苹果开发
与应用系列

HZ BOOKS
华章科技

PEARSON

（美） Robert Clair 著
李强 等译

Objective-C 2.0 Mac 和iOS开发实践指南

LEARNING OBJECTIVE-C 2.0
A HANDS-ON GUIDE TO OBJECTIVE-C FOR MAC AND IOS DEVELOPERS

机械工业出版社
China Machine Press

苹果开发
与应用系列

（美） Robert Clair 著
李强 等译

Objective-C 2.0 Mac 和iOS开发实践指南

LEARNING OBJECTIVE-C 2.0
A HANDS-ON GUIDE TO OBJECTIVE-C FOR MAC AND IPHONE DEVELOPERS

机械工业出版社
China Machine Press

本书专门为那些想要学习Objective-C以便为Mac OS X或iOS编写程序的程序员量身打造。本书分为四部分。第一部分介绍学习Objective-C编程所需的C语言基础知识。第二部分介绍Objective-C编程的核心知识，包括消息机制、类和对象、类对象等。第三部分介绍Objective-C中的高级概念，包括引用计数、垃圾收集和Objective-C 2.0的新功能块。第四部分提供了Objective-C程序员常用的信息和资料。本书大多数章末都提供了练习，读者可以通过练习巩固在书中学到的知识。

本书内容精练、可读性强、易于学习，侧重通过示例来介绍知识点和概念，是一本学习Objective-C 2.0不可多得的入门实践指南。本书适合想要学习Objective-C 2.0编程语言的初、中级程序员阅读，也可以作为社会培训机构的入门级培训教辅材料。

本书版权登记号：图字：01-2010-5851

图书在版编目（CIP）数据

Objective-C 2.0 Mac和iOS开发实践指南 /（美）克莱尔（Clair, R.）著；李强等译．—北京：机械工业出版社，2011.1

（苹果开发与应用系列）

书名原文：Learning Objective-C 2.0: a Hands-On Guide to Objective-C for Mac and iOS Developers

ISBN 978-7-111-32927-5

Ⅰ.O… Ⅱ.① 克… ② 李… Ⅲ.① 操作系统，Mac OS X－程序设计 ② 面向对象语言－程序设计 Ⅳ.① TP316.89 ② TP312

中国版本图书馆CIP数据核字（2010）第261509号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：秦 健
北京市荣盛彩色印刷有限公司印刷
2011年2月第1版第1次印刷
186mm × 240mm · 18.75印张
标准书号：ISBN 978-7-111-32927-5
定价：55.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991，88361066
购书热线：（010）68326294，88379649，68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

对本书的赞誉

在本书中，作者直切主题，不仅全面介绍了Objective-C，而且从深入实际、注重实践的角度给出了节省读者时间和精力的入门简介。精练的概括、示例及具体实现的细节，能帮助读者快速、完整地理解这一语言及其核心功能和概念。

——Scott D. Yelich，移动应用开发者

有很多关于Objective-C的图书试图介绍面向对象编程、Objective-C计算机语言和Apple平台上的应用开发的整个内容。如此范围的主题太宽泛了，岂是单独一本篇幅有限的图书所能完全囊括的？本书专注介绍Objective-C的基础知识，它能帮助你成为一名编写Objective-C代码的合格程序员。

——Joseph E. Sacco博士，J.E. Sacco & Associates公司

本书是一本大师级的教程，它针对Objective-C语言提供了深入的、有趣的介绍，对于程序员新手和资深专业人士都有启发意义。当有刚接触Objective-C的程序员询问该从何处开始时，我会推荐这本书给他们。

——Matt Long, Cocoa Is My Girlfriend (www.cimgf.com)网站

本书作者精通Objective-C语言，并且以一种易于学习的方式来介绍它。不管你是新手还是专业程序员，都可以在本书中有所收获，而这并不需要预先有C语言的知识。

——Cory Bohon，MaclLife独立开发者和博客

我喜欢这本书，因为它的技术内容不枯燥，可读性好且有深度。

——Andy Lee，AppKiDo的作者

译 者 序

当今移动互联网正在推动互联网第三波浪潮的到来。根据摩根士丹利的调查报告，2010年全球的移动互联网用户已达到10亿人，产值近千亿美元。而中国的手机上网用户也已达到2.77亿个，由此产生的市场总值在200亿元人民币左右。移动互联网不仅将深入改变人们的生活方式，而且会造就一个又一个新的数字财富英雄。

在移动互联网浪潮中，移动应用开发将起到重要作用。App Store的成功模式给移动应用开发搭建了良好的商业模式和环境。目前有越来越多的开发者加入到移动开发的队伍中。而基于Mac OS和iPhone的移动应用开发则是其中重要的一个分支。

Objective-C最初由Smalltalk演变而来，它是基于C语言的面向对象扩展。目前，Objective-C 是Mac OS的首要开发语言，也是GNUstep 在Linux 和其他平台上的开发语言。随着基于Mac OS和iPhone、iPad开发的兴起，Objective-C受到越来越多的关注，其流行度也在不断攀升。在TIOBE编程语言排行榜2010年10月的榜单中，Objective-C排在第8位，而与去年相比，Objective-C的份额提升最多，达到2.54%，同时入选了2010年度语言。因此，关注与学习Objective-C的人越来越多。

本书集中介绍了Objective-C语言，专门为那些想要学习Objective-C以便为Mac OS X或iOS编写程序的程序员量身打造（iOS用于iPhone、iPod Touch和iPad）。本书分为四部分。第一部分回顾了学习Objective-C编程所需的C语言基础知识，并简单介绍了Objective-C程序的结构。第二部分介绍Objective-C编程的核心知识，包括消息机制、类和对象、类对象、框架、Foundation类、控制结构、分类、特性、协议，等等。第三部分介绍了Objective-C中的高级概念，包括引用计数、垃圾收集和Objective-C 2.0的新功能块。第四部分包括5个附录，分别提供了Objective-C程序员常用的信息和资料。本书大多数章末都提供了练习，以帮助读者巩固在书中学到的知识。本书内容注重实用，侧重通过示例来介绍知识点和概念，因此本书对于初中级程序员来说，是一本不可多得的入门实践指南。

参加本书翻译的有李强、关志兴、王建勇、毛立涛、闫柳青、姜巧生、沈海峰、谢扣林、乔义峰、刘查强、王义强、刘国际、杨传辉和王建华等。读者在阅读过程中如有疑问，可通过translatebook@163.com与译者联系交流。

前　言

Objective-C是C语言的面向对象扩展。可以将其称为“带有对象的C”。如果你对学习Objective-C感兴趣，通过学习可以针对Mac OS X或iOS编写程序。但是，还存在学习Objective-C的另一个原因，那就是它是一门有趣的语言，并且相对容易学习。和现实世界的任何其他东西一样，Objective-C也有一些难点。但是，总体来说，它是比其他面向对象语言简单得多的一种语言，特别是和C++相比。Objective-C对C语言所进行的扩展，只需要一两页篇幅就能列举出来。

在Apple的世界中，Objective-C并不是单独工作的。它与称为框架的两个类库一起使用。Foundation框架包含了用于基本实体的类，例如，字符串和数组，以及包装了与操作系统进行交互的类。AppKit包含了用于窗口、视图、菜单、按钮及构建图形用户界面所需的其他各种挂件的类。将这两个框架综合起来就是Cocoa。在iOS中用一个叫做UIKit的不同框架取代了AppKit。Foundation和UIKit合称为Cocoa Touch。

Objective-C由Brad J. Cox在20世纪80年代早期创建。在1988年，由Steve Jobs在离开Apple后创建的NeXT Computer公司许可使用Objective-C，并且以其作为创建在NeXT的NeXTSTEP操作系统下运行的应用程序开发环境的基础。NeXT工程师开发了一组Objective-C库，以便构建应用程序。在NeXT于1933年减少硬件业务之后，它与Sun Microsystems合作开发了OpenStep，这是针对一种面向对象系统的开放性规范，以NeXTSTEP API为基础。Sun最终丧失了对OpenStep的兴趣。NeXT继续销售其OpenStep版本，直到Apple于1997年年初收购了NeXT。NeXTSTEP操作系统变成了Mac OS X的基础。NeXT Objective-C库变成了Cocoa的基础。

本书集中介绍了Objective-C语言。它不是教你如何编写Cocoa程序，或者使你成为Xcode的专业用户。本书只是涉及了Foundation框架的一小部分，并且只是介绍性地提及AppKit和UIKit。本书所做出的假设是，如果你首先对Cocoa所基于的语言有一个较好的理解，那么你学习Cocoa将会变得更容易。

本书的目标读者

本书是为那些想要学习Objective-C以便为Mac OS X或iOS编写程序的程序员量身打造的(iOS用于iPhone、iPod Touch和iPad)。尽管从技术上讲，也有可能使用其他的语言来编写完整的Mac OS X程序，但编写一个遵从Apple Human Interface Guidelines[⊖]并且拥有相应的Mac观感的应用程序，则需要使用Objective-C Cocoa框架。即便你使用不同的语言，例如纯C或C++，来编写一个Mac应用程序的核心，用户界面层还是应该使用Objective-C编写。在为iOS编写程

⊖ 参见http://developer.apple.com/mac/library/documentation/UserExperience/Conceptual/AppleHIGuidelines。

序时，程序员没有任何选择：iPhone应用程序的外层和用户界面必须使用Objective-C编写。

对于那些想要使用来自GNUStep项目[⊖]的软件为其他平台编写Objective-C程序的程序员来说，本书也很有用（GNUStep是OpenStep库的一个开源实现）。

你需要知道什么

本书假设读者拥有C语言的使用经验。Objective-C是C的一个扩展，本书主要关注Objective-C向C中添加了什么。对于那些具有C经验的人，以及那些适合快速学习一门新语言的人来说，本书第2章和第3章对C的基本部分进行了回顾，这些基本知识是编写一个Objective-C程序很可能要用到的。如果你没有C或任何类似C（C++、Java和C#）的计算机语言的经验，那么你需要在学习本书的同时阅读一本关于C的图书。此前接触过一种面向对象语言对学习本书有帮助，但不是必需的。随着本书的推进，将会分别介绍必需的面向对象概念。

Objective-C 2.0中的新内容

如果你已经知道一些Objective-C的知识，并且想要直接跳到2.0版新增内容的部分，那么关于这部分的知识在如下的各章中介绍：

- 快速枚举（第10章）为遍历对象的一个集合提供了一种简单（且快速）的方式。
- 声明特性（第12章）提供了一种简单的方式来指定一个对象的实例变量，并且让编译器为你创建访问这些变量的方法。
- 垃圾收集（第15章）添加了对Objective-C的自动内存管理。
- 块（第16章）允许你定义类似函数的对象，它们自身带有环境。

本书是如何组织的

本书分为四部分：第一部分是C的概览，然后介绍面向对象编程和Objective-C。本书的第二部分详细介绍了Objective-C语言，并且对Foundation框架进行了介绍。本书的第三部分介绍了在Objective-C中使用的两种内存管理形式，以及Objective-C 2.0新添加的块功能。本书的第四部分是附录，提供了Objective-C程序员常用的信息和资料。

第一部分　Objective-C简介

- 第1章概述了C的基本知识，介绍了在编写Objective-C程序时最可能用到的C知识。
- 第2章继续概述C，讨论C和Objective-C程序的内存布局，以及内存寻址和不同类型变量的生命周期。即便你熟悉C，也可能需要阅读本章。很多有经验的C程序员并不完全熟悉本章介绍的内容。
- 第3章开始介绍面向对象编程的概念，并且继续探讨这些概念是如何在Objective-C中体现的。

⊖ 参见www.gnustep.org。

- 第4章详细介绍了一个简单的Objective-C程序的组成。还介绍了如何使用Xcode来创建一个项目，然后编译并运行一个Objective-C程序。可以使用这些知识来完成本书中其余章的练习。

第二部分　语言基础

对象是面向对象编程中的主要实体，它们将变量（称为实例变量）和类似函数的代码块（叫做方法）组织到一个单独的实体中。类是对象的规范。它们列出了组成一个给定的对象的实例变量和方法，并且提供了实现这些方法的代码。对象更加具体，它们是一个内存范围，类似于C结构，保存了对象的类所定义的变量。一个特定的对象称为定义了它的类的一个实例。

- 第5章开始对Objective-C语言进行全面介绍。在Objective-C中可以通过向一个对象发送一条消息来让它“做某些事情”。消息是一个方法的名称加上方法所接收的任何参数。作为对接收到的消息的响应，对象会执行相应的方法。本章介绍方法、消息，以及Objective-C消息系统是如何工作的。
- 第6章介绍如何定义类，以及如何创建和复制对象实例。这一章还介绍了继承，即通过扩展一个已有的类来定义一个类的过程，而不是从头开始创建一个类。

  在执行一个Objective-C程序的过程中，使用的每个类都由包含关于类的信息的一段内存来表示。这段内存叫做类的类对象。类也可以定义类方法，它是类执行的方法，而不是由类的实例执行的。
- 第7章介绍类对象和类方法。与某些其他面向对象语言中的类不同，Objective-C的类没有类变量，即类的所有实例所共享的变量。这一章的最后几节介绍如何通过使用静态变量来达到类变量的效果。
- 第8章介绍Apple封装动态链接库的方式。这一章介绍了框架的定义和结构，并简单介绍了在编写Mac OS X或iOS程序时将会遇到的一些常见框架。
- 第9章介绍最常用的Foundation类：用于字符串、数组、字典、集合和数字对象等的类。
- 第10章讨论当使用带有C控制结构的Objective-C构造时，需要考虑的一些问题。本章继续介绍Objective-C所添加的其他控制结构，包括Objective-C 2.0的新的快速枚举构造。这一章最后介绍了Objective-C的异常处理系统。
- 第11章介绍如何向一个已有的类添加方法而不需要子类化它，以及如何隐藏你认为私有的方法的声明。这一章最后讨论Objective-C的安全性问题。
- 第12章介绍Objective-C 2.0最新声明的特性功能。特性是一个对象的特征。特性通常由对象的实例变量来构建。设置和获取一个特性的方法叫做访问器方法。通过使用声明特性功能，你可以要求编译器合成一个特性的访问器方法，从而大大节省自己的精力。
- 第13章介绍分类对象的一种不同方法。协议是定义的一组方法，一个类可以选择实现这些方法。在很多情况下，重要的不是一个对象的类，而是对象的类是否通过实现协议中声明的方法来采用一个特定的协议（可以有多个类采用一个给定的协议）。Java关于接口的概念就是借用自Objective-C协议。

第三部分　高级概念

Objective-C提供了两种不同的系统来管理对象内存：引用计数和自动垃圾收集。

- 第14章介绍Objective-C传统的引用计数系统。引用计数也叫做保留计数或管理内存。在一个使用引用计数的程序中，每个对象保持一个计数，命名为引用计数，这是使用该对象的其他对象的数目。当这个计数减少到0的时候，就会销毁该对象。这一章介绍了正确使用引用计数所需要的规则。
- 第15章介绍Objective-C的新的自动垃圾收集系统。使用垃圾收集时，有一个单独的名为垃圾收集器的线程，它负责确定哪些对象不再需要并释放它们。垃圾收集使你不必负责大多数内存管理的杂事。
- 第16章介绍Objective-C的新的块功能。块类似于一个匿名函数，但是块带有其包围环境中的变量的值。块是Apple的GCD并发机制的核心功能。

第四部分　附录

- 附录A提供对编译器来说具有特殊含义的名称的一个表，以及Objective-C编译器指令的列表。编译器指令是以一个@字符开头的单词，它们是在各种条件下传给编译器的指令。
- 附录B给出Foundation类的一个列表，其实例具有相同的内存布局，并且可以与来自低层级的C语言Core Foundation框架的对应的对象交互使用。
- 附录C简单介绍Apple向64位计算的迁移。
- 附录D。旧的“遗留”的Objective-C运行时用于32位的OS X程序，新的“现代”运行时用于在OS X 10.5 或更高版本上运行的64位Objective-C程序以及iOS程序，附录D介绍了二者之间的区别。
- 附录E列出了包含对Objective-C开发者有用信息的图书和Web站点。

编译时和运行时

当你编写程序时，有两个时刻很重要：编译时，即当你的源代码翻译成机器语言并且连接到一起形成一个可执行的程序时；还有就是运行时（也叫做执行时），即当可执行的程序作为某台计算机上的一个进程运行时。区分Objective-C和其他常用语言（特别是C++）的特征之一是，Objective-C是一种非常动态的语言。“动态”意味着其他语言在编译时所做出的决定，在Objective-C中都推迟到运行时做出。关于这一点的最突出的例子就是Objective-C的消息系统。当一个程序计算一个消息表达式时（等同于其他语言中的方法调用）所要执行的代码的部分，是在运行时确定的。

推迟决定直到运行时，这有很多的优点，你将会在阅读本书的时候看到这些优点，但是，它也有一个显著的缺点，那就是它限制了编译器所能进行的检查的数量。当使用Objective-C编写代码时，原本在其他语言中会在编译阶段捕获的一些错误，在Objective-C中都会出现在运行时。

内存管理

正如前面所提到的，Objective-C 2.0在使用手动引用计数系统和自动垃圾收集来管理对象内存方面提供了选择。除了第15章介绍的Objective-C 2.0的垃圾收集系统以外，本书的其他部分在所有的示例中一开始就在使用引用计数。第14章详细介绍了引用计数。

这么做的主要原因是垃圾收集在iOS上不可用。如果你想要为iPhone、iPod Touch或iPad编写程序，就必须学习Objective-C的引用计数系统。

从各种Objective-C和Cocoa的邮件列表中的内容来判断，引用计数可能是人们学习Objective-C的过程中引发混淆的最大源头。但是，如果你及早地学习其规则并规范地使用它们，你将会发现，实际上使用引用计数并不难。

如果在随后的某个时刻，你想要在一个项目中使用垃圾收集，转换过程更应该是相当容易的。尽管在从引用计数迁移到垃圾收集时（第15章将介绍垃圾收集），你需要注意一些架构性的问题，但使用垃圾收集大多数时候只不过是不做那些当你使用引用计数时所要做的那些事情。

关于示例

为一本介绍性的图书创建代码示例，这提出了一个挑战：如何不迷失在那些要构建一个能工作的程序所必需的模板文件的海洋中。在很多情况下，本书采取了使用某种程度上假设的示例，并且为了帮助你关注所讨论的要点而对其进行了简化。不相关的代码部分均已省略，或者用省略号(...)替代。

例如：

```
int averageScore = ...
```

上面的代码表示averageScore是一个整数变量，其值由程序的其他某个部分获得。averageScore 值的来源对于该示例来说是不相关的，你只需要认为它有一个值就行了。

关于程序清单

本书中的示例采用未编号的程序清单和编号的程序清单的混合编排。

• 未编号的程序清单

主要是正文中引用的较短的代码片段，示例的前后均为正文。

• 编号的程序清单

编号的程序清单有标题，并且按照它们在一章中出现的顺序来编号（例如程序清单4.1和程序清单8.3）。这些主要是较大的示例，在该章正文后面或者在该章后面的练习中会引用到它们。

在这两种情况下，那些需要一行一行说明的示例，都给出了一个行编号，以便说明性的正文能够引用代码中特定的行。

关于练习

本书中的大多数章最后都有一组练习。当然鼓励你完成它们。很多练习要求你编写较小的

程序来验证在那一章正文中介绍的知识点。一些练习可能看上去很冗长，但是编写代码和查看结果都会提供比仅仅阅读代码更加生动的学习体验。编写较小的程序来测试你的理解，这是应该养成的宝贵习惯；对于不清楚的每个知识点，你都应该编写一个这样的程序，即便本书没有提供一个相关的练习。

练习没有建议哪个程序需要一个用户界面。所有这些都可以通过使用一个文本编辑器来编写，并且从命令行来编译和运行它们，如第2章以前的练习所示，或者使用一个简单的Xcode项目来编辑、编译和运行它们，如第4章所介绍。

致　谢

写过书的人都知道，即便是一个作者写的一本书，那也是团队努力的结果。本书也不例外，Scott D. Yelich、Andy Lee、Matt Long、Cory Bohon和Joachim Bean评阅了初稿。他们不仅发现了错误，而且还促使我更加仔细地思考一些最初我忽视了的问题。Steve Peter激发我开始编写本书，Daniel Steinberg对于初稿的形成给予了帮助。我要感谢Addison-Wesley的Romny French和我的编辑Chuck Toporek。我低估了写作各个章节所花的时间，并且初次使用Microsoft Word时备受挫折，而Chuck容忍了这一切。

当事情进展得不好的时候，每个人都需要富有同情心的倾听。在我编写本书的时候，以及准备开始写这本书之前的几十年里，我的朋友Pat O' Brien、Michael Sokoloff和Bill Schwartz每个人都乐于扮演这种倾听的角色。

有两个人值得特别提及：

Joseph E. Sacco博士阅读了本书的几个版本的草稿，测试了其中的练习，并且给予了众所周知的“很多有价值的技术讨论”，还有很多有价值的非技术讨论。

Ekko Jennings阅读了一些章节，并给予了精神上的支持和帮助，即便在本该由我做饭的时候也下厨烹饪，并且总是忍受我不停写作。谢谢！

作者简介

Robert Clair拥有Oberlin学院的物理学学士学位，以及加州大学伯克利分校的物理学硕士和博士学位。他在商业软件开发领域拥有20年以上的经验，主要从事CAD、建模和图形方面的工作。在过去的7年里，他主要从事Mac上的Objective-C工作，现在转移到iPhone上。他编写过ZeusDraw，这是一款用于Mac OS X的矢量绘图程序；还有ZeusDraw Mobile，这是用于iPhone的一款绘图程序。他还是众多的iPhone和iPad应用程序的顾问。他居住在纽约市，是Chromatic Bytes公司的负责人，这是一家独立的软件公司。

目　录

第二部分 语言基础

第三部分 高级概念

第四部分 附录

第一部分
Objective-C简介

本书第一部分是Objective-C的简介。Objective-C是C语言的扩展，因此，本部分的前两章对C进行了回顾。回顾了C之后，介绍了面向对象编程的概念，以及这些概念在Objective-C中是如何实现的。第4章将带你一行一行地分析一个简单的Objective-C程序。

第1章 C，Objective-C的基础

Objective-C是C的一个扩展。本书大部分的内容都关注Objective-C对C添加了什么。但是，要使用Objective-C编程，必须知道C的基础知识。在Objective-C中进行两数相加、在代码中加入一条注释，或者使用一条if语句等之类的常规操作时，其方式都是与C相同的。Objective-C非对象的部分不是与C相似，或者与C类似，它简直就是C。目前的Objective-C 2.0基于C的C99标准。

本章是回顾C的两章中的第1章。注意，回顾并不是对C的完整介绍，而只是介绍了该语言的基本部分。像位运算符、类型转换的细节、Unicode字符、带有参数的宏，以及其他神奇之处并没有提及。回顾部分专门用来帮助那些曾经学过C语言的人们进行回忆，或者作为那些擅长从环境中捡起一门新语言的人的快速参考。第2章继续回顾C，并且介绍了声明变量、变量作用域及C将变量放置在内存中何处等话题。如果你是一位资深C/C++程序员，可以跳过这一章（然而，回顾一下也没什么坏处。在编写本章的过程中也学到了一些东西）。如果你从另一种类似C的语言（例如Java或C#）转向Objective-C，你至少应该快速浏览一下这些内容。如果你只有一种脚本语言的编程经验，或者你完全是一名初学者，那么，你会发现在阅读本书的同时阅读一本关于C语言的书会很有帮助。

注意 建议每个人都阅读本书的第2章。以我的经验，很多人应该熟悉这些内容，但实际上他们并不是很熟悉。

关于C语言的书有很多。最早的Kernighan和Ritchie所著的《The C Programming Language》一书，仍然是其中最好的一本。大多数人学习C语言都是使用的这本书。[⊖]关于C语法，或者说如果你要探究C语言的某些边边角角，那么，可以参考Harbison和Steele所著的《C: A Reference Manual》一书。[⊜]

考虑一下，你是如何开始学习一门新的自然语言的。要做的第一件事情是不是看看这门语言是如何写的：它使用哪些字母（或象形文字）？它是从左向右读、从右向左读，还是从上向下读？然后，开始学习一些单词。你至少需要从一个较小的词汇表起步。建立起自己的词汇表后，就可以开始用单词组建短语，开始把短语组合为完整的句子。最终，可以把句子组合为完整的段落。

对C语言的回顾大致按照同样的过程。1.1节看看一个C程序的结构，C代码是什么格式，以

⊖ Brian W. Kernighan和Dennis M. Ritchie著的《The C Programming Language, Second Edition》。相应的中文版《C程序设计语言（第2版）》由机械工业出版社于2004年1月出版。——编辑注

⊜ Samuel P. Harbison和Guy L. Steele著的《C: A Reference Manual, Fifth Edition》。相应的中文版《C语言参考手册（原书第5版）》由机械工业出版社于2008年4月出版。——编辑注

及命名各种实体的规则和惯例。接下来的小节，介绍变量和运算符，它们大致类似于自然语言中的名词和动词；然后，看看它们如何组合为较长的表达式和句子。随后的小节，介绍控制语句。最后一个小节介绍C预处理器，它允许我们在将源文件发送给编译器之前对其进行一些编程式的编辑，该小节还介绍了printf函数，它用于字符输出。

1.1 C程序的结构

本章首先介绍C程序结构的基本知识，例如main函数、格式化问题、注释、名称和命名惯例，以及文件类型等。

1.1.1 main函数

所有的C程序都有一个main函数。在操作系统载入一个C程序之后，程序从main函数的第一行代码开始执行。main函数的标准形式如下：

```
int main(int argc, const char * argv[])
{
  // The code that does the work goes here
  return 0;
}
```

main函数的主要特征是：

- 第一行中的int表示main向操作系统以返回代码的形式返回一个整数值。
- 名称main是必需的。
- 第一行剩下的部分是从操作系统传递给程序的命令行参数。main接受argc那么多个参数，并且作为字符串存储于argv数组中。这部分目前不是很重要，可以忽略。
- 所有的可执行代码放在一对花括号中。
- return 0;行表明了0作为返回代码传递回操作系统。在UNIX系统中（包括Mac OS X和iOS），为0的返回代码表示“没有出错”，而其他的任何值则意味着一个某种类型的错误。

如果你对于处理命令行参数或向OS返回一个错误代码不感兴趣（例如，在做后面几章的练习时），可以使用main的一种简化形式：

```
int main( void )
{

}
```

void表明main函数的这个版本不接受参数。在没有一条显式的return语句时，意味着返回值为0。

1.1.2 格式化

C语句由一个分号结束。空白字符（空格、制表符和换行符）是分隔名称和关键字所必需的。C忽略任何额外的空白：缩进和任何额外的空白对于编译的可执行代码都没有影响；可以自由地使用它们以使代码更具可读性。一条语句可以扩展到多行，例如，如下的3条语句是等价的：

```
distance = rate*time;

    distance     =     rate   *   time;

distance =
    rate *
        time;
```

1.1.3 注释

注释是程序员的启示性表示。编译器会忽略注释。C支持两种形式的注释：

- 跟在两个斜杠(//)后面，直到该行结束之前的所有内容，都是一条注释。例如：

```
// This is a comment.
```

- 在/*和*/之间的任何内容，也是一条注释：

```
/* This is the other style of comment */
```

这种类型的注释可以跨越多行。例如：

```
/* This is
      a longer
          comment. */
```

在编程的过程中，它也可以用来临时性地“注释掉”代码段。这种样式的注释不能像如下这样嵌套：

```
/* /* WRONG - won't compile */ */
```

然而，下面的形式是合法的：

```
/*
    // OK - can nest end of line comment
*/
```

1.1.4 变量和函数名

C中的变量和函数名，由字母、数字和下划线(_)字符组成：

- 第一个字符必须是一个下划线或一个字母。
- C名称是区分大小写的，例如，bandersnatch和Bandersnatch是不同的名称。
- 在一个名称的中间不能有任何空白。

下面是一些合法的名称：

```
j
taxesForYear2010
bananas_per_bunch
bananasPerBunch
```

如下的名称是不合法的：

```
2010YearTaxes
rock&roll
bananas per bunch
```

1.1.5 命名惯例

为了方便自己及必须阅读代码的任何人，应该为变量和函数使用具有描述性的名称。bpb很容易录入，但是，当你一年后返回来看时，它会让你感到疑惑；bananas_per_bunch则是一目了然的。很多普通的C程序在较长的变量和函数名中使用下划线来分隔开单词：

```
apples_per_basket
```

Objective-C程序员通常为变量使用CamelCase（骆驼命名法）名称。CamelCase名称使用首字母大写来表示名称中后续单词的开始：

```
applesPerBasket
```

以一个下划线开始的名称，通常用作私有的变量和函数，或者是供内部使用的名称：

```
_privateVariable
_leaveMeAlone
```

然而，这是一个惯例；C没有强制性的机制来保证变量或函数是私有的。

1.1.6 文件

普通C程序的代码放入到一个或多个以.c为扩展名的文件中：

```
ACProgram.c
```

注意 Mac OS X文件名不区分大小写。文件系统将会记住你用来命名一个文件的大小写，但是，它将myfile.c、MYFILE.c和MyFile.c当做相同的文件名。

使用Objective-C对象（本书从第3章开始介绍对象）的代码，放置在以.m为扩展名的一个或多个文件中：

```
AnObjectiveCProgram.m
```

注意 由于C是Objective-C的一个子集，因此将一个普通C程序放入到一个.m文件中也没有问题。

针对定义和实现Objective-C类（将在第3章介绍）的文件，还有一些命名惯例。但是，C对于扩展名之前的名称部分，没有任何正式的规则。将包含一个记账程序的代码的文件命名如下，会很傻，但是合法：

```
MyFlightToRio.m
```

C程序还使用头文件。头文件通常包含了由很多.c和.m文件共同使用的各种定义。通过使用一个#include或#import预处理器指令，可以将这些文件合并到其他文件中（参见本章稍后的1.6节）。头文件有一个.h扩展名，如下所示：

```
AHeaderFile.h
```

注意 这一话题超出了本书的讨论范围，但是，在同一个程序中混合使用Objective-C和C++代码是可能的。结果叫做Objective-C++。Objective-C++代码必须放置在一个扩展名为.mm的文件中：

```
AnObjectiveCPlusPlusProgram.mm
```

要了解更多信息，参见http://developer. apple.com/ mac/library/documentation/ Cocoa/ Conceptual/ ObjectiveC/ Articles/ ocCPlusPlus。

1.2 变量

变量是程序中用于某些内存字节的名称。当为一个变量赋一个值时，实际所做的事情是，将该值存储到这些字节中。计算机语言中的变量，就像自然语言中的名词。它们表示程序的问题空间中的项或量。

C要求通过声明变量来告诉编译器你将要使用的任何变量。变量声明的形式如下：

```
variabletype name;
```

C允许一次声明多个变量：

```
variabletype name1, name2, name3;
```

变量声明导致编译器为这些变量保留存储空间（内存）。变量的值就是其内存位置的内容。第2章将更详细地介绍变量声明。将介绍变量声明放置于何处，变量创建于内存中的何处，以及不同类型的变量的生命期。

1.2.1 整数类型

C提供了如下类型来保存整数：char、short、int、long和long long。表1-1给出了32位和64位Mac OS X可执行程序中的整数类型的字节大小（附录C将介绍32位和64位的可执行程序）。

表1-1 整数类型的大小

类 型	32位	64位
char	1字节	1字节
short	2字节	2字节
int	4字节	4字节
long	4字节	8字节
long long	8字节	8字节

char类型之所以叫做char，是因为它最初是用来保存字符的；但是，它经常用作一个8位的整数类型。

整数类型可以声明为unsigned的：

```
unsigned char a;
unsigned short b;
unsigned int c;
unsigned long d;
unsigned long long e;
```

当unsigned单独使用时，表示unsigned int：

```
unsigned a; // a is an unsigned int
```

无符号变量的位模式总是解释为一个正的数字。如果你给一个无符号变量赋一个负的值，结果是很大的正数。这几乎总是一个错误。

1.2.2 浮点类型

C的浮点类型有：float、double和long double。浮点类型的字节大小在32位和64位可执行文件中都是相同的：

```
float aFloat;   // floats are 4 bytes
double aDouble; // doubles are 8 bytes
long double aLongDouble;  // long doubles are 16 bytes
```

浮点值总是有符号的。

1.2.3 真值

普通的表达式经常用到真值。计算为零值的表达式认为是假，而计算为非零值的表达式认为是真。

_Bool、bool和BOOL

C的早期版本没有定义布尔类型。普通表达式使用（并且现在仍然使用）布尔值（真值）。正如上文所述，计算为零值的表达式认为是假，而计算为非零值的表达式认为是真。大多数的C代码仍然这样编写。

当前的C标准C99，引入了一个_Bool类型。_Bool是一个整数类型，它只有两个允许的值，即0和1。赋给_Bool任何非零值，将得到1：

```
_Bool b = 35;   // b is now 1
```

如果在源代码文件中包含了文件stdbool.h，可以使用bool作为_Bool的一个别名，而且可以使用布尔常量true和false（true和false只不过分别定义为1和0）。

```
#include <stdbool.h>
bool b = true;
```

我们很少会在Objective-C代码中看到_Bool或bool，这是因为Objective-C定义了自己的Boolean类型BOOL。第3章将介绍BOOL。

1.2.4 初始化

变量可以在声明的时候初始化：

```
int a = 9;

int b = 2*4;

float c = 3.14159;

char d = 'a';
```

包含在单引号中的单个字符是一个字符常量。它在数字上等于该字符的编码值。这里，变量d的数字值为97，它是字符a的ASCII值。

1.2.5 指针

指针是其值等于一个内存地址的一个变量。它“指向”内存中的一个位置。可以通过在声明中，在变量名前面放置一个*，将一个变量声明为一个指针变量。如下的代码把pointerVar声明为指向内存中保存一个整数的位置的一个变量：

```
int *pointerVar;
```

一元运算符&（“取址”运算符）用来获取一个变量的地址，以便将其存储到一个指针变量中。如下的代码将指针变量b的值设置为整数变量a的地址：

```
1 int a = 9;
2
3 int *b;
4
5 b = &a;
```

现在，我们一行一行地解释这个例子：

- 第1行将a声明为一个int变量。编译器拿出4个字节来存储a，并且将其初始化为值9。
- 第3行将b声明为一个指向int的指针。
- 第5行使用&运算符来获取a的地址，然后，将a的地址赋值给b。

图1-1说明了这个过程（假设编译器分配的起始地址为1048880）。图中的箭头表示了指向的概念。

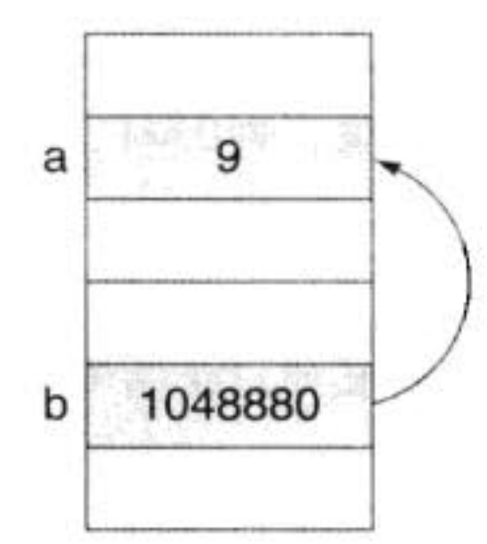


图1-1 指针变量

一元运算符*（叫做“求值”或“解引用”运算符），通过使用指向一个内存位置的指针变量，来设置或获取该内存位置的内容。可以这样来看，把表达式*pointerVar看做是一个别名，即针对存储在pointerVar的内容中的任何内存位置的另一个名称。表达式*pointerVar可以用来设置或获取该内存位置的内容。

在如下的代码中，将b设置为a的地址，因此，*b变成了a的别名：

```
int a;
int c;
int *b;
a = 9;
b = &a;
c = *b;  // c is now 9
*b = 10; // a is now 10
```

指针在C中用来引用动态分配的内存（参见第2章）。指针也用来避免将大块的内存，例如数组和结构（将在本章稍后介绍），从一个程序的一部分复制到另一部分。例如，可以把指向结构的一个指针传递给函数，而不是把一个较大的结构传递给函数。函数随后使用指针来访问结构。随后我们还会看到，Objective-C对象也总是通过指针来引用。

通用指针

声明为void的指针变量，是一个通用指针。

```
void *genericPointer;
```

可以将通用指针设置为任何变量类型的地址：

```
int a = 9;
void *genericPointer;
genericPointer = &a;
```

然而，试图从一个通用指针获取一个值，将导致一个错误发生，因为编译器没办法知道如何解释通用指针所表示的地址的字节：

```
int a = 9;
int b;
void *genericPointer;
genericPointer = &a;
b = *genericPointer;  // WRONG - won't compile
```

要通过void*指针获取一个值，必须将其转换为指向一个已知类型的指针：

```
int a = 9;
int b;
void *genericPointer;
genericPointer = &a;
b = *((int*) genericPointer) ;  // OK - b is now 9
```

强制转换运算符(int*)迫使编译器将genericPointer看做是指向一个整数的指针（参见本章后面的1.3.10节）。

C不会检查一个指针变量是否指向内存的一个有效区域。在C编程中错误地使用指针，可能会比其他错误原因引发更多的程序崩溃。

1.2.6 数组

C数组的声明，首先是数组的类型和数组名，然后是一个方括号，方括号中是数组元素的数目：

```
int a[100];
```

通过将元素的索引放到数组名称后面的[ ]中，可以访问数组的单个元素：

```
a[6] = 9;
```

索引是基于0的。在前面的例子中，合法的索引在0～99。访问C数组时系统不会在两端进行边界检查。C允许做如下的事情：

```
int a[100];
a[200] = 25;
a[-100] = 30;
```

使用超出数组边界的一个索引，将会导致访问属于其他变量的无用内存，这要么导致程序崩溃，要么导致数据毁坏。利用缺乏检查的缺陷，是恶意软件的惯用招数之一。

方括号表示法只是指针算术的一种不错的语法。不带有数组方括号的数组名称，是一个指向数组开始处的指针变量。如下两行代码是完全等价的：

```
a[6] = 9;
*(a + 6) = 9;
```

当要编译的表达式使用指针算术时，编译器会考虑指针所指向的类型的大小。如果a是一个int数组，表达式*(a+2)指的是距离数组a开始处8个字节（两个int）地址处的一个4字节内存（这是一个int类型的大小）内容。然而，如果a是一个char数组，表达式*(a+2)指的是距离数组a开始处两个字节（两个char）地址处的一个1字节内存（这是一个char类型的大小）内容。

多维数组

多维数组声明如下：

```
int b[4][10];
```

多维数组按照行顺序地存储。这里，b[0][0]是第一个元素，b[0][1]是第二个元素，b[1][0]是第11个元素。

使用指针表示法为：

```
b[i][j]
```

可以写作：

```
*(b + i*10 + j)
```

1.2.7 字符串

C字符串是字节（char类型）的一维数组，以一个0字节终结。C中的常量字符串，通过把字符串的字符放置在双引号("")之间来编写：

```
"A constant string"
```

当编译器在内存中创建一个常量字符串时，它自动在末尾添加0字节。但是，如果声明了一个用来保存字符串的char数组，在确定需要多少空间时，则必须记住包含0字节。如下的代码行将常量字符串“Hello”的5个字符及其最终的0字节复制到数组aString中：

```
char aString[6] = "Hello";
```

与任何其他的数组一样，表示字符串的数组也不进行边界检查。溢出程序输入所使用的字符串缓冲区，是黑客惯用的招数。

char*类型的变量，可以用一个常量字符串来初始化。可以将这样的一个变量设置为指向一个不同的字符串，但是，不能够使用它来修改一个常量字符串：

```
char *aString = "Hello";

aString = "World";

aString[4] = 'q';  // WRONG - causes a crash
```

第1行把aString指向常量字符串“Hello”。第2行将aString修改为指向常量字符串“World”。第3行引发程序崩溃，因为将常量字符串存储到了一段受保护的、只读的内存中。

1.2.8 结构

结构将相关变量组成了一个集合，以便能够将这些变量作为一个单独的实体引用。如下是一个结构声明的例子：

```
struct dailyTemperatures
  {
    float high;
    float low;
    int   year;
    int   dayOfYear;
};
```

结构中的单个变量叫做成员变量，或者简单地称为变量。跟在关键字struct后面的名称，是结构的标签。结构标签标识该结构。它可以用来声明类型为该结构的变量：

```
struct dailyTemperatures today;

struct dailyTemperatures *todayPtr;
```

在前面的例子中，today是一个dailyTemperatures结构，而todayPtr是一个指向dailyTemperatures结构的指针。

点运算符(.)用来通过一个结构变量访问结构中的单个成员。指针运算符(->)用来从一个变量访问结构成员，而该变量是指向结构的一个指针：

```
todayPtr = &today;

today.high = 68.0;

todayPtr->high = 68.0;
```

后两条语句完成同样的事情。

结构可以用其他的结构作为成员。前面的例子可以写成如下所示的形式：

```
struct hiLow
{
    float high;
    float low;
};

struct dailyTemperatures
{
    struct hiLow tempExtremes;
    int    year;
    int    dayOfYear;
};
```

那么，设置今天的最高温度可以用如下形式表示：

```
struct dailyTemperatures today;
today.tempExtremes.high = 68.0;
```

注意 编译器会很容易地在一个结构中插入补充，从而迫使结构成员按照内存中的一个特定边界来对齐。不要通过计算结构成员从结构开始处的偏移量来访问它们，或者做依赖于结构的二进制布局的任何其他事情。

1.2.9 typedef

typedef声明提供了一种方法来为变量类型创建别名：

```
typedef float Temperature;
```

现在，Temperature可以用来声明变量，就好像它是一个内建的类型一样：

```
Temperature high, low;
```

typedef只是为变量类型提供了一个替代名称。这里，high和low都仍然是浮点数。关键字typedef在C代码中往往用作一个动词，就像在“Temperature is typedef'd to float”中一样。

1.2.10 枚举常量

enum语句允许定义一组整数常量：

```
enum woodwind { oboe, flute, clarinet, bassoon };
```

该语句的结果是：oboe、flute、clarinet和bassoon，常量值分别为0、1、2和3。

如果不想按照从0开始的顺序，可以自己给常量赋值。任何没有被指定值的常量，其值都比前一个常量大1，如：

```
enum woodwind { oboe=100, flute=150, clarinet, bassoon=200 };
```

上面的语句将oboe、flute、clarinet和bassoon的值分别指定为100、150、151和200。

关键字enum后面的名称叫做枚举标签。枚举标签是可选的。枚举标签也可以用来声明变量：

```
enum woodwind soloist;
soloist = oboe;
```

枚举对于定义多个常量很有用，并且有助于增强代码的可读性，但是，它们不是明确的类型，并且没有得到编译器太多的支持。声明enum woodwind soloist;表明你希望soloist应该限制为oboe、flute、clarinet或bassoon之一，但是，遗憾的是，编译器没有做任何事情来强化这一限制。编译器认为soloist是一个int，并且，它允许将任何整数值赋给soloist，而不会产生一条警告：

```
enum woodwind { oboe, flute, clarinet, bassoon };
enum woodwind soloist;
soloist = 5280;  // No complaint from the compiler!
```

注意 枚举常量占据与变量名称相同的名称空间。所以不能让一个变量和枚举常量具有相同的名称。

1.3 运算符

运算符就像是动词。它们引发对变量的运算。

1.3.1 算术运算符

C中有+、−、*和/这些常用的二元运算符，它们分别用于加、减、乘和除运算。

注意 如果除法运算符(/)的两个操作数都是整数类型，则C执行整数除法。整数除法会把除法的结果进行截取。例如，7 / 3的值是2。

1.3.2 余数运算符

余数运算符或模运算符(%)计算一个整数除法的余数。例如，下面的表达式的结果是1：

```
int a = 7;
int b = 3;
int c = a%b;  // c is now 1
```

余数运算符的操作数必须都是整数类型。

1.3.3 自增和自减运算符

C为自增和自减变量提供了运算符：

```
a++;

++a;
```

两行都给a的值增加1。然而，当将两个表达式用作一个更大的表达式的一部分时，它们之间就有区别了。前缀版++a，会在任何计算发生之前增加a的值。在表达式中使用的是增加以后的a值。而后缀版a++则是在其他计算进行之后才增加其值。在表达式中，使用的是其最初的值。这可以通过如下的示例来说明：

```
int a = 9;
int b;
b = a++; // postfix increment

int c = 9;
int d;
d = ++c; // prefix increment
```

自增运算符的后缀版，是在将变量的初始值用于表达式的计算之后，才将该变量增加1。当示例中的代码执行完以后，b的值是9，而a的值是10。自增运算符的前缀版，是在该变量值用于表达式计算之前就增加它。因此在这个示例中，c和d的值都是10。

自减运算符a--和--a以类似的方式工作。

使用该运算符的前缀版和后缀版之间的差别的代码，很可能令除编写者之外的人产生混淆。

1.3.4 优先级

如下的表达式是等于18还是22：

```
2 *7 + 4
```

答案似乎是含糊的，因为这取决于是先进行加法还是先进行乘法运算。C通过指定一条规则，即在执行加法和减法之前，先执行乘法和除法，从而解决了二义性问题；因此，该表达式的值是18。从技术性上来说，就是乘法和除法拥有比加法和减法更高的优先级。

如果需要先进行加法运算，可以使用圆括号指定：

```
2 * (7 + 4)
```

编译器将会尊重你的请求，先执行加法，然后再执行乘法。

注意 C为其所有的运算符定义了一个复杂的优先级表（参见http://en.wikipedia. org/wiki/Order_of_operations）。使用圆括号来指定想要的运算顺序，比努力记住运算符优先级要容易得多。

1.3.5 取反

一元减法符号(-)把一个算术值取反：

```
int a = 9;
int b;
b =  -a;  // b is now -9
```

1.3.6 比较

C提供了用于比较的运算符。比较的值是一个真值。如下的表达式，如果为真，则取值为1；如果为假，则取值为0。

```
a > b // true, if a is greater than b

a < b // true, if a is less than b

a >= b // true, if a is greater than or equal to b

a <= b // true, if a is less than or equal to b

a == b // true, if a is equal to b

a != b // true, if a is not equal to b
```

与任何语言一样，测试浮点数的相等性是有风险的，因为可能会有舍入错误，并且，这样的比较可能会得不到正确的结果。

1.3.7 逻辑运算符

AND和OR逻辑运算符的形式如下：

```
expression1 && expression2  // Logical AND operator

expression1 || expression2  // Logical OR operator
```

C使用短路计算方式。表达式从左向右计算，并且，只要整个表达式可以得出真值，计算就停止。如果一个AND表达式中的expression1取值为假，那么，整个表达式的值都为假，从而expression2就不再计算了。同样，如果一个OR表达式中的expression1取值为真，那么，整个表达式都为真，从而expression2就不再计算了。如果第二个表达式有任何的副

作用，短路计算就会得到有趣的结果。在下面的例子中，如果b大于或等于a，就不会调用CheckSomething()函数（本章后面将要介绍if语句）了：

```
if (b < a && CheckSomething())
  {
    ...
  }
```

1.3.8 逻辑取反

叹号(!)是C中的一元逻辑取反运算符。在如下的代码行执行之后，如果expression为真(非0)，则a的值为0；并且，如果expression的值为假（0），则a的值为1：

```
a = ! expression;
```

1.3.9 赋值运算符

C提供了基本的赋值运算符：

```
a = b;
```

上面的语句将b的值赋给了a。当然，a必须是能够对其赋值的变量。可以对其赋值的实体叫做左值（lvalue）（因为它们放在赋值运算符的左边）。如下是左值的一些例子：

```
 /* set up */
float a;
float b[100]
float *c;
struct dailyTemperatures today;
struct dailyTemperatures *todayPtr;
c = &a;
todayPtr = &today;

/* legal lvalues */
a = 76;
b[0] = 76;
*c = 76;
today.high = 76;
todayPtr->high = 76;
```

有些内容不是左值。不能对一个数组名称、一个函数的返回值或者任何没有引用一个内存位置的表达式赋值：

```
float a[100];
int x;

a = 76; // WRONG
x*x = 76; // WRONG
GetTodaysHigh() = 76; // WRONG
```

1.3.10 转换和强制类型转换

如果赋值的两端具有不同的变量类型，则右边的类型会转换为左边的类型。从较短的类型转换为较长的类型，或者从整数类型转换为浮点类型，这不会引发问题。但是，反过来，从一个较长的类型转换为一个较短的类型，就可能会导致有意义的数字丢失、截断或者完全无意义。例如：

```
int a = 14;
float b;
b = a;  // OK, b is now 14.0

float c = 12.5;
int d;
d = c;  // Truncation, d is now 12

char e = 128;
int f;
f = e;  // OK, f is now 128

int g = 333;
char h;
h = g;  // Nonsense, h is now 77
```

可以使用强制类型转换，来迫使编译器把一个变量的值转换为不同的类型。在下面的示例中的最后一行，(float)强制转换强迫编译器把a和b转换为浮点数，并且进行浮点数除法运算：

```
int a = 6;
int b = 4;
float c, d;

c = a / b;  // c is equal to 1.0 because integer division truncates

d = (float)a / (float)b; // Floating-point division, d is equal to 1.5
```

可以强制转换指针，从而将一种类型的指针转换为另一种类型的指针。强制转换指针可能是有风险的操作，因为这可能会毁坏内存，但是，对于以void*类型传递给你的一个指针来说，这是将其解引用的唯一方式。成功地强制转换一个指针，需要理解指针“实际”所指向的实体的类型。

1.3.11 其他赋值运算符

C还有其他的快捷运算符，它们把计算和赋值组合了起来：

```
a += b;

a -= b;

a *= b;

a /= b;
```

其等价的形式分别如下：

```
a = a + b;

a = a - b;

a = a * b;

a = a / b;
```

1.4 表达式和语句

C中的表达式和语句相当于自然语言中的短语和句子。

1.4.1 表达式

最简单的表达式只是单个的常量或变量：

```
14
bananasPerBunch
```

每个表达式都有一个值。常量表达式的值，就是常量自身，例如，14的值就是14。变量表达式的值就是变量所保存的内容，例如，bananasPerBunch的值，就是在初始化或赋值的时候为其最终设置的值。

表达式可以组合以形成其他的表达式。如下的形式也是表达式：

```
j + 14
a < b
distance = rate * time
```

算术或逻辑表达式的值，就是通过执行算术或逻辑运算所得的结果。赋值表达式的值，就是作为赋值运算的目标的变量所得到的值。

函数调用也是表达式：

```
SomeFunction()
```

函数调用表达式的值就是函数的返回值。

1.4.2 计算表达式

当编译器遇到一个表达式时，它创建二进制代码来执行表达式并得到其值。对于原型表达式，没有什么事情可做，其值就是它们自身。对于较为复杂的表达式，编译器会生成执行特定算术计算、逻辑运算、函数调用和赋值的二进制代码。

计算表达式可能引起副作用。最常见的副作用是，由于赋值而修改了一个变量的值，或者由于函数调用而执行了函数中的代码。

在各种控制结构中，表达式的值用来决定一个程序的流程（参见后面的1.5节）。在其他情况下，可能计算表达式，只是为了得到计算它们的副作用。通常，一个赋值表达式的位置，就是赋值所发生的地方。在极少数情况下，值和副作用二者都很重要。

1.4.3 语句

当在表达式的末尾添加一个分号(;)时，它就变成了一条语句。这类似于在自然语言中，给一个短语添加一个句点来得到一个句子。代码中的一条语句等同于一个完整的想法。当通过编译一条语句而得到的所有机器语言指令都执行完毕，并且，该语句所影响到的所有内存位置的修改也都已经完成时，该语句的执行也就完成了。

1.4.4 复合语句

在能够使用单条语句的任何地方，都可以使用一系列的语句，不过要用一对花括号将其括起来：

```
{
  timeDelta = time2 - time1;
  distanceDelta = distance2 - distance1;
  averageSpeed = distanceDelta / timeDelta;
}
```

在结束花括号的后面没有分号。像这样的一组语句，叫做复合语句或语句块。复合语句经常与控制语句一起使用。

注意 使用“块（block）”这个词来作为复合语句的同义词，在C的描述中很常见，这可以追溯到C语言的创始之初。遗憾的是，Apple已经采用“块”来表示其对C添加的闭包（参见第16章）。为了避免混淆，本书后面的部分使用更加复杂一点的名称，即复合语句。

1.5 程序流程

程序中的语句是顺序执行的，除非由一个for、while、do-while、if、switch或goto语句或一个函数调用将流程导向到其他地方去做其他的事情。

- 一条if语句根据一个表达式的真值来有条件地执行代码。
- for、while和do-while语句用于构建循环。在循环中，重复地执行相同的语句或一组语句，直到满足一个条件为止。
- switch语句根据一个整数表达式的算术值，来选择一组语句执行。
- goto语句无条件地跳转到一条标记的语句。
- 函数调用跳入到函数体中的代码。当该函数返回时，程序从函数调用之后的位置开始执行。

这些控制语句都将在后面的小节中详细介绍。

注意 在阅读后面的内容时，记住，凡是提到语句的每个地方，我们都可以使用复合语句。

1.5.1 if

if语句根据一个表达式的真值来有条件地执行代码。其形式如下：

```
if ( expression )

  statement
```

如果expression计算为真（非零），将执行statement；否则，从if语句之后的下一条语句开始继续执行。可以通过添加else部分来扩展一条if语句：

```
if ( expression )

  statement1

else

  statement2
```

if表达式为真（非零），将执行statement1；否则，执行statement2。

也可以通过添加else if部分来扩展if语句，如下所示：

```
if ( expression1 )

  statement1

else if ( expression2 )

  statement2

else if ( expression3 )

  statement3

...

else

  statementN
```

这个表达式按照顺序执行。当表达式的值非零时，对应的语句会执行，并且从if语句之后的下一条语句开始继续执行。如果表达式都为假，则else子句后面的语句会执行（就像一条简单的if语句一样，else子句是可选的，并且可以省略）。

1.5.2 条件表达式

条件表达式由3个子表达式组成，其形式如下：

```
expression1 ? expression2 : expression3
```

当计算条件表达式时，先计算expression1的值。如果它为真，将会执行expression2，并且整个表达式的值就是expression2的值。expression3不会执行。

如果expression1为假，则执行expression3并且条件表达式的值就是expression3的值。expression2不会执行。

条件表达式往往用作一条简单的if语句的一种缩写形式。例如：

```
a = ( b > 0 ) ? c : d;
```

等价于：

```
if ( b > 0 )

  a = c;

else

  a = d;
```

1.5.3 while

while语句用来构成循环，如下所示：

```
while ( expression ) statement
```

当while语句执行时，计算expression的值，如果为真，则执行statement并且再次计算条件。重复这一过程，直到表达式的值为假。此时，从while后面的下一条语句开始继续执行。偶尔会见到这种结构：

```
while ( 1 )
  {
    ...
  }
```

从while的角度来看，这是一个无限循环。假设循环体内的内容是检查一个条件，并且当该条件满足的时候，跳出循环。

1.5.4 do-while

do-while语句类似于while，区别在于，它的测试位于statement之后，而不是在其之前：

```
do statement while ( expression );
```

其结果是，不管expression的值是什么，statement总是会执行一次。即便条件为假，循环体也至少要执行一次，然而，存在这种程序逻辑的情况并不常见。因此，do-while语句在实际中很少用到。

1.5.5 for

for语句是最常用的循环结构。其形式如下：

```
for (expression1; expression2; expression3) statement
```

当执行一条for语句时，会按照如下顺序进行：

1）在循环开始前，计算一次expression1。

2）计算expression2的值。

3）如果expression2为真，则执行statement；否则，循环结束，继续从循环后的下一条语句开始执行。

4）计算expression3。

5）重复步骤2、3、4直到expression2变为假。

计算expression1和expression3仅仅是为了其副作用。它们的值会丢弃。它们通常用来初始化和自增一个循环计数器变量：

```
int j;

for ( j=0; j < 10; j++ )
  {
    // Something that needs doing 10 times
  }
```

在for结构中，任何表达式都可以省略（分号必须保留）。如果省略了expression2，则循环就是一个无限循环，类似于while(1)：

```
for ( i=0; ; i++ )
  {
    ...
    // Check something and exit if the condition is met
  }
```

注意 当使用循环来遍历一个数组的元素时，记住，数组的索引是从0开始的，直到数组中元素的数目减1为止：

```
int j;
int a[25];

for (j=0; j < 25; j++ )
  {
    // Do something with a[j]
  }
```

在前面的示例中，常见的错误是，把for循环编写为for (j=1; j <= 25; j++)。

1.5.6 break

break语句用来跳出一个循环或一条switch语句。

```
int j;
for (j=0;  j < 100; j++ )
 {
    ...

    if ( someConditionMet ) break; //Execution continues after the loop
 }
```

从while、do、for或switch语句末尾后面的下一条语句开始继续执行。当有嵌套循环

时，break只是从最内层的循环跳出。编写一条break语句，而没有一个循环或switch结构包围它，这将会导致一个编译器错误：

```
error: break statement not within loop or switch
```

1.5.7 continue

continue用于while、do或for循环的内部，用来取消当前循环迭代的执行。例如：

```
int j;
for (j=0;  j < 100; j++ )
 {
    ...

    if ( doneWithIteration ) continue; // Skip to the next iteration
    ...
 }
```

当执行continue语句时，控制传递给循环的下一次迭代。在while或do循环中，控制表达式针对下一次迭代而计算。在for循环中，计算迭代表达式（即第三个表达式），然后，计算控制表达式（即第二个表达式）。编写一条continue语句，而没有一个循环包围它，这将会导致一个编译器错误。

1.5.8 逗号表达式

逗号表达式由逗号隔开的两个或多个表达式组成：

```
expression1, expression2, ..., expressionN
```

该表达式按照从左到右的顺序计算，并且，整个表达式的值就是最右侧的子表达式的值。

逗号运算符的首要用法就是，在一个for循环中初始化和更新多个循环变量。在如下示例的循环迭代中，j从0到MAX-1，并且k从MAX-1到0：

```
for ( j=0, k=MAX-1; j < MAX; j++, k--)
  {
    // Do something
  }
```

在一个for循环中使用逗号表达式时，只有计算子表达式的副作用（在前面的例子中，就是初始化并自增或自减j和k）是重要的。逗号表达式的值会被丢弃。

1.5.9 switch

switch根据一个整数表达式的值来分支到不同的语句。switch语句的形式如下所示：

```
switch ( integer_expression )
  {
     case value1:
       statement
       break;
```

```
    case value2:
      statement
      break;
    ...

    default:
      statement
      break;
}
```

这里与C的其他部分稍微有点不一致，每个case可以有多条语句，而不需要一条复合语句。

value1， value2， ...必须是整数、字符常量或者计算为一个整数的常量表达式（换句话说，它们在编译时必须得到一个整数）。不允许具有相同的整数值的重复的case。

当执行一条switch语句时，计算integer_expression，并且，switch将结果与整数case标签相比较。如果找到一个匹配，执行将跳到匹配的case标签后面的语句。执行顺序进行，直到遇到一条break语句或到达了switch的末尾。break语句会导致执行跳出到switch之后的第一条语句。

case后面并不一定必须有一条break语句。如果省略了break，则执行将跳入到后续的case。如果你看到已有的代码中省略了break，这可能是一个错误（这是很容易犯的错误），也可能是有意的（如果程序员想要一个case及其后续的case都执行相同的代码会这样做）。

integer_expression没有和任何case标签匹配，如果有该标签的话，执行将跳到可选的default:标签后面的语句。如果没有匹配也没有default:，那么switch什么也不做，它将从switch后面的第一条语句开始继续执行。

1.5.10 goto

C提供了一条goto语句：

```
goto label;
```

执行goto语句时，控制将会无条件地跳转到label所标记的语句：

```
label: statement
```

- 标签不是可执行的语句，它们只是标记出代码中的一个位置。
- 命名标签的规则与命名变量和函数的规则相同。
- 标签总是以一个冒号结束。

滥用goto语句，可能会导致杂乱的、令人混淆的代码（通常称之为面条式代码）。通常的标准建议是，尽量不要使用goto语句。尽管如此，goto语句在某些情况下还是有用的，例如，跳出嵌套的循环（break语句只能够跳出最内层的循环）：

```
for ( i=0; i < MAX_I; i++ )
  for ( j=0; j < MAX_J; j++ )
     {
        ...
       if ( finished ) goto moreStuff;
```

```
    }

moreStuff: statement      // more statements
```

注意 是否使用goto语句，这是计算机科学领域持续时间最长的争论之一。详情参见http://david.tribble.com/text/goto.html。

1.5.11 函数

函数通常的形式如下：

```
returnType functionName( arg1Type arg1, ..., argNType argN )
{
 statements
}
```

一个简单函数示例如下：

```
float salesTax( float purchasePrice, float taxRate )
{
  float tax = purchasePrice * taxRate;
  return tax;
}
```

可以这样来调用一个函数：先写函数名，后面跟着一个括号括起来的表达式列表，其中每个表达式对应函数的一个参数。每个表达式类型必须与声明中的函数参数的类型一一对应。如下的示例展示了一个简单的函数调用：

```
float carPrice = 20000.00;
float stateTaxRate = 0.05;

float carSalesTax = salesTax( carPrice, stateTaxRate );
```

当执行函数调用的那行代码时，控制跳转到函数体中的第一条语句。执行继续进行，直到遇到一条return语句，或者到达函数的末尾。执行随后返回到调用环境中。在调用环境中，函数表达式的值就是return语句所设置的值。

注意 函数并非必须有参数或返回一个值。没有返回值的函数，其类型为void：

```
void FunctionThatReturnsNothing( int arg1 )
```

在不返回值的函数中，可以省略return语句。通过对参数列表使用空的圆括号，来表示不接受任何参数的函数：

```
int FunctionWithNoArguments()
```

函数有时候仅仅为了其副作用而执行。下面这个函数打印出了销售税，但是，它对程序的状态没有做出任何修改：

```
void printSalesTax ( float purchasePrice, float taxRate )
{
  float tax = purchasePrice * taxRate;
```

```
  printf( "The sales tax is: %f.2\n", tax );

}
```

C函数根据值来调用。当调用一个函数时，计算调用语句的参数列表中的表达式，并且将它们的值传递给函数。函数不能直接修改调用环境中的任何变量的值。下面这个函数对于调用环境中的任何内容没有影响：

```
void salesTax( float purchasePrice, float taxRate, float carSalesTax)
{
  // Changes the local variable calculateTax but not the value of
  // the variable in the calling context

     carSalesTax = purchasePrice * taxRate;
     return;
}
```

要改变调用环境中的变量的值，必须传入一个指向该变量的指针，并且，使用指针来操作该变量的值：

```
void salesTax( float purchasePrice, float taxRate, float *carSalesTax)
{
  *carSalesTax = purchasePrice * taxRate; // this will work

  return;
}
```

注意 前面的例子仍然是按照值来调用的。将调用环境中的一个变量的指针的值，传递给了该函数。然后，该函数使用这个指针（它没有修改）来设置它所指向的变量的值。

1.5.12 声明函数

当调用一个函数时，编译器需要知道函数的参数和返回值的类型。它使用这一信息来建立函数与其调用者之间的通信。如果函数的代码出现在函数调用之前（在源代码文件中），那么你不必做任何其他的事情。如果函数代码在函数调用的后面，或者位于另一个文件中，则在使用函数之前必须先声明它。

函数声明重复了函数的第一行，而且在末尾添加了一个分号：

```
void printSalesTax ( float purchasePrice, float taxRate );
```

常用的做法是将函数声明放入到一个头文件中。然后，将头文件包含到（参见后面的小节）使用该函数的任何文件中。

注意 忘记声明函数，可能会导致很隐蔽的错误。如果调用一个函数，而这个函数的代码在另一个文件中（或者在同一个文件中，但是位于函数调用之后），并且，你没有声明该函数，那么编译器和连接器都不会给出提示。但是，该函数将会针对任何浮点参数接收垃圾信息，并且，如果函数的返回类型也是浮点数的话，那么它会返回垃圾信息。

1.6 预处理器

当编译C（和Objective-C）代码文件时，在将它们发送给相应的编译器之前，首先将它们发送给一个初始化程序，叫做预处理器。以一个#字符开始的行，是给预处理器的指令。使用预处理器指令，我们可以：

- 将一个文件的文本导入到指定位置的一个或多个文件中。
- 创建定义的常量。
- 有条件地编译代码（根据条件编译或忽略语句块）。

1.6.1 包含文件

如下的一行代码：

```
#include "HeaderFile.h"
```

导致预编译器把HeaderFile.h文件的文本插入到要编译的文件中#include行所在的位置。其效果就等同于，使用一个文本编辑器把HeaderFile.h的文本复制并粘贴到要编译的文件中。

如果被包含的文件放在双引号中(""):

```
#include "HeaderFile.h"
```

那么预处理器将首先在要编译的文件所在的同一目录下查找HeaderFile.h，然后在可以作为参数提交给编译器的位置列表中查找，最后在一系列的系统位置中查找。

如果包含的文件放在尖括号中(<>):

```
#include <HeaderFile.h>
```

预处理器将只在标准系统位置中查找包含的文件。

注意 在Objective-C中，#include被#import所取代了，后者也产生同样的结果，只不过它不允许指定的文件导入多次。如果预处理器遇到了同一个头文件的多条#import指令，这些指令将会被忽略。

1.6.2 #define

#define用于环境替换。#define最常见的用法是定义常量，如下所示：

```
#define MAX_VOLUME 11
```

预处理器会把将要编译的文件中的每个MAX_VOLUME都替换为11。通过在定义中除最后一行以外的所有行的末尾放置一个反斜杠(\)，一个#define可以延续到多行。

注意 如果你这么做，那么\必须是该行的最后一项。如果在\的后面再跟着某些内容的话（例如，以“//”开始的一条注释），将会导致一个错误。

经常采用的一种做法是，把#define放置到一个头文件中，然后，将其包含到各种源文件中。这样，就可以通过在头文件中更改单个的值，从而改变该常量在所有源文件中的值。传统

的C对于定义常量所使用的命名惯例是，使用全部大写字母。传统的Apple命名惯例是以一个k开始常量名称，剩下的名称使用CamelCase法命名：

```
#define kMaximumVolume 11
```

两种形式你都会遇到，有时候它们会出现在同一段代码中。

1.6.3 条件编译

预处理器允许条件编译：

```
#if condition
  statements
#else
  otherStatements
#endif
```

这里，condition必须是在编译时可以计算为真值的一个常量表达式。如果condition计算为真（非零），将会编译statements，而不会编译otherStatements。如果condition为假，将会略过statements，而编译otherStatements。

#endif是必需的，但是#else和替代的代码是可选的。条件编译语句块必须以一个#ifdef指令开始：

```
#ifdef name
  statements
#endif
```

这个示例的行为和前面的示例相似，只不过#ifdef的真值由name是否已经被#define过来决定。

#if的用途之一是，在调试的过程中，可以很容易地删除或替换代码块：

```
#if 1
  statements
#endif
```

通过把1更改为0，可以临时忽略statements而进行一项测试。随后，再通过把0更改为1来取代该测试。

#if和#ifdef指令可以嵌套，如下所示：

```
#if 0
#if 1
statements
#endif
#endif
```

在前面的例子中，编译器忽略了位于#if 0及其对应的#endif之间的所有代码，包括其他的编译器指令。statements不会被编译。

如果需要关闭或重新打开多个语句块，可以像下面这样编写每个语句块：

```
#if _DEBUG

  statements

#endif
```

可以将定义的常量_DEBUG使用编译器的_D标志添加到一个头文件中，或从头文件中删除。

1.7 printf

输入和输出(I/O)不是C语言的一部分。字符和二进制I/O都是通过C标准I/O库中的函数来处理的。

注意 标准I/O库是与每个C环境一起提供的一组函数库。

要使用标准I/O库中的函数，必须在自己的程序中包含该库的头文件：

```
#include <stdio.h>
```

这里唯一介绍的函数是printf；它把字符串打印到终端窗口（或者，如果你使用Xcode的话，是打印到Xcode控制台窗口）。printf函数接受可变数目的参数。printf的第一个参数是一个格式字符串。其他的任何参数，都是以格式字符串所指定的方式打印出来的内容：

```
printf( formatString, argument1, argument2, ... argumentN );
```

格式字符串包含了普通字符和转换修饰符：

- 控制字符串中的普通字符（没有%）不经修改就可以发送给输出。
- 转换修饰符以一个百分号（%）开始。跟在%后面的字母表示了修饰符所期待的参数的类型。
- 每个转换修饰符按照顺序作用于格式字符串后面的参数之一。参数转换为表示参数的值的字符，并且，这些字符发送给输出。

本书使用的转换修饰符只有用于char和int的%d，用于float和double的%f，用于C字符串的%s。C字符串表示为char*。

下面是一个简单的例子：

```
int myInt = 9;
float myFloat = 3.145926;
char* myString = "a C string";

printf( "This is an Integer: %d, a float: %f, and a string: %s.\n",
    myInt, myFloat, myString );
```

注意 \n是一个换行字符。它放在输出内容之前，以便任何后续的输出都出现在下一行。

前面的示例结果是：

```
This is an Integer: 9, a float: 3.145926, and a string: a C string.
```

如果跟在格式字符串后面的参数的数目，与转换修饰符的数目不一致，那么printf会忽略掉多余的参数或者针对多余的修饰符打印出垃圾信息。

注意 本书只是使用printf来记录和调试非对象变量，而不会将其用于一个正规程序的输出，因此，本节只是粗略地介绍一下格式字符串和转换修饰符。

printf处理大量的类型，并且它提供了对输出外观的非常精细的控制。关于转换修饰符的可用类型及如何控制格式化细节的完整介绍，可通过UNIX的man命令来了解。要查阅它们，在终端窗口中输入如下内容：

```
man 3 printf
```

注意 Foundation框架提供了NSLog，这是另一个日志函数。它类似于printf，但是，它添加了打印对象变量的功能。它还添加了对程序名称、日期，以及用小时、分钟、秒和毫秒表示的时间的输出。如果只想知道一两个变量的值，那么，这些添加的信息可能在视觉上会分散你的注意力，因此，本书在不需要NSLog的额外功能的某些地方使用了printf。NSLog将在第3章中介绍。

1.8 使用gcc和gdb

当为Mac OS X或iOS编写程序时，应该使用Xcode来编写和编译自己的程序，Xcode是Apple的集成开发环境。本书第4章介绍了如何建立一个简单的Xcode项目。然而，对于本章和第2章中的练习所需的简单C程序，你会发现，在自己喜爱的文本编辑器中编写程序，然后使用GNU编译器gcc通过命令行编译和运行它们，会更容易。为此，需要：

1）一个终端窗口。使用Mac OS X所带的Terminal app（/Applications/Terminal）。如果你使用另一种UNIX环境，并且习惯了xterms，则可能需要下载和使用iTerm，这是OS X本地终端应用程序，它的作用和xterm（http://iterm.sourceforge.net/）类似。

2）一个文本编辑器。Mac OS X带有vi和emacs，或者你可以使用自己所拥有的一款不同的编辑器。

3）命令行工具。这可能还没有在你的系统上安装。要检查一下，在命令提示行中输入gcc。如果得到的回应是/usr/bin/gcc，则说明它们都安装好了。然而，如果没有回应，或者回应是gcc: Command not found，则必须从安装盘来安装命令行工具，或者从下载的Xcode磁盘镜像来安装（可以在Mac Dev Center Web页面http://developer.apple.com/mac/中找到开发者工具的当前版本的一个链接）。开始安装过程，然后，当你进入Custom Install阶段时，确保选中UNIX Dev Support，如图1-2所示。继续安装过程。

现在准备好进行编译了。如果源代码文件名为MyCProgram.c，可以通过在命令提示行中输入如下命令来编译它：

```
gcc -o MyCProgram MyCProgram.c
```

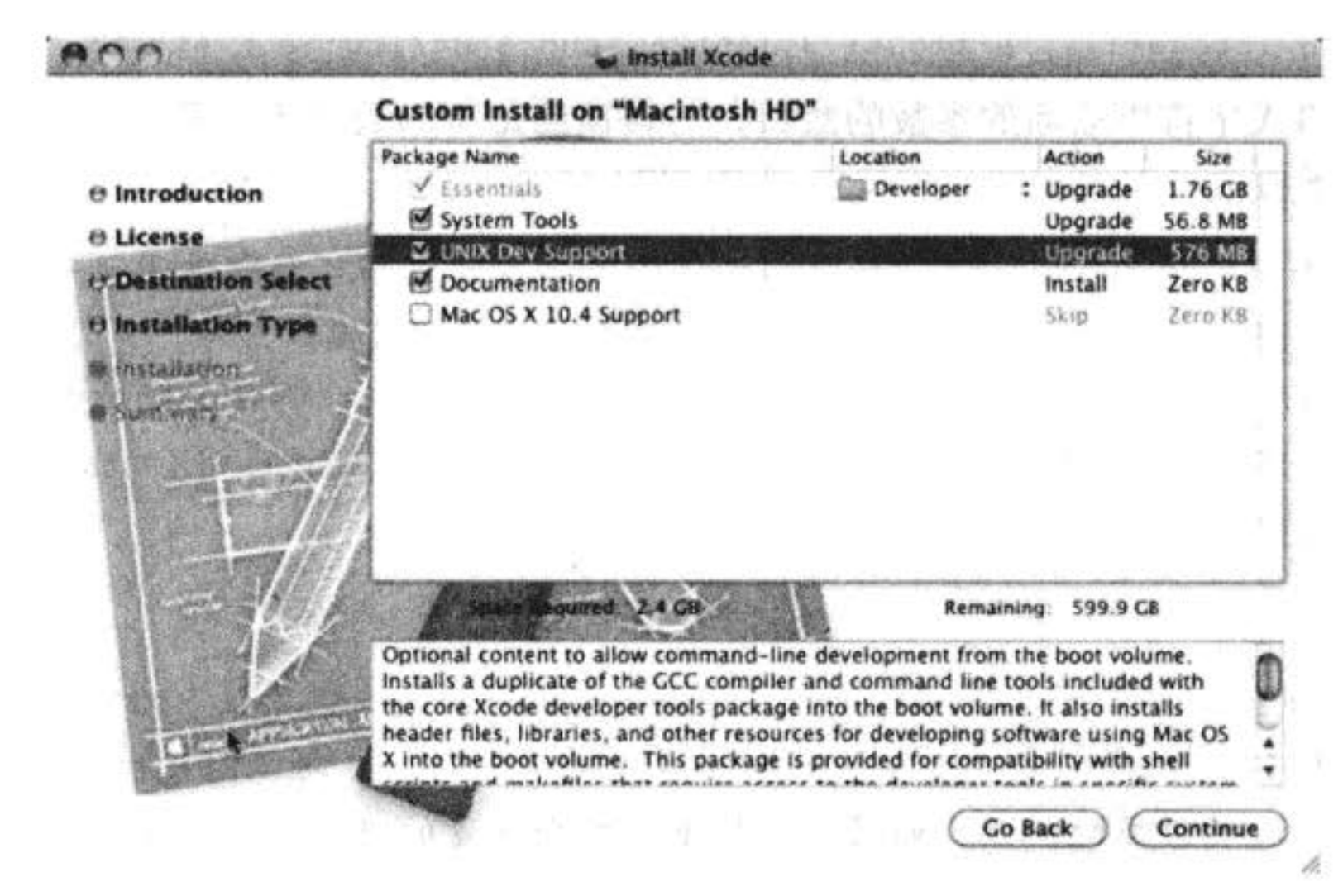


图1-2 安装命令行工具

-o标志允许给编译器指定一个名称，以用于最后的可执行文件。如果编译器指出你犯了一两个错误，那么返回去修正错误，然后再次尝试。当你的程序成功通过了编译，可以在命令提示行输入可执行文件名称来运行它：

```
MyCProgram
```

如果想要使用GNU调试器gdb调试你的程序，则在编译时必须使用-g标志：

```
gcc -g -o MyCProgram MyCProgram.c
```

-g标志使gcc将gdb的调试信息附加到最终的可执行文件。要使用gdb来调试一个程序，输入gdb，后面跟着可执行文件的名称：

```
gdb MyCProgram
```

可以在GNU的Web站点www.gnu.org/software/gdb/或者Apple站点http://developer.apple.com/mac/library/documentation/DeveloperTools/gdb/gdb/gdb_toc.html找到关于gdb的文档。此外，还有很多站点提供了关于使用gdb的说明。可通过搜索“gdb tutorial”找到相关文档。

1.9 小结

本章回顾了C语言的基本部分。第2章将继续回顾C程序的内存布局、声明变量、变量作用域和生命周期，以及内存的动态分配等内容。第3章开始进入本书的主题：面向对象编程和Objective-C的对象部分。

1.10 练习

1. 编写一个函数，返回两个浮点数的平均值。编写一个小程序，来测试你的函数，并记录输出。接下来，将函数放入到一个单独的源文件中，但是特意在拥有main函数的文件中不声明该

函数，看看会发生什么情况。现在，在有main程序的文件中添加函数声明，并且验证该声明解决了出现的问题。

2. 编写另一个平均数函数，但是，这一次，尝试将结果通过函数的某个参数传递回来。你的函数应该按如下声明：

```
void average( float a, float b, float average )
```

编写一个小的测试程序，并且验证你的函数无法工作。这说明你不能通过设置一个函数的参数来影响调用环境中的一个变量。现在，修改该函数及其调用，以传递一个指向调用环境中的变量的指针。验证该函数可以使用指针来修改调用环境中的一个变量。

3. 假设你有一个函数，int FlipCoin()，它随机地返回一个1表示头部或一个0表示尾部。说明如下的代码段是如何工作的：

```
int flipResult;

if ( flipResult = FlipCoin() )
  printf("Heads is represented by %d\n", flipResult );
else
  printf("Tails is represented by %d\n", flipResult );
```

在第6章我们将看到，类似该例子中的if条件，会在Objective-C对象的初始化过程中用到。

4. 单位矩阵是一个方形的数字数组，其中，对角线上（行号等于列号的元素）都是1，而其他的任何位置都是0。2×2的单位矩阵如下所示：

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

编写一个程序来计算并存储4×4的单位矩阵。并且程序在完成矩阵计算后，它应该以格式整齐的方形数组输出结果。

5. 斐波那契数（http://en.wikipedia.org/wiki/Fibonacci_number）是一个数字序列，它在自然界和数学中都经常出现。前两个斐波那契数定义为0和1。第n个斐波那契数是前面的两个斐波那契数的加和：

$$F_n = F_{n-1} + F_{n-2}$$

编写一个程序来计算和存储前20个斐波那契数。在计算这些数字之后，程序应该每行一个数字，并还带上索引地输出这些数。输出行应该如下所示：

```
Fibonacci Number 2 is: 1
```

