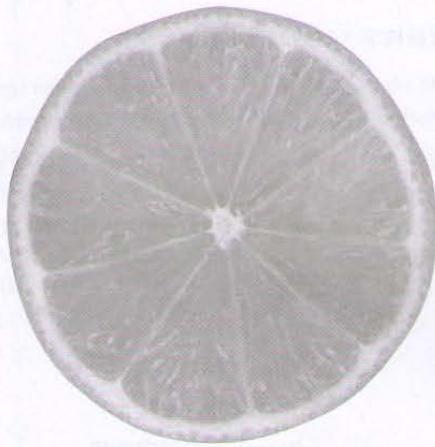


图灵程序设计丛书 移动开发系列



Learn Objective-C on the Mac

Objective-C基础教程

[美] Mark Dalrymple 著
Scott Knaster
高朝勤 杨越 刘霞 等译

人民邮电出版社

北京

图书在版编目 (CIP) 数据

Objective-C 基础教程 / (美) 达尔林普尔 (Dalrymple, M.), (美) 纳斯特 (Knaster, S.) 著; 高朝勤等译. — 北京: 人民邮电出版社, 2009.8
(图灵程序设计丛书)
书名原文: Learn Objective-C on the Mac
ISBN 978-7-115-20877-4

I. O… II. ①达…②纳…③高… III. C 语言-程序设计
IV. TP312

中国版本图书馆CIP数据核字 (2009) 第104204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, 全面而系统地讲述 Objective-C 编程的相关内容, 包括 Objective-C 在

C 的基础上引入的特性和 Cocoa 工具包的功能及其中的框架, 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

◆ 著 [美] Mark Dalrymple Scott Knaster
译 高朝勤 杨 越 刘 霞 等
责任编辑 傅志红
执行编辑 谢灵芝
◆ 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 电子函件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京隆昌伟业印刷有限公司印刷
◆ 开本: 800×1000 1/16
印张: 16.5
字数: 390千字 2009年8月第1版
印数: 1-3 000册 2009年8月北京第1次印刷
著作权合同登记号 图字: 01-2009-2897号
ISBN 978-7-115-20877-4/TP

定价: 49.00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* by Mark Dalrymple and Scott Knaster, published by Apress, 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705 USA.

Copyright © 2009 by Mark Dalrymple and Scott Knaster. Simplified Chinese-language edition copyright © 2009 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语言是C语言的一个扩展集，许多（可能是大多数）具备Mac OS X外观的应用程序都是使用该语言开发的。它以C语言为基础，添加了一些微妙但意义重大的特性。

苹果公司为Objective-C语言提供了Cocoa工具包。Cocoa是使用Objective-C编写的，它不仅包含Mac OS X用户界面的所有元素，还包含其他许多内容。Cocoa和Objective-C是苹果公司Mac OS X操作系统的核心。虽然Mac OS X相对较新，但Objective-C和Cocoa早在20世纪80年代就已诞生。它们由一个优秀的编程团队耗费数年时间完成，该团队从未停止过对Cocoa的更新和增强，所以直到现在，Cocoa仍在被广泛使用。

本书全面地介绍了Objective-C语言的基础知识。全书共分17章和1个附录，内容涵盖面向对象编程的基础知识，继承、复合、内存管理、对象初始化、协议、键/值编码等Objective-C语言特性，以及Xcode、Foundation Kit、AppKit等辅助工具。附录中还探讨了使用其他语言的编程人员在转向Objective-C时需要克服的一些思维障碍。本书内容丰富生动，采用示例程序与理论知识相结合的方式，不仅提供了代码片段，还有大量完整的示例程序。

本书由高朝勤、杨越、刘霞、刘炼、陈宝国等翻译，在翻译过程中得到了欧阳宇、盛海艳的热心帮助，在此表示衷心感谢。作为原著作者与中文读者之间的传话人，我们希望能把作者要表达的意思用中文完整且准确地传达出来，使更多读者能从中获益。虽然我们在翻译的过程中竭力以信、达、雅为目标，但由于水平有限，书中难免会出现疏漏，恳请广大读者给予批评指正。

译 者
2009年5月

前　　言

编程工作做久了，最初的新鲜感难免会消磨殆尽。幸好总是会有新的技术闪耀登场，重燃编辑人员的兴趣，Mac OS X就饱含这些神奇的技术。

Objective-C就是这样一种编程语言，它将快速而普及的C语言结合到优雅的面向对象环境中，使编程变得轻松惬意。学习Objective-C会使你迷上Apple的许多优秀技术，如Cocoa工具包和iPhone SDK。一旦掌握这门语言，你就可以进而征服Apple平台的其他技术，并由此笑傲江湖。

致 谢

技术书籍都会有致谢部分。虽然本书封面上只署了两个人的名字，但是还有许多人在幕后参与了本书的出版工作。

我们要特别感谢本书的总策划Denise Santoro Lincoln，我们给她出了一大堆难题，她则举重若轻且不失幽默地纷纷化解。感谢Clay Andres和Jeff LaMarche为本书所做的技术审查工作。真诚感谢本书的生产编辑Laura Esterman，她将一大堆文字用现在这样精美的形式呈现给了读者。感谢Heather Lang像我们一样绞尽脑汁斟酌书中的内容，出色地完成了文字编辑工作。

Mark致谢 感谢Aaron Hillegass，Aaron多年前指引他领略了Objective-C和Cocoa的精彩，并将他引荐给Scott和Dave。没有Aaron，就没有本书。还要感谢Greg Miller为他介绍了KVC和NSPredicate的优美之处。

Scott致谢 感谢Mark Dalrymple卓有成效的工作。

最后，万分感谢Dave Mark，没有他的远见卓识、坚持不懈和精益求精的精神，本书不可能出版。

目 录

第1章 启程	1
1.1 预备知识	1
1.2 历史背景	1
1.3 内容简介	2
1.4 小结	3
第2章 对C的扩展	4
2.1 最简单的Objective-C程序	4
2.2 解构Hello Objective-C程序	7
2.2.1 #import	7
2.2.2 NSLog()和@"字符串"	8
2.3 布尔类型	10
2.3.1 BOOL强大的实用功能	11
2.3.2 比较	13
2.4 小结	14
第3章 面向对象编程基础知识	15
3.1 间接	15
3.1.1 变量与间接	16
3.1.2 使用文件名的间接	18
3.2 在面向对象的编程中使用间接	24
3.2.1 过程式编程	24
3.2.2 实现面向对象编程	29
3.3 学习有关的术语	33
3.4 Objective-C中的OOP	34
3.4.1 @interface部分	34
3.4.2 @implementation部分	38
3.4.3 实例化对象	40
3.4.4 扩展Shapes-Object	41
3.5 小结	43
第4章 继承	45
4.1 为何使用继承	45
4.2 继承语法	48
4.3 继承的工作机制	51
4.3.1 方法调度	51
4.3.2 实例变量	53
4.4 重写方法	55
4.5 小结	57
第5章 复合	58
5.1 什么是复合	58
5.1.1 Car程序	58
5.1.2 自定义NSLog()	59
5.2 存取方法	62
5.2.1 设置发动机的属性	64
5.2.2 设置轮胎的属性	64
5.2.3 跟踪汽车的变化	66
5.3 扩展CarParts程序	67
5.4 复合还是继承	68
5.5 小结	69
第6章 源文件组织	70
6.1 拆分接口和实现部分	70
6.2 拆分Car程序	73
6.3 使用跨文件依赖关系	75
6.3.1 重新编译须知	75
6.3.2 让汽车开动	77
6.3.3 导入和继承	79
6.4 小结	80

2 目录

第 7 章 深入了解 Xcode	82
7.1 改变公司名称	82
7.2 使用编辑器的技巧与诀窍	83
7.3 在Xcode的帮助下编写代码	85
7.3.1 首行缩进	85
7.3.2 代码自动完成	85
7.3.3 括号匹配	88
7.3.4 批量编辑	88
7.3.5 代码导航	91
7.3.6 emacs不是Mac程序	91
7.3.7 任意搜索	92
7.3.8 芝麻开门	93
7.3.9 书签	93
7.3.10 集中注意力	94
7.3.11 开启导航条	95
7.4 获取信息	98
7.4.1 研究助手	98
7.4.2 文档管理程序	99
7.5 调试	100
7.5.1 暴力调试	100
7.5.2 Xcode的调试器	100
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
7.5 备忘表	105
7.6 小结	106
第 8 章 Foundation Kit 快速教程	107
8.1 一些有用的数据类型	108
8.1.1 范围的作用	108
8.1.2 几何数据类型	108
8.2 字符串	109
8.2.1 创建字符串	109
8.2.2 类方法	109
8.2.3 关于大小	110
8.2.4 比较的策略	110
8.2.5 不区分大小写的比较	112
8.2.6 字符串内是否还包含别的字符串	112
8.3 可变性	113
第 8 章 深入了解 Xcode	82
8.4 集合家族	115
8.4.1 NSArray	115
8.4.2 可变数组	118
8.4.3 枚举“王国”	119
8.4.4 快速枚举	120
8.4.5 NSDictionary	120
8.4.6 使用, 但不要扩展	122
8.5 各种数值	122
8.5.1 NSNumber	122
8.5.2 NSValue	123
8.5.3 NSNull	124
8.6 示例: 查找文件	124
8.7 小结	128
第 9 章 内存管理	129
9.1 对象生命周期	129
9.1.1 引用计数	130
9.1.2 对象所有权	132
9.1.3 访问方法中的保留和释放	133
9.2 自动释放	134
9.2.1 所有对象全部入池	135
9.2.2 自动释放池的销毁时间	135
9.2.3 自动释放池的工作过程	136
9.3 Cocoa内存管理规则	138
9.3.1 临时对象	138
9.3.2 拥有对象	139
9.3.3 垃圾回收	141
9.4 小结	142
第 10 章 对象初始化	143
10.1 分配对象	143
10.2 初始化对象	143
10.2.1 编写初始化方法	144
10.2.2 初始化时做什么	146
10.3 便利初始化函数	146
10.4 更多部件改进	147
10.4.1 Tire类的初始化	147
10.4.2 更新main()函数	149
10.4.3 清理Car类	152
10.5 支持垃圾回收风格的Car类清理	155

10.6 指定初始化函数.....	156	13.2.2 复制Tire	194
10.6.1 子类化问题.....	157	13.2.3 复制Car	196
10.6.2 改进Tire类的初始化函数.....	159	13.2.4 协议和数据类型.....	199
10.6.3 添加AllWeatherRadial类的初始化函数.....	160	13.3 Objective-C 2.0的新特性	199
10.7 初始化函数规则.....	160	13.4 小结	200
10.8 小结	161	第 14 章 AppKit 简介	201
第 11 章 特性	162	14.1 构建项目	201
11.1 修改特性值	162	14.2 构建AppController @interface	203
11.1.1 简化接口	163	14.3 Interface Builder	203
11.1.2 简化实现	164	14.4 布局用户界面	205
11.1.3 点表达式的妙用	166	14.5 连接	207
11.2 特性扩展	167	14.5.1 连接输出口	207
11.2.1 名称的使用	171	14.5.2 连接操作	208
11.2.2 只读特性	172	14.6 AppController实现	210
11.2.3 特性不是万能的	173	14.7 小结	212
11.3 小结	173	第 15 章 文件加载与保存	213
第 12 章 类别	175	15.1 属性列表	213
12.1 创建类别	175	15.1.1 NSDate	213
12.1.1 声明类别	175	15.1.2 NSData	214
12.1.2 实现类别	176	15.1.3 写入和读取属性列表	215
12.1.3 类别的局限性	178	15.2 编码对象	216
12.1.4 类别的作用	178	15.3 小结	221
12.2 利用类别分散实现	178	第 16 章 键/值编码	222
12.3 使用类别创建前向引用	182	16.1 入门项目	222
12.4 非正式协议和委托类别	183	16.2 KVC简介	224
12.4.1 iTunesFinder项目	184	16.3 路径	225
12.4.2 委托和类别	187	16.4 整体操作	226
12.4.3 响应选择器	187	16.4.1 中途小憩	227
12.4.4 选择器的其他应用	188	16.4.2 流畅地运算	231
12.5 小结	189	16.5 批处理	233
第 13 章 协议	190	16.6 nil仍然可用	234
13.1 正式协议	190	16.7 处理未定义的键	235
13.1.1 声明协议	190	16.8 小结	236
13.1.2 采用协议	191	第 17 章 NSPredicate	237
13.1.3 实现协议	192	17.1 创建谓词	237
13.2 复制	192	17.2 燃料过滤器	239
13.2.1 复制Engine	192	17.3 格式说明符	240
		17.4 运算符	241

4 目 录

17.4.1 比较和逻辑运算符	242
17.4.2 数组运算符	243
17.5 SELF足够了	243
17.6 字符串运算符	245
17.7 LIKE运算符	245
17.8 小结	246

附录 从其他语言转向 Objective-C	247
------------------------	-----

第1章

启 程

1

欢迎阅读本书！本书旨在介绍Objective-C语言的基础知识。Objective-C语言是C语言的一个扩展集，许多具备Mac OS X外观的应用程序都是使用该语言开发的。

本书介绍Objective-C语言以及苹果公司为其提供的Cocoa工具包。Cocoa是使用Objective-C编写的，它不仅包含Mac OS X用户界面的所有元素，还包含其他许多内容。通过本书掌握Objective-C之后，读者将能够使用Cocoa开发功能完备的项目，并且可以深入阅读Apress出版社2009年出版的*Learn Cocoa on the Mac*和*Beginning iPhone Development*^①等著作，这两部著作都是由Dave Mark和Jeff LaMarche编写的。

本章将介绍阅读本书所需的基本信息，还将介绍Objective-C的历史以及其他章节的概要信息。

1.1 预备知识

在阅读本书之前，读者应具备使用与C类似的编程语言（如C++、Java或C语言）的一些经验。无论使用哪种语言，都应熟悉其基本原理。应该理解什么是变量和函数，知道如何使用条件和循环语句控制程序流。我们将重点介绍Objective-C在其基础语言C中添加的特性，以及苹果公司Cocoa工具包的一些优秀特性。

对于不具备C语言基础的Objective-C学习者，可以先阅读附录或Dave Mark编写的*Learn C on the Mac* (Apress, 2009)。

1.2 历史背景

Cocoa和Objective-C是苹果公司Mac OS X操作系统的核心。虽然Mac OS X相对较新，但Objective-C和Cocoa的推出已有时日。早在20世纪80年代早期，Brad Cox就发明了Objective-C，意在将流行的、可移植的C语言与优雅的Smalltalk语言结合在一起。1985年，Steve Jobs成立了NeXT公司，致力于开发强大且经济的工作站。NeXT选择Unix作为其操作系统，创建了NextSTEP（使用Objective-C开发的一款强大的用户界面工具包）。NextSTEP只是创造了一些特性，拥有少量

^① 中文版《iPhone开发基础教程》已由人民邮电出版社出版。——编者注

忠实拥趸，并未在商业上获得成功。

在苹果公司于1996年收购NeXT之后，NeXTSTEP被重命名为Cocoa，并得到了Macintosh编程人员的广泛认可。苹果公司免费提供其开发工具（包括Cocoa），只需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Mac编程人员都可以利用这些工具。

有人可能会问：“既然Objective-C和Cocoa是在20世纪80年代发明的（Alf和A-Team的时代，更不用提古老的Unix了），难道它们现在还没有过时吗？”当然没有！Objective-C和Cocoa是由一个优秀的编程团队耗费数年时间完成的，并且该团队从未停止对Cocoa的更新和增强。经过多年发展，Objective-C和Cocoa已经演化成一个功能强大的优秀工具集。Objective-C在iPhone应用程序开发中也发挥着重要的作用。因此，在NeXT采用Objective-C二十多年后的今天，Objective-C仍然有着广泛的应用。

1.3 内容简介

Objective-C是C语言的一个扩展集。Objective-C以C语言为基础，在该语言中添加了一些微妙但意义重大的特性。使用过C++或Java的用户一定会惊叹Objective-C程序的简短。本书将详细介绍Objective-C在C语言的基础上添加的特性。

第2章着重介绍Objective-C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章介绍如何创建具备其父类特性的类。

第5章讨论结合对象协同工作的技巧。

第6章介绍创建程序源文件的实际策略。

第7章介绍一些快捷方法和强大特性，帮助你最大程度地提高编程效率。

第8章介绍Cocoa的两个主要框架之一，加深你对Cocoa中一些优秀特性的理解。

第9章介绍Cocoa应用程序。

第10章讨论对象的初始化。

第11章介绍Objective-C中新增的点表示法以及构建对象访问方法的一种简单方式。

第12章介绍Objective-C中的一个非常出色的特性：支持在现有类中添加自己的方法，甚至可以添加别人编写的方法。

第13章讨论Objective-C中的一种继承方式，这种方式允许类实现打包的特性集。

第14章介绍如何使用Cocoa的另一个主要框架开发优秀的应用程序。

第15章介绍如何保存和检索数据。

第16章介绍如何间接操作数据。

最后，第17章介绍如何分解数据。

如果读者之前使用的是Java或C++等其他语言，或使用Windows或Linux等其他平台，那么可以先阅读附录，其中指出了学习Objective-C需要克服的一些思维障碍。

1.4 小结

Mac OS X程序是使用Objective-C编写的，它所使用的技术可以追溯到20世纪80年代，这些技术已演化成一个强大的工具集。本书假定读者对C语言编程有一定的了解，在此基础上介绍Objective-C。

希望读者可以从本书中受益！

Objective-C只不过是拥有一些附加特性的C语言，但它很好用！本章将指导你构建第一个Objective-C程序，同时介绍一些关键的附加特性。

2.1 最简单的Objective-C程序

你可能见过C语言版本的经典“Hello World”程序，该程序可输出“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。我们将继承此优良传统，编写一个类似的程序，这里称之为“Hello Objective-C”。

构建Hello Objective-C

在阅读本书的过程中，我们假定你已经安装了苹果公司的Xcode工具。如果你尚未安装Xcode，或者从未使用过此工具，请参阅Dave Mark编写的*Learn C on the Mac* (Apress, 2009)，该书第2章将指导你购买、安装Xcode并使用它编写程序。

本节将详细介绍Xcode的使用步骤，创建第一个Objective-C项目。如果你已经熟悉Xcode，请略过此部分，这不会伤害我们的感情。在继续下一步之前，请确定打开了本书归档文件中名为Learn ObjC Projects的归档文件（可从Apress网站的Source Code/Download页面下载^①）。这个项目位于02.01-Hello Objective-C文件夹中。

要创建项目，首先请启动Xcode。可以在/Developer/Applications目录下找到Xcode应用程序。我们将Xcode图标放在Dock快捷工具栏中，以便于访问。你也可以这么做。

Xcode启动完毕后，从File菜单中选择New Project。Xcode将列表显示它支持创建的各种项目。集中注意力，别被其他许多项目类型吸引，选择窗口左侧的Command Line Utility，再选择右侧的Foundation Tool，如图2-1所示，单击Choose按钮。

Xcode将显示一个工作表，并要求你为项目命名。你可以指定任意名称，这里命名为Hello Objective-C，如图2-2所示。我们将此项目放在Projects目录下，以保证系统井然有序，而你可以将其保存在你希望的任何位置。

^① 本书代码也可以从图灵网站www.turingbook.com本书网页免费注册下载。——编者注

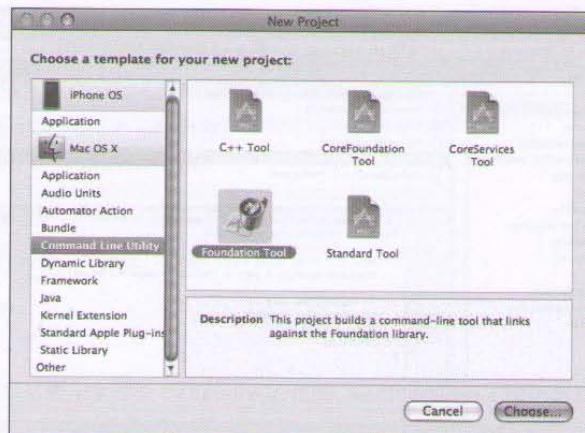


图2-1 生成新基础工具

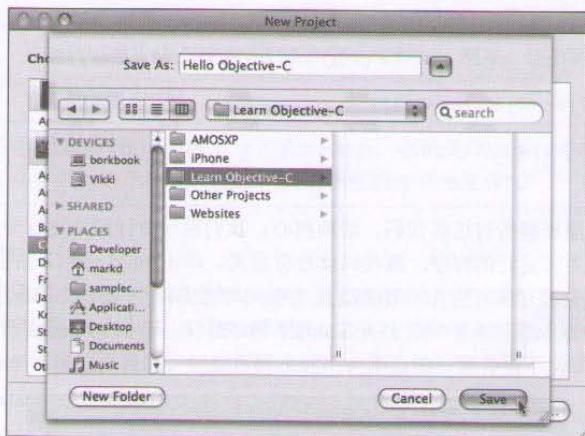


图2-2 为新基础工具命名

单击Save之后, Xcode会显示其主窗口, 即项目窗口(如图2-3所示)。此窗口显示项目的各组成部分以及编辑窗格。突出显示的文件Hello Objective-C.m为源文件, 其中包含Hello Objective-C的代码。

Xcode体贴地为每个新项目都准备了样本代码, Hello Objective-C.m中也存在这些代码。我们可以让Hello Objective-C应用程序比Xcode提供的实例更简单一点。删除Hello Objective-C.m中所有内容, 将其替换为以下代码:

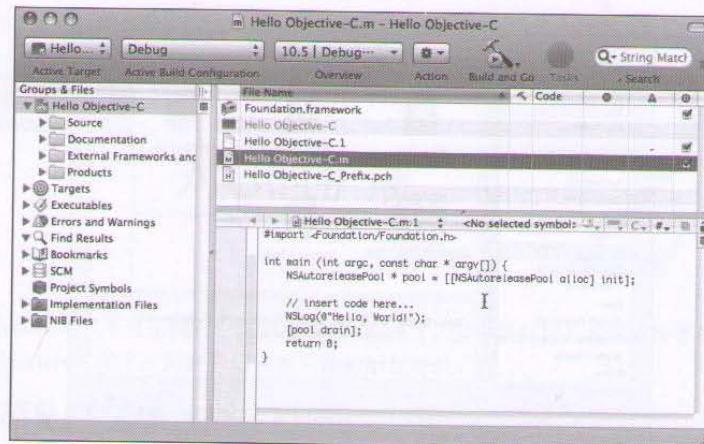


图2-3 Xcode的主窗口

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

int main (int argc, const char *argv[])
{
    NSLog (@"Hello, Objective-C!");
    return (0);
} // main
```

如果你目前不能理解所有这些代码,请别担心。我们稍后将详细地逐行分析此程序。

如果不能转换为可运行的程序,源代码就没有意义。单击Build and Go按钮或按下 ⌘R ,将生成并运行程序。如果没有任何恼人的语法错误,Xcode就会编译并链接你的程序,随后运行。从Run菜单中选择Console或按下 ⌘↑R ,打开Xcode控制台窗口,其中会显示程序的输出结果,如图2-4所示。

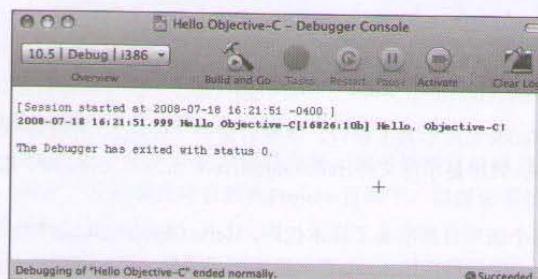


图2-4 运行Hello Objective-C

你已经完成了你的第一个Objective-C程序。恭喜你！接下来剖析它的[工作方式](#)。

2.2 解构 Hello Objective-C 程序

2

再次列出Hello Objective-C.m的内容：

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

int main (int argc, const char *argv[])
{
    NSLog (@"Hello, Objective-C!");
    return (0);
} // main
```

Xcode的.m扩展名表示文件含有Objective-C代码，应由Objective-C编译器处理。C编译器处理名称以.c结尾的文件，而C++编译器处理.cpp文件。在Xcode中，所有这些编译工作全由GCC（GNU Compiler Collection，GNU编译器集合）处理，这个编译器能够理解C语言的全部3个变体。

如果了解普通的C语言，那么你应该很熟悉.m主文件中包含的两行代码：`main()`声明和末尾的`return(0)`语句。请记住，Objective-C本质上就是C语言，它用来声明`main()`和返回值的语法和C语言是一样的。其他代码看上去与正规的C语言有些许差别，例如，这个陌生的`#import`是什么？如果想知道答案，请继续阅读！

说明 Objective-C刚诞生的时候，扩展名.m代表message，指的是Objective-C的一个主要特性，后面的章节将会介绍这个特性。现在，我们称它们为“.m文件”。

2.2.1 #import

和C语言一样，Objective-C使用头文件来包含元素声明，这些元素包括结构体、符号常量、函数原型等。C语言使用`#include`语句通知编译器应在头文件中查询定义。在Objective-C程序中也可使用`#include`来实现这个目的。但你可能永远不会那么做，而会像下面这样使用`#import`：

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

#import是GCC编译器提供的，Xcode在编译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，而不论此命令实际上在那个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，而后者又包含第一个文件的情况。

而在Objective-C中，程序员使用`#import`实现这个功能。

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

什么是框架？很高兴你问了这个问题。框架是一种聚集在一个单元的部件集合，包含头文件、库、图像、声音文件等。苹果公司将Cocoa、Carbon、QuickTime和OpenGL等技术作为框架集提供。Cocoa的组成部分有Foundation和Application Kit（也称为AppKit）框架。还有一个支持框架的套件，包含Core Animation和Core Image，这为Cocoa增添了多种精彩功能。

Foundation框架处理的是用户界面之下的层（layer）中的特性，例如数据结构和通信机制。本书中所有程序都以Foundation框架为基础。

说明 学完本书后，要想成为Cocoa权威专家，还需要精通Cocoa的Application Kit，它包含Cocoa的高级特性：用户界面元素、打印、颜色和声音管理、AppleScript支持等。欲了解更多信息，请参阅Dave Mark和Jeff LaMarche编著的*Learn Cocoa on the Mac* (Apress, 2009)。

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，它包含了所有框架的各个头文件。通过使用#import导入主头文件，可以使用所有框架的特性。

Foundation框架的头文件占用了近1MB的磁盘存储空间，包含一万四千多行代码，涵盖一百多个文件。使用#import <Foundation/Foundation.h>包含主头文件，就能够获得整个集合。也许你认为辛苦地读取每个文件的全部文本会耗去编译器很多时间，但是Xcode非常聪明：它会用预编译头文件（一种经过压缩的、摘要形式的头文件），在通过#import导入这种文件时，加载速度会非常快。

如果你想知道Foundation框架包含哪些头文件，可以查看其Headers目录（/System/Library/Frameworks/Foundation.framework/Headers/）。如果只是浏览文件，而不删除或更改它们，就不会造成任何破坏。

2.2.2 NSLog()和@"字符串"

使用#import导入了Foundation框架的主头文件之后，就可以开始利用Cocoa特性编写代码了。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()。而NSLog()这个Cocoa函数的作用和printf()很相似。

和printf()一样，NSLog()接受一个字符串作为其第一个参数，该字符串可包含格式说明符（如%d）。此函数还可以接受匹配格式说明符的其他参数，printf()可在打印之前将这些参数插入到作为第一个参数的字符串中。

之前说过，Objective-C只是增加了一点“特殊调料”的C语言，所以可以用printf()代替NSLog()。但我们建议使用NSLog()，因为它添加了特性，例如时间戳、日期戳和自动附加换行符（'\n'）等。

你也许不太理解函数名NSLog()。这里的“NS”是什么意思？其实，Cocoa对其所有函数、

变量和类型名称都添加了“NS”前缀。这个前缀告诉你函数来自Cocoa而不是其他工具包。

两个不同事物使用相同标识符时会导致名称冲突，而前缀可以预防这个大问题。如果Cocoa将此函数命名为`Log()`，那么这个名称很可能和一些程序员创建的`Log()`函数冲突。当包含`Log()`的程序和Cocoa一起构建时，Xcode会警告`Log()`被多次定义，将产生糟糕的结果。

知道了前缀的好处，可能你又会奇怪前缀为何是“NS”而不是“Cocoa”？“NS”前缀的来历要追溯至此工具包还被称为NextSTEP，而且是NeXT Software公司（前NeXT公司，于1996年被苹果公司收购）产品的时候。但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，不应该再为自建的变量或函数名称添加前缀“NS”。否则处理代码的阅读器会发生混乱，它们会认为你创建的内容实际上属于Cocoa。同样，假如将来苹果公司为Cocoa添加了一个函数，碰巧和你创建的名称相同，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，本书中不为代码使用前缀。

再看看这条`NSLog()`语句：

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

你是否注意到了字符串前的`@`符号？这可不是我们警惕的编辑漏掉的录入错误。`@`符号是Objective-C在标准C语言基础上添加的特性之一。双引号中的字符串前有一个`@`符号，这表示引用的字符串应该作为Cocoa的`NSString`元素来处理。

那么什么是`NSString`元素？去掉其“NS”前缀，就可以看到熟悉的术语“String”。你已经知道字符串就是一串字符，通常是人类可读懂的，所以你一定能（准确地）猜到`NSString`就是Cocoa中的一串字符。

`NSString`元素有许多打包的特性，Cocoa在需要字符串时可随时使用它们。下面是一些`NSString`功能。

- 告知其长度；
- 将自身与其他字符串比较；
- 将自身转换为整型值或浮点值。

还有许多功能是使用C风格字符串无法实现的。第8章将更多地使用并研究`NSString`元素。

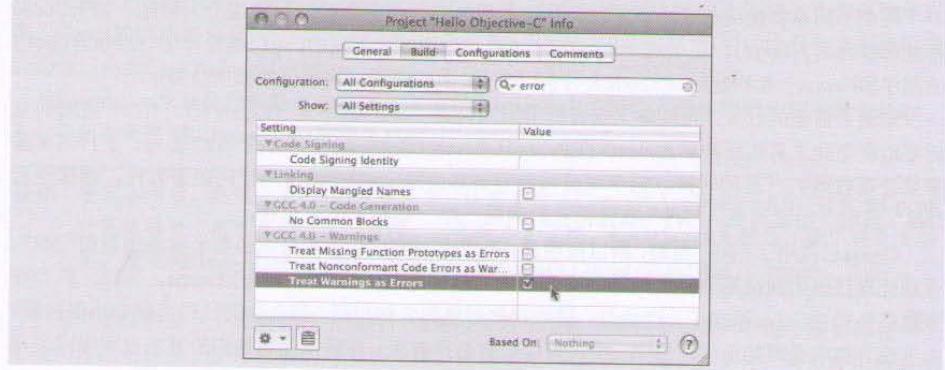
观察这些字符串

一个易犯的错误是将C风格字符串（而不是专门的`NSString`的`@"字符串"元素`）传递给`NSLog()`。如果是这样，编译器会给出警告：

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

如果要运行这个程序，它可能会崩溃。要捕捉这样的问题，可以让Xcode总是将警报作为错误来处理。方法是，选择Xcode Groups & Files列表顶端的项目，选择File ▶ Get Info命令，选择Build选项卡，在搜索区输入“error”，然后勾选Treat Warnings as Errors复选框，如下图所示。

示。还要确保在顶部的Configuration弹出菜单中选中了All Configurations。



`NSString`还说明了一个绝妙的事实：名称本身突出了Cocoa的一个好特性。大多数Cocoa元素都以非常直接的方式命名，名称尽量描述出它们可实现的特性。例如，`NSArray`提供数组，`NSDateFormatter`帮助你用不同方式来格式化日期，`NSThread`提供多线程编程工具，`NSSpeechSynthesizer`使你听到语音。

下面，我们单步调试这个小程序。此程序最后一行是返回语句，可终止执行`main()`并结束程序：

```
return (0);
```

返回的0值说明我们的程序成功完成了。C语言里返回语句的工作方式也是这样。

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

2.3 布尔类型

许多语言都有布尔（Boolean）类型，当然这是个专用术语，指的是存储真值和假值的变量。Objective-C也不例外。

C语言拥有布尔数据类型`bool`，它具有`true`值和`false`值。Objective-C提供了相似的类型`BOOL`，它具有`YES`值和`NO`值。顺便提一下，Objective-C的`BOOL`类型比C语言的`bool`类型早诞生十多年。这两种不同的布尔类型可以在同一个程序中共存，但在编写Cocoa代码时要使用`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`)的定义(`typedef`)，它使用8位存储空间。`YES`定义为1，而`NO`定义为0(使用`#define`)。

Objective-C并不将`BOOL`作为仅能保存`YES`或`NO`值的真正的布尔类型来处理。编译器将`BOOL`认做8位二进制数，`YES`和`NO`值只是一种约定。这引发了一个小问题：如果不小心将一个长于1字节的整型值(例如`short`或`int`值)赋给一个`BOOL`变量，那么只有低位字节会用作`BOOL`值。假设该低位字节刚好为0(例如8960，写成十六进制为`0x2300`)，`BOOL`值将会是0，即`NO`值。

2.3.1 BOOL 强大的实用功能

要展示BOOL的实用功能，我们看下一个项目：02.02 - BOOL Party。此项目比较几对整数，判断它们是否不同。除了main()之外，此程序还定义了两个函数。第一个函数是areIntsDifferent()，它接受两个整型值，返回一个BOOL：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参数并返回字符串，参数是YES时返回@"YES"，参数是NO时返回@"NO"。如果想以人类可读的形式输出BOOL值，那么这个函数很方便。main()使用这两个函数来比较整数，并输出结果。

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，则启动它。
- (2) 从File菜单中选择New Project。
- (3) 选择左边的Command Line Utility和右边的Foundation Tool。
- (4) 单击Choose。
- (5) 输入“BOOL Party”作为项目名（Project Name），再单击Save。

编辑BOOL Party.m，其内容如下所示：

```
#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 YES value, return the human-readable
// string "YES". Otherwise return "NO"

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

```

生成并运行程序。需要从Run菜单选择**Console**，或使用键盘快捷键 $\text{⌘}+\text{R}$ ，以调出Console窗口查看输出。在Run Debugger Console窗口中，应该可以看到如下输出：

```

2008-07-20 16:47:09.528 02 BOOL Party[16991:10b] are 5 and 5 different? NO
2008-07-20 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，逐个函数地看看发生了什么。我们要分析的第一个函数是**areIntsDifferent()**：

```

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。如果使用过C语言，你应该很熟悉其语法。可以看到，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，就返回YES。这非常直接明了，不是吗？

这样不会再得到布尔值

经验丰富的C语言程序员也许会试着将**areIntsDifferent()**函数写成一条语句：

```

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

```

他们之所以这样操作，是因为假定了非零值等于YES。但事实并非如此。是的，在C语言中此函数会返回一个真值或假值，但是，返回BOOL的函数调用者期望的是YES值或NO值。尝试像下面这样使用此函数的程序员将会失败，因为23减5等于18：

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

尽管上述函数在C语言中会得到真值，但在Objective-C中1不等于YES。

绝不要直接将BOOL值和YES比较。聪明过头的程序员有时会玩玩花样，使用类似的areIntsDifferent_faulty()函数。相反，应该将上述if语句改成如下所示：

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

直接和NO比较则一定安全，因为C语言中的假值就是0。

第二个函数boolString()将数值型BOOL值映射为人类可读的字符串：

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

// boolString

你应该很熟悉这个函数中的if语句。它只是比较了yesNo和常量NO，如果二者匹配则返回@"NO"。否则，yesNo一定是真值，所以它返回@"YES"。

注意，boolString()的返回类型是一个指向NSString的指针。这意味着函数会返回一个Cocoa字符串，这早在首次遇到NSLog()时你就看到过。如果观察返回语句，会发现返回的值前面有@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返回值作为boolString函数的参数，但添加这样一个变量而使代码易于阅读是没有坏处的。深度嵌套的结构经常令人困惑，也不好理解，而且往往也是bug的藏身之所。

2.3.2 比较

下面的两行代码使用areIntsDifferent()比较一对整数，将返回值存储到areTheyDifferent变量中，并通过NSLog()输出数字值和boolString()返回的人类易读的字符串：

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

```
5, 5, boolString(areTheyDifferent));
```

如前所见, `NSLog()`本质上就是Cocoa中的`printf()`函数, 它接受一个格式字符串, 并将后续参数的值插入到这个格式说明符中。在这里对`NSLog()`的调用中, 两个5将替换两个`%d`格式占位符。

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? 它表示`boolString()`返回一个`NSString`指针。`printf()`不能使用`NSString`, 所以没有我们能够使用的格式说明符。`NSLog()`的编写者添加`%@`格式说明符, 是为了通知`NSLog()`接受适当的参数, 将其作为`NSString`, 再使用该字符串中的字符, 并将其发送到控制台。

说明 我们还没有正式介绍过对象, 但现在可以简单地透露一下: 使用`NSLog()`输出任意对象的值时, 都会使用`%@`格式说明。在使用这个说明符时, 对象通过一个名为`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));
```

函数比较23和42。这次, 因为它们是不同的, 所以`areIntsDifferent()`返回YES。用户可以看到表示23和42不同的文本。

下面是最终的返回语句, 至此我们的BOOL Party就结束了。

```
return (0);
```

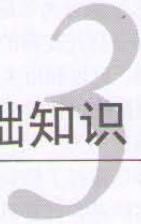
```
} // main
```

在这个程序中, 你学习了Objective-C的BOOL类型、指示真和假的常量YES和NO。可以像`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

本章中, 你编写了自己的前两个Objective-C程序, 这很有趣! 你还了解了Objective-C对语言的一些扩展, 例如`#import`让编译器引入一次头文件(且仅引入一次)。我们学习了`NSString`, 即以@符号开头的字符串, 例如`@"hello"`。使用了重要且通用的`NSLog()`, Cocoa提供的这个函数可将文本输出到控制台。还使用了`NSLog()`专用格式说明符`%@`, 它支持将`NSString`值插入到`NSLog()`输出中。你还掌握了一个诀窍: 在代码中发现@符号时, 你就会知道自己看到的是Objective-C对C语言的扩展。

下一章, 我们将进入面向对象编程的奇妙世界。



只要你曾使用过计算机且编写过计算机程序，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，最初是为编写模拟程序而开发的。OOP很快俘获了其他种类软件（例如涉及图形用户界面的软件）开发者的心。很快，OOP就成为一个重要的业内行话。它就像传说中有魔力的银弹，能够使编程变得简单而愉快。

当然，这种夸大之辞有些过分。未能免俗的是，要精通OOP，就需要学习和实践。OOP确实能简化某些编程任务，在某些情况下甚至能让编程变得很有趣。本书将大量讨论OOP，这主要是因为Cocoa基于OOP概念，并且Objective-C是一种面向对象的语言。

那么，什么是OOP？OOP这种方法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机器，它们互相交谈，协同完成工作。本章将着眼于一些基本的OOP概念，然后研究可实现OOP的编程风格，并讲述OOP特性背后的动机，最后将全面介绍一些OOP技巧。

说明 与许多“新”技术一样，OOP的起源要追溯到迷雾重重的过去。从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，最终演变产生了OOP。C++、Java、Python等现代语言，当然还有Objective-C，都从这些老的语言中获得了灵感。

在我们研究OOP的过程中，你也许需要一名贴身翻译，并准备好遭遇一些陌生的术语。OOP具有许多奇异而响亮的术语，于是看起来十分神秘而难以理解，但实际并非如此。你甚至可能认为，计算机科学家创造这些冗长而且极为夸张的词汇，是为了告诉所有人他们有多聪明；但是当然，他们并不总是这么做。好吧，请不用担心。因为我们会解释遇到的每个术语。

在讨论OOP本身之前，先看看OOP的一个关键概念：间接（indirection）。

3.1 间接

在编程行业有句老话，大意是：“只要多添加一个间接层，计算机科学中就没有解决不了的问题。”间接这个词的含义很简单——不在代码中直接使用某个值，而是使用指向该值的指针。下面是一个真实的例子：你可能不知道自己最喜欢的比萨饼店的电话号码，但你知道可以查阅电话号码簿来找到它。使用电话号码簿就是一种间接的形式。

间接还能解释为让其他人代替自己做事情。假设你有一箱书要还给朋友Andrew，他住在城镇另一头。你知道你的邻居今晚要去拜访Andrew。那么你不会开车横跨城镇，留下书，然后再开车回家；你会拜托友善的邻居送那箱书。这就是另一种间接：让他人代替你自己完成工作。

在编程时，可以利用多层间接，如编写一段代码来查询其他代码，而后者又可以访问另一层代码。你大概拨打过技术支持热线。你对支持员工说明了问题，他将你转接到能够处理此问题的代码。该部门员工又将你转接到下一级技术人员，他可帮你解决问题。如果你和我们一样，具体部门。在这时发现自己拨打了错误号码，那么你不得不转向另一部门寻求帮助。这种推诿就是一种形式的间接。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

你可能惊讶地发现自己在程序中已经使用过间接了。基本变量就是间接的一种实际应用。考虑下面这个输出数字1到5的小程序。在Learn ObjC Projects文件夹03.01 Count-1中可找到这个程序：

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

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

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

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

Count-1有一个运行5次的for循环，使用NSLog()来显示循环每次运行时i的值。运行此程序，你将看到如下输出：

```
2008-07-20 11:54:20.463 03.01 Count-1[17985:10b] The numbers from 1 to 5:
2008-07-20 11:54:20.466 03.01 Count-1[17985:10b] 1
2008-07-20 11:54:20.466 03.01 Count-1[17985:10b] 2
2008-07-20 11:54:20.466 03.01 Count-1[17985:10b] 3
2008-07-20 11:54:20.467 03.01 Count-1[17985:10b] 4
2008-07-20 11:54:20.467 03.01 Count-1[17985:10b] 5
```

现在，假设你想更新这个程序，使其输出数字1到10。你必须编辑如下代码清单中以粗体显示的两处代码，然后重新生成这个程序（这个版本的程序在文件夹03.02 Count-2中）：

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

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

```

{
    NSLog (@"The numbers from 1 to 10:");

    int i;
    for (i = 1; i <= 10; i++) {
        NSLog (@"%d\n", i);
    }
    return (0);
} // main

```

Count-2产生以下输出:

```

2008-07-20 11:55:35.909 03.02 Count-2[18001:10b] The numbers from 1 to 10:
2008-07-20 11:55:35.926 03.02 Count-2[18001:10b] 1
2008-07-20 11:55:35.927 03.02 Count-2[18001:10b] 2
2008-07-20 11:55:35.928 03.02 Count-2[18001:10b] 3
2008-07-20 11:55:35.935 03.02 Count-2[18001:10b] 4
2008-07-20 11:55:35.936 03.02 Count-2[18001:10b] 5
2008-07-20 11:55:35.936 03.02 Count-2[18001:10b] 6
2008-07-20 11:55:35.939 03.02 Count-2[18001:10b] 7
2008-07-20 11:55:35.939 03.02 Count-2[18001:10b] 8
2008-07-20 11:55:35.940 03.02 Count-2[18001:10b] 9
2008-07-20 11:55:35.940 03.02 Count-2[18001:10b] 10

```

这样修改程序显然不需要太多技巧，你可以用简单的搜索替换操作来完成，而且只需改变两处。然而，在比较大（如成千上万行代码）的程序中，执行搜索和替换就麻烦多了。仅仅是将5替换为10，我们也必须小心：毫无疑问，在有些情况下数字5是与此无关的，所以不应该改为10。

解决这个问题就是变量的目的。不必直接在代码中修改上限循环值（5或10），我们可以将这两个数字放到变量中，于是添加一个间接层，这样就能够解决问题。添加变量后，就是告诉程序“去查看名为count的变量，它会说明进行多少次该循环”，而不是“执行5次循环”。现在，程序Count-3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);

    int i;
    for (i = 1; i <= count; i++) {
        NSLog (@"%d\n", i);
    }
    return (0);
} // main

```

该程序的输出应当不会让人感到意外：

```
2008-07-20 11:58:12.135 03.03 Count-3[18034:10b] The numbers from 1 to 5:
2008-07-20 11:58:12.144 03.03 Count-3[18034:10b] 1
2008-07-20 11:58:12.144 03.03 Count-3[18034:10b] 2
2008-07-20 11:58:12.145 03.03 Count-3[18034:10b] 3
2008-07-20 11:58:12.146 03.03 Count-3[18034:10b] 4
2008-07-20 11:58:12.151 03.03 Count-3[18034:10b] 5
```

说明 `NSLog()`时间戳和其他信息会占用大量空间，为简明起见，以后的代码清单中将省略该信息。

如果想输出数字1~100，只需修改代码中很明显的一个地方：

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

int main (int argc, const char * argv[])
{
    int count = 100;

