectangle类型的形状。但令人失望的是，有时候一不留神就会将矩形的数据传给处理圆形的函数。

编写这种代码的另一个问题是程序的扩展和维护变得困难了。为了说明这一点，我们为

Shape-Procedural程序添加一种新的形状：三角形。你可以在03.09 Shapes-Procedural-2项目中找到修改后的程序。我们必须修改程序中至少4个不同的位置才能完成该任务。

首先，在ShapeType enum枚举的内容中增加kTriangle常量。

```
typedef enum {
 kCircle,
 kRectangle,
 kEgg,
 kTriangle
} ShapeType;
```

然后，编写看上去和其他函数一样的drawTriangle()函数。

```
void drawTriangle (ShapeRect bounds,
      ShapeColor fillColor)
{
 NSLog (@"drawing triangle at (%d %d %d %d) in %@",
    bounds.x, bounds.y,
    bounds.width, bounds.height,
    colorName(fillColor));

} // drawTriangle
```

接下来，在drawShapes()的switch语句中增加一个新的case判断，用于测试当前形状的type值是不是kTriangle，如果通过的话则调用drawTriangle()函数。

```
void drawShapes (Shape shapes[], int count)
{
 for (int i = 0; i < count; i++) {

  switch (shapes[i].type) {

   case kCircle:
    drawCircle (shapes[i].bounds, shapes[i].fillColor);
    break;

   case kRectangle:
    drawRectangle (shapes[i].bounds, shapes[i].fillColor);
    break;

   case kEgg:
    drawEgg (shapes[i].bounds, shapes[i].fillColor);
    break;

   case kTriangle:
    drawTriangle (shapes[i].bounds, shapes[i].fillColor);
    break;
  }
 }

} // drawShapes
```

最后，为shapes数组增加一个三角形。别忘记在shapes数组中增加形状的数量。

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

```
Shape shapes[4];

ShapeRect rect0 = { 0, 0, 10, 30 };
shapes[0].type = kCircle;
shapes[0].fillColor = kRedColor;
shapes[0].bounds = rect0;

ShapeRect rect1 = { 30, 40, 50, 60 };
shapes[1].type = kRectangle;
shapes[1].fillColor = kGreenColor;
shapes[1].bounds = rect1;

ShapeRect rect2 = { 15, 18, 37, 29 };
shapes[2].type = kEgg;
shapes[2].fillColor = kBlueColor;
shapes[2].bounds = rect2;

ShapeRect rect3 = { 47, 32, 80, 50 };
shapes[3].type = kTriangle;
shapes[3].fillColor = kRedColor;
shapes[3].bounds = rect3;

drawShapes (shapes, 4);

return (0);

} // main
```

让我们来看看Shapes-Procedural-2的运行结果。

```
drawing a circle at (0 0 10 30) in red
drawing a rectangle at (30 40 50 60) in green
drawing an egg at (15 18 37 29) in blue
drawing a triangle at (47 32 80 50) in red
```

添加对三角形的支持并不是非常难，不过我们的小程序仅用于实现一种操作——绘制形状（某种意义上就是输出图形的信息）。程序越复杂，扩展起来就越麻烦。假设程序不仅用于绘制各种形状，还必须能计算这些形状的面积，并判断鼠标光标是否位于这些形状中。在这种情况下，就必须修改每个对形状执行操作的函数，修改过去正常工作的代码很可能会引入新的错误。

还有另一种情况也非常危险：增加新形状需要更多信息来描述。例如，绘制圆角矩形需要知道其他形状所没有的圆角半径。为了支持圆角矩形的绘制，你可以在Shape结构体中增加半径域，但这样会浪费空间，因为绘制其他形状时并不会用到半径域。或者，你也可以使用C语言的联合体（union）来覆盖相同结构体中不同的数据布局，但是把各种形状融入联合体中并获取有用数据的过程也会使问题更加复杂。

OOP完美地解决了这些问题。等会我们在程序中使用OOP的时候，你就会看到OOP如何解决第一个问题——修改现有的代码来增加新的形状。

3.2.2　实现面向对象编程

过程式编程建立在函数之上，数据为函数服务，而面向对象编程则以程序的数据为中心，函数为数据服务。在OOP中，不再重点关注程序中的函数，而是专注于数据。

这听起来非常有趣，但它是如何工作的呢？在OOP中，数据通过间接方式引用代码，代码可以对数据进行操作。不是通知drawRectangle()函数“根据这个形状进行绘制”，而是要求这个形状“绘制自身”（天啊，这听起来太荒谬了，但事实的确如此）。借助间接的强大功能，这些数据能够知道如何查找相应的函数来进行绘制。

那么，对象到底是什么呢？它其实是就像C语言中的struct一样，神奇的是它能够通过函数指针查找与之相关的代码。图3-4中展示了4种Shape对象：两个正方形、一个圆形和一个椭圆形。每个对象都能查找相应的函数并实现其绘图功能。

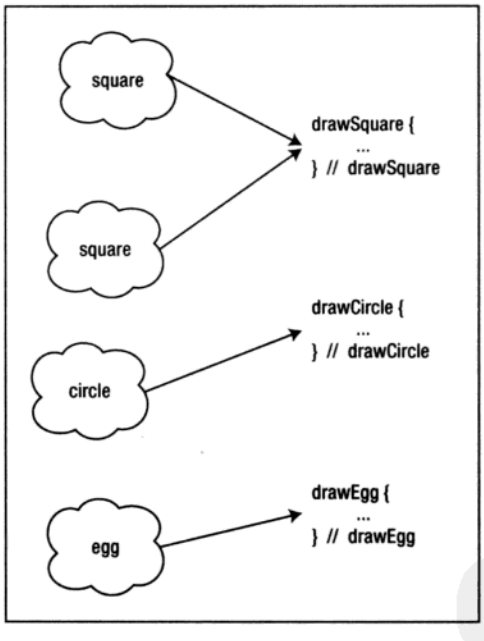


图3-4　基础的Shape对象

每个对象都有自己的draw()函数，知道如何绘制自身的形状。例如Circle对象的draw()函数知道如何绘制圆形，Rectangle对象的draw()函数知道如何绘制矩形。

Shapes-Object程序（源代码位于03.10 Shapes-Object文件夹）可以完成与Shapes-Procedural相同的功能，但它是通过Objective-C的面向对象特性来实现的。以下是Shapes-Object程序drawShapes()函数的代码内容。

```
void drawShapes (id shapes[], int count)
{
 for (int i = 0; i < count; i++) {
```

```
    id shape = shapes[i];
    [shape draw];
  }

} // drawShapes
```

该函数包含一个循环，用于遍历数组中的每个形状。在循环过程中，程序通知形状对象绘制自身。

注意区分该形式的drawShapes()与原始版本有何不同。首先，本函数更为简短，代码不必再检查每个形状的种类。

另一个不同点是函数的第一个参数shapes[]，此时它是一个id数组对象。那么什么是id？是那个与大脑有关的心理学术语，用于评测先天本能反应和初级心理过程的智力商数（Intelligence Quotient）吗？不是，它表示的是标识符（identifier）。id是一种泛型，可以用来引用任何类型的对象。回想一下，对象是一种包含代码的struct结构体，因此id实际上是一个指向结构体的指针。在本示例中，结构体就是那些形状对象。

Draw()函数的第三个变化是循环主体。

```
id shape = shapes[i];
[shape draw];
```

第一行看上去像普通的C语言。代码从shapes数组获取id（即指向某个对象的指针），并将其赋值给名为shape的变量，该变量具有id类型。这只是一种指针赋值过程，它实质上并不会复制shape的全部内容。观察图3-5中Shapes-Object程序中的各个形状。shapes[0]是一个指向红色圆形的指针，shapes[1]是一个指向绿色矩形的指针，shapes[2]是一个指向蓝色椭圆形的指针。

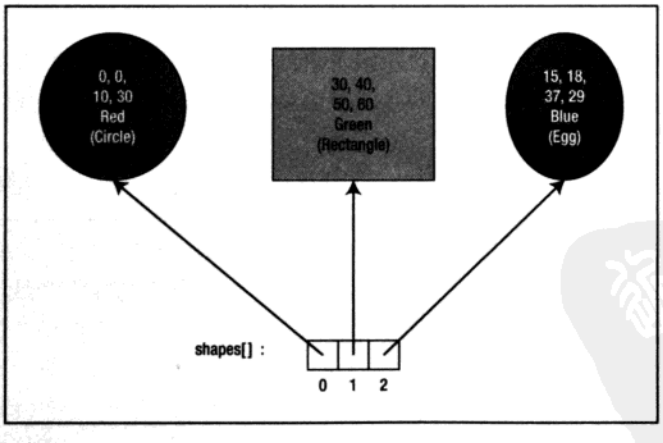


图3-5 shapes数组

现在，我们来看看函数的最后一行代码：

```
[shape draw];
```

非常奇怪吧。这是怎么回事呢？我们知道，C语言使用方括号来表示数组，但我们在这里并

没有打算使用数组的任何功能。原来在Objective-C中方括号还有其他意义：用于通知某个对象该去做什么。方括号里的第一项是对象，其余部分是需要对象执行的操作。在本示例中，我们通知名称为shape的对象执行draw操作。如果shape是圆形，我们会得到圆形；如果shape是矩形，我们会得到矩形。

在Objective-C中，通知对象执行某种操作称为发送消息（有些人也称之为“调用方法”）。代码[shape draw]表示向shape对象发送了draw消息。[shape draw]可以理解成“向shape发送draw消息”。至于形状如何实际绘制自身图形，则取决于shape的实现。

向对象发送消息后，如何调用所需的代码呢？这就需要叫做类的幕后帮手来协助完成。

请看图3-6。该图左侧展示了shapes数组中索引为0的circle对象，该对象最近在图3-5中出现过。circle对象含有一个指向其类的指针。类是一种能够实例化成对象的结构体。在图3-6中，Circle类含有一个指针指向用于绘制圆形、计算圆形的面积以及实现其他与圆形相关的必要功能的代码。

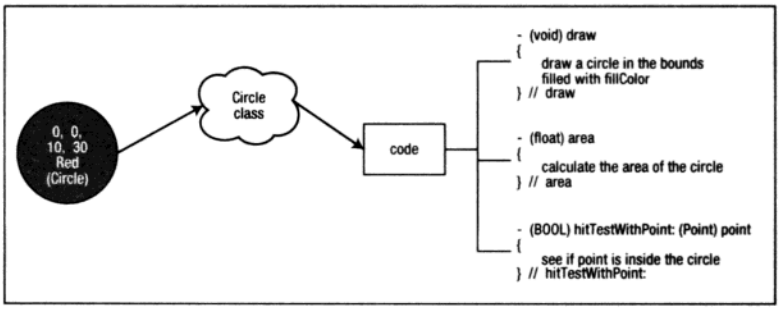


图3-6　circle对象和它的类

类对象有什么用呢？让每个对象直接指向各自的代码不是更简单吗？确实是更简单一些，而且某些OOP系统也是那样做的。但是拥有类对象会具备极大的优势：如果在运行时改变某个类，则该类的所有对象会自动继承这些变化（我们将在后面各章中进一步讨论）。

图3-7展示了draw消息如何调用circle对象中适当的函数。

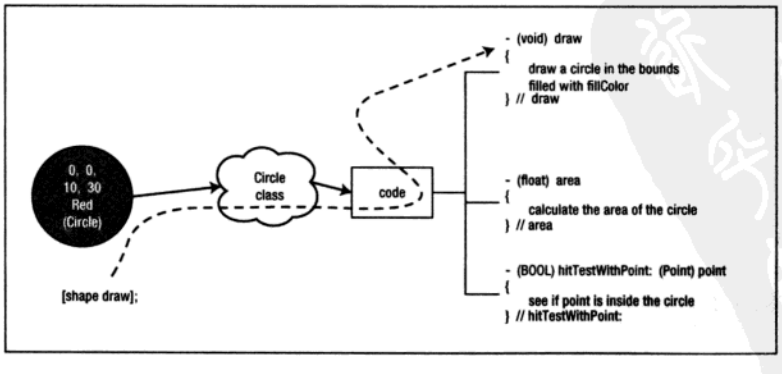


图3-7　circle对象查找draw函数代码的过程

以下是图3-7中所演示的步骤。

(1) 对象（当前是圆形）是消息的目标，需要查出它属于哪一个类。

(2) 在Circle类中浏览其代码，查找draw函数的位置。

(3) 找到draw函数后，执行绘制圆形的函数。

图3-8演示了数组中的第二个形状（矩形）调用[shape draw]时的情景。

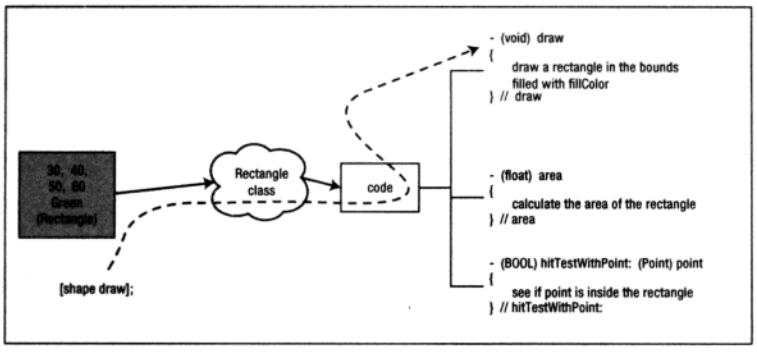


图3-8 rectangle对象查找draw函数代码的过程

图3-8中使用的步骤和图3-7中的步骤几乎相同：

(1) 查询消息的目标对象（当前是矩形）属于哪个类。

(2) Rectangle类查找其代码块，然后获取draw函数的地址。

(3) Objective-C运行可以绘制矩形的代码。

该程序展示了一些非常棒的间接操作！在该程序的过程式版本中，我们必须编写代码来决定要调用哪个函数。现在，可以由Objective-C在幕后决定，它将查询对象属于哪个类。这可以降低调用错误函数的几率，同时使代码更易于维护。

3.3 有关术语

在深入研究Shape-Object程序之前，先介绍一些有关面向对象的术语，其中一些已经讨论过。

- 类（class）是一种表示对象类型的结构体。对象通过它的类来获取自身的各种信息，尤其是执行每个操作需要运行的代码。简单的程序可能仅包含少量的类，中等复杂的程序会包含十几个类。建议开发人员在用Objective-C编程时采用首字母大写的类名。
- 对象（object）是一种包含值和指向其类的隐藏指针的结构体。运行中的程序通常都包含成百上千个对象。指向对象的变量通常不需要首字母大写。
- 实例（instance）是“对象”的另一种称呼。比方说circle对象也可以称为Circle类的实例。
- 消息（message）是对象可以执行的操作，用于通知对象去做什么。在[shape draw]的代码中，通过向shape对象发送draw消息来通知对象绘制自身。对象接收消息后，将查询相应的类，以便找到正确的代码来运行。

- 方法（method）是为响应消息而运行的代码。根据对象的类，消息（比如draw）可以调用不同的方法。
- 方法调度（method dispatcher）是Objective-C使用的一种机制，用于推测执行什么方法以响应某个特定的消息。我们将在下一章深入讨论Objective-C的方法调度机制。

以上是一些关键的OOP术语，本书后续章节中会用到这些术语。另外，还有以下两个重要的泛型编程术语。

- 接口（interface）是类为对象提供的特性描述。例如，Circle类的接口声明了Circle类可以接受draw消息。
- 实现（implementation）是使接口能正常工作的代码。在我们的示例中，circle对象的实现中含有在屏幕上绘制圆形的代码。向circle对象发送draw消息时，你不会知道也不必知道函数是如何工作的，只需知道它能在屏幕上画一个圆就可以了。

> **说明**　接口的概念不只用在OOP中。例如C语言的头文件提供了库接口，比如标准I/O库（通过#include<stdio.h>获取）和数学库（通过#include<math.h>获取）。接口不提供实现代码的细节信息，也就是说你不必了解。

3.4　Objective-C 语言中的 OOP

如果你现在开始有些头痛了，那是很正常的。因为我们向你的头脑中灌输了许多新知识，消化吸收所有的术语和技术需要一定的时间。在消化前两节内容的同时，让我们来看一看Shapes-Object代码的其余部分，其中包含一些声明类的新语法。

3.4.1　@interface部分

创建某个特定类的对象之前，Objective-C编译器需要一些有关该类的信息，尤其是对象的数据成员（即对象的C语言类型结构体应该是什么样子）及其提供的功能。可以使用@interface指令把这些信息传递给编译器。

> **说明**　在Shapes-Object程序中，我们将所有内容都放在它的Shapes-Object.m文件里。而在大型程序中，则需要使用多个文件，每个类都有自己的文件。我们将在第6章介绍组织类和文件的方式。

以下是Circle类的接口。

```
@interface Circle : NSObject
{
@private
```

```
 ShapeColor fillColor;
 ShapeRect bounds;
}
- (void) setFillColor: (ShapeColor) fillColor;

- (void) setBounds: (ShapeRect) bounds;

- (void) draw;

@end // Circle
```

以上代码含有一些我们尚未介绍的语法，下面将进行介绍。这几行代码包含了大量信息，我们来一一分析。

第一行代码如下所示：

```
@interface Circle : NSObject
```

我们在第2章中说过，在Objective-C中只要看到@符号，就可以把它看成是对C语言的扩展。@interface Circle告诉编辑器："这是新类Circle的接口。"

> **说明** @interface行中的NSObject告诉编译器，Circle类是基于NSObject类的。该语句表明每个Circle类都是一个NSObject，并且每个Circle类都将继承NSObject类定义的所有行为。我们将在下一章详细介绍更多关于继承的内容。

声明完新类之后，我们将告诉编译器Circle对象需要的各种数据成员。

```
{
 ShapeColor fillColor;
 ShapeRect bounds;
}
```

花括号中的内容是用于大量创建新Circle对象的模板。它表示创建的新对象由两个元素构成。第一个元素是fillColor，属于ShapeColor类型，是所绘制圆形的颜色。第二个元素是bounds，是圆形所在的矩形区域，属于ShapeRect类型。该元素用于确定在屏幕上绘制圆形的位置。

在类声明中指定fillColor和bounds后，每次创建Circle对象，对象中都将包括这两个元素。因此，每个Circle类对象都将拥有自己的fillColor和bounds。fillColor和bounds的值称为Circle类的*实例变量*（instance variable）。

结尾处的花括号告诉编译器，我们为Circle类指定了实例变量。

接下来几行代码看起来有点像C语言中的函数原型。

```
- (void) draw;
- (void) setFillColor: (ShapeColor) fillColor;
- (void) setBounds: (ShapeRect) bounds;
```

在Objective-C中，它们称为*方法声明*（method declaration）。它们看起来很像是旧式的C语言函数原型，用于说明"这是我支持的功能"。方法声明列出了每个方法的名称、方法返回值的类型和某些参数。

我们从最简单的draw方法开始：

```
- (void) draw;
```

前面的短线表明这是Objective-C方法的声明。这是区分函数原型与方法声明的一种方式，函数原型中没有先行短线。短线后面是方法的返回类型，位于圆括号中。在我们的示例中，draw方法仅用于绘制图形，并不返回任何值。Objective-C使用void表示无返回值。

Objective-C方法可以返回与C函数相同的类型：标准类型（整型、浮点型和字符串）、指针、引用对象和结构体。

接下来的方法声明更有意思：

```
- (void) setFillColor: (ShapeColor) fillColor;
- (void) setBounds: (ShapeRect) bounds;
```

其中的每个方法都有一个参数。setFillColor:有一个颜色参数，Circle类在绘制自身时会使用该颜色。setBounds:有一个矩形区域参数，Circle类使用该区域来确定它们的边界。

中缀符

Objective-C有一种名为**中缀符**（infix notation）的语法技术。方法的名称及其参数都是合在一起的。例如，你可以这样调用带一个参数的方法：

```
[circle setFillColor: kRedColor];
```

带两个参数的方法调用如下所示：

```
[textThing setStringValue: @"hello there" color: kBlueColor];
```

setStringValue:和color:是参数的名称（事实上它们是方法名称的一部分，后面会详细介绍），@"hello there"和kBlueColor是被传递的参数。

这种语法和C不同，在C语言中调用函数时，是把所有的参数都放在函数名之后，如下所示。

```
setTextThingValueColor (textThing, @"hello there", kBlueColor);
```

我们非常喜欢中缀语法，虽然乍一看有些古怪。它使代码的可读性更强，参数的用途更容易理解。使用C和C++时，有时候你需要在函数中使用4个或5个参数，如果不查询相关文档，就很难确切了解每个参数的功能。

setFillColor：声明以常见的先行短线和位于圆括号中的返回值类型开头：

```
- (void)
```

与draw方法一样，此处的先行短线表明“这是一个新方法的声明”。(void) 表明该方法不返回任何值。我们继续看以下代码：

```
setFillColor:
```

方法的名称是setFillColor:，结尾处的冒号是名称的一部分，它告诉编译器和编程人员后面会出现参数。

```
(ShapeColor) fillColor;
```

参数的类型是在圆括号中指定的。例如在本例中，它是某个ShapeColor枚举值（kRedColor、kBlueColor等）。紧随其后的名称fillColor是参数名。你可以在方法的主体中使用该名称引用参数。为了增强代码的可读性，可以选择有意义的参数名称，而不要以你的宠物或最喜欢的超级英雄命名。

注意冒号

注意，冒号是方法名称非常重要的组成部分。方法

```
- (void) scratchTheCat;
```

不同于

```
- (void) scratchTheCat: (CatType) critter;
```

许多初级Objective-C编程人员常犯的一个错误是在不含有参数的方法结尾乱添冒号。面对编译器错误，你可能会因为某个多余的冒号而不知所措，并希望能解决这样的错误。可以遵循这样一个规则：如果方法使用参数，则需要冒号，否则不需要冒号。

除了参数类型是ShapeRect而不是ShapeColor以外，方法serBounds:和serFillColor:的声明完全相同。

最后一行代码告诉编译器，我们已经完成了Circle类的声明。

```
@end // Circle
```

虽然这并不是必需的，但我们还是提倡在所有的@end语句后添加注释来注明类的名称。如果你已经跳到了文件的结尾处，或者位于大型文件的最后一页，通过注释可以很容易地知道当前看的是什么。

这就是完整的Circle类接口。现在，任何阅读该代码的人都知道，该类有两个实例变量和3个方法。一个方法用来设置边界区域，一个方法用来设置颜色，另一个方法用来绘制形状。

现在，我们已经完成了对接口的介绍。接下来我们将编写代码，使该类能真正实现某些功能。你该不会以为我们就此结束了吧？

3.4.2 @implementation部分

刚才我们讨论了@interface部分，它用于定义类的公共接口。通常，接口被称为API（application programming interface的三个首字母缩写）。而真正使对象能够运行的代码位于@implementation部分中。

以下是完整的Circle类实现：

```
@implementation Circle

- (void) setFillColor: (ShapeColor) c
{
 fillColor = c;
```

```
} // setFillColor

- (void) setBounds: (ShapeRect) b
{
 bounds = b;
} // setBounds
```

现在，我们将按照惯例详细解释这些代码。Circle的实现由以下代码开始：

```
@implementation Circle
```

@implementation是一个编译器指令，表明你将为某个类提供代码。类名出现在@implementation之后。该行的结尾处没有分号，因为在Objective-C编译器指令后不必使用分号。

接下来是各个方法的定义。它们不必按照在@interface指令中的顺序出现。你甚至可以在@implementation中定义那些在@interface中没有声明过的方法。你可以把它们看作是仅能在当前类实现中使用的私有方法。

> **说明**　你也许会认为，既然单独在@implementation指令中定义方法，就不能从该实现之外访问该方法。但事实并非如此，Objective-C中不存在真正的私有方法，也无法把某个方法标识为私有方法，从而禁止其他代码调用它。这是Objective-C动态本质的副作用。

要定义的第一个方法是：setFillColor:。

```
- (void) setFillColor: (ShapeColor) c
{
 fillColor = c;
} // setFillColor
```

setFillColor:定义的第一行看上去与@interface部分的声明非常类似，二者的主要区别是结尾处有没有分号。也许你已经注意到，我们把参数重新命名为简单的字符c了。@interface和@implementation间的参数名不同是允许的。在这里，如果我们继续使用参数名fillColor，就会覆盖fillColor实例变量，并且编译器会生成警告信息。

> **说明**　为何一定要给fillColor重新命名呢？我们已经通过类定义了一个名为fillColor的实例变量，因为它在作用域范围内，可以在该方法中引用该变量，所以如果使用相同的名称定义另一个变量，编译器将会阻止我们访问该实例变量。使用相同的变量名会覆盖初始变量，为参数取一个新的名称可以避免该问题。也可以将实例变量改为其他名称（如myFillColor），这样就可以继续把fillColor作为参数的名称了。在第18章中你将看到，如果为实例变量取一个和方法名相同的名字，Cocoa还可以发挥出某些神奇的威力。

在@interface部分的方法声明中使用名称fillColor，是为了告诉读者参数的真正作用。在实现中，我们必须区分参数名称和实例变量的名称，最简单的方式就是将参数重新命名。

该方法的主体只有一行代码：

```
fillColor = c;
```

如果你好奇心很重的话，可能会对实例变量的存储位置感兴趣。在Objective-C中调用方法时，一个名为self的秘密隐藏参数将被传递给接收对象，而这个参数引用的就是该接收对象。例如，在代码[circle setFillColor: kRedColor]中，方法将circle作为其self参数进行传递。因为self的传递过程是秘密和自动的，所以你不必亲自来实现。方法中引用实例变量的代码与下面的相似：

```
self->fillColor = c;
```

顺便提一下，传递隐藏的参数是间接操作的又一个示例。（你可能以为我们已经讨论完所有的间接方式了吧？）因为Objective-C运行时（runtime）可以将不同的对象当成隐藏的self参数传递，所以那些对象的实例变量发生更改时，运行时也可进行相应的更改。

> **说明** Objective-C运行时是指用户运行应用程序时，支持这些应用程序（包括我们自己的应用程序）的代码块。运行时负责执行非常重要的任务，如向对象发送消息和传递参数。从第9章开始我们将介绍更多有关运行时的内容。

第二个方法setBounds:与之前的setFillColor:方法很相似：

```
- (void) setBounds: (ShapeRect) b
{
 bounds = b;
} // setBounds
```

以上代码用于设置圆形对象的边界区域，圆形对象将被绘制在参数接收的矩形区域中。

最后一个方法是draw方法。注意，方法名的结尾处没有冒号，说明它不使用任何参数。

```
- (void) draw
{
 NSLog (@"drawing a circle at (%d %d %d %d) in %@",
    bounds.x, bounds.y,
    bounds.width, bounds.height,
    colorName(fillColor));
} // draw
```

draw方法使用隐藏的self参数查找其实例变量的值，这和setFillColor:和setBounds:方法一样。然后，draw方法使用NSLog()输出文本。

其他类（Rectangle和Egg）的@interface和@implementation几乎和Circle类的完全一样。

3.4.3 实例化对象

我们最后要介绍的是Shapes-Object程序中非常关键的过程。在该过程中，我们可以创建生动的形状对象，比如红色的圆形和绿色的矩形。这个过程的专业术语叫做实例化（instantiation）。实例化对象时，需要分配内存，然后将这些内存初始化并保存为有用的默认值，这些值不同于通过新分配的内存获得的随机值。内存分配和初始化工作完成之后，就意味着新的对象实例已经创建好了。

说明　由于对象的局部变量只在对象的实例中有效，因此我们称它们为实例变量，通常简写为ivar。

为了创建一个新的对象，我们需要向相应的类发送new消息。该类接收并处理完new消息后，我们就会得到一个可以使用的新对象实例。

Objective-C具有一个极好的特性，你可以把类当成对象来发送消息。对于那些不局限于某个特定的对象而是对全体类都通用的操作来说，这非常便捷。最好的例子就是给新对象分配空间，如果你要创建一个新的circle对象，向Circle类发送消息要比向某个已存在的circle对象发送消息更加合适。

以下是Shapes-Object程序的main()函数，它用于创建圆形、矩形和椭圆形。

```
int main (int argc, const char * argv[])
{
 id shapes[3];

 ShapeRect rect0 = { 0, 0, 10, 30 };
 shapes[0] = [Circle new];
 [shapes[0] setBounds: rect0];
 [shapes[0] setFillColor: kRedColor];

 ShapeRect rect1 = { 30, 40, 50, 60 };
 shapes[1] = [Rectangle new];
 [shapes[1] setBounds: rect1];
 [shapes[1] setFillColor: kGreenColor];

 ShapeRect rect2 = { 15, 19, 37, 29 };
 shapes[2] = [Egg new];
 [shapes[2] setBounds: rect2];
 [shapes[2] setFillColor: kBlueColor];

 drawShapes (shapes, 3);

 return (0);

} // main
```

可以看到，Shapes-Object程序的main()函数与Shapes-Procedural程序的主函数非常类似。但是，还存在一些区别：Shapes-Object含有id数组元素（还记得吧，它是指向任意类型对象的指针），而不是shapes数组。通过向需要创建对象的类发送new消息，可以创建多个独立的对象。

```
shapes[0] = [Circle new];
…
shapes[1] = [Rectangle new];
…
shapes[2] = [Egg new];
```

另一个不同之处是，Shapes-Procedural程序通过直接设置struct的成员来初始化对象，但Shapes-Object程序并不是直接设置对象的值，而是使用消息请求每个对象设置它自身的边界区域和填充颜色。

```
[shapes[0] setBounds: rect0]; [shapes[0] setFilIColor: kRedColor];

[shapes[1] setBounds: recti]; [shapes[1] setFilIColor: kCreenColor];

[shapes[2] setBounds: rect2]; [shapes[2] setFilIColor: kBlueColor];
```

完成初始化以后，就可以使用前面介绍的drawShapes()函数绘制图形了，代码如下所示。

```
drawShapes (shapes, 3);
```

3.4.4 扩展Shapes-Object程序

还记得我们在Shapes-Procedural程序中增加了绘制三角形的功能吗？下面我们在Shapes-Object中也增加同样的功能。这次操作就简单多了。你可以在源代码资源中的03.11 Shapes-Object-2文件夹里找到该项目对应的程序。

要在Shapes-Procedural-2中添加绘制三角形的功能，必须做很多工作：修改ShapeType枚举类型，添加drawTriangle()函数，在形状列表中添加三角形以及修改drawShapes()函数。其中很多工作都极具挑战性，尤其是对drawShapes()函数的修改，必须编辑控制所有形状绘制的循环代码，在此过程中，有可能会引入一些错误。

对Shapes-Object-2而言，我们仅需完成两件事情：创建新的Triangle类，然后将Triangle对象添加到将要绘制的对象列表中。

以下是Triangle类，它碰巧和Circle类几乎完全相同，只需将出现Circle的地方改成Triangle即可（很明显我们只是输出文本信息，而不是真的要画图。假如真的要绘图的话，代码肯定比现在还要复杂）。

```
@interface Triangle : NSObject
{
 ShapeColor fillColor;
 ShapeRect bounds;
}

- (void) setFillColor: (ShapeColor) fillColor;

- (void) setBounds: (ShapeRect) bounds;

- (void) draw;

@end // Triangle

@implementation Triangle

- (void) setFillColor: (ShapeColor) c
{
 fillColor = c;
} // setFillColor

- (void) setBounds: (ShapeRect) b
{
```

```
 bounds = b;
} // setBounds

- (void) draw
{
 NSLog (@"drawing a triangle at (%d %d %d %d) in %@",
    bounds.x, bounds.y,
    bounds.width, bounds.height,
    colorName(fillColor));
} // draw

@end // Triangle
```

说明　剪切和粘贴的编程风格（如Triangle类）的一个缺点是容易出现大量重复的代码（如setBounds:和setFillColor:方法）。我们将在下一章介绍继承，它能有效避免冗余代码。

接下来，我们需要编辑main()函数来创建新三角形。首先，需要把shapes数组的大小由3改为4，以便有足够的空间来存储新对象：

```
id shapes[4];
```

然后，添加可创建新Triangle对象的代码块，这和创建新Rectangle和Circle对象一样。

```
ShapeRect rect3 = { 47, 32, 80, 50 };
shapes[3] = [Triangle new];
[shapes[3] setBounds: rect3];
[shapes[3] setFillColor: kRedColor];
```

最后，用shapes数组的新对象重新调用drawShapes()方法：

```
drawShapes (shapes, 4);
```

这样，我们的程序就能绘制三角形了。

```
drawing a circle at (0 0 10 30) in red
drawing a rectangle at (30 40 50 60) in green
drawing an egg at (15 19 37 29) in blue
drawing a triangle at (47 32 80 50) in red
```

注意，我们可以添加这个新功能，而不必改变drawShapes()函数或任何其他处理形状的函数。这都是面向对象编程的实用优势。

说明　Shapes-Object-2中的代码正好验证了面向对象编程大师Bertrand Meyer的开放/关闭原则（Open/Closed Principle），即软件实体应该对扩展开放，而对修改关闭。drawShapes()函数对扩展是开放的，仅需向数组添加要绘制的某个新形状。同时，drawShapes()也是对修改关闭的，我们可以扩展它，而不必修改它。应对变化时，遵循开放/关闭原则的软件会更加坚实耐用，因为你不必修改那些可正常运行的代码。

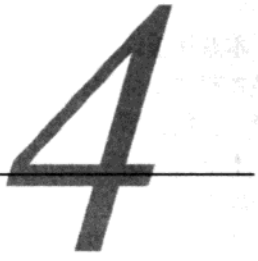
3.5 小结

本章涵盖许多重要概念和定义，篇幅较长。首先介绍了间接这个强大的概念，并指出其实你已经在程序中使用过间接技术（如变量和文件的使用）。然后讨论了过程式编程，并指出了“函数第一，数据第二”这种观念所导致的局限性。

本章还介绍了面向对象的编程，这种编程方法使用间接技术将数据和对数据执行操作的代码紧密联系在一起，它坚持“数据第一，函数第二”的编程风格。此外还讨论了消息，它们被发送给对象，对象通过执行方法（代码块）来处理这些信息。另外，你还了解了每个方法调用都包括一个名为self的隐藏参数，它是对象自身。使用self参数后，方法可以查找并操作对象的数据。方法的实现和对象数据的模板是由对象的类定义的。可以通过向类发送new消息来创建新对象。

下一章我们将介绍继承，该特性可以让你充分利用现有对象的行为，编写更少的代码来完成工作。听起来很棒吧，我们下一章再见！

第4章 继 承



编写面向对象的程序时（希望你将来能写很多），你所创建的类和对象之间存在一定的关系。它们协同工作才能使程序实现功能。

处理类和对象的关系时，尤其要重视OOP的两个方面。第一个方面是**继承**（**inheritance**），也是本章的主题。创建一个新类时，通常需要根据它与现有类的区别来定义。使用继承可以定义一个具有父类所有功能的新类，即它继承了父类的功能。

另一个和类有关的OOP技术是复合（composition），也就是在对象中可以再引用其他对象。例如，在游戏过程中，赛车模拟程序中的car（汽车）对象含有4个tire（轮胎）对象。对象引用其他对象时，可以利用其他对象提供的特性，这就是复合。我们将在下一章介绍与复合有关的内容。

4.1 为何使用继承

还记得上一章中的Shapes-Object程序吗？该程序包含几个接口和实现都非常相似的类。当然这是由于我们采用剪切和粘贴的方式编写了这些类，所以非常相似。

我们复习一下Circle和Rectangle类的接口声明，代码如下所示。

```
@interface Circle : NSObject
{
@private
 ShapeColor fillColor;
 ShapeRect bounds;
}
- (void) setFillColor:(ShapeColor)fillColor;
- (void) setBounds:(ShapeRect)bounds;
- (void) draw;
@end // Circle
@interface Rectangle : NSObject
{
@private
 ShapeColor fillColor;
 ShapeRect bounds;
}
- (void) setFillColor:(ShapeColor)fillColor;
- (void) setBounds:(ShapeRect)bounds;
- (void) draw;
@end // Rectangle
```

这些类的接口非常相似。事实上，除了类的名称不同，其他都是相同的。

此外Circle和Rectangle的实现也非常相似。回想一下，在前面一章中这两个类的setFillColor:和setBounds:方法的实现也完全相同，代码如下所示。

```
@implementation Circle
- (void)setFillColor:(ShapeColor)c
{
   fillColor = c;
} // setFillColor
- (void)setBounds:(ShapeRect)b
{
 bounds = b;
} // setBounds
// …
@end // Circle
@implementation Rectangle
- (void) setFillColor: (ShapeColor) c
{

   fillColor = c; } // setFillColor
- (void)setBounds:(ShapeRect)b
{
 bounds = b;
} // setBounds
// …
@end // Rectangle
```

这些方法具有完全相同的功能——为fillColor和bounds变量赋值。然而Circle和Rectangle的实现方式并不相同，比如说draw方法的名称和参数一样，但是实现却有所不同。

```
@implementation Circle // …
- (void) draw
{
   NSLog (@"drawing a circle at (%d %d %d %d) in %@",
   bounds.x, bounds.y,
   bounds.width, bounds.height,
   colorName(fillColor));
} // draw @end // Circle
@implementation Rectangle
// …
- (void) draw
{
   NSLog (@"drawing rect at (%d %d %d %d) in %@",
   bounds.x, bounds.y,
   bounds.width, bounds.height,
   colorName(fillColor));
} // draw @end // Rectangle
```

显然在Shapes-Object程序中，Circle和Rectangle类的大量代码和行为都是相同的。图4-1是这两个类的构成图。

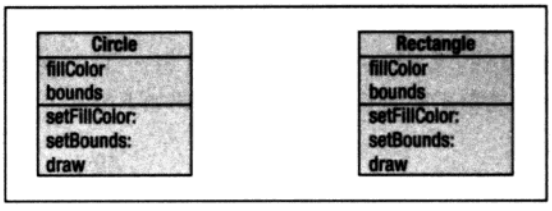


图4-1　没有继承的Shapes-Object架构

> **说明**　在图4-1中，类名位于每个方框的顶部，中间部分是实例变量，底部是类所提供的方法。这种图表是根据UML（Unified Modeling Language，统一建模语言）定义的，UML是一种用图表来表示类、类的内容以及它们之间关系的常见方式。

图4-1中包含了大量重复的内容，这样看起来很没有效率。编程时出现这样的重复内容就意味着它是一个失败的架构。你需要维护两倍的代码，修改代码时必须修改两处（或更多处），这样做很容易出错。如果你忘记更改其中一处的代码，一些奇怪的bug就有可能会出现。

如果能将所有重复的内容合并在一处，还能在需要的地方拥有自定义的方法就更好了（比如绘制圆形和矩形的draw方法各不一样）。我们需要一个功能，能够通知编译器“Circle类与其他类一样，只是其中几个地方略有调整”。嗯，你可能已经猜到了，继承恰好是支持该功能的强大OOP特性。

图4-2展示了使用了继承特性后的构架。我们已经创建了一个全新的类Shape，它用于保存共有的实例变量和声明方法。类Shape还包含了setFillColor:和setBounds:的实现代码。

观察一下图4-2中的新Circle类和Rectangle类，它们比以前小了很多。所有的公共元素都被放入Shape中。Circle类和Rectangle类中仅留下了各自所特有的元素，尤其是draw方法。现在我们可以说Circle类和Rectangle类都是从Shape类继承而来的。

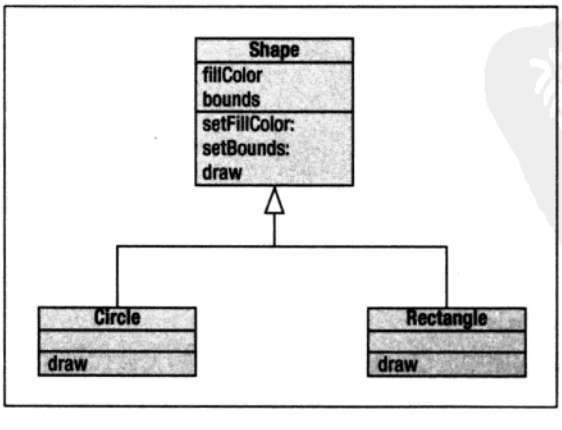


图4-2　通过继承方式改进后的Shapes-Object程序架构

> **说明** 如图4-2所示，UML使用末端带有箭头的竖线表示继承关系。这种竖线表明了Circle和Shape之间以及Rectangle和Shape之间的继承关系。

正如你会从亲生父母那里继承一些特性（例如头发的颜色、鼻子的形状）一样，OOP中的继承表明一个类从另一个类——它的父类或超类（superclass）——中获取了某些特性。由于Circle和Rectangle是从Shape类继承而来的，因此它们将获得Shape类的两个实例变量。

> **说明** 直接更改由继承得到的实例变量的值是一种不好的习惯。一定要通过方法或property属性（本书后面会讲到）来更改。

除实例变量之外，继承还会引入一些方法。所有的Circle对象和Rectangle对象都知道如何去响应setFillColor:和setBounds:方法的调用，因为它们从Shape类中继承了该功能。

4.2 继承的语法格式

先来看一下用于声明新类的语法：@interface Circle: NSObject。

冒号后面的标识符是需要继承的类。在Objective-C中，你可以选择不继承，但如果你使用的是Cocoa框架，就需要继承NSObject类，因为它提供了大量有用的特性（当继承一个继承自NSObject类的类时，也可以获取这些特性）。在第9章介绍内存管理时我们将更深入地讨论NSObject的特性。

只能继承一个

某些编程语言（例如C++）具有多继承性，也就是一个类可以直接从两个或多个类继承而来。但Objective-C不支持多继承。如果你尝试在Objective-C中使用多继承（多继承的形式类似于以下语句），是无法正常通过编译器审核的。

```
@interface Circle : NSObject, PrintableObject
```

你可以通过Objective-C的其他特性来达到多继承的效果，例如类别（category，请参阅第12章）和协议（protocol，请参阅第13章）。

现在，你已经熟悉了继承的语法，接下来我们将进一步改进架构，使我们的类从Shape类继承而来。将Circle和Rectangle的接口代码更改为以下形式（可在04.01 Shapes-Inheritance文件夹中找到该程序的代码）。

```
@interface Circle : Shape
@end // Circle

@interface Rectangle : Shape
@end // Rectangle
```

以上代码精简得不能再精简了。只有代码精简，bug才无处藏身。

现在不必再声明实例变量了，因为我们可以从Shape类中继承。可以看到，由于实例变量不存在了，也就不再需要花括号了（程序中没有声明实例变量时可以省略花括号）。另外也不必声明从Shape类中获取的方法（setBounds:和setFillColor:）了。

现在看一下Shape类的代码。以下是Shape类的接口声明：

```
@interface Shape : NSObject {
  ShapeColor fillColor;
  ShapeRect bounds;
}
- (void) setFillColor:(ShapeColor)fillColor;
- (void) setBounds:(ShapeRect)bounds;
- (void) draw;
@end // Shape
```

可以看出，Shape类将前面不同的类中所有重复的代码都绑定到了一个包中。

Shape类的实现代码也很简单并且很常见。

```
@implementation Shape
- (void)setFillColor:(ShapeColor)c
{
 fillColor = c;
} // setFillColor
- (void)setBounds:(ShapeRect)b
{
   bounds = b;
} // setBounds
- (void) draw
{
} // draw
@end // Shape
```

虽然draw方法没有实现任何功能，但我们仍然需要定义它，以便Shape的所有子类都能通过它来实现各自的方法。在方法的定义中不写任何内容或返回一个虚（dummy）值都是可以通过编译的。

让我们现在来看看Circle类的实现。不出你所料，现在它变得非常简单了。

```
@implementation Circle
- (void) draw
{
   NSLog(@"drawing a circle at (%d %d %d %d) in %@",
   bounds.x, bounds.y,
   bounds.width, bounds.height,
   colorName(fillColor));
} // draw
@end // Circle
```

接下来是新简化过的Rectangle类的实现：

```
@implementation Rectangle
- (void)draw
{
 NSLog(@"drawing rect at (%d %d %d %d) in %@",
       bounds.x, bounds.y,
       bounds.width, bounds.height,
       colorName(fillColor));
```

```
} // draw
@end // Rectangle
```

Triangle和Egg类的实现也同样精简了。详细内容请参见04.01 Shapes-Inheritance文件夹中的程序代码。

现在运行Shapes-Inheritance程序，可以看出它的功能和之前相比没有变化。奇妙的是，我们不需要更改main()函数中设置和使用对象的任何代码，因为我们没有改变对象响应的方法，也没有修改它们的行为。

> **说明** 这种移植和优化代码的方式称为重构（refactoring）。这在OOP中是一个非常流行的主题。进行重构时，会通过移植某些代码来改进程序的架构，正如我们在这里删除重复的代码一样，而不必改变代码的行为和运行结果。通常开发周期包括向代码中添加某些特性，然后通过重构删除所有重复的代码。
>
> 有时在面向对象的程序中添加某些新特性之后，程序反而变得更简单，就像我们在当前程序中添加了Shape类后出现的情况一样。你可能会对此感到不可思议。

有关术语

新技术的产生总是伴随着新术语的出现。为了全面理解继承的机制，你需要掌握以下词汇。

- 超类（superclass）是继承的类。Circle的超类是Shape，Shape的超类是NSObject。
- 父类（parent class）是超类的另一种表达方式。例如，Shape是Rectangle的父类。
- 子类（subclass）是执行继承的类。Circle是Shape的子类，而Shape又是NSObject的子类。
- 孩子类（child class）是子类的另一种表达方式。Circle是Shape的孩子类。你可以随意选择是用子类/超类还是父类/孩子类。实际上你可能偶尔会同时遇到这两对表达方式。在本书中，我们将统一使用超类和子类，也许是因为我们有点书呆子气，对“亲子”不感冒吧。
- 如果想改变方法的实现，需要重写（override）继承的方法。Circle类具有自己的draw方法，因此我们说它重写了draw方法。代码运行时，Objective-C会确保调用的是重写过的方法。

4.3 继承的工作机制

我们对Shapes-Object程序做了较大调整，提取了Circle和Rectangle的重复代码并将其放入Shape类中。绝妙的是程序的其他部分不需要修改依然可以正常运行。main()函数中创建和初始化所有不同形状的代码都没有改变，并且drawShapes()函数也完全相同，而程序运行产生的结果还是一样的。

```
drawing a circle at (0 0 10 30) in red
drawing a rect at (30 40 50 60) in green
drawing an egg at (15 19 37 29) in blue
drawing a triangle at (47 32 80 50) in red
```

在此你会发现OOP另一个强大的地方：你可以对一个程序做一些重大改变，如果你非常仔细，改变后程序仍然能正常运行。当然，你也可以在过程式编程中执行类似操作，不过使用OOP成功的几率通常会更高一些。

4.3.1　方法调度

对象在收到消息时，如何知道要执行哪个方法呢？假如我们已经将setFillColor:的代码移出了Circle类和Rectangle类，当向Circle的对象发送setFillColor:方法时，Shape类的代码如何响应呢？秘密在于：当代码发送消息时，Objective-C的方法调度机制将在当前的类中搜索相应的方法。如果无法在接收消息的对象的类文件中找到相应的方法，它就会在该对象的超类中进行查找。

图4-3展示了在还没有引入Shape父类版本的程序中，方法调度机制是如何通过代码向Circle类的对象发送setFillColor:消息的。对于类似[shape setFillColor: kRedColor]的代码，Objective-C的方法调度机制首先会寻找接收消息的对象，在本例中是Circle类的对象。该对象拥有一个指向Circle类的指针，Circle类也有一个指向其相应代码的指针。调度程序通过这些指针来查找正确的代码。

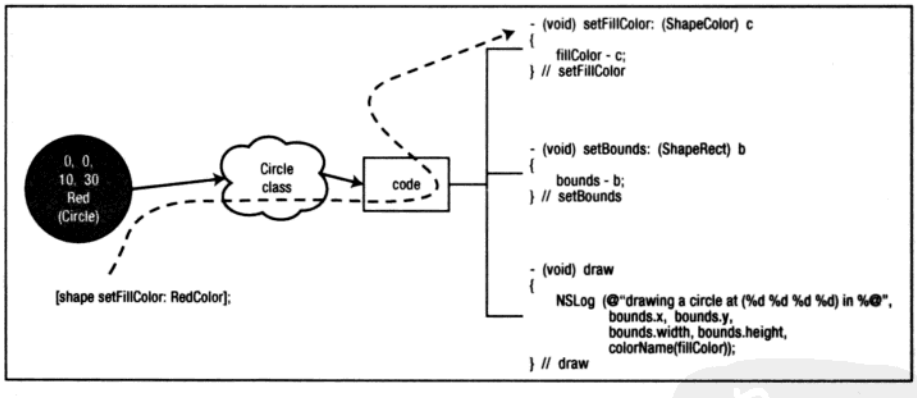


图4-3　不支持继承程序中的方法调度

再来看看图4-4，它展示了新增加了继承特性的架构。在该代码中，Circle类拥有一个指向其超类Shape的引用指针。消息传递时，Objective-C的方法调度机制会使用该信息来找到正确的方法实现。

图4-5展示了在支持继承的程序中方法调度机制的工作流程。当向Circle类的对象发送setFillColor:消息时，调度程序首先询问Circle类中的代码能否响应setFillColor:消息。在本例中，调度程序会发现Circle类中没有为setFillColor:定义方法，所以答案是不能。因此接下来它将在超类Shape中查找相应的方法实现。调度程序会溯流而上进入Shape类中，找到setFillColor:定义，之后运行这段代码。

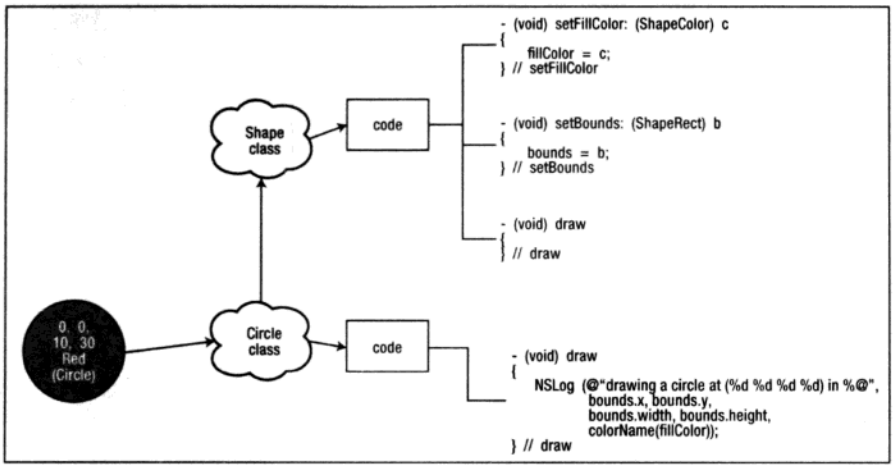


图4-4 继承和类代码

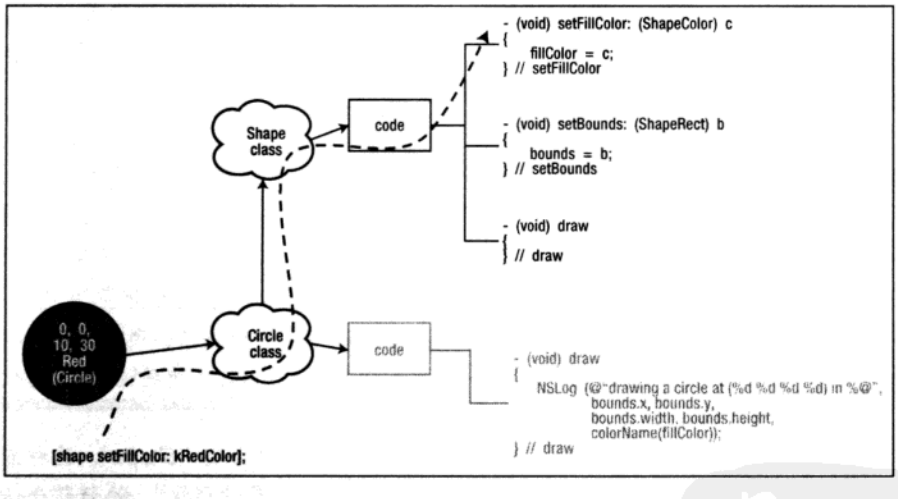


图4-5 支持继承程序中的方法调度

这种操作就像是在说“我在这里没有找到它，因此我将在它的超类中继续找”，必要时它将会在继承链的每一个类中重复地执行此操作。假如某个方法在Circle类和Shape类中都没有找到，调度程序会继续在NSObject类中寻找，因为它是继承链中的下一个超类。如果在最顶层的NSObject类中也没有找到该方法，则会出现一个运行时错误，同时还会出现一个编译时（compile-time）警告信息。

4.3.2 实例变量

之前我们讨论了方法是如何响应调用信息的。下面将介绍Objective-C如何访问实例变量，以及Circle类的draw方法如何找到Shape类中声明的bounds和fillColor实例变量。

在创建一个新类时，其对象首先会从它的超类继承实例变量，然后根据自身情况添加自己的实例变量。为了理解实例变量的继承机制，我们创建一个新的形状（圆角矩形）来添加一个新的实例变量。这个新类RoundedRectangle需要一个变量来记录绘制拐角时的半径。这个类的接口定义如下所示。

```
@interface RoundedRectangle : Shape
{
@private
 int radius;
}
@end // RoundedRectangle
```

图4-6展示了RoundedRectangle对象的内存布局。NSObject类声明了一个名为isa的实例变量，该变量保存一个指向对象当前类的指针。接下来是由Shape类声明的两个实例变量fillColor和bounds。最后是由RoundedRectangle声明的实例变量radius。

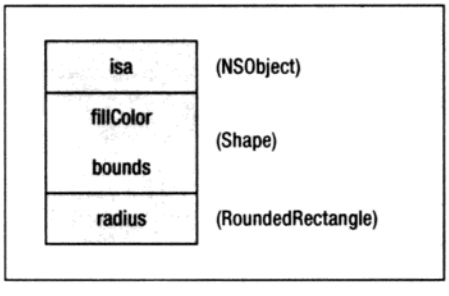


图4-6　对象中实例变量的布局

> **说明**　因为继承在子类和超类之间建立了一种“is a”（是一个）的关系，所以NSObject的实例变量叫做isa。即Rectangle是一个Shape，Circle是一个Shape。使用Shape类的代码也可以使用Rectangle类或Circle类。
>
> 使用更具体种类的对象（Rectangle或Circle）来代替一般类型（Shape），这种能力被称为多态性（polymorphism），该词源于希腊语，意思是“多种形状”，恰好符合这个程序的特点。

记住，每个方法调用都获得了一个名为self的隐藏参数，它是一个指向接收消息的对象的指针。这些方法通过self参数来寻找它们需要用到的实例变量。

图4-7展示了指向一个RoundedRectangle对象的self参数。self指向继承链中第一个类中的第一个实例变量。对RoundedRectangle类而言，它的继承链从NSObject类开始，然后是Shape类，最后以RondedRectangle类结束，因此self指向的第一个实例变量是isa。因为Objective-C编译器已经看到了所有这些类的@interface声明，所以它知道对象中实例变量的布局。通过这些重要的信息，编译器可以产生用来查找实例变量的代码。

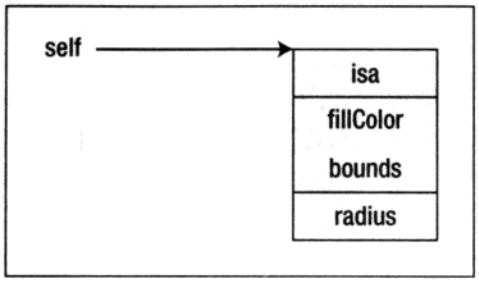


图4-7 指向圆角矩形对象的self参数

小心易碎

编译器使用“基地址加偏移”的机制实现奇妙的功能。有了对象基地址，即第一个实例变量的首个字节在内存中的位置，再在该地址上加上偏移地址，编译器就可以查找其他实例变量的位置。

例如，如果圆角矩形对象的基地址是0x1000，则isa实例变量的地址是0x1000+0，即位于0x1000位置。isa的值占4个字节，因此下一个实例变量fillColor的起始地址位于4个偏移值之后，即位于0x1000+4位置，或写作0x1004。每个实例变量与对象的基地址都有一个偏移位置。

如果访问方法中的fillColor实例变量，编译器生成代码并得到用来存储self的位置值，然后再加上偏移值（在本例中是4），就会指向存储变量值的位置。

时间长了，这也会产生一些问题。现在在编译器生成的程序中，这些偏移位置是通过硬编码实现的。尽管苹果公司的工程师希望向NSObject中添加其他的实例变量，但他们也无法做到，因为这样做会改变所有实例变量的偏移位置。这称为**脆弱的基类问题**（fragile base class problem）。通过在Leopard系统中引入新的64位Objective-C运行时（它使用间接寻址方式确定变量的位置），苹果公司解决了这个问题。

4.4 重写方法

在你制作全新的子类时，经常会添加自己的方法。有时你会添加一个能够向类中引入特有功能的新方法，有时你会替换或改进某个超类定义的现有方法。

例如，你可以从Cocoa的NSTableView类（用来显示滚动列表以供用户选择）着手，添加一个新行为，比如使用语音合成器来朗读列表的内容。你可以添加一个名为speakRows的新方法来向语音合成器提供表格的内容。

你也可以选择不添加新特性，而是创建一个子类并通过它重写继承自超类的行为。在Shapes-Inheritance示例中，通过设置形状的填充颜色和边界区域，Shape已经实现了我们想要完成的多数工作，但它不知道，也无法知道如何进行绘制，因为它是一个通用的抽象类，每个形状的

绘制方式都不相同。因此，当需要创建Circle类时，可以使它继承自Shape类，然后再编写具体用来绘制圆形的draw方法。

创建Shape类时，我们知道它的所有子类都要用于绘制图形，但不知道它们如何实现这一功能。因此我们为Shape类提供一个draw方法，但让该方法的实现内容为空，这样每个子类都能实现各自的功能。当类（比如Circle和Rectangle）实现各自的方法时，我们就说它们重写了draw方法。

向Circle对象发送draw消息时，方法调度机制将运行重写后的方法：Circle类的draw实现。超类（如Shape）中定义的所有draw实现都会被完全忽略掉。在当前情况下这没有问题，因为Shape类的draw实现中没有任何代码，但有时你可能不想忽略掉超类中定义的方法。如果想要了解更多内容，请接着往下看。

super关键字

Objective-C提供了一种方法，让你既可以重写方法的实现，又能调用超类中的实现方式。当需要超类实现自身的功能，同时在之前或之后执行某些额外的工作时，这种机制非常有用。为了调用继承的方法在父类中的实现，需要使用super作为方法调用的目标。

举个例子，假设我们获悉某些国家忌讳红色的圆，同时又想向这些国家出售Shapes-Inheritance软件。以前我们一直绘制红色的圆，但这次要将所有的圆绘制成绿色。因为这种颜色忌讳仅限于圆，所以一种办法是修改Circle类，使所有的圆都绘制成绿色。其他用红色绘制的形状没有问题，因此不必将它们消除。但为什么不直接修改Circle类中设置填充颜色的方法呢？在这里我们完全可以做到，但在其他场合则未必，比如说有些代码你根本看不到，你自然也就没有权限去修改了。

记住，setFillColor:方法是在Shape类中定义的，因此只要在Circle类中重写setFillColor:方法就可以解决该问题。代码会检查颜色的参数，如果是红色，则改为绿色，然后使用super通知超类（Shape）将更改的颜色存储到fillColor实例变量中（本程序的完整代码位于代码资源中的04.02 Shapes-Green-Circles文件夹）。

Circle类的@interface部分不需要修改，因为我们没有添加任何新方法或实例变量。只需向@implementation部分中添加以下代码：

```
@implementation Circle
- (void)setFillColor:(ShapeColor)c
{
 if (c == kRedColor)
 {
  c = kGreenColor;
 }
 [super setFillColor: c];
} // setFillColor
// and the rest of the Circle @implementation // is unchanged
@end // Circle
```

在这个新的setFillColor:实现中，我们会检查ShapeColor类型的参数是否是红色，如果是，

就将它改成绿色，然后请求超类响应消息[super setFillColor: c]并将该颜色放入实例变量中。

Super来自哪里呢？它既不是参数也不是实例变量，而是由Objective-C编译器提供的一种神奇的功能。当你向super发送消息时，实际上是在请求Objective-C向该类的超类发送消息。如果超类中没有定义该消息，Objective-C会和平常一样继续在继承链上一级中查找。

图4-8展示了Circle类的setFillColor:方法的执行流程。Circle对象接收setFillColor:消息，方法调度机制找到了自定义的setFillColor:方法，这是由Circle类实现的。

Circle类的setFillColor:方法会检查参数是不是kRedColor，并在必要时更改颜色，通过调用[super setFillColor: c]来调用超类的方法。这个super的调用将会运行Shape类的setFillColor:方法。

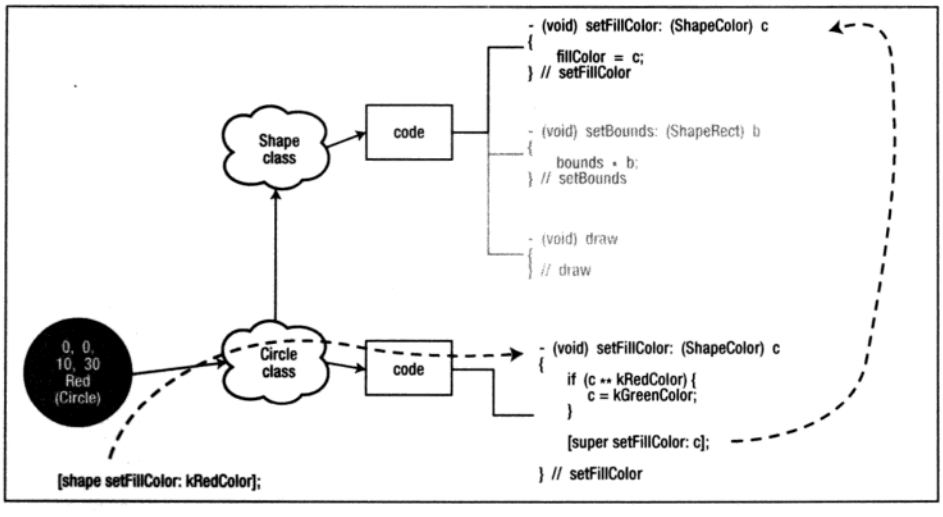


图4-8 调用超类的方法

说明 重写方法时，调用超类方法总是一个明智之举，这样可以实现更多的功能。在本例中，我们获取了Shape的源代码，因此知道所有Shape在其setFillColor:方法中只是将新颜色赋给了实例变量。但是，如果我们不熟悉Shape的内部实现细节，就不知道Shape是否还有其他功能。而且即使我们现在知道Shape的功能，但是修改或改进了类之后，可能又不知道了。调用继承的方法可以确保获得方法实现的所有特性。

4.5 小结

继承是一个非常重要的概念，OOP中很多高级技术都涉及继承。在本章中，我们学习了与继承有关的概念以及如何使用继承完善和简化Shapes-Object代码，讨论了如何通过现有类构造新

类，如何将超类的实例变量引用到子类中。

我们还学习了Objective-C的方法调度机制，同时了解了方法调度如何在继承链中查找相应的方法以响应某个特定的消息。最后，我们介绍了super关键字，并展示了如何通过它在重写方法中充分利用超类代码。

在下一章中，你将了解复合的有关内容，复合是让不同的对象协同工作的另一种方式。它可能不像继承那么神通广大，但也非常重要，我们下章再见吧。



第5章 复 合

通过上一章的学习，想必你已经对继承非常熟悉了，继承是在两个类之间建立关系的一种方式，它可以避免许多重复的代码。我们还提到建立类之间的关系也可以通过复合的方式，这就是本章的主题。使用复合可组合多个对象，让它们分工协作。在实际的程序中，你会同时用到继承和复合来创建自己的类，所以掌握这两个概念非常重要。

5.1 什么是复合

编程中的复合（composition）就好像音乐中的作曲（composition）一样：将多个组件组合在一起，配合使用，从而得到完整的作品。创作乐曲时，作曲人可能会选择低音管声部和双簧管声部组成交响乐的二声部。在软件开发中，程序员可能会用一个Pedal（脚踏板）对象和一个Tire（轮胎）对象组合出虚拟的Unicycle（独轮车）。

在Objective-C中，复合是通过包含作为实例变量的对象指针实现的。因此上述的虚拟独轮车应该拥有一个指向Pedal对象的指针和一个指向Tire对象的指针，如下所示。

```
@interface Unicycle : NSObject
{
 Pedal *pedal;
 Tire *tire;
}
@end // Unicycle
```

Pedal和Tire通过复合的方式组成了Unicycle。

> **说明** 在之前的Shapes-Object程序中，我们已经看到了某种形式的复合：Shape类使用了边界区域（一个struct结构体）和填充颜色（一个enum枚举）。但严格地说，只有对象间的组合才能叫做复合。诸如int、float、enum和struct等基本类型只被认为是对象的一部分。

Car程序

让我们暂时把Shapes程序放到一边（松了一口气吧），先来看看如何搭建汽车模型。在简化

后的模型中，1辆汽车只需要1台发动机和4个轮胎。我们不会去费力地研究真正的轮胎和发动机的物理模型，而是用两个仅包含一个方法的类来输出它们各自所代表的含义：轮胎对象会说它们是轮胎，发动机对象会说它是一台发动机。在真正的程序中，轮胎会有气压和操控能力等属性（attribute，意思同对象中的变量），发动机也会有马力和油耗等属性。这段程序的代码可以在05.01 CarParts文件夹中找到。

与Shapes-Object程序相同，CarParts程序的所有代码都包含在主文件mainCarParts.m中。代码首先以导入Foundation框架的头文件为开始：

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

然后是Tire类，里面只有一个description方法：

```
@interface Tire : NSObject
@end // Tire

@implementation Tire
- (NSString *) description {
  return (@"I am a tire. I last a while.");
} // description
@end // Tire
```

> **说明**　如果在类中没有包含任何实例变量，便可以省略掉接口定义中的花括号。

Tire类中唯一的方法是description，而且并没有在接口中声明。那么它是从哪儿来的呢？如果接口中并没有包含它，我们又怎么能知道可以在Tire类里调用description方法呢？这时就要靠Cocoa。

5.2　自定义 NSLog()

记住，NSLog()可以使用%@格式说明符来输出对象。NSLog()处理%@说明符时，会询问参数列表中相应的对象以得到这个对象的描述。从技术上来讲，也就是NSLog()给这个对象发送了description消息，然后对象的description方法生成一个NSString并将其返回。NSLog()就会在输出结果中包含这个字符串。在类中提供description方法就可以自定义NSLog()会如何输出对象。

在自定义的description方法中，你可以选择返回一个字面值NSString，如@"I am a cheese Danish Object"（“我是丹麦乳酪蛋糕”对象），也可以构造一个用来描述该对象各类信息的字符串，比如丹麦乳酪蛋糕的脂肪含量和卡路里值。在Cocoa中，NSArray类管理的是对象的集合，它的description方法提供了数组自身的信息，例如数组中对象的个数和每个对象所包含的描述。当然，对象的描述是通过向数组中的对象分别发送description消息来获得的。

让我们回到CarParts程序来看看Engine类。与Tire一样，它也包含了一个description方法。在真实的程序中，Engine类会包含start（启动）、accelerate（加速）等方法和RPM（转数）等实例变量。不过在这里，我们只是举个简单的例子来了解复合是如何工作的，所以Engine类只有一个description方法。

```
@interface Engine : NSObject
@end // Engine
@implementation Engine
- (NSString *) description
{
   return (@"I am an engine. Vrooom!");
} // description
@end // Engine
```

程序的最后一部分是Car本身，它拥有一个engine对象和一个由4个tire对象组成的数组（这个数组和C语言中数组的概念类似）。它通过复合的方式来组装自己。Car同时还有一个print方法，该方法使用NSLog()来输出轮胎和发动机的描述。

```
@interface Car : NSObject
{
  Engine *engine;
  Tire *tires[4];
}
- (void) print;
@end // Car
```

因为engine和tires是Car类的实例变量，所以它们是复合的。你可以说汽车是由4个轮胎和1台发动机组成的，但是通常人们不会这么说，因此你也可以说汽车有4个轮胎和1台发动机。

每一个Car对象都会为指向engine和tires的指针分配内存，但是真正包含在Car中的并不是engine和tires变量，只是内存中存在的其他对象的引用指针。为新建的Car对象分配内存时，这些指针将被初始化为nil（零值），也就是说这辆汽车现在既没有发动机也没有轮胎，你可以将它想象成还在流水线上组装的汽车框架。

下面让我们来看看Car类的实现。首先是一个初始化实例变量的init方法。该方法为我们的汽车创建了用来装配的1个engine变量和4个tire变量。使用new创建新对象时，系统其实在后台执行了两个步骤：第一步，为对象分配内存，即对象获得一个用来存放实例变量的内存块；第二步，自动调用init方法，使该对象进入可用状态。

```
@implementation Car
- (id) init
{
   if (self = [super init]) {
    engine = [Engine new];
    tires[0] = [Tire new];
    tires[1] = [Tire new];
    tires[2] = [Tire new];
    tires[3] = [Tire new];
   }
  return (self);
} // init
```

Car类的init方法创建了4个新轮胎并赋值给tires数组，还创建了一台发动机并赋值给engine实例变量。

接下来就是Car类的print方法：

```
- (void) print
{
   NSLog (@"%@", engine);
   NSLog (@"%@", tires[0]);
   NSLog (@"%@", tires[1]);
   NSLog (@"%@", tires[2]);
   NSLog (@"%@", tires[3]);
} // print
@end // Car
```

关于if语句　在init方法中，下面这行代码看起来有些奇怪。

```
if (self = [super init]) {
```

下面我们来解释这行代码的意思。为了让超类（在这里是NSObject）将所有需要的初始化工作一次性完成，你需要调用[super init]。init方法返回的值（id类型数据，即泛型对象指针）就是被初始化的对象。

将[super init]返回的结果赋给self是Objective-C的惯例。这么做是为了防止超类在初始化过程中返回的对象与一开始创建的不一致。在后面讲述init方法的章节里我们将深入研究，现在你跳过这行代码，继续学习下面的内容吧。

print方法通过NSLog()输出实例变量。记住，%@只是调用每个对象的description方法并显示结果。在真正的程序里，你可以用engine和tires变量计算出汽车的抓地能力。

CarParts.m的最后一部分是main()函数，也是程序的驱动力（抱歉现在才提到）。Main()函数创建了一辆新车（新对象car），并告诉它输出自身的信息，然后退出程序。

```
int main (int argc, const char * argv[])
{
  Car *car;
  car = [Car new];
  [car print];
  return (0);
} // main
```

生成并运行CarParts程序，你应该会看到与下面内容类似的输出。

```
I am an engine. Vrooom!
I am a tire. I last a while.
I am a tire. I last a while.
I am a tire. I last a while.
I am a tire. I last a while.
```

这辆车并不能为你赢得什么大奖，但是它真的运行了！

5.3 存取方法

编程人员很少会对自己编写的程序感到满意，因为软件开发是永无止境的。程序里总是有新的bug要去修正，总是有一些功能需要添加，总是有一些办法能让程序的规模更大，机制更强壮，运行速度更快。所以，CarParts程序尚不完美也不足为奇。我们可以使用存取方法来改进它，使它的代码更灵活。这个新版本的代码可以在05.02 CarParts-Accessors文件夹中找到。

经验丰富的编程人员看到Car类的init方法可能会问："为什么汽车要自己创建轮胎和发动机呢？"如果用户能为汽车定做不同类型的轮胎（例如冬季时选用雪地防滑轮胎）或发动机（例如用燃油喷射发动机取代化油器发动机），那么这个程序就会更加完善了。

如果用户可以为汽车选择专属的轮胎和发动机，就再好不过了。这样用户就可以自己搭配汽车部件，定制属于自己的汽车。

我们可以添加存取方法来实现上述想法。*存取*（accessor）方法是用来读取或改变某个对象属性的方法。例如，Shapes-Object中的setFillColor:就是一个存取方法。如果添加一个新方法去改变Car对象中的engine对象变量，那它就是一个存取方法。因为它为对象中的变量赋值，所以这类存取方法被称为setter方法。你也许听说过mutator方法，它是用来更改对象状态的。

你也许已经猜到了，另一种存取方法就是getter方法。getter方法为代码提供了通过对象自身访问对象属性的方式。在赛车游戏中，物理逻辑引擎可能会想要读取汽车轮胎的属性，以此来判断赛车以当前的速度行驶是否会在湿滑的道路上打滑。

> **说明** 如果要对其他对象中的属性进行操作，应该尽量使用对象提供的存取方法，绝对不能直接改变对象里面的值。例如，main()函数不应直接访问Car类的engine实例变量（通过car->engine的方法）来改变engine的属性，而应使用setter方法进行更改。
>
> 存储方法是程序间接工作的另一个例子。使用存取方法间接地访问car对象中的engine，可以让car的实现更为灵活。

下面为Car添加一些setter和getter方法，这样它就有选用轮胎和发动机的自主权了。下面是Car类的新接口，新添加的代码以粗体表示。

```
@interface Car : NSObject
{
  Engine *engine;
  Tire *tires[4];
}
- (Engine *) engine;
- (void) setEngine: (Engine *) newEngine;
- (Tire *) tireAtIndex: (int) index;
- (void) setTire: (Tire *) tire atIndex: (int) index;
- (void) print;
@end // Car
```

代码中的实例变量并没有变化，但是新增了两对方法：engine和setEngine:用来处理发动机的属性，tireAtIndex:和setTire:atIndex:用来处理轮胎的属性。存取方法总是成对出现的，一个用来设置属性的值，另一个用来读取属性的值。有时只有一个getter方法（用于只读属性，例如磁盘上文件的大小）或者只有一个setter方法（例如设置密码）也是合理的。但通常情况下，我们都会同时编写setter和getter方法。

对于存取方法的命名，Cocoa有自己的惯例。在为自己的类编写存取方法时，应当遵守这些惯例，这样你和其他人读代码时才不会感到困惑。

setter方法根据它所更改的属性的名称来命名，并加上前缀set。下面是几个setter方法的名称：setEngine:、setStringValue:、setFont:、setFillColor:和setTextLineHeight:。

getter方法则是以其返回的属性名称命名。所以上面的setter方法所对应的getter方法应该是engine、stringValue、font、fillColor和textLineHeight。不要将get用作getter方法的前缀。例如，方法getStringValue和getFont就违反了命名惯例。有些语言（如Java）有不同的命名惯例，它们用get做存取方法的前缀。但是如果编写Cocoa程序，请不要这么做。

> 说明　get这个词在Cocoa中有着特殊的含义。如果get出现在Cocoa的方法名称中，就意味着这个方法会将你传递的参数作为指针来返回数值。例如，NSData（Cocoa中的类，它可以存储任意序列的字节）中有一个getBytes:方法，它的参数就是用来存储字节的内存缓冲区的地址。而NSBezierPath（用于绘制贝塞尔曲线）中的getLineDash:count:phase:方法则有3个指针型参数：指向存储虚线样式的浮点型数组的指针，指向存储虚线样式中元素个数的整数型数据的指针，以及指向虚线起始点的浮点型数据的指针。
>
> 如果你在存取方法的名称中使用了get，那么有经验的Cocoa编程人员就会习惯性地将指针当做参数传入这个方法，当他们发现这不过是一个简单的存取方法时就会感到困惑。最好不要让其他编程人员被你的代码搞得一头雾水。

5.3.1　设置engine属性的存取方法

第一对存取方法用来访问发动机的属性：

```
- (Engine *) engine;
- (void) setEngine: (Engine *) newEngine;
```

在代码中调用Car对象的engine方法可以访问engine变量，调用setEngine:方法可以更改发动机的属性。下面是这两个方法的实现代码。

```
- (Engine *) engine
{
   return (engine);
} // engine
- (void) setEngine: (Engine *) newEngine
{
```

```
    engine = newEngine;
} // setEngine
```

getter方法engine返回实例变量engine的当前值。记住，在Objective-C中所有对象间的交互都是通过指针实现的，所以方法engine返回的是一个指针，指向Car中的发动机对象。

同样，setter方法setEngine:将实例变量engine的值赋为参数所指向的值。实际上被复制的并不是engine变量，而是指向engine的指针值。换一种方式说，就是在调用了对象Car中的setEngine:方法后，依然只存在一个发动机，而不是两个。

> **说明** 为了信息的完整性，我们需要说明，在内存管理和对象所有权方面，Engine的getter方法和setter方法还存在着问题。但是现在就把内存管理和对象生命周期管理的问题摆出来，肯定会让你困惑和沮丧，所以我们把如何确准无误地编写存取方法放到第8章再讲。

想要在代码中实际运用这些存取方法，可编写如下代码：

```
Engine *engine = [Engine new];
[car setEngine: engine];
NSLog (@"the car's engine is %@", [car engine]);
```

5.3.2 设置tires属性的存取方法

tires的存取方法稍微复杂一点：

```
- (void) setTire: (Tire *) tire atIndex: (int) index;
- (Tire *) tireAtIndex: (int) index;
```

由于汽车的4个轮胎都有自己不同的位置（汽车车体的4个角落各有一个轮胎），所以Car对象中包含一个轮胎的数组。在这里我们需要用索引存取器而不能直接访问tires数组。为汽车配置轮胎时，不仅需要知道是哪个轮胎，还要清楚每个轮胎在汽车上的位置。同样，当访问汽车上的某个轮胎时，访问的也是这个轮胎的具体位置。

下面是相关存取方法的实现代码。

```
- (void) setTire: (Tire *) tire atIndex: (int) index
{
  if (index < 0 || index > 3) {
   NSLog (@"bad index (%d) in setTire:atIndex:",index);
   exit (1);
  }
  tires[index] = tire;
} // setTire:atIndex:
- (Tire *) tireAtIndex: (int) index
{
  if (index < 0 || index > 3) {
   NSLog (@"bad index (%d) in "tireAtIndex:", index);
   exit (1);
  }
  return (tires[index]);
} // tireAtIndex:
```

tire存取方法中使用了通用代码来检查tires实例变量的数组索引，以保证它是有效数值。如果数组索引超出了0到3的范围，那么程序就会输出错误信息并退出。该代码就是所谓的**防御式编程**（defensive programming），这是种很好的编程思想。防御式编程能够在开发早期发现错误，比如tires数组的索引错误。

