HZ BOOKS 华章科技
PEARSON
EFFECTIVE 系列丛书

"Effective Software Development Series"系列经典著作，世界级软件开发大师Scott Meyers亲自担当顾问编辑，Amazon全五星评价

从语法、接口与API设计、内存管理、框架等方面总结和探讨了Objective-C编程中52个鲜为人知和容易被忽视的特性与陷阱

包含大量实用范例代码，为编写易于理解、便于维护、易于扩展和高效的Objective-C应用提供了解决方案

Effective Objective-C 2.0
52 Specific Ways to Improve Your iOS and OS X Programs

Effective Objective-C 2.0

编写高质量iOS与OS X代码的52个有效方法

（英） Matt Galloway 著
爱飞翔 译

机械工业出版社
China Machine Press

Effective Objective-C 2.0

52 Specific Ways to Improve Your iOS and OS X Programs

用Objective-C 2.0编写优秀的iOS及OS X代码！

本书介绍如何利用Objective-C语言强大的表现力，写出能够在生产环境中出色运行的iOS及OS X代码。Scott Meyers在其畅销书《Effective C++》中，首创了一套以应用场景来精准讲解编程技巧的行文风格，而本书作者Matt Galloway则采用此方式汇集了52条Objective-C编程心得，其中含有各种技巧与快速解决方案，以及在别处难以找到的真实工作环境里的范例代码。

作者通过这些实用范例，向大家揭示了Objective-C中鲜为人知的各种奇怪现象与易错之处，同时还讲解了很多能够改善代码行为并提高程序性能的复杂特性。读者将在书中学到如何从多个方案里选出一种最高效的办法来完成关键编程任务，还能掌握怎样写出易于理解、易于维护、易于扩展的代码。除了语言的核心部分之外，书中还介绍了如何使用Foundation框架中的类，以及如何将“大中枢派发”等当前流行的系统库集成到自己的项目中。

本书内容包括：

- 优化Objective-C对象之间的互动与关系。
- 掌握接口与API的设计原则，写出令开发者用起来得心应手的类。
- 善用协议与分类，编写便于维护且不易出现bug的代码。
- 在自动引用计数（ARC）环境下避免内存泄漏。
- 用“块”与“大中枢派发”编写呈模块化且功能强大的代码。
- 理解Objective-C中的协议与其他编程语言中的多重继承有何区别，并掌握协议的用法。
- 通过数组、字典、set等collection对象来提高代码性能。
- 揭示Cocoa与Cocoa Touch框架的强大之处。

PEARSON

www.pearson.com

PEARSON

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/程序设计/Objective-C

ISBN 978-7-111-45129-7

9 787111 451297 >

定价：69.00元

Effective Objective-C 2.0

52 Specific Ways to Improve Your iOS and OS X Programs

Effective Objective-C 2.0

编写高质量iOS与OS X代码的52个有效方法

（英） Matt Galloway 著

爱飞翔 译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Effective Objective-C 2.0：编写高质量 iOS 与 OS X 代码的 52 个有效方法 /（英）加洛韦（Galloway，M.）著；爱飞翔译 . —北京：机械工业出版社，2014.1
（Effective 系列丛书）
书名原文：Effective Objective-C 2.0: 52 Specific Ways to Improve Your iOS and OS X Programs

ISBN 978-7-111-45129-7

I. E…　II. ①加…　②爱…　III. C 语言—程序设计　IV. TP312

中国版本图书馆 CIP 数据核字（2013）第 300048 号

本书是世界级 C++ 开发大师 Scott Meyers 亲自担当顾问编辑的“ Effective Software Development Series”系列丛书中的新作，Amazon 全五星评价。从语法、接口与 API 设计、内存管理、框架等 7 大方面总结和探讨了 Objective-C 编程中 52 个鲜为人知和容易被忽视的特性与陷阱。书中包含大量实用范例代码，为编写易于理解、便于维护、易于扩展和高效的 Objective-C 应用提供了解决方案。

全书共 7 章。第 1 章通论与 Objective-C 的核心概念相关的技巧；第 2 章讲述的技巧与面向对象语言的重要特征（对象、消息和运行期）相关；第 3 章介绍的技巧与接口和 API 设计相关；第 4 章讲述协议与分类相关的技巧；第 5 章介绍内存管理中易犯的错误以及如何避免犯这些错误；第 6 章介绍块与大中枢派发相关的技巧；第 7 章讲解使用 Cocoa 和 Cocoa Touch 系统框架时的相关技巧。

机械工业出版社（北京市西城区百万庄大街 22 号　　邮政编码　100037）
责任编辑：关　敏
北京市荣盛彩色印刷有限公司印刷
2014 年 1 月第 1 版第 1 次印刷
186mm × 240mm · 13.75 印张
标准书号：ISBN 978-7-111-45129-7
定　　价：69.00 元

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

译 者 序

看到 Effective 这个词，大家一定会想到《 Effective C++ 》、《 Effective Java 》等业界名著，那些书里既汇聚了多项实用技巧，又系统而深入地讲解了各种编程知识。那么这本《 Effective Objective-C 2.0》是否也是如此呢？没错，它再次演绎了经典！

作为 Mac OS X 与 iOS 应用程序的开发语言，Objective-C 近年来颇受关注。尤其是智能手机与平板电脑兴起之后，越来越多的开发者都将目光转向移动平台，并开始学习 Objective-C。与 C++、Java 一样，Objective-C 也有“入门易，精通难”的问题。而本书作者 Matt Galloway 有多年开发经验，他将工作中遇到的各种问题分成 7 大类 52 小项，逐条罗列出来。在研读过程中，你不仅可以找到具体解决办法，而且还能体会到不同解决方案之间的优劣，更为重要的是，本书深入探讨了语言里一些鲜为人知或容易被人忽视的特性，令开发者明白其微妙之处，从而在实际工作中避开这些陷阱。书中每条心得几乎都给出了相当实用的范例代码，读者可直接将其运用在实际编程中，也可按照需要稍加改编，并举一反三，类推出更多相关技巧来。

除了讲解 Objective-C 语言本身外，书里还讲了与其密不可分的各种框架，相信你阅读完之后，会更深入地了解这门语言，同时编写易于理解、易于维护、易于扩展的高效应用程序应该也不再是难事了。在“从入门到精通”的过程中，本书定是你的良师益友。

本书由爱飞翔翻译，舒亚林和张军也参与了部分翻译工作，译文最后由爱飞翔统一整理。翻译过程中，得到机械工业出版社华章公司诸位编辑与工作人员的帮助，在此深表谢意。

由于时间仓促，译者水平有限，错误与疏漏之处敬请读者批评指正。大家可访问网页 http://agilemobidev.com/eastarlee/book/effective-objective-c 留言，亦可发电子邮件至 eastarstormlee@gmail.com。

爱飞翔

前　言

许多人认为 Objective-C 这门语言芜杂、笨拙、别扭，但笔者却看到其雅致、灵活、美观的一面。然而，为了领略这些优点，大家不仅要掌握基础知识，还要理解语言中的特性、陷阱及繁难之处。本书正是要讲述这些内容。

关于本书

本书假定读者已经熟悉了 Objective-C 的语法，所以不再赘述。笔者要讲的是怎样完全发挥这门语言的优势，以编写出良好的代码。由于其源自 Smalltalk，所以 Objective-C 是一门相当动态的语言。在其他语言中，许多工作都由编译器来完成；而在 Objective-C 中，则要于“运行期”（runtime）执行。于是，在测试环境下能正常运行的函数到了工作环境中，也许就会因为处理了无效数据而不能正确执行。避免此类问题的最佳方案当然是一开始就把代码写好。

严格地说，许多话题与 Objective-C 的核心部分并无关联。本书要谈到系统库中的技术，例如 libdispatch 库的“大中枢派发”（Grand Central Dispatch）等。因为当前所说的 Objective-C 开发就是指开发 Mac OS X 或 iOS 应用程序，所以，书中也要提及 Foundation 框架中的许多类，而不仅仅是其基类 NSObject。不论开发 Mac OS X 程序还是 iOS 程序，都无疑会用到系统框架，前者所用的各框架统称为 Cocoa，后者则叫 Cocoa Touch。

随着 iOS 的兴起，许多开发者都涌入 Objective-C 开发阵营。有的程序员初学编程，有的具备 Java 或 C++ 基础，还有的则是网页开发者出身。为了能高效运用 Objective-C，无论是哪种情况，你都必须花时间研究这门语言，从而写出执行迅速、便于维护、不易出错的代码来。

尽管这本书只用 6 个月就写好了，但是其酝酿过程却长达数年。笔者某天心血来潮，买了个 iPod Touch，等到第一版 SDK 发布之后，就决定开发个程序玩玩。我做的第一个“应用程序”叫“Subnet Calc”，其下载量比预想中要多。于是我确信，以后要和这个美妙的语言结缘了。从此就一直研究 Objective-C，并定期在自己的网站 www.galloway.me.uk 上发表博文。我对该语言的内部工作原理，诸如“块”（block）、“自动引用计数”（Auto Reference Count，ARC）等特别感兴趣。于是，

在有机会写作一本讲 Objective-C 的书时，自然就当仁不让了。

为使本书物尽其用，笔者建议大家跳读，直接翻到最感兴趣或与当前工作有关的章节来看。可以分开阅读每条技巧，也可以按其中所引用的条目跳至其他话题，互相参照。相关技巧归并成章，读者可根据各章标题快速找到谈及某个语言特性的数条技巧。

目标读者

本书面向那些有志于深入研究 Objective-C 的开发者，帮助其编写便于维护、执行迅速且不易出错的代码。如果你目前还不是 Objective-C 程序员，但是会用 Java 或 C++ 这样面向对象的语言，那么仍可阅读此书。在这种情况下，应先了解 Objective-C 的语法。

本书主要内容

本书不打算讲 Objective-C 语言的基础知识，在许多教材和参考资料中都能找到那些内容。本书要讲的是如何高效运用这门语言。书中内容分为若干条目，每条都是一小块易于理解的文字。这些条目按其所谈话题组织为如下各章。

第 1 章：熟悉 Objective-C

通论该语言的核心概念。

第 2 章：对象、消息、运行期

对象之间能够关联与交互，这是面向对象语言的重要特征。本章讲述这些特征，并深入研究代码在运行期的行为。

第 3 章：接口与 API 设计

很少有那种写完就不再复用的代码。即使代码不向更多人公开，也依然有可能用在自己的多个项目中。本章讲解如何编写与 Objective-C 搭配得宜的类。

第 4 章：协议与分类

协议与分类是两个需要掌握的重要语言特性。若运用得当，则可令代码易读、易维护且少出错。本章将帮助读者精通这两个概念。

第 5 章：内存管理

Objective-C 语言以引用计数来管理内存，这令许多初学者纠结，要是用过以“垃圾收集器”（garbage collector）来管理内存的语言，那么更会如此。“自动引用计数”机制缓解了此问题，不过使用时有很多重要的注意事项，以确保对象模型正确，不致内存泄漏。本章提醒读者注意内存管理中易犯的错误。

第 6 章：块与大中枢派发

苹果公司引入了“块”这一概念，其语法类似于C语言扩展中的“闭包”（closure）。在Objective-C语言中，我们通常用块来实现一些原来需要很多样板代码才能完成的事情，块还能实现“代码分离”（code separation）。“大中枢派发”（Grand Central Dispatch，GCD）提供了一套用于多线程环境的简单接口。“块”可视为GCD的任务，根据系统资源状况，这些任务也许能并发执行。本章将教会读者如何充分运用这两项核心技术。

第 7 章：系统框架

大家通常会用Objective-C来开发Mac OS X或iOS程序。在这两种情况下都有一套完整的系统框架可供使用，前者名为Cocoa，后者名为Cocoa Touch。本章将总览这些框架，并深入研究其中某些类。

致　　谢

在问到是否愿意写一本 Objective-C 的书时，我立刻兴奋起来。读过了 Effective 系列其他书后，我意识到要想写好这本 Objective-C 书籍可真是个挑战。然而在众人协助之下，这本书终于和大家见面了。

书中好些灵感都源自许多专述 Objective-C 的精彩博文。笔者要感谢博文作者 Mike Ash、Matt Gallagher 及“ bbum”等人。多年来，这些博客帮助我更深地理解了 Objective-C 语言。在编撰本书时，NSHipster 及 Mattt Thompson 所写的优秀文章也启迪了我的思路。还要感谢苹果公司提供了极为有用的开发文档。

在供职于 MX Telecom 期间，得良师益友之助，我学到了许多知识，若没有这段经历，恐怕就无法完成此书了。感谢 Matthew Hodgson，令我有机会以一套成熟的 C++ 代码库为基础，开发出公司首个 iOS 应用程序，在该项目中学到的本领为我参与后续项目打下了基础。

感谢历年来保持联系的各位同仁。大家时而切磋技艺，时而把酒言欢，这对我写作本书来说都是种帮助。

与培生集团旗下团队的合作相当愉快。Trina MacDonald、Olivia Basegio、Scott Meyers 及 Chris Zahn 都在需要时给我以帮助与鼓励。诸位为我提供了专心写书的工具，并回答了必要的问题。

笔者同技术编辑合作得也非常融洽，你们给了我莫大的帮助。仰赖严格的审校，方能使本书内容臻于完美。诸位在检查书稿时认真细致的态度，也令人称赞。

最后我要说，此书能问世，爱妻 Helen 的理解与支持必不可少。准备动笔那天，我们的第一个孩子降生了，真正开始写作是在几天之后。Helen 与 Rosie 伴我顺利写完这本书，你们俩真棒！

目　　录

第 1 章
熟悉 Objective-C

Objective-C 通过一套全新语法，在 C 语言基础上添加了面向对象特性。Objective-C 的语法中频繁使用方括号，而且不吝于写出极长的方法名，这通常令许多人觉得此语言较为冗长。其实这样写出来的代码十分易读，只是 C++ 或 Java 程序员不太能适应。

Objective-C 语言学起来很快，但有很多微妙细节需注意，而且还有许多容易为人所忽视的特性。另一方面，有些开发者并未完全理解或是容易滥用某些特性，导致写出来的代码难于维护且不易调试。本章讲解基础知识，后续各章谈论语言及其相关框架中的各个特定话题。

第 1 条：了解 Objective-C 语言的起源

Objective-C 与 C++、Java 等面向对象语言类似，不过很多方面有所差别。若是用过另一种面向对象语言，那么就能理解 Objective-C 所用的许多范式与模板了。然而语法上也许会显得陌生，因为该语言使用“消息结构”（messaging structure）而非“函数调用”（function calling）。Objective-C 语言由 Smalltalk[⊖] 演化而来，后者是消息型语言的鼻祖。消息与函数调用之间的区别看上去就像这样：

```
// Messaging (Objective-C)
Object *obj = [Object new];
[obj performWith:parameter1 and:parameter2];

// Function calling (C++)
Object *obj = new Object;
obj->perform(parameter1, parameter2);
```

关键区别在于：使用消息结构的语言，其运行时所应执行的代码由运行环境来决定；而使用函数调用的语言，则由编译器决定。如果范例代码中调用的函数是多态的，那么在运行

⊖ 20 世纪 70 年代出现的一种面向对象语言，详情参见：http://en.wikipedia.org/wiki/Smalltalk。——译者注

时就要按照“虚方法表”(virtual table)[⊖]来查出到底应该执行哪个函数实现。而采用消息结构的语言，不论是否多态，总是在运行时才会去查找所要执行的方法。实际上，编译器甚至不关心接收消息的对象是何种类型。接收消息的对象问题也要在运行时处理，其过程叫做“动态绑定”(dynamic binding)，第 11 条会详述其细节。

Objective-C 的重要工作都由“运行期组件”(runtime component)而非编译器来完成。使用 Objective-C 的面向对象特性所需的全部数据结构及函数都在运行期组件里面。举例来说，运行期组件中含有全部内存管理方法。运行期组件本质上就是一种与开发者所编代码相链接的“动态库”(dynamic library)，其代码能把开发者编写的所有程序粘合起来。这样的话，只需更新运行期组件，即可提升应用程序性能。而那种许多工作都在“编译期”(compile time)完成的语言，若想获得类似的性能提升，则要重新编译应用程序代码。

Objective-C 是 C 的“超集”(superset)，所以 C 语言中的所有功能在编写 Objective-C 代码时依然适用。因此，必须同时掌握 C 与 Objective-C 这两门语言的核心概念，方能写出高效的 Objective-C 代码来。其中尤为重要的是要理解 C 语言的内存模型(memory model)，这有助于理解 Objective-C 的内存模型及其“引用计数”(reference counting)机制的工作原理。若要理解内存模型，则需明白：Objective-C 语言中的指针是用来指示对象的。想要声明一个变量，令其指代某个对象，可用如下语法：

```
NSString *someString = @"The string";
```

这种语法基本上是照搬 C 语言的，它声明了一个名为 someString 的变量，其类型是 NSString*。也就是说，此变量为指向 NSString 的指针。所有 Objective-C 语言的对象都必须这样声明，因为对象所占内存总是分配在“堆空间”(heap space)中，而绝不会分配在“栈”(stack)上。不能在栈中分配 Objective-C 对象：

```
NSString stackString;
// error: interface type cannot be statically allocated
```

someString 变量指向分配在堆里的某块内存，其中含有一个 NSString 对象。也就是说，如果再创建一个变量，令其指向同一地址，那么并不拷贝该对象，只是这两个变量会同时指向此对象：

```
NSString *someString = @"The string";
NSString *anotherString = someString;
```

只有一个 NSString 实例，然而有两个变量指向此实例。两个变量都是 NSString* 型，这说明当前“栈帧”(stack frame)里分配了两块内存，每块内存的大小都能容下一枚指针(在 32 位架构的计算机上是 4 字节，64 位计算机上是 8 字节)。这两块内存里的值都一样，就是

⊖ virtual method table 是编程语言为实现“动态派发”(dynamic dispatch)或“运行时方法绑定”(runtime method binding)而采用的一种机制。——译者注

NSString 实例的内存地址。

图 1-1 描述了此时的内存布局。存放在 NSString 实例中的数据含有代表字符串实际内容的字节。

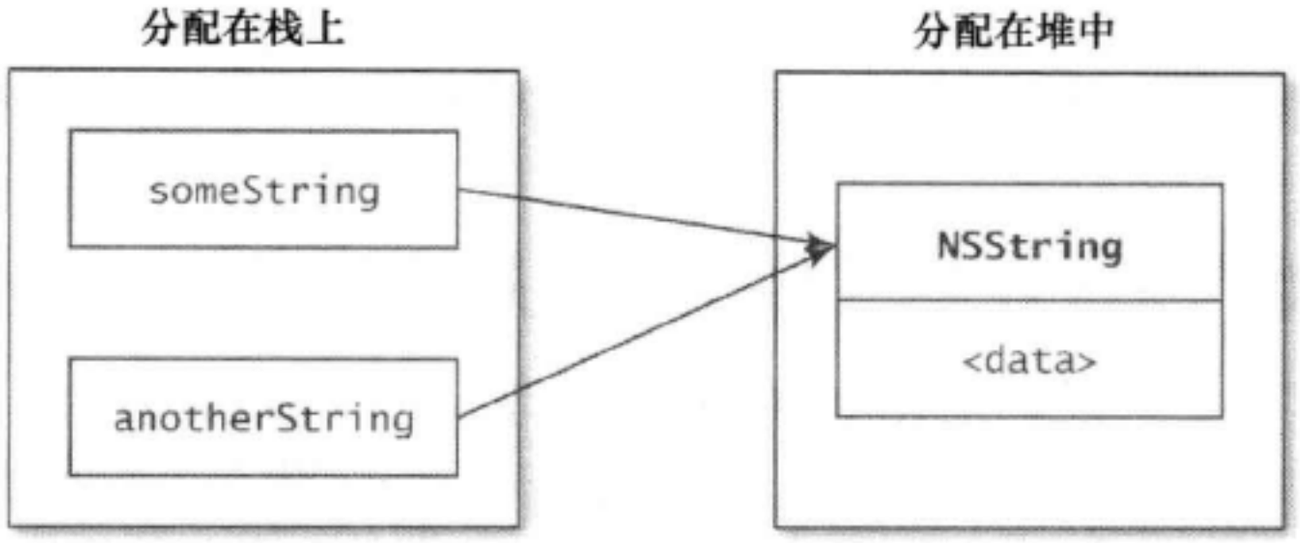

图 1-1 此内存布局图演示了一个分配在堆中的 NSString 实例，有两个分配在栈上的指针指向该实例

分配在堆中的内存必须直接管理，而分配在栈上用于保存变量的内存则会在其栈帧弹出时自动清理。

Objective-C 将堆内存管理抽象出来了。不需要用 malloc 及 free 来分配或释放对象所占内存。Objective-C 运行期环境把这部分工作抽象为一套内存管理架构，名叫“引用计数”（参见第 29 条）。

在 Objective-C 代码中，有时会遇到定义里不含 * 的变量，它们可能会使用“栈空间”（stack space）。这些变量所保存的不是 Objective-C 对象。比如 CoreGraphics 框架中的 CGRect 就是个例子：

```
CGRect frame;
frame.origin.x = 0.0f;
frame.origin.y = 10.0f;
frame.size.width = 100.0f;
frame.size.height = 150.0f;
```

CGRect 是 C 结构体，其定义是：

```
struct CGRect {
  CGPoint origin;
  CGSize size;
};
typedef struct CGRect CGRect;
```

整个系统框架都在使用这种结构体，因为如果改用 Objective-C 对象来做的话，性能会受影响。与创建结构体相比，创建对象还需要额外开销，例如分配及释放堆内存等。如果只需保存 int、float、double、char 等“非对象类型”（nonobject type），那么通常使用 CGRect 这种结构体就可以了。

在着手编写 Objective-C 代码之前，建议读者先看看 C 语言教程，以熟悉其语法。若是还没熟悉 C 语言就直接进入 Objective-C 的话，那么某些语法也许会令你困惑。

要点

- Objective-C 为 C 语言添加了面向对象特性，是其超集。Objective-C 使用动态绑定的消息结构，也就是说，在运行时才会检查对象类型。接收一条消息之后，究竟应执行何种代码，由运行期环境而非编译器来决定。
- 理解 C 语言的核心概念有助于写好 Objective-C 程序。尤其要掌握内存模型与指针。

第 2 条：在类的头文件中尽量少引入其他头文件

与 C 和 C++ 一样，Objective-C 也使用“头文件”（header file）与“实现文件”（implementation file）来区隔代码。用 Objective-C 语言编写“类”（class）的标准方式为：以类名做文件名，分别创建两个文件，头文件后缀用 .h，实现文件后缀用 .m。创建好一个类之后，其代码看上去如下所示：

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

@interface EOCPerson : NSObject
@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;
@end
// EOCPerson.m
#import "EOCPerson.h"

@implementation EOCPerson
// Implementation of methods
@end
```

用 Objective-C 语言编写任何类几乎都需要引入 Foundation.h。如果不在该类本身引入这个文件的话，那么就要引入与其超类所属框架相对应的“基本头文件”（base header file）。例如，在创建 iOS 应用程序时，通常会继承 UIViewController 类。而这些子类的头文件需要引入 UIKit.h。

现在看来，EOCPerson 类还好。其头文件引入了整个 Foundation 框架，不过这并没有问题。如果此类继承自 Foundation 框架中的某个类，那么 EOCPerson 类的使用者（consumer）可能会用到其基类中的许多内容。继承自 UIViewController 的那些类也是如此，其使用者可能会用到 UIKit 中的大部分内容。

过段时间，你可能又创建了一个名为 EOCEmployer 的新类，然后可能觉得每个 EOCPerson 实例都应该有一个 EOCEmployer。于是，直接为其添加一项属性：

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

```
@interface EOCPerson : NSObject
@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;
@property (nonatomic, strong) EOCEmployer *employer;
@end
```

然而这么做有个问题，就是在编译引入了 EOCPerson.h 的文件时，EOCEmployer 类并不可见。不便强迫开发者在引入 EOCPerson.h 时必须一并引入 EOCEmployer.h，所以，常见的办法是在 EOCPerson.h 中加入下面这行：

```
#import "EOCEmployer.h"
```

这种办法可行，但是不够优雅。在编译一个使用了 EOCPerson 类的文件时，不需要知道 EOCEmployer 类的全部细节，只需要知道有一个类名叫 EOCEmployer 就好。所幸有个办法能把这一情况告诉编译器：

```
@class EOCEmployer;
```

这叫做“向前声明”（forward declaring）该类。现在 EOCPerson 的头文件变成了这样：

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

@class EOCEmployer;

@interface EOCPerson : NSObject
@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;
@property (nonatomic, strong) EOCEmployer *employer;
@end
```

EOCPerson 类的实现文件则需引入 EOCEmployer 类的头文件，因为若要使用后者，则必须知道其所有接口细节。于是，实现文件就是：

```
// EOCPerson.m
#import "EOCPerson.h"
#import "EOCEmployer.h"

@implementation EOCPerson
// Implementation of methods
@end
```

将引入头文件的时机尽量延后，只在确有需要时才引入，这样就可以减少类的使用者所需引入的头文件数量。假设本例把 EOCEmployer.h 引入到 EOCPerson.h，那么只要引入 EOCPerson.h，就会一并引入 EOCEmployer.h 的所有内容。此过程若持续下去，则要引入许多根本用不到的内容，这当然会增加编译时间。

向前声明也解决了两个类互相引用的问题。假设要为 EOCEmployer 类加入新增及删除雇员的方法，那么其头文件中会加入下述定义：

```
- (void)addEmployee:(EOCPerson*)person;
- (void)removeEmployee:(EOCPerson*)person;
```

此时，若要编译 EOCEmployer，则编译器必须知道 EOCPerson 这个类，而要编译 EOCPerson，则又必须知道 EOCEmployer。如果在各自头文件中引入对方的头文件，则会导致“循环引用”（chicken-and-egg situation）。当解析其中一个头文件时，编译器会发现它引入了另一个头文件，而那个头文件又回过头来引用第一个头文件。使用 #import 而非 #include 指令虽然不会导致死循环，但却这意味着两个类里有一个无法被正确编译。如果不信的话，读者可以自己试试。

但是有时候必须要在头文件中引入其他头文件。如果你写的类继承自某个超类，则必须引入定义那个超类的头文件。同理，如果要声明你写的类遵从某个协议（protocol），那么该协议必须有完整定义，且不能使用向前声明。向前声明只能告诉编译器有某个协议，而此时编译器却要知道该协议中定义的方法。

例如，要从图形类中继承一个矩形类，且令其遵循绘制协议：

```
// EOCRectangle.h
#import "EOCShape.h"
#import "EOCDrawable.h"

@interface EOCRectangle : EOCShape<EOCDrawable>
@property (nonatomic, assign) float width;
@property (nonatomic, assign) float height;
@end
```

第二条 #import 是难免的。鉴于此，最好是把协议单独放在一个头文件中。要是把 EOCDrawable 协议放在了某个大的头文件里，那么只要引入此协议，就必定会引入那个头文件中的全部内容，如此一来，就像上面说的那样，会产生相互依赖问题，而且还会增加编译时间。

然而有些协议，例如“委托协议”（delegate protocol，参见第 23 条），就不用单独写一个头文件了。在那种情况下，协议只有与接受协议委托的类放在一起定义才有意义。此时最好能在实现文件中声明此类实现了该委托协议，并把这段实现代码放在“ class-continuation 分类”（class-continuation category，参见第 27 条）里。这样的话，只要在实现文件中引入包含委托协议的头文件即可，而不需将其放在公共头文件（public header file）里。

每次在头文件中引入其他头文件之前，都要先问问自己这样做是否确有必要。如果可以用向前声明取代引入，那么就不要引入。若因为要实现属性、实例变量或者要遵循协议而必须引入头文件，则应尽量将其移至“ class-continuation 分类”中（参见第 27 条）。这样做不仅可以缩减编译时间，而且还能降低彼此依赖程度。若是依赖关系过于复杂，则会给维护带来麻烦，而且，如果只想把代码的某个部分开放为公共 API 的话，太复杂的依赖关系也会出

问题。

要点

- 除非确有必要，否则不要引入头文件。一般来说，应在某个类的头文件中使用向前声明来提及别的类，并在实现文件中引入那些类的头文件。这样做可以尽量降低类之间的耦合（coupling）。
- 有时无法使用向前声明，比如要声明某个类遵循一项协议。这种情况下，尽量把“该类遵循某协议”的这条声明移至“ class-continuation 分类”中。如果不行的话，就把协议单独放在一个头文件中，然后将其引入。

第 3 条：多用字面量语法，少用与之等价的方法

编写 Objective-C 程序时，总会用到某几个类，它们属于 Foundation 框架。虽然从技术上来说，不用 Foundation 框架也能写出 Objective-C 代码，但实际上却经常要用到此框架。这几个类是 NSString、NSNumber、NSArray、NSDictionary。从类名上即可看出各自所表示的数据结构。

Objective-C 以语法繁杂而著称。事实上的确是这样。不过，从 Objective-C 1.0 起，有一种非常简单的方式能创建 NSString 对象。这就是“字符串字面量”(string literal)，其语法如下：

```
NSString *someString = @"Effective Objective-C 2.0";
```

如果不用这种语法的话，就要以常见的 alloc 及 init 方法来分配并初始化 NSString 对象了。在版本较新的编译器中，也能用这种字面量语法来声明 NSNumber、NSArray、NSDictionary 类的实例。使用字面量语法（literal syntax）可以缩减源代码长度，使其更为易读。

字面数值

有时需要把整数、浮点数、布尔值封入 Objective-C 对象中。这种情况下可以用 NSNumber 类，该类可处理多种类型的数值。若是不用字面量，那么就需要按下述方式创建实例：

```
NSNumber *someNumber = [NSNumber numberWithInt:1];
```

上面这行代码创建了一个数字，将其值设为整数 1。然而使用字面量能令代码更为整洁：

```
NSNumber *someNumber = @1;
```

大家可以看到，字面量语法更为精简。不过它还有很多好处。能够以 NSNumber 实例表示的所有数据类型都可使用该语法。例如：

```
NSNumber *intNumber = @1;
NSNumber *floatNumber = @2.5f;
```

```
NSNumber *doubleNumber = @3.14159;
NSNumber *boolNumber = @YES;
NSNumber *charNumber = @'a';
```

字面量语法也适用于下述表达式：

```
int x = 5;
float y = 6.32f;
NSNumber *expressionNumber = @(x * y);
```

以字面量来表示数值十分有用。这样做可以令 NSNumber 对象变得整洁，因为声明中只包含数值，而没有多余的语法成分。

字面量数组

数组是常用的数据结构。如果不使用字面量语法，那么就要这样来创建数组：

```
NSArray *animals =
    [NSArray arrayWithObjects:@"cat", @"dog",
                              @"mouse", @"badger", nil];
```

而使用字面量语法来创建则是：

```
NSArray *animals = @[@"cat", @"dog", @"mouse", @"badger"];
```

上面这种做法不仅简单，而且还利于操作数组。数组的常见操作就是取某个下标所对应的对象，这用字面量来做更为容易。如果不用字面量，那么通常会用“ objectAtIndex:”方法：

```
NSString *dog = [animals objectAtIndex:1];
```

若使用字面量，则是：

```
NSString *dog = animals[1];
```

这也叫做“取下标”操作（ subscripting），与使用字面量语法的其他情况一样，这种方式也更为简洁、更易理解，而且与其他语言中依下标来访问数组元素时所用的语法类似。

不过，用字面量语法创建数组时要注意，若数组元素对象中有 nil，则会抛出异常，因为字面量语法实际上只是一种“语法糖”(syntactic sugar) ⊖ ，其效果等于是先创建了一个数组，然后把方括号内的所有对象都加到这个数组中。抛出的异常会是这样：

```
*** Terminating app due to uncaught exception
'NSInvalidArgumentException', reason: '***
-[__NSPlaceholderArray initWithObjects:count:]: attempt to
insert nil object from objects[0]'
```

⊖ 也称“糖衣语法”，是指计算机语言中与另外一套语法等效但是开发者用起来却更加方便的语法。语法糖可令程序更易读，减少代码出错机率。详情参见：http://zh.wikipedia.org/wiki/ 语法糖。——译者注

在改用字面量语法来创建数组时就会遇到这个问题。下面这段代码分别以两种语法创建数组：

```
id object1 = /* ... */;
id object2 = /* ... */;
id object3 = /* ... */;

NSArray *arrayA = [NSArray array WithObjects:
                        object1, object2, object3, nil];
NSArray *arrayB = @[object1, object2, object3];
```

大家想想：如果 object1 与 object3 都指向了有效的 Objective-C 对象，而 object2 是 nil，那么会出现什么情况呢？按字面量语法创建数组 arrayB 时会抛出异常。arrayA 虽然能创建出来，但是其中却只含有 object1 一个对象。原因在于，“arrayWithObjects:”方法会依次处理各个参数，直到发现 nil 为止，由于 object2 是 nil，所以该方法会提前结束。

这个微妙的差别表明，使用字面量语法更为安全。抛出异常令应用程序终止执行，这比创建好数组之后才发现元素个数少了要好。向数组中插入 nil 通常说明程序有错，而通过异常可以更快地发现这个错误。

字面量字典

“字典”（Dictionary）是一种映射型数据结构，可向其中添加键值对。与数组一样，Objective-C 代码也经常用到字典。其创建方式如下：

```
NSDictionary *personData =
    [NSDictionarydictionaryWithObjectsAndKeys:
        @"Matt", @"firstName",
        @"Galloway", @"lastName",
        [NSNumber numberWithInt:28], @"age",
        nil];
```

这样写令人困惑，因为其顺序是＜对象＞，＜键＞，＜对象＞，＜键＞。这与通常理解的顺序相反，我们一般认为是把“键”映射到“对象”。因此，这种写法不容易读懂。如果改用字面量语法，就清晰多了：

```
NSDictionary *personData =
    @{@"firstName" : @"Matt",
      @"lastName" : @"Galloway",
      @"age" : @28};
```

上面这种写法更简明，而且键出现在对象之前，理解起来较顺畅。此范例代码还说明了使用字面量数值的好处。字典中的对象和键必须都是 Objective-C 对象，所以不能把整数 28 直接放进去，而要将其封装在 NSNumber 实例中才行。使用字面量语法很容易就能做到这一点，只需给数字前加一个 @ 字符即可。

与数组一样，用字面量语法创建字典时也有个问题，那就是一旦有值为 nil，便会抛出

异常。不过基于同样的原因，这也是个好事。假如在创建字典时不小心用了空值对象，那么“ dictionaryWithObjectsAndKeys:”方法就会在首个 nil 之前停下，并抛出异常，这有助于查错。

字典也可以像数组那样用字面量语法访问。按照特定键访问其值的传统做法是：

```
NSString *lastName = [personData objectForKey:@"lastName"];
```

与之等效的字面量语法则是：

```
NSString *lastName = personData[@"lastName"];
```

这样写也省去了冗赘的语法，令此行代码简单易读。

可变数组与字典

通过取下标操作，可以访问数组中某个下标或字典中某个键所对应的元素。如果数组与字典对象是可变的（mutable），那么也能通过下标修改其中的元素值。修改可变数组与字典内容的标准做法是：

```
[mutableArray replaceObjectAtIndex:1 withObject:@"dog"];
[mutableDictionary setObject:@"Galloway"forKey:@"lastName"];
```

若换用取下标操作来写，则是：

```
mutableArray[1] = @"dog";
mutableDictionary[@"lastName"] = @"Galloway";
```

局限性

字面量语法有个小小的限制，就是除了字符串以外，所创建出来的对象必须属于 Foundation 框架才行。如果自定义了这些类的子类，则无法用字面量语法创建其对象。要想创建自定义子类的实例，必须采用“非字面量语法”（nonliteral syntax）。然而，由于 NSArray、NSDictionary、NSNumber 都是业已定型的“子族”（class cluster，参见第 9 条），因此很少有人会从其中自定义子类，真要那样做也比较麻烦。而且一般来说，标准的实现已经很好了，无须再改动。创建字符串时可以使用自定义的子类，然而必须要修改编译器的选项才行。除非你明白这样做的后果，否则不鼓励使用此选项，用 NSString 就足够了。

使用字面量语法创建出来的字符串、数组、字典对象都是不可变的（ immutable）。若想要可变版本的对象，则需复制一份：

```
NSMutableArray *mutable = [@[@1, @2, @3, @4, @5]mutableCopy];
```

这么做会多调用一个方法，而且还要再创建一个对象，不过使用字面量语法所带来的好处还是多于上述缺点的。

要点

- 应该使用字面量语法来创建字符串、数值、数组、字典。与创建此类对象的常规方法相比，这么做更加简明扼要。
- 应该通过取下标操作来访问数组下标或字典中的键所对应的元素。
- 用字面量语法创建数组或字典时，若值中有 nil，则会抛出异常。因此，务必确保值里不含 nil。

第 4 条：多用类型常量，少用 #define 预处理指令

编写代码时经常要定义常量。例如，要写一个 UI 视图类，此视图显示出来之后就播放动画，然后消失。你可能想把播放动画的时间提取为常量。掌握了 Objective-C 与其 C 语言基础的人，也许会用这种方法来做：

```
#define ANIMATION_DURATION 0.3
```

上述预处理指令会把源代码中的 ANIMATION_DURATION 字符串替换为 0.3。这可能就是你想要的效果，不过这样定义出来的常量没有类型信息。“持续”（duration）这个词看上去应该与时间有关，但是代码中又未明确指出。此外，预处理过程会把碰到的所有 ANIMATION_DURATION 一律替换成 0.3，这样的话，假设此指令声明在某个头文件中，那么所有引入了这个头文件的代码，其 ANIMATION_DURATION 都会被替换。

要想解决此问题，应该设法利用编译器的某些特性才对。有个办法比用预处理指令来定义常量更好。比方说，下面这行代码就定义了一个类型为 NSTimeInterval 的常量：

```
static const NSTimeInterval kAnimationDuration = 0.3;
```

请注意，用此方式定义的常量包含类型信息，其好处是清楚地描述了常量的含义。由此可知该常量类型为 NSTimeInterval，这有助于为其编写开发文档。如果要定义许多常量，那么这种方式能令稍后阅读代码的人更易理解其意图。

还要注意常量名称。常用的命名法是：若常量局限于某“编译单元”（translation unit，也就是“实现文件”，implementation file）之内，则在前面加字母 k；若常量在类之外可见，则通常以类名为前缀。第 19 条详解了命名习惯（naming convention）。

定义常量的位置很重要。我们总喜欢在头文件里声明预处理指令，这样做真的很糟糕，当常量名称有可能互相冲突时更是如此。例如，ANIMATION_DURATION 这个常量名就不该用在头文件中，因为所有引入了这份头文件的其他文件中都会出现这个名字。其实就连用 static const 定义的那个常量也不应出现在头文件里。因为 Objective-C 没有“名称空间”（namespace）这一概念，所以那样做等于声明了一个名叫 kAnimationDuration 的全局变量。此名称应该加上前缀，以表明其所属的类，例如可改为 EOCViewClassAnimationDuration。本书第 19 条中深入讲解了一套清晰的命名方案。

若不打算公开某个常量，则应将其定义在使用该常量的实现文件里。比方说，要开发一个使用 UIKit 框架的 iOS 应用程序，其 UIView 子类中含有表示动画播放时间的常量，那么可以这样写：

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

@interface EOCAnimatedView : UIView
- (void)animate;
@end

// EOCAnimatedView.m
#import "EOCAnimatedView.h"

static const NSTimeInterval kAnimationDuration = 0.3;

@implementation EOCAnimatedView
- (void)animate {
    [UIViewanimateWithDuration:kAnimationDuration
                    animations:^(){
                        // Perform animations
                    }];
}
@end
```

变量一定要同时用 static 与 const 来声明。如果试图修改由 const 修饰符所声明的变量，那么编译器就会报错。在本例中，我们正是希望这样：因为动画播放时长为定值，所以不应修改。而 static 修饰符则意味着该变量仅在定义此变量的编译单元中可见。编译器每收到一个编译单元，就会输出一份“目标文件”（object file）。在 Objective-C 的语境下，“编译单元”一词通常指每个类的实现文件（以 .m 为后缀名）。因此，在上述范例代码中声明的 kAnimationDuration 变量，其作用域仅限于由 EOCAnimatedView.m 所生成的目标文件中。假如声明此变量时不加 static，则编译器会为它创建一个“外部符号”（external symbol）。此时若是另一个编译单元中也声明了同名变量，那么编译器就抛出一条错误消息：

```
duplicate symbol _kAnimationDuration in:
EOCAnimatedView.o
EOCOtherView.o
```

实际上，如果一个变量既声明为 static，又声明为 const，那么编译器根本不会创建符号，而是会像 #define 预处理指令一样，把所有遇到的变量都替换为常值。不过还是要记住：用这种方式定义的常量带有类型信息。

有时候需要对外公开某个常量。比方说，你可能要在类代码中调用 NSNotificationCenter 以通知他人。用一个对象来派发通知，令其他欲接收通知的对象向该对象注册，这样就能实现此功能了。派发通知时，需要使用字符串来表示此项通知的名称，而这个名字就可以声明

为一个外界可见的常值变量（constant variable）。这样的话，注册者无须知道实际字符串值，只需以常值变量来注册自己想要接收的通知即可。

此类常量需放在“全局符号表”（global symbol table）中，以便可以在定义该常量的编译单元之外使用。因此，其定义方式与上例演示的 static const 有所不同。应该这样来定义：

```
// In the header file
extern NSString *const EOCStringConstant;

// In the implementation file
NSString *const EOCStringConstant = @"VALUE";
```

这个常量在头文件中“声明”，且在实现文件中“定义”。注意 const 修饰符在常量类型中的位置。常量定义应从右至左解读，所以在本例中，EOCStringConstant 就是“一个常量，而这个常量是指针，指向 NSString 对象”。这与需求相符：我们不希望有人改变此指针常量，使其指向另一个 NSString 对象。

编译器看到头文件中的 extern 关键字，就能明白如何在引入此头文件的代码中处理该常量了。这个关键字是要告诉编译器，在全局符号表中将会有一个名叫 EOCStringConstant 的符号。也就是说，编译器无须查看其定义，即允许代码使用此常量。因为它知道，当链接成二进制文件之后，肯定能找到这个常量。

此类常量必须要定义，而且只能定义一次。通常将其定义在与声明该常量的头文件相关的实现文件里。由实现文件生成目标文件时，编译器会在“数据段”（data section）为字符串分配存储空间。链接器会把此目标文件与其他目标文件相链接，以生成最终的二进制文件。凡是用到 EOCStringConstant 这个全局符号的地方，链接器都能将其解析。

因为符号要放在全局符号表里，所以命名常量时需谨慎。例如，某应用程序中有个处理登录操作的类，在登录完成后会发出通知。派发通知所用的代码如下：

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

extern NSString *const EOCLoginManagerDidLoginNotification;

@interface EOCLoginManager : NSObject
- (void)login;
@end

// EOCLoginManager.m
#import "EOCLoginManager.h"

NSString *const EOCLoginManagerDidLoginNotification =
    @"EOCLoginManagerDidLoginNotification";

@implementation EOCLoginManager
```

```
- (void)login {
    // Perform login asynchronously, then call 'p_didLogin'.
}

- (void)p_didLogin {
    [[NSNotificationCenter defaultCenter]
      postNotificationName:EOCLoginManagerDidLoginNotification
                    object:nil];
}
@end
```