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

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

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

通过添加变量，代码现在更加干净，并且更易于扩展（特别是在其他编程人员需要修改此代码时）。为了修改循环值，他们不必仔细查看程序中使用的每个数字5，以确定是否需要修改，而是只需修改`count`变量就可获得期望的结果。

3.1.2 使用文件名的间接

文件是另一种间接的示例。请看Word-Length-1，此程序可输出单词及其长度，该程序位于03.04 Word-Length-1文件夹中。这个重要程序是新Web 2.0公司站点Length-o-words.com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;

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

return (0);
}

```

for循环可在任何时候确定要处理words数组中的哪个单词。循环内的NSLog()函数使用%s格式说明符来输出单词。之所以使用%s，是因为words是C字符串数组，而不是@"NSString"对象。%d格式说明符取strlen()函数的整数值，此函数计算字符串的长度，并输出单词本身及其长度。

运行Word-Length-1后，可看到如下输出信息：

```

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使用另一组单词。他们审查了你的业务计划，结论是如果你使用乡村音乐明星的名字，将有更广阔的市场。

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名字。在编辑时，必须注意标点符号，例如Joe Bob的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;

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

```

```

    }
    return (0);
} // main

```

因为我们很小心地修改了代码，所以程序仍然会如预期工作：

```

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

```

进行这种修改确实需要太多的工作：我们必须编辑Word-Length-2.m，解决所有录入错误，然后重新生成程序。如果程序在网站上运行，我们还必须重新测试和部署程序，以升级至Word-Length-2。

构造此程序的另一种方法是将名字完全移到代码之外，将其全部置于文本文件中，每行一个名字。大家一起说出来：“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。此程序从一个文本文件中读取一列名字，再输出名字及其长度。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] = '\0';

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

    fclose (wordFile);

    return (0);
} // main

```

浏览一下Word-Length-3，看看它的作用。首先，`fopen()`打开`words.txt`文件并读取文件内容。然后，`fgets()`从文件中读取一行文本并将其放入`word`中。`fgets()`调用会保留分开每一行的换行符，但我们不需要这个换行符，因为如果留下它，换行符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 看看 `fopen()` 使用的路径名称 `/tmp/words.txt`。这表示 `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` 来存储需要使用但不必保存的临时文件。在真实的程序中，应该将文件放在可永久存放的位置，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 `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

3

要通过文本编辑器将文件保存到 `/tmp` 目录中，请先输入文件的内容，然后选择 `Save`，按键盘上的斜杠键 (`/`)，输入 `tmp`，再按回车键。

如果愿意，你可以不输入名字，而是从 03.06 Word-Length-3 目录下将 `words.txt` 复制到 `/tmp` 中。要在 Finder 中查看 `/tmp`，请选择 `Go > Go to Folder`。

提示 如果你使用我们预生成的 Word-Length-3 项目，需要知道我们使用了一些 Xcode 技巧来帮助你将 `words.txt` 文件复制到 `/tmp` 中。看看你是否能发现我们是怎样做的。提示：查看 Groups & Files 窗格中的 Targets 区域。

运行 Word-Length-3，程序的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，我们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。请尝试一下向 `words.txt` 文件中添加数个单词，然后重新运行程序。我们等你完成这个任务。

这种方法要更好一些，因为文本文件易于编辑，而且远不像源代码那样容易被破坏。你能够让非编程人员朋友使用TextEdit 来执行编辑工作。市场营销人员能保证单词列表更新，让你可以继续从事更有趣的工作。

如你所知，在升级或改进程序方面人们总是有新想法。也许你的投资者又认为计算烹调术语的长度是盈利新途径。既然你的程序从文件中查找数据，那么你可以随意修改单词而不用修改代码。

尽管间接有很多优势，但 Word-Length-3 仍然相当脆弱，因为它一直使用单词文件的完整路径名。文件本身就处于不可靠的位置，因为如果计算机重启，`/tmp/words.txt` 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，他们可能会无意中覆盖你的文件。你可以每次都编辑程序来使用不同路径，但我们已经知道这样做很无趣，所以让我们使用

另一种间接技巧来简化我们的生活。

不在/tmp/words.txt中获取单词，我们修改程序，使其查看程序的第一个启动参数，确定单词文件的位置。下面是Word-Length-4程序（可在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。我们突出显示了对Word-Length-3所做的修改：

```
#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] = '\0';

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

    fclose (wordFile);

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

用来处理文件的循环与Word-Length-3中的相同，但是建立循环的代码是全新的且经过改进的。**if**语句验证用户是否提供了路径名作为启动参数。代码测试**main()**的**argc**参数，此参数保存启动参数的数目。因为程序名常用作启动参数传递，所以**argc**值常为1或更大。如果用户不提供文件路径，那么**argc**值为1，不能读取文件，于是我们输出错误消息并终止程序。

如果细心的用户提供了文件路径，那么**argc**将大于1。然后我们可查看**argv**数组来获知文件路径。**argv[1]**保存着用户提供的文件名（为满足你的好奇心，可以告诉你**argv[0]**参数保存程序名）。

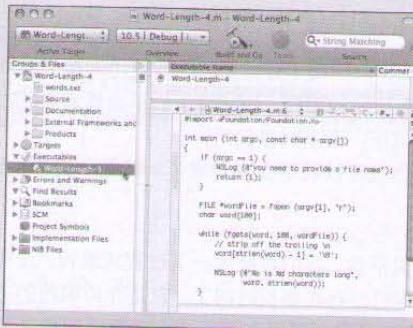
如果你正在Terminal中运行程序，那么很容易在命令行中指定文件名，如下所示：

```
$ ./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
```

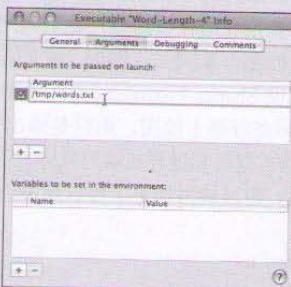
在Xcode中提供文件路径

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稍复杂些。启动参数也称为命令行参数，在Xcode中控制它比在Terminal中要更困难一点。修改启动参数所需的步骤如下。

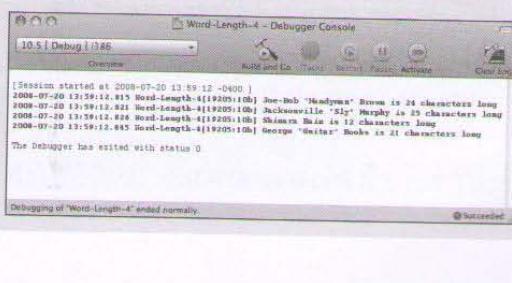
首先，在Xcode文件列表中展开Executables，并双击程序名（Word-Length-4），如下面的截图所示：



接下来，如下图所示，单击Arguments区域下的加号，并输入启动参数——在本例中是words.txt文件的路径：



现在，运行该程序，Xcode将启动参数传递给Word-Length-4的argv数组。下面是运行该程序时所显示的内容：



为了试验，你可以使用/usr/share/dict/words路径来运行程序，此文件中有二十三万多个单词。你的程序可以处理巨量数据！当你厌烦了观看单词在Xcode控制台窗口中呼啸而过时，可以单击红色的停止符号来停止程序。

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任意一组单词的长度，即使那组单词多得惊人也可以。当然，用户可以修改数据而不必修改代码。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 在面向对象的编程中使用间接

在OOP（面向对象编程）中，间接十分重要。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真正的革命性就是它在调用代码中使用间接。比如在调用函数时，不是直接调用，而是间接调用。

只要把握这一点，那么你就可以说是OOP专家了。其他一切都是这种间接的副产品。

3.2.1 过程式编程

为让你全面了解OOP的灵活性，我们将概述一下程式式编程，这样你就能明白创造OOP是为了解决哪种问题。程式式编程已经存在很长时间了，它是介绍编程的书籍和教程都要讲解的典型内容。使用诸如BASIC、C、Tcl和Perl等语言进行的大多数编程都是程式式的。

在程式式程序中，数据通常保存在简单的结构（例如C的**struct**元素）中。还有一些较为复杂的数据结构，例如链表和链树。当你调用一个函数时，你将数据传递给函数，而函数处理该数据。函数是程式式编程的中心：你决定使用什么函数，然后调用那些函数，传递其需要的数据。

假设有一个程序用来在屏幕上绘制一系列几何图形。感谢强大的计算机，你可以实现这一程序，而不仅仅是构思，你可以在03.08 Shapes-Procedural文件夹中找到该程序的源代码。为简明起见，Shapes-Procedural程序不是实际在屏幕上绘图，而只是输出一些与形状相关的文本。

Shapes-Procedural使用的是普通的C语言和程式式编程风格。代码的开始要定义一些常量和结构。

在强制包含的基础头文件之后，通过枚举指定了可以绘制的几种不同形状：圆形、方形和不规则的椭圆形。

```
#import <Foundation/Foundation.h>
typedef enum {
    kCircle,
    kRectangle,
    kOblateSpheroid
} ShapeType;

下面的enum定义了绘制形状时可用的颜色:
typedef enum {
    kRedColor,
    kGreenColor,
    kBlueColor
} ShapeColor;
```

然后，我们使用一个结构来描述一个矩形，此矩形指定屏幕上绘制形状的区域：

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

最后，我们用一个结构将所有内容结合起来，描绘一个形状：

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

在我们的例子中，`main()` 声明我们要绘制的形状的数组。声明数组之后，数组中的每个形状结构都通过分配字段而被初始化。下面的代码将产生一个红色的圆形、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 = kOblateSpheroid;
    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()`方法中的矩形是使用C语言方便的小技巧声明的：声明结构变量时，你可以一次性初始化该结构的所有元素。

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

结构元素按照声明的顺序取值。还记得`ShapeRect`吗？它是这样声明的：

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

上面对rect0的赋值表示rect0.x和rect0.y的值都为0, rect0.width为10, rect0.height为30。

此技巧可以减少程序中需输入的字符量, 并且不会牺牲可读性。

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

drawShapes()的循环先检查数组中的每个Shape结构, 再通过switch语句查看结构的type字段, 并选择绘制形状的函数。然后调用适当的绘图函数, 传递绘图屏幕区域和颜色的参数。其代码如下:

```
void drawShapes (Shape shapes[], int count)
{
    int i;

    for (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 kOblateSpheroid:
                drawEgg (shapes[i].bounds,
                         shapes[i].fillColor);
                break;
        }
    }
}
```

} // drawShapes

下面是drawCircle()的代码, 此函数仅输出有边框的矩形和传递给它的颜色:

```
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，例如 @"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，只不过它们绘制出的是矩形和椭圆形。

下面是Shapes-Procedural的输出（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
```

这一切十分简单明了，对不对？在使用过程式编程时，你的时间花在了连接数据与处理该类函数上。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()而不是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传递给准备处理圆的函数。

编写这种代码的另一个问题是程序的扩展和维护变得很困难。为了说明这点，让我们增强Shapes-Procedural，为之添加一种新形状：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中增加kTriangle常量：

```
typedef enum {
    kCircle,
    kRectangle,
    kOblateSpheroid,
    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

下一步，需要在drawShapes()的switch语句中增加一个新的case，用于测试kTriangle，并在测试通过时调用drawTriangle():

```
void drawShapes (Shape shapes[], int count)
{
    int i;

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

// drawShapes

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

```
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 = kOblateSpheroid;
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

```

扩展该程序，使其支持三角形并不难，不过我们的小程序仅用于实现一种操作——绘制形状。但程序越复杂，扩展起来就会越麻烦。例如，如果程序不仅仅是用于绘制各种形状，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，修改过去能正常工作的代码，并且可能会因此而引入一些错误。

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它。例如，绘制圆角矩形需要知道矩形的范围和圆形拐角的半径。为了支持绘制圆角矩形，你可以在Shape结构中增加半径域，但这会导致空间的浪费，因为绘制其他形状时无需使用半径域。或者，你也可以使用C并集来覆盖相同结构中不同的数据布局。但是，把各种形状融入并集中并获取有用数据的过程也会使问题复杂化。

OOP完美地解决了这些问题。当我们讨论在程序中使用OOP时，你会看到OOP如何解决第一个问题——修改现有的工作代码来增加新的形状。

3.2.2 实现面向对象编程

过程式程序建立在函数之上，数据为函数服务。面向对象编程从相反的角度来看待问题，它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，但它如何工作呢？在OOP中，数据通过间接方式包含对自身操作的引用代码。不是通知`drawRectangle()`函数“使用这个形状结构绘制矩形”，而是要求矩形“绘制自身”（天哪，听起来太荒谬了，但事实上却并不荒谬）。通过间接方式的威力，矩形数据知道如何查找相应的函数绘制图形。

那么，对象到底是什么呢？它其实是一种神奇的C struct。通常，该结构能通过函数指针查找与之相关的代码。图3-1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。

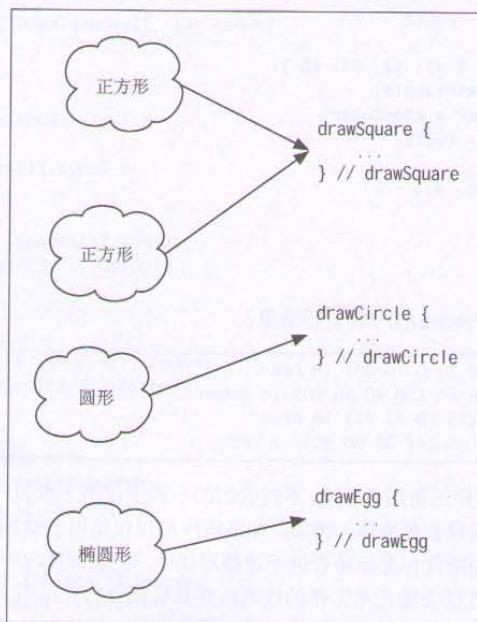


图3-1 基本形状对象

每个对象都有自己的`draw()`函数，知道如何绘制特定的形状。例如，`Circle`对象的`draw()`函数知道如何绘制圆形，`Rectangle`的`draw()`函数知道绘制由4条直线构成的矩形。

Shapes-Object程序（代码请参考03.10-Shapes-Object）可以完成与Shapes-Procedural相同的功能，但前者使用Objective-C的面向对象特性来实现。以下是Shapes-Object的`drawShapes()`代码：

```

void drawShapes (id shapes[], int count)
{
    int i;

    for (i = 0; i < count; i++) {
        id shape = shapes[i];
    }
}
  
```

```

    [shape draw];
}

} // drawShapes;

```

该函数包含一个循环，用于输出数组的每种形状。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，本函数更简短！代码不必询问每个形状的种类。

另一个不同点是函数的第一个参数`shapes[]`，此刻它是一个`id`数组对象。什么是`id`？它是与大脑有关的心理学术语，用于描述天生的本能冲动和原始过程吗？在本例中并非表示此意，它代表`identifier`（标识符）。`id`是一种泛型，用于表示任何种类的对象。回忆一下，对象是带有代码的`struct`。因此，`id`实际上是一个指针，指向其中的某个结构。在本例中，结构由各种形状构成。

`drawShapes()`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），并将其实值给名为`shape`（它具有类型`id`）的变量。这只不过是一种指针赋值过程，它实质上并不会复制`shape`的全部内容。看看图3-2，注意Shapes-Object中各种可用的形状。`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]`是个指向蓝色椭圆形的指针。

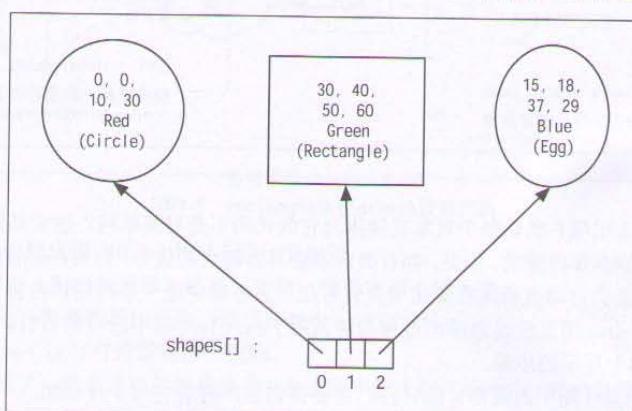


图3-2 `shapes`数组

现在，我们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使用方括号引用数组元素，但在此我们并没有使用数组实现任何功能。在Objective-C中，方括号还有其他意义：它们用于通知某个对象该做什么。在方括号内，第一项是对象，其余部分是你需要对象执行的操作。在本例中，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（有些人也将其称为“调用方法”）。代码[shape draw]表示向shape对象发送draw消息。[shape draw]可以理解成“向shape发送draw”。至于形状如何实际绘制图形，则取决于shape的实现。

向对象发送消息时，如何调用必要的代码呢？这是通过幕后名为类的帮手来协助完成的。

请看图3-3。该图的左侧展示了shapes数组中索引为0的circle对象，该对象最近在图3-2中出现过。circle对象含有一个指向其类的指针。类是一种结构，用于描述该种类对象的构造。在图3-3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必要功能的代码。

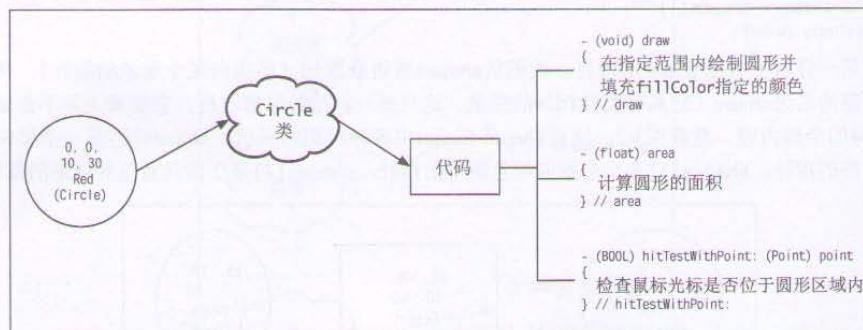


图3-3 circle和它的类

类对象有什么用呢？就让每个对象直接指向它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该内容）。

图3-4展示了draw消息经过怎样的过程最终调用了circle对象中适当的函数。

以下是图3-4中展示的步骤。

- (1) 对象是消息（图中的圆形）的目标，需要查询它，看看它属于什么类。
- (2) Circle类浏览其代码，查找draw函数的位置。
- (3) 找到draw函数后，将执行绘制圆形的函数。

图3-5展示了基于数组中的第二个形状（它是一个绿色的矩形）调用[shape draw]时的情况。

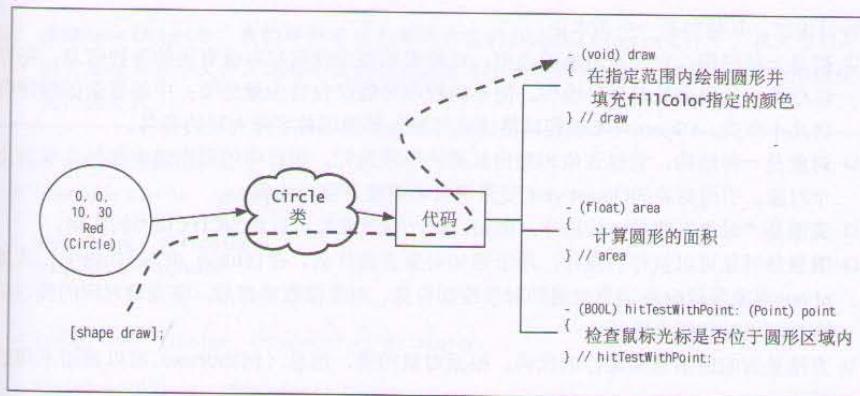


图3-4 circle查找draw函数的代码

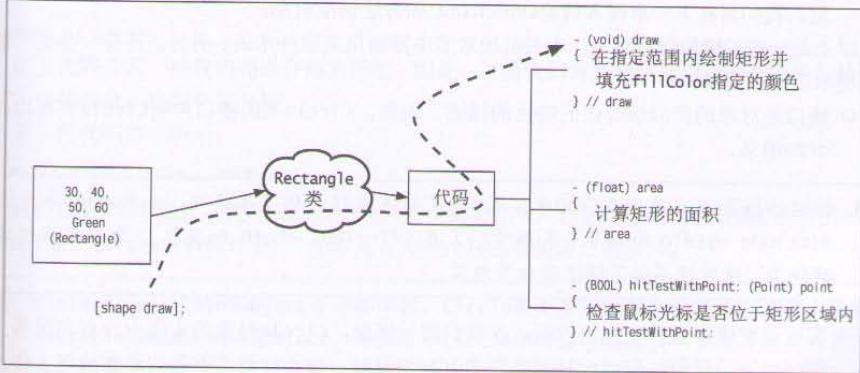


图3-5 rectangle查找draw函数的代码

图3-5中使用的步骤和图3-4中的步骤几乎相同。

- (1) 查询消息（图中的矩形）的目标对象，看看它属于什么类。
- (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在幕后作出决定，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，同时使程序代码更易于维护。

3.3 学习有关的术语

深入研究其余的Shapes-Object程序之前，先介绍一些有关面向对象的术语。其中有些术语我

们已经讨论过，其他的是一些新术语。

- **类**是一种结构，它表示对象的类型。对象引用类来获取和本身有关的各种信息，特别是运行什么代码来处理每种操作。简单的程序可能仅包含少量的类；中等复杂的程序会包含几十个类。Objective-C编程风格建议开发人员使用首字母大写的类名。
 - **对象**是一种结构，它包含值和指向其类的隐藏指针。运行中的程序通常都包含成百上千个对象。引用对象的Objective-C变量通常不需要首字母大写。
 - **实例**是“对象”的另一种称呼。例如，`circle`对象也可以称为`Circle`类的实例。
 - **消息**是对象可以执行的操作，用于通知对象去做什么。在`[shape draw]`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，将查询对应的类，以便查找正确的代码来运行。
 - **方法**是为响应消息而运行的代码。根据对象的类，消息（例如`draw`）可以调用不同的方法。
 - **方法调度程序**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，本书后续章节中需要用到这些术语。另外，还有一些更一般化的编程术语也很重要：
- **接口**是对象的类应该提供的特性的描述。例如，`Circle`类的接口声明`Circle`类可以接受`draw`消息。

说明 接口的概念不只是用在OOP中。例如，C的头文件提供了库接口，如标准I/O库（通过`#include <stdio.h>`获取）和数学库（通过`#include <math.h>`获取）。接口不提供实现的细节，通常你不必了解这些细节内容。

- **实现**是使接口正常工作的代码。在我们的示例中，`circle`对象的实现中含有在屏幕上绘制`circle`的代码。向`circle`对象发送`draw`消息时，你不知道或不关心函数如何工作，只要知道它可在屏幕上绘制一个圆形即可。

3.4 Objective-C 中的 OOP

如果你现在开始有些头痛了，这很正常。因为我们向你的头脑中灌输了许多新知识，消化吸收所有术语和技术需要一定的时间。在你的潜意识正在吸收前两节中内容的同时，接下来看一看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。特别地，它必须知道对象的数据成员（即对象的C `struct`应该是什么样子）和它提供的特性。可以使用`@interface`指令把这种信息传递给编译器。

说明 在Shapes-Object中，我们将所有信息都放在它的Shapes-Object.m文件中。在大型程序中，则需要使用多个文件，把每个类的信息放在各自的文件中。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
{
    ShapeColor fillColor;
    ShapeRect bounds;
}

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

#end // Circle
```

3

以上代码含有一些我们尚未介绍的语法，因此，下面我们将介绍这些新语法。这几行代码包含了大量的信息，我们分别分析。

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，你都可以把它看成是C语言的扩展。

`#interface Circle`告诉编译器：“这是为名为Circle的新类定义的接口。”

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都将继承NSObject类定义的所有行为。我们将在下一章详细介绍继承的有关内容。

声明完新类后，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。它表明，创建新Circle对象时，该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，是圆形的边界矩形，属于ShapeRect类型。该矩形用于确定在屏幕上的何处绘制圆形。

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`类对象的实例变量。

结尾处的花括号告诉编译器，我们为`Circle`指定了实例变量。

随后的代码行看起来像是C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。它们看起来很像是旧式的C函数原型，用于说明“这是我支持的特性”。方法声明指出每种方法的名称、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。这是一种区分函数原型与方法声明的方式，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`方法仅用来绘制图形，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、浮点型和字符型）、指针、对象引用和结构。

下一个方法声明更有意思：

```
- (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。圆形在绘制自身时使用该颜色。`setBounds:`有一个矩形参数，圆形使用该矩形定义它们的边界。

熟悉中缀符

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 setValue: @"hello there"
color: kBlueColor];
```

`setValue:`和`color:`实际上是参数的名称（实际上是方法名称的一部分，后面再详细介绍），`@"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);
```

我们确实喜欢中缀语法，虽然它乍一看有些古怪。它使代码的可读性更强，更容易理解参数的用途。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类型开头：

`- (void)`

与`draw`方法一样，此处的先行短线表明“这是新方法的声明”。`(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`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;`

在不含有参数的方法结尾乱添冒号是许多初级Objective-C编程人员常犯的错误。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，`setBounds:`和`setFillColor:`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`@end`语句后添加注释来说明类名。如果直接查看文件的结尾处，或者位于长文件输出的最后一页时，通过注释我们可轻松了解正在看什么。

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。一个方法设置边界，一个方法设置颜色，第三个方法绘制形状。

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。

你该不会认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”中3个首字母的缩写。使对象真正起作用的代码位于@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

- (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
```

现在，我们将按照惯例详细解释这些代码。Circle的实现从以下代码开始：

```
@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中无相应声明的方法。可以把它们看成是私有方法，仅在类的实现中使用。

说明 你也许认为，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。

```
setFillColor: is the first method defined:
- (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。二者间的主要差别是结尾处没有分号。也许你已经注意到，我们把参数重新命名为简单的 `c` 字符了。`@interface` 和 `@implementation` 间的参数名不同是正确的。在这里，如果我们继续使用参数名 `fillColor`，就会隐藏 `fillColor` 实例变量，并且编译器会生成警告信息。

3

说明 为何一定要重新命名 `fillColor` 呢？我们已经通过类定义了一个名为 `fillColor` 的实例变量，可以在该方法中引用该变量——它在作用域范围内。因此，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隐藏初始变量，可以为参数使用新的名称来避免该问题。例如，可以将实例变量改其他名称（如 `myFillColor`），这样就可以继续把 `fillColor` 作为参数名称。在后面第 16 章中你将看到，如果我们为实例变量起一个和方法名类似的名字，Cocoa 还可以发挥某些神奇威力。

`@interface` 部分的方法声明中使用了名称 `fillColor`，是为了确切告诉读者参数的用处。在实现中，我们必须区分参数名称和实例变量名称，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;
```

如果你特别好奇，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` 的传递过程是秘密的和自动的，因此你不必自己来实现。方法中引用实例变量的代码如下所示：

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

顺便提一下，传递隐藏的参数是另一种间接操作的示例。（你可能认为我们已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，所以哪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 用户运行应用程序时，Objective-C 运行时是支持这些应用程序（包括我们自己的应用程序）的代码块。运行时执行非常重要的任务，如向对象发送消息和传递参数。我们将从第 9 章开始介绍更多有关运行时的内容。

第二个方法 `setBounds:` 类似于我们的 `setFillColor:` 方法：

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

```
    bounds = b;
} // setBounds
```

以上代码用于设置圆形对象的边界矩形，圆形对象将被绘制在该矩形中。

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()`输出所有文本。

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

3.4.3 实例化对象

现在，我们介绍Shapes-Object最后的、非常关键的过程，在该过程中，我们创建可爱的形状对象，例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，然后这些内存被初始化并保存一些有用的默认值，这些值不同于你在获得新分配的内存时得到的随机值。内存分配和初始化完成后，就创建了一个新的对象实例。

说明 由于对象的局部变量特定于该对象的实例，因此我们称它们为实例变量，通常简写为`ivars`。

为了创建新对象，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向类发送消息。这种便捷的行为不局限于某个特定的对象，而是对全体类都通用。这种消息通常用在创建新对象时。如果需要创建新的`circle`对象，请求`Circle`类创建新对象比请求某个现有的`circle`对象更合适一些。

以下是Shapes-Object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] = [OblateSphereoid new];
[shapes[2] setBounds: rect2];
[shapes[2] setFillColor: kBlueColor];

drawShapes (shapes, 3);

return (0);

} // main

```

3

可以看到，Shapes-Object的main()函数与Shapes-Procedural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（也许你还记得，它是指向某种对象的指针），而不是shapes数组。通过向需要创建对象的类发送new消息，可以创建各个对象：

```

shapes[0] = [Circle new];
...
shapes[1] = [Rectangle new];
...
shapes[2] = [OblateSphereoid new];
...

```

另一个不同之处是，Shapes-Procedural通过直接分配struct成员来初始化对象，但Shapes-Object并不直接与对象混在一起。相反，Shapes-Object使用消息请求每个对象设置它的边界矩形和填充颜色：

```

...
[shapes[0] setBounds: rect0];
[shapes[0] setFillColor: kRedColor];
...
[shapes[1] setBounds: rect1];
[shapes[1] setFillColor: kGreenColor];
...
[shapes[2] setBounds: rect2];
[shapes[2] setFillColor: 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

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。这次任务应该简单多了。你可以在Learn ObjC Projects文件夹的03.11 Shapes-Object-2中查找该项目对应的程序。

为了在Shapes-Procedural-2中增加绘制三角形的功能，我们必须做很多工作：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。其中很多工作极具挑战性，特别是对drawShapes()函数的修改，我们必须编辑循环来控制对所有形状的绘制，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：

```
@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 剪切和粘贴编程方式（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。

3

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，以便有足够的空间存储新对象：

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

然后，添加可创建新Triangle的代码块，这和创建新Rectangle和Circle一样：

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

最后，用shapes数组的新长度重新调用drawShapes()：

```
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OOP的功劳。

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新的形状对象类型。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/关闭原则的软件会表现出更强的健壮性，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。我们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了间接技术（如变量和文件的使用）。然后，我们讨论了过程式编程，并指出了“函数第一，数据第二”这种观念所导致的局限性。

接着介绍了面向对象编程，这种编程方法使用间接技术将数据和对数据执行的操作紧密联系在一起，从而引入了“数据第一，函数第二”的编程风格。我们讨论了消息（它们被发送给对象），

通过执行方法，对象处理这些消息，代码段使对象产生各种行为。另外，你还知道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，通过该特性，可以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！

编写面向对象的程序时（我们希望你编写大量的面向对象程序），你所创建的类和对象之间存在一定的关系。它们协同工作才能实现程序相应功能。

处理类和对象间的关系时，尤其要重视OOP的两个方面。第一个方面是继承，本章将讨论该主题。创建一个新类时，通常需要定义新类以区别于其他类及现有类。使用继承可以定义一个具有父类所有功能的新类，它继承了父类的这些功能。

另一个和类有关的OOP技术是复合（composition）。在复合中，对象可以引用其他对象。例如，在游戏过程中，赛车模拟程序中的汽车对象含有4个轮胎对象。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复合的有关内容。

4.1 为何使用继承

还记得上一章中我们的老朋友Shapes-Object程序吗？该程序包含几个接口和实现非常相似的类。当然，由于我们采用剪切和粘贴的方式构建了这些类，所以它们非常类似。

我们回忆一下Circle和Rectangle类的接口程序，代码如下所示：

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

- (void) setFillColor: (ShapeColor) fillColor;
- (void) setBounds: (ShapeRect) bounds;
- (void) draw;
@end // Circle

@interface Rectangle : NSObject
{
    ShapeColor fillColor;
    ShapeRect bounds;
}

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

```

- (void) setBounds: (ShapeRect) bounds;
- (void) draw;
@end // Rectangle

```

这些类接口非常类似，非常非常类似。事实上，除了类名不同，其他方面都是相同的。

另外，`Circle`和`Rectangle`的实现也非常类似。回忆一下，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

```

这些方法实现了完全相同的功能——设置`fillColor`和`bounds`实例变量。但是，`Circle`和`Rectangle`的实现方式并不相同。例如，`draw`方法的签名，该方法的名称和参数在两个类中都相同，但实现方式不同：

```

@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

```

4

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是类的关系图。

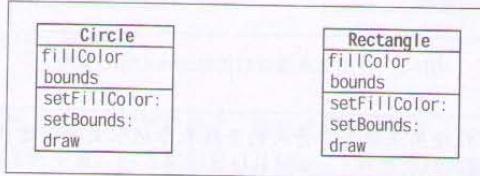


图4-1 没有继承的Shapes-Object架构

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是一种用图表表示类、类的内容以及它们之间关系的常见方式。

图4-1中含有大量的重复的内容，这会影响程序的执行效率。进行编程时，出现这样的重复内容就表明它是一个不好的架构。你需要维护两倍的代码，修改代码时，必须在两处（或更多处）进行修改，这将大大增加出错的可能性。如果你忘记更改某处的代码，就有可能出现一些奇怪的bug。

如果能将所有重复的内容在一个地方统一就太好了！如果我们还能在需要的地方添加自定义方法，那就更好了，例如在我们必须画圆圈和矩形时。我们需要一种系统，能够通知编译器“Circle类与其他类一样，只是对某些地方进行了一些调整”。嗯，你可能已经猜到，继承恰好是实现该功能的强大OOP特性。

图4-2展示了实施某些继承后，我们的架构看起来是什么样。我们已经创建了一个全新的类

Shape, 它用于保存公共实例变量和声明方法。类Shape还包含`setFillColor:`和`setBounds:`的实现。

查看(图4-2中)中的新Circle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和Rectangle类中仅留下那些特有的元素, 特别是`draw`方法。现在, 我们可以说Circle和Rectangle都是从Shape类继承而来。

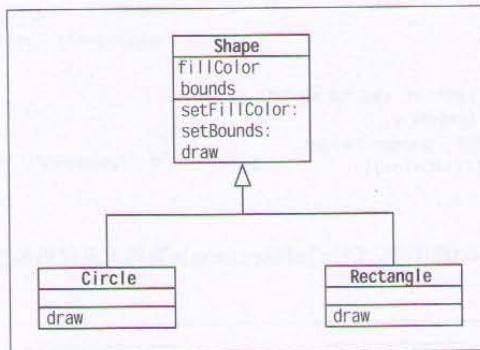


图4-2 使用继承改进后的Shapes-Object架构

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。

正如你从亲生父母那继承一些特性(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)中获取了某些特性。由于Circle和Rectangle是从Shape类继承而来, 因此它们将获得Shape类的两个实例变量。

说明 不要直接更改由继承得到的实例变量的值。一定要使用方法来更改它们。

除实例变量之外, 继承还会引入一些方法。所有Circle和Rectangle类都知道如何响应`setFillColor:`和`setBounds:`方法, 因为它们从Shape类中继承了该功能。

4.2 继承语法

看一下用于声明新类的语法:

```
@interface Circle : NSObject
```

冒号后的标识符是需要继承的类。在Objective-C中, 可以从非类中继承对象, 但如果使用Cocoa, 会希望从NSObject继承对象, 因为NSObject提供了大量有用的特性(当继承一个已继承自NSObject的类时, 你也能获取这些特性)。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

某些语言（例如C++）具有多继承特性，在这种情况下，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（多继承的形式可能类似于以下语句），编译器将不能正常识别它们：

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

你可以通过Objective-C的其他特性获取多继承的优点，例如分类（请参阅第12章）和协议（请参阅第13章）。

4

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，使我们的类由Shape类继承而来。将Circle和Rectang的接口代码更改为以下形式（可在04.01 Shapes-Inheritance中找到该程序的代码）：

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

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

你已经无法使代码比上面的形式更简单了。既然代码如此简单，bug就无处藏身了。

注意，不必再声明实例变量，因为我们可以从Shape中继承这些实例变量。你将会注意到，由于缺少实例变量，我们不再使用花括号（如果程序中没有变量，可以省略括号）。另外，也不必声明从Shape中获取的方法（setBounds:和setFillColor:）。

现在看一下Shape的功能代码。以下是Shape的声明：

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

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

可以看到，Shape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的实现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，我们仍然需要定义它，以便Shape的所有子类都能实现各自不同的方法。对于方法的定义，使用空正文或者返回一个虚（dummy）值都是可以的。

接下来介绍Circle的实现。你可能已经猜到，现在它的实现变得更简单：

```

@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

```

以下是新的、更简洁的Rectangle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

```

同样，Triangle和OblateSpheroid类的实现也很简单。详细内容请参见04.01 Shapes-Inheritance文件夹。

现在运行Shapes-Inheritance程序，可以看出，它的功能和前面完全相同。奇怪的是，我们并没有更改main()函数中设置和使用对象的任何代码。这是因为我们没有改变对象响应的方法，同时也没有修改它们的行为。

说明 这种移动和简化代码的方式称为重构。这在OOP社区中是一个非常时尚的话题。进行重构时，你通过移动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，而不改变代码的行为或运行结果一样。通常的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。

通常，在面向对象的程序中添加某些新特性后，程序反而变得更简单，你可能对此觉得很奇怪，这就像我们添加了Shapes类后所出现的情况。

学习有关术语

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

- **超类 (superclass)** 是你所继承的类。Circle的超类是Shape，Shape的超类是NSObject。
- **父类 (parentclass)** 是超类的另一种表达方式。例如，Shape是Rectangle的父类。
- **子类 (subclass)** 是实施继承的类。Circle是Shape的子类，而Shape又是NSObject的子类。
- **孩子类 (childclass)** 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。在实际中，你偶尔还能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，也许是因为我们比父母更另类一些。
- 改变方法的实现时，需要重写 (override) 继承方法。Circle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确保调用相应类的重写方法的实现。

4

4.3 继承的工作机制

我们对Shapes-Object程序做了较大调整，取出了Circle和Rectang的代码，并将其放入Shape中。令人兴奋的是，程序的其他部分仍然能正常工作，不必做任何修改。main()函数中创建和初始化所有不同形状的代码也没有改变，并且drawShapes()函数也相同，但程序仍然能正常工作：

- 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，你可以发现OOP另一个强大的方面：你可以从根本上改变程序，如果你非常仔细，完成这些改变后，程序仍然能正常工作。当然，你也可以在过程性编程中实现这种操作，但使用OOP成功的机会通常会更高一些。

4.3.1 方法调度

对象收到消息时，它们如何知道要运行哪些方法呢？例如，我们已经将setFillColor:的代码移出了Circle和Rectang类，当你向Circle对象发送setFillColor:方法时，Shape代码如何响应呢？秘密在于：当代码发送消息时，Objective-C的方法调度程序将在当前类中搜索相应的方法。如果调度程序无法在接收消息的对象类中找到相应的方法，它就在该对象的超类中进行查找。

图4-3使用前面老式的Shape程序版本，展示了方法调度如何通过代码向Circle对象发送setFillColor:消息。为了处理[shape setFillColor: kRedColor]之类的代码，Objective-C的方法调度程序将查找接收该消息的对象（在本例中是Circle类的对象）。该对象有一个指向Circle

类的指针，同时Circle类有一个指向其对应代码的指针。调度程序使用这些指针查找正确的代码来运行。

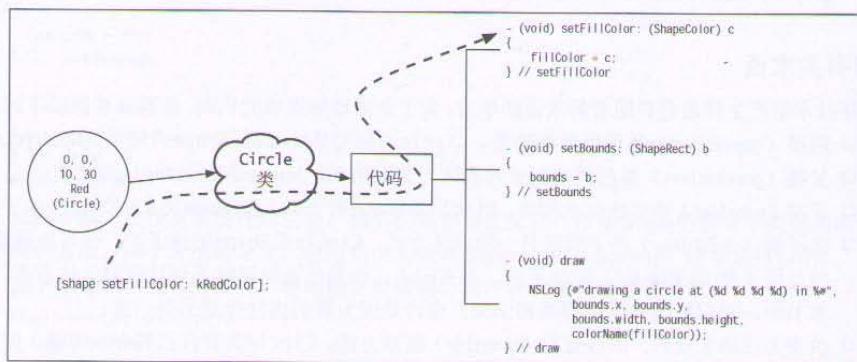


图4-3 无继承的方法调度

查看图4-4，它展示了可增强继承功能的时尚新结构。在该代码中，Circle类引用了其超类Shape。消息到来时，Objective-C方法调度程序使用该信息查找方法的正确实现。

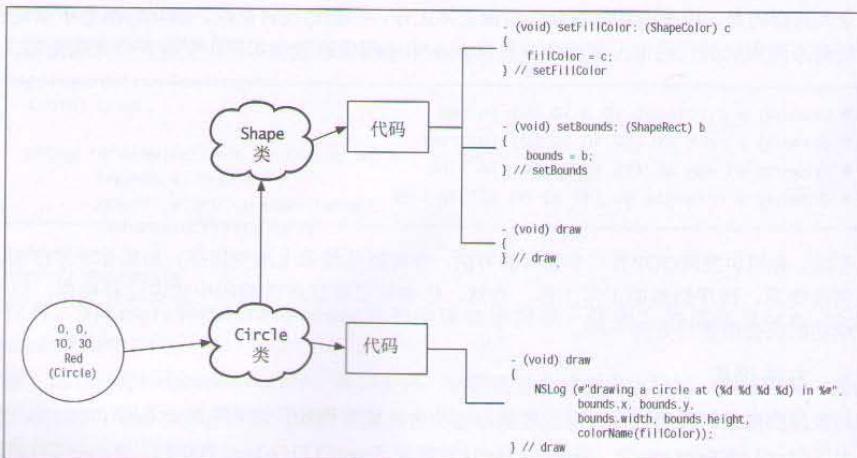
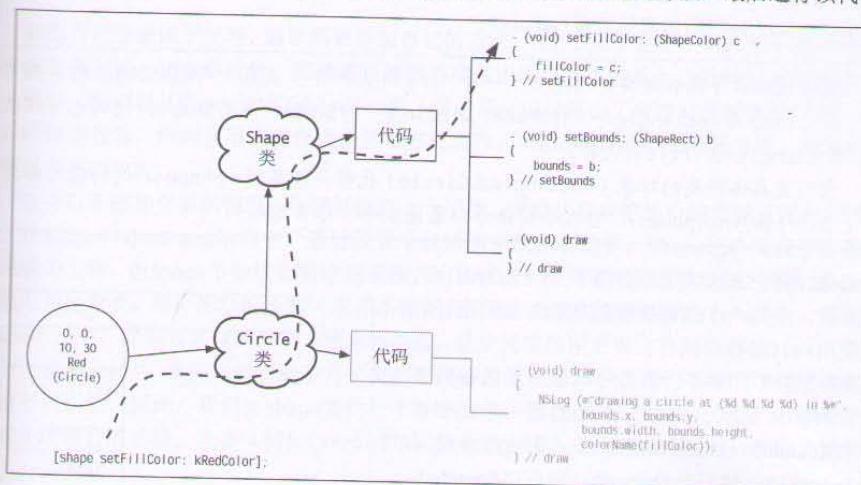


图4-4 继承和类代码

图4-5展示了使用继承功能的方法调度过程。向Circle对象发送setFillColor:消息时，调度程序首先询问Circle类是否使用自身的代码响应setFillColor:方法。在本例中，回答是不能，调度程序发现Circle类没有为setFillColor:定义方法，因此，它将在超类Shape中查找对应的

方法。然后，调度程序查找到Shape类根部，找到setFillColor:定义方法，最后运行该代码。



4

图4-5 使用继承的方法调度

“我在此没有找到它，接下来将在超类中进行查找”，必要时，这种操作将会在继承链的每个类中重复执行。如果在Circle和Shape类中都没有找到某种方法，调度程序将会检查NSObject类，因为它是继承链中的下一个超类。如果NSObject类（最高级别的超类）中也没有该方法，则会出现运行时错误（同时还会出现编译时警告信息）。

4.3.2 实例变量

上面我们讨论了如何调用方法来响应消息。接下来，我们将介绍Objective-C如何访问实例变量，以及Circle的draw方法如何查找Shape中声明的bounds和fillColor实例变量。

创建一个新类时，其对象首先从自身的超类中继承实例变量，然后（可选）添加它们自己的实例变量。为了了解实例变量的继承机制，我们创建一个新形状来添加新的实例变量。在绘制矩形的拐角时，新类RoundedRectangle需要某个变量来保存半径。RoundedRectangle类的定义如下所示：