使用#define来控制程序所产生的斐波那契数的编号，以便可以很容易地修改它。

6. 重新编写练习5中的程序，使用一个while循环来代替使用一个for循环。
7. 如果要求计算前75个斐波那契数呢？如果使用int来存储数字，将会有一个问题。你会发现第47个斐波那契数太大了，而无法用int来存储。如何解决这个问题呢？
8. 从iPhone App Store中可用的小费计算程序的数目来看，真的有很多人都已经忘记了如何做乘法运算。帮助那些不会做乘法运算而又买不起iPhone的人编写一个程序，在所有$10到$50之间的账单上计算一个15%的小费（为了简单起见，以$0.50为增量）。显示出账单和小费。

9. 现在，让这个小费计算程序看上去更专业一些。添加一栏用来计算20%的小费（Objective-C程序员都是在档次较高的地方吃饭）。给每一栏添加一个相应的标题，并且使用一对嵌套的循环，使程序按照每$10的增量来输出一个空白行。

   使用转换修饰符%.2f而不是%f，这将限定账单和小费都输出两位小数。在格式字符串中使用%%，将会使`printf`输出一个单个的%字符串。

10. 定义一个结构，用来存储一个矩形。定义的这个结构，应包含一个点的坐标，还包含另外一个结构，后者通过存储宽度和高度来表示大小。矩形结构中的点表示矩形的左上角，另一个结构表示矩形的大小。（Cocoa框架定义了类似这样的结构，但是，现在你需要自行定义一个）。

11. 高效的计算机图形的一个基本原则是，“不到万不得已，不要自行绘制”。图形程序通常为每个图形对象保留一个边界矩形。当需要在屏幕上绘制图形时，程序将图形的边界矩形与表示窗口的矩形进行比较。如果两个矩形之间没有重叠，那么程序将会跳过绘制图形的尝试。总的来说，这通常是好办法，因为比较矩形比绘制图形要简单多了。

    编写一个函数，它接受两个矩形结构作为参数（使用练习10定义的结构）。如果两个矩形之间有一个非零的重叠的话，那么函数应该返回1；否则的话，返回0。编写一个测试程序，来创建一些矩形，并用它测试你的函数是否有效。

第2章 C 变 量

当用大多数常见的脚本编程语言编写一个程序时，几乎不必花时间来考虑变量。只是在使用变量时才创建它们，并且不必担心用完它们之后会发生些什么。语言的解释器会负责所有的细节。

当你在编译语言中编写代码时，事情就没那么简单了。必须告诉编译器每个变量的类型和名称，以声明任何将要在程序中使用的变量。编译器随后查看变量的声明类型，为其保留相应数目的字节，并且将变量名与这些字节关联起来。

本章我们将介绍Objective-C中变量声明的形式，以及编译器如何存储各种不同的变量。

2.1 Objective-C程序的内存布局

要理解本章的内容，就要知道Objective-C程序是如何管理内存的。图2-1给出了运行程序的虚拟地址空间的一个简图。

注意 虚拟地址空间是程序“看到的”地址空间。虚拟地址空间和实际的物理地址空间之间的转换，由操作系统和计算机的内存管理单元（Memory Management Unit，MMU）隐式地进行。

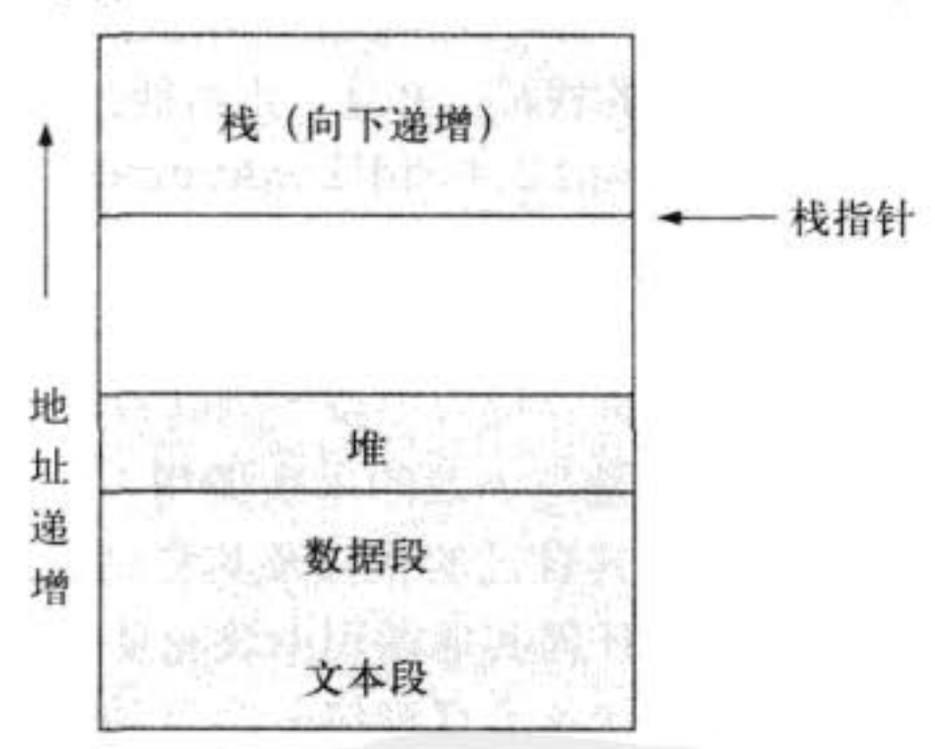


图2-1 一个运行的Objective-C程序的虚拟地址空间

按照虚拟地址从低向高的顺序：

- 文本段包含了程序的可执行代码和只读的数据。
- 数据段包含了可读写的数据，包括全局变量。
- 堆包含了根据请求分配给程序的内存块（参见本章稍后的2.6节）。当需要更多内存时，系统可能会向上扩展堆。
- 栈用于调用者函数。当调用一个函数时，系统为被调用的函数构建一个栈帧。栈帧是在栈的底部（最低的地址）构建的一个内存区域。栈指针（它指向栈的最低位置的地址）向下移动。栈帧包含了用于被调用函数的参数和局部变量的空间，以及用来保存在函数调用的过程中需要保存的任何寄存器的值的空间，还有用于一些控制信息的空间。当函数返回时，栈指针恢复到其最初的值（较高的地址），并且控制返回给调用者函数。关于这一过程，需要记住的重要的一点是，当函数返回之后，其栈帧的内容就不再有效了。

注意 一些人喜欢把栈看做是向上扩展的，类似于桌面上物理地堆放着的一摞纸。他们说栈的指针指向“栈的顶部”。尽管可以这么考虑，但是，栈在内存地址中是向下递增的。调用一个函数会将栈指针向一个较低的地址移动，并且，从函数返回则会将栈指针移回到一个较高的地址。

2.2 自动变量

在函数或子程序中声明的变量叫做自动变量或局部变量。考虑如下的简单函数：

```
void logAverage( float a, float b)
{
  float average;
  average = 0.5 * (a + b);
  printf( "The average is %f\n", average);
}
```

在上面的例子中，average是一个自动变量。

关于自动变量，有一些重要的事情需要记住：

- 自动变量是在栈上创建的。它们只是从声明它们的地方到函数的末尾有效（参见本章稍后的2.5节）。当函数返回时，系统将栈指针移动到前面的帧的底部。后续对其他的函数的调用会创建新的栈帧，并且，很可能会覆盖分配给前面的函数的自动变量的内存位置。如果使用参数10.0和12.0调用logAverage()，当执行到printf语句时，变量average的值为11.0。如果随后再次调用logAverage()，在该程序顶部，average的值不太可能是11.0。
- 自动变量不是由系统初始化的。在再次给它们分配一个值之前，它们包含了上次使用它们时在内存位置中留下的随机垃圾信息。
- 自动变量与函数的单次调用有关。如果有函数调用自己的递归代码，则每次对函数的调用都有其自己的栈帧及其自动变量的副本。在一个调用中修改该变量的值，不会修改堆栈上的任何其他调用中该变量的值。
- 使用一个&运算符将一个自动变量的地址取出，并且分配给函数之外的一个指针变量来保存，这是很糟糕的做法，因为只要函数退出，指针变量就指向了垃圾信息。

注意 自动变量所谓自动的含义是，当调用使用了自动变量的函数时，它们的存储在栈上分配。然而，仍然必须用一个声明告诉编译器与自动变量相关的信息。

函数参数

函数参数实际上是在调用函数时，已经用提供的值初始化了的自动变量。它们在栈上创建，并且当函数返回时，它们变成无效的。正如第1章所介绍的，可以在函数体中重置它们的值，但修改它们的值对于调用者函数中相应的变量没有影响。

2.3 外部变量

在main函数或任何子程序作用域之外的源文件中声明的变量，叫做外部变量。在如下的示例中，pageCount就是一个外部变量：

```
int pageCount;

main()
{
...
  printf("The current page count is: %d\n", pageCount );
}

void addpage()
{
  pageCount++;
  ...
}

void deletePage
{
  pageCount--;
  ...
}
```

尽管这并不总是一种好的设计选择，外部变量有时候也用作全局变量，以便在不同的函数和不同的源文件之间共享信息。关于外部变量，有一些重要的事情需要记住：

- 编译器在虚拟地址空间的数据段中分配了外部变量的内存位置。外部变量在程序的生命周期中都持久存在，它们不会超出作用域或消失，并且，只有当给它们分配一个新值时，它们的值才会改变。
- 如果没有显式初始化它，那么编译器会将外部变量初始化为0。
- 外部变量在单个函数的作用域之外是可见的（可用的）。实际上，外部变量是全局标记。除非将一个外部变量声明为static的（参见2.4.3小节），否则它可能对于任何源文件中的任何函数都是可见的。

2.4 声明关键字

Objective-C定义了几个关键字，它们可以用来修饰声明。

2.4.1 auto

auto关键字只在一个函数内使用，用来告诉编译器，该变量是一个自动变量。由于这是默认的，所以很少使用它。一般不会看到它。

2.4.2 extern

当想要引用一个在不同文件中声明的外部变量时，使用extern关键字。以extern开头的声明使得编译器知道变量的名称和类型，但是，该声明不会导致编译器为该变量保留任何的存储。

下面的语句告诉编译器，pageCount是一个int，并且，已经在程序中的其他某个地方使用一个声明为其保留了存储：

```
extern int pageCount;
```

如果还没有在程序中的其他地方将pageCount声明为一个外部变量（没有使用extern关键字），而使用如下的一条语句：

```
int pageCount;
```

则程序将会编译，但是，它不会连接。因为编译器通过extern声明得到满足，但是，连接器失败了，因为它查找不存在的pageCount。

2.4.3 static

根据是用于函数中的变量还是用于外部变量，static关键字具有不同的含义。当它用于函数中的变量时，static关键字创建了类似于外部变量的一个变量：

```
void countingFunction()
{
  static int timesCalled;
  timesCalled++;
  ...
}
```

在前面的例子中，timesCalled保存了调用countingFunction函数次数的一个计数。

关于函数的static变量，需要记住如下的几个要点：

- 编译器在数据段中为一个函数static变量创建存储。
- 函数static变量初始化为0，除非为其提供一个显式初始化值。
- 函数static变量的值在函数调用之间保持不变。
- 在对函数的多次调用中（即便是函数递归调用其自身），对一个函数static变量的引用，都是指向同一个内存位置。

函数static变量与外部变量的唯一区别在于，它只是在声明它的函数的作用域内可见。

当static关键字用于外部变量时，它将该变量的可见性限定在了声明它的文件之中，并且对其他的源文件隐藏。如果在一个函数体外部编写了：

```
static int pageCount;
```

那么pageCount只能由同一个源文件中的函数引用。如果试图使用一条extern语句来声明pageCount，从而要在一个不同的源文件中使用它：

```
extern int pageCount;
```

则连接器将会失败。

2.4.4 register

register关键字提示编译器，它所修饰的变量会频繁地被引用：

```
register int count;
```

结果，编译器可能选择将该变量存储在一个寄存器中（以便于快速访问），而不是存储在RAM中。然而，编译器并不一定遵从这一提示。

2.4.5 const

const关键字告诉编译器应该将一个变量视做常量。对于任何试图修改该变量的行为，编译器会提示一个错误：

```
const int maxAttempts = 100;
maxAttempts = 200;  // This will cause a compiler error.
```

当const用于指针变量时，顺序很重要。声明应该从左向右读取。如下的示例声明了ptr是指向一个整数的常量指针：

```
int a = 10;
int b = 15;

int *const ptr = &a;

*ptr = 20;  // OK. a is now 20.

ptr = &b; // Error, ptr is a constant.
```

不能修改ptr的值以指向一个不同的变量，但是，可以使用ptr来修改它所指向的变量。

将const放在声明的其他内容的前面，则有着不同的含义。ptr现在是指向一个int常量的指针：

```
int a = 10;
int b = 15;

const int *ptr = &a;

*ptr = 20; // Error, ptr's contents are constant.
a = 20;    // OK, a itself has not been declared as const.

ptr = &b;  // OK, ptr now points at b.
```

可以修改ptr以指向另一个变量，但是，不能通过ptr来修改它所指向的任何变量。即便在上面的例子中，变量自身并没有被声明为const，也不能通过ptr来修改它（但是，在上面的例子中，由于a没有被声明为const，因此如果直接访问它的话是可以修改它的）。

2.4.6 volatile

volatile关键字声明了一个变量的内容可能会被程序的main线程以外的其他参数修改。然后，编译器将避免像没有该关键字时那样优化。例如，如果变量的存储实际上是属于外部设

备的一个硬件寄存器，并且该设备已经被内存映射到了程序的地址空间中，就会发生这种情况。

考虑这样一种情况，其中shouldContinue被映射到一个硬件设备上的一条控制线：

```
int shouldContinue = 1;  // initialization

// Check the control line before each pass through the loop.

while (shouldContinue ) doStuff;
```

如果doStuff所表示的代码没有修改shouldContinue，优化的编译器将会得出shouldContinue没有被修改的结论，并且，它将代码优化为如下的等价形式：

```
while (1) doStuff;
```

遗憾的是，这是一个无限循环。将shouldContinue声明为volatile则阻止了编译器进行这一优化：

```
volatile int shouldContinue;
```

2.5 作用域

变量的作用域就是可以“看见”该变量语句的范围。看见，在某种意义上，意味着一条语句可以引用该变量，而编译器不会给出一条错误信息来提示说它不知道关于该变量的任何信息。

2.5.1 自动变量的作用域

自动变量可以在函数体中的任何位置声明。自动变量的作用域为从声明它的位置直到函数的末尾。在声明该变量之前，不能使用它：

```
void someFunction()
{
  int a;
  a = 7; // OK, a is in scope
  ...
  b = 7; // WRONG, b is not in scope at this point
  int b;
  ...
}
```

当声明了自动变量的函数返回之后，自动变量的作用域就消失了。在该函数之外，它是不可见的。

2.5.2 复合语句和作用域

一条复合语句的内部是一个独立的作用域（一条复合语句，是包含在花括号中的一个语句序列）。可以在复合语句中的任意位置声明一个变量，该变量从声明它的地方到复合语句结束的地方，都是可见的。复合语句存在于其中的作用域，叫做该复合语句的外围作用域。在外围作用域中声明的变量，在复合语句中是可见的；但是，在复合语句中声明的变量，在外围作用域中是不可见的：

```
void someFunction()
{
  int a = 7;
    {
      int b = 2;
      int c;

      c = a * b;  // OK, a is visible inside the compound statement
    }

  int d = 2 * c;  // WRONG,  c is not visible at this point
}
```

编译包含上述代码段的程序，将会产生一个编译器错误：

```
someFunction.c:12: error: 'c' undeclared (first use in this function)
someFunction.c:12: error: (Each undeclared identifier is reported only
someFunction.c:12: error: once for each function it appears in.)
```

如果在一条复合语句中声明了一个变量，而该变量与外围作用域中的一个变量具有相同的名称，那么，在内部变量的作用域中，内部的变量将会覆盖（隐藏）外围作用域中的变量：

```
void someFunction()
{
  int a = 7;
  int b = 2;
    {
      int c;
      c = a * b;  // c is 14, enclosing scope a is still visible;

      int a = 10; // This hides the enclosing scope's variable a
      c =  a * b; // c is now 20
    }
}
```

2.5.3 外部变量的作用域

外部变量（在函数之外声明的变量）从文件中声明它的地方到文件的末尾可见。正如本章前面介绍的，可以通过使用一个extern关键字，将一个外部变量的可见性扩展到不同的文件。而为遵循减少可见性的指导方针，以一个static关键字开始一个外部变量的声明，会将该变量的作用域限定在声明它的文件之中。static关键字优先于对同一变量的任何extern声明。

2.6 动态分配

本章到目前为止介绍的变量声明，都是静态分配的。在编译时，必须告诉编译器所需的变量的数目和类型。

> **Static和Static**
>
> 单词static在这里重载了。不要将通用术语“静态分配（static allocation）”和前面小节所讨论的static关键字的含义搞混淆了。是的，这可能会引起混淆。

在很多情况下，在编译时指定所需的内存量是一个问题。假设想要读入一个灰度（每像素8位）图像的像素。你将需要一个字节数组来存储它们：

```
#define MAX_WIDTH 1000
#define MAX_HEIGHT 1000
unsigned char pixels[MAX_WIDTH * MAX_HEIGHT];
```

但是，MAX_HEIGHT和MAX_WIDTH实际上应该是多少呢？如果把它们指定得太小了，将会无法读取图像。可以将它们设置得很大，但这也会有问题：对于一般情况来说，你浪费了很多内存，并且程序的“内存占用”没必要很大，否则这会影响到性能。这也很容易导致墨菲定律的出现，只要你直接编码大小，某人、某地总会产生一个比直接编码的大小更大的图像文件，并且期望你的程序来读取它。

这个问题的解决方法是使用动态分配。动态分配允许在运行时，也就是当程序在执行的时候，再请求更多的内存。当需要更多的存储时，向系统请求一块内存。当使用完内存时，要释放它。用于动态分配的字节，位于图2-1所示的名为“堆”的区域中。

> **注意** 堆是描述这一区域的常用Unix术语。Mac OS X实际上有一个更加复杂的系统。在OS X中，这一区域在技术上叫做一个malloc区域（参见下面“malloc区域”中的内容）。在大多数情况下，本书将继续使用术语“堆”。

动态分配由两个函数处理：malloc和free。使用一个参数来调用malloc以请求内存，该参数指定了所需的字节数。malloc返回指向请求的数目的字节的一个指针，随后可以将这个指针强制转换为想要的数据类型，并且赋给一个变量，而这个变量的类型就是指向所请求的类型的一个指针。

> **注意** 曾经，当没有可用的内存可分配时，malloc返回NULL。然而，OS X和iOS都使用延迟分配技术。使用延迟分配，malloc返回指向请求的内存的一个指针，但是，在执行一段代码来访问该内存之前，系统不会为该请求分配内存资源。结果，如果你多次请求内存，并且没有使用该内存或没有释放该内存，那么malloc可能会返回一个非NULL的值，即便没有更多内存可用也是如此。在这种情况下，使用返回的指针可能会导致程序崩溃。假设给RAM和交换空间指定一些典型值，即便有可能，你也是极少会看到这种情况在OS X系统上发生。在iOS中，遇到这种情况之前，应用程序很可能会从系统接收到一条低内存警告。

要计算需要的字节数，可以使用sizeof函数来确定每一项所需的字节数，然后，用该结果乘以所需的项数。例如，一个8位灰度图像所需的字节数是（假设已经从文件头部的某些信息中读取了图像的高度和宽度）：

```
numBytesNeeded = imageHeight * imageWidth * sizeof( unsigned char );
```

内存分配编写如下：

```
numBytesNeeded = imageHeight * imageWidth * sizeof( unsigned char );
unsigned char* pixels =
    (unsigned char*) malloc( numBytesNeeded );
```

使用完malloc分配的内存后，调用free函数释放这些内存：

```
free( pixels );
pixels = NULL;
```

通过对free的调用，从而使程序可以重用该内存。当这些字节释放以后，可以通过后续对malloc的调用，将它们拿来用于不同的用途。动态分配的黄金法则是：如果你分配了某些内存，当你用完之后，要负责把它们收回来。否则，你的应用程序将会“泄漏”内存。因此你的进程的大小将会比所需的还要大，并且，你的程序及同时运行的其他程序可能会为此遭殃。

注意 每个free调用必须匹配一个相应的malloc。释放没有分配内存的指针，或者释放相同的指针两次，都会导致程序立即崩溃。

注意 使用已经释放的一个指向字节的指针，可能会破坏程序的内存，并且导致随机的崩溃，而且这很难调试。将pixels设置为NULL提供了一种安全的方法。如果你在代码中犯了错误，并且试图使用NULL指针，那么你的程序将会立即崩溃。这样的问题及其位置将会很明显。

malloc区域

OS X有一个或多个malloc区域，而不是一个统一的堆。第一个malloc区域，叫做默认malloc区域，它在第一次调用malloc时创建。标准的malloc函数从默认的malloc区域获取其字节。也可以使用malloc_create_zone函数来创建额外的malloc区域。

使用其他的malloc区域是一项高级的技术，在特定的情况下，它可以用于优化性能。它允许成组地频繁访问内存位置相邻的变量，或者一次释放多个变量（通常释放一个完整的区域）。

尽管没有正式废弃，Apple现在不鼓励在Objective-C中使用除默认malloc区域以外的其他区域（参见http://developer.apple.com/mac/library/DOCUMENTATION/ Performance/ Conceptual/ ManagingMemory/Articles/MemoryAlloc.html）。

2.7 小结

本章介绍了如何声明Objective-C变量。要点如下：

- 必须声明要在Objective-C程序中使用的任何变量。声明告诉编译器，该变量的类型及其他信息。作为对声明的响应，编译器为变量安排存储（内存中的某些字节），并且将变量名与存储关联起来。

- 编译器可以将变量放置在3个地方：放在栈上，作为数据段的一部分，以及放在堆中。
- 在函数中声明的变量是自动变量。当调用函数时，自动变量在栈上创建。编译器不会初始化自动变量，并且，只有在声明它们的函数作用域之内，它们才是有效的。
- 在任何函数作用域之外声明的变量叫做外部变量。外部变量在编译时创建于数据段中，并且初始化为0（除非提供显式的初始化）。在程序的生命周期内，它们都有效。
- Objective-C提供了如下的关键字来修饰变量声明：`auto`、`static`、`extern`、`register`、`const`和 `volatile`。
- 在程序运行时，可以使用`malloc`函数来请求额外的内存。用`malloc`获取的内存来自于堆。
- 使用完用`malloc`获取的内存后，必须使用函数`free`将该内存释放回堆。

本章完成了对Objective-C的非对象部分的快速概览。在第3章中，本书将继续介绍面向对象编程及Objective-C对象的基础知识。

2.8 练习

1. 编写一个小程序并声明一个外部变量，该程序要调用一个函数，该函数声明并使用一个自动变量（使用哪个函数来做这个练习并不重要。你可以只是声明并初始化一个本地变量，然后，使用一条`printf`语句来输出两个变量的值）。构建该程序，然后在gdb中运行它。在函数中设置一个断点。当达到断点时，查看自动变量和外部变量的地址。可以通过在gdb命令提示行中输入如下内容来显示一个变量的地址：

   ```
   p &variableName
   ```

   结果与你在本章中学习的内容一致吗？
2. 修改练习1中的程序，添加第二个函数，它也声明一个自动变量。修改最初的函数，以使它调用新函数。验证栈确实是向着较低的地址扩展：在新函数内部设置一个断点并在此停止。显示新的函数的自动变量的地址。然后，在gdb的命令提示行中输入：

   ```
   up 1
   ```

   这将会把gdb的注意力转移到前一个栈帧，从而可以显示调用者函数的自动变量的地址。
3. 编写一个函数，显示调用它的次数。将次数存储在一个函数作用域的静态变量中。调用该函数数次以验证计数是正确的。有什么方法让计数从一个非零的值开始吗？
4. 本练习使用两个源文件。在第一个文件中，声明一个外部变量并且将其初始化为某个值。在第二个源文件中，放置一个函数来记录外部变量的值（别忘了，在第二个文件中进行`extern`声明）。构建并运行程序，以测试其有效性。如果忽略了`extern`声明，将会发生什么？恢复第二个源文件中的`extern`声明，并且在第一个文件中给变量声明添加关键字`static`。看看会发生什么事情？
5. 编写一个程序来计算并存储前10个整数的平方。当它确实计算了所有的10个数时，程序应该使用`printf`来显示结果。使用`malloc`来获取保存计算的值的内存，而不是声明一个保存10个整数的数组。不要忘了，在完成之后释放内存。

第3章　面向对象编程简介

早期的高级计算机语言，例如Fortran、COBOL和C都是面向过程的。以面向过程的语言编写的程序的实质结构，是要按照顺序执行的一系列任务。过程式编程特别适合于某些类型的问题，例如，求解数学方程式。然而，有很多类型的问题，例如，用户界面编程，却是过程式编程所不适合的。在如今计算的很多领域中，占有主导地位的编程泛型是另一种叫做面向对象的编程泛型。Objective-C是面向对象语言的一种。它添加了结构和语法来支持使用过程式的C语言进行面向对象编程。

本章从一个简介开始，介绍面向对象编程的基础知识。然后说明这些概念在Objective-C中是如何实现的。本章3.3节列出了Objective-C对于C语言添加的内容。

3.1　面向对象编程

面向对象编程是编程的一种形式，它将程序组织为彼此交互的对象的集合。对象是一组相关的变量及一些过程，这些变量对问题空间中的某些内容建模；而这些过程叫做方法，它们表示对象知道如何去执行的动作。对象通过发送和接收消息来通信。发给对象的一条消息，就是对对象的一个请求，要求它执行其方法之一。

例如，一个绘图程序可能有一个Shape对象，它表示用户可以在屏幕上绘制的任意形状（见图3-1）。一个Shape对象可能有多个变量，用于存储确定其边框的点、颜色及其在绘图中的位置。要在屏幕上绘制这个Shape，需要给Shape发送一条draw消息。作为响应，Shape将会执行其draw方法（该方法包含实际绘制的代码）。

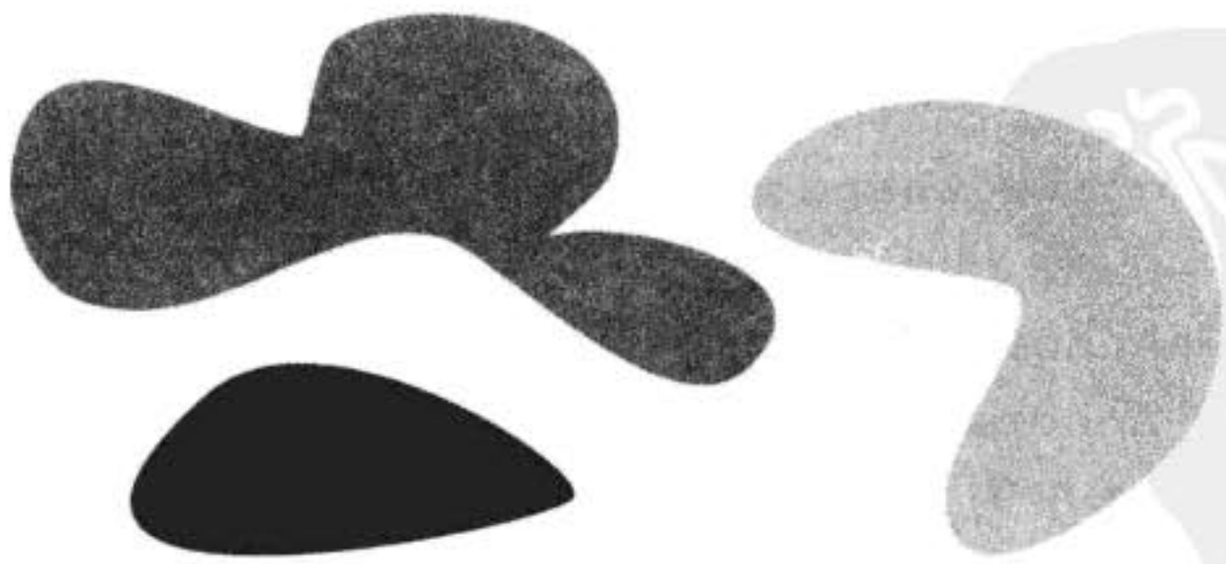


图3-1　某些图形

3.1.1　类和实例

对象是根据其类来分类的。每个对象都属于某个类，它是该类的一个实例。用现实世界的

事物来类比的话，一辆保时捷跑车的设计和规范中所包含的信息是一个类。这些信息定义了一辆保时捷跑车是什么，以及如何构造一辆保时捷跑车。你在赛道中实际驾驶的跑车是Porsche类的一个实例。回到计算机程序的世界，类是定义了一组变量和一组方法的一个模板或菜谱，这一组变量叫做实例变量，这一组方法则包含了实现这些方法的代码。对象（类的一个实例）是实际的一块内存，用来为类中定义的一组变量提供存储。一个给定的类可能有多个实例。每个实例是内存的一个单独区域，并且拥有类中定义的实例变量的一个副本。

3.1.2 方法

方法类似于函数，但是，它们并不完全相同。方法代表定义它们的类的一个实例而执行。当一个对象执行一个方法时，该方法可以访问对象的数据。如果向一个Shape对象发送一条draw信息，draw方法将使用Shape对象的边框、位置和颜色信息。如果向不同的Shape对象发送同样的draw消息，draw方法将使用相应的Shape的边框、位置和颜色信息。

3.1.3 封装

封装，有时候也叫做信息隐藏，指的是向类的用户隐藏一个类的内部工作。它是通过减少程序的不同部分之间的联系，从而减少复杂性的一种方法。对象只能由一个定义的接口来进行封装，也就是其类所实现的一组方法。

使用Shape类编写代码的程序员，不需要知道Shape的数据在内部是如何存储的或者绘制代码是如何工作的。他只需要知道，当一个Shape实例接收到一条draw消息时，它将绘制自身。

封装给了类的开发者修改其实现的自由，而不会影响到使用该类的代码。Shape类的开发者可能会改变Shape用来存储描述其边框的点的坐标系统；或者在执行绘制代码之前，通过检查以验证Shape位于窗口边界之中，从而添加一些改进来提高draw方法的性能。只要新的坐标系统和新的改进的draw方法在所有情况下都产生与旧版本同样的结果，其他的代码就都不需要修改。使用Shape类的代码，在程序下一次编译和连接时，自动从改进的draw方法获益。

3.1.4 继承

继承提供了一种方法，它通过扩展和修改一个已有类的行为来创建新的类。假设你想要给绘制程序添加一个AnnotatedShape类，用来绘制一个形状及形状下面的文本注释，如图3-2所示。

图3-2 一个AnnotatedShape

可以从头开始编写AnnotatedShape类，但是，这可能会有些浪费时间。AnnotatedShape类中的大多数代码都是重复Shape类中的代码。如果你可以指出AnnotatedShape与Shape存在的以下区别，就太好了：

- AnnotatedShape多了一个变量，这是一个字符串变量，用来保存注释的文本。
- AnnotatedShape有两个额外的方法，它们设置和获取注释的文本。
- AnnotatedShape的draw方法与Shape的draw方法的实现不同。它必须同时绘制注释和形状。

如果如下这样定义AnnotatedShape，那么，以面向对象的说法：

- AnnotatedShape继承自Shape。
- AnnotatedShape的draw方法的实现覆盖了Shape的draw实现。
- AnnotatedShape是Shape的一个子类。
- Shape是AnnotatedShape的一个超类。

3.1.5 多态

多态是不同类的对象响应同一消息的能力。例如，可以给绘制程序添加一个Image类，使它也可以绘制图像（位图）。就像Shape类一样，Image类定义了一个draw方法。每个类的draw方法的实现是不同的：Shape的draw方法知道如何绘制一个Shape，Image的draw方法知道如何绘制图像。如果有一个要绘制的图形对象的列表，可以通过给列表中的每个对象发送一条draw消息，从而将它们绘制到屏幕上。即便每种情况下的消息都是相同的，也是列表中的Shape对象将执行draw的Shape版本，而Image对象将执行draw的Image版本。

3.1.6 面向对象语言的主要特点是什么

使用一种过程式语言来实现面向对象的编程风格，这是可能的，但是，会很困难，并且有些不令人愉快。以C为例，可以把结构用作对象，并且将常规的C函数用作方法。要给对象添加方法，可以给对象的结构添加字段，来保存指向方法函数的指针。也可以给每个方法函数定义一个参数，该参数允许传入一个指针，该指针指向在其上执行该方法函数的对象。但是，你将很快遇到问题。在单个数组中存储对象的不同类并遍历它们，即便像这样简单的事情，也需要艰难地使用强制类型转换运算符。

还有其他的一些缺点。如果你没有想清楚自己设计的所有用意，构建自己的对象系统可能导致不稳定的代码，并且可能出现不希望的后果。一个更为严重的问题是，自建的对象系统很可能是一次性的设计。一次性的定制设计，使得很难将来自其他源的代码或库整合到你的程序中。你将必须从头构建一切，这要花费较长的开发时间，并且产生bug的机会要增加很多。

对于已经内建了面向对象编程系统的语言来说，使用其进行一个面向对象的设计，要容易很多。

3.2 Objective-C简介

Objective-C是C的面向对象的扩展。正如你在本节及后面将看到的，Objective-C对C的添加很少。主要添加是，定义类的方式、调用方法的方式，以及十几个关键字和针对编译器的指令。这些概念很精巧而且很强大，但是，需要记住的语法相对很少。

除了给C添加了一个对象系统，Objective-C没有剔除掉C的过程式部分。这样你可以吸取两个世界的精华。可以使用对象来完成那些适合它们的任务，而使用Objective-C（普通C）的过程式部分来解决它们最为适用的问题。用普通C作为该语言的基础，也使得Objective-C程序可以很容易地使用那些用普通C编写的库和代码。

本节以概览的方式介绍了Objective-C如何实现面向对象编程的概念。Objective-C的核心部分将在第5章、第6章和第7章中介绍。

3.2.1 定义类

Objective-C类定义有两个部分：接口部分和实现部分。接口部分声明了类的实例变量及其方法。实现部分则包含了实现类的方法的代码。接口部分通常放在一个头文件中，按照惯例，头文件按照类来命名。实现部分放在一个以.m为扩展名的文件中，也以类来命名。.m文件告诉编译器，该文件包含了Objective-C的源代码。

接口部分

类定义的接口部分如下所示：

```
@interface className : superclassName
{
   Instance variable declarations
}

Method declarations

@end
```

- @interface和@end用来表示接口部分的开始和结束。Objective-C中以@字符开始的单词，是编译器指令，用来指示编译器，而不是可执行的代码。
- @interface后面跟着类名、一个冒号及类的超类的名字。
- 类的接口变量的声明，位于一对花括号之间，放在@interface行的后面。
- 在实例变量声明之后，声明类的方法。

注意 *方法声明必须位于实例变量声明之后。如果将方法声明放置在花括号之前，那么这些代码将不会编译。*

为了让这些内容更加具体，考虑一个类Accumulator，它保存加和的总数。Accumulator类有一个示例变量来保存总和，还有向总和添加，打印出总和将总和及重置为0的方法。这是一个很小的例子，但是，它让你对语法的了解更具体。如下是Accumulator的接口部分：

```
1 @interface Accumulator : NSObject
2 {
3   int total;
4 }
5
6 - (void) addToTotal:(int) amount;
7 - (int) total;
8 - (void) zeroTotal;
9
10 @end
```

按照命名惯例，将这段代码放入到一个名为Accumulator.h的文件中。

- 第1行：这个类的名称为Accumulator。按照惯例，Objective-C类名以一个大写字母开头。Accumulator的超类是NSObject。
- 第3行：Accumulator类有一个实例变量total。该变量在类的接口部分的声明，没有保留任何存储。当创建Accumulator的一个实例的时候，新的实例会得到它自己的total的副本，但是，没有与该类自身相关的存储。
- 第6～8行：这些是方法声明。方法声明的形式如下：

```
-(return type) method_name:(argument_type) argument;
```

打头的连字符(-)表示这些是实例方法，也就是由类的一个实例来执行的方法。也有可能声明类方法，即由类自身执行的方法。类方法以一个加号打头(+)：

```
+ (return type) class_method_name:(argument_type) argument;
```

类方法将在本书第7章介绍。

按照惯例，Objective-C方法名以一个小写字母开头。方法名的剩下部分则遵循CamelCased命名法，即名称中其余的每个单词，都以一个大写字母开头，并且，不会用一个下划线字符(_)将单词分隔开。

实现部分

实现部分包含了类的方法的实现。Accumulator的实现部分如下所示：

```
 1 #import "Accumulator.h"
 2
 3 @implementation Accumulator
 4
 5 - (void) addToTotal:(int) amount
 6 {
 7   total = total + amount;
 8 }
 9
10 - (int) total
11 {
12   return total;
13 }
14
15 - (void) zeroTotal
16 {
17   total = 0;
18 }
19
20 @end
```

将这些代码放入到一个名为Accumulator.m的文件中：

- 第1行：编译器在编译相应的实现部分时，需要类的接口部分的信息，因此Accumulator.m导入了Accumulator.h文件。

- 第3行和第20行：方法的实现放在一个@implementation指令（后面跟着类名）和一个@end指令之间。
- 第5～8行：一个方法实现包括一个重复的方法声明（后面没有结束的分号），其后跟着实现该方法的代码，这些代码放在一对花括号中。方法实现的主体，其编写的方式与编写一个普通C函数的方式相同。
- 第7行：注意，一个方法实现能够直接访问其类的实例变量，即便它没有声明这些变量。

继承

所有的Objective-C类，除了根类以外，都继承自某个其他的类。在某些特殊的条件下，例如分布式对象系统，需要关注的唯一根类就是NSObject。几乎所有的Objective-C类，都直接或间接地是NSObject的子类。NSObject定义了类工厂方法alloc，它负责为那些需要与Objective-C的内存管理系统交互的对象实例和实例方法分配内存。其他的类，例如Accumulator，通过继承NSObject而获取这些方法。本书第6章将详细介绍继承。

3.2.2 类名作为类型

Objective-C对象创建于堆上。这意味着，当创建一个对象实例时（在本章稍后，将会看到这一点），我们得到的是一个指向堆上的某些字节的指针。当人们说："这个变量保存了一个Accumulator"或者"这个变量保存了一个Accumulator对象"，他们实际的意思是"这个变量保存了一个指针，它指向Accumulator类的一个实例"。

使用类名作为类型，来声明保存指向对象实例的指针的变量。下面这行代码声明了变量anAccumulator，它保存了指向Accumulator类的一个实例的一个指针：

```
Accumulator *anAccumulator;
```

显式的星号(*)在表示anAccumulator是一个指针时是必需的。如果忽略了星号，例如：

```
Accumulator anAccumulator;  //WRONG
```

将会导致一个编译器错误（编译器认为你试图在栈上创建该对象，而这在Objective-C中是不允许的）。

3.2.3 消息（调用方法）

如果不能够让对象为你做某些事情，那么即便拥有对象，也没有太大的用处。我们需要使用某种方式来让对象执行其方法。在某些面向对象的语言中，例如Java和C++中，方法调用就是与特定对象相关联的函数调用，它们通过在对象变量的后面用一个点号，然后再附加函数来完成调用。如果用C++编写Accumulator，应该编写如下的代码来给anAccumulator的总和添加一个数字：

```
anAccumulator.addToTotal( 137 );
```

Objective-C使用一种不同的方法，叫做消息，它借用自计算机语言Smalltalk (http://en.wikipedia.org/wiki/Smalltalk)。使用消息机制，我们要给对象发送一条消息，而不是调用一个函数。如下的代码行显示了一条消息的表达形式：

```
[receiver message]
```

receiver是想要其执行一个方法的对象。之所以称其为接收者（receiver），是因为它接收该消息。message就是想要执行的方法的名称，加上给该方法的任何参数。当计算message表达式时，receiver执行与message中的名称对应的方法（第5章将详细介绍这是如何实现的）。

以Accumulator类的一个实例为例，如下的代码把137加到了anAccumulator的总和中：

```
[anAccumulator addToTotal: 137];
```

如下的代码获取了anAccumulator的总和的当前值，并且将其存储到名为currentTotal的变量中：

```
int currentTotal = [anAccumulator total];
```

首先，消息似乎只是编写函数调用的另一种方法。然而，它不是。差别很细微，但是，功能差异很大。这里接收者和消息在编译时没有绑定到一起。消息只是包含一个方法名。它不会将该方法名与任何特定的类或方法实现联系起来。不同的类可能提供同一方法名的不同实现。当计算消息表达式时，接收者对象执行在其类中定义的该方法的版本。

消息还考虑到了动态技术，而这一技术对于一种语言是不可能实现的，例如C++，它要求在编译时将方法调用绑定（确定）。例如，一个Objective-C消息表达式的消息部分是一个变量，这是有可能的。在这样的一个消息表达式中，实际发送的消息是在执行时确定的，而不是在编译时。也可以通过编程来构建消息表达式，并且将其保存为对象以便稍后执行（参见第16章相关内容）。

多态

正如本章前面所述，向作为不同的类的成员的对象发送同一条消息，并且让它们以特定于类的方式执行该消息，这一功能叫做多态。

作为示例，考虑一个简单的绘图程序。该程序定义了Shape类和Image类，以表示程序可以绘制的不同的项。两个类都实现了一个名为draw的方法，但是，每个类的draw方法的实现是不同的。要在屏幕上绘制一个图形，程序遍历对象的一个列表，并且按照顺序绘制每个对象：

```
while( /* check for loop ending condition goes here */ )
  {
    id graphic = [graphicEnumerator nextObject];
    [graphic draw];
  }
```

graphicEnumerator是负责将集合中的对象一个一个地传递出来的类的一个实例（第10章将介绍枚举器）。

id是一个Objective-C类型，它被定义为一个通用的“指向对象的指针”。当一个变量将要在不同的时刻保存不同的类的对象时；或者在编译时，我们不知道在运行时将要在该变量中存储何种类型的对象；就使用该类型。可以将任何类型的对象赋值给一个id变量，而不会得到编译器的警告。

在这个例子中，graphic可能在循环的一次迭代中包含一个Shape对象，而在另一次迭代

中包含一个Image对象。当graphic包含一个Shape的时候，draw消息引发执行Shape的draw实现。当它包含一个Image的时候，draw消息引发执行Image的draw实现。

尽管id使得编写多态代码很方便，多态的真正源头还是Objective-C消息系统。相反，我们应该将graphic声明为指向一个完全无关的类的一个指针，这个类没有实现draw方法：

```
while( /* check for loop ending condition goes here */ )
  {
    // Bad practice, but might work at run time
    Shmoo *graphic = [graphicEnumerator nextObject];
    [graphic draw];
  }
```

如果你这么做，编译器将会在一条警告中指出，Shmoo类没有实现draw方法，你的同事会认为你疯了，而你的老板可能炒你的鱿鱼；但是，只要在执行时graphic包含一个Shape、一个Image，或者任何其他实现了draw方法的类的一个实例，这段代码就会工作。总之，重要的是运行时的对象类型，而不是变量类型。

注意 将一个对象赋值给一个变量，而该变量声明为指向一个不同的类的对象的指针，这么做并不会改变最初的对象的类。对象的类是在创建对象的时候设置的，并且不会改变。

3.2.4 类对象和对象创建

Objective-C中的每个类都由一个类对象来表示，该类对象可以代表类执行方法。一个Objective-C类对象实际上是一块内存，其中包含了它所代表的类的实例变量和方法的相关信息。类对象是一个名为Class的特殊类的实例。

可以在消息表达式中使用类名作为接收者，从而给一个类对象发送消息：

```
[classname classmessage]
```

当给一个类对象发送消息时，消息中的方法名必须是一个类方法的名称。

类对象的主要用法是作为一个工厂，来创建类的实例。使用类方法alloc来创建对象实例，所有的类都从NSObject继承了alloc方法：

```
Accumulator *anAccumulator = [Accumulator alloc];
```

alloc为对象分配了内存；填充了名为isa的编译器所创建的一个特殊的实例变量，它是指向对象的类对象的一个指针；清空所有其他的实例变量，并且返回指向该对象的一个指针。

警告 前面的代码行是一个说明，可以用一条初始化消息来嵌套alloc消息，如下一个例子所示。

所有的Objective-C对象都需要初始化。创建Accumulator的一个初始化实例的语法是：

```
Accumulator *anAccumulator = [[Accumulator alloc] init];
```

执行前面的代码行：

1）向Accumulator类的类对象发送一条alloc消息。

2）类方法alloc为Accumulator对象分配内存，并且返回了指向未初始化的对象的一个指针。

3）未初始化的Accumulator对象变成了一条init消息的接收者。init是Accumulator从NSObject继承的一个实例方法。

4）init返回指向已经初始化后的Accumulator对象的一个指针，然后，将其赋值给变量anAccumulator。

要了解关于类对象和类方法的更多内容，参见本书第7章。

注意 你不必关心如何创建类对象。编译器会为你创建类对象。

3.2.5 内存管理

当在堆上分配内存时，当内存不再需要时，需要释放该内存。如果使用malloc分配了一块内存，当使用完以后，必须使用free释放它。Objective-C使用一种略微不同的方法来管理对象内存，即所谓的引用计数（也叫做保留计数或管理内存）。每个Objective-C对象都有一个引用计数，它记录了使用该对象的地方的数目。当一个对象的引用计数变为0时，对象就销毁了，其字节也返还给堆。

方法retain和release会自增或自减一个对象的引用计数：

```
[anObject retain];  //Increments anObject's retain count
[anObject release];  //Decrements anObject's retain count
```

对象通过一个值为1的引用计数来创建。如果使用一个名称以alloc或new开头（或名称包含了copy）的方法创建了一个对象，你就“拥有”了该对象。当使用完该对象后，必须向其发送一条release消息，以结束其创建并放弃你的所有权。如果通过名称不是以alloc或new开头（或不包含单词copy）的方法来接收一个对象，则你并不拥有该对象。在大多数情况下，对象将在获取它的作用域中保持有效。但是，如果想要更长久地使用对象，例如，将其存储到一个实例变量中，那么，必须向其发送一条retain消息来“获得对象的所有权”。如果对象的创建者释放了对象，那么这么做会防止对象的引用计数变为0。最终，当使用完对象后，必须给对象发送一条release消息来对应retain消息。

要更系统地了解本节所介绍的内存管理，参见本书第14章。要了解垃圾收集的内存管理替代方法，参见第15章。

3.3 Objective-C添加

本节是Objective-C对C所做出的添加的一个简单概览。

3.3.1 运行时

Objective-C需要运行时。运行时是C函数的一个动态链接库，在支持Objective-C的所有系统上都提供了运行时库。它负责建立和运行Objective-C的消息系统。运行时在幕后默默地坚守岗位。在一般情况下，不需要调用运行时的函数与其直接交互。

3.3.2 名称

和普通C一样，Objective-C的对象中的名称也是区分大小写的。Objective-C有多个命名惯例，例如本章前面介绍的针对类名和方法名的命名惯例。这些惯例并不是语言强制要求的，但是，它们总是被广泛遵从的。如果你违反了它们，熟悉Objective-C的人将很难阅读你的代码。

其他的一些命名规则如下：

- 类可以声明一个实例变量，它可以与另一个不同的类中的实例变量具有相同的名称。
- 类可以声明一个方法，该方法可以与一个不同的类中的方法具有相同的名称，就像本章前面关于多态的小节中所展示的那样。
- 方法可以与实例变量具有相同的名称（对于返回一个实例变量的值的方法来说，这很常见。参见本书第12章）。
- 类的实例方法可以拥有与同一个类的类方法相同的名称。
- Apple将名称以下划线(_)字符开始的方法，视为保留供Apple内部使用的方法。

3.3.3 消息表达式

Objective-C对于C所做出的最为重要的添加，就是消息表达式：

```
[receiver message]
```

`receiver`是一个对象，或者，严格地讲，是类的一个实例的指针。`message`是`receiver`的类所定义的一个方法的名称，以及该方法所接受的任何参数。当执行包含了消息表达式的语句时，Objective-C运行时确定接收者的类，并且在该类的方法列表中查找`message`中所包含的方法名，从而执行该方法相应的代码。第5章将介绍消息表达式。

注意 当消息中的方法是一个类方法时，接收者是一个类对象或者是类名（可以用来在消息表达式中代替类对象）。

3.3.4 编译器指令

以@字符开头的单词是编译器指令，而不是可执行代码。我们已经看到了`@interface`，它表示一个类定义的接口部分的开始；还有`@implementation`，它表示实现部分的开始；还有`@end`，用来表示这些部分的结束。附录A给出了Objective-C编译器指令的一个完整列表。

3.3.5 直接量字符串

直接量字符串是保存字符串文本的常量。Objective-C使用`NSString`的常量实例而不是普通的C字符串，而`NSString`是在`Foundation`框架中定义的一个类。可以用创建一个C字符串直接量相同的方式来创建一个`NSString`直接量，只不过在字符串开始处添加一个@，例如：

```
"The Big Apple"   // Literal C string
@"The Big Apple"  // Literal NSString
```

注意 严格地讲，`@"The Big Apple"`是一条编译器指令，它告诉编译器创建一个`NSString`直接量，其文本为`"The Big Apple"`。

3.3.6 Objective-C关键字

id

id是保存“指向对象的指针”的类型。声明为id的变量，可以保存指向任何Objective-C对象的一个指针，该对象独立于对象的类：

声明：

```
id myObject;
```

它告诉你（和编译器）myObject是指向一个对象的指针。位于该地址的字节，是某些类的一个实例的内存表示。

可以把一个更加具体的类型的变量，赋值给一个id类型的变量，或者反之亦然，不需要显式的强制类型转换，例如：

```
NSString* string1 = @"The Big Apple";
id something;
NSString* string2;
something = string1;
string2 = something;
```

编译器将会静默地假设你知道自己在做什么。注意，实际上，id保存的指针是内建到类型中的。这里不需要使用一个星号(*)：

```
id *myObject;  // WRONG !
```

最后，不要把id与void*搞混淆了。声明：

```
void *someBytes;
```

没有说明someBytes是什么。它只是指向某些字节的一个指针。这些字节表示什么，对于编译器来说是未知的。

nil

nil是一个定义的常量，它表示“指向没有对象的一个指针”。尽管nil定义为零，但它通常用于一个对象指针必须是空的情况，而不是直接使用零。

使用零不会导致任何真正的问题：

```
[anObject setObjectInstanceVariable: 0];  // Bad form
```

然而，这被认为是一种糟糕的形式。使用nil来替代，这提醒你将要设置为空的是一个对象指针。考虑如下替代方式：

```
[anObject setObjectInstanceVariable: nil];  // Correct form
```

在Objective-C中，可以向一个接收者发送一条消息，消息的值为nil。nil消息表示没有操作：它们不做任何事情，并且继续执行下一行代码。它们不会导致程序崩溃。

这一功能使你不必再编写如下所示的守护代码：

```
// This check is unnecessary
if ( anObject )
  {
```

```
    [anObject someMethod];
}
```

BOOL

在其大多数历史阶段，C都缺乏一个定义的布尔类型。它通过计算表达式来确定真值。如果一个表达式计算为0，则被认为是假；相反，则为真。C99标准添加了一个布尔类型，bool，并且还添加了真值true和假值false。

Objective-C有自己的布尔类型，BOOL，并且还有真值常量YES和假值常量NO。BOOL不是一个基本类型。它是无符号的char的一个typedef（别名）。YES和NO只是为1和0定义的常量。

由于Objective-C继承了所有的C类型，因此我们可以在自己的Objective-C程序中使用bool类型。然而，Cocoa框架和大多数已有的Objective-C代码使用BOOL。尽管可以在bool和BOOL之间相互转换，除非你的程序用到的库使用了bool，否则干脆忘掉bool会更容易一些。

注意 尽管Objective-C当前的版本是基于C99标准的，但Objective-C最初只是作为缺乏布尔类型的一个较早版本的C的扩展而开发的。尽管C99的bool类型也可以使用，但大多数Objective-C社群使用Objective-C的BOOL。

SEL

SEL是保存了一个Objective-C方法名表示的一种类型。SEL是selector的缩写。方法名有时候叫做选择器，因为运行时使用它们来选择要执行的代码，以响应一条消息（见第5章）。

由于性能的原因，用于这一决定的是一个SEL，而不是方法的实际字符串名称。字符串不便于操作，只是比较两个字符串的相等性，就是一个很慢的、多步骤的过程。比较两个SEL要快很多：

```
SEL aSelector = @selector( aMethodName );
```

IMP

IMP是一个typedef，用于“一个指针，它指向接受参数id、SEL及可能的其他参数并且返回id的函数”。

编译器最终将Objective-C方法转换为常规的C函数。当执行一个方法时，运行时使用方法的选择器来找到执行该方法的实际函数，然后执行该函数。但是，偶尔为了性能的目的或者某些幕后的技巧，会有直接调用实现了一个方法函数的情况。可以通过获取该函数的一个指针，然后使用指针来调用该函数，从而做到直接调用。可以用来获取这样的指针的方法和运行时函数，以IMP类型返回该指针：

```
IMP methodImplementation =
   [anObject methodForSelector: @selector( aMethodName )];
```

注意 在文档和图书中，IMP也用作表示方法实现的一个术语，例如“使用methodForSelector:获取IMP”。

Class

Class是保存了Objective-C的类的引用的一个类型。在Objective-C中，类自身也是对象。由于它们是对象，类必须实例化某些类。这个类，也就是类的类，称作（请注意）Class。就像id和IMP一样，而且与任何其他的类类型不同，Class本身是一个指针类型，并且不需要星号(*);例如：

```
Class *myClass; //WRONG !
```

第7章将介绍Class对象。

3.3.7 Cocoa数字类型

本节中的类型不是Objective-C语言的一部分。它们都定义于Cocoa框架中，但是，我们会看到它们经常在这里提及。从Mac OS X 10.5开始，Apple已经使用定义的类型替代了Cocoa框架中int和float的大多数出现，而定义的类型的长度取决于代码是编译为32位可执行程序还是64位可执行程序（参见附录C）。

NSInteger

NSInteger替代了Cocoa框架中大多数int的出现。它在32位环境中定义为int，在64位环境中定义为long（64位整数）。

NSUInteger

NSUInteger是NSInteger的无符号形式。它替代了Cocoa框架中的大多数unsigned的出现。在32位环境中，它是一个无符号的32位整数；在64位环境中，它是一个无符号的long（无符号的64位整数）。

CGFloat

CGFloat替代了float。当针对32位环境编译的时候，它是一个float（32位）；当针对64位环境编译的时候，它是一个double（64位）。Foundation框架提供了一个定义的常量，CGFLOAT_IS_DOUBLE，如果需要通过编程知道CGFloat在当前的环境中是一个float还是一个double，使用下面的语句：

```
if ( CGFLOAT_IS_DOUBLE )
  NSLog( @"Double !\n");
else
  NSLog(@"Float !\n");
```

NSLog

NSLog是Foundation框架中定义的一个用于字符输出的函数。NSLog不是Objective-C语言自身的一部分，但是，这里介绍它，是因为它用于本书中的很多示例和练习中。

NSLog类似于printf，但是有如下一些区别：

- NSLog写入控制台日志，也写入一个终端窗口。控制台日志是操作系统负责维护的一个消息日志。在OS X上，可以使用Console应用程序(/Applications/Utilities/ Console.app)来查看控制台日志。

- NSLog的格式字符串是一个NSString直接量，而不是一个C字符串直接量。
- NSLog在打印后自动换到一个新行。不需要在格式字符串末尾添加一个额外的\n。
- NSLog使用一个额外的转换修饰符，%@，它接受一个Objective-C对象作为其参数。在转换中，NSLog调用参数对象的description方法。该description方法返回一个NSString，用以描述该对象。返回的NSString替代了输出中的%@，如下面的例子所示：

```
NSString *aString =  @"Hello New York!";

NSLog( @"The object description is: %@", aString );
```

NSString的描述只是字符串本身。执行以上代码，将会产生如下的输出：

```
The object description is: Hello New York!
```

当创建自己的类时，可以覆盖description方法为自己的类提供定制的描述。

注意 *如果使用带有%@描述符的格式字符串，但是，忘记了提供一个对应的对象参数，那么NSLog将尝试向位于对象参数所应该放置的地址的字节发送一条description消息。这通常会导致程序崩溃。*

在一个发布的程序中，不应该使用NSLog语句（这么做会在客户的控制台日志中产生杂乱信息），但是，对于在学习和调试过程的简单输出来说，它很有用。

NSLog有一项功能可能很恼人，它在你要求其输出的内容前面加了一个长长的字符串信息，其中包括执行该语句的时间和日期（详细到毫秒）、可执行程序的名称，以及执行它的程序的进程id。如下的NSLog语句：

```
NSLog( @"Greetings from NSLog" );
```

将产生如下的输出：

```
2010-02-01 11:41:26.556 a.out[33955:903] Greetings from NSLog
```

为了显示上的清晰，在本书后面的部分中，在显示NSLog的输出时，我们去除了额外的信息。

3.4 小结

本章是对面向对象编程的一个一般性介绍。在阅读完本章后，你应该对于如下的概念有了一个基本的理解：类、对象、实例变量、方法、封装、继承、多态和消息。

本章的第二部分尝试说明这些概念是如何在Objective-C中实现的。第4章将在这部分内容的基础上，带领你一步一步地了解一个较小但完整的Objective-C程序。我们将学习如何使用Apple的集成开发环境Xcode来构建和运行程序。

第4章　第一个Objective-C程序

第3章介绍了面向对象编程的概念，并且介绍了Objective-C的基本语法。本章将这些概念和语法用到一个简单的程序中。按照传统，这个程序应该是一个“Hello World”程序，但是纠正一点，它应该是一个面向对象的“Hello World”程序。本章将一行一行地分析该程序。我们将使用Apple的集成开发环境Xcode，来一步一步地构建这个程序。

注意　IDE是管理应用程序开发的所有方面的一个程序，包括文件管理、代码的编写和编辑、编译、连接和调试。Apple的IDE是Xcode，它在Mac OS X上运行，用于为Mac OS X和iOS构建程序。对于快速测试程序来说，命令行编译和调试很有用，但是，对于较大的应用程序来说，这种方法变得不切实际。如果针对iPhone开发，需要使用Xcode来获取iPhone Simulator，并且要在一个iPhone、iPad或iPod Touch上载入和调试应用程序。

每一台新的Mac笔记本或台式机系统所附带的Mac OS X安装DVD中，带有Xcode的安装程序。如果你没有安装DVD，或者你的Xcode版本比较旧，可以到http://developer.apple.com/mac/下载最新版本。

4.1　使用Xcode构建

Xcode是一个较大的应用程序，但是，本书只是介绍构建一个简单程序所需的功能。在开始之前，需要建立一个目录来保存自己的项目，例如`~/ObjectiveCProjects`或者`~/Code/ObjectiveCProjects`。

注意　波浪线(~)是Mac OS X（及UNIX）用于表示根目录的缩写。例如，在我们的例子中：

```
~/ObjectiveCProjects
```

等同于：

```
/Users/rclair/ObjectiveCProjects
```

打开Xcode（位于/Developer/Applications目录下），并使用File > New Project (Shift-⌘-N)新建一个项目。应该会看到New Project窗口，如图4-1所示。

注意　本书中的屏幕截图都是使用Xcode 3.2。如果你使用Xcode的不同版本，你在自己系统上所看到的，可能与在本书中看到的有细微的差别。

图4-1 Xcode New Project窗口

在New Project窗口中进行如下操作：

1. 在左边的面板中，选择Mac OS X下方的Application。

2. 单击右上边的面板中的Command Line Tool图标以选中它。

3. 从右下方的弹出菜单中选择Foundation。

4. 单击Choose按钮（或者按下回车键）。

5. 弹出一个下拉表单，在这里为项目输入一个名称和一个用来保存它的目录。将项目命名为HelloObjectiveC，选择你刚才创建的目录，并且单击Save按钮（或者按下回车键）。

完成后，应该会看到HelloObjectiveC的项目窗口，如图4-2所示。

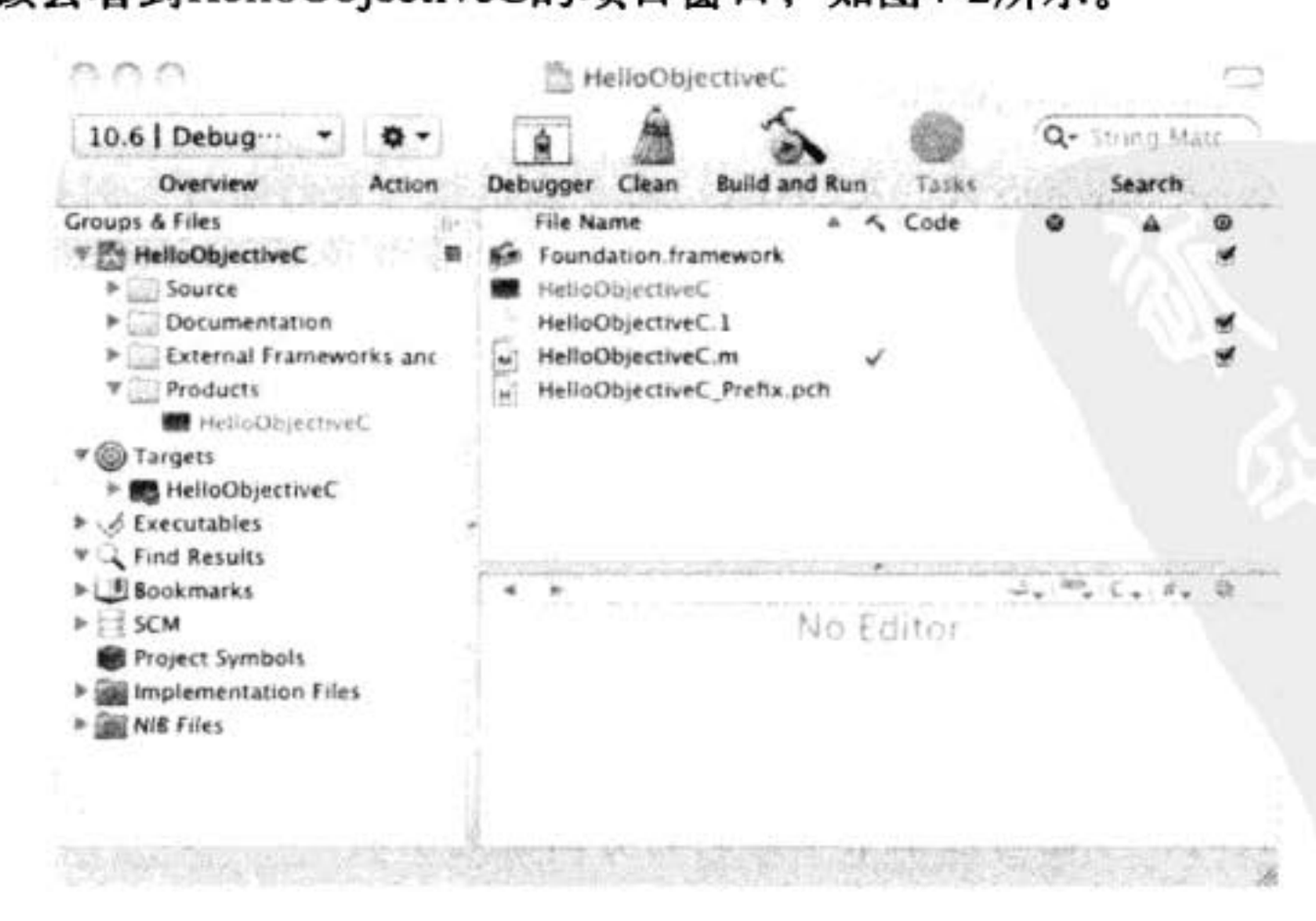


图4-2 HelloObjectiveC的项目窗口

注意 Xcode的可配置性很强。可以通过选择Xcode > Preferences (⌘)，并且当Preferences窗口出现时单击General，从而修改窗口的布置。也可以通过单击窗口使其获得焦点，并且选择View > Customize Toolbar来定制一个窗口的工具栏。可以从弹出的下拉列表中选择想要放到工具栏中的项，并且选择它们的排列方式。要了解使用Xcode的更多信息，选择Help > Xcode Quick Start。

项目窗口组织了关于项目的信息。左边的面板（名称为Groups & Files）是一个树状结构，它为程序组织了文件和其他资源。右边的面板是一个细节面板。它显示了关于你在树结构中选取的内容的额外信息。左上端的项目HelloObjectiveC，表示了程序及构建该程序所需的所有项。当选择HelloObjectiveC时，这些项的一个列表会出现在细节面板中。我们依次来看看这些项：

- Foundation.framework是一个框架，其中包含了非GUI Cocoa类（框架是一个包含动态链接库的目录结构，见第8章）。由于你告诉Xcode，你将构建一个Foundation工具，Xcode假设你将要使用Foundation中定义的类或函数。Xcode向项目中添加了Foundation.framework，以便程序能够根据它进行连接。
- HelloObjectiveC是完成后的程序。它显示为红色，因为Xcode没有找到它（Xcode无法找到程序HelloObjectiveC，因为你还没有构建它）。
- HelloObjectiveC.1是用于HelloObjectiveC的UNIX man页面的文件。Xcode将HelloObjectiveC.1添加到项目中，因为它认为你在构建一个UNIX工具。直接忽略这一项。
- HelloObjectiveC.m是你的程序的一个Objective-C文件，它包含了一个框架main函数。4.2节将详细介绍这个文件。
- HelloObjectiveC_Prefix.pch是一个预编译的头文件，当处理较多的头文件时，它帮助编译器更加高效地工作。Xcode为你配置了HelloObjectiveC_Prefix.pch，从而我们不需要在这个文件上做任何事情。

4.2 Objective-C程序结构

在细节面板中双击HelloObjectiveC.m，从而在一个编辑窗口中打开它。打开后，它应该如图4-3所示。

Xcode产生的代码如程序清单4.1所示。

程序清单4.1 HelloObjectiveC.m（第1版）

```
1 #import <Foundation/Foundation.h>
2
3 int main (int argc, const char * argv[]) {
4     NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
5
6     // insert code here...
7     NSLog(@"Hello, World!");
```

```
 8      [pool drain];
 9      return 0;
10 }
```

图4-3 编辑窗口中的HelloObjectiveC.m

注意 Xcode已经创建了一个较小的“Hello World”程序。第7行的NSLog语句打印出“Hello World”。如果你的唯一目标是得到一个“Hello World”程序，这很方便，但是，它没有提供太多的学习体验。在4.3节中，将删除NSLog语句并且用发出问候的一个对象来替代它，从而创建HelloObjectiveC的面向对象的版本。

接下来看看它的代码：

- 第1行：这一行代码为Foundation框架中的类导入头文件。Foundation包含了用于字符串、数组、字典和其他必需的类。要使用这些类，必须导入相应的头文件，因此，大多数程序将它们作为一个组导入。可以直接导入（如这里所示），也可以导入Cocoa/Cocoa.h。
- 第3行：这一行代码开始了程序的main函数（花括号之间的代码行）。每一个Objective-C程序都需要一个main函数。当程序开始执行的时候，OS把控制交给main函数。然后程序开始执行main函数体中的语句，直到到达main函数底部的return语句。

  main函数前面的int表明这个main函数应该返回一个整数。

  int argc和const char *argv[]都是main的参数。argv（表示argument variables，参数变量）是C字符串的一个数组。argc（表示argument count，参数数目）是argv中的元素的数目。

  argv的第一个元素argv[0]总是会存在，它是程序的名称。如果argc大于1，argv其余的元素都是从命令行传入到程序中的参数。

  对于大多数程序，不需要这些参数，但是，Xcode在为一个新程序生成文件时，都已经把它们准备好了。

- 第4行和第8行清空了自动释放池。自动释放池是Objective-C的引用计数内存管理系统的一项高级功能。第14章将介绍自动释放池，但是，对于本章的示例来说，你并不需要了解它。现在，只是忽略这些行。
- 第9行：在Mac OS X和UNIX系统中，返回值0表示程序完成了且没有错误；任何其他的返回值则意味着一个错误。除非你编写一个命令行工具，并且它拥有有意义的错误条件；否则，应该像第9行这样只是返回0。

构建和运行程序

在进入4.3节之前，先构建和运行HelloObjectiveC的这个版本。打开Xcode的控制台窗口，以便可以看到结果（Run > Console或Shift-⌘-R）。然后，构建程序并运行它(Build > Build and Run或者⌘-Return)。你将会在控制台窗口中看到“Hello World!”，如图4-4所示。

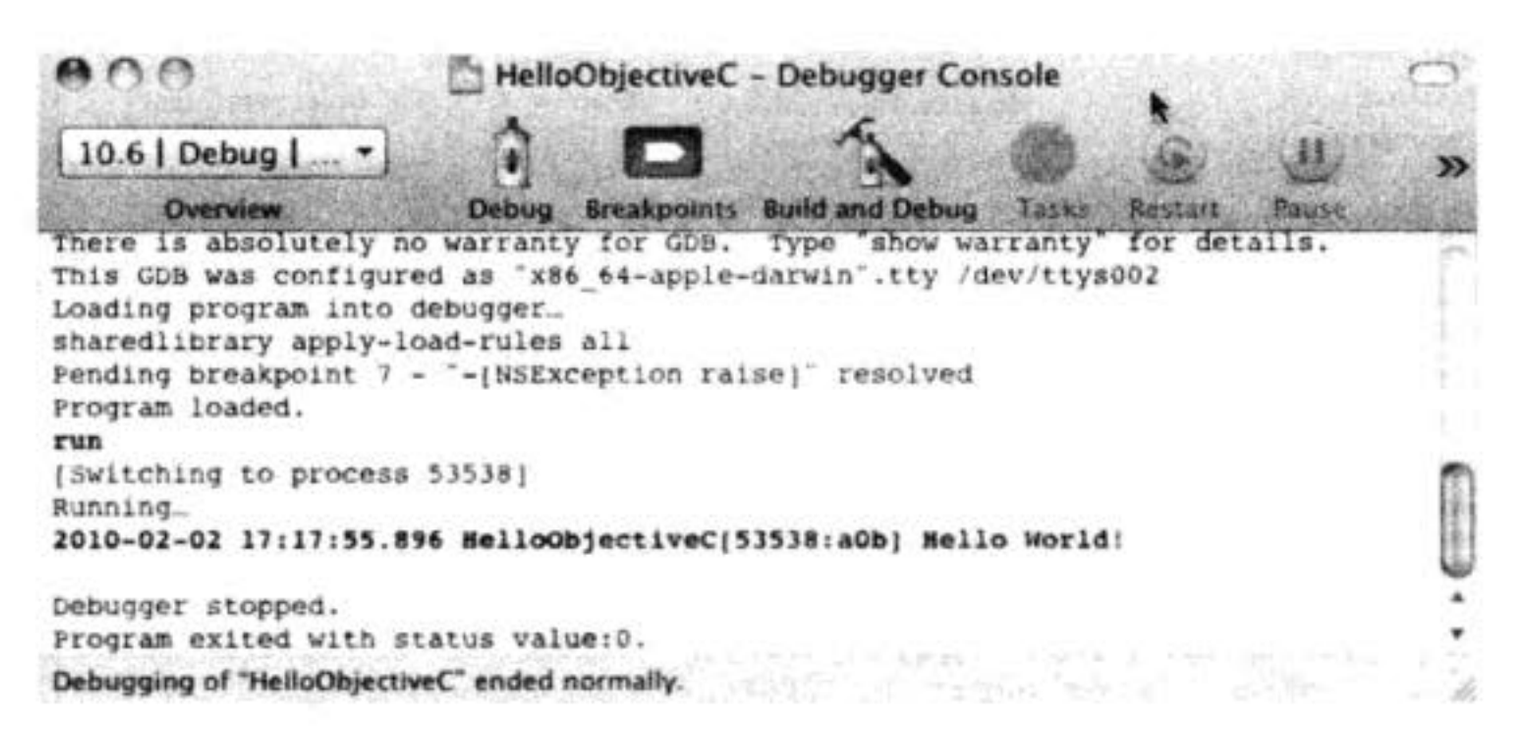


图4-4 运行HelloObjectiveC（第1版）的结果

GUI应用程序和main

Cocoa GUI应用程序的main函数非常简单：

```
int main(int argc, char *argv[])
{
    return NSApplicationMain(argc,
                             (const char **) argv);
}
```

main函数直接调用函数NSApplicationMain。

NSApplicationMain开始一个事件循环，它处理与用户的交互。当退出Cocoa GUI应用程序时，它跳出事件循环，并且NSApplicationMain返回。

Xcode为我们产生main函数。在GUI应用程序中，很少需要修改产生的main函数。

4.3 面向对象的Hello World

本节将带领你创建HelloObjectiveC的一个面向对象版本。它定义了一个类，名为

Greeter，它能够记住问候的文本并且根据命令发布问题。新版本的HelloObjectiveC的main函数创建了Greeter类的一个实例，设置了问候文本，然后，要求Greeter发布其问候。和所有的Objective-C类一样，Greeter类有一个接口部分，它声明了类的实例变量和方法；还有一个实现部分，其中包含了实现这些方法的代码。Greeter类的接口部分放在一个名为Greeter.h的头文件中，而实现部分放在一个名为Greeter.m的文件中。可以使用Xcode来创建这些文件的主体。

选择File > New File (⌘-N)。你应该会看到New File窗口，如图4-5所示。

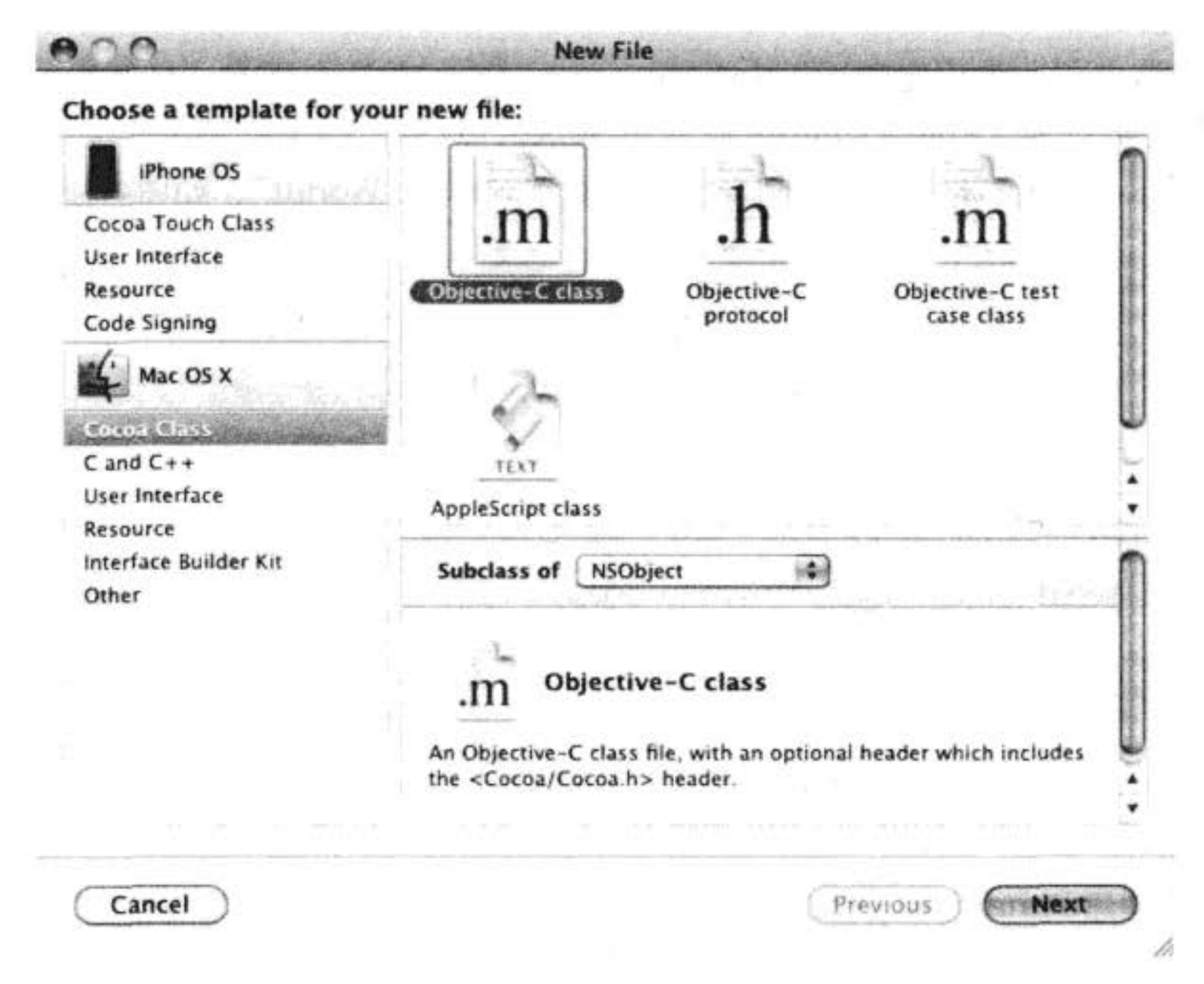


图4-5 Xcode New File窗口

在New File窗口中，进行如下的操作：

1）在左边面板的Mac OS X下方选择Cocoa Class。

2）单击左上方面板中的Objective-C图标以选中它。

3）从右下方面板中的弹出菜单中选择NSObject。

4）单击Next按钮（或者按下回车键）。New File窗口现在应该如图4-6所示。

5）将File Name从untitled.m修改为Greeter.m。

6）确保Also create“Greeter.h”的复选框为选中状态。

7）单击Finish按钮或按回车键。

查看项目窗口，可以注意到Xcode已经向项目的源添加了Greeter.h和Greeter.m。下面将填充这两个文件。

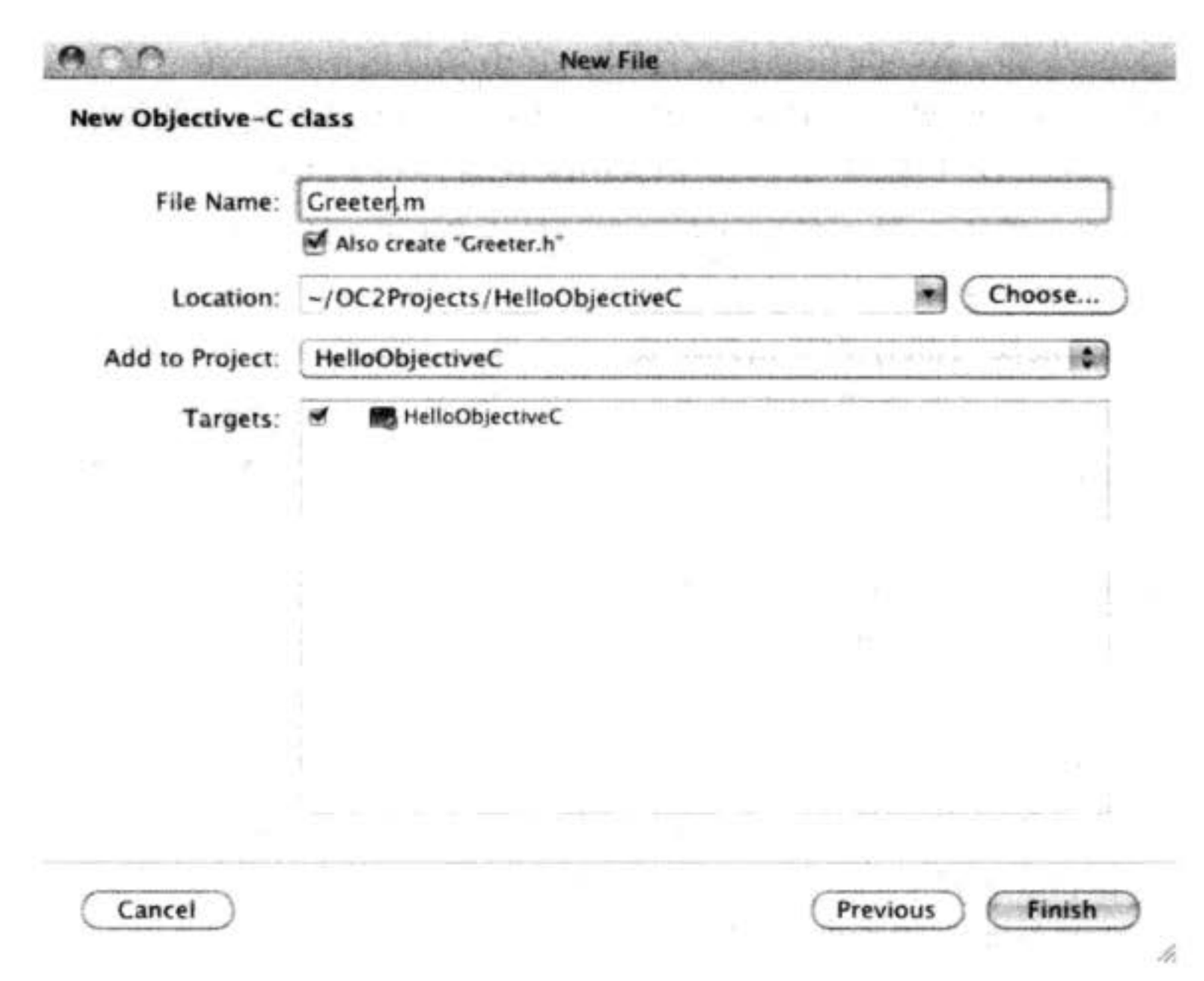


图4-6 Xcode New File窗口，第二个视图

4.3.1 Greeter.h

在项目窗口中双击Greeter.h，以打开一个编辑窗口，并且修改该文件的内容，以使其如程序清单4.2所示。

注意 Xcode在文件的顶部放置了很多注释，列出了项目的名称、文件名称、作者和版权提示。这些注释没有在这里给出。与所有的注释一样，它们对于程序的编译或执行没有影响。

