

提供各种IT类书籍pdf下载，如有需要，请 **QQ: 2011705918**

注：链接至淘宝，不喜者勿入！整理那么多资料也不容易，请多多见谅！非诚勿扰！

更多资源请点击

[美] Stephen G. Kochan 著
林冀 范俊 朱奕欣 译

PEARSON

Broadview
www.broadview.com.cn

Objective-C

程序设计

第6版

iOS与OS X开发者
Objective-C语言
首选入门

Programming in Objective-C Sixth Edition



电子工业出版社
PUBLISHING HOUSE
of ELECTRONICS INDUSTRY



Objective-C

程序设计 第6版

Objective-C已经成为Mac OS X与iOS平台上开发应用程序的标准程序设计语言。这一简单但强大的面向对象语言是以C语言为基础的。Objective-C除了可在Apple的操作系统平台上使用外，也适用于其他的操作系统，包括Linux、UNIX以及Windows。

本书是针对初学者编写的一个完整又循序渐进的Objective-C语言权威入门。阅读本书，不需要先学过C语言或其他面向对象语言。本书包含许多详细且实用的范例，用来告诉你如何在实践中运用Objective-C。

本书第6版的内容已全面更新，覆盖新版iOS、Xcode及OS X Mavericks等所有重要升级要点。

本书完整展示如何利用Foundation framework所提供的丰富自带类库(Library of classes)，并对iOS 程序设计原理有所讲解。

这是我读过的介绍程序语言书中最好的一本，如果你想学Objective-C，切莫犹豫！

——Calvin Wolcott

对于想要以Objective-C作为第一门程序设计语言的初学者来说，这本书是一个非常好的资源。

——Pat Hughes

作者简介

Stephen G. Kochan是多本畅销书的作者或合著者，其中有关于C语言的，包括*Programming in C*、*Programming in ANSI C*和*Topics in C Programming*，也有关于UNIX的，包括*Exploring the UNIX System*和*UNIX Programming*。从1984年Mac最初引进时，他已经在Macintosh计算机上编程了，他编写的*Programming C for the Mac*是Apple Press Library的一部分。2003年，Kochan编写了*Programming in Objective-C*，之后写了另一本与Mac有关的书籍*Beginning AppleScript*。

本书支持网站：<http://classroomm.com/objective-c/>
有关本书之范例代码和练习题解答等，可上网获取相关资源

PEARSON

www.pearson.com



责任编辑：张春雨
封面设计：李玲

PEARSON

上架建议：程序设计>Objective-C

ISBN 978-7-121-23715-7



9 787121 237157 >

定价：89.00元

Objective-C

程序设计

第6版

[美] Stephen G. Kochan 著
林冀 范俊 朱奕欣 译

Programming in Objective-C Sixth Edition

電子工業出版社
Publishing House of Electronics Industry
北京·BEIJING

内 容 简 介

《Objective-C 程序设计》是为在苹果 iOS 和 OS X 平台上,使用 Objective-C 语言和面向对象程序设计模式进行专业开发而编写的简洁、细致的入门读物。本书假设读者无面向对象程序语言或 C 语言编程经验,以保障初学者与有经验的程序员一样,可用本书迅速和有效地学习 Objective-C。本书提供的学习方法独特,结合众多程序示例及章末练习,使得《Objective-C 程序设计(第 6 版)》同时适合自学和课堂教学。第 6 版已全面更新,充分纳入 Objective-C 的新功能与技术,同时覆盖对新版 Xcode、iOS 和 Mac OS X Mavericks 的介绍。

Authorized translation from the English language edition, entitled PROGRAMMING IN OBJECTIVE-C, 6E, 9780321967602 by STEPHEN G. KOCHAN, published by PEARSON EDUCATION, INC., publishing as AddisonWesley Professional, Copyright © 2013 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and PUBLISHING HOUSE OF ELECTRONICS INDUSTRY Copyright © 2014.

本书简体中文版专有出版权由 Pearson Education 培生教育出版亚洲有限公司授予电子工业出版社。未经出版者预先书面许可,不得以任何方式复制或抄袭本书的任何部分。

本书简体中文版贴有 Pearson Education 培生教育出版集团激光防伪标签,无标签者不得销售。

版权贸易合同登记号 图字:01-2014-4179

图书在版编目(CIP)数据

Objective-C 程序设计:第 6 版/(美)寇肯(Kochan,S.G.)著;林冀,范俊,朱奕欣译. —北京:电子工业出版社,2014.7

书名原文:Programming in objective-C(6th edition)

ISBN 978-7-121-23715-7

I. ①O… II. ①寇… ②林… ③范… ④朱… III. ①C 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字(2014)第 147838 号

策划编辑:张春雨

责任编辑:张春雨

印 刷:三河市双峰印刷装订有限公司

装 订:三河市双峰印刷装订有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编:100036

开 本:720×1000 1/16 印张:32.5 字数:572 千字

版 次:2012 年 9 月第 1 版

2014 年 7 月第 2 版

印 次:2014 年 7 月第 1 次印刷

定 价:89.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn,盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线:(010) 88258888。



To Roy and Ve, two people whom I dearly miss.

To Ken Brown, “It’s just a jump to the left.”



作者简介

Stephen G.Kochan 是多本畅销书的作者或合著者，其中有关于 C 语言的，包括 *Programming in C* (Sams, 2004)、*Programming in ANSI C* (Sams, 1994) 和 *Topics in C Programming* (Wiley, 1991)，也有关于 UNIX 的，包括 *Exploring the UNIX System* (Sams, 1992) 和 *UNIX Shell Programming* (Sams, 2003)。从 1984 年 Mac 最初引进时，他就已经在 Macintosh 计算机上编程了，他编写的 *Programming C for the Mac* 是 Apple Press Library 的一部分。2003 年，Kochan 编写了 *Programming in Objective-C* (Sams, 2003)，之后编写了另一本与 Mac 有关的书籍 *Beginning AppleScript* (Wiley, 2004)。

技术审校人员简介

Michael Trent 从 1997 年开始使用 Objective-C 编程，之前在 Mac 上编程。他定期为 Steven Frank 的网站 (www.cocoadev.com) 供稿，为大量的书籍和杂志文章做过技术审校，偶尔也涉足 Mac OS X 开源项目。目前，他正在使用 Objective-C 和苹果计算机的 Cocoa 框架生成 Mac OS X 使用的专业视频应用程序。Michael 拥有比洛特学院（位于威斯康辛州比洛特）的计算机科学学士学位和音乐艺术学位。他与妻子 Angela 居住在加利福尼亚州的圣克拉拉。

Wendy Mui 是旧金山湾区的程序员和软件开发经理。她通过 Steve G.Kochan 的第 2 版书籍学习了 Objective-C，在 Bump Technologies 公司找到了一份工作，将她的编程技能用在客户端应用程序和 Bump 第三方开发者使用的 API/SDK 上。在从事 iOS 开发之前，Wendy 在位于硅谷和加利福尼亚的 Sun 公司及其他科技公司工作。她迷上编程是在加州大学伯克利分校获得数学硕士学位的时候。Wendy 不工作的时候，就在冲击她的跆拳道黑带四段。

预览目录

1 引言	1
第一部分 Objective-C 语言	
2 Objective-C 编程	9
3 类、对象和方法	29
4 数据类型和表达式	53
5 循环结构	73
6 选择结构	95
7 类	129
8 继承	155
9 多态、动态类型和动态绑定	181
10 变量和数据类型	199
11 分类和协议	223
12 预处理程序	239
13 基本的 C 语言特性	253
第二部分 Foundation 框架	
14 Foundation 框架简介	311
15 数字、字符串和集合	315
16 使用文件	379
17 内存管理和自动引用计数	409
18 复制对象	423
19 归档	435
第三部分 Cocoa、Cocoa Touch 与 iOS 的 SDK	
20 Cocoa 和 Cocoa Touch 简介	453
21 编写 iOS 应用程序	457
附录 A 术语表	489
附录 B 地址簿示例源代码	501

目录

1 引言	1
1.1 本书内容	2
1.2 本书组织方式	3
1.3 支持	5
1.4 致谢	6
1.5 第 6 版前言	7
第一部分 Objective-C 语言	
2 Objective-C 编程	9
2.1 编译并运行程序	9
2.1.1 使用 Xcode	10
2.1.2 使用 Terminal	17
2.2 解释第一个程序	20
2.3 显示变量的值	24
2.4 小结	26
2.5 练习	27
3 类、对象和方法	29
3.1 到底什么是对象	29
3.2 实例和方法	30
3.3 用于处理分数的 Objective-C 类	32
3.4 @interface 部分	35
3.4.1 选择名称	35
3.4.2 类方法和实例方法	37

3.5	@implementation 部分	39
3.6	program 部分	41
3.7	实例变量的访问及数据封装	47
3.8	小结	51
3.9	练习	51
4	数据类型和表达式	53
4.1	数据类型和常量	53
4.1.1	int 类型	53
4.1.2	float 类型	54
4.1.3	char 类型	54
4.1.4	限定词: long、long long、short、unsigned 及 signed	55
4.1.5	id 类型	56
4.2	算术表达式	57
4.2.1	运算符的优先级	57
4.2.2	整数运算和一元负号运算符	60
4.2.3	模运算符	62
4.2.4	整型值和浮点值的相互转换	63
4.2.5	类型转换运算符	65
4.3	赋值运算符	66
4.4	Calculator 类	67
4.5	练习	70
5	循环结构	73
5.1	for 语句	74
5.1.1	键盘输入	81
5.1.2	嵌套的 for 循环	83
5.1.3	for 循环的变体	85
5.2	while 语句	86
5.3	do 语句	90

5.4	break 语句	92
5.5	continue 语句	92
5.6	小结	93
5.7	练习	93
6	选择结构	95
6.1	if 语句	95
6.1.1	if-else 结构	100
6.1.2	复合条件测试	103
6.1.3	嵌套的 if 语句	106
6.1.4	else if 结构	107
6.2	switch 语句	117
6.3	Boolean 变量	120
6.4	条件运算符	125
6.5	练习	127
7	类	129
7.1	分离接口和实现文件	129
7.2	合成存取方法	135
7.3	使用点运算符访问属性	137
7.4	具有多个参数的方法	139
7.4.1	不带参数名的方法	141
7.4.2	关于分数的操作	141
7.5	局部变量	145
7.5.1	方法的参数	146
7.5.2	static 关键字	147
7.6	self 关键字	150
7.7	在方法中分配和返回对象	151
7.7.1	扩展类的定义和接口文件	153
7.8	练习	153

8 继承	155
8.1 一切从根类开始	155
8.1.1 找出正确的方法	159
8.2 通过继承来扩展: 添加新方法	160
8.2.1 Point 类和对象创建	164
8.2.2 @class 指令	165
8.2.3 具有对象的类	169
8.3 覆写方法	173
8.3.1 选择哪种方法	174
8.4 抽象类	177
8.5 练习	178
9 多态、动态类型和动态绑定	181
9.1 多态: 相同的名称, 不同的类	181
9.2 动态绑定和 id 类型	184
9.3 编译时和运行时检查	186
9.4 id 数据类型与静态类型	187
9.4.1 动态类型的参数和返回类型	188
9.5 有关类的问题	189
9.6 使用 @try 处理异常	194
9.7 练习	196
10 变量和数据类型	199
10.1 对象的初始化	199
10.2 作用域回顾	203
10.2.1 关于属性、存取方法和实例变量	203
10.2.2 全局变量	204
10.2.3 静态变量	206
10.3 枚举数据类型	208
10.4 typedef 语句	212

10.5	数据类型转换	213
10.5.1	转换规则	213
10.6	位运算符	215
10.6.1	按位与运算符	216
10.6.2	按位或运算符	217
10.6.3	按位异或运算符	218
10.6.4	一次求反运算符	218
10.6.5	向左移位运算符	220
10.6.6	向右移位运算符	221
10.7	练习	221
11	分类和协议	223
11.1	分类	223
11.2	类的扩展	228
11.2.1	关于分类的注意事项	229
11.3	协议和代理	230
11.3.1	代理	233
11.3.2	非正式协议	234
11.4	合成对象	235
11.5	练习	236
12	预处理程序	239
12.1	#define 语句	239
12.1.1	更高级的定义类型	241
12.2	#import 语句	246
12.3	条件编译	247
12.3.1	#ifdef、#endif、#else 和#ifndef 语句	248
12.3.2	#if 和#elif 预处理程序语句	250
12.3.3	#undef 语句	251
12.4	练习	251

13 基本的 C 语言特性	253
13.1 数组	254
13.1.1 数组元素的初始化	256
13.1.2 字符数组	257
13.1.3 多维数组	258
13.2 函数	260
13.2.1 参数和局部变量	262
13.2.2 函数的返回结果	263
13.2.3 函数、方法和数组	267
13.3 块 (Blocks)	268
13.4 结构	272
13.4.1 结构的初始化	275
13.4.2 结构中的结构	276
13.4.3 关于结构的补充细节	278
13.4.4 不要忘记面向对象编程思想	279
13.5 指针	279
13.5.1 指针和结构	283
13.5.2 指针、方法和函数	285
13.5.3 指针和数组	287
13.5.4 指针运算	297
13.5.5 指针和内存地址	299
13.6 它们不是对象	299
13.7 其他语言特性	300
13.7.1 复合字面量	300
13.7.2 goto 语句	301
13.7.3 空语句	301
13.7.4 逗号运算符	301
13.7.5 sizeof 运算符	302
13.7.6 命令行参数	303

13.8 工作原理	305
13.8.1 事实#1: 实例变量存储在结构中	306
13.8.2 事实#2: 对象变量实际上是指针	306
13.8.3 事实#3: 方法是函数, 而消息表达式是函数调用	307
13.8.4 事实#4: id 类型是通用指针类型	307
13.9 练习	307
第二部分 Foundation 框架	
14 Foundation 框架简介	311
14.1 Foundation 文档	311
15 数字、字符串和集合	315
15.1 数字对象	315
15.2 字符串对象	321
15.2.1 NSLog 函数	321
15.2.2 description 方法	322
15.2.3 可变对象与不可变对象	323
15.2.4 可变字符串	330
15.3 数组对象	336
15.3.1 制作地址簿	341
15.3.2 数组排序	358
15.4 词典对象	364
15.4.1 枚举词典	366
15.5 集合对象	369
15.5.1 NSMutableIndexSet	373
15.6 练习	375
16 使用文件	379
16.1 管理文件和目录: NSFileManager	380
16.1.1 使用 NSData 类	385

16.1.2	使用目录	386
16.1.3	枚举目录中的内容	389
16.2	使用路径: <code>NSPathUtilities.h</code>	391
16.2.1	常用的路径处理方法	394
16.2.2	复制文件和使用 <code>NSProcessInfo</code> 类	396
16.3	基本的文件操作: <code>NSFileHandle</code>	400
16.4	<code>NSURL</code> 类	405
16.5	<code>NSBundle</code> 类	406
16.6	练习	407
17	内存管理和自动引用计数	409
17.1	自动垃圾收集	411
17.2	手工管理内存计数	411
17.2.1	对象引用和自动释放池	412
17.3	事件循环和内存分配	414
17.4	手工内存管理规则的总结	416
17.5	自动引用计数 (ARC)	417
17.6	强变量	418
17.7	弱变量	419
17.8	<code>@autoreleasepool</code> 块	420
17.9	方法名和非 ARC 编译代码	420
18	复制对象	423
18.1	<code>copy</code> 和 <code>mutableCopy</code> 方法	424
18.2	浅复制与深复制	426
18.3	实现 <code><NSCopying></code> 协议	428
18.4	用设值方法和取值方法复制对象	431
18.5	练习	433
19	归档	435
19.1	使用 XML 属性列表进行归档	435

19.2	使用 NSKeyedArchiver 归档	437
19.3	编码方法和解码方法	439
19.4	使用 NSData 创建自定义档案	446
19.5	使用归档程序复制对象	450
19.6	练习	451

第三部分 Cocoa、Cocoa Touch 与 iOS 的 SDK

20	Cocoa 和 Cocoa Touch 简介	453
20.1	框架层	453
20.2	Cocoa Touch	454
21	编写 iOS 应用程序	457
21.1	iOS SDK	457
21.2	第一个 iPhone 应用程序	457
21.2.1	创建新的 iPhone 应用程序项目	460
21.2.2	输入代码	463
21.2.3	设计界面	466
21.3	iPhone 分数计算器	474
21.3.1	启动新的 Fraction_Calculator 项目	475
21.3.2	定义视图控制器	476
21.3.3	Fraction 类	481
21.3.4	处理分数的 Calculator 类	485
21.3.5	设计 UI	486
21.4	小结	487
21.5	练习	488
附录 A	术语表	489
附录 B	地址簿示例源代码	501

C 程序设计语言是由 AT&T 贝尔实验室的 Dennis Ritchie 于 20 世纪 70 年代早期首创的。但是，直到 20 世纪 70 年代晚期，这种程序设计语言才获得了广泛的支持并流行开来。因为在此之前，C 编译器还不能用于贝尔实验室以外的商业用途。最初，UNIX 操作系统有可能更快的普及速度也在某种程度上促进了 C 语言的快速普及，UNIX 操作系统几乎完全是由 C 语言编写的。

Brad J.Cox 在 20 世纪 80 年代早期设计了 Objective-C 语言，它以一种叫作 SmallTalk-80 的语言为基础。Objective-C 在 C 语言的基础上加了一层，这意味着对 C 进行了扩展，从而创造出一门新的程序设计语言，支持对象的创建和操作。

1988 年，NeXT 计算机公司获得了 Objective-C 语言的授权，并发展了 Objective-C 的语言库和一个开发环境，即 NEXTSTEP。1992 年，自由软件基金会的 GNU 开发环境增加了对 Objective-C 的支持。所有 Free Software Foundation (FSF) 产品的版权归 FSF 所有。它以 GNU 通用公共许可证来发布产品。

1994 年，NeXT 计算机公司^①和 Sun 公司联合发布了一个针对 NEXTSTEP 系统的标准规范，名为 OPENSTEP。OPENSTEP 在 FSF's 的实现名称为 GNUStep。有一个 Linux 版本，它包括 Linux 内核和 GNUStep 开发环境，这个 Linux 发行版被十分贴切地命名为 LinuxSTEP。

1996 年 12 月 20 日，苹果公司宣布收购 NeXT 软件公司，NEXTSTEP/OPENSTEP 环境将成为苹果操作系统下一个主要发行版本 OS X 的基础。这个开发环境的版本被苹果公司称为 Cocoa。它内置了对 Objective-C 语言的支持，

① NeXT 计算机公司在 1994 年更名为 NeXT 软件公司。——译者注

并结合了 Project Builder（或它的后继版本 Xcode）和 Interface Builder 等开发工具，苹果公司为 Mac OS X 上的应用程序开发创建了一个功能强大的开发环境。

2007 年，苹果公司发布了 Objective-C 语言的升级版，并称为 Objective-C 2.0。本书第 2 版已涵盖了该版本语言的内容。

当 iPhone 于 2007 年发布时，开发人员要求为这款革新性的设备开发应用程序。起初，苹果公司不欢迎第三方应用程序开发。苹果公司安抚那些超级崇拜 iPhone 的开发人员的办法，就是允许他们开发基于 Web 的应用。这些基于 Web 的应用在 iPhone 内置的 Safari Web 浏览器下运行，但需要用户连接到托管该应用程序的网站。基于 Web 应用的很多固有限制，开发人员对此非常不满，于是苹果公司不久之后就宣布，开发人员能够为 iPhone 开发所谓的本机应用。

本机应用是驻留在 iPhone 上并且在 iPhone 操作系统下运行的应用，其运行方式与该设备上运行的内置 iPhone 应用（如 Contacts、Stocks 和 Weather）相同。iPhone 的操作系统实际上是某个 OS X 版本，这意味着可以在 MacBook Pro 上开发并调试这些应用。实际上，苹果公司很快就提供了强大的软件开发套件（SDK），允许快速开发 iPhone 应用并进行调试。iPhone 模拟器使得开发人员能够直接在开发系统上调试其应用，而无须在真机设备上进行。

随着 2010 年 iPad 的推出，苹果公司开始统一操作系统上使用的术语和 SDK，以支持使用不同尺寸的物理屏幕和屏幕分辨率的各种设备。在本书写作时已能够通过 iOS SDK 为各种 iOS 设备开发应用程序，当前发布的操作系统版本为 iOS 7。

1.1 本书内容

在计划编写这本有关 Objective-C 的图书时，我必须做出一些基本的决策。和其他介绍 Objective-C 的内容一样，可以假定读者已经知道如何编写 C 语言程序。可以从使用丰富的例程库（例如，Foundation 框架和 UIKit 框架）的角度介绍这门语言。介绍如何使用一些开发工具（如 Mac 的 Xcode 和设计 UI 使用的 Interface Builder 等）。

但是采用这种方式有一些问题：首先，学习 Objective-C 之前必须完整地学习 C 语言，这种说法是错误的。C 语言是一门过程性的编程语言，有很多特性

是在使用 Objective-C 进行程序设计时不必要的，特别是对于初学者。事实上，采用其中的某些特性违反了坚持良好的面向对象的程序设计方法的本质。同样，在学习面向对象的编程语言之前，最好不要了解过程性语言的所有细节。这会导致程序员误入歧途，并在养成良好的面向对象的程序设计风格方面造成错误的导向和思维定式。Objective-C 是 C 语言的扩展，但这并不意味着必须首先学习 C 语言。

因此，我既不首先讲述 C 语言，也不事先假定你具备了该语言的知识。相反，我决定采用一种非常规的方式，从面向对象编程的视角出发，将 Objective-C 和基础的 C 语言作为一门单独的集成语言来讲解。顾名思义，本书的目的是教你如何使用 Objective-C 进行程序设计。这并不表示我会详细介绍可用于开发和调试程序的工具，或者讲解如何开发交互式图形应用。学会如何使用 Objective-C 编写程序后，所有的这些资料都可在其他地方获得。事实上，你会发现在具备了如何使用 Objective-C 进行程序设计的坚实基础后，掌握这些知识是轻而易举的。本书并不假设读者需要编程经验，即使有，也不会很多。如果你是一名程序设计的初学者，通过一些努力，你应该可以将 Objective-C 作为第一门程序设计语言。根据本书前几版的反馈，其他读者已经做到了。

本书以示例的方式讲述 Objective-C 语言。在介绍这门语言的每个新特性时，通常会提供一个完整的小例子来阐述这一特性。正如“一图胜千言”一样，一个经过严格筛选的例子也有如此功效。强烈建议运行每个程序，并比较系统中获得的结果与本书中的结果。这么做，你不仅可以学会 Objective-C 语言及其语法，而且还能熟悉编译和运行 Objective-C 程序的过程。

1.2 本书组织方式

本书从逻辑上分为三部分：第一部分（第 1~13 章）是“Objective-C 语言”，介绍该语言的基础知识。第二部分（第 14~19 章）是“Foundation 框架”，讲述如何使用构成 Foundation 框架的种类丰富的预定义。第三部分（第 20、21 章）是“Cocoa、Cocoa Touch 和 iOS SDK”，简要介绍 Cocoa 和 Cocoa Touch 框架，然后逐步演示如何使用 iOS SDK 开发简单的 iOS 应用。

框架就是一组从逻辑上组合在一起的类和例程，它们使开发程序变得更加

容易。使用 Objective-C 进行程序设计时需要的许多能力都来源于大量可用的框架。

第2章“Objective-C 编程”，首先讲述如何使用 Objective-C 编写第一个程序。

因为本书并非主要讲解 Cocoa 或者 iOS 程序设计，所以在第三部分之前没有过多地介绍图形用户界面（GUI），甚至很少提及它。这就需要使用一种方法实现程序输入并产生输出。本书中的大多数例子都是从键盘获得输入，并在一个窗口中产生输出的：如果在命令行中，那么这个窗口是 Terminal 窗口；如果使用 Xcode，那么这个窗口是调试输出窗口。

第3章“类、对象和方法”，介绍了面向对象程序设计的基础。本章引入了一些术语，但数量控制到了最少。另外还介绍了定义类的机制，以及向实例或对象发送消息的方式。教师或者有经验的 Objective-C 程序员将会注意到，本书使用静态类型声明对象。我认为这种方法是学生起步的最好方式，因为编译器能捕捉更多的错误，程序有更强的自文档化（self-documenting）功能，同时还能鼓励新程序员显式地声明已知的数据类型。这样，id 类型的概念及其强大功能直到第9章“多态、动态类型和动态绑定”才会完全展现出来。

第4章“数据类型和表达式”，描述了基本的 Objective-C 数据类型，以及如何在程序中使用它们。

第5章“循环结构”，介绍了在程序中可以使用的3种循环语句，即 for、while 和 do。

第6章“选择结构”，详细讲述了 Objective-C 语言的 if 和 switch 语句。判断语句是任何计算机程序设计语言的基础。

第7章“类”，更深入地研究了类和对象的使用，详细讨论了方法、方法的多个参数及局部变量的相关内容。

第8章“继承”，介绍了继承的主要概念。这一特性使得程序更容易开发，因为我们可以利用以前编写的代码，使用继承及子类的概念可以方便地修改和扩展现有的类定义。

第9章讨论了 Objective-C 语言的3个重要特性。多态、动态类型及动态绑定是本章的关键概念。

第10章至第13章对 Objective-C 进行了深入讨论，既包含对象的初始化、

区块、协议、分类、预处理程序，还包括一些基本的 C 语言特性，如函数、数组、结构和指针。第一次开发面向对象的程序时，通常不必（最好避免）使用这些特性。建议你首次通读本书时略过第 13 章，只在需要更多地了解这门语言的特殊特性时，再回来学习它。

第二部分从第 14 章开始，这部分介绍 Foundation 框架，以及如何使用它的大量文档。

第 15 章至第 19 章讲解了 Foundation 框架的重要特性，包括数字和字符串对象、集合、文件系统、内存管理及对象的复制和归档。

学习完第二部分后，你将能够使用 Foundation 框架开发出相当复杂的 Objective-C 程序。

第三部分从第 20 章“Cocoa 和 Cocoa Touch 简介”开始，本章简要介绍了框架，它提供了在 Mac 和 iOS 设备上开发复杂图形应用所需的各种类。

第 21 章介绍了 iOS SDK 和 UIKit 框架。本章阐述了如何以迭代的方式编写简单的 iOS 应用，然后列举了一个计算器应用的示例，通过它可使用 iPhone 进行简单的分数算术运算。

因为面向对象的用语涉及大量术语，所以本书附录 A 提供了一些常用术语的定义。

附录 B“地址簿示例源代码”，给出了本书第二部分中开发并大量使用的两个类的源代码。这些类定义了地址卡和地址簿类，其中的方法提供了一些简单的操作，如在地址簿中添加和删除地址卡、查找某人、列出地址簿的内容等。

学会如何编写 Objective-C 程序后，可以继续向几个不同的方向发展。你可能希望学习有关 C 语言的更多内容，或开始编写在 OS X 上运行的 Cocoa 程序，或者你可能想开发更复杂的 iOS 应用。

1.3 支持

读者可在 classroomM.com/objective-c 论坛查找到更加丰富的内容，可以获取到一些源代码（这里并不能找到所有例子的“官方”源代码，本人坚信学习过程的最大一部分在于自己输入例子中的程序，并且学习如何判断和修正错误）、练习的答案、勘误表和测验题；也可以向我或其他论坛中的成员提问题。

这个论坛有积极的成员，他们乐于帮助其他成员解决疑问，并且答复一些问题，使之变成内容丰富的社区。请加入并参与进来吧！

1.4 致谢

我要感谢在本书第 1 版的准备阶段为我提供帮助的朋友们。首先，要感谢 Tony Iannino 和 Steven Levy 对原稿的审阅，并感谢 Mike Gaines 对本书的贡献。

其次，还要感谢本书的技术编辑 Jack Purdum（第 1 版），Wendy Mui（第 3 版）和 Mike Trent（第 1，2，5，6 版）。我格外地幸运，因为 Mike 审阅了本书的两个版本，他对我编写的本书前两版都进行了最详细的审阅，不仅指出了书中的不足之处，而且还十分慷慨地提出了建议。正是因为 Mike 对第 1 版提供的意见，我改变了介绍内存管理的方法，并尽力确保本书中的每个示例都是“无漏洞的”。在第 4 版之前，内存管理是比较重要的内容，但引入 ARC 后，这部分内容变得过时了。Mike 还为有关 iPhone 程序设计的章节做出了很多贡献。

在第 1 版中，Catherine Babin 提供了封面的图片，他还提供了许多有价值的图片供我选择。朋友为我制作的封面的艺术效果，使得本书具有更强的专业性。

我非常感谢 Pearson 的 Mark Taber（所有的版本），他忍受了我的推迟交稿，并且非常和蔼地让我按照自己的进度工作。同样的荣誉给皮尔逊的 Mandie Frank。Mandie 为我的延迟交付不知疲倦地工作，以帮助本书的各种版本及时出版。我非常感谢 Michael de Haan 和 Wendy Mui 不可思议地主动要求完成校对第 2 版的工作。

注意

正如先前介绍的，Dennis Ritchie 发明了 C 语言，他也是 UNIX 操作系统的共同发明者，这是 OS X 和 iOS 的基础。但遗憾的是，2011 年，在一个星期内这个世界失去了 Dennis Ritchie 和 Steve Jobs。这两位对我的职业生涯有重大的影响。如果没有他们，这本书也就不存在。

最后，我要感谢 classroomM.com/objective-c 论坛的会员对我的反馈、支持和鼓励。

1.5 第 6 版前言

上个版本没有多少变化。由于 Xcode 5 的引入，所有的截图已被更新。我这样做是为了让新手也能跟随当前截图，不至于在学习之前迷茫！有一些语言细节的增加，也已反映在此版中。

Stephen G.Kochan

2013 年 10 月

Objective-C 编程

本章将直接切入正题，并演示如何编写第一个 Objective-C 程序。你现在还不需要使用对象，对象是下一章讨论的主题。希望你能理解本章编写的程序，编译并运行它的各个步骤。

首先，举一个十分简单的例子，在屏幕上显示短语“Programming is fun!”的程序。无须大费周折，代码清单 2-1 显示了用于完成此任务的 Objective-C 程序。

代码清单 2-1

// 第一个程序示例

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

int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSLog(@"Programming is fun!");
    }
    return 0;
}
```

2.1 编译并运行程序

在详细解释这个程序之前，首先要学习编译和运行此程序的步骤。可以使用 Xcode 编译并运行程序，也可以使用 GNU Objective-C 编译器在 Terminal 窗口中编译并运行程序。我们将用这两种方法分别实现这一系列步骤。然后，你可以确定如何处理本书其余部分的程序。

注意

可访问 developer.apple.com 获取最新版本的 Xcode 开发工具，在这里你能够免费获取 Xcode 和 iOS 软件开发包（SDK）。如果你没有注册过开发者账号，首先需要进行免费注册。需要注意的是，Xcode 已经能够从 Mac 应用商店获取。

2.1.1 使用 Xcode

Xcode 是一款功能齐全的应用程序，使用它可轻松输入、编译、调试和执行程序。如果希望在 Mac 机上开发应用程序，学习和使用这个强大的工具是值得的。这里只介绍入门知识，后面会再次回到 Xcode 并逐步介绍如何使用它开发图形应用程序。

注意

Xcode 是一款功能齐全的工具，Xcode 4 引入了更多的功能。但众多的功能容易使你在使用该工具时迷失方向。如果遇到这种情况，试一试阅读 Xcode 用户手册，你能通过访问 Xcode 帮助菜单来找到答案。

Xcode 位于 Developer 文件夹内一个名称为 Applications 的子文件夹中。图 2.1 显示了该工具的图标。



Xcode

图 2.1 Xcode 图标

启动 Xcode（当你第一次启动程序时，必须经历类似于同意许可协议之类的一次性的步骤），你可以从起始页面中选择“Create a new Xcode Projects”（如图 2.2 所示）。

此时出现一个窗口，如图 2.3 所示。

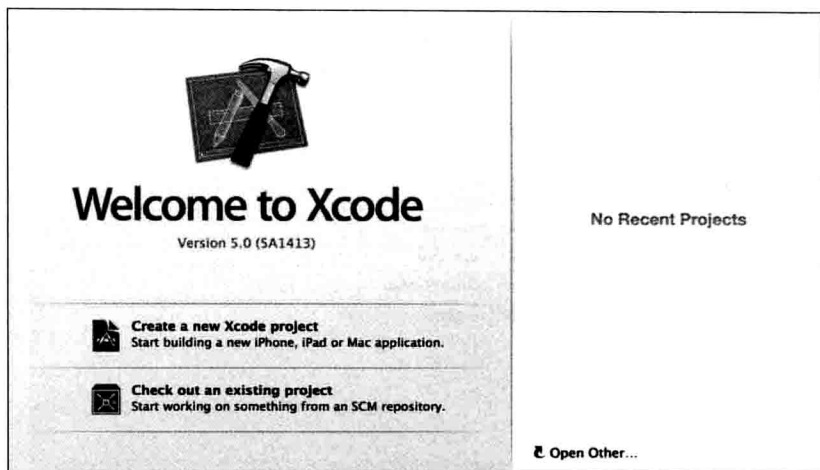


图 2.2 创建一个新项目

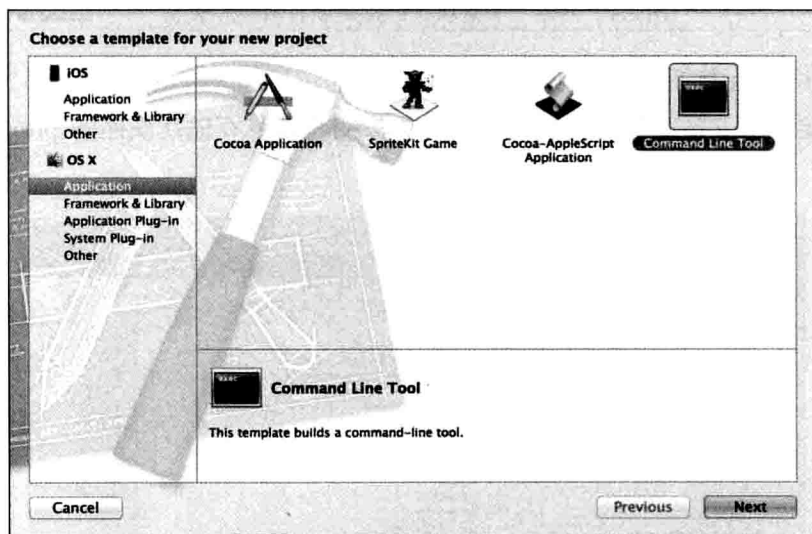


图 2.3 创建一个新项目：选择应用类型

在图 2.3 的左侧面板中，能够看见标记为 OS X 的区域，选择 Application，在右侧面板中选择 Command Line Tool 后，会弹出一个面板，输入应用的名字，如将 prog1 作为产品的名字，在 Company Identifier 和 Bundle Identifier 里随便输入一些字符。后者用来创建 iOS 程序，所以我们现在不必太在意此处输入的是什么。选择类型为 Foundation，如图 2.4 所示。

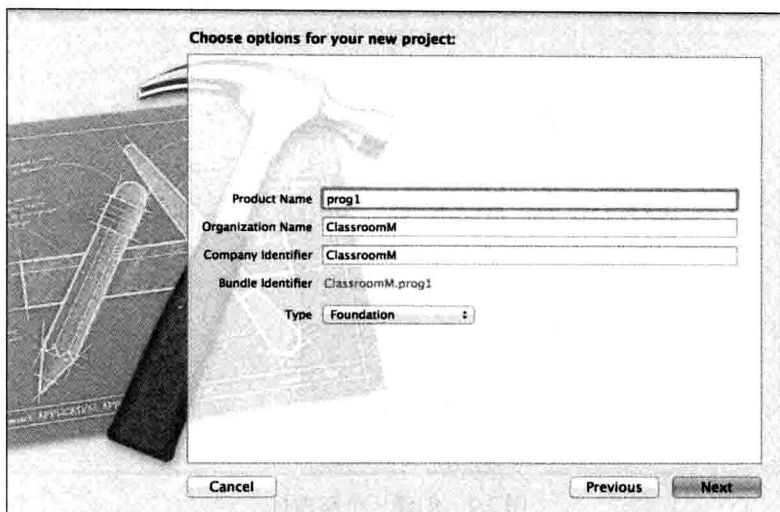


图 2.4 创建一个新项目：指定项目名称和类型

单击 Next 按钮，在出现的表上你能选择与项目相关联的项目目录，同时你能够指定项目文件存储的位置。图 2.5 中表明将项目存储在桌面的 prog1 文件夹中。



图 2.5 选择项目存储路径和项目目录的名称

单击 Create 按钮创建新项目，此时显示一个项目窗口，如图 2.6 所示。注

意，如果你已经使用过 Xcode 或更改了其中的选项，那么显示的窗口可能与此处的稍有不同。该图显示了 Utilities 窗格（最右边的窗格中）。你可以通过取消选择 Xcode 工具栏右上角的视图类中列出的第三个图标来关闭该窗格。请注意，类别默认情况下不标注，为了让其出现在标签上，用鼠标右键单击工具栏，然后选择图标和文本。

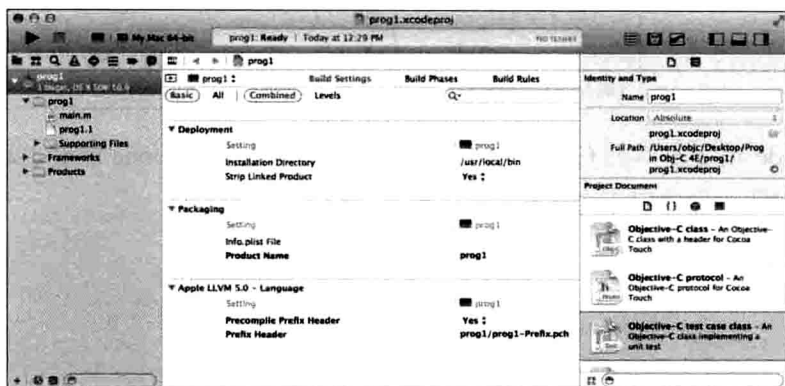


图 2.6 Xcode 项目 prog1 的窗口

现在可以输入第一个程序了。在左侧窗格中选择文件 `main.m`（你可能会需要单击项目名称下的提示三角形 Xcode 才会显示相应的项目文件），此时你的 Xcode 看上去应该和图 2.7 所示的一样。

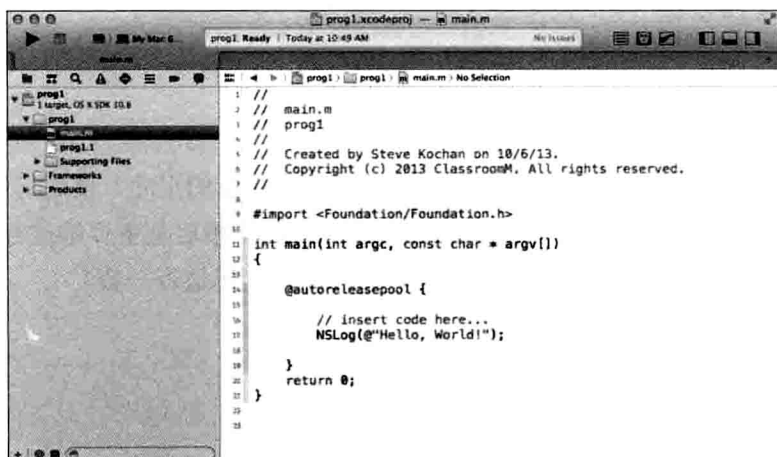


图 2.7 文件 main.m 和编辑窗口

Objective-C 源文件使用.m 作为文件名的最后两个字符（称为扩展名）。表 2.1 列出了常用的文件扩展名。

表 2.1 常见的文件扩展名

扩展名	含 义	扩展名	含 义
.c	C 语言源文件	.mm	Objective-C++源文件
.cc、.cpp	C++语言源文件	.pl	Perl 源文件
.h	头文件	.o	Object（编译后的）文件
.m	Objective-C 源文件		

Xcode 项目窗口的右侧显示了文件 main.m 的内容，Xcode 能够自动创建一个模板文件，包含以下内容：

```
//
// main.m
// prog1
//
// Created by Steve Kochan on 10/16/13.
// Copyright (c) 2013 ClassroomM. All rights reserved.
//
#import <Foundation/Foundation.h>

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

        // 在这里插入代码
        NSLog(@"Hello World!");
    }
    return 0;
}
```

你可在该窗口内编辑文件。修改 Edit 窗口中显示的程序，使之与代码清单 2-1 相符。以两个斜杠字符 (//) 开始的行称为注释，稍后会更详细地介绍注释。

现在 Edit 窗口中的程序应该如下（不必在意注释是否一致）

代码清单 2-1

```
// 第一个程序示例

#import <Foundation/Foundation.h>

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

```

autoreleasepool {
    NSLog(@"Programming is fun!");
}
return 0;
}

```

注意

不要担心屏幕上为文本显示的各种颜色。Xcode 使用不同的颜色指示值、保留字等内容。这种显示方式会提示一些潜在错误，这对刚开始编程的人来说很有帮助。

现在可以用 Xcode 的术语编译并运行第一个程序了，即构建并运行。在这之前，通过单击视图工具栏中间的图标打开一个窗格，用于显示程序的运行结果（输出）。将鼠标置于这个图标上会显示“Hide or show the Debug area.”字样。窗口如图 2.8 所示，在有数据写入调试区域时，Xcode 一般会自动显示调试区域。



图 2.8 Xcode 调试区域显示

现在，如果你单击位于顶部工具栏左侧的 Play 按钮或者在菜单 Product 的窗格中选择 Run，Xcode 将会分别执行编译程序的过程和运行程序的过程。当你的程序没有错误时，运行程序的过程才会执行。

注意

请不用担心屏幕上为你的文本显示的颜色。Xcode 为标识值、保留字使用了不同的颜色。当你开始编程时，它可以标示出潜在的错误，这将非常有价值。

如果在你的程序中出现错误，会出现含有惊叹号的红色终止记号表示的错误——这称为严重错误，只有修正了这些错误，才能运行程序。如果出现含有惊叹号的黄色三角形表示的警告——在消除这些警告之前，尽管你仍然能够运行程序，但通常需要检查并修正这些问题。修正所有的错误，运行程序之后，右下侧区域会显示程序的输出，如图 2.9 所示。

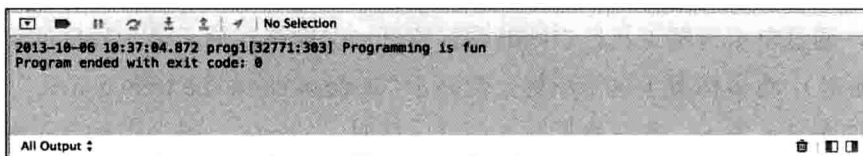


图 2.9 Xcode 调试输出

使用 Xcode 编译并运行第一个程序，这一过程已经完成了。下面总结一下使用 Xcode 创建新程序的步骤。

(1) 启动 Xcode 应用程序。

(2) 如果开发新项目，选择 **File**→**New**→**Project...**，或者在起始页面选择“Create a new Xcode Project”选项。

(3) 对于应用程序类型，选择 **Application**→**Command Line Tool**，然后单击 **Next** 按钮。

(4) 为应用程序取一个名称，在其他字段填入内容。单击 **Next** 按钮。

(5) 选择项目目录的名称，还可以选择在那个目录中存储项目文件，然后单击 **Create** 按钮。

(6) 在左上窗格中会看到文件 **main.m**（在与项目名同名的文件夹下可以找到这个文件），突出显示该文件。在该窗口下面的编辑窗口中输入你的程序。

(7) 在右上方的工具栏中，选择位于 **View** 中间的按钮，将显示调试区域，在这个区域将会显示一些输出。

(8) 在工具栏中单击 **Play** 按钮，或者从 **Product** 菜单中选择 **Run**，编译并运行程序。

注意

Xcode 内建一个强劲的静态代码分析器，它能够分析你的代码并发现一些逻辑错误。在 Product 菜单中选择 Analyze，或者在工具栏中单击 Run 按钮，就能使用分析功能。

(9) 如果出现任何编译错误或者输出与预期的不一致，这时就需要对程序做一些修改，并重新编译和运行程序。

2.1.2 使用 Terminal

有些人可能不想从学习使用 Xcode 开始 Objective-C 程序设计之旅。如果习惯于使 UNIX shell 和命令行工具，可能会用 Terminal 应用来编辑、编译和运行程序。下面说明如何使用 Terminal 执行上述操作。

在试图通过命令行编译你的程序前，请确保你已经在系统上安装了 Xcode 的命令行工具。进入 Xcode 中，Preferences, Downloads, Components。你会看到类似图 2.10 的东西。这个状态表明，该命令行工具没有安装此系统上，此时一个安装按钮将会显示，你可以单击安装打着钩的工具。

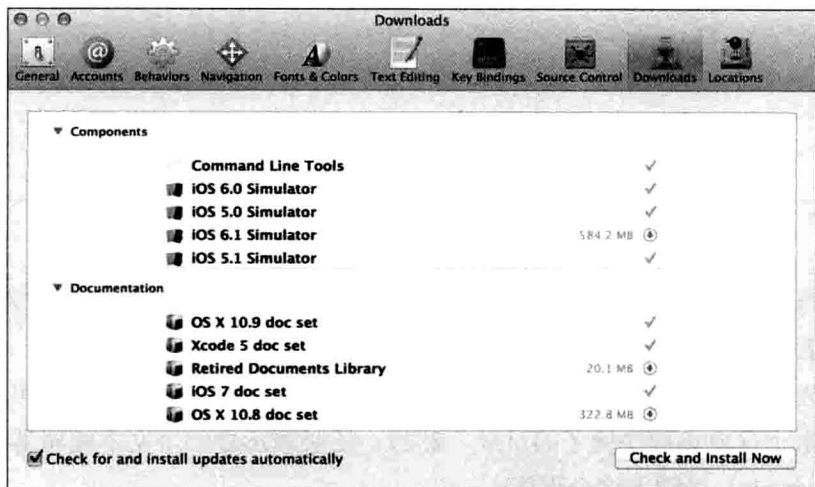
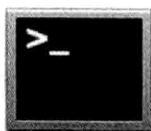


图 2.10 安装命令行工具

当命令行工具安装完毕后，启动 Mac 机器上的 Terminal 应用程序。Terminal 应用程序位于 Application 文件夹下，保存在 Utilities 中，其图标如图 2.11 所示。



Terminal

图 2.11 Terminal 程序图标

启动 Terminal 应用程序后，将会看到一个类似于图 2.12 的窗口。

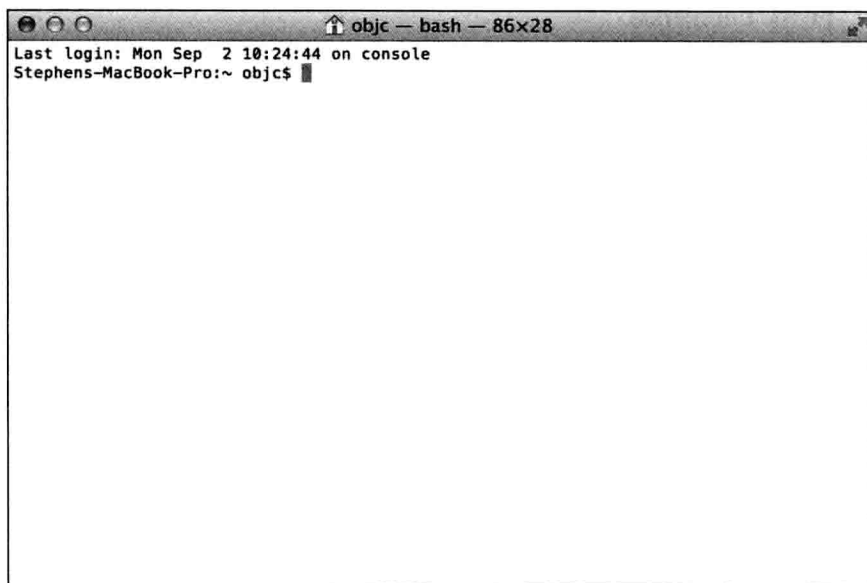


图 2.12 Terminal 窗口

在每一行的\$（或者%，这取决于 Terminal 应用程序的配置）之后输入命令。如果读者熟悉 UNIX 的使用，这是很容易理解的。

首先，需要将代码清单 2-1 中的程序行输入到一个文件中。开始可以先创建一个用于存放该程序例子的目录。然后，还必须运行一个文本编辑器（例如 vi 或者 emacs）来输入这个程序：

```
sh-2.05a$ mkdir Progs      Create a directory to store programs in
sh-2.05a$ cd Progs         Change to the new directory
sh-2.05a$ vi main.m        Start up a text editor to enter program
```

--

注意

在前一个例子及本书其他部分中，由用户输入的命令都用黑体表示。

对于 Objective-C 文件，可以选择使用任何名称，只要确保最后两个字符是 `.m` 即可。这样编译器就知道有一个 Objective-C 程序。

将程序输入到文件中（这里暂不演示输入和保存文件的一些命令）并确定你已经安装了正确的工具之后，可以使用名为 `clang` 的 LLVM Clang Objective-C 编译器来编译并链接这个程序。`clang` 命令的一般格式为：

```
clang -fobjc-arc files -o program
```

`files` 表示要编译的文件列表。在这个的例子中，这样的文件只有一个，我们将其命名为 `main.m`。如果编译时没有任何错误，那么包含这个可执行文件的文件名将是 `program`。

我们把这个程序命名为 `prog1`，下面是用于编译第一个 Objective-C 程序的命令行：

```
$ clang -fobjc-arc main.m -o prog1 Compile main.m & call it prog1
$
```

命令提示符在返回时没有附带任何消息，这意味着编译器没有在程序中发现错误。现在，可以在命令提示符后输入名称 `prog1` 来继续执行这个程序：

```
$ prog1 Execute prog1
sh: prog1: command not found
$
```

除非以前使用过 **Terminal**，否则，很可能得到上面的结果。导致这种情况的原因是：UNIX shell（即运行该程序的应用程序）并不知道 `prog1` 位于何处（这里我们不会讲述所有的细节），所以此时有两种选择：其一，将字符 `./` 放在程序名之前，以便告知 shell 在当前目录查找要执行的程序。其二，将用于存放程序的目录（或者只是当前目录）添加到 shell 的 `PATH` 变量中。此处采用第一种方法：

```
$ ./prog1 Execute prog1
2012-09-03 18:48:44.210 prog1[7985:10b] Programming is fun!
$
```

注意，通过 **Terminal** 编写和调试 Objective-C 程序是一个有效的方法。但是，

这并不是很好的长期策略。如果想构建 OS X 或 iOS 应用程序，除了可执行文件，还有很多文件都需要“打包”到应用程序软件包中。在 Terminal 中执行这些操作并不容易，而 Xcode 则对此很擅长。因此，在此建议你首先学习如何使用 Xcode 开发程序。这样学起来虽然有一定的难度，但所有的努力最终会证明这是值得的。

2.2 解释第一个程序

现在，你熟悉了编译并运行 Objective-C 程序的各个步骤，我们将更详细地讲述这个程序。下面再次给出该程序。

```
// 第一个程序示例
//

#import <Foundation/Foundation.h>

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

        NSLog(@"Programming is fun!");

    }
    return 0;
}
```

在 Objective-C 中，小写字母和大写字母是有区分的。同样，Objective-C 并不关心你在程序行的何处开始输入——程序行的任何位置都能输入语句。利用这个事实，可以开发出容易阅读的程序。

程序的第一行引入了注释的概念。程序中使用的注释语句为程序提供说明，并增强程序的可读性。注释负责告诉该程序的读者，不管他是程序员还是其他负责维护该程序的人，这只是程序员在编写特定程序和特定语句序列时的想法。

向 Objective-C 程序中插入注释有两种方式。其一，使用两个连续的斜杠（//）。双斜杠后直到这行结尾的任何字符都将被编译器忽略。

其二，注释还能以/和*两个字符开头，表示注释的开始，但必须终止这种注释。要终止注释，需要再次使用*和/字符，而且中间不能插入任何空格。开始/*和结束*/之间的所有字符都被看作是注释语句的一部分，从而被 Objective-C

编译器忽略。注释跨越很多程序行时，通常使用这种注释格式，例如下面的注释：

```
/*
This file implements a class called Fraction, which
represents fractional numbers. Methods allow manipulation of
fractions, such as addition, subtraction, etc.

For more information, consult the document:
/usr/docs/classes/Fractions.pdf
*/
```

使用哪种风格的注释完全由你决定。只要注意/*风格的注释不能嵌套使用即可。

在编写程序或者将其输入到计算机上时，应该养成在程序中插入注释的习惯。这种习惯有3个好处：首先，当特殊的程序逻辑在你的大脑中出现时就说明程序，要比程序完成后再回来重新思考这个逻辑简单得多。其次，通过在工作早期阶段把注释插入程序中，可在调试阶段隔离和调试程序逻辑错误时受益匪浅。最后，注释不仅可以帮助你（或者其他入）通读程序，而且还有助于指出逻辑错误的根源。然而我还没有发现很喜欢为程序写注释的程序员。事实上，在调试完程序后，你很可能不喜欢再返回程序来插入注释。在开发程序的过程中插入注释会使这项乏味的工作变得稍微容易。

代码清单 2-1 的下一行程序告诉编译器找到并处理名为 Foundation.h 的文件：

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

这是一个系统文件，也就是说，这个文件不是你创建的。#import 表示将该文件的信息导入或包含到程序中，就像在这里输入该文件的内容。要导入文件 Foundation.h，因为它包含其他类和函数的有关信息，在程序后面的部分中会用到。

在代码清单 2-1 中，这一行指定程序的名称为 main：

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

main 是一个特殊名称，用于准确地表示程序将在何处开始执行。main 之前的保留字 int 指定 main 返回的值类型，该值为整型（后面将更加详细地讨论这个话题）。我们将暂时忽略出现在圆括号开始和结束之间的内容，它们与名为命

令行参数的内容有关，我们将在第 13 章中讨论。

你已经向系统标明了 `main`，接下来就可以准确地指定这个函数要执行哪些工作。只要将函数的所有程序语句放入一对花括号中就可以了。在最简单的情况下，一条语句就是一个以分号结束的表达式。系统将位于花括号之间的所有程序语句看作是 `main` 函数的组成部分。

在 `main` 中下一行语句如下：

```
@autoreleasepool {
```

“{”和与之匹配的“}”之间的程序语句会在被称为“自动释放池 (autorelease pool)”的语境中执行。自动释放池的机制是：它使得应用在创建新对象时，系统能够有效地管理应用所使用的内存。相关内容在第 17 章“内存管理和自动引用计数”中会详细介绍。这里，只有一个语句在 `@autoreleasepool` 语境中。

下一条语句指定要调用名为 `NSLog` 的函数。传递给 `NSLog` 函数的参数是以下字符串：

```
@ "Programming is fun!"
```

此处的 `@` 符号在位于一对双引号的字符串前面，这称为常量 `NSString` 对象。

注意

如果你有一定的 C 程序设计经验，可能会被前面的 `@` 字符所迷惑。如果没有前面的 `@` 字符，就是在编写常量 C 类型的字符串；有了这个符号，就是在编写 `NSString` 字符串对象。更多内容将在第 15 章介绍，“数字、字符串和集合”。

`NSLog` 是一个函数，它仅仅显示或记录其参数（或者参数列表，后面将会看到）。但是之前它会显示该函数的执行日期和时间、程序名，以及其他在此不会介绍的数值。在本书后面部分中，我们不会列出 `NSLog` 在输出前插入的这些文本。

Objective-C 的所有程序语句必须使用分号 (;) 结束。这就是为什么 `NSLog` 的右圆括号之后有分号立即出现。

`main` 中最后一条程序语句是：

```
return 0;
```

它表示要终止 `main` 的执行并发送回（或返回）一个状态值 0。按照约定，0 意味着程序正常结束。任何非零值通常表示出现了一些问题，例如，很可能无法找到程序所需的文件。

现在，你将结束第一个程序的讨论，对它进行修改，以便同时显示短语“`And programming in Objective-C is even more fun!`”。这项工作可以通过简单地添加另一个对 `NSLog` 函数的调用来实现，如代码清单 2-2 所示。记住，每个 Objective-C 程序语句必须使用分号结束。需要注意的是，下列程序已经去掉了首行注释。

代码清单 2-2

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

int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSLog(@"Programming is fun!");
        NSLog(@"Programming in Objective-C is even more fun!");
    }
    return 0;
}
```

如果输入代码清单 2-2，然后编译并运行它，你可能期望在终端上看到以下输出（同样，通常不会显示 `NSLog` 在输出前加入的那些文本）。

代码清单 2-2 输出

```
Programming is fun!
Programming in Objective-C is even more fun!
```

在下一个程序示例中将看到，无须为每行输出单独调用 `NSLog` 函数。

首先，让我们看看特殊的两字符序列。反斜杠（\）和字母 `n` 合起来称为换行符。换行符通知系统要准确完成其名称所暗示的工作：转到一个新行。任何要在换行符之后输出的字符随后将出现在显示器的下一行。事实上，换行符非常类似于一台打字机上“回车”的概念。

研究代码清单 2-3 中列出的程序，并在检验输出之前尝试预测结果（不要作弊，开始!）。

代码清单 2-3

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

```
int main (int argc, const char *argv[])
{
    @autoreleasepool {
        NSLog(@"Testing...\n..1\n...2\n....3");
    }
    return 0;
}
```

代码清单 2-3 输出

```
Testing...
..1
...2
....3
```

2.3 显示变量的值

用 NSLog 不仅能显示简单短语，还能显示变量的值及计算结果。代码清单 2-4 使用 NSLog 函数显示两个数（即 50 和 25）相加的结果。

代码清单 2-4

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

int main (int argc, const char *argv[])
{
    @autoreleasepool {
        int sum;

        sum = 50 + 25;
        NSLog(@"The sum of 50 and 25 is %i", sum);
    }

    return 0;
}
```

代码清单 2-4 输出

```
The sum of 50 and 25 is 75
```

main 中自动释放池后面的第一条语句将变量 sum 定义为整型。在应用程序中使用所有的程序变量之前，都必须先定义它们。变量的定义向 Objective-C 编

译器说明程序将如何使用这些变量，编译器需要这些信息来生成正确的指令，以便将值存储到变量中或者从变量中检索值。定义成 `int` 的变量只能用于存储整型值，也就是说，没有小数位的值。整型值的例子有 3、5、-20 和 0。带有小数位的数字，例如 2.14、2.455 和 27.0，称为浮点数，它们是实数。

整型变量 `sum` 用于存储两个整数 50 和 25 相加的结果。我们故意在这个变量定义的下面留一空行，以便在视觉上区分函数的变量定义和程序语句。严格地讲，这是一个风格问题。有时候，在程序中添加单个空白行可使程序的可读性更强。

程序语句

```
sum = 50 + 25;
```

在作用上与在其他大多数程序设计语言中一样：数字 50 和数字 25 相加（如加号所示），并把结果存储（如赋值运算符，或者等号所示）到变量 `sum` 中。

现在，代码清单 2-4 中的 `NSLog` 函数调用的圆括号中有两个参数，这些参数用逗号隔开。`NSLog` 函数的第一个参数总是要显示的字符串。然而，在显示字符串的同时，通常还希望要显示某些程序变量的值。在这个例子中，你希望在显示字符之后，还要显示变量 `sum` 的值：

```
The sum of 50 and 25 is
```

第一个参数中的百分号是一个特殊字符，它可被 `NSLog` 函数识别。紧跟在百分号后的字符指定在这种情况下将要显示的值类型。在前一个程序中，字母 `i` 被 `NSLog` 函数识别，它表示将要显示的是一个整数。

只要 `NSLog` 函数在字符串中发现 `%i` 字符，它都将自动显示函数第二个参数的值。因为 `sum` 是 `NSLog` 的下一个参数，所以，它的值将在显示字符 “The sum of 50 and 25 is” 之后自动显示。

现在，请尝试预测代码清单 2-5 的输出。

代码清单 2-5

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

int main (int argc, const char *argv[])
{
    @autoreleasepool {
        int value1, value2, sum;
```



```
value1 = 50;
value2 = 25;
sum = value1 + value2;

NSLog(@"The sum of %i and %i is %i", value1, value2, sum);

}
return 0;
}
```

代码清单 2-5 输出

The sum of 50 and 25 is 75

main 中的第二条语句定义了 3 个 int 变量，分别名为 value1、value2 和 sum。这条语句可以等价地表示为以下 3 条独立的语句：

```
int value1;
int value2;
int sum;
```

定义这 3 个变量后，程序将 50 赋值给变量 value1，将 25 赋值给变量 value2。然后计算两个变量的和并将结果赋值给变量 sum。

现在，NSLog 函数的调用包含 4 个参数。同样，第一个参数通常叫作格式字符串，它向系统描述其他参数的显示方式。value1 的值将在 The sum of 之后立即显示。类似地，value2 和 sum 两者的值将在适当的位置输出，该位置由格式字符串中后两个%i 字符指定。

2.4 小结

本章是简述性的一章，上述内容包括了使用 Objective-C 开发程序的相关内容。至此，你应该对使用 Objective-C 进行程序设计有了很好的体验，而且应该能够独立地开发小程序。在下一章中，你将开始领略这个功能强大而且灵活的程序设计语言的一些复杂内容。但请先尝试一下能否完成以下练习，确保你理解了本章提出的概念。

2.5 练习

1. 输入并运行本章出现的 5 个程序。将每个程序产生的输出与原文中每个程序后面列的输出进行比较。
2. 编写一个可显示以下文本的程序。

In Objective-C, lowercase letters are significant.
main is where program execution begins.
Open and closed braces enclose program statements in a routine.
All program statements must be terminated by a semicolon.

3. 以下程序输出什么内容？

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        int i;
        i = 1;
        NSLog (@"Testing...");
        NSLog (@"...%i", i);
        NSLog (@"...%i", i + 1);
        NSLog (@"..%i", i + 2);
    }
    return 0;
}
```

4. 编写一个程序，执行 87 减 15 这个操作并显示其结果，同时还要显示一条适当的消息。
5. 找出以下程序中的语法错误。然后输入并运行改正之后的程序，以确保找出了所有的错误。

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

int main (int argc, const char *argv[]);
{
    @autoreleasepool {
        INT sum;
        /* COMPUTE RESULT //
        sum = 25 + 37 - 19
        / DISPLAY RESULTS /
        NSLog (@'The answer is %i' sum);
    }
    return 0;
}
```

6. 以下程序输出什么结果？

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

int main (int argc, const char *argv[])
{
    @autoreleasepool {
        int answer, result;

        answer = 100;
        result = answer - 10;

        NSLog(@"The result is %i\n", result + 5);
    }
    return 0;
}
```

类、对象和方法

本章介绍面向对象程序设计的一些关键概念，并开始使用 Objective-C 中的类。你需要学习少量术语，但我们不会用太正式的形式向你介绍。本章只会讲解一些基本的术语，因为一下讲太多的内容，你可能不容易接受。有关这些术语更精确的定义，可以参见本书附录 A。

3.1 到底什么是对象

对象就是一个物件。面向对象的程序设计可以看成是一个物件和你想对它做的事情。这与 C 语言不同，C 语言通常称为过程性语言。在 C 语言中，通常是先考虑要做什么，然后才关注对象，这几乎总是与面向对象的思考过程相反。

我们举一个日常生活中的例子。假定你有一辆汽车，显然它是一个对象，而且是你拥有的一个对象。你并不是拥有任意一辆汽车，而是一辆特定的汽车，它由一家制造厂制造，可能在底特律，可能在日本，也可能在其他地方。你的汽车拥有一个车辆识别号码（vehicle identification number，VIN），在美国它能唯一标识你的汽车。

在面向对象的用语中，你的汽车是汽车的一个实例。如果继续使用术语，car 就是类的名称，这个实例就是从该类创建的。因此，每制造一辆新汽车，就会创建汽车类的一个新实例，而且汽车的每个实例都称为一个对象。

你的汽车可能是银白色的，内部装饰为黑色，是辆敞篷车或者有金属顶盖，等等。此外，你还用这辆汽车执行特定的操作。例如，驾驶汽车、加油、（可能还会）洗车、接受维修，等等，这些情况在表 3.1 中做了描述。

表 3.1 对象的操作

对 象	使用对象执行的操作
你的汽车	驾驶
	加油
	洗车
	维修

表 3.1 所列的操作可以用你的汽车实现，也可以用其他汽车实现。例如，你姐姐驾驶她的汽车、洗车、加油，等等。

3.2 实例和方法

类的独特存在就是一个实例，对实例执行的操作称为方法。在某些情况下，方法可以应用于类的实例或者类本身。例如，洗车适用于一个实例（事实上，表 3.1 列出的所有方法都将作为实例方法）。找出一家制造厂制造了多少款汽车来适用于类，所以它是一个类方法。

假设你有两辆使用装配线制作的汽车，它们看上去是一样的：都有相同的内部设计、相同的喷漆颜色，等等。它们可能同时启动，但是由于每部汽车是由它各自的主人驾驶的，这就使它们获得了自身独特的特征和属性的改变。例如，一辆汽车可能后来有了刮痕，而另一辆汽车可能行驶了更远的距离。每个实例或对象不仅包含从制造商那里获得的有关原始特性的信息，还包含它的当前特性，这些特性可以动态改变。当你驾驶汽车时，油箱的油渐渐耗尽，汽车越来越脏，轮胎也逐渐磨损。

对象使用方法可以影响对象的状态。如果方法是“给汽车加油”，那么执行这个方法之后，汽车的油箱将会加满。这个方法影响了汽车油箱的状态。

这里的关键概念是：对象是类的独特表示，每个对象都包含一些通常对该对象来说是私有的信息（数据）。方法提供访问和改变这些数据的手段。

Objective-C 采用特定的语法对类和实例应用方法：

```
[ ClassOrInstance method ];
```

在这条语句中，左方括号后要紧跟类的名称或者该类的实例名称，它后面可以是一个或多个空格，空格后面是将要执行的方法。最后，使用右方括号和

分号来终止。请求一个类或实例来执行某个操作时，就是在向它发送一条消息，消息的接收者称为接收者。因此，有另一种方式可以表示前面所描述的一般格式，具体如下：

```
[ receiver message ] ;
```

回顾上一个列表，使用这个新语法为它编写所有的方法。在此之前，你需要获得一辆新车，去制造厂购买一辆，语句如下：

```
yourCar = [Car new];    get a new car
```

向 car 类（消息的接收者）发送一条新消息，请求它卖给你一辆新车，获取到的对象（它代表你的独特汽车）将被存储到变量 yourCar 中。从现在开始，可用 yourCar 引用你的汽车实例，就是你从制造厂买的那辆汽车。

因为你到制造厂购买了一辆新汽车，所以这个新方法可叫作制造厂方法，或者类方法。对新车执行的其他操作都将是实例方法，因为它们应用于你的新车。下面是一些你可能为这辆新车编写的示例消息表达式：

```
[yourCar prep];          get it ready for first-time use
[yourCar drive];         drive your car
[yourCar wash];          wash your car
[yourCar getGas];        put gas in your car if you need it
[yourCar service];       service your car

[yourCar topDown];       if it's a convertible
[yourCar topUp];
currentMileage = [yourCar odometer];
```

最后一个示例显示的实例方法可返回信息，即当前的行驶里程，这通过里程表（odometer）可看出来。我们将该信息存储在程序中的 currentMileage 变量内。

这里有一个将特定值作为方法参数的例子，与方法直接调用有所不同：

```
[yourCar setSpeed: 55]; set the speed to 55 mph
```

你姐姐 Sue 可以对她的汽车实例使用相同的方法：

```
[suesCar drive];
[suesCar wash];
[suesCar getGas];
```

将同一个方法应用于不同的对象是面向对象程序设计背后的主要理念之一，稍后将学到这方面更多的内容。

你可能无须在程序中对汽车进行操作。你的对象很可能是面向计算机的对象，例如窗口、矩形、一段文本，甚至是计算器或歌曲的播放列表。就像作用于汽车的方法，这些方法可能看上去类似于下列语句：

<code>[myWindow erase];</code>	清除窗口
<code>theArea = [myRect area];</code>	计算矩形的面积
<code>[userText spellCheck];</code>	对一些文本进行拼写检查
<code>[deskCalculator clearEntry];</code>	清除最后一项
<code>[favoritePlaylist showSongs];</code>	显示播放列表收藏夹中的歌曲
<code>[phoneNumber dial];</code>	拨打一个电话号码
<code>[myTable reloadData];</code>	在表格中显示更新的数据
<code>n = [aTouch tapCount];</code>	记录显示屏被点击的次数

3.3 用于处理分数的 Objective-C 类

现在，我们将用 Objective-C 定义一个实际的类，并学习如何使用类的实例。

同样，我们将先学习过程。因此，实际的程序范例可能不是特别实用，那些更加实际的内容将在稍后讨论。

假设要编写一个用于处理分数的程序，可能需要处理加、减、乘、除等运算。如果你还不知道什么是类，那么可以从一个简单的程序开始，代码如下：

代码清单 3-1

// 采用分数的简单程序

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

int main (int argc, char * argv[])
{
    @autoreleasepool {
        int numerator = 1;
        int denominator = 3;
        NSLog(@"The fraction is %i/%i", numerator, denominator);
    }
    return 0;
}
```

代码清单 3-1 输出

The fraction is 1/3

在代码清单 3-1 中，分数是以分子和分母的形式表示的。在 `@autoreleasepool`

指令之后，main 中的前两行将变量 `numerator` 和 `denominator` 都声明为整型，并分别给它们赋予初值 1 和 3。这两个程序与下面的程序行等价：

```
int numerator, denominator;
```

```
numerator = 1;
denominator = 3;
```

将 1 存储到变量 `numerator` 中，将 3 存储到变量 `denominator` 中，这样就表示分数 $1/3$ 。如果需要在程序中存储多个分数，这种方法可能比较麻烦。每次要引用分数时，都必须引用相应的分子和分母，而且操作这些分数也相当困难。

如果能把一个分数定义成单个实体。用单个名称（例如 `myFraction`）来共同引用它的分子和分母，那就会更好。这种方法可以利用 Objective-C 来实现，从定义一个新类开始。

代码清单 3-2 通过一个名为 `Fraction` 的新类，重写了代码清单 3-1 中的函数。下面给出这个程序，随后将详细介绍它是如何工作的。

代码清单 3-2

// 使用分数的程序——类版本

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

//---- @interface 部分 ----

@interface Fraction: NSObject

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;

@end

//---- @implementation 部分 ----

@implementation Fraction
{
    int numerator;
    int denominator;
}
-(void) print
{
```



```

    NSLog(@"%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}

@end

//---- program 部分 ----

int main (int argc, char * argv[])
{
    @autoreleasepool {
        Fraction *myFraction;

        // 创建一个分数实例

        myFraction = [Fraction alloc];
        myFraction = [myFraction init];

        // 设置分数为 1/3

        [myFraction setNumerator: 1];
        [myFraction setDenominator: 3];

        // 使用打印方法显示分数

        NSLog(@"The value of myFraction is:");
        [myFraction print];
    }
    return 0;
}

```

代码清单 3-2 输出

```

The value of myFraction is:
1/3

```

从代码清单 3-2 的注释中可以看到，程序在逻辑上分为以下 3 个部分：

- @interface 部分
- @implementation 部分
- program 部分

其中，@interface 部分用于描述类和类的方法；@implementation 部分用于描述数据（类对象的实例变量存储的数据），并实现在接口中声明方法的实际代码；program 部分的程序代码实现了程序的预期目的。

注意

也可以在 interface（接口）部分为类声明实例变量。从 Xcode 4.2 开始，已经可以在 implementation（实现）部分添加实例变量，这是为了能够以一种更好的方式来定义类。在后面章节中说明了原因。

以上 3 个部分存在于每个 Objective-C 程序中，即使你可能不需要自己编写每一部分。你会看到，每一部分通常放在它自己的文件中。然而，目前来说，我们将它们放在一个单独的文件中。

3.4 @interface 部分

定义新类时，首先需要告诉 Objective-C 编译器该类来自何处。也就是说，必须为它的父类命名。其次，还必须定义在处理该类的对象时将要用到的各种操作或方法的类型。在随后的章节中你还会学习到，在@interface 部分中还会列出一些元素，称为属性。这部分的一般格式类似于下列语句：

```
@interface NewClassName: ParentClassName
    propertyAndMethodDeclarations;
@end
```

按照约定，类名以大写字母开头，尽管没有要求这么做。但这种约定能使其他人在阅读你的程序时，仅仅通过观察名称的第一个字母就能把类名和其他变量类型区分开来。让我们暂时转换一下话题，先谈论一些在 Objective-C 中制定名称的有关规则。

3.4.1 选择名称

在第 2 章“Objective-C 编程”中，使用了几个变量来存储整型（int）值。

例如，在代码清单 2-4 中使用变量 `sum` 来存储 50 和 25 两个数相加的结果。

Objective-C 语言还允许变量存储非整型的数据类型，在程序中使用变量之前，只要对它进行适当的声明即可。变量可以用于存储浮点数、字符，甚至是对象（或者更确切地说，是对对象的引用）。

制定名称的规则相当简单：名称必须以字母或下画线（`_`）开头，之后可以是任何（大写或小写）字母、下画线或者 0 到 9 之间的数字组合。下面是一些合法的名称：

- `sum`
- `pieceFlag`
- `i`
- `myLocation`
- `numberOfMoves`
- `sysFlag`
- `ChessBoard`

另外，根据规定，以下名称是非法的：

- `sum$value`——`$`是一个非法字符。
- `piece flag`——名称中间不允许插入空格。
- `3Spencer`——名称不能以数字开头。
- `int`——这是一个保留字。

`int` 不能用作变量名，因为其用途对 Objective-C 编译器而言有特殊含义，这种用法称为保留名或保留字。一般来说，任何对 Objective-C 编译器有特殊意义的名称都不能作为变量名使用。

应该时刻记住，Objective-C 中的大写字母和小写字母是有区别的。因此，变量名 `sum`、`Sum` 和 `SUM` 均表示不同的变量。前面曾提到，在命名类时，类名通常以大写字母开头。另一方面，实例变量、对象以及方法的名称，通常以小写字母开头。为使程序具有可读性，名称中要用大写字母来表示新单词的开头，例如下面的例子：

- `AddressBook`——可能是一个类名。

- `currentEntry`——可能是一个对象。
- `current_entry`——一些程序员还使用下画线作为单词的分隔符。
- `addNewEntry`——可能是一个方法名。

确定名称时，要遵循同样的标准：千万不要偷懒。要找到能反映变量或对象使用意图的名称。原因很明显，就像使用注释语句一样，富有意义的名称可以显著增强程序的可读性，并可在调试和文档编写阶段受益匪浅。事实上，因为程序具有更强的自解释性（self-explanatory），所以编写文档的任务将很可能大大减少。

以下是代码清单 3-2 中的 `@interface` 部分：

```
//---- @interface 部分 ----

@interface Fraction: NSObject

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;

@end
```

新类的名称为 `Fraction`，其父类为 `NSObject`（我们将在第 8 章“继承”中讲解有关父类更详细的内容）。`NSObject` 类在文件 `NSObject.h` 中定义，导入 `Foundation.h` 文件时会在程序中自动包括这个类。

3.4.2 类方法和实例方法

必须定义各种方法才能使用 `Fractions`，且需要将分数的值设为特定的值。因为你不能直接访问分数的内部表示（就是直接访问它的实例变量），因此，必须编写方法来设置分子和分母。还要编写一个名为 `print` 的方法来显示分数的值。下面是 `print` 方法的声明，位于接口文件中：

```
-(void) print;
```

开头的负号（-）通知 Objective-C 编译器，该方法是一个实例方法。此外，只有一种选择，就是正号（+），它表示类方法。类方法是对类本身执行某些操作的方法，例如，创建类的新实例。

实例方法能够对类的实例执行一些操作，例如，设置值、检索值和显示值

等。在汽车这个例子中，制造出一辆汽车后，可能要执行给它加油的操作。这个加油操作是对特定的汽车执行的，因此，它类似于实例方法。

1. 返回值

声明新方法时，必须告诉 Objective-C 编译器该方法是否有返回值，如果有返回值，是哪种类型的值。要做到这一点，需要将返回类型放入开头的负号或者正号之后的圆括号。因此，声明

```
-(int) currentAge;
```

指定名为 `currentAge` 的实例方法将返回一个整型值。

类似地，语句

```
-(double) retrieveDoubleValue;
```

声明了一个返回双精度值的方法（第 4 章“数据型和表达式”将介绍有关这个数据型的更多内容）。

使用 Objective-C 中的 `return` 语句可以从方法中返回一个值，这与前一个程序例子中从 `main` 函数返回值的方式类似。

如果方法不返回值，可用 `void` 类型指明，语句如下：

```
-(void) print;
```

这条语句声明了一个名为 `print` 的方法，它不返回任何值。在这种情况下，无须在方法的末尾执行一条 `return` 语句。或者也可以执行一条不带任何指定值的 `return` 语句，语句如下：

```
return;
```

2. 方法的参数

代码清单 3-2 的 `@interface` 部分声明了其他两个方法：

```
-(void) setNumerator: (int) n;  
-(void) setDenominator: (int) d;
```

它们都是不返回值的实例方法。每个方法都有一个整型参数，这是通过参数名前面的 `(int)` 指明的。就 `setNumerator` 来说，其参数名是 `n`。这个名称可以是任意的，它是用来引用参数的方法名称的。因此，`setNumerator` 的声明指定向该方法传递一个名为 `n` 的整型参数，而且该方法没有要返回的值。这类似

于 `setDenominator` 的声明，不同之处是后者的参数名是 `d`。

要注意声明这些方法的语法。每个方法名都以冒号结束，这告诉 Objective-C 编译器该方法会有参数。接下来，指定参数的类型，并将其放入一对圆括号中，这与为方法自身指定返回类型的方式十分相似。最后，使用象征性的名称来确定方法所指定的参数。整个声明以一个分号结束，其语法如图 3.1 所示。

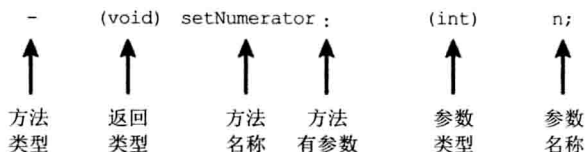


图 3.1 声明一个方法

如果方法接受一个参数，那么在引用该方法时，在方法名之后附加一个冒号。因此，`setNumerator:` 和 `setDenominator:` 是指定这两个方法的正确方式，每个方法都有一个参数。同样，在指定 `print` 方法时没有使用后缀的冒号，这表明此方法不带有任何参数。在第 7 章“类”中将学习如何指定带有多个参数。

3.5 @implementation 部分

与注释的一样，`@implementation` 部分包含声明在 `@interface` 部分的方法的实际代码，且需要指定存储在类对象中的数据类型。就像术语指出的那样，在 `@interface` 部分声明方法，并在 `@implementation` 部分定义它们（也就是说，给出实际的代码）。`@implementation` 部分的一般格式如下：

```
@implementation NewClassName
{
    memberDeclarations;
}
    methodDefinitions;
@end
```

`NewClassName` 表示的名称与 `@interface` 部分的名称相同。可以在父的名称之后使用冒号，如同在 `@interface` 部分使用冒号一样：

```
@implementation Fraction: NSObject
```

然而，它是可选的，而且通常并不这么做。

memberDeclarations 部分指定了哪种类型的数据将要存储到 `Fraction` 中，以及这些数据类型名称。可以看到，这一部分放入自己的一组花括号内。对于 `Fraction` 类而言：

```
int numerator;
int denominator;
```

声明表示 `Fraction` 对象有两个整型成员，即 `numerator` 和 `denominator`。

在这一部分声明的成员称为实例变量。你将看到，每次创建新对象时，将同时创建一组新的实例变量，而且是唯一的一组。因此，如果拥有两个 `Fraction`，一个名为 `fracA`，另一个名为 `fracB`，那么每一个都将有自己的一组实例变量。就是说，`fracA` 和 `fracB` 各自将拥有 `numerator` 和 `denominator`。Objective-C 系统将自动追踪这些实例变量，对使用对象而言，这是一件令人愉快的事情。`@implementation` 部分中的 *methodDefinitions* 部分包含在 `@interface` 部分指定的每个方法的代码中。与 `@interface` 部分类似，每种方法的定义通过方法的类型（或者实例）、它的返回值和参数进行标识，我们并没有使用分号来结束这行，而是将之后的方法代码放入一对花括号中。需要注意的是，使用 `@synthesize` 指令能够让编译器自动为你生成一些方法。第 7 章中会详细描述具体内容。

以下是代码清单 3-2 的 `@implementation` 部分：

```
//---- @implementation 部分 ----
@implementation Fraction
{
    int numerator;
    int denominator;
}

-(void) print
{
    NSLog ("%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
```

```

    denominator = d;
}

@end

```

`print` 方法使用 `NSLog` 显示实例变量 `numerator` 和 `denominator` 的值。但是这个方法引用 `numerator` 还是 `denominator` 呢？它引用的实例变量包含在作为消息接收者的对象中。这是一个重要的概念，我们简单回顾一下。

`setNumerator`：方法带有一个名为 `n` 的整型参数，并简单地存储到实例变量 `numerator` 中。

`setDenominator`：将其参数 `d` 的值存储到实例变量 `denominator` 中。

3.6 program 部分

`program` 部分包含解决特定问题的代码，如果有必要，它可以跨越多个文件。前面提到，必须在其中某个地方有一个名为 `main` 的函数。通常情况下，这是程序开始执行的地方。以下是代码清单 3-2 的 `program` 部分：

```

//---- program 部分 ----

int main (int argc, char * argv[])
{
    @autoreleasepool {
        Fraction *myFraction;

        // 创建一个分数实例并初始化

        myFraction = [Fraction alloc];
        myFraction = [myFraction init];

        // 设置分数为 1/3

        [myFraction setNumerator: 1];
        [myFraction setDenominator: 3];

        // 用打印方法显示分数

        NSLog (@@"The value of myFraction is:");
        [myFraction print];
    }

    return 0;
}

```


在 `main` 中，通过以下程序行定义了一个名为 `myFraction` 的变量：

```
Fraction *myFraction;
```

这一行表示 `myFraction` 是一个 `Fraction` 类型的对象。也就是说，`myFraction` 用于存储新的 `Fraction` 类的值。变量名前面的星号（*）在下面会有更详细的描述。

现在，你拥有一个用于存储 `Fraction` 的对象，需要创建一个分数，就像要求制造厂制造一辆汽车一样，可以用以下程序行实现：

```
myFraction = [Fraction alloc];
```

`alloc` 是 `allocate` 的缩写。因为要为新分数分配内存存储空间，表达式

```
[Fraction alloc]
```

向新创建的 `Fraction` 类发送一条消息。你请求 `Fraction` 类执行 `alloc` 方法，但是你还从未定义过这个 `alloc` 方法，那么它来自何处呢？此方法继承自一个父类。第 8 章将会详细讨论这个主题。

如果向某个类发送 `alloc` 消息，便获得该类的新实例。在代码清单 3-2 中，返回值存储在变量 `myFraction` 中。`alloc` 方法保证对象的所有实例变量都变成初始状态。然而，这并不意味着该对象已经进行了适当的初始化，从而可以使用。在创建对象之后，还必须对它初始化。

这项工作可以通过代码清单 3-2 中的下一条语句来完成：

```
myFraction = [myFraction init];
```

这里再次使用了一个并非自己编写的方法。`init` 方法用于初始化类的实例变量。注意，你正将 `init` 消息发送给 `myFraction`。也就是说，要在这里初始化一个特殊的 `Fraction` 对象，因此，它没有发送给类，而是发送给了类的一个实例。继续介绍下面的内容之前，务必理解这一点。

`init` 方法也可以返回一个值，即被初始化的对象。将返回值存储到 `Fraction` 的变量 `myFraction` 中。

代码创建新的实例并进行初始化的两行代码在 Objective-C 中特别常见，所以这两条消息通常组合在一起，语句如下：

```
myFraction = [[Fraction alloc] init];
```

内部消息表达式

```
[Fraction alloc]
```

将首先求值。可以看到，这条消息表达式的作用是创建实际的 `Fraction`。对它直接应用 `init` 方法，而不是像以前那样把创建的结果存储到一个变量中。所以，同样是先创建一个新的 `Fraction`，然后对它初始化。初始化的结果赋给了变量 `myFraction`。

作为最终的简写形式，经常把创建和初始化直接合并到声明行，语句如下：

```
Fraction *myFraction = [[Fraction alloc] init];
```

回到代码清单 3-2，现在已经可以设置分数的值。程序行

```
// 设置分数为 1/3
```

```
[myFraction setNumerator: 1];
[myFraction setDenominator: 3];
```

用于完成这项工作。第一条消息语句向 `myFraction` 发送 `setNumerator:` 消息，并提供一个值为 1 的参数。于是将控制转到 `Fraction` 类中定义的 `setNumerator:` 方法。因为 Objective-C 系统知道 `myFraction` 是 `Fraction` 类的对象，所以它知道要执行这个类的方法。

在 `setNumerator:` 方法中，传递来的值 1 存储在变量 `n` 中。该方法中唯一的程序行获得该值，并由实例变量 `numerator` 存储这个值。因此，`myFraction` 的分子已经被有效地设置为 1。

其后的消息用于调用 `myFraction` 的 `setDenominator:` 方法。在 `setDenominator:` 方法中，参数 3 被赋值给变量 `d`。然后把这个值存储到实例变量 `denominator` 中，这样就将 `myFraction` 赋值为 1/3。现在可以用代码清单 3-2 来实现显示此分数的值：

```
// 用打印方法显示分数
```

```
NSLog(@"The value of myFraction is:");
[myFraction print];
```

`NSLog` 调用仅显示以下文本：

```
The value of myFraction is:
```

使用以下消息表达式调用 `print` 方法：

```
[myFraction print];
```

在 `print` 方法中，将显示实例变量 `numerator` 和 `denominator` 的值，并用斜杠字符（/）分隔。

注意

在过去，iOS 程序员需要给对象发送 `release` 消息，通知系统释放对象，这在内存管理系统中称为手工引用计数。现在由于使用了 Xcode 4.2，程序员不必再担心内存释放的问题，并且可以依靠系统来释放内存。通过自动引用计数（Automatic Reference Counting，简称 ARC）的机制就可以做到。ARC 默认会开启。

似乎需要在代码清单 3-2 中编写大量的代码，才能完成代码清单 3-1 所实现的工作。对于这个简单的例子来说，确实如此，然而，使用对象的最终目的是使程序易于编写、维护和扩展。将来你会认识到这一点。

让我们重新回到 `myFraction` 的声明

```
Fraction *myFraction;
```

随后为其赋值。

`myFraction` 前的星号（*）表明 `myFraction` 是 `Fraction` 对象的引用（或指针）。变量 `myFraction` 实际上并不存储 `Fraction` 的数据（即分数的分子和分母），而是存储了一个引用（其实是内存地址），表明对象数据在内存中的位置。在声明 `myFraction` 时，它的值是未定义的，它没有被设定为任何值，并且没有默认值。在概念上，我们可以认为 `myFraction` 是一个能够容纳值的盒子。在初始化盒子时包含了一些未定义的值，它并没有被指定任何值，如图 3.2 所示。

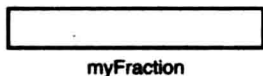


图 3.2 声明 `Fraction * myFraction;`

如果你创建一个新对象（例如，使用 `alloc`），就会在内存中为它保留足够的空间用于存储对象数据，这包括它的实例变量的空间，再另外预留了一些。通常，`alloc` 会返回存储数据的位置（对数据的引用），并赋给变量 `myFraction`。代码清单 3-2 中有如下语句：

```
myFraction = [Fraction alloc];
```

图 3.3 中表明创建一个对象，并将对象的引用赋给 myFraction。

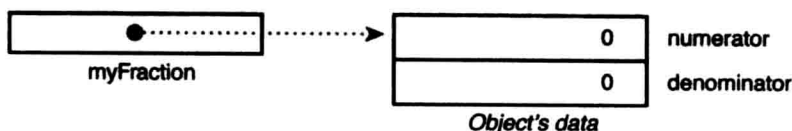


图 3.3 myFraction 及其数据的关系

注意

对象中存储的数据比标明的要多，不过不必在意这些。图中所示实例变量被指定为 0，这正是 alloc 方法做到的。然而对象始终没有被正确地初始化，你仍需使用 init 方法初始化新创建的对象。

注意图 3.3 中的箭头指示，这表明变量 myFraction 和创建的对象之间的关联（存储在 myFraction 中的是内存地址。对象数据就存储在这个内存地址中）。

代码清单 3-2 中，随后就设定了分数的分子和分母。图 3.4 描述了 Fraction 对象完全初始化的过程，分子被设定为 1，分母被设定为 3。

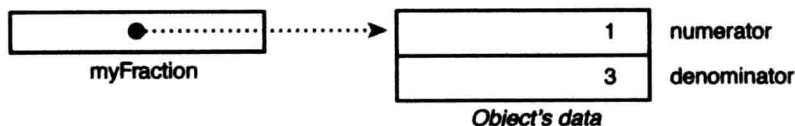


图 3.4 设定分数的分子和分母

下面的一个例子将展示如何在程序中使用多个分数。在代码清单 3-3 中，将一个分数设置为 2/3，另一个设置为 3/7，然后同时显示它们。

代码清单 3-3

```
// 使用分数的程序 - cont'd

#import <Foundation/Foundation.h>

//---- @interface 部分 ----

@interface Fraction: NSObject

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
```

```

@end

//---- @implementation 部分 ----

@implementation Fraction
{
    int numerator;
    int denominator;
}

-(void) print
{
    NSLog ("%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}

@end

//---- program 部分 ----

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

        Fraction *frac1 = [[Fraction alloc] init];
        Fraction *frac2 = [[Fraction alloc] init];

        // 设置第一个分数为 2/3

        [frac1 setNumerator: 2];
        [frac1 setDenominator: 3];

        // 设置第二个分数为 3/7

        [frac2 setNumerator: 3];
        [frac2 setDenominator: 7];

        // 显示分数
    }
}

```

```

NSLog(@"First fraction is:");

[frac1 print];

NSLog(@"Second fraction is:");
[frac2 print];

}
return 0;
}

```

代码清单 3-3 输出

```

First fraction is:
2/3
Second fraction is:
3/7

```

@interface 和 @implementation 部分与代码清单 3-2 一样，该程序创建了两个名为 frac1 和 frac2 的 Fraction 对象，然后将它们分别赋值为 2/3 和 3/7。注意，当 frac1 使用 setNumerator: 方法将其分子设置为 2 时，实例变量 frac1 也将实例变量 numerator 设置为 2。同样，frac2 使用相同的方法将其分子设置为 3 时，它特有的实例变量 numerator 也被设置为 3。每次创建新对象时，它就获得了自己特有的一组实例变量。图 3.5 描述了这个情况。

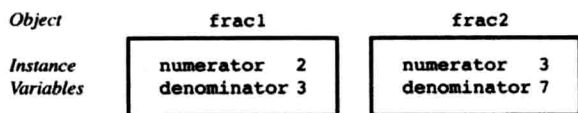


图 3.5 特有的实例变量

根据收到消息的对象，会引用正确的实例变量。因此，在

```
[frac1 setNumerator: 2];
```

语句中，只要 setNumerator: 方法用到名称 numerator，引用的都是 frac1 的 numerator，这是因为 frac1 是此消息的接收者。

3.7 实例变量的访问及数据封装

你已经看到处理分数的方法如何通过名称直接访问两个实例变量

`numerator` 和 `denominator`。事实上，实例方法总是可以直接访问它的实例变量的。然而，类方法则不能，因为它只处理本身，并不处理任何类实例（仔细想想）。但是，如果要从其他位置访问实例变量（例如，从 `main` 函数内部来访问，该如何实现？），在这种情况下，不能直接访问这些实例变量，因为它们是隐藏的。将实例变量隐藏起来的这种做法实际上涉及一个关键概念——“数据封装”。它使得编写定义的人在不必担心程序员（即类的使用者）是否破坏类的内部细节的情况下，扩展和修改其定义。数据封装提供了程序员和其他开发者之间的良好隔离层。

通过编写特殊方法来检索实例变量的值，可以用一种新的方式来访问它们。编写 `setNumerator:` 和 `setDenominator:` 方法用于给 `Fraction` 类的两个实例变量设定值。为了获取这些实例变量的值，我们需要编写新的方法。例如，创建两个名为 `numerator` 和 `denominator` 的新方法，用于访问相应的 `Fraction` 实例变量，这些实例是消息的接收者。结果是对应的整数值，你将返回这些值。以下是这两个新方法的声明：

```
-(int) numerator;
-(int) denominator;
```

下面是定义：

```
-(int) numerator
{
    return numerator;
}

-(int) denominator
{
    return denominator;
}
```

注意，它们访问的方法名和实例变量名是相同的，这样做不存在任何问题（虽然似乎有些奇怪）。事实上，这是很常见的情况。代码清单 3-4 用来测试这两个新方法。

代码清单 3-4

```
// 访问实例变量的程序 - cont'd

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

```

//---- @interface 部分 ----

@interface Fraction: NSObject

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
-(int) numerator;
-(int) denominator;

@end

//---- @implementation 部分 ----

@implementation Fraction
{
    int numerator;
    int denominator;
}

-(void) print
{
    NSLog ("%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}

-(int) numerator
{
    return numerator;
}

-(int) denominator
{
    return denominator;
}

@end

//---- program 部分 ----

```



```

int main (int argc, char *argv[])
{
    @autoreleasepool {
        Fraction *myFraction = [[Fraction alloc] init];

        // 设置分数为 1/3

        [myFraction setNumerator: 1];
        [myFraction setDenominator: 3];

        // 使用两个新的方法显示分数

        NSLog(@"The value of myFraction is: %i/%i",
              [myFraction numerator], [myFraction denominator]);
    }

    return 0;
}

```

代码清单 3-4 输出

```
The value of myFraction is 1/3
```

NSLog 语句显示发送给 myFraction: 的两条消息的结果，第一条消息检索 numerator 的值，第二条则检索 denominator 的值。

```

NSLog(@"The value of myFraction is: %i/%i",
      [myFraction numerator], [myFraction denominator]);

```

在第一条消息调用时，numerator 消息会发送给 Fraction 类的对象 myFraction。在这个方法中，分数中 numerator 的实例变量的值被返回。记住，方法执行的上下文环境就是接收到消息的对象。当访问 numerator 方法并且返回 numerator 实例变量值的时候，会取得 myFraction 的分子并返回，返回的整数传入 NSLog，从而显示出来。第二条消息调用时，denominator 方法会被调用并返回 myFraction 的分母，它仍通过 NSLog 显示。

顺便说一下，设置实例变量值的方法通常总称为设值方法（setter），而用于检索实例变量值的方法叫作取值方法（getter）。对 Fraction 而言，setNumerator: 和 setDenominator: 是设值方法，numerator 和 denominator 是取值方法。取值方法和设值方法通常称为访问器（accessor）方法。

确定你已经理解了设值方法和取值方法的不同。设值方法不会返回任何值，

因为其主要目的是将方法参数设为对应的实例变量的值。在这种情况下并不需要返回值。相反，取值方法的目的是获取存储在对象中的实例变量的值，并通过程序返回发送出去。为此，取值方法必须返回实例的值作为 `return` 的参数。

你不能在类的外部编写方法直接设置或获取实例变量的值，而需要编写设置方法和取值方法来设置或获取实例变量的值，这便是数据封装的原则。你必须通过使用一些方法来访问这些通常对“外界”隐藏的数据。这种做法集中了访问实例变量的方式，并且能够阻止其他一些代码直接改变实例变量的值。如果可以直接改变，会让程序很难跟踪、调试和修改。

这里还应指出一个名为 `new` 的方法可以将 `alloc` 和 `init` 的操作结合起来。因此，程序行

```
Fraction *myFraction = [Fraction new];
```

可用于创建和初始化新的 `Fraction`。但用两步来实现创建和初始化的方式通常更好，这样可以在概念上理解正在发生两个不同的事件：首先创建一个对象，然后对它初始化。

3.8 小结

现在，你知道了如何定义自己的类，如何创建该类的对象或实例，以及如何向这些对象发送消息。我们将在随后的章节中介绍 `Fraction`。你将了解到如何向某个方法传递多个参数，如何将定义划分到不同的文件，同时还了解到一些关键概念，如继承和动态绑定。然而，现在需要学习更多的数据类型，并使用 `Objective-C` 编写表达式的更多内容。首先，请尝试完成以下练习，测试是否已经理解本章所讲的重点。

3.9 练习

1. 下列名称中，哪些是不合法的？为什么？

<code>Int</code>	<code>playNextSong</code>	<code>6_05</code>
<code>_calloc</code>	<code>Xx</code>	<code>alphaBetaRoutine</code>
<code>clearScreen</code>	<code>_1312</code>	<code>z</code>
<code>ReInitialize</code>	<code>_</code>	<code>A\$</code>

2. 根据本章中的汽车示例，举出一个每天都要使用的对象。为这个对象确定一个类，并编写 5 个用于处理该对象的操作。
3. 给出练习 2 中的程序清单，使用以下语法：

```
[instance method];
```

中的格式重写程序清单。

4. 设想你拥有一艘船、一辆摩托车和一辆汽车。列出对其中每个对象执行的操作。这些操作之间有重叠吗？
5. 根据练习 4，设想有一个名为 `vehicle` 的类和一个名为 `myVehicle` 的对象，这个对象可以是汽车、摩托车或船中的任何一个。假设编写以下操作：

```
[myVehicle prep];  
[myVehicle getGas];  
[myVehicle service];
```

可以向这几个类的某一个对象执行一个操作，知道这样做的好处吗？

6. 在 C 这样的过程性语言中，思考涉及各种对象的操作，然后编写代码来执行这些操作。参见汽车例子，可用 C 语言编写清洗交通工具的过程，然后在该过程中编写代码来处理清洗汽车、清洗船及清洗摩托车等操作。如果采用这种方法，同时希望添加一种新的交通工具（参见以前的练习），那么能指出使用这种过程性的方法比使用面向对象的方法有什么好处和缺点吗？
7. 定义一个名为 `XYpoint` 的类，用来保存笛卡儿坐标 (x, y) ，其中 x 和 y 均为整数。定义一些方法，分别用来设置点的坐标 x 和 y ，并检索它们的值。编写一个 Objective-C 程序，实现这个新类并测试它。

数据类型和表达式

本章将讲解 Objective-C 的基本数据类型，并描述构成算术表达式的一些基本规则。

4.1 数据类型和常量

你已经遇到过 Objective-C 的基本数据类型 `int`。回顾一下，声明为 `int` 类型的变量只能用于保存整型值，也就是不包含小数位数的值。

Objective-C 还提供了另外 3 种基本数据类型：`float`、`double` 和 `char`。声明为 `float` 类型的变量可以存储浮点数（即包含小数位数的值）。`double` 类型和 `float` 类型一样，通常，前者表示的范围大约是后者的两倍。`char` 数据类型可存储单个字符，例如字母 `a`、数字字符 `6` 或者一个分号（后面将详细讨论有关内容）。

在 Objective-C 中，任何数字、单个字符或者字符串通常都称为常量。例如，数字 `58` 表示一个常量整数值。字符串 `@"Programming in Objective-C is fun."` 表示一个常量字符串对象。表达式 `@5` 表示一个常量数字对象。你将在第 15 章“数字、字符串和集合”中学到。

4.1.1 `int` 类型

整数常量由一个或多个数字的序列组成。序列前的负号表示该值是一个负数。值 `158`、`-10` 和 `0` 都是合法的整数常量。数字中间不允许插入空格，并且不能使用逗号（因此，`12,000` 是一个非法的整数常量，它必须写成 `12000`）。

每个值无论是字符、整数还是浮点数字，都有与其对应的值域。这个值域

与系统为特定类型的值分配的内存量有关。一般来说，在语言中没有规定这个量，它通常依赖于所运行的计算机，因此，叫作设备或机器相关量。例如，一个整数可在计算机上占用 32 位，或者可以使用 64 位存储。如果使用 64 位存储，整型变量将能够存储比 32 位更大的数值。

4.1.2 float 类型

声明为 float 类型的变量可以存储包含小数位的值。要区分浮点常量，可以看它是否包含小数点。值 3.、125.8 及 -0.0001 都是合法的浮点常量。要显示浮点值，可用 NSLog 转换字符 %f 或者 %g。

浮点常量也能用所谓的科学计数法来表示。值 1.7e4 就是使用这种计数法来表示的浮点值，它表示值 1.7×10^4 。

如上所述，double 类型与 float 类型非常相似，只是 double 类型的变量可存储的范围大概是 float 变量的两倍。

4.1.3 char 类型

char 变量可存储单个字符。将字符放入一对单引号中就能得到字符常量。因此，'a'、';' 和 '0' 都是合法的字符常量。第一个常量表示字母 a，第二个表示分号，第三个表示字符 0，它并不等同于数字 0。不要把字符常量和 C 语言风格的字符串混为一谈，字符常量是放在单引号中的单个字符，而字符串则是放在双引号中的任意个数的字符。正如在第 3 章提及的，前面有 @ 字符并且放在双引号中的字符串是 NSString 字符串对象。

字符常量 '\n'（即换行符）是一个合法的字符常量，尽管它似乎与前面提到的规则矛盾。这是因为反斜杠符号被认为是特殊符号。换句话说，Objective-C 编译器将字符 '\n' 看作单个字符，尽管它实际上由两个字符组成。其他特殊字符也是以反斜杠字符开头的。在 NSLog 调用中可以使用格式字符 %c，以便显示 char 变量的值。

在代码清单 4-1 中，使用了基本的 Objective-C 数据类型。

代码清单 4-1

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

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