```

@interface RoundedRectangle : Shape
{
    int radius;
}

@end // RoundedRectangle

```

图4-6展示了圆角矩形对象的内存布局。NSObject声明了一个名为isa的实例变量，该变量保存一个指针，指

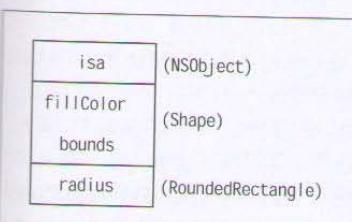


图4-6 对象实例变量的布局

向对象的类。接下来是由Shape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。

说明 因为继承在子类和超类之间建立了一种“is a”（是一个）关系，所以NSObject的实例变量称为isa。即Rectangle是一种Shape，Circle是一种Shape。使用Shape的代码也可以使用Rectangle或Circle来代替。

使用更具体种类的对象（Rectangle或Circle）代替一般类型（Shape），这种能力称为多态性（polymorphism），它在希腊语中形象地表示“很多形状”。

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。方法使用self参数查找它们要使用的实例变量。

图4-7展示了指向一个圆角矩形对象的self参数。self指向继承链中第一个类的第一个实例变量。对RoundedRectangle类而言，它的继承链从NSObject开始，然后是Shape类，最后以RoundedRectangle结束，因此，self指向第一个实例变量isa。因为Objective-C编译器已经看到了所有这些类的@interface声明，因此，它能知道对象中实例变量的布局。通过这些重要的信息，编译器可以产生代码并查找任何需要的实例变量。

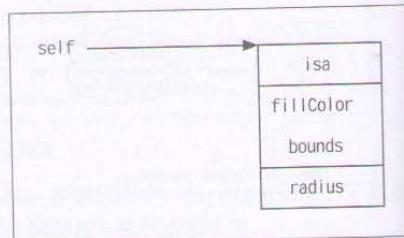


图4-7 指向circle对象的self参数

小心易碎

编译器使用“基地址加偏移”机制实现奇妙的功能。给定的对象地址，是指第一个实例变量的首个字节在内存中的位置。通过在该地址加上偏移地址，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个偏移地址之后，即位于0x1000 + 4位置，或写作0x1004。每个实例变量与对象的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存储self的位置值，然后加上偏移值（在本例中为4），得到指向存储变量值的位置。

随着时间的推移，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，但他们无法做到。因为这样做会改变所有实例变量的偏移位置。这被称为脆弱的基类问题（fragile base class problem）。通过在Leopard中引入新的64位Objective-C运行（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

制作自己全新的子类时，通常需要添加自己的方法。有时，为了在类中引入某个独特的特性，需要添加新方法。还有些时候，可能需要替换或增强由这个新类的某个超类所定义的现有方法。

例如，你可以从Cocoa的NSTableView类（用于显示用户单击了哪些对象的滚动列表）着手，然后添加新行为，例如使用语音合成器公布列表内容。可以添加speakRows新方法，向语音合成器提供表格的内容。

也可以不添加全新的特性，你可以创建一个子类，调整从自身的某个超类继承而来的现有行为。在Shapes-Inheritance示例中，通过设置形状的填充颜色和边界，Shape已经实现了多数我们要完成的工作，但Shape不知道如何绘制事物。并且它也无法知道如何进行绘制，因为Shape是一个通用的抽象类，每个形状的绘制方法都不相同。因此，当我们需要创建Circle类时，可以创建Shape的子类，并编写具体的绘制方法来绘制圆。

创建Shape时，我们知道它的所有子类都要用于绘制图形，尽管我们不知道这些子类如何实现图形的绘制。因此，我们为Shape提供一个draw方法，但使该方法的内容为空，这样每个子类就能实现各自的功能。当类（例如Circle和Rectang）实现各自的draw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，方法调度程序将运行重写的方法：Circle的draw实现。超类（如Shape）定义的所有draw实现都会被完全忽略。在此情况下这没有问题，因为Shape的draw实现中没有任何代码。但有时你可能不想忽略超类的方法。要了解更多相关内容，请继续阅读。

super关键字

Objective-C提供某种方式来重写方法，并且仍然调用超类的实现方式。当需要超类实现自身的功能，同时在前面或后面执行某些额外的工作时，这种机制非常有用。为了调用继承方法的实现，需要使用super作为方法调用的目标。

例如，假设我们正好获悉某些国家的文化忌讳红色的圆，同时我们希望在这些国家出售Shapes-Inheritance软件。与我们一贯使用的绘制红色圆的方法不同，我们希望所有的圆都被绘制为绿色。因为这种限制仅影响圆，所以一种方式是修改Circle类，将所有的圆都绘制为绿色。其他用红色绘制的形状没有问题，因此不必将它们删除。为什么不直接修改Circle的填充颜色方法呢？在此，我们可以这样做。但在其他场合未必能这样做，例如，你无法获取需要修改的类代码。

记住，setFillColor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。我们考虑颜色参数，如果它是红色，则把它改成绿色。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（本程序的完整代码清单请参见04.02 Shapes-Green-Circles文件夹）。

Circle的@interface部分没有变化，因为我们没有添加任何新方法或实例变量。只需向@implementation部分添加以下代码：

```
@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，我们查看`ShapeColor`的参数是否为红色，如果是，就将它改成绿色。然后，我们请求超类使用代码`[super setFillColor: c]`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编译器提供的某种神奇功能。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将按照通常的方式在继承链中继续查找对应的消息。

图4-8展示了Circle的setFillColor:的执行流程。setFillColor:消息发送给Circle对象。方法调度程序查找自定义的setFillColor:方法，这是由Circle类实现的。

Circle的setFillColor:方法检查kRedColor变量，并在必要时更改颜色后，通过调用[super setFillColor: c]来调用超类的方法。这样，super调用将运行Shape的setFillColor:方法。

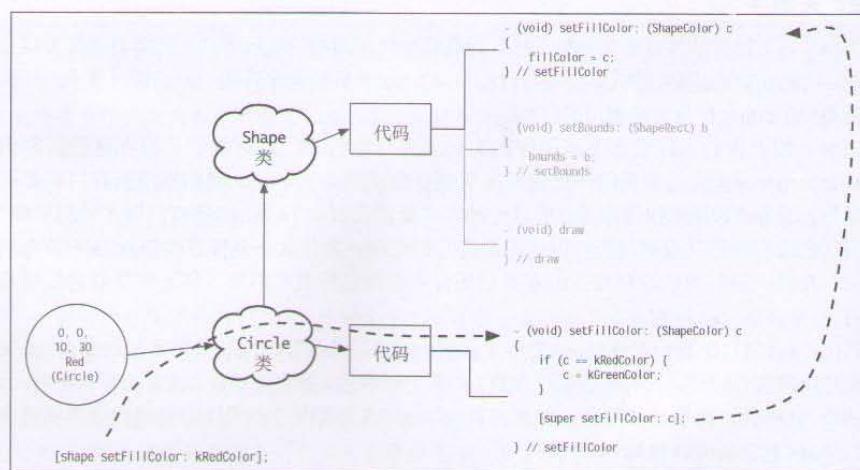


图4-8 调用超类的方法

说明 重写方法时，调用超类方法总是一个不错的选择，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，因此我们知道所有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。并且即使我们现在知道Shape的功能，但是，更改或改进类之后，我们可能无法了解它的功能。通过调用继承的方法，可以确保能够获得方法实现的所有特性。

4.5 小结

4

在OOP中，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，你学习了继承的有关概念，如何使用继承完善和简化Shapes-Object代码。我们讨论了如何通过现有类构造新的类，如何将超类的实例变量引用到子类中。

我们学习了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对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，但它非常重要，让我们下章再见吧。

通过上一章的学习，想必读者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。在上一章中，我们也提到过类之间的关系也可以通过复合的方式来建立，这就是本章的主题。使用复合可组合多个对象，使之分工协作。在实际的程序中，会用到同时使用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

编程中的复合就好像音乐中的作曲一样：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（独轮车）。

说明 在之前的Shapes-Object程序中，我们已经看到了一种形式的复合：Shape类使用了矩形（一个struct）和颜色（一个enum）。严格地讲，只有对象间的组合才能叫做复合。诸如int、float、enum和struct等基本类型都被认为是对象的一部分。

5.1.1 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，一辆汽车拥有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，而是用两个只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，发动机也会有马力和油耗等属性。这段程序的代码可在文件夹05.01 CarParts中找到。

与Shapes程序相同，CarParts的代码都包含在主程序mainCarParts.m中。首先，导入Foundation框架的头文件：

```
#import <Foundation/Foundation.h>
然后是Tire类，类里只有一个description方法。
@interface Tire : NSObject
@end // Tire
@implementation Tire
- (NSString *)description
{
    return @"I am a tire. I last a while";
} // description
@end // Tire
```

5

说明 如果类中没有包含任何实例变量，就可以省去上面代码中的花括号。

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，我们又怎么知道Tire类里用到了description呢？这时就要靠Cocoa神奇的帮助了。

5.1.2 自定义 NSLog()

记住，通过 NSLog() 可以使用 %@ 格式说明符来输出对象。 NSLog() 处理 %@ 说明符时，它会询问参数列表中相应的对象以得到这个对象的描述。从技术上讲，就是 NSLog() 给这个对象发送描述消息，然后对象的 description 方法生成一个 NSString 并返回。之后 NSLog() 在其输出中包含这个字符串。在类中添加 description 方法就可以自定义 NSLog() 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），也可以构造一个描述该对象所有此类信息的字符串，例如丹麦乳酪蛋糕的脂肪含量和卡路里。在 Cocoa 中，NSArray 类管理的是对象集合，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，对象的描述则是通过向数组中的对象分别发送 description 消息来获得。

让我们回到 CarParts 并看看 Engine 类。与 Tire 一样，它也包含了一个 description 方法。在真正的程序中，Engine 类会包含 start（启动）、accelerate（加速）等方法和 RPM（转数）等实例变量。但是在这里，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（汽车）本身，它拥有一个engine对象和一个由4个tires对象组成的C数组。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

```

engine和tires实例变量是复合的，因为它们是Car的实例变量。你可以说汽车是由4个轮胎和1台发动机组成的。但是通常我们都不这么说，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，而只是内存中存在的其他对象的引用。为新建的Car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还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。

下面让我们看看Car类的实现。首先是一个初始化实例变量的init方法。init方法创建了1个engine变量和4个tires变量，用以装配汽车。使用new创建新对象时，实际上系统要完成两个步骤。第一步，为对象分配内存，即对象获得一个用来存放其实例变量的内存块；第二步，自动调用init方法，让该对象处于可用状态。

```

@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个新轮胎并将其赋值给tires数组，接着又创建了一台发动机并将其赋值给engine实例变量。

接下来就是Car的print方法：

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

5

关于if语句

在init方法中，下面这行代码看起来有点奇怪：

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

下面我们来解释这行的意思。若要超类（这里是NSObject）可以完成所需的一次性初始化，需要调用[super init]。init方法返回的值（id型数据，即泛型对象指针）描述了被初始化的对象。

将[super init]的结果赋给self是Objective-C的标准惯例。这么做是为了防止超类在初始化过程中返回的对象不同于原先创建的对象。我们将在后面详细讲述init方法的章节里深入研究这个主题，所以现在，请微笑点头并且继续学习下面的内容。

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（抱歉最后才提到）。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，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.
```

这辆车不会赢得什么汽车大奖赛，但是它真的运行了！

5.2 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。程序里总是有一个或多个bug要去修正，总是有一些功能需要添加，总是有另外的方法能让程序的规模更大，机制更强壮，运行速度更快。所以毫无疑问，CarParts并不完美。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的init方法时可能会说：“为什么汽车要自己创建自己的轮胎和发动机呢？”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完善了。

如果我们可以为这辆汽车选择某种类型的轮胎和发动机，那该有多好。这样，用户就可以自己选择搭配汽车部件来定制属于他们自己的汽车了。

我们可以添加存取方法来实现上述想法。存取方法（accessor method）是用来读取或改变对象特定属性的方法。例如，Shapes-Object中的setFillColor就是一个存取方法。如果我们要添加一个新方法去改变Car对象中的发动机类，那么这个方法也是一个存取方法。这类存取方法称为**setter方法**，因为它为对象中的某属性赋值。术语**修改方法**（mutator）是用来改变对象状态的方法。

也许读者已经猜到了，另一种存取方法就是**getter方法**。getter方法为使用对象的代码提供了读取对象属性的途径。在赛车游戏中，物理逻辑程序可能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会不会在湿滑的道路上打滑。

说明 在对其他对象的属性进行操作时，应该始终使用对象所提供的存取方法，永远不要直接改变其他对象属性的数值。例如，main()函数不应直接访问Car的engine实例变量（使用car->engine）来改变发动机的属性，而应使用setter方法进行更改。

存取方法是程序间接工作的另一个例子。使用存取方法来间接地访问car的engine，可以让car的实现更灵活。

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控制权了。下面是Car的新接口，新添加的代码用粗体表示：

```
©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

```

5

代码中的实例变量并没有变，但是新增加了两对方法：`engine`和`setEngine:`用来处理发动机属性，`tireAtIndex:`和`setTire atIndex:`用来处理轮胎属性。存取方法总是成对出现的，一个设置属性的值，另一个读取属性的值。然而有时，只有一个`getter`方法（用于只读属性，例如硬盘上文件的大小）或者只有一个`setter`方法（例如设置密码）也是合理的。但在通常情况下，我们都会同时编写`setter`和`getter`方法。

在为存取方法命名时，Cocoa有自己的惯例。在你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`方法则仅仅根据其返回的属性的名称来命名。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`就破坏了命名惯例。有些语言（如Java）有不同的命名惯例，它们用“`get`”作存取方法的前缀。但是如果编写Cocoa程序，就不要这么做。

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通过你当作参数传入的指针来返回数值。例如，`NSData`（Cocoa中的类，创建可存储任意字节序列的对象）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`（用于绘图）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，以及指向存储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“`get`”，那么有经验的Cocoa编程人员就会想到将指针当成参数传入这个方法，当他们发现这不过是一个简单的存取方法时会觉得困惑。最好不要让其他编程人员被你的代码搅得一头雾水。

5.2.1 设置发动机的属性

第一对存取方法用来访问发动机的属性。

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

使用对象Car的代码调用engine方法来访问发动机，调用setEngine:方法来更改发动机的属性。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的值设为方法参数所指向的值。实际上被复制的并不是engine本身，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扔给你，你一定会觉得困惑并且感到沮丧。所以我们把如何完全正确地编写存取方法的讨论推到第8章。

要想实际使用这些存取方法，可编写与下面类似的代码：

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

 NSLog(@"%@", [car engine]);
```

5.2.2 设置轮胎的属性

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个角各有1个轮胎），所以对象Car有一个轮胎数组。我们用索引存取器来访问它们，而不是直接操作整个tires数组。为汽车配置某个轮胎时，不仅要知道使用哪个轮胎，还要清楚每个轮胎在汽车上的位置。同样，当访问汽车上某个轮胎时，也要指定这个轮胎的具体位置才可以。

下面是tire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 setTireAtIndex:",
               index);
        exit (1);
    }

    tires[index] = tire;
} // setTireAtIndex;

- (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实例变量tires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，这是种很好的编程思想。防御式编程能够在开发周期的早期发现错误，例如tires数组的非法索引。

由于tires是C型数组，而当访问C型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的有效性。像tires[-5]和tires[23]这样的写法编译器也可以编译通过。而虽然此数组只有4个元素，所以使用-5或23这样的索引将会访问随机内存，从而产生bug甚至导致程序崩溃。

检查索引之后，新的tire变量将会被放到tires数组中恰当的位置上。使用上述存取方法的示例如下所示：

```

Tire *tire = [Tire new];
[car setTire: tire

```

```

atIndex: 2];
NSLog (@"tire number two is %@", [car tireAtIndex: 2]);

```

5.2.3 跟踪汽车的变化

完成CarParts存取方法的讨论之前，还有一些细节问题需要提及。

首先是Car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。实际上，完全可以删除init方法，因为已经不需要在Car中做这些工作了。新车的车主将会得到一辆没有轮胎和发动机的汽车，但是轮胎和发动机也很容易配置安装（有时，软件中的生活比现实生活要容易得多）。

由于Car类不再创建自身的移动部件，所以必须更新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];

    int i;
    for (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，然后为其配置一个新的发动机并进行安装。接下来是一个循环4次的for循环，每次循环都创建一个轮胎并让汽车使用它。最后，描绘这辆新车，程序退出。

从用户的角度来看，程序并没有任何改变：

```

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

```

与Shapes-Object一样，我们重构了该程序，改进了程序的内部架构，但是并没有影响到它的外部行为。

5.3 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。我们不用现有的发动机和轮胎，而是对这些部件做一些改变。用继承方式来创建新的发动机和轮胎种类，然后使用Car类的存取方法（复合方式）给汽车配置新的移动部件。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ThreeFiftyOneWindsor，也可以用它们替代Slant6）。

```
@interface Slant6 : Engine
@end // Slant6

@implementation Slant6
- (NSString *) description
{
    return @"I am a slant-6. VR00OM!";
} // description

@end // Slant6
```

5

Slant6是发动机的一种，因此它可以是Engine的子类。还记得吗？通过继承关系，我们可以在需要超类（Engine）的地方传递子类（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，用来输出新消息。由于Slant6并没有调用超类中的description方法（即它没有包括[super description]），所以新消息完全替代了Slant6所继承的描述信息。

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，然后提供新的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（改动的代码用粗体字表示）：

```

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

```

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Car却可以在不改变Car类中任何代码的情况下，使用你配置的任何类型的发动机和轮胎。现在，程序的输出就有了很大的改变：

```

I am a slant-6. VR000M!
I am a tire for rain or shine.

```

5.4 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这章中所介绍的两个“万能”新工具。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。

继承在对象间建立了“is a”（是一个）的关系。三角形是一个形状，Slant6是一个发动机，AllWeatherRadial是一个轮胎。如果能说“X是一个Y”，就可以使用继承。

另一方面，复合建立的则是“has a”（有一个）的关系。形状有一个填充颜色，汽车有一个发动机和一个轮胎。与继承相反，汽车不是一个发动机，也不是一个轮胎。如果说“X有一个Y”，就可以使用复合。

面向对象编程的新手通常会犯这样的错误，他们对任何东西都想使用继承，例如让类Car继承Engine类。继承确实是个有趣的新工具，但它并非适用于所有情况。你可以用上述的继承结构创建可正常工作的程序，因为Car代码中使发动机运行的那部分可访问。但是其他人在阅读这段代码时会觉得不合理。汽车是台发动机？当然不是。所以，只应在恰当的时候使用继承。

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。例如，设计汽车相关的程序时，你可能会想“汽车有轮胎、发动机和变速器”，于是你会选择使用复合，并在Car类中声明以上的实例变量。

而在其他情况下，你可能会使用继承。例如，你的程序可能会涉及有照车辆，即需要某种执照才能合法使用的车辆。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——听起来很适合继承。所以你可能会创建一个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），而Automobile（汽车）、MotorCycle（摩托车）等类就可以继承LicensedVehicle类了。

5.5 小结

复合是OOP的基本概念，我们用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个发动机对象和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，我们通常都会为被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的存取方法命名规则。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。

下一章将不谈论任何关于OOP理论的内容，转而学习如何分割不同的类，把它们放到多个源文件中，而不是把所有代码都写到一个大文件里。

目前为止，我们讨论过的项目都把所有的源代码统统放入main.m文件中：类中的main()函数、@interface和@implementation部分都被塞进同一个文件。这种结构对于小程序和应急程序来说没什么问题，但是它并不适用于较大的项目。程序越来越大时，文件会变得越来越长，在文件内查询信息也会变得越来越困难。回想学生时代（假设你已经完成学业了），你并不会把每个学期的论文都放在同一个字处理文档里吧（假设你拥有字处理软件）。你会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个好主意。将程序拆分为多个更小的文件有助于更快地找到重要的代码，并且其他人也能够在查看项目时有一个大致了解。另外，通过将代码放入多个文件，我们也可以更容易地将有趣的类代码发给朋友：只需要打包相关的几个文件而不用打包整个项目。在本章中，我们将会讨论把程序的不同部分放入不同文件的策略和想法。

6.1 拆分接口和实现部分

在前面已经看到，Objective-C类的源代码被分成了两部分。一部分是接口（interface），用来提供类的公共描述。接口包含了所有使用该类所需的信息。编译器编译@interface部分后，你就能够使用该类的对象，调用类方法，将对象复合到其他类中和创建子类。

类的源代码的另一个组成部分是实现（implementation）。@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中声明的方法。

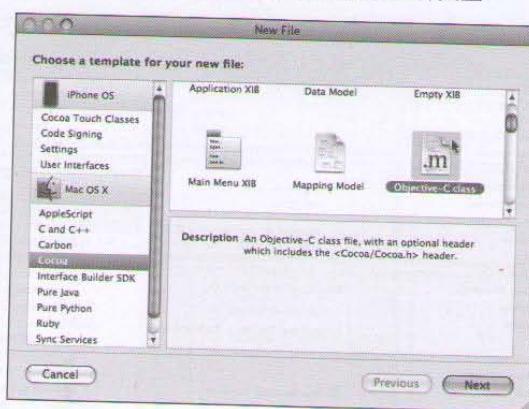
由于在类的定义中，代码被自然地拆分为接口和实现两个部分，所以类的代码通常也被分别放在两个文件里。一个文件存放接口部分的代码：类的@interface指令、公共struct定义、enum常量、#defines和extern全局变量等。由于Objective-C继承了C的特点，所以上述代码通常都被放在头文件中。头文件名称与类名称相同，只是用.h做后缀。例如，Engine类的头文件会被命名为Engine.h，而Circle类的头文件名称则是Circle.h。

所有实现细节（如类的@implementation指令、全局变量的定义、私有struct等）都被放在与类同名，但是以.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++编写代码，这样你就可以同时使用C++语言和Objective-C来编程了。

用 Xcode 创建新文件

生成新类时，Xcode会简化生成过程，自动创建.h和.m文件。在Xcode中选择File>New File时，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。



6

图6-1 在Xcode中创建新文件

选择Objective-C Class，然后单击Next，此时会弹出另一个窗口，要求你填写类的名称，如图6-2所示。

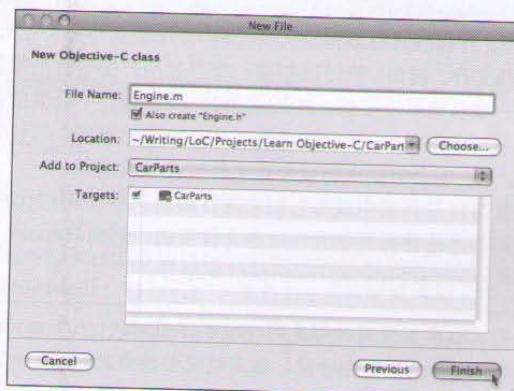


图6-2 命名新文件

在这个窗口中还有其他一些选项。文件名下方有一个复选框，可以选择让Xcode为你创建Engine.h文件。如果你同时打开了多个项目，可以在Add to project弹出菜单中选择存放新文件的项目。在这里我们不对Targets部分做过多的论述，只能说复杂的项目可以有多个目标，每个目标都有自己的源文件配置和不同的生成规则。

单击Finish按钮后，Xcode会把相关文件添加到项目中，并在项目窗口中显示添加结果，如图6-3所示。

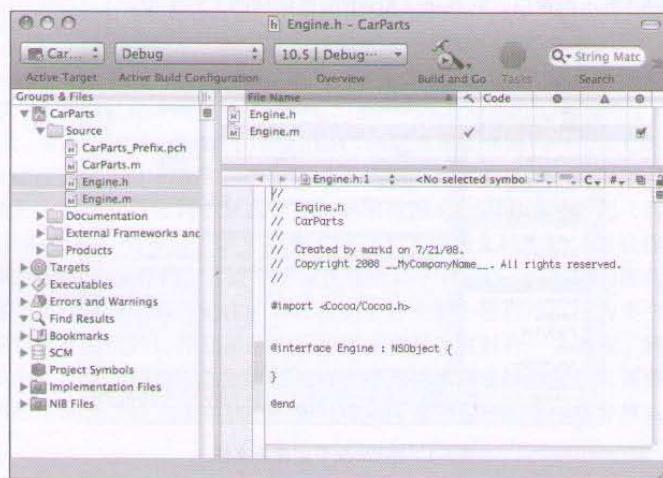


图6-3 在Xcode项目窗口中显示的新文件

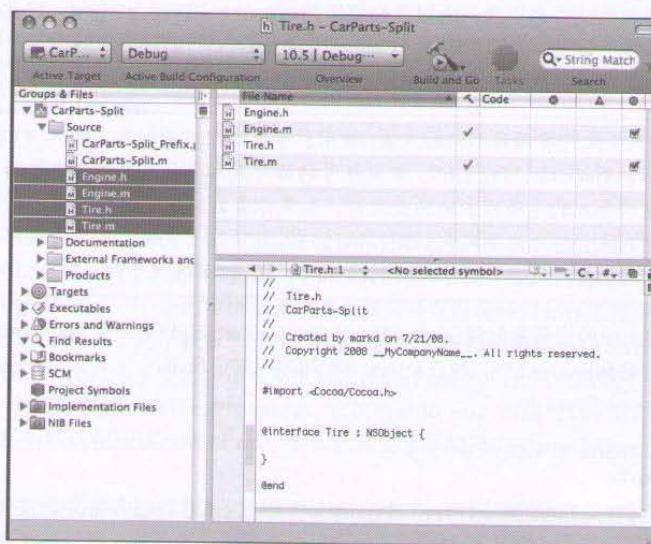
Xcode将新文件放在Groups & Files窗格中选定的文件夹里（如果在创建新文件前你选择的是Source文件夹，那么这些文件将会被存放在该文件夹内）。这些文件夹（在Xcode中被称作组）能够帮助你组织项目中的源文件。例如，你可以创建一个存放用户接口类的组和一个存放数据处理类的组，这样可以使你的项目更容易于导航。在设置组时，Xcode并不会真的在硬盘上移动文件或者创建目录。组关系只是一个由Xcode管理的虚拟工具。如果你愿意，可以创建一个指向文件系统中某个特定目录的组，Xcode会帮助你将新创建文件放在该目录里。

只要创建了文件，就可以在列表中双击它们来进行编辑。Xcode会自动写入一些有用的标准样本代码，这些样本代码都是通常需要包含在这些文件里的，例如#import <Foundation/Foundation.h>以及需要你来填充的空的@interface和@implementation部分。

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的这一部分功能。但是将它替换成#import <Foundation/Foundation.h>也是可以的。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 将所有的类从 CarParts-Split.m 中拿出来并放到它们自己的文件中，该项目可以在文件夹 06.01CarParts-Split 文件夹中找到。每一个类都存放在自己的头 (.h) 文件和实现 (.m) 文件中。下面让我们看看创建这个项目需要做些什么。首先，我们要创建两个分别继承自 NSObject：Tire 和 Engine 的类。选择 New File，然后选择 Objective-C Class，输入类名称 Tire。用同样的方法创建 Engine。图 6-4 显示了项目列表中新建的 4 个文件。



6

图 6-4 添加到项目中的 Tire 和 Engine

现在，从文件 CarParts-Split.m 中剪切 Tire 的 @interface 部分，并将其粘贴到 Tire.h 中，该文件看起来应如下所示：

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

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

接下来，从 CarParts-Split.m 中剪切 Tire 的 @implementation 部分并将其粘贴到 Tire.m 中，你需要在文件首行加上语句 #import "Tire.h"，文件 Tire.m 看起来应如下所示：

```
#import "Tire.h"

@implementation Tire
- (NSString *) description
```

```

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

@end // Tire

```

文件中的首个#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，以后在你所创建的项目里基本上都会这么写。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，将此信息告诉编译器。在程序编译时，如果你碰到了诸如“Cannot find interface definition for Tire”（无法找到Tire的接口定义）之类的错误消息，通常意味着你忘记用#import导入类的头文件了。

说明 注意，导入头文件有两种不同的方法：使用引号或者使用尖括号。例如，#import <Cocoa/Cocoa.h>和#import "Tire.h"。带尖括号的语句是用来导入系统头文件的，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。如果头文件名是被引号括起来的，那么你（或参与这个项目的其他人）就可以编辑它。

现在，重复上面的步骤来创建Engine类。从文件CarParts-Split.m中剪切Engine的@interface部分，并将其粘贴到Engine.h中，现在Engine.h看起来应如下所示：

```

#import <Cocoa/Cocoa.h>

@interface Engine : NSObject
@end // Engine

```

接下来，剪切Engine的@implementation部分并将其粘贴到Engine.m中，文件Engine.m看起来应如下所示：

```

#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和Engine的声明。这个问题很好解决。只需要将下面的两行代码添加到CarParts-Split.m的顶部，即放于#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`#include`这个由C预处理程序处理的命令一样。在这里, C预处理程序只是做了基本的剪切和粘贴的操作, 继续执行后面的操作之前, 将`Tire.h`和`Engine.h`的内容粘贴到`CarParts-Split.m`中。

现在, 你可以生成并且运行`CarParts-Split`了。你会发现, 与原先的版本相比, 程序的输出并没有变化, 还是使用`AllWeatherRadials`和`Slant6`的结果:

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

6.3 使用跨文件依赖关系

6

依赖关系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`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`来接受这个变化。

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`。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`来继承这个变化。例如, 假设`Tire.h`中有一个常量`kDefaultTirePressure`, 它的值是30 psi。而编写`Tire.h`程序的编程人员觉得头文件中胎压的默认值应该改成40 psi, 那么就需要重新编译`CarParts-Split.m`来使用新的值40替代原有值30。

导入头文件在头文件和进行导入的源文件之间建立了一种很强的依赖关系。如果这个头文件有任何变化, 则所有依赖于它的文件都需要被重新编译。这样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, 我们称它为`UserInterfaceConstants.h`吧。如果你改变了`UserInterfaceConstants.h`, 那么所有这100个.m文件都会被重新生成, 即便是基于Intel的Xserve主机群集任你使用, 这也需要花费相当长的时间。

由于依赖关系是可传递的, 即头文件也可以互相依赖, 所以重新编译的问题甚至可能会变得越发严重。例如, 如果`Thing1.h`导入了`Thing2.h`, 而`Thing2.h`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`Thing1.h`的文件。尽管重新编译需要花费很长的时间, 可是至少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所带来的影响。依赖关系问题的存在是因为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, 不需要了解它的整个定义。

例如，对象复合后（如前一章所述），这个复合使用指向对象的指针。这样之所以能行得通，是因为所有Objective-C对象都存放在动态分配的内存中。编译器只需要知道这是一个类就可以了。然后它就会发现这个实例变量是指针的大小，而这个指针的大小在整个程序中都不会改变。

Objective-C引入了关键字@class来告诉编译器：“这是一个类，所以我只需要通过指针来引用它。”这样编译器就会“平静”下来：它并不需要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，用和移动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
{
    Tire *tires[4];
    Engine *engine;
}

- (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。这条错误信息大概会像这样：error: parse error before "Tire"，是编译器在说“我不明白这个”。

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大量信息。

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：

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

@class Tire;
@class Engine;
```

```

@interface Car : NSObject
{
    Tire *tires[4];
    Engine *engine;
}

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

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

- (void) print;

@end // Car

```

这就是编译器处理Car的@interface部分所需要知道的全部信息了。

6

说明 @class创建了一个前向引用。就是在告诉编译器：“相信我，以后你会知道这个类到底是什么。但是现在，你只需要知道这些。”

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，那么最后就会出现编译错误。但是如果在A.h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 让汽车开动

以上完成了Car类的头文件部分。但是Car.m需要关于Tire和Engine的更多信息。编译器需要知道Tire类和Engine类继承自什么类，这样才能检查对象，以保证它们能够响应发送过来的消息。为此，我们在Car.m中导入Tire.h和Engine.h，我们还需要从CarParts-Split.m中剪切出Car的@implementation部分并粘贴到Car.m中。现在Car.m文件看起来应该是：

```

#import "Car.h"
#import "Tire.h"
#import "Engine.h"

@implementation Car

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

现在你可以再一次生成并运行这个程序了，程序的输出结果与原来一样。是的，我们又对这个程序进行了重构（嘘——不要告诉其他人）。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中拆分出两个类：Slant6和AllWeatherRadial。这有点困难，因为它们是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，所以不能在头文件中使用@class语句。我们只能在Slant6.h中使用#import "Engine.h"，在AllWeatherRadial.h中使用#import "Tire.h"。

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。它需要了解超类实例变量的配置（数据类型、大小和排序）。还记得吗？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。然后编译器用这条信息来判断在内存的什么位置能找到这些实例变量，从每个方法调用自身的self隐藏指针开始寻找。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，那么现在Slant6.h应该是如下所示的样子：

```
#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>，所以不需要我们自己再导入一遍了。然而，如果你想在该文件里加上#import <Cocoa/Cocoa.h>，这是可以的，因为#import命令具有足够的智能，它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. VR00OM!");
} // description

@end // Slant6
```

重复执行上面的步骤，将AllWeatherRadial移到它自己的两个文件中。毫无疑问，现在你对这种操作已经驾轻就熟了。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

```

如果现在生成并运行这个项目，会得到与拆分文件之前一样的输出结果。

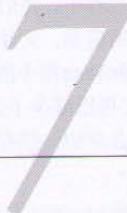
6.4 小结

在本章中，我们学习了组织源代码的基本技巧——使用多个文件。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

入（使用`#import`指令）头文件来获得该类的功能。

在学习过程中，我们认识了跨文件依赖关系，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。文件之间互相重复的导入会增加编译次数，也会导致不必要的重复编译。而巧妙地使用`@class`指令能够减少编译时间，`@class`告诉编译器“相信我，你最终能够了解这个名称的类”，它可以减少不得不导入的头文件数量。

接下来会介绍一些有趣的Xcode功能。我们下一章见。



Mac程序员大部分时间都在Xcode内编写代码。Xcode是一个很好用的工具，有很多奇妙的功能，但并不是所有功能都是显而易见的。当你需要长时间使用一个强大的工具时，就会想要尽可能多地了解它。本章将介绍一些Xcode编辑器的使用技巧与诀窍，这些技巧对于编写和浏览代码以及查找所需的信息都是大有帮助的。我们也会提到一些用Xcode调试程序的方法。

Xcode是一个很大的应用程序，它有很强的自定义功能，有时，该功能甚至强大到不可思议（我提过它很大吗）。仅是介绍Xcode就可以写一整本书（有人曾写过），所以我们只讲解重要部分，以便你能够更高效地使用它。建议你在开始的时候使用Xcode的默认设置。然而，当你被某些问题困扰时，可能也会找到根据个人喜好来定制的设置。

面对像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，然后使用它一段时间，接着再浏览一遍文档。每次浏览文档时，就会理解并掌握更多的内容。洗头发，冲头发，再重复，你的头发会焕发光彩。

这里要讲的是Xcode 3.1，也就是写这本书时的最新版本。苹果公司热衷于在Xcode版本升级时添新删旧。所以如果你在使用Xcode 42.0，那么本章的屏幕截图恐怕已经过时了。现在，让我们开始学习这些使用技巧吧！

7.1 改变公司名称

你可能已经注意到，新建Objective-C源文件时，Xcode会自动生成注释块：

```
//  
// TapDance.h  
// Groovilicous  
//  
// Created by markd on 7/25/08.  
// Copyright 2008 __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，还有它的生成时间。但是，默认的公司名称是不恰当的。上次我调查过了，__MyCompanyName__并不聘用Mac程序员，他们只聘用TPS报告者。

不知道出于什么原因, Xcode 3.1并没有任何用来修改占位符 `_MyCompanyName_` 的用户接口。你需要到 Terminal 中去修改它。因为你会新建很多源文件, 所以我们将默认的公司名称改为更合理的名称。可以是你自己的姓名、你的公司名称或者是随便编出来的什么。

首先在 Finder 中打开 Utilities 文件夹, 也可以使用快捷键 `⌘U` 直接打开该文件夹。找到 Terminal 应用程序并运行该程序。在 Terminal 中, 准确地输入下面的代码, 所有代码都在一行中, 不过要用你自己的公司名称替代 “Length-O-Words.com”。

```
defaults write com.apple.Xcode PBXCustomTemplateMacroDefinitions
  '{"ORGANIZATIONNAME" = "Length-O-Words.com";}'
```

将这些内容输入到一行中后, 按下 `enter` 键。如果运行正常, 你将看不到任何输出结果。所幸, 这条命令只需要运行一次。退出并重启 Xcode, 现在为新文件生成的文件注释就合理多了:

```
//
//  HulaDance.h
//  Untitled3
//
//  Created by markd on 7/25/08.
//  Copyright 2008 Length-O-Words.com. All rights reserved.
```

我保证在本章剩下的部分中, 你不会再看到 Terminal 程序了。

7.2 使用编辑器的技巧与诀窍

Xcode 为我们提供了几种组织项目和源代码编辑器的基本方式。我们之前看到的是默认界面, 界面的主要部分是管理项目和编码任务的一体化窗口, 还有一些辅助窗口, 如图 7-1 中所示的运行日志。默认界面中有一个编辑面板, 可以在其中编辑所有的源文件, 编辑器中的内容会根据在界面左边 Groups & Files 窗格中选定的源文件而发生变化。

Xcode 还有一种工作模式, 即在编辑时每个源代码文件都分别在各自的窗口中打开。如果你的显示屏很大并且不介意使用多个窗口, 那么这种工作模式可能很适合。但是在这里, 我们假设你使用的是嵌入在 Project 窗口中的代码编辑器, 因为这样制作屏幕截图会方便得多。

窗口左侧是 Groups & Files 列表, 它显示出项目的所有组成部分: 源文件、链接的框架以及描述如何实际生成各个程序的 Targets。在项目中, 你也可以找到一些其他的工具, 如书签 (之后我们将会谈及书签)、源代码控制存储库通道 (便于同其他程序员分工协作)、所有的项目符号和一些智能文件夹。

在窗口的上方, 工具栏下面是一个浏览器, 其中显示了你从 Groups & Files 中选择的文件。可以使用搜索框来减少所显示的文件列表。图 7-2 显示了选择 Source 文件夹后, 查询字母 `n` 的搜索结果。

此时浏览器显示了每个名称里有字母 “`n`” 的源文件。在浏览器中单击文件名就可以在编辑器中显示该文件内容。大型项目可能会有上百个源文件, 而对于这种大量文件的导航, 编辑器是个很方便的工具。在本章的后面部分, 我们会深入介绍源文件导航的相关话题。

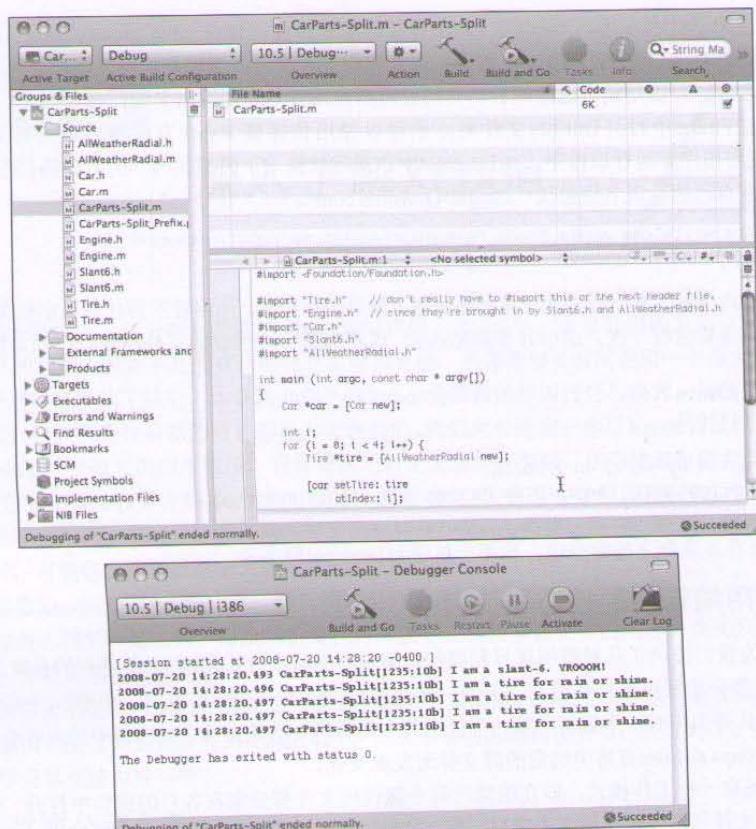


图7-1 Xcode的默认用户界面，包含源代码和调试器



图7-2 缩小文件列表范围

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。在窗口的最右边（屏幕截图中没有显示），有一个名为Editor的默认工具栏的图标，它可以切换是否显示浏览器，该操作的

快捷键是 $\text{⌘}+\text{E}$ 。

即使你使用单窗口模式，用一个窗口显示一两个源文件也是有帮助的，特别是要比较两个文件的时候。在Groups & Files窗格中双击源文件名称可以在新窗口中打开这个文件。也可以在两个窗口中打开同一个文件，但是要注意，有时候这两个窗口中显示的内容并不是同步的，只有你分别单击了它们，才能同步更新内容。

7.3 在 Xcode 的帮助下编写代码

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

7.3.1 首行缩进

你可能已经注意到本书所有的代码都有着整齐的缩进格式，**if**语句和**for**循环的正文代码都比外层代码缩进得更多。Objective-C并没有要求你缩进代码，但是这么做是个好习惯，因为它能够使你的代码结构更清晰。Xcode会自动缩进你输入的代码。

7

有时，对代码进行大量的编辑会使其变得混乱，Xcode也能改善这种状况。按住Control键单击（或右键单击），打开编辑器的上下文菜单，然后选择Re-indent selection，Xcode将会重新整理选定内容的格式。这项功能没有内置的热键，但是你可以在Xcode设置菜单的Key Bindings窗格中添加。

$\text{⌘}[$ 和 ⌘] 键可以把选定的代码左移或右移，如果你只是在已有代码外添加一个**if**语句，使用它们可以很方便地移动代码。

假设编辑器中有下列代码：

```
Engine *engine = [S1ant6 new];
[car setEngine: engine];
```

然后，你决定只有在用户需要时才会创建一台新的发动机：