程序清单4.2 Greeter.h

```
1 #import <Cocoa/Cocoa.h>
2
3 @interface Greeter : NSObject
4 {
5   NSString *greetingText;
6 }
7
8 - (NSString*) greetingText;
9 - (void) setGreetingText: (NSString*) newText;
10 - (void) issueGreeting;
11
12 @end
```

第1行导入Cocoa头部。Xcode在创建文件时，它不知道你的程序是一个命令行程序还是一个使用AppKit（Cocoa UI框架）的类的一个GUI程序。Xcode选择稳妥的做法，它包含了完整的Cocoa头文件集合。这个程序不会使用AppKit的任何内容，因此，它可能只是导入Foundation头文件。然而，导入Cocoa头文件（导入Foundation头文件和AppKit头文件）并没有什么坏处，因此，没有理由自找麻烦修改这一行。编译器指令，第3行的@interface和第12行的@end，表示接口部分的开始和结束。

@interface行包含了要定义的类的名称，后面跟着一个冒号及该类的超类的名称。第3行告诉编译器，这个部分定义了类Greeter，它是NSObject的一个子类。NSObject类提供了一些基本的行为，这是所有的Objective-C类都需要的行为。本书涉及的所有类，都是NSObject的子类或者是直接或间接是NSObject的子类的某些其他类。第6章将介绍类和继承。

当定义一个类时，编译器需要关于类的超类的信息。通过导入超类的头文件来包含这些信息。在这个示例中，Cocoa.h自动导入NSObject.h，我们不必做任何其他的事情。

花括号之间的代码行列出了类变量或实例变量。类的每个实例都有每个实例变量的副本。Greeter类定义了一个实例变量，greetingText。greetingText的类型是指向一个NSString对象的指针。NSString是一个Foundation类，它用于保存字符串（见第9章）。实例变量可能是对象指针类型或者C类型。

注意 greetingText声明的指针部分很重要。所有的Objective-C对象都是在堆上创建的，并且，通过一个指针来引用。新的Block功能（在Mac OS X 10.6时添加到了Objective-C 中）针对对象的类的这一规则引入了一个例外。要了解这一点，阅读第16章。

第8行、第9行和第10行声明了Greeter类的方法。声明方法的形式如下：

```
- (returnType) methodName;
```

如果该方法有一个参数，将会给该方法传递一个值，其形式是：

```
- (returnType) methodName:(ArgumentType) argumentName;
```

如果该方法不返回任何内容，则返回类型为void。打头的连字符(-)表示这些方法是实例方法。

setGreetingText:和greetingText分别设置和获取实例变量greetingText中保存的NSString。这样的方法叫做访问器方法，或者，更加随意地称为setter和getter方法。以实例变量的名称来命名该实例变量的访问器方法，这是惯例（并且，如果要使用Cocoa的所谓的键-值编码这一高级功能的话，这是必需的）。一个名为someObject的实例变量的setter和getter方法分别是setSomeObject:和someObject（这里采用了首字母大写方式）。

注意 在实例变量保存一个布尔值的情况下，getter也可能命名为isPropertyName。名为visible的布尔实例变量的getter，可能命名为visible或isVisible。

访问器方法的用法说明了信息隐藏（也叫做封装），这是面向对象编程的一个重要概念。使用Greeter类的代码并不会直接接触实例变量greetingText。这使得类的维护者能够自

由地修改类的实现，只要访问器方法仍然能够正确工作即可。例如，更新的Greeter类可能选择用NSDictionary对象而不是单独的实例变量来存储问候文本（NSDictionary是一个关联数组，它通过将每个对象与叫做键的另一个对象关联起来，从而存储对象，见第9章）。只要方法greetingText和setGreetingText：更新为使用字典来存储和访问问候文本，而使用Greeter对象的任何代码都不会受到这一修改的影响。

最终的方法issueGreeting，是使Greeter发布其问候的方法。

4.3.2 Greeter.m

类定义的实现部分包含了实际工作的代码。关闭Greeter.h的编辑窗口，然后，返回到项目窗口，并且双击Greeter.m从而在编辑窗口中打开它。修改该文件的内容，使其如程序清单4.3所示。

程序清单4.3 Greeter.m

```
1 #import "Greeter.h"
2
3 @implementation Greeter
4
5 - (NSString*) greetingText
6 {
7   return greetingText;
8 }
9
10 - (void) setGreetingText: (NSString*) newText
11 {
12   [newText retain];
13   [greetingText release];
14   greetingText = newText;
15 }
16
17 - (void) issueGreeting
18 {
19   NSLog( @"%@", [self greetingText]);
20 }
21
22 - (void) dealloc
23 {
24   [greetingText release];
25   [super dealloc];
26 }
27
28 @end
```

类的实现文件导入了包含类接口的头文件。Greeter.m首先导入Greeter.h。将头文件放在引号("") 中，这告诉编译器在当前目录中查找头文件，在本例中，也就是你的项目目录。没有必

要为框架的头文件添加一条import语句，因为Greeter.h已经导入了框架头文件，将Greeter.h导入到Greeter.m中，这会自动将框架头文件导入到Greeter.m中。

实现部分以一个@implementation编译器指令开头，后面跟着类名，且类名位于同一行中。它以@end指令结束。方法实现位于两个指令之间。

方法实现包括方法的声明文本（不带最后的结束分号），后面跟着一对花括号，其中包含了实际工作的代码。下面依次介绍了Greeter方法的实现。

greetingText

greetingText是一个非常简单的方法，它只是返回存储在实例变量greetingText中的、指向NSString的一个指针。

setGreetingText：

setGreetingText:有一些新内容。它接受一个参数，是指向NSString对象的一个指针，该对象则包含了我们想要让Greeter记住的问候文本。冒号是名称的一部分，方法名称是：

```
setGreetingText:
```

而不是

```
setGreetingText
```

程序清单4.3的第12行和第13行是消息表达式。消息表达式的形式是：

```
[receiver message]
```

receiver是要求其执行一个方法的对象。Message是方法的名称及方法所接受的任何参数。当执行消息表达式时，运行时系统确定了receiver的类，并在类的方法列表中查找该方法，然后，执行与该方法名称对应的代码。第5章将详细地介绍这一过程。

retain和release是NSString继承自NSObject的方法。它们是Objective-C的引用计数系统的一部分。在使用引用计数的程序中，每个对象都有一个引用计数。[⊖]对象的引用计数，记录了有多少其他的对象使用了该对象。当初次创建一个对象时，它的引用计数为1。向一个对象发送一条retain消息，会将对象的引用计数增加1；发送一条release消息，则会将引用计数减少1。当对象的引用计数减少到0时，会删除该对象；其内存也会返还给堆，并且，对该对象的任何其他引用都变得无效。

Greeter不会创建传递到参数newText中的对象，并且，它没有关于是否有其他代码会导致newText的引用计数减少为0的信息，也没有出现这种情况的信息。为了防止在Greeter仍然使用newText的时候释放了newText，Greeter通过向newText对象发送一条retain消息，获得其所有权。这确保了在Greeter使用newText的时候，其引用计数至少会保持为1。

在这个setter方法中，Greeter通过当前在greetingText中保存的NSString对象完成了操作。在第13行，它通过向对象发送一条release消息，从而对应了它初次存储的时候

⊖ retain count和 reference count这两个术语会在Objective-C的资料中交叉使用，本书中统一译为引用计数。——译者注

向该对象发送的一条retain消息，从而也就放弃了对该对象的所有权。发送一条release消息，并不一定意味着会销毁一个对象。release消息只是将对象的引用计数减1。只有当新的引用计数是0的时候，才会销毁对象。

> **注意** 初次调用setGreetingText的时候，greetingText将为nil。执行带有一个nil的消息表达式，并非是一个错误。它只是不执行任何操作，也就是说，什么也没有发生。

最后，第14行在greetingText中存储了新的NSString。

第12行和第13行的顺序很重要。如果两个变量都引用相同的对象，那么在保留newText之前释放greetingText，将会导致程序崩溃。release消息可能会将对象的引用计数减少为0，从而导致销毁该对象。后续的retain消息可能会被发送给一个无效的接收者。你可能认为，某人不可能在一行之中使用同样的参数两次调用setGreetingText:，但是，你不能保证这种情况不会发生。最好通过编写代码来防止这种情况。

issueGreeting

issueGreeting是发布问候的方法。它使用Foundation函数NSLog打印出问候字符串。在第19行，NSLog的第二个参数是一个消息表达式，而不是一个对象。当执行到第19行时，计算该消息表达式，并且该表达式的返回值用作与NSLog格式字符串中的%@修饰符相对应的参数。

self来自何处，它是什么？self是一个特殊的变量，编译器和运行时系统会自动创建它。self保存了指向一个对象的指针，而当使用self的方法被调用时，该对象作为接收者。当执行第19行代码时，self指向调用issueGreeting方法时用作接收者的对象。这里，self用作接收者以获取对象的greetingText。

dealloc

当一个对象的引用计数减少到0时，就会调用对象的dealloc方法，然后，其字节将返回给堆。这给了对象一个机会来执行任何必需的清理任务。dealloc在Greeter的超类NSObject中声明。Greeter类通过提供自己的dealloc实现，从而覆盖了其超类的实现。当向Greeter类的一个实例发送一条dealloc消息时，执行的是Greeter的实现。

Greeter的清理工作包括释放greetingText中保存的对象。引用计数的规则之一是，每次你向对象发送一条retain消息时，你必须最终用一条release消息来与retain对应。如果一个Greeter对象没有释放greetingText中保存的NSString，则它可能在Greeter对象被销毁之后仍然存在。如果没有对该字符串的其他引用，则可能会导致内存泄漏。

第25行很有趣。和self不同，super不是一个实际的变量或指针。当super用作接收者的时候，它告诉运行时执行对象的超类所提供的该消息的实现。这里，[super dealloc]调用NSObject对于dealloc的实现。实际上是NSObject的dealloc将一个对象的字节返回给了堆。

如果你的类实现了一个dealloc方法，你必须总是记住，在你的类的dealloc实现中调用超类的dealloc。如果你忘记了这么做，那么NSObject的dealloc实现将不会被调用，

并且，分配给该对象的字节也不会释放。消息[super dealloc]应该是一个类的dealloc方法的最后一行。当它返回时，该对象已经销毁了，并且不再有效。

注意 尽管前面介绍了调用[super dealloc]的方法，但是，你不应该自己调用dealloc。当一个对象的引用计数减少到0时，将自动调用dealloc。

4.4 HelloObjectiveC.m

现在，可以将Greeter类付诸使用了。main函数中的代码创建了类型为Greeter的一个对象，配置它，然后，要求它向你问候。关闭用来操作Greeter.m的编辑窗口，并且，在项目窗口中双击HelloObjectiveC.m。当编辑窗口打开时，编辑该文件，使其如程序清单4.4所示。

程序清单4.4 HelloObjectiveC.m（第2版）

```
1 #import <Foundation/Foundation.h>
2 #import "Greeter.h"
3
4 int main (int argc, const char * argv[])
5 {
6   NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
7
8   Greeter* myGreeter = [[Greeter alloc] init];
9
10  [myGreeter setGreetingText: @"Hello Objective-C!!"];
11
12  [myGreeter issueGreeting];
13  [myGreeter release];
14
15  [pool drain];
16  return 0;
17 }
```

第2行为Greeter类导入头文件。由于这个文件创建了Greeter的一个实例，它必须导入Greeter的头文件。

第8行分配Greeter类的一个实例，初始化该实例，并且，将指向它的指针存储到局部变量myGreeter中。查看右边最内层的消息：

```
[Greeter alloc]
```

这个消息表达式的接收者是类Greeter。Objective-C类自身就是对象，并且可以有方法，这种方法叫做类方法。alloc方法是一个类方法，它为Greeter的（未初始化的）实例分配存储，并且返回指向它的一个指针。随后，未初始化的Greeter实例变成了init消息的接收者。由于你没有覆盖init以提供自己的实现，因此将会调用NSObject对于init的实现。NSObject的init只是返回指向初始化的对象的一个指针。

第10行设置myGreeter的问候文本。指令：

```
@"Hello Objective-C"
```

告诉编译器，使用请求的文本来创建一个常量NSString对象。它变成了setGreeting:消息的参数。

注意 细心的读者可能会问，当这里的常量NSString对象在setGreeting:方法中碰到retain和release时，将会发生什么。答案是：什么也没有发生。它们对于一个常量字符串来说，不执行任何操作。那么，为什么还这么麻烦呢？因为对于将要传递给它什么类型的NSString对象，setGreeting:是毫不知晓的。如果在不同的程序中使用Greeter类，就可能会给它传递一个常规分配的NSString对象。如果想要该类正确地执行，就不应该对于它所接收到的输入类型做出假设。

第12行是实际操作。它通过调用Greeter的issueGreeting方法，使myGreeter真正地发布问候。

最后，第13行释放myGreeter中保存的对象。由于你创建了该对象（使用alloc），你要负责在用完它之后释放它。

构建和运行程序

再次打开Xcode的控制台(Run > Console或Shift-⌘-R)。构建该程序并运行它(Build > Build and Run或⌘-Return)。这时你应该会在控制台窗口中看到问候，如图4-7所示。

图4-7 运行HelloObjectiveC（第2版）的结果

4.5 小结

本章围绕一个非常小的程序介绍了很多。如果你已经使用Xcode完成并实现了HelloObjectiveC，那么现在应该掌握了构建和运行一个非GUI的Objective-C程序的技能了。你可以在本书后面的练习中将这些技能付诸应用。

本章结束了本书的第一部分，完成了对C的概览及对Objective-C的介绍。后面的部分将更

详细地介绍Objective-C语言。第5章介绍方法名称和声明、消息表达式，以及Objective-C消息系统是如何工作的。

4.6 练习

1. 修改本章中使用的HelloObjectiveC程序。
   - 定义一个Greeting类，使用NSString实例变量来保存问候文本及对问候的描述（类似的描述可能是“This is the everyday greeting.”或“This is the VIP greeting.”）。确保提供方法来设置和获取问候文本和问候的描述。将Greeting类的接口部分和实现部分的文件分别命名为Greeting.h和Greeting.m。即便你此时没有完全适应Objective-C，也应该能够看着Greeter类，用类比的方法来定义Greeting类。
   - 修改Greeter类，使用一个Greeting对象而不是一个NSString来进行第一次问候。将Greeting对象存储到一个实例变量中。为此，必须在Greeter.h文件的顶部放置如下的代码行：

     ```
     @class Greeting;
     ```

     还必须在Greeter.m文件的顶部添加如下代码行：

     ```
     #import "Greeting.h"
     ```

     这些代码行是必需的，以便编译器在编译Greeter类时知道Greeting类（见第6章）。
   - 修改HelloObjectiveC.m以分配Greeting的一个实例，并且将其传递给Greeter对象。
   - 构建并运行你的程序，验证其能正确地工作。
2. 扩展练习1中的程序。
   - 修改Greeter类，使它保存另一个Greeting对象。命名新的实例变量vipGreeting。确保提供方法来设置和获取新的变量。
   - 修改Greeter的issueGreeting方法，使它接受一个整数参数：

     ```
     -(void) issueGreeting:(int) whichGreeting;
     ```

     如果whichGreeting为0，则发布一条常规问候；如果它为1，则发布VIP问候。对于任何其他的值，输出一个错误。
   - 修改HelloObjectiveC.m以分配第二个Greeting对象，并且将其作为vipGreeting传递给Greeter对象。然后，添加一条语句，要求Greeter发布vipGreeting及最初的问候。
   - 构建并运行你的程序，验证其能正确地工作。

第二部分
语言基础

本书的第二部分将详细介绍Objective-C 2.0语言。前3章（第5章、第6章和第7章）介绍我们需要知道的Objective-C核心知识。余下的各章介绍要全面理解该语言必须学习的内容。当你学完了这一部分，应该是踏上了成为一名熟练的Objective-C程序员之路。

第二部分包括如下各章：

第5章　消息
第6章　类和对象
第7章　类对象
第8章　框架
第9章　常用Foundation类
第10章　Objective-C中的控制结构
第11章　分类、扩展和安全
第12章　特性
第13章　协议

第5章 消 息

在Objective-C中，当想要让一个对象做某些事情的时候，会给对象发送一条消息，作为响应，对象将执行自己的一个方法。这与调用函数并不完全相同。函数通常是静态绑定的，当程序被编译的时候，函数所执行的代码就确定了。而使用消息，为响应消息执行的实际代码在执行时才确定。

本章我们将学习Objective-C消息系统是如何工作的。同时将介绍方法、消息表达式和消息系统机制。

5.1 方法

Objective-C方法的结构与C函数的结构非常类似（实际上，编译器最终将方法转变为C函数）。二者之间有3个主要的区别：

- 方法在定义它们的类的一个特定实例上执行。方法接受指向对象实例的一个指针，其形式是隐藏变量self（“隐藏”意味着，即便没有在方法参数列表中或其他的地方声明变量self，你也可以在方法中使用它）。
- 方法可以直接访问对象的实例变量，而不需要声明它们。
- 命名方法及其参数的语法，与命名函数的语法不同。

5.1.1 一个简单的方法

假设编写一个绘图程序，并且有一个Shape类，参见如下的头文件：

```
@interface Shape : Graphic
{
  NSColor *outlineColor;
  NSColor *fillColor;
 ... // Other instance variables
}

... // Method declarations

@end
```

注意 NSColor是在AppKit框架中定义的一个类。对于本书中的示例，我们需要知道的唯一一件事情是，NSColor这个类表示在屏幕上绘制或向打印机绘制时所使用的颜色。

Shape应该提供了一个方法来返回对象的填充颜色。该方法应如下所示：

```
1 - (NSColor*) fillColor
2 {
3   return fillColor; // fillColor is an instance variable
4 }
```

来看看这个示例：

- 第1行：打头的连字符(-)表示这是一个实例方法。当这个方法用于消息表达式中时，接收者是Shape类的一个实例。以一个加号(+)开头的方法是类方法，它与一个接收者（类名）一起使用。第7章将介绍类方法。
- 第3行：一个方法的返回类型总是用圆括号括起来。这里，fillColor的返回类型是NSColor*，它是指向NSColor对象的一个指针，参见第1行。
- 第2行和第4行：方法的主体包含用花括号({ ... })括起来的一条或多条语句。

5.1.2 带有参数的方法

有时候，必须向方法传递信息，因此，需要方法接受参数。新手发现关于Objective-C最令人吃惊的事情之一是，方法名的构成方式。参数类型用圆括号括起来，并且参数名称也放入到了方法声明中，类型的前面有一个冒号。设置Shape的填充颜色的一个方法的声明应如下所示：

```
- (void) setFillColor:(NSColor*) newFillColor;
```

冒号是方法名称的一部分。这个方法的名称是：

```
setOutlineColor:
```

而不是：

```
setOutlineColor
```

如果方法有多个参数呢？在这种情况下，事情开始变得有些奇怪了。当一个方法有多个参数的时候，参数与方法名的各部分混杂在一起。例如，同时设置一个Shape的边框颜色和填充颜色的方法，其声明应如下所示：

```
- (void) setOutlineColor:(NSColor*) outlineColor
                fillColor:(NSColor*) fillColor;
```

上述方法的名称是：

```
setOutlineColor:fillColor:
```

注意，每个参数的前面都有个冒号。

方法的名称部分不是参数的标签，并且不允许将它们重新排列。

setOutlineColor:fillColor:和setFillColor:outlineColor:是不同的方法。尽管你刚开始可能会觉得这种命名方法很奇怪，但是，它可以使我们很容易写出具有描述性、不言自明的方法名称。

方法和函数一样，可以接受可变数目的参数。接受可变数目的参数的方法，如下这样声明：

```
- (void) methodWithVariableNumberOfArguments:(id) arg1, ...;
```

逗号和省略号不是名称的一部分。这个方法的名称是：

```
methodWithVariableNumberOfArguments:
```

方法名惯例

这不是Objective-C语言规范的一部分，但是，它几乎是Objective-C中通用的命名惯例，即每个方法名都以一个小写字母开头，并且，名称部分中任何后续的单词都是首字母大写的，如下所示：

```
// Good form
- (void) setOutlineColor:(NSColor*) outlineColor
         fillColor:(NSColor*) fillColor;
```

如下的方法名被认为是糟糕的命名，因为它没有遵从这一惯例：

```
// Bad form
- (void) SetOutlineColor:(NSColor*) outlineColor
         FillColor:(NSColor*) fillColor;
```

严格地讲，在冒号之前并不一定要有任何的名称字符。可以这样编写：

```
// Very bad form
- (void) setColors:(NSColor*)outlineColor
                :(NSColor*) fillColor;
```

上面的方法的名称是setColors::。

像setColors::这样的名称是合法的，但是，它们会使你的代码难以阅读。

5.2 消息

要求Objective-C对象执行自己的一个方法，这与调用一个函数并不是同样的事情。函数调用通常是静态绑定的，即在运行时所发生的那些事情，在编译时就已经确定了。当编译器遇到一个函数调用时，它插入一条指令，以跳转到构成函数体的代码。而Objective-C使用一种叫做消息的更为动态的系统。当想要Objective-C对象执行自己的一个方法时，就向该对象发送一条消息。对象通过执行与该消息对应的方法来做出响应。

消息表达式的形式如下：

```
[receiver message]
```

消息表达式用方括号括起来。receiver是将要执行方法的对象。message是想要让receiver执行的方法的名称，以及该方法接受的任何参数。

```
NSColor *newColor = ...
Shape  *aShape = ...
[aShape setFillColor: newColor];
```

在上面的例子中，消息是：

```
setFillColor: newColor
```

这是如何工作的呢？当执行一个消息表达式时，Objective-C运行时检查接收者并确定其类。运行时针对每个类都有一个表，列出了该类定义的方法，并且将每个方法与指向实现该方法的代码的一个指针关联起来。运行时从消息获取方法名，在接收者的类表中找到相对应的指针，

然后，使用该指针来执行方法的代码。本章稍后的5.4节，将详细讨论这一过程。

注意 经常使用短语“向aShape发送setFillColor:消息”可能有些繁琐。最终，人们（包括本书的作者）通常会较随意地说，“调用setFillColor:”。

什么是运行时？

做所有这些事情的运行时到底是什么呢？它一点也不神秘。运行时只是普通C函数和结构的一个共享库。在程序开始运行时，会调用一些这样的函数。这些函数根据编译器放置在你的可执行代码中的信息来构建表。在执行程序的不同时刻，还会调用其他的函数，来查找并操作这些表中的信息。这些函数调用都是编译器自动创建的。

多态

多态是不同类的对象响应同一消息的能力。假设你的绘制程序有一个Shape类，用于绘制线条图；还有一个Text类，用于绘制文本；还有一个Image类，用于绘制图像。类Shape、类Text和类Image，都实现了一个draw方法。尽管在任何情况下，方法名称都是相同的，都是draw，但是，每个类对于draw的实现是不同的。Shape的draw方法知道如何绘制图形，Text的draw方法知道如何绘制文本，而Image的draw方法则绘制图像。

绘制程序通常保存了一个列表，叫做显示列表，其中是要绘制的项。要在屏幕上绘制，程序循环遍历显示列表，依次绘制列表中的每一项。在下面的示例中，显示列表是Foundation类NSMutableArray的一个实例。假设它包含了Shape、Text和Image对象的一个混合体。这里提到的循环方法，将在第10章中介绍。对于这个示例，你只需要知道，枚举器是一个对象，每当它接收到一个nextObject消息，它就依次返回displayList中存储的对象。

```
1 NSMutableArray *displayList = ...
2 NSEnumerator *enumerator =[displayList objectEnumerator];
3 id graphic;
4
5 while (graphic = [enumerator nextObject] )
6 {
7   [graphic draw];
8 }
```

在第3行，变量graphic的类型是id，即通用的“指向对象的指针”的类型。这允许graphic保存任何类型的图形，包括nextObject方法所返回的Shape、Text或Image。

第5～8行中的循环貌似简单，但非常有用。它允许我们绘制不同种类的对象的一个列表，而不需要知道每个对象的类，或不需要做任何额外的工作。随着循环进行，变量graphic保存属于不同的类的不同对象，其中每一个类都以一种完全不同的方式实现了draw方法。每次执行第7行的时候，运行时都查看graphic中保存的对象的类，并执行该类的draw版本。用于这个循环中的类，不一定是一个通用超类的子类或者以任何其他方式与其相关的类。唯一的要求是，循环中的每一个类都实现了一个名为draw的方法。

多态使我们可以编写灵活的、稳定的代码。假设随后你决定要给绘制程序添加新的图形类，例如，一个BrushStroke类，你可以给程序添加这个新的类，而这对其他的代码产生的影响极小。只要新的BrushStroke类实现了一个draw方法，BrushStroke的实例就可以用于已有的显示列表和绘制循环，并且不用修改甚至不用重新编译代码。

谁发送消息？

通常来讲，“发送一条消息”的说法带有3件隐含的事情：一个发送者、一个接收者和一条消息，消息从发送者传送到接收者。你已经看到过接收者和消息，但是，谁是这其中的发送者呢？老实讲，类比在这里不适用。笼统地说，包含了消息表达式的函数或代码块就是发送者，但是，Objective-C消息语法中没有“sender”的形式表示。

初学者可能会搞混淆的一件事情是，GUI应用程序中的按钮、滑块条和其他控件使用AppKit或UIKit的方式。例如，当你配置一个按钮时，你将一个对象注册为按钮的目标。当用户单击该按钮的时候，按钮的对象发送给目标一条消息，目标对象执行为按钮设定的某些动作来响应单击。发送给目标的消息的形式如下：

```
- (void) somethingChanged:(id) sender
```

有一个单独的参数sender。该控件向其自身传递一个指针，用来作为sender的值。随后，被控制的对象使用这个值作为一个接收者，以将消息发送给控件来获取，例如，一个滑块条的数字值或者一个按钮的开关状态。注意，这只是AppKit和UIKit所设计的一种特殊的方式。sender是一个普通的方法参数，它并不是Objective-C消息系统的一个功能或一部分。参数名sender是一个惯例名称，但是，如果将其命名为widgetWhoseValueChanged也是可以的。

5.3 消息细节

现在，我们知道了基本的消息表达式是如何工作的，下面来看看它们如何在一些常见情形中使用。

5.3.1 嵌套

任何消息参数都可以用返回相应类型的一个消息表达式来替换。

```
1 Shape *shape1 = ...
2 Shape *shape2 = ...
3 NSColor *tmpFillColor;
4
5 // Set shape2's fill color to be the same as shape1's fill color
6 tmpFillColor = [shape1 fillColor];
7 [shape2 setFillColor: tmpFillColor];
```

上面的代码行和下面的代码行产生同样的结果：

```
Shape *shape1 = ...
Shape *shape2 = ...
```

```
// Set shape2's fill color to be the same as shape1's fill color
[shape2 setFillColor: [shape1 fillColor]];
```

像第一个例子一样使用一个中间变量，尽管程序更长一些，但是，在调试时，它更为有用。这样我们可以有机会在第7行（在第6行执行以后）设置一个断点，并且得到tmpFillColor的值，然后，可以使用调试程序查看其值而不必进入到方法fillColor中。

消息接收者也可以用返回相应对象的一个消息表达式来替代：

```
NSMutableArray *displayList = ...
id graphic;

// Draw the last object on the display list
graphic = [displayList lastObject];
[graphic draw];
```

可以编写为：

```
NSMutableArray *displayList = ...

// Draw the last object on the display list
[[displayList lastObject]  draw];
```

使用嵌套可以扩展到多个层级。计算嵌套的消息表达式的方法显而易见：先计算最内层的消息表达式，然后，使用其返回值替代该消息表达式。重复这一过程，直到计算完最外层的消息表达式。

当然，所有的方括号必须是匹配的（有经验的Lisp程序员会有宾至如归的感觉）。在编写嵌套多个层级的表达式时，一种常见的录入错误是，结束方括号的数目比开始方括号的数目多出一个或少一个。如果方括号不匹配，编译器将会报错并且拒绝编译代码。Xcode中使用的默认的文本编辑器，会帮助你解决这一问题，无论何时，当你输入一个结束方括号时，它会暂时突出显示所匹配的开始方括号。

5.3.2 向nil发送消息

按照Objective-C的设计，向一个nil接收者发送一条消息没有任何效果。当计算一个带有nil接收者的消息表达式时，没有什么方法会执行。程序默默地忽略nil接收者，而不是引发一个异常或崩溃，这允许我们编写整洁的代码。不必像下面这样搞乱了代码结构而保证自己不出错：

```
Shape *aShape = ...

if ( aShape!= nil )
{
  aShape draw];
}
```

而是可以直接写成：

```
[aShape draw];
```

如果aShape刚好是nil，就不会绘制什么内容，并且，从下一行代码开始继续执行。如果消息表达式中的方法的返回类型是一个指针或数字类型，则发送给nil的一条消息的返回值是0。对于其他的返回类型（例如，结构），在某些情况下，规则是复杂的并且返回的值是不确定的。

有些地方你可能需要使用一个nil接收者来引发一次异常，但是，在大多数情况下，避免守护的if语句将会使代码更整齐。如果在特定的情况下，捕获一个nil接收者很重要，那么你总是可以显式测试它。

> **为什么方法没有执行呢？**
>
> 如果你碰到了一个方法没有执行的问题，即你确信应该执行它，但是，程序忽略了该方法，那么，请检查接收者。在执行的时候，它可能是nil。这可能是由于你在自己的代码中的某处忘记初始化一个变量，从而导致的一个令人困惑的结果。

5.3.3 向self发送消息

如果想要在一个对象的方法中调用同一对象的另一个方法，必须使用变量self作为消息接收者。self是一个隐藏变量，编译器会将其传递给方法实现。它是一个指针，所指向的对象就是最初的方法调用的时候充当接收者的对象（在方法签名中，self没有显式表示为一个参数，但是，它总是可用的）。

```
- (void) someMethod
{
  [self someOtherMethod]; // Different method in the same class
}
```

这种模式通常与访问器方法一起使用，也就是设置和获取一个实例变量的值的方法。尽管方法能够直接访问一个对象的实例变量，但有些时候，还是应该使用实例变量的访问器方法来替代。如果访问器方法有副作用，例如，retain或release了存储在一个实例变量中的对象、限制了一个数字变量的范围或者执行某些簿记，那么，在一个方法中直接访问该变量则可以避免这些副作用。

当需要在多个方法中重复相同的功能时，self也很有用。可以将代码放入到一个“辅助”方法中，然后从其他的方法调用辅助方法（使用self作为接收者），而不是重复编写代码。

5.3.4 覆盖并向super发送消息

类可以通过在其@implementation部分提供一个不同的实现，来覆盖在其超类中定义的方法。当使用子类的一个实例作为接收者来调用覆盖的方法的时候，将会执行其子类对于该方法的实现。

例如，假设定义了一个AnnotatedShape类，它是Shape的一个子类。AnnotatedShape在其形状旁绘制了一个文本注解（对于图表任务来说，这是很有用的功能）。AnnotatedShape添加了一个实例变量来保存注解的文本，并且，它覆盖了Shape的一些方法。下面给出了AnnotatedShape的接口部分：

```
@interface AnnotatedShape : Shape
{
  NSString *annotationText;
}

- (NSRect) boundingBox;
- (void) draw;

...

@end
```

假设Shape有一个方法boundingBox，它会返回一个刚好包含该形状的矩形。一个智能的图形程序，在尝试绘制图形之前，将根据程序的窗口的矩形来测试这一矩形。如果图形的矩形位于窗口矩形之外，那么该程序将节省一些CPU周期并略过图形的绘制代码。AnnotatedShape覆盖了boundingBox，因为它必须返回包含注解的一个更大的矩形。

```
1 Shape *aShape = ...
2 AnnotatedShape *anAnnotatedShape = ...
3 NSRect shapeBounds;
4 NSRect annotatedShapeBounds;
5
6 shapeBounds = [aShape boundingBox];
7 annotatedShapeBounds = [anAnnotatedShape boundingBox];
```

在前面的代码中，当执行第6行的时候，将执行Shape的boundingBox实现。在第7行，执行的是AnnotatedShape的boundingBox实现。在某些情况下，子类需要覆盖一个方法以获得不同的行为。在前面的例子中，AnnotatedShape和Shape的boundingBox返回不同的矩形。在其他的情况下，覆盖的方法需要提供额外的行为。AnnotatedShape的draw方法必须像Shape的draw方法一样绘制形状，但是，它还必须绘制文本注解。一种方法是，将Shape的draw方法中的绘制代码复制到AnnotatedShape的draw方法中。然而，这不是一个好主意。不仅因为这需要增加程序的大小，而且因为它不可靠。因为如果需要修改Shape的绘制代码，则必须记住要在两个不同的地方进行修改。相反，应该像下面这样编写AnnotatedShape的draw方法：

```
1 - (void) draw
2 {
3     [super draw]; // Draw the shape
4
5   // Code to draw the annotation text
6   ...
7 }
```

当计算一个如下形式的表达式时：

```
[super message]
```

运行时使用当前对象（self指向的对象）作为一个真正的接收者，但是，执行message

中指定的方法的超类实现。在上面示例的第3行，真正的接收者是AnnotatedShape的一个实例，但是，执行的是Shape的draw方法的实现。AnnotatedShape的draw方法，使用Shape的draw方法实现中的代码，来绘制AnnotatedShape的Shape部分。然后，它继续绘制文本注解部分，然后才返回。

注意 使用self或super作为接收者，只是在这样的环境中才有意义：在一个方法中，有一个对象（self）表示将要执行的方法的对象。在普通C函数中，不能使用self或super。

self和super之间还有一个重要的区别：self是一个合法的指针变量，而super不是。self指向内存中的一个实际位置，它表示涉及的对象的一个实例。可以在一个方法调用中将其作为一个参数传递：

```
- (void) passItOn
{
 [someOtherObject doSomethingWithMe: self];
}
```

而另一方面，super不是一个真正的变量。它只是向运行时表示用self作为接收者，但是，要使用对象的超类中的该方法的实现，而不是该对象自己的方法实现。

注意 当通过super调用超类的一个方法的实现时，在超类的实现中，任何对self的引用仍然指向最初的接收者，即子类的一个实例。如果超类实现继续调用另一个类的方法，并且，该方法被子类所覆盖了，那么，将会调用子类的实现。

5.3.5 选择器

一条消息的方法名部分有时候叫做选择器或方法选择器，因为，运行时使用它来选择要执行哪一个接收者方法。选择器只是名称，它们不带有任何类型的信息。

Objective-C定义了一个类型，SEL，来保存选择器的表示。SEL与选择器名称有一对一的关系，但是，一个SEL自身并非一个字符串。带有相同名称的所有选择器，具有相同的SEL，并且，不同名称总是对应不同的SEL。在内部，为了提高效率，Objective-C使用SEL类型来表示方法。使用字符串将会很慢，因为仅仅测试两个字符串是否相同，就需要遍历字符串中的所有字符。

注意 当前，一个方法的SEL定义为指向一个独特结构的指针，该结构保存了方法名字符串。然而，这是一个实现的细节，而不是你应该使用的方式。在未来的版本中，这可能会有变化。

编译器指令@selector()将一个方法名称转换为一个SEL：

```
SEL aSelector = @selector( someMessageName );
```

如果该方法没有任何参数，那么在创建一个SEL时，必须记住包含冒号。冒号是方法名的一部分。

```
@selector( aMethod )
```

和

```
@selector( aMethod: )
```

是不同的SEL。

SEL可以用来作为消息表达式语法中的消息部分。NSObject定义了如下的方法：

```
- (id)performSelector:(SEL)aSelector
```

当执行performSelector:时，该对象执行的选择器是其参数。如下的两行代码是等价的：

```
[aShape draw];
[aShape performSelector: @selector( draw )];
```

以一种较长的方式来编写同样的事情，不是很有用；performSelector:真正有用之处在于，当它的参数是一个变量的时候。例如，当用户从绘制程序中选择一个对象，绘制该对象的程序突出显示它以表示选中。如果graphic对象有一个不同的方法，drawHighlighted，用来在选中的状态下绘制自身，那么，绘制代码可能如下这样编写：

```
BOOL isHighlightDraw = ...
SEL  shapeDrawingSelector;

if ( isHighlightDraw )
  {
 shapeDrawingSelector = @selector( drawHighlighted );
  }
else
  {
 shapeDrawingSelector = @selector( draw );
  }

...

Shape *aShape = ...

[aShape performSelector: shapeDrawingSelector];
```

NSObject将performSelector:withObject:和performSelector:withObject:withObject:定义为接受一个或两个对象参数的方法。它们与performSelector:以同样的方式工作，只不过它们通过选择器参数，向指定的方法传入了它们的额外参数。

5.3.6 具有相同名称的方法

如果使用动态类型（变量的类型为id），则具有相同名称的所有方法应该拥有同样的参数类型和返回值，即便它们由不相关的类声明。如果违反了这一规则，将会得到编译器的警告；如果忽略了这一警告，你的程序可能会有非常细微的bug。

例如，假设有一个类ClassWithInt，它声明了如下的方法：

```
- (void) setValue:(int) newInt;
```

还有另一个类ClassWithFloat，它声明了具有相同名称的另一个方法，但是该方法具有不同的参数类型：

```
- (void) setValue:(float) newFloat;
```

然后，通过编写如下代码，将两个类的头文件都导入：

```
1 // DON'T DO THIS
2 #import "ClassWithFloat.h"
3 #import "ClassWithInt.h"
4
5 id numberObj = [[ClassWithInt alloc] init];
6 [numberObj setValue: 21];
```

编译器将会发出警告：

```
warning: also found '-(void)setValue:(int)newInt'
warning: using '-(BOOL)setValue:(float)newFloat'
warning: multiple methods named '-setValue:' found
```

如果你毫不在意继续向前并运行该程序，将会遇到麻烦。当执行第6行的时候，numberObj包含了ClassWithInt的一个实例，并且执行的（如同期待的）是ClassWithInt的setValue:实现。但是，方法在newInt参数中接受的值是1101529088，而不是21。

发生这个问题，是因为编译器现在对于方法名setValue:有两组不同的类型信息。由于numberObj的类型是id，没有什么方法能够告诉编译器，第6行中的参数应该是int还是float。编译器首先遇到ClassWithFloat.h，因此，它选择float。当执行第6行代码的时候，参数21（一个整数）提升为float。float的位传递给了该方法的ClassWithInt实现，它随后将它们解释为int。这只是一个隐含的bug，而不是崩溃。你起初可能不会注意到修改了值，并且，bug的出现也完全是由不相干的事情所控制的，即import语句的顺序。

注意 与C++不同，Objective-C没有方法重载。C++类可以拥有具有相同名称的方法，并且通过不同的参数类型或返回值来区分。而这在Objective-C中是不允许的。

5.3.7 动态类型和静态类型

类型为id的变量声明了它保存的是指向一个对象的指针，但是，它没有指明该对象属于什么类。给类型为id的变量，赋值为指向任何类的对象的一个指针，都是合法的。尽管在你想要利用多态的优点时，这是很有用的，但它还是有缺点，就是没有了编译器类型检查。如果不需要id的灵活性，那么可以使用静态类型，并且编译器将帮助对你的赋值进行类型检查：

```
Shape *aShape = [[Shape alloc] init];
```

声明为Shape*的变量，可以保存一个指针，它指向类Shape的一个对象或者从Shape派生的任何子类的对象。

```
Shape *aShape = [[AnnotatedShape alloc] init];   // OK
```

反之则不然，不能将一个超类的实例赋值给类型为指向其一个子类的指针的变量：

```
AnnotatedShape *aShape = [[Shape alloc] init];   // NOT OK!
```

上面的这行代码将会产生如下的警告消息：

```
warning: incompatible Objective-C types assigning 'struct Shape *'
expected 'struct AnnotatedShape *'
```

忽略该警告，并向超类发送一条仅在子类中定义了的消息，将会导致一次崩溃。

使用静态类型也很容易，规则是，具有相同名称的所有方法必须具有同样的参数类型和返回值。如果你不打算将对象赋值给类型为id的变量，这条规则就变成了“不同的类定义的两个方法，可能具有相同的名称，但是参数类型或返回类型不同，只要定义类中没有一个是另一个的子类”。然而，这么做也有混淆的可能，最好是避免这么做。

如果不需要id的灵活性，应该使用静态类型并且让编译器为你做类型检查。

5.4 幕后工作

当执行如下代码行的时候，对象anObject接收了doSomething消息，并且通过执行其名为doSomething的方法来响应。

```
id anObject;
[anObject doSomething];
```

它如何知道怎么做这些事情呢？当anObject可能在不同的时刻是不同的类的实例，而每个类拥有不同的doSomething实现的时候，如何知道实际执行哪一段代码呢？简短的答案是，“你不需要知道，它就是有效”。但是，对于那些不喜欢神秘性的人们来说，下面将说明Objective-C消息系统是如何工作的。

当计算一个消息表达式时，运行时查看接收者并确定接收者的类。然后，它查看这个类的类信息，以找到与消息中的方法名对应的方法。最后，它执行该方法。

具体细节如下：

- 所有的Objective-C对象知道它们的类。要完成这一任务，编译器给每个对象实例添加一个实例变量，名为isa；isa说明一个实例是什么类型的对象。在运行时，isa指向类的类结构，这是一种不透明的类型，其中包含了有关类的信息；包括该类所定义的实例变量和方法；并且还有一个指针，指向关于其超类的类信息。当启动程序时，这些信息在运行时填充。
- 编译器将方法转换为实际编译的C函数。在这一过程中，它在参数列表的开始处添加两个额外的参数，self和_cmd。self是一个指针，指向发送消息时充当接收者的对象；_cmd是与消息的方法名称对应的选择器。

考虑如下的Objective-C方法：

```
-(void) doSomethingWithArg:(int) anArg
{
 // ...
}
```

在编译过程中，编译器将doSomethingWithArg:转换为一个C函数，它带有如下的参数

列表：

```
( id self, SEL _cmd, int anArg )
```

- 每个Objective-C类都有一个表，这个表将一个方法的选择器与指向实现该方法的函数的一个指针连接起来。给定了选择器，运行时可以使用这个表来找到匹配的函数的指针。每个类也有一个相关的缓存机制，这加速了对一个给定方法的后续搜索。

现在，可以搞清楚从一个消息表达式到一个实际函数的顺序了。当编译器遇到如下所示的一个消息表达式时：

```
Shape *aShape = ...
NSColor *newFillColor = ...

[aShape setFillColor: newFillColor];
```

它使用对运行时函数objc_msgSend的一次调用来替代消息表达式：

```
objc_msgSend( aShape, @selector( setFillColor: ), newFillColor );
```

objc_msgSend的第一个参数是一个接收者对象。第二个参数是方法名称，转换为SEL。剩下的参数（如果有的话），是要传递给该方法的参数。

当在运行时执行对objc_msgSend的调用时：

1）objc_msgSend查找存储在aShape的isa变量中的指针，并且，顺着该指针找到Shape类的类结构。

2）然后，它查看Shape方法的缓存，看看它是否能够快速找到指向函数实现setFillColor:的一个指针。

3）如果setFillColor:的函数指针不在缓存中，objc_msgSend查看Shape类的方法表，查找函数指针。

4）在有了函数指针以后，运行时带参数调用该函数：

```
( aShape, @selector( setFillColor: ), newFillColor )
```

每个类结构还包含指向其超类的类结构的一个指针。如果运行时在类结构缓存及其方法表中都没有找到选择器，那么它将沿着指向类的超类的类结构的指针，重复这一查找。继续这一过程，直到运行时找到一个匹配的函数指针，或者它到达了对象层级的顶端，并且查完了所有查看的地方。后一种情况将在5.5节中介绍。

Objective-C 2.0的运行时变化

在Objective-C 2.0的运行时实现中，有一些主要的变化。变化之一是，在Objective-C之前的版本中定义的用来表示类和方法的运行时信息的结构，已经用不透明类型及操作这些类型的一组函数替代了。这种类型其自身基本上是一种基本对象系统，直接在C中实现。从功能上讲，它和以前一样工作，但是，你不用再深入其中或直接操作结构。

5.5 消息转发

如果objc_msgSend到达了对象层级的顶端而没有找到与消息中的选择器对应的方法实现，

则会发生什么事情呢？如果遇到这种情况，objc_msgSend使用初始消息表达式所形成的NSInvocation参数（参见下面关于NSInvocation的介绍），来调用NSObject的forwardInvocation:方法。NSObject的forwardInvocation:实现调用了NSObject方法doesNotRecognizeSelector:。

NSObject的doesNotRecognizeSelector:实现引发一个异常，并且输出一条错误消息，然后，程序崩溃。

但是，不一定必须按这种方式进行。你可以覆盖forwardInvocation:，然后使用调用参数中包含的信息，向一个不同的对象发送该消息，而该对象能够正确地处理该消息。这叫做转发。转发可以用来从一个不同的对象借用行为。最初的对象处理某些消息，但是，其他的消息传递给一个辅助对象。

> **NSInvocation**
>
> NSInvocation是封装为对象的一个Objective-C消息表达式。NSInvocation对象的实例变量保存了一个对象（最初表达式的接收者，在这里叫做目标），还有发送给该对象的一条消息（以一个选择器的形式，以及该方法接收的任何参数）。它可以存储起来，以待后用，或者像任何其他对象一样作为参数传递给一个方法。
>
> 当需要执行该调用所封装的消息表达式时，向NSInvocation对象发送invoke消息：
>
> ```
> NSInvocation* anInvocation;
> [anInvocation invoke];
> ```
>
> NSInvocation类定义了用来设置和获取接收者、选择器及封装的消息表达式的参数的方法，以及用来获取返回值的方法。第16章将介绍NSInvocation。

5.6 效率

你可能会对自己说："所有这些查找工作都比直接函数调用要慢。"是这样的，运行时代码为了让方法查找变得尽可能快，已经取得了很大的进展，但是，它仍然比一次C函数调用要慢。然而，这通常不会有太大的实际差异。几乎在每种情况下，执行一个方法体所需的时间比发送消息所需的时间长太多了。在极少数情况下，发送消息的负载可能会是一个问题。在一个密集的循环中，重复做某件事情很多次，这是很常见的情况。如果你发现这样的情况，可以使用NSObject的methodForSelector:方法来获取指向实现该方法的C函数的一个指针：

```
IMP anIMP = [aShape methodForSelector: @selector( draw )];
```

记得在第3章，IMP定义为指向一个函数的指针。当拥有了函数指针后，可以使用它来调用函数，如下所示：

```
(*anIMP) (aShape, @selector( draw ));
```

还记得吗？当编译器将你的方法转换为C函数的时候，它在参数列表的前面添加了self和_cmd参数。当通过其IMP调用函数时，必须记得填充这些参数。如果该方法接受参数，那么这些参数在函数调用中跟在self和_cmd的后面。

直接函数调用有多快呢？下面的代码执行testMethod 100万次，首先使用常规的方法发送，然后，通过直接调用实现了testMethod的函数：

```
int j;
// Loop 1, regular message dispatch

for ( j=0; j < 1000000; j++ )
 {
  [testObject testMethod];
 }
IMP testIMP = [testObject methodForSelector: @selector( testMethod )];

// Loop 2, calling the function with its IMP

for ( j=0; j < 1000000; j++ )
 {
  (*testIMP) (testObject, @selector( testMethod ) );
 }
```

如果testMethod只包含一条return语句，则两个循环的执行时间的比率是：[⊖]

```
(time for Loop 1) / (time for Loop2 )  ~ 2.63
```

直接调用函数的速度几乎是消息发送的2.5倍还多一点。真是有很大的改进！遗憾的是，仅仅返回值的方法不是很有用。如果testMethod包含4个浮点数乘法指令，则比率是~1.25；如果testMethod有一个单独的sin()计算，则比率是~1.04。你不应该太在意具体的数字，但是，这一趋势很明显。只要你开始在方法中做一些实际的事情，执行方法代码所花费的时间，就会远远超过发送方法所需的时间。

注意 当获取方法的IMP并直接调用它时，有一件事情必须要注意。IMP使用如下的typedef来定义：

```
typedef id  (*IMP)(id, SEL, ...);
```

这个typedef假设所有的方法的参数及其返回值要么是指针，要么是整数类型。如果方法有浮点参数或者浮点返回值，那么必须接受返回的IMP并且将其强制转换为正确的函数指针类型。如果不这么做，编译器将不能正确地为函数调用建立栈帧，并且，函数会接受或返回垃圾信息。

对于如下的函数声明：

```
- (void) setLinewidth: (float) input;
```

正确的调用代码如下所示：

```
void (*functionPtr)( id, SEL, float) =
    (void (*)( id, SEL, float))[aShape methodForSelector:
        @selector(setLinewidth:)];
```

⊖ 所用时间是2009年秋季在一台使用3.06 GHz Core 2 Duo处理器的iMac上测试的，该机器运行Mac OS X Snow Leopard (version 10.6.2)。

```
// Set line width to 1.5
*functionPtr( aShape, @selector(setLinewidth:), 1.5 );
```

函数指针声明的详细内容将在第16章介绍。

5.7 内省和其他运行时乐趣

Objective-C的运行时支持在运行时发现对象的各种属性。这个过程叫做内省。可以使用class和superclass方法来找到一个对象的类和超类：

```
Class objectsClass;
Class objectsSuperClass;
id anObject;

objectsClass = [anObject class];
objectsSuperClass = [anObject superclass];
```

有时候，无法确保一个对象可以响应一条给定的消息。在那些情况下，在发送消息之前，可以使用方法respondsToSelector:检查对象是否会响应一条消息。例如，一些类声称它们符合一个特定的协议。

注意 协议是一组定义的方法。当一个类遵从一个协议时，它承诺实现集合中必需的所有方法。第13章将介绍协议。

然而，协议中的方法可以声明为可选的。遵从一个协议的类可以自由地实现或不实现一个可选的方法。如果接收者的类没有实现可选的方法，就不能调用它。这么做可能会引发程序崩溃。可以使用respondsToSelector:来避免这一问题，如下面的代码所示：

```
id anObject = ...  // object that conforms to a protocol
SEL optionalSelector = @selector( optionalMethod );

if ( [anObject respondsToSelector: optionalSelector] )
  {
    [anObject optionalMethod];
  }
```

class、superclass和respondsToSelector: 是NSObject定义的方法，并且由从NSObject派生的所有对象继承。

除了这些例子，可以直接调用C语言运行时函数，以执行一些有趣的（并且有时候是危险的）任务，例如获取类的实例变量或方法的列表，或者在运行时使用不同的实现来替代一个方法实现。这里不打算介绍这些内容，但是，如果你感兴趣或想要探究，可以查看Objective-C 2.0 Runtime Reference，相关网址为http://developer.apple.com/mac/library/ documentation/Cocoa/Reference/ObjCRuntimeRef/Reference/reference.html。

注意 如果你感到好奇，并且真的想知道运行时机制是怎样的，那么运行时是开源的，其代码可以从www.opensource.apple.com/source/objc4/objc4-437.1/下载（但是提前警告一下，这些代码并不容易阅读）。

Objective-C消息系统的功能很强大，并且，它使得我们很容易构建灵活的、创新的软件。这也是有一定代价的，它限制了编译器执行某些有效性检查的能力。后果是一些错误可能会在运行时出现。这使一部分人感到很烦恼，他们认为这很危险，并且更喜欢严格的类型检查，就像在C++这样的语言中所见到的那样。对这些人来说，我想要做如下的一个类比：一把制作精良的8英寸到12英寸的厨刀也很危险，如果不小心地使用或充满恶意地使用的话，甚至能置人于死地。但是，如果你因此就不准备使用它的话，制作精美的食物也就不太可能了。

5.8 小结

Objective-C方法通过消息调用，而不是通过直接函数调用。消息表达式包含一个接收者（将要调用其方法的对象）和一个消息（要执行的方法的名称，还有方法接收的任何参数），它们都包括在一对方括号中。当计算一个消息表达式时，运行时确定了接收者的类，然后查看该类的类结构，以找到指向实现该方法的代码的一个指针。然后，运行时填充隐藏变量self（指向接收者的一个指针）和_cmd（方法的名称，它会转换为一个SEL），并且使用该指针来执行代码。

这种安排有几个优点。首先，消息表达式的两个部分都是动态的，它们可以在运行时而不是编译时确定。而且它还考虑到了完全的多态。其次，如果接收者的类型是id（指向对象的指针），则它可以保存任何类型的对象，只要对象实现了名称与消息中的名称一致的方法即可。当需要从不同类型的对象集合中提取同一类型的行为时，需要用到多态，例如，要求一个不同类型的对象集合中的对象绘制自身。

关键字self和super用于方法中，它们分别用来调用其类中或其超类中的其他方法。

Objective-C消息系统的灵活性带来了一些性能上的代价。对于时间性重要的代码来说，也可能绕过消息系统而直接调用方法，尽管在实际情况中，很少需要这么做。消息发送所花的时间，通常比执行方法体所花费的时间要少很多。

5.9 练习

1. 修改在第4章的练习2中创建的HelloObjectiveC程序版本。
   - 给Greeter类添加一个方法，同时设置常规的问候和VIP问候。不要忘了保留greeting和vipGreeting的新值，并且以正确的顺序释放旧的值。
   - 修改HelloObjectiveC.m以使用新方法。
   - 构建并运行你的程序，以验证它能够正确工作。
2. 修改练习1中创建的新方法。
   - 使用关键字self和已有的setter方法setGreeting和setVIPGreeting来设置新值，而不是直接设置实例变量greeting和vipGreeting。
   - 构建并运行你的程序，以验证它能够正确工作。

   注意，当你使用已有的setter方法来编写新方法时，它们自动为你进行内存管理。

3. 练习使用选择器：
   - 在HelloObjectiveC.m中，将如下的消息表达式：

     ```
     [myGreeter issueGreeting];
     ```

     替换为使用performSelector:的一个消息表达式。
   - 构建并运行你的程序，以验证它仍然能够正确工作。

     在这个例子中，使用performSelector:并没有什么真正的原因。这个练习的要点是确保你理解如何获取和使用一个选择器。
4. 验证向nil发送一条消息会什么事情也不做（并且，特别是，它不会引发程序崩溃）。
   - 向HelloObjectiveC.m添加另一个Greeter*类型的变量：

     ```
     Greeter* greeter2;
     ```
   - 给greeter2赋值为nil。
   - 使用greeter2作为一条issueGreeting消息的接收者。
   - 构建并运行你的程序，验证它按预期的那样工作。
5. C++允许方法重载。一个C++类可以定义两个（或更多的）具有相同名称的方法，而且，通过它们拥有的不同类型的参数和返回值来区分它们。使用你学习的Objective-C消息发送系统如何工作的知识，说明为什么在Objective-C中实现C++样式的重载会很困难。

第6章 类和对象

对象是面向对象编程的基础。本章将深入到产生对象的过程。本章划分为5个主要部分：

- 定义类
- 通过子类化来扩展类
- 创建对象（类的实例）
- 销毁对象
- 复制对象

一些主题在第3章中已有简单提及。在这里，将更为详细地介绍它们。

6.1 定义类

当定义一个Objective-C类时，必须为编译器提供两组信息。第一组是构建类的实例的一个基本的蓝图。必须指定类名、类的超类、类的实例变量的名称和类型的列表，最后是类的方法的声明。第二组信息是实现类的方法的代码。第一组信息放入到类的接口部分。第二组信息放入到类的实现部分。

这两部分通常放在不同的文件中，文件的名称都以类名来命名。例如，如果定义一个Guitar类，应该将接口部分放入到一个名为Guitar.h的头文件中，而将实现部分放入到一个名为Guitar.m的文件中。

6.1.1 接口部分

假设我们要为一个模拟游戏创建一个RockStar类。一个RockStar实例完成摇滚歌星通常所做的所有的事情：唱歌、弹吉他、搞乱酒店的房间等。RockStar的接口部分如下所示：

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

@class Band;
@class Guitar;

@interface RockStar : NSObject

{
  NSString *name;
  Band *band;
  Guitar *guitar;

  // More instance variables
}
```

```
- (void) sing;
- (void) playGuitar;
- (void) destroyHotelRoom:(float) damageLevel;

// More method declarations

@end
```

接口部分包括：

- 一个@interface编译器指令，后面跟着要定义的类的名称（在这个例子中是RockStar），一个冒号，以及该类的超类（在这个例子中是NSObject）。
- 类的实例变量的声明。实例变量声明给出了实例变量的类型和名称，以一个分号结束。实例变量部分用一对花括号括起来。如果该类没有定义任何实例变量，则可以省略花括号。
- 方法声明的列表。方法声明必须放在实例变量声明的后面。

  声明以一个连字符(-)开始的方法，是实例方法；它们与作为接收者的一个类的实例一起使用。声明以一个加号(+)开始的方法，是类方法；它们与作为接收者的一个类的类名一起使用。第7章将介绍类方法。
- 最终的@end指令用来结束这一部分。

编译器需要有关类的超类的一些信息，因此，类的@interface部分必须导入包含了其超类的接口部分的头文件。在这个例子中，RockStar的超类是NSObject，这是一个Foundation框架类。导入Foundation/Foundation.h，将会自动导入超类头文件NSObject.h。

6.1.2 @class指令

如果在一个实例变量声明或一个方法声明中使用一个类类型，例如Band或Guitar，那么必须通知编译器，该类型是一个类。如果该类的接口在任何一个导入的头文件（一个Cocoa框架头文件、超类头文件，或者反过来，它们导入的任何头文件）中定义了，就不必再做任何事情了。否则，需要使用一条@class指令。

RockStar类有类型为Band和Guitar的实例变量。但是，RockStar.h没有导入Band.h或Guitar.h，因此，需要在文件中放置如下两行代码：

```
@class Band;
@class Guitar;
```

@class指令必须放在任何要使用的类型的声明之前，这是前向声明的一个例子。在这个例子中，@class Band;告诉编译器，Band是一个类，并且类型为Band*的内容就是指向一个类的指针，但是，它没有提供关于Band的任何其他信息。从技术上讲，在很多情况下，我们可以只是直接导入相关的头文件，让编译器知道该类型。但是，使用一条@class指令是更好的设计选择：

- 使用@class指令可以通过减少依赖性而缩短编译时间。假设把Band.h导入到RockStar.h中，如果随后对Band.h做出了修改，那么，必须重新编译导入RockStar.h的所有文件，以及导入Band.h的所有文件。如果使用了@class Band;，

相反，只需要重新编译导入Band.h中的文件。

- 在一些很少的情况下，使用@class指令是必需的。假设Band类有一个leadGuitarist实例变量，其类型就是一个RockStar*：

你现在遇到一种情况，RockStar.h需要知道Band是一个类，并且Band.h需要知道RockStar是一个类。如果试图通过将Band.h导入到RockStar.h，并且将RockStar.h导入到Band.h，则代码将无法编译。必须至少在两个头文件的一个之中使用一条@class指令。

6.1.3 实现部分

实现部分包含了实现类的方法的代码。实现部分的结构非常简单。它以包含一条@implementation指令和类名的一行开始，以一条@end指令结束。所有的方法实现都放置在这两行之间。

如下是RockStar类的实现文件：

```
#import "RockStar.h"
#import "Guitar.h"
@implementation RockStar

-(void) playGuitar
{
  [guitar tune];

  ...
}

// More method implementations

@end
```

如果在类的接口部分声明了一个方法，然后，忘记了提供该方法的实现，编译器将警告你忘记了实现。如果忘记了实现destroyHotelRoom:，那么在编译RockStar.m时，将会得到如下的警告：

```
warning: incomplete implementation of class RockStar
warning: method definition for '-destroyHotelRoom:' not found
```

如果忽略了警告，并且试图调用不存在的方法，程序将会崩溃。

6.1.4 导入

类的实现文件导入了包含相应的接口部分的头文件。它还必须为实现将要使用的任何其他类导入头文件。在RockStar的示例中，playGuitar的实现调用了Guitar的tune方法，因此，为了避免编译器的警告，RockStar.m必须导入Guitar.h。即便RockStar.h有一个针对Guitar的@class指令，也是如此。编译器需要查看Guitar.h，以便找到tune方法的方法声明。

如果没有导入Guitar.h，只要Guitar类是Xcode项目的一部分，并且Guitar实际上实

现了tune，代码在运行时也能工作。但是，编译器会为了程序的健全而提出警告，并且，养成忽略编译器的警告的习惯就是朝开发问题软件的道路迈出了一大步。

接口部分和实现部分放入不同的文件中

类的接口部分放在一个单独的头文件中，以便接口部分（它提供了关于类的编译器信息）可以导入到使用该类的代码中。例如，如果想要创建RockStar的一个实例，调用RockStar的一个方法，或声明RockStar的子类，你的代码就必须导入RockStar.h。如果接口部分和实现部分在同一个文件中，则导入接口部分也会导入实现了该类的代码，这会导致实现代码的多个副本。这么做会导致编译器错误，编译器会给出一条关于重复符号的提示。

在一种情况下，将类定义的两个部分都放入到同一文件中是没有问题的。如果你有一个辅助类，它只在另一个类中使用，则可以将辅助类的两个部分都放入到主类的实现文件中。辅助类的接口部分应该放在辅助类和主类的实现部分之前。

6.2 子类化一个类

如果每次编写一个程序时，都必须从头开始，那将是很耗费时间的（更别提多让人烦）。像Objective-C这样的面向对象语言，允许我们重用代码而免去重复编码的烦恼。可以通过修改已有的类，或者向其添加内容，来创建新的类。尽管我们可以在任何语言中在源代码层级重用代码，但是这里要强调的是，面向对象语言允许在已有的类之上构建，即便你没有最初类的源代码。

如下是在讨论这一主题时所使用的一些术语：

- 通过扩展已有的类来定义的一个类，叫做初始类的子类。
- 已有的类是新类的超类。
- 除了6.2.1节中的修改提示，子类与其超类具有相同的实例变量和方法。我们说子类继承了超类的实例变量和方法。

6.2.1 定义一个子类

子类可以用如下三种方式中的一种或多种来扩展或修改一个已有的类：

1) 子类可以添加其超类中没有定义的实例变量。然而，子类不能删除其超类定义的实例变量。

2) 子类可以添加其超类中没有定义的方法。

3) 子类可以对其超类中声明的一个方法提供自己的实现。这叫做覆盖一个方法。

注意 要保证技术上的正确，子类不一定必须做前面3件事情中的任何一件。但是，不添加实例变量或方法，或不覆盖一个方法，子类将是毫无意义的。它将只不过是其超类的一个别名而已。

6.2.2 子类示例

本节将为第3章中介绍的Accumulator类定义一个子类。初始的Accumulator类继续计算总和，并有方法向总和加值、报告当前总和，以及将总和重置为0。新的类NamedAccumulator对初始类Accumulator做出了如下的修改：

- NamedAccumulator能够存储一个名称。
- NamedAccumulator拥有方法来设置和获取名称。
- NamedAccumulator覆盖了Accumulator的addToTotal:方法的版本。新的addToTotal:方法记录了每次相加。

为了帮助你回忆，程序清单6.1和程序清单6.2给出了最初的Accumulator类的代码。

程序清单6.1 Accumulator.h

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

@interface Accumulator : NSObject
{
  int total;
}

- (void) addToTotal:(int) amount;
- (int) total;
- (void) zeroTotal;

@end
```

程序清单6.2 Accumulator.m

```
#import "Accumulator.h"

@implementation Accumulator

- (void) addToTotal:(int) amount
{
  total = total + amount;
}

 (int) total
{
  return total;
}

- (void) zeroTotal
{
  total = 0;
}

@end
```

程序清单6.3给出了子类NamedAccumulator的接口文件。

程序清单6.3 NamedAccumulator.h

```
1 #import <Foundation/Foundation.h>
2 #import "Accumulator.h"
3
4 @interface NamedAccumulator : Accumulator
5 {
6   NSString *name;
7 }
8
9 - (NSString*) name;
10 - (void) setName:(NSString*) newName;
11 - (void) addToTotal:(int) amount;
12
13 @end
```

在程序清单6.3中：

- 第2行：一个子类接口文件导入其超类的头文件。要创建一个子类，编译器必须查看其超类的定义。
- 第4行：在@interface行中，冒号之前的名称是子类；冒号后的名称是超类。
- 第6行：子类的实例变量部分只列出了子类对于初始类添加的实例变量。这里，NamedAccumulator只添加了一个新的实例变量，即name。
- 第9～10行：这些是NamedAccumulator定义的新方法；它们获取和设置name实例变量。
- 第11行：NamedAccumulator覆盖了addToTotal:。在子类的接口部分重新声明了一个覆盖的方法，这不是必需的，但是，出于文档化的目的，这是一种很好的做法。这告诉子类的用户，该子类为该方法提供了其自己的版本。不需要重新声明那些继承了而没有覆盖的方法。

子类的实现文件只包含了子类所添加的或覆盖的那些方法的实现。程序清单6.4给出了NamedAccumulator的实现文件。

程序清单6.4 NamedAccumulator.m

```
1 #import "NamedAccumulator.h"
2
3 @implementation NamedAccumulator
4
5 - (void) setName:(NSString*) newName
6 {
7  [newName retain];
8  [name release];
9   name = newName;
10 }
11
```

```
12 - (NSString*) name
13 {
14   return name;
15 }
16
17 - (void) addToTotal:(int) amount
18 {
19   NSLog(@"Adding %d to the %@ Accumulator", amount, [self name] );
20   [super addToTotal: amount];
21 }
22
23 - (void) dealloc
24 {
25   [name release];
26   [super dealloc];
27 }
28
29 @end
```

在程序清单6.4中：

- 第1行：和任何其他的类定义一样，子类的实现文件导入了其对应的接口文件。
- 第5～10行和第12～15行：这些是设置和获取NamedAccumulator的name实例变量的方法的实现。
- 第17～21行：这是addToTotal:的新版本。它记录了事务，然后，调用addToTotal:的超类(Accumulator)版本进行实际的加法。这里给出的模式，是子类化中非常常见的一种模式。当一个子类需要为超类的方法添加行为时，它覆盖该方法。新的实现编写了新的行为，然后，使用关键字super调用该方法的超类版本。
- 第23～27行：当对象的引用计数减少到0时，调用其dealloc方法，让对象在返回给堆之前能够有机会执行任何必需的清理工作。Accumulator对象没有任何保存对象的实例变量，因此，它不需要任何清理。Accumulator继承自NSObject的dealloc方法就足够了。

NamedAccumulator不同，它在实例变量name中保存了一个NSString对象。当保存在name中的对象赋值给name时就保留了，在NamedAccumulator销毁之前，必须使用一个释放来对应该保留。为此，NamedAccumulator覆盖了dealloc。在NamedAccumulator的dealloc方法中，它释放了name中的对象，然后，调用dealloc的超类实现。本章稍后还将更详细地介绍dealloc方法。

注意 NamedAccumulator类的定义并没有对其超类Accumulator的源代码做任何引用。当创建一个子类时，唯一所需超类的信息，就是超类的名称及超类的头文件。不需要超类的源代码（当然，在连接程序时，要有超类的对象代码）。

现在，可以让NamedAccumulator类工作了，如下面的代码段所示：

```
NamedAccumulator* ac = [[NamedAccumulator alloc] init];

[ac zeroTotal];
[ac setName: @"Bandersnatch"];

[ac addToTotal: 100];
[ac addToTotal: 37];

NSLog( @"The total in the %@ Accumulator is... %d",
        [ac name],[ac total] );
```

执行前面的代码，将产生如下输出：

```
Adding 100 to the Bandersnatch Accumulator
Adding 37 to the Bandersnatch Accumulator
The total in the Bandersnatch Accumulator is... 137
```

6.2.3 类继承

重复的子类化结果产生了类的一个链，每一个类都在其前一个类的基础上构建，这叫做类层级。比类层级中给定的类出现得更早的类，叫做祖先类（这个术语并不常用，但是，也可以将一个类的超类称为其直接祖先类）。位于类层级顶部的类，叫做根类。根类没有超类。所有Objective-C对象的根类都是NSObject，只是在极少数涉及NSProxy类的情况下，才有例外。

注意 NSProxy是一个框架类，它用作负责替代其他对象的那些对象的根类。例如，当构建使用分布式对象的系统（使用多个进程的系统）时，可以使用NSProxy的子类来替代另一个进程中的对象。当代理对象接收到一条消息时，该代理将消息转发给它所代表的对象。要了解关于分布式对象的更多信息，参见http://developer. apple. com/mac/library/documentation/ Cocoa/Conceptual/ObjectiveC/Articles/ocRemoteMessaging.html。

NSObject定义和实现了一个对象要与Objective-C运行时正确交互或者成为一个良好的Objective-C公民所必需的基本方法。这些方法的一些示例有，用于对象分配和内存管理的方法，以及用于内省的方法。当创建一个类时，不必亲自去实现这些方法，从NSObject继承，会自动带有这些方法。

注意 内省指的是在运行时确定一个对象的类的能力。

6.2.4 类层级示例

图6-1给出了一个假设的餐馆经营程序的一部分类层级。

随着从类层级的根类向下移动，也就从通用走向了具体。在图6-1中，MenuItem类拥有关于菜品的实例变量，例如价格和总热量，这是餐馆提供的每一道菜品所常见的信息。BeverageItem类将添加针对饮料的实例变量，例如，是否可以免费续杯。如果食物和饮料之间没有区别，就不需要区分FoodItem和BeverageItem类了。类层级中最低的类是最为具体的。例如，Wine类仅仅保存了与酒相关的信息：品种、年份和生产商的名称。

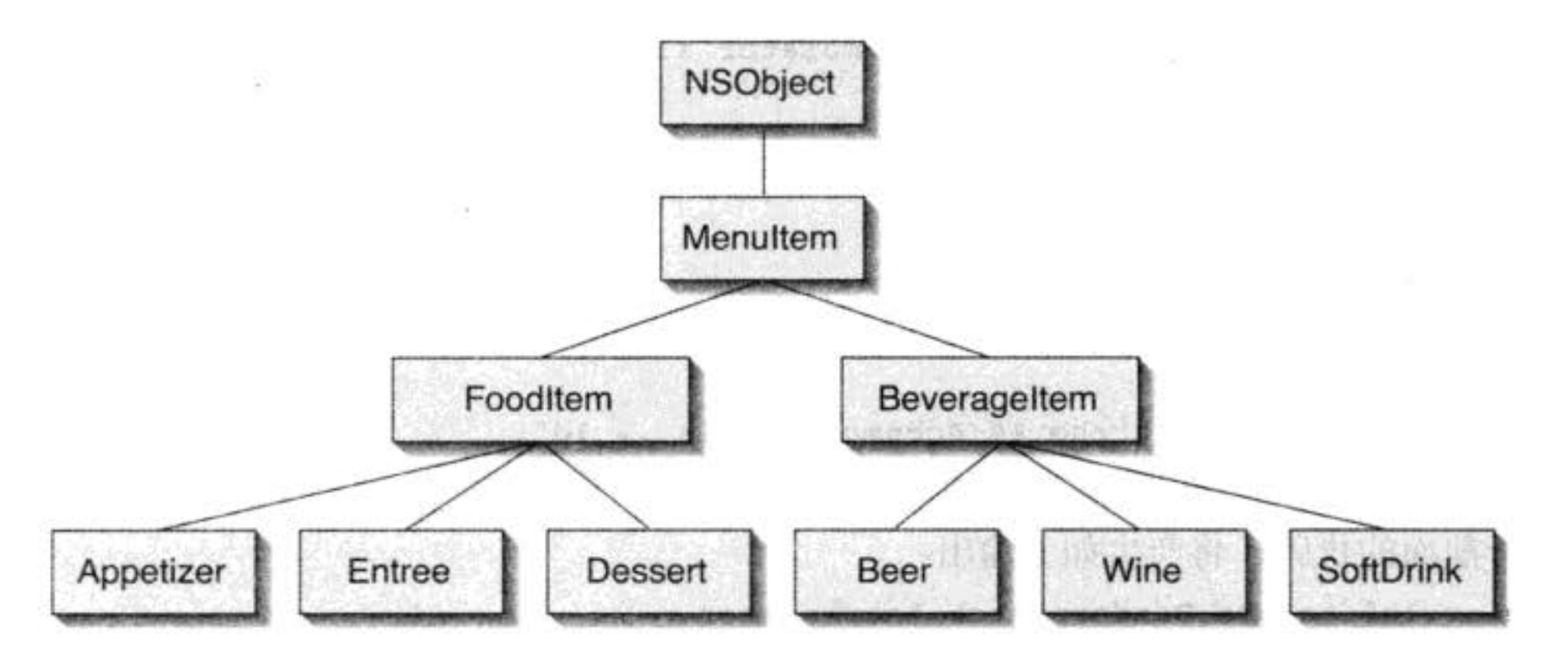


图6-1 一个餐馆经营程序的类层级

注意 尽管继承是强大的工具，但也不应该肆意地使用它。太长的继承链可能会使得程序不够稳定。如果你必须修改靠近一条长长的继承链顶部的一个类，其效果将会传递到整个程序，从而可能产生无法预料的结果。还有一种不同的技术，即添加一个分类，它允许给一个类添加方法而不需要将其子类化。关于这个技术参见第11章的介绍。

6.2.5 抽象类

在前面的餐馆示例中，MenuItem、FoodItem和BeverageItem都是抽象类。抽象类是专门用作子类化的模板的类。通常不需要创建抽象类的实例。

直接创建一个MenuItem实例毫无意义。你不能吃喝一个菜品的抽象概念。但是，通过在一个抽象超类中实现存储菜品的价格和热量的功能，该抽象类的所有子类也就自动地具备了这一功能。在这个示例中，只需要编写一次代码来设置和获取一个菜品的价格和热量值。没有该抽象类，将必须编写该代码6次。使用一个抽象类不只是节省精力，而且也不容易出错。

一些抽象类声明了专门供其子类实现的方法。MenuItem类可能定义一个menuBlurb方法，以提供描述菜品的文本。MenuItem类可能会提供menuBlurb的一个空的（不做任何事情）实现，并且期待其一个非抽象的子孙，使用有具体类代码的实现，来覆盖其空的实现。

没有规定子类要覆盖方法。然而，如果抽象类的用户忘记了覆盖一个“必需”的方法，则可以通过记录调用抽象类的实现的任何尝试来警告他们。

在这种情况下，MenuItem的menuBlurb实现可能如下所示：

```
- (NSString*) menuBlurb
{
  NSLog(@"Warning: concrete subclass of class MenuItem must override -menuBlurb." );

  return nil;
}
```

也可以创建一个“半抽象的”类。当这些类自身实例化时有某些用途，但是，这要求子类

执行更加高级的功能。Cocoa框架中这样的类的一个典型的例子就是NSView，它表示屏幕的一个区域。纯NSView的实例提供了某些功能：它可以用作其他视图的一个容器，而其他的视图则称为子视图。但是，要在一个视图中绘制任何内容，必须定义一个NSView的子类，并且覆盖drawRect:方法。类似地，如果需要一个视图来响应用户输入，那么视图必须是NSView的一个子类，并且覆盖一个或多个事件处理方法，例如mouseDown:。

抽象类或方法没有特殊的语法

Objective-C对于抽象类没有明确的语法。如果有一个类，它实际上是一个抽象类，你应该在自己的文档中注明这一点（在该类的头文件及任何类文档中）。和Java或C++中的抽象类不同，Objective-C没有方法来阻止某人真正地去实例化一个你认为是抽象的类。

Objective-C没有语法用来声明一个方法是抽象的或虚拟的。如果你创建了一个抽象类，那么对于要求抽象类的子类实现的方法，应该提供一个空的实现。如果你的抽象类声明了这些方法，但是没有为其提供一个实现，那么你将会得到一条编译器警告。如果随后子类化了该抽象类，并且仍然忘记提供一个实现，如果调用该方法，将会得到一个运行时错误。

6.3 创建对象

创建Objective-C对象需要两步：分配和初始化。分配步骤从堆分配用于该对象的内存；初始化步骤填充那些需要初始值的任何实例变量，获取对象所需的任何资源，并且返回指向完成后的对象的一个指针。这两个步骤通常组合到一条语句中：

```
Foo* myFoo = [[Foo alloc] init];
```