```

{
    @autoreleasepool {
        int         integerVar = 100;
        float       floatingVar = 331.79;
        double      doubleVar = 8.44e+11;
        char        charVar = 'W';

        NSLog(@"integerVar = %i", integerVar);
        NSLog(@"floatingVar = %f", floatingVar);
        NSLog(@"doubleVar = %e", doubleVar);
        NSLog(@"doubleVar = %g", doubleVar);
        NSLog(@"charVar = %c", charVar);
    }
    return 0;
}

```

代码清单 4-1 输出

```

integerVar = 100
floatingVar = 331.790009
doubleVar = 8.440000e+11
doubleVar = 8.44e+11
charVar = W

```

在程序输出的第二行，你会注意到指定给 `floatingVar` 的值 331.79，实际显示成了 331.790009。事实上，实际显示的值是由具体使用的计算机系统决定的。出现这种不准确值的原因在于，计算机内部使用了特殊的方式表示数字。使用计算器处理数字时，很可能遇到相同的不准确性。如果用计算器计算 1 除以 3，将得到结果.33333333，很可能结尾带有一些附加的 3。这一串 3 是计算器计算 $1/3$ 的近似值。理论上，应该存在无限个 3。然而该计算器只能保存这些位的数字，这就是计算机的不确定性。同样的不确定性也出现在这里：在计算机内存中不能精确地表示一些浮点值。

4.1.4 限定词：long、long long、short、unsigned 及 signed

如果直接把限定词 `long` 放在 `int` 声明之前，那么所声明的整型变量在某些计算机上具有扩展的值域。一个 `long int` 声明的例子为：

```
long int factorial;
```

这条语句将变量 `factorial` 声明为 `long` 的整型变量。就像 `float` 和 `double` 变量一样，`long` 变量的具体范围也是由具体的计算机系统决定的。

要用 NSLog 显示 long int 的值，就要使用字母 l 作为修饰符，放在整型格式符号之前。这意味着格式符号 %li 将用十进制格式显示 long int 的值。

你也可以使用 long long int 变量，甚至是具有更大范围带有浮点数的 long double 变量。

把限定词 short 放在 int 声明之前时，它告诉 Objective-C 编译器要声明的特定变量用来存储相当小的整数。之所以使用 short 变量，主要出于对节约内存空间的考虑，当程序员需要大量内存而可用的内存量又十分有限时，就可用 short 变量来解决这个问题。

最后一个可以放在 int 变量之前的限定词，是在整数变量只用来存储正数的情况下使用的。以下语句

```
unsigned int counter;
```

向编译器声明，变量 counter 只用于保存正值。通过限制整型变量的使用，让它专门用于存储正整数，可以扩展整型变量的范围。

4.1.5 id 类型

id 数据类型可存储任何类型的对象。从某种意义说，它是一般对象类型。例如，程序行

```
id graphicObject;
```

将 graphicObject 声明为 id 类型的变量。可声明方法使其具有 id 类型的返回值，如下：

```
-(id) newObject: (int) type;
```

这个程序行声明了一个名为 newObject 的实例方法，它具有名为 type 的单个整型参数并有 id 类型的返回值。

id 类型是本书经常使用的一种重要的数据类型。这里介绍该类型的目的是为了保持本书的完整性。id 类型是 Objective-C 中十分重要的特性，它是多态和动态绑定的基础，这两个特性将在第 9 章“多态、动态类型和动态绑定”中详细讨论。

表 4.1 总结了基本数据类型和限定词。

表 4.1 基础数据类型

类 型	实 例	NSLog 字符
char	'a'、'\n'	%c
short int	—	%hi、%hx、%ho
unsigned short int	—	%hu、%hx、%ho %hu、%hx、%ho
int	12、-97、0xFFE0、0177	%i、%x、%o
unsigned int	12u、100U、0XFFu	%u、%x、%o
long int	12L、-2001、0xffffL	%li、%lx、%lo
unsigned long int	12UL、100ul、0xffeeUL	%lu、%lx、%lo
long long int	0xe5e5e5e5LL、500ll	%lli、%llx、&llo
unsigned long long int	12ull、0xffeeULL	%llu、%llx、%llo
float	12.34f、3.1e-5f、0x1.5p10、0x1P-1	%f、%e、%g、%a
double	12.34、3.1e-5、0x.1p3	%f、%e、%g、%a
long double	12.34L、3.1e-5l	%Lf、\$Le、%Lg
id	nil	%p

注意

在表 4.1 中，在整型常量中以 0 开头表示常量是八进制（基数 8）的，以 0x 开头或 (0X) 表示它是十六进制（基数 16）的，数字 0x.1p3 表示十六进制浮点常量。不必担心这些格式，这里只是为了使表格完整进行的总结。此外，前缀 f、l(L)、u(U)和 ll(LL)用来明确表示常量是 float、long、unsigned 和 long long 类型。

4.2 算术表达式

在 Objective-C 中，事实上与所有的程序设计语言一样，在两个数相加时使用加号 (+)，在两个数相减时使用减号 (-)，在两个数相乘时使用乘号 (*)，在两个数相除时使用除号 (/)。这些运算符称为二元算术运算符，因为它们运算两个值或项。

4.2.1 运算符的优先级

你已经看到如何在 Objective-C 中执行简单的运算，例如，加法。下面的程序进一步说明了减法、乘法和除法运算。在程序中执行的最后两个运算引入了

一个概念，即一个运算符比另一个运算符有更高的优先级。事实上，Objective-C 中的每一个运算符都有与之相关的优先级。

优先级用于确定包含多个运算符的表达式如何求值：优先级较高的运算符首先求值。如果表达式包含优先级相同的运算符，可按照从左到右或从右到左的方向来求值，具体按哪个方向求值取决于运算符。这就是通常所说的运算符结合性。

代码清单 4-2

// 说明各种算术运算符的用法

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        int  a = 100;
        int  b = 2;
        int  c = 25;
        int  d = 4;
        int  result;

        result = a - b;    // 减法
        NSLog (@\"a - b = %i\", result);

        result = b * c;    // 乘法
        NSLog (@\"b * c = %i\", result);

        result = a / c;    // 除法
        NSLog (@\"a / c = %i\", result);

        result = a + b * c; // 优先级
        NSLog (@\"a + b * c = %i\", result);

        NSLog (@\"a * b + c * d = %i\", a * b + c * d);
    }
    return 0;
}
```

代码清单 4-2 输出

```
a - b = 98
b * c = 50
a / c = 4
a + b * c = 150
```

```
a * b + c * d = 300
```

在声明整型变量 `a`、`b`、`c`、`d` 及 `result` 之后,程序将 `a` 减 `b` 的结果指派给 `result`,然后用恰当的 `NSLog` 调用来显示它的值。

下一条语句

```
result = b * c;
```

将 `b` 的值和 `c` 的值相乘并将其结果存储到 `result` 中。然后用 `NSLog` 调用来显示这个乘法的结果。到目前为止,你应该很熟悉这个过程了。

之后的程序语句引入了除法运算符——斜杠 (`/`)。100 除以 25 得到 4,可用 `NSLog` 语句在 `a` 除以 `c` 之后立即显示。

在某些计算机系统中,尝试用一个整数除以 0 将导致程序异常终止或出现异常。即使程序没有异常终止,执行这样的除法所得的结果也毫无意义。在第 6 章“选择结构”中,将看到如何在执行除法运算之前检验除数是否为 0。如果除数为 0,可采用适当的操作来避免除法运算。

表达式

```
a + b * c
```

不会产生结果 2550 (即 102×25); 相反,相应的 `NSLog` 语句显示的结果为 150。这是因为 Objective-C 与其他大多数程序设计语言一样,对于表达式中多重运算或项的顺序有自己的规则。通常情况下,表达式的计算按从左到右的顺序执行。然而,为乘法和除法运算指定的优先级比加法和减法的优先级要高。因此,Objective-C 认为表达式

```
a + b * c
```

等价于

```
a + (b * c)
```

(如果采用基本的代数规则,那么该表达式的计算方式是相同的。)

如果要改变表达式中项的计算顺序,可使用圆括号。事实上,前面列出的表达式是相当合法的 Objective-C 表达式。这样,可用表达式

```
result = a + (b * c);
```

替换代码清单 4-2 中的表达式,也可以获得同样的结果。然而,如果用表达式

```
result = (a + b) * c;
```

来替换，则赋给 `result` 的值将是 2550，因为要首先将 `a` 的值（100）和 `b` 的值（2）相加，然后将结果与 `c` 的值（25）相乘。圆括号也可以嵌套，在这种情况下，表达式的计算要从最里面的一对圆括号依次向外进行。只要确保结束圆括号和开始圆括号的数目相等即可。

从代码清单 4-2 中的最后一条语句可发现，对 `NSLog` 指定表达式作为参数时，无须将该表达式的结果先指派给一个变量，这种做法是完全合法的。表达式

```
a * b + c * d
```

可根据以上述规则，按照

```
(a * b) + (c * d)
```

即

```
(100 * 2) + (25 * 4)
```

来计算。

求出的结果 300 将传递给 `NSLog` 函数。

4.2.2 整数运算和一元负号运算符

代码清单 4-3 巩固了前面讨论的内容，并引入了整数运算的概念。

代码清单 4-3

// 更多的算术表达式

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        int a = 25;
        int b = 2;
        float c = 25.0;
        float d = 2.0;

        NSLog (@@"6 + a / 5 * b = %i", 6 + a / 5 * b);
        NSLog (@@"a / b * b = %i", a / b * b);
        NSLog (@@"c / d * d = %f", c / d * d);
        NSLog (@@"-a = %i", -a);
    }
    return 0;
}
```

```
}
```

代码清单 4-3 输出

```
6 + a / 5 * b = 16  
a / b * b = 24  
c / d * d = 25.000000  
-a = -25
```

前3条语句中，在 `int` 和 `a`、`b` 及 `result` 的声明之间插入了额外的空格，以便对齐每个变量的声明，使用这种方法书写语句可使程序更容易阅读。还可以注意到，在迄今出现的每个程序中，每个运算符前后都有空格。这种做法同样不是必需的，仅仅是出于美观上的考虑。一般来说，在允许单个空格的任何位置都可以插入额外的空格。如果能使程序更容易阅读，输入空格键的操作还是值得做的。

在代码清单 4-3 中，第一个 `NSLog` 调用中的表达式巩固了运算符优先级的概念。该表达式的计算按以下顺序执行：

(1) 因为除法的优先级比加法高，所以先将 `a` 的值（25）除以 5。该运算将给出中间结果 5。

(2) 因为乘法的优先级也高于加法，所以随后中间结果（5）将乘以 2（即 `b` 的值），并获得新的中间结果（10）。

(3) 最后计算 6 加 10，并得出最终结果（16）。

第二条 `NSLog` 语句引入了一种新误解。你希望 `a` 除以 `b`，再乘以 `b` 的操作返回 `a`（已经设置为 25）。但此操作并不会产生这一结果，在输出显示器上显示的是 24。难道计算机在某个地方迷失了方向？如果这样就太不幸了。其实该问题的实际情况是，这个表达式是采用整数运算来求值的。

如果回头看一下变量 `a` 和 `b` 的声明，你会想起它们都是作为 `int` 类型声明的。当包含两个整数的表达式求值时，Objective-C 系统都将使用整数运算来执行这个操作。在这种情况下，数字的所有小数部分将丢失。因此，计算 `a` 除以 `b`，即 25 除以 2 时，得到的中间结果是 12，而不是期望的 12.5。这个中间结果乘以 2，就得到最终结果 24。这样，就解释了出现“丢失”数字的情况。

在代码清单 4-3 的倒数第二个 `NSLog` 语句中看到，如果用浮点值代替整数来执行同样的运算，就会获得期望的结果。

决定使用 `float` 变量还是 `int` 变量应该基于变量的使用目的。如果无须使用任何小数位，就可以使用整型变量。这将使程序更加高效，换言之，它可以在大多数计算机上更快速地执行。另一方面，如果需要精确到小数位，那就很清楚应该选择什么。此时，唯一需要回答的问题是使用 `float` 还是 `double`。对此问题的回答取决于使用数据所需的精度以及它们的量级。

在最后一行 `NSLog` 语句中，使用了一元负号运算符对变量 `a` 的值取负。这个一元运算符是用于单个值的运算符，而二元运算符作用于两个值。负号实际上扮演了一个双重角色：作为二元运算符，它执行两个数相减的操作；作为一元（或单目）运算符，它对一个值取负。

与其他算术运算符相比，一元负号运算符具有更高的优先级，但一元正号运算符（`+`）除外，一元正号运算符和算术运算符的优先级相同。因此，表达式

```
c = -a * b;
```

将执行 `-a` 乘以 `b`。

4.2.3 模运算符

本章介绍的最后一个运算符是模运算符，它由百分号（`%`）表示。通过分析代码清单 4-4 的输出，请尝试确定这种运算符的工作方式。

代码清单 4-4

// 模数运算符

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        int a = 25, b = 5, c = 10, d = 7;

        NSLog(@"a %% b = %i", a % b);
        NSLog(@"a %% c = %i", a % c);
        NSLog(@"a %% d = %i", a % d);
        NSLog(@"a / d * d + a %% d = %i", a / d * d + a % d);
    }
    return 0;
}
```

代码清单 4-4 输出

```
a % b = 0
a % c = 5
a % d = 4
a / d * d + a % d = 25
```

注意，main 中的语句定义并初始化了变量 a、b、c 和 d，这些工作均在一条语句内完成。

你已经知道。NSLog 使用百分号之后的字符来确定如何输出下一个参数。然而，如果它后面紧跟另一个百分号，那么 NSLog 函数认为你的目的是想显示百分号，并在程序输出的适当位置插入一个百分号。

如果你总结出用模运算符%的功能是计算第一个值除以第二个值所得的余数，那就对了。在第一个例子中，25 除以 5 所得的余数是 0。如果用 25 除以 10，余数是 5，输出中的第二行语句可以证实。执行 25 除以 7 将得到余数 4，它显示在输出的第三行。

现在，我们把注意力转移到最后一条语句求值的表达式上。前面曾提到，Objective-C 使用整数运算来执行两个整数间的任何运算。因此，两个整数相除所产生的任何余数将被完全丢弃。如果使用表达式 a/d 表示 25 除以 7，将会得到中间结果 3。将这个结果乘以 d 的值（即 7），将会产生中间结果 21。最后，加上 a 除以 b 的余数，该余数由表达式 a%d 来表示，会产生最终结果 25。这个值与变量 a 的值相同并非巧合。一般来说，表达式

```
a / b * b + a % b
```

的值将始终与 a 的值相等。当然，这是在假定 a 和 b 都是整型值的条件下做出的。事实上，定义的模运算符%只用于处理整数。

就优先级而言，模运算符的优先级与乘法和除法的优先级相等。毫无疑问，这意味着表达式

```
table + value % TABLE_SIZE
```

等价于

```
table + (value % TABLE_SIZE)
```

4.2.4 整型值和浮点值的相互转换

若要更有效地开发 Objective-C 程序，必须理解 Objective-C 中浮点值和整

型值之间进行隐式转换的规则。代码清单 4-5 表明数值数据类型间的一些简单转换。

代码清单 4-5

// Objective-C 中的基本转换

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        float f1 = 123.125, f2;
        int i1, i2 = -150;

        i1 = f1; // 浮点数到整数的转换
        NSLog(@"%f assigned to an int produces %i", f1, i1);

        f1 = i2; // 整数到浮点数的转换
        NSLog(@"%i assigned to a float produces %f", i2, f1);

        f1 = i2 / 100; // 整数除以整数
        NSLog(@"%i divided by 100 produces %f", i2, f1);

        f2 = i2 / 100.0; // 整数除以浮点数
        NSLog(@"%i divided by 100.0 produces %f", i2, f2);

        f2 = (float) i2 / 100; // 类型强制转换运算符
        NSLog(@"(float) %i divided by 100 produces %f", i2, f2);
    }
    return 0;
}
```

代码清单 4-5 输出

```
123.125000 assigned to an int produces 123
-150 assigned to a float produces -150.000000
-150 divided by 100 produces -1.000000
-150 divided by 100.0 produces -1.500000
(float) -150 divided by 100 produces -1.500000
```

在 Objective-C 中，只要将浮点值赋值给整型变量，数字的小数部分都会被删节。因此，在前一个程序中，把 f1 的值指派给 i1 时，数字 123.125 将被删节，这意味着只有整数部分（即 123）存储到了 i1 中。程序输出的第一行验证了上述程序就是这种情况。

把整型变量指派给浮点变量的操作不会引起数字值的任何改变，该值仅由系统转换并存储到浮点变量中。程序输出的第二行验证了这一情况：i2 的值 (-150) 进行了正确转换并存储到 float 变量 f1 中。

接下来的两行程序输出说明了在编写算术表达式时要记住的两点。第一点与整数运算有关，在前一章已经讨论了这一点。只要表达式中的两个运算数是整型（这一情况还适用于 short、unsigned 和 long 整型），该运算就将在整数运算的规则下进行。因此，由乘法运算产生的任何小数部分都将删除，即使该结果赋给一个浮点变量（如同我们在程序中所做的那样），也是如此。当整型变量 i2 除以整数常量 100 时，系统将该除法作为整数除法来执行。因此，-150 除以 100 的结果是 -1，将 -1 存储到 float 变量 f1 中。

前一个程序中的下一个除法涉及一个整数变量和一个浮点常量。在 Objective-C 中，对于任何处理两个值的运算，如果其中一个值是浮点变量或常量，那么这一运算将作为浮点运算来处理。因此，当 i2 的值除以 100.0 时，系统将除法作为浮点除法来计算，并产生结果 -1.5，该结果将赋给 float 变量 f1。

4.2.5 类型转换运算符

你已经看到，在声明和定义方法时，如何将类型放入圆括号中来声明返回值和参数的类型。在表达式中使用类型时，它表示一个特殊的用途。

代码清单 4-5 中的最后一个除法运算

```
f2 = (float) i2 / 100; // 类型强制转换运算符
```

引入了类型转换运算符。

为了求表达式的值，类型转换运算符将变量 i2 的值转换成 float 类型。该运算符永远不会影响变量 i2 的值。它是一元运算符，行为和其他一元运算符一样。因为表达式 -a 永远不会影响 a 的值，因此，表达式 (float) a 也不会影响 a 的值。

类型转换运算符比其他所有的算术运算符的优先级都高，但一元减号 and 一元加号运算符除外。当然，如果需要，可经常使用圆括号进行限制，以任何想要的顺序来执行一些项。

下面是使用类型转换运算符的另一个例子，表达式：

```
(int) 29.55 + (int) 21.99
```


在 Objective-C 中等价于

```
29 + 21
```

因为将浮点值转换成整数的后果就是舍弃其中的浮点值。表达式

```
(float) 6 / (float) 4
```

得到的结果为 1.5，与下列表达式的执行效果相同：

```
(float) 6 / 4
```

类型转换运算符通常用于将一般 id 类型的对象转换成特定类的对象。例如，

```
id myNumber;
Fraction *myFraction;
...
myFraction = (Fraction *) myNumber;
```

将 id 变量 myNumber 的值强制类型转换成一个 Fraction 对象。转换结果赋给 Fraction 变量 myFraction。

4.3 赋值运算符

Objective-C 语言允许使用以下的一般格式将算术运算符和赋值运算符合并到一起：

```
op=
```

在这个格式中，op 是任何算术运算符，包括+、-、*、/和%。此外，op 还可以是任何用于移位和屏蔽操作的位运算符，这些内容将在以后讨论。

请考虑下面这条语句：

```
count += 10;
```

通常所说的“加号等号”运算符(+=)将运算符右侧的表达式和左侧的表达式相加，再将结果保存到运算符左边的变量中。因此，上面的语句和以下语句等价：

```
count = count + 10;
```

表达式

```
counter -= 5
```

使用“减号等号”赋值运算符将 counter 的值减 5，它和下面这个语句等价

```
counter = counter - 5
```

下面是一个稍微复杂一些的表达式：

```
a /= b + c
```

无论等号右侧出现何值（或者 b 加 c 的和），都将用它除以 a，再把结果存储到 a 中。因为加法运算符比赋值运算符的优先级高，所以表达式会首先执行加法。事实上，除逗号运算符外的所有运算符都比赋值运算符的优先级高。而所有的赋值运算符的优先级相同。

在这个例子中，该表达式的作用和下列表表达式相同：

```
a = a / (b + c)
```

使用赋值运算符的目的有 3 个：首先，程序语句更容易书写，因为运算符左侧的部分没有必要在右侧重写。其次，结果表达式通常容易阅读。最后，这些运算符的使用可使程序的运行速度更快，因为编译器有时在计算表达式时能够产生更少的代码。

4.4 Calculator 类

现在定义一个新类，我们将创建一个 **Calculator** 类，它是一个简单的四则运算计算器，可用来执行加、减、乘和除运算。类似于常见的计算器，这种计算器必须能够记录累加结果，即通常所说的累加器。因此，方法必须能够执行以下操作：将累加器设置为特定值、将其清空（或设置为 0），以及在完成时检索它的值。代码清单 4-6 包括这个新类的定义和一个用于试验该计算器的测试程序。

代码清单 4-6

```
// 实现 Calculator 类

#import <Foundation/Foundation.h>

@interface Calculator: NSObject

// 累加方法
-(void) setAccumulator: (double) value;
-(void) clear;
-(double) accumulator;
```

```
// 算术方法
-(void) add: (double) value;
-(void) subtract: (double) value;
-(void) multiply: (double) value;
-(void) divide: (double) value;
@end

@implementation Calculator
{
    double accumulator;
}

-(void) setAccumulator: (double) value
{
    accumulator = value;
}

-(void) clear
{
    accumulator = 0;
}

-(double) accumulator
{
    return accumulator;
}

-(void) add: (double) value
{
    accumulator += value;
}

-(void) subtract: (double) value
{
    accumulator -= value;
}

-(void) multiply: (double) value
{
    accumulator *= value;
}

-(void) divide: (double) value
{
    accumulator /= value;
}
@end
```

```

int main (int argc, char *argv[])
{
    @autoreleasepool {
        Calculator *deskCalc = [[Calculator alloc] init];

        [deskCalc setAccumulator: 100.0];
        [deskCalc add: 200.];
        [deskCalc divide: 15.0];
        [deskCalc subtract: 10.0];
        [deskCalc multiply: 5];
        NSLog(@"The result is %g", [deskCalc accumulator]);
    }
    return 0;
}

```

代码清单 4-6 输出

```
The result is 50
```

Calculator 类只有一个实例变量，以及一个用于保存累加器值的 **double** 变量。方法定义的本身非常直观。

要注意调用 **multiply** 方法的消息：

```
[deskCalc multiply: 5];
```

该方法的参数是一个整数，而它期望的参数类型却是 **double**。因为方法的数值参数会自动转换以匹配期望的类型，所以此处不会出现任何问题。**multiply** 期望使用 **double** 值，因此调用该函数时，整数 5 将自动转换成双精度浮点值。虽然自动转换过程会自己进行，但在调用方法时提供正确的参数类型仍是一个较好的程序设计习惯。

要认识到与 **Fraction** 类不同，**Fraction** 类可能使用多个不同的分数，在这个程序中可能希望只处理单个 **Calculator** 对象。然而，定义一个新类以便更容易处理这个对象仍是有意义的。在某个时候，你可能会为计算器添加一个图形前端，以使用户能够在屏幕上真正单击按钮，就像系统或手机中已安装的计算器应用程序一样。

在第 10 章“变量和数据类型”中会更多地讨论有关数据类型转换和位操作。

在以后的一些练习中，可以看到定义 **Calculator** 类的另一个好处，即便于扩展。

4.5 练习

1. 下列常量中，哪些是非法的？为什么？

123.456	0x10.5	0X0G1
0001	0xFFFF	123L
0Xab05	0L	-597.25
123.5e2	.0001	+12
98.6F	98.7U	17777s
0996	-12E-12	07777
1234uL	1.2Fe-7	15,000
1.234L	197u	100U
0XABCDEFL	0xabcu	+123

2. 编写一个程序，使用以下公式将华氏温度 (F) 27° 转换成摄氏温度 (C)：

$$C = (F - 32) / 1.8$$

不需要定义一个类来执行计算。只需要简单地列出表达式就满足要求。

3. 以下程序将输出什么结果？

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        char c, d;

        c = 'd';
        d = c;
        NSLog (@@"d = %c", d);
    }
    return 0;
}
```

4. 编写一个程序，求以下多项式的值（计算表达式时，只需直接计算；因为在 Objective-C 中没有幂指操作符）：

$$3x^3 - 5x^2 + 6$$

(令 $x = 2.55$)

5. 编写一个程序，求下列表达式的值，并显示其结果（记住要使用指数格式显示结果）。

$$(3.31 \times 10^{-8} + 2.01 \times 10^{-7}) / (7.16 \times 10^{-6} + 2.01 \times 10^{-8})$$

6. 复数包含两个部分：实部和虚部。如果 a 是实部，b 是虚部，那么符号

$a + bi$

可用来表示复数。

编写一个 Objective-C 程序，定义一个名为 **Complex** 的新类。依照为 **Fraction** 类创建的范例，为该类定义以下方法：

```
-(void) setReal: (double) a;
-(void) setImaginary: (double) b;
-(void) print;    // 显示为 a + bi
-(double) real;
-(double) imaginary;
```

编写一个测试程序测试这个新类和各个方法。

7. 假设你正开发操作图形对象的函数库。从定义名为 **Rectangle** 的新类开始。目前，仅记录矩形的宽和高即可。开发一些方法用于设置矩形的宽和高、检索这些值以及计算矩形的面积和周长。假定这些矩形对象使用整数坐标栅格来描述矩形，例如，一台计算机屏幕。在这种情况下，假定矩形的宽和高都是整数值。

以下是 **Rectangle** 类的 **@interface** 部分：

```
@interface Rectangle: NSObject
-(void) setWidth: (int) w;
-(void) setHeight: (int) h;
-(int) width;
-(int) height;
-(int) area;
-(int) perimeter;
@end
```

请编写 **implementation** 部分，并编写一个测试程序来测试新类的方法。

8. 修改代码清单 4-6 中的 **add:**、**subtract:**、**muntiply:**和 **divide:**方法，使其返回累加器的结果值。测试这些新方法。
9. 完成练习 8 后，把以下方法添加到 **Calculator** 类中并测试它们：

```
-(double) changeSign; // 改变累加器的正负号
-(double) reciprocal; // 累加器
-(double) xSquared;   // 累加器的平方
```

10. 为代码清单 4-6 中的 **Calculator** 添加一项存储功能。实现以下方法声明并测试它们：

```
-(double) memoryClear;    // 清理内存
-(double) memoryStore;    // 设置内存为累加器
```

```
-(double) memoryRecall;           // 设置累加器到内存  
-(double) memoryAdd: (double) value; // 添加值到内存  
-(double) memorySubtract: (double) value; // 与内存的值相减
```

为最后两组方法设置一个累加器，并能够对内存执行指定的操作。所有的方法都需要返回累加器的值。

5

循环结构

在 Objective-C 中，有若干方法可以重复执行一系列代码。本章的主题是循环功能，它们由以下几部分组成：

- for 语句
- while 语句
- do 语句

我们从一个简单的例子开始讨论：计数。

如果要把 15 个弹球排列成一个三角形，排列后的弹球可能如图 5.1 所示。

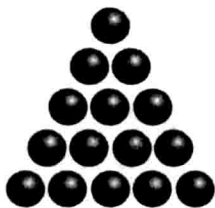


图 5.1 三角形排列示例

三角形的第一行包含一个弹球，第二行包含两个弹球，以此类推。

一般来说，包含 n 行的三角形可容纳的弹球总数等于 $1 \sim n$ 之间所有整数之和，这个和称为三角数（triangular number）。如果从 1 开始，第 4 位三角数将等于 1 到 4 之间连续整数的和（ $1+2+3+4$ ），即 10。

假设要编写一个程序来计算第 8 位三角数的值。显然可以在头脑中计算这个值，但是为了学习参数，假设你用 Objective-C 编写一个带有参数的程序来执行这个任务。代码清单 5-1 显示了这个程序。

代码清单 5-1

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

// 计算第 8 个三角数的程序

int main (int argc, char * argv[])
{
    @autoreleasepool {
        int triangularNumber;

        triangularNumber = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8;

        NSLog (@"The eighth triangular number is %i", triangularNumber);
    }

    return 0;
}
```

代码清单 5-1 输出

```
The eighth triangular number is 36
```

如果计算相对较小的三角数，代码清单 5-1 中的方法工作良好，但是要找出第 200 位三角数的值，该程序将如何处理呢？必须修改代码清单 5-1，以便显式地将 1~200 之间的所有整数相加，这项工作肯定是冗长乏味的。值得庆幸的是，有一个比较简单的方法。

计算机的基本属性之一就是它能够重复执行一组语句。这种循环能力让程序员能够开发出包含重复过程的简捷程序，这些过程能够以不同的方式执行成百上千的程序语句。Objective-C 包含 3 种用于编写循环结构的程序语句。

5.1 for 语句

让我们来看一个使用 for 语句的程序。代码清单 5-2 的目的是计算第 200 位三角数。看看你是否可以找出 for 语句的工作方式。

代码清单 5-2

```
// 计算第 200 个三角数的程序
// 介绍 for 语句

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

```

int main (int argc, char *argv[])
{
    @autoreleasepool {
        int n, triangularNumber;

        triangularNumber = 0;

        for ( n = 1; n <= 200; n = n + 1 )
            triangularNumber += n;

        NSLog (@"The 200th triangular number is %i", triangularNumber);
    }

    return 0;
}

```

代码清单 5-2 输出

```
The 200th triangular number is 20100
```

需要对代码清单 5-2 进行一些解释。用于计算第 200 位三角数的方法其实与代码清单 5-1 中用于计算第 8 位三角数的方法是相同的，就是求 1~200 之间的整数之和。

在执行 for 语句之前，变量 `triangularNumber` 被设置为 0。一般来说，在程序使用变量之前，需要将所有的变量初始化为某个值（和处理对象一样）。后面将会学到，某些类型的变量有默认的初始值，但是无论如何都应该为变量设置初始值。for 语句提供的机制让你不用显式地写出 1~200 之间的每个整数。从某种意义上讲，这条语句将为你生成这些数字。

for 语句的一般格式如下：

```

for ( init_expression; loop_condition; loop_expression )
    program_statement

```

圆括号中的 3 个表达式 *init_expression*、*loop_condition* 和 *loop_expression* 建立了程序循环的“环境”。其后的 *program_statement*（当然是以一个分号结束）可以是任何合法的 Objective-C 程序语句，它们组成循环体。这条语句执行的次数由 for 语句中设置的参数决定。

for 语句的第一部分标着 *init_expression* 用于在循环开始之前设置初始值。在代码清单 5-2 中，for 语句的第一部分将 `n` 的初始值设置为 1。可以看到，赋值是一种合法的表达式形式。

for 语句的第二部分用于指定继续执行循环所需的条件。换言之，只要满足这个条件，循环就将继续执行。再次参见代码清单 5-2，for 语句中的 *loop_condition* 是由以下关系表达式指定的：

```
n <= 200
```

这个表达式可读成“*n* 小于或等于 200”。Objective-C 程序设计语言提供了若干关系运算符，“小于或等于”运算符（由小于号[<]和紧跟在其后的等号[=]组成）只是其中一个。这些关系运算符用于测试特定的条件，如果满足条件，测试结果为真（或“true”）；如果不满足条件，测试结果为假（或“false”）。

表 5.1 列出了 Objective-C 中可用的所有关系运算符。

表 5.1 关系运算符

运 算 符	含 义	例 子
==	等于	count == 10
!=	不等于	flag != DONE
<	小于	a < b
<=	小于或等于	low <= high
>	大于	points > POINT_MAX
>=	大于或等于	j >= 0

关系运算符的优先级比所有的算术运算符都低。这意味着，表达式

```
a < b + c
```

将按

```
a < (b + c)
```

来求值。

这与你的期望一致。如果 *a* 的值小于 *b + c* 的值，表达式将为 true，否则表达式为 false。要特别注意等于运算符（==），不要将其与赋值运算符（=）混淆。表达式 *a* == 2 用于测试 *a* 的值是否等于 2，而表达式 *a* = 2 用于将值 2 赋值给变量 *a*。

选择要使用哪个关系运算符由所做的具体测试决定，有的情况下由你的具体偏好决定。例如，关系表达式

```
n <= 200
```

可以等价地表示为

```
n < 201
```

回到前一个例子中，只要关系测试结果为 `true`，在这个例子中， n 的值小于或等于 200 时，形成 `for` 循环体的程序语句 (`triangularNumber += n;`) 将被重复执行。这条语句的作用是将 n 的值和 `triangularNumber` 的值加到一起。

不再满足 `loop_condition` 时，程序在 `for` 循环之后的程序语句继续执行。在该程序中，该循环终止之后将继续执行 `NSLog` 语句。

`for` 语句的最后一部分包含一个表达式，它在每次执行循环体之后求值。在代码清单 5-2 中，`loop-expression` 的作用是将 n 的值加 1。因此，每次把 n 的值加到 `triangularNumber` 之后，它的值都要加 1，而且该值将从 1 一直增加到 201。

值得注意的是， n 的最终值（即 201）将不会加到 `triangularNumber` 的值上，因为只要不再满足循环条件，或只要 n 等于 201，循环就会终止。

总之，`for` 语句将按以下步骤执行：

(1) 先求初始表达式的值。这个表达式通常设置一个将在循环中使用的变量，对于某些初始值（例如 0 或 1）来说，通常称为索引变量。

(2) 求循环条件的值。如果条件不满足（即表达式为 `false`），循环就立即终止。然后执行在循环之后的程序语句。

(3) 执行组成循环体的程序语句。

(4) 求循环表达式的值。这个表达式通常用于改变索引变量的值，最常见的情况是，将索引变量的值加 1 或减 1。

(5) 返回到步骤 (2)。

记住，循环条件要在进入循环时在第一次执行循环体之前立即求值。还要记住，不要在循环末尾处的结束圆括号后面放置分号，这将导致循环立即终止。

代码清单 5-2 在计算最终结果的过程中，实际生成了所有前 200 个三角数，所以，生成这些数字的一个表格是不错的主意。然而，为了节省空间，假定你只想打印一张包含前 10 个三角数的表，代码清单 5-3 执行这个任务。

代码清单 5-3

```
// 生成三角数表的程序
```

```

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    @autoreleasepool {
        int n, triangularNumber;

        NSLog(@"TABLE OF TRIANGULAR NUMBERS");
        NSLog(@" n Sum from 1 to n");
        NSLog(@"-- -----");

        triangularNumber = 0;

        for ( n = 1; n <= 10; ++n ) {
            triangularNumber += n;
            NSLog(@" %i      %i", n, triangularNumber);
        }

        return 0;
    }
}

```

代码清单 5-3 输出

TABLE OF TRIANGULAR NUMBERS

N Sum from 1 to n

```

-- -----
1      1
2      3
3      6
4     10
5     15
6     21
7     28
8     36
9     45
10    55

```

在代码清单 5-3 中，前 3 个 NSLog 语句的目的仅仅是提供总标题和输出列的标题。

在显示适当的标题后，程序将计算前 10 个三角数。当你正在计算 1 到 n 的和时，使用变量 n 记录当前计算的数字，使用变量 triangularNumber 存储 n 的三角数值。

for 语句的执行首先是将变量 n 的值设置为 1。前面提到过，在 for 语句之后的程序语句构成了程序循环的主体。但是如果不是只想执行单个程序语句，而是想执行一组语句，该怎么办呢？将这些程序语句放入一对花括号中就可以达到这个目的。系统会把这组（或块）语句看作单个实体。一般来说，在 Objective-C 程序中能使用单个语句的任何位置都能使用语句块，不过要记住，语句块必须放在一对花括号中才能使用。

因此，在代码清单 5-3 中，将 n 加到 triangularNumber 上的表达式和紧跟其后的 NSLog 语句构成了循环体。要特别注意程序语句的缩进方式。快速扫视一下，可以轻易确定哪些语句构成了 for 循环。还应该注意程序员采用不同的编码风格，一些人更喜欢用以下方式输入循环：

```
for ( n = 1; n <= 10; ++n )
{
    triangularNumber += n;
    NSLog ("%i %i", n, triangularNumber);
}
```

其中开始的花括号位于 for 的下一行。严格地说，这只是一个爱好问题，并不会影响程序。

若要计算下一个三角数，只要将 n 的值加到前一个三角数即可。第一次遍历 for 循环时，第一个三角数为 0，因此， n 等于 1 时，triangularNumber 的新值就是 n 的值，即 1。然后显示 n 的值和 triangularNumber，并带有适当数目的空格，这些空格将插入到格式字符串中，以确保这两个变量的值可以排列到相应的列标题之下。

因为现在执行的是循环体，所以随后将求循环表达式的值。然而，这条 for 语句中的表达式看上去有些奇怪。当然，这肯定是一个印刷错误，本来应该插入 $n=n+1$ ，而不是这个看上去相当奇怪的表达式：

```
++n
```

事实是：++ n 其实是合法的 Objective-C 表达式。它引入了 Objective-C 程序设计语言中的一个新的（而且相当独特的）运算符——自增运算符。双加号（或叫作自增运算符）的作用是将其运算数加 1。加 1 运算在程序设计中很常见，所以创造了一个特殊的操作符，专门完成这项任务。因此，表达式 ++ n 等价于表达式 $n=n+1$ 。乍一看时，可能觉得 $n=n+1$ 更易阅读，但是很快你就会习惯这

种运算符，甚至更喜欢它的简洁性。

当然，有自增运算符执行加 1 的操作，就有相应的运算符执行减 1 操作。正如你所猜测的，这种运算符叫作自减运算符，它由双减号来表示。因此，用 Objective-C 书写的表达式

```
bean_counter = bean_counter - 1
```

可用自减运算符等价地表示成以下形式：

```
--bean_counter
```

一些程序员喜欢将++或--放到变量名后面，如 n++或 bean_counter--。这种情况是可以接受的，只不过是使用 for 语句的个人喜好问题。

你可能已经注意到，代码清单 5-3 输出的最后一行没有对齐。使用以下 NSLog 语句来代替代码清单 5-3 中对应的语句，可改正这个小毛病。

```
NSLog ("%2i %i", n, triangularNumber);
```

要验证这个改变是否解决了上述问题，下面给出修改后的程序（名为代码清单 5-3A）的输出。

代码清单 5-3A 输出

TABLE OF TRIANGULAR NUMBERS

n Sum from 1 to n

1	1
2	3
3	6
4	10
5	15
6	21
7	28
8	36
9	45
10	55

NSLog 语句所做的主要改动是它包含了字段宽度说明。字符%2i 告知 NSLog 函数：不仅在特定点显示整数值，而且要展示的整数应该占用显示器的两列。通常，占用空间少于两列的任何整数（即，0~9 之间的整数）在显示时都带有一个前导空格。这种情况称为向右对齐。

因而，通过使用字符宽度说明%2i，可以确保至少有两列将用于显示 n 的值，还能保证对齐 triangularNumber 的值。

5.1.1 键盘输入

代码清单 5-2 可计算出第 200 个三角数，但不能计算更多的数。如果要计算第 50 个或第 100 个三角数，该怎么办呢？好吧，如果是这种情况，就不得不更改程序，以便 for 循环可以执行合适的次数。还必须更改 NSLog 语句来显示正确的消息。

更加简单的解决方案可能是，通过某种方式允许程序向你询问要计算哪个三角数。在给出回答后，程序就可以计算出期望的三角数。使用一个名为 scanf 的函数，就可以实现这样的解决方案。在概念上，scanf 函数与 NSLog 函数类似。但是 NSLog 函数用于显示值，而 scanf 函数的用途是程序员可以把值输入到程序中。当然，如果使用图形用户界面 (UI) 编写 Objective-C 程序，如 Cocoa 或 iOS 应用程序，那么在程序中可能根本不用 NSLog 或 scanf。

代码清单 5-4 首先询问用户要计算哪个三角数，然后计算该数并显示结果。

代码清单 5-4

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

int main (int argc, char * argv[])
{
    @autoreleasepool {
        int n, number, triangularNumber;

        NSLog(@"What triangular number do you want?");
        scanf ("%i", &number);

        triangularNumber = 0;

        for ( n = 1; n <= number; ++n )
            triangularNumber += n;

        NSLog(@"Triangular number %i is %i\n", number, triangularNumber);
    }

    return 0;
}
```

在后面的程序输出中，用户输入的数字（100）用黑体显示，以便与由程序显示的输出相区别。

代码清单 5-4 输出

```
What triangular number do you want?  
100  
Triangular number 100 is 5050
```

根据输出可以看出，数字 100 是由用户输入的。然后该程序计算第 100 个三角数并将结果 5050 显示在终端上。如果用户想要计算一个特定的三角数，可以输入 10 或者 30 这样的数字。

在代码清单 5-4 中，第一个 NSLog 语句用于提示用户输入数字。当然，向用户提示输入内容总是好的。输出消息后，调用 scanf 函数。scanf 的第一个参数是格式字符串，它不以 @ 字符开头。NSLog 的第一个参数始终是 NSString，而 scanf 的第一个参数是 C 风格的字符串。在前面已经提及过，C 风格的字符串前面不用加字符 @。

格式字符串告知 scanf 要从控制台（或者是 Terminal 窗口，如果正在使用 Terminal 应用程序编译程序的话）读入的值类型。和 NSLog 一样，%i 字符用于指定整型值。

scanf 函数的第二个参数用于指定将用户输入的值存储在哪里。在这种情况下，变量 number 之前的 & 字符是必需的。不同于给变量 number 赋值，而是指定输入的值存储在哪里。现在不要担心此处。在第 13 章中介绍指针时，我们将详细讨论这个字符，它实际上是一个运算符。

根据前面的讨论，可以看到代码清单 5-4 中的 scanf 调用指定要输入整型值，并将其存储到变量 number 中。这个值代表用户希望计算哪个三角数。

输入这个数字之后（按下键盘上的 Enter 键，表示该数字的输入工作已完成），程序便计算指定的三角数。实现方式和代码清单 5-2 中的一样，唯一的不同是，此处没有用 200 作为界限，而是用 number 作为界限。

注意

输入数字键盘上的 Enter 键可能并不能将你输入的数字发送给程序，此时使用键盘上的 Return 键试试。

计算出期望的三角数之后，显示结果，然后程序的执行结束。

5.1.2 嵌套的 for 循环

代码清单 5-4 向用户提供了以下灵活性：程序可以计算出任何想要的三角数。但是假设用户要计算 5 个三角数的列表，该怎么办呢？这种情况下，用户可简单地将程序执行 5 次，每次输入要计算的列表中下一个三角数即可。

实现上述目标还有另一种方式，并且就学习 Objective-C 而言，它更为有趣，就是让程序处理这种情况。向程序插入循环，让整个计算过程重复执行 5 次，就可以最好地完成这一任务。可以使用 for 语句来建立这样的循环，代码清单 5-5 及其输出说明了这类语句。

代码清单 5-5

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

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

{
    @autoreleasepool {
        int n, number, triangularNumber, counter;

        for ( counter = 1; counter <= 5; ++counter ) {
            NSLog (@"What triangular number do you want?");
            scanf ("%i", &number);

            triangularNumber = 0;

            for ( n = 1; n <= number; ++n )
                triangularNumber += n;

            NSLog (@"Triangular number %i is %i", number, triangularNumber);
        }
    }

    return 0;
}
```

代码清单 5-5 输出

```
What triangular number do you want?
12
Triangular number 12 is 78
```

```
What triangular number do you want?
```

```
25
```

```
Triangular number 25 is 325
```

```
What triangular number do you want?
```

```
50
```

```
Triangular number 50 is 1275
```

```
What triangular number do you want?
```

```
75
```

```
Triangular number 75 is 2850
```

```
What triangular number do you want?
```

```
83
```

```
Triangular number 83 is 3486
```

该程序包含两层 **for** 循环语句。最外层的 **for** 循环语句如下：

```
for ( counter = 1; counter <= 5; ++counter )
```

这条语句指定该程序循环正好执行 5 次。因为 **counter** 的值最初设为 1，并且依次加 1，直到它的值不再小于或等于 5 为止（换句话说，直到它到达 6），所以可以看出执行了 5 次。

与上一个程序例子不同，该程序的其他位置都没有使用变量 **counter**。它的作用仅仅是充当 **for** 语句中的循环计数器。然而，因为它是一个变量，所以必须在程序中声明。

该程序的循环实际上由其他所有的程序语句组成，如花括号所示。如果有以下概念，也许能更容易地理解该程序的工作方式：

```
For 5 times
{
    Get the number from the user.
    Calculate the requested triangular number.
    Display the results.
}
```

前面的循环部分指的是 **calculate the requested triangular number**，它实际上包括：将变量 **triangularNumber** 的值设为 0，以及计算三角数的 **for** 循环。这样，这个 **for** 语句实际上包含另一个 **for** 语句。这在 Objective-C 中是相当合法的，而且可以继续嵌套，甚至可嵌套任何想要的层。

处理比较复杂的程序结构（如嵌套的 **for** 语句）时，适当地使用缩进显得尤为重要。扫视一下，就能轻易确定每个 **for** 语句中包含哪些语句。

5.1.3 for 循环的变体

在结束 for 循环的讨论之前，先探讨一下生成这种循环时允许的一些语法变化。编写 for 循环时，你可能发现在开始循环之前需要初始化多个变量，或者可能每次循环都要计算多个表达式。for 循环的任何位置都可包含多个表达式，只要使用逗号分隔这些表达式即可。例如，使用以下形式开始的 for 循环：

```
for ( i = 0, j = 0; i < 10; ++i )  
    ...
```

在循环开始前，将 *i* 的值设为 0，将 *j* 的值设为 0。两个表达式 *i*=0 和 *j*=0 通过逗号隔开，而且两者都是循环的 `init_expression` 部分。另一个 for 循环的例子为：

```
for ( i = 0, j = 100; i < 10; ++i, j -= 10 )  
    ...
```

for 循环开始设置了两个索引变量：*i* 和 *j*，在循环开始之前，它们分别被初始化为 0 和 100。每次执行完循环体之后，*i* 的值加 1，*j* 的值减 10。

就像可能希望 for 循环的特定字段包含多个表达式一样，可能还需要省略语句中的一个或多个字段。通过省略指定的字段并使用分号标记其位置，可简单地实现这一点。省略 for 语句中某个字段的最常见情形发生在无须计算初始表达式的值时。在这种情况下，`init_expression` 字段可简单地保留空白，只要仍然包括分号即可，语句如下：

```
for ( ; j != 100; ++j )  
    ...
```

如果在进入循环之前已经将 *j* 设置了初始值，则可采用这条语句。

省略 `looping_condition` 字段的 for 循环实际上是无限循环，就是理论上执行无限次的循环。只要有其他方式退出循环（例如，执行 `return`、`break` 或 `goto` 语句，相关内容将在本书后面讨论），就可以使用这一循环。

在 for 循环中，还可定义一个变量作为初始表达式的一部分。使用以前定义变量的传统方式可实现。例如，下面的语句可用于设置 for 循环，它定义了整型变量 `counter`，并将其初始化为 1，语句如下：

```
for ( int counter = 1; counter <= 5; ++counter )
```

变量 `counter` 只在 for 循环的整个执行过程中是已知的（它被称为局部变

量)，并且不能在循环外部访问。

最后一种 for 循环的变体可以在对象集合上执行所谓的快速枚举。第 15 章“数字、字符串和集合”会详细介绍此内容。

5.2 while 语句

while 语句进一步扩展了 Objective-C 语言中的循环功能指令系统。这个经常使用的结构的语法如下：

```
while ( expression )
    program statement
```

圆括号中指定的表达式 (expression) 将被求值。如果表达式求值的结果为 true，则执行随后的程序语句 (program statement)。执行完这条语句（或位于花括号中的一组语句）后，将再次对表达式求值，如果求值的结果为 true，将再次执行程序语句。继续这个过程，直到表达式的最终求值结果是 false 时，循环将终止。然后，程序执行 program statement 之后的语句。

作为它的用法示例，代码清单 5-6 建立了一个 while 循环，它只是从 1 数到 5。

代码清单 5-6

```
// 此程序引入了 while 语句

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    @autoreleasepool {
        int count = 1;

        while ( count <= 5 ) {
            NSLog ("%i", count);
            ++count;
        }
    }

    return 0;
}
```

代码清单 5-6 输出

```
1
2
3
4
5
```

程序最初将 `count` 的值设为 1，然后开始执行 `while` 循环。因为 `count` 的值小于或等于 5，所以将执行它后面的语句。花括号将 `NSLog` 语句和对 `count` 执行加 1 操作的语句定义为 `while` 循环。从程序的输出可以看出，这个程序执行了 5 次，直到 `count` 的值是 5 为止。

从以上程序中你可能认识到，使用 `for` 语句同样可以方便地完成该任务。事实上，`for` 语句都可转换成等价的 `while` 语句，反之亦然。例如，下面这个普通的 `for` 语句

```
for ( init_expression; loop_condition; loop_expression )
    program statement
```

可用 `while` 语句的形式等价地表示为：

```
init_expression;
while ( loop_condition )
{
    program statement
    loop_expression;
}
```

熟悉了 `while` 语句的用法之后，对于何时用 `while` 语句、何时用 `for` 语句，你会有更好的认识。一般来说，在执行预定次数的循环时，首选 `for` 语句。同样，如果初始表达式、循环表达式和循环条件都涉及同一变量，那么 `for` 语句很可能是合适的选择。

下一个程序提供了使用 `while` 语句的另一个例子。这个程序计算两个整数值的最大公约数（greatest common divisor）。两个整数的最大公约数（此后将其缩写为 `gcd`）是可整除这两个整数的最大整数值。例如，10 和 15 的 `gcd` 是 5，因为 5 是可整除 10 和 15 的最大整数。

可使用一个过程（或算法）来获得任意两个整数的 `gcd`，它基于欧几里得在公元前 300 年左右首次研究出的一个方法，其说明如下。

问题：找出两个非负整数 `u` 和 `v` 的最大公约数。

步骤 1: 若 v 等于 0, 则结束, 即 gcd 等于 u 。

步骤 2: 计算 $\text{temp}=u\%v$, $u=v$, $v=\text{temp}$, 并回到步骤 1。

不要过分关注上述算法的运行细节, 相信它就可以了。此处我们更关心开发一个程序来找出最大公约数, 而不是分析这一算法的实现方式。

使用算法描述找出最大公约数这一问题的解决方案之后, 开发一个计算机程序成了一项十分简单的工作。算法步骤的分析揭示: 只要 v 的值不等于 0, 就会重复执行步骤 2。由于有了这一发现, 该算法在 Objective-C 中利用 `while` 语句来实现是很自然的。

代码清单 5-7 用于找出用户输入的两个整数的 gcd 。

代码清单 5-7

// 找到两个非负整数的最大公约数

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

int main (int argc, char * argv[])
{
    @autoreleasepool {
        unsigned int u, v, temp;

        NSLog (@"Please type in two nonnegative integers.");
        scanf ("%u%u", &u, &v);

        while ( v != 0 ) {
            temp = u % v;
            u = v;
            v = temp;
        }

        NSLog (@"Their greatest common divisor is %u", u);
    }

    return 0;
}
```

代码清单 5-7 输出

Please type in two nonnegative integers.

150 35

Their greatest common divisor is 5

代码清单 5-7A 输出（重新执行）

```
Please type in two nonnegative integers.
```

```
1026 540
```

```
Their greatest common divisor is 54
```

输入两个整型值并分别存储到变量 u 和 v 之后（使用 `%u` 格式字符读入一个无符号的整型值），程序进入一个 `while` 循环来计算它们的最大公约数。退出 `while` 循环之后， u 的值会显示出来，即代表 v 和 u 的原始值的 gcd，并且显示一条适当的消息。

第7章“类”中，返回处理分数时，将利用这个算法来找出最大公约数。利用这个算法将分数化简到最简单的形式。

对于下一个说明 `while` 语句的程序，设想的任务是翻转从终端输入的整数位。例如，如果用户输入数字 1234，该程序将把这个数字的位颠倒过来，并显示结果 4321。

注意

`NSLog` 调用会导致每个数字出现在输出的单独行上。熟悉 `printf` 函数的 C 程序员可以使用该函数，而不是让数字连续地显示。

要编写这样的程序，首先必须提出一个算法来实现所陈述的工作。最常见的情况是，分析自己解决问题的方法可以产生一个算法。对于颠倒一个数字的各位的这项工作，解决方案可简单地陈述为“从右到左依次读取数字的位。”通过开发一个过程，从数字最右边的位开始依次分离或取出该数字的每个位，计算机程序就可以依次读取数字的各个位，提取的位随后可以作为已颠倒数字的下一位显示在终端上。

通过将整数除以 10 之后取其余数，可提取整数最右边的数字。例如， $1234\%10$ 会得出值 4，就是 1234 最右边的数字，也是颠倒后数字的第一位（记住，可以使用模运算符得到一个整数除以另一个整数所得的余数）。先将数字除以 10（回忆一下整数除法的工作方式），再重复这个过程，就可以获得下一个数字。因此， $1234/10$ 的结果为 123，而 $123\%10$ 的结果为 3，它是颠倒后数字的第二个数。

这个过程可一直继续执行，直到计算出最后一个数字为止。一般情况下，如

果最后一个整数除以 10 的结果为 0，那么这个数字就是最后一个要提取的数字。

代码清单 5-8 提示用户输入一个数值，然后从右向左依次显示该数值各个位的数字。

代码清单 5-8

// 颠倒显示数字的位数

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

int main (int argc, char * argv[])
{
    @autoreleasepool {
        int number, right_digit;

        NSLog(@"Enter your number.");
        scanf ("%i", &number);

        while ( number != 0 ) {
            right_digit = number % 10;
            NSLog ("%i", right_digit);
            number /= 10;
        }
    }

    return 0;
}
```

代码清单 5-8 输出

```
Enter your number.
13579
9
7
5
3
1
```

5.3 do 语句

迄今为止，本章讨论的两个循环结构都要在循环开始前测试一组条件。因此，如果条件不满足，那么可能永远不会执行循环体。开发程序时，有时需要在循环结尾（而不是在开始）处执行测试。很自然，Objective-C 语言也提供了

专门的语言结构用于处理这种情况，即 **do** 语句。该语句的语法如下：

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

do 语句的执行过程为：首先，执行程序语句（**program statement**）。其次，求圆括号中表达式（**expression**）的值，如果表达式的求值结果为 **true**，循环将继续，并再次执行程序语句。只要表达式的计算结果始终为 **true**，就将重复执行程序语句。当表达式求出的值为 **false** 时，循环将终止并以正常顺序执行程序中的下一条语句。

do 语句只是 **while** 语句的简单转置，它把循环条件放在循环的结尾，而不是开头。

代码清单 5-8 使用 **while** 语句来翻转数字中的各个位。回到这个程序，并确定如果用户输入 0 而不是 13579，将会发生什么？在这种情况下，**while** 循环中的语句将永远不会执行，输出结果不会有任何显示。如果用 **do** 语句代替 **while** 语句，可确保程序至少要循环执行一次，从而保证在所有的情况下都至少显示一个数字。代码清单 5-9 展示了如何使用 **do** 语句。

代码清单 5-9

// 颠倒显示数字的位数

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        int number, right_digit;

        NSLog(@"Enter your number.");
        scanf ("%i", &number);

        do {
            right_digit = number % 10;
            NSLog(@"%i", right_digit);
            number /= 10;
        }
        while ( number != 0 );
    }

    return 0;
}
```

```
}
```

代码清单 5-9 输出

```
Enter your number.  
135  
5  
3  
1
```

代码清单 5-9A 输出（重新执行）

```
Enter your number.  
0  
0
```

从该程序的输出可以看到，当输入 0 时，程序就会正确地显示数字 0。

5.4 break 语句

在执行循环的过程中，有时候你希望只要发生特定的条件（例如，检测到错误条件或在遍历一系列数据时查找到了），就立即退出循环。**break** 语句可以实现这个目的，只要执行 **break** 语句，程序将立即退出正在执行的循环，而无论此循环是 **for**、**while** 还是 **do**。循环内 **break** 之后的语句将被跳过，并且该循环的执行也将终止，而转去执行循环外的其他语句。

如果在嵌套循环中执行 **break** 语句，仅会退出执行 **break** 语句的最内层循环。

break 语句的格式仅是在关键字 **break** 之后添加一个分号，形式如下：

```
break;
```

5.5 continue 语句

continue 语句和 **break** 语句类似，但它并不会使循环结束。执行 **continue** 语句时，循环会跳过该语句之后直到循环结尾处之间的所有语句。除此之外，循环将和平常一样执行。

continue 通常用来根据某个条件绕过循环中的一组语句，除此之外，循环会继续执行。**continue** 语句的格式如下：

```
continue;
```

不要使用 `break` 和 `continue` 语句，除非你非常熟悉编写程序循环，并知道如何从中优雅地退出。这些语句很容易滥用，并导致程序很难理解。

5.6 小结

现在你已经熟悉了 Objective-C 语言提供的所有基本的循环结构，接下来可以继续学习这门语言中的其他语句，利用它们在程序运行过程中做出判断。下一章将详细介绍选择结构。

5.7 练习

1. 编写一个程序，为所有从 1~10 之间的整数 n 生成并显示 n 和 n^2 的表，确保能打印正确的列标题。
2. 使用以下公式，同样能为任何整数 n 生成三角数：

$$\text{triangularNumber} = n(n + 1) / 2$$

例如，第 10 个三角数，也就是 55，通过把上述公式中的 n 用 10 来代替，可以生成。编写一个程序，使用上述公式生成三角数表。用该程序在 5~50 之间每隔 5 个数生成一个三角数（也就是说，生成第 5、10、15、…、50 个三角数）。

3. 整数 n 的阶乘可写成 $n!$ ，它表示 1~ n 之间所有连续整数的乘积。例如，5 的阶乘可用以下方法计算： $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$ 。编写一个程序，生成并打印前 10 个整数的阶乘表。
4. 字段宽度说明前面的负号能使字段按左对齐方式显示。用以下 `NSLog` 语句代替代码清单 5-3 中对应的语句，运行程序并比较这两种情况产生的结果：

```
NSLog(@"%-2i %i", n, triangularNumber);
```

5. 代码清单 5-5 只允许用户输入 5 个不同的数字。修改这个程序，使用户能够输入要计算三角数的数字。
6. 对代码清单 5-2~代码清单 5-5，用等价的 `while` 语句代替所有用到

的 for 语句。运行每个程序，验证这两种方案是恒等的。

7. 如果在代码清单 5-8 中输入负数，会发生什么情况？试试看。
8. 编写一个程序，计算整数各位上数字的和。例如，整数 2155 各位上的数字和为 $2+1+5+5$ ，即 13。该程序可接收用户输入的任意整数。

选择结构

对于任何程序语言来说，有能力进行判断是一项基本特性。需要在执行循环语句时，判断何时终止循环。Objective-C 程序设计语言也构造了以下几种判断结构：

- if 语句
- switch 语句
- conditional 运算符

6.1 if 语句

Objective-C 程序设计语言提供了一般的判断能力，即 if 语句这样的语言结构。这种语句的一般格式如下：

```
if ( expression )  
    program statement
```

假设，要将“如果不下雨，我就去游泳”这样的句子转换成 Objective-C 语言，可使用上述 if 语句格式，将这个句子“编写”成以下形式：

```
if ( it is not raining )  
    I will go swimming
```

if 语句根据指定的条件，限定程序语句的执行（或者括在花括号中的多条语句）——“如果不下雨，我就去游泳”。类似地，在程序语句

```
if ( count > MAXIMUM_SONGS )  
    [playlist maxExceeded];
```

中，只要 count 的值大于 MAXIMUM_SONGS，就会发送消息 maxExceeded 给

playlist; 否则，这条消息会被忽略。

下面将举实际的例子说明。假如想要编写一个程序，接收键盘输入的整数，然后显示出这个整数的绝对值。直接的方法就是简单地在整数小于 0 时对它取负。上述语句中的短语“如果它小于 0”，表示程序必须进行判断，这个判断可以通过 if 语句来实现，程序如下。

代码清单 6-1

```
// 计算一个整数的绝对值

#import <Foundation/Foundation.h>

int main (int argc, char * argv[])
{
    @autoreleasepool {
        int number;

        NSLog(@"Type in your number: ");
        scanf ("%i", &number);

        if ( number < 0 )
            number = -number;

        NSLog(@"The absolute value is %i", number);
    }
    return 0;
}
```

代码清单 6-1 输出

```
Type in your number:
-100
The absolute value is 100
```

代码清单 6-1 输出（重新运行）

```
Type in your number:
2000
The absolute value is 2000
```

程序执行了两次，目的是验证它能正确运行。当然，你可能希望多次执行程序，以获得更高的可信度，这样就知道程序确实能正确工作，但是你至少应该知道，已经检查了程序判断的两种可能的结果。

向用户显示一条消息之后，用户将输入整数值，程序将该值存储到 `number` 中，然后程序测试 `number` 的值，确定该值是否小于 0。如果这个值小于 0，将执行以下程序语句，对 `number` 的值取负。如果 `number` 的值不小于 0，将自动略过这条程序语句（如果这个值已经是正的，则无须对它取负）。随后，程序将显示 `number` 的绝对值，并终止运行。

看一看另一个使用 `if` 语句的程序。再向 `Fraction` 类添加一个名为 `convertToNum` 的方法，该方法将提供一个用实数表示的分数值。换言之，方法用分子除以分母并用双精度的值返回结果。因此，对于分数 $1/2$ ，该方法将返回值 0.5。

方法的声明可能是这样的：

```
-(double) convertToNum;
```

而且，它的定义形如：

```
-(double) convertToNum
{
    return numerator / denominator;
}
```

当然，并非完全如此。实际上，在定义这个方法时会遇到两个重要的问题。能发现它们吗？第一个和算术转换有关。回想起 `numerator` 和 `denominator` 都是整型的实例变量。因此，对两个整数执行除法时，会出现什么情况？正常的情况下，这会被作为整数除法来完成！因此，会将分数 $1/2$ 转换成实数，上述代码结果为 0。在执行除法之前，只要使用类型强制运算符将一个或两个运算数转换成浮点值，就可以轻松解决这个错误。

```
(double) numerator denominator
```

回忆一下，这种运算符的优先级相对较高，因此，在执行除法之前先将 `numerator` 转换成 `double` 类型。此外，无须转换 `denominator`，因为算术运算规则将替你完成这项工作。

使用这种方法还要注意第二个问题，即应该检查被除数是否为 0（总应该检查这种情况）。这个方法的调用者可能由于疏忽，忘记了设置分数的分母或将分母设置为 0，当然，你并不希望程序异常终止。

修改后的 `convertToNum` 方法如下：


```

-(double) convertToNum
{
    if (denominator != 0)
        return (double) numerator / denominator;
    else
        return NAN;
}

```

如果分数的 **denominator** 为 0, 此处规定它返回 **NAN** (表示不是一个数字)。这个字符被定义在系统头文件 **math.h** 中, 它会被自动引入你的程序。

下面使用这个新方法。

代码清单 6-2

```

#import <Foundation/Foundation.h>

@interface Fraction: NSObject

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
-(int) numerator;
-(int) denominator;
-(double) convertToNum;
@end

@implementation Fraction
{
    int numerator;
    int denominator;
}

-(void) print
{
    NSLog(@" %i/%i ", numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}

```

```

-(int) numerator
{
    return numerator;
}

-(int) denominator
{
    return denominator;
}

-(double) convertToNum
{
    if (denominator != 0)
        return (double) numerator / denominator;
    else
        return NAN;
}
@end

int main (int argc, char * argv[])
{
    @autoreleasepool {
        Fraction *aFraction = [[Fraction alloc] init];
        Fraction *bFraction = [[Fraction alloc] init];

        [aFraction setNumerator: 1]; // 第一个分数是 1/4
        [aFraction setDenominator: 4];

        [aFraction print];
        NSLog(@"=");
        NSLog(@"%g", [aFraction convertToNum]);

        [bFraction print]; // 永远不会分配一个值
        NSLog(@"=");
        NSLog(@"%g", [bFraction convertToNum]);
    }
    return 0;
}

```

代码清单 6-2 输出

```

1/4
=
0.25
0/0
=
nan

```

将 `aFraction` 设置成 $1/4$ 后，程序会利用 `convertToNum` 方法将此分数转换成一个十进制的值，这个值随后就会显示出来，即 `0.25`。

第二种情况，没有明确设置 `bFraction` 的值，因此，分数的分子和分母会初始化为 `0`，这是实例变量默认的初始值。这就解释了 `print` 方法显示的结果。同时，还导致 `convertToNum` 方法中的 `if` 语句返回值 `NAN`，你会注意到 `NSLog` 实际显示为 `nan`。

6.1.1 if-else 结构

如果有人问你某个数是偶数还是奇数，你的判断思路可能是，检查这个数的最后一位数字。如果最后一位数字是 `0`、`2`、`4`、`6` 或 `8` 中的任何一个，就能说明这个数是偶数；否则，它是奇数。

确定某个数是偶数还是奇数时，计算机使用一种更简便的方法，它并不检查数的最后一位数字是否是 `0`、`2`、`4`、`6` 或 `8`，而是检验这个数能否整除 `2`。如果能够整除 `2`，这个数是偶数；否则就是奇数。

你已经知道如何使用模运算符`%`计算两个整数相除所得的余数。这使它成为一个理想的运算符，可以判断整数能否整除 `2`。如果某个数除以 `2` 所得的余数为 `0`，它就是偶数；否则就是奇数。

现在，编写一个程序判断用户输入的整型值是偶数还是奇数，随后在终端显示一条消息，参见代码清单 6-3。

代码清单 6-3

// 确定数字是偶数还是奇数的程序

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

int main (int argc, char * argv[])
{
    @autoreleasepool {
        int number_to_test, remainder;

        NSLog(@"Enter your number to be tested: ");
        scanf ("%i", &number_to_test);

        remainder = number_to_test % 2;

        if ( remainder == 0 )
```

```

    NSLog(@"The number is even.");

    if ( remainder != 0 )
        NSLog(@"The number is odd.");
}

return 0;
}

```

代码清单 6-3 输出

```

Enter your number to be tested:
2455
The number is odd.

```

代码清单 6-3 输出（重新运行）

```

Enter your number to be tested:
1210
The number is even.

```

输入一个数后，计算此数除以 2 所得的余数。第一条 if 语句测试了这个余数的值，检验它是否等于 0。如果等于 0，将显示消息“The number is even”。

第二条 if 语句测试余数，检验它是否不等于 0，如果不等于 0，就会显示一条消息声明这个数是奇数。

实际的情况是：只要第一条 if 语句成功，第二条 if 语句肯定失败，反之亦然。如果回顾本节开始处有关偶数/奇数的讨论，就会知道：如果一个数能被 2 整除，它就是偶数；否则就是奇数。

编写程序时，这个“否则”的概念使用得相当频繁，所以被作为一个特殊结构来处理这一情况。在 Objective-C 中，它通常称为 if-else 结构，一般格式如下：

```

if ( expression )
    program statement 1
else
    program statement 2

```

实际上，if-else 仅仅是 if 语句一般格式的一种扩展形式。如果表达式的计算结果是 true，将执行之后的 program statement 1；否则，将执行 program statement 2。在任何情况下，都会执行 program statement 1 或 program statement 2 中的一个，而不是两个都执行。

可将 if-else 语句结合到前面的程序中，用单个 if-else 语句替换程序中的两个 if 语句。你将看到，这种新的程序语句实际上有助于以某种方式减少程序的复杂性，同时又提高可读性。

代码清单 6-4

// 确定数字是偶数还是奇数的程序（第 2 个版本）

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

int main (int argc, char * argv[])
{
    @autoreleasepool {
        int number_to_test, remainder;

        NSLog(@"Enter your number to be tested:");
        scanf ("%i", &number_to_test);

        remainder = number_to_test % 2;

        if ( remainder == 0 )
            NSLog(@"The number is even.");
        else
            NSLog(@"The number is odd.");
    }

    return 0;
}
```

代码清单 6-4 输出

```
Enter your number to be tested:
1234
The number is even.
```

代码清单 6-4 输出（重新运行）

```
Enter your number to be tested:
6551
The number is odd.
```

不要忘记，双等号（==）用于相等测试，而单个等号是赋值运算符。如果忘记了这一点，因为疏忽而在 if 语句中使用了赋值运算符，那就会导致许多令人头痛的事。

6.1.2 复合条件测试

到目前为止，本章使用过的 if 语句都只是两个数之间的简单条件测试。代码清单 6-1 将 number 的值与 0 进行比较，而代码清单 6-2 则将分数的分母与 0 进行比较。有时候虽然不是必需的，但最好是建立一些复杂的测试。例如，假设要计算一次考试中分数介于 70~79 之间的分数个数（含 70 和 79）。在这种情况下，不是根据一个界限来比较考试分数的值，而是根据两个界限 70 和 79 进行比较，以确保一个考试分数位于指定的范围之内。

Objective-C 语言提供了用于执行这种复合条件测试所必需的机制。所谓复合条件测试，就是用逻辑与（AND）或者逻辑或（OR）运算符连接起来的一个或多个简单条件测试。这两个运算符分别使用字符对 && 和 ||（两个竖线字符）来表示。举一个例子，Objective-C 语句

```
if ( grade >= 70 && grade <= 79 )  
    ++grades_70_to_79;
```

只在 grade 的值大于或等于 70，或者小于或等于 79 时才对 grade_70_to_79 的值加 1。使用类似的方式，语句

```
if ( index < 0 || index > 99 )  
    NSLog(@"Error - index out of range");
```

在 index 小于 0 或大于 99 时会执行 NSLog 语句。

在 Objective-C 中，复合运算符可形成极其复杂的表达式。Objective-C 向程序员提供了在构成表达式时极大的灵活性，但这种灵活性经常被滥用。简单的表达式往往更容易阅读和调试。

在形成复合条件表达式时，慷慨地使用圆括号可以加强表达式的可读性，避免由于错误假设表达式中的运算符优先级而陷入麻烦（与任何算术运算符或关系运算符相比，&& 运算符有更低的优先级，但它比 || 运算符的优先级要高）。在表达式中还应该使用空格，以加强表达式的可读性。在 && 和 || 运算符两旁插入空格，可以在视觉上把使用这些运算符连接起来的表达式与运算符分隔开。

为了在实际的例子中说明复合条件测试的用途，让我们编写一个程序来测试某个年份是不是闰年。我们知道，如果某个年份能被 4 整除，它就是闰年。然而，你可能没有认识到，能被 100 整除的年份并不是闰年，除非它能同时被 400 整除。

请思考一下，如何判断这样的条件。首先，计算某个年份除以 4、100 和 400 所得的余数，将这些值分别赋给合适的名称变量，如 `rem_4`、`rem_100` 和 `rem_400`。然后，可以继续测试这些余数，以确定是否满足闰年的标准。

如果重新表达上述闰年的定义，可将它表示为：如果某个年份能被 4 整除，但不能被 100 整除，或某个年份能被 400 整除，这一年就是闰年。稍后思考一下最后一个句子，并自己验证这个句子等价于前面陈述的定义。现在，已经用这些条件重新阐述了闰年的定义，将这些句子转换成以下程序语句的过程变得相对简单和直观，语句如下：

```
if ( (rem_4 == 0 && rem_100 != 0) || rem_400 == 0 )
    NSLog(@"It's a leap year.");
```

子表达式

```
rem_4 == 0 && rem_100 != 0
```

两边的圆括号不是必需的，因为表达式就是以这种方式求值的，记住，`&&`比`||`有更高的优先级。

事实上，在这个具体的例子中，下面的判断运行良好：

```
if ( rem_4 == 0 && ( rem_100 != 0 || rem_400 == 0 ) )
```

如果在测试之前添加几条语句，声明变量并允许用户从终端输入年份，最终就会生成一个程序，确定某个年份是不是闰年，如代码清单 6-5 所示。

代码清单 6-5

// 确定一年是否是闰年的程序

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

int main (int argc, char * argv[])
{
    @autoreleasepool {
        int year, rem_4, rem_100, rem_400;

        NSLog(@"Enter the year to be tested: ");
        scanf ("%i", &year);

        rem_4 = year % 4;
        rem_100 = year % 100;
        rem_400 = year % 400;
```

```

    if ( (rem_4 == 0 && rem_100 != 0) || rem_400 == 0 )
        NSLog (@@"It's a leap year.");
    else
        NSLog (@@"Nope, it's not a leap year.");
}

return 0;
}

```

代码清单 6-5 输出

```

Enter the year to be tested:
1955
Nope, it's not a leap year.

```

代码清单 6-5 输出（重新运行）

```

Enter the year to be tested:
2000
It's a leap year.

```

代码清单 6-5 输出（重新运行）

```

Enter the year to be tested:
1800
Nope, it's not a leap year.

```

上述例子使用了 3 个年份：第一个（1955）并非闰年，因为它不能被 4 整除；第二个（2000）是闰年，因为它能被 400 整除；第三个（1800）并非闰年，因为它虽然能被 100 整除，但不能被 400 整除。要使测试用例完整，还应该检验能被 4 整除但不能被 100 整除的年份是否是闰年。这将作为练习，由你来完成。

我们曾提到，Objective-C 在创建表达式方面为程序员提供了极大的灵活性。例如，在前一个程序中，不必计算中间结果 `rem_4`、`rem_100` 和 `rem_400`——其实你可能直接在 `if` 语句中进行了计算，语句如下：

```
if ( ( year % 4 == 0 && year % 100 != 0 ) || year % 400 == 0 )
```

用于分隔各种运算符的空格还增强了上述表达式的可读性。如果决定不添加空格并删除非必需的圆括号，就会得到以下表达式：

```
if(year%4==0&&year%100!=0||year%400==0)
```

这个表达式是完全合法的，运行结果将等价于（不管你是否相信）前面刚

刚显示的表达式。显然，这些额外的空格在帮助理解复杂表达式方面起到了很大的作用。

6.1.3 嵌套的 if 语句

讨论 if 语句的一般格式时，我们指出：如果圆括号中表达式的求值结果是 true，将执行之后的语句。如果这条程序语句是另外一条 if 语句，也是完全合法的，如以下语句：

```
if ( [chessGame isOver] == NO )
    if ( [chessGame whoseTurn] == YOU )
        [chessGame yourMove];
```

如果向 chessGame 发送的 isOver 消息所返回的值为 NO，将执行随后的语句，即另一条 if 语句。这条 if 语句又对 whoseTurn 方法返回的值与 YOU 进行比较。如果这两个值相等，将向 chessGame 对象发送 yourMove 消息。因此，只有在双方的对弈未结束，且轮到你移动棋子时，才发送 yourMove 消息。事实上，使用复合关系可将这条语句等价地表示成以下形式：

```
if ( [chessGame isOver] == NO && [chessGame whoseTurn] == YOU )
    [chessGame yourMove];
```

对于嵌套的 if 语句，更实际的例子可能是对上述例子添加了 else 子句之后的形式，语句如下：

```
if ( [chessGame isOver] == NO )
    if ( [chessGame whoseTurn] == YOU )
        [chessGame yourMove];
    else
        [chessGame myMove];
```

这条语句的执行方式和前面描述过的相同。然而，如果对弈没有结束，并且不该你移动棋子，将执行 else 子句，这将向 chessGame 发送消息 myMove。如果对弈结束，就会跳过后面整个 if 语句，包括对应的 else 子句。

要注意 else 子句如何与 if 语句对应，即用于测试 whoseTurn 方法返回的值的 if 语句，而非用于测试对弈是否结束的 if 语句。一般规则是：else 子句通常与最近的不包含 else 子句的 if 语句对应。

在上述例子中，可进一步为最外层的 if 语句添加 else 子句。这条 else 子句将在对弈结束时执行：

```

if ( [chessGame isOver] == NO )
    if ( [chessGame whoseTurn] == YOU )
        [chessGame yourMove];
    else
        [chessGame myMove];
else
    [chessGame finish];

```

当然，即使使用缩进来表明你认为是 Objective-C 语言解释语句的方式，但它并不总是与系统实际解释语句的方式一致。例如，从上述例子中删除第一个 **else** 子句：

```

if ( [chessGame isOver] == NO )
    if ( [chessGame whoseTurn] == YOU )
        [chessGame yourMove];
else
    [chessGame finish];

```

系统并不会以这种格式解释这条语句，而是使用以下形式解释这条语句：

```

if ( [chessGame isOver] == NO )
    if ( [chessGame whoseTurn] == YOU )
        [chessGame yourMove];
    else
        [chessGame finish];

```

这是因为 **else** 子句与最近的无 **else** 子句的 **if** 语句对应，在最内部的 **if** 语句不包含 **else** 子句，但外部的 **if** 语句却包含 **else** 子句的情况下，可用花括号强制表示不同的关联。花括号具有隔离 **if** 语句的作用。因此，下列语句可实现所希望的结果。

```

if ( [chessGame isOver] == NO ) {
    if ( [chessGame whoseTurn] == YOU )
        [chessGame yourMove];
}
else
    [chessGame finish];

```

6.1.4 else if 结构

你已经看到，测试两个可能的条件（任何数不是偶数，就是奇数；任何年份或者是闰年，或者不是）时，**else** 语句是如何工作的。然而，程序中要进行的判断并非总是这么明确。考虑以下任务：编写一个程序，在用户输入的数小于 0 时，显示 -1；在此数等于 0 时，显示 0；在该数大于 0 时，显示 1（这实际

上是所谓的 `sign` 函数的一种实现)。显然，在这种情况下，必须做 3 次测试，以确定输入的数是否是负数、0 或正数。此时，简单的 `if else` 结构就不再适用。当然，在这种情况下，还是可以使用 3 条独立的 `if` 语句，但这种解决方案并不总是可行的，特别是在所做的测试并非相互排斥的时候更是如此。

通过向 `else` 子句添加一条 `if` 语句，就能应对这种情况。我们曾提到过，在 `else` 子句之后的语句可以是任何合法的 Objective-C 程序语句，为什么不能是另一条 `if` 语句呢？因此，在一般情况下，可编写成以下形式：

```
if ( expression 1 )
    program statement 1
else
    if ( expression 2 )
        program statement 2
    else
        program statement 3
```

这种形式有效地扩展了 `if` 语句，使其从双值逻辑判定变成 3 个值的逻辑判定。可以用刚才展示的方式继续向 `else` 子句添加 `if` 语句，以便有效地将这种选择扩展成 n 个值的逻辑判定。

上述结构的使用相当频繁，它通常称为 `else if` 结构，其格式经常与上面显示的不同，例如：

```
if ( expression 1 )
    program statement 1
else if ( expression 2 )
    program statement 2
else
    program statement 3
```

后面这种格式提高了语句的可读性，使得 3 种选择路径更加清晰。

下面的程序通过实现前面讨论的 `sign` 函数说明了 `else if` 结构的使用。

代码清单 6-6

// 实现正负函数的程序

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

int main (int argc, char * argv[])
{
    @autoreleasepool {
        int number, sign;
```

```

NSLog(@"Please type in a number: ");
scanf ("%i", &number);

if ( number < 0 )
    sign = -1;
else if ( number == 0 )
    sign = 0;
else      // 必须是正的
    sign = 1;

NSLog(@"Sign = %i", sign);
}

return 0;
}

```

代码清单 6-6 输出

```

Please type in a number:
1121
Sign = 1

```

代码清单 6-6 输出（重新运行）

```

Please type in a number:
-158
Sign = -1

```

代码清单 6-6 输出（重新运行）

```

Please type in a number:
0
Sign = 0

```

如果输入的数小于 0，将给 sign 赋值为-1；如果此数等于 0，将给 sign 赋值为 0；否则，这个数一定大于 0，因此，给它赋值为 1。

下面的程序分析从终端输入的字符，并根据字母符号（a~z 或 A~Z）、数字（0~9）或特殊字符（其他任何字符）将其分类。要从终端读取单个字符，需要在 scanf 调用中使用格式字符%c。

代码清单 6-7

```

// 对单个字符进行分类的程序
//      从键盘输入

```

```

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    @autoreleasepool {
        char c;

        NSLog (@"Enter a single character:");
        scanf (" %c", &c);

        if ( (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') )
            NSLog (@"It's an alphabetic character.");
        else if ( c >= '0' && c <= '9' )
            NSLog (@"It's a digit.");
        else
            NSLog (@"It's a special character.");
    }

    return 0;
}

```

代码清单 6-7 输出

```

Enter a single character:
&
It's a special character.

```

代码清单 6-7 输出 (重新运行)

```

Enter a single character:
8
It's a digit.

```

代码清单 6-7 输出 (重新运行)

```

Enter a single character:
B
It's an alphabetic character.

```

注意

正如程序中的例子所示, 在 `scanf` 的格式化字符 `%c` 前插入一个空格 (如 `"%c"`), 这会引起 `scanf` “跳过” 输入中的空白字符 (例如, 换行、回车、制表符、换行符), 省略该空格会导致 `scanf` 读取的并不是你期望读取的字符。虽然在这个例子中并不是什么问题, 但当你在本章的其他例子中 (包括练习) 希望读入单个字符时都需要牢记这个用法。

读入字符之后构建的第一个测试确定了 `char` 变量 `c` 是不是字母符号。这是通过测试该字符是不是小写字母或大写字母来完成的。测试的前半部分由以下表达式组成：

```
( c >= 'a' && c <= 'z' )
```

如果 `c` 位于字符'a'和'z'之间，表达式为 `true`。也就是说，如果 `c` 是小写字母，表达式为 `true`。测试的后半部分由下面这个表达式组成：

```
( c >= 'A' && c <= 'Z' )
```

如果 `c` 位于字符'A'和'Z'之间，表达式为 `true`。也就是说，如果 `c` 是大写字母，表达式为 `true`。这些测试适用于以 ASCII 的格式存储字符的计算机系统上。

如果变量 `c` 是字母符号，且第一个 `if` 测试获得成功，将显示消息 “It's an alphabetic character”。如果测试失败，将执行 `else if` 子句。这个子句确定此字符是否为数字。注意，这个测试将字符 `c` 与字符'0'和'9'进行比较，而不是整数 0 和 9。这是因为字符是从终端读入的，并且字符'0'~'9'不同于数字 0~9。事实上，在 ASCII 中，字符'0'在内部实际表示为数字 48，字符'1'表示为 49，依此类推。

如果 `c` 是数字字符，将显示短语 “It's a digit.”；否则，如果 `c` 不是字母符号，也不是数字字符，将执行最后的 `else` 子句，并在终端显示短语 “It's a special character.”。然后，程序就会结束。

应该注意到，尽管此处使用 `scanf` 读取单个字符，但输入字符之后仍需按下 `Return` 键，以便向程序发送输入。一般来说，无论何时从终端读入数据，在按下 `Return` 键之前，程序都不会看到在数据行输入的任何数据。

假设在下一个例子中，你希望编写一个程序，允许用户使用以下形式输入简单的表达式：

```
number operator number
```

程序将计算表达式并在终端显示结果。可以识别的运算符是普通的加法、减法、乘法和除法运算符。在这里使用第 4 章“数据类型和表达式”的代码清单 4-6 中的 `Calculator` 类。因此，每个表达式都通过计算器进行计算。

以下程序使用具有多个 `else if` 子句的大型 `if` 语句来确定要执行的运算。

注意

最好使用标准库中的函数 `islower` 和 `isupper`，并彻底避免内部表示问题。为此，需要在程序中包含程序行 `#import <ctype.h>`。我们在这里这样写，目的只是为了提供例证。

代码清单 6-8

```
// 评估简单表达式的值

// 实现 Calculator 类

#import <Foundation/Foundation.h>

@interface Calculator: NSObject

// 累加器方法
-(void) setAccumulator: (double) value;
-(void) clear;
-(double) accumulator;

// 算术方法
-(void) add: (double) value;
-(void) subtract: (double) value;
-(void) multiply: (double) value;
-(void) divide: (double) value;
@end

@implementation Calculator
{
    double accumulator;
}

-(void) setAccumulator: (double) value
{
    accumulator = value;
}

-(void) clear
{
    accumulator = 0;
}

-(double) accumulator
{
    return accumulator;
}
```

```

-(void) add: (double) value
{
    accumulator += value;
}

-(void) subtract: (double) value
{
    accumulator -= value;
}

-(void) multiply: (double) value
{
    accumulator *= value;
}

-(void) divide: (double) value
{
    accumulator /= value;
}

@end

int main (int argc, char * argv[])
{
    @autoreleasepool {
        double    value1, value2;
        char      operator;
        Calculator *deskCalc = [[Calculator alloc] init];

        NSLog(@"Type in your expression.");
        scanf ("%lf %c %lf", &value1, &operator, &value2);

        [deskCalc setAccumulator: value1];
        if ( operator == '+' )
            [deskCalc add: value2];
        else if ( operator == '-' )
            [deskCalc subtract: value2];
        else if ( operator == '*' )
            [deskCalc multiply: value2];
        else if ( operator == '/' )
            [deskCalc divide: value2];

        NSLog(@"%.2f", [deskCalc accumulator]);
    }

    return 0;
}

```

代码清单 6-8 输出

```
Type in your expression.  
123.5 + 59.3  
182.80
```

代码清单 6-8 输出（重新运行）

```
Type in your expression.  
198.7 / 26  
7.64
```

代码清单 6-8 输出（重新运行）

```
Type in your expression.  
89.3 * 2.5  
223.25
```

`scanf` 调用指定了要读入到变量 `value1`、`operator` 和 `value2` 中的 3 个值。`double` 值可用 `%lf` 格式字符读入。这就是用于读入变量 `value1` 值的格式，而变量 `value1` 是表达式的第一个运算数。

接下来读入运算符（`operator`）。因为运算符是字符（`+`、`-`、`*`或`/`）而非数字，因此，需要将它读入到字符变量运算符中。`%c` 格式字符可通知系统从终端读入下一个字符。格式字符串中的空格表示输入中允许存在任意个数的空格。这样在输入这些值时可以使用空格将运算数和运算符分隔开。

读入两个值和运算符之后，程序将第一个值存储到计算器的累加器中。然后，将 `operator` 的值和 4 个允许的运算符进行比较。如果存在正确的匹配，相应的消息就会发送给计算器来执行运算。在最后一个 `NSLog` 中，检索累加器的值并显示。然后程序就会结束。

在这里，我们将简单讨论一下有关程序的“全面性（*thoroughness*）”的话题。尽管以上程序确实可以完成设置的任务，但是因为它并没有考虑用户所犯的错误，所以实际并不完善。如果用户为运算符错误地输入了一个“`?`”，将发生什么情况？执行 `if` 语句时，程序只会失败，而且终端上不会显示消息警告用户他错误地输入了表达式。

另一种被忽略的情况是，用户输入一个使用 0 作为除数的除法运算。至此，你知道在 Objective-C 中永远不要试图用某个数去除以 0。该程序应该能检查这种情况。

尽量预测程序可能失败或产生非预期结果的情形，然后采取预防性措施应付这些情况，是编写优秀而可靠的程序的必要部分。对程序运行大量的测试用例，通常可以找出没有考虑到的特定情形。但这还远远不够。编写程序时，总是询问“如果……，将发生什么情况？”，并插入必要的程序语句来适当地处理该情况，这必须成为一种自我素养。

代码清单 6-8A 即是代码清单 6-8 修改后的代码，考虑了除以 0 和输入未知运算符的情形。

代码清单 6-8A

// 评估简单表达式的值

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

```
// 接口和实现部分
```

```
// Calculator 类
```

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

```
{
```

```
    @autoreleasepool {
```

```
        double    value1, value2;
```

```
        char      operator;
```

```
        Calculator *deskCalc = [[Calculator alloc] init];
```

```
        NSLog (@\"Type in your expression.\");
```

```
        scanf ("%lf %c %lf", &value1, &operator, &value2);
```

```
        [deskCalc setAccumulator: value1];
```

```
        if ( operator == '+' )
```

```
            [deskCalc add: value2];
```

```
        else if ( operator == '-' )
```

```
            [deskCalc subtract: value2];
```

```
        else if ( operator == '*' )
```

```
            [deskCalc multiply: value2];
```

```
        else if ( operator == '/' )
```

```
            if ( value2 == 0 )
```

```
                NSLog (@\"Division by zero.\");
```

```
            else
```

```
                [deskCalc divide: value2];
```

```
        else
```

```
            NSLog (@\"Unknown operator.\");
```

```
        NSLog (@\"%.2f\", [deskCalc accumulator]);
```

```

    }

    return 0;
}

```

代码清单 6-8A 输出

```

Type in your expression.
123.5 + 59.3
182.80

```

代码清单 6-8A 输出（重新运行）

```

Type in your expression.
198.7 / 0
Division by zero.
198.7

```

代码清单 6-8A 输出（重新运行）

```

Type in your expression.
125 $ 28
Unknown operator.
125

```

输入的运算符是斜杠时，对除法而言，要执行另一个测试以确定 `value2` 是否为 0。如果是 0，将在终端显示一条适当的消息；否则，将执行除法运算并显示结果。在这种情况下，要特别注意嵌套的 `if` 语句和对应的 `else` 子句。

程序结尾的 `else` 子句会捕获所有的失败。因此，任何与测试的 4 个字符不匹配的 `operator` 值都会导致执行 `else` 子句，并导致在终端上显示“Unknown operator”。

要处理除以 0 这个问题，较好的方法是在计算除法的方法内部执行测试。因此，可将 `divide:` 方法修改成以下形式：

```

-(void) divide: (double) value
{
    if (value != 0.0)
        accumulator /= value;
    else {
        NSLog(@"Division by zero.");
        accumulator = NAN;
    }
}

```

如果 `value` 非 0，将执行除法；否则，将显示消息并将累加器设置为 NAN 值。一般来说，最好让方法处理特殊的情况，而不是依赖程序员使用方法。

6.2 switch 语句

在前一个例子中遇到的 if-else 语句串类型（其中一个变量的值与不同的值连续进行比较）在开发程序时很常见，所以，Objective-C 语言提供一种专门的语句实现这种功能。该语句的名称为 switch 语句，它的一般格式如下：

```
switch ( expression )
{
    case value1:
        program statement
        program statement
        ...
        break;
    case value2:
        program statement
        program statement
        ...
        break;
    ...
    case valuen:
        program statement
        program statement
        ...
        break;
    default:
        program statement
        program statement
        ...
        break;
}
```

括在圆括号中的 `expression` 与 `value 1`、`value 2`、`...`、`value n` 连续进行比较，后者必须是单个常量或常量表达式。某种情况下，如果发现某个 `value` 的值与 `expression` 的值相等，就执行该情况之后的程序语句。注意，包含多条这样的程序语句时，它们不必括在圆括号中。

`break` 语句表示一种特定情况的结束，并导致 switch 语句的终止。记住，每种情况的结尾都要包含 `break` 语句。如果忘记为特定的情况执行这项操作，只要执行这种情况，程序就会继续执行下一种情况。有时需要有意这么做，如

果选择这样的方式，请务必插入注释，以便将你的目的告知他人。

如果 `expression` 的值不与任何情况的值匹配，将执行一种特殊的名为 `default` 的可选情况。这在概念上等价于前一个例子中使用的 `else`。事实上，`switch` 语句的一般格式可以等价地表示成以下 `if` 语句：

```
if ( expression == value1 )
{
    program statement
    program statement
    ...
}
else if ( expression == value2 )
{
    program statement
    program statement
    ...
}
...
else if ( expression == valuen )
{
    program statement
    program statement
    ...
}
else
{
    program statement
    program statement
    ...
}
```

考虑一下前面的代码，可以将代码清单 6-8A 中的大型 `if` 语句转换成等价的 `switch` 语句，如代码清单 6-9 所示。

代码清单 6-9

```
// 评估简单表达式的值

#import <Foundation/Foundation.h>

// 接口和实现部分
// Calculator 类

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

```

double value1, value2;
char operator;
Calculator *deskCalc = [[Calculator alloc] init];

NSLog(@"Type in your expression.");
scanf ("%lf %c %lf", &value1, &operator, &value2);

[deskCalc setAccumulator: value1];

switch ( operator ) {
    case '+':
        [deskCalc add: value2];
        break;
    case '-':
        [deskCalc subtract: value2];
        break;
    case '*':
        [deskCalc multiply: value2];
        break;
    case '/':
        [deskCalc divide: value2];
        break;
    default:
        NSLog(@"Unknown operator.");
        break;
}

NSLog(@"%.2f", [deskCalc accumulator]);
}

return 0;
}

```

代码清单 6-9 输出

```

Type in your expression.
178.99 - 326.8
-147.81

```

在读入表达式之后，`operator` 的值会连续与每种情况指定的值进行比较。发现一个匹配的值时，将执行包含在这种情况下中的语句。然后，`break` 语句将终止 `switch` 语句的执行，程序也会在此处结束。如果不存在匹配 `operator` 值的情况，将执行可显示 “Unknown operator.” 的 `default` 语句。

以上程序中，`default` 情况中的 `break` 语句实际上不是必需的，因为这种情

况之后的 `switch` 中不存在任何语句。然而，记得在每种情况的结尾都包含 `break` 语句是一种良好的程序设计习惯。

编写 `switch` 语句时，应该记住任何两种情况的值都不能相同。但是，可以将多个情况的值与一组程序语句关联起来。简单地在要执行的普通语句之前列出多个情况的值（每种情况中值的前面都使用关键字 `case`，而且后面要有一个冒号）就能实现该任务。举一个例子，在以下 `switch` 语句中，如果 `operator` 等于星号或是小写字母 `x`，将执行 `multiply` 方法。

```
switch ( operator )
{
    ...
    case '*':
    case 'x':
        [deskCalc multiply: value2];
        break;
    ...
}
```

6.3 Boolean 变量

几乎所有学习程序设计的人很快就会发现自己要解决这样一个问题，即编写一个程序生成素数表。这里回顾一下：如果一个正整数 `p` 不能被 1 和它本身之外的其他任何整数整除，那它就是一个素数。第一个素数规定为 2，第二个素数是 3，因为它不能被 1 和 3 之外的任何整数整除；而 4 不是素数，因为它能被 2 整除。

可以采用几种方法来生成素数表。例如，如果你的任务是生成 50 以内的所有素数，那么最直接的（也是最简单的）算法就是生成这样一个表：它仅仅对每个整数 `p` 测试是否能被 2 到 `p-1` 间的所有整数整除。如果存在这样的整数能整除 `p`，那么 `p` 就不是素数；否则，`p` 就是素数。

代码清单 6-10 生成了 2 到 50 之间的素数列表。

代码清单 6-10

// 生成素数表的程序

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

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

```

{
    @autoreleasepool {
        int p, d, isPrime;

        for ( p = 2; p <= 50; ++p ) {
            isPrime = 1;

            for ( d = 2; d < p; ++d ) {
                if ( p % d == 0 )
                    isPrime = 0;

                if ( isPrime != 0 )
                    NSLog (@"%i ", p);
            }
        }

        return 0;
    }
}

```

代码清单 6-10 输出

```

2
3
5
7
11
13
17
19
23
29
31
37
41
43
47

```

关于代码清单 6-10，有几个要点值得注意。最外层的 for 语句建立了一个循环，周期性地遍历 2 到 50 间的整数。循环变量 *p* 表示当前正在测试以检查是否是素数的值。循环中的第一条语句将值 1 指派给变量 *isPrime*。你很快就会明白这个变量的用途。

建立的第二个循环用于将 *p* 除以从 2 到 *p*-1 间的所有整数。在该循环中，要执行一个测试以检查 *p* 除以 *d* 的余数是否为 0。如果为 0，就知道 *p* 不可能是素数，因为它能被不同于 1 和它本身的整数整除。为了表示 *p* 不再可能是素数，

可将变量 `isPrime` 的值设置为 0。

执行完最内层的循环时，测试 `isPrime` 的值。如果 `isPrime` 的值不等于 0，表示没有发现能整除 `p` 的整数。否则，`p` 肯定是素数，并显示它的值。

你可能已经注意到，变量 `isPrime` 只接受值 0 或值 1，而不是其他值。只要 `p` 还有条件成为素数，它的值就是 1。但是一旦发现它有整数约数时，它的值将设为 0，以表示 `p` 不再满足成为素数的条件。以这种方式使用的变量一般称为 Boolean 变量。通常，一个标记只接受两个不同值中的一个。此外，标记的值通常要在程序中至少测试一次，以检查它是 on (`true` 或 `YES`) 还是 off (`false` 或 `NO`)，并根据测试结果采取特定的操作。

在 Objective-C 中，标记为 `true` 或 `false` 的概念大部分被自然地转换成值 1 或 0。因此，在代码清单 6-10 的循环中，将 `isPrime` 的值设置为 1 时，实际上是将把它设置成 `true`，表示 `p` “是素数”。如果在内层 `for` 循环的执行过程中发现了一个偶数约数，`isPrime` 的值将设置为 `false`，表示 `p` 不再“是素数”。

值 1 通常用于表示 `true` 或 on 状态，而 0 用于表示 `false` 或 off 状态，这种情况并非巧合。这种表示与计算机内部单个比特的概念对应。当比特位于开状态时，它的值是 1；当它位于关状态时，值是 0。但在 Objective-C 中，有一种更令人信服的理由来支持这些逻辑值。这与 Objective-C 语言处理 `true` 和 `false` 概念的方式有关。

在开始本章的讨论时我们注意到，如果满足 `if` 语句内部指定的条件，将执行其后的程序语句。但是满足的确切含义是什么？在 Objective-C 中，满足意味着非零，而不是其他的值。因此，语句

```
if ( 100 )
    NSLog (@"This will always be printed.");
```

将导致执行 `NSLog` 语句，因为 `if` 语句中的条件（本例中仅指值 100）非零，因此，它是满足的。

本章的每一个程序中，都会用到这个概念，即“非零意味着满足”和“零意味着不满足”。这是因为，只要对 Objective-C 中的关系表达式进行求值，如果满足表达式，结果将为 1，不满足时将为 0。因此，语句

```
if ( number < 0 )
    number = -number;
```

的求值实际上按以下步骤进行：求关系表达式 `number<0` 的值。如果条件满足，即如果 `number` 小于 0，表达式的值将是 1；否则，它的值是 0。

`if` 语句测试了表达式求值的结果。如果结果非零，将执行其后的语句；否则，将略过这条语句。

前面的讨论也适用于 `for`、`while` 和 `do` 语句中的条件求值。如以下语句中复合关系表达式的求值也按前面指出的方法进行：

```
while ( char != 'e' && count != 80 )
```

如果指定的两个条件都满足，结果是 1；如果有任何一个条件不满足，求值的结果就是 0。然后，检查求值的结果。如果结果为 0，`while` 循环就会终止；否则，它会继续执行。

回到代码清单 6-10 和标记的概念上，使用表达式测试标记的值是否为 `true`，这在 Objective-C 中是相当合法的，例如，语句

```
if ( isPrime )
```

等价于

```
if ( isPrime != 0 )
```

为方便测试标记的值是否为 `false`，需要使用逻辑非 (!) 运算符。在以下表达式中使用逻辑非运算符来测试 `isPrime` 的值是否为 `false`（这条语句可读为“如果非 `isPrime`”）。

```
if ( ! isPrime )
```

一般来说，如下表达式

```
! expression
```

用于对 `expression` 的逻辑值求反。因此，如果 `expression` 为 0，逻辑非运算符将产生 1；如果 `expression` 的求值结果为非零，逻辑非运算符就会产生 0。

可以使用逻辑非运算符轻易地翻转标记的值，如在以下表达式中：

```
my_move = ! my_move;
```

正如你期望的，这种运算符的优先级和一元运算符相同。这意味着与所有的二元算术运算符和关系运算符相比，它的优先级较高。因此，要测试变量 `x` 的值是否不小于变量 `y` 的值，如在

```
! ( x < y )
```

中，圆括号是必需的，它用于确保表达式的正确求值。当然，也可将上述语句等价地表示为

```
x >= y
```

Objective-C 中有两个内置的特性，可以使 Boolean 变量的使用更容易。一种是特殊类型 **BOOL**，它可以用于声明值非真即假的变量。另一种是预定义的值 **YES** 和 **NO**。在程序中使用这些预定义的值可使它们更易于编写和读取。下面是使用这些特性重写代码清单 6-10。

注意

类型 **BOOL** 其实是由一种称为预处理程序的机制添加的。

代码清单 6-10A

```
// 生成素数表的程序
// 第二个版本使用 BOOL 类型和预定义的值

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    @autoreleasepool {
        int    p, d;
        BOOL   isPrime;

        for ( p = 2; p <= 50; ++p ) {
            isPrime = YES;

            for ( d = 2; d < p; ++d )
                if ( p % d == 0 )
                    isPrime = NO;

            if ( isPrime == YES )
                NSLog ("%i ", p);
        }
    }
    return 0;
}
```

代码清单 6-10A 输出

```
2
3
5
```

7
11
13
17
19
23
29
31
37
41
43
47

在系统库中有许多方法，要么返回一个 **BOOL** 类型的值，要么多个方法参数中有一个是这种类型的，在后续篇幅中将看到更多这样的例子。

6.4 条件运算符

Objective-C 语言中最不寻常的运算符很可能就是条件运算符。与 Objective-C 中其他所有的运算符（它们不是一元运算符就是二元运算符）不同，条件运算符是三元运算符。也就是说，它要接受 3 个运算数，用于表示这种运算符的两个符号是问号（**?**）和冒号（**:**）。第一个运算数放在问号（**?**）之前，第二个运算数放在问号（**?**）与冒号（**:**）之间，而第三个运算符放在冒号（**:**）之后。

条件运算符的一般格式为：

```
condition ? expression1 : expression2
```

在这个语法中，**condition** 表示一个表达式，通常是关系表达式，只要遇到关系运算符，Objective-C 系统就会先对它求值。如果 **condition** 求值的结果是 **true**（就是说，它是非零的），将执行 **expression1**，而且其结果将成为该运算的结果。如果 **condition** 的求值结果为 **false**（即它是零），将执行 **expression2**，而且它的结果会成为该运算的结果。

条件表达式最常用在根据条件将两个值中的一个指派给变量时。例如，假设有一个整型变量 *x* 和另一个整型变量 *s*，如果希望在 *x* 小于 0 时将 -1 指派给 *s*，若不小于 0，则将 *x*² 的值指派给 *s*，可以写成以下语句：

```
s = ( x < 0 ) ? -1 : x * x;
```

执行上述语句时, 首先要测试条件 $x < 0$ 。通常, 在条件表达式两边放置圆括号来增强语句的可读性。虽然, 这一般不是必需的, 因为条件运算符的优先级非常低, 实际上, 它低于其他所有运算符的优先级, 但赋值运算符和逗号运算符除外。

如果 x 的值小于 0, 就会求问号 (?) 之后表达式的值, 该表达式仅是常量整数值 -1, 它将在 x 小于 0 时赋给变量 s 。

如果 x 的值不小于 0, 将求冒号 (:) 之后的表达式的值, 并将该值赋值给 s 。因此, 如果 x 大于或等于 0, 将把 $x*x$ (或 x^2) 的值赋给 s 。

作为使用条件运算符的另一个例子, 使用以下语句将 a 和 b 的最大值指派给变量 `max_value`:

```
max_value = ( a > b ) ? a : b;
```

如果在冒号 (:) 之后使用表达式 (即 “else” 部分) 中包含的另一个条件运算符, 就可以达到 `else if` 子句的效果。例如, 代码清单 6-6 中实现的 `sign` 函数可以使用两个条件运算符写到一个程序行上, 语句如下:

```
sign = ( number < 0 ) ? -1 : (( number == 0 ) ? 0 : 1);
```

如果 `number` 小于 0, 将给 `sign` 赋值 -1; 如果 `number` 等于 0, 将给 `sign` 赋值 0; 否则, 将给它赋值 1。上面的表达式中, “else” 部分两旁的圆括号实际上是不必要的。这是因为条件运算符是从右到左结合的, 这意味着在单个表达式中可以使用多个运算符, 如在语句

```
e1 ? e2 : e3 ? e4 : e5
```

中, 条件运算符从右到左结合, 因此, 用以下形式求值:

```
e1 ? e2 : ( e3 ? e4 : e5 )
```

条件表达式并非必须用在赋值运算符的右边, 它们可以用在能使用的表达式的任何位置。这意味着可以在没有将变量 `number` 的符号指派给变量的情况下使用 `NSLog` 语句来显示它, 语句如下:

```
NSLog(@"Sign = %i", ( number < 0 ) ? -1
                               : ( number == 0 ) ? 0 : 1);
```

Xcode 支持如下条件运算符的非 ANSI 扩展的语句:

```
condition ? : expression
```

在这个语句中，对 `condition` 求值，若为 `true`，则返回值为 `condition`。若 `condition` 为 `false`，则操作的返回值为 `expression`。

逻辑上，这个语句等价于如下：

```
condition ? condition : expression
```

这是一个当 `index` 非零时设置 `result` 的值为 `index`，否则设置它为-1 的例子：

```
Result = index ? : -1;
```

使用 Objective-C 编写预处理程序宏指令时，条件运算符非常便利。详细内容将在第 12 章“预处理程序”介绍。

6.5 练习

1. 编写一个程序，请求用户在终端输入两个整数。测试这两个值，确定第一个数是否可以被第二个数整除，然后在终端显示一条适当的消息。
2. 即使输入非法运算符或试图除以 0 时，代码清单 6-8A 也会显示累加器中的值。请修改该程序。
3. 修改 `Fraction` 类的 `print` 方法，使其同样可以显示整数（因此，分数 5/1 可以只显示成 5）。再修改这个方法，使其能将分子是 0 的分数显示成 0。
4. 编写一个程序，使其作为简单的打印计算器。该程序应该允许用户输入以下形式的表达式：

```
number operator
```

程序还应该能识别以下运算符：

```
+ - * / S E
```

其中，`S` 运算符通知程序将累加器设置成输入的数字模式，而 `E` 运算符通知程序终止运行。使用输入的数字作为第二个运算数对累加器的内容执行算术运算。以下是显示程序应该如何运算的示例操作：

```
Begin Calculations
10 S          Set Accumulator to 10
= 10.000000   Contents of Accumulator
2 /          Divide by 2
= 5.000000    Contents of Accumulator
55 -         Subtract 55
= -50.000000
```

```

100.25 S           Set Accumulator to 100.25
= 100.250000
4 *               Multiply by 4
= 401.000000
0 E               End of program
= 401.000000
End of Calculations.