```
if (userWantsANewEngine) {
    Engine *engine = [S1ant6 new];
    [car setEngine: engine];
}
```

这时，你可以选择中间两行代码，然后使用 ⌘] 向右移动这些代码。

你可以随意调整Xcode的缩进。可能你更喜欢空格缩进，而不是tab键缩进，可能你更愿意把括号重起一行而不是放在**if**语句的同行结尾。不管怎样，都可以按照你自己的统一格式来调整Xcode。这里有一个小提议：如果你想快速地和其他程序员展开热烈的网络讨论，那么从讨论代码格式的喜好开始吧。

7.3.2 代码自动完成

也许你已经注意到了，有时Xcode会在你键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功能。编写程序时，Xcode会为所有内容生成索引，包括项目

中变量名和方法名，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习惯是好是坏。输入代码时，Xcode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7-3所示。

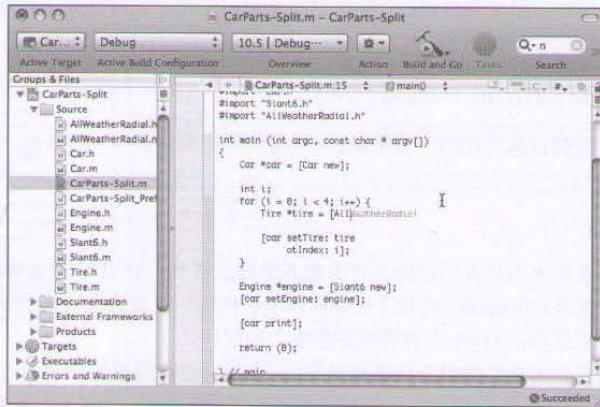


图7-3 Xcode的代码自动完成功能

上图中，我们键入了[All，Xcode认为我们想要给AllWeatherRadial类发送消息。它猜对了，所以我们可以按tab键，使用AllWeatherRadial作为自动完成的内容。

你会说：“啊，这也太简单了！我们只有一个以All开头的类啊！”这话没错，但即使有很多可能的代码，Xcode也能够提供自动完成列表。并且，在任何时候你都可以按Esc键，让Xcode打开这个包含所有可能代码的自动完成列表，如图7-4所示。

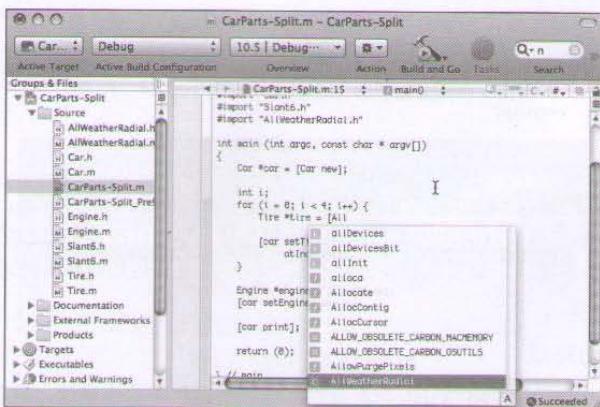


图7-4 all (All) 开头的代码自动完成列表

我们可以看到有很多以“all”开头的可能代码。Xcode发现当前项目中的AllWeatherRadial.h开头的，于是认为它是合理的第一选择。名称旁边的彩色方框表示这个符号的类型：E表示字符串，f表示函数，#表示#define指令，m表示方法，C表示类，等等。

如果不想要显示自动完成列表，可以使用“control+.”在各个选项之间循环，也可以通过“shift+control+.”反向循环。如果你没有记住我们提到过的快捷键也没关系，本章的最后会有一个快

速参考表单。

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，可以使用一系列的参数来生成字典的键值和对象。这个方法是dictionaryWithKeysAndObjects:dictionaryWithObjectsAndKeys:。谁能记得住呢？查找这个名称的一个简单方法是输入dictionary的方法调用指令，输入一个空格表示类名已经输入完毕，然后按Esc键。Xcode知道你需要在这里输入一个方法名，于是显示出NSDictionary中的所有方法。果然，我们看到了方法dictionaryWithObjectsAndKeys，它位于图7-5中提示列表的上方。

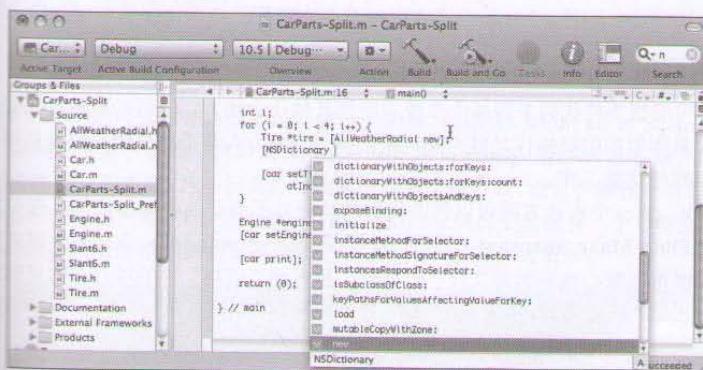


图7-5 用代码提示功能查看类的内容

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7-6所示。这又是怎

么回事呢？在这里Xcode提示的是接收两个参数的方法setTireAtIndex:。Xcode的代码提示功能只是补充名称。这里显示的两个参数实际上是占位符。如果再次按tab键，这个方法的参数会补充成setTire:atIndex:，如图7-7所示。

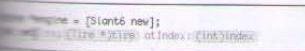


图7-6 代码自动完成中的占位符



图7-7 选择一个占位符

占位符突出显示了，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来选择它。你甚至可以不用把手从键盘上拿开，按control和“/”键就可以将光标移到下一个占位符。

占位符。

7.3.3 括号匹配

输入程序代码时，也许你会注意到在输入了某些字符（如）、]或{}）后屏幕会有些闪烁。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7-8所示。

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

图7-8 括号匹配

这个功能有时被称为“括号匹配”，当你关闭一连串复杂的分隔符时，使用它会非常方便。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会发出蜂鸣声，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。

7.3.4 批量编辑

有时，你可能想要将代码中的某个改变应用到其他几个地方，但是又不想自己逐一编辑。而且手动去做大量相似的编辑操作是件充满危险的事情，因为人类不是很擅长重复枯燥的工作。所幸，计算机很胜任这类工作。

在Xcode中，第一个在这方面可以帮助我们的功能并不会真的去操作代码，而是要建立一个安全网。选择File>Make Snapshot（或用快捷键command-control-S），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，可以随心所欲地“破坏”你的项目。如果你意识到犯了一个很严重的错误，可以通过File>Snapshots打开快照窗口，这样就可以从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 实际上快照被存储在一个磁盘镜像中，它存放于~/Library/Application Support/Developer/Shared/SnapshotRepository.sparseimage。有时这个磁盘镜像可能会被损坏（也许因为生活压力太大了），这时Xcode会向你报告一个“神秘”的错误：Snapshot Failed: A project snapshot cannot be created。如果看到了这样的错误，请尝试删除这个镜像并重新启动。

当然，Xcode也有查找和替换的功能。我们有子菜单Edit>Find，里面有几个非常方便的选项。Find in Project可以在项目的所有文件里进行查找和替换。图7-9是在项目范围内的查找和替换窗口。

假设我们要将car替换成automobile。在输入框中输入两个单词，单击Find，你可以看到指向Car类和car局部变量的引用。可以取消选中Ignore case复选框，这样就只改变main函数里的局部变量。你也可以单击Replace All，在整个项目里应用这个替换操作。

但是，对于这种替换操作，查找和替换功能并不好用。如果你只是想重命名函数中的变量，那么它做了过多的操作（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，它又无法办到。另外重要的一点是，它不能重命名源文件。

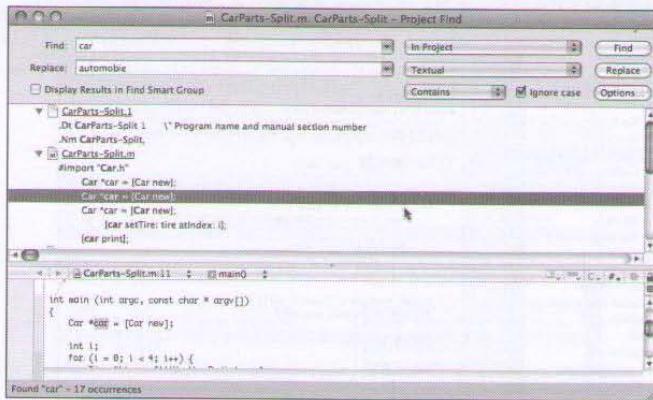


图7-9 在项目范围内进行查找和替换

7

Xcode有两个功能可以弥补这些不足。第一个功能可以简称为Edit all in Scope。你可以选定一个符号，如局部变量或参数，然后选择Edit>Edit all in Scope，然后，在你输入时，所有该符号出现的地方都会立即更新。这不但是进行大量改动的快捷方式，而且在操作时看起来也很“酷”。

图7-10显示在范围内编辑car的过程。注意，所有car局部变量都有一个方框围着。只要开始输入Automobile，所有方框里的内容都会随之发生改变，如图7-11所示。

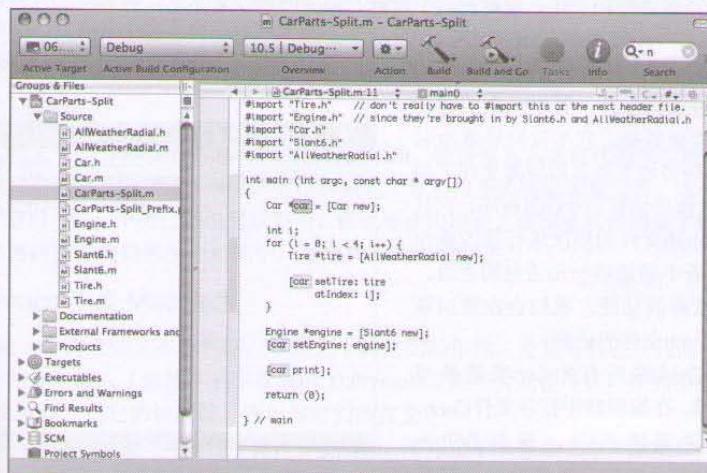


图7-10 开始在范围内编辑全部内容，将car替换成automobile

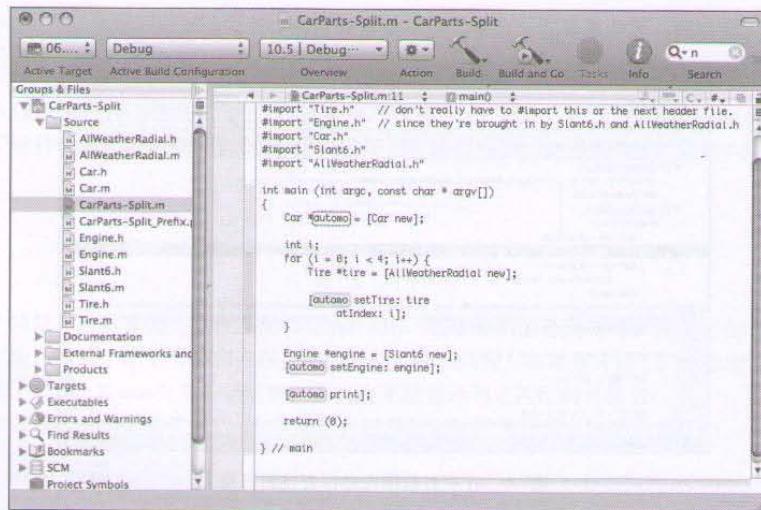


图7-11 在范围内编辑全部内容

输入完毕后，只要在源文件编辑窗口中单击其他地方，就会离开**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。因为这个功能与Xcode中的语法着色功能联系紧密，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——这里需要很多技巧。

还记得前面提到过的“重构”吗？这不是为了让人感觉我们很聪明而编造出来的。Xcode拥有一些内置的重构工具，其中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妙事。并且如果你有一个GUI程序，它甚至能够深入到nib文件的层次进行修改操作（如果你完全看不懂最后一句话也别着急。这是一个非常棒的功能，我们会在第14章讨论更多关于nib文件的话题）。

现在，尝试将所有的Car类替换成Automobile类。在编辑器中打开文件Car.h，在字Car中设置插入点，选择**Edit > Refactor**，你会看到如图7-12所示的对话框，在图中已经输入了Car的替代字

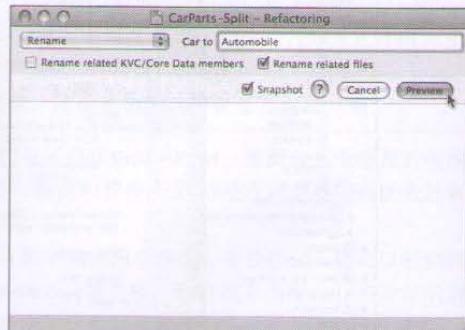


图7-12 开始重构

Automobile。

为了保险，要确保Snapshot复选框被选中。单击Preview之后，Xcode会分析出要做什么，并将结果展示出来，如图7-13所示。

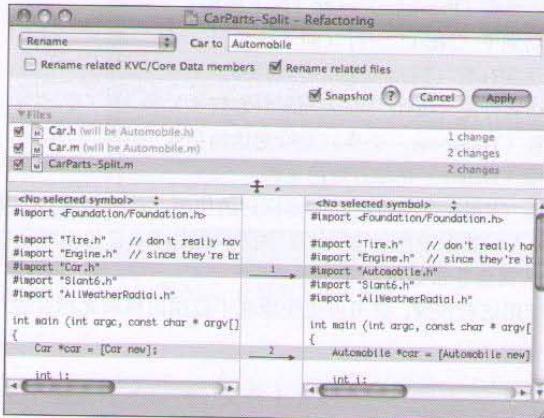


图7-13 Xcode告诉我们它将如何进行重构

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，在窗口下方的文件合并查看器中查看Xcode将会对这个文件做哪些更改。在这个截图中，能够看到的是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。

可惜的是，重构并不能重命名注释中的文字。所以，类末注释、Xcode生成的文件头注释或着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。

7.3.5 代码导航

大多数源文件都有相似的生命周期。它们被创建，被快速写入大量代码来实现它们的功能，通过增增改改的阶段后进入维护模式，在维护模式下你不得不先阅读大量文件，然后才能添加或修改代码。最后，在一个类已经编写成熟后，你也要在应用它之前浏览其代码来了解它是如何工作的。本节将研究在代码的生命周期中，浏览代码的不同方法。

7.3.6 emacs 不是 Mac 程序

emacs是一个古老的文本编辑器，它诞生于20世纪70年代，并且可以在现代的Mac操作系统上运行。一些怀旧的人（包括本书作者Mark Dalrymple）还在天天使用它。我们不会过多地谈及emacs，只是介绍一些它的快捷键，告诉你它们的含义。

“emacs快捷键”描述的是一些按键，有了它们，不用把手从键盘上拿开就能移动光标。就像很多人更喜欢使用箭头键而不是鼠标一样，emacs用户则更倾向于使用这些光标移动按键而不是箭头键。最妙的是，这些移动按键在任何Cocoa的文本域内都是有效的，这不仅包括Xcode，还有

TextEdit、Safari的URL地址栏和文本域、Pages和Keynote文本域等。下面列举了这些按键。

- control-F: 向右边前(Forward)移(同右箭头)。
- control-B: 向左边后(Backward)移(同左箭头)。
- control-P: 移动到前(Previous)一行(同上箭头)。
- control-N: 移动到下(Next)一行(同下箭头)。
- control-A: 移动到行首(同command+左箭头)。
- control-E: 移动到行尾(End, 同command+右箭头)。
- control-T: 转置(Transpose, 交换)光标两边的符号。
- control-D: 删除(Delete)光标右边的字符。
- control-K: 删除(Kill)光标所在行中光标后的代码, 便于你重写行尾的代码。
- control-L: 将插入点置于窗口正中。如果你找不到光标或者想要移动窗口使插入点快速位于正中, 这个按键会非常好用。

如果你记住并且应用这些按键, 就可以更快地在小范围内移动光标并进行编辑操作(不仅仅在Xcode中)。

7.3.7 任意搜索

Xcode项目窗口的右上角有一个搜索框, Xcode通过你输入的字符来过滤浏览器中显示的内容。要使用这个搜索框, 首先要在Groups & Files列表中选定任意项, 如Source文件夹, 然后浏览器会显示该文件夹里的所有源文件。在搜索框中输入字符, 如Car, 你将会看到如图7-14所示的过滤后的列表。

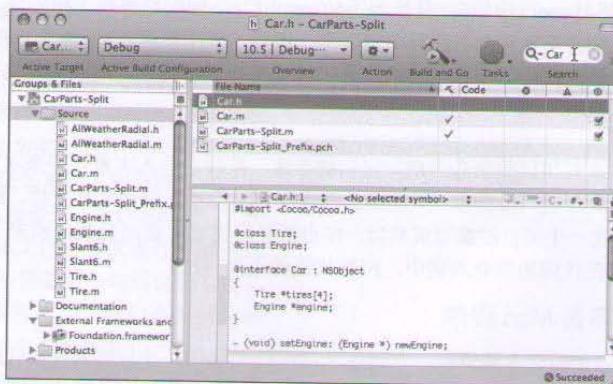


图7-14 过滤Car

也可以在你使用的框架中进行搜索。在文件夹External Frameworks and Libraries中选择Foundation Framework, 你将会看到该框架提供的头文件列表。在搜索框中输入文本可以对这些文件进行筛选, 例如, 输入array将会得到与数组相关的两个Foundation框架头文件, 如图7-15所示。

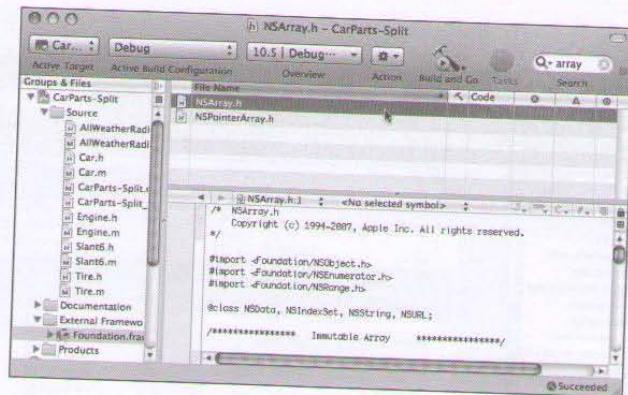


图7-15 在框架文件中搜索

7.3.8 芝麻开门

假如你正在看某个源文件，看到了文件上方的 `#import`，如果能够迅速打开这个头文件而不用鼠标点来点去的，是不是很好呢？这个可以实现！只要选定文件名（甚至不用选择.h），然后选择 **File>Open Quickly**，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** 将会打开一个对话框，这是另外一种文件查找方法，它的快捷键是 **⌘D**。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，但它却是非常快捷的项目内容搜索方式。在搜索框内输入 `tire` 可查找和轮胎相关的文件，如图7-16所示。你也可以输入其他文本，如输入 `NSArray` 去查找 `NSArray` 头文件。

如果你只是想查看某文件的配套文件（如果在查看 `Blah.m`，就显示 `Blah.h`，反之亦然），那么你可以通过按 **⌘↑**（`command+option+上箭头`）来实现。

7.3.9 书签

Xcode允许你在代码中插入书签。在代码中可能有一些需要引起人们注意的地方，例如你要稍后修改的行，或者你可能想标记一个非常重要的类的定义。要创建书签，首先在源文件中放入插入点或者选定一个文本区域，然后选择 **Edit>Add to Bookmarks**，或者使用默认的快捷键 **⌘D**（与Safari中的一样），最后当出现提示时，输入书签名称即可。

7



图7-16 Open Quickly对话框

你将在Groups & Files窗格的Bookmarks部分中看到你的书签，如图7-17所示。

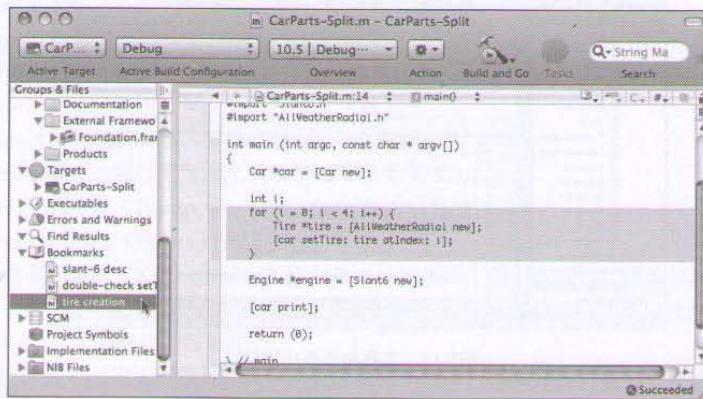


图7-17 Xcode书签

7.3.10 集中注意力

你可能已经注意到紧挨着源代码左侧的两个空列。左边较宽一些的列叫做边列 (gutter)，我们会在之后讨论调试的时候谈到它。较窄的一列叫做焦点列 (focus ribbon)，顾名思义，焦点列能够让你在浏览代码段时集中注意力。

注意焦点列的灰度：代码嵌套得越深，在它旁边的焦点列中，灰色也会越深。这种颜色编码能够使代码的复杂程度一目了然。你可以在焦点列的不同灰色区域悬停鼠标来突出显示相应的代码段，如图7-18所示。

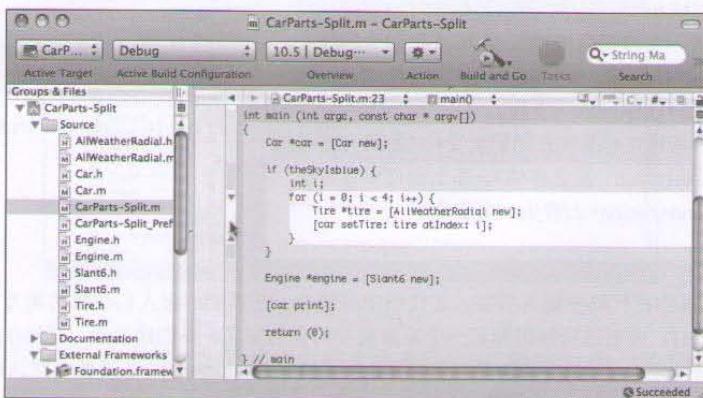


图7-18 用焦点列高亮显示代码

也可以单击焦点列来折叠相应的代码段。假设你非常确定图7-18中所示的if语句和for循环都是正确的，不想再浏览它们了，这样就可以专心关注函数中的其他代码（如前面所说的，集中注意力）。那么单击if语句的左侧，该语句的正文就折叠起来，如图7-19所示。

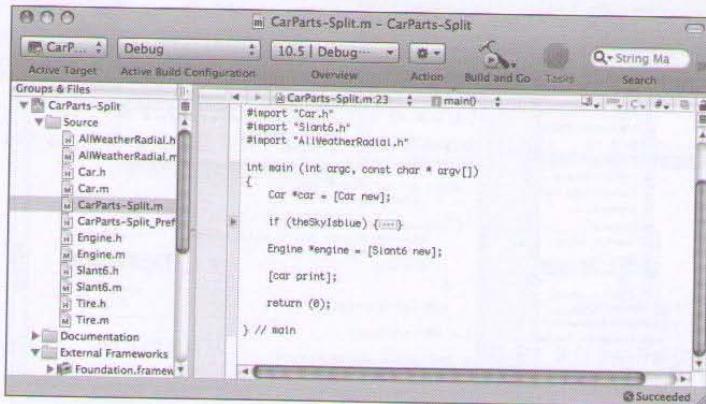


图7-19 折叠代码

现在，你可以看到if语句的代码主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，也可以通过单击焦点列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并且运行。这种功能也可以叫做代码折叠。选择View>Code Folding菜单可以查看其他选项。

7.3.11 开启导航条

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。这里面的很多控件可以帮助你快速地在项目中的源文件之间切换。

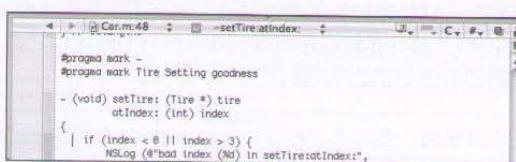


图7-20 导航条

从左往右看，导航条中首先是后退和前进按钮，使用它们可依次浏览本次编辑会话中打开过的文件，其工作方式与Safari的后退和前进箭头一样。在这两个按钮的右边是一个弹出菜单，它显示的是当前文件（Car.m）和插入点所在的行号（48）。单击这个菜单可以查看文件打开的记录，如果愿意，也可以选择一个文件并在编辑器中打开它。

接下来是功能菜单，它显示出插入点目前位于方法`-setTireAtIndex:`中，这就是这个菜单的内容。单击它可以查看所有你感兴趣的符号，如图7-21所示。

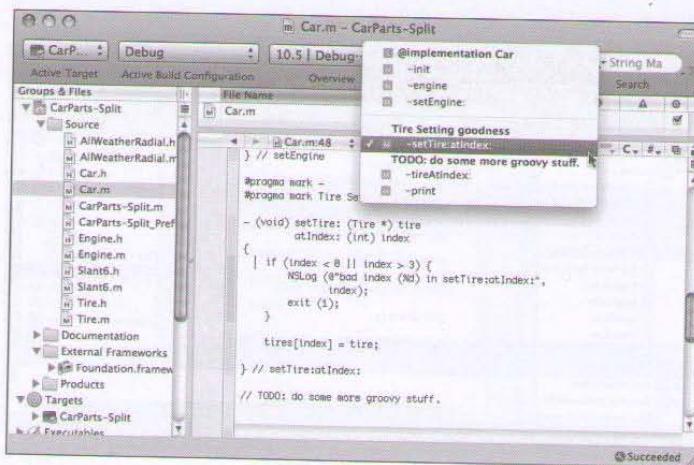


图7-21 文件中的符号

`-setTireAtIndex:`被突出显示，因为光标在这个方法里面。你可以在它的上下看到其他方法，它们按照在文件中的顺序排列的。按住Option键并单击功能菜单可以将这些方法按字母顺序排列。你也会注意到菜单里还有一些其他内容，如Tire Setting goodness和TODO，这些是从哪里来的？

你可以通过两种方法在菜单中放入其他内容。一种方法是用`#pragma mark stuff`将字符串`stuff`放入菜单，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，并将这些文本放入功能菜单中。这些都是程序员所做的记号：“最好在将程序发布到难以预料的环境中之前，先回过头来看看这些标记。”

说明 “Pragma”源自希腊单词，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，则应该“点头微笑”并且忽略它，而不是生成警告或错误信息。

下一个控件看起来像一本小小的书，它能显示当前文件中所有已被设定的书签。你会注意到这个图标和Groups & Files窗格中的书签图标是一样的。

下个控件显示文件中的所有断点。我们会在7.5节中更多地讨论断点。

下一个标着C的菜单可以上下浏览类的层次结构。如果代码编辑器中显示的是文件Engine.m,那么选择类菜单时将会看到如图7-22所示的菜单。

在图7-22中, Engine被突出显示。Xcode知道Slant6是它的子类而NSObject是它的超类。选择Slant6可以打开Slant6.m, 选择NSObject则可以打开Foundation头文件中的NSObject.h。由于NSObject.m属于苹果公司, 所以我们并不能访问它, 于是Xcode退而求其次, 打开对应的头文件。如果选择NSObject, 打开头文件NSObject.h, 然后再次选择类菜单, 那么会看到所有的NSObject子类, 很多的子类! 和你看到的一样, 这个菜单非常便于查看代码的继承树。

下一个标着#的菜单是被包含文件的菜单。如果正在查看Slant6.h, 那么将会出现如图7-23所示的菜单。

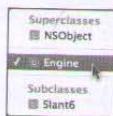


图7-22 导航条的类菜单



图7-23 被包含文件的菜单

7

这个菜单显示出Slant6.h引入（#imports或#include）了Engine.h。Xcode也知道其他两个文件CarParts-Split.m和Slant6.m同样包含了Slant6.h。这也是另一种快速查看项目依赖关系的方法。

下一个有两个重叠方块的图标可以打开当前文件的配套文件, 就像快捷键⌘⌥↑的功能一样。本行最后一个图标是一个小锁, 它可以将文件设置为只读。如果你正在浏览一个文件, 可以将它设置为只读, 这样就不会因为猫跳上键盘行走之类的意外行为而改动文件内容了。

在小锁图标的正下方, 滚动条上面的是拆分按钮。单击它可以将源代码窗口一分为二, 这样就能同时看到文件的两个部分, 如图7-24所示。也可以用拆分条来重新调整这两个窗口的大小。

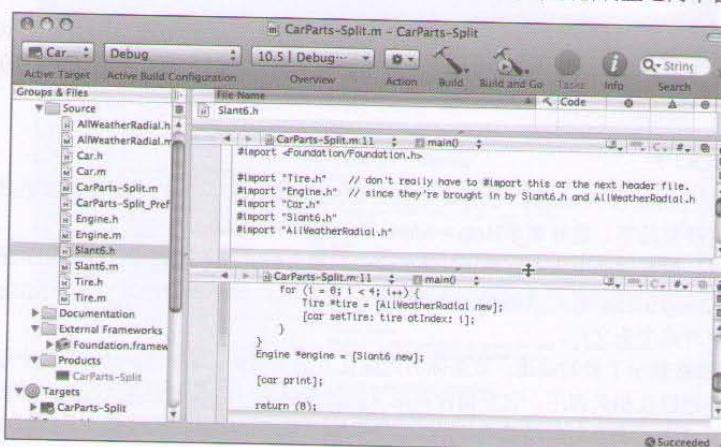


图7-24 拆分代码编辑器

拆分后的每个编辑窗格都有自己的导航条，可以用它在窗格中显示不同的文件部分。按住Option键并单击拆分框可以垂直拆分窗口，而不是水平拆分。你也可以不停地拆分下去直到窗口如图7-25所示那般杂乱。单击拆分按钮下面的方形按钮可以关闭该编辑窗格。

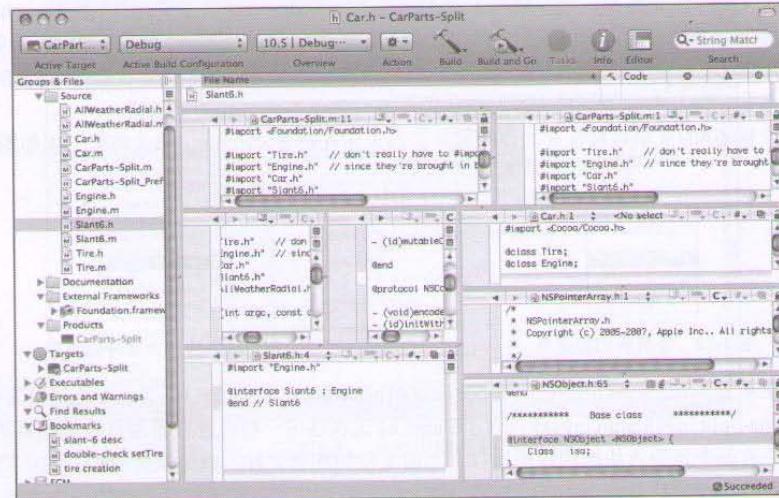


图7-25 多次拆分代码编辑器

7.4 获取信息

可在代码和Cocoa头文件之间自由切换浏览是很好的功能，也很方便，但有时你也需要从自己的代码之外获得信息。幸运的是，Xcode有一个存储着极具价值的文档和参考材料的“宝库”。（顺便问一句，宝库是什么？）

7.4.1 研究助手

研究助手（Research Assistant）的浮动小窗口会根据你在Xcode里的交互操作来更新所显示的内容。要打开研究助手，选择菜单Help>Show Research Assistant。

举个例子，假设你的插入点在单词NSString中，那么研究助手看起来就会如图7-26所示。

这里有大量的信息。前两项用来在文档窗口中显示NSString类的参考文档，NSString.h项则在编辑器中打开这个头文件。

Abstract窗格显示了类的描述。如果你的光标在方法调用中，那么Abstract窗格中显示的就是这个方法的描述以及相关调用，包括指向高层文档的指针和一些频繁使用NSString的示例代码。

查看你能够调整的不同生成设置时，研究助手非常有用。只需要在项目信息面板中选择生成设置，就可以在研究助手中查看它的解释了。

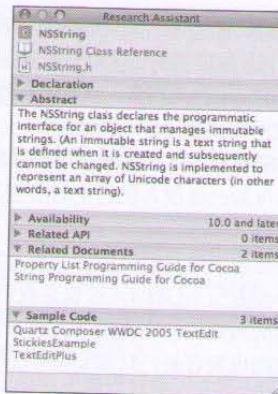


图7-26 研究助手

7.4.2 文档管理程序

7

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`（`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查看`UTF8String`，如图7-27所示。

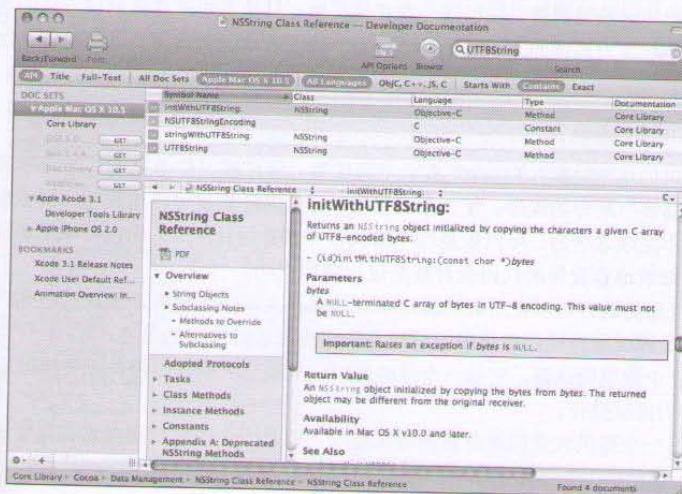


图7-27 Xcode文档浏览器窗口

这是一个强大并且很热闹的窗口，它包含了大量的信息。窗口右上角是一个搜索框，里面预先填写了单词UTF8String。工具栏下方的按钮条用来细分要搜索哪些文档集。你可以在所有文档里搜索，也可以只搜索Mac OS X或iPhone的文档集。或者按程序语言来限制搜索范围，就是说如果你只是在编辑JavaScript，那么就不需要显示C++的信息。

在这个按钮条下方的左侧窗格中显示了文档集和书签组。你可以使用快捷键⌘D在文档的某段中添加书签。这种书签和之前你可能已经放入代码中的书签不同，文档书签在Xcode中是共用的，并不只局限于某个特定的项目。

窗口右侧的上方是一个显示所有搜索结果的浏览器。在图7-27中，它搜索出了3个方法和1个常量。我们最感兴趣的是最后一个搜索结果UTF8，这也是我们实际要搜索的文本。

在这个浏览器下方是显示文档的浏览器，它是用来阅读文档的小窗口。但是要在Xcode的其他窗口甚至是网页浏览器中打开文档也很容易，只需要按住control键并单击（或右键单击）文档区域来打开菜单即可。

7.5 调试

bug无处不在。程序中有错误是不可避免的，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，首先深呼吸，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 暴力调试

最简单的一种调试方式是暴力的，即暴力调试。在程序中放入输出语句（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的名字。你可能碰到过一些瞧不起暴力调试的人，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。调试器是位于你编写的程序和操作系统之间的程序，它能够中断你的程序，使之在运行中停止，这样你就可以检查程序的数据，甚至可以修改程序。在你完成这些后，可以恢复程序的执行并查看运行结果。你也可以单步执行代码，逐行运行程序来细致地查看你的代码会对数据进行哪些改动。

Xcode中有几处地方可以使用调试器。第一处就在文本编辑器里，通过在边列上设置断点来实现。断点就是调试器应该停止程序的运行并让你查看运行情况的地方。

Xcode有一个微型调试器，它是一个浮动窗口，里面有一些基本控件能够跳过Xcode调试器，直接实现简单的调试操作。

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。

说明 Xcode使用的调试器是GDB。GDB是GNU项目的一部分，它可以在很多不同的平台上使用。如果你愿意，可以通过命令行来运行它。GDB有着完善的文档系统，尽管它的文档有些难于理解并且网络上流传着好几个版本的GDB教程。

下面我们将讨论Xcode文本编辑器里的调试器。如果你想知道更多内容，就应该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生成配置。可以通过Xcode工具栏中的弹出菜单**Active Build Configuration**来检验。Debug配置告诉编译器发出额外的调试符号，而调试器通过这些符号可以知道程序在什么地方都有些什么。

同时，还要确认程序是用调试器模式来运行的。在Xcode里有两种运行程序的方法。选择菜单**Run>Run**或者按快捷键 ⌘R ，将会不使用调试器运行程序。若要使用调试器，选择菜单**Run>Go (Debug)**、**Run>Debug**或者按快捷键 ⌘Y 。

7.5.4 开始调试

开始使用调试器时，这个调试GUI程序比我们一直使用的命令行程序要容易一些。GUI程序停止运行并等待用户操作，所以使用它能够给我们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。如果使用我们的命令行程序，运行结果会一闪即过，来不及做多少调试工作。所以让我们先在**main**函数中设置一个断点，我们会使用上一章提到的06.01 CarParts-split程序。

打开文件CarParts-Split.m，单击边列，也就是之前看到过的焦点列左边的宽条。你应该能看到一个蓝色箭头状的物体，那就是新断点，如图7-28所示。

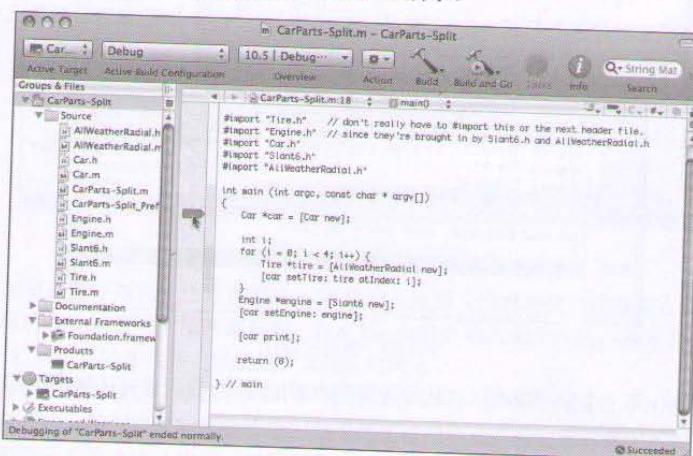
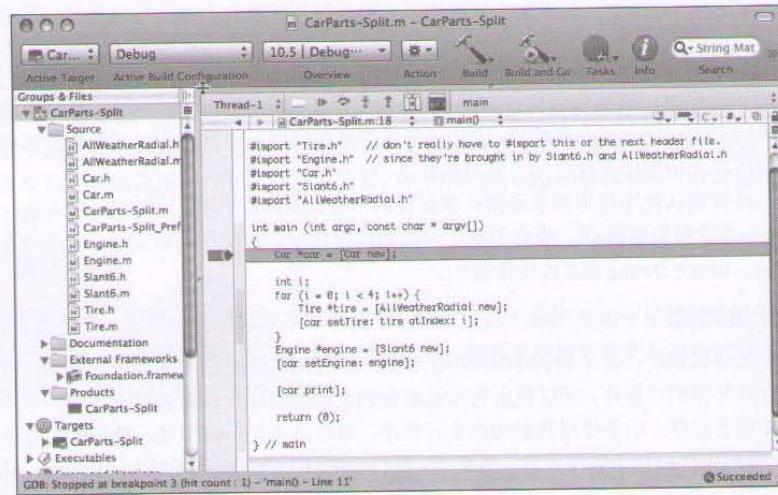


图7-28 设置断点

你可以把断点拖出边列来删除它，也可以单击来禁用它。打开其上下文菜单会出现一系列选项。一定要选中一些内置断点。比如，你可以让Xcode和你交流！是的，我知道人们普遍认为程序员都是孤独的，但是有时候，你也需要听一听与你想法不一样的声音。

现在选择Run>Debug来运行程序。你的程序应该如图7-29所示的那样停在断点处。注意指向代码行的红色箭头，它就像是商业街地图上标明“您在这里”的标志。



```

#import <Engine.h> // since they're brought in by Slants.h and AllWeatherRadial.h
#import <Car.h>
#import <Slants.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 = [Slants new];
    [car setEngine: engine];

    [car print];
    return (0);
} // main

```

图7-29 您在这里

Xcode窗口底部的状态行写着：GDB: Stopped at breakpoint ...。你能看到导航条上方新出现了一条控制栏，如图7-30所示。

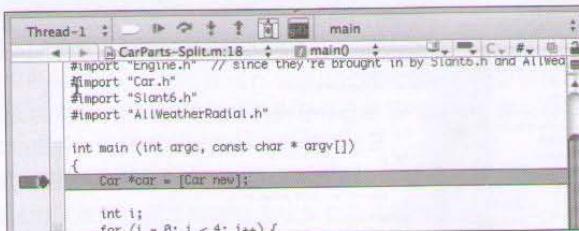


图7-30 调试器控件

从左边开始，第一个弹出菜单可以让你选择要查看的线程。我们暂时还接触不到多线程编程，所以现在可以先忽略它。

说明 多线程编程是一种同时处理多个执行流的编程方式，正确应用它是很困难的。通常，多线程编程所产生的错误非常难于找到。如果有人告诉你多线程编程很容易，那么他们不是被骗子了就是试图向你推销什么东西。

接下来的控件看起来像一个断点，它可以切换所有断点的开关状态。你可能会认为“嘿，我想修复了所有的错误”，这时，与其删除所有的断点，不如只是禁用它们然后再运行程序。当你发现了其他bug时，就可以开启这些断点并重新进行调试了。

下面的4个控件用来处理程序中接下来将会发生的事情。第一个控件看起来像CD机的开始按钮（想起来了吗？如果没有，可以去问你的父母）。它是继续按钮，你也可以使用快捷键 ⌘P 。单击它之后，程序会接着运行直到碰见下一个断点，然后结束或者崩溃。

第二个控件看起来像一个人正在跳过一个点，它是跳过按钮（也可以使用快捷键 ⌘O ）。单击它会执行一行代码，然后程序的控制权会交还给你。如果你单击了3次跳过按钮，那么“您在这里”的红箭头将会移到`-setTireAtIndex`调用那一行，如图7-31所示。

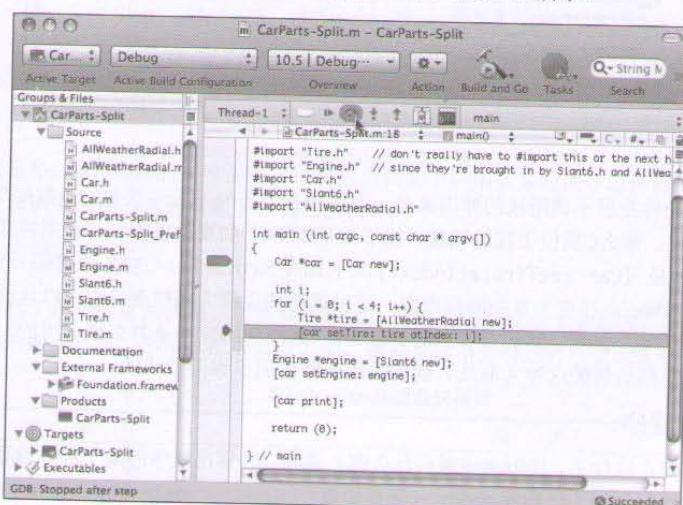


图7-31 单步执行之后

第三个按钮，向下指向一个点的箭头，它是跳入按钮（也可以使用快捷键 ⌘I ）。如果程序当前光标所在的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”的箭头设置在代码起始位置，如图7-32所示。

第四个按钮是跳出按钮（快捷键 ⌘T ），单击它会终止当前运行的函数并且程序会停在调用下一行代码，控制权又回到你手中。如果你正在跟着我实验，那么先不要单击这个按钮，我们还要看一看这个方法中的一些数据值。

继续介绍控制栏，下一个按钮（有一个喷雾罐的方框）是用来打开Xcode调试窗口的。再下一个按钮用来打开GDB控制台，你可以在这个调试器中直接输入调试命令。

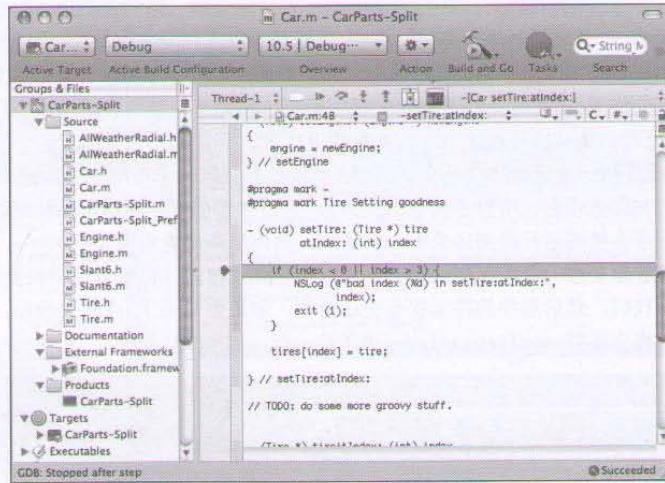


图7-32 跳入一个方法之后

最后一个控件是显示调用栈的弹出菜单。调用栈是当前处于活动状态的函数的集合。如果调用B，B调用C，那么C就位于栈的顶部，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 setTireAtIndex:]，下面是main函数。这就说明main函数调用了-setTireAtIndex:。在更为复杂的程序中，这种调用栈也称为栈跟踪，它可以包含许多方法函数。在调试会话中，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步执行时，就说明你想了解程序状态——变量的值。

Xcode有数据提示功能，类似于告诉你鼠标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7-33所示。

在图7-33中，我们将鼠标停在index上。果然，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tires

处，直到箭头展开，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，所以没有什么可看的。但是如果类中含有实例变量，它们将会显示出来并且可以编辑。你可以在图7-34中看到上述鼠标移动和悬停的结果。

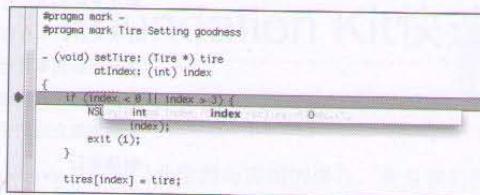


图7-33 Xcode数据提示

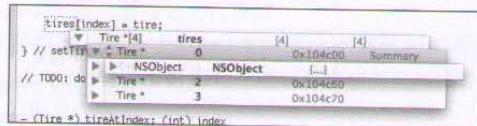


图7-34 输入程序中的数据

以上就是Xcode调试器的快速讲解。这些信息再加上你用大量时间学习所得的知识，应该可以让你应付调试中遇到的任何问题了。调试愉快！

7.5 备忘表

本章提到了很多键盘快捷键。之前我承诺过，会把它们做成一个备忘表，如表7-1所示。在寄这本书送给别人之前，你可以把本页撕下来，前提是你不觉得这样做是无礼的。

表7-1 Xcode键盘快捷键

按 键	描 述
⌘ ⌘ E	扩展编辑器
⌘ [左移代码块
⌘]	右移代码块
Tab	接受代码提示
Esc	显示代码提示菜单
Control-. (句点)	循环浏览代码提示
Shift-control-. (句点)	反向循环浏览代码提示
Control-/	移动到代码提示中的下一个占位符
Command-control-S	创建快照
Control-F	前移光标

(续)

按 键	描 述
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	将插入点置于窗口正中
⌘D	显示Open Quickly窗口
⌘↑	打开配套文件
⌘D	添加书签
Option - 双击	在文档中搜索
⌘Y	以调试方式运行程序
⌘P	继续（在调试器中）
⌘O	跳过
⌘I	跳入
⌘T	跳出

7.6 小结

本章包含的信息非常多，而且我们并没怎么谈及Objective-C语言。这是怎么回事呢？就像木工需要知道除了木头之外的知识（比如，他们得非常了解自己使用的工具）一样，Objective-C程序员需要了解的也不仅仅只是这个语言。能够在Xcode中熟练的编写、浏览和调试代码就意味着你可以用更少的时间与语言环境“斗争”，而把更多的时间放在更有趣的编程上。

接下来我们将会详细介绍Cocoa中的一些类。这应该会很有趣！

Foundation Kit快速教程

你已经知道Objective-C是一门非常精巧实用的语言，并且我们还没有研究完它所提供的全部功能。但是现在，让我们先做一次“短途旅行”，快速了解一下Cocoa的Foundation框架。尽管Foundation框架只是Cocoa的一部分，并且没有内置于Objective-C语言中，但是它依然十分重要，我们认为在本书中有必要对它进行讲解。

正如你在第2章所看到的，实际上Cocoa是由两个不同的框架组成的：Foundation Kit和Application Kit。Application Kit包含了所有的用户接口对象和高级类，第14章将会介绍AppKit（时髦的小孩这么称呼它）的相关内容。

Cocoa Foundation框架中有很多有用的、面向数据的低级类和数据类型，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，查看Xcode自带的文档可以了解它们。这些文档存放在/Developer/ADC Reference Library/documentation/index.html中。

在我们继续之前，有一个注意事项适用于本章及以后章节所提到的项目。我们依然会创建Foundation工具项目，但是只使用样本代码，并且样本代码跟在如下代码段之后（重新调整了格式，以适应本书的页面）：

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

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool
    = [[NSAutoreleasePool alloc] init];
    // insert code here...
    NSLog(@"Hello, World!");

    [pool drain];
    return 0;
}
```