接下来将更为详细地介绍这些步骤。

6.3.1 对象分配

对象分配由一个类方法alloc处理，所有的类都从NSObject继承了它。

注意 在第7章中，将学习关于类对象和类方法的更多内容。现在，只需要知道在使用类对象，或者，更通俗地说，当调用一个类方法时，类的名称作为接收者就可以了。

alloc做几件事情：

- alloc为新的对象分配内存，并且返回指向该内存的一个指针。
- alloc填充对象的isa实例变量。编译器自动为每个对象实例添加一个名为isa的实例变量。isa指向关于对象的类的信息的一个结构。
- alloc清空其他的对象内存。除了isa以外的所有实例变量都以一个0值开始。
- alloc将新创建的对象的引用计数设置为1。通过发送alloc消息来创建对象的代码，也负责在最后给该对象发送一条release或autorelease消息来对应alloc（第14章将介绍autorelease）。

alloc返回的对象变成了一条init消息的接收者。init执行任何所需的初始化，并且返回一个指针指向完全构造好的对象。

new

new是NeXT时期的遗留物。如果需要处理某些遗留代码，可能会看到它。它是一个将分配和初始化组合到一个单一步骤中的类方法：

```
Foo *myFoo = [Foo new];
```

+new等同于（并且当前也实现为）alloc后面跟着init。new并没有被废弃，但它如今很少使用了。当初始化一个对象时，如果该对象需要通过参数将信息传入到一个init函数中，则不能使用new。大多数人喜欢对其所有的对象分配使用alloc...init模式。

> **alloc和allocWithZone：**
>
> Mac OS X将堆（作为动态分配的字节来源的内存区域）划分为称为malloc区域的区域。类方法allocWithZone:允许指定哪个区域用作一个对象的字节的来源。
>
> alloc实现为allocWithZone: nil。nil参数告诉allocWithZone从默认malloc区域获取字节。使用malloc区域而非默认malloc区域，这是一项特别的技术，在某些情况下可能会引发问题。Apple当前不鼓励使用malloc区域，因此，对你的对象分配直接使用alloc就可以了。

6.3.2 对象初始化

Objective-C的初始化方法是按照惯例以单词init开头的实例方法。如果你要设计一个类，其对象不需要任何特定的初始化，那么，可以使用初始化方法init，其所有的类都继承自NSObject。NSObject的init实现不做任何事情，只是返回self。但是，如果你的类的实例需要定制初始化，例如将实例变量设置为特定的值、注册回调或者获取资源，则必须覆盖init并且提供你自己的实现。

init的形式

如下是一个init实现的一般形式：

```
1 - (id) init
2 {
3   if ( self = [super init] )
4     {
5       // Class-specific initializations
6     }
7   return self;
8 }
```

来看看这段代码：

- 第3行：（是的，只有一个=）。这一行代码调用了超类init方法来初始化对象的超类部分。超类初始化返回的对象，随后将其赋值给了self变量。还记得吧，self是一个隐藏参数，它由编译器添加给所有方法。

  对self赋值是必要的。当调用init时，self的值是指向最初的接收者（alloc返回的

对象）的一个指针。然而，超类init可能会失败并返回nil，或者，它可能会释放最初的接收者，分配并返回一个不同的对象。由于self最终用来返回指向初始化后的对象的一个指针，因此必须将其设置为超类init返回的对象。

- 第5行：如果超类已经返回了一个非nil的对象，则可以以任何必需的初始化来处理它。
- 第7行：最后，必须记得返回指向新创建的对象的指针。如果忘记了，那么该方法返回的值将是垃圾。

init的返回类型

你可能注意到了（并且感到惊奇），init方法的返回类型是一个id，而不是指向一个特定类的实例的指针。这有两个原因：

- 正如前面提到的，init方法可能会释放其接收者，并且，替代了属于最初的接收者的超类的对象。对于那些实现为类簇的类来说，经常发生这种情况。

  类簇设计为一个公共装饰类，它负责管理一组私有子类。当分配并初始化这个公共类的一个实例时，我们实际接受的是一个私有子类的实例。很多常见的Foundation类，例如NSString、NSArray和 NSDictionary都是类簇的例子。更多内容参见本书第9章。
- 相同的init方法可以用于不同的类的接收者。将一个init方法的返回值的类型确定为id，这允许继承了声明该方法的初始类的那些类，继承并使用该方法。

组合alloc和init

将分配和初始化组合到一条单个的语句中，这是通常的惯例：

```
Foo* myFoo =  [[Foo alloc] init]; // Correct
```

这么做防止了如下的这个错误：

```
Foo* myFoo = [Foo alloc]; // WRONG!
[myFoo init];
```

上面的代码有何错误？它丢弃了init的返回值。如果init替代了属于所请求的类的一个子类的对象（如果Foo是一个类簇的话，就会发生这种情况），或者如果初始化失效并且init返回了nil，myFoo将会最终保存一个指针，指向已经返回给堆的内存。

如下的代码在技术上是正确的，但是，它很冗长而且是糟糕的做法，因为它可能导致你忘记将init的返回值赋给myFoo：

```
// Technically correct but verbose and poor practice

Foo* myFoo = [Foo alloc];
myFoo = [myFoo init];
```

注意 类簇是一种设计模式，它将多个私有子类隐藏到一个公有的可见的抽象类的背后。当请求公有类的一个实例时，接收到的是一个私有子类的实例。更多内容参见本书第9章。

带参数的初始化程序

如果想要将自己的类的实例变量初始化为一个特定的值，该怎么办呢？需要一个初始化程序，它接受一个参数，以便可以传入值（对于默认的初始化程序init来说，也没什么特殊的，除了每个类都继承自NSObject这一事实外。需要的话，一个类可以有多个初始化程序）。

注意 如果想要让一个实例变量的初始值为0或nil，那么使用带有一个参数的初始化程序并非是必须的。和函数或方法的自动变量不同，在对象创建时，实例变量是（由alloc）初始化为0的。

例如，假设需要设计一个TeeShirt类。TeeShirt类应该有一个NSUInteger实例变量shirtSize，用来保存T恤的尺寸。在创建TeeShirt对象时，使用如下所示的一个初始化方法来初始化T恤的尺寸：

```
- (id) initWithShirtSize:(NSUInteger) inShirtSize
{
  if ( self = [super init] )
    {
      shirtSize = inShirtSize;
    }
  return self;
}
```

注意 以init打头的名字来命名初始化程序，这是通用的惯例。接收参数的初始化程序，通常带有形式为initWithFoo:的名字，其中Foo是初始化程序所要设置的属性（这里使用表达式init...作为“初始化方法”的缩写）。

初始化程序可以有多个参数。如果你重新设计TeeShirt类，使其拥有另一个实例变量，用来保存T恤的颜色，则应该丢弃最初的initWithShirtSize:，并添加一个能够同时设置T恤尺寸和颜色的初始化程序：

```
 1 - (id) initWithShirtSize:(NSUInteger) inShirtSize
 2         color:(NSColor*) inShirtColor
 3 {
 4   if ( self = [super init] )
 5     {
 6       shirtSize = inShirtSize;
 7       shirtColor = [inShirtColor retain];
 8     }
 9   return self;
10 }
```

在第7行，要确保当TeeShirt对象使用NSColor对象时，还没有销毁NSColor，这条retain消息是必需的。该retain消息必须通过TeeShirt的dealloc中的一条release消息来与之对应。

当有多个实例变量要初始化时，可以将初始化程序设计为，只是将某些变量设置为作为参

数传入的值，其他的变量则设置为默认值。这里有theTeeShirt类的一个初始化程序，它直接设置shirtSize，但是，将shirtColor设置为一个默认颜色：

```
- (id) initWithShirtSize:(int) inShirtSize
{
  NSColor *defaultShirtColor = [NSColor whiteColor];
  return      [self initWithShirtSize: inShirtSize
                    color: defaultShirtColor];
}
```

注意，上面的代码采用了一种不同的模式，initWithShirtSize:获取T恤颜色的一个默认值。然后，调用TeeShirt的另一个初始化程序initWithShirtSize:color:，来完成初始化对象的实际工作。

指定的初始化程序

每个类必须有一个初始化程序作为指定的初始化程序。指定的初始化程序将会完全初始化类的实例。指定的初始化程序通常会拥有大多数参数，或者在建立对象时完成大部分的工作。对于不需要带有参数的初始化程序的类来说，指定的初始化程序是init。

在前面的TeeShirt示例中，initWithShirtSize:color:是TeeShirt类的指定的初始化程序。使用initWithShirtSize:color:将会产生TeeShirt的一个完全初始化的实例。

在为一个类设计初始化程序时，有两条规则必须遵守：

1) 所有其他的类初始化程序，最终必须调用该类的（而不是超类的）指定初始化程序。例如，TeeShirt的initWithShirtSize:调用了initWithShirtSize:color:，后者是TeeShirt的指定初始化程序。

2) 类的指定初始化程序必须调用其超类的指定初始化程序。指定初始化程序，是调用超类的指定初始化程序的唯一初始化程序。

TeeShirt遵从这些规则：

1) initWithShirtSize:调用initWithShirtSize:color:，后者是TeeShirt的指定初始化程序。

2) initWithShirtSize:color:调用超类的指定初始化程序（NSObject的）init。

注意 *应该（和Cocoa框架一样）在类的文档中注明哪一个初始化程序是指定初始化程序。然而，尽管需要遵守前面的规则，在Objective-C语法中，对于一个指定初始化程序还是没有正式的表示法。*

实际上，TeeShirt类还有一个问题需要注意，那就是TeeShirt有一个与其超类NSObject不同的指定初始化程序（二者分别是initWithShirtSize:color:和init）。如果编写如下代码，请判断将会发生什么情况：

```
TeeShirt *aTeeShirt = [[TeeShirt alloc] init];
```

TeeShirt没有覆盖init，因此，在上面这行代码中，调用的将是NSObject的init实

现。NSObject不知道关于TeeShirt的任何事情。新创建的TeeShirt实例的shirtSize和shirtColor，将会错误地保留为未初始化。为了确保在这种情况下正确地初始化TeeShirt的实例，必须覆盖init，然后从覆盖中调用TeeShirt的指定初始化程序：

```
- (id) init
{
  NSUInteger *defaultShirtSize = 36;
  NSColor *defaultShirtColor = [NSColor whiteColor];
  return
    [self
      initWithShirtSize: defaultShirtSize color: defaultShirtColor];
}
```

图6-2给出了TeeShirt的初始化程序调用的模式。带有阴影背景的方框，是该类的指定初始化程序。

如果类的指定初始化程序与超类的不同（具有不同的名称），则必须覆盖在类的祖先类中声明的任何init...方法，并且从覆盖方法中调用类的指定初始化程序。实际操作比听上去要容易。Objective-C类层级通常不是很深，并且，在大多数情况下，只需要注意几个初始化程序就可以了。

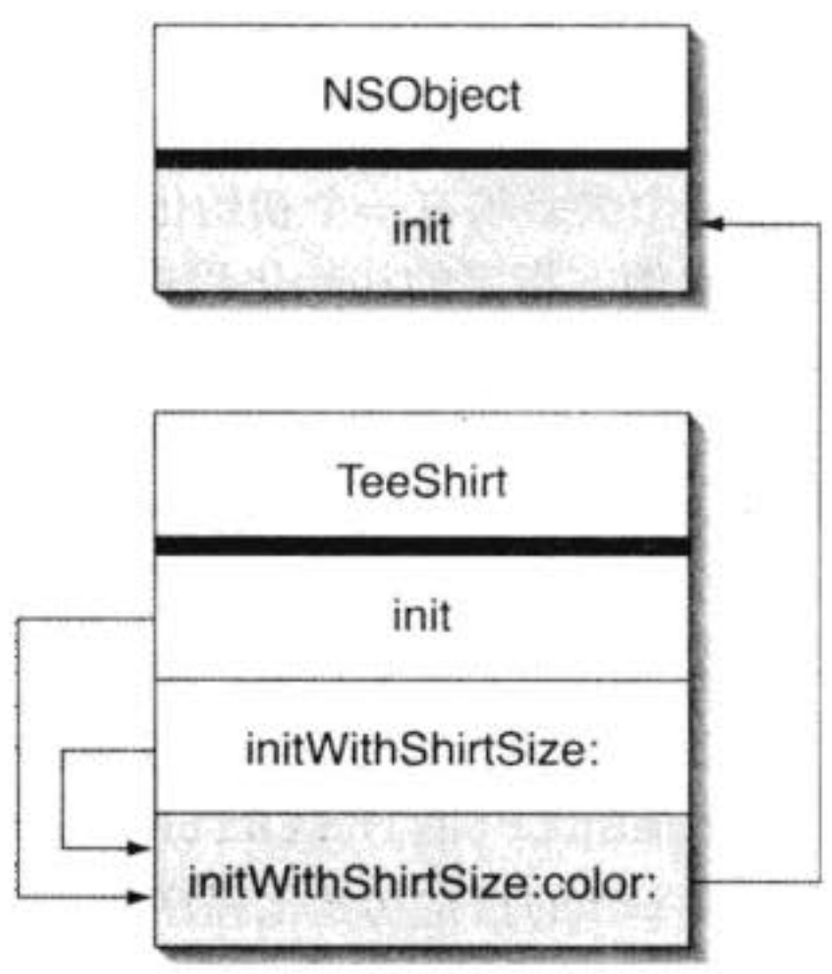


图6-2 TeeShirt类的初始化程序。阴影框是指定初始化程序

初始化失败

如果不能成功地初始化一个对象，将会发生什么呢？可能你的对象为了要发挥功能而所需的一些资源会无法使用。如果发生这种情况，则你应该合理地清理失败：

- 对于dealloc没有解决的任何清理，都要处理好。例如，如果你创建了一个临时对象或连接，并且通常应该随后在init方法中释放掉它，那么，你需要在返回之前释放它。
- 释放self。因为self由alloc创建，带有一个值为1的引用计数，你必须使用一个release来对应alloc。
- 返回nil。这表示初始化调用代码失败，并且，没有创建任何对象。

注意 在一个初始化方法内部执行[self release]，会导致对象的引用计数减少为0，从而导致调用对象的dealloc方法（参见6.4节）。如果类的设计要求你处理一个初始化失败，你必须也确保，在失败的时候调用类的dealloc是安全的。

例如，假设有一个对象，它有两个实例变量anArray1和anArray2，它们都是数组对象（第9章将介绍NSArray和NSMutableArray，对于这个例子，不需要了解与它们相关的任何细节）。进一步，假设要让对象正确地工作，这两个数组都必须正确初始化。

anArray1分配为一个空数组，而anArray2使用NSArray初始化方法

initWithContentsOfFile:，用一个文件的内容来初始化。但是，如果在指定的路径没有文件，那么initWithContentsOfFile:将失效并且返回nil。如果发生这种情况，你的对象就不能正确地初始化，因此，你的init应该负责清理并返回nil。

如下的代码显示了如何处理这种情况：

```
 1 - (id) init
 2 {
 3 if ( self = [super init] )
 4   {
 5     anArray1 = [[NSMutableArray alloc] init];
 6
 7     if ( anArray1 == nil )
 8       {
 9         [self release];
10         return nil;
11       }
12
13     anArray2 =
14           [[NSArray alloc]
15                initWithContentsOfFile: @"/Users/rclair/stuff"];
16
17     if ( anArray2 == nil )
18       {
19         [self release];
20           return nil;
21         }
22     }
23   return self;
24 }
25
26 - (void) dealloc
27 {
28    [anArray1 release];
29   [anArray2 release];
30   [super dealloc];
31 }
```

现在，让我们看看这个示例中的代码：

- 第3行：调用超类的init，将返回值赋给self，并且检查以确定它不是nil。
- 第5行：分配和初始化anArray1。
- 第7行：只有在你的内存完全耗尽时，anArray1的分配才会失败。如果发生这种情况，你将会遇到很多其他的问题，并且程序有可能会很快崩溃。然而，出于正确的和实用的良好编码习惯，这一行代码检查anArray1的值是否是nil。
- 第9行：如果anArray1是nil，则该对象不能初始化，因此，代码会释放self。如果你没有释放self，则分配给self的字节将会泄漏。它们不能用于任何内容，并且无法返回给堆。

释放self会导致调用对象的dealloc方法。第28行和第29行没有引发任何问题，因为anArray1和anArray2此刻都是nil。第30行调用了超类的dealloc实现，这会将存储在超类所声明的实例变量中的任何对象都释放掉。

- 第10行：返回nil表示初始化失败。

如果成功地创建了anArray1，则init方法继续：

- 第13行：分配anArray2并且使用/Users/rclair/stuff文件的内容来初始化它。
- 第17行：查看anArray2是否成功创建。
- 第19行：如果anArray2没有创建，释放self，这导致调用dealloc。第28行释放了anArray1，它是在第5行分配的。
- 第20行：返回nil表示初始化失败。
- 第23行：如果anArray1和anArray2都不是nil，并且，超类的初始化没有失败，则这一行返回一个完成初始化的对象。

注意 *在[self release]执行之后，self中保存的对象不再有效。不要在调用self的release和返回nil之间执行任何引用self的代码。如果违反了这一规则，并且执行了错误的代码，将会导致程序崩溃。*

6.4 销毁对象

当使用完对象之后，该如何删除它呢？正确地处理不需要的对象，以便其字节能够返回到堆以供重用，这是很重要的。在Objective-C中，不需要显式地销毁对象。

如果使用垃圾收集（参见第15章），不需要（大多数时候）担心对象的销毁。一个称为垃圾收集的单独进程，将会周期性地被调用，以确定不再需要哪些对象，并且删除它们。

如果使用引用计数，当使用完了所创建或保留的对象时，要向其发送一条release消息。如果这条release消息将对象的引用计数减少到0，则释放方法将调用对象的dealloc方法。对象的dealloc方法应该：

1) 向将要消失的对象所创建或引用的任何对象发送一条release消息。

2) 调用超类的dealloc方法。

前面讲述的TeeShirt类的dealloc方法如下所示：

```
- (void) dealloc
{
  [shirtColor release];
  [super dealloc];  // Important!
}
```

在上面的代码中，shirtColor保存了一个对象（一个NSColor），当创建TeeShirt实例时，保留了该对象。当它要消失的时候，TeeShirt对象必须使用一条release消息来对应该retain。否则，shirtColor中保存的对象将变成一个潜在的内存泄漏。

[super dealloc]这一行很重要。它是dealloc的一个NSObject实现，向堆返回了对

象的字节。如果在继承链中，在你的类和NSObject之间的任何类忘了在自己的dealloc中调用[super dealloc]，将不会调用NSObject的实现。用于对象存储的字节将会泄漏。如果忘记了[super dealloc]，编译器将会通过警告来帮助你。

> **注意** 要注意拼写。如果你由于错误拼写或者意外，将dealloc名称写成了deallloc或者其他错误形式，将会调用超类的dealloc。你的dealloc实现应该释放的任何对象，都将会泄漏。编译器不会对此提出警告。它认为deallloc只不过是你为子类声明的另一个方法。

当编写一个dealloc方法时，[super dealloc]消息应该放在该方法的最后一行。在[super dealloc]返回之后，对象就已经被销毁了，因此对self的任何引用都将是无效的。

例如，考虑向NamedAccumulator类的dealloc方法（参见程序清单6.4）添加最后一条记录语句。

如果如下这样编写dealloc方法：

```
- (void) dealloc
{
  // WRONG!
  [super dealloc];

  NSLog(@"The final total of %@ is %d", [self name], [self total] );
  [name release];
}
```

当执行dealloc方法时，程序将会崩溃：

```
*** -[NamedAccumulator name]:
          message sent to deallocated instance 0x10010d6b0
```

编写这个方法的正确方式如下所示：

```
- (void) dealloc
{
  // Correct
  NSLog(@"The final total of %@ is %d", [self name], [self total] );
  [name release];

  [super dealloc];
}
```

在上面的代码中，还要注意位于[name release]消息之前的[self name]消息。释放一个实例变量可能导致它被收回。

6.5 复制对象

复制对象比我们想象的更困难，这是由一些所谓的欺骗性广告造成的。NSObject实现了一个名为copy的方法。看到这里，你可能会尝试着使用继承的copy方法来复制一个对象：

```
SomeClass *anObject = [[SomeClass alloc] init];
SomeClass *anObjectCopy = [anObject copy];
```

编译器没有给出警告，但是，除非SomeClass已经正确地编写了，否则这将导致程序立即崩溃。

NSObject的copy实现使用一个nil参数调用名为copyWithZone:的方法。问题是，NSObject没有实现copyWithZone:。NSObject不能实现copyWithZone:，因为它没有办法知道特定的子类应该实现什么类型的副本，或者甚至不知道该子类是否允许复制。

注意 copyWithZone:允许指定，在获取用于复制的字节时，使用哪个malloc区域。copy和copyWithZone的关系，就像是alloc对allocWithZone的关系。Apple现在不鼓励使用默认区域以外的任何区域（默认区域就是当使用一个nil参数调用copyWithZone:时所使用的区域）。

允许复制的类采用了NSCopying协议。协议将在第13章中介绍，但是，现在只需要知道，这只是意味着这个类必须要实现copyWithZone:。当试图复制一个Cocoa框架对象时，确保对象的类采用了NSCopying协议（应该在该类的文档中注明这一点）。有很多框架类是不能够复制的。

如果需要复制你所定义的一个类的对象，必须自己实现copyWithZone:。在处理之前，你需要针对你的类应该提供什么类型的复制来做出一些决定。下面将介绍在为你的类设计copyWithZone:时，应该考虑的一些事情。

6.5.1 浅复制和深复制

如果要复制的对象中的任何实例变量包含指针（指向其他的对象或者指向malloc的内存），那么你必须决定，是否用保存在最初对象中的同样的指针，来填充新创建的副本中对应的实例变量。如果你给新副本指定了最初的指针，你将进行一次浅复制。替代的方法是，依次复制任何子对象，并且将子对象的副本指定给新对象。这种类型的复制叫做深复制。

你应该怎么做？这个问题不能笼统地回答。当你设计自己的类时，必须根据自己的需求来做出决定。下面是一些问题：

- 浅复制使得副本和最初的对象纠结在一起。如果通过改变一个共同保存的子对象的状态，来修改一个对象，这一修改将会反映到最初的对象和副本中。
- 深复制保持了两个完全独立的对象（只要任何复制的子对象本身都实现了一次深复制），但是，进行深复制涉及比浅复制代价更大的操作（在执行时和内存中）。
- 一个副本不一定只是一个深复制或只是一个浅复制。可以对一些实例变量实现深复制，而对另一些实例变量实行浅复制。
- 如果一个类的实例变量都是基本的C类型，深复制和浅复制之间就没有区别。

大多数Cocoa框架类都实现了浅复制。

6.5.2 可变复制和不可变复制

我们将会在第9章中看到，一些Cocoa框架类创建了不可变的实例。一个不可变的对象的实

例变量，在对象创建之后就不可以对其修改。例如，NSNumber和NSColor的实例是不可变的。如果想要一个有不同值的NSNumber或者有不同颜色成分的NSColor，就必须创建新的对象。这是如何强制实现的呢？这些类不直接提供任何方法来修改其内部状态。

其他的框架类，如NSString及集合类（NSArray、NSDictionary和NSSet），创建了不可变的实例，但是，它们却拥有能够创建可变的实例的子类。对于这样的类，NSObject对象提供了mutableCopy方法。

注意 与复制的方式类似，NSObject的mutableCopy方法只是使用一个nil参数调用mutableCopyWithZone:。你可能已经猜到了，NSObject没有实现mutableCopyWithZone:。mutableCopyWithZone:的实现，留给了那些提供了可变子类的任何不可变类。实现了mutableCopyWithZone:的类，据说采用NSMutableCopying协议。

例如，NSArray允许复制和可变复制。要获取NSArray的一个可变副本（与最初的数组具有同样的内容，但是允许你向数组添加或删除对象），可以如下这样编写代码：

```
NSArray *anArray = ...
NSMutableArray *aMutableCopyOfAnArray = [anArray mutableCopy];
```

当然，你应该检查文档，以便在试图进行一次可变复制之前，确保你要复制的对象的类支持可变复制。

注意 Foundation文档位于http://developer.apple.com/mac/library/ documentation/Cocoa/Reference/ Foundation/ObjC_classic/，AppKit的文档位于http:// developer.apple.com/mac/ library/documentation/Cocoa/Reference/ApplicationKit/ObjC_classic/。

6.5.3 在自己的类中实现复制

如果类的实例假设是不可变的，则可以通过向将要复制的对象发送一条额外的retain消息来实现copyWithZone:

```
- (id) copyWithZone:(NSZone*) zone
{
  return [self retain];  // self is an instance of an immutable class
}
```

由于不可变对象是不能修改的，因此在最初的版本和副本之间将没有任何区别。可以使用最初的版本作为副本，只要像这个例子中所示的那样添加一条retain即可，以便在准备释放该对象之前，它不会消失。

在更加典型的情况中，你所复制的对象是不可变的，你必须为一个新的对象分配存储，然后，使用从最初对象派生的值来初始化新的对象。

对于实现copyWithZone:通常的建议是，调用copyWithZone:的超类实现以创建新的副本，并且填充超类中定义的任何实例变量。copyWithZone:的子类实现的其余部分，随后填充了子类中定义的任何实例变量，并且返回该副本。

遗憾的是，如果你的类是NSObject的直接子类，或者没有实现copyWithZone:的任何其他的类，你则不能这么做。在这种情况下，必须调用allocate并自行初始化副本。以TeeShirt类为例，对copyWithZone:方法的第一次尝试可能如程序清单6.5所示。

程序清单6.5 TeeShirt copyWithZone:（版本1）

```
- (id) copyWithZone:(NSZone*) zone
{
  id copiedTeeShirt =
    [[TeeShirt allocWithZone: zone]  // Bug in waiting!
       initWithShirtSize: [self shirtSize] color: [self color]];

  return copiedTeeShirt;
}
```

这对于复制TeeShirt实例有效，但是，如果你确定要子类化TeeShirt，它是一个潜在的bug。为了看看为什么会这样，定义一个SloganTeeShirt类，它通过在T恤的前面添加一个印刷的字符串，来扩展TeeShirt，如下面的代码所示：

```
#import "TeeShirt.h"

@interface SloganTeeShirt : TeeShirt
{
  NSString* slogan;
}
-(void) setSlogan:(NSString*) slogan;
-(NSString*) slogan;

...

@end
```

然后，可以为SloganTeeShirt类实现copyWithZone:，如程序清单6.6所示。

程序清单6.6 SloganTeeShirt copyWithZone:

```
1 - (id) copyWithZone:(NSZone*) zone
2 {
3   // This is a BUG because the superclass copyWithZone is
4   // incorrectly coded
5
6   id copiedSloganTeeShirt = [super copyWithZone: zone];
7   [copiedSloganTeeShirt setSlogan: [self slogan];
8   return copiedSloganTeeShirt;
9 }
```

然而，这里有一个问题。当执行到第6行时，由于一个不被识别的选择器异常而导致程序崩溃。问题在于copyWithZone:的TeeShirt实现（参考程序清单6.5）。当在程序清单6.6的

第6行调用[super copyWithZone: zone]时，TeeShirt的copyWithZone:返回一个TeeShirt对象，而不是一个SloganTeeShirt对象。

当执行第6行时，给保存在copiedSloganTeeShirt中的TeeShirt对象发送了一条setSlogan消息。由于TeeShirt没有实现setSlogan:方法，因此这一行将导致程序崩溃。

解决方案是，修改程序清单6.5，使TeeShirt的copyWithZone:返回一个对象，该对象与用来调用它的接收者具有相同的类。如果以一个TeeShirt对象作为接收者调用它，它返回TeeShirt的一个完整副本；如果以一个SloganTeeShirt对象作为接收者调用它，它返回一个SloganTeeShirt对象，其中填充了超类实例变量。

TeeShirt的copyWithZone:的正确版本如程序清单6.7所示。

程序清单6.7 TeeShirt copyWithZone:（版本2）

```
1 -(id) copyWithZone:(NSZone*) zone
2 {
3   id copiedTeeShirt =
4     [[[self class] allocWithZone: zone]
5       initWithShirtSize: [self shirtSize] color: [self shirtColor]];
6
7   return copiedTeeShirt;
8 }
```

这里的关键是第4行。消息[self class]返回了self所指向的对象的类。当以一个TeeShirt对象作为接收者调用TessShirt的copyWithZone:实现时，[self class]返回TeeShirt，并且该方法返回的对象是一个TeeShirt对象。但是，当以一个SloganTeeShirt作为接收者调用TessShirt的copyWithZone:实现时，如程序清单6.6的第6行所示，self是SloganTeeShirt的一个实例。[self class]产生了SloganTeeShirt（前面讲过，当使用super调用一个方法时，该方法是超类的实现，但是，接收者是最初的对象）。新创建的对象是一个设置自己的slogan实例变量的SloganTeeShirt。

最后，如果使用引用计数，按照Objective-C命名惯例，可以通过在方法名中包含单词“copy”的方法，来拥有返回给你的任何对象。最终，当你使用完复制的对象时，你必须向它发送一条release消息。当编写自己的copyWithZone:方法时，你应该通过确保返回引用计数为1的对象，来小心地遵守这一惯例。在通常情况下，当你为副本（见程序清单6.7）返回一个新分配的对象，或者是返回从另一个复制方法获得的一个对象（见程序清单6.6）时，你会很满足的。

6.6 小结

学习完本章，你应该知道了如何定义一个类，以及如何分配、初始化、销毁和复制对象。本章和第5章的内容都是关于核心Objective-C的消息机制的。此时，你应该能够编写一个中等但完整的Objective-C程序了。

6.7 练习

1. 创建一个RockStar类和一个Band类的框架。RockStar类应该有一个类型为Band的实例变量，Band类应该有一个类型为RockStar的实例变量。
   - 验证一下，如果将Band.h导入到RockStar.h并且将RockStar.h导入到Band.h，两个类将不编译。
   - 验证一下，可以通过使用一条@class指令来替代其中一个或两个导入，从而修正这一问题。
2. 创建一个较小的测试程序，并且使用它来验证NamedAccumulator类能够像预期的那样工作（必须将程序清单6.1、程序清单6.2、程序清单6.3和程序清单6.4中的内容添加到你的项目中）。
3. 定义并编写NamedAccumulator类的初始化程序。初始化程序应该有一个NSString参数来设置名称，以及一个int参数来为总和设置一个初始值。第二个初始化程序应该接受一个NSString参数作为名称，但是，将总和的默认值设置为0。确保你的初始化程序遵从Objective-C命名惯例。
   - 你的新初始化程序之一必须作为类的指定初始化程序。这应该是哪一个呢？
   - 确保所有的初始化程序最终都调用指定初始化程序。
   - 不要忘了修改init。
   - 使用练习2的测试程序，来验证你的初始化程序能够正确地工作。
4. 实现对Accumulator和NamedAccumulator类的复制。使用练习2的测试程序，来验证你的复制方法能够正确工作。
5. 创建Accumulator的一个新子类，命名为AveragingAccumulator。Averaging-Accumulator应该保存调用其addToTotal:方法的总的次数，它还应该有一个方法来返回该数值，并且还有一个方法averagePerTransaction，它会报告所有addToTotal:事务的平均值。使用一个测试程序，验证新的类能够正确工作。

第7章 类 对 象

Objective-C与Java和C++不同，Objective-C中的类自身就是对象，它们是名为Class的特殊类的实例。你可以发送类消息，并且，它们可以执行方法。本章介绍类对象和类方法的技术细节，然后，简单地认识一些类方法的示例。本章7.4节给出一个扩展的示例，通过这个示例，我们将了解到如何实现对给定的类的所有实例来说都是全局的变量。

7.1 类对象

Objective-C程序中的每个类都由一个单个的类对象表示，该类对象包含了定义该类的信息。我们不必为实例化类对象做任何事情，编译器会根据类定义中的信息来创建它们。

类对象可以接收消息，并且可以执行称为类方法的方法。我们已经见过这方面最常见的示例，即alloc类方法：

```
[SomeClass alloc]
```

类消息中的接收者是一个类对象。在消息表达式中，类名称可以用来表示该类对象，如上面的例子所示。

注意 *类消息表达式中的接收者，是类名称可以替代其对应的类对象的唯一地方。*

类对象不是它们表示的类的实例。不能将类对象用作任何类的实例方法的接收者。例如，count是NSArray类的一个实例方法。它返回NSArray中对象的数目。如果试图将NSArray类对象作为接收者来使用count：

```
[NSArray count];
```

将会得到如下的编译器警告：

```
testbed.m:6: warning: 'NSArray' may not respond to '+count'
testbed.m:6: warning: (Messages without a matching method signature
testbed.m:6: warning: will be assumed to return 'id' and accept
testbed.m:6: warning: '...' as arguments.)
```

如果忽略了警告并执行代码，程序将会崩溃：

```
*** Terminating app due to uncaught exception
  'NSInvalidArgumentException',
reason: '+[NSArray count]:
   unrecognized selector sent to class 0x7fff707ebd28'
```

尽管Objective-C类是对象，但类对象也没有实例变量。在其他面向对象语言中见到的类变量，在Objective-C中没有对等的概念。

注意 类变量（在拥有类变量的语言中）是对一个类来说全局性的变量。与一般的实例变量相反，类变量只有一个副本，供类的所有实例共享。

然而，在7.3节中，将了解到如何使用普通C外部变量来模拟类变量的功能。

7.1.1 类类型

类型为Class的变量，用作指向类对象的指针。可以通过使用类名作为接收者来调用类方法，从而获得指向类对象的一个指针：[⊖]

```
Class aClass = [NSString class];

// aClass now holds a pointer to the NSString class object
```

注意 类本身是指针类型。在声明Class变量时，没有必要使用*：

```
Class *someClass;   // WRONG!
```

类似地，可以使用类方法superclass来获取一个类的超类的类对象：

```
Class aClass = [NSString superclass];

// aClass now holds a pointer to NSString's superclass, NSObject
```

注意 也有NSObject实例方法class和superclass，它们可以返回一个对象的类和超类。对于那些并非是你自己编写的类使用这些实例方法，应该特别小心。很多常见的类实际上是类簇。当请求一个类簇的实例时，可能接收最初类的一个私有子类的实例。在一个类簇的实例上调用class或superclass的结果，可能与你所预期的不同（参见本书第9章）。

Class类型的变量可以用来替代类名，而用作类方法的接收者。当需要创建一个对象时，这很有用。但是，新对象的类在运行时才能够知道。例如，可以根据在配置文件中作为字符串存储的类型来创建一个新对象。

```
NSString *classNameAsString = ... // Read string from file
Class classToInstantiate = NSClassFromString( classNameAsString );
id newObject = [[classToInstantiate alloc] init];
```

在上面的代码中，NSClassFromString是一个常规的C函数，它接受一个NSString参数并且返回指向相应的类对象的一个指针。返回的指针存储在Class变量classToInstantiate中。最后，classToInstantiate用作类方法alloc的接收者，来创建请求的类的新对象。

类型内省

可以使用NSObject方法isMemberOfClass:和isKindOfClass:来询问Objective-C对象的类成员关系，如下所示：

⊖ 抱歉，不得不在这个句子中重复使用“类”这个词。这也令我头疼。

```
[mysteriousObject isMemberOfClass: [Dog class]];
```

如果mysteriousObject是Dog类的一个实例，那么上面的代码返回YES，否则的话，返回NO；而如下的代码：

```
[mysteriousObject isKindOfClass: [Dog class]];
```

在mysteriousObject是Dog类的一个实例，或者是继承自Dog的任何类的一个实例时，它都会返回YES。如果已经定义了一个RescueDog类，而它是Dog类的一个子类；而且如果mysteriousObject是Dog或RescueDog的一个实例，则上面的语句将会返回YES。

我们应该少依赖这些方法。如果你发现自己要使用其中的一个，应该三思而后行，看看有没有更好的办法来实现同样的目的。例如，如果你已经得到了一个类型为id的对象，并且想要找到其类，来看看给该对象发送一条特定的消息是否安全，那么应该使用respondsToSelector:来替代：

```
id mysteriousObject = ...
if ( [mysteriousObject respondsToSelector: @selector( rollOver )] )
{
  [mysteriousObject rollOver];
}
```

上面的示例测试了行为（对象是否响应你想要发送的消息）而不是类（不应该关注类）（参见本章末尾的练习2）。

7.1.2 类方法

类方法的一种描述方式是指在一个类的类对象上定义的方法。另一种描述方式是，类对象（或类名）不是类的一个实例，而是对应一个类方法的一条消息的接收者。还有如下的一些细节：

- 类方法的声明以一个加号开始。例如，alloc的声明是：

```
+ (id) alloc
```

- 类方法声明和实现分别位于类的@interface部分和@implementation部分，就像实例方法一样。它们通常放置在实例方法的前面，但是，这只是一个惯例而不是必需的。
- 类方法不能直接访问任何针对类的实例定义的实例变量。（如果它这么做的话，也不会有什么意义。假设你已经创建了一个类的4个实例，每个实例针对一个实例变量都拥有一个不同的值。如果类方法访问该实例变量，那么它应该获取哪一个值呢？）
- 和实例方法一样，类方法由子类继承。子类可以覆盖一个继承的类方法。例如，如果你创建一个新的类，那么你的类可以覆盖类方法description的NSObject实现，并且返回一个带有定制描述的字符串。
- 让一个类方法与该类的实例变量具有相同的名称，这也是可能的。运行时处理它，就好像处理任何其他的方法分派一样。消息发送给接收者。如果接收者是一个类对象，那么最终调用的是类方法。如果接收者是类的一个实例，那么调用的是实例方法。
- 编写类方法时需要记住的非常重要的一点是，类方法中的self引用类对象，而不是类的

实例。如果你忘记这一点，不仅很可能遇到bug或程序崩溃，而且还会很难找到引发这一问题的原因。

7.2 其他类方法

类对象的主要作用是，充当一个工厂对象来创建类的实例。正如我们在第6章中所见到的，使用类方法alloc来创建对象实例。下面介绍类常常定义其他类方法的领域。简便构造函数和单体是Cocoa框架常用的模式，并且，我们可能会在自己的类中使用它们。

7.2.1 简便构造函数

很多Cocoa框架类定义了简便构造函数，它将分配和初始化组合到一个单一的方法中。NSString类方法字符串就是一个典型的例子。它返回一个初始化的但是为空的字符串：

```
NSString *emptyString = [NSString string];
```

NSString提供了多种简便构造函数，如stringWithFormat:、stringWithString:和stringWithCharacters:length:（要获取完整列表，参见http: //developer.apple.com/mac/ library/documentation/ Cocoa/Reference/Foundation/Classes/NSString_Class/ Reference/NSString.html）。当一个类定义了多个不同的init方法时，它通常使用接受同样参数的一个简便构造函数来匹配每个init...方法。

在使用带有引用计数的简便构造函数时，有一件非常重要的事情需要注意，不能拥有一个简便构造函数所返回的对象（简便构造函数的名称，不包含“alloc”或“copy”，也不以“new”开头）。如果想要在当前环境之外使用从简便构造函数所接收到的一个对象（例如，通过将其存储到一个实例变量中），那么必须通过保留该对象来获得其所有权。如果不能获得所有权，则对象可能会在将来的某个时间消失，留给你一个指向已销毁的对象的指针。

考虑如下的示例：

```
NSArray *arrayOne =
  [[NSArray alloc] initWithObjects: @"iPhone", @"iPod", nil];
NSArray *arrayTwo =
   [NSArray arrayWithObjects: @"iPhone", @"iPod", nil];
```

你拥有arrayOne，是因为你使用alloc创建了它。当使用完一个arrayOne时，必须发送一条release消息来对应alloc。相反，你没有拥有arrayTwo，是因为你通过一个简便构造函数来获取它。如果需要arrayTwo在当前调用环境之外还保持存在，则必须向myArrayTwo发送一条retain消息（并且，当然，当使用完它时，需要一个对应的release）。

创建简便构造函数

假设有一个类Dog。我们可以创建一个简便构造函数，它返回Dog的一个初始化的实例，如下所示：

```
+ (id) dog
{
  return [[[Dog alloc] init] autorelease];
}
```

alloc使用值为1的引用计数来创建Dog的实例。autorelease完成了你要使用一条release来对应alloc的职责（autorelease在未来的某个时刻安排了一条release消息。参加下面autorelease的补充内容）。

自动释放

在引用计数下，当你使用名称中包含了“alloc”或“copy”，或者名称以“new”开头的方法来创建了一个对象时，返回的对象带有一个值为1的引用计数。你最终负责通过向新创建的对象发送一条释放消息来对应对象的创建。在某些情况下，这一需求会导致笨拙的问题。如果你有一个方法，它创建了一个对象可以作为返回值传递，那么使用一条释放消息来对应该对象的创建的唯一的地方，就是在该方法内部。但是，给这个新创建的对象发送一条平衡消息，将会导致它在返回之前销毁。需要一种方式来注册一次释放，从而延迟该释放的执行。这正是自动释放消息所做的事情。

当一个对象接收一条自动释放消息时，指向该对象的一个指针会被添加到当前的自动释放池中。当自动释放池在未来的某个时间用尽的时候，其中的所有对象都将接收一条释放消息。如果一个对象接收到多条自动释放消息，则它将会因为所接收的每一条自动释放消息而接收一条释放消息。在GUI程序中，池通常会在执行（事件）循环的底部耗尽。为了保留计数簿计数的目的，一条自动释放会统计为一条释放操作。它只是在未来某个未指定的时间发生的释放。第14章将详细介绍引用计数。

当使用自己的简便构造函数时，如果你想保留Dog，则必须记住获取返回的Dog的所有权：

```
Dog *spot = [Dog dog];
[spot retain];
```

实际上，dog对Dog类正确地工作，但是，如果子类化Dog，将会出现一个问题。如果创建一个Dog的子类RescueDog，则此子类将会继承简便构造函数dog。dog的当前版本无法对RescueDog正确工作。例如：

```
[RescueDog dog]
```

返回Dog的一个实例，而不是RescueDog的实例。可以使用self而不是Dog来作为alloc的接收者，从而修正这一问题：

```
+ (id) dog
{
  return [[[self alloc] init] autorelease];
}
```

由于dog是一个类方法，因此self引用作为接收者的类对象。修改后的简便构造函数对于Dog和RescueDog都能够正确工作。例如，

```
[Dog dog]
```

返回Dog的一个实例，并且，

```
[RescueDog dog]
```

返回RescueDog的一个实例。

注意 简便构造函数通常的返回类型是id，而不是特定类的一个实例，因此，它们可以由子类继承。

当返回的对象只打算在调用简便构造函数的环境中使用时，通常会使用简便构造函数。你会从简便构造函数获取一个对象，然后，将其用于该方法的剩余部分。由于不打算保留该对象，因此，不一定要保留它。当简便构造函数将该对象放入到一个自动释放池时，返回的对象的内存管理也受到关注。然而，如果对象很大，或者，如果你在一个iOS程序中遇到内存问题，那么更好的做法还是使用alloc...init序列来创建对象，然后，在用完对象之后，尽快地释放该对象。通过将该对象占用内存的时间最小化，从而将使用该对象的内存影响最小化。

7.2.2 单体

单体是一个只有单个共享的实例的类。单体类通常用来包装操作系统服务，或者用于检查面板这样的UI项。下面是一个单体的典型设置：

```
+ (InspectorPanel*) sharedInspectorPanel
{
  static InspectorPanel *inspectorPanel = nil;

  if ( ! inspectorPanel )
    {
      inspectorPanel = [[InspectorPanel alloc] init];
    }
  return inspectorPanel;
}
```

创建单个的共享的InspectorPanel实例，并且将其存储到一个静态变量中。当向类对象发送sharedInspectorPanel消息时，它返回存储在静态变量中的指针。

这段代码也说明了延迟实例化。在第一次请求共享的实例之前，没有创建它。这是Cocoa程序中一种常见的模式。延迟实例化减少了程序的启动时间。这样我们可以在需要的时候创建对象，而不是在程序启动的时候一次性创建它们。

注意 要了解有关设计模式的更多内容，以及如何使用它们来改进代码，参见Erik M. Buck和Donald A. Yacktman所著的《Cocoa Design Patterns》一书(Addison-Wesley, 2010)。

7.2.3 初始化类

Objective-C运行时使得我们可以在使用一个类之前先初始化它。当第一次向一个类对象发送消息时（通常是一条alloc消息来创建该类的第一个实例），运行时居中调停并且向该类对象发送一条initialize消息。该initialize消息在最初的消息之前发送。

注意 使用initialize消息来初始化类，与使用init...方法来初始化类的实例不同。

继承自NSObject的默认initialize方法什么也不做。大多数类不需要任何初始化，但是，在少数情况下，需要覆盖initialize：

- Foundation框架提供了一个单体类，NSUserDefaults，它管理程序与Mac OS X默认系统的交互（默认系统以每个用户为基础来管理用户偏好和其他值）。可以在一个initialize方法中使用共享的NSUserDefaults实例，针对类所使用的任何偏好项来设置默认值或获取当前值。
- 如果类使用外部变量来模拟类变量，可以做任何必要的工作，从而在initialize中初始化这些变量。

例如，假设有一个游戏，模拟一个商人团体管理一个公共资金库。Trader类自身使用一个外部变量来管理该资金库。可以使用NSUserDefaults系统在游戏会话之间存储资金库的值。当游戏重新启动后，使用Trader类的initialize方法从默认值恢复currentMoneyPool的值：

```
static NSInteger currentMoneyPool;

@implementation Trader
+ (void) initialize
{
  currentMoneyPool =
     [[NSUserDefaults standardUserDefaults]
         integerForKey: @"MoneyPool" ];
}

...

@end
```

当第一次向Trader类发送消息时，initialize获取资金库中存储的值，并且将其存储到外部变量currentMoneyPool中。

使用initialize的一个问题

使用initialize有一个潜在的问题需要注意。如果你有一个覆盖initialize的类，并且，该类有一个子类没有覆盖initialize，那么初始类将接收initialize消息两次：一次是初次使用初始类时，另一次是初次使用该子类时。这可能不是你想要的行为，因此，你可以如下这样编写initialize方法，将初始化限制于初始类（称其为Dog）：

```
+ (void) initialize
{
  if ( self == [Dog class] )
    {
      // Do the initialization for Dog
    }
}
```

注意 理论上讲，在前面的代码中，可以省略if语句中的检查，并且只要求Dog的所有子类实现initialize方法。但是，这是一个糟糕的想法。这会使得Dog类的正确工作要依赖于其所有子类的正确编码。如所示的那样编写Dog类，可以确保其初始化只发生一次，即便是子类没有实现initialize。

7.3 模拟类变量

正如本章前面提到的，Objective-C没有类变量。大多数时候，可以在类的实现文件中使用普通的文件作用域变量来模拟类变量。然后，可以定义类方法，它们不需要你拥有该类的实例就可以获取或设置外部变量的值。

为了说明这一点，在程序清单7.1和程序清单7.2展示了如何创建一个Coupon类，它只能发出固定数目的优惠券：

程序清单7.1 CouponDispenser/Coupon.h

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

@interface Coupon : NSObject
{
   ...
}

+ (int) numberCouponsLeft;
+ (void) resetCoupon;

@end
```

程序清单7.2 CouponDispenser/Coupon.m

```
#import "Coupon.h"

#define INITIAL_COUPON_ALLOCATION 100

// The static keyword here limits the visibility
// of availableCoupons to this file

static int availableCoupons = INITIAL_COUPON_ALLOCATION;

@implementation Coupon

-(id) init
{
  if (self = [super init] )
  {
    if ( availableCoupons == 0 )
      {
        [self release];
        return nil;
      }
    availableCoupons--;
  }
return self;
```

```
}

+ (int) numberCouponsLeft
{
  return availableCoupons;
}
+ (void) resetCoupon
{
  availableCoupons = INITIAL_COUPON_ALLOCATION;
}

...

@end
```

availableCoupons记录了Coupon类可用的优惠券的总数目。每次调用init时，它检查availableCoupons，得到剩余的优惠券的数目。如果availableCoupons是0，则init释放self并返回nil。否则，init将availableCoupons减少1并且返回优惠券。

这种安排的弱点在于，它不能正确地处理继承。如果你盲目地子类化Coupon，那么Coupon及其子类（名为NewCoupon）将共享相同的静态变量availableCoupons。如果产生几个子类，那么它们都与最初的Coupon类共享这一变量。这可能不是你想要的结果。Coupon和NewCoupon需要保持它们各自可以发布的优惠券的总数。可以修正这一问题，但这需要一些额外的手动操作：

1) 为Coupon和NewCoupon分别声明单独的静态变量availableCoupons和availableNewCoupons。这使得每个类都有了自己的可用优惠券的数目。

2) 在两个实现文件中，都实现如下的类访问器：

```
+ (int) availableCoupons;

+ (void) setAvailableCoupons:(int) inAvailableCoupons;
```

在Coupon中实现的访问器访问availableCoupons。在NewCoupon中实现的访问器（它覆盖了Coupon中的那些实现）访问availableNewCoupons。

3) 在该类及其子类中，重新编写使用类访问器来访问优惠券数目的任何方法（类方法或实例方法）。在实例方法中，访问器应该如下这样调用：

```
[[self class] availableCoupons];

[[self class] setAvailableCoupons: newNumber];
```

这确保了调用访问器的正确版本（在一个类方法中，不需要使用[self class]，因为在这种情况下，self已经引用了正确的类对象，即用作接收者的那个类对象）。

程序清单7.3展示了修改后的版本。对代码的修改用粗体显示。只要你遵守前面的步骤，它就可以子类化。它在一个扩展中为静态变量声明了访问器。它们仅供内部使用，并且在公共头文件中应该是不可见的。

注意 扩展（第11章将要介绍）将方法的声明移动到了实现文件中。它们提供了一种方式，将那些专门供内部使用的方法的声明，隐藏起来以使得公共视野看不到。扩展的接口部分以@interface开始，在同一行中后面跟着所扩展的类的名称，以及一对空的圆括号。它通常以@end结束。在扩展中声明的方法，在类的@implementation部分以一般的方式实现。

程序清单7.3 CouponDispenserFixed/Coupon.m

```
#import "Coupon.h"

#define INITIAL_COUPON_ALLOCATION 100

// The static keyword here just limits the scope of availableCoupons
// to this file

static int availableCoupons = INITIAL_COUPON_ALLOCATION;

// Declare private accessors in an extension

@interface Coupon ()

+ (int) availableCoupons;
+ (void) setAvailableCoupons:(int) inAvailableCoupons;

@end

@implementation Coupon

+ (int) availableCoupons
{
  return availableCoupons;
}

+ (void) setAvailableCoupons:(int) inAvailableCoupons
{
  availableCoupons = inAvailableCoupons;
}
- (id) init
{
  if (self = [super init] )
    {
      if ( [[self class] availableCoupons] == 0 )
        {
          [self release];
          return nil;
        }
```

```
        [[self class] setAvailableCoupons:
            [[self class] availableCoupons]-1 ];
    }
  return self;
}

+ (int) numberCouponsLeft
{
  return [self availableCoupons];
}

+ (void) resetCoupon
{
  [self setAvailableCoupons: INITIAL_COUPON_ALLOCATION];
}

@end
```

程序清单7.4和程序清单7.5给出了NewCoupon子类。给定了NewCoupon其自身的静态变量availableNewCoupons，并且覆盖了私有访问器方法availableCoupons和setAvailableCoupons:以引用availableNewCoupons，从而让Coupon和NewCoupon对于最大的优惠券数目都拥有独立的值。

程序清单7.4 NewCoupon.h

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


@interface NewCoupon : Coupon
{

}

@end
```

程序清单7.5 NewCoupon.m

```
#import "NewCoupon.h"

#define INITIAL_NEW_COUPON_ALLOCATION 50

static int availableNewCoupons = INITIAL_NEW_COUPON_ALLOCATION;

@interface NewCoupon ()

+ (int) availableCoupons;
```

```
+ (void) setAvailableCoupons:(int) inAvailableCoupons;

@end


@implementation NewCoupon

+ (int) availableCoupons
{
  return availableNewCoupons;
}

+ (void) setAvailableCoupons:(int) inAvailableCoupons
{
  availableNewCoupons = inAvailableCoupons;
}


// This method overrides the superclass implementation so each
// class can have its own value for the initial allocation.

+ (void) resetCoupon
{
  [self setAvailableCoupons: INITIAL_NEW_COUPON_ALLOCATION];
}
...

@end
```

在程序清单7.3和程序清单7.5中，Coupon和NewCoupon分别拥有各自的静态变量，availableCoupons和availableNewCoupons。每个类也都有类访问器方法availableCoupons和setAvailableCoupons:的自己的实现，以访问类的静态变量。

在程序清单7.3中，init通过调用类访问器方法availableCoupons和setAvailable-Coupons:，而不是直接访问静态变量，来获取和设置剩下的优惠券的数目。构造：

```
[[self class] availableCoupons]
```

确保了对于下面二者都调用availableCoupons的正确版本：

```
[[Coupon alloc] init]
```

和：

```
[[NewCoupon]alloc] init]
```

7.4 小结

Objective-C类是对象并且拥有方法，其方法叫做类方法。对于类对象和类方法，需要记住的关键内容是：

- 类方法以+而不是-开头的声明来表示。
- 调用类方法时，使用类名或指向类对象的一个指针作为消息接收者。
- 指向类对象的指针可以存储在类型为Class的变量中。
- 类对象没有实例变量，并且，它们不能使用该类的实例的实例变量或方法。
- 最常见的类方法是alloc，它是一个工厂方法，可以创建该类的实例。
- 类方法的第二种最常见的用法是，简便构造函数。简便构造函数返回类的初始化后的实例。
- 类变量的功能可以使用静态变量来复制，但是，要将使用外部变量的一个类子类化，需要额外的工作。

7.5 练习

1. 创建一个类Train及初始类的子类BulletTrain（为了本练习的目的，这个类不做任何事情）。
   - 为Train编写一个简便构造函数，它将由BulletTrain继承。
   - 编写一个小程序，它获取Train的一个实例及BulletTrain的一个实例，在两种情况下都使用简便构造函数。
   - 记录返回的实例，以证明你的简便构造函数能够正确地工作。
2. 假设你有一个抽象类Dog，及其子类AmericanDog、JapaneseDog和ItalianDog。要求你编写一小段代码，它记录Dog的吠声。下面的代码有什么错误（这是一个设计问题，而不是一个语法问题）：

```
Dog *myDog = ...

if ( [myDog isMemberOfClass: [AmericanDog class]] )
  {
    NSLog(@"woof woof");
  }
else if ( [myDog isMemberOfClass: [JapaneseDog class]] )
  {
    NSLog(@"wan wan");
  }
else if ( [myDog isMemberOfClass: [ItalianDog class]] )
  {
    NSLog(@"bau bau");
  }
```

   如何改进这一设计？如果你添加一个FrenchDog子类，那么代码（旧版本和新版本）如何修改？
3. 使用Coupon类的第一个版本编写一个程序（参见程序清单7.1和程序清单7.2）。
   - 创建子类NewCoupon的一个简单版本（没有程序清单7.3、程序清单7.4和程序清单7.5中给出的改进）。
   - 在程序中，获取Coupon的一个实例，然后显示剩下的Coupon和NewCoupon的数目。
   - 然后，获取NewCoupon的一个实例，再次显示剩下的Coupon和NewCoupon的数目。

- 验证类Coupon和NewCoupon共享单个的剩余优惠券的数目。

4. 在练习3的程序中，用正确的版本（参见程序清单7.3、程序清单7.4和程序清单7.5）来替代Coupon和NewCoupon的版本。验证类Coupon和NewCoupon现在分别记录了各自剩余的优惠券数目。
5. 获取练习4的程序，并且添加一个新的优惠券类型NewCoupon2，它是NewCoupon的一个子类。验证Coupon、NewCoupon和NewCoupon2都各自拥有剩余优惠券的独立版本。
6. 获取练习5的程序，并且将保存剩余优惠券数目的静态变量的初始化，放入到initialize方法中。对于每个类，你都需要一个initialize方法（在简单赋值的情况下，这么做没有好处，但是，在实际程序中，你可能使用initialize方法从默认系统或其他的一些磁盘资源中读取优惠券的初始数目）。

第8章 框 架

假设你有很好的木匠技能，并且需要构建一座房屋。你可能会带着自己的工具走进森林中，砍伐树木，砍伐木材，并且动手从原材料开始制作需要的所有东西，以便盖好房子。最终的小屋可能是工艺中的精品，但是，它很难以一种高效的方式成为有用的居住地。一种更高效的过程，可能是找到当地的建材供应商店，并且找到最好的一套预先制作好的门、窗户和其他部件。Objective-C相当于当地建材供应商店，它提供一组框架。框架是预定义的类和有用的函数的一个库，你可以用它们来构建自己的应用程序。它们相当于Java或C++的类库。

框架的一个重要属性是，它们动态地载入共享的资源。框架所包含的可执行代码或资源，只有一个副本需要在系统上呈现。不论何时，只有一个副本载入到内存中并且在需要它的所有应用程序中共享。尽管现在看来，磁盘空间似乎越来越不是一个问题，但内存仍然很宝贵。

如果你构建一个GUI应用程序，为你的UI使用框架对象，这将在你的UI和其他应用程序的UI之间提供很大程度的一致性。这对用户很有帮助，因为通过其他应用程序所获得的体验也带入到了你的应用程序中。

本章介绍什么是框架，以及如何对你的程序使用框架。然后，继续对你在编写Mac OS X或iOS程序时可能遇到的一些框架进行更深入的介绍。第9章将详细地深入到`Foundation`框架。`Foundation`框架定义了`NSObject`，这是Objective-C的对象的根类，该类用于字符串、数组、集合、字典和数字对象及包装了与OS交互的对象。

8.1 什么是框架

框架是包的一种类型，它是带有特定布局的一个目录层级，这些特定布局用来将共享的动态库、头文件和资源（图像、声音和nib文件）组织到一个地方。进行Mac OS X和iPhone OS开发所需的共享动态库都包装为框架。框架包的名称总是存在`frameworkName.framework`的形式。

框架可以包装以任何语言编写的动态库，但是，我们最常遇到的框架都包含Objective-C或普通C接口。

注意 *在Mac OS X和iOS编程中，框架很常见，但是，也可以连接到程序所需的任何常规的静态或动态库。*

使用框架

要使用Objective-C或C框架，必须为将要在自己代码中使用的任何框架定义的类或函数导入头文件。大多数框架都有一个组头文件，它为框架导入所有单个的头文件。一些框架是伞状框架。伞状框架是包含了两个或多个其他框架的框架。

如下的代码为Foundation框架中的类导入所有相关的头文件：

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

组导入不会导致膨胀问题，它只是使得框架中的所有类可用。导入正确的头文件，编译器就满意了。还必须让连接器满意，要告诉连接器，对于添加到其列表中的框架，要到哪里去查找可执行的代码。当创建项目时，Xcode（根据项目类型）自动把需要的基本框架添加到项目中。要向项目中添加框架，找到Xcode项目窗口的Groups and Files部分，并且用鼠标右键（control-click）在External Frameworks and Libraries文件夹上单击。然后，在如图8-1所示的弹出菜单中选择Add > Existing Frameworks。

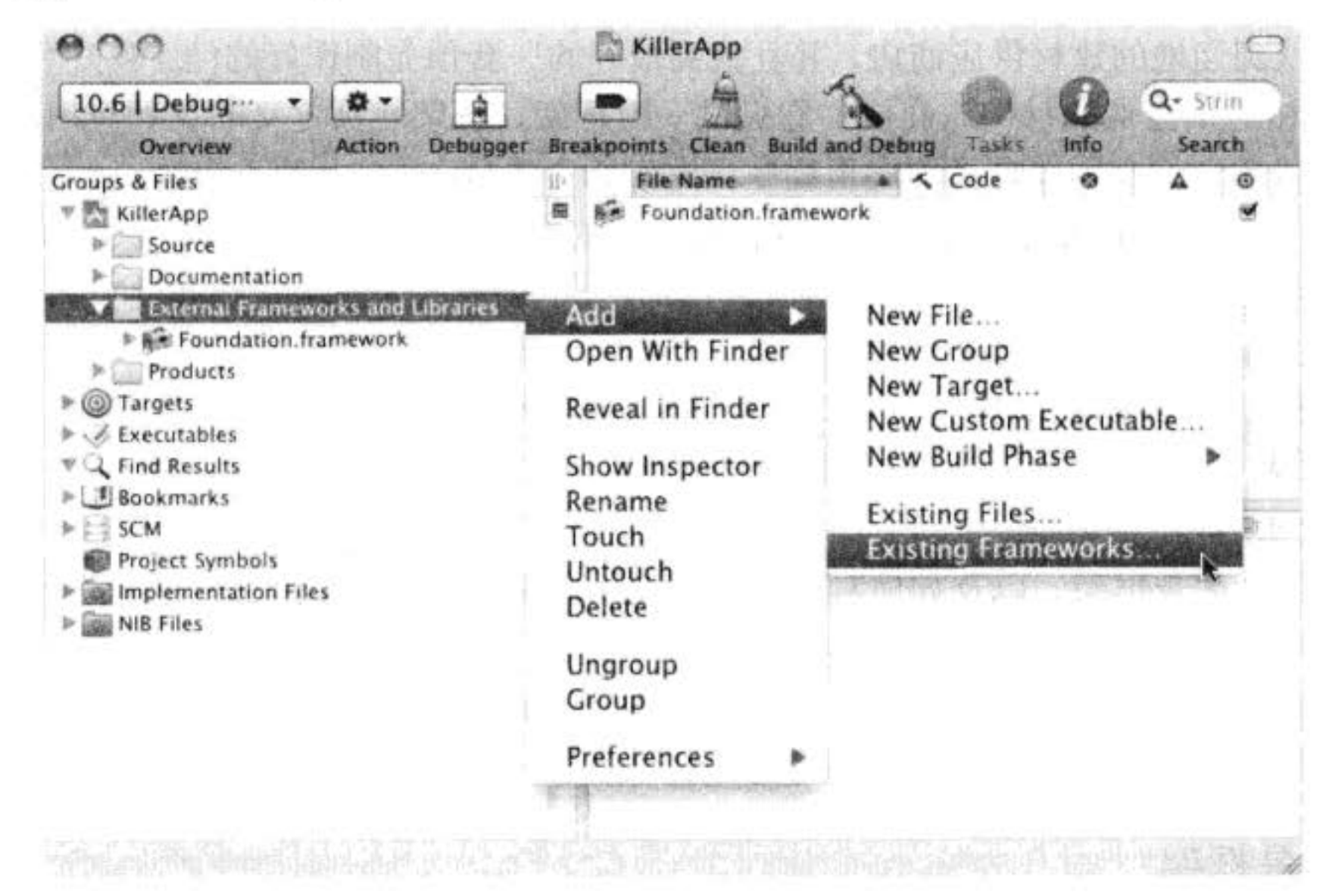


图8-1 向一个Xcode项目添加一个框架

如果要使用GNU编译器gcc从命令行编译自己的程序，则必须将要使用的任何框架的名称传递给gcc。这可以通过一个-framework标志，后面跟着不带.framework扩展的框架的名称来完成。例如：

```
gcc myProgram.m -framework Foundation
```

如果要使用多个框架，则必须在gcc命令行中列出所有的框架，每一个框架都带有其自己的-framework标志。

```
gcc myProgram.m -framework Foundation -framework CoreFoundation
```

8.2 Cocoa框架

Cocoa是Apple用来表示用Objective-C技术编写的Mac OS X程序的名称，Apple从NeXTStep购买来该技术。Cocoa框架是一个伞式框架，它包含了用来构建Cocoa应用程序的3个

主要框架：

- Foundation框架：Foundation包含了几乎任何应用程序都需要的基本对象。第9章将介绍一些重要的Foundation类。
- AppKit框架：AppKit是Application Kit的缩写。AppKit包含了在OS X上构建GUI应用程序所需要的类。
- Core Data: Core Data是最新的添加（从Mac OS X 10.4开始）。它是管理持久性对象集合（也就是说，在程序调用之间存储在磁盘之外）的一个框架。

可以通过导入获得所有这3个框架的头文件：

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

如果编写一个命令行程序，它没有使用Core Data或AppKit，则可以只导入Foundation组头文件：

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

Objective-C语言和Cocoa框架之间的关系类似于Java语言和Java类库之间的关系。语言自身是紧凑的，相对容易学习，但是，实际的工作是学习库的细节。如果只是学习Objective-C就能够使你成为一名称职的Cocoa程序员，那真是不错，然而遗憾的是，这只是第一步。在掌握了本书的内容之后，你需要花相当多的时间来研究Cocoa框架的文档，或者阅读一本类似《Cocoa Programming for Mac OS X》（Aaron Hillegass著）这样的图书，[⊖]然后，才能创建自己的重量级应用程序。

iPhone

在iPhone上，UIKit框架替代了AppKit并且提供了组成iPhone的用户界面的类。导入UIKit的头文件，会自动导入Foundation头文件，以及用于UIKit的头文件：

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

UIKit和Foundation合称为Cocoa Touch。

注意 尽管Cocoa Touch的名称与Cocoa类似，但实际上没有Cocoa Touch伞状框架，也没有CocoaTouch.h头文件。

8.3 AppKit

AppKit是一个很大的框架，其中有将近200个用于构建Mac OS X GUI应用程序的相关的类。对AppKit的详细介绍超出了本书的范围，但是，对于AppKit的一些主要类，本节会给出一个简单的介绍。

其最简单的形式就是一个GUI应用程序在屏幕上的一个区域或多个区域显示一些信息（文本、图像）。然后，用户使用鼠标、键盘或一些其他的输入设备来与该应用程序交互。操作系统接收这些交互，将它们打包到一个叫做事件的容器中，然后，将事件转发给该应用程序。随

⊖ Aaron Hillegass,《Cocoa Programming for Mac OS X, 3rd Ed》(Addison-Wesley, 2008)。

后，应用程序使用事件来修改其内部状态并将修改显示回屏幕上。

要编写一个GUI应用程序，还需要了解其他的一些AppKit类：

- NSApplication是一个Cocoa GUI应用程序的中心类，每个应用程序都必须包含其一个单个的实例。任何所需的定制都通过创建一个单独的对象并将其赋值为NSApplication的代理来处理，而不是子类化NSApplication。
- NSWindow表示屏幕上的一个窗口。
- NSView表示窗口的一个区域。大多数窗口都有NSView层级。
- NSEvent携带了用户与应用程序的交互信息：事件的类型、屏幕上发生事件的位置；还有一个时间戳，记录了事件何时发生，以及在事件中是否按下了一个修改键。
- NSButton、NSSlider、NSMenu、NSSegmentedControl和NSColorWell这样的控件，为用户提供了与应用程序交互的一种方法。
- NSDocument是用来构建这样的应用程序的一个主要的类：用户可以在应用中同时打开或使用多个数据集。

AppKit还提供了Objective-C类来包装很多C语言Core Graphics的低层级功能。可以使用AppKit类来绘制矢量图形(NSBezierPath)或位图图像(NSImage)。

很多AppKit类都相当复杂，并且有拥有众多的方法。例如，Mac OS X所带的TextEdit程序的很多编辑功能都直接来自于类NSTextView（这是NSView的一个子类）。可以在Xcode工具所提供的示例中，看到TextEdit程序的完整源代码（/Developer/Examples/TextEdit）。

8.4 Core Foundation

Core Foundation是一个低层级的C语言框架，（部分地）与Objective-C Foundation框架并行。它定义了数据类型，并且提供了很多其他C语言框架所使用的服务。在这里介绍Core Foundation，是因为有时候在Cocoa或Cocoa Touch程序中必须使用Core Foundation类型。如果你的程序使用其他低层级C框架，例如Core Graphics（Core Graphics将在8.5节介绍），往往也必须使用Core Foundation。

尽管Core Foundation是用纯C编写的，它也被组织到了对象中。一些常见的Core Foundation类型有：CFString、CFArray、CFMutableArray、CFDictionary、CFMutableDictionary、CFData和 CFNumber。

提示 尽管C不是一种面向对象语言，但它可以用来实现一个定制的对象系统。

使用不透明指针（参见后面补充内容“不透明指针”）来访问Core Foundation对象。为Core Foundation对象声明的指针，通过使用类型名并将Ref添加到末尾的typedef来隐藏。CFMutableArray*的声明由typedef CFMutableArrayRef隐藏。

作为一个Core Foundation对象的示例，如下的代码行创建了一个Core Foundation可变数组：

```
CFMutableArrayRef cfMutableArray =
    CFArrayCreateMutable ( CFAllocatorRef allocator,
```

```
                CFIndex capacity,
                const CFArrayCallBacks *callBacks
               );
```

allocator参数指定了对象内存的动态分配如何处理，capacity是数组中最大的条目数，callbacks是指向回调函数的结构的一个指针。当对象存储到数组中或从数组中删除的时候，回调函数用来执行类似保留和释放对象的任务。

现在不要关心这些细节，上面代码的主要目的是展示一个Core Foundation对象的实际例子。

比较Core Foundation可变数组和创建等价的Foundation对象：

```
NSMutableArray *nsMutableArray =
  [[NSMutableArray alloc] initWithCapacity: capacity];
```

要注意的事情是，Core Foundation对象比其等价的Foundation更加复杂和难以使用。可能有时候你被迫会使用一个Core Foundation对象，但是，可能的话，停留在Objective-C级别是一种更容易且较少出错的编码方式。

> **不透明指针**
>
> Core Foundation对象引用是不透明指针。不透明指针指向一个结构，结构的声明放在一个私有头文件中，而该文件没有提供给用户。如果没有头文件，则只能使用一组提供的函数来访问结构成员。不能通过将指向该结构的指针解引用来直接访问结构的内容，也不能在调试器中询问对象。不透明指针的关键在于信息隐藏。不透明指针允许代码的所有者修改对象的（私有的）内部实现，而不用修改其公有接口。Core Foundation也提供了一个函数CFShow，它可以在调试器中用来显示关于Core Foundation对象的信息，但是，由于使用不透明指针，那些使用Core Foundation对象的代码的调试过程可能会令人沮丧。

8.4.1 Core Foundation对象的内存管理

Core Foundation对象有一个引用计数内存管理系统，它类似于Objective-C的引用计数：

- 通过调用名称中有Create或Copy的Core Foundation函数所创建的任何Core Foundation对象，你都可以拥有它。
- 不能拥有从任何其他函数（通常是其名称中某处带有Get的Core Foundation函数）得到的任何Core Foundation对象。如果想要让该对象在当前作用域之外仍然保持有效，则必须以该对象作为函数参数来调用CFRetain函数，从而获得该对象的所有权。
- 当用完了所拥有的Core Foundation对象时，必须通过使用该对象作为函数参数来调用CFRelease函数，以放弃对该对象的所有权。

Objective-C的release和retain方法和任何其他Objective-C方法一样，以一个nil为接收者调用retain或release不会有任何操作。然而，使用一个NULL参数调用CFRetain或CFRelease将会导致程序崩溃。如果使用CFRetain或CFRelease，首先确保检查非NULL参数。

注意 诸如Core Graphics（将在8.5节介绍）这样的其他低层级C框架，和Core Foundation有着相同的对象和内存管理系统。

8.4.2 自由转换

某些Foundation类几乎有着与对应的Core Foundation类相同的内存布局。对于这些类，指向Foundation对象的指针也可以用来引用对应的Core Foundation对象，反之亦然。所需要做的只是相应的强制转型。Apple将这种功能称为自由转换（toll-free bridging）。程序清单8.1给出了一个使用NSString和CFString自由转换的示例。

程序清单8.1 TollFreeBridge.m

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

int main( void )
{
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

  // Create an NSString

  NSString *nsString = @"Toll-Free";

  // Cast nsString to a CFString and get its length with a CF function

  CFStringRef cfString;
  cfString = (CFStringRef) nsString; // Cast it to a CFString
  int length = CFStringGetLength ( cfString );
  NSLog( @"CFString length is: %d", length );

  // Create a CFString
  CFStringRef cfString2 =
           CFStringCreateWithCString( NULL,
                                      "Bridge",
                                      kCFStringEncodingASCII);

  // Cast cfString to an NSString and log it

  NSString *nsString2;
  nsString2 = (NSString*) cfString2;
  NSLog( @"NSString is: %@", nsString2 );

  [pool drain];
}
```

上面程序的运行结果是：

```
CFString length is: 9
NSString is: Bridge
```

附录B给出了一个自由转换类的完整列表。

8.5 Core Graphics

Core Graphics是用于Quartz 2D图形的低层级API。和Core Foundation一样，它是面向对象的，但是，它是直接用C编写的。Quartz实现了一个设备独立的2D图像模型，它几乎与PDF（之前的PostScript）图像模型相同。这里的设备独立意味着框架中的所有位置和维度引用一个确定的浮点坐标系统，而不是引用设备像素。最终，大多数图形都转换为像素，但是，除非你创建用于输出的位图，否则极少情况下是必须直接考虑像素的。

Core Graphics为设置颜色、透明度和混合；创建路径（线条和贝塞尔曲线）；描边和填充路径；创建和绘制图像和PDF；绘制文本及裁剪提供了方便。它还管理颜色空间和当前绘制坐标系统。

Core Graphics的大多数功能已经包装到了Objective-C对象中，并且在AppKit框架中可用。但是，并非所有的都包装了。如果你要进行更高级的绘制，可能需要直接深入到Core Graphics中。好在可以在AppKit层级对象消息和Core Graphics函数调用之间轮流使用，而没有什么问题。尽可能在AppKit层级做更多事情，然后，到Core Graphics中做剩下的事情。

在iPhone上的情况有所不同。UIKit包装了Core Graphics的较少的模块。使用AppKit可以绘制的一些内容，如果想要将其绘制到iPhone上，则需要使用Core Graphics。

注意 David Gelphman和Bunny Laden编著的《Programming with Quartz: 2D and PDF Graphics in Mac OS X》一书(San Francisco: Morgan Kaufmann, 2006)详细地介绍了Core Graphics。

8.6 Core Animation

给你的应用程序添加动画，也给了你成为“超越迪斯尼”的迪斯尼的机会。一些动画一直是Mac OS X的一部分。将一个应用程序从dock拖出时的“冒烟”效果，以及将窗口最小化到dock上的“扭曲”效果，都已经存在多年了。但是，从Leopard和iPhone开始，很多之前静止的内容现在开始变得动了起来。这些动作都是源于Core Animation的作用，它是在实现动画方面承担了很多工作的一个Objective-C框架。最简单的动画，你只需要指定要对什么内容实现动画，要让其去往何处（或者，如果你要对一个属性实现动画的话，想要其最终值是多少），并且告诉Core Animation去这么做。Core Animation负责处理剩下的事情。

注意 要深入了解Core Animation，参阅Marcus Zarra和Matt Long所著的《Core Animation: Simplified Animation Techniques for Mac and iPhone Development》一书(Addison-Wesley, 2009)。

8.7 其他Apple提供的框架

Apple提供了很多其他的框架，有高层级的也有低层级的。其中，你可能需要用到的一些是：

- `WebKit`：这是一个Objective-C框架，用来向应用程序添加基于Web的内容和浏览器功能。
- `ImageIO`：一个C语言框架，用来读取和写入图像文件格式，它所支持的格式比你已经知道的还要多。
- `Core Image`：一个Objective-C框架，用来使用GPU进行快速图像处理。
- `Core Audio`：一个C语言框架，用于音频处理。

尽管下面这些框架并非是由Apple编写的，但是Mac OS X和iOS还是提供了它们：

- OpenGL（Mac OS X）和OpenGL ES（iOS）：C语言框架，用于实现快速的、硬件加速的3D图形。
- OpenAL：一个C语言框架，用于开源的OpenAL音频库。

注意 可以通过目录/System/Library/Frameworks查看更多Mac OS X上可用的框架。

8.8 第三方框架

一些公司和个人开发者提供Objective-C或C框架来执行特殊的任务。这些框架要么是商业许可的，要么是不同程度的“免费”框架。可以与使用Apple提供的任何框架同样的方式来使用这些框架。将其添加到Xcode项目的项目窗口中的Frameworks目录下，就可以连接到第三方框架。

这里只有一点比较复杂，如果你将应用程序连接到一个框架，那么应用程序在任何系统上运行时，该系统都必须有一个该框架的副本可用。这对于Apple提供的框架来说，是不成问题的。那些框架都保证可以安装到任何有效的Mac OS X系统上。但是，对于第三方框架来说，情况并非如此。

如果OS不能够找到一个应用程序所使用的框架的副本，那么该应用程序在启动时将会崩溃：

```
Dyld Error Message:
  Library not loaded:
@executable_path/../Frameworks/VitalStuff.framework/Versions/A/VitalStuff
  Referenced from: /Applications/KillerApp.app/Contents/MacOS/KillerApp
  Reason: image not found
```

为了防止程序崩溃，必须向该应用程序的Xcode项目添加一个复制阶段，以便将任何第三方框架复制到应用程序包的Frameworks目录下。

注意 如果创建了使用相同框架的多个应用程序，应用程序的安装过程应该会将框架复制到/Library/Frameworks下，以便可以共享它。

也可以创建自己的框架。如果你开发一组类，打算在多个应用程序中使用它们。你可能会设计、编写并测试类，然后使用Xcode将其构建到一个框架中。在构建了你的框架后，可以像使用任何第三方框架一样使用它。向应用程序的Xcode项目添加新的框架，并且确保将框架复制到应用程序的包中。

8.9 幕后揭秘

从技术上讲，框架是版本化的包。一个版本化的包的目录结构设计为使得内容的不同版本可以共存。这允许框架的维护者向框架添加内容，但是，旧的代码仍然可以继续使用框架最初的版本。如下是一个典型的框架目录的结构：

```
SomeFramework.framework/
    SomeFramework@ -> Versions/Current/SomeFramework
    Headers@ -> Versions/Current/Headers
    Resources@-> Versions/Current /Resources
    Versions/
        A/
            SomeFramework*
            Headers/
            Resources/
        Current@ -> A
```

上面例子中的框架仍然是初始（A）版本。顶级的条目SomeFramework、Headers和Resources是最终导向目录的当前版本的符号链接。

注意 在前面的列表中，符号链接通过目录名末尾的@符号来表示。包含了可执行代码的文件用一个结尾的星号(*)表示：