注意常量的名字。为避免名称冲突，最好是用与之相关的类名做前缀。系统框架中一般都这样做。例如 UIKit 就按照这种方式来声明用作通知名称的全局常量。其中有类似 UIApplicationDidEnterBackgroundNotification 与 UIApplicationWillEnterForegroundNotification 这样的常量名。

其他类型的常量也是如此。假如要把前例中 EOCAnimatedView 类里的动画播放时长对外公布，那么可以这样声明：

```
// EOCAnimatedView.h
extern const NSTimeInterval EOCAnimatedViewAnimationDuration;

// EOCAnimatedView.m
const NSTimeInterval EOCAnimatedViewAnimationDuration = 0.3;
```

这样定义常量要优于使用 #define 预处理指令，因为编译器会确保常量值不变。一旦在 EOCAnimatedView.m 中定义好，即可随处使用。而采用预处理指令所定义的常量可能会无意中遭人修改，从而导致应用程序各个部分所使用的值互不相同。

总之，勿使用预处理指令定义常量，而应该借助编译器来确保常量正确，比方说可以在实现文件中用 static const 来声明常量，也可以声明一些全局常量。

要点

- 不要用预处理指令定义常量。这样定义出来的常量不含类型信息，编译器只是会在编译前据此执行查找与替换操作。即使有人重新定义了常量值，编译器也不会产生警告信息，这将导致应用程序中的常量值不一致。
- 在实现文件中使用 static const 来定义“只在编译单元内可见的常量”（translation-unit-specific constant）。由于此类常量不在全局符号表中，所以无须为其名称加前缀。
- 在头文件中使用 extern 来声明全局常量，并在相关实现文件中定义其值。这种常量要出现在全局符号表中，所以其名称应加以区隔，通常用与之相关的类名做前缀。

第 5 条：用枚举表示状态、选项、状态码

由于 Objective-C 基于 C 语言，所以 C 语言有的功能它都有。其中之一就是枚举类型：

enum。系统框架中频繁用到此类型，然而开发者容易忽视它。在以一系列常量来表示错误状态码或可组合的选项时，极宜使用枚举为其命名。由于 C++11 标准扩充了枚举的特性[⊖]，所以最新版系统框架使用了“强类型”（strong type）的枚举。没错，Objective-C 也能得益于 C++11 标准。

枚举只是一种常量命名方式。某个对象所经历的各种状态就可以定义为一个简单的枚举集（enumeration set）。比如说，可以用下列枚举表示“套接字连接”（socket connection）的状态：

```
enum EOCConnectionState {
    EOCConnectionStateDisconnected,
    EOCConnectionStateConnecting,
    EOCConnectionStateConnected,
};
```

由于每种状态都用一个便于理解的值来表示，所以这样写出来的代码更易读懂。编译器会为枚举分配一个独有的编号，从 0 开始，每个枚举递增 1。实现枚举所用的数据类型取决于编译器，不过其二进制位（bit）的个数必须能完全表示下枚举编号才行。在前例中，由于最大编号是 2，所以使用 1 个字节[⊜] 的 char 类型即可。

然而定义枚举变量的方式却不太简洁，要依如下语法编写：

```
enum EOCConnectionState state = EOCConnectionStateDisconnected;
```

若是每次不用敲入 enum 而只需写 EOCConnectionState 就好了。要想这样做，则需使用 typedef 关键字重新定义枚举类型：

```
enum EOCConnectionState {
    EOCConnectionStateDisconnected,
    EOCConnectionStateConnecting,
    EOCConnectionStateConnected,
};
typedef enum EOCConnectionState EOCConnectionState;
```

现在可以用简写的 EOCConnectionState 来代替完整的 enum EOCConnectionState 了：

```
EOCConnectionState state = EOCConnectionStateDisconnected;
```

C++11 标准修订了枚举的某些特性。其中一项改动是：可以指明用何种“底层数据类型”（underlying type）来保存枚举类型的变量。这样做的好处是，可以向前声明枚举变量了。若不指定底层数据类型，则无法向前声明枚举类型，因为编译器不清楚底层数据类型的大小，所以在用到此枚举类型时，也就不知道究竟该给变量分配多少空间。

指定底层数据类型所用的语法是：

```
enum EOCConnectionStateConnectionState : NSInteger { /* ... */ };
```

⊖ 详情可参阅：http://upto11blog.blogspot.com/2012/12/strong-enum-enum-class-strong-enum-enum.html。——译者注

⊜ 一个字节含 8 个二进制位，所以至多能表示可取 256 种（2^8 个）枚举（编号为 0 ~ 255）的枚举变量。——译者注

上面这行代码确保枚举的底层数据类型是 NSInteger。也可以在向前声明时指定底层数据类型：

```
enum EOCConnectionStateConnectionState : NSInteger;
```

还可以不使用编译器所分配的序号，而是手工指定某个枚举成员所对应的值。语法如下：

```
enum EOCConnectionStateConnectionState {
    EOCConnectionStateDisconnected = 1,
    EOCConnectionStateConnecting,
    EOCConnectionStateConnected,
};
```

上述代码把 EOCConnectionStateDisconnected 的值设为 1，而不使用编译器所分配的 0。如前所述，接下来几个枚举的值都会在上一个的基础上递增 1。比如说，EOCConnectionState-Connected 的值就是 3。

还有一种情况应该使用枚举类型，那就是定义选项的时候。若这些选项可以彼此组合，则更应如此。只要枚举定义得对，各选项之间就可通过“按位或操作符”（bitwise OR operator）来组合。例如，iOS UI 框架中有如下枚举类型，用来表示某个视图应该如何在水平或垂直方向上调整大小：

```
enum UIViewAutoresizing {
    UIViewAutoresizingNone                 = 0,
    UIViewAutoresizingFlexibleLeftMargin   = 1 << 0,
    UIViewAutoresizingFlexibleWidth        = 1 << 1,
    UIViewAutoresizingFlexibleRightMargin  = 1 << 2,
    UIViewAutoresizingFlexibleTopMargin    = 1 << 3,
    UIViewAutoresizingFlexibleHeight       = 1 << 4,
    UIViewAutoresizingFlexibleBottomMargin = 1 << 5,
}
```

每个选项均可启用或禁用，使用上述方式来定义枚举值即可保证这一点，因为在每个枚举值[⊖] 所对应的二进制表示中，只有 1 个二进制位的值是 1。用“按位或操作符”可组合多个选项，例如：UIViewAutoResizingFlexibleWidth| UIViewAutoresizingFlexibleHeight。图 1-2 列出了每个枚举成员的二进制值，并演示了刚才那两个枚举组合之后的值。用“按位与操作符”（bitwise AND operator）即可判断出是否已启用某个选项：

```
enum UIViewAutoresizing resizing =
    UIViewAutoresizingFlexibleWidth |
    UIViewAutoresizingFlexibleHeight;
if (resizing & UIViewAutoresizingFlexibleWidth) {
    // UIViewAutoresizingFlexibleWidth is set
}
```

⊖ UIViewAutoresizingNone 除外，它的值是 0，对应的二进制值也是 0，其中没有值为 1 的二进制位。——译者注

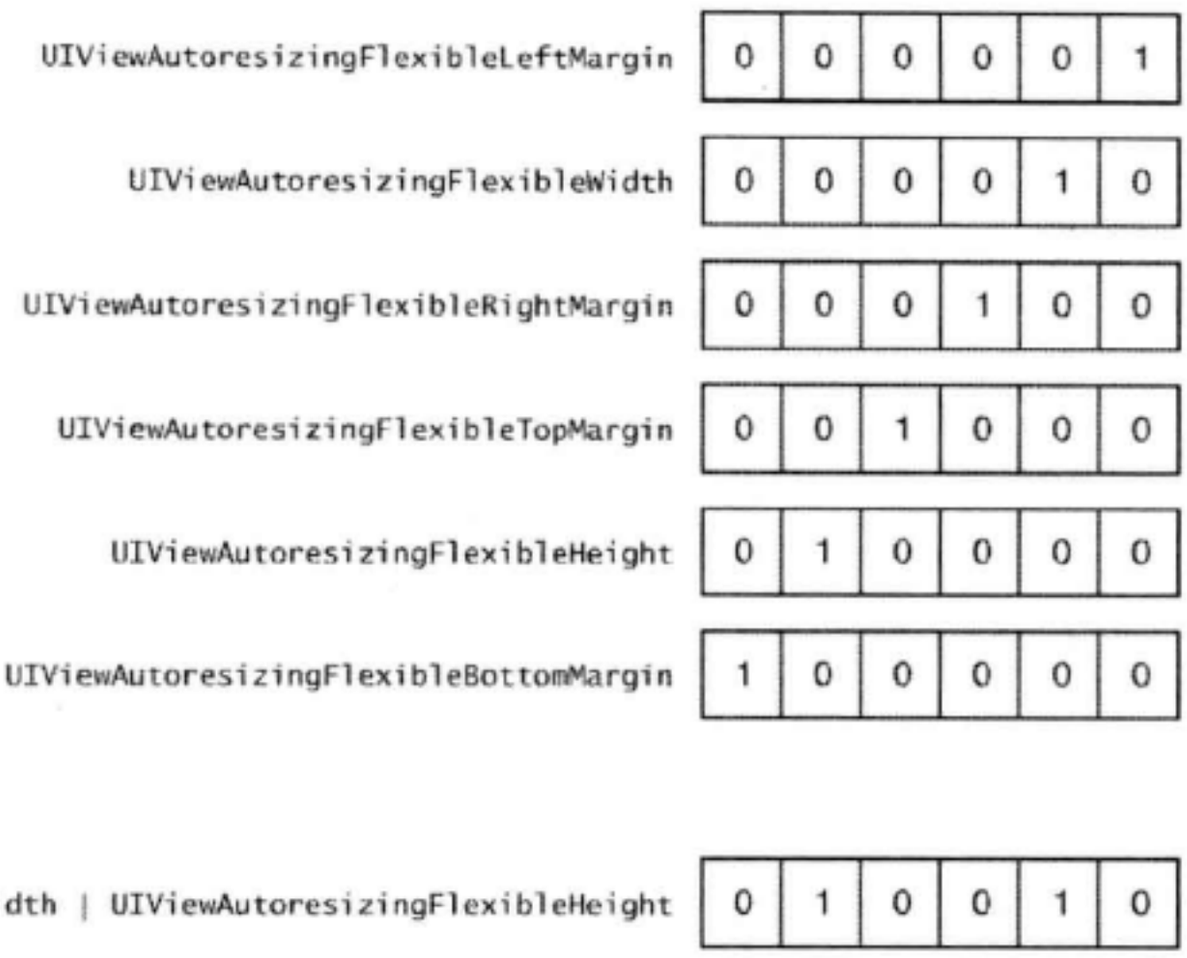

图 1-2　每个枚举值的二进制表示，以及对其中两个枚举值执行按位或操作之后的二进制值

系统库中频繁使用这个办法。iOS UI 框架中的 UIKit 里还有个例子，用枚举值告诉系统视图所支持的设备显示方向。这个枚举类型叫做 UIInterfaceOrientationMask，开发者需要实现一个名为 supportedInterfaceOrientations 的方法，将视图所支持的显示方向告诉系统：

```
- (NSUInteger)supportedInterfaceOrientations {
    return UIInterfaceOrientationMaskPortrait |
           UIInterfaceOrientationMaskLandscapeLeft;
}
```

Foundation 框架中定义了一些辅助的宏，用这些宏来定义枚举类型时，也可以指定用于保存枚举值的底层数据类型。这些宏具备向后兼容（backward compatibility）能力，如果目标平台的编译器支持新标准，那就使用新式语法，否则改用旧式语法。这些宏是用 #define 预处理指令来定义的，其中一个用于定义像 EOCConnectionState 这种普通的枚举类型，另一个用于定义像 UIViewAutoresizing 这种包含一系列选项的枚举类型，其用法如下：

```
typedef NS_ENUM(NSUInteger, EOCConnectionState) {
    EOCConnectionStateDisconnected,
    EOCConnectionStateConnecting,
    EOCConnectionStateConnected,
};
typedef NS_OPTIONS(NSUInteger, EOCPermittedDirection) {
    EOCPermittedDirectionUp    = 1 << 0,
    EOCPermittedDirectionDown  = 1 << 1,
    EOCPermittedDirectionLeft  = 1 << 2,
    EOCPermittedDirectionRight = 1 << 3,
};
```

这些宏的定义如下：

```
#if (__cplusplus && __cplusplus >= 201103L&&
        (__has_extension(cxx_strong_enums) ||
         __has_feature(objc_fixed_enum))
    ) ||
    (!__cplusplus && __has_feature(objc_fixed_enum))
    #define NS_ENUM(_type, _name)
            enum _name : _type _name; enum _name : _type
    #if (__cplusplus)
        #define NS_OPTIONS(_type, _name)
                type _name; enum : _type
    #else
        #define NS_OPTIONS(_type, _name)
                enum _name : _type _name; enum _name : _type
    #endif
#else
    #define NS_ENUM(_type, _name) _type _name; enum
    #define NS_OPTIONS(_type, _name) _type _name; enum
#endif
```

由于需要分别处理不同情况，所以上述代码用多种方式来定义这两个宏。第一个 #if 用于判断编译器是否支持新式枚举。其中所用的布尔逻辑看上去相当复杂，不过其意思就是想判断编译器是否支持新的枚举特性。如果不支持，那么就用老式语法来定义枚举。

如果支持新特性，那么用 NS_ENUM 宏所定义的枚举类型展开之后就是：

```
typedef enum EOCConnectionState : NSUInteger EOCConnectionState;
enum EOCConnectionState : NSUInteger {
    EOCConnectionStateDisconnected,
    EOCConnectionStateConnecting,
    EOCConnectionStateConnected,
};
```

根据是否要将代码按 C++ 模式编译，NS_OPTIONS 宏的定义方式也有所不同。如果不按 C++ 编译，那么其展开方式就和 NS_ENUM 相同。若按 C++ 编译，则展开后的代码略有不同。原因在于，用按位或运算来操作两个枚举值时，C++ 编译模式的处理办法与非 C++ 模式不一样。而上面已经提到了，作为选项的枚举值经常需要用按位或运算来组合。在用或运算操作两个枚举值时，C++ 认为运算结果的数据类型应该是枚举的底层数据类型，也就是 NSUInteger。而且 C++ 不允许将这个底层类型“隐式转换”（implicit cast）为枚举类型本身。我们用 EOCPermittedDirection 来演示一下，假设按 NS_ENUM 方式将其展开：

```
typedef enum EOCPermittedDirection : int EOCPermittedDirection;
enum EOCPermittedDirection : int {
    EOCPermittedDirectionUp    = 1 << 0,
    EOCPermittedDirectionDown  = 1 << 1,
    EOCPermittedDirectionLeft  = 1 << 2,
    EOCPermittedDirectionRight = 1 << 3,
};
```

然后考虑下列代码：

```
EOCPermittedDirection permittedDirections =
    EOCPermittedDirectionLeft | EOCPermittedDirectionUp;
```

若编译器按 C++ 模式编译（也可能是按 Objective-C++ 模式编译），则会给出下列错误信息：

```
error: cannot initialize a variable of type
'EOCPermittedDirection' with an rvalue of type 'int'
```

如果想编译这行代码，就要将按位或操作的结果显式转换（explicit cast）为 EOCPermittedDirection。所以，在 C++ 模式下应该用另一种方式定义 NS_OPTIONS 宏，以便省去类型转换操作。鉴于此，凡是需要以按位或操作来组合的枚举都应使用 NS_OPTIONS 定义。若是枚举不需要互相组合，则应使用 NS_ENUM 来定义。

能够用到枚举的情况还有很多。前面已经提到，枚举可以表示选项与状态，然而还有许多东西也能用枚举来表示。比如状态码就是个好例子。可以把逻辑含义相似的一组状态码放入同一个枚举集里，而不要用 #define 预处理指令或常量来定义。以枚举来表示样式（style）也很合宜。假如创建某个 UI 元素时可以使用不同的样式，那么在这种情况下就最应该把样式声明为枚举类型了。

最后再讲一种枚举的用法，就是在 switch 语句里。有时可以这样定义：

```
typedef NS_ENUM(NSUInteger, EOCConnectionState) {
    EOCConnectionStateDisconnected,
    EOCConnectionStateConnecting,
    EOCConnectionStateConnected,
};

switch (_currentState) {
    EOCConnectionStateDisconnected:
        // Handle disconnected state
        break;
    EOCConnectionStateConnecting:
        // Handle connecting state
        break;
    EOCConnectionStateConnected:
        // Handle connected state
        break;
}
```

我们总习惯在 switch 语句中加上 default 分支。然而，若是用枚举来定义状态机（state machine），则最好不要有 default 分支。这样的话，如果稍后又加了一种状态，那么编译器就会发出警告信息，提示新加入的状态并未在 switch 分支中处理。假如写上了 default 分支，那么它就会处理这个新状态，从而导致编译器不发警告信息。用 NS_ENUM 定义其他枚举类型时也要注意此问题。例如，在定义代表 UI 元素样式的枚举时，通常要确保 switch 语句能正确处理所有样式。

要点

- 应该用枚举来表示状态机的状态、传递给方法的选项以及状态码等值，给这些值起个易懂的名字。
- 如果把传递给某个方法的选项表示为枚举类型，而多个选项又可同时使用，那么就将各选项值定义为 2 的幂，以便通过按位或操作将其组合起来。
- 用 NS_ENUM 与 NS_OPTIONS 宏来定义枚举类型，并指明其底层数据类型。这样做可以确保枚举是用开发者所选的底层数据类型实现出来的，而不会采用编译器所选的类型。
- 在处理枚举类型的 switch 语句中不要实现 default 分支。这样的话，加入新枚举之后，编译器就会提示开发者：switch 语句并未处理所有枚举。

第2章

对象、消息、运行期

用 Objective-C 等面向对象语言编程时，“对象”（object）就是“基本构造单元”（building block），开发者可以通过对象来存储并传递数据。在对象之间传递数据并执行任务的过程就叫做“消息传递”（Messaging）。若想编写出高效且易维护的代码，就一定要熟悉这两个特性的工作原理。

当应用程序运行起来以后，为其提供相关支持的代码叫做“ Objective-C 运行期环境”（Objective-C runtime），它提供了一些使得对象之间能够传递消息的重要函数，并且包含创建类实例所用的全部逻辑。在理解了运行期环境中各个部分协同工作的原理之后，你的开发水平将会进一步提升。

第6条：理解“属性”这一概念

“属性”（property）是 Objecive-C 的一项特性，用于封装对象中的数据。Objective-C 对象通常会把其所需要的数据保存为各种实例变量。实例变量一般通过“存取方法”（access method）来访问。其中，“获取方法”（getter）用于读取变量值，而“设置方法”（setter）用于写入变量值。这个概念已经定型，并且经由“属性”这一特性而成为 Objective-C 2.0 的一部分，开发者可以令编译器自动编写与属性相关的存取方法。此特性引入了一种新的“点语法”（dot syntax），使开发者可以更为容易地依照类对象来访问存放于其中的数据。你也许已经使用过“属性”这个概念了，不过你未必知道其全部细节。而且，还有很多与属性有关的麻烦事。第6条将会告诉大家有哪些问题可以用属性来解决，并指出其中所体现出来的关键特性。

在描述个人信息的类中，也许会存放人名、生日、地址等内容。可以在类接口的 public 区段中声明一些实例变量：

```
@interface EOCPerson : NSObject {
@public
    NSString *_firstName;
    NSString *_lastName;
@private
```

```
    NSString *_someInternalData;
}
@end
```

原来编过 Java 或 C++ 程序的人应该比较熟悉这种写法，在这些语言中，可以定义实例变量的作用域。然而编写 Objective-C 代码时却很少这么做。这种写法的问题是：对象布局在编译期（compile time）就已经固定了。只要碰到访问 _firstName 变量的代码，编译器就把其替换为“偏移量”（offset），这个偏移量是“硬编码”（hardcode），表示该变量距离存放对象的内存区域的起始地址有多远。这样做目前来看没问题，但是如果又加了一个实例变量，那就麻烦了。比如说，假设在 _firstName 之前又多了一个实例变量：

```
@interface EOCPerson : NSObject {
@public
    NSDate *_dateOfBirth;
    NSString *_firstName;
    NSString *_lastName;
@private
    NSString *_someInternalData;
}
@end
```

原来表示 _firstName 的偏移量现在却指向 _dateOfBirth 了。把偏移量硬编码于其中的那些代码都会读取到错误的值。图 2-1 演示了此情况，请对比在类中加入 _dateOfBirth 这一实例变量之前与之后的内存布局，其中假设指针为 4 个字节。

	Person
+0	_firstName
+4	_lastName
+8	_someInternalData

	Person
+0	_dateOfBirth
+4	_firstName
+8	_lastName
+12	_someInternalData

图 2-1　在类中新增另一个实例变量前后的数据布局图

如果代码使用了编译期计算出来的偏移量，那么在修改类定义之后必须重新编译，否则就会出错。例如，某个代码库中的代码使用了一份旧的类定义。如果和其相链接的代码使用了新的类定义，那么运行时就会出现不兼容现象（incompatibility）。各种编程语言都有应对此问题的办法。Objective-C 的做法是，把实例变量当做一种存储偏移量所用的“特殊变量”（special variable），交由“类对象”（class object）保管（第 14 条详述了类对象）。偏移量会在运行期查找，如果类的定义变了，那么存储的偏移量也就变了，这样的话，无论何时访问实例变量，总能使用正确的偏移量。甚至可以在运行期向类中新增实例变量，这就是稳固的

“应用程序二进制接口”（Application Binary Interface，ABI）。ABI 定义了许多内容，其中一项就是生成代码时所应遵循的规范。有了这种“稳固的”（nonfragile）的 ABI，我们就可以在“class-continuation 分类”或实现文件中定义实例变量了。所以说，不一定要在接口中把全部实例变量都声明好，可以将某些变量从接口的 public 区段里移走，以便保护与类实现有关的内部信息。

这个问题还有一种解决办法，就是尽量不要直接访问实例变量，而应该通过存取方法来做。虽说属性最终还是得通过实例变量来实现，但它却提供了一种简洁的抽象机制。你可以自己编写存取方法，然而在正规的 Objective-C 编码风格中，存取方法有着严格的命名规范。正因为有了这种严格的命名规范，所以 Objective-C 这门语言才能根据名称自动创建出存取方法。这时 @property 语法就派上用场了。

在对象接口的定义中，可以使用属性，这是一种标准的写法，能够访问封装在对象里的数据。因此，也可以把属性当做一种简称，其意思是说：编译器会自动写出一套存取方法，用以访问给定类型中具有给定名称的变量。例如下面这个类：

```
@interface EOCPerson : NSObject
@property NSString *firstName;
@property NSString *lastName;
@end
```

对于该类的使用者来说，上述代码写出来的类与下面这种写法等效：

```
@interface EOCPerson : NSObject
- (NSString*)firstName;
- (void)setFirstName:(NSString*)firstName;
- (NSString*)lastName;
- (void)setLastName:(NSString*)lastName;
@end
```

要访问属性，可以使用“点语法”，在纯 C 中，如果想访问分配在栈上的 struct 结构体里面的成员，也需使用类似语法。编译器会把“点语法”转换为对存取方法的调用，使用“点语法”的效果与直接调用存取方法相同。因此，使用“点语法”和直接调用存取方法之间没有丝毫差别。通过下列范例代码可以看出，这两者等效：

```
EOCPerson *aPerson = [Person new];

aPerson.firstName = @"Bob"; // Same as:
[aPerson setFirstName:@"Bob"];

NSString *lastName = aPerson.lastName; // Same as:
NSString *lastName = [aPerson lastName];
```

然而属性还有更多优势。如果使用了属性的话，那么编译器就会自动编写访问这些属性所需的方法，此过程叫做“自动合成”（autosynthesis）。需要强调的是，这个过程由编译器在编译期执行，所以编辑器里看不到这些“合成方法”（synthesized method）的源代码。除

了生成方法代码之外，编译器还要自动向类中添加适当类型的实例变量，并且在属性名前面加下划线，以此作为实例变量的名字。在前例中，会生成两个实例变量，其名称分别为 _firstName 与 _lastName。也可以在类的实现代码里通过 @synthesize 语法来指定实例变量的名字：

```
@implementation EOCPerson
@synthesize firstName = _myFirstName;
@synthesize lastName = _myLastName;
@end
```

前述语法会将生成的实例变量命名为 _myFirstName 与 _myLastName，而不再使用默认的名字。一般情况下无须修改默认的实例变量名，但是如果你不喜欢以下划线来命名实例变量，那么可以用这个办法将其改为自己想要的名字。笔者还是推荐使用默认的命名方案，因为如果所有人都坚持这套方案，那么写出来的代码大家都能看得懂。

若不想令编译器自动合成存取方法，则可以自己实现。如果你只实现了其中一个存取方法，那么另外一个还是会由编译器来合成。还有一种办法能阻止编译器自动合成存取方法，就是使用 @dynamic 关键字，它会告诉编译器：不要自动创建实现属性所用的实例变量，也不要为其创建存取方法。而且，在编译访问属性的代码时，即使编译器发现没有定义存取方法，也不会报错，它相信这些方法能在运行期找到。比方说，如果从 CoreData 框架中的 NSManagedObject 类里继承了一个子类，那么就需要在运行期动态创建存取方法。继承 NSManagedObject 时之所以要这样做，是因为子类的某些属性不是实例变量，其数据来自后端的数据库中。例如：

```
@interface EOCPerson : NSManagedObject
@property NSString *firstName;
@property NSString *lastName;
@end

@implementation EOCPerson
@dynamic firstName, lastName;
@end
```

编译器不会为上面这个类自动合成存取方法或实例变量。如果用代码访问其中的属性，编译器也不会发出警示信息。

属性特质

使用属性时还有一个问题要注意，就是其各种特质（attribute）设定也会影响编译器所生成的存取方法。比如下面这个属性就指定了三项特质：

```
@property (nonatomic, readwrite, copy) NSString *firstName;
```

属性可以拥有的特质分为四类：

原子性

在默认情况下，由编译器所合成的方法会通过锁定机制确保其原子性（atomicity）[⊖]。如果属性具备 nonatomic 特质，则不使用同步锁。请注意，尽管没有名为“atomic”的特质（如果某属性不具备 nonatomic 特质，那它就是“原子的”（atomic）），但是仍然可以在属性特质中写明这一点，编译器不会报错。若是自己定义存取方法，那么就应该遵从与属性特质相符的原子性。

读 / 写权限

- 具备 **readwrite**（读写）特质的属性拥有“获取方法”（getter）与“设置方法”（setter）[⊜]。若该属性由 @synthesize 实现，则编译器会自动生成这两个方法。
- 具备 **readonly**（只读）特质的属性仅拥有获取方法，只有当该属性由 @synthesize 实现时，编译器才会为其合成获取方法。你可以用此特质把某个属性对外公开为只读属性，然后在“class-continuation 分类”中将其重新定义为读写属性。第 27 条详述了这种做法。

内存管理语义

属性用于封装数据，而数据则要有“具体的所有权语义”（concrete ownership semantic）。下面这一组特质仅会影响“设置方法”。例如，用“设置方法”设定一个新值时，它是应该“保留”（retain）[⊝] 此值呢，还是只将其赋给底层实例变量就好？编译器在合成存取方法时，要根据此特质来决定所生成的代码。如果自己编写存取方法，那么就必须同有关属性所具备的特质相符。

- **assign** “设置方法”只会执行针对“纯量类型”（scalar type，例如 CGFloat 或 NSInteger 等）的简单赋值操作。
- **strong** 此特质表明该属性定义了一种“拥有关系”（owning relationship）。为这种属性设置新值时，设置方法会先保留新值，并释放旧值，然后再将新值设置上去。
- **weak** 此特质表明该属性定义了一种“非拥有关系”（nonowning relationship）。为这种属性设置新值时，设置方法既不保留新值，也不释放旧值。此特质同 assign 类似，然而在属性所指的对象遭到摧毁时，属性值也会清空（nil out）。
- **unsafe_unretained** 此特质的语义和 assign 相同，但是它适用于“对象类型”（object type），该特质表达一种“非拥有关系”（“不保留”，unretained），当目标对象遭到摧毁时，属性值不会自动清空（“不安全”，unsafe），这一点与 weak 有区别。
- **copy** 此特质所表达的所属关系与 strong 类似。然而设置方法并不保留新值，而是

⊖ 在并发编程中，如果某操作具备整体性，也就是说，系统其他部分无法观察到其中间步骤所生成的临时结果，而只能看到操作前与操作后的结果，那么该操作就是“原子的”（atomic），或者说，该操作具备“原子性”（atomicity）。详情参见：http://en.wikipedia.org/wiki/Atomicity_(programming)。——译者注

⊜ 也叫做“获取器”、“设置器”。——译者注

⊝ 也叫做“保有”。——译者注

将其“拷贝”（copy）。当属性类型为NSString*时，经常用此特质来保护其封装性，因为传递给设置方法的新值有可能指向一个NSMutableString类的实例。这个类是NSString的子类，表示一种可以修改其值的字符串，此时若是不拷贝字符串，那么设置完属性之后，字符串的值就可能会在对象不知情的情况下遭人更改。所以，这时就要拷贝一份“不可变”（immutable）的字符串，确保对象中的字符串值不会无意间变动。只要实现属性所用的对象是“可变的”（mutable），就应该在设置新属性值时拷贝一份。

方法名

可通过如下特质来指定存取方法的方法名：

- **getter=<name>** 指定“获取方法”的方法名。如果某属性是Boolean型，而你想为其获取方法加上“is”前缀，那么就可以用这个办法来指定。比如说，在UISwitch类中，表示“开关”（switch）是否打开的属性就是这样定义的：

```
@property (nonatomic, getter=isOn) BOOL on;
```

- **setter=<name>** 指定“设置方法”的方法名。这种用法不太常见。

通过上述特质，可以微调由编译器所合成的存取方法。不过需要注意：若是自己来实现这些存取方法，那么应该保证其具备相关属性所声明的特质。比方说，如果将某个属性声明为copy，那么就应该在“设置方法”中拷贝相关对象，否则会误导该属性的使用者，而且，若是不遵从这一约定，还会令程序产生bug。

如果想在其他方法里设置属性值，那么同样要遵守属性定义中所宣称的语义。例如，我们扩充一下前面提到的EOCPerson类。由于字符串值可能会改变，所以要把相关属性的“内存管理语义”声明为copy。该类中新增了一个“初始化方法”（initializer）[⊖]，用于设置“名”（first name）和“姓”（last name）的初始值：

```
@interface EOCPerson : NSManagedObject

@property (copy) NSString *firstName;
@property (copy) NSString *lastName;

- (id)initWithFirstName:(NSString*)firstName
               lastName:(NSString*)lastName;
@end
```

在实现这个自定义的初始化方法时，一定要遵循属性定义中宣称的“copy”语义，因为“属性定义”就相当于“类”和“待设置的属性值”之间所达成的契约。初始化方法的实现代码可以这样写：

```
- (id)initWithFirstName:(NSString*)firstName
               lastName:(NSString*)lastName
```

⊖ 也叫做“初始化器”。——译者注

```
{
    if ((self = [super init])) {
        _firstName = [firstName copy];
        _lastName = [lastName copy];
    }
return self;
}
```

读者也许会问：为何不调用属性所对应的“设置方法”呢？如果用了“设置方法”的话，不是总能保证准确的语义吗？笔者在第 7 条中将会详细解释为什么决不应该在 init（或 dealloc）方法中调用存取方法。

要是看过第 18 条的话，你就会明白，应该尽量使用不可变的对象。如果将这一条套用到 EOCPerson 类身上，那就等于说，其两个属性都应该设为“只读”。用初始化方法设置好属性值之后，就不能再改变了。在本例中，仍需声明属性的“内存管理语义”。于是可以把属性的定义改成这样：

```
@property (copy, readonly) NSString *firstName;
@property (copy, readonly) NSString *lastName;
```

由于是只读属性，所以编译器不会为其创建对应的“设置方法”，即便如此，我们还是要写上这些属性的语义，以此表明初始化方法在设置这些属性值时所用的方式。要是不写明语义的话，该类的调用者就不知道初始化方法里会拷贝这些属性，他们有可能会在调用初始化方法之前自行拷贝属性值。这种操作是多余而且低效的。

atomic 与 nonatomic 的区别是什么呢？前面说过，具备 atomic 特质的获取方法会通过锁定机制来确保其操作的原子性。这也就是说，如果两个线程读写同一属性，那么不论何时，总能看到有效的属性值。若是不加锁的话（或者说使用 nonatomic 语义），那么当其中一个线程正在改写某属性值时，另外一个线程也许会突然闯入，把尚未修改好的属性值读取出来。发生这种情况时，线程读到的属性值可能不对。

如果开发过 iOS 程序，你就会发现，其中所有属性都声明为 nonatomic。这样做的历史原因是：在 iOS 中使用同步锁的开销较大，这会带来性能问题。一般情况下并不要求属性必须是“原子的”，因为这并不能保证“线程安全”（thread safety），若要实现“线程安全”的操作，还需采用更为深层的锁定机制才行。例如，一个线程在连续多次读取某属性值的过程中有别的线程在同时改写该值，那么即便将属性声明为 atomic，也还是会读到不同的属性值。因此，开发 iOS 程序时一般都会使用 nonatomic 属性。但是在开发 Mac OS X 程序时，使用 atomic 属性通常都不会有性能瓶颈。

要点

- 可以用 @property 语法来定义对象中所封装的数据。
- 通过“特质”来指定存储数据所需的正确语义。
- 在设置属性所对应的实例变量时，一定要遵从该属性所声明的语义。

■ 开发 iOS 程序时应该使用 nonatomic 属性，因为 atomic 属性会严重影响性能。

第 7 条：在对象内部尽量直接访问实例变量

在对象之外访问实例变量时，总是应该通过属性来做，然而在对象内部访问实例变量时又该如何呢？ Objective-C 的开发者们一直在激烈争论这个问题。有的人认为，无论什么情况，都应该通过属性来访问实例变量；也有人说，“通过属性访问”与“直接访问”这两种做法应该搭配着用。除了几种特殊情况之外，笔者强烈建议大家在读取实例变量的时候采用直接访问的形式，而在设置实例变量的时候通过属性来做。

请看下面这个类：

```
@interface EOCPerson : NSObject
@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;

// Convenience for firstName + " " + lastName:
- (NSString*)fullName;
- (void)setFullName:(NSString*)fullName;
@end
```

fullName 与 setFullName 这两个“便捷方法”可以这样来实现：

```
- (NSString*)fullName {
    return [NSString stringWithFormat:@"%@ %@",
            self.firstName, self.lastName];
}

/** The following assumes all full names have exactly 2
*  parts. The method could be rewritten to support more
*  exotic names.
*/
- (void)setFullName:(NSString*)fullName {
    NSArray *components =
        [fullName componentsSeparatedByString:@" "];
    self.firstName = [components objectAtIndex:0];
    self.lastName = [components objectAtIndex:1];
}
```

在 fullName 的获取方法与设置方法中，我们使用“点语法”，通过存取方法来访问相关实例变量。现在假设重写这两个方法，不经由存取方法，而是直接访问实例变量：

```
- (NSString*)fullName {
    return [NSStringstringWithFormat:@"%@ %@",
            _firstName, _lastName];
}

- (void)setFullName:(NSString*)fullName {
```

```
    NSArray *components =
        [fullName componentsSeparatedByString:@" "];
    _firstName = [components objectAtIndex:0];
    _lastName = [components objectAtIndex:1];
}
```

这两种写法有几个区别。

- 由于不经过 Objective-C 的“方法派发”（method dispatch，参见第 11 条）步骤，所以直接访问实例变量的速度当然比较快。在这种情况下，编译器所生成的代码会直接访问保存对象实例变量的那块内存。
- 直接访问实例变量时，不会调用其“设置方法”，这就绕过了为相关属性所定义的“内存管理语义”。比方说，如果在 ARC 下直接访问一个声明为 copy 的属性，那么并不会拷贝该属性，只会保留新值并释放旧值。
- 如果直接访问实例变量，那么不会触发“键值观测”(Key-Value Observing，KVO)[⊖] 通知。这样做是否会产生问题，还取决于具体的对象行为。
- 通过属性来访问有助于排查与之相关的错误，因为可以给“获取方法”和 / 或“设置方法”中新增“断点”(breakpoint)，监控该属性的调用者及其访问时机。

有一种合理的折中方案，那就是：在写入实例变量时，通过其“设置方法”来做，而在读取实例变量时，则直接访问之。此办法既能提高读取操作的速度，又能控制对属性的写入操作。之所以要通过“设置方法”来写入实例变量，其首要原因在于，这样做能够确保相关属性的“内存管理语义”得以贯彻。但是，选用这种做法时，需注意几个问题。

第一个要注意的地方就是，在初始化方法中应该如何设置属性值。这种情况下总是应该直接访问实例变量，因为子类可能会“覆写”（override）设置方法。假设 EOCPerson 有一个子类叫做 EOCSmithPerson，这个子类专门表示那些姓“ Smith”的人。该子类可能会覆写 lastName 属性所对应的设置方法：

```
- (void)setLastName:(NSString*)lastName {
    if (![lastName isEqualToString:@"Smith"]) {
        [NSException raise:NSInvalidArgumentException
                    format:@"Last name must be Smith"];
    }
    self.lastName = lastname;
}
```

在基类 EOCPerson 的默认初始化方法中，可能会将姓氏设为空字符串。此时若是通过“设置方法”来做，那么调用的将会是子类的设置方法，从而抛出异常。但是，某些情况下却又必须在初始化方法中调用设置方法：如果待初始化的实例变量声明在超类中，而我们又无法在子类中直接访问此实例变量的话，那么就需要调用“设置方法”了。

⊖ 一种通知机制，当某对象属性改变时，可通知其他对象。详情参见：http://developer.apple.com/library/ios/documentation/general/conceptual/DevPedia-CocoaCore/KVO.html。——译者注

另外一个要注意的问题是“惰性初始化”(lazy initialization)[⊖]。在这种情况下，必须通过“获取方法”来访问属性，否则，实例变量就永远不会初始化。比方说，EOCPerson类也许会用一个属性来表示人脑中的信息，这个属性所指代的对象相当复杂。由于此属性不常用，而且创建该属性的成本较高，所以，我们可能会在“获取方法”中对其执行惰性初始化：.

```
- (EOCBrain*)brain {
    if (!_brain) {
        _brain = [Brain new];
    }
    return _brain;
}
```

若没有调用“获取方法”就直接访问实例变量，则会看到尚未设置好的brain，所以说，如果使用了“惰性初始化”技术，那么必须通过存取方法来访问brain属性。

要点

- 在对象内部读取数据时，应该直接通过实例变量来读，而写入数据时，则应通过属性来写。
- 在初始化方法及dealloc方法中，总是应该直接通过实例变量来读写数据。
- 有时会使用惰性初始化技术配置某份数据，这种情况下，需要通过属性来读取数据。

第8条：理解“对象等同性”这一概念

根据“等同性”（equality）来比较对象是一个非常有用的功能。不过，按照==操作符比较出来的结果未必是我们想要的，因为该操作比较的是两个指针本身，而不是其所指的对象。应该使用NSObject协议中声明的“isEqual”:方法来判断两个对象的等同性。一般来说，两个类型不同的对象总是不相等的（unequal）。某些对象提供了特殊的“等同性判定方法”（equality-checking method），如果已经知道两个受测对象都属于同一个类，那么就可以使用这种方法。以下述代码为例：

```
NSString *foo = @"Badger 123";
NSString *bar = [NSStringstringWithFormat:@"Badger %i", 123];
BOOL equalA = (foo == bar); //< equalA = NO
BOOL equalB = [foo isEqual:bar]; //< equalB = YES
BOOL equalC = [foo isEqualToString:bar]; //< equalC = YES
```

大家可以看到==与等同性判断方法之间的差别。NSString类实现了一个自己独有的等同性判断方法，名叫“isEqualToString:”。传递给该方法的对象必须是NSString，否则结果未定义（undefined）。调用该方法比调用“isEqual:”方法快，后者还要执行额外的步骤，因

⊖ 也叫做“延迟初始化”。——译者注

为它不知道受测对象的类型。

NSObject 协议中有两个用于判断等同性的关键方法：

```
- (BOOL)isEqual:(id)object;
- (NSUInteger)hash;
```

NSObject 类对这两个方法的默认实现是：当且仅当其“指针值”(pointer value)[⊖]完全相等时，这两个对象才相等。若想在自定义的对象中正确覆写这些方法，就必须先理解其约定（contract）。如果“isEqual:”方法判定两个对象相等，那么其 hash 方法也必须返回同一个值。但是，如果两个对象的 hash 方法返回同一个值，那么“isEqual:”方法未必会认为两者相等。

比如有下面这个类：

```
@interface EOCPerson : NSObject
@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;
@property (nonatomic, assign) NSUInteger age;
@end
```

我们认为，如果两个 EOCPerson 的所有字段均相等，那么这两个对象就相等。于是“isEqual:”方法可以写成：

```
- (BOOL)isEqual:(id)object {
    if (self == object) return YES;
    if ([self class] != [object class]) return NO;

    EOCPerson *otherPerson = (EOCPerson*)object;
    if (![_firstName isEqualToString:otherPerson.firstName])
        return NO;
    if (![_lastName isEqualToString:otherPerson.lastName])
        return NO;
    if (_age != otherPerson.age)
        return NO;
    return YES;
}
```

首先，直接判断两个指针是否相等。若相等，则其均指向同一对象，所以受测的对象也必定相等。接下来，比较两对象所属的类。若不属于同一个类，则两对象不相等。EOCPerson 对象当然不可能与 EOCDog 对象相等。不过，有时我们可能认为：一个 EOCPerson 实例可以与其子类（比如 EOCSmithPerson）实例相等。在继承体系（inheritance hierarchy）中判断等同性时，经常遭遇此类问题。所以实现“isEqual:”方法时要考虑到这种情况。最后，检测每个属性是否相等。只要其中有不相等的属性，就判定两对象不等，否则两对象相等。

⊖ 可以理解为“内存地址”。——译者注

接下来该实现hash方法了。回想一下，根据等同性约定：若两对象相等，则其哈希码(hash)[⊖]也相等，但是两个哈希码相同的对象却未必相等。这是能否正确覆写“isEqual:”方法的关键所在。下面这种写法完全可行：

```
- (NSUInteger)hash {
    return 1337;
}
```