```

确保这个程序可以检测除数为零的情况，还能对未知的运算符进行检查。使用代码清单 6-8 中开发的 **Calculator** 类来执行运算。注意：记住在使用 **scanf** 的格式化字符串中使用空格字符（如 “%f %c”）来跳过输入中的空白字符。

5. 开发代码清单 5-9 用于翻转从终端输入数的各个位。然而，如果输入负数，这个程序就不能很好地运行。找出这种情况下发生了什么事情，然后修改这个程序，以便正确地处理负数。“正确地处理”指的是如果输入数-8645，程序的输出将是 5468-。
6. 编写一个程序，用于接受从终端输入的整数，提取并用英语显示这个数的每一个数字。

因此，如果用户输入 932，程序就会显示以下内容：

```

nine
three
two

```

（记住，用户只输入一个 0 时，将显示 **zero**。）注意：这个练习很难！

7. 代码清单 6-10 存在几个低效的方面。其中一个低效的方面是由检查偶数引起的。因为任何大于 2 的偶数显然不能是素数，因此，可以简单地略过偶数作为可能的素数和可能的除数。内层的 **for** 循环也是低效的，因为始终使用 $2 \sim p-1$ 之间的所有 d 值除以 p 。如果在 **for** 循环的条件中添加用于判断 **isPrime** 值的测试，可以避免这种低效性。使用这种方式时，只要没有发现除数，而且 d 的值小于 p ，**for** 循环就将继续执行。修改代码清单 6-10，将这两种修改合并到一起，然后运行程序以验证它的运算。

7 类

在本章中，你将继续学习如何使用类及如何编写方法。还将用到以前章节中所学的一些概念，例如，程序循环、选择判断和使用表达式。首先，讨论一下将程序分成多个文件，以便使较大的程序更容易处理。

7.1 分离接口和实现文件

现在，是时候将类的声明和定义放在单独的文件中了。

如果你正在使用 Xcode，可新建一个称为 `FractionTest` 的项目。在文件 `main.m` 中输入以下程序。

代码清单 7-1 主测试程序：main.m

```
#import "Fraction.h"

int main (int argc, char * argv[])
{
    @autoreleasepool {
        Fraction *myFraction = [[Fraction alloc] init];

        // 设置分数为 1/3

        [myFraction setNumerator: 1];
        [myFraction setDenominator: 3];

        // 显示分数

        NSLog (@"The value of myFraction is:");
        [myFraction print];
    }

    return 0;
}
```


}

注意，该文件未包括 `Fraction` 类的定义。然而，它确实导入了一个称为 `Fraction.h` 的文件。

通常，类的声明（即，`@interface` 部分）要放在它自己的名为 `class.h` 的文件中。而类的定义（即，`@implementation` 部分包含的代码）通常放在相同名称的文件中，但扩展名要使用 `.m`。所以，把 `Fraction` 类的声明放到 `Fraction.h` 文件中，把 `Fraction` 类的定义放到 `Fraction.m` 文件中。

要在 Xcode 中完成该工作，在 `File` 菜单中选择 `New File`；在左侧窗格中，选择 `Cocoa Touch`；在右上窗格中，选择 `Objective-C class`。现在窗口显示如图 7.1 所示。

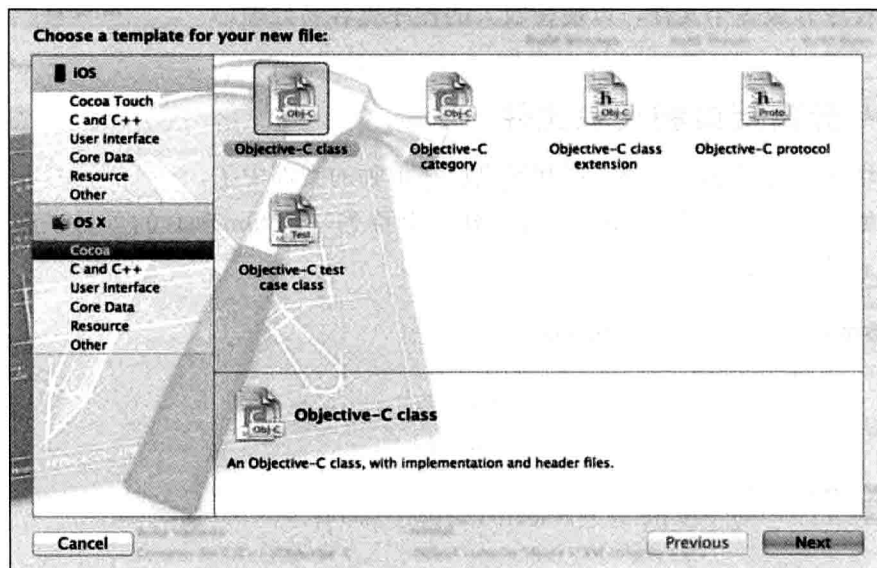


图 7.1 Xcode 新建文件菜单

单击 `Next`，在 `class` 中输入 `Faction` 作为你的类名，并在 `Subclass of` 中输入 `NSObject`（如图 7.2 所示）。再次单击 `Next`。

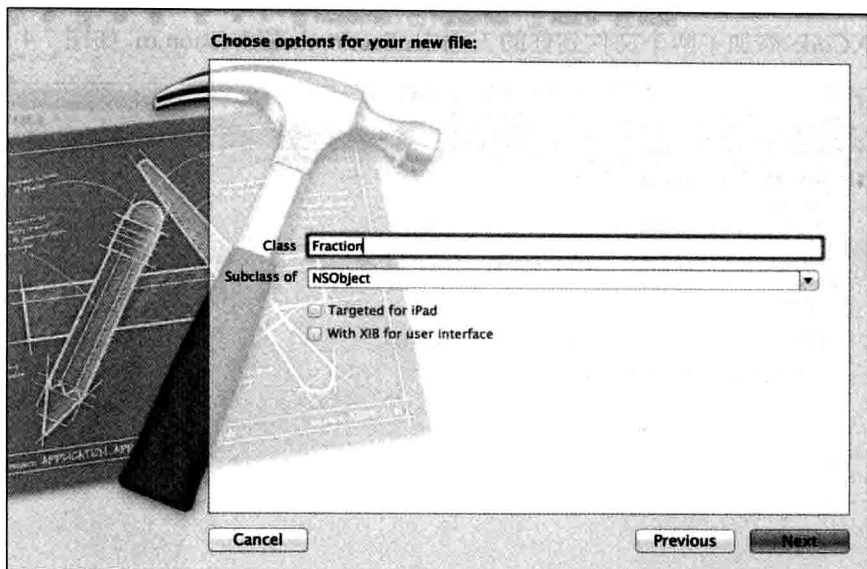


图 7.2 在项目中添加新类

你将得到一个模拟表，类似图 7.3，你可以放着其他的東西不管，直接单击 Create。

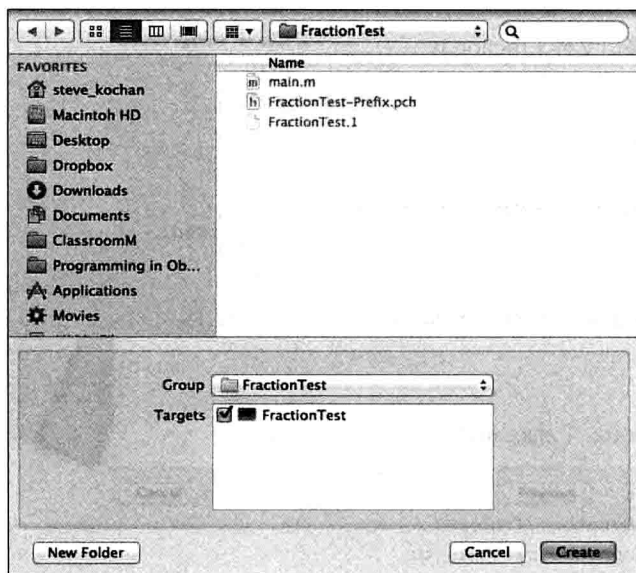


图 7.3 Xcode 为新类创建了文件

XCode 添加了两个文件到你的工程中: Fraction.h 和 Fraction.m, 如图 7.4 所示。

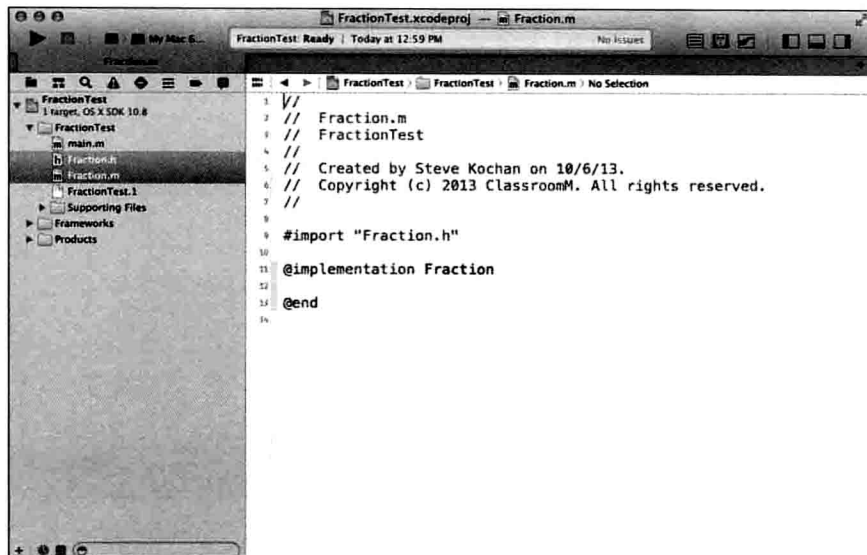


图 7.4 Xcode 为新类创建了文件

在文件 Fraction.h 中, 现在可输入 Fraction 类的接口部分, 如代码清单 7-1。

代码清单 7-1 接口文件 Fraction.h

```
//
// Fraction.h
// FractionTest
//
// Created by Steve Kochan on 10/6/13.
// Copyright (c) 2013 ClassroomM. All rights reserved.
//

#import <Foundation/Foundation.h>

// Fraction 类

@interface Fraction : NSObject

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
-(int) numerator;
-(int) denominator;
-(double) convertToNum;
```

@end

接口文件告知编译器(并告知其他程序员,以后你将学到这些内容)Fraction 类的外观特征,它包含 6 个实例方法: `print`、`setNumerator:`、`setDenominator:`、`numerator`、`denominator`、`convertToNum`。前 3 个方法无返回值,第 4、5 个方法返回 `int` 值,最后一个方法返回 `double` 值。`setNumerator:`和 `setDenominator:`方法各接收一个整型参数。

Fraction 类的实现细节需要输入到文件 Fraction.m 中。

代码清单 7-1 实现文件: Fraction.m

```
//
// Fraction.m
// FractionTest
//
// Created by Steve Kochan on 10/6/13.
// Copyright (c)2013 ClassroomM. All rights reserved.
//
#import "Fraction.h"

@implementation Fraction
{
    int numerator;
    int denominator;
}

-(void) print
{
    NSLog(@"%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}

-(int) numerator
{
    return numerator;
}
```

```
}

-(int) denominator
{
    return denominator;
}

-(double) convertToNum
{
    if (denominator != 0)
        return (double) numerator / denominator;
    else
        return NAN;
}

@end
```

注意，使用以下语句将接口文件导入实现文件中：

```
#import "Fraction.h"
```

这样做的目的是，将接口和实现的部分分别放入两个文件。编译器会单独编译每个文件。当编译器处理包含实现部分的文件时（如 `Fraction.m`），需要知道接口部分的信息（如方法的名字和参数的类型）。通过导入 `.h` 文件，使编译器知道为 `Fraction` 类声明的类和方法，同时还能确保这两个文件的一致性。

需要注意的一件事：导入的文件要用一对引号引起来，而不是 `<Foundation/Foundation.h>` 中的 “<” 和 “>” 字符。双引号适用于本地文件（你自己创建的文件），而不是系统文件，这样就通知编译器在哪里能够找到指定的文件。使用双引号时，编译器一般会首先在项目目录寻找指定文件，然后转到其他位置查找。如果有必要，可以指定编译器要查找的不同位置。

还要注意，测试程序 `main.m`（本章的开始部分有提到）包括接口文件 `Fraction.h`，而不包括实现文件 `Fraction.m`。当在其他文件中需要使用一个类时，编译器通过类的接口部分获取所需要的全部信息。类的实现部分包含方法的实际代码，当你编译你的程序时，Xcode 会处理引入了其他代码的情况。接口文件包含类的公开信息，即能够与这个类的使用者共享一些信息。另一方面，实现部分包含的是私有信息，即实例变量和代码。

注意

事实上，代码确实存储在某处，正如框架或库。

现在，该程序分成了 3 个独立的文件。对于小程序例子而言，这似乎要花费很多工作，但是，着手处理较大的程序并和其他程序员共享类声明时，这种实用性就会变得十分明显。

现在可以编译并运行程序，方法与以前使用的一样：选择 **Product** 菜单中的 **Run**，或者单击工具栏中的 **Play** 图标。

如果使用命令行编译程序，要为 Objective-C 编辑器提供两个“.m”文件名。如果使用 Clang，则命令行可能如下：

```
clang -fobjc-arc -framework Foundation Fraction.m FractionTest.m -o FractionTest
```

这将构建一个名为 **FractionTest** 的可执行文件。下面就是运行该程序的输出结果。

代码清单 7-1 FractionTest 输出

```
The value of myFraction is:
1/3
```

7.2 合成存取方法

从 Objective-C 2.0 开始，可自动生成设值方法和取值方法（统称为存取方法）。我们到目前为止都没有介绍如何实现，原因是知道如何自己编写这些方法非常重要。然而，该语言提供了一个很方便的功能，所以现在应该学习如何充分利用这个功能了。

第一步是在接口部分中使用 `@property` 指令标识属性。这些属性的命名与实例变量相同，尽管不是必须要这样做的。在 **Fraction** 类中，两个实例变量 **numerator** 和 **denominator** 都属于此类属性。下面是新的接口部分，其中添加了新的 `@property` 指令。

```
@interface Fraction : NSObject

@property int numerator, denominator;

-(void) print;
-(double) convertToNum;
@end
```

注意，我们没有包括下列设值方法和取值方法的定义：**numerator**、

`denominator`、`setNumerator`和 `setDenominator`。我们要让 Objective-C 编译器自动生成或合成这些方法。如何完成呢？只需在实现部分使用`@synthesize`指令即可，代码如下：

```
#import "Fraction.h"

@implementation Fraction

@synthesize numerator, denominator;

-(void) print
{
    NSLog ("%i/%i", numerator, denominator);
}

-(double) convertToNum
{
    if (denominator != 0)
        return (double) numerator / denominator;
    else
        return NAN;
}

@end
```

如果使用了`@property`指令，就不需要在实现部分声明相应的实例变量。当然也可以再声明相应的实例变量，但是那不是必须要做的，编译器会有一些提示。

下面这行语句告诉 Objective-C 编译器，为两个属性生成一对设值方法和取值方法：

```
@synthesize numerator, denominator;
```

通常，如果有称为 `x` 的属性，那么在实现部分包括以下行会导致编译器自动实现一个取值方法 `x` 和一个设值方法 `setX`：

```
@synthesize x;
```

即使此处看起来并非什么大事，但是让编译器完成这项工作是值得的，因为生成的存取方法是高效的，并且在使用多个核心的多台机器上，使用多线程时也可正常运行。

注意

你并不需要使用@synthesize 指令，使用@property 指令就足够了。编译器会自动为你生成 setter 和 getter。但是，注意，如果你不试用@synthesize，那么编译器生成的实例变量会以下划线(_) 字符作为其名称的第一个字符。

现在回到代码清单 7-1，并按照上述内容对接口和实现部分进行更改，为你合成存取方法。确定最终的输出与没有任何变化的 main.m 是相同的。

7.3 使用点运算符访问属性

Objective-C 语言允许你使用非常简便的语法访问属性。要获得 myFraction 中存储的 merator 的值，可使用以下语句：

```
[myFraction numerator]
```

这会向 myFraction 对象发送 numerator 消息，从而返回所需的值。在 Objective-C 中也可以使用点运算符编写以下等价的表达式：

```
myFraction.numerator
```

一般格式为如下：

```
instance.property
```

还可使用类似的语法进行赋值：

```
instance.property = value
```

这等价于编写以下表达式：

```
[instance setProperty: value]
```

在代码清单 7-1 中，使用以下两行代码将分数的 numerator 和 denominator 设置为 1/3：

```
[myFraction setNumerator: 1];  
[myFraction setDenominator: 3];
```

下面是两行等价的代码：

```
myFraction.numerator = 1;  
myFraction.denominator = 3;
```

我们为合成方法使用这些新功能，并在本书后面部分都使用这种方法访问

属性。

代码清单 7-2 演示如何像刚刚描述的那样使用点运算符。

代码清单 7-2 主测试代码：main.m 演示使用点运算符

```
#import "Fraction.h"
int main (int argc, char * argv[])
{
    @autoreleasepool {
        Fraction *myFraction = [[Fraction alloc] init];

        // set fraction to 1/3 using the dot operator

        myFraction.numerator = 1;
        myFraction.denominator = 3;

        // display the fraction's numerator and denominator

        NSLog(@"The numerator is %i, and the denominator is %i",
            myFraction.numerator, myFraction.denominator);
    }
    return 0;
}
```

代码清单 7-2 FractionTest 输出

```
The numerator is 1, and the denominator is 3
```

需要指出的是，也可以对自定义的方法使用点运算符，不仅仅是使用在 `synthesize` 上。如果有一个取值方法称为 `numerator`，仍然能够在程序中编写表达式 `myFraction.numerator`，尽管 `numerator` 并未定义为属性。

注意

前面的讨论是基于语句的语法构成上是否正确，如语句 `myFraction.print`，并未考虑编码风格是否良好。点运算符通常用在属性上，用于设置或取得实例变量的值。做其他工作的方法通常不是由点运算符执行的，而是使用传统的方括号形式的消息表达式作为首选的语法。

注意

使用合成（`synthesize`）的存取方法，属性名称的前面不要以 `new`、`alloc`、`copy` 或者 `init` 这些词开头。这与编译器的一些假定有关，因为编译器会合成

相应的方法，在第17章“内存管理和引用计数”会有更详细的描述。

注意

如果在程序7.2从Fraction.m中省略@synthesize指令，那么编译器会自动分别命名底层的实例变量_numerator和_denominator。在这种情况下，程序7.2的实现部分将必须进行修改，以正确执行。

7.4 具有多个参数的方法

让我们继续使用Fraction类，并对它做一些补充。你已经定义了6个方法，如果有一个方法只用一条消息同时设置numerator和denominator，就太好了。通过列出每个连续的参数并用冒号将其连起来，就可以定义一个接收多个参数的方法。用冒号连接的参数将成为这个方法名的一部分。例如，方法名addEntryWithName:andEmail:表示接收两个参数的方法，这两个参数可能是姓名和电子邮件地址。方法addEntryWithName:andEmail:andPhone:是接收3个参数的方法：一个姓名、一个电子邮件地址和一个电话号码。

同时设置numerator和denominator的方法可以命名为setNumerator:andDenominator:，能采用以下形式：

```
[myFraction setNumerator: 1 andDenominator: 3];
```

这种方法还不错。实际上，这是命名方法的首选方式。但是必须为方法指定更易阅读的名称。例如，觉得setTo:over:如何？这个名称乍一看并没有什么吸引力，通过将myTraction设置为1/3，比较这个名称和前面的名称：

```
[myFraction setTo: 1 over: 3];
```

笔者认为这个名称的可读性更强，但是，究竟选择哪种命名方式由你决定（实际上，有些人更喜欢第一种命名方式，因为它明确地引用了类中的实例变量名称）。再者，选择好的方法名对整个程序的可读性是很重要的。写出实际的消息表达式可帮助你找出一个最好的。

让我们应用这种新方法。首先，在接口文件中添加setTo:over:声明，如代码清单7-3所示。

代码清单 7-3 接口文件：Fraction.h

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

// 定义 Fraction 类

@interface Fraction : NSObject

@property int numerator, denominator;

-(void) print;
-(void) setTo: (int) n over: (int) d;
-(double) convertToNum;
@end
```

然后，在实现文件中添加新方法定义。

代码清单 7-3 实现文件：Fraction.m

```
#import "Fraction.h"

@implementation Fraction

@synthesize numerator, denominator;

-(void) print
{
    NSLog(@"%i/%i", numerator, denominator);
}

-(double) convertToNum
{
    if (denominator != 0)
        return (double) numerator / denominator;
    else
        return NAN;
}

-(void) setTo: (int) n over: (int) d
{
    numerator = n;
    denominator = d;
}

@end
```

新的 setTo:over:方法仅接收两个整型参数 n 和 d, 并把它们赋值给该分数对应的域 numerator 和 denominator。

下面是新方法的测试程序。

代码清单 7-3 测试文件: main.m

```
#import "Fraction.h"

int main (int argc, char * argv[])
{
    @autoreleasepool {
        Fraction *aFraction = [[Fraction alloc] init];

        [aFraction setTo: 100 over: 200];
        [aFraction print];

        [aFraction setTo: 1 over: 3];
        [aFraction print];
    }

    return 0;
}
```

代码清单 7-3 输出

```
100/200
1/3
```

7.4.1 不带参数名的方法

创建方法名时, 参数名实际上是可选的。例如, 可以用如下语句声明一个方法:

```
-(int) set: (int) n: (int) d;
```

注意, 和前面的例子不同, 这个方法的第二个参数没有命名。这个方法名为 `set::`, 两个冒号表示这个方法仍然有两个参数, 虽然没有对所有的参数命名。

要调用 `set::` 方法, 可以使用冒号作为参数分隔符, 语句如下:

```
[aFraction set: 1 : 3];
```

在编写新方法时, 省略参数名不是一种好的编程风格, 因为它使程序很难读懂并且很不直观, 特别是当使用的方法参数特别重要时, 更是如此。

7.4.2 关于分数的操作

继续探讨上面提到的 `Fraction` 类。首先, 编写一个方法, 将一个分数与另

一个分数相加, 将一个方法命名为 `add:`, 并且把一个分数作为参数。这个新方法的声明如下:

```
-(void) add: (Fraction *) f;
```

注意参数 `f` 的声明

```
(Fraction *) f
```

这条语句说明 `add:` 方法的参数是 `Fraction` 类对象的一个引用。星号是必需的, 所以声明

```
(Fraction) f
```

是不正确的。你将把一个分数作为参数传递给 `add:` 方法, 而这一方法将其添加到消息的接收者中。所以消息表达式

```
[aFraction add: bFraction];
```

将 `Fraction bFraction` 和 `Fraction aFraction` 相加。

作为快速的数学复习, 分数 a/b 和 c/d 相加, 你要执行如下计算:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

下面是 `@implementation` 部分的新方法代码:

```
// 添加 Fraction 到消息接收者
```

```
-(void) add: (Fraction *) f
{
    // 添加两个分数
    // a/b + c/d = ((a*d) + (b*c)) / (b * d)

    numerator = numerator * f.denominator + denominator * f.numerator;
    denominator = denominator * f.denominator;
}
```

不要忘了通过域 `numerator` 和 `denominator` 可以指定 `Fraction` 作为消息的接收者。也就是说, 在 `add:` 方法中, 你可以直接指定这个对象的实例变量并直接通过名字发送消息。

然而, 不能用这种方法直接指定参数 `f` 的实例变量。而是通过名字 `f` 标识这个对象, 然后通过对 `f` 应用点运算符来获得实例变量。

所以 `add:` 方法的第一个语句

```
numerator = numerator * f.denominator + denominator * f.numerator;
```

表明用第一个分数的分子（消息的接收者）与参数的分母相乘（`numerator * f.denominator`），并加上接收者的分母与参数分子的结果（`denominator * f.numerator`）。最终的和存储在接收器的分子上。

请重新阅读前面的段落就能够完全理解操作符的执行。关键是理解接收者的引用和参数的引用。

假设你已将新 `add` 方法的声明和定义添加到接口和实现文件中。代码清单 7-4 是示例测试和输出。

代码清单 7-4 接口文件：Fraction.h

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

// Define the Fraction class

@interface Fraction : NSObject

@property int numerator, denominator;

-(void) print;
-(void) setTo: (int) n over: (int) d;
-(double) convertToNum;
-(void) add: (Fraction *) f;

@end
```

代码清单 7-4 实现文件：Fraction.m

```
#import "Fraction.h"

@implementation Fraction

@synthesize numerator, denominator;

-(void) print
{
    NSLog(@"%i/%i", numerator, denominator);
}

-(double) convertToNum
{
    if (denominator != 0)
        return (double) numerator / denominator;
    else
        return NAN;
}
```

```

}

-(void) setTo: (int) n over: (int) d
{
    numerator = n;
    denominator = d;
}

// add a Fraction to the receiver
- (void) add: (Fraction *) f
{
    // To add two fractions:
    //  $a/b + c/d = ((a*d) + (b*c)) / (b * d)$ 
    numerator = numerator * f.denominator + denominator * f.numerator;
    denominator = denominator * f.denominator;
}
@end

```

代码清单 7-4 测试文件: FractionTest.m

```

#import "Fraction.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        Fraction *aFraction = [[Fraction alloc] init];
        Fraction *bFraction = [[Fraction alloc] init];

        // 设置两个分数为 1/4 和 1/2, 并将它们加到一起

        [aFraction setTo: 1 over: 4];
        [bFraction setTo: 1 over: 2];

        // 打印结果

        [aFraction print];
        NSLog (@"+");
        [bFraction print];
        NSLog (@"=");

        [aFraction add: bFraction];
        [aFraction print];
    }

    return 0;
}

```

代码清单 7-4 输出

```
1/4
+
1/2
=
6/8
```

这个测试程序简单易懂。两个 `Fraction`，名为 `aFraction` 和 `bFraction`，各自分配空间和初始化，接着分别被赋值为 $1/4$ 和 $1/2$ 。然后将 `Fraction bFraction` 与 `Fraction aFraction` 相加，最后显示相加的结果。注意 `add` 方法将消息对象的参数相加，所以对象被修改了。在 `main` 程序尾打印 `aFraction` 的值时将证明这一点。应该注意，要在调用 `add` 方法之前显示 `aFraction` 的值，这样可以在 `add` 方法改变 `aFraction` 的值之前显示其原值。在本章的结尾，你将重新定义 `add` 方法，以便让 `add` 方法不影响其接收者的值。

7.5 局部变量

你也许注意到 $1/4$ 和 $1/2$ 相加的结果显示为 $6/8$ ，而不是期望的 $3/4$ ，这是因为加法函数只执行算术，而不做其他处理，它不会将结果相约。所以，我们继续练习如何编写有关分数运算的新方法。编写一个新的 `reduce` 方法，将分数相约到它的最简形式。

回顾一下中学学过的数学课程，化简分数需要找出分子和分母都能够整除的最大数，然后将它们除以这个数。从技术上讲，需要找出分子和分母的最大公约数（gcd）。代码清单 5-7 已经知道如何得到它。稍微回忆一下也许就会想起这个程序。

根据这些算法，可以编写出名为 `reduce` 的新方法。

```
- (void) reduce
{
    int u = numerator;
    int v = denominator;
    int temp;

    while (v != 0) {
        temp = u % v;
        u = v;
        v = temp;
    }
}
```



```

    }

    numerator /= u;
    denominator /= u;
}

```

注意，这个 `reduce` 方法中有一些新东西：它声明了 3 个整型变量 `u`、`v` 和 `temp`。这些变量是局部变量，意思是它们的值只在 `reduce` 方法运行时才存在，并且只能在定义它们的方法中访问。从这个意义上说，它们类似于在 `main` 函数中定义的变量，这些变量对 `main` 来说是局部变量，可以在 `main` 函数中直接进行访问。你定义的其他方法都不能访问 `main` 中的局部变量。

局部变量是基本的 C 数据类型，并没有默认的初始值，所以在使用前要先赋值。`Reduce` 方法中有 3 个局部变量在使用之前被赋值，所以没有问题。局部对象变量默认初始化为 `nil`。和实例变量不同（它们在多次方法调用时保持自己的值），这些局部变量没有记忆力。也就是说，当方法返回时，这些变量的值都消失了。每次调用方法时，该方法中的局部变量（如果有的话）都使用变量声明重新初始化一次。

7.5.1 方法的参数

方法的参数名也是局部变量。执行方法时，通过方法传递的任何参数都被复制到局部变量中。因为方法使用参数的副本，所以不能改变通过方法传递的原值。这是一个重要概念。假设有个名为 `calculate:` 的方法，其定义如下：

```

-(void) calculate: (double) x
{
    x *= 2;
    ...
}

```

并假设使用以下消息表达式来调用它：

```
[myData calculate: ptVal];
```

执行 `calculate` 方法时，`ptVal` 变量所含的任何值都被复制到局部变量 `x`。所以，改变 `calculate:` 中 `x` 的值，对 `ptVal` 的值没有影响，只影响 `x` 变量的数据副本。

顺便提一下，如果参数是对象，可以更改其中的实例变量值。当你传递一个对象作为参数时，实际上是传递了一个数据存储位置的引用。正因为如此，

你才能够修改这些数据。更多内容将在下一章介绍。

7.5.2 static 关键字

在变量声明前加上关键字 `static`，可以使局部变量保留多次调用一个方法所得的值。例如，语句

```
static int hitCount = 0;
```

声明整数 `hitCount` 是一个静态变量。和其他基本数据类型的局部变量不同，静态变量的初始值为 0，所以前面显示的初始化是多余的。此外，它们只在程序开始执行时初始化一次，并且在多次调用方法时保存这些数值。

所以，下列编码序列

```
-(int) showPage
{
    static int pageCount = 0;
    ...
    ++pageCount;
    ...
    return pageCount;
}
```

可能出现在一个 `showPage` 方法中，它用于记录该方法的调用次数（在这种情况下，还可能是要打印的页数）。只在程序开始时局部静态变量设置为 0，并且在连续调用 `showPage` 方法时获得新值。

注意将 `pageCount` 设置为局部静态变量和实例变量之间的区别。对于前一种情况，`pageCount` 能记录调用 `showPage` 方法的所有对象打印页面的数目。对后一种情况，`pageCount` 变量可以计算每个对象打印的页面数目，因为每个对象都有自己的 `pageCount` 副本。

记住：只能在定义静态变量和局部变量的方法中访问这些变量。所以也只能在 `showPage` 函数中访问静态变量 `pageCount`。可以将变量的声明移到所有方法声明的外部（通常放在 `implementation` 文件的开始处），这样所有的方法都可以访问它们，如：

```
#import "Printer.h"
static int pageCount;

@implementation Printer
...

```

```
@end
```

现在，该文件中包含的所有实例或者类方法都可以访问变量 `pageCount`。
第 10 章“变量和数据类型”详细讲述了变量作用域的内容。

回到关于分数的讨论，将 `reduce` 方法的代码结合到实现文件 `Fraction.m` 中。不要忘了在接口文件 `Fraction.h` 中声明 `reduce` 方法，然后就可以在代码清单 7-5 中测试这个新方法了。下面有 3 个文件：接口文件、实现文件、测试程序文件。

代码清单 7-5 接口文件：Fraction.h

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

// 定义 Fraction 类

@interface Fraction : NSObject

@property int numerator, denominator;

-(void) print;
-(void) setTo: (int) n over: (int) d;
-(double) convertToNum;
-(void) add: (Fraction *) f;
-(void) reduce;
@end
```

代码清单 7-5 实现文件：Fraction.m

```
#import "Fraction.h"

@implementation Fraction

@synthesize numerator, denominator;

-(void) print
{
    NSLog ("%i/%i", numerator, denominator);
}

-(double) convertToNum
{
    if (denominator != 0)
        return (double) numerator / denominator;
    else
        return NAN;
}
```

```

-(void) setTo: (int) n over: (int) d
{
    numerator = n;
    denominator = d;
}

// 添加 Fraction 到消息接收者

-(void) add: (Fraction *) f
{
    // 添加两个分数:
    //  $a/b + c/d = (a*d) + (b*c) / (b * d)$ 

    numerator = numerator * f.denominator + denominator * f.numerator;
    denominator = denominator * f.denominator;
}

-(void) reduce
{
    int u = numerator;
    int v = denominator;
    int temp;

    while (v != 0) {
        temp = u % v;
        u = v;
        v = temp;
    }

    numerator /= u;
    denominator /= u;
}

@end

```

代码清单 7-5 测试文件 main.m

```

#import "Fraction.h"

int main (int argc, char * argv[])
{
    @autoreleasepool {
        Fraction *aFraction = [[Fraction alloc] init];
        Fraction *bFraction = [[Fraction alloc] init];

        [aFraction setTo: 1 over: 4]; // 设置第 1 个分数为 1/4
        [bFraction setTo: 1 over: 2]; // 设置第 2 个分数为 1/2
    }
}

```

```

    [aFraction print];
    NSLog(@"+");
    [bFraction print];
    NSLog(@"=");

    [aFraction add: bFraction];

    // 化简相加后的值并打印结果

    [aFraction reduce];
    [aFraction print];
}

return 0;
}

```

代码清单 7-5 输出

```

1/4
+
1/2
=
3/4

```

这次更好了!

7.6 self 关键字

在代码清单 7-5 中，我们决定在 `add:` 方法之外约简分数，还可以在 `add:` 中进行约简。怎么做完全是随意的。然而，如何确定要约简的分数？我们想约简哪一个分数？我们希望约简的分数就是收到 `add:` 消息的分数。

我们知道如何通过名称指明方法内的实例变量，但我们不知道如何直接指明消息的接收者。幸运的是，有办法能够做到这一点。

关键字 `self` 用来指明对象是当前方法的接收者。如果在 `add:` 消息中编写

```
[self reduce];
```

就可以对 `Fraction` 应用 `reduce` 方法，它正是你希望的 `add:` 消息的接收者。在学习本书的过程中，可以看到关键字 `self` 很有用，在 `iOS` 编程中时刻都会用到。现在，在 `add:` 方法中使用它，下面是修改后的结果：

```

- (void) add: (Fraction *) f
{

```

```

// 添加两个分数:
//  $a/b + c/d = ((a*d) + (b*c)) / (b * d)$ 

numerator = numerator * f.denominator + denominator * f.numerator;
denominator = denominator * f.denominator;

[self reduce];
}

```

这样，执行加法之后，分数被约简了。`reduce` 消息取得了发送给 `add:` 消息的接收者。所以，你在测试程序中包含了这样一行代码：

```
[aFraction add: bFraction];
```

在执行 `add:` 方法时，`self` 是作为 `aFraction` 的引用，分数会得到约简。

7.7 在方法中分配和返回对象

我们曾提到 `add:` 方法改变了接收该消息的对象值。让我们来创建一个新版本的 `add:` 函数，这个方法创建了一个新的分数来存储相加的结果。在这种情况下，需要向消息的发送者返回新的 `Fraction`。

下面是新方法 `add:` 的定义。

```

-(Fraction *) add: (Fraction *) f
{
    // 添加两个分数:
    //  $a/b + c/d = ((a*d) + (b*c)) / (b * d)$ 

    // 存储相加后的结果
    Fraction *result = [[Fraction alloc] init];

    result.numerator = numerator * f.denominator +
                        denominator * f.numerator;
    result.denominator = denominator * f.denominator;

    [result reduce];

    return result;
}

```

方法定义的第一行是

```
-(Fraction *) add: (Fraction *) f
```

这一行说明 `add:` 方法将返回一个名为 `Fraction` 的对象，并且说明它还使用

一个 `Fraction` 作为参数，这个参数将与消息的接收者相加，这个接收者同样是 `Fraction`。请注意，你需要更改接口部分，表明 `add:` 方法现在返回一个 `Fraction` 对象。

这一方法分配并初始化了一个新的 `Fraction` 对象，名为 `result`，用于存储程序中相加的结果。

和以前一样执行完加法运算之后，指定结果的分子和分母给新创建的 `Fraction` 对象约简结果之后，使用 `return` 语句将结果返回给消息的发送者。我们希望约简结果，这就是为什么消息再一次发送给了那个对象。

代码清单 7-6 测试新的 `add:` 方法。

代码清单 7-6 测试文件 `main.m`

```
#import "Fraction.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        Fraction *aFraction = [[Fraction alloc] init];
        Fraction *bFraction = [[Fraction alloc] init];

        Fraction *resultFraction;

        [aFraction setTo: 1 over: 4]; // 设置第一个分数为 1/4
        [bFraction setTo: 1 over: 2]; // 设置第二个分数为 1/2

        [aFraction print];
        NSLog (@"+");
        [bFraction print];
        NSLog (@"=");

        resultFraction = [aFraction add: bFraction];
        [resultFraction print];
    }

    return 0;
}
```

代码清单 7-6 输出

```
1/4
+
1/2
=
```

3/4

3/4

这里要做一点补充说明。首先，你定义了两个 `Fraction` 对象：`aFraction` 和 `bFraction`，并将它们的值分别设为 $1/4$ 和 $1/2$ ，然后又定义了名为 `resultFraction` 的一个 `Fraction`，这个变量将用来存储相加操作的结果。

以下代码

```
resultFraction = [aFraction add: bFraction];
```

首先发送 `add:` 消息到 `aFraction` 类，同时将类 `Fraction` 的 `bFraction` 作为它的参数。

在这个方法里，新的 `Fraction` 对象被创建并且加出了结果，该结果存储在通过方法返回的 `Fraction` 对象 `result` 中，然后将它保存在变量 `resultFraction` 中。你可能注意到你并未在 `main` 中为 `resultFraction` 创建或初始化一个 `Fraction` 对象。这是因为 `add:` 方法中已经创建了一个对象，并通过方法返回一个对象的引用，这个引用存储在 `resultFraction` 中。所以，`resultFraction` 最终保存了一个 `Fraction` 的对象引用，该对象是在 `add:` 方法中创建的。这一段是很重要的！在你完全理解它之前，值得反复回顾。

7.7.1 扩展类的定义和接口文件

你可能不需要处理分数，但是这些例子告诉你如何通过加入新方法来定义和扩展一个类。其他处理分数的人将使用你的 `Fraction.h` 接口文件，并且使用这个文件足够编写他自己的处理分数的程序。如果他需要添加新方法，通过直接扩展类定义或者定义自己的子类并添加自己的新方法直接扩展该类，可以实现该目的。下一章将学习如何实现它。

7.8 练习

1. 将下列方法加到 `Fraction` 类，以扩展关于分数的算术运算。在每个例子中都约简结果。

```
// 减去消息接收者的参数
-(Fraction *) subtract: (Fraction *) f;
// 消息接收者乘以参数
-(Fraction *) multiply: (Fraction *) f;
// 消息接收者除以参数
```



```
-(Fraction *) divide: (Fraction *) f;
```

2. 从 Fraction 类中修改 Print 方法，使之能够接收一个可选的 BOOL 参数，它表明是否应该约简分数并显示它。如果要约简它（参数为 YES），一定不要对它进行永久更改。
3. Fraction 类对负分数适用吗？例如， $-1/4$ 和 $-1/2$ 能得出正确结果吗？如果想出答案，编写测试程序进行尝试。
4. 修改 Fraction 类的 print 方法，以便显示比 1 大的分数，例如， $5/3$ 能显示为 $1\ 2/3$ 。
5. 从代码 7.2 中移除 @synthesize 指令并修改程序使新的变量命名通过编译。
6. 第 4 章“数据类型和表达式”练习 6，为复数创建一个新类名为 Complex。创建一个新的方法名为 add:，用来求两个复数之和。要将复数相加，你只需要分别对实数部分和虚数部分求和，如下：

$$(5.3 + 7i) + (2.7 + 4i) = 8 + 11i$$

add:方法可以存储并返回一个新的复数，基于如下方法描述：

```
-(Complex *) add: (Complex *) complexNum;
```

7. 第 4 章练习 6 的 Complex 类在这一章的练习 6 中进行了扩展，请单独创建接口文件 Complex.h 和实现文件 Complex.m。再创建一个单独的测试程序文件进行测试。

8

继承

本章学习面向对象编程的一个重要原理。通过学习继承的概念，学会如何使用现有类的定义，并适用于自己的应用程序。

8.1 一切从根类开始

在第3章“类、对象和方法”中学过父类的概念。父类自身也可以有父类。没有父类的类位于类层次结构的最顶层，称为根（root）类。在 Objective-C 中，允许定义自己的根类，但通常不这么做，而是希望利用现有的类。至此，我们所定义的类都属于 NSObject 根类的派生类，在接口文件中通常这样指定根类：

```
@interface Fraction: NSObject
...
@end
```

类 Fraction 是从类 NSObject 派生来的。因为 NSObject 是层次结构的最顶端（也就是它上面没有任何类），因此称为根类，如图 8.1 所示。类 Fraction 称为子或者子类（subclass）。

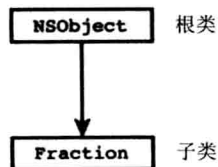


图 8.1 根类和子类

如果使用术语，可以将类称为子类和父类。同样，也可以将类称为子类和超类，你应该熟悉这两种术语。

只要定义一个新类（不是一个新的根类），都会继承一些属性。例如，很明显，父类的非私有实例变量和方法都会成为新类定义的一部分。子类可以直接访问这些方法和实例变量，就像在类定义中直接定义了这些子类一样。

需要注意的是，在子类中使用实例变量，必须先在接口部分声明，而不是像书中其他的例子中在实现部分声明。在实现部分声明和合成（synthesize）的实例变量是私有的，子类中并不能直接访问，需要明确定义或合成取值方法，才能访问实例变量的值。

举一个简单的例子，虽然不太自然，但有助于解释继承这个关键概念。下面是一个名为 **ClassA** 的对象的声明，它有一个方法 **initVar**（需要注意的是，在接口部分声明实例变量 *x* 是为了使子类能够访问到）。

```
@interface ClassA: NSObject
{
    int x;
}

-(void) initVar;
@end
```

initVar 方法简单地把 100 赋值给 **ClassA** 的实例变量：

```
@implementation ClassA
-(void) initVar
{
    x = 100;
}
@end
```

现在，再定义一个名为 **ClassB** 的类：

```
@interface ClassB: ClassA
-(void) printVar;
@end
```

声明的第一行

```
@interface ClassB: ClassA
```

说明 **ClassB** 并非 **NSObject** 的子类，而是 **ClassA** 的子类。所以，尽管 **ClassA** 的父类（或超类）是 **NSObject**，但是 **ClassB** 的父类却是 **ClassA**，如图 8.2 所示。

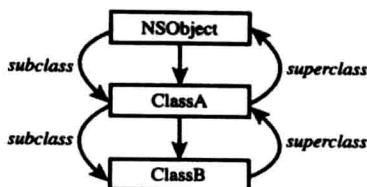


图 8.2 子类和父类

如图 8.2 中，根类没有超类，而 ClassB 位于继承的最底部，没有子类。因此，ClassA 是 NSObject 的子类，而 ClassB 是 ClassA 的子类，也是 NSObject 的子类（从学术术语上讲，它是子类的子类或者孙类）。同样，NSObject 是 ClassA 的超类，也是 ClassB 的超类。因为 ClassB 位于 NSObject 层次结构的下方，所以 NSObject 也是 ClassB 的超类。

下面是 ClassB 的完整声明，ClassB 定义了一个名为 printVar 的方法。

```

@interface ClassB: ClassA
-(void) printVar;
@end

@implementation ClassB
-(void) printVar
{
    NSLog(@"x = %i", x);
}
@end
  
```

虽然在 ClassB 中没有定义任何实例变量，但可以通过 printVar 方法输出实例变量 *x* 的值。这是由于 ClassB 是 ClassA 的子类，因此，它继承 ClassA 的公有实例变量（在这个例子中，实例变量只有一个），图 8.3 描述了这种情况。

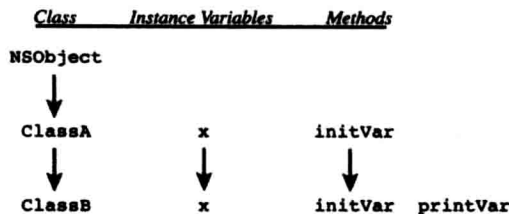


图 8.3 继承实例变量和方法

（当然，图 8.3 并没有显示从 NSObject 类所继承的任何方法或实例变量，虽然 NSObject 有几个方法及实例变量。）

把它集中在一个完整的程序例子中，看看它的工作方式。为了简洁，将所有类的声明和定义放在单个文件中（参见代码清单 8-1）。

代码清单 8-1

// 使用简单的示例说明

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

// ClassA 的声明和定义

@interface ClassA: NSObject
{
    int x;
}

-(void) initVar;
@end

@implementation ClassA
-(void) initVar
{
    x = 100;
}
@end

// ClassB 的声明和定义

@interface ClassB : ClassA
-(void) printVar;
@end

@implementation ClassB
-(void) printVar
{
    NSLog(@"x = %i", x);
}
@end

int main (int argc, char *argv[])
{
    @autoreleasepool {
        ClassB *b = [[ClassB alloc] init];

        [b initVar];    // 将使用继承的方法
        [b printVar];   // 显示 x 的值
    }
}
```

```
    return 0;
}
```

代码清单 8-1 输出

```
x = 100
```

首先，定义 `b` 为一个 `ClassB` 对象。在分配空间并初始化 `b` 后，使用 `initVar` 方法给它发送一条消息。但是回过头看 `ClassB` 的定义，就会发现从未定义过这样的方法。事实上，`initVar` 定义在 `ClassA` 中，由于 `ClassA` 是 `ClassB` 的父类，所以，`ClassB` 可以使用 `ClassA` 的所有方法。对于 `ClassB` 而言，`initVar` 是继承来的方法。

注意

简单提一下，`alloc` 和 `init` 是你使用过但从未在类中定义过的方法，因为你已经利用这些都是从 `NSObject` 类继承的方法这个事实。

向 `b` 发送 `initVar` 消息之后，调用 `printVar` 方法显示实例变量 `x` 的值。输出结果 `x=100` 证实了 `printVar` 能够访问这个实例变量。这是因为和 `initVar` 方法一样，这个变量也是继承的。

记住，继承的概念作用于整个继承链。因此，如果如下语句定义一个父类是 `ClassB` 的新类 `ClassC`：

```
@interface ClassC: ClassB
...
@end
```

那么，`ClassC` 将继承 `ClassB` 的所有方法和实例变量，同时也依次继承 `ClassA` 的所有方法和实例变量，还依次继承 `NSObject` 的所有方法和实例变量。

一定要理解以下事实：类的每个实例都拥有自己的实例变量，即使这些实例变量是继承来的。因此，对象 `ClassC` 与对象 `ClassB` 具有完全不同的实例变量。

8.1.1 找出正确的方法

向对象发送消息时，你可能希望知道如何选择正确的方法应用到这个对象。规则其实很简单。首先，检查该对象所属的类，查看在类中是否明确定义了一个具有指定名称的方法。如果有定义，就会使用这个方法。如果没有定义，就

检查它的父类。如果父类中有定义，就使用这个方法；否则，继续寻找，直到发现下面两种情况中的一种，才会停止查找父类：发现包含指定方法的类，或者一直搜索到根类也没有发现任何方法。如果是第一种情况，就会停止查找；如果是第二种情况，说明存在问题，就会生成类似下面的警告消息：

```
warning: 'ClassB' may not respond to '-inity'
```

在这个例子中，你无意中向类 `ClassB` 发送了一条名为 `inity` 的消息。编译器会告知你：类的对象不能响应这种方法。同样，这是在检查过 `ClassB` 的方法及一直到根类（在这个例子中，根类为 `NSObject`）的父类的方法才确定的。

你将了解更多有关系统是如何检查出正确的方法去执行，相关内容将在第9章“多态、动态类型和动态绑定”中简要介绍。

8.2 通过继承来扩展：添加新方法

继承通常用于扩展一个类。举个例子，假如你刚接受一项任务，要开发一些处理 2D 图形对象的类（如矩形、圆形和三角形）。目前，我们先只考虑矩形。我们回顾第4章“数据类型和表达式”的练习7，从以下例子的 `@interface` 部分开始：

```
@interface Rectangle: NSObject

@property int width, height;
-(int) area;
-(int) perimeter;
@end
```

当前的方法可以设置矩形的宽与高，返回它们的值并计算面积与周长。再添加一个方法，使用一个消息调用设置矩形的宽度与高度值，语句如下：

```
-(void) setWidth: (int) w andHeight: (int) h;
```

假设新类声明在 `Rectangle.h` 文件。实现文件 `Rectangle.m` 如下：

```
#import "Rectangle.h"

@implementation Rectangle

@synthesize width, height;

-(void) setWidth: (int) w andHeight: (int) h
```

```

{
    width = w;
    height = h;
}

-(int) area
{
    return width * height;
}

-(int) perimeter
{
    return (width + height) * 2;
}

@end

```

每个方法的定义都很直观。代码清单 8-2 显示了一个测试它的 `main` 函数。

代码清单 8-2

```

#import "Rectangle.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        Rectangle *myRect = [[Rectangle alloc] init];

        [myRect setWidth: 5 andHeight: 8];

        NSLog(@"Rectangle: w = %i, h = %i", myRect.width, myRect.height);
        NSLog(@"Area = %i, Perimeter = %i", [myRect area],
            [myRect perimeter]);
    }
    return 0;
}

```

代码清单 8-2 输出

```

Rectangle: w = 5, h = 8
Area = 40, Perimeter = 26

```

给 `myRect` 分配内存并将其初始化，然后将宽设为 5、高设为 8。输出的第一行验证了这一点。接着调用合适的消息，计算矩形的面积和周长，由 `NSLog` 显示返回值。

处理完矩形之后，假设现在需要处理正方形。定义一个名为 `Square` 的新类，并在类中定义与 `Rectangle` 类相似的方法。意识到正方形只是矩形的特例，即长

和宽恰好相等的矩形。

因此，简单的处理方法就是定义一个名为 `Square` 的新类，并使它成为 `Rectangle` 的子类。现在，需要添加的方法是将正方形的边设置为特定的值，并获取这个值的方法。代码清单 8-3 显示了 `Square` 类的接口文件和实现文件。

代码清单 8-3 `Square.h` 接口文件

```
#import "Rectangle.h"

@interface Square: Rectangle

-(void) setSide: (int) s;
-(int) side;
@end
```

代码清单 8-3 `Square.m` 实现文件

```
#import "Square.h"

@implementation Square: Rectangle

-(void) setSide: (int) s
{
    [self setWidth: s andHeight: s];
}

-(int) side
{
    return self.width;
}
@end
```

注意，此处将 `Square` 定义为 `Rectangle` 的子类，是在头文件 `Rectangle.h` 中声明的。在这里不必添加任何实例变量，但是添加了名为 `setSide` 和 `side` 的两个新方法。`side` 方法不能直接获取 `Rectangle` 的实例变量 `width`，这个实例变量是私有的，所以 `Square` 类不能访问到。然而，取值方法是继承自父类的，而且能够用来获取 `width` 的值。记得表达式

```
self.width
```

与

```
[self width]
```

是相同的。

发送 `width` 消息到 `side` 消息的接收者。换句话说，执行取值方法 `width`，而不是直接获取实例变量 `width`（正如前面讨论的，确实是这样）。需要理解这个重要概念。

虽然正方形的 4 条边都相等，即只有一个值，但在内部也可用两个数表示。对于 `Square` 类的用户是隐藏的。如果需要，随时可以重新定义 `Square` 类。根据前面介绍的封装概念，任何用户都无须担心内部的细节问题。

`setSide:`方法利用了一个事实，即你已从 `Rectangle` 类继承了一个方法来设定矩形的宽度和高度值。因此，`setSide:`方法会调用 `Rectangle` 类的 `setWidth:andHeight:`方法，传递参数作为宽度与高度值，不必进行其他操作。使用 `Square` 对象，可以通过 `setSide:`方法设置正方形的大小，并利用 `Rectangle` 类的方法计算正方形的面积、周长等。代码清单 8-3 表示新的 `Square` 类的测试程序和输出。

代码清单 8-3 测试程序

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        Square *mySquare = [[Square alloc] init];

        [mySquare setSide: 5];

        NSLog(@"Square s = %i", [mySquare side]);
        NSLog(@"Area = %i, Perimeter = %i",
            [mySquare area], [mySquare perimeter]);
    }
    return 0;
}
```

代码清单 8-3 输出

```
Square s = 5
Area = 25, Perimeter = 20
```

这种定义 `Square` 类的方式使用了 Objective-C 中类的基本技术，即扩展自己或其他人以前实现的类，使它适合自己的需要。另外，所谓的分类（category）机制允许你以模块的方式向现有类定义添加新的方法，也就是说，不必经常给

同一接口和实现文件增加新的定义。当你希望对没有源代码访问权限的类添加新的定义，这样特别方便。在第 11 章“分类和协议”中将会学习到分类。

8.2.1 Point 类和对象创建

Rectangle 类只存储矩形的大小。在实际的图形应用中，需要保存各种附加消息，例如：矩形的填充色、线条颜色、窗口中的位置（原点），等等，可以很方便地通过扩展类来处理这些情况。现在，让我们看看矩形原点的概念。假设“原点”是指笛卡儿坐标系 (x, y) 中矩形左下角的位置。如果正在编写绘图应用程序，这一点可能表示矩形在窗口中的位置，如图 8.4 所示。在图 8.4 中，矩形的原点是 (x_1, y_1) 。

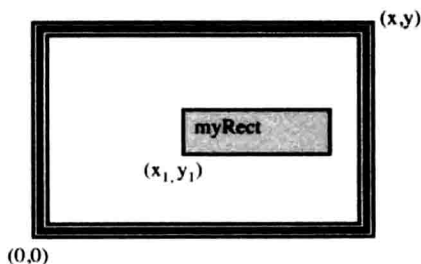


图 8.4 窗口中绘制的矩形

扩展 **Rectangle** 类，将矩形原点 (x, y) 保存为两个不同的值。或许你已经意识到，在开发图形应用程序的过程中需要处理很多坐标，因此，定义一个名为 **XYPoint** 的类（在第 3 章的练习 7 中出现过这个问题）：

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

@interface XYPoint: NSObject
@property int x, y;

-(void) setX: (int) xVal andY: (int) yVal;
@end
```

现在回到 **Rectangle** 类，我们希望在这个类存储矩形的原点。所以，向 **Rectangle** 类的定义添加另一个实例变量 **origin**：

```
@implementation Rectangle
{
    XYPoint *origin;
```

```
}
...

```

为矩形的原点添加设值方法和取值方法也是合理的。为突出重点，我们这里不合成原点的存取方法，而是自己编写。

8.2.2 @class 指令

现在可以设置矩形（或正方形）的宽、高和原点。首先，完整查看一下接口文件 `Rectangle.h`：

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

@class XYPoint;

@interface Rectangle: NSObject

@property int width, height;

-(XYPoint *) origin;
-(void) setOrigin: (XYPoint *) pt;
-(void) setWidth: (int) w andHeight: (int) h;
-(int) area;
-(int) perimeter;
@end

```

在 `Rectangle.h` 头文件中使用一个新的指令：

```
@class XYPoint;

```

这是因为编译器在遇到 `Rectangle` 定义的实例变量 `XYPoint` 时，必须了解 `XYPoint` 是什么。类名还会分别用在 `setOrigin:` 和 `origin` 方法的参数及返回类型声明。还有另一个选择，可以导入头文件替代这条指令，语句如下：

```
#import "XYPoint.h"

```

使用 `@class` 指令提高了效率，因为编译器不需要引入和处理整个 `XYPoint.h` 文件（虽然它很小），只需知道 `XYPoint` 是一个类名。如果需要引用 `XYPoint` 类的方法（在实现部分中），`@class` 指令是不够的，因为编译器需要更多的消息。它需要清楚方法有多少参数、它们是什么类型、方法的返回类型是什么。

确定已经理解了 `@class` 指令的使用。它的作用是当编辑器遇见这样的语句

```
XYPoint *origin;

```

时，可以告知编译器 `XYPoint` 是一个类的名字，`origin` 是 `XYPoint` 类的一个对象。这是编译器需要了解的。

我们需要填补 `XYPoint` 类和 `Rectangle` 方法的空白，这样就可以在一个程序中测试所有的内容。代码清单 8-4 显示了 `XYPoint` 类的实现文件。

首先，代码清单 8-4 显示了 `Rectangle` 类的新方法。

代码清单 8-4 `Rectangle.m` 添加的方法

```
#import "XYPoint.h"

-(void) setOrigin: (XYPoint *) pt
{
    origin = pt;
}

-(XYPoint *) origin
{
    return origin;
}
@end
```

以下是完整的 `XYPoint` 和 `Rectangle` 类定义，接着是测试程序。

代码清单 8-4 `XYPoint.h` 接口文件

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

@interface XYPoint: NSObject

@property int x, y;

-(void) setX: (int) xVal andY: (int) yVal;
@end
```

代码清单 8-4 `XYPoint.m` 实现文件

```
#import "XYPoint.h"

@implementation XYPoint

@synthesize x, y;

-(void) setX: (int) xVal andY: (int) yVal
{
    x = xVal;
    y = yVal;
}
```

```

}
@end

```

代码清单 8-4 Rectangle.h 接口文件

```

#import <Foundation/Foundation.h>

@class XYPoint;
@interface Rectangle: NSObject

@property int width, height;

-(XYPoint *) origin;
-(void) setOrigin: (XYPoint *) pt;
-(void) setWidth: (int) w andHeight: (int) h;
-(int) area;
-(int) perimeter;
@end

```

代码清单 8-4 Rectangle.m 实现文件

```

#import "Rectangle.h"

@implementation Rectangle
{
    XYPoint *origin;
}

@synthesize width, height;

-(void) setWidth: (int) w andHeight: (int) h
{
    width = w;
    height = h;
}

-(void) setOrigin: (XYPoint *) pt
{
    origin = pt;
}

-(int) area
{
    return width * height;
}

-(int) perimeter
{

```

```

    return (width + height) * 2;
}

-(CGPoint *) origin
{
    return origin;
}

@end

```

代码清单 8-4 测试程序

```

#import "Rectangle.h"
#import "CGPoint.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        Rectangle *myRect = [[Rectangle alloc] init];
        CGPoint *myPoint = [[CGPoint alloc] init];

        [myPoint setX: 100 andY: 200];

        [myRect setWidth: 5 andHeight: 8];
        myRect.origin = myPoint;

        NSLog(@"Rectangle w = %i, h = %i", myRect.width, myRect.height);

        NSLog(@"Origin at (%i, %i)", myRect.origin.x, myRect.origin.y);

        NSLog(@"Area = %i, Perimeter = %i",
              [myRect area], [myRect perimeter]);
    }
    return 0;
}

```

代码清单 8-4 输出

```

Rectangle w = 5, h = 8
Origin at (100, 200)
Area = 40, Perimeter = 26

```

在 `main` 方法中，为矩形 `myRect` 和点 `myPoint` 分配空间并进行初始化。使用 `setX: andY:` 将 `myPoint` 值设置为 (100, 200)。将矩形的宽和高设置为 5 和 8 之后，调用 `setOrigin` 方法，将矩形原点设置为指定的点 `myPoint`。调用 3 次 `NSLog` 方法获取并输出值。表达式

```
myRect.origin.x
```

由存取方法 `origin` 返回 `XYPoint` 对象，方法 `x` 获取矩形原点的 `x` 坐标。以同样的方式，表达式

```
myRect.origin.y
```

检索矩形原点的 `y` 坐标。这个表达式与

```
[[myRect origin] y]
```

是相同的。理解编译器是如何解释点运算符就会清楚这一点。
译器是如何解释点运算符就会清楚这一点。

8.2.3 具有对象的类

能够解释代码清单 8-5 的输出结果吗？

代码清单 8-5

```
#import "Rectangle.h"
#import "XYPoint.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        Rectangle *myRect = [[Rectangle alloc] init];
        XYPoint *myPoint = [[XYPoint alloc] init];

        [myPoint setX: 100 andY: 200];

        [myRect setWidth: 5 andHeight: 8];
        myRect.origin = myPoint;

        NSLog(@"Origin at (%i, %i)", myRect.origin.x, myRect.origin.y);

        [myPoint setX: 50 andY: 50];
        NSLog(@"Origin at (%i, %i)", myRect.origin.x, myRect.origin.y);
    }
    return 0;
}
```

代码清单 8-5 输出

```
Origin at (100, 200)
Origin at (50, 50)
```

将 `myPoint` 从源程序中的 (100, 200) 改为 (50, 50)，显然也改变了矩形的原点！但是为什么出现这种情况？你并没有显式地重新设置矩形的原点，为什么矩形的原点会改变？回顾 `setOrigin:` 方法的定义，就会明白：

```
-(void) setOrigin: (XYPoint *) pt
{
    origin = pt;
}
```

当使用表达式

```
myRect.origin = myPoint;
```

调用 `setOrigin:` 方法时，`myPoint` 的值作为参数传递给该方法。这个值指向存储 `XYPoint` 对象的内存，如图 8.5 所示。



图 8.5 内存中的 `myPoint` 对象

存储在 `myPoint`（它是一个指向内存的指针）中的值被复制到在该方法中定义的本地变量 `pt` 中。现在 `pt` 与 `myPoint` 引用内存中相同的数据，如图 8.6 所示。

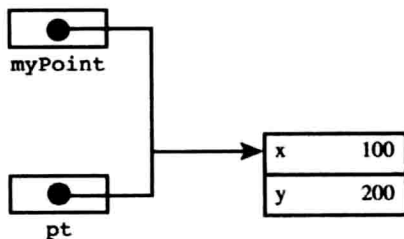


图 8.6 将矩形原点传递给该方法

在方法中将 `origin` 变量设置为 `pt` 时，`pt` 存储的指针复制到实例变量 `origin`，如图 8.7 所示。

因为 `myPoint` 和存储在 `myRect` 中的 `origin` 变量引用内存中的同一区域（与局部变量 `pt` 相同）。所以，将 `myPoint` 的值改为 (50, 50) 时，矩形的原点也被更改。

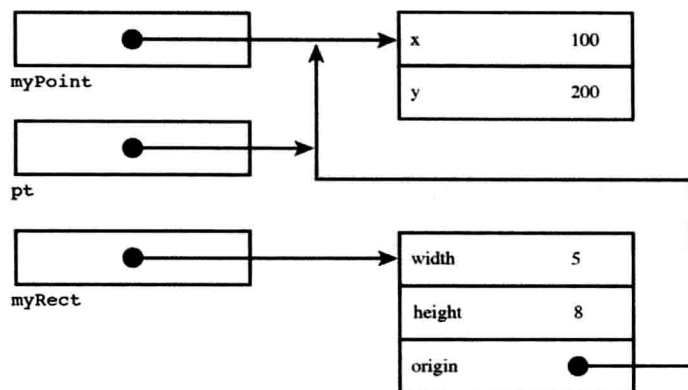


图 8.7 设置矩形的原点

避免这一问题的办法就是修改 `setOrigin:` 方法，这样就能够创建自己的点，并将 `origin` 设置为这个点，语句如下：

```

-(void) setOrigin: (XYPoint *) pt
{
    if (! origin)
        origin = [[XYPoint alloc] init];

    origin.x = pt.x;
    origin.y = pt.y;
}

```

这个方法首先判断实例变量 `origin` 是否为非零（前提是已经理解了判断和逻辑否定运算符 `!` 的使用）。曾提到，所有的实例变量在初始化时均为空。当创建一个新的 `Rectangle` 对象时，它的实例变量（包括 `origin`）将为空。

如果 `origin` 为零，`setOrigin:` 方法将创建和初始化一个新的 `XYPoint` 对象并存储在引用 `origin` 中。

然后为 `XYPoint` 对象设置参数中的 `x`、`y` 坐标。请研究该方法，直到了解它的工作原理。

`setOrigin:` 方法的改变意味着每个 `Rectangle` 实例现在都有它的 `XYPoint` 实例。既然它负责为 `XYPoint` 分配内存，所以还应该负责释放该内存。通常，当一个类包含其他对象时，有时你希望拥有部分或全部对象。以矩形为例，它有自己的原点很合理，因为这是矩形的基本属性。只有类的访问者方法能够设置或者获取 `origin`。这与数据封装的概念是一致的。

使用修改后的方法，重新编译和运行代码清单 8-5，生成如图 8.8 所示的出错消息。

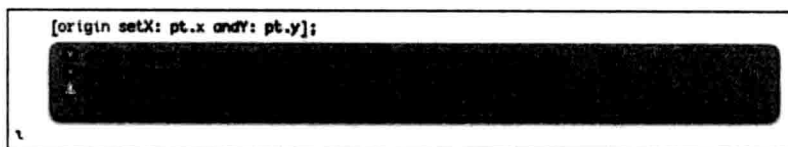


图 8.8 编译器出错消息

糟了，这么多问题！这里的问题是由于在修改后的方法中引用了 `XYPoint` 类的一些实例变量，所以现在编译器需要的信息多于 `@class` 指令提供的。在这个例子中，返回去看 `Rectangle.h` 的头文件并用 `import` 代替这个指令，如：

```
#import "XYPoint.h"
```

代码清单 8-5B 输出

```
Origin at (100, 200)
Origin at (100, 200)
```

这样就好多了。这次在 `main` 方法中将 `myPoint` 的值更改为 (50, 50) 时，对矩形原点没有影响，因为在矩形的 `setOrigin:` 方法中创建了该点的副本。

顺便说一下，这里我们没有合成 `origin` 方法，因为如果采用合成设值 `setOrigin:` 方法，其行为将与你之前编写的那个方法一样。也就是说，在默认情况下，合成的设值方法只是简单地复制对象指针，而不是对象本身。

你可以合成另一种设值方法，而不是制作对象的副本。但是，如果那样做，就需要学习如何编写特殊的复制方法。第 18 章“复制对象”将再次讨论这一内容。

在结束本节之前，我们思考一个问题，先不讨论 `Rectangle` 类，假设在代码清单 8-4 的测试程序中插入以下一段代码，在设置完 `Rectangle` 的值后：

```
XYPoint *theOrigin = myRect.origin;

theOrigin.x = 200;
theOrigin.y = 300;
```

你会想这段代码执行后，`myRect` 的 `origin` 发生了什么变化？对了，`origin` 将变为 (200, 300)。取值方法直接将 `origin` 对象返回，这样显得很“脆弱”。任何人

改变了 x 或 y 的值，都会直接影响到 `rectangle` 的 x 或 y 的值。出于这种原因，更安全的方法是编写一个这样的取值方法，创建一个对象的副本，并返回副本的对象。使用这种方式，可以使实例变量避免不经意的修改。在这里就不做此修改，留做一个练习。注意复制对象并返回时面临的性能开销。判断一下在你设计类时，这样做是否有价值。

8.3 覆写方法

前面曾提到过，不能通过继承删除或减少方法。但可以利用覆写来更改继承方法的定义。

回头看这两个类：`ClassA` 与 `ClassB`。假定要为 `ClassB` 编写自己的 `initVar` 方法。你已经知道 `ClassB` 将继承定义在 `ClassA` 中的 `initVar` 方法，但是否可以新建一个同名的方法来替代继承的方法呢？答案是可以，只要定义一个同名的新方法即可。使用和父类相同的名称定义的方法代替或覆写了继承的定义。新方法必须具有相同的返回类型，并且参数的数目与覆写的方法相同。

代码清单 8-6 展示了简单的例子来说明这个概念。

代码清单 8-6

```
// 覆写方法

#import <Foundation/Foundation.h>

// ClassA 的声明和定义

@interface ClassA: NSObject
{
    int x; // 将由子类继承
}

-(void) initVar;
@end

////////////////////////////////////
@implementation ClassA
-(void) initVar
{
    x = 100;
}
@end
```

```

// ClassB 的声明和定义

@interface ClassB: ClassA
-(void) initVar;
-(void) printVar;
@end
//////////
@implementation ClassB
-(void) initVar    // 添加方法
{
    x = 200;
}

-(void) printVar
{
    NSLog(@"x = %i", x);
}
@end
//////////
int main (int argc, char *argv[])
{
    @autoreleasepool {
        ClassB *b = [[ClassB alloc] init];

        [b initVar]; // 使用 B 中覆盖的方法

        [b printVar]; // 显示 x 的值
    }
    return 0;
}

```

代码清单 8-6 输出

```
x = 200
```

显然，消息语句

```
[b initVar];
```

导致使用定义在 ClassB 中的 initVar 方法，而不是使用 ClassA 中所定义的方法，前一示例也是如此，如图 8.9 所示。

8.3.1 选择哪种方法

前面曾讲到系统如何上溯类层次查找应用于对象的方法。如果在不同的类中有名称相同的方法，则根据作为消息的接收者的类选择正确的方法。代码清

单 8-7 使用与前面的 ClassA 和 ClassB 相同的类定义。

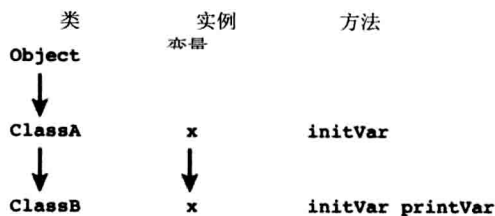


图 8.9 覆写 initVar 方法

代码清单 8-7

```

#import <Foundation/Foundation.h>

// 在这里插入 ClassA 和 ClassB 的定义

int main (int argc, char *argv[])
{
    @autoreleasepool {
        ClassA *a = [[ClassA alloc] init];
        ClassB *b = [[ClassB alloc] init];

        [a initVar]; // 使用 ClassA 方法
        [a printVar]; // 显示 x 的值

        [b initVar]; // 使用 ClassB 中覆盖的方法
        [b printVar]; // 显示 x 的值
    }
    return 0;
}

```

编译该程序时会得到以下警告消息：

```
warning: 'ClassA' may not respond to '-printVar'
```

这里有什么问题？前一节讨论过此问题。观察 ClassA 的声明

```

// ClassA 的声明和定义

@interface ClassA: NSObject
{
    int x;
}

-(void) initVar;
@end

```

注意到没有声明 `printVar` 方法。该方法声明并定义在 `ClassB` 中。因此，尽管 `ClassB` 对象及其派生类可以通过继承使用此方法，但是 `ClassA` 对象却不能使用此方法，因为此方法的定义在类层次中较低的位置。

注意

你可以通过某种方式强制使用此方法，但是我们在这里不对此进行讨论。此外，这也不是一个好的编程实践。

回到例子中，为 `ClassA` 添加一个 `printVar` 方法，以便显示实例变量的值

// ClassA 的声明和定义

```
@interface ClassA: NSObject
{
    int x; // 将由子类继承
}

-(void) initVar;
-(void) printVar;
@end

@implementation ClassA
-(void) initVar
{
    x = 100;
}

-(void) printVar
{
    NSLog(@"x = %i", x);
}

@end
```

`ClassB` 的声明与定义保持不变。现在，再次尝试编译并运行该程序。

代码清单 8-7 输出

```
x = 100
x = 200
```

现在讨论这个实际的例子。首先，分别将 `a` 和 `b` 定义为 `ClassA` 和 `ClassB` 对象。经过内存分配与初始化后，向 `a` 发送一条消息，让它应用 `initVar` 方法。此方法在 `ClassA` 中定义，所以，它是所选的方法。此方法只是简单地将实例变

量值 x 设为 100 并返回。然后调用刚刚添加到 ClassA 中的 `printVar` 方法来显示 x 的值。

与 ClassA 对象类似, ClassB 对象 b 经过内存分配及初始化后, 将实例变量 x 设为 200, 最后显示其值。

一定要理解如何按照 a 与 b 所属的类选择相应的方法。这是 Objective-C 中面向对象编程的基础概念。

作为练习, 考虑从 ClassB 中删除 `printVar` 方法可行吗? 为什么可行, 或为什么不可行?

当定义一个子类时, 不仅可以添加新方法来有效地扩展类的定义, 还可以添加新的实例变量。以上两种情况的影响是累加的, 不能通过继承减少方法或实例变量, 只能添加。对于方法来说, 可以添加或者覆写。

分析一下, 为什么需要创建子类? 有如下 3 点理由:

- (1) 希望继承一个类的函数, 也许加入一些新的方法和/或实例变量。
- (2) 希望创建一个类的特别的版本 (如图形对象的特定类型)。
- (3) 希望通过覆写一个或多个方法来改变类的默认行为。

8.4 抽象类

除了介绍一些术语之外, 还有什么更好的办法来结束本章内容吗? 这里介绍这个概念, 是因为它与继承的概念直接相关。

有时, 创建类只是为了更容易创建子类。因此, 这些类名为抽象 (abstract) 类, 或等价地称为抽象超类 (abstract superclasses)。在该类中定义方法和实例变量, 但不期望任何人从这个类创建实例。例如, 考虑根对象 `NSObject`, 能想出这个类定义对象的任何用途吗?

Foundation 框架将在第二部分讲述, 它包含几个所谓的抽象类。举一个例子, Foundation 的 `NSNumber` 类是为了将数字作为对象处理而创建的抽象类。整数与浮点数字通常有不同的内存需求。每种数字类型都有单独的 `NSNumber` 子类, 因为这些子类与它们的抽象超类不同, 这些子类是具体存在的, 它们名为具体子类。每个具体子类属于 `NSNumber` 类, 总称为簇 (cluster)。向 `NSNumber` 类发送消息来创建新的整数对象时, 使用合适的子类为整数对象分配必需的空

间，并正确地设定它的值。这些子类实际上是私有的。你自己无法直接访问这些子类，只能通过抽象的超类间接访问。抽象超类提供了处理所有的数字对象类型的公共接口，你无须了解存储在数字对象中的数字类型及如何设置与检索它的值。

的确，这个讨论看起来可能有些“抽象”（报歉!），不用担心，只需掌握基本的概念就可以。

8.5 练习

1. 向代码清单 8-1 添加一个名为 ClassC 的新类，它是 ClassB 的子类。创建一个 initVar 方法，它将实例变量 x 的值设置为 300。编写一个测试方法，它声明对象 ClassA、ClassB 及 ClassC，并且调用相应的 initVar 方法。
2. 使用高分辨率的设备时，可能需要使用允许将点指定为浮点值，而不是简单的整数（iOS 中使用 CGRect 结构用在矩形上，矩形所有的坐标和尺寸均用浮点数表示）坐标系。修改本章的 XYPoint 与 Rectangle 类，以处理浮点数字。矩形的宽度、高度、面积与周长也都使用浮点数字进行处理。
3. 修改代码清单 8-1，向其添一个名为 ClassB2 的新类，ClassB2 和 ClassB 一样，都是 ClassA 的子类。ClassB 与 ClassB2 之间有什么关系？指出 NSObject 类、ClassA、ClassB 及 ClassB2 之间的层次关系。ClassB 的超类是什么？ClassB2 的超类是什么？一个类可以有多少个子类、多少个超类？
4. 编写一个名为 translate: 的 Rectangle 方法，使用 XYPoint 对象作为其参数。通过指定的向量对矩形的原点进行变换。注意，变换是指一个点移动到另一个点。
5. 定义一个名为 GraphicObject 的新类，使其成为 NSObject 的子类。在新类中定义如下一些实例变量：

```
int    fillColor;    // 32 位颜色
BOOL   filled;       // 是否为对象填充了？
int    lineColor;    // 32 位线的颜色
```

编写一个方法，设定并检索前面定义的变量。使 `Rectangle` 类成为 `GraphicObject` 的子类。

定义两个新类 `Circle` 和 `Triangle`，它们都是 `GraphicObject` 的子类。编写一些方法来设定及检索这些对象的各种参数，并计算圆的圆周、面积及三角形的周长、面积。

6. 编写一个名为 `containsPoint:` 的 `Rectangle` 方法，使用 `XYPoint` 对象作为参数

```
-(BOOL) containsPoint: (XYPoint *) aPoint;
```

这个方法返回 `BOOL` 值，如果矩形包含有这个点，返回 `YES`，否则返回 `NO`。

7. 编写一个名为 `intersect:` 的 `Rectangle` 方法，该方法使用一个矩形作为参数，并返回代表两个矩形的重叠区域。例如，给定图 8.10 所展示的两个矩形，这个方法应该返回原点位于 (400, 420) 的矩形，其宽度为 50、高度为 60。

如果矩形没有相交，返回宽度与高度均为零的矩形，其原点为 (0, 0)。

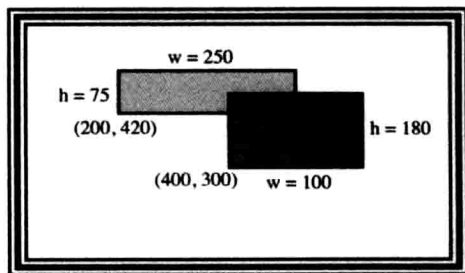


图 8.10 相交的矩形

8. 为 `Rectangle` 类编写一个名为 `draw` 的方法，此方法使用虚线与垂直的条形字符绘制矩形。以下代码序列

```
Rectangle *myRect = [[Rectangle alloc] init];
[myRect setWidth: 10 andHeight: 3];
[myRect draw];
```

将产生如下输出结果：

```
-----
|       |
```

```
|      |  
|      |  
-----
```

注意

应该使用 `printf` 绘制字符，因为每次调用 `NSLog` 时都会显示一个新行。

多态、动态类型和动态绑定

本章将介绍 Objective-C 语言的一些特性, 这些特性使它成为一门功能强大的编程语言, 并且使其区别于其他面向对象的程序设计语言, 如 C++。本章讲述了 3 个关键概念: 多态、动态类型和动态绑定。多态能够使来自不同类的对象定义相同名称的方法。动态类型能使程序直到执行时才确定对象所属的类。动态绑定则能使程序直到执行时才确定实际要调用的对象方法。

9.1 多态: 相同的名称, 不同的类

代码清单 9-1 为 Complex 类的接口文件, 它用于表示程序中的复数。

代码清单 9-1 接口文件 Complex.h

```
// Complex 类的接口文件

#import <Foundation/Foundation.h>

@interface Complex: NSObject

@property double real, imaginary;
-(void) print;
-(void) setReal: (double) a andImaginary: (double) b;
-(Complex *) add: (Complex *) f;
@end
```

你应该已经在第 7 章的练习 5 和练习 6 中完成了该类的实现部分。我们在这两个练习中加入了一个补充的 setReal:andImaginary:方法, 使用它就可以一条消息设置数字的实数和虚数部分, 语句如下。

代码清单 9-1 实现文件 Complex.m

// Complex 类的实现文件

```

#import "Complex.h"

@implementation Complex

@synthesize real, imaginary;

-(void) print
{
    NSLog ("%g + %gi ", real, imaginary);
}

-(void) setReal: (double) a andImaginary: (double) b
{
    real = a;
    imaginary = b;
}

-(Complex *) add: (Complex *) f
{
    Complex *result = [[Complex alloc] init];

    result.real = real + f.real;
    result.imaginary = imaginary + f.imaginary;

    return result;
}

@end

```

代码清单 9-1 测试程序 main.m

// 共享的方法名: 多态性

```

#import "Fraction.h"
#import "Complex.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        Fraction *f1 = [[Fraction alloc] init];
        Fraction *f2 = [[Fraction alloc] init];
        Fraction *fracResult;
        Complex *c1 = [[Complex alloc] init];
        Complex *c2 = [[Complex alloc] init];
        Complex *compResult;
    }
}

```

```

[f1 setTo: 1 over: 10];
[f2 setTo: 2 over: 15];

[c1 setReal: 18.0 andImaginary: 2.5];
[c2 setReal: -5.0 andImaginary: 3.2];

// 将两个 Complex 数相加并显示

[c1 print]; NSLog(@"      +"); [c2 print];
NSLog(@"-----");
compResult = [c1 add: c2];
[compResult print];
NSLog(@"\n");

// 将两个分数相加并显示
[f1 print]; NSLog(@"      +"); [f2 print];
NSLog(@"-----");
fracResult = [f1 add: f2];
[fracResult print];
}
return 0;
}

```

代码清单 9-1 输出

```

18 + 2.5i
    +
-5 + 3.2i
-----
13 + 5.7i

1/10
  +
2/15
----
7/30

```

请注意，`Fraction` 和 `Complex` 类都包含 `add:` 和 `print` 方法。那么，在执行以下消息表达式

```

compResult = [c1 add: c2];
[compResult print];

```

时，系统如何知道执行哪个方法呢？其实很简单：Objective-C 运行时知道第一条消息的接收者 `c1` 是一个 `Complex` 对象。因此，选择定义在 `Complex` 类中的

add:方法。

Objective-C 的运行时系统还确定 compResult 是一个 Complex 对象。因而，它选择定义在 complex 类中的 print 方法来显示加法的结果。同样的论述也适用于以下消息表达式：

```
fracResult = [f1 add: f2];
[fracResult print];
```

注意

第 13 章“基本的 C 语言特征”会有更详尽的描述，系统总是携带有关“一个对象属于哪个类”这样的信息的，使系统在运行时才做出关键性的决定，而不是在编译时。

基于 f1 和 fracResult 类，Fraction 类中相应的方法将被选中，用于执行消息表达式。

前面提到过，使不同的类共享相同方法名称的能力称为多态。它让你可以开发一组类，这组类中的每一个类都能响应相同的方法名。每个类的定义都封装了响应特定方法所需的代码，这就使得它独立于其他的类定义。多态还允许你以后添加新的类，这些新类能够响应相同的方法名。

9.2 动态绑定和 id 类型

第 4 章“数据类型和表达式”简单介绍了 id 数据类型，并指出这是一种通用的对象类型。也就是说，id 可以用来存储属于任何类的对象。当以这种方式在一个变量中存储不同类型的对象时，程序的执行期间这种数据类型的真正优势就体现了。请研究代码清单 9-2 和它对应的输出。

代码清单 9-2 测试程序 main.m

```
// 说明动态类型绑定
```

```
#import "Fraction.h"
#import "Complex.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        id    dataValue;
```

```

Fraction *f1 = [[Fraction alloc] init];
Complex *c1 = [[Complex alloc] init];

[f1 setTo: 2 over: 5];
[c1 setReal: 10.0 andImaginary: 2.5];

// 第一个 dataValue 获得了一个分数

dataValue = f1;
[dataValue print];

// 第一个 dataValue 获得了一个 Complex 数

dataValue = c1;
[dataValue print];
}
return 0;
}

```

代码清单 9-2 输出

```

2/5
10 + 2.5i

```

变量 `dataValue` 被声明为 `id` 对象类型。因此，`dataValue` 可用来保存程序中任何类型的对象。务必注意，声明中并没有使用星号：

```
id dataValue;
```

`Fraction f1` 被设为 $2/5$ ，并且 `Complex` 数 `c1` 被设为 $(10+2.5i)$ 。赋值语句 `dataValue = f1;`

将 `Fraction f1` 存储到 `dataValue` 中。现在，使用 `dataValue` 可以做什么呢？其实，可以对 `dataValue` 调用可用于 `Fraction` 对象的任何方法，即使 `dataValue` 是一个 `id` 类型，而不是 `Fraction`。然而，如果 `dataValue` 可以存储任何类型的对象，那么系统如何知道应该调用哪一个方法呢？就是说，当系统遇到消息表达式

```
[dataValue print];
```

时，如何知道应该调用哪个 `print` 方法呢？你知道，在 `Fraction` 和 `Complex` 类中都定义有 `print` 方法。

前面提到，答案就在于以下事实：Objective-C 系统总是跟踪对象所属的类。答案同样存在于动态类型和动态绑定的概念之中，就是说，先判定对象所属的

类，然后在运行时确定需要动态调用的方法，而不是在编译的时候。

因此，在程序执行期间，当系统准备将 `print` 消息发送给 `dataValue` 时，它首先检查 `dataValue` 中存储的对象所属的类。在代码清单 9-2 的第一种情况中，这个变量保存一个 `Fraction` 对象，因此，使用在 `Fraction` 类中定义的 `print` 方法。这一点使用程序的输出即可验证。

在第二种情况下，发生了同样的事情。首先，`Complex` 数 `cl` 被指派给 `dataValue`。然后，执行以下消息表达式

```
[dataValue print];
```

这次因为 `dataValue` 包含属于 `complex` 的对象，所以选择该类相应的 `print` 方法来执行。

这是一个简单的例子，但是笔者认为你可以将这个概念推广到更复杂的应用。结合多态、动态绑定和动态类型时，你就能轻易地编写出可以向不同类的对象发送相同消息的代码。

例如，考虑可用来在屏幕上绘制图形对象的 `draw` 方法。你可能为每个图形对象（比如文本、圆、矩形、窗口，等等）定义了不同的 `draw` 方法。如果要绘制的特定对象存储在一个名为 `currentObject` 的 `id` 变量中，仅仅通过发送 `draw` 消息就可以在屏幕上进行绘制，语句如下：

```
[currentObject draw];
```

甚至可以首先测试它，确保存储在 `currentObject` 中的对象的确响应 `draw` 方法。在本章后面将学到如何这样做。

9.3 编译时和运行时检查

因为存储在 `id` 变量中的对象类型在编译时无法确定，所以一些测试推迟到运行时进行。也就是说，推迟到程序执行时。

考虑下列代码序列：

```
Fraction *f1 = [[Fraction alloc] init];  
[f1 setReal: 10.0 andImaginary: 2.5];
```

回顾一下，`setReal:andImaginary:` 方法应用于复数，而不是分数，当编译包含这些语句的程序时，将显示以下消息：

```
'Fraction' may not respond to 'setReal:andImaginary:'
```

Objective-C 编译器知道 `f1` 是一个 `Fraction` 对象，因为它就是这样声明的。编译器同样知道，当遇到消息表达式

```
[f1 setReal: 10.0 andImaginary: 2.5];
```

时，`Fraction` 类并不包含 `setReal:andImaginary:` 方法（并且也没有继承该方法）。所以，生成前面所示的警告消息。

现在考虑下面的代码序列：

```
id dataValue = [[Fraction alloc] init];
...
[dataValue setReal: 10.0 andImaginary: 2.5];
```

这些代码行在编译时，编译器不会产生警告消息。这是因为编译器在处理源文件时并不知道存储在 `dataValue` 中的对象类型（即编译器并不知道 `Fraction` 对象早已存在于这个变量中）。

直到运行包含有这些代码的程序时，程序崩溃，出现如下出错消息：

```
-[Fraction setReal:andImaginary:]: unrecognized selector sent to instance 0x103f00
```

当程序运行时，系统首先检查存储在 `dataValue` 中的对象类型。因为在 `dataValue` 中存储有 `Fraction`。所以运行时系统检查并查找定义在 `Fraction` 类中的 `setReal:andImaginary:` 方法。因为未找到这个方法，所以显示前面的出错消息并且终止程序的运行。

9.4 id 数据类型与静态类型

如果 `id` 数据类型可以用来存储任何类型的对象，为什么不把所有的对象都声明为 `id` 类型呢？不要养成滥用这种通用数据类型的习惯。

首先，将一个变量定义为特定类的对象时，使用的是静态类型。“静态”指的是对存储在变量中对象的类型进行显式声明。这样，存储在这种形态中的对象的类是预定义的，也就是静态的。使用静态类型时，编译器尽可能确保变量的用法在程序中始终保持一致。编译器能够通过检查来确定应用于对象的方法是由该类定义的还是由该类继承的，否则它将显示警告消息。这样，在程序中声明名为 `myRect` 的 `Rectangle` 变量时，编译器就会检查对 `myRect` 调用的每个

方法是定义在 `Rectangle` 类中还是从父类继承的。

注意

在本章后面你会看到，调用对象上的方法也可以通过指定方法的名称作为变量。这种情况下，编译器无法检查你的方法参数或者返回类型。

然而，如果检查是在运行时执行的，为什么还要关心静态类型呢？关心静态类型是因为它能更好地在程序编译阶段而不是在运行时指出错误。如果把它留到运行时，那么在错误发生时，你甚至都可能不在现场。假如程序投入到生产环境，一些倒霉的用户在运行程序时会意外地发现，特定的对象不能够识别某个方法，从而引发程序崩溃。

使用静态类型的另一个原因是它能够提高程序的可读性。考虑以下声明

```
id f1;
```

对比

```
Fraction *f1;
```

你认为哪个声明更容易理解。也就是说，对读者来说，哪个更能清楚地说明使用 `f1` 变量的目的？结合使用静态类型和有意义的变量名（在以前的例子中，我们并没有刻意选择合适的变量名），将会对程序的自说明性产生深远的影响。

注意

你不能为 `id` 变量使用点运算符；如果你尝试这样做，编译器会报错。

9.4.1 动态类型的参数和返回类型

如果使用动态类型来调用一个方法，需要注意以下规则：如果在多个类中实现名称相同的方法，那么每个方法都必须符合各个参数的类型和返回值类型。这样编译器才能为消息表达式生成正确的代码。

编译器会对它所遇到的每个类声明执行一致性检查。如果有一个或多个方法在参数或者返回类型上存在冲突，编译器就会显示警告消息。例如，`Fraction` 和 `Complex` 类中都包含 `add:` 方法。然而，`Fraction` 类的参数和返回类型都是 `Fraction` 对象，而 `Complex` 类的参数和返回类型是 `Complex` 对象。如果 `frac1` 和 `myFract` 都是 `Fraction` 对象，而 `compl` 和 `myComplex` 都是 `Complex` 对象，那么，以下声明

```
result = [myFract add: frac1];
```

和

```
result = [myComplex add: comp1];
```

不会导致编译器显示任何警告消息，这是因为，在这两种情况下，消息的接收者都是静态类型，并且编译器可以检查这些方法的使用是否和在接收者类中定义的一致。

如果 `dataValue1` 和 `dataValue2` 是 id 变量，那么语句

```
result = [dataValue1 add: dataValue2];
```

会导致编译器生成代码，将参数传递给 `add:` 方法，并通过假设来处理其返回值。

在运行时，Objective-C 运行时系统仍然会检查存储在 `dataValue1` 中对象所属的类选择相应的方法来执行。然而，在大多数情况下，编译器可能生成不正确的代码来向方法传递参数或处理返回值。例如当一个方法选取对象作为它的参数，而另一个方法选取浮点数作为参数时，很有可能发生这种情况。或者一个方法以对象作为返回值，而另一个以整型数作为返回值。如果这两个方法之间的不一致仅在于对象类型的不同（例如，`Fraction` 的 `add:` 方法使用 `Fraction` 对象作为其参数和返回值，而 `Complex` 的 `add:` 方法使用 `complex` 对象作为参数），编译器仍然能够生成正确的代码，因为传递给对象的引用是内存地址（即指针）。

9.5 有关类的问题

开始使用可以包含来自不同类的对象的变量时，可能会遇到以下问题：

- 这个对象是矩形吗？
- 这个对象支持 `print` 方法吗？
- 这个对象是 `Graphics` 类或是其子类的成员吗？

这些问题的答案可以用来执行不同的代码序列，避免错误或在运行程序时检查程序的完整性。

表 9.1 总结了 `NSObject` 类所支持的一些基本方法，它们用来提出这类问题。在这个表中，`class-object` 是一个类对象（通常是由 `class` 方法产生的），`selector` 是一个 `SEL` 类型的值（通常是由 `@selector` 指令产生的）。

表 9.1 处理动态类型的方法

方 法	问题或行为
-(BOOL) isKindOfClass: class-object	对象是不是 class-object 或其子类的成员
-(BOOL) isKindOfClass: class-object	对象是不是 class-object 的成员
-(BOOL) respondsToSelector: Selector	对象是否能够响应 selector 所指定的方法
+(BOOL) instancesRespondToSelector: Selector	指定的类实例是否能响应 selector
+(BOOL) isSubclassOfClass: class-object	对象是否是指定类的子类
-(id) performSelector: selector	应用 selector 指定的方法
-(id) performSelector: selector withObject: object	应用 selector 指定的方法，传递参数 object
-(id) performSelector: selector withObject: object1 withObject: object2	应用 selector 指定的方法，传递参数 object1 和 object2

这里没有描述其他可用的方法，这些方法允许你提出关于是否符合某项协议的问题（将在第 11 章“分类和协议”中讲述），还有一些方法允许你提出有关动态解析方法的问题（本书中不讨论）。

要根据类名或另一个对象生成一个类对象，可以向它发送 class 消息。所以，要从名为 square 的类中获得类对象，可以编写如下代码：

```
[Square class]
```

如果 mySquare 是 Square 对象的实例，可以通过如下代码知道它所属的类：

```
[mySquare class]
```

要查看存储在变量 obj1 和 obj2 中的对象是不是相同的类实例，可以编写如下代码：

```
if ([obj1 class] == [obj2 class])
...