```
SomeFramework*.
```

大多数时候，不必深入到一个框架的目录中，但是，看看框架的头文件往往是有用的。如果你的文档配备了一个不是很清晰的框架，就更应该看它的头文件了（或者，第三方框架没有为你提供文档）。

一些框架是伞状框架，即包含其他框架的框架。本章前面介绍的Cocoa框架，就是伞状框架的一个例子。伞状框架提供的间接的层级允许其维护者能够自由地识别框架的子框架，而不会打扰到框架的用户。

在/System/Library/Frameworks目录下可以找到Apple提供的框架。应用程序所安装的供一个系统上的所有用户使用的框架，可以在/Library/Frameworks目录下找到。应用程序也可以包含供他们自己使用的私有框架。这些框架可以在位于应用程序包MyApp.app/Contents/Frameworks中的Frameworks目录下找到。

Mac OS X和iOS还有很多私有框架供内部使用。如果你发现了这些框架，没有什么理由会阻止你使用它。然而，使用私有框架是一种糟糕的想法。它们可能会在未来的OS发布时修改或消失。此外，在iOS上，使用私有框架将会导致Apple的AppStore拒绝你的应用程序。

8.10 小结

Frameworks是提供对动态库的可执行代码和资源的方便访问的包（目录层级）。所有的Objective-C程序都使用Foundation框架，它定义了任何非一般程序所需的基本的类。Mac OS X GUI程序使用AppKit框架的类构造它们的UI，iPhone UI使用UIKit框架的类。Apple为音频和图形这样的特殊任务提供了很多其他的框架。其中的一些是C语言框架，它们使用Core Foundation框架中定义的对象系统和内存管理函数。

第9章　常用Foundation类

Objective-C是一种非常紧凑的语言。Objective-C的大多数功能位于其相关联的类库中，也称为框架。第8章介绍了框架的一般性主题。在本章中，我们将介绍Foundation框架中的几个类。

Foundation框架为字符串、数组、字典和数字对象这样的基本实体提供了类。尽管可以编写一个不使用Foundation框架的类的程序，但是，使用Foundation类而不是你自己开发的类将会缩短开发时间并减少bug。

本章首先介绍拥有可变和不可变版本的Foundation类及实现为类簇的类的一般性信息。然后，继续概括介绍一些应用最广泛的Foundation类：NSString、NSArray、NSDictionary、NSSet、NSData，及其可变子类NSNull、NSNumber和NSURL。最后介绍Foundation定义的一些非对象的结构：NSRange、NSPoint、NSSize和NSRect。

注意　可以从http://developer. apple.com/mac/library/ documentation/Cocoa/ Reference/ Foundation/ObjC_classic/找到Foundation类及其文档的一个完整列表。

9.1　可变类和不可变类

Foundation类是其他的Objective-C对象的容器，它以两种不同的形式用于字符或用于字节：可变的和不可变的。基本的类是不可变的。当你创建了一个不可变的类的实例之后，不能修改其内容。例如，如果你创建了NSArray（它是不可变的）的一个实例，并且想要给该数组添加一个对象，必须创建一个新的数组，其中包含了额外的对象，并且释放最初的数组。可变的容器类都是其对应的不可变类的子类。

为容器分别创建可变类和不可变类，这考虑到了如何实现不可变类的幕后的一些效率。当你创建一个不可变的数组时，它所保存的对象的数目在创建时是固定的。对象的实际存储可能是内存的一个简单的、固定大小的块。可变的数组必须使用一个更加复杂的存储系统，它允许其内容增加或减少。本章要介绍的成对的可变的类和不可变的类是：NSString和NSMutableString、NSArray和NSMutableArray、NSDictionary和NSMutableDictionary、NSSet和NSMutableSet、NSData和NSMutableData，以及NSIndexSet和NSMutableIndexSet。

9.2　类簇

类NSString、NSArray、NSDictionary、NSSet、NSNumber和NSData都实现为类簇。类簇是将复杂性隐藏到一个简单的接口的后面的一种方式。在类簇中，公有可见的类是一个抽象类。类的实际是由隐藏在公有的可见类后面的一组私有“实体”子类提供的。私有子类

都实现了公有的可见类所定义的接口，但是，它们的内部实现可能各有不同，从而优化了存储和性能。当请求公有类的实例时，我们接收类簇的私有子类的实例。由你所使用的init方法来做出选择。

注意 要详细了解类簇，参见Erik M. Buck和Donald A. Yachtman所编著的《Cocoa Design Patterns》一书(Addison-Wesley, 2010)。

将一个类实现为类簇会有以下几个方面的结果：

- 实现为簇的一个类的init方法，用相应的具体子类的一个不同对象，替代了alloc所返回的对象。簇的alloc方法返回了属于特定的占位类的一个临时对象，init方法释放了该临时对象，并且返回了簇的具体子类之一的一个实例。这一后果有一个重要的隐含意义：你不应该保留alloc所返回的对对象的引用。将alloc和init编写为一条单个的嵌套语句：

```
SomeClassCluster *classCluster = [[SomeClassCluster alloc] init];
```

正如第6章提到的，嵌套的alloc、init是创建对象的标准惯例。如果你习惯于使用这一标准惯例，那么不必考虑这一问题。

- NSObject内省方法isMemberOfClass:不应该用在实现为类簇的一个类的实例之上。

```
NSArray *array = [NSArray arrayWithObject: @"Foo"];
BOOL test = [array isMemberOfClass: [NSArray class]];
```

在执行了上面的代码行之后，test是NO，因为array是NSArray的一个私有子类，而不是NSArray的一个实例。

- 将实现为类簇的一个类子类化，这是比子类化一个简单类更为复杂的过程。这是一项超出本书讨论范围的高级技术。

这个类是一个类簇吗？

如何知道一个给定的类实现为类簇了？唯一的方式是，阅读该类的文档。可能应该有如下所示的一个类方法：

```
+(BOOL)[SomeClass isClassCluster]
```

然而，目前还没有这样的一个方法可用。

9.3 NSString

普通C中的字符串只是表示字符的字节的一个数组，后面跟着一个NULL（零）字节。NULL字节是确定字符串在哪里结束的一种方便的方式。C字符串没有其他结构。

C字符串对于简单任务来说，表现得很好，但是，它缺乏任何实际的结构，这使得它们可能导致各种问题和错误——缓存溢出、在原以为是文本的内容中出现不可预料的零字节、忘记给NULL字节留出空间。并且，对于任何字符串处理的用途来说，这都是相当痛苦的事情。

Foundation通过创建一个字符串类NSString绕过了C字符串（及其相关的麻烦）。有两

种类型的NSString：NSString直接量是编译器所创建的常量对象，而普通的NSString对象像其他的Objective-C对象一样分配。

通过将文本放置在双引号之间并且用一个@放在其前面，来创建一个NSString直接量：

```
NSString* greeting = @"Hello";
```

上面的代码行将greeting声明为指向一个NSString的指针，并且使用指向带有文本"Hello"的一个常量NSString的地址的一个指针来初始化它。

除了直接量字符串，可以通过常用的alloc...init序列来创建NSString对象，或者使用NSString众多的简便构造函数。如下的示例使用一个简便构造函数，从一个文件中存储的字符来创建一个NSString：

```
NSString* textOfFile =
   [NSString stringWithContentsOfFile: @"/Users/you/textfile.txt"
              encoding: NSUTF8StringEncoding
              error: NULL]; // Ignore errors for now
```

注意 文件路径参数本身就是一个NSString。

注意 字符编码是一个复杂的主题，它超出了本书的范围。这些示例使用普通的ASCII字符。ASCII字符是一种合法的单字节UTF8（Unicode）字符。

NSString是一个较大的类（有超过100个方法），带有从文件、URL及从其他字符串来创建字符串的方法。它拥有比较字符串、获取子字符串、将字符串写入到一个文件或URL、格式化字符串的方法，并且还有一些特殊的方法用来操作作为文件系统路径的字符串。NSString对象在整个Objective-C中使用。除了操作文本和文件系统路径的明确用法，NSString对象还主要用作Foundation字典类的键。

9.3.1 NSString示例

本小节给出了可以使用NSString对象做的一些事情的几个示例。每个代码段后面都给出了执行代码的结果。

NSLog

```
NSLog( @"The sky is %@.", @"blue" );
```

上面的代码行将会有如下的输出：

```
The sky is blue.
```

注意：

- NSLog的格式字符串自身是一个NSString。
- NSLog发送给对应于%@修饰符的参数一条description消息。description方法返回的NSString的文本，替换了输出中的%@。而NSString的description实现只是返回self。

获得一个NSString的长度

```
NSString *string = @"A string to measure";
```

```
NSUInteger stringLength = [string length];
NSLog( @"The length is: %d.", stringLength);
```

产生输出：

```
The length is: 19.
```

获取一个大写的NSString

```
NSString *string1 = @"Make me taller.";
NSString *string2 = [string1 uppercaseString];
NSLog( @"Uppercased string: %@", string2 );
```

产生输出：

```
Uppercased string: MAKE ME TALLER.
```

还有一个对应的lowercaseString方法。注意，uppercaseString方法没有把string1转换为一个大写字符串。它返回一个新的（自动释放）字符串string2，它是string1的大写版本。string1保持不变。

注意 有时候，你会看到如下所示的代码行：

```
string1 = [string1 uppercaseString];
```

这并不意味着最初的NSString已经转换为大写了。NSString对象是不可变的。上面这行代码的含义是：变量string1的类型是“指向NSString的指针”，给它赋了一个新值，是由uppercaseString方法返回的字符串的地址。如果string1最初的值是一个字符串直接量，或者一个自动释放的字符串，那么这里没有问题。然而，如果最初的string1不是一个直接量，不是自动释放的，并且没有其他引用，那么最初的string1就泄漏了。

把一个NSString追加到另一个NSString

```
NSString *string1 = @"The First Part ";
NSString *string2 = @"The Second Part";
NSString *result = [string1 stringByAppendingString: string2];
NSLog( @"Resulting string: %@", result );
```

产生输出：

```
Resulting string: The First Part The Second Part
```

在上面的示例中，string1没有修改。stringByAppendingString返回一个新的（自动释放）字符串。

将一个句子分解为单独的单词

```
NSString *string1 = @"This sentence is falling apart";
NSArray *words =
  [string1 componentsSeparatedByCharactersInSet:
    [NSCharacterSet whitespaceCharacterSet]];
NSLog( @"Word array: %@", words);
```

产生输出：

```
Word array: (
    This,
    sentence,
    is,
    falling,
    apart
)
```

方法componentsSeparatedByCharactersInSet返回一个数组，其中包含了作为一个单独的NSString的句子中的每个单词。

componentsSeparatedByCharactersInSet使用其参数中指定的字符作为划分单个单词的分隔符。NSCharacterSet是一个Foundation类，它表示字符的集合。whitespaceCharacterSet是一个简便构造函数，它返回包含空格和制表符的字符集合。

9.3.2 C字符串和NSString之间的转换

如果你的程序与普通C库或UNIX级别的操作系统接口，你可能需要在C字符串和NSString对象之间来回转换。如下的代码从一个C字符串生成一个NSString：

```
char* cString = "Hello Universe";
NSString* nsString = [NSString stringWithUTF8String: cString];
```

如下的代码做反向的工作：

```
NSString* nsString = @"Anybody Home?";
char* cString = [nsString UTF8String];
```

UTF8String返回UTF-8 Unicode字符的一个数组。UTF-8是ASCII的一个超集。值0～127在UTF-8和ASCII中都表示相同的字符。旧的NSString方法stringWithCString:和cString现在都废弃了。

注意 *系统拥有UTF8String方法返回的字符的内存。当前的自动释放池空了时，系统会释放这些内存。如果需要C字符串在当前函数作用域以外保持有效，应该生成该字节的自己的副本。*

NSMutableString

NSString对象是不可变的。当你创建了一个NSString之后，就不能够修改它了。如果想要修改一个NSString中的文本，必须使用NSString的可变子类NSMutableString。如下面的代码所示：

```
NSMutableString* mutableString =
  [NSMutableString stringWithString: @"Shun the Bandersnatch."];

[mutableString insertString: @"frumious " atIndex: 9];

NSLog( @"%@", mutableString );
```

将产生：

```
Shun the frumious Bandersnatch.
```

在NSString或NSMutableString中指定一个字符位置的索引从0开始，就像所有的C索引一样。

9.3.3 直接量字符串

直接量NSString实例是由编译器在只读内存中创建的静态对象。在能够使用指向NSString实例的指针的任何地方，都可以使用直接量字符串。

```
NSString *string = [@"speak louder!" uppercaseString];
NSLog( @"Uppercased string: %@", string );
```

将产生：

```
Uppercased string: SPEAK LOUDER!
```

直接量字符串在程序的生命周期内都是有效的。它们不会受到Objective-C内存管理系统的影响。然而，由于你通常不知道一个类型为NSString*的变量是指向一个直接量NSString还是一个分配的NSString，因此你应该总是遵守内存管理规则。遵守了规则，如果你最终向一个直接量字符串发送一条retain或release消息，也是无害的。这样的一条消息什么也不做。

9.4 集合类

NSArray、NSDictionary、NSSet及其可变的子类，都是集合类。它们为处理成组的对象提供了容器。在你熟悉的其他语言（特别是脚本语言）中，容器是语言的一部分。然而，在Objective-C中，容器是由Foundation框架提供的。NSDictionary和NSArray广泛地用在Foundation和AppKit中，用来将多个对象传递到一个方法中，或者获取多个对象作为返回值。

9.4.1 NSArray

常规C数组只是字节连续的区域，其大小足够保存所请求数目的确定类型的项。例如：

```
unsigned char name[25];
```

只是设置了从地址name[0]开始的25个字节。对于数组的长度没有运行时信息，也没有边界检测。没什么会阻止你做下面的事情：

```
unsigned char name[25];
name[100] = 200;
```

上面的代码中的赋值将会覆盖内存位置&name+100处所存储的任何内容，这可能会导致毁坏数据或程序崩溃。这种错误，不管是偶然的还是有意的（缓存溢出攻击），都是C编程中很令人头疼的问题。

NSArray类（及其可变子类NSMutableArray）提供了解决这些问题的对象的数组。NSArray有如下的特征：

- NSArray对象只保存Objective-C对象。不能够在NSArray中放置基本的C类型。

- NSArray对象没有边界检查。任何试图访问索引小于0或索引超过数组末尾的一个数组位置，都将导致一个运行时错误。
- NSArray对象是密集的。有效的索引从0开始依次到数组长度减1，中间没有任何遗漏。
- NSArray对象保留它们的内容。当对象放入到一个数组中时，它接收一条retain消息；当对象从数组删除时，它接收一条release消息。

NSArray的长度

count方法返回NSArray中的项的数目：

```
NSArray *fruitBasket =
  [NSArray arrayWithObjects:
     @"Apple", @"Orange", @"Banana", nil];

NSUInteger numFruits = [fruitBasket count];
NSLog (@"Number of fruits in the basket: %d", numFruits );
```

arrayWithObjects是一个简便构造函数，它返回带有指定对象的一个数组。其参数是要放入到该数组中的对象的列表，对象之间用逗号分隔开。该列表末尾是一个nil。当运行上面的代码时，结果是：

```
Number of fruits in the basket: 3
```

NSArray边界检查

用一个负的索引或超过数组末尾的一个索引来引用一个数组位置的任何尝试，都将导致一个运行时错误。如下代码：

```
NSArray *fruitBasket =
  [NSArray arrayWithObjects:
     @"Apple", @"Orange", @"Banana", nil];

id myFruit = [fruitBasket objectAtIndex: 5];
```

会导致如下错误：

```
*** Terminating app due to uncaught exception 'NSRangeException',
reason: '*** -[NSCFArray objectAtIndex:]: index (5) beyond bounds (3)
```

注意 错误消息中提到的NSCFArray是NSArray的一个私有子类。还记得吧，NSArray是一个类簇。当请求一个类簇的实例时，接收类簇的公有类的一个私有子类的实例。

把一个对象添加到NSMutableArray

把一个对象添加到NSMutableArray，会使得该数组的长度增加1。可以在已有的索引范围中插入对象，或者将其添加到数组的末尾。不能以任意的索引添加一个对象。

如果你尝试如下的代码：

```
NSMutableArray *mutableFruitBasket =
  [NSMutableArray arrayWithObjects:
```

```
    @"Apple", @"Orange", @"Banana", nil];

[mutableFruitBasket insertObject: @"Kiwi" atIndex: 6];
```

就像前面的例子一样，会得到一个超出边界的错误。

如果在一个数组中间插入一个对象，那么位于插入的索引及其以上的所有对象，都将移动一个位置：

```
NSMutableArray *mutableFruitBasket =
  [NSMutableArray arrayWithObjects:
    @"Apple", @"Orange", @"Banana", nil];

NSLog( @"Fruit basket before adding kiwi: %@", mutableFruitBasket );

[mutableFruitBasket insertObject: @"Kiwi" atIndex: 1];

NSLog( @"Fruit basket after adding kiwi: %@", mutableFruitBasket );
```

上面的代码运行结果是：

```
Fruit basket before adding kiwi: (
    Apple,
    Orange,
    Banana
)
Fruit basket after adding kiwi: (
    Apple,
    Kiwi,
    Orange,
    Banana
)
```

类似的，如果你从一个可变数组删除一个对象，那么索引比删除的对象高的所有对象，都将移动一个位置来填充空位。数组的长度将减去1。

NSArray中对象的内存管理

当将对象添加到一个NSArray中时，给其发送一条retain消息；而当销毁数组或删除对象时，给对象发送一条release消息。这意味着，当你试图使用已经从NSMutableArray删除的一个对象时，必须要小心。如果该数组只保留了该对象的内容，那么当从数组中删除对象时，对象的引用计数将减少到0。因而在你有机会使用对象之前，对象将会销毁：

```
// Dangerous!
id objectOfInterest = [myMutableArray objectAtIndex: 0];
[myMutableArray removeObject: objectOfInterest];

// objectOfInterest may have just been deallocated

[objectOfInterest someMessage];
```

上面的示例中的最后一行代码，向一个销毁了的对象发送消息，并且由此引发一次程序崩溃。如果你打算使用从一个数组删除的对象，那么在删除它之前，应该retain它：

```
// Better!

id objectOfInterest = [[myMutableArray objectAtIndex: 0] retain];

[myMutableArray removeObject: objectOfInterest];

[objectOfInterest someMessage];
```

如果你确实retain了该对象，必须记住当你用完它的时候，要用一条release或一条autorelease消息来对应该retain消息。

9.4.2 NSDictionary

NSDictionary和NSMutableDictionary提供了一种方式来处理键-值对集合（字典在其他语言中有时候称为关联数组）。值是你试图分组或记录的对象，键是用作记录值的句柄或名称的对象。键-值对称为一个条目。

当向一个字典添加条目时，字典复制了该键。这意味着，如果将一个对象用作一个键，那么对象的类必须实现copyWithZone:（参见第6章）。除了这一限制，键可以是任何类型的对象，但是实际上，键通常是NSString对象。

下面的示例展示了如何创建一个可变的字典，并向其添加一些条目，并且使用一个键来访问一个存储的对象：

```
NSMutableDictionary *favoriteFruits =
    [NSMutableDictionary dictionaryWithCapacity: 3];
// Add some entries
[favoriteFruits setObject: @"Orange" forKey: @"FavoriteCitrusFruit"];
[favoriteFruits setObject: @"Apple"  forKey: @"FavoritePieFruit"];
[favoriteFruits setObject: @"Blueberry"  forKey: @"FavoriteBerry"];

NSLog( @"Favorite fruits are: %@", favoriteFruits);

// Replace an entry
[favoriteFruits setObject: @"Raspberry"  forKey: @"FavoriteBerry"];

NSLog( @"Favorite fruits are: %@", favoriteFruits );

// Retrieve an entry by key
id pieFruit = [favoriteFruits objectForKey: @"FavoritePieFruit"];

NSLog( @"Favorite pie fruit is: %@", pieFruit );
```

运行上面的示例的结果是：

```
Favorite fruits are: {
    FavoriteBerry = Blueberry;
```

```
    FavoriteCitrusFruit = Orange;
    FavoritePieFruit = Apple;
}
Favorite fruits are: {
    FavoriteBerry = Raspberry;
    FavoriteCitrusFruit = Orange;
    FavoritePieFruit = Apple;
}
Favorite pie fruit is: Apple
```

简便构造函数dictionaryWithCapacity的参数只是指定了字典的初始容量。随着条目的添加，NSMutableDictionary将自动增长到所需的大小。

方法setObject:forKey:用来向一个可变的字典添加条目。在一个给定的字典中，一个键只能够用一次。如果使用字典中已经使用的一个键来添加条目，与该键相关联的最初的值，将会被新的值替代。在上面的示例中，再次使用键@"FavoriteBerry"，将会导致@"Blueberry"值被@"Raspberry"所替代。

方法objectForKey:用来从一个NSDictionary或NSMutableDictionary获取值。如果没有与作为参数提供的键相对应的条目，则该方法返回nil。

当值添加到字典中时，向它们发送一条retain消息，但是，复制相对应的键，则并不会发送一条retain消息。当删除一个条目或者释放该字典时，向键（字典的副本，而不是最初的）和值都发送一条release消息。和NSMutableArray一样，如果你试图使用从NSMutableDictionary删除的一个对象，则应该在删除对象之前保存该对象。如果你没有保存该对象，则从字典中删除该对象可能会导致其销毁。

9.4.3 NSSet

NSSet和NSMutableSet将对象的集合实现为一个数学集合。该集合的主要特征是：

- 集合的成员是无序的。没有办法从集合获取一个特定的对象。可以使用allObjects方法请求一个包含集合中的所有对象的数组，或者使用NSEnumerator对象来遍历该集合（参见第10章）。allObjects返回的数组中的对象的顺序，或者连续调用集合的枚举器而获得的数组中的对象的顺序，都是不确定的。
- 一个给定的对象在数组中只出现一次，不管它被添加到一个集合中多少次。就像NSArray和NSDictionary一样，当将对象添加到一个集合时，向其发送一条retain消息；并且，当从集合删除对象时或者销毁集合时，向其发送一条release消息。如下的示例给出了将一个对象添加到已经包含该对象的一个NSMutableSet中的结果：

```
NSMutableSet *favoriteFruits =
    [NSMutableSet setWithCapacity: 3];

// Add some objects
[favoriteFruits addObject: @"Orange"];
[favoriteFruits addObject: @"Apple"];
```

```
NSLog( @"Favorite fruits are: %@", favoriteFruits );

// Add the same object again
[favoriteFruits addObject: @"Apple"];
NSLog( @"Favorite fruits are: %@", favoriteFruits );
```

运行上面的代码的结果是：

```
Favorite fruits are: {(
    Apple,
    Orange
)}
Favorite fruits are: {(
    Apple,
    Orange
)}
```

尽管添加了两次，但@"Apple"只在集合中出现一次。

9.5 NSNumber

可能令Objective-C初学者（至少那些没有用过Java的人）感到惊讶的一件事情是，你可以将C数字类型直接放入到Objective-C数组、字典或集合中。实际上，在需要一个对象的任何地方，不能使用一个常规的C类型（数字或结构）或指向常规C类型的一个指针。基本C类型不是对象。

如果你试图在NSArray中放置一个浮点数：

```
float x = 99.9;
NSMutableArray* array = [NSMutableArray array];
[array addObject:  x];
```

这段代码不会编译：

```
error: incompatible type for argument 1 of 'addObject:'
```

如果你使用一个int而不是一个float来尝试同样的事情：

```
int n = 99;
NSMutableArray* array = [NSMutableArray array];
[array addObject:  n];
```

这段代码的编译将会得到一条警告，指出你将int和指针混淆了：

```
warning: passing argument 1 of 'addObject:' makes pointer from integer without a
cast
```

但是，在运行时程序则会崩溃：

```
Segmentation fault
```

要将一个数字存储到一个Objective-C集合类中，或者要在需要一个对象的任何地方使用一个数字，必须将数字包装到NSNumber类的一个实例中：

```
float aFloat = 99.9;
```

```
NSNumber *floatAsAnObject = [NSNumber numberWithFloat: aFloat];

NSMutableArray* array = [NSMutableArray array];
[array addObject:  floatAsAnObject];

NSLog( @"Array contains: %@", array );
```

运行上面的代码，将会产生如下结果：

```
Array contains: (
    "99.9"
)
```

当你需要返回实际的数字时，NSNumber将其返回给你：

```
float y;

y = [floatAsAnObject floatValue];  // y now = 99.0
```

NSNumber可以存储并返回你喜欢的所有数字类型，以及BOOL和char。尽管可以使用通常的alloc... init来创建NSNumber对象，但更常见的是使用类简便构造函数：

```
NSNumber* fiveAsNSNumber = [NSNumber numberWithInt: 5];
```

NSNumber的实例是可变的。在创建一个NSNumber对象之后，没有方法来修改它所表示的数字值。如果需要包装一个不同的数字值，则必须创建一个新的NSNumber。

9.6 NSNull

关于Foundation的另一件令人惊讶的事情是，它很容易将表示“没有对象”的某物存储到一个集合对象中。有时我们需要在一个数组中放置一个“没有对象”以作为一个占位符，或者在字典中用一个键输入“没有对象”以表示还没有内容与该键相关联。尽管对于大多数用途，nil用作指向没有对象的一个指针，但它不是一个对象实例。不能在一个数组、字典或集合中放置nil。

如下的示例甚至比试图将一个int或一个float放入到数组中更危险：

```
id someObject = nil;

NSMutableArray* myArray = [NSMutableArray array];

[myArray addObject: someObject];
```

该示例在编译时甚至没有警告（因为当你试图将someObject插入到数组时，编译器没有办法知道它将是nil），但是，在执行时，它导致一个错误：

```
*** Terminating app due to uncaught exception
    'NSInvalidArgumentException',
reason:
    '*** -[NSCFArray insertObject:atIndex:]: attempt to insert nil'
```

为了解决这个问题，Foundation提供了NSNull类。NSNull只有一个类方法，它返回

null对象：

```
+(NSNull*) null
```

可以安全地将null对象放入到一个集合类中：

```
NSMutableArray* array = [NSMutableArray array];
[array addObject: [NSNull null]];
NSLog( @"Array contains: %@", array );
```

上面的代码产生如下的输出：

```
Array contains: (
    "<null>"
)
```

除非NSNull对象只是作为一个对象存在，否则它绝对不会做任何事情。它没有实例方法。由于NSNull没有做任何事情，因此不需要拥有它的多个实例。它是一个单体，对[NSNull null]的每次调用都返回相同的对象。在不想引发任何程序膨胀的多个地方，可以多次使用它。

9.7 NSData

总是停留在对象的世界里是不可能的。可能有时候，我们必须处理一段字节。可能字节是你必须自己解析的某种格式的一个文件的内容，或者是原始的图像数据。NSData提供了一种方便的方式，将一块字节包装起来，并且将其当做一个Objective-C对象对待。

使用NSData引发一些不可避免的问题，“谁拥有这些字节”，“这些字节来自何处”？NSData提供了几种选择。你可以要求NSData通过复制某些已有的字节来创建一个实例：

```
unsigned howManyBytes = 100;

char* someBytes = (char*) malloc( howManyBytes );

NSData* data = [NSData dataWithBytes: someBytes length: howManyBytes];
```

上面的代码将someBytes复制到NSData对象所创建和拥有的一个缓冲区中。当NSData对象销毁的时候，释放该缓冲区。你仍然负责最终释放最初的someBytes。如果你想避免一次复制的代价，可以按照如下进行操作：

```
unsigned howManyBytes = 100;

char* someBytes = (char*) malloc( howManyBytes );

NSData* myData =
    [NSData dataWithBytesNoCopy: someBytes
            length: howManyBytes
            freeWhenDone: YES];
```

NSData对象使用已有的字节作为其缓冲区。如果freeWhenDone:参数是YES，那么NSData对象获得字节的所有权，并且当该对象销毁的时候释放它们（如果someBytes不是最初使用malloc所分配的，那么freeWhenDone:应该是NO。传递YES将会导致你的程序崩溃）。

9.7.1 访问NSData的字节

可以使用bytes方法来获得对一个NSData对象的字节缓冲区的读取访问。bytes方法的返回值是const void*，因此，它应该赋值给类型为指向常量的指针的一个变量：

```
NSData *data = ...
const char*  constantBytes = [data bytes];
```

如果你需要对字节做读取它们以外的一些事情，或者，如果你想要从头创建一个字节缓冲区的内容，那么必须使用NSData的可变子类NSMutableData。NSMutableData的字节的数目和内容都是可以改变的。

可以使用方法mutableBytes来获取指向一个NSMutableData对象的字节缓冲区的指针：

```
NSMutableData* mutableData = [NSMutableData dataWithLength: 100];
char* mutableBytes = [myMutableData mutableBytes];
```

该示例的第一行使用一个100字节长的缓冲区创建了一个NSMutableData对象。第二行获取了指向缓冲区中的第一个字节的一个指针。当然，mutableBytes是一个纯粹的C指针。使用它写入超出如下字节的地址的一个地址，将是非常糟糕的想法：

```
mutableBytes[[myMutableData length] - 1]
```

9.7.2 文件和NSData之间的转换

NSData拥有能够将其字节放入到文件中，以及从文件中移出内容的方法：

```
NSString* pathToFile = ...

NSData* myData = [NSData dataWithContentsOfFile: pathToFile];

NSString* pathToOutputFile = ...

[myData writeToFile: pathToOutputFile atomically: YES];
```

如果atomically:参数为YES，则将字节写入到一个备份文件中。当写入完成后，备份文件更改为路径参数中指定的名称。

9.8 NSURL

Apple朝着使用URL来指定文件资源和网络资源的方向前进。它们使用引用文件的方法，该方法带有一个file://URL，从而替代了使用路径来引用一个文件的方法。Foundation将URL字符串包装在一个NSURL中，而不是使用一个NSString来保存URL。NSURL既用于文件URL也用于网络URL：

```
NSURL *someFile = [NSURL fileURLWithPath: @"/Users/rclair/notes.txt"];
NSURL *website =
    [NSURL URLWithString:  @"http://www.chromaticbytes.com"];
NSLog(@"Host: %@ URL: %@", [someFile host], [someFile absoluteString]);
NSLog(@"Host: %@ URL: %@", [website host], [website absoluteString]);
```

上面的代码根据一个路径创建了一个文件NSURL，并且从一个网络URL的一个NSString版本创建了一个网络NSURL。然后，对于每个NSURL对象，它使用host方法显示了主机，并使用absoluteString方法显示了URL的字符串。执行这段代码将产生如下的输出：

```
Host: localhost URL: file://localhost/Users/rclair/notes.txt
Host: www.chromaticbytes.com URL: http://www.chromaticbytes.com
```

9.9 结构

Foundation和Objective-C的一般原则一样，它们不是面向对象编程的教条主义者。有几个小实体只是普通的结构而不是对象。你可能会遇到的结构有：

- NSRange定义一个起点和长度：

```
typedef struct _NSRange
{
  NSUInteger location;
  NSUInteger length;
} NSRange;
```

- NSPoint在二维空间中定义一个点：

```
typedef struct _NSPoint

{
  CGFloat x;
  CGFloat y;
} NSPoint;
```

- NSSize定义一个矩形区域的大小。偶尔会看到它被误用为一个偏移量。

```
typedef struct _NSSize

{
  CGFloat width;
  CGFloat height;
} NSSize;
```

- NSRect根据原点（x和y最小值的一角）定义一个2D矩形及其大小：

```
typedef struct _NSRect

{
  NSPoint origin;
  NSSize size;
} NSRect;
```

NSRange在字符串中广泛地用来表示子字符串：

```
NSString *string = @"abcdefghijklmnopqrstuvwxyz";
NSRange *range = NSMakeRange( 5, 10 );
NSString *substring = [string substringWithRange: range];
NSLog( @"Substring is %@", substring );
```

上面的代码使用Foundation函数NSMakeRange来创建一个从5开始长度为10的NSRange结构。这个范围结构用作NSString方法substringWithRange:的参数，来创建一个从索引5开始并扩展到10个字符的一个子字符串。执行这段代码将产生如下的输出：

```
Substring is fghijklmno
```

尽管NSPoint、NSSize和NSRect在Foundation中声明，但它们在整个AppKit中用来定义视图、控件和窗口的大小与位置，并且用于绘图过程中。

注意 在Foundation的iPhone版中，没有定义NSPoint、NSSize和NSRect。UIKit使用Core Graphics结构CGPoint、CGSize和CGRect。

NSPoint、NSSize和NSRect与Core Graphics中对应的CGPoint、CGSize和CGRect有着相同的内存布局。如果你为了某些绘图必须降低到Core Graphics层级，可以通过使用Foundation函数NSRectToCGRect和NSRectFromCGRect在Foundation和Core Graphics之间来回转换：

```
NSRect nsRect = NSMakeRect( 0.0, 0.0, 100.0, 100.0 );
CGRect cgRect = NSRectToCGRect( nsRect );
```

在另一端：

```
CGRect cgRect = CGRectMake( 0.0, 0.0, 100.0, 100.0 );
NSRect nsRect = NSRectFromCGRect( cgRect );
```

除了创建函数，Foundation定义了多个其他的函数用来使用NSRange、NSPoint、NSSize和NSRect。[⊖]

9.10 小结

本章介绍了一些最为常用的Foundation类：NSString、NSArray、NSDictionary、NSSet、NSNumber、NSData和NSURL。NSString和NSArray比普通C中对应的结构要安全，因为它们有边界检查。NSDictionary类实现了其他语言中类似的关联数组。NSNumber类为数字值提供了一个包装器，以便在必要的时候可以将数字当做对象处理。NSData为字节块提供了类似的包装服务。本章的9.9节介绍了一些普通C结构的Foundation实体：NSRange、NSPoint、NSSize和NSRect。

9.11 练习

1. 验证Foundation数组类是进行边界检查的：
   - 创建一个NSMutableArray并且用3个NSString对象类填充它。
   - 使用方法objectAtIndex:，尝试从index-1获取一个对象。再次用索引3来尝试，那将会发生什么情况？

⊖ 参见http://developer.apple.com/mac/library/ documentation/Cocoa/Reference/ Foundation/Miscellaneous/ Foundation_Functions/Reference/reference.html。

- 使用方法insert Object:atIdex，尝试在index-1处插入另一个NSString，那将会发生什么情况。
- 现在，尝试在索引3插入新的字符串。说明你的结果。

2. 练习使用字典：
   - 创建一个空的NSMutableDictionary。
   - 将你的家庭和朋友的名称（作为NSString）添加到字典中。使用表示他们与你的关系的字符串作为键：

```
[friendsAndFamilyDict addObject" @"Martin" forKey: @"Father"];
```

   - 验证你可以多次使用不同的键来将同一对象插入到字典中。
   - 尝试使用一个已经用过了的键来插入一个不同的对象，看会发生什么情况。
   - 使用objectForKey:，尝试获取一个不存在的键的值。程序会崩溃吗？
3. 创建一个Adder类，它带一个类方法，该方法接受两个NSNumber对象作为参数，并且以NSNumber返回它们的和（为了本练习，假设NSNumber对象包含浮点数值）。编写一个小程序来测试你的方法。
4. 排序一个NSArray：
   - 创建一个NSArray，它保存如下的字符串：@"Raspberry"、@"Peach"、@"Banana"、@"Blackberry"、@"Blueberry"和@"Apple"。
   - 使用NSArray方法sortedArrayUsingSelector:获取一个数组的排序版本。使用NSString方法caseInsensitiveCompare:的选择器作为参数。在进行排序之前，你应该先阅读sortedArrayUsingSelector:和caseInsensitiveCompare:的文档。
   - 显示新创建的排序的数组的内容。
5. 练习使用NSData：
   - 使用50个字节的一个缓冲区来创建一个NSMutableData对象。
   - 获取指向对象缓冲区的一个指针。
   - 使用指针向缓冲区中写入前50个整数。
   - 把NSMutableData对象的内容写入到一个文件中。
   - 在另一个程序中，使用该文件来初始化一个NSData对象。
   - 获取指向NSData对象的缓冲区的指针。
   - 使用该指针来显示缓冲区中存储的值，并验证一切都正确地工作。
6. 创建一个类，它显示自己的retain和release消息。你可以通过覆盖retain和release来做到这一点。在这两个方法中，插入一条NSLog语句作为这两个方法的第一行，然后，调用该方法的superclass版本。对NSMutableArray和NSMutableDictionary使用你的类，以验证：
   - 当对象放入到一个集合中时，会保留该对象。
   - 当从集合删除一个对象时，会释放该对象。
   - 当销毁集合时，如果对象在集合中的话，则会释放它。

第10章　Objective-C中的控制结构

第1章介绍了用在循环和分支中的基本的C语言控制语句：if、for和while。本章将介绍这些语句通常如何用于Objective-C环境中。本章还将介绍其他的一些Objective-C循环结构，包括Objective-C 2.0中新增的快速枚举功能。最后，介绍Objective-C的异常系统。

10.1　if语句

正如第3章提到的，Objective-C定义了自己的Boolean类型BOOL，还有常量YES和NO来表示真和假。返回BOOL的消息表达式可以放入到if语句的条件部分，如下所示：

```
NSArray *employeeNames = ...
NSString* nameOne = @"Ralph";

if ( [employeeNames containsObject: nameOne] )
  {
    ...
  }
```

可以将返回BOOL的表达式与常见的逻辑运算符一起使用，以产生复合条件：

```
NSArray *employeeNames = ...

NSString *nameOne = @"Ralph";
NSString *nameTwo = @"Alice";
if ( [employeeNames containsObject: nameOne] &&
     [employeeNames containsObject: nameTwo] )
  {
    ...
  }
```

记住，C对于复合条件使用短路计算。如果nameOne没有包含在employeeNames中，条件中的第二个消息表达式将不再计算。在上面的例子中，这没有任何区别，但是，如果消息表达式有副作用，它将会有所不同。

关于显式比较的一条警告

与YES和NO进行显式比较的时候，要小心。如下的代码太冗长了：

```
NSArray *employeeNames = ...
NSString* nameOne = @"Ralph";

// Technically OK, but verbose

if ( [employeeNames containsObject: nameOne] == NO )
```

```
  {
    NSLog( @"%@ doesn't work here." );
  }
```

然而，与YES的显式比较很可能会导致bug：

```
NSArray *employeeNames = ...
NSString* nameOne = @"Ralph";

// Not yet a bug, but dangerous

if ( [employeeNames containsObject: nameOne] == YES )
  {
    NSLog( @"%@ works here." );
  }
```

在上面的例子中，显式比较没有问题，因为containsObject:只会返回两个值中的一个，即YES或者NO。但是，如果习惯于与YES进行显式比较，那么在实际中可能会让你遇到麻烦。例如，假设你有一个NSString，并且只有当字符串是非零长度时，才用它来做某些事情。如果像下面这样编写代码，认为“非零意味着真”，那么，你会遇到麻烦：

```
NSString* nameOne = @"Ralph";

// WRONG!
if ( [nameOne length] == YES )
  {
    // Do something with non-zero length string
  }
```

该条件为假。

还记得吗？BOOL实际上是一个整数类型，并且，YES是值为1的一个确定的常量：

```
#define YES  (BOOL)1
```

在前面的例子中、字符串的长度是5。5不等于1。

下面是编写前面的if语句的正确方法：

```
// Correct

if ( [nameOne length] )
  {
    // Do something with non-zero length string
  }
```

如果想要更加明确，可以这样编写：

```
// Also correct

if ( [nameOne length] != 0 )
  {
    // Do something with non-zero length string
  }
```

测试对象的相等性

if语句一种常见的用法是，检查两个表达式是否相同。对基本数字类型来说，这很简单而且很稳定：

```
int b = ...
int a = 9;

if ( a == b )
  {
    printf("The same!\n" );
  }
```

当b设置为9时，括号中的表达式为真；否则，为假。

> **小心==和=**
>
> 当进行类型比较时，要小心。==是相等性测试，而=是赋值运算符。意外地漏掉一个=，而输入：
>
> ```
> if ( a=b )
> ```
>
> 而实际上你是想输入：
>
> ```
> if ( a==b )
> ```
>
> 这是一个典型的C错误。第一个表达式把b的值赋给了a，然后，通过查看a是零还是非零来确定表达式的真值。而第二个表达式检查a和b是否相等。

当比较对象时，必须考虑“相等”的含义是什么。必须区分指针相等和相等性。指针相等很容易。如果两个对象（这里的对象的意思是“指向类的一个实例的指针”）都指向相同的内存位置，则这两个对象是相等的。可以使用与比较基本类型相同的方式，来检查指针相等性：

```
SomeClass *object1 = ...
SomeClass *object2 = ...

if ( object1 == object2 )
  {
    NSLog( "The objects are identical" );
  }
```

对于某些类，有可能类的实例相等，但不完全相同。例如，有两个不同的NSString对象，它们包含完全相同的文本。还有两个不同的NSColor对象，它们拥有相同的颜色空间和颜色成分。

NSObject定义了一个方法isEqual:，它允许你检查相等性。NSObject的isEqual:实现只是检查指针相等性。如果SomeClass没有覆盖isEqual:，则如下的示例与前面的示例具有相同的效果。

```
SomeClass *object1 = ...
SomeClass *object2 = ...

if ( [object1 isEqual: object2] )
```

```
{
  NSLog( @"The objects are equal" );
}
```

很多类确实覆盖了isEqual:。例如，如果两个对象都是NSColor的实例，并且拥有相同的颜色空间和颜色成分，则NSColor的实现将会返回YES。

一些Cocoa类定义了特殊的方法，当我们知道要测试的对象的某些内容时，可以用这些方法来测试相等性。NSString定义了方法isEqualToString:，当我们知道对象是字符串时，它比使用isEqual:要快：

```
NSString *string1 = ...
NSString *string2 = ...
if ( [string1 isEqualToString: string2] )
  {
    NSLog( @"The strings are the same." );
  }
```

注意 不要把方法isEqual:和isEqualTo:搞混淆了。方法isEqualTo:是用来排序对象的Cocoa脚本协议的一部分。

10.2 for语句和隐式循环

我们最常见的编程任务之一，就是遍历一个对象集合。

10.2.1 for语句

可以使用一个标准的C的for循环来迭代一个Objective-C数组，例如：

```
NSArray *displayList = ... // Array of graphic objects to draw
NSUInteger j;

NSUInteger numToDraw = [displayList count];

for (j=0; j < numToDraw; j++ )
  {
    [[displayList objectAtIndex: j] draw];
  }
```

objectAtIndex:返回的对象，是作为类型id返回的。只要其中的每一个都能够响应一条draw消息，循环将会正确地运行。

10.2.2 隐式循环

可以使用NSArray的makeObjectsPerformSelector:方法来创建一个隐式循环，从而让前面的示例更清晰一些。

```
NSArray *displayList = ...
[displayList makeObjectsPerformSelector: @selector( draw )];
```

隐式循环所做的事情与显式循环完全相同。依次给displayList中的每个对象发送一条draw消息。

NSSet也实现了一个makeObjectsPerformSelector:方法：

```
NSSet *aSet = ...
[aSet makeObjectsPerformSelector: @selector( doSomething )];
```

aSet中的每个对象只接收到一条doSomething消息一次。向对象发送消息的顺序是不确定的。

NSArray和NSSet也实现了方法：makeObjectsPerformSelector:withObject:这个方法允许使用接受单个对象参数的选择器来创建隐式循环：

```
NSMutableArray *displayList = ...
[displayList makeObjectsPerformSelector: @selector( drawWithColor: )
             withObject: [NSColor blueColor]];
```

displayList中的每个对象都被发送了一条drawWithColor:[NSColor blueColor]消息。

如果需要使用接受一个基本类型作为参数的选择器，或者接受多个参数的选择器，来遍历一个集合或数组，则必须回到一个显式循环。

注意 用作方法makeObjectsPerformSelector:或makeObjectsPerformSelector:withObject:的参数的选择器，不能修改作为接收者的数组或集合。

10.2.3 带有块的隐式循环

从Mac OS X Snow Leopard (v 10.6)开始，Apple已经添加了众多的方法，来使用块遍历Foundation集合对象。块类似于函数，但是，又与函数不同，它们自身带有一个环境。块以及使用块的新的迭代方法，将在第16章中介绍。

10.3 while语句和NSEnumerator

while循环和NSEnumerator类的实例结合使用，这是遍历一个Objective-C集合的理想方式。NSEnumerator接受集合中的对象，并将其传递出来，每次一个，并且使用nextObject方法。当所有的对象发送完毕，对nextObject的再次调用将返回nil。

不能自己创建NSEnumerator对象。NSEnumerator是一个抽象类，它没有用来创建实例的公有接口。相反，可以从它所枚举的集合类来获取一个枚举器对象。

如下的示例重用了前面小节中的代码，但是它是使用while循环来重新编写的：

```
NSMutableArray *displayList = ... // Array of graphic objects to draw
GraphicObject *graphicObject;

NSEnumerator *myEnumerator = [displayList objectEnumerator];

while ( graphicObject = [myEnumerator nextObject] )
```

```
{
  [graphicObject draw];
}
```

括号中的赋值表达式的真值（注意只有一个=）是graphicObject的值。每次经过循环的时候，nextObject的返回值都赋值给graphicObject，然后测试graphicObject看看它是计算为真还是为假。如果nextObject返回一个对象，则graphicObject就是某个非空的值，将其计算为真。循环体将会执行。当遍历完一个集合，nextObject返回nil。graphicObject的真值就变成了假（注意，nil只是0的别名），循环结束。

此时，枚举器对象被删除了。如果想要再次遍历集合，在开始第二个循环之前，必须获取一个新的NSEnumerator对象：

```
myEnumerator = [displayList objectEnumerator];
```

数组的枚举器按照顺序枚举数组的对象。针对集合和字典的枚举顺序是不确定的。一些集合类也拥有专门的枚举器：NSArray提供reverseObjectEnumerator，它从后向前遍历数组，从最高的索引位置的对象开始遍历。NSDictionary提供了keyEnumerator，它遍历用于字典的键的集合，而不是遍历字典所包含的对象。

枚举的同时修改可变的集合

修改一个将要枚举的可变的集合（添加或删除对象）并不安全。结果可能是不可预期的。有两种方式解决这个问题：

- 获取一个新的数组，它包含集合中的所有对象，并且，在修改最初的集合的同时枚举新的数组。
- 枚举集合并且创建必需的修改的一个列表。当你完成了枚举集合时，可以继续前进并进行自己的修改。

假设一个GraphicObject类实现了一个isBitmap方法，该方法用来判断接收者是否是一个位图图像。下面介绍如何通过枚举displayList的一个副本，将一个数组displayList中的所有位图图像都删除掉：

```
1 NSMutableArray *displayList = ... // Array of graphic objects
2 GraphicObject *graphicObject;
3
4 NSEnumerator *enumerator =
5   [[NSArray arrayWithArray: displayList] objectEnumerator];
6
7 while ( graphicObject = [enumerator nextObject] )
8  {
9    if ( [graphicObject isBitmap] )
10     {
11       [displayList removeObject: graphicObject];
12     }
13  }
```

第4行和第5行的语句创建了displayList的一个自动释放（不可变的）版本，并且获得了副本的对象的枚举器。随后，枚举器用来遍历副本数组，同时对初始的集合做出修改。

重要的一点是，在枚举的过程中，将要枚举的数组（displayList的副本）不会被修改。

注意 可以使用allObjects方法，来获取NSSet（或NSMutableSet）中的所有对象的一个数组。可以使用allValues方法，来获取NSDictionary（或NSMutable-Dictionary）中的所有对象的一个数组。在这两种情况下，结果数组中的对象的顺序都是不确定的。

如下的示例展示了如何通过枚举最初的displayList数组，并构建一个应该删除掉的对象的临时数组，来完成前面的任务：

```
NSMutableArray *displayList = ... // Array of graphic objects

GraphicObject *graphicObject;
NSMutableArray *tmpArray = [NSMutableArray array];

NSEnumerator *enumerator = [displayList objectEnumerator];

// Build an array of objects to remove

while ( graphicObject = [enumerator1 nextObject] )
  {
    if ( [graphicObject isBitmap] )
      {
        [tmpArray addObject: graphicObject];
      }
  }
// Remove the objects

[displayList removeObjectsInArray: tmpArray];
```

tmpArray必须是一个NSMutableArray，以便可以向其添加对象。在枚举完成之后，tmpArray中的对象可以安全地从displayList中删除。

10.4 快速枚举

Objective-C 2.0添加了一个新的语法，叫做for...in，用于枚举集合。for...in结构易于编写，并且运行起来比基于NSEnumerator的传统循环要快。有了这些优点，你应该使用这一语法而不是基于while循环的NSEnumerator。但是，在本节稍后会看到，在某些情况下，NSEnumerator作为辅助类仍然是有用的。

for...in循环的基本形式是：

```
for ( type loopVariable in expression )
  {
    statements
  }
```

expression必须计算一个对象（通常是某种类型的一个Cocoa集合类），它遵守

NSFastEnumeration协议（如果一个类实现了一组特定的方法，那么它是遵守一个协议的类。第13章将介绍协议）。

当执行for...in循环时，loopVariable设置为集合中的第一项，并且随后，循环体中的语句将会执行。随后，这些步骤针对集合中的每一项依次重复。

如下的示例重用了前面小节中的代码，但是，它已经使用for...in循环重新编写了：

```
NSMutableArray *displayList = ... // Array of graphic objects to draw

for ( GraphicObject *graphicObject in displayList )
  {
   [graphicObject draw];
  }
```

在for...in循环中，可以使用已有的变量作为循环变量：

```
NSMutableArray *displayList = ... // Array of graphic objects to draw
GraphicObject *graphicObject;

for ( graphicObject in displayList )
  {
    [graphicObject draw];
  }
```

对于for...in循环，需要注意一些重要的事情：

- 可以用一条break语句来中断loop循环，或者使用一条continue语句跳到下一次迭代。
- 如果loopVariable声明为for...in语句的一部分，则它在循环体之外是未定义的。
- 如果loopVariable是一个已有的变量，是在for...in语句之外声明的，则在for...in循环之后，其值取决于循环如何结束。如果循环执行完成，loopVariable设置为nil。
- 如果遇到一条break语句，loopVariable保留为执行该break语句时的任何值。

Cocoa集合类NSArray、NSDictionary、NSSet及NSEnumerator，都遵守NSFastEnumeration协议。为什么有NSEnumerator？有的时候，类提供了NSEnumerator的子类，来执行特殊的任务。例如，按照相反的顺序迭代并绘制displayList，可以这么做：

```
NSMutableArray *displayList = ... // Array of graphic objects to draw

for ( GraphicObject *graphicObject in
      [displayList reverseObjectEnumerator] )
  {
    [graphicObject draw];
  }
```

快速迭代使你的代码看起来很整齐，但是，它也提供了几个额外的好处，其中之一是速度。它很快。它经过优化，执行起来比手写的循环更快。它通过循环的内嵌部分来做到这一点：在幕后，它一次请求多个对象，这避免了很多消息派送或函数调用。

快速枚举能够防止你在迭代一个集合时修改它，从这个意义上来讲，它也是安全的。如下

的代码：

```
NSMutableArray *displayList = ... // Array of graphic objects


for ( GraphicObject *graphicObject in displayList )
  {
    if ( [graphicObject isBitmap] )
      {
        [displayList removeObject: graphicObject];
      }
  }
```

将导致一个运行时异常：

```
*** Terminating app due to uncaught exception 'NSGenericException',
    reason:
*** Collection <NSCFArray: 0x1032b0> was mutated while being enumerated.
```

要修改一个集合，必须做一些与前面小节中展示的类似的事情，例如：

```
NSMutableArray *displayList = ... // Array of graphic objects

NSEnumerator *myEnumerator = [displayList objectEnumerator];
NSArray *allTheObjects = [myEnumerator allObjects];

for ( GraphicObject *graphicObject in allTheObjects )
  {
    if ( [graphicObject isBitmap] )
      {
        [displayList removeObject: graphicObject];
      }
  }
```

NSDictionary和快速枚举

如果你没有仔细阅读相关细则的话，那么在这里你会明白的。当for...in与一个NSDictionary对象一起使用时，它遍历字典的键，而不是其对象。要快速遍历一个字典的对象，必须使用字典的对象枚举器，如下所示：

```
NSDictionary *myDictionary = ...
for ( id objectInDictionary in [myDictionary objectEnumerator] )
 {
  ...
 }
```

10.5 一个使用快速枚举的示例

这里有一个更为广泛的示例，它使用了我们刚刚讨论过的一些结构。它是一个程序，即myls，它模拟了Unix ls命令的基本行为。

注意 ls是列出文件和目录的Unix命令。如果你不熟悉ls，打开Terminal窗口(/Applications/Utilities/Terminal)并输入man ls。或者只是尝试一下：输入ls并按下回车键来查看结果。

在构建了程序清单10.1中的示例之后，可以在Unix提示符下输入myls：

```
%myls
```

myls将列出当前目录下的文件。如果你输入：

```
%myls foo
```

myls将会：

- 如果foo是一个目录，则列出foo。
- 如果foo是一个文件，则回应“foo”。
- 如果foo不存在，则给出一条错误消息。

如果有多个参数，myls对每个参数重复这一过程。

myls使用printf而不是NSLog把输出发送到标准输出（终端窗口）而不是控制台。在几个地方，需要使用NSString方法stringWithUTF8String:和UTF8String，从而在C字符串和NSString实例之间来回转换。

注意 在可能的情况下，应该使用NSString对象而不是C字符串。遗憾的是，OS使用C字符串，因此，要与OS交互，必须来回进行转换。

myls还必须用到我们还没有介绍过的一个Foundation对象：NSFileManager。NSFileManager是一个包装了与OS交互的Foundation对象的例子。它是一个单体对象，提供了方法来通过编程操作文件系统。当程序拥有一个文件管理器对象时，它可以使用filesystem完成大多数那些你能够在一个终端窗口中所做的事情。文件管理器可以创建、移动和删除文件与目录，列出目录，复制文件，检查文件是否存在，检查许可，创建符号链接，以及获取当前的工作目录等。

注意 在使用它之前，你可能需要查看NSFileManager类的文档：http://developer.apple.com/mac/library/documentation/Cocoa/Reference/Foundation/Classes/NSFileManager_Class/Reference/Reference.html。

要做的第一件事情是，使用NSFileManager类方法defaultManager来获取文件管理器的一个单个的实例：

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

然后，可以使用文件管理器来获取一个目录的内容，这些内容作为NSString对象的一个数组：

```
NSArray* directoryContents =
  [fileManager contentsOfDirectoryAtPath: @"/Users/you" error: NULL];
```

注意 contentsOfDirectoryAtPath:error:中的error:参数，用来返回指向一个NSError对象的指针。如果你对获取错误信息感兴趣，则可以传入指向一个NSError对象的指针的地址：

```
NSError *returnedError;
NSArray* directoryContents =
  [fileManager contentsOfDirectoryAtPath: @"/Users/you"
            error: &returnedError];
```

如果调用contentsOfDirectoryAtPath:error:导致一个错误，那么返回的NSError对象将会包含关于该错误的信息。myls检查以确保它传递给contentsOfDirectoryAtPath:的路径存在并且是一个目录。它不带有处理其他错误条件的功能，因此，它传入NULL作为错误参数。

完整的程序如程序清单10.1所示。

程序清单10.1 myls.m

```
1 #import <Foundation/Foundation.h>
2
3 int main (int argc, const char * argv[])
4 {
5   NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
6
7   BOOL isDirectory;
8
9   NSMutableArray* pathsToList =
10            [NSMutableArray arrayWithCapacity: 20];
11  NSFileManager* fileManager = [NSFileManager defaultManager];
12
13  // argv[0] is the command name
14  // Any others are directories or file to list
15
16  if ( argc == 1 ) // Just command name so list the current directory
17    {
18      [pathsToList addObject:  [fileManager currentDirectoryPath]];
19    }
20  else
21    {
22        // Convert input arguments to NSString and
23        // add them to the array
24      int j;
25      for ( j=1; j < argc; j++ )
26        {
27          [pathsToList addObject:
28                [NSString stringWithUTF8String: argv[j]]];
29        }
30    }
```

```
31
32   for ( NSString* path in pathsToList )
33     {
34       if ( [fileManager fileExistsAtPath: path
35                          isDirectory: &isDirectory] )
36         {
37           if ( isDirectory ) // List the files in the directory
38             {
39             NSArray* directoryContents =
40               [fileManager contentsOfDirectoryAtPath: path
41                            error: NULL];
42             for ( NSString* contentPath in directoryContents )
43               {
44                 printf( "%s\n", [contentPath UTF8String] );
45               }
46             }
47           else  // It's a file, just list it
48             {
49               printf( "%s\n", [path UTF8String] );
50             }
51         }
52       else    // Oops, no file or directory with that name
53         {
54           printf("myls: %s:  No such file or directory\n",
55                  [path UTF8String]);
56         }
57     }
58   [pool drain];
59   return 0;
60 }
```

Argv是一个C数组而不是对象，因此myls必须使用一个常规的for循环（从第25行开始）来将任何参数转换到NSString对象中，并且将它们存储到一个NSMutableArray中。

pathsToList和directoryContents是NSMutableArray实例，因此，程序可以使用for...in循环来遍历它们（第32行和第42行）。

NSFileManager方法fileExistsAtPath:isDirectory（第34行）很有趣。其第二个参数类型为BOOL*（指向BOOL的指针）。第二个参数必须是在方法的调用者中声明的一个BOOL的地址。这是因为该方法需要返回两个BOOL值，其中一个表示具体路径上是否有一个条目，另一个表示该条目是否是一个目录。它返回第一个BOOL作为该方法的返回值。通过设置将其地址作为第二个参数传入的BOOL变量的值，它返回了第二个BOOL。

10.6 异常

异常是阻止程序继续执行的不正常条件。创建一个异常，指的是引发一个异常或抛出一个异常。引发一个异常的通常情况是，向一个对象发送一条消息，而对象没有响应该消息；或者

试图使用超出数组边界的一个索引，来访问一个NSArray中的对象。如果不对异常做任何处理，该异常将会导致程序终止。

Objective-C提供了一种机制，它使用编译器指令@try、@catch和@finally来处理异常。这些指令在处理异常时都会引入程序清单10.2所示的一段代码。

注意 Mac OS X在10.3之前的版本并不支持异常。如果使用异常，程序只能够在Mac OS X 10.3及其以后的版本上运行。

程序清单10.2 异常/ SimpleExceptionHandler.m

```
id hamburger = @"hamburger";

@try
  {
    hamburger = [hamburger addCheese];
  }
@catch( NSException *exc )
  {
    NSLog( @"Exception: %@ : %@", [exc name], [exc reason] );
  }
@finally
  {
    NSLog( @"This block is always executed.");
  }
```

可能引发问题的代码位于@try块中。在程序清单10.2中，执行如下一行：

```
hamburger = [hamburger addCheese];
```

将引发一个异常，因为在这个示例中，hamburger是一个NSString，并且NSString没有实现方法addCheese。

@catch块捕获了该异常。异常总是用一个异常对象抛出，异常对象变成了@catch块的参数。运行时和Cocoa框架使用类NSException的实例作为异常对象。在这里，@catch块只是使用来自NSException对象的信息来显示异常。

@finally块中的代码总是会执行。如果@try或@catch块包含了一条return语句，则返回延迟直到@finally块中的代码执行之后。在程序清单10.2中，@finally块显示了一条消息，并且，程序继续执行@finally块之后的语句。

注意 在一个多线程程序中，只有当引发异常的代码在与@try/@catch/@finally相同的线程上执行时，才会捕获异常。

10.6.1 抛出自己的异常

运行时系统和Cocoa框架在各种遗憾的情况下为你抛出异常。然而，你也可以使用@throw编译器指令抛出自己的异常，如下面的示例所示：

```
- (void) addOnion
{
  if ( [pantry numOnions] <= 0 )
    {
      NSException* exception =
        [NSException exceptionWithName:@"OutOfOnionException"
                     reason:@"PantryEmpty" userInfo:nil];

      @throw exception;
    }
  else
    {
      ... // Add the onion
    }
}
```

> NSException
>
> NSException是一个Foundation类，它包装了关于异常的有用信息。当创建一个NSException时，可以指定异常的名称、发生异常的原因，并且提供一个NSDictionary来保存额外的信息。尽管在抛出自己的异常时，可以使用任何对象作为异常对象，但运行时系统和Cocoa框架总是使用NSException。

前面的示例使用一个NSException对象（参见上面关于NSException的内容），但是，可以使用任何的Objective-C对象作为异常对象。例如，可以使用NSException的一个定制子类或者一个NSString作为异常对象：

```
@throw @"Sorry. We're out of Onions";
```

@catch语句块用来捕获一个异常，该异常必须有一个参数类型，与@throw语句所使用的对象的类型一致。如果像前面的代码行一样，抛出了一个NSString作为异常对象，那么相应的@catch必须如下所示：

```
@catch( NSString* s )
{
  ...
}
```

10.6.2 多个@catch语句块

可以设置自己的代码针对不同的情况抛出不同类型的异常对象。然后，可以使用多个@catch块，每个块针对一种不同类型的异常对象，而使用一个单个的@try语句块，如下面的示例所示：

```
@try
  {
    ...
  }
```

```
@catch (NSString* myString)
  {
    // Process exceptions thrown with an object that is an
    // NSString or a subclass of NSString
  }
@catch (NSException *myException)
  {
    // Process exceptions thrown with an NSException object
  }
@finally
  {
    ...
  }
```

当有多个@catch语句块时，与异常对象匹配的异常，被第一个@catch语句块（按照程序中的顺序）捕获。匹配的条件是，异常对象是@catch语句块中所使用的类型的类的一个成员，或者是其一个子类的成员。如果抛出异常的代码所使用的对象与任何@catch语句块都不匹配，则不会捕获异常。程序立即终止，并且@finally语句块中的代码也不会执行。

10.6.3 嵌套异常处理

一个@try语句块中的代码可能自身包含了一组@try/@catch/@finally，相当于：

```
@try
  {
    @try
      {
      }
    @catch( NSException *exc )
      {
      }
    @finally
      {
      }
  }
@catch( NSException *exc )
  {
  }
@finally
  {
  }
```

注意 这只是一个简单的说明。内部的@try/@catch/@finally可能实际上在外围@try块的代码中埋藏得更深。

当引发一个异常时，第一个@catch块捕获了它（在栈中，从引发异常的地方向上工作），其参数类型与异常对象类型匹配。@catch块自身可能抛出一个异常（最初的异常或者是一个

新的异常），它可以由位于栈中较高位置的一个例程所包含的另一个@catch语句块捕获以进一步处理。

注意 与最初@catch语句块对应的@finally语句块中的任何代码，在处理新的异常之前执行。

框架中有什么类型的异常

Apple文档并没有运行时和Cocoa框架可能引发的所有异常的一个完整列表。各种单独的框架方法可能引发的异常，在那些方法的文档中加以说明。

10.6.4 使用异常

要在程序中使用异常，必须告诉编译器，将要使用它们。如果从命令行开始编译，必须把标志fobjc-exceptions传递给gcc。如果使用Xcode，就可以在GCC的Build Settings的Language部分选中Enable Objective-C Exceptions复选框，如图10-1所示。

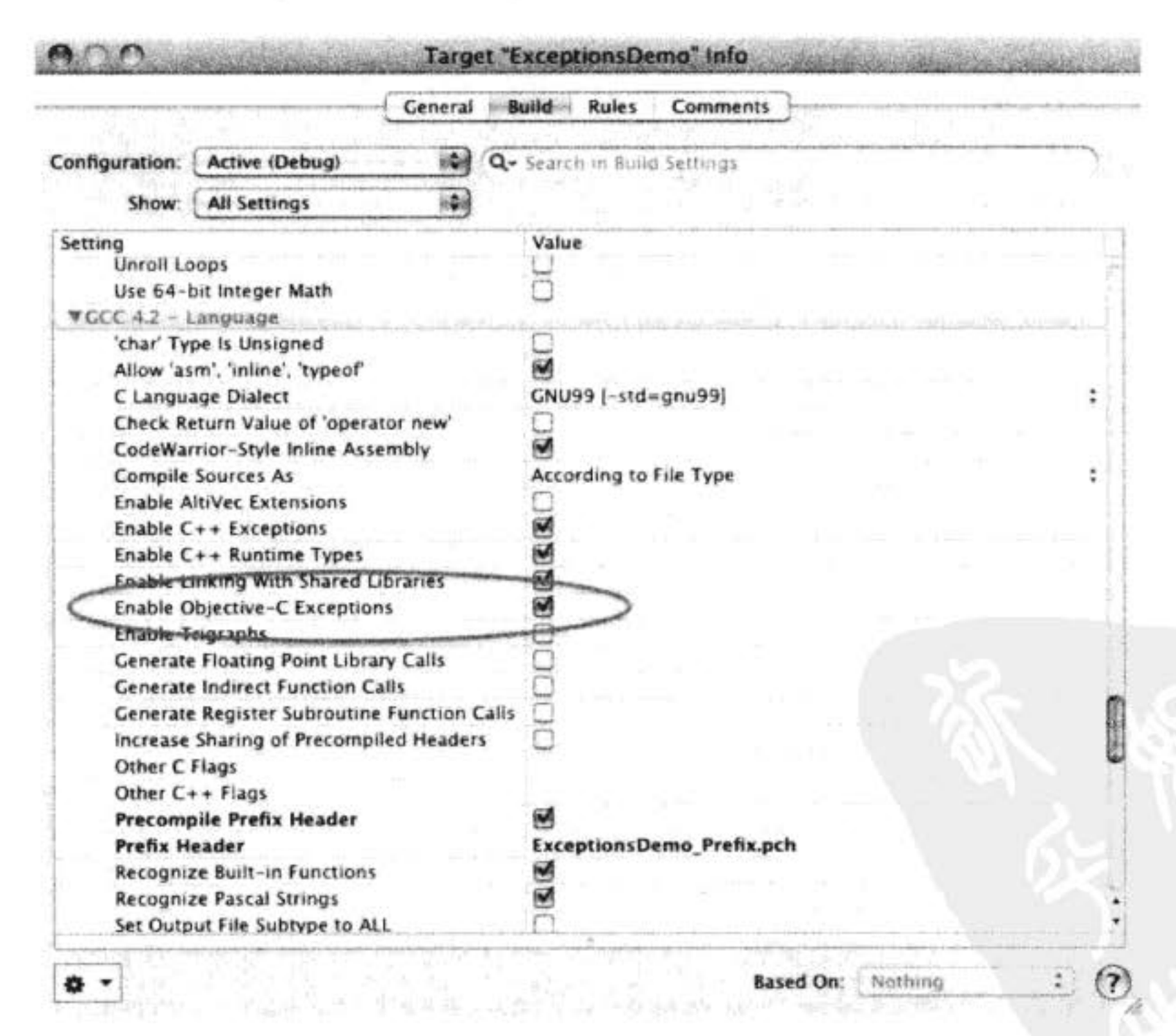


图10-1 在Xcode中支持异常

10.6.5 应该使用异常吗

我们已经了解了异常的所有内容，但这里必须告诉你，它们在Objective-C中很少用到。特

别是，不能使用它们进行普通的流控制。一些语言，如Python，针对异常处理进行了优化。Objective-C没有这么做。Objective-C中的异常处理很昂贵。

Objective-C中的异常主要设计用来捕获程序错误或其他的不可恢复的问题。在能够提前测试异常条件的情况下，应该先测试，而不是添加代码来捕获潜在的异常。

例如，如果有一个对象类型为id，并且，不确定它是否将响应一条特定的消息，则可以使用respondsToSelector方法来测试该对象，如下面的示例所示：

```
id someObject;

if ( [someObject respondsToSelector: @selector(addCheese)] )
  {
     [someObject addCheese];
  }
else
  {
    NSLog(@"Sorry. Can't add cheese to %@", [someObject description]);
  }
```

上面的代码，比在一个@try语句块中放置如下一行，要好很多：

```
[myObject addCheese];
```

一般的错误条件应该由return值传递出来：

```
- (BOOL) addOnion
{
  if ( [pantry numOnions] <= 0 )
    {
      return NO;
    }
  else
    {
     ... //  Add the onion
     return YES;
    }
}
```

如果需要有关错误条件的信息，可以给该方法添加一个参数，以传递回对一个NSError对象的引用，该对象封装了有关错误的信息。

使用异常的一个问题是，引发异常的代码可能在栈中比捕获异常的@catch语句块要低几个层级。当引发异常时，执行会跳到@catch语句块，绕过了栈中的所有中间程序。如果绕过了包含清理任务的代码，或者期望要执行的其他代码，那么当从@finally语句块之后继续执行时，程序可能会处于一种不稳定的状态。程序可以继续，但是，它可能会产生错误的结果，或者在稍后的某个时刻崩溃。尤其是在使用Cocoa框架的情况下，这更是一个问题，所绕过的代码并非设计用来以这种方式工作的。

10.7 小结

本章介绍了Objective-C环境中的循环和分支。要点列举如下：

- 在需要一个真值的普通C控制结构中的任何位置，可以使用Objective-C布尔类型BOOL。然而，应该避免显式地与YES或NO比较。
- 当比较两个对象时，需要决定是否需要指针相同（两个对象是相同的对象）或是相等（两个对象不同，但是就你的目的而言，它们是相等的）。
- 可以使用一个普通的for循环来遍历一个集合，但是，如果集合是一个NSArray或一个NSSet，并且循环体是一条单独的消息，则使用makeObjectsPerformSelector:的一个隐式循环会使代码更整齐。
- 在Objective-C 2.0之前，遍历一个集合的最佳方式是使用将while循环和NSEnumerator对象结合使用。NSEnumerator对象是从集合对象中获取的，它使用nextObject方法将集合的对象每次一个地分发出来。当集合中所有的对象分发完之后，nextObject返回nil。
- Objective-C添加了快速枚举和for...in语法。快速枚举比传统的循环结构性能更好。
- Objective-C实现了一个使用@try/@catch/@finally指令的异常处理系统。

Objective-C异常很昂贵。只有在真正异常的情况下，才应该使用它们，而对于普通流程控制来说，不应该使用。

第11章将介绍如何给一个已有的类添加方法，并且还将介绍Objective-C的安全性能（某种程度上比较有限）。

10.8 练习

1. 编写一个程序，它包含如下的代码：

```
NSArray *arrayOfStrings =
    [NSArray arrayWithObjects: @"", @"Objective-C", nil];

for ( NSString *string in arrayOfStrings )
  {
    if ( [string length] == YES )
      NSLog( @"%@ has non-zero length!", string );
    else
       NSLog( @"%@ has zero length!", string );
  }
```

验证程序运行很糟糕，然后，修改该程序，使它能够正确地工作。

2. 创建一个NSString对象的数组（使用喜欢的字符串）。使用一个普通的C for循环和NSArray的objectAtIndex:方法，计算数组中的字符串所保存的字符的总数。
3. 重新编写练习2中的程序，改用一个while循环和一个NSEnumerator对象。
4. 使用一个for...in循环重新编写练习3中的程序。
5. 创建如下的可变数组：

```
NSMutableArray *animalArray =
    [NSMutableArray arrayWithObjects: @"Lion", @"Tiger",
        @"Elephant", @"Duck", @"Rhinoceros", nil];
```

使用一个for...in循环遍历animalArray，并尝试将数组中超过5个字符长度的所有字符串删除。若直接在for...in循环中这么做，将会发生什么情况？

6. 使用练习5中的数组，编写一个程序，它正确地从数组中删除任何长度超过5的字符串。通过在删除字符串之前和删除之后，使用如下的一条语句来显示animalArray，以查看你的代码的工作情况：

```
NSLog( @"%@", [animalArray description] );
```

你的程序应该仍然使用一个for...in循环。

7. 创建一个包含3个字符串的数组，例如：

```
NSArray *duckArray =

    [NSArray arrayWithObjects: @"Huey", @"Dewie",
                               @"Louie", nil];
```

接下来，建立@try/@catch/@finally语句块。@catch语句块可以捕获一个NSException对象并且显示异常的名称和原因。@finally语句块应该包含一条显示语句，以便你可以知道其语句块是否已经执行。在@try/@catch/@finally语句块之后，放置另一条显示语句。

在@try语句块内部放置一条语句访问数组。尝试访问位于索引2的数组元素。验证@finally块中的代码已经执行了。接下来，修改你的代码，并试图访问索引3位置的数组元素，看看会发生什么情况。

第11章 分类、扩展和安全

我们在第6章中学习了定义类和分配、初始化及复制对象实例的基本知识。但是，还有比简单对象创建更多的内容要学习。本章介绍如何向一个已有的类添加一个方法，如何向（至少某些）偷窥者隐藏已有的方法，以及如何限制访问一个对象的实例变量。本章还将介绍Objective-C的安全性（以及这方面的不足）。

11.1 分类

分类允许我们向一个已有的类添加方法，而不用子类化它，并且不必访问类的源代码。使用一个分类来扩展一个类的行为，这是比子类化更轻量级的过程。

为了便于说明，这里假设你将要接受一个类似“`Four score and seven years ago`”的字符串，并且将其转换为“`FourScoreAndSevenYearsAgo`”。将最初的字符串中的每个单词都转换为首字母大写，然后压缩到一起形成一个单独的字符串块（没有空格），这样的一个字符串，叫做CamelCase字符串（之所以这么叫，是因为它看上去很像是骆驼的驼峰）。

注意 Objective-C类的名称遵守一般惯例CamelCase。Objective-C方法名称通常是CamelCase的一个变体，其中第一个单词的首字母不是大写的。

应该做的第一件事情是，查看`NSString`的文档，看看`NSString`是否已经有一个`-camelCase`方法。遗憾的是，它没有。

下一个想法可能是通过子类化`NSString`来添加一个`camelCase`方法。这是一个糟糕的想法，原因如下：

- 正如在第9章中讨论的，`NSString`是一个类簇。尽管不是不可能，但子类化一个类簇不是小事情，它需要一些正式的思路才能正确地实现。
- 如果通过子类化来添加方法camelCase，想要转换为CamelCase字符串的每个字符串，都将必须是一个CamelCase子类的实例，而不是一个普通的`NSString`的实例。如果从其他类接受字符串，则这一需求可能会成为负担。大多数处理字符串的类，都返回普通的`NSString`对象。

总而言之，对于这一任务来说，子类化是太笨拙的方法。相反，使用一个分类很容易。必须做的只是在一个`@interface`部分中提供类声明，并且在一个`@implementation`部分提供该方法的代码。运行时随后将分类方法添加到类中。之后，分类方法的行为就好像它是由最初的程序员编写到类中的一样。

如下是`NSString`的一个CamelCase分类方法的示例。分类的头文件和常规类的头文件类似，如程序清单11.1所示。

程序清单11.1 NSString+CamelCase.h

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

@interface NSString (CamelCase)

-(NSString*) camelCaseString;

@end
```

分类头文件必须为要扩展的类导入头文件。在这个示例中，NSString的头文件通过Foundation/Foundation.h导入。@interface行包含了要对其进行添加的类的名称(NSString)，以及分类的名称（CamelCase）。分类名称是任意的（可以称其为InitialCase，或者任何你想要的名称）。这个示例遵从对分类文件的常用命名惯例，ClassName+CategoryName，即头文件的名称是NSString+CamelCase.h。

注意 分类和子类之间的一个较大的差别是，分类没有向类添加任何变量。头文件反映出了这一点：它没有实例变量部分。

现在来看看分类的实现。分类实现文件（如程序清单11.2所示）看上去就像一个普通的类实现文件。它必须导入相应的头文件。方法实现位于@implementation指令和@end指令之间。唯一的区别在于，@implementation行在类名后的括号中有一个分类名称。

程序清单11.2 NSString+CamelCase.m

```
1 #import "NSString+CamelCase.h"
2
3 @implementation NSString  (CamelCase)
4
5 -(NSString*) camelCaseString
6 {
7   NSString* capitalizedString = [self capitalizedString];
8
9   NSArray* components =
10    [capitalizedString componentsSeparatedByCharactersInSet:
11       [NSCharacterSet whitespaceCharacterSet]];
12
13  NSString* output = @"";
14
15
16  for ( NSString* word in components )
17    {
18      output = [output stringByAppendingString: word];
19    }
20
21  return output;
22 }
23 @end
```

现在，我们来看看程序清单11.2中的代码：

- 第7行使用NSString方法capitalizedString来获取一个新的字符串，它将最初的字符串中的每个单词的首字母转换为大写的字母，并且将字符串中的其他字母设置为小写。注意，和任何其他方法一样，分类方法可以使用self来引用调用它的接收者。分类方法也可以与常规方法一样访问一个类的实例变量。
- 第9行得到了一个数组，其中包含了最初的字符串中的每个单词，它们分别作为一个单独的字符串。从第16行开始的循环，将单个的单词连接在一起，形成一个字符串（它将字符串连接起来，并且删除掉单词之间的空格）。
- 第21行返回了最终的结果。stringByAppendingString:返回的字符串已经自动释放了，因此，可以直接返回它。

现在，可以在任何NSString上使用该分类了，如下面的代码所示：

```
#import "NSString+CamelCase.h"

NSString* string = @"Four score and seven years ago";
NSLog( @"%@", string );
string = [string camelCaseString];
NSLog( @"%@", string );
```

如果运行包含前面代码段的代码，将会看到如下的结果：

```
Four score and seven years ago
FourScoreAndSevenYearsAgo
```

11.1.1 使用分类覆盖方法

也可以使用分类来覆盖一个类所声明的已有方法：

- 这么做有一些潜在性的危险。除非非常确定地知道该方法在所有情况下将会做什么，以及该方法用于何处，否则，这可能会破坏某些事情。
- 当使用一个分类来覆盖一个方法时，新的版本替代了最初的方法。分类的版本没有办法调用最初的方法。但是，如果最初的版本自身也是覆盖了一个超类的版本，那么分类版本仍然可以通过super来调用该方法的超类版本。
- 对于一个给定的类，可以有任意多个分类，但是，每个分类必须有一个不同的名称。
- 不能使用一个分类方法来覆盖另外一个分类方法。如果在不同的分类中用相同的名称声明了方法，则可能不会导致编译器或连接错误，但是，所使用的该方法的版本，则取决于分类载入的顺序。

应该使用分类还是子类？

正如R. Crumb的Mr. Natural所说："工欲善其事，必先利其器。"[⊖]但是，什么是利器呢？因为工作的不同而变化，对于利器来说，也没有确定的答案。但是，有些事情是应该

⊖ 参见http://en.wikipedia.org/wiki/Mr._Natural_(comics)。

考虑的：

- 简单就好。如果分类方法能够做该工作，就用它。加长一个类的继承链，可能会使代码难以维护。
- 实现为类簇的类，例如NSString，，需要很多工作来子类化。如果可以的话，使用分类。
- 分类有广泛的可见性。在分类中添加到一个类中的方法，可以由该类的任何实例使用，继承该类的其他类的任何实例也可以使用它。这可能使得能够使用该方法的范围，比你所想要的还要广。如果想自己在NSObject上添加一个分类，要三思而后行。这样的一个分类肯定不是一个好的想法，因为它会影响到程序的每个对象。
- 使用一个分类覆盖可能是危险的。当使用一个分类来覆盖类中一个已有的方法时，要特别小心。在替代最初的方法之前，应该确切地知道它做什么及用于何处。
- 添加实例变量需要子类化。如果需要添加实例变量来扩展一个类，将需要子类化它。
- 有的时候，没有选择而只能使用子类。如果需要在程序的不同部分，用不同的方式来扩展同一个类，可能必须使用子类。
- 一些类有意要扩展为子类。一些类部分地或完全地抽象，并且需要子类化它们以使其完全发挥作用。一个例子就是NSView，要进行定制绘制或响应鼠标手势，就必须子类化NSView。
- 尝试组合。也可以通过组合来扩展一个类。使用组合（它是一种设计模式，而不只是一种语言功能），可以创建一个全新的类，并且将想要扩展的类的一个实例保存到新的类的一个实例变量中。最初的类可以处理的消息，通过外围类来传递给它。调用新的功能的消息，由外围类处理。