由于tires是C语言风格的数组，而且在访问这个数组时编译器无法对索引进行错误检查，因此我们必须自己检查数组索引值是否有效。像tires[-5]和tires[23]这样的写法编译器也可以编译通过，而此数组显然只有4个元素，所以使用-5或23这样的索引值便会访问到内存中的随机值，由此产生的bug会导致程序崩溃。

索引值检查完毕后，新的tire变量将会被放入tires数组中适当的位置。使用了上述存取方法的代码如下所示。

```
Tire *tire = [Tire new];
[car setTire: tire atIndex: 2];
NSLog (@"tire number two is %@", [car tireAtIndex: 2]);
```

5.3.3　Car类代码的其他变化

在CarParts-Accessors程序完成之前，代码中还有几个遗留的细节需要整理。

首先是Car类的init方法。由于Car现在已经有了访问engine和tires变量的方法，所以init方法就不需要再创建这两个变量了，而是由创建汽车的代码来负责配置发动机和轮胎。事实上我们完全可以剔除init方法，因为已经不需要在Car中做这些工作了。新车的车主会得到一辆没有轮胎和发动机的汽车，不过装配是轻而易举的事（有时软件中的生活比现实生活要容易得多啊）。

因为Car类不再通过自身的方法进行部件装配，所以必须要更新main()函数来创建它们。将main()函数改为如下形式。

```
int main (int argc, const char * argv[]) {
  Car *car = [Car new];
  Engine *engine = [Engine new];
  [car setEngine: engine];
  for (int i = 0; i < 4; i++) {
   Tire *tire = [Tire new];
   [car setTire: tire atIndex: i];
  }
  [car print];
  return (0);
} // main
```

与前一版本的main()函数相同，在这里main()也创建了一辆新车，然后为其创建并配置了一个新的发动机。接下来是一个循环了4次的for循环，每次都创建一个轮胎并安装在汽车上。最后输出这辆汽车的详细信息并退出程序。

从用户的角度来看，程序的运行结果并没有任何改变。

```
I am an engine. Vrooom!
I am a tire. I last a while.
I am a tire. I last a while.
I am a tire. I last a while.
I am a tire. I last a while.
```

与Shapes-Object程序一样，我们重构了该程序，改进了内部架构，但是并没有影响它的外部行为。

5.4 扩展 CarParts 程序

既然Car类已经有了存取方法，就应该充分利用它。我们不会照搬当前的发动机和轮胎，而是对这些部件做一些改变。用集成方式来创建新的发动机和轮胎，然后使用Car类的存取方法（复合方式）给汽车配置新的部件。该程序的代码可以在05.03 CarParts-2文件夹中找到。

首先创建一个新型的发动机Slant6（如果喜欢V8或其他型号的发动机，也可以用它们来代替Slant6）。

```
@interface Slant6 : Engine
@end // Slant6
@implementation Slant6
- (NSString *) description
{
  return (@"I am a slant- 6. VROOOM!");
} // description
@end // Slant6
```

Slant6是一种发动机，因此它可以是Engine的子类。要记住，继承可以让我们在需要超类（Engine）的地方使用子类（Slant6）。在Car类中，setEngine:方法需要的是Engine型的参数，所以我们可以放心地传递Slant6型的参数。

在Slant6类中，description方法被重写，用来输出新信息。由于Slant6并没有调用超类中的description方法（即它没有使用[super descritption]），所以新信息完全替代了Slant6继承自Engine的描述信息。

轮胎的新类AllWeatherRadial的实现步骤与创建Slant6的步骤非常相似。将其定义为现有类Tire的子类，然后重写description方法。

```
@interface AllWeatherRadial : Tire
@end // AllWeatherRadial
@implementation AllWeatherRadial
- (NSString *) description
{
  return (@"I am a tire for rain or shine.");
} // description
@end // AllWeatherRadial
```

最后调整main()函数，使用新型的发动机和轮胎（粗体字为改动的代码）。

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

```
    Car *car = [Car new];
    for (int i = 0; i < 4; i++) {
     Tire *tire = [AllWeatherRadial new];
     [car setTire: tire atIndex: i];
    }
    Engine *engine = [Slant6 new];
    [car setEngine: engine];
    [car print];
    return (0);
} // main
```

在这里我们添加了两个新类，稍微调整了两行代码，并没有修改Car类。而我们的汽车却可以在不改变Car类中任何代码的情况下，使用我们配置的任何类型的发动机和轮胎。现在，程序的输出已经有了很大的改变。

```
I am a slant- 6. VROOOM!
I am a tire for rain or shine.
I am a tire for rain or shine.
I am a tire for rain or shine.
I am a tire for rain or shine.
```

5.5　复合还是继承

CarParts-2同时用到了继承和复合，也就是我们在前一章和本章中所介绍的两个“万能”工具。那么，什么时候用继承，什么时候用复合呢？这个问题提得非常好。

继承的类之间建立的关系为“is a”（是一个）。三角形是一个形状，Slant6是一个发动机，AllWeatherRadial是一种轮胎的名字。如果可以说“X是一个Y”，那就可以使用继承。

另一方面，复合的类之间建立的关系为“has a”（有一个）。形状有一个填充颜色，汽车有一个发动机和四个轮胎。与继承不同，汽车不是一个发动机，也不是一个轮胎。如果可以说“X有一个Y”，那么就可以使用复合。

新手在进行面向对象编程时通常会犯这样的错误：对任何东西都想使用继承，例如让Car类继承Engine类。继承确实是一个有趣的新工具，但它并非适用于所有的场合。使用继承架构确实可以创建出能正常运行的程序，因为Car代码中使发动机运行的那部分代码是可以访问的，但是其他人在阅读这段代码时会不理解。汽车是一台发动机？当然不是。所以，只应在适当的时机使用继承特性。

下面举个例子来说明在设计数据结构时应该如何思考。创建新对象时，先花时间想清楚什么时候应该用继承，什么时候应该用复合。比如，设计与汽车有关的程序时，你可能会想“汽车有轮胎、发动机和变速器”，于是选择使用复合，并在Car类中声明这些东西的复合变量。

而在其他场合下，你可能会使用继承。这次假设你的程序可能会涉及有照车辆，即需要某种执照才能合法驾驶的车辆。汽车、摩托车和牵引式挂车设备都是有照车辆。汽车是一个有照车辆，摩托车是一个有照车辆——听起来很适合用继承。所以你可以创建一个LicensedVehicle（有照车

辆）类来存储汽车牌照的所在地和牌照号码（用复合方式），而Automobile（汽车）、MotorCycle（摩托车）等类就可以继承LicensedVehicle类了。

5.6 小结

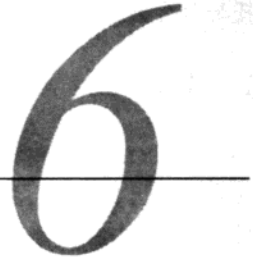
复合是OOP的基础概念，我们通过这种技巧来创建引用其他对象的对象。例如，汽车对象引用了1个发动机和4个轮胎对象。在本章关于复合的讨论中，我们介绍了存取方法，它既为外部对象提供了改变其属性的途径，同时又能保护实例变量本身。

存取方法和复合是密不可分的，因为我们通常都会为复合的对象编写存取方法。我们还学习了两种类型的存取方法：setter方法和getter方法，前者告诉对象将属性改为什么，后者要求对象提供属性的值。

本章还介绍了Cocoa存取方法的命名规则。需要特别指出的是，对于返回属性值的存取方法，名称中不能使用get这个词。

下一章我们不会谈论任何OOP理论，而是介绍如何分割不同的类，并将它们放入多个源文件中，而不是把所有代码都写到一个大文件里。

第6章 源文件组织



到目前为止，我们讨论过的所有项目都是把源代码统统放入main.m文件中。类的main()函数、@interface和@implementation部分都被塞进同一个文件里。这种结构对于小程序和简便应用来说没什么问题，但是并不适用于较大的项目。随着程序规模越来越大，文件内容会越来越多，查找信息也会越来越困难。

回想一下你的学生时代（假设你已经完成了学业）。你不会把所有的期末论文都放在同一个文档里（假设你安装了一款文字处理程序）而是把每篇论文单独存档，并起一个描述性的名字。同样，将程序的源代码拆分成多个文件并且给每个文件起一个容易记住的名称也是一个好主意。

将程序拆分为多个小文件有助于更快地找到重要的代码，而且其他人在查看项目时也能有个大致的了解。另外将代码放入多个文件还可以更容易地将有趣的类代码发给朋友：只需打包其中几个文件即可，不用打包整个项目。本章将讨论把程序代码拆分到不同文件中的方法。

6.1 拆分接口和实现

前面已提到，Objective-C类的源代码分为两部分。一部分是接口，用来展示类的构造。接口包含了使用该类所需的所有信息。编译器将@interface部分编译后，你才能使用该类的对象，调用类方法，将对象复合到其他类中，以及创建子类。

源代码的另一个组成部分是实现。@implementation部分告诉Objective-C编译器如何让该类工作。这部分代码实现了接口所声明的方法。

在类的定义中，代码很自然地被拆分为接口和实现两个部分，所以类的代码通常分别放在两个文件里。一个文件存放接口部分的代码：类的@interface指令、公共struct定义、enum常量、#defines和extern全局变量等。由于Objective-C继承了C的特点，所以上述代码通常放在头文件中。头文件名称与类名相同，只是用.h做后缀。例如，Engine类的头文件会被命名为Engine.h，而Circle类的头文件名称是Circle.h。

所有的实现内容（如类的@implementation指令、全局变量的定义、私有struct等）都被放在了与类同名但以.m为后缀的文件中（有时叫做.m文件）。上面两个类的实现文件将会被命名为Engine.m和Circle.m。

> **说明** 如果用.mm做文件扩展名，编译器就会认为你是用Objective-C++编写的代码，这样你就可以同时使用C++和Objective-C来编程了。

在Xcode中创建新文件

创建新类时，Xcode会自动生成.h和.m文件。在Xcode程序中选择File ➤ New ➤ New File后，会出现如图6-1所示的窗口，它列出了Xcode能够创建的文件类型。

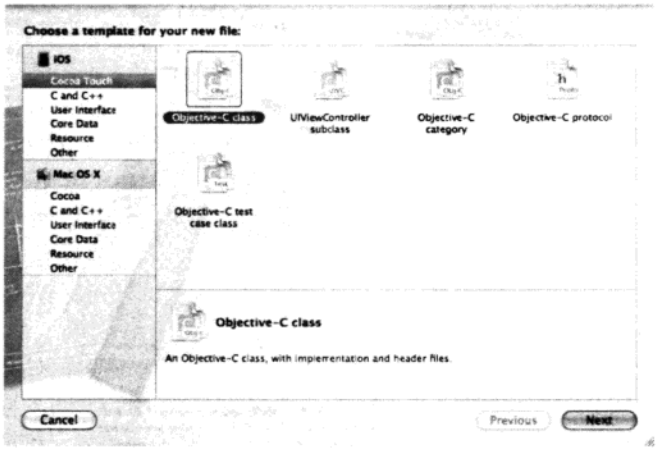


图6-1 在Xcode中创建一个新类

选中Objective-C Class，然后点击Next按钮，会弹出另一个窗口要求你填写类的名称，如图6-2所示。

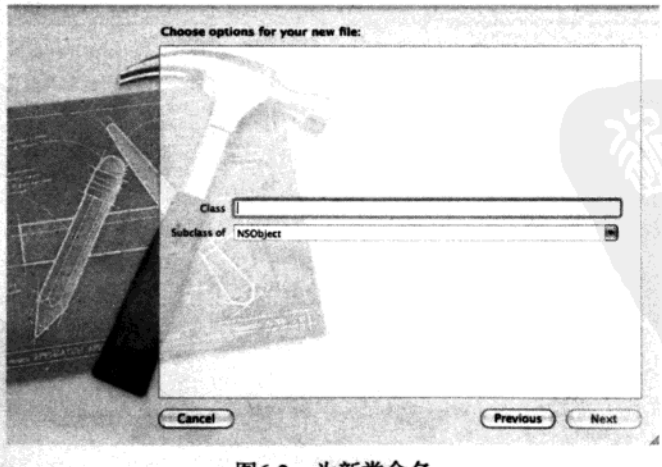


图6-2 为新类命名

你还可以选择新创建的类的父类（默认是NSObject）。每个类都必须有一个父类（除了NSObject本身，它是所有类的父类）。你可以在下拉菜单中指定这个新类的父类，如果下拉菜单中没有你想要的父类，也可以直接输入类的名称。

点击Next按钮之后，Xcode会询问你将文件存储在哪里（如图6-3所示）。最好选择当前项目中其他文件所在的目录位置。



图6-3　选择存储目录

此外你还会看到其他有趣的东西。比方说，你可以选择将新文件放入哪个群组（Group）。目前我们不会详细讨论目标（Target）这个概念，只是顺便提一下：复杂的项目可以拥有多个目标，它们源文件的配置各不相同，构建规则也不同。

新类创建完毕后，Xcode会在项目中添加相应的文件，并在项目窗口中展示出来，如图6-4所示。

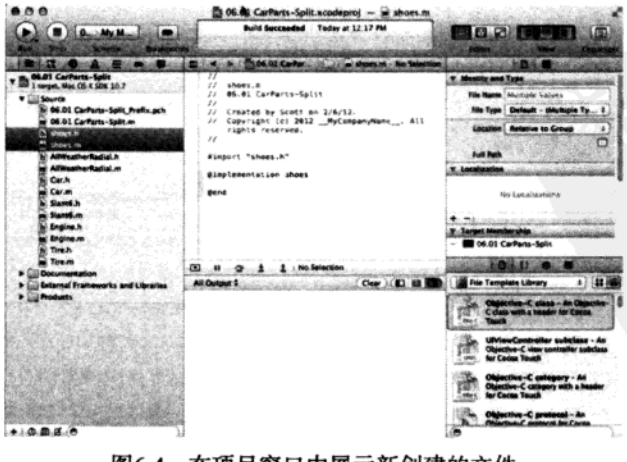


图6-4　在项目窗口中展示新创建的文件

Xcode中有一个与项目同名的群组，文件都放在群组内的文件夹中。（你可以在项目导航器中浏览项目文件的构造。）这些文件夹（在Xcode中称作群组）能够帮你组织项目中的源文件。例如，你可以创建一个用来存放用户界面类的群组，再建一个用来存放数据处理类的群组，这样你的项目将更易于浏览。在设置群组时，Xcode并不会在硬盘上移动文件或者创建目录。群组关系仅仅是由Xcode负责管理的一项奇妙的功能。当然如果你愿意的话，可以设置群组指向文件系统中某个特定的目录，Xcode会帮你将新建的文件放入该目录中。

只要创建了文件，就可以在列表中点击它们进行编辑。Xcode会自动写入一些有用的标准模板代码，它们通常都是这些文件所需要的，比如`#import <Cocoa/Cocoa.h>`以及没有内容的`@interface`和`@implementation`部分，需要你来填充。

> **说明** 到目前为止，我们在程序里使用的都是`#import <Foundation/Foundation.h>`，因为这些程序都只用到了Cocoa中的这一部分功能。但是将它替换成`#import<Cocoa/Cocoa.h>`也是没有问题的。这条语句既导入了Foundation框架的头文件，也导入了其他一些文件。

6.2 拆分 Car 程序

CarParts-Split程序可以在06.01 CarParts-Split文件夹中找到，它已经将CarParts-Split.m中的所有类都移植到每个类自己的文件中了。它们由头文件（.h）和实现文件（.m）组成。下面看看如何自己创建这样的项目吧。首先要创建两个继承自`NSObject`的类：`Tire`和`Engine`。方法是点击File菜单下的New File选项，然后选择添加Objective-C Class，并在接下来的对话框中输入它的名称Tire。使用同样的方法创建`Engine`的类文件。在图6-5中的项目文件列表中可以找到新添加的4个文件。

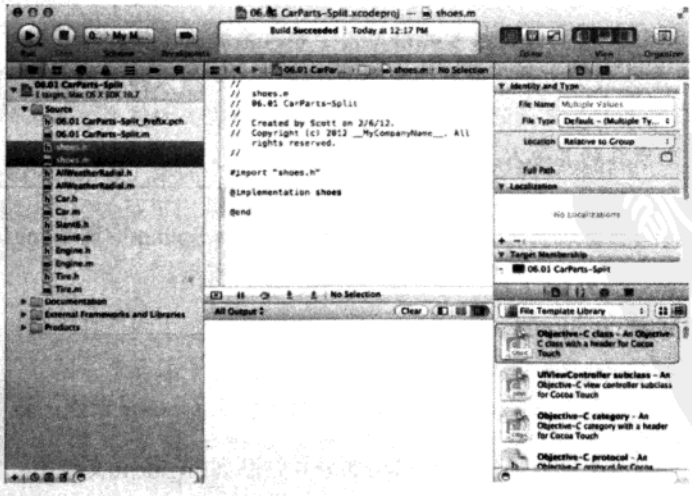


图6-5 添加到项目中的`Tire`和`Engine`类文件

现在，从CarParts-Split.m文件中剪切Tire类的@interface部分，并粘贴到Tire.h中，文件应如下所示。

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

@interface Tire : NSObject
@end // Tire
```

接下来，从CarParts-Split.m中剪切Tire类的@implementation部分并粘贴到Tire.m中，你需要在文件首行加上语句#import "Tire.h"。文件应如下所示。

```
#import "Tire.h"

@implementation Tire

- (NSString *) description
{
  return (@"I am a tire. I last a while");
} // description

@end // Tire
```

文件中的第一个#import语句很值得研究。它并没有像之前那样导入头文件Cocoa.h或者Foundation.h，而是导入了该类的头文件。这其实是标准的过程，在你以后所创建的项目里基本都会这么写。编译器需要知道类里的实例变量配置，这样才能生成合适的代码，但是它并不知道与源文件配套的头文件也存在。所以我们要添加#import "Tire.h"语句，将此信息告知编译器。在程序编译时，如果你碰到了诸如“Cannot find interface declaration for Tire”（无法找到Tire类的接口定义）之类的错误信息，通常是因为你忘记用#import导入类的头文件了。

> **说明**　注意，导入头文件有两种方法：使用引号或者尖括号。例如，#import <Cocoa/Cocoa.h>和#import "Tire.h "。带尖括号的语句用于导入系统头文件，而带引号的语句则说明导入的是项目本地的头文件。如果你看到的头文件名是用尖括号括起来的，那么这个头文件对你的项目来说是只读的，因为它属于系统。如果头文件名前后用的是引号，那么你（或参与这个项目的其他人）便可以编辑它。

现在，重复上面的步骤来创建Engine类。从文件CarParts-Split.m中剪切Engine类的@interface部分，并粘贴到Engine.h文件中，现在的Engine.h应该如下所示。

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

@interface Engine : NSObject

@end // Engine
```

接下来，剪切Engine类的#implementation部分并粘贴到Engine.m文件中，文件Engine.m应如下所示。

```
#import "Engine.h"

@implementation Engine
- (NSString *) description
{
  return (@"I am an engine. Vrooom!");
} // description

@end // Engine
```

如果想现在就编译这个程序，那么CarParts-Split.m将会报告错误，因为没有Tire类和Engine类的声明。这个问题很好解决，只需要将下面两行代码添加到CarParts-Split.m文件的顶部，也就是放在#import <Foundation/Foundation.h>语句之后即可。

```
#import "Tire.h"
#import "Engine.h"
```

> **说明** 记住，#import语句就像C语言中预处理程序的命令#include一样。在这里本质上它只进行了剪切和粘贴操作，将Tire.h和Engine.h的文件内容粘贴到CarParts-Split.m文件中，然后才继续执行接下来的操作。

现在，你可以构建并运行CarParts-Split程序了。你会发现，该程序的输出结果与使用AllWeatherRadials和Slant6的输出结果相比，没有任何变化。

```
I am a slant-6. VROOOM!
I am a tire for rain or shine
I am a tire for rain or shine
I am a tire for rain or shine
I am a tire for rain or shine
```

6.3 使用跨文件依赖关系

依赖关系（dependency）是两个实体之间的一种关系。在编程和开发过程中，经常会出现关于依赖关系的问题。依赖关系可以存在于两个类之间，例如，Slant6类因继承关系而依赖于Engine类。如果Engine类发生了变化，例如添加了一个新的实例变量，那么就需要重新编译Slant6来适应这个变化。

依赖关系也可以存在于两个或多个文件之间。CarParts-Split.m依赖于Tire.h和Engine.h文件。如果两个文件中的任何一个发生了变化，都需要重新编译CarParts-Split.m来适应这个变化。例如，假设Tire.h文件中有一个常量kDefaultTirePressure，它的值是30psi（磅每平方英寸），而编写Tire.h文件的编程人员觉得头文件中胎压的默认值应该改成40psi，那么就需要重新编译CarParts-Split.m，用新的值40替代原有值30。

导入头文件使头文件和源文件之间建立了一种紧密的依赖关系。如果头文件有任何变化，那

么所有依赖它的文件都得重新编译。这会在需要编译的文件中引发一连串的变化。假设你编写了100个.m文件，并且所有的.m文件都导入了同一个头文件（假设是UserInterfaceConstants.h）。如果你修改了UserInterfaceConstants.h，那么所有这100个.m文件都会重新生成，即使有一堆超频超多核的Mac Pro主机群集任你使用，这也需要花费相当长的时间。

不仅如此，由于依赖关系是传递的，头文件之间也可以互相依赖，所以重新编译的问题会更加严重。例如，如果Thing1.h中导入了Thing2.h，而Thing2.h中又导入了Thing3.h，那么Thing3.h中发生任何变化，都需要重新编译那些导入了Thing1.h的文件。尽管重新编译需要花费很长的时间，但至少Xcode帮你记录了所有的依赖关系。

6.3.1　重新编译须知

好在Objective-C提供了一种方法，能够减少由依赖关系引起的重新编译带来的负面影响。导致依赖关系问题的原因是Objective-C编译器需要某些信息才能够工作。有时编译器需要知道类的全部信息，例如它的实例变量配置、它所继承的所有类等，而有的时候，编译器只需要知道类名即可，不需要了解整个定义。

例如，在对象复合后（在前一章已介绍过），复合通过指针指向对象。这之所以行得通，是因为所有Objective-C对象都使用动态分配的内存。编译器只需要知道这是一个类就可以了，然后就会知道实例变量就是指针的大小，在整个程序中都是如此。

Objective-C引入了关键字@class来告诉编译器："这是一个类，所以我只会通过指针来引用它。"这样编译器就放心了：它不必知道关于这个类的更多信息，只要了解它是通过指针来引用的即可。

在将Car类移动到属于其自身的文件时会用到@class。在Xcode中，使用和移植Tire类以及Engine类相同的方法来创建文件Car.h和Car.m，复制Car的@interface部分并粘贴到Car.h中，如下所示。

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

@interface Car : NSObject
- (void) setEngine: (Engine *) newEngine;
- (Engine *) engine;

- (void) setTire: (Tire *) tire
      atIndex: (int) index;

- (Tire *) tireAtIndex: (int) index;

- (void) print;

@end // Car
```

如果我们现在就想使用这个头文件，就会从编译器那里得到错误消息，告诉我们它不明白

Tire和Engine是什么。这条错误信息很可能会像这样：error: expected a type "Tire"，编译器这是在说“我无法理解这个”。

我们有两种方法来解决这个错误问题。第一种就是用#import语句导入Tire.h和Engine.h，这样编译器会获得关于这两个类的许多信息。

此外还有一个更好的方法。如果仔细观察Car类的接口，你会发现它只是通过指针引用了Tire和Engine。这是@class可以完成的工作。下面是加入了@class代码的Car.h文件内容。

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

@class Tire;
@class Engine;

@interface Car : NSObject

- (void) setEngine: (Engine *) newEngine;

- (Engine *) engine;

- (void) setTire: (Tire *) tire
      atIndex: (int) index;

- (Tire *) tireAtIndex: (int) index;

- (void) print;

@end // Car
```

这样就足以告知编译器处理Car类的@interface部分所需要的全部信息了。

> **说明** @class创建了一个前向引用。这是在告诉编译器：“相信我。以后你自然会知道这个类到底是什么，但是现在，你知道这些足矣。”
>
> 如果有循环依赖关系，@class也很有用。即A类使用B类，B类也使用A类。如果试图通过#import语句让这两个类互相引用，那么就会出现编译错误。但是如果在A.h文件中使用@class B，在B.h中使用@class A，那么这两个类就可以互相引用了。

6.3.2 让汽车跑一会儿

上一节完成了Car类的头文件部分，但是Car.m需要更多关于Tire和Engine的信息，编译器需要知道Tire类和Engine类继承自什么类，才能查找对象中有没有方法可以响应发送过来的消息。为此，我们在Car.m文件中导入Tire.h和Engine.h，还需要从CarParts-Split.m文件中剪切出Car类的@implementation部分并粘贴到Car.m文件中。现在Car.m文件应如下所示。

```
#import "Car.h"
#import "Tire.h"
```

```
#import "Engine.h"
@implementation Car {
    Tire *tires[4];
    Engine *engine;
}

- (void) setEngine: (Engine *) newEngine
{
  engine = newEngine;
} // setEngine
- (Engine *) engine
{
  return (engine);
} // engine

- (void) setTire: (Tire *) tire
      atIndex: (int) index
{
 if (index < 0 || index > 3) {
     NSLog (@"bad index (%d) in setTire:atIndex:", index);
     exit (1);
}

tires[index] = tire;

} // setTire:atIndex:

- (Tire *) tireAtIndex: (int) index
{
 if (index < 0 || index > 3) {
     NSLog (@"bad index (%d) in setTire:atIndex:", index);
     exit (1);
}

return (tires[index]);

} // tireAtIndex:

- (void) print
{
   NSLog (@"%@", tires[0]);
   NSLog (@"%@", tires[1]);
   NSLog (@"%@", tires[2]);
   NSLog (@"%@", tires[3]);

   NSLog (@"%@", engine);
} // print

@end // Car
```

现在你可以再次构建并运行这个程序了，程序的输出结果与原来一模一样。是的，我们又对这个程序进行了重构。我们改进了程序的内部结构，但是并没有影响它的外部行为。

6.3.3 导入和继承

我们还需要从CarParts-Split.m文件中拆分出两个类：Slant6和AllWeatherRadial。这有些困难，因为它们继承自我们自己创建的类：Slant6继承自Engine类，而AllWeatherRadial继承自Tire类。由于它们是继承自其他类而不是通过指针指向其他类的，所以不能在头文件中使用@class语句。我们只能在Slant6.h文件中使用#import "Engine.h"语句，在AllWeatherRadial.h文件中使用#import "Tire.h"语句。

那么，为什么不能在这里使用@class语句呢？因为编译器需要先知道所有关于超类的信息才能成功地为其子类编译@interface部分。它需要了解超类中实例变量的配置信息（数据类型、大小和排序）。回想一下，在子类中添加实例变量时，它们会被附加在超类实例变量的后面。然后编译器就利用这些信息计算在内存的什么位置能找到这些实例变量，每个方法都通过自身的self隐藏指针进行寻找。为了能够准确地计算出实例变量的位置，编译器必须先了解该类的所有内容。

接下来要生成的类是Slant6。在Xcode中创建文件Slant6.m和Slant6.h，然后从CarParts-Split.m中剪切出Slant6的@interface部分。如果操作正确，Slant6.h应该如下所示。

```
#import "Engine.h"

@interface Slant6 : Engine

@end // Slant6
```

这个文件只导入了Engine.h而没有导入<Cocoa/Cocoa.h>，这是为什么呢？我们知道，Engine.h中已经导入了<Cocoa/Cocoa.h>，所以不需要再导入一遍了。然而，在该文件里加上#import <Cocoa/Cocoa.h>语句也是允许的，因为#import命令非常智能，不会重复导入已导入的文件。

Slant6.m只是把CarParts-Split.m中的@implementation部分剪切并粘贴出来，再加上常用的#import命令导入头文件Slant6.h。

```
#import "Slant6.h"

@implementation Slant6

- (NSString *) description
{
   return (@"I am a slant-6. VROOOM!");
} // description

@end // Slant6
```

重复执行上面的步骤，将AllWeatherRadial移植到它自己的两个文件中。毫无疑问，现在对于这种操作你已经驾轻就熟了。下面是AllWeatherRadial.h。

```
#import "Tire.h"

@interface AllWeatherRadial : Tire
```

```
@end // AllWeatherRadial
```

接下来是AllWeatherRadial.m：

```
#import "AllWeatherRadial.h"

@implementation AllWeatherRadial

- (NSString *) description
{
   return (@"I am a tire for rain or shine");
} // description

@end // AllWeatherRadial
```

可怜的CarParts-Split.m文件只剩下一副躯壳了。现在，它只有一组#import命令和一个孤零零的函数，如下所示。

```
#import <Foundation/Foundation.h>
#import "Tire.h"
#import "Engine.h"
#import "Car.h"
#import "Slant6.h"
#import "AllWeatherRadial.h"

int main (int argc, const char * argv[])
{
   Car *car = [Car new];

   int i;
   for (i = 0; i < 4; i++) {
    Tire *tire = [AllWeatherRadial new];
    [car setTire: tire atIndex: i];
}

Engine *engine = [Slant6 new];
[car setEngine: engine];

[car print];

return (0);
} // main
```

如果现在构建并运行这个项目，会得到与拆分文件之前一样的输出结果。

6.4　小结

在本章中，我们学习了使用多个文件来组织源代码的基本技巧。通常，每个类都有两个文件：包含类@interface部分的头文件和包含@implementation部分的.m文件。类的使用者可以通过#import命令导入头文件来获得该类的功能。

在学习过程中，我们认识了文件之间的依赖关系，在这种关系中，头文件或源文件需要使用另一个头文件中的信息。文件导入过于混乱会延长编译时间，也会导致不必要的重复编译，而巧妙地使用@class指令告诉编译器“相信我，你最终肯定会了解这个名称的类”，可以减少必须导入的头文件的数量，从而可以缩短编译时间。

接下来我们将领略一些有趣的Xcode功能，下一章见。

第7章 深入了解Xcode

Mac程序员和iOS程序员大部分时间都是在Xcode内编写代码。Xcode是一个很好用的工具，有很多强大的功能，不过并不是所有的功能都易于发现。如果你打算长期使用这个强大的工具，就肯定要尽可能多地了解它。本章将介绍一些Xcode编辑器的使用技巧，这对于编写和浏览代码以及查找信息都是大有帮助的。此外，还会提到一些用Xcode调试程序的方法。

Xcode是一个庞大的应用程序，它的自定义功能非常强大，有时会让人觉得不可思议。仅是介绍Xcode就可以写一整本书（已经有人写过了），所以我们只讲解精要的部分，以便你快速上手。建议你在刚开始的时候使用Xcode的默认设置，当遇到问题时，再根据个人喜好来调整。

面对Xcode这样的大型工具，最好的学习方法就是先粗略浏览一遍文档，然后使用一段时间，接着再游览一遍文档。每浏览一遍，都会有更多了解。洗头发也是这样，洗头，冲头，多次重复，头发便会焕发光彩。

这里介绍的是Xcode 4.3.2，也就是写这本书时的最新版本。苹果公司热衷于在Xcode版本升级时添加或删除一些东西，所以如果你用的是Xcode 42那么高的版本，本章的屏幕截图恐怕早已过时了。好了，我们开始学习这些使用技巧吧。

7.1 窗口布局一览

作为一名Xcode程序员，你的舞台就是Xcode程序的主窗口。如图7-1所示。下面介绍一下这个窗口的组成部分，不然，说到后面可能就不知所云了。

- 工具栏：位于程序窗口的最顶端，上面有很多工具按钮。
- 导航器面板：位于窗口左边，通常用来显示项目中的文件列表，也可以浏览其他内容，比如符号（symbol）、搜索（Search）、问题（Issue）、调试（Debug）、断点（Breakpoints）和日志（Logs）。可以按住Command键和一个数字键（从1到7）或点击面板顶端的图标来切换视图。
- 编辑器面板：位于中间偏右的位置。你大部分时间都要在这里大展身手，写下一行行源代码，改变世界，飞黄腾达。
- 检查器面板：位于窗口右边，显示的是与上下文（当前选中的内容）有关的信息，以及修改选中项属性值的按钮。

- **调试器面板**：位于底部居中位置。调试器运行的时候，堆栈和调试器控制器会出现在这里。
- **库面板**：隐藏在窗口右下角，列有项目资源、对象、代码片段和其他在项目中可能会用到的东西。

现在你已经了解窗口的基本布局了。让我们开始学习Xcode的旅程吧。



图7-1 OS X和iOS程序员每天都要面对的窗口

7.2 改变公司名称

你可能已经注意到，新建Objective-C源文件时，Xcode会自动帮你生成注释文字。

```
//
// Calculator
// CKStoreManager.m
//
// Created by Waqar Malik on 4/15/12.
// Copyright 2012 __MyCompanyName__. All rights reserved.
//
```

Xcode在注释块中写入了文件名称、项目名称以及创建者和创建时间，包含这些信息是为了让你一眼看去便知道所查看的是哪个文件，谁创建了它，以及它的生成时间。

更改公司名称的方法很简单。在导航器面板选中项目，并确保在编译器面板的Project栏目下选中项目名称。仔细观察右边的检查器面板，在Project Document栏目下你将会看到Organization文本框。在里面输入你的公司名称。如图7-2所示。完成之后，项目便会使用你输入的公司名称来替代之前的__MyCompanyName__。

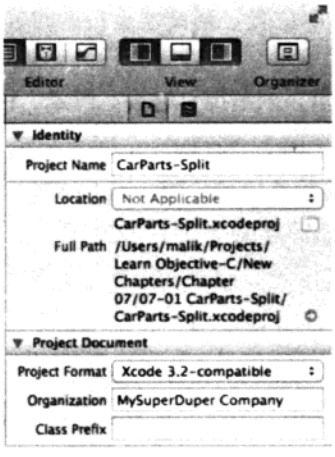


图7-2　更改公司名称

7.3　使用编辑器的技巧

Xcode提供了几种组织项目和源代码编辑器的基本方式。我们已经介绍了默认界面，它主要是用来管理实时项目和执行编码任务的一体化窗口，编辑器面板则是用来显示源文件的，内容由窗口左边导航器面板选中的源文件而定。

这里再稍微详细讲解一下你在导航器面板中所看到的内容。文件列表展示了项目中所有有用的部分：源文件、链接的框架和构建程序的Products文件夹。你还可以找到其他实用工具，比如访问源代码版本控制（方便与其他程序员协作），所有的项目符号，还有智能文件夹。

选中了某个文件后，你会在窗口顶端的工具栏下面看到文件的路径，告诉你文件在项目中的位置。你可以使用导航器面板底部的搜索框来过滤列表文件。图7-3列出了搜索名称中带字符car的文件。你可以在任意导航器视图中使用这个过滤搜索框。

浏览器列出了每个名称中带字符car的源文件。你可以在浏览器中选中它们使编辑器显示它们的内容。因为大型项目可能拥有一百多个源文件，所以如果文件特别多的话，浏览器是一种很方便的管理工具。本章后面会深入介绍源文件导航的相关话题。

编写代码时，可以隐藏浏览器，这样屏幕的可用面积会更大一些。在窗口的右上方有一组标记为View的工具栏按钮。其中有三个按钮，你可以把鼠标悬停在上面来查看其作用，不过，还是在这里跟你直说吧：左边的按钮用来隐藏或显示导航器面板，你也可以使用Command+0快捷键来调用它；中间的开关按钮负责调试器区域是否可见；右边的按钮负责检查器面板。

用独立的窗口来显示各自的源文件也是有用的，尤其是当你想比较两个不同文件的时候。在导航器面板中双击源文件就可以在新窗口中打开它。你也可以在两个不同的窗口中显示同一个文件，不过要注意有时这两个窗口会出现内容不同步的情况，因此你需要点击其中某个窗口来使它们同步。

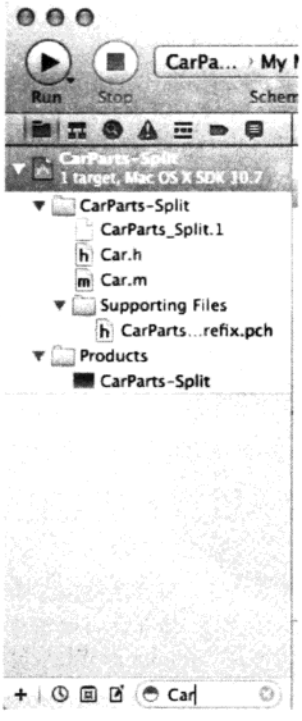


图7-3 过滤列表中的文件

你可能更倾向于使用标签（就像在Safari中那样）而不是多窗口的显示方式。Xcode已经料到了。就好像Xcode用户都了解Safari用户似的。如果想要显示标签（如图7-4所示），请选择View ➤ Show Tab Bar选项。如果想要添加标签，就点击标签栏右边的加号（+）按钮。

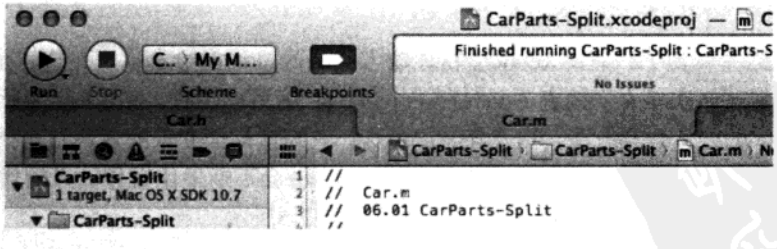


图7-4 显示标签栏

7.4 在 Xcode 的帮助下编写代码

许多程序员都夜以继日地写代码。Xcode给所有程序员提供了一些功能，使编写代码变得更容易，也更有趣。

7.4.1　首行缩进（美观排版）

你可能已经注意到了，本书所有的代码都有着整齐的缩进格式，if语句和for循环的代码都比外层代码缩进得更多。Objective-C并不要求缩进代码，但是这么做是个好习惯，因为它能够使你的代码解构更清晰。输入代码时，Xcode能自动帮你缩进。

有时，对代码进行大量的编辑会使其变得混乱不堪，Xcode可以改善这种状况。选中文本后，按住Control键点击（或直接右击）就能看到编辑器的上下文菜单，之后选择Structure ➤ Re-Indent选项，Xcode将会重新整理好一切。你可以使用快捷键Control+I来达到同样的效果。

通过Structure菜单，或按下Command+[键和Command+]键可以将选定的代码进行左移或右移。如果你只是在已有代码外添加一个if语句，那么使用它们来格式化代码会很方便。

假设现在编辑器中有下列代码：

```
Engine *engine = [Slant6 new];
[car setEngine: engine];
```

然后，你决定只有在用户需要时才会创建一台发动机。

```
if (userWantsANewEngine) {
 Engine *engine = [Slant6 new];
 [car setEngine: engine];
}
```

你可以选择中间两行代码，然后使用Command+]键，向右移动这些代码。

你可以随意调整Xcode的缩进。可能你更喜欢使用空格缩进，而不是tab制表符。可能你更愿意把花括号放在下一行而不是放在if语句的同行结尾。无论你想怎样做，都可以按照自己的统一风格来调整Xcode内的代码，主要技巧见Xcode ➤ Preferences ➤ Text Editing ➤ Indentation。这里有一个小提议：如果你想轻松快速地在程序员中展开一场热烈的网络讨论，那么从讨论代码格式的喜好开始吧。

7.4.2　代码自动完成

也许你已经注意到了，有时Xcode会在你输入代码的过程中给出建议，这就是Xcode的代码提示功能，通常叫做*代码自动完成*（code completion）。编写程序时，Xcode会为所有内容生成索引，包括项目中的变量名和方法名以及导入的框架。它知道局部变量的名称及其类型，甚至可能知道你的编码风格是好还是坏。当你输入代码时，Xcode会不断地比较你输入的代码和它生成的符号索引，如果两者匹配，Xcode就会给出建议。如图7-5所示。

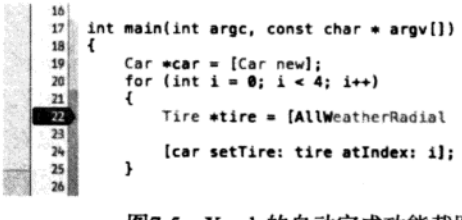


图7-5　Xcode的自动完成功能截图

上图中，我们先输入了“[All”，Xcode认为我们想要给AllWeatherRadial类发送消息（如图7-5中AllW字符之后的灰色文字）。Xcode猜得没错，所以我们可以按下tab键，接受AllWeather-Radial，以完成全文输入。

你可能会说：“啊，这也太简单了！毕竟我们只有一个以All开头的类啊！”这话没错，但哪怕有很多符合要求的选项，Xcode也会通过自动完成列表来显示它们（如图7-6所示）。如果你想关闭这个列表，只需按下esc键就可以了，再次按esc键又会重新调出自动完成列表。

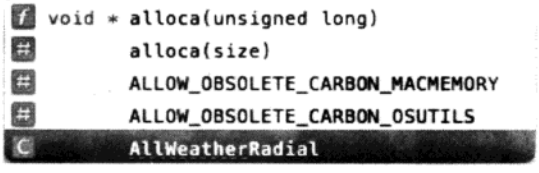


图7-6 包含All的所有关键字

可以看到有很多以All开头的符合要求的代码。Xcode发现当前项目中的AllWeatherRadial是以All开头的，于是认为它是合理的首选。名称旁边的彩色方框表示这个符号的类型：E表示枚举符号，f表示函数，#表示#define指令，m表示方法，C表示类，等等。

在列表中你可以使用Control+.（半角句号）快捷键向后翻页，或者使用Shift+Control+.快捷键向前翻页。不必担心记不住这些快捷键，本章最后提供了一个快捷键的备忘表单。

你可以把自动完成提示列表当成类的API便捷参考来使用。NSDictionary类有一个方法，可以指定一系列的参数用作生成字典的键值和对象。这个方法是dictionaryWithKeysAndObjects还是dictionaryWithObjectsAndKeys？谁能记得住呢？查找这个名称的一个简单方法是输入[NSDictionary的方法调用指令前端，然后输入一个空格表示类名已经输入完毕，接着按下esc键。Xcode知道你将要在这里输入一个方法名，于是显示出NSDictionary类中的所有方法。当然，方法dictionaryWithObjectsAndKeys就在其中。如图7-7所示，列表的最下方便是。

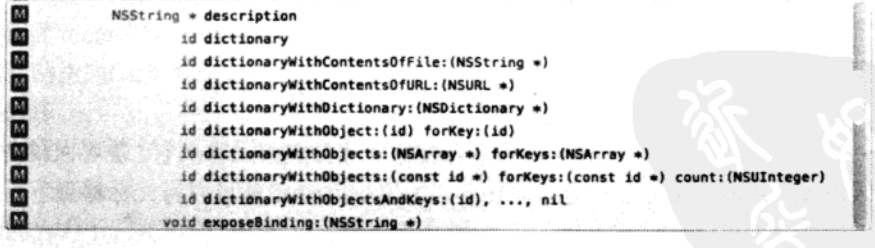


图7-7 通过代码智能感应列出的类数据

还可以更加方便。把鼠标悬停在方法名称上，便会看到窗口右边有一个问号图标，点击它就会打开一个描述方法信息的迷你帮助窗口（如图7-8所示）。

有时，使用代码完成功能时，会在提示里出现一些奇怪的小方框，如图7-9所示。这又是怎么回事？

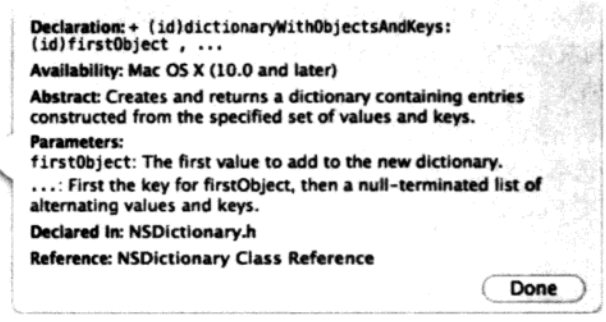


图7-8　在帮助窗口的文件名称是一个链接，点击它会打开一个新窗口

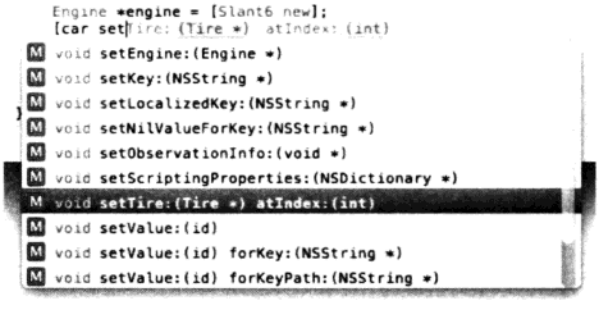


图7-9　代码自动完成列表中的占位符

注意，在这里Xcode提示的是接受两个参数的方法setTire:atIndex:。Xcode的代码提示功能不仅限于补充名称。这里显示的两个参数实际上是占位符。如果再次按tab键，这个方法的名称将会一直补充至setTire部分，如图7-10所示。

```
Engine *engine = [Slant6 new];
[car setTire: (Tire *) atIndex: (int)
```

图7-10　高亮选中占位符

第一个占位符高亮显示了，你可以输入任何数值当做实参来替换它，也可以单击第二个占位符然后再替换。甚至不用把手从键盘上移开，再按一次tab键就可以将光标移到下一个占位符的位置。

7.4.3　括号配对

输入程序代码时，也许你会发现输入某些字符（比如“)”、“]”或“}”）后屏幕会闪烁。如果发生了这种情况，就是Xcode在告诉你这个闭括号所对应的开括号在哪里，如图7-11所示。我们在第二行末尾输入了一个右括号，完成后前方相应的左括号就会以背景色短暂闪烁一下。

```
Car *car = [Car new];
  for (int i = 0; i < 4; i++)
```

图7-11 括号配对

这个功能有时被称为“括号配对”（kissing the parentheses），它能在你结束一系列复杂的分隔符（也就是括号）集时提供帮助。要保证你输入的每个闭括号都和你想关闭的开括号相互匹配。如果你弄错了，比如应该输入“)”的地方用了“]”，Xcode就会发出警报声，而且也无法显示与之匹配的开括号。

也可以双击某个分隔符，Xcode会选定它以及与它匹配的括号之间的全部代码。

7.4.4 批量编辑

有时，你可能想要将代码中的某个更改应用到其他几个地方，但是又不想自己逐个编辑。手动去做大量相似的编辑操作是件危险的事情，因为人类不擅长重复枯燥的工作。所幸，计算机很胜任这类工作。

在这方面对我们有所帮助的第一个Xcode功能并不是去操作代码，而是要建立一个安全网。选择File ➤ Create Snapshot...选项（或使用Cmmand+Cntrol+S快捷键），Xcode会记住项目当前的状态，现在你就可以放心地编辑源文件了，甚至可以随心所欲地用各种方式“玩坏”你的项目。如果你意识到自己犯了一个很严重的错误，可以通过File ➤ Restore Snapshot...选项打开快照窗口，这样就可以用前一个快照恢复项目了。在你做任何冒险的事情之前最好先创建一个快照。

> **说明** 快照文件存储在~/Library/Developer/Xcode/Snapshots/目录中。

当然，Xcode也有自己的查找和替换功能。Edit ➤ Find子菜单包含几个非常方便的选项。选择Find in Workspace（Cmmand+Shift+F），或者在导航器面板中选择搜索选项，就可以对整个项目中的所有文件进行内容搜索或替换了。图7-12显示的是在整个项目范围内搜索和替换窗口。

假设我们要将car替换成automobile。在搜索和替换文本框中输入这两个单词，然后点击Find按钮，可以看到指向Car类和car实例变量的引用。你可以点击Replace All按钮，在整个项目里应用这个替换操作。

不过对于这种替换任务，查找和替换的功能并不好用。如果你只是想重命名函数中的变量，那么它会做多余的操作（因为它可能会改变整个文件中的变量名称）。而如果你想重命名一个类时，它又无能为力了，尤其是无法修改源文件的名称。

Xcode有两个功能可以弥补这些不足。第一个功能我们简单地称为Edit all in Scope（在范围内编辑全部内容）。你可以选定一个符号，如局部变量或参数，然后点击它，就会看到右边出现了一个向下的箭头。点击箭头就会看到一个菜单，选择Edit all in Scope选项。之后你输入内容，就会使所有该符号出现的地方同时更新。这可不只是进行简单批量改动的快捷功能，你在实际操作时会觉得它非常酷的。

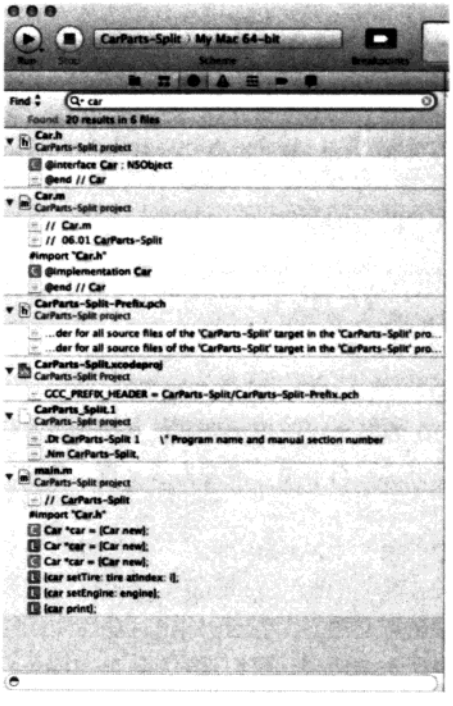


图7-12　项目范围内的搜索和替换

图7-13显示了在变量的有效范围内编辑car对象名的过程。可以看到所有的car局部变量都有一个方框围着。一旦输入Automobile，所有方框里的内容都会随之发生变化，如图7-14所示。



图7-13　开始在范围内编辑全部内容，将car替换成automobile

```
CarParts-Split › CarParts-Split › main.m
1  //
2  //  main.m
3  //  CarParts-Split
4  //
5  //  Created by Waqar Malik on 3/22/12.
6  //  Copyright (c) 2012 __MyCompanyName__. All ri
7  //
8
9  #import <Foundation/Foundation.h>
10
11 #import "Tire.h"    // don't really have to #imp
12 #import "Engine.h"  // since they're brought in
13 #import "Car.h"
14 #import "Slant6.h"
15 #import "AllWeatherRadial.h"
16
17 int main(int argc, const char * argv[])
18 {
19     Car *autom = [Car new];
20     for (int i = 0; i < 4; i++)
21     {
22         Tire *tire = [AllWeatherRadial new];
23
24         [autom setTire: tire atIndex: i];
25     }
26
27     Engine *engine = [Slant6 new];
28     [autom setEngine: engine];
29
30     [autom print];
31
32     return (0);
33 }
34
35
```

图7-14 在范围内编辑全部内容

输入完毕后，只要在源文件编辑窗口中单击其他地方，就会退出Edit all in Scope模式。

有时，当你想要执行上述更改时，却发现Edit all in Scope菜单被禁用。这是因为这个功能与Xcode中的语法着色功能密切相关，所以如果你关闭了语法着色功能或者对它改动过多，Edit all in Scope功能也许就会拒绝工作。要解决这个问题，需要回到设置菜单，对语法着色进行调整，直到它能正常工作为止——这里需要很多技巧。

还记得上一章讲过的重构吗？我们造这个词出来可不是为了炫耀自己有多聪明。Xcode中有一些内置的重构工具，其中就有一个能够让你轻松地重新命名类。它不仅能够重命名类，还能做一些诸如重命名相应源文件之类的漂亮活儿。而且如果你开发的是GUI程序，它还能够深入到nib文件进行修改操作（如果最后一句话对你来说完全是火星文，也别着急。这是一个非常棒的功能，我们会在接下来的章节中详细介绍nib文件的）。

让我们试着将所有的Car类名称替换成Automobile类吧。请在编辑器中打开Car.h文件，并把光标放在类名Car中。选择Edit ➤ Refactor ➤ Rename . . .选项，你会看到如图7-15所示的对话框，在图中已经输入了Car的替代类名Automobile。

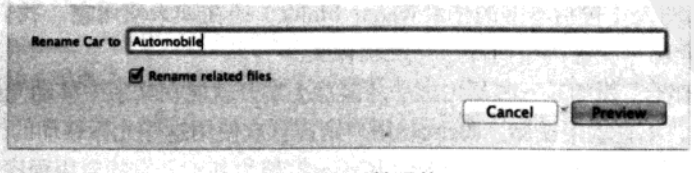


图7-15 开始重构

你点击Preview按钮之后，Xcode就会明白要做什么，并将要做的任务显示出来。如图7-16所示。

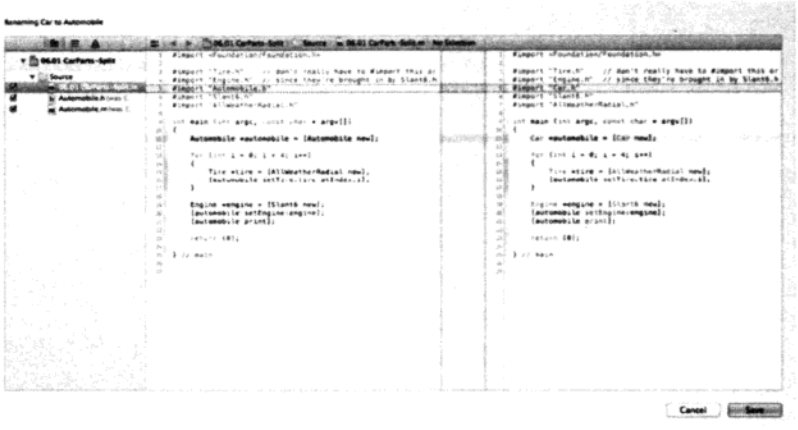


图7-16　Xcode告诉我们它将如何进行重构

可以看到Xcode会将文件Car.h和Car.m重命名为相应的Automobile文件。你可以单击一个源文件，在窗口底部的文件合并查看器中浏览Xcode将会对这个文件做哪些更改。在这个截图中，可以看到Xcode将#import中的Car替换为了Automobile，同时也替换了相应地方出现的类名。

确认这些更改没有问题之后，请点击Save按钮。如果你是第一次在这个项目中重构代码，Xcode会询问你是否要启用自动快照备份。选择yes是一个明智的决定，因为如果你做的修改过多并且不太满意，想要撤销，通过它恢复到之前的版本会非常容易。

可惜重构并不能重命名注释中的文字。所以，类里面的注释、Xcode生成的文件头注释或者任何你编写的文档注释都需要手工编辑。你可以使用查找和替换功能来简化这一过程。

7.4.5　代码导航

大多数源文件的生命周期都大同小异，创建后很快就被写入大量代码来实现各种功能，接下来进入增增改改的阶段，然后进入维护阶段。你必须先阅读大量文件才能添加或修改代码。最后，当类编写成熟后，你要在应用之前浏览其代码来了解它是如何工作的。本节主要介绍在代码的生命周期中浏览代码的不同方法。

1. emacs

emacs是一个很早就有的文本编辑器，它诞生于20世纪70年代，可以在现代的Mac操作系统上运行。有些怀旧的人（包括本书的作者Waqar Malik）还在天天使用它。我们不会过多地谈及emacs，只是简要介绍一些它的快捷键组合及其含义。

“emacs快捷键组合”指的是一些不用把手从键盘上拿开就能在文字中移动光标的快捷键。很多人喜欢使用方向键，不喜欢用鼠标，而emacs用户则更喜欢使用这些光标移动的快捷键来替代方向键。不可思议的是，这些光标快捷键也适用于以Cocoa框架为基础开发的应用程序，不仅仅是Xcode，

还有TextEdit、Safari的地址栏和文本域、Pages和Keynote文本域等。下面将这些按键列举出来。

- control-F：光标前（Forward）移（效果同右方向键）。
- control-B：光标后（Backward）退（效果同左方向键）。
- control-P：光标移动到上（Previous）一行（效果同上方向键）。
- control-N：光标移动到下（Next）一行（效果同下方向键）。
- control-A：光标移动到行首位置（效果同按住command键和左方向键）。
- control-E：光标移动到行尾（End）位置（效果同按住command键和右方向键）。
- control-T：交换（Transpose）光标两边的字符。
- control-D：删除（Delete）光标右边的字符。
- control-K：将当前行光标以后的所有字符全部删除（Kill），便于你重写行尾的代码。
- control-L：将光标置于窗口正中央。如果你找不到光标位置或者想要移动窗口使光标快速定位于正中央，这个快捷键会非常好用。

如果你记住并且掌握了这些快捷键，就可以更快地在小范围内移动光标并进行编辑操作（不只在Xcode应用程序中）。

2. 快速打开的窍门

假如你在浏览某个源文件时，看到了文件上方的#import语句，如果能够迅速打开这个头文件而不用鼠标点来点去的，是不是很方便？事实上这是可以的！只要选定文件名（甚至可以不用选择.h），然后选择File ➤ Open Quickly（快速打开）选项，Xcode就会为你打开这个头文件。

如果你没有选择任何文本，那么选择Open Quickly选项将会打开一个对话框，这是另一种文件查找方法，你也可以使用快捷键Command+Shift+D来执行Open Quickly命令。这个对话框是个非常简单的窗口，只有一个搜索域和一个表格，但却是非常快捷的项目内容搜索方式。在搜索框内输入tire可查找与轮胎有关的文件，如图7-17所示。你也可以输入其他文字，比如输入“NSArray”来查找NSArray类的头文件。

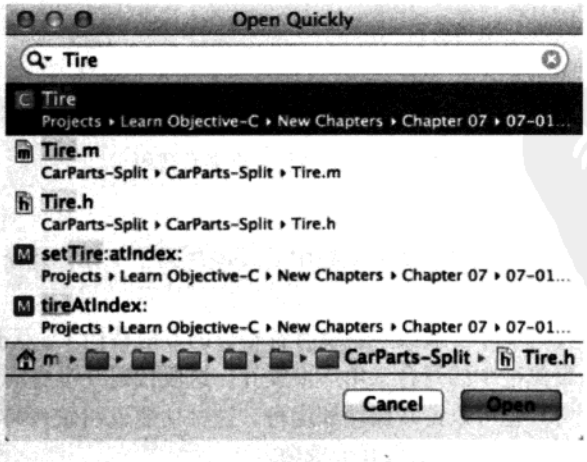


图7-17 Open Quickly对话框

如果你的显示器很大，就可以在主编辑器旁边使用辅助窗口（View ➤ Assistant Editor ➤ Show Assistant Editor）。默认情况下，如果一个面板显示的是头文件，那么另一个面板显示的就一定是实现文件，不过你也可以根据自己的喜好更改这个设定。点击工具栏的Counterparts菜单可以看到其他选项。

7.4.6　集中精力

你可能已经注意到紧挨着源代码左侧的两个空栏。左边较宽的那栏叫做*边栏*（gutter），我们会在之后讨论调试的时候谈到它。较窄的一栏叫做*聚焦栏*（focus ribbon），顾名思义，聚焦栏能够让你将注意力集中在代码中某个部分。

注意聚焦栏的灰度：代码嵌套得越深，它旁边的聚焦栏中的灰色也会越深。这种颜色编码能够使代码的复杂程序一目了然。你可以在聚焦栏的不同灰色区域悬停鼠标来高亮显示相应的代码片段，如图7-18所示。

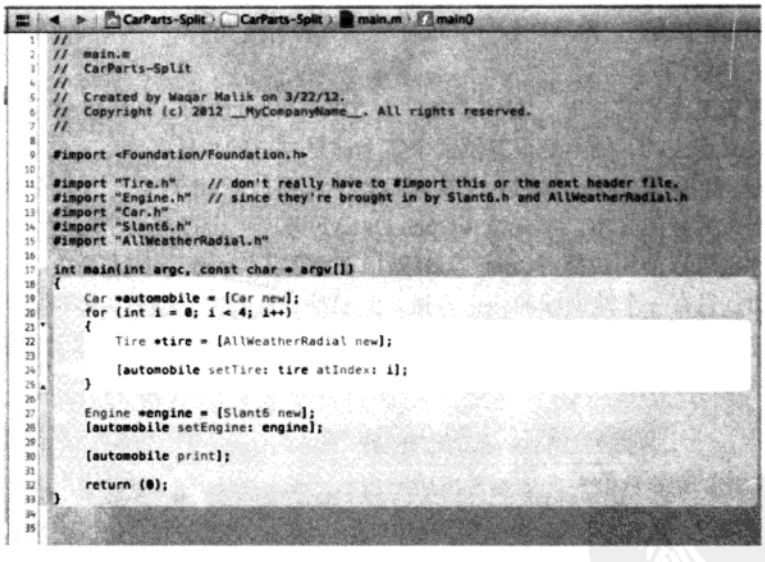


图7-18　用聚焦栏高亮显示代码

你也可以点击聚焦栏来折叠相应的代码片段。假设你确定图7-18中所示的if语句和for循环都是正确的，不想再浏览了，想专心关注函数中的其他代码了（如标题所说的，集中精力），那么点击if语句的左侧，该语句的代码内容就会折叠起来，如图7-19所示。

可以看到if语句的代码本体已经被一个带有省略号的方框替代了。双击这个方框可以将代码展开，也可以通过点击聚焦栏中的开合三角形来展开。这段代码并没有消失，只是隐藏了起来，所以就算不展开，这个文件也可以正常编译和运行。这种功能也叫做*代码折叠*（code folding）。点击Editor ➤ Code Folding菜单可以查看其他选项。

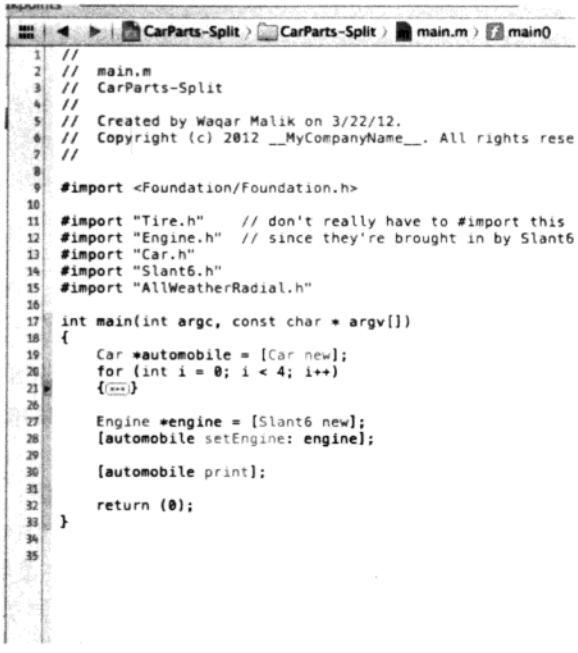


图7-19 折叠代码后的外观。看起来好像是放了一段代码的引用

7.4.7 使用导航条

在代码编辑器的顶部有一个如图7-20所示的小控件条，也就是导航条（navigation bar）。这里面的很多控件可以让你在项目中的源文件之间快速切换。

导航条最左边是一个菜单按钮，可以快速打开编辑器最近访问的历史文件或执行其他高级的操作。接着是后退和前进按钮，可以打开曾经编辑过的文件，工作方式与Safari浏览器的后退和前进按钮一样。这两个按钮后面是显示当前文件（Car.m）在项目中位置的路径（不仅是目录的路径）。路径中的每一项都是一个按钮，可以点击其中任意一个并在弹出的菜单中进行文件导航。

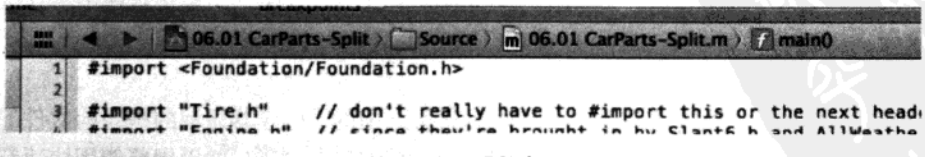


图7-20 导航条

最后一项是功能菜单，它显示的是光标当前位于方法-setTire:atIndex:中，点击菜单便可以看到文件中所有的符号，如图7-21所示。

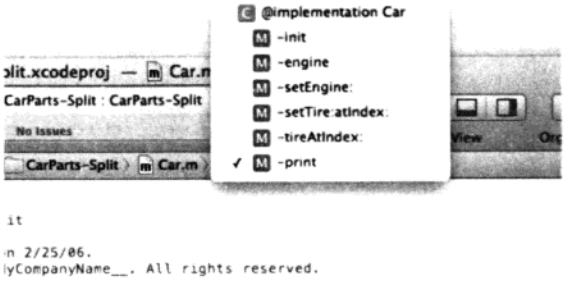


图7-21　文件中的符号

-setTire:atIndex:被突出显示，这是因为光标在这个方法里面。你可以在它的上下方看到其他方法，它们按照在文件中的顺序排列。按住Command键并点击功能菜单可以将这些方法按字母顺序排列。

除了方法名之外，你还可以向菜单中加入其他内容。有两种方法：一种是用#pragma mark whatever，后面的whatever处可以填写任何文字，它会出现在功能菜单中。这种方法便于添加可读的标记，供其他程序员查看和使用。而#pragma mark -（减号）则会在菜单中插入分割线。Xcode也会在注释里查看那些以诸如“MARK:”（与#pragma mark的功能相同）、“TODO:”、“FIXME:”、“!!!:”和“???:”之类符号开头的文本，并且将这些文本放入功能菜单中。这些都是程序员所做的记号：“在程序发布之前最好先看看这些标记。”

> **说明**　“pragma”源自希腊语，意思是“行动”。#pragma指令将Objective-C常规代码之外的信息或说明传递给编译器和代码编辑器。通常，pragma是被忽略的，但它在一些软件开发工具中可能有其他的含义。如果某个工具并不知道pragma是什么，并不会生成警告或错误信息，而是很聪明地忽略它。

7.4.8　获取信息

在代码和Cocoa头文件之间自由切换浏览是很好的功能，也很方便，但有时你也需要从代码之外获得信息。幸好，Xcode有一个存储文档和参考材料的“宝库”。（至于宝库中有什么宝物从本书可就无从知晓啦。）

1. 获得帮助

检查器的顶端有两个图标。目前第一个图标是被选中的，检查器显示的是当前文件的各种属性。另一个图标可以在检查器中启用快速帮助（Quick Help）功能。如果想要使用快速帮助，在代码中点击任意位置并观察检查器中出现的内容。如果再点击源代码中的其他位置，快速帮助面板便会更新。

举个例子，比方说你的光标位于文字NSString中，快速面板便会如图7-22所示。

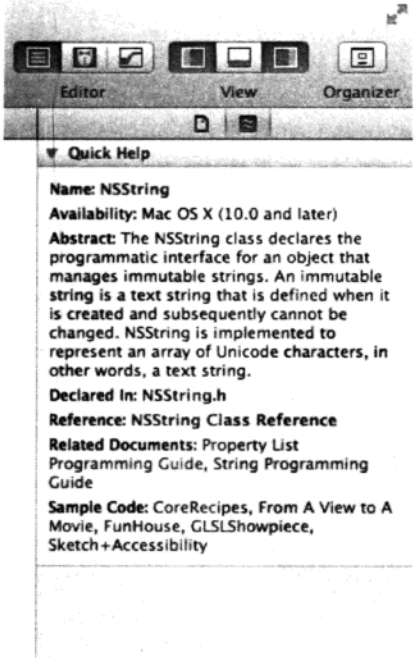


图7-22 快速帮助

图中有很多有帮助的信息。点击前两项便能在文档窗口中浏览NSString类的参考文档，而NSString.h那项会在编辑器内打开它的头文件。

Abstract那部分的文字对当前类进行了描述。如果你的光标位于某个方法调用中，这部分就会描述该方法和相关的调用方法。这里还有很多指向了包含NSString的高级文档和代码案例的链接。蓝色的文字都是链接，点击它们可以获取更多信息。你也可以按住Option键点击某符号来从快速帮助面板中获取更多信息。如果你这样做，便会看到如图7-23所示的内容。如果想要更多的信息，请点击右上角的书本图标以打开这个符号的相关文档。

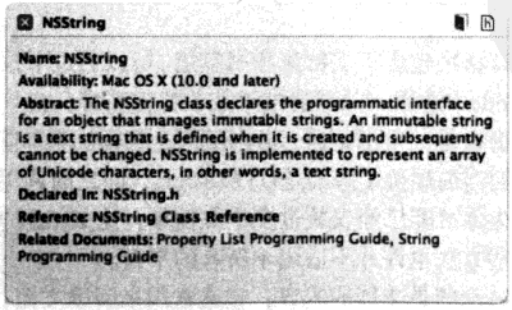


图7-23 点击NSString链接会获取更多信息

PDG

2. 文档管理程序

如果你想直接访问苹果公司的官方API文档，最快的方法是按住option键并双击某个符号，这是查找该符号相关文档的快捷方式。

假设你有一行类似于[someString UTF8String]的代码，它将someString（属于NSString型数据）转换成Unicode编码的C型字符串。如果按住option键并双击UTF8String，将会打开文档浏览器并搜索UTF8String，如图7-24所示。

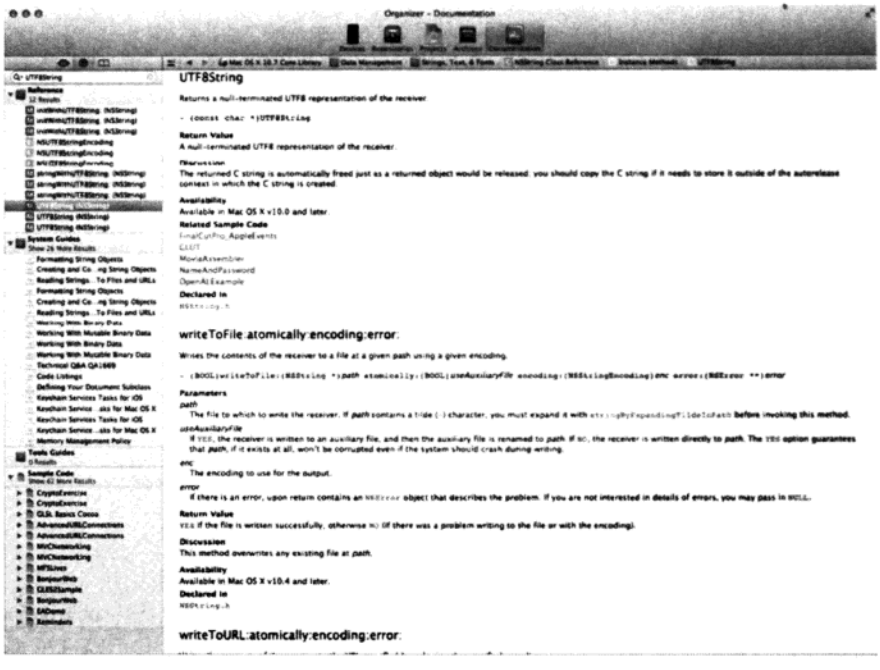


图7-24　Xcode文档浏览窗口

该窗口包含了大量的信息。左上角是一个搜索框，里面预先填写了UTF8String。工具栏下方的按钮用来筛选想要搜索的文档集。你可以在所有文档里进行搜索，也可以只搜索OS X或iOS的文档集。

这些按钮下方的左侧窗格中包含了文档集和书签组。可以通过下拉菜单在特定文档的某段中添加书签。文档书签在Xcode中通用，并不局限于某个特定项目。

位于窗口顶端左侧的窗格是一个浏览器，显示了所有匹配结果。图7-24中显示了符合条件的3个方法名和1个常量。我们当前想要了解的是UTF8String，即之前调用的方法。你会发现里面有些文字出现多次，这是因为你可能已经安装过多个文档集，比如iOS、OS X等，所以请选择你想要了解的平台。记住，有些方法也许并不适用于所有的平台。

旁边的另一个浏览器显示的是文档的内容，它通常用来阅读文档。也可以在另一个Xcode窗口或某个网页浏览器中打开文档。

7.5 调试

bug无处不在。程序中有错误是难免的，特别是在当你刚开始使用新的平台和新的语言时。发现问题后，深吸一口气，喝一口你最爱的饮料，然后系统地查找到底哪里做错了。这种查找程序错误的过程叫做调试（debugging）。

7.5.1 暴力测试

最简单的一种调试方式是使用“暴力”，即暴力测试（caveman debugging）。在程序中写入输出语句（如NSLog）来输出程序的控制流程和一些数据值。你也许一直在这么做，只是不知道这种调试方法就叫暴力测试。可能有些人瞧不起暴力测试，但是它确实是一种很有效的调试工具，特别是在你刚刚开始学习一个新系统的时候。所以不要理会这些唱反调的人。

7.5.2 Xcode的调试器

除了上面讲的这些功能，Xcode还有一个调试器。调试器（debugger）是位于你编写的程序和操作系统之间的程序，它能够中断程序，这样你就可以检查程序的数据，甚至修改程序。完成这些后，就可以恢复程序并查看运行结果。你也可以单步执行代码，减缓程序运行的速度，细致地查看代码会对数据进行哪些改动。

Xcode还有一个提供大量概述信息的调试窗口，以及一个可以直接向调试器发送调试命令的调试控制台（console）。

> **说明** Xcode中有两种调试器可供选择：GDB和LLDB。GDB是GNU计划的一部分，可以在各种平台上兼容，享誉已久。而LLDB是当前热门的新竞争者，是LLVM计划多个Xcode工具中的一员。目前两者之间的不同主要是精细度和内联性。

下面介绍在Xcode的文本编辑器中进行调试的方法。如果你想知道更多的内容，建议你去了解其他的调试模式。

7.5.3 精巧的调试符号

打算调试程序时，需要确保你正在使用Debug构建配置。你可以在Xcode工具栏中Edit Scheme的下拉菜单中找到它。构建配置通知编译器发出额外的调试符号，调试器通过这些符号可以知道它们在程序中的位置。

7.5.4 开始调试

开始使用调试器时，使用GUI程序比我们一直使用的命令行程序要容易一些。GUI程序会停

止运行并等待用户操作，所以你有足够的时间去找到调试器按钮，中断程序的执行并开始检查程序。如果使用命令行程序，运行结果会一闪即过，来不及做多少调试工作。所以先在**main**函数中设置一个断点，我们会使用上一章提到的07.01 CarParts-split程序。

打开文件CarParts-Split.m，单击边栏，也就是之前看到过的聚焦栏左边的宽条。你应该能看到一个蓝色箭头状的物体，那就是新断点，如图7-25所示。你可以将断点拖出边栏删除，也可以点击断点来禁用它。



图7-25　设置一个断点

现在点击运行按钮来运行程序，程序应该如图7-26所示的那样停在断点处。注意指向代码行的矩形方框，它就像是商业街地图上标明“您在这里”的标志。



图7-26　您在这里

当程序停止之后，你会在调试器窗格上方看到一个新的控件条，如图7-27所示。



图7-27 调试器控件

哇，又多了几个按钮。它们是用来干什么的？从左边开始，第一个按钮用来打开或关闭调试器窗格。

后面的四个按钮控制着程序接下来的行为。第一个按钮看起来像CD播放器的播放按钮（如果你不知道什么是CD播放器，就去问你的父母吧），它是继续按钮，点击之后程序会继续运行，直到遇到下一个断点、结束或崩溃。

第二个控件看起来像一个人在跃过一个点，它是跳过按钮。点击它会执行一行代码，然后程序的控制权又交还给你。如果你单击了3次跳过按钮，那么“您在这里”的箭头将会移到-setTire:atIndex调用那一行，如图7-28所示。

```
int main(int argc, const char * argv[])
{
    Car *car = [Car new];
    for (int i = 0; i < 4; i++)
    {
        Tire *tire = [AllWeatherRadial new];

        [car setTire: tire atIndex: i];
    }

    Engine *engine = [Slant6 new];
    [car setEngine: engine];

    [car print];

    return (0);
}
```

图7-28 单步执行三次之后

第三个按钮，向下指向一个点的箭头，是跳入按钮。如果程序里有当前光标所在函数或方法的源代码，那么Xcode将会跳入那个方法，显示其代码，并且将“您在这里”的箭头设置在代码的起始位置，如图7-29所示。

```
- (void) setEngine: (Engine *) newEngine
{
    engine = newEngine;
} // setEngine


- (void) setTire: (Tire *) tire
         atIndex: (int) index
{
    if (index < 0 || index > 3) {
        NSLog (@"bad index (%d) in setTire:atIndex:",
               index);
        exit (1);
    }

    tires[index] = tire;

} // setTire:atIndex:
```

图7-29 跳入一个方法之后

第四个按钮是跳出按钮，单击它会终止当前运行的函数，并且程序会停在调用函数那行的下一行代码，控制权又回到你手中。如果你正在跟着我同步操作，那么先不要点击这个按钮，因为我们还要看一看这个方法中的一些数据值。

后面一个控件是一个可以让你选择观察哪条线程的下拉菜单。你现在不需要理会线程编程，所以可以忽略线程的选择。

在所有按钮的右边是调用栈（call stack），它是当前处于活动状态的函数的集合。如果A调用B，B调用C，那么C就位于栈的底部，接下来是B和A。如果你现在打开调用栈菜单，那么显示出来的将会是`-[Car setTire:atIndex:]`，下面是`main`函数。这就说明`main`函数调用了`-setTire:atIndex:`。在更为复杂的程序中，这种调用栈也称为栈跟踪（stack trace），它可以包含许多方法函数。在调试的过程中，你能了解的最有用的信息是："这段代码究竟是怎么被调用的？"通过查看调用栈，可以看到当前的这种（混乱）状态是因为谁调用了谁而造成的。

7.5.5　检查程序

现在程序停止执行了，接下来该做些什么呢？通常，当你在程序的某个部分设置断点或者单步执行时，就说明你想了解程序状态（program state）——变量的值。

Xcode有数据提示（detatips）功能，类似于告诉你鼠标指针所在的按钮有何用途的工具提示。在Xcode编辑器中，你可以在变量或方法参数上悬停鼠标，Xcode会弹出一个小窗口来显示它的数值，如图7-30所示。