让我们看一下这段代码。首先，main()函数创建（通过alloc）并初始化（通过init）了一个NSAutoreleasePool实例。在main()函数结尾，这个池被排空。这就是Cocoa内存管理的预览，我们将会在下一章中谈及。所以现在，请点头微笑，暂时不要考虑NSAutoreleasePool。如果删去NSAutoreleasePool也可以，不过在程序运行时，你会得到一些奇怪的消息。

8.1 一些有用的数据类型

在深入研究Cocoa的类之前，让我们先看看Cocoa为我们准备的一些结构体（**struct**）。

8.1.1 范围的作用

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“Objective- C is a cool language”中，单词“cool”可以用**location**为17，**length**为4的范围来表示。也许由于**location**字段未被初始化，所以它的值可以是**NSNotFound**，用来表示无意义范围。

可以用3种方式创建新的**NSRange**。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（这个名称让人印象深刻吧）：

```
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当成参数传递：

```
[anObject flarbulateWithRange: NSMakeRange (13, 15)];
```

8.1.2 几何数据类型

你会经常看到用来处理几何图形的数据类型，如**NSPoint**和**NSSize**。**NSPoint**代表的是笛卡尔平面中的一个点(x,y)：

```
typedef struct _NSPoint {
    float x;
    float y;
} NSPoint;
```

NSSize用来存储长度和宽度：

```
typedef struct _NSSize {
    float width;
    float height;
} NSSize;
```

在之前的Shapes程序中，我们本可以用一个**NSPoint**和一个**NSSize**来表示形状，而不是用自定义的表示矩形的**struct**来表示形状。但是当时，我们是想让程序尽可能简单。Cocoa提供了一

个矩形数据类型，它是由点和大小复合而成的：

```
typedef struct _NSRect {
    NSPoint origin;
    NSSize size;
} NSRect;
```

Cocoa也为我们提供了创建这些数据类型的快捷函数：NSMakePoint()、NSMakeSize()和NSMakeRect()。

说明 为什么这些数据类型是C的**struct**而不是对象呢？原因归结起来就是因为性能。程序（尤其是GUI程序）会用到许多临时的点、大小和矩形来完成它们的工作。还记得吧，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高的操作，它会消耗大量的时间。所以将这些结构建成第一等级的对象都会在使用过程中增加大量的系统开销。

8.2 字符串

8

我们第一个要介绍的真正的类是**NSString**，也就是Cocoa中用来处理字符串的类。字符串就是一组人类可读的字符序列。由于计算机通常是与人类交互的，所以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，其指示标志是双引号内的字符串前的@，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。

如果你用C语言处理过字符串，像*Learn C on the Mac* (Apress, 2009)一书中所涉及的字符串处理，你会知道这种处理令人非常痛苦。C语言将字符串作为简单的字符数组处理，并且在数组最后添加尾部零字节作为结束标志。而Cocoa中的**NSString**则有很多内置方法，它们让字符串的处理变得简单很多。

8.2.1 创建字符串

我们已经看到过诸如**printf()**和**NSLog()**之类的函数，它们接收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”。

8.2.2 类方法

stringWithFormat:的声明中有两个值得讨论的地方。第一个是定义最后的省略号(...)，它

告诉我们（和编译器）这个方法可以接收多个以逗号隔开的其他参数，就像`printf()`和`NSLog()`一样。

`stringWithFormat:`的另一个古怪却更重要的地方是声明中非常特别的开始字符：一个加号。这是怎么回事？Objective-C运行时生成一个类的时候，它会创建一个代表该类的类对象。类对象包含了指向超类的指针、类名和指向类方法列表的指针。类对象还包含一个`long`型数据，为新创建的类实例对象指定大小（以字节为单位）。

如果你在声明方法时添加了加号，那么就是把这个方法定义为类方法。这个方法属于类对象（而不是类的实例对象）并且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。

`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中的`NSColor`类有一些以不同颜色命名的类方法，如`redColor`和`blueColor`。要用蓝色绘图，可以像这样编写代码：

```
NSColor *haveTheBlues = [NSColor blueColor];
```

你所创建的大部分方法都是实例方法，要用前导减号（-）来开始声明。这些方法将会在某个对象实例中运行，比如获取一个`Circle`的颜色或者一个`Tire`的气压。如果某方法用来实现常规功能，比如创建一个实例对象或者访问一些全局类数据，那么最好使用前导加号（+）将它声明为类方法。

8.2.3 关于大小

`NSString`中另一个方便的方法（实例方法）是`length`，它返回的是字符串中的字符个数。

```
- (unsigned int) length;
```

可以这样使用它：

```
unsigned int 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`方法能够准确无误地处理国际字符串，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非常头疼的事情，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。

8.2.4 比较的策略

比较是字符串间常见的操作。有时，你会想知道两个字符串是否相等，（例如，`username`等

于'markd'吗？）而有时，你也会想要看看两个字符串可以怎样排列，这样就可以给姓名列表排序了。NSString提供了几个用于比较的方法。

`isEqualToString:`可以用来比较接收方（接收消息的对象）和当作参数传递来的字符串。`isEqualToString:`返回一个BOOL（YES或NO）型数据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;
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，声明如下：

```
- (NSComparisonResult) compare: (NSString *) string;
```

`compare:`将接收对象和传递来的字符串逐个字符地进行比较，它返回一个`NSComparisonResult`（就是一个enum型数据）来显示比较结果：

```
typedef enum _NSComparisonResult {
    NSOrderedAscending = -1,
    NSOrderedSame,
    NSOrderedDescending
} NSComparisonResult;
```

8

正确比较字符串

比较两个字符串是否相等时，应该用`isEqualToString:`，而不能仅仅只是比较字符串的指针值，例如：

```
if ([thing1 isEqualToString: thing2]) {
    NSLog(@"The strings are the same!");
}
和
if (thing1 == thing2) {
    NSLog(@"They are the same object!");
}
```

是不同的。

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。

有时，我们想检查两个对象的标识：`thing1`和`thing2`是同一个对象吗？这时就应该使用运算符`==`。如果是想查看等价性（即这两个字符串是否代表同一个事物吗），那么请使用`isEqualToString:`。

如果你曾经用过C语言中的函数qsort()或bsearch(), 那么这看起来也许会比较熟悉。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compare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。例如, [@"aardvark" compare:@"zygote"]将会返回NSOrderedAscending:。

同样, [@"zoinks" compare: @"jinkies"]将会返回NSOrderedDescending。并且, 如你所料, [@"fnord" compare: @"fnord"]返回的是NSOrderedSame。

8.2.5 不区分大小写的比较

compare:进行的是区分大小写的比较。换句话说, @"Bork"和@"bork"的比较是不会返回NSOrderedSame的。我们还有一个方法compare:options:, 它能给我们更多的控制权:

```
- (NSComparisonResult) compare: (NSString *) string
                           options: (unsigned) mask;
```

options参数是一个位掩码。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。
- NSNumericSearch: 比较字符串的字符个数, 而不是字符值。如果没有这个选项, “100”会排在“99”的前面, 这会让人觉得奇怪, 甚至也可能是错误的排序。

例如, 如果你想进行字符串比较, 要忽略大小写但按字符个数的多少正确排序, 那么应该这么做:

```
if ([thing1 compare: thing2
           options: NSCaseInsensitiveSearch
                  | NSNumericSearch]
    == NSOrderedSame) {
    NSLog (@"They match!");
}
```

8.2.6 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有另一个字符串。例如, 你也许想知道某个文件名的结尾是否是“.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, 判断它是否是文档的草稿版。这里有两个方法能帮助你判断: 第一个检查字符串是否以另一个字符串开头, 第二个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`将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` struct来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;
range = [fileName rangeOfString: @"chapter"];
```

返回的`range.start`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`range.start`则等于`NSNotFound`。

8.3 可变性

8

`NSString`是不可变的, 这并不意味着你不能操作它们。不可变的意思是一旦`NSString`被创建, 我们就不能改变它。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, 查找字符串将它与其他字符串比较, 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。

注意 Java程序员应该很熟悉这种区别。`NSString`就像Java中的`String`类一样, 而`NSMutableString`则与Java中的`StringBuffer`类很相似。

你可以使用类方法`stringWithCapacity:`来创建一个新的`NSMutableString`, 声明如下:

```
- (id) stringWithCapacity: (unsigned) capacity;
```

这个容量只是给`NSMutableString`的一个建议, 就像告诉小孩什么时候要回家一样。字符串并不仅限于所提供的容量, 这个容量仅是个最优值。例如, 如果你知道你要创建一个大小为42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会快很多。可按如下方式创建一个新的可变字符串:

```
NSMutableString *string;
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, 就像这样:

```
- (void) appendString: (NSString *) aString;
- (void) appendFormat: (NSString *) format, ...;
```

`appendString`接受参数`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, 但它将格式化的字符串附加在接收字符串的末尾, 而不是创建新的字符串对象, 例如:

```
NSMutableString *string;
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!”。

你可以用`deleteCharactersInRange:`方法删除字符串中的字符:

```
- (void) deleteCharactersInRange: (NSRange) range;
```

你会经常将`deleteCharactersInRange:`和`rangeOfString:`连在一起使用。记住,`NSMutableString`是`NSString`的子类。通过神奇的OOP方法, 你也可以在`NSMutableString`中使用`NSString`的所有功能, 包括`rangeOfString:`、字符串比较方法和其他任何功能。例如, 假设你列出了所有朋友的列表, 但是你又觉得你不喜欢Jack了, 想要把他从列表删除:

首先, 创建朋友列表:

```
NSMutableString *friends;
friends = [NSMutableString stringWithCapacity: 50];
[friends appendString: @"James BethLynn Jack Evan"];
```

接下来, 找到Jack的字符范围:

```
NSRange jackRange;
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从圣诞卡列表中删除了:

```
[friends deleteCharactersInRange: jackRange];
```

这样, 这个字符串就变成了“James BethLynn Evan”。

在实现描述方法时, 使用可变字符串是非常方便的。你可以通过`appendString`和`appendFormat`为对象创建一个详尽的描述。

由于`NSMutableString`是`NSString`的子类, 所以我们“免费”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。任何接受`NSString`的方法也都会接受`NSMutableString`。字符串的使用者其实并不关心它是否可变。

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`。你可以简单地以某种格式来创建一个可变字符串:

```
NSMutableString *string;
string = [NSMutableString stringWithFormat: @"jo%dy", 2];
```

`string`的起始值是“jo2y”, 但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

在特定的位置插入字符。可以查阅关于NSString和NSMutableString的文档来学习这两个类中所有方法的全部细节。

8.4 集合家族

随处可见的对象是很好玩，但是通常我们会希望它们更有条理。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。

8.4.1 NSArray

你应该用过C语言中的数组。事实上，在本书前面的内容中，我们也用到了数组来存储汽车的4个轮胎。你也许还记得在编写那段代码时我们遇到了一些困难。例如，我们不得不去检查数组的索引来确认它是否有效：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，也就是说我们的车的轮胎不能多于4。当然，这看起来似乎不是什么问题，但是你永远也不会知道那种被预言会研发出来的未来导弹车是否会需要多于4个的轮胎才能平稳降落。

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或者是其他你想要存储的对象。

只要有了一个NSArray，就可以通过各种方式来操作它，例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，以及其他许多操作。

NSArray有两个限制。首先，它只能存储Objective-C的对象，而不能存储C语言中的基本数据类型，如int、float、enum、struct，或者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（顺便说一下，这就是不能在数组中存储nil的原因之一）：

```
NSArray *array;
array = [NSArray arrayWithObjects:
        @"one", @"two", @"three", nil];
```

上面的代码创建了一个由NSString字面量对象组成的3个元素的数组。只要有了一个数组，就可以获得它所包含的对象个数：

```
- (unsigned) count;
并且你也可以获取特定索引处的对象:
- (id) objectAtIndex: (unsigned int) index;
结合这两个操作就可以输出数组的内容:
int i;
for (i = 0; i < [array count]; i++) {
    NSLog(@"index %d has %@", i, [array objectAtIndex: i]);
```

输出的结果应该像这样：

```
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。例如，运行这行代码：

```
[array objectAtIndex: 208000];
```

你将会看到：

```
*** Terminating app due to uncaught exception 'NSRangeException',
reason: '*** -[NSCFArray objectAtIndex:]: index (208000) beyond bounds (3)'
```

就是这样。

进行Cocoa编程时，你可能经常会看到这样的提示信息，所以，让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)。异常是Cocoa说明“我不知道该如何处理”的方式。在编码时，有许多捕获异常并且自己处理该异常的方法，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。接受参数的方法是NSCFArray objectAtIndex:。NSCFArray看起来很像NSArray，这也暗示了出错的位置。

说明 在Cocoa中，当你看到字符“CF”时，就看到了与苹果公司的Core Foundation框架相关的内容。Core Foundation框架与Cocoa一样，但它是用C语言实现的，它的大部分代码都是开源的，你可以下载下来查看它们。Core Foundation框架中的许多对象和Cocoa对象之间是免费桥接的，就是说它们是可以互换使用的。这里的NSCFArray是苹果公司实现的NSArray，但却使用CFArray执行具体操作。

异常消息的第二行更有趣，它指出我们在读取数组中索引为208 000的数据，但是该数组只有3个元素(差了那么多)。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。

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Run窗口中的这条消息。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。为此可选择Run>Show>Breakpoints。之后你将会看到一个窗口，其中显示所有当前断点(断点就是Xcode调试器中止程序运行，以便让你查看程序代码的地方)、当前项目的断点和应用于全局的断点，如图8-1所示。

下面我们会添加两个断点，它们能让我们更轻松地追查异常。首先，我们将为-[NSException raise]创建一个断点。选择Global Breakpoints，双击Double-Click for Symbol框，输入-[NSException raise]，然后按return键。你的断点窗口应该如图8-2所示。

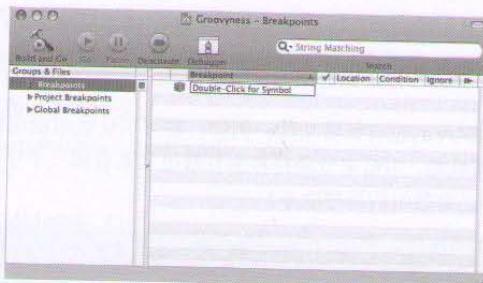


图8-1 Xcode的断点窗口

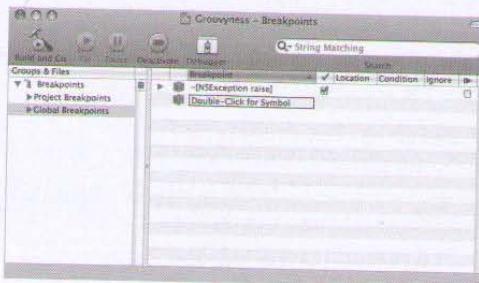


图8-2 添加了断点-[NSException raise]后

8

接下来添加另一个全局断点`objc_exception_throw`。现在，运行的程序有异常抛出时，调试器就会中止程序并且指向有问题的代码行，如图8-3所示。你可能需要单击栈跟踪窗格（图8-3中左上方的窗格）来将焦点移动到相应的源文件。在这个示例中，我们单击栈跟踪列表中的`main`函数来查看代码。

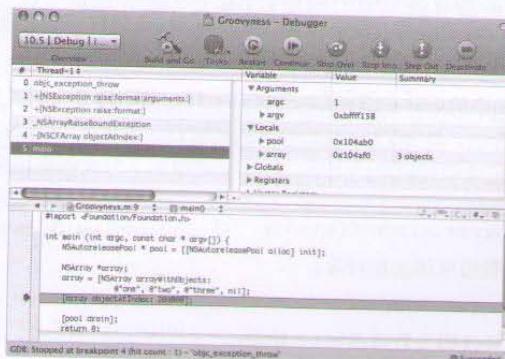


图8-3 指向问题代码行的Xcode调试器

你也许会不太高兴,因为你的程序仅仅只是超过了数组的界限就导致Cocoa有这么大的反应。但是相信我:你会意识到这其实是一个强大的功能,因为它允许你捕获那些可能无法检查出来的错误。

与NSString一样,NSArray也有很多功能。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,用给定分隔符将所有数组元素合成一个字符串(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:来切分NSArray,像这样:

```
NSString *string = @"oop:ack:bork:greeble:ponies";
NSArray *chunks = [string componentsSeparatedByString: @":"];
```

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。

8.4.2 可变数组

与NSString一样,NSArray创建的是不可变对象的数组。一旦你用特定数量的对象创建了一个数组,那么它就固定下来了: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(例如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。

于是,出现了NSArray的补充类NSMutableArray,这样我们就可以随意地添加和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: (unsigned) 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Cocoa能够对代码进行一些优化。Cocoa既不会将对象预写入数组,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;
array = [NSMutableArray arrayWithCapacity: 17];
```

使用addObject:在数组末尾添加对象:

```
- (void) addObject: (id) anObject;
```

你可以使用如下循环代码为数组添加4个轮胎:

```
for (i = 0; i < 4; i++) {
    Tire *tire = [Tire new];
```

```
[array addObject: tire];
}
```

你还可以删除特定索引处的对象。例如，如果你不喜欢第二个轮胎，可以用`removeObjectAtIndex:`来删除它。下面是方法定义：

```
- (void) removeObjectAtIndex: (unsigned) index;
```

像这样使用它：

```
[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，这同C数组一样。

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。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，`NSArray`还提供了大量好用的功能。

8.4.3 枚举“王国”

对数组中的每个元素执行操作是`NSArray`的常见操作。例如，如果你喜欢绿色可以将数组中的所有形状对象的颜色都改成绿色。你也可以给汽车的每个轮胎放一些气，以搭建适合匹兹堡地区路况的驾驶模拟器。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`，它是Cocoa用来描述这种集合迭代运算的方式。要想使用`NSEnumerator`，需通过`objectEnumerator`向数组请求枚举器：

```
- (NSEnumerator *) objectEnumerator;
```

你可以按如下方式使用这个方法：

```
NSEnumerator *enumerator;
enumerator = [array objectEnumerator];
```

如果你想要从后向前浏览集合，还有一个方法`reverseObjectEnumerator`可以使用。

在获得枚举器之后，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;
enumerator = [array objectEnumerator];

id thingie;
while (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

数组容器。如果你这么做了，枚举器就会觉得困惑，而你将会得到未定义结果。“未定义结果”可以表示“嘿，我已经操作过它了”，也可以是“噢，它让我的程序崩溃了”。

8.4.4 快速枚举

在Mac OS X 10.5 (Leopard) 中，为了将Objective-C升级到2.0版本，苹果公司对其进行了一系列的小调整。我们要研究的第一个调整叫做快速枚举，它的语法与脚本语言的类似。如下面代码所示：

```
for (NSString *string in array) {
    NSLog(@"%@", string);
}
```

这个循环体将会遍历数组中的每个元素，并且用变量`string`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的所有新特性一样，快速枚举也不能在Tiger (Mac OS X 10.4) 系统上使用。如果你或你的用户需要在Tiger系统上运行程序，那么就不能使用这个新语法。这真是太糟糕了。

好了，现在我们有3种方式可遍历数组了：通过索引、使用`NSEnumerator`和快速枚举。你使用的是哪个方法？

如果你只会在Leopard或更高版本的操作系统上运行程序，那么请使用快速枚举，因为它更简洁快速。

如果你的程序还需要支持Tiger系统，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`，例如跳跃浏览数组时（如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。

8.4.5 NSDictionary

你肯定听说过字典，可能你也使用过。在编程中，字典就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`在给定的关键字（通常是一个`NSString`字符串）下存储一个数值（可以是任何类型的对象）。然后你就可以用这个关键字来查找相应的数值。所以，（例如）如果你有一个存储了某人所有联系信息的`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存储方式。它可以立即找出要查询的数据，而不需要遍历整个数组进行查找。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。实际上字典非常快。

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。开始学习使用字典的最简单的方法就是用类方法`dicti-`

aryWithObjectsAndKeys:来创建字典。

```
+ (id) dictionaryWithObjectsAndKeys:
    (id) firstObject, ...;
```

该方法接受对象和关键字交替存储的系列，以nil值作为终止符号（可能猜到了，不能在NSDictionary中存储nil值）。假设我们想创建一个存储汽车轮胎的字典，轮胎用人类可读的标签表示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;

tires = [NSDictionary dictionaryWithObjectsAndKeys:
    t1, @"front-left", t2, @"front-right",
    t3, @"back-left", t4, @"back-right", nil];
```

使用方法objectForKey:来获取字典中的值，向方法传递之前用来存储该值的关键字：

```
- (id) objectForKey: (id) aKey;
```

所以要查找右后边的轮胎，可以这样写：

```
Tire *tire = [tires objectForKey: @"back-right"];
```

8

如果字典里没有右后边的轮胎（假设它是个古怪的三轮车），objectForKey:会返回nil值。

要创建新的NSMutableDictionary对象，向类NSMutableDictionary发送dictionary消息。你也可以使用dictionaryWithCapacity:方法来创建新的可变字典并且告诉Cocoa该字典的最终大小（你开始注意到了吗？Cocoa的命名系统非常正规）。

```
+ (id) dictionaryWithCapacity: (unsigned int) numItems;
```

与我们之前提到过的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。

可以使用setObject:forKey:方法给字典添加元素：

```
- (void) setObject: (id) anObject forKey: (id) aKey;
```

下面是另一种创建存储轮胎的字典的方法：

```
NSMutableDictionary *tires;
tires = [NSMutableDictionary dictionary];

[tires setObject: t1 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:，那么这个方法将会用新值替换原有数值。如果你想在可变字典中删除一个关键字，可使用removeObjectForKey:方法：

```

- (void) removeObjectForKey: (id) aKey;
所以, 如果我们想模拟一只轮胎脱落的场景, 就可以把那只轮胎删除:
[tires removeObjectForKey: @"back-left"];

```

8.4.6 使用, 但不要扩展

你可能很有创造力, 试图去创建 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。举一个例子, 回想本章前面所遇到的问题, 当我们编制的索引超过了数组范围而得到一个异常消息时, 实际上在消息中出现的类是 NSCFArray。我们可以回到前面去看一看。

NSString 或 NSArray 的使用者不用在意系统内部到底用的是哪个类。但是给一个类簇创建子类是一件非常痛苦和令人沮丧的事情。通常, 你可以通过将 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) 来解决这种编程问题, 而不用创建子类。

8.5 各种数值

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 或 struct。但是你可以用对象来封装基本数值。例如, 将 int 型数据封装到一个对象中, 然后就可以将这个对象放入 NSArray 或 NSDictionary 中了。

8.5.1 NSNumber

Cocoa 提供了 NSNumber 类来包装 (即以对象形式实现) 基本数据类型。你可以使用下列类方法创建新的 NSNumber 对象:

```

+ (NSNumber *) numberWithChar: (char) value;
+ (NSNumber *) numberWithInt: (int) value;
+ (NSNumber *) numberWithFloat: (float) value;
+ (NSNumber *) numberWithBool: (BOOL) value;

```

还有许多这种创建方法, 包括无符号版本和各种 long 型数据及 long long 整型数据。但是上面的 4 个是最常用的方法。创建 NSNumber 之后, 你可以把它放入一个字典或数组中:

```

NSNumber *number;
number = [NSNumber numberWithInt: 42];
[array addObject: number];
[dictionary setObject: number forKey: @"Bork"];

```

只要将一个基本类型数据封装到 NSNumber 中后, 就可以通过下面的实例方法重新获得它:

```

- (char) charValue;
- (int) intValue;
- (float) floatValue;
- (BOOL) boolValue;
- (NSString *) stringValue;

```

将创建方法和提取方法配对使用是完全可以的。例如，用`numberWithFloat:`创建的`NSNumber`可以用`intValue`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包装成对象叫做装箱（boxing），从对象中提取基本类型的数
据叫做取消装箱（unboxing）。有些语言有自动装箱功能，它可以自动包装基础类型的数据，
也可以自动从包装后的对象中提取基础数据。Objective-C语言不支持自动装箱。

8.5.2 NSValue

`NSNumber`实际上是`NSValue`的子类，`NSValue`可以包装任意值。你可以用`NSValue`将结构放入`NSArray`和`NSDictionary`中。可使用下面的类方法创建新的`NSValue`：

```
+ (NSValue *) valueWithBytes: (const void *) value
    objCType: (const char *) type;
```

传递的参数是你想要包装的数值的地址（如一个`NSSize`或你自己的`struct`）。通常，得到的是你想要存储的变量的地址（在C语言中使用操作符`&`）。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`编译器指令可以接受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`NSRect`放入`NSArray`中：

```
NSRect rect = NSMakeRect (1, 2, 30, 40);

NSValue *value;
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 *) value;
```

调用`getValue:`时，要传递的是要存储这个数值的变量的地址：

```
value = [array objectAtIndex: 0];
[value getValue: &rect];
```

说明 在上面的`getValue:`例子中，你可以看到方法名中使用了`get`，它表明我们提供的是一个指针，而指针所指向的空间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) point;
+ (NSValue *) valueWithSize: (NSSize) size;
+ (NSValue *) valueWithRect: (NSRect) rect;

- (NSPoint) pointValue;
```

```

- (NSSize) sizeValue;
- (NSRect) rectValue;

可按下面的方式在NSArray中存储和检索NSRect:
value = [NSValue valueWithRect: rect];
[array addObject: value];
...
NSRect anotherRect = [value rectValue];

```

8.5.3 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，假设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电话号码，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就可以消除这种歧义。你可以设定关键字@"home fax machine"下的NSNull值代表的是这个人没有传真机，而关键字没有数值则代表你不知道他是否有传真机。

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"];
```

访问它的方法如下所示：

```
id homefax;
homefax = [contact objectForKey: @"home fax machine"];

if (homefax == [NSNull null]) {
    // ... no fax machine. rats.
}
```

[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。

8.6 示例：查找文件

好了，我们已经说了一大堆的理论知识了，也详细了解了以NS开头的各种类。下面是一个真实的工作程序，它用到了本章中所讲到的一些类。程序FileWalker（可以在项目文件夹08-01 FileWalker中找到）将会翻查主目录，查找.jpg文件并输出找到的文件列表。我们得承认这个程序并不太令人兴奋，但是它确实实现了一些功能。

FileWalker程序用到了NSString、NSArray、NSEnumerator以及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，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为我们创建

NSDirectoryEnumerator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[])
{
    NSAutoreleasePool *pool;
    pool = [[NSAutoreleasePool alloc] init];

    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]
             isEqualToString: @"jpg"]) {
            [files addObject: filename];
        }
    }

    NSEnumerator *fileenum;
    fileenum = [files objectEnumerator];

    while (filename = [fileenum nextObject]) {
        NSLog(@"%@", filename);
    }

    [pool drain];
    return (0);
} // main
```

现在，让我们逐步分解这个程序。最上面的是自动释放池的样板代码（你将会在第9章看到它的详细介绍）：

```
NSAutoreleasePool *pool;
pool = [[NSAutoreleasePool alloc] init];
```

下一步是获取NSFileManager对象。NSFileManager中有一个叫做defaultManager的类方法，可以为我们创建一个属于我们自己的NSFileManager对象：

```
NSFileManager *manager;
manager = [NSFileManager defaultManager];
```

这在Cocoa中是很常见的，很多类都是单实例架构，即只需要一个实例。你真的只需要一个文件管理器，或一个字体管理器，或一个图形内容。这些类都提供了一个类方法用来访问唯一的共享对象，你可以用它来完成你的工作。

在这个例子中，我们需要一个目录迭代器。但是在我们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/markd/，但这就有局限性了，该路径只有在主目录是markd时才有效。所幸，Unix系统（和Mac OS X系统）有一个代表主目录的速记符号~（也称为代字符）。是的，即使你没有输入西班牙语，这个符号也还是有用途的。~/Documents代表目录Documents，~/junk/oopack.txt在Mark的计算机上就是/Users/markd/junk/oopack.txt。NSString中有一个方法可以接受~字符并将其展开。接下来的两行代码中使用了这个方法：

```
NSString *home;
home = [@"~" stringByExpandingTildeInPath];
```

stringByExpandingTildeInPath将~替换成当前用户的主目录。在Mark的计算机上，主目录是/Users/markd。接下来，我们将路径字符串传递给文件管理器：

```
NSDirectoryEnumerator *direnum;
direnum = [manager enumeratorAtPath: home];
```

enumeratorAtPath:返回一个NSDictionaryEnumerator，它是NSEnumerator的子类。每次在这个枚举器对象中调用nextObject时，都会返回该目录中一个文件的另一个路径。这个方法也能搜索子目录。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还额外提供了一些功能，例如为每个文件创建一个属性字典，但是在这里我们不会用到。

由于我们要查找.jpg文件（即路径名称以“.jpg”结尾）并且要输出它们的名称，所以我们需要存储这些名称的空间。现在，当在枚举中遇到这些名称时，我们可以用NSLog()来直接进行输出。但是也许在以后，我们会想要在程序的其他地方对所有这些文件做一些操作。这样，NSMutableArray就是第一选择。我们创建一个可变数组并将匹配的路径添加进去：

```
NSMutableArray *files;
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，因为……嗯，你知道的。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。

最后是程序的真正内容。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，当枚举结束时它会返回nil值，于是循环终止。

`NSString`提供了许多处理路径名称和文件名称的便捷工具。例如，方法`pathExtension`输出文件的扩展名（去掉了扩展名前面的点）。所以为`oopack.txt`文件调用`pathExtension`将会返回`@"txt"`，而`vikkiCat.jpg`则会返回`@"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:`。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] isEqualToString: @"jpg"]) {
    [files addObject: filename];
}
```

目录循环结束后，遍历文件数组，用`NSLog()`将数组内容输出：

```
NSEnumerator *fileenum;
fileenum = [files objectEnumerator];
while (filename = [fileenum nextObject]) {
    NSLog(@"%@", filename);
}
```

下面，我们用自动释放池的样板代码做一些清理工作。最后，通知`main()`函数返回0来表示程序成功退出：

```
[pool drain];
return (0);
} // main
```

下面是在Mark的计算机上运行该程序的结果示例：

```
cocoaheds/DSCN0798.jpg
cocoaheds/DSCN0804.jpg
cow.jpg
Development/Borkware/BorkSort/cant-open-file.jpg
Development/Borkware/BSL/BWLog/accident.jpg
```

成功了！但是显示全部的结果可能会需要一些时间，因为程序也许不得不去翻查成千上万个图片文件来完成查询工作。

“小心 Leopard 系统”背后的含义

`FileWalker`程序使用的是典型的迭代方式，而08.02 `FileWalkerPro`则展示了使用快速枚举的方法。快速枚举语法有一个很好的特性，即你可以将已有的`NSEnumerator`对象或其子类传递给它。而正好`NSDictionaryEnumerator`是`NSEnumerator`的子类，所以我们可以放心地将`-enumerator-AtPath:`的结果传递给快速枚举：

```
int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSFfileManager *manager;
    manager = [NSFileManager defaultManager];
```

```
NSString *home;
home = [@"~" stringByExpandingTildeInPath];

NSMutableArray *files;
files = [NSMutableArray arrayWithCapacity: 42];

for (NSString *filename
     in [manager enumeratorAtPath: home]) {
    if ([[filename pathExtension]
         isEqualToString: @"jpg"]) {
        [files addObject: filename];
    }
}
for (NSString *filename in files) {
    NSLog(@"%@", filename);
}
```

可以看到，这个版本的程序比之前的那个版本要简单：我们舍弃了两个枚举器变量及其支持代码。

8.7 小结

本章介绍了许多内容，包括3个新的语言功能：类方法，即由类本身而不是某个实例来处理的方法；`@encode()`指令，用于需要生成C型描述的方法；快速枚举。

我们研究了很多有用的Cocoa类，包括`NSString`、`NSArray`和`NSDictionary`。`NSString`存储人类可读的文本，而`NSArray`和`NSDictionary`存储对象的集合。这些对象都是不可变的：一旦被创建就不能被改动。Cocoa提供了这些类的可变版本，你可以随意更改它们的内容。

尽管我们已经很努力了（并且尽管本章篇幅不短），但是我们也只是认识了Cocoa几百个类中的一小部分。你可以查阅资料，学习更多有关这些类的知识并获得乐趣及智慧。

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，你将学到在系统垃圾生成后如何清理程序。

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，这些资源包括内存、打开文件数量以及网络连接等。如果你使用了某种资源，例如因打开文件而占用了资源，那么你需要随后对其进行清理（在此例中是通过关闭文件实现的）。如果你不断打开文件并且保持打开状态而从不关闭，最终将无法再打开新的文件。设想一下公共图书馆的场景。如果每个人都只借不还，最终图书馆将会因无书可借而倒闭，每个人都无法再使用图书馆。任何人都不希望出现这种结果。

诚然，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，程序将可能崩溃。

虽然不是每个程序都会使用文件或网络连接，但是每个程序都会消耗内存。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无需考虑此类问题：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，而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，则将发生内存泄漏：程序的内存占用不断增加，最终会被耗尽并导致程序崩溃。同样需要我们注意的是，不要使用任何刚释放的内存。否则我们将可能误用陈旧的数据，从而引起各种各样的错误，而如果该内存已经加载了其他数据，我们将破坏这些新数据。

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要些时日。即使是有几十年经验的程序员，第一次阅读这些内容时也有一定的困难，因此，如果你一时未能完全读懂，请不必担心。

如果只打算在Leopard或更高版本的Mac OS X操作系统上运行你的程序，可以利用Objective-C 2.0的垃圾回收机制，我们将在本章末尾讨论这个问题。你可以直接跳到本章末尾阅读这部分内容。如果希望程序在旧版本的Mac OS X操作系统上运行，或者要从事iPhone开发，则有必要阅读本章的全部内容。

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(接收消息和执行操作)、交友(借助方法的组合和参数)以及当它们的生命结束时最终死去(被释放)。当对象的生命周期结束时,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,对于何时创建对象已经很清楚了,我们也讨论了关于如何使用对象的许多内容,但是我们怎么知道对象的使用寿命什么时候结束呢?Cocoa采用了一种称为引用计数(reference counting)的技术,有时也叫做保留计数。每个对象有一个与之相关联的整数,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值减1,表示它不再访问该对象。当保留计数器值为0时,表示不再有代码访问该对象了,(可怜的对象!)因此对象将被销毁,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(生成接收对象的一个副本)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值,可以给对象发送一条retain消息。要减少对象的保留计数器值,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。可以通过这种方法释放已经分配的全部相关资源。一定不要直接调用dealloc方法。可以利用Objective-C在需要销毁对象时调用dealloc方法。要获得保留计数器的当前值,可以发送retainCount消息。下面是retain、release和retainCount 3种方法的签名:

- (id) retain;
- (void) release;
- (unsigned) retainCount;

retain方法返回一个id类型的值。通过这种方式,可以嵌套执行带有其他消息发送参数的保留调用,增加对象的保留计数器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。下面的程序创建一个RetainTracker类的对象,该对象在初始化和销毁时调用 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，我们将在下一章讨论这种方式。正如前面所讲，当对象的保留计数器值归0时，将自动发送dealloc消息（其结果是调用dealloc方法）。在我们的例子中，init和dealloc这两个方法使用NSLog()输出一条消息来表明它们被调用。

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。当创建一个新的RetainTracker类对象时，Objective-C会发送retain消息和release消息，以增加和减少对象的保留计数器值，稍后我们就可以看到NSLog()函数有趣且规律的输出：

```
int main (int argc, const char *argv[])  
{  
    RetainTracker *tracker = [RetainTracker new];  
    // count: 1  
  
    [tracker retain]; // count: 2  
    NSLog(@"%@", [tracker retainCount]);  
  
    [tracker retain]; // count: 3  
    NSLog(@"%@", [tracker retainCount]);  
  
    [tracker release]; // count: 2  
    NSLog(@"%@", [tracker retainCount]);  
  
    [tracker release]; // count: 1  
    NSLog(@"%@", [tracker retainCount]);  
  
    [tracker retain]; // count 2  
    NSLog(@"%@", [tracker retainCount]);  
  
    [tracker release]; // count 1  
    NSLog(@"%@", [tracker retainCount]);  
  
    [tracker release]; // count: 0, dealloc it  
  
    return (0);  
}
```

```
 } // main
```

当然，在实际工作中，你不需要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，对象可能会像在本例子中一样历经由程序中不同调用引起的一系列保留和释放状态。让我们运行该程序来看看保留计数器值是如何变化的：

```
init: Retain count of 1.
2
3
2
1
2
1
dealloc called. Bye Bye.
```

因此，如果使用`alloc`、`new`或`copy`来操作一个对象，你只需要释放该对象以销毁它并收回它占用的内存。

9.1.2 对象所有权

你可能会想，“你不是说这很难吗？这有什么大不了的？我可以创建对象、使用对象、释放对象，内存管理也易如反掌，听起来没那么复杂。”但是，当你开始考虑对象所有权（object ownership）这一概念时，内存管理就变得更加复杂了。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。

如果一个对象具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，则称该函数拥有它创建的这个对象。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复杂了，这也是保留计数器值可能大于1的原因。在`RetainCount1`程序的例子中，`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`的`engine` setter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`Car`类。

解决办法是让`Car`类保留`engine`对象，将`engine`对象的保留计数器值增加到2。这是因为`Car`类和`main()`函数这两个实体都正在使用`engine`对象。`Car`类应该在`setEngine:`方法中保留`engine`对象，而`main()`函数应该释放`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

编写内存管理程序的第一种方法——setEngine方法的常见版本可能如下所示:

```
- (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,而会泄漏的引擎从来都不是什么好事。第一个engine对象将一直空转(对不起,我们不会再使用双关语了)并消耗大量内存。

9

下面是编写setEngine:方法的另一种方法:

```
- (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。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`对象的保留计数器值归0，并释放`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，这会引起错误。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，而`newEngine`与`engine`是同一个对象，保留计数器值将先增加，然后立即减少，但是不会归0，`engine`对象意外地未被销毁，从而导致错误。在访问方法中，如果先保留新对象，然后再释放旧对象，则不会出问题。

说明 关于应该如何编写适当的访问方法，存在不同的看法，各个邮件列表上基于半正则的争论进行得如火如荼。9.1.3节中给出的技术可以很好地工作，而且也比较易于理解。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，你也不必感到惊讶。

9.2 自动释放

内存管理是一个棘手的问题，前面已经讲过，我们在编写setter方法时遇到了一些细微的问题。接下来该解决另一个难题了。大家都知道，当我们不再使用对象时必须将其释放。但是，在某些情况下弄清什么时候不再使用一个对象并不容易。考虑下面这个`description`方法（该方法返回一个描述对象的`NSString`类型的值）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值设置为1），然后返回该字符串实例。请考虑一下，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，则保留计数器值归0，对象马上被销毁。

虽然使用`description`对象的代码可以继续将该字符串对象保留在一个变量中，等到执行结束时再释放它，但是那样做会导致`description`对象使用起来非常不便。实际上，只需将上例中

的一行代码改为如下所示的3行即可：

```
NSString *desc = [someObject description];
NSLog(@"%@", desc);
[desc release];
这就是更好的办法！
```

9.2.1 所有对象全部入池

Cocoa中有一个自动释放池（autorelease pool）的概念。你可能已经在由Xcode生成的样板代码中见到过`NSAutoreleasePool`。现在我们看看自动释放池到底是怎么回事。

顾名思义，它是一个存放实体的池（集合），这些实体可能是对象，能够被自动释放。`NSObject`类提供了一个`autorelease`方法：

```
- (id) autorelease;
```

该方法预先设定了一条在将来某个时间发送的`release`消息，其返回值是接收消息的对象。`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`NSAutoreleasePool`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，但是无需另起炉灶编写自动释放池——苹果公司已经替你完成了这项艰巨的任务。

9

因此，现在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]);
}
```

你可以编写如下代码：

```
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，自动释放池被自动销毁。

9.2.2 自动释放池的销毁时间

自动释放池什么时候被销毁，以便可以向其包含的所有对象发送`release`消息？而且，什么

时候首先创建自动释放池？在我们一直使用的Foundation库工具中，创建和销毁自动释放池的方法非常明确：

```
NSAutoreleasePool *pool;
pool = [[NSAutoreleasePool alloc] init];
...
[pool release];
```

创建一个自动释放池时，该池自动成为活动的池。释放该池时，其保留计数器值归0，然后该池被销毁。在销毁过程中，该池释放其包含的所有对象。

当使用AppKit时，Cocoa定期自动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（如鼠标单击或按键）以后执行这些操作。你可以使用任意多的自动释放对象，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 你可能已经在Xcode的自动生成代码中遇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销毁它。-drain方法只适用于Mac OS X 10.4 (Tiger)及更高版本。在我们自己编写（而不是由Xcode生成）的代码中，我们使用-release方法，因为该方法适用于Mac OS的所有版本。

9.2.3 自动释放池的工作过程

程序RetainTracker2展示了自动释放池的工作过程，它位于09-02 RetainTracker-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类相同，在初始化和释放对象时调用NSLog()函数。

```
RetainTracker2的main()函数如下：
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

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

首先，创建一个自动释放池：

```
NSAutoreleasePool *pool;
pool = [[NSAutoreleasePool alloc] init];
```

现在，我们任何时候向一个对象发送`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，一个新的`tracker`对象被创建，因为它在创建时接收了一条`new`消息，所以其保留计数器值为1。

```
[tracker retain]; // count: 2
```

下面，我们保留该对象（仅仅是出于好玩和演示目的），于是该对象的保留计数器值增加到2。

```
[tracker autorelease]; // count: still 2
```

然后该对象被自动释放，但是其保留计数器值保持不变：仍然为2。特别需要说明的是，我们之前创建的自动释放池现在仍有一个指向该对象的引用。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
```

接下来，释放该对象以抵消此前对它的保留操作。该对象的保留计数器值仍然大于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，与其他任何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，所以其保留计数器值为1。这条`release`消息将其保留计数器值减少为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，哪个先被调用？

运行`RetainTracker2`程序的输出结果如下：

```
init: Retain count of 1.
releasing pool
dealloc called. Bye Bye.
```

你可能已经猜到，释放自动释放池之前的`NSLog`函数比`RetainTracker`类中的`NSLog`函数先被调用。

9.3 Cocoa 内存管理规则

现在我们已经学习了这些方法：`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，可一致地应用于整个工具包。

说明 与将这些规则复杂化一样，忽略这些规则也是一种常犯的错误。如果你发现自己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。这意味着你需要放慢速度，休息一下，吃点零食，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值为1。当不再使用该对象时，你要负责向该对象发送一条`release`或`autorelease`消息。这样，该对象将在其使用寿命结束时被销毁。
- 当你通过任何其他方法获得一个对象时，则假设该对象的保留计数器值为1，而且已经被设置为自动释放，你不需要执行任何操作来确保该对象被清理。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，你需要（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。