```

要查看变量 myFract 是不是 Fraction 类的实例，可用如下语句测试表达式的结果：

```
[myFract isKindOfClass: [Fraction class]]
```

要生成表 9.1 中列出的所谓的 selector，可以对一个方法名应用 @selector 指令。例如：

```
@selector (alloc)
```

为名为 alloc 的方法生成一个 SEL 类型的值，该方法是从 NSObject 类继承的。

表达式

```
@selector (setTo:over:)
```

为 `setTo:over:` 方法生成一个 `selector`，这个 `setTo:over:` 方法是在 `Fraction` 类中实现的（切记方法名称中的冒号字符）。

要查看 `Fraction` 类的实例是否响应 `setTo:over` 方法，可用如下语句测试表达式的返回值：

```
[Fraction instancesRespondToSelector: @selector (setTo:over:)]
```

记住，测试包括继承的方法，并不是只测试直接定义在类中的方法。

`performSelector:` 方法和它的变体（在表 9.1 中没有显示）允许你向对象发送消息，这条消息可以是存储在变量中的 `selector`。例如，考虑以下代码序列：

```
SEL      action;
id        graphicObject;
...
action = @selector (draw);
...
[graphicObject performSelector: action];
```

在这个例子中，`SEL` 变量 `action` 所指定的方法被发送到存储在 `graphicObject` 中的任何图形对象。根据推测，即使已经把这个行为指定为 `draw`，在程序执行时，行为也可能发生变化，可能依赖于用户的输入。要先确定对象是否可以响应这个动作，可使用以下方式：

```
if ([graphicObject respondsToSelector: action] == YES)
    [graphicObject performSelector: action]
else
    // 错误处理代码
```

注意

在 iOS 中，`respondToSelector:` 方法广泛用于实现委托（delegation）的概念。正如你将在第 10 章“变量和数据类型”中学习的，系统需要你在类中实现一个或多个方法用来响应某些事件和为了提供一些信息（如表格的区块数量）。为了让系统能够检查你确实实现了特定的方法，使用 `respondToSelector:` 判断是否可以将事件的处理委托给你的方法。如果你没有实现这个方法，它会自己处理该事件，按定义的默认行为来执行。

也可以采用其他策略：使用 `forwardInvocation:` 方法将消息转发给其他对象

处理，后面将简要介绍这个技术。

代码清单 9-3 对在第 8 章“继承”中定义的 Square 和 Rectangle 类提出了一些问题。在查看实际的输出之前，请试着预测程序的结果（现在可不能偷看）。

代码清单 9-3

```
#import "Square.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        Square *mySquare = [[Square alloc] init];

        // isMemberOf:

        if ( [mySquare isKindOfClass: [Square class]] == YES )
            NSLog (@"mySquare is a member of Square class");

        if ( [mySquare isKindOfClass: [Rectangle class]] == YES )
            NSLog (@"mySquare is a member of Rectangle class");

        if ( [mySquare isKindOfClass: [NSObject class]] == YES )
            NSLog (@"mySquare is a member of NSObject class");

        // isKindOfClass:

        if ( [mySquare isKindOfClass: [Square class]] == YES )
            NSLog (@"mySquare is a kind of Square");

        if ( [mySquare isKindOfClass: [Rectangle class]] == YES )
            NSLog (@"mySquare is a kind of Rectangle");

        if ( [mySquare isKindOfClass: [NSObject class]] == YES )
            NSLog (@"mySquare is a kind of NSObject");

        // respondsToSelector:

        if ( [mySquare respondsToSelector: @selector (setSide:)] == YES )
            NSLog (@"mySquare responds to setSide: method");

        if ( [mySquare respondsToSelector: @selector (setWidth:andHeight:)] == YES )
            NSLog (@"mySquare responds to setWidth:andHeight: method");

        if ( [Square respondsToSelector: @selector (alloc)] == YES )
            NSLog (@"Square class responds to alloc method");
```

```
// instancesRespondTo:

if ([Rectangle instancesRespondToSelector: @selector (setSide:)] == YES)
    NSLog (@"Instances of Rectangle respond to setSide: method");

if ([Square instancesRespondToSelector: @selector (setSide:)] == YES)
    NSLog (@"Instances of Square respond to setSide: method");

if ([Square isKindOfClass: [Rectangle class]] == YES)
    NSLog (@"Square is a subclass of a rectangle");
}

return 0;
}
```

一定要使用 **Square**、**Rectangle** 和 **XYPoint** 类（它们都出现在第8章中）的实现文件编译这个程序。

代码清单 9-3 输出

```
mySquare is a member of Square class
mySquare is a kind of Square
mySquare is a kind of Rectangle
mySquare is a kind of NSObject
mySquare responds to setSide: method
mySquare responds to setWidth:andHeight: method
Square class responds to alloc method
Instances of Square respond to setSide: method
Square is a subclass of a rectangle
```

代码清单 9-3 的输出应该很清晰。记住 `isMemberOfClass:` 测试类中的直接成员关系，而 `isKindOfClass:` 检测继承层次中的关系。所以，`mySquare` 是 `Square` 类的成员，但是它同样是“某种”`Square`、`Rectangle` 和 `NSObject`，因为它存在于这些类的层次结构中（很明显，所有的对象都应该在有关 `NSObject` 类的 `isKindOfClass:` 测试时返回 YES，除非定义了新的根对象）。

测试语句

```
if ( [Square respondsToSelector: @selector (alloc)] == YES )
```

检测 `Square` 类是否响应 `alloc` 类方法，确实响应了，因为 `Square` 类是从根对象 `NSObject` 继承来的。现在你认识到总能在消息表达式中直接使用类名作为接收者，并且在以前的表达式中不必编写

```
[Square class]
```


不过如果你想编写，也可以编写，这是唯一可以这样做的地方。在其他地方就需要应用 `class` 的方法来获取类对象。

9.6 使用@try 处理异常

好的编程实践是指能预测程序中可能出现的问题。为此，你可以测试使程序异常终止的条件并处理这些情况，可能要记录一条消息并完全终止程序，或者采取其他的正确措施。例如，本章前面讲过如何测试来查看一个对象是否响应特定的消息。以避免错误为例，在程序运行时执行测试可以避免向对象发送未识别的消息。当试图发送这类未识别的消息时，程序通常会立即终止并抛出一个异常。

看一下代码清单 9-4。Fraction 类中未定义任何名为 `noSuchMethod` 的方法。当你编译程序时，就会得到相关的警告消息。

代码清单 9-4

```
#import "Fraction.h"

int main (int argc, char *argv [])
{
    @autoreleasepool {
        Fraction *f = [[Fraction alloc] init];
        [f noSuchMethod];
        NSLog (@@"Execution continues!");
    }
    return 0;
}
```

你可以不管警告消息而继续运行程序。如果这样做，程序可能会异常终止，并出现类似如下的错误：

代码清单 9-4 输出

```
*** Terminating app due to uncaught exception 'NSInvalidArgumentException', reason: '-[Fraction noSuchMethod]: unrecognized selector sent to instance 0x103f00'
*** Call stack at first throw:'''
```

为了避免在这类情况下程序异常终止，可以在一个特殊的语句块中加入一条或多条语句，格式如下：

```
@try {
    statement
    statement
    ...
}
@catch (NSException *exception) {
    statement
    statement
    ...
}
```

在@try 块中加入这些 **statement** 后，程序正常执行。但是，如果块中的某一条语句抛出异常，执行不会终止，而是立即跳到@catch 块，在那里继续执行。在@catch 块内可以处理异常。这里可行的执行顺序是记录出错消息、清除和终止执行。

代码清单 9-5 演示了异常处理。紧跟着的是程序的输出。

代码清单 9-5 异常处理

```
#import "Fraction.h"

int main (int argc, char *argv [])
{
    @autoreleasepool {
        Fraction *f = [[Fraction alloc] init];

        @try {
            [f noSuchMethod];
        }
        @catch (NSException *exception) {
            NSLog(@"Caught %@@", [exception name], [exception reason]);
        }
        NSLog(@"Execution continues!");
    }
    return 0;
}
```

代码清单 9-5 输出

```
*** -[Fraction noSuchMethod]: unrecognized selector sent to instance 0x103280
Caught NSInvalidArgumentException: *** -[Fraction noSuchMethod]:
unrecognized selector sent to instance 0x103280
Execution continues!
```

代码清单 9-5 建立了一个数组，你将在第 15 章“数字、字符串和集合”中

学到，它尝试使用 `NSArray` 类的 `objectAtIndex:` 方法引用一个并不存在的元素。这将导致发生一个异常。

当出现异常时，`@catch` 块被执行。包含异常信息的 `NSException` 对象作为参数传递给这个块。如你所见，`name` 方法检索异常的名称，`reason` 方法给出原因（运行时系统还会将原因自动输出）。

在 `@catch` 块中的最后一条语句执行后（这里只有一条语句），程序会立即继续执行之后的语句。在这个例子中会执行 `NSLog` 语句，验证能够继续执行并没有终止。

这是一个非常简单的例子，演示了如何在程序中捕获异常。无论 `@try` 块中是否抛出异常 `@finally` 块中的语句代码都会执行。

`@throw` 指令允许你抛出自己的异常。可以使用该指令抛出特定的异常，或者在 `@catch` 块内抛出带你进入类似如下代码块的异常：

```
@throw;
```

自行处理异常后（例如，可能是在执行清除工作后），可能需要这么做。然后便可以让系统处理其余的工作。最后，可以使用多个 `@catch` 块按顺序捕获并处理各种异常。

一般来说，你并不希望程序在运行时发生异常。这就需要考虑更好的编程实践，在错误发生之前做测试，而不是错误发生后捕获它。测试方法的错误并返回一些值作为错误的标识，而不是抛出异常。抛出异常通常会使用大量的系统资源，Apple 反对非必要的使用异常（例如，你不希望因为一个文件无法打开而抛出异常）。

9.7 练习

1. 如果在代码清单 9-1 中执行加法之后插入以下消息表达式，将发生什么情况？试一试并查看结果。

```
[compResult reduce];
```

2. 可以将代码清单 9-2 中定义 `id` 变量 `dataValue` 分配给在第 8 章中定义的 `rectangle` 对象吗？就是说，表达式

```
dataValue = [[Rectangle alloc] init];
```

是否合法？为什么？

3. 给第8章中定义的 XYPoint 类中添加一个 print 方法。让它以格式 (x, y) 显示一个点。然后修改代码清单 9-2 来结合一个 XYPoint 对象，使修改后的程序创建一个 XYPoint 对象，设置其值，把这个值分配给 id 变量 dataValue，最后显示它的值。
4. 基于本章关于参数和返回类型的讨论，修改 Fraction 和 Complex 类的 add:方法来选取并返回 id 对象。然后编写一个程序并添加以下代码序列

```
result = [dataValue1 add: dataValue2];  
[result print];
```

其中，result、dataValue1 和 dataValue2 都是 id 对象。确保在程序中适当地设置 dataValue1 和 dataValue2，并且在程序结束之前释放所有的对象。

注意

必须将方法名改为 add:的其他名称。这是因为系统的 NSObjectController 类也有一个 add:方法。如 9.4 节所述，如果在不同的类中有多个同名的方法，并且在编译时不知道接收器的类型，那么编译器就会执行一致性检查，确保参数和返回类型在名称类似的方法之间保持一致。

5. 根据本章中使用的 Fraction 和 Complex 类定义以及如下定义：

```
Fraction *fraction = [[Fraction alloc] init];  
Complex *complex = [[Complex alloc] init];  
id number = [[Complex alloc] init];
```

确定以下消息表达式的返回值。然后将它们输入一个程序，验证结果。

```
[fraction isKindOfClass: [Complex class]];  
[complex isKindOfClass: [NSObject class]];  
[complex isKindOfClass: [NSObject class]];  
[fraction isKindOfClass: [Fraction class]];  
[fraction respondsToSelector:@selector (print)];  
[complex respondsToSelector:@selector (print)];  
[Fraction instancesRespondToSelector:@selector (print)];  
[number respondsToSelector: @selector(print)];  
[number isKindOfClass: [Complex class]];  
[[number class] respondsToSelector:@selector (alloc)];
```


变量和数据类型

本章会深入讨论变量的作用域、对象的初始化方法，以及数据类型。

对象的初始化是本章需要特别关注的内容。

在第 7 章“类”中，已经简要介绍了实例变量、静态变量及局部变量的作用域，这里将更加深入地讲述静态变量，并引入全局变量和外部变量的概念。同时，还会给出一些指令，以便 Objective-C 编译器能够更准确地控制实例变量的作用域，本章内容将讲述这些指令。

枚举数据类型可以为存储一些值的链表定义名称。Objective-C 语言的 typedef 语句允许你为内置或派生的数据类型指定自己的名称。最后，在表达式求值的过程中会详细描述 Objective-C 编译器转换数据类型所遵循的具体步骤。

10.1 对象的初始化

前面已经出现过这种模式，即创建对象的新实例，然后对它进行初始化，正如下面的语句：

```
Fraction *myFract = [[Fraction alloc] init];
```

并不需要编写自己的 init 方法，直接使用继承自父类 NSObject 的方法。

调用这两个方法之后，通常会向这个新对象赋一些值，语句如下：

```
[myFract setTo: 1 over: 3];
```

初始化对象和设置初始值的过程通常可以合并到一个方法中。例如，你可以定义一个 initWith:over: 方法，初始化一个分数，将分子和分母设置为两个给定的参数（没有给出名称）。

包含很多方法和实例变量的类通常还有几个初始化方法。例如，Foundation

框架中的 `NSArray` 类包含以下 6 个额外的初始化方法：

```
initWithArray:
initWithArray:copyItems:
initWithContentsOfFile:
initWithContentsOfURL:
initWithObjects:
initWithObjects:count:
```

可以使用下面的语句序列完成数组的空间分配和初始化工作：

```
myArray = [[NSArray alloc] initWithArray: myOtherArray];
```

常见的编程习惯是类中所有的初始化方法都以 `init` 开头。可以看到，`NSArray` 的初始化方法遵循这个惯例。编写初始化方法时，应该遵循以下两个策略。

如果希望在类对象初始化时做一些事情。例如，在创建类的对象时需要使用 and 引用一个或多个实例变量，比如 `Rectangle` 类，在 `init` 方法中需要为矩形指定 `CGPoint` 原点。可以通过重载 `init` 方法达到这个目的。

这是重载 `init` 方法的一个标准“模板”，看起来是这样的：

```
- (instancetype)init
{
    self = [super init];
    if (self) {
        // 初始化代码
    }

    return self;
}
```

这个方法首先会调用父类的初始化方法。执行父类的初始化方法，使得继承的实例变量能够正常初始化。

必须将父类 `init` 方法的执行结果赋值给 `self`，因为初始化过程改变了对象在内存中的位置（意味着引用将要改变）。

如果父类的初始化过程成功，返回的值将是非空的，通过 `if` 语句可以验证。注释说明可以在这个代码块的位置放入自定义的初始化代码。通常可以在这个位置创建并初始化实例变量。

注意

特殊类型 `instancetype` 表明, 从 `init` 方法返回的类型与它的初始化类 (也就是初始化消息的接收者) 相同。这为编译器提供了辅助, 它可以检查你的程序和标记的潜在类型是否匹配 - 决定了基于上下文对象的返回类型, 也就是说, 如果你发送初始化消息给新分配的 `Fraction` 对象, 编译器会推断出 `init` 方法的返回值 (其返回类型已声明为 `instancetype`) 将是一个 `Fraction` 对象。在过去, 初始化方法的返回类型声明为 `id` 类型。在你考虑子类时这种新型的类型声明更有意义, 因为被继承的初始化方法不能明确定义它们需要返回的对象类型。

如果你的类包含多个初始化方法, 其中一个就应该是指定的 (`designated`) 初始化方法, 并且其他所有的初始化方法都应该使用这个方法。通常, 它是最复杂的初始化方法 (一般是参数最多的初始化方法)。通过创建指定的初始化方法, 可以把大部分初始化代码集中到单个方法中。然后, 任何人要想从这个类派生子类, 都可以重载这个指定的初始化方法, 以保证正确地初始化新的实例。

基于这个讨论, `Fraction` 类的初始化方法 `initWithOver:` 可能如下:

```
-(Fraction *) initWith: (int) n over: (int) d
{
    self = [super init];

    if (self)
        [self setTo: n over: d];

    return self;
}
```

完成 `super` 的初始化 (返回的非零值表示初始化成功) 之后, 使用 `setTo:over:` 方法设置 `Fraction` 的分子和分母。和其他初始化方法一样, 希望由你返回初始化的对象, 这里就是这样做的。

代码清单 10-1 测试了新的初始化方法 `initWithOver:`。

代码清单 10-1

```
#import "Fraction.h"

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



```

@autoreleasepool {

    Fraction *a, *b;

    a = [[Fraction alloc] initWith: 1 over: 3];
    b = [[Fraction alloc] initWith: 3 over: 7];

    [a print];
    [b print];

    [a release];
    [b release];

}
return 0;
}

```

代码清单 10-1 输出

```

1/3
3/7

```

为了使用指定的初始化规则，你需要修改 **Fraction** 类的 **init** 方法。这一点对于子类来说特别重要。

init 方法如下：

```

-(instancetype) init
{
    return [self initWith: 0 over: 0];
}

```

再一次注意，**init** 被定义为返回 **instancetype** 类型，这是编写可能会被继承的类 **init** 方法的一般规则。你并不想硬编码一个类名，比如子类的对象并不等同于父类。

程序开始执行时，它向所有的类发送 **initialize** 调用方法。如果存在一个类及相关的子类，则父类首先得到这条消息。这个消息只向每个类发送一次，并且向类发送其他任何消息之前，保证向其发送初始化消息。这样做的目的是程序开始时能够执行到所有类的初始化工作。例如，你可能希望在开始时初始化与类相关的静态变量。

10.2 作用域回顾

可以使用多种方式影响程序中变量的作用域。可以改变实例变量及定义在函数外部或内部的普通变量的作用域。在下面的讨论中,我们使用模块(module)来引用包含在一个源文件中任何数目的方法或者函数定义。

10.2.1 关于属性、存取方法和实例变量

目前的趋势是使用下划线(_)作为实例变量名字的起始字符。通过 Xcode 生成的模板代码可以看到,引用实例变量的变量名字是以“_”开头的。

遇到的@synthesize 指令如下:

```
@synthesize window=_window;
```

表明合成(synthesize)属性 window 的取值方法和设值方法,并将属性与实例变量 _window (实例变量并没有显性声明)关联起来。这对区别属性和实例变量的使用是有帮助的,鼓励通过设值方法来设置实例变量的值,通过取值方法获取实例变量。比如

```
[window makeKeyAndVisible]; // 无法运行
```

就会失败,因为没有实例变量命名为 window。而是使用

```
[_window makeKeyAndVisible];
```

或者,最好使用获取函数:

```
[self.window makeKeyAndVisible];
```

在实现部分显式地声明实例变量(或者使用@synthesize 指令隐性声明的实例变量)是私有的,这就意味着并不能在子类中通过名字直接获取到实例变量。在子类中只能使用继承的存取方法获取实例变量的值。

基于它们的属性,合成方法还能做其他一些事情(例如,管理内存、复制值等),给实例变量赋值或者从实例变量中获取值就不会做这些事情。这是属性和实例变量之间另一个水平的抽象,这种抽象使得当存取实例变量时系统有机会做其他工作(这些工作不一定会被注意到)。

如第 7 章中所说的,你不必用@synthesize 指令来生成你的存取方法。你只要在属性中声明它们后,编译器不用这个指令也会生成这些方法。

10.2.2 全局变量

如果编写如下语句：

```
int gMoveNumber = 0;
```

那么这个模块中的任何位置都可以引用这个变量的值。这种情况下，我们说 `gMoveNumber` 被定义为全局变量。为了向阅读程序的人说明变量的作用域，按照惯例，用小写的 `g` 作为全局变量的首字母。

实际上，这样的定义使得其他文件也可以访问变量 `gMoveNumber` 的值。确切地说，前面的语句不仅将 `gMoveNumber` 定义为全局变量，而且将其定义为外部全局变量。

外部变量是可以被其他任何方法或函数访问和更改值的变量。在需要访问外部变量的模块中，变量声明和普通方式一样，只是需要在声明前加上关键字 `extern`。这就告知系统，要访问其他文件中定义的全局变量。下面这个例子说明如何将 `gMoveNumber` 声明为外部变量：

```
extern int gMoveNumber;
```

现在，包含前面这个声明的模块就可以访问和改变 `gMoveNumber` 的值。同样，通过在文件中使用类似的 `extern` 声明，其他模块也可以访问 `gMoveNumber` 的值。

使用外部变量时，必须遵循下面这条重要的原则：变量必须定义在源文件中的某个位置。即在所有的方法和函数之外声明变量，并且前面不加关键字 `extern`，语句如下：

```
int gMoveNumber;
```

这里，如前面所示，可以选择为这个变量赋初值。

确定外部变量的第二种方式是在所有的函数之外声明变量，在声明前面加上关键字 `extern`，同时显式地为变量指派初始值，语句如下：

```
extern int gMoveNumber = 0;
```

然而，这并不是首选的方法，编译器将给出警告消息，提示你已将变量声明为 `extern`，并同时为变量赋值。这是因为使用关键字 `extern` 表明这条语句是变量的声明，而不是定义。记住，声明不会引起变量存储空间分配，而定义会引起变量存储空间的分配。前面的例子强行将声明当作定义处理（通过指派初

始值)，所以违背了这个规则。

处理外部变量时，变量可以在许多地方声明为 **extern**，但是只能定义一次。在此通过观察一个小程序例子来说明外部变量的用法。假设我们定义了一个名为 **Foo** 的类，并将以下代码输入一个名为 **main.m** 的文件中：

```
#import "Foo.h"

int gGlobalVar = 5;

int main (int argc, char *argv[])
{
    @autoreleasepool {
        Foo *myFoo = [[Foo alloc] init];
        NSLog ("%i ", gGlobalVar);

        [myFoo setgGlobalVar: 100];
        NSLog ("%i", gGlobalVar);
    }
    return 0;
}
```

在前面的程序，**gGlobalVar** 定义为全局变量，因此，任何方法（或函数）只要正确地使用 **extern** 声明，都可以访问它。假设 **Foo** 方法 **setgGlobalVar** 如下所示：

```
-(void) setgGlobalVar: (int) val
{
    extern int gGlobalVar;
    gGlobalVar = val;
}
```

这个程序将在终端生成以下结果：

```
5
100
```

这就证明了方法 **setGlobalVar** 可以访问和改变外部变量 **gGlobalVar** 的值。

如果有很多方法需要访问 **gGlobalVar** 的值，只在文件的开始进行一次 **extern** 声明比较简便。但是，如果只有一个或少数几个方法要访问这个变量，就应该在其中的每个方法中单独进行 **extern** 声明。这样程序的组织结构更加清晰，并且实际使用到变量的不同函数可以单独使用这个变量。注意，如果变量定义在包含访问这个变量的文件中，那么不需要单独进行 **extern** 声明。

10.2.3 静态变量

前面所示的例子与数据封装原则及良好的面向对象编程技术相违背。然而，可能需要这种变量，它们的值经过不同的方法调用时是共享的。虽然在 `Foo` 类中将 `gGlobalVar` 定义为实例变量似乎也不太合理，但是，更好的方法可能是使变量的访问限制在类定义的 `setter` 和 `getter` 方法，将实例变量“隐藏”在 `Foo` 类中。

现在，你知道在方法之外定义的变量不仅是全局变量，而且是外部变量。然而很多情况你想要将变量定义为全局变量，但不是外部变量。换句话说，希望定义的全局变量只在特定模块（文件）中是全局的。这种变量在某一种情况中很有意义，即除了特定类中的方法之外，再没有其他方法需要访问这个特定的变量。要做到这一点，可以在包含这个特定类实现的文件中将这个变量定义为 `static`。

如果语句

```
static int gGlobalVar = 0;
```

声明在任何方法（或函数）之外，那么在该文件中，所有位于这条语句之后的方法或函数都可以访问 `gGlobalVar` 的值，而其他文件中的方法和函数则不行。

你会想起类方法不能访问实例变量（你可能会想为什么又是这样）。然而，你可能希望类的方法可以设定和访问一些变量。简单的例子是类的分配器方法，它要记录类已经分配空间的对象数目。实现这项任务的方式是在类的实现代码文件中设定静态变量。由于这个变量不是实例变量，所以分配器方法可以直接访问它。类的用户不用知道这个变量，因为它是定义在实现文件中的静态变量，作用域只是文件内部。因此，用户不能直接访问该变量，也就没有违背数据封装的概念。如果需要从类之外访问该变量，则可以编写一个方法来获取该变量的值。

代码清单 10-2 对 `Fraction` 的类定义进行了扩充，增加了两个新方法。`allocF` 类方法分配一个新的 `Fraction` 对象，同时记录分配了多少 `Fraction`，`count` 方法则返回这个数的值。注意，后者也是类方法，也可以作为实例方法实现，然而，与向类的特定实例发送消息相比，询问类已经分配了多少实例更有意义。

下面是在头文件 `Fraction.h` 中添加的这两个新的类方法声明：

```
+(Fraction *) allocF;
```

```
+(int) count;
```

你可能注意到，继承来的 `alloc` 方法并没有被重载，而是定义了自己的分配器方法。这个方法将利用继承来的 `alloc` 方法。下面是需要在实现文件 `Fraction.m` 中加入的代码：

```
static int gCounter;

@implementation Fraction

+(Fraction *) allocF
{
    extern int gCounter;
    ++gCounter;

    return [Fraction alloc];
}

+(int) count
{
    extern int gCounter;

    return gCounter;
}

// Fraction 类的其他方法
...
@end
```

注意

重载 `alloc` 并不是好的编程实践，因为这个方法处理内存的物理分配。你没有必要研究得那么深。

声明 `gCounter` 为静态，使得定义在执行文件中的方法可以访问它，但是文件之外都不可以访问。`allocF` 方法仅仅递增 `gCounter` 变量，然后使用 `alloc` 方法创建一个新的 `Fraction`，并返回结果。`count` 方法只是返回计数器的值，这样就隔离了来自用户的直接访问。

回忆一下，因为 `gCounter` 变量定义在该文件中，因此，不需要在这两个方法中使用 `extern` 声明。声明只是为了让阅读该方法的人明白，访问的变量是定义在方法之外的。变量名加 `g` 前缀也是出于同样的目的。因此，大多数程序员一般不使用 `extern` 声明。

代码清单 10-2 测试了这个新方法。

代码清单 10-2

```
#import "Fraction.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        Fraction *a, *b, *c;

        NSLog(@"Fractions allocated: %i", [Fraction count]);

        a = [[Fraction allocF] init];
        b = [[Fraction allocF] init];
        c = [[Fraction allocF] init];

        NSLog(@"Fractions allocated: %i", [Fraction count]);
    }
    return 0;
}
```

代码清单 10-2 输出

```
Fractions allocated: 0
Fractions allocated: 3
```

程序开始运行时，`gCounter` 的值会自动置为 0（你应该还记得，如果要将类作为整体进行任何特殊的初始化，例如，将其他静态变量的值设置为一些非零值，可以重载继承类的 `initialize` 方法）。在 `allocF` 方法分配 3 个 `Fraction` 实例之后，使用 `count` 方法检索 `counter` 变量的值，变量 `counter` 被正确地设置为 3。如果要重置计数器或将其设为特定的值，可以在类中添加一个 `setter` 方法。然而，在这个程序中并不需要这么做。

10.3 枚举数据类型

在 Objective-C 语言中可以将一系列值指派给一个变量。枚举数据类型的定义以关键字 `enum` 开头，之后是枚举数据类型的名称，然后是标识符序列（包含在一对花括号内），定义了所有允许给这个类型指派的值。例如，语句

```
enum flag { false, true };
```

定义了一个数据类型 `flag`。从理论上说，这个数据类型在程序中只能赋为 `true` 或 `false`，不能赋其他值。遗憾的是，即使违背了这个规则，Objective-C 编译器

也不会发出警告消息。

要声明一个 `enum flag` 类型的变量，仍需要用到关键字 `enum`，之后是枚举类型名称，最后是变量序列。所以语句

```
enum flag endOfData, matchFound;
```

定义了两个 `flag` 类型的变量 `endOfData` 和 `matchFound`。能指派给这两个变量的值只有（理论上是这样）`true` 和 `false`。因此，

```
endOfData = true;
```

和

```
if ( matchFound == false )
    ...
```

之类的语句都是合法的。

如果希望一个枚举标识符对应一个特定的整数值，那么可以在定义数据类型时给这个标识符指定整数值。列表中随后出现的枚举标识符被依次赋以整数值，从指定的整数值加 1 开始。

在定义

```
enum direction { up, down, left = 10, right };
```

中，定义了一个包含值 `up`、`down`、`left` 和 `right` 的枚举数据类型 `direction`。因为 `up` 位于序列首位，所以编译器给它赋值为 0；`down` 接着 `up`，因此，赋给它的值为 1；由于 `left` 明确指定了一个整数，赋给它的值就是 10；`right` 的值由列表中前一个 `enum` 的值递增得到，因此，给它赋的值为 11。

枚举标识符可以共享相同的值。例如，

```
enum boolean { no = 0, false = 0, yes = 1, true = 1 };
```

给 `enum boolean` 变量指派 `no` 和 `false` 时，就是向其赋值 0，指派 `yes` 和 `true` 时赋值 1。

再举另一个枚举数据类型定义的例子。下面定义了类型 `enum month`，对于这种类型的变量，可以指派的值为一年中 12 个月的名字：

```
enum month { january = 1, february, march, april, may, june, july,
            august, september, october, november, december };
```

Objective-C 编译器实际上将枚举标识符作为整型常量来处理。如果你的程

序包含以下两行

```
enum month thisMonth;
...
thisMonth = february;
```

那么给 `thisMonth` 赋的值是整数 2（而不是 `february` 这个名字）。

代码清单 10-3 展示了使用枚举数据类型的简单程序。该程序首先读取一个月份数，然后进入 `switch` 语句判断要进入哪个月份。回忆一下，编译器把枚举值当作整型常量处理，因此它们都是有效的 `case` 值。将变量 `days` 赋值为这个月的天数，在 `switch` 退出后显示 `days` 的值。程序中包含特定的测试，用以查看这个月是否为二月。

代码清单 10-3

```
#import <Foundation/Foundation.h>
// 打印每月的天数
int main (int argc, char *argv[])
{
    @autoreleasepool {
        enum month { january = 1, february, march, april, may, june,
                     july, august, september, october, november,
                     december };
        enum month amonth;
        int    days;

        NSLog(@"Enter month number: ");
        scanf ("%i", &amonth);

        switch (amonth) {
            case january:
            case march:
            case may:
            case july:
            case august:
            case october:
            case december:
                days = 31;
                break;
            case april:
            case june:
            case september:
            case november:
                days = 30;
                break;
```

```

        case february:
            days = 28;
            break;
        default:
            NSLog(@"bad month number");
            days = 0;
            break;
    }

    if ( days != 0 )
        NSLog(@"Number of days is %i", days);

    if ( amonth == february )
        NSLog(@"...or 29 if it's a leap year");
}
return 0;
}

```

代码清单 10-3 输出

```

Enter month number:
5
Number of days is 31

```

代码清单 10-3 输出（再次运行）

```

Enter month number:
2
Number of days is 28
...or 29 if it's a leap year

```

可以明确地给枚举类型的变量赋予一个整数值,应该使用类型转换运算符。因此,如果 **monthValue** 是值为 6 的整型变量,那么表达式

```
lastMonth = (enum month) (monthValue - 1);
```

是允许的。如果不使用类型转换运算符,编译器也不会有异议(很遗憾)。

使用包含枚举数据类型的程序时,尽量不要依赖枚举值被作为整数这个事实。相反,把它们当作独立的数据类型。枚举类型提供了一种方法,使你能把整数值和有象征意义的名称对应起来。如果以后需要更改这个整数的值,只能在定义枚举的地方更改。如果根据枚举数据类型的实际值进行假设,就丧失了使用枚举带来的好处。

在定义枚举类型时你可以将一个整数类型和一个枚举名称对应起来。例如

下面定义的枚举类型 `iPhoneModels` 使用了 `unsigned short int` 作为相关数据类型：

```
enum iPhoneModels : unsigned short int { iPhone, iPhone3G,
iPhone3GS, iPhone4 iPhone4S, iPhone5, iPhone5C, iPhone5S };
```

编译器选项 `(-Wconversion)` 和 `(-Wswitch)` 已经被加入，当赋值和使用枚举类型表达式以及枚举常量应用在 `switch` 语句时，以提供更好的类型检查。

定义枚举数据类型时，也允许有所变化：可以省略数据类型的名称，定义类型可以将变量声明为特定枚举数据类型中的一个。举出一个能够同时体现的例子，语句

```
enum { east, west, south, north } direction;
```

定义了一个（未命名的）枚举数据类型，它包含的值为 `east`、`west`、`south` 和 `north`，同时还声明了该类型的变量 `direction`。

在代码块中定义的枚举数据类型作用域限于块的内部。另一方面，在程序的开始及所有块之外定义的枚举数据类型对于文件是全局的。

定义枚举数据类型时，必须确保枚举标识符与定义在相同作用域之内的变量名和其他标识符不同。

10.4 typedef 语句

Objective-C 允许编程者为数据类型另外指派一个名称。这是通过 `typedef` 语句实现的。语句

```
typedef int Counter;
```

定义名称 `Counter` 等价于 Objective-C 数据类型 `int`。随后的变量就可以声明为 `Counter` 类型，如在以下语句

```
Counter j, n;
```

中，Objective-C 编译器实际上是将变量 `j` 和 `n` 的声明当作前面显示的普通整型变量。在这种情况下使用 `typedef` 语句的主要好处是增加了变量定义的可读性。从 `j` 和 `n` 的定义中就可以清晰地看出这些变量在程序中的使用目的。用传统方式将变量定义为 `int` 类型不能很清晰地表示出它们的用途。

下面的 `typedef` 定义了一个名为 `NumberObject` 的类型，它是 `Number` 对象：

```
typedef Number *NumberObject;
```

随后将一些变量声明为 `NumberObject` 类型，如在语句

```
NumberObject myValue1, myValue2, myResult;
```

中，它们的使用方式和以常规方式在程序中声明一样，语句如下：

```
Number *myValue1, *myValue2, *myResult;
```

若要使用 `typedef` 定义一个新类型名，可按照下面的步骤：

- (1) 像声明所需类型的变量那样编写一条语句。
- (2) 通常应该出现声明变量名的地方，将其替换为新的类型名。
- (3) 在语句的前面加上关键字 `typedef`。

作为这个过程的例子，定义一个名为 `Direction` 的枚举数据类型，它包含 4 个方向：东、南、西和北。写出枚举类型的声明，在通常出现变量名称的地方使用 `Direction` 替代。在开始其他工作之前，语句前加上关键字 `typedef`：

```
typedef enum { east, west, south, north } Direction;
```

将 `typedef` 放在合适的位置之后，就可以声明 `Direction` 类型的变量了，如下语句所示：

```
Direction step1, step2;
```

10.5 数据类型转换

第 4 章“数据类型和表达式”中曾经简要说明，表达式求值的过程中，系统有时会进行隐式的数据类型转换。已知的例子是 `float` 和 `int` 数据类型。你看到了涉及一个 `float` 和 `int` 的运算如何作为浮点运算进行求值，整型数被自动转换成为浮点型。

还看到了如何使用类型转换运算符显式地控制转换。因此，假设 `total` 和 `n` 都是整型变量

```
average = (float) total / n;
```

在运算之前，变量 `total` 的值被转换为 `float` 型，这就保证了将除法运算作为浮点运算求值。

10.5.1 转换规则

对含有不同类型数据的表达式求值时，Objective-C 编译器会遵循一些非常

严格的规则。

下面总结了表达式求值的过程中,不同类型的操作数发生转换的先后顺序:

(1)如果其中一个操作数是 `long double` 型,另一操作数被转换为 `long double` 型,则计算结果也是这种类型。

(2)如果其中一个操作数是 `double` 型,另一操作数转换为 `double` 型,则计算结果也是这种类型。

(3)如果其中一个操作数是 `float` 型,另一操作数转换为 `float` 型,则计算结果也是这种类型。

(4)如果其中一个操作数是 `Bool`、`char`、`short int`、`bit field` 或枚举数据类型,则全部转换为 `int` 型。

(5)如果其中一个操作数是 `long long int` 型,另一操作数转换为 `long long int` 型,则计算结果也是这种类型。

(6)如果其中一个操作数是 `long int` 型,则另一操作数转换为 `long int` 型,计算结果也是这种类型。

(7)如果到达这一步,则可知两个操作数均为 `int` 型,计算结果也是这种类型。

以上列出的实际上只是一个简化版本,说明了表达式操作数类型转换过程所涉及的步骤。涉及 `unsigned` 操作数时,规则还会更复杂。在这里列的还不够详细。

从这一系列步骤中可以认识到,只要到达“计算结果也是这种类型”,转换过程就结束了。

举一个例子说明如何遵循这些步骤。观察以下表达式的求值方式。其中, f 定义为 `float` 变量, i 为 `int` 变量, l 为 `long int` 变量, s 为 `short int` 变量:

```
f * i + l / s
```

首先,考虑 f 和 i 的相乘运算,这是 `float` 和 `int` 相乘。由步骤 (3) 可知,由于 f 是 `float` 型,另外一个操作数 (i) 会被转换为 `float` 类型,相乘的结果也是 `float`。

其次,考虑 l 和 s 的除法运算,这是 `long int` 除以 `short int`。由步骤 (4) 可知, `short int` 会被转换为 `int`。然后,由步骤 (6) 表明,由于其中一个操作数 (1)

是 `long int` 类型，所以另一个操作数将被转换为 `long int`，计算结果也是这个类型。因此，这个除法的结果是 `long int` 类型，实际上删节了除法运算结果的小数部分。

最后，按照步骤 (3)，如果表达式中一个操作数为 `float` 类型（变量 `f` 和 `i` 相乘的结果），则另一个操作数将被转换为 `float` 类型，运算结果也是 `float` 类型。因此，在 `l` 和 `s` 的除法运算完成之后，其结果将被转换为 `float` 类型，然后与 `f` 和 `i` 的乘积相加。整个表达式的最终计算结果是 `float` 类型的值。

记住，总是可以用类型转换运算符显式地强制类型转换，从而控制特定表达式的求值方式。

因此，在前面的表达式求值的过程中，如果不希望 `l` 和 `s` 相除得到截取的结果，可以将其中一个变量转换为 `float` 类型，这样就能强制表达式作为浮点除法求值，语句如下：

```
f * i + (float) l / s
```

在这个表达式中，由于类型转换运算符比除法运算符的优先级高，所以 `l` 首先转换为 `float`，然后进行除法运算。因为这个除法运算中的一个操作数变成了 `float` 类型。所以，其他一个或多个操作数也会自动转成 `float` 类型，计算结果也是这个类型。

10.6 位运算符

Objective-C 语言中有各种各样的运算符可处理数字中的特定位。表 10.1 列出了这些运算符。

表 10.1 位运算符

符 号	运 算	符 号	运 算
&	按位与	~	一次求反
	按位或	<<	向左移位
^	按位异或	>>	向右移位

注意

你并不会大量使用到这些位运算符，尽管你会在 Objective-C 程序的框架头文件中遇到。但这些内容对于新程序员有些难度，你可以暂时将这些内

容放下，如果有需要，再查阅。

表 10.1 列出的所有运算符中，除一次求反运算符（~）外，都是二元运算符，因此，需要两个运算数。位运算符可处理任何类型的整型值，但不能处理浮点值。

在下面的示例中，你会学习到如何在二进制和十六进制表示法之间进行转换。十六进制（以 16 为基数）数字由 4 位组成。表 10.2 显示了如何在这些基数之间进行转换。

表 10.2 二进制数、十进制数和十六进制数

二进制数	十进制数	十六进制数	二进制数	十进制数	十六进制数
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	a
0011	3	3	1011	11	b
0100	4	4	1100	12	c
0101	5	5	1101	13	d
0110	6	6	1110	14	e
0111	7	7	1111	15	f

10.6.1 按位与运算符

对两个值执行“与”运算时，会逐位比较两个值的二进制数表示。第一个值与第二个值对应位都为 1 时，在结果的对位上就会得到 1；其他的组合在结果中都得到 0。如果 b1 和 b2 表示两个运算数的对应位，那么下表（称为真值表）就显示了在 b1 和 b2 所有可能的值下对 b1 和 b2 执行“与”操作的结果。

b1	b2	b1 & b2
0	0	0
0	1	0
1	0	0
1	1	1

例如，如果 w1 和 w2 都定义为 short int，w1 等于十六进制数的 15，w2 等于十六进制数的 0c，那么以下 C 语句会将值 0x04 指派给 w3：

```
w3 = w1 & w2;
```

将 w1、w2 和 w3 都表示为二进制数后，可更清楚地看到此过程。假设所处理的 short int 大小为 16 位：

w1	0000 0000 0001 0101	0x15
w2	0000 0000 0000 1100	& 0x0c
w3	0000 0000 0000 0100	0x04

按位“与”运算经常用于屏蔽运算。就是说，该运算符可轻易地将数据项的特定位设置为 0。例如，语句

```
w3 = w1 & 3;
```

将 w1 与常量 3 按位“与”所得的值指派给 w3。它的作用是将 w3 中的全部位（而非最右边的两位）设置为 0，并保留 w1 中最左边的两位。

与 Objective-C 中使用的所有二元运算符相同，通过添加等号，二元位运算符可同样用作赋值运算符。因此，语句

```
word &= 15;
```

与下列语句

```
word = word & 15;
```

执行相同的功能。

此外，它还能将 word 的全部位设置为 0，但最右边的 4 位除外。

10.6.2 按位或运算符

在 Objective-C 中对两个值执行按位“或”运算时，会逐位比较两个值的二进制数表示。此时，只要第一个值或者第二个值的相应位是 1，那么结果的对应位就是 1。按位“或”操作符的真值表如下：

b1	b2	b1 b2
0	0	0
0	1	1
1	0	1
1	1	1

所以，如果 w1 是 short int，等于十六进制数的 19，w2 也是 short int，等于十六进制数的 6a，那么对 w1 和 w2 执行按位“或”会得到十六进制数的 7b，表示如下：


```
w1  0000 0000 0001 1001      0x19
w2  0000 0000 0110 1010      | 0x6a
```

```
0000 0000 0111 1011      0x7b
```

按位“或”操作通常称为按位 OR，用于将某个词的特定位置设为 1。例如，以下语句将 w1 最右边的 3 位设为 1，而不管这些位操作前的状态是什么，都是如此。

```
w1 = w1 | 07;
```

当然，可以在语句中使用特殊的赋值运算符，如下面的语句：

```
w1 |= 07;
```

我们在后面会提供一个程序例子，演示如何使用按位“或”运算符。

10.6.3 按位异或运算符

按位异或运算符通常称为 XOR 运算符，遵守以下规则：对于两个运算数的相应位，如果任何一个位是 1，但不是两者都为 1，那么结果的对应位将是 1，否则是 0。该运算符的真值表如下：

```
b1    b2    b1 ^ b2
```

```
0      0      0
0      1      1
1      0      1
1      1      0
```

如果 w1 和 w2 分别等于十六进制数的 5e 和 d6，那么 w1 与 w2 执行异或运算后的结果是十六进制数 e8，表示如下：

```
w1  0000 0000 0101 1110      0x5e
w2  0000 0000 1011 0110      ^ 0xd6
```

```
0000 0000 1110 1000      0xe8
```

10.6.4 一次求反运算符

一次求反运算符是一元运算符，它的作用仅是对运算数的位“翻转”，将运算数中每个是 1 的位翻转为 0，而将每个是 0 的位翻转为 1。此处提供真值表只是为了保持内容的完整性。

```
b1    ~b1
```

```
0   1
1   0
```

如果 `w1` 是 `short int`, 16 位长, 等于十六进制数 `a52f`, 那么对该值执行一次求反运算会得到十六进制数 `5ad0`:

```
w1  1010 0101 0010 1111    0xa52f
~w1 0101 1010 1101 0000    0x5ad0
```

如果不知道运算中数值的准确位大小, 那么一次求反运算符非常有用, 使用它可让程序不会依赖于整数数据类型的特定大小。例如, 要将类型为 `int` 的 `w1` 的最低位设为 0, 可将一个所有的位都是 1, 但最右边的位是 0 的 `int` 值与 `w1` 进行“与”运算。所以像下面这样的 C 语句在用 32 位表示整数的机器上可正常工作。

```
w1 &= 0xFFFFF000;
```

如果用

```
w1 &= ~1;
```

替换上面的语句, 那么 `w1` 在任何机器中都会和正确的值进行“与”运算。

这是因为这条语句会对 1 求反, 然后在左侧会加入足够多的 1, 以满足 `int` 的大小要求 (在 64 位机器上, 会在左侧的 63 个位上加入 1)。

下面给出一个实际的程序例子, 说明各种位运算符的用途 (参见代码清单 10-4)。

代码清单 10-4

// 按位运算符的表示

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        unsigned int w1 = 0xA0A0A0A0, w2 = 0xFFFF0000,
            w3 = 0x00007777;

        NSLog (@"%x %x %x", w1 & w2, w1 | w2, w1 ^ w2);
        NSLog (@"%x %x %x", ~w1, ~w2, ~w3);
        NSLog (@"%x %x %x", w1 ^ w1, w1 & ~w2, w1 | w2 | w3);
        NSLog (@"%x %x", w1 | w2 & w3, w1 | w2 & ~w3);
    }
}
```

```

    NSLog(@"%x %x", ~(~w1 & ~w2), ~(~w1 | ~w2));
}
return 0;
}

```

代码清单 10-4 输出

```

a0a00000 ffffa0a0 5f5fa0a0
5f5f5f5f ffff ffff8888
0 a0a0 fffff7f7
a0a0a0a0 ffffa0a0
ffffa0a0 a0a00000

```

请将代码清单 10-4 中的每个运算都演算一遍，确定你理解了这些结果是如何得到的。

在第 4 个 NSLog 调用中，需要注意重要的一点，即按位“与”运算符的优先级要高于按位“或”运算符，因为这会实际影响表达式的最终结果值。

第 5 个 NSLog 调用展示了 DeMorgan 的规则： $\sim(\sim a \& \sim b)$ 等于 $a|b$ ， $\sim(\sim a|\sim b)$ 等于 $a\&b$ 。

10.6.5 向左移位运算符

对值执行向左移位运算时，顾名思义，值中包含的位将向左移动。与该操作关联的是该值要移动的位置（或位）数目。超出数据项的高位的位将丢失，而从低位移入的值总为 0。因此，如果 w1 等于 3，那么表达式

```
w1 = w1 << 1;
```

可同样表示成

```
w1 <<= 1;
```

结果就是 3 向左移一位，这样产生的 6 将赋值给 w1

```

w1      ... 0000 0011   0x03
w1 << 1 ... 0000 0110   0x06

```

“<<”运算符左侧的运算数表示将要移动的值，而右侧的运算数表示该值所需移动的位数。

如果将 w1 再向左移动一次，那么会得到十六进制数 0c:

```

w1      ... 0000 0110   0x06
w1 << 1 ... 0000 1100   0x0c

```

10.6.6 向右移位运算符

顾名思义，向右移位运算符（>>）是把值的位向右移动，从值的低位移出的位将丢失。把无符号的值向右移位总是左侧（就是高位）移入 0。对于有符号值而言，左侧移入 1 还是 0 取决于被移动数字的符号，还取决于该操作在计算机上的实现方式。如果符号位是 0（表示该值是正的），不管是哪种机器，都将移入 0。然而，如果符号位是 1，那么在一些计算机上将移入 1，而其他计算机上则移入 0。前一类型的运算符通常称为算术右移，而后者通常称为逻辑右移。

警告

对于系统使用算术右移还是逻辑右移，千万不要进行猜测。如果进行此类的假设，那么在一个系统中可正确进行有符号右移运算的程序，有可能在其他系统上运行失败。

如果 `w1` 是 `unsigned int`，用 32 位表示它，并且 `w1` 等于十六进制数的 `F777EE22`，那么使用语句

```
w1 >>= 1;
```

将 `w1` 右移一位后，`w1` 等于十六进制数的 `7BBBF711`，表示如下：

```
w1      1111 0111 0111 0111 1110 1110 0010 0010    0xF777EE22
w1 >> 1 0111 1011 1011 1011 1111 0111 0001 0001    0x7BBBF711
```

如果将 `w1` 声明为（有符号的）`short int`，在某些计算机上会得到相同的结果，而在其他计算机上，如果将该运算作为算术右移来执行，结果将会是 `FB BB F711`。

应该注意到，如果试图用大于或等于该数据项的位数将值向左或向右移位，那么该 Objective-C 语言对结果没有规定。因此，如果计算机用 64 位表示整数，那么把一个整数向左或向右移动 64 位或更多位时，不能保证在你的程序中得到确定的结果。还应注意到，如果使用负数对值移位，结果将同样是未定义的。

10.7 练习

1. 使用第 8 章“继承”中的 `Rectangle` 类，根据下面的声明增加一个初始化方法：

（注意：一定要使用这个初始化器重载 `init`。）

```
-(id) initWithWidth: (int) w andHeight: (int) h;
```

2. 假设将练习 1 中的初始化方法标记为 `Rectangle` 类的指定初始化方法，根据第 8 章定义的 `Square` 和 `Rectangle` 类，结合下面的声明，为 `Square` 类增加一个初始化方法：

```
-(id) initWithSide: (int) side;
```

3. 为 `Fraction` 类的 `add` 方法增加一个计数器来计算它的调用次数。如何获取这个变量的值？
4. 使用 `typedef` 定义一个名为 `Day` 的类型，可能的值为 `Sunday`、`Monday`、`Tuesday`、`Wednesday`、`Thursday`、`Friday` 和 `Saturday`。
5. 使用 `typedef` 和枚举数据类型定义名为 `FractionObj` 的类型，该类型允许编写如下语句：

```
FractionObj f1 = [[Fraction alloc] init],
            f2 = [[Fraction alloc] init];
```

6. 根据下面的定义：

```
float      f = 1.00;
short int  i = 100;
long int   l = 500L;
double     d = 15.00;
```

和本章讲解表达式中操作数类型转换时列举的 7 个步骤，确定以下表达式的类型和值：

```
f + i
l / d
i / l + f
l * i
f / 2
i / (d + f)
l / (i * 2.0)
l + i / (double) l
```

分类和协议

在本章中，你将学习如何通过分类（category）以模块的形式向类添加方法，以及如何创建标准化的方法列表提供给他人实现。

11.1 分类

有时候在面对一个类定义时，可能想要添加一些新方法。例如，对于 `Fraction` 类，除了将两个分数相加的 `add:` 方法之外，还想要将两个分数相减、相乘、相除的方法。

再举一个例子，假如你参与一个大型程序设计项目，作为项目的一部分，正在定义一个新类，它包含许多方法。你的任务就是为这个类编写处理文件系统的方法。其他项目成员负责创建和初始化类的实例、对类中的对象执行操作，以及在屏幕上绘制类对象的展现。

最后一个例子，假如你已经知道如何使用库中的类（例如，Foundation 框架的数组类，名为 `NSArray`），并且意识到你希望这个类能实现另一个或多个方法。当然，可以编写 `NSArray` 类新的子类并实现新方法，但是可能存在更简单的方式。

以上所有情况的实用解决方案就一个：分类。分类提供了一种简单的方式，用它可以将类的定义模块化到相关方法的组或分类中。它还提供了扩展现有类定义的简便方式，并且不必访问类的源代码，也无须创建子类。分类是一个功能强大且简单的概念。

回到第一个例子，展示如何为 `Fraction` 类添加新的分类，以处理基本的四则数学运算。首先，给出原始的 `Fraction` 接口部分：

```
#import <Foundation/Foundation.h>
// 定义 Fraction 类

@interface Fraction : NSObject

@property int numerator, denominator;

-(void) setTo: (int) n over: (int) d;
-(Fraction *) add: (Fraction *) f;
-(void) reduce;
-(double) convertToNum;
-(void) print;
@end
```

然后，从接口部分删除 `add:` 方法，并将其添加到新分类中，同时添加其他 3 种要实现的数学运算。新的 `MathOps` 分类的接口部分代码如下：

```
#import "Fraction.h"

@interface Fraction (MathOps)
-(Fraction *) add: (Fraction *) f;
-(Fraction *) mul: (Fraction *) f;
-(Fraction *) sub: (Fraction *) f;
-(Fraction *) div: (Fraction *) f;
@end
```

注意，这既是接口部分的定义，也是现有接口部分的扩展。因此，必须包括原始接口部分，这样编译器就知道 `Fraction` 类（除非直接将新分类结合到原始 `Fraction.h` 头文件，这是一种选择）。

在 `#import` 之后，有下面这一行：

```
@interface Fraction (MathOps)
```

告诉编译器你正在为 `Fraction` 类定义新的分类，而且它的名称为 `MathOps`。这个名称括在类名称之后的一对圆括号中。注意，此处没有列出 `Fraction` 的父类，因为编译器已从 `Fraction.h` 中知道这个内容。而且，你没有向编译器告知实例变量，因为以前定义的接口部分已经这样做了。实际上，如果尝试列出父类或实例变量，将会收到编译器发出的语法错误。

这个接口部分告知编译器，你正在为 `MathOps` 分类下名为 `Fraction` 的类添加扩展。`MathOps` 分类包括 4 个实例方法：`add:`、`mul:`、`sub:` 和 `div`。每个方法均使用一个分数作为参数，并返回一个分数。

可以将所有方法的定义放在一个实现部分。也就是说，可以在一个实现文件中定义 `Fraction.h` 接口部分中的所有方法，以及 `MathOps` 分类中的所有方法。或者，在单独的实现部分定义分类的方法。在这种情况下，这些方法的实现部分还必须找出方法所属的分类。与接口部分一样，通过将分类名称括在类名称之后的圆括号中来确定方法所属的分类，语句如下：

```
@implementation Fraction (MathOps)
    // 分类方法的代码
    ...
@end
```

在代码清单 11-1 中，新的 `MathOps` 分类的接口和实现部分组合在一起，连同测试函数都放在一个文件中。

代码清单 11-1 `MathOps` 分类和测试程序

```
#import "Fraction.h"

@interface Fraction (MathOps)
-(Fraction *) add: (Fraction *) f;
-(Fraction *) mul: (Fraction *) f;
-(Fraction *) sub: (Fraction *) f;
-(Fraction *) div: (Fraction *) f;
@end

@implementation Fraction (MathOps)
-(Fraction *) add: (Fraction *) f
{
    // 将两个分数相加：
    //  $a/b + c/d = ((a*d) + (b*c)) / (b * d)$ 

    Fraction *result = [[Fraction alloc] init];

    result.numerator = (numerator * f.denominator) +
        (denominator * f.numerator);
    result.denominator = denominator * f.denominator;
    [result reduce];

    return result;
}

-(Fraction *) sub: (Fraction *) f
{
    // 将两个分数相减：
    //  $a/b - c/d = ((a*d) - (b*c)) / (b * d)$ 
```



```

    Fraction *result = [[Fraction alloc] init];

    result.numerator = (numerator * f.denominator) -
        (denominator * f.numerator);
    result.denominator = denominator * f.denominator;
    [result reduce];

    return result;
}

-(Fraction *) mul: (Fraction *) f
{
    Fraction *result = [[Fraction alloc] init];

    result.numerator = numerator * f.numerator;
    result.denominator = denominator * f.denominator;
    [result reduce];

    return result;
}

-(Fraction *) div: (Fraction *) f
{
    Fraction *result = [[Fraction alloc] init];

    result.numerator = numerator * f.denominator;
    result.denominator = denominator * f.numerator;
    [result reduce];

    return result;
}
@end

int main (int argc, char *argv[])
{
    @autoreleasepool {
        Fraction *a = [[Fraction alloc] init];
        Fraction *b = [[Fraction alloc] init];
        Fraction *result;

        [a setTo: 1 over: 3];
        [b setTo: 2 over: 5];

        [a print]; NSLog(@" +"); [b print]; NSLog(@"-----");
        result = [a add: b];
        [result print];
        NSLog(@"\n");
    }
}

```

```

    [a print]; NSLog(@" -"); [b print]; NSLog(@"-----");
    result = [a sub: b];
    [result print];
    NSLog(@"\n");

    [a print]; NSLog(@" *"); [b print]; NSLog(@"-----");
    result = [a mul: b];
    [result print];
    NSLog(@"\n");

    [a print]; NSLog(@" /"); [b print]; NSLog(@"-----");
    result = [a div: b];
    [result print];
    NSLog(@"\n");
}
return 0;
}

```

代码清单 11-1 输出

```

1/3
+
2/5
-----
11/15

1/3
-
2/5
-----
-1/15

1/3
*
2/5
-----
2/15

1/3
/
2/5
-----
5/6

```

代码清单 11-1 把新分类的接口和实现部分与测试程序放在一个文件中。前面提到过，这个分类的接口部分可以放在原始的 `Fraction.h` 头文件中（这样，

所有的方法都在同一个位置声明)，也可以放在自己的头文件中。

注意

按照惯例，作为分类的.h和.m文件的基本名称是由类的名称紧接分类的名称。例如，我们把分类的接口部分的文件名命名为FractionMathOps.h，实现部分命名为FractionMathOps.m。一些程序员使用符号“+”来分隔类和分类的名字，比如Fraction+MathOps.h。

如果将分类放到一个主类定义文件中，那么这个类的所有用户都将访问这个分类中的方法。如果不能直接修改原始的头文件（考虑从一个库向现有类添加一个分类，如第二部分“Foundation 框架”中所示），除了单独保存它之外，别无选择。

11.2 类的扩展

有一种特殊的情况是创建一个未命名的分类，且在括号“()”之间不指定名字。这种特殊的语法定义称为类的扩展。定义一个像这样的未命名分类时，可以通过定义额外的实例变量和属性来扩展类，这在有命名的分类中是不允许的。未命名分类中声明的方法需要在主实现区域实现，而不是在分离的实现区域中实现。

未命名分类是非常有用的，因为它们的方法都是私有的。如果需要写一个类，而且数据和方法仅供这个类本身使用，未命名分类比较适合。

注意

方法并没有在接口部分登记，在这个意义上它们是私有的。然而，如果你知道私有方法的名字，那么也可以调用到它。

回到我们Fraction类中的reduce方法。假设你只想要从Fraction类的实现区域访问这个方法；也就是说，你不想让类的使用者直接访问这个方法，你可以把这个方法定义为私有，在Fraction.h头文件中移除它的声明，并在Fraction.m实现文件中编写如下代码：

假设有一个名为 `GraphicObject` 的类，而且 `GraphicObject.m` 的实现文件中有如下代码：

```
#import "Fraction.h"
// 使 reduce 方法私有化
@interface Fraction ()
- (void) reduce;
@end

// 为 Fraction 类定义方法
@implementation Fraction
@synthesize numerator, denominator;

- (void) setNumerator: (int) n andDenominator: (int) d
{
    ...
}
...
@end
```

11.2.1 关于分类的注意事项

分类可以覆写该类中的另一个方法，但是通常认为这种做法是拙劣的设计习惯。其一，覆写一个方法之后，再也不能访问原来的方法。因此，必须小心地将被覆写方法中的所有功能复制到替换方法中。如果确实需要覆写方法，正确的选择可能是创建子类。如果要在子类中覆写方法，仍然可以通过向 `super` 发送消息来引用父类的方法。因此，不必了解要被覆写方法的复杂内容，就能够调用父类的方法，并向子类的方法添加自己的功能。

如果喜欢，可以拥有许多分类，只要遵守此处指出的规则即可。如果一个方法定义在多个分类中，该语句不会指定使用哪个分类。

记住，通过使用分类添加新方法来扩展类不仅会影响这个类，同时也会影响它的所有子类。例如，如果为根对象 `NSObject` 添加新方法，就存在潜在的危险性，因为每个人都将继承这些新方法，无论你是否愿意。

通过分类为现有类添加新方法可能对你有用，但它们可能和该类的原始设计或意图不一致。例如，通过添加一个新分类和一些方法修改 `Square` 类的定义，将 `Square` 变成 `Circle`（确实有些夸张），并非好的编程习惯。

同样，对象/分类命名必须是唯一的。但是，在给定的 Objective-C 名称空间中，只能存在一个 `NSString Name Utilities` 分类。这样做可能比较复杂，因

为 Objective-C 名称空间是程序代码与所有的库、框架和插件共享的。对于编写屏幕保护首选窗格和其他插件的 Objective-C 程序员来说，这尤为重要，因为这些代码将插入到他们无法控制的应用程序或框架代码中。

11.3 协议和代理

协议是多个类共享的一个方法列表。协议中列出的方法没有相应的实现，计划由其他人来实现（比如你！）。协议提供了一种方式，用指定的名称定义一组多少有点相关的方法。这些方法通常有文档说明，所以你知道它们将如何执行。因此，如果需要，可在自己的类定义中实现它们。

协议列出了一组方法，有些可以是选择实现，有些是必须实现。如果决定实现特定协议的所有方法，也就意味着要遵守（confirm to）或者采用（adopt）这项协议。可以定义协议中的所有方法都是必须实现的，也可以都是选择实现的。

定义一个协议很简单：只要使用 `@protocol` 指令，后面跟上你给出的协议名称。然后，和处理接口部分一样，声明一些方法。`@end` 指令之前的所有方法声明都是协议的一部分。

如果选择使用 Foundation 框架，你将发现一些已定义的协议。其中一个名为 `NSCopying`，而且它声明了一个方法，如果你的类要支持使用 `copy`（或者 `copyWithZone:`）方法来复制对象，则必须实现这个方法。我们将在第 18 章“复制对象”中讲述复制对象的内容。

下面是在标准的 Foundation 头文件 `NSObject.h` 中定义 `NSCopying` 协议的方式：

`NSObject.h`:

```
@protocol NSCopying
- (id)copyWithZone: (NSZone *)zone;
@end
```

如果你的类采用 `NSCopying` 协议，则必须实现名为 `copyWithZone:` 的方法。通过在 `@interface` 行的一对尖括号（<...>）内列出协议名称，可以告知编译器你正在采用一个协议。这项协议的名称放在类名和它的父类名称之后，语句如下：

```
@interface AddressBook: NSObject <NSCopying>
```

这说明, `AddressBook` 是父类为 `NSObject` 的对象, 并且它遵守 `NSCopying` 协议。因为系统已经知道以前为这个协议定义的方法 (在这个例子中, 它是从 头文件 `NSObject.h` 中得知的), 所以不必在接口部分声明这些方法。但是, 要在实现部分定义它们。

因此, 在这个例子中, 在 `AddressBook` 的实现部分, 编译器期望找到定义的 `copyWithZone:` 方法。

如果你的类采用多项协议, 只需把它们都列在尖括号中, 并用逗号分开, 语句如下:

```
@interface AddressBook: NSObject <NSCopying, NSCodering>
```

以上语句告诉编译器, `AddressBook` 类采用 `NSCopying` 和 `NSCoding` 协议。这次, 编译器将期望在 `AddressBook` 的实现部分看到这些协议列出所有方法的实现。

如果你定义了自己的协议, 那么不必由自己实现它。但是, 这就告诉其他程序员, 如果要采用这项协议, 则必须实现这些方法。这些方法可以从超类继承。这样, 如果一个类遵守 `NSCopying` 协议, 则它的子类也遵守 `NSCopying` 协议 (不过这并不意味着对该子类而言, 这些方法得到了正确的实现)。

如果希望继承你的类的用户实现一些方法, 则可以使用协议定义这些方法。可以为你的 `GraphicObject` 类定义一个 `Drawing` 协议, 并且可以在其中定义 `paint`、`erase` 和 `outline` 方法, 代码如下:

```
@protocol Drawing
-(void) paint;
-(void) erase;
@optional
-(void) outline;
@end
```

作为 `GraphicObject` 类的创建者, 你不必实现这些绘制方法。但是, 需要指定一些方法, 要求从 `GraphicObject` 创建子类的人实现这些方法, 以便符合他要创建的绘图对象的标准。

注意, 这里使用了 `@optional` 指令。该指令之后列出的所有方法都是可选的。也就是说, 采用 `Drawing` 方法不一定要实现 `outline` 方法来遵守该协议 (之后可以通过在协议定义内使用 `@required` 指令来列出需要的方法)。

注意

无论如何，这是理论。编译器允许你声明遵守一项协议，并且只有当你没有实现这些方法时，才发出警告消息。

因此，如果创建名为 `Rectangle` 的 `GraphicObject` 的子类，并宣称（也就是文档说明）这个 `Rectangle` 类遵守 `Drawing` 协议，那么这个类的用户就知道他们可以向这个类的实例发送 `paint`、`erase` 和 `outline`（可能有）消息。

注意，协议不引用任何类，它是无类的（`classless`）。任何类都可以遵守 `Drawing` 协议，不仅仅是 `GraphicObject` 的子类。

可以使用 `conformsToProtocol:` 方法检查一个对象是否遵循某项协议。例如，如果有一个名为 `currentObject` 的对象，并且想要查看它是否遵循 `Drawing` 协议，可以向它发送绘图消息，代码如下：

```
id currentObject;
...
if ([currentObject conformsToProtocol: @protocol (Drawing)] == YES)
{
    // 给 currentObject 发送 paint、erase 或者 outline 消息
    ...
}
```

这里使用的专用 `@protocol` 指令用于获取一个协议名称，并产生一个 `Protocol` 对象，`conformsToProtocol:` 方法期望这个对象作为它的参数。

为了测试 `currentObject` 是否实现了可选的 `outline` 方法，可以编写下列代码：

```
if ([currentObject respondsToSelector: @selector (outline)] == YES)
    [currentObject outline];
```

通过在类型名称之后的尖括号中添加协议名称，借助编译器来检查变量的一致性，语句如下：

```
id <Drawing> currentObject;
```

告知编译器 `currentObject` 将包含遵守 `Drawing` 协议的对象。如果向 `currentObject` 指派静态类型的对象，这个对象不遵守 `Drawing` 协议（假定有一个 `Square` 对象，它不遵守 `Drawing` 协议），编译器将发出一条警告消息，语句如下：

```
warning: class 'Square' does not implement the 'Drawing' protocol
```

这里存在一项编译器校验, 所以向 `currentObject` 指派一个 `id` 变量不会产生这条信息, 因为编译器无法知道存储在 `id` 变量中的对象是否遵守 `Drawing` 协议。

如果这个变量保存的对象遵守多项协议, 则可以列出多项协议, 如以下代码:

```
id <NSCopying, NSCoder> myDocument;
```

定义一项协议时, 可以扩展现有协议的定义。所以, 以下协议定义

```
@protocol Drawing3D <Drawing>
```

说明 `Drawing3D` 协议也采用了 `Drawing` 协议。因此, 任何采用 `Drawing3D` 协议的类都必须实现此协议列出的方法, 以及 `Drawing` 协议的方法。

最后, 分类也可以采用一项协议, 语句如下:

```
@interface Fraction (Stuff) <NSCopying, NSCoder>
```

此处, `Fraction` 拥有一个分类 `Stuff` (当然, 并非最佳的名称), 这个分类采用了 `NSCopying` 和 `NSCoding` 协议。

如果你不想在公开的 `.h` 文件中声明你的类符合特定的协议, 你可以将协议象分类扩展一样写在你的 `.m` 文件中。

```
@interface ProfilePreviewController () <NavigationBarAdjustments>
...
@end
```

和类名一样, 协议名必须是唯一的。

11.3.1 代理

协议也是一种两个类之间的接口定义。定义了协议的类可以看作是将协议定义的方法代理给了实现它们的类。这样, 类的定义可以更为通用, 因为具体的动作由代理类来承担, 响应某些事件或者定义某些参数。Cocoa 和 iOS 非常依赖代理这个概念。例如, 当你在 iPhone 上建立一个表格时, 会使用到 `UITableView` 类。但是这个类不清楚表格的标题是什么, 需要包含多少个区块或是包含多少行, 填充表格 (单元行) 的内容是什么。所以, 代理为你定义了一个 `UITableViewDataSource` 协议。如果它需要信息 (比如表格中的每个区块有多少行), 它就会调用类中实现协议的相关方法 (`tableView:numberOfRowsInSection:`), 你必须写在你的类中, 与协议相匹配。 `UITableView`

类还定义了其他的协议，如 `UITableViewDelegate`。协议中还定义了一些方法，如表格中某些行被选中需要怎么样（`tableView: didSelectRowAtIndexPath:`）。`UITableView` 类并不知道还要做些什么事情，所以将这个责任代理给你。

11.3.2 非正式协议

你可能遇到过非正式（informal）协议的概念。它实际上是一个分类，列出了一组方法但并没有实现它们。每个人（或者几乎每个人）都继承相同的根对象，因此，非正式分类通常是根类定义的。有时，非正式协议也称为抽象（abstract）协议。

如果查看头文件 `<NSScriptWhoseTests.h>`，可能会发现如下一些方法声明：

```
@interface NSObject (NSComparisonMethods)
- (BOOL)isEqualTo:(id) object;
- (BOOL)isLessThanOrEqualTo:(id) object;
- (BOOL)isLessThan:(id) object;
- (BOOL)isGreaterThanOrEqualTo:(id) object;
- (BOOL)isGreaterThan:(id) object;
- (BOOL)isNotEqualTo:(id) object;
- (BOOL)doesContain:(id) object;
- (BOOL)isLike: (NSString *) object;
- (BOOL)isCaseInsensitiveLike: (NSString *) object;
@end
```

这些代码为 `NSObject` 类定义了一个名为 `NSComparisonMethods` 的分类。这项非正式协议列出了一组方法（这里列出了 9 个），可以将它们实现为协议的一部分。非正式协议实际上仅仅是一个名称之下的一组方法。在文档说明和模块化方法时，可能有所帮助。

声明非正式协议的类自己并不实现这些方法，并且选择实现这些方法的子类需要在它的接口部分重新声明这些方法，同时还要实现这些方法中的一个或多个。和正式协议不同，编译器不提供有关非正式协议的帮助，这里没有遵守协议或者由编译器测试这样的概念。

如果一个对象采用正式协议，则它必须遵守协议中的所有信息。这可以在运行及编译时强制执行。如果一个对象采用非正式协议，则它可能不需要采用此协议的所有方法，具体取决于这项协议。可以在运行时强制要求遵守一项非正式协议（借助 `respondsToSelector:`），但是在编译时不可以。

注意

前面描述的@optional 指令添加到了 Objective-C 2.0 语言中，用于取代非正式协议的使用。你可以看到几个 UIKit 类（UIKit 是 Cocoa Touch 框架的一部分）是这样用的。

11.4 合成对象

你已经学习了通过派生子类 and 分类等扩展类定义的几种方式。还有一项技术可以定义一个类包含其他类的一个或多个对象。这个新类的对象就是所谓的合成（composite）对象，因为它是由其他对象组成的。

举个例子，考虑第 8 章中定义的 Square 类。将这个类定义为 Rectangle 的子类，因为你知道正方形就是等边的矩形。定义子类时，它继承了父类的所有实例变量和方法。在一些情况下，这种做法不合适。例如，在父类中定义的一些方法可能不适合子类使用。Square 类继承了 Rectangle 的 setwidth:andHeight: 方法，但并不适用（即使它能够正常工作）。此外，创建子类时，必须确保所有被继承的方法能够正常工作，因为这个类的用户可能会访问它们。最终，子类依赖于父类，改变父类有可能会使得子类中的方法不能工作。

作为创建子类的替代方式，可以定义一个新类，它包含要扩展类的实例变量。然后，只需在新类中定义适合该方法的方法。返回 Square 的例子，下面是 Square 另一种方式的定义：

```
@interface Square: NSObject
{
    Rectangle *rect;
}
-(int) setSide: (int) s;
-(int) side;
-(int) area;
-(int) perimeter;
@end
```

此处定义的 Square 类有 4 个方法。和子类版本不同，子类版本允许你直接访问 Rectangle 的方法（setWidth:、setHeight:、setWidth:andHeight:、width 和 height），但是这个 Square 的定义中不包括这些方法。这样做很有意义，因为处理 Square 时，这些方法确实不适合。

如果以这种方式定义 `Square`，就需要为它包含的矩形分配存储空间。例如，如果不覆写方法，则以下语句

```
Square *mySquare = [[Square alloc] init];
```

分配了一个新的 `Square` 对象，但是没有为存储在实例变量中的 `Rectangle` 对象 `rect` 分配存储空间。

一个解决方案就是覆写 `init` 或添加 `initWithSide:` 之类的新方法来分配空间。这个方法可以为 `Rectangle rect` 分配存储空间，并相应地设置它的边。

在 `Square` 类中定义方法时，可以使用 `Rectangle` 的方法。例如，下面说明了如何实现 `area` 方法：

```
-(int) area
{
    return [rect area];
}
```

其他方法的实现作为练习留给大家（参见本章练习 5）。

11.5 练习

1. 扩展代码清单 11-1 中的 `MathOps` 分类，使之包含一个 `invert` 方法，这个方法返回一个 `Fraction`，它是接收者的倒置。
2. 向类 `Fraction` 添加一个名为 `Comparison` 的分类。根据以下声明，在这个分类中添加两个方法：

```
-(BOOL) isEqualTo: (Fraction *) f;
-(int) compare: (Fraction *) f;
```

如果两个分数相同，第一个方法应该返回 YES；否则，返回 NO。注意分数的比较方式（如，比较 $3/4$ 和 $6/8$ 应当返回 YES）。

如果接收者小于参数传递来的分数，则第二个方法应当返回 -1；如果二者相等，应返回 0；如果接收者大于参数，则应当返回 1。

3. 通过添加遵守非正式协议 `NSCompairsonMethods`（本章前面所列出的）的方法来扩展 `Fraction` 类。根据该协议实现前 6 个方法（`isEqualTo:`、`isLessThanOrEqualTo:`、`isLessThan:`、`isGreaterThanEqualTo:`、`isGreaterThan:`、`isNotEqualTo:`），并测试它们。

4. 函数 `sin()`、`cos()` 和 `tan()` 是 C 标准库的一部分（与 `scanf()` 一样）。这些函数在系统头文件 `<math.h>` 中声明了，当导入 `Foundation.h` 时，这些就会被自动导入到你的程序中。

这些函数分别可以用来计算用弧度表示的 `double` 参数的 `sine`、`cosine` 或者 `tangent` 值返回的结果，也是一个双精度的浮点值。所以，

```
result = sin (d);
```

用于计算 `d` 的 `sine` 值，角度 `d` 的值用弧度表示。为第 6 章“选择结构”中的 `Calculator` 类添加一个名为 `Trig` 的分类。根据以下声明，为这个分类添加一些方法来计算 `sine`、`cosine` 和 `tangent` 的值：

```
-(double) sin;
-(double) cos;
-(double) tan;
```

5. 根据本章对合成对象的讨论以及以下接口部分：

```
@interface Square: NSObject
-(instancetype *) initWithSide: (int) s;
-(void) setSide: (int) s;
-(int) side;
-(int) area;
-(int) perimeter;
@end
#import "Rectangle.h"
@implementation Square
{
    Rectangle *rect;
}
// 这里插入 Square 的方法
...
@end
```

编写 `Square` 的实现部分，以及用来检验其方法的测试程序。注意：记住需要覆写 `init` 方法，`initWithSide:` 将作为重新设计过的初始化方法。

12

预处理程序

预处理程序提供了一些工具，使用这些工具更易于开发、阅读、修改程序，也易于将程序移植到不同的系统中。你也可以使用预处理程序定制 Objective-C 语言，以适应特定应用的编程或自己的编程风格。

预处理程序是 Objective-C 编译过程的一部分，它可以识别散布在程序中的特定语句。顾名思义，预处理程序实际上是在分析 Objective-C 程序之前先处理这些语句。预处理程序语句使用井号（#）标记，这个符号必须是一行中的第一个非空格字符。你会看到，预处理程序语句的语法与 Objective-C 语句略微不同。下面先从 `#define` 语句开始介绍。

12.1 `#define` 语句

`#define` 语句的基本用途之一就是给符号名称指定程序常量。预处理程序语句

```
#define TRUE 1
```

定义了名称 `TRUE`，并使它等于值 `1`。之后，名称 `TRUE` 可用于程序中任何需要常量 `1` 的地方。只要出现这个名称，预处理程序自动在程序中将这个名称替换为预定义的值 `1`。例如，可能遇到下面这条 Objective-C 语句，语句使用了预定义的名称 `TRUE`：

```
gameOver = TRUE;
```

这条语句向 `gameOver` 指定了值 `TRUE`。你无须关心 `TRUE` 定义的确切值。但是因为已经知道它定义为 `1`，所以前面语句的作用就是将 `1` 赋给 `gameOver`。

预处理程序语句

```
#define FALSE 0
```

定义了名称 **FALSE**，随后在程序中它就等价于 0。因此，语句

```
gameOver = FALSE;
```

将 **FALSE** 的值赋给 **gameOver**，而语句

```
if ( gameOver == FALSE )
...
```

比较 **gameOver** 的值和 **FALSE** 的预定义值。

预定义名称不是变量。因此，不能为它赋值，除非替换指定值的结果实际上是一个变量。只要在程序中使用预定义名称，**#define** 语句中预定义名称右边的所有字符都会被预处理程序自动替换到程序中。这类似于在文本编辑器中进行搜索和替换，这种情况下，预处理程序会将出现的所有预定义名称替换为相应的文本。

你会发现**#define** 语句的语法很特别：将 **TRUE** 赋值为 1 没有用到等号。而且，在语句末尾也没有出现分号。不过你马上就会明白这种特殊语法存在的原因。

#define 语句经常放在程序的开始，但**#import** 或 **include** 语句之后。这并不是必需的，它们可以出现在程序的任何地方。但是，在程序引用这个名称之前，必须先定义它们。预定义的名称和变量的行为方式不同：没有局部定义之类的说法。在定义一个名称之后，就可以在程序的任何地方使用它。大多数程序员把定义放在头文件中，以便在多个源文件中使用它们。

举另一个使用预定义名称的例子。假设想要编写两个方法来计算 **Circle** 对象的面积和周长。这两个方法都需要使用常量 π ，但是它不容易记住，因此，合理的情况是，在程序的开始部分定义常量的值。然后根据需要在每个方法中使用这个值。

所以，可以在程序中包含以下代码：

```
#define PI 3.141592654
```

然后就可以像下面这样在两个 **Circle** 方法中使用它（下面假设 **Circle** 类中有一个名为 **radius** 的实例变量）：

```
-(double) area
{
```

```

    return PI * radius * radius;
}

-(double) circumference
{
    return 2.0 * PI * radius;
}

```

（注意：你已经预定义了一个字符为 `PI`，就可以在需要的地方直接使用 π 的值了）给符号名称指定一个常量，每次想在程序中使用它们时，就不必记住这个特定常量的值。此外，如果需要更改常量的值（例如，你发现使用了错误的值），只需要在程序的一个地方更改这个值，那就是在 `#define` 语句中。如果没有这种方式，将不得不从头到尾搜索程序，并在使用值的地方显式修改这个常量的值。

你可能已经注意到了，目前为止显示的所有定义（`TRUE`、`FALSE` 和 `PI`）都是大写字母组合。这是为了从视觉上区分预定义的值和变量。一些程序员有这样的习惯：所有预定义的名称都用大写，这样容易区分一个名称是变量名、对象名、类名，还是预定义名称。另一种常见的惯例是在定义之前加字母 `k`。这种情况下，`k` 之后的字符并不用全部大写。例如，`kMaximum Values` 和 `kSignificantDigits` 是符合惯例的两个预定义名称。

对常量值使用预定义名称有助于加强程序的可扩展性。例如，学习如何使用数组可以不通过硬编码分配数组大小，而是使用如下定义：

```
#define MAXIMUM_DATA_VALUES 1000
```

这样，所有的引用都可以以这个数组的大小为基础（如在内存中分配数组的内存），并且根据这个预定义的值确定数组的有效下标。

另外，假设程序在任何用到数组大小的地方都使用 `MAXIMUM_DATA_VALUES`，如果后来需要改变数组的大小，程序中唯一必须改动的语句就是前面的定义。

12.1.1 更高级的定义类型

名称的定义不仅可以包括简单的常量值，你很快就会看到，它也可以包括表达式和其他任何内容。

下面的语句是将名称 `TWO_PI` 定义为 2.0 与 3.141592654 的积：


```
#define TWO_PI 2.0 * 3.141592654
```

随后就可以在程序中表达式 2.0×3.141592654 有效的任何地方，使用这个预定义名称。因此，可以使用以下语句替换前面例子中 `circumference` 方法的 `return` 语句：

```
return TWO_PI * radius;
```

在 Objective-C 程序中遇到预定义的名称时，使用 `#define` 语句中预定义名称右边的所有字符字面替换程序中点的名称。因此，当预处理程序遇到前面所示的 `return` 语句中的名称 `TWO_PI` 时，它使用 `#define` 语句中相应名称的全部字符替换这个名称。因此，只要程序中出现 `TWO_PI`，预处理程序就将其字面替换为 2.0×3.141592654 。

事实上，预定义名称一出现，预处理程序就执行文本替换，这可以解释为什么通常不能使用分号结束 `#define` 语句的原因。如果使用了分号，只要出现预定义名称，分号也将替换到程序中。如果如下定义 `PI`：

```
#define PI 3.141592654;
```

然后这样编写代码：

```
return 2.0 * PI * r;
```

那么预处理程序将使用 3.141592654 替换预定义名称 `PI`。因此，在预处理程序完成替换之后，编译器将把这条语句看作：

```
return 2.0 * 3.141592654; * r;
```

这会导致语法错误。记住，除非十分确定需要分号，否则不要在定义语句的末尾添加分号。

预处理程序定义的右面不必是合法的 Objective-C 表达式，只要在使用它的时候，结果表达式正确就可以了。例如，可以如下设置定义：

```
#define AND &&
#define OR ||
```

然后可以如下编写表达式：

```
if ( x > 0 AND x < 10 )
    ...
```

和

```
if ( y == 0 OR y == value )
...
```

甚至可以包含**#define**，用于测试相等性：

```
#define EQUALS ==
```

然后，可以编写如下表达式：

```
if ( y EQUALS 0 OR y EQUALS value )
...
```

这样就消除了使用单个等号“=”错误进行等价判断的可能性。

虽然这些例子显示了**#define**的强大功能，但是应该注意，以这种方式重新定义底层语言语法的行为通常是不好的编程习惯，而且会使其他人难以理解你的代码。

更有趣的是，预定义的值本身可以引用另一个预定义的值。所以，以下两个定义：

```
#define PI 3.141592654
#define TWO_PI 2.0 * PI
```

是完全合法的。名称 **TWO_PI** 是按照前面的预定义名称 **PI** 定义的，这样就不必重复拼写值 3.141592654。

如果把这两个定义的顺序颠倒一下，即语句

```
#define TWO_PI 2.0 * PI
#define PI 3.141592654
```

也是合法的。规则就是，在程序中使用预定义名称时，只要所有的符号都是定义过的，那么就可以在定义中引用其他预定义的值。

合理使用**#define**通常可以减少程序中对注释的需要。考虑如下语句：

```
if ( year % 4 == 0 && year % 100 != 0 || year % 400 == 0 )
...
```

这个表达式检测变量 **year** 是不是闰年。现在，考虑以下定义及后续的 **if** 语句：

```
#define IS_LEAP_YEAR year % 4 == 0 && year % 100 != 0 \
|| year % 400 == 0

...
if ( IS_LEAP_YEAR )
...
```

通常，预处理程序假定义包含在程序的一行中。如果需要第二行，那么上一行的最后一个字符必须是反斜线符号。这个字符告诉预处理程序这里存在一个后续，否则将被忽略。对于多个后续行，也是如此，每个要继续的行都必须以反斜线结尾。

这条 if 语句远比前面的 if 语句更容易理解。因为语句很清楚，所以无须注释。当然，这个定义只限于测试判断变量 `year` 是不是闰年。最好能够编写一个定义，它能够判定任何一年是否是闰年，而不只是变量 `year`。实际上，可以编写带有一个或多个自变量的定义。这就引出下一个讨论要点。

可以将 `IS_LEAP_YEAR` 定义为带有一个名 `y` 的参数：

```
#define IS_LEAP_YEAR(y) y % 4 == 0 && y % 100 != 0 \
    || y % 400 == 0
```

和方法定义不同，这里没有定义参数 `y` 的类型，因为此时仅执行字面文本替换，并没有调用函数。注意，在定义带有参数的名称时，预定义名称和参数列表的左括号之间不允许空格。

依照前面的定义，可以编写如下语句：

```
if ( IS_LEAP_YEAR (year) )
    ...
```

这个语句判断 `year` 的值是不是闰年。或者，可以编写如下语句来测试 `nextYear` 的值是不是闰年：

```
if ( IS_LEAP_YEAR (nextYear) )
    ...
```

在前面的语句中，`IS_LEAP_YEAR` 的定义直接替换到 if 语句中，同时只要定义中出现 `y`，就使用参数 `nextYear` 替换它。这样，编译器实际上将这个 if 语句看作：

```
if ( nextYear % 4 == 0 && nextYear % 100 != 0 || nextYear % 400 == 0 )
    ...
```

这种预定义通常称为“宏”。这个术语经常用于带有一个或多个参数的预定义。

下面这个宏名为 `SQUARE`，它简单地将参数乘方：

```
#define SQUARE(x) x * x
```

虽然 `SQUARE` 的宏定义简单明了，但是在定义宏时有一个有趣的陷阱，必须小心避开。我们描述过，语句

```
y = SQUARE (v);
```

把 v^2 的值赋给 `y`。你认为对于以下语句会发生什么情况？

```
y = SQUARE (v + 1);
```

这个语句并不像你期望的那样，把 $(v+1)^2$ 的值赋给 `y`。因为预处理程序对宏定义的参数实行文本替换，前面的表达式实际上是如下求值的：

```
y = v + 1 * v + 1;
```

这显然不能得到期望的结果。要正确解决这个问题，需要在 `SQUARE` 宏的定义中加入括号：

```
#define SQUARE(x) ( (x) * (x) )
```

虽然这个定义看起来可能有点奇怪，但要记住定义中任何出现 `x` 的地方都要使用整个表达式进行字面替换，和 `SQUARE` 宏定义的一样。使用新的 `SQUARE` 宏定义，语句

```
y = SQUARE (v + 1);
```

将正确地作为以下表达式进行求值：

```
y = ( (v + 1) * (v + 1) );
```

以下的宏允许你方便地根据 `Fraction` 类动态地创建新分数：

```
#define MakeFract(x,y) ([[Fraction alloc] initWith: x over: y])
```

然后，可以编写如下表达式：

```
myFract = MakeFract (1, 3); //创建分数 1/3
```

定义宏时，使用条件表达式的运算符可以非常方便。以下语句定义了一个名为 `MAX` 的宏，它给出两个值的最大值：

```
#define MAX(a,b) ( ((a) > (b)) ? (a) : (b) )
```

这个宏允许以后写出如下语句：

```
limit = MAX (x + y, minValue);
```

这个式子把 `x+y` 和 `minValue` 的最大值赋给 `limit`。用括号把整个 `MAX` 定义括起来是为了确保正确地计算如下表达式：

```
MAX (x, y) * 100
```

每个自变量都用括号括起来是为了确保正确地计算如下表达式：

```
MAX (x & y, z)
```

&运算符是按位 AND 运算符，它的优先级低于宏中使用的>运算符。如果宏定义中没有括号，>运算符将在按位 AND 之前求值，从而导致错误的结果。

以下宏语句测试字符是不是小写字母：

```
#define IS_LOWER_CASE(x) ( ((x) >= 'a') && ((x) <= 'z') )
```

因此，允许编写以下表达式：

```
if ( IS_LOWER_CASE (c) )
    ...
```

甚至可以在另一个宏定义中使用这个宏把字符从小写转换为大写，同时不改变非小写字符：

```
#define TO_UPPER(x) ( IS_LOWER_CASE (x) ? (x) - 'a' + 'A' : (x) )
```

这里再次用到标准 ASCII 字符集。在第二部分学习 Foundation 字符串对象时，将看到如何对国际（Unicode）字符集执行大小写转换。

12.2 #import 语句

经过一段时间的 Objective-C 编程后，你发现自己正开发一组宏，并且想在每一个程序中使用它们。但是不必在编写的每个新程序中输入这些宏，相反，预处理程序允许你将所有的定义收集到一个单独文件中，然后使用#import 语句把它们包含在程序中。这些文件类似于前面遇到的但是不必是你自己编写的程序，它们通常以 h 结尾，人们将其称为头文件或包含文件。

假设你正在编写一系列程序，用于执行各种量度转换。你可能想为执行转换所需要的各种常量设置一些#define 语句：

```
#define INCHES_PER_CENTIMETER 0.394
#define CENTIMETERS_PER_INCH (1 / INCHES_PER_CENTIMETER)

#define QUARTS_PER_LITER 1.057
#define LITERS_PER_QUART (1 / QUARTS_PER_LITER)

#define OUNCES_PER_GRAM 0.035
#define GRAMS_PER_OUNCE (1 / OUNCES_PER_GRAM)
```

...