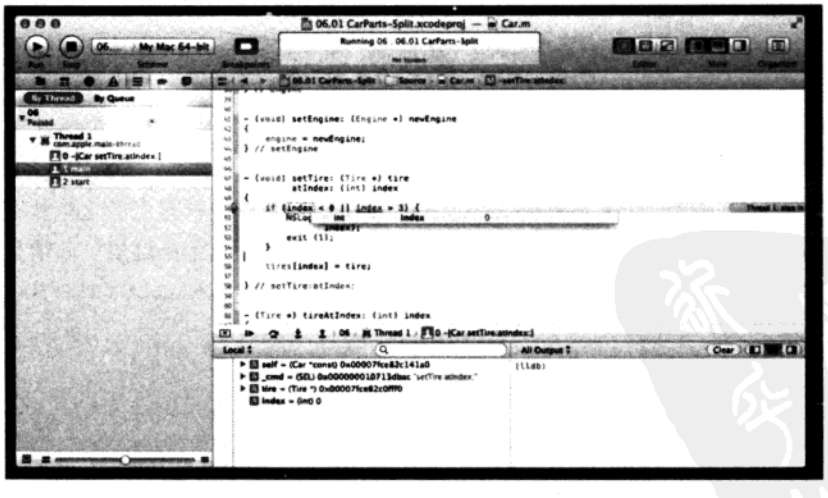


图7-30　Xcode数据提示

图7-30中，我们将鼠标指针悬停在`index`变量上。弹出的数据提示窗口中显示的数值为0，单击0并且输入一个新数值就可以改变`index`的值。例如，你可以输入37，然后通过命令行运行两步程序，你会看到程序因为索引超出范围而退出。

程序还在循环中时，将鼠标停在tires处，你将会看到数组的数据提示。将鼠标下移到箭头处，直到箭头展开，就会显示出4个tire变量的信息。接下来，移动鼠标停在第一个tire处，Xcode将会显示出这个tire的全部信息。我们的tire中不包含实例变量，因此，也就没有任何信息。如果类中含有实例变量，它们将会显示出来并且可以编辑。图7-31展示的就是鼠标悬停的结果。

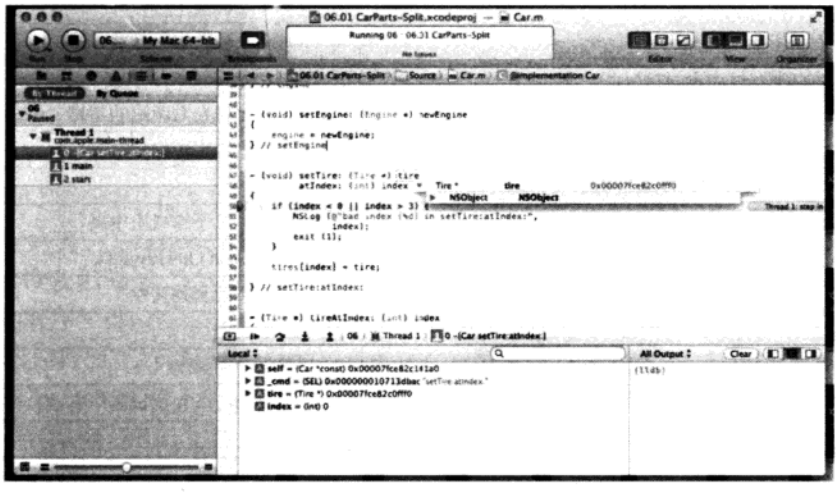


图7-31 获取程序中的数据值

以上就是对Xcode调试器的快速讲解。花一些时间好好学习这些知识，你应该可以应付调试中遇到的任何问题了。祝你调试愉快！

7.6 备忘表

本章提到了很多键盘快捷键。我已经按照承诺把它们做成了一个备忘表，如表7-1所示。如果你阅读的是纸质书，在将这本书送给别人之前，你可以把本页撕下来，只要你觉得这样并无不妥。

表7-1 Xcode键盘快捷键

按 键	描 述
Command+[	左移代码块
Command+]	右移代码块
Tab键	接受代码自动完成提示
Esc键	显示代码提示菜单
Control+.（半角句号）	循环浏览代码提示
Shift+Control+.（半角句号）	反向循环浏览代码提示
Command+Control+S	创建快照

（续）

按　键	描　述
Control+F	前移光标
Control+B	后移光标
Control+P	移动光标到上一行
Control+N	移动光标到下一行
Control+A	移动光标到本行行首
Control+E	移动光标到本行行尾
Control+T	交换光标左右两边的字符
Control+D	删除光标右边的字符
Control+K	删除本行
Control+L	将光标置于窗口正中央
Command+Shift+O	显示Open Quickly窗口
Command+Control+向上方向键	打开相配套的文件
按住Option键双击鼠标	搜索文档
Command+Y	激活/禁用断点
Command+Control+Y	继续运行（在调试器中有效）
F6	跳过
F7	跳入
F8	跳出

7.7　小结

本章介绍的知识非常多，但并没怎么谈及Objective-C语言。这是为什么呢？磨刀不误砍柴工嘛！木工不光要了解木头，还得非常了解自己使用的工具。同样，Objective-C程序员需要了解的也不仅仅是这个语言。要在Xcode中熟练地编写、浏览和调试代码，你得少花时间与语言环境“斗争”，要把更多的时间放在有趣的编程上。

接下来我们将会详细介绍Cocoa中的一些类。这会非常有趣的！

第 8 章

Foundation Kit介绍

Objective-C是一门非常精巧实用的语言，目前我们还没有研究完它提供的全部功能。不过现在，我们先探索另一个方向，快速了解一下Cocoa中的Foundation框架。尽管Foundation框架只是Cocoa的一部分，没有内置于Objective-C的语言中，但是它依然十分重要，在本书中有必要对它进行讲解。

通过第2章我们知道，Cocoa实际上是由许多个不同的框架组成的，其中最常用于桌面端（OS X）应用程序的是Foundation和Application Kit。它包含了所有的用户界面（UI）对象和高级类，在第16章你将接触AppKit（时髦的人习惯这么称呼它）。

如果打算开发iOS平台上的应用程序，那么你将会用到User Interface Kit（UIKit）框架。我们将在第15章详细讲解UIKit。UIKit可与iOS平台的AppKit框架相提并论，它包含了iOS应用程序所需要的所有界面对象。

8.1 稳固的 Foundation

Foundation，顾名思义，就是两类UI框架的基础，因为它不包含UI对象，所以它的对象可以在iOS或OS X应用程序中兼容。

Foundation框架中有很多有用的、面向数据的简单类和数据类型，我们将会讨论其中的一部分，例如`NSString`、`NSArray`、`NSEnumerator`和`NSNumber`。Foundation框架拥有100多个类，所有的类都可以在Xcode安装文档中找到。你可以在Xcode的Organizer窗口选择Documentation选项卡，来查看这些文档。

Foundation框架是以另一个框架CoreFoundation为基础创建的。CoreFoundation框架是用纯C语言写的，你愿意的话也可以使用它，不过本书不予讨论。在介绍名称相似的框架时不要混淆。如果函数或变量的名称以CF开头，那么它们就是CoreFoundation框架中的。其中很多都可以在Foundation框架中找到相对应的，它们之间的转换也非常简单。

8.2 使用项目样本代码

在继续学习之前，有一点需要注意，即在本章及以后章节将提到的项目中，我们仍会创建基于Foundation模板的项目，不过，以下默认的样本代码都会保留。

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

```
int main (int argc, const char * argv[])
{
 @autoreleasepool
 {
  // insert code here…
  NSLog(@"Hello, World!");
 } return 0;
}
```

来看一下这段代码。main()函数后面首先是关键字@autoreleasepool，并且所有代码都写在了关键字和return语句之间的两个大括号中。这只是Cocoa内存管理的冰山一角，下一章会详细介绍。所以现在，你只要一笑而过就好了，与@autoreleasepool相关的内容先暂时忽略掉。当然不使用这个关键字倒不至于引发问题，只不过在程序运行时会冒出一些奇怪的信息。

8.3　一些有用的数据类型

在深入研究Cocoa的类之前，先看看Cocoa为我们提供的一些结构体（struct）。

8.3.1　范围

第一个结构体是NSRange。

```
typedef struct _NSRange
{
 unsigned int location;
 unsigned int length;
} NSRange;
```

这个结构体用来表示相关事物的范围，通常是字符串里的字符范围或者数组里的元素范围。Location字段存放该范围的起始位置，而length字段则是该范围内所含元素的个数。在字符串“Objetive-C is a cool language”中，单词cool可以用location为17，length为4的范围来表示。location还可以用NSNotFound这个值来表示没有范围，比如变量没有初始化。

创建新的NSRange有三种方式。第一种，直接给字段赋值：

```
NSRange range;
range.location = 17;
range.length = 4;
```

第二种，应用C语言的聚合结构赋值机制（是不是听起来很耳熟）：

```
NSRange range = { 17, 4 };
```

第三种方式是Cocoa提供的一个快捷函数NSMakeRange()：

```
NSRange range = NSMakeRange (17, 4);
```

使用NSMakeRange()的好处是你可以在任何能够使用函数的地方使用它，例如在方法调用中将其作为参数进行传递。

```
[anObject flarbulateWithRange:NSMakeRange (13, 15)];
```

8.3.2 几何数据类型

之后你会经常看到用来处理集合图形的数据类型，它们的名称都带有CG前缀，如CGPoint和CGSize。这些类型是由Core Graphics框架提供的，用来进行2D渲染。Core Graphics是用C语言所写的，因此可以在代码中使用C语言的数据类型。CGPoint表示的是笛卡尔平面中的一个坐标（x, y）。

```
struct CGPoint
{
 float x;
 float y;
};
```

CGSize用来存储长度和宽度：

```
struct CGSize
{
 float width;
 float height;
};
```

在之前Shapes相关的程序中，我们本可以使用一个CGPoint和一个CGSize而不是用自定义的表示矩形的struct来表示形状，不过当时我们想让程序尽可能简单。Cocoa提供了一个矩形数据类型，它由坐标和大小复合而成。

```
struct CGRect
{
 CGPoint origin;
 CGSize size;
};
```

Cocoa也为我们提供了创建这些数据类型的快捷函数：CGPointMake()、CGSizeMake()和CGRectMake()。

> **说明** 为什么这些数据类型是C的struct结构体而不是对象呢？原因就在于性能。程序（尤其是GUI程序）会用到许多临时的坐标、大小和矩形区域来完成工作。记住，所有的Objective-C对象都是动态分配的，而动态分配是一个代价较大的操作，它会消耗大量的时间。所以将这些结构体创建成第一级的对象会在使用过程中大大增加系统开销。

8.4 字符串

我们要介绍的第一个真正的类是NSString，也就是Cocoa中用来处理字符串的类。字符串其实就是一组人类可读的字符序列。由于计算机与人类进行定期交互，因此最好让它们有一个可以存储和处理人类可读文本的方式。我们之前已经见过NSString型数据了，它们是特殊的NSString字面量，其标志为双引号内字符串前面的@符号，例如@"Hi!"。这些字面量字符串与你在编程过程中创建的NSString并无差别。

假如你曾经写过C语言的字符串处理，在Dave Mark所著的*Learn C on the Mac*（Apress 2009）中就有很多，那么你一定会明白其中的痛苦。C语言将字符串作为简单的字符数组进行处理，并且在数组最后添加尾部的零字节作为结束标志。而Cocoa中的NSString则有很多内置方法，简化了字符串的处理。

8.4.1　创建字符串

我们知道诸如printf()和NSLog()之类的函数会接受格式字符串和一些参数来输出格式化的结果。NSString的stringWithFormat:方法就是这样通过格式字符串和参数来创建NSString的。

```
+ (id) stringWithFormat: (NSString *) format, …;
```

你可以按如下方式创建一个新的字符串：

```
NSString *height;
height = [NSString stringWithFormat:@"Your height is %d feet, %d inches", 5, 11];
```

得到的字符串是“Your height is 5 feet, 11 inches”。

8.4.2　类方法

stringWithFormat:的声明中有两个值得注意的地方。第一个是定义最后的省略号（...），它告诉我们和编译器这个方法可以接受多个以逗号隔开的其他参数，就像printf()和NSLog()一样。

而另一个古怪但是非常重要的地方是声明语句中有一个非常特别的起始字符：一个加号。难道和Google的社交网络Google+有关吗？当然不是。Objective-C运行时生成一个类的时候，会创建一个代表该类的*类对象*（class object）。类对象包含了指向超类、类名和类方法列表的指针，还包含一个long类型的数据，为新创建的实例对象指定大小（以字节为单位）。

如果你在声明方法时添加了加号，就是把这个方法定义为*类方法*（class method）。这个方法属于类对象（而不是类的示例对象），通常用于创建新的实例。我们称这种用来创建新对象的类方法为*工厂方法*（factory method）。

stringWithFormat:就是一个工厂方法，它根据你提供的参数创建新对象。用stringWithFormat:来创建字符串比创建空字符串然后生成所有元素要容易得多。

类方法也可以用来访问全局数据。AppKit（基于OS X平台）中的NSColor类和UIKit（基于iOS平台）中的UIColor类都拥有以各种颜色命名的类方法，比如redColor和blueColor。要用蓝色绘图，可以像下面这样编写代码。

```
NSColor *haveTheBlues = [NSColor blueColor];
```

或

```
UIColor *blueMan = [UIColor blueColor];
```

你所创建的大部分方法都是实例方法，要用减号（-）作为前缀来进行声明。这些方法将会在指定的对象实例中起作用，比如获取一个Circle的颜色或者一个Tire的压强。如果某个方法所

实现的是很通用的功能，比如创建一个实例对象或者访问一些全局类数据，那么最好使用加号（+）作为前缀将它声明为类方法。

8.4.3 关于大小

NSString中另一个好用的方法（实例方法）是length，它返回的是字符串中的字符个数。

```
- (NSUInteger) length;
```

可以这样使用它：

```
NSUInteger length = [height length];
```

也可以在表达式中使用它，如下所示。

```
if ([height length] > 35)
{
  NSLog (@"wow, you're really tall!");
}
```

> **说明** NSString的length方法能够精确无误地处理各种语言的字符串，如含有俄文、中文或者日文字符的字符串，以及使用Unicode国际字符标准的字符串。在C语言中处理这些国际字符串是件令人头疼的事情，因为一个字符占用的空间可能多于1个字节，这就意味着如strlen()之类只计算字节数的函数会返回错误的数值。

8.4.4 字符串比较

比较是字符串之间常见的操作。有时你会想知道两个字符串是否相等（比如用户名是否为wmalik），而有时你也会想要看看两个字符串可以怎样排列，以便给姓名列表排序。NSString提供了几个用于比较的方法。

isEqualToString:可以用来比较接收方（receiver，接收消息的对象）和作为参数传递过来的字符串。isEqualToString:返回一个BOOL值（YES或NO）来表示两个字符串的内容是否相同。它的声明如下。

```
- (BOOL) isEqualToString: (NSString *) aString;
```

下面是它的使用方法。

```
NSString *thing1 = @"hello 5";
NSString *thing2 = [NSString stringWithFormat: @"hello %d", 5];

if ([thing1 isEqualToString: thing2])
{
  NSLog (@"They are the same!");
}
```

要比较两个字符串，可以使用compare:方法，其声明如下。

```
- (NSComparisonResult) compare: (NSString *) aString;
```

compare:将接收对象和传递过来的字符串逐个进行比较，它返回一个NSComparisonResult（也就是一个enum型枚举）来显示比较结果。

```
enum
{
  NSOrderedAscending = -1,
  NSOrderedSame,
  NSOrderedDescending
};
typedef NSInteger NSComparisonResult;
```

正确比较字符串

比较两个字符串是否相等时，应该使用isEqualToString:，而不能仅仅比较字符串的指针值，举个例子：

```
if ([thing1 isEqualToString: thing2])
{
    NSLog (@"The strings are the same!");
}
```

不同于

```
if (thing1 == thing2)
{
    NSLog (@"They are the same object!");
}
```

这是因为==运算符只判断thing1和thing2的指针数值，而不是它们所指的对象。由于thing1和thing2是不同的字符串，所以第二种比较方式会认为它们是不同的。

因此，如果你想检查两个对象（thing1和thing2）是否为同一事物，就应该使用运算符==。如果是想查看是否相等（即这两个字符串是否内容相同），那么请使用isEqualToString:。

如果你曾经用过C语言中的函数qsort()或bsearch()，那么对此应该比较熟悉。如果compare:返回的结果是NSOrderedAscending，那么左侧的数值就小于右侧的数值，即比较的目标在字母表中的排序位置比传递进来的字符串更靠前。比如，[@"aardvark" compare: @"zygote"]将会返回NSOrderedAscending:。

同样，[@"zoinks" compare: @"jinkies"]将会返回NSOrderedDescending。当然，[@"fnord " compare: @" fnord "]返回的是NSOrderedSame。

8.4.5　不区分大小写的比较

compare:进行的是区分大小写的比较。也就是说，@"Bork"和@"bork"的比较是不会返回NSOrderedSame的。我们还有一个方法compare:options:，它能给我们更多的选择权。

```
- (NSComparisonResult) compare: (NSString *) aString
    options: (NSStringCompareOptions) mask;
```

options参数是一个掩位码。你可以使用位或bitwise-OR运算符（|）来添加选项标记。一些常

用的选项如下。

- NSCaseInsensitiveSearch：不区分大小写字符。
- NSLiteralSearch：进行完全比较，区分大小写字符。
- NSNumericSearch：比较字符串的字符个数，而不是字符串值。如果没有这个选项，100会排在99的前面，程序员以外的人会觉得奇怪，甚至会觉得它是错的。

假如你想比较字符串，需要忽略大小写并按字符个数进行排序，那么应该这么做：

```
if ([thing1 compare: thing2 options: NSCaseInsensitiveSearch | NSNumericSearch]
    == NSOrderedSame)
{
  NSLog (@"They match!");
}
```

8.4.6 字符串内是否还包含别的字符串

有时你可能想看看一个字符串内是否还包含另一个字符串。例如，你也许想知道某个文件名的后缀名是否是.mov，这样你就知道能否用QuickTime Player打开它；你想要查看文件名是否以draft开头以判断它是否是文档的草稿版。有两个方法能帮助你判断：检查字符串是否以另一个字符串开头，判断字符串是否以另一个字符串结尾。

```
- (BOOL) hasPrefix: (NSString *) aString;
- (BOOL) hasSuffix: (NSString *) aString;
```

你可以按如下方式使用这两个方法：

```
NSString *fileName = @"draft-chapter.pages";

if ([fileName hasPrefix: @"draft"])
{
  // this is a draft
}
if ([fileName hasSuffix: @".mov"])
{
  // this is a movie
}
```

于是，draft-chapters.pages会被识别为文档的草稿版本（因为它以draft开头），但是不会将它识别为电影（它的结尾是.pages而不是.mov）。

如果你想知道字符串内的某处是否包含其他字符串，请使用rangeOfString:。

```
- (NSRange) rangeOfString: (NSString *) aString;
```

将rangeOfString:发送给一个NSString对象时，传递的参数是要查找的字符串。它会返回一个NSRange结构体，告诉你与这个字符串相匹配的部分在哪里以及能够匹配上的字符个数。所以下面的示例

```
NSRange range = [fileName rangeOfString: @"chapter"];
```

返回的range.location为6，range.length为7。如果传递的参数在接收字符串中没有找到，那么range.location则等于NSNotFound。

8.4.7　可变性

NSString是不可变（immutable）的，这并不意味着你不能操作它们。不可变的意思是NSString一旦被创建，便不能改变。你可以对它执行各种各样的操作，例如用它生成新的字符串、查找字符或者将它与其他字符串进行比较，但是你不能以删除字符或者添加字符的方式来改变它。

Cocoa提供了一个NSString的子类，叫做NSMutableString。如果你想改变字符串，请使用这个子类。

> **说明**　Java程序员应该很熟悉这种区别。NSString就像Java中的String一样，而NSMutableString则与Java中的StringBuffer类似。

你可以使用类方法stringWithCapacity:来创建一个新的NSMutableString，声明如下：

```
+ (id) stringWithCapacity: (NSUInteger) capacity;
```

这个容量只是给NSMutableString一个建议，可以超过其大小，就像告诉青少年应几点回家一样。字符串的大小并不仅限于所提供的容量，这个容量仅是个最优值。例如，如果你要创建一个大小为40M的字符串，那么NSMutableString可以预分配一块内存来存储它，这样后续操作的速度就会快很多。可按如下方式创建一个新的可变字符串：

```
NSMutableString *string = [NSMutableString stringWithCapacity:42];
```

一旦有了一个可变字符串，就可以对它执行各种操作了。一种常见的操作是通过appendString:或appendFormat:来附加新字符串，如下所示。

```
- (void) appendString: (NSString *) aString;
- (void) appendFormat: (NSString *) format, …;
```

appendString:接收参数aString，然后将其复制到接收对象的末尾。appendFormat:的工作方式与stringWithFormat:类似，但并没有创建新的字符串，而是将格式化的字符串附加在了接收字符串的末尾。例如：

```
NSMutableString *string = [NSMutableString stringWithCapacity:50];
[string appendString: @"Hello there "];
[string appendFormat: @"human %d!", 39];
```

这段代码最后的结果是string被赋值为“Hello there human 39!”。

你也可以使用deleteCharactersInRange:方法删除字符串中的字符。

```
- (void) deleteCharactersInRange: (NSRange) aRange;
```

你将来或许会经常把deleteCharactersInRange:和rangeOfString:连在一起使用。记住，NSMutableString是NSString的子类。凭借面向对象编程的优势，你也可以在NSMutableString中使用NSString的所有功能，包括rangeOfString:方法、字符串比较方法等。举个例子，假设你列出了所有朋友的名字，但又觉得不喜欢Jack了，想要把他从列表中删除，就可以这样操作：

首先，创建朋友列表

```
NSMutableString *friends = [NSMutableString stringWithCapacity:50];
[friends appendString: @"James BethLynn Jack Evan"];
```

接下来，找到Jack的名字在字符串中的范围

```
NSRange jackRange = [friends rangeOfString: @"Jack"];
jackRange.length++; // eat the space that follows
```

在这个例子中，字符范围开始于15，长度为5。现在，我们就可以把Jack从圣诞贺卡名单中踢掉了：

```
[friends deleteCharactersInRange: jackRange];
```

如此一来，这个字符串就剩下了“James BethLynn Evan”。

在实现description方法时，使用可变字符串是非常方便的。你可以通过appendString和appendFormat方法为对象创建一个详尽的描述。

由于NSMutableString是NSString的子类，所以我们“无偿”获得了两个特性。第一个就是任何使用NSString的地方，都可以用NSMutableString来替代。任何NSString可行的场合NSMutableString也能畅通无阻。程序员在使用字符串时完全不必担心它是否是可变的字符串。

另一个特性源于继承，与实例方法一样，继承对类方法也同样适用。所以，NSString中非常方便的类方法stringWithFormat:也可以用来创建新的NSMutableString对象。你可以轻松地以字符格式来创建一个可变字符串：

```
NSMutableString *string = [NSMutableString stringWithFormat: @"jo%dy", 2];
```

stirng的初始值是jo2y，当然你也可以执行其他的操作，例如在给定的范围内删除字符或者在特定的位置插入字符。查阅关于NSString和NSMutableString的文档，可以学习这两个类中所有方法的全部细节。

8.5 集合大家族

各种独立的对象聚在一起感觉很有趣，不过通常你会希望它们井井有条。Cocoa提供了许多集合类，如NSArray和NSDictionary，它们的实例就是为了存储其他对象而存在的。

8.5.1 NSArray

你在C语言中应该用到过数组。事实上，在本书前面的内容中，我们也用过数组来存储汽车的4个轮胎。你或许还记得在编写那段代码时我们遇到了不少困难，比如不得不去检查数组的索引来确定它是否有效：索引既不能小于0也不能大于数组的长度。另一个问题是这个长度为4的数组是被硬编码进Car类的，也就是说我们车的轮胎不能多于4个。当然，这似乎不是什么问题，但是谁知道未来的火箭飞行器是否需要4个以上的轮胎才能够平稳降落呢？

NSArray是一个Cocoa类，用来存储对象的有序列表。你可以在NSArray中放入任意类型的对象：NSString、Car、Shape、Tire或者其他你想要存储的对象，甚至可以是其他数组或字典对象。

只要拥有一个NSArray对象，就可以通过各种方式来操作它，比如让某个对象的实例变量指

向这个数组，将该数组当作参数传递给方法或函数，获取数组中所存对象的个数，提取某个索引所对应的对象，查找数组中的对象，遍历数组等。

NSArray类有两个限制。首先，它只能存储Objective-C的对象，而不能存储原始的C语言基础数据类型，如int、float、enum、struct和NSArray中的随机指针。同时，你也不能在NSArray中存储nil（对象的零值或NULL值）。有很多种方法可以避开这些限制，你马上就会看到。

可以通过类方法arrayWithObjects:创建一个新的NSArray。发送一个以逗号分隔的对象列表，在列表结尾添加nil代表列表结束（顺便提一下，这就是不能在数组中存储nil的一个原因）。

```
NSArray *array = [NSArray arrayWithObjects:@"one", @"two", @"three", nil];
```

这行代码创建了一个由NSString字面量对象组成的含3个元素的数组。你也可以使用数组字面量格式来创建一个数组，它与NSString字面量格式非常类似，区别是用方括号代替了引号，如下所示。

```
NSArray *array2 = @[@"one", @"two", @"three"];
```

虽然array和array2对象不同，但它们的内容是一样的，而且后一种的输入量明显比前一种少很多。

> **说明**　使用字面量语法时不必在结尾处特意补上nil。

只要有了一个数组，就可以获取它所包含的对象个数：

```
- (NSUInteger)count;
```

也可以获取特定索引处的对象：

```
- (id)objectAtIndex:(NSUInteger)index;
```

通过字面量访问数组的语法与C语言中访问数组项的语法类似。

```
id *myObject = array1[1];
```

你可以结合计数和取值功能来输出数组中的内容：

```
for (NSInteger i = 0; i < [array count]; i++)
{
    NSLog (@"index %d has %@.",i, [array objectAtIndex:i]);
}
```

你也可以使用数组字面量语法来写以上代码，如下所示。

```
for (NSInteger i = 0; i < [array count]; i++)
{
  NSLog (@"index %d has %@.",i, array[i]);
}
```

输出结果如下所示。

```
index 0 has one.
index 1 has two.
index 2 has three.
```

如果你引用的索引大于数组中对象的个数，那么Cocoa在运行时会输出错误。比如说运行以下代码。

```
[array objectAtIndex:208000];
array[208000];
```

你将会看到如下警告：

```
*** Terminating app due to uncaught exception 'NSRangeException',
reason: '*** -[__NSArray objectAtIndex:]: index 208000 beyond bounds [0 .. 2]'
```

进行Cocoa编程时，你可能会经常看到这样的提示信息，所以我们花些时间来讲解一下。在告诉你程序被终止这个坏消息之后，这个提示信息还提到终止的原因之一是“未捕获的异常”（uncaught exception）。异常（exception）是Cocoa说明“我不知道该如何处理”的方式。你有许多方法来捕获和处理代码中的异常，但是如果你只是刚开始学习，就没有必要这样做了。这里的异常是NSRangeException，表明传递给方法的范围参数有问题。这个方法也就是NSArray类下的objectAtIndex:方法。

异常信息中最后一部分内容便是我们需要关心的，它指出我们在读取数组中索引为208000的数据，但是在该数组中只有3个元素（差了不止一个数量级）。通过这条信息，我们可以追查到问题代码并且找到错误所在的位置。

查找异常产生的原因是件令人沮丧的事情。你所获得的所有信息就是Console窗口中的内容，而GUI程序会继续运行不会停止。有一种方法可以在异常发生时让Xcode中断程序并进入调试器，这样会更方便操作。在导航栏面板中选中断点选项卡，你会看到断点以列表形式显示出来。当前我们只设置了一个断点，如图8-1所示。

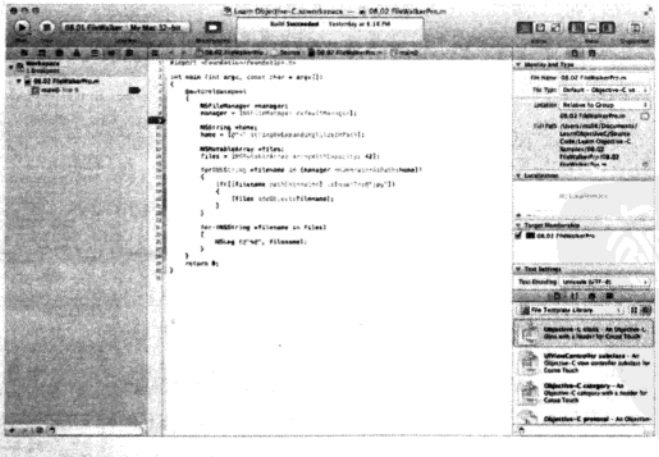


图8-1 Xcode的断点窗口

如果想要添加其他断点，只需点击面板左下角的加号（+）按钮，然后在弹出菜单中选择“Add Exception Breakpoint...”选项。你将会看到另一个弹出对话框，如图8-2所示。

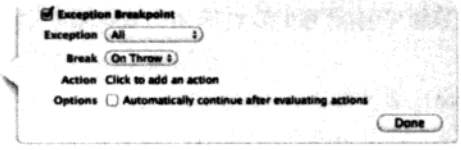


图8-2　添加异常断点

在设置断点之前，它会让你自定义断点的行为。举个例子，你可以让它输出错误信息或终止程序并执行一些命令。目前我们还不会做这么复杂的事情，仅需要让程序在异常抛出时能终止运行。完成之后点击Done按钮就能添加断点了。添加完之后，窗口会如图8-3所示。现在当运行程序遇到异常抛出时，调试器会终止运行并指向包含断点的那行，如图8-4所示。

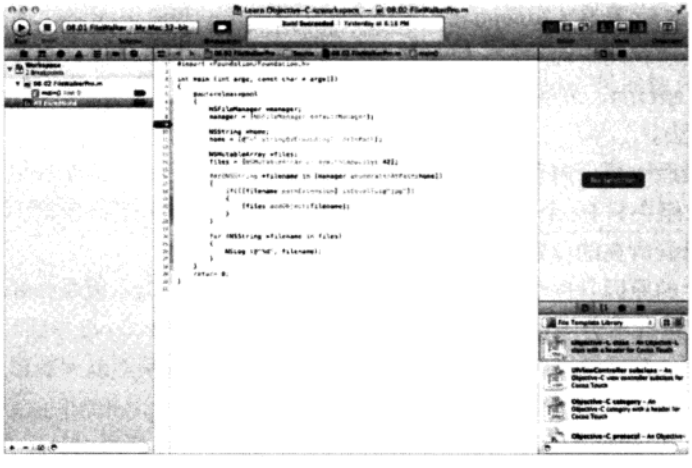


图8-3　Xcode调试器指向了包含断点的那行

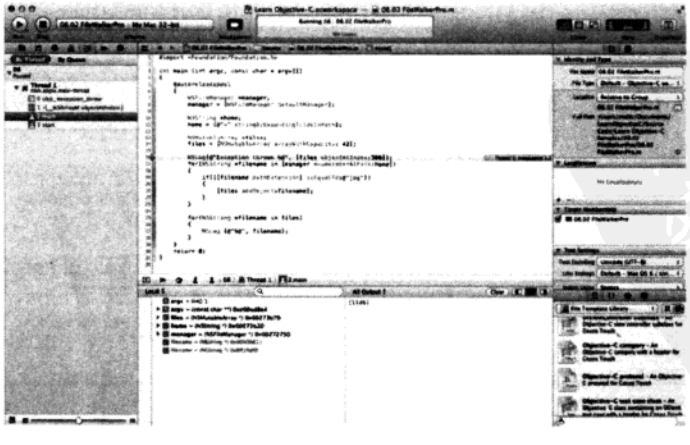


图8-4　程序停在了断点处

你也许会不太高兴，因为程序仅仅溢出了数组的上限就导致Cocoa有这么大的反应。但是相信我，你将来会意识到这其实是一个强大的功能，因为它可以让你捕获那些可能无法检查出来的错误。

与NSString一样，NSArray也有很多特性。例如，你可以让数组给出特定对象的位置，根据当前数组中的元素创建新数组，用给定的分隔符将所有数组元素合成为一个字符串（便于创建以逗号分割的列表），或者创建已有数组的排序版本。

切分数组

如果你曾经使用过Perl或Python这种脚本语言，那么可能已经习惯于将字符串切分成数组和将数组元素合并成字符串这种操作了。NSArray也可以执行这种操作。

使用-componentsSeparatedByString:可以切分NSArray，像这样：

```
NSString *string = @"oop:ack:bork:greeble:ponies";
NSArray *chunks = [string componentsSeparatedByString: @":"];
```

还可以用componentsJoinedByString:来合并NSArray中的元素并创建字符串：

```
string = [chunks componentsJoinedByString: @" :- ) "];
```

上面的代码行将会创建一个内容为"oop :-) ack :-) bork :-) greeble :-) ponies "的NSString字符串。

8.5.2 可变数组

与NSString一样，NSArray创建的是不可变对象的数组。一旦你创建了一个包含特定数量的对象的数组，它就固定下来了：你既不能添加任何元素也不能删除任何元素。当然数组中包含的对象是可以改变的（比如Car在安全检查失败后可以获得一套新的Tire），但数组对象本身是一直都不会改变的。

为了弥补NSArray类的不足，便有了NSMutableArray这个可变数组类，这样就可以随意地添加或删除数组中的对象了。NSMutableArray通过类方法arrayWithCapacity来创建新的可变数组：

```
+ (id) arrayWithCapacity: (NSUInteger) numItems;
```

与NSMutableString中的stringWithCapacity:一样，数组容量也只是数组最终大小的一个参考。容量数值之所以存在，是为了让Cocoa能够对代码进行一些优化。Cocoa既不会将对象预写入数组中，也不会用该容量值来限制数组的大小。你可以按照如下方式创建可变数组：

```
NSMutableArray *array = [NSMutableArray arrayWithCapacity: 17];
```

使用addObject:在数组末尾添加对象：

```
- (void) addObject: (id) anObject;
```

使用如下循环代码为数组添加4个轮胎：

```
for (NSInteger i = 0; i < 4; i++)
{
 Tire *tire = [Tire new];
 [array addObject: tire];
}
```

还可以删除特定索引处的对象。假设你不喜欢第二个轮胎，可以用**removeObjectAtIndex:**来删除它。下面是该方法的定义。

```
- (void) removeObjectAtIndex: (NSUInteger) index;
```

像这样使用它就行了：

```
[array removeObjectAtIndex:1];
```

注意，第二个轮胎的索引是1。NSArray中的对象是从零开始编制索引的，这与C数组的规范一样。

现在只剩下3个轮胎了。删除一个对象之后，数组中并没有留下漏洞。位于被删除对象后面的数组元素都被前移来填补空缺了。

可变数组的其他方法也能够用来实现很多出色的操作，例如在特定索引处插入对象、替换对象、为数组排序，还包括了从NSArray所继承的大量好用的功能。

> **说明**　没有可以用来创建NSMutableArray对象的字面量语法。

8.5.3　枚举

NSArray经常要对数组中的每个元素都执行同一个操作。比方说，你喜欢绿色，可以让数组中的所有几何形状都变成绿色。你也可以给汽车的每个轮胎放一些气，使你在匹兹堡地区崎岖路况中的驾驶模拟器更逼真。你可以编写一个从0到[array count]的循环来读取每个索引处的对象，也可以使用NSEnumerator，Cocoa可以用它来表示集合中迭代出的对象。要想使用NSEnumerator，需通过objectEnumerator向数组请求枚举器：

```
- (NSEnumerator *)objectEnumerator;
```

你可以使用这个方法：

```
NSEnumerator *enumerator = [array objectEnumerator];
```

如果你想要从后往前浏览某个集合，还有一个reverseObjectEnumerator方法可以使用。

在获得枚举器之后，便可以开始一个while循环，每次循环都向这个枚举器请求它的nextObject（下一个对象）：

```
- (id) nextObject;
```

nextObject返回nil值时，循环结束。这也是不能在数组中存储nil值的另一个原因：我们没有办法判断nil是存储在数组中的数值还是代表循环结束的标志。

整个循环代码如下所示：

```
NSEnumerator *enumerator = [array objectEnumerator];
while (id thingie = [enumerator nextObject])
{
 NSLog (@"I found %@", thingie);
}
```

对可变数组进行枚举操作时，有一点需要注意：你不能通过添加或删除对象这类方式来改变数组的容量。如果这么做了，枚举器就会出现混乱，你也会获得未定义结果（undefined result）。“未定义结果”可以代表任何意思，可以是“诶，好像重复了”，也可能是“噢，我的程序崩溃了”，等等。

8.5.4 快速枚举

在Mac OS X 10.5（Leopard）中，苹果公司对Objective-C引入了一些小改进，使语言版本升级到了2.0。我们要介绍的第一个改进叫做快速枚举（fast enumeration），它的语法与脚本语言类似，如下面的代码所示。

```
for (NSString *string in array)
{
 NSLog (@"I found %@", string);
}
```

这个循环体将会遍历数组中的每一个元素，并且用变量string来存储每个数组值。它比枚举器语法更加简洁快速。

与Objective-C 2.0的其他新特性一样，快速枚举不能在旧的Mac（Mac OS X 10.5之前的）系统上运行。如果你或者你的用户需要在一台不支持Objective-C 2.0及更新版本的电脑上运行你的程序，就不能使用这个新语法。现在这个问题依然存在。

最新版本的苹果编译器（基于Clang和LLVM项目）为纯C语言添加了一个叫做代码块（block）的特性。我们将会在第14章讨论代码块。为了支持代码块功能，苹果公司添加了一个能在NSArray中通过代码块枚举对象的方法，代码如下所示：

```
- (void)enumerateObjectsUsingBlock:(void (^)(id obj, NSUInteger idx, BOOL *stop))block
```

这是什么东西？你也许不知道怎样去读这行代码，更别说怎么去用了。没事，不用担心。它其实没有看起来那么复杂。我们可以用之前的那个数组来重写这个枚举方式。

```
[array enumerateObjectsUsingBlock:^(NSString *string, NSUInteger index, BOOL *stop) {
    NSLog (@"I found %@", string);
}];
```

现在的问题是：“为什么我们要使用它来代替快速枚举？”因为通过代码块可以让循环操作并发执行，而通过快速枚举，执行操作要一项项地线性完成。

好了，现在我们有4种方法来遍历数组了：通过索引、使用NSEnumerator、使用快速枚举和最新的代码块方法。你会使用哪一种方法呢？

如果你使用的不是10.5版本之前的老爷机，那么建议使用快速枚举或代码块，因为它们更简洁快速。顺便提一下，代码块只在Apple LLVM编译器上才会有效。

如果你还要支持Mac OS X 10.4或更早的系统（这事基本上没有任何可能了），那就使用NSEnumerator。Xcode有重构功能，可以将代码转换成Objective-C 2.0，它会自动将NSEnumerator循环转换成快速枚举。

只有在真的需要用索引访问数组时才应使用-objectAtIndex方法，例如跳跃地浏览数组时（每隔两个对象读取一次数组对象）或者同时遍历多个数组时。

8.5.5 NSDictionary

你肯定听说过字典，也许偶尔还会用到它。在编程中，字典（dictionary）是关键字及其定义的集合。Cocoa中有一个实现这种功能的集合类NSDictionary。NSDictionary能在给定的关键字（通常是一个NSString字符串）下存储一个数值（可以是任何类型的Objective-C对象），然后你就可以用这个关键字来查找相应的数据。因此假如你有一个存储了某人所有联系方式的NSDictionary，那么你可以对这个字典说“给我关键字home-address下的值”或者“给我关键字email-address下的值”。

> **说明**　为什么不用数组存储然后在数组里查询数值呢？字典（也被称为散列表或关联数组）使用的是键查询的优化方式。它可以立即找出要查询的数据，而不需要遍历整个数组。对于频繁的查询和大型的数据集来说，使用字典比数组要快得多。

你可能已经猜到，NSDictionary就像NSString和NSArray一样是不可变的对象。但是NSMutableDictionary类允许你随意添加和删除字典元素。在创建新的NSDictionary时，就要提供该字典所存储的全部对象和关键字。

学习字典的最简单的方法就是用字典字面量语法，它与类方法dictionaryWithObjectsAndKeys:非常相似。

字面量语法即使用类似@{key:value,…}的方法来定义。需要注意它使用的是大括号而不是方括号。还要注意dictionaryWithObjectsAndKeys:后面的参数先是要存储的对象，然后才是关键字，而字面量的语法则是关键字在前，数值在后。关键字和数值之间以冒号分开，而每对键值之间则用逗号区分开。

```
+ (id) dictionaryWithObjectsAndKeys: (id) firstObject, …;
```

该方法接收对象和关键字交替出现的序列，以nil值作为终止符号（你可能已经猜到了，NSDictionary中不能存储nil值，而且字面量语法中不需要nil值）。假设我们想创建一个存储汽车轮胎的字典，而轮胎要使用可读的标签进行查找而不是数组中的任意索引。你可以按下面的方式创建这种字典。

```
Tire *t1 = [Tire new];
Tire *t2 = [Tire new];
Tire *t3 = [Tire new];
```

```
Tire *t4 = [Tire new];
NSDictionary *tires = [NSDictionary dictionaryWithObjectsAndKeys: t1,
    @"front-left", t2, @"front-right", t3, @"back-left", t4, @"back-right", nil];
```

也可以使用

```
NSDictionary *tires = @{@"front-left" : ti, @"front-right" : t2, @"back-left" : t3,
    @"back-right" : t4};
```

使用objectForKey:方法并传递前面用来存储的关键字，就可以访问字典中的数值了：

```
- (id) objectForKey: (id) aKey;
```

或者是

```
tires[key];
```

所以假设你要查找右后方的轮胎，可以这样写：

```
Tire *tire = [tires objectForKey:@"back-right"];
```

或者是

```
Tire *tire = tires[@"back-right"];
```

如果字典里没有右后方的轮胎（假设它是一辆只有3个轮子的奇怪汽车），字典将会返回nil值。

向NSMutableDictionary类发送dictionary消息，便可以创建新的NSMutableDictionary对象，也可以使用dictionaryWithCapacity:方法来创建新的可变字典并告诉Cocoa该字典的最终大小。（有没有发现Cocoa的命名方式很有规律？）

```
+ (id) dictionaryWithCapacity: (NSUInteger) numItems;
```

与之前提到过的NSMutableString和NSMutableArray一样，在这里，字典的容量也仅仅是一个建议，而不是对其大小的限制。你可以使用setObject:forKey:方法为字典添加元素：

```
- (void)setObject:(id)anObject forKey:(id)aKey;
```

存储轮胎的字典还有一种创建方法：

```
NSMutableDictionary *tires = [NSMutableDictionary dictionary];
[tires setObject:tl forKey:@"front-left"];
[tires setObject:t2 forKey:@"front-right"];
[tires setObject:t3 forKey:@"back-left"];
[tires setObject:t4 forKey:@"back-right"];
```

如果对字典中已有的关键字使用setObject:forKey:方法，那么这个方法将会用新值替换掉原有的数值。如果想在可变字典中删除一些关键字，可使用removeObjectForKey:方法：

```
- (void) removeObjectForKey: (id) aKey;
```

所以，如果想模拟一只轮胎脱落的场景，就可以把那只轮胎从字典中删除：

```
[tires removeObjectForKey:@"back-left"];
```

与NSArray一样，没有适用于NSMutableDictionary的字面量初始化语法。

8.5.6　请不要乱来

如果你很有创造力，并跃跃欲试地想要创建NSString、NSArray或NSDictionary的子类，请不要这么做。在一些语言中，你确实会用到字符串和数组的子类来完成工作，但是在Cocoa中，许多类实际上是以类簇（class clusters）的方式实现的，即它们是一群隐藏在通用接口之下的与实现相关的类。创建NSString对象时，实际上获得的可能是NSLiteralString、NSCFString、NSSimpleCString、NSBallOfString或者其他未写入文档的与实现相关的对象。

程序员在使用NSString和NSArray时不必在意系统内部到底用的是哪个类，但要给一个类簇创建子类是一件令人非常痛苦和沮丧的事。通常，可以将NSString或NSArray复合到你的某个类中或者使用类别（将在第12章中介绍）来解决这种编程中的问题，而不用特意去创建子类。

8.6　其他数值

NSArray和NSDictionary只能存储对象，而不能直接存储任何基本类型的数据，如int、float和struct。不过你可以用对象来封装基本数值，比如将int型数据封装到一个对象中，就可以将这个对象放入NSArray或NSDictionary中了。

如果你想使用对象来处理基本类型，就可以使用NSInteger和NSUInteger。这些类型也要针对32位和64位处理器对数值进行统一。

8.6.1　NSNumber

Cocoa提供了NSNumber类来封装（wrap，即以对象形式来实现）基本数据类型。可以使用以下类方法来创建新的NSNumber对象：

```
+ (NSNumber *) numberWithChar: (char) value;
+ (NSNumber *) numberWithInt: (int) value;
+ (NSNumber *) numberWithFloat: (float) value;
+ (NSNumber *) numberWithBool: (BOOL) value;
```

还有许多这样的创建方法，包括无符号整数和long型整数以及long long整型数据的版本，不过以上是几个最常用的方法。

也可以使用字面量语法来创建这些对象：

```
NSNumber *number;
number = @ 'X' ; //字符型
number = @12345; //整型
number = @12345ul; //无符号长整数
number = @12345ll; // long long
number = @123.45f; //浮点型
number = @123.45; //双浮点型
number = @YES; //布尔值
```

创建完NSNumber之后，可以把它放入一个字典或数组中：

```
NSNumber *number = @42;
```

```
[array addObject number];
[dictionary setObject: number forKey: @"Bork"];
```

在将一个基本类型数据封装到NSNumber中后，你可以通过下面的实例方法来重新获得它：

```
- (char) charValue;
- (int) intValue;
- (float) floatValue;
- (BOOL) boolValue;
- (NSString *)stringValue;
```

将创建方法和提取方法搭配在一起使用是完全可以的。例如，用numberWithFloat:创建的NSNumber对象可以用intValue方法来提取数值。NSNumber会对数据进行适当的转换。

> **说明** 通常将一个基本类型的数据封装成对象的过程被称为**装箱**（boxing），从对象中提取基本类型的数据叫做**开箱**（unboxing）。有些语言有自动装箱的功能，可以自动封装基础类型的数据，也可以自动从封装后的对象中提取基础数据。Objective-C语言不支持自动装箱功能。

8.6.2 NSValue

NSNumber实际上是NSValue的子类，NSValue可以封装任意值。你可以使用NSValue将结构体放入NSArray或NSDictionary中。使用下面的类方法便能创建新的NSValue对象。

```
+ (NSValue *) valueWithBytes: (const void *) value objCType: (const char *) type;
```

传递的参数是你想要封装的数值的地址（如一个NSSize或你自己的struct）。通常你得到的是想要存储的变量的地址（在C语言要使用操作符&）。你也可以提供一个用来描述这个数据类型的字符串，通常用来说明struct中实体的类型和大小。你不用自己写代码来生成这个字符串，@encode编译器指令可以接收数据类型的名称并为你生成合适的字符串，所以按照如下方式把NSRectfangru NSArray中吧。

```
NSRect rect = NSMakeRect (1, 2, 30, 40);
NSValue *value = [NSValue valueWithBytes:&rect objCType:@encode(NSRect)];
[array addObject:value];
```

可以使用方法getValue:来提取数值：

```
- (void)getValue:(void *)buffer;
```

调用getValue:时，需要传递存储这个数值的变量地址：

```
value = [array objectAtIndex: 0]; [value getValue: &rect];
```

> **说明** 在上面的getValue:的例子中，你可以看到方法名中使用了get，表明我们提供的是一个指针，而指针所指向的空间则用来存储该方法生成的数据。

Cocoa提供了将常用的struct型数据转换成NSValue的便捷方法，如下所示。

```
+ (NSValue *)valueWithPoint:(NSPoint)aPoint;
+ (NSValue *)valueWithSize:(NSSize)size;
+ (NSValue *)valueWithRect:(NSRect)rect;
- (NSPoint)pointValue;
- (NSSize)sizeValue;
- (NSRect)rectValue;
```

可按照以下方式在NSArray中存储和提取NSRect值：

```
value = [NSValue valueWithRect:rect];
[array addObject: value];
…
NSRect anotherRect = [value rectValue];
```

8.6.3　NSNull

我们提到过不能在集合中放入nil值，因为在NSArray和NSDictionary中nil有特殊的含义，但有时你确实需要存储一个表示“这里什么都没有”的值。例如，你有一个存储某人联系信息的字典，在关键字@"home fax machine"下存储的是这个人的家庭传真号码。如果这个关键字下存储了一个传真号码，那么你就知道这个人有一台传真机。但是，如果在这个字典里没有这个值，是代表这个人没有家用传真机还是代表你不知道他到底有没有家用传真机呢？

使用NSNull便可以消除这种歧义。你可以设定关键字@"home fax machine"下的NSNull值代表的是这个人确实没有传真机，而关键字没有数值则代表你还不确定他是否有传真机。

NSNull大概是Cocoa里最简单的类了，它只有一个方法：

```
+ (NSNull *) null;
```

你可以按照下面的方式将它添加到集合中：

```
[contact setObject: [NSNull null] forKey: @"home fax machine"];
```

而访问它的方式则如下所示：

```
id homefax = [contact objectForKey: @"home fax machine"];
if (homefax == [NSNull null])
{
// ……确定没有传真机后的行为
}
```

[NSNull null]总是返回一样的数值，所以你可以使用运算符==将该值与其他值进行比较。

8.7　示例：查找文件

好了，我们已经说了一大堆理论知识了。下面是一个实用的程序，它是用本章中介绍的类组成的。程序FileWalker（可以在项目文件夹08.01 FileWalker中找到）可以翻查你在Mac电脑上的主目录以查找.jpg文件并打印到列表中。我们必须承认这个程序不算酷得惊人，但是它的确实现了一些功能。

FileWalker程序用到了NSString、NSArray、NSEnumerator和其他两个用来与文件系统交互的Foundation类。

这个示例还用到了NSFileManager，它允许你对文件系统进行操作，例如创建目录、删除文件、移动文件或者获取文件信息。在这个例子中，我们将会要求NSFileManager类创建一个NSDirectoryEnumerator对象来遍历文件的层次结构。

整个FileWalker程序都存放在main()函数中，因为我们并没有创建任何属于我们自己的类。下面是main()函数的全部代码。

```
int main (int argc, const char * argv[])
{
  @autoreleasepool
  {
    NSFileManager *manager;
    manager = [NSFileManager defaultManager];

    NSString *home;
    home = [@"~" stringByExpandingTildeInPath];

    NSDirectoryEnumerator *direnum;
    direnum = [manager enumeratorAtPath:home];

    NSMutableArray *files;
    files = [NSMutableArray arrayWithCapacity:42];

    NSString *filename;
    while (filename = [direnum nextObject])
    {
      if ([[filename pathExtension] isEqualTo: @"jpg"]) {
        [files addObject: filename];
      }
    }
    NSEnumerator *fileenum;
    fileenum = [files objectEnumerator];

    while (filename = [fileenum nextObject])
    {
      NSLog (@"%@", filename);
    }
  }
  return 0;

} // main
```

现在，让我们逐步了解这个程序。最上面是自动释放池（@autoreleasepool）的样本代码（第9章详细介绍）。

下一步是获取NSFileManager对象。NSFileManager中有一个叫做defaultManager的类方法，可以创建一个属于我们自己的NSFileManager对象。

```
NSFileManager *manager = [NSFileManager defaultManager];
```

在Cocoa中这种情况很常见，很多类都支持单例架构（singleton architecture），也就是说你只需要一个实例就够了。文件管理器、字体管理器或图形上下文确实只需要一个就够了。这些类都提供了一个类方法让你访问唯一的共享对象，完成你的工作。

在这个示例中，我们需要一个目录迭代器。但是在要求文件管理器创建目录迭代器之前，需要先确定从文件系统中的什么位置开始查找文件。若是从硬盘最上层目录开始查询，就会消耗大量的时间，所以我们仅在主目录里查找。

如何指定目录呢？可以使用绝对路径，如"/Users/wmalik/"（其中wmalik是作者姓名的缩写），但这就有局限性了，该路径只有在主目录是wmalik时才有效。所幸，Unix系统（包括OS X系统）有一个代表主目录的速记符号~（也被称为波浪号）。即使你不会西班牙语（Español），这个符号也还是有用处的。~/Documents代表文稿（Documents）目录，而~/junk/oopack.txt在Waqar的计算机上就是/Users/wmalik/junk/oopack.txt。NSString中有一个方法可以接收~字符并将其展开成主目录路径，也就是接下来的这一行代码：

```
NSString *home = [@"~" stringByExpandingTildeInPath];
```

stringByExpandingTildeInPath方法将~替换成了当前用户的主目录。在Waqar的计算机上，主目录是/Users/wmalik。接下来，我们将路径字符串传递给文件管理器。

```
NSDirectoryEnumerator *direnum = [manager enumeratorAtPath: home];
```

enumeratorAtPath:返回一个NSDictionaryEnumerator对象，它是NSEnumerator的子类。每次在这个枚举器对象中调用nextObject方法时，都会返回该目录中下一个文件的路径。这个方法也能搜索子目录中的文件。迭代循环结束时，你将会得到主目录中每一个文件的路径。NSDictionaryEnumerator还提供了一些其他功能，比如为每个文件创建一个属性字典，但是在这里我们并不会用到。

由于我们要查找.jpg文件（即路径名称以.jpg结尾）并输出它们的名称，所以需要存储这些名称的空间。可以在枚举中通过NSLog()把它们都打印出来，不过也许在以后，我们会想要在程序的其他位置对所有这些文件执行一些操作。所以，NSMutableArray就是最佳选择。我们创建一个可变数组并将查找到的路径添加进去：

```
NSMutableArray *files = [NSMutableArray arrayWithCapacity:42];
```

我们不知道到底会找到多少个.jpg文件，所以就选择了42[①]，因为……你懂的。由于容量参数并不能限制数组的大小，所以在任何情况下，这样定义都是可以的。

最后就是程序的核心内容了。所有的准备工作都已就绪，现在该开始循环了。

```
NSString *filename;
while (filename = [direnum nextObject]) {
```

目录枚举器将会返回一个代表文件路径的NSString字符串。就像NSEnumerator一样，当枚举器结束时它会返回nil值，于是循环终止。

① 《银河系漫游指南》一书提到过42是"生命、宇宙与万事万物的终极解答"。如果在Google中搜索"the answer to life, the universe, and everything"，Google会直接回答42，而且还是用Google计算器得出来的。——译者注

NSString提供了许多处理路径名称和文件名称的便捷工具，比如说pathExtension方法能输出文件的扩展名（其中不包括前面的那个点）。因此为oopack.txt文件调用pathExtension将会返回@"txt"，而VikkiCat.jpg会返回@"jpg"。

我们使用嵌套的方法调用来获取路径扩展名并将获得的字符串传递给isEqualTo:方法。如果该调用的返回结果是YES，则该文件名将会被添加到文件数组中，如下所示。

```
if ([[filename pathExtension] isEqualTo: @"jpg"])
{
 [files addObject: filename];
}
```

目录循环结束后，遍历文件数组，用NSLog()函数将数组内容输出。

```
NSEnumerator *fileenum = [files objectEnumerator];
while (filename = [fileenum nextObject])
{
 NSLog (@"%@", filename);
}
```

接下来，我们通知main()函数返回0来表示程序成功退出。

```
return (0);
} // main
```

下面是在Waqar的计算机上运行该程序的结果示例。

```
cocoaheads/DSCN0798.jpg
cocoaheads/DSCN0804.jpg
cow.jpg
Development/Borkware/BorkSort/cant-open-file.jpg
Development/Borkware/BSL/BWLog/accident.jpg
```

成功了！不过要显示完全部的结果可能需要一些时间，因为程序也许将不得不去翻查成千上万张图片文件以完成查询工作。

通过快速枚举方法（不支持Leopard之前的系统）

FileWalker程序使用的是典型的迭代方式，而08.02 FileWalkerPro文件夹中的程序则展示了使用快速枚举的解决方法。快速枚举语法有一个很棒的特性，即你可以将已有的NSEnumerator对象或其子类传递给它。而刚好NSDictionaryEnumerator是NSEnumerator的子类，所以我们可以放心地将-enumeratorAtPath:方法返回的结果传递给快速枚举。

```
int main (int argc, const char * argv[])
{
  @autoreleasepool
  {
    NSFileManager *manager;
    manager = [NSFileManager defaultManager];
```

```
        NSString *home;
        home = [@"~" stringByExpandingTildeInPath];

        NSMutableArray *files;
        files = [NSMutableArray arrayWithCapacity: 42];

        for(NSString *filename in [manager enumeratorAtPath:home])
        {
          if([[filename pathExtension] isEqualTo:@"jpg"])
          {
            [files addObject:filename];
          }
        }
        for(NSString *filename in files)
        {
          NSLog (@"%@", filename);
        }
      }
      return 0;

} // main
```

如你所见，这个版本的程序比之前的那个版本要简单：我们删除了两个枚举器变量及其支持代码。

8.8　小结

本章介绍了很多内容，包括三个新的语言特性：类方法，即由类本身而不是某个示例来处理的方法；@encode()指令，它用于需要描述C语言基础类型的方法；快速枚举。

我们研究了很多有用的Cocoa类，包括NSString、NSArray和NSDictionary。NSString用来存储人可以直接看懂的文本，而NSArray和NSDictionary用来存储对象的集合。这些对象都是不可变的：一旦创建就不能再改动。Cocoa提供了这些类的可变版本，你可以随意更改它们的内容。

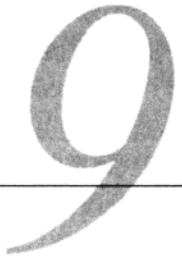
尽管我们已经尽力而为（并且本章的篇幅并不短），也只是认识了Cocoa类的冰山一角。你可以查阅资料，学习更多有关这些类的知识，这会让你收获更多乐趣，积累更多智慧。

最后，我们利用所学的类在主目录中搜索图片文件。

下一章将深入研究内存管理的奇妙之处，你将学到在系统垃圾生成后如何清理程序。

第9章

内存管理



本章将介绍如何使用Objective-C和Cocoa进行内存管理。内存管理是程序设计中常见的资源管理（resource management）的一部分。每个计算机系统可供程序使用的资源都是有限的，包括内存、打开的文件以及网络连接等。如果使用了某种资源，比如因打开文件而占用了资源，那么需要随后对其进行清理（这种情况下，关闭文件即可）。如果你不断打开文件并且保持打开状态而从不关闭，最终将耗尽文件资源。以公共图书馆为例，假如每个人都只借不还，那么图书馆最终将会因无书可借而倒闭，每个人也都无法再使用图书馆。任何人都不希望出现这种结果。

虽说当程序运行结束时，操作系统将收回其占用的资源，但是只要程序还在运行，它就会一直占用资源。如果不进行清理，某些资源最终将被耗尽，程序有可能会崩溃。而且随着操作系统的发展，程序何时会终止运行会变得更加难以捉摸。

虽然不是每个程序都会使用文件或网络连接，但都会消耗内存。每个C语言程序员都会遇到与内存相关的错误，这种错误是灾难性的。而使用Java和脚本语言的程序员则不需要考虑此类问题：这些语言的内存管理是自动进行的，就像父母会给孩子打扫房间一样。另一方面，我们必须确保在需要的时候分配内存，在程序运行结束时释放占用的内存。如果我们只分配而不释放内存，则会发生**内存泄漏**（leak memory）：程序的内存占用量不断增加，最终会被耗尽并导致程序崩溃。同样需要注意的是，不要使用任何刚释放的内存，否则可能误用陈旧的数据，从而引发各种各样的错误，而且如果该内存已经加载了其他数据，将会破坏这些新数据。

> **说明** 内存管理不是一个容易解决的问题。虽然Cocoa的解决方案非常简洁，但是要想精通它还需费些时日。即使是有几十年经验的程序员，第一次遇到这种情况时也会有一定的困难，因此，如果你一时未能完全读懂，请不必担心。

9.1 对象生命周期

正如现实世界中的鸟类和蜜蜂一样，程序中的对象也有生命周期。对象的生命周期包括诞生（通过alloc或new方法实现）、生存（接收消息并执行操作）、交友（通过复合以及向方法传递参数）以及最终死去（被释放掉）。当生命周期结束时，它们的原材料（内存）将被回收以供新的对象使用。

9.1.1　引用计数

现在，对象何时诞生我们已经很清楚了，而且也讨论了如何使用对象，但是怎么知道对象生命周期结束了呢？Cocoa采用了一种叫做引用计数（reference counting）的技术，有时也叫做保留计数（retain counting）。每个对象都有一个与之相关联的整数，被称作它的引用计数器或保留计数器。当某段代码需要访问一个对象时，该代码就将该对象的保留计数器值加1，表示“我要访问该对象”。当这段代码结束对象访问时，将对象的保留计数器减1，表示它不再访问该对象。当保留计数器的值为0时，表示不再有代码访问该对象了（唉，可怜的对象），因此它将被销毁，其占用的内存被系统回收以便重用。

当使用alloc、new方法或者通过copy消息（接收到消息的对象会创建一个自身的副本）创建一个对象时，对象的保留计数器值被设置为1。要增加对象的保留计数器的值，可以给对象发送一条retain消息。要减少的话，可以给对象发送一条release消息。

当一个对象因其保留计数器归0而即将被销毁时，Objective-C会自动向对象发送一条dealloc消息。你可以在自己的对象中重写dealloc方法，这样就能释放掉已经分配的全部相关资源。一定不要直接调用dealloc方法，Objective-C会在需要销毁对象时自动调用它。

要获得保留计数器当前的值，可以发送retainCount消息。下面是retain、release和retainCount的方法声明。

```
- (id) retain;
- (oneway void) release;
- (NSUInteger) retainCount;
```

retain方法返回一个id类型的值。通过这种方式，可以在接收其他消息的同时进行retain调用，增加对象的保留计数器的值并要求对象完成某种操作。例如，[[car retain] setTire: tire atIndex:2];表示要求car对象将其保留计数器的值加1并执行setTire操作。

本章的第一个项目是RetainCount1，位于09.01 RetainCount-1项目文件夹内。以下程序创建了一个RetainTracker类的对象，该对象在初始化和销毁时调用了NSLog()函数。

```
@interface RetainTracker : NSObject
@end // RetainTracker

@implementation RetainTracker
 - (id) init
 {
  if (self = [super init]) { NSLog (@"init: Retain count of %d.",
     [self retainCount]);
  }
  return (self);
} // init

- (void) dealloc
{
   NSLog (@"dealloc called. Bye Bye.");
  [super dealloc];
```

```
} // dealloc
@end // RetainTracker
```

init方法遵循标准的Cocoa对象初始化方式，我们将在下一章中讨论这种方式。正如前面所讲，当对象的保留计数器的值归0时，将自动发送dealloc消息（dealloc方法也会被调用）。在我们的例子中，init和dealloc这两个方法使用NSLog()输出一条消息，表明它们被调用了。

在main()函数中我们创建了一个新的RetainTracker类的对象，并间接调用了由RetainTracker类定义的两个方法。当一个新的RetainTracker类的对象创建完毕后，就会向它发送retain消息或release消息，以增加或减少对象的保留计数器的值，以下是NSLog()函数有趣的输出结果。

```
int main (int argc, const char *argv[])
{
 RetainTracker *tracker = [RetainTracker new];
 // count: 1

[tracker retain]; // count: 2
NSLog (@"%d", [tracker retainCount]);

[tracker retain]; // count: 3
NSLog (@"%d", [tracker retainCount]);

[tracker release]; // count: 2
NSLog (@"%d", [tracker retainCount]);

[tracker release]; // count: 1
NSLog (@"%d", [tracker retainCount]);

[tracker retain]; // count 2
NSLog (@"%d", [tracker retainCount]);

[tracker release]; // count 1
NSLog (@"%d", [tracker retainCount]);

[tracker release]; // count: 0, dealloc it
return (0);
} // main
```

当然，在实际开发过程中，你不会像这样在一个函数中多次保留和释放对象。在其生命周期中，随着程序的运行，对象可能会像在本示例中一样，经历由程序中的不同调用而引起的一系列保留和释放行为。让我们运行该程序来看看保留计数器的值是如何变化的。

```
init: Retain count of 1.
2
3
2
1
2
1
dealloc called. Bye Bye.
```

因此，如果对一个对象使用了alloc、new或copy操作，释放该对象就能销毁它并收回它所占用的内存。

9.1.2　对象所有权

“等下，你不是说很难吗？这有什么大不了的？创建一个对象，然后使用它，释放它，内存管理也很容易嘛，听起来也不怎么复杂呀。”你肯定是这么想的。不过，当你开始考虑对象所有权（object ownership）这一概念时，内存管理就会变得复杂了。当我们说某个实体“拥有一个对象”时，就意味着该实体要负责确保对其拥有的对象进行清理。

如果一个对象内有指向其他对象的实例变量，则称该对象拥有这些对象。例如，在CarParts类中，car对象拥有其指向的engine和tire对象。同样，如果一个函数创建了一个对象，则称该函数拥有这个对象。在CarParts类中，main()函数创建了一个新的car对象，因此称main()函数拥有car对象。

当多个实体拥有某个特定对象时，对象的所有权关系就更加复杂了，这也是保留计数器的值大于1的原因。在RetainCount1程序的示例中，main()函数拥有RetainTracker类的对象，因此main()函数要负责清理该类的对象。

回忆一下Car类中的变量engine的存取方法：

```
- (void) setEngine: (Engine *) newEngine;
```

以及如何在main()函数中调用该方法：

```
Engine *engine = [Engine new];
[car setEngine: engine];
```

现在哪个实体拥有engine对象？是main()函数还是Car类？哪个实体负责确保当engine对象不再被使用时能够收到release消息？因为Car类正在使用engine对象，所以不可能是main()函数。因为main()函数随后可能还会使用engine对象，所以也不可能是Car类。

解决办法是让Car类保留engine对象，将engine对象的保留计数器的值增加到2。这是因为Car类和main()函数这两个实体都在使用engine对象。Car类应该在setEngine:方法中保留engine对象，而main()函数负责释放engine对象。然后，当Car类完成其任务时再释放engine对象（在其dealloc方法中），最后engine对象占用的资源被回收。

9.1.3　访问方法中的保留和释放

编写setEngine方法的第一个内存管理版本，可能如下所示。

```
- (void) setEngine: (Engine *) newEngine
{
 engine = [newEngine retain];

 // BAD CODE: do not steal. See fixed version below.
} // setEngine
```

可惜的是，这样做还远远不够。设想一下，在main()函数中如下所示的调用顺序会出现什么情况。

```
Engine *engine1 = [Engine new]; // count: 1
[car setEngine: engine1]; // count: 2
[engine1 release]; // count: 1

Engine *engine2 = [Engine new]; // count: 1
[car setEngine: engine2]; // count: 2
```

坏了，engine1对象出问题了：它的保留计数器的值仍然为1。main()函数已经释放了对engine1对象的引用，但是Car类一直没有释放engine1对象。现在engine1对象已经发生了泄漏，这肯定不是什么好事。第一个engine对象将会一直空转并消耗大量的内存。

下面是对编写setEngine:方法的另一种尝试：

```
- (void) setEngine: (Engine *) newEngine
{
 [engine release];
 engine = [newEngine retain];

// More BAD CODE: do not steal. Fixed version below.
} // setEngine
```

该例子修复了前一个例子中engine1对象泄漏的错误，但是当newEngine对象和原来的engine对象是同一个对象时，这段代码也会出问题。请考虑下面的情况：

```
Engine *engine = [Engine new]; // count: 1
Car *car1 = [Car new];
Car *car2 = [Car new];

[car1 setEngine: engine]; // count: 2
[engine release]; // count 1

[car2 setEngine: [car1 engine]]; // oops!
```

为什么会出现这样的问题？让我们来看一下哪里出了问题。[car1 engine]返回一个指向engine对象的指针，该对象的保留计数器的值为1。setEngine方法的第一行是[engine release]，该语句将engine对象的保留计数器的值归0，并释放掉engine对象。现在，newEngine和engine这两个实例变量都指向刚释放的内存区，这会引发错误。下面是编写setEngine的一种更好的方法。

```
- (void) setEngine: (Engine *) newEngine
{
 [newEngine retain];
 [engine release];
 engine = newEngine;

} // setEngine
```

如果首先保留新的engine对象，即使newEngine与engine是同一个对象，保留计数器的值也将先增加，然后立即减少。由于没有归0，engine对象意外地未被销毁，这样就不会引发错误了。在访问方法中，如果先保留新对象，然后再释放对象就不会出现问题了。

> **说明**　关于应该如何编写合理的访问方法，一直存在着不同的看法，各个邮件列表上基于半正则的争论仍然在进行。本节中给出的技术可以很好地完成工作，而且易于理解。但是，如果在其他人的代码中看到不同的访问方法管理技术，你也大可不必感到惊讶。

9.1.4　自动释放

内存管理是一个棘手的问题，前面你已经看到了我们在编写setter方法时遇到的各种细微的问题。接下来该解决另一个难题了。大家都知道，当不再使用对象时必须将其释放，但是在某些情况下弄清楚什么时候不再使用一个对象并不容易。观察下面这个description方法（该方法返回一个用来描述对象信息的NSString类型值）的例子。

```
- (NSString *) description
{
 NSString *description;
 description = [[NSString alloc]
  initWithFormat: @"I am %d years old", 4];

 return (description);
} // description
```

在这个例子中，我们使用alloc方法创建一个新的字符串实例（alloc方法将该对象的保留计数器的值设置为1），然后返回该字符串实例。请思考一下，哪个实体负责清理该字符串对象呢？

不可能是description方法。如果先释放description字符串对象再返回它，则保留计数器的值归0，对象马上会被销毁。

调用description方法的代码将返回的字符串赋在某个变量中，并在使用完毕后将其释放，不过像这样使用description方法会非常不方便。本来只需要一行代码，结果却写了3行。

```
NSString *desc = [someObject description];
NSLog (@"%@", desc);
[desc release];
```

应该还有更好的解决办法。你很幸运，接下来就是！

9.1.5　所有对象放入池中

Cocoa中有一个*自动释放池*（autorelease pool）的概念。你可能已经在Xcode生成的样本代码中见到过@autoreleasepool或NSAutoreleasePool。现在我们来看看自动释放池究竟是怎么一回事。

从名称上来推测，它大概是一个用来存放对象的池子（集合），并且能够自动释放。

NSObject类提供了一个叫做autorelease的方法：

```
- (id) autorelease;
```

该方法预先设定了一条会在未来某个时间发送的release消息，其返回值是接收这条消息的对象。这一特性与retain消息采用了相同的技术，使嵌套调用更加容易。当给一个对象发送autorelease消息时，实际上是将该对象添加到了自动释放池中。当自动释放池被销毁时，会向该池中的所有对象发送release消息。

> **说明** 自动释放池的概念并不神秘。你可以使用NSMutableArray来编写自己的自动释放池，以容纳对象并在dealloc方法中向池中的所有对象发送release消息，但是并无此必要，因为苹果公司已经替你完成了这项艰巨的任务。

现在我们可以编写一个能够很好地管理内存的description方法。

```
- (NSString *) description
{
 NSString *description;
 description = [[NSString alloc]
  initWithFormat: @"I am %d years old", 4];

 return ([description autorelease]);
} // description
```

你现在只编写一行这样的代码就够了：

```
NSLog (@"%@", [someObject description]);
```

因为description方法首先创建了一个新的字符串对象，然后自动释放该对象，最后将其返回给NSLog()函数，所以内存管理问题至此得到了圆满解决。由于description方法中的字符串对象是自动释放的，该对象暂时被放入了当前活动的自动释放池中，等到调用NSLog()函数的代码运行结束以后，自动释放池会被自动销毁。

9.1.6 自动释放池的销毁时间

自动释放池什么时候才能销毁，并向其包含的所有对象发送release消息？还有自动释放池应该什么时候创建？其实有两种方法可以创建一个自动释放池。

- 通过@autoreleasepool关键字。
- 通过NSAutoreleasePool对象。

在我们一直使用的Foundation库工具集中，创建和销毁自动释放池已经由@autorelease关键字完成。当你使用@autorelease{}时，所有在花括号里的代码都会被放入这个新池子里。如果你的程序运算是内存密集型的，你可以使用这种自动释放池。

有一点需要记住，任何在花括号里定义的变量在括号外就无法使用了。这就像是典型的C语言中的有效范围，比如说循环代码。

第二种更加明确的方法就是使用NSAutoreleasePool对象。如果你使用了这个方法，创建和释放NSAutoreleasePool对象之间的代码就会使用这个新的池子。

```
NSAutoreleasePool *pool;
pool = [NSAutoreleasePool new];
…
[pool release];
```

创建了一个自动释放池后，该池就会自动成为活动的池子。释放该池后，其保留计数器的值归0，然后该池被销毁。在销毁的过程中，该池将释放其包含的所有对象。

那么你应该使用哪一种方法呢？推荐使用关键字方法。它比对象方法更快，因为Objective-C语言创建和释放内存的能力远在我们之上。

使用AppKit时，Cocoa定期自动地为你创建和销毁自动释放池。通常是在程序处理完当前事件（如鼠标单击或键盘按下）以后执行这些操作。你可以使用任意数目的自动释放对象，当不再使用它们时，自动释放池将自动为你清理这些对象。

> **说明**　你可能在Xcode的自动生成代码中见过另一种销毁自动释放池中对象的方式：-drain方法。该方法只是清空自动释放池而不会销毁它，而且只适用于Mac OS X 10.4（Tiger）以后的操作系统版本。在自己编写（而不是由Xcode生成）的代码中，我们使用-release方法，因为该方法可以支持更古老的OS X版本。

9.1.7　自动释放池的工作流程

程序RetainRetainCount2展示了自动释放池的工作流程，它位于09.02 RetainCount-2项目文件夹内。该程序使用的RetainTracker类与我们在RetainTracker1中构建的RetainTracker类一样，在初始化和释放对象时会调用NSLog()函数。

RetainCount2的main()函数如下所示。

```
int main (int argc, const char *argv[])
{
 NSAutoreleasePool *pool;
 pool = [[NSAutoreleasePool alloc] init];

 RetainTracker *tracker;
 tracker = [RetainTracker new]; // count: 1

 [tracker retain]; // count: 2
 [tracker autorelease]; // count: still 2
 [tracker release]; // count: 1
 NSLog (@"releasing pool");
 [pool release];
 // gets nuked, sends release to tracker

 @autoreleasepool
  {
    RetainTracker *tracker2;
    tracker2 = [RetainTracker new]; // count: 1
```

```
        [tracker2 retain]; // count: 2
        [tracker2 autorelease]; // count: still 2
        [tracker2 release]; // count: 1

        NSLog (@"auto releasing pool");
      }

    return (0);
    } // main
```

首先，我们创建了一个自动释放池：

```
NSAutoreleasePool *pool;
pool = [[NSAutoreleasePool alloc] init];
```

现在每次向某对象发送autorelease消息，该对象都会被添加到这个自动释放池中，如下所示。

```
RetainTracker *tracker;
tracker = [RetainTracker new]; // count: 1
```

这时，创建了一个新的tracker对象，因为它在创建时接收了一条new消息，所以其保留计数器的值为1。

```
[tracker retain]; // count: 2
```

接下来，我们保留了该对象（仅仅是为了好玩和演示），于是该对象的保留计数器的值增加到了2。

```
[tracker autorelease]; // count: still 2
```

然后该对象被自动释放，但是其保留计数器的值仍保持不变，依旧为2。需要特别说明的是，我们之前创建的自动释放池中现在有一个引用指向了该对象。当自动释放池被销毁时，将向tracker对象发送一条release消息。

```
[tracker release]; // count: 1
```

之后释放该对象以抵消之前对它执行的保留操作。该对象的保留计数器的值仍大于0，所以仍处于活动状态。

```
NSLog (@"releasing pool");
[pool release];
// gets nuked, sends release to tracker
```

现在，我们销毁该自动释放池。NSAutoreleasePool是一个普通对象，与其他对象一样要遵从相同的内存管理规则。因为我们创建该自动释放池时向其发送了一条alloc消息，所以其保留计数器的值为1。这条release消息将其保留计数器的值减少为0，因此该自动释放池将被销毁，其dealloc方法被调用。

最后，main()函数返回0，表明全部操作成功执行：

```
return (0);
} // main
```

你能猜测一下输出结果会是什么样子吗？释放自动释放池之前的NSLog()函数和RetainTracker类的dealloc方法中的NSLog()函数，哪一个先被调用？

使用@autoreleasepool也能达到同样的目的，不过它并不需要分配或销毁自动释放池对象。

运行RetainCount2程序的输出结果如下：

```
init: Retain count of 1.
releasing pool
dealloc called. Bye Bye.
init: Retain count of 1.
auto releasing pool
dealloc called. Bye Bye.
```

你可能已经猜到，释放自动释放池之前的NSLog()函数先于RetainTracker类中的NSLog()函数被调用。

9.2　Cocoa 的内存管理规则

我们已经学习了各种方法：retain、release和autorelease。Cocoa有许多内存管理约定，它们都是一些很简单的规则，可应用于整个工具集内。

> **说明**　我们常常将这些规则复杂化，同样，忽略这些规则也是一种常犯的错误。如果你正在漫无目的地滥用retain和release方法以修正某些错误，那就说明你还没有真正掌握这些规则，你需要放慢速度，休息一下，然后再继续阅读。