不过若是这么写的话，在collection中使用这种对象将产生性能问题，因为collection在检索哈希表(hash table)时，会用对象的哈希码做索引。假如某个collection是用set[⊜]实现的，那么set可能会根据哈希码把对象分装到不同的数组[⊜]中。在向set中添加新对象时，要根据其哈希码找到与之相关的那个数组，依次检查其中各个元素，看数组中已有的对象是否和将要添加的新对象相等。如果相等，那就说明要添加的对象已经在set里面了。由此可知，如果令每个对象都返回相同的哈希码，那么在set中已有1 000 000个对象的情况下，若是继续向其中添加对象，则需将这1 000 000个对象全部扫描一遍。

hash方法也可以这样来实现：

```
- (NSUInteger)hash {
    NSString *stringToHash =
        [NSStringstringWithFormat:@"%@:%@:%i",
            _firstName, _lastName, _age];
    return [stringToHash hash];
}
```

这次所用的办法是将NSString对象中的属性都塞入另一个字符串中，然后令hash方法返回该字符串的哈希码。这么做符合约定，因为两个相等的EOCPerson对象总会返回相同的哈希码。但是这样做还需负担创建字符串的开销，所以比返回单一值要慢。把这种对象添加到collection中时，也会产生性能问题，因为要想添加，必须先计算其哈希码。

再来看最后一种计算哈希码的办法：

```
- (NSUInteger)hash {
    NSUInteger firstNameHash = [_firstName hash];
    NSUInteger lastNameHash = [_lastName hash];
    NSUInteger ageHash = _age;
    return firstNameHash ^ lastNameHash ^ ageHash;
}
```

这种做法既能保持较高效率，又能使生成的哈希码至少位于一定范围之内，而不会过于频繁地重复。当然，此算法生成的哈希码还是会碰撞（collision），不过至少可以保证哈希码

⊖ “hash”一词也叫做“杂凑”、“散列”——译者注

⊜ “collection”与“set”在中文里都叫做“集合”，前者是Array、Dictionary、Set等数据结构的总称。译文为避免混淆，保留这两个词的英文写法。——译者注

⊜ 这种数组在后文中也称为“箱子”(bin)。——译者注

有多种可能的取值。编写 hash 方法时，应该用当前的对象做做实验，以便在减少碰撞频度与降低运算复杂程度之间取舍。

特定类所具有的等同性判定方法

除了刚才提到的 NSString 之外，NSArray 与 NSDictionary 类也具有特殊的等同性判定方法，前者名为“ isEqualToArray:”，后者名为“ isEqualToDictionary:”。如果和其相比较的对象不是数组或字典，那么这两个方法会各自抛出异常。由于 Objective-C 在编译期不做强类型检查（ strong type checking），这样容易不小心传入类型错误的对象，因此开发者应该保证所传对象的类型是正确的。

如果经常需要判断等同性，那么可能会自己来创建等同性判定方法，因为无须检测参数类型，所以能大大提升检测速度。自己来编写判定方法的另一个原因是，我们想令代码看上去更美观、更易读，此动机与 NSString 类“ isEqualToString:”方法的创建缘由相似，纯粹为了装点门面。使用此种判定方法编出来的代码更容易读懂，而且不用再检查两个受测对象的类型了。

在编写判定方法时，也应一并覆写“ isEqual:”方法。后者的常见实现方式为：如果受测的参数与接收该消息的对象都属于同一个类，那么就调用自己编写的判定方法，否则就交由超类来判断。例如，在 EOCPerson 类中可以实现如下两个方法：

```
- (BOOL)isEqualToPerson:(EOCPerson*)otherPerson {
    if (self == object) return YES;

    if (![_firstName isEqualToString:otherPerson.firstName])
        return NO;
    if (![_lastName isEqualToString:otherPerson.lastName])
        return NO;
    if (_age != otherPerson.age)
        return NO;
    return YES;
}

- (BOOL)isEqual:(id)object {
    if ([self class] == [object class]) {
        return [self isEqualToPerson:(EOCPerson*)object];
    } else {
        return [super isEqual:object];
    }
}
```

等同性判定的执行深度

创建等同性判定方法时，需要决定是根据整个对象来判断等同性，还是仅根据其中几个字段来判断。NSArray 的检测方式为先看两个数组所含对象个数是否相同，若相同，则在每

个对应位置的两个对象身上调用其“ isEqual:”方法。如果对应位置上的对象均相等，那么这两个数组就相等，这叫做“深度等同性判定”（deep equality）。不过有时候无须将所有数据逐个比较，只根据其中部分数据即可判明二者是否等同。

比方说，我们假设 EOCPerson 类的实例是根据数据库里的数据创建而来，那么其中就可能会含有另外一个属性，此属性是“唯一标识符”(unique identifier)，在数据库中用作“主键”(primary key)：

```
@property NSUInteger identifier;
```

在这种情况下，我们也许只会根据标识符来判断等同性，尤其是在此属性声明为 readonly 时更应该如此。因为只要两者标识符相同，就肯定表示同一个对象，因而必然相等。这样的话，无须逐个比较 EOCPerson 对象的每条数据，只要标识符相同，就说明这两个对象就是由同一个数据源所创建的，据此我们能够断定，其余数据也必然相同。

是否需要在等同性判定方法中检测全部字段取决于受测对象。只有类的编写者才可以确定两个对象实例在何种情况下应判定为相等。

容器中可变类的等同性

还有一种情况一定要注意，就是在容器中放入可变类对象的时候。把某个对象放入 collection 之后，就不应再改变其哈希码了。前面解释过，collection 会把各个对象按照其哈希码分装到不同的“箱子数组”中。如果某对象在放入“箱子”之后哈希码又变了，那么其现在所处的这个箱子对它来说就是“错误”的。要想解决这个问题，需要确保哈希码不是根据对象的“可变部分”（mutable portion）计算出来的，或是保证放入 collection 之后就不再改变对象内容了。笔者将在第 18 条中解释为何要将对象做成“不可变的”(immutable)。这里先举个例子，此例能很好地说明其中缘由。

用一个 NSMutableSet 与几个 NSMutableArray 对象测试一下，就能发现这个问题了。首先把一个数组加入 set 中：

```
NSMutableSet *set = [NSMutableSetnew];

NSMutableArray *arrayA = [@[@1, @2]mutableCopy];
[set addObject:arrayA];
NSLog(@"set = %@", set);
// Output: set = {((1,2))}
```

现在 set 里含有一个数组对象，数组中包含两个对象。再向 set 中加入一个数组，此数组与前一个数组所含的对象相同，顺序也相同，于是，待加入的数组与 set 中已有的数组是相等的：

```
NSMutableArray *arrayB = [@[@1, @2]mutableCopy];
[set addObject:arrayB];
NSLog(@"set = %@", set);
// Output: set = {((1,2))}
```

此时 set 里仍然只有一个对象：因为刚才要加入的那个数组对象和 set 中已有的数组对象相等，所以 set 并不会改变。这次我们来添加一个和 set 中已有对象不同的数组：

```
NSMutableArray *arrayC = [@[@1]mutableCopy];
[set addObject:arrayC];
NSLog(@"set = %@", set);
// Output: set = {((1),(1,2))}
```

正如大家所料，由于 arrayC 与 set 里已有的对象不相等，所以现在 set 里有两个数组了：其中一个是最早加入的，另一个是刚才新添加的。最后，我们改变 arrayC 的内容，令其和最早加入 set 的那个数组相等：

```
[arrayC addObject:@2];
NSLog(@"set = %@", set);
// Output: set = {((1,2),(1,2))}
```

set 中居然可以包含两个彼此相等的数组！根据 set 的语义是不允许出现这种情况的，然而现在却无法保证这一点了，因为我们修改了 set 中已有的对象。若是拷贝此 set，那就更糟糕了：

```
NSSet *setB = [set copy];
NSLog(@"setB = %@", setB);
// Output: setB = {((1,2))}
```

复制过的 set 中又只剩一个对象了，此 set 看上去好像是由一个空 set 开始、通过逐个向其中添加新对象而创建出来的。这可能符合你的需求，也可能不符合。有的开发者也许想要忽略 set 中的错误，“照原样”（verbatim）复制一个新的出来，还有的开发者则会认为这样做挺合适的。这两种拷贝算法都说得通，于是就进一步印证了刚才提到的那个问题：如果把某对象放入 set 之后又修改其内容，那么后面的行为将很难预料。

举这个例子是想提醒大家：把某对象放入 collection 之后改变其内容将会造成何种后果。笔者并不是说绝对不能这么做，而是说如果真要这么做，那就得注意其隐患，并用相应的代码处理可能发生的问题。

要点

- 若想检测对象的等同性，请提供“isEqual:”与 hash 方法。
- 相同的对象必须具有相同的哈希码，但是两个哈希码相同的对象却未必相同。
- 不要盲目地逐个检测每条属性，而是应该依照具体需求来制定检测方案。
- 编写 hash 方法时，应该使用计算速度快而且哈希码碰撞几率低的算法。

第 9 条：以“类族模式”隐藏实现细节

“类族”（class cluster）[⊖] 是一种很有用的模式（pattern），可以隐藏“抽象基类”（abstract

⊖ 也叫做“类簇”。——译者注

base class）背后的实现细节。Objective-C 的系统框架中普遍使用此模式。比如，iOS 的用户界面框架（user interface framework）UIKit 中就有一个名为 UIButton 的类。想创建按钮，需要调用下面这个“类方法”(class method)[⊖]：

```
+ (UIButton*)buttonWithType:(UIButtonType)type;
```

该方法所返回的对象，其类型取决于传入的按钮类型（button type）。然而，不管返回什么类型的对象，它们都继承自同一个基类：UIButton。这么做的意义在于：UIButton 类的使用者无须关心创建出来的按钮具体属于哪个子类，也不用考虑按钮的绘制方式等实现细节。使用者只需明白如何创建按钮，如何设置像“标题”（title）这样的属性，如何增加触摸动作的目标对象等问题就好。

回到开头说的那个问题上，我们可以把各种按钮的绘制逻辑都放在一个类里，并根据按钮类型来切换：

```
- (void)drawRect:(CGRect)rect {
    if (_type == TypeA) {
        // Draw TypeA button
    } else if (_type == TypeB) {
        // Draw TypeB button
    } /* ... */
}
```

这样写现在看上去还算简单，然而，若是需要依按钮类型来切换的绘制方法有许多种，那么就会变得很麻烦了。优秀的程序员会将这种代码重构为多个子类，把各种按钮所用的绘制方法放到相关子类中去。不过，这么做需要用户知道各种子类才行。此时应该使用“类族模式”，该模式可以灵活应对多个类，将它们的实现细节隐藏在抽象基类后面，以保持接口简洁。用户无须自己创建子类实例，只需调用基类方法来创建即可。

创建类族

现在举例来演示如何创建类族。假设有一个处理雇员的类，每个雇员都有“名字”和“薪水”这两个属性，管理者可以命令其执行日常工作。但是，各种雇员的工作内容却不同。经理在带领雇员做项目时，无须关心每个人如何完成其工作，仅需指示其开工即可。

首先要定义抽象基类：

```
typedef NS_ENUM(NSUInteger, EOCEmployeeType) {
    EOCEmployeeTypeDeveloper,
    EOCEmployeeTypeDesigner,
    EOCEmployeeTypeFinance,
};

@interface EOCEmployee : NSObject
```

⊖ 也叫做“类别方法”。——译者注

```
@property (copy) NSString *name;
@property NSUInteger salary;

// Helper for creating Employee objects
+ (EOCEmployee*)employeeWithType:(EOCEmployeeType)type;

// Make Employees do their respective day's work
- (void)doADaysWork;

@end

@implementation EOCEmployee

+ (EOCEmployee*)employeeWithType:(EOCEmployeeType)type {
    switch (type) {
        case EOCEmployeeTypeDeveloper:
            return [EOCEmployeeDeveloper new];
            break;
        case EOCEmployeeTypeDesigner:
            return [EOCEmployeeDesigner new];
            break;
        case EOCEmployeeTypeFinance:
            return [EOCEmployeeFinance new];
            break;
    }
}

- (void)doADaysWork {
    // Subclasses implement this.
}

@end
```

每个“实体子类”(concrete subclass)[⊖] 都从基类继承而来。例如：

```
@interface EOCEmployeeDeveloper : EOCEmployee
@end

@implementation EOCEmployeeDeveloper

- (void)doADaysWork {
    [self writeCode];
}

@end
```

⊖ 实体子类一词中的“concrete”与“抽象基类”一词中的“abstract”相对，意思是“非抽象的，可以实例化的”。该词也译为“具体”、“具象”。——译者注

在本例中，基类实现了一个“类方法”，该方法根据待创建的雇员类别分配好对应的雇员类实例。这种“工厂模式”（Factory pattern）是创建类族的办法之一。

可惜 Objective-C 这门语言没办法指明某个基类是“抽象的”（abstract）。于是，开发者通常会在文档中写明类的用法。这种情况下，基类接口一般都没有名为 init 的成员方法，这暗示该类的实例也许不应该由用户直接创建。还有一种办法可以确保用户不会使用基类实例，那就是在基类的 doADaysWork 方法中抛出异常。然而这种做法相当极端，很少有人用。

如果对象所属的类位于某个类族中，那么在查询其类型信息（introspection）[㊀] 时就要当心了（参见第 14 条）。你可能觉得自己创建了某个类的实例，然而实际上创建的却是其子类的实例。在 Employee 这个例子中，[employee isMemberOfClass:[EOCEmployee class]] 似乎会返回 YES，但实际上返回的却是 NO，因为 employee 并非 Employee 类的实例，而是其某个子类的实例。

Cocoa 里的类族

系统框架中有许多类族。大部分 collection 类都是类族[㊁]，例如 NSArray 与其可变版本 NSMutableArray。这样看来，实际上有两个抽象基类，一个用于不可变数组，另一个用于可变数组。尽管具备公共接口的类有两个，但仍然可以合起来算作一个类族[㊂]。不可变的类定义了对所有数组都通用的方法，而可变的类则定义了那些只适用于可变数组的方法。两个类共属同一类族，这意味着二者在实现各自类型的数组时可以共用实现代码，此外，还能够把可变数组复制为不可变数组，反之亦然。

在使用 NSArray 的 alloc 方法来获取实例时，该方法首先会分配一个属于某类的实例，此实例充当“占位数组”（placeholder array）。该数组稍后会转为另一个类的实例，而那个类则是 NSArray 的实体子类。这个过程稍显复杂，其完整的解释已经超出本书范围。

像 NSArray 这样的类的背后其实是个类族（对于大部分 collection 类而言都是这样），明白这一点很重要，否则就可能会写出下面这种代码：

```
id maybeAnArray = /* ... */;
if ([maybeAnArray class] == [NSArray class]) {
    // Will never be hit
}
```

你要是知道 NSArray 是个类族，那就会明白上述代码错在哪里：其中的 if 语句永远不可能为真。[maybeAnArray class] 所返回的类绝不可能是 NSArray 类本身，因为由 NSArray 的

㊀ type introspection 的简称，是某些面向对象语言可以在运行期检视对象类型与属性的一种功能。中文译作“内省”或“类型内省”。——译者注

㊁ 作者有时把“类族中的抽象基类”（the abstract base class of a class cluster）直接称为“类族”。这句话实际上是说：大部分 collection 类都是某个类族中的抽象基类。——译者注

㊂ 在传统的类族模式中，通常只有一个类具备“公共接口”（public interface），这个类就是类族中的抽象基类。——译者注

初始化方法所返回的那个实例其类型是隐藏在类族公共接口（public facade）[⊖]后面的某个内部类型（internal type）。

不过，仍然有办法可以判断出某个实例所属的类是否位于类族之中。我们不用刚才那种写法，而是改用类型信息查询方法（introspection method）。本书第 14 条解释了这些方法的用法。若想判断某对象是否位于类族中，不要直接检测两个“类对象”是否等同，而应该采用下列代码：

```
id maybeAnArray = /* ... */;
if ([maybeAnArray isKindOfClass:[NSArray class]]) {
    // Will be hit
}
```

我们经常需要向类族中新增实体子类，不过这么做的时候得留心。在 Employee 这个例子中，若是没有“工厂方法”（factory method）的源代码，那就无法向其中新增雇员类别了。然而对于 Cocoa 中 NSArray 这样的类族来说，还是有办法新增子类的，但是需要遵守几条规则。这几条规则如下。

■ 子类应该继承自类族中的抽象基类。

若要编写 NSArray 类族的子类，则需令其继承自不可变数组的基类或可变数组的基类。

■ 子类应该定义自己的数据存储方式。

开发者编写 NSArray 子类时，经常在这个问题上受阻。子类必须用一个实例变量来存放数组中的对象。这似乎与大家预想的不同，我们以为 NSArray 自己肯定会保存那些对象，所以在子类中就无须再存一份了。但是大家要记住，NSArray 本身只不过是包在其他隐藏对象外面的壳，它仅仅定义了所有数组都需具备的一些接口。对于这个自定义的数组子类来说，可以用 NSArray 来保存其实例。

■ 子类应当覆写超类文档中指明需要覆写的方法。

在每个抽象基类中，都有一些子类必须覆写的方法。比如说，想要编写 NSArray 的子类，就需要实现 count 及“ objectAtIndex:”方法。像 lastObject 这种方法则无须实现，因为基类可以根据前两个方法实现出这个方法。

在类族中实现子类时所需遵循的规范一般都会定义于基类的文档之中，编码前应该先看看。

要点

- 类族模式可以把实现细节隐藏在一套简单的公共接口后面。
- 系统框架中经常使用类族。
- 从类族的公共抽象基类中继承子类时要当心，若有开发文档，则应首先阅读。

第 10条：在既有类中使用关联对象存放自定义数据

有时需要在对象中存放相关信息。这时我们通常会从对象所属的类中继承一个子类，然

⊖ facade 又称为“外观”、“门面”。——译者注

后改用这个子类对象。然而并非所有情况下都能这么做，有时候类的实例可能是由某种机制所创建的，而开发者无法令这种机制创建出自己所写的子类实例。Objective-C 中有一项强大的特性可以解决此问题，这就是“关联对象”（Associated Object）。

可以给某对象关联许多其他对象，这些对象通过“键”来区分。存储对象值的时候，可以指明“存储策略”（storage policy），用以维护相应的“内存管理语义”。存储策略由名为 objc_AssociationPolicy 的枚举所定义，表 2-1 列出了该枚举的取值，同时还列出了与之等效的 @property 属性：假如关联对象成为了属性，那么它就会具备对应的语义（第 6 条详解了“属性”这个概念）。

表 2-1 对象关联类型

关联类型	等效的 @property 属性
OBJC_ASSOCIATION_ASSIGN	assign
OBJC_ASSOCIATION_RETAIN_NONATOMIC	nonatomic, retain
OBJC_ASSOCIATION_COPY_NONATOMIC	nonatomic, copy
OB]C_ASSOCIATION_RETAIN	retain
OBJC_ASSOCIATION_COPY	copy

下列方法可以管理关联对象：

- void objc_setAssociatedObject（id object, void*key, id value, objc_AssociationPolicy policy）

此方法以给定的键和策略为某对象设置关联对象值。

- id objc_getAssociatedObject(id object, void*key)

此方法根据给定的键从某对象中获取相应的关联对象值。

- void objc_removeAssociatedObjects(id object)

此方法移除指定对象的全部关联对象。

我们可以把某对象想象成 NSDictionary，把关联到该对象的值理解为字典中的条目，于是，存取关联对象的值就相当于在 NSDictionary 对象上调用 [object setObject:value forKey:key] 与 [object objectForKey:key] 方法。然而两者之间有个重要差别：设置关联对象时用的键（key）是个“不透明的指针”（opaque pointer）[⊖]。如果在两个键上调用“isEqual:”方法的返回值是 YES，那么 NSDictionary 就认为二者相等；然而在设置关联对象值时，若想令两个键匹配到同一个值，则二者必须是完全相同的指针才行。鉴于此，在设置关联对象值时，通常使用静态全局变量做键。

关联对象用法举例

开发 iOS 时经常用到 UIAlertView 类，该类提供了一种标准视图，可向用户展示警告信

⊖ 其所指向的数据结构不局限于某种特定类型的指针。详情参见：http://en.wikipedia.org/wiki/Opaque_pointer。——译者注

息。当用户按下按钮关闭该视图时，需要用委托协议（delegate protocol）来处理此动作，但是，要想设置好这个委托机制，就得把创建警告视图和处理按钮动作的代码分开。由于代码分作两块，所以读起来有点乱。比方说，我们在使用 UIAlertView 时，一般都会这么写：

```
- (void)askUserAQuestion {
    UIAlertView *alert = [[UIAlertView alloc]
                          initWithTitle:@"Question"
                          message:@"What do you want to do?"
                          delegate:self
                          cancelButtonTitle:@"Cancel"
                          otherButtonTitles:@"Continue", nil];
    [alert show];
}

// UIAlertViewDelegate protocol method
- (void)alertView:(UIAlertView *)alertView
    clickedButtonAtIndex:(NSInteger)buttonIndex
{
    if (buttonIndex == 0) {
        [self doCancel];
    } else {
        [self doContinue];
    }
```

如果想在同一个类里处理多个警告信息视图，那么代码就会变得更为复杂，我们必须在 delegate 方法中检查传入的 alertView 参数，并据此选用相应的逻辑。要是能在创建警告视图的时候直接把处理每个按钮的逻辑都写好，那就简单多了。这可以通过关联对象来做。创建完警告视图之后，设定一个与之关联的“块”（block）[⊖]，等到执行 delegate 方法时再将其读出来。此方案的实现代码如下：

```
#import <objc/runtime.h>

static void *EOCMyAlertViewKey = "EOCMyAlertViewKey";

- (void)askUserAQuestion {
    UIAlertView *alert = [[UIAlertViewalloc]
                          initWithTitle:@"Question"
                          message:@"What do you want to do?"
                          delegate:self
                          cancelButtonTitle:@"Cancel"
                          otherButtonTitles:@"Continue", nil];

    void (^block)(NSInteger) = ^(NSInteger buttonIndex){
        if (buttonIndex == 0) {
            [self doCancel];
```

⊖ 也称“区块”或“代码块”。——译者注

```
        } else {
            [self doContinue];
        }
    };

    objc_setAssociatedObject(alert,
                             EOCMyAlertViewKey,
                             block,
                             OBJC_ASSOCIATION_COPY);

    [alert show];
}

// UIAlertViewDelegate protocol method
- (void)alertView:(UIAlertView*)alertView
    clickedButtonAtIndex:(NSInteger)buttonIndex
{
    void (^block)(NSInteger) =
        objc_getAssociatedObject(alertView, EOCMyAlertViewKey);
    block(buttonIndex);
}
```

以这种方式改写之后，创建警告视图与处理操作结果的代码都放在一起了，这样比原来更易读懂，因为我们无须在两部分代码之间来回游走，即可明白警告视图的用处。但是，采用该方案时需注意：块可能要捕获（capture）某些变量，这也许会造成“保留环”（retain cycle）。第 40 条详述了此问题。

正如大家所见，这种做法很有用，但是只应该在其他办法行不通时才去考虑用它。若是滥用，则很快就会令代码失控，使其难于调试。“保留环”产生的原因很难查明，因为关联对象之间的关系并没有正式的定义（formal definition），其内存管理语义是在关联的时候才定义的，而不是在接口中预先定好的。使用这种写法时要小心，不能仅仅因为某处可以用该写法就一定要用它。想创建这种 UIAlertView 还有个办法，那就是从中继承子类，把块保存为子类中的属性。笔者认为：若是需要多次用到 alert 视图，那么这种做法比使用关联对象要好。

要点

- 可以通过“关联对象”机制来把两个对象连起来。
- 定义关联对象时可指定内存管理语义，用以模仿定义属性时所采用的“拥有关系”与“非拥有关系”。
- 只有在其他做法不可行时才应选用关联对象，因为这种做法通常会引入难于查找的 bug。

第 11 条：理解 objc_msgSend 的作用

在对象上调用方法是 Objective-C 中经常使用的功能。用 Objective-C 的术语来说，这叫

做“传递消息”(pass a message)。消息有“名称”(name) 或“选择子”(selector)[⊖]，可以接受参数，而且可能还有返回值。

由于 Objective-C 是 C 的超集，所以最好先理解 C 语言的函数调用方式。C 语言使用“静态绑定”(static binding)，也就是说，在编译期就能决定运行时所应调用的函数。以下列代码为例：

```
#import <stdio.h>

void printHello() {
    printf("Hello, world!\n");
}
void printGoodbye() {
    printf("Goodbye, world!\n");
}

void doTheThing(int type) {
    if (type == 0) {
        printHello();
    } else {
        printGoodbye();
    }
    return 0;
}
```

如果不考虑“内联”(inline)，那么编译器在编译代码的时候就已经知道程序中有 printHello 与 printGoodbye 这两个函数了，于是会直接生成调用这些函数的指令。而函数地址实际上是硬编码在指令之中的。若是将刚才那段代码写成下面这样，会如何呢？

```
#import <stdio.h>

void printHello() {
    printf("Hello, world!\n");
}
void printGoodbye() {
    printf("Goodbye, world!\n");
}

void doTheThing(int type) {
    void (*fnc)();
    if (type == 0) {
        fnc = printHello;
    } else {
        fnc = printGoodbye;
    }
    fnc();
    return 0;
```

⊖ 也叫“选择器”、“选取器”。——译者注

```
}
```

这时就得使用“动态绑定”（dynamic binding）了，因为所要调用的函数直到运行期才能确定。编译器在这种情况下生成的指令与刚才那个例子不同，在第一个例子中，if 与 else 语句里都有函数调用指令。而在第二个例子中，只有一个函数调用指令，不过待调用的函数地址无法硬编码在指令之中，而是要在运行期读取出来。

在 Objective-C 中，如果向某对象传递消息，那就会使用动态绑定机制来决定需要调用的方法。在底层，所有方法都是普通的 C 语言函数，然而对象收到消息之后，究竟该调用哪个方法则完全于运行期决定，甚至可以在程序运行时改变，这些特性使得 Objective-C 成为一门真正的动态语言。

给对象发送消息可以这样来写：

```
id returnValue = [someObject messageName:parameter];
```

在本例中，someObject 叫做“接收者”（receiver），messageName 叫做“选择子”（selector）。选择子与参数合起来称为“消息”（message）。编译器看到此消息后，将其转换为一条标准的 C 语言函数调用，所调用的函数乃是消息传递机制中的核心函数，叫做 objc_msgSend，其“原型”（prototype）如下：

```
void objc_msgSend(id self, SEL cmd, ...)
```

这是个“参数个数可变的函数”（variadic function）[⊖]，能接受两个或两个以上的参数。第一个参数代表接收者，第二个参数代表选择子（SEL 是选择子的类型），后续参数就是消息中的那些参数，其顺序不变。选择子指的就是方法的名字。“选择子”与“方法”这两个词经常交替使用。编译器会把刚才那个例子中的消息转换为如下函数：

```
id returnValue = objc_msgSend(someObject,
                              @selector(messageName:),
                              parameter);
```

objc_msgSend 函数会依据接收者与选择子的类型来调用适当的方法。为了完成此操作，该方法需要在接收者所属的类中搜寻其“方法列表”（list of methods），如果能找到与选择子名称相符的方法，就跳至其实现代码。若是找不到，那就沿着继承体系继续向上查找，等找到合适的方法之后再跳转。如果最终还是找不到相符的方法，那就执行“消息转发”（message forwarding）操作。消息转发将在第 12 条中详解。

这么说来，想调用一个方法似乎需要很多步骤。所幸 objc_msgSend 会将匹配结果缓存在“快速映射表”（fast map）里面，每个类都有这样一块缓存，若是稍后还向该类发送与选择子相同的消息，那么执行起来就很快了。当然啦，这种“快速执行路径”（fast path）还是不如“静态绑定的函数调用操作”（statically bound function call）那样迅速，不过只要把选择子缓存起来了，也就不会慢很多，实际上，消息派发（message dispatch）并非应用程序的瓶颈所在。

⊖ 也称为“可变参数函数”。——译者注

假如真是个瓶颈的话，那你可以只编写纯 C 函数，在调用时根据需要，把 Objective-C 对象的状态传进去。

前面讲的这部分内容只描述了部分消息的调用过程，其他“边界情况”(edge case)[㊀] 则需要交由 Objective-C 运行环境中的另一些函数来处理：

- objc_msgSend_stret。如果待发送的消息要返回结构体，那么可交由此函数处理。只有当 CPU 的寄存器能够容纳得下消息返回类型时，这个函数才能处理此消息。若是返回值无法容纳于 CPU 寄存器中（比如说返回的结构体太大了），那么就由另一个函数执行派发。此时，那个函数会通过分配在栈上的某个变量来处理消息所返回的结构体。
- objc_msgSend_fpret。如果消息返回的是浮点数，那么可交由此函数处理。在某些架构的 CPU 中调用函数时，需要对“浮点数寄存器”(floating-point register）做特殊处理，也就是说，通常所用的 objc_msgSend 在这种情况下并不合适。这个函数是为了处理 x86 等架构 CPU 中某些令人稍觉惊讶的奇怪状况。
- objc_msgSendSuper。如果要给超类发消息，例如 [super message:parameter]，那么就交由此函数处理。也有另外两个与 objc_msgSend_stret 和 objc_msgSend_fpret 等效的函数，用于处理发给 super 的相应消息。

刚才曾提到，objc_msgSend 等函数一旦找到应该调用的方法实现[㊁] 之后，就会“跳转过去”。之所以能这样做，是因为 Objective-C 对象的每个方法都可以视为简单的 C 函数，其原型如下：

```
<return_type> Class_selector(id self, SEL _cmd, ...)
```

真正的函数名和上面写的可能不太一样，笔者用“类”（class）和“选择子”（selector）来命名是想解释其工作原理。每个类里都有一张表格，其中的指针都会指向这种函数，而选择子的名称则是查表时所用的“键”。objc_msgSend 等函数正是通过这张表格来寻找应该执行的方法并跳至其实现的。请注意，原型的样子和 objc_msgSend 函数很像。这不是巧合，而是为了利用“尾调用优化”(tail-call optimization）[㊂] 技术，令“跳至方法实现”这一操作变得更简单些。

如果某函数的最后一项操作是调用另外一个函数，那么就可以运用“尾调用优化”技术。编译器会生成调转至另一函数所需的指令码，而且不会向调用堆栈中推入新的“栈帧”(frame stack)。只有当某函数的最后一个操作仅仅是调用其他函数而不会将其返回值另作他用时，才能执行“尾调用优化”。这项优化对 objc_msgSend 非常关键，如果不这么做的话，那么每次调用 Objective-C 方法之前，都需要为调用 objc_msgSend 函数准备“栈帧”，大家在“栈踪迹”（stack trace）中可以看到这种“栈帧”。此外，若是不优化，还会过早地发生“栈溢出”（stack overflow）现象。

㊀ 可理解为“特殊情况”。——译者注

㊁ 作者在本书中所说的“实现”(implementation）大多是名词，意为“实现代码”或“可执行的指令”。——译者注

㊂ 又称“尾递归优化”，其技术细节可参考：http://en.wikipedia.org/wiki/Tail_call。——译者注

在实际编写 Objective-C 代码的过程中，大家无须担心这一问题，不过应该了解其底层工作原理。这样的话，你就会明白，在发送消息时，代码究竟是如何执行的，而且也能理解，为何在调试的时候，栈“回溯”（backtrace）信息中总是出现 objc_msgSend。

要点

- 消息由接收者、选择子及参数构成。给某对象“发送消息”（invoke a message）[⊖] 也就相当于在该对象上“调用方法”（call a method）。
- 发给某对象的全部消息都要由“动态消息派发系统”（dynamic message dispatch system）来处理，该系统会查出对应的方法，并执行其代码。

第 12 条：理解消息转发机制

第 11 条讲解了对象的消息传递机制，并强调了其重要性。第 12 条则要讲解另外一个重要的问题，就是对象在收到无法解读的消息之后会发生什么情况。

若想令类能理解某条消息，我们必须以程序码实现出对应的方法才行。但是，在编译期向类发送了其无法解读的消息并不会报错，因为在运行期可以继续向类中添加方法，所以编译器在编译时还无法确知类中到底会不会有某个方法实现。当对象接收到无法解读的消息后，就会启动“消息转发”（message forwarding）机制，程序员可经由此过程告诉对象应该如何处理未知消息。

你可能早就遇到过经由消息转发流程所处理的消息了，只是未加留意。如果在控制台中看到下面这种提示信息，那就说明你曾向某个对象发送过一条其无法解读的消息，从而启动了消息转发机制，并将此消息转发给了 NSObject 的默认实现。

```
-[__NSCFNumber lowercaseString]: unrecognized selector sent to
instance 0x87
*** Terminating app due to uncaught exception
'NSInvalidArgumentException', reason: '-[__NSCFNumber
lowercaseString]: unrecognized selector sent to instance 0x87'
```

上面这段异常信息是由 NSObject 的“doesNotRecognizeSelector:”方法所抛出的，此异常表明：消息接收者的类型是 __NSCFNumber，而该接收者无法理解名为 lowercaseString 的选择子。本例所列举的这种情况并不奇怪，因为 NSNumber 类里本来就没有名为 lowercaseString 的方法。控制台中看到的那个 __NSFCNumber 是为了实现“无缝桥接”（toll-free bridging，第 49 条将会详解此技术）而使用的内部类（internal class），配置 NSNumber 对象时也会一并创建此对象。在本例中，消息转发过程以应用程序崩溃而告终，不过，开发者在编写自己的类时，可于转发过程中设置挂钩，用以执行预定的逻辑，而不使应用程序崩溃。

⊖ “invoke”也是“调用”的意思，此处为了与“call”相区隔，将其临时译为“发送”，也可理解为“激发”、“触发”。——译者注

消息转发分为两大阶段。第一阶段先征询接收者，所属的类，看其是否能动态添加方法，以处理当前这个“未知的选择子”（unknown selector），这叫做“动态方法解析”（dynamic method resolution）。第二阶段涉及“完整的消息转发机制”（full forwarding mechanism）。如果运行期系统已经把第一阶段执行完了，那么接收者自己就无法再以动态新增方法的手段来响应包含该选择子的消息了。此时，运行期系统会请求接收者以其他手段来处理与消息相关的方法调用。这又细分为两小步。首先，请接收者看看有没有其他对象能处理这条消息。若有，则运行期系统会把消息转给那个对象，于是消息转发过程结束，一切如常。若没有“备援的接收者”（replacement receiver），则启动完整的消息转发机制，运行期系统会把与消息有关的全部细节都封装到 NSInvocation 对象中，再给接收者最后一次机会，令其设法解决当前还未处理的这条消息。

动态方法解析

对象在收到无法解读的消息后，首先将调用其所属类的下列类方法：

```
+ (BOOL)resolveInstanceMethod:(SEL)selector
```

该方法的参数就是那个未知的选择子，其返回值为 Boolean 类型，表示这个类是否能新增一个实例方法用以处理此选择子。在继续往下执行转发机制之前，本类有机会新增一个处理此选择子的方法。假如尚未实现的方法不是实例方法而是类方法，那么运行期系统就会调用另外一个方法，该方法与“resolveInstanceMethod:”类似，叫做“resolveClassMethod:”。

使用这种办法的前提是：相关方法的实现代码已经写好，只等着运行的时候动态插在类里面就可以了。此方案常用来实现 @dynamic 属性（参见第 6 条），比如说，要访问 CoreData 框架中 NSManagedObjects 对象的属性时就可以这么做，因为实现这些属性所需的存取方法在编译期就能确定。

下列代码演示了如何用“resolveInstanceMethod:”来实现 @dynamic 属性：

```
id autoDictionaryGetter(id self, SEL _cmd);
void autoDictionarySetter(id self, SEL _cmd, id value);

+ (BOOL)resolveInstanceMethod:(SEL)selector {
    NSString *selectorString = NSStringFromSelector(selector);
    if ( /* selector is from a @dynamic property */ ) {
        if ([selectorString hasPrefix:@"set"]) {
            class_addMethod(self,
                            selector,
                            (IMP)autoDictionarySetter,
                            "v@:@");
        } else {
            class_addMethod(self,
                            selector,
                            (IMP)autoDictionaryGetter,
                            "@@:");
```

```
        }
        return YES;
    }
return [super resolveInstanceMethod:selector];
}
```

首先将选择子化为字符串，然后检测其是否表示设置方法。若前缀为 set，则表示设置方法，否则就是获取方法。不管哪种情况，都会把处理该选择子的方法加到类里面，所添加的方法是用纯 C 函数实现的。C 函数可能会用代码来操作相关的数据结构，类之中的属性数据就存放在那些数据结构里面。以 CoreData 为例，这些存取方法也许要和后端数据库通信，以便获取或更新相应的值。

备援接收者

当前接收者还有第二次机会能处理未知的选择子，在这一步中，运行期系统会问它：能不能把这条消息转给其他接收者来处理。与该步骤对应的处理方法如下：

```
- (id)forwardingTargetForSelector:(SEL)selector
```

方法参数代表未知的选择子，若当前接收者能找到备援对象，则将其返回，若找不到，就返回 nil。通过此方案，我们可以用“组合”（composition）来模拟出“多重继承”（multiple inheritance）的某些特性。在一个对象内部，可能还有一系列其他对象，该对象可经由此方法将能够处理某选择子的相关内部对象返回，这样的话，在外界看来，好像是该对象亲自处理了这些消息似的。

请注意，我们无法操作经由这一步所转发的消息。若是想在发送给备援接收者之前先修改消息内容，那就得通过完整的消息转发机制来做了。

完整的消息转发

如果转发算法已经来到这一步的话，那么唯一能做的就是启用完整的消息转发机制了。首先创建 NSInvocation 对象，把与尚未处理的那条消息有关的全部细节都封于其中。此对象包含选择子、目标（target）及参数。在触发 NSInvocation 对象时，“消息派发系统”（message-dispatch system）将亲自出马，把消息指派给目标对象。

此步骤会调用下列方法来转发消息：

```
- (void)forwardInvocation:(NSInvocation*)invocation
```

这个方法可以实现得很简单：只需改变调用目标，使消息在新目标上得以调用即可。然而这样实现出来的方法与“备援接收者”方案所实现的方法等效，所以很少有人采用这么简单的实现方式。比较有用的实现方式为：在触发消息前，先以某种方式改变消息内容，比如追加另外一个参数，或是改换选择子，等等。

实现此方法时，若发现某调用操作不应由本类处理，则需调用超类的同名方法。这样

的话，继承体系中的每个类都有机会处理此调用请求，直至 NSObject。如果最后调用了 NSObject 类的方法，那么该方法还会继而调用“ doesNotRecognizeSelector:”以抛出异常，此异常表明选择子最终未能得到处理。

消息转发全流程

图 2-2 这张流程图描述了消息转发机制处理消息的各个步骤。

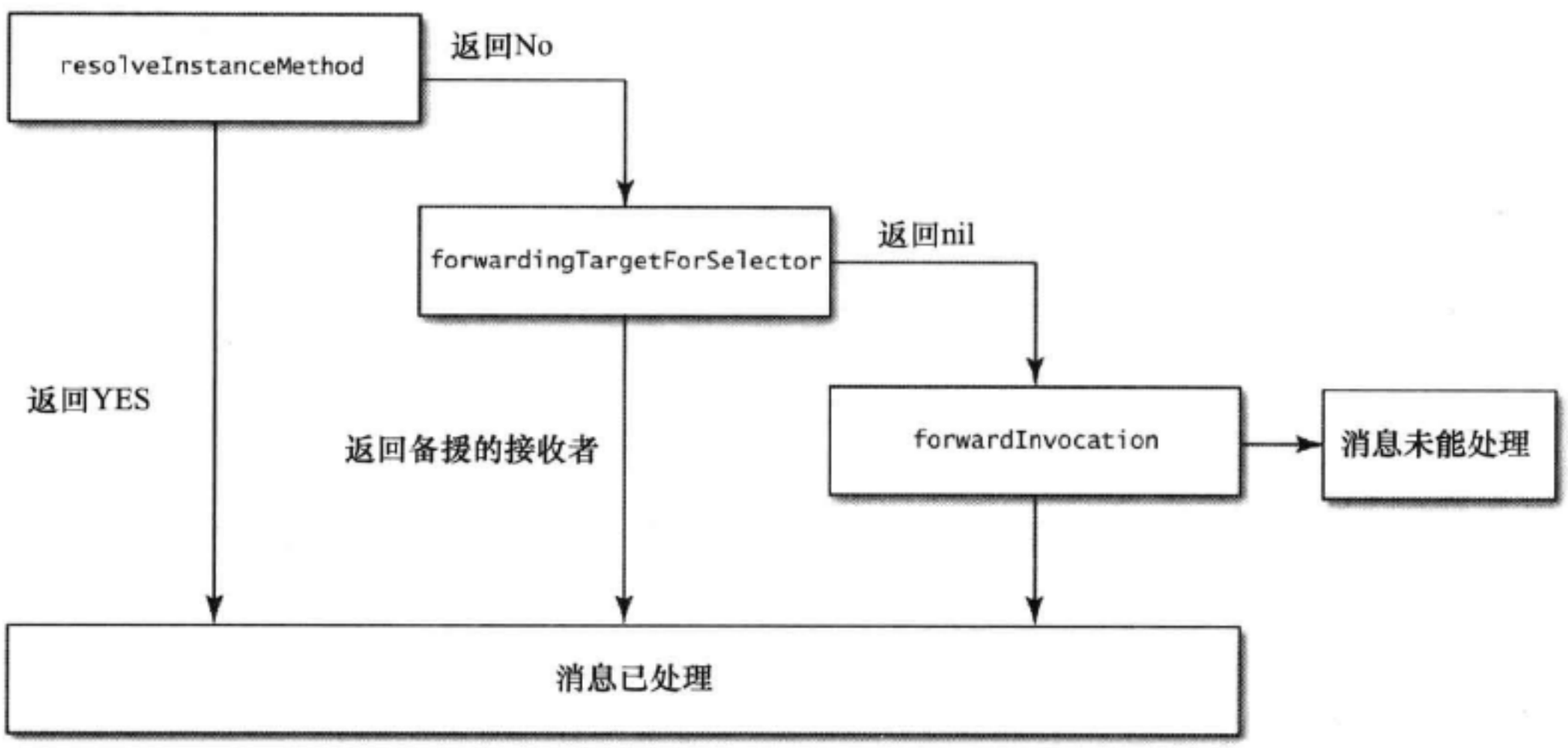

图 2-2　消息转发

接收者在每一步中均有机会处理消息。步骤越往后，处理消息的代价就越大。最好能在第一步就处理完，这样的话，运行期系统就可以将此方法缓存起来了。如果这个类的实例稍后还收到同名选择子，那么根本无须启动消息转发流程。若想在第三步里把消息转给备援的接收者，那还不如把转发操作提前到第二步。因为第三步只是修改了调用目标，这项改动放在第二步执行会更为简单，不然的话，还得创建并处理完整的 NSInvocation。