假设将前面的定义输入到系统中一个名为 `metric.h` 的独立文件中。随后任何需要使用包含在 `metric.h` 中定义的程序都只需简单地使用以下预处理程序指令：

```
#import "metric.h"
```

在引用 `metric.h` 中的定义之前，必须出现这条语句，并且通常放在源文件的开始处。预处理程序在系统中寻找指定的文件，并且有效地把文件的内容复制到程序中出现 `#import` 语句的确切位置。这样，文件中的所有语句似乎都是直接在程序中这个位置插入的。

头文件名两侧的双引号指示预处理程序在一个或者多个文件目录（通常，首先在包含源文件的目录中查找，但是通过修改适当的“项目设置”，可以用 Xcode 指定预处理程序搜索的确切位置）中寻找指定的文件。

把文件名放在“<”和“>”字符之间，例如

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

将导致预处理程序只在特殊的“系统”头文件目录中寻找包含文件，当前目录不会被搜索。同样，使用 Xcode 可以通过指定一些需要搜索的目录。

导入文件最出色的能力之一是它使你能够把定义集中起来，从而确保所有的程序引用相同的值。此外，如果包含文件中发现任何错误值，只需在这个位置修改，不必更正每个使用该值的程序。任何引用这个错误值的程序只需简单地重新编译一下，而不必重新编辑。

在 Xcode 5 中，你可能会看到这样一个指令，它看起来像这样：

```
#import UIKit;
```

这是被称为模块（module）的功能，它的功能是为系统头文件提供更快的处理（重编译它们）速度和避免了潜在的命名冲突。

12.3 条件编译

Objective-C 预处理程序提供了一项名为“条件编译”的功能。条件编译通常用于创建可以在不同计算机系统上编译运行的程序。它还经常用来开关程序中的各种语句，例如，用来输出变量值或跟踪程序执行流程的调试语句。

12.3.1 #ifdef、#endif、#else 和#ifndef 语句

遗憾的是，程序有时必须依靠系统相关的参数，这些参数需要在不同的设备（例如，iPhone 与 iPad）或特定版本的操作系统（例如，Leopard 与 Snow Leopard）上分别指定。

如果有大型程序，它对计算机系统的特定硬件和 / 或软件有很多这样的依赖（应尽可能减少这种依赖性），最终你可能会有许多这样的定义，在程序移植到其他计算机系统时，它们的值不得不更改。

通过利用预处理程序的条件编译能力，能够减少对这些值的改变，并且能够把每种机器关于这些定义的值结合到程序中。举个简单的例子，如果前面已经定义了符号 IPAD，下面的语句

```
#ifdef IPAD
# define kImageFile @"barnHD.png"
#else
# define kImageFile @"barn.png"
#endif
```

就把 kImageFile 定义为@"barnHD.png"，否则就定义为@"barn.png"。

这里可以看到，允许在标志预处理语句开始的#符号之后放置一个或多个空格。

#ifdef、#else 和#endif 语句的行为和你期望的一样。如果程序编译时，#ifdef 行中所指定的符号已经通过#define 语句或命令行定义了，那么编译器将处理从此处开始到#else、#elif 或#endif 的程序段，否则就忽略这个程序段。

要为预处理程序定义符号 IPAD，使用如下语句：

```
#define IPAD 1
```

或者仅仅

```
#define IPAD
```

就足够了。正如你所见，定义的名称之后没有必要出现文本来满足#ifdef 检测。编译器还允许使用编译器命令的特殊选项在程序编译时为预编译器定义名称。

命令行

```
clang -fobjc -arc -D IPAD program.m -
```

为预处理程序定义了名称 IPAD，它使 program.m 中的所有#ifdef IPAD 语句都

判断为 TRUE（注意，在命令行中，`-D IPAD` 必须在程序名称之前输入）。这种技术使得不必编辑源程序就可以定义名称。

使用 Xcode，通过 Build Settings 可以添加新的预定义名称并指定它们的值，Apple LLVM x.x -预处理处理宏。图 12.1 表示在 Xcode 4 中定义了一个 IPAD 字符和 DEBUG（定义为 1）字符。

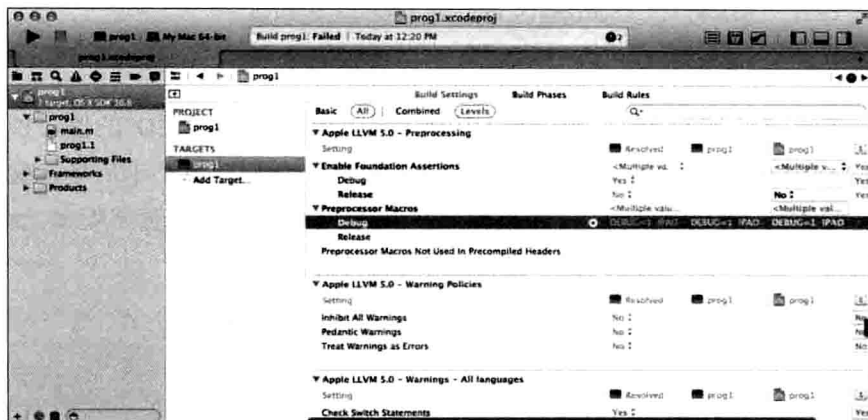


图 12.1 定义 IPAD 预处理标识符

`#ifndef` 语句与 `#ifdef` 在一条线上。这个语句的使用方式类似，只是如果指定的符号没有定义，它将导致程序处理后续行。

前面提到过，在调试程序时，条件编译很有用。你可能在程序中嵌入了很多 `NSLog` 调用，用于显示中间结果并跟踪执行流程。如果定义了一个特定名称（如 `DEBUG`），通过将这些语句条件编译到程序中，就可以打开它们。例如，只要程序在编译时定义了名称 `DEBUG`，就可以使用如下一系列语句来显示一些变量的值：

```
#ifdef DEBUG
    NSLog(@"User name = %@, id = %i", userName, userId);
#endif
```

程序中可能有很多这样的调试语句。无论何时调试这个程序，都能够通过 `DEBUG` 使所有的调试语句都编译。当程序已经能正确工作时，就可以不加 `DEBUG` 重新进行编译。这样做还可以削减程序大小，因为没有编译所有的调试语句。

12.3.2 #if 和#elif 预处理程序语句

#if 预处理程序语句为控制条件编译提供了更通用的方法。if 语句可以用来检测常量表达式是否为非零。如果表达式的结果为非零，就会处理到#else、#elif 或#endif 为止的所有后续行；否则将跳过它们。

举一个如何使用它的例子，Foundation 头文件 NSString.h 中有以下行：

```
#if MAC_OS_X_VERSION_MIN_REQUIRED < MAC_OS_X_VERSION_10_5
#define NSMaximumStringLength (INT_MAX-1)
#endif
```

这将预定义变量 MAC_OS_X_VERSION_MIN_REQUIRED 的值与预定义变量 MAC_OS_X_VERSION_10_5 的值进行比较。如果前者小于后者，就处理随后的#define，否则就跳过它。如果程序在 MAC OS X 10.5 或更高版本上编译，这大概将一个字符串的最大长度设置为整型的最大值减 1。

特殊运算符

```
defined (name)
```

也能够用在#if 语句中。预处理程序语句集

```
#if defined (DEBUG)
...
#endif
```

和

```
#ifdef DEBUG
...
#endif
```

的作用相同。

以下语句出现在 NSObjcRuntime.h 头文件中，用于根据使用的特定编译器定义 NS_INLINE（如果之前未定义）：

```
#if !defined(NS_INLINE)
  #if defined(__GNUC__)
    #define NS_INLINE static __inline_attribute__((always_inline))
  #elif defined(__MWERKS__) || defined(__cplusplus)
    #define NS_INLINE static inline
  #elif defined(_MSC_VER)
    #define NS_INLINE static __inline
  #elif defined(__WIN32__)
    #define NS_INLINE static __inline__
  #endif
#endif
```

```
#endif
#endif
```

另一种常见的用法是在如下代码序列中：

```
#if defined (DEBUG) && DEBUG
...
#endif
```

这使得 `#if` 到 `#endif` 之间的语句只有在定义了 `DEBUG` 而且具有非零值时才被处理。

因为可以使用表达式且因为 0 代表 `false`，程序员（包括笔者自己）经常使用 `#if 0 ... #endif` 这样一对预处理语句包围需要注释的代码块。

12.3.3 #undef 语句

在一些情况下，可能需要使一些已经定义的名称成为未定义的，通过使用 `#undef` 语句就可以这么做。要消除特定名称的定义，编写如下语句：

```
#undef name
```

这样，语句

```
#undef IPAD
```

将消除 `IPAD` 的定义。之后的 `#ifdef IPAD` 或 `#if defined(IPAD)` 语句都将判断为 `FALSE`。这里总结了关于预处理程序的讨论。

12.4 练习

1. 在机器上找到系统头文件 `limits.h` 和 `float.h`。检查这些文件，看看其内容。如果这些文件包含其他头文件，确保跟踪它们并查看它们的内容。
2. 定义一个名为 `MIN` 的宏，它给出两个值的最小值。然后编写程序来测试这个宏定义。
3. 定义一个名为 `MAX3` 的宏，它给出 3 个值的最大值。然后编写一个程序来测试这个宏定义。
4. 编写一个名为 `IS_UPPER_CASE` 的宏，其作用是如果字符是大写字母，就给出非零值。
5. 编写一个名为 `IS_ALPHABETIC` 的宏，其作用是如果一个字符是字母，

就给出非零值。使用本章定义的 `IS_LOWER_CASE` 宏和本章练习 4 中定义的 `IS_UPPER_CASE` 宏。

6. 编写一个名为 `IS_DIGIT` 的宏，其作用是如果字符是 0~9 的数字，就给出非零值。在另一个名为 `IS_SPECIAL` 的宏定义中使用它。如果字符是一个特殊字符（也就是说，它既不是字母，也不是数字），`IS_SPECIAL` 将给出非零结果。一定要使用练习 5 中定义的 `IS_ALPHABETIC` 宏。
7. 编写一个名为 `ABSOLUTE_VALUE` 的宏，其作用是计算参数的绝对值。确保能够正确计算 `ABSOLUTE_VALUE (x+delta)` 之类的表达式的值。

```
ABSOLUTE_VALUE (x + delta)
```

基本的 C 语言特性

本章要介绍的一些特性，在编写 Objective-C 程序时不一定必须了解。事实上，这些特性大部分来源于基本的 C 语言。函数、结构、指针、联合和数组之类的特性可以在需要了解的时候再学习。因为 C 语言是一门过程式语言，其中一些特性与面向对象编程的思想是对立的。这些特性也会妨碍 Foundation 框架的实现策略，比如分配内存的方式，或者处理包含多字节字符的字符串。

注意

Objective-C 有多种使用多字节字符的方式，但是 Foundation 通过 NSString 类提供了比较好的解决方案。

然而，有一些应用程序为了优化要求使用底层方法。比如，使用大型的数据数组，可能会使用 C 语言的内置数据结构，而不是 Foundation 的数组对象（详见第 15 章）。如果使用恰当，函数也可以方便地进行组合重复使用，并将程序模块化。

建议你先浏览本章，在阅读完第二部分之后再学习本章的内容。或者完全忽略本章，开始学习第二部分。如果将来需要维护其他人编写的代码，开始研究 Foundation 框架的头文件，你将会接触到本章所讲的一些结构。很多 Foundation 数据类型（如 NSRange、NSPoint 和 NSRect）都要求对本章所讲的结构有一些基本了解。在这些情况下，可以回到本章，阅读相关部分以了解所需的概念。

13.1 数组

Objective-C 提供了一项功能，它允许用户定义一组有序的数据项，即数组。本节将讲述如何定义和操作数组。后面的内容，我们会进一步讨论数组，阐述它们如何与函数、结构、字符串及指针一起使用。

假设你有一组成绩要录入计算机，并要对这些成绩执行一些操作，比如按升序排列、计算平均值或查找中值。在排序过程中，除非录入所有的成绩，否则排序无法进行。

在 Objective-C 中，可以定义一个名为 `grades` 的变量，它代表的不是一个成绩值，而是一组成绩值。然后通过索引或下标的数字引用其中的各个元素。数学中，有下标的变量 x_i 指的是 x 集合中的第 i 个元素，而在 Objective-C 语言中，类似的符号为：

`x[i]`

所以表达式

`grades[5]`

（读作“`grades sub 5`”）指的是名为 `grades` 的数组中索引为 5 的元素。在 Objective-C 语言中，数组元素的索引以 0 开头

`grades[0]`

实际上指的是数组的第一个元素。

任何一个数组元素都可用在常规变量上。比如，使用以下语句将数组的值赋给另一个变量

`g = grades[50];`

表达式使用 `grades[50]` 的值，并将它赋给变量 `g`。一般来说，如果将 `i` 声明为整型变量，那么语句

`g = grades[i];`

使用索引为 `i` 的 `grades` 数组元素的值，将其赋给 `g`。

通过在等号的左侧指定数组元素，就可以将数值存储到数组元素中。下面的语句

`grades[99] = 95;`

将数值 95 存储到数组 `grades` 索引为 99 的元素中。

通过改变数组下标的变量值，可以轻松地浏览数组中的元素。因此，如下 `for` 循环：

```
for ( i = 0; i < 100; ++i )
    sum += grades[i];
```

依次浏览数组 `grades` 的前 100 个元素（元素 0~99），并且将每个成绩相加之和赋给变量 `sum`。当 `for` 循环结束时，变量 `sum` 包含数组 `grades` 的前 100 个元素值之和（假设在循环开始之前 `sum` 值设为 0）。

和其他变量类型一样，必须在使用之前先声明数组。数组的声明涉及声明数组所包含元素的数值类型，如 `int`、`float` 或者对象，以及将存储在数组中的最大元素数目。

定义

```
Fraction *fracts [100];
```

规定 `fracts` 为包含 100 个分数的数组。通过使用下标 0~99，可以有效地引用该数组的元素。

表达式

```
fracts[2] = [fracts[0] add: fracts[1]];
```

调用 `Fraction` 类的 `add:` 方法将数组 `fracts` 的前两个分数相加，并将结果存储到数组第 3 个位置。

代码清单 13-1 会显示前 15 个斐波那契数的表格，预计一下输出结果，表中每个数值之间有什么关系？

代码清单 13-1

```
// 生成前 15 个斐波那契数的程序
#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    @autoreleasepool {
        int Fibonacci[15], i;

        Fibonacci[0] = 0; /* by definition */
        Fibonacci[1] = 1; /* ditto */
```

```

    for ( i = 2; i < 15; ++i )
        Fibonacci[i] = Fibonacci[i-2] + Fibonacci[i-1];

    for ( i = 0; i < 15; ++i )
        NSLog (@"%i", Fibonacci[i]);
}
return 0;
}

```

代码清单 13-1 输出

```

0
1
1
2
3
5
8
13
21
34
55
89
144
233
377

```

对于前两个斐波那契数，我们称为 F_0 和 F_1 ，分别定义为 0 和 1。此后的每个斐波那契数 F_i 都定义为前两个斐波那契数 F_{i-2} 和 F_{i-1} 之和。所以， F_0 和 F_1 数值之和是 F_2 的值。前面的程序中，计算 `Fibonacci[0]` 和 `Fibonacci[1]` 之和就可以直接计算出 `Fibonacci[2]`。这个计算公式是在 `for` 循环中执行的，计算出 F_2 到 F_{14} 的值（即 `Fibonacci[2]` 到 `Fibonacci[14]` 的值）。

13.1.1 数组元素的初始化

正如在声明变量时可以给它赋初值一样，也可以给数组元素赋初值。从第一个元素开始，简单地列出数组元素的初值，就可以实现。数组中的值由逗号分隔，将整个数组置于一对大括号内。

语句

```
int integers[5] = { 0, 1, 2, 3, 4 };
```

将 0 赋给 `integers[0]`，1 赋给 `integers[1]`，2 赋给 `integers[2]`，以此类推。

字符数组以同样的方式初始化，所以表达式

```
char letters[5] = { 'a', 'b', 'c', 'd', 'e' } ;
```

定义了字符数组 `letters`，并将 5 个元素分别初始化为字符'a'、'b'、'c'、'd'和'e'。

不必完全初始化整个数组。如果指定了较少的初始化值，那么只会初始化等量的元素，并且数组中其余元素被设为 0。因此，声明

```
float sample_data[500] = { 100.0, 300.0, 500.5 } ;
```

将数组 `sample_data` 的前 3 个元素初始化为 100.0、300.0 和 500.5，其余 497 个元素被设为 0。

通过将元素编号放在一对大括号中，允许以任意顺序初始化指定的数组元素。比如

```
int x = 1233;
int a[] = { [9] = x + 1, [2] = 3, [1] = 2, [0] = 1 } ;
```

定义了一个名为 `a` 的 10 个元素数组（根据数组中最大索引得出的），并将数组的最后一个元素初始化为 `x+1(1234)`。此外，它的前 3 个元素分别被初始化为 1、2 和 3。

13.1.2 字符数组

代码清单 13-2 的目的是阐明如何使用字符数组。然而，下面有一个值得讨论的问题，你能发现它吗？

代码清单 13-2

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        char word[] = { 'H', 'e', 'l', 'l', 'o', '!' } ;
        int i;

        for ( i = 0; i < 6; ++i )
            NSLog ("%c", word[i]);
    }
    return 0;
}
```

代码清单 13-2 输出

```
H
e
l
l
o
!
```

如果这样定义，则会自动根据初始化元素的数目确定数组的大小。因为代码清单 13-2 为数组 `word` 列出了 6 个初始值，所以 Objective-C 语言隐式地将该数组定义为 6 个元素。

只要在定义数组时初始化了数组中的每个元素，这种方式就没有问题。但如果不是这种情况，就必须显式地给出数组的大小。

如果在字符数组结尾添加一个终止空字符（`'\0'`），就产生了一个通常称为字符串的变量。如果将代码清单 13-2 中数组 `word` 的初始化语句替换为

```
char word[] = { 'H', 'e', 'l', 'l', 'o', '!', '\0' } ;
```

随后，就可以使用 `NSLog` 语句显示这个字符串：

```
NSLog (@"%s", word);
```

因为 `%s` 格式字符表明 `NSLog` 会持续地显示字符，直到遇到终止空字符，也就是在 `word` 数组最后添加的这个字符，这个语句是没有问题的。

13.1.3 多维数组

到目前为止，看到的数组都是线性数组。也就是说，它们都是一维的。C 语言允许定义任意维的数组。本节将会讲述二维数组的内容。

二维数组比较普遍的应用是矩阵。考虑下面的 4×5 矩阵。

10	5	-3	17	82
9	0	0	8	-7
32	20	1	0	14
0	0	8	7	6

在数学中，通常使用双下标引用矩阵中的元素。如果将上面的矩阵命名为 M ，符号 M_{ij} 则代表在第 i 行第 j 列的元素，其中 i 的范围从 1 到 4， j 的范围从 1 到 5。符号 $M_{3,2}$ 指的值为 20，它位于矩阵第 3 行第 2 列。类似地， $M_{4,5}$ 指的

是第 4 行第 5 列的元素（值为 6）。

在 Objective-C 语言中，引用二维数组的元素也会使用到类似的概念。然而，因为 Objective-C 语言都是从 0 开始计数的，所以矩阵的第一行实际上是 0 行，而第一列是 0 列。那么上面矩阵的行列标识如下所示：

Row (i)	Column (j)				
	0	1	2	3	4
0	10	5	-3	17	82
1	9	0	0	8	-7
2	32	20	1	0	14
3	0	0	8	7	6

数学上使用符号 M_{ij} ，而 Objective-C 语言中使用等价的符号：

`M[i][j]`

记住，第一个索引指的是行数，第二个索引指的是列数。因此，语句

```
sum = M[0][2] + M[2][4];
```

将 0 行 2 列的数值（-3）和 2 行 4 列的数值（14）相加，并将结果 11 赋给变量 `sum`。

定义二维数组的方式和一维数组的相同，于是

```
int M[4][5];
```

声明 `M` 为 4 行 5 列的二维数组，总共包含 20 个元素。数组中的每个位置都包含整型值。

初始化二维数组的方法类似于一维等价数组的初始化方式。列出初始化元素时，它们的值是按行排列的。利用花括号分别初始化各行。因此，要将数组 `M` 定义和初始化为前面表中列出的元素，可以使用下面的表达式：

```
int M[4][5] = {
    { 10, 5, -3, 17, 82 },
    { 9, 0, 0, 8, -7 },
    { 32, 20, 1, 0, 14 },
    { 0, 0, 8, 7, 6 }
};
```

需要特别注意上面这一个语句的语法，除了最后一行，每行结束的花括号之后必须加上逗号。实际上，内层括号可以省略。如果没有内层括号，将按行

进行初始化。所以，上一个语句可以如下编写：

```
int M[4][5] = { 10, 5, -3, 17, 82, 9, 0, 0, 8, -7, 32,
               20, 1, 0, 14, 0, 0, 8, 7, 6 } ;
```

和一维数组一样，也不必初始化整个二维数组。如下语句：

```
int M[4][5] = {
    { 10, 5, -3 } ,
    { 9, 0, 0 } ,
    { 32, 20, 1 } ,
    { 0, 0, 8 }
} ;
```

只初始化了矩阵每行的前 3 个元素，其余的值都设为 0。注意，在这种情况下，需要使用内层花括号对，确定能够正确地初始化。如果没有这些括号，将会初始化前两行和第三行的前两个元素（请自行验证这种做法）。

13.2 函数

到目前为止，每个程序中接触到的 `NSLog` 函数就是一个函数例子。事实上，每个程序还用到一个名为 `main` 的函数。回顾你编写的第一个程序（代码清单 2-1），它在终端显示“`programming is fun.`”这个短语。

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

int main (int argc, char * argv[])
{
    @autoreleasepool {
        NSLog(@"Programming is fun.");
    }
    return 0;
}
```

下面是名为 `printMessage` 的函数，它产生同样的结果：

```
void printMessage (void)
{
    NSLog(@"Programming is fun.");
}
```

第一行函数定义会向编译器说明函数的 4 件事情：

- 谁可以调用这个函数。

- 函数的返回值类型。
- 函数的名称。
- 函数使用的参数数目和类型。

函数 `printMessage` 的第一行告知编译器 `printMessage` 是这个函数的名称，并且函数不返回任何值（第一个 `void` 关键字的用途）。与方法不同，这里不必将函数的返回值类型放在一对圆括号中。事实上，如果这么做，将得到一条编译错误消息。

在告知编译器 `printMessage` 不返回任何值之后，第二个 `void` 关键字的用途说明该函数没有任何参数。

回忆一下，Objective-C 系统中 `main` 是一个特殊的名称，它总是指示程序开始运行的位置。每个程序都必须有一个 `main` 函数。所以，可以向前面的代码添加一个 `main` 函数，使这个程序完整，如代码清单 13-3 所示。

代码清单 13-3

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

void printMessage (void)
{
    NSLog(@"Programming is fun.");
}

int main (int argc, char *argv[])
{
    @autoreleasepool {
        printMessage ();
    }
    return 0;
}
```

代码清单 13-3 输出

```
Programming is fun.
```

代码清单 13-3 包含两个函数 `printMessage` 和 `main`。前面提到，调用函数并不是新的概念。因为 `printMessage` 不带有参数，所以可以简单地在名称后面添加一对圆括号调用。

13.2.1 参数和局部变量

在第5章“循环结构”中，编写了一个计算三角数的程序。这里定义一个产生三角数的函数，并将其命名为 `calculateTriangularNumber`。通过函数的一个参数，指定要计算哪个三角数。然后这个函数计算出所求的数值并显示结果。代码清单 13-4 显示了完成这项任务的函数，以及测试它的 `main` 函数。

代码清单 13-4

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

// 计算第 n 个三角数的函数

void calculateTriangularNumber (int n)
{
    int i, triangularNumber = 0;

    for ( i = 1; i <= n; ++i )
        triangularNumber += i;

    NSLog (@\"Triangular number %i is %i\", n, triangularNumber);
}

int main (int argc, char *argv[])
{
    @autoreleasepool {
        calculateTriangularNumber (10);
        calculateTriangularNumber (20);
        calculateTriangularNumber (50);
    }
    return 0;
}
```

代码清单 13-4 输出

```
Triangular number 10 is 55
Triangular number 20 is 210
Triangular number 50 is 1275
```

函数 `calculateTriangularNumber` 的第一行是

```
void calculateTriangularNumber (int n)
```

它告诉编译器 `calculateTriangularNumber` 是一个函数，它不返回任何值（关键字 `void`），并带有一个名为 `n` 的 `int` 参数。再次注意，不能像编写方法那样，

将参数类型放在圆括号中。

左花括号表示函数定义的开始。因为你想要计算第 n 个三角数，所以必须设置一个变量，以便在计算过程中存储三角数的值。还需要一个变量作为循环的索引。为此，定义变量 `TriangularNumber` 和 `i`，将其声明为整型变量。这些变量的定义及初始化方式与前面程序 `main` 函数中的方式相同。

在函数中，局部变量的作用与方法中的一样：如果在函数内给变量赋予初始值，那么每次调用该函数时，都会指定相同的初始值。如果使用了自动引用计数（ARC），那么每次调用函数（方法）时，局部对象的变量都会默认初始化为空。

在函数中（和在方法中一样）定义的变量称为自动局部变量。因为每次调用该函数时，它们都自动“创建”，并且它们的值对于函数来说是局部的。

静态局部变量用关键字 `static` 声明，它们的值在函数调用的过程中保留下来，并且初始值默认为 0。

局部变量的值只能在定义该变量的函数中访问，不能从函数之外访问。

回到我们的程序示例，定义局部变量之后，函数计算三角数并在终端显示结果。右花括号表示函数的结束。

在 `main` 函数中，数值 10 是在第一次调用函数 `calculateTriangularNumber` 作为参数传递的。然后，执行到该函数，数值 10 成为函数中形参 n 的值。该函数计算出第 10 个三角数的值并显示结果。

再次调用函数 `calculateTriangularNumber` 时，传递参数 20。经过与前面描述相似的过程，该数值成为函数中的 n 值。然后，该函数计算出第 20 个三角数的值并显示答案。

13.2.2 函数的返回结果

和方法一样，函数也可以返回值。`return` 语句返回的值类型必须和函数声明的返回类型一致。如下函数

```
float kmh_to_mph (float km_speed)
```

定义了函数 `kmh_to_mph`，它使用一个名为 `km_speed` 的 `float` 参数，返回浮点型小数。类似地，

```
int gcd (int u, int v)
```

定义一个名为 `gcd` 的函数, 它传递整型参数 u 和 v 并返回一个整型值。

让我们利用函数重新编写代码清单 5-7 中求最大公约数的算法。该函数的两个参数是要计算最大公约数 (`gcd`) 的两个数 (参见代码清单 13-5)。

代码清单 13-5

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

```
// 查找两个数的最大公约数的函数
```

```
// 非负整数值, 并返回结果
```

```
int gcd (int u, int v)
```

```
{
```

```
    int temp;
```

```
    while ( v != 0 )
```

```
    {
```

```
        temp = u % v;
```

```
        u = v;
```

```
        v = temp;
```

```
    }
```

```
    return u;
```

```
}
```

```
main ()
```

```
{
```

```
    @autoreleasepool {
```

```
        int result;
```

```
        result = gcd (150, 35);
```

```
        NSLog(@"The gcd of 150 and 35 is %i", result);
```

```
        result = gcd (1026, 405);
```

```
        NSLog(@"The gcd of 1026 and 405 is %i", result);
```

```
        NSLog(@"The gcd of 83 and 240 is %i", gcd (83, 240));
```

```
    }
```

```
    return 0;
```

```
}
```

代码清单 13-5 输出

```
The gcd of 150 and 35 is 5
```

```
The gcd of 1026 and 405 is 27
```

```
The gcd of 83 and 240 is 1
```

函数 `gcd` 的说明带有两个整型参数。函数通过形参名称 `u` 和 `v` 指明这些参数。将变量 `temp` 声明为整型，程序将在终端显示参数 `u`、`v` 的值和相关消息。然后，通过函数计算并返回这两个整数的最大公约数。

表达式

```
result = gcd (150, 35);
```

调用函数 `gcd` 使用参数 150 和 35，将返回值存储到变量 `result` 中。

倘若省略函数的返回类型声明，如果函数确实返回值，编译器就会假设该值为整数。许多程序员利用这个事实，省略整数的函数返回类型声明。但这是不好的编程习惯，应该避免。

函数默认的返回类型与方法默认的不同。回想一下，如果没有为方法指定返回类型，编译器就会假设它返回 `id` 类型的值。同样，应该为方法声明返回类型，不要依赖这个事实。

声明返回类型和参数类型

前面提到过 Objective-C 语言编译器假设函数的默认返回值是整型值。更确切地说，无论什么时候调用一个函数，编译器都会假设这个函数的返回类型为 `int`，除非发生以下两种情况之一：

- 在函数被调用之前，已经在程序中定义了该函数。
- 在遇到函数调用之前，已经声明了该函数的返回值类型。声明函数的返回值类型和参数类型称为原型（`prototype`）声明。

函数声明不仅用于声明函数的返回类型，而且用于告知编译器，该函数带有多少个参数及其类型。这类似于定义新类时在 `@interface` 部分中声明方法。

要将 `absoluteValue` 定义为一个返回 `float` 型值并带有一个 `float` 类型参数的函数，可以使用以下原型声明：

```
float absoluteValue (float);
```

可以看到，只需要在圆括号中指定参数类型，而不是参数名称。如果愿意，可以选择在类型之后指定“伪”名称：

```
float absoluteValue (float x);
```


这个名称和函数定义所用到的不必相同，反正编译器会忽略它。

编写原型声明最简单的方法，就是简单地复制函数实际定义的第一行代码。记住：在结尾处添加分号。

如果函数的参数数目不定（比如 `NSLog` 和 `scanf`），必须告知编译器。如下声明

```
void NSLog (NSString *format, ...);
```

告知编译器 `NSLog` 将使用一个 `NSString` 对象作为它的第一个参数，之后是任意数目的附加参数（……的用途）。`NSLog` 在一个特殊文件 `Foundation/Foundation.h`^①中声明，这就是为什么要在每个程序的开头添加下面一行语句。

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

如果没有这一行语句，编译器可以假设 `NSLog` 使用固定数目的参数，这可能导致产生不正确的代码。

只有已经在调用函数之前添加了函数的定义或声明了该函数及其参数类型时，编译器才会在调用该函数时自动将参数转换为相应的类型。

下面是关于函数的一些注意事项和建议：

- 默认情况下，编译器假设函数返回 `int`。
- 定义返回值为 `int` 的函数时，直接将它定义为 `int`。
- 当定义没有返回值的函数时，将它定义为 `void`。
- 只有当前面已经定义或声明了这个函数，编译器才会将参数转换成函数认可的类型。

为了安全起见，建议在程序中声明所有的函数，即使它们在被调用之前已经定义了（将来你可能决定将这些函数移到文件的其他位置或者移到另一个文件中）。好的策略是将函数声明放到一个头文件中，然后只将这个头文件导入（`import`）你的模块即可。

默认情况下，函数是外部的。即对于函数默认的作用域来说，所有与该函数链接在一起的文件中的所有函数或方法都可以调用它。通过将其定义为 `static`

① 从技术上讲，它是在 `NSObjCRuntime.h` 中定义的，`Foundation.h` 文件内部导入了这个文件。

(静态的), 可以限制函数的作用域。将关键字 `static` 放在函数声明前即可, 语句如下:

```
static int gcd (int u, int v)
{
    ...
}
```

静态函数只可以由和该函数定义位于同一文件的其他函数或者方法调用。

13.2.3 函数、方法和数组

若要向函数或方法传递单个数组元素, 可使用常规方式将数组元素指定为参数。因此, 如果有一个用来计算平方根的函数 `squareRoot`, 想要计算 `averages[i]` 的平方根并将结果赋给名为 `sq_root_result` 的变量, 使用如下表达式即可。

```
sq_root_result = squareRoot (averages[i]);
```

向函数或方法传递整个数组是完全不同的情况。要传递数组, 只需在函数调用或者方法调用中列出数组名称, 并且不需要任何下标。举个例子, 假设前面将 `grade_scores` 定义为包含 100 个元素的数组, 那么下面的表达式

```
minimum (grade_scores)
```

实际上将数组 `grade_scores` 中的 100 元素都传递给名为 `minimum` 的函数。很显然, 函数 `minimum` 必须使用整个数组作为参数, 也必须有适当的形参声明。

下面有一个函数用于寻找包含指定元素个数的数组中的最小整数值:

// 在数组中查找最小值的函数

```
int minimum (int values[], int numElements)
{
    int minValue, i;

    minValue = values[0];

    for ( i = 1; i < numElements; ++i )
        if ( values[i] < minValue )
            minValue = values[i];

    return (minValue);
}
```

函数 `minimum` 定义为带有两个参数：第一个是要查找最小数的数组，第二个是数组中的元素个数。在函数头中，`values` 之后的一对方括号用来告知 Objective-C 编译器：`values` 是整型数组。编译器并不关心这个数组有多大。

形参 `numElements` 用作 `for` 语句的上限。这样，`for` 语句依次查找 `values[1]` 到数组最后一个元素，即 `values[numElements-1]`。

如果函数或者方法更改了数组元素的值，那么这个变化将影响到传递到该函数或方法的原始数组，而且这个变化在函数或方法执行完之后依然有效。

值得讨论一下数组的行为和单个变量或数组元素（函数或方法不能更改它们的值）不同的原因。我们说过，调用函数或方法时，作为参数传递的值将被复制到相应的形参中。这个论述依然是有效的。但是，使用数组时，并非将整个数组的内容复制到形参数组中，而是传递一个指针，它表示数组所在的计算机内存地址。所以，对形参数组所做的所有更改实际上都是对原始数组而不是数组的副本执行的。因此，函数或方法返回时，这些变化仍然有效。

13.3 块 (Blocks)

块是对 C 语言的一种扩展。它并未作为标准 ANSI C 所定义的部分，而是由 Apple 公司添加到语言中的。块看起来更像是函数，这样的语法需要一些时间来适应。可以给块传递参数，正如给函数传递一样，块也具有返回值。与函数不同的是，块定义在函数或者方法内部，并能够访问在函数或者方法范围内块之外的任何变量。一般来说，这些变量能够访问但是并不能够改变这些变量的值。有一个特殊的块修改器（由块前面含有两个下画线的字符组成）能够修改块内变量的值，后面会简短介绍如何使用。

块能够作为参数传递给函数或方法，在本书第二部分“Foundation Framework”将会学习到以块作为参数的一些方法。块的其中一个优势在于能够让系统分配给其他处理器或应用的其他线程执行。

下面看一个简单的例子，曾经写过叫作 `printMessage:` 的一个函数。

```
void printMessage (void)
{
    NSLog(@"Programming is fun.");
}
```

这样一个块能够完成同样的任务：

```
^(void)
{
    NSLog(@"Programming is fun.");
}
```

块是以插入字符“^”开头为标识的。后面跟的一个括号表示块所需要的参数列表。在这个例子中，块并没有参数，所以在函数定义中仅需填入 `void`。

同样，也可以将这个块赋给一个名为 `printMessage` 的变量，只要变量声明正确：

```
void (^printMessage)(void) =
    ^(void){
        NSLog(@"Programming is fun.");
    } ;
```

等号左边表示 `printMessage` 指向一个没有参数和返回值的块指针。需要注意的是，赋值语句是以分号终止的。

执行一个变量引用的块，与函数的调用方式一致。

```
printMessage ();
```

代码清单 13-6 将这些都放在一个例子中。

代码清单 13-6

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        void (^print_Message)(void) =
            ^(void) {
                NSLog(@"Programming is fun.");
            } ;

        printMessage ();
    }
    return 0;
}
```

代码清单 13-6 输出

```
Programming is fun.
```

代码清单 13-7 是代码清单 13-4 使用块替代函数重新编写的。块能够定义为全局或者局部的。在这个例子中，将块定义在 `main` 外部，让它扩展到全局范围。

代码清单 13-7

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

// 计算第 n 个三角数的块

void (^calculateTriangularNumber) (int) =
    ^(int n) {
        int i, triangularNumber = 0;

        for ( i = 1; i <= n; ++i )
            triangularNumber += i;

        NSLog (@\"Triangular number %i is %i\", n, triangularNumber);
    } ;

int main (int argc, char *argv[])
{
    @autoreleasepool {
        calculateTriangularNumber (10);
        calculateTriangularNumber (20);
        calculateTriangularNumber (50);
    }
    return 0;
}
```

代码清单 13-7 输出

```
Triangular number 10 is 55
Triangular number 20 is 210
Triangular number 50 is 1275
```

比较代码清单 13-4 中的函数字符和代码清单 13-7 中定义的块。指向块的指针变量 `calculateTriangularNumber` 以 `int` 作为参数，并且没有返回值。

在本节开始部分提到块是可以有返回值的。对代码清单 13-5 中的 `gcd` 函数使用块的形式重新改写：

```
int (^gcd) (int, int) =
    ^(int u, int v){
        int temp;
```

```

while ( v != 0 )
{
    temp = u % v;
    u = v;
    v = temp;
}

return u;
} ;

```

块可以访问在其范围内定义的变量。变量的值同时作为块中定义的值，正如代码清单 13-8。

代码清单 13-8

```

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    @autoreleasepool {
        int foo = 10;

        void (^printFoo)(void) =
            ^(void) {
                NSLog(@"foo = %i", foo);
            } ;

        foo = 15;

        printFoo ();
    }
    return 0;
}

```

代码清单 13-8 输出

```

foo = 10

```

块 `printFoo` 可以访问本地变量 `foo` 的值。注意 `foo` 值显示的是 10，并不是 15。这是因为变量 `foo` 在定义块的同时已经具有了值，而不是在块执行的时候。

不可以在块外部修改已经定义过变量的值。所以，如果你试图在块内部改变 `foo` 的值（如代码清单 13-9），就会得到编译器提示的错误消息：给只读变量“`foo`”赋了值。

代码清单 13-9

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        int foo = 10;

        void (^printFoo) (void) =
            ^(void) {
                NSLog (@"foo = %i", foo);
                foo = 20;    // ** 该行会产生编译错误
            };

        foo = 15;

        printFoo ();
        NSLog (@"foo = %i", foo);
    }
    return 0;
}
```

代码清单中，如果在定义本地变量 `foo` 之前插入 `__block` 修改器：

```
__block int foo = 10;
```

然后运行程序，会得到两条输出语句：

```
foo = 15
foo = 20
```

第一行显示的是调用块时 `foo` 的值，第二行验证块中能否将 `foo` 值改变为 20。

在这里先结束对块的介绍。在第 15 章中还会有更多使用的例子。

13.4 结构

除了数组之外，Objective-C 语言还提供了另一种组合元素的工具。这种工具就是结构，它构成了本节讨论的基础。

假设要在程序中存储日期（比如 7/18/14），它也许用于程序输出的开头，或是出于计算需要。存储日期的自然方法就是：将月份赋给名为 `month` 的整型变量，日期赋给整型变量 `day`，年份赋给整型变量 `year`。那么语句

```
int month = 7, day = 18, year = 2011;
```

可以实现该任务，这是完全可接受的方式。但是，如果你的程序还需要存储很多日期，该怎么办？如果可以使用某种方式将这 3 个变量组合起来就更好了。

在 Objective-C 语言中，可以定义一个名为 `date` 的结构，它包含 3 个分别代表年、月、日的组成部分。这种定义的语法相当直观，语句如下：

```
struct date
{
    int month;
    int day;
    int year;
};
```

`date` 结构恰好定义了 3 个整型元素，分别名为 `month`、`day` 和 `year`。本质上说，`date` 的定义在语言中定义了一种新类型，因为随后就可以将变量声明为 `struct date` 类型了，像下面这样：

```
struct date today;
```

还可以像下面这样另外定义一个名为 `purchaseDate` 的同类型变量：

```
struct date purchaseDate;
```

或者，可以直接在同一行中包含两个定义，如下面一行语句：

```
struct date today, purchaseDate;
```

不同于 `int`、`float` 或 `char` 型变量，处理结构变量时需要特殊语法。通过指明变量名称，在之后加上句点来访问结构成员。举个例子，要将变量 `today` 的 `day` 值设置为 21，可以编写如下语句：

```
today.day = 21;
```

注意变量名、句点和成员名称之间不允许出现空格。

现在，请等一下！这不是我们在访问对象的属性时使用的操作符吗？回忆一下，我们可以编写如下语句

```
myRect.width = 12;
```

来调用 `Rectangle` 对象的 `setter` 方法（称为 `setWidth:`），给它传递参数值 12。这里的意思很明了：编译器确定点运算符左边的是一个结构还是一个对象，然后进行相应的处理。

返回到 `struct date` 示例，要将 `today` 中的 `year` 赋值为 2014，可以使用如下表达式：

```
today.year = 2014;
```

最后，要检测 `month` 的值是否等于 12，可以使用如下表达式：

```
if ( today.month == 12 )
    next_month = 1;
```

代码清单 13-10 将前面的讨论组合成一个实际的程序。

代码清单 13-10

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

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

        struct date
        {
            int month;
            int day;
            int year;
        };

        struct date today;

        today.month = 9;
        today.day = 25;
        today.year = 2014;

        NSLog(@"Today's date is %i/%i/%.2i.", today.month,
              today.day, today.year % 100);
    }
    return 0;
}
```

代码清单 13-10 输出

```
Today's date is 9/25/14.
```

`main` 函数的第一条语句定义了名为 `date` 的结构，它包含 3 个整型成员，分别是 `month`、`day` 和 `year`。在第二条语句中，变量 `today` 声明为 `struct date` 类型。所以，第一条语句只是简单地向 Objective-C 编译器说明了 `date` 的结构，并没

有在计算机中分配存储空间。第二条语句定义了一个 `struct date` 类型的变量，所以导致内存中分配了空间，以便存储结构变量 `today` 的 3 个整型成员。

在赋值完成后，通过调用适当的 `NSLog` 语句来显示包含在结构中的值。`today.year` 除以 100 的余数是在传递给 `NSLog` 函数之前计算的，这样年份只显示 11。`NSLog` 中的 `%2i` 格式符号指明了最少显示两位字符，从而强制显示年份开头的 0。

谈到表达式的求值时，结构成员遵循的法则和 Objective-C 语言中普通变量一样。这样，将整型的结构成员除以整数的除法和整数除法一样，语句如下：

```
century = today.year / 100 + 1;
```

假设要编写一个简单的程序，它接收今天的日期作为输入数据，并向用户显示明天的日期。第一眼看上去，这似乎是一项非常简单的任务。可以让用户输入今天的日期，然后通过一系列语句计算出明天的日期，语句如下：

```
tomorrow.month = today.month;  
tomorrow.day = today.day + 1;  
tomorrow.year = today.year;
```

当然，对于大多数日期来讲，上面的语句都可以得出正确结果，但是不能正确处理以下两种情况：

- 如果今天的日期是一个月的最后一天。
- 如果今天的日期是一年的最后一天（即今天的日期是 12 月 31 日）。

确定今天的日期是不是一个月最后一天的一种简便方法是设置对应于每月天数的整型数组。在数组中查找特定的月份，就可以得到当月的天数。

13.4.1 结构的初始化

初始化结构与初始化数组类似，将元素列在一对花括号中，元素之间以逗号隔开。

要将 `date` 结构的变量初始化为 2014 年 7 月 2 日，可以使用下面的语句：

```
struct date today = { 7, 2, 2014 } ;
```

和数组的初始化一样，列出的值可以少于结构中包含的元素个数。所以，语句

```
struct date today = { 7 } ;
```

将 `today.month` 初始化为 7，但是没有给 `today.day` 或者 `today.year` 赋初值。在这种情况下，它们的默认初始值是未定义的（`undefined`）。

在初始化列表中，用下面的表示方式

```
.member = value
```

可以以任意顺序初始化结构中指定的成员，语句如下：

```
struct date today = { .month = 7, .day = 2, .year = 2014 } ;
```

和

```
struct date today = { .year = 2014 } ;
```

最后一个语句只将该结构中的 `year` 元素设置为 2014。你知道，其余两个元素是未定义的。

13.4.2 结构中的结构

Objective-C 语言在定义结构方面提供了极大的灵活性。比如，可以定义一个结构，它本身包含其他结构作为自己的一个或多个成员，或者可以定义包含数组的结构。

第 10 章“变量和数据类型”中学习过 `typedef` 语句。在 iOS 程序中经常用于矩形。例如，使用矩形定义 iPhone 或者 iPad 屏幕尺寸和窗口的位置。同样使用矩形定义子窗口（称为子视图）的尺寸和位置。将用到 3 种基本数据类型，它们都是由 `typedef` 定义的。

(1) `CGPoint` 用于描述 (x, y) 点。

(2) `CGSize` 用于描述宽和高。

(3) `CGRect` 用于描述包含原点（一个 `CGPoint`）和尺寸（一个 `CGSize`）的矩形。

Apple 的 `CGGeometry.h` 头文件是由 `typedef` 定义的：

```
/* 点 */
```

```
struct CGPoint {
    CGFloat x;
    CGFloat y;
};
typedef struct CGPoint CGPoint;
```

```

/* 宽和高的尺寸 */

struct CGSize {
    CGFloat width;
    CGFloat height;
};
typedef struct CGSize CGSize;

/* 矩形 */

struct CGRect {
    CGPoint origin;
    CGSize size;
};
typedef struct CGRect CGRect;

```

typedef 提供了声明变量更简便的方式，不需要使用 **struct** 关键字。**CGFloat** 是由 **typedef** 定义的基本浮点数据类型。因此，如果希望声明一个 **CGPoint** 变量并且把成员 *x* 设置为 100 和成员 *y* 设置为 200，可编写如下代码：

```

CGPoint startPt;

startPt.x = 100;
startPt.y = 200;

```

记得 **startPt** 是一个结构，而不是一个对象（特征是变量名前缺少星号）。**Apple** 也提供了简便的函数用于创建 **CGRect**、**CGSize** 和 **CGRect** 结构。例如，

```
CGPoint startPt = CGPointMake (100.0, 200.0);
```

函数 **CGSizeMake** 和 **CGRectMake** 的作用正如函数名称一样。

如果希望定义一个矩形，按照以下方式指定尺寸：

```

CGSize rectSize;

rectSize.width = 200;
rectSize.height = 100;

```

或者使用 **CGSizeMake** 函数：

```
CGSize rectSize = CGSizeMake (200.0, 100.0);
```

继续创建一个包含尺寸和原点的矩形：

```

CGRect theFrame;
theFrame.origin = startPt;
theFrame.size = rectSize;

```

（这里就不再介绍 `CGRectMake` 函数的使用。）

如果希望取得矩形的宽度，需要编写如下语句：

```
theFrame.size.width
```

改变宽度为 175：

```
theFrame.size.width = 175;
```

最后设置原点为 (0, 0)：

```
theFrame.origin.x = 0.0;
```

```
theFrame.origin.y = 0.0;
```

这些只是使用结构的一部分例子。你将会在程序中经常使用到这些结构。

13.4.3 关于结构的补充细节

这里应该提到定义结构时有一些灵活性。首先，将变量定义为特定结构类型的同时，声明这个结构是合法的。只需要将变量名称放在结构定义的终止分号之前即可。比如，表达式

```
struct date
{
    int month;
    int day;
    int year;
} todaysDate, purchaseDate;
```

定义了 `date` 结构，同时也声明了变量 `todaysDate` 和 `purchaseDate` 是这个类型。还可以按常规方式来给变量赋初值。因此

```
struct date
{
    int month;
    int day;
    int year;
} todaysDate = { 9, 25, 2014 } ;
```

定义了 `date` 结构，同时也将变量 `todaysDate` 赋了初值。

如果定义结构时，也定义了该结构类型的所有变量，那么可以省略结构名称。所以表达式

```
struct
{
    int month;
```

```
int day;  
int year;  
} dates[100];
```

定义了包含 100 个元素的数组 `dates`。每个元素都是包含月、日和年 3 个整型成员的结构。因为没有为这个结构提供名称，所以定义同类型变量的唯一方式就是再次显式地定义这个结构。

13.4.4 不要忘记面向对象编程思想

现在你知道如何定义结构来存储日期，并且编写了各种函数来操纵这些 `date` 结构。但是，面向对象编程又体现在哪里？难道不应该建立一个名为 `Date` 的类，然后构造方法来使用 `Date` 对象？这难道不是一种更好的方法？当然，答案是肯定的。我们在讨论程序中存储日期时，希望你能够想到这个问题。

当然，如果必须在程序中处理大量日期，那么定义一个类和方法是更好的途径。事实上，Foundation 框架有两个类 `NSDate` 和 `NSDateCalendarDate`，正是用于这个目的。这里会留一个练习，让你设计一个 `Date` 类，用对象方式而不是结构方式来处理日期。

13.5 指针

指针可以高效地表示复杂的数据结构，作为参数传递给函数和方法的值，并且更准确、高效地处理数组。在本章的结尾处，我们还会提示在 Objective-C 语言中，指针对于对象实现的重要性。

我们在第 8 章“继承”中引入了指针的概念，那时我们谈到了 `Point` 和 `Rectangle` 类，并指出同一对象可以有多个引用。

要了解指针操作方式，首先必须明白间接寻址的概念。在日常生活中，我们已经习惯了这种说法。比如，假设需要为我的打印机买一个彩色墨盒。在我工作的公司，所有的采购活动都由采购部门负责。所以，可以打电话给负责采购的 Jim，让他为我订购一个新墨盒。Jim 将打电话给本地供应商店来订购该墨盒。这样，事实上，我获得新墨盒的方式就是间接的，因为我并没有直接从供应商店处订购墨盒。

这种间接方式也是 Objective-C 中指针的工作方式。指针提供了一种途径，

间接地访问特定数据项的值。有理由认为通过采购部门订购墨盒很有道理（比如，不必了解从哪家供应商店订购墨盒），所以也有很好的理由认为有时在 Objective-C 中使用指针很有道理。

说得足够多了，应该看看指针的工作方式了。假设已经定义了一个名为 `count` 的变量，语句如下：

```
int count = 10;
```

还可以通过以下声明定义一个名为 `intPtr` 的变量，它允许间接访问 `count` 的值：

```
int *intPtr;
```

在 Objective-C 系统中，星号定义变量 `intPtr` 是 `int` 的指针类型。这表示在这个程序中，`intPtr` 用于间接访问一个或多个整型变量的值。

在前面的程序中，我们学习了如何在 `scanf` 调用中使用 `&` 运算符。在 Objective-C 语言中，它是一元运算符（又称为地址运算符），用来得到变量的指针。所以，如果 `x` 是特定类型的变量，那么表达式 `&x` 就是该变量的指针。如果需要，`&x` 可以赋值给任何指针变量，只要该指针指向的类型与 `x` 相同。

所以，给出 `count` 和 `intPtr` 的定义，可以编写如下表达式：

```
intPtr = &count;
```

目的是建立起 `intPtr` 和 `count` 之间的间接引用。地址运算符将 `intPtr` 赋值为指向变量 `count` 的指针，而不是 `count` 的值。图 13.1 描述了 `intPtr` 和 `count` 之间的关系。箭头说明 `intPtr` 并不直接包含 `count` 的值，而是包含变量 `count` 的指针。

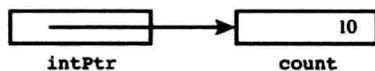


图 13.1 一个整型指针

要通过指针变量 `intPtr` 引用 `count` 的内容，可以使用间接寻址运算符，即星号（`*`）。如果 `x` 是 `int` 类型，那么语句

```
x = *intPtr;
```

会将 `intPtr` 间接指向的值赋给变量 `x`。因为之前将 `intPtr` 设置为指向 `count`，所以这个语句的作用就是将变量 `count` 的数值 10 赋给变量 `x`。

代码清单 13-11 组合了之前的语句，演示两个基本的指针运算符：地址运算符（&）和间接寻址运算符（*）。

代码清单 13-11

// 说明指针的程序

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        int count = 10, x;
        int *intPtr;

        intPtr = &count;
        x = *intPtr;

        NSLog (@"count = %i, x = %i", count, x);
    }
    return 0;
}
```

代码清单 13-11 输出

```
count = 10, x = 10
```

变量 `count` 和 `x` 以常规方式声明为整型变量。在下一行，变量 `intPtr` 声明为“int 指针”类型。注意，这两个声明行可以合并为一行，语句如下：

```
int count = 10, x, *intPtr;
```

然后，对变量 `count` 应用地址运算符，它的作用就是创建该变量的指针。接着程序将该指针赋给变量 `intPtr`。

程序中下一条语句

```
x = *intPtr;
```

的执行过程为：间接运算符告知 Objective-C 系统创建变量 `intPtr`，它包含指向另一个数据项的指针。然后这个指针用来访问所需的数据项，该数据项的类型是由指针变量的声明指定的。因为在声明该变量时，已告知编译器 `intPtr` 指向整数。所以编译器知道表达式 `*intPtr` 指向的是整型数据值。而且，因为在前面的程序语句中，已经将 `intPtr` 设为指向整型变量 `count` 的指针，所以 `count` 的值

可以使用表达式 `*intPtr` 间接访问。

代码清单 13-12 演示了指针变量一些有趣的属性。这里用到了指向字符的指针。

代码清单 13-12

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        char c = 'Q';
        char *charPtr = &c;

        NSLog (@"%c %c", c, *charPtr);

        c = '/';
        NSLog (@"%c %c", c, *charPtr);

        *charPtr = '(';
        NSLog (@"%c %c", c, *charPtr);
    }
    return 0;
}
```

代码清单 13-12 输出

```
Q Q
/ /
( (
```

程序中定义了字符变量 `c`，并且将其初始化为字符 `Q`。在程序的下一行代码中，变量 `charPtr` 定义为“`char` 指针”类型，意思是无论存储在该变量中的是什么值，都应该看作字符的间接引用（即指针）。你将注意到可以使用常规方式给这个变量赋初值。在程序中赋给 `charPtr` 的值是指向变量 `c` 的指针，它是通过在变量 `c` 前面加上地址运算符得到的（注意，如果在该语句之后定义 `c`，那么这个初始化语句将产生编译错误，因为必须在表达式中引用变量的值之前声明这个变量）。

变量 `charPtr` 的声明和初始值的分配都可以等效地使用如下两个语句表示：

```
char *charPtr;
charPtr = &c;
```

(而不是像前一行声明暗示的那样, 通过语句

```
char *charPtr;
*charPtr = &c;
```

来表示)。

始终要记住: 在 Objective-C 语言中, 将指针设置指向一些值之前, 指针的值是没有意义的。

第一个 NSLog 调用仅仅显示变量 *c* 的内容及 *charPtr* 所引用的变量内容。因为你将 *charPtr* 设为指向变量 *c*, 所以显示的就是 *c* 的内容, 在程序输出的第一行就能得到验证。

该程序的下一行将字符 *'l'* 赋给字符变量 *c*。因为 *charPtr* 仍然指向变量 *c*, 所以最后的 NSLog 语句中显示 **charPtr* 的值就正确地显示为 *c* 的新值。这是非常重要的概念。除非更改 *charPtr* 的值, 否则表达式 **charPtr* 总是访问 *c* 的值。这样, 当 *c* 的值发生变化时, **charPtr* 的值也相应地改变。

前面的讨论有助于理解程序中下一条语句的工作方式。我们提到过, 除非更改 *charPtr*, 否则表达式 **charPtr* 将总是引用 *c* 的值。所以, 在以下表达式中

```
*charPtr = '(';
```

将左括号赋给 *c*。从形式上说是将字符 *'('* 赋给 *charPtr* 指向的变量。你知道这个变量是 *c*, 因为程序开始时, 将 *c* 的指针存入了 *charPtr*。

上述概念是理解指针操作的关键。如果还不是很清楚, 请再回顾一下这些知识。

13.5.1 指针和结构

你已经看到如何将指针定义为指向基本数据类型 (如 *int* 和 *char*)。但是指针还可以指向结构。在本章的前面, 定义了如下 *date* 结构:

```
struct date
{
    int month;
    int day;
    int year;
};
```

与使用下面的语句定义 *struct date* 类型的变量一样

```
struct date todaysDate;
```

也可以定义指向 `struct date` 变量的指针变量：

```
struct date *datePtr;
```

然后就可以用期望的方式使用刚才定义的变量 `datePtr`。比如，可以使用下面的赋值语句，将其设为指向变量 `todayDate` 的指针：

```
datePtr = &todayDate;
```

这样赋值之后，就可以用下面的方式，通过 `datePtr` 间接访问 `date` 结构的任何成员：

```
(*datePtr).day = 21;
```

这个语句的作用，就是将 `datePtr` 指向的 `date` 结构中的 `day` 成员设置为 21。括号是必需的，因为结构成员点运算符（.）比间接寻址运算符（*）的优先级高。

要测试存储在 `datePtr` 指向的 `date` 结构中 `month` 成员的值，可以使用如下语句：

```
if ( (*datePtr).month == 12 )
    ...
```

因为经常用到结构指针，所以该语言中存在着一个特殊的运算符。结构指针运算符 `->`，即短画线和紧跟其后的大于号，它让表达式

```
(*x).y
```

可以更清楚地表示为

```
x->y
```

所以，前面的 `if` 语句可以方便地写为

```
if ( datePtr->month == 12 )
    ...
```

我们写的代码清单 13-10 是第一个说明结构的程序。利用结构指针的概念，这个程序被改写成代码清单 13-13。

代码清单 13-13

```
// 说明结构的指针
#import <Foundation/Foundation.h>

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

```

{
    @autoreleasepool {

        struct date
        {
            int month;
            int day;
            int year;
        };

        struct date today, *datePtr;

        datePtr = &today;
        datePtr->month = 9;
        datePtr->day = 25;
        datePtr->year = 2014;

        NSLog(@"Today's date is %i/%i/%.2i.",
              datePtr->month, datePtr->day, datePtr->year % 100);
    }
    return 0;
}

```

代码清单 13-13 输出

```
Today's date is 9/25/14.
```

13.5.2 指针、方法和函数

可以按一般方式将指针作为参数传递给方法或函数，并且可以让函数或者方法返回指针。`alloc` 和 `init` 方法一直都在这么做，也就是返回指针。本章结尾对此有更详细的介绍。现在考虑代码清单 13-14。

代码清单 13-14

```

// 指针作为参数的函数
#import <Foundation/Foundation.h>

void exchange (int *pint1, int *pint2)
{
    int temp;

    temp = *pint1;
    *pint1 = *pint2;
    *pint2 = temp;
}

```

```
int main (int argc, char *argv[])
{
    @autoreleasepool {
        void exchange (int *pint1, int *pint2);
        int i1 = -5, i2 = 66, *p1 = &i1, *p2 = &i2;

        NSLog(@"i1 = %i, i2 = %i", i1, i2);
        exchange (p1, p2);
        NSLog(@"i1 = %i, i2 = %i", i1, i2);

        exchange (&i1, &i2);
        NSLog(@"i1 = %i, i2 = %i", i1, i2);
    }
    return 0;
}
```

代码清单 13-14 输出

```
i1 = -5, i2 = 66
i1 = 66, i2 = -5
i1 = -5, i2 = 66
```

函数 `exchange` 的目的是交换由两个参数指向的两个整型值。局部整型变量 `temp` 用于在交换时存放其中一个整数的值，它的值设为等于 `pint1` 指向的整型值。然后，`pint2` 指向的整数被复制到 `pint1` 指向的整数中，最后 `temp` 的数值被复制到 `pint2` 指向的整数中，这样就完成了交换。

`main` 函数定义了整数 `i1` 和 `i2`，并给它们分别赋值为 -5 和 66。然后，定义了两个整型指针 `p1` 和 `p2`，并分别将其设为指向 `i1` 和 `i2`，这个程序显示 `i1` 和 `i2` 的值，并且传递两个指针（`p1` 和 `p2`）作为参数来调用函数 `exchange`。函数 `exchange` 交换 `p1` 指向的整数值和 `p2` 指向的整数值。因为 `p1` 指向 `i1`，`p2` 指向 `i2`，所以函数交换的是 `i1` 和 `i2` 的值。第二个 `NSLog` 调用的输出验证交换函数运行良好。

第二次调用 `exchange` 函数比较有意思。这次，传递给该函数的参数是通过对 `i1` 和 `i2` 应用地址运算符手动创建的指针。因为表达式 `&i1` 产生了指向整型变量 `i1` 的指针，所以它符合函数所期望的第一个参数类型（整型指针）。这同样适用于第二个参数。从程序的输出结果中可以看出，函数 `exchange` 完成了这项任务，将 `i1` 和 `i2` 的数值交换回它们原本的值。

请仔细研究代码清单 13-14。它通过一个小例子，阐明了 Objective-C 语言中处理指针所需要了解的主要概念。

13.5.3 指针和数组

如果有一个包含 100 个整数的数组 `values`，那么可以定义一个名为 `valuesPtr` 的指针，它可以通过下面的表达式访问数组中的整数：

```
int *valuesPtr;
```

当定义用于指向数组元素的指针时，不能将指针定义为“数组指针”，而是要将指针定义为数组中所包含的元素类型。

如果你有一个 `Fraction` 对象数组 `fracts`。同样，可以通过下面的语句定义一个指针，用于指向 `fracts` 中的元素：

```
Fraction **fractsPtr;
```

要将 `valuesPtr` 设为指向数组 `values` 的第一个元素的指针，可以写成：

```
valuesPtr = values;
```

在这个例子中并没有用到地址运算符，因为 Objective-C 编译器将没有下标的数组名称看作是指向数组第一个元素的指针。所以，仅仅指明 `values` 而不带下标，其作用就是产生一个指向 `values` 第一个元素的指针。

要产生指向 `values` 首元素的指针，还有另一个等效方式，就是对数组第一个元素应用地址运算符。因此，语句

```
valuesPtr = &values[0];
```

将数组 `values` 第一个元素的指针存放到指针变量 `valuesPtr` 中。

要显示数组 `fracts` 中 `fractsPtr` 指向的 `Fraction` 对象，可以编写下面的表达式：

```
[*fractsPtr print];
```

对数组使用指针的真正好处体现在按顺序访问数组元素时。如果按照前面提到的方法定义 `valuesPtr`，并且将其设置为指向 `values` 的第一个元素，那么表达式

```
*valuesPtr
```

访问数组 `values` 的第一个整数，即 `values[0]`。要通过变量 `valuesPtr` 引用 `values[3]`，可以将 `valuesPtr` 加 3，然后应用间接寻址运算符：

```
*(valuesPtr + 3)
```

更一般的情况是，可以使用表达式

```
*(valuesPtr + i)
```

来访问 `values[i]` 的数值。

所以，要将 `values[10]` 设置为 27，显然可以如下编写表达式：

```
values[10] = 27;
```

或者使用 `valuesPtr`，写为：

```
*(valuesPtr + 10) = 27;
```

要使 `valuesPtr` 指向数组 `values` 中的第二个元素，可以在 `values[1]` 之前加上地址运算符，并将结果赋给 `valuesPtr`：

```
valuesPtr = &values[1];
```

如果 `valuesPtr` 指向 `values[0]`，通过将 `valuesPtr` 的数值加 1，可以让它指向 `values[1]`：

```
valuesPtr += 1;
```

在 Objective-C 语言中，这是完全合法的表达式，并且可以用于指向任何数据类型的指针。

所以，一般来说，如果 `a` 是元素类型为 `x` 的数组，`px` 是“`x` 指针”类型，并且 `i` 和 `n` 都是变量的整数常量，则表达式

```
px = a;
```

将 `px` 设置为指向 `a` 的第一个元素，而后，表达式

```
*(px + i)
```

指向 `a[i]` 中包含的值。此外，表达式

```
px += n;
```

将 `px` 设为指向数组中比原来指向的元素多了 `n` 个元素的指针，无论数组中包含的元素是什么类型。

假设 `fractsPtr` 指向存储在分数数组中的分数。另外，假设想要将它与数组的下一个元素包含的分数相加，并将结果赋给 `Fraction` 对象 `result`。通过编写下面的表达式即可实现：

```
result = [*fractsPtr add: *(fractsPtr + 1)];
```

处理指针时，运用自增和自减运算符（++和--）非常方便。对指针应用自增运算符相当于将指针加 1，而对指针应用自减运算符则相当于将指针减 1（这里的“1”代表一个单元，或是指针声明指向的数据元素的大小）。所以，如果将 `textPtr` 定义为 `char` 指针，并将其设置为指向 `chars` 数组 `text` 的第一个字符，则表达式

```
++textPtr;
```

会使 `textPtr` 指向数组 `text` 的后一个字符，即 `text[1]`。类似地，表达式

```
--textPtr;
```

会将 `textPtr` 指向数组 `text` 的前一个字符（当然要假设在这条语句执行之前，指针 `textPtr` 并没有指向数组 `text` 的首字符）。

在 Objective-C 语言中，比较两个指针变量的做法是完全合法的。这在比较指向同一数组的两个指针时是非常有用的。比如，你可以测试指针 `valuesPtr`，看它的指向是否超出了包含有 100 个元素的数组的范围，方法是将它与指向数组最后一个元素的指针相比较。所以，如果 `valuesPtr` 超出了数组 `values` 的最后一个元素，表达式

```
valuesPtr > &values[99]
```

的结果将为 `TRUE`（非零），反之，表达式的值为 `FALSE`（零）。根据前面的讨论，可以将上面的表达式相应地改写为：

```
valuesPtr > values + 99
```

这是可能的，因为 `values` 不带有下标，它是指向数组 `values` 首字符的指针（记住，这和写成 `&values[0]` 是一样的）。

代码清单 13-15 举例说明了指向数组的指针。函数 `arraySum` 计算整型数组所包含的元素之和。

代码清单 13-15

```
// 将一个整型数组的元素求和的函数
```

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

```
int arraySum (int array[], int n)
{
```



```

int sum = 0, *ptr;
int *arrayEnd = array + n;

for ( ptr = array; ptr < arrayEnd; ++ptr )
    sum += *ptr;

return (sum);
}

int main (int argc, char *argv[])
{
    @autoreleasepool {
        int arraySum (int array[], int n);
        int values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };

        NSLog (@"The sum is %i", arraySum (values, 10));
    }
    return 0;
}

```

代码清单 13-15 输出

```
The sum is 21
```

在函数 `arraySum` 中，定义了整型指针 `arrayEnd`，并使其指向数组最后一个元素之后的指针。然后，设置 `for` 循环来顺序浏览 `array` 的元素，当循环开始时，`ptr` 的值被设置为 `array` 的首字符。每循环一次，`ptr` 所指向的 `array` 元素的值都被加到 `sum` 中。然后 `for` 循环自动递增 `ptr` 的值，将它设置为指向 `array` 的下一个元素。当 `ptr` 超出了数组范围时，就退出 `for` 循环，并将 `sum` 的值返回给调用者。

1. 是数组还是指针

要将数组传递给函数，只要和前面调用函数 `arraySum` 一样，简单地指定数组名称即可。但是在本节中还提到过，要产生指向数组的指针，只需指定数组名称即可。这暗示着在调用函数 `arraySum` 时，传递给函数的实际上是数组 `values` 的指针。这确切地解释了为什么能够在函数中更改数组元素的值。

但是，如果实际情况是将数组的指针传递给函数，那么为什么函数中的形参不声明为指针呢？换句话说，在函数 `arraySum` 的 `array` 声明中，为什么没有用到以下声明：

```
int *array;
```

在函数中，是不是所有对数组的引用都是通过指针变量实现的？

要回答这些问题，首先必须重申前面提到的关于指针和数组的话题。我们提到过，如果 `valuesPtr` 指向的数据类型和 `values` 数组中包含的元素的类型相同，并且假设将 `valuesPtr` 设为指向 `values` 的首字符，那么表达式 `*(valuesPtr + i)` 与 `values[i]` 完全相同。然后，还可以用表达式 `*(values + i)` 来引用数组 `values` 的第 i 个元素，并且，一般说来，如果 x 是任意类型的数组，则在 Objective-C 语言中，可以将表达式 `x[i]` 等价地表示为 `*(x+i)`。

可以看到，在 Objective-C 语言中，指针和数组是密切相关的，这就是为什么可以在函数 `arraySum` 中将 `array` 声明为“int 数组”类型，或者是“int 指针”类型。在前面的例子中，以上每种声明都可以正常工作。请尝试并查看结果。

如果要使用索引数来引用数组元素，那么要将对应的形参声明为数组。这能更准确地反映该函数对数组的使用情况。类似地，如果要将参数作为指向数组的指针，则要将其声明为指针类型。

2. 字符串指针

指向数组的指针最广泛的应用之一，就是作为字符串指针，原因是符号表达的便利和效率。要说明字符串指针使用起来多方便，这里编写一个名为 `copyString` 的函数，它将一个字符串复制到另一个字符串中。如果使用常规数组索引方式编写这个函数，则可以如下编码：

```
void copyString (char to[], char from[])
{
    int i;

    for ( i = 0; from[i] != '\0'; ++i )
        to[i] = from[i];

    to[i] = '\0';
}
```

`for` 循环在将空字符复制到数组 `to` 之前退出，这就解释了该函数中最后一行代码存在的必要性。

如果使用指针编写 `copyString`，那么不需要使用索引变量 i 。代码清单 13-16 展示了该程序的指针版本。

代码清单 13-16

```

#import <Foundation/Foundation.h>
void copyString (char *to, char *from)
{
    for ( ; *from != '\0'; ++from, ++to )
        *to = *from;

    *to = '\0';
}

int main (int argc, char *argv[])
{
    @autoreleasepool {
        void copyString (char *to, char *from);
        char string1[] = "A string to be copied.";
        char string2[50];

        copyString (string2, string1);
        NSLog (@"%s", string2);

        copyString (string2, "So is this.");
        NSLog (@"%s", string2);
    }
    return 0;
}

```

代码清单 13-16 输出

```

A string to be copied.
So is this.

```

函数 `copyString` 将两个形参 `to` 和 `from` 定义为字符指针，而不是像前一个 `copyString` 那样定义为字符数组。这反映出该函数将如何使用这两个变量。

然后进入 `for` 循环（没有初始条件），它将 `from` 指向的字符串复制到 `to` 指向的字符串中。每循环一次，指针 `from` 和 `to` 都会分别自增 1。这样 `from` 指针就指向源字符串中下一个要复制的字符，而指针 `to` 则指向目标字符串中下一个要存储的字符。

当指针 `from` 指向空字符时，`for` 循环将退出。然后，该函数在目标字符串的结尾处放置一个空字符。

在 `main` 函数中，函数 `copyString` 被调用了两次，第一次用来将 `string1` 的

内容复制到 `string2` 中，第二次用来将字符串常量“`So is this`”复制到 `string2`^②中。

3. 字符串常量和指针

前面的程序中，调用函数

```
copyString (string2, "So is this.");
```

可以正常工作。这一事实意味着向函数传递字符串作为参数时，实际上传递的是指向该字符串的指针。这种情况不仅正确，而且还可以推广到只要在 Objective-C 语言中用到字符串，就会产生指向该字符串的指针。

以上观点可能有些让人混乱，正如第 4 章“数据类型和表达式”提到的：这里我们提到的字符串常量称为 C 样式字符串。它们不是对象。你知道，通过在字符串前面添加标志@（如@“This is okay.”），就可以创建一个常量字符串对象。不能使用一个替代另一个。

所以，如果将 `textPtr` 声明为字符指针，语句如下：

```
char *textPtr;
```

那么表达式

```
textPtr = "A character string.;"
```

则将 `textPtr` 设为指向字符串常量“A character string”的指针。需要注意字符指针和字符数组之间的区别，因为前面显示的赋值类型对于字符数组并不合法。例如，如果将 `text` 定义为 `chars` 数组，语句如下：

```
char text[80];
```

那么就不能编写如下表达式：

```
text = "This is not valid.;"
```

只有在初始化字符数组时，Objective-C 语言才允许对字符数组使用这种赋值方式，语句如下：

```
char text[80] = "This is okay.;"
```

② 请注意，在程序中使用了字符串“A string to be copied.”和“`So is this.`”，这些都不是字符串对象，属于 C 风格的字符串，可以通过字符串的前面是否具有@字符进行区分。这两种类型是不能互换的。如果一个函数需要字符数组作为参数，你可以将 `char` 类型的数组或文字 C 风格字符串传递给它，而不能传递一个字符串对象。

以这种方式初始化 `text` 时，并没有在 `text` 中存储指向字符串 "This is okay." 的指针，而是在相应的 `text` 数组元素中存储实际的字符本身及最后的终止空字符。

如果 `text` 是一个字符指针，则使用以下表达式初始化 `text`：

```
char *text = "This is okay.";
```

将赋给它一个指向字符串 "This is okay" 的指针。

4. 回顾自增和自减运算符

到目前为止，在使用自增或自减运算符时，它们都是表达式中唯一出现的运算符。编写表达式 `++x` 时，你知道这将使变量 `x` 的值加 1。你刚刚学过，如果 `x` 是指向数组的指针，这将使 `x` 指向数组的下一个元素。

自增和自减运算符也可以用于有其他运算符出现的表达式中。在这种情况下，更准确地了解这两个运算符的作用是非常重要的。

使用自增和自减运算符时，总是将它们放在自增或自减变量之前。所以，要使变量 `i` 自增，只需要写：

```
++i;
```

事实上，将自增运算符放在变量的后面同样合法，比如：

```
i++;
```

两种表达式都是合法的，而且都有相同的结果，也就是将 `i` 的数值加 1。第一种情况，即 `++` 放在操作数前，可以更准确地将这种自增运算定义为前缀自增。第二种情况，即 `++` 放在操作数后，可以将这种自增运算定义为后缀自增。

自减运算符也一样。所以，语句

```
--i;
```

实现了 `i` 的前缀自减操作，而语句

```
i--;
```

则实现了 `i` 的后缀自减操作。它们的最终结果相同，即 `i` 的数值减 1。

在更复杂的表达式中使用自增和自减运算符时，运算符前缀和后缀的不同之处就体现出来了。

假设有两个整数 `i` 和 `j`。如果将 `i` 的值设为 0，然后编写语句

```
j = ++i;
```

那么赋给 j 的值是 1，不是你可能认为的 0。使用前缀自增运算符时，变量的值在被用到之前先自增。所以，在前面的语句中， i 的值先从 0 加到 1，然后将它的值赋给 j ，结果与下面两个表达式一样：

```
++i;
j = i;
```

如果在语句中使用后缀自增运算符

```
j = i++;
```

那么 i 在它的数值被赋给 j 之后才自增。所以，如果在执行前面的语句之前， i 的值为 0，那么 j 将被赋值为 0，然后 i 的值再加 1，就像使用下面两个表达式

```
j = i;
++i;
```

另举个例子，如果 i 等于 1，那么语句

```
x = a[--i];
```

的结果是把 $a[0]$ 的值赋给 x ，因为变量 i 的数值在作为 a 的索引之前，值已经减 1。语句

```
x = a[i--];
```

会将 $a[i]$ 的值赋给 x ，因为 i 是在作为 a 的索引之后自减 i 的。

作为描述前缀和后缀自增/自减运算符区别的第三个例子，函数调用语句

```
NSLog(@"%i", ++i);
```

将 i 的值自增 1，然后将该值发送给 NSLog 函数。而调用

```
NSLog(@"%i", i++);
```

则在将 i 的值发送给函数之后才自增 1。所以，如果 i 等于 100，那么第一个 NSLog 语句将显示 101，而第二个 NSLog 将显示 100。在以上任何情况下，执行该语句之后， i 的数值都会等于 101。

在介绍程序之前，再举一个关于此话题的例子，如果 `textPtr` 是一个字符指针，那么表达式

```
* (++textPtr)
```

首先将 `textPtr` 的值加 1，然后获取它所指向的字符，而表达式

```
*(textPtr++)
```

在 `textPtr` 自增之前，先获取它指向的字符。这两种情况都不要求必须存在括号，因为 `*` 和 `++` 运算符的优先级相同，按照从左向右的顺序进行运算。

请回顾代码清单 13-16 的 `copyString` 函数，重写这个函数。让它将自增运算直接合并到赋值语句中。

因为每次执行 `for` 循环中的赋值语句之后，指针 `to` 和 `from` 都会自增 1，所以，它们应该以后缀自增形式合并到赋值语句中。将代码清单 13-16 的 `for` 循环重写为：

```
for ( ; *from != '\0'; )
    *to++ = *from++;
```

循环中的赋值语句将按以下方式执行：先检索 `from` 指向的字符，然后 `from` 将自加 1，从而指向源字符串的下一个字符。引用的字符将被存储到 `to` 指向的位置，然后 `to` 自加 1，从而指向目标字符串的下一个地址。

前面的 `for` 语句看上去并不实用，因为它没有初始表达式，也没有循环表达式。事实上，使用 `while` 循环的形式来表示时，逻辑性可能会更明显。代码清单 13-17 就是这样实现的，该程序给了函数 `copyString` 的新版本。`while` 循环用到了空字符等于数值 0 这一事实，熟练的 Objective-C 编程人员经常这样使用。

代码清单 13-17

```
// 将一个字符串复制到另一个字符串的函数
//      第 2 版本的指针

#import <Foundation/Foundation.h>
void copyString (char *to, char *from)
{
    while ( *from )
        *to++ = *from++;
    *to = '\0';
}

int main (int argc, char *argv[])
{
    @autoreleasepool {
        void copyString (char *to, char *from);
        char string1[] = "A string to be copied.";
        char string2[50];

        copyString (string2, string1);
```

```

    NSLog ("%s", string2);

    copyString (string2, "So is this.");
    NSLog ("%s", string2);
}
return 0;
}

```

代码清单 13-17 输出

```

A string to be copied.
So is this.

```

13.5.4 指针运算

在本章已经学过，可以给指针加减整型数值。此外，可以比较两个指针来查看它们是否相等，或者检测一个指针是大于还是小于另一个指针。指针所允许的其他唯一操作就是相同类型的两个指针的减法。在 Objective-C 语言中，两指针相减的结果是它们之间所包含的元素个数。这样，如果 a 是指向任意类型的元素数组，而 b 是指向同一数组中索引值更大的其他元素，那么表达式 $b-a$ 代表的就是这两个指针之间的元素个数。比如，如果 p 指向数组 x 中的某个元素，那么语句

```
n = p - x;
```

赋给变量 n （假设为整型变量）的是数组 x 中 p 指向元素的索引数。所以，如果通过以下语句将 p 设置为指向 x 中的第 100 个元素：

```
p = &x[99];
```

那么经过前面的减法运算， n 的值应该为 99。

函数指针

函数指针的概念有些高级，但出于完整性的考虑，我们在这里介绍它。处理函数指针时，Objective-C 编译器不但需要知道指向函数的指针变量，而且要知道函数返回值的类型，以及参数的数目和类型。要声明变量 `fnPtr` 为“指向返回 `int` 并且不带参数的函数的指针”，可以编写下面的声明来实现：

```
int (*fnPtr) (void);
```

`*fnPtr` 两侧的括号是必需的；否则，Objective-C 编译器就会认为，在上述

语句中，名为 `fnPtr` 的函数声明返回一个 `int` 指针（因为函数调用运算符 `()` 比指针间接寻址运算符 `*` 的优先级高）。

要使函数指针指向特定函数，可以简单地将函数的名称赋给该指针。因此，如果 `lookup` 是返回 `int` 并且不带参数的函数，则语句

```
fnPtr = lookup;
```

将指向 `lookup` 函数的指针存入函数指针变量 `fnPtr`。编写一个函数名称，并且不带随后的一对括号，类似于编写没有下标的数组名称那样。Objective-C 编译器将自动产生指向特定函数的指针。允许在函数名称之前添加 `&` 标志，但这不是必需的。

如果之前在程序中没有定义函数 `lookup`，则必须在实现前面的赋值运算之前声明该函数。如下语句

```
int lookup (void);
```

在将函数指针赋给变量 `fnPtr` 之前是必需的。

通过对指针变量应用函数调用运算符，同时在括号内列出该函数的所有参数，可以间接引用指针变量引用的函数。比如

```
entry = fnPtr ();
```

调用 `fnPtr` 所指向的函数，并将返回值存储在变量 `entry` 中。

函数指针的一个常见应用是将其作为参数传递给其他函数。Standard Library 在函数 `qsort` 中就是这样用的，该函数实现数组元素的快速排序。它将一个函数指针作为一个参数，只要 `qsort` 需要比较待排序的数组中的两个元素时，它就会调用这个函数。使用这种方式，`qsort` 可以用来对任何类型的数组进行排序，因为数组中任意两个元素的比较都是由用户提供的函数实现的，而不是函数 `qsort` 本身。

在 Foundation 框架中，一些方法使用函数指针作为参数。比如，方法 `sortUsingFunction:context:` 定义在类 `NSMutableArray` 中，只要待排序数组中的两个元素需要比较时，就调用指定的函数。

函数指针的另一个常见应用是建立分派表。你不能将函数本身保存在数组元素中。然而，可以在数组中存储函数指针。这样，就可以创建包含要调用的函数指针的表。比如，可能创建一个表，用于处理用户输入的不同命令。该表

中的每项都可以包含命令名称和指向处理这项特定命令要调用的函数的指针。现在，只要用户输入一条命令，就可以查询该表中的命令，并调用相应的函数去处理它。

13.5.5 指针和内存地址

在结束 Objective-C 中的指针讨论之前，应该指出实际如何实现指针的细节。计算机的内存可以理解为存储单元的顺序集合，计算机内存中的每个单元都有一个相关的编号，称为地址。通常，计算机内存的首地址为 0。在大多数计算机系统中，一个单元就是一字节。

计算机使用内存来存储计算机程序的指令和程序相关变量的值。所以，如果声明变量 `count` 为 `int` 数据，那么在程序执行时，系统会分配内存地址来存储 `count` 的值。比如，这个位置也许是内存地址 `1000FF16`。

幸运的是，你自己不需要考虑指定给变量的特定内存地址，它们是系统自动处理的。然而，掌握每个变量都有唯一的内存地址这个知识，对于理解指针的工作方式会有所帮助。

在 Objective-C 语言中，对变量应用地址运算符，产生的值是变量在计算机内存中的实际存储地址（显然，这就是地址运算符名称的由来）。所以，语句

```
intPtr = &count;
```

向 `intPtr` 分配指定给变量 `count` 的计算机内存地址。这样，如果 `count` 位于地址 `1000FF16`，那么这个语句将数值 `0x1000FF` 赋给 `intPtr`。

对指针变量应用间接寻址运算符，表达式

```
*intPtr
```

的作用就是将包含在指针变量中的数值作为内存地址。然后获取该内存地址中存储的值，并按照指针变量声明的类型进行解释。所以，如果 `intPtr` 是 `int` 指针，那么系统将存储在内存地址中由 `*intPtr` 给出的值解释为整型数据。

13.6 它们不是对象

你现在已经知道了如何定义数组、结构、字符串和联合，以及如何在程序中操纵它们。要记住一个基本原则：它们不是对象。这意味着不能给它们传递

消息，也不能利用它们获得 Foundation 框架提供的内存分配策略之类的最大优势。这是笔者鼓励你跳过本章，以后再学习的原因之一。概括地说，笔者更希望你学习如何使用 Foundation 中将数组和字符串定义为对象的类，而不是使用该语言中内置的类。只应该在真正需要时才使用本章所定义的类型，并且希望你不要遇到这种情况！

13.7 其他语言特性

一些语言特性不能归入其他章节，所以就在这里介绍它们。

13.7.1 复合字面量

复合字面量是包含在括号内的类型名称，之后是一个初始化列表。它创建特定类型的未命名值，它的作用域限于创建它的块；如果它是在所有的程序块之外定义的，则是全局作用域。在后一种情况下，初始化表达式必须都是常量表达式。

下面是一个例子：

```
(struct date) { .month = 7, .day = 2, .year = 2014 }
```

这个表达式产生 `struct date` 类型的结构，并且有一些初始值。可以将它赋值给另一个 `struct date`，语句如下：

```
theDate = (struct date) { .month = 7, .day = 2, .year = 2014 } ;
```

或者，它可以传递给带有 `struct date` 参数的函数或方法，语句如下：

```
setStartDate ((struct date) { .month = 7, .day = 2, .year = 2014 } ) ;
```

还可以定义结构之外的其他类型。例如，如果 `IntPtr` 为 `int *` 类型，那么语句

```
IntPtr = (int [100]) { [0] = 1, [50] = 50, [99] = 99 } ;
```

（它可以出现在程序中的任何位置）将 `IntPtr` 设置为指向包含 100 个整数的数组，数组的前 3 个元素初始化为特定的数值。

如果数组的大小没有说明，则由初始列表来确定。

13.7.2 goto 语句

`goto` 语句的执行导致在程序中产生一个到达特定点的直接分支。要确定程序中分支所在的位置，需要一个标签。标签是根据生成变量名称相同的规则生成的名称，它之后必须紧跟一个冒号。标签直接放在分支语句之前，而且必须与 `goto` 语句处在同一个函数或方法中。

比如，语句

```
goto out_of_data;
```

使程序立即跳到标签 `out_of_data` 开头的语句分支。这个标签可以放在函数或方法中的任何地方，无论是在 `goto` 语句之前还是之后，并且可以这样使用语句：

```
out_of_data: NSLog (@"Unexpected end of data.");  
...
```

懒惰的程序员经常滥用 `goto` 语句来转移到代码的其他部分。`goto` 语句打断了程序的常规顺序流程，结果就会造成程序难以读懂。在程序中使用很多 `goto` 语句会使程序变得难以解释。由于这个原因，使用 `goto` 语句并不认为是好的编程方式。

13.7.3 空语句

Objective-C 语言允许将孤立的分号放在可以出现常规语句的地方。这种称为空语句的语句不做任何操作。这看上去没有什么用，但是程序员经常将它用在 `while`、`for` 和 `do` 语句中。比如，下面语句的作用是将所有从标准输入（默认为终端）读入的字符存储到指针 `text` 指向的字符数组，直到出现换行字符为止。它使用库函数 `getchar` 每次从标准输入读入并返回单个字符：

```
while ( (*text++ = getchar ()) != '\n' )  
;
```

所有的操作都是在 `while` 语句的循环条件部分中实现的，需要有空语句是因为编译器认为循环语句后的下一条语句是循环体。如果没有空语句，无论下一条语句是什么，都会被编译器认为是循环体。

13.7.4 逗号运算符

优先级列表的底层是逗号运算符。在第 5 章中，我们提到过在 `for` 语句中

可以通过使用逗号将每条语句分开，可以在任一部分包含多条表达式。比如，开始的 `for` 语句

```
for ( i = 0, j = 100; i != 10; ++i, j -= 10 )
...
```

在循环开始前将 i 初始化为 0， j 初始化为 100，然后，每次执行循环体之后， i 的值自增 1， j 的值自减 10。

因为在 Objective-C 语言中，所有的运算符都产生一个值，所以逗号运算符的值是最右边的表达式值。

13.7.5 sizeof 运算符

尽管不应该假设程序中数据类型的大小，但是有的时候需要知道这些信息。使用库函数（如 `malloc`）来实现动态内存分配，或对文件读出或写入数据时，可能需要这些信息。Objective-C 语言提供了 `sizeof` 运算符，它可以用来确定数据类型或对象的大小。`sizeof` 运算符返回的是指定项的字节大小。`sizeof` 运算符的参数可以是变量、数组名称、基本数据类型名称、对象、派生数据类型名称或表达式。比如，

```
sizeof (int)
```

给出了存储整型数据所需的字节数。在笔者的 MacBook Air 计算机上，上述符号会产生结果 4（或 32 位）。如果 x 声明为包含 100 个 `int` 数据的数组，则表达式

```
sizeof (x)
```

将给出存储 x 中的 100 个整数所需要的存储空间。

假设 `myFract` 是一个 `Fraction` 对象，它包含两个 `int` 实例变量（分子和分母），那么表达式

```
sizeof (myFract)
```

在任何使用 4 字节表示指针的系统中都会产生值 4。事实上，这是 `sizeof` 对任何对象产生的值，因为这里询问的是指向对象数据的指针大小。要获得实际存储 `Fraction` 对象实例的数据结构大小，可以编写以下语句：

```
sizeof (*myFract)
```

在笔者的 MacBook Air 计算机上，上述表达式给出了值 12，即分子和分母

分别用 4 字节，加上另外的 4 字节存储继承来的 `isa` 成员，本章后面“工作原理”一节中将提到这个成员。

表达式

```
sizeof (struct data_entry)
```

的值将是存储 `data_entry` 结构所需的空间总数。如果将 `data` 定义为包含 `struct data_entry` 元素的数组，则表达式

```
sizeof (data) / sizeof (struct data_entry)
```

将给出包含在 `data` (`data` 必须是前面定义的，并且不是形参，也不是外部引用的数组) 中的元素个数。表达式

```
sizeof (data) / sizeof (data[0])
```

也产生同样的结果。

尽可能地使用 `sizeof` 运算符来避免在程序中计算和硬编码数据大小。

13.7.6 命令行参数

很多时候，程序都要求用户在终端输入少量信息。这些信息可能包含一些数字，它们表示你想要计算的三角数或者是想要在字典中查询的单词。

除了让程序请求用户输入相关信息外，还可以在程序执行时给该程序提供信息。这种能力是由命令行参数提供的。

我们提到过 `main` 函数唯一的与众不同之处在于它的名称很特别：它指明了程序开始执行的位置。事实上，`main` 函数是在程序开始执行时，由运行时系统调用，就像在自己的程序中调用函数一样。当 `main` 执行完毕时，控制权返回给运行的系统，这样系统便知道程序已经执行完毕了。

当运行时系统调用 `main` 函数时，系统向该函数传递两个参数。第一个参数按照规定称为 `argc` (`argument count` 的简写)，是一个整型值，它指明了从命令行输入的参数个数。第二个传递给 `main` 的参数是一个字符指针数组，按照规定称为 `argv` (`argument vector` 的简写)。另外，这个数组中包含 `argc+1` 个字符指针。该数组的第一个元素是执行程序的名称指针，如果你的系统中没有程序名称，则是空串。数组的其他项指向由启动程序执行的命令行所指定的值。数组 `argv` 中的最后一个指针 `argv[argc]` 规定为空。

要访问命令行参数，必须将 `main` 函数适当地声明为带有两个参数。使用本书所有程序中用到的惯例声明就可以了，语句如下：

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

记住，`argv` 的声明定义了一个包含“char 指针”类型元素的数组。命令行参数的一个实际用途是，假设你开发了一个程序，它在字典中查询某个单词并显示单词的含义，那么可以使用命令行参数。这样在执行程序的同时就可以指定想要查找含义的单词，如以下命令：

```
lookup aerie
```

这就不必提示用户键入一个单词，因为单词已经从命令行输入了。

如果执行上面的命令，那么系统自动向 `main` 函数传递 `argv[1]` 中字符串“aerie”的指针。你可能会想起，`argv[0]` 包含的是指向程序名称的指针，在这个例子中是“lookup”。

`main` 函数可能显示如下：

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

int main (int argc, char *argv[])
{
    struct entry dictionary[100] =
    { { "aardvark", "a burrowing African mammal" },
      { "abyss", "a bottomless pit" },
      { "acumen", "mentally sharp; keen" },
      { "addle", "to become confused" },
      { "aerie", "a high nest" },
      { "affix", "to append; attach" },
      { "agar", "a jelly made from seaweed" },
      { "ahoy", "a nautical call of greeting" },
      { "aigrette", "an ornamental cluster of feathers" },
      { "ajar", "partially opened" } };

    int entries = 10;
    int entryNumber;
    int lookup (struct entry dictionary [], char search[],
                int entries);

    if ( argc != 2 )
    {
```

```

    NSLog (@@"No word typed on the command line.");
    return (1);
}

entryNumber = lookup (dictionary, argv[1], entries);

if ( entryNumber != -1 )
    NSLog (@@"%s", dictionary[entryNumber].definition);
else
    NSLog (@@"Sorry, %s is not in my dictionary.", argv[1]);

return (0);
}

```

执行程序时，`main` 函数测试以确保在程序名称之后输入一个单词。如果没有，或输入了多个单词，那么 `argc` 的数值将不等于 2。在这种情况下，程序将写明出错消息并终止运行，返回终止状态 1。

如果 `argc` 等于 2，将调用函数 `lookup`，在字典中寻找 `argv[1]` 指向的单词。如果找到了该单词，则显示它的定义。

需要记住命令行参数总是存储为字符串的。所以，带有命令行参数 2 和 16 的 `power` 程序：

```
power 2 16
```

的执行结果是将字符串"2"的指针保存在 `argv[1]` 中，将字符串"16"的指针保存在 `argv[2]`。如果程序将参数解释为数字（就像我们猜想程序 `power` 一样），程序本身必须转换它。程序库中有一些函数可以用来实现这种转换，如 `scanf`、`atof`、`atoi`、`strtod` 和 `strtol`。在第二部分中，将学习如何使用名为 `NSProcessInfo` 的类来以字符串对象而不是 C 字符串的形式访问命令行参数。

13.8 工作原理

如果没有尝试将一些事物联系在一起就结束本章，那么我们就没有尽到责任。因为 Objective-C 语言以 C 语言为基础，所以值得讨论两者的关系。下面是一些可以忽略的实现细节，或可以用来更好地理解系统工作方式的细节，就像学习指针实际上是内存地址可以帮助你更好地理解指针一样。我们没有涉及太多详细的内容，只是阐明关于 Objective-C 语言和 C 语言联系的 4 个事实。

13.8.1 事实#1：实例变量存储在结构中

定义一个新类和它的实例变量时，这些实例变量实际上存放在一个结构中。这说明了可以如何处理对象，对象实际上是结构，结构中的成员是实例变量。所以继承的实例变量加上你在类中添加的变量就组成了一个结构。使用 `alloc` 分配新对象时，系统预留了足够的空间来存储这些结构。

结构中继承的成员（从根对象中获得的）之一是名为 `isa` 的保护成员，它确定对象所属的类。因为它是结构的一部分（因此，也是对象的一部分），所以由对象携带。这样，运行时系统只需通过查看 `isa` 成员，就可以确定对象的类（即使将其赋给通用的 `id` 对象变量）。

通过将成员定义为 `@public`，可以获得对象结构成员的直接存储权限（详见第 10 章）。举个例子，如果对 `Fraction` 类中的 `numerator` 和 `denominator` 成员进行了这项操作，那么可以在程序中编写如下表达式

```
myFract->numerator
```

来直接访问 `Fraction` 对象 `myFract` 的 `numerator` 成员。但是，我们强烈建议不要这么做。在第 10 章中提到过，它违反了数据封装的特性。

13.8.2 事实#2：对象变量实际上是指针

定义 `Fraction` 之类的对象变量时，如

```
Fraction *myFract;
```

事实上是定义了一个名为 `myFract` 的指针变量。这个变量定义为指向 `Fraction` 类型的数据，即你的类名称。使用

```
myFract = [Fraction alloc];
```

来创建 `Fraction` 的新实例时，是在为 `Fraction` 对象的新实例分配存储内存（即存放结构的内存），然后使用结构的指针，并将指针变量 `myFract` 存储在其中。

将对象变量赋给另一个对象变量时，语句

```
myFract2 = myFract1;
```

只是简单地复制了指针。这两个变量最后都指向存储在内存中的同一结构。因此，改变 `myFract2` 引用的（即指向的）一个成员，将更改 `myFract1` 引用的同一个实例变量（即结构成员）。

13.8.3 事实#3: 方法是函数, 而消息表达式是函数调用

方法实际上是函数。调用方法时, 是在调用与接收者类相关的函数。传递给函数的参数是接收者 (self) 和方法的参数。所以, 无论是函数还是方法, 关于传递参数给函数、返回值及自动和静态变量的规则都是一样的。Objective-C 编译器通过类名称和方法名称的组合为每个函数产生一个唯一的名称。

13.8.4 事实#4: id 类型是通用指针类型

因为通过指针 (也就是内存地址) 来引用对象, 所以可以自由地将它们在 id 变量之间来回赋值。因此, 返回 id 类型值的方法只是返回指向内存中某对象的指针。然后可以将该值赋给任何对象变量。因为无论在哪里, 对象总是携带它的 isa 成员, 所以, 即使将它存储在 id 类型的通用对象变量中, 也总是可以确定它的类。

13.9 练习

1. 编写一个函数, 计算包含 10 个浮点数的数组的平均值并返回结果。
2. Fraction 类中的方法 reduce 用于找出分子和分母的最大公约数来简约分数。修改这个方法, 使其使用代码清单 13-5 中的函数 gcd。你认为应该在什么地方定义该函数呢? 将函数定为 static 有什么好处? 你认为哪种方式更好, 是使用函数 gcd 还是像以前一样将代码直接合并到方法中呢? 为什么?
3. 可以使用一种名为 Sieve of Erastosthenes 的算法产生素数。这个过程的算法如下。编写一个程序来实现这个算法, 假设程序找出 150 之前的所有质数, 比较本文中其他计算质数的算法, 如何评价这个算法?
 - 步骤 1: 定义整型数组 P, 将所有的元素 P_i 置为 0、 $2 \leq i \leq n$ 。
 - 步骤 2: 将 i 设为 2。
 - 步骤 3: 如果 $i > n$, 算法终止。
 - 步骤 4: 如果 P_i 等于 0, 则 i 是质数。
 - 步骤 5: 对于所有满足 $i*j \leq n$ 的正整数 j , 将 P_{i*j} 设为 1。

步骤 6: i 加 1, 并且回到步骤 3。

4. 编写一个函数, 将所有传递给它的数组中的 `Fraction` 相加, 并以 `Fraction` 类型返回结果。

5. 为名为 `Date` 的 `struct date` 编写定义 `typedef`, 它允许在程序中进行如下声明

```
Date todaysDate;
```

6. 在文中提到过, 定义 `Date` 类而不是 `date` 结构更符合面向对象的编程思想。定义这样的类, 并定义适当的 `setter` 和 `getter` 方法。同时, 添加一个名为 `dateUpdate` 的方法来返回参数之后的日期。

能看出将 `Date` 定义为类而不是结构的好处吗? 能看出这样做的缺点吗?