这些规则如下所示。

- 当你使用new、alloc或copy方法创建一个对象时，该对象的保留计数器的值为1。当不再使用该对象时，你应该向该对象发送一条release或autorelease消息。这样，该对象将在其使用寿命结束时被销毁。
- 当你通过其他方法获得一个对象时，假设该对象的保留计数器的值为1，而且已经被设置为自动释放，那么你不需要执行任何操作来确保该对象得到清理。如果你打算在一段时间内拥有该对象，则需要保留它并确保在操作完成时释放它。
- 如果你保留了某个对象，就需要（最终）释放或自动释放该对象。必须保持retain方法和release方法的使用次数相等。

只有这三条规则，就这么简单。

“如果我使用了new、alloc或copy方法获得一个对象，就释放或自动释放该对象。”只要你记住了这条规则，就不用再担心内存释放的问题了。

无论什么时候拥有一个对象，有两件事必须弄清楚：怎样获得该对象的？打算拥有多长时间？（参见表9-1）

在下一节中我们将详细讨论临时使用的对象和长期使用的对象。

表9-1 内存理规则

获得途径	临时对象	拥有对象
alloc/new/copy	不再使用时释放对象	在dealloc方法中释放对象
其他方法	不需要执行任何操作	获得对象时保留，在dealloc方法中释放对象

9.2.1 临时对象

下面我们来看看一些常用的内存管理生命周期。你正在代码中使用某个对象，但是并未打算长期拥有该对象。如果你是用new、alloc或copy方法获得的这个对象，就需要安排好该对象的内存释放，通常使用release消息来实现。

```
NSMutableArray *array;
array = [[NSMutableArray alloc] init]; // count: 1
// use the array
[array release]; // count: 0
```

如果你使用其他方法获得一个对象，比如arrayWithCapacity:方法，则不需要去关心如何销毁该对象。

```
NSMutableArray *array;
array = [NSMutableArray arrayWithCapacity: 17];
// count: 1, autoreleased
// use the array
```

arrayWithCapacity:方法与alloc、new、copy这三个方法不同，因此可以假设该对象被返回时保留计数器的值为1且已经被设置为自动释放。当自动释放池被销毁时，向array对象发送release消息，该对象的保留计数器的值归0，其占用的内存被回收。

使用NSColor类对象的部分代码如下：

```
NSColor *color;
color = [NSColor blueColor];
// use the color
```

blueColor方法也不属于alloc、new、copy这三个方法，因此可以假设该对象的保留计数器的值为1并且已经被设置为自动释放。blueColor方法返回一个全局单例（singleton）对象——每个需要访问它的程序都可以共享的单一对象，这个对象永远不会被销毁，不过你不需要关心其实现细节，只需知道不需要自己手动来释放color。

9.2.2 拥有对象

通常，你可能希望在多段代码中一直拥有某个对象。典型的方法是，把它们加入到诸如NSArray或NSDictionary等集合中，作为其他对象的实例变量来使用，或作为全局变量来使用（比较罕见）。

如果你使用了new、alloc或copy方法获得了一个对象，则不需要执行任何其他操作。该对象的保留计数器的值为1，因此它将一直存在着，你只需确保在拥有该对象的dealloc方法中释放它即可。

```
- (void) doStuff
{
```

```
 // flonkArray is an instance variable
 flonkArray = [NSMutableArray new]; // count: 1
} // doStuff

- (void) dealloc
{
 [flonkArray release]; // count: 0
 [super dealloc];
} // dealloc
```

如果你使用除alloc、new或copy以外的方法获得了一个对象，需要记得保留该对象。如果你编写的是GUI应用程序，要考虑到事件循环。你需要保留自动释放的对象，以便这些对象在当前的事件循环结束以后仍能继续存在。

什么是事件循环呢？一个典型的图形应用程序往往花费许多时间等待用户操作。在控制程序运行的人慢腾腾地作出决定（比如点击鼠标或按下某个键）以前，程序将一直处于空闲状态。当发生这样的事件时，程序将被唤醒并开始工作，执行必要的操作以响应这一事件。在处理完这一事件后，程序返回到休眠状态并等待下一个事件发生。为了降低程序的内存空间占用，Cocoa会在程序开始处理事件之前创建一个自动释放池，并在事件处理结束后销毁。这样可以尽量减少累积的临时对象的数量。

当使用自动释放对象时，前面的方法可以按如下形式重写。

```
- (void) doStuff
{
 // flonkArray is an instance variable
 flonkArray = [NSMutableArray arrayWithCapacity: 17];
 // count: 1, autoreleased
 [flonkArray retain]; // count: 2, 1 autorelease
} // doStuff
- (void) dealloc
{
 [flonkArray release]; // count: 0
 [super dealloc];
} // dealloc
```

在当前事件循环结束（如果这是一个GUI程序）或自动释放池被销毁时，flonkArray对象会接收到一条release消息，因而其保留计数器的值从2减少到1。因为其保留计数器的值大于0，所以该对象将继续存在。因此，我们需要在自己的dealloc方法中释放它，以便它被清理掉。如果我们在doStuff方法中没有向flonkArray对象发送retain消息，则flonkArray对象将会被意外地销毁。

请记住，自动释放池被清理的时间是完全确定的：要么是在代码中你自己手动销毁，要么是使用AppKit时在事件循环结束时销毁。你不必担心程序会随机地销毁自动释放池，也不必保留使用的每一个对象，因为在调用函数的过程中自动释放池不会被销毁。

清理自动释放池

有时自动释放池未能按照你想要的方式进行清理。Cocoa邮件列表的一个常见问题就是："虽然我已经自动释放了使用的所有对象，但是程序占用的内存却一直增长。"下面的代码通常

是引起这类问题的原因。

```
int i;
for (i = 0; i < 1000000; i++) {
id object = [someArray objectAtIndex: i];
NSString *desc = [object description];
// and do something with the description
}
```

该程序执行了一个循环，在大量的迭代中每次都会生成一个（也可能是两个或十个）自动释放对象。请记住，自动释放池的销毁时间是完全确定的，它在循环执行过程中不会被销毁。这个循环创建了100万个description字符串对象，所有这些对象都被放进当前的自动释放池中，因此就产生了100万个闲置的字符串。这100万个字符串对象一直存在，只有当自动释放池被销毁时才最终得以释放。

解决这类问题的方法是在循环中创建自己的自动释放池。这样一来，循环每执行1000次，就会销毁当前的并创建一个新的。代码如下，新增的代码以粗体表示。

```
NSAutoreleasePool *pool;
pool = [[NSAutoreleasePool alloc] init];
int i;
for (i = 0; i < 1000000; i++) {
  id object = [someArray objectAtIndex: i];
  NSString *desc = [object descrption];
  // and do something with the description
  if (i % 1000 == 0) {
  [pool release];
  pool = [[NSAutoreleasePool alloc] init];
  }
}
[pool release];
```

循环每执行1000次，新的自动释放池就会被销毁，同时有一个更新的自动释放池被创建。现在，自动释放池中同时存在的description字符串不会超过1000个，因此程序的内存占用不会持续增加。自动释放池的分配和销毁操作代价很小，因此你甚至可以在循环的每次迭代中创建一个新的自动释放池。

自动释放池以栈的形式实现：当你创建了一个新的自动释放池时，它就被添加到栈顶。接收autorelease消息的对象将被放入最顶端的自动释放池中。如果将一个对象放入一个自动释放池中，然后创建一个新的自动释放池，再销毁该新建的自动释放池，则这个自动释放池对象仍将存在，因为容纳该对象的自动释放池仍然存在。

以上方法在这里无法运行，因为它的括号数量并没有前后对等。

9.2.3 垃圾回收

Objective-C 2.0引入了自动内存管理机制，也称垃圾回收。熟悉Java或Python等语言的程序员应该非常熟悉垃圾回收的概念。对于已经创建和使用的对象，当你忘记清理时，系统会自动识别

哪些对象仍在使用，哪些对象可以回收。启用垃圾回收功能非常简单，但它是一个可选择是否启用的功能。只需要转到项目信息窗口的Build Settings选项卡，并选择Required[-fobjc-gc-only]选项即可，如图9-1所示。

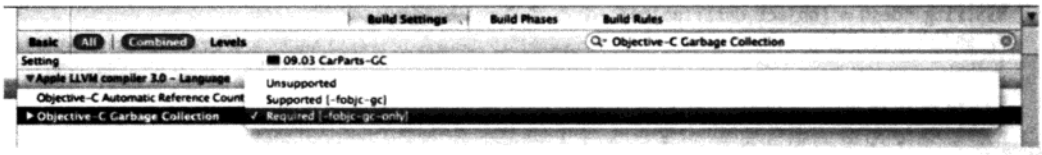


图9-1　启用垃圾回收功能

> **说明**　-fobjc-gc选项是针对既支持垃圾回收又支持对象的保留和释放的代码，例如在两种环境下都能使用的库代码。

启用垃圾回收以后，平常的内存管理命令全都变成了空操作指令，说得直白点就是它们不执行任何操作。

Objective-C的垃圾回收器是一代新型的垃圾回收器。与那些面世已久的对象相比，新创建的对象更可能被当成垃圾。垃圾回收器定期检查变量和对象并且跟踪它们之间的指针，当发现没有任何变量指向某个对象时，就将该对象视为应该丢弃的垃圾。最糟糕的事情莫过于让指针指向一个不再使用的对象。因此，如果你在一个实例变量中指向某个对象（回忆一下第5章中的内容），一定要将该实例变量赋值为nil，以取消对该对象的引用并告知垃圾回收器该对象可以清理了。

与自动释放池一样，垃圾回收器也是在事件循环结束时触发的。当然，如果编写的不是GUI程序，你也可以自己触发垃圾回收器，不过这超出了本章的讨论范围。

有了垃圾回收，你就不必再担心内存管理问题了。当使用从malloc函数（或利用Core Foundation库中的对象）回收的内存时，会有一些细微的差别，但是我们不会讨论这些过于晦涩难懂的内容。现在你可以只考虑对象的创建，无需关心它们的释放问题。接下来我们将继续讨论垃圾回收问题。

请注意垃圾回收功能只支持OS X应用开发，无法用在iOS应用程序上。实际上在iOS开发中，苹果公司建议你不要在自己的代码中使用autorelease方法，也不要使用会返回自动释放对象的一些便利的方法。

一般这些便利的方法都会返回一个新对象的类方法。比如说NSString，所有以stringWith开头的方法都是便利方法。

9.2.4　自动引用计数

在iOS中无法使用垃圾回收是怎么一回事？主要原因是你无法知道垃圾回收器什么时候会起作用。就像在现实生活中，你可能知道周一是垃圾回收日，但是不知道精确的时间。假如你正要

出门的时候垃圾车到了怎么办？垃圾回收机制会对移动设备的可用性产生非常不利的影响，因为移动设备比电脑更私人化，资源更少。用户可不想在玩游戏或打电话的时候因为系统突然进行内存清理而卡住。

苹果公司的解决方案被称为自动引用计数（automatic reference counting，ARC）。顾名思义，ARC会追踪你的对象并决定哪一个仍会使用而哪一个不会再用到，就好像你有了一位负责内存管理的管家或私人助理。如果你启用了ARC，只管像平常那样按需分配并使用对象，编译器会帮你插入retain和release语句，无需你自己动手。

ARC不是垃圾回收器。我们已经讨论过了，垃圾回收器在运行时工作，通过返回的代码来定期检查对象。与此相反，ARC是在编译时进行工作的。它在代码中插入了合适的retain和release语句，就好像是你自己手动写好了所有的内存管理代码。不过，是编译器替你完成了内存管理的工作。程序在运行的时候，retain和release会像往常一样发挥作用。系统不会知道也不会关心这些命令是你写的还是编译器写的。

ARC是一个可选的功能，也就是说你必须明确地启用或禁用它。

以下是编写ARC代码所需的条件：

- Xcode 4.2以上的版本；
- Apple LLVM 3.0以上版本的编译器；
- OS X 10.7以上版本的系统。

以下是可以运行ARC代码的设备必须满足的条件：

- iOS 4.0以上的移动设备或OS X 10.6以上版本的64位系统的电脑；
- 归零弱引用（zeroing weak reference，本章后面会介绍）需要iOS 5.0或OS X 10.7以上版本的系统。

> **说明** ARC只对可保留的对象指针（ROPs）有效。可保留的对象指针主要有以下三种：
> (1) 代码块指针
> (2) Objective-C对象指针
> (3) 通过__attribute__((NSObject))类型定义的指针
> 本章后面会详细讲解可保留的对象指针，尤其是在介绍Xcode的ARC转换工具的时候。所有其他的指针类型，比如char *和CF对象（例如CFStringRef）都不支持ARC特性。如果你使用的指针不支持ARC，那么你将不得不亲自手动管理它们。这样也没有问题，因为ARC可以与手动的内存管理共同发挥作用。

如果你想要在代码中使用ARC，必须满足以下三个条件：

- 能够确定哪些对象需要进行内存管理；
- 能够表明如何去管理对象；
- 有可行的办法传递对象的所有权。

我们来讲讲这三个条件。第一个条件是对象的最上层集合知道如何去管理它的子对象。比方

说你有一个通过malloc:方法创建的字符串数组：

```
NSString **myString;
myString = malloc(10 * sizeof(NSString *));
```

这段代码创建了一个指向10个字符串的C型数组。因为C型数组不是可保留的对象，所以你无法在这个结构体里使用ARC特性。

第二个条件是你必须能够对某个对象的保留计数器的值进行加1或减1的操作。也就是说所有NSObject类的子类都能进行内存管理。这包括了大部分你需要管理的对象。

第三个条件是在传递对象的时候，你的程序需要能够在调用者和接收者（后面会详细介绍）之间传递所有权。

看到所有这些要求后，你可能会问自己：我真的需要使用ARC特性吗？答案是肯定的，我们保证，从长远看它可以帮你减少烦恼，节省时间。

1. 有时用Weak会好一些

本章前面已经讨论过，当用指针指向某个对象时，你可以管理它的内存（通过retain和release），也可以不管理。如果你管理了，就拥有对这个对象的*强引用*（strong reference）。如果你没有管理，那么你拥有的则是（估计你也该猜到了）*弱引用*（weak reference）。比方说，对属性使用了assign特性，你便创建了一个弱引用。

为什么会有弱引用？因为它们有助于处理保留循环（retain cycle），我们马上就会讲到。

假设你拥有一个由其他对象创建并且保留计数器的值为1的对象A（如图9-2所示）。对象A创建了保留计数器的值为1的对象B并将其作为子对象（如图9-3所示）。对象B需要能够访问它的父对象。这是一个很常见的Objective-C程序设计模式。

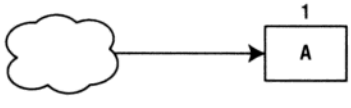


图9-2　对象A的保留计数器的值为1

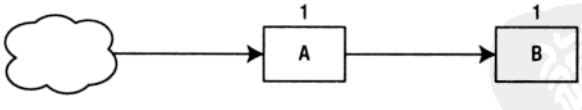


图9-3　对象B为对象A的子对象，保留计数器的值也为1

在这个示例中，因为对象A创建了对象B，所以对象A拥有一个指向对象B的强引用。现在，如果对象B有一个指向对象A的强引用，那么对象A的保留计数器的值会增加到2（如图9-4所示）。

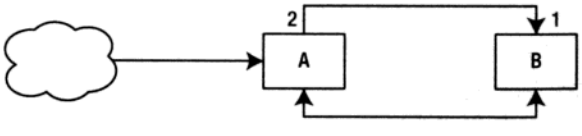


图9-4　对象B拥有一个指向对象A的强引用，并且对象A的保留计数器的值增加到了2

任何事物都有结束的那一天，当对象A的拥有者不再需要它的时候，就会向对象A发送release消息，这样会让对象A的保留计数器的值减少到1。

不过由对象A所创建的对象B的保留计数器的值仍为1。因为两个对象的保留计数器的值都不为0，所以它们都没有被释放掉。这就是一个经典的内存泄漏：程序无法访问到这些对象，但它们仍占用着内存容量（如图9-5所示）。

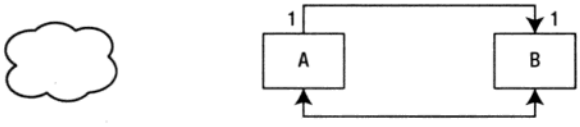


图9-5 经典的内存泄漏

为了解决这个问题，可以使用弱引用。使用assign来获取对象B指向对象A的引用。由于是弱引用，保留计数器的值不会增加，所以当对象A的拥有者释放它的时候，它的保留计数器就会变成0，它也会释放对象B，这样AB两个对象都会交出之前占用的内存（如图9-6所示）。怎么样？很完美吧。

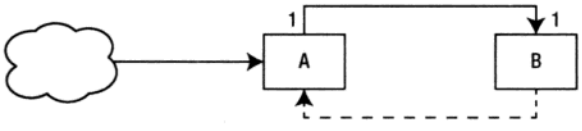


图9-6 对象B通过弱引用指向对象A

不错，但还算不上完美。在你拥有3个对象的时候（分别称为对象ABC好了），假设对象A通过强引用指向对象B，而对象C通过弱引用指向了对象B（如图9-7所示）。

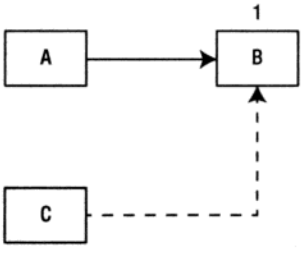


图9-7 两个对象指向了对象B，一个强一个弱

如果对象A释放了对象B，那么对象C仍将拥有指向对象B的弱引用，但这个引用已经失效了，直接使用它会导致问题，因为指向的地方已经没有有效值了（如图9-8所示）。

解决方法就是让对象自己去清空弱引用的对象。这种特殊的弱引用被称为*归零弱引用*（zeroing weak reference），因为在指向的对象释放之后，这些弱引用就会被设置为零（即nil），就可以像平常的指向nil值的指针一样被处理（如图9-9所示）。归零弱引用只在iOS 5和OS X 10.7以上的版本中有效。

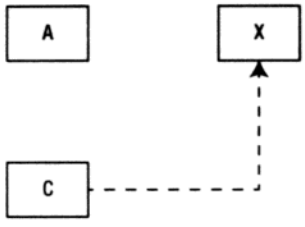


图9-8　对象C仍对相关对象存在引用，这很不好

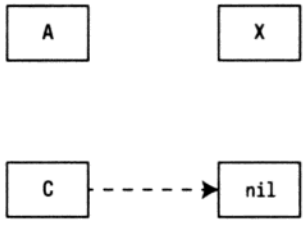


图9-9　对象C的归零弱引用会在指向的值失效的时候自动转换成nil

如果想要使用归零弱引用，必须明确地声明它们。有两种方式可以声明归零弱引用：声明变量时使用__weak关键字或对属性使用weak特性。

```
__weak NSString *myString;
@property(weak) NSString *myString;
```

如果你想在不支持弱引用的旧系统上使用ARC应该怎么办呢？苹果公司提供了__unsafe_unretained关键字和unsafe_unretained特性，它们会告诉ARC这个特殊的引用是弱引用。

使用ARC的时候有两种命名规则需要注意：

- 属性名称不能以new开头，比如说@property NSString *newString是不被允许的。
- 属性不能只有一个read-only而没有内存管理特性。如果你没有启用ARC，可以使用@property (readonly) NSString *title语句，但如果你启用了ARC功能，就必须指定由谁来管理内存。因为默认的特性是assign，所以你可以进行一个简单的修复，使用unsafe_unretained就可以了。

同样强引用也有自己的__strong关键字和strong特性。需要注意，内存管理的关键字和特性是不能一起使用的，两者相互排斥。

我们现在准备让CarParts程序支持ARC。完成之后你会发现代码比之前更少了。我们的座右铭或者说所有程序员的座右铭就是“少即多”（less is more）。

2. 一辆新车

Xcode提供了一个简单的步骤可以把我们已有的项目转换成支持ARC的。开始之前，必须确保垃圾回收机制没有启用，垃圾回收和ARC是无法一同使用的。如果你没有禁用垃圾回收功能就进行项目转换的话，将会看到如图9-10所示的警告。

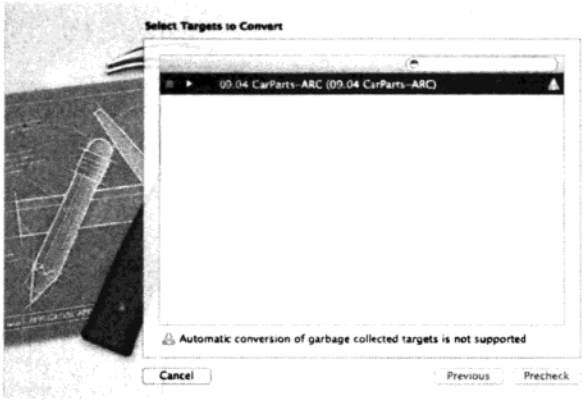


图9-10 转换项目前必须禁用垃圾回收功能

让我们来看看如何禁用垃圾回收机制。第一步是选中你想要禁用垃圾回收功能的目标（target）。如果你的项目还没有打开，请打开它。在导航栏里选中项目，便会在编辑区内看到项目编辑器，项目名称会显示在顶端，目标名称会显示在下面（如图9-11所示）。

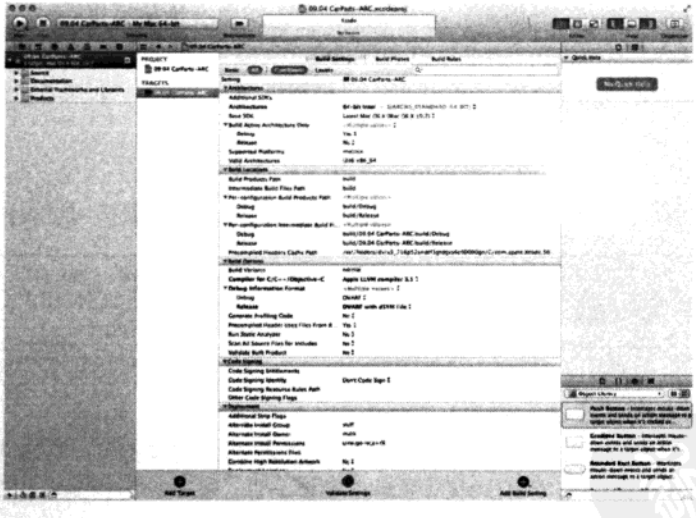


图9-11 在Xcode中打开项目

现在，选择需要禁用垃圾回收的目标，然后点击编辑器窗格顶端的Build Settings选项卡，你将会看到有非常多的设置项。你可以浏览这些条目并寻找Objective-C Garbage Collection这一项，但还有一种更好的方法。

苹果公司知道你很难找到所需的设置项，所以在Build Settings选项卡下，你会看到一个便利的搜索框。输入文字garbage collection，搜索框就会筛选出与垃圾回收相关的条目出来。这样太好了！现在你会看到两条设置项（如图9-12所示）。

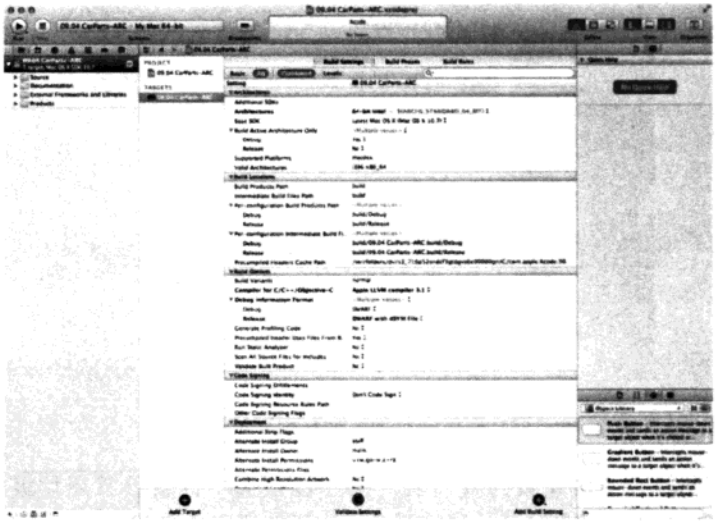


图9-12　设置垃圾回收

找到Objective-C Garbage Collection这一项后，弹出菜单会为你提供3个选择：

- Unsuppported（不支持）：如果你想令项目支持ARC，则必须选择这一项。当然，如果你因为某些原因不想启用垃圾回收机制，也可以选择此项。
- Supported（支持）：你的应用程序可以使用垃圾回收功能。
- Required（需要）：你的应用程序必须使用垃圾回收功能。

我们需要选择第一项：Unsupported（如图9-13所示），这样就能禁用垃圾回收机制了。

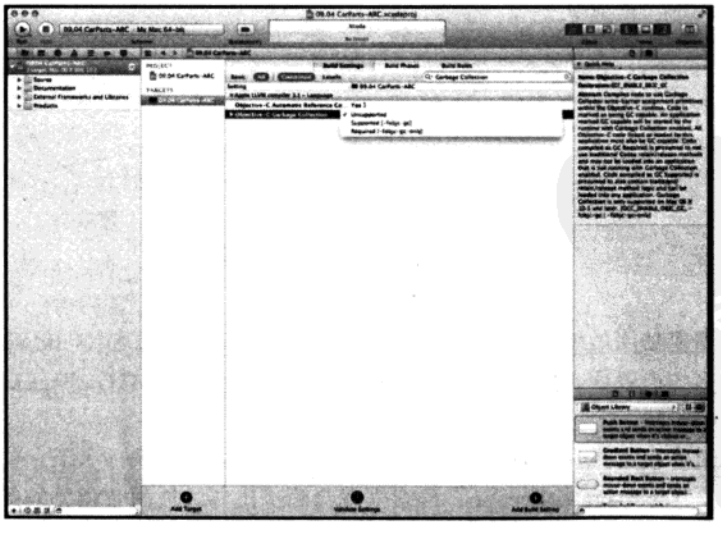


图9-13　选择Unsupported以禁用垃圾回收

转换过程一共需要两步。首先必须确保你的代码能符合ARC的需求，然后执行转换操作。

> **说明** ARC转换是一个单程的操作。一旦你转换成ARC版本，就不可以恢复了。

选择想要转换的目标，然后点击Edit ➤ Refactor ➤ Convert to Objective-C ARC，就可以开始转换过程了（如图9-14所示）。

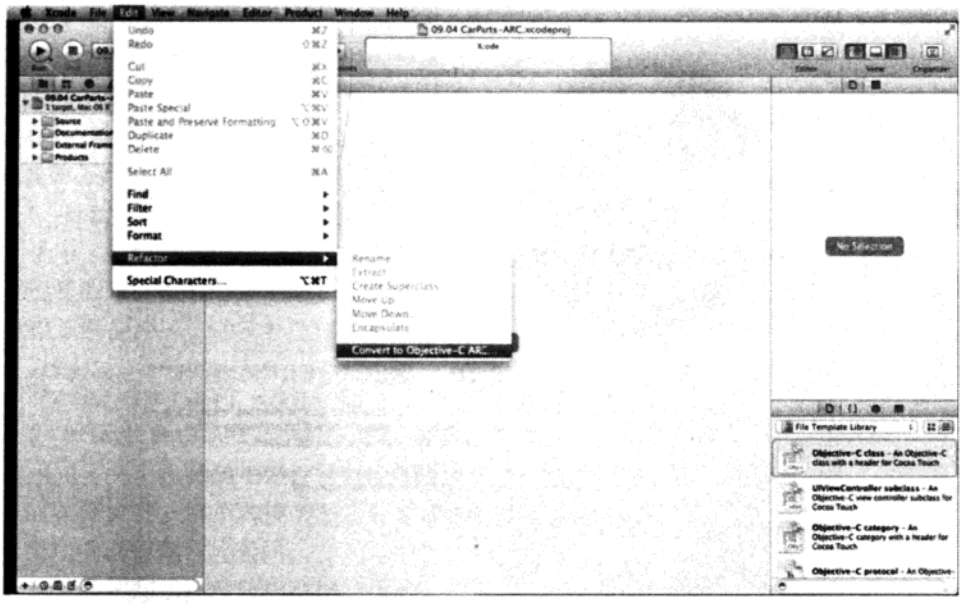


图9-14 通过Edit菜单选择ARC转换选项

因为ARC是基于文件进行工作的，所以你可以选择在一个项目中经过ARC转换和未经过转换的文件共存。

接下来，你将会看到想要转换的目标列表（如图9-15所示）。如果你的目标是依赖于其他目标的，你可以选择逐步进行这些步骤。举个例子，你先从框架和库开始着手，完成之后再去处理使用到它们的目标。

当你选择了需要转换的目标后，默认会选中所有的实现文件。如果有些文件在其他项目中也会用到，所以你不想转换的话，可以只选中想要转换的文件。选完之后请点击Precheck按钮。预检验步骤完成后，就可以进行实际的转换了（终于要开始正戏了），如图9-16所示。

阅读完帮助引导信息后请点击Next按钮，这样你的项目就会转换成可使用ARC特性的了。在这一步中，Xcode会找到所有符合ARC条件的文件并按照需求对源代码进行转换。

当这一步骤完成后，Xcode会展示源代码比较视图，在上面你可以看到每个文件在转换前和转换后的样子，并决定是否接受这些变化。

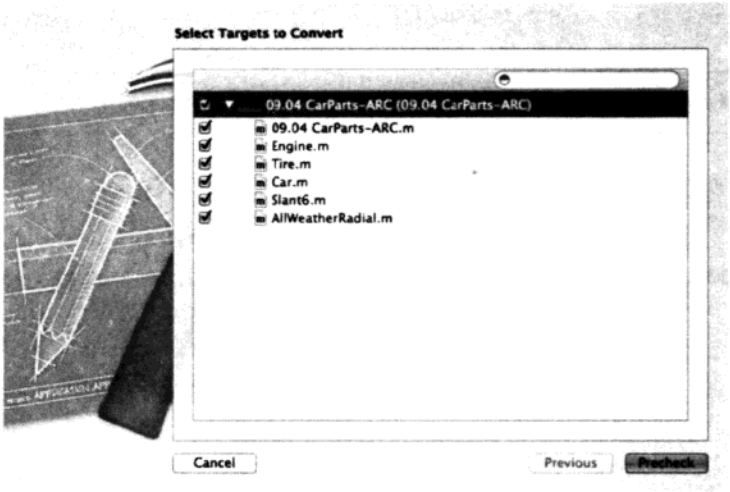


图9-15　选择你想要转换的文件

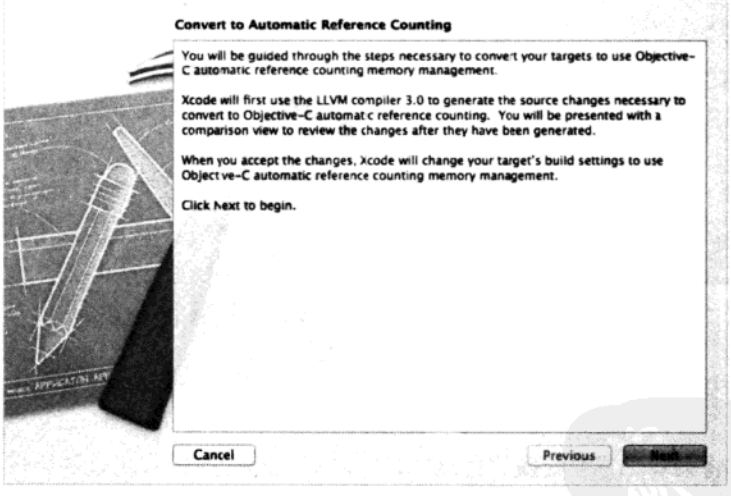


图9-16　ARC转换的引导界面

马上就要结束了，不过项目还没有完成转换。这些更改后的文件只是你原来文件的副本，你可以改变主意并退出转换过程（我们之前提到过这步如果完成了就无法撤销）。保存这些文件的副本后，整个过程就全部完成了。

这些都不难，不过我们必须告诉你这个示例只是理想状况——所有东西都很完美而且是自动执行的。但是现实操作并不会一直这么轻松。转换器只对源代码有效，它不会影响注释内容，而且目前的版本是无法读取你的想法和意图的。如果转换器遇到了难以理解的内容，它会标出来，你必须亲自解决问题，转换器才能继续进行工作。

3. 拥有者权限

我们之前讨论过指针支持ARC的一个条件是必须是可保留对象指针（ROP）。这意味着，你不能简单地将一个ROP表示成不可保留对象指针（non-ROP），因为指针的所有权会移交。来看下面的代码。

```
NSString *theString = @"Learn Objective-C";
CFStringRef cfString = (CFStringRef)theString;
```

如果你看过很多Objective-C代码，可能见过类似的结构。它在做什么？theString指针是一个ROP，而另一个CFStringRef则不是。为了让ARC便于工作，需要告诉编译器哪个对象是指针的拥有者。为此可以使用一种被称为桥接转换（bridged cast）的C语言技术。这是一个标准的C语言类型转换，不过使用的是其他关键字：__bridge、__bridge_retained和__bridge_transfer。术语bridge指的是使用不同的数据类型达到同一目的的能力，而不是当你陷入ARC转换困境时帮你脱离困境的工具。以下是对三种桥接转换类型的详细介绍。

- (__bridge类型)操作符：这种类型的转换会传递指针但不会传递它的所有权。在前面的例子中，操作符是theString，而类型是CFStringRef。如果你使用了这个关键字，则一个指针是ROP，而另一个不是。在这种情况下指针的所有权仍会留在操作符上。

  以下是使用了这种转换类型的代码示例：

  ```
  cfString = (__bridge CFStringRef)theString;
  cfString 接收了指令，但指针的所有权仍然由 theString 保留。
  ```

- (__bridge_retained类型)操作符：使用这种类型，所有权会转移到non-ROP上。与上一个相同，一个指针是ROP，另一个则不是。因为ARC只会注意到ROP，所以你要在不用的时候释放它。这个转换类型会给对象的保留计数器加1，所以你必须要让它减1，这与标准的内存管理方式相同。

  以下是使用了这种转换类型的代码示例：

  ```
  cfString = (__bridge_retained CFStringRef)theString
  ```

  在这个示例中，cfString字符串拥有指针并且它的保留计数器加1。你要使用retain和release来管理它的内存。

- (__bridge_transfer类型)操作符：这种转换类型与上一个相反，它把所有权转交给ROP。

  在这个示例中，ARC拥有对象并能确保它会像其他ARC对象一样得到释放。

另一个限制是结构体（struct）和集合体（union）不能使用ROP作为成员，因此下面这样的代码是不被允许的。

```
// Bad code. Do not steal or sell.
struct {
  int32_t foo;
  char *bar;
  NSString *baz;
} MyStruct;
```

你可以通过使用void *和桥接转换来解决这个问题。如果想要分配并获取字符串，可以使用如下代码：

```
struct {
  int32_t foo;
  char *bar;
  void *baz;
} MyStruct;
MyStruct.baz = (__bridge_retained void *)theString;
NSString *myString = (__bridge_transfer NSString *)MyStruct.baz;
```

如你所见，在两端的ARC代码并不总是自动编写的。这个主题相当深奥，需要花很长时间去学习。最后我们列出了不能对ARC管理的对象调用的管理方法：

- retain
- retainCount
- release
- autorelease
- dealloc

因为你有时需要释放不支持ARC的对象或执行其他清理操作，所以仍要实现dealloc方法，但是不能直接调用[super dealloc]。

以下是不能对ARC对象进行重写的方法：

- retain
- retainCount
- release
- autorelease

9.3　异常

什么是异常？异常就是意外事件，比如数组溢出，因为程序不知道怎么处理就会扰乱程序流程。

当发生这种情况时，程序可以创建一个异常对象，让它在运行时系统中计算出接下来该怎么做。Cocoa中使用NSException类来表示异常。Cocoa要求所有的异常必须是NSException类型的异常，虽然你可以通过其他对象来抛出异常，但Cocoa并不会处理它们。此外，你也可以创建NSException的子类来作为你自己的异常。

> **说明**　异常处理的真正目的是处理程序中生成的错误。Cocoa框架处理错误的方式通常是退出程序，这不是你想要的方式。为了找到出错的原因，你应该抛出并捕捉代码中的异常。

如果想要支持异常特性，请确保-fobj-exceptions项被打开。可以在Xcode中启用Enable Objective-C Exceptions项（如图9-17所示）。

在运行时系统中创建并处理异常的行为被称为抛出（throwing）异常，或者说是提出（raising）异常。需要注意NSException拥有一些以raise开头的方法名。有些是类方法，而raise本身是实例方法。

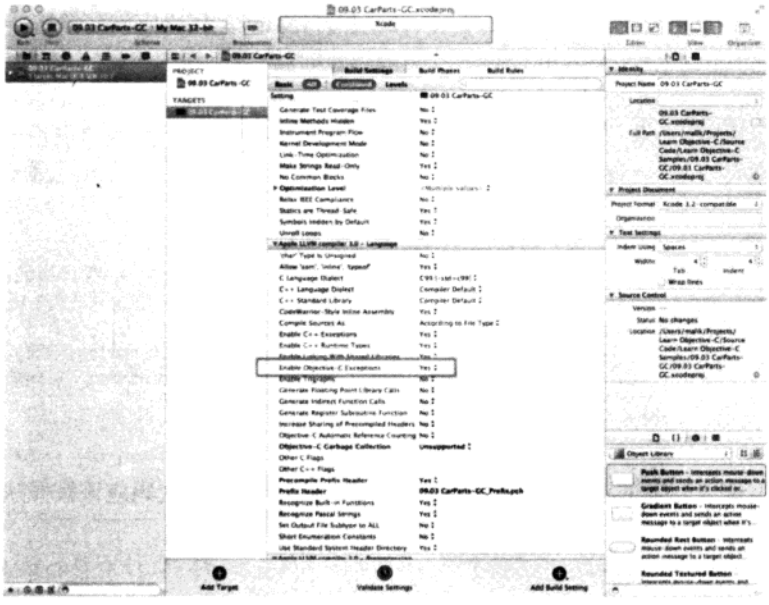


图9-17 启用异常特性

处理被抛出的异常的行为被称为捕捉（catching）异常。

> **说明** 如果一个异常被抛出但没有被捕捉到，程序会在异常断点处停止运行并通知有这个异常。

9.3.1 与异常有关的关键字

异常的所有关键字都是以@开头的。以下是它们的各自作用。

- ❑ @try：定义用来测试的代码块以决定是否要抛出异常。
- ❑ @catch()：定义用来处理已抛出异常的代码块。接收一个参数，通常是NSException类型，但也可能是其他类型。
- ❑ @finally：定义无论是否有抛出异常都会执行代码块，这段代码总是会执行的。
- ❑ @throw：抛出异常。

> **说明** 之前提到过除了NSException实例以外你也可以用其他对象来抛出异常，但要尽量避免这么做，因为并不是所有的Cocoa框架都会捕捉其他对象抛出的异常。为了确保Cocoa能够正常处理异常，你应该只用NSException对象来抛出异常。就像俗话说的：你没有必要让所有的人都满意，只需让你在乎的人满意就行了。

你通常会在一个结构中同时使用@try、@catch和@finally。代码应该会像这样：

```
@try
{
    // code you want to execute that might throw an exception.
}
@catch (NSException *exception)
{
    // code to execute that handles exception
}
@finally
{
    // code that will always be executed. Typically for cleanup.
}
```

9.3.2　捕捉不同类型的异常

你可以根据需要处理的异常类型使用多个@catch代码块。处理代码应该按照从具体到抽象的顺序排序，并在最后使用一个通用的处理代码。

```
@try{
} @catch (MyCustomException *custom) {
} @catch (NSException *exception) {
} @catch (id value) {
} @finally {
}
```

> **说明**　C语言程序员经常会在异常处理代码中使用setjmp和longjmp语句。你不能使用setjmp和longjmp来跳出@try代码块，但可以使用goto和return语句退出异常处理代码。

9.3.3　抛出异常

当程序检测到了异常，就必须向处理它的代码块（有时叫做异常处理代码）报告这个异常。之前讲过，通知异常的过程被称为抛出（或提出）异常。

程序会创建一个NSException实例来抛出异常，并会使用以下两种技术之一：

- 使用 "@throw异常名;" 语句来抛出异常；
- 向某个NSException对象发送raise消息。

举个例子，我们创建了一个异常：

```
NSException *theException = [NSException exceptionWithName: …];
```

要抛出这个异常可以用这个语句

```
@throw theException;
```

或者用

```
[theException raise];
```

两种方法都可以使用，但不要两种都使用。两种方法的区别是raise只对NSException对象有效，而@throw也可以用在其他对象上。

你通常会在异常处理代码中抛出异常。代码可以通过再发送一次raise消息或使用@throw关键字来通知异常。

```
@try
 {
   NSException *e = …;
   @throw e;
 }
 @catch (NSException *e) {
    @throw; // rethrows e.
}
```

> **说明** 在@catch异常处理代码中，你可以重复抛出异常而无需指定异常对象。

与当前@catch异常处理代码相关的@finally代码块会在@throw引发下一个异常处理调用之前执行代码，因为@finally是在@throw发生之前调用的。

Objective-C的异常机制与C++的异常机制兼容。

> **说明** 在Objective-C中的异常会对程序资源有影响。对一般流程你不能使用异常或简单地标记错误。虽然用@try建立异常不会产生消耗，但捕捉异常会消耗大量资源并影响程序运行的速度。

9.3.4 异常也需要内存管理

如果代码中有异常，内存管理执行起来会比较复杂。来看一段简单的代码：

```
- (void)mySimpleMethod
{
    NSDictionary *dictionary = [[NSDictionary alloc] initWith….];
    [self processDictionary:dictionary];
    [dictionary release];
}
```

现在假设processDictionary抛出了一个异常。程序从这个方法中跳出并寻找异常处理代码。由于现在方法已经退出来了，所以字典对象并没有被释放，于是就会出现内存泄漏。

一种简单的解决办法就是使用@try和@finally代码块，因为@finally总是会执行的（之前提到过），所以它可以在里面进行清理工作。

```
- (void)mySimpleMethod
{
```

```
    NSDictionary *dictionary = [[NSDictionary alloc] initWith….];
    @try {
     [self processDictionary:dictionary];
    }
    @finally {
    [dictionary release];
    }
}
```

这种方式也可以用在C语言类型的内存管理上，比如malloc和free。

9.3.5 异常和自动释放池

异常处理有时会遇到异常对象被自动释放的小问题。因为你不知道该什么时候释放，所以异常几乎总是作为自动释放对象而创建。当自动释放池销毁了之后，自动释放池中所有的对象也会被销毁，包括异常。观察一下这段代码，它看起来没有问题。

```
- (void)myMethod
{
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  NSDictionary *myDictionary =
    [[NSDictionary alloc] initWithObjectsAndKeys:@"asdfads", nil];
  @try {
    [self processDictionary:myDictionary];
  } @catch (NSException *e) {
    @throw;
  } @finally {
    [pool release];
  }
}
```

这看起来是段正确的代码：创建了对象，而且释放了它们，因为我们是习惯良好的程序员。但别高兴太早，仔细看看。异常处理有一个问题。之前说过，我们可以在@catch代码块中再次抛出异常，而@finally代码块会在异常重新抛出之前执行代码。这样就会导致本地pool的释放早于异常通知，因此它会变成可怕的僵尸异常（zombie exception）。幸好有一个简单的方法能解决它：只需在pool外保留它就行了。

我们的新方法如下所示：

```
- (void)myMethod
{
  id savedException = nil;
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  NSDictionary *myDictionary =
    [[NSDictionary alloc] initWithObjectsAndKeys:@"asdfads", nil];
  @try {
        [self processDictionary:myDictionary];
  } @catch (NSException *e) {
      savedException = [e retain];
      @throw;
  } @finally {
```

```
        [pool release];
        [savedException autorelease];
    }
}
```

通过使用retain方法，我们在当前池中保留了异常。当池被释放时，我们早已保存了一个异常指针，它会同当前池一同释放。

9.4 小结

本章介绍了Cocoa的内存管理方法：retain、release和autorelease，还讨论了垃圾回收和自动引用计数（ARC，苹果最新的内存管理技术）。

每个对象都维护一个保留计数器。对象被创建时，其保留计数器的值为1；对象被保留时，其保留计数器的值加1；对象被释放时，其保留计数器的值减1；当保留计数器的值归0时，对象被销毁。在销毁对象时，首先调用对象的dealloc方法，然后回收其占用的内存以供其他对象使用。

当对象接受到一条autorelease消息时，其保留计数器的值并不会发生改变。该对象只是被放入了NSAutoreleasePool当中。当自动释放池被销毁时，会向池中的所有对象发送release消息，所有被自动释放的对象都将其保留计数器的值减1。如果保留计数器的值归0了，则对象被销毁。在使用AppKit或UIKit的时候，自动释放池会在明确的时间创建或释放，比如在处理当前用户事件的时候。除此以外，你要创建自己的自动释放池。Foundation库工具集的模板中包含了这些代码。

Cocoa有三个关于对象及其保留计数器的规则：

- 如果使用new、alloc或copy操作获得了一个对象，则该对象的保留计数器的值为1。
- 如果通过其他方法获得一个对象，则假设该对象的保留计数器的值为1，而且已经被设置为自动释放。
- 如果保留了某对象，则必须保持retain方法和release方法的使用次数相等。

通常ARC会在编译过程中通过插入这些语句来帮你执行保留或释放操作。

下一章将讨论init方法：如何使对象在创建之后即可使用。

第 10 章 对象初始化 10

到目前为止，我们已经使用了两种不同的方法创建对象。第一种是[类名 new]，第二种是[[类名 alloc] init]。这两种方法是等价的，不过Cocoa惯例是使用alloc和init而不使用new。一般情况下，Cocoa程序员只在熟练使用alloc和init方法前将new作为辅助方法使用。现在是时候把它丢掉了。

10.1 分配对象

分配（allocation）是一个新对象诞生的过程。最美好的一刻，就是从操作系统获得一块内存，并将其指定为存放对象的实例变量的位置。向某个类发送alloc消息，就能为类分配一块足够大的内存，以存放该类的全部实例变量。同时alloc方法还顺便将这块内存区域全部初始化为0。这样，你就不必担心由于未初始化内存而引起各种随机bug（这在许多语言中都曾出现过）。所有的BOOL类型变量被初始化为NO，所有的int类型变量被初始化为0，所有的float类型变量被初始化为0.0，所有的指针被初始化为nil。

刚刚分配的对象并不能立即使用，你需要先初始化，然后才能使用。有些语言（包括C++和Java），使用构造函数在一次操作中便执行完对象的分配和初始化。Objective-C将这两种操作拆分为两个明确的步骤：分配和初始化。初学者常犯的一个错误就是只执行分配操作而未进行初始化，例如：

```
Car *car = [Car alloc];
```

这样的代码也许能运行，但由于未进行初始化，可能会出现奇怪的行为（也就是bug）。本章剩余的篇幅将全部用于讲解初始化这一核心概念。

10.1.1 初始化对象

与分配对应的操作是初始化。初始化从操作系统取得一块内存用于存储对象。init方法（即执行初始化操作的方法）一般都会返回其正在初始化的对象。你应该像下面这样嵌套调用alloc和init方法：

```
Car *car = [[Car alloc] init];
```

而不是像这样：

```
Car *car = [Car alloc]; [car init];
```

这种嵌套调用技术非常重要，因为初始化方法返回的对象可能与分配的对象不同。虽然这种情况很奇怪，但是它的确会发生。

为什么程序员会允许init方法返回不同的对象呢？如果你还记得第8章末尾对类簇的讨论，那么就会明白，像NSString和NSArray这样的类实际上只是一些特殊类的虚假表象。由于init方法可以接收参数，因此它的代码能够检查其接收的参数，并决定返回另一个类的对象可能更适合。例如，我们假设一条字符串可能是由一段很长的文本组成的，也可能是由一串阿拉伯数字组成的。基于这些情况，字符串初始化函数可能会决定创建一个不同类的对象（该对象更符合目标字符串的要求），然后返回该对象而不是原来的对象。

10.1.2 编写初始化方法

前面在展示初始化方法的用法时，曾要求你暂且不去细究，主要是因为这些方法看起来有点奇怪。如下所示是一个早期版本的CarParts类的init方法。

```
(id) init
{
 if (self = [super init])
 {
   engine = [Engine new];
   tires[0] = [Tire new];
   tires[1] = [Tire new];
   tires[2] = [Tire new];
   tires[3] = [Tire new];
  }
  return (self);
} // init
```

最令人感到奇怪的正是第一行：

```
if (self = [super init]) {
```

这行代码意味着self可能发生了改变。在方法中更改self？你疯了吗？也许吧，不过这次没疯。该声明中最先运行的代码是[super init]，其作用是让超类完成其自身的初始化工作。对于继承了NSObject的类来说，调用超类的init方法可以让NSObject执行它所需的所有操作，以便对象能够响应消息并处理保留计数器。而对于从其他类继承的类，通过这种方法可以实现自身的全新初始化。

刚才已经提到过，像这样的init方法可能会返回完全不同的对象。请记住，self参数是通过固定的距离寻找实例变量所在的内存位置的。如果从init方法返回一个新对象，则需要更新self，以便其后的实例变量的引用可以被映射到正确的内存位置。这也是需要使用self = [super init]这种形式进行赋值的原因。还要记住这个赋值操作只影响该init方法中self的值，而不影响该方法范围以外的任何内容。

如果在初始化一个对象时出现问题，则init方法可能会返回nil。比如使用init方法接收一个URL，并使用网站的图像文件初始化一个图像对象。如果网络出现故障，或者重新设计的网站删除了这幅图像，你就无法获得一个有效的图像对象。init方法将返回nil，表明未能初始化该对象。如果从[super init]返回的值是nil，则if (self = [super init])的判断不会让主体代码执行。像这样将赋值和检查是否为空值结合起来是一种典型的C语言风格，Objective-C沿袭了这一风格。

获得对象并使其运行的代码位于if语句正文部分的一对花括号里面。在最初的Car类的init方法中，if语句创建了一个engine对象和4个tire对象。从内存管理的角度来看，这段代码是完全正确的，因为通过new方法返回的对象在开始运行时保留计数器的值被设置为1。

该init方法的最后一行代码是"return (self);"。

init方法返回了刚刚已经被初始化的对象。我们已经将[super init]的返回值赋给了self，这正是应该返回的值。

初始化的正确方法

有些程序员不喜欢将赋值和判断是否为空结合起来，而是使用如下方式编写自己的init方法：

```
self = [super init];
if (nil != self)
{
...
}
return (self);
```

这样也很好。关键是你要将返回值赋给self，尤其是即将访问某个实例变量时。无论使用哪种方式来实现init方法，一定要明白，将赋值和条件判断结合起来是一种常见的技术，你将会在其他人的代码中经常见到这种情况。

self = [super init]这种形式是引起一些争议的根源。一部分人认为，初始化过程中超类可能改变一些对象，所以以防万一应该使用这种形式。另一部分人认为，这种对象改变的情况非常少见而且难以理解，因此不用理会，只使用简单明了的[super init]形式就行了。持有这种观点的人指出，即使init改变了对象，新对象也可能无法使用已经添加的新实例变量。

在理论上这是一个非常棘手的问题，但实际上这种情况很少发生。我们建议一直使用if (self = [super init])这种技术，以确保安全并能捕获某些init方法“返回nil”的行为。当然，如果使用简单明了的[super init]形式也可以，只不过遇到难以理解的个别情况时，你可能需要进行一些调试。

10.1.3　初始化时要做些什么

你应该在init方法中进行哪些工作？在这里，你要执行全新的初始化工作，给实例变量赋值并创建你的对象进行工作时所需的其他对象。在编写自己的init方法时，你必须确定在该方法中希望完成多少工作。在CarParts程序不断演化的过程中出现了两种不同的初始化方式。

第一种方式使用init方法创建了engine对象和全部的4个tire对象。这种方式使Car类变得可以“出产即用”（调用完alloc和init方法后就可以用来工作了）。而在另一种方式中，我们在init方法中不创建任何对象，只为engine对象和tire对象预留位置。创建了Car对象的方法还必须负责创建其中的engine对象和tire对象，并通过访问方法为其赋值。

哪种方式更适合你？这取决于灵活性与性能的权衡，在程序设计中经常要做这种取舍。前一种方式中原始的init方法非常方便。如果Car类的用途是创建和使用默认的car对象，则第一种方式更为合适。

另一方面，如果car对象经常要定制不同种类的引擎和轮胎（比如在赛车游戏中），我们创建的这些默认engine对象和tire对象只能丢掉。这样太浪费资源了，创建的对象从未使用就被销毁了。

> **说明** 即便你目前没有设置自定义属性的值，也应该等到调用者需要时再创建对象。这种技术被称为**惰性求值**（lazy evaluation）。如果你要在自己的init方法中创建复杂但实际可能用不上的对象，则使用这种技术可以提高程序的性能。

10.2 便利初始化函数

有些对象拥有多个以init开头的方法名。需要记住，这些init方法实际上没什么特别的，只是遵循命名约定的普通方法。

许多类包含*便利初始化函数*（convenience initializer），它们是用来完成某些额外工作的初始化方法，可以减轻你的负担。为了帮助你理解，下面给出NSString类中的一些初始化方法的示例。

```
- (id) init;
```

这一基本方法初始化一个新的空字符串。对于不可变的NSString类来说，这个方法没有多大的用处。不过，你可以分配和初始化一个新的NSMutableString类的对象并开始向该类中添加字符。可以像下面这样使用此对象：

```
NSString *emptyString = [[NSString alloc] init];
```

上面的代码返回一个空字符串。

```
- (id) initWithFormat: (NSString *) format, …;
```

正如我们使用NSLog()函数和第7章中的类方法stringWithFormat:接收格式化的字符串并输出格式化的结果一样，这个版本的代码初始化了一个新的字符串作为格式化操作的结果。使用此初始化方法的例子如下：

```
string = [[NSString alloc]
initWithFormat: @"%d or %d", 25, 624];
```

上面的代码返回一个字符串，其值为“25 或 624”。

```
- (id) initWithContentsOfFile:(NSString *) path encoding:(NSStringEncoding) enc error:(NSError **) error
```

这个initWithContentsOfFile:encoding:error:方法用来打开指定路径上的文本文件，读取文件内容，并使用文件内容初始化一个字符串。读取文件/tmp/words.txt的代码如下：

```
NSError *error = nil;
NSString *string =
[[NSString alloc] initWithContentsOfFile:@"/tmp/words.txt"
                encoding:NSUTF8StringEncoding
                  error:&error];
```

encoding参数将文件内容的类型告诉了API。一般来说，你应该使用NSUTF8StringEncoding，它表示文件内容是用UTF8格式进行编码的。

第三个参数会在初始化没有发生错误时返回nil值。如果出现了错误，你可以使用localizedDescription方法来查明情况。把它也放进去的话，代码应该会如下所示：

```
NSError *error = nil;
NSStringEncoding encoding = NSUTF8StringEncoding;
NSString *string = [[NSString alloc] initWithContentsOfFile:@"/tmp/words.txt"
                          usedEncoding:&encoding
                             error:&error];
if(nil != error)
{
  NSLog(@"Unable to read data from file, %@", [error localizedDescription]);
}
```

这段代码功能非常强大。在C语言中完成这一工作将需要一大堆代码（必须打开文件，读取数据块，添加到字符串，确保尾部零字节位于正确的位置以及关闭文件）。对于Objective-C爱好者来说，这些操作简化成了几行代码，真是太方便了。

> **说明**　初始化函数的一般规则是，假如你的对象必须要用某些信息进行初始化，那么你应该将这些信息作为init方法的一部分添加进来。这些都是典型的不可变对象。

10.3　更多部件改进

让我们来重温一下在第6章（当时我们将每一个类组织到各自的源文件中）中出现的CarParts项目。这一次，我们将在Tire类中加入一些更高级的初始化操作，并使用这些方法进行Car类的内存管理。

> **说明**　我们在第9章讨论过，苹果公司把内存管理的垃圾回收转向了一个叫做自动引用计数（ARC）的新技术。如果是做iOS开发，由于不支持垃圾回收，必须使用ARC技术。本章中支持ARC的项目放在了10.01 CarPartsInit-ARC文件夹内。不过因为垃圾回收机制依然存在，所以我们也提供了一个支持垃圾回收的版本，它位于文件夹10.01 CarPartsInit-GC中。

10.3.1 Tire类的初始化

现实世界中的轮胎比我们在CarParts项目中模拟的轮胎要有趣得多。对于真实的轮胎，必须留心轮胎压力（不能太小）和胎面花纹深度（小于几毫米就不安全了）。下面让我们扩展Tire类以记录轮胎压力和花纹深度。增加了两个实例变量和相应的访问方法的类声明如下：

```
#import <Cocoa/Cocoa.h>
@interface Tire : NSObject
 {
 float pressure;
 float treadDepth;
}
-(void) setPressure: (float) pressure;
-(float) pressure;
-(void) setTreadDepth: (float) treadDepth;
-(float) treadDepth;
@end // Tire
```

Tire类的实现非常简单，其代码如下：

```
#import "Thre.h"
@implementation Tire
- (id) init
{
 if (self = [super init])
 {
  pressure = 34.0; treadDepth = 20.0;
 }
 return (self);
} // init
- (void) setPressure: (float) p
{
  pressure = p;
} // setPressure
- (float) pressure
{
  return (pressure);
} // pressure
- (void) setTreadDepth: (float) td
{
  treadDepth = td;
} // setTreadDepth
- (float) treadDepth
{
 return (treadDepth);
} // treadDepth
- (NSString *) description
{
 NSString *desc;
  desc = [NSString stringWithFormat:
   @"Tire: Pressure: %.1f TreadDepth: %.1f", pressure, treadDepth];
  return (desc);
} // description
@end // Tire
```

访问方法为tire对象的使用者提供了一种修改轮胎压力和花纹深度的方法。让我们快速浏览一下init方法。

```
- (id) init
{
  if (self = [super init])
 {
  pressure = 34.0;
  treadDepth = 20.0;
 }
 return (self);
} // init
```

你应该不会感到奇怪。超类（在本例中是NSObject）被告知初始化它自己，对超类调用init方法的返回值被赋给了self。然后，实例变量设置成了默认的有效值。下面，我们创建一个全新的tire对象：

```
Tire *tire = [[Tire alloc] init];
```

这个tire对象的轮胎压力将是34psi（1 psi = 0.06896bar），花纹深度将是20mm。我们还应该修改description方法：

```
- (NSString *) description
{
    NSString *desc;
    desc = [NSString stringWithFormat:
  @"Tire: Pressure: %.1f TreadDepth: %.1f", pressure, treadDepth];
    return (desc);
} // description
```

现在description方法使用NSString的类方法stringWithFormat:生成了一个包含轮胎压力和花纹深度的字符串。该方法有没有遵守良好的内存管理规则？有的，因为desc对象不是通过alloc、copy和new方法创建的，所以它的保留计数器的值为1，而且我们可以认为它是自动释放的。因此，当自动释放池销毁时，该字符串对象也将被清理。

10.3.2　更新main()函数

以下是更新后的main()函数，它比之前的版本要复杂一些：

```
#import "Engine.h"
#import "Car.h"
#import "Slant6.h"
#import "AllWeatherRadial.h"
int main (int argc, const char * argv[])
{
@autoreleasepool
{
 Car *car = [[Car alloc] init];
 for (int i = 0; i < 4; i++)
{
 Tire *tire;
```

```
 tire = [[Tire alloc] init];
 [tire setPressure: 23 + i]; [tire setTreadDepth: 33 - i];
 [car setTire: tire atIndex: i];
 [tire release];
}
 Engine *engine = [[Slant6 alloc] init];
 [car setEngine: engine];
 [car print];
 [car release];
}
 return (0);
} // main
```

我们来逐步讲解main()函数。首先创建一个自动释放池，为自动释放的对象提供容身之处，以等待自动释放池被销毁。

```
@autoreleasepool {
```

然后使用alloc和init创建一个全新的car对象：

```
Car *car = [[Car alloc] init];
```

之后，执行4次循环（从0到3）。通过该循环创建所有的tire对象：

```
for (int i = 0; i < 4; i++) {
```

每执行一次循环，都创建并初始化一个全新的tire对象：

```
Tire *tire;
tire = [[Tire alloc] init];
```

每个tire对象开始时的轮胎压力和花纹深度都是由Tire类的init设置的，但是我们准备自己来设置这些值（只是为了好玩）。因为在现实中没有两个轮胎是完全相同的，所以我们打算使用访问方法调整轮胎压力和花纹深度的值。

```
[tire setPressure: 23 + i];
[tire setTreadDepth: 33 - i];
```

接下来将tire对象提供给car对象：

```
[car setTire: tire atIndex: i];
```

至此，tire对象的使命已经完成，我们将其释放掉：

```
[tire release];
```

这行代码假定Car类已经正确地执行过内存管理，保留了tire对象。请注意，第6章中提供的Car类不符合我们的内存管理规则（那时我们还不太清楚如何做），稍后将告诉你如何解决这个问题。

与之前一样，在轮胎组装完成以后，我们创建了一个新的engine对象，并将该engine对象放在了car中。

```
Engine *engine = [[Slant6 alloc] init];
[car setEngine: engine];
[engine release];
```

与tire对象一样，engine对象也被释放，因为它的使命已经完成了。之后是由Car类来负责

销毁engine实例对象了。

最后，car对象被告知要输出自己的内容，完成之后我们就可以释放car对象了。

```
[car print];
[car release];
```

自动释放池现在保留计数器的值归0并得到释放，该池被销毁并向池中的每一个对象发送了release消息。此时，由Tire类的description方法生成的所有NSString字符串都在main()最终结束时被清理掉了。不过在运行这个程序之前，我们需要修改一下Car类，使它能够正确地执行内存管理。此外在这里我们也可以先看看支持垃圾回收版本的main()是如何实现的。

```
int main (int argc, const char * argv[])
{
Car *car = [[Car alloc] init];
for (int i = 0; i < 4; i++)
{
 Tire *tire;
 tire = [[Tire alloc] init];
 [tire setPressure: 23 + i];
 [tire setTreadDepth: 33 - i];
 [car setTire: tire atIndex: i]; }
 Engine *engine = [[Slant6 alloc] init];
 [car setEngine: engine];
 [car print];
 return (0);
} // main
```

正如你所看到的，main()函数没有额外的内存管理调用，代码更加简短。

10.3.3　清理Car类

我们使用NSMutableArray来替代Car类中常规的C数组，因为这样就不用执行上限检查了。此外，我们修改了Car类的@interface部分，以便它可以支持使用可变数组（改动的代码以粗体显示）。

```
#import <Cocoa/Cocoa.h>
@class Tire;
@class Engine;
@interface Car : NSObject
 {
 NSMutableArray *tires;
 Engine *engine;
}
- (void) setEngine: (Engine *) newEngine;
- (Engine *) engine;
- (void) setTire: (Tire *) tire
atIndex: (int) index;
- (Tire *) tireAtIndex: (int) index;
- (void) print;
@end // Car
```

我们几乎修改了Car类中的每一个方法，以使其遵循内存管理规则。先来看一下init方法。

```
- (id) init
{
 if (self = [super init])
{
  tires = [[NSMutableArray alloc] init];
  for (int i = 0; i < 4; i++)
 {
   [tires addObject: [NSNull null]];
 }
}
 return (self);
} // init
```

你已经不止一次见到过self = [super init]，对于这种风格应该已经熟记于心了。就目前你所知道的，它可以确保超类能初始化对象并使其能够正常运行。

接下来，我们创建了NSMutableArray数组。在NSMutableArray类里面有一个简便的方法replaceObjectAtIndex:withObject:，该方法最适合用来实现setTire:atIndex:方法。如果要使用replaceObjectAtIndex:withObject:方法，在指定的索引位置处必须存在一个能够被替换的对象。全新的NSMutableArray数组不包含任何内容，因此我们需要使用一些对象来作为占位符，而NSNull类的对象非常适合完成此项工作。因此，我们在该数组中添加4个NSNull对象（第8章中介绍过这些对象）。一般情况下，你不需要使用NSNull类的对象预置NSMutableArray数组，在本例中我们这样做只是为了使随后的实现更加容易。

在init方法结束时，我们返回self，这正是我们刚刚完成初始化的对象。

下面是两个访问方法setEngine:和engine。setEngine:使用的是之前讲过的“保留以前已传入的对象并释放当前对象”的技术。

```
- (void) setEngine: (Engine *) newEngine
{
 [newEngine retain];
 [engine release];
 engine = newEngine;
} // setEngine
```

而engine的访问方法只是简单地返回当前的engine对象：

```
- (Engine *) engine
{
 return (engine);
} // engine
```

现在我们来实现tire对象的访问方法。首先是setter方法：

```
- (void) setTire: (Tire *) tire
atIndex: (int) index
{
 [tires replaceObjectAtIndex: index withObject: tire];
} // setTire:atIndex:
```

setTire:方法使用replaceObjectAtIndex:withObject:从数组集合中删除现有对象并用新对象替代。因为NSMutableArray数组会自动保留新的tire对象并释放索引位置上的对象（无论该对

象是NSNull占位符还是tire对象），所以对于tire对象不需要执行其他内存管理。当NSMutableArray数组被销毁时，它将释放数组中的所有对象，因此tire对象会被清理。

Getter方法tireAtIndex:使用了由NSArray类提供的objectAtIndex:方法从数组中获取tire对象：

```
- (Tire *) tireAtIndex: (int) index
{
 Tire *tire;
 tire = [tires objectAtIndex: index];
 return (tire);
} // tireAtIndex:
```

快速返回

可以直接返回objectAtIndex:方法的结果值，将如下方法简化为一行，这完全符合语法。

```
(Tire *) tireAtIndex: (int) index
{
return ([tires objectAtIndex: index]);
} // tireAtIndex:
```

不过在原来的tireAtIndex:方法中引入额外的变量，可以使代码更易于阅读（至少对我们来说是这样的），使设置断点更加方便，因此我们可以明白该方法返回的是哪个对象。使用这种技术也使得原始调试在调用objectAtIndex:方法到该方法返回tire对象的过程中输出NSLog()函数变得更加容易。

我们仍然需要确保car能够清理其保留的对象，尤其是engine和tire数组，而dealloc方法就可以完成此类工作。

```
- (void) dealloc
{
 [tires release];
 [engine release];
 [super dealloc];
} // dealloc
```

这可以确保在该car对象被销毁时所有的内存都将被回收。一定要调用超类的dealloc方法，这一点很容易被忽视。并且要确保[super dealloc]是dealloc方法的最后一条语句。

最后是car对象的print方法，该方法输出tire对象和engine对象。

```
- (void) print
{
for (int i = 0; i < 4; i++)
{
 NSLog (@"%@", [self tireAtIndex: i]);
}
 NSLog (@"%@", engine);
} // print
```

循环中的print方法依次输出每个tire对象的描述并将它们显示出来。有趣的是，该循环使

用tireAtIndex:方法间接取值而不是直接从数组中取值。如果希望直接访问NSMutableArray数组，完全可以这么做。不过如果你使用的是访问方法（即便是在类的实现中），应该让代码不受将来更改的影响。比方说，tire对象的存储机制以后又发生了改变（例如又回到了之前的C风格的数组），你就不需要再去修改print方法了。

现在，我们终于可以运行CarPartsInit了，其运行结果如下。

```
Tire: Pressure: 23.0 TreadDepth: 33.0
Tire: Pressure: 24.0 TreadDepth: 32.0
Tire: Pressure: 25.0 TreadDepth: 31.0
Tire: Pressure: 26.0 TreadDepth: 30.0
I am a slant- 6. VROOOM!
```

10.4 Car类的内存清理（垃圾回收方式和ARC方式）

那么垃圾回收是如何执行的呢？在支持垃圾回收的Objective-C中，Car类是什么样子的呢？setEngine方法变得更加简单。

```
- (void) setEngine: (Engine *) newEngine
{
 engine = newEngine;
} // setEngine
```

我们修改了engine对象的实例变量。当Cocoa的垃圾回收机制运行时，它知道没有其他实例变量指向原来的engine对象，因此，垃圾回收器销毁了原来的engine对象。另一方面，因为有一个实例变量指向了newEngine对象，所以newEngine对象不会被回收。垃圾回收器知道有变量正在使用newEngine对象。

dealloc方法完全消失了，因为在支持垃圾回收的Objective-C中，dealloc方法毫无用武之地。如果需要在销毁对象时执行某些操作，则需要重写-finalize方法，当对象最终被回收时该方法会被调用。虽说使用-finalize方法会引起一些细小的问题，不过对于Cocoa环境下的程序设计来说，你无需担心。

ARC版本与垃圾回收版本类似。我们唯一要加的就是@autoreleasepool，来告诉编译器我们想要它来处理保留和释放事件。

构造便利初始化函数

没有什么代码是完美的。回忆一下我们在main()函数中创建tire对象的方式：

```
tire = [[Tire alloc] init];
[tire setPressure: 23 + i];
[tire setTreadDepth: 33 - i];
```

这里使用了4条消息和3行代码。如果能在单次操作中完成此功能，那再好不过了。下面，我们构造一个能够同时获取轮胎压力和花纹深度的便利初始化函数。带有新增的初始化函数（以粗

体表示）的Tire类代码如下所示。

```
@interface Tire : NSObject
{
 float pressure;
 float treadDepth;
}
- (id) initWithPressure: (float) pressure
treadDepth: (float) treadDepth;
- (void) setPressure: (float) pressure;
- (float) pressure;
- (void) setTreadDepth: (float) treadDepth;
- (float) treadDepth;
@end // Tire
```

毫无疑问，该方法的实现非常简单：

```
- (id) initWithPressure: (float) p treadDepth: (float) td
{
  if (self = [super init]) {
     pressure = p;
     treadDepth = td;
 }
 return (self);
} // initWithPressure:treadDepth:
```

现在，我们可以在单步操作中完成tire对象的分配和初始化了。

```
Tire *tire;
tire = [[Tire alloc]
initWithPressure: 23 + i
treadDepth: 33 - i];
```

10.5　指定初始化函数

不幸的是，并不是所有的初始化方法都能够正常运行。当在类中增加便利初始化函数时，会出现一些小问题。下面，我们在Tire类中增加两个便利初始化函数。

```
@interface Tire : NSObject
{
 float pressure;
 float treadDepth;
}
- (id) initWithPressure: (float) pressure;
- (id) initWithTreadDepth: (float) treadDepth;
- (id) initWithPressure: (float) pressure treadDepth: (float) treadDepth;
- (void) setPressure: (float) pressure;
- (float) pressure;
- (void) setTreadDepth: (float) treadDepth;
- (float) treadDepth;
@end // Tire
```

initWithPressure:和initWithTreadDepth:这两个新的初始化函数适用于那些知道轮胎需要

特定轮胎压力或花纹深度而不关心其他属性（并且乐意接受默认值）的人。第一次编写的初始化方法如下（随后我们将对其进行改进）。

```
- (id) initWithPressure: (float) p
{
 if (self = [super init])
 {
  pressure = p;
  treadDepth = 20.0;
 }
 return (self);
} // initWithPressure
- (id) initWithTreadDepth: (float) td
{
 if (self = [super init])
 {
   pressure = 34.0;
  treadDepth = td;
 }
 return (self);
} // initWithTreadDepth
```

现在我们已经有了4个初始化方法：init、initWithPressure:、initWithTreadDepth:和initWithPressure:treadDepth:，每个都知道默认的轮胎压力值（34）或花纹深度值（20）。这些方法能正常工作，代码也是正确的。

但是在开始子类化Tire类时，问题就来了。

10.5.1 子类化问题

我们已经创建了Tire类的子类AllWeatherRadial。现在假设AllWeatherRadial类希望增加两个实例变量：rainHandling和snowHandling。这两个变量都是浮点值，用于表示轮胎在潮湿和积雪的道路上的性能值。在创建一个新的AllWeatherRadial类对象时，必须确保这两个变量具有合理的值。

因此，带有新的实例变量和访问方法的AllWeatherRadial类的新interface代码将如下所示。

```
@interface AllWeatherRadial : Tire
{
 float rainHandling;
 float snowHandling;
}
- (void) setRainHanding: (float) rainHanding;
- (float) rainHandling;
- (void) setSnowHandling: (float) snowHandling;
- (float) snowHandling;
@end // AllWeatherRadial
```

以下是访问方法（已经看得有些烦了）：

```
- (void) setRainHandling: (float) rh
{
  rainHandling = rh;
```

```
} // setRainHandling
- (float) rainHandling
{
  return (rainHandling);
} // rainHandling
- (void) setSnowHandling: (float) sh
{
 snowHandling = sh;
} // setSnowHandling
- (float) snowHandling
{
  return (snowHandling);
} // snowHandling
```

我们修改了description方法以显示tire对象的各个参数：

```
- (NSString *) description
{
 NSString *desc;
 desc = [[NSString alloc] initWithFormat:
  @"AllWeatherRadial: %.1f / %.1f / %.1f / %.1f",
 [self pressure], [self treadDepth],
 [self rainHandling],
 [self snowHandling]];
 return (desc);
} // description
```

下面是main()函数的for循环，该循环创建了具有默认变量值的新AllWeatherRadial对象：

```
for (int i = 0; i < 4; i++)
{
AllWeatherRadial *tire;
 tire = [[AllWeatherRadial alloc] init];
 [car setTire: tire atIndex: i];
 [tire release];
}
```

然而，当我们运行程序时，问题出现了：

```
AllWeatherRadial: 34.0 / 20.0 / 0.0 / 0.0
AllWeatherRadial: 34.0 / 20.0 / 0.0 / 0.0
AllWeatherRadial: 34.0 / 20.0 / 0.0 / 0.0
AllWeatherRadial: 34.0 / 20.0 / 0.0 / 0.0
I am a slant- 6. VROOOM!
```

AllWeatherRadial对象的新变量未被设置为合理的默认值。哪里出了问题？因为是在init方法中赋值的，所以我们必须重写init方法。但是Tire类中还有initWithPressure:、initWithTreadDepth:和initWithPressure:treadDepth:三个初始化方法，难道我们必须重写所有这些方法吗？而且，即使我们这么做了，假如Tire类中又增加了一个新的初始化函数，那又会怎么样呢？如果Tire类的改动会影响到AllWeatherRadial类，那么可以说这种程序设计的方式是有问题的。

幸好Cocoa的设计人员已经预料到这个问题并提出了*指定初始化函数*（designated initializer）

这一概念，即类中的某个初始化方法被指派为指定初始化函数。该类的所有初始化方法都使用指定初始化函数执行初始化操作，而子类使用其超类的指定初始化函数进行超类的初始化。通常，接收参数最多的初始化方法是最终的指定初始化函数。如果你使用了其他人缩写的代码，则一定要检查文档，弄清楚哪个方法是指定初始化函数。

10.5.2 Tire类的初始化函数改进后的版本

首先，我们需要确定应该将Tire类的哪个初始化函数指派为指定初始化函数。initWithPressure:treadDepth:就是一个不错的选择。该方法参数最多，而且在这些初始化函数中是最灵活的。

为了保证指定初始化函数的顺利执行，所有其他的初始化函数应该按照initWithPressure:treadDepth:的形式实现，可能如下所示。

```
- (id) init
{
 if (self = [self initWithPressure: 34 treadDepth: 20])
 {
 }
 return (self);
} // init
- (id) initWithPressure: (float) p
{
 if (self = [self initWithPressure: p treadDepth: 20.0])
 {
 }
 return (self);
} // initWithPressure
- (id) initWithTreadDepth: (float) td
{
 if (self = [self initWithPressure: 34.0 treadDepth: td])
 {
 }
return (self);
} // initWithTreadDepth
```

> **说明** 你并不需要真的像以上代码那样在if语句中不写任何代码。我们之所以这么做只是为了保持所有的初始化方法具有一致的形式。

10.5.3 添加AllWeatherRadial类的初始化函数

现在应该向AllWeatherRadial类中添加指定初始化函数了，我们只需添加重写的指定初始化函数。

```
- (id) initWithPressure: (float) p
treadDepth: (float) td
{
 if (self = [super initWithPressure: p treadDepth: td])
```

```
    {
     rainHandling = 23.7;
     snowHandling = 42.5;
    }
    return (self);
} // initWithPressure:treadDepth
```

这时你再运行该程序，AllWeatherRadial类的对象便会设置合适的默认值了。

```
AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
I am a slant- 6. VROOOM!
```

哇，汽车真的发动了！

10.6　初始化函数规则

并不是一定要为你自己的类创建初始化函数。如果不需要设置任何状态，或者alloc方法将内存清零的默认行为相当不错，则不必去在意init方法。

如果创建了一个指定初始化函数，则一定要在你自己的指定初始化函数中调用超类的指定初始化函数。

如果初始化函数不止一个，则需要选择一个作为指定初始化函数。被选定的方法应该调用超类的指定初始化函数。要按照指定初始化函数的形式实现所有其他初始化函数，就像我们在前面代码中所实现的那样。

10.7　小结

本章主要介绍了对象的分配和初始化。在Cocoa中，分配和初始化是两个分离的操作：来自NSObject的类方法alloc为对象分配一块内存区域并将其清零，实例方法init用于获得一个对象并使其运行。

一个类可以拥有多个初始化方法。这些初始化方法通常是便利初始化方法，可以更容易地按照你的想法进行对象设置。你可以从这些初始化方法中选择一个作为指定初始化函数。所有其他初始化方法应该按照指定初始化函数的形式编写。

在你自己的初始化方法中，需要调用自己的指定初始化函数或者超类的指定初始化函数。一定要将超类的初始化函数的返回值赋给self，并返回你自己init方法的值。需要注意的是，超类可能会返回一个完全不同的对象。

下一章将介绍属性，这是创建访问方法最快速、最便捷的方式。

第 11 章 属 性 11

还记得我们在为实例变量编写访问方法时曾被搞得一头雾水吗？我们编写了大量雷同的代码，不但要编写-setBlah方法来设置对象的blah变量，还要编写-blah方法来取回对象的blah变量。如果变量是对象，则需要保留新对象并释放旧对象。虽然可以向文件中粘贴方法的声明和定义，但是编写访问方法仍然是一项索然无味的工作，而我们完全可以利用这些时间来为自己的程序实现更棒的特有功能。

苹果公司在Objective-C 2.0中引入了属性（property），它组合了新的预编译指令和新的属性访问器语法。新的属性功能显著减少了必须编写的冗长代码的数量。本章我们会修改CarPartsInit项目以使用属性功能，本章的最终代码位于11.01 CarProperties项目文件夹中。