以完整的例子演示动态方法解析

为了说明消息转发机制的意义，下面示范如何以动态方法解析来实现 @dynamic 属性。假设要编写一个类似于“字典”的对象，它里面可以容纳其他对象，只不过开发者要直接通过属性来存取其中的数据。这个类的设计思路是：由开发者来添加属性定义，并将其声明为 @dynamic，而类则会自动处理相关属性值的存放与获取操作。怎么样，这项功能听起来不错吧？

该类的接口可以写成：

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

@interface EOCAutoDictionary : NSObject
@property (nonatomic, strong) NSString *string;
```

```
@property (nonatomic, strong) NSNumber *number;
@property (nonatomic, strong) NSDate *date;
@property (nonatomic, strong) id opaqueObject;
@end
```

本例中，这些属性具体是什么其实无关紧要。笔者用了这么多种数据类型，只是想演示此功能很有用。在类的内部，每个属性的值还是会存放在字典里，所以我们先在类中编写如下代码，并将属性声明为 @dynamic，这样的话，编译器就不会为其自动生成实例变量及存取方法了：

```
#import "EOCAutoDictionary.h"
#import <objc/runtime.h>

@interface EOCAutoDictionary ()
@property (nonatomic, strong) NSMutableDictionary
*backingStore;
@end
@implementation EOCAutoDictionary

@dynamic string, number, date, opaqueObject;

- (id)init {
    if ((self = [super init])) {
        _backingStore = [NSMutableDictionary new];
    }
    return self;
}
```

本例的关键在于 resolveInstanceMethod: 方法的实现代码：

```
+ (BOOL)resolveInstanceMethod:(SEL)selector {
    NSString *selectorString = NSStringFromSelector(selector);
    if ([selectorString hasPrefix:@"set"]) {
        class_addMethod(self,
                        selector,
                        (IMP)autoDictionarySetter,
                        "v@:@");
    } else {
        class_addMethod(self,
                        selector,
                        (IMP)autoDictionaryGetter,
                        "@@:");
    }
    return YES;
}
@end
```

当开发者首次在 EOCAutoDictionary 实例上访问某个属性时，运行期系统还找不到对应的选择子，因为所需的选择子既没有直接实现，也没有合成出来。现在假设要写入 opaqueObject 属性，那么系统就会以“ setOpaqueObject:”为选择子来调用上面这个方法。同理，在读取该属性时，系统也会调用上述方法，只不过传入的选择子是 opaqueObject。

resolveInstanceMethod 方法会判断选择子的前缀是否为 set，以此分辨其是 set 选择子还是 get 选择子。在这两种情况下，都要向类中新增一个处理该选择子所用的方法，这两个方法分别以 autoDictionarySetter 及 autoDictionaryGetter 函数指针的形式出现。此时就用到了 class_addMethod 方法，它可以向类中动态地添加方法，用以处理给定的选择子。第三个参数为函数指针，指向待添加的方法。而最后一个参数则表示待添加方法的“类型编码”（type encoding）。在本例中，编码开头的字符表示方法的返回值类型，后续字符则表示其所接受的各个参数[⊖]。

getter 函数可以用下列代码实现：

```
id autoDictionaryGetter(id self, SEL _cmd) {
    // Get the backing store from the object
    EOCAutoDictionary *typedSelf = (EOCAutoDictionary*)self;
    NSMutableDictionary *backingStore = typedSelf.backingStore;

    // The key is simply the selector name
    NSString *key = NSStringFromSelector(_cmd);

    // Return the value
    return [backingStore objectForKey:key];
}
```

而 setter 函数则可以这么写：

```
void autoDictionarySetter(id self, SEL _cmd, id value) {
    // Get the backing store from the object
    EOCAutoDictionary *typedSelf = (EOCAutoDictionary*)self;
    NSMutableDictionary *backingStore = typedSelf.backingStore;

    /** The selector will be for example, "setOpaqueObject:".
     *  We need to remove the "set", ":" and lowercase the first
     *  letter of the remainder.
     */
    NSString *selectorString = NSStringFromSelector(_cmd);
    NSMutableString *key = [selectorString mutableCopy];

    // Remove the ':' at the end
    [key deleteCharactersInRange:NSMakeRange(key.length - 1, 1)];

    // Remove the 'set' prefix
    [key deleteCharactersInRange:NSMakeRange(0, 3)];

    // Lowercase the first character
```

⊖ 类型编码的详情参见：https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtTypeEncodings.html。——译者注

```
NSString *lowercaseFirstChar =
    [[key substringToIndex:1] lowercaseString];
[key replaceCharactersInRange:NSMakeRange(0, 1)
                   withString:lowercaseFirstChar];

if (value) {
    [backingStore setObject:value forKey:key];
} else {
    [backingStore removeObjectForKey:key];
}
}
```

EOCAutoDictionary 的用法很简单：

```
EOCAutoDictionary *dict = [EOCAutoDictionarynew];
dict.date = [NSDatedateWithTimeIntervalSince1970:475372800];
NSLog(@"dict.date = %@", dict.date);
// Output: dict.date = 1985-01-24 00:00:00 +0000
```

其他属性的访问方式与 date 类似，要想添加新属性，只需用 @property 来定义，并将其声明为 @dynamic 即可。在 iOS 的 CoreAnimation 框架中，CALayer 类就用了与本例相似的实现方式，这使得 CALayer 成为“兼容于键值编码的”(key-value-coding-compliant)[⊖] 容器类，也就等于说，能够向里面随意添加属性，然后以键值对的形式来访问。于是，开发者就可以向其中新增自定义的属性了，这些属性值的存储工作由基类直接负责，我们只需在 CALayer 的子类中定义新属性即可。

要点

- 若对象无法响应某个选择子，则进入消息转发流程。
- 通过运行期的动态方法解析功能，我们可以在需要用到某个方法时再将其加入类中。
- 对象可以把其无法解读的某些选择子转交给其他对象来处理。
- 经过上述两步之后，如果还是没办法处理选择子，那就启动完整的消息转发机制。

第 13 条：用“方法调配技术”调试“黑盒方法”

第 11 条中解释过：Objective-C 对象收到消息之后，究竟会调用何种方法需要在运行期才能解析出来。那你也许会问：与给定的选择子名称相对应的方法是不是也可以在运行期改变呢？没错，就是这样。若能善用此特性，则可发挥出巨大优势，因为我们既不需要源代码，也不需要通过继承子类来覆写方法就能改变这个类本身的功能。这样一来，新功能将在

⊖ 该词的大意是，除了使用存取方法和“点语法”之外，还可以用字符串做键，通过“valueForKey:”与“setValue:forKey:”这种形式来访问属性。详情参见：https://developer.apple.com/library/ios/DOCUMENTATION/Cocoa/Conceptual/KeyValueCoding/Articles/KeyValueCoding.html。——译者注

本类的所有实例中生效，而不是仅限于覆写了相关方法的那些子类实例。此方案经常称为“方法调配”（method swizzling）[⊖]。

类的方法列表会把选择子的名称映射到相关的方法实现之上，使得“动态消息派发系统”能够据此找到应该调用的方法。这些方法均以函数指针的形式来表示，这种指针叫做 IMP，其原型如下：

```
id (*IMP)(id, SEL, ...)
```

NSString 类可以响应 lowercaseString、uppercaseString、capitalizedString 等选择子。这张映射表中的每个选择子都映射到了不同的 IMP 之上，如图 2-3 所示。

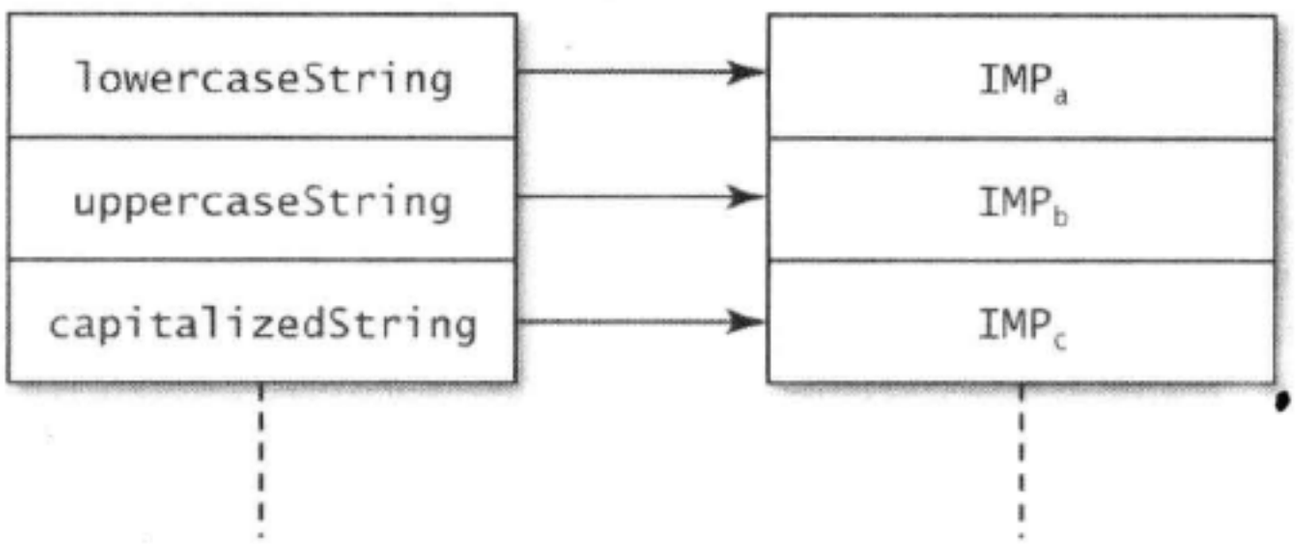

图 2-3　NSString 类的选择子映射表[⊜]

Objective-C 运行期系统提供的几个方法都能够用来操作这张表。开发者可以向其中新增选择子，也可以改变某选择子所对应的方法实现，还可以交换两个选择子所映射到的指针。经过几次操作之后，类的方法表就会变成图 2-4 这个样子。

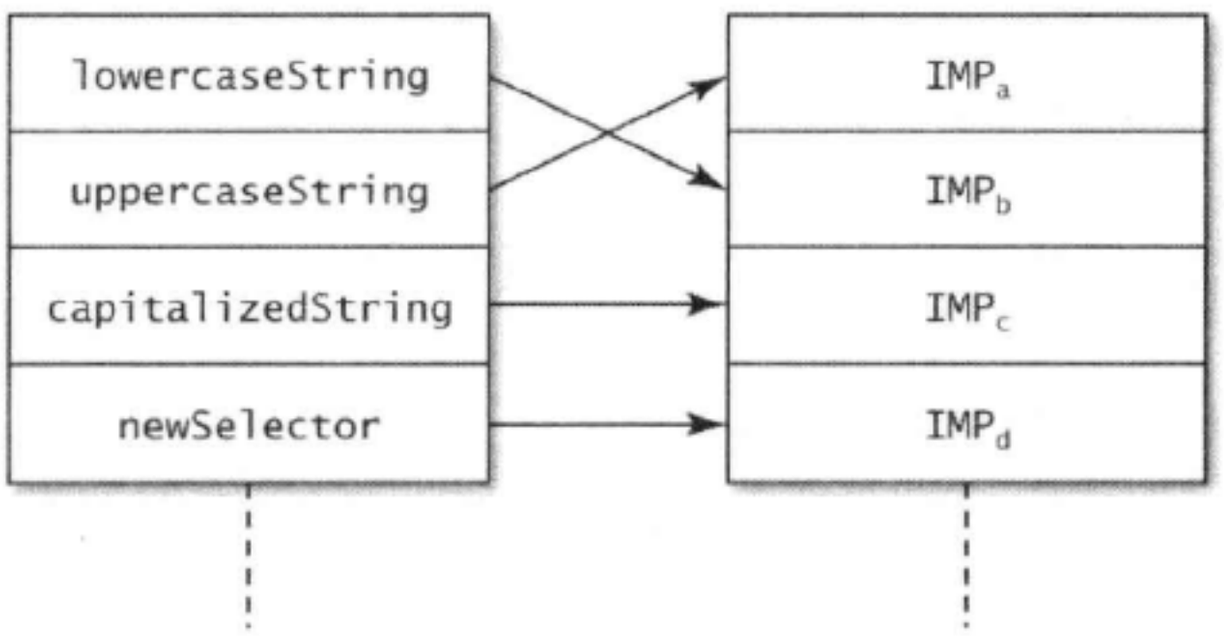

图 2-4　经过数次操作之后的 NSString 选择子映射表

在新的映射表中，多了一个名为 newSelector 的选择子，capitalizedString 的实现也变了，而 lowercaseString 与 uppercaseString 的实现则互换了。上述修改均无须编写子类，只要修改了“方法表”的布局，就会反映到程序中所有的 NSString 实例之上。这下大家见识到此特性

⊖ 也叫做“方法混合”、“方法调和”。——译者注

⊜ selector table，也称“选择器表”。——译者注

的强大之处了吧？

本条将会谈到如何互换两个方法实现。通过此操作，可为已有方法添加新功能。不过在讲解怎样添加新功能之前，我们先来看看怎样互换两个已经写好的方法实现。想交换方法实现，可用下列函数：

```
void method_exchangeImplementations(Method m1, Method m2)
```

此函数的两个参数表示待交换的两个方法实现，而方法实现则可通过下列函数获得：

```
Method class_getInstanceMethod(Class aClass, SEL aSelector)
```

此函数根据给定的选择从类中取出与之相关的方法。执行下列代码，即可交换前面提到的 lowercaseString 与 uppercaseString 方法实现：

```
Method originalMethod =
    class_getInstanceMethod([NSStringclass],
                            @selector(lowercaseString));
Method swappedMethod =
    class_getInstanceMethod([NSStringclass],
                            @selector(uppercaseString));
method_exchangeImplementations(originalMethod, swappedMethod);
```

从现在开始，如果在 NSString 实例上调用 lowercaseString，那么执行的将是 uppercaseString 的原有实现，反之亦然：

```
NSString *string = @"ThIs iS tHe StRiNg";

NSString *lowercaseString = [string lowercaseString];
NSLog(@"lowercaseString = %@", lowercaseString);
// Output: lowercaseString = THIS IS THE STRING

NSString *uppercaseString = [string uppercaseString];
NSLog(@"uppercaseString = %@", uppercaseString);
// Output: uppercaseString = this is the string
```

笔者刚才向大家演示了如何交换两个方法实现，然而在实际应用中，像这样直接交换两个方法实现的，意义并不大。因为 lowercaseString 与 uppercaseString 这两个方法已经各自实现得很好了，没必要再交换了。但是，可以通过这一手段来为既有的方法实现增添新功能。比方说，想要在调用 lowercaseString 时记录某些信息，这时就可以通过交换方法实现来达成此目标。我们新编写一个方法，在此方法中实现所需的附加功能，并调用原有实现。

新方法可以添加至 NSString 的一个“分类”（category）中：

```
@interface NSString (EOCMyAdditions)
- (NSString*)eoc_myLowercaseString;
@end
```

上述新方法将与原有的 lowercaseString 方法互换，交换之后的方法表如图 2-5 所示。

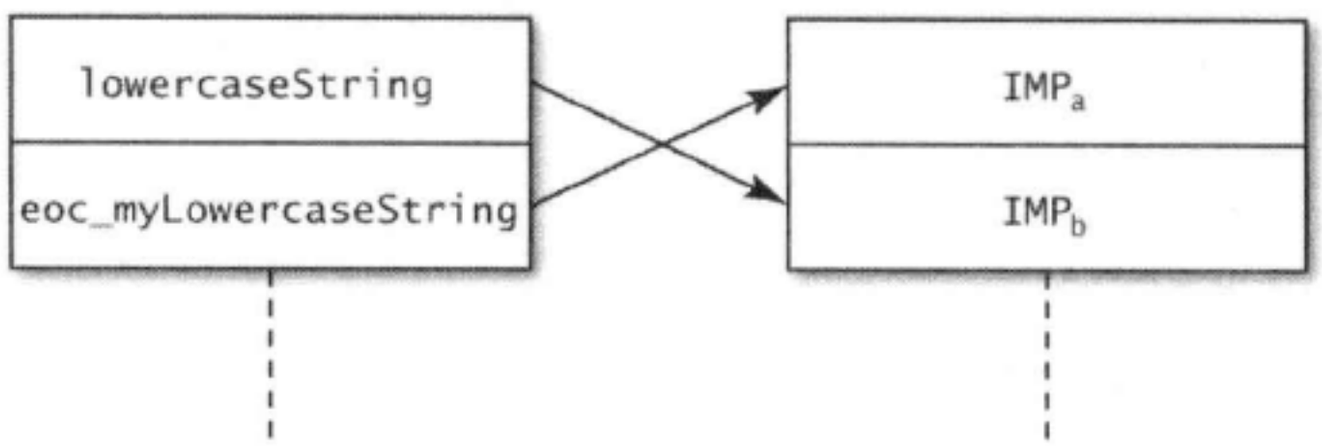

图 2-5　交换 lowercaseString 与 eoc_myLowercaseString 的方法实现

新方法的实现代码可以这样写：

```
@implementation NSString (EOCMyAdditions)
- (NSString*)eoc_myLowercaseString {
    NSString *lowercase = [self eoc_myLowercaseString];
    NSLog(@"%@ => %@", self, lowercase);
    return lowercase;
}
@end
```

这段代码看上去好像会陷入递归调用的死循环，不过大家要记住，此方法是准备和 lowercaseString 方法互换的。所以，在运行期，eoc_myLowercaseString 选择子实际上对应于原有的 lowercaseString 方法实现。最后，通过下列代码来交换这两个方法实现：

```
Method originalMethod =
    class_getInstanceMethod([NSString class],
                            @selector(lowercaseString));
Method swappedMethod =
    class_getInstanceMethod([NSString class],
                            @selector(eoc_myLowercaseString));
method_exchangeImplementations(originalMethod, swappedMethod);
```

执行完上述代码之后，只要在 NSString 实例上调用 lowercaseString 方法，就会输出一行记录消息：

```
NSString *string = @"ThIs iS tHe StRiNg";
NSString *lowercaseString = [string lowercaseString];
// Output: ThIs iS tHe StRiNg => this is the string
```

通过此方案，开发者可以为那些“完全不知道其具体实现的”（completely opaque，“完全不透明的”）黑盒方法增加日志记录功能，这非常有助于程序调试。然而，此做法只在调试程序时有用。很少有人在调试程序之外的场合用上述“方法调配技术”来永久改动某个类的功能。不能仅仅因为 Objective-C 语言里有这个特性就一定要用它。若是滥用，反而会令代码变得不易读懂且难于维护。

要点

- 在运行期，可以向类中新增或替换选择子所对应的方法实现。

- 使用另一份实现来替换原有的方法实现，这道工序叫做“方法调配”，开发者常用此技术向原有实现中添加新功能。
- 一般来说，只有调试程序的时候才需要在运行期修改方法实现，这种做法不宜滥用。

第14条：理解“类对象”的用意

Objective-C实际上是一门极其动态的语言。第11条讲解了运行期系统如何查找并调用某方法的实现代码，第12条则讲述了消息转发的原理：如果类无法立即响应某个选择子，那么就会启动消息转发流程。然而，消息的接收者究竟是何物？是对象本身吗？运行期系统如何知道某个对象的类型呢？对象类型并非在编译期就绑定好了，而是要在运行期查找。而且，还有个特殊的类型叫做id，它能指代任意的Objective-C对象类型。一般情况下，应该指明消息接收者的具体类型，这样的话，如果向其发送了无法解读的消息，那么编译器就会产生警告信息。而类型为id的对象则不然，编译器假定它能响应所有消息。

如果看过第12条，你就会明白，编译器无法确定某类型对象到底能解读多少种选择子，因为运行期还可向其中动态新增。然而，即便使用了动态新增技术，编译器也觉得应该能在某个头文件中找到方法原型的定义，据此可了解完整的“方法签名”(method signature)，并生成派发消息所需的正确代码。

“在运行期检视对象类型”这一操作也叫做“类型信息查询”(introspection，“内省”)，这个强大而有用的特性内置于Foundation框架的NSObject协议里，凡是由公共根类（common root class，即NSObject与NSProxy）继承而来的对象都要遵从此协议。在程序中不要直接比较对象所属的类，明智的做法是调用“类型信息查询方法”，其原因笔者稍后解释。不过在介绍类型信息查询技术之前，我们先讲一些基础知识，看看Objective-C对象的本质是什么。

每个Objective-C对象实例都是指向某块内存数据的指针。所以在声明变量时，类型后面要跟一个“*”字符：

```
NSString *pointerVariable = @"Some string";
```

编过C语言程序的人都知道这是什么意思。对于没写过C语言的程序员来说，pointerVariable可以理解成存放内存地址的变量，而NSString自身的数据就存于那个地址中。因此可以说，该变量“指向”（point to）NSString实例。所有Objective-C对象都是如此，若是想把对象所需的内存分配在栈上，编译器则会报错：

```
String stackVariable = @"Some string";
// error: interface type cannot be statically allocated
```

对于通用的对象类型id，由于其本身已经是指针了，所以我们能够这样写：

```
id genericTypedString = @"Some string";
```

上面这种定义方式与用NSString*来定义相比，其语法意义相同。唯一区别在于，如果声明时指定了具体类型，那么在该类实例上调用其所没有的方法时，编译器会探知此情况，

并发出警告信息。

描述 Objective-C 对象所用的数据结构定义在运行期程序库的头文件里，id 类型本身也在定义在这里：

```
typedef struct objc_object {
    Class isa;
} *id;
```

由此可见，每个对象结构体的首个成员是 Class 类的变量。该变量定义了对象所属的类，通常称为“is a”指针。例如，刚才的例子中所用的对象“是一个”(is a) NSString，所以其“is a”指针就指向 NSString。Class 对象也定义在运行期程序库的头文件中：

```
typedef struct objc_class *Class;
struct objc_class {
    Class isa;
    Class super_class;
    const char *name;
    long version;
    long info;
    long instance_size;
    struct objc_ivar_list *ivars;
    struct objc_method_list **methodLists;
    struct objc_cache *cache;
    struct objc_protocol_list *protocols;
};
```

此结构体存放类的“元数据”(metadata)，例如类的实例实现了几个方法，具备多少个实例变量等信息。此结构体的首个变量也是 isa 指针，这说明 Class 本身亦为 Objective-C 对象。结构体里还有个变量叫做 super_class，它定义了本类的超类。类对象所属的类型（也就是 isa 指针所指向的类型）是另外一个类，叫做“元类”(metaclass)，用来表述类对象本身所具备的元数据。“类方法”就定义于此处，因为这些方法可以理解成类对象的实例方法。每个类仅有一个“类对象”，而每个“类对象”仅有一个与之相关的“元类”。

假设有个名为 SomeClass 的子类从 NSObject 中继承而来，则其继承体系如图 2-6 所示。

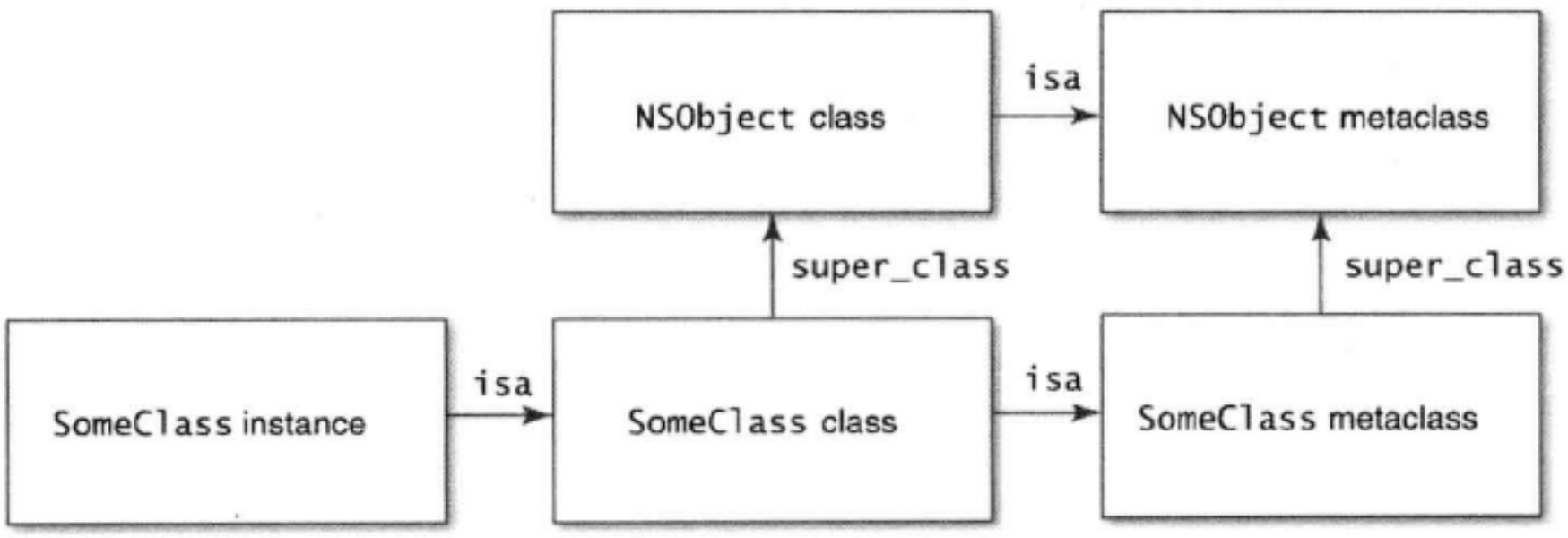

图 2-6 SomeClass 实例所属的“类继承体系”，此类继承自 NSObject，图中也画出了两个对应“元类”之间的继承关系

super_class 指针确立了继承关系，而 isa 指针描述了实例所属的类。通过这张布局关系图即可执行“类型信息查询”。我们可以查出对象是否能响应某个选择子，是否遵从某项协议，并且能看出此对象位于“类继承体系”（class hierarchy）的哪一部分。

在类继承体系中查询类型信息

可以用类型信息查询方法来检视类继承体系。“isMemberOfClass:”能够判断出对象是否为某个特定类的实例，而“isKindOfClass:”则能够判断出对象是否为某类或其派生类的实例。例如：

```
NSMutableDictionary *dict = [NSMutableDictionary new];
[dict isMemberOfClass:[NSDictionary class]]; ///< NO
[dict isMemberOfClass:[NSMutableDictionary class]]; ///< YES
[dict isKindOfClass:[NSDictionary class]]; ///< YES
[dict isKindOfClass:[NSArray class]]; ///< NO
```

像这样的类型信息查询方法使用 isa 指针获取对象所属的类，然后通过 super_class 指针在继承体系中游走。由于对象是动态的，所以此特性显得极为重要。Objective-C 与你可能熟悉的其他语言不同，在此语言中，必须查询类型信息，方能完全了解对象的真实类型。

由于 Objective-C 使用“动态类型系统”（dynamic typing），所以用于查询对象所属类的类型信息查询功能非常有用。从 collection 中获取对象时，通常会查询类型信息，这些对象不是“强类型的”（strongly typed），把它们从 collection 中取出来时，其类型通常是 id。如果想知道具体类型，那就可以使用类型信息查询方法。例如，想根据数组中存储的对象生成以逗号分隔的字符串（comma-separated string），并将其存至文本文件，就可以使用下列代码：

```
- (NSString*)commaSeparatedStringFromObjects:(NSArray*)array {
    NSMutableString *string = [NSMutableStringnew];
    for (id object in array) {
        if ([object isKindOfClass:[NSStringclass]]) {
            [string appendFormat:@"%@,", object];
        } else if ([object isKindOfClass:[NSNumberclass]]) {
            [string appendFormat:@"%d,", [object intValue]];
        } else if ([object isKindOfClass:[NSDataclass]]) {
            NSString *base64Encoded = /* base64 encoded data */;
            [string appendFormat:@"%@,", base64Encoded];
        } else {
            // Type not supported
        }
    }
    return string;
}
```

也可以用比较类对象是否等同的办法来做。若是如此，那就要使用 == 操作符，而不要

使用比较 Objective-C 对象时常用的“ isEqual:”方法（参见第 8 条）。原因在于，类对象是“单例”（singleton），在应用程序范围内，每个类的 Class 仅有一个实例。也就是说，另外一种可以精确判断出对象是否为某类实例的办法是：

```
id object = /* ... */;
if ([object class] == [EOCSomeClassclass]) {
    // 'object' is an instance of EOCSomeClass
}
```

即便能这样做，我们也应该尽量使用类型信息查询方法，而不应该直接比较两个类对象是否等同，因为前者可以正确处理那些使用了消息传递机制（参见第 12 条）的对象。比方说，某个对象可能会把其收到的所有选择子都转发给另外一个对象。这样的对象叫做“代理”（proxy），此种对象均以 NSProxy 为根类。

通常情况下，如果在此种代理对象上调用 class 方法，那么返回的是代理对象本身（此类是 NSProxy 的子类），而非接受的代理的对象所属的类。然而，若是改用“ isKindOfClass:”这样的类型信息查询方法，那么代理对象就会把这条消息转给“接受代理的对象”（proxied object）。也就是说，这条消息的返回值与直接在接受代理的对象上面查询其类型所得的结果相同。因此，这样查出来的类对象与通过 class 方法所返回的那个类对象不同，class 方法所返回的类表示发起代理的对象，而非接受代理的对象。

要点

- 每个实例都有一个指向 Class 对象的指针，用以表明其类型，而这些 Class 对象则构成了类的继承体系。
- 如果对象类型无法在编译期确定，那么就应该使用类型信息查询方法来探知。
- 尽量使用类型信息查询方法来确定对象类型，而不要直接比较类对象，因为某些对象可能实现了消息转发功能。

第 3 章

接口与 API 设计

我们在构建应用程序时，可能想将其中部分代码用于后续项目，也可能想把某些代码发布出来，供他人使用。即便现在还不想这么做，将来也总会有用到的时候。如果决定重用代码，那么我们在编写接口时就会将其设计成易于复用的形式。这需要用到 Objective-C 语言中常见的编程范式（paradigm），同时还需了解各种可能碰到的陷阱。

近年来，开源社区与开源组件随着 iOS 开发而流行起来，所以我们经常会在开发自己的应用程序时使用他人所写的代码。与此同时，别人也会用到你的代码，所以，要把代码写得清晰一些，以便其他开发者能够迅速而方便地将其集成到他们的项目里。没准会有成千上万个应用程序使用你所写的程序库呢！

第 15 条：用前缀避免命名空间冲突

Objective-C 没有其他语言那种内置的命名空间（namespace）机制。鉴于此，我们在起名时要设法避免潜在的命名冲突，否则很容易就重名了。如果发生命名冲突（naming clash），那么应用程序的链接过程就会出错，因为其中出现了重复符号：

```
duplicate symbol _OBJC_METACLASS_$_EOCTheClass in:
    build/something.o
    build/something_else.o
duplicate symbol _OBJC_CLASS_$_EOCTheClass in:
    build/something.o
    build/something_else.o
```

错误原因在于，应用程序中的两份代码都各自实现了名为 EOCTheClass 的类，这导致 EOCTheClass 所对应的类符号和“元类”（参见第 14 条）符号各定义了两次。你也许是把两个相互独立的程序库都引入到当前项目中，而它们又恰好有重名的类，所以产生了这一问题。

比无法链接更糟糕的情况是，在运行期载入了含有重名类的程序库。此时，“动态加载器”（dynamic loader）就遭遇了“重名符号错误”（duplicate symbol error），很可能会令整个应用程序崩溃。

避免此问题的唯一办法就是变相实现命名空间：为所有名称都加上适当前缀。所选前缀可以是与公司、应用程序或二者皆有关联之名。比方说，假设你所在的公司叫做 Effective Widgets，那么就可以在所有应用程序都会用到的那部分代码中使用 EWS 作前缀，如果有些代码只用于名为 Effective Browser 的浏览器项目中，那就在这部分代码中使用 EWB 作前缀。即便加了前缀，也难保不出现命名冲突，但是其几率会小很多。

使用 Cocoa 创建应用程序时一定要注意，Apple 宣称其保留使用所有“两字母前缀”（two-letter prefix）的权利，所以你自己选用的前缀应该是三个字母的。举个例子，假如开发者不遵循这条守则，使用 TW 这两个字母作前缀，那么就会出问题。iOS 5.0 SDK 发布时，包含了 Twitter 框架，此框架就使用 TW 作前缀，其中有个类叫做 TWRequest，它可以发送 HTTP 请求以调用 Twitter API。如果你所在的公司叫做 Tiny Widgets，那么很有可能把访问本公司 API 所用的那个类也命名为 TWRequest。

不仅是类名，应用程序中的所有名称都应加前缀。如果要为既有类新增“分类”（category），那么一定要给“分类”及“分类”中的方法加上前缀，第 25 条解释了这么做的原因。开发者可能会忽视另外一个容易引发命名冲突的地方，那就是类的实现文件中所用的纯 C 函数及全局变量，这个问题必须要注意。大家可别忘了：在编译好的目标文件中，这些名称是要算作“顶级符号”（top-level symbol）的。比方说，iOS SDK 的 AudioToolbox 里有个函数能播放声音文件。开发者可向其传入回调函数（callback），以便在播放完毕时调用。你也许想编写一个 Objective-C 类，把这套逻辑封装起来，当播放完声音文件之后，即命令其中的委托对象（delegate）处理回调事宜：

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

@class EOCSoundPlayer;
@protocol EOCSoundPlayerDelegate <NSObject>
- (void)soundPlayerDidFinish:(EOCSoundPlayer*)player;
@end

@interface EOCSoundPlayer : NSObject
@property (nonatomic, weak) id <EOCSoundPlayerDelegate> delegate;
- (id)initWithURL:(NSURL*)url;
- (void)playSound;
@end
// EOCSoundPlayer.m
#import "EOCSoundPlayer.h"
#import <AudioToolbox/AudioToolbox.h>

void completion(SystemSoundID ssID, void *clientData) {
    EOCSoundPlayer *player =
        (__bridge EOCSoundPlayer*)clientData;
    if ([player.delegate
            respondsToSelector:@selector(soundPlayerDidFinish:)])
    {
```

```
            [player.delegate soundPlayerDidFinish:player];
        }
    }

    @implementation EOCSoundPlayer {
        SystemSoundID _systemSoundID;
    }

    - (id)initWithURL:(NSURL*)url {
        if ((self = [super init])) {
            AudioServicesCreateSystemSoundID((__bridge CFURLRef)url,
                                             &_systemSoundID);
        }
        return self;
    }

    - (void)dealloc {
        AudioServicesDisposeSystemSoundID(_systemSoundID);
    }

    - (void)playSound {
        AudioServicesAddSystemSoundCompletion(
            _systemSoundID,
            NULL,
            NULL,
            completion,
            (__bridge void*)self);
        AudioServicesPlaySystemSound(_systemSoundID);
    }

    @end
```

这段代码看上去完全正常，不过你再看看该类目标文件中的符号表（symbol table），就会发现问题了：

```
00000230 t -[EOCSoundPlayer .cxx_destruct]
0000014c t -[EOCSoundPlayer dealloc]
000001e0 t -[EOCSoundPlayer delegate]
0000009c t -[EOCSoundPlayer initWithURL:]
00000198 t -[EOCSoundPlayer playSound]
00000208 t -[EOCSoundPlayer setDelegate:]
00000b88 S _OBJC_CLASS_$_EOCSoundPlayer
00000bb8 S _OBJC_IVAR_$_EOCSoundPlayer._delegate
00000bb4 S _OBJC_IVAR_$_EOCSoundPlayer._systemSoundID
00000b9c S _OBJC_METACLASS_$_EOCSoundPlayer
00000000 T _completion
00000bf8 s l_OBJC_$_INSTANCE_METHODS_EOCSoundPlayer
00000c48 s l_OBJC_$_INSTANCE_VARIABLES_EOCSoundPlayer
00000c78 s l_OBJC_$_PROP_LIST_EOCSoundPlayer
00000c88 s l_OBJC_CLASS_RO_$_EOCSoundPlayer
00000bd0 s l_OBJC_METACLASS_RO_$_EOCSoundPlayer
```

符号表中间有个名叫 _completion 的符号，这就是为了处理声音播放完毕之后的逻辑而创建的那个 completion 函数。虽说此函数是在实现文件里定义的，并没有声明于头文件中，不过它仍然算作“顶级符号”。这样的话，若在别处又创建了一个名叫 completion 的函数，则会于链接时发生类似下面这种“重复符号错误”：

```
duplicate symbol _completion in:
    build/EOCSoundPlayer.o
    build/EOCAnotherClass.o
```

如果将代码发布为程序库，供他人在开发应用程序时使用，那么就更糟糕了。这等于办了件坏事：因为已经有了名叫 _completion 的符号，所以使用此程序库的开发者就无法再创建名为 completion 的函数了。

由此可见，我们总是应该给这种 C 函数的名字加上前缀。比方说， 在刚才那个例子中，播放完声音之后所执行的处理程序[⊖]可以改名为 EOCSoundPlayerCompletion。这么做还有个好处：若此符号出现在栈回溯信息中，则很容易就能判明问题源自哪块代码。

如果用第三方库编写自己的代码，并准备将其再发布为程序库供他人开发应用程序所用，那么尤其要注意重复符号问题。你的程序库所包含的那个第三方库也许还会为应用程序本身所引入，若是如此，那就很容易出现重复符号错误了。这时应该给你所用的那一份第三方库代码都加上你自己的前缀。例如，你准备发布的程序库叫做 EOCLibrary，其中引入了名为 XYZLibrary 的第三方库，那么就应该把 XYZLibrary 中的所有名字都冠以 EOC。于是，应用程序就可以随意使用它自己直接引入的那个 XYZLibrary 库了，而不必担心与 EOCLibrary 里的这个 XYZLibrary 相冲突，图 3-1 演示了此时的情况。

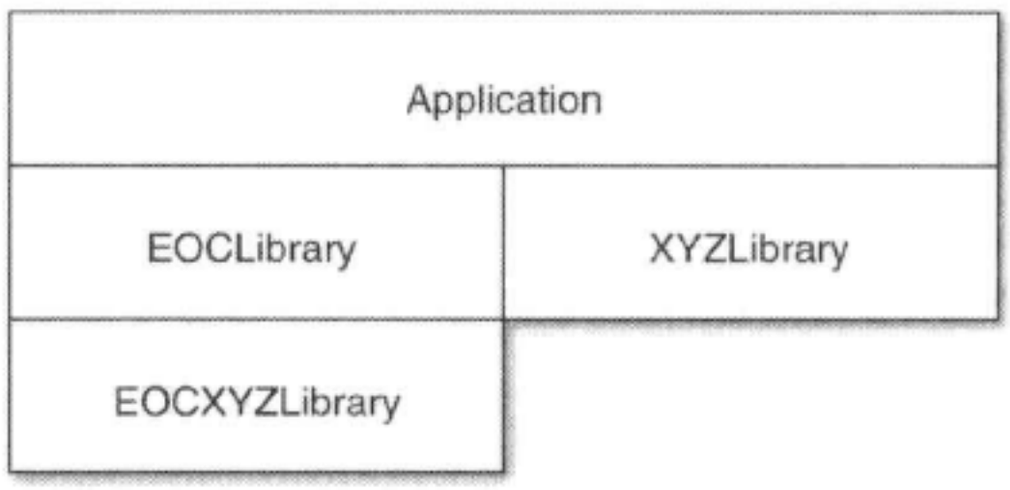

图 3-1 若应用程序自身和其所用的程序库都引入了同名的第三方库，则后者应加前缀以避免命名冲突

虽说逐个改名是很令人厌烦的事情，不过若想避免命名冲突，还是得费这番工夫才行。读者也许会问：为什么非要这么做呢？应用程序自己不要直接引入 XYZLibrary，改用 EOCLibrary 里面的那个不就行了吗？没错，可以这么做，但是，应用程序也许还会引入另一个名为 ABCLibrary 的第三方库，而该库中又包含了 XYZLibrary。此时，如果你和 ABCLibrary 库的作者都不给各自所用的 XYZLibrary 加前缀，那么应用程序依然会出现重复符号错误。还有一种可能就是，你的库里所用的 XYZLibrary 是 X 版本的，而应用程序却需要使用 Y 版本的某些功能，所以它必须自己再引入一份。你可以花些时间，使用几个流行的第三方库来开发一下 iOS 程序，那时会经常看到这种前缀的。

⊖ handler，也称“处理器”、“处置器”。——译者注

要点

- 选择与你的公司、应用程序或二者皆有关联之名称作为类名的前缀，并在所有代码中均使用这一前缀。
- 若自己所开发的程序库中用到了第三方库，则应为其中的名称加上前缀。

第16条：提供“全能初始化方法”

所有对象均要初始化。在初始化时，有些对象可能无须开发者向其提供额外信息，不过一般来说还是要提供的。通常情况下，对象若不知道必要的信息，则无法完成其工作。以iOS的UI框架UIKit为例，其中有个类叫做UITableViewCell，初始化该类对象时，需要指明其样式及标识符，标识符能够区分不同类型的单元格。由于这种对象的创建成本较高，所以绘制表格时可依照标识符来复用，以提升程序效率。我们把这种可为对象提供必要信息以便其能完成工作的初始化方法叫做“全能初始化方法”（designated initializer）[⊖]。

如果创建类实例的方式不止一种，那么这个类就会有多个初始化方法。这当然很好，不过仍然要在其中选定一个作为全能初始化方法，令其他初始化方法都来调用它。NSDate就是个例子，其初始化方法如下：

```
- (id)init
- (id)initWithString:(NSString*)string
- (id)initWithTimeIntervalSinceNow:(NSTimeInterval)seconds
- (id)initWithTimeInterval:(NSTimeInterval)seconds
                 sinceDate:(NSDate*)refDate
- (id)initWithTimeIntervalSinceReferenceDate:
                                 (NSTimeInterval)seconds
- (id)initWithTimeIntervalSince1970:(NSTimeInterval)seconds
```

正如该类的文档所述的那样，在上面几个初始化方法中，“initWithTimeIntervalSinceReferenceDate:”是全能初始化方法。也就是说，其余的初始化方法都要调用它。于是，只有在全能初始化方法中，才会存储内部数据。这样的话，当底层数据存储机制改变时，只需修改此方法的代码就好，无须改动其他初始化方法。

比如说，要编写一个表示矩形的类。其接口可以这样写：

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

@interface EOCRectangle : NSObject
@property (nonatomic, assign, readonly) float width;
@property (nonatomic, assign, readonly) float height;
@end
```

⊖ 常译为“指定初始化方法”。为明确“designated”一词之含义，译文将其译成“全能”。——译者注

根据第 18 条中的建议，我们把属性声明为只读。不过这样一来，外界就无法设置 Rectangle 对象的属性了。开发者可能会提供初始化方法以设置这两个属性：