大道至简，就这3条规则。

“如果我使用了`new`、`alloc`或`copy`方法获得一个对象，则我必须释放或自动释放该对象。”只要你记住了这条定律，你就平安无事了。

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？打算拥有该对象多长时间（参见表9-1）？

表9-1 内存管理规则

获得途径	临时对象	拥有对象
<code>alloc</code> / <code>new</code> / <code>copy</code>	不再使用时释放对象	在 <code>dealloc</code> 方法中释放对象
任何其他方法	不需要执行任何操作	获得对象时保留，在 <code>dealloc</code> 方法中释放对象

9.3.1 临时对象

下面我们来看看一些常见的内存管理生命周期的情景。在第一个例子中，你正在某些代码中使用（暂时）某个对象，但是你并未打算长期拥有该对象。如果你使用`new`、`alloc`或`copy`方法获得一个对象，则需要安排该对象的死亡，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
```

如果你使用任何其他方法获得一个对象，例如`arrayWithCapacity:`方法，则不需要关心如何销毁该对象：

NS
ar
//
ar
象被返
送reli
传
NS
cc
//
b1
为1且
都可以
需知道
9.3.2
通
量中保
量使用
如
的保
放该
-
{
}
}
}
如
GUI应
束以后
制
制
程
序
状
态。

```
NSMutableArray *array;
array = [NSMutableArray arrayWithCapacity: 17];
// count: 1, autoreleased
// use the array
```

arrayWithCapacity:方法不属于`alloc`、`new`、`copy`这3个方法中的一个，因此可以假设该对象被返回时保留计数器值为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3个方法，因此可以假设该对象的保留计数器值为1且已经被设置为自动释放。`blueColor`方法返回一个全局单例对象——每个需要访问它的程序都可以共享的单一对象，这个对象实际上永远不会被销毁，不过你不需要关心其实现细节。你只需要知道，你不需要显式地释放`color`。

9.3.2 拥有对象

通常，你可能希望在多个代码行中一直拥有某个对象。常见的方法是：在其他对象的实例变量中使用这些对象，将它们加入到诸如`NSArray`或`NSDictionary`等集合中，或者将其作为全局变量使用（更罕见的情况）。

如果你正在使用`new`、`alloc`或`copy`方法获得一个对象，则不需要执行任何其他操作。该对象的保留计数器值为1，因此它将一直被保留，只是一定要在拥有该对象的对象的`dealloc`方法中释放该对象。

```
- (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`以外的方法获得一个对象，你需要保留该对象。考虑编写GUI应用程序时事件循环的情况。你希望保留自动释放的对象，使这些对象在当前的事件循环结束以后仍能继续存在。

那么，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非常缓慢地作出决定（例如单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，程序被唤醒并开始工作，执行某些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该自动释放池。这样可以使累积的临时对象的数量保持在最低程度。

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（例如一个GUI程序）或自动释放池被销毁时，`flonkArray`对象将会接收到一条`release`消息，因而其保留计数器值从2减少到1。因为其保留计数器值大于0，所以该对象将继续存在。因此，我们需要在自己的`dealloc`方法中释放该对象，使该对象被清理。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：要么是在你自己的代码中显式地销毁，要么是在事件循环结束时使用AppKit销毁。你不必关心守护程序如何随机地销毁自动释放池。因为在调用函数的过程中自动释放池不会被销毁，所以你也不必保留使用的每一个对象。

清理自动释放池

有时，自动释放池未能按照通常预期的情况进行清理。这里有一个来自Cocoa邮件列表的常见问题：“虽然我已经自动释放了我使用的所有对象，但是我的程序占用的内存一直保持绝对的增长。”下面的代码通常会引起这样的问题：

```
int i;
for (i = 0; i < 1000000; i++) {
    id object = [someArray objectAtIndex: i];
    NSString *desc = [object description];
    // and do something with the description
}
```

该程序执行一个循环，在大量的迭代中每次都会生成一个（或2个、10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

内存空间占用，
该自动释放池。

对象将会接收
所以该对象将
。如果我们在
外地销毁。
式地销毁，要
释放池。因为

邮件列表的
一直保持绝

释放对象。
这个循环创
中，因此就
被销毁时才

最终获得释放。

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1 000次，就销毁当前的自动释放池并创建一个新的自动释放池。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 description];
    // and do something with the description
    if (i % 1000 == 0) {
        [pool release];
        pool = [[NSAutoreleasePool alloc] init];
    }
}
[pool release]
```

循环每执行1 000次，新的自动释放池就被销毁，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1 000个，从而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：当你创建一个新的自动释放池时，它将被添加到栈顶。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，则这个自动释放对象仍将存在，因为容纳该对象的自动释放池仍然存在。

9

9.3.3 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，当你忘记清理它们时，系统会自动识别哪些对象仍在使用，哪些对象可以回收。启用垃圾回收非常简单，只不过这是一种可选择启用的功能。只需转到项目信息窗口的Build选项卡，然后选择Required [-fobjc-gc-only]选项即可，如图9-1所示。

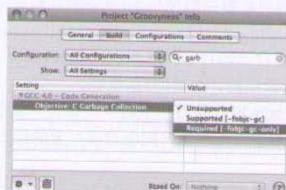


图9-1 启用垃圾回收

说明 “-fobjc-gc” 选项是为了使代码既支持垃圾回收又支持对象的保留和释放，例如两种环境都使用的库代码。

启用垃圾回收以后，通常的内存管理命令全都变成了空操作指令，不执行任何操作。

Objective-C的垃圾回收器是一种继承性的垃圾回收器。与那些已经存在了一段时间的对象相比，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以及它们之间的指针，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被丢弃的垃圾。最糟糕的事情莫过于保留一个指向不再使用的对象的指针。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），一定要在某个时候使该实例变量赋值为nil，以取消对该对象的引用并使垃圾回收器知道该对象可以被清理了。

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，如果不是编写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，你就不必再过多地关心内存管理问题。当使用从malloc函数（或利用Core Foundation库中的对象）回收的内存时，还有一些细微的差别，但是我们不讨论这些过于晦涩难懂的内容了。现在，你可以只考虑对象的创建而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，如果开发iPhone软件，则不能使用垃圾回收。实际上，在编写iPhone程序时，苹果公司建议你不要在自己的代码中使用autorelease方法，同时还要避免使用创建自动释放对象的便利函数。

9.4 小结

本章介绍了Cocoa的内存管理方法：retain、release和autorelease。

每个对象都维护一个保留计数器。对象被创建时，其保留计数器值为1；对象被保留时，保留计数器值加1；对象被释放时，保留计数器值减1；当保留计数器值归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。

当对象接收到一条autorelease消息时，其保留计数器值并不立即改变。相反，该对象只是被放入NSAutoreleasePool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值减1。如果保留计数器值归0，则对象被销毁。当使用AppKit时，Objective-C会在适当的时间为你创建和销毁自动释放池，例如在当前用户事件处理完毕时。除此以外，你要负责创建自己的自动释放池，Foundation库工具的模板包含了这些代码。

下面是Cocoa中关于对象及其保留计数器的3条规则。

- 如果使用new、alloc或copy操作获得一个对象，则该对象的保留计数器值为1。
- 如果通过任何其他方法获得一个对象，则假设该对象的保留计数器值为1，而且已经被设置为自动释放。
- 如果保留了某个对象，则必须保持retain方法和release方法的使用次数相等。

下一章将讨论init方法：如何使对象在创建之后即可使用。

10 对象初始化

到目前为止，我们已经讲了两种创建新对象的不同方法：第一种方法是[类名 new]，第二种方法是[[类名 alloc] init]。这两种方法是等价的，不过，通常的Cocoa惯例是使用alloc和init，而不使用new。一般情况下，Cocoa程序员只是在他们尚不具备足够的水平来熟练使用alloc和init方法时，才将new作为辅助方法使用。现在是时候摘掉辅助轮了。

10.1 分配对象

分配(allocation)是一个新对象诞生的过程。最美好的时刻，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消息的结果，就是为该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。由于未初始化内存而引起的各种各样的随机bug曾经折磨着许多语言，而alloc方法将分配的内存初始化为0，这样就不用担心此类问题了。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，所有的地址都属于我们了(对不起，不能自己了)。

一个刚刚分配的对象并不能立即使用，需要先初始化该对象，然后才能使用它。有些语言(包括C++和Java)，使用构造函数在单次操作中执行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。初学者常犯的一种错误就是只执行分配操作而未进行初始化，例如：

```
Car *car = [Car alloc];
```

这样的代码也能运行，但由于未进行初始化，随后可能表现出令人匪夷所思的行为(也叫bug)。本章其余的篇幅将全部用于讲解初始化这一重要概念。

10.2 初始化对象

与分配对应的操作是初始化。初始化(initialization)从操作系统取得一块内存，准备用于存储对象。init方法(即执行初始化操作的方法)几乎总是返回它们正在初始化的对象。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。

为什么程序员希望init方法返回不同的对象呢？如果还记得第8章末尾对类簇的讨论，你就会明白，像NSString和NSArray这样的类实际上只是大量专用类的虚假表象。由于init方法可以接受参数，所以该方法的代码能够检查其接受的参数，并断定返回另一个类的对象可能更合适。例如，从一个很长的字符串或一个阿拉伯文字符串生成一个新的字符串的情况。基于这一认知，字符串初始化函数可能决定创建一个属于其他类的对象（该对象更适合被期望的字符串的要求），然后返回该对象而不是原来的对象。

10.2.1 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显得有点奇怪。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
```

最奇怪的正是第一行：

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

这行代码暗示self对象可能发生改变。在方法中更改self对象？我们疯了吗？当然不是。该声明中最先运行的代码是[super init]，其作用是使超类完成它们自己的初始化工作。从根类NSObject继承的类调用超类的初始化方法，可以使NSObject执行所需的任何操作，以便对象能够响应消息并处理保留计数器。而从其他类继承的类调用超类的初始化方法，可以使子类有机会实现自己全新的初始化。

我们刚才已经提到，像这样的init方法可能返回完全不同的对象。请记住，实例变量所在的内存位置到隐藏的self参数之间的距离是固定的。如果从init方法返回一个新对象，则需要更新self，以便其后的任何实例变量的引用可以被映射到正确的内存位置。这也是我们需要使用self

对象不同。虽然这种类簇的讨论，你就由于`init`方法可以的对象可能更合适。已。基于这一认知。的字符串的要求)。

些方法显得有点奇

吗？当然不是。该化工作。从根类作，以便对象能以使子类有机会

实例变量所在的象，则需要更新们需要使用`self`

`= [super init]`这种形式进行赋值的原因。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`方法可能返回`nil`。例如，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用的图像对象。`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])`测试的主体代码不会执行。像这样将赋值和检查非零值合并起来是一种典型的C风格的用法，Objective-C沿袭了这一用法。

获得对象并使其运行的代码位于`if`语句的正文部分的一对大括号之间。在最初的`Car` `init`方法中，`if`语句创建了1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`方法返回已经被初始化的对象。我们已经将`[super init]`的返回值赋给`self`，这正是我们应该返回的值。

稳操胜券

有些程序员不喜欢将赋值和测试非零值合并起来，而是使用如下方式编写自己的`init`方法：

```
self = [super init];
if (self) {
    ...
}
return (self);
```

这样也很好。关键是要将返回值赋给`self`，尤其是当你正在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，将赋值和测试非零值合并起来是一种常用的技术，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，正是因为在初始化过程中超类可能改变一些对象，所以应该一直使用这种形式。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，新对象也不可能接受任何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。只不过当偶尔遇到难以理解的个别情况时，你可能需要调试。

10.2.2 初始化时做什么

`init`方法应该完成哪些工作？在该方法中，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完成任务所需要的其他对象。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`engine`对象和全部的4个`tire`对象。这种方式使`Car`类变得开箱即用：调用`alloc`和`init`，并开出汽车（`car`）试驾。下一个版本中，我们将在`init`方法中不创建任何对象，只是为`engine`对象和`tire`对象预留位置。创建`car`对象的代码将必须创建`engine`对象和`tire`对象，并使用访问方法为其赋值。

哪种方式最适合你？这取决于对灵活性与性能的权衡，这和程序设计中的许多取舍情况一样。最初的`Car` `init`方法非常方便。如果`Car`类的预期用途是创建和使用基本的`car`对象，则第一种方式更合适。

另一方面，如果`car`对象经常被自定义，具有不同种类的`tire`和`engine`（如在赛车游戏中），我们创建的这些`tire`和`engine`对象只能丢弃。多么浪费啊！创建的对象从未使用就被销毁了。

说明 即使不提供对自定义对象属性的调用，也仍然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上可能用不上的对象，则使用这种技术可以提高程序的性能。

10.3 便利初始化函数

有些对象拥有多个以`init`开头的方法。重要的是要知道，这些`init`方法实际上没什么特别的，它们都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，可以减少你自己完成这些工作的麻烦。为了让你理解我们正在讨论的内容，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，这个方法没有多大用处。但是，你可以分配和初始化一个新的`NSMutableString`类的对象并开始向该对象中保存字符。你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:`接受格式化的字符串并输出格式化的结果一样，此版本初始化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 or 624”。
- (id) initWithContentsOfFile: (NSString *) path;
initWithContentsOfFile:方法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的代码行如下：
string = [[NSString alloc]
          initWithContentsOfFile: @"/tmp/words.txt"];

```

这行代码功能非常强大。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，追加到字符串，确保尾部零字节位于适当的位置以及关闭文件）。对于Objective-C爱好者来说，这些操作简化成了一行代码。真是太好了。

10.4 更多部件改进

让我们来重温一下在第6章中出现的CarParts项目，当时我们将每一个类组织到各自的源文件中。这一次，我们将在Tire类中加入一些更高效的初始化操作，并使用这些方法执行Car类的内存管理。本章程序的项目目录是10.01 CarPartsInit，支持垃圾回收版本的项目目录是10.01 CarPartsInit-GC。

10.4.1 Tire 类的初始化

现实世界中的轮胎比我们迄今为止在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（不希望它太小）和胎面花纹深度（小于几毫米就不安全了）。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 "Tire.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，从对超类的`init`方法的调用中返回的值被赋值给`self`。然后，有用的默认值被赋给实例变量。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34 psi①，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:`生成一个包含轮胎压力和花纹深度的字符串。该方法遵守我们良好的内存管理行为规则吗？答案是肯定的。因为`desc`对象不是通过`alloc`、`copy`或`new`方法创建的，所以它的保留计数器值为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，该字符串对象将被清理。

10

10.4.2 更新`main()`函数

`main.m`文件的形式比以前更加复杂，其代码如下：

```
#import "Engine.h"
#import "Car.h"
#import "Slant6.h"
#import "AllWeatherRadial.h"

int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool;
    pool = [[NSAutoreleasePool alloc] init];
}
```

① 1 psi = 0.06895 bar。——编者注

```

Car *car = [[Car alloc] init];

int i;
for (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];
[engine release];
[car print];
[car release];

[pool release];

return (0);
}

```

下面，我们逐步剖析main()函数。首先创建一个自动释放池，为自动释放的对象在等待自动释放池被销毁时提供容身之处：

```

NSAutoreleasePool *pool;
pool = [[NSAutoreleasePool alloc] init];

```

然后使用alloc和init方法创建一个新的car对象：

```

Car *car = [[Car alloc] init];

```

之后，执行4次循环（从0到3），在该循环中创建新的tire对象：

```

int i;
for (i = 0; i < 4; i++) {

```

每执行一次循环，都创建并初始化一个新的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由Tire类的init方法设置。但是我们准备自定义这些值（只是为了好玩儿）。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：

```
[tire release];
```

这行代码假设Car类能够正确地执行内存管理，将其分配用于保留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（在那时我们还不太清楚如何做），但是我们稍后将告诉你如何解决这个问题。

与以前一样，在轮胎组装完以后，我们创建一个新的engine对象，并将该engine对象放在car中：

```
Engine *engine = [[Slant6 alloc] init];  
[car setEngine: engine];  
[engine release];
```

与tire对象一样，engine对象被释放，因为它的使命已经完成了。Car类应该确保engine对象被销毁。

最后，Car类被告知要输出自己并被释放，因为我们已经大功告成：

```
[car print];  
[car release];
```

现在，自动释放池被释放，其保留计数器值归0，该池被销毁并向池中的每一个对象发送release消息。此时，由Tire类的description方法生成的NSStrings被清理：

```
[pool release];
```

然后main()函数结束。不过，在运行该程序之前我们需要改进Car类，使其能够正确处理内存管理问题。但我们首先需要看一下在支持垃圾回收的Objective-C中，main()函数是如何实现的：

```
int main (int argc, const char * argv[]){  
    Car *car = [[Car alloc] init];  
  
    int i;  
    for (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];
```

对象在等待自动

我们准备自定义
我们打算使用访问

10

```
    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。

10.4.3 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边界检查了。为此，我们修改`Car`类的@interface部分，使用可变数组（改动的代码以粗体表示）：

```
#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];
```

```
        int i;
```

```
        for (i = 0; i < 4; i++) {
            [tires addObject: [NSNull null]];
        }
    }
```

```
    return (self);
```

```
} // init
```

你见到过的`self = [super init]`已经不计其数，对于这种形式你实际上已耳熟能详了。就

如前所知，它只是确保超类能够获得对象并使其运行。

接下来，创建`NSMutableArray`数组。有一个简便的`NSMutableArray`方法`replaceObjectAtIndex:withObject:`，该方法最适合用来实现`setTireAtIndex:`。要使用`replaceObjectAtIndex:withObject:`方法，在指定的索引位置必须存在一个能够被替代的对象。新的`NSMutableArray`数组不包含任何内容，因此需要使用一些对象作为占位符。`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（我们在第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`对象，因为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:`使用前面讲过的“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;
}
```

而`engine`访问方法只返回当前的`engine`对象：

```
- (Engine *) engine
{
    return (engine);
}
```

现在我们实现`tire`访问方法。首先是`setter`方法：

```
- (void) setTire: (Tire *) tire
    atIndex: (int) index
{
    [tires replaceObjectAtIndex: index
        withObject: tire];
}
```

`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`占位符还是此前存储的`tire`对象），所以对于`tire`对象不需要执行显式的内存管理。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:`使用由`NSArray`数组提供的`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

通过直接返回`objectAtIndex:`方法的结果值，将`tireAtIndex:`方法简化为如下所示的一行，这完全符合语法：

```
- (Tire *) tireAtIndex: (int) index
{
    return ([tires objectAtIndex: index]);
} // tireAtIndex:
```

在原来的`tireAtIndex:`方法中引入额外变量，可以使代码更易于阅读（至少对我们来说是这样的），而且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，在从调用`objectAtIndex:`方法到该方法返回`tire`对象而结束的过程中输出`NSLog()`函数的原始调试方法也更加容易。

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。`dealloc`方法正好能完成此类工作：

```
- (void) dealloc
{
    [tires release];
    [engine release];

    [super dealloc];
} // dealloc
```

这足以确保当该`car`对象被销毁时所有内存都被回收。一定要调用超类的`dealloc`方法，这一点很容易被忽视。

最后是`car`对象的`print`方法，该方法输出`tire`对象和`engine`对象：

```
- (void) print
{
    int i;
    for (i = 0; i < 4; i++) {
        NSLog(@"%@", [self tireAtIndex: i]);
    }
    NSLog(@"%@", engine);
} // print
```

`print`方法依次输出每个`tire`和`log`对象。有趣的是，该循环使用`tireAtIndex:`方法访问`NSArray`数组而不直接访问数组本身。如果希望直接访问`NSArray`数组，也没有关系。然而，如果使用访问方法（即使是在类的实现中），则必须隔离这些代码以避免其发生任何改变。例如，

如果tire对象的存储机制将来又发生了改变(例如回到了C风格的数组),则不必修改print方法。

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. VR000M!
```

化为如下所示的一

(至少对我们来说
那个对象。使用这
中输出NSLog()函
且。dealloc方法正

10.5 支持垃圾回收风格的 Car 类清理

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类是什么样子呢?setEngine方法变得更加简单。

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

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,垃圾回收器销毁原来的engine对象。另一方面,因为有一个实例变量指向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,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被调用。不过,使用finalize方法会引起一些细微的问题。但是,对于Cocoa环境下的程序设计来说,你不需要担心finalize方法。

10

构造便利初始化函数

dealloc方法,这一

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。带有新增的便利初始化函数(粗体表示)的Tire类如下:

```
#interface Tire : NSObject
{
    float pressure;
    float treadDepth;
}
- (id) initWithPressure: (float) pressure
    treadDepth: (float) treadDepth;
- (void) setPressure: (float) pressure;
```

AtIndex:方法访问
有关系。然而,若
任何改变。例如,

```
- (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执行tire对象的分配和初始化：

```
Tire *tire;
tire = [[Tire alloc]
    initWithPressure: 23 + i
    treadDepth: 33 - i];
```

10.6 指定初始化函数

遗憾的是，并不是所有的初始化方法都能够正常运行。当我们在类中增加便利初始化函数时，会出现一些细微的问题。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
```

这两个新的初始化函数`initWithPressure:`和`initWithTreadDepth:`，适用于这些人：他们知道其需要具有特定的轮胎压力或花纹深度的`tire`对象而不关心其他属性值，并且乐意接受默认值。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:`，每个初始化方法都知道默认的轮胎压力值（34）和/或花纹深度值（20）。这些方法可以工作，其代码也是正确的。

但是，开始子类化`Tire`类时，问题就来了。

10

10.6.1 子类化问题

我们已经创建了一个`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，用于表示在潮湿和有积雪的道路上如何处理轮胎。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`类的新接口如下所示：

```
@protocol AllWeatherRadial : Tire
{
    float rainHandling;
    float snowHandling;
}

- (void) setRainHandling: (float) rainHandling;
- (float) rainHandling;
```

```
- (void) setSnowHandling: (float) snowHandling;
- (float) snowHandling;
@end // AllWeatherRadial

下面是这些相当枯燥乏味的访问方法:
- (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()函数中的for循环, 该循环创建具有默认值的AllWeatherRadials类的新对象:
int i;
for (i = 0; i < 4; i++) {
    AllWeatherRadial *tire;
    tire = [[AllWeatherRadial alloc] init];
```

```
[car setTire: tire
      atIndex: i];

[tire release];
```

AllWeatherRadial类的对象的属性未被设置为合理的默认值。哪里出了问题？因为需要在init方法中设置这些值，所以我们必须重写init方法。但是Tire类还有initWithPressure:、initWithTreadDepth:和initWithPressure:treadDepth: 3个初始化方法，难道我们必须重写所有这些方法吗？而且，即使重写了这3个方法，如果Tire类中增加一个新的初始化函数，那又会怎么样呢？如果Tire类的改变会影响AllWeatherRadial类，那么这种设计是很糟糕的。

幸运的是，Cocoa的设计人员已经预料到这个问题，他们提出了指定初始化函数（designated initializer）的概念。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使用指定初始化函数执行初始化操作。子类使用其超类的指定初始化函数实现超类的初始化。通常，接受参数最多的初始化方法最终成为指定初始化函数。如果你使用别人的代码，则一定要检查文档，弄清哪个方法是指定初始化函数。

10.6.2 改进 Tire 类的初始化函数

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最灵活。

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，其可能的实现方式如下：

```
- (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);
}
```

10

```

    return (self);
} // initWithPressure

- (id) initWithTreadDepth: (float) td
{
    if (self = [self initWithPressure: 34.0
                                treadDepth: td]) {
    }
    return (self);
} // initWithTreadDepth

```

说明 你并不是真正需要像在initWithPressure:treadDepth:方法中一样使用if声明的空的主体部分，我们这样做只是为了保持所有的初始化方法具有一致的形式。

10.6.3 添加 AllWeatherRadial 类的初始化函数

现在该向AllWeatherRadial类中添加初始化函数了，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

```

此刻，当我们运行该程序时，AllWeatherRadial类的对象设置了恰当的默认值：

```

AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
I am a slant-6. VR000M!

```

哇，汽车真的发动了！

10.7 初始化函数规则

不需要为自己的类创建初始化函数方法。如果不需要设置任何状态，或者只需要alloc方法将内存清零的默认行为，则不需要担心init。

如果构造了一个初始化函数，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，就像我们在前面的实现一样。

10.8 小结

本章主要讲述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方法，可以更容易地实现你希望的对象配置方式。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值赋给`self`对象，并返回你自己的初始化方法的值。超类可能决定返回一个完全不同的对象。

下一章将讲述属性，这是创建访问方法的一种快速、简便的技术。

第 11 章

特性

11

还记得我们在为实例变量编写访问器方法时曾被搞得一头雾水吗？我们编写了大量的样板代码，不但要（显式地）编写`-setBlah`方法来设置对象的`blah`属性（attribute），还要编写`-blah`方法取回对象的`blah`属性。如果属性是对象，则需要保留新对象，并释放旧对象。虽然有些实用工具能够将类定义转换为可以直接在文件中使用的方法声明和定义，但是编写访问器方法仍然是一项索然无味的工作，而我们完全可以更好地利用这些时间来为自己的程序实现更棒的功能。

苹果公司在Objective-C 2.0中引入了特性（property），它组合了新的预编译指令和新的属性访问器语法。新的特性功能显著减少了必须编写的冗长代码的数量。我们将在整章中修改10.01 CarParts-Init项目以使用特性，本章的最终代码位于11.01 CarProperties项目文件夹中。

请记住，Objective-C 2.0的特性只适用于Mac OS X 10.5（Leopard）或更高版本。特性主要应用于Cocoa的新组件（尤其是华丽夺目的Core Animation效果），在iPhone开发中也有大量应用，因此值得我们学习。

11.1 修改特性值

首先，我们转换一个比较简单的类（AllWeatherRadial）以使用特性。为了增强讨论的趣味性，我们在`main()`函数中增加了几次调用，以修改我们创建的`AllWeatherRadials`类的一些值。我们模拟有人从不同的商店购买4个降价销售的轮胎的情景，因此这4个轮胎具有不同的调节特性。

修改后的`main()`函数如下（新增的代码以粗体表示）：

```
int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool;
    pool = [[NSAutoreleasePool alloc] init];

    Car *car = [[Car alloc] init];

    int i;
    for (i = 0; i < 4; i++) {
        AllWeatherRadial *tire;
        tire = [[AllWeatherRadial alloc] init];
        tire.rideHeight = i;
        tire.tireWidth = i;
        tire.tireType = i;
        tire.wearLevel = i;
        car.tires = tire;
    }
}
```

```

[tire setRainHandling: 20 + i];
[tire setSnowHandling: 28 + i];
NSLog(@"the tire's handling is %.f %.f",
      [tire rainHandling],
      [tire snowHandling]);

[car setTire: tire
atIndex: i];

[tire release];
}

Engine *engine = [[Slant6 alloc] init];
[car setEngine: engine];

[car print];
[car release];

[pool release];

return (0);
} // main

```

如果现在运行该程序，你会得到如下的输出结果，该结果显示了我们最新修改过的轮胎的调节值：

```

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. VR000M!

```

11

11.1.1 简化接口

现在来研究一下AllWeatherRadial类的接口：

```

#import <Foundation/Foundation.h>
#import "Tire.h"

@interface AllWeatherRadial : Tire {
    float rainHandling;
    float snowHandling;
}

- (void) setRainHandling: (float) rainHandling;
- (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特性风格的新形式：

```
#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个方法定义。注意，我们引入了两个以@为前缀的关键字。还记得@符号标志着“你将使用Objective-C的特殊用法”吗？@property是一种新的编译器功能，表示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。现在你可以运行该程序，将得到和以前一样的结果。@property预编译指令的作用是自动声明属性的setter和getter方法。实际上，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。我们稍后将讨论这个话题。特性还有一些其他的用法，我们以后再讨论这些内容，请耐心等待。

11.1.2 简化实现

下面再来看一下AllWeatherRadial类的实现：

```
#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setter和getter方法以及description方法。现在，我们打算彻底消除全部setter和getter方法，而代之以两行简单的代码：

```
#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

```

@synthesize也是一种新的编译器功能，表示“创建该属性的访问器”。当遇到代码**@synthesize rainHandling;**时，编译器将输出**-setRainHandling:**和**-rainHandling**方法的已编译的代码。

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中生成访问器的方式，并可能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修改**AllWeatherRadial**类的实现之前一样。

11.1.3 点表达式的妙用

Objective-C 2.0的特性引入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。这些新特性也使那些习惯了C++和Java等语言的编程人员觉得Objective-C比较容易上手。

回忆一下我们在**main()**函数中增加的、用于修改轮胎调节值的两行新代码：

```

[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，你将会得到同样的结果。我们使用`NSLog()`函数来输出轮胎的调节值：

```
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点类似，其实这是Objective-C的设计人员有意为之。如果点表达式出现在等号`(=)`左边，该属性名称的`setter`方法`(-setRainHandling:和-setSnowHandling:)`将被调用。如果点表达式出现在对象变量右边，则该属性名称的`getter`方法`(-rainHandling和-snowHandling)`将被调用。

说明 点表达式只是调用访问器方法的一种快捷方式，这种用法并没有其他神秘之处。我们将在第16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。特性的点表达式和流行的键/值编码的后台工作之间没有联系。

如果你在使用特性时遇到奇怪的错误消息，提示访问的对象不是`struct`类型，请检查你是否已经包含了使用的类所需的所有必要的头文件。

在引入了特性以后，这种错误是初学编程的人最容易犯的。当然，对于正确地处理对象的属性以及避免`setter`和访问器的公开，还有一些其他情况需要讨论，我们将在下一章介绍这些内容。

11

11.2 特性扩展

迄今为止，我们已经研究了标量类型，尤其是`float`类型的特性，但是，该技术同样适用于`int`、`char`、`BOOL`和`struct`类型。例如，只要你愿意，你甚至可以定义一个`NSRect`类的对象的特性。

不过，使用对象的特性也会带来一些麻烦。回想一下我们在使用访问器访问对象时需要保留和释放对象的情景。对于某些对象的值，尤其是字符串的值，你总是希望复制它们。而对于其他对象的值，如委托（我们将在下一章讨论），你根本不会希望保留它们。

说明 稍等一下！复制对象和保留对象有什么用？

你希望能够复制字符串参数。一种常见的错误就是从用户界面（如一个文本域）中获得一个字符串，并将其作为字符串的名称使用。从文本域中获得的字符串通常是可变的字符串，当用户输入新的字符串时其内容将发生改变。复制该字符串可以防止其值意外变化。

那么，不留对象会出现什么情况呢？有一种特殊的情况，叫保留周期（retain cycle），引用计数器在该周期中归零。如果两个实体之间具有拥有和被拥有关系，例如Car类和Engine类，则我们希望car对象保留（拥有）engine对象（而不是相反的关系）。engine对象不应该保留已经安装了引擎的car对象。如果car对象保留了engine对象，而engine对象也保留了car对象，则这两个对象的引用计数的值都不会归零，这两个对象也永远不会被清理。直到engine对象在自己的dealloc方法中释放了car对象以后，Car类的dealloc方法才能被调用。同样地，直到car对象在自己的dealloc方法中释放了Engine类以后，engine对象的dealloc方法才能被调用。它们都一直等待着对方先释放自己。一般原则是所有者对象保留被拥有的对象，而不是被拥有的对象保留所有者对象。好在使用垃圾回收的用户不需要担心这种情况。

下面给Car类添加一种新功能，以便我们能够使用某些新的特性语法。这真是一种令人激动的功能。我们在car对象中添加一个新的实例变量name。让我们再次回到从前，使用传统的访问器方法。首先来看一下Car.h文件的内容（新增的代码以粗体表示）：

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

@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，我们增加访问器方法的实现（注意，我们在复制name对象），同时为car对象选择默认的名称并在其description方法中输出：

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

```
- (id) init
{
    if (self = [super init]) {
        name = @"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];
}

} // setTireAtIndex:

- (Tire *) tireAtIndex: (int) index
{
    Tire *tire;
    tire = [tires objectAtIndex: index];

    return (tire);
} // tireAtIndex:

- (void) print
{
    NSLog (@"%@ has:", name);
    int i;
    for (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"];
现在运行该程序，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类中添加特性。功能空前强大的Car.h文件如下：
#import <Cocoa/Cocoa.h>

@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

```

你将会注意到，简单的访问器方法的声明消失了，已经被`@property`声明取代。你还可以声明具有其他属性的`@property`，以表达你希望特性具有何种行为的确切意愿。通过向`name`对象添加`copy`方法，编译器和该类的使用者知道`name`对象将被复制。因为编程人员知道他们不需要复制从文本域获得的字符串，这样可以简化编程人员使用该类的工作。另一方面，对`engine`对象的管理只有保留和释放操作。如果你不曾指定这两种操作中的任一种，编译器将默认执行赋值操作，而这通常不是你希望对对象执行的操作。

说明 你还可以使用一些其他的声明（如`nonatomic`），如果不在多线程环境中使用，这些声明可以提高访问器方法的调用速度。台式机的速度已经够快了，因此通过创建特性的`nonatomic`声明对于性能的提升实际上不起什么作用，但是iPhone开发人员经常使用这种技术在资源有限的设备上获得更好的性能。如果不希望保留属性的对象，你可以使用`assign`方法以避免保留周期问题。

Car.m有两个重大变化，删除了`name`和`engine`对象的访问器方法，增加了两个`@synthesize`预编译指令：

```

@implementation Car
@synthesize name;
@synthesize engine;
最后，main()函数使用点表示法给对象赋值：
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，特性的名称始终与支持特性的实例变量的名称相同。这种方式非常普遍，而且可能将是您使用最多的方式。不过，有时您可能希望实例变量有一个名称，而公开的属性`name`具有另一个名称。

例如，如果希望使用其他名称（如`appellation`）调用`Car`类中的实例变量`name`，则只需要在`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，修改`@synthesize`指令：

```
@synthesize name = appellation;
```

编译器仍将创建`-setName:`和`-name`方法，但在其实现中却使用实例变量`appellation`。

但这样做，在编译时将会遇到一些错误。你可能还记得，我们直接访问的实例变量`name`已经被修改了。我们既可以选择搜索并替换`name`，也可以将直接的实例变量访问修改为使用访问器访问。在`init`方法中，将

```
name = @"Car";
```

修改为：

```
self.name = @"Car";
```

`self.name`是什么意思呢？该语句的作用是消除歧义，使编译器知道我们期望使用访问器访问`name`。如果只使用裸名`name`，编译器将假设我们直接修改了实例变量。要使用访问器方法，我们可以写`[self setName:@"Car"]`。请记住，点表示法只是准确调用`name`的一种快捷方式，因此`self.name = @"Car"`只不过是准确调用`name`的另一种写法。

在`dealloc`方法中，我们采用一种高明的技巧：

```
self.name = nil;
```

这行代码表示使用`nil`参数调用`setName:`方法。生成的访问器方法将自动释放以前的`name`对象，并使用`nil`替代`name`。该方法完成了释放`name`对象所占用内存的操作。当然，也可以只释放`name`对象以清理其占用的内存。如果你在`dealloc`方法以外的地方清除特性，那么使用“将`nil`赋值给对象”的方法可以将特性设置为`nil`，同时可以使我们避免对已释放内存的悬空引用问题。

最后，`-description`方法需要使用第一次被修改的 `NSLog()`函数：

```
NSLog(@"%@", has:, self.name);
```

现在，可以将`appellation`重命名为其他一些名字，如`nickname`或`moniker`。只需修改实例变量的名称以及在`@synthesize`中使用的名称。

11.2.2 只读特性

可以使某个对象具有只读属性。该属性可能是一个即时计算的值（如香蕉的表面积），也可

能是一个其他对象只能读却不能修改的值，如你的驾驶证号码。你可以使用@property预编译指令的更多属性编写处理这些情况的代码。

默认情况下，特性是可修改的：既可读取又可写入。你可以使用特性的**readwrite**属性。由于特性的默认属性是可读写，因此你通常不会使用这一属性。但是，当希望明确表示自己的意图时，你可能需要使用它。我们本来也可以在Car.h文件中使用读写属性：

```
@property (readwrite, copy) NSString *name;
@property (readwrite, retain) Engine *engine;
```

但是我们没有这样做，因为我们通常希望能够杜绝和消除冗余和重复。

我们接着讨论只读特性。比如驾驶证号码或鞋码这类特性，我们当然不希望任何人更改它。

我们可以使用@property的只读属性。使用只读属性的一个示例类如下：

```
@interface Me : NSObject {
    float shoeSize;
    NSString *licenseNumber;
}
@property (readonly) float shoeSize;
@property (readonly) NSString *licenseNumber;
@end
```

当知道@property是只读特性时，编译器将只为该属性生成一个getter方法而不生成setter方法。Me类的使用者可以调用-*shoeSize*和-*licenseNumber*，但是不能调用-*setShoeSize*，否则编译器将会报错。使用点表示法时的效果也是一样的。

11.2.3 特性不是万能的

你可能已经注意到，我们并没有转换Car类的tire方法以使用特性：

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

这是因为这些方法并不适合特性所能涵盖的方法的相当狭小的范围。特性只支持替代-*setBlah*和-*blah*方法，但是不支持那些需要接受额外参数的方法，例如car对象中轮胎的状况。

11.3 小结

本章主要讲述了特性，在对对象的属性执行常见的操作时，利用特性可以减少需要编写（以及随后需要读）的代码数量。使用@property预编译指令可以通知编译器：“嘿，这个对象具有这种类型的这个名称的这种属性。”使用这一指令还可以传递一些关于特性（例如它的可变性：只读或读写）和对象的内存管理（retain、assign或copy）的信息。此外，编译器还可以为对象的属性的setter和getter自动生成方法声明。

使用@synthesize预编译指令可以通知编译器生成访问器方法。你还可以控制由编译器生成的访问器方法对哪些实例变量起作用。如果不使用苹果公司提供的默认行为，你也可以自由地编写自己的访问器代码。

尽管点表示法通常出现在特性的上下文中，但是它只是调用对象的setter和getter方法的一种快捷方式。例如，`dealie.blah = greeble`完全等价于`[dealie setBlah: greeble]`，而`shronk = dealie.greeble`完全等价于`shronk = [dealie greeble]`。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允许你扩展现有类（即使你没有这些类的源代码）的一种技术。千万不要错过了。

在 编写面向对象的程序时，你经常希望向现有的类添加一些新的行为：你总是能够为对象提供使用这些新方法的新手段。例如，你设计了一种新轮胎，因此你应该创建Tire类的子类并添加新的成员变量。当希望为现有的类增加新行为时，我们通常会创建子类。

但是有时子类并不方便。例如，你可能会希望为NSString类增加一些新行为，但是你知道NSString实际上只是一个类簇的前台表示，因而无法为这样的类创建子类。在其他情况下，你也许可以创建子类，但你使用的却是工具包或类库，因而又无法处理新类的对象。例如，当使用类方法stringWithFormatFor生成一个新的字符串时，你新建的NSString类的子类无法返回。

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！这些新方法的Objective-C术语称为“类别”（category）。

12.1 创建类别

12

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，包括那些你没有源代码的类。

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，确定每个字符串的长度并将这些字符串的长度存入NSArray数组或NSDictionary字典中。你需要先将每个长度包装在一个NSNumber对象中，然后才能将其存入NSArray数组或NSDictionary字典中。

你可以像下面这样编写代码：

```
NSNumber *number;
number = [NSNumber numberWithUnsignedInt: [string length]];
// ... do something with number
```

但是，这样做会让你很快就感到厌烦。相反，你可以为NSString类添加一个类别，该类别可以替你完成这项工作。下面我们就来完成这样的工作。位于12.01 LengthAsNSNumber项目目录中的LengthAsNSNumber项目，包含了向NSString类中添加这样一个类别的代码。

12.1.1 声明类别

类别的声明格式与类的声明格式相似：

```
@interface NSString (NumberConvenience)
```

```
- (NSNumber *) lengthAsNumber;
```

```
@end // NumberConvenience
```

你应该可以看出，该声明具有两个特点。首先，现有的类位于@interface关键字之后，其后是位于圆括号中的一个新名称。该声明表示，类别的名称是NumberConvenience，而且该类别将向NSString类中添加方法。换句话说：“我们正在向NSString类添加一个名称为NumberConvenience的类别。”只要保证类别名称的唯一性，你可以向一个类中添加任意多的类别。

其次，你可以指定希望向其添加类别的类（在本例中是NSString）以及类别的名称（在本例中是NumberConvenience），而且你还可以列出添加的方法，最后以@end结束。由于不能添加新实现变量，因此与类的声明不同的是，类别的声明中没有实例变量部分。

12.1.2 实现类别

与@interface部分对应的还有一个@implementation部分，这一点没什么奇怪的。你可以在@implementation部分实现自己的方法：

```
@implementation NSString (NumberConvenience)
```

```
- (NSNumber *) lengthAsNumber
```

```
{
```

```
    unsigned int 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正文部分。

lengthAsNumber方法通过调用[self length]来获得字符串的长度。你可以向该字符串发送lengthAsNumber消息。然后，Objective-C创建一个具有指定长度的NSNumber类的新对象。

我们暂停一会儿，讨论一下我们最近爱谈论的话题之一：内存管理。这样的代码正确吗？答案是肯定的。numberWithUnsignedInt不同于alloc、copy和new方法。由于numberWithUnsignedInt不属于这三个方法，因此它能够返回这样一个对象：我们可以假设该对象的保留计数器的值为1且已经被设置为自动释放。在当前的活动自动释放池被销毁时，该NSNumber类的对象将被清理。

下面是我们实现的新的类别。main()函数创建一个新的NSMutableDictionary类的对象，添加3个字符串及其长度，分别作为键和值：

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

```
{
```

```
    NSAutoreleasePool *pool;
```

```
    pool = [[NSAutoreleasePool alloc] init];
```

```

NSMutableArray *dict;
dict = [NSMutableArray 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);

[pool release];

return (0);
} // main

```

下面，我们像以前一样，一点一点地剖析main()函数。首先，创建一个自动释放池（这话可能已经让你耳朵生茧了）：

```

NSAutoreleasePool *pool;
pool = [[NSAutoreleasePool alloc] init];

```

提醒一下，该自动释放池为所有的自动释放对象提供了容身之处。特别地，与我们使用类别创建的所有NSNumber类的对象将在自动释放池中被销毁一样，可变字典也将在这里被销毁。

在创建自动释放池以后，再创建一个新的可变字典。还记得吗，这种便利的Cocoa类支持我们成对地存储键和对象。

```

NSMutableArray *dict;
dict = [NSMutableArray dictionary];

```

由于不能将像int这样的基本类型存入字典，因此我们必须使用像NSNumber这样的包装类。不过，有了引人注目的新类别，我们可以轻松地将字符串的长度嵌入到一个NSNumber类的对象中。使用键@"hello"将整数值5添加到字典中的代码如下：

```

[dict setObject: @"hello" lengthAsNumber]
    forKey: @"hello"];

```

虽然这行代码看起来有点奇怪，但是它的确能够完成这一任务。要知道，@"字符串"这种形式实际上是地道的NSString类的对象。它们对消息的响应正如任何其他NSString类的对象对消息的响应一样。因为我们已经创建了NSString类的这种类别，所以任何字符串，即使是文本字符串，都将响应lengthAsNumber消息。

要反复强调的是，任何NSString类的对象都将响应lengthAsNumber消息，这些对象包括文本字符串、来自description方法的字符串、可变字符串、来自工具包的其他部件的字符串、从文件加载的字符串、从整个因特网的海量内容中提取的字符串等。正是这种兼容性使类别成为一种非常伟大的概念。不需要创建NSString类的子类来获得这种行为，类别就可以完成同样的工作。

如果运行该程序，你将会得到如下所示的结果：

```
{  
    "Once upon a time" = 16;  
    hello = 5;  
    ILikeFish = 9;  
}
```

12.1.3 类别的局限性

现在，你可能已经完全陶醉于类别的强大功能了，那么让我给你泼点冷水。类别有两方面的局限性。第一，无法向类中添加新的实例变量。类别没有位置容纳实例变量。

第二，名称冲突，即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，从而无法再使用初始方法。有些编程人员在自己的类别方法名中增加一个前缀，以确保不发生名称冲突。

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，可以使用全局字典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。

12.1.4 类别的作用

Cocoa中的类别主要用于3个目的：将类的实现分散到多个不同文件或多个不同框架中，创建对私有方法的前向引用，以及向对象添加非正式协议。如果你还不理解“非正式协议”（informal protocol）的含义，请不要担心，我们稍后将简单地讨论这一概念。

12.2 利用类别分散实现

正如在第6章中所见到的，你可以将类的接口放入头文件中，而将类的实现放入.m文件中。但是你不能将@implementation分散到多个不同的.m文件中。如果希望将大型的单个类分散到多个不同的.m文件中，你可以使用类别来完成这一工作。

以AppKit提供的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NSWindow类的文档在打印时将超过60页。

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而难以驾驭，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（位于/System/Library/Frameworks/AppKit.framework/Headers/NSWindow.h）并查找“@interface”，你将会看到官方的类接口：

```
@interface NSWindow : NSResponder  
然后是一大堆的类别，其中包括下面这些：  
@interface NSWindow(NSKeyboardUI)  
@interface NSWindow(NSToolbarSupport)
```

```
@interface NSWindow(NSDrag)
@interface NSWindow(NSCarbonExtensions)
@interface NSObject(NSWindowDelegate)
```

利用类别，所有的键盘用户界面代码都位于一个源文件中，工具栏代码位于一个源文件中，拖放功能位于另一个源文件中，等等。类别还可以将方法组织到逻辑分组中，使编程人员可以更加容易地阅读头文件。这正是我们准备小规模地实现的内容。

在项目中使用类别

位于12.02 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 {
    int thing1;
    int thing2;
    int thing3;
}

@end // CategoryThing

类声明之后是3个类别，每个类别具有一个实例变量的访问器方法。我们将这些实现分散到不同的文件中。
```

```
@interface CategoryThing (Thing1)
- (void) setThing1: (int) thing1;
- (int) thing1;

@end // CategoryThing (Thing1)

@interface CategoryThing (Thing2)
- (void) setThing2: (int) thing2;
- (int) thing2;

@end // CategoryThing (Thing2)

@interface CategoryThing (Thing3)
- (void) setThing3: (int) thing3;
- (int) thing3;

@end // CategoryThing (Thing3)
```

以上就是CategoryThing.h的代码。

CategoryThing.m文件相当简单，包含一个description方法，我们可以按照 NSLog() 函数中的%@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
```

我们暂停一会儿来检查一下内存管理问题。description方法能够完成内存管理的任务吗？答案是肯定的。因为stringWithFormat不同于alloc、copy和new，它能够返回这样一个对象：我们可以假设该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: (int) t1  
{  
    thing1 = t1;  
} // setThing1  
- (int) thing1  
{  
    return (thing1);  
} // thing1  
@end // CategoryThing
```

特别值得指出的是，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非常相似：

```
#import "CategoryThing.h"  
@implementation CategoryThing (Thing2)
```

```
- (void) setThing2: (int) t2
{
    thing2 = t2;
} // setThing2

- (int) thing2
{
    return (thing2);
} // thing2

@end // CategoryThing
```

读到这里，你可能已经知道Thing3.m文件的内容该怎么写了（提示：使用剪切、粘贴、查找和替换）。

main.m文件包含main()函数，实际上使用我们创建的类别的正是该函数。首先是#import声明：

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

我们需要先导入CategoryThing的头文件，以使编译器能够找到类定义和类别。然后是main()函数：

```
int main (int argc, const char *argv[])
{
    NSAutoreleasePool * pool;
    pool = [[NSAutoreleasePool alloc] init];

    CategoryThing *thing;
    thing = [[CategoryThing alloc] init];

    [thing setThing1: 5];
    [thing setThing2: 23];
    [thing setThing3: 42];

    NSLog (@"Things are %@", thing);

    [thing release];

    [pool release];

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

main()函数的前面两行是已经令你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() 函数使用的自动释放的description方法的字符串。

接下来，一个CategoryThing类的对象被分配和初始化：

```
CategoryThing *thing;
thing = [[CategoryThing alloc] init];
```

内存管理策略的强制性报告：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，所以我们必须在使用完该对象后将其释放。

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并不重要。

在`thing`对象被赋值以后，`NSLog()`函数输出该对象的值。正如`CategoryThing`类的`description`方法一样，`NSLog()`函数输出了`thing`对象的3个实例变量的值：

```
NSLog(@"%@", thing);
```

因为是使用`alloc`方法创建`thing`对象的，所以我们在使用完该对象以后要负责将其释放。这种释放是立即进行的：

```
[thing release];
最后, 自动释放池被释放, main()函数返回0:
[pool release];
return (0);
```

以上就是我们这个小小的程序。运行该程序将得到下面的结果：

```
Things are 5 23 42
```

不但可以将一个类的实现分散到多个不同的源文件中，还可以将其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、绘图等）都包含在AppKit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对象发送绘图消息。当输出一个字符串时，该方法将字符串的文本绘制在屏幕上。由于这是一种有用的图形功能，所以它在AppKit中实现。但`NSString`又是Foundation类的对象。Cocoa的设计人员使用类别将数据功能放在Foundation中实现，而将绘图功能放在AppKit中实现。作为编程人员，我们只需处理`NSString`即可，通常不用关心特定的方法来自何处。

12.3 使用类别创建前向引用

前面提到过，Cocoa没有任何真正的私有方法。只要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该方法的声明，你也可以调用该方法。

不过，编译器会尽力提供帮助。如果编译器遇到你调用对象的某个方法，但是没有找到该方法的声明或定义，则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获许多打字或排印

错误。

但是,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将会带来一些问题。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,编译器将会提出警告。修复所有的编译器警告是一种良好的习惯,那么能做些什么呢?