请记住，Objectve-C 2.0的特性只适用于Mac OS X 10.5（Leopard）以上的版本，如果你必须支持旧的系统，那么就要三思了。属性在Cocoa（尤其是华丽夺目的Core Animation效果）里广泛使用并且在iOS开发中也经常使用，因此值得我们去学习。

11.1 使用属性值

首先，我们转换一个比较简单的类（AllWeatherRadial）以使用属性。为了增强趣味性，我们在main()函数中增加了几次调用，以修改我们创建的AllWeatherRadial类的一些值。我们假设有人从不同的商店购买了4个降价销售的轮胎，因此这4个轮胎具有不同的性能值。

修改后的mian()函数如下（新增的代码以粗体显示）。

```
int main(int argc, const char * argv[])
{
  @autoreleasepool
  {
    Car *car = [[Car alloc] init];
    for (int i = 0; i < 4; i++)
    {
      AllWeatherRadial *tire;

      tire = [[AllWeatherRadial alloc] init];
      [tire setRainHandling:20+i];
      [tire setSnowHandling:28+i];
      NSLog(@"tire %'s handling is %.f %.f", i,
```

```
        [tire rainHandling], [tire snowHandling]);
      [car setTire:tire atIndex:i];

      [tire release];
    }

    Engine *engine = [[Slant6 alloc] init];
    [car setEngine:engine];

    [car print];
    [car release];
  }
    return (0);
}
```

如果现在运行该程序，你会得到如下输出结果，该结果显示了我们最新修改过的轮胎的性能。

```
tire 0's handling is 20 28
tire 1's handling is 21 29
tire 2's handling is 22 30
tire 3's handling is 23 31
AllWeatherRadial: 34.0 / 20.0 / 20.0 / 28.0
AllWeatherRadial: 34.0 / 20.0 / 21.0 / 29.0
AllWeatherRadial: 34.0 / 20.0 / 22.0 / 30.0
AllWeatherRadial: 34.0 / 20.0 / 23.0 / 31.0
I am a slant-6. VROOOM!
```

11.1.1　简化接口代码

现在来研究一下AllWeatherRadial类的接口代码。

```
#import <Foundation/Foundation.h>
#import "Tire.h"
@interface AllWeatherRadial : Tire
{
 float rainHandling;
 float snowHandling;
}
- (void) setRainHandling: (float) rainHanding;
- (float) rainHandling;
- (void) setSnowHandling: (float) snowHandling;
- (float) snowHandling;
@end // AllWeatherRadial
```

这种写法已经过时了，我们将其改写为具有属性风格的新形式。

```
#import <Foundation/Foundation.h>
#import "Tire.h"
@interface AllWeatherRadial : Tire
{
 float rainHandling;
 float snowHandling;
```

```
}
@property float rainHandling;
@property float snowHandling;
@end // AllWeatherRadial
```

是不是更简单了？不需要4个方法的定义语句了。注意，我们引入了两个以@为前缀的关键字。之前曾介绍过，@符号标志着“这是Objective-C语法”。@property是一种新的编译器功能，它意味着声明了一个新对象的属性。

@property float rainHandling;语句表明AllWeatherRadial类的对象具有float类型的属性，其名称为rainHandling。而且你还可以通过调用-setRainHandling:来设置属性，通过调用-rainHandling来访问属性。现在你运行该程序，将得到和之前一样的结果。@property预编译指令的作用是自动声明属性的setter和getter方法。事实上，属性的名称不必与实例变量的名称相同，但大多数情况下它们是一样的，稍后将讨论这个话题。属性还有一些其他的用处，也将在稍后讨论，请耐心等待。

11.1.2 简化实现代码

现在让我们来看一下AllWeatherRadial类的实现代码。

```
#import "AllWeatherRadial.h"
@implementation AllWeatherRadial
- (id) initWithPressure: (float) p treadDepth: (float) td
{
 if (self = [super initWithPressure: p treadDepth: td])
 {
   rainHandling = 23.7;
   snowHandling = 42.5;
 }
return (self);
} // initWithPressure:treadDepth
- (void) setRainHandling: (float) rh
{
 rainHandling = rh;
} // setRainHandling

- (float) rainHandling
{
  return (rainHandling);
} // rainHandling
- (void) setSnowHandling: (float) sh
{
 snowHandling = sh;
} // setSnowHandling
- (float) snowHandling
{
  return (snowHandling);
} // snowHandling
- (NSString *) description
{
 NSString *desc;
 desc = [[NSString alloc] initWithFormat:
```

```
    @"AllWeatherRadial: %.1f / %.1f / %.1f / %.1f",
    [self pressure], [self treadDepth],
    [self rainHandling],
    [self snowHandling]];
  return (desc);
} // description
@end // AllWeatherRadial
```

在上一章中，我们曾经讨论了AllWeatherRadial类的init方法、指定初始化函数、所有的setter和getter方法以及description方法。现在，我们打算彻底删除全部的setter和getter方法，而用两行简单的代码来代替。

```
#import "AllWeatherRadial.h"
@implementation AllWeatherRadial
@synthesize rainHandling;
@synthesize snowHandling;
- (id) initWithPressure: (float) p treadDepth: (float) td
{
 if (self = [super initWithPressure: p treadDepth: td])
 {
  rainHandling = 23.7;
  snowHandling = 42.5;
 }
 return (self);
} // initWithPressure:treadDepth
- (NSString *) description
{
 NSString *desc;
 desc = [[NSString alloc] initWithFormat:
  @"AllWeatherRadial: %.1f / %.1f / %.1f / %.1f", [self pressure], [self treadDepth],
  [self rainHandling], [self snowHandling]];
 return (desc);
} // description
@end // AllWeatherRadial
```

@synthesize[①]也是一种新的编译器功能，它表示“创建了该属性的访问代码”。当遇到@synthesize rainHandling;这行代码时，编译器将添加实现-setRainHandling:和-rainHandling方法的预编译代码。

> **说明**　你可能非常熟悉代码生成：Cocoa的访问器编写实用工具和其他平台上的UI生成器可以生成源代码，这些源代码随后会被编译。但是@synthesize预编译指令不同于代码生成。你永远也不会看到实现-setRainHandling:和-rainHandling的代码，但是这些方法确实存在并可以被调用。这种技术使苹果公司可以更加灵活地改变Objective-C中生成访问方法的方式，并获得更安全的实现和更高的性能。
>
> 如果现在运行该程序，得到的结果将与我们在修改AllWeatherRadial类实现代码之前的一样。

① 在Xcode 4.5以后的版本中，可以不必使用@synthesize了。

所有的属性都是基于变量的，所以在你合成（synthesize）getter和setter方法的时候，编译器会自动创建与属性名称相同的实例变量。需要注意头文件中有两个叫做rainHandling和snowHandling的实例变量（合成的setter和getter方法会用到这些变量）。如果你没有声明这些变量，编译器也会声明的。有两个地方可以用来添加实例变量声明：头文件和实现文件。我们甚至可以拆开来，一部分在头文件中声明，一部分在实现文件中声明。

那么变量的声明语句放在这两个地方有什么区别呢？假设你有一个子类，并且想要从子类直接通过属性来访问变量。在这种情况下，变量就必须放在头文件中。如果变量只属于当前类，则可以把它们放在.m文件里（并且要删除原interface代码中的声明语句）。修改后的头文件如下所示。

```
@interface AllWeatherRadial : Tire
@property float rainHandling;
@property float snowHandling;
@end // AllWeatherRadial
```

而实现文件应该如下所示：

```
implementation AllWeatherRadial
{
  float rainHandling;
  float snowHandling;
}

@synthesize rainHandling;
@synthesize snowHandling;

- (id) initWithPressure: (float)p treadDepth: (float)td
{
  if (self = [super initWithPressure: p treadDepth: td])
  {
    rainHandling = 23.7;
    snowHandling = 42.5;
  }
  return (self);
} // initWithPressure:treadDepth
- (NSString *) description
{
  NSString *desc;
  desc = [[NSString alloc] initWithFormat:
          @"AllWeatherRadial: %.1f / %.1f / %.1f / %.1f",
        [self pressure], [self treadDepth],
        [self rainHandling],
        [self snowHandling]];
  return (desc);
} // description
@end // AllWeatherRadial
```

如你所见，头文件的代码变少了，看起来更简洁。这也意味着你出错的概率降低了。

我们还可以更进一步。请记住，如果没有指定实例变量，编译器会自动帮我们创建，所以我们可以清除以下代码，这不会有任何影响。

```
{
  float rainHandling;
  float snowHandling;
}
```

（可以删除所有代码，包括花括号。）现在我们已经删去了大量代码，这样就能为我们节省打字和调试的时间了。

11.1.3　点表达式的妙用

Objective-C 2.0的属性引用了一些新的语法特性，使我们可以更加容易地访问对象的属性。这些新特性也方便那些习惯了C++和Java等语言的编程人员学习Objective-C。

回想一下我们在main()函数中添加的用于修改轮胎性能值的两行代码：

```
[tire setRainHandling: 20 + i];
[tire setSnowHandling: 28 + i];
```

我们可以将其替换为下面的代码：

```
tire.rainHandling = 20 + i;
tire.snowHandling = 28 + i;
```

如果现在运行该程序，你将会得到与之前一样的结果。以下是我们之前使用NSLog()函数来输出轮胎的性能值的代码。

```
NSLog(@"tire %d's handling is %.f %.f", i, [tire rainHandling], [tire snowHandling]);
```

现在，我们可以将其替换为下面的代码：

```
NSLog(@"tire %d's handling is %.f %.f", i, tire.rainHandling, tire.snowHandling);
```

点表达式（.）看起来与C语言中的结构体访问以及Java语言中的对象访问有些相似，其实这是Objective-C的设计人员有意为之。如果点表达式出现在了等号（=）的左边，该变量名称的setter方法（-setRainHandling:和-setSnowHandling:）将被调用。如果点表达式出现在了对象变量的右边，则该变量名称的getter方法（-rainHandling和-snowHandling）将被调用。

> **说明**　点表达式只是调用访问方法的一种便捷方式，并没有什么神秘之处。我们将在第18章中讨论键/值编码，该机制实际上使用了一些基本的运行时技术。属性的点表达式和键/值编码的后台工作之间没有联系。

如果你在访问属性时遇到了奇怪的错误信息，提示访问的对象不是struct类型，请检查当前的类是否已经包含了所需的所有必备头文件。

在引入了属性概念后这种问题经常出现。当然，我们会在后面讲解如何正确地处理对象变量以及如何避免暴露setter和getter方法。

11.2 属性扩展

到目前为止，我们已经研究过float类型的属性，这些技术同样适用于int、char、BOOL和struct类型。而且，只要你愿意，甚至可以定义一个NSRect对象的属性。

不过对象也会带来一些麻烦。回想一下，我们在使用访问方法来访问对象时需要保留和释放对象。对于某些对象的值，尤其是字符串的值，你总是会复制（-copy）它们。而对于其他对象的值，如委托（将在下一章中讨论），你根本不会想要保留它们。

> **说明** 为什么要复制对象？为什么不保留对象？
>
> 你想要复制字符串参数。一种常见的错误就是从用户界面（如文本框）中获得一个字符串，并将其作为某事物的名称使用。文本框中的字符串通常都是可变字符串，会因为用户输出新的内容而发生变化。复制该字符串可以防止因意外的变化而产生不利影响。
>
> 那么，不保留对象又会怎样呢？有一种特殊的情况，叫做**保留死循环**（retain cycle），它会令引用计数器发生故障。如果两个实体是拥有和被拥有的关系，比如Car类和Engine类，则我们会让car对象来保留（拥有）engine对象，而不能反过来。Engine对象不应该保留包含了自己的car对象。如果car保留了engine对象，而engine对象也保留了car对象，那么这两个对象的引用计数的值永远不会归零，也永远不会被释放。除非engine对象释放了car对象，Car类的dealloc方法才会被调用，但如果car对象的dealloc方法没有调用的话，engine对象也不会被释放。它们都一直等待着对方先释放。所以一般的规则是所有者对象保留被拥有的对象，而不是被拥有者的对象保留所有者的对象。

下面给Car类添加一种新的特性，这样我们就可以使用一些新的属性语法了。这会是一个非常棒的功能。我们在car对象中添加一个新的实例变量name。这里用的还是原来的访问方法。首先来看一下Car.h文件的内容（新增的代码以粗体显示）。

```
@class Tire;
@class Engine;
@interface Car : NSObject
{
  NSString *name;
  NSMutableArray *tires;
  Engine *engine;
}
- (void)setName: (NSString *) newName;
- (NSString *) name;
- (void) setEngine: (Engine *) newEngine;
- (Engine *) engine;
-(void) setTire: (Tire *) tire atIndex: (int) index;
```

```
- (Tire *) tireAtIndex: (int) index;
- (void) print;
@end // Car
```

现在，我们添加访问方法的实现（请注意我们是在复制name变量），同时为car对象选择默认的名称（它会在print方法中输出）。

```
#import "Car.h"
@implementation Car
- (id) init
{
  if (self = [super init])
 {
  name = [[NSString alloc] initWithString:@"Car"];
  tires = [[NSMutableArray alloc] init];
  int i;
  for (i = 0; i < 4; i++) {
   [tires addObject: [NSNull null]];
  }
 }
 return (self);
} // init

- (void) dealloc
{
 [name release];
 [tires release];
 [engine release];
 [super dealloc];
} // dealloc

- (void)setName: (NSString *)newName
{
  [name release];
  name = [newName copy];
} // setName
- (NSString *)name
{
  return (name);
} // name

- (Engine *) engine
{
 return (engine);
} // engine

- (void) setEngine: (Engine *) newEngine
{
 [newEngine retain];
 [engine release];
 engine = newEngine;
} // setEngine

- (void) setTire: (Tire *) tire atIndex: (int) index
```

```
{
  [tires replaceObjectAtIndex: index withObject: tire];
} // setTire:atIndex:
- (Tire *) tireAtIndex: (int) index
{

 Tire *tire;
  tire = [tires objectAtIndex: index];
  return (tire);
} // tireAtIndex:
- (void) print
{
  NSLog (@"%@ has:", name);
 for (int i = 0; i < 4; i++)
  {
   NSLog (@"%@", [self tireAtIndex: i]);
  }
  NSLog (@"%@", engine);
} // print
@end // Car
```

然后我们在main()函数中设置name对象的值：

```
Car *car = [[Car alloc] init];
[car setName: @"Herbie"];
```

运行该程序，你将会在输出结果的开头部分看到汽车的名称。好了，我们开始向Car类添加属性。修改后的Car.h文件如下所示。

```
@class Tire;
@class Engine;
@interface Car : NSObject
{
  NSString *name;
  NSMutableArray *tires;
  Engine *engine;
}
@property (copy) NSString *name;
@property (retain) Engine *engine;
- (void) setTire: (Tire *) tire atIndex: (int) index;
- (Tire *) tireAtIndex: (int) index;
- (void) print;
@end // Car
```

你应该会注意到，访问方法的声明已经被@property声明所取代。你还可以声明具有其他特性的@property语句，表达你希望属性具有怎样的行为。因为name属性使用的是copy特性，所以编译器和类的使用者会知道name属性将被复制。这样程序员就知道自己无需复制文本框内的字符串。另一方面，可以对engine属性使用的只有保留和释放特性。如果你两者都没有使用的话，编译器默认会使用assign，这并不是你想要的结果。

> **说明**　你还可以使用一些其他的声明（如nonatomic），如果不在多线程中使用，这些声明可以提高访问方法的调用速度。台式机的速度已经够快了，因此使用nonatomic特性对于性能的提升实际上起不了多大作用，不过iOS程序员经常会使用这种技术，从而在资源有限的设备上获得更好的性能。如果你不想保留某个变量对象，可以使用assign特性，这样可以避免发生保留死循环。
>
> 如果你没有为属性指定任何特性，它们会默认使用nonatomic和assign。你也可以为可保留的指针（即Objective-C对象）指定retain和copy特性，而其他C类型和不可保留的指针必须使用assign特性并且要手动来管理内存。
>
> 如果你自己定义了setter或getter方法，那么就不能使用atomic特性了，必须使用nonatomic特性。

Car.m有两个重大变化，一个是name和engine的访问方法删除了，另一个是添加了两条@synthesize指令。

```
@implementation Car

@synthesize name;
@synthesize engine;
```

最后，main()函数使用了点表示法来给对象赋值。

```
Car *car = [[Car alloc] init];
car.name = @"Herbie";
…
car.engine = [[Slant6 alloc] init];
```

11.2.1　名称的使用

在本章的所有代码中，属性的名称始终与支持属性的实例变量名称相同。这种情况非常普遍，而且有可能将是你使用最多的情况。不过，有时你可能希望实例变量是一个名称，而公开的属性是另一个名称。

假设我们想要在Car类中使用其他名称（比如appellation）来调用实例变量，只需在Car.h文件中修改该实例变量的名称。

```
@interface Car : NSObject
{
  NSString *appellation;
  NSMutableArray *tires;
  Engine *engine;
}
@property (copy) NSString *name;
@property (retain) Engine *engine;
```

然后再修改@synthesize指令：

```
@synthesize name = appellation;
```

编译器仍将创建-setName:和-name方法，但在其实现代码中用的却是appellation实例变量。

不过这样做的话，编译的时候将会遇到一些错误。你可能还记得，我们直接访问的实例变量已经被修改了。我们可以选择用搜索并替换name的方式来解决，也可以将实例变量的直接调用改成用访问方法。比如在init方法中把

```
name = @"Car";
```

改成

```
self.name = @"Car";
```

self.name是什么意思呢？该语句的作用是消除歧义，使编辑器知道我们会用访问方式来读取变量。如果只使用普通的name，编译器会误以为我们要修改这个名为name的实例变量。如果要使用访问方法来进行赋值，可以写成[self setName:@"Car"]。请记住，点表示法只是调用相同方法的便捷方式，所以self.name = @"Car"只不过是实现同样内容的另一种写法。

最后，我们必须修改NSLog()函数的以下这句：

```
NSLog (@"%@ has:", self.name);
```

现在，我们还可以将appellation重命名为其他名字，比如nickname或moniker。只需修改实例变量的名称以及在@synthesize语句中使用的名称就可以了。

因为car对象的子类不必直接访问变量，所以我们可以把它们从头文件中删掉。我们可以删掉name和engine变量的声明，这样它们就会被编译器创建。也可以把tire数组放在实现文件中。这样头文件就会更加简洁（这样更好些）。

下面我们再回到AllWeatherRadial.m文件中。请仔细阅读，你会注意到在description方法中，我们仍然使用方法来获取tire变量的值。现在改成使用点表达式语法，description方法看起来应该如下所示。

```
- (NSString *) description
{
  NSString *desc;
  desc = [[NSString alloc] initWithFormat:
      @"AllWeatherRadial: %.1f / %.1f / %.1f / %.1f",
      self.pressure, self.treadDepth, self.rainHandling, self.snowHandling];

return (desc);

} // description
```

接下来我们将修改Tire类的pressure和treadDepth属性，还要删除getter和setter方法。被简化的Tire类并不会丢失任何功能。

```
@interface Tire : NSObject

@property float pressure;
@property float treadDepth;

- (id) initWithPressure: (float) pressure;
```

```
- (id) initWithTreadDepth: (float) treadDepth;

- (id) initWithPressure: (float) pressure
    treadDepth: (float) treadDepth;
@end // Tire
```

我们还删除了实现文件中的setter和getter方法。把多余的代码删掉是一个很愉悦的过程，出现bug的可能性也会更小。

顺便说一句，你可能还记得我们向Car类添加了name属性，它具有copy特性。这意味着当我们给name赋值的时候，Car类会创建这条字符串的副本并将其存储下来。如果我们没有启用ARC，则需要在Car类的dealloc方法中添加release调用，就像这样：

```
[name release];
```

如果我们启用了ARC，则不需要担心这个问题，编译器会为我们自动添加的。这就是ARC的优点。

11.2.2　只读属性

你可以使某个对象具有只读属性。这个属性可能是一个即时计算的值，比如香蕉的表面积，也可能是一个其他对象只能读取但无法更改的值，比如你的驾驶证号码。你可以使用@property的其他特性来处理这些情况。

默认情况下，属性是可变的（你可以读取也可以修改）。你可以使用属性的readwrite特性。由于属性默认是可读写的，因此你通常不会使用这个特性。但是，如果你想要表明自己的意图，可能会需要用到它。我们可以在Car.h文件中使用readwrite特性：

```
@property (readwrite, copy) NSString *name;
@property (readwrite, retain) Engine *engine;
```

不过我们不会这样做，因为我们一般都会尽量杜绝和消除冗余重复代码。

回到之前关于只读属性的讨论，假设我们拥有一个属性（比如驾驶证号码或鞋码），并且不想让任何人修改它，则可以对这个@property使用readonly特性，代码如下所示。

```
@interface Me : NSObject
{
  float shoeSize;
  NSString *licenseNumber;
}
@property (readonly) float shoeSize;
@property (readonly) NSString *licenseNumber;
@end
```

当编译器知道这个@property属性是只读的，它将只生成一个getter方法而不会生成setter方法。用户可以调用-shoeSize和-licenseNumber方法，但如果你调用了-setShoeSize:方法，编译器将会报错。使用点表达式也会得到同样的结果。

11.2.3 自己动手有时更好

我们之前提到过属性是基于变量的，并且编译器会为你创建getter和setter方法。但是如果你不想要变量、getter和setter方法的话应该怎么做呢？

这种情况下，你可以使用关键字@dynamic来告诉编译器不要生成任何代码或创建相应的实例变量。继续上一个示例，这次我们向类中添加了bodyMassIndex（身体质量指数）属性。

因为身体质量指数是由身高和体重计算得来的，所以我们不能存储这个值，而是创建一个能在运行时计算出此值的访问方法。我们使用@dynamic指令来指定这个属性并告诉编译器不需要去创建变量或getter方法——我们可以自己来。

```
@property (readonly) float bodyMassIndex;

@dynamic bodyMassIndex;
- (float)bodyMassIndex
{
 ///compute and return bodyMassIndex
}
```

如果你声明了dynamic属性，并且企图调用不存在的getter或setter方法，你将会得到一个报错。

我不喜欢这个方法名

有时你可能会不喜欢默认生成的方法名称。它们都是blah和setBlah:格式的。如果想要换掉它们，可以指定编译器生成的getter和setter方法的名称。使用getter=和setter=特性就可以自定义想要的方法名称。如果这样做的话，需要注意会破坏键/值规则（第19章中会详细介绍），因此除非有必须使用这些特性的原因，否则请尽量避免使用。

以下就是对布尔型的属性使用了这个特性的例子：

```
@property (getter=isHidden) BOOL hidden;
```

它告诉编译器生成名为isHidden的getter方法，并生成名为默认setHidden:的setter方法。

11.2.4 特性不是万能的

你可能已经注意到了，我们并没有转换tire对象的访问方法以支持属性。

```
- (void) setTire: (Tire *) tire atIndex: (int) index;
- (Tire *) tireAtIndex: (int) index;
```

这是因为这些方法并不适合属性所能涵盖的较小范围。属性只支持替代-setBlah和-blah方法，但是不支持那些需要接收额外参数的方法，例如car对象中tire对象的代码。

11.3 小结

本章主要介绍了属性。在为对象变量执行常见的操作时，利用属性可以减少需要编写以及随后需要阅读的代码数量。使用@property预编译指令可以告诉编译器："嘿，这个对象具有这些类

型的特性。”你还可以让属性传递其他信息，比如可变性（只读或读写）。编译器在后台会自动生成对象变量的setter和getter方法的声明语句。

使用@synthesize预编译指令可以通知编译器生成访问方法。你还可以控制由编译器生成的访问方法对哪些实例变量起作用。如果不想使用默认的行为，你完全可以编写自己的访问方法。你还可以使用@dynamic指令告诉编译器不要生成变量和代码。

尽管点表达式通常出现在有属性的代码中，但是它只是调用对象的setter和getter方法的一种便捷方式。点表示法减少了需要键入的字符数量，而且也进一步方便了曾经使用其他语言的编程人员。

下一章将讨论类别，类别是Objective-C允许你扩展现有的类（即使你没有这些类的源代码）的方式。千万不要错过了。



第 12 章 类 别 12

在编写面向对象的程序时，你经常想为现有的类添加一些新的行为。例如，你设计了一种新型轮胎，因此需要创建Tire类的子类并添加一些有趣的功能。为已经存在的类添加行为时，通常采用创建子类的方法。

不过有时子类并不方便。比如说，你想要为NSString类添加一个新的行为，但是NSString实际上只是一个类簇的表面形式，因而为这样的类创建子类会非常困难。在其他情况下，也许你可以创建它的子类，但是你用到的工具集和库是无法帮你处理新类的对象的。例如，当使用stringWithFormat:类方法生成新字符串时，你创建的NSString类的新子类就无法返回。

利用Objective-C的动态运行时分配机制，你可以为现有的类添加新方法。嘿，这听起来很酷！这些新方法在Objective-C里被称为*类别*（category）。

12.1 创建类别

类别是一种为现有的类添加新方法的方式。想为一个类添加新方法吗？请继续阅读下面的内容。你可以为任何类添加新的方法，包括那些没有源代码的类。

例如，你正在编写一个纵横字谜游戏程序，该程序将接收一系列的字符串，然后确定每个字符串的长度并存入NSArray数组或NSDictionary字典中。你需要先将每个长度值包装在一个NSNumber对象中，然后才能将其存入NSArray数组或NSDictionary字典中。

你可以按如下方式编写代码：

```
NSNumber *number;
number = [NSNumber numberWithUnsignedInt: [string length]];
// … do something with number
```

但是这样做你很快就会感到厌烦。其实你可以为NSString类添加一个类别，让它替你完成这项工作。下面开始行动吧。位于12.01 LengthAsNSNumber项目目录中的LengthAsNSNumber程序中包含了向NSString类添加这样一个类别的代码。

程序员总是习惯把类别代码放在独立的文件中，通常会以“类名称+类别名称”的风格命名。因此在我们的项目中，类别文件的名称为NSString+NumberConvenience。这并不是硬性规定，但却是一个好习惯。

12.1.1　开始创建类别

使用Xcode往项目中添加类别非常容易，它甚至会以我们刚才说到的规范给类别文件正确地命名。

要创建类别文件，就要打开项目，在导航栏中选择你想让文件出现的群组。接下来可以选择File ➤ New ➤ New File选项，也可以按下Command+N快捷键。在弹出的新文件窗口的左侧选择Cocoa并在右侧选择Objective-C category图标（如图12-1所示）。

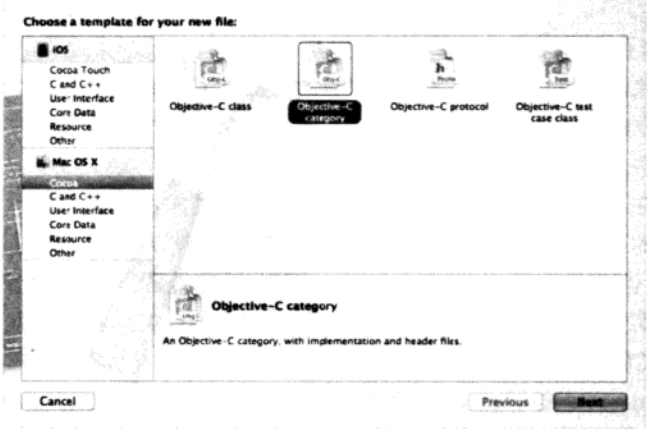


图12-1　创建类别文件

点击Next按钮，然后在下一个界面（如图12-2所示）的Category文本框中输入Number Convenience，并在Category on文本框中输入NSString。NSString是我们想要添加方法的类。完成后点击Next按钮继续。

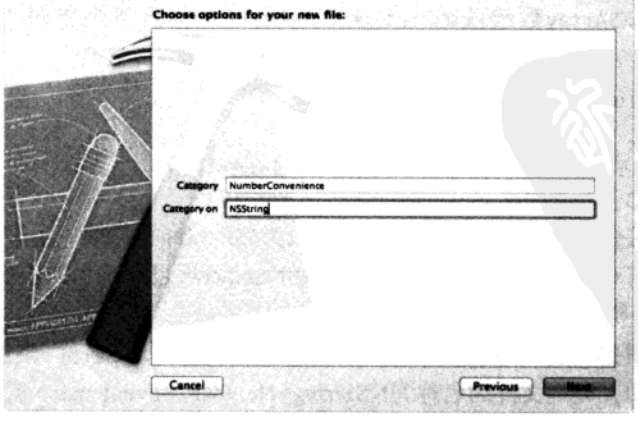


图12-2　输入类别名称和相关联的类

在下一个界面（如图12-3所示）中，选择文件存储的位置以及要添加到的目标和群组。通常你可以使用Xcode默认的选择。点击Create按钮之后，就有了一个类别头文件和一个用来添加方法代码的实现文件。

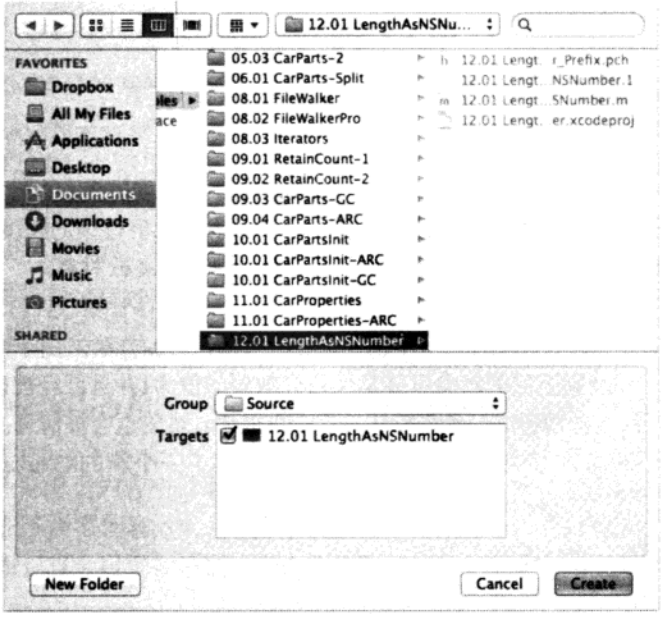


图12-3 指定文件存储位置以及源代码群组和目标

12.1.2 @interface 部分

类别的声明看起来非常像类的声明：

```
@interface NSString (NumberConvenience)
- (NSNumber *) lengthAsNumber;
@end // NumberConvenience
```

你应该注意到，该声明具有两个非常有趣的特点。首先，类的名称后面是位于括号中的新名称，这意味着类别叫做NumberConvenience，而且它是添加给NSString类的。换句话说就是："我们为NSString类添加了一个名为NumberConvenience的类别。"只要保证类别名称唯一，你可以向一个类中添加任意数量的类别。

其次，你可以指定想要添加类别的类（在本例中是NSString）和类别的名称（在本例中是NumberConvenience），还可以列出你要添加的方法，最后以@end语句结束。由于不能添加新的实例变量，因此这里不会像类声明那样包含实例变量的声明部分。

你可以在类别中添加属性，但是不能添加实例变量，而且属性必须是@dynamic类型的。添加属性的好处在于你可以通过点表达式来访问setter和getter方法。

12.1.3 @implementation部分

既然有@interface，肯定也会有@implementation。你可以在@implementation中实现自己的方法：

```
@implementation NSString (NumberConvenience)
- (NSNumber *) lengthAsNumber
{
    NSUInteger length = [self length];
    return ([NSNumber numberWithUnsignedInt:length]);
} // lengthAsNumber
@end // NumberConvenience
```

与类别的@interface部分相似，@implementation部分也包含类名、类别名以及新方法的实现。

lengthAsNumber方法中通过调用[self length]方法来获取字符串的长度值。向该字符串发送lengthAsNumber消息，就会创建出一个包含了长度值的新NSNumber对象。

我们先抽出几分钟时间，讨论一下我们一直在说的话题：内存管理。这段代码符合内存管理吗？答案是肯定的。numberWithUnsignedInt不同于alloc、copy和new方法。由于numberWithUnsignedInt不属于这3个方法，因此它返回的对象可能是保留计数器的值为1并且已经被设置为自动释放了。在当前活动的自动释放池被销毁时，我们创建的这个NSNumber对象也会被清理。

下面是使用了新类别方法的代码。main()函数创建了一个新的NSMutableDictionary类的对象，添加了3个字符串作为键，并将它们的长度值作为值。

```
int main(int argc, const char * argv[])
{
  @autoreleasepool
  {
    NSMutableDictionary *dict = [NSMutableDictionary dictionary];

    [dict setObject:[@"hello" lengthAsNumber]
         forKey:@"hello"];

    [dict setObject:[@"iLikeFish" lengthAsNumber]
         forKey:@"iLikeFish"];

    [dict setObject:[@"Once upon a time" lengthAsNumber]
         forKey:@"Once upon a time"];

    NSLog (@"%@", dict);
  }
  return 0;
}
```

我们像以前一样，逐步讲解main()函数。

首先向main()函数导入头文件，这样才能知道我们拥有这个为NSString类而定义的方法。在其他#import语句的最后，我们添加这样一句代码：

```
#import "NSString+NumberConvenience.h"
```

接下来，创建一个自动释放池，这是启用了ARC的方法。

```
@autoreleasepool {
```

不要忘记在代码的末尾合上花括号。所有的自动释放对象都会进入池子里。尤其是可变字典将会在这里被最终释放掉，包括通过类别创建出来的NSNumber对象。

在创建自动释放池以后，再创建一个新的可变字典。还记得吧，这个便利的Cocoa类可以让我们成对地存储键和对象。

```
NSMutableDictionary *dict = [NSMutableDictionary dictionary];
```

由于不能将像int这样的基本类型存入字典，因此必须使用像NSNumber这样的包装类。不过幸好我们的新类别可以将字符串长度轻松地放入NSNumber对象中。以下是使用@"hello"作为键将数值5添加进字典的方法。

```
[dict setObject: [@"hello" lengthAsNumber] forKey: @"hello"];
```

这行代码看起来有点奇怪，但是它的确能够完成任务。请记住@"字符串"这种形式的字符串实际上就是地地道道的NSString对象。它们对消息的响应正如任何其他NSString对象一样。因为我们已经为NSString类创建了这个类别，所以任何字符串（包括像这样的字面量类型）都能响应lengthAsNumber消息。

因为很重要，所以要反复强调。任何NSString对象都能响应lengthAsNumber消息，包括字面量字符串、description方法返回的字符串、可变字符串、其他工具集某部分字符串、文件中加载的字符串、从因特网海量内容中提取的字符串，等等。正是这种兼容性使类别成为了一种非常强大的概念。通过它不需要创建NSString类的子类就可以获得一种新的行为。

当你运行程序时，将会得到如下所示的结果。

```
{
"Once upon a time" = 16;
hello = 5;
iLikeFish = 9;
}
```

12.1.4 类别的缺陷

你可能已经完全陶醉于类别的强大功能了，让我们把你拉回现实吧。类别有两个局限性。第一是无法向类中添加新的实例变量。类别没有空间容纳实例变量。

第二个就是名称冲突，也就是类别中的方法与现有的方法重名。当发生名称冲突时，类别具有更高的优先级。你的类别方法将完全取代初始方法，导致初始方法不再可用。有些编程人员会在自己的类别方法名中添加一个前缀，以确保不会发生名称冲突。

> **说明** 也有一些技术可以克服类别无法增加新实例变量的局限。例如，使用全局字典来存储对象与你想要关联的额外变量之间的映射。但此时你可能需要认真考虑一下，类别是否是完成当前任务的最佳选择。

12.1.5　类别的优势

在Cocoa中，类别主要有3个用途：将类的实现代码分散到多个不同文件或框架中，创建对私有方法的前向引用，以及向对象添加非正式协议（informal protocol）。如果你还不理解非正式协议的含义，请不要担心，稍后将讨论这一概念。

12.1.6　类扩展

现在我们已经讨论完类别的缺陷与优势了，接下来再介绍一个特殊的类别，它有一个特殊的名称：*类扩展*（class extension）。这个类别的特点之一就是不需要名字。这是什么意思？我们之前都要为类别命名并且会在定义@interface部分的时候使用这个名字，而这个特殊的类扩展类别不需要命名，其特点如下。

- 正如我们之前指出的，它不需要名字。
- 你可以在包含你的源代码的类（也就是你自己的类）中使用它。
- 你可以添加实例变量。
- 你可以将只读权限改成可读写的权限。
- 创建数量不限。

观察一下项目12.02 ClassExtension。我们创建了一个名为Things的简单类，如下所示。

```
@interface Things : NSObject
@property (assign) NSInteger thing1;
@property (readonly, assign) NSInteger thing2;

- (void)resetAllValues;
@end
```

这个类包含两个属性和一个方法。我们将thing2属性标记为只读，在类的公共接口中可见的只有这些。不过实际上拥有的比这些要多，正如我们在实现文件中所看到的。首先看看下面这段代码。

```
@interface Things ()
{
  NSInteger thing4;
}
@property (readwrite, assign) NSInteger thing2;
@property (assign) NSInteger thing3;
@end
```

这看起来就像在定义一个类，只不过这里没有继承的父类。我们所做的基本上就是获取Things类，并通过添加私有属性和方法来扩展它。这就是它被称为类扩展的原因。

仔细看一下thing2属性，你可能注意到了，我们已经在头文件中定义过这个属性。那我们在这里对它做了什么？我们改变了它的读写权限，将它标记为readwrite，这样编译器就会生成setter方法了，不过它是只能在这个类中访问的私有方法，在公共的接口则只有getter方法。

我们还添加了私有属性thing3，它只可以在这个类内部使用。此外还添加了一个名为thing4的实例变量，它同样是私有的。

这段代码符合内存管理吗？是的，因为我们使用了ARC，所以不需要释放对象，就像它们会在自动释放池中最后被释放掉一样。没有内存泄漏的问题。

那么为什么要做这些事情？面向对象编程的一个特征就是信息隐藏。你只会把用户需要看到的展示出来，其他的则不需要，比如内部实现的细节。这些技术可以帮助你实现这一目标。

我们把这个类别放在.m文件中，也可以放在私有的头文件（ThingsPrivate.h）中，这样就可以让Things类的子类或友类访问这些内容了。

你可以拥有多个类扩展类别，不过这样会引发很难察觉的bug，所以请理智地使用。

运行这个程序会获得如下结果：

```
1   0   0
200 300 400
```

12.2 利用类别分散实现代码

正如在第6章中所见到的，你可以将类的接口放入头文件并将类的实现代码放入.m文件中，但是不能将@implementation分散到多个不同的.m文件中。如果想将大型的单个类分散到多个不同的.m文件中，可以使用类别。

以AppKit中的NSWindow类为例。如果查看NSWindow的文档，你将会发现该类有数百个方法。该类的文档若打印出来将超过60页（不要尝试在家里这么做）。

如果将NSWindow类的所有代码都组织在一个文件中，即使是Cocoa的开发团队也会觉得其过于庞大，难以驾驭，更不用说我们这些普通的开发人员了。如果查看一下该类的头文件并查找文字“@interface”，你将会看到官方的类接口

```
@interface NSWindow : NSResponder
```

以及大量的类别声明，其中包括以下这些。

```
@interface NSWindow(NSKeyboardUI)
@interface NSWindow(NSToolbarSupport)
@interface NSWindow(NSDrag)
@interface NSWindow(NSCarbonExtensions)
```

利用类别，所有的键盘用户界面代码都位于一个源文件中，工具栏代码位于另一个源文件中，拖放功能位于下一个源文件中，依次类推。类别还可以将方法分散到逻辑群组中，使编程人员可以更加容易地阅读头文件。这正是我们想要实现的小规模效果。

在项目中使用类别

位于12.03 CategoryThing文件夹中的CategoryThing项目有一个简单的类，该类被分散到几个不同的实现文件中。

首先是头文件CategoryThing.h，它包含类的声明和一些类别。该文件在开头先导入Foundation

框架，然后是带有3个整型实例变量的类声明。

```
#import <Foundation/Foundation.h>
@interface CategoryThing : NSObject
{
 NSInteger thing1;
 NSInteger thing2;
 NSInteger thing3;
}
@end // CategoryThing
```

类声明之后是3个类别声明，每个类别具有一个实例变量的setter访问方法。**我们将把这些实现代码放入不同的文件中。**

```
@interface CategoryThing (Thing1)
- (void)setThing1:(NSInteger)thing1;
- (NSInteger)thing1;
@end // CategoryThing (Thing1)

@interface CategoryThing(Thing2)
- (void)setThing2:(NSInteger)thing2;
- (NSInteger)thing2;
@end // CategoryThing (Thing2)

@interface CategoryThing (Thing3)
- (void)setThing3:(NSInteger)thing3;
- (NSInteger)thing3;
@end // CategoryThing (Thing3)
```

以上就是CategoryThing.h的代码。

CategoryThing.m文件相当简单，它包含了一个description方法，**我们可以按照NSLog()函数中的%d格式说明符的方式使用该方法。**

```
#import "CategoryThing.h"
@implementation CategoryThing
- (NSString *) description
{
NSString *desc;
desc = [NSString stringWithFormat: @"%d %d %d", thing1, thing2, thing3];
return (desc);
} // description
@end // CategoryThing
```

我们暂停一会儿来看一下内存管理问题。description方法能够处理好内存释放吗？答案是肯定的。因为stringWithFormat不同于alloc、copy和new，所以我们可以假设它返回的对象的保留计数器的值为1且已经被设置为自动释放。在当前的自动释放池被销毁时，该对象将被清理。

下面我们来看一下各个类别。Thing1.m文件包含了Thing1类别的实现：

```
#import "CategoryThing.h"

@implementation CategoryThing (Thing1)
- (void)setThing1:(NSInteger)t1
{
```

```
  thing1 = t1;
} // setThing1

- (NSInteger) thing1
{
  return (thing1);
} // thing1

@end // CategoryThing
```

值得一提的是，类别可以访问其继承的类的实例变量。类别的方法具有更高的优先级。

Thing2.m文件的内容与Thing1.m文件非常相似：

```
#import "CategoryThing.h"

@implementation CategoryThing (Thing2)

- (void)setThing2:(NSInteger) t2
{
  thing2 = t2;
} // setThing2

- (NSInteger)thing2
{
  return (thing2);
} // thing2

@end // CategoryThing
```

读到这里，你已经知道Thing3.m文件的内容该怎么写了吧（提示：可能会用到剪切、粘贴、查找和替换）。

main.m文件包含main()函数，实际上正是该函数使用了我们创建的类别。代码中首先是#import语句：

```
#import "CategoryThing.h"
```

我们需要先导入CategoryThing的头文件，以便编译器找到类的定义和类别。然后才是main()函数：

```
int main (int argc, const char *argv[])
{
@autoreleasepool {
 CategoryThing *thing = [[CategoryThing alloc] init];
 [thing setThing1: 5];
 [thing setThing2: 23];
 [thing setThing3: 42];
 NSLog (@"Things are %@", thing);
 }
return (0);
} // main
```

main()函数的第一行是已经令我们爱不释手的标准自动释放池代码。该自动释放池将用于存放NSLog()函数调用的description方法所返回的会自动释放的字符串。

接下来，一个CategoryThing类的对象被分配和初始化：

```
CategoryThing *thing = [[CategoryThing alloc] init];
```

我们有义务在创建新对象时对内存管理进行分析：因为这里使用的是alloc方法，该thing对象的保留计数器的值为1，它未被放入自动释放池中。不过因为这个项目使用了ARC特性，所以我们不需要担心在哪里添加release语句。我们的好帮手编译器会在正确的位置帮我们添加好的。

然后，一些消息被发送给该对象，以分别设置thing1、thing2和thing3的值：

```
[thing setThing1: 5];
[thing setThing2: 23];
[thing setThing3: 42];
```

在使用一个对象时，对象的方法是在接口中声明、在父类中声明还是在类别中声明的并不重要，也不会影响结果。

在thing对象被赋值之后，NSLog()函数将输出该对象中所有的值。正如CategoryThing类中的description方法一样，NSLog()函数将输出thing对象的3个实例变量的值：

```
NSLog (@"Things are %@", thing);
```

最终，自动释放池会被释放，而main()函数会返回0：

```
}
return (0);
} // main
```

以上就是我们的这个小程序。运行该程序将得到下面的结果：

```
Things are 5 23 42
```

不仅可以将一个类的实现代码分散到多个不同的源文件中，还可以分散到多个不同的框架中。NSString是Foundation框架中的一个类，包含了许多面向数据的类，如字符串、数字和集合。所有的视觉组件（如窗口、颜色和绘图等）都包含在AppKit和UIKit中。虽然NSString类是在Foundation框架中声明的，但是AppKit中有一个NSString的类别，称为NSStringDrawing，该类别允许你向字符串发送绘图消息。在绘制一个字符串时，该方法会将字符串的文本渲染到屏幕上。由于这是一种图形功能，所以它是AppKit框架中的方法。不过NSString又是Foundation框架的对象。Cocoa的设计人员通过类别使数据功能放在Foundation框架中，而绘图功能放在AppKit框架中。作为编程人员，我们直接使用NSString类即可，通常不必关心某个方法来自何处。

12.3　通过类别创建前向引用

前面提到过，Cocoa没有任何真正的私有方法。如果你知道对象支持的某个方法的名称，即使该对象所在的类的接口中没有声明该方法，你也可以调用它。

不过，编译器会尽力提供帮助。如果编译器发现你调用对象的某个方法，但是却没有找到该方法的声明或定义，那么它将发出这样的错误提示：warning:'CategoryThing' may not respond to '-setThing4:'。通常情况下，这种错误提示是有益的，因为它将有助于你捕捉许多输入错误。

不过，如果你的某些方法的实现使用了在类的@interface部分未列出的方法，编译器的这种

警惕性会带来一些问题。对于为什么不想在类的@interface部分列出自己的全部方法，有许多充分的理由。这些方法可能是纯粹的实现细节，你可能将根据方法的名称来确定要使用哪个方法。但是不管怎样，如果你在使用自己的方法之前没有声明它们，编译器就会提出警告。而修复掉所有的编译器警告是一种良好的习惯，那么该如何去做呢？

如果能够先定义一个方法然后再使用它，编译器将会找到你的方法定义，因而不会产生警告。但是，如果不方便这样做，或者你使用的是另一个类中尚未发布的方法，那么就需要采取其他措施。

通过类别来救急

只要在类别中声明一个方法，编译器就会表示："好了，该方法已经存在，如果遇到编程人员使用该方法，我不会提出警告。"实际上，如果你不想实现这个方法的话，放着不管就可以了。

常用的策略是将类别置于实现文件的最前面。假设Car类有一个名为rotateTires的方法。我们可以使用另一个名为moveTireFromPosition:toPosition:的方法来实现rotateTires方法，以交换两个位置的轮胎。第二个方法是实现细节，而不是将其放在Car类的公共接口中。通过在类别中声明moveTireFromPosition:toPosition:方法，rotateTires方法可以使用它，但不会引发编译器产生警告。该类别如下所示：

```
@interface Car (Private)
- (void) moveTireFromPosition: (int) pos1 toPosition: (int) pos2;
@end // Private
```

当你实现moveTireFromPosition:toPosition:方法时，它不一定要出现在@implementation Car(PrivateMethods)代码块中，不过如果没有的话，编译器会给你一个警告，所以良好的习惯是把它们放入各自的@implementation中。这样如果以后你需要查找这些方法，可以很容易就找到。你可以把这个方法放在@implementation Car部分中实现，这样你可以将方法分散到自己的类别中，以便于组织和生成文档。同时，你仍然可以在实现文件中将自己所有的方法组织在一起。

当访问其他类的私有方法时，你甚至不必提供该方法的实现。只要在类别中声明这些方法，编译器就不会产生警告。顺便提一下，实际上你不应该访问其他类的，但有时你必须解决Cocoa或其他人代码中的bug，或者编写测试代码，这时你不得不这么做。

需要注意，苹果公司官网拥有向Mac和iOS的App Store发布应用程序的指导方针（guideline），而其中一条就是应用程序不能访问类里面的私有变量和方法。如果你的应用程序有这样的行为，那么苹果公司肯定会拒绝让它上架的。

12.4 非正式协议和委托类别

现在我们要讨论在面向对象编程中经常会遇到的术语和概念，就是那些听起来比实际上复杂得多的东西。

Cocoa中的类经常会使用一种名为委托（delegate）的技术，委托是一种对象，由另一个类请求执行某些工作。比方说，当在应用程序启动时，AppKit的NSApplication类会询问其委托对象

是否应该打开一个无标题窗口。NSWindow类的对象会询问它们自己的委托对象是否允许关闭某个窗口。

最常见的情况是，编写委托对象并且将其提供给其他一些对象，通常是Cocoa框架中的对象。通过实现特定的方法，你可以控制Cocoa中对象的行为。

Cocoa中的滚动列表是由AppKit中的NSTableView类处理的。当tableView对象准备好执行某些操作（比如选择用户刚刚点击的行）时，它询问其委托对象是否能选择此行。tableView对象会向其委托对象发送一条消息：

```
- (BOOL) tableView: (NSTableView *) tableView shouldSelectRow: (NSInteger) rowIndex;
```

委托方法可以查看tableView对象和第几行并确定是否能够选择该行。如果表中包含了不该选择的行，则委托对象会告诉我们这些行是无法选择的。

12.4.1　ITunesFinder项目

支持你查找由Bonjour（该技术以前被称为Rendezvous）发布的网络服务的Cocoa类是NSNetServiceBrowser。你可以告诉网络服务浏览器你需要的服务并为其提供一个委托对象。浏览器对象将会向该委托对象发送消息，告知其发现新服务的时间。

ITunesFinder程序位于12.04 iTunesFinder项目文件夹中，它使用NSNetServiceBrowser列出了其所能找到的所有共享音乐库。

对于这一项目，我们从位于main.m文件中的main()函数开始讨论。委托对象是ITunesFinder类的一个实例变量，因此我们需要导入相应的头文件：

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

然后是main()函数。我们首先要设置自动释放池：

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

接下来，创建一个新的NSNetServiceBrowser对象：

```
NSNetServiceBrowser *browser = [[NSNetServiceBrowser alloc] init];
```

并且创建ITunesFinder类的一个新对象：

```
ITunesFinder *finder = [[ITunesFinder alloc] init];
```

因为这些对象是使用alloc方法创建的，所以要确保在程序结束的时候释放它们。由于我们启用了ARC特性，因此不需要自己来添加release语句。假如你没有启用ARC的话，请手动释放。

下面告诉网络服务浏览器使用ITunesFinder类的对象作为委托对象：

```
[browser setDelegate: finder];
```

然后告诉浏览器去搜索iTunes共享：

```
[browser searchForServicesOfType: @"_daap._tcp" inDomain: @"local."];
```

字符串"_daap._tcp"告诉了网络服务浏览器使用TCP网络协议去搜索DAAP（Digital Audio Access Protocol，数字音频访问协议）类型的服务。该语句可以找出由iTunes发布的库。域“local.”表示只在本地网络中搜索该服务。互联网编号分配机构（Internet Assigned Numbers Authority，IANA）维护着一个互联网协议族列表，该列表通常被映射为Bonjour服务名称。

接下来，main()函数会在启动一个run循环之前输出预示信息：

```
NSLog (@"begun browsing");
[[NSRunLoop currentRunLoop] run];
```

run循环是一种Cocoa概念，它在等待某些事情发生之前一直处于阻塞状态，即不执行任何代码。在本示例中，它等待的事情是指网络服务浏览器发现了新的iTunes共享。

除了监听网络流量以外，run循环还处理像等待用户事件（如按键或鼠标单击）之类的其他操作。事实上，run方法将一直保持运行而不会返回，因此其后的代码永远也不会被执行。不过我们还是留下了这段代码，以便浏览代码的人能够知道我们一直在进行适当的内存管理（我们可以构造一个只运行特定次数的run循环，但是那样的代码会更复杂，而且对我们讨论委托也没有益处）。因此，下面的清理代码实际上将不会执行。

```
}
return (0);
} // main
```

现在，我们已经构造了网络服务浏览器和run循环。浏览器发送查找特定服务的网络数据包，而返回的数据包表示“我在这里”。当这些数据包返回时，run循环告诉网络服务浏览器“这里有一些你的数据包”。然后，浏览器查看这些数据包。如果数据包来自浏览器以前未曾见过的服务，则浏览器将给委托对象发送消息，告诉它发现了新的服务。

现在该研究一下ITunesFinder委托的代码了。ITunesFinder类的接口最为简单：

```
#import <Foundation/Foundation.h>
@interface ITunesFinder : NSObject <NSNetServiceBrowserDelegate>
@end // ITunesFinder
```

请记住，我们并不一定要在@interface中声明方法。委托对象只需实现已经打算调用的方法。

需要注意我们在NSObject后面使用了<NSNetServiceBrowserDelegate>协议。它告诉了编译器和其他对象，ITunesFinder类符合这个名称的协议并且实现了其方法（更多内容详见下一章）。现在我们只需要添加这句代码，以防止编译器发出警告。

该实现包含两个方法。首先是一些起始代码：

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

然后是第一个委托方法：

```
- (void) netServiceBrowser: (NSNetServiceBrowser *) b
     didFindService: (NSNetService *) service
       moreComing: (BOOL) moreComing {
 [service resolveWithTimeout: 10];

 NSLog (@"found one! Name is %@",
```

```
  [service name]);
} // didFindService
```

当NSNetServiceBrowser对象发现一个新服务时，它将向委托对象发送netService-Browser:didFindService:moreComing:消息。浏览器被作为第一个参数（与main()函数中browser变量的值相同）传递。如果有多个服务浏览器在同时进行搜索，你可以利用参数来分析哪个浏览器发现了新的服务。

作为第二个参数传递的NSNetService类的对象，描述了被发现的服务（例如iTunes共享）。最后一个参数moreComing，用于标记一批通知是否已经完成。为什么Cocoa的设计人员要包含moreComing参数？如果你在具有100个iTunes共享的大型校园网络中运行该程序，则该委托方法被调用的前99次中，moreComing参数的值都为YES，最后一次调用时为NO。这一信息有助于弄清何时构造用户界面，因此你可以知道何时需要更新窗口内容。随着新的iTunes共享的不断变化，该方法被一次又一次地调用。

[service resolveWithTimeout: 10]告诉Bonjour系统获取关于该服务的所有有趣的属性。我们尤其希望获得共享的名称，如Scott's Groovy Tunes，从而可以输出该名称。[service name]为我们获取该共享的名称。

随着人们关闭他们的笔记本电脑或离开网络，iTunes共享会不断变化。ITunesFinder类实现了第二个委托方法，该方法在某个网络服务消失时被调用。

```
- (void) netServiceBrowser: (NSNetServiceBrowser *) b
      didRemoveService: (NSNetService *) service
         moreComing: (BOOL) moreComing
{
 [service resolveWithTimeout: 10];

 NSLog (@"lost one! Name is %@",
  [service name]);

} // didRemoveService
```

该方法与didFindService方法非常相似，只不过它是在某个服务不再可用时被调用。

现在运行该程序，看看会出现什么结果。Waqar的家庭网络中有一台陈旧的、名为Waqar Malik's Home Library的Mac mini电脑，该计算机在屋子内共享了iTunes音乐。该程序会产生这样的输出结果：

```
begun browsing
found one! Name is Waqar Malik's Home Library
```

我们在一台名为indigo的笔记本电脑上启动了iTunes，并且共享了资源库中的音乐。

```
found one! Name is indigo
```

在关闭笔记本电脑上的iTunes以后，ITunesFinder会告诉我们：

```
lost one! Name is indigo
```

12.4.2　委托和类别

好了，那么所有这些委托对象与类别有什么关系呢？委托强调类别的另一种应用：被发送给

委托对象的方法可以声明为一个NSObject的类别。NSNetService委托方法的部分声明如下。

```
@interface NSObject (NSNetServiceBrowserDelegateMethods)
- (void) netServiceBrowserWillSearch:(NSNetServiceBrowser *) browser;
- (void) netServiceBrowser:(NSNetServiceBrowser *) aNetServiceBrowser
 didFindService:(NSNetService *) service moreComing: (BOOL) moreComing;
- (void) netServiceBrowserDidStopSearch:(NSNetServiceBrowser *) browser;
- (void) netServiceBrowser:(NSNetServiceBrowser *) browser didRemoveService:
 (NSNetService *) service moreComing: (BOOL) moreComing;
@end
```

通过将这些方法声明为NSObject的类别，NSNetServiceBrowser的实现可以将这些消息之一发送给任何对象，无论这些对象实际上属于哪个类。这也意味着，只要对象实现了委托方法，任何类的对象都可以成为委托对象。

> **说明** 通过这种方式创建NSObject的类别，任何类的对象都可以作为委托对象使用。既不需要从特定的serviceBrowserDelegate类中继承（如在C++中那样），也不需要符合某个特定的接口（如在Java中那样）。

创建一个NSObject的类别称为“创建一个非正式协议”。大家都知道，计算机术语中的“协议”是一组管理通信的规则。非正式协议只是一种表达方式，它表示“这里有一些你可能希望实现的方法，你可以使用它们更好地完成工作”。在NSNetServiceBrowserDelegateMethods非正式协议中还有一些方法，我们在ITunesFinder类中没有实现。没关系，使用非正式协议，你可以只实现你想要的方法。

你可能已经猜到，应该还有一个正式协议的概念，我们将在下一章讨论。

12.4.3 响应选择器

你可能想知道：“NSNetServiceBrowser如何知道其委托对象是否能够处理那些发送给它的消息？”当试图发送一个对象无法理解的消息时，你可能已经遇到过Objective-C的运行时错误：

```
-[ITunesFinder addSnack:]: selector not recognized
```

那么，NSNetServiceBrowser是如何逃避这一问题的呢？其实，它并没有逃避这一问题。NSNetServiceBrowser首先检查对象，询问其能否响应该选择器。如果该对象能够响应该选择器，NSNetServiceBrowser则给它发送消息。

什么是选择器（selector）呢？选择器只是一个方法名称，但它以Objective-C运行时使用的特殊方式编码，以快速执行查询。你可以使用@selector()编译指令圆括号中的方法名称来指定选择器。因此，Car类的setEngine:方法的选择器是：

```
@selector(setEngine:)
```

而Car类的setTire:atIndex:方法的选择器如下所示：

```
@selector(setTire:atIndex:)
```

NSObject提供了一个名为respondsToSelector:的方法，该方法询问对象以确定其是否能够响应某个特定的消息。下面的代码段使用了respondToSelector:方法：

```
Car *car = [[Car alloc] init];
if ([car respondsToSelector: @selector(setEngine:)])
{
 NSLog (@"yowza!");
}
```

这段代码将输出“yowza!”，因为Car类的对象确实能够响应该setEngine:消息。

现在，我们来看看下面这段代码：

```
ITunesFinder *finder = [[ITunesFinder alloc] init];
if ([finder respondsToSelector:@selector(setEngine:)])
{
 NSLog (@"yowza!");
}
```

这次未能输出“yowza!”，因为ITunesFinder类的对象没有setEngine:方法。

为了确定委托对象能否响应消息，NSNetServiceBrowser将调用respondsToSelector:@selector(netServiceBrowser:didFindService:moreComing:)。如果该委托对象能够响应给定的消息，则浏览器向该对象发送此消息。否则，浏览器将暂时忽略该委托对象，继续正常运行。

12.4.4　选择器的其他应用

选择器可以被传递，可以作为方法的参数使用，甚至可以作为实例变量被存储。这样可以生成一些非常强大和灵活的构造。

Foundation框架中的NSTimer就是一个这样的类，它能够反复地向一个对象发送消息，当你希望在游戏中使怪物定期向玩家移动时，这样做非常方便。当创建NSTimer类的一个新对象时，你可以指定希望NSTimer向其发送消息的对象，并指定一个选择器表明希望被NSTimer调用的方法。例如，你可以创建一个定时器来调用游戏引擎中的moveMonsterTowardPlayer:方法，也可以再创建一个定时器来调用animateOneFrame:方法。

12.5　小结

本章介绍了类别。类别可以向现有的类添加新方法，即使你没有这些类的源代码。

除了可以向现有的类添加新功能以外，类别还可以将对象的实现代码分散到多个不同的源文件甚至多个不同的框架中。例如，NSString类的数据处理方法在Foundation框架中实现，而绘图方法则分放在UIKit和AppKit框架中。

利用类别可以声明非正式协议。非正式协议是NSObject的一个类别，它列出了对象可以响应的方法。非正式协议用于实现委托，委托是一种允许你轻松定制对象行为的技术。另外，我们还学习了选择器，通过选择器可以在代码中指定特定的Objective-C消息。

下一章将介绍Objective-C的协议，这些正式协议与非正式协议类似但功能更强大。

第 13 章 协 议 13

在第12章中，我们讨论了类别和非正式协议的奇妙之处。正如第12章所述，在使用非正式协议时，可以只实现你想要获得响应的方法。我们只实现了第12章中NSNetServiceBrowser委托对象的两个方法，这两个方法分别在新的服务加入或离开网络时被调用，而不必实现NSNetServiceBrowserDelegate非正式协议中的其他6个方法。我们也不必在对象中声明任何内容以表示该对象可用作NSNetServiceBrowser委托对象。所有这些任务可以用最少的代码完成。

你可能已经猜到了，Objective-C和Cocoa还有一个正式协议的概念，本章我们就看看正式协议是如何工作的。

13.1 正式协议

与非正式协议一样，正式协议是包含了方法和属性的有名称列表。但与非正式协议不同的是，正式协议要求显式地采用。采用（adopt）协议的办法是在类的@interface声明中列出协议的名称。采用协议后，你的类就要遵守该协议（你本以为自己是个不妥协主义者）。采用协议就意味着你承诺实现该协议的所有方法。否则，编译器会生成警告来提醒你。

> **说明** 正式协议就像Java的接口一样。事实上，Objective-C的协议正是受了Java接口的启发。

为什么要创建或采用正式协议呢？实现协议的每一个方法似乎都需要完成大量的工作。依赖协议的话，我们有时还可能会更忙一些。但是，通常情况下，一个协议只有少数几个需要实现的方法，你必须实现所有这些方法才能获得一系列有用的功能。因此，一般来说，正式协议的要求并不是一种负担。Objective-C 2.0中增加了一些良好的特性，使我们可以更轻松地使用协议，我们将在本章的末尾讨论这些特性。

13.1.1 声明协议

我们来看一下由Cocoa声明的一个协议，NSCopying。如果你采用了NSCopying协议，你的对象将会知道如何创建自身的副本。

```
@protocol NSCopying
- (id) copyWithZone: (NSZone *) zone;
@end
```

声明协议的语法看起来和声明类或类别的语法有点像，不过这里使用的不是@interface，而是使用@protocol告诉编译器："下面将是一个新的正式协议。"@protocol后面是协议名称，协议名称必须要唯一。

你也可以继承父协议，这点与继承父类相似。在声明语句协议名称后面的尖括号内可以指定父协议的名称。

```
@protocol MySuperDuberProtocol <MyParentProtocol>
@end
```

第一行代码表示MySuperDuperProtocol协议继承于MyParentProtocol协议，因此你必须实现两个协议中所有需要实现的方法。你通常可以使用NSObject作为根协议。请不要将其与NSObject类混淆。NSObject类符合NSObject协议，这意味着所有的对象都符合NSObject协议。将NSObject协议作为你编写的协议的父协议是一个不错的方式。

接下来是一个方法声明列表，所有采用了此协议的类都必须实现这些方法。协议声明以@end结束。在协议中不会引入新的实例变量。

我们再来看一个例子。这是Cocoa的NSCoding协议：

```
@protocol NSCoding
- (void) encodeWithCoder: (NSCoder *) encoder;
- (id) initWithCoder: (NSCoder *) decoder;
@end
```

当某个类采用NSCoding协议时，便意味着该类承诺将实现这两个方法。encodeWithCoder:方法用于接受对象的实例变量并将其转换为NSCoder类的对象。initWithCoder:方法从NSCoder类的对象中提取经过转换雪藏的实例变量，并使用它们去初始化新的对象。这两个方法总是成对实现的。如果你从未将一个对象转换为另一个新对象，那么对该对象进行编码是毫无意义的；如果你不对对象进行编码，那么也就不能创建新的对象。

13.1.2　采用协议

要采用某个协议，你可以在类的声明中列出该协议的名称，并用尖括号括起来。例如，Car类想要采用NSCopying协议，则其类的声明会如下所示。

```
@interface Car : NSObject <NSCopying>
{
// instance variables
}
// methods
@end // Car
```

而如果Car类要同时采用NSCopying和NSCoding这两个协议，则其类声明将如下所示。

```
@interface Car : NSObject <NSCopying, NSCoding>
{
```

```
// instance variables
}
// methods
@end // Car
```

你可以按任意顺序列出这些协议，这对结果没有任何影响。

采用了某个协议，就相当于给阅读该类声明的编程人员传递了一条信息，表明该类的对象可以完成两个非常重要的操作：一是能够对自身进行编码或解码，二是能够创建自身的副本。

13.1.3 实现协议

关于协议需要知道的就这么多（这里省略了声明变量时的一些语法细节，后面将讨论这些内容）。我们准备用大量篇幅来练习如何为CarParts项目采用NSCopying协议。

13.2 复制

让我们一起复习一下内存管理的规则："如果你使用alloc、copy或new方法获得了一个对象，则该对象的保留计数器的值为1，而且你要释放它。"我们已经学过了alloc和new方法，但是还未讨论copy方法。既然叫做copy方法，就肯定能创建对象的副本。copy消息会告诉对象创建一个全新的对象，并让新对象与接收copy消息的原对象完全一样。

现在扩展CarParts项目，这样就可以直接创建car对象的副本了（我们一直在等待这个激动人心的时刻）。这段代码位于13.01 CarParts-Copy项目文件夹内。整个编码过程中，我们还将讨论在实现副本创建代码时会遇到的一些有趣的细节问题。

复制的种类

事实上，你可以使用多种方法复制对象。大多数对象都引用（即指向）其他对象。如果你使用的是浅层复制（shallow copy），那么不会复制所引用的对象，新复制的对象只会指向现有的引用对象。当复制一个NSArray类的对象时，复制的对象只会复制指向引用对象的指针，而不会复制引用对象本身。如果复制了一个包含了5个NSString对象的NSArray对象，你最终得到的是5个可供程序使用的字符串对象，而不是10个。这种情况下每个对象最终都会指向一个字符串。

另一方面，深层复制（deep copy）将复制所有的引用对象。如果NSArray的copy方法是深层复制，则在复制操作完成以后得到10个字符串对象。对于CarParts项目，我们需要使用深层复制技术。这样当复制一个car对象时，你可以更改其引用的一个值（如轮胎压力），而不必担心所有car对象的轮胎压力都会更改。

你可以根据特定的类的需要，自由混搭深层复制和浅层复制两种技术。

要复制一个car对象，还需复制engine和tire对象，就让我们从复制engine对象开始吧！

13.2.1　复制Engine

我们将会处理的第一个类是Engine。为了能够复制engine对象，Engine类需要采用NSCopying协议。下面是Engine类的新接口：

```
@interface Engine : NSObject <NSCopying>
@end // Engine
```

因为Engine类采用了NSCopying协议，所以我们必须实现copyWithZone:方法。zone是NSZone类的一个对象，指向一块可供分配的内存区域。当你向一个对象发送copy消息时，该copy消息在到达你的代码之前会被转换为copyWithZone:方法。虽然NSZone类以前的作用比现在还要强大，不过我们仍然只使用它的一小部分功能。

Engine类的copyWithZone:方法的实现如下：

```
implementation:
- (id) copyWithZone: (NSZone *) zone
{
    Engine *engineCopy;
    engineCopy = [[[self class]
    allocWithZone: zone] init];
    return (engineCopy);
} // copyWithZone
```

由于Engine类没有实例变量，因此我们必须创建一个新的engine对象。不过这事说起来容易做起来难。看看engineCopy对象右边的语句有多复杂。消息的发送嵌套深度多达3层！

copyWithZone:方法的首要任务是获得self参数所属的类，然后向self对象所属的类发送allocWithZone:消息，以分配内存并创建一个该类的新对象。最后，copyWithZone:方法给这个新对象发送allocWithZone:消息使其初始化。下面我们来讨论为什么要使用复杂的消息嵌套，尤其是[self class]这个方法。

回想一下，alloc是一个类方法。由于allocWithZone:方法的声明是以加号开头的，因此它也是一个类方法。

```
+ (id) allocWithZone: (NSZone *) zone;
```

我们需要将该消息发送给一个类，而不是一个实例变量。那么应该发送给哪个类呢？直觉告诉我们是Engine类，像下面这样：

```
[Engine allocWithZone: zone];
```

这行代码适用于Engine类，但不适用于Engine类的子类。为什么呢？考虑一下Engine类的子类Slant6。如果给一个Slant6类的对象发送copy消息，因为我们最后使用的是Engine类的复制方法，所以这行代码最终会在Engine类的copyWithZone:方法中结束。而如果直接给Engine类发送allocWithZone:消息，则将创建一个新的Engine类的对象，不是Slant6类的对象。假如给Slant6类增加了一些实例变量，情况会变得更加扑朔迷离。这样做的话，Engine类的对象将无法容纳多余的变量，从而导致内存溢出错误。

现在，你可能已经明白了我们为什么要使用[self class]。通过使用[self class]，allocWithZone:消息将会被发送给正在接收copy消息的对象所属的类。如果self是一个Slant6的对象，则这里将创建一个Slant6类的新对象。如果我们的程序在将来添加了一些全新的发动机品牌，新的发动机对象也会被正确地复制。

allocWithZone:方法的最后一行会返回新创建的对象。

我们再来检查一下内存管理问题。copy方法应该会返回一个保留计数器的值为1的对象，且该对象不会自动释放。但这个新对象是我们通过alloc方法获得的，alloc方法总是会返回一个保留计数器的值为1且不需要释放的对象，因此这个方法的内存管理没有问题。

这正是Engine类可以被复制的原因。我们不需要触及Slant6类，因为Slant6类没有添加任何实例变量，所以在执行复制操作时不必完成任何额外工作。由于继承机制和创建对象时[self class]技术的使用，Slant6类的对象也可以被复制。

13.2.2 复制Tire

Tire类比Engine类更加难以复制。Tire类有两个实例变量（pressure和treadDepth）需要被复制到Tire类的新对象中，而且AllWeatherRadial子类又引入了另外两个实例变量（rainHandling和snowHandling），这两个变量也需要被复制到新对象中。

首先来看一下Tire类，其采用了协议的接口代码如下：

```
@interface Tire : NSObject <NSCopying>
@property float pressure;
@property float treadDepth;

// … methods

@end // Tire
```

下面是copyWithZone:方法的实现：

```
- (id) copyWithZone: (NSZone *) zone
{
 Tire *tireCopy;
 tireCopy = [[[self class] allocWithZone: zone] initWithPressure:
       pressure treadDepth: treadDepth];
 return (tireCopy);
} // copyWithZone
```

你在该实现中可以看到[[self class] allocWithZone: zone]这种形式，就像在Engine类中一样。由于在创建对象时必须调用init方法，所以我们可以方便地使用Tire类的initWithPressure:treadDepth:方法，将新的tire对象的pressure和treadDepth设置为我们正在复制的tire对象的值。该方法正好是Tire类的指定初始化函数，但并不是必须要使用指定初始化函数来执行复制操作。只要你愿意，也可以使用简单的init方法和访问器方法来修改对象的属性。

方便的指针

你可以像下面这样使用C语言风格的指针运算符直接访问实例变量。

```
tireCopy->pressure = pressure;
tireCopy->treadDepth = treadDepth;
```

一般来说，当设置属性不太可能涉及额外工作时，我们尽量使用init方法和访问器方法。

下面，我们来讨论AllWeatherRadial类，它的接口内容保持不变。

```
@interface AllWeatherRadial : Tire
// … properties
// … methods
@end // AllWeatherRadial
```

等一下，<NSCopying>哪里去了？你并不需要它了，也许你能想通其中的原因。当AllWeatherRadial类继承于Tire类时，它便已经获得了Tire类的所有属性，包括对NSCopying协议的遵守。

不过，我们需要重写copyWithZone:方法，因为我们必须确保AllWeatherRadial类中的rainHandling和snowHandling这两个实例变量被复制。

```
- (id) copyWithZone: (NSZone *) zone
{
 AllWeatherRadial *tireCopy;
 tireCopy = [super copyWithZone: zone];
 tireCopy.rainHandling = rainHandling;
 tireCopy.snowHandling = snowHandling;
 return (tireCopy);
} // copyWithZone
```

因为AllWeatherRadial是一个可以复制的类的子类，所以它既不需要实现allocWithZone:方法，也不需要使用前面曾经用过的[self class]形式。AllWeatherRadial类只需请求其父类执行copy操作，并期望父类正确地复制以及在分配对象时使用[self class]技术。因为Tire类的copyWithZone:方法会使用[self class]来确定要复制的对象所属的类，所以该方法将创建一个AllWeatherRadial类的新对象，这正是我们所期望的。该方法还替我们复制了pressure和treadDepth的值。这样是不是更方便了？

剩下的工作就是设置rainHandling和snowHandling这两个实例变量的值，访问器方法完全可以胜任这个工作。

13.2.3　复制Car

由于我们已经能够复制Engine、Tire及其子类，下面就来复制Car类本身。

如你所料，Car类需要采用NSCopying协议。

```
@interface Car : NSObject <NSCopying>
// properties
// … methods
@end // Car
```

若要遵守NSCopying协议，Car类必须实现我们原来的友元方法"copyWithZone:"。下面是Car类的copyWithZone:方法的实现。

```
- (id) copyWithZone: (NSZone *) zone
{
 Car *carCopy;
 carCopy = [[[self class] allocWithZone: zone] init];
 carCopy.name = self.name;
 Engine *engineCopy;
 engineCopy = [[engine copy] autorelease];
 carCopy.engine = engineCopy;
 for (int i = 0; i < 4; i++)
 {
   Tire *tireCopy;
   tireCopy = [[self tireAtIndex: i] copy];
   [tireCopy autorelease];
   [carCopy setTire: tireCopy atIndex: i];
 }
 return (carCopy);
} // copyWithZone
```

上面的copyWithZone:方法只比我们以前编写的多了一点代码，但所有这些代码都与你曾经见过的类似。

首先，通过给正在接收copy消息的对象所属的类发送allocWithZone:消息以分配一个新的car对象。

```
Car *carCopy;
carCopy = [[[self class] allocWithZone: zone] init];
```

虽然目前CarParts-copy项目中不包含Car类的子类，但有朝一日Car类可能会有新的子类。你永远无法知道何时会有人发明一台时光旅行汽车。我们只能通过使用self所属的类来分配新对象以防止未来可能出现的问题，就像我们之前所做的那样。

我们需要复制car对象的名称：

```
carCopy.name = self.name;
```

请记住，name属性复制其字符串对象，因此新的car对象将拥有正确的名称。

接下来，复制engine对象，并通知carCopy使用复制的engine对象作为自己的engine属性。

```
Engine *engineCopy;
engineCopy = [[engine copy] autorelease];
carCopy.engine = engineCopy;
```

engine对象为什么要自动释放？有必要这样处理吗？让我们再次仔细地考虑一下内存管理问题。[engine copy]将返回一个保留计数器的值为1的对象。setEngine:方法将保留接收到的engine对象，并将其保留计数器的值增加为2。当carCopy（最终）被销毁时，Car类的dealloc方法将释放engine对象，因此它的保留计数器的值又减少为1。到这些事件发生时，这段代码已经运行了很长时间，因此没有人会出来为其发送最后的release消息使其被销毁。如果那样的话，该engine对象将发生内存泄漏。通过自动释放engine对象，其保留计数器的值将在未来某个时间

自动释放池被销毁时减少1。

我们能够使用简单的[engineCopy release]替代engine对象的自动释放吗？当然可以。不过，你必须在setEngine:方法被调用以后再释放该engineCopy对象。否则，engineCopy对象可能在使用之前就被销毁了。采用哪种方式来管理内存取决于你自己的喜好。有些编程人员喜欢将内存清理的代码集中组织到函数中的某个方法，而有些人则喜欢在创建的位置便设置对象为自动释放对象，以免以后忘记释放这些对象。这两种方法都是有效的，不过请注意，对于iOS应用，苹果公司建议你直接使用release而不是autorelease自动释放，因为你无法知道什么时候自动释放池会释放保留计数器的值。

在carCopy创建了新的engine对象以后，copyWithZone方法执行了4次for循环，分别复制每个tire对象，并将复制的对象安装到新的car对象中。

```
for (int i = 0; i < 4; i++)
{
 Tire *tireCopy;
 tireCopy = [[self tireAtIndex: i] copy];
 [tireCopy autorelease];
 [carCopy setTire: tireCopy atIndex: i];
}
```

循环中的代码使用访问器方法在每趟循环中先后获得位置为0的tire对象、位置为1的tire对象，以此类推。然后，这些tire对象被复制并设置为自动释放，因而它们的内存可以被正确回收。接下来，carCopy被告知在同一位置使用新的tire对象。因为我们已经在Tire类和AllWeatherRadial类中精心构造了copyWithZone:方法，所以这段代码可以使用这两个类的对象正常工作。你应该注意到了，我们没有使用NSArray的copy方法，因为这样只会创建一个浅层复制而不是深层复制。

最后是完整的main()函数，其中的大多数代码你已经在前几章中见过，而新增的代码会以粗体形式显示出来。

```
int main (int argc, const char * argv[]) {
@autoreleasepool
{
 Car *car = [[Car alloc] init];
 car.name = @"Herbie";
 for (int i = 0; i < 4; i++)
 {
  AllWeatherRadial *tire;
  tire = [[AllWeatherRadial alloc] init];
  [car setTire: tire atIndex: i];
  [tire release];
 }
 Slant6 *engine = [[Slant6 alloc] init];
 car.engine = engine; [engine release];
 [car print];
 Car *carCopy = [car copy];
 [carCopy print];
 [car release];
```

```
 [carCopy release];
}
return (0);
} // main
```