这是一个很长的列表，但是万变不离其宗，即在编写代码之前要仔细思考。

11.1.2 分类的其他用法

分类还有其他的一些用法：

- 一个较大的类可能有几百个方法和上千行实现这些方法的代码（一个极端的例子请看AppKit类NSWindow的头文件，/System/Library/ Frameworks/ AppKit.framework/Headers/NSWindow.h）。可以使用分类将这样的一个类分解为几个在大小上更加便于管理的文件。实例变量和基本方法放入到常规接口文件和实现文件中。其他的方法，则分解为有意义的组，分别放入到一系列分类文件中。这就允许多个程序员同时操作该类。
- 分类也可以用来控制哪一个类方法对于类的用户是公有可见的。要做到这一点，将类方法分为公有的和私有的。将类的实例变量和想要公有的方法的定义，放入到一个常规的头文件中。构成类的私有API的方法，放入到一个分类文件中。
- 然后，将唯一的公有头文件分发给类的潜在用户。但是，记住，这只是一个信息隐藏方案。如果某人通过偷窥而发现了一个私有方法的名称，那么没有什么办法可以阻止他们调用它。

- 最后，分类头文件（没有对应的实现文件）可以用来声明非正式的协议（见第13章）。使用分类头文件告诉编译器，协议中的方法是存在的，并且在采用该协议的类中实现了。

分类方法是头等公民

分类中定义的方法，拥有类接口中定义的方法的所有权利和权限。实际上，在运行时，二者之间没有区别。它们都能够完全访问类的任意实例变量。如果类子类化，也会继承它们。

11.2 扩展

假设要编写一个类，其他的程序员将要使用该类。你希望有某些方法仅仅供你内部使用。如果在自己的类的头文件中声明了这些方法，那么它们将对于使用该类的任何人都是可见的。扩展允许你通过在类的声明文件中添加一个接口部分，从而在公有视野之外声明方法（扩展将接口声明扩展到了.m文件中）。

扩展的一种常见用法是，提供一个内部的setter方法，来访问你希望公有且只读的实例变量。程序清单11.3给出了这样的一个类的头文件。该类针对实例变量readOnlyBool声明了一个getter，但是，它没有声明setter方法。这使得readOnlyBool实际上成为一个只读变量（至少对那些没有偷窥的人来说，是这样的。参见本章稍后的11.6节）。

程序清单11.3 ClassWithExtension.h

```
@interface ClassWithExtension : NSObject
{
  BOOL readOnlyBool;
}

- (BOOL) readOnlyBool;

@end
```

扩展看上去和分类声明一样，只不过它们没有名称。扩展放到了类的实现文件中，在@implementation指令之前，如程序清单11.4所示。注意，在@interface行中的空圆括号是必需的。

程序清单11.4 ClassWithExtension.m

```
#import "ClassWithExtension.h"

@interface ClassWithExtension ()

- (void) setReadOnlyBool: (BOOL) newValue;

@end

@implementation ClassWithExtension
```

```
- (BOOL) readOnlyBool
{
  return readOnlyBool;
}

- (void) setReadOnlyBool:(BOOL) newValue
{
  readOnlyBool = newValue;
}

- (id) init
{
  if (self = [super init] )
    {
      [self setReadOnlyBool: YES];
    }
  return self;
}

@end
```

可以将一个扩展当做是一个匿名分类，但是，还有一些区别：

- 扩展方法的实现必须放在相同文件的@implementation部分中。
- 和分类不同，编译器为你检查，如果你在扩展中声明了一个方法但是忘记实现它，则将会得到一条警告。

11.3 实例变量作用域（访问控制）

Objective-C提供了3条指令来控制对一个对象的实例变量的访问：

- @private

该实例变量只能在声明它的类的方法中访问。试图从一个子类访问它，将会导致一个错误。

- @protected

该实例变量在声明它的类的方法中及继承了该类的任何类的方法中可以访问。

- @public

该实例变量随处可以访问，即便是在声明它的类之外的代码中，或者该类的任何子类中。除了允许在声明类或其一个子类内部的方法中访问实例变量外，@public允许将指向该类的一个实例的指针当做是指向一个C结构的指针对待。可以像下面这样访问一个实例变量：

```
instance->instance variable;
```

不要直接访问实例变量

曾经告诉过你，可以直接访问直接变量，那么需要告诉你的是，不要这么做。访问一个类的内部，违反了封装的原则，而这是面向对象编程的基本原则之一。如果你直接访问一个类的内部工作，并且，该类的实现有了变化，你的代码将会出错。

在类的接口部分中，这些指令位于它们所描述的实例变量之前。一旦给定，那么这条指令就是有效的，直到一条不同的指令替代它。如果你没有提供一条指令，默认情况下将是@protected，如程序清单11.5所示。

程序清单11.5 DinnerDish.h

```
@interface DinnerDish : NSObject
{
    Chef* chef;
  @private
    Recipe* recipe;
    int     calories;
  @protected
    BOOL lowFat;
    BOOL spicy;
  @public
    NSString* description;
}

@end
```

在程序清单11.5中，DinnerDish的实例变量chef、lowFat和spicy都是@protected的；recipe和calories是@private的，而description是@public的。

一直以来，Objective-C程序员对于使用这些指令相当松懈（尽管有一个缓慢的变化过程）。大多数程序员似乎甘愿使用默认的@protected（参见后面的11.6节）。然而，如果你发现自己使用一个类，它拥有标记为@private的实例变量，那么，这是在提示你不要直接访问这些变量，或尝试在一个分类或子类中使用或修改它们（使用一个分类来访问子类中的私有实例变量，这肯定会破坏@private指令）。最初的程序员告诉你，这些实例变量不是类的公有接口的一部分，并且它们可能会在类的未来版本中发生变化。

11.4 对方法的访问控制

没有这方面的控制。Objective-C对于方法没有任何私有、保护的和公有的表示方式。如果你知道一个方法的名称，就可以使用它给任何选中的对象发送消息。如果编译器认为（通过查看当前作用域中的方法声明）该对象不能响应该方法，则会得到一条警告。但是，最终的测试是在运行时进行的，如果对象能够响应该方法，它就会响应。否则，对象要么转发消息（如果类的设计者安排这么做的话），要么引发一个异常。

可以通过将你认为私有的方法的声明放入到扩展中或私有分类文件中，从而尝试阻止类的用户调用该方法。但是，最终这种方法的作用就像是锁上了前门。它保证诚实的人仍然诚实，但是，如果某人决定调用你的一个“私有”方法，那是没有办法可以阻止他的。

11.5 命名空间

Objective-C目前没有使用命名空间。其他语言中的命名空间用来防止不同库中定义的对象之间产生名称冲突（Objective-C中的库叫做框架，参见第8章）。如果想要连接两个单独的框架，且每个框架都定义了一个Button类，则可能会遇到问题。大多数Objective-C框架在一个特殊方法的基础上解决这个问题，即在框架中所有类的前面添加两个或三个字母。Cocoa框架使用NS（当然，因为NeXTStep的原因）；例如，AppKit框架中按钮类名称为NSButton。

11.6 安全

正如前面所介绍的，实际上，相对来说，对@private、@protected和public指令感到烦恼的Objective-C程序员较少。原因之一是（除了懒之外），很容易使用一个分类击败他们。

程序清单11.6给出了带有一个秘密的类：

程序清单11.6 ClassWithSecret.h

```
@interface ClassWithSecret : NSObject
{
  @private
    int secretNumber;
}

@end
```

编译器不允许子类访问实例变量secretNumber，如果你试图这么做，将会得到一个错误。但是，如果你真的想要获得secretNumber，必须做的事情是，在ClassWithSecret上的一个分类中定义一些访问器方法，[⊖]如程序清单11.7和程序清单11.8所示。

程序清单11.7 ClassWithSecret+Cheating.h

```
@interface ClassWithSecret (Cheating)

- (int) revealSecretNumber;
- (void) changeSecretNumber:(int) newSecretNumber;

@end
```

程序清单11.8 ClassWithSecret+Cheating.m

```
#import "ClassWithSecret+Cheating.h"

@implementation ClassWithSecret (Cheating)
- (int) revealSecretNumber;
```

⊖ 在这里没有对访问器使用标准命名惯例。这是因为在一个私有头文件或扩展中，ClassWithSecret可能已经使用这些名字定义了它们自己内部使用的方法。这里不想意外地替代这些方法，因为如果它们有副作用，那么替换它们可能会破坏类。

```
{
  return secretNumber;
}
- (void) changeSecretNumber:(int) newSecretNumber
{
  secretNumber = newSecretNumber;
}

@end
```

分类方法工作起来，就好像ClassWithSecret的最初创建者已经将其作为ClassWith-Secret的一部分提供给你了。使用它们，可以直接访问secretNumber，或者生成ClassWithSecret的一个子类，它读取和写入@private变量secretNumber而不会引发编译器的警告；如程序清单11.9和程序清单11.10所示。

程序清单11.9 CheatingNosySubclass.h

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

@interface CheatingNosySubclass : ClassWithSecret
{
}

- (void) revealSecret;
- (void) resetSecretNumber:(int) newSecretNumber;

@end
```

程序清单11.10 CheatingNosySubclass.m

```
#import "CheatingNosySubclass.h"
#import "ClassWithSecret+Cheating.h"

@implementation CheatingNosySubclass

- (void) revealSecret
{
  printf("The secret number is... %d\n", [self revealSecretNumber] );
}

- (void) resetSecretNumber:(int) newSecretNumber
{
  [self changeSecretNumber: newSecretNumber];
}
@end
```

在一个Objective-C可执行程序中，运行时用来填充其结构的信息，该信息可供任何程序使用和读取。有几个命令行程序，它们读取这些信息，并且以人类可读的方式显示它们（参见“转储类”的补充内容）。

转储类

命令行工具otool（这是标准Mac OS X发布的一部分），可以显示一个Objective-C程序或框架中的实例变量和方法，虽然其格式有点难懂。甚至更好的方式是类转储(www.codethecode.com/projects/class-dump)，它打印出的信息，其格式如同Objective-C接口部分一样整齐，每个类一段信息。

otool和类转储对于教育目的很有用，但是，不应该使用发现的任何私有方法或框架。私有方法和未文档化的API，可能会变化而不给出提示。如果使用了它们，并且它们被修改了或消失了，其程序就可能会失效。

正如前面提到的，Objective-C不能对方法名提供任何访问控制。可以尝试将私有方法的声明隐藏到一个私有头文件中，或者隐藏到实现文件的一个扩展中的一个分类中。但是，如果某人使用类转储发现了该名称和签名，他就可以成功地调用该方法。编译器将会编译而不会给出警告。但是，可以通过构造一个定制的分类头文件，使它使用来自类转储的信息来声明该方法，从而避免这一问题。

不建议你采用这些做法中的任何一种，但是，你应该知道有这些方法。一些新手发现这有点令人震惊，但实际上，这不会引发太多麻烦。大多数人信奉这一点，即根据没有文档化的、其他人的内部代码或库来编写代码是糟糕的想法。

注意 如果App Store的一款应用程序使用一个私有的或未文档化的Apple方法，Apple将会拒绝它。

然而，还有一些地方需要小心。一个例子就是许可代码。尽管你的应用程序几乎不可避免地会被破解，但是，我们不希望它很容易就被破解了，因此，可以直接使用C编写你的许可代码或者类似的敏感内容。

11.7 从Objective-C调用C函数

新手常问的一个问题是，“我可以在一个Objective-C方法内部调用C函数吗？”我将这个问题划分为三个部分：技术上的、实践上的和哲学上的。

11.7.1 技术上

绝对可以。Objective-C是C的一个超集。可以在能够使用一条语句的任何地方，或一个表达式内的任何地方（只要函数返回的值的类型对于表达式来说是合适的），放置对C函数的一个调用。也可以使用一个对象指针作为一个C函数的参数，在函数的主体中向一个对象（作为一个参数传入的对象，或者在函数内部创建的一个对象）发送消息，或者使用一个对象指针作为返回值。

11.7.2 实践上

有时候，可能必须使用C函数。如果编写一个Mac OS X或iPhone应用程序，则可能需要使用Apple C语言库来完成某些任务。一个基本的例子是iPhone上的图形。目前，iPhone UIKit在绘制的方式上提供的很少。为了进行屏幕以外的任何位图绘制，必须使用Core Graphics C API。在Apple的世界之外，还有广泛的第三方C库，可以实现各种各样有用的任务。因为Objective-C与普通C互操作，可以连接和使用这些库，即便你不访问它们的源代码。

11.7.3 哲学上

现在，我们处在了危险的领域。一些面向对象编程实践者对于使用C函数一点也不开心。他们认为应该尽可能地避免普通C。并不是每个人都同意这一观点。一种好办法是，在一个Objective-C邮件列表中启动一个较长的线程，询问类似如下的问题："我有一个数学程序，需要在我的类中用作辅助工具。我应该完全使用C函数编写它，将代码放到我的实现文件中，然后从我的方法中调用该C函数吗？" OOP坚持者肯定会告诉提问者，包含C代码，并且使其成为类方法之一。

这真的是个人喜好（两种方法编写的代码都有效），但是，我持相反的观点。对于我来说，关于Objective-C的一件精彩的事情是，它不需要所有的事情都严格按照面向对象的方式处理。冒着遭受抨击的风险，我要说："面向对象编程是一种方法，而不是宗教。"它对于大多数任务来说，是极其有用的，但是对于某些任务来说，其他的方法可能更合适。很多数学计算本身是过程式的，并且，将它们放入到一个Objective-C包装器中就说它们是"面向对象的"，这不会有什么好处。如果你需要编写一个数学辅助函数，继续前进并用C编写它。如果你需要在多个地方使用这样的函数，那么将源代码放入到一个单独的.c文件中（或者，如果它引用Objective-C对象的话，将其放入到一个.m文件中），并且将函数声明放入到一个匹配的头文件中。然后，可以包含头文件并在任何需要的地方使用该函数。

11.8 小结

分类允许我们给一个已有的类添加方法，而不需要访问该类的源代码。这提供了一种方式来扩展类，而不需要子类化它们；新的方法变成了已有的类的一部分。分类头文件，只是方法声明的列表，可以用来将较大的类划分到几个文件中，或者为公有方法和私有方法提供单独的头文件。

扩展提供了另一种方法来隐藏私有方法的声明。Objective-C方法只有在其声明的可见性的意义上是私有的。知道方法名称的程序员，就可以成功地调用它，即便该方法的声明是不可见的。Objective-C使用@public、@protected和@private编译器指令提供对实例变量的访问控制；然而，可以使用一个分类方法来绕过它们并访问@private变量。

11.9 练习

1. 在NSString上编写一个分类方法，用来将字符串中的单词的顺序反向。反向分类方法应该将

字符串“dog bites man”变成字符串“man bites dog”（最初的字符串是接收者，并且，反向的字符串是方法的返回值）。编写一个较小的程序来测试-reverseWords。

2. 编写一个类，它有一个扩展，并且编写一个程序来测试它（你的类不一定要做任何有用的事情，你可以只是在方法体中放入printf或NSLog语句）。
   - 暂时忽略在扩展中声明的方法的实现。验证编译器会对此给出警告。
   - 添加扩展方法的实现，并且尝试从测试程序调用它。验证编译器会警告你，对象可能不会响应该方法。然后提示你，不管怎样你可以成功地调用该方法。
3. 验证实例变量作用域规则。定义一个类，它拥有@private、@protected和@public实例变量。然后：
   - 尝试从类的外部直接访问每个实例变量（object->instance variable）。
   - 创建初始类的一个子类。尝试在一个子类方法中访问每个实例变量。
4. 编写一个分类，它在你在练习2中定义的类上偷窥@private实例变量。使用程序清单11.7和程序清单11.8中给出的方法。
5. 下载class-dump (www.codethecode.com/projects/class-dump)。针对你在本书练习中编写的一些Objective-C程序运行它，看看输出的样子。class-dump是一个命令行程序，它可以使用如下命令从一个终端窗口运行：

```
class-dump executable name
```

   如果你使用过Xcode，就会知道一个项目的可执行文件与项目具有同样的名称，并且，它位于项目目录的build/[configuration name]目录中。默认的配置名称是Debug，因此，除非你创建一个不同的配置，否则可执行文件将位于你的项目目录的build/Debug 子目录中。

第12章 特 性

最常见的编程任务之一，就是设置或访问在一个对象的实例变量中存储的值。尽管在某些情况下可能是直接访问实例变量，但这么做是糟糕的做法。访问一个实例变量的正确方式是编写访问器方法。设置一个实例变量的值的访问器方法叫做setter，而返回一个实例变量的值的访问器方法叫做getter。

例如，如果对象有一个整数实例变量numberOfDogs，就应该声明并实现如下的两个方法：

```
-(int) numberOfDogs;  // returns the current value
-(void) setNumberOfDogs:(int) newNumberOfDogs; //  sets a new value
```

注意 将getter命名为instanceVariableName和将setter命名为setInstanceVariableName是惯例，而不是Objective-C语言所必需的。但是，这是你应该学习遵守的惯例，因为某些高级Cocoa功能（如键-值编码）依赖于这一惯例。键-值编码允许你使用一个NSString键来表示将要访问的变量，从而设置和获取一个实例变量的值。

Aaron Hillegass所著的《Cocoa Programming for Mac OS X, Third Edition》一书（Addison-Wesley, 2008）的第7章介绍了键-值编码。

使用访问器的缺点之一是，它们的编码很繁琐，特别是当类拥有大量的实例变量时。Objective-C 2.0引入了一种叫做声明特性的新功能。它添加了两条新的语句，来减少编写和实现访问器所必需的工作：

- @property语句提供了声明访问器方法的一种快捷方式。
- @synthesize语句引发编译器为你编写指定的访问器方法。

注意 Apple将这一功能称为“声明特性”，但是，为简单起见，本章后面的部分将使用更简单的“特性”。

本章首先展示如何不访问实例变量，然后展示编写访问器方法的正确方式。本章将详细介绍新的声明特性功能。介绍如何声明特性，如何让编译器创建访问器方法，以及何时需要自行编写访问器。本章最后的部分介绍Objective-C 2.0的另一项新功能：点语法。点语法针对使用传统的方括号消息表达式来调用访问器方法提供了一种替代方法。

12.1 在对象之外访问实例变量（不要这么做）

假设你要构建一个业务系统的一部分，它负责记录公司的雇员。从一个雇员的简单模型开始，它存储了雇员的名称、ID编号、上级和薪水等。

程序清单12.1给出了一个Employee类，它用来声明实例变量以便保存这些信息。

程序清单12.1 Employee.h

```
@interface Employee : NSObject
{
  int employeeNumber;
  NSString *name;
  Employee *supervisor;
  int salary;
}

...

@end
```

假设你想要使用这个类的实例来计算总的年薪（你为一家负责的公司工作，并且管理层想要看看公司是否有足够的钱运营一年）。刚刚学习面向对象编程的C程序员，可能会尝试直接在一个循环中访问salary实例变量，并且将其加到总的年薪中，如程序清单12.2所示。

程序清单12.2 在对象之外访问实例变量

```
NSArray *allEmployees = // ...
int payroll = 0;

for ( Employee* employee in allEmployees )
  {
    payroll += employee->salary; // BAD PRACTICE!
  }
```

不要这么做。这从技术上讲是可能的，尽管实际上你会得到一条编译器警告，[⊖]因为Employee类中的实例变量默认是@protected的（别忘了，@protected的变量只能由属于该类的方法或者属于该类的一个子类的方法访问）。你甚至可以通过进入Employee.h并声明该实例变量是@public的，从而删除警告，但是，这真是一种可怕的做事方式。

面向对象编程的原则之一是封装。你应该能够使用一个对其内部工作一无所知的Employee对象。通过这种方式，如果类实现中的某些内容改变了，使用该类的代码也不会失效。

12.2 声明和实现访问器

由于我们不打算直接访问Employee实例的实例变量，因此必须编写一些访问器方法（也就是说，setter和getter）来操作实例变量。程序清单12.3给出了Employee.h的一个修改版本，它添加了访问器方法的声明。

⊖ 尝试直接访问一个@protected实例变量，可能会是一个错误，但是，编译器的当前版本只能给出一条相关的警告，在未来，这将是一个错误。

程序清单12.3 Employee.h (访问器)

```
@interface Employee : NSObject
{
  int employeeNumber;
  NSString *name;
  Employee *supervisor;
  int salary;
}

- (void) setName:(NSString*) newName;
- (NSString*) name;
- (void) setSupervisor:(Employee*) newSupervisor;
- (Employee*) supervisor;
- (void) setSalary:(int) newSalary;
- (int) salary;

...

@end
```

注意，没有用于employeeNumber实例变量的setter。一个雇员的工资、上级甚至名称都有可能改变，但是，可能雇员的编号是在一个人入职的时候分配的，不会改变（当创建一个Employee实例时，employeeNumber实例变量在init方法内初始化）。

访问器的形式

getter的典型形式，只是返回实例变量，如程序清单12.4所示。

程序清单12.4 getter的形式

```
- (Employee*) supervisor
{
  return supervisor;
}
```

在大多数情况下，存储在一个实例变量中的对象，在存储的时候应该保留；当用一个新的对象替代它的时候，应该释放。setter的典型实现形式，如程序清单12.5所示。

程序清单12.5 使用保留的setter的形式

```
- (void) setSupervisor:(Employee*) newSupervisor
{
  [newSupervisor retain];
  [supervisor release];
  supervisor = newSupervisor;
}
```

向存储在supervisor中的当前对象发送一条release消息，会对应当存储它的时候所接受的一条retain消息。上面的代码中的语句顺序很重要。如果supervisor和

newSupervisor都是相同的对象，那么在保留newSupervisor之前释放supervisor将导致该对象在保留之前销毁掉。这会在supervisor中存储指向一个销毁了的对象的指针，并且如果supervisor随后在一个消息表达式中用作接收者，将会导致一次崩溃。

在某些情况下，需要存储对象的一个副本而不是保留它。当想要保存对象的当前状态的一个快照时，就会这么做。使用复制的setter的形式如程序清单12.6所示。

程序清单12.6 使用复制的setter的形式

```
- (void) setName:(NSString*) newName
{
  NSString *tmp = name;
  name = [newName copyWithZone: nil];
  [tmp release];
}
```

注意 必须确保所复制的任何对象都属于实现了copyWithZone:的类。

简单的访问器的形式只是模板，但是，可能会有很多的模板。在Employee类中，仅仅针对4个变量实现了访问器的代码，就占了几乎整整一页。在下一节中，你将会看到，在很多情况下，Objective-C 2.0的新特性功能免除了这样的苦力。

12.3 使用特性的访问器

可以使用@property语句在接口部分声明访问器，并且在实现部分使用@synthesize指令告诉编译器创建它们的实现，从而重新编写上述的Employee类。程序清单12.7给出了修改后的接口部分。修改用粗体显示。

程序清单12.7 使用@property语句编写的Employee.h

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

@interface Employee : NSObject
{
  int employeeNumber;
  NSString  *name;
  Employee  *supervisor;
  int salary;
}

@property (nonatomic, readonly) int employeeNumber;
@property (nonatomic, retain)  NSString *name;
@property (nonatomic, retain)  Employee *supervisor;
@property (nonatomic, assign) int salary;

...

@end
```

@property语句替代了访问器方法的声明。这是一个改进。它需要的代码较少：单独一行就声明了getter和setter（如果有的话）。它还易于阅读，并且，另外的好处是，它包含了有关如何构造访问器的一些信息（不同的属性，即圆括号中的项，将在下面介绍）。

实现文件中的代码的数量真的令人吃惊，如程序清单12.8所示。

程序清单12.8 Employee.m

```
#import "Employee.h"

@implementation Employee

@synthesize employeeNumber;
@synthesize name;
@synthesize supervisor;
@synthesize salary;
...

@end
```

这就是所有的代码。@synthesize语句放入到@implementation部分。它导致编译器为你创建访问器方法。不用再干苦力活了。

更加具体地说，实例中的如下代码导致编译器为两个方法setName:和name创建代码，如下所示：

```
@synthesize name;
```

这两个方法将在运行时存在。可以调用它们，就好像你自己编写的一样。

我们甚至可以节省更多的录入，对多个特性，使用一条单个的@synthesize：

```
@synthesize employeeNumber, name, supervisor, salary;
```

合成访问器方法不是必需的。在某些情况下，我们需要自己编写它们，以执行简单的set或get之外的处理。在这种情况下，可以使用一条@dynamic语句告诉编译器，你将提供访问器：

```
@dynamic propertyName;
```

也可以省略该语句，@dynamic是默认的。如果编译器没有看到@synthesize或@dynamic，它假设@dynamic并且期望你为特性的访问器提供实现（参见12.5节）。

注意 如果你@synthesize一个特性，并且还针对一个或两个特性的访问器提供了一个实现，编译器将使用你所提供的代码。例如，如果你使用@synthesize并且为相应的setter提供了实现，那么编译器将使用你的setter实现并且合成getter。

实例变量名称可以与特性名称不同

有可能对一个特性及表示它的实例变量使用不同的名称。如果想要使用不同的名称，你就必须在@synthesize语句中通知编译器：

```
@synthesize propertyName = instance variableName;
```

在Employee示例中（见程序清单12.7和程序清单12.8），可以将supervisor实例变量重命名为boss，但是，针对该特性保留名称supervisor。修改后的接口部分如下所示：

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

@interface Employee : NSObject
{
  int employeeNumber;
  NSString *name;
  Employee *boss;
  int salary;
}

@property (nonatomic, readonly) int employeeNumber;
@property (nonatomic, retain)  NSString *name;
@property (nonatomic, retain)  Employee *supervisor;
@property (nonatomic, assign) int salary;

...

@end
```

并且，实现部分如下所示：

```
#import "Employee.h"

@implementation Employee

@synthesize employeeNumber;
@synthesize name;
@synthesize supervisor = boss;
@synthesize salary;
...

@end
```

合成的访问器仍然是supervisor和setSupervisor，boss只是在内部使用。

12.4 @property语句

@property声明的形式是：

```
@property ( attributes )  type name;
```

type和name的含义一目了然。attributes描述了如何编写访问器，后面的小节将介绍它。

12.4.1 assign、retain和copy

这些属性影响到合成的setter如何构建。如果使用assign，setter直接将新值赋值给

你的特性：

```
myProperty = newValue;
```

assign属性是一个非对象特性的唯一选择。

对于在引用计数下的对象特性，assign创建了一个弱引用（参见第14章）。

如果使用retain并要求编译器@synthesize你的访问器，它创建一个等价于本章前面的程序清单12.5中所示的setter。在垃圾收集下（第15章），retain与assign相同，并且，我们不需要使用它，除非编写必须在引用计数和垃圾收集下都能工作的框架代码。

如果使用copy并且要求编译器来@synthesize你的访问器，它创建一个等价于程序清单12.6中所示的setter。

注意 要使用copy，特性的类型必须是能够复制的类。类必须通过实现copyWithZone：而采用NSCopying协议（参见本书第6章）。

如果没有显式指定assign、retain或copy，编译器假设一个默认值assign。这对于基础类型不是问题。然而，对于引用计数下的对象特性，如果你没有指定三者之一，那么将会得到一个编译器警告。

如何在复制和保留之间做出选择？这取决于你的对象的设计，以及特性表示什么。如果特性要用作一个可变对象的当前状态的快照，则应该复制它。

如果用来设置一个特性的对象是要对一个实体自身建模，而不是对实体的当前状态建模，则应该保留它。想象一下类型Employee的业务系统有几种带有特性的对象（president、vicePresident以及employeeOfTheMonth等）。如果关于雇员的某些内容发生了变化，而你不想在整个系统中进行查找以更新一个雇员的所有副本。在这种情况下，使用retain（或者，如果想要使用一个弱引用的话，使用assign），将会确保用来保存一个给定的雇员的所有实例变量都引用Employee的同一个实例。

12.4.2 readwrite和readonly

如果一个特性声明为readwrite的，则该特性的值可以设置也可以获取，两个访问器都合成，这是默认的。如果特性声明为readonly的，只有getter合成。如果试图使用方法（不存在的）setMyProperty:，则你的程序将会由于一个"does not respond to selector"异常而崩溃。

12.4.3 nonatomic

这个属性有点奇怪，因为它没有对应的atomic关键字。没有声明为nonatomic的特性，默认都是atomic的。如果指定了nonatomic，则编译器合成访问器而不考虑线程安全性。

当在多线程程序中访问一个特性时，有可能在完成设置和获取之前，一个线程被另一个线程中断了。如果第二个线程在中断过程中修改了特性，那么在第一个线程中返回的或设置的值可能会不一致或错误。为了防止这种情况，可以声明特性为atmoic的（通过省略关键字nonatomic）。然后，合成的访问器将是atomic的，这样设置或获取将会正确地完成，而不

会受到另一个线程的打扰。

对于一个非对象特性或者当使用垃圾收集的时候，getter是一条简单的返回语句，而setter是一次简单的赋值，并且总是atomic的。对于引用计数下的对象特性，还会有更多的困难。合成的setter和getter使用锁来防止其他线程中断设置或获取。此外，getter保留然后自动释放返回的对象。

注意 在getter部分的行为，防止了另一个线程释放该对象，从而潜在地导致在你的线程有机会使用或保留它之前，其引用计数减少到了0。

锁在性能方面付出了代价，如果你不打算编写多线程代码，则应该将自己的特性声明为nonatomic的。然后，合成的访问器不会使用锁。getter直接返回对象，并且，setter看上去如程序清单12.5和程序清单12.6所示。

12.4.4 setter=name和getter=name

通常，特性myProperty的合成名称，对于setter和getter来说，分别是setMyProperty和myProperty。setter=name和getter=name允许你提供其他的名称。其主要用途是允许针对一个布尔特性的getter，例如caffeinated，可以命名为isCaffeinated，如下所示：

```
@property (nonatomic, getter=isCaffeinated) BOOL caffeinated;
```

这提高了代码的可读性。

12.4.5 attributes和@dynamic

如果提供了自己的访问器方法的实现，那么编译器不会检查你提供的方法是否与对应的@property声明中的attributes一致。它默默地允许你做一些不正确的事情，例如，对声明为copy的特性赋值，或者为声明为readonly的特性提供一个setter方法。

12.5 关于@dynamic的更多内容

与@synthesize相对的是@dynamic。当你为一个特性指定@dynamic时，编译器期待你为该特性提供你自己的访问器方法实现。如果你没有这么做，将会在编译时得到一条警告，然后，当你试图使用该特性时，会产生运行时错误。

编译器能够为我们工作，我们为何还要编写访问器方法呢？编译器只是合成相对简单的访问器，它们只是设置和返回特性。有的时候，我们可能需要做更多的事情。当设置一个特性的时候，可能需要采取一些其他的动作。例如可能想要过滤输入，以确保设置的特性处在一个可接受的范围之内。

考虑保存喇叭的音量设置的一个特性：

```
#define kMaxVolume 1.0
#define kMinVolume 0.0
{
```

```
  float speakerVolume;
}
@property (atomic, assign) float speakerVolume;
```

如果可接受的值范围在0.0～1.0，可以编写一个setter，将新值限制在这一范围中：

```
- (void) setSpeakerVolume: (float) newVolume
{
  // protect against out of range input
  if (newVolume > kMaxVolume)
    {
      speakerVolume = kMaxVolume;
    }
  else if (newVolume < kMinVolume)
    {
      speakerVolume = kMinVolume;
    }
  else
    {
      speakerVolume = newVolume;
    }
}
```

编写自己的访问器的另一个原因是，可能甚至没有与一个特性对应的实例变量。特性实际上是一个对象的公共接口的一般性描述。它们表示了一个对象的特征和状态。但是，特性不需要明确对象的内部结构。特性通常存储在实例变量中（并且，本章的大多数讨论都关注这种情况），但是，它们不一定必须如此。它们可以存储在对象所拥有的一个辅助对象中，存储在一个外部文件中，或者甚至临时计算。只要getter返回了特性，并且setter（如果有一个的话）设置了它，其他的无关紧要。

在如下的一个示例中，特性由计算的getter返回。考虑一个RestaurantEmployee类：

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

@interface RestaurantEmployee : NSObject
{
  int age;
  ...
}

@property (nonatomic, assign) int age;
@property (nonatomic, readonly) BOOL canServeBeer;
...

@end
```

在某些地方，还有一些规定，卖酒的雇员必须达到一定的年龄，因此，应该定义一个只读的布尔特性canServeBeer。不一定真的将canServeBeer存储到它所拥有的实例变量中，可以根据age实例变量来计算它，如下所示：

```
#import "RestaurantEmployee.h"

@implementation RestaurantEmployee

@synthesize age;
@dynamic canServeBeer;

- (BOOL) canServeBeer
{
  if ( [self age] >= 21 )
    return YES;
  else
    return NO;
}

...

@end
```

12.6 特性和内存管理

使用特性和合成setter（或正确地手动编写setter），在引用计数以外还花费了很多工作。只要你确保总是使用一个实例变量的setter方法来设置其值，那么大多数引用计数会自动发生。

注意 使用访问器方法是好的做法，即便是从一个方法中访问一个实例变量。这一规则的一个例外是，当在init方法中初始化一个实例变量，或者在一个dealloc方法中释放它时。这些实际上都是内部操作。在初始化或销毁的典型情况下，我们不想要或不需要调用访问器方法所引起的副作用。在init方法中，应该直接设置实例变量（如果必须，就保留要存储的对象）。在dealloc方法中，应该直接释放实例变量，而不是使用一个nil参数调用实例变量的setter。

dealloc

如果能够在编译器为你做所有工作时，出来透透气并喝杯咖啡，这将会很棒。尽管特性朝着这个方向迈出了一步，但是仍然必须自己做一些事情。编译器能够合成访问器方法，但是它不会完全负责内存管理。如果你的类有任何特性保留或复制了，你必须记得在该类的dealloc方法中释放这些对象。即便这样，特性也是很有帮助的。看一下特性的声明，可以提醒你需要释放哪些对象。

程序清单12.8中的Employee类的dealloc方法如下所示：

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

12.7 子类化和特性

子类可以从其超类继承任何特性（及其访问器）。可以在子类的头部重复property语句（为了文档化的目的），但是，除非初始头文件和子类头文件中的property语句相同，否则编译器会抱怨。

有一种例外情况，即在子类中，可以将一个readonly特性声明为readwrite的。这可以使你遵从常见的设计模式，即拥有一个不可变的类的可变子类。

例如，从一个Shape对象开始，它在创建时给出一个固定的颜色：

```
@interface Shape : NSObject
{
  NSColor* color;
  ...
}

@property (nonatomic, retain, readonly) NSColor* color;

  ...
@end
```

现在，让我们定义Shape的一个子类，ColorableShape，它允许修改颜色：

```
@interface ColorableShape : Shape
{
}
@property (nonatomic, retain, readwrite) NSColor* color;

...

@end
```

这是对应的@implementation部分：

```
@implementation ColorableShape

@dynamic color;

- (void) setColor:(NSColor*) newColor
{
  if ( color != newColor)
    {
      [newColor retain];
      [color release];
      color = newColor;
    }
}

@end
```

在这里必须要注意，如果在ColorableShape中使用了@synthesize而不是@dynamic，并且对setColor:提供了自己的实现，编译器也会合成颜色的getter。新的getter将会覆盖超类中的。如果Shape中的颜色特性的getter也合成了，那么两个版本将会是相同的。唯一的代价是，可执行文件中多出一些额外的字节。但是，如果Shape版本有一些额外的行为，那么这些行为将会在ColorableShape中丢失。

12.8 对readonly特性隐藏setter

可以使用类似的方法为公开声明为readonly的特性提供一个隐藏的setter（供内部使用）。通过在类的.m文件中的一个扩展中（参见第11章），重新声明readonly特性，可以做到这一点。要让程序清单12.7和程序清单12.8中的Employee类的employeeNumber成为内部可写的，可以对程序清单12.8做如下修改：

```
#import "Employee.h"

@interface Employee ()
@property (nonatomic, readwrite) int employeeNumber;
@end

@implementation Employee

@synthesize employeeNumber;
@synthesize name;
@synthesize supervisor;
@synthesize salary;
...

@end
```

现在，编译器会合成两个访问器。setEmployeeNumber:在实现文件中可供内部使用，但是对于Employee类的用户来说是不可见的。

注意 正如第11章提到的，没有什么能够阻止Employee类的用户调用setEmployeeNumber，如果他通过猜测或者通过转储类而知道存在该方法的话。

12.9 特性作为文档

除了其他的优点，特性还可以作为类的文档的一种形式。它们比它们所替代的方法声明更加紧凑并且可读性更好，并且，它们带有更多的信息。它们告诉我们一个特性是否是read-only的，关于如何存储特性的一些信息，以及其访问器是否是线程安全的。

看一下Employee类中employeeNumber特性的特性声明：

```
@property (nonatomic, readonly) int employeeNumber;
```

它立即告诉我们，employeeNumber是read-only的。不必再查看整个头文件，来看看

getter是否有一个对应的setter。[⊖]这对于本章中的较小的示例类来说，显然是微不足道的，但是，在真实系统中的头文件，可能会非常的大。例如Cocoa的NSWindow类的头文件NSWindow.h，有超过650行的代码，并且差不多有250个方法声明。

12.10 点语法

除了特性，Objective-C 2.0还引入了一种新的语法，叫做点语法，它是调用访问器方法的一种简便语法。

不必再编写这样的代码：

```
Employee *employee =  // ...

int currentSalary = [employee salary];
[employee setSalary: 100000];
```

而是可以这样编写：

```
Employee *employee =  // ...

int currentSalary = employee.salary;
employee.salary = 1000000;
```

在运行时，点语法纯粹是“语法糖”。这两个示例产生几乎相同的可执行代码。

语法糖

语法糖是向计算机语言添加的一种语法，它使得用该语言编写的代码对于人类程序员来说更加可读。它对编译器所产生的二进制文件或最终程序的执行没有影响。例如，我们在第1章介绍过，常规的C数组的元素实际上是通过指针来访问的。

如下的两条语句做同样的事情。它们都将一个整数数组someArray的第3个元素设置为17：

```
*(someArray + 2) = 17;
someArray[2] = 17;
```

第二种形式使用方括号，这就是语法糖。它与指针表达式产生相同的可执行代码，但是它更易于阅读。

尽管使用点语法在运行时没有什么不同，但是，当编写程序时，它是有好处的。在编写代码时，使用点语法要求的输入较少，这对于缺乏耐性或关节有问题的人来说是一个优点。并且，当熟悉了点语法之后，它使得代码更容易阅读。当需要引用一个特性的特性（属于另一个特性）时，这更加形象直观。例如，定义Employee的Manager子类，使它带一个额外的assistant特性：

```
@property(nonatomic, retain) Employee *assistant;
```

⊖ 头文件中方法声明的顺序对于最终的二进制文件没有影响，但是，考虑到某人会查看文件，相关的方法应该放置得相对较近。然而，遗憾的是，这与其他的理想情况一样，并不是每个人都遵守这一原则。

像下面这样设置assistant的薪水：

```
Manager *manager = //...
manager.assistant.salary = 50000;
```

这比下面的形式容易输入和阅读：

```
Manager *manager = //...
[[manager assistant] setSalary: 50000];
```

点语法允许编译器执行更严格的代码检查。如果试图像下面这样设置一个read-only特性：

```
Employee *employee = // ...
[employee setEmployeeNumber: 999];
```

将会得到一条警告：

```
warning: 'Employee' may not respond to '-setEmployeeNumber:'
```

并且，如果随后尝试运行程序，它将会崩溃，因为该特性是只读的并且没有-setEmployeeNumber:。

然而，如果试图使用点语法来设置一个只读的特性，如下所示：

```
Employee *employee = // ...
employee.setEmployeeNumber = 999;
```

编译器将产生一个错误，而不是给出一条警告：

```
error: object cannot be set - either readonly property or no setter found
```

编译器防止程序将来崩溃。

类似地，像下面这样试图设置一个不存在的特性：

```
[employee setCaffeinationLevel: 100];
```

也只是得到一条编译器警告（并且，再次会导致运行时崩溃）；而如下的语法则会得到一个编译器错误：

```
employee.setCaffeinationLevel = 100;
```

12.10.1 点语法和特性

本章结合特性讨论了点语法，但是这两个概念实际上是彼此不相关的。可以直接使用property语句定义的访问器，而不需要使用点语法。点语法与特性没有依赖关系。如果编写如下代码：

```
int raise = // ...
Employee *employee = // ...

int salary = employee.salary;
salary += raise;
employee.salary = salary;
```

所有计算完全取决于Employee是否定义和实现了两个方法salary和setSalary:。是使用@property语句还是使用传统方式声明的这两个方法，无关紧要。方法实现是合成的还

是显式编码的，也无关紧要。

12.10.2 点语法和C结构

点语法有时候会导致很多抱怨。一些人根本不喜欢语法糖。其他的人则指出，点语法在C中已经有了很好的含义：它是访问一个结构的成员的方式。他们反对的理由是，点语法妨碍了可读性，因为要正确地解释如下的一个表达式：

```
foo.bar.baz
```

必须要分别了解foo、bar和baz是一个对象还是一个结构。他们进一步的观点是，这会产生二义性，并且会使得该语言的初学者感到困难。在各种博客上，关于这一话题的双方都发表了很多的观点和议论。这一主题正在成为另一场（较小规模的）计算机科学宗教战争。但是，不管你是否选择在自己的代码中使用点语法，你都应该能够阅读和理解它，因为Apple在其示例代码中广泛地使用点语法。

我不想为那些恐糖症[⊖]的人做任何事情，但是，关于二义性的问题，你（即编译器）可以通过查看表达式中的每个部分的类型来正确地解析一个点表达式。如果定义了一个结构：

```
struct vector3
{
  float x, y, z;
};
```

可以将其用作类中的一个特性：

```
@interface Rocket : NSObject
{
  vector3 velocity;
  ...
}
@property (nonatomic) vector3 velocity;
  ...
@end
```

要获得一个火箭速率的垂直部分，可以这样编写代码：

```
Rocket *rocket = // ...
float verticalVelocity = rocket.velocity.z;
```

从左向右，rocket.velocity是一个vector3结构。因此，第二个点号是结构成员运算符。

点语法不能直接访问实例变量

用于访问器方法的点语法与引用一个C结构的元素的相似性，导致某些人认为点语法可以直接访问实例变量。如果是这样，它将会违反“信息隐藏”原则，但是实际上并非如此。点语法只是调用相应的访问器方法。任何副作用（保留、释放、复制或其他的）发生的时候，就好像你已经用传统语法编写了这些语句。

⊖ saccharophobia：恐糖症。

必须注意的一种情况是，当在对象自己的方法中使用其实例变量时，考虑程序清单12.1的Employee类。在Employee的一个方法中，编写如下代码：

```
name = newName;
```

来通过赋值直接设置实例变量name。

像下面这样使用点语法：

```
self.name = newName;
```

使用-setName:方法设置name。这并非细微的差别：在第二个示例中，保留了newName；而在第一个示例中没有这样。

12.11 小结

本章主要介绍了如何使用新的Objective-C 2.0声明特性功能，来设置和获取在对象的实例变量中存储的值。关于访问实例变量和使用特性，有如下一些要点：

- 应该通过使用访问器方法来获取实例变量的值，从而访问一个对象的实例变量。不应该从对象外部直接访问实例变量。
- @property语句允许以一种简便的形式声明访问器方法。
- 特性声明提供了关于访问器方法的额外信息，例如，对象实例变量是赋值、保留还是复制，以及访问器是否使用锁来保证原子性。
- 当使用一个特性声明来声明访问器时，可以通过使用一条@synthesize指令，要求编译器为访问器方法创建代码。合成的访问器反映了在@property语句中指定的属性。
- 使用@synthesize创建非常基础的访问器，它只是设置和获取特性值。如果在访问一个特性时，需要进行额外的处理（例如，将一个输入值限定在指定的范围内），可以省略@synthesize语句（或者使用@dynamic），并且自行编写该特性的访问器代码。
- Objective-C 2.0还引入了点语法，这是调用访问器方法的一种快捷方式。

12.12 练习

1. 定义并编写一个TripDuration类，它有表示一段旅行的距离、旅行的速度及旅行的名称的实例变量。TripDuration应该包含一个方法，以根据公式time=distance / speed计算并返回旅程需要多长时间。不使用特性，声明并实现访问器方法。

   之后，编写一个使用TripDuration类的小测试程序，设置距离、速度和旅行名称，然后获取旅行时间。还应该通过打印出距离、速度、时间和旅行名称来显示TripDuration。

2. 使用特性重新编写TripDuration。让duration成为一个只读的特性（即便没有对应的实例变量）。在相应的地方使用@synthesize。

3. 注释掉@synthesize语句，并且尝试构建自己的程序。会发生什么情况？
4. 你不想拿到快速票，因此，重新编写TripDuration以将速度限制为一个合理的值。此外，如果速度为0，旅行将会花费无限的时间。计算机不喜欢无限，因此，确保速度有某个最小值（这意味着，速度必须是@dynamic的，并且必须自己编写访问器）。
5. 保持之前的TripDuration版本，重新编写测试程序，在调用TripDuration的访问器时使用点语法。

第13章 协 议

协议是一组预定义的方法，一个类可以选择来实现它们。当对象需要与一个或多个对象对话，但是不知道（或不是特别关心）对象的类时，协议特别有用。Objective-C协议相当于Java的接口。实际上，Java的设计者从Objective-C借用了这一概念。协议在整个Cocoa框架中使用。Objective-C拥有正式的和非正式的协议。正式的协议声明了一组必需的方法，可能还有某些可选的方法。随后，类可以通过实现协议所必需的所有方法来采用该协议。当一个类声明它采用一个正式的协议时，编译器会检查，确保该类实现了协议所必需的所有方法。非正式的协议则只是在一个子类头文件中的一个方法声明的列表，一个类可以选择来实现这些方法。Cocoa中使用的大多数非正式协议将要被正式协议所替代。

在本章中，我们将学习如何使用一个正式协议，如何声明一个正式协议，以及如何编写采用一个正式协议的类。然后，本章还将介绍一个完整的示例，一个`TablePrinter`类。`TablePrinter`对象可以以表格的形式打印另一个对象中的数据。`TablePrinter`不需要知道提供数据的对象的类。`TablePrinter`需要知道的唯一的事情就是，提供数据的对象属于采用了`TablePrinterDataSource`正式协议的一个类。

13.1 协议

例如，假设你想要设计一个草图（机械绘图）系统。你可能要求用户从头开始，用线条和弧线绘制所有内容，但是，如果你能够为他们提供一个常用部件的库，即具体的细节，而这些部件都知道如何绘制自己，那么，这将是很方便的。我们将这些部件称为“可绘制项”。

进行绘图的视图对象并不真正关心一个可绘制项属于哪个类。它只是关心作为一个可绘制项的对象能够响应一组特定的消息。一个可绘制项应该能够返回一个边界矩形（以便你可以知道它是否在屏幕上可见），设置并获取其颜色，并且绘制自身。实现了这些方法的任何类的实例，都可以充当一个可绘制项。[⊖]

如果将这些方法收集到一起，并且给该集合起一个名称，你就有了一个协议。一个类实现了一个协议的方法，就说这个类采用了该协议。在本章稍后将会看到一个类在其接口部分声明它所采用的协议。

> **为何不只是使用继承呢？**
>
> 为什么不只是让程序中的所有可绘制项都继承自一个单个的`DrawableItem`抽象类，而让该抽象类声明所有必需的方法呢？

⊖ 这显然是一个高度简化的示例。在真实的绘制系统中，可能需要方法来设置部件的位置和方向，还要有方法能够获取大小以及其他信息。

这是可能的，如果你要设置一个较小的系统，只带有数目有限的绘制项，那么，可以这么做。然而，在其他情况下，子类化是笨拙的解决方案，这表示你需要绘制的各种部件的类应该在你开始程序之前就创建了（由于其他原因，或者根据其自身的对象层级）。它们可能已经由一个第三方提供了。与重构整个类层级从而在接近顶部的地方放入一个抽象类相比，添加功能来实现一个协议是更整齐划一的解决方案。协议之美就是只与功能相关，而与类无关。

13.2 使用协议

本节我们将继续探讨可绘制项的示例，以说明声明和使用协议的方法。在本章稍后，我们将详细介绍一个不同的示例协议。

13.2.1 声明一个协议

协议名称放在尖括号中。一个可绘制项的协议可能是<DrawableItem>这样的名字。

通过在@protocol和@end编译器指令之间声明其方法，从而声明一个协议。对于<DrawableItem>协议，声明如下所示：

```
@protocol DrawableItem
- (void) drawItem;
- (NSRect) boundingBox;
- (NSColor*) color;
- (void) setColor:(NSColor*) color;
@end
```

协议声明放在一个头文件中，以便可以将这一声明放入到一个名为DrawableItem.h的头文件中。没有对应的实现文件。采用该协议的每个类，针对协议方法提供了它自己的实现。

在Objective-C 2.0之前，采用一个协议的类需要实现协议声明的所有方法。Objective-C 2.0允许将协议方法标记为可选的或必需的：

- 采用一个协议的类，必须实现协议的所有必需方法。
- 采用一个协议的类，可以自由地实现或不实现协议的任何可选方法。

可选的和必需的方法分别使用@optional和@required指令来标记。按照文件中声明这些方法的顺序来考虑它们。在@required指令后面声明的方法是必需的。类似地，在@optional指令后面声明的方法是可选的。如果没有指定任何内容，则默认方法是必需的。

在绘制草图的示例中，一些可绘制项可能提供文本以用作标签。例如，可以使用一个可选的方法来获取文本，如下所示：

```
@protocol DrawableItem

@required
- (void) drawItem;
- (NSRect) boundingBox;
```

```
- (NSColor*) color;
- (void) setColor:(NSColor*) color;

@optional
- (NSString) labelText;
@end
```

如果一个特定的<DrawableItem>类有某些标签文本，则它实现了-labelText方法；否则，它没有实现它。检查一个给定的<DrawableItem>是否实现了-labelText，由使用<DrawableItem>的任何类决定。

13.2.2 采用协议

类通过实现协议的所有必需方法及任意的或根本没有实现协议的可选方法，从而采用该协议。通过在类的@interface行添加协议名称，并用尖括号将其括起来，从而告诉编译器你的类采用了一个协议：

```
@interface AcmeUniversalBolt : AcmeComponent <DrawableItem>
```

如果一个类采用了协议，或者如果它继承了采用了该协议的一个类，这个类就遵从该协议。

类可以采用多个协议。在协议的每对尖括号之间用逗号隔开。如果AcmeUniversalBolt遵守NSCopying协议（通过实现copyWithZone:），也遵从<DrawableItem>，那么其@interface行将如下所示：

```
@interface AcmeUniversalBolt : AcmeComponent <DrawableItem, NSCopying>
```

13.2.3 协议作为类型

可以通过在类型声明中添加协议名称，来将一个变量的类型声明为，保存遵从了协议的一个对象：

```
id <DrawableItem> currentGraphic;
```

这不会告诉编译器类currentGraphic是什么，只是说它遵从<DrawableItem>协议。如果试图给currentGraphic赋值一个不遵从<DrawableItem>协议的一个对象，那么，编译器会给出警告。

下面这句可能更严格：

```
AcmeComponent<DrawableItem> currentGraphic;
```

这告诉编译器，currentGraphic是AcmeComponent的一个实例，或者是其一个子类的实例，并且其类实现了<DrawableItem>协议。

注意 这种结构很少见到。协议的主要思想是，遵从协议的对象的用户对类毫无所知，只知道它遵从该协议。

13.2.4 特性和协议

如果某些协议方法是访问器方法，可以使用property语句在协议头文件中声明它们。在

<DrawableItem>示例中，协议声明可能如下所示：

```
@protocol DrawableItem

@required
- (void) drawItem;
@property (nonatomic, readonly) NSRect boundingBox;
@property (nonatomic, retain) NSColor* color;

@optional
- (NSString) labelText;

@end
```

采用该协议的任何类，要么必须合成对应该特性的访问器，要么为其提供实现，就好像在类的@interface部分声明了该特性一样。

在Objective-C 2.0的当前版本中，不能将协议中使用的特性标记为@optional的。在这个例子中，必须让labelText保持一个常规的方法声明。

13.3 TablePrinter示例

让我们详细地看看另一个示例。TablePrinter类为其他对象提供了一个表格打印服务。它使用可选的标题和可选的行号来打印出字符串的一个单列表格。TablePrinter可以为任何对象创建一个表格，只要该对象遵从<TablePrinterDataSource>协议。这是开展业务的一种吸引人的方式。TablePrinter自身不存储表中使用的任何数据。它不必知道数据源是什么类，不需要知道该类的任何信息，只知道数据源可以回答某些问题（通过一条消息），例如“这个表中有多少行”或“行中有什么信息”。

> **注意** TablePrinter类以一种简化的方式模拟了AppKit的NSTableView（在桌面计算机上）和UIKit的UITableView（在iPhone上）。NSTableView和UITableView都是比TablePrinter更为复杂的类，但是它们都以与TablePrinter相同的方式来获取所显示的数据。它们接受一个期待遵守一个协议的dataSource对象，然后使用协议中的方法来查询数据源。

TablePrinter示例的代码在后面给出，并且它包含如下几个文件：

- TablePrinterDataSource.h定义了<TablePrinterDataSource>协议（见程序清单13.1）。
- TablePrinter.h和TablePrinter.m是TablePrinter类的接口文件和实现文件（见程序清单13.2和程序清单13.3）。
- FruitBasket.h和FruitBasket.m是一个FruitBasket类的接口文件和实现文件（见程序清单13.4和程序清单13.5）。
- TablePrinterExample.m是创建一个FruitBasket并使用一个TablePrinter打印出FruitBasket的内容的一个表格的主程序（见程序清单13.6）。

读完本部分，可以构建并运行该程序了。

13.3.1 TablePrinterDataSource

TablePrinter可以针对遵从<TablePrinterDataSource>协议的任何对象打印出一个表格，该协议如程序清单13.1所示。

程序清单13.1 TablePrinter/TablePrinterDataSource.h

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

@protocol TablePrinterDataSource

@required
- (NSString*) stringForRowAtIndex:(int) index;
- (int) numberOfRowsInTable;

@optional
- (NSString*) tableTitle;
- (BOOL) printLineNumbers;

@end
```

这个协议有两个必需的方法：

- numberOfRowsInTable告诉TablePrinter表格中的行号。
- stringForRowAtIndex:为请求的索引提供数据（NSString）。

有两个可选的方法：一个提供了表的标题(tableTitle)，一个指定了TablePrinter是否应该使用行号(printLineNumbers)。如果这两个方法没有实现，则TablePrinter使用“Table”作为标题，并且省略行号。

13.3.2 TablePrinter

程序清单13.2给出了TablePrinter类的头文件。

程序清单13.2 TablePrinter/TablePrinter.h

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

@protocol TablePrinterDataSource;

@interface TablePrinter : NSObject
{
  id <TablePrinterDataSource> dataSource;
}

@property(nonatomic, assign)  id <TablePrinterDataSource> dataSource;
- (void) printTable;
@end
```

有一个单独的实例变量来存储数据源、数据源的访问器方法以及实际打印表格的方法。实例变量和特性的类型是id <TablePrinterDataSource>，表示存储在这里的任何对象都期望其遵从该协议。如下的代码行是一个前置声明：

```
@protocol TablePrinterDataSource;
```

它告诉编译器，是的，TablePrinterDataSource是一个协议。这是编译器需要知道的头文件中的所有内容。

注意 dataSource的特性声明是assign而不是retain。这是因为，在一个更加实际的程序中，数据源是主要对象，它持续存在，并且表格打印只是一个（可能是临时的）用来显示主要对象的内容的附件。在一个变量中保存一个对象而不保留该对象，这叫做对该对象进行弱引用（见第14章）。

TablePrinter类的实现文件如程序清单13.3所示。

程序清单13.3 TablePrinter/TablePrinter.m

```
#import "TablePrinter.h"
#import "TablePrinterDataSource.h"

@implementation TablePrinter

@synthesize dataSource;

- (void) printTable
{
  NSString* separator = @"-------------------------";
  NSString* title = @"Table";

  if ( [dataSource respondsToSelector: @selector( tableTitle )] )
    {
      title = [dataSource tableTitle];
    }
  printf( "\n%s\n%s\n", [title UTF8String],
                        [separator UTF8String] );

  int numRows = [dataSource numberOfRowsInTable];
  int j;
  BOOL printLineNumbers = NO;

  if ( [dataSource respondsToSelector: @selector(printLineNumbers)] )
    {
      printLineNumbers = [dataSource printLineNumbers];
    }

  for ( j=0; j < numRows; j++ )
    {
```

```
    NSString* outputString = [dataSource stringForRowAtIndex:j ];

    if ( printLineNumbers )
      {
        printf( "%d | %s\n", j+1, [outputString UTF8String] );
      }
    else
      {
          printf( "%s\n", [outputString UTF8String] );
          }
        }
    }

    @end
```

printTable方法打印出标题，询问表格中的行号，然后循环，向数据源请求在每一行中打印出的字符串。

有如下几件事情需要注意：

- 必须导入协议的头文件，以便编译器能够检查你在调用协议定义的方法。
- 在调用可选的协议方法tableTitle和printLineNumbers之前，printTable使用NSObject的respondsToSelector:来查看dataSource是否实现了它们。如果dataSource没有实现这些可选的方法，则printTable对tableTitle和printLineNumbers使用默认值。
- 使用printf()而不是NSLog()，这纯粹是为了美观。printf()不会使用NSLog()添加到每一行的日期、时间和处理信息来分散我们的注意力。此外，由于printf()对NSStrings一无所知，因此可以使用UTF8字符串方法将输出字符串转换为C字符串，然后再将它们传递给printf()。

13.3.3 FruitBasket

FruitBasket是一个非常简单的类，它只是在数组中存储了一些水果名称。在这里，它用来测试TablePrinter类。它通过在其头文件的@interface行添加<TablePrinterDataSource>，声明它采用了TablePrinterDataSource协议，如程序清单13.4所示。

程序清单13.4 TablePrinter/FruitBasket.h

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

@interface FruitBasket : NSObject <TablePrinterDataSource>
{
  NSArray* fruits;
}
@end
```

程序清单13.5所示的FruitBasket.m，有具体实现。两个可选的方法都注释掉了，以模拟一个只是实现了协议必需的方法的类。暂且不实现可选方法。

程序清单13.5 TablePrinter/FruitBasket.m

```
#import "FruitBasket.h"
@implementation FruitBasket

- (id) init
{
  if (self = [super init] )
    {
      fruits =
        [[NSArray alloc] initWithObjects:
            @"Apple", @"Orange", @"Banana",
             @"Kiwi", @"Pear", nil];
    }
  return self;
}

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

- (int) numberOfRowsInTable
{
  return [fruits count];
}
- (NSString*) stringForRowAtIndex:(int) index
{
  return (NSString*)[fruits objectAtIndex: index];
}
/*- (NSString*) tableTitle
{
  return @"Available Fruits";
}*/

/*- (BOOL) printLineNumbers
{
  return YES;
}*/

@end
```

13.3.4 main

程序清单13.6是项目的测试程序。它创建了一个TablePrinter对象，然后创建了一个FruitBasket对象，并将其作为数据源赋给TablePrinter。最后，要求TablePrinter打印出表格。

程序清单13.6 TablePrinter/TablePrinterExample.m

```
#import <Foundation/Foundation.h>
#import "TablePrinter.h"
#import "FruitBasket.h"

int main (int argc, const char *argv[])
{
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  FruitBasket* myFruitBasket =[[FruitBasket alloc] init];

  [myTablePrinter setDataSource: myFruitBasket];

  [myTablePrinter printTable];

  [myTablePrinter release];
  [myFruitBasket release];
  [pool drain];
  return 0;
}
```

运行该程序，将会看到如下结果：

```
Table
-------------------------
Apple
Orange
Banana
Kiwi
Pear
```

协议声明放于何处？

协议声明必须对于采用该协议的类的接口部分可见。在当前的示例中，TablePrinterDataSource协议的声明放在其自己的头文件中，即TablePrinterDataSource.h（见程序清单13.1）中，它随后导入到了FruitBasket.h（见程序清单13.4）中。作为替代，TablePrinterDataSource的声明可以放在TablePrinter.h（见程序清单13.2）中。随后，FruitBasket.h需要导入TablePrinter.h。通常，在Cocoa框架中，一个正式协议在一个类的头文件中声明，而这个类是遵从该协议的对象的主要用户。然而，少数Cocoa正式协议也在其自身的头文件中声明。

13.3.5 一个问题

尽管这个程序能够工作，但是应该注意到，当构建它的时候有一个小小的问题，编译器会给出如下的警告：

```
TablePrinter.m:23: warning:
     '-respondsToSelector:' not found in protocol(s)
```

编译器抱怨respondsToSelector:不是<TablePrinterDataSource>协议的一部分。它确实不是，但是这无关紧要，因为respondsToSelector:是由NSObject实现的，而每个对象（包括作为数据源传递的对象）都继承自NSObject。这不要紧，程序还是能够很好地工作，不是吗？是的。但是养成了忽略编译器警告的习惯是糟糕的，即便这个警告是半真半假的。如果你开始忽略警告，那么总有一天，就像太阳必然会升起来一样，你将会忽略那种不是伪造的警告而由此陷入麻烦。

你可以花点时间来修正它。在TablePrinter.h中，可以在实例变量和特性声明中修改类型：

```
NSObject <TablePrinterDataSource> *dataSource;
```

这明确地告诉编译器，dataSource是NSObject的一个子类，并且也实现了<TablePrinterDataSource>。一种更恰当的解决方案是，将TablePrinterDataSource.h中的@protocol行修改为：

```
@protocol TablePrinterDataSource <NSObject>
```

这意味着，采用了<TablePrinterDataSource>的任何类也采用了<NSObject>协议（这说明了重要的一点，即一个协议也可以采用另一个协议）。<NSObject>是NSObject对象所采用的一个协议（参见补充内容“NSObject和<NSObject>”）。再一次，由于NSObject实现了respondsToSelector:，因此你不必做任何额外的工作，让编译器觉得一切正常。

> **NSObject和<NSObject>**
>
> 这可能有点令人迷惑。<NSObject>是一个正式协议，它列出了任何类要成为一个良好的Objective-C公民所必须实现的方法。这是一些非常基础的方法，例如respondsToSelector:、superclass及引用计数retain和release（要了解这些方法的完整列表，参见头文件NSObject.h）。NSObject（类）采用了<NSObject>协议，然后大多数类通过（直接的或间接的）继承从NSObject获得这些方法。Foundation的另一个根类NSProxy，也采用了<NSObject>。

13.3.6 实现可选的方法

现在，通过在FruitBasket.m中取消掉相关的注释，来实现<TablePrinterDataSource>中的两个可选方法。这给表格添加了一个标题和行号。如果你再次编译程序并运行它，结果将会是：

```
Available Fruits
------------------------
1 | Apple
2 | Orange
3 | Banana
4 | Kiwi
5 | Pear
```

13.4 协议对象和一致性测试

协议对象是表示协议的对象。它们是类Protocol的成员。可以使用@protocol()指令，通过协议的名称来获取一个协议对象。

```
Protocol myDataSourceProtocol = @protocol( TablePrinterDataSource );
```

和类对象不同，协议对象没有方法，它们的用途仅限于作为NSObject方法conformsToProtocol:的一个参数。如果接收者实现了该协议中的所有必需的方法，则这个方法返回YES，否则返回NO。可以使用一个协议对象，通过自行编写dataSource实例变量的访问器方法，并做一些额外的工作，从而构建一个TablePrinter较安全的版本。更安全的setter如程序清单13.7所示。

程序清单13.7 针对TablePrinter数据源的一个更安全的setter

```
- (void) setDataSource: (id <TablePrinterDataSource>) newDataSource
{
   if ( ! [newDataSource
              conformsToProtocol: @protocol(TablePrinterDataSource)] )
     {
       dataSource = nil;
       NSLog(@"TablePrinter: non-conforming data source.");
     }
  else
    dataSource = newDataSource;
}
```

上面的setDataSource:的版本，防止了将没有遵从<TablePrinterDataSource>协议的一个对象当做dataSource来赋值。如果将这样的一个对象当做dataSource来赋值，则当调用一个必需的协议方法时，可能会导致程序崩溃。

NSObject还拥有一个类方法conformsToProtocol:，可以使用它来测试一个类（而不是类的一个实例）是否遵从一个协议。如下的代码将返回YES：

```
[FruitBasket conformsToProtocol: @protocol(TablePrinterDataSource)];
```

测试协议一致性是防御性代码的一个例子。使用区区数行额外的代码，就可以防止一次运行时崩溃。

13.5 非正式协议

到目前为止，本章所介绍的协议都是正式协议。正式协议使用一条@protocol语句声明。采用正式协议的类，在类的@interface行用一个<ProtocolName>标示出来。Objective-C还有非正式协议。和正式协议一样，非正式协议也是一个类可能要实现的一组相关的方法。非正式协议在一个分类中声明其方法（通常是NSObject上的一个分类），但是，没有对应的分类实现：

```
@interface NSObject (MyInformalProtocol)

- (void) informalProtocolMethod;

@end
```

非正式协议本质上是一部分编写采用该协议的类的程序员所达成的君子协定。程序员在编写该类时，同意实现非正式协议的方法，但是编译器对于他是否遵守协议不做任何检查。分类充当一段文档，来列出协议中的方法。在编译时没有类型检查，在运行时也没有办法来确定一个类是否实现了一个非正式协议。采用一个非正式协议的类，必须在其接口部分声明协议方法，并且在其实现部分放置这些方法的代码。

注意 当你编写采用非正式协议的一个类的代码时，协议方法的实现必须位于该类的@implementation部分中，而不是在一个分类的@implementation部分中。如果将实现放到了一个单独的分类@implementation部分中，那么你的方法实现将会添加到所有的类（假设该协议声明为NSObject上的一个分类）中。这可能不是你想要的结果。

那么，使用非正式协议的要点是什么呢？在Objective-C 2.0之前，一个正式协议中的所有方法都是必需的。让一个协议中拥有可选方法的唯一办法，是使用一个非正式协议，并且在文档中注明可选的方法。随着@optional和@required的出现，已经没有太多的理由再使用非正式协议了。你可以让方法可选，并且仍然享受一个正式协议的优点。

可以在Apple框架中看到这一进步。在AppKit的早期版本中，NSTableDataSource（AppKit的NSTableView的数据源协议）是一个非正式协议。从Mac OS X Snow Leopard (v 10.6)开始，NSTableViewDataSource正式协议替代了NSTableDataSource。

13.6 小结

协议通过允许你根据行为来分类对象，而不是根据类来分类对象，从而给程序设计增加了灵活性。在很多情况下，一个对象所需要做的只是实现特定的一组方法。在这些情况下，对象的类及其行为是非实质性的。协议允许我们将这些模式正式化：

- 通过在@protocol和@end指令之间给出一个名称并且声明其方法，从而声明一个协议。
- 协议方法要么是@required的，要么是@optional的。
- 类通过实现协议所必需的所有方法，并且可能实现协议的一些或所有可选方法，从而采用一个协议。类通过在类的@interface行的超类名称之后的一个尖括号中放置协议的

名称，从而声明其采用了该协议。包含了协议声明的头文件，对于采用协议的类的@interface部分，必须是可见的。

- 可以在一个实例变量或方法参数的类型声明中添加一个协议名称。如果你这么做，编译器将检查你赋值给该实例变量的一个对象或用作方法参数的一个对象，是否采用类指定的协议。
- 可以使用类或conformsToProtocol:的实例版本来查看一个给定的类或给定的对象是否采用了一个特定的协议。
- 在调用@optional之前，应该使用respondsToSelector:来确保接收者已经实现了该方法。
- 前面几点中描述的协议叫做正式协议。还有非正式协议，这只是类可以选择实现的方法的一个列表。没有编译器支持检查非正式协议。它们最初用来声明拥有可选方法的协议。随着Objective-C 2.0中的@optional指令的引入，我们没有理由再使用一个非正式协议而不使用正式协议了。

13.7 练习

1. 为库存系统设计并声明一个StockKeepingUnit协议。采用了<StockKeepingUnit>的一个类的实例，应该能够报告这样的内容：产品名称、厂商、价格和SKU号码。
2. 定义一个CheeseBasket类，它保存了奶酪的名称。确保它采用了TablePrinterDataSource协议。向TablePrinter示例（见程序清单13.6）添加CheeseBasket的一个实例，并且使用同一个TablePrinter实例来打印出FruitBasket中的水果，然后打印出CheeseBasket中的奶酪。这是一个简单的练习，但是它强调了一点，即TablePrinter可以使用任何类作为其数据源，只要该类实现了TablePrinterDataSource协议中必需的方法。
3. 定义一个类，它没有实现TablePrinterDataSource中必需的方法。尝试使用这个类的一个实例作为程序清单13.6中的TablePrinter的数据源，而不是使用FruitBasket的一个实例。那么编译该程序的时候将会发生什么？当运行该程序时，会发生什么？
4. 修改TablePrinter的源以使用dataSource实例变量的一个安全的setter（如程序清单13.7所示），并且重复前面的练习。

第三部分
高 级 概 念

第三部分的各章详细介绍了引用计数和垃圾收集，这是Objective-C内存管理的两种形式。第16章介绍了Objective-C新的块功能，它是Apple's Grand Central Dispatch并发机制的核心功能。

- 第14章　引用计数
- 第15章　垃圾收集
- 第16章　块

第14章 引用计数

居住在公寓中的人们经常面临着存储的问题。他们的东西总是比放东西的地方多。解决这个问题需要将储物柜中的东西换入和换出，并且定期检查公寓，丢掉那些不再需要的东西。管理计算机中的内存使用，就相当于解决公寓居住者的问题。

内存管理很重要，因为内存是有限的资源。尽管当前的计算机拥有的物理内存的数量，对于几年前来说似乎就是痴人说梦，但是，平均的程序内存需求和用户期望也在增长。要保持一个程序的内存占用尽可能的小从而提高程序的性能，需要减少计算机的虚拟内存系统必须将程序的一部分交换到磁盘随后再将其从磁盘读入。通过减少所有运行的程序对于物理内存的竞争，也可以提高系统的整体性能。

内存管理在Mac OS X上很重要，但是，它在iOS上也很关键。iOS拥有虚拟内存（每个进程有其自身的地址空间），但是它没有一个完整的页面系统（像程序文本这样的内存只读页面只能换出，但是没有页面用于内存的可写区域）。此外，众多iOS设备上的物理内存的量非常有限。iPhone和iPod touch的早期版本只有128MB的RAM（iPhone 3GS有256MB）。OS和任何在后台运行的Apple程序，使用了这些存储的大部分。你的程序可用的RAM量可能有40MB或更少。当剩下的未使用的内存量低于一个阈值时，OS会向所有运行的应用程序发送一个低内存警告。它期待你的应用程序力所能及地通过释放尽可能多的内存，来回应这样的警告。如果忽略了警告，并且持续低内存条件，OS将会终止你的应用程序。

对Objective-C来说，内存管理意味着记录已经创建的对象，并且当不再需要它们的时候释放它们。Objective-C 2.0为内存管理提供了两种替代系统：引用计数（也叫做保留引用或管理内存）和自动垃圾收集。使用引用计数的内存，每个对象保持使用自己的其他对象的数目的一个计数。当该计数达到0时，就销毁该对象。使用垃圾收集，有一个叫做垃圾收集器的单独线程，它定期唤醒，查看程序的内存，并且释放任何不再需要的对象。OS X上运行的一个Objective-C程序必须使用引用计数或垃圾收集，不能将两个系统混合到一个单独的程序中。针对iOS编写的程序，必须使用引用计数。垃圾收集当前在iOS上不支持。

本章将介绍引用计数。在本章中，将学习引用计数是如何工作的，引用计数的一些更加高级的内容，例如自动释放池、保持引用计数正确所必须遵守的规则及使用引用计数时可能发生的一些问题（例如，保留循环）。下一章将专门介绍Objective-C 2.0的自动垃圾收集系统。

14.1 问题

所有的Objective-C对象都是动态分配的，它们的字节来自于堆。如果你试图声明保存对象而不是指向对象的指针的一个自动变量，从而在栈上创建一个对象，例如：

```
NSString aString;
```

编译器将会抱怨并且拒绝编译该文件：

```
error: statically allocated instance of Objective-C class 'NSString'
```

创建一个对象的正确方式是，声明一个指针变量，并且使用方法alloc从堆上为你的对象分配内存。如下的代码行创建了一个NSString：

```
NSString *aString = [[NSString alloc] initWithString: @"Hi!"];
```

在创建了某些对象并使用它们之后，问题出现了。当用完它们时，如何返还它们呢？内存管理的黄金规则是，当用完所分配的内容时，应该将字节还给堆。一种可能是Objective-C提供了一个方法，当调用该方法时，只是释放一个对象所使用的字节：

```
[aString freeObject];
```

然而，这并非是一个令人满意的解决方案。它需要你在整个程序中记录使用了对象的每个位置，并且，当确定用完对象时，调用freeObject方法。对于一个较大的程序来说，这显然是不实际的。对于可能包含数百个对象的程序，这些对象用于数百个文件中，整个团队的程序员都使用这些文件，这种方法就不只是不实际那么简单了。

为了避免这一问题，Objective-C使用一种叫做引用计数的系统。引用计数当前是默认的内存管理系统。如果想要使用垃圾收集，则必须在你的项目中显式地支持它，如第15章所介绍的。

注意 *在Objective-C的世界中，引用计数、保留计数和管理内存都是指同一种内存管理系统。它们只是同一事物的不同名称。*

内存泄漏

当一个程序从堆分配内存，然后当程序完成的时候，没有将内存返回给堆，这时候就会发生内存泄漏。泄漏浪费了惊人的系统资源。泄漏的字节不能用于它们最初的目的，因为它们没有返还给堆，它们也不能用作其他用途。

内存泄漏导致一个程序的内存占用比应该的样子更大，这对性能很不利。如果丢失的只是零碎的少量字节的话，泄漏的影响可能无关紧要；但如果泄漏位于一个循环中或要执行一段较长的时期的程序中，则可能会耗尽系统内存并导致程序崩溃。

14.2 引用计数

引用计数的原理非常简单。每个对象存储了一个计数，叫做引用计数或保留计数，它是使用该对象的其他对象的数目。当一个对象的引用计数变为0的时候，销毁该对象，并且，它的字节都返回给堆以供可能的重用。使用引用计数的整个技巧就是，确保正确地保存了引用计数。这需要自己做一些工作。引用计数不是一个自动化系统。

当第一次使用alloc创建一个对象时，其引用计数为1。当不再需要该对象时，可以通过向对象发送一条release消息，表明用完了该对象：

```
[anObject release];
```

release消息将对象的引用计数减去1。如果新的引用计数为0，则会销毁对象，并且其内

存返还给堆。要点值得重复一次：一条release消息不会销毁接收者，它只是将引用计数减去1。下一节我们会看到，有的情况下，一个对象的引用计数大于1。只有当release消息导致引用计数变为0的时候，对象才会回收到堆。

如下代码展示了最简单的引用计数模式：

```
1 -(void) simpleMethod
2 {
3   NSMutableArray* anArray = [[NSMutableArray alloc] init];
4
5   // Code that uses anArray goes here
6
7  // Finished with anArray
8  [anArray release];
9  anArray =  nil;
10
11
12  ...
13
14 }
```

现在，让我们来详细地看看代码：

- 第3行：用alloc创建了anArray（引用计数是1）。创建对象的方法就是名称以“alloc”或“new”开头，或者名称中包含单词“copy”的方法。
- 第5行：anArray用来做某事（引用计数仍然是1）。
- 第8行：该方法用完anArray。它发送给anArray一条release消息。anArray的引用计数从1变为0，这导致销毁anArray。
- 第9行：记住保存了指向对象的指针的对象变量。当anArray指向的对象释放的时候，对象的字节返回给了堆。保存在变量anArray中的指针不再有效，堆自由地将这些字节用于其他的对象。这一行将anArray设置为nil，以防止意外地使用无效的指针以及潜在毁坏另一个对象。

可以在这里看到某些规则的端倪。将其看做是一份合约：如果你使用alloc创建了一个对象，你负责使用一条对应的release来匹配该alloc。如下的代码肯定会导致内存泄漏：

```
-(void) badlyCodedMethod
{
  NSMutableArray *anArray = [[NSMutableArray alloc] init];

  // Do stuff with anArray

  ...
  // Return without releasing anArray
  return;

}
```

变量anArray是一个自动变量。当方法退出时，该变量就消失了。该变量所指向的对象仍然拥有一个为1的引用计数，但是，没有什么保存指向它的指针。该对象不会释放，它仍然保留，什么也不做，但是占用了本来可以用于其他事情的内存，这种情况一直持续到程序退出。

14.3 接受对象

到目前为止，引用计数似乎只是“记住释放你已经分配的东西”的一个有趣的名称。当开始将对象从一个方法传递到另一个方法时，它真正的有用性开始显现出来了。

接受自另一个方法的对象，假设应该在接受方法的剩余部分中保持有效。在如下的示例中，NSDate的date方法返回了一个对象，表示当前的时间和日期：

```
- (void) doSomethingWithTheCurrentDate
{
  NSDate *currentDate = [NSDate date];

  // currentDate is used, but not stored in an instance variable
  ...
}
```

doSomethingWithTheCurrentDate方法可以使用currentDate，直到它返回，而不必担心currentDate会销毁。

注意 这一条件只在单线程程序中为真。在一个多线程程序中，有可能另一个线程在接受方法返回之前销毁所接受的对象。在多线程环境中，如果你想要让返回对象的方法满足这一条件，需要特别关照这些方法（参见本章稍后的14.8节）。

如果想要将一个接收的对象存储到一个实例变量中，并且，在接受方法返回后继续使用它，则必须采取一个积极的步骤，来确保对象保持有效并且不会销毁。为了说明这一点，假设就像在第6章所做的一样，要为一个摇滚模拟游戏创建某些类。一个RockStar对象需要一个Guitar对象：

```
@Class Guitar;

@interface RockStar : NSObject
{
  Guitar* guitar;
  ...
}
  -(Guitar*) guitar;
  -(void) setGuitar:(Guitar*) newGuitar;

  ...