```
- (id)initWithWidth:(float)width
          andHeight:(float)height
{
    if ((self = [super init])) {
        _width = width;
        _height = height;
    }
    return self;
}
```

可是，如果有人用 [[EOCRectangle alloc]init] 来创建矩形会如何呢？这么做是合乎规则的，因为 EOCRectangle 的超类 NSObject 实现了这个名为 init 的方法，调用完该方法后，全部实例变量都将设为 0（或设置成符合其数据类型且与 0 等价的值）。如果把 alloc 方法分配好的 EOCRectangle 交由此方法来初始化，那么矩形的宽度与高度就是 0，因为全部实例变量都设为 0 了。这也可能正是你想要的效果，不过此时我们一般希望能自己设定默认的宽度与高度值，或是抛出异常，指明本类实例必须用“全能初始化方法”来初始化。也就是说，在 EOCRectangle 这个例子中，应该像下面这样，参照其中一种版本来覆写 init 方法：

```
// Using default values
- (id)init {
    return [self initWithWidth:5.0fandHeight:10.0f];
}

// Throwing an exception
- (id)init {
    @throw [NSException
            exceptionWithName:NSInternalInconsistencyException
             reason:@"Must use initWithWidth:andHeight: instead."
           userInfo:nil];
}
```

请注意，设置默认值的那个 init 方法调用了全能初始化方法。如果采用这个版本来覆写，那么也可以直接在其代码中设置 _width 与 _height 实例变量的值。然而，若是类的底层存储方式变了（比如开发者决定把宽度与高度一起放在某结构体中），则 init 与全能初始化方法设置数据所用的代码就都要修改。在本例这种简单的情况下没有太大问题，但是如果类的初始化方法有很多种，而且待初始化的数据也较为复杂，那么这样做就麻烦得多。很容易就忘了修改其中某个初始化方法，从而导致各初始化方法之间相互不一致。

现在假定要创建名叫 EOCSquare 的类，令其成为 EOCRectangle 的子类。这种继承方式完全合理，不过，新类的初始化方法应该怎么写呢？因为本类表示正方形，所以其宽度与高度必须相等才行。于是，我们可能会像下面这样创建初始化方法：

```
#import "EOCRectangle.h"

@interface EOCSquare : EOCRectangle
- (id)initWithDimension:(float)dimension;
@end

@implementation EOCSquare

- (id)initWithDimension:(float)dimension {
    return [super initWithWidth:dimension andHeight:dimension];
}

@end
```

上述方法就是EOCSquare类的全能初始化方法。请注意，它调用了超类的全能初始化方法。回过头看看EOCRectangle类的实现代码，你就会发现，那个类也调用了其超类的全能初始化方法。全能初始化方法的调用链一定要维系。然而，调用者可能会使用“initWithWidth:andHeight:”或init方法来初始化EOCSquare对象。类的编写者并不希望看到此种情况，因为这样做可能会创建出“宽度”和“高度”不相等的正方形。于是，就引出了类继承时需要注意的一个重要问题：如果子类的全能初始化方法与超类方法的名称不同，那么总应覆写超类的全能初始化方法。在EOCSquare这个例子中，应该像下面这样覆写EOCRectangle的全能初始化方法：

```
- (id)initWithWidth:(float)width andHeight:(float)height {
    float dimension = MAX(width, height);
    return [self initWithDimension:dimension];
}
```

请注意看此方法是如何利用EOCSquare的全能初始化方法来保证对象属性正确的。覆写了这个方法之后，即便使用init来初始化EOCSquare对象，也能照常工作。原因在于，EOCRectangle类覆写了init方法，并以默认值为参数，调用了该类的全能初始化方法。在用init方法初始化EOCSquare对象时，也会这么调用，不过由于“initWithWidth:andHeight:”已经在子类中覆写了，所以实际上执行的是EOCSquare类的这一份实现代码，而此代码又会调用本类的全能初始化方法。因此一切正常，调用者不可能创建出边长不相等的EOCSquare对象。

有时我们不想覆写超类的全能初始化方法，因为那样做没有道理。比方说，现在不想令“initWithWidth:andHeight:”方法以其两参数中较大者作边长来初始化EOCSquare对象；反之，我们认为这是方法调用者自己犯了错误。在这种情况下，常用的办法是覆写超类的全能初始化方法并于其中抛出异常：

```
- (id)initWithWidth:(float)width andHeight:(float)height {
    @throw [NSException
        exceptionWithName:NSInternalInconsistencyException
                   reason:@"Must use initWithDimension: instead."
                 userInfo:nil];
}
```

这样做看起来似乎显得突兀，不过有时却是必需的，因为那种情况下创建出来的对象，其内部数据有可能相互不一致（inconsistent internal data）。如果这么做了，那么在 EOCRectangle 与 EOCSquare 这个例子中，调用 init 方法也会抛出异常，因为 init 方法也得调用“initWithWidth:andHeight:”。此时可以覆写 init 方法，并在其中以合理的默认值来调用“initWithDimension:”方法：

```
- (id)init {
    return [self initWithDimension:5.0f];
}
```

不过，在 Objective-C 程序中，只有当发生严重错误时，才应该抛出异常（参见第 21 条），所以，初始化方法抛出异常乃是不得已之举，表明实例真的没办法初始化了。

有时候可能需要编写多个全能初始化方法。比方说，如果某对象的实例有两种完全不同的创建方式，必须分开处理，那么就会出现这种情况。以 NSCoding 协议为例，此协议提供了“序列化机制”（serialization mechanism），对象可依此指明其自身的编码（encode）及解码（decode）方式。Mac OS X 的 AppKit 与 iOS 的 UIKit 这两个 UI 框架都广泛运用此机制，将对象序列化，并保存至 XML 格式的“NIB”文件中。这些 NIB 文件通常用来存放视图控制器（view controller）及其视图布局。加载 NIB 文件时，系统会在解压缩（unarchiving）的过程中解码视图控制器。NSCoding 协议定义了下面这个初始化方法，遵从该协议者都应实现此方法：

```
- (id)initWithCoder:(NSCoder*)decoder;
```

我们在实现此方法时一般不调用平常所使用的那个全能初始化方法，因为该方法要通过“解码器”（decoder）将对象数据解压缩，所以和普通的初始化方法不同。而且，如果超类也实现了 NSCoding，那么还需调用超类的“initWithCoder:”方法。于是，子类中有不止一个初始化方法调用了超类的初始化方法，因此，严格地说，在这种情况下出现了两个全能初始化方法。

具体到 EOCRectangle 这个例子上，其代码就是：

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

@interface EOCRectangle : NSObject<NSCoding>
@property (nonatomic, assign, readonly) float width;
@property (nonatomic, assign, readonly) float height;
- (id)initWithWidth:(float)width
          andHeight:(float)height;
@end

@implementation EOCRectangle

// Designated initializer
- (id)initWithWidth:(float)width
          andHeight:(float)height
```

```
{
    if ((self = [super init])) {
        _width = width;
        _height = height;
    }
    return self;
}

// Superclass's designated initializer
- (id)init {
    return [self initWithWidth:5.0fandHeight:10.0f];
}
// Initializer from NSCoding
- (id)initWithCoder:(NSCoder*)decoder {
    // Call through to super's designated initializer
    if ((self = [super init])) {
        _width = [decoder decodeFloatForKey:@"width"];
        _height = [decoder decodeFloatForKey:@"height"];
    }
    return self;
}

@end
```

请注意，NSCoding协议的初始化方法没有调用本类的全能初始化方法，而是调用了超类的相关方法。然而，若超类也实现了NSCoding，则需改为调用超类的“initWithCoder:”初始化方法。例如，在此情况下，EOCSquare类就得这么写：

```
#import "EOCRectangle.h"

@interface EOCSquare : EOCRectangle
- (id)initWithDimension:(float)dimension;
@end

@implementation EOCSquare

// Designated initializer
- (id)initWithDimension:(float)dimension {
    return [super initWithWidth:dimension andHeight:dimension];
}

// Superclass designated initializer
- (id)initWithWidth:(float)width andHeight:(float)height {
    float dimension = MAX(width, height);
    return [self initWithDimension:dimension];
}

// NSCoding designated initializer
- (id)initWithCoder:(NSCoder*)decoder {
```

```
    if ((self = [super initWithCoder:decoder])) {
        // EOCSquare's specific initializer
    }
    return self;
}

@end
```

每个子类的全能初始化方法都应该调用其超类的对应方法，并逐层向上，实现“initWithCoder:”时也要这样，应该先调用超类的相关方法，然后再执行与本类有关的任务。这样编写出来的 EOCSquare 类就完全遵守 NSCoding 协议了（fully NSCoding compliant）。如果编写“initWithCoder:”方法时没有调用超类的同名方法，而是调用了自制的初始化方法，或是超类的其他初始化方法，那么 EOCRectangle 类的“initWithCoder:”方法就没机会执行，于是，也就无法将 _width 及 _height 这两个实例变量解码了。

要点

- 在类中提供一个全能初始化方法，并于文档里指明。其他初始化方法均应调用此方法。
- 若全能初始化方法与超类不同，则需覆写超类中的对应方法。
- 如果超类的初始化方法不适用于子类，那么应该覆写这个超类方法，并在其中抛出异常。

第 17 条：实现 description 方法

调试程序时，经常需要打印并查看对象信息。一种办法是编写代码把对象的全部属性都输出到日志中。不过最常用的做法还是像下面这样：

```
NSLog(@"object = %@", object);
```

在构建需要打印到日志的字符串时，object 对象会收到 description 消息，该方法所返回的描述信息将取代“格式字符串”（format string）里的“%@”。比方说，object 是个数组，若用下列代码打印其信息：

```
NSArray *object = @[@"A string", @(123)];
NSLog(@"object = %@", object);
```

则会输出：

```
object = (
  "A string",
  123
)
```

然而，如果在自定义的类上这么做，那么输出的信息却是下面这样：

```
object = <EOCPerson: 0x7fd9a1600600>
```

与object为数组时所输出的信息相比，上面这种内容不太有用。除非在自己的类里覆写description方法，否则打印信息时就会调用NSObject类所实现的默认方法。此方法定义在NSObject协议里，不过NSObject类也实现了它。因为NSObject并不是唯一的“根类”，所以许多方法都要定义在NSObject协议里。比方说，NSProxy也是一个遵从了NSObject协议的“根类”。由于description等方法定义在NSObject协议里，因此像NSProxy这种“根类”及其子类也必须实现它们。如前所见，这些实现好的方法并没有打印出较为有用的内容，只不过是输出了类名和对象的内存地址。只有在你想判断两指针是否真的指向同一对象时，这种信息才有用处。除此之外，再也看不出其他有用的内容了。我们想打印出来的对象信息应该比这更多才对。

要想输出更为有用的信息也很简单，只需覆写description方法并将描述此对象的字符串返回即可。例如，有下面这个代表个人信息的类：

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

@interface EOCPerson : NSObject

@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;

- (id)initWithFirstName:(NSString*)firstName
               lastName:(NSString*)lastName;
@end

@implementation EOCPerson

- (id)initWithFirstName:(NSString*)firstName
               lastName:(NSString*)lastName
{
    if ((self = [super init])) {
        _firstName = [firstName copy];
        _lastName = [lastName copy];
    }
    returnself;
}

@end
```

该类的description方法通常可以这样实现：

```
- (NSString*)description {
    return [NSStringstringWithFormat:@"<%@: %p, \"%@ %@\">",
            [self class], self, _firstName, _lastName];
}
```

假如按上面的代码来写，那么EOCPerson对象就会输出如下格式的信息：

```
EOCPerson *person = [[EOCPersonalloc]
                     initWithFirstName:@"Bob"
                              lastName:@"Smith"];
NSLog(@"person = %@", person);
// Output:
// person = <EOCPerson: 0x7fb249c030f0, "Bob Smith">
```

这样就比覆写之前所输出的信息更加清楚，也更为有用了。笔者建议：在新实现的 description 方法中，也应该像默认的实现那样，打印出类的名字和指针地址，因为这些内容有时也许会用到。不过大家刚才也看到了，NSArray 类的对象就没有打印这两项内容。显然，在实现 description 方法时，没有固定规则可循，应根据当前对象来决定在 description 方法里打印何种信息。

有个简单的办法，可以在 description 中输出很多互不相同的信息，那就是借助 NSDictionary 类的 description 方法。此方法输出的信息的格式如下：

```
{
    key: value;
    foo: bar;
}
```

在自定义的 description 方法中，把待打印的信息放到字典里面，然后将字典对象的 description 方法所输出的内容包含在字符串里并返回，这样就可以实现精简的信息输出方式了。例如，下面这个类表示某地点的名称和地理坐标（纬度与经度）：

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

@interface EOCLocation : NSObject
@property (nonatomic, copy, readonly) NSString *title;
@property (nonatomic, assign, readonly) float latitude;
@property (nonatomic, assign, readonly) float longitude;
- (id)initWithTitle:(NSString*)title
           latitude:(float)latitude
          longitude:(float)longitude;
@end
@implementation EOCLocation
- (id)initWithTitle:(NSString*)title
           latitude:(float)latitude
          longitude:(float)longitude
{
      if ((self = [super init])) {
            _title = [title copy];
            _latitude = latitude;
            _longitude = longitude;
       }
       returnself;
}
@end
```

要是这个类的description方法能够打印出地名和经纬度就好了。我们可以像下面这样编写description方法，用NSDictionary来实现此功能：

```
- (NSString*)description {
    return [NSStringstringWithFormat:@"<%@: %p, %@>",
            [self class],
            self,
            @{@"title":_title,
              @"latitude":@(_latitude),
              @"longitude":@(_longitude)}
           ];
}
```

输出的信息格式为：

```
location = <EOCLocation: 0x7f98f2e01d20, {
    latitude = "51.506";
    longitude = 0;
    title = London;
}>
```

这比仅仅输出指针和类名要有用多了，而且对象中的每条属性都能打印得很好。也可以在格式字符串中直接为每个实例变量留好位置，然后逐个打印出来，不过，用NSDictionary来实现此功能可以令代码更易维护：如果以后还要向类中新增属性，并且要在description方法中打印，那么只需修改字典内容即可。

NSObject协议中还有个方法要注意，那就是debugDescription，此方法的用意与description非常相似。二者区别在于，debugDescription方法是开发者在调试器（debugger）中以控制台命令打印对象时才调用的。在NSObject类的默认实现中，此方法只是直接调用了description。以EOCPerson类为例，我们在创建实例所用的代码后面插入断点，然后通过调试器（假设使用LLDB）运行程序，使之暂停于此：

```
EOCPerson *person = [[EOCPersonalloc]
                        initWithFirstName:@"Bob"
                                 lastName:@"Smith"];
NSLog(@"person = %@", person);
// Breakpoint here
```

当程序运行到断点时，开发者就可以向调试控制台里输入命令了。LLDB的“po”命令可以完成对象打印（print-object）工作，其输出如下：

```
EOCTest[640:c07] person = <EOCPerson: 0x712a4d0, "Bob Smith">
(lldb) po person
(EOCPerson *) $1 = 0x0712a4d0 <EOCPerson: 0x712a4d0, "Bob Smith">
```

请注意，控制台中的“(EOCPerson *) $1=0x712a4d0”是由调试器所添加的，其后的内容才是由debugDescription所返回的信息。

你也许只想把人名放在EOCPerson对象的普通描述信息中，而把更详尽的内容放在调试

所用的描述信息里，此时可用下列代码实现这两个方法：

```
- (NSString*)description {
    return [NSString stringWithFormat:@"%@ %@",
            _firstName, _lastName];
}
- (NSString*)debugDescription {
    return [NSString stringWithFormat:@"<%@: %p, \"%@ %@\">",
            [self class], self, _firstName, _lastName];
}
```

写好之后，再把刚才的程序码运行一遍，这次 po 命令所打印出来的对象信息如下所示：

```
EOCTest[640:c07] person = Bob Smith
(lldb) po person
(EOCPerson *) $1 = 0x07117fb0 <EOCPerson: 0x7117fb0, "Bob Smith">
```

你可能不想把类名与指针地址这种额外内容放在普通的描述信息里，但是却希望调试的时候能够很方便地看到它们，在此情况下，就可以使用这种输出方式来实现。Foundation 框架的 NSArray 类就是这么做的。例如：

```
NSArray *array = @[@"Effective Objective-C 2.0", @(123), @(YES)];
NSLog(@"array = %@", array);
// Breakpoint here
```

运行上述程序码，待其停在断点处，然后用 po 命令打印数组对象，就可以看到如下信息：

```
EOCTest[713:c07] array = (
    "Effective Objective-C 2.0",
    123,
    1
)
(lldb) po array
(NSArray *) $1 = 0x071275b0 <__NSArrayI 0x71275b0>(
Effective Objective-C 2.0,
123,
1
)
```

要点

- 实现 description 方法返回一个有意义的字符串，用以描述该实例。
- 若想在调试时打印出更详尽的对象描述信息，则应实现 debugDescription 方法。

第 18 条：尽量使用不可变对象

设计类的时候，应充分运用属性来封装数据（参见第 6 条）。而在使用属性时，则可将其声明为“只读”（read-only）。默认情况下，属性是“既可读又可写的”（read-write），这样设计

出来的类都是“可变的”（mutable）。不过，一般情况下我们要建模的数据未必需要改变。比方说，某数据所表示的对象源自一项只读的网络服务（web service），里面可能包含一系列需要显示在地图上的相关点，像这种对象就没必要改变其内容。即使修改了，新数据也不会推送回服务器。正如第8条所述，如果把可变对象（mutable object）放入collection之后又修改其内容，那么很容易就会破坏set的内部数据结构，使其失去固有的语义。因此，笔者建议大家尽量减少对象中的可变内容。

具体到编程实践中，则应该尽量把对外公布出来的属性设为只读，而且只在确有必要时才将属性对外公布。例如，要编写一个类来处理地图上的景点，这些点的数据通过某个网络服务来获取。一开始写出来的代码也许是这样：

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

@interface EOCPointOfInterest : NSObject

@property (nonatomic, copy) NSString *identifier;
@property (nonatomic, copy) NSString *title;
@property (nonatomic, assign) float latitude;
@property (nonatomic, assign) float longitude;

- (id)initWithIdentifier:(NSString*)identifier
                   title:(NSString*)title
                latitude:(float)latitude
               longitude:(float)longitude;
@end
```

对象中的值都经由网络服务获取，在与网络服务通信的过程中，以identifier来指代相关的景点。用网络服务所提供的数据创建好某个点之后，就无须改动其值了。如果用其他编程语言来写，则可能会通过相应的机制创建出私有的实例变量，这些变量只有get存取方法，没有set存取方法。然而使用Objective-C编程时则会简单许多，根本无须考虑私有变量。

为了将EOCPointOfInterest做成不可变的类，需要把所有属性都声明为readonly：

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

@interface EOCPointOfInterest : NSObject

@property (nonatomic, copy, readonly) NSString *identifier;
@property (nonatomic, copy, readonly) NSString *title;
@property (nonatomic, assign, readonly) float latitude;
@property (nonatomic, assign, readonly) float longitude;
- (id)initWithIdentifier:(NSString*)identifier
                   title:(NSString*)title
                latitude:(float)latitude
               longitude:(float)longitude;

@end
```

如果有人试着改变属性值，那么编译的时候就会报错。对象中的属性值可以读出，但是无法写入，这就能保证 EOCPointOfInterest 中的各个数据之间总是相互协调的。于是，开发者在使用对象时就能肯定其底层数据不会改变。因此，对象本身的数据结构也就不可能出现不一致的现象。比如说，在将 EOCPointOfInterest 对象显示到地图视图上时，这些点的底层经纬度数据不会变动。

读者也许会问，既然这些属性都没有设置方法（setter），那为何还要指定内存管理语义呢？如果不指定，采用默认的语义也可以：

```
@property (nonatomic, readonly) NSString *identifier;
@property (nonatomic, readonly) NSString *title;
@property (nonatomic, readonly) float latitude;
@property (nonatomic, readonly) float longitude;
```

虽说如此，我们还是应该在文档里指明实现所用的内存管理语义，这样的话，以后想把它变为可读写的属性时，就会简单一些。

有时可能想修改封装在对象内部的数据，但是却不想令这些数据为外人所改动。这种情况下，通常做法是在对象内部将 readonly 属性重新声明为 readwrite。当然，如果该属性是 nonatomic 的，那么这样做可能会产生“竞争条件”（race condition）[⊖]。在对象内部写入某属性时，对象外的观察者也许正读取该属性。若想避免此问题，我们可以在必要时通过“派发队列”（dispatch queue，参见第 41 条）等手段，将（包括对象内部的）所有数据存取操作都设为同步操作。

将属性在对象内部重新声明为 readwrite 这一操作可于“ class-continuation 分类”（参见第 27 条）中完成，在公共接口中声明的属性可于此处重新声明，属性的其他特质必须保持不变，而 readonly 可扩展为 readwrite。以 EOCPointOfInterest 为例，其“ class-continuation 分类”可以这样写：

```
#import "EOCPointOfInterest.h"

@interface EOCPointOfInterest ()
@property (nonatomic, copy, readwrite) NSString *identifier;
@property (nonatomic, copy, readwrite) NSString *title;
@property (nonatomic, assign, readwrite) float latitude;
@property (nonatomic, assign, readwrite) float longitude;
@end

@implementation EOCPointOfInterest

/* ... */

@end
```

现在，只能于 EOCPointOfInterest 实现代码内部设置这些属性值了。其实更准确地说，在对象外部，仍然能通过“键值编码”（Key-Value Coding，KVC）技术设置这些属性值，比

⊖ 又名“竞态条件”。——译者注

如说，可以像下面这样，使用“setValue:forKey:”方法来修改：

```
[pointOfInterest setValue:@"abc"forKey:@"identifier"];
```

这样做可以改动identifier属性，因为KVC会在类里查找“setIdentifier:”方法，并借此修改此属性。即便没有于公共接口中公布此方法，它也依然包含在类里。不过，这样做等于违规地绕过了本类所提供的API，要是开发者使用这种“杂技代码”（hack）的话，那么得自己来应对可能出现的问题。

有些“爱用蛮力的”（brutal）程序员甚至不通过“设置方法”，而是直接用类型信息查询功能查出属性所对应的实例变量在内存布局中的偏移量，以此来人为设置这个实例变量的值。这样做比绕过本类的公共API还要不合规范。从技术上来讲，即便某个类没有对外公布“设置方法”，也依然可以想办法修改对应的属性，然而，不应该因为这个原因而忽视笔者所提的建议，大家还是要尽量编写不可变的对象。

在定义类的公共API时，还要注意一件事情：对象里表示各种collection的那些属性究竟应该设成可变的，还是不可变的。例如，我们用某个类来表示个人信息，该类里还存放了一些引用，指向此人的诸位朋友。你可能想把这个人的全部朋友都放在一个“列表”（list）里，并将其做成属性。假如开发者可以添加或删除此人的朋友，那么这个属性就需要用可变的set来实现。在这种情况下，通常应该提供一个readonly属性供外界使用，该属性将返回不可变的set，而此set则是内部那个可变set的一份拷贝。比方说，下面这段代码就能够实现出这样一个类：

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

@interface EOCPerson : NSObject

@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;
@property (nonatomic, strong, readonly) NSSet *friends;

- (id)initWithFirstName:(NSString*)firstName
            andLastName:(NSString*)lastName;
- (void)addFriend:(EOCPerson*)person;
- (void)removeFriend:(EOCPerson*)person;

@end
// EOCPerson.m
#import "EOCPerson.h"

@interface EOCPerson ()
@property (nonatomic, copy, readwrite) NSString *firstName;
@property (nonatomic, copy, readwrite) NSString *lastName;
@end

@implementation EOCPerson {
    NSMutableSet *_internalFriends;
```

```
}

- (NSSet*)friends {
    return [_internalFriends copy];
}

- (void)addFriend:(EOCPerson*)person {
    [_internalFriends addObject:person];
}

- (void)removeFriend:(EOCPerson*)person {
    [_internalFriends removeObject:person];
}

- (id)initWithFirstName:(NSString*)firstName
            andLastName:(NSString*)lastName {
    if ((self = [super init])) {
        _firstName = firstName;
        _lastName = lastName;
        _internalFriends = [NSMutableSet new];
    }
    return self;
}
@end
```

也可以用 NSMutableSet 来实现 friends 属性，令该类的用户不借助“addFriend:”与“removeFriend:”方法而直接操作此属性。但是，这种过分解耦（decouple）数据的做法很容易出 bug。比方说，在添加或删除朋友时，EOCPerson 对象可能还要执行其他相关操作，若是采用这种做法，那就等于直接从底层修改了其内部用于存放朋友对象的 set。在 EOCPerson 对象不知情时，直接从底层修改 set 可能会令对象内的各数据之间互不一致。

说到这里，笔者还要强调：不要在返回的对象上查询类型以确定其是否可变。比方说，你正在使用一个包含 EOCPerson 类的库来开发程序。为了省事，该库的开发者可能并没有将内部那个可变的 set 拷贝一份再返回，而是直接返回了可变的 set。这样做也算合理，因为 set 可能很大，拷贝起来太耗时了。返回 NSMutableSet 也合乎语法，因为该类是 NSSet 的子类，于是，你可能会像这样来使用 EOCPerson：

```
EOCPerson *person = /* ... */;
NSSet *friends = person.friends;
if ([friends isKindOfClass:[NSMutableSet class]]) {
    NSMutableSet *mutableFriends = (NSMutableSet*)friends;
    /* mutate the set */
}
```

然而笔者要说：大家应该竭力避免这种做法。在你与 EOCPerson 类之间的约定（contract）里，并没有提到实现 friends 所用的那个 NSSet 一定是可变的，因此不应像这样使用类型信息查询功能来编码。这依然说明：开发者或许不宜从底层直接修改对象中的数据。所以，不要假设这个 NSSet 就一定能直接修改。

要点

- 尽量创建不可变的对象。
- 若某属性仅可于对象内部修改，则在“class-continuation分类”中将其由readonly属性扩展为readwrite属性。
- 不要把可变的collection作为属性公开，而应提供相关方法，以此修改对象中的可变collection。

第19条：使用清晰而协调的命名方式

类、方法及变量的命名是Objective-C编程的重要环节。新手通常会觉得这门语言很繁琐，因为其语法结构使得代码读起来和句子一样。名称中一般都带有“in”、“for”、“with”等介词，其他编程语言则很少使用这些它们认为多余的字眼。以下面这段代码为例：

```
NSString *text = @"The quick brown fox jumped over the lazy dog";
NSString *newText =
    [text stringByReplacingOccurrencesOfString:@"fox"
                                    withString:@"cat"];
```

此代码用了比较啰嗦的方式来描述一个看上去如此简单的表达式。对于执行替换操作的那个方法，其名字居然有48个字符长。不过这样做的好处是，代码读起来像日常语言里的句子：

> “将文本中出现的‘fox’字符串替换为‘cat’字符串，并返回替换后的新字符串。”
>
> （Take text and give me a new string by replacing the occurrences of the string ‘fox’ with the string ‘cat’.）

这个句子准确描述了开发者想做的事。在命名不像Objective-C这般繁复的语言中，类似的程序可能会写成这样：

```
string text = "The quick brown fox jumped over the lazy dog";
string newText = text.replace("fox", "cat");
```

这样写有个问题，就是text.replace()的两个参数到底按何种顺序解读。是“fox”为“cat”所替换，还是“cat”为“fox”所替换呢？还有个疑问：replace函数是把所有出现的字符串都替换掉呢，还是只替换第一次找见的那个？其名称没能清楚地表达出这两个意思。而Objective-C的命名方式虽然长一点，但是却非常清晰。

读者也会注意到，方法与变量名使用了“驼峰式大小写命名法”（camel casing）——以小写字母开头，其后每个单词首字母大写。类名也用驼峰命名法，不过其首字母要大写，而且前面通常还有两三个前缀字母（参见第15条）。在编写Objective-C代码时，大家一般都使用这种命名方式。如果你愿意，也可使用自己的风格来命名，不过按照驼峰命名法写出来的代码更容易为其他Objective-C开发者所接受。

方法命名

你要是写过 C++ 或 Java 代码的话，应该会习惯那种较为简省的函数名，在那种命名方式下，若想知道每个参数的用途，就得查看函数原型。这会令代码难于读懂：为了明白函数用法，你必须经常回过头参照其原型。比方说，要写一个表示矩形的类。用 C++ 代码可以这样定义此类：

```
class Rectangle {
public:
    Rectangle(float width, float height);
    float getWidth();
    float getHeight();
private:
    float width;
    float height;
};
```

不熟悉 C++ 也没关系，你只要知道这个类包含名为 width 及 height 的两个实例变量就好。若想创建该类的实例，只有一种办法，就是以矩形尺寸为参数，调用其“构造器”（constructor）。宽度与高度都有对应的存取方法。可以用下面这行代码来创建该类的实例：

```
Rectangle *aRectangle = new Rectangle(5.0f, 10.0f);
```

回顾这行代码时，并不能一下子看出 5.0f 和 10.0f 表示什么。你可能觉得这两个参数是矩形尺寸，可是到底宽度在先还是高度在先呢？要想确定这一点，还得去查函数定义才行。

Objective-C 语言就不会有这个问题了，因为其方法名可以起得更长一些。熟悉 C++ 的人可能会像下面这样把 Rectangle 改写为等价的 Objective-C 代码：

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

@interface EOCRectangle : NSObject

@property (nonatomic, assign, readonly) float width;
@property (nonatomic, assign, readonly) float height;

- (id)initWithSize:(float)width :(float)height;

@end
```

写这个类的人显然知道，在 Objective-C 语言中，和 C++ 构造器等效的东西是 init- 系列方法，于是将其命名为“ initWithSize::”。这看上去很奇怪，你也许觉得语法有误，第二个冒号前面怎么没有字呢？实际上语法完全没错，之所以会觉得写错了，是因为它和改写前的 C++ 构造器有着相同的问题：使用此类的开发者还是不清楚每个变量的含义：

```
EOCRectangle *aRectangle =
    [[EOCRectangle alloc] initWithSize:5.0f :10.0f];
```

下面这种命名方式就要好很多：

```
- (id)initWithWidth:(float)width andHeight:(float)height;
```

这么写是很长，然而这次绝对不会混淆每个变量的含义了：

```
EOCRectangle *aRectangle =
    [[EOCRectanglealloc] initWithWidth:5.0f andHeight:10.0f];
```

虽说使用长名字可令代码更为易读，但是Objective-C新手还是难于习惯这种详尽的方法命名风格。不要吝于使用长方法名。把方法名起得稍微长一点，可以保证其能准确传达出方法所执行的任务。然而方法名也不能长得太过分了，应尽量言简意赅。

以EOCRectangle类为例。好的方法名应该像这样：

```
- (EOCRectangle*)unionRectangle:(EOCRectangle*)rectangle
- (float)area
```

而下面这种命名方式则不好[⊖]：

```
- (EOCRectangle*)union:(EOCRectangle*)rectangle // Unclear
- (float)calculateTheArea // Too verbose
```

清晰的方法名从左至右读起来好似一段文章。并不是说非得按照那些命名规则来给方法起名，不过这样做可以令代码变得更好维护，而且也能使其他人更易读懂。

NSString这个类就展示了一套良好的命名习惯。下面列出几个方法及其命名缘由：

- + string

  工厂方法（factory method），用于创建新的空字符串。方法名清晰地描述了返回值的类型。

- + stringWithString

  工厂方法，根据某字符串创建出与之内容相同的新字符串。与创建空字符串所用的那个工厂方法一样，方法名的第一个单词也指明了返回类型。

- + localizedStringWithFormat:

  工厂方法，根据特定格式创建出新的“本地化字符串”（localized string）。返回值类型是方法名的第二个单词（string），因为其前面还有个修饰语（localized）用来描述其逻辑含义。此方法的返回值依然是“字符串”（string），只不过是一种经过本地化处理的特殊字符串。

- - lowercaseString

  把字符串中的大写字母都转为小写。该方法不会修改接收此消息的字符串本身，而是要新创建一个字符串，此做法也符合方法名中应该包含返回值类型这一规范，然而描述返回值类型的单词（string）前面还有个定语（lowercase）。

- - intValue

  将字符串解析为整数。由于返回值是int，所以方法名以这个词开头。通常情况下我

⊖ 第一个方法名不清晰，第二个太啰嗦。——译者注

们不会像这样来简写返回值的类型，比如 string 不简写为 str，然而由于“Integer”（整数）一词的简称“int”本身就是返回值的类型名称，所以此处这么做是合理的。为了把方法名凑足两个词，所以加了后缀“Value”。只有一个词的名字通常用来表示属性。由于 int 不是字符串对象的属性，所以要加 Value 以限定其含义。

- - length
  获取字符串长度（也就是其字符个数）。这个方法只有一个词，因为实际上 length 也是字符串的一个属性。这个属性可能不是由实例变量来实现的，然而即便如此，它也依然是字符串中的属性。此方法若是命名为 stringLength 就不好了。string 一词多余，因为该方法的接收者肯定是个字符串。
- - lengthOfBytesUsingEncoding:
  若字符串是以给定的编码格式（ASCII、UTF8、UTF16 等）来编码的，则返回其字节数组的长度。此方法与 length 相似，所以其命名原因也和刚才说的一样。此外，该方法还需一个参数。该参数紧跟着方法名中描述其类型的那个名词（encoding）。
- - getCharacters:range:
  获取字符串中给定范围内的字符。其他语言里的获取方法也许会以 get 开头，但 Objective-C 中一般不这么做，然而此处例外，该方法用 get 作其前缀。原因在于，调用此方法时，要在其首个参数中传入数组，而该方法所获取的字符正是要放到这个数组里面。此方法的完整签名为：
- - (void)getCharacters:(unichar*)buffer range:(NSRange)aRange
  首个参数 buffer 应该指向一个足够大的数组，以便容纳所请求范围内的那些字符。此方法要通过其参数来返回（这种参数通常称为“输出参数”（out-parameter）），而不通过返回值来返回，从内存管理的角度看，这样做更好。所有内存管理事宜均由方法调用者处理，而不是先在此方法中创建一个数组，然后再由调用者释放。第二个参数前有个描述其类型的名词（range），如果还有其他参数，也应该在方法名中提到其类型。有时参数名前面还会加介词，例如，此方法可以命名为“getCharacters:inRange:”。当需要特别强调众参数中的某一个时，通常会这样命名。
- - hasPrefix:
  判断本字符串是否以另一个字符串开头。由于返回值是 Boolean 类型，所以为了读起来像个句子，这种方法的名称中通常都包括 has（“是否有”）一词。例如：

```
[@"Effective Objective-C" hasPrefix:@"Effective"] == YES
```

  要是把方法名直接写成“prefix:”，读起来就不这么顺了。反之，若将其叫成“isPrefixedWith:”，则听上去冗长而别扭。
- - isEqualToString:
  判断两字符串是否相等。其返回值和“hasPrefix:”一样，都是 Boolean 型，为了便于述说，方法名用 is 开头。还有个地方也会用到 is 这个前缀词，那就是 Boolean

型的属性。比方说，有个属性叫做enabled，则其两个存取方法应该分别起名为“setEnabled:”与isEnabled。

给方法命名时的注意事项可总结成下面几条规则。

- 如果方法的返回值是新创建的，那么方法名的首个词应是返回值的类型，除非前面还有修饰语，例如localizedString。属性的存取方法不遵循这种命名方式，因为一般认为这些方法不会创建新对象，即便有时返回内部对象的一份拷贝，我们也认为那相当于原有的对象。这些存取方法应该按照其所对应的属性来命名。
- 应该把表示参数类型的名词放在参数前面。
- 如果方法要在当前对象上执行操作，那么就应该包含动词；若执行操作时还需要参数，则应该在动词后面加上一个或多个名词。
- 不要使用str这种简称，应该用string这样的全称。
- Boolean属性应加is前缀。如果某方法返回非属性的Boolean值，那么应该根据其功能，选用has或is当前缀。
- 将get这个前缀留给那些借由“输出参数”来保存返回值的方法，比如说，把返回值填充到“C语言式数组”（C-style array）里的那种方法就可以使用这个词做前缀。

类与协议的命名

应该为类与协议的名称加上前缀，以避免命名空间冲突（参见第15条），而且应该像给方法起名时那样把词句组织好，使其从左至右读起来较为通顺。例如，在NSArray的子类中，有一个用于表示可变数组的类，叫做NSMutableArray，mutable这个词放在array前面，用以表明这是一种特殊的array（数组）。

下面以iOS的UI库UIKit为例，演示类与协议的命名惯例：

- UIView（类）

  所有“视图”（View）均继承于此类。视图是构造用户界面的基本单元，它们负责绘制按钮、文本框（text field）、表格等控件。这个类的名字无须解释即可自明其意（self-explanatory），开头的两个字母“UI”是UIKit框架的通用前缀。

- UIViewController（类）

  视图类（UIView）负责绘制视图，然而却不负责指定视图里面应该显示的内容。这项工作由本类，也就是“视图控制器”（view controller）来完成。其名称从左至右读起来很顺。

- UITableView（类）

  这是一种特殊类型的视图，可以显示表格中的一系列条目。所以，它在超类（UIView）名称中的View一词前面加了Table这个修饰词，用以和其他类型的视图相区隔。在超类名称前加修饰语是一种常用的命名惯例。本类也可以叫做UITable，不过这个名字无法完整传达出“视图”这个概念。开发者必须查看接口声明方能确定这一点。比方说，想创建一个专门用来显示图像的表格视图，那么就可以将这个继

承自 UITableView 的子类命名为 EOCImageTableView。不过这时要加上自己的前缀 EOC，而不是沿用超类的前缀 UI（UIKit 框架中的类以 UI 为前缀）。这么做的原因在于，你不应该把自己的类放到其他框架的命名空间里面，那些框架以后也许会新建同名的类。

- UITableViewController（类）

  正如 UITableView 是一种特殊的 view（视图）一样，UITableViewController 也是一种特殊的 view controller（视图控制器），它专门用于控制表格视图。因此，其命名方式与 UITableView 类似。

- UITableViewDelegate（协议）

  此协议定义了表格视图与其他对象之间的通信接口，命名时，把定义“委托接口”（delegate interface）的那个类名（UITableView）放在前面，后面加上 Delegate 一词，这样读起来顺口。（第 23 条详述了“委托模式”（Delegate pattern）。）

说了这么多，其中最重要的一点就是，命名方式应该协调一致。而且，如果要从其他框架中继承子类，那么务必遵循其命名惯例。比方说，要从 UIView 类中继承自定义的子类，那么类名末尾的词必须是 View。同理，若要创建自定义的委托协议，则其名称中应该包含委托发起方的名称，后面再跟上 Delegate 一词。如果能坚持这种命名习惯，那么在稍后回顾自己的代码或他人使用你所写的代码时，很容易就能理解其含义。

要点

- 起名时应遵从标准的 Objective-C 命名规范，这样创建出来的接口更容易为开发者所理解。
- 方法名要言简意赅，从左至右读起来要像个日常用语中的句子才好。
- 方法名里不要使用缩略后的类型名称。
- 给方法起名时的第一要务就是确保其风格与你自己的代码或所要集成的框架相符。

第 20 条：为私有方法名加前缀

一个类所做的事情通常都要比从外面看到的更多。编写类的实现代码时，经常要写一些只在内部使用的方法。笔者建议，应该为这种方法的名称加上某些前缀，这有助于调试，因为据此很容易就能把公共方法和私有方法区别开。

为私有方法名加前缀还有个原因，就是便于修改方法名或方法签名。对于公共方法来说，修改其名称或签名之前要三思，因为类的公共 API 不便随意改动。如果改了，那么使用这个类的所有开发者都必须更新其代码才行。而对于内部方法来说，若要修改其名称或签名，则只需同时修改本类内部的相关代码即可，不会影响到面向外界的那些 API。用前缀把私有方法标出来，这样很容易就能看出哪些方法可以随意修改，哪些不应轻易改动。

具体使用何种前缀可根据个人喜好来定，其中最好包含下划线与字母 *p*。笔者喜欢用 p_

作为前缀，*p* 表示“ private”（私有的），而下划线则可以把这个字母和真正的方法名区隔开。下划线后面的部分按照常用的驼峰法来命名即可，其首字母要小写。例如，包含私有方法的EOCObject类可以这样写：

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

@interface EOCObject : NSObject
- (void)publicMethod;
@end

@implementation EOCObject

- (void)publicMethod {
    /* ... */
}

- (void)p_privateMethod {
    /* ... */
}

@end
```

与公共方法不同，私有方法不出现在接口定义中。有时可能要在“ class-continuation 分类”（参见第27条）里声明私有方法，然而最近修订的编译器已经不要求在使用方法前必须先行声明了。所以说，私有方法一般只在实现的时候声明。

如果写过C++或Java代码，你可能就会问了：为什么要这样做呢？直接把方法声明成私有的不就好了吗？ Objective-C语言没办法将方法标为私有。每个对象都可以响应任意消息（参见第12条），而且可在运行期检视某个对象所能直接响应的消息（参见第14条）。根据给定的消息查出其对应的方法，这一工作要在运行期才能完成（参见第11条），所以Objective-C中没有那种约束方法调用的机制用以限定谁能调用此方法、能在哪个对象上调用此方法以及何时能调用此方法。开发者会在命名惯例中体现出“私有方法”等语义。新手也许不适应这一点，但是必须用心领悟Objective-C语言这种强大的动态特性。想掌握其动态特性，确实得花大功夫，不过培养良好的命名习惯也是一条成功之道。

苹果公司喜欢单用一个下划线作私有方法的前缀。你或许也想照着苹果公司的办法只拿一个下划线作前缀，这样做可能会惹来大麻烦：如果从苹果公司提供的某个类中继承了一个子类，那么你在子类里可能会无意间覆写了父类的同名方法。鉴于此，苹果公司在文档中说，开发者不应该单用一个下划线做前缀。不能将方法限定于某个范围内，这也许是Objective-C的缺点，然而作为“动态方法派发系统”（dynamic method dispatch system，参见第11条）这个强大组件的一部分，此特性也带来了诸多好处。

你或许觉得刚才提到的那种情况不太常见，其实未必。例如，要在iOS应用程序中创建一个视图控制器，就得编写UIViewController的子类。自定义的视图控制器里可能保存着许

多状态信息。你可能想编写一个方法，当视图出现在屏幕上时，可经由此方法把控制器里的所有状态都重置一遍。于是，该方法的实现代码也许会写成这样：

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

@interface EOCViewController : UIViewController
@end

@implementation EOCViewController
- (void)_resetViewController {
    // Reset state and views
}
@end
```