该程序在输出了原始的car对象信息之后，复制了该car对象并将其内容输出。因此，我们应该得到两个完全相同的输出结果。运行该程序，你将看到如下所示的结果。

```
Herbie has:
AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
I am a slant-6. VROOOM!
Herbie has:
AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
I am a slant-6. VROOOM!
```


13.2.4 协议和数据类型

你可以在使用的数据类型中为实例变量和方法参数指定协议名称。这样，你可以给Objective-C的编译器提供更多的信息，从而有助于检查代码中的错误。

回想一下，id类型表示一个可以指向任何类型的对象的指针，它是一个泛型对象类型。你可以将任何对象复制给一个id类型的变量，也可以将一个id类型的变量复制给任何对象的对象指针。如果一个用尖括号括起来的协议名称跟随在id之后，则编译器（以及阅读此代码的人）将知道你会接受任意类型的对象，但前提是要遵守该协议。

举个例子，NSControl类中有一个名为setObjectValue:的方法，该方法要求对象遵守NSCopying协议。

```
- (void) setObjectValue: (id<NSCopying>) object;
```

编译器在编译该方法时，将检查参数类型，如果没有遵守协议则提出警告，如“class 'Triangle' does not implement the 'NSCopying' protocol”。这真是太方便了。

13.3 Objective-C 2.0 的新特性

苹果公司向来追求完美。Objective-C 2.0中增加了两个新的协议修饰符：@optional和@required。等一下，我们之前不是说如果要遵守一个协议就必须实现该协议的所有方法吗？是的，不过如果你指定了@optional关键字，那么就可以写如下代码。

```
@protocol BaseballPlayer
- (void)drawHugeSalary;
```

```
@optional
- (void)slideHome;
- (void)catchBall;
- (void)throwBall;
@required
- (void)swingBat;
@end // BaseballPlayer
```

这段代码采用了BaseballPlayer协议并且必须要实现-drawHugeSalary和-swingBat方法，而另外几个方法可以选择是否去实现：slideHome、catchBall和throwBall。

看起来非正式协议就可以实现这样的效果，为什么苹果公司还要添加这个特性呢？这是Cocoa提供的又一利器，可以用来在类声明和方法声明中明确表达我们的意图。比方说，你在一个头文件中看到下面这样的代码：

```
@interface CalRipken : Person <BaseballPlayer>
```

你可以立即看出我们所涉及的是一个领取丰厚薪水、会打棒球、会滑垒以及投球或接球的人。而如果使用非正式协议，就无法表达这些信息。同样，你可以使用协议修饰方法的参数：

```
-(void)draft:(Person<BaseballPlayer>);
```

这行代码明确指出了应该选拔什么样的人参加棒球比赛。如果你从事过iOS开发，那么应该已经注意到了Cocoa中的非正式协议逐渐被替换成了带有@optional方法的正式协议。

13.4　委托方法

现在我们要研究Objective-C中的协议，首先来讨论委托（delegation），它是一个经常与协议共用的特性。委托就是某个对象指定另一个对象处理某些特定任务的设计模式。

如果你以前看过很多Objective-C代码，可能会见过有关委托的示例。举个例子，UITableView和NSTableView都有一个由委托控制的数据源（dataSource）属性。因为委托要遵守协议，所以对象就会知道委托可以完成的任务。

下面我们用一个真实的示例来展示这一点。假设你老板要创建一个iOS或OS X应用程序，那么他要做所有的工作吗？应该不是。他会将项目拆分成较小的部分，只做其中一部分，并将剩余的工作指派（委托）给其他专业的程序员。有些雇员比其他人更擅长某项工作。因为老板懂得管理，所以能够知人善用。

事实上在上一章中你已经见过一个有关委托的示例项目。在12.04 iTunesFinder示例中，ITunesFinder类就是NSNetServiceBrowser类的委托。

NSNetServiceBrowser类如何知道对象能否完成它所需的工作？我们先来看一下NSNetServiceBrowser类的委托方法。

```
- (id <NSNetServiceBrowserDelegate>)delegate;
- (void)setDelegate:(id <NSNetServiceBrowserDelegate>)delegate;
```

第一个方法会返回当前的委托对象（如果没有的话则返回nil）。第二个是用来设置委托的，参数的委托类型告诉我们，只要遵守所需的协议，就可以设置任意对象为委托。

如果你浏览NSNetService.h文件，将会看到协议中有很多方法，它们全部都是可选的。你可以全都实现，也可以只实现其中一部分或一个都不实现。

我们定义了ITunesFinder类并让它处理所需协议的工作。

```
@interface ITunesFinder : NSObject <NSNetServiceBrowserDelegate>
@end // ITunesFinder
```

如果你移走了<NSNetServiceBrowserDelegate>部分代码，再指派对象作为委托会如何？编译器会抱怨“你无法处理这些方法!”并提出错误警告（不过它使用的是机器语言）。

```
Sending'ITunesFinder *__strong' to parameter of incompatible type'id<NSNetServiceBrowser
Delegate>'
```


我们来看一下13.02 Delegation文件夹中的示例。我们拥有3个对象：Worker1、Worker2和Manager。员工有一些必须完成的工作，并且可以选择做额外的工作以给老板留下好印象。请记住这个项目同样使用了ARC，所以不需要在对象使用结束后向其发送release方法。

```
int main(int argc, const char * argv[])
{
  @autoreleasepool
  {
    Manager *manager = [[Manager alloc] init];
    Worker1 *worker1 = [[Worker1 alloc] init];
    manager.delegate = worker1;
    [manager doWork];

    Worker2 *worker2 = [[Worker2 alloc] init];
    manager.delegate = worker2;
    [manager doWork];
  }
  return 0;
}
```

运行这个程序，就会看到如下输出结果：

```
Worker1 doing required work.
I am a manager and I am working
Worker2 doing required work.
Worker doing optional work.
I am a manager and I am working
```

主管（Manager）会执行自身的工作，并且也会要求员工（Worker）来执行必须完成的工作，而员工还可以执行那些可选的工作。

```
- (void)doWork
{
  [delegate doSomeRequiredWork];
  if(YES == [delegate respondsToSelector:@selector(doSomeOptionalWork)])
  {
    [delegate doSomeOptionalWork];
  }
```

```
    [self myWork];
}
```

Manager使用委托来让Worker执行必须的和可选的工作。代码首先询问委托是否实现了某个方法，如果实现了，它会要求委托处理这个方法。最终，Manager会完成它的工作。

看到这段代码你可能会想：为什么不在每次调用方法前都询问对象是否拥有这个方法的实现呢？那样程序就永远不会因为调用了不存在的方法而崩溃了。这样做确实有用，不过它会大大降低程序的速度。所以只有在针对委托时使用这个方法比较好。

13.5　小结

本章介绍了正式协议的概念。你可以通过在@protocol部分列出一组方法名来定义一个正式协议。在@interface声明中的类名之后列出用尖括号括起来的协议名称，对象便可以采用这些协议了。采用了正式协议后，对象就承诺了要实现该协议中列出的每一个要求实现的方法。如果你没有实现协议中的所有方法，编译器将向你发出警告，提醒你遵守协议。

另外，我们还讨论了面向对象编程中会出现的一些小问题，主要是当复制位于类层次结构中的对象时发生的问题。

祝贺你！你已经学习了Objective-C的绝大部分内容，并深入研究了在OOP中经常遇到的一些问题。这为你继续学习Cocoa编程或开发自己的项目打下了坚实的基础。下一章将要学习代码块，它是Objective-C的一个新特性，可以让你在函数中执行更多强大的功能。还将讨论并发性，了解它是如何让电脑和移动设备同时执行几项任务的。

第 14 章

代码块和并发性 14

本章将讨论代码块，一个可以增强函数功能的Objective-C特性。你可以在运行着iOS（版本4以上）和OS X（版本10.6以上）的应用程序中使用代码块。还将讨论并发性，也就是如何让现代设备同时执行多个任务。

14.1 代码块

代码块对象（通常简称为“代码块”）是对C语言中函数的扩展。除了函数中的代码，代码块还包含变量绑定。代码块有时也被称为闭包（closure）。

代码块包含两种类型的绑定：自动型与托管型。自动绑定（automatic binding）使用的是栈中的内存，而托管绑定（managed binding）是通过堆创建的。

因为代码块实际上是由C语言实现的，所以它们在各种以C作为基础的语言内都是有效的，包括Objective-C、C++以及Objective-C++。

代码块在Xcode的GCC和Clang工具中是有效的，但它不属于ANSI的C语言标准。关于代码块的提议已经提交给了C程序语言标准团体。

14.1.1 代码块和函数指针

在研究代码块之前，首先来讨论一下好用的老式函数指针。为什么呢？因为代码块借鉴了函数指针的语法。所以如果你知道如何声明函数指针，也就知道了如何声明一个代码块。与函数指针类似，代码块具有以下特征：

- 返回类型可以手动声明也可以由编译器推导；
- 具有指定类型的参数列表；
- 拥有名称。

让我们先声明一个函数指针：

```
void (*my_func)(void);
```

这是非常基础的函数指针，它没有参数和返回结果。只要把*（星号）替换成^（幂符号），就可以把它转换成一个代码块的定义了，比如像这样：

```
void (^my_block)(void);
```

在声明代码块变量和代码块实现的开头位置都要使用幂操作符。与在函数中一样，代码块的代码要放在{}（花括号）中。

“无码无真相”，你是不是这么想的？那么就展示给你看吧（你可以在14.01 Square文件夹中找到这些代码）。

```
int (^square_block)(int number) =
 ^(int number) {return (number * number);};
int result = square_block(5);
printf("Result = %d\n", result);
```

这个特别的代码块获取了一个整型参数并返回了这个数字的平方。等号前面是代码块的定义，而等号后面是实现内容。一般我们可以用如下关系来表示它们：

```
<returntype> (^blockname)(list of arguments) = ^(arguments){ body; };
```

编译器可以通过代码块的内容推导出返回类型，所以你可以省略它。如果代码块没有参数，也可以省略。因此代码块通常都是很简洁明了的，就像在14.02 Hello Blocks文件夹中的一样。

```
void (^theBlock)() = ^{ printf("Hello Blocks!\n"); };
```

1. 使用代码块

因为你将代码块声明成了变量，所以可以像函数一样使用它，就像在14.01 Square项目中你所看到的一样。

```
int result = square_block(5);
```

你可能注意到了，这行代码中没有幂符号，这是因为只有在定义代码块的时候才需要使用它，调用时则不需要。

代码块还有一个非常酷的特性可用来替换原先的函数：代码块可以访问与它相同的（本地）有效范围内声明的变量，也就是说代码块可以访问与它同时创建的有效变量。

```
int value = 6;
int (^multiply_block)(int number) = ^(int number)
 {return (value * number);};
int result = multiply_block(7);
printf("Result = %d\n", result);
```

2. 直接使用代码块

使用代码块时通常不需要创建一个代码块变量，而是在代码中内联代码块的内容。通常，你会需要一个将代码块作为参数的方法或函数（在14.04 Sorting Array文件夹中可以看到以下代码）。

```
 NSArray *array = [NSArray arrayWithObjects:
      @"Amir", @"Mishal", @"Irrum", @"Adam", nil];
NSLog(@"Unsorted Array %@", array);
NSArray *sortedArray = [array sortedArrayUsingComparator:^(NSString
  *object1, NSString *object2) {
    return [object1 compare:object2];
}];
```

```
NSLog(@"Sorted Array %@", sortedArray);
```

你简单地创建了一个代码块，用完之后就可以不管它了。

3. 使用typedef关键字

就像你所看到的，如此长的变量定义语句有些令人生畏。输入这些代码时很容易引起错误。幸好typedef可以帮上忙（参考14.05 Typedefed Blocks文件夹中的代码）。

```
typedef double (^MKSampleMultiply2BlockRef)(double c, double d);
```

这行语句定义了一个名称为MKSampleMultiply2BlockRef的代码块变量，它包含两个双浮点型参数并返回一个双浮点型数值。有了typedef，就可以像下面这样使用这个变量。

```
MKSampleMultiply2BlockRef multiply2 = ^(double c, double d)
 { return c * d; };
printf("%f, %f", multiply2(4, 5), multiply2(5, 2));
```

如果你看过了14.06 More Typedefs文件夹中的代码，会发现我们还定义了其他一些不同类型的代码块变量。

```
typedef void (^MKSampleVoidBlockRef)(void);
typedef void (^MKSampleStringBlockRef)(NSString *);
typedef double (^MKSampleMultiplyBlockRef)(void);
```

4. 代码块和变量

代码块被声明后会捕捉创建点时的状态。代码块可以访问函数用到的标准类型的变量：

- 全局变量，包括在封闭范围内声明的本地静态变量。
- 全局函数（明显不是真实的变量）。
- 封闭范围内的参数。
- 函数级别（即与代码块声明时相同的级别）的__block变量。它们是可以修改的变量。
- 封闭范围内的非静态变量会被获取为常量。
- Objective-C的实例变量。
- 代码块内部的本地变量。

接下来会详细研究这些变量类型。

5. 本地变量

本地变量就是与代码块在同一范围内声明的变量。我们来看一个示例，它位于14.06 More Typedefs文件夹中。

```
typedef double (^MKSampleMultiplyBlockRef)(void);
double a = 10, b = 20;

MKSampleMultiplyBlockRef multiply = ^(void){ return a * b; };
NSLog(@"%f", multiply());
a = 20;
b = 50;

//你认为会输出什么内容?
NSLog(@"%f", multiply());
```

你认为第二个NSLog语句会输出什么值呢？1000？很遗憾，不是。为什么？因为变量是本地的，代码块会在定义时复制并保存它们的状态。因为这个原因，NSLog只会输出200。

6. 全局变量

在上面的本地变量示例中，我们说过变量与代码块拥有相同的有效范围，你可以根据需要把变量标记为静态的（全局的）。

```
static double a = 10, b = 20;

MKSampleMultiplyBlockRef multiply = ^(void){ return a * b; };
NSLog(@"%f", multiply());
a = 20;
b = 50;
NSLog(@"%f", multiply());
```

7. 参数变量

代码块中的参数变量与函数中的参数变量具有同样的作用。

```
typedef double (^MKSampleMultiply2BlockRef)(double c, double d);
MKSampleMultiply2BlockRef multiply2 = ^(double c, double d) { return c * d; };
NSLog(@"%f, %f", multiply2(4, 5), multiply2(5, 2));
```

8. __block变量

本地变量会被代码块作为常量获取到。如果你想要修改它们的值，必须将它们声明为可修改的，否则像下面这个错误的示例，编译时会出现错误。

```
double c = 3;
MKSampleMultiplyBlockRef multiply = ^(double a, double b) { c = a * b; };
```

编译器将会警告这个错误：

```
Variable is not assignable (missing __block type specifier)
```

如果想要修复这个编译错误，你需要听取前面编译器的友好建议，将变量c标记为__block（参考14.07 Mutable Variables文件夹中的代码）。

```
__block double c = 3;
 MKSampleMultiplyBlockRef multiply = ^(double a, double b)
  { c = a * b; };
```

有些变量是无法声明为__block类型的。它们有两个限制：

- 没有长度可变的数组；
- 没有包含可变长度数组的结构体。

9. 代码块内部的本地变量

这些变量与函数中的本地变量具有同样的作用。

```
void (^MKSampleBlockRef)(void) = ^(void){
   double a = 4;
   double c = 2;
   NSLog(@"%f", a * c);
};
MKSampleBlockRef();
```

14.1.2 Objective-C变量

代码块是Objective-C语言中的优秀公民，你可以像使用其他对象一样使用它。使用时会遇到的最大问题就是内存管理。我们之前也说过，在代码块中访问Objective-C变量时必须小心。与往常一样，以下规则能帮助你处理内存管理。

- 如果引用了一个Objective-C对象，必须要保留它。
- 如果通过引用访问了一个实例变量，要保留一次self（即执行方法的对象）。
- 如果通过数值访问了一个实例变量，变量需要保留。

第一个规则非常简单，不过与其他两个有些不一样。我们来看一下14.08 Objective Blocks文件夹中的示例。在项目中浏览ProcessStrings.m文件。

```
NSString *string1 = ^{
  return [_theString stringByAppendingString:_theString];
  };
```


在这个示例中，__theString是声明了代码块的类里面的实例变量。因为在代码块中直接访问了实例变量，所以包含它的对象（self）需要保留。现在再来看一个例子。

```
NSString *localObject = _theString;
NSString *string2 = ^{
  return [localObject stringByAppendingString:localObject];
};
```

在这个例子中，我们是间接访问：创建了一个指向实例变量的本地引用并在代码块里面使用。因此这次要保留的是localObject，而不是self。

因为代码块是对象，所以可以向它发送任何与内存管理有关的消息。在C语言级别中，必须使用Block_copy()和Block_release()函数来适当地管理内存。14.09 Block Copy文件夹中的项目展示了如何复制代码块。

```
MKSampleVoidBlockRef block1 = ^{
  NSLog(@"Block1");
};
block1();

MKSampleVoidBlockRef block2 = ^{
  NSLog(@"Block2");
};
block2();
Block_release(block2);

block2 = Block_copy(block1);
block2();
```

14.2　并发性

到目前为止，我们讨论过的所有代码都是一个接着一个按照顺序执行的。现在我们将讨论如何打破这个限制。

用来运行Xcode的Mac电脑的处理器至少拥有两个核心，也可能更多。现在最新的iOS设备都是多核的。这意味着你可以在同一时间进行多项任务。苹果公司提供了多种可以利用多核特性的API。能够在同一时间执行多项任务的程序称为并发的（concurrent）程序。

利用并发性最基础的方法是使用POSIX线程来处理程序的不同部分使其能够独立执行。POSIX线程拥有支持C语言和Objective-C的API。编写并发性程序需要创建多个线程，而编写线程代码是很有挑战性的。因为线程是级别较低的API，你必须手动管理。根据硬件和其他软件运行的环境，需要的线程数量会发生变化。处理所有的线程是需要技巧的，一旦遇到问题，你可能会觉得不使用线程会更好一些。

苹果公司为我们带来了希望。为了减轻在多核上编程的负担，苹果公司引入了Grand Central Dispatch，我们亲切地称之为GCD。这个技术减少了不少线程管理的麻烦，因此你可以集中精力编码。如果想要使用GCD，你需要提交代码块或函数作为线程来运行。GCD是一个系统级别（system-level）的技术，因此你可以在任意级别的代码中使用它。GCD决定需要多少线程并安排它们运行的进度。因为它是运行在系统级别上的，所以可以平衡应用程序所有内容的加载，这样可以提高计算机或设备的执行效率。

14.2.1　同步

我们来打个比方。你在一条多车道的高速公路上驾驶汽车，不时地被一些车超过，也会超过更慢的车辆。你的计算机就像这样的多车道。在马路上，有些执行通路会花很长时间，有些会很快。如果你想要在高峰时段进入高速公路，必须等待前面的车依次驶进公路中。车道和交通信号灯则尽可能地控制着车流平稳前进。现在，回到我们的计算机上，我们该如何在由多核组成的通路中管理交通呢？可以使用同步装置，比如在通道入口立一个标记（flag）或一个互斥（mutex）。

> **说明**　mutex是mutual exclusion的缩写，它指的是确保两个线程不会在同一时间进入临界区。

Objective-C提供了一个语言级别的（language-level）关键字@synchronized。这个关键字拥有一个参数，通常这个对象是可以修改的。

```
@synchronized(theObject)
{
// Critical section.
}
```

它可以确保不同的线程会连续地访问临界区的代码。

若你定义了一个属性并且没有指定关键字nonatomic作为属性的特性，编译器会生成强制彼此互斥的getter和setter方法。如果你不在意，或者知道这些属性值不会被多个线程访问的话，可以添加nonatomic特性。为什么要这么做？为了提高性能。那它是如何做到的？编译器生成了@synchronize (mutex, atomic)语句来确保彼此互斥。这样设置代码和变量会产生一些消耗，它会比直接访问更慢一些。

1. 选择性能

如果你只想让一些代码在后台执行，NSObject也提供了方法。这些方法的名字中都有performSelector:，最简单的就是performSelectorInBackground:withObject:了，它能在后台执行一个方法。它通过创建一个线程来运行方法。定义这些方法时必须遵从以下限制。

- 这些方法运行在各自的线程里，因此你必须为这些Cocoa对象创建一个自动释放池，而主自动释放池是与主线程相关的。
- 这些方法不能有返回值，并且要么没有参数，要么只有一个参数对象。换句话说，你只能使用以下代码格式中的一种。

```
- (void)myMethod;
- (void)myMethod:(id)myObject;
```

记住这些限制，我们实现的代码应该如下所示（参考14.10 Selectors文件夹中的项目）。

```
- (void)myBackgroundMethod
{
 @autoreleasepool
 {
   NSLog(@"My Background Method");
 }
}
```

或如下所示

```
- (void)myOtherBackgroundMethod:(id)myObject
{
 @autoreleasepool
 {
   NSLog(@"My Background Method %@", myObject);
 }
}
```

如果想要在后台执行你的方法，只需要调用performSelectorInBackground:withObject:方法，就像下面这样。

```
[self performSelectorInBackground:@selector(myBackgroundMethod)
 withObject:nil];
[self performSelectorInBackground:@selector(myOtherBackgroundMethod:)
 withObject:arugmentObject];
```

这样就完成了。当方法执行结束后，Objective-C运行时会特地清理并弃掉线程。需要注意，方法执行结束后并不会通知你：这是比较简单的代码。如果想要做一些更复杂的事情，你需要继续阅读下去，探索调度队列的神奇世界。

2. 调度队列

GCD可以使用调度队列（dispatch queue），它与线程很相似但使用起来更简单。只需写下你的代码，把它指派为一个队列，系统就会执行它了。你可以同步或异步执行任意代码。一共有以下3种类型的队列。

- 连续队列：每个连续队列都会根据指派的顺序执行任务。你可以按自己的想法创建任意数量的队列，它们会并行操作任务。
- 并发队列：每个并发队列都能并发执行一个或多个任务。任务会根据指派到队列的顺序开始执行。你无法创建连续队列，只能从系统提供的3个队列内选择一个来使用。
- 主队列：它是应用程序中有效的主队列，执行的是应用程序的主线程任务。

接下来讨论各种类型的队列及其使用方法。

- 连续队列

有时有一连串任务需要按照一定的顺序执行，这时便可以使用连续队列。任务执行顺序为先入先出（FIFO）：只要任务是异步提交的，队列会确保任务根据预定顺序执行。这些队列都是不会发生死锁的。

死　锁

死锁（deadlock）是一个令人不悦的情况，指的是两个或多个任务在等待他方运行结束，就像是几辆汽车同时位于一个很拥挤的停车场里。

让我们使用一次连续队列：

```
dispatch_queue_t my_serial_queue;
my_serial_queue = dispatch_queue_create
 ("com.apress.MySerialQueue1", NULL);
```

第一个参数是队列的名称，第二个负责提供队列的特性（现在用不到，所以必须是NULL）。当队列创建好之后，就可以给它指派任务。因为队列是引用计数的对象，所以等下我们还要讨论内存管理。

- 并发队列

并发调度队列适用于那些可以并行运行的任务。并发队列也遵从先入先出（FIFO）的规范，且任务可以在前一个任务结束前就开始执行。一次所运行的任务数量是无法预测的，它会根据其他运行的任务在不同时间变化。所以每次你运行同一个程序，并发任务的数量可能会是不一样的。

说明　如果需要确保每次运行的任务数量都是一样的，可以通过线程API来手动管理线程。

每个应用程序都有3种并发队列可以使用：高优先级（high）、默认优先级（default）和低优先级（low）。如果想要引用到它们，可以调用dispatch_get_global_queue方法（参考14.11 Hello Queue文件夹中的项目）。

```
dispatch_queue_t myQueue;
myQueue = dispatch_get_global_queue (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
```

其他优先级的选项分别是DISPATCH_QUEUE_PRIORITY_HIGH和DISPATCH_QUEUE_PRIORTY_LOW。第二个参数暂时都用0。因为它们都是全局的，所以无需为它们管理内存。你不需要保留这些队列的引用，在需要的时候使用函数来访问就可以了。

- 主队列

使用dispatch_get_main_queue可以访问与应用程序主线程相关的连续队列。

```
dispatch_queue_t main_queue = dispatch_get_current_queue(void);
```

因为这个队列与主线程相关，所以必须小心安排这个队列中的任务顺序，否则它们可能会阻塞主应用程序运行。通常要以同步方式使用这个队列，提交多个任务并在它们操作完毕后执行一些动作。

- 获取当前队列

你可以通过调用dispatch_get_current_queue()来找出当前运行的队列代码块。如果你在代码块对象之外调用了这个函数，则它将会返回主队列。

```
dispatch_queue_t myQueue = dispatch_get_current_queue();
```


14.2.2 队列也要内存管理

调度队列是引用计数对象。可以使用dispatch_retain()和dispatch_release()来修改队列的保留计数器的值。如你所想，它们与一般对象的retain和release语句类似。你只能对你自己创建的队列使用这些函数，而无法用在全局调度队列上。事实上，如果你向全局队列发送这些消息，它们会被直接忽略掉，所以即使这样做也是无害的。如果你编写的是一个使用了垃圾回收的OS X应用程序，那么你必须手动管理这些队列。

1. 队列的上下文

你可以向调度对象（包括调度队列）指派全局数据上下文，可以在上下文中指派任意类型的数据，比如Objective-C对象或指针。系统只能知道上下文包含了与队列有关的数据，上下文数据的内存管理只能由你来做。你必须在需要它的时候分配内存并在队列销毁之前进行清理。在为上下文数据分配内存的时候，可以使用dispatch_set_context()和dispatch_get_context()函数。

```
NSMutableDictionary *myContext = [[NSMutableDictionary alloc] initWithCapacity:5];
[myContext setObject:@"My Context" forKey:@"title"];
[myContext setObject:[NSNumber numberWithInt:0] forKey:@"value"];
dispatch_set_context(_serial_queue, (__bridge_retained void *)myContext);
```

在这个示例中，我们创建了一个字典来存储上下文，当然也可以使用其他的指针类型。分配好内存之后就可以使用了。

在最后一行代码中，由于我们必须保证对象是有效的，所以使用了__bridge_retained来给myContext的保留计数器的值加1。

● 清理函数

设置完上下文对象的数据之后，怎么知道什么时候要清理呢？答案其实很简单，你不需要真的知道上下文对象在何时何地会被弃用。如果想要解决这个问题，你可以让对象在它弃用的时候调用一个函数，就像类里面的dealloc函数。函数的格式应该如下所示。

```
void function_name(void *context);
```

我们将创建一个会在上下文对象弃用时调用的示例函数，通常被称为终结器（finalizer）函数。

```
void myFinalizerFunction(void *context)
{
  NSLog(@"myFinalizerFunction");
  NSMutableDictionary *theData = (__bridge_transfer NSMutableDictionary*)context;
  [theData removeAllObjects];
}
```

顺便讨论一下__bridge_transfer关键字。这个关键字把对象的内存管理由全局释放池变换成了我们的函数。当我们的函数结束时，ARC将会给它的保留计数器的值减1，如果保留计数器的值被减到了0，对象就会被释放。如果对象没有被释放，myContext就会一直留在内存中。

如何在代码块里面访问上下文内容呢？

```
NSMutableDictionary *myContext = (__bridge NSMutableDictionary *)
    dispatch_get_context(dispatch_get_current_queue());
```

为什么要在这里添加__bridge关键字呢？因为要告诉ARC我们并不想自己来管理上下文的内存，而想交给系统来管理。

● 添加任务

有两种方式可以向队列中添加任务。

- 同步：队列会一直等待前面任务结束。
- 异步：添加任务后，不必等待任务，函数会立刻返回。推荐优先使用这种方式，因为它不会阻塞其他代码的运行。

你可以选择向队列提交代码块或函数。一共有4个调度函数，分别是代码块和函数各自的同步与异步方式。接下来了解每一个调度函数的类型。

> **说明**　如果想要避免出现死锁，那么绝对不要给运行在同一队列中的任务调用dispatch_sync或dispatch_sync_f函数。

2. 调度程序

添加任务的最简单的方法是通过代码块。代码块必须是dispatch_block_t这样的类型，要定义为没有参数和返回值才行。

```
typedef void (^dispatch_block_t)(void)
```

先添加异步代码块。这个函数拥有两个参数，分别是队列和代码块。

```
dispatch_async(_serial_queue, ^{ NSLog(@"Serial Task 1"); });
```

如果你愿意的话，也可以定义一个代码块对象，而不是通过内联方式来创建。

```
dispatch_block_t myBlock = ^{ NSLog(@"My Predfined block"); };
dispatch_async(_serial_queue, myBlock);
```

如果想要同步添加，请使用dispatch_sync函数。

之前说过，我们也可以向队列中添加函数。函数的标准原型必须要像下面这样：

```
void function_name(void *argument)
```

以下是示例函数：

```
void myDispatchFuction(void *argument)
{
  NSLog(@"Serial Task %@", (__bridge NSNumber *)argument);
  NSMutableDictionary *context = (__bridge NSMutableDictionary *)
    dispatch_get_context(dispatch_get_current_queue());
  NSNumber *value = [context objectForKey:@"value"];
  NSLog(@"value = %@", value);
}
```


接下来需要向队列添加这个函数。调度函数拥有3个参数：队列、需要传递的任意上下文以及函数。如果你没有信息要发送给函数，也可以只传递一个NULL值。

```
dispatch_async_f(_serial_queue, (__bridge void *)[NSNumber numberWithInt:3],
    (dispatch_function_t)myDispatchFuction);
```

队列创建完之后，就做好接收任务的准备了。当我们添加了一个任务，队列就会安排好它的进度。如果想要以同步方式添加到队列中，请调用dispatch_sync_f函数。

如果出于某个原因你想要暂停队列，请调用dispatch_susend()函数并传递队列名称。

```
dispatch_suspend(_serial_queue);
```

队列暂停之后，可以调用dispatch_resume()函数来重新启用。这些全都由你来掌握。

```
dispatch_resume(_serial_queue);
```

14.2.3 操作队列

你可能注意到了，这些东西的层级都非常低。你一定想知道怎样能让它在Objective-C中使用。你现在阅读的可是关于这个语言的书啊。实际上，有一些被称为操作（operation）的API，可以让队列在Objective-C层级上使用起来更加简单。

如果想要使用操作，首先需要创建一个操作对象，然后将其指派给操作队列，并让队列执行它。一共有3种创建操作的方式。

- NSInvocationOperation：如果你已经拥有一个可以完成工作的类，并且想要在队列上执行，可以尝试使用这种方法。
- NSBlockOperation：这有些像包含了需要执行代码块的dispatch_async函数。
- 自定义的操作：如果你需要更灵活的操作类型，可以创建自己的自定义类型。你必须通

过NSOperation子类来定义你的操作。

我们将花一些时间和笔墨（或者是电池，如果你使用的是电子书阅读器的话）来详细讨论这些内容。

创建调用操作（invocation operation）

NSInvocationOperation会为执行任务的类调用选择器。因此，如果你拥有了一个包含所需方法的类，使用这种方式来创建会非常方便（参考14.12 Objective Queue文件夹中的项目）。

```
@implementation MyCustomClass
- (NSOperation *)operationWithData:(id)data
{
  return [[NSInvocationOperation alloc] initWithTarget:self
   selector:@selector(myWorkerMethod:) object:data];
}

// This is the method that does the actual work
- (void)myWorkerMethod:(id)data
{
    NSLog(@"My Worker Method %@", data);
}
@end
```

一旦向队列添加了操作，任务即将执行时便会调用类里面的myWorkerMethod:方法。

- 创建代码块操作（block operation）

如果你有一个需要执行的代码块，那么可以创建这个操作并让队列执行它。

```
NSBlockOperation *blockOperation = [NSBlockOperation blockOperationWithBlock:^{
  // Do my work
}];
```

这些操作的代码块与我们在调度队列中使用的是同一种类型。一旦创建了第一个代码块操作，你便可以通过addExecutionBlock:方法继续添加更多的代码块。根据队列的类型（连续的或并发的）代码块会分别以连续或并行的方式运行。

```
[blockOperation addExecutionBlock:^{
    // do some more work
}];
```

- 向队列中添加操作

一旦创建了操作，你就需要向队列中添加代码块。这次我们将使用NSOperationQueue来取代之前使用的dispatch_queue_t函数。NSOperationQueue一般会并发执行操作。它具有相关性，因此如果某操作是基于其他操作的，它们会相应地执行。

如果要确保你的操作是连续执行的，可以设置最大并发操作数为1，这样任务将会按照先入先出的规范执行。在向队列添加操作之前，需要某个方法来引用到那个队列。你可以创建一个新队列或使用之前已经定义过的队列（比如当前运行的队列）。

```
NSOperationQueue *currentQueue = [NSOperationQueue currentQueue];
```

或主队列：

```
NSOperationQueue *mainQueue = [NSOperationQueue mainQueue];
```

以下就是我们创建自己队列的代码：

```
NSOperationQueue *_operationQueue = [[NSOperationQueue alloc] init];
```

现在我们拥有了一个队列，可以使用addOperation:来添加操作：

```
[theQueue addOperation:blockOperation];
```

也可以添加需要执行的代码块来代替操作对象：

```
[theQueue addOperationWithBlock:^{
  NSLog(@"My Block");
}];
```

一旦向队列中添加了操作，它就会被安排进度并执行。

14.3 小结

本章介绍了代码块，它是Objective-C的新特性，增强了函数的功能。有了代码块，就可以通过绑定变量来创建程序中会使用到的对象。代码块在实现并发性功能时尤其方便。

并发性很复杂，本章仅讨论对OS X和iOS程序有效的并发性功能。

苹果公司的Grand Central Dispatch（GCD）特性提供了一种方法，你无需花很多时间在系统的低层级编码，应用程序就可以使用并发性。你应该多尝试GCD和其他并发性编程功能，以找出哪些对于你的应用程序是可行的，哪些很好用。

随着你的水平不断提高以及苹果公司添加更多的工具，你的应用程序将能够并行执行更多的任务，从而更快地作出响应。不过，一旦超过了临界点，给应用程序添加并行的任务（花大量时间编码和调试）就会得不偿失。

如果你经常要使用并发任务，请避免发生死锁（任务互相关联导致程序永远无法结束）或出现其他麻烦的bug。这种问题你肯定不希望发生。

第 15 章 AppKit简介 15

到目前为止，本书中的所有程序都使用了Foundation Kit，并通过将文本输出结果发送到控制台的原始方式向我们传达信息。用Foundation Kit确实不错，但真正有趣的还是可以构建一个像Mac那样的用户界面，既可以点击又可以玩的。本章将介绍Application Kit（简称AppKit）的一些重要特性，你会在AppKit里面看到Cocoa中关于用户界面的大量资源。

本章将要创建的应用程序的名称是CaseTool，你可以在15.01 CaseTool项目文件夹中找到它。CaseTool显示了一个窗口，该窗口的外观类似于图15-1的截图。它包含了一个文本框、一个标签和两个按钮。向文本框中输入文本并点击一个按钮后，输入的文本就会转换为大写或小写形式。这项功能确实很酷，但你肯定想在应用程序在Mac App Store上架并收费前为其添加更多实用的功能。

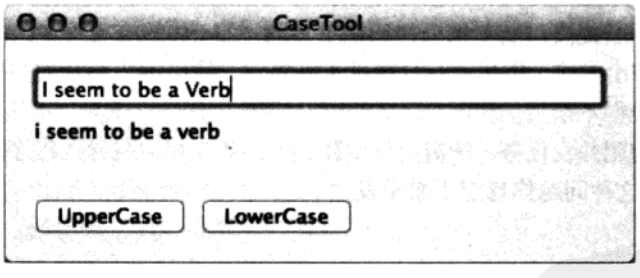


图15-1　CaseTool的应用界面

15.1　构建项目

使用Xcode来构建这个项目，我们将逐步引导你完成整个过程。你要做的第一件事情是创建项目文件，然后对用户界面进行布局，最后将建立UI与代码之间的关联。

创建Cocoa应用程序项目之前要先打开Xcode，在启动界面中点击Create a New Xcode Project选项。（如果Xcode已经在运行，则点击File ➤ New ➤ New Project选项。）如果左边列表的Mac OS X标题下的Application选项没有选中的话，请选择它。然后再选择Cocoa Application图标（如图15-2所示），之后点击Next按钮以设置接下来的选项。

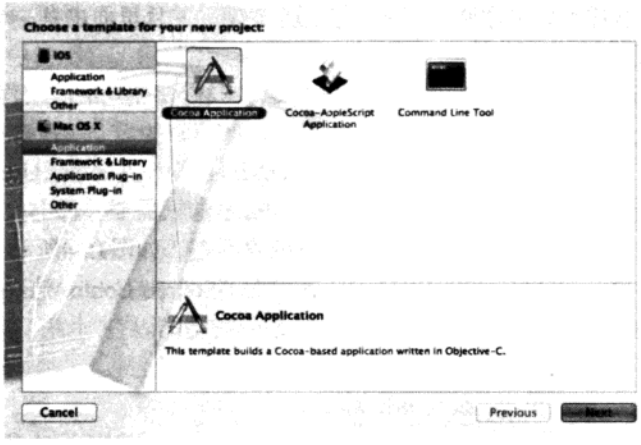


图15-2 创建一个新的Cocoa应用程序

浏览一下图15-3中设置选项的对话框。第一项是Product Name（应用名称），我们在这里输入CaseTool。下一个文本框是Company Identifier（企业标识符），App Store依靠它来区分你的应用程序。Company Identifier通常都是*反域名*（reverse-domain-name）格式，也就是说以com开头，然后是句号和企业名称。在这个示例中，我们用的是com.MySuperCompany。这个文本框中的内容是区分大小写的，在创建自动应用程序的时候需要注意这一点。

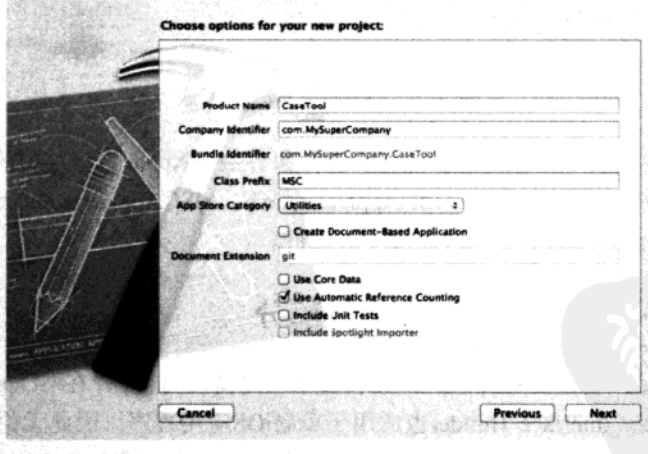


图15-3 为项目命名

接下来的文本框是Class Prefix（类前缀）。通常应在这里输入3个以上的字母，这个字符串会出现在你创建的新类的名称开头。我们在这里输入MSC（即My Super Company的缩写）。如果你创建了一个名称为`Circle`的类，Xcode会自动将其命名为`MSCCircle`。为什么要这么做？因为Objective-C中没有命名空间（namespace），它用的是一种伪命名空间的方法，将应用程序中的一

些名称储存起来了。这样，如果你使用了某个第三方框架，并且里面也有一个名为Circle的类，那么它将不会与你的名称发生冲突。

在App Store Category下拉列表中选择Utilities，因为我们要创建的应用程序其实就是一个实用工具。

至于其他的选项，请确保没有选中Create Document-Based Application、Use Core Data和Include Unit Tests复选框。只有Use Automatic Reference Counting复选框需要被选中。

最后会弹出一个让你选择保存项目的目录位置的对话框（如图15-4所示）。本书不会讨论源代码管理，当然如果你愿意的话，可以选择对话框下面的Source Control选项，创建一个Git版本控制。Xcode内置了Subversion（SVN）和Git的源代码管理功能。

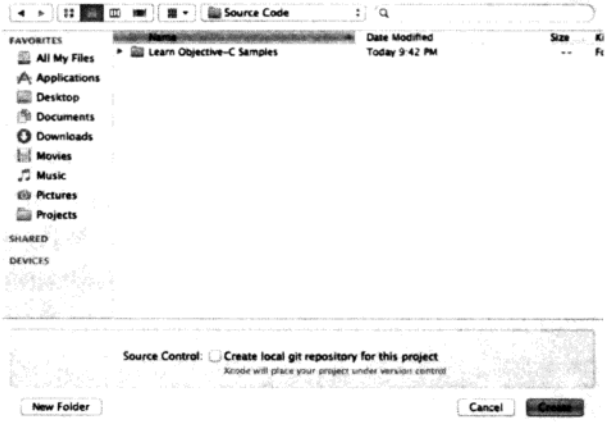


图15-4　指定项目存储的位置

完成这些之后你的新项目就已经创建好了。如果你展开Xcode窗口左边的CaseTool群组，将会看到项目中已经存在了一些文件：MSCAppDelegate.h、MSCAppDelegate.m和MainMenu.xib。

其中MSCAppDelegate类负责控制应用程序的运行。

15.2　创建委托文件的@interface 部分

我们将使用Xcode中的Interface Builder编辑器来布局窗口内容，并在MSCAppDelegate与用户界面之间创建各种连接。Interface Builder也适用于布局iOS应用程序，因此无论你要为哪一个平台开发应用程序，都需要在Interface Builder上进行很多工作。我们将向MSCAppDelegate类添加一些代码，这样Interface Builder就能够识别我们添加的内容，以便于我们构建用户界面。

首先来浏览一下委托的头文件：

```
#import <Cocoa/Cocoa.h>
@interface MSCAppDelegate : NSObject <NSApplicationDelegate>
@property (assign) IBOutlet NSWindow *window;
@end
```

你将会看到一个叫做IBOutlet的关键字。它不是一个真正的Objective-C关键字，但我们通常会在Interface Builder中使用它。你可能注意到了在IBOutlet那行代码的左边边栏有一个小点。点击那个点就会直接跳转到Interface Builder中与其相关的对象上。

我们之后还会遇到另一个像关键字的IBAction。IBAction被定义为void的作用，这意味着这个在文件中声明的方法返回的类型将是void（也就是什么都不返回）。

如果IBOutlet和IBAction不执行任何操作，为什么还要定义它们呢？原因是它们不是用于编译的，而是为Interface Builder以及阅读代码的人提供的标记。通过查找文件中的IBOutlet和IBAction语句，Interface Builder就能知道MSCDelegate对象拥有两个可以用来连接的实例变量，并且MSCDelegate还提供了两个方法作为按钮点击（也可能是其他界面操作）的目标。稍后将讲解其运行原理。

15.3 Interface Builder

现在是时候使用Xcode的Interface Builder编辑器了，有些人也会亲切地称其为IB。我们现在要编辑项目创建时附带的MainMenu.xib文件。这个文件配备了一个菜单栏以及一个可以放置用户控件的窗口。

在Xcode项目窗口中，找到并双击MainMenu.xib文件（如图15-5所示）。

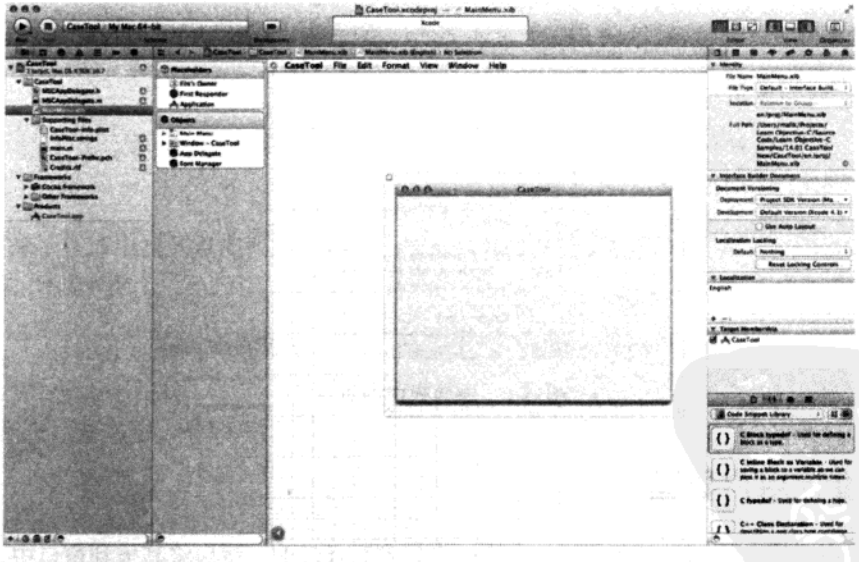


图15-5 打开MainMenu.xib文件，查看Interface Builder编辑器

这样就会打开Xcode中的Interface Builder编辑器并展示了文件的内容。虽然文件的扩展名为.xb，我们仍称之为nib文件。nib是NeXT Interface Builder的首字母缩写，是Cocoa从NeXT公司沿袭下来的技术结晶。nib文件是包含了压缩（freeze-dried）对象的二进制文件，而.xib文件是XML

格式的nib文件。在编译时，.xib文件将会编辑为nib格式。

注意一下编辑器区域的左边。Interface Builder的dock面板包含了表示nib文件内容的图标。可能你一时无法理解它们都是什么，各自有什么功能，不过不必担心，马上就会学到了。目前，我们可以展开dock面板来显示对象的名称。有没有看到dock面板底部右边的圆形中有个箭头（看起来有些像播放按钮）？点击它就可以看到dock面板中各项的名称。

编辑器窗口顶端是我们所创建的应用程序的菜单栏。你可以添加或编辑菜单以及菜单项。在这个项目中我们不会对菜单栏进行修改。

菜单栏下面就是我们用来放置文本框和按钮的空窗口。dock面板上窗口形状的图标所表示的就是这个真实可见的窗口。现在，我们要使用右下角的库面板，第7章谈到过它。在库面板中点击左起第三个看起来像立方体的图标，面板就会切换到对象库（如图15-6所示），它包含了大量可以拖入窗口的不同类型的对象。你可以在底部的搜索框输入一些文字以便筛选可见的内容。为了便于你使用，每一种可以使用的对象都提供了一段描述信息。

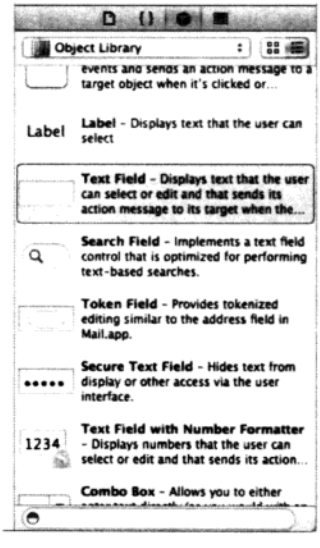


图15-6　库面板中的对象库

15.4　设计用户界面

现在可以对用户界面进行布局了。在库中先找到Text Field，然后将其拖到窗口中，如图15-7所示。在拖动过程中，你将会看到窗口中出现了蓝色的引导线。这有助于你按照苹果公司的用户界面规范对对象进行布局。

现在，拖入一个标签。点击库里面的Label图标并将其拖入窗口，如图15-8所示。在这个标签中将会显示转换成大写或小写的字符串结果。

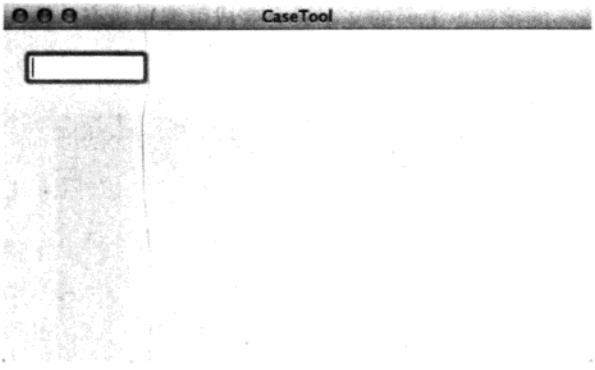


图15-7 拖动文本框

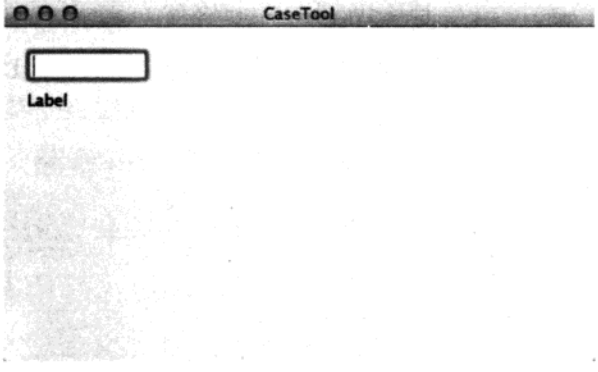


图15-8 拖入一个标签

接下来，在库中找到点击按钮（push button），并将其拖到标签的下方，如图15-9所示。这样做很有趣，不是吗？

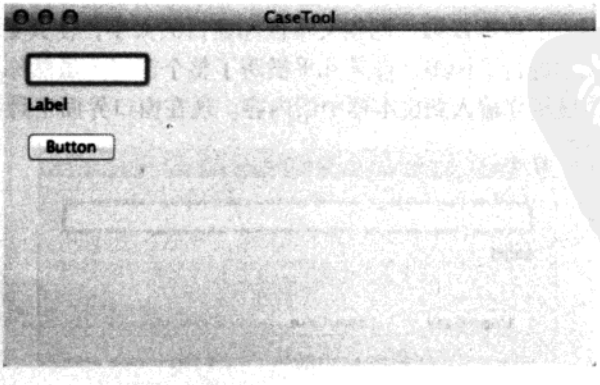


图15-9 拖动按钮到窗口中

现在，双击刚刚放入的按钮，上面的文字就会变成可编辑状态。输入UpperCase，并按下回车键以完成修改。图15-10展示了按钮在编辑时的状态。

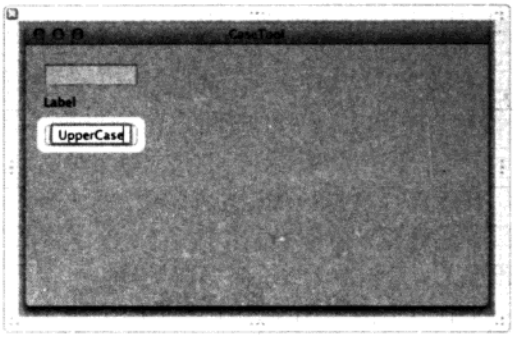


图15-10　编辑按钮的名称

现在，从库面板拖入另一个按钮并将它的名称修改成LowerCase。图15-11展示了添加第二个按钮后的窗口界面。

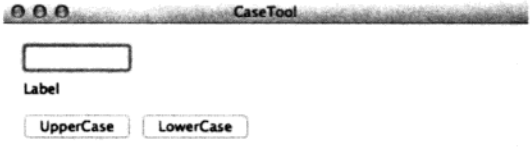


图15-11　所有的控件都已添加完毕

接下来，简单进行一下内部修饰，调整文本框和窗口的大小，使其更加美观，如图15-12所示。我们也对标签的尺寸进行了调整，使其几乎横跨了整个窗口（虽然你无法在图中看出来）。标签必须足够宽，以便显示你输入到文本框中的内容。现在窗口界面正是我们想要的外观。

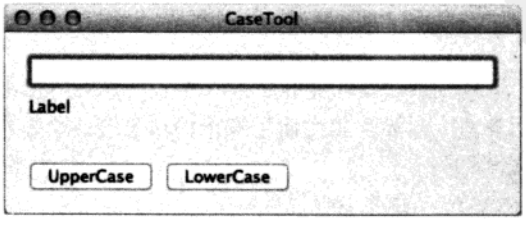


图15-12　窗口布局已经整理好了

15.5 创建连接

本节将介绍如何将代码与刚才创建的用户界面元素相连接。

15.5.1 连接输出口（IBOutlet）

现在是时候进行连接了。首先需要告诉NSTextField类型的实例变量textField以及resultsField分别指向哪个控件。我们将会用到辅助编辑器，第7章简单介绍过。在Xcode窗口的右上角有3个标记为Editor的按钮。点击中间的按钮，就会将编辑器从中间垂直分成两部分，这就是辅助编辑器。如果你想要更多的有效空间，可以把dock面板折叠回迷你形态，即只有图标显示出来。

接下来，按住Control键并将光标从文本框拖动到头文件——没错，就这么直接拖到@property那一行的下面，出现“Insert Outlet or Action”提示消息（如图15-13所示）再松手。松开鼠标之后，将会弹出一个如图15-14所示的对话框。在Name文本框输入textField，并点击Connect按钮。这样做便会在头文件中创建textField属性和其他所需的关键字。到目前为止，我们还未输入任何代码，但是我们却已经“写了”很多内容。这真是太有趣了。接下来对标签重复之前的连接步骤并将其命名为resultsField。

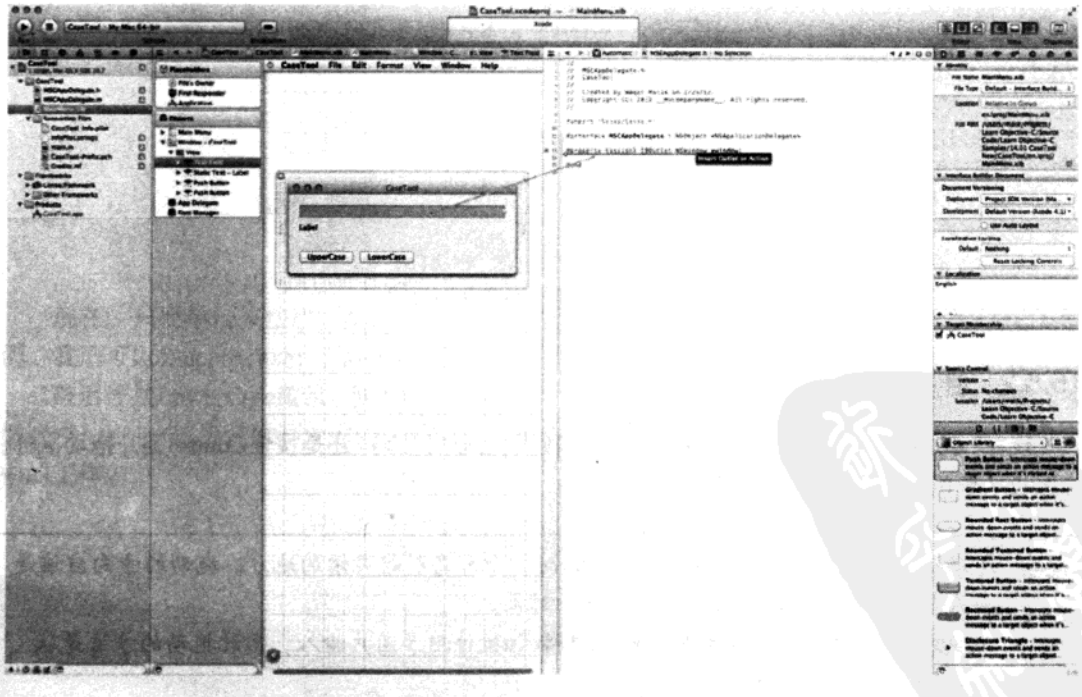


图15-13 开始创建连接

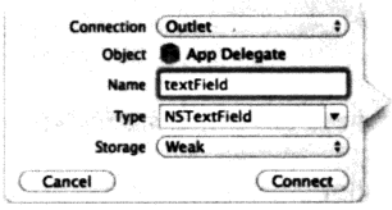


图15-14　编辑连接

请在连接检查器中重新确认你之前的工作。选择View ➤ Utilities ➤ Show Connection Inspector选项或点击检查器面板上的连接按钮（看起来像右箭头）就能看到连接检查器面板了。你将会在检查器顶端看到刚才的连接，如图15-15所示。

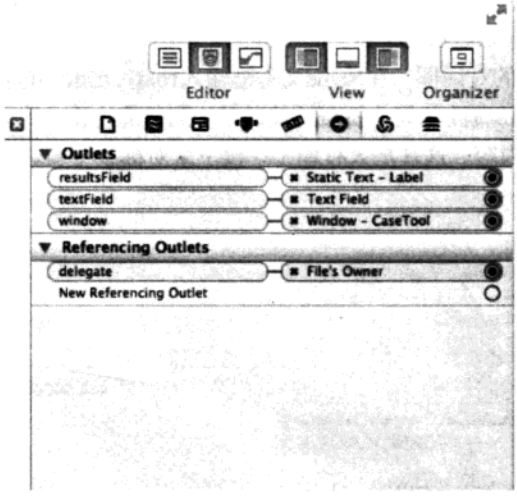


图15-15　复核一遍之前的连接

15.5.2　连接操作（IBAction）

现在，我们将按钮连接到操作，这样按下按钮就会触发代码。还是按住Control键并拖动鼠标，这次是从按钮一直拖到MSCAppDelegate头文件。

> **说明**　拖动路径的方向是使用Interface Builder时一个容易引起混淆的地方。拖动的方向应该是从需要了解某些内容的对象到被了解的对象。
>
> MSCAppDelegate需要了解将哪个NSTextField控件用于用户输入，因此拖动的方法是从MSCAppDelegate到文本框。
>
> 按钮需要了解应告诉哪个对象："嘿！有人按了我！"因此是从按钮拖动到AppController。

按住Contorl键并按下UpperCase按钮，拖动一条直线到头文件的最后一行@property语句下面，如图15-16所示。连接对话框会再次弹出。这次你要将连接类型改成Action。在Name文本框中输入**uppercase**，并点击Connect按钮。这样将会在头文件中创建方法的声明，并会在.m文件中创建方法的实现（内容为空）。

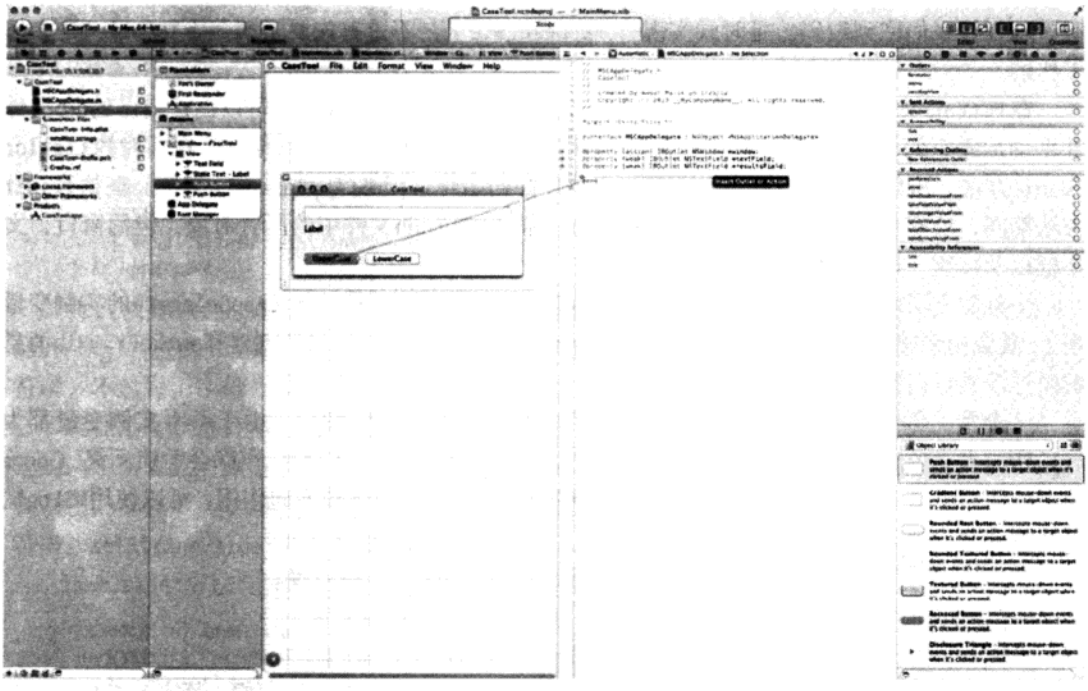


图15-16 连接UpperCase按钮

现在，只要单击该按钮，uppercase:消息就会如我们所想的那样被发送给MSCAppDelegate实例。我们可以在uppercase:方法中执行任何想要的代码。

最后连接LowerCase按钮。按住Control键并从LowerCase按钮一直拖动到头文件，并将连接类型改成Action，连接名称设置为lowercase（如图15-17所示）。这样就完成了所有在Interface Builder中的工作。

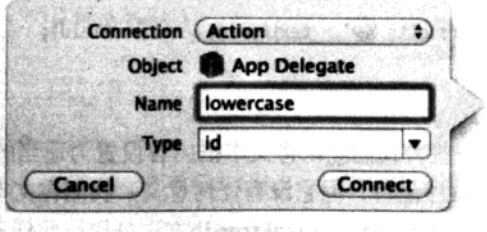


图15-17 再创建一次连接

在布局对象和进行连接时，你可能会动作慢一些。不用担心，熟练之后操作就会变快了。熟练IB的程序员可以在一分钟之内完成所有这些步骤。

15.6　应用程序委托的实现

现在让我们回到编码。是时候实现MSCAppDelegate了，不过首先要了解一下IBOutlet是如何工作的。

在加载nib文件时（MainMenu.nib会在应用程序启动时自动加载，你可以创建自己的nib文件并手动加载它们），存储在nib文件中的任何对象都会被重新创建。这意味着会在后台执行alloc和init方法。所以，当应用程序启动时，会分配并初始化一个MSCAppDelegate实例。在执行init方法期间，所有IBOutlet实例变量都是nil。只有在生成了nib文件中的所有对象（包括窗口、文本框以及按钮）后，所有连接才算完成。

一旦建立了所有连接（也就是将NSTextField等对象的地址添加到MSCAppDelegate的实例变量中），就会向创建的每个对象发送消息awakeFromNib。需要注意，对象的创建和awakeFromNib消息的发送没有任何既定的顺序。

一个常见的错误是试图在init方法中使用IBOutlet执行一些操作。由于所有实例变量都为nil，所有发送给它们的消息都不执行任何操作，所以在init中尝试任何操作都会无疾而终（Cocoa会令你不悦并且浪费你的调试时间）。如果你想知道为什么这些操作不起作用，可以使用NSLog输出实例变量的值，查看它们是否都为nil。

我们继续分析MSCAppDelegate的实现。下面是一些必需的准备步骤。

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

接下来是我们定义的属性。但是代码是从哪里来的？我们没有添加它们。其实某种意义上我们确实添加了它们：当我们将项目从nib文件的控件拖动到头文件时，Interface Builder就为我们添加了这些代码。

然后添加一个init方法，我们并不是真的需要用它，只是要通过它在初始化时显示IBOutlet实例变量的值（也就是nil）。

```
- (id) init
{
 if (nil != (self = [super init]))
 {
 NSLog (@"init: text %@ / results %@", _textField, _resultsField);
 }
 return self;
}
```

为了让用户界面更美观一些，我们应该将文本框内容设置为适当的默认值，而不是Label。虽然Label作为默认值不会影响程序运行，但它没有任何意义。我们在文本框中输入Enter text here，并将结果标签的文本内容预设为Result。awakeFromNib方法是执行此任务的理想位置。

```
- (void)awakeFromNib
{
 NSLog (@"awake: text %@ / results %@", _textField, _resultsField);
 [_textField setStringValue:@"Enter text here"];
 [_resultsField setStringValue:@"Results"];
}
```

NSTextField类拥有一个setStringValue:方法，它接收一个NSString作为参数，并更改文本框的内容来显示该字符串的值。我们正是使用这个方法将文本框的文字更改为对用户更有意义的内容。

现在来看一下操作方法，首先是upppercase。

```
- (IBAction) uppercase: (id) sender
{
    NSString *original = [_textField stringValue];
    NSString *uppercase = [original uppercaseString];
    [_resultsField setStringValue:uppercase];
} // uppercase
```

我们使用stringValue消息从_textField获取最初的字符串，然后将其更改为大写形式。NSString类为我们提供了方便的uppercaseString方法，我们根据接收字符串的内容创建一个新字符串，不过每个字母都是大写形式。然后将该字符串的值赋给_resultsField。

现在是时候进行不可或缺的内存管理检查了。是否每句代码都是正确的？答案是肯定的。创建的两个新对象（原始字符串和大写形式的字符串）都来自于除alloc、copy和new以外的方法，所以它们都位于自动释放池中，并且到时将会被清除。setStringValue:负责复制或保留传入的字符串。setStringValue:方法内部的操作我们无从知晓，但我们知道代码是符合内存管理的。

lowercase:方法与uppercase:方法一样，但它将所有字母都转换成了小写形式。

```
- (IBAction) lowercase: (id) sender {
    NSString *original = [_textField stringValue];
    NSString *lowercase = [original lowercaseString];
    [_resultsField setStringValue:lowercase];
} // lowercase
```

到这就全部结束了。当你运行这个程序时，就会看到一个窗口。输入一个字符串并更改其大小写，如图15-18所示。

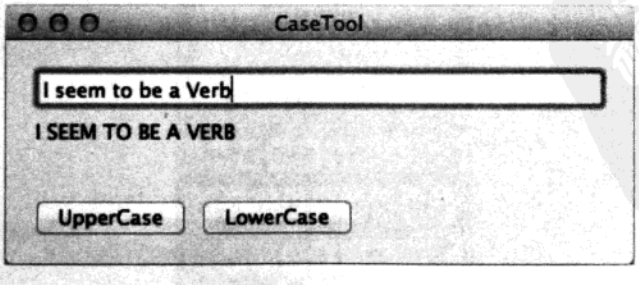


图15-18 完成后的CaseTool程序运行正常

15.7　小结

本章简单介绍了Interface Builder和Application Kit的一些基础知识。我们仅直接使用了一个AppKit类（`NSTextField`），并间接使用了两个类（控制按钮功能的`NSButton`和负责控制窗口的`NSWindow`）。AppKit中有100多种不同的类可供使用，其中许多都可以在Interface Builder中见到。

第16章 UIKit简介

到目前为止，我们学习了如何为OS X平台（即Mac操作系统）编写应用程序。不过关于Mac系统的开发先告一段落，现在我们来看看如何为iOS平台编写应用程序。Mac应用程序使用的是AppKit框架，而iOS应用程序使用的是UIKit框架，它包含了所有的UI组件和构成iOS应用程序的资源。

iOS没有命令行使用权，所以我们无法像在OS X上一样创建命令行工具程序。我们这次要重构在第16章为OS X平台写的CaseTool程序。你甚至不需要iOS设备就可以做这个练习，只需要有附带了iOS SDK的Xcode应用程序就可以了。从Mac App Store上获取的Xcode会默认安装iOS SDK。

请记住iOS在以下方面与OS X不同：

- 没有shell和控制台。
- 应用程序在Mac电脑的模拟器中运行。
- 无法支持一些无UI界面的API。
- 大部分程序员都认为开发iOS应用更加轻松。

完成后的应用看起来会像图16-1中的iPhone程序一样。与第15章的CaseTool程序相似，这个应用程序能将你的字符串全部转换成大写或小写格式。

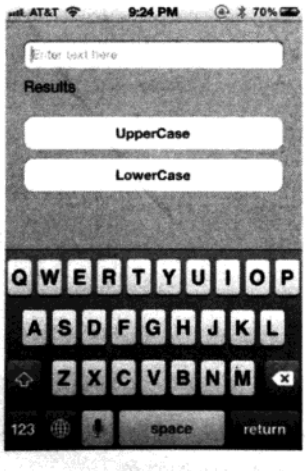


图16-1 iPhone平台上CaseTool程序的预览界面

启动Xcode程序(如果还没打开的话)来开始我们的旅程吧。我们要做的第一件事就是用iOS SDK创建项目。请选择File ➤ New ➤ New Project选项或按下command+shift+N快捷键（如图16-2所示）。

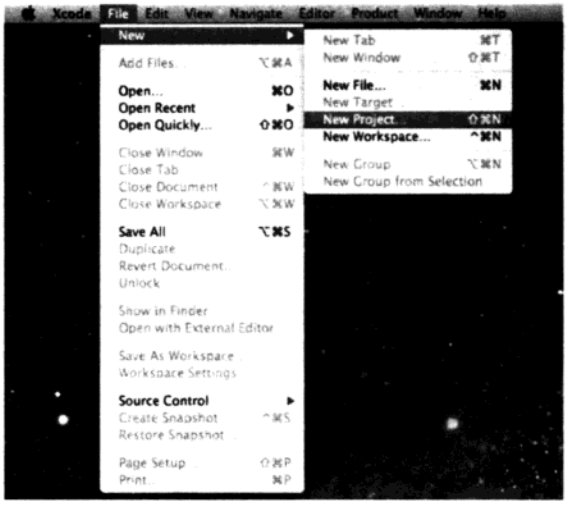


图16-2　创建一个新项目

这样会弹出一个让我们选择项目类型的对话框。我们要在左边显示应用程序类型的列表（如图16-3所示）中选择iOS下的Application选项。

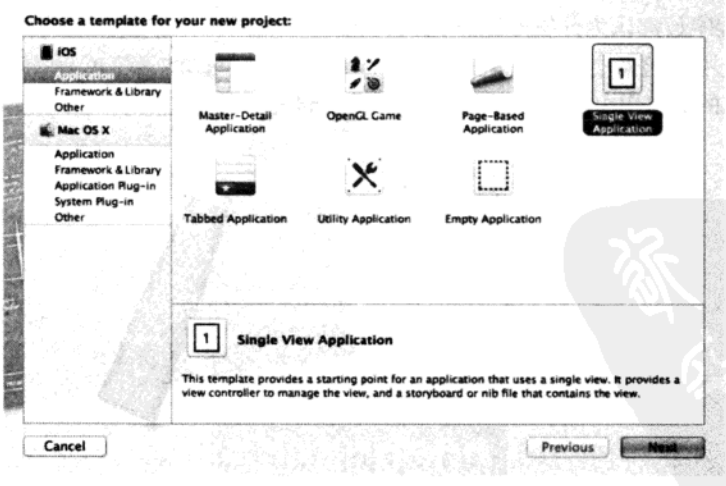


图16-3　选择应用程序类型

接下来选择Single View Application图标。顾名思义，这个应用程序只显示了一个视图。这种类型的应用程序通常非常简单，用户界面也很简单。

以下是其他模板的简单介绍。

- Master-Detail应用程序用一个导航控制器和一个表视图来显示项目列表以及项目的详细信息。
- OpenGL Game可以用来创建打发时间的绝妙游戏。
- Page-Based能帮你创建一个类似书本的应用程序，它拥有翻页动画效果（只支持iPad）。
- Tabbed用来创建多视图应用程序，就像你平常在iPhone上见到的底部有一个标签栏并且每个标签都与一个视图有关联的那种应用程序。
- Utility应用程序模板拥有一个主视图，与Single View Application中的相似，但还多出一个翻转视图。
- Empty应用程序是一个高级选项，如果没有合适的模板，或是你非常了解如何构建你的应用程序，那么可以选择使用这个模板。

选择好Single View Application图标之后，点击Next按钮，你将会看到向你询问应用程序名称的对话框。我们将使用之前在OS X平台上创建的应用程序的名字：CaseTool。如图16-4所示。

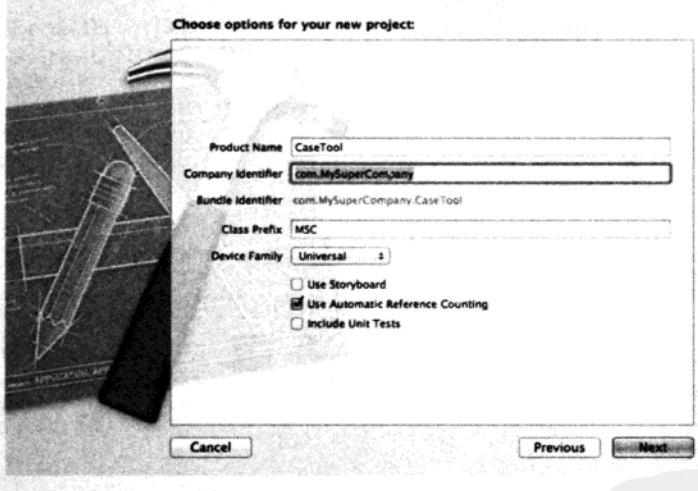


图16-4 为项目命名

在这个界面上，确保Use Storyboard和Include Unit Tests复选框没有被选中，而Use Automatic Reference Counting复选框要被选中。在Device Family下拉列表中，选择Universal项，它意味着应用程序可以同时运行在iPhone、iPod touch和iPad上。选完所有内容之后，点击Next按钮进入下一个界面，选择要在哪一个目录中存储项目（如图16-5所示）。

创建完项目之后，Xcode会打开它并将其展示出来（如图16-6所示）。

浏览一遍文件列表。我们已经拥有了应用程序委托文件，此外还有另一个类和两个nib文件，它们分别是用于iPhone和iPad的。（如果你在创建项目的时候没有选择Universal，那么你只能得到其中一个文件了。）



图16-5　选择存储项目的位置

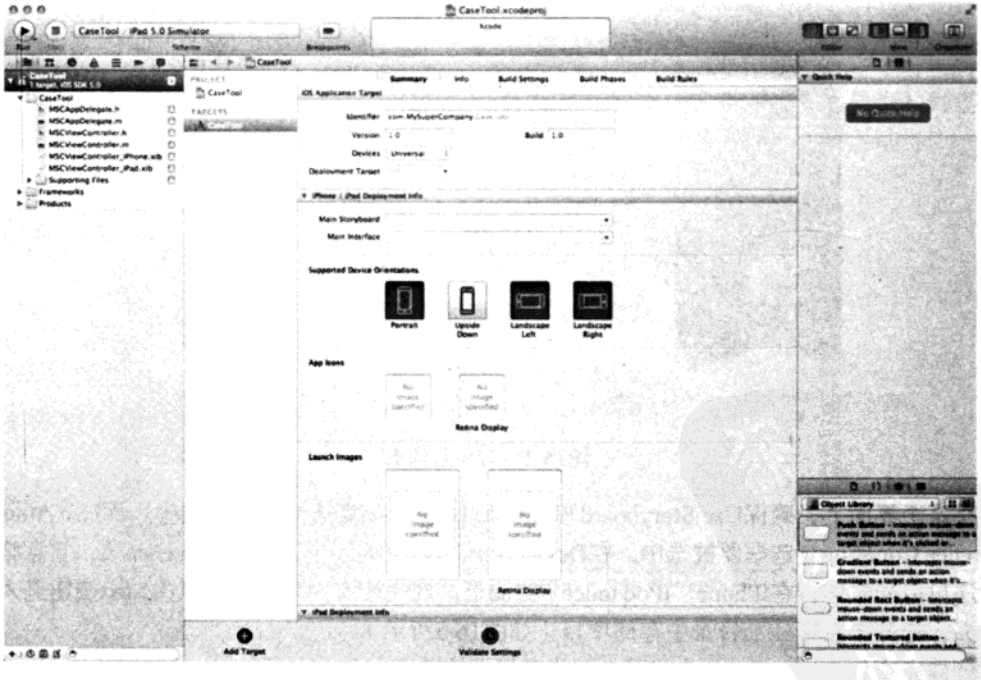


图16-6　我们的新项目

我们来研究一下应用程序委托的头文件。首先是我们的老朋友#import，第15章曾见过它。因为我们这次使用的是UIKit框架而不是AppKit框架，所以iOS界面元素都是以UI为前缀的。我们

还导入了Foundation框架，它在iOS平台上的工作原理与在OS X上相似。因此像NSString这样的类在iOS上是有效的，且使用方式与在OS X平台上类似。

```
#import <UIKit/UIKit.h>
@class MSCViewController;
@interface MSCAppDelegate : UIResponder <UIApplicationDelegate>
@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) MSCViewController *viewController;
@end
```