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,因而不产生警告。但是,如果不方便这样做,或者你使用了另一个类尚未发布的方法,那么就需要采取其他措施。

用于急救的类别

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,你不必实现那些你不想实现的方法。

我们经常采用的技术是将类别置于实现文件的最前面。比如,Car类有一个名为rotateTires的方法。我们可以按照另一个名为moveTireFromPosition:toPosition:的方法的形式来实现rotateTires方法,以交换两个位置的轮胎。第二个方法是实现细节,而且我们不想将其放在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,同时不会引起编译器产生警告。该类别如下所示:

```
@interface Car (PrivateMethods)
- (void) moveTireFromPosition: (int) pos1
    toPosition: (int) pos2;
@end // Private Methods
```

当你实现moveTireFromPosition:toPosition:方法时,它不必出现在@interface Car (PrivateMethods)代码块中。你可以将其留在@implementation Car部分实现。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私有方法,但有时你必须解决Cocoa或其他人的代码中的bug,或者编写测试代码)。

12

12.4 非正式协议和委托类别

接下来讨论更多的重大词语和重要意图了。你知道,它们经常出现在面向对象的编程中,其字面意思比它们的实际含义更加复杂。

Cocoa中的类经常使用一种名为委托(delegate)的技术,委托是一种对象,另一个类的对象会要求委托对象执行它的某些操作。例如,当AppKit类的NSApplication启动时,它会询问其委托对象是否应该打开一个无标题窗口。NSWindow类的对象询问它们自己的委托对象是否应该允许关闭某个窗口。

更常用的是，编写委托对象并将其提供给其他一些对象，通常是提供给Cocoa生成的对象。通过实现特定的方法，你可以控制Cocoa中的对象的行为。

Cocoa中的滚动列表是由AppKit类的NSTableView处理的。当tableView对象准备好执行某些操作（例如选择用户刚刚点击的行）时，它询问其委托对象是否选择此行。tableView对象给其委托对象发送一条消息：

```
- (BOOL) tableView: (NSTableView *) tableView  
shouldSelectRow: (int) row;
```

委托方法可以查看tableView对象和行并确定是否应该选择该行。如果表中包含了不该选择的行，则委托对象可能禁用那些不被选择的行。

12.4.1 iTunesFinder 项目

支持你查找由Bonjour（该技术以前称为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期待的服务并为其提供一个委托对象。然后，该浏览器对象将向委托对象发送消息，告之其发现新服务的时间。

位于12.03 iTunesFinder项目文件夹中的iTunesFinder项目，使用NSNetServiceBrowser来列出其能找到的所有共享的iTunes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。我们首先创建自动释放池：

```
int main (int argc, const char *argv[])  
{  
    NSAutoreleasePool *pool;  
    pool = [[NSAutoreleasePool alloc] init];
```

接下来，我们创建NSNetServiceBrowser类的一个新对象：

```
NSNetServiceBrowser *browser;  
browser = [[NSNetServiceBrowser alloc] init];
```

再然后，创建iTunesFinder类的一个新对象：

```
iTunesFinder *finder;  
finder = [[iTunesFinder alloc] init];
```

因为这些对象是使用alloc方法创建的，所以我们必须负责确保当不再使用这些对象时将其释放。

下面，我们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，我们告诉浏览器去搜索iTunes共享：

```
[browser searchForServicesOfType: @"_daap._tcp"  
    inDomain: @"local.@"];
```

字符串“_daap._tcp”告诉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, 该列表通常被映射到Bonjour服务名称。

接下来, main() 函数记录了其已经开始浏览并启动一个run循环的事实:

```
 NSLog (@"begin browsing");
```

```
 [[NSRunLoop currentRunLoop] run];
```

run循环是一种Cocoa构造, 它一直处于阻塞状态 (即不执行任何处理), 直到某些有趣的事情发生为止。在本例中, “有趣的事情”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。实际上, 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, 我们没有改正这个问题, 反正该代码的读者知道我们意识到了正确的内存管理 (我们本来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什么实际的帮助)。因此, 下面的清理代码实际上将不会执行:

```
[browser release];  
[finder release];
```

```
[pool release];
```

```
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  
@end // iTunesFinder
```

请记住, 我们并不需要在@interface中声明方法。要成为一个委托对象, 我们只需实现已经打算调用的方法。

该实现包含两个方法。首先是一些基本的声明:

```
#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发现新服务时，它给委托对象发送netServiceBrowser: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，你可以利用参数检查计算出哪个浏览器发现了新服务。

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，用于指示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moreComing参数的值为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。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系统取得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。Mark的网络中有一台陈旧的、名为iLamp的G4 iMac计算机，该计算机共享了iTunes音乐“Around the House”。该程序的输出结果如下：

```

begin browsing
found one! Name is iLamp

```

我们在一台笔记本电脑上启动iTunes，并以markd's music的名义共享音乐：

```
found one! Name is markd's music
在关闭笔记本电脑上的iTunes以后, ITunesFinder会告诉我们:
lost one! Name is markd's music
```

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 通过像这样创建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)。

12

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,因此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类中没有实现这些方法。没关系。使用非正式协议,你可以只实现你想要的方法。

你可能已经想到,应该还有一个正式协议的概念。我们将在下一章讨论正式协议。

12.4.3 响应选择器

你可能会问自己:“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？其实，它并没有逃避这一问题。`NSNetServiceBrowser`首先检查对象，询问其能否响应应该选择器。如果该对象能够响应应该选择器，`NSNetServiceBrowser`则给它发送消息。

什么是选择器呢？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()`预编译指令指定选择器，其中方法名位于圆括号中。因此，`Car`类的`setEngine:`方法的选择器将是：

```
@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`respondsToSelector:`方法：

```
Car *car = [[Car alloc] init];
if ([car respondsToSelector:@selector(setEngine:)]) {
    NSLog(@"yowza!");
}
```

这段代码将输出“yowza！”，因为一个`Car`类的对象确实能够响应`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存储。这样可以生成一些非常强大和灵活的构造。

AppKit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调用你的游戏引擎中的`moveMonsterTowardPlayer:`方法。或者，你再创建另外一个定时器调用`animateOneFrame:`方法。

12.5 小结

本章介绍了类别。类别提供了向现有的类添加新方法的手段，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，类别还提供了将对象的实现分散到多个不同的源文件、甚至多个不同的框架中的方法。例如，`NSString`类的数据处理方法在Foundation框架中实现，独立于其在AppKit中实现的绘图方法。

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，它可以列出对象能够响应的方法。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，选择器是一种在代码中指定特定的Objective-C消息的方法。

下一章将讲述Objective-C的协议，与非正式协议类似但功能更强的正式协议。

在上一章中，我们讨论了类别和非正式协议的魅力。正如第12章所述，在使用非正式协议时，可以只实现你期望响应的方法。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。

你可能已经想到，Objective-C和Cocoa还有一个正式协议（formal protocol）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，正式协议是一个命名的方法列表。但与非正式协议不同的是，正式协议要求显式地采用协议。采用协议的办法是在类的`@interface`声明中列出协议的名称。此时，你的类遵守该协议（你可以认为自己是一个不妥协主义者）。采用协议意味着你承诺实现该协议的所有方法。否则，编译器会通过生成警告来提醒你。

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，有时我们还可能会瞎忙。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的有用功能。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

下面我们来看看由Cocoa声明的一个协议——`NSCopying`。如果你采用了`NSCopying`协议，你的对象将知道如何复制自己：

```
@protocol NSCopying
- (id) copyWithZone: (NSZone *) zone;
@end
```

声明协议的语法看起来与声明类或类别的语法有点像。不过这里不是使用@interface，而是使用@protocol告诉编译器：“下面将是一个新的正式协议。”@protocol之后是协议名称，协议名称必须唯一。

然后是一个方法声明列表，协议的每个采用者都必须实现这些方法。协议声明以@end结束。使用协议不引入新的实例变量。

我们来看另一个例子。下面是Cocoa的NSCoding协议：

```
@protocol NSCoding
- (void) encodeWithCoder: (NSCoder *) aCoder;
- (id) initWithCoder: (NSCoder *) aDecoder;
@end
```

当某个类采用NSCoding协议时，意味着该类承诺实现这两个方法。`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的冻结的(freeze-dried)实例变量并使用它们初始化一个新对象。这两个方法总是成对实现的。如果你从来不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。如果你从不对对象进行编码，那么你也无法创建一个新对象。

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，并用尖括号将协议名称括起来。例如，如果Car类要采用NSCopying协议，则其类声明像下面这样：

```
@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，则其类声明如下：

```
@interface Car : NSObject <NSCopying, NSCoding>
{
    // instance variables
}

// methods

@end // Car
```

你可以按任意顺序列出这些协议，没有什么影响。

采用某个协议，相当于给阅读类声明的编程人员发送一条消息，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，二是能够复制自身。

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，我们将在晚些时候讨论这些内容）。我们准备花费本章的大部分篇幅来完成采用CarParts项目的NSCopying协议的练习。

13.2 复制

让我们一起齐声朗诵内存管理规则：“如果你使用alloc、copy或new方法获得一个对象，则该对象的保留计数器的值为1，而且你要负责释放它。”我们已经学习了alloc和new方法，但还没有讨论过copy方法。当然，copy方法可以复制对象。copy消息通知对象创建一个全新的对象，并使新对象与接收copy消息的原对象一样。

现在，我们将扩展CarParts项目，以便复制car对象（我们一直在等待这个激动人心的时刻）。完成这一任务的代码位于13.01 - CarParts-Copy项目文件夹中。另外，我们还将讨论在实现复制代码时涉及的一些有趣的细节问题。

复制的种类

实际上，你可以使用不同的方法复制对象。大多数对象都引用（即指向）其他对象。浅层复制（Shallow Copy）不复制引用对象，新复制的对象只指向现有的引用对象。NSArray类的copy方法是浅层复制。当复制一个NSArray类的对象时，你复制的对象只复制指向引用对象的指针，而不复制引用对象本身。如果复制一个NSArray类的对象，该对象包含5个NSString类的对象，则你最终得到的是5个可供程序使用的字符串对象，而不是10个字符串对象。如果那样的话，每个新对象最终获得一个指向一个字符串对象的指针。

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你将得到10个可用的字符串对象。对于CarParts项目，我们准备使用深层复制。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），而不需要更改每个car对象的轮胎压力。

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你的组合对象。

要复制一个car对象，我们需要能够复制engine和tire对象。编程人员们，就让我们从复制engine对象开始吧！

13.2.1 复制 Engine

我们处理的第一个类是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被转换为copyWithZone:方法。虽然以前的NSZone类的功能比现在更强大，但我们仍然只使用了它的一小部分功能。

Engine类的copyWithZone:方法的实现如下：

```
- (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。但是，这事说起来容易做起来难。看看engineCopy对象右边的声明多么复杂。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对象所属的类。然后，copyWithZone:方法向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init消息以使其初始化。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]这种用法。

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。那么我们将其发送给哪个类呢？我们的第一直觉是将allocWithZone:消息发送给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，所以这行代码最终在Engine类的copyWithZone:方法中结束。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，而不是Slant6类的对象。如果Slant6类增加一些实例变量，情况会变得更加扑朔迷离。如果那样，Engine类的对象将无法容纳额外的变量，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类的对象，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engine对象（如MatterAntiMatterReactor），则这些新的engine对象也会被正确地复制。

`allocWithZone`:方法的最后一行返回新创建的对象。

我们再来检查一下内存管理问题。`copy`方法应该返回一个保留计数器值为1的对象，且该对象不会自动释放。我们通过`alloc`方法获得这个新对象，`alloc`方法总是返回一个保留计数器值为1且我们不会释放的对象，因此内存管理问题顺利解决。

这正是我们使`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它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`类更难以复制。`Tire`类有两个实例变量（`pressure`和`treadDepth`），这两个实例变量需要被复制到`Tire`类的新对象中，而且`AllWeatherRadial`子类又引入了两个额外的实例变量（`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，其遵守协议采用语法的接口如下：

```
@interface Tire : NSObject <NSCopying>
{
    float pressure;
    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
```

你可以在该实现中看到`[[[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，你也可以使用简单的`init`方法和访问器方法来修改对象的属性。

方便的指针运算

你可以像下面这样使用C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，它的接口保持不变：

```
@interface AllWeatherRadial : Tire
{
    float rainHandling;
    float snowHandling;
}

// ... methods
@end // AllWeatherRadial
```

等一下，`<NSCopying>`哪里去了？你并不需要`<NSCopying>`，你也许能想通其中的原因。当AllWeatherRadial类继承Tire类时，它已经获得了Tire类的所有属性，包括对`NSCopying`协议的遵守。

不过，我们需要实现`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 setRainHandling: rainHandling];
    [tireCopy setSnowHandling: 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，也不需要使用我们前面曾经用过的`[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。

13.2.3 复制Car

由于我们已经能够复制Engine、Tire及其子类，下面我们就来复制Car类本身。

如你所料，Car类需要采用NSCopying协议：

```
@protocol Car : NSObject <NSCopying>
{
    NSMutableArray *tires;
    Engine *engine;
}
```

```
// ... methods
@end // Car
```

要履行对NSCopying协议的承诺，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;

    int i;
    for (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copyWithZone:方法多了少量代码，但所有这些代码都与你已经见到过的类似。

首先，通过给正在接收copy消息的对象所属的类发送allocWithZone:消息分配一个新的car对象：

```
Car *carCopy;
carCopy = [[[self class]
allocWithZone: zone]
init];
```

虽然CarParts-copy项目现在不包含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所属的类分配新对象来保障Car类的未来，就像我们以前所做的那样。

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，因此将没有人会出来为其发送最后的`release`消息以使其被销毁。如果那样的话，该engine对象将发生泄漏。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地方，而另一些编程人员则喜欢在创建点自动释放对象，以免以后忘记释放这些对象。两种办法都是有效的。

在carCopy保留了新的engine对象以后，`copyWithZone`方法执行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：

```
int i;
for (i = 0; i < 4; i++) {
    Tire *tireCopy;
    tireCopy = [[[self tireAtIndex: i] copy]
    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0的tire对象、位置1的tire对象，依次类推。然后，这些tire对象被复制并被自动释放，因而它们的内存可以被正确回收。接下来，

`carCopy`被告之使用同一位置的新的`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`tire`对象正常工作。

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，优美的新代码以粗体表示：

```
int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool;
    pool = [[NSAutoreleasePool alloc] init];

    Car *car = [[Car alloc] init];
    car.name = @"Herbie";

    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];
    car.engine = engine;
    [engine release];

    [car print];

    Car *carCopy = [car copy];
    [carCopy print];

    [car release];
    [carCopy release];

    [pool release];
    return (0);
} // main
```

该程序在输出原始的`car`对象以后，复制该`car`对象并将其输出。因此，我们应该得到两个完全相同的输出结果。运行该程序，你将看到像下面这样的结果：

```
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. VRROOM!
Herbie has:
AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
I am a slant-6. VRROOM!

```

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的编译器提供更多一点的信息，从而有助于检查你代码中的错误。

还记得吗，`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。你可以将任何对象赋值给一个`id`类型的变量，也可以将一个`id`类型的变量赋值给任何类型的对象指针。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）将知道你期望任意类型的对象，只要其遵守该协议。

例如，`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>) obj;
```

编译器在编译该方法时，将检查参数类型并提出警告，如“`class 'Triangle' does not implement the 'NSCopying' protocol`”。真是太方便了！

13.3 Objective-C 2.0 的新特性

Apple从来都是以追求臻善臻美而著称。Objective-C 2.0增加了两个新的协议修饰符：`@optional`和`@required`。等一下，我们刚才是说如果遵守一个协议就必须实现该协议的所有方法吗？是的，早期版本的Objective-C的确是这样。如果已经使用了豪华的Objective-C 2.0，那么你可以像下面这样编写优美的代码：

```

@protocol BaseballPlayer
- (void)drawHugeSalary;
@optional
- (void)slideHome;
- (void)catchBall;
- (void)throwBall;
@required
- (void)swingBat;
@end // BaseballPlayer

```

因此，一个采用BaseballPlayer协议的类有两个要求实现的方法：`-drawHugeSalary`和`-swingBat`，还有3个不可选择实现的方法：`slideHome`、`catchBall`和`throwBall`。

非正式协议似乎就可以胜任，为什么苹果公司还要这样设计呢？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。例如，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正在处理BaseballPlayer类的对象，谁领取丰厚的薪水、谁击球、谁滑垒、谁接球以及谁投球。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。从事任何类型的iPhone开发，你都将会注意到，Cocoa中的非正式协议逐渐被带有许多`optional`方法的正式协议所代替。

13.4 小结

本章介绍了正式协议的概念。可以通过在`@protocol`部分列出一组方法名来定义一个正式协议。通过在`@interface`声明中的类名之后列出用尖括号括起来的协议名称，对象可以采用该协议。当对象采用一个正式协议时，它承诺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，编译器将向你提出警告，从而帮助你履行自己的承诺。

另外，我们还探讨了伴随着面向对象编程而出现的一些细小的问题，主要是当复制位于类层次结构中的对象时发生的问题。

现在，祝贺你！你已经学习了Objective-C语言的绝大部分内容，并深入研究了在OOP中经常遇到的一些问题。你已经为继续学习Cocoa编程或开发自己的项目打下了坚实的基础。本书的下一章，将带你快速体验如何使用Interface Builder和AppKit编写图形化的Cocoa应用程序。Interface Builder和AppKit是Cocoa编程的灵魂，也是大多数Cocoa书籍和项目的中心主题，这些内容也非常有趣。之后，我们将进一步深入探讨Cocoa的一些更低层特性。

到目前为止，本书中的所有程序都使用了Foundation Kit，通过将文本输出发送到控制台的便捷方法与我们通信。这种方法很不错，但真正有趣的是可以构建与Mac类似的界面，其中包括可以点击和处理的内容。本章将介绍Application Kit（也称为AppKit）的一些重要特性，Cocoa的用户界面中包含很多有用的功能。

本章将构造的程序称为CaseTool，可以在14.01 CaseTool项目文件夹中找到。CaseTool显示一个窗口，该窗口类似于图14-1中的截图。该窗口包含一个文本域、一个标签和两个按钮。当向文本域输入一些文本并点击一个按钮时，输入的文本将转换为大写或小写。尽管这项功能确实很酷，但在花费5美元共享软件费用在VersionTracker上发布应用程序之前，你一定想添加其他有用功能。

14.1 构建项目

我们将使用Xcode和Interface Builder，并逐步引导你构建这个项目。要做的第一件事是创建项目文件。然后，我们对用户界面进行布局，最后将UI与代码连接起来。

首先进入Xcode，创建一个新Cocoa Application项目。运行Xcode，从File菜单选择New Project，选择Cocoa Application（如图14-2所示），并输入新项目的名称（如图14-3所示）。

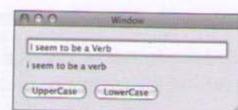


图14-1 完成的产品

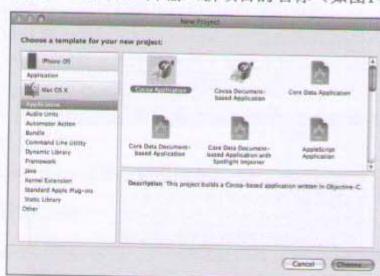


图14-2 创建新的Cocoa应用程序

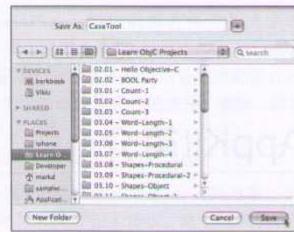


图14-3 命名新项目

现在，我们添加一个新的Objective-C类文件，命名为AppController，这样命名是因为它将是我们的应用程序的控制对象。在项目窗口的Groups & Files窗格中选择Sources文件夹。从File菜单选择New File。图14-4显示Xcode要求选择想要创建的文件类型（在本章中，选择Objective-C class），然后需要命名文件，如图14-5所示。确保选中了Also create 'AppController.h'复选框。

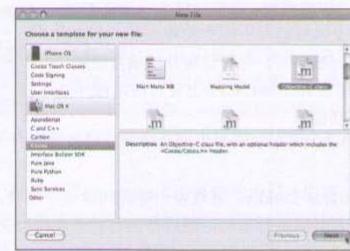


图14-4 创建新Objective-C类

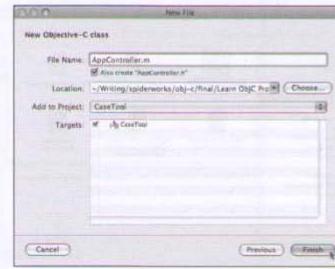


图14-5 命名新类

14.2 构建 AppController @interface

我们将使用Interface Builder应用程序来布局窗口的内容，在AppController与用户界面控件之间进行各种连接。Interface Builder也用于布局iPhone应用程序，因此无论为哪种平台编程，都需要在Interface Builder中花费大量时间。我们将向AppController类添加一些内容，然后Interface Builder将发现添加的内容，并让我们构建用户界面。

首先，建立AppController的头文件：

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

@interface AppController : NSObject {
    IBOutlet NSTextField *textField;
    IBOutlet NSTextField *resultsField;
}

- (IBAction) uppercase: (id) sender;
- (IBAction) lowercase: (id) sender;

@end // AppController
```

这里有两个类似关键字的词：IBOutlet和IBAction。它们实际上只是AppKit提供的#define。IBOutlet的定义没有任何作用，因此将不会对它进行编译。IBAction定义为void，这意味着在AppController中声明的方法的返回类型将是void（也就是什么都不返回）。

如果IBOutlet和IBAction不执行任何操作，那么为什么还要定义它们呢？答案是，它们不是用于编译的，IBOutlet和IBAction实际上是为Interface Builder以及阅读代码的人提供的标记。通过查找IBOutlet和IBAction，Interface Builder知道AppController对象具有两个能够连接的实例变量，AppController提供两个方法作为按钮单击（和其他用户界面操作）的目标。我们稍后将介绍具体原理。

在Interface Builder中，将textField实例变量连接到一个NSTextField对象。这个文本域是用户输入要转换的字符串的地方，通常是NSTextField扮演的角色。

resultsField将被连接到一个只读NSTextField。当处于只读模式时，NSTextField的表现就像一个文本标签。这个文本域是显示字符串的大写或小写形式的地方。

uppercase：是单击UpperCase按钮时调用的方法。参数sender是用户单击的NSButton对象。有时候，可以查看一个操作的sender参数来获得与所发生事情相关的附加信息。但是对于CaseTool应用程序，我们将忽略它。

lowercase：是单击LowerCase按钮时调用的方法。

14.3 Interface Builder

现在是时候使用Interface Builder了，一些朋友也将其称为IB。我们想要编辑项目附带的MainMenu.xib文件。这个文件配备了一个菜单栏，以及一个可以加入用户控件的窗口。

在Xcode项目窗口中，找到并双击MainMenu.xib（参见图14-6）。

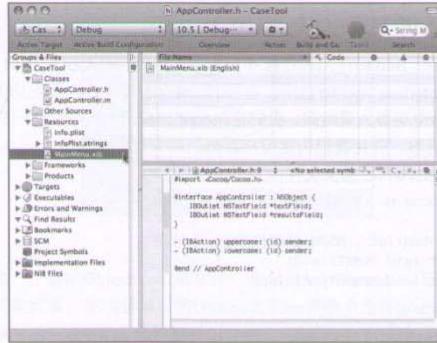


图14-6 打开MainMenu.xib

这将启动Interface Builder来打开该文件。虽然文件扩展名为.xib，我们仍将其称为nib文件。“nib”是NeXT Interface Builder的缩写，是Cocoa的一个工件（artifact），由NeXT公司开发。nib文件是包含被冻结的对象的二进制文件，而.xib文件是XML格式的nib文件。在编译时，.xib文件将编译为nib格式。

IB打开文件之后，其界面与图14-7相似，包含4个窗口。首先看一下左上部，这是一个IB固定窗口，包含表示nib文件的内容的图标。这是nib文件的主窗口。下面是应用程序的一个非常短（不要因此忽略它）的菜单栏。可以添加新菜单和菜单项，还可以编辑现有菜单项。本程序不会涉及这些功能。

菜单栏下面是一个空窗口，我们将在其中放置文本域和按钮。这个真实的活动窗口与固定窗口中的微型窗口形状的图标相对应。只要双击该图标，就会打开该窗口。这些窗口的右侧是IB

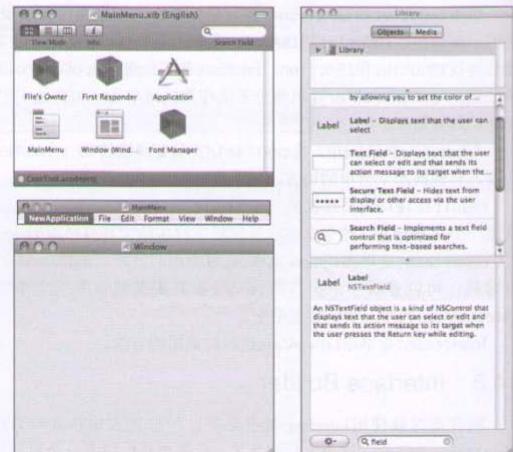


图14-7 Interface Builder界面

Library面板。这个面板包含可以拖到窗口中的对象，其中包含很多内容。可以在底部的搜索框中输入一些文本来减少显示的内容。为了便于理解，可以使用的每种对象都有一段描述内容。

现在我们继续使用Interface Builder来构建程序。我们将告诉Interface Builder创建一个AppController对象。当程序运行时，Cocoa将加载nib文件，我们将有一个新AppController对象供处理。但是，首先需要创建AppController。

将一个NSObject从库中拖到CaseTool.xib固定窗口中。它将有一个形象的名称Object，如图14-8所示。

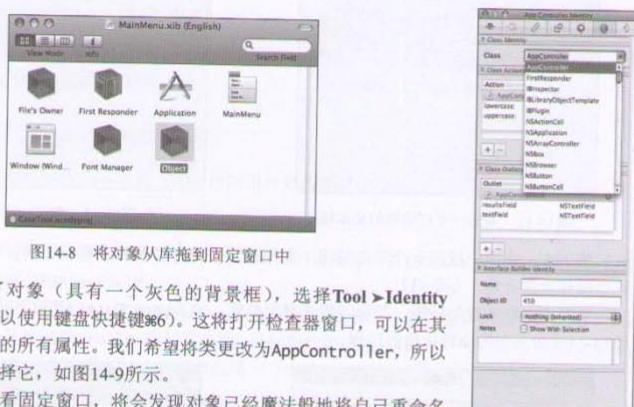


图14-8 将对象从库拖到固定窗口中

确保选中了对象（具有一个灰色的背景框），选择Tool > Identity Inspector（也可以使用键盘快捷键⌘6）。这将打开检查器窗口，可以在其中调整所选对象的所有属性。我们希望将类改更为AppController，所以从下拉菜单中选择它，如图14-9所示。

如果现在查看固定窗口，将会发现对象已经魔法般地将自己重命名为AppController，如图14-10所示。

图14-9 更改对象的类



图14-10 对象名自动改更为AppController

14.4 布局用户界面

现在布局用户界面。在库中找到一个Text Field（不是Text Field Cell），将其拖到窗口中，如

图14-11所示。将元素拖到窗口中时，将会显示蓝色的引导线。这有助于根据苹果用户界面规范布置对象。

现在拖出一个Label。从库中找到一个Label对象，将其拖到窗口中，如图14-12所示。这是将显示大写和小写结果的地方。



图14-11 拖出一个可编辑的文本域

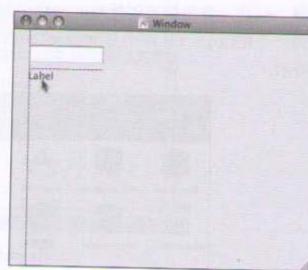


图14-12 拖出一个标签

接下来，在库中找到一个单击按钮，并拖到窗口中。将其放在标签下面，如图14-13所示。这些操作非常有趣，不是吗？

现在双击新放置的按钮，其标签将变得可编辑。输入UpperCase，并按下回车键接受编辑。图14-14展示了实际编辑按钮的过程。

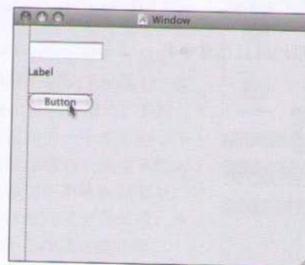


图14-13 将一个按钮拖到窗口中



图14-14 编辑按钮的标签

从面板中拖出另一个按钮，并将其标签更改为LowerCase。图14-15显示了添加第二个按钮之后的窗口。

接下来，稍微进行一下内部装饰，调整文本域和窗口本身的大小，使其更加美观，如图14-16所示。我们也对Label的大小进行调整，使其横跨窗口的宽度。标签必须足够宽，以显示输入到字段中的文本。现在，窗口的外观已符合我们的期望。

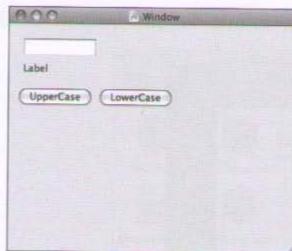


图14-15 所有项都已添加



图14-16 窗口布局井然有序

14.5 连接

本节将介绍如何将代码与我们刚才创建的用户界面元素相连接。

14.5.1 连接输出口

现在是时候连接了。首先，需要告诉AppController对象，其textField和resultsField实例变量应该指向哪个NSextField。

首先，对窗口进行排列，使MainMenu.xib固定窗口和带有文本域的窗口同时可见。接下来，按下Control键并从AppController拖到文本域。鼠标指针经过之处将显示一条蓝线。拖到文本域，如图14-17所示。拖到文本域上时，应该会出现一个Text Field标签。



图14-17 开始连接

当释放鼠标按钮时，将显示一个包含可能的IB输出口的菜单。选择textField选项，如图14-18所示。

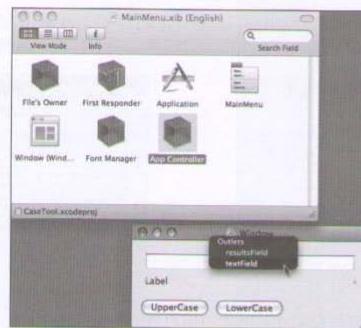


图14-18 进行连接

现在，按住Control键并从AppController拖到Label，选择resultsField项以完成连接。

仔细检查所进行的连接，选择检查器的Connections面板或使用键盘快捷键 $\text{⌘}5$ 。应该可以在检查器的顶部看到两个连接，如图14-19所示。

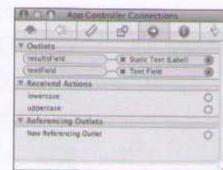


图14-19 仔细检查连接

14.5.2 连接操作

现在可以将按钮连接到操作了，这样，按下按钮就会触发代码。按住Control键并拖动鼠标，进行所需的连接，现在从按钮连接到AppController。

说明 拖动连接的路径是使用Interface Builder时一个容易引起混淆的地方。拖动的方向是从需要知道某些内容的对象到该对象需要了解的对象。

AppController需要知道将哪个NSTextField用于用户输入，因此拖动方向是从AppController到文本域。

按钮需要知道告诉哪个对象：“嘿！有人按下了我！”因此从按钮拖到AppController。

按住Control并单击UpperCase按钮，拖一条线到AppController，如图14-20所示。

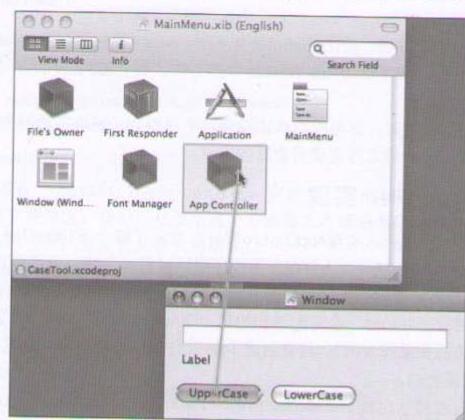


图14-20 连接UpperCase按钮

将一条线从按钮拖到AppController之后，在检查器中选择uppercase:，如图14-21所示。



图14-21 进行连接（第二次）

现在,只要单击该按钮,uppercase:消息就会发送到AppController实例,这和我们希望该按钮所做的一样。然后可以在uppercase:方法中执行想要的操作。

最后,连接LowerCase按钮。按住Control键并从LowerCase按钮拖到AppController,然后选择lowercase:。现在已经完成了Interface Builder中的工作。保存nib文件,如果愿意,可以退出Interface Builder。

在布局对象和进行连接时,你可能动作比较慢。不用担心,熟练之后操作速度就会变快了。熟练的IB操作员可以在一分钟之内完成所有这些步骤。

14.6 AppController 实现

现在让我们编写一些代码,实现AppController。首先了解一下IBOutlet是如何工作的。

当加载nib文件时(MainMenu.nib会在应用程序启动时自动加载,可以创建你自己的nib文件并自行加载),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AppController实例。在执行init方法期间,所有IBOutlet实例变量都为nil。只有创建了nib文件中的所有对象(这包括窗口和文本域和按钮),所有连接才算完成。

一旦建立了所有连接(也就是将NSTextField对象的地址添加到AppController的实例变量中),会向创建的每个对象发送消息awakeFromNib。一个非常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,发送给他们的所有消息不执行任何操作,所以在init中的任何尝试都会发生无提示失败(这是Cocoa导致效率降低和占用大量调试时间的一个方面)。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,并查看它们是否都为nil。对于创建的对象和发送的awakeFromNib消息,都不存在预定义的顺序。

我们继续分析AppController的实现。下面是一些必需的准备步骤:

```
#import "AppController.h"

@implementation AppController
接下来是一个init方法,用于在初始化时显示IBOutlet实例变量的值(它们将会是nil):
- (id) init
{
    if (self = [super init]) {
        NSLog(@"init: text %@ / results %@", textField, resultsField);
    }
    return (self);
} // init
```

要得到更加漂亮的用户界面,我们应该将文本域设置为合理的默认值,而不是Label。虽然Label是一个正确的默认值,但它没有太大意义。我们在文本域中输入Enter text here,将结果域预

设为Results。awakeFromNib是完成此任务的理想位置：

```
- (void) awakeFromNib
{
    NSLog(@"awake: text %@ / results %@", textField, resultsField);
    [textField setStringValue:@"Enter text here"];
    [resultsField setStringValue:@"Results"];
}
```

NSTextField拥有一个setStringValue:方法，它接受一个NSString作为参数，并更改文本域的内容来反映该字符串值。我们正是使用这个方法将文本域更改为对用户更有意义的内容。

现在看一下操作方法，首先它具有大写形式：

```
- (IBAction) uppercase: (id) sender
{
    NSString *original;
    original = [textField stringValue];
    NSString *uppercase;
    uppercase = [original uppercaseString];
    [resultsField setStringValue: 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，但是每个字母都是大写形式。然后将该字符串设置为resultsField的内容。

现在是时候进行必不可少的内存管理检查了：是否每部分都是正常的？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，并且将被清除。setStringValue:负责复制或保留传入的字符串。setStringValue:所做的事是其份内的事。但我们知道我们的内存管理是正确的。

lowercase:与uppercase:一样，但它将所有字母都改为小写形式：

```
- (IBAction) lowercase: (id) sender
{
    NSString *original;
    original = [textField stringValue];
    NSString *lowercase;
    lowercase = [original lowercaseString];
    [resultsField setStringValue: lowercase];
}
```

就这么简单。当运行程序时，你会看到一个窗口。输入一个字符串并更改其大小写，就像图14-22一样。

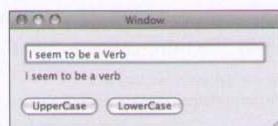
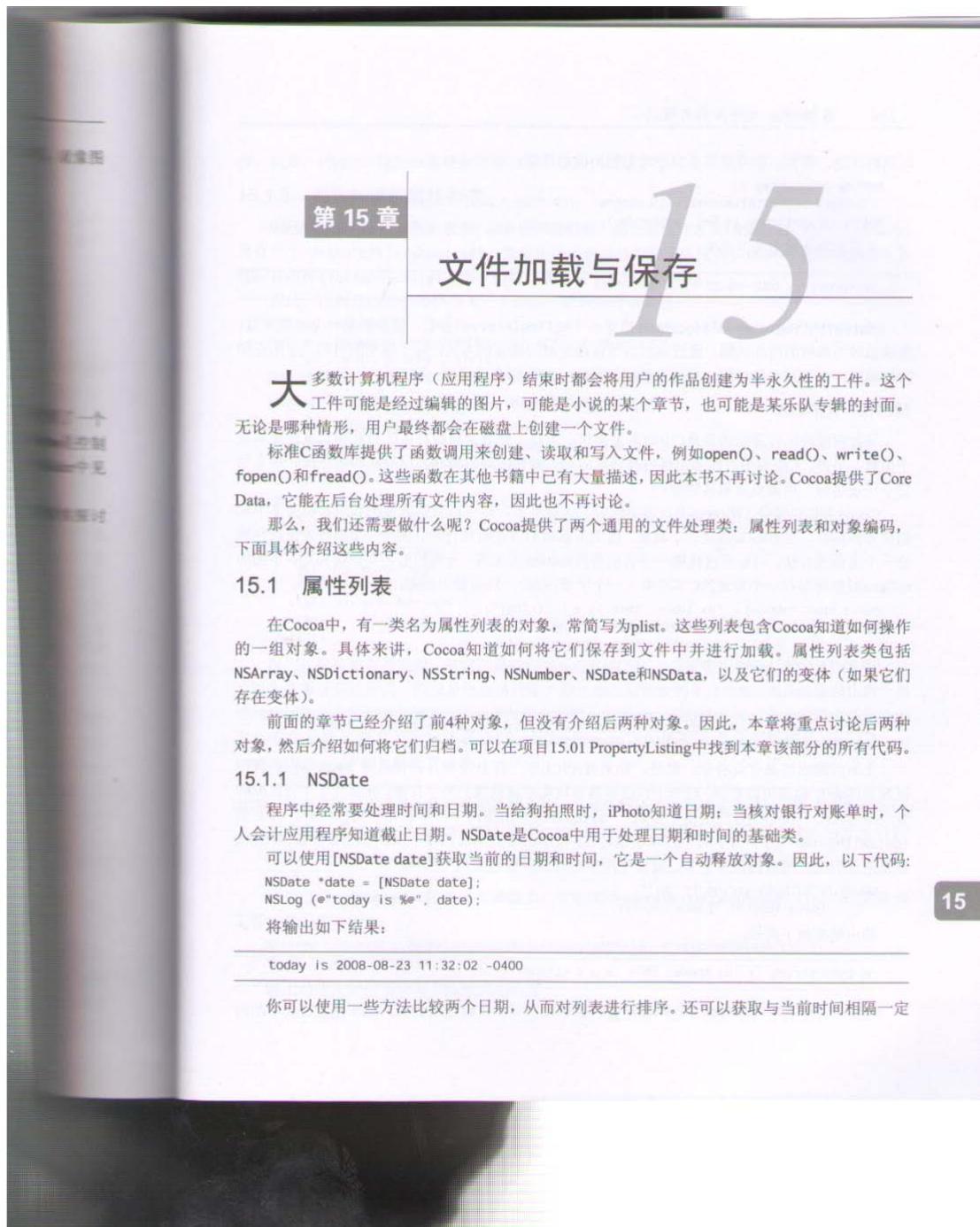


图14-22 完成后的CaseTool程序运行正常

14.7 小结

本章只简单介绍了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 (NSButton丰富了按钮的功能，NSWindow是控制窗口的对象)。AppKit中有一百多种不同的类可供使用，其中许多都可以在Interface Builder中见到。

现在，你应该已经准备好深入学习Cocoa知识或构建Cocoa项目了。下一章将继续探讨Foundation Kit的一些低级特性。



时差的日期。例如，你可能需要24小时之前的确切日期：

```
NSDate *yesterday =
[NSDate dateWithTimeIntervalSinceNow: -(24 * 60 * 60)];
NSLog(@"%@", yesterday);
```