7. 给出下列定义:

```
char *message = "Programming in Objective-C is fun";
char message2[] = "You said it";
int x = 100;
```

确定下列语句集中的 `NSLog` 语句是否都合法, 产生的输出是否和其他语句一样。

```
/** set 1 */
NSLog(@"Programming in Objective-C is fun");
NSLog(@"%s", "Programming in Objective-C is fun");
NSLog(@"%s", message);

/** set 2 */
NSLog(@"You said it");
NSLog(@"%s", message2);
NSLog(@"%s", &message2[0]);

/** set 3 */
NSLog(@"said it");
NSLog(@"%s", message2 + 4);
NSLog(@"%s", &message2[4]);
```

8. 编写程序, 该程序在终端输出所有的命令行参数, 每行一个。注意引号中包含空格字符的封装参数有什么作用。
9. 下面哪组语句会输出 `This is a test`? 并说明原因。

```
NSLog(@"This is a test");
```

```
NSLog ("This is a test");

NSLog ("%s", "This is a test");
NSLog ("%s", @"This is a test");

NSLog ("%s", "This is a test");
NSLog ("%s", @"This is a test");

NSLog ("%@", @"This is a test");
NSLog ("%@", "This is a test");
```

10. 以区块的形式重新编写代码清单 13-14 中的 `exchange` 函数并进行测试。

Foundation 框架简介

框架是由许多类、方法、函数和文档按照一定的逻辑组织起来的集合，以使研发程序变得更容易。在 OS X 系统下有 100 多个框架，这些框架可以用来开发应用程序，处理 Mac 的 Address Book 结构、刻录 CD、播放 DVD、使用 QuickTime 播放电影、歌曲，等等。

为所有的程序开发奠定基础的框架称为 Foundation 框架。该框架是本书第二部分的主题，它允许使用一些基本对象，如数字和字符串，以及一些对象集合，如数组、字典和集合。其他功能包括处理日期和时间、自动化的内存管理、处理基础文件系统、存储（或归档）对象、处理几何数据结构（如点和长方形）。

Application Kit 框架包含广泛的类和方法，它们用来开发交互式图形应用程序，使得开发文本、菜单、工具栏、表、文档、剪贴板和窗口之类的过程变得十分简便。在 OS X 系统中，术语 Cocoa 总的来说指的是 Foundation 框架、Application Kit 框架和名为 Core Data 的第三方框架。术语 Cocoa Touch 是指 Foundation、Core Data 和 UIKit 框架。本书的第三部分“Cocoa、Cocoa Touch 和 iOS SDK”会介绍有关这一主题的详细内容。

14.1 Foundation 文档

应该利用存储在系统中（如果选择下载本地副本）的 Foundation 框架文档，这些文档在 Apple 网站上也有提供。大多数文档为 HTML 格式文件，可以通过浏览器查看，同时也提供了 Acrobat pdf 格式的文件。这个文档包含 Foundation 的所有类及其实现的所有方法和函数的描述。可以将 Foundation 文档的 URL 作为书签添加到浏览器的收藏夹中，这样更方便查找 Foundation 类的信息。

如果正在使用 Xcode 开发程序，可以通过多种方式轻松访问文档。访问 Help 菜单中的 Documentation and API Reference，可以访问到文档窗口的主界面。通过这个窗口，可以轻松搜索和访问存储在计算机本机中或者在线的文档。图 14.1 显示了在 Xcode 文档窗口中搜索字符串“NSString”的结果。

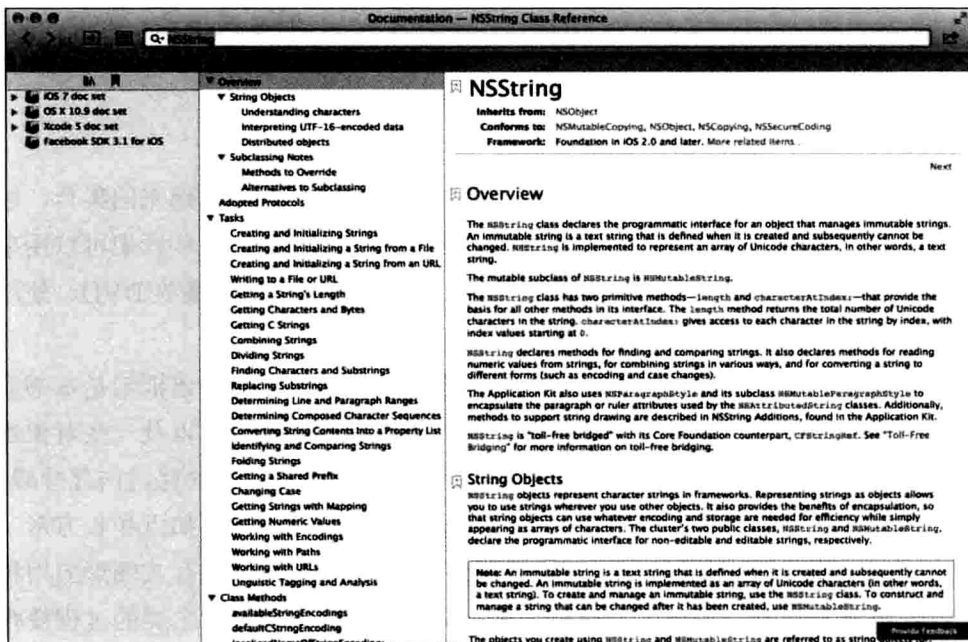


图 14.1 使用 Xcode 访问参考文档

如果你正在 Xcode 中编辑文件，并且想要快速访问某个特定的头文件、方法或类的文档，可以将光标放在需要搜索的类、方法或变量上，按住 Option 键，同时单击，你会看到所选择内容的快速概要。如图 14.2 所示，当鼠标光标位于文本 NSString 之上，按住 Option 键并单击时，就显示出概要面板。

单击帮助面板中的相关链接可以获得 NSString.h 的头文件以及类的参考文档。这确实是一个既好用又方便的工具，应该经常使用！

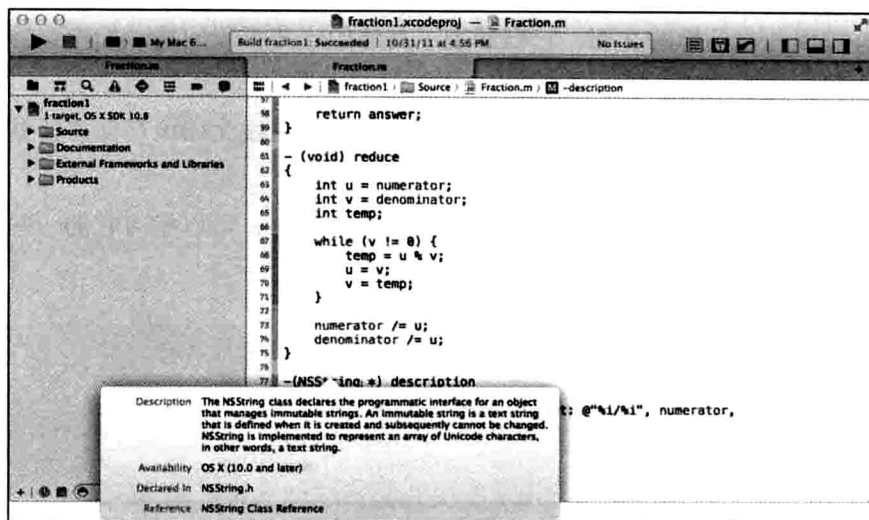


图 14.2 NSString 快速参考

也可以让快速帮助面板一直显示，在程序中输入或选中某项时，如果选择 View、Utilities、Quick Help，面板中的内容会自动更新。图 14.3 中显示的布局窗口，Quick Help 菜单默认情况下会出现在最右边的窗格中。

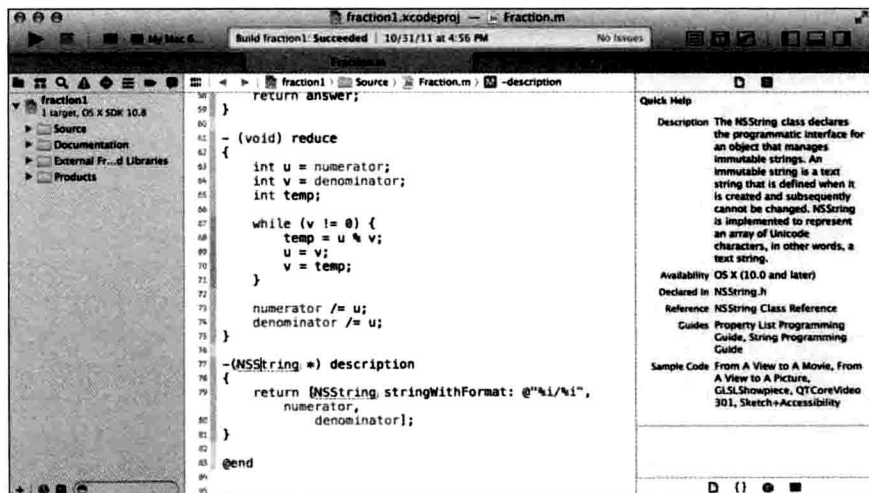


图 14.3 在视图窗格中显示快速帮助

还可以在线访问 MAC OS X 在线参考库，地址是 <http://developer.apple.com/library/mac/navigation/index.html>，浏览自己需要的参考手册。在这个站点中，

还能够发现各式各样的文档，它们涵盖了具体的一些编程问题，如内存管理、字符串和文件管理等。

除非你在 Xcode 中订阅了特定的文档集，否则在线文档比存储在计算机硬盘中的文档更新。

以上对 Foundation 框架进行了简要的介绍。现在，可以学习它的一些类，并在应用程序中使用它们。

数字、字符串和集合

本章讲解如何使用 Foundation 框架提供的一些基本对象。这些基本对象包括数字、字符串和集合，以及数组、字典和集合形式使用的成组对象。

Foundation 框架包括大量的类、方法和函数。在 OS X 下，大约有 125 个可用的头文件，可以通过一种简便的形式进行导入，语句如下：

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

实际上，因为 Foundation.h 文件导入了 Foundation 所有的头文件，所以不必担心是否导入了正确的头文件。Xcode 会自动将这个头文件插入程序，正如你在书中见到的示例。

使用这条语句会明显增加程序的编译时间。然而，使用预编译的头文件可以避免额外的时间开销。预编译的头文件是编译器预先处理过的文件。通常，所有的 Xcode 项目都会受益于预编译的头文件。此外，在 Xcode5 中，名为“模块”的新功能可以帮助加快编译，以及帮助避免应用程序中使用不同的库出现的命名冲突。

15.1 数字对象

到目前为止，我们讨论了所有的数字数据类型，int 型、float 型和 long 型都是 Objective-C 语言中的基本数据类型，它们都不是对象。也就是说，不能够向它们发送消息。然而，有时需要将这些值作为对象使用。比如，使用 Foundation 的 NSArray 对象创建一个数组，它要求存储的值必须是对象，因此，不能将任何基本数据类型直接存储到数组中。如果需要存储基本数据类型（包括 char 数据类型），可以使用 NSNumber 类，它会依据这些数据的类型创建对象（参见

代码清单 15-1)。

代码清单 15-1

```
// 使用 Number

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSNumber      *myNumber, *floatNumber, *intNumber;
        NSInteger      myInt;

        // integer 型值

        intNumber = [NSNumber numberWithInt: 100];
        myInt = [intNumber integerValue];
        NSLog ("%li", (long) myInt);

        // long 型值

        myNumber = [NSNumber numberWithLong: 0xabcdef];
        NSLog ("%lx", [myNumber longValue]);

        // char 型值

        myNumber = [NSNumber numberWithChar: 'X'];
        NSLog ("%c", [myNumber charValue]);

        // float 型值

        floatNumber = [NSNumber numberWithFloat: 100.00];
        NSLog ("%g", [floatNumber floatValue]);

        // double

        myNumber = [NSNumber numberWithDouble: 12345e+15];
        NSLog ("%lg", [myNumber doubleValue]);

        // 发生错误

        NSLog ("%li", (long) [myNumber integerValue]);

        // 验证两个 Number 是否相等

        if ([intNumber isEqualToNumber: floatNumber] == YES)
            NSLog ("%Numbers are equal");
```

```

else
    NSLog(@"Numbers are not equal");

// 验证一个 Number 是否小于、等于或大于另一个 Number

if ([intNumber compare: myNumber] == NSOrderedAscending)
    NSLog(@"First number is less than second");
}
return 0;
}

```

代码清单 15-1 输出

```

100
abcdef
X
100
1.2345e+19
-9223372036854775808
0
Numbers are equal
First number is less than second

```

`NSNumber` 类包含多个方法，可以使用初始值创建 `NSNumber` 对象。例如，语句

```
intNumber = [NSNumber numberWithInt: 100];
```

表示创建一个值为 100 的整数对象。

从 `NSNumber` 对象获得的值必须和存储在对象中的值类型一致。因此，之后的语句

```
myInt = [intNumber integerValue];
```

获取存储在 `intNumber` 中的整型值，将它存储在 `NSInteger` 变量 `myInt` 中。注意，`NSInteger` 不是一个对象，而是基本数据类型的 `typedef`。它实际上是 64 位的 `long` 或者 32 位的 `int`。`NSUInteger` 也是类似的 `typedef`，在程序中表示无符号整数。

`NSLog` 在调用过程中，我们将 `NSInteger` 变量 `myInt` 转换为 `long`，并使用格式化字符 `%li`，目的是确保值能够传递而且正确显示，即使程序编译后是 32 位架构的。

对于每种基本数据类型，类方法都能为它创建一个 `NSNumber` 对象，并设置为指定的值。这些方法以 `numberWith` 开头，紧接数据的类型，如

numberWithLong:、numberWithFloat:等。此外,可以使用实例方法将以前创建的 NSNumber 对象设置为指定的值。这些都是以 initWith 开头的,如 initWithLong: 和 initWithFloat:。

表 15.1 列出了为 NSNumber 对象设值的类和实例方法,以及获取这些值相应的实例方法。

注意

在 Xcode 4.2 之前的版本,使用哪个版本的方法是很重要的,有一类方法创建的对象是会自动释放的,然而使用 alloc 版本创建的对象需要在使用完后自己负责释放(在 iOS 中)。随着 Objective-C 中引入了自动引用计数 (ARC),已经能够自动处理内存管理,就不再偏向于使用某一类方法。

表 15.1 NSNumber 的创建方法和检索方法

创建和初始化方法	初始化实例方法	检索实例方法
numberWithChar:	initWithChar:	charValue
numberWithUnsignedChar:	initWithUnsignedChar:	unsignedCharValue
numberWithShort:	initWithShort:	shortValue
numberWithUnsignedShort:	initWithUnsignedShort:	unsignedShortValue
numberWithInteger:	initWithInteger:	integerValue
numberWithUnsignedInteger:	initWithUnsignedInteger:	unsignedIntegerValue
numberWithInt:	initWithInt:	intValue
numberWithUnsignedInt:	initWithUnsignedInt:	unsignedIntValue
numberWithLong:	initWithLong:	longValue
numberWithUnsignedLong:	initWithUnsignedLong:	unsignedLongValue
numberWithLongLong:	initWithLongLong:	longlongValue
numberWithUnsignedLongLong:	initWithUnsignedLongLong:	unsignedLongLongValue
numberWithFloat:	initWithFloat:	floatValue
numberWithDouble:	initWithDouble:	doubleValue
numberWithBool:	initWithBool:	boolValue

回到代码清单 15-1,这个程序使用类方法创建了 long、char、float 和 double 型 NSNumber 对象。注意,使用程序语句

```
myNumber = [NSNumber numberWithInt: 12345e+15];
```

创建 double 对象后将出现什么情况? 尝试(不正确地)使用下列语句获取并显

示它的值:

```
NSLog(@"%li", (long) [myNumber integerValue]);
```

将得到以下输出:

```
0
```

而且系统也没有提示出错消息。一般来说, 你需要确保使用正确的方式获取对象的值, 如果在 `NSNumber` 对象中存储了一个值, 那么也需要用一致的方式去获取。

在 `if` 语句中, 消息表达式

```
[intNumber isEqualToNumber: floatNumber]
```

使用了 `isEqualToNumber:` 方法比较两个 `NSNumber` 对象的数值。程序会返回一个 `BOOL` 值, 查看这两个值是否相等。

可以使用 `compare:` 方法测试一个值是否在数值上小于、等于或大于另一个值。当消息表达式

```
[intNumber compare: myNumber]
```

中 `intNumber` 的值小于 `myNumber` 的值时, 返回 `NSOrderedAscending`; 如果这两个数刚好相等, 则返回 `NSOrderedSame`; 如果第一个值大于第二个值, 则返回 `NSOrderedDescending`。在引用 `Foundation.h` 时已经包含了 `NSObject.h` 头文件。

应该注意, 不能修改前面创建的 `NSNumber` 对象的值。例如,

```
NSNumber *myNumber = [[NSNumber alloc] initWithInt: 50];
...
[myNumber initWithInt: 1000];
```

并不能正常运行。

程序执行时, 最后一条语句会引起程序崩溃。所有的数字对象都必须都是新创建的, 这意味着必须对 `NSNumber` 类调用表 15.1 中第一列的某个方法, 或者对 `alloc` 方法的结果调用第二列的某个方法, 如前一例中第一行语句。

方法 `numberWithInt:` 和 `numberWithInteger:` 使用有些差别, 遵循以下一些规则:

(1) 如果使用 `numberWithInt:` 方法创建一个整型数, 需要使用 `intValue` 获取

它的值，使用%i 作为格式化字符串显示它的值。

(2) 如果使用 `numberWithInteger:` 方法创建一个整型数，需要使用 `integerValue`，也可以转换成 `long` 显示或者使用 `stringWithFormat:` 将它格式化成字符串。使用%i 作为格式化字符串。

方法 `numberWithUnsignedInt:` 和 `numberWithUnsignedInteger:` 的使用类似。

最近对 Objective-C 语言的扩展是允许通过@表达式创建数字对象的。查看代码清单 15.2 如何使用此语法。

代码清单 15-2

```
// 使用数字对象
#import <Foundation/Foundation.h>
int main (int argc, char * argv[]) {
    @autoreleasepool {
        NSNumber *myNumber, *floatNumber, *intNumber;
        NSInteger myInt;
        // 整型 integer
        intNumber = @100;
        myInt = [intNumber integerValue];
        NSLog(@"%li", (long) myInt);
        // 长整型 long value
        myNumber = @0xabcdefL;
        NSLog(@"%lx", [myNumber longValue]);
        myNumber = @'X';
        NSLog(@"%c", [myNumber charValue]);
        // 浮点型 float value
        floatNumber = @100.0f;
        NSLog(@"%g", [floatNumber floatValue]);
    }
    return 0;
}
```

代码清单 15-2 输出

```
100
abcdef
X
100
```

代码清单 15-2 展示了如何从一个简单对象创建数字对象，但是你也可以编写如下一个表达式：

```
NSNumber *center = @((start + end) / 2.0);
```

需要注意的是，如果在@之后的值是一个表达式或者变量需要使用括号。`NSNumber` 对象在本章其他程序中还会出现。进入下一节之前，可以查看一下 `NSDecimalNumber` 类的相关文档。它是 `NSNumber` 的子类，在对象层面提供了一些数字的四则运算方法。

15.2 字符串对象

在前面的程序中已经遇到过字符串对象只需要使用一对双引号将一组字符串引起来，语句

```
@ "Programming is fun"
```

就是使用 Objective-C 语言创建的一个字符串对象。`Foundation` 框架支持一个名为 `NSString` 的类，用于处理字符串对象。然而 C 样式的字符串由 `char` 字符组成，`NSString` 对象由 `unichar` 字符组成。`unichar` 字符是符合 Unicode 标准的多字节字符。这样，就可以处理包含数百万字符的字符集。幸运的是，不必担心字符串的内部表示，因为 `NSString` 类已经自动处理了。^①使用这个类的方法更容易开发出具有本地化的应用程序，能够在世界上不同的语言环境下使用。

要使用 Objective-C 语言创建一个常量字符串对象，需要在字符串开头放置一个@字符。于是，表达式

```
@ "Programming is fun"
```

创建了一个常量字符串对象。特殊情况下，它属于 `NSString` 类的常量字符串对象。`NSString` 类是字符串对象 `NSString` 类的子类。

15.2.1 NSLog 函数

在接下来的代码清单 15-3 中，说明了如何定义 `NSString` 对象，并赋给它一个初始值，同时也说明如何使用格式字符%@来显示 `NSString` 对象。

代码清单 15-3

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

① 目前，`unichar` 字符占用 16 位，但 Unicode 标准规定的字符比这个尺寸大。所以在将来，`unichar` 字符可能会大于 16 位。所以，不要对 Unicode 字符的大小进行假设。


```
int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSString *str = @"Programming is fun";

        NSLog ("%@", str);
    }
    return 0;
}
```

代码清单 15-3 输出

```
Programming is fun
```

常量字符串对象 `Programming is fun` 被赋值给 `NSString` 变量 `str`。然后使用 `NSLog` 显示它的值。

`NSLog` 格式字符 `%@` 不仅可以显示 `NSString` 对象, 而且可以显示其他对象。例如, 以下语句:

```
NSNumber *intNumber = @100;
// the NSLog call
NSLog ("%@", intNumber);
```

输出结果为:

```
100
```

15.2.2 description 方法

你也可以使用格式化字符 `%@` 显示数组、字典和集合的全部内容。事实上, 通过覆盖继承的 `description` 方法, 还可使用这些格式字符显示你自己的类对象。如果不覆盖方法, `NSLog` 仅仅显示类名和该对象在内存中的地址, 这是从 `NSObject` 类继承的 `description` 方法的默认实现。

以下是 `description` 方法的一个例子, 把它添加到 `Fraction` 类的实现部分用于 `Fraction` 对象的格式化。`NSString` 的 `stringWithFormat:` 方法在使用上与 `NSLog` 相似。不同的是这个方法返回格式化的字符串, 而不是写入到控制台。

```
-(NSString *) description
{
    return [NSString stringWithFormat:@"%i/%i", numerator, denominator];
}
```

注意

像 `stringWithFormat:` 这样的方法允许提供多个参数（`stringWithFormat:` 需要的参数是格式化字符串和需要格式化的数据）。这些参数使用一系列逗号分隔提供给方法。

在 `Fraction` 类中定义一个方法（对两个 `Fraction` 对象 `f1` 和 `f2` 进行运算），编写以下语句：

```
sum = [f1 add: f2];
NSLog(@"The sum of %@ and %@ is %@", f1, f2, sum);
```

得到一个单行输出

```
The sum of 1/2 and 1/4 is 3/4
```

将自己的 `description` 方法添加到类中可以作为一种不错的调试工具，它能够使对象的显示更有意义。

15.2.3 可变对象与不可变对象

编写如下语句：

```
@ "Programming is fun"
```

创建字符串对象时，会创建一个内容不可更改的对象，称为不可变对象。可以使用 `NSString` 类处理不可变字符串。你经常需要处理字符串并更改字符串中的字符。例如，可能希望从字符串中删除一些字符，或对字符串执行搜索替换操作。这种类型的字符串是使用 `NSMutableString` 类处理的。

代码清单 15-3 说明了在程序中处理不可变字符串的基本方式。

代码清单 15-4

// 基本的字符串操作

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSString *str1 = @"This is string A";
        NSString *str2 = @"This is string B";
        NSString *res;
        NSComparisonResult compareResult;
```

```

// 计算字符串中的字符

NSLog(@"Length of str1: %lu", [str1 length]);

// 将一个字符串复制到另一个字符串

res = [NSString stringWithString: str1];
NSLog(@"copy: %@", res);

// 将一个字符串复制到另一个字符串的末尾

str2 = [str1 stringByAppendingString: str2];
NSLog(@"Concatentation: %@", str2);

// 验证两个字符串是否相等

if ([str1 isEqualToString: res] == YES)
    NSLog(@"str1 == res");
else
    NSLog(@"str1 != res");

// 验证一个字符串是否小于、等于或大于另一个字符串

compareResult = [str1 compare: str2];

if (compareResult == NSOrderedAscending)
    NSLog(@"str1 < str2");
else if (compareResult == NSOrderedSame)
    NSLog(@"str1 == str2");
else // 必须是 NSOrderedDescending
    NSLog(@"str1 > str2");

// 将字符串转换为大写

res = [str1 uppercaseString];
NSLog(@"Uppercase conversion: %s", [res UTF8String]);

// 将字符串转换为小写

res = [str1 lowercaseString];
NSLog(@"Lowercase conversion: %@", res);

NSLog(@"Original string: %@", str1);
}
return 0;
}

```

代码清单 15-4 输出

```

Length of str1: 16
Copy: This is string A
Concatentation: This is string AThis is string B
str1 == res
str1 < str2
Uppercase conversion: THIS IS STRING A
Lowercase conversion: this is string a
Original string: This is string A

```

代码清单 15-3 首先定义了 3 个不可变的 NSString 对象：str1、str2 和 res。前两个初始化为常量字符串对象。声明

```
NSComparisonResult compareResult;
```

用于保存后面执行的字符串比较操作的结果。

length 方法可以用来对字符串中的字符进行计数。它将返回 NSUInteger 类型的无符号整数值。字符串输出

```
@ "This is string A"
```

包含 16 个字符。

语句

```
res = [NSString stringWithString: str1];
```

说明如何使用另一个字符串的内容生成一个新的字符串。NSString 对象被赋值给 res，然后显示出来以验证结果。在这里进行的实际上是字符串内容的复制，而不是对内存中同一字符串的引用。即 str1 和 res 指向两个不同的字符串对象，这与简单地执行以下赋值操作是不同的：

```
res = str1;
```

在前面讨论过，这只是创建了内存中同一对象的另一个引用。

使用 stringByAppendingString: 方法可以连接两个字符串。所以，表达式

```
[str1 stringByAppendingString: str2]
```

创建了一个新的字符串对象，这个对象由 str1 和 str2 的字符拼接而成，然后返回结果。这项操作没有改变原字符串对象 str1 和 str2（它们不能更改，因为都是不可变字符串对象）。

然后在程序中使用 isEqualToString: 方法检测两个字符串是否相等，即是否

包含相同的字符。如果需要确定两个字符串的顺序，例如，要对字符串数组进行排序，可以使用 `compare:` 方法代替。与前面比较两个 `NSNumber` 对象的 `compare:` 方法相似，如果词汇中第一个字符串小于第二个字符串，结果是 `NSOrderedAscending`；如果两个相等，结果是 `NSOrderedSame`；如果第一个字符串大于第二个，结果是 `NSOrderedDescending`。如果不想进行大小写敏感的比较，可使用 `caseInsensitiveCompare:` 方法，而不使用 `compare:` 方法。在这个例子中，使用 `caseInsensitiveCompare` 比较字符串对象 `@“Gregory”` 和 `@“gregory”` 会相等。

代码清单 15-3 中用到的最后两个 `NSString` 方法是 `uppercaseString` 和 `lowercaseString`，分别转换成大写字符和小写字符。同样，这样转换并不影响原始的字符串，最后一行的输出可以验证这一点。

将 `str1` 和 `str2` 声明为不可变字符串对象，意味着它们所引用的字符串对象的字符不可以改变。然而，引用 `str1` 和 `str2` 是可以改变的。也就是说，可以重新为它们指定其他的不可变字符串对象。这一点非常重要。图 15.1 说明了变量 `res` 和 `str1` 声明和初始化后的情形。变量 `res` 并未设置任何初始值，因此，它的内容为空。然而，`str1` 指向了存储在内存某处的常量字符串对象 `@“This is string A”`。

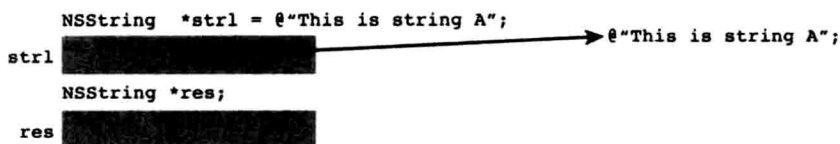


图 15.1 不可变字符串的声明和初始化

在代码清单 15-3 中给 `str1` 发送一个 `uppercaseString` 消息，会得到一个依据 `str1` 创建的新字符串，字符串中的小写字符均被替换成大写字符，如图 15.2 所示。

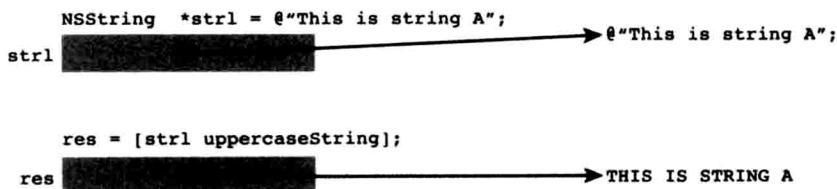


图 15.2 向一个字符串对象发送一个 `uppercaseString` 消息

请注意，新字符串对象创建后，`str1` 仍然指向原始的字符串对象。

向 `str1` 发送消息 `lowercaseString` 也是一样，将 `str1` 的大写字符转化成小写字符，创建出一个新的字符串。新创建的字符串对象引用存储在变量 `res` 中，如图 15.3 所示。请注意，前一步中创建的大写字符串不再被引用。不必担心这点，系统的内存管理会为你清理这个对象。

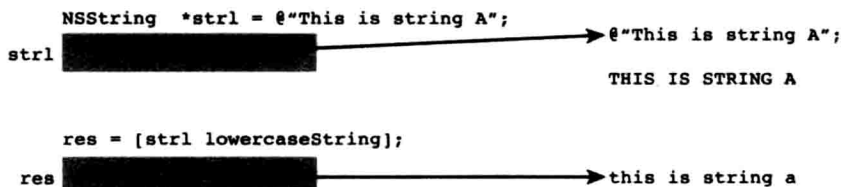


图 15.3 向一个字符串对象发送 `lowercaseString` 消息

代码清单 15-5 举例说明更多的字符串处理方法。这些方法允许你提取字符串中的子字符串，即在一个字符串中搜索另一个字符串。

一些方法需要指定一个范围确定子字符串，包括开始索引数和字符数，索引数以 0 开始，因此，使用数字对 `{0, 3}` 指定字符串中的前 3 个字符。`NSString` 类（和其他的 Foundation 类）的一些方法中，使用了特殊的数据类型 `NSRange` 创建范围对象。实际上，它是结构的 `typedef` 定义，包含 `location` 和 `length` 两个成员。代码清单 15-5 中使用了这个数据类型。

注意

你可以阅读第 13 章“基本的 C 语言特性”中关于结构的内容。但是，在这一章后面的讨论中你可以获得足够有关如何使用它们的信息。

代码清单 15-5

// 基本的字符串操作——续

```

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSString *str1 = @"This is string A";
        NSString *str2 = @"This is string B";
        NSString *res;
        NSRange subRange;
    }
}

```

```

// 从字符串中提取前 3 个字符

res = [str1 substringToIndex: 3];
NSLog(@"First 3 chars of str1: %@", res);

// 提取从索引 5 开始直到结尾的子字符串

res = [str1 substringFromIndex: 5];
NSLog(@"Chars from index 5 of str1: %@", res);

// 提取从索引 5 开始到索引 13 的子字符串 (6 个字符)

res = [[str1 substringFromIndex: 8] substringToIndex: 6];
NSLog(@"Chars from index 8 through 13: %@", res);

// 更简单的方法

res = [str1 substringWithRange: NSRange (8, 6)];
NSLog(@"Chars from index 8 through 13: %@", res);

// 从另一个字符串中查找一个字符串

subRange = [str1 rangeOfString: @"string A"];
NSLog(@"String is at index %lu, length is %lu",
      subRange.location, subRange.length);

subRange = [str1 rangeOfString: @"string B"];

if (subRange.location == NSNotFound)
    NSLog(@"String not found");
else
    NSLog(@"String is at index %lu, length is %lu",
          subRange.location, subRange.length);

}
return 0;
}

```

代码清单 15-5 输出

```

First 3 chars of str1: Thi
Chars from index 5 of str1: is string A
Chars from index 8 through 13: string
Chars from index 8 through 13: string
String is at index 8, length is 8
String not found

```

substringToIndex:方法创建了一个子字符串，包括首字符到指定的索引数，但不包括这个字符。因为索引数是从 0 开始的，所以参数 3 表示从字符串中提取字符 0、1 和 2，并返回结果字符串对象。对于所有采用索引数作为参数的字符串方法，如果提供的索引数对该字符串无效，就会获得 **Range or index out of bounds** 的出错消息。

substringFromIndex:方法返回了一个子字符串，它从接收者指定的索引字符开始，直到字符串的结尾。

表达式

```
res = [[str1 substringFromIndex: 8] substringToIndex: 6];
```

显示了如何结合这两个方法，提取字符串内部的子字符串。首先使用 **substringFromIndex:**方法从索引数 8 开始，直到字符串结尾的字符。然后对结果应用 **substringToIndex:**方法，以获得前 6 个字符。最终结果是一个子字符串是原字符串中 {8, 6} 范围的字符。

使用 **substringWithRange:**方法能一步完成我们刚刚用两步所做的工作：接受一个范围，返回指定范围的字符。特殊函数

```
NSMakeRange (8, 6)
```

根据参数创建一个范围，并返回结果。这个结果可以作为 **substringWithRange:**方法的参数。

要在另一个字符串中查找一个字符串，可以使用 **rangeOfString:**方法。如果在接收者中找到指定的字符串，则返回的范围是找到的精确位置。然而，如果没有找到这个字符串，则返回范围的 **location** 成员被设置为 **NSNotFound**。

所以，语句

```
subRange = [str1 rangeOfString: @"string A"];
```

把方法返回的 **NSRange** 结构赋值给 **NSRange** 变量 **subRange**。一定要注意，**subRange** 不是对象变量，而是一个结构变量（并且程序中的 **subRange** 声明不包括星号，这通常意味着不是在处理一个对象，不过 **id** 类型是一个例外）。通过使用结构成员操作符（**.**），可以检索其成员。所以，表达式 **subRange.location** 给出了该结构中成员 **location** 的值，**subRange.length** 给出结构成员 **length** 的值，将这些值传递给 **NSLog** 函数显示。

15.2.4 可变字符串

NSMutableString 类可以用来创建可以更改字符的字符串对象。因为是 NSString 类的子类，所以可以使用 NSString 类的所有方法。

在讲述可变与不可变字符串对象时，我们谈到了更改字符串中的实际字符。任意一个可变或不可变字符串对象在程序执行期间，总是可以被设为完全不同的字符串对象的。这在程序 15-4 中讨论过，例如，考虑以下代码：

```
str1 = @"This is a string";
...
str1 = [str1 substringFromIndex: 5];
```

在这个例子中，首先将 str1 设置为一个常量字符串对象。后来在程序中将其设为一个子字符串。在这个例子中，str1 可以声明为可变的字符串对象，也可以声明为不可变的字符串对象。一定要理解这一点。

代码清单 15-6 展示了处理程序中可变字符串的几种方式。

代码清单 15-6

// 可变字符串的基本字符串操作

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSString *str1 = @"This is string A";
        NSString *search, *replace;
        NSMutableString *mstr;
        NSRange substr;

        // 从不可变字符串创建可变字符串

        mstr = [NSMutableString stringWithString: str1];
        NSLog ("%s", mstr);

        // 插入字符

        [mstr insertString: @" mutable" atIndex: 7];
        NSLog ("%s", mstr);

        // 插入末尾进行有效拼接

        [mstr insertString: @" and string B" atIndex: [mstr length];
```

```

NSLog (@"%@", mstr);

// 直接使用 appendString

[mstr appendString: @" and string C"];
NSLog (@"%@", mstr);

// 根据范围删除子字符串

[mstr deleteCharactersInRange: NSMakeRange (16, 13)];
NSLog (@"%@", mstr);

// 查找然后将其删除

substr = [mstr rangeOfString: @"string B and "];

if (substr.location != NSNotFound) {
    [mstr deleteCharactersInRange: substr];
    NSLog (@"%@", mstr);
}

// 直接设置为可变的字符串

[mstr setString: @"This is string A"];
NSLog (@"%@", mstr);

// 替换一些字符

[mstr replaceCharactersInRange: NSMakeRange(8, 8)
    withString: @"a mutable string"];
NSLog (@"%@", mstr);

// 查找和替换

search = @"This is";
replace = @"An example of";

substr = [mstr rangeOfString: search];

if (substr.location != NSNotFound) {
    [mstr replaceCharactersInRange: substr
        withString: replace];
    NSLog (@"%@", mstr);
}

// 查找和替换所有的匹配项

search = @"a";

```

```

    replace = @"X";

    substr = [mstr rangeOfString: search];

    while (substr.location != NSNotFound) {
        [mstr replaceCharactersInRange: substr
            withString: replace];
        substr = [mstr rangeOfString: search];
    }

    NSLog(@"%@", mstr);

}
return 0;
}

```

代码清单 15-6 输出

```

This is string A
This is mutable string A
This is mutable string A and string B
This is mutable string A and string B and string C
This is mutable string B and string C
This is mutable string C
This is string A
This is a mutable string
An example of a mutable string
An exXmple of X mutXble string

```

声明

```
NSMutableString *mstr;
```

将 `mstr` 声明为一个变量，用来存储在程序执行过程中值可能更改的字符串对象。语句行

```
mstr = [NSMutableString stringWithString: str1];
```

将 `mstr` 设置为字符串对象，内容是 `str1` 中字符的副本，即“This is string A”。将 `stringWithString:` 方法发送给 `NSMutableString` 类，返回一个可变的字符串对象。而将 `stringWithString:` 方法发送给 `NSString` 类时，如代码清单 15-5 所示，则返回一个不可变的字符串对象。

`insertString:atIndex:` 方法将指定的字符串插入接收者，插入点从指定的索引值开始。这个例子中，在字符串的索引数 7 处插入字符串 `@“mutable”`，即在第

8 个字符之前。与不可变字符串对象不同的是，这里没有返回值，因为被修改的是接收者，它是可变的字符串对象，所以可以这么做。

第二个 `insertString:atIndex:`调用 `length` 方法将一个字符串插入另一个字符串结尾。`appendString:`可使这个任务更简单一些。

使用 `deleteCharactersInRange:`方法可以删除字符串中指定数目的字符。对以下字符串

`This is mutable string A and string B and string C`

应用范围{16, 13}，从索引数 16（或者字符串中的第 17 个字符）开始删除 13 个字符“string A and”，如图 15.4 所示。

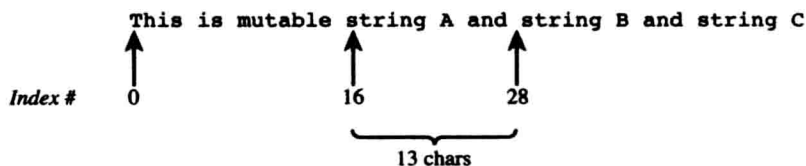


图 15.4 字符串中的索引

在代码清单 15-5 中，接下来的代码用到 `rangeOfString:`方法，说明了如何找到字符串并进行删除。首先验证 `mstr` 中的确存在字符串 `@"string B and"`，然后使用 `rangeOfString:`方法返回一个范围作为 `deleteCharactersInRange:`方法的参数，最后删除这个字符串。

`setString:`方法可以用来设置可变字符串对象的内容。使用这个方法将 `mstr` 设置为字符串 `@"This is string A"`，`replaceCharactersInRange:withString:`方法可用另一个字符串替换这个字符串中的部分字符。字符串的大小不必相同，可以使用大小相等或不等的字符串替换另一个字符串。因此，语句

```
[mstr replaceCharactersInRange: NSMakeRange(8, 8)
    withString: @"a mutable string"];
```

的结果是将 8 个字符“string A”替换成 16 个字符“a mutable string”。

代码清单 15-5 中其余几行说明了如何执行查找和替换操作。首先，在字符串 `mstr` 中（它包含 `@"This is a mutable string"`）查找字符串 `@"This is"`。如果在搜索字符串中找到这个内容，就用字符串 `@"An example of"` 替换匹配的字符串。最终 `mstr` 中包含的字符串变成 `@"An example of a mutable string"`。

然后，设置一个循环说明如何实现“查找并全部替换”的操作。搜索字符串被设置为@"a"，替换字符串被设置为@"X"。

注意，如果替换字符串中还包括搜索字符串（例如，考虑使用字符串"aX"替换字符串"a"），那么将会陷入无限循环。

其次，如果替换字符串为空（也就是不包含字符），那么将有效地删除所有搜索到的字符串。没有空格隔开的相邻引号可指定为空的常量字符串，语句如下：

```
replace = @"";
```

当然，如果只想删除字符串，则可以使用 `deleteCharactersInRange:` 方法，前面我们已经学过。

最后，`NSMutableString` 类还包含一个名为 `replaceOccurrencesOfString:withString:options:range:` 的方法，可以用来执行搜索并全部替换操作。实际上，代码清单 15-5 中的 `while` 循环可以替换为以下一行代码：

```
[mstr replaceOccurrencesOfString: search
      withString: replace
      options: nil
      range: NSRange (0, [mstr length])];
```

这会得到相同的结果，而且可以避免潜在的无限循环，因为这个方法会防止这样的事情发生。

`NSString` 类包含 100 多个方法，它可以用来处理不可变的字符串对象。表 15.2 总结了一些常用的方法，表 15.3 列出了 `NSMutableString` 类提供的一些附加方法。其他一些 `NSString` 方法（例如，处理路径名并将文件的内容读入一个字符串）将在本书后面部分进行介绍。如果希望掌握更多的方法，可以查看 `NSString` 类的文档。

表 15.2 常见的 `NSString` 方法

方 法	描 述
<code>+(instancetype) stringWithContentsOfFile: path encoding: enc error: err</code>	创建一个新字符串，并将其设置为 <code>path</code> 指定的文件的内容，使用字符编码 <code>enc</code> ，如果非零，则返回 <code>err</code> 中的错误
<code>+(instancetype) stringWithContentsOfURL: url encoding: enc error: err</code>	创建一个新字符串，并将其设置为 <code>url</code> 的内容，使用字符编码 <code>enc</code> ，如果非零，则返回 <code>err</code> 中的错误
<code>+(instancetype) string</code>	创建一个新的空字符串

续表

方 法	描 述
+(instancetype) stringWithString: <i>nsstring</i>	创建一个新字符串，并将其设置为 <i>nsstring</i>
+(NSString *) stringWithFormat: <i>format, arg1, arg2, arg3 ...</i>	通过指定的 <i>format</i> 和参数 <i>arg1, arg2, arg3...</i> 创建一个新字符串
-(instancetype) initWithString: <i>nsstring</i>	将新分配的字符串设置为 <i>nsstring</i>
-(instancetype) initWithContentsOfFile: <i>path</i> encoding: <i>enc</i> error: <i>err</i>	将字符串设置为 <i>path</i> 指定的文件的内容
-(instancetype) initWithContentsOfURL: <i>url</i> encoding: <i>enc</i> error: <i>err</i>	将字符串设置为 <i>url</i> (NSURL *) <i>url</i> 的内容，使用字符编码 <i>enc</i> ，如果非零，则返回 <i>err</i> 中的错误
-(NSUInteger) length	返回字符串中的字符数目
-(unichar) characterAtIndex: <i>i</i>	返回索引 <i>i</i> 的 Unicode 字符
-(NSString *) substringFromIndex: <i>i</i>	返回从 <i>i</i> 开始直到结尾的子字符串
-(NSString *) substringWithRange: <i>range</i>	根据指定的范围返回子字符串
-(NSString *) substringToIndex: <i>i</i>	返回从该字符串开始直到索引 <i>i</i> 的子字符串
-(NSComparator *) caseInsensitiveCompare: <i>nsstring</i>	比较两个字符串，忽略大小写
-(NSComparator *) compare: <i>nsstring</i>	比较两个字符串
-(BOOL) hasPrefix: <i>nsstring</i>	测试字符串是否以 <i>nsstring</i> 开始
-(BOOL) hasSuffix: <i>nsstring</i>	测试字符串是否以 <i>nsstring</i> 结尾
-(BOOL) isEqualToString: <i>nsstring</i>	测试两个字符串是否相等
-(NSString *) capitalizedString	返回每个单词首字母大写的字符串（每个单词的其余字母转换为小写）
-(NSString *) lowercaseString	返回转换为小写的字符串
-(NSString *) uppercaseString	返回转换为大写的字符串
-(const char *) UTF8String	返回转换为 UTF-8 C 样式的字符串
-(double) doubleValue	返回转换为 double 的字符串
-(float) floatValue	返回字符串表示的双精度浮点数
-(NSInteger) integerValue	返回字符串的 NSInteger 整数表示
-(int) intValue	返回转换为整数的字符串

在表 15.2 和表 15.3 中，*url* 是一个 NSURL 对象，*path* 是指明文件路径的 NSString 对象，*nsstring* 是一个 NSString 对象，*i* 表示字符串中有效字符数的 NSUInteger 值，*enc* 指明字符编码的 NSStringEncoding 对象，*err* 是描述所发生错误的 NSError 对象，*size* 和 *opts* 是 NSUInteger，*range* 是指明字符串中有效范围的 NSRange 对象。

表 15.3 说明了创建或修改 NSMutableString 对象的方法。

NSString 对象广泛地应用在本文的其他部分。如果需要把字符串分解为语言符号，可以查看 Foundation 的 NSScanner 类。

表 15.3 常见的 NSMutableString 方法

方 法	描 述
+(id) stringWithCapacity: <i>size</i>	创建一个初始包含 <i>size</i> 字符的字符串
-(id) initWithCapacity: <i>size</i>	使用初始容量为 <i>size</i> 的字符来初始化字符串
-(void) setString: <i>nsstring</i>	将字符串设置为 <i>nsstring</i>
-(void) appendString: <i>nsstring</i>	在接收者的末尾附加 <i>nsstring</i>
-(void) deleteCharactersInRange: <i>range</i>	删除指定 <i>range</i> 中的字符
-(void) insertString: <i>nsstring</i> atIndex: <i>i</i>	以索引 <i>i</i> 为起始位置插入 <i>nsstring</i>
-(void) replaceCharactersInRange: <i>range</i> withString: <i>nsstring</i>	使用 <i>nsstring</i> 替换 <i>range</i> 指定的字符
-(void) replaceOccurrencesOfString: <i>nsstring</i> withString: <i>nsstring2</i> options: <i>opts</i> range: <i>range</i>	根据选项 <i>opts</i> ，在指定范围 <i>range</i> 中用 <i>nsstring2</i> 替换所有的 <i>nsstring</i> 。选项可以是以下值的按位或组合：NSBackwardsSearch（查找从范围尾部开始）、NSAnchoredSearch（ <i>nsstring</i> 必须匹配范围的开始）、NSLiteralString（执行逐字符比较）以及 NSCaseInsensitiveSearch

15.3 数组对象

Foundation 数组是有序的对象集合。最常见的是，一个数组中的元素都是一种特定类型，但不是必需的。就像存在可变字符串和不可变字符串，同样也存在可变数组和不可变数组。不可变数组是由 NSArray 类处理的，而可变数组是由 NSMutableArray 处理的。后者是前者的子类，也就是说，后者继承了前者的方法。

代码清单 15-7 设置了一个数组存储一年中月份的名字，然后显示这些月份。

代码清单 15-7

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

int main (int argc, char *argv[])
{
    int i;
    @autoreleasepool {
```

```

// 创建一个数组包含月份的名称

NSArray *monthNames = [NSArray arrayWithObjects:
    @"January", @"February", @"March", @"April",
    @"May", @"June", @"July", @"August", @"September",
    @"October", @"November", @"December", nil ];

// 列出数组中所有的元素

NSLog(@"Month Name");
NSLog(@"===== ");

for (i = 0; i < 12; ++i)
    NSLog(@" %2i    %@", i + 1, [monthNames objectAtIndex: i]);
}
return 0;
}

```

代码清单 15-7 输出

```

Month Name
=====
1    January
2    February
3    March
4    April
5    May
6    June
7    July
8    August
9    September
10   October
11   November
12   December

```

类方法 `arrayWithObjects:` 使用一系列对象作为元素创建数组。这种情况下，需要按顺序列出对象并使用逗号隔开。这种方法使用的特殊语法可以接收可变数量的参数。但需要标记参数数组的结束，将这个数组的最后一个值指定为 `nil`，它实际上并不会存储在数组中。

在代码清单 15-7 中，`monthNames` 被设置为 `arrayWithObjects:` 参数所指定的 12 个字符串。

数组中的元素是由它们的索引数确定的。与 `NSString` 对象类似，索引从 0 开始。所以，包含 12 个元素的数组的有效索引数是 0~11。使用 `objectAtIndex:`

方法需要索引数检索数组中的元素。

程序中仅仅使用 `objectAtIndex:` 方法简单地执行了一个 `for` 循环，从数组中提取每个元素，将每个检索到的元素使用 `NSLog` 进行显示。

Objective-C 使用如下语法能够轻松创建一个 `NSArray` 对象：

```
@[elem1, elem2, ... elemn]
```

在这种情况下，你不必将 `nil` 放在这些元素列表的末尾。Objective-C 允许你使用以下语法引用数组的元素：

```
array[index]
```

这相当于如下表达式：

```
[array objectAtIndex: index]
```

你还可以使用这种语法将对象的引用存储到数组中：

```
array[index] = object
```

这相当于如下表达式：

```
[array setObject: object forIndex: index]
```

代码清单 15-8 使用另一种语法重写了代码清单 15-7 中的语句定义了数组并引用数组中的元素。

代码清单 15-8

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

int main (int argc, char * argv[])
{
    int i;
    @autoreleasepool {
        // 创建一个包含各月份名称的数组

        NSArray *monthNames = @[@"January", @"February", @"March", @"April", @"May",
                                @"June", @"July", @"August", @"September", @"October", @"November",
                                @"December"];

        // 列出数组中的所有元素

        NSLog(@"Month Name");
        NSLog(@"=====");

        for (i = 0; i < 12; ++i)
            NSLog(@" %2i %@", i + 1, monthNames[i]);
    }
}
```

```

    }
    return 0;
}

```

代码清单 15-9 中创建了一个包含 10 个数字对象的数组，值从 0 到 9。可以使用 for 循环取出这些值，然后通过 NSLog 格式化字符串%@显示整个数组。本章后面，你还将了解到一种快速迭代的技术，能够顺序遍历数组的元素。

代码清单 15-9

```

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSMutableArray *numbers = [NSMutableArray array];
        int i;

        // 创建 0~9 数字的数组

        for (i = 0; i < 10; ++i ) {
            numbers[i] = @(i);
        }

        // 遍历数组与显示其值

        for (i = 0; i < 10; ++i ) {
            NSLog (@"%@", numbers[i]);
        }

        // 使用带有%@格式的 NSLog 显示

        NSLog (@"==== Using a single NSLog");
        NSLog (@"%@", numbers);
    }
    return 0;
}

```

代码清单 15-9 输出

```

2013-11-12 15:25:42.701 prog15.7[6379:903] 0
2013-11-12 15:25:42.704 prog15.7[6379:903] 1
2013-11-12 15:25:42.704 prog15.7[6379:903] 2
2013-11-12 15:25:42.704 prog15.7[6379:903] 3
2013-11-12 15:25:42.705 prog15.7[6379:903] 4
2013-11-12 15:25:42.705 prog15.7[6379:903] 5
2013-11-12 15:25:42.705 prog15.7[6379:903] 6

```

```

2013-11-12 15:25:42.706 prog15.7[6379:903] 7
2013-11-12 15:25:42.706 prog15.7[6379:903] 8
2013-11-12 15:25:42.706 prog15.7[6379:903] 9
2013-11-12 15:25:42.707 prog15.7[6379:903] ===== Using a single NSLog
2013-11-12 15:25:42.707 prog15.7[6379:903] (
    0,
    1,
    2,
    3,
    4,
    5,
    6,
    7,
    8,
    9
)

```

（这里列出所有的 NSLog 输出只是为了区分第一组和第二组输出的差异。）

使用 NSMutableArray 的方法 array 创建一个空的可变数组对象。数组元素的个数并未指定并且可以根据需要增长。

曾经提到，不能将整数这样的基本数据类型存储在数组之类的集合中。所以，我们需要构造每个值 i 的 NSNumber 对象，范围从 0 到 9。

如果需要向数组的末尾添加一个对象，可以使用 addObject: 方法。正如代码清单 15-9 中的那样。

```
[numbers addObject: @(i)];
```

将创建的 NSNumber 对象依次添加到数组的末尾。

然后程序进入 for 循环，显示存储在数组中的每一个数字对象。

最后在代码清单 15-9 中，使用单个 NSLog 格式化字符%@显示整个数组。

注意

NSLog 是如何显示存储在数组中的对象的？对于数组中的每一个元素，NSLog 将使用属于每个元素类的 description 方法。如果使用的是从 NSObject 对象继承的默认方法，获取到的是对象的类和地址，正如前面提到的。然而，在这个例子中，能够获取到更有意义的内容，说明 NSNumber 类有自定义的 description 方法实现。

Foundation 类为使用数组提供了许多便利。然而，如果使用复杂的运算法

则操纵大型数字数组，学习使用 C 语言提供的低级数组构造执行这种任务可能更加有效，对于内存使用和执行速度来说，都是如此。参考第 13 章中有关“数组”的内容获取更多信息。

15.3.1 制作地址簿

下面来看一个例子，它结合到目前为止学到的知识，生成一个地址簿^②。该地址簿包含一些地址卡片。为简单起见，地址卡片仅包含某人的姓名和 email 地址。将这个概念扩展到其他信息很简单，比如地址和电话号码，这将作为本章末尾的练习题留给你来完成。

下面创建两个类 `AddressBook` 和 `AddressCard`，两者的关系如图 15.5 所示。

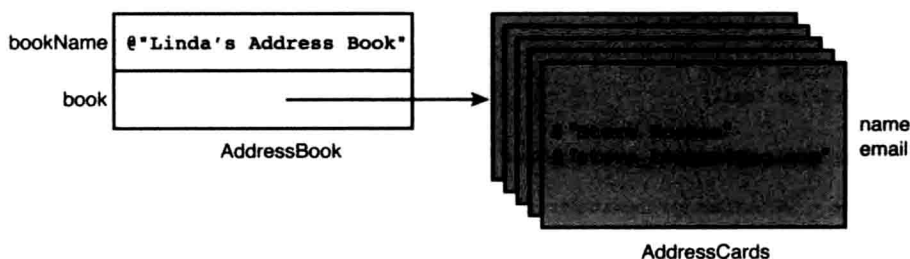


图 15.5 `AddressBook` 包含 `AddressCards`

1. 生成一个地址卡片

这里从定义名为 `AddressCard` 的类开始。需要实现以下功能：创建一个新的地址卡片、设置卡片的姓名字段和 `email` 字段、检索这些字段的内容，并打印卡片。在图形化的环境中，可以使用一些简便的方法在计算机或者 iOS 设备屏幕上绘制卡片。但在这里，继续使用简单的终端界面显示地址卡片。

代码清单 15-10 为新的 `AddressCard` 类接口文件。这里不再介绍访问器方法，你可以自己编写并从中学习到更多的内容。

代码清单 15-10 接口文件 `AddressCard.h`

```

#import <Foundation/Foundation.h>

@interface AddressCard: NSObject

```

^② OS X 和 iOS 提供了整个地址簿的框架，提供了极其强大的功能用于处理地址簿。

```

-(void) setName: (NSString *) theName;
-(void) setEmail: (NSString *) theEmail;
-(NSString *) name;
-(NSString *) email;

-(void) print;

@end

```

这与代码清单 15-10 中的实现文件一样简单、直观。

代码清单 15-10 实现文件 AddressCard.m

```

#import "AddressCard.h"

@implementation AddressCard
{
    NSString *name;
    NSString *email;
}

-(void) setName: (NSString *) theName
{
    name = [NSString stringWithString: theName];
}

-(void) setEmail: (NSString *) theEmail
{
    email = [NSString stringWithString: theEmail];
}

-(NSString *) name
{
    return name;
}

-(NSString *) email
{
    return email;
}

-(void) print
{
    NSLog (@"=====");
    NSLog (@"|                               |");
    NSLog (@"|  %-31s  |", [name UTF8String]);
    NSLog (@"|  %-31s  |", [email UTF8String]);
}

```

```

NSLog(@"|                                |");
NSLog(@"|                                |");
NSLog(@"|                                |");
NSLog(@"|          O          O          |");
NSLog(@"|=====|");

}
@end

```

可以使用方法 `setName:` 和 `setEmail:` 直接将这些对象存储在各自的实例变量中，语句如下：

```

-(void) setName: (NSString *) theName
{
    name = theName;
}

-(void) setEmail: (NSString *) theEmail
{
    email = theEmail;
}

```

但是，`AddressCard` 对象并不包含它自己的成员对象（仅仅包含方法传递的参数引用）。在第 8 章“继承”中，我们以 `Rectangle` 类包含的 `origin` 对象为例，讨论对象的归属。

`print` 方法尝试采用类似于 Rolodex 卡片（还记得它们吗？）的格式向用户展示良好效果的地址卡片。`NSLog` 中的 `"%-31s"` 字符表示要用 31 个字符的字段宽度左对齐打印 UTF8 C-字符串，确保输出的地址卡片右边缘是整齐的。在这个例子中起到修饰结果的作用。

使用 `AddressCard` 类编写一个测试程序创建地址卡片、设置卡片的值及显示卡片（参见代码清单 15-10）。

代码清单 15-10 测试程序

```

#import "AddressCard.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSString *aName = @"Julia Kochan";
        NSString *aEmail = @"jewls337@axlc.com";
        AddressCard *card1 = [[AddressCard alloc] init];
    }
}

```

```

    [card1 setName: aName];
    [card1 setEmail: aEmail];
    [card1 print];
}
return 0;
}

```

代码清单 15-10 输出

```

=====
|                               |
| Julia Kochan                 |
| jewls337@axlc.com           |
|                               |
|                               |
|                               |
|      O      O               |
|                               |
=====

```

下面是新的 setName: 和 setEmail: 方法:

```

-(void) setName: (NSString *) theName
{
    if (name != theName)
        name = [NSString stringWithString: theName];
}

-(void) setEmail: (NSString *) theEmail
{
    if (email != theEmail)
        email = [NSString stringWithString: theEmail];
}

```

if 语句测试了发送到访问器方法的对象是否已经存在于实例变量中。如果传入的是同一个对象, 就不需要再进行设置。

2. 同步 AddressCard 方法

我们已经编写了访问器方法 setName: 和 setEmail:, 你也理解了那些重要的原理, 那么我们可以退一步, 让系统生成访问器方法。考虑 AddressCard 接口文件的第 2 个版本:

```

#import <Foundation/Foundation.h>

@interface AddressCard: NSObject

@property (copy, nonatomic) NSString *name, *email;

```

```
-(void) print;
@end
```

语句

```
@property (copy, nonatomic) NSString *name, *email;
```

列出了属性的 `copy` 特性和 `nonatomic` 特性。和你编写的版本一样，`copy` 特性会在 `setter` 方法内生成实例变量的副本。默认的方法不会生成副本，仅仅进行赋值（默认为 `assign` 特性），而这不是我们期望的。

`nonatomic` 特性表明不必担心在竞争条件下多个线程试图同时获取实例变量的情形。相关内容在第 18 章“复制对象”将会详细介绍。

代码清单 15-11 表示新的 `AddressCard` 实现文件，存取方法会被同步（这里显性声明实例变量是因为需要列出实例变量的属性）。

代码清单 15-11 带同步方法的实现文件 `AddressCard.m`

```
#import "AddressCard.h"

@implementation AddressCard

@synthesize name, email;

-(void) print
{
    NSLog(@"=====");
    NSLog(@"|                |");
    NSLog(@"| %-31s |", [name UTF8String]);
    NSLog(@"| %-31s |", [email UTF8String]);
    NSLog(@"|                |");
    NSLog(@"|                |");
    NSLog(@"|                |");
    NSLog(@"|          O          O          |");
    NSLog(@"|=====");
}

@end
```

现在，为 `AddressCard` 类添加另一个方法，你可能希望在调用方法的同时设置卡片的姓名和 `email` 这两个字段。为此，添加一个新方法 `setName:`

andEmail:, ^③语句如下:

```
-(void) setName: (NSString *) theName andEmail: (NSString *) theEmail
{
    self.name = theName;
    self.email = theEmail;
}
```

曾经提过, `self.name = theName;`等价于`[self setName: theName];`, 这是因为使用了 setter 方法为实例变量赋值。与此不同的是, `name = theName;`的写法绕过了 setter 方法, 直接为实例变量赋了参数的值。

使用同步 setter 方法设置实例变量（而不是在方法中直接设置它们），会增加一层抽象性，从而使程序更加独立于它的内部数据结构。你也可以同步属性，本例中是为了实例变量复制而不仅仅是赋值。代码清单 15-11 测试了新的方法。

代码清单 15-11 测试程序

```
#import "AddressCard.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSString *aName = @"Julia Kochan";
        NSString *aEmail = @"jewels337@axlc.com";
        NSString *bName = @"Tony Iannino";
        NSString *bEmail = @"tony.iannino@techfitness.com";

        AddressCard *card1 = [[AddressCard alloc] init];
        AddressCard *card2 = [[AddressCard alloc] init];

        [card1 setName: aName andEmail: aEmail];
        [card2 setName: bName andEmail: bEmail];

        [card1 print];
        [card2 print];
    }
    return 0;
}
```

代码清单 15-11 输出

```
=====
```

③ 你可能还需要 `initWithName:andEmail:` 的初始化方法，但这里我们不做说明。

```

|                                     |
| Julia Kochan                       |
| jewls337@axlc.com                 |
|                                     |
|                                     |
|                                     |
|      O      O                     |
|=====|
|                                     |
| Tony Iannino                       |
| tony.iannino@techfitness.com       |
|                                     |
|                                     |
|                                     |
|      O      O                     |
|=====|

```

3. AddressBook 类

你的 AddressCard 类看起来良好，但如果需要使用很多 AddressCard，该怎么办呢？把它们集中到一起也很合理，可以定义一个名为 AddressBook 的新类实现这项任务。AddressBook 类存储地址簿的名字和一个 AddressCard 的集合，将这个集合存储在一个数组对象中。首先，你需要创建新的地址簿，向其添加地址卡片，计算地址簿的记录数，列出地址簿的内容。然后你可能需要更多的功能，如搜索地址簿、删除记录、编辑现有记录、将记录排序，甚至复制记录。

先看一个简单的接口文件（参见代码清单 15-12）。

代码清单 15-12 AddressBook.h 接口文件

```

#import <Foundation/Foundation.h>
#import "AddressCard.h"

@interface AddressBook: NSObject

@property (nonatomic, copy) NSString *bookName;
@property (nonatomic, strong) NSMutableArray *book;

-(instancetype) initWithName: (NSString *) name;
-(void) addCard: (AddressCard *) theCard;
-(int) entries;
-(void) list;

```

```
@end
```

strong 是属性的一个特性。在第 17 章“内存管理和自动引用计数”有更多描述。

使用 **initWithName:** 方法设置初始数组，用于存放地址卡片，同时保存地址簿的名称，而使用 **addCard:** 方法是向地址簿添加 **AddressCard**。**entries** 方法可以获取地址簿中卡片的数量，而 **list** 方法可以显示地址簿中全部的内容。**AddressBook** 类的实现文件参见代码清单 15-12。

代码清单 15-12 AddressBook.m 实现文件

```
#import "AddressBook.h"
@implementation AddressBook
@synthesize bookName, book;

// 设置 AddressBook 的名称和一个空的 AddressBook

-(instancetype) initWithName: (NSString *) name
{
    self = [super init];

    if (self) {
        bookName = [NSString stringWithString: name];
        book = [NSMutableArray array];
    }

    return self;
}

-(instancetype) init
{
    return [self initWithName: @"NoName"];
}

-(void) addCard: (AddressCard *) theCard
{
    [book addObject: theCard];
}

-(int) entries
{
    return [book count];
}

-(void) list
```

```

{
    NSLog (@"===== Contents of: %@ =====", bookName);

    for ( AddressCard *theCard in book )
        NSLog (@"%-20s    %-32s", [theCard.name UTF8String],
                [theCard.email UTF8String]);

    NSLog (@"=====");
}
@end

```

initWithName:方法首先会调用超类的 **init** 方法执行初始化过程。然后，将方法参数传递过来的字符串复制一份存储在实例变量 **bookName** 中，再创建一个空的 **NSMutableArray** 对象赋给实例变量 **book**。

定义的 **initWithName:**方法返回一个 **instancetype** 对象，而不是 **AddressBook** 对象（或者 **id**）。如果创建 **AddressBook** 的子类，那么 **initWithName:**消息的接收者（以及返回值）不是 **AddressBook** 对象，它的类型是子类的类型。因此，需将返回类型定义为一般的对象类型。

当编译器遇见 **instancetype** 作为返回类型，它就知道返回的类是发送消息的对象。它实际上并没有影响应用程序的执行，编译器会提供更多的线索帮助发现程序中潜在的问题。

覆写 **init** 方法能够保证：如果某人在调用 **alloc** 方法之后调用 **init** 方法，仍然可以正确地创建地址簿，而且默认的名字是“NoName”。这里的 **initWithName:**方法是我们指定的初始化方法，目的是确保 **init** 方法会调用到它。

addCard:方法将 **AddressCard** 对象作为参数，把它添加到地址簿中。

count 方法返回数组元素的个数。**entries** 方法中使用这个方法返回地址簿中存储的地址卡片数目。

4. 快速枚举

list 方法的 **for** 循环展示了一个你以前从未见过的结构：

```

for ( AddressCard *theCard in book )
    NSLog (@"%-20s    %-32s", [theCard.name UTF8String],
            [theCard.email UTF8String]);

```

在这里对 **book** 数组中的每个元素序列使用名为快速枚举的技术。它的语法非常简单：首先定义一个能够保留数组中每个元素的变量（**AddressCard**

`*theCard`)。使用关键字 `in`，然后列出数组的名称。当执行 `for` 循环时，它会先将数组的第一个元素赋给指定的变量，并执行循环体。然后，将第二个元素赋给变量，并执行循环体。这样一直持续下去，直到数组的所有元素都已赋给变量，这样数组的每个元素都执行了循环体。

注意，如果 `theCard` 已经被定义为 `AddressCard` 对象，那么 `for` 循环会变得更加简单，语句如下：

```
for ( theCard in book )
    ...
```

代码清单 15-12 是新类 `AddressBook` 的测试程序。

代码清单 15-12 测试程序

```
#import "AddressBook.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSString *aName = @"Julia Kochan";
        NSString *aEmail = @"jewls337@axlc.com";
        NSString *bName = @"Tony Iannino";
        NSString *bEmail = @"tony.iannino@techfitness.com";
        NSString *cName = @"Stephen Kochan";
        NSString *cEmail = @"steve@classroomM.com";
        NSString *dName = @"Jamie Baker";
        NSString *dEmail = @"jbaker@classroomM.com";

        AddressCard *card1 = [[AddressCard alloc] init];
        AddressCard *card2 = [[AddressCard alloc] init];
        AddressCard *card3 = [[AddressCard alloc] init];
        AddressCard *card4 = [[AddressCard alloc] init];

        // 创建一个新的地址簿

        AddressBook *myBook = [[AddressBook alloc]
                               initWithName: @"Linda's Address Book"];

        NSLog (@@"Entries in address book after creation: %i",
               [myBook entries]);

        // 创建 4 个地址卡片

        [card1 setName: aName andEmail: aEmail];
        [card2 setName: bName andEmail: bEmail];
```

```

[card3 setName: cName andEmail: cEmail];
[card4 setName: dName andEmail: dEmail];

// 将地址卡片添加到地址簿

[myBook addCard: card1];
[myBook addCard: card2];
[myBook addCard: card3];
[myBook addCard: card4];

NSLog(@"Entries in address book after adding cards: %i",
      [myBook entries]);

// 列出地址簿的所有条目

[myBook list];
}
return 0;
}

```

代码清单 15-12 输出

```

Entries in address book after creation: 0
Entries in address book after adding cards: 4

===== Contents of: Linda's Address Book =====
Julia Kochan      jewels337@axlc.com
Tony Iannino      tony.iannino@techfitness.com
Stephen Kochan    steve@classroomM.com
Jamie Baker      jbaker@classroomM.com
=====

```

这个程序创建了一个名为 **Linda's Address Book** 的新地址簿，然后建立了 4 个地址卡片，接着使用 **addCard:** 方法在地址簿中添加了 4 个卡片，使用 **list** 方法列出地址簿的内容，并进行校验。

还有一种很有用的枚举数组元素的方法，通过调用 **enumerateObjectsUsingBlock:**。类似于快速枚举，数组中的每个元素传递给块作为参数。此外，通过块还可以获取到数组中对象的索引号，和一个用来停止枚举执行 **BOOL** 变量的指针（见第 13 章）。

下面是将前面代码中的方法使用这种新方法改写的：

```

-(void) list {
NSLog(@"===== Contents of: %@ =====", bookName);

```

```

[self enumerateObjectsUsingBlock:
 ^ (AddressCard *theCard, NSUInteger idx, BOOL *stop) {
     NSLog ("%s %s", [theCard.name UTF8String],
             [theCard.email UTF8String]);
     NSLog ("%s", [theCard.name UTF8String]);
 }];
}

```

不必担心这个方法的语法。对于 `AddressBook` 中的每一个 `AddressCard`，随着块的执行，可获取到卡片，数组中的索引号，用来停止枚举的指针。在这里，我们忽略最后一个参数（以及索引号），因为我们希望列出通讯簿中的每个条目。

5. 在地址簿中查询某人

当你有一本容量较大的地址簿，并希望每次查询都不列出簿中的所有内容时，可增加一个很有意义的方法，将这个方法称为 `lookup:`，把需要查找的姓名作为参数。这个方法会搜索整个地址簿进行匹配寻找（忽略大小写），如果匹配成功，则返回这个记录。如果电话簿中不存在需要查找的姓名，则返回 `nil`。

以下是新的 `lookup:` 方法。

// 通过名称查找地址卡片——假定精确匹配

```

-(AddressCard *) lookup: (NSString *) theName
{
    for ( AddressCard *nextCard in book )
        if ( [nextCard.name caseInsensitiveCompare: theName] == NSOrderedSame )
            return nextCard;

    return nil;
}

```

该方法声明放在接口文件中，将定义放在实现文件中，就可以编写一个程序测试这个新方法。代码清单 15-13 就是这个程序，后面是输出结果。

代码清单 15-13 测试程序

```

#import "AddressBook.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSString *aName = @"Julia Kochan";
        NSString *aEmail = @"jewels337@axlc.com";
        NSString *bName = @"Tony Iannino";
    }
}

```

```
NSString *bEmail = @"tony.iannino@techfitness.com";
NSString *cName = @"Stephen Kochan";
NSString *cEmail = @"steve@classroomM.com";
NSString *dName = @"Jamie Baker";
NSString *dEmail = @"jbaker@classroomM.com";
AddressCard *card1 = [[AddressCard alloc] init];
AddressCard *card2 = [[AddressCard alloc] init];
AddressCard *card3 = [[AddressCard alloc] init];
AddressCard *card4 = [[AddressCard alloc] init];

AddressBook *myBook = [[AddressBook alloc]
                        initWithName: @"Linda's Address Book"];
AddressCard *myCard;

// 现在创建 4 个地址卡片

[card1 setName: aName andEmail: aEmail];
[card2 setName: bName andEmail: bEmail];
[card3 setName: cName andEmail: cEmail];
[card4 setName: dName andEmail: dEmail];

// 将一些卡片添加到地址簿

[myBook addCard: card1];
[myBook addCard: card2];
[myBook addCard: card3];
[myBook addCard: card4];

// 通过名字查找一个人

NSLog(@"Stephen Kochan");
myCard = [myBook lookup: @"stephen kochan"];

if (myCard != nil)
    [myCard print];
else
    NSLog(@"Not found!");

// 尝试另一种查找

NSLog(@"Haibo Zhang");
myCard = [myBook lookup: @"Haibo Zhang"];

if (myCard != nil)
    [myCard print];
else
    NSLog(@"Not found!");
}
```



```

    return 0;
}

```

代码清单 15-13 输出

```
Lookup: Stephen Kochan
```

```

=====
|                                     |
| Stephen Kochan                     |
| steve@classroomM.com               |
|                                     |
|                                     |
|                                     |
|      O      O                      |
|                                     |
=====

```

```
Lookup: Haibo Zhang
```

```
Not found!
```

通过 `lookup:` 方法在地址簿中查找到 **Stephen Kochan**（注意，我们匹配时并未区分大小写）时，该方法匹配到卡片后会立即执行 `return` 语句，使循环终止并返回 `nextCard` 对象的值，使用 `AddressCard` 的 `print` 方法显示结果。在第二次查询时，没有找到姓名 **Haibo Zhang**，因此，返回了上面的结果信息。

这个 `lookup:` 方法非常简单，因为必须找到整个 `name` 的精确匹配。更好的方法可以实现部分匹配，也可以处理多重匹配。比如，记录 **Steve Kochan**、**Fred Stevens** 和 **Steven Levy** 都可以满足消息表达式

```
[myBook lookup: @"steve"]
```

的匹配条件。因为可能存在多重匹配，所以有效的方法是创建一个包含所有匹配的数组，并将数组返回给方法的调用者（参见本章的练习 2），语句如下：

```
matches = [myBook lookup: @"steve"];
```

6. 从地址簿中删除某人

如果地址簿管理程序仅具有添加记录功能，却不能删除记录，那么它是不完整的。你可以构造一个 `removeCard:` 方法，将指定的 `AddressCard` 从地址簿中删除，或创建一个 `remove:` 方法，根据名字删除记录（见本章的练习 6）。

因为已经对接口文件做了几次改动，所以代码清单 15-14 是包含了新的 `removeCard:` 方法的接口文件。以下是新方法的实现。

代码清单 15-14 Addressbook.h 接口文件

```
#import "AddressCard.h"

@interface AddressBook: NSObject

@property (nonatomic, copy) NSString *bookName;
@property (nonatomic, strong) NSMutableArray *book;

-(instancetype) initWithName: (NSString *) name;

-(void) addCard: (AddressCard *) theCard;
-(void) removeCard: (AddressCard *) theCard;

-(AddressCard *) lookup: (NSString *) theName;
-(int) entries;
-(void) list;

@end
```

下面是新的 `removeCard:` 方法。

```
-(void) removeCard: (AddressCard *) theCard
{
    [book removeObjectIdenticalTo: theCard];
}
```

关于什么是同一对象，我们的观点是对象在内存中位于同一位置。所以，当两个包含相同信息的地址卡片对象处于不同的内存单元时（比如，复制 `AddressCard` 对象时，就会出现这种情况），`removeObjectIdenticalTo:` 方法并不把它们视为同一对象。

顺便提一句，`removeObjectIdenticalTo:` 方法会删除和参数相同的所有对象。但只有一个对象在数组中出现多次，才会遇到这个问题。

在使用 `removeObject:` 方法时，使用自己的方法处理相等的对象更加合理，然后编写自己的 `isEqual:` 方法判断两个对象是否相同。当使用 `removeObject:` 方法时，系统会自动针对数组中的每个元素调用 `isEqual:` 方法对两个元素进行比较。这个例子中，因为你的地址簿包含 `AddressCard` 对象作为成员，必须将 `isEqual:` 方法添加到类中（应该覆盖从 `NSObject` 继承的方法），这样可以自己决定如何确定等同性。比较相应的 `name` 字段和 `email` 字段是有意义的，如果都相等，则可以从方法返回 `YES`；否则，返回 `NO`。方法可以这样写：

```

-(BOOL) isEqual: (AddressCard *) theCard
{
    if ([name isEqualToString: theCard.name] == YES &&
        [email isEqualToString: theCard.email] == YES)
        return YES;
    else
        return NO;
}

```

下面应该注意 NSArray 方法，比如 containsObject: 方法和 indexOfObject: 方法都依赖 isEqual: 策略来决定两个对象是否相等。

代码清单 15-14 测试了新的 removeCard: 方法。

代码清单 15-14 测试程序

```

#import "AddressBook.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSString *aName = @"Julia Kochan";
        NSString *aEmail = @"jewls337@axlc.com";
        NSString *bName = @"Tony Iannino";
        NSString *bEmail = @"tony.iannino@techfitness.com";
        NSString *cName = @"Stephen Kochan";
        NSString *cEmail = @"steve@classroomM.com";
        NSString *dName = @"Jamie Baker";
        NSString *dEmail = @"jbaker@classroomM.com";

        AddressCard *card1 = [[AddressCard alloc] init];
        AddressCard *card2 = [[AddressCard alloc] init];
        AddressCard *card3 = [[AddressCard alloc] init];
        AddressCard *card4 = [[AddressCard alloc] init];

        AddressBook *myBook = [[AddressBook alloc]
                                initWithName: @"Linda's Address Book"];

        AddressCard *myCard;

        // 创建 4 个地址卡片

        [card1 setName: aName andEmail: aEmail];
        [card2 setName: bName andEmail: bEmail];
        [card3 setName: cName andEmail: cEmail];
        [card4 setName: dName andEmail: dEmail];
    }
}

```

```

// 将一些卡片添加到地址簿

[myBook addCard: card1];
[myBook addCard: card2];
[myBook addCard: card3];
[myBook addCard: card4];

// 通过名字查找一个人

NSLog(@"Lookup: Stephen Kochan");
myCard = [myBook lookup:@"Stephen Kochan"];

if (myCard != nil)
    [myCard print];
else
    NSLog(@"Not found!");

// 从电话簿中删除条目

[myBook removeCard: myCard];
[myBook list];    // 验证它消失了
}

return 0;
}

```

代码清单 15-14 输出

Lookup: Stephen Kochan

```

=====
|                               |
| Stephen Kochan               |
| steve@classroomM.com        |
|                               |
|                               |
|                               |
|      O      O               |
|                               |
=====

```

===== Contents of: Linda's Address Book =====

```

Julia Kochan    jewls337@axlc.com
Tony Iannino    tony.iannino@techfitness.com
Jamie Baker     jbakker@classroomM.com
=====

```

在地址簿中查询 Stephen Kochan，验证它确实出现在地址簿中，则将结果 AddressCard 传递给 removeCard:方法进行移除。最后列出地址簿，结果说明已

经删除成功。

15.3.2 数组排序

如果地址簿包含大量记录，那么按字母排序可能会很方便。在 `AddressBook` 类中增加 `sort` 方法，利用 `NSMutableArray` 类中 `sortUsingSelector:` 的方法可以很容易实现这项功能。`sort` 方法使用 `selector` 作为参数，`sortUsingSelector:` 方法会使用这个 `selector` 比较两个元素。由于数组可以包含任何类型的对象，所以要实现一般的排序方法，唯一途径就是由你来判定数组中的元素是否有序。为此，你必须添加一个方法比较数组中的两个元素。^④这个方法返回的结果是 `NSComparisonResult` 类型的值。如果希望排序方法将第一个元素放在第二个元素之前，那么方法的返回值应是 `NSOrderedAscending`。如果认为这两个元素相等，那么返回 `NSOrderedSame`。如果排序后的数组中，第一个元素应放在第二个元素之后，那么返回 `NSOrderedDescending`。

首先，下面是 `AddressBook` 类中新的 `sort` 方法：

```
-(void) sort
{
    [book sortUsingSelector: @selector(compareNames)];
}
```

在第9章“多态、动态类型和动态绑定”中学过，表达式

```
@selector (compareNames:)
```

利用指定的方法名创建一个 `SEL` 类型的 `selector`。使用方法 `sortUsingSelector:` 比较数组中的两个元素，在需要进行比较的时候，它会调用指定的方法向数组中的第一个元素（接收者）发送消息，与它的参数进行比较。前面描述过，返回值为 `NSComparisonResult` 类型。

因为地址簿的元素是 `AddressCard` 对象，还需要向 `AddressCard` 类添加比较方法。回到 `AddressCard` 类，为其添加 `compareNames:` 方法。下面给出具体实现。

// 比较指定的地址卡片中的两个名字

```
-(NSComparisonResult) compareNames: (id) element
{
```

④ 调用方法 `sortUsingFunction:context:` 时你需要提供一个函数而不是一个方法进行比较。

```

    return [name compare: [element name]];
}

```

因为是地址簿中两个名字字符串的比较，可以用 NSString 类的 compare: 方法实现这个功能。

如果为 AddressBook 类添加 sort 方法，并向 AddressCard 类添加 compareNames: 方法，可以编写程序进行测试（参见代码清单 15-15）。

代码清单 15-15 测试程序

```

#import "AddressBook.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSString *aName = @"Julia Kochan";
        NSString *aEmail = @"jewels337@axlc.com";
        NSString *bName = @"Tony Iannino";
        NSString *bEmail = @"tony.iannino@techfitness.com";
        NSString *cName = @"Stephen Kochan";
        NSString *cEmail = @"steve@classroomM.com";
        NSString *dName = @"Jamie Baker";
        NSString *dEmail = @"jbaker@classroomM.com";

        AddressCard *card1 = [[AddressCard alloc] init];
        AddressCard *card2 = [[AddressCard alloc] init];
        AddressCard *card3 = [[AddressCard alloc] init];
        AddressCard *card4 = [[AddressCard alloc] init];

        AddressBook *myBook = [AddressBook alloc];

        //首先创建 4 个地址卡片

        [card1 setName: aName andEmail: aEmail];
        [card2 setName: bName andEmail: bEmail];
        [card3 setName: cName andEmail: cEmail];
        [card4 setName: dName andEmail: dEmail];

        myBook = [myBook initWithName: @"Linda's Address Book"];

        //将一些卡片添加到地址簿

        [myBook addCard: card1];
        [myBook addCard: card2];
        [myBook addCard: card3];
    }
}

```

```

[myBook addCard: card4];

//列出未排序的地址簿

[myBook list];

//对其进行排序并再次列出它

[myBook sort];
[myBook list];
}
return 0;
}

```

代码清单 15-15 输出

```

===== Contents of: Linda's Address Book =====
Julia Kochan      jewels337@axlc.com
Tony Iannino      tony.iannino@techfitness.com
Stephen Kochan    steve@classroomM.com
Jamie Baker      jbaker@classroomM.com
=====

===== Contents of: Linda's Address Book =====
Jamie Baker      jbaker@classroomM.com
Julia Kochan      jewels337@axlc.com
Stephen Kochan    steve@classroomM.com
Tony Iannino      tony.iannino@techfitness.com
=====

```

请注意，排列是升序的。但是，可以很容易修改 `AddressCard` 类中的 `compareNames:` 方法，使返回的值呈降序排列。

1. 使用区块（Block）排序

`NSArray` 和 `NSMutableArray` 类中具有使用区块对数组中元素进行排序的方法。

`NSArray` 的排序方法一般格式为：

```
-(NSArray *) sortedArrayUsingComparator: (NSComparator) block
```

`NSMutableArray` 的排序方法格式为：

```
-(void) sortUsingComparator: (NSComparator) block
```

`NSComparator` 作为 `typedef` 定义在系统头文件中：

```
typedef NSComparisonResult (^NSComparator)(id obj1, id obj2);
```

`NSComparator` 是一个区块，使用两个对象作为参数，并返回 `NSComparisonResult` 类型的值。这个方法因为使用了区块，对大数组排序可能会快一些。如果为了速度，可以考虑在程序中使用。

区块将需要比较的两个对象作为参数，预期会返回一个标识，说明第一个对象是否小于、等于或者大于第二个对象。这与不使用区块的数组排序方法是一致的。

这个区块作为参数传递给 `sortUsingComparator:` 方法，它可以简单地调用 `compareNames:` 方法对地址卡片进行比较。

```
-(void) sort
{
    [book sortUsingComparator:
        ^(id obj1, id obj2) {
            return [obj1 compareNames: obj2];
        } ];
}
```

这样能够运行，但是并没有提升性能，因为区块调用 `compareNames:` 方法和 `sortUsingSelector:` 方法一样。更好的方法是在区块中做更多的工作，使性能得到提升：

```
-(void) sort
{
    [book sortUsingComparator:
        ^(id obj1, id obj2) {
            return [[obj1 name] compare: [obj2 name]];
        } ];
}
```

回到代码清单 15-15，使用刚才开发的方法替换 `AddressBook` 的 `sort` 方法。验证程序是否能够正常运行，是否能够正确排列地址簿。

使用带有区块的 `sort` 方法有一个好处，即不必为需要比较的对象添加比较方法。在之前版本的 `sort` 方法中，添加过一个 `compareNames:` 方法到 `AddressCard` 类。

另一个好处在于，如果需要改变进行比较的地址卡片的方式，只需要直接修改 `sort` 方法，并不需要对 `AddressCard` 类进行更改。

数组对象的处理方法有 50 多个，表 15.4 和表 15.5 分别列出了不变数组和

可变数组的常用方法。因为 `NSMutableArray` 类是 `NSArray` 类的子类, 所以前者继承了后者的方法。

表 15.4 和表 15.5 中的 `obj`、`obj1` 和 `obj2` 是任意对象, `i` 是呈现数组中有效索引号的 `NSUInteger` 整数, `selector` 是 SEL 类型的 selector 对象, `size` 是一个 `NSUInteger` 整数。

表 15.4 常用的 NSArray 方法

方 法	描 述
<code>+(instancetype) arrayWithObjects: obj1, obj2, ... nil</code>	创建一个新数组, <code>obj1</code> 、 <code>obj2</code> , ... 是其元素
<code>-(BOOL) containsObject: obj</code>	确定数组中是否包含对象 <code>obj</code> (使用 <code>isEqual:</code> 方法)
<code>-(NSUInteger) count</code>	数组中元素的个数
<code>-(NSUInteger) indexOfObject: obj</code>	第一个包含对象 <code>obj</code> 的元素索引号 (使用 <code>isEqual:</code> 方法)
<code>-(NSUInteger) indexOfObjectPassingTest: (BOOL (^)(id obj, NSUInteger idx, BOOL *stop)) block</code>	传递每个对象 <code>obj</code> (带有索引号 <code>idx</code>) 到区块 <code>block</code> 中, 如果 <code>obj</code> 通过测试返回 YES, 未通过返回 NO, 设置变量指针 <code>stop</code> 为 YES 结束处理
<code>-(id) lastObject</code>	返回数组最后的对象
<code>-(id) objectAtIndex: i</code>	存储在元素 <code>i</code> 的对象
<code>-(void) makeObjectsPerform Selector: (SEL) selector</code>	将 <code>selector</code> 指示的消息发送给数组中的每个元素
<code>-(void) enumerateObjectsUsingBlock: (void (^)(id obj, NSUInteger idx, BOOL *stop)) block</code>	通过块获取数组中的每个元素, 包括对象 <code>obj</code> 和对应的索引号 <code>idx</code> , 只有当所有的元素都遍历完成或设置变量指针 <code>stop</code> 为 YES 才处理结束
<code>-(NSArray *) sortedArrayUsingSelector: (SEL) selector</code>	根据指定 <code>selector</code> 方法的比较器对数组进行排序
<code>-(NSArray *) sortedArrayUsingComparator: (NSComparator) block</code>	通过执行区块 <code>block</code> 对数组进行排序
<code>-(BOOL) writeToFile: path atomically: (BOOL) flag</code>	将数组写入指定的文件中, 如果 <code>flag</code> 为 YES, 则先创建一个临时文件

表 15.5 常用的 NSMutableArray 方法

方 法	描 述
<code>+(instancetype) array</code>	创建一个空数组
<code>+(instancetype) arrayWithCapacity: size</code>	使用指定的初始 <code>size</code> 创建一个数组
<code>-(instancetype) initWithCapacity: size</code>	使用指定的初始 <code>size</code> 初始化新分配的数组
<code>-(void) addObject: obj</code>	将对象 <code>obj</code> 添加到数组的末尾
<code>-(void) insertObject: obj atIndex: i</code>	将对象 <code>obj</code> 插入数组的 <code>i</code> 元素

续表

方 法	描 述
-(void) replaceObjectAtIndex: <i>i</i> withObject: <i>obj</i>	将数组中序号为 <i>i</i> 的对象用对象 <i>obj</i> 替换
-(void) removeObject: <i>obj</i>	从数组中删除所有的 <i>obj</i>
-(void) removeObjectAtIndex: <i>i</i>	从数据中删除元素 <i>i</i> ，将序号为 <i>i</i> +1 的对象移至数组的结尾
-(void) sortUsingSelector: (SEL) <i>selector</i>	用 <i>selector</i> 指定的比较方法将数组排序
-(void) sortUsingComparator: (NSComparator) <i>block</i>	通过执行区块 <i>block</i> 对数组进行排序

2. NSValue 类

曾经提过，像数组这样的 Foundation 集合只能存储对象，不能存储像 `int` 这样的基本数据类型。为了解决这个问题，需要使用 `NSNumber` 对象数组，而不是 `int` 数组。

在 iOS 程序开发时，还需要在集合中存储其他类型的数据。这些类型源于 C 语言的一种数据类型，它不是对象。例如，你会在程序中用 `CGPoint`（typedef 名）定义 (x, y) 坐标指定一个矩形的原点。事实上，矩形是 `CGRect` 类型，这种结构本身包含两种结构：`CGPoint` 结构定义矩形的原点，`CGSize` 结构定义矩形的宽和高。

上面的讨论说明需要将结构存储在集合中，但这又不能直接做到。`NSValue` 类正好可以将结构转化为对象，并把它存储在集合中。这种将结构转化为对象的方式，简称为包装（`wrapping`），逆向的处理是从对象中解出基本类型，简称展开（`unwrapping`）。

表 15.6 显示的是包装 C 数据类型到对象对应的方法和展开对象相应的逆向方法。更多信息可以查询 `NSValue` 类文档。

表 15.6 NSValue 包装和展开方法

Typedef 数据类型	描 述	包装方法	展开方法
<code>CGPoint</code>	x 和 y 值组成的点	<code>valueWithPoint:</code>	<code>pointValue</code>
<code>CGSize</code>	宽和高组成的尺寸	<code>valueWithSize:</code>	<code>sizeValue</code>
<code>CGRect</code>	矩形包含原点和尺寸	<code>valueWithRect:</code>	<code>rectValue</code>
<code>NSRange</code>	描述位置和大小范围	<code>valueWithRange:</code>	<code>rangeValue</code>

以下代码块采用 `CGPoint` 结构，将它加入到可变数组 `touchPoints` 中：

```

CGPoint myPoint;
NSValue *pointObj;
NSMutableArray *touchPoints = [NSMutableArray array];
...
myPoint.x = 100;      // 设置点为 (100, 200)
myPoint.y = 200;
...
pointObj = [NSValue valueWithPoint: myPoint]; // 使之成为对象
[touchPoints addObject: pointObj];

```

相信你已经理解了这样做的必要性。因为 `myPoint` 是一个结构，并不能直接将它存储在 `touchPoints` 数组中，因此，需要先将它转化为一个对象，可以使用 `valueWithPoint:` 方法。

如果希望从数组 `touchPoints` 中取出最后一个点并将它转化成 `CGPoint`，下列语句可以做到：

```
myPoint = [[touchPoints lastObject] pointValue];
```

15.4 词典对象

词典（dictionary）是由键-对象对组成的数据集合。与在词典中查找单词定义一样，可以通过对象的键从 Objective-C 词典中获取需要的值（即那个对象）。词典中的键必须是单值的，通常它们是字符串，但也可以是其他对象类型。和键关联的值可以是任何对象类型，但不能是 `nil`。

词典可以是固定的，也可以是可变的。可变词典中的记录可以动态添加和删除。可以使用键检索词典，也可以枚举它们的内容。代码清单 15-16 表示创建一个可变词典实现 Objective-C 的术语表，前 3 条记录已经添加到词典。

代码清单 15-16

```

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSMutableDictionary *glossary = [NSMutableDictionary dictionary];

        // 存储 3 个条目在类别中

        [glossary setObject: @"A class defined so other classes can inherit from it"
            forKey: @"abstract class" ];
    }
}

```

```

[glossary setObject: @"To implement all the methods defined in a protocol"
                  forKey: @"adopt"];
[glossary setObject: @"Storing an object for later use"
                  forKey: @"archiving"];

// 检索并显示它们

NSLog(@"abstract class: %@", [glossary objectForKey: @"abstract class"]);
NSLog(@"adopt: %@", [glossary objectForKey: @"adopt"]);
NSLog(@"archiving: %@", [glossary objectForKey: @"archiving"]);
}
return 0;
}

```

代码清单 15-16 输出

```

abstract class: A class defined so other classes can inherit from it
adopt: To implement all the methods defined in a protocol
archiving: Storing an object for later use

```

表达式

```
[NSMutableDictionary dictionary]
```

创建了一个空的可变词典。使用 `setObject:forKey:` 方法将键-值对添加到词典中。生成词典之后，使用 `objectForKey:` 方法检索键的值。代码清单 15-16 表示如何检索和显示 Objective-C 术语表中的 3 条记录。实际的应用程序中，当用户输入他需要的单词后，程序会搜索术语并寻找它的定义。

值得注意的是，最近的一些语法已经被补充到 Objective-C 语言有关词典的使用上了。表达式

```
dict[key]
```

等同于

```
[dict objectForKey: key]
```

存储键-值对到词典中，表达式

```
dict[key] = object
```

等同于

```
[dict setObject: object forKey: key]
```

代码清单 15-17 已经重写了代码清单 15-16 中的语法，使用新的语法在词

典中添加条目和获取条目。

代码清单 15-17

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

int main (int argc, char * argv[]) {
    @autoreleasepool {
        NSMutableDictionary *glossary = [NSMutableDictionary dictionary];
        // Store three entries in the glossary
        glossary[@"abstract class"] = @"A class defined so other classes can inherit from it";
        glossary[@"adopt"] = @"To implement all the methods defined in a protocol";
        glossary[@"archiving"] = @"Storing an object for later use"; // Retrieve and display them
        NSLog(@"abstract class: %@", glossary[@"abstract class"]);
        NSLog(@"adopt: %@", glossary[@"adopt"]);
        NSLog(@"archiving: %@", glossary[@"archiving"]);
    }
    return 0;
}
```

15.4.1 枚举词典

代码清单 15-18 使用 `dictionaryWithObjectsAndKeys:` 方法创建带有初始键-值对的词典，这样就创建了一个不可变词典。程序还演示了如何利用快速枚举进行循环，一次一个键检索词典各个元素。和数组对象不一样，词典对象是无序的。所以，当枚举词典时，第一个放到词典中的键-对象对并不一定是第一个取出。

代码清单 15-18

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSDictionary *glossary =
            [NSDictionary dictionaryWithObjectsAndKeys:
             @"A class defined so other classes can inherit from it",
             @"abstract class",
             @"To implement all the methods defined in a protocol",
             @"adopt",
             @"Storing an object for later use",
             @"archiving",
             nil
            ];
    }
}
```

```

];

// 打印词典中所有的键-值对

for ( NSString *key in glossary )
    NSLog (@"%@: %@", key, [glossary objectForKey: key]);

}

return 0;
}

```

代码清单 15-18 输出

```

abstract class: A class defined so other classes can inherit from it
adopt: To implement all the methods defined in a protocol
archiving: Storing an object for later use

```

dictionaryWithObjectsAndKeys:的参数是对象-键对的数组（是的，就是这种顺序），每个对象-键用逗号隔开。数组必须以 `nil` 对象结束。

创建词典后，利用循环语句枚举词典的内容。键是从词典中依次检索的，没有特定顺序。

你还可以使用特殊的语法

```
@{ key1 : object1, key2 : object2, ..., keyn : object }
```

若需要创建一个不可变的字典，并列出键/对象对为它初始化，可以像代码清单 15-18 中那样进行创建：

```

NSDictionary *glossary = @{ @"abstract class" :
    @"A class defined so other classes can inherit from it", @"adopt" : @"To implement all
    the methods defined in a protocol", @"archiving" : @"Storing an object for later use"
};

```

如果需要以字母顺序显示词典中的内容，可以先取出词典中的所有键，对它们排序，然后按照排好序的键从词典中取出所有的值。使用 `NSDictionary` 的 `allKeys` 方法从词典中抽取出所有的键组成一个数组并进行排序，然后枚举排序过的键组成的数组，从字典中获取相应的值。假设你有一个名为 `states` 的词典，它包含美国每个州的名称，把对应的州政府所在地作为值。以下是一个代码片段，按字母顺序显示每个州的名称以及相应的州政府所在地：

```

NSArray *keys = [states allKeys];

keys = [keys sortedArrayUsingComparator:

```

```
^(id obj1, id obj2) {
    return [obj1 compare: obj2];
}];
```

```
for (NSString *aState in keys)
    NSLog(@"State: %@ Capital: %@", aState, [states objectForKey: aState]);
```

在此已经演示了词典的一些基本操作。表 15.7 和表 15.8 分别总结了不变和可变词典常用的一些方法。因为 `NSMutableDictionary` 类是 `NSDictionary` 类的子类，所以它继承了 `NSDictionary` 类的方法。

表 15.7 和表 15.8 中的 `key`、`key1`、`key2`、`obj`、`obj1` 和 `obj2` 是任意对象，`size` 是一个 `NSUInteger` 无符号整数。

表 15.7 常用的 NSDictionary 方法

方 法	描 述
<code>+(instancetype) dictionaryWithObjectsAndKeys: obj1, key1, obj2, key2, ..., nil</code>	使用键-对象对 <code>{key1,obj1}</code> 、 <code>{key2,obj2}</code> 、... 创建词典
<code>-(instancetype) initWithObjectsAndKeys: obj1, key1, obj2, key2, ..., nil</code>	将新分配的词典初始化为键-对象对 <code>{key1, obj1}</code> 、 <code>{key2, obj2}</code>
<code>-(NSArray *) allKeys</code>	返回一个数组包含词典中所有的键
<code>-(NSUInteger) count</code>	返回词典中的记录数
<code>-(NSEnumerator *) keyEnumerator</code>	为词典中所有的键返回一个 <code>NSEnumerator</code> 对象
<code>-(NSArray *) keysSortedByValueUsingSelector: (SEL) selector</code>	返回词典中的键数组，它根据 <code>selector</code> 指定的比较方法进行排序
<code>-(NSEnumerator *) objectEnumerator</code>	为词典中的所有值返回一个 <code>NSEnumerator</code> 对象
<code>-(id) objectForKey: key</code>	返回指定 <code>key</code> 的对象
<code>-(void) enumerateKeysAndObjectsUsingBlock: (void(^)(id key, id obj, BOOL *stop)) block</code>	通过块获取词典中的每个条目，条目包括键-对象对，只有当所有的元素都遍历完成或设置变量指针 <code>stop</code> 为 YES 才处理结束

表 15.8 常用的 NSMutableDictionary 方法

方 法	描 述
<code>+(instancetype) dictionaryWithCapacity: size</code>	使用一个初始指定的 <code>size</code> 创建可变词典
<code>-(instancetype) initWithCapacity: size</code>	将新分配的词典初始化为指定的 <code>size</code>
<code>-(void) removeAllObjects</code>	删除词典中所有的记录
<code>-(void) removeObjectForKey: key</code>	删除词典中指定 <code>key</code> 对应的记录
<code>-(void) setObject: obj forKey: key</code>	向词典为 <code>key</code> 的键添加 <code>obj</code> ，如果 <code>key</code> 已存在，则替换该值

15.5 集合对象

set 是一组单值对象集合，它可以是可变的，也可以是不变的。操作包括：搜索、添加、删除集合中的成员（仅用于可变集合），比较两个集合，计算两个集合的交集和并集等。

在本节中会遇到 3 个类：NSSet、NSMutableSet 和 NSIndexSet，还会提到 NSCountedSet 类，有可能今后需要用到这种集合。

代码清单 15-19 演示了集合的一些基本操作。假如你需要在程序执行过程中多次显示集合的内容，因此，决定创建一个名为 `print` 的新方法。通过创建一个名为 `Printing` 的新分类，将 `print` 方法加入到 `NSSet` 类中。因为 `NSMutableSet` 类是 `NSSet` 类的子类，可变集合也可以使用这个新的 `print` 方法。

代码清单 15-19

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

// 使用 Printing 类别，将打印方法添加到 NSSet

@interface NSSet (Printing)
-(void) print;
@end

@implementation NSSet (Printing)
-(void) print {
    printf ("{ ");

    for (NSNumber *element in self)
        printf (" %li ", (long) [element integerValue]);

    printf ("} \n");
}
@end

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

        NSMutableSet *set1 = [NSMutableSet setWithObjects:
            @1, @3, @5, @10, nil];
        NSSet *set2 = [NSSet setWithObjects:
            @-5, @100, @3, @5, nil];
        NSSet *set3 = [NSSet setWithObjects:
```



```

        @12, @200, @3, nil];

    NSLog(@"set1: ");
    [set1 print];
    NSLog(@"set2: ");
    [set2 print];

    // 相等性测试
    if ([set1 isEqualToSet: set2] == YES)
        NSLog(@"set1 equals set2");
    else
        NSLog(@"set1 is not equal to set2");

    // 成员测试

    if ([set1 containsObject: @10] == YES)
        NSLog(@"set1 contains 10");
    else
        NSLog(@"set1 does not contain 10");

    if ([set2 containsObject: @10] == YES)
        NSLog(@"set2 contains 10");
    else
        NSLog(@"set2 does not contain 10");

    // 在可变集合 set1 中添加和移除对象

    [set1 addObject: @4];
    [set1 removeObject: @10];
    NSLog(@"set1 after adding 4 and removing 10: ");
    [set1 print];

    // 获得两个集合的交集

    [set1 intersectSet: set2];
    NSLog(@"set1 intersect set2: ");
    [set1 print];

    // 两个集合的并集

    [set1 unionSet: set3];
    NSLog(@"set1 union set3: ");
    [set1 print];

}

return 0;
}

```

代码清单 15-19 输出

```

set1:
{ 3 10 1 5 }
set2:
{ 100 3 -5 5 }
set1 is not equal to set2
set1 contains 10
set2 does not contain 10
set1 after adding 4 and removing 10:
{ 3 1 5 4 }
set1 intersect set2:
{ 3 5 }
set1 union set3:
{ 12 3 5 200 }

```

`print` 方法使用了前面描述的快速枚举技术，从集合中提取每个元素。当然，因为你的 `print` 方法只适用于元素类型为整型的集合，所以它并不通用。这里有一个很好的提示，提醒我们如何通过创建分类将方法添加到类中^⑤（注意：在 `print` 方法中，使用了 C 库中的 `printf` 函数显示集合中的每个元素）。

`setWithObjects:` 方法以一个 `nil` 结尾的对象数组创建新集合。当创建了 3 个集合后，程序使用新的 `print` 方法显示前两个集合的内容。然后用 `isEqualToSet:` 方法测试集合 `set1` 和 `set2` 是否相等，结果是它们并不相等。

使用 `containsObject:` 方法查询整数 10 是否在集合 `set1` 中，然后是否在集合 `set2` 中。方法返回的 `Boolean` 值验证了整数 10 在集合 `set1` 中，但不在 `set2` 中。

接下来，使用 `addObject:` 方法和 `removeObject:` 方法在集合 `set1` 中添加 4 和删除 10，显示的集合内容说明操作成功。

使用 `intersect:` 和 `union:` 方法计算两个集合的交集和并集。在这两种情况中，运算的结果取代了消息的接收者。

`Foundation` 框架同样提供了一个名为 `NSCountedSet` 的类。这种集合同一对象可以出现多次，然而并非在集合中存放多次这个对象，而是维护一个次数计数。当第一次将对象添加到集合中时，对象的 `count` 值被置为 1，然后每次将这

⑤ 更为通用的方法是通过调用每个对象的描述方法来显示集合中的每个成员。对象集包含任何类型的对象，均可以显示为可阅读的格式。此外请注意，你可以调用一次 `NSLog` 显示任何集合的内容，使用 “%@” 作为“打印对象”的格式字符。

个对象添加到集合中, `count` 值就会增 1, 每次从集合删除对象, `count` 值就会减 1。当对象的 `count` 值为零时, 实际上对象本身就被删除了。使用方法 `countForObject`: 可以在集合中检索某个对象的 `count` 值。

计数集合的一个应用是单词计数器。每次从文本中发现一个单词, 就将它添加到计数集合中, 扫描完文本之后, 就可以从集合检索出每个单词和计数, 单词的计数表明在文本中出现的次数。

这里只有集合的一些基本操作。表 15.9 和表 15.10 分别总结了不变和可变集合的一些常用方法。因为 `NSMutableSet` 类是 `NSSet` 类的子类, 所以它继承了 `NSSet` 类的方法。

在表 15.9 和表 15.10 中, `obj`、`obj1` 和 `obj2` 是任意对象, `nsset` 是 `NSMutableSet` 类或 `NSSet` 类的对象, `size` 是一个 `NSUInteger` 整数。

表 15.9 常用的 `NSSet` 方法

方 法	描 述
<code>+(instancetype) initWithObjects: obj1, obj2, ..., nil</code>	使用一系列对象创建新集合
<code>-(id) anyObject</code>	返回集合中任一对象
<code>-(instancetype) initWithObjects: obj1, obj2, ..., nil</code>	使用一系列对象初始化新分配的集合
<code>-(NSUInteger) count</code>	返回集合的成员个数
<code>-(BOOL) containsObject: obj</code>	确定集合是否包含 <code>obj</code>
<code>-(BOOL) member: obj</code>	使用 <code>isEqual:</code> 方法确定集合是否包含 <code>obj</code>
<code>-(NSEnumerator *) objectEnumerator</code>	为集合中的所有对象返回一个 <code>NSEnumerator</code> 对象
<code>-(BOOL) isSubsetOfSet: nsset</code>	确定接收者的每个成员是否都出现在 <code>nsset</code> 中
<code>-(BOOL) intersectsSet: nsset</code>	确定接收者中是否至少有一个成员出现在对象 <code>nsset</code> 中
<code>-(BOOL) isEqualToSet: nsset</code>	确定两个集合是否相等

表 15.10 常用的 `NSMutableSet` 方法

方 法	描 述
<code>-(void) addObject: obj</code>	将对象 <code>obj</code> 添加到集合中
<code>-(void) removeObject: obj</code>	从集合中删除对象 <code>obj</code>
<code>-(void) removeAllObjects</code>	删除接受者的所有成员
<code>-(void) unionSet: nsset</code>	将对象 <code>nsset</code> 的所有成员添加到接收者
<code>-(void) minusSet: nsset</code>	从接收者中删除 <code>nsset</code> 的所有成员
<code>-(void) intersectSet: nsset</code>	将接收者中所有不属于 <code>nsset</code> 的元素删除

15.5.1 NSMutableIndexSet

再来看另一种类型的集合：NSMutableIndexSet。这个类用于存储有序的索引到某种数据结构，比如数组。例如，使用这个类可以生成一份数组对象的索引号清单，这些对象满足指定的查询条件。需要注意的是，并没有可变版本的 NSMutableIndexSet 类。

NSMutableArray 的方法 indexOfObjectPassingTest: 以一个区块作为参数。针对数组中的每个元素对区块执行操作，传递数组的元素、索引号和一个指向 BOOL 变量的指针给区块。使用区块中的代码测试元素是否满足条件，如果数组中的元素满足条件，则返回 YES，不满足则返回 NO。即使方法返回 YES，仍然会继续执行，直到所有的元素均被处理过。可以设置 BOOL 指针引用的值为 YES 终止处理（参考第 13 章讨论的指针）。

如果 indexOfObjectPassingTest: 查找到一个匹配的元素（区块至少会返回一次 YES），则返回从集合匹配到的最小索引值；否则，返回 NSNotFound。

下面是对 AddressBook 类的 lookup: 方法进行修改。

```
-(AddressCard *) lookup: (NSString *) theName
{
    NSUInteger result = [book indexOfObjectPassingTest:
        ^ (id obj, NSUInteger idx, BOOL *stop)
        {
            if ([[obj name] caseInsensitiveCompare: theName] == NSOrderedSame) {
                *stop = YES; // 找到一个匹配，一个足够了
                return YES;
            }
            else
                return NO; // 继续查找
        }
    ];

    // 如果找到一个匹配，则查看它

    if (result != NSNotFound) // 只有一个元素
        return book[result];
    else
        return nil;
}
```

如果从数组中查找到匹配的所有对象，则需要将它们存储在区块的数组中，并且不要设置 stop 指针为 YES。以这种方式使用 indexOfObjectPassingTest: 方法

测试数组中所有的元素。这个方法语句如下：

```
-(NSMutableArray *) lookupAll: (NSString *) theName
{
    NSMutableArray *matches = [NSMutableArray array];

    NSUInteger result = [book indexOfObjectPassingTest:
        ^ (id obj, NSUInteger idx, BOOL *stop)
        {
            if ([[obj name] caseInsensitiveCompare: theName] == NSOrderedSame) {
                [matches addObject: obj];
                return YES;
            }
            else
                return NO;
        } ];

    // 如果找到匹配，则查看它

    if ([matches count])
        return matches;
    else
        return nil;
}
```

如果需要从数组中查找多个匹配的对象，也可以使用 `indexesOfObjectsPassingTest:` 方法。这个方法会返回 `NSIndexSet`，包含数组中满足条件的所有元素的索引。下面是使用新的方法查找并返回所有匹配地址卡片的索引号。

```
-(NSIndexSet *) lookupAll: (NSString *) theName
{
    NSIndexSet *result = [book indexesOfObjectsPassingTest:
        ^ (id obj, NSUInteger idx, BOOL *stop)
        {
            if ([[obj name] caseInsensitiveCompare: theName] == NSOrderedSame)
                return YES; // 找到一个匹配后，继续查找
            else
                return NO; // 继续查找
        } ];

    // 返回结果

    return result;
}
```

修改 `lookupAll:` 方法并返回匹配的地址卡片的数组，这个作为练习（提示：

在使用 `indexesOfObjectsPassingTest:` 方法后，迭代索引集合的每个索引，并添加相应的元素到数组中）。

表 15.11 列出了 `NSIndexSet` 的一些方法。其中，`idx` 是一个 `NSUInteger` 整型。你可以查看相关文档加深对这个类的了解。`NSOrderedSet` 和 `NSMutableOrderedSet` 是用来处理有序、不重复集合（类似于数组）的。

表 15.11 `NSIndexSet` 的一些方法

方 法	描 述
<code>+(instancetype) indexSet</code>	创建一个空的索引集合
<code>-(BOOL) containIndex: idx</code>	如果索引集合包含索引 <code>idx</code> ，则返回 YES，否则返回 NO
<code>-(NSUInteger) count</code>	返回索引集合中索引的数量
<code>-(NSUInteger) firstIndex</code>	返回集合中的第一个索引，如果集合为空，则返回 <code>NSNotFound</code>
<code>-(NSUInteger) indexLessThanIndex: idx</code>	返回集合中小于 <code>idx</code> 的最接近的索引，如果没有小于 <code>idx</code> 的索引，则返回 <code>NSNotFound</code> ，类似 <code>indexLessThanOrEqualToIndex:</code> 、 <code>indexGreaterThanIndex</code> 和 <code>indexGreaterThanOrEqualIndex:</code>
<code>-(NSUInteger) lastIndex</code>	返回集合的最后一个索引，如果集合为空，则返回 <code>NSNotFound</code>
<code>-(NSIndexSet *) indexesPassingTest: (BOOL) (^) (NSUInteger idx, BOOL *stop) block</code>	区块应用在集合中的每个元素。 <code>idx</code> 添加到了 <code>NSIndexSet</code> 中返回 YES，否则返回 NO；设置指针变量 <code>stop</code> 为 YES，表示中断处理

15.6 练习

1. 在文档中查找 `NSDate` 类，然后向该类添加一个名为 `ElapsedDays` 的新分类。在这个新分类中，使用以下方法声明添加一个新方法：

```
-(unsigned long) elapsedDays: (NSDate *) theDate;
```

让新方法返回接收者到方法的参数之间经过的天数，并编写一个测试程序测试新方法（提示：可参见 `years:months:days:hours:minutes:seconds:sinceDate:` 方法）。

2. 修改本章为类 `AddressBook` 编写的 `lookup:` 方法，使它能够找出匹配的 `name`。消息表达式 `[myBook lookup: @"steve"]` 匹配名称中任何位置包含字符串“steve”的记录。将 `sortedArrayUsingSelector:` 方法替换为本章最后出现的 `indexesOfObjectsPassingTest:` 方法。

3. 使用练习 2 的结果修改 `lookup:`方法，使它能够搜索地址簿，并找到所有匹配的地址卡片，返回值为包含所有匹配的地址卡片的数组，若匹配不成功，则返回 `nil`。使用 `sortedArrayUsingSelector:`方法替换为本章最后出现的 `indexesOfObjectsPassingTest:`方法（注意，本章最后的例子中返回的是 `NSIndexSet`，但是这里希望返回 `AddressCards` 数组）。
4. 在 `AddressCard` 类添加新字段，建议将 `name` 字段分隔成姓氏和名字字段，添加地址（可能包含单独的州、城市、邮编和国家字段）和电话号码字段。编写合适的 `setter` 和 `getter` 方法，确定 `print` 方法和 `list` 方法能够显示这些字段。
5. 完成练习 4 后，修改练习 3 的 `lookup:`方法，使它能够对地址卡片中所有的字段进行搜索。能否想出一种方式设计 `AddressCard` 类和 `AddressBook` 类，使得后者不必了解存储在前者中的所有字段。
6. 在 `AddressBook` 类中添加 `removeName:`方法，以便删除地址簿中的某条记录。给定以下声明：

```
-(BOOL) removeName: (NSString *) theName;
```

使用练习 2 中的 `lookup:`方法。如果名字未查找到或者查找到多条记录，则方法返回 `NO`。如果记录成功移除，则返回 `YES`。

7. 使用在第一部分定义的 `Fraction` 类，根据任意一些值创建一个分数数组，为 `Fraction` 添加 `description` 方法。使用 3 种不同的方法显示分数的值：1) 常规的 `for` 循环，2) 快速迭代，3) 使用 `%@`。
8. 使用第一部分中定义的 `Fraction` 类，根据任意一些值创建一个可变的分数数组。使用 `NSMutableArray` 类的 `sortUsingSelector:`方法给数组排序，向 `Fraction` 类添加一个 `Comparison` 类型，实现 `comparison` 方法。
9. 定义 3 个新类，分别命名为 `Song`、`Playlist` 和 `MusicCollection`。`Song` 对象包含歌曲的信息，如歌曲名、艺术家、专辑、歌曲长度等；`Playlist` 对象包含播放列表名称和一个歌曲的集合；`MusicCollection` 对象包含播放列表集合，包括一个名为 `library` 的主播放列表，这个列表包含歌曲

集合中的所有歌曲。定义上述的 3 个类，编写方法实现以下任务：

- 创建一个 Song 对象，设置歌曲信息。
- 创建一个 Playlist 对象，向播放列表中添加和删除歌曲。如果一首新歌不在主列表中，将其添加进去。当从主播放列表删除一首歌时，也要从音乐集合的其他播放列表删除歌曲。
- 创建一个 MusicCollection 对象，并对集合添加和删除播放列表。
- 搜索和显示所有歌曲、播放列表或整个音乐集合的信息。

注意

这可能是最具指导性的练习，但不容易，图 15.6 演示了名为 mymusic 的 MusicCollection 示例。它具有 3 个播放列表，其中有一个主播放列表 library 包含 5 首歌，playlist1 有两首歌，playlist2 有一首。这里有一个提示：充分利用 NSMutableArray 类存储的仅是每个新的播放列表中歌曲的引用（而不是复制）（使用 addObject:方法即可）。

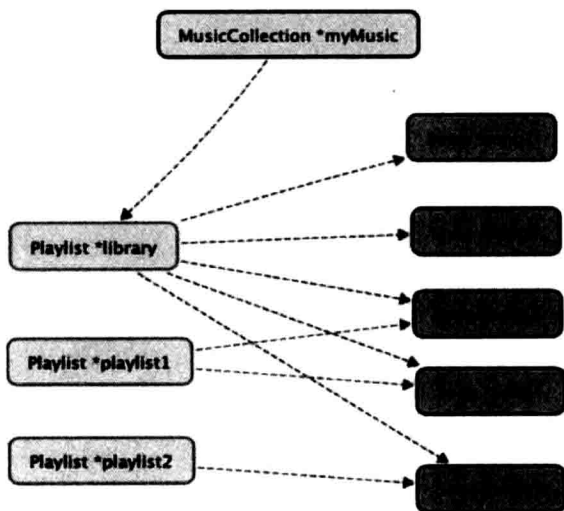


图 15.6 示例中的音乐集合数据结构

10. 编写一个程序，使用 NSNumber 对象的 NSArray（每个 NSNumber 代表一个整数）生成一个频率图表，列出每个整数和它在数组中出现的次数。

11. 当使用方法 `addCard:` 向地址簿添加地址卡片时，谁拥有这些地址卡片？这些卡片上的信息发生改变是否会影响存储在地址簿中的卡片？思考一下，如何实现一个更加安全的 `addCard:` 方法。

16

使用文件

Foundation 框架允许你利用文件系统对文件或目录执行基本操作。这些基本操作是由 **NSFileManager** 类提供的，这个类的方法具有如下功能：

- 创建一个新文件。
- 从现有文件中读取数据。
- 将数据写入文件。
- 重命名文件。
- 删除文件。
- 测试文件是否存在。
- 确定文件的大小和其他属性。
- 复制文件。
- 测试两个文件的内容是否相同。

上面的多数方法也可以对目录进行操作。例如，创建目录、读取目录的内容，或者删除目录。另一个重要特性是链接文件，即同一个文件存在两个不同的名字，有时甚至位于不同的目录中。

使用 **NSFileHandle** 类提供的方法，可以打开文件并对文件执行多次读/写操作。**NSFileHandle** 类的方法可以实现如下功能：

- 打开一个文件，执行读、写或更新（读取和写入）操作。
- 在文件中查找指定位置。
- 从文件中读取特定数目的字节，或将指定数目的字节写入文件中。

NSFileHandle 类提供的方法也可用于各种设备或套接字。然而，本章只涉

及普通文件的处理。

NSURL 类允许在应用中使用 URL 方法。下面通过简单的例子可以了解如何从互联网中读取数据。

NSBundle 类提供了允许在应用中使用包（bundle）的方法，包括搜索包中的特定资源（如 JPEG 图片）。在本章结尾部分会涉及这个类。

16.1 管理文件和目录：NSFileManager

对于 NSFileManager，文件或目录是使用文件的路径名的唯一标识。每个路径名都是一个 NSString 对象，它既可以是相对路径，也可以是完整的路径。相对路径是相对于当前目录的路径名。所以，文件名 copy1.m 表示在当前目录中的文件 copy1.m。斜线字符用于分隔路径中的目录列表。文件名 ch16/copy1.m 也是相对路径，表示存储在目录 ch16 中的文件 copy1.m，而 ch16 也包含在当前目录中。

完整路径也称为绝对路径，以斜线（/）开始。斜线实际上就是一个目录，称为根目录。在笔者的 Mac 上，主目录的完整路径为/users/stevekochan，这个路径名指出了 3 个目录：/（根目录）、Users 和 stevekochan。

特殊的代字符（~）作为用户主目录的缩写。因此，~linda 表示用户 linda 主目录的缩写，这个目录的路径可能是/User/linda。单个代字符表示当前用户的主目录，路径名~/copy1.m 会引用存储在当前用户主目录中的文件 copy1.m。其他特殊的 UNIX 风格的路径名字符，如表示当前目录的“.”和表示父目录的“..”，需要在使用 Foundation 的文件处理方法之前将这些字符从路径名中移除。还可以使用一些路径实用工具，在本章最后部分会有介绍。

在程序中，应该尽量避免使用硬编码的路径名。我们已经提到，可以使用方法和函数来获取当前目录的路径名、用户的主目录及可以用来创建临时文件的目录，我们应尽可能利用这些函数和方法。在本章后面会提到，Foundation 有一个函数，可以用于获取一些特殊的目录，如用户的 Documents 目录。这个函数在 OS X 和 iOS 应用中均适用。

表 16.1 总结了一些基本的 NSFileManager 方法，可以使用这些方法处理文件。在表 16.1 中，path、path1、path2、from 和 to 都是 NSString 对象；attr 是

一个 `NSDictionary` 对象；`err` 是一个指向 `NSError` 对象的指针，能提供更多的错误信息。如果 `err` 指定为 `NULL`，就会采取默认的行为，具有 `BOOL` 返回值的方法，即如果操作成功，就会返回 `YES`；否则就会返回 `NO`。本章中并未使用这个对象。

表 16.1 常见的 `NSFileManager` 文件方法

方 法	描 述
<code>-(NSData *) contentsAtPath: path</code>	从一个文件中读取数据
<code>-(BOOL) createFileAtPath: path contents: (NSData *) data attributes: attr</code>	向一个文件写入数据
<code>-(BOOL) removeItemAtPath: path error: err</code>	删除一个文件
<code>-(BOOL) moveItemAtPath: from toPath: to error: err</code>	重命名或者移动一个文件（ <code>to</code> 不能是已存在的）
<code>-(BOOL) copyItemAtPath: from toPath: to error: err</code>	复制文件（ <code>to</code> 不能是已存在的）
<code>-(BOOL) contentsEqualAtPath: path1 andPath: path2</code>	比较这两个文件的内容
<code>-(BOOL) fileExistsAtPath: path</code>	测试文件是否存在
<code>-(BOOL) isReadableFileAtPath: path</code>	测试文件是否存在，并且是否能执行读操作
<code>-(BOOL) isWritableFileAtPath: path</code>	测试文件是否存在，并且是否能执行写操作
<code>-(NSDictionary *) attributesOfItemAtPath: path error: err</code>	获取文件的属性
<code>-(BOOL) setAttributesOfItemAtPath: attr error: err</code>	更改文件的属性

每个文件方法都是对 `NSFileManager` 对象的调用，而 `NSFileManager` 对象是通过向类发送一条 `defaultManager` 消息创建的，语句如下：

```
NSFileManager *fm;
...
fm = [NSFileManager defaultManager];
```

例如，要从当前目录删除名为 `todolist` 的文件，首先要创建一个 `NSFileManager` 对象（如前面所示），然后调用 `removeItemAtPath:` 方法，代码如下：

```
[fm removeItemAtPath: @"todolist" error: NULL];
```

可以测试返回结果，以确保成功地删除该文件。

```
if ([fm removeItemAtPath: @"todolist" error: NULL] == NO) {
    NSLog(@"Couldn't remove file todolist");
    return 1;
}
```

除了其他事情之外，属性字典还允许你指定要创建的文件的权限，以便获取或者更改现有文件的信息。对于文件的创建，如果将该参数指定为 `nil`，该文

件会被设置为默认权限。`attributesOfItemAtPath:traverseLink:`方法返回一个包含指定文件属性的字典。对于符号链接（symbolic link），`traverseLink:`参数的值为 YES 或 NO。如果该文件是一个符号链接，则指定 YES，并且返回链接到的文件属性。如果指定 NO，则返回链接本身的属性。

对于现有的文件，属性字典包括各种信息，如文件的所有者、文件大小、文件的创建日期，等等。字典的每个属性可以通过键值提取，而所有的键都定义在头文件 `<Foundation/NSFileManager.h>` 中。例如，表示文件大小的键值为 `NSFileSize`。

代码清单 16-1 展示了一些基本的文件操作。这个例子假设当前目录中存在一个名为 `testfile` 的文件，文件的内容如下：

```
This is a test file with some data in it.
Here's another line of data.
And a third.
```

代码清单 16-1

```
// 基本的文件操作
// 假定存在一个名为“testfile”的文件
// 在当前目录

#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) {
    @autoreleasepool {
        NSString      *fName = @"testfile";
        NSFileManager  *fm;
        NSDictionary   *attr;

        // 需要创建文件管理器的实例

        fm = [NSFileManager defaultManager];

        // 首先确认测试文件存在

        if ([fm fileExistsAtPath: fName] == NO) {
            NSLog(@"File doesn't exist!");
            return 1;
        }

        // 创建一个副本

        if ([fm copyItemAtPath: fName toPath: @"newfile" error: NULL] == NO) {
```

```

        NSLog(@"File Copy failed!");
        return 2;
    }

    // 测试两个文件是否一致

    if ([fm contentsEqualAtPath: fName andPath: @"newfile"] == NO) {
        NSLog(@"Files are Not Equal!");
        return 3;
    }

    // 重命名副本

    if ([fm moveItemAtPath: @"newfile" toPath: @"newfile2"
        error: NULL] == NO){
        NSLog(@"File rename Failed");
        return 4;
    }

    // 获取 newfile2 的大小

    if ((attr = [fm attributesOfItemAtPath: @"newfile2" error: NULL])
        == nil) {
        NSLog(@"Couldn't get file attributes!");
        return 5;
    }

    NSLog(@"File size is %llu bytes",
        [[attr objectForKey: NSFileSize] unsignedLongLongValue]);

    // 最后删除原始文件

    if ([fm removeItemAtPath: fName error: NULL] == NO) {
        NSLog(@"file removal failed");
        return 6;
    }

    NSLog(@"All operations were successful");

    // 显示新创建的文件内容

    NSLog(@"%@", [NSString stringWithContentsOfFile:
        @"newfile2" encoding:NSUTF8StringEncoding error:NULL]);
}
return 0;
}

```

代码清单 16-1 输出

```
File size is 84 bytes
All operations were successful!