在代码中，你可以看出我们拥有一个窗口对象和一个视图控制器对象。以下是实现代码。

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
  self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
  // Override point for customization after application launch.
  if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPhone) {
    self.viewController = [[MSCViewController alloc] initWithNibName:
@"MSCViewController_iPhone" bundle:nil];
  } else {
    self.viewController = [[MSCViewController alloc] initWithNibName:
@"MSCViewController_iPad" bundle:nil];
  }
  self.window.rootViewController = self.viewController;
  [self.window makeKeyAndVisible];
  return YES;
}
```

这段代码在窗口创建时便会调用。所有的应用程序都是在主窗口内运行的。接下来，我们创建了视图控制器，根据设备的不同执行不同的代码。然后，将视图控制器的视图添加到了应用程序的层级中。大量应用程序都使用这样的范例代码。事实上，即便你基于其他模板创建应用程序，它的内容依然会像这样。

16.1 视图控制器

我们在第15章曾经谈到过，Cocoa主要使用的是MVC（Model-View-Controller）模式。确实，我们在应用程序拥有一个视图、一个控制器以及一组数据。

我们是从nib文件中获取视图的。这种方式非常便利，因为通过nib文件界面来设计和加载视图要比手动来做快得多。

我们的类MSCViewController是UIViewController的子类。UIViewController知道如何管理视图，比如将其放在屏幕上、调整大小、旋转，等等。如果想要管理视图的话，创建这个类的子类是非常有必要的。

那为什么不直接使用UIViewController类呢？因为我们需要向视图添加一些内容，而UIViewController并不知道有这些东西，所以我们需要创建它的子类并告诉子类如何处理我们新添加的内容。

在Nib文件中添加控件

选择为iPhone定制的nib文件MSCViewController_iPhone.xib。你将会看到以iPhone屏幕尺寸显示的应用程序视图。你还会注意到，库对象面板中显示的内容都是针对iOS平台的（如图16-7所示）。

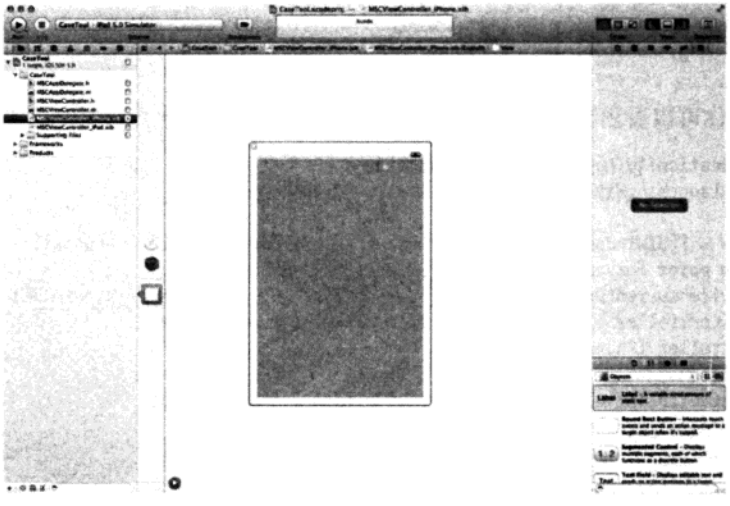


图16-7　针对iPhone设备的nib文件

下面我们开始编辑nib文件。从右下角的对象面板中选择一个TextField对象并将其拖动到视图中。调整其大小以适应屏幕的宽度，调整大小时可以参考引导线（如图16-8所示）。

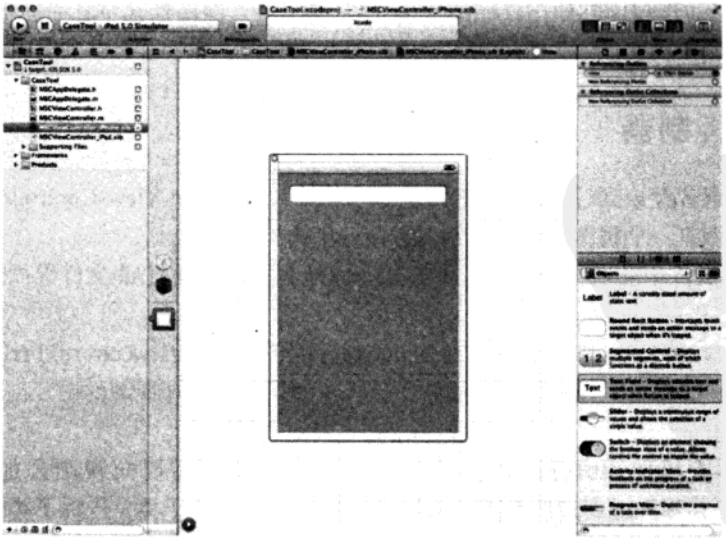


图16-8　添加一个文本框

接下来，我们将一个label对象拖出并放入视图中，然后调整其大小。完成后的预览效果如图16-9所示。

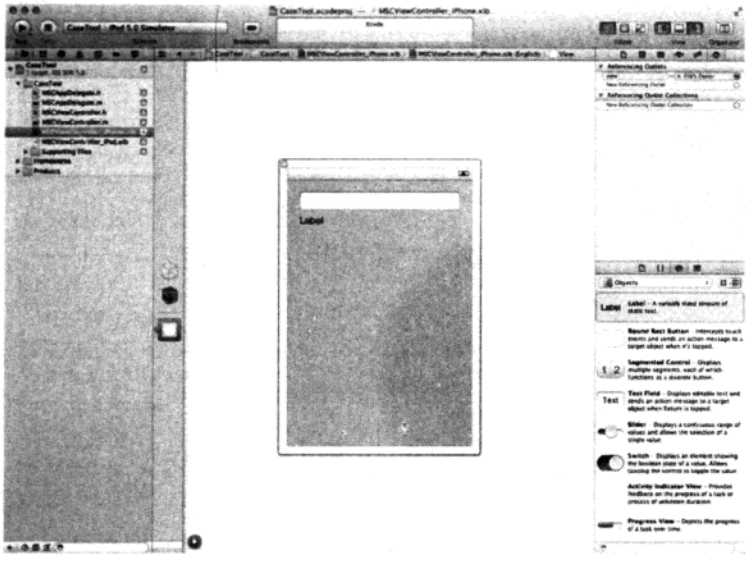


图16-9 将标签放在文本框之下

下一步是添加一个按钮。在对象面板中选择Round Rect Button图标，并将其拖动到视图中文本框和标签的下面。按钮添加完之后，调整按钮大小使其能够匹配文本框的宽度，如图16-10所示。

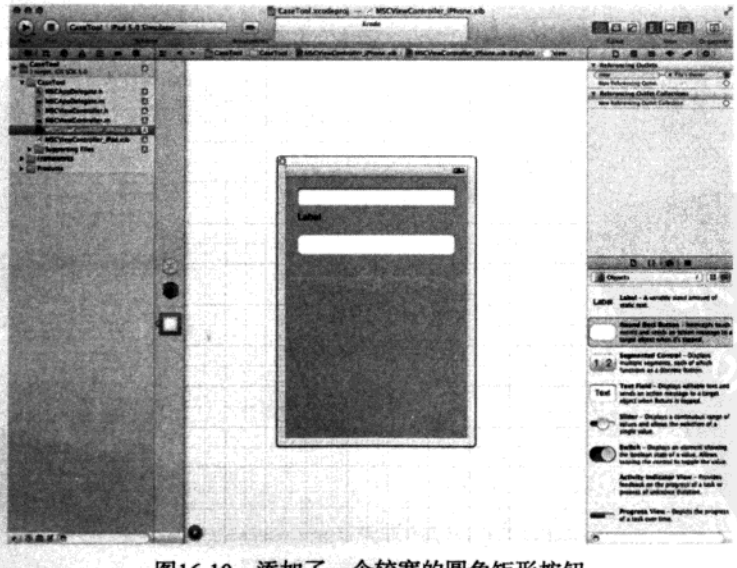


图16-10 添加了一个较宽的圆角矩形按钮

现在双击圆角按钮并给它命名，我们可以叫它UpperCase（如图16-11所示）。

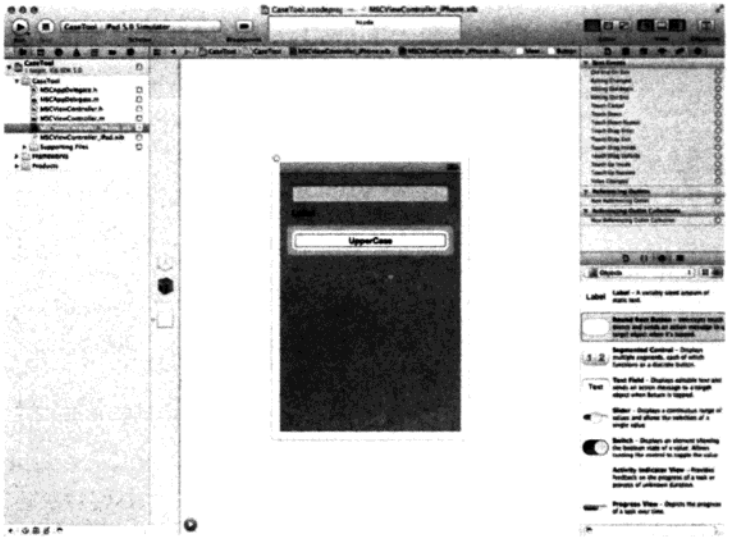


图16-11　为按钮命名

我们有了一个用来将文本转换成大写的按钮，还应该有一个起相反作用的按钮。拖动另一个圆角按钮到视图，并放在第一个按钮的下方。调整其大小使它看上去与第一个按钮相似，并给它命名为LowerCase，如图16-12所示。（有一个快捷方法：选择第一个按钮，并点击Edit ➤ Duplicate选项或按下Command+D快捷键。）

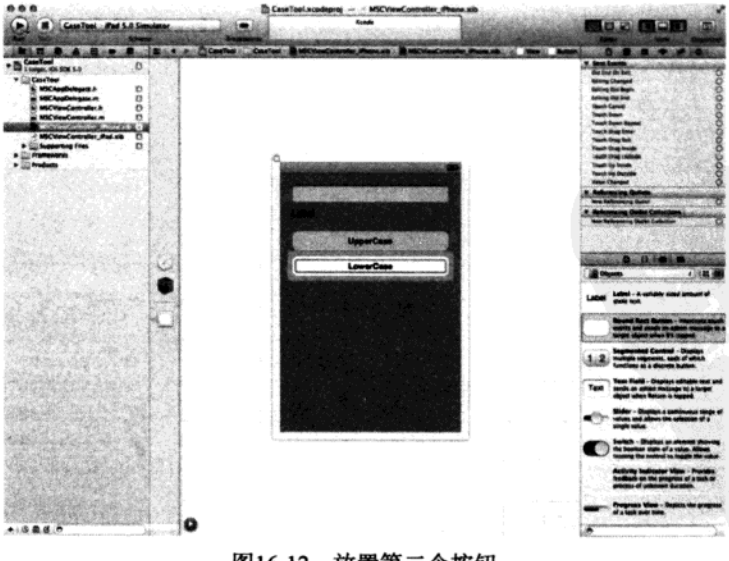


图16-12　放置第二个按钮

当然，你并不一定非要以我们的方式对视图进行布局。你可以按你的想法任意调整对象的大小和位置，所以不必过于拘谨。完成后的视图布局看起来应该如图16-13所示的那样。

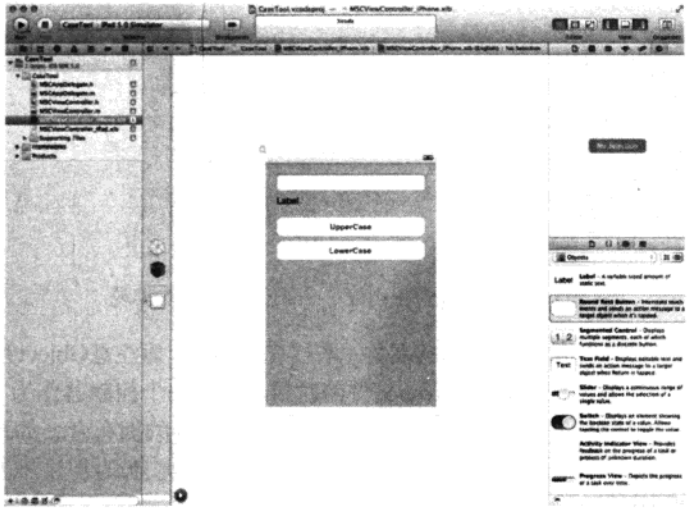


图16-13 视图已经布局完毕

现在我们有了想要的视图，是时候去体验连接输出口和操作的乐趣了。在工具栏的右侧，点击Editor组中间的按钮以打开辅助窗口（也可以使用快捷键Command+Option+Return）。并排显示nib文件和头文件MSCViewController.h，这样便于在两者之间拖动并进行连接。还记得我们在第15章中按住Control键拖动的有趣方式吗？我们再来做一次。按住Control键并将鼠标从视图中的文本框一直拖到头文件中去（如图16-14所示）。

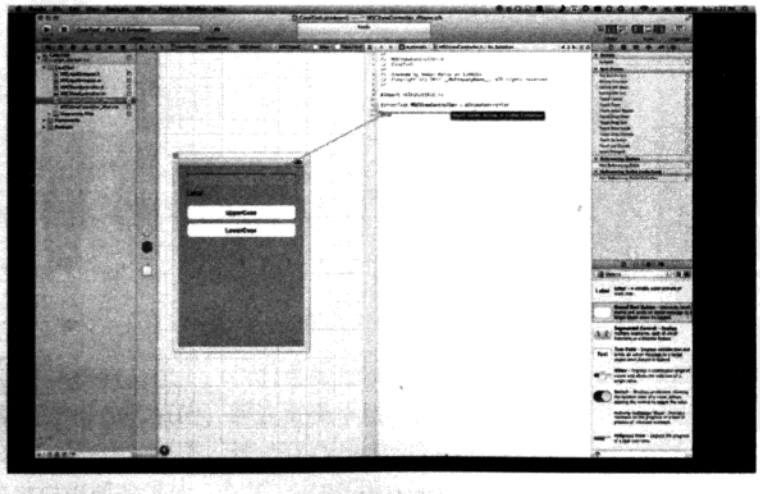


图16-14 按住Control键拖动以创建第一个连接

在头文件的@interface和@end之间的位置放开鼠标（如图16-14），Xcode会弹出一个很有趣的小窗口（如图16-15所示）。

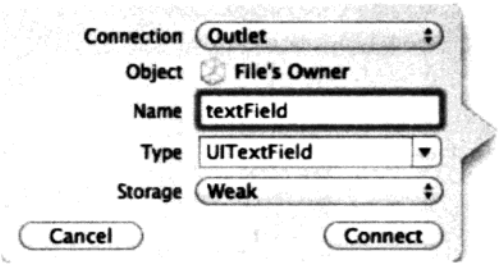


图16-15　窗口会在创建一个新连接的时候出现

图16-15中的窗口要求你填入输出口的名称。不过在填之前，请注意Object旁边的文字是File's Owner。这是什么意思？其实当你在加载nib文件的时候，会有一个控制器作为它的拥有者。当通过模板自动创建nib文件和控制器时，它会让控制器成为nib文件的拥有者。如果你需要的话，可以在nib文件中选择File's Owner图标并更改它的类。不过目前我们暂时保留不变。现在我们要为输出口命名了，输入textField，并点击Connect按钮。

完成这步之后，我们便添加了一个输出口（如图16-16所示）。如果观察头文件，你会发现关于输出口的代码已经被添加了。注意类的名称是UITextField。回想第15章中Mac平台上的例子，我们使用的是NSTextField，它与UITextField类似但并不一样。

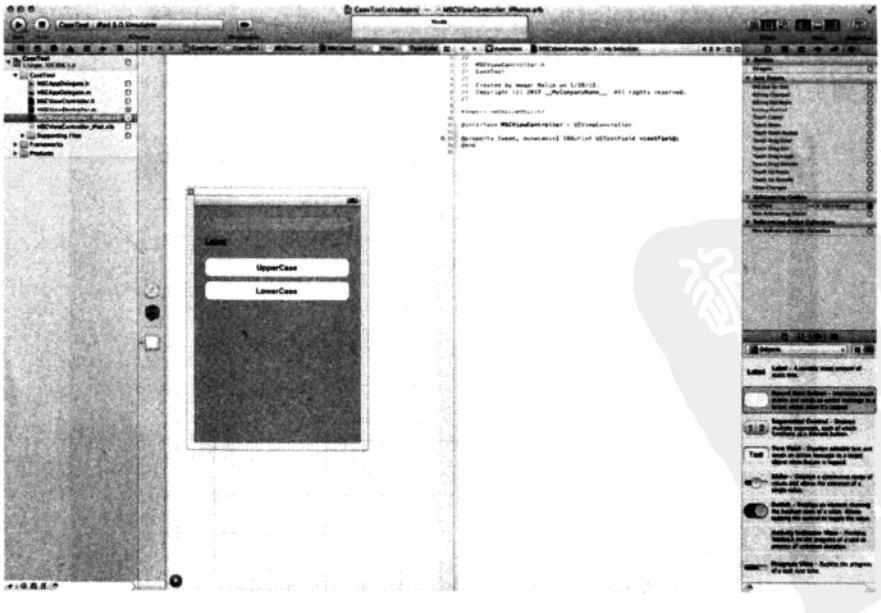


图16-16　添加了一个输出口

现在，让我们连接用来显示结果的标签。按住Control键并将鼠标从标签一直拖动到textField输出口代码的下面。

在Name文本框中输入resultsField（如图16-17所示），并点击Connect按钮。现在，头文件中显示我们拥有了两个输出口（如图16-18所示）。

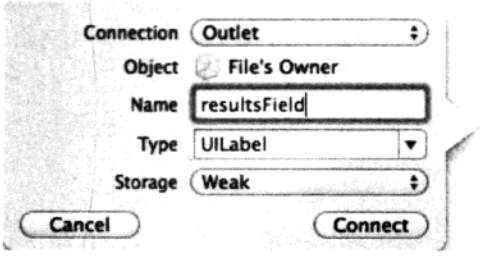


图16-17 为标签输出口命名

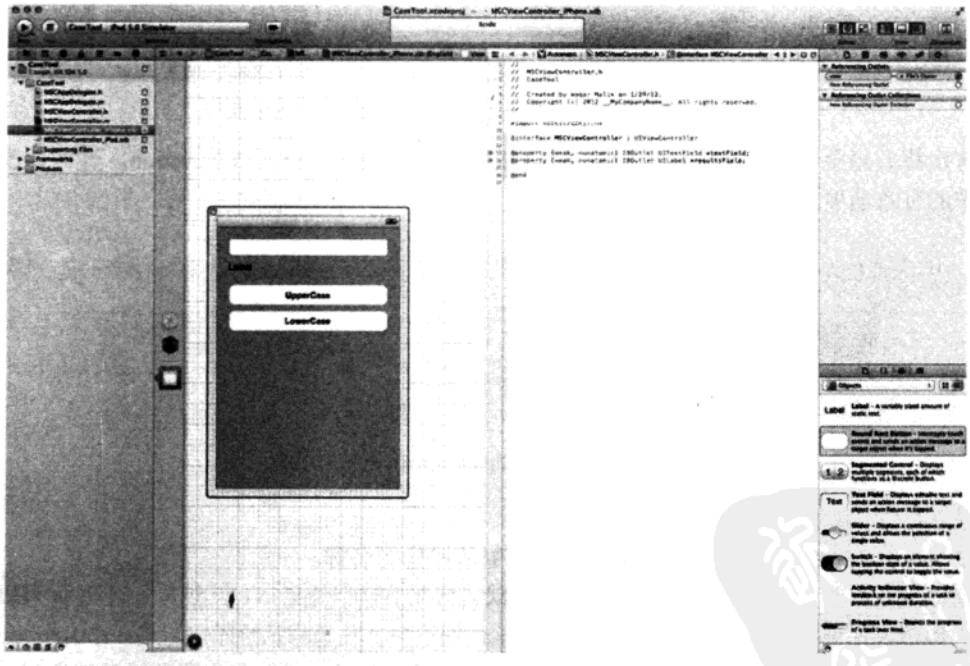


图16-18 我们拥有了两个输出口

现在是时候让按钮工作了。我们需要为按钮创建操作方法。我们不需要为按钮创建输出口，因为输出口是用来修改数值的，而这里根本用不到。

要创建第一个按钮的操作，请按住Control键并将鼠标从UpperCase按钮一直拖动到头文件的resultsField输出口声明的下面，如图16-19所示。

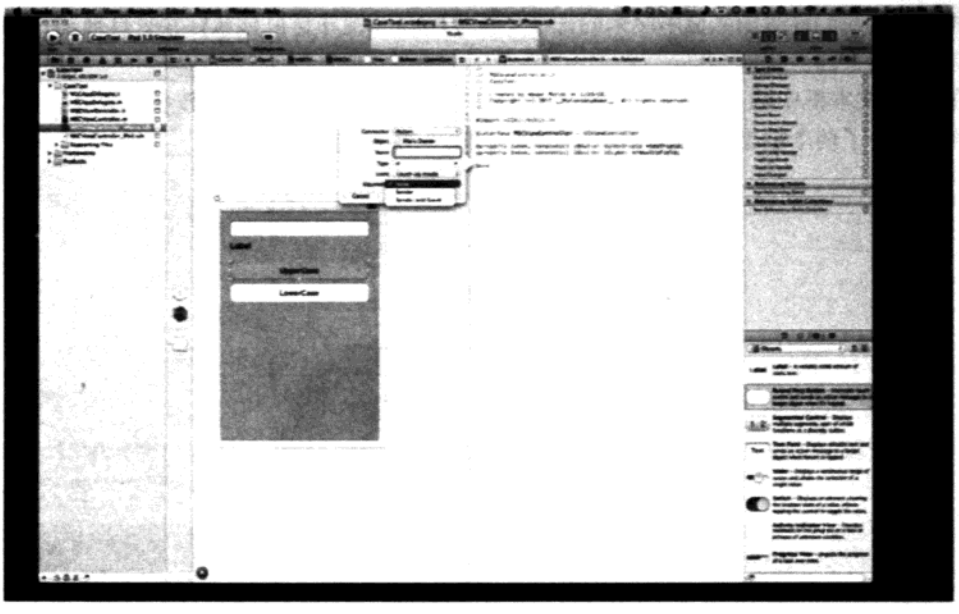


图16-19　按住Control并拖动鼠标以创建第一个按钮操作

这次我们想要创建一个操作，因此要点击Outlet弹出按钮并将其改为Action。这样就改变了方框中的有效选项（如图16-20所示）。

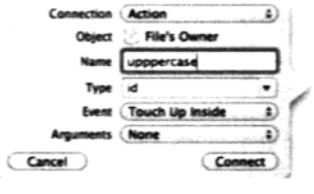


图16-20　用来设置UpperCase按钮的对话框

以下是Connection对话框中各项的信息。

- Name（名称）：该项包含了我们正在创建的操作的名称。
- Type（类型）：操作方法参数的类名称。默认情况下这个值是id（泛型类），不过你可以将其改成发送消息给操作的任何类。在我们这个例子中，发送消息给操作的是一个UIButton类。
- Event（事件）：在OS X和iOS平台之间有很大不同。在iOS中有非常多的事件类型，都是由于触摸界面引起的。在这个例子中，我们打算使用Touch Up Inside事件。这意味着，当手指还在按钮上时停止屏幕触摸，便会调用按钮的操作方法。图16-21展示了iOS上各种有效操作的类型。
- Arguments（参数）：在OS X应用程序中，没有Arguments选项，因为所有的操作都只有一个参数。在iOS环境下，我们有3种参数可以选择：None、Sender（OS X上的默认选项）

和Event（包含了一个UIEvent参数，你可以通过它来决定应该做什么）。我们将为操作选择None选项（如图16-22）。

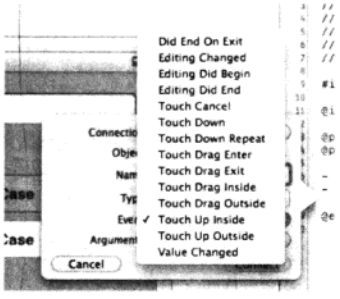


图16-21 选择一个操作的事件类型

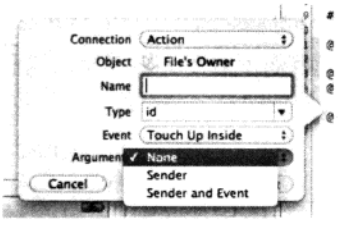


图16-22 在Arguments弹出列表中选择None

完成这一步后，操作的代码便创建好了，代码被添加到头文件中（如图16-23所示）。

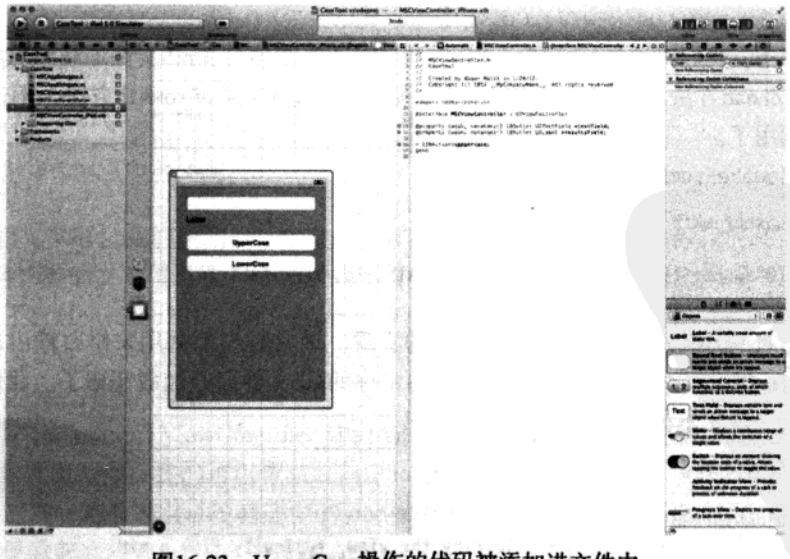


图16-23 UpperCase操作的代码被添加进文件中

接下来，我们要创建LowerCase按钮的操作方法（如图16-24所示）。

现在已经完成了界面连接，让我们手动添加一些代码。（很遗憾，该来的总会来的。）一些样本代码已经自动生成了，比如合成属性以及操作的实现（内容暂时为空）。

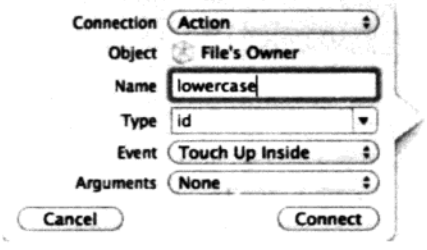


图16-24　创建第二个按钮的操作

回想一下，在第15章中我们在应用程序委托中调用了

```
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
```

这个方法是父类的方法，因此我们不需要去实现它。我们调用这个代码以加载nib文件来访问输出口。

```
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
  self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
  if(nil != self)
  {
    NSLog (@"init: text %@ / results %@", textField, resultsField);
  }
  return self;
}
```

在第15章中，我们讨论过awakeFromNib方法。视图控制器会在nib文件加载和对象初始化完成后调用viewDidLoad方法。一些较早版本的iOS系统还会调用awakeFromNib方法，不过在iOS 5中，已经不会再调用了。

下面添加awakeFromNib方法：

```
- (void)awakeFromNib
{
    NSLog(@"awake: text %@ / results %@", textField, resultsField);
}
```

我们将为viewDidLoad方法添加一些简单的实现。通常它们都是用来修改输出口和设置其他UI用的，当viewDidLoad方法调用了，你便能确定你的nib文件已经加载完毕了。我们打算使用这个方法来设置一些文本框的默认值。

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  // Do any additional setup after loading the view, typically from a nib.
  NSLog (@"viewDidLoad: text %@ / results %@", textField, resultsField);
```

```
    [textField setPlaceholder:@"Enter text here"];
    resultsField.text = @"Results";
}
```

这段代码设定了textField文本框的占位符，它是在用户向文本框输入任意内容前显示的灰色文本。我们还要设置标签的默认值，以便用户知道这里将能看到转换后的结果。

在代码的后面，你将会看到viewDidUnload方法。视图从视图层级中移除后会调用这个方法。为什么要在意这些细节？为了保护内存不会泄漏。

iOS不支持虚拟内存。应用程序只能使用设备中可用的内存。此外如果使用了太多的内存，iOS将会强行退出应用程序以示抗议。我们可以通过viewDidUnload方法在事后清理内存。

在iOS（尤其是iPhone）应用程序中，大多数情况下，一个视图离开后另一个视图会来填满屏幕。这时，先前的那个视图便看不到了，因此就不需要再保留它了。iOS需要卸载视图来节约内存，所以viewDidUnload方法可以帮我们移除视图上的各项内容以节约一些内存。在这个示例中，我们移除了textField和resultsField控件。这两个方法（viewDidLoad和viewDidUnload）在视图的声明周期内只会调用到一次。

还有4个方法（viewWillAppear:、viewDidAppear:、viewWillDisappear:、viewDidDisappear:）也会在视图离开或出现的时候得到调用。每当条件合适的时候它们就会被调用，即便视图没有被卸载。

现在，让我们来添加实际执行大小写转换的代码吧。

```
- (IBAction)uppercase
{
    NSString *original = textField.text;
    NSString *uppercase = [original uppercaseString];
    resultsField.text = uppercase;
}

- (IBAction)lowercase
{
    NSString *original = textField.text;
    NSString *lowercase = [original lowercaseString];
    resultsField.text = lowercase;
}
```

对于iOS的文本框，我们使用一个NSString方法来转换它的文本（就像我们在第15章中所做的那样），并将标签的文本值设置为修改后的字符串。就像在OS X上的那个应用一样，我们在这个例子中使用了stringValue和setStringValue:方法。

我们之前说要创建一个通用应用程序，它可以同时运行在iPhone和iPad上，不过我们还未对iPad的nib文件进行过编辑。现在开始吧。

选择iPad版本的nib文件ViewControler_iPad.xib。添加所需的UI元素使其看起来与我们之前创建的iPhone版本界面相似。我们可以像在之前iPhone版本的界面中一样，一次添加一个元素，也可以为了加快进度，通过按住shift键选中iPhone视图中的所有内容，并复制粘贴到iPad视图中。界面的预览效果看起来应该如图16-25所示。



图16-25　创建iPad版的用户界面

离目标很近了，不过我们还没有完成。如果现在构建并运行应用程序的话，可以在文本框中输入文字也可以点击按钮，不过它们不会有任何响应。为了让它们起作用，必须连接这些输出口和操作。

在iPad版的nib文件中，按住Control键并将鼠标从File's Owner拖动到文本框（如图16-26所示）。

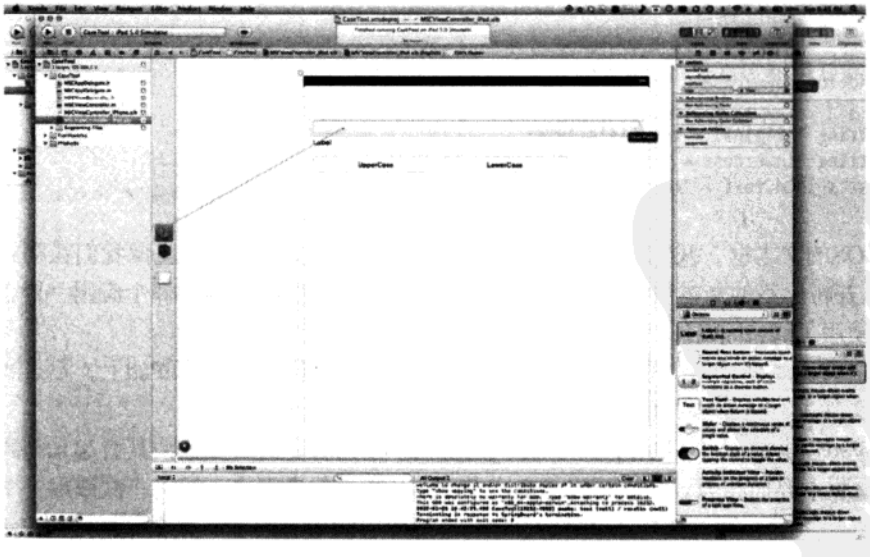


图16-26　按住Control键并将鼠标从File's Owner拖动到文本框

在小型弹出菜单出现后，选择textField（如图16-27所示）。

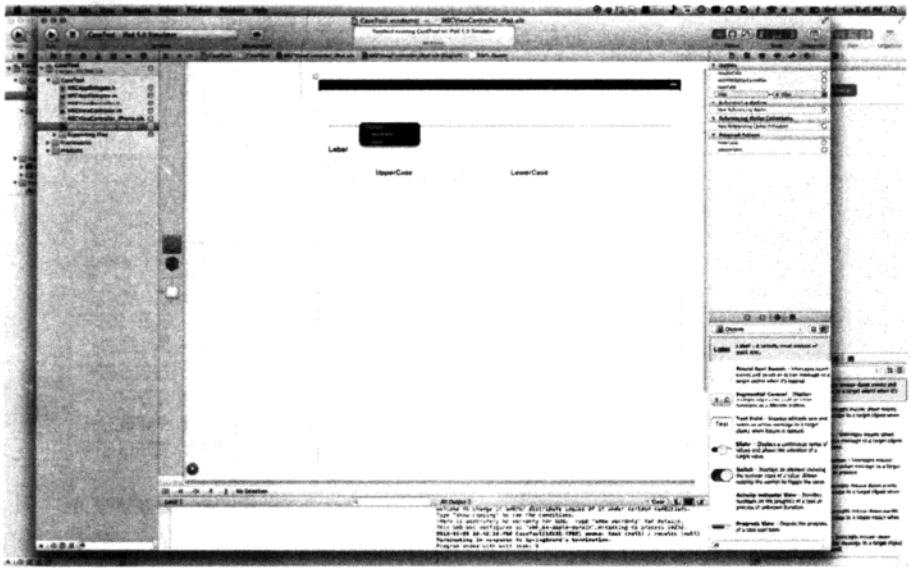


图16-27 在弹出窗口中选择textField输出口

对标签执行同样的行为：按住Control键并将鼠标从File's Owner拖动到标签处（如图16-28所示），然后在弹出的菜单（如图16-29所示）中选择resultsField。

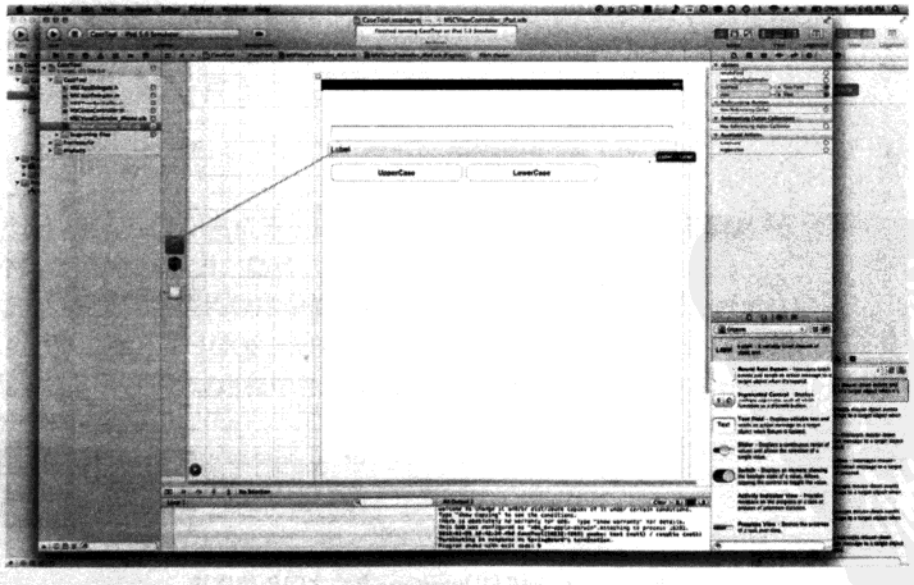


图16-28 创建标签与File's Owner之间的连接

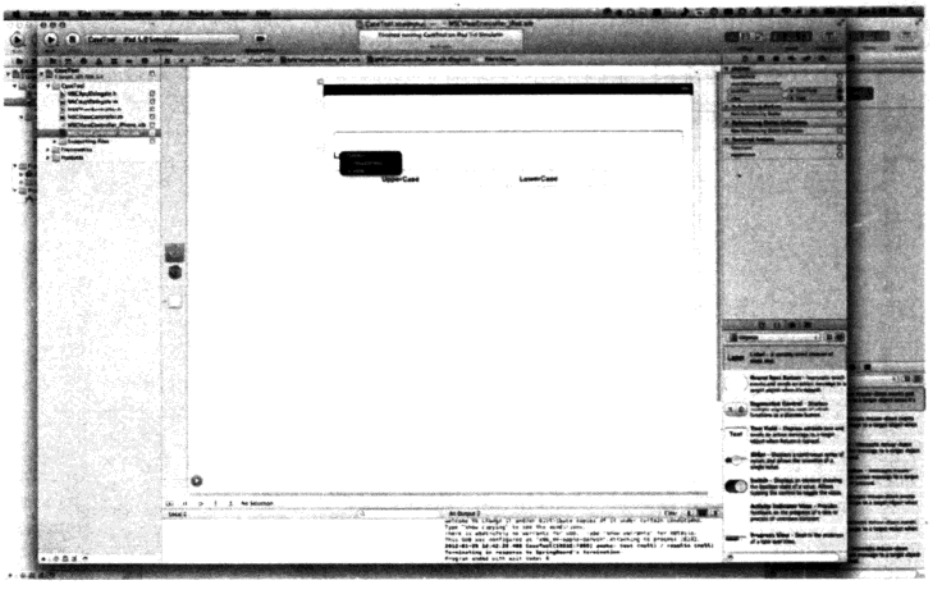


图16-29　连接resultsField输出口

好的，现在我们要让这些控件有所作为！我们打算让它们连接到操作。**按住Control键并将鼠标从UpperCase按钮拖动到File's Owner图标（如图16-30所示），很明显，要在弹出菜单中选择UpperCase。**

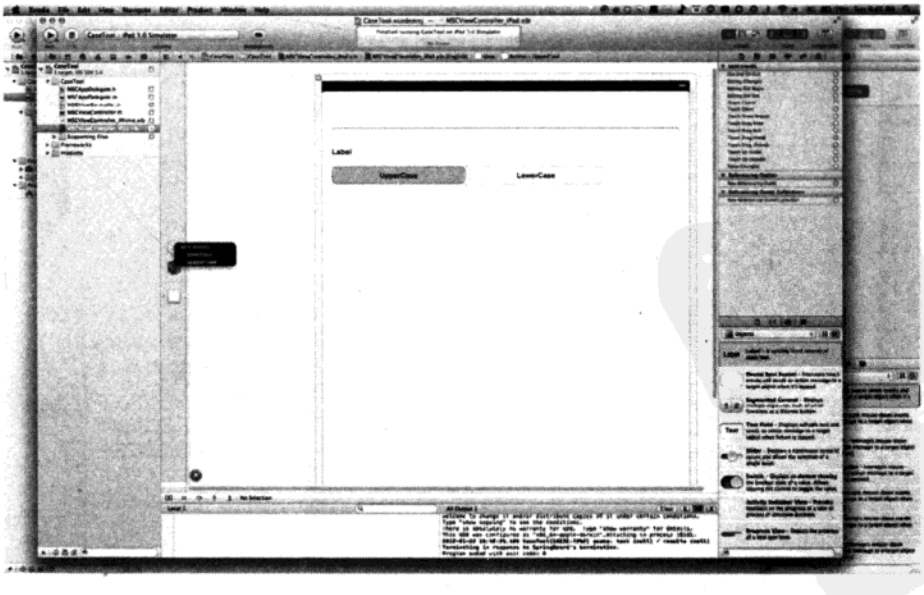


图16-30　在大写按钮和UpperCase方法之间创建连接

对LowerCase按钮执行同样的步骤（如图16-31所示）。



图16-31 连接LowerCase按钮

你可能已注意到弹出菜单的upperCase方法旁边有一个减号。这意味着此方法已经有一个连接了。你可以将某个方法指定到多个操作中去。

最后，点击Scheme弹出菜单，并选择iPad。然后，点击工具栏左边的播放按钮。这样就会启动iPad模拟器并运行这个应用程序。不仅如此，我们还可以在iPhone版本中看到这个新应用程序，方法是在模拟器中选择“硬件 ➤ 设备 ➤ iPhone”。

16.2 小结

为了让应用程序能在模拟器里运行，尽管Xcode帮了很多忙，但我们要做不少功。开发iOS应用程序需要用到大量API，显然，我们学到的还只是一些皮毛。

本章介绍了如何在应用程序中使用视图控制器以及iOS是如何管理视图内存的。还讨论了与iOS有关的类，讲解了iOS与OS X的异同。

假如你打算开发iOS应用程序并且还想学习更多的内容，可以参考图灵公司出版的《iOS 5基础教程》。

现在，你可以更深入地学习Cocoa知识和创建项目了。下一章将探讨Cocoa存储及加载文件的特性。

第 17 章 文件加载与保存

大多数计算机程序（应用程序）在关闭时都会为用户的当前成果创建一个临时的（非永久的）文件，可能是编辑过的图片，也可能是小说的某个章节，或是某个乐队专辑的封面。但无论是哪种情形，用户都会有个文件保存在磁盘上。

标准C函数库提供了函数调用来创建、读取和写入文件，例如open()、read()、write()、fopen()和fread()。这些函数在其他书籍中已有大量描述，因此本书不再讨论。而Cocoa提供了Core Data，它能在后台进行文件的以上所有操作，这里也不会讨论。

那么，我们还有什么可讨论的呢？Cocoa提供了两个处理文件的通用类：属性列表和对象编码，下面具体介绍。

17.1 属性列表

在Cocoa中，有一类名为*属性列表*（property list）的对象，通常简写为plist。这些列表用来放置一些Cocoa能够处理（主要是存储到文件和从文件中加载）的对象。这些属性列表类是NSArray、NSDictionary、NSString、NSNumber、NSDate和NSData，以及它们的可修改形态（只要它们能拥有前缀为Mutable的类）。

前面的章节已经介绍了前四种对象，但没有介绍后两种。因此，本章将重点讨论后两种对象，然后介绍如何将它们归档。你可以在项目17.01 PropertyListing中找到本章该部分的所有代码。

17.1.1 NSDate

程序中经常要处理时间和日期。iPhoto知道你为狗拍照的时间，私人财务应用程序知道处理你银行报表的结账日期。NSDate类是Cocoa中用于处理日期和时间的基础（Foundation）类。

可以使用[NSDate date]来获取当前的日期和时间，它会返回一个能自动释放的对象。因此以下这段代码

```
NSDate *date = [NSDate date];
NSLog (@"today is %@", date);
```

将输出如下结果：

```
today is 2012-01-23 11:32:02 -0400
```

你可以使用一些方法比较两个日期，从而对列表进行排序，还可以获取与当前时间相隔一定时差的日期。比如说，你可能需要24小时之前的确切时间：

```
NSDate *yesterday = [NSDate dateWithTimeIntervalSinceNow: -(24 * 60 * 60)];
NSLog (@"yesterday is %@", yesterday);
```

上面的代码将输出如下结果：

```
yesterday is 2012-01-23 11:32:02 -0400
```

+dataWithTimeIntervalSinceNow:接收一个NSTimeInterval参数，该参数是一个双精度值，表示以秒为单位计数的时间间隔。通过该参数可以指定时间偏移的方式：对于将来的时间，使用时间间隔的整数；对于过去的时间，使用时间间隔的负数（我们在这里就是这么做的）。

> **说明** 如果你想要设定输出结果的时间格式，苹果公司提供了一个叫做NSDateFormatter的类，它符合35号Unicode技术标准，能为用户提供多种时间的显示格式。

17.1.2 NSData

将缓冲区（buffer）的数据传递给函数是C语言中常见的操作。通常是将缓冲区的指针和长度传递给某个函数。另外，C语言中可能会出现内存管理问题。例如，如果缓冲区已经被动态分配，那么当它不再使用时，由谁来负责将其清除？

Cocoa提供了NSData类，该类可以包含大量字节。你可以获得数据的长度和指向字节起始位置的指针。因为NSData是一个对象，所以常规的内存管理对它是有效的。因此，如果想将数据块传递给一个函数或方法，可以通过传递一个支持自动释放的NSData来实现，而无需担心内存清理的问题。下面的NSData对象将保存一个普通的C字符串（一个字节序列），然后输出数据。

```
const char *string = "Hi there, this is a C string!";
NSData *data = [NSData dataWithBytes: length: strlen(string) + 1];
NSLog (@"data is %@", data);
```

输出结果如下所示：

```
data is <48692074 68657265 2c207468 69732069 73206120 43207374 72696e67 2100>
```

上面的输出结果有点特别，但是如果你有ASCII表（打开终端，并键入命令man ascii就可以找到该表），就可以看到，这个十六进制数据块实际就是我们的字符串，0x48表示字符H，0x69表示字符i等。-length方法给出字节数量，-byte方法给出指向字符串起始位置的指针。注意到

+dataWithBytes:调用中的+1了吗？它用于包含C字符串所需的尾部的零字节。还要注意NSLog输出结果末尾的00。通过包含零字节，就可以使用%s格式的说明符输出字符串

```
NSLog (@"%d byte string is '%s'", [data length], [data bytes]);
```

输出结果如下所示：

```
30 byte string is "Hi there, this is a C string!"
```

NSData对象是不可变更的，创建后就不能改变。你可以使用它们，但不能更改其中的内容。不过NSMutableData支持在数据内容中添加和删除字节。

17.1.3　写入和读取属性列表

你已经见过了所有的属性列表类，我们要如何使用它们呢？集合型属性列表类（NSArray和NSDictionary）具有一个-writeToFile:atomically:方法，用于将属性列表的内容写入文件。NSString和NSData也具有writeToFile:atomically:方法，不过只能写出字符串或数据块。

因此，我们可以将字符串存入一个数组，然后保存它。

```
NSArray *phrase;
phrase = [NSArray arrayWithObjects: @"I", @"seem", @"to", @"be", @"a", @"verb", nil];
[phrase writeToFile: @"/tmp/verbiage.txt" atomically: YES];
```

现在看一下文件/tmp/verbiage.txt，你应该会看到如下代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd"> <plist version="1.0"> <array>
<string>I</string>
<string>seem</string>
<string>to</string>
<string>be</string>
<string>a</string>
<string>verb</string> </array> </plist>
```

以上代码虽然有些烦琐，但正是我们要保存的内容：一个字符串数组。如果保存的数组中是包含了各种字符串数组、数字和数据的字典，那么这些属性列表文件的内容将相当复杂。Xcode也包含了一个属性列表编辑器，所以你可以查看plist文件并进行编辑。如果浏览一下操作系统，你会发现许多属性列表文件和系统配置文件，如主目录的资源库/Preferences下所有的首选项文件和/System/资源库/LaunchDaemons下的系统配置文件。

> **说明**　有些属性列表文件，特别是首选项文件，是以压缩的二进制格式存储的。通过使用plutil命令：plutil -convert xml1文件名.plist，可以将这些文件转换成人可以理解的字面形式。

现在我们的磁盘中已经有了verbiage.txt文件，可以使用+arrayWithContentsOfFile:方法读取该文件。代码如下所示。

```
NSArray *phrase2 = [NSArray arrayWithContentsOfFile: @"/tmp/verbiage.txt"];
NSLog (@"%@", phrase2);
```

输出结果刚好与前面保存的形式相匹配：

```
(
I,
seem,
to,
be,
a,
verb
)
```

> **说明** 注意到writeToFile:方法中的单词atomically了吗？atomically:参数的值为BOOL类型，它会告诉Cocoa是否应该首先将文件内容保存在临时文件中，再将该临时文件和原始文件交换。这是一种安全机制：如果在保存过程中出现意外，不会破坏原始文件。但这种安全机制需要付出一定的代价：在保存过程中，由于原始文件仍然保存在磁盘中，所以需要使用双倍的磁盘空间。你应该尽量使用atomically的方式保存文件，除非保存的文件容量非常大，会占用用户大量的硬盘空间。

如果能将数据归结为属性列表类型，则可以使用这些非常便捷的方法调用来将内容保存到磁盘中，供以后读取。如果你正在从事一个新创意或设计一个新项目，可以使用这些便捷方法来快速编写和运行程序。即使你没有用到对象，只是想把数据块保存到磁盘中，也可以使用NSData来简化工作。只需要将该数据包装到一个NSData对象中，然后在NSData对象上调用writeToFile:atomically:方法即可。

这些函数的一个缺点是不会返回任何错误信息。如果无法加载文件，你只能从方法中获得一个nil指针，但无法得知出现错误的具体原因。

17.1.4 修改对象类型

需要注意，当使用集合类从某文件读取数据时，你无法修改数据的类型。一种解决方法是强制转换，遍历plist文件的内容并创建一个平行结构的可修改对象。不过还有另一种方法。事实上，假如你在Cocoa中遇到了做起来很复杂的东西，说不定苹果公司已经提供了更加简单的解决方法。

接下来要用到的类叫做NSPropertyListSerialization（读起来相当绕口）。正如名字所提示的，它可以为存储和加载属性列表的行为添加很多你需要的设定项。

尤其要注意propertyListFromData:mutabilityOption:format:errorDescription:方法。它能

把plist数据返回给你，并且能在出现异常的时候提供错误信息。

以下是将plist数据内容以二进制形式写入文件的代码：

```
NSString *error = nil;
NSData *encodedArray = [NSPropertyListSerialization dataFromPropertyList:capitols
       format:NSPropertyListBinaryFormat_v1_0 errorDescription:&error];
[encodedArray writeToFile:@"/tmp/capitols.txt" atomically:YES];
```

如你所见，我们将数组数据转换成了NSData类型并写入了文件中。

将数据读取回内存要多执行一步，即指定文件的类型。我们创建了一个指针，如果文件格式与指定的类型不同，可以换用原格式类型的指针，也可以将读取的内容转换成新的格式。

```
NSPropertyListFormat propertyListFormat = NSPropertyListXMLFormat_v1_0;
NSString *error = nil;
NSMutableArray *capitols = [NSPropertyListSerialization propertyListFromData:data
        mutabilityOption:NSPropertyListMutableContainersAndLeaves
                  format:&propertyListFormat
        errorDescription:&error];
```

其中一个选项就是以什么方式来读取数据：我们是否想要能够修改plist的类型？我们是想获取列表结构还是仅获取二进制数据？

17.2　编码对象

不幸的是，你无法总是将对象信息表示为属性列表类。要是能将所有对象都表示为包含数组的字典，也就没有必要再创建自定义的类了。所幸，Cocoa具备一种将对象转换成某种格式并保存到磁盘中的机制。对象可以将它们的实例变量和其他数据编码为数据块，然后保存到磁盘中。这些数据块以后还可以读回内存中，并且还能基于保存的数据创建新对象。这个过程被称为*编码与解码*（encoding and decoding），也可以叫做*序列化与反序列化*（serialization and deserialization）。

如果你还记得，前一章在介绍Interface Builder时，我们从库中将对象拖到窗口，这些对象会被存储到nib文件中。换句话说，NSWindow和NSTextField对象都被序列化并保存到磁盘中。当程序运行时，会将nib文件加载到内存中，对象被反序列化，新的NSWindow与NSTextField对象会被创建并建立关系。

你可能会猜到，通过采用NSCoding协议，可以使自己的对象实现相同的功能。该协议与下面的代码类似。

```
@protocol NSCoding
- (void) encodeWithCoder: (NSCoder *) encoder;
- (id) initWithCoder: (NSCoder *) decoder;
@end
```

通过采用该协议，可以实现两种方法。当对象需要保存自身时，就会调用-encodeWithCoder方法；当对象需要加载自身时，就会调用-initWithCoder:方法。

那么这个编码器是什么呢？NSCoder是一个抽象类，它定义了一些有用的方法，便于对象与

NSData之间的转换。你完全没必要创建一个新的NSCoder对象，因为它实际上并不会起作用。不过有一些具体实现的NSCoder子类可以用来编码和解码对象，我们将使用其中两个子类NSKeyedArchiver和NSKeyedUnarchiver。

示例可能是理解这些内容的最简单方式。你可以在17.02 SimpleEncoding项目中查看所有的代码。

首先来看一下包含了一些实例变量的简单类。

```
@interface Thingie : NSObject <NSCoding>
{
 NSString *name;
 int magicNumber;
 float shoeSize;
 NSMutableArray *subThingies;
}
@property (copy) NSString *name;
@property int magicNumber;
@property float shoeSize;
@property (retain) NSMutableArray *subThingies;
- (id)initWithName: (NSString *) n magicNumber: (int) mn shoeSize: (float) ss;
@end // Thingie
```

你应该很熟悉上面这段代码，我们有4个对象和标量类型的实例变量，其中包括一个集合类。属性默认是可读写的，因此我们没有在属性的定义中指定该特性。每个实例变量都有一个公共属性声明，此外还有一个信息量略大的init方法，它可以独力创建一个新的Thingie对象。

请注意Thingie类采用了NSCoding协议，这意味着我们将要实现encodeWithCoder和initWithCoder方法。不过现在我们先将这两个方法的内容设置为空。

```
@implementation Thingie
@synthesize name;
@synthesize magicNumber;
@synthesize shoeSize;
@synthesize subThingies;
- (id)initWithName: (NSString *) n magicNumber: (int) mn shoeSize: (float) ss
{
 if (self = [super init])
 {
  self.name = n;
  self.magicNumber = mn;
  self.shoeSize = ss;
  self.subThingies = [NSMutableArray array];
 }
return (self);
}
- (void) dealloc
{
 [name release];
 [subThingies release];
 [super dealloc];
} // dealloc
- (void) encodeWithCoder: (NSCoder *) coder
{
```

```
// nobody home
} // encodeWithCoder
- (id) initWithCoder: (NSCoder *) decoder
{
 return (nil);
} // initWithCoder
- (NSString *) description
{
 NSString *description =
 [NSString stringWithFormat: @"%@: %d/%.1f %@", name,
   magicNumber, shoeSize, subThingies];
 return (description);
} // description
@end // Thingie
```

该段代码初始化了一个新对象，清除了我们创建的所有东西，并且为NSCoding协议的方法创建了结构框架以避免编译器报错，最后返回了描述信息的方法。

请注意，在init方法中，我们在赋值表达式的左侧使用了self.attribute。记住这实际上意味着我们正在调用这些属性的访问方法（访问方法由@synthesize关键字创建）。我们并没有直接为实例变量赋值。通过这种方式接收我们创建的NSString和NSMutableArray对象，可以对它们进行适当的内存管理，因此就不需要再为其提供内存管理方法了。

因此，在main()函数中，我们创建一个Thingie对象并输入其内容。

```
Thingie *thing1;
thing1 = [[Thingie alloc] initWithName: @"thing1" magicNumber: 42 shoeSize: 10.5];
NSLog (@"some thing: %@", thing1);
```

以上代码将输出如下结果：

```
some thing: thing1: 42/10.5 ( )
```

真有趣。现在，我们来将这个对象归档。按以下方式实现Thingie类的encodeWithCoder:方法。

```
- (void) encodeWithCoder: (NSCoder *) coder
{
 [coder encodeObject: name forKey: @"name"];
 [coder encodeInt: magicNumber forKey: @"magicNumber"];
 [coder encodeFloat: shoeSize forKey: @"shoeSize"];
 [coder encodeObject: subThingies forKey: @"subThingies"];
} // encodeWithCoder
```

我们将使用NSKeyedArchiver把对象归档到NSData中。顾名思义，KeyedArchiver使用键/值来保存对象的信息。Thingie的-encodeWithCoder方法会使用与每个实例变量名称匹配的键对其进行编码。你也可以不这么做。你可以用像flarblewhazzit这样无规则的文字作为键来编码name实例变量。保证键的名称与实例变量的名称相似有助于识别它们之间的映射关系。

可以使用像上面代码那样的字面量字符串作为编码键，也可以定义一个常量以避免录入错误。可以使用#define kSubthingiesKey @"subThingies"来定义常量，也可以使用文件的局部变量，

比如static NSString *kSubthingiesKey = @"subThingies";所定义的静态变量。

请注意，每种类型的encodeSomething:forKey:方法都不同。需要确保你的类型使用的是正确的编码方法。对于所有的Objective-C对象类型，都要使用encodeObject:forKey:方法。

如果需要恢复某个归档后的对象，可以使用decodeSomethingForKey方法。

```
- (id) initWithCoder: (NSCoder *) decoder
{
 if (self = [super init]) {
  self.name = [decoder decodeObjectForKey: @"name"];
  self.magicNumber = [decoder decodeIntForKey: @"magicNumber"];
  self.shoeSize = [decoder decodeFloatForKey: @"shoeSize"];
  self.subThingies = [decoder decodeObjectForKey: @"subThingies"];
 }
 return (self);
} // initWithCoder
```

initWithCoder:和其他init方法一样，在为对象执行操作之前，需要使用超类进行初始化。可以采用两种方式，具体取决于父类。如果父类采用了NSCoding协议，则应该调用[super initWithCoder: decoder]，否则只需要调用[super init]。NSObject不采用NSCoding协议，因此我们可以使用简单的init方法。

当你使用decodeIntForKey:方法时，会把一个int值从decoder中取出；当你使用decodeObjectForKey:方法时，会把一个对象从decoder中取出，如果里面还有嵌入的对象，就会对其递归调用initWithCoder:方法。内存管理的工作方式与你预想的一样：通过alloc、copy和new以外的方法获取对象，因此可以假设对象能被自动释放。属性声明确保了所有的内存管理都在正确地执行。

你将会注意到，编码和解码的顺序和示例对象的顺序完全相同。当然，你也可以不这么做，这只是一种简便的习惯，目的在于确保每个对象都进行了编码和解码操作，没有对象被漏掉。使用键进行调用的原因之一就是可以将示例对象按任意顺序放入或取出。

接下来将实际操作这些对象。我们使用前面创建的thing1对象，并将其归档。

```
NSData *freezeDried;
freezeDried = [NSKeyedArchiver archivedDataWithRootObject: thing1];
```

类方法+archivedDataWithRootObject:对这个对象进行了编码。首先，它在后台创建了一个NSKeyedArchiver实例，然后传递给了对象thing1的-encodeWithCoder方法。在thing1编码自身的实例变量时，也会引发其他对象对自身进行编码，比如字符串、数组及放入数组中的任何对象。当所有的对象完成了键值编码后，会被放入一个NSData并返回。

如果愿意，可以使用-writeToFile:atomically:方法将这个NSData对象保存到磁盘中。而在这里，我们只是释放了thing1，并通过归档的freezeDried对象重新创建了thing1对象并输出其内容。

```
[thing1 release];
thing1 = [NSKeyedUnarchiver unarchiveObjectWithData: freezeDried];
NSLog (@"reconstituted thing: %@", thing1);
```

输出结果与之前我们看到的完全相同：

```
reconstituted thing: thing1: 42/10.5 ( )
```

在契诃夫话剧的第一幕，当看到墙上有一只枪时，你一定会感到好奇。[1]同样，你可能会对叫做subThing的可变数组感到好奇。我们可以将对象放入该数组中，当数组被编码时，这些对象将被自动编码。NSArray中encodeWithCoder:方法的实现会对所有对象调用encodeWithCoder方法，直到所有对象都被编码。让我们向thing1中的subThingies添加一些对象。

```
Thingie *anotherThing;
anotherThing = [[[Thingie alloc]
   initWithName: @"thing2"
   magicNumber: 23
   shoeSize: 13.0] autorelease];
[thing1.subThingies addObject: anotherThing];
anotherThing = [[[Thingie alloc]
   initWithName: @"thing3"
   magicNumber: 17
   shoeSize: 9.0] autorelease];
[thing1.subThingies addObject: anotherThing];
NSLog (@"thing with things: %@", thing1);
```

以上代码将输出thing1和subthingies中的内容：

```
thing with things: thing1: 42/10.5 (
 thing2: 23/13.0 (
 ),
 thing3: 17/9.0 (
 )
)
```

编码和解码的工作机制完全相同：

```
freezeDried = [NSKeyedArchiver archivedDataWithRootObject: thing1];

thing1 = [NSKeyedUnarchiver unarchiveObjectWithData: freezeDried];
NSLog (@"reconstituted multithing: %@", thing1);
```

并且输出结果同之前的完全一样。

如果被编码的数据中含有循环会怎样？例如thing1就在自己的subThingies数组中会怎么样？thing1里面包含了数组，数组里面又包含了thing1，而其中又有数组，数组里面又有thing1，不停地这样重复的话该怎么编码？幸好Cocoa的归档和反归档实现非常智能，对象循环也可以进行保存或恢复。

如果要进行实验的话，可以将thing1放入其自身的subThingies数组中。

```
[thing1.subThingies addObject: thing1];
```

① 契诃夫曾经说过，第一幕墙上挂了一把枪，那么在幕终之前，这把枪一定会放枪，不然这把枪就不必挂在这儿。——译者注

但是不要尝试在thing1中使用NSLog，NSLog不够智能，不能检测对象循环，因此它将陷入一个无限的递归，试图创建日志信息，最终会导致成千上万的-description调用进入调试器中。

不过，如果对thing1进行编码和解码，它可以完美地运行，也不会陷入混乱中。

```
freezeDried = [NSKeyedArchiver archivedDataWithRootObject: thing1];
thing1 = [NSKeyedUnarchiver unarchiveObjectWithData: freezeDried];
```

17.3 小结

本章介绍到Cocoa提供了两种方式来加载和保存文件：属性列表以及对象编码。属性列表数据类型是一种集合类，它知道如何加载和保存自身。如果对象集合中的对象类型全为属性列表，可以使用这种便捷的函数将它们保存到磁盘中，或者从磁盘中读出。

和大多数Cocoa程序员一样，如果你拥有自己的对象，而这些对象又不是属性列表类型，可以采用NSCoding协议和实现方法来编码和解码对象：将大量对象转换成NSData，然后保存到磁盘中供以后读取。通过这种NSData，可以重新创建对象。

下一章将介绍键/值编码，它让你能在更高的抽象级别上与对象进行交互。

第 18 章 键/值编码 18

现在再来看一下间接机制。许多编程技术都基于间接机制，包括整个面向对象编程领域。本章将介绍另一种间接机制，这种机制不属于Objective-C语言的特性，而是Cocoa提供的一种特性。

到目前为止，我们已经介绍了通过直接调用方法、属性的点表示法或设置实例变量来直接更改对象状态。键/值编码（key-value coding）是一种间接更改对象状态的方式，许多人亲切地称其为KVC，其实现方法是使用字符串表示要更改的对象状态。本章通篇都会讨论键/值编码。

一些更加高级的Cocoa特性，比如Core Data和Cocoa Bindings（本书不会介绍），也在基础机制中使用了KVC功能。

18.1 入门项目

我们将再次使用大家熟悉的CarParts示例。参考一下项目18.01 Car-Value-Coding中的代码。为了让项目更加生动，我们向Car类添加了一些属性，例如常见的品牌和型号。为了保持统一，我们将appellation重命名为name。

```
@interface Car : NSObject <NSCopying>
{
 NSString *name;
 NSMutableArray *tires;
 Engine *engine;
 NSString *make;
 NSString *model;
 int modelYear;
 int numberOfDoors;
 float mileage;
}
@property (readwrite, copy) NSString *name;
@property (readwrite, retain) Engine *engine;
@property (readwrite, copy) NSString *make;
@property (readwrite, copy) NSString *model;
@property (readwrite) int modelYear;
@property (readwrite) int numberOfDoors;
@property (readwrite) float mileage;
…
@end // Car
```

我们添加了@synthesize指令，这样编译器将会自动生成setter和getter方法。

```
@implementation Car
@synthesize name;
@synthesize engine;
@synthesize make;
@synthesize model;
@synthesize modelYear;
@synthesize numberOfDoors;
@synthesize mileage;
```

我们还修改了-copyWithZone方法，以匹配新的属性。

```
- (id) copyWithZone: (NSZone *) zone
{
 Car *carCopy;
 carCopy = [[[self class]allocWithZone: zone] init];
 carCopy.name = name;
 carCopy.make = make;
 carCopy.model = model;
 carCopy.numberOfDoors = numberOfDoors;
 carCopy.mileage = mileage;
// plus copying tires and engine, code in Chapter 13.
```

此外，还更改了-description方法，以输出这些新属性并省略Engine和Tire的详细输出。

```
- (NSString *) description
{
  NSString *desc;
  desc = [NSString stringWithFormat:
  @"%@, a %d %@ %@, has %d doors, %.1f miles, and %d tires.", name, modelYear, make, model, numberOfDoors
   mileage, [tires count]];
  return desc;
} // description
```

最后，在main函数中，我们将为car设定以上属性，并将它们输出。同时，我们使用了autorelease以及alloc和init方法，这样可以在同一位置执行完所有的内存管理。

```
int main (int argc, const char * argv[])
{
  @autoreleasepool
{
  Car *car = [[[Car alloc] init] autorelease];
  car.name = @"Herbie";
  car.make = @"Honda";
  car.model = @"CRX";
  car.numberOfDoors = 2;
  car.modelYear = 1984;
  car.mileage = 110000;
  int i;
  for (i = 0; i < 4; i++)
  {
    AllWeatherRadial *tire;
    tire = [[AllWeatherRadial alloc] init];
    [car setTire: tire atIndex: i];
```

```
    [tire release];
  }
  Slant6 *engine = [[[Slant6 alloc] init] autorelease];
  car.engine = engine;
  NSLog (@"Car is %@", car);
 }
 return (0);
} // main
```

运行该程序，将得到如下结果。

```
Car is Herbie, a 1984 Honda CRX, has 2 doors, 110000.0 miles, and 4 tires.
```

18.2　KVC 简介

键/值编码中的基本调用是-valueForKey:和-setValue:forKey:方法。你可以向对象发送消息，并传递你想要访问的属性名称的键作为参数。

那么，我们可以这样访问car对象的name属性：

```
NSString *name = [car valueForKey:@"name"];
NSLog (@"%@", name);
```

以上代码将输出Herbie。使用类似的方法，我们还可以获取品牌信息。

```
NSLog (@"make is %@", [car valueForKey:@"make"]);
```

valueForKey:的功能非常强大，它可以找到make属性的值并将其返回。

valueForKey:会首先查找以参数名命名（格式为-key或-isKey）的getter方法。对于以上两个调用，valueForKey:会先寻找-name和-make方法。如果没有这样的getter方法，它将会在对象内寻找名称格式为_key或key的实例变量。如果我们没有使用@synthesize来提供访问方法，valueForKey方法将会寻找名称为_name和name以及_make和make的实例变量。

最后非常重要的一点是，-valueForKey在Objective-C运行时中使用元数据打开对象并进入其中查找需要的信息。在C或C++语言中不能执行这种操作。通过使用KVC，没有相关getter方法也能获取对象值，不需要通过对象指针来直接访问实例变量。

可以对型号年份使用同样的技术：

```
NSLog (@"model year is %@", [car valueForKey: @"modelYear"]);
```

它的输出结果为：model year is 1984。

NSLog中的%@是用来输出对象的，但modelYear是一个int值，而不是对象。这该如何处理呢？对于KVC，Cocoa会自动装箱和开箱标量值。也就是说，当使用setValueForKey时，它自动将标量值（int、float和struct）放入NSNumber或NSValue中；当使用-setValueForKey时，它自动将标量值从这些对象中取出。仅KVC具有这种自动装箱功能，常规方法调用和属性语法不具备该功能。

除了检索值外，还可以使用-setValue:forKey:方法依据名称设置值。

```
[car setValue: @"Harold" forKey: @"name"];
```

这个方法的工作方式和-valueForKey:相同。它首先查找名称的setter方法，例如-setName，然后调用它并传递参数@"Harold"。如果不存在setter方法，它将在类中寻找名为name或_name的实例变量，然后为它赋值。

谨记以下规则

编译器和苹果公司都以下划线开头的形式保存实例变量名称，如果你尝试在其他地方使用下划线，可能会出现严重的错误。你可以不遵守这条规则，但如果不遵循它，有可能会遇到某种风险。

如果你想要设置一个标量值，在调用-setValue:forKey:方法之前需要将它们包装起来，也就是装箱到对象中。

```
[car setValue: [NSNumber numberWithFloat: 25062.4] forKey: @"mileage"];
```

此外，-setValue:forKey:方法会先开箱取出该值，再调用-setMileage:方法或更改mileage实例变量。

18.3 键路径

除了通过键设置值之外，键/值编码还支持指定键路径，就像文件系统路径一样，你可以遵循一系列关系来指定该路径。

为了更深入地了解这项功能，不妨加大引擎的马力。向Engine类添加一个新的实例变量。

```
@interface Engine : NSObject <NSCopying>
{
  int horsepower;
}
@end // Engine
```

请注意我们没有添加任何访问方法或属性。通常需要为有用的对象变量添加访问方法或属性，不过为了展示KVC是如何直接获取对象的，在这里我们不会使用这些方法。

为了让引擎有马力能够启动，我们添加了一个init方法。

```
- (id) init
{
  if (self = [super init])
  {
    horsepower = 145;
  }
  return (self);
} // init
```

另外，我们也在-copyWithZone方法中添加了关于horsepower实例变量的代码，这样在复制对

象时也能获取这个值，并且在-description方法中也添加了相关代码。我们已经很熟悉这个方法了，所以在此就不详细介绍了。

在实现文件中添加以下代码，以确保能够获取并设置值。

```
NSLog (@"horsepower is %@", [engine valueForKey: @"horsepower"]);
[engine setValue: [NSNumber numberWithInt: 150]forKey: @"horsepower"];
NSLog (@"horsepower is %@", [engine valueForKey: @"horsepower"]);
```

运行这段代码将会输出：

```
horsepower is 145
horsepower is 150
```

如何表示这些键路径呢？可以在对象和不同的变量名称之间用圆点分开。通过查询car的engine.horsepower，就能够获取马力值。现在我们试着使用-valueForKeyPath和-setValueForKeyPath方法来访问键路径。将以下消息发送给car对象，而不是发送给engine。

```
[car setValue: [NSNumber numberWithInt: 155]forKeyPath: @"engine.horsepower"];
NSLog (@"horsepower is %@", [car valueForKeyPath: @"engine.horsepower"]);
```

这些键路径的深度是任意的，具体取决于对象图（object graph，可以表示对象之间的关系）的复杂度，可以使用诸如car.interior.airconditioner.fan.velocity这样的键路径。在某种程度上，使用键路径比使用一些列嵌套方法调用更容易访问到对象。

18.4　整体操作

关于KVC非常棒的一点是，如果使用某个键值来访问一个NSArray数组，它实际上会查询相应数组中的每个对象，然后将查询结果打包到另一个数组中并返回给你。这种方法也同样适用于通过键路径访问的位于对象中的数组（是不是想到了以前说过的复合？）。

在KVC中，通常认为对象中的NSArray具有一对多的关系。举个例子，汽车与多个（一般都是4个）轮胎具有联系。因此，我们可以说Car与Tire之间存在一对多的关系。如果键路径中含有一个数组属性，则该键路径的其余部分将被发送给数组中的每个对象。

一对一关系

你现在已经了解了一对多关系，可能还想知道什么是一对一的关系。一般对象的复合都是一对一关系。例如，汽车与其引擎之间就是一对一的关系。

还记得Car类中有一个tires数组吗？每个轮胎都有它自己的空气压力。我们可以在一个调用中获取所有的轮胎压力值。

```
NSArray *pressures = [car valueForKeyPath: @"tires.pressure"];
```

调用以下代码之后

```
NSLog (@"pressures %@", pressures);
```

就会输出如下结果：

```
pressures (
34,
34,
34,
34 )
```

除了告诉我们轮胎状态之外，这里还发生了什么呢？valueForKeyPath:将路径分解并从左向右进行处理。首先，它向car对象请求轮胎信息，然后使用键路径的剩余部分（在本示例中是pressure）向tires对象调用valueForKeyPath:方法。NSArray实现valueForKeyPath:的方法是循环遍历它的内容并向每个对象发送消息。因此NSArray向每个在自身之中的tire对象发送了参数以pressure作为键路径的valueForKeyPath:消息，结果就会将tire对象的pressure变量封装到NSNumber对象中并返回。非常方便！

不幸的是，不能在键路径中索引这些数组，例如使用tires[0].pressure来获取第一个轮胎的压力值。

18.4.1 休息一下

在介绍键/值编码的下一个优点之前，我们将添加一个名为Garage的新类，用于存放各种不同类型的car对象。你可以在名为18.02 Car-Value-Garaging的项目中找到这个新类。下面是Garage类的接口内容。

```
#import <Cocoa/Cocoa.h>
@class Car;
@interface Garage : NSObject
{
 NSString *name;
 NSMutableArray *cars;
}
@property (readwrite, copy) NSString *name;
- (void) addCar: (Car *) car;
- (void) print;
@end // Garage
```

此处没有涉及新内容。我们在一开始就声明了Car类，因为需要知道这个对象类型被用作-addCar:方法的参数。name是一个属性值，而@property语句表示使用Garage类的人可以访问和更改name属性值。并且代码中还有用来输出对象内容的方法。为了实现一个cars对象集合，我们添加了一个可变数组的实例变量。

实现代码的内容同样很简单。

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

```
@synthesize name;
- (void) addCar: (Car *) car
{
  if (cars == nil)
  {
    cars = [[NSMutableArray alloc] init];
  }
  [cars addObject: car];
} // addCar
- (void) dealloc
{
  [name release];
  [cars release];
  [super dealloc];
} // dealloc
- (void) print
{
  NSLog (@"%@:", name);
  for (Car *car in cars)
  {
   NSLog (@" %@", car);
  }
} // print
@end // Car
```

像往常一样，我们包含了Garage.h头文件并使用@synthesize合成了name属性的存取方法。

-addCar:是cars数组惰性初始化（lazy initialization）的一个示例，我们仅在需要时才创建它。-dealloc用于清理name属性和数组，而-print遍历数组并输出各种类型汽车的信息。

与之前的程序版本相比，我们全面修改了Car-Value-Garage项目的main.m文件。这一次，程序构造了一组汽车并将它们放入车库中。

首先，需要使用#import导入将要使用的对象。

```
#import <Foundation/Foundation.h>
#import "Car.h"
#import "Garage.h"
#import "Slant6.h"
#import "Tire.h"
```

接下来，使用一个函数构造汽车的各种属性。我们可以创建一个Car类的类方法，也可以创建多种类型的工厂类，因为Objective-C仍然是一种C语言，所以可以使用函数。再次，我们使用函数是因为组装汽车的函数代码与实际组装汽车的方式比较接近。

```
Car *makeCar (NSString *name, NSString *make, NSString *model, int modelYear, int numberOfDoors, float
  mileage, int horsepower)
{
  Car *car = [[[Car alloc] init] autorelease];
  car.name = name;
  car.make = make;
  car.model = model;
  car.modelYear = modelYear;
  car.numberOfDoors = numberOfDoors;
```

```
    car.mileage = mileage;
    Slant6 *engine = [[[Slant6 alloc] init] autorelease];
    [engine setValue: [NSNumber numberWithInt: horsepower] forKey: @"horsepower"];
    car.engine = engine;
  // Make some tires.
  for (int i = 0; i < 4; i++)
  {
    Tire * tire= [[[Tire alloc] init] autorelease];
    [car setTire: tire atIndex: i];
   }
    return (car);
  } // makeCar
```

现在上面的代码你基本都已熟悉了。按照Cocoa的惯例构造并自动释放了一个新Car对象，因为通过这个函数获得的car对象没有调用new、copy或alloc方法。然后，我们设置了一些属性。请记住，这项技术与KVC不同，我们没有使用setValue:forKey方法。接下来，我们创建了一个engine对象，因为我们没有为它创建存取方法，所以使用KVC设置其马力值。最后，构造一些tire对象并将它们安置在car对象中。最终会返回新的car对象。

以下是新版的main()函数。

```
int main (int argc, const char * argv[])
{
@autoreleasepool
{
 Garage *garage = [[Garage alloc] init];
 garage.name = @"Joe's Garage";
 Car *car; car = makeCar (@"Herbie", @"Honda", @"CRX", 1984, 2, 110000, 58);
 [garage addCar: car];
 car = makeCar (@"Badger", @"Acura", @"Integra", 1987, 5, 217036.7, 130);
 [garage addCar: car];
 car = makeCar (@"Elvis", @"Acura", @"Legend", 1989, 4, 28123.4, 151);
 [garage addCar: car];
 car = makeCar (@"Phoenix", @"Pontiac", @"Firebird", 1969, 2, 85128.3, 345);
 [garage addCar: car];
 car = makeCar (@"Streaker", @"Pontiac", @"Silver Streak", 1950, 2, 39100.0, 36);
 [garage addCar: car];
  car = makeCar (@"Judge", @"Pontiac", @"GTO", 1969, 2, 45132.2, 370);
  [garage addCar: car];
 car = makeCar (@"Paper Car", @"Plymouth", @"Valiant", 1965, 2, 76800, 105);
 [garage addCar: car];
 [garage print];
 [garage release];
 };
return (0);
} // main
```

main()函数进行了一些信息录入，创建了一个garage对象，还创建了一些car对象并保存在garage对象中。最后，main函数输出了garage对象的信息并将其释放。

运行程序后，你应该已经看到结局了。

```
Joe&apos;s Garage:
   Herbie, a 1984 Honda CRX, has 2 doors, 110000.0 miles, 58 hp and 4 tires
   Badger, a 1987 Acura Integra, has 5 doors, 217036.7 miles, 130 hp and 4 tires
   Elvis, a 1989 Acura Legend, has 4 doors, 28123.4 miles, 151 hp and 4 tires
   Phoenix, a 1969 Pontiac Firebird, has 2 doors, 85128.3 miles, 345 hp and 4 tires
   Streaker, a 1950 Pontiac Silver Streak, has 2 doors, 39100.0 miles, 36 hp and 4 tires
   Judge, a 1969 Pontiac GTO, has 2 doors, 45132.2 miles, 370 hp and 4 tires
Paper Car, a 1965 Plymouth Valiant, has 2 doors, 76800.0 miles, 105 hp and 4 tires
```

现在，对键/值编码我们已经有足够的了解了，可以开始了解它的下一个优点了。

18.4.2　快速运算

键路径不仅能引用对象值，还可以引用一些运算符来进行一些运算，例如能获取一组值的平均值或返回这组值中的最小值和最大值。

举个例子，通过以下代码可以计算汽车的数量。

```
NSNumber *count;
count = [garage valueForKeyPath: @"cars.@count"];
NSLog (@"We have %@ cars", count);
```

运行该程序将会输出如下结果：

```
We have 7 cars
```

我们将键路径cars.@count拆开来理解。cars用于获取cars属性，它是取自garage对象的NSArray类型的值。我们知道，它是一个NSMutableArray，但如果我们不打算更改数组的任何内容，可以将其视为NSArray类型。接下来是@count，其中的@符号意味着后面将进行一些运算。对编译器来说，@"blah"是一个字符串，而@interface用于声明类。此处的@count用于通知KVC机制计算键路径左侧值的对象总数。

此外，我们还可以计算某些值的总和，例如，汽车行驶的总英里数。以下代码段

```
NSNumber *sum;
sum = [garage valueForKeyPath: @"cars.@sum.mileage"];
NSLog (@"We have a grand total of %@ miles", sum);
```

运行后将输出

```
We have a grand total of 601320.6 miles
```

该数字让我们在地球与月球之间往返一次还有余。

这项功能是如何做到的？@sum运算符将键路径分成两部分。第一部分可以看成一对多关系的键路径，在本例中代表cars数组。另一部分可以看成任何包含一对多关系的键路径。它被当作用于关系中每个对象的键路径。因此mileage消息被发送给了cars关系中所有的对象，然后将结果

值相加。当然，每个键路径的长度可以是任意的。

如果需要得到平均每辆汽车行驶的距离，可以用其总数除以汽车数量，但还有一种更简单的方法。以下几行代码

```
NSNumber *avgMileage;
avgMileage = [garage valueForKeyPath: @"cars.@avg.mileage"];
NSLog (@"average is %.2f", [avgMileage floatValue]);
```

将输出

```
average is 85902.95
```

非常简单吧？如果没有键/值编码的这个优点，我们必须对这些汽车（假设我们能够从garage对象中获得cars数组的话）编写一个循环，查找每辆汽车的行驶距离，将它们累加，然后再去除以汽车数量——虽然并不是很难，但还是需要一小段代码。

现在我们将键路径cars.@avg.mileage分开。和@sum一样，@avg运算符将键路径分成了两部分。在本例中，@avg之前的部分为cars，是汽车一对多关系的键路径；@avg之后是另一个键路径，它仅表示距离。在后台，KVC能够轻松地进行循环，将值累加，并计算总数，然后再进行除法运算。

还有@min和@max运算符，它们的功能很明显。