上面的将会输出如下结果：

```
yesterday is 2008-08-22 11:32:02 -0400
```

`+dateWithTimeIntervalSinceNow:`接受一个`NSTimeInterval`参数，该参数是一个双精度值，表示以秒为单位的时间间隔。通过该参数可以指定时间偏移的方式：对于将来的时间，使用正的时间间隔；对于过去的时间，使用负的时间间隔（和我们在此做的一样）。

15.1.2 NSData

将数据缓冲区传递给函数是C中的常见操作。为此，你通常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，由谁负责将其清除？

Cocoa为我们提供了`NSData`类，该类包装了大量字节。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，适用于常规的内存管理行为。因此，如果将数据块传递给一个函数或方法，可以通过传递一个自动释放`NSData`来实现，无需担心内存清除问题。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: string
                               length: strlen(string) + 1];
NSLog(@"%@", 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，如果有ASCII表（打开终端，并键入命令`man ascii`就可以找到该表），你就可以看到，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，`-bytes`方法给出指向字符串起始位置的指针。注意到`+dataWithBytes:`中的`+ 1`了吗？它用于包含C字符串所需的尾部的零字节。你还会注意到`NSLog`结果末尾的`00`。通过包含零字节，可以使用`%s`格式的说明符输出字符串：

```
NSLog(@"%@", [data bytes]);
```

输出结果如下所示：

```
30 byte string is 'Hi there, this is a C string!'
```

`NSData`对象是不可改变的。它们被创建后就不能改变。可以使用它们，但不能更改其中的内

容。但是，`NSMutableData`支持在数据内容中添加和删除字节。

15.1.3 写入和读取属性列表

你已经看到了所有属性列表类，能够使用它们呢？集合属性列表类(`NSArray`、`NSDictionary`)具有一个`-writeToFile:atomically:`方法，用于将属性列表写入文件。`NSString`和`NSData`也具有`writeToFile:atomically:`方法，但它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，然后保存该数组：

```
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>
```

以上代码虽然有些繁琐，但它正是我们要保存的内容：一个字符串数组。这些属性列表文件可以为任意复杂的形式，可以是包含字符串、数字和日期数组的字典数组。Xcode还包含一个属性列表编辑器，所以可以查看plist文件并进行编辑。如果看一下操作系统，你会发现许多属性列表文件和系统配置文件，如主目录`Library/Preferences`下的所有首选项文件和`/System/Library/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 filename.plist`，可以将这些文件转换成人们可读的形式。

现在已将`verbiage.txt`文件存放在磁盘上，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);
```

输出结果正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？这种调用有什么负面作用吗？没有。`atomically`参数的值为`BOOL`类型，用于通知Cocoa是否应该首先将文件内容保存在临时文件中，当文件成功保存后，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。除非保存的文件非常大，将会占用用户硬盘的空间，否则应该自动保存文件。

如果能将数据精简为属性列表类型，则可以使用这些非常便捷的调用来将内容保存到磁盘中，供以后读取。如果你正在从事一项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。即使只想把数据块保存到磁盘中，并且根本不需要使用对象，也可以使用`NSData`来简化工作。只需将数据包装在一个`NSData`对象中，然后在`NSData`对象上调用`writeToFile:atomically:`。

这些函数的一个缺点是，它们不会返回任何错误信息。如果不能加载文件，只能从方法中得到`nil`指针，而不能确定出现了何种错误。

15.2 编码对象

遗憾的是，无法总是将对象信息表示为属性列表类。如果能将所有对象都表示为数组字典，我们就没有必要使用自己的类了。所幸，Cocoa具备一种机制来将对象自身转换为某种格式并保存到磁盘中。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。以后将这些数据块读回到内存中，并且还能基于保存的数据创建新对象。这个过程称为编码和解码，或称为序列化和反序列化。

在第14章中介绍Interface Builder时，我们从库中将对象拖到窗口中，这些对象被保存到nib文件中。换言之，`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，串行化对象，同时创建新的`NSWindow`和`NSTextField`对象并将其相互关联起来。

你可能会猜到，通过采用`NSCoding`协议，可以使用自己的对象实现相同功能。该协议与下面的代码类似：

```
@protocol NSCoding
- (void) encodeWithCoder: (NSCoder *) aCoder;
```

```
- (id) initWithCoder: (NSCoder *) aDecoder;
@end
```

通过采用该协议，可以实现这两种方法。当对象需要保存自身时，`-encodeWithCoder`方法将被调用；当对象需要加载自身时，`-initWithCoder`方法将被调用。

那么这个编码器是什么呢？ `NSCoder`是一个抽象类，定义一些有用的方法来在对象与`NSData`之间来回转换）。完全不需要创建新 `NSCoder`，因为它实际上并无多大作用。但是我们实际上要使用 `NSCoder`的一些具体的子类来编码和解码对象。我们将使用其中两个子类`NSKeyedArchiver`和`NSKeyedUnarchiver`。

利用示例可能是理解这些对象的最容易的方式。可以查看项目15.02 SimpleEncoding获取所有代码。

首先看一下一个含有实例变量的简单类：

```
@protocol 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个对象实例变量和标量类型，其中包括一个集合。属性默认为`readwrite`，因此我们没有在属性定义中指定该属性。每个实例变量都具有一些公共属性，以及一个方便的一站式`init`方法，可用于从头创建新`Thingie`类。

注意，`Thingie`类采用`NSCoding`协议，这意味着我们将实现`encodeWithCoder`和`initWithCoder`方法。至于现在，我们先将这两个方法设置为空：

```
@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
```

以上代码将初始化一个新对象，清除我们生成的所有无用信息，创建存根方法来使编译器适合NSCoding协议，并返回相关描述。

注意，在init方法中，我们在赋值表达式的左侧使用了self.attribute。请记住，这实际上意味着我们需要为这些属性调用访问方法，这些方法由@synthesize创建，我们不直接对实例变量进行赋值。这种对象创建技术将为传入的NSString和我们创建的NSMutableArray提供适当的内存管理方法，因此我们不必为这些对象明确提供内存管理方法。

因此，在main()函数中，我们构造一个Thingie并输出：

```
Thingie *thing1;
thing1 = [[Thingie alloc]
```

```
initWithName: @"thing1"  
magicNumber: 42  
shoeSize: 10.5];  
  
 NSLog(@"%@", thing1);
```

以上代码将输出如下结果：

```
some thing: thing1: 42/10.5 (
```

这很有趣。现在，我们保存该对象。并按以下方式实现Thingie的encodeWithCoder::

```
- (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/值对保存对象的信息。Thingie的-encodeWithCoder方法在与每个实例变量名称匹配的键下编码这些实例变量。也可以不这样做，可以在键flarblewhazzit下编码实例变量名称。使键的名称与实例变量的名称尽可能相似，这样便于识别它们之间的映射关系。

可以使用这样的裸字符串作为编码键；也可以定义一个常量来避免录入错误。可以使用#define kSubthingiesKey @"subThingies"等定义常量，也可以使用文件的局部变量，例如static NSString *kSubthingiesKey = @"subThingies"。

注意，每种类型都有不同的encodeSomething:forKey:。一定要使用合适的方法来对类型进行编码。对于任何Objective-C对象类型，都可以使用encodeObject:forKey:。

如果需要恢复某个对象，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:和任何其他init方法一样，在对对象执行操作之前，需要使用超类对它们进

行初始化。为此，可以采用两种方式，具体取决于父类。如果父类采用`NSCoding`协议，则应该调用`[super initWithCoder: decoder]`；否则，只需要调用`[super init]`即可。`NSObject`不采用`NSCoding`协议，因此我们可以使用简单的`init`方法。

当使用`decodeIntForKey:`时，把一个`int`值从解码器中取出。当使用`decodeObjectForKey:`方法时，把一个对象从解码器中取出，在嵌入的对象上递归使用`initWithCoder:`方法。内存管理的工作方式与你预期的一样：从除`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。我们的属性声明用于确保正确地进行内存管理。

你将会注意到，编码和解码的顺序和实例对象的顺序完全相同。当然，你也可以不这样做，这只是一个简单的习惯，目的在于确保每个对象都进行了编码和解码，而不会被忽略。这也是使用键进行调用的原因之一——可以将实例对象按任意顺序放入和取出。

接下来将实际使用这些对象。我们使用前面创建的`thing1`对象，并将其归档：

```
NSData *freezeDried;  
freezeDried = [NSKeyedArchiver archivedDataWithRootObject: thing1];
```

`+archivedDataWithRootObject:`类方法编码`thing1`对象。首先，它在后台创建一个`NSKeyedArchiver`实例；然后，它将`NSKeyedArchiver`实例传递给对象`thing1`的`-encodeWithCoder`方法。当`thing1`编码自身的属性时，它可能对其他对象也进行编码，例如字符串、数组以及我们可能输入到该数组中的任何内容。整个对象集合完成对键和值的编码后，具有键/值对的归档程序将所有对象扁平化为一个`NSData`类并将其返回。

如果愿意，可以使用`-writeToFile:atomically:`方法将这个`NSData`类保存到磁盘中。在此，我们将先处理`thing1`对象，然后通过`freezeDried`表示法重新创建它，并将它输出：

```
[thing1 release];  
thing1 = [NSKeyedUnarchiver unarchiveObjectWithData: freezeDried];  
NSLog (@"reconstituted thing: %@", thing1);
```

输出结果与前面的结果完全相同：

```
reconstituted thing: thing1: 42/10.5 (
```

在契诃夫话剧的第一幕中，当看到墙上有一支枪时，你一定觉得很奇怪。同样，你可能对`subThingies`可变数组很好奇。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`的`encodeWithCoder:`实现在所有对象上调用`encodeWithCoder`方法，最后使所有对象都被编码。让我们向`thing1`添加一些`subThingies`：

```
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(@"%@", thing1);
以上代码将输出thing1和subthings:

```

```

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(@"%@", thing1);
上面的代码将输出与前面相同的结果。

```

如果被编码的数据中含有循环将会怎么样？例如，如果thing1包含在自身的subThingies数组中会怎么样？thing1会对数组进行编码吗？哪个对象对thing1进行编码，哪个对象对数组进行编码，哪个对象再次对thing1进行编码，依此类推？幸运的是，Cocoa在归档程序和解压程序实现上非常灵活，能够保存并恢复对象周期。

要查看会出现的结果，将thing1放入其自身的subThingies数组中：

```
[thing1.subThingies addObject: thing1];
```

但是，不要尝试在thing1中使用NSLog类。NSLog类不能检测对象循环，它将执行一个无限递归来尝试构造日志字符串，最后，将会有成千上万的-description调用进入调试器中。

但是，如果现在尝试对thing1进行编码和解码，它将能完美地完成工作，而不会陷入混乱状态：

```
freezeDried = [NSKeyedArchiver archivedDataWithRootObject: thing1];
thing1 = [NSKeyedUnarchiver unarchiveObjectWithData: freezeDried];
```

15.3 小结

在本章中，我们看到Cocoa提供了两种方式来加载和保存文件。属性列表数据类型是一种类集合，它知道如何加载和保存其自身。如果对象集合中的对象类型全为属性列表，可以使用这种简单便捷的函数将这些对象保存到磁盘中，或者从磁盘中将它们读出。

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，则可以采用NSCoding协议和实现方法来编码和解码对象：可以将自己的大量对象转换成NSData类，然后将它保存到磁盘中，并可以在以后读取它。通过这种NSData类，可以重新创建对象。

下一章将介绍键/值编码，它使你能在更高的抽象级别上与对象交互。

现在回过来看一下间接机制。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。许多人将键/值编码亲切地称为KVC，它是一种间接更改对象状态的方式，其实现方法是使用字符串描述要更改的对象状态部分。本章通篇只介绍键/值编码。

一些更加高级的Cocoa特性，例如Core Data和Cocoa Bindings（本书将不介绍该内容），在基础机制中包含了KVC。

16.1 入门项目

我们将再次使用大家熟悉的CarParts示例。看一下项目16.01 Car-Value-Coding。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 <NFFirstSCopying> {
    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;  
...
```

```
另外，我们还更新了-copyWithZone方法，以添加新属性：
```

```
- (id) copyWithZone: (NSZone *) zone  
{  
    Car *carCopy;  
    carCopy = [[[self class]  
               allocWithZone: zone]  
              init];
```

```
carCopy.name = name;  
carCopy.make = make;  
carCopy.model = model;  
carCopy.modelYear = modelYear;  
carCopy.numberOfDoors = numberOfDoors;  
carCopy.mileage = mileage;  
// plus copying tires and engine, code in chapter 13.
```

```
同时，我们更改了-description方法，以输出这些新属性并省略Engine和Tire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设置这些属性，并将它们输出。同时，我们结合使用了  
autorelease与alloc和init方法，这样可以在一个位置执行所有内存管理：
```

```
int main (int argc, const char * argv[]){  
    NSAutoreleasePool *pool;  
    pool = [[NSAutoreleasePool alloc] init];  
  
    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);

[pool release];

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.
```

16.2 KVC简介

键/值编码中的基本调用包括-`valueForKey:`和-`setValue:forKey:`。以字符串的形式向对象发送消息，这个字符串是我们关注的属性的关键。

因此，我们可以请求`car`的名称：

```

NSString *name = [car valueForKey:@"name"];
NSLog (@"%@", name);

```

以上代码将为我们输出`Herbie`。同样，我们可以获取品牌信息：

```

NSLog (@"make is %@", [car valueForKey:@"make"]);

```

`valueForKey:`的功能非常简单，它计算车的品牌并将其返回。

`valueForKey:`首先查找以键-`key`或-`isKey`命名的getter方法。对于这两种调用，`valueForKey:`查找-`name`和-`make`。如果不存在getter方法，它将在对象内部查找名为`_key`或`key`的实例变量。如

果我们没有通过@synthesize提供存取方法, `valueForKey`将会查找实例变量`_name`和`name`, 或`_make`和`make`。

最后一点非常重要: `-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, 可以获取不存在getter方法的对象值, 无需通过对象指针直接访问实例变量。

可以对型号年份对象使用相同的技术:

```
NSLog(@"%@", [car valueForKey: @"modelYear"]);
```

输出结果为: `model year is 1984`。

嘿, 再等一下! `NSLog`中的`%@`输出一个对象, 但`modelYear`是一个`int`值, 而不是对象。该如何处理呢? 对于KVC, Cocoa自动放入和取出标量值。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具有这种自动包装功能。常规方法调用和属性语法不具备该功能。

除用于检索值外, 还可以使用`-setValue:forKey:`按名称设置值:

```
[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`, 并使用参数`@"Harold"`调用它。如果不存在setter方法, 它将在类中查找名为`name`或`_name`的实例变量, 然后为它赋值。

谨记这条规则

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。这条规则实际上不是强制的, 但如果遵循它, 你可能会遇到某种风险。

如果在调用`-setValue:forKey:`之前设置一个标量值, 你需要将它包装起来(把它打包):

```
[car setValue: [NSNumber numberWithFloat: 25062.4]
          forKey: @"mileage"];
```

同时, `-setValue:forKey:`在调用`-setMileage:`或更改`mileage`实例变量之前会取出该值。

16.3 路径

除了通过键设置值外, 键/值编码还支持指定键路径, 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`添加一个新的实例变量:

```
@interface Engine : NSObject <NSCopying> {
    int horsepower;
```

```

}
end // Engine

```

注意，我们没有添加任何存取方法或特性。通常，我们希望为关注的对象属性使用存取方法或特性。但这里我们避免使用它们，以便真实地展示KVC直接深入到对象中的功能。

为了以非零马力启动引擎，我们添加一个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horsepower实例变量的副本，使副本能够获取相应的值，然后将它添加到description方法中。我们现在已经很熟悉这个方法了，因此不再详细解释。

为了证实我们能够获取并设置值，以下代码：

```

 NSLog(@"%@", [engine valueForKey: @"horsepower"]);
 [engine setValue: [NSNumber numberWithInt: 150]
             forKey: @"horsepower"];
 NSLog(@"%@", [engine valueForKey: @"horsepower"]);

```

将会输出：

```

horsepower is 145
horsepower is 150

```

如何表示这些键路径呢？可以指定以圆点分隔的不同属性名称。这样，通过查询car的“engine.horsepower”，就能够获取马力值。我们实际尝试一下使用-valueForKeyPath和-setValueForKeyPath方法访问键路径。将以下消息发送给car，而不发送给Engine：

```

[car setValue: [NSNumber numberWithInt: 155]
         forKeyPath: @"engine.horsepower"];
NSLog(@"%@", [car valueForKeyPath: @"engine.horsepower"]);

```

这些键路径的深度是任意的，具体取决于对象图（对象图是一种表示相关对象集合的有趣方式）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系列嵌套方法调用更容易访问对象。

16.4 整体操作

关于KVC非常棒的一点是，如果向NSArray请求一个键值，它实际上会查询数组中的每个对

象来查找这个键值，然后将查询结果打包到另一个数组中并返回给你。这种方法也适用于通过键路径访问的对象内部的数组。(还记得复合吗？)

在KVC中，通常认为嵌入到其他对象中的NSArray具有一对多的关系。例如，汽车与多个轮胎(一般有4个)存在联系。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的每个对象。

一对一关系

你现在已经了解了一对多关系，你还可能想知道什么是一对一的关系。普通对象的构成都是一对一关系。例如，汽车与其引擎之间就是一对一的关系。

记住，Car具有一个轮胎数组，并且每个轮胎都有自己的空气压力。通过一次调用，我们可以获取所有的轮胎压力值：

```
NSArray *pressures = [car valueForKeyPath: @"tires.pressure"];
```

执行以下调用：

```
 NSLog(@"%@", pressures);
```

输出结果如下：

```
pressures (34, 34, 34, 34)
```

除了让我们知道轮胎状态之外，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请求轮胎信息。获取轮胎信息后，它通过键路径的其余部分请求tires对象的valueForKeyPath：，在本例中为“pressure”。NSArray实现valueForKeyPath：方法，循环查找其内容并将消息发送给每个对象。因此，使用“pressure”作为键路径，NSArray向内部的每个轮胎发送一个valueForKeyPath：，结果会将轮胎压力值打包到NSNumber中并返回。非常方便！

然而，不能在键路径中为这些数组添加索引，例如，通过使用“tires[0]. pressure”获取第一个轮胎的压力值。

16.4.1 中途小憩

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汽车。可以在名为16.02 Car-Value-Garaging的项目中找到这个新类。下面是Garage的接口：

```
#import <Foundation/Foundation.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。我们提前声明了Car，因为只需要知道这个对象类型被用作-addCar:方法的参数。名称是一种特性，并且@property语句表明Garage用户可以访问和更改名称。我们可以使用一个方法来输出内容。为了实现一个汽车集合，我们在后台使用一个可变数组。

实现同样很简单：
#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 // Garage
```

像往常一样，我们包含Garage.h头文件并使用@synthesize合成名称存取方法。

-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用于清理名称和数组，-print遍历数组并输出各种类型的汽车。

与前面的程序版本相比，我们还全面修改了主要的Car-Value-Garage.m源文件。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 "Siant6.h"
#import "Tire.h"

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;

    Siant6 *engine = [[[Siant6 alloc] init] autorelease];
    [engine setValue: [NSNumber numberWithInt: horsepower]
        forKey: @"horsepower"];
    car.engine = engine;

    // Make some tires.
    int i;
    for (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约定构造并自动释放了一个新Car，因为通

过这个函数获得Car的实体自身不会调用new、copy或alloc方法。然后，我们设置一些属性。记住，这项技术与KVC不同，我们没有使用setValue:forKey方法。接下来，我们构造一个Engine并使用KVC设置马力值，因为我们没有为它构造存取方法。最后，构造一些tire并将它们安置在汽车中。最终返回新Car。

以下是新版的main()函数：

```
int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool;
    pool = [[NSAutoreleasePool alloc] init];

    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];
    [pool release];

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

`main()`函数进行了一些登记，构造了一个`garage`，并构建了一些`car`保存在`garage`。最后，主函数输出`garage`的对像，并将其释放。

运行程序，可以得到以下令人激动的输出：

```
Joe'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，我们基本了解了键/值编码的下一个优势。

16.4.2 流畅地运算

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，例如获取一组值的平均值或返回这组值中的最小值和最大值。

例如，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We have 7 cars。

我们将键路径“`cars.@count`”拆开，`cars`用于获取`cars`属性，它是来自`garage`的`NSArray`类型的值。我们知道，它是一个`NSMutableArray`，但如果我不打算更改任何内容，可以将它视为`NSArray`。接下来的部分是`@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，`@interface`用于引入一个类。此处的`@count`用于通知KVC机制计算键路径左侧的结果。

此外，我们还可以计算某些特定值的总和，例如，车队行驶的总英里数。以下代码段：

```
NSNumber *sum;
sum = [garage valueForKeyPath: @"cars.@sum.mileage"];
NSLog(@"%@", @"We have a grand total of %@" miles, sum);
```

将输出We have a grand total of 601320.6 miles，该数字足够我们在地球与月球之间往返一次了。

这项功能是如何实现的？`@sum`运算符将键路径分成两部分。第一部分可以看成多对多关系的键路径，在本例中代表`cars`数组。另一部分可以看成包含一对多关系的任何键路径。它被当做用

于关系中每个对象的键路径。`mileage`被发送到`cars`描述的关系中的每个对象，然后将这些值相加。当然，每个键路径的长度是任意的。

如果需要得到平均每辆汽车行驶的距离，可以将总数再除以汽车数量。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/值编码的优点，我们必须对这些汽车（假设我们从`garage`中获得`cars`数组）编写一个循环，查找每辆汽车的行驶距离，将它们累加，然后除以汽车数量——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`之后的部分是一个键路径，它仅表示距离。在后台，KVC能够轻松地进行循环，将值累加，并进行计数，然后进行除法运算。

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`。

KVC不是免费的

KVC能非常轻松地处理集合。那么，为什么不使用KVC来处理所有对象，并且抛弃那些方法和代码编写？天下没有免费的午餐，除非你在某个硅谷技术大公司工作。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`cars.@avg.mileage`，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。

有时使用的属性仅含有一小组值，例如上面构造的所有汽车。即使我们有100万辆汽车，品牌的种类也会很少。通过使用键路径“`cars.@distinctUnionOfObjects.make`”，就可以从集合中得到需要的品牌：

```
NSArray *manufacturers;
manufacturers =
  [garage valueForKeyPath: @"cars.@distinctUnionOfObjects.make"];
NSLog(@"%@", @"makers: %@", manufacturers);
```

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.
```

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发生变化。

在本程序中，这个工具不是很有用，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 Lift and Stamp工具，可以在其他图片上更改某个图片的部分内容。可以使用`dictionaryWithValuesForKeys`方法调整所有属性，并将字典的所有内容显示在用户界面中。用户可以从字典中读取这些属性，然后调用`setValuesForKeysWithDictionary`方法，使用修改过的字典更改其他图片。设计好用户界面类以后，可以将相同的Lift and Stamp面板应用于不同的对象，例如照片、汽车和食谱。

由于字典不能包含`nil`值，如果出现`nil`值将会出现什么情况（例如汽车没有名称）？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"`下将返回`[NSNull null]`。也可以为`setValuesForKeysWithDictionary`提供`[NSNull null]`，但顺序相反。

16.6 `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`？表示`pi`？Cocoa无法知道`nil`表示什么。你可以尝试以下代码：

```
[car setValue: nil forKey: @"mileage"];
```

但Cocoa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`-setNilValueForKey`，提供逻辑上有意义的任何值。在此，我们约定，`nil` `mileage`表示清除汽车行驶的距离，而不使用其他值（例如`-1`）来实现此功能：

```
- (void) setNilValueForKey: (NSString *) key {
    if ([key isEqualToString: @"mileage"]) {
        mileage = 0;
    } else {
        [super setNilValueForKey: key];
    }
} // setNilValueForKey
```

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，如果某人试图对某个我们不理解的键使用键/值编码，调用者将会得到适当的警告消息。一般来说，除非有某些特殊的好原因（例如特意避免某个特殊操作），否则，当重写`-setNilValueForKey`时，应该总是调用超类方法。

16.7 处理未定义的键

我们的键/值之旅（花了3小时吗）的最后一站是处理未定义的键。如果你使用过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无法确定你使用这个键的意图，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。也许你能够猜到，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`setValue:forUndefinedKey:`方法。

如果KVC机制无法找到处理方式，它会返回询问类如何处理。默认实现会取消操作，如前文所示。但是我们可以更改默认行为。我们将`Garage`转换为一个非常灵活的对象，通过这个对象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;
}
...
```

接下来添加`valueForUndefinedKey`方法：

```
- (void) setValue: (id) value forKey: (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
```

并使用`-dealloc`释放字典。

现在可以设置`garage`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"];
```

然后返回它们：

```
NSLog(@"values are %@ %@ %@ and %@",  
      [garage valueForKey: @"fluffy"],  
      [garage valueForKey: @"bork"],  
      [garage valueForKey: @"snorgie"],  
      [garage valueForKey: @"gronk"]);
```

这个`NSLog`将输出以下结果：

```
values are bunny greeble <null> and (null)
```

注意`<null>`与`(null)`之间的区别。`<null>`是一种`[NSNull null]`对象，而`(null)`是一个真实存在的`nil`值。由于字典中没有“gronk”，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。如果为`NSDictionary`的`setObject:forKey:`提供`nil`值，它将会给出警告信息，但是，如果在字典中将`setValue:forKey:`设置为`nil`值，将会把对应的键从字典中删除。

16.8 小结

虽然本章介绍的内容只是键/值编码的很小一部分，但你现在应该为学习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`方法来完成你要求的操作。如果KVC无法找到任何方法，它将直接进入对象并更改值。

此外，我们还介绍了键路径，它们是由点分隔的键，用于在对象网络中指定路径。也许这些键路径看起来很像访问属性，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，我们介绍了可以进行重写来定制个别行为（corner-case behavior）的方法。

下一章将介绍Cocoa的谓词特性。

NSPredicate 17

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，从而提供一些有意义的对象。

在使用软件（例如iPhoto）的过程中，经常会看到这种现象。如果通知iPhoto仅显示等级为三星级或三星级以上的图片，则指定的条件为“照片的等级必须为三星级或三星级以上”。这样，所有照片都需要经过该过滤器过滤。满足条件的对象通过了过滤器，而其他对象被筛选了。最后，iPhoto将显示出所有高质量的图片。

类似地，iTunes也有自己的搜索框。如果搜索条件是歌唱家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循环播放列表。

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，通过该对象准确地描述所需的条件，对每个对象通过谓词进行筛选，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”不相同。这里的“谓词”通常用在数学和计算机科学概念中，表示计算真值或假值的函数。

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，例如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式查询。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（Open/Closed Principle）的另一个应用。

17.1 创建谓词

在将NSPredicate应用于某个对象之前，首先需要创建它。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，如果正在构建通用用户接口来指定查询，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，本书中我们将重点介绍查询字符串。常见的面向字符串的API警告信息适用于查询字符串，特别适用于缺少编译器错误检查及有时出现奇怪的运行时错误等情况。

我们仍然使用CarParts示例，本章的示例基于上一章创建的汽车车库示例。你可以在17.01 Car-Part-Predicate项目中查找完整的代码。

首先看一下一辆汽车的情况。代码如下所示：

```
Car *car;
car = makeCar(@"Herbie", @"Honda", @"CRX", 1984, 2, 110000, 58);
[garage addCar: car];
```

在前面，我们已编写了makeCar函数，用于构造一辆汽车，并为它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58，已行驶距离为110 000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是一个常用的Objective-C对象指针，它将指向NSPredicate对象。使用NSPredicate类方法+predicateWithFormat:来实际创建谓词。将某个字符串赋给谓词，+predicateWithFormat:使用该字符串在后台构建对象树，这些树将用来计算谓词的值。

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样式格式说明符插入某些内容。稍后你将看到，可以使用predicateWithFormat方法实现相同的功能。Cocoa采用一致的命名模式，这样做非常好。

这种谓词字符串看上去像是标准的C表达式。它的左侧是键路径name，随后是一个等于运算符“==”，右侧是一个引用字符串。如果谓词字符串中的文本块未被引用，则该谓词字符串被看做是键路径；如果引用了文本块，则认为它是文本字符串。你可以使用单引号或者双引号（只要它们匹配即可）。通常应该使用单引号；否则，必须在字符串中对每个双引号进行转义。

计算谓词

通过以上步骤我们就可以得到一个谓词。接下来做什么呢？我们将通过某个对象计算谓词！

```
BOOL match = [predicate evaluateWithObject: car];
NSLog(@"%@", (match) ? "YES" : "NO");
```

-evaluateWithObject:通知接收对象（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（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。以上代码段将输出最高马力的汽车：

```
Elvis
Phoenix
Judge
```

合理吗？不。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**中的汽车名称调用。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，它可能在后台包含各种强大的功能。

17.2 燃料过滤器

编程人员最显著的优点/缺点都是懒惰。如果我们不必编写**for**循环和**if**语句，这有什么不好吗？程序中仅有几个代码行，但不包含代码就更好了。幸运的是，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的对象累积到将被返回的新数组中：

```
NSArray *results;
results = [cars filteredArrayUsingPredicate: predicate];
NSLog(@"%@", results);
```

以上过程将输出以下结果：

```
{
    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；在前面的示例中，我们得到的结果是名称。我们可以使用KVC（键/值编码）提取其中的名称。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"];
```

如果输出名称，将看到以下结果：

```
(  
    Elvis,  
    Phoenix,  
    Judge  
)
```

假设有一个可变数组，你需要删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];
```

如果输出carsCopy，结果将是前面我们看到的3辆汽车的集合。

也可以使用-*filteredArrayUsingPredicate*:方法和NSMutableArray数组来构建新（不可变）数组，因为NSMutableArray是NSArray的超类。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，这种循环并不会带来很大的性能影响，因为当今的计算机运行速度非常快。继续编写尽可能简易的代码。如果你遇到速度问题，可以使用苹果公司的工具（如Shark或Instruments）测试程序性能。不过，iPhone编程人员应该随时密切关注程序的性能。

17.3 格式说明符

资深编程人员都知道，硬编码并非好办法。如果需要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，我们该怎么办？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字符串中：格式说明符和变量名。首先，我们将介绍格式说明符。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，可以通过用户接口或某些扩展机制来驱动它。

除了使用printf说明符，也可以使用%@插入字符串值，将%@看成是一个引用字符串：

```
predicate = [NSPredicate predicateWithFormat: @"name == %@", @"Herbie"];
```

注意，这里的格式字符串中没有引用%@。如果你需要引用%@，例如“name == '%@'”，应该将字符%和@放在谓词字符串中。

通过NSPredicate字符串，也可以使用%K指定键路径。该谓词和其他谓词相同，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所见过的其他谓词完全相同。

可以使用不同的对象作为变量名称，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大于150。

除了使用NSNumber和NSString之外，也可以使用[NSNull null]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紹。注意，不能用\$VARIABLE作为键路径，因为它只表示值。使用谓词格式字符串时，如果想通过程序改变键路径，需要使用%K格式说明符。

谓词机制不进行类型检查。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消息，或者出现其他不可预知的操作。

17.4 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。

17.4.1 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=。

不等号运算符具有各种形式：

>: 大于

>=和>=: 大于或等于

<: 小于

<=和<=: 小于或等于

!=和!=: 不等于

此外，谓词字符串语法还支持括号表达式（确实如此！）和AND、OR、NOT逻辑运算符或者C样式的等效表达式&&、||和!。

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(@"%@", results);
```

如果将以上代码应用到车队中，我们将得到以下结果：

```
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或or。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。

不等号既适用于数值又适用于字符串值。如果需要按字母表顺序从开始查看所有汽车，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
)
```

17.4.2 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是数组的下界，第二个元素看成是数组的上界。

也可以使用%@格式说明符插入你自己的NSArray对象：

```
NSArray *between = [NSArray arrayWithObjects:
    [NSNumber numberWithInt: 50],
    [NSNumber numberWithInt: 200], nil];
predicate = [NSPredicate predicateWithFormat:
    @"engine.horsepower BETWEEN %@", between];
```

也可以使用变量：

```
predicateTemplate =
    [NSPredicate predicateWithFormat: @"engine.horsepower BETWEEN $POWERS"];
varDict =
    [NSDictionary dictionaryWithObjectsAndKeys: between, @"POWERS", nil];
predicate =
    [predicateTemplate predicateWithSubstitutionVariables: varDict];
```

数组并不仅仅用来指定某个区间的端点值。你可以使用IN运算符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"]);
```

果然，仅返回以下两个对象：

```
(  
    Herbie,  
    Badger  
)
```

17.5 SELF 足够了

某些时候，可能需要将谓词应用于简单的值（例如那些纯文本老式字符串），而并非那些可以通过键路径进行操作的对象。假设我们有一个汽车名称数组，并且需要应用前面相同的过滤器。从NSString对象中查询name时，将不能起到预期效果，那么，我们用什么来代替name呢？

用SELF来解决！SELF可以引用用于谓词计算的对象。事实上，我们可以将谓词中所有的键路径表示成对应的SELF。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名称数组中，该怎么办呢？我们来分析一下。

首先，需要从某处获取仅含有名称的数组。因为已经熟悉了CarPart中的各种对象，因此，我们将借助于数组，使用KVC技术获取valueForKey:方法，以便处理这些对象示：

```
NSArray *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", @"Finto", nil];

predicate = [NSPredicate predicateWithFormat: @"SELF IN %@", names1];
results = [names2 filteredArrayUsingPredicate: predicate];
NSLog(@"%@", results);
```

答案如下所示：

```
(  
    Judge,  
    Badger  
)
```

对于取两个数组的交集的运算而言，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，它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。在两个数组中同时存在的字符串将会保留在name2中，并使SELF IN子句确定它是正确的，因此，结果数组中将会包含那些满足条件的字符串。如果第二个数组中的对象与谓词中的任何字符串都不匹配，则该对象将被筛选掉。第一个数组中的字符串将保留在原位，用于作比较，而且绝不会将它放到结果中。

17.6 字符串运算符

前面使用字符串时，我们介绍过关系运算符。此外，还有以下一些针对字符串的关系运算符。

- **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 BEGINSWITH '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"é"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”，d表示“不区分发音符号”（即没有重音符），[cd]表示“既不区分大小写，又不区分发音符号”。你永远不会知道用户何时会按下大写锁定键，以全大写的形式与应用程序交互。

该谓词字符串会将Herbie与" name BEGINSWITH[cd] 'HERB'"相匹配。

17.7 LIKE 运算符

某些时候，将一个字符串的开头或结尾（或中间）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还接受[cd]修饰符，用于忽略大小写和发音符号。

如果你热衷于正则表达式，可以使用MATCHES运算符。赋给该运算符一个正则表达式，谓词将会计算出它的值。

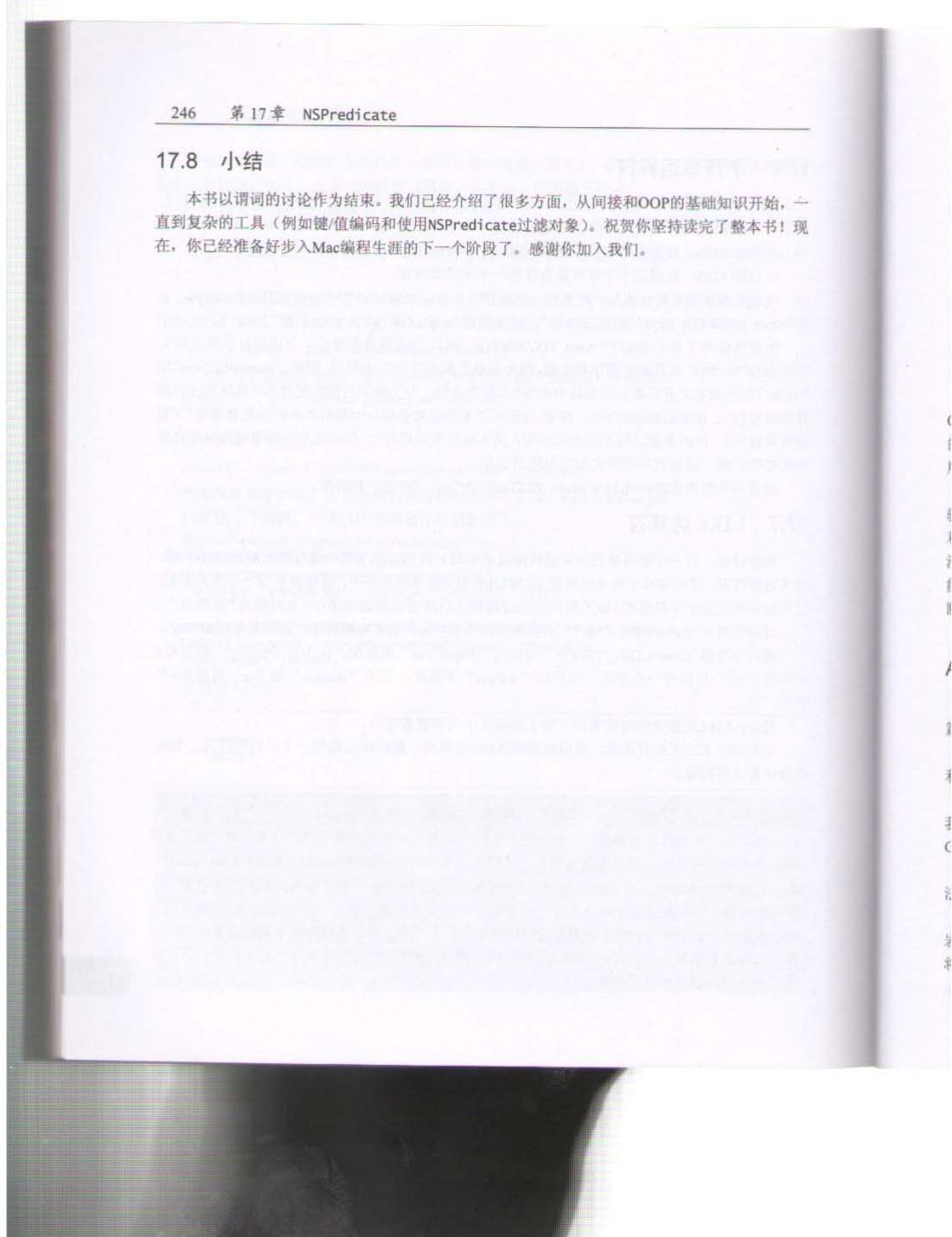
表达自己

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，正则表达式的形式可能会变得复杂而费解，已经有大量书籍讨论了这一主题。NSPredicate正则表达式使用International Components for Unicode (ICU) 语法，你可以借助因特网搜索引擎了解有关内容。

虽然正则表达式的功能强大，但它们的计算开销非常大。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。

17.8 小结

本书以谓词的讨论作为结束。我们已经介绍了很多方面，从间接和OOP的基础知识开始，一直到复杂的工具（例如键/值编码和使用NSPredicate过滤对象）。祝贺你坚持读完了整本书！现在，你已经准备好步入Mac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存在很大差异，因此，大家在学习过程中会遇到很多困难。初次接触Objective-C的编程人员会经常抱怨它不是一种好语言，和他们最喜欢的语言相比，缺少某些有用的特性。我们暂不考虑Objective-C的一些特性，摆在我面前的事实是：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都有丰富编程经验的编程人员来说，我们的建议是：先撇开以前形成的工作观念，暂时独立接受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，你就可以了解哪些技术和方法来自其他语言，并适用于Cocoa和Objective-C，而哪些技术和方法并非如此。任何语言都不可能适用于所有的环境，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，在此我们将不再一一重复，但在需要时，还会讨论一些有意义的主题。

Objective-C编程人员可以使用与C绑定的所有工具，例如标准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，有些人在Cocoa的邮件列表上询问：“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低级框架（例如Core Foundation和Core Graphics）时，也会遇到这样的问题。

简单的回答是：“不能这样做。”回调仅适用于那些通过库签名的C函数。实现Objective-C方法的函数首先必须含有自身参数和选择器参数，否则，它们就不可能与必要的签名相匹配。

多数面向回调的库都需要你为它们提供某些用户数据或指针。可以将用户数据或指针看成是岩石，你可以隐藏在其后面。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会返回对应的指针。

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，假设你正在使用假想的C API XML解析库，解析XML文件时，向TreeWalker类提供数据。首先，我们构造新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对象作为上下文：
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提供了有关特性和技术，用于代替或模拟这些特性。

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。类别和协议非常适合这种场合。还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时，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。通过这种技术，可以避免编写很多小存根方法。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，例如使用名称前缀替代命名空间。

A.2.1 C++虚拟表与 Objective-C 动态分配

C++与Objective-C的最大区别在于分配方法（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。

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，它将从虚拟表的开头计算偏移位置，并发送机器代码使函数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，Objective-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++路由要慢得多。

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，成员函数的分配速度要更快。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链接器执行类型安全链接的方式）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/或委托。缺少多重继承可能会给你带来诸多不便，但是，我们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的类型安全性。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中没有类型安全容器，任何对象都可以放入某个容器中。

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，这样可以避免可能出现的运行时错误。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，你可能更希望使用C++来实现，或者至少使用模糊的名称，而不会使用类似于`SerialNumberVerifier`的类或方法名。

在Objective-C中，可以直接向`nil`（零）对象发送消息，而不必检查发送的消息是否为`NULL`。

向nil对象发送的消息代表停止操作指令。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比其更小的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。在另一方面，这种技术会掩盖某些错误，并由此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，一些更高级的实体开始（例如NSPoint和NSRange）成为取代一类对象的结构。

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，你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，没有可靠的方法可用于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（在本附录的前面部分已讨论了该内容）。

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。一方认为，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。但另一方认为，这些仅是一些实现细节，类用户不必关注。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类似（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
```

然后，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个具有不同根的树。

A.2.2 Objective-C++

还有一种两全齐美的方式。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，可以使用类型安全和低级性能；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。可以获得C++的性能和类型安全，用户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+文件系统是不区分大小写的，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在某个类中或在栈上进行声明。需要首先在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;
};
```

可以将C++对象放入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，当解除分配Objective-C对象时，C++对象会调用它们的析构函数。

A.3 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，经典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，但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节。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，阻止更改其中的任何子类。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），在本书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，例如使用此类的代码将使用的任何新的枚举、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。很多编程人都认为Java中缺少预处理器是一个良好的特性。

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通常仅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A.2节。

在Objective-C中，通过使用类别向现有类中添加方法，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的大小，所以任何类只要包含子类头文件，就可以把它设置为子类。

实际上，相对于Java而言，在Objective-C中很少使用子类化行为。因为，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，可以将某些功能放到含有较少功能的类中，也可以将功能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那些特性。

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），或者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式。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，不必实现所有的数据源和委托方法。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能够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，这样容易产生一些棘手的内存错误。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为基础的，因此，可以使用C语言的所有特性，同时包括使用预处理器可能带来的危险，并可能出现与指针相关的内存管理错误。

A.4 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，但当他们转向Cocoa和Objective-C时可能变得非常困惑。

BASIC（Visual和REAL）环境提供的集成开发环境由完整的工作区所组成。Cocoa将开发环境分成两个部分：Interface Builder和Xcode。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，然后使用Xcode（或TextMate、BBEdit、emacs，以及任何你喜欢的文本编辑器）将控制逻辑添加到源代码中。

在BASIC中，用户界面项目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，然后将它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将它集成到代码库中，而不必自己去构建它。

A.5 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，例如非常健壮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器）、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），以及可以下载并使用的大量程序包。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，你也许会认为Objective-C在很多方面落后于时代的发展。Objective-C语言源于80年代，而脚本语言源于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()样式输出字符串也和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链接阶段，这导致编写代码和查看结果之间存在延迟。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，可以使用C语言携带的所有特性，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？性能是原因之一，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Tk语言开发的Tk工具包。这种工具包简单易用，但它所具备的用户界面特性远不及Cocoa。最后一个原因是，使用Tk构建的应用程序的外观和感觉都不同于Mac程序。

不过，使用脚本作为桥梁，可以获取两家之长。通过在Objective-C和Python之间（称为PyObjC），以及Objective-C和Ruby之间（称为RubyObjC）搭建桥梁，这两种脚本语言能够成为Objective-C的一类（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.6 小结

Objective-C和其他编程语言及工具不同。由于它的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。特别是健壮的字符串处理、命名空间及元编程特性，其他语言中早就具备了这些特性，但你无法在Objective-C中使用它们。

编程中所做的每件事都需要进行权衡。需要确定，与当前使用的语言相比，使用Objective-C所获得的优势能否弥补失去的功能。对我们而言，要能够使用Cocoa工具构建应用程序，需要花费的时间和付出的努力要比熟悉Objective-C语言多得多。