@end
```

当一个游戏控制器对象设置了该游戏，它分配一个RockStar实例和一个Guitar实例，

并且通过调用RockStar的setGuitar:方法，将Guitar实例传递给RockStar。编写setGuitar:方法的明显方式，有一个问题：

```
- (void) setGuitar:(Guitar *) newGuitar
{
  guitar = newGuitar;  // WRONG
}
```

游戏控制器使用一个alloc创建了Guitar。如果稍后，游戏控制器使用一条release来对应其alloc，将会发生什么情况？Guitar对象的引用计数将变为零，并且会销毁它。RockStar继续保存指向一个无效的对象的指针。为了防止这种情况发生，RockStar应该给向guitar发送一条retain消息：

```
[guitar retain];
```

retain消息是release消息的对应。它将接收者的引用计数增加1。当RockStar向guitar发送一条retain消息时，该消息将guitar中保存的对象的引用计数增加到了2。这防止了该对象被销毁掉，即便最初分配它的代码向其发送了一条release消息。

程序清单14.1 给出了setGuitar:方法的最终形式。

程序清单14.1 setGuitar:

```
1 - (void) setGuitar:(Guitar*) newGuitar
2 {
3   if ( newGuitar != guitar )
4     {
5       [guitar release];
6       guitar = newGuitar;
7       [guitar retain];
8     }
9 }
```

setGuitar:方法必须考虑到guitar已经包含了一个对象的可能性（可能RockStar换了吉他，因为最初的那个已经在舞台上砸坏了）。当前保留在guitar中的Guitar实例，在存储的时候接收一条retain消息；现在，当替换它时，它必须接收一条release消息。

在前面的代码中：

- 第5行：当前保存在guitar中的对象释放了。如果guitar当前为nil，则这一行什么也不做。
- 第6行：新的对象存储到guitar中。
- 第7行：向新的Guitar实例发送一条retain消息。一条retain消息将其接收者作为返回值。第6行和第7行可以合并编写如下：

```
guitar = [newGuitar retain];
```

- 第3行：如果guitar和newGuitar都指向同一个对象，并且第5行和第7行执行了，那么它们将向该对象连续发送一条release消息和一条retain消息。release消息可能导致销毁该对象。retain消息随后将拥有一个无效的接收者，并且程序将会崩溃。第3

行的if语句防止了这种情况的发生。

注意 前面的代码基本上是当编译器合成property语句时所产生的：

```
@property (nonatomic, retain) Guitar* guitar;
```

在这里将其写出来加以说明，在真实的程序中，使用@property和@synthesize会更整齐。参见第12章。

14.4 所有权

关于Objective-C引用计数，经常讨论所谓的所有权：

- 如果你创建了一个对象，你拥有它。
- 通过向其他人所创建的一个对象发送一条retain消息，你就获得该对象的所有权。
- 当你用完自己所拥有的一个对象时，必须通过向其发送一条release消息，来放弃对其的所有权。
- 当一个对象不再有所有者时（这意味着它的引用计数是0），销毁该对象。

注意 Objective-C所有权不是独占的所有权。一个对象可以有多个所有者。通过向一个对象发送一条retain消息，获得该对象的所有权，这不会取消其他人对该对象的所有权。

对于返回对象的方法，所有权遵守一个命名惯例：

- 如果你通过调用名称以“alloc”或“new”开头的方法或者名称中包含单词“copy”的方法来创建一个对象，你将会拥有返回的对象。
- 如果你从另一个方法接受了一个对象，则你并不会拥有返回的对象。如果你想要在接受该对象的方法的作用域之外使用该对象（通过将其存储到一个实例变量或一个全局变量中），你必须获得该对象的所有权。

通过复制获得所有权

可以通过复制对象而不是保留它，来获得对象的所有权。

注意 在试图复制一个对象之前，必须确保对象的类支持复制。一个能够复制的类，必须通过实现copyWithZone:采用NSCopying协议或通过实现mutableCopyWithZone:采用NSMutableCopying协议。

严格地讲，当通过复制获得了所有权，你拥有所有权的是新创建的副本，而不是最初的对象。

应该使用复制还是保留来获取所有权？这取决于你的程序和对象的设计。如果对象是可变的，则有可能保留对象的其他某些事物能够修改它。如果是当前环境所需要的，只是对象现在所处的状态的一个快照，则应该复制它；否则，应该保留它。在执行时间和内存方面，复制比简单保留更昂贵，特别是，当所涉及的对象很复杂并且实现了一次深复制时。

注意 一些可变类将复制实现为一次额外的retain，并且直接返回接收者，而不是分配一个新的对象。对于这些类，复制和保留之间没有区别。

14.5 dealloc

当一条release消息将一个对象的引用计数减少到0时，就会销毁对象。但是，如果一个对象要销毁了，而其实例变量中存储了其他的对象，那么对于那些存储的对象，会发生什么情况呢？当销毁一个RockStar对象时，对于存储在其guitar实例变量中的Guitar对象，会发生什么？当存储Guitar对象时，它接收到一条retain消息（见程序清单14.1）。如果RockStar对象只是消失了，Guitar对象没有接收到一条对应的release消息，则它将会泄漏。

为了防止泄漏，在自己的实例变量中存储了其他对象的一个对象，必须在一个dealloc方法中释放被存储的对象。例如，RockStar使用自己的实现覆盖了NSObject的dealloc方法：

```
- (void) dealloc
{
  [guitar release];

  // Release any other objects held in instance variables.
  ...

  [super dealloc];
}
```

完整的销毁过程如图14-1所示。

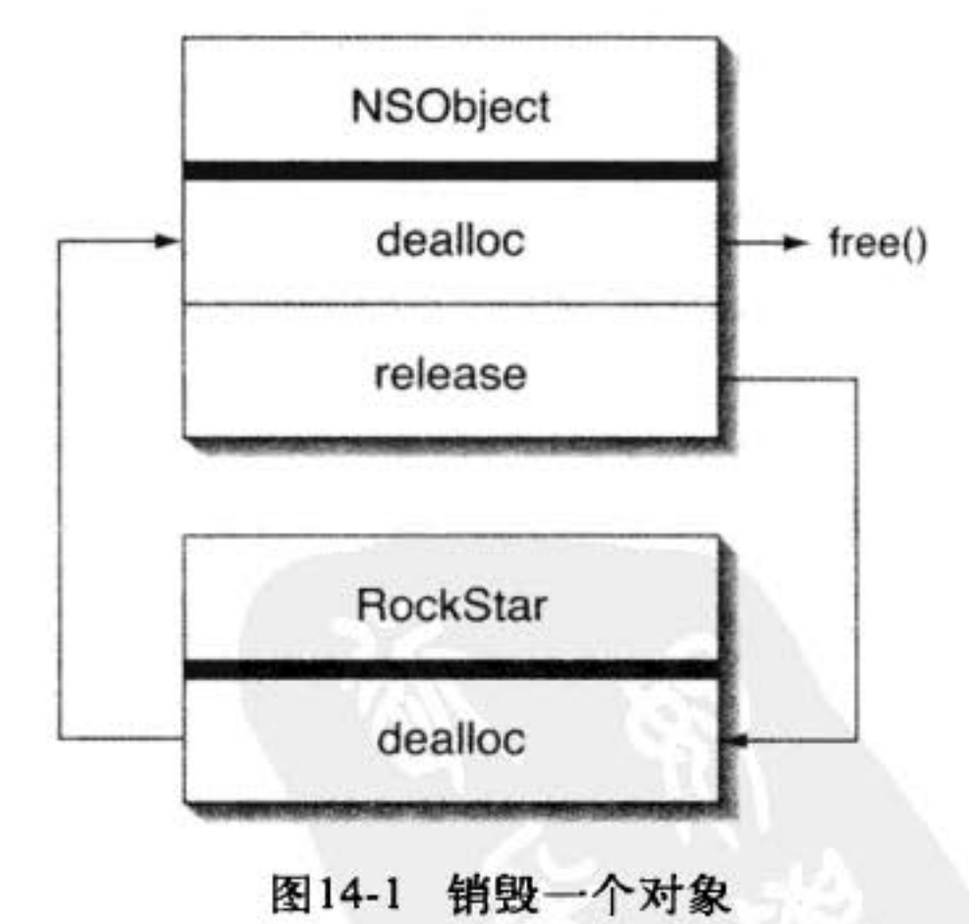


图14-1 销毁一个对象

1) release在NSObject中实现。如果一条release消息导致一个为0的引用计数，release就调用dealloc。

2) 由于RockStar覆盖了dealloc，因此执行的是dealloc的RockStar的实现。

3) RockStar向guitar发送一条release消息，以对应当存储guitar时它所接受到的一条retain消息。如果没有其他的对象拥有guitar中存储的对象，则它也会销毁掉。RockStar还向在其实例变量中存储的任何其他对象发送了release消息。

4) RockStar的dealloc方法使用super来调用NSObject的dealloc实现。

5) NSObject的dealloc实现调用系统库函数free，它执行将RockStar的字节返回给堆的工作。

[super dealloc]一行很重要。如果忘记了这一行，NSObject的dealloc实现将不会执行，对象的字节就会泄漏。将其作为dealloc方法的最后一行，这一点也很重要。当[super dealloc]返回时，对象已经销毁，并且不再有效。

注意 不应该直接调用dealloc。当一个对象的引用计数变为0时，它会被自动调用。

14.6 返回对象

还有一种情况，就是引用计数没有按你想要的那样顺利地工作。假设Guitar拥有一个类方法，它返回Guitar的实例：

```
@interface Guitar : NSObject
{
  // Instance variables
}

+ (Guitar*) guitar;

  // Other methods

@end
```

这类方法叫做简便构造函数。简便构造函数在Cocoa框架中很常见。你首先的冲动可能是像下面这样编写guitar方法：

```
+ (Guitar*) guitar
{
  Guitar *newGuitar = [[Guitar alloc] init];
  return newGuitar;
}
```

上面的代码创建并返回一个正确初始化的Guitar实例，但是，它有一个内存管理问题。它使用alloc创建了一个Guitar对象，但是，没有使用一条release来对应该alloc。实际上，guitar方法泄漏了新分配的对象。

注意 你可能认为，调用guitar方法并接受返回对象的代码应该负责释放该对象。但是，实际情况并不遵守这条规则。guitar不是以“alloc”或“new”开始的，或者并不包含“copy”。因此，调用者并没有拥有返回的对象。如果调用代码通过给返回对象发送一条retain消息，而拥有了该对象，那么它将必须最后用一条release来对应retain。但是，在guitar方法中，该release并没有对应alloc。

尝试快速修补的一种方案也没有效果：

```
- (Guitar*) guitar
{
  Guitar *newGuitar = [[Guitar alloc] init];
  [newGuitar release]; // WRONG!
  return newGuitar;
}
```

上面的代码使用一条release对应了alloc，但是，它引发了一个更大的问题。该release将newGuitar的引用计数减少到了0，导致它立即销毁。返回的是指向现在并不存

在的对象的一个指针。当引用这个失效的指针时，会导致一次程序崩溃。

修正这个问题需要Objective-C的autorelease机制，第14.6.1节将介绍它。

14.6.1 autorelease

简便构造函数或类似的方法所面临的问题是，它创建了一个对象。但是，它没有办法记录该对象，并且没有办法在合适的时候向其发送一条release消息来对应该对象的创建。

Objective-C使用自动释放机制解决了这一问题。自动释放是将一个对象转交给一个信任的第三方以在将来的某个时间释放它的一种方式，这个第三方就是称为自动释放池的一个对象。称之为自动释放有点不太合适，实际上应该叫做延迟释放。

14.6.2 自动释放池

自动释放是使用Foundation类NSAutoreleasePool来实现的。要使用自动释放，创建NSAutoreleasePool的一个实例：

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

上面的这行代码分配了池并且通知运行时，该池是当前的自动释放池。当向一个对象发送一条autorelease消息时，该对象被放入到当前自动释放池中：

```
[anObject autorelease];  // Places anObject in the current pool
```

Autorelease方法将接收者作为其返回值返回。这允许一条autorelease消息嵌入到其他消息中，或者用作return语句的参数：

```
SomeClass *autoreleasedObject =
    [[[SomeClass] alloc] init] autorelease];
```

在上面的代码中，向[[SomeClass] alloc] init]所创建的对象发送一条autorelease消息。autorelease消息的返回值是最初的对象，然后，将其存储到变量autoreleased-Object中。

向一个对象发送一条autorelease消息，只是注册的一种形式。它向该池的对象列表添加一个对象，但是，它没有导致向该对象发送任何消息。执行了autorelease消息之后，接收者仍然完全存在并有效。

向一个autorelease池发送一条release消息，会销毁该池，并且向池中的每一个对象发送一条release消息：

```
[pool release];
```

关于自动释放的一些要点如下：

- 向你拥有的一个对象发送一条autorelease消息，会履行合同使你最终失去该对象的所有权。一条autorelease消息是一条release消息，只不过它在将来发生。
- 也可能向一个对象发送多条autorelease消息。当自动释放池释放时，这个对象将接受多条release消息，每次自动释放它的时候都接受到一条。
- 释放一个自动释放池不一定意味着池中的所有对象都销毁了。池中的每一个对象都被发

送一条release消息。如果该release消息导致一个对象的引用计数变为0，则销毁该对象。

14.6.3 管理自动释放池

运行时维护了自动释放池的一个栈。创建一个池，则会将其推入到这个栈中；释放一个池，则会将其从栈中弹出。接受一条autorelease消息的对象，放置到位于栈的顶部的池中。

注意 *不能保留自动释放池，只能创建和释放自动释放池。向一个自动释放池发送一条retain消息将会引发一个异常。*

在一个对象上调用autorelease，而栈上却没有池，这将是一个错误。运行时将会给出抱怨，因为缺乏自动释放池会导致内存泄漏，但是，程序将会继续执行。如下所示的消息会写入到控制台输出（如果是一个命令行应用程序，将会写入到Terminal窗口）：

```
__NSAutoreleaseNoPool(): Object 0x100001050 of class Guitar
        autoreleased with no pool in place - just leaking
```

如果编写一个GUI程序，AppKit或UIKit会为你创建自动释放池。GUI程序有一个事件循环，该事件循环等待某些用户交互（例如，鼠标单击或者按键），然后将交互信息打包为一个事件对象，并且，将事件对象分发给程序的其他部分。该循环随后返回到顶部并等待下一次交互。AppKit（或iPhone上的UIKit）在每次事件循环的最初，创建一个自动释放池（这会将该池压入到栈中），并且，在循环的末尾释放该池（这会将其从栈中弹出，并向池中的所有对象发送一条release消息）。图14-2显示了在事件循环的一次迭代中，创建一个对象然后自动释放它的生命周期。如果在到达事件循环的底部之前，任何其他对象保留了该对象，那么对象的生命将得以延续。

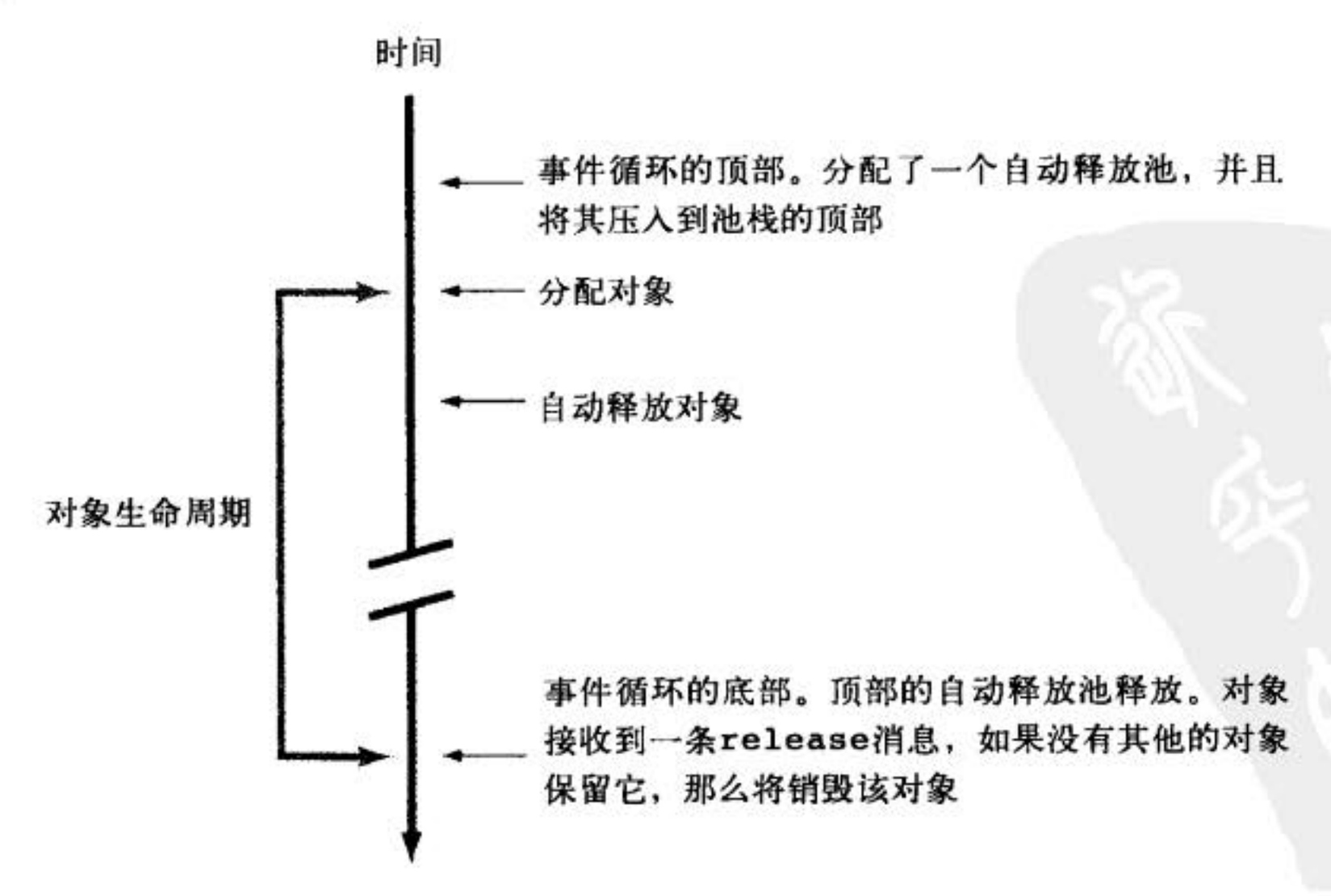


图14-2 一个GUI程序中的自动释放对象的生命周期

如果要编写一个Foundation（非GUI）程序，至少必须自己创建一个自动释放池：

```
main ()
{
  NSAutoreleasePool *pool  = [[NSAutoreleasePool alloc] init];
  ...
  // Whatever your program does
  ...
  [pool release];
  return 0;
}
```

注意 NSAutorelease类也提供了一个名为drain的方法。在针对引用计数而编译的代码中，drain等同于release。在针对垃圾收集而编译的代码中（见第15章），release消息什么也不做，而drain消息则充当垃圾收集器的一个提示，收集器将会考虑进行一次回收。只有在编写的同时针对引用计数和垃圾收集而编译的框架代码时，release和drain之间的区别才显著。

14.6.4 回顾简便构造函数

现在，有了设计不会导致内存管理问题的简便构造函数的方法，例如，如下代码展示了Guitar类的一个简便构造函数的正确形式：

```
+ (Guitar*) guitar
{
  Guitar *newGuitar = [[Guitar alloc] init];
  [newGuitar autorelease];
  return newGuitar;
}
```

autorelease消息对应了alloc消息，因此，该方法遵守内存管理规则。调用guitar方法的代码有一个机会来保留返回的Guitar对象，因为autorelease所隐含的release消息直到将来释放自动释放池时还没有发送。

14.6.5 自动释放和iPhone

如果编写一个iPhone应用程序，应该避免对较大的对象使用自动释放。驻留在自动释放池中的对象，根据程序的内存占用来计数，并且可能导致程序接受一个低内存警告的情况。应该直接使用alloc分配较大的对象并且一旦用完它们就释放它们，而不是使用一个类的简便构造函数（它通常会返回自动释放的对象）。

14.6.6 使用额外的自动释放池来控制内存使用

在有些情况下，创建自己的自动释放池是有用的。如果拥有一个方法，它创建并自动释放多个对象，并且，在一个循环中执行该方法，这些对象就会在顶级自动释放池中累积，无用地占用内存直到循环退出并且池释放掉。

```
int j;
id  myObject = ...

for ( j=0; j < MANY_TIMES; j++)
  {
    // Do something that creates and autoreleases many objects

    [myObject prolificAllocatorAndAutoreleaser];
  }
```

像这样的情况，可以通过在每次循环时提供一个新的池并且在循环底部释放它，从而减少程序的内存占用，如下所示：

```
int j;
id myObject = ...

for ( j=0; j <  MANY_TIMES; j++)
  {
    NSAutoreleasePool *pool  = [[NSAutoreleasePool alloc] init];

    // Do something that creates and autoreleases many objects

    [myObject prolificAllocatorAndAutoreleaser];

    // Release all the objects created during this iteration

    [pool release];
  }
```

保留者和被保留者彼此不认识

有一个细微而经常被误解的情况，即当对象A保留对象B时，两个对象在任何有意义的层面都是互不相识的。一个对象所知道的，只是有多少个对象在使用它（也就是说，其引用计数）。对象没有列表，来列出指向保留它的对象的指针。类似地，对象也没有一个列表，来列出指向它所保留的对象的指针。Objective-C语言或运行时中也没有什么方法，能够做类似这样的事情，对一个将要销毁的对象说："嗨，你！你保留了下面这些对象的引用。在你离开之前，释放它们。"

作为程序员，当你编写代码时，必须记录自己的alloc、copy和retain，并且使用一个release或autorelease来对应它们。

14.7 retainCount

NSObject实现了NSObject方法，它返回一个对象的引用计数的当前值：

```
NSUInteger currentRetainCount = [anObject retainCount];
```

乍一看，似乎retainCount对于调试内存管理问题很有用。然而，在使用Cocoa框架的实际

程序中，很多框架对象通过将对象传递到数组和字典中来保留和释放对象。结果是，一个对象所报告的引用计数，往往比你期望的通过查看自己的代码来记录的引用计数要高。此外，一些常量对象，例如常量NSString对象，可能报告没有意义的引用计数。实际上，retainCount方法几乎帮不上忙。

14.8 多线程

引用计数在多线程程序中变得更加复杂一些。例如，典型的setter模式如程序清单14.1所示，在多线程环境中，这是有问题的。如果两个线程试图在几乎相同的时间调用setGuitar:，则第二个线程可能会在第一个线程执行完该方法前中断它。这可能导致一个对象具有不正确的引用计数，或者，如果最初存储在guitar中的对象接受到两条release消息（每个线程一条消息），则将会导致程序崩溃。

要防止在使用setter方法时候线程之间彼此干扰，需要使用一个锁，如程序清单14.2所示。

程序清单14.2 带有锁的setGuitar:

```
-(void) setGuitar:(Guitar*)newGuitar
{
@synchronized(self)
  {
    if ( newGuitar != guitar )
      {
        [guitar release];
         guitar = newGuitar;
        [guitar retain];
      }
  }
}
```

@synchronized()编译器指令建立了一个锁机制，它一次只允许一个线程执行它后面的复合语句（代码块）中的代码。如果一个线程试图在另一个线程执行代码块时进入其中，则第二个线程将会被阻塞，直到第一个线程完成执行。在上面的代码中，@synchronized语句块保证了设置guitar是一个原子性的操作。

注意 @synchronized()的参数用来标识锁。它可以是任何Objective-C对象，在这里，self是一个方便的选择。

程序清单14.2中的代码，等同于当编译器从如下这条property语句合成一个setter时所创建的代码：

```
@property (retain) Guitar *guitar;
```

在一个多线程程序中，getter也需要修改。如下的简单的getter可能在多线程程序中引发问题：

```
(Guitar*) guitar
```

```
{
  return guitar;
}
```

按照惯例，getter所返回的对象应该保持有效，直到调用getter的方法返回。在多线程程序中，另一个线程可能会在getter返回之后的任何时刻，导致guitar中保存的对象销毁。最初的线程甚至可能没有机会保留该对象。使用如下的模式重新编写getter，将会消除这一问题：

```
(Guitar*) guitar
{
  @synchronized(self)
    {
      return [[guitar retain] autorelease];
    }
}
```

向guitar发送一条retain消息，随后再发送一条autorelease消息，这对于该对象的引用计数没有影响。但是，这样将该对象放入调用线程的自动释放池中，会延迟release消息，使得调用代码有机会来保留该对象。@synchronized语句块对于确保另一个线程不会中断这一对消息是必需的。

注意 本节给出的代码使得设置和获取一个实例变量在多线程环境中变成原子性的。但是，访问器的原子性操作并不会使你的代码自身变成线程安全的。线程安全比原子性操作要求更多，它要求仔细分析和设计。参见http://developer. apple. com/mac/library/documentation/Cocoa/Conceptual/Multithreading/Introduction/Introduction.html。

14.9 何时引用计数不好

引用计数使用得好坏取决于你是否遵守规则，使用相应的release或autorelease消息来对应每次对象创建和每条retain消息。如果犯了错误，会发生什么情况？忘记释放一个对象将会导致内存泄漏。该销毁的对象而没有销毁，造成其内存不会再返回给堆。

过度释放的结果更加直接和可怕。如果你向一个对象发送了一条多余的release消息，或者向你所没有的一个对象发送了一条release消息，那么release消息将导致对象销毁。当程序发送下一条release消息时，它会向一个已销毁的对象发送一条消息。结果通常是程序崩溃。如果在分散为各个部分的代码中，发送一条正确的release消息和一条可疑的release消息，这类问题将很难发现。

NSZombie

记录过度释放问题的一种方法是，使用Foundation的NSZombie功能。当打开NSZombie时，销毁的对象不会返回给堆。相反，它们变成了一个NSZombie对象。NSZombie对象会显示发送给它们的任何消息。也可以在[NSZombie release]和[NSZombie autorelease]上设置一个调试断点。这样可以看到额外的release来自何处。

要使用NSZombie，将环境变量NSZombieEnabled设置为YES。完成这个操作最简单的方式是使用Xcode。在项目窗口的左面板中，双击可执行文件的名称（提示，它在Executables之下）。当可执行代码的窗口出现时，选择Arguments面板。使用Variables to be set in the environment:部分底部的加号(+)来进行设置。将新变量的名称设置为NSZombieEnabled，并且将其值设置为YES，如图14-3所示。

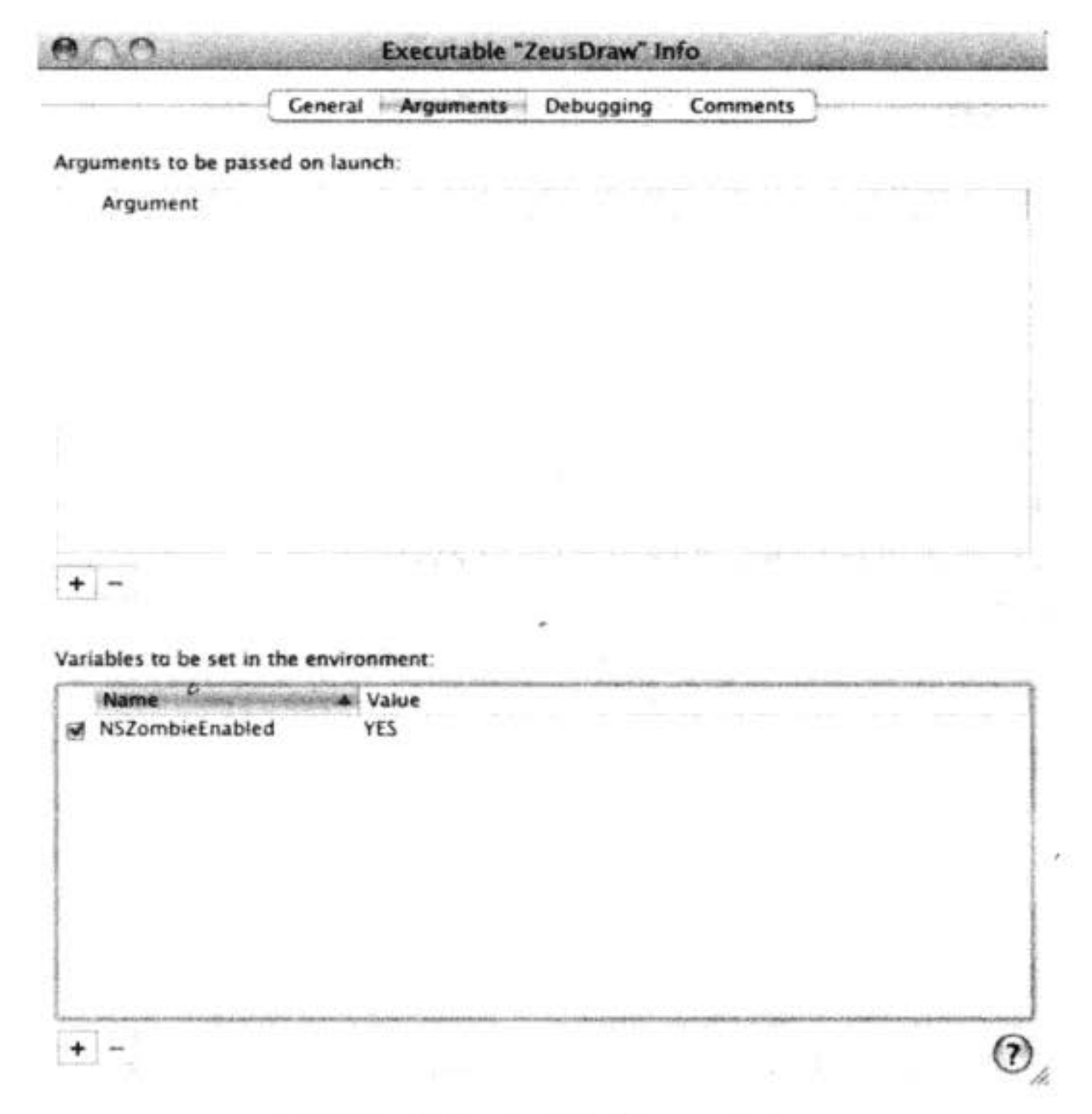


图14-3 在Xcode中使用NSZombie

注意 *在发布程序或将其推进到产品阶段之前，不要忘了删除NSZombieEnabled变量。*

14.10 保留循环

这是引用计数的另一个潜在问题。如果你创建两个或多个对象，且它们彼此交互引用，就会发生这一问题。假设摇滚游戏中有一个Band对象。RockStar有一个实例变量band保存了RockStar的Band，如程序清单14.3所示。

程序清单14.3 RockStar.h

```
@Class Band;

@interface RockStar : NSObject
```

```
{
  Band *band;
  ...
}
@property (nonatomic, retain) Band* band;

  ...

@end
```

RockStar在其dealloc方法中，释放了band中保存的对象，如程序清单14.4所示。

程序清单14.4 RockStar.m

```
@implementation RockStar

@synthesize band;
...

- (void) dealloc
{
  [band release];

  ...

  [super dealloc];
}

@end
```

类似地，Band对象也有一个RockStar变量leadSinger，如程序清单14.5所示。

程序清单14.5 Band.h

```
@Class RockStar;

@interface Band : NSObject
{
  RockStar *leadSinger;
  ...
}
@property (nonatomic, retain) RockStar* leadSinger;;

  ...

@end
```

Band也遵守这一规则。它在自己的dealloc方法中释放了leadSinger中保存的对象，如程序清单14.6所示。

程序清单14.6 Band.m

```
@implementation Band

@synthesize leadSinger;
...

- (void) dealloc
{
  [leadSinger release];

  ...

  [super dealloc];
}

@end
```

RockStar和Band都遵守内存管理规则，但是，还是有一个问题。一旦一个RockStar对象和一个Band对象彼此连接起来，就哪一个都销毁不了。即便没有其他的对象拥有它们中的任何一个，情况也是如此。RockStar的引用计数不会变为0，因为Band拥有它。Band也不会释放RockStar，因为RockStar对于Band的所有权阻止了调用Band的dealloc。当游戏控制器试图释放两个对象时，两个对象的内存都泄漏了。

这种遗憾的状态叫做保留循环。这只是保留循环的最简单的示例。你可以创建更为精巧的保留循环：对象A保留对象B，对象B保留对象C，等等，最后一个对象保留对象A。

解决这一问题的唯一方法是，修改代码，以便负责保留循环的对象中至少有一个不保留另一个对象。例如，对RockStar的接口部分的property语句作如下修改：

```
@property (nonatomic, assign) Band* band;
```

由于Band对象不再被保留，因此还必须从RockStar的dealloc方法中删除[band release]语句。这个修改打破了循环，现在两个对象可以销毁了。

一个对象保持对另一个对象的引用而不保留它，这称之为对该对象进行弱引用。在常规情况下，一个对象将自己的实例变量中保存的一个对象保留，这叫做强引用。使用弱引用编写对象，需要注意一些事项。当销毁一个Band对象时，它放弃了自己对leadSinger实例变量中保存的RockStar对象的所有权。如果在Band对象销毁之后，RockStar对象还有继续存在的任何可能性（例如，某个其他的对象也拥有RockStar），那么，应该修改Band对象的dealloc方法以将RockStar的band变量设置为nil，如下所示：

```
- (void) dealloc
{
  [leadSinger setBand: nil];
  [leadSinger release];

  ...
```

```
    [super dealloc];
}
```

将RockStar的band变量设置为nil，这防止了RockStar不小心使用指向一个已销毁对象的指针。

Objective-C没有自动的方法来检测保留循环。防止保留循环的唯一方法，就是仔细地编写代码。运用关于类的结构的知识，并且不要创建保留循环。在大多数情况下，一对对象中的一个被认为是主对象或父对象，而另一个被当做是次对象或子对象。主对象（在前面的例子中是Band）可以保留次对象（在前面的例子中是RockStar），但是，次对象不应该保留主对象。当主对象销毁时，应该释放次对象并且将次对象对于主对象的弱引用设置为nil。

很多Cocoa类接受代理对象，惯例是一个对象不保留其代理对象。

14.11 最后说再见：曲终人散时

当一个程序终止时，操作系统会收回该程序所使用的所有系统资源。支持你所使用或分配的任何内存的虚拟内存页面，都返回到操作系统的释放列表中。忘了释放一块内存的糟糕的结果，并不会在程序执行之外持久化。因此，当程序终止时，仍然存在的任何对象的dealloc方法永远不会再调用。无论如何，一切都消失了，没必要再展开一幅对象画卷并单独释放每个对象。但是，这里有一个陷阱。如果你养成了习惯在自己的dealloc方法中进行其他的各种清理，那么进行这些清理的代码可能不会调用。例如，假设你有一些代码，用来在程序终止时将某些状态信息写入到磁盘文件中。如果你将这些代码放入到某个对象的dealloc方法中，则这段代码将不会执行。

Mac OS X上的AppKit和iPhone上的UIKit提供了一种方法来处理这种情况。它们都提供一个对象（分别是NSApplication和UIApplication）来表示应用程序。在两种情况下，应用程序对象允许你注册一个定制的对象，定义并创建它，并将其作为应用程序对象的代理。当应用程序要终止时，应用程序对象向代理发送一条applicationWillTerminate:消息。你的代理对象通过提供applicationWillTerminate:的一个实现，其中包含了任何必要的清理代码，从而执行清理工作。

如果要编写一个非GUI Foundation程序，则可以直接在主程序的return语句之前放置任何清理代码。

一些程序员不愿意麻烦地为那些在程序执行的整个时间段都存在的对象使用一个dealloc方法。例如，在一个GUI程序中，用作检查器面板的那些对象。省略dealloc方法是糟糕的做法。如果你养成了习惯，当一个对象存储了其他对象时，总是提供一个dealloc方法，这样就不太可能在需要的情况下忘记使用它。并且，如果你在另一个程序的不同环境中，重用你的“永久性”对象之一的代码，而该环境中该对象不会持续程序的执行生命那么久，那么，正确的dealloc将会准备到位。

14.12 小结

Objective-C的引用计数内存管理系统有以下一些要点：

- 一个对象的引用计数，保存了使用该对象的其他对象的数目。引用计数通常表示为一个对象的所有者的数目。
- 创建一个对象，它的引用计数为1。
- 向对象发送一条retain消息，会将其引用计数增加1。
- 向对象发送一条release消息，会将其引用计数减少1。
- 一条autorelease消息没有直接效果，但是，它安排在未来的某个时刻向其接收者发送一条release消息。
- 当对象的引用计数为0时，销毁该对象。

引用计数是管理内存的一种有效方式，但是你必须遵守规则。

规则是：

- 使用名称以“alloc”、“new”开始的方法或方法名中包含“copy”的方法创建的任何对象，你都拥有它。
- 你不能拥有由任何其他方法返回给你的对象。返回的对象可能是自动释放的，也可能不是。你不必关心。相关的唯一事实是，你不能拥有该对象。
- 可以通过向对象发送一条retain消息来获取其所有权。
- 当你使用完你所拥有的一个对象时，必须通过向该对象发送一条release消息或一条autorelease消息来放弃所有权。如果你没有拥有一个对象，你不能够向其发送一条release消息或autorelease消息。

一些辅助规则：

- 如果你接受了来自一个方法的对象，而这个方法不是一个创建方法，并且你想要将该对象保存到一个实例变量中以供以后使用，那么你必须获得接受的对象的所有权（这条规则对于弱引用例外）。
- 如果你有一个对象，它对于存储在其实例变量中的任何对象有所有权，则你的对象必须实现一个dealloc方法，以便在销毁自己的时候释放那些对象。

14.13 练习

1. 编写一个较小的程序来创建一个对象，然后，向该对象发送一条retain消息，随后是两条release消息，一条对应创建，一条对应retain。
   - 添加语句，分别在创建之后、retain消息之后和第一条release消息之后，显示对象的引用计数（使用retainCount方法）。验证引用计数和你预料的一样。
   - 尝试并验证，在第二条release消息之后对象的引用计数是0，为什么这是个糟糕的想法？

   在这个示例中，retainCount方法用于说明的目的，但是，正如本章所讲到的，在调试一个真正的程序时，它没有多大帮助。
2. 本练习查看自动释放的顺序：
   - 定义一个类，它覆盖了release和dealloc的实现，只是当调用这些方法时给出显示。在两种情况下，NSLog语句都位于覆盖的实现的开始处，并且，在实现的底部调用一次超类的实现。

- 编写一个小程序，它分配了类的一个实例，并且随后自动释放它。将NSLog语句放在自动释放消息之前和之后以及释放自动释放池的消息之前和之后。这些显示语句只是告诉你，已经到达了程序中的一个特定位置。例如：

```
NSLog( @"Just before the autorelease." );
```

- 运行该程序。确保你理解并且能够说明其输出。

3. 创建一个保留循环：
   - 实现RockStar类和Band类的框架版本，如程序清单14.3、程序清单14.4、程序清单14.5和程序清单14.6所示。
   - 给两个类的dealloc方法添加显示语句。
   - 编写一个程序来分配一个RockStar对象和一个Band对象。将Band的leadSinger实例变量设置为RockStar，而将RockStar的band实例变量设置为Band对象。
   - 发送release消息来对应两个对象的分配。验证两个对象都不会销毁。
   - 修改代码，使RockStar现在对其band进行一个弱引用。再次运行程序，验证两个对象现在都销毁了。
4. 如下两行代码都创建了一个空的可变的数组。二者之间有什么区别？

```
NSMutableArray *ar1 = [[NSMutableArray alloc] init];
NSMutableArray *ar2 = [NSMutableArray array];
```

内存管理的隐含意义是什么？（你的任何回答都不应该包括“自动释放”的字眼）。

第15章 垃圾收集

一个典型的运行中的Objective-C程序会持续不断地分配新的对象。这些对象中的大多数都只是临时需要的，在程序终止后，就不再需要保持它们有效了。如果程序不能返回不需要的对象所占用的字节，则程序的内存占用将持续增长并且其性能也会受到影响。第14章介绍了引用计数内存，这是管理内存的一种手动系统。本章将介绍垃圾收集，这是管理内存的一种自动系统，它是从Mac OS X 雪豹（v 10.5）开始添加到Objective-C中的。垃圾收集在iOS上还不可用。

当使用垃圾收集进行内存管理时，有一个叫做垃圾收集器的单独的线程会定期运行。垃圾收集器检查程序的内存，确定哪些对象在使用中，并且销毁任何不再使用的对象。

本章介绍垃圾收集的理论，Objective-C垃圾收集器如何工作，以及如何在一个Objective-C程序中使用垃圾收集器，并介绍Objective-C的垃圾收集器当前实现中的一些问题。

15.1 垃圾收集：理论

垃圾收集器的任务就是通过一个运行程序来找到不再使用的对象并销毁它们，并将它们的字节返还给堆，以便这些字节可以重复使用。

垃圾收集在原理上很简单。任何已知在使用中的对象，或者从一个已知在使用的对象沿着一条引用链（指针）可以达到的任何对象，我们都认为它是“活着的”。所有其他的对象都被认为是“垃圾”并要对其进行收集。垃圾对象所使用的内存返还给堆。

垃圾收集从叫做根集合的一个对象初始集合开始。根集合中的对象，是那些知道在使用中的对象，因为它们都保存在堆栈上的变量中或外部变量中。收集器随后进行一次膝盖骨连接到大腿骨式的分析。它标记根集合为活的，然后，将根集合引用的任何对象都标记为活的。它继续按照引用进行标记，直到它标记了从根集合出发按照一个引用链所能标记的所有对象。最后，它清扫内存并收回那些没有标记为活的对象。

图15-1给出了一个假想的程序内存中的一组对象在垃圾收集刚开始时的样子。从一个对象到另一个对象的箭头，表示第一个对象保存了对第二个对象的一个引用（指针）。对象A和对象B保存在栈上的变量中，对象C保存在一个外部变量中。它们构成了这个例子的根集合。对象D、E、F、I和J都是活的，因为其中的每一个都可以沿着从根集合对象出发的一条引用链而达到。对象G、H、K和L是垃圾，并且将会被回收。注意，对象K是垃圾，即便对象H保留了一个指向它的指针。因为二者都不能从根集合达到。

垃圾收集解决了保留循环的问题。正如在第14章看到的，在引用计数的内存管理之下，有可能会有两个或多个对象彼此保留，按这样的方式，它们的引用计数都不会达到0，即便循环外没有对象保留它们。如图15-2所示，在垃圾收集下，循环不是一个问题。

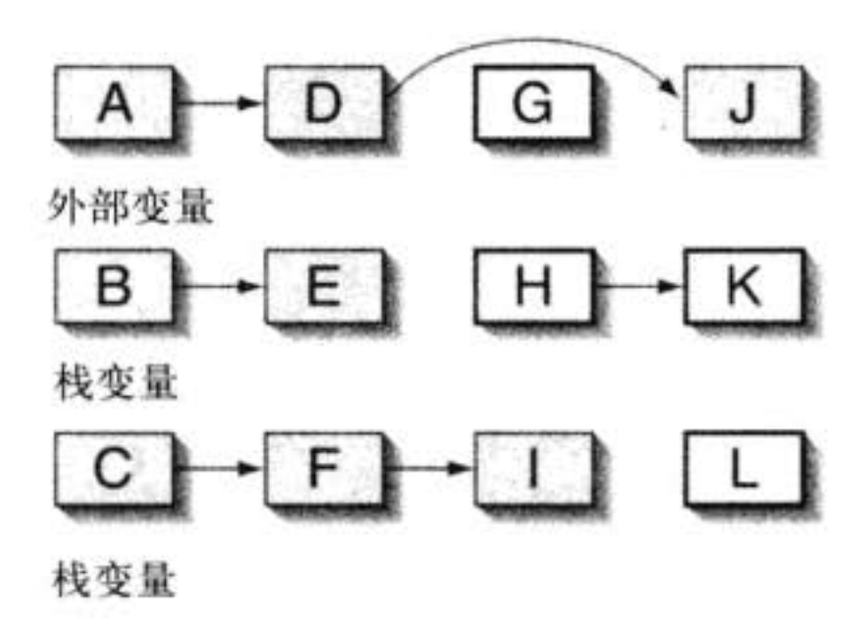


图15-1 可从根集合达到的对象

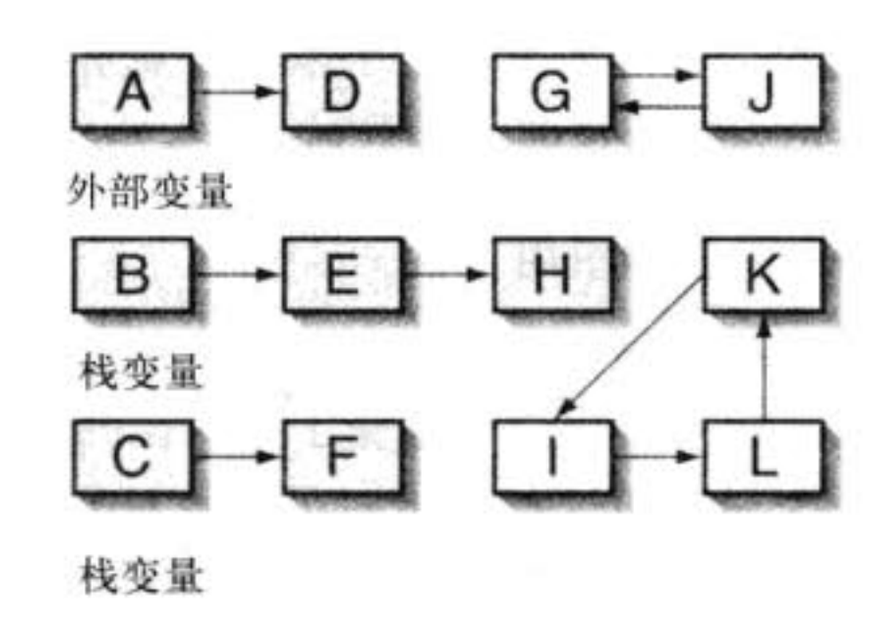


图15-2 保留循环

对象G和J相互引用对方。对象I、K和L在3个对象的一个循环中彼此引用。然而，所有5个对象都标记为垃圾并回收，因为它们中没有一个是从根集合中的对象可达到的。

15.2 垃圾收集：实践

Objective-C的垃圾收集器的实际实现要比15.1节所介绍的简单标记和清理垃圾收集器更复杂。它是分代式（generational）垃圾收集器的一个示例。

在大多数程序中，分配的对象要么是临时的，要么持续较长一段时间。分代式垃圾收集器猜测大多数最新分配的内存是最可能成为垃圾的内存，并且，首先检查这些内存。

为此，垃圾收集器根据年龄将分配进行分代。当收集器从一代开始时，它在新的一代和旧的一代之间放置一个写屏障（write barrier），如图15-3所示。

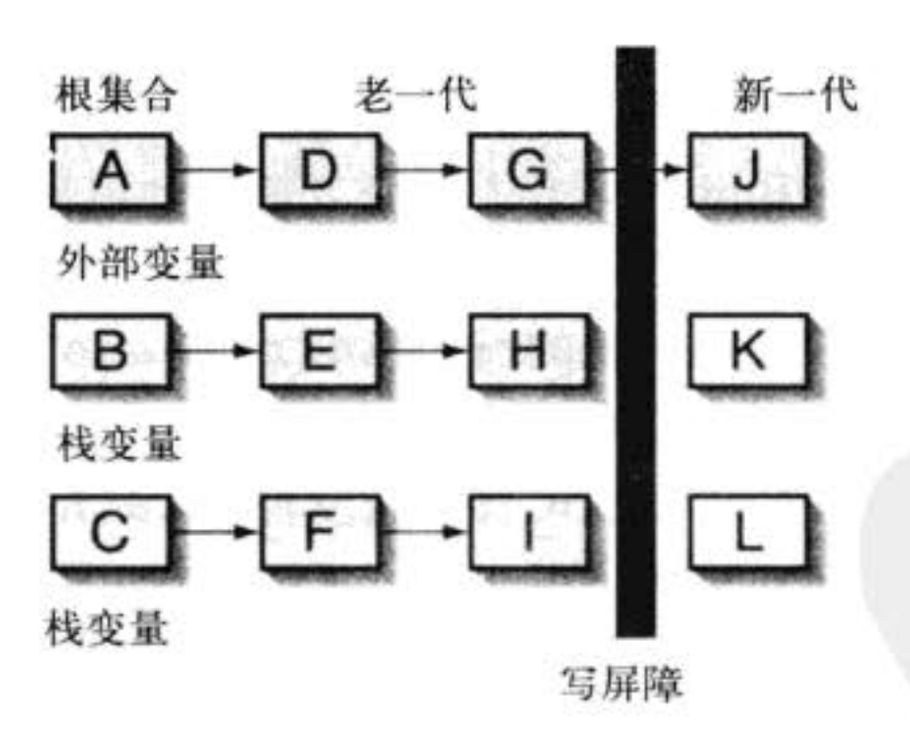


图15-3 收集器分代

当将指向新一代中的一个对象的指针，赋值给老一代中的一个对象的实例变量时，写屏障会注意到这一点，并且通过标记老一代的对象来记录下这次赋值。在图15-3中，当将指向对象J的一个指针存储到对象G中时，写屏障会标记对象G。收集器随后可以使用标记的对象作为一个临时性的根集合，并且跟踪引用到新一代，来进行一次部分的收集。当部分收集完成后，新一代中仍然活着的对象将进入到老一代中，并且，再次从一个空的新一代开始这一过程。与必须扫描收集器所控制的所有内存的完全收集相比，部分收集要快很多。

注意 “写屏障”这个术语可能有点令人误解。写屏障并不是阻止你进行赋值，它只是记录下通过它的指针赋值。

强引用和弱引用

第14章介绍了强引用和弱引用的概念。在引用计数之下，一个对象通过保留另一个对象，形成对其的强引用。这确保了被保留的对象保持活的状态，直到保留对象用完它并释放它。相反，一个对象通过存储指向另一个对象的指针而不保留它，从而形成对另一个对象的弱引用。弱引用用来防止保留循环，并用于缓存这样的模式，其中，如果被引用的对象消失了，引用的对象也不会有问题。

垃圾收集器也在强引用和弱引用之间做出了区分。收集器只跟踪强引用。在垃圾收集中，只有存储指向一个对象的指针，才构成该对象的一个强引用。如果一个对象是活的（可通过根集合达到），则它保存的指针所指向的任何对象也必须是活的。要创建一个弱引用（收集器将不会跟踪的引用），必须使用关键字__weak显式地标记该变量用于弱引用。编译器对这一语法比较随意，__weak关键字可以放在类型的前面或后面，也可以放在声明中*的前面或后面：

```
__weak SomeClass *anInstance;
SomeClass __weak *anInstance2;  // Equivalent
SomeClass * __weak anInstance3; // Also equivalent
```

当收集器收集到__weak变量中保存的一个对象时，它将该变量设置为nil，以防止你意外地使用指向一个已销毁的对象的指针。

15.3 使用垃圾收集器

使用Objective-C垃圾收集器很简单。可以通过向gcc传入相应的标志来使用垃圾收集功能。如果使用Xcode，则可以在Target/Build设置中打开该功能，如图15-4所示。

可能的标志有：

- 没有标志设置。表示代码将使用传统的retain/release引用计数编译并且不会连接或载入到使用垃圾收集的可执行环境中。
- -fobjc-gc-only。表示代码只有GC代码而没有retain/release。它不能连接或载入到使用传统引用计数的可执行环境中。
- -fobjc-gc。表示代码使用传统的retain/release和GC内存管理逻辑写成。支持两种内存管理模式的代码，主要用于编写那些可以用GC或非GC执行环境连接的框架。双模式代码很难正确编写，因此，除非你要编写这样的框架，否则应该避免使用它。当双模式代码载入到使用GC的程序中时，运行时会短路任何retain、release、autorelease、retainCount和dealloc消息，它们将不会发送给接收者。

尽管有可能让代码在GC或非GC环境下都工作，但任何给定的最终程序，必须要么在GC下要么在非GC下编译连接。不能让可执行文件二者都做到，然后在运行时再选择想要使用哪一种，也不能让一个可执行文件一部分是GC的而另一部分是非GC的。

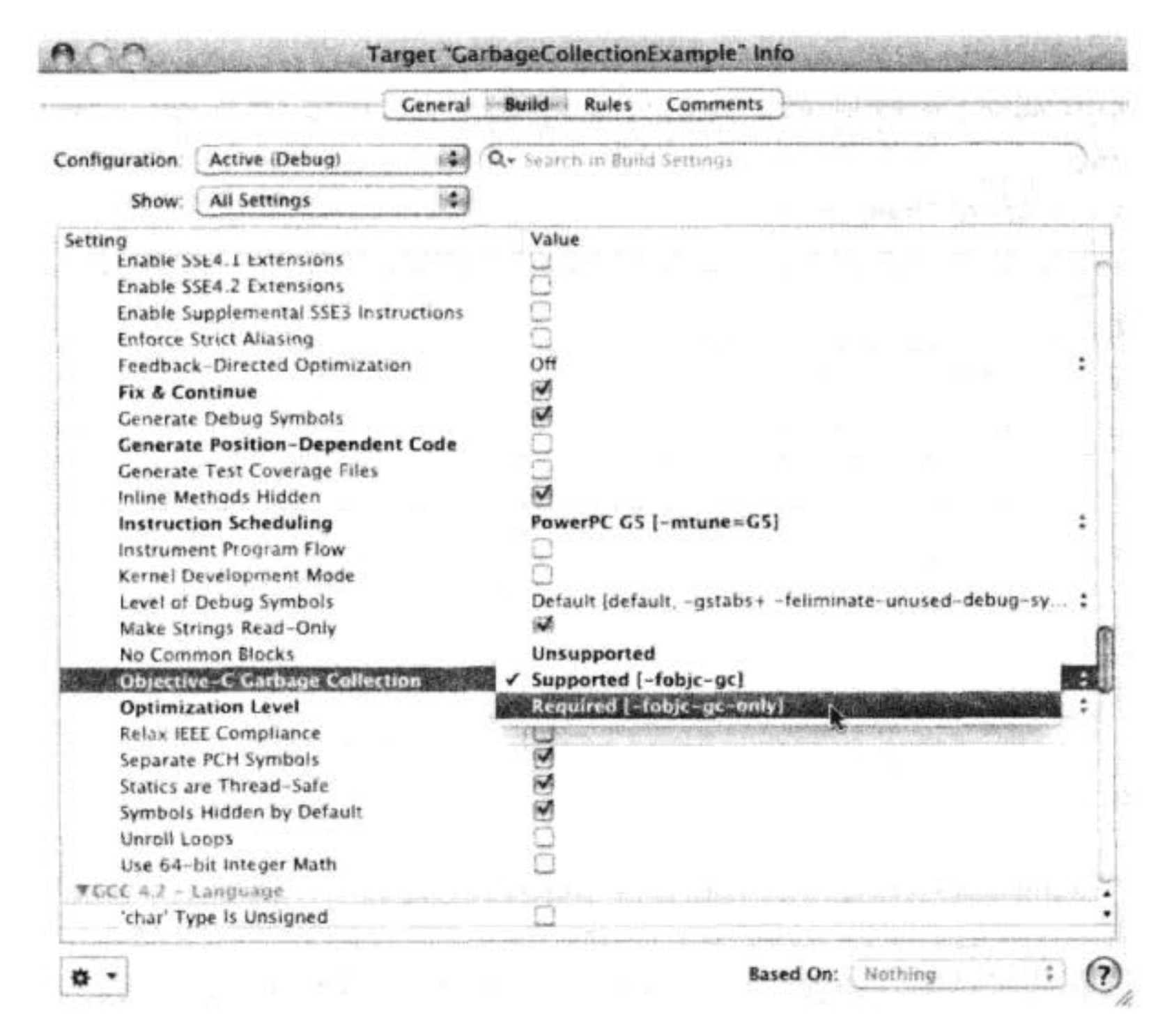


图15-4 在Xcode中启动垃圾收集

垃圾收集器在单独的线程上运行。如果要编写一个GUI程序，AppKit会为你启动垃圾收集器线程。如果要编写一个Foundation命令行程序，则必须通过导入头文件objc/objc-auto.h并且调用Foundation函数objc_startCollectorThread，自己启动该线程：

```
#import <Foundation/Foundation.h>
#import <objc/objc-auto.h>

int main ( void )
{
  objc_startCollectorThread();
  ...

  return 0;
}
```

进行收集时的控制

垃圾收集器是自动化的。它使用一个内部的算法来查看当前的内存使用，从而决定什么时候运行。然而，当你通过自己的程序逻辑知道有大量的垃圾对象存在时，可以要求垃圾收集器进行收集以帮助其更高效地工作。

可以通过单体Foundation类NSGarbageCollector来访问收集器，如下所示：

```
NSGarbageCollector *collector = [NSGarbageCollector defaultCollector];
```

如果必须创建大量的临时对象，而现在用完了这些对象，那么可以使用collectIfNeeded方法给收集器发送一条提示：

```
[collector collectIfNeeded];
```

如果你有大量的旧对象现在变成了垃圾，例如，在基于文档的应用程序中关闭一个文档，则可以要求收集器执行一次完全收集：

```
[collector collectExhaustively];
```

收集器线程必须临时停止那些使用要收集的内存的任何线程。如果你的代码有一个时间关键的部分，你不想让收集器中断它，则可以临时关闭收集器，并且随后再重新启动它：

```
[collector disable];
[collector enable];
```

确保在收集器关闭时不要分配过多数目的对象，并且，当退出代码的关键部分之后，记得要重新打开收集器。

15.4 终结器

设计为使用引用计数的类和设计为使用垃圾收集的类之间有一个显著的区别。在引用计数下，当对象的引用计数达到0时，会调用一个dealloc方法。dealloc方法通常会释放将要消失的对象所保留的任何对象。Objective-C程序员过去必须使用dealloc来执行在销毁一个对象之前所必须执行的任何其他清理工作。这包括释放其他资源，例如malloc的内存块和文件描述符，以及关闭网络连接等任务。在垃圾收集下，没有dealloc方法（如果一个带有dealloc方法的类用于一个GC程序中，该方法将不会被调用）。

> **注意** 不要把销毁和dealloc混淆了。销毁是将一个对象的字节返回给堆的过程。这在引用计数和垃圾收集下都会发生。dealloc是一个方法，只有在引用计数环境中，在一个对象将要销毁之前才会调用该方法。

在垃圾收集下，NSObject提供了一种替代方法，finalize，在要销毁一个对象之前会调用它。尽管finalize看上去和dealloc类似，但是，二者之间有一些显著区别：

- 在一个finalize方法中，不需要释放（或nil）在将要消失的对象的实例变量中所保存的对任何其他对象的引用。如果对另一个对象的唯一的引用位于将要消失的对象的一个实例变量中，那么，被引用的对象也会变得无法触及并会被收集。
- 调用dealloc方法的顺序是确定的，而调用finalize方法的顺序是不确定的。在一次收集循环的末尾，在任何对象销毁之前，收集器调用所有那些标记为要收集的对象的finalize方法。调用这一组finalize方法的顺序是不确定的。
- 将要被收集的一组对象，可以通过它们的终结器彼此发送消息。对象A的finalize方法可以给对象B发送消息，即便对象B的finalize方法已经执行过了也是如此。这考虑到可能会向一个处于无效或不一致状态的对象发送消息。应该避免在一个终结器中向其他对象发送消息。

然而，和dealloc一样，如果实现了一个finalize方法，则必须记住要调用finalize的超类实现：

```
[super finalize];
```

Apple的官方推荐是，尽可能避免使用终结器。它推荐可以实现一个单独的invalidate方法，在代码中合适的时机可以调用这一方法来清理文件描述符和网络连接这样的无用资源。然而，这个建议有一点不真诚，因为，在某些情况下，确定调用invalidate方法的合适时机的唯一方法，就是针对资源实现你自己的引用计数系统。如果面临这样的一种情况，你应该继续并且将清理代码放在一个finalize方法中。

15.5 malloc和垃圾收集

当使用垃圾收集时，堆划分为两个部分：自动区域和malloc区域。Objective-C对象都是从自动区域分配的，该区域由收集器管理。直接从malloc获得的内存，来自于malloc区域，它对收集器来说是不可见的。我们负责手动地管理从malloc区域获得的内存，如果从malloc分配一块内存，必须记得用完它的时候要释放它。

可以使用Foundation函数NSAllocateCollectable，来分配一块由收集器管理的字节：

```
void *__strong NSAllocateCollectable (
    NSUInteger size,
    NSUInteger options
    );
```

size是所请求的字节数目。options有两个允许的值：

- 如果options等于0，则NSAllocateCollectable返回的内存块由收集器管理，但是，收集器不会扫描该内存。这意味着，收集器不会在内存块中查找对象。该块自身由收集器管理，但是，块的内容对于收集器来说是不可见的。
- 如果options等于定义的常量NSScannedOption，那么，返回的内存块由收集器管理，此外，收集器会扫描内存块的内容以查找对象。

收集器将类型为指向非对象的C类型的指针的变量，看做是弱引用。为了防止NSAllocateCollectable返回的内存块被立即收集（因为没有对其的强引用），必须将指向内存块的指针存储到使用__strong关键字所声明的一个变量中：

```
char * __strong mybytes =
    (void *__strong) NSAllocateCollectable( numBytesRequested, 0 );
```

__strong关键字与__weak相对，收集器将声明为__strong的变量当做一个强引用对待。

注意 不需要对保存指向Objective-C对象的指针的变量来使用__strong。收集器默认将对象引用当做强引用对待。

15.6 Core Foundation对象和垃圾收集

Core Foundation对象可以从自动区域（垃圾收集）或从malloc区域（非垃圾收集）分

配。可以通过用来创建对象的CFxxxCreate函数的allocator参数的值，来指定使用哪一个区域。如果指定了NULL、kCFAllocatorDefault或kCFAllocatorSystemDefault，则该对象将来自于自动区域。如果指定了kCFAllocatorMallocZone或kCFAllocatorMalloc，则该对象将来自于malloc区域。

从malloc区域分配的Core Foundation对象与引用计数下的对象的行为方式相同，它们都从创建函数返回，并带有一个值为1的引用计数。当它们的引用计数减少到0时，它们将立即销毁。

注意 与Objective-C方法retain和release不同，Core Foundation对象函数CFRetain和CFRelease通常在垃圾收集环境下使用。

从自动区域分配的Core Foundation对象，其行为略微不同。即便该对象是由收集器管理的，它创建后仍然带有一个值为1的引用计数。引用计数大于0的对象，就像一个根对象一样。收集器不会收集它。当对象的引用计数减少到0时，不会立即销毁它；它只是变得符合收集条件。如果没有对该对象的强引用，下一次运行收集器的时候，就收集它。

如果想要让收集器从一开始就管理Core Foundation对象，可以使用函数CFMakeCollectable：

```
__strong CFMutableArrayRef myCFArray =
     CFMakeCollectable( CFArrayCreateMutable( kCFAllocatorDefault, ...
                        ));
```

CFMakeCollectable类似于CFRelease，但是，在减少引用计数前，它会先检查以确保对象在自动区域中分配，并且，它也会返回初始对象的一个引用。__strong关键字是必需的，因为像CFMutableArrayRef这样的Core Foundation类型，是普通C指针的typedef。收集器将保存Core Foundation类型的变量当做弱引用来对待，除非使用__strong说明它不是。

15.7 过程中的一些难点

在像C和Objective-C这样使用指针的语言中，改进一个垃圾收集器，将是非常困难的任务。Apple已经做了一项惊人的工作，但是，仍然有一些难点和极端情况可能会给你带来麻烦。

15.7.1 AppKit中的透明指针问题

AppKit中的几个方法以类型为void*的透明指针作为其参数之一。典型的示例是，用来绘制表单的方法，这些方法有一个用于didEnd选择器的参数，当删除表单时调用该选择器；还有一个环境对象，类型为void*，它作为一个参数传递给didEnd选择器。该环境对象通常是一个NSDictionary，用来将创建表单的对象的信息传递给处理用户输入的方法。

注意 表单是较小的临时性面板，它从窗口帧的顶部延伸下来。它们通常用于像文件选择这样的任务。

如下的代码给出了一个简单的示例：

```
@implementation SheetController

-(IBAction)showTheSheet:(id)sender
{
  NSDictionary* contextInfo = ...  // Info to pass to callback

  [NSApp beginSheet:sheetWindow
           modalForWindow: window
           modalDelegate: sheetControllerDelegate
           didEndSelector:
             @selector(sheetDidEnd:returnCode:contextInfo:);
           contextInfo:(void *) contextInfo];
}
@end


@implementation SheetControllerDelegate

- (void)sheetDidEnd:(NSWindow *)sheet
         returnCode:(int)returnCode
        contextInfo:(void *)contextObject
{
// Do something with contextObject
  ...
}
@end
```

当用户选择与该表单对应的菜单项时，会调用showTheSheet:方法。showTheSheet:调用NSApplication方法beginSheet:modalForWindow:modalDelegate:didEndSelector:contextInfo:，它会显示表单并返回。

当用户取消表单时，指向最初的环境对象的一个指针，用作contextInfo 参数的值，来调用openPanelDidEnd:returnCode:contextInfo:。图15-5给出了这一过程的先后顺序。

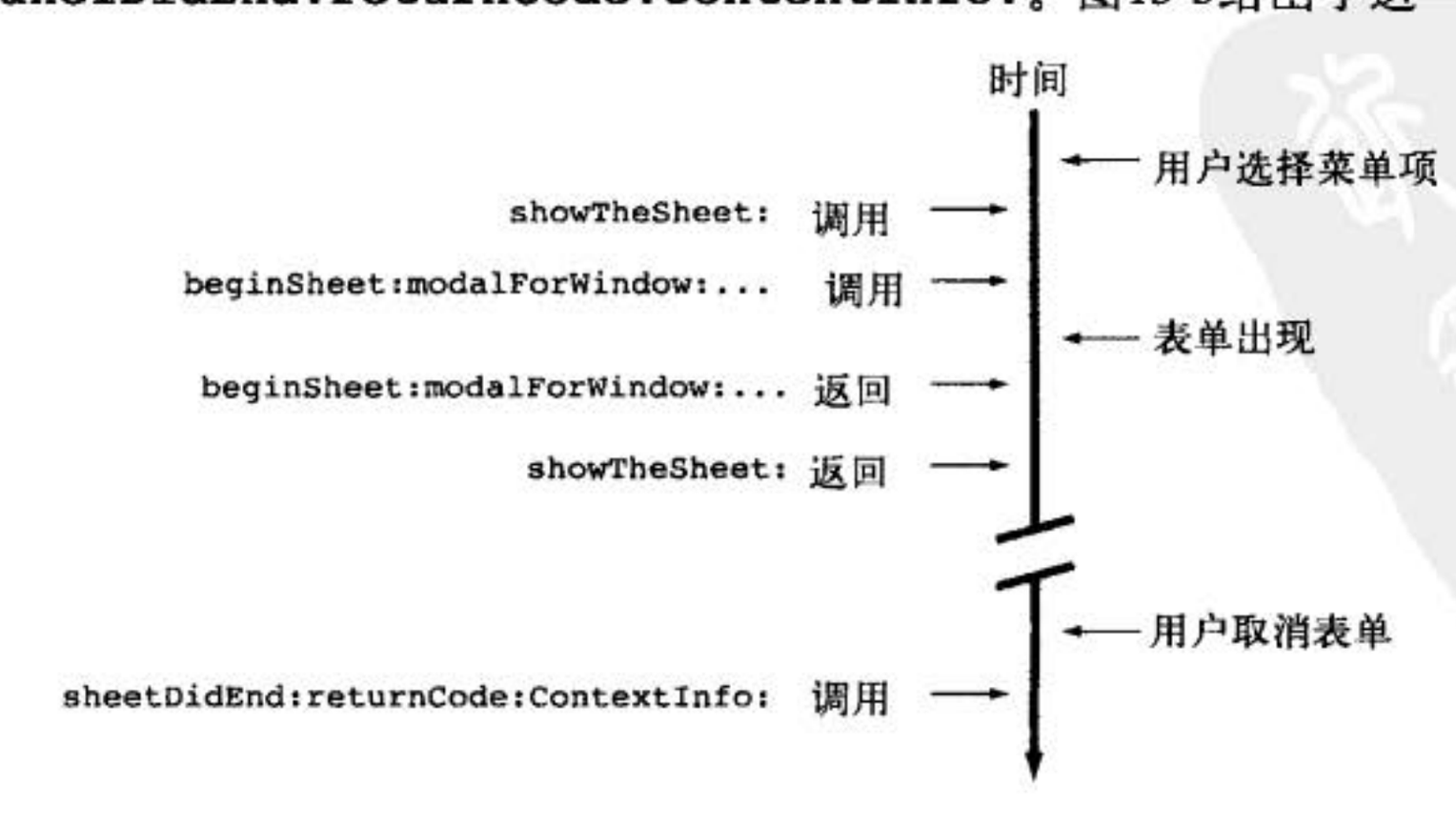


图15-5 显示和取消一个表单

问题是，AppKit将该环境对象存储为一个void*。收集器将void*当做一个弱引用对待，如果收集器线程在showTheSheet:返回的时刻和调用openPanelDidEnd:returnCode:contextInfo:的时刻之间运行，该环境对象可能会被标记为垃圾并被收集。这可能会给openPanelDidEnd:returnCode:contextInfo:留下一个无效的对象作为参数，并且可能会导致程序崩溃。

注意 崩溃不是每次必然发生的。前面的代码的行为，从用户的视角来看，可能是随机的。它可能会也可能不会崩溃，这取决于收集器线程运行的时刻。

解决方法略微有点令人惊讶。引用计数仍然对Objective-C对象有效。retain和release在垃圾收集下什么也不做，但是Core Foundation对象的CFRetain和CFRelease，在Objective-C对象上和Core Foundation对象上都有效。例如：

```
CFRetain(contextObject);
```

将contextObject的引用计数设置为1，并且阻止收集器收集contextObject，直到用完它为止。然后，通过调用CFRelease来对应CFRetain的调用，如下所示：

```
@implementation SheetController

-(IBAction)showTheSheet:(id)action
{
  NSDictionary* contextObject = ...  // Info to pass to callback
  CFRetain(contextObject);

  [NSApp beginSheet:sheetWindow
           modalForWindow: window
           modalDelegate: sheetControllerDelegate
           didEndSelector:
           @selector(sheetDidEnd:returnCode:contextInfo:);
         contextInfo:(void *) contextObject];
}
@end

@implementation SheetControllerDelegate

- (void)sheetDidEnd:(NSWindow *)sheet
         returnCode:(int)returnCode
         contextInfo:(void *)contextObject
{
  // Do something with the objects in contextObject
  ...
  // Finished with contextObject
  CFRelease(contextObject );

}
@end
```

15.7.2 内部指针

一些对象，如NSData、NSMutableData和NSBitmapImageRep（保存位图的AppKit对象），它们拥有返回内部指针的方法。这些是访问对象所保存的数据的普通C指针。如下的代码返回指向一个NSMutableData对象所包装的字节的指针：

```
NSMutableData *mutableData =  ...
char *bytes = [mutableData mutableBytes];
```

使用一个内部指针，可能会导致一种情况，即垃圾收集器对于通过编译器来优化性能的反映很差。考虑如下的代码：

```
NSMutableData *mutableData =  ...
char *mutableBytes = [mutableData mutableBytes];
NSUInteger length = [mutableData length];
while ( length )
 {
    // Do something with mutableBytes.
    length--;
 }
 ...
```

如果使用gcc这样的优化编译器来编译前面的代码，随后再执行它，则它最终会崩溃或损坏你的数据。乍看起来，这一点令人困惑。mutableBytes是从mutableData获取的一个有效的指针。mutableData保存在一个栈变量中，因此它应该是一个根对象并且不会被收集。然而，优化的编译器不遵从这一点，它注意到在下面这一行之后，mutableData不再被引用：

```
NSUInteger length = [mutableData length];
```

由于编译器认为不再需要mutableData（它不能理解mutableBytes和mutableData之间的关系），它偷取了mutableData在栈上的位置以用于另一个变量（在存储mutableData的栈地址上，节省出空间供编译器存储其他的变量）。如果收集器在while循环中唤醒并运行，那么它注意到，不再有对mutableData的强引用。收集器会收集mutableData，而仍然让mutableBytes指向已经收集了并且可能会重新分配给一个不同的对象的内存。

可以在while循环之后放置一条给mutableData的消息（任何消息），从而修正这一问题：

```
NSMutableData *mutableData =  ...
char *bytes = [mutableData mutableBytes];
NSUInteger length = [mutableData length];
while ( length )
 {
    // Do something with mutableBytes.
    length--;
 }
[mutableData self];
 ...
```

[mutableData self]并不实际地做任何事情，它返回指向mutableData的一个指针，随后会丢弃它。但是，这足以防止编译器将mutableData在栈上的位置让出，直到while循环结束。

15.7.3 错误的根对象

另一个奇怪之处是，由于栈的实现方式，收集器无法总是确切地断定栈在何处结束。为了避免泄漏，收集器宁滥勿失地给出警告，并且查找稍微超出栈的末尾的对象。如果从前面的一个函数调用中，收集器找到指向一个遗留在栈上的对象的指针，该对象会被误认为是根。收集器认为这是一个根对象，并且即便不再使用它，也不会收集它。如下的函数通过将栈的未使用的部分清零，从而修正该问题：

```
objc_clear_stack(OBJC_CLEAR_RESIDENT_STACK);
```

如果你要编写一个GUI程序，不必担心这一点。AppKit为你安排了定期的objc_clear_stack调用。然而，如果你要编写一个Foundation程序，就必须自己定期调用objc_clear_stack。

15.8 垃圾收集的优点和缺点

使用垃圾收集是一个选项，而不是必需的。下面介绍在决定是否在自己的应用程序中使用垃圾收集时，需要考虑的一些事情。

15.8.1 积极方面

垃圾收集最重要的优点是，它使得我们不必再关心对象的所有权规则（即使用正确数目的release或autorelease，来对应对象的创建和retain）。这要少写很多代码，发生内存泄漏或保存过时的指针所导致的问题的可能性也会大大减少，也不会有保留循环的问题。如果几个对象彼此保留引用，只要没有从一个根对象可达的对象，它们将仍然被视作垃圾并回收。

垃圾收集还简化了多线程代码。由于不需要retain或release，原子访问器成为简单的任务，而不需要使用锁。

15.8.2 消极方面

垃圾收集也有一些潜在的缺点。由于现在收集器来做你此前所做的工作（搞清楚需要保留哪个对象），因此使用垃圾收集的程序会使用更多的CPU周期。完全收集可能要对所有的对象内存分页，这使得分页行为比同等的非GC程序要糟糕很多。

如果你在一个紧凑的循环中分配较多的对象，分配可能会耗尽收集器，导致程序的内存占用增长得毫无必要的大，反过来影响到性能。在一个引用计数程序中，可以通过使用额外的自动释放池来手动调整内存管理，从而避免这种情况。

最后，收集器在一个单独的线程上运行。即便是一个最简单的程序，也会转变为一个多线程程序。尽管垃圾收集的大多数方面都是自动化的，但还是有一些区域需要手动编写代码。如果在编写这样的代码时犯了一个错误，那么最终的bug可能变得无法确定并且很难跟踪。

15.8.3 应该使用垃圾收集吗

这很难回答，需要考虑下面一些因素：

- Apple鼓励程序员对于新的Mac OS X程序使用垃圾收集。使用垃圾收集使得编写代码更

容易，并且，它避免了错误匹配的retain/release消息及保留循环这样的问题。

- 将已有的使用引用计数的程序转换为垃圾收集的程序，这可能是一个糟糕的想法。二者在设计模式上有区别（例如，dealloc对应finalize），这可能需要重新编写大量的代码，由此也会潜在地增加新的bug。
- 对于一致性、性能的可预料性很重要的某些类型的应用程序来说（例如，音频或类似的实时应用程序），这可能是个问题。在这些情况下，可能需要构建一个测试应用程序，并且在确定使用垃圾收集之前，验证垃圾收集没有影响到必须保证的性能。
- 垃圾收集是Objective-C的一个相对较新的添加，并且仍然在改进中。在编写本书的时候，在某些领域，仍然有一些人报告bug或奇怪现象。可以查看Cocoa邮件列表存档，以了解最新的进展。
- 还有些情况不能使用垃圾收集。如果你要修改使用了传统内存管理的程序，则不能使用垃圾收集，因为你不能在一个程序中将引用计数和垃圾收集混合使用。当然，如果你所工作的环境不支持垃圾收集，例如iOS，也不能使用它。

15.9 小结

垃圾收集将内存管理的过程自动化。垃圾收集自动地找到不再使用的对象并且销毁它们，而不是使用retain、release和autorelease来确定对象的生命期。垃圾收集易于使用，在大多数情况下，所需要做的只是向编译器传递-fobjc-gc-only标志。编写既使用垃圾收集又使用引用计数进行内存管理的代码是可能的，但是，很难正确地做到，并且只能对框架或插件这么做。尽管垃圾收集主要是自动的，但使用Core Foundation对象或malloe分配的内存，则需要额外的代码。还有一些需要程序员额外注意的特殊情况。

垃圾收集在Mac OS X中可用，但是，在iOS中不可用。Apple没有对此发表评论，很可能是考虑到性能的原因。

15.10 练习

本章的练习要使用程序清单15.1～程序清单15.5。

程序清单15.1 ralph.h

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

@interface Ralph : NSObject
{
   Norton *norton;
}

@end
```

程序清单15.2 ralph.m

```
#import "Ralph.h"

@implementation Ralph
- (id) init
{
  if ( self = [super init])
    {
      norton = [[Norton alloc] init];
    }
  return self;
}

- (void)finalize
{
  NSLog( @"Ralph being collected");
  [super finalize];
}

@end
```

程序清单15.3 norton.h

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

@interface Norton : NSObject
{
}

@end
```

程序清单15.4 norton.m

```
#import "Norton.h"

@implementation Norton

- (void)finalize
{
  NSLog( @"Norton being collected");
  [super finalize];
}

@end
```

程序清单15.5 CGExercise.m

```
#import <Foundation/Foundation.h>
#include <objc/objc-auto.h>
#import "Ralph.h"
int main (void)
{
  // Start the collector thread
  objc_startCollectorThread();

  Ralph *ralph = [[Ralph alloc] init];

  // Force a collection
  [[NSGarbageCollector defaultCollector] collectExhaustively];

  // Make sure logging finishes before the program exits
  sleep( 1 );
  return 0;
}
```

程序清单15.1、程序清单15.2、程序清单15.3和程序清单15.4定义了一对较小的类，Ralph和Norton。Ralph实例在一个实例变量中保存一个Norton实例，两个类都显示出其finalize方法。它们没有其他的行为。

程序清单15.5是一个简单的程序，它使用Ralph和Norton来展示垃圾收集的某些方面。如下一行：