This is a test file with some data in it.
Here's another line of data.
And a third.
```

这个程序首先测试 `testfile` 文件是否存在。如果存在，则复制 `testfile` 文件，然后比较原文件和复制文件是否相等。经验丰富的 UNIX 用户应该注意到，只为方法 `copyItemAtPath:toPath:error:` 和 `moveItemAtPath:toPath:error:` 指定目标目录，并不能将文件移动或复制到这个目录中，必须明确地指定目标目录中的文件名。

注意

可以使用 Xcode 创建文件 `testfile`，方法是选择菜单 `File→New→New File...`，在出现的左窗格中，突出显示 `Other`，然后选择右侧窗格中的 `Empty File`。输入 `testfile` 作为文件名，在创建时，该文件所在的目录一定要与可执行文件所处的目录相同。如果在定位目录上遇到问题，可以使用本节后面描述的 `currentDirectoryPath:` 方法；或者可以使用这个文件的全路径名，如 `/Users/steve/testfile`（需要替代这里的用户名“steve”）。

`moveItemAtPath:toPath:` 方法可以用来将文件从一个目录移到另一个目录中（也可以用来移动整个目录）。如果两个路径引用同一目录中的文件（如本例所示），其结果仅仅是重新命名这个文件。因此，在代码清单 16-1 中，使用这个方法将文件 `newfile` 重新命名为 `newfile2`。

如表 16.1 所示，在执行复制、重命名或移动操作时，目标文件不能是已存在的；否则，操作将失败。

`newfile2` 的大小是通过使用 `attributesOfItemAtPath:error` 方法确定的。测试并确保返回了一个非 `nil` 目录，然后使用 `NSDictionary` 类中的方法 `objectForKey:`，并用键 `NSFileSize` 从字典中获得文件的大小。最后，显示字典中表示文件大小的整数值。

程序使用 `removeItemAtPath:error:` 方法来删除原始文件 `testfile`。

最后，使用 `NSString` 的 `stringWithContentsOfFile:encoding:error:` 方法将文件

`newfile2` 的内容读入一个字符串对象，然后这个对象作为参数传递给要显示的 `NSLog`。`encoding` 参数指定文件中的字符数据如何表示。可供选择的参数定义在 `NSString.h` 头文件中。使用 `NSUTF8StringEncoding` 可说明文件包含常规的 UTF8 ASCII 字符。

在代码清单 16-1 中，测试每个文件操作以检查它是否成功。如果任何一个操作失败，就会使用 `NSLog` 来记录错误，并且程序将通过返回一个非零的退出状态而退出。根据约定，每个非零值都表示一次程序失败，并且根据错误类型，这个值都是唯一的。如果正在编写命令行工具，这将是一项有用的技术，因为可以由另一个程序来测试返回值，比如，从一个 `shell` 脚本中测试。

16.1.1 使用 NSData 类

使用文件时，需要频繁地将数据读入到一个临时存储区，这个临时存储区通常称为缓冲区。当收集数据，以便随后将这些数据输出到文件中时，通常也使用存储区。`Foundation` 的 `NSData` 类提供了一种简单的方式，它用来设置缓冲区、将文件的内容读入缓冲区，或将缓冲区的内容写到一个文件。有一点不要奇怪，对于 32 位应用程序，`NSData` 缓冲区最多可存储 2GB 的数据。对于 64 位应用程序，最多可存储 8EB（注意是 EB），即 8 亿 GB 的数据。

正如你所期望的，我们既可以定义不可变缓冲区（使用 `NSData` 类），也可以定义可变的缓冲区（使用 `NSMutableData` 类）。在本章和后续几章将介绍这个类的方法。

代码清单 16-2 展示了如何方便地将文件的内容读入到内存缓冲区。

这个程序读取文件 `newfile2` 的内容，并将其写入一个名为 `newfile3` 的新文件中。从某种意义上说，它实现了文件的复制操作，尽管它采取的方式并不像方法 `copyItemAtPath:toPath:error:` 那样直接。

代码清单 16-2

```
// 制作文件的副本
#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSFileManager *fm;
```



```

NSData          *fileData;

fm = [NSFileManager defaultManager];

// 读取文件 newfile2

fileData = [fm contentsAtPath: @"newfile2"];

if (fileData == nil) {
    NSLog(@"File read failed!");
    return 1;
}

// 将数据写入 newfile3

if ([fm createFileAtPath: @"newfile3" contents: fileData
    attributes: nil] == NO) {
    NSLog(@"Couldn't create the copy!");
    return 2;
}

NSLog(@"File copy was successful!");
}
return 0;
}

```

代码清单 16-2 输出

```
File copy was successful!
```

contentsAtPath:方法仅仅接收一个路径名，并将指定文件的内容读入该方法创建的存储区，如果读取成功，这个方法将返回存储区对象作为结果，否则将返回 `nil`（例如，该文件不存在或者你不能读取）。

方法 **createFileAtPath:Contents:attributes:**创建了一个具有特定属性的文件（或者如果 `attributes` 参数提供为 `nil`，则采用默认的属性值）。然后，将指定的 `NSData` 对象内容写入这个文件中。在本例中，这个数据区包含前面读取的文件内容。

16.1.2 使用目录

表 16.2 总结了 `NSFileManager` 提供的用于处理目录的一些方法。其中大多数方法与用于普通文件的方法相同，如表 16.1 所示。

表 16.2 常见的 NSFileManager 目录方法

方 法	描 述
-(NSString *) currentDirectoryPath	获取当前目录
-(BOOL) changeCurrentDirectoryPath: <i>path</i>	更改当前目录
-(BOOL) copyItemAtPath: <i>from</i> toPath: <i>to</i> error: <i>err</i>	复制目录结构 (to 不能是已存在的)
-(BOOL) createDirectoryAtPath: <i>path</i> withIntermediateDirectories: (BOOL) <i>flag</i> attributes: <i>attr</i>	创建一个新目录
-(BOOL) fileExistsAtPath: <i>path</i> isDirectory: (BOOL *) <i>flag</i>	测试文件是不是目录 (flag 中存储结果 YES/NO)
-(NSArray *) contentsOfDirectoryAtPath: <i>path</i> error: <i>err</i>	列出目录内容
-(NSDirectoryEnumerator *) enumeratorAtPath: <i>path</i>	枚举目录的内容
-(BOOL) removeItemAtPath: <i>path</i> error: <i>err</i>	删除空目录
-(BOOL) moveItemAtPath: <i>from</i> toPath: <i>to</i> error: <i>err</i>	重命名或移动一个目录 (to 不能是已存在的)

代码清单 16-3 展示了一些使用目录的基本操作。

代码清单 16-3

// 一些基本的目录操作

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSString      *dirName = @"testdir";
        NSString      *path;
        NSFileManager  *fm;

        // 需要创建文件管理器的实例

        fm = [NSFileManager defaultManager];

        // 获取当前目录

        path = [fm currentDirectoryPath];
        NSLog(@"Current directory path is %@", path);

        // 创建一个新目录

        if ([fm createDirectoryAtPath: dirName withIntermediateDirectories: YES
            attributes: nil error: NULL] == NO) {
            NSLog(@"Couldn't create directory!");
        }
    }
}
```

```

        return 1;
    }

    // 重命名新的目录

    if ([fm moveItemAtPath: dirName toPath: @"newdir" error: NULL] == NO) {
        NSLog(@"Directory rename failed!");
        return 2;
    }

    // 更改目录到新的目录

    if ([fm changeCurrentDirectoryPath: @"newdir"] == NO) {
        NSLog(@"Change directory failed!");
        return 3;
    }

    // 获取并显示当前的工作目录

    path = [fm currentDirectoryPath];
    NSLog(@"Current directory path is %@", path);

    NSLog(@"All operations were successful!");
}
return 0;
}

```

代码清单 16-3 输出

```

Current directory path is /Users/stevekochan/progs/ch16
Current directory path is /Users/stevekochan/progs/ch16/newdir
All operations were successful!

```

代码清单 16-3 很容易理解。出于获得信息的目的, 首先获取当前的目录路径。

注意

使用终端运行程序, 程序的输出显示已经取得了当前目录。如果在 Xcode 中运行这个程序, 取得的当前目录是: /Users/steve_kochan/Library/Developer/Xcode/DerivedData/prog2cnoljvycenooipiddzwoyraybqlza/Build/Products/Debug。实际的输出可能会有所不同。在 iOS 设备上, 程序是运行在沙盒中的, 它严格限定了文件的访问。如果在设备中运行这个程序, 会看到当前目录是/, 这说明应用的根目录是在运行它的沙盒中, 并不是整个 iOS 设备文件目录的根。

接着, 在当前目录中创建一个名为 testdir 的新目录。然后使用

`moveItemAtPath:toPath:error:`方法将新目录 `testdir` 重命名为 `newdir`。记住，这个方法还可以用来将整个目录结构（这意味着包括目录的内容）从文件系统的—个位置移动到另一个位置。

重命名新目录之后，程序使用 `changeCurrentDirectoryPath:`方法将这个新目录设置为当前目录。然后，显示当前目录路径，以验证修改是否成功。

16.1.3 枚举目录中的内容

有时需要获得目录的内容列表。使用 `enumeratorAtPath:`方法或者 `contentsOfDirectoryAtPath:error:`方法都可以完成枚举过程。如果使用第一种方法，一次可以枚举指定目录中的每个文件，默认情况下，如果其中一个文件为目录，那么也会递归枚举它的内容。在这个过程中，通过向枚举对象发送一条 `skipDescendants` 消息，可以动态地阻止递归过程，从而不再枚举目录中的内容。

使用 `contentsOfDirectoryAtPath:error:`方法可以枚举指定目录的内容，并在一个数组中返回文件列表。如果这个目录中的任何文件本身是一个目录，这个方法并不递归枚举它的内容。

代码清单 16-4 演示了如何在程序中使用这两个方法。

代码清单 16-4

// 枚举目录中的内容

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSString      *path;
        NSFileManager  *fm;
        NSDirectoryEnumerator *dirEnum;
        NSArray        *dirArray;

        // 需要创建文件管理器的实例

        fm = [NSFileManager defaultManager];

        // 获取当前工作目录的路径

        path = [fm currentDirectoryPath];
```

```

// 枚举目录

dirEnum = [fm enumeratorAtPath: path];

NSLog(@"Contents of %@", path);

while ((path = [dirEnum nextObject]) != nil)
    NSLog(@"%@", path);

// 另一种枚举目录的方法
dirArray = [fm contentsOfDirectoryAtPath:
    [fm currentDirectoryPath] error: NULL];
NSLog(@"Contents using contentsOfDirectoryAtPath:error:");

for ( path in dirArray )
    NSLog(@"%@", path);
}
return 0;
}

```

代码清单 16-4 输出

```

Contents of /Users/stevekochan/mysrc/ch16:
a.out
dir1.m
dir2.m
file1.m
newdir
newdir/file1.m
newdir/output
path1.m
testfile

Contents using contentsOfDirectoryAtPath:error:
a.out
dir1.m
dir2.m
file1.m
newdir
path1.m
testfile

```

注意

在你的系统中，实际输出会与这里显示的不同。

让我们仔细看看以下代码：

```
dirEnum = [fm enumeratorAtPath: path];

NSLog(@"Contents of %", path);
while ((path = [dirEnum nextObject]) != nil)
    NSLog(@"%@", path)
```

你可以通过向文件管理器对象（此处是 `fm`）发送 `enumeratorAtPath:` 消息来开始目录的枚举过程。`enumeratorAtPath:` 方法返回了一个 `NSDirectoryEnumerator` 对象，这个对象存储在 `dirEnum` 中。现在，每次向该对象发送 `nextObject` 消息时，都会返回所枚举的目录中下一个文件的路径。没有其他文件可供枚举过程使用时，会返回 `nil`。

从代码清单 16-4 的输出中，可以看到这两种枚举技术的不同之处。`enumeratorAtPath:` 方法列出了 `newdir` 目录中的内容，而方法 `contentsOfDirectoryAtPath:error:` 没有。如果 `newdir` 包含子目录，那么方法 `enumeratorAtPath:` 也会枚举其中的内容。

前面提到过，在执行代码清单 16-4 的 `while` 循环过程中，通过对代码做如下更改，可以阻止任何子目录中的枚举。

```
while ((path = [dirEnum nextObject]) != nil) {
    NSLog(@"%@", path);

    [fm fileExistsAtPath: path isDirectory: &flag];

    if (flag == YES)
        [dirEnum skipDescendents];
}
```

这里，`flag` 是一个 `BOOL` 变量。如果指定的路径是目录，则 `fileExistsAtPath:` 在 `flag` 中存储 `YES`，否则存储 `NO`。

另外提醒一下，无须像在这个程序中那样进行快速枚举，使用以下 `NSLog` 调用可显示整个 `dirArray` 的内容：

```
NSLog(@"%@", dirArray);
```

16.2 使用路径：NSPathUtilities.h

`NSPathUtilities.h` 包含了 `NSString` 的函数和分类扩展，它允许你操作路径名。应该尽可能地使用这些函数，以便使程序更独立于文件系统结构及特定文

件和目录的位置。

代码清单 16-5 展示了如何使用 `NSPathUtilities.h` 提供的几种函数和方法。

代码清单 16-5

//一些基本路径的操作

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSString      *fName = @"path.m";
        NSFileManager  *fm;
        NSString      *path, *tempdir, *extension, *homedir, *fullpath;

        NSArray        *components;

        fm = [NSFileManager defaultManager];

        //获取临时的工作目录

        tempdir = NSTemporaryDirectory ();

        NSLog (@@"Temporary Directory is %@", tempdir);

        //从当前目录中提取基本目录

        path = [fm currentDirectoryPath];
        NSLog (@@"Base dir is %@", [path lastPathComponent]);

        //创建文件 fName 在当前目录中的完整路径

        fullpath = [path stringByAppendingPathComponent: fName];
        NSLog (@@"fullpath to %@ is %@", fName, fullpath);

        //获取文件扩展名

        extension = [fullpath pathExtension];
        NSLog (@@"extension for %@ is %@", fullpath, extension);

        //获取用户的主目录

        homedir = NSHomeDirectory ();
        NSLog (@@"Your home directory is %@", homedir);

        //拆分路径为各组成部分
```

```

    components = [homedir pathComponents];

    for ( path in components )
        NSLog ("%@", path);
    }
    return 0;
}

```

代码清单 16-5 输出

```

Temporary Directory is /var/folders/HT/HTyGLvSNHTuNb6NrMuo7QE+++TI/-Tmp-/
Base dir is examples
fullpath to path.m is /Users/stevekochan/progs/examples/path.m
extension for /Users/stevekochan/progs/examples/path.m is m
Your home directory is /Users/stevekochan
/
Users
stevekochan

```

注意

在你的系统中，实际输出会与这里显示的不同。

函数 `NSTemporaryDirectory` 返回系统中可以用来创建临时文件的目录路径名。如果在这个目录中创建临时文件，一定要在完成之后将它们删除。另外，还要确保文件名是唯一的，特别是在应用程序的多个实例同时运行时（参见本章的练习 5）更应如此。如果多个用户登录到系统，并且运行同一个应用程序，这种情况就很容易发生。如果你创建的临时文件并未移除，系统将会在某个时候为你移除。然而，最好不要依靠系统来做这件事情，而是自己移除临时文件。

方法 `lastPathComponent` 用来从路径中提取最后一个文件名。当你有一个绝对路径名，并且只想从中获取基本的文件名时，这个函数很有用。

`stringByAppendingPathComponent:` 方法用于将文件名附加到路径的末尾。如果指定为接收者的路径名不是以斜线结束，那么该方法将在路径名中插入一条斜线，将路径名和附加的文件名分开。结合使用 `currentDirectoryPath` 方法和 `stringByAppendingPathComponent:` 方法，可以在当前目录中创建文件的完整路径名。这种技术展示在代码清单 16-5 中。

`PathExtension` 方法给出了指定路径名的文件扩展名。在这个例子中，文件

path.m 的扩展名为 m，该方法返回这个扩展名。如果所给的文件没有扩展名，那么方法仅仅返回一个空字符串。

NSHomeDirectory 函数返回当前用户的主目录。使用 NSHomeDirectoryForUser 函数，同时提供用户名作为函数的参数，可以获得任何特定用户的主目录。

PathComponents 方法返回一个数组，这个数组包含指定路径的每个组成部分。代码清单 16-5 按顺序显示了返回数组的每一元素，并且在单独的输出行上显示每个路径的组成部分。

16.2.1 常用的路径处理方法

表 16.3 总结了许多常用的使用路径方法。其中，components 是一个 NSArray 对象，它包含路径中每一部分的字符串对象，Path 是一个字符串对象，它指定了文件的路径；ext 是路径扩展名的字符串对象（如，@"mp4"）。

表 16.3 常用的路径工具方法

方 法	描 述
+(NSString *) pathWithComponents: <i>components</i>	根据 <i>components</i> 中的元素构造有效路径
-(NSArray *) pathComponents	析构路径，获得组成此路径的各个部分
-(NSString *) lastPathComponent	提取路径的最后一个组成部分
-(NSString *) pathExtension	从路径的最后一个组成部分中提取其扩展名
-(NSString *) stringByAppendingPathComponent: <i>path</i>	将 <i>path</i> 添加到现有路径的末尾
-(NSString *) stringByAppendingPathExtension: <i>ext</i>	将指定的扩展名添加到路径的最后一个组成部分
-(NSString *) stringByDeletingLastPathComponent	删除路径的最后一个组成部分
-(NSString *) stringByDeletingPathExtension	从文件的最后一部分删除扩展名
-(NSString *) stringByExpandingTildeInPath	将路径中的代字符扩展成用户主目录(~)或指定用户的主目录(~-user)
-(NSString *) stringByResolvingSymlinksInPath	尝试解析路径中的符号链接
-(NSString *) stringByStandardizingPath	通过尝试解析~、..(父目录符号)、.(当前目录符号)和符号链接来标准化路径

表 16.4 展示了一些函数，它可用于获取用户、用户的主目录和存储临时文件的目录信息。

表 16.4 常用的路径工具函数

函 数	描 述
NSString *NSUserName (void)	返回当前用户的登录名
NSString *NSFullUserName (void)	返回当前用户的完整用户名
NSString *NSHomeDirectory (void)	返回当前用户主目录的路径
NSString *NSHomeDirectoryForUser (NSString *user)	返回用户 user 的主目录
NSString *NSTemporaryDirectory (void)	返回可用于创建临时文件的路径目录

你可能还想查看 Foundation 函数 `NSSearchPathForDirectoriesInDomains`，它可用于查找系统的特殊目录，如 `Application` 和 `Documents` 目录。例如，定义一个方法 `saveFilePath`，它返回文件 `saveFile` 在 `Documents` 目录中的路径。这个方法可以用于应用里保存一些数据到一个文件中。

```
-(NSString *) saveFilePath
{
    NSArray *dirList = NSSearchPathForDirectoriesInDomains
        (NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *docDir = dirList[0];

    return [docDir stringByAppendingPathComponent:@"saveFile"];
}
```

注意

为了保存数据直到下一次运行程序仍能够使用，可以使用 `Documents` 目录。每个 `iOS` 应用都有自己的 `Documents` 目录供数据写入。函数 `NSSearchPathForDirectoriesInDomains` 的第一个参数是指定需要查找目录的路径。应用中的 `Caches` 目录也可以用来存储一些数据。对于 `iOS 5` 来说，Apple 鼓励开发者存储持久化数据到云端，表 16.5 列出了 `iOS` 中常用的目录。

表 16.5 常用的 iOS 目录

目 录	用 途
Documents (NSDocumentDirectory)	用于写入应用相关数据文件的目录。在 <code>iOS</code> 中写入这里的文件能够与 <code>iTunes</code> 共享并访问，存储在这里的文件会自动备份到 <code>iCloud</code>
Library/Caches (NSCachesDirectory)	用于写入应用支持文件的目录，保存应用程序再次启动需要的信息。 <code>iTunes</code> 不会对这个目录的内容进行备份
tmp (use NSTemporaryDirectory())	这个目录用于存放临时文件，在程序终止时需要移除这些文件。当应用程序不再需要这些临时文件时，应该将其从这个目录中删除
Library/Preferences	这个目录包含应用程序的偏好设置文件。使用 <code>NSUserDefaults</code> 类进行偏好设置文件的创建、读取和修改

函数的第二个参数可以是多个值，用于指定需要列出的目录，如用户的（正如例子）、系统的或者所有目录。最后一个参数用于指定是否展开路径中的~字符。

`NSSearchPathForDirectoriesInDomains` 返回一组路径的数组。如果仅是查找用户的目录，这个数组只包含一个元素，如果第二个参数指定多个值，该数组会包含多个元素。

注意

当为 iOS 编写程序时，`NSSearchPathForDirectoriesInDomains` 函数的第二个参数应是 `NSUserDomainMask`，并希望得到一个包含单个路径的数组作为返回。

16.2.2 复制文件和使用 `NSProcessInfo` 类

代码清单 16-6 说明了如何使用命令行工具来实现简单的文件复制操作。这个命令的用法如下：

```
copy from-file to-file
```

与 `NSFileManager` 的 `copyPath:toPath:handler:` 方法不同，命令行工具允许 `to-file` 是目录名。在这个例子中，文件以名称 `from-file` 被复制到 `to-file` 目录中。与 `copyPath:toPath:handler:` 方法不同的还有，如果 `to-file` 目录已存在，允许覆盖其内容。这更像标准 UNIX 的复制命令 `cp`。

通过在 `main` 函数中使用 `argv` 和 `argc` 参数，可以从命令行中获得文件名。这两个参数分别包括命令行中键入的参数个数（包括命令名），以及指向 C 风格的字符串数组的指针。

并非必须处理 C 字符串（使用 `argv` 参数时，必须处理 C 字符串），而是利用 Foundation 中的类 `NSProcessInfo`。`NSProcessInfo` 类中包含一些方法，它们允许你设置或检索正在运行的应用程序（即进程）的各种类型的信息。表 16.6 总结了这些方法。

表 16.6 NSProcessInfo 类方法

方 法	描 述
+(NSProcessInfo *) processInfo	返回当前进程的信息
-(NSArray *) arguments	以 NSString 对象数组的形式返回当前进程的参数
-(NSDictionary *) environment	返回变量/值对词典, 以描述当前的环境变量 (比如 PATH 和 HOME) 及其值
-(int) processIdentifier	返回进程标识符, 它是操作系统赋予进程的唯一数字, 用于识别每个正在运行的进程
-(NSString *) processName	返回当前正在执行的进程名称
-(NSString *) globallyUniqueString	每次调用这个方法时, 都返回不同的单值字符串。可以用这个字符串生成单值临时文件名 (参见练习 5)
-(NSString *) hostname	返回主机系统的名称 (在笔者的 OS X 系统中, 返回的是 Steve-Kochans-Computer.local)
-(NSInteger) operatingSystem	返回表示操作系统的数字 (在笔者的 Mac 上, 返回的值是 5)
-(NSString *) operatingSystemName	返回操作系统的名称 (在笔者的 Mac 机器上, 返回常量 NSMACHOperatingSystem, 其中可能的返回值定义存于 NSProcessInfo.h 中)
-(NSString *) operatingSystemVersionString	返回操作系统的当前版本 (在笔者的 Mac OS X 系统中, 返回 Version 10.6.7 (Build 10J869))
-(void) setProcessName: (NSString *) name	将当前进程名称设置为 name。应该谨慎地使用这个方法, 因为关于进程名称存在一些假设 (比如用户默认的设置)

代码清单 16-6

// 实现基本的复制工具

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSFileManager *fm;
        NSString      *source, *dest;
        BOOL          isDir;
        NSProcessInfo *proc = [NSProcessInfo processInfo];
        NSArray        *args = [proc arguments];

        fm = [NSFileManager defaultManager];

        // 检查命令行中的两个参数
```

```

if ([args count] != 3) {
    NSLog(@"Usage: %@ src dest", [proc processName]);
    return 1;
}

source = args[1];
dest = args[2];

//确定能够读取源文件

if ([fm isReadableFileAtPath: source] == NO) {
    NSLog(@"Can't read %@", source);
    return 2;
}

//目标文件是否是一个目录
//若是，添加源到目标的结尾

[fm fileExistsAtPath: dest isDirectory: &isDir];

if (isDir == YES)
    dest = [dest stringByAppendingPathComponent:
            [source lastPathComponent]];

//若目标文件已存在，则删除文件

[fm removeItemAtPath: dest error: NULL];

//好了，执行复制

if ([fm copyItemPath: source toPath: dest error: NULL] == NO) {
    NSLog(@"Copy failed!");
    return 3;
}

NSLog(@"Copy of %@ to %@ succeeded!", source, dest);
}
return 0;
}

```

代码清单 16-6 输出

```

$ ls -l          see what files we have
total 96
-rwxr-xr-x 1 stevekoc staff 19956 Jul 24 14:33 copy
-rw-r--r-- 1 stevekoc staff 1484  Jul 24 14:32 copy.m

```

```

-rw-r--r--1 stevekoc staff 1403 Jul 24 13:00 file1.m
drwxr-xr-x 2 stevekoc staff 68 Jul 24 14:40 newdir
-rw-r--r--1 stevekoc staff 1567 Jul 24 14:12 path1.m
-rw-r--r--1 stevekoc staff 84 Jul 24 13:22 testfile
$ copy      try with no args
Usage: copy src dest
$ copy foo copy2
Can't read foo
$ copy copy.m backup.m
Copy of copy.m to backup.m succeeded!
$ diff copy.m backup.m  compare the files
$ copy copy.m newdir    try copy into directory
Copy of copy.m to newdir/copy.m succeeded!
$ ls -l newdir
total 8
-rw-r--r- 1 stevekoc staff 1484 Jul 24 14:44 copy.m
$

```

注意

这里显示的输出会与你系统中运行程序的当前路径有所不同。

NSProcessInfo 类中的 arguments 方法返回一个字符串对象数组。数组的第一个元素是进程名称，其余的元素是在命令行中输入的参数。

首先检查，以确保在命令行中输入这两个参数。这是通过测试数组 args 的大小实现的，这个数组是从方法 arguments 返回的。如果测试成功，那么程序将从数组 args 中提取源文件名和目标文件名，并将它们的值分别赋给 source 和 dest。

然后程序测试源文件是否能够读取。如果不能，则生成一条出错消息，并退出程序。语句

```
[fm fileExistsAtPath: dest isDirectory: &isDir];
```

检查 dest 指定的文件是否是目录。前面讲过，将答案（YES 或 NO）存储到变量 isDir 中。

如果 dest 是目录，则要将源文件名的最后一部分加到 dest 目录名的末尾，使用路径工具方法 stringByAppendingPathComponent:来实现这个功能。因此，如果 Source 的值是 chl6/copy1.m，dest 的值是/users/stevekochan/progs，并且后者是一个目录，那么 dest 的值将更改为/Users/stevekochan/progs/copy1.m。

方法 copyPath:ToPath:handler:不允许重写文件。因此，要避免错误，程序

尝试用方法 `removeFileAtPath:handler:` 来删除目标文件。没有必要担心这个方法能否成功，因为如果目标文件不存在，它将失败。

到达程序末尾时，可以假设程序的所有部分都运行良好，并为此生成一条消息。

16.3 基本的文件操作：NSFileHandle

利用 `NSFileHandle` 类提供的方法，允许更有效地处理文件。在本章的开始，我们列举了利用这些方法可以完成的一些操作。

一般而言，我们处理文件时都要经历以下步骤：

(1) 打开文件，并获取一个 `NSFileHandle` 对象，以便在后面的 I/O 操作中引用该文件。

(2) 对打开的文件执行 I/O 操作。

(3) 关闭文件。

表 16.7 总结了一些常用的 `NSFileHandle` 方法。在这个表中，`fh` 是一个 `NSFileHandle` 对象，`data` 是一个 `NSData` 对象，`path` 是一个 `NSString` 对象，`offset` 是一个 `unsigned long long`。

表 16.7 常用的 `NSFileHandle` 方法

方 法	描 述
<code>+(id) fileHandleForReadingAtPath: path</code>	打开一个文件准备读取
<code>+(id) fileHandleForWritingAtPath: path</code>	打开一个文件准备写入
<code>+(id) fileHandleForUpdatingAtPath: path</code>	打开一个文件准备更新（读取和写入）
<code>-(NSData *) availableData</code>	从设备或者通道返回可用的数据
<code>-(NSData *) readDataToEndOfFile</code>	读取其余的数据直到文件的末尾（最多 <code>UINT_MAX</code> 字节）
<code>-(NSData *) readDataOfLength: (NSUInteger) bytes</code>	从文件中读取指定字节的内容
<code>-(void) writeData: data</code>	将 <code>data</code> 写入文件
<code>-(unsigned long long) offsetInFile</code>	获取当前文件的偏移量
<code>-(void) seekToFileOffset: offset</code>	设置当前文件的偏移量
<code>-(unsigned long long) seekToEndOfFile</code>	将当前文件的偏移量定位到文件的尾
<code>-(void) truncateFileAtOffset: offset</code>	将文件的长度设置为 <code>offset</code> 字节（如果需要，可以填充内容）
<code>-(void) closeFile</code>	关闭文件

表 16.7 中并未列出获取 `NSFileHandle` 以用于标准输入、标准输出、标准错误和空设备的方法。它们的格式为 `fileHandleWithDevice`，其中 `Device` 可以是 `standardInput`、`standardOutput`、`StandardError` 或 `NullDevice`。

这里也没有列出用于后台（即异步）读取和写入的方法。

应该注意到 `FileHandle` 类并没有提供创建文件的功能。前面描述过，必须使用 `FileManager` 方法来创建文件。因此，方法 `fileHandleForWritingAtPath:` 和 `fileHandleForUpdatingAtPath:` 都假定文件已存在，否则返回 `nil`。对于这两个方法，文件的偏移量都设为文件的开始，所以都是在文件的开始位置开始写入（或更新模式的读取）。另外，如果在 UNIX 系统下编程，应该注意，打开文件进行写入并不会截断文件，需要自己完成截断。

代码清单 16-7 打开本章开始创建的原始 `testfile` 文件，读取它的内容，并将其复制到名为 `testout` 的文件中。

代码清单 16-7

```
// 文件的一些基本操作
// 假设存在一个“testfile”文件
// 在当前工作目录

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSFileHandle *inFile, *outFile;
        NSData *buffer;

        // 打开文件 testfile 并读取

        inFile = [NSFileHandle fileHandleForReadingAtPath: @"testfile"];

        if (inFile == nil) {
            NSLog(@"Open of testfile for reading failed");
            return 1;
        }

        // 如果需要，首先创建输出文件

        [[NSFileManager defaultManager] createFileAtPath: @"testout"
            contents: nil attributes: nil];
```



```

//打开 outfile 文件进行写入

outFile = [NSFileHandle fileHandleForWritingAtPath: @"testout"];

if (outFile == nil) {
    NSLog(@"Open of testout for writing failed");
    return 2;
}

//因为它可能包含数据，截断输出文件

[outFile truncateFileAtOffset: 0];

//从 inFile 中读取数据，将它写到 outFile

buffer = [inFile readDataToEndOfFile];

[outFile writeData: buffer];

//关闭这两个文件

[inFile closeFile];
[outFile closeFile];

//验证文件的内容

NSLog(@"%@", [NSString stringWithContentsOfFile: @"testout" encoding:
    NSUTF8StringEncoding error: NULL]);
}
return 0;
}

```

代码清单 16-7 输出

```

This is a test file with some data in it.
Here's another line of data.
And a third.

```

方法 `readDataToEndOfFile`: 每次从文件中读取最多 `UINT_MAX` 字节的数据，这个量定义在头文件 `<limits.h>` 中，并且在许多系统中值等于 `FFFFFFFF16`。这个值对于你编写的任何应用程序而言，已经足够大了。还可以中断这项操作，以执行少量读取和写入。利用方法 `readDataOfLength`: 甚至可以设置循环，一次在文件之间传输一缓冲区的字节。例如，缓冲区的大小可能是 8192B（即 8KB），也可能是 131072B（即 128KB）。经常使用的是 2 的乘方，这是因为底层的操

作系统通常以块为单位执行 I/O 操作，而块的大小一般为 2 的乘方字节。可能要在系统上试用不同的值，以查看哪个值最适合。

如果读取方法到达文件的末尾，并且没有读到任何数据，那么这个方法将返回一个空的 `NSData` 对象（也就是说，缓冲区中没有字节）。可以对这个缓冲区应用 `length` 方法，并测试长度是否等于零，以查看该文件中是否还剩有数据可以读取。

如果打开一个需要更新的文件，则文件的偏移量要被设为文件的开始。通过在文件中定位（`seeking`），可以更改偏移量，然后执行该文件的读写操作。因此，要定位到文件（文件的句柄为 `databaseHandle`）的第 10 字节，可以编写如下消息表达式：

```
[databaseHandle seekToFileOffset: 10];
```

通过获得当前文件的偏移量，然后加上或者减去这个值，就得到相应文件的位置。因此，要跳过文件中当前位置之后的 128 字节，编写如下代码：

```
[databaseHandle seekToFileOffset:
    [databaseHandle offsetInFile] + 128];
```

要在文件中向回移动 5 个整数所占的字节数，编写如下代码：

```
[databaseHandle seekToFileOffset:
    [databaseHandle offsetInFile] - 5 * sizeof (int)];
```

代码清单 16-8 将一个文件的内容附加到另一个文件中。通过打开另一个用于写入的文件，然后定位到该文件的结尾，最后将第一个文件中的内容写入第二个文件中来实现。

代码清单 16-8

```
//追加文件“fileA”到“fileB”的末尾

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSFileHandle      *inFile, *outFile;
        NSData            *buffer;

        //打开文件 fileA 进行读取
```

```

inFile = [NSFileHandle fileHandleForReadingAtPath: @"fileA"];

if (inFile == nil) {
    NSLog(@"Open of fileA for reading failed");
    return 1;
}

//打开文件 fileB 进行更新

outFile = [NSFileHandle fileHandleForWritingAtPath: @"fileB"];

if (outFile == nil) {
    NSLog(@"Open of fileB for writing failed");
    return 2;
}

//在 outFile 的末尾进行查找

[outFile seekToEndOfFile];

//从 inFile 中读取数据，将它写到 outFile

buffer = [inFile readDataToEndOfFile];
[outFile writeData: buffer];

//关闭这两个文件

[inFile closeFile];
[outFile closeFile];

//验证它的内容

NSLog(@"%@", [NSString stringWithContentsOfFile: @"fileB"
    encoding::NSUTF8StringEncoding error: NULL]);
}
return 0;
}

```

代码清单 16-8 输出

```

Contents of fileB
This is line 1 in the second file.
This is line 2 in the second file.
This is line 1 in the first file.
This is line 2 in the first file.

```

从输出可知，第一个文件的内容成功地附加到第二个文件的末尾。顺便说

明一下，在搜索操作执行完毕后，`seekToEndOfFile` 返回当前文件的偏移量。选择忽略这个值，如果需要，可以使用这个信息来获得程序中文件的大小。

16.4 NSURL 类

NSURL 类提供在应用中使用 URL 地址的相关方法。例如，给出映射到互联网上一个文件路径的 HTTP 地址，调用一些方法就很容易读取这些文件的内容。在 Foundation 中还能找到许多方法是以 NSURL 对象作为参数的。NSURL 对象并不是一个字符串（比如 `@“http://www.apple.com”`），但是使用 `URLWithString:` 方法可以由一个字符串对象创建出 NSURL 对象。

代码清单 16-9 描述了如何通过程序从一个网站读取 HTML 内容。

代码清单 16-9

//读取一个 URL 文件的内容

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

int main (int argc, const char * argv[]) {
    @autoreleasepool {
        NSURL *myURL = [NSURL URLWithString: @"http://classroomM.com"];

        NSString *myHomePage = [NSString stringWithContentsOfURL: myURL
                                encoding: NSASCIIStringEncoding error: NULL];

        NSLog(@"%@", myHomePage);
    }
    return 0;
}
```

代码清单 16-9 部分输出

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<meta http-equiv="Content-Style-Type" content="text/css" />
<meta name="google-site-verification" content="J75blyb6mDQItzHDxWDphlBNc8rVuc0OzLj8gzlj9y8" />
<title>iPhone Online Course and iPhone Programming Training - Home</title>
...
```

可以看出，从互联网中获取数据还是比较容易的。如果词典作为属性列表存储在一个网址上，使用 `dictionaryWithContentsOfURL:` 方法同样能够读取。或者数组存储在属性列表中，可以使用 `arrayWithContentsOfURL:` 方法。如果希望从一个网站中读取任何类型的数据，可以使用 `dataWithContentsOfURL:` 方法。

16.5 NSBundle 类

当创建一个应用时，系统存储了应用相关联的所有数据（其中包括图片、本地化字符串、图标等），将这些内容放入一个称为应用包（application bundle）的包中。为了访问应用中的这些资源，需要熟悉 `NSBundle` 类。

在应用中添加一个资源（如图片或者文本文件）是很方便的：仅需将文件拖到 Xcode 的左边窗格中。当出现对话框时，通常选择复制资源文件到你的项目目录中，这样你的项目都是自包含的。

下面的语句能够返回存储在应用包中 `instructions.txt` 文件的路径：

```
NSString *txtFilePath = [[NSBundle mainBundle]
    pathForResource:@"instructions" ofType:@"txt"];
```

`mainBundle` 方法给出了应用包所在的目录。这个方法在 OS X 和 iOS 应用中都适用。`pathForResource:ofType:` 方法会列出该目录指定的文件并返回文件的路径，随后就可以读取文件的内容到程序中了：

```
NSString *instructions = [NSString stringWithContentsOfFile: txtFilePath
    encoding::NSUTF8StringEncoding error: NULL];
```

如果希望列出在应用包的图片目录中所有以 `jpg` 为文件后缀名的 JPEG 图片，需要使用到 `pathsForResourceOfType:inDirectory:` 方法：

```
NSArray *birds = [[NSBundle mainBundle] pathsForResourceOfType:@"jpg"
    inDirectory:@"birdImages"];
```

这个方法会返回路径名的数组。如果 JPEG 并未存储在应用的子目录中，可以指定 `@""` 作为 `inDirectory:` 参数值。

`NSBundle` 类还有其他一些方法，通过阅读文档可以获得更多的内容。

16.6 练习

1. 修改代码清单 16-6 中编写的复制程序，以便它像标准的 UNIX 命令 `cp` 一样，可以接收多个要复制到该目录的源文件。如下命令

```
$ copy copy1.m file1.m file2.m progs
```

应该将 3 个文件 `copy1.m`、`file1.m` 和 `file2.m` 复制到目录 `progs` 中。如果指定了多个源文件，那么最后一个参数实际上是现有的目录。

2. 编写一个名为 `myfind` 的命令行工具，它带有两个参数。第一个是开始搜索的初始目录，第二个参数是需要定位的文件名。命令行

```
$ myfind /Users proposal.doc
/Users/stevekochan/MyDocuments/proposals/proposal.doc
$
```

首先搜索 `/users` 的文件系统以查找文件 `proposal.doc`。如果找到该文件，则输出该文件的完整路径名；如果没有找到，则输出一条合适的消息。

3. 编写自己的标准 UNIX 工具：`basename` 和 `dirname`。
4. 使用 `NSProcessInfo` 编写一个程序，用于显示每个取值函数方法所返回的所有信息。
5. 给定本章中介绍过的 `NSPathUtilities.h` 函数 `NSTemporaryDirectory` 和 `NSProcessInfo` 方法 `globallyUniqueString`，将名为 `TempFiles` 的分类添加到 `NSString` 中，并定义一个名为 `temporaryFileName` 的方法，每次调用这个方法都返回单值的文件名。
6. 修改代码清单 16-7，以便该文件一次读取和写入 `kBufSize` 字节，其中 `kBufSize` 定义在程序的开始部分。一定要对大型文件测试这个程序（也就是大于 `kBufSize` 字节的文件）。
7. 打开一个文件，一次从中读取 128B，并将其写入终端。利用 `NSFileHandle` 的方法 `fileHandleWithStandardOutput` 来获得终端输出的句柄。

8. 将词典（dictionary）作为属性列表（property list）存储在 URL 中：
<http://bit.ly/aycNwd>。编写程序读取并显示词典中的内容。词典中包含什么样的数据？

内存管理和自动引用计数

记得曾使用 `NSMutableArray` 类创建一个数组，并为其添加对象和移除对象。程序启动时，可能需要将一个文件的内容读入数组中。假定创建的新数组为 `myData`，读取属性列表文件中的内容用于初始化数组（在第 19 章“归档”中有详细的描述）。这需要使用 `NSArray` 的 `arrayWithContentsOfFile:` 方法：

```
NSSArray *myData = [NSArray arrayWithContentsOfFile: @"database1"];
```

该方法会读取文件并进行解析，将结果存储在一个新创建的数组中，然后返回指向这个数组的引用，并将引用存储在变量 `myData` 中。

当处理完 `myData` 中的数据后，希望再从其他文件读取数据并做相似的处理，使用以下语句：

```
myData = [NSArray arrayWithContentsOfFile: @"database2"];
```

这时变量 `myData` 已经改变，引用到了另一个数组，其内容来自第二个文件。那么第一个数组如何处理？已经不再引用这个数组（当覆盖 `myData` 后，数组不再被引用）。第一个数组的元素会怎么样？假如你需要重复从成百上千份文件中读取数据，这些数组对象和它们的元素不再会被引用，并且应用也不再需要这些数据，如何处理这种情况？这些对象可能始终留在内存的某处，即使你不再使用它们。当然，没有经过“清理”过程，内存会继续填充这些未引用的对象，直到应用没有足够的内存去做更多的事情时，应用就有崩溃的可能。

本章会涉及内存管理的一些观点，尽管这是比较高层面的内容。内存管理关心的是清理（回收）不用的内存，以便内存能够再次利用。如果一个对象不再使用，就需要释放对象占用的内存。这听起来比较简单。然而，事实上并不

是那么简单。必须弄清楚这一简单的事实。你、计算机或你们一起，必须能够以某些方式确定一个对象不再需要使用了，并且它占用的内存能够被回收。

为了达到这个目的，人们已经开发出多种内存管理策略。其中有两种自动化的方法，即计算机自动追踪对象，以及在需要的时候自动释放它们的内存。第三种方法是一种混合方法，系统会为你做一些工作，但同时也需要程序员在对象不再使用时通知系统。

在 Xcode 4.2 发布之前，内存管理是比较令人恐惧的主题，程序员需要花很大力气去理解并规划这个部分。需要细致处理引用计数，使用保留（retain）、释放（release）和自动释放（autorelease），才能让开发的应用能够明智地处理内存使用，并且在大多数时候不会崩溃。但结果常常导致试图去引用的对象已经在无意间被销毁了。

随着 Xcode 4.2 版本的发布，有一个新特性是自动引用计数（Automatic Reference Counting, ARC），程序员不再需要思考内存管理的问题。这里有几个说明的例子（阅读 Apple 文档可以获得更详细的内容）。ARC 对于 iOS 开发者来说确实是不错的东西。在本章之前还未提及内存管理的内容就足以说明这一点。在本书的前一个版本中，笔者费了很大心血解释这些内存管理的内容，以确定读者能够充分理解并能够明智地应用内存管理技术。

在本章中，你会看到一个不同的内存管理策略的概述，Cocoa 和 iOS 开发者都可以用它。我们也会简要描述手工内存管理是如何工作的。这些内容是为了帮助你支持遗留的代码，或者出于某些原因，你不想使用自动内存管理特性。

尽管你不需要担心对象的生命周期（比如使用完对象，需要释放这些对象的内存），但是根据应用的类型，明智地决定使用内存仍然很重要。例如，你编写一个交互画板程序，在程序的运行期间创建了许多对象，你需要关心程序能否消耗更多的内存资源。正是这样，你需要灵活地管理这些内存资源，并保证没有创建不必要的对象。

提供给 Objective-C 程序员的基本内存管理模型有以下 3 种：

- 自动垃圾收集。
- 手工引用计数和自动释放池。
- 自动引用计数（ARC）。

17.1 自动垃圾收集

在 Objective-C 2.0 中，有一种称为垃圾收集的内存管理形式。通过垃圾收集，系统就能够自动检测对象是否拥有其他的对象，当程序执行需要空间的时候，不再被引用的对象会被自动释放（垃圾回收）。

iOS 运行环境并不支持垃圾收集，在这个平台开发程序时没有这方面的选项。在 OS X 10.8 (Mountain Lion) 中垃圾收集已经不推荐使用（未来不会继续支持这项特性）。现在 Apple 不建议使用垃圾收集，建议使用 ARC 进行替代。

17.2 手工管理内存计数

在创建应用时，如果你不打算使用垃圾收集或 ARC，或者你需要支持一些不能够迁移到 ARC 上运行的代码，那么你就需要知道如何管理内存。也就是说，需要学习引用计数是如何工作的。

一般概念就是：当创建对象时，初始的引用计数为 1。为保证对象的存在，每当创建引用到对象需要为引用数加 1，可以给对象发送 `retain` 消息：

```
[myFraction retain];
```

当不再需要对象时，通过给对象发送 `release` 消息，为引用计数减 1，语句如下：

```
[myFraction release];
```

当对象的引用计数为 0 时，系统就知道这个对象不再需要使用了（依照理论，在应用中不再有引用到这个对象），所以可以释放它的内存。通过给对象发送 `dealloc` 消息发起这个过程。在多数情况下，会使用继承自 `NSObject` 的 `dealloc` 方法。然而，为了能够释放由对象创建或保持的实例变量或者其他对象，需要覆盖 `dealloc` 方法。例如，在你的类中，有一个实例变量为 `NSArray` 的对象，并且为它创建了一个数组，当对象销毁时，你需要负责释放数组，`dealloc` 方法中可以做这些事情。

手工管理引用计数策略成功与否取决于程序员，需要确保在程序运行时，引用计数能够合理递增和递减。系统能够帮助你做一些事情，但不是全部，其他的事情需要由你来做。

当你使用手工管理引用计数时，需要注意到 Foundation 框架中的一些方法

可能会增加对象的引用计数，比如 `NSMutableArray` 的 `addObject:` 方法用于将一个对象添加到数组，或者 `UIView` 的 `addSubview:` 方法用于将一个视图作为子视图添加到另一个视图中使用。有一些方法会减少对象的引用计数，如 `removeObjectAtIndex:` 和 `removeFromSuperview:` 这两个方法。

当对象被销毁后（它的引用计数减至 0，并且调用过 `dealloc` 方法），对象的引用变得失效。如果你有一个这样的引用，通常被称为悬挂指针（`dangling pointer`）的引用。给悬挂指针发送消息往往会出现意外，甚至应用发生崩溃。一些程序会给已经释放的对象发送 `release` 消息，主要是未追踪对象的 `retain` 和 `release`。这样过度释放对象，会引起程序崩溃。

17.2.1 对象引用和自动释放池

可能会编写这样一个方法，先创建一个对象（使用 `alloc`），然后将它作为方法调用的结果返回。这样就进入一个困境：尽管方法不再使用这个对象，但是并不能释放它，因为需要将这个对象作为方法的返回值。`NSAutoreleasePool` 类创建的目的就是希望能够解决这个问题，自动释放池可以帮助追踪需要延迟一些时间释放的对象。通过给自动释放池发送 `drain` 消息，自动释放池会被清理，对象会被释放。

给对象发送 `autorelease` 消息，将一个对象添加到由自动释放池维护的对象列表中，语句如下：

```
[result autorelease];
```

程序中使用来自 `Foundation`、`UIKit`、`AppKit` 框架的类时，首先需要创建一个自动释放池，这样来自这些框架的类才会创建并返回自动释放的对象，需要在程序中使用 `@autoreleasepool` 指令：

```
@autoreleasepool {
    statements
}
```

如果在 `Xcode` 中创建一个新项目时未启用 `ARC` 特性，生成的模板文件在 `main` 起始的位置就会有这条语句。

当执行到 `autorelease` 块的末尾时，系统会释放池。这将影响到所有发送过 `autorelease` 消息并被添加到自动释放池中的对象，会对池中的每个对象发送

release 消息。当这些对象的引用计数减至 0 时，会发送出 dealloc 消息，并且它们的内存将会被释放。

需要注意，自动释放池并不包含实际的对象，而是只包含对象的引用，对象将在自动释放池清理的时候被释放。

并不是所有新创建的对象都会被添加到自动释放池中。事实上，任何由以 alloc、copy、mutableCopy 和 new 为前缀的方法创建的对象都不会被自动释放。在这种情况下，可以说你拥有这个对象。当你拥有一个对象时，需要在使用完这些对象后负责释放这些对象的内存。主动给这些对象发送 release 消息，或者给对象发送 autorelease 消息将对象加入到自动释放池中。

下面是代码清单 3-3 的 main 函数使用手工引用计数的代码：

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

        Fraction *frac1 = [[Fraction alloc] init];
        Fraction *frac2 = [[Fraction alloc] init];

        // 设置第一个分数为 2/3
        [frac1 setNumerator: 2];
        [frac1 setDenominator: 3];

        // 设置第二个分数为 3/7
        [frac2 setNumerator: 3];
        [frac2 setDenominator: 7];

        //显示分数

        NSLog(@"First fraction is:");
        [frac1 print];

        NSLog(@"Second fraction is:");
        [frac2 print];

        [frac1 release];
        [frac2 release];
    }
    return 0;
}
```

注意，在 main 结尾部分使用 release 在两个 alloc 的分数对象上。尽管应用终止的时候会释放所有的内存，这个例子是在说明，当你使用完拥有的对象时，

需要自己主动释放这些对象。

这个例子还表明在 `main` 开始的位置创建自动释放池，在应用返回之前，`main` 结束的位置清理自动释放池。

下面看看代码清单 7-5 中 `Fraction` 类中的 `add` 方法：

```
-(Fraction *) add: (Fraction *) f
{
    // 对两个分数求和：
    // a/b + c/d = (a*d) + (b*c) / (b * d)
    // 存储相加后的结果

    Fraction *result = [[Fraction alloc] init];

    result.numerator = numerator * f.denominator + denominator *
        f.numerator;
    result.denominator = denominator * f.denominator;

    [result reduce];
    return result;
}
```

如果使用手工内存管理，这个方法会出现一个问题，创建的对象会在计算执行后，被方法作为结果返回。因为方法返回这个对象，所以并不能释放它。解决这个问题最好的方法是自动释放这个对象，这样它的值就能够作为结果返回，能够延迟对象的释放直到自动释放池被清理。需要充分利用这一点，`autorelease` 方法返回它的消息接收者，将它嵌入到表达式中：

```
Fraction *result = [[[Fraction alloc] init] autorelease];
```

或者这样：

```
return [result autorelease];
```

17.3 事件循环和内存分配

Cocoa 和 iOS 应用运行在所谓的事件循环中。事件是伴随着某些行为（例如，按下 iPhone 的一个按钮）或者隐性的行为（例如，通过网络获取一些内容）而发生的。为了处理这样一个新事件，系统会创建一个新的自动释放池，可能会调用到你应用中的一些方法来处理这个事件。当处理完这个事件，并从你的方法返回后，系统开始等待下一个事件的发生。然而在此之前，系统会清理自

动释放池，这就意味着在事件处理过程中创建的自动释放对象都将被销毁，除非对象使用 `retain`，才能从清空自动释放池的过程中幸存下来。当使用手工引用计数时，你需要对自动释放池进行思考，在事件循环结束后清理自动释放池幸存的这些对象。

查看一下来自 iOS 应用程序的接口部分，下面定义了 `MyView` 类包含一个名为 `data` 的属性：

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

@interface myView : UIView
@property (nonatomic, retain) NSMutableArray *data;
@end
```

属性 `data` 的 `retain` 特性（仅会被未使用 ARC 编译的代码识别）表明设值方法需要保持已赋值给属性的任何对象，首先会释放旧的值。假设已经在实现部分为 `data` 属性的存取方法使用同步（`synthesize`）。进一步假定有一个 `viewDidLoad` 方法会在视图载入内存时被系统调用（学习 iOS 程序编程会掌握更多内容，在这里并不做细述）。

在 `viewDidLoad` 方法内需要创建一个 `data` 数组，插入以下语句到方法中：

```
data = [NSMutableArray array];
```

前面提到，Foundation 中的方法默认会创建自动释放的对象，`array` 方法创建了一个自动释放的数组，直接将它赋值给实例变量 `data`。问题是数组会在当前事件结束后被立即销毁。这是因为创建的自动释放池会在事件循环结束后被清理。为了确保数组在事件循环后还能够存在，需要做一些改变，下面 3 条不同的代码可以实现：

```
data = [[NSMutableArray array] retain]; // 从释放池中幸存
```

或

```
data = [[NSMutableArray alloc] init]; // 并不会自动释放
```

或

```
self.data = [NSMutableArray array]; // 使用设值方法
```

最后一条语句中，因为 `data` 属性使用了 `retain` 特性，自动释放的数组会被保持（对 `self.data` 进行赋值会调用设值方法）。注意，上述 3 条语句都需要覆盖

`dealloc` 方法用于在销毁 `myView` 对象的时候释放数组。

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

（调用 `super` 是为了任何继承来的对象都能够被释放。使用手工内存管理的另一麻烦，就是在释放完自身的一些对象之后，都需要使用`[super dealloc]`）。在大多数用例中，使用框架中的方法创建一个新的对象时，需要选择是创建自动释放（`autorelease`）的对象还是使用 `alloc` 方法创建对象。如果在事件循环结束和自动释放池清理之前，你的应用会创建许多对象，那么你可能希望使用 `alloc` 方法。这样就可以在使用完这些对象时能够及时进行释放，而不需要等待事件处理结束。

在手工引用计数环境中，可以为属性添加 `atomic`（默认）或 `nonatomic` 特性，也可以添加 `assign`（默认）、`retain`、`copy` 特性。

当使用设值方法为属性赋值时，看看 `assign`、`retain` 和 `copy` 这 3 个特性是如何实现的。例如：

```
self.property = newValue;
```

assign 特性会是这样：

```
property = newValue;
```

retain 特性会是这样：

```
if (property != newValue) {
    [property release];
    property = [newValue retain];
}
```

copy 特性会是这样：

```
if (property != newValue) {
    [property release];
    property = [newValue copy];
}
```

17.4 手工内存管理规则的总结

下面是一些不使用垃圾收集或 ARC 编译的项目规划：

- 如果需要保持一个对象不被销毁，可以使用 `retain`。在使用完对象后，需要使用 `release` 进行释放。
- 给对象发送 `release` 消息并不会必须销毁这个对象，只当这个对象的引用计数减至 0 时，对象才会被销毁。然后系统会发送 `dealloc` 消息给这个对象用于释放它的内存。
- 对使用了 `retain` 或者 `copy`、`mutableCopy`、`alloc` 或 `new` 方法的任何对象，以及具有 `retain` 和 `copy` 特性的属性进行释放，需要覆盖 `dealloc` 方法，使得在对象被释放的时候能够释放这些实例变量。
- 在自动释放池被清空时也会为自动释放的对象做些事情。系统每次都会在自动释放池被释放时发送 `release` 消息给池中的每个对象。如果池中的对象引用计数降为 0，系统会发送 `dealloc` 消息销毁这个对象。
- 如果在方法中不再需要用到这个对象，但需要将其返回，可以给这个对象发送 `autorelease` 消息用以标记这个对象延迟释放。`autorelease` 消息并不会影响到对象的引用计数。
- 当应用终止时，内存中的所有对象都会被释放，不论它们是否在自动释放池中。
- 当开发 Cocoa 或者 iOS 应用程序时，随着应用程序的运行，自动释放池会被创建和清空（每次的事件都会发生）。在这种情况下，如果要使自动释放池被清空后自动释放的对象还能够存在，对象需要使用 `retain` 方法，只要这些对象的引用计数大于发送 `autorelease` 消息的数量，就能够在池清理后生存下来。

17.5 自动引用计数 (ARC)

ARC 可以避免手工引用计数的一些潜在陷阱。基于此，引用计数仍然被维护和追踪。然而，系统会检测出何时需要保持对象，何时需要自动释放对象，何时需要释放对象，这些你都不用担心。

你不必担心返回了方法内创建的对象。编译器会管理好对象的内存，通过生成正确的代码去自动释放或保持返回的对象。

17.6 强变量

通常，所有对象的指针变量都是强变量。也就是说，将对象的引用赋给变量使对象自动保持。然后，旧对象的引用会在赋值前被释放。最终，强变量默认会被初始化为零。无论它是实例变量、局部变量还是全局变量，这都成立。

下面是创建并设置两个 `Fraction` 对象的代码：

```
Fraction *f1 = [[Fraction alloc] init];
Fraction *f2 = [[Fraction alloc] init];

[f1 setTo: 1 over: 2];
[f2 setTo: 2 over: 3];
```

当你使用手工内存管理时，需要编写以下语句：

```
f2 = f1;
```

复制 `Fraction` 对象的引用 `f1` 到 `f2`，会导致 `Fraction` 对象的引用 `f2` 丢失，它的值被覆盖，从而产生内存泄漏，即一个变量不再被引用，但又不能够释放。

如果使用 ARC，`f1` 和 `f2` 都是强变量。前面的赋值其实会是这样的：

```
[f1 retain]; //保留新的值
[f2 release]; //释放旧值
f2 = f1;      //复制引用
```

当然，你并不会看到是如何运作的，因为编译器都帮你做了，你只要编写赋值语句就可以了。

因为所有的对象变量默认都是强变量的，所以不需要先声明。但你仍然可以为变量使用关键字 `__strong`：

```
__strong Fraction *f1;
```

值得注意的是，属性默认不是 `strong`，其默认的特性是 `unsafe_unretained`（相当于 `assign`）。你需要这样为属性声明 `strong` 特性：

```
@property (strong, nonatomic) NSMutableArray *birdNames;
```

编译器会保证在事件循环中通过对赋值执行保持操作强属性能够存活下来。带有 `unsafe_unretained`（相当于 `assign`）或 `weak` 的属性不会执行这些操作。

17.7 弱变量

有时候需要为对象建立两者之间的关系，每个对象需要引用到其他的对象（可能是两个对象或者对象的循环链表）。例如，iOS 应用程序使用视图对象在屏幕上展现图形。视图会维护一个层次结构。一个视图上可能展现一张图片，在图片视图内可能展现出图片的标题。将图片视图设置为主视图，标题视图设置为子视图。当主视图显示的时候，子视图会自动显示出来。你可以把主视图想象为父视图，标题视图想象为子视图。图片主视图具有子视图。

当使用视图继承结构时，父视图确实持有子视图的引用，但子视图能够知道父视图也是很有帮助的。所以父视图会持有子视图的引用，同时子视图也会持有父视图的引用。这种循环引用会引起一些问题。例如，当父视图销毁时如何处理？子视图对父视图的引用不再有效。实际上，试图引用到不存在的父视图会引起应用的崩溃。

当两个对象都持有彼此的强引用时，将会产生循环保持（retain cycle）。如果对象仍然有引用，系统将不能销毁这个对象。如果两个对象都强引用到彼此，这样就不可以被销毁。

解决这个问题可以通过创建其他类型的对象变量，并允许使用不同类型的引用。这种引用被称为弱引用，能够建立在两个对象之间。在这个例子中，弱引用建立在子到父。为什么呢？因为考虑到一个拥有其他对象的对象（在这个例子中是父视图）是强引用，其他对象可能是弱引用。

通过父视图持有子视图的强引用，子视图持有父视图的弱引用，这样就没有循环保持。弱变量也不能阻止引用的对象被销毁。

当你声明一个弱变量时，系统会追踪赋值给这个变量的引用。当引用的对象释放时，弱变量会被自动设置为 nil。这也避免了无意间给这个变量发送消息引起的崩溃。因为变量被设置为 nil，给 nil 对象发送任何消息不会有反应，从而有效避免了崩溃的发生。

可以使用 `_weak` 关键字声明一个弱变量：

```
_weak UIView *parentView;
```

或者为属性指定 `weak` 特性：

```
@property (weak, nonatomic) UIView *parentView;
```

弱变量能够和代理（delegate）很好地协作。创建一个代理的弱变量引用，如果代理对象被销毁，变量就会被清零。ARC 出现之前，这能够为程序员避免一些令人头痛的各种系统崩溃。

需要注意的是，在 iOS 4 和 Mac OS X v10.6 中不支持弱变量。在这种情况下，你仍然可以为属性使用 `unsafe_unretained`（或 `assign`）特性，或者将变量声明为 `__unsafe_unretained`。然而当引用的对象被销毁时，变量不再被清零。

17.8 @autoreleasepool 块

本书中的每个例子都会在 `main` 中有 `@autoreleasepool` 指令，这个指令围住的语句块定义了自动释放池的上下文。任何在这个上下文中创建的对象都是自动释放的，在自动释放池块结束的时候销毁这些对象（除非编译器在自动释放池块结束后还需要保证这个对象的存在）。

如果在程序中产生了大量临时的对象（在循环中执行代码），如果你希望在程序中创建多个自动释放池块。例如，以下代码段表示如何使用自动释放池块管理每次由 `for` 循环迭代创建的临时对象：

```
for (i = 0; i < n; ++i) {
    @autoreleasepool {
        ... // 与临时对象打交道
    }
}
```

在本章开始时提到，Cocoa 和 iOS 应用运行在事件循环中。为了处理新的事件，系统会创建一个新的自动释放池上下文，调用到应用中的一些方法用于处理事件，再从方法返回，系统会继续等待下一个事件的发生。然而，做这些事情之前，自动释放池上下文已经结束，意味着自动释放的对象可能会被销毁。使用 ARC，这些都会在“底层”发生，你不需要为此担心。

需要注意的是，`autoreleasepool` 对于应用程序使用或者不使用 ARC 进行编译的情形均适用。

17.9 方法名和非 ARC 编译代码

ARC 与未使用 ARC 编译的代码一起运行。例如，需要链接到旧的框架。

只要非 ARC 代码与标准的 Cocoa 命名规则一致，都会运行良好。当 ARC 遇到方法调用时，会检查方法名，如果名字以 `alloc`、`new`、`copy`、`mutableCopy` 或 `init` 这些词开头，它会假定方法返回对象的所有者给方法的调用者。

这里所讨论的“词”使用了驼峰拼写法（`camelCase`），即名字中每个新词的第一个字母都以大写字母开头。编译器会假定具有 `allocFraction`、`newAddressCard` 和 `initWithWidth:andHeight:` 名字的方法都会返回对象的拥有者，而 `newlyWeds`、`copycat` 和 `initials` 则不会。这些在 ARC 中都会自动处理，不需要操心，除非你使用的方法不符合标准的命名规则。在这种情况下，根据方法的名字，需要使用到这种方式隐性通知编译器该方法会返回对象的拥有者。

注意，如果你试图合成属性，而属性的名字是以第一段提到的特殊词开头，编译器会提示一些错误。

复制对象

本章将讨论复制对象相关的一些细节。我们将介绍浅复制和深复制的概念，并讨论如何在 Foundation 框架下实现对象复制。

在第 8 章“继承”中，我们讨论了使用简单的赋值语句将对象赋值给另一个对象时发生的情况，比如：

```
origin = pt;
```

在这个例子中，`origin` 和 `pt` 都是带有两个整型实例变量 `x` 和 `y` 的 `XYPoint` 对象。

这样赋值的结果仅仅是将对象 `pt` 的地址复制到 `origin` 中。在赋值操作结束时，两个变量都指向内存中的同一个地址。使用一条消息对实例变量进行修改，如：

```
[origin setX: 100 andY: 200];
```

改变了 `origin` 和 `pt` 变量共同引用的 `XYPoint` 对象的 `x`、`y` 坐标，因为它们都引用内存中的同一个对象。

这同样适用于 Foundation 对象：将一个变量赋值给另一个对象仅仅创建另一个对这个对象的引用。所以，如果 `dataArray` 和 `dataArray2` 都是 `NSMutableArray` 对象，那么语句

```
dataArray2 = dataArray;  
[dataArray2 removeObjectAtIndex: 0];
```

将从这两个变量引用的同一个数组中删除第一个元素。

18.1 copy 和 mutableCopy 方法

Foundation 类实现了名为 `copy` 和 `mutableCopy` 的方法，可以使用这些方法创建对象的副本。通过实现一个符合<NSCopying>协议（用于制作副本）的方法来完成此任务。如果类必须区分要产生对象的是可变副本还是不可变副本，还需要根据<NSMutableCopying>协议实现一个方法。本章后面将学习如何实现上述方法。

回顾 Foundation 类的 `copy` 方法，给定前面描述的两个 `NSMutableArray` 对象 `dataArray2` 和 `dataArray`，语句

```
dataArray2 = [dataArray mutableCopy];
```

在内存中创建了一个新的 `dataArray` 副本，并复制了它的所有元素。随后，执行语句

```
[dataArray2 removeObjectAtIndex: 0];
```

删除了 `dataArray2` 中的第一个元素，但却不删除 `dataArray` 中的第一个元素。代码清单 18-1 说明了这种情况。

代码清单 18-1

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSMutableArray *dataArray = [NSMutableArray arrayWithObjects:
            @"one", @"two", @"three", @"four", nil];
        NSMutableArray *dataArray2;

        //简单赋值

        dataArray2 = dataArray;
        [dataArray2 removeObjectAtIndex: 0];

        NSLog(@"dataArray: ");
        for ( NSString *elem in dataArray )
            NSLog(@"  %@", elem);

        NSLog(@"dataArray2: ");

        for ( NSString *elem in dataArray2 )
            NSLog(@"  %@", elem);
```

```

//复制一份，然后删除副本的第一个元素

dataArray2 = [dataArray mutableCopy];
[dataArray2 removeObjectAtIndex: 0];

NSLog(@"dataArray: ");

for ( NSString *elem in dataArray )
    NSLog(@"  %@", elem);

NSLog(@"dataArray2: ");

for ( NSString *elem in dataArray2 )
    NSLog(@"  %@", elem);

}
return 0;
}

```

代码清单 18-1 输出

```

dataArray:
    two
    three
    four
dataArray2:
    two
    three
    four
dataArray:
    two
    three
    four
dataArray2:
    three
    four

```

这个程序定义了可变数组对象 `dataArray`，并分别将其元素设置为字符串对象 `@ "one"`、`@ "two"`、`@ "three"`和`@ "four"`。前面讨论过，赋值语句

```
dataArray2 = dataArray;
```

仅仅创建了对内存中同一数组对象的另一个引用。当从 `dataArray2` 中删除第一个对象并随后输出两个数组对象中的元素时，不出意料，这两个引用中的第一个元素（字符串`@ "one"`）都会消失。

然后，创建一个 `dataArray` 的可变副本并将它赋值给 `dataArray2` 的最终副本。这就在内存中创建了两个截然不同的可变数组，两者都包含 3 个元素。现在，删除 `dataArray2` 中的第一个元素时，不会对 `dataArray` 的内容有任何影响，正如程序最后输出的一样。

注意，产生一个对象的可变副本并不要求被复制的对象本身是可变的。这种情况同样适用于不可变副本：可以创建可变对象的不可变副本。

18.2 浅复制与深复制

代码清单 18-1 使用不可变字符串来填充 `dataArray` 的元素（回忆一下，常量字符串对象是不可变的）。在代码清单 18-2 中，将使用可变字符串代替它来填充数组，这样就可以改变数组中的一个字符串。观察代码清单 18-2，看自己能否理解程序的输出结果。

代码清单 18-2

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSMutableArray *dataArray = [NSMutableArray arrayWithObjects:
            [NSString stringWithString: @"one"],
            [NSString stringWithString: @"two"],
            [NSString stringWithString: @"three"],
            nil
        ];
        NSMutableArray *dataArray2;
        NSMutableString *mStr;

        NSLog(@"dataArray: ");
        for ( NSString *elem in dataArray )
            NSLog(@"  %@", elem);

        //复制一份，然后删除副本的第一个元素

        dataArray2 = [dataArray mutableCopy];

        mStr = dataArray[0];
        [mStr appendString: @"ONE"];
```

```

NSLog(@"dataArray: ");
for ( NSString *elem in dataArray )
    NSLog(@"  %@", elem);

NSLog(@"dataArray2: ");
for ( NSString *elem in dataArray2 )
    NSLog(@"  %@", elem);
}
return 0;
}

```

代码清单 18-2 输出

```

dataArray:
    one
    two
    three
dataArray:
    oneONE
    two
    three
dataArray2:
    oneONE
    two
    three

```

使用下面的语句检索 **dataArray** 的第一个元素

```
mStr = dataArray[0];
```

然后，使用下面的语句将字符串@"ONE"附加到这个元素中：

```
[mStr appendString: @"ONE"];
```

注意，原始数组及其副本中第一个元素的值都被修改了。或许你能理解为什么 **dataArray** 的第一个元素发生改变，但不明白为什么它的副本也会改变。从集合中获取元素时，就得到了这个元素的一个新引用，但并不是一个新副本。所以，**dataArray[0]** 返回的对象与 **dataArray** 中的第一个元素都指向内存中的同一个对象。随后，修改字符串对象 **mStr** 的副作用就是同时改变了 **dataArray** 的第一个元素，从输出结果中可以看到。

但你制作的副本怎么样了？为什么它的第一个元素也会改变？这与默认的浅复制方式有关。它意味着使用 **mutableCopy** 方法复制数组时，在内存中为新的数组对象分配了空间，并且将单个元素复制到新数组中。然而将原始数组中

的每个元素复制到新位置意味着：仅将引用从一个数组元素复制到另一个数组元素。这样做的最终结果，就是这两个数组中的元素都指向内存中的同一个字符串。这与将一个对象赋值给另一个对象没有区别，就像我们本章开始提到的一样。

若要为数组中的每个元素创建完全不同的副本，需要执行所谓的深复制。这就意味着要创建数组中的每个对象内容的副本，而不仅是这些对象的引用的副本（并且考虑一下，如果一个数组中的元素本身是数组对象，深复制意味着该如何处理）。然而使用 Foundation 类的 `copy` 或 `mutableCopy` 方法时，深复制并不是默认执行的。在第 19 章“归档”中，我们将向你展示如何使用 Foundation 的归档功能来创建对象的深复制。

例如，复制一个数组、词典或集合时，会获得这些集合的新副本。然而，如果想要更改其中一个集合，而不是它的副本，则可能需要为单个元素创建自己的副本。例如，假设想要更改代码清单 18-2 中 `dataArray2` 的第一个元素，但不更改 `dataArray` 的第一个元素，可以创建一个新字符串（使用 `stringWithString:` 之类的方法），并将它存储到 `dataArray2` 的第一个位置，语句如下：

```
mStr = [NSMutableString stringWithString: dataArray[0]];
```

然后，可以更改 `mStr`，并使用 `replaceObjectAtIndex:withObject:` 方法将它添加到数组中，语句如下：

```
[mStr appendString @"ONE"];
[dataArray2 replaceObjectAtIndex: 0 withObject: mStr];
```

如果顺利的话，你会发现即使替换了数组中的对象之后，`mStr` 和 `dataArray2` 的第一个元素仍指向内存中的同一个对象。这意味着随后在程序中对 `mStr` 做的修改也将更改数组的第一个元素。

18.3 实现<NSCopying>协议

如果尝试使用自己类（例如，地址簿）中的 `copy` 方法，语句如下：

```
NewBook = [myBook mutableCopy];
```

将会收到一条出错消息，内容可能如下：

```
*** -[AddressBook copyWithZone:]: selector not recognized
```

```
*** Uncaught exception:
*** -[AddressBook copyWithZone:]: selector not recognized
```

正如注释的那样，要实现使用自己的类进行复制，必须根据<NSCopying>协议实现其中一两个方法。

我们将展示如何为 `Fraction` 类添加 `copy` 方法，这个类在第一部分“Objective-C 语言”中被广泛使用。注意，这里描述的复制策略的技巧非常适用于你自己的类。如果这些类是任何 Foundation 类的子类，那么可能需要实现较为复杂的复制策略。必须考虑这样一个事实：超类可能已经实现了它自己的复制策略。

还记得 `Fraction` 类包含两个整型实例变量，分别名为 `numerator` 和 `denominator`。要产生其中一个对象的副本，需要分配一个新分数的空间并简单地将这两个整数的值复制到新分数中。

实现<NSCopying>协议时，类必须实现 `copyWithZone:`方法来响应 `copy` 消息（这条 `copy` 消息仅将一条带有 `nil` 参数的 `copyWithZone:`消息发送给你的类）。注意，如果想要区分可变副本和不可变副本，还需要根据<NSMutableCopying>协议实现 `mutableCopyWithZone:`方法。如果两个方法都实现，那么 `copyWithZone:`应该返回不可变副本，而 `mutableCopyWithZone:`将返回可变副本。产生对象的可变副本并不要求被复制的对象本身也是可变的（反之亦然）；想要产生不可变对象的可变副本是很合理的（例如，考虑字符串对象）。

@interface 指令应该如下：

```
@interface Fraction: NSObject <NSCopying>
```

`Fraction` 是 `NSObject` 的子类，并且符合 `NSCopying` 协议。

在实现文件 `Fraction.m` 中，为新方法添加下列定义：

```
-(id) copyWithZone: (NSZone *) zone
{
    Fraction *newFract = [[Fraction allocWithZone: zone] init];

    [newFract setTo: numerator over: denominator];
    return newFract;
}
```

参数 `zone` 与不同的存储区有关，你可以在程序中分配并使用这些存储区。只有在编写要分配大量内存的应用程序并且想要通过将空间分配分组到这些存

储区中来优化内存分配时，才需要处理这些 `zone` 参数。可以使用传递给 `copyWithZone:` 的值，并将它传给名为 `allocWithZone:` 的内存分配方法。这个方法在指定存储区中分配内存。

分配新的 `Fraction` 对象之后，将接收者的 `numerator` 和 `denominator` 变量复制到其中。`copyWithZone:` 方法应该返回对象的新副本，这个对象就是在你的方法中实现的。

代码清单 18-3 测试了你的新方法。

代码清单 18-3

// 复制分数

```
#import "Fraction.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        Fraction *f1 = [[Fraction alloc] init];
        Fraction *f2;

        [f1 setTo: 2 over: 5];
        f2 = [f1 copy];

        [f2 setTo: 1 over: 3];

        [f1 print];
        [f2 print];
    }
    return 0;
}
```

代码清单 18-3 输出

```
2/5
1/3
```

该程序创建了一个名为 `f1` 的 `Fraction` 对象，并将其设置为 $2/5$ 。然后，它调用 `copy` 方法来产生副本，`copy` 方法向你的对象发送 `copyWithZone:` 消息。这个方法产生了一个新的 `Fraction`，将 `f1` 的值复制到其中，并返回结果。回到 `main` 函数中，你将这个结果赋给 `f2`。随即，将 `f2` 中的值设置为分数 $1/3$ ，这样就验证了这些操作对原始分数 `f1` 没有影响。将程序中的下列语句

```
f2 = [f1 copy];
```

简单地改成

```
f2 = f1;
```

观察获得的不同结果。

如果你的类可以产生子类，那么 `copyWithZone:` 方法将被继承。在这种情况下，该方法中的程序行

```
Fraction *newFract = [[Fraction allocWithZone: zone] init];
```

应该改为

```
id newFract = [[[self class] allocWithZone: zone] init];
```

这样，可以从该类分配一个新对象，而这个类是 `copy` 的接收者（例如，如果它产生了一个名为 `NewFraction` 的子类，那么应该确保在继承的方法中分配了新的 `NewFraction` 对象，而不是 `Fraction` 对象）。

如果编写一个类的 `copyWithZone:` 方法，而该类的超类也实现了 `<NSCopying>` 协议，那么应该先调用超类的 `copy` 方法以复制继承来的实例变量，然后加入自己的代码已复制想要添加到该类中任何附加的实例变量（如果有的话）。

你必须确定是否在类中实现浅复制或深复制，并为其编写文档，以告知类的其他使用者。

18.4 用设值方法和取值方法复制对象

只要实现设值方法或取值方法，都应该考虑实例变量中存储的内容、要检索的内容及是否需要保护这些值。例如，考虑使用相应的赋值方法来设置 `AddressCard` 对象的名称时：

```
newCard.name = newName;
```

假设 `newName` 是一个字符串对象，它包含新地址卡的名称。假定在设置方法函数内，你只是简单地将参数赋值给相应的实例变量：

```
-(void) setName: (NSString *) theName
{
    name = theName;
}
```

现在,如果程序随后修改了 `newName` 中的一些字符会怎样(如果 `newName` 是一个可变的字符串对象呢;使用 `NSString` 对象对其赋值呢,前者是后者的子类)?这会无意间改变地址卡片中相应的字段,因为它们都引用到相同的字符串对象。

综合前面讨论的内容,为了避免操作不小心而改变其他对象的值,比较安全的做法是在设置方法中将对象复制一份。

如果没有合成设置方法,可以编写使用 `copy` 版本的 `setName:` 方法,正如下面这样:

```
-(void) setName: (NSString *) theName
{
    name = [theName copy];
}
```

第 15 章“数字、字符串和集合”中已提到,如果属性声明中指定了 `copy` 特性,合成的方法会使用类的 `copy` 方法(自己编写的或继承自父类),`property` 声明如下:

```
@property (nonatomic, copy) NSString *name;
```

与 `@synthesize` 指令一起使用,生成的方法与下面这个方法行为相似:

```
-(void) setName: (NSString *) theName
{
    if (theName != name)
        name = [theName copy];
}
```

此处使用 `nonatomic` 是为了告诉系统不要使用互斥(`mutex`)锁定保护属性的存取方法。编写线程安全代码的人会使用互斥锁定防止同一段代码中两个线程同时执行,如果同时执行,会导致可怕的问题。然而这种锁定也会让程序变慢,如果知道这段代码只会在单线程中运行,就可避免使用这种锁定方法。

如果未指定 `nonatomic` 或者指定了 `atomic`(默认值),使用互斥锁可以保护实例变量,能够保证独占访问一个实例变量,并且能够阻止潜在的竞争条件的发生,比如多线程的程序中多个线程试图并发访问同样的实例变量。

注意

属性并没有 `mutableCopy` 特性。即使是可变的实例变量,也是使用 `copy` 特性,正如方法 `copyWithZone:` 的执行结果。所以,按照约定会生成一个对

象的不可变副本。

关于保护实例变量值的讨论同样适用于取值函数。如果返回一个可变对象，那么必须确保对返回值的更改不影响你的实例变量的值。在这样的情况下，可以生成实例变量的副本，并将它替代原始值作为返回值。

回到 `copy` 方法的实现，如果正在复制的实例变量包含不可变的对象（如不可变的字符串对象），那么可能不需要生成这个对象内容的新副本。仅通过保持该对象来生成它的新引用，可能就足够了。例如，为 `AddressCard` 类实现 `copy` 方法时，这个类包含 `name` 和 `email` 成员，下面实现的 `copyWithZone:` 方法就足够了。

```
-(AddressCard *) copyWithZone: (NSZone *) zone
{
    AddressCard *newCard = [[AddressCard allocWithZone: zone] init];

    [newCard assignName: name andEmail: email];
    return newCard;
}

-(void) assignName: (NSString *) theName andEmail: (NSString *) theEmail
{
    name = theName;
    email = theEmail;
}
```

这里没有使用 `setName:andEmail:` 方法来设置实例变量，因为该方法会生成参数的副本，但这个练习的目的不是这样的。所以使用新的方法 `assignName:andEmail:` 为两个变量赋值。

在这里可以不给实例变量赋值（而是对它们进行完全复制），因为复制的卡片并不会影响到原始的卡片对象（包含不可变的字符串对象）中的成员变量 `name` 和 `email`。因为默认的两个实例变量均是强引用（`strong`），简单的赋值操作就会创建其他的引用到这些对象，在第 17 章“内存管理和自动引用计数”中有对这个问题的详细解释。

18.5 练习

1. 根据 `NSCopying` 协议为 `AddressBook` 类实现一个 `copy` 方法。也实现

`mutableCopy` 方法是否可行？为什么？思考一下，如果使用了 `AddressBook` 类 `book` 属性的设值方法会怎样。如果将地址簿作为设置函数的参数传递，谁会拥有这个地址簿？如何修正这个问题？

2. 修改第 8 章中定义的 `Rectangle` 类和 `XYPoint` 类，使其符合 `<NSCopying>` 协议的要求。然后为两个类添加 `copyWithZone:` 方法，确保 `Rectangle` 使用 `XYPoint` 的 `copy` 方法复制它的 `XYPoint` 成员 `origin`。为这些类实现可变和不可变副本是否行得通？解释原因。
3. 创建一个 `NSDictionary` 对象，并使用键/对象对来填充它，然后产生可变和不可变副本。这些复制是深复制还是浅复制？验证你的答案。

19

归 档

在 Objective-C 语言中，归档是一个过程，即用某种格式来保存一个或多个对象，以便以后还原这些对象。通常，这个过程包括将（多个）对象写入文件中，以便以后读取该对象。我们将在本章讨论两种归档数据的方法：属性列表和带键值的编码。

19.1 使用 XML 属性列表进行归档

Mac OS X 上的应用程序使用 XML 属性列表（或 `plist`s）存储诸如默认参数选择、应用程序设置和配置信息这样的数据，因此，了解如何创建和读取这些数据很有用。然而，这些列表的归档用途是有限的，因为当为某个数据结构创建属性列表时，并没有保存特定的对象类，也没有存储对同一对象的多个引用，同样，也没有保持对象的可变性。

注意

所谓的“老式”属性列表存储数据的格式与 XML 属性列表不同。如果可能，尽量在程序中使用 XML 属性列表。

如果你的对象是 `NSString`、`NSDictionary`、`NSArray`、`NSData` 或 `NSNumber` 类型，你可以使用这些类中实现的 `writeToFile:atomically:` 方法将数据写到文件中。在写出某个字典或数组的情况下，该方法可以使用 XML 属性列表的格式写出数据。代码清单 19-1 显示了如何将在第 15 章“数字、字符串和集合”中作为简易术语表而创建的字典作为属性列表写入文件中。

代码清单 19-1

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSDictionary *glossary = @{
            @"abstract class": @"A class defined so other classes can inherit from it.",
            @"adopt": @"To implement all the methods defined in a protocol",
            @"archiving": @"Storing an object for later use. "
        };

        if ([glossary writeToFile: @"glossary" atomically: YES] == NO)
            NSLog(@"Save to file failed!");
    }
    return 0;
}
```

`writeToFile:atomically:`消息被发送给字典对象 `glossary`，使字典以属性列表的形式写到文件 `glossary` 中。`atomically` 参数被设为 YES，表示希望首先将字典写入临时备份文件中，并且一旦成功，将把最终数据转移到名为 `glossary` 的指定文件中。这是一种安全措施，它保护文件在一些情况下（如系统在执行操作的过程中崩溃时）免受破坏。在这种情况下，原始的 `glossary` 文件（如果该文件已存在）不会受到损害。

如果查看代码清单 19-1 中创建的 `glossary` 文件，它的内容可能如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>abstract class</key>
    <string>A class defined so other classes can inherit from it.</string>
    <key>adopt</key>
    <string>To implement all the methods defined in a protocol</string>
    <key>archiving</key>
    <string>Storing an object for later use. </string>
</dict>
</plist>
```

从所创建的 XML 文件中可以看到，是以一种键（`<key>...</key>`）值（`<string>...</string>`）对的形式将字典写入文件的。

当根据字典创建属性列表时，字典中的键必须全都是 NSString 对象。数组中的元素或字典中的值可以是 NSString、NSArray、NSDictionary、NSData、NSDate 或 NSNumber 对象。

若要将文件中的 XML 属性列表读入你的程序，使用 dictionaryWithContentsOfFile: 或 arrayWithContentsOfFile: 方法。要读取数据，使用 dataWithContentsOfFile: 方法；要读取字符串对象，使用 stringWithContentsOfFile: 方法。代码清单 19-2 读取了代码清单 19-1 中编写的术语表，然后输出其内容。

代码清单 19-2

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSDictionary *glossary;

        glossary = [NSDictionary dictionaryWithContentsOfFile: @"glossary"];

        for (NSString *key in glossary)
            NSLog ("%@: %@", key, glossary[key]);
    }
    return 0;
}
```

代码清单 19-2 输出

```
archiving: Storing an object for later use.
abstract class: A class defined so other classes can inherit from it.
adopt: To implement all the methods defined in a protocol
```

你的属性列表不必从 Objective-C 程序中创建，属性列表可以来自任何源，即可以使用简单的文本编辑器，或使用 Mac OS X 系统中位于 /Developer/Applications/Utilities 目录下的 PropertyList Editor 程序来创建属性列表。如果计划在程序中使用属性列表，可以查看一下 NSPropertyListSerialization 类，使用这个类在文件中写入或读取属性列表可以在不同的平台之间移植。

19.2 使用 NSKeyedArchiver 归档

若要将各种类型的对象存储到文件中，而且不仅仅是字符串、数组和字典

类型，有一种更灵活的方法，就是利用 `NSKeyedArchiver` 类创建带键（keyed）的档案来完成。

Mac OS X 从版本 10.2 开始支持带键的档案。在此之前，要使用 `NSArchiver` 类创建连续的（sequential）归档。连续的归档需要完全按照写入时的顺序读取归档中的数据。

在带键的档案中，每个归档字段都有一个名称。归档某个对象时，会为它提供一个名称，即键。从归档中获取该对象时，是根据这个键来检索它的。这样，可以按照任意顺序将对象写入归档并进行检索。另外，如果向类中添加了新的实例变量或删除了实例变量，程序也可以进行处理。

代码清单 19-3 展示了如何使用 `NSKeyedArchiver` 类中的 `archiveRootObjectToFile:` 方法将 `glossary` 存储到磁盘中。

代码清单 19-3

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSDictionary *glossary = @{
            @"abstract class": @"A class defined so other classes can inherit from it.",
            @"adopt": @"To implement all the methods defined in a protocol",
            @"archiving": @"Storing an object for later use. "
        };

        [NSKeyedArchiver archiveRootObject: glossary toFile: @"glossary.archive"];
    }
    return 0;
}
```

代码清单 19-3 并不在终端产生任何输出。但是，语句

```
[NSKeyedArchiver archiveRootObject: glossary toFile: @"glossary.archive"];
```

将字典 `glossary` 写入文件 `glossary.archive` 中。可以为该文件指定任何路径名。在本例中，文件被写入当前目录下。

以后通过 `NSKeyedUnarchiver` 的 `unarchiveObjectWithFile:` 方法将创建的归档文件读入执行程序中，如代码清单 19-4 所示。

代码清单 19-4

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSDictionary *glossary;

        glossary = [NSKeyedUnarchiver unarchiveObjectWithFile:
                    @"glossary.archive"];

        for ( NSString *key in glossary )
            NSLog (@"%@: %@", key, glossary[key]);
    }
    return 0;
}
```

代码清单 19-4 输出

```
abstract class: A class defined so other classes can inherit from it.
adopt: To implement all the methods defined in a protocol
archiving: Storing an object for later use.
```

语句

```
glossary = [NSKeyedUnarchiver unarchiveObjectWithFile:
            @"glossary.archive"];
```

将指定的文件打开并读取文件的内容，该文件必须是前面归档操作的结果。可以为文件指定完整路径名或相对路径名，如本例所示。

在恢复 `glossary` 之后，程序可以简单地通过枚举其内容来验证恢复是否成功。

19.3 编码方法和解码方法

可以使用刚刚描述的方式归档和恢复 `NSString`、`NSArray`、`NSDictionary`、`NSSet`、`NSDate`、`NSNumber` 和 `NSData` 之类的基本 Objective-C 类对象。这还包括嵌套的对象，如包含字符串，甚至其他数组对象的数组。

这意味着在第 15 章中我们不能直接使用这种方法归档 `AddressBook`，因为 Objective-C 系统不知道如何归档 `AddressBook` 对象。如果在程序中插入如下一行来尝试归档它：

```
[NSKeyedArchiver archiveRootObject: myAddressBook toFile: @"addrbook.arch"];
```

运行该程序时将会得到以下消息：

```
*** -[AddressBook encodeWithCoder:]: selector not recognized
*** Uncaught exception: <NSInvalidArgumentException>
*** -[AddressBook encodeWithCoder:]: selector not recognized
archiveTest: received signal: Trace/BPT trap
```

从这些出错消息中，可以看到系统正在 `AddressBook` 类中查找一个名为 `encodeWithCoder:` 的方法，但你从未定义过这样的方法。

要归档前面没有列出的对象，必须告知系统如何归档（或编码）你的对象，以及如何解归档（或解码）它们。这是按照 `<NSCoding>` 协议，在类定义中添加 `encodeWithCoder:` 方法和 `initWithCoder:` 方法实现的。对于本书地址簿的例子，必须向 `AddressBook` 类和 `AddressCard` 类添加这些方法。

每次归档程序想要根据指定的类编码对象时，都将调用 `encodeWithCoder:` 方法，该方法告知归档程序如何进行归档。类似地，每次从指定的类解码对象时，都会调用 `initWithCoder:` 方法。

一般而言，编码方法应该指定如何归档想要保存的对象中的每个实例变量。幸运的是，这些都有帮助可查。对于前面描述的基本 Objective-C 类，可以使用 `encodeObject:forKey:` 方法。相反，对于基本的 C 数据类型（如整型和浮点型），可以使用表 19.1 中列出的某种方法。解码方法 `initWithCoder:` 的工作方式正好相反，它使用 `decodeObject:forKey:` 来解码基本的 Objective-C 类，使用 19-1 中列出的相应解码方法来解码基本的数据类型。

表 19.1 在带键的档案中编码和解码基本数据类型

编码方法	解码方法
<code>encodeBool:forKey:</code>	<code>decodeBool:forKey:</code>
<code>encodeInt:forKey:</code>	<code>decodeInt:forKey:</code>
<code>encodeInt32:forKey:</code>	<code>decodeInt32:forKey:</code>
<code>encodeInt64:forKey:</code>	<code>decodeInt64:forKey:</code>
<code>encodeFloat:forKey:</code>	<code>decodeFloat:forKey:</code>
<code>encodeDouble:forKey:</code>	<code>decodeDouble:forKey:</code>

代码清单 19-5 为 `AddressCard` 类和 `AddressBook` 类都添加了两个编码和解码方法。

代码清单 19-5 AddressCard.h 接口文件

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

@interface AddressCard: NSObject <NSCoding, NSCopying>

@property (copy, nonatomic) NSString *name, *email;

-(void) setName: (NSString *) theName andEmail: (NSString *) theEmail;
-(NSComparisonResult) compareNames: (id) element;
-(void) print;

// 添加 NSCopying 协议的方法
-(void) assignName: (NSString *) theName andEmail: (NSString *) theEmail;

@end
```

下面是要添加到 AddressCard 类实现文件的两个新方法:

```
-(void) encodeWithCoder: (NSCoder *) encoder
{
    [encoder encodeObject: name forKey: @"AddressCardName"];
    [encoder encodeObject: email forKey: @"AddressCardEmail"];
}