可问题是，UIViewController 类本身其实已经实现了一个名叫 _resetViewController 的方法了！如果这样写的话，那么所有调用都将执行子类中的这个方法，本来该调用超类方法的地方现在调用的却是 EOCViewController 中覆写过的这个版本。由于超类中的同名方法并未对外公布，所以除非深入研究这个库，否则你根本不会察觉到自己在无意间覆写了这个方法。这毕竟是个用下划线开头的私有方法，所以没有对外公布也是合理的。由于超类方法永远不可能执行，所以这个视图控制器的行为会很奇怪，到时你可能会纳闷：为什么子类的这个方法调用得这么频繁呢，按道理不应该执行这么多次呀？

总之，在确定使用了前缀的情况下，如果子类所继承的那个类既不在苹果公司的框架中，也不在你自己的项目中，而是来自别的框架，那么除非该框架在文档中明示，否则你无法知道其私有方法所加的前缀是什么。此时可以把自己一贯使用的类名前缀（参见第 15 条）用作子类私有方法的前缀，这样能有效避免重名问题。同时还应该考虑到其他人会如何从你所写的类中继承子类，这也是私有方法应该加前缀的原因。除非使用一些相当复杂的工具，否则，在没有源代码的情况下，无法知道某个类在其公共接口之外还定义并实现了哪些方法。

要点

- 给私有方法的名称加上前缀，这样可以很容易地将其同公共方法区分开。
- 不要单用一个下划线做私有方法的前缀，因为这种做法是预留给苹果公司用的。

第 21 条：理解 Objective-C 错误模型

当前很多种编程语言都有“异常”（exception）机制，Objective-C 也不例外。写过 Java 代码的程序员应该很习惯于用异常来处理错误。如果你也是这么使用异常的，那现在就把它忘了吧，我们得从头学起。

首先要注意的是，“自动引用计数”（Automatic Reference Counting，ARC，参见第 30 条）

在默认情况下不是“异常安全的”(exception safe)。具体来说，这意味着：如果抛出异常，那么本应在作用域末尾释放的对象现在却不会自动释放了。如果想生成“异常安全”的代码，可以通过设置编译器的标志来实现，不过这将引入一些额外代码，在不抛出异常时，也照样要执行这部分代码。需要打开的编译器标志叫做 -fobjc-arc-exceptions。

即使不用 ARC，也很难写出在抛出异常时不会导致内存泄漏的代码。比方说，设有段代码先创建好了某个资源，使用完之后再将其释放。可是，在释放资源之前如果抛出异常了，那么该资源就不会被释放了：

```
id someResource = /* ... */;
if ( /* check for error */ ) {
    @throw [NSException exceptionWithName:@"ExceptionName"
                                   reason:@"There was an error"
                                 userInfo:nil];
}
[someResource doSomething];
[someResource release];
```

在抛出异常之前先释放 someResource，这样做当然能解决此问题，不过要是待释放的资源有很多，而且代码的执行路径更为复杂的话，那么释放资源的代码就容易写得很乱。此外，代码中加入了新的资源之后，开发者经常会忘记在抛出异常前先把它释放掉。

Objective-C 语言现在所采用的办法是：只在极其罕见的情况下抛出异常，异常抛出之后，无须考虑恢复问题，而且应用程序此时也应该退出。这就是说，不用再编写复杂的“异常安全”代码了。

异常只应该用于极其严重的错误，比如说，你编写了某个抽象基类，它的正确用法是先从中继承一个子类，然后使用这个子类。在这种情况下，如果有人直接使用了这个抽象基类，那么可以考虑抛出异常。与其他语言不同，Objective-C 中没办法将某个类标识为“抽象类”。要想达成类似效果，最好的办法是在那些子类必须覆写的超类方法里抛出异常。这样的话，只要有人直接创建抽象基类的实例并使用它，即会抛出异常：

```
- (void)mustOverrideMethod {
    NSString *reason = [NSStringstringWithFormat:
                        @"%@ must be overridden",
                        NSStringFromSelector(_cmd)];
    @throw [NSException
        exceptionWithName:NSInternalInconsistencyException
                   reason:reason
                 userInfo:nil];
}
```

既然异常只用于处理严重错误（fatal error，致命错误），那么对其他错误怎么办呢？在出现“不那么严重的错误”(nonfatal error，非致命错误）时，Objective-C 语言所用的编程范式为：令方法返回 nil/0，或是使用 NSError，以表明其中有错误发生。比方说，如果初始化方法无法根据传入的参数来初始化当前实例，那么就可以令其返回 nil/0：

```
- (id)initWithValue:(id)value {
    if ((self = [super init])) {
        if ( /* Value means instance can't be created */ ) {
            self = nil;
        } else {
            // Initialize instance
        }
    }
    return self;
}
```

在这种情况下，如果 if 语句发现无法用传入的参数值来初始化当前实例（比如这个方法要求传入的 value 参数必须是 non-nil 的），那么就把 self 设置成 nil，这样的话，整个方法的返回值也就是 nil 了。调用者发现初始化方法并没有把实例创建好，于是便可确定其中发生了错误。

NSError 的用法更加灵活，因为经由此对象，我们可以把导致错误的原因回报给调用者。NSError 对象里封装了三条信息：

- *Error domain*（错误范围，其类型为字符串）
  错误发生的范围。也就是产生错误的根源，通常用一个特有的全局变量来定义。比方说，“处理 URL 的子系统”（URL-handling subsystem）在从 URL 中解析或取得数据时如果出错了，那么就会使用 NSURLErrorDomain 来表示错误范围。
- *Error code*（错误码，其类型为整数）
  独有的错误代码，用以指明在某个范围内具体发生了何种错误。某个特定范围内可能会发生一系列相关错误，这些错误情况通常采用 enum 来定义。例如，当 HTTP 请求出错时，可能会把 HTTP 状态码设为错误码。
- *User info*（用户信息，其类型为字典）
  有关此错误的额外信息，其中或许包含一段“本地化的描述”（localized description），或许还含有导致该错误发生的另外一个错误，经由此种信息，可将相关错误串成一条“错误链”（chain of errors）。

在设计 API 时，NSError 的第一种常见用法是通过委托协议来传递此错误。有错误发生时，当前对象会把错误信息经由协议中的某个方法传给其委托对象（delegate）。例如，NSURLConnection 在其委托协议 NSURLConnectionDelegate 之中就定义了如下方法：

```
- (void)connection:(NSURLConnection *)connection
  didFail WithError:(NSError *)error
```

当 NSURLConnection 出错之后（比如与远程服务器的连接操作超时了），就会调用此方法以处理相关错误。这个委托方法未必非得实现不可：是不是必须处理此错误，可交由 NSURLConnection 类的用户来判断。这比抛出异常要好，因为调用者至少可以自己决定 NSURLConnection 是否回报此错误。

NSError 的另外一种常见用法是：经由方法的“输出参数”返回给调用者。比如像这样：

```
- (BOOL)doSomething:(NSError**)error
```

传递给方法的参数是个指针，而该指针本身又指向另外一个指针，那个指针指向NSError对象。或者也可以把它当成一个直接指向NSError对象的指针。这样一来，此方法不仅能有普通的返回值，而且还能经由“输出参数”把NSError对象回传给调用者。其用法如下：

```
NSError *error = nil;
BOOL ret = [object doSomething:&error];
if (error) {
    // There was an error
}
```

像这样的方法一般都会返回Boolean值，用以表示该操作是成功了还是失败了。如果调用者不关注具体的错误信息，那么直接判断这个Boolean值就好；若是关注具体错误，那就检查经由“输出参数”所返回的那个错误对象。在不想知道具体错误的时候，可以给error参数传入nil。比方说，可以如下使用此方法：

```
BOOL ret = [object doSomething:nil];
if (ret) {
    // There was an error
}
```

实际上，在使用ARC时，编译器会把方法签名中的NSError**转换成NSError*__autoreleasing*，也就是说，指针所指的对象会在方法执行完毕后自动释放。这个对象必须自动释放，因为“doSomething:”方法不能保证其调用者可以把此方法中创建的NSError释放掉，所以必须加入autorelease。这就与大部分方法（以new、alloc、copy、mutableCopy开头的方法当然不在此列）的返回值所具备的语义相同了。

该方法通过下列代码把NSError对象传递到“输出参数”中：

```
- (BOOL)doSomething:(NSError**)error {
    // Do something that may cause an error

    if ( /* there was an error */ ) {
        if (error) {
            // Pass the 'error' through the out-parameter
            *error = [NSErrorerrorWithDomain:domain
                                        code:code
                                    userInfo:userInfo];
        }
        return NO; ///< Indicate failure
    } else {
        return YES; ///< Indicate success
    }
}
```

这段代码以*error语法为error参数“解引用”（dereference），也就是说，error所指的那

个指针现在要指向一个新的 NSError 对象了。在解引用之前，必须先保证 error 参数不是 nil，因为空指针解引用会导致“段错误”（segmentation fault）并使应用程序崩溃。调用者在不关心具体错误时，会给 error 参数传入 nil，所以必须判断这种情况。

NSError 对象里的“错误范围”（domain）、“错误码”（code）、“用户信息”（user information）等部分应该按照具体的错误情况填入适当内容。这样的话，调用者就可以根据错误类型分别处理各种错误了。错误范围应该定义成 NSString 型的全局常量，而错误码则定义成枚举类型为佳。例如，可以把这些值定义成下面这样：

```
// EOCErrors.h
extern NSString *const EOCErrorDomain;

typedef NS_ENUM(NSUInteger, EOCError) {
    EOCErrorUnknown             = -1,
    EOCErrorInternalInconsistency = 100,
    EOCErrorGeneralFault        = 105,
    EOCErrorBadInput            = 500,
};
// EOCErrors.m
NSString *const EOCErrorDomain = @"EOCErrorDomain";
```

最好能为你自己的程序库中所发生的错误指定一个专用的“错误范围”字符串，使用此字符串创建 NSError 对象，并将其返回给库的使用者，这样的话，他们就能确信：该错误肯定是由你的程序库所回报的。用枚举类型来表示错误码也是明智之举，因为这些枚举不仅解释了错误码的含义，而且还给它们起了个有意义的名字。此外，也可以在定义这些枚举的头文件里对每个错误类型详加说明。

要点

- 只有发生了可使整个应用程序崩溃的严重错误时，才应使用异常。
- 在错误不那么严重的情况下，可以指派“委托方法”（delegate method）来处理错误，也可以把错误信息放在 NSError 对象里，经由“输出参数”返回给调用者。

第 22 条：理解 NSCopying 协议

使用对象时经常需要拷贝它。在 Objective-C 中，此操作通过 copy 方法完成。如果想令自己的类支持拷贝操作，那就要实现 NSCopying 协议，该协议只有一个方法：

```
- (id)copyWithZone:(NSZone*)zone
```

为何会出现 NSZone 呢？因为以前开发程序时，会据此把内存分成不同的“区”（zone），而对象会创建在某个区里面。现在不用了，每个程序只有一个区：“默认区”（default zone）。所以说，尽管必须实现这个方法，但是你不必担心其中的 zone 参数。

copy方法由NSObject实现，该方法只是以“默认区”为参数来调用“copyWithZone:”。我们总是想覆写copy方法，其实真正需要实现的却是“copyWithZone:”方法，这个问题大家一定要注意。

若想使某个类支持拷贝功能，只需声明该类遵从NSCopying协议，并实现其中的那个方法即可。比方说，有个表示个人信息的类，可以在其接口定义中声明此类遵从NSCopying协议：

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

@interface EOCPerson : NSObject <NSCopying>


@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;

- (id)initWithFirstName:(NSString*)firstName
            andLastName:(NSString*)lastName;

@end
```

然后，实现协议中规定的方法：

```
- (id)copyWithZone:(NSZone*)zone {
    EOCPerson *copy = [[[self class] allocWithZone:zone]
                       initWithFirstName:_firstName
                           andLastName:_lastName];
    return copy;
}
```

在本例所实现的“copyWithZone:”中，我们直接把待拷贝的对象交给“全能初始化方法”（designated initializer），令其执行所有初始化工作。然而有的时候，除了要拷贝对象，还要完成其他一些操作，比如类对象中的数据结构可能并未在初始化方法中设置好，需要另行设置。举个例子，假如EOCPerson中含有一个数组，与其他EOCPerson对象建立或解除朋友关系的那些方法都需要操作这个数组。那么在这种情况下，你得把这个包含朋友对象的数组也一并拷贝过来。下面列出了实现此功能所需的全部代码：

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

@interface EOCPerson : NSObject<NSCopying>

@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;

- (id)initWithFirstName:(NSString*)firstName
            andLastName:(NSString*)lastName;
- (void)addFriend:(EOCPerson*)person;
- (void)removeFriend:(EOCPerson*)person;

@end
```

```
@implementation EOCPerson {
    NSMutableSet *_friends;
}

- (id)initWithFirstName:(NSString*)firstName
            andLastName:(NSString*)lastName {
    if ((self = [super init])) {
        _firstName = [firstName copy];
        _lastName = [lastName copy];
        _friends = [NSMutableSetnew];
    }
    return self;
}

- (void)addFriend:(EOCPerson*)person {
    [_friends addObject:person];
}

- (void)removeFriend:(EOCPerson*)person {
    [_friends removeObject:person];
}

- (id)copyWithZone:(NSZone*)zone {
    EOCPerson *copy = [[[self class] allocWithZone:zone]
                       initWithFirstName:_firstName
                             andLastName:_lastName];
    copy->_friends = [_friends mutableCopy];
    return copy;
}
@end
```

这次所实现的方法比原来多了一些代码，它把本对象的 _friends 实例变量复制了一份，令 copy 对象的 _friends 实例变量指向这个复制过的 set。注意，这里使用了 -> 语法，因为 _friends 并非属性，只是个在内部使用的实例变量。其实也可以声明一个属性来表示它，不过由于该变量不会在本类之外使用，所以那么做没必要。

这个例子提出了一个有趣的问题：为什么要拷贝 _friends 实例变量呢？不拷贝这个变量，直接令两个对象共享同一个可变的 set 是否更简单些？如果真的那样做了，那么在给原来的对象添加一个新朋友后，拷贝过的那个对象居然也“神奇地”（magically）与之为友了。在本例中，这显然不是我们想要的效果。然而，那个 set 若是不可变的，则无须复制，因为其中的内容毕竟不会改变，所以不用担心此类问题。如果复制了，那么内存中将会有两个一模一样的 set，反而造成浪费。

通常情况下，应该像本例这样，采用全能初始化方法来初始化待拷贝的对象。不过有些时候不能这么做，因为全能初始化方法会产生一些“副作用”（side effect），这些附加操作对目前要拷贝的对象无益。比如，初始化方法可能要设置一个复杂的内部数据结构，可是在拷贝后的对象中，这个数据结构立刻就要用其他数据来覆写，所以没必要再设置一遍。

仔细看看刚才的“copyWithZone:”方法，你就会发现，存放朋友对象的那个set是通过mutableCopy方法来复制的。此方法来自另一个叫做NSMutableCopying的协议。该协议与NSCopying类似，也只定义了一个方法，然而方法名不同：

```
- (id)mutableCopyWithZone:(NSZone*)zone
```

mutableCopy这个“辅助方法”（helper）与copy相似，也是用默认的zone参数来调“mutableCopyWithZone:”。如果你的类分为可变版本（mutable variant）与不可变版本（immutable variant），那么就应该实现NSMutableCopying。若采用此模式，则在可变类中覆写“copyWithZone:”方法时，不要返回可变的拷贝，而应返回一份不可变的版本。无论当前实例是否可变，若需获取其可变版本的拷贝，均应调用mutableCopy方法。同理，若需要不可变的拷贝，则总应通过copy方法来获取。

对于不可变的NSArray与可变的NSMutableArray来说，下列关系总是成立的：

```
-[NSMutableArray copy] =>NSArray
-[NSArray mutableCopy] =>NSMutableArray
```

有个微妙的情况要注意：在可变对象上调用copy方法会返回另外一个不可变类的实例。这样做是为了能在可变版本与不可变版本之间自由切换。要实现此目标，还有个办法，就是提供三个方法：copy、immutableCopy、mutableCopy，其中，copy所返回的拷贝对象与当前对象的类型一致，而另外两个方法则分别返回不可变版本与可变版本的拷贝。但是，如果调用者并不知道其所用的实例是否真的可变，那么这种做法就不太好了。某个方法可能会把NSMutableArray对象当作NSArray返回给你，而你在上面调用copy方法来复制它。此时你以为拷贝后的对象应该是不可变的数组，但实际上它却是可变的。

可以查询类型信息（参见第14条）以判断待拷贝的实例是否可变，不过那样做比较麻烦，因为每次复制对象的时候都得查询。为了安全起见，最后还是会使用immutableCopy和mutableCopy这两个方法来复制对象，而这样做就和只有copy与mutableCopy方法的设计方案毫无二致了。把拷贝方法称为copy而非immutableCopy的原因在于，NSCopying不仅设计给那些具有可变版本和不可变版本的类来用，而且还要供其他一些类使用，而那些类没有“可变”与“不可变”之分，所以说，把拷贝方法叫成immutableCopy不合适。

在编写拷贝方法时，还要决定一个问题，就是应该执行“深拷贝”（deep copy）还是“浅拷贝”（shallow copy）。深拷贝的意思就是：在拷贝对象自身时，将其底层数据也一并复制过去。Foundation框架中的所有collection类在默认情况下都执行浅拷贝，也就是说，只拷贝容器对象本身，而不复制其中数据。这样做的主要原因在于，容器内的对象未必都能拷贝，而且调用者也未必想在拷贝容器时一并拷贝其中的每个对象。图3-2描述了深拷贝与浅拷贝的区别。

一般情况下，我们会遵照系统框架所使用的那种模式，在自定义的类中以浅拷贝的方式实“copyWithZone:”方法。但如果有必要的话，也可以增加一个执行深拷贝的方法。以NSSet为例，该类提供了下面这个初始化方法，用以执行深拷贝：

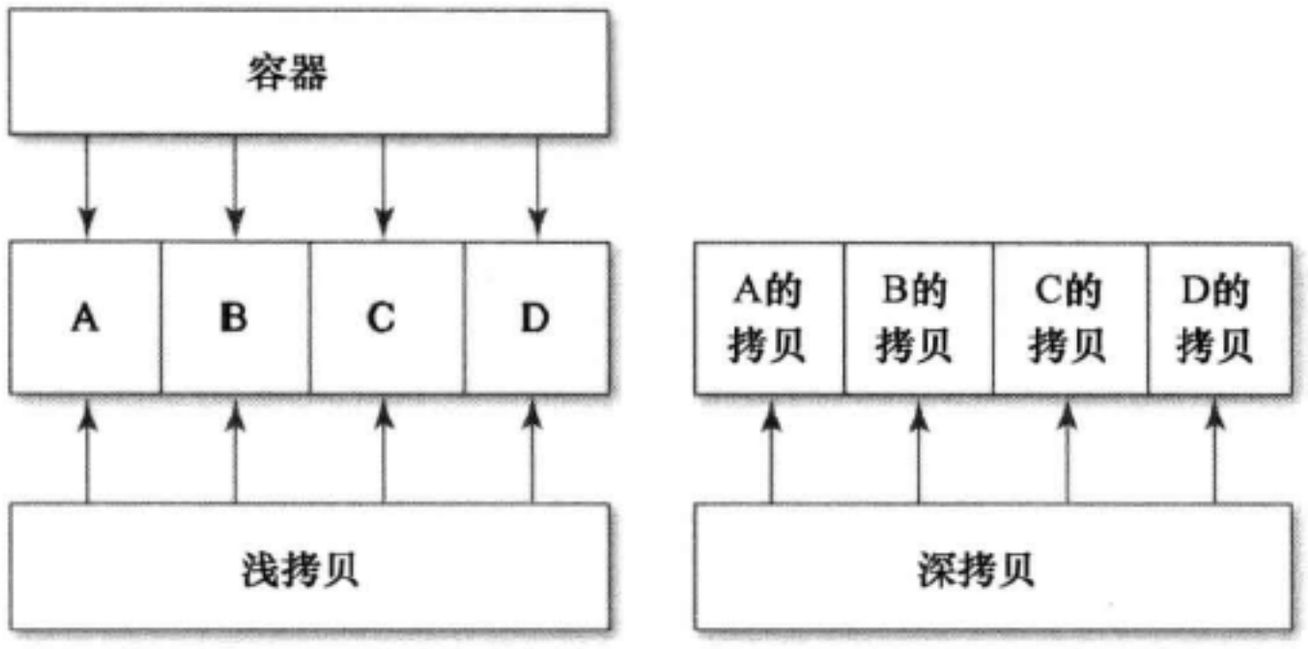

图 3-2　浅拷贝与深拷贝对比图。浅拷贝之后的内容与原始内容均指向相同对象。而深拷贝之后的内容所指的对象是原始内容中相关对象的一份拷贝

```
- (id)initWithSet:(NSArray*)array copyItems:(BOOL)copyItems
```

若 copyItem 参数设为 YES，则该方法会向数组中的每个元素发送 copy 消息，用拷贝好的元素创建新的 set，并将其返回给调用者。

在 EOCPerson 那个例子中，存放朋友对象的 set 是用“ copyWithZone:”方法来拷贝的，根据刚才讲的内容可知，这种浅拷贝方式不会逐个复制 set 中的元素。若需要深拷贝的话，则可像下面这样，编写一个专供深拷贝所用的方法：

```
- (id)deepCopy {
    EOCPerson *copy = [[[self class] alloc]
                       initWithFirstName:_firstName
                              andLastName:_lastName];
    copy->_friends = [[NSMutableSet alloc] initWithSet:_friends
                                             copyItems:YES];
    return copy;
}
```

因为没有专门定义深拷贝的协议，所以其具体执行方式由每个类来确定，你只需决定自己所写的类是否要提供深拷贝方法即可。另外，不要假定遵从了 NSCopying 协议的对象都会执行深拷贝。在绝大多数情况下，执行的都是浅拷贝。如果需要在某对象上执行深拷贝，那么除非该类的文档说它是用深拷贝来实现 NSCopying 协议的，否则，要么寻找能够执行深拷贝的相关方法，要么自己编写方法来做。

要点

- 若想令自己所写的对象具有拷贝功能，则需实现 NSCopying 协议。
- 如果自定义的对象分为可变版本与不可变版本，那么就要同时实现 NSCopying 与 NSMutableCopying 协议。
- 复制对象时需决定采用浅拷贝还是深拷贝，一般情况下应该尽量执行浅拷贝。
- 如果你所写的对象需要深拷贝，那么可考虑新增一个专门执行深拷贝的方法。

第 4 章

协议与分类

Objective-C 语言有一项特性叫做“协议”（protocol），它与 Java 的“接口”（interface）类似。Objective-C 不支持多重继承，因而我们把某个类应该实现的一系列方法定义在协议里面。协议最为常见的用途是实现委托模式（参见第 23 条），不过也有其他用法。理解并善用协议可令代码变得更易维护，因为协议这种方式能很好地描述接口。

“分类”（Category）也是 Objective-C 的一项重要语言特性。利用分类机制，我们无须继承子类即可直接为当前类添加方法，而在其他编程语言中，则需通过继承子类来实现。由于 Objective-C 运行期系统是高度动态的，所以才能支持这一特性，然而，其中也隐藏着一些陷阱，因此在使用分类之前，应该先理解它。

第 23 条：通过委托与数据源协议进行对象间通信

对象之间经常需要相互通信，而通信方式有很多种。Objective-C 开发者广泛使用一种名叫“委托模式”（Delegate pattern）的编程设计模式来实现对象间的通信，该模式的主旨是：定义一套接口，某对象若想接受另一个对象的委托，则需遵从此接口，以便成为其“委托对象”（delegate）。而这“另一个对象”则可以给其委托对象回传一些信息，也可以在发生相关事件时通知委托对象。

此模式可将数据与业务逻辑解耦。比方说，用户界面里有个显示一系列数据所用的视图，那么，此视图只应包含显示数据所需的逻辑代码，而不应决定要显示何种数据以及数据之间如何交互等问题。视图对象的属性中，可以包含负责数据与事件处理的对象。这两种对象分别称为“数据源”（data source）与“委托”（delegate）。

在 Objective-C 中，一般通过“协议”这项语言特性来实现此模式，整个 Cocoa 系统框架都是这么做的。如果你的代码也这样写，那么就能和系统框架很好地融合在一起了。

为演示此模式，我们举个例子，假设要编写一个从网上获取数据的类。此类也许要从远程服务器的某个资源里获取数据。那个远程服务器可能过很长时间才会应答，而在获取数据的过程中阻塞应用程序则是一种非常糟糕的做法。于是，在这种情况下，我们通常会

使用委托模式：获取网络数据的类含有一个“委托对象”，在获取完数据之后，它会回调这个委托对象。图 4-1 演示了此概念：EOCDataModel 对象就是 EOCNetworkFetcher 的委托对象。EOCDataModel 请求 EOCNetworkFetcher“以异步方式执行一项任务”（perform a task asynchronously），而 EOCNetworkFetcher 在执行完这项任务之后，就会通知其委托对象，也就是 EOCDataModel。

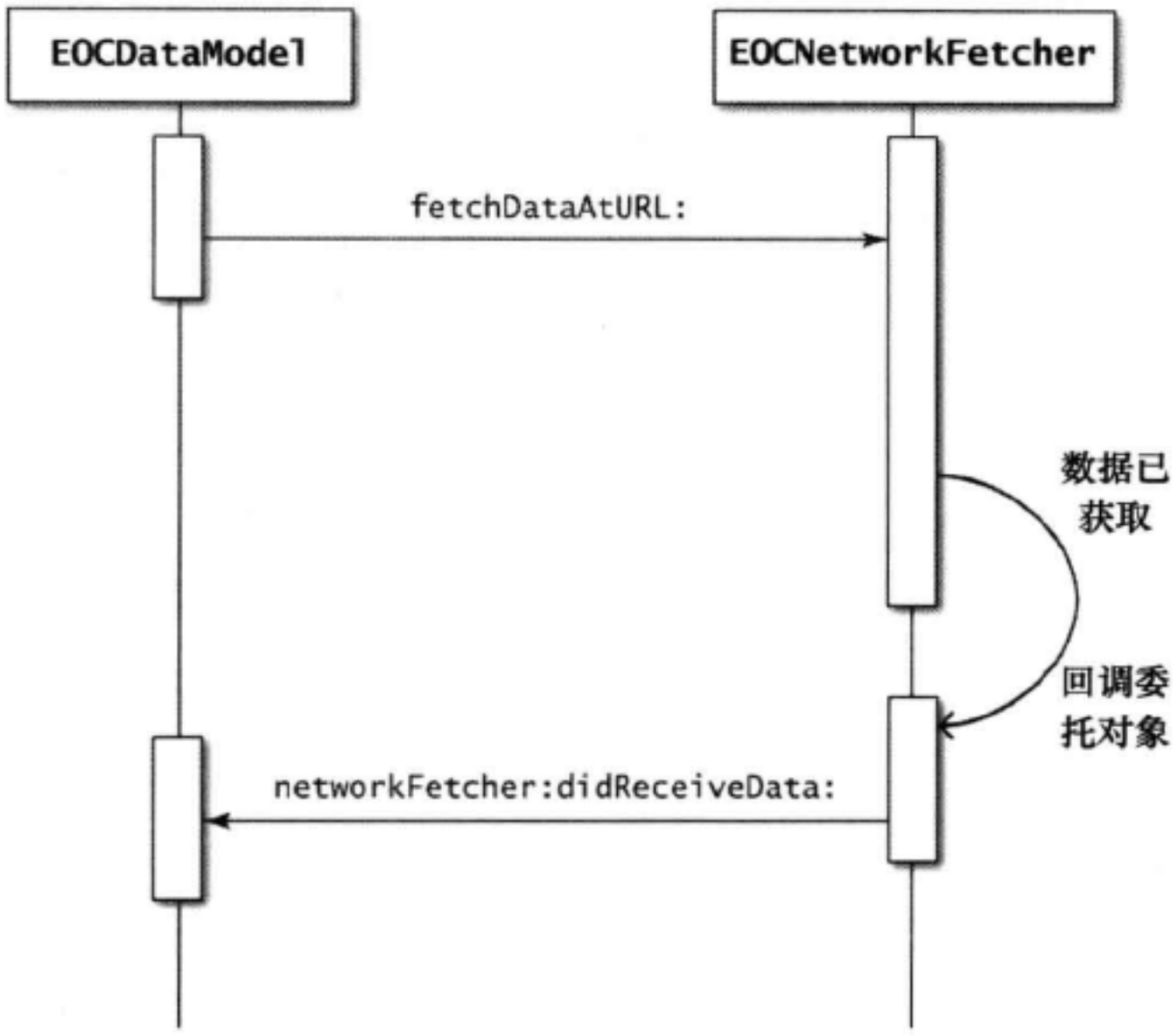

图 4-1　回调委托对象的流程。请注意，“委托对象”未必非得由 EOCDataModel 实例来担任不可，也可以由另外一个对象扮演此角色

利用协议机制，很容易就能以 Objective-C 代码实现此模式。在图 4-1 所演示的这种情况下，协议可以这样来定义：

```
@protocol EOCNetworkFetcherDelegate
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didReceiveData:(NSData*)data;
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
      didFailWithError:(NSError*)error;
@end
```

委托协议名通常是在相关类名后面加上 Delegate 一词，整个类名采用“驼峰法”来写。以这种方式来命名委托协议的话，使用此代码的人很快就能理解其含义了。

有了这个协议之后，类就可以用一个属性来存放其委托对象了。在本例中，这个类就是 EOCNetworkFetcher 类。于是，此类的接口可以写成这样：

```
@interface EOCNetworkFetcher : NSObject
@property (nonatomic, weak)
```

```
    id <EOCNetworkFetcherDelegate> delegate;
@end
```

一定要注意：这个属性需定义成 weak，而非 strong，因为两者之间必须为“非拥有关系”（nonowning relationship）。通常情况下，扮演 delegate 的那个对象也要持有本对象。例如在本例中，想使用 EOCNetworkFetcher 的那个对象就会持有本对象，直到用完本对象之后，才会释放。假如声明属性的时候用 strong 将本对象与委托对象之间定为“拥有关系”，那么就会引入“保留环”(retain cycle)。因此，本类中存放委托对象的这个属性要么定义成 weak，要么定义成 unsafe_unretained，如果需要在相关对象销毁时自动清空（autoniling，参见第 6 条），则定义为前者，若不需要自动清空，则定义为后者。图 4-2 演示了本对象与委托对象之间的所有权关系。

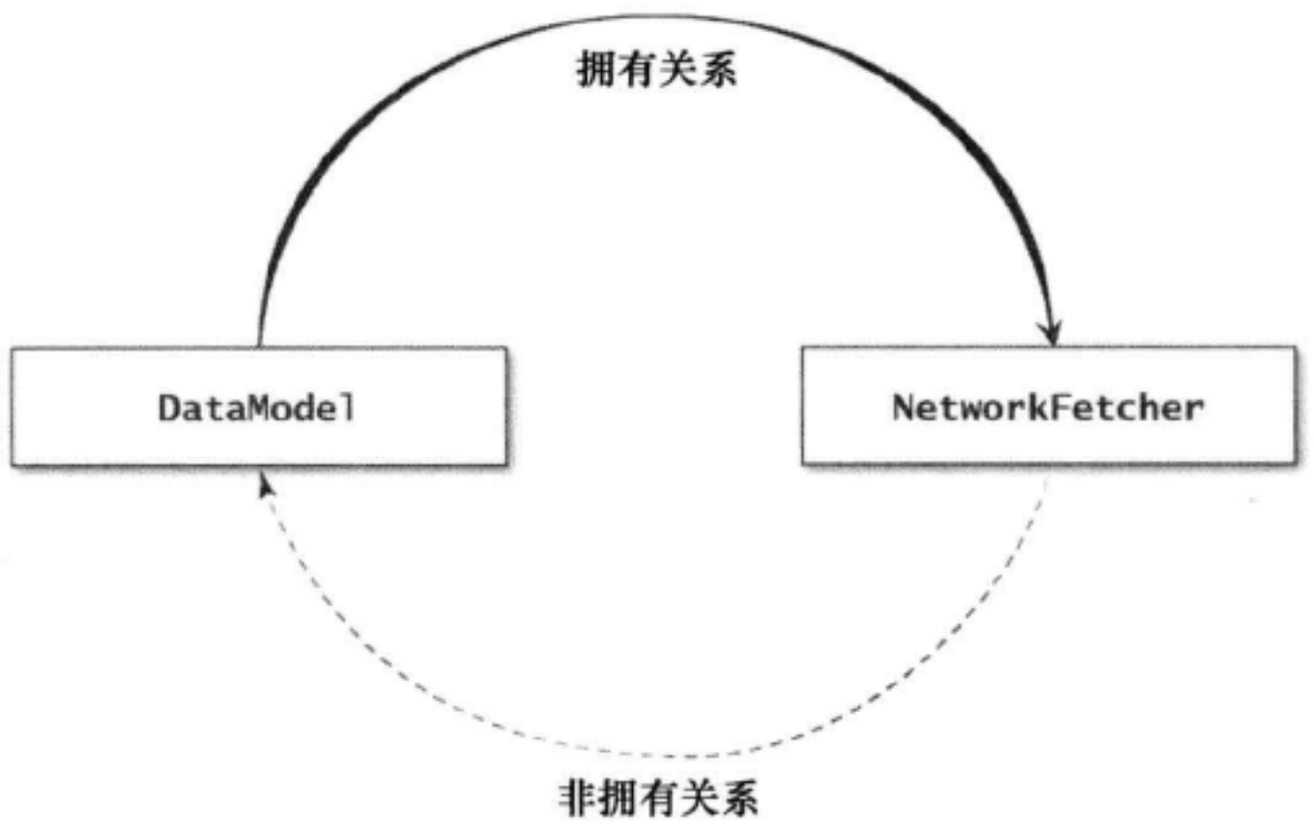

图 4-2 所有权关系图：为了避免“保留环”，NetworkFetcher 不保留其 delegate 属性

实现委托对象的办法是声明某个类遵从委托协议，然后把协议中想实现的那些方法在类里实现出来。某类若要遵从委托协议，可以在其接口中声明，也可以在“ class-continuation 分类”（参见第 27 条）中声明。如果要向外界公布此类实现了某协议，那么就在接口中声明，而如果这个协议是个委托协议的话，那么通常只会在类的内部使用。所以说，这种情况一般都是在“class-continuation 分类”里声明的：

```
@implementation EOCDataModel () <EOCNetworkFetcherDelegate>
@end
@implementation EOCDataModel
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didReceiveData:(NSData*)data {
    /* Handle data */
}
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didFailWithError:(NSError*)error {
    /* Handle error */
}
@end
```

委托协议中的方法一般都是“可选的”（optional），因为扮演“受委托者”角色的这个对象未必关心其中的所有方法。在本例中，DataModel 类可能并不关心获取数据的过程中是否有错误发生，所以此类也许不会实现“ networkFetcher:didFailWithError:”方法。为了指明可选方法，委托协议经常使用 @optional 关键字来标注其大部分或全部的方法：

```
@protocol EOCNetworkFetcherDelegate
@optional
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didReceiveData:(NSData*)data;
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didFailWithError:(NSError*)error;
@end
```

如果要在委托对象上调用可选方法，那么必须提前使用类型信息查询方法（参见第 14 条）判断这个委托对象能否响应相关选择子。以 EOCNetworkFetcher 为例，应该这样写：

```
NSData *data = /* data obtained from network */;
if ([_delegate respondsToSelector:
          @selector(networkFetcher:didReceiveData:)])
{
    [_delegate networkFetcher:self didReceiveData:data];
}
```

这段代码用“ respondsToSelector:”来判断委托对象是否实现了相关方法。如果实现了，就调用，如果没实现，就不执行任何操作。这样的话，delegate 对象就可以完全按照其需要来实现委托协议中的方法了，不用担心因为哪个方法没实现而导致程序出问题。即便没有设置委托对象，程序也能照常运行，因为给 nil 发送消息将使 if 语句的值成为 false。

delegate 对象中的方法名也一定要起得很恰当才行。方法名应该准确描述当前发生的事件以及 delegate 对象为何要获知此事件。在本例中，delegate 对象里的方法名读起来非常清晰，表明某个“网络数据获取器”（network fetcher）对象刚刚接收到某份数据。正如上一段代码所示，在调用 delegate 对象中的方法时，总是应该把发起委托的实例也一并传入方法中，这样，delegate 对象在实现相关方法时，就能根据传入的实例分别执行不同的代码了。比方说可以这样写：

```
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didReceiveData:(NSData*)data
{
    if (fetcher == _myFetcherA) {
        /* Handle data */
    } else if (fetcher == _myFetcherB) {
        /* Handle data */
    }
}
```

上面这段代码表明，委托对象有两个不同的“网络数据获取器”，所以它必须根据传入

的参数来判断到底是哪个 EOCNetworkFetcher 获取到了数据。若没有此信息，则委托对象在同一时间只能使用一个网络数据获取器，这么做不太好。

delegate 里的方法也可以用于从获取委托对象中获取信息。比方说，EOCNetworkFetcher 类也许想提供一种机制：在获取数据时如果遇到了“重定向”（redirect），那么将询问其委托对象是否应该发生重定向。delegate 对象中的相关方法可以写成这样：

```
- (BOOL)networkFetcher:(EOCNetworkFetcher*)fetcher
    shouldFollowRedirectToURL:(NSURL*)url;
```

通过这个例子，大家应该很容易理解此模式为何叫做“委托模式”：因为对象把应对某个行为的责任委托给另外一个类了。

也可以用协议定义一套接口，令某类经由该接口获取其所需的数据。委托模式的这一用法旨在向类提供数据，故而又称“数据源模式”（Data Source Pattern）。在此模式中，信息从数据源（Data Source）流向类（Class）；而在常规的委托模式中，信息则从类流向受委托者（Delegate）。图 4-3 演示了这两条信息流。

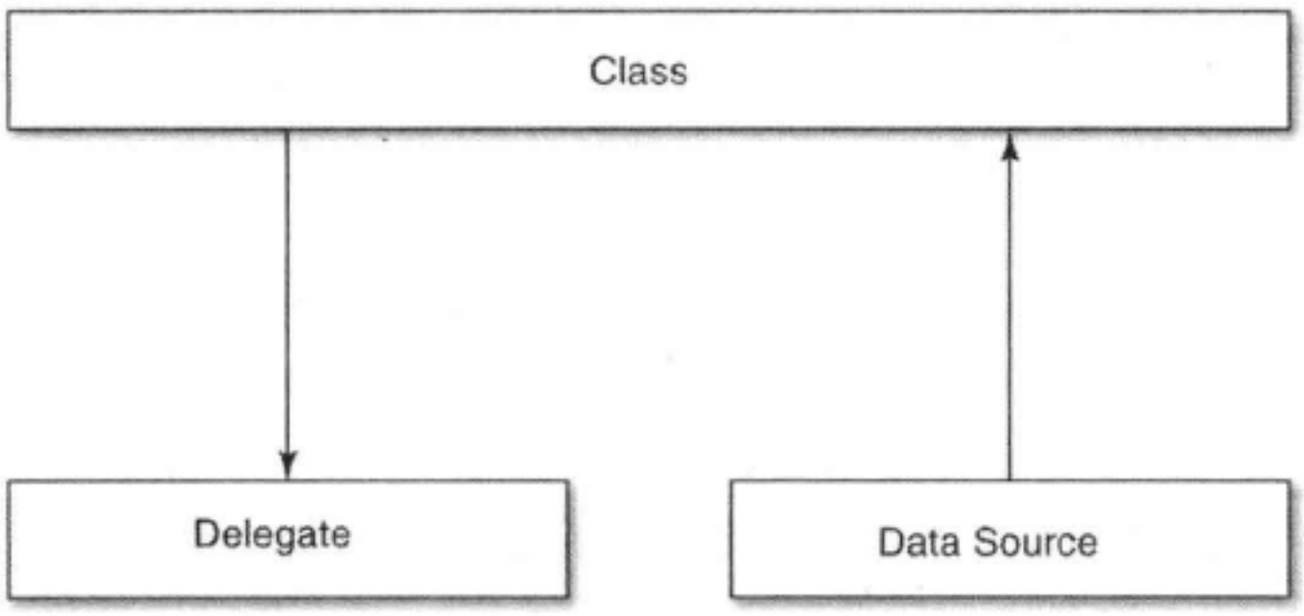

图 4-3　在信息源模式中，信息从数据源流向类，而在普通的委托模式中，信息则从类流向受委托者

比方说，用户界面框架中的“列表视图”（list view）对象可能会通过数据源协议来获取要在列表中显示的数据。除了数据源之外，列表视图还有一个受委托者，用于处理用户与列表的交互操作。将数据源协议与委托协议分离，能使接口更加清晰，因为这两部分的逻辑代码也分开了。另外，“数据源”与“受委托者”可以是两个不同的对象。然而一般情况下，都用同一个对象来扮演这两种角色。

在实现委托模式与数据源模式时，如果协议中的方法是可选的，那么就会写出一大批类似下面这样的代码来：

```
if ([_delegate respondsToSelector:
            @selector(someClassDidSomething:)])
{
    [_delegate someClassDidSomething];
}
```

很容易用代码查出某个委托对象是否能响应特定的选择子，可是如果频繁执行此操作的

话，那么除了第一次检测的结果有用之外，后续的检测可能都是多余的。如果委托对象本身没变，那么不太可能会突然响应某个原来不能响应的选择子，也不太会突然无法响应某个原来可以响应的选择子。鉴于此，我们通常把委托对象能否响应某个协议方法这一信息缓存起来，以优化程序效率。假设在“网络数据获取器”那个例子中，delegate 对象所遵从的协议里有个表示数据获取进度的回调方法，每当数据获取有进度时，委托对象就会得到通知。这个方法在网络数据获取器的生命期（life cycle）里会多次调用，如果每次都检查委托对象是否能响应此选择子，那就显得多余了。

将刚才说的那个选择子加入之后，delegate 对象所要实现的委托协议就扩充成：

```
@protocol EOCNetworkFetcherDelegate
@optional
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didReceiveData:(NSData*)data;
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didFailWithError:(NSError*)error;
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didUpdateProgressTo:(float)progress;
@end
```

扩充后的协议只增加了一个方法，就是可选的“networkFetcher:didUpdateProgressTo:”方法。将方法响应能力缓存起来的最佳途径是使用“位段”(bitfield)[⊖]数据类型。这是一项乏人问津的 C 语言特性，但在此处用起来却正合适。我们可以把结构体中某个字段所占用的二进制位个数设为特定的值。比如像这样：

```
struct data {
    unsigned int fieldA : 8;
    unsigned int fieldB : 4;
    unsigned int fieldC : 2;
    unsigned int fieldD : 1;
};
```