```
[NSGarbageCollector defaultCollector] collectExhaustively];
```

是必需的，因为如果没有它的话，垃圾收集器无法在如此简短的一个程序中运行。sleep语句也是使用一个非常简短的程序来展示垃圾收集要用到的工具。要在程序退出之前让编译器有机会运行，它是必需的。

在做下面的练习时，不要忘了打开垃圾收集，如程序清单15.4所示。

1. 使用程序清单15.1～程序清单15.5编译并运行程序。验证Ralph的实例没有被收集。为什么没有收集？（提示：主程序的本地变量在栈上分配）。
2. 在分配ralph的那行代码后面，插入如下一行：

   ```
   ralph=nil;
   ```

   重新编译并运行该程序。发生了什么情况？为什么？
3. 在

   ```
   ralph=nil;
   ```

   行的前面，插入如下一行：

   ```
   [ralph retain];
   ```

   重新编译并运行该程序。发生了什么情况？为什么？
4. 将你在前面的练习中插入的行替换为：

```
CFRetain( ralph );
```

重新编译并运行该程序。发生了什么情况？为什么？

5. 直接在

```
CFRetain( ralph );
```

行的后面，插入如下一行：

```
CFRelease( Ralph );
```

重新编译并运行该程序。发生了什么情况？为什么？

6. 从程序中删除如下的行：

```
CFRetain( ralph );

CFRelease( ralph );
ralph = nil;
```

在程序清单15.1的norton实例变量的声明中，添加__weak关键字。重新编译并运行该程序。发生了什么情况？为什么？

第16章 块

块提供了一种方法，来将一些可执行代码和一个环境（各种变量）包装到一个单个的实体中，以便它们可以稍后或在一个不同的线程上执行。在其他的语言中，块或类似的结构有时候叫做闭包或匿名函数。块是Apple提供的一项对C、Objective-C 2.0和C++的扩展。Apple已经将块当做一项提案提交给了C标准工作组。在编写本书的时候，块只能在Mac OS X Snow Leopard (v 10.6和iOS 4)上使用。它们不能在Mac OS X更早的版本或iPhone iOS上使用。

注意 如果你安装了Plausible Blocks (PLBlocks)的话，可以在iPhone OS 3.x和Mac OS X Leopard (v 10.5)上使用块。Plausible Blocks是源自于Apple发布的开源Darwin OS代码的一个反向工程端口，它提供了使用块所需的编译器和运行时。可以从http://code.google.com/p/plblocks/获取Plausible Blocks。

传递一包内容在很多情况下很有用，但是，支持采用块的主要驱动力之一是，Apple的新的Grand Central Dispatch (GCD)功能。GCD设计用来使得程序更容易并行，并且执行起来更高效。实际上，GCD是操作系统为我们管理的一个线程池。GCD背后的思想是，操作系统对于Mac上运行的所有进程有一个全局的视角，并且，当需要让程序更高效地运行时，它会分配资源（CPU、GPU和RAM）。对于要使用的线程的数目，以及规划它们在何时执行，GCD比一个用户空间程序能够做出更好的决定。可以使用块提交要执行的GCD工作单元。

注意 GCD提供了一个C接口以提交块。Cocoa通过NSOperationQueue、NSBLockOperation、NSInvocationOperation类为GCD提供了一个高层级的接口。

NSInvocationOperation允许将工作单元作为NSInvocation对象而不是块提交，但是，在16.3节我们将看到，NSInvocation对象多少有点难以设置。块要更容易使用。

本章是对块的简单介绍。我们将学习如何定义一个块，一个块如何访问其包围环境中的变量，如何在自己的代码中使用一个块，以及针对块的内存管理的技术话题。本章还将介绍使用块的时候要小心的一些陷阱。

在详细介绍块之前，先花点时间介绍两种早期的包装功能的方式：函数指针和Foundation类NSInvocation。

16.1 函数指针

当编译器遇到一个函数调用时，它在执行该函数的代码中插入一条跳转指令（跳转指令会

使程序执行跳转到特定的代码，而不是执行紧跟着跳转指令后面的代码行）。为了返回，函数执行一条跳转指令，回到紧跟着最初的函数调用后面的代码行。在一般的函数调用中，跳转指令的着陆点（及相应的调用的函数）是静态的，在编译时确定。但是，可以通过使用一个函数指针，使一个函数调用变为动态的。如下的代码行将myFunctionPtr声明为一个函数指针，它指向一个接受两个int参数并返回一个int的函数：

```
int (*myFunctionPtr) (int, int);
```

图16-1给出了对函数指针的剖析。

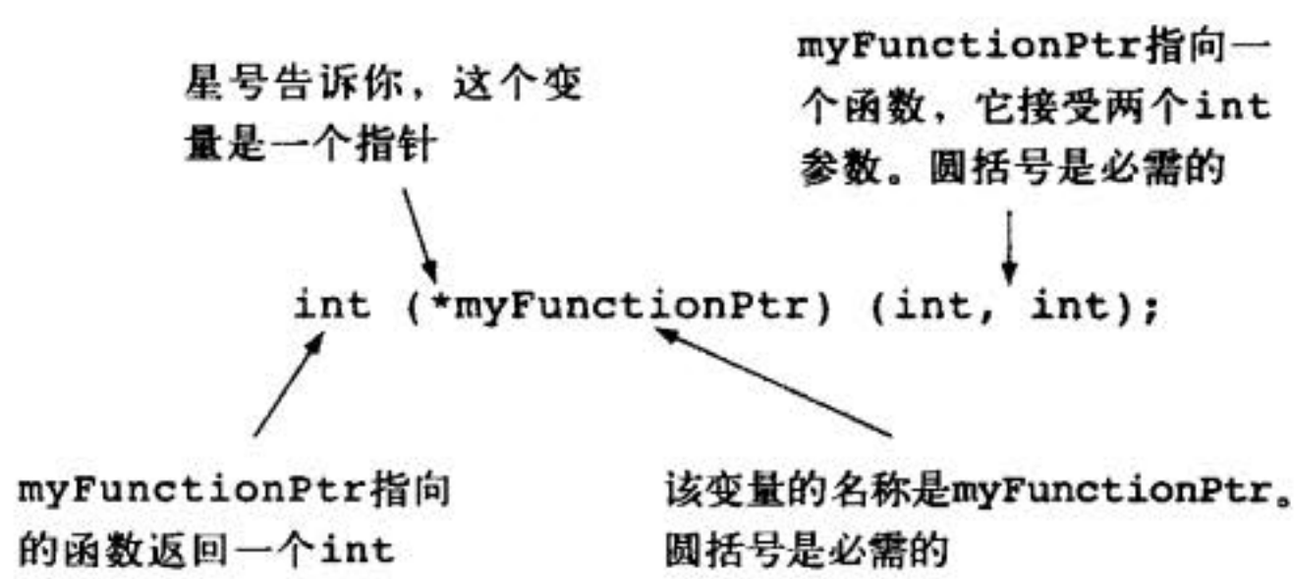


图16-1 函数指针剖析图

函数指针的通常形式是：

```
return_type (*name)(list of argument types);
```

函数指针是C语法的一个低谷。它不是从左向右读，也不是从右向左读，而是从内部向外部读。更复杂的函数指针声明可能很快就变成谜题，你将会在练习16.1中见到这样的函数指针声明。

Objective-C的一种怪异语法

函数指针从内向外读的声明方式，符合Objective-C对于方法参数的语法。还记得吧，Objective-C要求一个方法参数的类型放入到参数名称之前的圆括号中。通过支持方法参数的语法，解决了这一冲突。

当声明一个作为函数指针的方法参数时，名称位于外部。例如，指向一个没有参数或返回值的函数的指针，通常声明为如下形式：

```
void (*functionPtr)(void);
```

接受一个上面类型的函数指针作为参数的函数，则声明如下：

```
void functionWithFPArg(void (*fp)(void));
```

但是，带有同样参数类型的方法，则声明如下：

```
-(void) methodWithFFPArg:(float(*)(float)) fp;
```

只有在声明一个方法参数的类型时，将函数指针的名称放在一个函数指针声明的最后才有效。在其他的情况下，将名称放在最后将会导致编译器错误。

可以声明函数指针的数组。如下的代码行将fpArray声明为一个有10个函数指针的数组。每个函数接受单个的参数，是指向一个浮点数的指针，并且，函数返回void：

```
void (*fpArray[10])(float*);
```

函数指针可以指向这样的一个函数，它用另一个函数指针作为一个参数或返回值：

```
void (*(*complicatedFunctionPointer)(void))(void);
```

complicatedFunctionPointer是指向一个函数的指针，该函数没有参数，并且返回一个指向函数的指针，所指向的函数没有参数并且返回void。

像前面的声明很难看，但是，可以通过用一个typedef来隐藏难看的地方，从而使代码更整齐：

```
typedef void (*(*complicatedFunctionPointer)(void))(void);
complicatedFunctionPointer fp;
```

16.1.1 用函数指针调用一个函数

如下的示例展示了如何给一个函数赋值一个函数指针，以及如何使用该函数指针调用函数：

```
void logInt( int n )
{
  NSLog("The integer is: %d", n);
}

void (*myFunctionPtr)(int); // Declare a function pointer
myFunctionPtr = logInt;     // Make it point to logInt
myFunctionPtr( 5 );         // Execute the function through the pointer
```

要让函数指针引用一个函数，直接将函数的名称赋值给它。该函数必须在位于赋值之前的一个声明中定义过或可见。

要通过函数指针调用一个函数，直接给函数指针添加参数，并且参数放在圆括号中。通过函数指针调用函数，就像常规的函数调用一样，只不过使用的是函数指针变量，而不是函数名称，如上面的代码段所示。

注意 不需要对函数指针使用取址运算符(&)或解引用运算符(*)。编译器知道哪个名称是函数或函数指针，以及哪个名称是常规变量。然而，如果想要使用它们，也没有问题，如下所示：

```
myFunctionPtr = &logInt; // The same as myFunctionPtr = logInt;
(*myFunctionPtr)( 5 ) // The same as myFunctionPtr( 5 );
```

编译器不在乎使用哪种形式。

16.1.2 使用函数指针

函数指针的一个主要用途是用于回调。当你有一个函数或方法要为你做一些工作，但是，你想要有机会在这一过程中的某些地方插入自己的代码，回调就用于这种情况。为此，给工作

函数或方法传递一个指针，指向包含你想要执行的代码的一个函数。在适当的时候，工作函数或方法将会为你调用你的函数。

例如，NSMutableArray提供了如下的方法用于定制的排序：

```
- (void)sortUsingFunction:
            (NSInteger (*)(id, id, void *))compare
                        context:(void *)context
```

当调用sortUsingFunction:context:时，该方法会对接收者的内容排序。要执行排序，sortUsingFunction:context:必须检查成对的数组元素并确定它们的顺序。为了做出这些决定，sortUsingFunction:context:调用compare函数，而你在调用该方法时通过指针传入了compare函数。

compare必须查看它接受的两个对象，并且（根据你所要求的任何标准）确定它们的顺序是NSOrderedAscending、NSOrderedSame或NSOrderedDescending。

注意 NSOrderedAscending、NSOrderedSame或NSOrderedDescending都是Foundation框架定义的整数常量。

sortUsingFunction:context:还将它接受的void*指针传递给compare，作为其context参数。这是一个纯粹的传递过程，sortUsingFunction:context:不会查看或修改context。如果compare不需要任何额外的信息，则context可能是NULL。

程序清单16.1对包含了一些NSNumber对象的一个数组排序，传入了一个BOOL的地址，以控制数值排序的方向。

程序清单16.1 ArraySortWithFunctionPointer.m

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

NSInteger numericalSortFn( id obj1, id obj2, void* ascendingFlag )
{
  int value1 = [obj1 intValue];
  int value2 = [obj2 intValue];
  if ( value1 == value2 ) return NSOrderedSame;
  if ( *(BOOL*) ascendingFlag )
    {
     return ( value1 < value2 ) ?
          NSOrderedAscending : NSOrderedDescending;
    }
  else
  {
     return ( value1 < value2 )
     ? NSOrderedDescending : NSOrderedAscending;
    }
}

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

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

  // Put some number NSNumber objects in an array
  NSMutableArray *numberArray = [[NSMutableArray alloc] initWithCapacity: 5];
  [numberArray addObject: [NSNumber numberWithInt: 77]];
  [numberArray addObject: [NSNumber numberWithInt: 59]];
  [numberArray addObject: [NSNumber numberWithInt: 86]];
  [numberArray addObject: [NSNumber numberWithInt: 68]];
  [numberArray addObject: [NSNumber numberWithInt: 51]];

  NSLog( @"Before sort: %@", [numberArray description] );

  // This flag controls the sort direction.
  // Change it to NO to sort in descending order.
    BOOL ascending = YES;

  // Sort the array
  [numberArray sortUsingFunction: numericalSortFn context: &ascending];

  NSLog( @"After sort: %@", [numberArray description] );

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

注意：

- ascendingFlag作为void*传入。在能够解引用它以获得BOOL值之前，它必须强制转换为BOOL*。
- 函数的名称，在这个例子中是numericalSortFn，可以充当该函数的一个正确类型的指针。这里，在调用sortUsingFunction:context:而没有定义一个单独的函数指针变量时，它用作参数。

16.2 使用函数指针的问题

在某些情况下，需要将某些代码交给程序的其他部分或其他线程来执行，这时候，使用函数指针作为回调有一个很大的不便之处。函数需要的任何环境必须打包，并且作为一个单独的变量或多个变量来提交。

使用回调的大多数设计，都用到了程序清单16.1中的模式。传递回调函数的函数或方法，接受一个盲指针作为额外的参数，然后，将该指针传递回到回调函数，如图16-2所示。当环境像前面的示例中一样是一个单个的变量时，这只是一个小小的不方便。然而，如果函数需要一个更加复杂的环境，必须创建并载入一个定制的结构来保存环境，然后，将指向该结构的指针作为环境变量来传递。作为一种替代方法，可以将环境变量包装到一个NSDictionary中，然后，将该字典作为环境传递。如果环境涉及多个变量，则这两种方法都显得很笨拙。

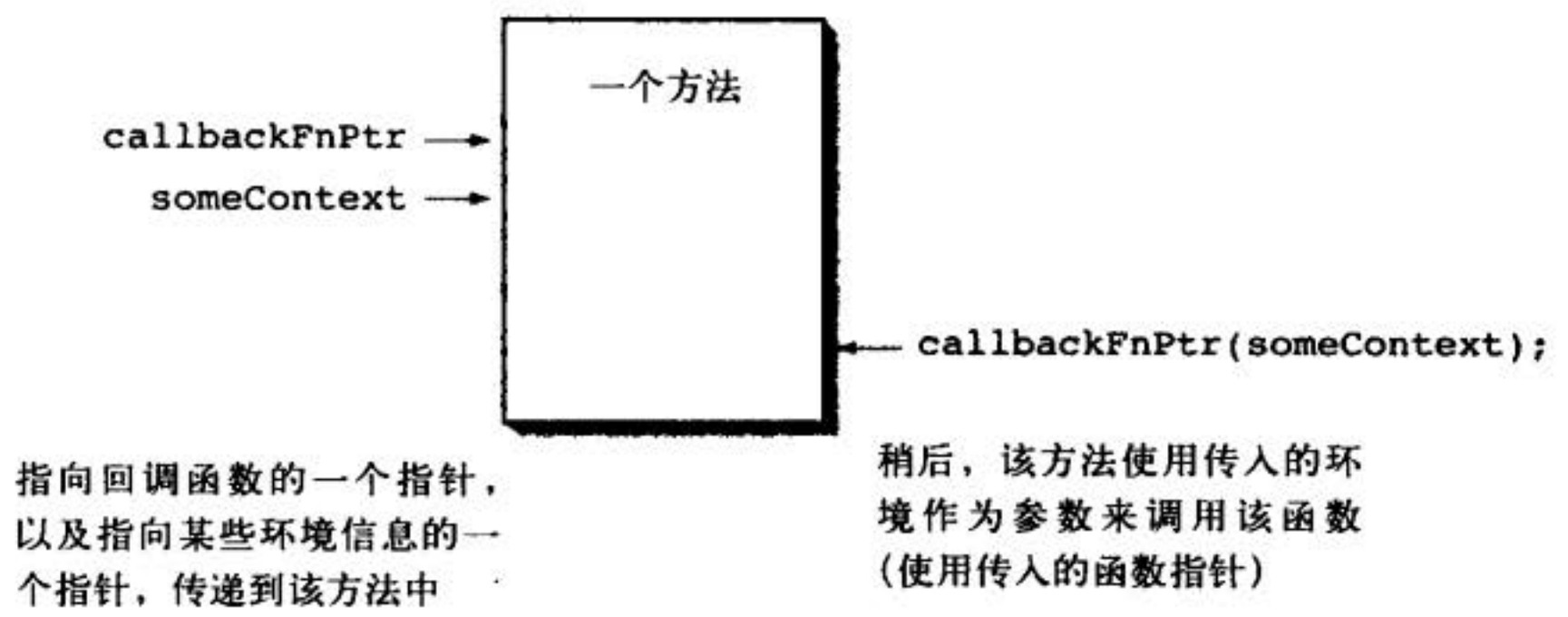


图16-2 向一个方法传递一个回调函数

16.3 NSInvocation

NSInvocation对象接收一条Objective-C消息，并且将其转换为一个对象。调用对象存储了消息的接收者（用调用术语来说，叫做目标）、选择器和参数。调用对象可以保存，以用于随后的调用或者传递给代码的其他部分。当向调用对象发送一条invoke消息时，调用对象会使用保存的选择器和参数，向目标对象发送一条消息。

例如，考虑一个LineGraphic类，它带有一个drawWithColor:width:方法，该方法使用指定的颜色和线条宽度来绘制一条线条：

```
LineGraphic *graphic = ...
[graphic drawWithColor: [NSColor redColor] width: 2.0];
```

程序清单16.2展示了如何将前面的消息转换为一次调用。

程序清单16.2 构造一个NSInvocation

```
LineGraphic *graphic = ...

NSInvocation *drawInvocation =
    [NSInvocation invocationWithMethodSignature:
        [graphic methodSignatureForSelector:
            @selector(drawWithColor:width:)]];

[drawInvocation setTarget: graphic];
[drawInvocation setSelector: @selector(drawWithColor:width:)];

[drawInvocation retainArguments];

NSColor *color = [NSColor redColor];
float linewidth = 2.0;
```

```
[drawInvocation setArgument: &color atIndex: 2];
[drawInvocation setArgument: &linewidth atIndex: 3];
```

要建立一次调用，NSInvocation需要知道将要封装的消息的返回类型及消息的参数类型。消息的选择器只是一个名称，不会携带任何类型信息，因此，必须通过调用目标的methodSignatureForSelector:来获取类型信息。这是所有的类继承自NSObject的一个方法。它返回一个NSMethodSignature对象，该对象是选择器的返回类型和参数类型的编码表示。最后，将返回的NSMethodSignature传递给NSInvocation的invocationWithMethodSignature:类方法，以创建调用。

注意 *在程序清单16.2中，对methodSignatureForSelector:的调用嵌套在对invocationWithMethodSignature:的调用中，因此，没有显式的NSMethodSignature变量。*

接下来，使用setTarget:和setSelector:设置调用的目标。

NSInvocation没有默认地保留其目标或者其任何的参数。如果想要保存一个调用以供将来执行，应该通过向调用发送一条retainArguments消息，要求调用保留其目标和参数。这防止了目标和参数在调用该调用之前被释放。

封装的消息的参数使用setArgument:atIndex:方法来设置：

- 传入将要用作参数的变量的地址，而不是变量自身。不能在setArgument:atIndex:消息中直接使用一个值：

```
[drawInvocation setArgument: 2.0 atIndex: 3]; // WRONG!
```

- 如果选择器有一个参数，它不是一个对象（这个参数是一个基本C类型，例如int或float），则必须直接使用基本类型的地址。不需要将width参数包装到一个NSNumber对象中。
- 参数通过它们在消息中的位置来识别。注意，索引从2开始。索引0和1为隐藏的方法参数self和_cmd而保留（关于方法的隐藏参数，参阅第5章）。

既然已经创建了drawInvocation，可以将其存储，或者将其转交给其他的代码。当准备好绘制线条时，直接执行如下的一行代码就可以了：

```
[drawInvocation invoke];
```

像上面的这个调用，可以在一个绘图程序中用作显示列表方案的一部分。每次调用，被存储到一个数组中，它封装了在最终的场景中绘制一个元素所需的消息。当准备好绘制场景时，遍历调用的数组并且依次调用每一个：

```
NSMutableArray *displayList = ...
for NSInvocation invocation in displayList
{
  [invocation invoke];
}
```

还有另外两点：

- NSInvocation可以调用任意多次。
- 在一个NSInvocation中，也可能使用返回值来封装一条消息。要获取该返回值，向调用发送一条getReturnValue:消息，如下所示：

```
double result;
[invocationReturningDouble getReturnValue: &result];
```

getReturnValue:的参数必须是指向与调用的返回类型相同类型的一个变量的指针。如果在向调用发送一条invoke消息之前，向它发送了一条getReturnValue:消息，那么结果是不确定的。值是垃圾，并且如果试图用它做任何事情，则可能会导致程序崩溃。

在Cocoa框架中，NSInvocation对象用来调度在一个时间间隔(NSTimer)后执行的操作，并且，它还用于Cocoa撤销机制中(NSUndoManager)。NSInvocation对象解决了使用函数指针的一个问题，它们以封装消息的参数的形式，至少自身携带了其某些环境。也就是说，NSInvocation对象有一个主要的缺点，正如你在程序清单16.2中所看到的，它们难以构造。

16.4 块

块在很多方面与函数类似，但有一个重要的差异是，块能够直接访问其包围的环境中的变量。“直接访问”意味着块可以使用在其包围环境中声明的变量，即便这些变量没有通过一个参数列表传递到块中。块声明如下：

```
^(argument_list){ body };
```

块以一个脱字符号(^)开始，后面跟着圆括号括起来的一个参数列表，以及花括号括起来的一条或多条语句。这个表达式叫做块直接量，其定义如下：

- 块可以通过在主体中使用一条return语句来返回一个值。
- 不必指定返回类型，编译器通过查看块的主体来确定返回类型。
- 如果块没有参数，则参数列表写成(void)。
- 块直接量没有名称，这就是为什么有时候称其为匿名函数的原因。

下面是一个简单的块示例，它接受一个参数，将其乘以2，然后返回倍增后的值：

```
^(int n){ return n*2; };
```

可以通过在一个块直接量的后面圆括号括起来的参数中添加值，来直接调用它：

```
int j = ^(int n){ return n*2; }( 9 ); // j is now 18
```

这有效，但是有点傻，可以通过直接编写如下的代码来得到同样的结果：

```
int j = 2 * 9;
```

在通常的应用中，一个块直接量会赋值给类型为块指针的一个变量，或者将块直接量用作一个函数或方法的参数，而该函数或方法接受一个块指针参数。

16.4.1 块指针

作为块指针（指向一个块的指针）的一个变量声明如下：

```
return_type (^name)(list of argument types);
```

这应该看上去很熟悉，因为它与一个函数指针的声明具有几乎相同的形式，只不过*换成了^。

注意 不要将保存了一个块指针的变量称为块变量。本章后面会讲到，术语“块变量”保留用于表示一种特殊的实体。

下面的例子展示了如何声明一个块指针，为其分配一个块，然后，通过块指针来调用该块：

```
int (^doubler)(int); // doubler is typed as a pointer to a block

doubler = ^(int n){ return n*2; };

int j = doubler( 9 ); // j is now 18
```

可以使用块指针作为一个函数或方法的参数：

```
// someFunction is a function that takes a block pointer as an argument

void someFunction( int (^blockArg)(int) );

int (^doubler)(int)= ^(int n){ return n*2; };

someFunction( doubler );
```

也可以在一个函数调用或一个Objective-C消息中直接使用一个块直接量：

```
void someFunction( int (^blockArg)(int) );

someFunction( ^(int n){ return n*2; } );
```

注意 Objective-C方法声明中对于块指针也有奇怪的语法，与它们对函数指针所做的一样，当声明接受一个块指针参数的方法时，名称放在类型声明之外：

```
- (void) doSomethingWithBlockPointer:
          (float (^)(float)) blockPointer;
```

16.4.2 访问变量

块拥有：

- 对其封闭作用域内可见的自动变量的只读访问。[⊖]
- 对于一个函数中声明的静态变量的读/写访问，以及对于外部变量的读/写访问。
- 对于声明为块变量的特殊变量的读/写访问。

下面是一个块访问其封闭作用域中的一个本地变量的简单示例：

```
int j = 10;

int (^blockPtr)(int) = ^(int n){ return j+n; };

int k = blockPtr( 5 ); // k is now 15
```

⊖ 各种变量的作用域在本书第2章中讨论。

当执行流程通过块直接量时，块用于其环境中的一个本地变量的值，就是该本地变量的值，如下所示：

```
1 int j = 10;
2
3 int (^blockPtr)(int) = ^(int n){ return j+n; };
4
5 j = 20;
6
7 int k = blockPtr ( 5 ); // k is 15, not 25 as you might expect
```

在前面的代码中：

- 第1行：本地变量j设置为10。
- 第3行：定义块blockPtr。blockPtr能够访问本地变量j。blockPtr使用的j值，绑定为当程序执行通过该行时j所拥有的值。实际上，blockPtr现在有了j的一个私有副本，其值设置为10。
- 第5行：为了展示blockPtr的j是在第3行绑定的，将j重新设置为20。
- 第7行：使用参数5计算blockPtr，产生一个10（在执行第3行的时候j的值）+5（参数）=15的返回值。

块对于本地变量的访问是只读的。如下的代码试图在blockPtr块中设置j的值，这会导致一个编译器错误，因为blockPtr对于本地变量j的访问是只读的：

```
int j = 10;

void (^blockPtr)(void) = ^(void){ j = 20; };
```

注意 如果一个本地变量保存了指向一个对象的指针，那么块不能修改该变量从而指向一个不同的对象。但是，块仍然可以修改该变量所指向的对象：

```
NSMutableArray *localArray = ...

void (^shortenArray)(void) = ^(void){ [localArray removeLastObject]; };

// Removes the last object in localArray

shortenArray();
```

编译器允许块通过指针来访问静态和外部变量。块为该类型的变量所查找的值，是当执行块时该变量的值，而不是在定义该块时的值：

```
static int j = 10;

int (^blockPtr)(int) = ^(int n){ return j+n; };

j = 20;
int k = blockPtr ( 5 ); // k is 25
```

16.4.3 块变量

以新的类型修饰符__block定义的变量，叫做块变量：

```
__block int integerBlockVariable;
```

对于在块变量作用域中定义的任何块，它都是可见的、可共享的并且可修改的。例如：

```
1 __block int j = 10;
2
3 void (^blockPtr_1)(void) = ^(void){ j += 15; };
4 void (^blockPtr_2)(void) = ^(void){ j += 25; };
5
6 blockPtr_1(); // j is now 25
7 blockPtr_2(); // j is now 50
```

在前面的代码中：

- 第1行：j声明为一个__block变量，并设置为10。
- 第3行：定义blockPtr_1。因为j已经声明为一个__block变量，所以允许blockPtr_1设置j的值。
- 第4行：类似地，允许blockPtr_2设置j的值。blockPtr_1和blockPtr_2共享了对变量j的读写访问。
- 第6行：执行blockPtr_1，将j增加15，最终得到一个值为25。
- 第7行：执行blockPtr_2，将j增加25，最终得到一个值为50。

块变量像任何其他的自动变量一样，从栈开始。但是，如果复制引用了一个块变量的块，则该块变量随着该块一起从栈移动到堆中（参见16.4.7节）。

注意 正如前面所述，术语“块变量”指的是使用__block修饰符声明的一个变量。不要把“块变量”和“保存指向块的一个指针的变量”搞混淆了。块变量不是一个块。

16.4.4 块是基于栈的

当在一个函数或方法中定义一个块直接量时，编译器在栈上创建一个结构来保存块所引用的任何本地变量的值、它引用的读/写变量的地址及指向块的可执行代码的指针。

注意 块结构是在栈上创建的，但是，块的可执行代码并不在栈上。它位于带有所有其他的可执行代码的程序的文本部分。

块与自动变量拥有同样的生命期。当块直接量超出作用域时，它就是未定义的，就像自动变量超出其作用域一样。块直接量的作用域，与一个自动变量的作用域的确定方式相同（参见第2章）：如果一个块直接量定义于一条复合语句内部（位于一对花括号之间），当程序执行离开该复合语句之后，块直接量就超出了其作用域。如果一个块直接量定义于一个函数中，当函数返回时，它就超出了作用域。如下的代码是不正确的：

```
int (^doubler)(int);
{
```

```
    ...
    doubler = ^(int n){ return n*2; };
    ...
}
...
int j = doubler( 9 ); // WRONG! Bug!
```

在上面的示例中，j是未定义的。当执行doubler(9)时，doubler变量所指向的块已经超出作用域了，并且该块已经销毁了。

注意 如果你尝试前面的例子，它有可能看上去正确地工作了，j可能很好地设置为18。但是，这可能是编译器在这个示例中已经安排了代码的一种偶然的方式。当代码超出作用域之后，编译器自由地重新使用块在栈帧中所占用的空间。如果编辑器重新使用该空间，尝试执行超出作用域的块的结果可能是一个不正确的j的值，更有可能会导致程序崩溃。

16.4.5 全局块

也可以将一个块直接量分配给一个文件全局块指针变量：

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

void (^logObject)(id) =
    ^(id obj){ NSLog( @"Object Description: %@", [obj description]); };

// logObject( someObj ) may be used anywhere in the file.
```

编译器在较低的内存中创建全局作用域的块，就像创建其他文件作用域变量一样。全局块不会超出作用域。

16.4.6 块是Objective-C对象

这似乎令人吃惊，但是，块也是Objective-C对象。一个新创建的块，只不过是在堆栈上创建的一个Objective-C对象的唯一示例。块是NSObject的几个私有子类之一的示例。Apple没有提供块类的头文件，因此，无法子类化它们或者以一种Objective-C的意义对它们做任何事情，除了可以向它们发送copy、retain、release和autorelease消息。块的复制和内存管理在第16.4.7节中介绍。

16.4.7 复制块

块的一个主要用途是将一部分工作（一些代码加上一些环境）传递到作用域之外，以便稍后处理。向调用的一个函数或方法传递一个块，这是安全的（只要该函数或方法将要在同一线程上执行）。但是，如果想要把一个块传递给不同的线程，或者将一个块作为返回值传递出当前作用域，将会发生什么情况呢?当前函数或方法返回时，其栈帧将会销毁。在其作用域中定义的任何块也变得无效。

为了保留一个块，必须复制它。当复制一个块时，在堆上创建副本。基于堆的副本随后安全地将栈返回给调用函数或者传递给其他的线程。

如果直接用C编写程序，可以使用Block_copy()函数，如下所示：

```
int(^doublerCopy)(int) = Block_copy( ^(int n){ return n * 2; } );
```

在Objective-C中，可以向块发送一条copy消息：

```
int(^doublerCopy)(int) = [^(int n){ return n * 2; } copy];
```

上面的两个示例是对等的。在任何一条语句中，都可以使用一个块指针而不是块直接量。

当复制一个块时，新的块会获取块所引用的任何自动变量的值的副本（块根据值来访问自动变量。当块对象创建时，该变量的值会复制到块对象中）。

什么是块变量

块变量根据引用来访问。复制一个块，随后当程序执行离开当前作用域时，让复制的块引用一个已销毁的变量，这么做不会很有用。

为了弥补这一点，在你复制一个块时，编译器会将块所引用的任何块变量从栈中移动到堆中的一个位置。随后，编译器更新引用了该块变量的任何块，以便它们拥有该变量的新的地址。

在这种情况下，编译器行为的后果之一是，获取一个块变量的地址并用它做任何事情是非常糟糕的想法。在复制操作之后，最初的地址所引用的一个内存位置现在可能是垃圾。

16.4.8 块的内存管理

如果使用Block_copy()复制块，最终必须使用Block_release()调用来对应前面的调用。如果使用Objective-C copy消息，并且将要使用引用计数，则必须使用一条release或一条autorelease消息来对应copy消息：

```
int(^getDoublerBlock())(int)
{
  int(^db)(int) = ^(int n){ return 2*n; };
  // The returned block is autoreleased. This balances the copy
  // and makes getDoublerBlock conform to the naming convention
  // for memory management.
  return [[db copy] autorelease];
}
...
int(^doubler )(int) = getDoublerBlock(); // Get the block

int sevenDoubled = doubler(7); // Use the block
```

不要将Block_copy()和Block_release()调用与Objective-C的copy和release消息混合使用。

如果一个块引用了保存一个对象的变量，则当复制该块时，会保留该对象；而释放该块时，也会释放该对象。

注意 当复制引用了__block变量的一个块时，__block变量中保存的对象不会保留。

当在一个方法主体中复制一个块时，规则略微复杂一些：

- 要复制的块中对于self的直接引用，会导致self被保留。
- 要复制的块中对一个对象的实例变量的引用（直接的或者通过一个访问器方法），会导致self被保留。
- 方法中对于本地变量中保存的一个对象的引用，会导致该对象而不是self被保留。

当复制一个块时，应该小心。如果复制该块的代码位于一个方法内部，并且该块引用对象的任何实例变量，则复制会导致self保留。很容易建立一个保留循环，它会阻止对象被销毁。

程序清单16.3给出了一个类的接口部分，这个类有一个实例变量name用来存储名称，还有一个方法logMyName用来显示该名称。logMyName使用实例变量loggingBlock中存储的一个块来完成实际的显示。

程序清单16.3 ObjectWithName.h

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

@interface ObjectWithName : NSObject
{
  NSString *name;
  void (^loggingBlock)(void);
}

- (void) logMyName;
- (id) initWithName:(NSString*) inName;

@end
```

程序清单16.4给出了对应的实现文件。

程序清单16.4 ObjectWithName.m

```
 1 #import "ObjectWithName.h"
 2
 3 @implementation ObjectWithName
 4
 5 - (id) initWithName:(NSString*) inputName
 6 {
 7 if (self = [super init] )
 8 {
 9 name = [inputName copy];
10 loggingBlock = [^(void){ NSLog( @"%@", name ); } copy];
11 }
12 return self;
13 }
14
```

```
15 - (void) logMyName
16 {
17 loggingBlock();
18 }
19
20 - (void) dealloc
21 {
22 [loggingBlock release];
23 [name release];
24 [super dealloc];
25 }
```

ObjectWithName是一个非常简单的类。然而，ObjectWithName的这个版本有一个保留循环。如果创建了一个ObjectWithName对象，则当你释放它时，它不会销毁。

问题在于程序清单16.4的第10行：

```
loggingBlock = [^(void){ NSLog( @"%@", name ); } copy];
```

要在实例变量loggingBlock中存储块，必须复制块直接量并且将副本赋值给该实例变量。这是因为当initWithName:返回时，块直接量会超出作用域。复制块会将副本放到堆中（就像一个常规的Objective-C对象一样）。然而，块直接量引用该实例变量名，因此，副本会导致self被保留，建立一个保留循环。块现在拥有了该对象的所有权，该对象拥有了块的所有权（因为它已经复制了该块）。对象的引用计数不会变为0，并且，它的dealloc方法永远不会调用。

可以通过将程序清单16.4的第10行修改为如下所示，从而修正这一问题：

```
loggingBlock = [^(void){ NSLog( @"%@", inputName ); } copy];
```

通过这一修改，块复制操作保留了输入参数inputName，而不是实例变量名称。由于该块不再引用对象的任何实例变量，所以self不会保留，并且没有保留循环。对象将仍然具有相同的行为，因为name和inputName拥有相同的内容。

注意 前面的规则只是对于在一个方法中进行块复制而单独存在，从而强调了一个块访问一个对象的实例变量的后果。但是，在方法内部和外部复制一个块，真的没有什么显著的差别。唯一的区别是，在方法的外部，没有办法不引用对象自身而引用其实例变量。

16.4.9 陷阱

由于块是基于栈的对象，因此它们表现出一些陷阱。如下的代码段是不正确的：

```
void(^loggingBlock)(void);

BOOL canWeDoIt = ...

// WRONG
```

```
if ( canWeDoIt )
  loggingBlock = ^(void){ NSLog( @"YES" ); };
else
  loggingBlock = ^(void){ NSLog( @"NO" ); };

// Possible crash
loggingBlock();
```

在这段代码的末尾，loggingBlock是未定义的。一条if语句的if和else子句及循环体是独立的语法作用域，即便它们都是单条语句而不是复合语句。当程序执行离开该作用域时，编译器可以自由地销毁该块并且让loggingBlock指向垃圾。

要修改这段代码，必须复制该块，然后，记住当用完它之后释放它：

```
void(^loggingBlock)(void);

BOOL canWeDoIt = ...

if ( canWeDoIt )
  loggingBlock = [^(void){ NSLog( @"YES" ); } copy];
else
  loggingBlock = [^(void){ NSLog( @"NO" ); } copy];

// Remember to release loggingBlock when you are finished
```

这个示例也是对的：

```
NSMutableArray *array = ...

// WRONG!

[array addObject: ^(void){ doSomething; }];
return array; //
```

还记得，添加到集合对象中的对象接收一条retain消息。然而，在这个例子中，retain帮不上忙，因为对于一个基于栈的块来说，retain是没有任何操作的。再一次，为了修正这一问题，必须复制该块：

```
NSMutableArray *array = ...
[array addObject: [[^(void){ doSomething; } copy] autorelease]];
return array;
```

在上面的代码段中，copy消息将块的一个副本放到了堆上。autorelease消息对应了copy。当复制的对象放到数组中时，它接收一条retain消息，而随后从数组中删除该块时或销毁该数组时，一条release消息对应于这条retain消息。

16.4.10 Cocoa中的块

从Mac OS X Snow Leopard (v 10.6)开始，Apple已经开始在整个Cocoa框架中部署块了。本小节简单介绍Apple已经添加了使用块的功能的3个部分。

使用NSOperationQueue实现并发

并发（多线程）编程很难正确地进行。为了让程序员易于编写正确的多线程程序，Apple已经引入了GCD。GCD通过创建和管理一个线程池来实现并发。线程池是一组线程，它们可以分配给各种任务并且当任务完成之后再重用。GCD隐藏了管理线程池的细节，并且为程序员提供了一个相对简单的界面。

Cocoa类NSOperationQueue为GCD提供了一个高层级的接口。思路很简单：创建NSOperationQueue并且添加工作单元，以块的形式，排队等待执行。在NSOperationQueue之下，GCD安排在一个单独的线程上执行块：

```
NSOperationQueue *queue = [[NSOperationQueue alloc] init];

[queue addOperationWithBlock: ^(void){ doSomething; } ];

// doSomething will now execute on a separate thread
```

- 传递给GDC（通过NSOperationQueue或通过低层级的C接口）的一个块必须拥有如下形式：

```
void (^block)(void)
```

它不接受参数，也没有返回值。

- GCD机制负责复制提交给它的块，以及当不再需要的时候释放它们。

注意 并发编程是一个复杂的话题。要了解NSOperationQueue和GCD的完整介绍，参见Apple的Concurrency Programming Guide。[⊖]

集合类

Foundation集合类现在有了允许对集合中的每个对象应用一个块的方法。NSArray拥有如下的方法：

```
- (void)enumerateObjectsUsingBlock:
        (void (^)(id obj, NSUInteger idx, BOOL *stop))block
```

这个方法针对数组中的每个对象调用一次块，块的参数是：

- obj，指向当前对象的一个指针。
- Idx，当前对象的索引（idx等同于一个普通for循环中的循环索引）。
- stop，指向一个BOOL的指针。如果块将stop设置为YES，则当块返回时，-enumerateObjectsUsingBlock:终止。它等同于普通C循环中的一条break语句。

程序清单16.5使用-enumerateObjectsUsingBlock:来显示数组中的每个对象的描述。

程序清单16.5 DescribeArrayContents.m

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

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

⊖ 参见http://developer.apple.com/mac/library/documentation/General/Conceptual/ConcurrencyProgrammingGuide。

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

  NSArray *array =
      [NSArray arrayWithObjects: @"dagger", @"candlestick",
           @"wrench", @"rope", nil];

  void (^loggingBlock)(id obj, NSUInteger idx, BOOL *stop) =
      ^(id obj, NSUInteger idx, BOOL *stop)
          { NSLog( @"Object number %d is a %@",
               idx, [obj description] ); };

  [array enumerateObjectsUsingBlock: loggingBlock];

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

如果编译并运行该程序，将会看到如下的结果：

```
DescribeArrayContents [50642:a0b] Object number 0 is a dagger
DescribeArrayContents [50642:a0b] Object number 1 is a candlestick
DescribeArrayContents [50642:a0b] Object number 2 is a wrench
DescribeArrayContents [50642:a0b] Object number 3 is a rope
```

Did-End回调

在本书中，对于AppKit没有介绍太多，但是，本书假设你已经熟悉了在Mac OS X上保存文件。可以在应用程序中选择File > Save，并且，如果这是第一次保存一个文件，将会出现一个保存表单，以便你可以命名该文件并选择保存该文件的位置。你可以自己做出选择并单击“Save”按钮，或者，如果你改变了想法，单击“Cancel”按钮。在单击了一个按钮之后，表单会收起并消失。

当调用启动该表单的方法时，Cocoa给你机会来注册当用户取消表单时要执行的一些代码（这就是你放置真正将文件保存到磁盘中的代码的地方）。

在Mac OS X Snow Leopard (v 10.6)之前，一个Save表单将从如下的代码而不是一个复杂的方法开始：

```
- (void)beginSheetForDirectory:(NSString *)path
          file:(NSString *)name
          modalForWindow:(NSWindow *)docWindow
          modalDelegate:(id)modalDelegate
          didEndSelector:(SEL)didEndSelector
          contextInfo:(void *)contextInfo
```

当用户取消表单时，表单向注册为modalDelegate的对象发送选择器didEndSelector所表示的消息。通常，modalDelegate是初始化面板的对象。didEndSelector形式如下：

```
- (void)savePanelDidEnd:(NSSavePanel *)sheet
              returnCode:(int)returnCode
             contextInfo:(void *)contextInfo;
```

- Sheet是NSSavePanel对象自身的一个指针。
- returnCode是一个整数，它指定用户单击了哪个按钮。
- contextInfo是当beginSheetForDirectory: 调用的时候传递给它的信息的一个盲指针。这就是如何将调用表单的对象的信息，传递给负责操作用户输入的代码。

对于Mac OS X Snow Leopard及其以上版本，上面的方法已经废弃了，[⊖]并且将用如下的方法替代：

```
- (void)beginSheetModalForWindow:(NSWindow *)window
                completionHandler:(void (^)(NSInteger result))handler
```

当取消表单的时候，直接传入一个块来执行。块可以捕获任何所需的环境，因此，不再需要盲指针contexInfo。

注意 file和directoryPath参数将被删除，转换为一个单独清理的一部分，该清理不涉及块。

16.4.11 样式问题

将一个块直接量的语句放在单独的一行，使得调试很困难，例如：

```
^(void){doStuff; doMoreStuff; evenMore; keepGoing; lastStatement;}
```

可以在doStuff;上设置一个调试断点，但是，没有办法在块中的其他任何语句上设置步进或断点。如果在doStuff;上停止并尝试步进，调试器将跳转到紧跟在块直接量之后的行，这使得不可能调试直接量中后续的行。如果块直接量很重要并且可能需要调试，应该在块的主体中单独一行上放置该语句，如下所示：

```
^(void){doStuff;
        doMoreStuff;
        evenMore;
        keepGoing;
        lastStatement;}
```

正如前面提到的，可以把一个块直接量直接放置到一个函数或方法调用中：

```
someFunction( otherArgs, ^(void){ doStuff;
                                  doMoreStuff;
                                  evenMore;
                                  keepGoing;
                                  lastStatement;} );
```

也可以把一个块赋值给一个块指针变量，并且使用该块指针作为参数。选择哪一种方法，

⊖ 在给定的主OS版本中，状态修改为废弃的方法在该版本中仍然可用，但是，在将来的OS主版本中将取消。例如，在Mac OS X 10.6中标记为废弃的方法，在Mac OS X 10.7中可能无法使用。

完全是个人偏好的问题，有些人很讨厌创建一个额外的变量（编译器可能会优化掉它），而另一些人则认为将块直接量放入到函数调用中会使得代码难以阅读。

16.5 一些原理上的限制

块是功能全面的，并且，它们显然是Apple为未来的Objective-C和Mac OS X规划的一个重要部分。然而，块带来了一些值得三思的问题：

- 术语“块”已经用过了。在关于C语言的几乎每一本书中，它与“复合语句”交叉使用。这可能会在某些情况下引起混淆。
- 块是面向函数的而不是完全面向对象的。如果你对其强加给一种纯面向对象思想的话，这可能是一个问题。
- 块完全打破了封装。对于那些不是通过一个参数列表或访问器方法访问的变量，块对它们的访问，带来了与使用全局变量相同的很多问题。

  使用`__block`变量并复制块会产生相关的对象，你可以创建单独的对象（可能属于不同的类），它们彼此看不到对方，但是，却通过堆上的一个变量通信。
- 和C++中的运算符重载一样，块的使用方法可能会导致“设计你自己的语言综合征”，代码非常简单，但是很难让其他人（或者你自己在几个月之后）阅读和理解。对于独立的开发者来说，这可能不是问题，但是，如果你是一个开发团队的一分子，这就是个问题。

16.6 小结

本章介绍了包装功能以便稍后执行或在一个不同线程上执行的几种方法。函数指针允许将函数转交，但是，如果需要传递函数所带的一些环境的话，则需要提供一个额外的变量与函数指针配合使用。`NSInvocation`对象将目标、选择器及Objective-C消息表达式的参数包装到一个单独的对象中，可以存储该对象随后转交给执行。它们很容易使用，但是难以构建。

块是Apple添加的对C、Objective-C 2.0和C++的扩展，将其包含环境中的一系列的语句和变量包装到了一个单个的实体中。Apple用于管理并发的系统GCD，使用块作为提交在其他线程上执行的任务的媒介。从Mac OS X Snow Leopard (v 10.6)开始，Apple在整个Cocoa框架中部署了块，来替代使用`NSInvocation`对象或需要单个目标、选择器和环境参数进行回调的旧方法。

16.7 练习

1. 这是比任何其他问题更加使人迷惑的一个问题，但是，它仍然测试你对于函数指针声明的（并且，由此扩展到块指针）理解。考虑如下的声明：

```
int (*(*myFunctionPointer)(int (*)(int))) (int);
```

   myFunctionPointer是什么？

2. 使用一个`NSInvocation`重新编写第4章中的`HelloWorld`的Objective-C程序：

   不要将问候文本作为`NSString`传递给`Greeter`，创建一个`Greeting`类，它封装了问

候及发布问候的方法（该方法应该接受问候字符串作为参数并显示它）。将发布问候打包为一个NSInvocation，并且将其传递给Greeter。然后，Greeter应该通过向调用对象发布-invoke消息来发布问候。

3. 编写一个程序，使用一些简单的块，并且自己验证：
   - 在执行通过块直接量时，一个块所看到的普通自动变量的值是固定的，并且，如果在随后代码中修改该变量的值，它也不会变化。
   - 块不能在其作用域中修改一个普通自动变量的值。
   - 块不能读取和设置使用__block类型修饰符声明的一个变量。
4. 重新编写程序清单16.1中的程序，以使用一个块而不是一个函数。使用NSMutableArray方法：

```
- (void)sortUsingComparator:(NSComparator)cmptr
```

   NSComparator是指向块的指针的typedef，它接收两个对象参数并且返回与程序清单16.1中的函数相同的整数常量。

5. 编写一个程序，在一个名称的数组中查找名称，并且报告该名称在数组中的索引：
   - 使用某些名称创建一个NSArray（使用NSString对象）。
   - 创建一个本地NSString变量来保存要查找的名称，并且，用一个整数块变量来报告找到名称的索引。
   - 使用-enumerateObjectsUsingBlock:搜索该数组。
   - 确保当找到名称时，块使用stop参数来停止查找。
   - 如果要找的名称不在数组中，则保存索引的块变量的值应该为-1。
6. 使用ObjectWithName类编写一个程序（参见程序清单16.3和程序清单16.4）。
   - 给ObjectWithName的dealloc函数添加一条显示语句。
   - 在主程序中，分配ObjectWithName的一个实例。
   - 释放该对象，并且验证它没有销毁。
   - 做出文中所建议的修改，并且验证该对象现在销毁了。

第四部分
附　录

附录部分包含了对 Objective-C开发者来说其他有用的信息。

附录A　保留字和编译器指令
附录B　自由转换类
附录C　32位和64位
附录D　运行时，旧的和新的
附录E　Objective-C的资源

附录A 保留字和编译器指令

表A.1 C保留字

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
inline	int	long	register
restrict	return	short	signed
sizeof	static	struct	switch
typedef	union	unsigned	void
volatile	while	_Bool	_Complex
_Imaginary			

从Mac OS X Snow Leopard (v 10.6)开始，Apple向C中添加了关键字__block。

表A.2 额外的Objective-C保留字

id	Class	SEL	IMP
BOOL	nil	Nil	YES
NO	self	super	__strong
__weak			

表A.3中的单词只有在Objective-C协议声明中使用的时候是保留字（见第13章）。在远程消息中，它们用作类型修饰符。本书中没有介绍远程消息。

表A.3 只有在Objective-C协议声明中使用的时候是保留字

oneway	In	out	inout
bycopy	byref		

表A.4 Objective-C编译器指令

@interface	@implementation	@protocol	@end
@private	@protected	@public	@try
@throw	@catch()	@finally	@property
@synthesize	@dynamic	@class	@selector()
@protocol()	@encode	@"string"	@synchronized()

附录B　自由转换类

表B.1　自由转换为Foundation类的Core Foundation类型

Core Foundation类型	Foundation 类	可用类型 (列出的OS及其以后版本)
CFArrayRef	NSArray	Mac OS X v10.0
CFAttributedStringRef	NSAttributedString	Mac OS X v10.4
CFCalendarRef	NSCalendar	Mac OS X v10.4
CFCharacterSetRef	NSCharacterSet	Mac OS X v10.0
CFDataRef	NSData	Mac OS X v10.0
CFDateRef	NSDate	Mac OS X v10.0
CFDictionaryRef	NSDictionary	Mac OS X v10.0
CFLocaleRef	NSLocale	Mac OS X v10.4
CFMutableArrayRef	NSMutableArray	Mac OS X v10.0
CFMutableAttributedStringRef	NSMutableAttributedString	Mac OS X v10.4
CFMutableCharacterSetRef	NSMutableCharacterSet	Mac OS X v10.0
CFMutableDataRef	NSMutableData	Mac OS X v10.0
CFMutableDictionaryRef	NSMutableDictionary	Mac OS X v10.0
CFMutableSetRef	NSMutableSet	Mac OS X v10.0
CFMutableStringRef	NSMutableString	Mac OS X v10.0
CFNumberRef	NSNumber	Mac OS X v10.0
CFReadStreamRef	NSInputStream	Mac OS X v10.0
CFRunLoopTimerRef	NSTimer	Mac OS X v10.0
CFSetRef	NSSet	Mac OS X v10.0
CFStringRef	NSString	Mac OS X v10.0
CFTimeZoneRef	NSTimeZone	Mac OS X v10.0
CFURLRef	NSURL	Mac OS X v10.0
CFWriteStreamRef	NSOutputStream	Mac OS X v10.0

注意　这些是（当前）唯一的自由转换类。不应该以类型或类名作为指南。如果一个类和类型不在这个表中，那么它们是不能自由转换的，即便类名和类型名是相同的或相似的。

参见http://developer.apple .com/legacy/mac/library/ documentation/ Cocoa/Conceptual/CarbonCocoaDoc/Articles/InterchangeableDataTypes.html。

附录C　32位和64位

在过去的几年里，Apple已经转向了64位计算。Mac OS X Tiger (v 10.4)可以运行64位的命令行程序，Mac OS X Leopard (v 10.5)可以运行基于GUI（AppKit）的64位程序，而Mac OS X Snow Leopard (v 10.6)既可以运行基于GUI（AppKit）的64位程序，也可以在一些最近的Mac上启动一个64位的内核。

注意　最早的Intel Macs（带有Core Duo或Core Solo的处理器）都只是32位的。iOS设备也只是32位的。

32位和64位指的是地址内存的位数。32位的程序使用的指针是4个字节的宽度，并且可以寻址到2^{32}或大约4GB的内存。64位的程序理论上可以寻址到2^{64}或者1.8×10^{19}字节的地址。实际上，当前的Intel机器只是在一个8位的指针中使用64位的47位来寻址，其最大的寻址范围大约为140TB（1TB大约是10^{12}个字节）。

如此大的数字的要点是什么呢？计算机可以变得更快，某些学科的人们已经开始编写程序来使用非常大的地址集，达到在寻址方面突破4GB的限制。在硬件方面，4GB的物理内存已经变得很普通了（Apple现在销售的笔记本带有8GB的物理内存，而桌面计算机的物理内存达到32GB）。将可寻址地址的大小扩展到超越4GB的压力很明显。计算机结构最好以2的次方表示大小，因此，4字节之后的下一个逻辑指针大小将是8字节。

注意　当研究这一话题时，记住，Mac OS X有一个交换系统，并且每个进程都有自己的虚拟地址空间。为了执行，一个进程不需要完全载入到物理内存，交换系统负责在物理内存和磁盘上的交换区之间移动内存页。使用2GB内存的一个进程将会在拥有1GB物理内存上的机器上运行，尽管这可能要比其在拥有4GB物理内存的机器上运行慢很多（假设4GB机器不会同时运行其他的更大程序）。

C.1　64位的内核和用户程序

内核和用户程序是分开的，它们都可以是32位的或64位的。32位的内核可以运行32位和64位的用户程序。因此，64位的内核也可以。尽管最新的Intel Mac能够启动一个64位的内核，但大多数客户的计算机都安装并启动了32位的内核。这是因为，整个内核（包括设备驱动和内核扩展）必须要么是32位的，要么是64位的，没法二者兼顾。目前可以使用的64位设备驱动相对较少。大多数运行64位内核的Mac系统都倾向于运行Mac OS X Server（而不是Mac OS X的桌面版本）。这些机器通常没有需要驱动程序的打印机或者其他的外围设备。随着64位设备驱动变得可用，这种情况毫无疑问会有变化。

对应用于用户空间程序的混合大小上，也有同样的限制。如果针对64位编译一个程序，所连接的所有库和框架，以及程序所使用的任何插件，都必须在64位版本中可以使用。这对于系统提供的库和框架来说不是问题，例如Cocoa框架，它们都可以在64位版本中使用。然而，如果使用其他的库，则可能要自己针对64位编译它们（如果能够访问源代码的话），或者继续将程序构建为32位的应用程序，直到该资源的64位版本变得可用。

C.2 与为64位程序编码的区别

32位和64位执行环境之间的主要区别在于一些基本类型的大小。在32位可执行文件中，int、long和指针都是4字节的。在64位可执行文件中，int仍然是4字节的，但是，long和指针是8字节的。主要的一点，也是可能让你陷入麻烦的一点是，在64位可执行程序中，long和指针不再与int具有相同的大小。

将一个类型赋值给一个较长的类型，没有问题。但是，将一个较长的类型赋值给一个较短的类型，则会截断在较长的类型中保存的值，并且可能导致错误。这似乎是很明显的一点，但是，在C编程中，有很长一段历史忽略了指针和int之间的区别，并且忽略了很多系统调用实际的返回值比一个int要长这一事实。忽略一个指针和一个int之间的区别，或者说一个long和一个int之间的区别，当它们都是4字节的时候，没有真正的影响；但是，针对64位编译相同的代码时，将会引发真正的问题。当针对64位编码时，进行指针算术运算应该要小心，不应该将一个指针强制转型为一个int变量。

C.3 性能

将一个程序编译为64位的可执行文件会使其运行得更快吗？如果没有分别将程序编译为32位和64位的可执行文件，然后比较其执行时间的话，没办法下结论。当你走向64位的可执行文件时，还有一些竞争效应。一方面，当一个Intel CPU在x86_64 (64-bit)模式下运行时，它能够访问更多的寄存器。在函数调用中，它可以使用这些寄存器而不是堆栈来传递参数，这会使性能变得更好。但是，从32位可执行环境迁移到64位可执行环境，可能导致程序的内存占用增加（因为很多4字节变量现在是8字节变量了），有时候，这种情况很突出。更大的内存大小，可能意味着更多的指令缓存错误，这会导致糟糕的性能。

C.4 针对64位编译

要将一个程序编译为64位的，我们必须告诉编译器，你想要64位的可执行文件。如果你从命令行使用gcc来编译，则需要使用-arch标志。如下的命令将program.m编译为64位的可执行文件：

```
gcc -arch x86_64 program.m -framework Foundation
```

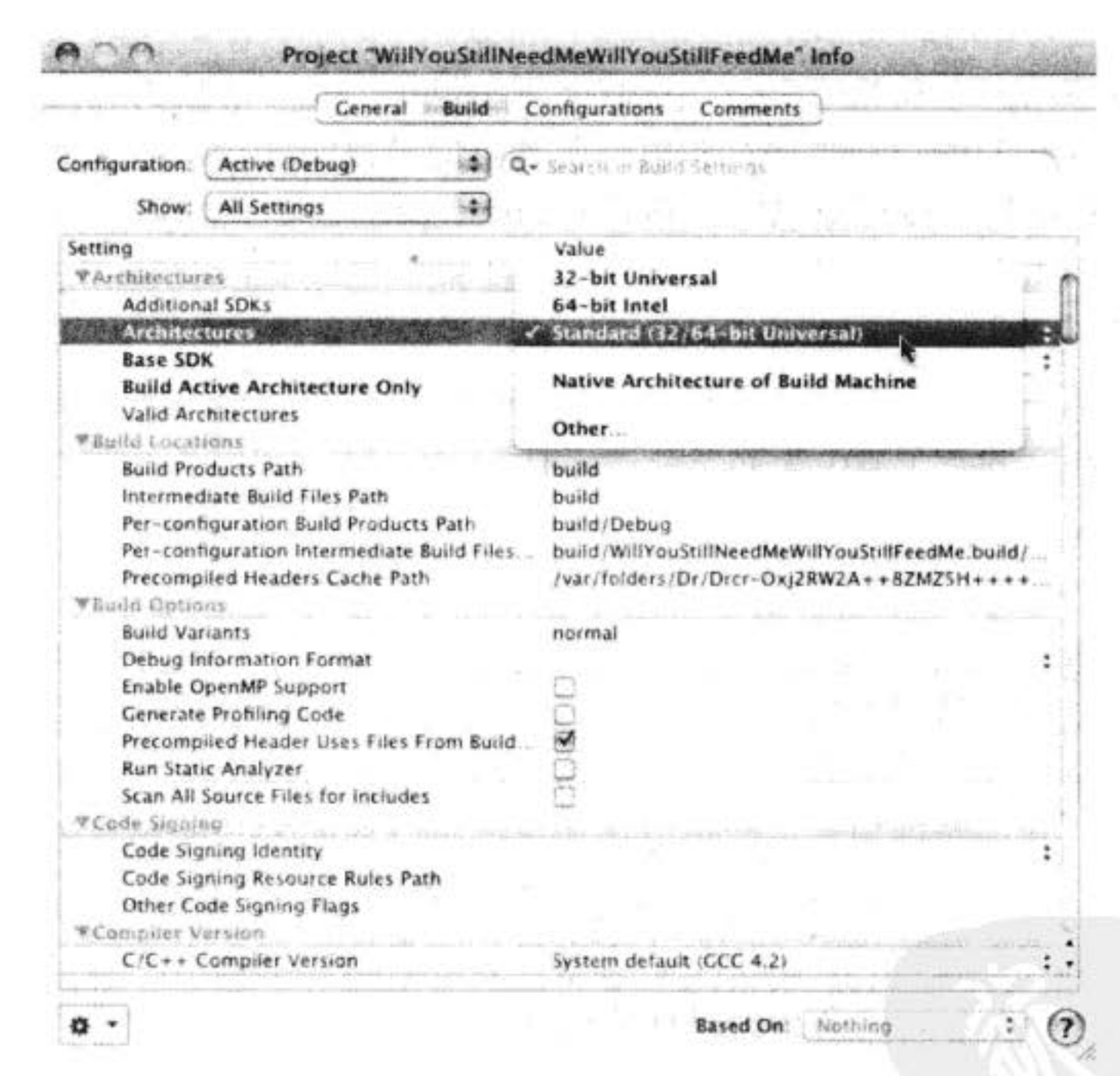
如果指定了多个-arch标志，gcc将构建一个通用的二进制文件，它包含针对你指定的每种体系结构的一个可执行文件。例如，如下的一行命令将program.m编译为一个使用3种方式的通用二进制文件（也叫做胖二进制文件），分别用于32位PowerPC、32位Intel和64位Intel的可

执行文件：

```
gcc -arch ppc -arch i386 -arch x86_64 -arch ppc program.m -framework Foundation
```

通用二进制文件允许将一个程序编译为64位，以用于最新的Intel机器，但是，它保留了对于较旧的Intel或PowerPC机器的用户的兼容性。通用二进制文件的缺点是，编译的时间较长（编译器必须重复构建程序数次），并且，可执行文件的大小也要大上好几倍（一个通用二进制文件，是由针对每种体系结构的二进制文件组成的，它们融合到一起成为一个单独的文件）。然而，通过一个通用二进制文件执行的程序的内存占用不会太大，因为载入程序只是载入适用于当前环境的可执行文件。

要使用Xcode将一个项目编译成64位可执行文件或一个通用二进制文件，可以在Project设置窗口的Build选项卡中设置体系结构，如图C-1所示。



图C-1 使用Xcode针对64位编译

C.5 更多信息

Apple关于64位Mac OS X的文档位于http://developer.apple.com/mac/library/ documentation/Darwin/Conceptual/64bitPorting/intro/intro.html。特定Cocoa的64位文档位于http:// developer.apple.com/ mac/library/documentation/Cocoa/ Conceptual /Cocoa64BitGuide/ Introduction/Introduction.html。

附录D 运行时，旧的和新的

Apple提供了两种不同版本的Objective-C运行时："传统的"和"现代的"。传统的版本用于32位OS X程序，而现代的版本用于iPhone应用程序和在OS X 10.5及其以后版本上运行的64位应用程序。在编写本书的时候（2010年1月），名称"传统的"还用得有点为时过早，因为现存的主要Mac应用程序都是32位的。

从一个程序员的视角来看，两种运行时之间的主要区别在于，它们如何存储一个对象的实例变量。传统的运行时将实例变量一起存储到一个C结构中，并且，通过它们到结构的偏移量来引用实例变量。现代的运行时使用一种不同的方法，并且通过名称来定位实例变量。这有两种结果。它允许根据声明特性来合成实例变量，并且，这解决了所谓的不稳定的基类的问题。

D.1 合成实例变量

在将要在传统运行时中使用的代码中，如下这样来声明一个特性（见第12章）：

```
@interface Greeter : NSObject
{
  NSString *greetingText;
}
@property (nonatomic, retain) NSString *greetingText;

...

@end
```

如果在对应的实现部分放置了如下的一行代码：

```
@synthesize greetingText;
```

编译器将创建访问器方法greetingText和setGreetingText:。这为你节省了很多录入。但是，如果你真的不愿意费力录入，你可能会抱怨，必须输入NSString*greetingText两次似乎有些多余了，其中一次用在实例变量声明时，一次用于property语句中。

现代运行时避免了这一额外录入。可以省略花括号之间的实例变量声明，并且，编译器会使用property语句中的声明自动为你创建实例变量：

```
@interface Greeter : NSObject
{
}
@property (nonatomic, retain) NSString *greetingText;

...

@end
```

当合成特性greetingText时，编译器创建了两个访问器方法和实例变量。再一次，这只是在现代编译器中有效。如果你试图在传统运行时中编译前面的代码，会得到如下的错误：

```
error: synthesized property 'greetingText' must either be named the same as a
compatible instance variable or must explicitly name an instance variable
```

D.2 合成实例变量和Mac OS X Leopard (v 10.5)

当在Mac OS X Leopard (v 10.5)上使用合成实例变量时，有一点要注意，不能直接访问一个合成实例变量。访问一个合成实例变量的唯一方式是使用合成的访问器方法。这意味着，甚至在init和dealloc方法中，也必须使用访问器：

```
(void) dealloc
{
  self.greetingText = nil;
  [super dealloc];
}
```

在上面的代码中，合成的setter向当前存储在greetingText中的对象发送了一条release消息。它还向nil发送了retain消息，不过这没什么坏处。这一限制也意味着，你不能够为一个合成的实例变量提供定制的访问器（不能为你无法访问的一个实例变量编写一个访问器）。

好在，在Mac OS X Snow Leopard (v 10.6)中，这一问题已经得到解决了。针对10.6上的现代运行时库编译的代码，可以直接访问合成实例变量了。这允许你为合成实例变量创建定制的访问器，以替代编译器为了响应@synthesize指令而创建的一个或两个访问器。也可以在init和dealloc方法中直接设置合成实例变量。

> **合成实例变量和iPhone**
>
> iPhone使用一个32位的现代运行时，因此，可以在iPhone上使用合成实例变量。然而，有一个奇怪的问题。只有当你为一个设备而编译代码时，才能够这么做。iPhone模拟器在传统32位桌面运行时之上运行。合成实例变量不能针对模拟器编译。

D.3 不稳定基类问题——已解决

很多面向对象的语言都饱受所谓的不稳定基类问题的困扰。如果向一个类添加了方法或实例变量，从修改了的类派生出来的任何类，都必须重新编译。对于在自己的应用程序中创建的类来说，这可能没有什么影响。良好的实践方法要求你在释放自己的程序之前进行一次清理编译。但是，对于框架提供者来说，这是个问题。想象一下，如果Apple想要在Cocoa框架类中进行某些修改，每次你都必须重新编译所有的程序。你（或每个其他人）也需要并非自己所创建的任何程序的新副本。人们会对此不爽。

不稳定基类问题的根源就是，实例变量和方法的静态内存布局（在编译时确定）。由于Objective-C的动态方法分配机制，在方法那一部分，它没有遇到不稳定基类问题。你总是可以给一个类添加方法，而不需要重新编译派生自它的子类。甚至可以在运行时给一个类添加方法

（见第11章）。

过去，Objective-C框架提供者使用各种技术来解决这一问题的内存变量部分。例如，在类第一次定义时，添加几个实例变量保留“以供将来使用”。这一方法有缺点，因为现在不知道将来需要什么样的实例变量。如果“以供将来使用”的实例变量没有使用的话，它还会浪费内存。处理这一问题的另一种方法是，提供一个单个的额外的实例变量，它是指向NSMutableDictionary的一个指针。该字典可以用来存储额外的值，而不需要更改类的内存布局。

现代运行时解决了这个问题的实例变量部分，而不需要任何技巧。由于现代运行时定位实例变量的方法，使得合成实例变量可以添加到一个类中，而不需要重新编译从该类派生的那些类。

附录E　Objective-C的资源

E.1　Apple资源

如果你针对Mac OS X或iOS开发，则你应该成为一个注册的Apple Developer(http://developer.apple.com/programs/register/)。注册是免费的。注册为Apple Developer，就可以下载Xcode和iPhone SDK。还允许你访问Mac和iPhone的在线文档、示例代码和Apple的在线bug报告系统。

Apple还有两个付费程序：Mac Developer Program (http://developer. apple.com/programs/mac/)和iPhone Developer Program (http://developer. apple.com/programs/iphone/)。每一个程序一年需要花费99美金。

Mac Developer Program提供了预发布的Mac OS X软件、视频，还可以访问Apple Developer论坛，还有两次技术支持活动（Technical Support Incidents）。Technical Support Incidents为你提供来自Apple's Developer Technical Support团队的工程师的支持。

iPhone Developer Program提供视频、文档，可以访问Apple Developer论坛，还有两次技术支持活动（Technical Support Incidents）。还允许你在一个iOS设备上安装apps并且将apps提交到iPhone App Store上进行销售。

> **注意**　免费的Apple Developer program允许使用iPhone SDK开发应用程序，并且在iPhone模拟器上测试它们。要在iPhone设备上测试应用程序并且在App Store上销售应用程序，必须申请或接受iPhone Developer Program。

Apple还提供了大量的邮件列表。对Objective-C开发者最有用的是，Objective-C、Cocoa和Xcode的邮件列表：

http://lists.apple.com/mailman/listinfo/objc-language
http://lists.apple.com/mailman/listinfo/cocoa-dev
http://lists.apple.com/mailman/listinfo/xcode-users

尽管这并非其官方职责的一部分，但很多Apple工程师会阅读这些列表并发帖。

E.2　Internet资源

Apple Cocoa和Xcode邮件列表以一种方便的形式存档在www.cocoabuilder.com。

Cocoa列表还存档在www.mail-archive.com/cocoa-dev@lists.apple.com/。

Omni Group是一个独立的Mac软件公司，总部在西雅图，它们编写了很多的框架，并且它们都作为一个开源项目可供使用。即便你不直接使用这些框架，看看其示例代码也是很值得的

(www.omnigroup.com/developer/)。

GNUStep提供了AppKit和Foundation框架的一个开源实现，它们都基于OpenStep API及一些基于Cocoa的附加内容。GNUStep在多种系统上工作，包括OS X、Linux和Windows (www.gnustep.org)。

CocoaDev是一个维基站点，带有众多关于Objective-C和Cocoa的教程和文章的链接 (www.cocoadev.com)。

Web上还有很多关于Objective-C的其他文章和帖子，有一些很不错。由于现代世界中充满了如此多的信息，所以Google（或者你喜爱的搜索引擎）会是我们很好的朋友。

E.3 组织

很多城市有按月举行的CocoaHeads会议(http://cocoaheads.org/)。除了演讲，CocoaHeads会议也是向有经验的开发者学习的好机会。www.meetup.com列出了很多与Objective-C相关的聚会，特别是iPhone组织的聚会。查看一下，哪些离你最近。

NSCoder之夜是非正式组织的编码会议，在那里，你可以交换思想并向其他的Objective-C程序员学习(http://nscodernight.com/)。

E.4 图书

Erik M. Buck和Donald A. Yacktman，《Cocoa Design Patterns》(Addison-Wesley, 2009)。

Samuel P. Harbison和Guy L. Steele，《C: A Reference Manual, Fifth Edition》 (Prentice Hall, 2002)。

Aaron Hillegass，《Cocoa Programming for Mac OS X, Third Edition》 (Addison-Wesley, 2008)。

Brian W. Kernighan和Dennis M. Ritchie，《The C Programming Language, Second Edition》 (Prentice Hall, 1988)。

Stephen G. Kochan，《Programming in Objective-C 2.0, Second Edition》 (Addison-Wesley, 2009)。

Amit Singh，《Mac OS X Internals: A Systems Approach》 (Addison-Wesley, 2006)。

苹果开发与应用系列丛书

Cocoa设计模式
作者：Erik M. Buck Donald A. Yacktman
译者：陈宗斌 等
原出版社：Pearson Education USA
ISBN：978-7-111-31740-1
定价：59.00元

内容简介

本书介绍了Cocoa框架中用到的面向对象的设计模式。Cocoa框架是Apple的面向对象开发环境，用来开发 Mac OS X 的应用程序，主要的开发语言为 Objective-C。本书共分为5个部分，按照由浅入深、循序渐进的原则详细介绍各种设计模式在Cocoa中的应用，并且提供了丰富的代码示例，方便读者学习。

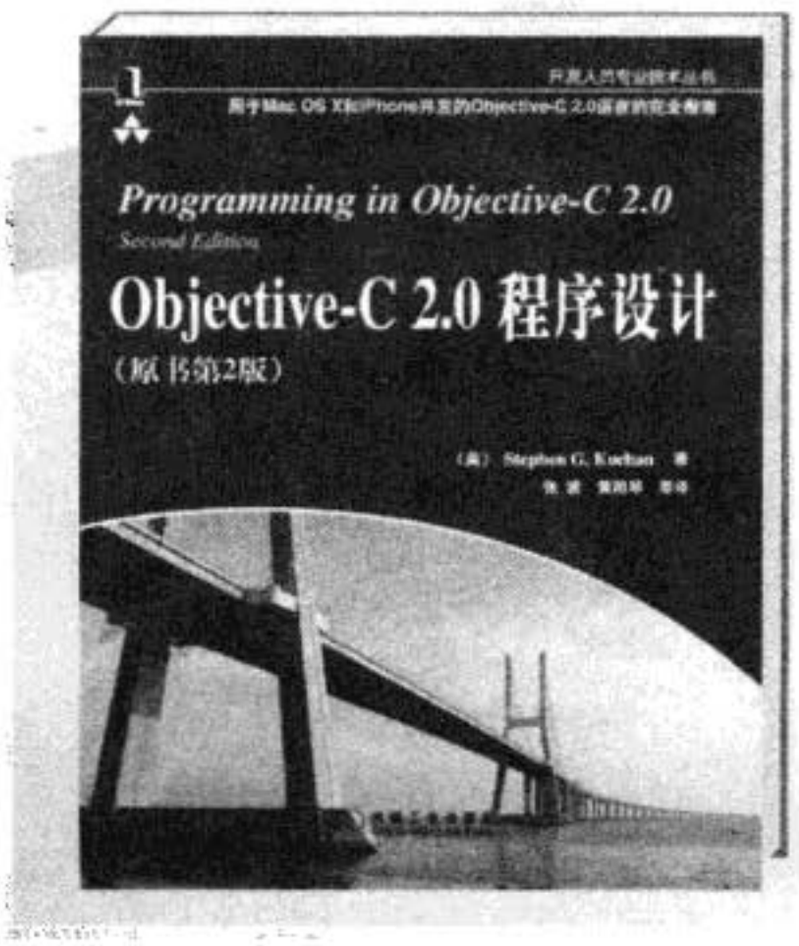
本书适合于正在使用或者考虑使用Mac OS X上的Apple Cocoa框架或者iPhone和iPod touch上的Cocoa Touch框架的读者阅读。另外，Objective-C、C语言、C++和Java程序员都可以从本书中获益。

Objective-C 2.0编程快速上手
作者：Steven Holzner
译者：刘红伟 等
原出版社：Pearson Education USA
ISBN：978-7-111-30874-4
定价：35.00元

内容简介

本书是介绍Objective-C编程的基础教程。全书采用实例讲解、按部就班的方式，全面详细地介绍了Objective-C面向对象编程的基础知识和基本技能。主要内容包括Objective-C的基础知识、流程控制，数据操作，函数、类和对象，继承，分类、扮演和协议，数组和字典等基础内容，以及内存管理和异常处理等相对高级的话题。

全书内容精练、讲解生动，非常适合初学者快速上手，便于掌握Objective-C编程的知识和技能。



Objective-C 2.0程序设计
作者：Stephen G. Kochan
译者：张波
原出版社：Pearson Education Asia
ISBN：978-7-111-27686-9
定价：66.00元

内容简介

本书提供了Objective-C 2.0语言的完整介绍。全书共分为三大部分和四个附录。第一部分讲述Objective-C语言的基础知识，包括类、对象和方法；数据类型和表达式；程序结构；继承；多态、动态类型和动态绑定；函数、数组、结构和指针等；第二部分介绍了Foundation框架，讲述如何使用Foundation框架的各种预定义类；第三部分介绍了Cocoa和iPhone SDK；附录给出了Objective-C语言的快速参考。

本书结构清晰，简单易学，适合于程序设计初学者或者Objective-C程序员阅读，也可作为大学程序设计语言的入门教材。

专业成就人生
立体服务大众

HZ BOOKS

www.hzbook.com

填写读者调查表　加入华章书友会
获赠精彩技术书　参与活动和抽奖

尊敬的读者：

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的"华章书友会"，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者，免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名：　　书号：7-111-(　　)

姓名：	性别：□男　□女	年龄：	职业：
通信地址：		E-mail：	
电话：	手机：	邮编：	

1. 您是如何获知本书的：

□朋友推荐　□书店　□图书目录　□杂志、报纸、网络等　□其他

2. 您从哪里购买本书：

□新华书店　□计算机专业书店　□网上书店　□其他

3. 您对本书的评价是：

技术内容	□很好	□一般	□较差	□理由_____
文字质量	□很好	□一般	□较差	□理由_____
版式封面	□很好	□一般	□较差	□理由_____
印装质量	□很好	□一般	□较差	□理由_____
图书定价	□太高	□合适	□较低	□理由_____

4. 您希望我们的图书在哪些方面进行改进？

_____

_____

5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。

_____

_____

6. 您有没有写作或翻译技术图书的想法？

□是，我的计划是_____　□否

7. 您希望获取图书信息的形式：

□邮件　□信函　□短信　□其他_____

请寄：北京市西城区百万庄南街1号　机械工业出版社　华章公司　计算机图书策划部收

邮编：100037　电话：（010）88379512　传真：（010）68311602　E-mail: hzjsj@hzbook.com

LEARNING OBJECTIVE-C 2.0

A HANDS-ON GUIDE TO OBJECTIVE-C FOR MAC AND IOS DEVELOPERS

Objective-C 2.0 Mac和iOS开发实践指南

"当刚接触Objective-C的程序员询问该从何处开始学习时，我会推荐这本书给他们。"
—— Matt Long，Cocoa Is My Girlfriend(www.cimgf.com)网站

如果你想要学习Objective-C 2.0以便为Mac OS X、iPhone、iPod touch或iPad编写程序，本书是你的最佳选择。本书内容精练、可读性强、易于学习，是学习Objective-C 2.0的最佳入门指南。

作为资深的Mac OS X和iPhone开发者，作者从Objective-C 2.0的最新的基础知识开始深入介绍了所有内容。本书首先有效地回顾了C和面向对象编程的基础知识，然后介绍如何用Xcode创建第一个Objective-C 2.0程序。接下来介绍每种核心语言功能，从对象和类到消息、框架和协议。每一个概念都用简单的示例来阐明，很多章后都附带有动手实践的练习，方便检验学习效果。

本书主要内容

- 理解方法、消息和Objective-C消息系统
- 定义类、创建对象实例和使用类对象
- 使用分类扩展类而不需要子类化
- 使用Objective-C 2.0的声明特性简化开发
- 使用协议而不是类来强调行为
- 针对字符串、数组、字典、集合和数字对象使用常见的Foundation类
- 使用Objective-C控制结构，包括Objective-C 2.0的最新的快速枚举构造
- 理解应用程序安全以及隐藏应该保持私有的方法声明
- 使用Objective-C 2.0提供的最新的块功能

作者简介

Robert Clair拥有20年以上的商业图形软件和CAD软件开发经验。在过去的8年中，他主要从事Mac OS X和iPhone开发。作为Chromatic Bytes公司的负责人，他曾经设计过OS X软件（ZeusDraw和Shades）以及iPhone软件（ZeusDraw Mobile和Orfeo）。他还是多个iPhone和iPad应用程序的顾问。

PEARSON Education

PEARSON

www.pearsonhighered.com

上架指导：计算机 程序设计

ISBN 978-7-111-32927-5

9 787111 329275

定价：55.00元

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

封面设计 · 锡 彬