```
NSNumber *min, *max;
min = [garage valueForKeyPath: @"cars.@min.mileage"];
max = [garage valueForKeyPath: @"cars.@max.mileage"];
NSLog (@"minimax: %@ / %@", min, max);
```

输出结果为：minimax: 28123.4 / 217036.7。

不要滥用KVC

既然KVC能非常轻松地处理集合类，为什么不用它来处理所有对象，抛弃存取方法和其他代码的编写呢？天下没有免费的午餐，除非你在某个硅谷技术大公司工作。[1]KVC需要解析字符串来计算你需要的答案，因此速度比较慢。此外，编译器还无法对它进行错误检查。你可能想要处理karz.@avg.millage，但编译器不能判断它是否是错误的键路径。因此，当你尝试使用它时，就会出现运行时错误。

有时你使用的变量的值只有几个，例如上面构造的所有汽车。即便我们有100万辆汽车，品牌的种类也会很少。通过使用键路径cars.@distinctUnionOfObjects.make，就可以从集合中只获取各个品牌的名称。

```
NSArray *manufacturers; manufacturers =
[garage valueForKeyPath: @"cars.@distinctUnionOfObjects.make"];
NSLog (@"makers: %@", manufacturers);
```

① 这里指的是谷歌与其他一些公司的免费午餐制度。——译者注

运行上述代码，将得到以下结果：

```
makers: (
    Honda,
    Plymouth,
    Pontiac,
    Acura
)
```

键路径中间的运算符名称为@distinctUnionOfObjects，它看上去很复杂，但由名称就能了解它的功能。它和其他运算符的应用原理相同：获取左侧指定的集合，对该集合中的每个对象使用右侧的键路径，然后将结果转换为一个集合。名称中的union指一组对象的并集，distince用于删除重复内容。还有很多其他运算符也沿用了这种工作方式，这些内容留给你自己去探索。不过，你无法添加自己的运算符，这一点比较遗憾。

18.5　批处理

KVC包含两个调用，可以使用它们为对象进行批量更改。第一个调用是dictionaryWithValuesForKeys:方法，它接收一个字符串数组。该调用获取键的名称，并对每个键使用valueForKey:方法，然后为键字符串和刚刚获取的值构建一个字典。

我们从garage对象中挑选一个car对象，并使用其中一些变量来创建一个字典。

```
car = [[garage valueForKeyPath: @"cars"] lastObject];
NSArray *keys = [NSArray arrayWithObjects: @"make", @"model", @"modelYear", nil];
NSDictionary *carValues = [car dictionaryWithValuesForKeys: keys];
NSLog (@"Car values : %@", carValues);
```

运行以上代码，我们将获取一些相关信息。

```
Car values : {
make = Plymouth;
model = Valiant;
modelYear = 1965;
}
```

我们还可以更改这些值，将Valiant变成新的型号，升级为Chevy Nova。

```
NSDictionary *newValues =
[NSDictionary dictionaryWithObjectsAndKeys:
@"Chevy", @"make",
@"Nova", @"model",
[NSNumber numberWithInt:1964], @"modelYear",
nil];
[car setValuesForKeysWithDictionary: newValues];
NSLog (@"car with new values is %@", car);
```

运行以上这些代码后，我们会发现它确实变成了一辆新车。

```
car with new values is Paper Car, a 1964 Chevy Nova, has 2 doors, 76800.0 miles, and 4 tires.
```

请注意，某些值（品牌、型号和年份）发生了变化，但名称和行驶距离等没有变。

在本程序中，这个工具不是特别有用，不过它还支持在用户界面代码中实现一些不错的功能。例如，通过苹果公司的Aperture程序中的Lift and Stamp工具，可以把对某一张图片的部分修改同样用在其他图片上。可以使用dictionaryWithValuesForKeys方法获取所有变量，并将字典中的内容全部显示在用户界面上。用户可以使用setValuesForKeysWithDictionary方法获取字典内容并对其他图片进行更改。如果你正确地设计了你的用户界面类，也可以对其他对象（比如图片、cars对象或食谱）使用相同的lift and stamp面板。

字典不能包含nil值，但如果出现nil值会怎样呢（例如一辆没有名字的汽车）？回想一下第7章中的内容，我们使用[NSNull null]表示nil值。同样地，当调用dictionaryWithValuesForKeys时，对于没有名称的汽车，@"name"键下将返回[NSNull null]。你也可以为setValuesForKeysWithDictionary提供[NSNull null]，这样汽车就会没有名字了。

18.6 nil仍然可用

对nil值的讨论引出了一个有趣的问题。标量值（例如mileage）中的nil表示什么？0？-1？圆周率？Cocoa无法知道它代表的是什么。你可以尝试以下代码。

```
[car setValue: nil forKey: @"mileage"];
```

不过Cocoa会给出以下警告信息。

```
[<Car 0x105740> setNilValueForKey]: could not set nil as the value for the key mileage.';
```

为了解决该问题，可以重写18.02 Car-Value-Garaging项目中-setNilValueForKey方法的实现，并提供逻辑上有意义的任何值。我们先约定好，nil值表示行驶了零距离，而不是像-1之类的其他值。

```
- (void) setNilValueForKey: (NSString *) key
{
  if ([key isEqualToString: @"mileage"])
 {
  mileage = 0;
 } else {
  [super setNilValueForKey: key];
 }
} // setNilValueForKey
```

请注意，如果得到一个意料之外的键，我们将调用超类方法。这样的话，如果某人试图对键/值编码使用了我们不能理解的键，调用者将会得到警告信息。一般来说，除非你有某些特殊的原因（比如不想执行某个操作），否则应该总是在重写的代码中调用超类的方法。

18.7　处理未定义的键

键/值编码学习过程（你花了有3个小时吗？）的最后一站是处理未定义的键。如果你使用了KVC，并且输入了错误的键，你可能会看到以下消息。

```
'[<Car 0x105740> valueForUndefinedKey:]: this class is not key value coding-compliant for the key
garbanzo.'
```

以上消息的主要含义是，Cocoa不能识别你使用的这个键，因此放弃了操作。

如果仔细分析错误消息，你会注意到，它提到了valueForUndefinedKey:方法。你也许能够猜到，我们可以通过重写该方法来处理未定义的键。也许你还能猜到，如果要更改未知键的值，还可以使用相应的setValue:forUndefinedKey:方法。

如果KVC机制无法找到处理方式，会退回并询问类该如何处理。默认的实现会取消操作，就像你在前面所看到的。但是我们可以更改默认的行为，将Garage转换成一个非常灵活的对象，通过它可以设置和获取任何键。我们首先添加一个可变字典。

```
@interface Garage : NSObject
{
  NSString *name;
  NSMutableArray *cars;
  NSMutableDictionary *stuff;
}
… @end // Garage
```

接下来添加 valueForUndefinedKey方法的实现

```
- (void) setValue: (id) value forUndefinedKey: (NSString *) key
{
  if (stuff == nil)
  {
    stuff = [[NSMutableDictionary alloc] init];
  }
  [stuff setValue: value forKey: key];
} // setValueForUndefinedKey
- (id) valueForUndefinedKey:(NSString *)key
{
  id value = [stuff valueForKey: key];
  return (value);
} // valueForUndefinedKey
```

并在-dealloc方法中释放字典。

现在可以设置garage对象上的任何值：

```
[garage setValue: @"bunny" forKey: @"fluffy"];
[garage setValue: @"greeble" forKey: @"bork"];
[garage setValue: [NSNull null] forKey: @"snorgle"];
[garage setValue: nil forKey: @"gronk"];
```

然后将它们的内容输出：

```
NSLog (@"values are %@ %@ %@ and %@", [garage valueForKey: @"fluffy"], [garage valueForKey: @"bork"],
```

```
[garage valueForKey: @"snorgle"], [garage valueForKey: @"gronk"]);
```

这个NSLog将输出以下结果：

```
values are bunny greeble <null> and (null)
```

请注意<null>与(null)之间的区别。<null>是一种[NSNull null]对象，而(null)是一个真正的nil值。由于字典中没有键为gronk的值，所以此处我们得到了nil值。还要注意，在使用stuff字典时，我们使用了KVC的setValue:forKey:方法。通过这种方法，调用者可以直接传入nil值，我们不必在代码中检查它。而如果为NSDictionary类的setObject:forKey:提供nil值，它将会给出警告信息。此外，如果在字典中对setValue:forKey:方法传入nil值，可能会把对应键的值从字典中删除。

18.8 小结

虽然本章介绍的只是键/值编码的一小部分内容，但应该为你学习KVC的其他内容打下了牢固的基础。本章演示了使用单个键设置和获取值的示例，其中KVC通过查找setter和getter方法来完成你想要的操作。如果KVC无法找到任何方法，将直接进入对象并更改值。

此外还介绍了键路径，它们是由点分割的键，用于在对象的网络中指定路径。也许这些键路径看起来很像访问属性，但实际上它们是两种完全不同的机制。可以将各种运算符嵌入到键路径中，以使KVC实现其他功能。最后介绍了可以进行重写来定制个别行为（corner-case）的方法。

下一章将介绍Xcode的静态分析器。

第 19 章 使用静态分析器 19

在创建应用程序的时候，大部分编译器可以检测可疑的代码并提出警告，指出可能会在运行时出现问题的代码。为了比这种警告做得更好，几年前苹果就在Xcode 3.2版本中添加了静态分析器（static analyzer）。静态分析器是一个不需要运行程序就可以从逻辑上检测代码的工具，它可以寻找会演变成bug的错误。在本章中，我们将要了解如何使用静态分析器来寻找代码中的错误。

19.1 静态工作

静态分析器能做些什么？它又是如何工作的？静态分析器非常了解Objective-C程序想要做什么，并会检查你的程序。它不是简单地浏览一遍源代码，而是在应用程序的代码通道中查找逻辑错误并反馈给你。你可以在构建并运行之前就对它们进行修复。

静态分析器可以认出以下几种错误：

- 安全问题，比如内存泄漏和缓冲区溢出。
- 并发性问题，比如静态条件（也就是依赖时间的两个或多个任务失效）。
- 逻辑问题，包括废代码和不好的编码习惯。

这非常棒！不过除了这些优点，分析器也有以下不足之处：

- 因为需要消耗时间来进行分析，所以会拖慢构建程序的过程。
- 有时会误报错误。
- 改变了你熟悉的工作流程，因为你必须要适应它。

19.1.1 开始分析

虽然有些缺点，不过听起来也没那么坏。我们来试试这个工具吧。要怎么做呢？

其实它真的简单得不能再简单了。首先打开一个项目，我们使用19.01 CarParts Error。点击Product菜单并选择Analyze选项（如图19-1所示），也可以使用command+shift+B快捷键。这样就完成了！你已经在分析了，不过还没有结束，这只是你使用这个工具的第一步。

在构建19.01 CarParts Error程序的时候，我们没有看到它编译出错误，运行正常。因为有静态分析过程，你可能会发现我们的程序编译时间有些久。

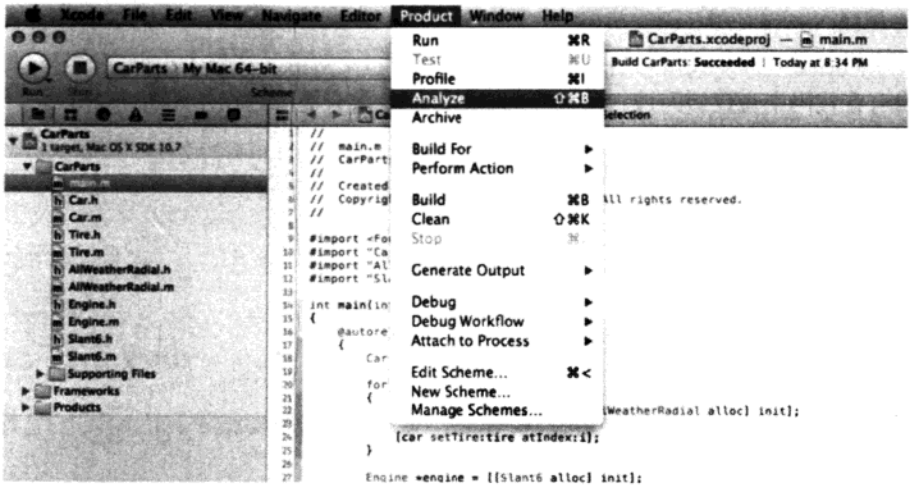


图19-1 选择Product菜单下的Analyze选项

你应该还看到分析器用分支箭头报告了一些内容。静态分析器为我们找到了4个之前不知道的问题。太酷了，我们来看看它们。

问题导航器（如图19-2所示）列出了分析器找到的问题，我们将一个个地检测。

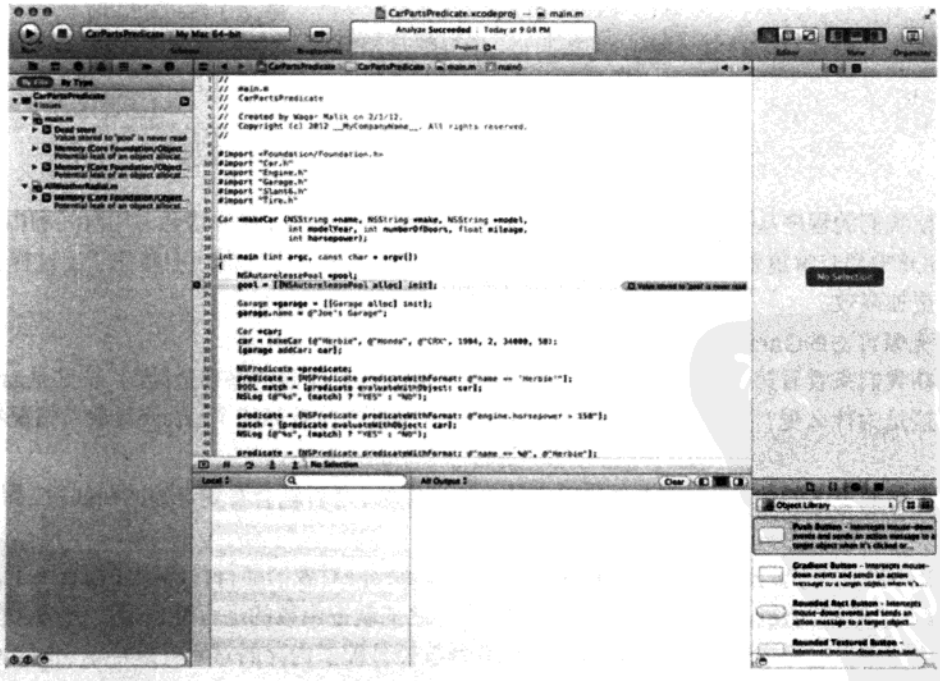


图19-2 分析器找到了四个问题

1. 废代码

第一个问题是“无效变量”。它指的是我们创建了一个对象（在这个示例中的名称是pool），但从来没有在代码中直接访问过，没有向它发送消息也没有更改过它（如图19-3所示）。

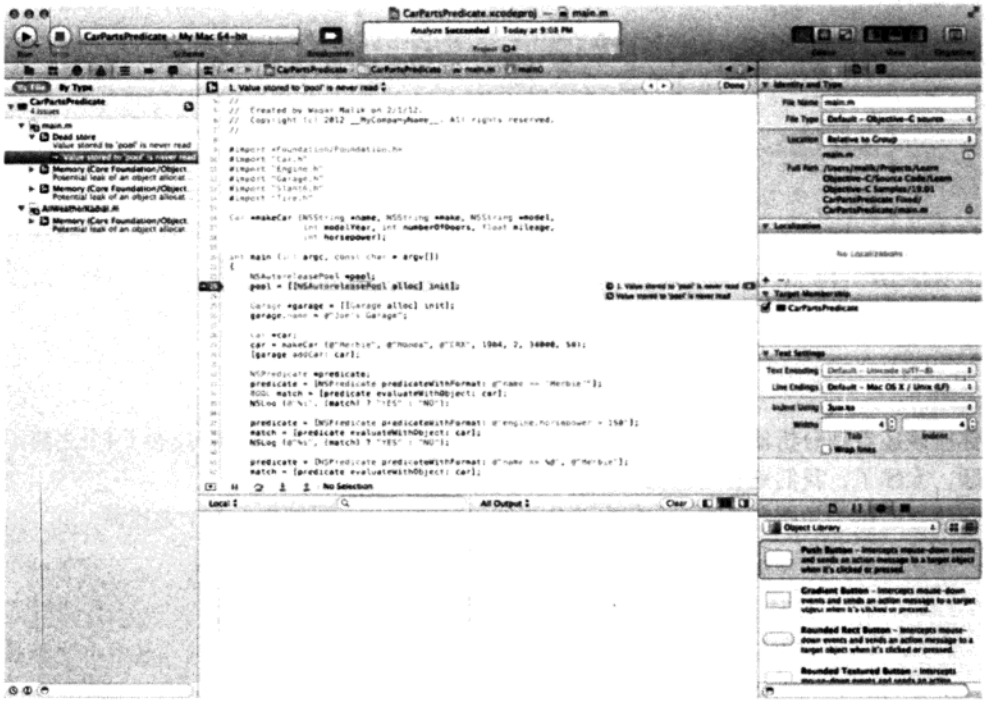


图19-3　分析器报告了代码中的问题

即使我们的程序从技术上说是正常的，但分配和释放内存的过程显然会占用时间和内存，这两点在iOS中都非常重要。感谢分析器，我们可以将这个变量从应用程序中移除了，这样会使应用程序更加高效。

2. 无懈可击的Garage

现在我们来看看第二个问题。它指出“某个对象有内存泄漏的潜在危险”，特别是garage对象。但这是为什么呢？我们在main函数末尾释放pool变量之前就释放了garage对象。情况有些复杂了。

为了得到更多的信息，我们点击代码中的分析器图标，之后就会看到奇妙的曲线段，如图19-4所示。

图19-4展示了代码运行的流程，很明显没有释放garage对象的那行代码。仔细看一下，在第177行有一个单独的return语句，它在我们清理内存之前就把函数的运行终止了。

令人惊讶，这种错误很常见。通常你还没有释放掉分配了内存的对象就过早返回时，便会出现这种情况。

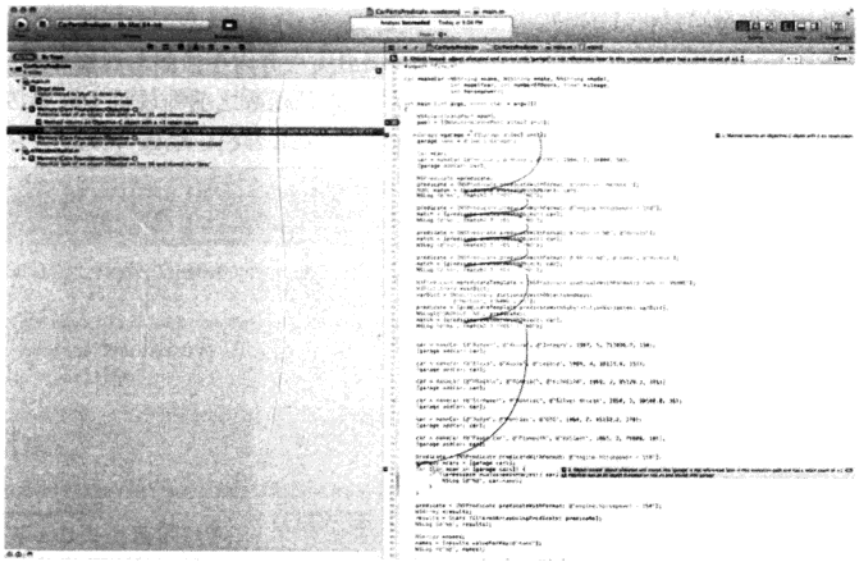


图19-4 分析器显示在代码中的运行路径

3. 释放之前创建的对象

完成了两个，还剩两个。接下来的分析器问题是carsCopy的内存泄漏。我们为变量cars创建了一个可变的字典副本carsCopy，但是没有释放副本（如图19-5所示）。可以通过在main函数结尾处释放carsCopy来修复这个问题。

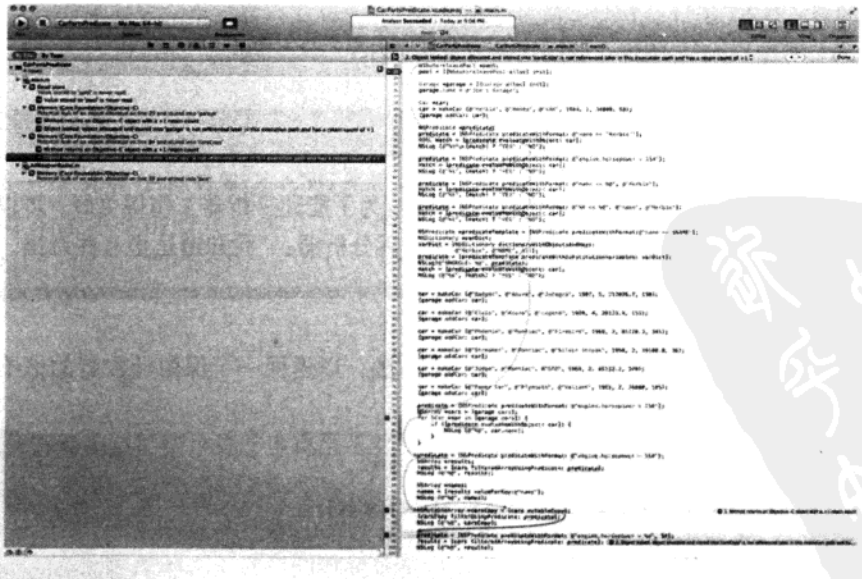


图19-5 创建了carsCopy但没有释放

4. 返回前记得释放

剩下最后一个分析器问题了，即在AllWeatherRadial类中有内存泄漏。观察一下，我们发现description方法中分配了一个字符串desc，但并没有在返回函数之前释放它（如图19-6所示）。在这个示例中，我们可以将返回语句改为return[desc autorelease]，以告诉释放池清理其内存。

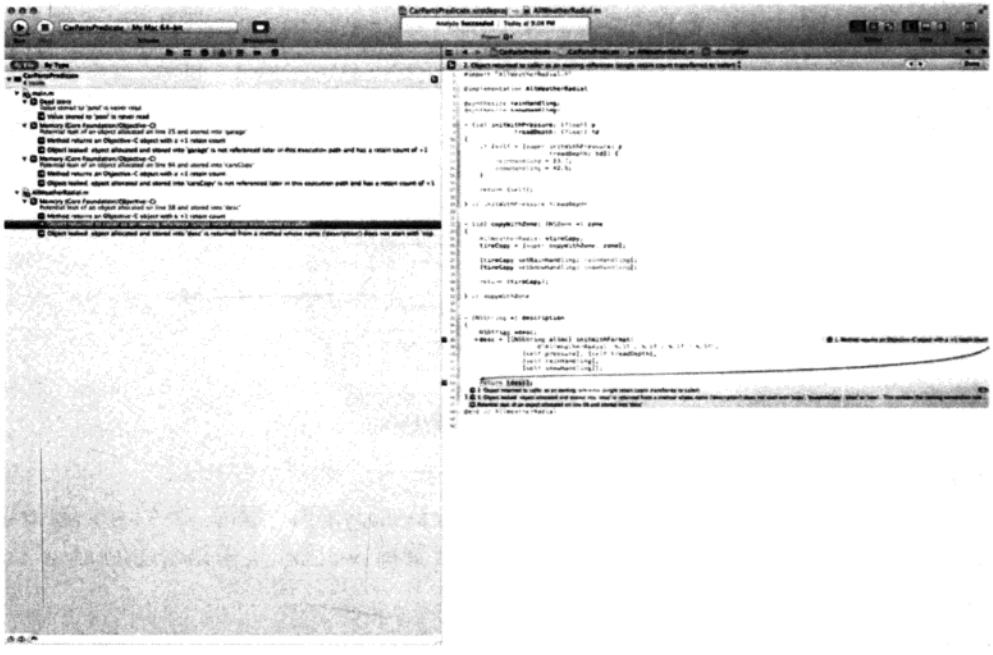


图19-6　我们给desc分配了内存但没有释放它

19.1.2　协助分析器

静态分析器是一个不错的工具，但算不上完美。为了能让分析器工作得更好，可以在你的方法中使用关键字以避免误报。这些关键字能让你告诉分析器："我知道这里有些奇怪，但我保证我知道我在做什么，所以请不用提醒我。"

1. 返回一个保留的对象

你可以使用NS_RETURNS_RETAINED来标记一个方法，以返回一个保留计数器的值不是零的对象。假设你编写了这个方法：

```
- (NSMutableArray *)superDuperArrayCreator
{
  NSMutableArray *myArray = [[NSMutableArray alloc] init];
  // … process the myArray
  return myArray;
}
```

你知道你想要这个方法返回myArray变量，并且计划返回之后再释放这个数组。但是分析器不知道，它只知道从惯例上来说，你必须在给它分配内存的地方清理其内存。

为了使分析器平静下来，你可以为这个方法标记NS_RETURNS_RETAINED。代码如下所示：

```
- (NSMutableArray *)superDuperArrayCreator NS_RETURNS_RETAINED;
```

同理，你可以使用C对象的Core Foundation版本。使用关键字CF_RETURNS_RETAINED就能返回对象了。比如以下代码

```
- (CFMutableArrayRef)superDuperArrayCreator
{
  CFMutableArrayRef myArrayRef = CFArrayCreateMutable(kCFAllocatorNull, 10, NULL);
  // … process the myArray
  return myArrayRef;
}
```

可以写成

```
- (CFMutableArrayRef)superDuperArrayCreator CF_RETURNS_RETAINED;
```

2. 返回一个未保留的对象

你也可以反过来，告诉分析器当你试图返回一个保留对象时作出反应。关键字分别是NS_RETURNS_NOT_RETAINED和CF_RETURNS_NOT_RETAINED。

如果使用这个定义

```
- (NSMutableArray *)superDuperArrayCreator NS_RETURNS_NOT_RETAINED;
```

分析器便会在我们返回一个保留对象时提出问题。

3. 不返回对象

为了确保方法返回void（即不返回值），可以在定义方法的时候使用关键字CLANG_ANALYZER_NORETURN。在这个示例中，如果你试图返回一个值，就会出现一个分析器问题。

这是什么？

最后一个关键字的前缀CLANG是什么？Clang是一个加强了C编译器并在Xcode中为静态分析器提供基础的开源项目。如果想要了解更多信息，可以浏览网址http://clang.llvm.org。

19.1.3 了解更多

静态分析器在我们的小程序中找到了4个不同的问题。本节将讨论静态分析器可以找到的其他问题。

1. 等号错误

在Objective-C程序中一个常见的模式就是在条件语句（比如if和while）中获取一个值并对

它进行判断。

```
if(myValue = [self getValue])
{
  // do something
}
```

有两种方式来理解if语句：

- 为myValue赋一个值并判断它是否为nil。
- myValue等于那个方法的返回值。

因为这里有些意义不明并且分析器无法理解我们的想法，它会将其标记为一个问题。如果我们想要的是第一个意思，可以将其写成if((myValue = [self getValue]))或是if(nil != (myValue = [self getValue]))。

如果我们想要的是第二个意思，那这里确实有错误，需要更改为正确的代码。

```
if(myValue == [self getValue])
```

2. 内存泄漏

我们来看一下这段代码：

```
- (void)myMethod
{
  NSString *string = [[NSString alloc] initWithFormat:@"%d, %d", 1, 2];
  if(nil == string)
  {
    return;
  }

  NSArray *array = [[NSArray alloc] initWithObjects:string, nil];

  if(nil == array)
  {
    return;
  }
  // do some stuff
  // Much later
  [array release];
  [string release];
}
```

这段代码看起来不错。我们适当地分配了内存并在方法结束时释放掉。一切都看起来那么正常，不是吗？但分析器会告诉你残酷的真相。

代码中有一条路径会导致内存泄漏。如果为array数组分配内存失败，方法便不能运行，并会立即返回出来。但是在这时，我们已经分配了string字符串，因为我们根本没有执行后面的release释放代码，所以它还没有被释放。这说明了一个好习惯：无论在什么时候退出了方法，都需要想一想有没有分配了但还没有释放的对象。

为了修复这个问题，我们更改了第二个if语句的代码，如下所示。

```
if(nil == array)
{
  [string release];
  return;
}
```

3. 过度释放

你通常是创建一个对象，使用它，并在方法结束的时候向其发送autorelease消息，对象便会被释放。

```
NSString *myString = [[[NSString alloc] initWithFormat:…] autorelease];

// later that same app

[myString autorelease];
```

在这个示例中，myString起初保留计数器的值是0，之后我们告诉编译器释放它，但这时分析器就会提出意见。为了修复这个问题，我们移除了autorelease消息。

4. @synchronized语句中的空值

假设我们需要修改一个对象，但想确保在修改它的时候没有其他对象会访问它。我们知道可以使用@synchronized (object)，这样修改完之后，它就可以有效访问了。

不过如果对象是nil（空值），静态分析器将会发出警告，并且@synchronized语句将没有任何效果。为了修复这个问题，我们必须确保对象不是nil。

5. 静态分析器的评价

静态分析器会在代码中报告这些以及其他潜在的问题，而且很善于释放语句。你不必在意它提出的所有问题，因为有时它们会让你迷茫，不过花些时间检查一下也不错。把静态分析器当做一位善意但有些烦人的朋友吧，它所说的通常都是对的。

另一方面，不要太依赖静态分析器来查找每一个内存泄漏和废代码。你自己要对自己的代码负责。

19.2 小结

静态分析器在本章的小程序中找到了4个问题。哇！想象一下它在一个大型的复杂项目中查找问题会是怎样一种结果。你并不是必须要在每次构建时运行分析器，不过在工作流程中适当使用它绝对是一个好习惯。

提醒一下：如你在本章所见，分析器能帮助你找到问题，但你仍需要花费一番功夫来找出问题的原因。相信总有一天分析器会精确指出问题所在并替你修复它，但今天还是得靠你自己。

第20章 NSPredicate

编写软件时，经常需要获取一个对象集合，并通过某些已知条件计算该集合的值。你需要保留符合某个条件的对象，删除那些不满足条件的对象，从而提供一些有用处的对象。

在使用软件（比如iPhoto）的过程中，经常会看到这种现象。如果通知iPhoto仅显示评级为三星或三星以上的图片，则指定的条件为“照片的评级必须为三星或三星以上”。所有的照片都会经过该过滤器，满足条件的对象会通过，而其他对象则被过滤掉了。最后，iPhoto将显示出所有优秀图片。

类似地，iTunes也有自己的搜索框。如果搜索条件是歌手Marilyn Manson或Barry Manilow的歌曲，那么所有非摇滚乐和慢板音乐将被隐藏。这样，你就成功地创建了一个奇妙的舞曲混合列表。

Cocoa提供了一个名为NSPredicate的类，它用于指定过滤器的条件。可以创建NSPredicate对象，通过它准确地描述所需的条件，通过谓词筛选每个对象，判断它们是否与条件相匹配。

这种意义上的“谓词”与在英语语法课上学习的“谓语”大不相同。这里的谓语用在数学和计算机科学概念中，表示计算真值和假值的函数。

Cocoa用NSPredicate描述查询的方式，原理类似于在数据库中进行查询。可以在数据库风格的API中使用NSPredicate类，比如Core Data和Spotlight。在此，我们不打算介绍这两种技术（但可以将本章中的很多内容应用到这两种技术中，也可以应用到自己的对象中）。可以将NSPredicate看成另一种间接操作方式。例如，如果需要查询满足条件的机器人，可以使用谓词对象进行检查，而不必使用代码进行显示查询。通过交换谓词对象，可以使用通用代码对数据进行过滤，而不必对相关条件进行硬编码。这也是第3章中提到的开/闭原则的另一个应用。

20.1 创建谓词

首先需要创建NSPredicate对象，才能将它应用于其他对象。可以通过两种基本方式来实现。第一种是创建许多对象，并将它们组合起来。这需要使用大量代码，如果你正在构建通用用户界面来指定查询，采用这种方式比较简单。另一种方式是查询代码中的字符串。对初学者来说，这种方式比较简单。因此，本书将重点介绍查询字符串。通常建议使用面向字符串的API，尤其是在缺少编译器的错误检查和奇怪的运行时错误的时候。

我们仍然使用CarParts示例，本章的示例基于第18章创建的汽车车库示例。你可以在20.01 CarParts Error项目中查找完整的代码。

首先我们来看一看这辆车的情况。

```
Car *car;
car = makeCar (@"Herbie", @"Honda", @"CRX", 1984, 2, 34000, 58);
[garage addCar: car];
```

之前我们已经编写了makeCar函数，它可以创建一辆汽车，并为其加上引擎和一些车胎。在本例中，我们使用的具体汽车信息是：品牌为Herbie，型号为双门1984 Honda CRX，马力引擎为68，已行驶距离为34000英里。

现在创建谓词：

```
NSPredicate *predicate;
predicate = [NSPredicate predicateWithFormat: @"name == 'Herbie'"];
```

我们将以上代码拆开分析。predicate是一个普通的Objective-C对象指针，它将指向NSPredicate对象。我们使用NSPredicate的类方法+predicateWithFormat:来创建一个真实的谓词。将某个字符串赋给谓词，+predicateWithFormat:使用该字符串在后台构建对象数，来计算谓词的值。

predicateWithFormat方法听起来很像stringWithFormat方法，后者由NSString类提供，可以通过它使用printf样式的格式说明符来插入某些内容。稍后你将看到，也可以使用predicateWithFormat方法实现相同的功能。Cocoa采用一致的命名模式，最好遵循它。

这种谓词字符串看上去像是标准的C语言表达式。它的左侧是键路径name，随后是一个等于运算符“==”，右侧是一个用单引号括起来的字符串。如果谓词字符串中的这段文本没有打引号，就会被当做键路径。只有打了引号，它才能被当做字符串的字面量来处理。可以使用单引号也可以双引号（只要前后匹配就可以了）。通常，还是应该使用单引号，否则必须在字符串中对每一个双引号进行转移。

20.2 计算谓词

通过以上步骤就可以得到一个谓词。接下来做什么呢？通过某个对象来计算它。

```
BOOL match = [predicate evaluateWithObject: car];
NSLog (@"%s", (match) ? "YES" : "NO");
```

-evaluateWithObject:通知接收对象（即谓词）根据指定的对象计算自身的值。在本例中，接收对象为car，使用name作为键路径，应用valueForKeyPath:方法获取名称。然后，它将自身的值（即名称）与Herbie相比较。如果名称和Herbie相同，则-evaluateWithObject:返回YES，否则返回NO。此处，NSLog使用三元运算符将数值BOOL转换成人们可读的字符串形式。

以下是另一个谓词：

```
predicate = [NSPredicate predicateWithFormat: @"engine.horsepower > 150"];
match = [predicate evaluateWithObject: car];
```

此谓词字符串左侧是一个键路径。该键路径链接到汽车内部，查找引擎，然后查找引擎的马

力。接下来，它将马力值与150进行比较，看它是否更大。

通过Herbie计算出这些内容后，得到match值为NO，因为小型Herbie的马力值（58）小于150。

通过特定的谓词条件检查单个对象时进展都很顺利，如果需要检查对象集合，情况就会变得更加有趣。假设我们需要查看车库中哪些汽车的功率最大，可以循环测试每个汽车的谓词。

```
NSArray *cars = [garage cars];
for (Car *car in [garage cars])
{
  if ([predicate evaluateWithObject: car])
  {
   NSLog (@"%@", car.name);
 }
}
```

从车库中获取所有汽车，对它们进行循环，通过谓词计算每个汽车的马力。以上代码段将输出较高马力的汽车：

```
Elvis
Phoenix
Judge
```

看懂了吗？没有？在继续介绍以下内容之前，我们先要确保理解了这里涉及的所有语法。仔细查看NSLog中关于汽车名称的调用。它使用了Objective-C 2.0的点语法，这与调用[car name]是等效的。这里没什么陌生的语法。这里的谓词字符串是"engine.horsepower > 150"，engine.horsepower是键路径，它在后台会执行各种强大的功能。

20.3　数组过滤器

懒惰一直是编程人员的缺点，但某种意义上也是优点。如果不必编写for循环和if语句，这有什么不好的？程序中仅有几行代码，当然，要是能不包含代码就更好了。幸好某些类别将谓词过滤方法添加到了Cocoa集合类中。

-filteredArrayUsingPredicate:是NSArray数组中的一种类别方法，它将循环过滤数组内容，根据谓词计算每个对象的值，并将值为YES的对象累计到将被返回的新数组中。

```
NSArray *results;
results = [cars filteredArrayUsingPredicate: predicate];
NSLog (@"%@", results);
```

这些代码将输出以下结果：

```
(
Elvis, a 1989 Acura Legend, has 4 doors, 28123.4 miles, 151 hp and 4 tires,
Phoenix, a 1969 Pontiac Firebird, has 2 doors, 85128.3 miles, 345 hp and 4 tires,
Judge, a 1969 Pontiac GTO, has 2 doors, 45132.2 miles, 370 hp and 4 tires
)
```

以上这些结果同前面的结果不一样。这里是一组汽车的所有信息，在前面的示例中，我们得

到的结果是汽车名称。我们可以使用KVC（键/值编码）提取其中的名称。请记住，将valueForKey:发送给数组时，键将作用于数组中的所有元素。

```
NSArray *names;
names = [results valueForKey:@"name"];
NSLog (@"%@", names);
```

如果我们输出的是汽车名称，将会看到以下结果：

```
(
   Elvis,
   Phoenix,
   Judge
)
```

假设你有一个可变数组，而且需要剔除不属于该数组的所有项目。NSMutableArray具有-filterUsingPredicate方法，它能轻松实现你的目标。

```
NSMutableArray *carsCopy = [cars mutableCopy];
[carsCopy filterUsingPredicate: predicate];
```

如果你输出carCopy，结果将是前面我们看到的3辆汽车的集合。

因为NSMutableArray是NSArray的子类，所以你也可以对NSMutableArray数组使用-filteredArrayUsingPredicate:方法来构建新的不可变数组。NSSets中也有类似的调用方法。

我们在讨论KVC时提到过，使用谓词确实很便捷，但它的运行速度不会比你自己编写全部代码快。因为它无法避免在所有汽车之间使用循环和对每辆汽车进行某些操作。一般来说，这种循环并不会对OS X上应用的性能产生很大的影响，因为当今的计算机运行速度非常快。尽量编写最便捷的代码。如果遇到了速度问题，可以使用苹果公司的工具（比如Instruments）来测试程序性能，不过iOS程序员应该随时密切关注程序的性能。

20.4 格式说明符

资深编程人员都知道，硬编码并非好办法。如果首先想知道哪些汽车的马力高于200，稍后又需要知道哪些汽车的马力高于50，该怎么办？可以使用谓词字符串，例如"engine.horsepower > 200"和"engine.horsepower > 50"，但我们必须重新编译程序，并会再次遇到第3章中的麻烦问题。

可以通过两种方式将不同的内容放入谓词格式字符串中：格式说明符和变量名。首先介绍格式说明符。可以在你熟知的%d和%f格式说明符中使用数字形式的值。

```
predicate = [NSPredicate predicateWithFormat: @"engine.horsepower > %d", 50];
```

当然，我们一般不直接在代码中使用值50，可以通过用户界面或某些扩展机制来接收某个值。除了使用printf说明符，也可以使用%@插入字符串值，而%@会被当做一个有引号的字符串。

```
predicate = [NSPredicate predicateWithFormat: @"name == %@", @"Herbie"];
```

请注意，这里的格式字符串中%@并没有打单引号。如果你为%@打了引号，例如"name == '%@' ",

字符%和@就会被当做谓语字符串中的普通字符，失去格式说明符的作用。

NSPredicate字符串中也可以使用%K来指定键路径。该谓词和其他谓词一样，使用name == 'Herbie'作为条件。

```
Predicate =
[NSPredicate predicateWithFormat: @"%K == %@", @"name", @"Herbie"];
```

为了构造灵活的谓词，一种方式是使用格式说明符，另一种方式是将变量名放入字符串中，类似于环境变量。

```
NSPredicate *PredicateTemplate =
[NSPredicate predicateWithFormat:@"name == $NAME"];
```

现在，我们有一个含有变量的谓词。接下来可以使用predicateWithSubstitutionVariables调用来构造新的专用谓词。创建一个键/值对字典，其中键是变量名（不包含美元符号$），值是想要插入谓词的内容，代码如下所示。

```
NSDictionary *varDict;
varDict = [NSDictionary dictionaryWithObjectsAndKeys: @"Herbie", @"NAME", nil];
```

这里使用字符串"Herbie"作为键"NAME"的值。因此构造以下形式的新谓词。

```
predicate = [predicateTemplate predicateWithSubstitutionVariables: varDict];
```

该谓词的工作方式和你之前所见过的其他谓词完全相同。

你也可以使用其他对象作为变量的值，例如NSNumber。以下谓词用于过滤引擎的功率。

```
predicateTemplate =
 [NSPredicate predicateWithFormat: @"engine.horsepower > $POWER"];
varDict = [NSDictionary dictionaryWithObjectsAndKeys:
  [NSNumber numberWithInt: 150], @"POWER", nil];
predicate = [predicateTemplate predicateWithSubstitutionVariables: varDict];
```

以上代码创建了一个谓词，它的条件是引擎功率要大于150。

除了使用NSNumber和NSString之外，也可以使用[NSNull null]来设置nil值，甚至可以使用数组，这些内容将在本章稍后的部分中介绍。请注意，不能使用“$变量名”作为键路径，它只能表示值。使用谓词格式字符串时，如果想在程序中通过代码改变键路径，需要使用%K格式说明符。

谓词机制不进行静类型检查。你也许会在要输入数字的地方不小心插入字符串，这样就会出现运行时错误信息，或者其他不可预知的行为。

20.5　运算符

NSPredicate的格式字符串包含大量不同的运算符。这里我们将介绍大多数运算符，并给出每个运算符的示例。其余运算符可以通过苹果公司的在线文档进行查询。

20.5.1　比较和逻辑运算符

谓词字符串语法支持C语言中一些常用的运算符，例如等号运算符“==”和“=”等。

而不等号运算符具有多种形式，如表20-1所示。

表20-1 不等号运算符

运 算 符	比较作用
>	大于某数
>= 和 =>	大于或等于某数
<	小于某数
<= 和 =<	小于或等于某数
!= 和 <>	不等于某数

此外，谓词字符串语法还支持括号表达式（真的！）和AND、OR和NOT逻辑运算符，以及用C语言样式表示具有相同功能的“&&”、“||”和“!”符号。

以下是一个示例。你可以将功率最大和最小的汽车过滤掉，留下中等功率的汽车。

```
predicate = [NSPredicate predicateWithFormat:
        @"(engine.horsepower > 50) AND (engine.horsepower < 200)"];
results = [cars filteredArrayUsingPredicate: predicate];
NSLog (@"oop %@", results);
```

如果将以上代码应用到这些汽车中，将得到以下结果。

```
Herbie, a 1984 Honda CRX, has 2 doors, 34000.0 miles, 58 hp and 4 tires,
Badger, a 1987 Acura Integra, has 5 doors, 217036.7 miles, 130 hp and 4 tires,
Elvis, a 1989 Acura Legend, has 4 doors, 28123.4 miles, 151 hp and 4 tires,
Paper Car, a 1965 Plymouth Valiant, has 2 doors, 76800.0 miles, 105 hp and 4 tires
```

谓词字符串中的运算符不区分大小写。你可以随意使用AnD、ANd和AND。这里我们将统一使用大写字母，但在实际代码中可以不区分大小写。

不等号既适用于数字值又适用于字符串值。如果需要按字母表顺序从头查看所有汽车，可以使用以下谓词。

```
predicate = [NSPredicate predicateWithFormat: @"name <'Newton'"];
results = [cars filteredArrayUsingPredicate: predicate];
NSLog (@"%@", [results valueForKey: @"name"]);
```

将会输出以下结果：

```
(
   Herbie,
   Badger,
   Elvis,
   Judge
)
```

20.5.2 数组运算符

谓词字符串"(engine.horsepower > 50) OR (engine.horsepower < 200)"是一种十分常见的模

式。该谓词字符串用于查找介于50到200之间的马力值。如果我们能使用某个运算符来查找介于这两个值之间的数值，那就太好了！事实上，我们可以使用以下代码来实现该功能。

```
predicate = [NSPredicate predicateWithFormat:
      @"engine.horsepower BETWEEN { 50, 200 }"];
```

花括号表示数组，BETWEEN将数组中第一个元素看成数组的下限，第二个元素看成数组的上限。

可以使用%@格式说明符向你自己的NSArray数组中插入对象。

```
NSArray *betweens = [NSArray arrayWithObjects:
  [NSNumber numberWithInt: 50],
  [NSNumber numberWithInt: 200], nil];
predicate = [NSPredicate predicateWithFormat:
  @"engine.horsepower BETWEEN %@", betweens];
```

也可以使用变量：

```
predicateTemplate =
  [NSPredicate predicateWithFormat: @"engine.horsepower BETWEEN $POWERS"];
varDict =
  [NSDictionary dictionaryWithObjectsAndKeys: betweens, @"POWERS", nil];
predicate = [predicateTemplate predicateWithSubstitutionVariables: varDict];
```

数组并不仅可以用来指定某个区间的端点值，你还可以使用IN运算符来查找数组中是否含有某个特定值，具有SQL编程经验的编程人员应该对以下代码非常熟悉。

```
predicate = [NSPredicate predicateWithFormat:
 @"name IN {'Herbie','Snugs','Badger','Flap'}"];
```

名称为Herbie和Badger的汽车将会在过滤中存留下来。

```
results = [cars filteredArrayUsingPredicate: predicate];
NSLog (@"%@", [results valueForKey: @"name"]);
```

已经看到结果了，只有以下两个对象被返回：

```
(
  Herbie,
  Badger
)
```

20.6　有 SELF 就足够了

某些时候，可能需要将谓词应用于简单的值（例如纯文本的字符串），而非那些可以通过键路径进行操作的复杂对象。假设我们有一个汽车名称的数组，并且需要应用与之前相同的过滤器。从NSString对象中查询name时，将无法起作用，那么我们用什么来代替name呢？

用SELF就可以解决了！SELF表示的是响应谓词计算的对象。事实上我们可以将谓词中所有的键路径表示成对应的SELF形式。此谓词和前面的谓词完全相同，代码如下所示。

```
predicate = [NSPredicate predicateWithFormat:
 @"SELF.name IN {'Herbie','Snugs','Badger','Flap'}"];
```

现在，再回到那个字符串数组。如果某个字符串也在这个名称数组中，该怎么办呢？我们来看一下。

首先，需要从某处获取仅含有名称的数组。因为我们已经很熟悉CarParts中的各种对象了，所以对数组使用KVC技术的valueForKey:方法就可以获取到了。

```
names = [cars valueForKey: @"name"];
```

以上字符串数组包含我们拥有的所有汽车名称。接下来构造一个谓词

```
predicate = [NSPredicate predicateWithFormat:
 @"SELF IN {'Herbie','Snugs','Badger', 'Flap'}"];
```

并计算该谓词的值。

```
results = [names filteredArrayUsingPredicate: predicate];
```

如果现在查看结果，将会看到和前面示例相同的两个名称：Herbie和Badger。

这里提一个问题，以下代码将输出什么结果呢？

```
NSArray *names1 = [NSArray arrayWithObjects:
 @"Herbie", @"Badger", @"Judge", @"Elvis", nil];
NSArray *names2 = [NSArray arrayWithObjects:
 @"Judge", @"Paper Car", @"Badger", @"Phoenix", nil];

predicate = [NSPredicate predicateWithFormat: @"SELF IN %@", names1];
results = [names2 filteredArrayUsingPredicate: predicate];
NSLog (@"%@", results);
```

答案如下所示：

```
(
  Judge,
  Badger
)
```

对于取两个数组交集的运算而言，这是一种很巧妙的方式。但它是如何实现的呢？谓词包含了第一个数组的内容，因此看起来和下面的形式类似。

```
SELF IN {"Herbie", "Badger", "Judge", "Elvis"}
```

现在，使用该谓词过滤第二个名称数组。在name2中如果具有同时存在两个数组中的字符串，那么SELF IN字句会确定它是符合条件的，因此它就会保留在结果数组中。如果对象只存在于第二个数组中，那么它不会与谓词中的任何字符串匹配，所以该对象将被过滤掉。而只存在于第一个数组中的字符串因为要用来进行比较，所以将一直保留在原来的位置，不会出现在结果数组中。

20.7　字符串运算符

前面使用字符串时介绍过关系运算符。此外，还有以下一些针对字符串的关系运算符，如表20-2所示。

表20-2　针对字符串的运算符

运　算　符	意　义
BEGINSWITH	检查某个字符串是否以另一个字符串开头
ENDSWITH	检查某个字符串是否以另一个字符串结尾
CONTAINS	检查某个字符串是否在另一个字符串内部

使用关系运算符可以执行一些有用的操作，例如使用"name BEGINSWITH 'Bad'"匹配Badger，使用"name ENDSWITH 'vis'"匹配Elvis，以及使用"name CONTAINS udg"匹配Judge。

如果你编写了某个类似于"name ENDSWITH 'HERB'"的谓词字符串，会出现什么情况呢？它不会与Herbie或其他字符串相匹配，因为这些匹配是区分大小写的。同样，"name BEGINSWITH 'Hérb'"也不会与之相匹配，因为其中的e含有重音符。为了减少名称匹配规则，可以为这些运算符添加[c]、[d]或[cd]修饰符。其中，c表示“不区分大小写（case insensitive）”，d表示“不区分发音符号（diacritic insensitive，即忽略重音符）”，[cd]表示“既不区分大小写，也不分区发音符号”。通常，除非你拥有需要区分大小写或重音符号的特殊原因，否则请尽量使用[cd]修饰符。你无法预知用户什么时候会按下大写锁定键致使向应用程序输入的文字全变成了大写。

该谓词字符串会将Herbie与"name BEGINSWITH[cd]'HERB'"相匹配。

20.8　LIKE运算符

某些时候，将一个字符串的开头或结尾（也可能是中间）与另一个字符串进行匹配的功能还不够。对于这种情况，谓词格式字符串还提供了Like运算符。在该运算符中，问号表示与一个字符匹配，星号表示与任意个字符匹配。SQL和Unix shell编程人员应该认识这种操作（有时称为“通配符”）。

谓词字符串"name LIKE '*er*'"将会与任何含有er的名称相匹配。这等效于CONTAINS。

谓词字符串"name LIKE'???er*'"将会与Paper Car相匹配，因为其中的er前面有3个字符，er后面有一些字符。但它与Badger不匹配，因为Badger的er前面有4个字符。

另外，LIKE也接收[cd]修饰符，用于忽略对大小写和发音符号的区分。

如果你热衷于正则表达式，可以使用MATCHES运算符。赋给它一个正则表达式，谓语将会计算出它的值。

表达自己

正则表达式功能非常强大，它是一种指定字符串匹配逻辑的非常紧凑的方式。有时候，正则表达式的形式会变得复杂而费解，已经有大量书籍讨论了这一主题。NSPredicate正则表达式使用ICU（International Components for Unicode）语法，你可以借助因特网搜索引擎了解相关内容。

虽然正则表达式的功能强大，但计算开销非常大。如果在谓词中有某些简单的运算符，例如基本字符串运算符和比较运算符，那么在使用MATCHES之前可以先执行简单的运算，这样将会提高程序的运算速度。

20.9 结语

学完谓词之后本书的学习就结束了。我们已经介绍了很多方面，从间接和OOP的基础知识，一直到复杂的工具（例如键/值编码和使用NSPredicate类过滤对象）。祝贺你坚持读完了整本书！现在，你已经正式准备好步入iOS和OS X编程生涯的下一个阶段了。感谢你加入我们。

附 录

从其他语言转向Objective-C

很多编程人员都是从其他语言转向Objective-C和Cocoa的，由于Objective-C的操作同其他流行语言有很大差异，因此大家在学习过程中会遇到很多困难。初次接触Objective-C的编程人员经常会抱怨它不是一个好语言，因为和他们喜欢的语言相比，它缺少了某些有用的特性。暂不考虑Objective-C的一些特性，摆在我们面前的事实是：在某些方面，Objective-C所表现的强大功能简直不可思议。

对于初次接触Objective-C和Cocoa的编程人员，以及那些在其他语言和平台上有丰富经验的编程人员来说，我们的建议是：先撇开原有的观念，暂时接受Objective-C、Cocoa和Xcode。通过阅读一些图书和教程，获得一些Objective-C经验后，你就会了解哪些技术和方法来自其他语言但适用于Cocoa和Objective-C，而哪些并非如此。任何语言都不可能适用于所有环境，任何工具也不可能适用于所有的工作。最好的方法是，深入了解对象以判断某个语言和工具库是否能满足你的需要，同时权衡相关利弊。

在本附录中，我们将提供相关信息来帮助你从其他流行语言轻松过渡到Objective-C。

A.1 从C转向Objective-C

Objective-C实质上就是在简单的老式C语言基础上添加了一些面向对象的特性。本书用较大篇幅描述了这些附加特性，因此这里不再赘述。不过有一两个有趣的主题还是值得讨论的。

你应该还记得，Objective-C编程人员可以使用与C语言相关的所有工具，比如标准C库函数。可以使用`malloc()`和`free()`函数处理动态内存管理问题，或者使用`fopen()`和`fgets()`函数处理文件。

有时候，有些人会在网上问："调用基于回调工作方式的ANSI C库函数时，我该如何让它调用某个方法呢？"当你使用苹果公司的底层框架（例如Core Foundation和Core Graphics）时，也会遇到这样的问题。

最简短的回答是："没有办法做到。"回调仅适用于那些通过库签名的C函数。实现Objective-C方法的函数必须含有`self`参数和`selector`参数，否则就不可能与所需的签名相匹配。

多数面向回调的库都需要你提供某些用户数据或指针。你可以将用户数据或指针看成是掩饰，隐藏其后。注册回调时，需要为库提供某些有意义的指针，库调用你的回调时将返回对应的指针。

所有的Objective-C对象都是动态分配地址的，因此使用对象地址作为环境指针是安全的。不必担心下面的某些栈分配对象会消失。在回调内部，可以将环境指针强制转换成你的对象类型，然后向它发送消息。

例如，你正在使用假想的C API XML解析库，解析XML文件时需要向TreeWalker类提供数据。首先构造新TreeWalker类。

```
TreeWalker *walker = [[TreeWalker alloc] init];
```

然后，构造XML解析器：

```
XMLParser parser;
parser = XMLParserLoadFile ("/tmp/badgers.xml");
```

接下来，为解析器设置回调函数（C函数），并使用TreeWalker对象作为相关内容。

```
XMLSetParserCallback (parser, elementCallback, walker);
```

然后，当XML文件被解析时，将调用该回调函数：

```
void elementCallback (XMLParser *parser, XMLElement *element, userData *context)
{
  TreeWalker *walker = (TreeWalker *) context;
  [walker handleNewElement: element inParser: parser];
} // someElementCallback
```

可以看到，上下文指针被强制转换成了对象指针，并且向该对象发送了一些消息。

A.2　从C++转向Objective-C

C++具备很多Objective-C所没有的特性：多重继承、命名空间、运算符重载、模板、类变量、抽象类、标准模板库（STL）等。如果你还怀念这些特性，Objective-C为你提供了可以代替或模拟这些功能的特性和技术。

例如，可以使用类别和协议作为一种多重继承形式，或用于实现抽象基类。多重继承的一个常见功能是提供接口，以便其他代码在你的对象中调用特定的方法。类别和协议是完成此项工作的不二之选。你可以使用协议提供纯抽象基类。

如果你使用多重继承引入附加的实例变量（C++中称为成员变量），类别和协议将无法帮助你。为此，可以使用复合在另一个对象中包含某个对象，然后使用存根方法将消息重定向到第二个对象（这种技术在Java中很常见）。也可以通过重写forwardInvocation:方法模拟多重继承。如果某个消息被接收，但该对象不知如何处理，将调用forwardInvocation:方法。通过检查NSInvocation对象，可以确定是否应将它转发到“多重继承”对象，并且在必要时发送它。通过检查这种技术，可以避免编写很多存根方法。但是，与真实的多重继承相比，forwardInvocation:方法的运行速度要慢得多，同时构建该方法的过程也很复杂。

通过其他约定还能替代更多C++特性，例如存根方法可以为那些不能被实例化的抽象类调用abort方法。但某些特性不宜进行替换，例如使用名称前缀代替命名空间。

A.3　C++虚拟表与Objective-C动态分配

C++与Objective-C的最大区别在于调度方法（在C++中称为*成员函数*）的机制不同。C++基于虚拟表（vtable）机制确定虚函数调用什么代码。

你可以认为每个C++对象都有一个指针指向函数指针数组。编译器知道代码需要调用某个虚函数后，将从虚拟表的开头计算偏移位置，并发送机器代码使指针指向该偏移位置，然后执行该位置的代码段程序。该过程需要编译器在编译时知道调用成员函数的对象类型，这样才能正确计算虚拟表的偏移位置。这种分配方式非常快，只需要几个指针操作和一个获取函数指针的读操作。

第3章中已经详细描述过，Objecive-C使用运行时函数进入各种类结构中查找相应的代码以供调用。这种技术比C++的要慢得多。

Objective-C以牺牲一定的速度和安全为代价，增加了灵活性和便捷性，这是一种典型的权衡利弊的做法。使用C++模式，成员函数的调度速度要更快。此外，因为编译器和链接器可以确保使用的对象能处理对应的方法，所以C++模式非常安全。但是，C++方法缺乏一定的灵活性，因为你无法真正改变对象的种类，必须使用继承才能使不同对象类响应相同的消息。

C++编译器不再保留很多类信息，例如类的继承链、类的成员等。通常，C++在运行时处理对象的能力是有限的，最多只能进行动态强制转换操作。通过该操作，可以了解某个对象是否是另一个对象的指定子类。

在运行时，不能改变C++的继承层次结构。编译并链接程序后，程序几乎就固定了。C++库的动态加载也是一个常见的问题，这在某种程度上是由C++*名称重整*（name mangling，它使用自身带有的原始Unix链接器执行安全型的链接方式）的复杂性所导致的。

在Objective-C中，对象只需要方法就可以实现自身的可调用性。这样，任意对象都可以成为其他对象的数据源和委托。缺少多重继承可能会带来诸多不便，但是向任何对象发送任何消息时不必再考虑它的继承血统，这在很大程度上简化了我们的工作。

显然，与C++相比，向任何对象发送任何消息的功能降低了Objective-C的安全性。如果被发送消息的对象不能处理该消息，你将会得到运行时错误。Cocoa中不支持类型安全容器，任何对象都可以放入容器中。

Objective-C携带了许多关于类的元数据，因此，可以通过反射判断某个对象是否响应某个消息。含有数据源或委托的对象经常采用这种方法。但是首先要检查委托是否响应某个消息，这样可以避免出现运行时错误。此外也可以通过类别向其他类中添加方法。

通过元数据，可以轻松操作程序中用到的类。可以确定实例变量、实例变量在对象中的布局，以及类定义的方法。甚至删除可执行对象的调试信息也不会删除Objective-C的元数据。如果存在某些高度机密的算法，可能更需要使用C++来实现，或者至少使用意义不明显的名称，而不要使用类似于`SerialNumberVerifier`的类或方法名。

在Objective-C中，可以直接向`nil`（零）对象发送消息，而不必检查发送消息的对象是否为

NULL。向nil对象发送的消息代表操作指令。发送给nil对象的消息返回值取决于方法的返回类型。如果方法返回某个指针类型（例如对象指针），则返回值是nil，表示可以安全地将消息链接到某个nil对象——nil仅起着传递作用。如果方法返回一个与指针相当或更小的int，则返回值是零。如果返回某个浮点数或结构，你将会得到某个未定义的结果。因此可以使用nil对象模式避免测试对象指针是否为NULL。另一方面，这种技术会掩盖某些错误，并导致一些难以追踪的bug。

在Objective-C中，所有对象都是动态分配的，不存在基于栈的对象，不存在临时对象的自动创建和销毁，也不存在类型间的自动类型转换。因此，Objective-C对象比基于栈的C++对象更复杂。正因为此，一些更高级的实体（例如NSPoint和NSRange）开始成为取代对象的结构。

最后，Objective-C是一种非常松散的语言。C++中含有公共、保护、私有成员变量和成员函数。Objective-C只提供一些基本机制来支持受保护的实例变量（这些机制都很容易避开），但并不保护成员函数。任何人只要知道方法的名称就可以向对象发送消息。利用Objective-C的反射特性，可以了解某个给定对象支持的所有方法。即使某些方法不包含在头文件中，也能调用它们，没有可靠的方法来计算是哪个对象调用了该方法，因为发送的消息还可能来自C函数（该内容前面已经讨论过）。

你已经看到，在重写子类时，不必重新声明方法。对此，有两种不同的观点。一种认为，重新声明向读者提供了相关信息，便于读者获取超类的变化情况。另一种认为，这些仅是实现细节，用户不必关注。同时重写某个新方法时，也不必重新编译所有从属类（dependent class）。

Objective-C中不存在类变量。可以使用文件范围内的全局变量来模拟类变量，并为它们提供访问器。类声明示例可能与以下代码相似（其中包含实例变量声明和方法声明）。

```
@interface Blarg : NSObject
{
}
+ (int) classVar;
+ (void) setClassVar: (int) cv;
@end // Blarg
```

实现示例可能类似于以下代码：

```
#import "Blarg.h"
static int g_cvar;
@implementation Blarg
+ (int) classVar
{
  return (g_cvar);
} // classVar
+ (void) setClassVar: (int) cv
{
  g_cvar = cv;
} // setClassVar
@end // Blarg
```

Cocoa对象的层次结构具有共同的祖先类NSObject。创建新类时，几乎总是会创建NSObject的子类或现有的Cocoa类。C++对象的层次结构看上去就像几棵根系不同的树。

A.4　Objective-C++

还有一种两全其美的方式。Xcode自带的GCC编译器支持一种名为Objective-C++的混合语言。通过该编译器，除了少量限制外，可以自由混合C++和Objective-C代码。在必要时，可以使用安全类型和低级性能；在某些场合，也可以使用Objective-C的动态特性和Cocoa工具包。

常见的开发场景是将应用程序所有的核心逻辑都放入可移植C++库中（如果你正在创建跨平台应用程序），并在平台的本地工具包中编写用户界面。Objective-C++为这种开发风格带来了新体验。用户可以获得C++的性能和安全性，也可以获得由本地工具包创建的应用程序，同时这些工具箱能无缝地集成到平台中。

为了让编译器将你的代码识别为Objective-C++，可以在源文件中使用.mm文件扩展名。另外，也可以使用.M扩展名，但Mac的HFS+是不区分大小写的，而且会保留原来的大小写形式。因此，最好避免任何种类的大小写依赖性。

就像物质与反物质一样，我们无法混合Objective-C和C++对象的层次结构，因此不能让一个C++类继承NSView，同样，也不能让一个Objective-C类继承std::string。

可以将指向Objective-C对象的指针放入C++对象中。由于所有Objective-C对象都是动态分配的，所以不能将完整的对象嵌入某个类中或在栈上进行声明。首先需要在C++构造函数（或者任何合适的地方）中分配和初始化所有Objective-C对象，然后在析构函数（或者其他地方）中释放它们。以下是有效的类声明。

```
class ChessPiece
{
  ChessPiece::PieceType type;
  int row, column;
  NSImage *pieceImage;
};
```

你可以将C++对象嵌入Objective-C对象中。

```
@interface SWChessBoard : NSView
{
  ChessPiece *piece[32];
}
@end // SWChessBoard
```

嵌入到Objective-C对象中的C++对象之间不只是指针关系，当分配Objective-C对象时，C++对象还会调用它们的构造函数。同样，当释放Objective-C对象时，C++对象会调用它们的析构函数。

异常

NS Exceptions和C++异常都是公用的，你可以自由地抛出和捕获代码块。一旦有异常展开，C++析构函数和@finally代码块就会被调用。同样，catch(...)和@catch(...)也可捕获和重抛异常。

A.5　从Java转向Objective-C

和C++一样，Java含有很多Objective-C所不具备的特性和不同的实现方法。例如，传统的Objective-C没有垃圾回收器，却含有保留/释放方法和自动释放池。必要时，也可以在Objective-C程序中进行垃圾回收。

Java接口与Objective-C正式协议类似，因为它们都需要实现一组方法。Java具有抽象类，而Objective-C没有。Java具有类变量，但在Objective-C中，可以使用文件范围内的全局变量并为它们提供对应的访问器，具体内容请参考A.2部分。Objective-C的公共和私有方法的形式比较松散。我们已经说过，在Objective-C中，对象支持的任何方法都可以被调用，即使它们没有以任何外部形式出现（例如头文件中）。Java允许声明final类，以防止子类的内容被更改。而Objective-C则与此相反，允许在运行时向任何类添加方法。

通常，Objective-C中类的实现方式可以分成两个文件：头文件和自身的实现文件，但并不是一定要这样划分（例如某些小的私有类），这在本书中的某些代码中已经有所反映。头文件（带有.h扩展名）保留类的公开信息，比如调用此类的代码将使用任何新的枚举、类型、结构以及代码。其他代码段使用预处理器（使用#import）导入该文件。Java中缺少C语言的预处理器。C预处理器是一种文本替换工具，它能在C、Objective-C和C++源代码进入编译器之前，先对它们进行自动处理。以#开头的指令表示一个预处理器命令。C预处理器实际上并不知道C语言家族的具体机制，它只是完成一些看不见的文本替换工作。预处理器是一个功能非常强大但又危险的工具。很多编程人员都认为Java中缺少预处理器正是它的特色所在。

在Java中，几乎所有错误都是通过异常来处理的。而在Objective-C中，错误处理的方式取决于所使用的API。Unix API通常会返回值-1和一个全局错误编号（errno），以设置某个特定的错误。Cocoa API通常只在编程人员出现错误或无法清除时才抛出异常。Objective-C语言提供的异常处理特性与Java及C++类似，采用@try、@catch和@finally结构。

在Objective-C中，空（零）对象使用nil表示。可以向nil对象发送消息，而不必担心出现NullPointerException异常。向nil对象发送的消息代表停止操作指令，因此，不必检查发送的消息是否为NULL。向nil发送消息的具体内容请参考“从C++转向Objective-C”一部分。

在Objective-C中，通过使用类别向现有类添加方法，可以改变类的行为。Objective-C中没有类似于final的类。因为编译器需要知道超类定义的对象大小，所以任何类只要包含子类头文件，就可以设置为子类。

实际上，相对于Java而言，在Objective-C中很少使用子类化行为。通过类别和动态运行时机制，可以向任何对象发送任何消息，所以可以将某些功能放到含有较少功能的类中，也可以放到最有意义的类中。例如，可以在NSString上加入类别来添加反转字符串或删除所有空格等特性，然后可以在任何NSString类中调用该方法，无论调用来自何处。当然你也可以使用自己的字符串子类来提供这些特性。

一般来说，只有当创建某个全新的对象（位于对象层次结构的顶部），需要从根本上改变某个对象的行为，或者由于类不能实现某个功能而需要使用子类时，才需要在Cocoa中设置子类。

例如，Cocoa使用NSView类构造用户界面组件，却无法实现它的drawRect:方法。因此，需要设置NSView的子类并重写drawRect:方法来绘制视图。但对其他大多数对象，通常采用委托和数据源的方法。由于Objective-C可以向任何对象发送任何消息，对象不必含有特定的子类或遵从特定的接口，这样，单个类就可以成为任意个不同对象的委托和数据源。

因为类别中已经声明了数据源和委托方法，因此不必实现。在Objective-C中，Cocoa编程很少使用空存根方法，某些方法会在嵌入式对象中调用相同的方法来使编译器顺利地适应一种正式协议。

当然，功能越强，责任越大。Objective-C采用手动保留、释放和自动释放的内存管理系统，这样更容易产生棘手的内存错误。在其他类中添加类别是一种功能强大的工作机制，但如果随意滥用，会降低代码的可读性，导致其他人无法理解。另外，Objective-C是以C语言为基础的，因此，在获得C语言的所有强大功能的同时，你也会面临处理器可能带来的危险，包括与指针相关的内存管理错误。

A.6　从BASIC转向Objective-C

许多编程人员都知道如何使用Visual Basic或REALbasic进行编程，但当转向Cocoa和Objective-C时可能非常困惑。

BASIC（Visua和REAL版本）环境提供的集成开发环境由完整的工作区组成。Cocoa将开发环境分成两个部分：Interface Builder编辑器和代码编辑器，它们都在Xcode界面中。你可以使用Interface Builder创建用户界面，并通知用户界面将要在某个特定对象上调用的方法名，然后在Xcode的代码编辑器上（也可以是TextMate、BBEdit、emacs以及你喜欢的任何文本编辑器）将控制逻辑添加到源代码中。

在BASIC中，用户界面项及其使用的代码紧密结合在一起。可以将代码段放入按钮和文本框代码中，使它们按照需要的方式进行工作，也可以将该代码段从普通类中分离出来，然后放在按钮代码中同该类进行交互。但在多数情况下，BASIC编程需要将代码放在用户界面项目中。如果你不注意，这种风格会导致许多不同项目的程序与逻辑对象的分布不统一。通常，BASIC编程都需要更改对象的属性，以便它们按照需要的方式进行工作。

在Cocoa中可以发现，它明确分离了界面和运行在界面之后的逻辑。它通过对象集合进行交互。你请求对象更改属性，而不是在某个对象上设置属性。这种区别很微妙但非常重要。在Cocoa编程中，大部分时间都用于思考需要发送什么消息，而不是需要设置什么属性。

BASIC含有非常丰富的第三方控件和支持代码。通常，可以购买某些现成产品并集成到代码库中，而不必自己去构建它。

A.7　从脚本语言转向Objective-C

对于使用脚本语言（例如Perl、PHP、Python和Tcl）的编程人员来说，可能很难适应Objective-C和Cocoa的编程环境。

脚本语言给编程人员提供了许多便捷之处，例如非常强大的字符串处理和操作功能、自动内存管理功能（无论是引用计数还是后台的垃圾回收）、快速的开发周期、灵活的输入（能够在数字、字符串和列表间轻松切换），以及大量可以下载并使用的程序包。同时，脚本语言中的运行时环境也非常灵活，可以通过它随意设计自己的对象类型和控制结构。

如果你是使用脚本语言的编程人员，也许会认为Objective-C在很多方面落后于时代的发展。Objective-C语言源于20世纪80年代，而脚本语言源于20世纪90年代。在Objective-C中，处理字符串可能非常痛苦，因为它没有内置正则表达式功能。不过，使用`printf()`风格输出字符串也和Cocoa一样巧妙。虽然Objective-C含有成熟的垃圾回收功能，但在因特网上你可以看到很多现有代码都使用手动内存管理技术（如`retain`和`release`）。Objective-C的开发过程包括编译和连接阶段，这导致编写代码和查看结果之间存在延迟。此外还必须手动处理不同的类型，例如整数、字符数组和字符串对象。另外还带有C语言的所有特性，例如指针、按位运算以及很容易产生的内存错误等。

为什么要承受使用Objective-C的痛苦呢？性能是一个原因，对于某些应用程序，Objective-C的性能优于脚本语言。另一个重要原因是能够访问本地用户界面工具包（Cocoa）。多数脚本语言都支持最初为Tcl语言开发的Tk工具包。这种工具包简单易用，但所具备的用户界面特性远不及Cocoa。最后一个重要的原因是，使用Tk构建的应用程序的外观和感觉都不同于iOS和Mac程序。

不过，使用脚本作为桥梁，就可以获取两家之长。通过在Objective-C和Python之间（被称为PyObjC），以及Objective-C和Ruby之间（被称为RubyObjC）搭建桥梁，这两种脚本语言都能够成为Objective-C的优秀（first class）成员。通过这些桥梁，可以使用Python或Ruby设置Cocoa子类对象，从而使用Cocoa的所有特性。

A.8　小结

Objective-C和其他编程语言及工具不同。由于其动态运行时分配性质，它具有一些灵巧的特性和行为。有些功能可以在Objective-C中实现，但无法在其他语言中实现。

Objective-C中也缺少其他语言早已具备的一些优良特性，特别是强大的字符串处理、命名空间及元编程特性。

编程中所做的每件事都需要进行权衡。需要确定的是，与使用当前的语言相比，使用Objective-C是否利大于弊。对我们而言，熟悉Objective-C很容易，而要使用Cocoa工具构建应用程序，还需要花费更多的时间，付出更大的努力。

索　引

C++, 1-3, 351-358
  混合语言, 355
  元数据, 315, 353

Foundation Kit, 3, 119, 264
  NSArray, 17, 128-136, 322-324,340-346
  NSDictionary, 98, 136-138, 140-141, 342-345
  NSNull, 141, 190-191, 207, 325-327, 343
  NSNumber, 119, 322-324, 343-345
  NSValue, 140, 316
  UIKit, 3, 119, 123, 179, 227, 234, 280, 284
  单例架构, 142
  封装, 138-140, 318
  可变数组, 133-135, 143, 189, 311, 319, 341
  快速枚举, 135-136, 144-145
  枚举, 33-36, 134-136, 143-145, 357
  装箱和开箱, 316
  字符串, 14-16, 358-360

Java, 1-3, 357-358

Objective-C, 1
  Hello World, 7
  Java, 1
  布尔类型, 18
  实例化, 41, 49, 351

B

并发性, 247, 248, 255, 263, 328
  操作队列, 261
    代码块操作, 262
    调用操作, 262
  调度, 43, 61-62, 65-67, 259-261, 352
  队列, 257-263
    调度队列, 257-259, 262
    死锁, 257, 260, 263
    先入先出（FIFO）, 257-258
    应用程序主线程, 258
  内存管理, 3, 58, 145-147, 257-259, 358-359
    引用计数对象, 258

D

代码块, 48, 52-53, 175- 178, 247-255, 258-263
  本地变量, 251- 252
  闭包, 248
  变量, 1, 21-26, 248-253, 323-325
  参数对象, 256
  函数指针, 39, 248-249, 352
  全局变量, 81-82, 158, 251, 353, 357
对象初始化, 3, 148, 180, 295
  ARC, 162-163, 166-173, 259-260
  便利初始化函数, 183, 193-194
  分配, 34, 146-147, 238-242, 331-335
  垃圾回收, 160-162, 166-169, 192-193
  指定初始化函数, 194, 196-198, 202, 240

F

访问方法, 64, 150-151, 185-186, 205-208
  引擎, 74, 316-318, 339-340
复合, 54, 67, 69, 71, 78-81
继承, 2-3, 52-58, 60-69, 78-80, 351-352, 355
  超类方法, 67, 325
  类代码, 62, 67, 77, 81, 193
  实例变量, 44, 353-354
  重构, 59, 78, 280

J

间接, 3, 23-30, 338, 348
文件名, 27, 30-31, 82, 86, 94, 99, 143-144, 305
命令行参数, 30-31
键/值编码, 205, 312-313, 325-327
对象所有权, 76, 149
临时对象, 157-158, 353
释放池清理, 333
引用计数, 147, 162, 179, 206, 257, 359
自动释放池, 142, 152-156, 158-161, 357
静态分析器, 327-329, 333-334, 336
过度释放, 336
无效变量, 330

L

类别, 3-4, 58, 340-341, 358
委托方法, 228, 230-232, 245-246, 358
NSString, 15-17, 19-22, 358
创建方法, 138-139, 275
类方法, 81, 122-123 , 339
类扩展, 221-223
实现文件, 82, 84, 317, 357
项目, 1, 7, 359
main()函数, 17-18, 321
类的声明, 46, 86, 219, 224, 236
前向引用, 89, 221, 227

M

面向对象的编程（OOP）
编程技术, 23, 313

S

属性, 57, 198-206, 212-215, 310-322
点表示法, 3, 210-211, 215, 313

T

调试, 93-94, 112-118, 130-132
暴力测试, 112
调用栈, 115
断点, 93, 113-117, 130-132
栈跟踪, 115

W

谓词, 338-348
过滤器, 338, 340, 345
数组运算符, 344
文件处理
属性列表, 302, 304-306, 312

X

协议, 3, 58, 357-358

Y

异常处理, 174, 176-178, 357
僵尸异常, 178
抛出异常, 173-176, 178, 357
自动释放池, 142, 152-156, 158-161, 357
应用程序工具集（AppKit）, 228
MSCAppDelegate, 266-267, 275-277, 284
连接, 36, 299-301, 359
用户界面, 1-2, 358-360
标签, 96-97, 298-299
库, 14, 268- 271, 338-340, 349-352
添加控件, 285
文本框, 9, 264-266, 295-298, 359
源文件组织, 81

Z

自动引用计数（ARC）, 185
强引用, 163-166
桥接转换, 172-173
弱引用, 163-166
归零弱引用, 162, 165-166
转换, 11, 305-307, 352-353
垃圾回收, 160-162, 166-169

[General Information 本信息由OnlyDown 1.6秋意版生成]
书名=Objective-C基础教程（第2版）
作者=（美）Scott Knaster, Waqar Malik, Mark Dalrymple著；周庆成译
页码=315
ISBN=
SS号=13248733
dxNumber=000011757330
出版时间=2013.05
出版社=人民邮电出版社
定价:
试读地址=http://book.szdnet.org.cn/views/specific/2929/bookDetail.jsp?dxNumber=000011757330&d=41BFD992389FD76E04E070A75A5F4E5C&fenlei=1817040302#dtop

全文地址=26415580721b13b70c81053cc7938771/img15/B3AA00729269986610D80E6E181AF89F13107E3CB82986C95C2760A4AA03B8613901E10358414F0F604A6C92AD42001CF79382EAD6B8A887D5CD16CEF183D36E5CA0DD18F10FE41A9CD0CA18C0C2CFBD9956B64AE276BD086326E3E27F37A317901F7DD2E5B7E1BCD464D36B03A12EC0F8DB/b48/qw/