在结构体中，fieldA 位段将占用 8 个二进制位，fieldB 占用 4 个，fieldC 占用两个，fieldD 占用 1 个。于是，fieldA 可以表示 0 至 255 之间的值，而 fieldD 则可以表示 0 或 1 这两个值。我们可以像 fieldD 这样，把委托对象是否实现了协议中的相关方法这一信息缓存起来。如果创建的结构体中只有大小为 1 的位段，那么就能把许多 Boolean 值塞入一小块数据里面了。以网络数据获取器为例，可以在该实例中嵌入一个含有位段的结构体作为其实例变量，而结构体中的每个位段则表示 delegate 对象是否实现了协议中的相关方法。此结构体的用法如下：

```
@interface EOCNetworkFetcher () {
    struct {
        unsigned int didReceiveData      : 1;
        unsigned int didFailWithError    : 1;
        unsigned int didUpdateProgressTo : 1;
```

⊖ 又称“位域”、“位字段”。——译者注

```
    } _delegateFlags;
}
@end
```

笔者使用第27条所讲的“class-continuation分类”来新增实例变量，而新增的这个实例变量是个结构体，其中含有三个位段，每个位段都与delegate所遵从的协议中某个可选方法相对应。在EOCNetworkFetcher类里，可以像下面这样查询并设置结构体中的位段：

```
// Set flag
_delegateFlags.didReceiveData = 1;

// Check flag
if (_delegateFlags.didReceiveData) {
    // Yes, flag set
}
```

这个结构体用来缓存委托对象是否能响应特定的选择子。实现缓存功能所用的代码可以写在delegate属性所对应的设置方法里：

```
- (void)setDelegate:(id<EOCNetworkFetcher>)delegate {
    _delegate = delegate;
    _delegateFlags.didReceiveData =
      [delegate respondsToSelector:
                @selector(networkFetcher:didReceiveData:)];
    _delegateFlags.didFailWithError =
      [delegate respondsToSelector:
                @selector(networkFetcher:didFailWithError:)];
    _delegateFlags.didUpdateProgressTo =
      [delegate respondsToSelector:
                @selector(networkFetcher:didUpdateProgressTo:)];
}
```

这样的话，每次调用delegate的相关方法之前，就不用检测委托对象是否能响应给定的选择子了，而是直接查询结构体里的标志：

```
if (_delegateFlags.didUpdateProgressTo) {
    [_delegate networkFetcher:self
         didUpdateProgressTo:currentProgress];
}
```

在相关方法要调用很多次时，值得进行这种优化。而是否需要优化，则应依照具体代码来定。这就需要分析代码性能，并找出瓶颈，若发现执行速度需要改进，则可使用此技巧。如果要频繁通过数据源协议从数据源中获取多份相互独立的数据，那么这项优化技术极有可能会提高程序效率。

要点

- 委托模式为对象提供了一套接口，使其可由此将相关事件告知其他对象。

- 将委托对象应该支持的接口定义成协议，在协议中把可能需要处理的事件定义成方法。
- 当某对象需要从另外一个对象中获取数据时，可以使用委托模式。这种情境下，该模式亦称“数据源协议”(data source protocal)。
- 若有必要，可实现含有位段的结构体，将委托对象是否能响应相关协议方法这一信息缓存至其中。

第 24 条：将类的实现代码分散到便于管理的数个分类之中

类中经常容易填满各种方法，而这些方法的代码则全部堆在一个巨大的实现文件里。有时这么做是合理的，因为即便通过重构把这个类打散，效果也不会更好。在此情况下，可以通过 Objective-C 的“分类”机制，把类代码按逻辑划入几个分区中，这对开发与调试都有好处。

比如说，我们把个人信息建模为类。那么这个类就可能包含下面几个方法：

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

@interface EOCPerson : NSObject
@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;
@property (nonatomic, strong, readonly) NSArray *friends;

- (id)initWithFirstName:(NSString*)firstName
            andLastName:(NSString*)lastName;

/* Friendship methods */
- (void)addFriend:(EOCPerson*)person;
- (void)removeFriend:(EOCPerson*)person;
- (BOOL)isFriendsWith:(EOCPerson*)person;

/* Work methods */
- (void)performDaysWork;
- (void)takeVacationFromWork;

/* Play methods */
- (void)goToTheCinema;
- (void)goToSportsGame;

@end
```

在实现该类时，所有方法的代码可能会写在一个大文件里。如果还向类中继续添加方法的话，那么源代码文件就会越来越大，变得难于管理。所以说，应该把这样的类分成几个不同的部分。例如，可以用“分类”机制把刚才的类改写成下面这样：

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

@interface EOCPerson : NSObject
@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;
@property (nonatomic, strong, readonly) NSArray *friends;

- (id)initWithFirstName:(NSString*)firstName
            andLastName:(NSString*)lastName;
@end

@interface EOCPerson (Friendship)
- (void)addFriend:(EOCPerson*)person;
- (void)removeFriend:(EOCPerson*)person;
- (BOOL)isFriendsWith:(EOCPerson*)person;
@end

@interface EOCPerson (Work)
- (void)performDaysWork;
- (void)takeVacationFromWork;
@end

@interface EOCPerson (Play)
- (void)goToTheCinema;
- (void)goToSportsGame;
@end
```

现在，类的实现代码按照方法分成了好几个部分。所以说，这项语言特性当然就叫做“分类”啦。在本例中，类的基本要素（诸如属性与初始化方法等）都声明在“主实现”（main implementation）里。执行不同类型的操作所用的另外几套方法则归入各个分类中。

使用分类机制之后，依然可以把整个类都定义在一个接口文件中，并将其代码写在一个实现文件里。可是，随着分类数量增加，当前这份实现文件很快就膨胀得无法管理了。此时可以把每个分类提取到各自的文件中去。以 EOCPerson 为例，可以按照其分类拆分成下列几个文件：

- EOCPerson+Friendship(.h/.m)
- EOCPerson+Work(.h/.m)
- EOCPerson+Play(.h/.m)

比方说，与交友功能相关的那个分类可以这样写：

```
// EOCPerson+Friendship.h
#import "EOCPerson.h"

@interface EOCPerson (Friendship)
- (void)addFriend:(EOCPerson*)person;
- (void)removeFriend:(EOCPerson*)person;
- (BOOL)isFriendsWith:(EOCPerson*)person;
```

```
@end

// EOCPerson+Friendship.m
#import "EOCPerson+Friendship.h"

@implementation EOCPerson (Friendship)
- (void)addFriend:(EOCPerson*)person {
    /* ... */
}
- (void)removeFriend:(EOCPerson*)person {
    /* ... */
}
- (BOOL)isFriendsWith:(EOCPerson*)person {
    /* ... */
}
@end
```

通过分类机制，可以把类代码分成很多个易于管理的小块，以便单独检视。使用分类机制之后，如果想用分类中的方法，那么要记得在引入 EOCPerson.h 时一并引入分类的头文件。虽然稍微有点麻烦，不过分类仍然是一种管理代码的好办法。

即使类本身不是太大，我们也可以使用分类机制将其切割成几块，把相应代码归入不同的“功能区”（functional area）中。Cocoa 中的 NSURLRequest 类及其可变版本 NSMutableURLRequest 类就是这么做的。这个类用于执行从 URL 中获取数据的请求，而且通常使用 HTTP 协议从因特网中的某个服务器上获取，不过，由于该类设计得较为通用，所以也可以使用其他协议。与标准的 URL 请求相比，执行 HTTP 请求时还需要另外一些信息，例如“HTTP 方法”[⊖]（HTTP method，GET、POST 等）或 HTTP 头（HTTP header）。

然而却不便从 NSURLRequest 中继承子类以实现 HTTP 协议的特殊需求，因为本类包裹了一套操作 CFURLRequest 数据结构所需的 C 函数，所有“HTTP 方法”都包含在这个结构里。于是，为了扩展 NSURLRequest 类，把与 HTTP 有关的方法归入名为 NSHTTPURLRequest 的分类中，而把与可变版本有关的方法归入名为 NSMutableHTTPURLRequest 的分类中。这样，所有底层 CFURLRequest 函数就都封装在同一个 Objective-C 类里了，而在这个类里，与 HTTP 有关的方法却又要单独放在一处，因为若是不这么做的话，该类的使用者就会有疑问：为什么能在使用 FTP 协议的 request 对象上设置“HTTP 方法”呢？

之所以要将类代码打散到分类中还有个原因，就是便于调试：对于某个分类中的所有方法来说，分类名称都会出现在其符号中。例如，“addFriend:”方法的“符号名”（symbol name）如下：

```
-[EOCPerson(Friendship) addFriend:]
```

在调试器的回溯信息中，会看到类似下面这样的内容：

⊖ 这里的“方法”是“动作”的意思，其含义与编程语言中用以称呼函数的“方法”一词不同。——译者注

```
frame #2: 0x00001c50 Test'-[EOCPerson(Friendship) addFriend:]
+ 32 at main.m:46
```

根据回溯信息中的分类名称，很容易就能精确定位到类中的方法所属的功能区，这对于某些应该视为私有的方法来说更是极为有用。可以创建名为 Private 的分类，把这种方法全都放在里面。这个分类里的方法一般只会在类或框架内部使用，而无须对外公布。这样一来，类的使用者有时可能会在查看回溯信息时发现 private 一词，从而知道不应该直接调用此方法了。这可算作一种编写“自我描述式代码”（self-documenting code）的办法。

在编写准备分享给其他开发者使用的程序库时，可以考虑创建 Private 分类。经常会遇到这样一些方法：它们不是公共 API 的一部分，然而却非常适合在程序库之内使用。此时应该创建 Private 分类，如果程序库中的某个地方要用到这些方法，那就引入此分类的头文件。而分类的头文件并不随程序库一并公开，于是该库的使用者也就不知道库里还有这些私有方法了。

要点

- 使用分类机制把类的实现代码划分成易于管理的小块。
- 将应该视为“私有”的方法归入名叫 Private 的分类中，以隐藏实现细节。

第 25 条：总是为第三方类的分类名称加前缀

分类机制通常用于向无源码的既有类中新增功能。这个特性极为强大，但在使用时也很容易忽视其中可能产生的问题。这个问题在于：分类中的方法是直接添加在类里面的，它们就好比这个类中的固有方法。将分类方法加入类中这一操作是在运行期系统加载分类时完成的。运行期系统会把分类中所实现的每个方法都加入类的方法列表中。如果类中本来就有此方法，而分类又实现了一次，那么分类中的方法会覆盖原来那一份实现代码。实际上可能会发生很多次覆盖，比如某个分类中的方法覆盖了“主实现”中的相关方法，而另外一个分类中的方法又覆盖了这个分类中的方法。多次覆盖的结果以最后一个分类为准。

比方说，要给 NSString 添加分类，并在其中提供一些辅助方法，用于处理与 HTTP URL 有关的字符串。你可能会把分类写成这样：

```
@interface NSString (HTTP)

// Encode a string with URL encoding
- (NSString*)urlEncodedString;

// Decode a URL encoded string
- (NSString*)urlDecodedString;

@end
```

现在看起来没什么问题，可是，如果还有一个分类也往 NSString 里添加方法，那会如何呢？那个分类里可能也有个名叫 urlEncodedString 的方法，其代码与你所添加的大同小异，但却不能正确实现你所需的功能。那个分类的加载时机如果晚于你所写的这个分类，那么其代码就会把你的那一份覆盖掉，这样的话，你在代码中调用 urlEncodedString 方法时，实际执行的是那个分类里的实现代码。由于其执行结果和你预期的值不同，所以自己所写的那些代码也许就无法正常运行了。这种 bug 很难追查，因为你可能意识不到实际执行的 urlEncodedString 代码并不是自己实现的那一份。

要解决此问题，一般的做法是：以命名空间来区别各个分类的名称与其中所定义的方法。想在 Objective-C 中实现命名空间功能，只有一个办法，就是给相关名称都加上某个共用的前缀。与给类名加前缀（参见第 15 条）时所应考虑的因素相似，给分类所加的前缀也要选得恰当才行。一般来说，这个前缀应该与应用程序或程序库中其他地方所用的前缀相同。于是，我们可以给刚才那个 NSString 分类加上 ABC 前缀：

```
@interface NSString (ABC_HTTP)

// Encode a string with URL encoding
- (NSString*)abc_urlEncodedString;

// Decode a URL encoded string
- (NSString*)abc_urlDecodedString;

@end
```

从技术角度讲，并不是非得用命名空间把各个分类的名称区隔开不可。即便两个分类重名了，也不会出错。然而这样做不好，编译器会发出类似下面这种警告信息：

```
warning: duplicate definition of category 'HTTP' on interface
'NSString'
```

即便加了前缀，也难保其他分类不会覆盖你所写的方法，然而几率却小了很多，因为其他程序库很少会和你选用同一个前缀。这样做也能避免类的开发者以后在更新该类时所添加的方法与你在分类中添加的方法重名。比方说，假如苹果公司决定在 NSString 类里添加 urlEncodedString 方法，而你在分类中所写的方法又没加前缀，那么可能就会覆盖苹果公司的方法，这样做不合适，因为 NSString 类的其他使用者想得到由苹果公司实现的代码所输出的结果，而非你所返回的那个结果。还有一种可能，就是苹果公司编写的实现代码带有一些附加效果，该方法若为你所写的代码所覆盖，则会令对象内的数据互不一致，从而造成难于查找的 bug。

此外还要记住，如果向某个类的分类中加入方法，那么在应用程序中，该类的每个实例均可调用这些方法。比方说，若是向 NSString、NSArray、NSNumber 这种系统类里加入方法，那么这些类的每个实例均可调用你所加的方法，即便这些实例不是由你的代码创建出来的，也依然会如此。如果你无意中把自己分类里的方法名起得和其他分类一样，或是与第三方库

所添分类中的方法重名了，那么就可能出现奇怪的 bug，因为你以为此方法执行的是自己所写的那份代码，然而实际上却不是。与之相似，刻意覆写分类中的方法也不好，尤其是当你把代码发布为程序库供其他开发者使用，而他们又要依赖系统中现存的功能时，更不应该这么做。若是其他开发者又覆写了同一个方法，那么情况会更糟，因为无法确定最后到底会执行哪份实现代码。这又一次说明了为何要给分类中的方法名加上前缀。

要点

- 向第三方类中添加分类时，总应给其名称加上你专用的前缀。
- 向第三方类中添加分类时，总应给其中的方法名加上你专用的前缀。

第 26 条：勿在分类中声明属性

属性是封装数据的方式（参见第 6 条）。尽管从技术上说，分类里也可以声明属性，但这种做法还是要尽量避免。原因在于，除了“class-continuation 分类”（参见第 27 条）之外，其他分类都无法向类中新增实例变量，因此，它们无法把实现属性所需的实例变量合成出来。

比方说你实现过一个表示个人信息的类，在读过第 24 条之后，决定用分类机制将其代码分段。那么你可能会设计一个专门处理交友事务的分类，其中所有方法都与操作某人的朋友列表有关。若是不知道刚才讲的那个问题，可能就会把代表朋友列表的那项属性也放到 Friendship 分类里面去了：

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

@interface EOCPerson : NSObject
@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;

- (id)initWithFirstName:(NSString*)firstName
            andLastName:(NSString*)lastName;
@end

@implementation EOCPerson
// Methods
@end

@interface EOCPerson (Friendship)
@property (nonatomic, strong) NSArray *friends;
- (BOOL)isFriendsWith:(EOCPerson*)person;
@end

@implementation EOCPerson (Friendship)
// Methods
@end
```

编译这段代码时，编译器会给出如下警告信息：

```
warning: property 'friends' requires method 'friends' to be
defined - use @dynamic or provide a method implementation in
this category [-Wobjc-property-implementation]
warning: property 'friends' requires method 'setFriends:' to be
defined - use @dynamic or provide a method implementation in
this category [-Wobjc-property-implementation]
```

这段警告信息有点令人费解，意思是说此分类无法合成与 friends 属性相关的实例变量，所以开发者需要在分类中为该属性实现存取方法。此时可以把存取方法声明为 @dynamic，也就是说，这些方法等到运行期再提供，编译器目前是看不见的。如果决定使用消息转发机制（参见第 12 条）在运行期拦截方法调用，并提供其实现，那么或许可以采用这种做法。

关联对象（参见第 10 条）能够解决在分类中不能合成实例变量的问题。比方说，我们可以在分类中用下面这段代码实现存取方法：

```
#import <objc/runtime.h>

static const char *kFriendsPropertyKey = "kFriendsPropertyKey";

@implementation EOCPerson (Friendship)

- (NSArray*)friends {
    return objc_getAssociatedObject(self, kFriendsPropertyKey);
}

- (void)setFriends:(NSArray*)friends {
    objc_setAssociatedObject(self,
                             kFriendsPropertyKey,
                             friends,
                             OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}

@end
```

这样做可行，但不太理想。要把相似的代码写很多遍，而且在内存管理问题上容易出错，因为我们在为属性实现存取方法时，经常会忘记遵从其内存管理语义。比方说，你可能通过属性特质（attribute）修改了某个属性的内存管理语义。而此时还要记得，在设置方法中也得修改设置关联对象时所用的内存管理语义才行。所以说，尽管这个做法不坏，但笔者并不推荐。

此外，你可能会选用可变数组来实现 friends 属性所对应的实例变量。若是这样做，就得在设置方法中将传入的数组参数拷贝为可变版本，而这又成为另外一个编码时容易出错的地方。因此，把属性定义在“主接口”（main interface）中要比定义在分类里清晰得多。

在本例中，正确做法是把所有属性都定义在主接口里。类所封装的全部数据都应该定义在主接口中，这里是唯一能够定义实例变量（也就是数据）的地方。而属性只是定义实例变量及相关存取方法所用的“语法糖”，所以也应遵循同实例变量一样的规则。至于分类机制，

则应将其理解为一种手段，目标在于扩展类的功能，而非封装数据。

虽说如此，但有时候只读属性还是可以在分类中使用的。比方说，要在 NSCalendar 类中创建分类，以返回包含各个月份名称的字符串数组。由于获取方法并不访问数据，而且属性也不需要由实例变量来实现，所以可像下面这样来实现此分类：

```
@interface NSCalendar (EOC_Additions)
@property (nonatomic, strong, readonly) NSArray *eoc_allMonths;
@end

@implementation NSCalendar (EOC_Additions)
- (NSArray*)eoc_allMonths {
    if ([self.calendarIdentifier
            isEqualToString:NSGregorianCalendar])
    {
        return @[@"January", @"February",
                 @"March", @"April",
                 @"May", @"June",
                 @"July", @"August",
                 @"September", @"October",
                 @"November", @"December"];
    } else if ( /* other calendar identifiers */ ) {
        /* return months for other calendars */
    }
}
@end
```

由于实现属性所需的全部方法（在本例中，属性是只读的，所以只需实现一个方法）都已实现，所以不会再为该属性自动合成实例变量了。于是，编译器也就不会发出警告信息。然而，即便在这种情况下，也最好不要用属性。属性所要表达的意思是：类中有数据在支持着它。属性是用来封装数据的。在本例中，应该直接声明一个方法，用以获取月份名称列表：

```
@interface NSCalendar (EOC_Additions)
- (NSArray*)eoc_allMonths;
@end
```

要点

- 把封装数据所用的全部属性都定义在主接口里。
- 在“ class-continuation 分类”之外的其他分类中，可以定义存取方法，但尽量不要定义属性。

第 27 条：使用“class-continuation 分类”隐藏实现细节

类中经常会包含一些无须对外公布的方法及实例变量。其实这些内容也可以对外公布，并且写明其为私有，开发者不应依赖它们。Objective-C 动态消息系统（参见第 11 条）的工作

方式决定了其不可能实现真正的私有方法或私有实例变量。然而，我们最好还是只把确实需要对外公布的那部分内容公开。那么，这种不需对外公布但却应该具有的方法及实例变量应该怎么写呢？此时，这个特殊的“class-continuation 分类”就派上用场了。

“class-continuation 分类”和普通的分类不同，它必须定义在其所接续的那个类的实现文件里。其重要之处在于，这是唯一能声明实例变量的分类，而且此分类没有特定的实现文件，其中的方法都应该定义在类的主实现文件里。与其他分类不同，“class-continuation 分类”没有名字。比如，有个类叫做 EOCPerson，其“class-continuation 分类”写法如下：

```
@interface EOCPerson ()
// Methods here
@end
```

为什么需要有这种分类呢？因为其中可以定义方法和实例变量。为什么能在其中定义方法和实例变量呢？只因有“稳固的 ABI”这一机制（第 6 条详解了此机制），使得我们无须知道对象大小即可使用它。由于类的使用者不一定需要知道实例变量的内存布局，所以，它们也就未必非得定义在公共接口中了。基于上述原因，我们可以像在类的实现文件里那样，于“class-continuation 分类”中给类新增实例变量。只需在适当位置上多写几个括号，然后把实例变量放进去：

```
@interface EOCPerson () {
    NSString *_anInstanceVariable;
}
// Method declarations here
@end

@implementation EOCPerson {
    int _anotherInstanceVariable;
}
// Method implementations here
@end
```

这样做有什么好处呢？公共接口里本来就能定义实例变量。不过，把它们定义在“class-continuation 分类”或“实现块”中可以将其隐藏起来，只供本类使用。即便在公共接口里将其标注为 private，也还是会泄漏实现细节。比方说，你有个绝密的类，不想给其他人知道。假设你所写的某个类拥有那个绝密类的实例，而这个实例变量又声明在公共接口里面：

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

@class EOCSuperSecretClass;

@interface EOCClass : NSObject {
@private
    EOCSuperSecretClass *_secretInstance;
}
@end
```

那么，信息就泄漏了，别人就会知道有个名叫 EOCSuperSecretClass 的类。为解决此问题，可以不把实例变量声明为强类型，而是将其类型由 EOCSuperSecretClass* 改为 id。然而这么做不够好，因为在类的内部使用此实例时，无法得到编译器的帮助。没必要只因为想对外界隐藏某个内容就放弃编译器的辅助检查功能吧？这个问题可以由“class-continuation 分类”来解决。那个代表绝密类的实例可以声明成这样：

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

@interface EOCClass : NSObject
@end

// EOCClass.m
#import "EOCClass.h"
#import "EOCSuperSecretClass.h"

@interface EOCClass () {
    EOCSuperSecretClass *_secretInstance;
}
@end

@implementation EOCClass
// Methods here
@end
```

实例变量也可以定义在实现块里，从语法上说，这与直接添加到“class-continuation 分类”等效，只是看个人喜好了。笔者喜欢将其添加在“class-continuation 分类”中，以便将全部数据定义都放在一处。由于“class-continuation 分类”里还能定义一些属性，所以在这里额外声明一些实例变量也很合适。这些实例变量并非真的私有，因为在运行期总可以调用某些方法绕过此限制，不过，从一般意义上来说，它们还是私有的。此外，由于没有声明在公共头文件里，所以将代码作为程序库的一部分来发行时，其隐藏程度更好。

编写 Objective-C++ 代码时“class-continuation 分类”也尤为有用。Objective-C++ 是 Objective-C 与 C++ 的混合体，其代码可以用这两种语言来编写。由于兼容性原因，游戏后端一般用 C++ 来写。另外，有时候要使用的第三方库可能只有 C++ 绑定，此时也必须使用 C++ 来编码。在这些情况下，使用“class-continuation 分类”会很方便。假设某个类打算这样写：

```
#import <Foundation/Foundation.h>
#include "SomeCppClass.h"
@interface EOCClass : NSObject {
@private
    SomeCppClass _cppClass;
}
@end
```

该类的实现文件可能叫做 EOCClass.mm，其中 .mm 扩展名表示编译器应该将此文件按 Objective-C++ 来编译，否则，就无法正确引入 SomeCppClass.h 了。然而请注意，名为 SomeCppClass 的这个 C++ 类必须完全引入，因为编译器要完整地解析其定义方能得知 _cppClass 实例变量的大小。于是，只要是包含 EOCClass.h 的类，都必须编译为 Objective-C++ 才行，因为它们都引入了 SomeCppClass 类的头文件。这很快就会失控，最终导致整个应用程序全部都要编译为 Objective-C++。这么做确实完全可行，不过笔者觉得相当别扭，尤其是将代码发布为程序库供其他应用程序使用时，更不应该如此。要求第三方开发者将其源文件扩展名均改为 .mm 不是很合适。

你可能认为解决此问题的办法是：不引入 C++ 类的头文件，只是向前声明该类，并且将实例变量做成指向此类的指针。

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

class SomeCppClass;

@interface EOCClass : NSObject {
@private
    SomeCppClass *_cppClass;
}
@end
```

现在实例变量必须是指针，若不是，则编译器无法得知其大小，从而会报错。但所有指针的大小确实都是固定的，于是编译器只需知道其所指的类型即可。不过，这么做还是会遇到刚才那个问题，因为引入 EOCClass 头文件的源码里都包含 class 关键字，而这是 C++ 的关键字，所以仍然需要按 Objective-C++ 来编译才行。这样做既别扭又无必要，因为该实例变量毕竟是 private 的，其他类为什么要知道它呢？这个问题还是得用“ class-continuation 分类”来解决。将刚才那个类改写之后，其代码如下：

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

@interface EOCClass : NSObject
@end
```

```
// EOCClass.mm
#import "EOCClass.h"
#include "SomeCppClass.h"

@interface EOCClass () {
    SomeCppClass _cppClass;
}
@end

@implementation EOCClass
@end
```

改写后的EOCClass类，其头文件里就没有C++代码了，使用头文件的人甚至意识不到其底层实现代码中混有C++成分。某些系统库用到了这种模式，比如网页浏览器框架WebKit，其大部分代码都以C++编写，然而对外展示出来的却是一套整洁的Objective-C接口。CoreAnimation里面也用到了此模式，它的许多后端代码都用C++写成，但对外公布的却是一套纯Objective-C接口。

“class-continuation分类”还有一种合理用法，就是将public接口中声明为“只读”的属性扩展为“可读写”，以便在类的内部设置其值。我们通常不直接访问实例变量，而是通过设置访问方法来做（参见第7条），因为这样能够触发“键值观测”（Key-Value Observing，KVO）通知，其他对象有可能正监听此事件。出现在“class-continuation分类”或其他分类中的属性必须同类接口里的属性具备相同的特质（attribute），不过，其“只读”状态可以扩充为“可读写”。例如，有个描述个人信息的类，其公共接口如下：

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

@interface EOCPerson : NSObject
@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;

- (id)initWithFirstName:(NSString*)firstName
               lastName:(NSString*)lastName;
@end
```

我们一般会在“class-continuation分类”中把这两个属性扩展为“可读写”：

```
@interface EOCPerson ()
@property (nonatomic, copy, readwrite) NSString *firstName;
@property (nonatomic, copy, readwrite) NSString *lastName;
@end
```

只需要用上面几行代码就行了。现在EOCPerson的实现代码可以随意调用“setFirstName:”或“setLastName:”这两个设置方法，也可以用“点语法”来设置属性。这样做很有用，既能令外界无法修改对象，又能在其内部按照需要管理其数据。这样，封装在类中的数据就由实例本身来控制，而外部代码则无法修改其值。第18条曾详述了这一话题。请注意，若观察者（observer）[⊖]正读取属性值而内部代码又在写入该属性时，则有可能引发“竞争条件”（race condition）。合理使用同步机制（参见第41条）能缓解此问题。

只会在类的实现代码中用到的私有方法也可以声明在“class-continuation分类”中。这么做比较合适，因为它描述了那些只在类实现代码中才会使用的方法。这些方法可以这样写：

```
@interface EOCPerson ()
- (void)p_privateMethod;
@end
```

⊖ 也称“监听器”。——译者注

此处根据第 20 条所述的建议为方法名加了前缀，以体现其为私有方法。新版编译器不强制要求开发者在使用方法之前必须先声明。然而像上面这样在“class-continuation 分类”中声明一下通常还是有好处的，因为这样做可以把类里所含的相关方法都统一描述于此。笔者在编写类的实现代码之前，经常喜欢像这样先把方法原型写出来，然后再逐个实现。要想使类的代码更易读懂，可以试试这个好办法。

最后还要讲一种用法：若对象所遵从的协议只应视为私有，则可在“class-continuation 分类”中声明。有时由于对象所遵从的某个协议在私有 API 中，所以我们可能不太想在公共接口中泄漏这一信息。比方说，EOCPerson 遵从了名为 EOCSecretDelegate 的协议。如果声明在公共接口里，那么要像下面这样来写：

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

@interface EOCPerson : NSObject<EOCSecretDelegate>
@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;

- (id)initWithFirstName:(NSString*)firstName
               lastName:(NSString*)lastName;
@end
```

你可能会说，只需要向前声明 EOCSecretDelegate 协议就可以不引入它了（或者说，不引入定义该协议的头文件了）。用下面这行向前声明语句来取代 #import 指令：

```
@protocol EOCSecretDelegate;
```

但是这样一来，只要引入 EOCPerson 头文件的地方，编译器都会给出下列警告信息：

```
warning: cannot find protocol definition for 'EOCSecretDelegate'
```

由于编译器看不到协议的定义，所以无法得知其中所含的方法，于是就会像这样警告开发者。然而，这毕竟是个私有的内部协议，你甚至连名字都不想给别人知道。此时还得请“class-continuation 分类”来帮忙。不要在公共接口中声明 EOCPerson 类遵从了 EOCSecretDelegate 协议，而是改到“class-continuation 分类”里面声明：

```
#import "EOCPerson.h"
#import "EOCSecretDelegate.h"

@interface EOCPerson () <EOCSecretDelegate>
@end

@implementation EOCPerson
/* ... */
@end
```

公共接口内所有提到 EOCSecretDelegate 的地方都可删去。这个私有协议现在已经不为

外界所知了，使用 EOCPerson 的人若不深入探索一番，则很难发现其身影。

要点

- 通过“class-continuation 分类”向类中新增实例变量。
- 如果某属性在主接口中声明为“只读”，而类的内部又要用设置方法修改此属性，那么就在“class-continuation 分类”中将其扩展为“可读写”。
- 把私有方法的原型声明在“class-continuation 分类”里面。
- 若想使类所遵循的协议不为人所知，则可于“class-continuation 分类”中声明。

第 28 条：通过协议提供匿名对象

协议定义了一系列方法，遵从此协议的对象应该实现它们（如果这些方法不是可选的，那么就必须实现）。于是，我们可以用协议把自己所写的 API 之中的实现细节隐藏起来，将返回的对象设计为遵从此协议的纯 id 类型。这样的话，想要隐藏的类名就不会出现在 API 之中了。若是接口背后有多个不同的实现类，而你又不想指明具体使用哪个类，那么可以考虑用这个办法——因为有时候这些类可能会变，有时候它们又无法容纳于标准的类继承体系中，因而不能以某个公共基类来统一表示。

此概念经常称为“匿名对象”(anonymous object)，这与其他语言中的“匿名对象”不同，在那些语言中，该词是指以内联形式所创建出来的无名类[⊖]，而此词在 Objective-C 中则不是这个意思。第 23 条解释了委托与数据源对象，其中就曾用到这种匿名对象。例如，在定义“受委托者”(delegate) 这个属性时，可以这样写：

```
@property (nonatomic, weak) id <EOCDelegate> delegate;
```

由于该属性的类型是 id<EOCDelegate>，所以实际上任何类的对象都能充当这一属性，即便该类不继承自 NSObject 也可以，只要遵循 EOCDelegate 协议就行。对于具备此属性的类来说，delegate 就是“匿名的”（anonymous）。如有需要，可在运行期查出此对象所属的类型（参见第 14 条）。然而这样做不太好，因为指定属性类型时所写的那个 EOCDelegate 契约已经表明此对象的具体类型无关紧要了。

NSDictionary 也能实际说明这一概念。在字典中，键的标准内存管理语义是“设置时拷贝”，而值的语义则是“设置时保留”。因此，在可变版本的字典中，设置键值对所用的方法的签名是：

```
- (void)setObject:(id)object forKey:(id<NSCopying>)key
```

表示键的那个参数其类型为 id<NSCopying>，作为参数值的对象，它可以是任何类型，只要遵从 NSCopying 协议就好，这样的话，就能向该对象发送拷贝消息了（参见第 22 条）。

⊖ 这个概念在某些语言中也叫“匿名类”(anonymous class)。——译者注

这个 key 参数可以视为匿名对象。与 delegate 属性一样，字典也不关心 key 对象所属的具体类，而且它也决不应该依赖于此。字典对象只要能确定它可以给此实例发送拷贝消息就行了。

处理数据库连接（database connection）的程序库也用这个思路，以匿名对象来表示从另一个库中所返回的对象。对于处理连接所用的那个类，你也许不想叫外人知道其名字，因为不同的数据库可能要用不同的类来处理。如果没办法令其都继承自同一基类，那么就得返回 id 类型的东西了。不过我们可以把所有数据库连接都具备的那些方法放到协议中，令返回的对象遵从此协议。协议可以这样写：

```
@protocol EOCDatabaseConnection
- (void)connect;
- (void)disconnect;
- (BOOL)isConnected;
- (NSArray*)performQuery:(NSString*)query;
@end
```

然后，就可以用“数据库处理器”（database handler）单例来提供数据库连接了。这个单例的接口可以写成：

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

@protocol EOCDatabaseConnection;

@interface EOCDatabaseManager : NSObject
+ (id)sharedInstance;
- (id<EOCDatabaseConnection>)connectionWithIdentifier:
                                (NSString*)identifier;
@end
```

这样的话，处理数据库连接所用的类的名称就不会泄漏了，有可能来自不同框架的那些类现在均可以经由同一个方法来返回了。使用此 API 的人仅仅要求所返回的对象能用来连接、断开并查询数据库即可。这一点很重要。本例中，处理数据库连接所用的后端代码可能使用了各种第三方库来连接不同类型的数据库（例如 MySQL、PostgreSQL 等）。由于这些类都在多个第三方库里，所以也许没办法令所有的连接类都继承自同一基类。因此，可以创建匿名对象把这些第三方类简单包裹一下，使匿名对象成为其子类，并遵从 EOCDatabaseConnection 协议。然后，用“connectionWithIdentifier:”方法来返回这些类对象。在开发后续版本时，无须改变公共 API，即可切换后端的实现类。

有时对象类型并不重要，重要的是对象有没有实现某些方法，在此情况下，也可以用这些“匿名类型”（anonymous type）来表达这一概念。即便实现代码总是使用固定的类，你可能还是会把它写成遵从某协议的匿名类型，以表示类型在此处并不重要。

CoreData 框架里也有这种用法。查询 CoreData 数据库所得的结果由名叫 NSFetchedResultsController 的类来处理，如有需要，处理时还会把数据分区。在负责处理查询结果的控制器中，有个 sections 属性，用以表示数据分区。此属性是个数组，但其中的对

象却没有指明具体类型，只是说这些对象都遵从了 NSFetchedResultsSectionInfo 协议。下面这段代码通过控制器来获取数据分区信息：

```
NSFetchedResultsController *controller = /* some controller */;
NSUInteger section = /* section index to query */;

NSArray *sections = controller.sections;
id <NSFetchedResultsSectionInfo> sectionInfo = sections[section];
NSUInteger numberOfObjects = sectionInfo.numberOfObjects;
```

sectionInfo 是个匿名对象。设计此种 API 时，要把“通过对象能够访问数据分区信息”这一功能于接口中清晰地表达出来。在幕后，此对象可能是由处理结果的控制器所创建的内部状态对象（internal state object）。没必要把表示此种数据的类对外公布，因为使用控制器的人绝对不用关心查询结果中的数据分区是如何保存的，他们只需要知道可以在这些对象上查询数据就行了。我们可以把 section 数组中返回的内部状态对象视为遵从 NSFetchedResultsSectionInfo 协议的匿名对象。使用者只要明白这种对象实现了某些特定的方法即可，其余实现细节都隐藏起来了。

要点

- 协议可在某种程度上提供匿名类型。具体的对象类型可以淡化成遵从某协议的 id 类型，协议里规定了对象所应实现的方法。
- 使用匿名对象来隐藏类型名称（或类名）。
- 如果具体类型不重要，重要的是对象能够响应（定义在协议里的）特定方法，那么可使用匿名对象来表示。

第5章
内存管理

在 Objective-C 这种面向对象语言里，内存管理是个重要概念。要想用一门语言写出内存使用效率高而且又没有 bug 的代码，就得掌握其内存管理模型的种种细节。

一旦理解了这些规则，你就会发现，其实 Objective-C 的内存管理没那么复杂，而且有了“自动引用计数”（Automatic Reference Counting，ARC）之后，就变得更为简单了。ARC 几乎把所有内存管理事宜都交由编译器来决定，开发者只需专注于业务逻辑。

第 29 条：理解引用计数

Objective-C 语言使用引用计数来管理内存，也就是说，每个对象都有个可以递增或递减的计数器。如果想使某个对象继续存活，那就递增其引用计数；用完了之后，就递减其计数。计数变为 0，就表示没人关注此对象了，于是，就可以把它销毁。上面这几句话只是个概述，要想写出优秀的 Objective-C 代码，必须完全理解此问题才行，即便打算用 ARC 来编码（参见第 30 条）也是如此。

从 Mac OS X 10.8 开始，“垃圾收集器”（garbage collector）已经正式废弃了，以 Objective-C 代码编写 Mac OS X 程序时不应再使用它，而 iOS 则从未支持过垃圾收集。因此，掌握引用计数机制对于学好 Objective-C 来说十分重要。Mac OS X 程序已经不能再依赖垃圾收集器了，而 iOS 系统不支持此功能，将来也不会支持。

已经用过 ARC 的人可能会知道：所有与引用计数有关的方法都无法编译，然而现在先暂时忘掉这件事。那些方法确实无法用在 ARC 中，不过本条就是要从 Objective-C 的角度讲解引用计数，而 ARC 实际上也是一种引用计数机制，所以，还是要谈谈这些在开启 ARC 功能时不能直接调用的方法。

引用计数工作原理

在引用计数架构下，对象有个计数器，用以表示当前有多少个事物想令此对象继续存活下去。这在 Objective-C 中叫做“保留计数”（retain count），不过也可以叫“引用计数”

（reference count）。NSObject 协议声明了下面三个方法用于操作计数器，以递增或递减其值：

- Retain　递增保留计数。
- release　递减保留计数。
- autorelease　待稍后清理“自动释放池”（autorelease pool）时，再递减保留计数。（本条后面几页与本书第 34 条将会详细讲解“自动释放池”。）

查看保留计数的方法叫做 retainCount，此方法不太有用，即便在调试时也如此，所以笔者（与苹果公司）并不推荐大家使用这个方法。更多内容请参阅第 36 条。

对象创建出来时，其保留计数至少为 1。若想令其继续存活，则调用 retain 方法。要是某部分代码不再使用此对象，不想令其继续存活，那就调用 release 或 autorelease 方法。最终当保留计数归零时，对象就回收了（deallocated），也就是说，系统会将其占用的内存标记为“可重用”（reuse）。此时，所有指向该对象的引用也都变得无效了。

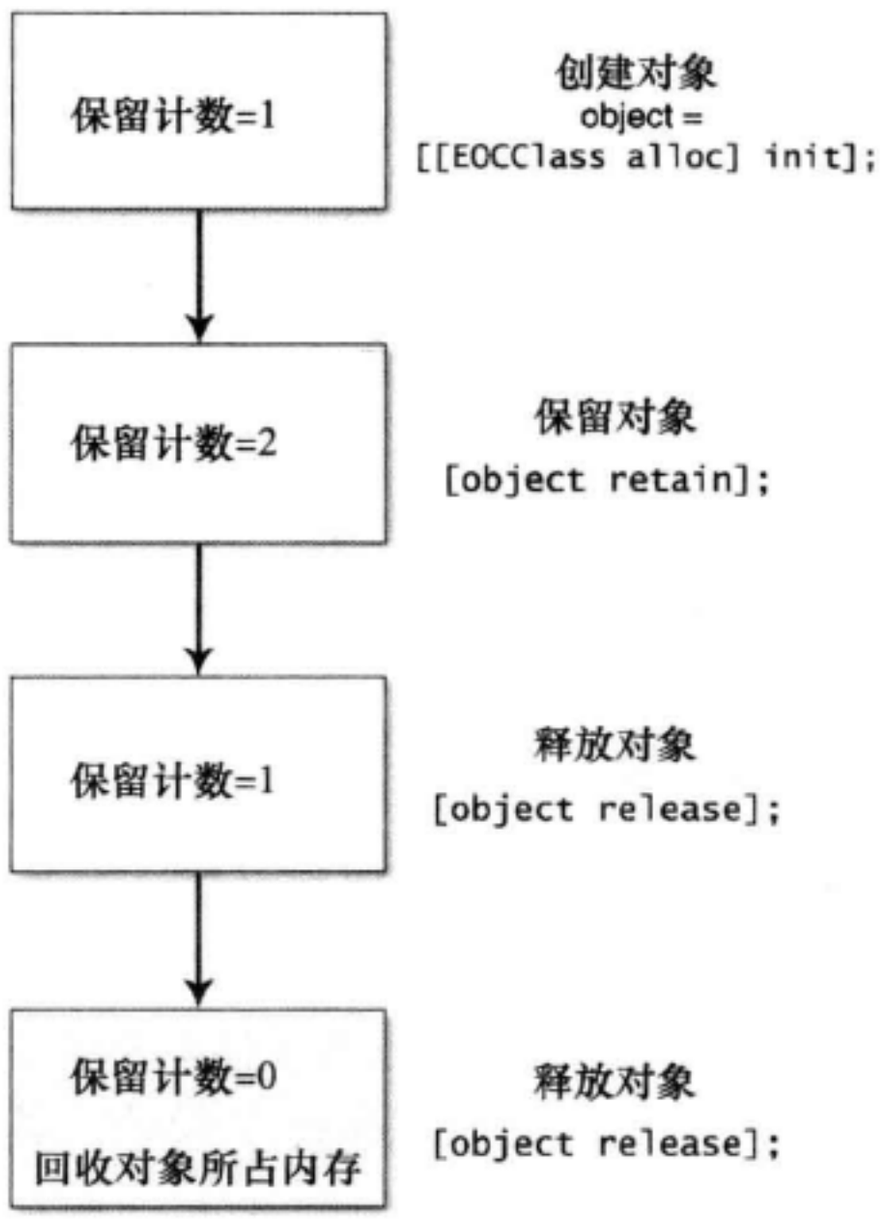

图 5-1　在对象的生命期中，其保留计数时而递增，时而递减，最终归零

图 5-1 演示了对象自创造出来之后历经一次“保留”及两次“释放”操作的过程。

应用程序在其生命期中会创建很多对象，这些对象都相互联系着。例如，表示个人信息的对象会引用另一个表示人名的字符串对象，而且可能还会引用其他个人信息对象，比如在存放朋友的 set 中就是如此，于是，这些相互关联的对象就构成了一张“对象图”（object graph）。对象如果持有指向其他对象的强引用（strong reference），那么前者就“拥有”（own）后者。也就是说，对象想令其所引用的那些对象继续存活，就可将其“保留”。等用完了之后，再释放。