-(id) initWithCoder: (NSCoder *) decoder
{
    name = [decoder decodeObjectForKey: @"AddressCardName"];
    email = [decoder decodeObjectForKey: @"AddressCardEmail"];

    return self;
}
```

该程序向编码方法 `encodeWithCoder:` 传入一个 `NSCoder` 对象作为参数。由于 `AddressCard` 类直接继承自 `NSObject`，所以无须担心编码继承的实例变量。如果的确担心，并且知道类的子类符合 `NSCoding` 协议的要求，那么应该用下面的语句开始编码方法，确保继承的实例变量也被编码：

```
[super encodeWithCoder: encoder];
```

对于地址簿来说，有两个名为 `name` 和 `email` 的实例变量。因为它们都是 `NSString` 对象，所以方法依次使用 `encodeObject:forKey:` 方法对它们进行编码，然后将这两个实例变量添加到归档文件中。

`encodeObject:forKey:` 方法编码对象并将其存储在指定的键下，以后可使用

这个键检索对象。键名是任意的，所以只要在检索（编码）数据时使用的名称与归档（编码）时使用的名称相同，就可以指定任意键名。唯一可能出现冲突的情况是，为正在编码的对象子类使用了相同的键。为了防止这种情况出现，制订归档的键时，可将类名放在实例变量名的前面，代码清单 19-5 就是这样做的。

注意，`encodeObject:ForKey:` 方法可以用于任何在其类中实现对应 `encodeWithCoder:` 方法的对象。

解码的过程刚好相反。传递给 `initWithCoder:` 的参数也是 `NSCoder` 对象。不必担心这个参数，只要记住它是获得该消息（对于每个想要从归档文件中提取的对象）的对象即可。

同样，由于 `AddressCard` 类直接继承自 `NSObject`，所以不必担心解码继承的实例变量。如果的确担心，那么应在解码方法的开始插入下列语句（假设类的超类符合 `NSCoding` 协议的要求）：

```
self = [super initWithCoder: decoder];
```

通过调用 `decodeObject:ForKey:` 方法并传递在编码变量时使用的相同键，就可解码每个实例变量。

与 `AddressCard` 类类似，为 `AddressBook` 类添加了编码和解码方法。在接口文件中只需更改 `@interface` 指令，以声明现在 `AddressBook` 类已经符合 `NSCoding` 协议。更改如下：

```
@interface AddressBook: NSObject <NSCoding, NSCopying>
```

下面是实现文件中所含的方法定义：

```
-(void) encodeWithCoder: (NSCoder *) encoder
{
    [encoder encodeObject: bookName forKey: @"AddressBookBookName"];
    [encoder encodeObject: book forKey: @"AddressBookBook"];
}

-(id) initWithCoder: (NSCoder *) decoder
{
    bookName = [decoder decodeObjectForKey: @"AddressBookBookName"];
    book = [decoder decodeObjectForKey: @"AddressBookBook"];

    return self;
}
```

代码清单 19-6 是它的测试程序。

代码清单 19-6 测试程序

```
#import "AddressBook.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSString *aName = @"Julia Kochan";
        NSString *aEmail = @"jewls337@axlc.com";
        NSString *bName = @"Tony Iannino";
        NSString *bEmail = @"tony.iannino@techfitness.com";
        NSString *cName = @"Stephen Kochan";
        NSString *cEmail = @"steve@steve_kochan.com";
        NSString *dName = @"Jamie Baker";
        NSString *dEmail = @"jbaker@hitmail.com";

        AddressCard *card1 = [[AddressCard alloc] init];
        AddressCard *card2 = [[AddressCard alloc] init];
        AddressCard *card3 = [[AddressCard alloc] init];
        AddressCard *card4 = [[AddressCard alloc] init];

        AddressBook *myBook = [AddressBook alloc];

        // 首先设置 4 个地址卡片

        [card1 setName: aName andEmail: aEmail];
        [card2 setName: bName andEmail: bEmail];
        [card3 setName: cName andEmail: cEmail];
        [card4 setName: dName andEmail: dEmail];

        myBook = [myBook initWithName: @"Steve's Address Book"];

        // 添加一些卡片到通讯簿

        [myBook addCard: card1];
        [myBook addCard: card2];
        [myBook addCard: card3];
        [myBook addCard: card4];

        [myBook sort];

        if ([NSKeyedArchiver archiveRootObject: myBook toFile:
            @"addrbook.arch"] == NO)
            NSLog(@"archiving failed");
    }
    return 0;
}
```

}

这个程序创建了一个地址簿，然后将它归档到文件 `addrbook.arch` 中。在创建归档文件的过程中，注意 `AddressBook` 类和 `AddressCard` 类中的编码方法都被调用了。如果想要验证，可以向这些方法添加一些 `NSLog` 调用。

代码清单 19-7 展示了如何将归档读入内存以根据文件创建地址簿。

代码清单 19-7

```
#import "AddressBook.h"

int main (int argc, char *argv[])
{
    AddressBook      *myBook;
    @autoreleasepool {
        myBook = [NSKeyedUnarchiver unarchiveObjectWithFile: @"addrbook.arch"];

        [myBook list];
    }
    return 0;
}
```

代码清单 19-7 输出

```
===== Contents of: Steve's Address Book =====
Jamie Baker      jbakera@hitmail.com
Julia Kochan     jewels337@axlc.com
Stephen Kochan   steve@steve_kochan.com
Tony Iannino     tony.iannino@techfitness.com
=====
```

在解码地址簿的过程中，自动调用向两个类添加的解码方法。注意，将地址簿读回程序是非常容易的。

前面说过，`encodeObject:forKey:` 方法作用于内置类及根据 `NSCoding` 协议为其编写编码和解码方法的类。如果你的实例包含基本数据类型，如整型或浮点型，那么需要知道如何对它们进行编码和解码（参见表 19.1）。

下面是一个类的简单定义，这个类名为 `Foo`，它包含 3 个实例变量：一个是 `NSString` 类型，一个 `int` 型，一个 `float` 型。这个类包含一个赋值方法、3 个取值方法及两个用于归档的编码/解码方法：

```
@interface Foo: NSObject <NSCoding>
```

```

@property (copy, nonatomic) NSString *strVal;
@property int intVal;
@property float floatVal;
@end

```

实现文件如下:

```

@implementation Foo

@synthesize strVal, intVal, floatVal;

-(void) encodeWithCoder: (NSCoder *) encoder
{
    [encoder encodeObject: strVal forKey: @"FoostrVal"];
    [encoder encodeInt: intVal forKey: @"FoointVal"];
    [encoder encodeFloat: floatVal forKey: @"FoofloatVal"];
}

-(id) initWithCoder: (NSCoder *) decoder
{
    strVal = [decoder decodeObjectForKey: @"FoostrVal"];
    intVal = [decoder decodeIntForKey: @"FoointVal"];
    floatVal = [decoder decodeFloatForKey: @"FoofloatVal"];

    return self;
}
@end

```

编码函数使用前面用过的 `encodeObject:forKey:` 方法来编码字符串值 `strVal`, 如上面内容所示。

在代码清单 19-8 中, 我们创建了一个 `Foo` 对象, 把它归档到一个文件, 并解归档, 然后显示。

代码清单 19-8 测试程序

```

#import <Foundation/Foundation.h>
#import "Foo.h" //定义 Foo 类

int main (int argc, char *argv[])
{
    @autoreleasepool {
        Foo *myFoo1 = [[Foo alloc] init];
        Foo *myFoo2;

        myFoo1.strVal = @"This is the string";
        myFoo1.intVal = 12345;
        myFoo1.floatVal = 98.6;
    }
}

```

```

[NSKeyedArchiver archiveRootObject: myFoo1 toFile: @"foo.arch"];

myFoo2 = [NSKeyedUnarchiver unarchiveObjectWithFile: @"foo.arch"];
NSLog(@"%@\\n%i\\n%g", myFoo2.strVal, myFoo2.intVal, myFoo2.floatVal);

}
return 0;
}

```

代码清单 19-8 输出

```

This is the string
12345
98.6

```

以下消息归档了对象的 3 个实例变量：

```

[encoder encodeObject: strVal forKey: @"FoostrVal"];
[encoder encodeInt: intVal forKey: @"FoointVal"];
[encoder encodeFloat: floatVal forKey: @"FoofloatVal"];

```

一些基本数据类型，如 `char`、`short`、`long` 和 `long long` 在表 19.1 中没有列出。你必须确定数据对象的大小并使用相应的函数。例如，`short int` 通常是 16 位的，而 `int` 和 `long` 可以是 32 位或 64 位的，`long long` 是 64 位的（可以使用第 13 章介绍的 `sizeof` 运算符确定任何数据类型的大小）。所以要归档 `short int` 的数据，首先将其存储在 `int` 中，然后使用 `encodeIntForKey:forKey:` 归档它。反向执行该过程可恢复它；使用 `decodeIntForKey:`，然后将其赋值给 `short int` 变量。

19.4 使用 NSData 创建自定义档案

有时可能不想和前面的示例程序一样，使用 `archiveRootObject:ToFile:` 方法将对象直接写入文件。比如，可能想收集一些或所有的对象，并将其存储到单个档案文件中。在 Objective-C 中，通过使用名为 `NSData` 的通用数据流对象类，可以实现上述功能，在第 16 章，我们简单地提到过这个类。

正如第 16 章所提到的，`NSData` 对象可以用来保留一块内存空间，以备将来存储数据。这些数据空间的典型应用是为一些数据提供临时存储空间，以便随后写入文件，或存放从磁盘读取的文件内容。创建可变数据空间的最简单方式是使用 `data` 方法：

```
dataArea = [NSMutableData data];
```

该语句创建一个空缓冲区，其大小将随着程序执行的需要而扩展。

这里举一个简单的例子，假设你想将地址簿和一个 Foo 对象归档到同一个文件，并且你已经向 AddressBook 和 AddressCard 类添加了一个带键的归档方法（参见代码清单 19-9）。

代码清单 19-9

```
#import "AddressBook.h"
#import "Foo.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        Foo          *myFool = [[Foo alloc] init];
        NSMutableData *dataArea;
        NSKeyedArchiver *archiver;
        AddressBook    *myBook;

        // 插入代码清单 19-7 中的代码创建一个地址簿
        // 在 mybook 中包含 4 个地址卡

        myFool.strVal = @"This is the string";
        myFool.intVal = 12345;
        myFool.floatVal = 98.6;

        //设置数据区，并将其连接到一个 NSKeyedArchiver 对象
        dataArea = [NSMutableData data];

        archiver = [[NSKeyedArchiver alloc]
                    initWithWritingWithMutableData: dataArea];
        //现在可以开始存档对象
        [archiver encodeObject: myBook forKey: @"myaddrbook"];
        [archiver encodeObject: myFool forKey: @"myfool"];
        [archiver finishEncoding];

        //将存档的数据区写到文件
        if ([dataArea writeToFile: @"myArchive" atomically: YES] == NO)
            NSLog(@"Archiving failed!");
    }
    return 0;
}
```

分配一个 `NSKeyedArchiver` 对象之后，发送 `initWithWritingWithMutableData:` 消息，以指定要写入归档数据的存储空间。这就是前面创建的 `NSMutabledata` 空间 `dataArea`。此时，就可以向存储在 `archiver` 中的 `NSKeyedArchiver` 对象发送编码消息，以归档该程序中的对象。实际上，所有的编码消息在收到 `finishEncoding` 消息之前都被归档并存储在指定的数据空间内。

这里，有两个对象需要编码：一个是地址簿，另一个是 `Foo` 对象。对于这些对象，可以使用 `encodeObject:forKey:`，因为在前面你已经为 `AddressBook`、`AddressCard` 和 `Foo` 类实现了编码方法和解码方法（理解这个概念很重要）。

在归档这两个对象时，向 `archiver` 对象发送一条 `finishEncoding` 消息。之后，就不能编码其他对象，此时你需要发送此消息以完成归档过程。

现在，你预留的那块名为 `dataArea` 的空间包含归档对象，这些对象能够以一种可写入文件的格式存在。消息表达式

```
[dataArea writeToFile: @"myArchive" atomically: YES]
```

向你的数据流发送 `writeToFile:atomically:` 消息，请求它把它的数写入指定的文件，这个文件名为 `myArchive`。

从 `if` 语句可以看到，`writeToFile:atomically:` 方法返回一个 `BOOL` 值：如果写操作成功，就返回 `YES`；如果失败（可能是指定了无效的路径名或文件系统已满），就返回 `NO`。

从档案文件中恢复数据很简单：所做的工作只需和归档文件相反。首先，需要像以前那样分配一个数据空间。其次，把档案文件中的数据读入该数据空间。然后，需要创建一个 `NSKeyedUnarchiver` 对象，并告知它从指定的空间解码数据。必须调用解码方法来提取和解码归档的对象，做完之后，向 `NSKeyedUnarchiver` 对象发送一条 `finishDecoding` 消息。

代码清单 19-10 实现了所有的任务。

代码清单 19-10

```
#import "AddressBook.h"
#import "Foo.h"

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSData      *dataArea;
```

```

NSKeyedUnarchiver *unarchiver;
Foo *myFool;
AddressBook *myBook;
// 从归档文件中读取并连接
// NSKeyedUnarchiver 对象

dataArea = [NSData dataWithContentsOfFile: @"myArchive"];

if (! dataArea) {
    NSLog (@"Can't read back archive file!");
    return 1;
}

unarchiver = [[NSKeyedUnarchiver alloc]
               initWithReadingWithData: dataArea];

// 解码以前存储在归档文件中的对象
myBook = [unarchiver decodeObjectForKey: @"myaddrbook"];
myFool = [unarchiver decodeObjectForKey: @"myfool"];

[unarchiver finishDecoding];

// 验证是否还原成功
[myBook list];
NSLog (@"%@\n%i\n%g", myFool.strVal, myFool.intVal, myFool.floatVal);

}
return 0;
}

```

代码清单 19-10 输出

```

===== Contents of: Steve's Address Book =====
Jamie Baker      jbaker@hitmail.com
Julia Kochan     jewls337@axlc.com
Stephen Kochan   steve@steve_kochan.com
Tony Iannino     tony.iannino@techfitness.com
=====
This is the string
12345
98.6

```

输出结果验证了你的地址簿和 Foo 对象已成功地从档案文件中恢复了。

19.5 使用归档程序复制对象

在代码清单 18-2 中，尝试创建了包含可变字符串元素的数组副本，并且了解了如何进行浅复制。也就是说，没有复制实际的字符串本身，只是复制对它们的引用。

可以使用 Foundation 的归档功能来创建对象的深复制。例如，代码清单 19-11 通过 `dataArray` 归档到一个缓冲区，然后把它解归档，将结果指派给 `dataArray2`，从而将 `dataArray` 复制给 `dataArray2`。对于这个过程，不需要使用文件，归档和解归档过程都可以在内存中发生。

代码清单 19-11

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

int main (int argc, char *argv[])
{
    @autoreleasepool {
        NSData      *data;
        NSMutableArray *dataArray = [NSMutableArray arrayWithObjects:
            [NSString stringWithString: @"one"],
            [NSString stringWithString: @"two"],
            [NSString stringWithString: @"three"],
            nil
        ];

        NSMutableArray *dataArray2;
        NSMutableString *mStr;

        // 使用归档器进行深层复制

        data = [NSKeyedArchiver archivedDataWithRootObject: dataArray];
        dataArray2 = [NSKeyedUnarchiver unarchiveObjectWithData: data];

        mStr = dataArray2[0];
        [mStr appendString: @"ONE"];

        NSLog (@@"dataArray: ");
        for ( NSString *elem in dataArray )
            NSLog (@"%@", elem);

        NSLog (@"\ndataArray2: ");
        for ( NSString *elem in dataArray2 )
            NSLog (@"%@", elem);
    }
}
```

```

    }
    return 0;
}

```

代码清单 19-11 输出

```

dataArray:
one
two
three

dataArray2:
oneONE
two
three

```

这个输出结果验证了更改 `dataArray2` 的第一个元素对 `dataArray` 的第一个元素并没有影响，这是因为在归档和解归档过程中产生的是字符串的新副本。

代码清单 19-11 中的复制操作是通过以下两行实现的：

```

data = [NSKeyedArchiver archivedDataWithRootObject: dataArray];
dataArray2 = [NSKeyedUnarchiver unarchiveObjectWithData: data];

```

甚至可以避免中间赋值，只用一条语句来执行复制，语句如下：

```

dataArray2 = [NSKeyedUnarchiver unarchiveObjectWithData:
    [NSKeyedArchiver archivedDataWithRootObject: dataArray]];

```

下次需要生成一个对象（或不支持 `NSCopying` 协议的对象）的深复制时，应该记住这项技术。

19.6 练习

1. 在第 6 章“选择结构”的代码清单 6-10 中生成了一个素数表。修改此程序，将结果数组作为 XML 属性列表写入文件 `primes.pl` 中。然后，检查该文件的内容。
2. 编写一个程序，该程序从练习 1 中创建的 XML 属性列表中读取数据，并将它们的值存储到一个数组对象。打印输出这个数组的所有元素，以验证存储操作成功。
3. 修改代码清单 19-2，显示/Library/ Preferences 文件夹中存储的某个 XML 属性列表（.plist 文件）的内容。

4. 写一个程序，使它读取已归档的 `AddressBook`，并根据在命令行提供的名称进行查找，如：

```
$ lookup gregory
```

Cocoa 和 Cocoa Touch 简介

本书中开发的所有程序都具有简单的用户界面，依据 `NSLog` 函数以简单的文字形式输出。这个函数虽然有用，但功能十分有限。毫无疑问，Mac 上或 iPhone 中使用的其他程序都是用户友好的。事实上，Mac 的良好声誉正是建立在用户友好的对话框和易用性上。幸运的是，Xcode 和内置的用户界面设计工具组合在一起可以达到要求。这种组合不仅提供了一个包含编辑和调试工具的强大程序开发环境，可以方便地访问在线文档，还提供了一个可以轻松开发复杂图形用户界面（GUI）的环境。

Cocoa 是一种为 Mac OS X 应用程序提供了丰富用户体验的框架，实际上由 3 个框架组成：已经熟知的 Foundation 框架，便于使用数据库存储和管理数据的 Core Data 框架，以及 Application Kit（AppKit）框架。AppKit 框架提供了与窗口、按钮、列表等相关的类。

20.1 框架层

使用示意图来说明最顶层应用程序与底层硬件之间的各个层次，如图 20.1 所示。

内核以设备驱动程序的形式提供与硬件的底层通信。它负责管理系统资源，包括调度需要执行的程序、管理内存和电源，以及执行基本的 I/O 操作。

顾名思义，核心服务提供的支持比它上面层次更加底层或更加“核心”。例如，提供对集合、网络、调试、文件管理、文件夹、内存管理、线程、时间和电源的管理。

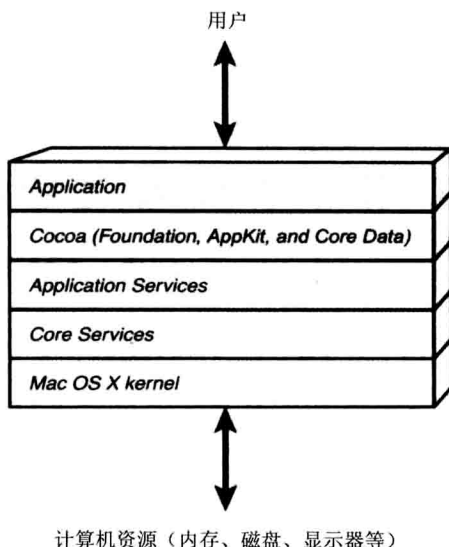


图 20.1 应用程序层次结构

应用服务层包含打印和图形渲染的支持，包括 Quartz、OpenGL 和 Quicktime。

Cocoa 层位于应用程序层之下。如图 20.1 所示，Cocoa 包括 Foundation、Core Data 和 AppKit 框架。Foundation 框架提供处理集合、字符串、内存管理、文件系统、存档等相关的类。AppKit 框架提供管理视图、窗口、文档和使 Mac OS X 闻名于世的多用户界面相关的类。

根据上面的描述，有些层的功能似乎有重复。Cocoa 层和核心服务层中都存在集合。然而，后者是前者的基础。此外，某些情形也可以绕过或者“桥接”到某一层。例如，Foundation 中有些类，比如处理文件系统的那些类直接依赖核心服务层的功能，实际上绕过了应用程序服务层。很多情况下，Foundation 框架作为底层核心服务层（优先采用过程化的 C 语言编写），定义的数据结构为一种面向对象的映射。

20.2 Cocoa Touch

iOS 设备如 iPhone、iPod touch 和 iPad 包含一台运行 Mac OS X 精简版本的计算机。iPhone 硬件中的有些功能（比如它的加速器）是电话中独一无二的，

而且在其他 Mac OS X 计算机（如 MacBook Pro 或 iMac）中也找不到。

注意

Mac 笔记本电脑实际上包含一个加速器，用于当计算机跌落到地上时对硬盘进行保护，然而，不能直接从程序中访问这个加速器。

Cocoa 框架应用于 Mac OS X 桌面与笔记本电脑应用程序的开发，而 Cocoa Touch 框架应用于 iOS 设备上应用程序的开发。

Cocoa 和 Cocoa Touch 都有 Foundation 和 Core Data 框架。然而在 Cocoa Touch 下，UIKit 代替了 AppKit 框架，提供了很多相同类型对象的支持，比如窗口、视图、按钮、文本域等。另外，Cocoa Touch 还提供使用陀螺仪和加速器（它与 GPS 和 WiFi 信号一样都能跟踪你的位置）的类和触摸式界面，去掉了不需要的类。

对 Cocoa 和 Cocoa Touch 的概要介绍就到此结束。在下一章中，你将了解如何使用 iOS 软件开发工具包（SDK）包含的模拟器为 iPhone 编写应用程序。

编写 iOS 应用程序

本章将介绍两个简单的 iPhone 应用程序。第一个应用程序说明一些基础性概念，目的是让你熟悉如何使用 Xcode 内置的界面设计工具（Interface Builder）建立连接，并理解委托（delegate）、出口（outlet）和操作（action）。第二个应用程序是一个分数计算器，它很好地结合了你在开发第一个应用程序的过程中所学的知识和本书其他部分学到的知识。这里学到的原则也适用于在其他 iOS 设备上开发应用程序。

21.1 iOS SDK

要编写 iPhone 应用程序，必须先安装 Xcode 和 iOS 软件开发工具包（SDK）。这个 SDK 可以从 Apple 的 Web 站点上免费下载。下载之前，你首先需要注册成为 Apple 开发人员，这个过程同样也是免费的。为了找到正确的链接，你可以从 developer.apple.com 开始访问，然后导航到正确的地址。熟悉这个站点是明智的选择。

本章中的讨论基于 Xcode 5 和 iOS SDK for iOS 7。它们的新版本与我们描述的内容也是兼容的。如果你注意到屏幕显示的与这里不同，或许你使用了不同版本的 Xcode。在这种情况下，可以查看 classroomM.com/objective-c 论坛获取当前最新的信息。

21.2 第一个 iPhone 应用程序

第一个应用程序说明了如何在 iPhone 屏幕上放置一个黑色窗口，只要用户按下一个按钮，窗口中就会显示一些文本。

注意

第二个应用程序更有趣！你需要使用从第一个应用程序中学到的知识构建一个进行分数运算的简单计算器。你可以使用本书中出现过的 `Fraction` 类，以及经过修改的 `Calculator` 类。这次，你的计算器需要知道如何处理分数。

让我们开始第一个程序。因篇幅有限，本章并不会面面俱到。但我们会依次讲解每个步骤，让你拥有必要的基础知识，以便通过单独的 Cocoa 或 iOS 程序设计教材学习更多的概念。

图 21.1 显示开发的第一个 iPhone 应用程序，它运行在 iPhone 模拟器（很快就会详细介绍）上。

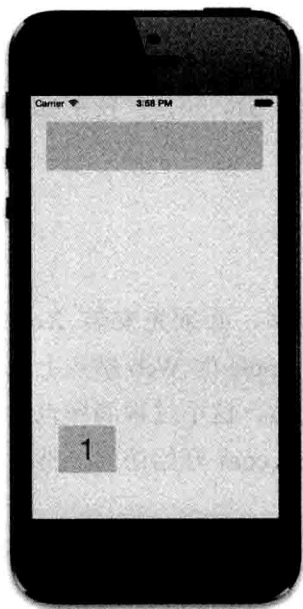


图 21.1 第一个 iPhone 应用程序

这个应用程序的设计目的是：按下标记为“1”的按钮时，显示屏上就会出现相应的数字（见图 21.2）。这就是它的全部功能！这个简单的应用程序为第二个分数计算器应用程序打下了基础。

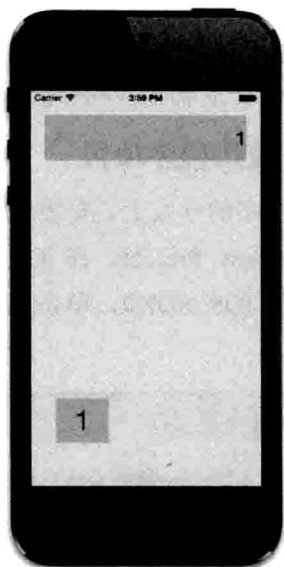


图 21.2 iPhone 应用程序的运行结果

到目前为止，如果你一直使用 Xcode 输入和测试程序，对它应该很熟悉了。如前面所述，通过界面构造器这个工具将各种 UI 元素（如表、标签和按钮）放入 iPhone 屏幕的窗口中，设计出用户界面。与任何其他强大的开发工具一样，这个功能确实需要花一些时间来熟悉。

注意

在 Xcode 4 之前的版本，被称为界面构造器的 UI 设计工具是单独的应用程序。

Apple 在 iOS SDK 中发布了一个 iPhone 模拟器。该模拟器复制了大部分的 iPhone 环境，包括它的主屏幕、Safari Web 浏览器、Contacts 应用程序等。模拟器使调试应用程序变得更加轻松，进行调试时，不必每次都将应用程序下载到真正的 iPhone 设备上。这可以节省大量的时间和工作量。

为了在 iOS 设备上运行应用程序，需要注册 iOS 开发人员计划，并付给 Apple 99 美元（此数字截止到本书撰写之际）的费用。然后，你将会收到一个激活码，它让你能够获取一个 iOS 开发证书，从而可以在 iOS 设备上测试并安装应用程序。不幸的是，如果没有完成这个过程，即使是在你自己的 iOS 设备

上，也无法开发应用程序。注意，本章中，我们开发的应用程序将在 iPhone 模拟器上进行加载和测试，而不是在 iPhone 设备上。

21.2.1 创建新的 iPhone 应用程序项目

让我们回到第一个应用程序的开发上。安装 iOS SDK 后，启动 Xcode 应用程序。选择菜单 File→New→New Project，在 iOS 下（如果在左面板中看不到这个区域，证明你还没有安装 iOS SDK），单击 Application，此时应该看到一个如图 21.3 所示的窗口。

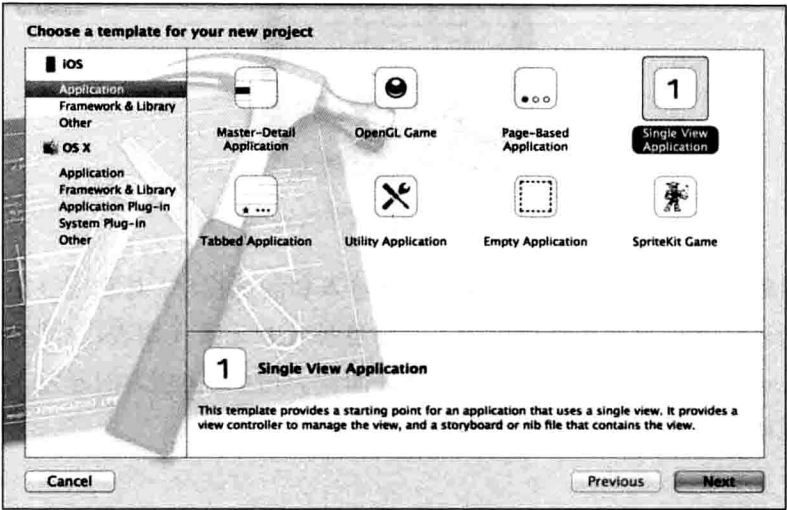


图 21.3 创建新的 iOS 项目

图 21.3 窗口中显示了为不同类型的应用程序提供的各种模板，表 21.1 对此进行了总结。

表 21.1 iOS 应用程序模板

应用程序类型	说 明
Master-Detail	针对使用导航控制器的应用程序。Contacts 应用程序就是这类模板的一个例子。为大尺寸屏幕的设备如 iPad 使用拆分视图（split view）
OpenGL Game	针对基于 OpenGL ES 图形的应用程序，比如游戏
SpriteKit Game	使用 SpriteKit 的应用程序，为开发游戏程序提供了良好的起点
Page-based Application	针对使用页面视图控制器的程序，用于管理页面的显示
Single View Application	针对拥有单一视图的应用程序。绘制视图后便将它显示在窗口中

续表

应用程序类型	说 明
Tabbed Application	针对使用选项卡栏的应用程序。音乐应用程序就是一个例子
Utility Application	针对拥有反面视图的应用程序。Stock Quote 应用程序就是这类模板的一个例子
Empty Application	针对只从 iPhone 主窗口开始的应用程序。可以使用此模板作为任意应用程序的起点

回到 New Project 窗口中，选择位于右上角的 Single View Application，然后单击 Next 按钮。为项目名称输入文本 iPhone_1，填写组织名称和企业标识，填写类前缀为 CM，选择设备系列为 iPhone，如图 21.4 所示。

注意

你可能希望遵循典型的风格，指定 com.company 作为企业标识。也就是颠倒域名，在你的应用程序中也作为包标识符（bundle identifier）形式的基础。

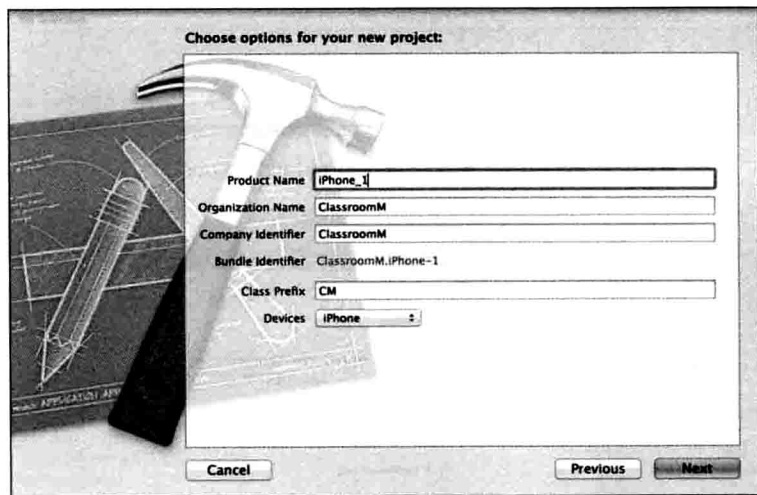


图 21.4 设置项目选项

单击 Next 按钮，配置项目文件夹存储的位置，不必关心 Source Control 复选项是否选中，如图 21.5 所示。

然后单击 Create 按钮。和以前一样创建一个 Xcode 的新项目，创建的新项目中包含你需要使用的模板文件，如图 21.6 所示。iPhone_1 文件夹有 5 个文件

CMAppDelegate.h、AppDelegate.m、CMViewController.h、CMViewController.m 和 CMViewController.xib。每个自定义类均有 CM 前缀。大多数公司会选择他们自己的字符（一般两个）作为类名的前缀。Apple 使用 NS 作为类名前缀，比如 NSArray 和 NSString。

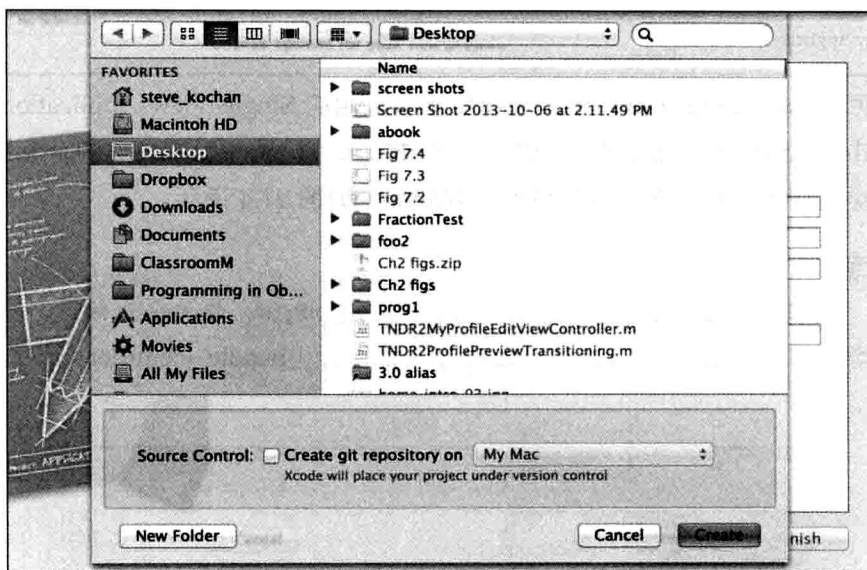


图 21.5 配置项目文件夹存储的位置

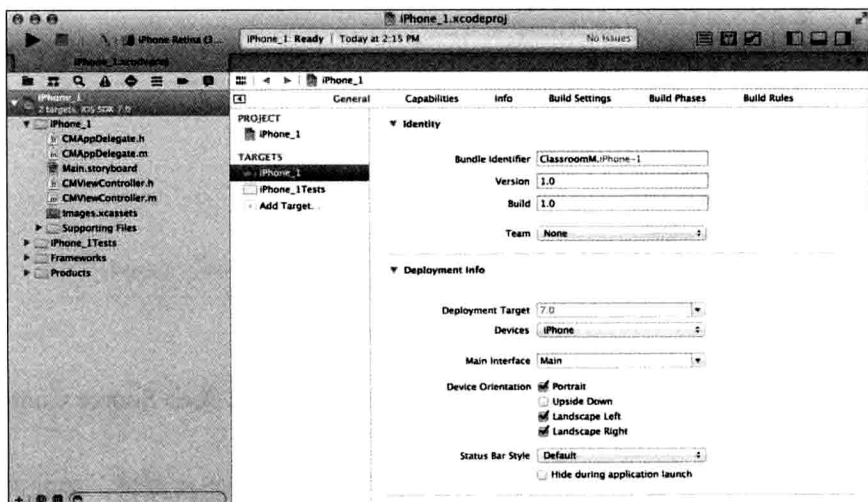


图 21.6 已创建新的 iOS 项目 iPhone_1

根据你的设置和前面对 Xcode 的使用情况，你的窗口外观可能不会与图 21.6 中显示的完全一样。你可以选择保持当前的布局组成，或者尝试让它与图中的窗口更相似。在 Xcode 窗口的左上角（状态窗口的前面），这里有一个下拉列表用于选择在哪里运行你的应用程序。因为我们开发的并非是在 iPhone 上直接运行的应用程序，因此，需要将 SDK 设置为运行 iPhone 模拟器。选择下拉列表中的 iPhone Retina (4-inch 64-bit) 模拟器，请按照图 21.7 所示设置正确的选项。

如果你希望在小尺寸屏幕上测试你的应用程序，比如 3.5 英寸屏幕的 iPhone 4，可以在这里进行选择。这个练习中，我们针对的是 4 英寸的屏幕。

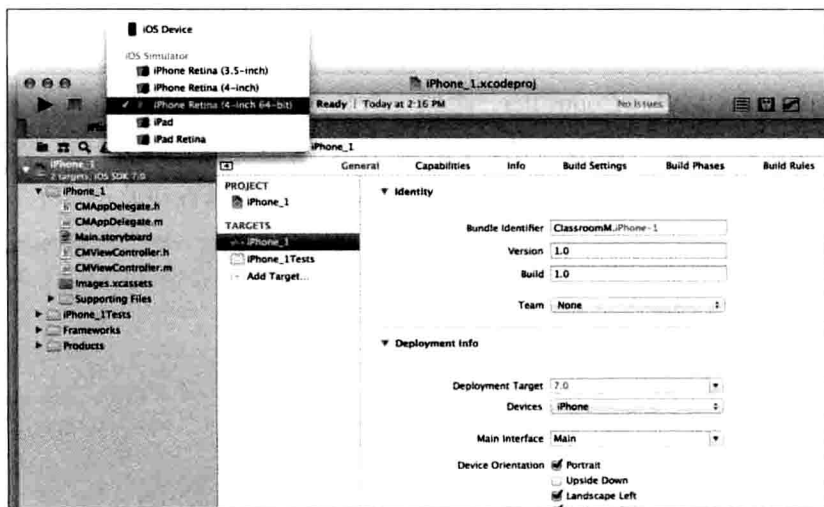


图 21.7 选择 iPhone 模拟器

21.2.2 输入代码

现在，我们可以修改一些项目文件。注意，已经创建了名为 `CAppDelegate` 的类。每一个新的 iOS 应用程序被创建出来，相应的应用代理（delegate）也同时被创建。一般来说，这个类控制着应用的执行，包括的事情如启动应用、进入应用、离开应用做些什么事情，或者在应用使用了过多的内存时进行通知，或者应用终止的时候进行通知。本章中的两个例子并没有对这个类做修改。

创建的第二个类称为 `CMViewController`。Xcode 会为你分别在 .h 和 .m 中创建相应的接口和实现文件。一个视图控制器负责管理一个或者多个视图的显示。

在这个应用中仅使用一个视图，如图 21.1 所示。应用中显示多个不同的视图并不常见，一般使用多个视图控制器对多个视图进行管理。

所以，我们需要视图控制器响应按下标记“1”按钮的事件。在这个类中，定义方法响应 iPhone 窗口中出现的操作，比如按下按钮。你可以看到，如何将这些事件和响应方法联系在一起。

这个对象还有一些实例变量，它们的值对应于 iPhone 窗口中的一些控件，比如，标签的名称或可编辑文本框内显示的文本。这些变量称为出口（*outlet*），你可以看到，如何将实例变量与 iPhone 窗口中实际的控件进行连接。

对于第一个应用程序，我们需要一个方法来响应按下标记“1”按钮的操作。我们还需要一个出口变量，包含我们在 iPhone 窗口顶部标签显示的文本和其他信息。

编辑文件 `CMViewController`，添加一个名为 `display` 的新 `UILabel` 变量，并声明一个名为 `click1` 的操作方法，以便响应按下按钮的操作。接口文件的内容应该如代码清单 21-1 所示（在文件头部自动插入的注释行在此不显示）。

代码清单 21-1 `CMViewControlleriPhone.h`

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

@interface CMViewControlleriPhone : UIViewController

@property (strong, nonatomic) IBOutlet UILabel *display;

-(IBAction) click1;

@end
```

注意到 iPhone 应用程序导入了头文件 `<UIKit/UIKit.h>`。这个头文件又导入了其他 `UIKit` 头文件，导入方式与 `Foundation.h` 头文件导入其他所需头文件（如 `NSString.h` 和 `NSObject.h`）类似。

添加的另一个名为 `display` 的属性属于 `UILabel` 类。这是一个出口属性，将它连接到一个标签。当设置这个属性的文本字段时，会更新窗口中标签的对应文本。通过 `UILabel` 类的其他方法可以设置和获取标签的多种属性，比如它的颜色、行数和文本大小。

如果你学习了这里没有介绍到的 iOS 编程内容，可能会在界面中使用一些

类。类的名称就已经表明了它的用途，如 UITextField、UIFont、UIView、UITableView、UIImageView、UIImage 和 UIButton。

display 属性是出口，在属性的声明中，需要注意 IBOutlet 标识符的使用。实际上，在 UIKit 头文件 UINibDeclarations.h 中，IBOutlet 中的#define 没有任何意义（也就是说，预处理器可以在源文件中将它逐一替换成空白）。然而，Xcode 在读取头文件时，只有定义为 IBOutlet 的变量，才可以用作为出口并连接到界面上的 UI 元素。

将 click1 方法的返回值类型定义为 IBAction（UINibDeclarations.h 头文件中定义为 void）。IBOutlet 一样，在读取头文件时，Xcode 将使用这个标识符的方法识别为操作。可以在用户界面编辑器中修改触发控件的响应事件。

现在修改相应类的实现文件 CMViewController.m。为 display 属性编写存取方法。

编辑实现文件并添加@synthesize 指令和方法 click1 的定义，见代码清单 21-1 中的一些语句（注意到这里缺少一些 Xcode 放入实现文件的一些方法，我们并不对它们进行修改）。

代码清单 21-1 CMViewController.m

```
#import "CMViewController.h"

@interface CMViewController ()

@end

@implementation CMViewController

@synthesize display;

-(IBAction) click1
{
    display.text = @"1";
}

- (void)viewDidLoad {
    [super viewDidLoad];
    // 视图加载以后做一些额外设置，特别是从 nib 中载入的
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}
```



```
// 处理可以重新创建的任何资源
}
```

```
@end
```

请注意 Xcode 会添加类的扩展到你的类里，你可以用来定义私有的方法和属性：

```
@interface ViewController ()
@end
```

私有方法仅可在类的实现部分使用（第 11 章，“分类和协议”会涵盖有关类扩展的内容）。

通过 click1 方法设置 UILabel 的 text 属性将出口变量 display 设为字符串“1”。再将按钮的按下动作连接到这个方法，它能够执行预定的操作，将 1 放入 iPhone 窗口显示。为了进行连接，需要学习如何使用 Xcode 的界面设计工具。

21.2.3 设计界面

在图 21.4 和 Xcode 主窗口中，注意一个名为 Main.storyboard 的文件，包含了与程序的用户界面有关的信息，包括它的窗口、按钮、标签、选项卡栏、文本字段等信息。当然，你现在还没有用户界面，这是接下来要完成的工作。同时它也可以描述，在程序执行时一些操作引发的视图转换（例如，有人单击了标签栏的第一个标签按钮会发生什么）。在过去，iOS 开发者通常使用 XIB 设计他们的界面（过去是 NIB）。仍然可以这样做，但是 storyboard 已经作为 Xcode 默认的文件，比 XIB 文件更加强大。

选择左边面板中的 Main.storyboard 文件，这时出现了界面设计工具，如图 21.8 所示。右边面板显示了 iPhone 的主窗口，显示是空的并以灰色作为默认背景。

注意

这里绘制的窗口是为 iPhone 5 更长的屏幕提供的。如果你编写的应用程序希望支持 iPhone 4（或者早期的型号），需要考虑你的窗口布局，以适应更小屏幕的长度。

从菜单 View 中选择 Utilities，会显示属性检查器（Attributes Inspector）。

图 21.9 是其中一种显示方式。

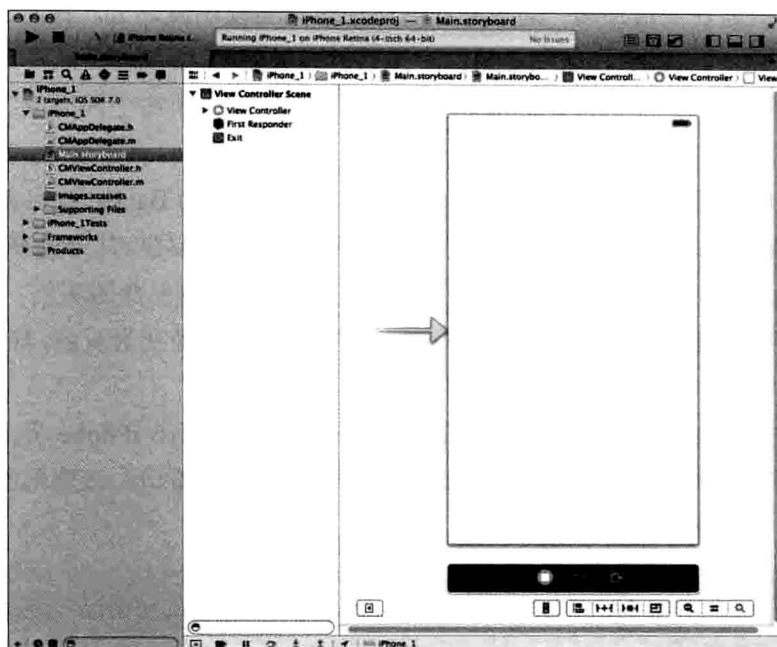


图 21.8 用户界面设计面板

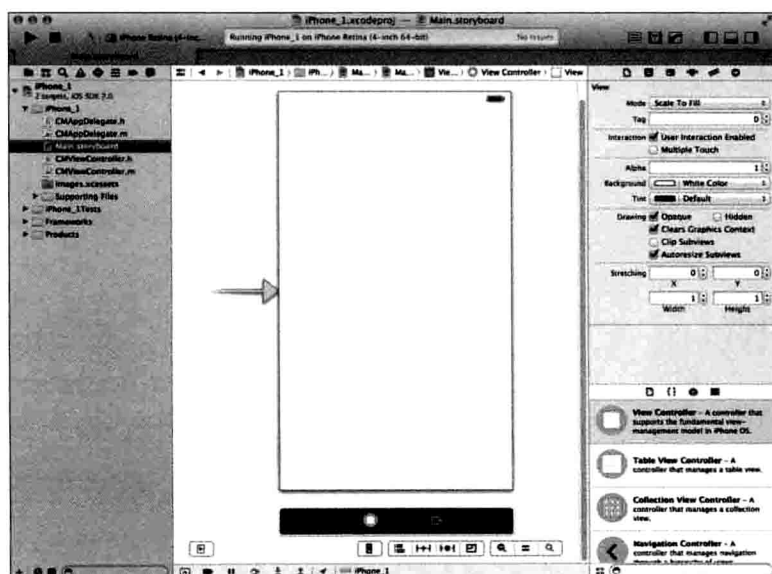


图 21.9 查看 Utilities 窗格

（如果希望和图中显示的一样，可以在编辑器和视图栏中查看各种选项。图中属性检查器在右边顶端，对象库在右边底端图标视图模式下。我们也可以收起“视图控制器场景”，即左边第二个窗格以节省空间。）

首先要做的事情是将 iPhone 的窗口设置为浅灰色。单击中间面板的 iPhone 窗口。

如果向下看到检查器面板的 View 部分，一个标记为 Background 的属性。单击 Background 旁边的白色实心矩形，可以打开颜色拾取器，从中选择浅灰色，这将把 Inspector 窗口中 Background 属性旁边的矩形从白色变为灰色。

如果在中间面板看到表示的 iPhone 窗口，发现它已经变为灰色，如图 21.10 所示。

在对象库面板中拖动一个对象到 iPhone 窗口中，就在 iPhone 界面中创建了新的对象。现在拖动一个标签（Label），当标签接近窗口左边顶部的位置时释放鼠标，如图 21.11 所示。

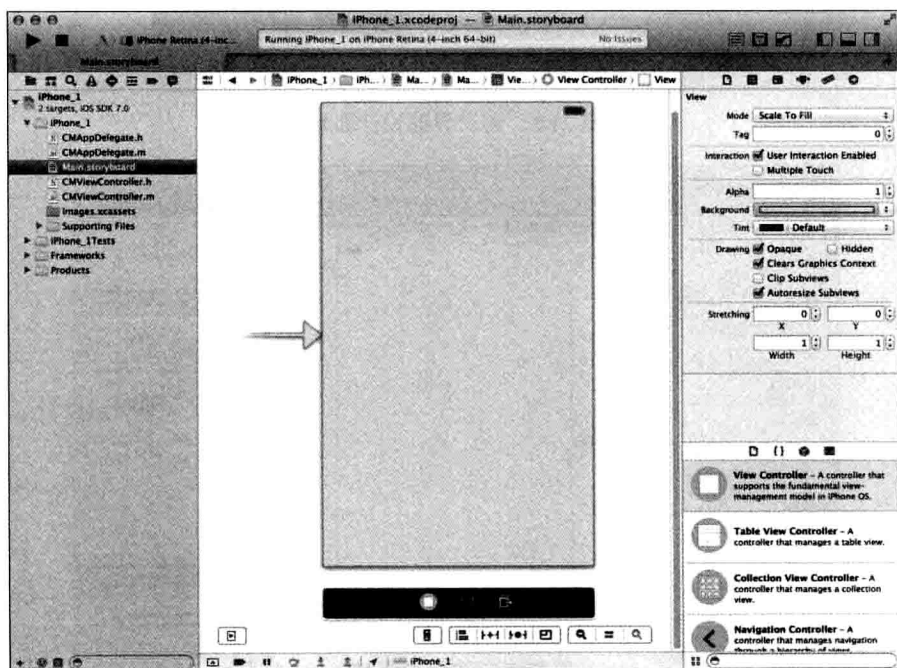


图 21.10 iPhone 窗口变为灰色

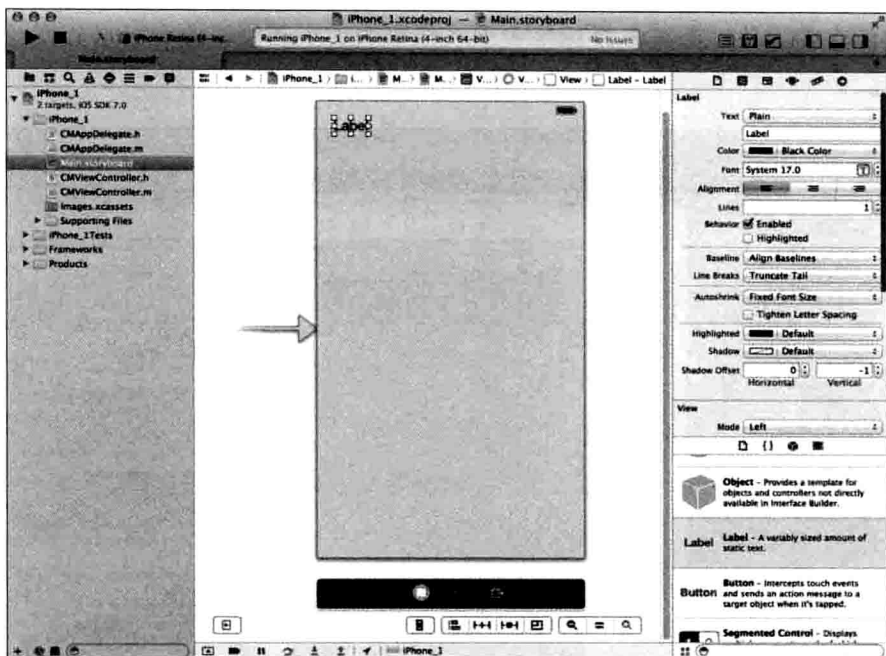


图 21.11 添加一个标签

在窗口内四处移动标签，会出现蓝色的导线。有时候，显示导线是为了将对象与前面在窗口中放置的其他对象对齐，其他时候是为了确保对象与其他对象和窗口边缘之间具有足够的间隔，从而与 Apple 的界面导线保持一致。

以后随时可以改变标签在窗口中的位置，方法是选中并将其拖动到窗口内其他位置。

现在让我们设置这个标签的一些属性。如果 iPhone 窗口中当前没有选中标签，单击你刚刚创建的标签以选中它。注意，Inspector 面板会提供当前选中对象的信息。

就像将窗口的背景颜色改变为浅灰色一样，将标签的背景颜色改为银色（或者你选择的其他颜色）。我们不想让这个标签默认显示任何文本，因此，将 Text 值修改为一个空字符串（也就是说，从 Inspector 面板显示的文本字段中删除字符串 Label）。

对于 Alignment 属性，选择 right-justified（右对齐）。将文本的字体属性设置为 System 28.0。最后，我们来改变标签的大小。返回 Window 窗口，只要向

外拉伸它的边角，即可改变标签大小。标签的大小和位置改变后，如图 21.12 所示。

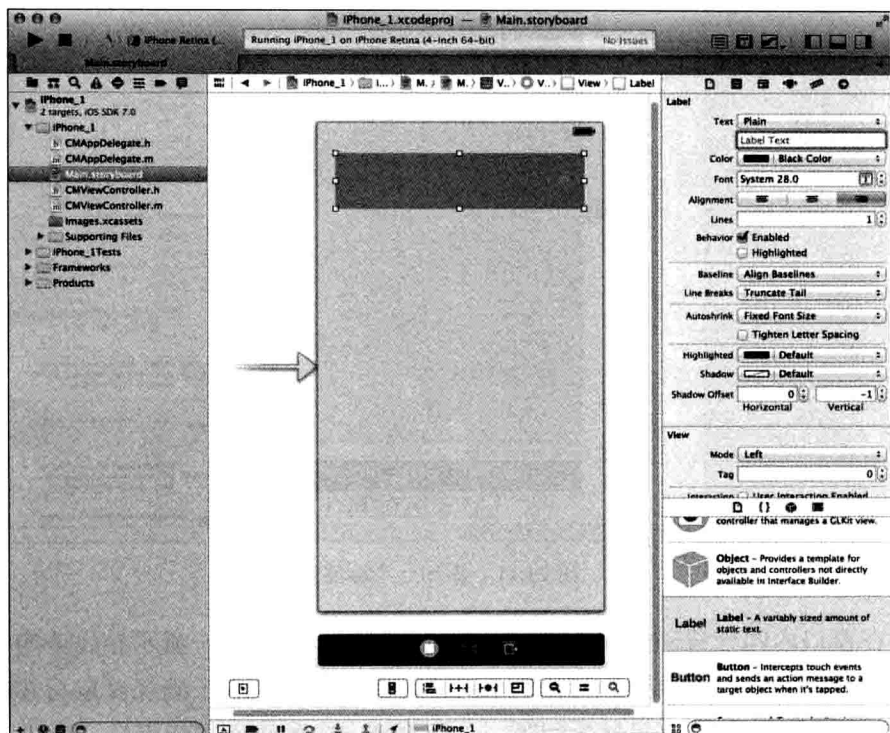


图 21.12 改变标签的属性和尺寸

为界面添加一个按钮。从对象库窗口中选择一个 Rounded Rect Button 对象并将其拖到界面窗口中，把它放置在窗口的左下角，如图 21.13 所示（将按钮的背景颜色设置得和标签相同——这里，设置成银色）。修改按钮上的文本有两种途径：一是通过双击按钮，然后输入文本，二是在 Inspector 面板中第二行设置 Title 字段。任意选择一种途径，让窗口变成如图 21.13 所示的形式。

现在，我们有一个连接到程序中 `display` 实例变量的标签，这样只要设置变量的值，标签文本就会随之发生变化。

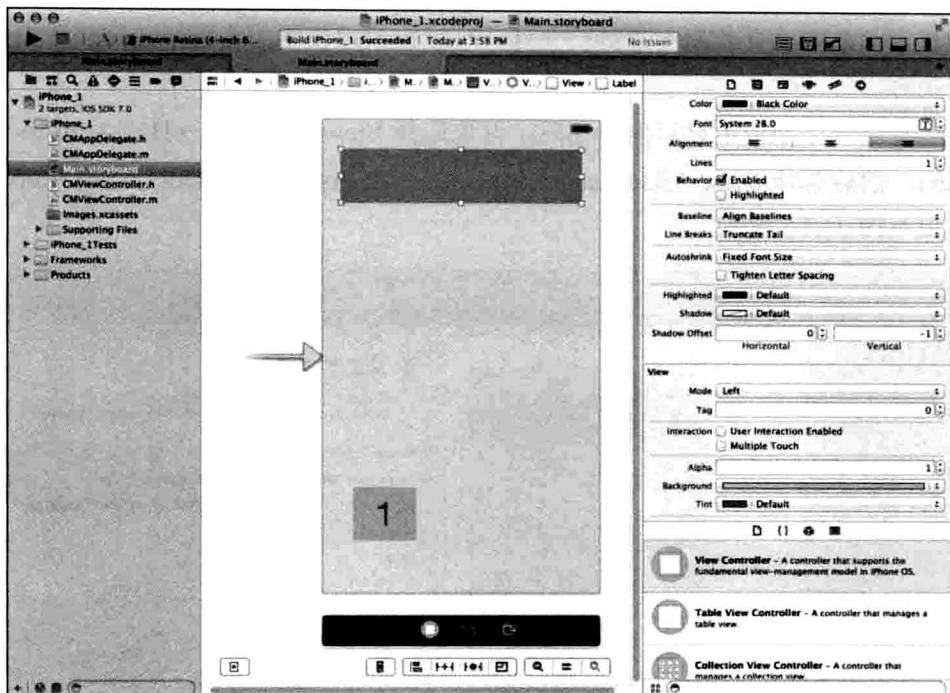


图 21.13 为界面添加一个按钮

我们还有一个标记为 1 的按钮，只要按下它，就会调用我们的 `click1` 方法。该方法将 `display` 的文本字段设置为 1，而且因为该变量连接到标签，所以标签文本也会更新。下面是我们要实现的过程：

- (1) 用户按下标为 1 的按钮。
- (2) 这个事件导致 `click` 方法被执行。
- (3) `click` 方法将实例变量 `display` 的文本属性修改为字符串 1。

(4) 因为 `UILabel` 对象 `display` 连接到了 iPhone 窗口中的标签，所以这个标签也会更新为相应的文本值，也就是值 1。

为了实现上述过程，我们只需要将两者连接起来。有许多种方法可以做到，这里只描述其中一种方法。

首先，让右边窗格显示源代码。为此，单击视图右边工具栏分组的第三个图标将工具面板隐藏起来。接下来，选择工具栏编辑器组中间的图标（一个像燕尾服的图标）打开编辑器。这样，你的 Xcode 窗口应该如图 21.14 所示。如

果显示的是实现文件而不是接口文件，可以在顶部窗格的路径快速导航条中进行选择。

在第三个面板中显示出源代码。首先选择 View→Utilities→Hide Utilities→Next，然后选择 View→Assistant Editor，显示出 Assistant Editor。这时 Xcode 窗口与图 21.14 相似。

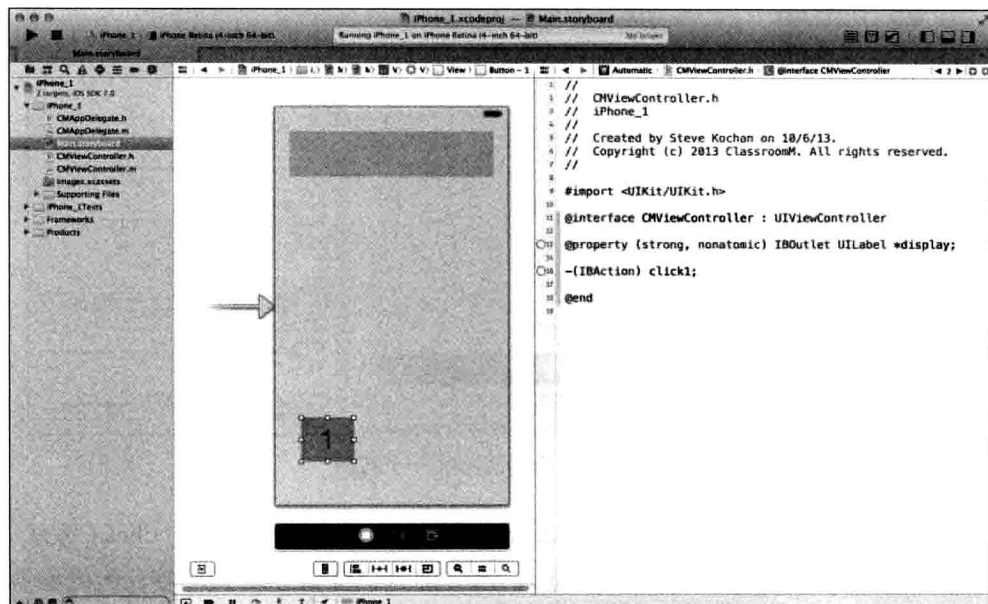


图 21.14 与界面并排显示源代码

将按钮连接到 IBAction 方法 click1，方法是按下 Ctrl 键的同时单击按钮，然后将出现在屏幕上的蓝线拖动到右边面板显示的 click1 方法（既可以拖动方法接口声明的部分，也可以是实现定义的部分），如图 21.15 所示。

现在，将 display 变量连接到标签。选择 iPhone 窗口中的标签，按住 Ctrl 键并单击拖动出现的蓝线到接口文件声明的 display 属性，如图 21.16 所示。

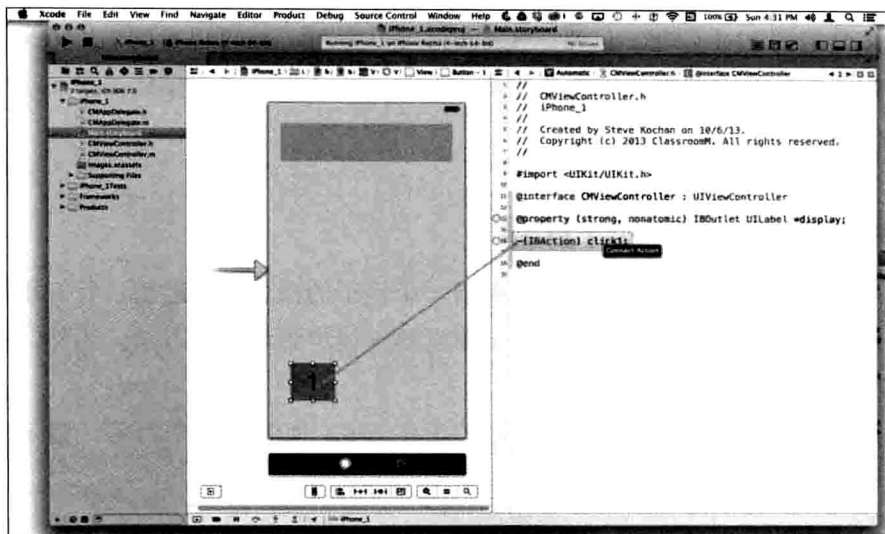


图 21.15 为按钮添加一个操作

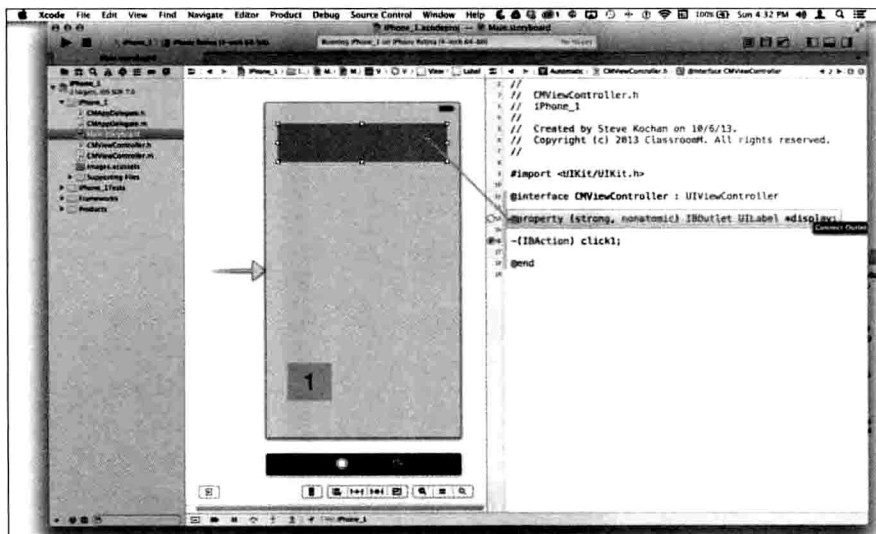


图 21.16 连接一个出口变量

释放鼠标时，连接就已经建立了。

到这里全部过程就结束了！选择 **Product** 菜单中 **Run** 或者工具栏中的 **Play** 按钮。如果一切顺利，程序成功编译并开始执行。开始执行时，程序将被加载到 **iPhone** 模拟器中，模拟器会出现在计算机显示器上。模拟器窗口应该如本章

开始部分的图 21.1 所示。只要在模拟器中单击按钮，就可以看到模拟效果。此时，按照我们讲述的操作步骤和建立的连接，会在显示器顶部标签中显示字符串 1，如图 21.2 所示。

21.3 iPhone 分数计算器

下一个例子要稍微棘手一点，但前一个例子的概念同样适用。我们不准备说明创建这个例子的所有步骤，而是概要地讲述步骤和设计方法。当然，我们也会给出所有的代码。

首先，让我们看看这个应用程序的工作方式。如图 21.17 所示，显示了应用程序启动后在模拟器中的样子。



图 21.17 启动后的分数计算器

计算器应用程序的使用方法是首先输入分子，接着按下标有 Over 的键，然后输入分母。例如，输入分数 $2/5$ 的步骤是，按下 2 数字键，接着按下 Over 键，然后按下 5 数字键。可以看到，和其他计算器不同，这个计算器会在显示器上

原样显示分数，因此， $2/5$ 就显示为 $2/5$ 。

输入一个分数后，接下来可以选择运算——加法、减法、乘法或除法，只要分别按下标有 $+$ 、 $-$ 、 $\times$ 或 $\div$ 的键即可。

输入第二个分数后，按下 $=$ 键，即可完成运算，这一点与标准计算器相同。

注意

这个计算器的设计方式是在两个分数之间进行一次运算。本章末尾有一个练习，让你来消除这个限制。

按下键时显示会不断更新。图 21.18 显示了输入分数 $4/6$ 并按下乘法键后的结果。

图 21.19 显示了分数 $4/6$ 和 $2/8$ 相乘后的结果。你会注意到结果为 $1/6$ ，表明结果已经简化了。



图 21.18 一次运算中的输入



图 21.19 两个分数相乘的结果

21.3.1 启动新的 Fraction_Calculator 项目

第二个程序示例将从创建一个新项目开始。从 New Project 窗口中选择

Single View Application，给新项目起名为 Fraction_Calculator 并使用 FC 作为类名的前缀。

创建项目时，这次可以注意到已经定义好了两个类模板：FCAppDelegate.h 和 FCAppDelegate.m 定义了项目的应用程序代理类，而 FCViewController.h 和 FCViewController.m 定义了项目的视图控制器类。与第一个例子一样，所有的工作是在后一个类中进行的。

21.3.2 定义视图控制器

现在让我们编写视图控制器类 FCViewController 的代码，我们将从接口文件开始，如代码清单 21-2 所示。

代码清单 21-2 FCViewController.h 接口文件

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

@interface FCViewController : UIViewController

@property (strong, nonatomic) IBOutlet UILabel *display;

-(void) processDigit: (int) digit;
-(void) processOp: (char) theOp;
-(void) storeFracPart;

// 数字键

-(IBAction) clickDigit: (UIButton *) sender

// 算术操作键

-(IBAction) clickPlus;
-(IBAction) clickMinus;
-(IBAction) clickMultiply;
-(IBAction) clickDivide;

// Misc 键

-(IBAction) clickOver;
-(IBAction) clickEquals;
-(IBAction) clickClear;

@end
```

有一些内部变量分别用于构造分数（currentNumber、firstOperand 和 isNumerator）和构造显示字符串（displayString）。还有一个 Calculator 对象（myCalculator）用于执行两个分数间的实际计算。我们把一个名为 clickDigit: 的方法关联到处理按下数字键 0~9 的操作上。按住数字键会短暂出现方法参数的提示。最后，我们定义一些方法保存需要执行的运算（clickPlus、clickMinus、clickMultiply、clickDivide）；当按下=键时，再执行实际的计算（clickEquals），清除当前运算（clickClear）；当按下 Over 键时，分离分子与分母（clickOver）。还有一些方法（processDigit:、processOp:和 storeFracPart:）可帮助处理以上提到的一些琐事。

代码清单 21-2 显示了这个控制器类的实现文件。注意，我们在实现部分已声明了实例变量，也可以声明属性并合成这些属性（synthesize），这两种方式都可以。使用这种声明实例变量的方法可以保证实例变量是私有的，能够清楚地知道只在这个类的内部使用。

代码清单 21-2 FCViewController.m 实现文件

```
#import "FCViewController.h"
#import "Calculator.h"

@implementation FCViewController
{
    char                op;
    int                 currentNumber;
    BOOL                firstOperand, isNumerator;
    Calculator           *myCalculator;
    NSMutableString     *displayString;
}

@synthesize display;

-(void) viewDidLoad {

    //覆盖应用程序载入的自定义方法

    firstOperand = YES;
    isNumerator = YES;
    displayString = [NSMutableString stringWithCapacity: 40];
    myCalculator = [[Calculator alloc] init];
}
```

```

-(void) processDigit: (int) digit
{
    currentNumber = currentNumber * 10 + digit;

    [displayString appendString:
     [NSString stringWithFormat:@"%i", digit]];
    display.text = displayString;
}

- (IBAction) clickDigit: (UIButton *) sender
{
    int digit = sender.tag;

    [self processDigit: digit];
}

-(void) processOp: (char) theOp
{
    NSString *opStr;

    op = theOp;

    switch (theOp) {
        case '+':
            opStr = @" + ";
            break;
        case '-':
            opStr = @" - ";
            break;
        case '*':
            opStr = @" * ";
            break;
        case '/':
            opStr = @" / ";
            break;
    }

    [self storeFracPart];
    firstOperand = NO;
    isNumerator = YES;

    [displayString appendString: opStr];
    display.text = displayString;
}

-(void) storeFracPart
{
    if (firstOperand) {

```

```

        if (isNumerator) {
            myCalculator.operand1.numerator = currentNumber;
            myCalculator.operand1.denominator = 1; // 例如 3 * 4/5 =
        }
        else
            myCalculator.operand1.denominator = currentNumber;
    }
    else if (isNumerator) {
        myCalculator.operand2.numerator = currentNumber;
        myCalculator.operand2.denominator = 1; // 例如 3/2 * 4 =
    }
    else {
        myCalculator.operand2.denominator = currentNumber;
        firstOperand = YES;
    }

    currentNumber = 0;
}

-(IBAction) clickOver
{
    [self storeFracPart];
    isNumerator = NO;
    [displayString appendString: @"/"];
    display.text = displayString;
}

// 算术操作键

-(IBAction) clickPlus
{
    [self processOp: '+'];
}

-(IBAction) clickMinus
{
    [self processOp: '-'];
}

-(IBAction) clickMultiply
{
    [self processOp: '*'];
}

-(IBAction) clickDivide
{
    [self processOp: '/'];
}

```

```

// Misc 键

-(IBAction) clickEquals
{
    if ( firstOperand == NO ) {
        [self storeFracPart];
        [myCalculator performOperation: op];

        [displayString appendString: @" = "];
        [displayString appendString: [myCalculator.accumulator
            convertToString]];
        display.text = displayString;

        currentNumber = 0;
        isNumerator = YES;
        firstOperand = YES;
        [displayString setString: @""];
    }
}

-(IBAction) clickClear
{
    isNumerator = YES;
    firstOperand = YES;
    currentNumber = 0;
    [myCalculator clear];

    [displayString setString: @""];
    display.text = displayString;
}

@end

```

和前一个应用程序一样，计算器窗口仍然包含一个标签，我们仍然称之为 `display`。当用户逐位输入一个数时，我们需要同步构造这个数。变量 `currentNumber` 保存了这个输入中的数字，而 `BOOL` 变量 `firstOperand` 和 `isNumerator` 分别跟踪输入的是第一个还是第二个操作数，以及用户当前输入的是操作数的分子还是分母。

在计算器上按下一个数字按钮，我们会将它组装起来，这样可以将一些识别信息传递给 `clickDigit:` 方法，从而识别出按下的是哪一个数字按钮。这可以通过在属性检查面板中将每个数字按钮的 `tag` 属性设置为唯一值来实现。在这个例子中，我们把 `tag` 设置为对应的数字。因此，标记为 0 按钮的 `tag` 将被设置

为 0，而标记为 1 按钮的 tag 将被设置为 1，以此类推。发送到方法 clickDigit: 的 sender 参数实际上是在 iPhone 窗口中按下的 UIButton 对象。通过获取对象的 tag 属性，就可以获得按钮 tag 的值。这是在 clickDigit: 方法中完成的，语句如下：

```
- (IBAction) clickDigit: (UIButton *) sender
{
    int digit = sender.tag;

    [self processDigit: digit];
}
```

代码清单 21-2 中的按钮比第一个应用程序中要多得多。视图控制器实现文件中的复杂性大部分来自构造和显示分数。如前所述，按下数字按钮 0~9 时，就会执行操作方法 clickDigit:。该方法会调用 processDigit: 方法，并将相应的数字附加到变量 currentNumber 中正在构造的数字末尾。该方法还会把数字添加到变量 displayString 保存的当前显示字符串中，并更新显示内容：

```
-(void) processDigit: (int) digit
{
    currentNumber = currentNumber * 10 + digit;

    [displayString appendString:
     [NSString stringWithFormat:@"%i", digit]];
    display.text = displayString;
}
```

按下=键时，就会调用 clickEquals: 方法执行运算。计算器将执行两个分数之间的运算，并将结果保存到其累加器中。这个累加器的值是在 clickEquals 方法中获取的，然后把结果添加到显示中。

21.3.3 Fraction 类

Fraction 类在以前的例子和这个例子中几乎没有什么变化，只是增加了一个新的 convertToString 方法，用于将分数转换为相应的字符串表示。代码清单 21-2 显示了 Fraction 接口文件和对应的实现文件。

代码清单 21-2 Fraction.h 接口文件

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

@interface Fraction : NSObject
```



```

@property int numerator, denominator;

-(void)          print;
-(void)          setTo: (int) n over: (int) d;
-(Fraction *)    add: (Fraction *) f;
-(Fraction *)    subtract: (Fraction *) f;
-(Fraction *)    multiply: (Fraction *) f;
-(Fraction *)    divide: (Fraction *) f;
-(void)          reduce;
-(double)        convertToNum;
-(NSString *)    convertToString;

@end

```

代码清单 21-2 Fraction.m 实现文件

```

#import "Fraction.h"

@implementation Fraction

@synthesize numerator, denominator;

-(void) setTo: (int) n over: (int) d
{
    numerator = n;
    denominator = d;
}

-(void) print
{
    NSLog ("%i/%i", numerator, denominator);
}

-(double) convertToNum
{
    if (denominator != 0)
        return (double) numerator / denominator;
    else
        return NAN;
}

-(NSString *) convertToString
{
    if (numerator == denominator)
        if (numerator == 0)
            return @"0";
    else

```

```
        return @"1";
    else if (denominator == 1)
        return [NSString stringWithFormat:@"%i", numerator];
    else
        return [NSString stringWithFormat:@"%i/%i",
            numerator, denominator];
}
```

// 添加一个分数到消息的接收器

```
-(Fraction *) add: (Fraction *) f
{
    // 将两个分数相加:
    //  $a/b + c/d = ((a*d) + (b*c)) / (b * d)$ 

    // 存储相加后的结果
    Fraction *result = [[Fraction alloc] init];

    result.numerator = numerator * f.denominator +
        denominator * f.numerator;
    result.denominator = denominator * f.denominator;

    [result reduce];
    return result;
}
```

```
-(Fraction *) subtract: (Fraction *) f
{
    // 将两个分数相减:
    //  $a/b - c/d = ((a*d) - (b*c)) / (b * d)$ 

    Fraction *result = [[Fraction alloc] init];

    result.numerator = numerator * f.denominator -
        denominator * f.numerator;
    result.denominator = denominator * f.denominator;

    [result reduce];
    return result;
}
```

```
-(Fraction *) multiply: (Fraction *) f
{
    Fraction *result = [[Fraction alloc] init];

    result.numerator = numerator * f.numerator;
    result.denominator = denominator * f.denominator;
    [result reduce];
}
```

```

        return result;
    }

-(Fraction *) divide: (Fraction *) f
{
    Fraction *result = [[Fraction alloc] init];

    result.numerator = numerator * f.denominator
    result.denominator = denominator * f.numerator];
    [result reduce];

    return result;
}

- (void) reduce
{
    int u = numerator;
    int v = denominator;
    int temp;

    if (u == 0)
        return;
    else if (u < 0)
        u = -u;

    while (v != 0) {
        temp = u % v;
        u = v;
        v = temp;
    }

    numerator /= u;
    denominator /= u;
}

@end

```

convertToString:方法用于检查分数的分子和分母，以产生更适合查看的结果。如果分子和分母相等（但不是0），将返回@"1"。如果分子为0，则返回字符串@"0"。如果分母为1，则整个字符串不需要显示分母。

回忆一下曾经在 **convertToString:**方法内部使用的 **stringWithFormat:**方法，它根据一个指定格式的字符串（类似于 NSLog）和一个用逗号隔开的参数列表，返回一个字符串。将参数传递给一个参数数量不定的方法时，可以使用逗号将多个参数隔开，这与将参数传递给 NSLog 函数时的做法是一样的。

21.3.4 处理分数的 Calculator 类

接下来，需要了解 Calculator 类。这个类的概念类似于本书中前面开发的相同名称的类。然而在这个例子中，我们的计算器必须知道如何处理分数。下面给出了我们新的 Calculator 类接口与实现文件。

代码清单 21-2 Calculator.h 接口文件

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

@interface Calculator : NSObject

@property (strong, nonatomic) Fraction *operand1;
@property (strong, nonatomic) Fraction *operand2;
@property (strong, nonatomic) Fraction *accumulator;

-(Fraction *) performOperation: (char) op;
-(void) clear;

@end
```

代码清单 21-2 Calculator.m 实现文件

```
#import "Calculator.h"

@implementation Calculator

@synthesize operand1, operand2, accumulator;

-(id) init
{
    self = [super init];

    if (self) {
        operand1 = [[Fraction alloc] init];
        operand2 = [[Fraction alloc] init];
        accumulator = [[Fraction alloc] init];
    }

    return self;
}

-(void) clear
{
```

```

        accumulator.numerator = 0;
        accumulator.denominator = 0;
    }

    -(Fraction *) performOperation: (char) op
    {
        Fraction *result;

        switch (op) {
            case '+':
                result = [operand1 add: operand2];
                break;
            case '-':
                result = [operand1 subtract: operand2];
                break;
            case '*':
                result = [operand1 multiply: operand2];
                break;
            case '/':
                result = [operand1 divide: operand2];
                break;
        }

        accumulator.numerator = result.numerator;
        accumulator.denominator = result.denominator;

        return accumulator;
    }

@end

```

21.3.5 设计 UI

在这个项目中，nib 文件被称为 Fraction_CalculatorViewController.xib。选中界面设计文件，将按钮和标签布局到界面上，如图 21.17 所示（当然，界面设计可以自由发挥）。

建立视图中每个数字键和 `clickDigit:` 方法之间的连接。按住 **Ctrl** 键并单击拖动按钮到视图控制器的接口或实现文件中的 `clickDigit:` 方法。在检查器面板中为每一个数字键设定 **Tag** 值为与按钮标题对应的数字。如数字键标记为 0，则设置 **Tag** 值为 0；数字键标记为 1，则设置 **Tag** 值为 1，以此类推。

在 **View** 窗口中绘制其他的按钮并建立相应的连接。以上就是全部过程，

你的界面设计已经完成，现在可以使用这个分数计算器应用程序了。

21.4 小结

图 21.20 显示了 Xcode 项目窗口，在其中可以看到与分数计算器项目相关的所有文件。

下面总结了创建 iPhone 分数计算器应用程序的步骤：

- (1) 创建一个新的 Single View 应用程序。
- (2) 在 `FCViewController.h` 和 `.m` 文件中输入 UI 代码。
- (3) 给项目添加：Fraction 和 Calculator 类。
- (4) 打开 `Main.storyboard.xib`，以便创建 UI。

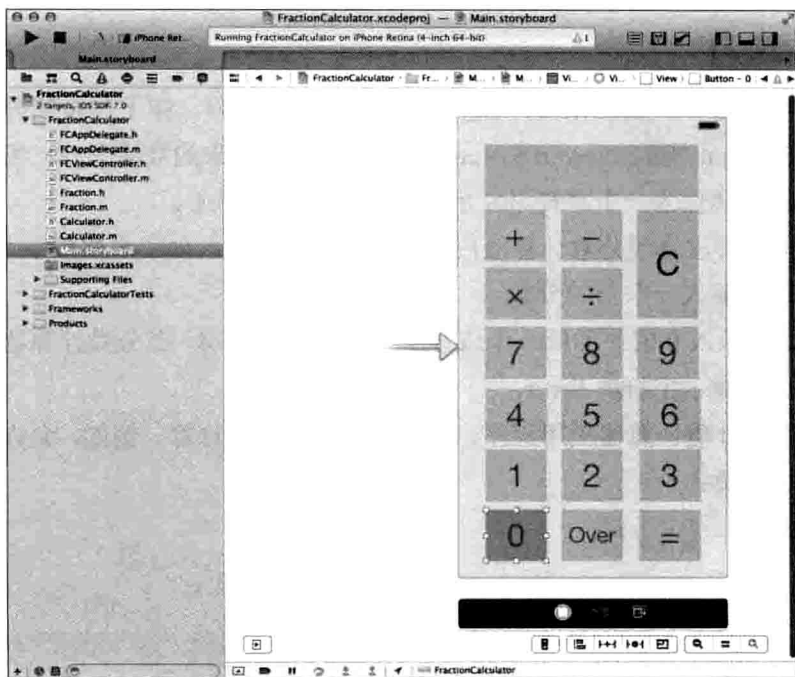


图 21.20 分数计算器项目文件

- (5) 将 View 窗口的背景改为浅灰色 (Mercury)。
- (6) 创建一个标签和多个按钮，并在 View 窗口中排好它们的位置。
- (7) 按下 `Ctrl` 键的同时，单击视图中的标签并拖动到 `UILabel IBOutlet` 属

性 display 上。

(8) 在 View 窗口中, 在按下 Ctrl 键的同时, 依次单击每个按钮并将其拖到相应的 IBAction 方法中。为每个数字按钮选择 clickDigit: 方法。此外, 将每个数字的 tag 属性设置为对应的数字 0~9, 这样 clickDigit: 方法就能识别按下的是哪个按钮。

希望这些对开发 iOS 应用程序的简要介绍, 能够为你编写自己的 iPhone 应用程序提供一个良好的起点。如前所述, UIKit 中提供的功能非常多, 等待着你去探讨!

我们的分数计算器具有几个限制。在下面的练习中需要解决其中的一些限制。

21.5 练习

1. 给分数计算器应用程序添加一个 Convert 按钮。按下此按钮时, 使用 Fraction 类的 convertToNum 方法生成分数结果的数值形式。将得到的结果转化为一个字符串显示在计算器的显示屏上。
2. 按照要求修改分数计算器应用程序: 如果在输入分子之前按下“-”键, 可以输入一个负的分。
3. 如果输入 0 值作为第一个或第二个操作数的分母, 在分数计算器的显示屏中显示字符串 Error。
4. 修改分数计算器应用程序, 让它可以进行连续计算。例如, 允许输入以下运算:

$$1/5 + 2/7 - 3/8 =$$

附录 A

术 语 表

本附录包含了很多会用到的非正式定义术语。有些术语与 Objective-C 语言有关，其他术语则有自己的语源，来自面向对象程序设计的规范。在后一种情况中，术语的含义只有明确应用于 Objective-C，才提供此类定义。

抽象类

为了方便地创建子类而定义的类。实例是从子类创建的，而不是抽象类。参见具体的子类。

存取方法

实例变量的取值或设值方法。使用取值方法可以获取实例变量的值，使用设值方法可以设置实例变量的值，这与数据封装的方法论是一致的。

Application Kit

用于开发应用程序用户界面的框架，用户界面包含各种对象，如菜单、工具栏和窗口。该框架是 Cocoa 的组成部分，通常称为 AppKit。

ARC

参见自动引用计数。

归档

将对象数据转换成一种以后可恢复（未归档）的格式。

数组

一个有序值的集合。数组可定义为 Objective-C 中的基本类型，通过 NSArray 和 NSMutableArray 类实现为 Foundation 下的对象。

自动引用计数（ARC）

用于编译器进行对象内存管理的特性。在 Xcode 4.2 版本之前，iOS 程序员需要手工使用方法管理内存，会使用到 `retain`、`release`、`autorelease` 和 `dealloc` 方法。

自动变量

输入一个语句块时会自动分配空间、退出语句块时会自动释放空间的变量。自动变量的作用域仅限于定义它的程序块之内，这些变量没有默认的初始值。在它们的前面可选择性地放置关键字 `auto`。

自动释放池

在使用 ARC 之前，自动释放池是一个由 `NSAutoreleasePool` 类管理的对象。现在通过 `@autoreleasepool` 指令实现。自动释放池会追踪需要系统延迟释放的对象。在 iOS 和 Cocoa 应用中典型的例子是，对象需要在循环完成后进行释放。

位域

包含一个或多个具有指定位宽度整数域的结构。可以访问和操作位域，方式与其他结构成员所采用的相同。

区块

Apple 公司对 C 语言的一种扩展。区块具有和函数相似的语义，可以获取区块定义范围内变量的值，可以将它赋值给一个变量，作为函数方法的参数进行传递。区块能够有效地调度到另一个线程或处理器执行。

分类

特定名称所代表的一组方法。分类可以模块化方法的定义，可用于向现有类添加新方法。

字符串

一种以 `null` 结尾的字符序列。

类

一组实例变量和可访问这些变量的方法。定义类之后，即可创建类的实例

(即对象)。

类方法

类对象调用的方法（定义有一个前导的+号）。参见实例方法。

类对象

标识特定类的对象。可以将类名用作消息的接收者调用类方法。在其他地方，可以对类调用 `class` 方法来创建类对象。

群集

组合了一组私有具体子类的抽象类，它通过抽象类向用户提供了一个简单的接口。

Cocoa

一种开发环境，它由 Foundation、Core Data 和 Application Kit 框架组成。

Cocoa Touch

一种开发环境，它由 Foundation、Core Data 和 UIKit 框架组成。

集合

一种 Foundation 框架对象，可以是数组、字典或用于分组和操作相关对象的集。

编译时

分析源代码并将其转换成所谓目标编码的底层格式的时期。

合成类

来自其他类对象组成的类。通常，它可替代子类的使用。

具体子类

抽象类的子类。可从具体子类创建实例。

符合

如果直接实现或间接继承的类采用了协议中的所有方法，则称该类符合这项协议。

常量字符串

引在一对双引号中的字符序列。如果以 `@character` 开头，通常定义 `NSString` 类型的常量字符串对象。

数据封装

将对象的数据存储在对象的实例变量中，并且只能通过对象的方法进行访问，这样可维护数据的完整性。

委托

让另一个对象实现某项行为的对象。

指定的初始化函数

将调用类或子类（通过向 `super` 发送消息）中其他所有初始化的方法。

字典

在 `Foundation` 下，利用 `NSDictionary` 和 `NSMutableDictionary` 类实现的键/值对集合。

指令

`Objective-C` 中的一种特殊结构，它以 `at` 符号（`@`）开始。`@interface`、`@implementation`、`@end` 和 `@class` 都是指令的例子。

分布式对象（Distributed Object）

一个应用程序中的 `Foundation` 对象与另一个（很可能是运行在另一台计算机上）应用程序的 `Foundation` 对象进行通信的能力。

动态绑定

在运行时而不是编译时确定对象需要调用的方法。

动态类型

在运行时而不是编译时确定对象所属的类。参见静态类型。

封装

参见数据封装。

extern 变量

参见全局变量。

工厂方法

参见类方法。

工厂对象

参见类对象。

正式协议

使用@protocol 指令定义在一个名称下的相关方法集。不同的类（不必是相关的）可以采用一个正式协议，只要实现（或继承）这个正式协议的所有方法即可。参见非正式协议。

转发

向另一个方法发送一条消息及相关（多个）参数，并进行执行的过程。

Foundation 框架

类、函数和协议的集合，这些类、函数和协议形成了应用程序开发的基础，提供了各种基本的工具性程序。如内存管理、文件和 URL 访问、归档，以及集合、字符串、数字和日期对象的使用。

框架

类、函数、协议、文档、头文件和其他相关资源的集合。如 Cocoa 框架是在 OS X 下开发交互式图形应用程序的框架。

函数

利用一个名称标识的语句块，它可以通过值传递一个或多个参数，并且可选择返回一个值。对于定义函数的文件而言，函数可以是局部的（静态的），也可以是全局的。后一种情况中，可以从定义在其他文件中的函数或方法调用这些函数。

垃圾回收

一种运行时的内存管理系统，可自动释放未被引用的对象所使用的内存。

在 iOS 运行环境中不支持垃圾回收。

gcc

它是 Free Software Foundation (FSF) 开发的一种编译器名称。gcc 支持多种程序设计语言，包括 C、Objective-C 和 C++。

gdb

由 gcc 编译的程序的调试工具。

取值方法

一种存取方法，可检索实例变量的值。参见赋值方法。

全局变量

在所有方法或函数外部定义的变量，同一个源文件中或将该变量定义为 extern 的其他源文件中的任何方法或函数都可以访问这个变量。

头文件

包含有共同的定义、宏和变量声明的文件，可以使用 #import 或 #include 语句将这种文件包含到程序中。

id

通用数据类型，可容纳指向任何类型对象的指针。

不可变对象

不能修改值的对象。如 Foundation 框架中包含的 NSString、NSDictionary 和 NSArray 对象。参见可变对象。

实现部分

类定义的部分，它包含声明在相应接口部分（或者由协议定义所指定的）方法的实际代码（即实现）。

非正式协议

作为一个分类（通常作为根类的分类）声明的逻辑上相关的方法集。与正式协议不同，非正式协议中的方法不必全部实现。参见正式协议。

继承

将一个类的方法和实例变量从根对象开始向下传递到子类的过程。

实例

类的具体表示。实例通常通过向类对象发送一条 `alloc` 或 `new` 消息来创建的对象。

实例方法

可被类实例调用的方法。参见类方法。

实例变量

在接口部分（它包含该对象的每个实例）声明的（或从父类继承来的）变量。实例方法可直接访问它们的实例变量。

Interface Builder

OS X 下为应用程序构建图形用户界面的工具。

接口部分

用于声明类、类的超类、实例变量和方法的部分。对每个方法而言，还需声明参数类型和返回类型。参见实现部分。

国际化

参见本地化。

isa

在根对象中定义并且所有的对象都要继承的一个特殊的实例变量。`isa`变量用于在运行时识别对象所属的类。最近也用来存储对象有关的附属信息。

链接

利用一个或多个对象文件并将它们转换成可执行程序的过程。

局部变量

作用域限于定义它的程序块之内的变量。对于方法、函数或语句块，变量可以是局部的。

本地化

使程序适合在特定的地理区域内执行的过程。通常是通过将消息转换成本地语言，并处理各种情况（如当地时区、货币符号、日期格式等）实现的。有时本地化只是指语言翻译过程，而术语国际化则表示这一过程的其他方面。

消息

发送给对象（接收者）的方法及相应的参数。

消息表达式

括在一对方括号中的表达式，它指定对象（接收者）和发送给该对象的消息。

方法

属于某个类的过程，通过向该类的对象或实例发送消息，可以执行方法。参见类方法和实例方法。

模块

在 Xcode 5 中添加了一项功能，能够更加方便、高效地在项目中包含框架和头文件。需要使用 `@import` 指令支持。

可变对象

值可更改的对象。Foundation 框架支持可变和不可变数组、集、字符串和字典。参见不可变对象。

nil

一个 id 类型的对象，用来表示无效对象。它的值定义为 0。可向 nil 发送消息。

通知

当发生特殊事件时，向已注册的可收到警告（通知）的对象发送消息的过程。

NSObject

Foundation 框架下的根对象。

空字符

值为 0 的字符。空字符常量用 '\0' 表示。

空指针

无效的指针值。通常定义为 0。

对象

一组变量和相应的方法。可以向对象发送消息来执行它的方法。

面向对象的程序设计

一种基于类、对象和对象执行操作的程序设计方法。

父类

被其他类继承的类。也可称为超类。

指针

用于引用另一个对象或数据类型的值。指针在内存中作为特定对象或值的地址来实现。类的实例是一个指针，它指向内存中保存对象数据的位置。

多态

来自不同类的对象可接受同一消息的能力。

预处理程序

首次执行源代码处理行的程序，它以一个 # 开始，还可能包含特殊的预处理程序语句。常见的用途是使用 #define 来定义宏指令，包括用 #import 和 #include 导入其他源文件，以及用 #if、#ifdef 和 #ifndef 有条件地包含源程序行。

过程式程序设计语言

使用过程和函数定义程序的语言，过程和函数可操作一组数据。

属性声明

这种方法可指定实例变量的属性，允许编译器为实例变量生成无内存泄漏

并且线程安全的存取方法。属性声明也可用于声明存取方法的属性，这些方法在运行时动态加载。

属性列表

使用标准化的和可移植的格式的不同类型对象的表示。通常，属性列表以 XML 格式进行存储。

协议

类为了符合协议或采用协议而必须实现的方法列表。协议提供了跨多个类标准化接口的方式。参见正式协议和非正式协议。

接收者

消息发送到的对象。可以从调用的方法内部作为 `self` 来引用接收者。

引用计数

参见保持计数。

保持计数

关于引用对象次数的计数。通过向该对象发送 `retain` 消息对其执行加 1 操作，发送 `release` 消息执行减 1 操作。

根对象

位于继承层次结构中最顶层且没有父类的对象。

运行时

程序正在执行的那段时间；也指负责执行程序指令的机制。

选择程序（selector）

用于选择对象要执行的方法名称。编译的选择程序是 `SEL` 类型的，并且可用 `@selector` 指令生成。

self

在方法内使用的变量，用于引用消息的接收者。

集

单值对象的无序集合，可使用 `NSSet`、`NSMutableSet` 和 `NSCountedSet` 类在 Foundation 下实现。

设值方法

这种存取方法可设置实例变量的值。参见取值方法。

语句

以分号结束的一个或多个表达式。

语句块

括在一对花括号内的一条或多条语句。局部变量可以在语句块内声明，而它们的作用域也限制在该语句块中。

静态函数

使用关键字 `static` 声明的函数，只能由定义在同一源文件中的其他函数或方法调用它。

静态类型

在编译时显式地识别对象所属的类。参见动态类型。

静态变量

其作用域限制在定义它的块或模块内的变量。静态变量具有默认的初始值 0，且在方法或函数的调用过程中会保持它们的值。

结构

一种集合数据类型，它包含类型不相同的成员。可将结构赋值给其他结构，作为参数传递给函数和方法，还可由函数和方法返回。

子类 (subclass)

也称为“child class”，子类从它的父类或超类继承方法和实例变量。

super

方法中使用的关键字，用于引用接收者的父类。

超类

特定类的父类。参见 `super`。

合并方法

编译器自动创建的一种赋值方法或取值方法。Objective-C 2.0 语言中添加了这个方法。

UIKit

在 iOS 设备上开发应用程序的框架。除了提供可使用普通 UI 元素（如窗口、按钮和标签）的类以外，它还定义了处理设备特定功能的类型，如加速计和触摸界面。UIKit 是 Cocoa Touch 的一部分。

Unicode 字符

一种包含数百万字符的字符集中代表字符的标准。可使用 `NSString` 和 `NSMutableString` 类处理包含在 Unicode 字符中的字符串。

联合

一种与结构类似的集合数据类型，它包含的成员共享一个存储区。任意时间只有一个成员可以占用此存储区。

Xcode

一种用于 Mac OS X 和 iOS 程序开发的编译和调试工具。

XML

可扩展标记语言。它是 Mac OS X 上生成的属性列表的默认格式。

附录 B

地址簿示例源代码

为了便于参考，下面给出本书第二部分中一直使用的地址簿示例完整的接口文件和实现文件。该附录包含 `AddressCard` 和 `AddressBook` 类的定义。你应该在你的系统上实现这些文件，然后扩展这些类定义，以使它们更实用，功能更强。这是你学习这门语言并熟悉生成程序、使用类和对象以及使用 `Foundation` 框架的一种极好方式。

B.1 `AddressCard` 接口文件

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

@interface AddressCard : NSObject <NSCopying, NSCoding>

@property (nonatomic, copy) NSString *name, *email;

-(void) setName: (NSString *) theName andEmail: (NSString *) theEmail;
-(void) assignName: (NSString *) theName andEmail: (NSString *) theEmail;
-(NSComparisonResult) compareNames: (id) element;

-(void) print;

@end
```

B.2 `AddressBook` 接口文件

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

@interface AddressBook: NSObject <NSCopying, NSCoding>

@property (nonatomic, copy) NSString *bookName;
@property (nonatomic, strong) NSMutableArray *book;

-(id) initWithName: (NSString *) name;
```

```

-(void) sort;
-(void) sort2;
-(void) addCard: (AddressCard *) theCard;
-(void) removeCard: (AddressCard *) theCard;
-(int) entries;
-(void) list;
-(AddressCard *) lookup: (NSString *) theName;

```

```
@end
```

B.3 AddressCard 实现文件

```

#import "AddressCard.h"

@implementation AddressCard

@synthesize name, email;

-(void) setName: (NSString *) theName andEmail: (NSString *) theEmail
{
    self.name = theName;
    self.email = theEmail;
}

//比较指定地址卡片的名称
-(NSComparisonResult) compareNames: (id) element
{
    return [name compare: [element name]];
}

-(void) print
{
    NSLog(@"=====");
    NSLog(@"|                               |");
    NSLog(@"| %-31s |", [name UTF8String]);
    NSLog(@"| %-31s |", [email UTF8String]);
    NSLog(@"|                               |");
    NSLog(@"|                               |");
    NSLog(@"|                               |");
    NSLog(@"|               O               O |");
    NSLog(@"|=====");
}

-(id) copyWithZone: (NSZone *) zone
{
    id newCard = [[[self class] allocWithZone: zone] init];

    [newCard retainName: name andEmail: email];
}

```

```

        return newCard;
    }

    -(void) assignName: (NSString *) theName andEmail: (NSString *) theEmail
    {
        name = theName;
        email = theEmail;
    }

    -(void) encodeWithCoder: (NSCoder *) encoder
    {
        [encoder encodeObject: name forKey: @"AddressCardName"];
        [encoder encodeObject: email forKey: @"AddressCardEmail"];
    }

    -(id) initWithCoder: (NSCoder *) decoder
    {
        name = [decoder decodeObjectForKey: @"AddressCardName"];
        email = [decoder decodeObjectForKey: @"AddressCardEmail"];

        return self;
    }
@end

```

B.4 AddressBook 实现文件

```

#import "AddressBook.h"

@implementation AddressBook

@synthesize book, bookName;

//设置 AddressBook 的名称和一个空的地址簿

-(instancetype) initWithName: (NSString *) name
{
    self = [super init];

    if (self) {
        bookName = [NSString stringWithString: name];
        book = [NSMutableArray array];
    }

    return self;
}

-(instancetype) init
{

```

```

    return [self initWithName: @"Unnamed Book"];
}

//编写自己的地址簿设置方法, 以便创建一个可变的副本

-(void) setBook: (NSArray *) theBook
{
    book = [theBook mutableCopy];
}

-(void) sort
{
    [book sortUsingSelector: @selector(compareNames)];
}

//使用区块取代 sort 方法

-(void) sort2
{
    [book sortUsingComparator:
        ^(id obj1, id obj2) {
            return [[obj1 name] compare: [obj2 name]];
        }];
}

-(void) addCard: (AddressCard *) theCard
{
    [book addObject: theCard];
}

-(void) removeCard: (AddressCard *) theCard
{
    [book removeObjectIdenticalTo: theCard];
}

-(int) entries
{
    return [book count];
}

-(void) list
{
    NSLog(@"===== Contents of: %@ =====", bookName);

    for ( AddressCard *theCard in book )
        NSLog(@"%-20s  %-32s", [theCard.name UTF8String],
            [theCard.email UTF8String]);

    NSLog(@"=====");
}

```

```
}

//通过名称查找地址卡片——假定精确匹配

-(AddressCard *) lookup: (NSString *) theName
{
    for ( AddressCard *nextCard in book )
        if ( [[nextCard name] caseInsensitiveCompare: theName]
            == NSOrderedSame )
            return nextCard;

    return nil;
}

-(void) encodeWithCoder: (NSCoder *) encoder
{
    [encoder encodeObject:bookName forKey: @"AddressBookBookName"];
    [encoder encodeObject:book forKey: @"AddressBookBook"];
}

-(id) initWithCoder: (NSCoder *) decoder
{
    bookName = [[decoder decodeObjectForKey: @"AddressBookBookName"];
    book = [decoder decodeObjectForKey: @"AddressBookBook"];

    return self;
}

// NSCopying 协议的方法

-(id) copyWithZone: (NSZone *) zone
{
    id newBook = [[[self class] allocWithZone: zone] init];

    [newBook setBookName: bookName];

    // 下面执行地址簿的浅复制

    newBook setBook: book];

    return newBook;
}

@end
```