在图 5-2 所示的对象图中，ObjectB 与 ObjectC 都引用了 ObjectA。若 ObjectB 与 ObjectC 都不再使用 ObjectA，则其保留计数降为 0，于是便可摧毁了。还有其他对象想令 ObjectB 与 ObjectC 继续存活，而应用程序里又有另外一些对象想令那些对象继续存活。如果按“引用树”回溯，那么最终会发现一个“根对象”（root object）。在 Mac OS X 应用程序中，此对象就是 NSApplication 对象；而在 iOS 应用程序中，则是 UIApplication 对象。两者都是应用程序启动时创建的单例。

下面这段代码有助于理解这些方法的用法：

```
NSMutableArray *array = [[NSMutableArray alloc] init];

NSNumber *number = [[NSNumber alloc] initWithInt:1337];
```

```
[array addObject:number];
[number release];

// do something with 'array'

[array release];
```

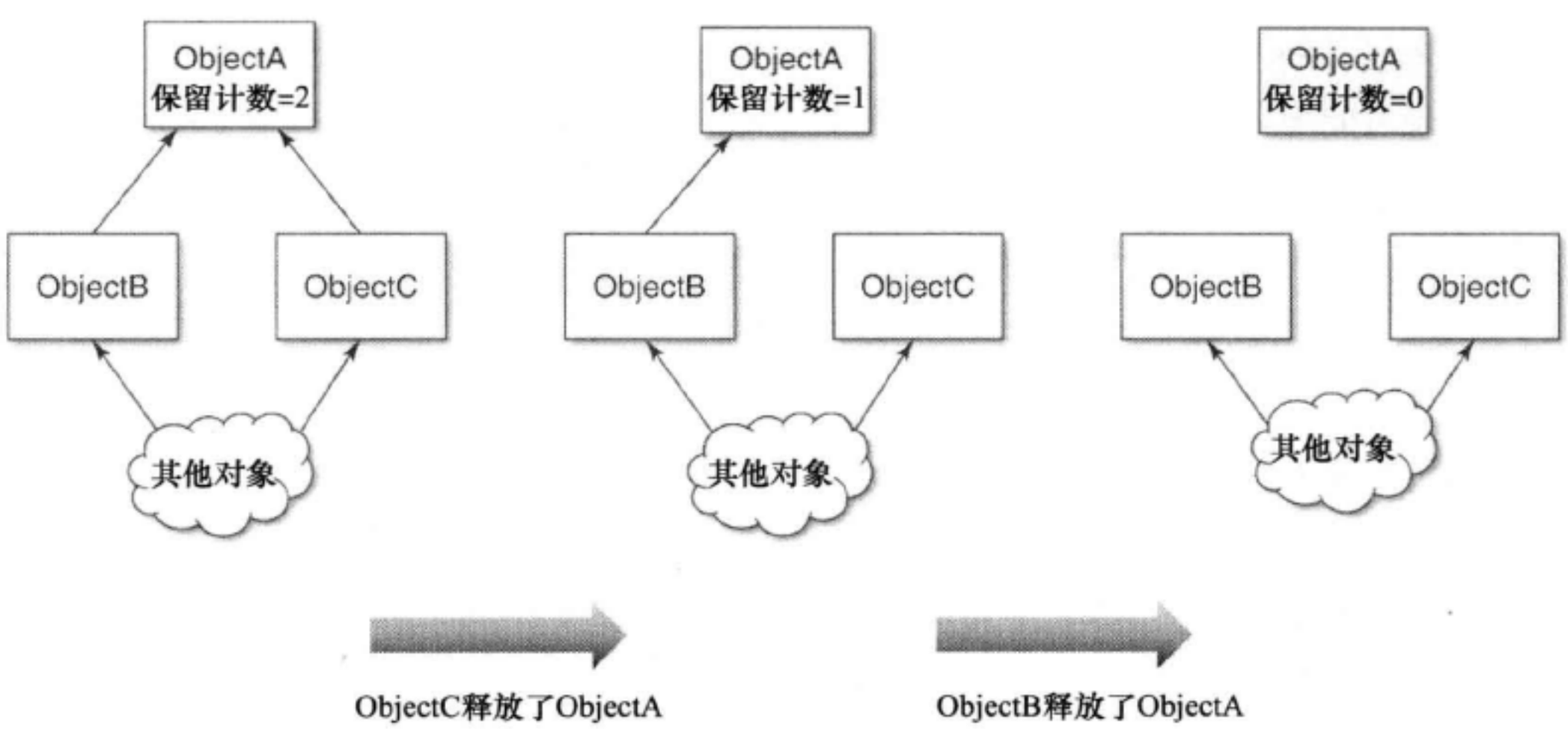

图 5-2　对象图里所有指向 ObjectA 对象的引用均释放之后，此对象所占内存亦可回收

如前所述，由于代码中直接调用了 release 方法，所以在 ARC 下无法编译。在 Objective-C 中，调用 alloc 方法所返回的对象由调用者所拥有。也就是说，调用者已通过 alloc 方法表达了想令该对象继续存活下去的意愿。不过请注意，这并不是说对象此时的保留计数必定是 1。在 alloc 或“ initWithInt:”方法的实现代码中，也许还有其他对象也保留了此对象，所以，其保留计数可能会大于 1。能够肯定的是：保留计数至少为 1。保留计数这个概念就应该这样来理解才对。绝不应该说保留计数一定是某个值，只能说你所执行的操作是递增了该计数还是递减了该计数。

创建完数组后，把 number 对象加入其中。调用数组的“ addObject:”方法时，数组也会在 number 上调用 retain 方法，以期继续保留此对象。这时，保留计数至少为 2。接下来，代码不再需要 number 对象了，于是将其释放。现在的保留计数至少为 1。这样就不能照常使用 number 变量了。调用 release 之后，已经无法保证所指的对象仍然存活。当然，根据本例中的代码，我们显然知道 number 对象在调用了 release 之后仍然存活，因为数组还在引用着它。然而绝不应假设此对象一定存活，也就是说，不要像下面这样编写代码：

```
NSNumber *number = [[NSNumber alloc] initWithInt:1337];
[array addObject:number];
[number release];
NSLog(@"number = %@", number);
```

即便上述代码在本例中可以正常执行，也仍然不是个好办法。如果调用 release 之后，基

于某些原因，其保留计数降至0，那么number对象所占内存也许会回收，这样的话，再调用NSLog可能就将使程序崩溃了。笔者在这里只说“可能”，而没说“一定”，因为对象所占的内存在“解除分配”（deallocated）之后，只是放回“可用内存池”（avaiable pool）。如果执行NSLog时尚未覆写对象内存，那么该对象仍然有效，这时程序不会崩溃。由此可见：因过早释放对象而导致的bug很难调试。

为避免在不经意间使用了无效对象，一般调用完release之后都会清空指针。这就能保证不会出现可能指向无效对象的指针，这种指针通常称为“悬挂指针”(dangling pointer)[⊖]。比方说，可以这样编写代码来防止此情况发生：

```
NSNumber *number = [[NSNumber alloc] initWithInt:1337];
[array addObject:number];
[number release];
number = nil;
```

属性存取方法中的内存管理

如前所述，对象图由互相关联的对象所构成。刚才那个例子中的数组通过在其元素上调用retain方法来保留那些对象。不光是数组，其他对象也可以保留别的对象，这一般通过访问“属性”（参见第6条）来实现，而访问属性时，会用到相关实例变量的获取方法及设置方法。若属性为“strong关系”（strong relationship），则设置的属性值会保留。比方说，有个名叫foo的属性由名为_foo的实例变量所实现，那么，该属性的设置方法会是这样：

```
- (void)setFoo:(id)foo {
    [foo retain];
    [_foo release];
    _foo = foo;
}
```

此方法将保留新值并释放旧值，然后更新实例变量，令其指向新值。顺序很重要。假如还未保留新值就先把旧值释放了，而且两个值又指向同一个对象，那么，先执行的release操作就可能导致系统将此对象永久回收。而后续的retain操作则无法令这个已经彻底回收的对象复生，于是实例变量就成了悬挂指针。

自动释放池

在Objective-C的引用计数架构中，自动释放池是一项重要特性。调用release会立刻递减对象的保留计数（而且还有可能令系统回收此对象），然而有时候可以不调用它，改为调用autorelease，此方法会在稍后递减计数，通常是在下一次“事件循环”（event loop）时递减，不过也可能执行得更早些（参见第34条）。

此特性很有用，尤其是在方法中返回对象时更应该用它。在这种情况下，我们并不总是想令方法调用者手工保留其值。比方说，有下面这个方法：

⊖ 亦称“迷途指针”、“悬垂指针”、“悬摆指针”。——译者注

```
- (NSString*)stringValue {
    NSString *str = [[NSString alloc]
                        initWithFormat:@"I am this: %@", self];
    return str;
}
```

此时返回的 str 对象其保留计数比期望值要多 1（+1 retain count），因为调用 alloc 会令保留计数加 1，而又没有与之对应的释放操作。保留计数多 1，就意味着调用者要负责处理多出来的这一次保留操作。必须设法将其抵消。这并不是说保留计数本身就一定是 1，它可能大于 1，不过那取决于“ initWithFormat:”方法内的实现细节。你要考虑的是如何将多出来的这一次保留操作抵消掉。

但是，不能在方法内释放 str，否则还没等方法返回，系统就把该对象回收了。这里应该用 autorelease，它会在稍后释放对象，从而给调用者留下了足够长的时间，使其可以在需要时先保留返回值。换句话说，此方法可以保证对象在跨越“方法调用边界”（method call boundary）后一定存活。实际上，释放操作会在清空最外层的自动释放池（参见第 34 条）时执行，除非你有自己的自动释放池，否则这个时机指的就是当前线程的下一次事件循环。改写 stringValue 方法，使用 autorelease 来释放对象：

```
- (NSString*)stringValue {
    NSString *str = [[NSString alloc]
                        initWithFormat:@"I am this: %@", self];
    return [str autorelease];
}
```

修改之后，stringValue 方法把 NSString 对象返回给调用者时，此对象必然存活。所以我们能够像下面这样使用它：

```
NSString *str = [self stringValue];
NSLog(@"The string is: %@", str);
```

由于返回的 str 对象将于稍后自动释放，所以多出来的那一次保留操作到时自然就会抵消，无须再执行内存管理操作。因为自动释放池中的释放操作要等到下一次事件循环时才会执行，所以 NSLog 语句在使用 str 对象前不需要手工执行保留操作。但是，假如要持有此对象的话（比如将其设置给实例变量），那就需要保留，并于稍后释放：

```
_instanceVariable = [[self stringValue] retain];
// ...
[_instanceVariable release];
```

由此可见，autorelease 能延长对象生命期，使其在跨越方法调用边界后依然可以存活一段时间。

保留环

使用引用计数机制时，经常要注意的一个问题就是“保留环”（retain cycle），也就是呈环

状相互引用的多个对象。这将导致内存泄漏，因为循环中的对象其保留计数不会降为0。对于循环中的每个对象来说，至少还有另外一个对象引用着它。图5-3里的每个对象都引用了另外两个对象之中的一个。在这个循环里，所有对象的保留计数都是1。

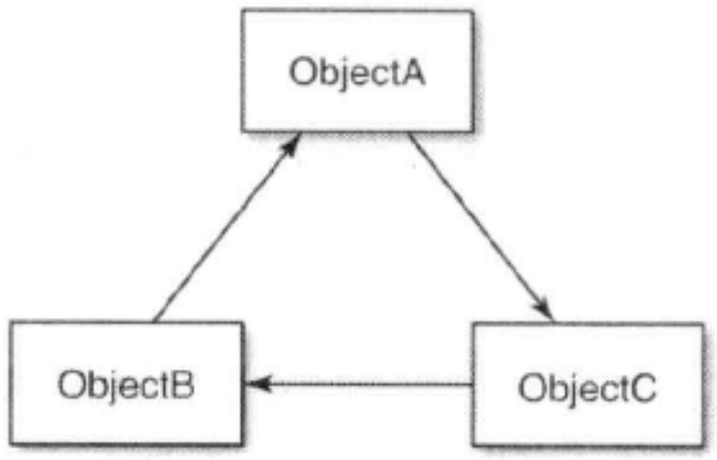

图5-3 对象图中的保留环

在垃圾收集环境中，通常将这种情况认定为“孤岛”（island of isolation）。此时，垃圾收集器会把三个对象全都回收走。而在Objective-C的引用计数架构中，则享受不到这一便利。通常采用“弱引用”（weak reference，参见第33条）来解决此问题，或是从外界命令循环中的某个对象不再保留另外一个对象。这两种办法都能打破保留环，从而避免内存泄漏。

要点

- 引用计数机制通过可以递增递减的计数器来管理内存。对象创建好之后，其保留计数至少为1。若保留计数为正，则对象继续存活。当保留计数降为0时，对象就被销毁了。
- 在对象生命期中，其余对象通过引用来保留或释放此对象。保留与释放操作分别会递增及递减保留计数。

第30条：以ARC简化引用计数

引用计数这个概念相当容易理解（参见第29条）。需要执行保留与释放操作的地方也很容易就能看出来。所以Clang编译器项目带有一个“静态分析器”（static analyzer），用于指明程序里引用计数出问题的地方。举个例子，假设下面这段代码采用手工方式管理引用计数：

```
if ([self shouldLogMessage]) {
    NSString *message = [[NSString alloc] initWithFormat:
                         @"I am object, %p", self];
    NSLog(@"message = %@", message);
}
```

此代码有内存泄漏问题，因为if语句块末尾并未释放message对象。由于在if语句之外无法引用message，所以此对象所占的内存泄漏了[⊖]。判定内存是否泄漏所用的规则很简明：调用NSString的alloc方法所返回的那个message对象的保留计数比期望值要多1。然而却没有与之对应的释放操作来抵消。因为这些规则很容易表述，所以计算机可以简单地将其套用在程序上，从而分析出有内存泄漏问题的对象。这正是“静态分析器”要做的事。

静态分析器还有更为深入的用途。既然可以查明内存管理问题，那么应该也可以根据需要，预先加入适当的保留或释放操作以避免这些问题，对吧？自动引用计数这一思路正是源

⊖ 这里“泄漏”的意思是：没有正确释放已经不再使用的内存。——译者注

于此。自动引用计数所做的事情与其名称相符，就是自动管理引用计数。于是，在前面那段代码的 if 语句块结束之前，可以于 message 对象上自动执行 release 操作，也就是把代码自动改写为下列形式：

```
if ([self shouldLogMessage]) {
    NSString *message = [[NSString alloc] initWithFormat:
                         @"I am object, %p", self];
    NSLog(@"message = %@", message);
    [message release]; ///< Added by ARC
}
```

使用 ARC 时一定要记住，引用计数实际上还是要执行的，只不过保留与释放操作现在是由 ARC 自动为你添加。稍后将会看到，除了为方法所返回的对象正确运用内存管理语义之外，ARC 还有更多的功能。不过，ARC 的那些功能都是基于核心的内存管理语义而构建的，这套标准语义贯穿于整个 Objective-C 语言。

由于 ARC 会自动执行 retain、release、autorelease 等操作，所以直接在 ARC 下调用这些内存管理方法是非法的。具体来说，不能调用下列方法：

- retain
- release
- autorelease
- dealloc

直接调用上述任何方法都会产生编译错误，因为 ARC 要分析何处应该自动调用内存管理方法，所以如果手工调用的话，就会干扰其工作。此时必须信赖 ARC，令其帮你正确处理内存管理事宜，而这会使那些惯于手动管理引用计数的开发者不太放心。

实际上，ARC 在调用这些方法时，并不通过普通的 Objective-C 消息派发机制，而是直接调用其底层 C 语言版本。这样做性能更好，因为保留及释放操作需要频繁执行，所以直接调用底层函数能节省很多 CPU 周期。比方说，ARC 会调用与 retain 等价的底层函数 objc_retain。这也是不能覆写 retain、release 或 autorelease 的缘由，因为这些方法从来不会被直接调用。笔者在本节后面的文字中将用等价的 Objective-C 方法来指代与之相关的底层 C 语言版本，这对于那些手动管理过引用计数的开发者来说更易理解。

使用 ARC 时必须遵循的方法命名规则

将内存管理语义在方法名中表示出来早已成为 Objective-C 的惯例，而 ARC 则将之确立为硬性规定。这些规则简单地体现在方法名上。若方法名以下列词语开头，则其返回的对象归调用者所有：

- alloc
- new
- copy
- mutableCopy

归调用者所有的意思是：调用上述四种方法的那段代码要负责释放方法所返回的对象。也就是说，这些对象的保留计数是正值，而调用了这四种方法的那段代码要将其中一次保留操作抵消掉。如果还有其他对象保留此对象，并对其调用了 autorelease，那么保留计数的值可能比 1 大，这也是 retainCount 方法不太有用的原因之一（参见第 36 条）。

若方法名不以上述四个词语开头，则表示其所返回的对象并不归调用者所有。在这种情况下，返回的对象会自动释放，所以其值在跨越方法调用边界后依然有效。要想使对象多存活一段时间，必须令调用者保留它才行。

维系这些规则所需的全部内存管理事宜均由 ARC 自动处理，其中也包括在将要返回的对象上调用 autorelease，下列代码演示了 ARC 的用法：

```
+ (EOCPerson*)newPerson {
    EOCPerson *person = [[EOCPerson alloc] init];
    return person;
    /**
     * The method name begins with 'new', and since 'person'
     * already has an unbalanced +1 retain count from the
     * 'alloc', no retains, releases, or autoreleases are
     * required when returning.
     */
}

+ (EOCPerson*)somePerson {
    EOCPerson *person = [[EOCPerson alloc] init];
    return person;
    /**
     * The method name does not begin with one of the "owning"
     * prefixes, therefore ARC will add an autorelease when
     * returning 'person'.
     * The equivalent manual reference counting statement is:
     *   return [person autorelease];
     */
}

- (void)doSomething {
    EOCPerson *personOne = [EOCPerson newPerson];
    // ...

    EOCPerson *personTwo = [EOCPerson somePerson];
    // ...

    /**
     * At this point, 'personOne' and 'personTwo' go out of
     * scope, therefore ARC needs to clean them up as required.
     * - 'personOne' was returned as owned by this block of
     *   code, so it needs to be released.
     * - 'personTwo' was returned not owned by this block of
     *   code, so it does not need to be released.
```

```
     * The equivalent manual reference counting cleanup code
     * is:
     *     [personOne release];
     */
}
```

ARC 通过命名约定将内存管理规则标准化，初学此语言的人通常觉得这有些奇怪，其他编程语言很少像 Objective-C 这样强调命名。但是，想成为优秀的 Objective-C 程序员就必须适应这套理念。在编码过程中，ARC 能帮程序员做许多事情。

除了会自动调用“保留”与“释放”方法外，使用 ARC 还有其他好处，它可以执行一些手工操作很难甚至无法完成的优化。例如，在编译期，ARC 会把能够互相抵消的 retain、release、autorelease 操作约简。如果发现在同一个对象上执行了多次“保留”与“释放”操作，那么 ARC 有时可以成对地移除这两个操作。

ARC 也包含运行期组件。此时所执行的优化很有意义，大家看过之后就会明白为何以后的代码都应该用 ARC 来写了。前面讲到，某些方法在返回对象前，为其执行了 autorelease 操作，而调用方法的代码可能需要将返回的对象保留，比如像下面这种情况就是如此：

```
// From a class where _myPerson is a strong instance variable
_myPerson = [EOCPerson personWithName:@"Bob Smith"];
```

调用“personWithName:”方法会返回新的 EOCPerson 对象，而此方法在返回对象之前，为其调用了 autorelease 方法。由于实例变量是个强引用，所以编译器在设置其值的时候还需要执行一次保留操作。因此，前面那段代码与下面这段手工管理引用计数的代码等效：

```
EOCPerson *tmp = [EOCPerson personWithName:@"Bob Smith"];
_myPerson = [tmp retain];
```

此时应该能看出来，“personWithName:”方法里的 autorelease 与上段代码中的 retain 都是多余的。为提升性能，可将二者删去。但是，在 ARC 环境下编译代码时，必须考虑“向后兼容性”（backward compatibility），以兼容那些不使用 ARC 的代码。其实本来 ARC 也可以直接舍弃 autorelease 这个概念，并且规定，所有从方法中返回的对象其保留计数都比期望值多 1。但是，这样做就破坏了向后兼容性。

不过，ARC 可以在运行期检测到这一对多余的操作，也就是 autorelease 及紧跟其后的 retain。为了优化代码，在方法中返回自动释放的对象时，要执行一个特殊函数。此时不直接调用对象的 autorelease 方法，而是改为调用 objc_autoreleaseReturnValue。此函数会检视当前方法返回之后即将要执行的那段代码。若发现那段代码要在返回的对象上执行 retain 操作，则设置全局数据结构（此数据结构的具体内容因处理器而异）中的一个标志位，而不执行 autorelease 操作。与之相似，如果方法返回了一个自动释放的对象，而调用方法的代码要保留此对象，那么此时不直接执行 retain，而是改为执行 objc_retainAutoreleasedReturnValue 函数。此函数要检测刚才提到的那个标志位，若已经置位，则不执行 retain 操作。设置并检测标志位，要比调用 autorelease 和 retain 更快。

下面这段代码演示了 ARC 是如何通过这些特殊函数来优化程序的：

```
// Within EOCPerson class
+ (EOCPerson*)personWithName:(NSString*)name {
    EOCPerson *person = [[EOCPerson alloc] init];
    person.name = name;
    objc_autoreleaseReturnValue(person);
}

// Code using EOCPerson class
EOCPerson *tmp = [EOCPerson personWithName:@"Matt Galloway"];
_myPerson = objc_retainAutoreleasedReturnValue(tmp);
```

为了求得最佳效率，这些特殊函数的实现代码都因处理器而异。下面这段伪代码描述了其中的步骤：

```
id objc_autoreleaseReturnValue(id object) {
    if ( /* caller will retain object */ ) {
        set_flag(object);
        return object; ///< No autorelease
    } else {
        return [object autorelease];
    }
}

id objc_retainAutoreleasedReturnValue(id object) {
    if (get_flag(object)) {
        clear_flag(object);
        return object; ///< No retain
    } else {
        return [object retain];
    }
}
```

objc_autoreleaseReturnValue 函数究竟如何检测方法调用者是否会立刻保留对象呢？这要根据处理器来定。由于必须查看原始的机器码指令方可判断出这一点，所以，只有编译器的作者才能实现此函数。要想判断出方法调用者会不会保留方法所返回的对象，先得把调用方法的那段代码编排好才行，而这项任务只能由编译器的开发者来完成。

将内存管理交由编译器和运行期组件来做，可以使代码得到多种优化，上面所讲的只是其中一种。我们由此应该可以了解到 ARC 所带来的好处。待编译器与运行期组件日臻成熟，笔者相信还会出现其他优化技术。

变量的内存管理语义

ARC 也会处理局部变量与实例变量的内存管理。默认情况下，每个变量都是指向对象的强引用。一定要理解这个问题，尤其要注意实例变量的语义，因为对于某些代码来说，其语

义和手动管理引用计数时不同。例如，有下面这段代码：

```
@interface EOCClass : NSObject {
    id _object;
}

@implementation EOCClass
- (void)setup {
    _object = [EOCOtherClass new];
}
@end
```

在手动管理引用计数时，实例变量 _object 并不会自动保留其值，而在 ARC 环境下则会这样做。也就是说，若在 ARC 下编译 setup 方法，则其代码会变为：

```
- (void)setup {
    id tmp = [EOCOtherClass new];
    _object = [tmp retain];
    [tmp release];
}
```

当然，在此情况下，retain 和 release 可以消去。所以，ARC 会将这两个操作化简掉，于是，实际执行的代码还是和原来一样。不过，在编写设置方法（setter）时，使用 ARC 会简单一些。如果不用 ARC，那么需要像下面这样来写：

```
- (void)setObject:(id)object {
    [_object release];
    _object = [object retain];
}
```

但是这样写会出问题。假如新值和实例变量已有的值相同，会如何呢？如果只有当前对象还在引用这个值，那么设置方法中的释放操作会使该值的保留计数降为 0，从而导致系统将其回收。接下来再执行保留操作，就会令应用程序崩溃。使用 ARC 之后，就不可能发生这种疏失了。在 ARC 环境下，与刚才等效的设置函数可以这么写：

```
- (void)setObject:(id)object {
    _object = object;
}
```

ARC 会用一种安全的方式来设置：先保留新值，再释放旧值，最后设置实例变量。在手动管理引用计数时，你可能已经明白这个问题了，所以应该能正确编写设置方法，不过用了 ARC 之后，根本无须考虑这种“边界情况”（edge case）。

在应用程序中，可用下列修饰符来改变局部变量与实例变量的语义：

- __strong：默认语义，保留此值。
- __unsafe_unretained：不保留此值，这么做可能不安全，因为等到再次使用变量时，其对象可能已经回收了。

- __weak：不保留此值，但是变量可以安全使用，因为如果系统把这个对象回收了，那么变量也会自动清空。
- __autoreleasing：把对象“按引用传递”（pass by reference）给方法时，使用这个特殊的修饰符。此值在方法返回时自动释放。

比方说，想令实例变量的语义与不使用 ARC 时相同，可以运用 __weak 或 __unsafe_unretained 修饰符：

```
@interface EOCClass : NSObject {
    id __weak _weakObject;
    id __unsafe_unretained _unsafeUnretainedObject;
}
```

不论采用上面哪种写法，在设置实例变量时都不会保留其值。只有使用新版（Mac OS X 10.7、iOS 5.0 及其后续版本）运行期程序库时，加了 __weak 修饰符的 weak 引用才会自动清空，因为实现自动清空操作，要用到新版所添加的一些功能。

我们经常会给局部变量加上修饰符，用以打破由“块”（block，参见第 40 条）所引入的“保留环”（retain cycle）。块会自动保留其所捕获的全部对象，而如果这其中有某个对象又保留了块本身，那么就可能导致“保留环”。可以用 __weak 局部变量来打破这种“保留环”：

```
NSURL *url = [NSURL URLWithString:@"http://www.example.com/"];
EOCNetworkFetcher *fetcher =
    [[EOCNetworkFetcher alloc] initWithURL:url];
EOCNetworkFetcher * __weak weakFetcher = fetcher;
[fetcher startWithCompletion:^(BOOL success){
    NSLog(@"Finished fetching from %@", weakFetcher.url);
}];
```

ARC 如何清理实例变量

刚才说过，ARC 也负责对实例变量进行内存管理。要管理其内存，ARC 就必须在“回收分配给对象的内存”（deallocate）[⊖] 时生成必要的清理代码（cleanup code）。凡是具备强引用的变量，都必须释放，ARC 会在 dealloc 方法中插入这些代码。当手动管理引用计数时，你可能会像下面这样自己来编写 dealloc 方法：

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

用了 ARC 之后，就不需要再编写这种 dealloc 方法了，因为 ARC 会借用 Objective-C++

⊖ “deallocate”也称为“释放（内存）”，然而在 Objective-C 中，“释放”一词与“release”操作相对应，所以为了避免混淆，译文改用“回收”、“解除分配”等说法来对应“deallocate”。——译者注

的一项特性来生成清理例程（cleanup routine）。回收 Objective-C++ 对象时，待回收的对象会调用所有 C++ 对象的析构函数（destructor）。编译器如果发现某个对象里含有 C++ 对象，就会生成名为 .cxx_destruct 的方法。而 ARC 则借助此特性，在该方法中生成清理内存所需的代码。

不过，如果有非 Objective-C 的对象，比如 CoreFoundation 中的对象或是由 malloc() 分配在堆中的内存，那么仍然需要清理。然而不需要像原来那样调用超类的 dealloc 方法。前文说过，在 ARC 下不能直接调用 dealloc。ARC 会自动在 .cxx_destruct 方法中生成代码并运行此方法，而在生成的代码中会自动调用超类的 dealloc 方法。ARC 环境下，dealloc 方法可以像这样来写：

```
- (void)dealloc {
    CFRelease(_coreFoundationObject);
    free(_heapAllocatedMemoryBlob);
}
```

因为 ARC 会自动生成回收对象时所执行的代码，所以通常无须再编写 dealloc 方法。这能减少项目源代码的大小，而且可以省去其中一些样板代码（boilerplate code）。

覆写内存管理方法

不使用 ARC 时，可以覆写内存管理方法。比方说，在实现单例类的时候，因为单例不可释放，所以我们经常覆写 release 方法，将其替换为“空操作”（no-op）。但在 ARC 环境下不能这么做，因为会干扰到 ARC 分析对象生命期的工作。而且，由于开发者不可调用及覆写这些方法，所以 ARC 能够优化 retain、release、autorelease 操作，使之不经过 Objective-C 的消息派发机制（参见第 11 条）。优化后的操作，直接调用隐藏在运行期程序库中的 C 函数。这就意味着 ARC 可以执行各种优化了，比如刚才提到：如果方法命令即将返回的对象稍后“自动释放”，而方法调用者立刻“保留”这个返回后的对象，那么这两个操作就会为 ARC 所化简。

要点

- 有 ARC 之后，程序员就无须担心内存管理问题了。使用 ARC 来编程，可省去类中的许多“样板代码”。
- ARC 管理对象生命期的办法基本上就是：在合适的地方插入“保留”及“释放”操作。在 ARC 环境下，变量的内存管理语义可以通过修饰符指明，而原来则需要手工执行“保留”及“释放”操作。
- 由方法所返回的对象，其内存管理语义总是通过方法名来体现。ARC 将此确定为开发者必须遵守的规则。
- ARC 只负责管理 Objective-C 对象的内存。尤其要注意：CoreFoundation 对象不归 ARC 管理，开发者必须适时调用 CFRetain/CFRelease。

第31条：在dealloc方法中只释放引用并解除监听

对象在经历其生命期后，最终会为系统所回收，这时就要执行dealloc方法了。在每个对象的生命期内，此方法仅执行一次，也就是当保留计数降为0的时候。然而具体何时执行，则无法保证。也可以理解成：我们能够通过人工观察保留操作与释放操作的位置，来预估此方法何时即将执行。但实际上，程序库会以开发者察觉不到的方式操作对象，从而使回收对象的真正时机和预期的不同。你决不应该自己调用dealloc方法。运行期系统会在适当的时候调用它。而且，一旦调用过dealloc之后，对象就不再有效了，后续方法调用均是无效的。

那么，应该在dealloc方法中做些什么呢？主要就是释放对象所拥有的引用，也就是把所有Objective-C对象都释放掉，ARC会通过自动生成的.cxx_destruct方法（参见第30条），在dealloc中为你自动添加这些释放代码。对象所拥有的其他非Objective-C对象也要释放。比如CoreFoundation对象就必须手工释放，因为它们是由纯C的API所生成的。

在dealloc方法中，通常还要做一件事，那就是把原来配置过的观测行为（observation behavior）都清理掉。如果用NSNotificationCenter给此对象订阅（register）过某种通知，那么一般应该在这里注销（unregister），这样的话，通知系统就不再把通知发给回收后的对象了，若是还向其发送通知，则必然会令应用程序崩溃。

dealloc方法可以这样来写：

```
- (void)dealloc {
    CFRelease(coreFoundationObject);
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```

请注意，如果手动管理引用计数而不使用ARC的话，那么最后还需调用“[super dealloc]”。ARC会自动执行此操作，这再次表明其比手动管理更简单、更安全。若选择手动管理，则还要将当前对象所拥有的全部Objective-C对象逐个释放。

虽说应该于dealloc中释放引用，但是开销较大或系统内稀缺的资源则不在此列。像是文件描述符（file descriptor）、套接字（socket）、大块内存等，都属于这种资源。不能指望dealloc方法必定会在某个特定的时机调用，因为有一些无法预料的东西可能也持有此对象。在这种情况下，如果非要等到系统调用dealloc方法时才释放，那么保留这些稀缺资源的时间就有些过长了，这么做不合适。通常的做法是，实现另外一个方法，当应用程序用完资源对象后，就调用此方法。这样一来，资源对象的生命期就变得更为明确了。

比方说，如果某对象管理着连接服务器所用的套接字，那么也许就需要这种“清理方法”（cleanup method）。此对象可能要通过套接字连接到数据库。对于对象所属的类，其接口可以这样写：

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

@interface EOCServerConnection : NSObject
```

```
- (void)open:(NSString*)address;
- (void)close;
@end
```

该类与开发者之间的约定是：想打开连接，就调用“open:”方法；连接使用完毕，就调用 close 方法。“关闭”操作必须在系统把连接对象回收之前调用，否则就是编程错误（programmer error），这与通过“保留”及“释放”操作来平衡引用计数是类似的。

在清理方法而非 dealloc 方法中清理资源还有个原因，就是系统并不保证每个创建出来的对象的 dealloc 都会执行。极个别情况下，当应用程序终止时，仍有对象处于存活状态，这些对象没有收到 dealloc 消息。由于应用程序终止之后，其占用的资源也会返还给操作系统，所以实际上这些对象也就等于是消亡了。不调用 dealloc 方法是为了优化程序效率。而这也说明系统未必会在每个对象上调用其 dealloc 方法。在 Mac OS X 及 iOS 应用程序所对应的 application delegate 中，都含有一个会于程序终止时调用的方法。如果一定要清理某些对象，那么可在此方法中调用那些对象的“清理方法”。

在 Mac OS X 系统里，应用程序终止时会调用 NSApplicationDelegate 之中的下述方法：

```
- (void)applicationWillTerminate:(NSNotification *)notification
```

而在 iOS 系统里，应用程序终止时则会调用 UIApplicationDelegate 之中的下述方法：

```
- (void)applicationWillTerminate:(UIApplication *)application
```

如果对象管理着某些资源，那么在 dealloc 中也要调用“清理方法”，以防开发者忘了清理这些资源。忘记清理资源的情况经常会发生，所以最好能输出一行消息，提示程序员代码里含有编程错误。在系统回收对象之前，必须调用 close 以释放其资源，否则 close 方法就失去意义了，因此，没有适时调用 close 方法就是编程错误。输出错误消息可促使开发者纠正此问题。而且，在程序员忘记调用 close 的情况下，我们应该在 dealloc 中补上这次调用，以防泄漏内存。下面举例说明 close 与 dealloc 方法应如何来写：

```
- (void)close {
    /* clean up resources */
    _closed = YES;
}

- (void)dealloc {
    if (!_closed) {
        NSLog(@"ERROR: close was not called before dealloc!");
        [self close];
    }
}
```

有时可能不想只输出一条错误消息，而是要抛出异常来表明不调用 close 方法是严重的编程错误。

编写 dealloc 方法时还需注意，不要在里面随便调用其他方法。刚才那段范例代码中，

dealloc 方法确实调用了另外一个方法，不过那是为了侦测编程错误而破例。无论在这里调用什么方法都不太应该，因为对象此时“已近尾声”（in a winding-down state）。如果在这里所调用的方法又要异步执行某些任务，或是又要继续调用它们自己的某些方法，那么等到那些任务执行完毕时，系统已经把当前这个待回收的对象彻底摧毁了。这会导致很多问题，且经常使应用程序崩溃，因为那些任务执行完毕后，要回调此对象，告诉该对象任务已完成，而此时如果对象已摧毁，那么回调操作就会出错。

请再注意一个问题：调用 dealloc 方法的那个线程会执行“最终的释放操作”（final release），令对象的保留计数降为 0，而某些方法必须在特定的线程里（比如主线程里）调用才行。若在 dealloc 里调用了那些方法，则无法保证当前这个线程就是那些方法所需的线程。通过编写常规代码的方式，无论如何都没办法保证其会安全运行在正确的线程上，因为对象处于“正在回收的状态”（deallocating state），为了指明此状况，运行期系统已经改动了对象内部的数据结构。

在 dealloc 里也不要调用属性的存取方法，因为有人可能会覆写这些方法，并于其中做一些无法在回收阶段安全执行的操作。此外，属性可能正处于“键值观测”（Key-Value Observation，KVO）机制的监控之下，该属性的观察者（observer）可能会在属性值改变时“保留”或使用这个即将回收的对象。这种做法会令运行期系统的状态完全失调，从而导致一些莫名其妙的错误。

要点

- 在 dealloc 方法里，应该做的事情就是释放指向其他对象的引用，并取消原来订阅的“键值观测”（KVO）或 NSNotificationCenter 等通知，不要做其他事情。
- 如果对象持有文件描述符等系统资源，那么应该专门编写一个方法来释放此种资源。这样的类要和其使用者约定：用完资源后必须调用 close 方法。
- 执行异步任务的方法不应在 dealloc 里调用；只能在正常状态下执行的那些方法也不应在 dealloc 里调用，因为此时对象已处于正在回收的状态了。

第 32 条：编写“异常安全代码”时留意内存管理问题

许多时下流行的编程语言都提供了“异常”（exception）这一特性。纯 C 中没有异常，而 C++ 与 Objective-C 都支持异常。实际上，在当前的运行期系统中，C++ 与 Objective-C 的异常相互兼容，也就是说，从其中一门语言里抛出的异常能用另外一门语言所编的“异常处理程序”（exception handler）来捕获。

Objective-C 的错误模型表明，异常只应在发生严重错误后抛出（参见第 21 条），虽说如此，不过有时仍然需要编写代码来捕获并处理异常。比如使用 Objective-C++ 来编码时，或是编码中用到了第三方程序库而此程序库所抛出的异常又不受你控制时，就需要捕获及处理异常了。此外，有些系统库也会用到异常，这使我们想起从前那个频繁使用异常的年

代。比如，在使用“键值观测”（KVO）功能时，若想注销一个尚未注册的“观察者”，便会抛出异常。

发生异常时应该如何管理内存是个值得研究的问题。在 try 块中，如果先保留了某个对象，然后在释放它之前又抛出了异常，那么，除非 catch 块能处理此问题，否则对象所占内存就将泄漏。C++ 的析构函数（destructor）由 Objective-C 的异常处理例程（exception-handle routine）来运行。这对于 C++ 对象很重要，由于抛出异常会缩短其生命期，所以发生异常时必须析构，不然就会泄漏，而文件句柄（file handle）等系统资源因为没有正确清理，所以就更容易因此而泄漏了。

异常处理例程将自动销毁对象，然而在手动管理引用计数时，销毁工作有些麻烦。以下面这段使用手工引用计数的 Objective-C 代码为例：

```
@try {
    EOCSomeClass *object = [[EOCSomeClass alloc] init];
    [object doSomethingThatMayThrow];
    [object release];
}
@catch (...) {
    NSLog(@"Whoops, there was an error. Oh well...");
}
```

乍一看似乎没问题，但如果 doSomethingThatMayThrow 抛出异常了呢？由于异常会令执行过程终止并跳至 catch 块，因而其后的那行 release 代码不会运行。在这种情况下，如果代码抛出异常，那么对象就泄漏了。这么做不好。解决办法是使用 @finally 块，无论是否抛出异常，其中的代码都保证会运行，且只运行一次。比方说，刚才那段代码可改写如下：

```
EOCSomeClass *object;
@try {
    object = [[EOCSomeClass alloc] init];
    [object doSomethingThatMayThrow];
}
@catch (...) {
    NSLog(@"Whoops, there was an error. Oh well...");
}
@finally {
    [object release];
}
```

注意，由于 @finally 块也要引用 object 对象，所以必须把它从 @try 块里移到外面去。要是所有对象都得如此释放，那这样做就会非常乏味。而且，假如 @try 块中的逻辑更为复杂，含有多条语句，那么很容易就会因为忘记某个对象而导致泄漏。若泄漏的对象是文件描述符或数据库连接等稀缺资源（或是这些稀缺资源的管理者），则可能引发大问题，因为这将导致应用程序把所有系统资源都抓在自己手里而不及时释放。

在 ARC 环境下，问题会更严重。下面这段使用 ARC 的代码与修改前的那段代码等效：

```
@try {
    EOCSomeClass *object = [[EOCSomeClass alloc] init];
    [object doSomethingThatMayThrow];
}
@catch (...) {
    NSLog(@"Whoops, there was an error. Oh well...");
}
```

现在问题更大了：由于不能调用 release，所以无法像手动管理引用计数时那样把释放操作移到 @finally 块中。你可能认为这种状况 ARC 自然会处理的。但实际上 ARC 不会自动处理，因为这样做需要加入大量样板代码，以便跟踪待清理的对象，从而在抛出异常时将其释放。可是，这段代码会严重影响运行期的性能，即便在不抛异常时也如此。而且，添加进来的额外代码还会明显增加应用程序的大小。这些副作用都不甚理想。

虽说默认状况下未开启，但 ARC 依然能生成这种安全处理异常所用的附加代码。-fobjc-arc-exceptions 这个编译器标志用来开启此功能。其默认不开启的原因是：在 Objective-C 代码中，只有当应用程序必须因异常状况而终止时才应抛出异常（参见第 21 条）。因此，如果应用程序即将终止，那么是否还会发生内存泄漏就已经无关紧要了。在应用程序必须立即终止的情况下，还去添加安全处理异常所用的附加代码是没有意义的。

有种情况编译器会自动把 -fobjc-arc-exceptions 标志打开，就是处于 Objective-C++ 模式时。因为 C++ 处理异常所用的代码与 ARC 实现的附加代码类似，所以令 ARC 加入自己的代码以安全处理异常，其性能损失并不太大。此外，由于 C++ 频繁使用异常，所以 Objective-C++ 程序员很可能也会使用异常。

如果手工管理引用计数，而且必须捕获异常，那么要设法保证所编代码能把对象正确清理干净。若使用 ARC 且必须捕获异常，则需打开编译器的 -fobjc-arc-exceptions 标志。但最重要的是：在发现大量异常捕获操作时，应考虑重构代码，用第 21 条所讲的 NSError 式错误信息传递法来取代异常。

要点

- 捕获异常时，一定要注意将 try 块内所创立的对象清理干净。
- 在默认情况下，ARC 不生成安全处理异常所需的清理代码。开启编译器标志后，可生成这种代码，不过会导致应用程序变大，而且会降低运行效率。

第 33 条：以弱引用避免保留环

对象图里经常会出现一种情况，就是几个对象都以某种方式互相引用，从而形成“环”（cycle）。由于 Objective-C 内存管理模型使用引用计数架构，所以这种情况通常会泄漏内存，因为最后没有别的东西会引用环中的对象。这样的话，环里的对象就无法为外界所访问了，但对象之间尚有引用，这些引用使得它们都能继续存活下去，而不会为系统所回收。

最简单的保留环由两个对象构成，它们互相引用对方。图 5-4 举例说明了这种情况。

这种保留环的产生原因不难理解，且很容易就能通过查看代码而侦测出来：

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

@class EOCClassA;
@class EOCClassB;

@interface EOCClassA : NSObject
@property (nonatomic, strong) EOCClassB *other;
@end

@interface EOCClassB : NSObject
@property (nonatomic, strong) EOCClassA *other;
@end
```

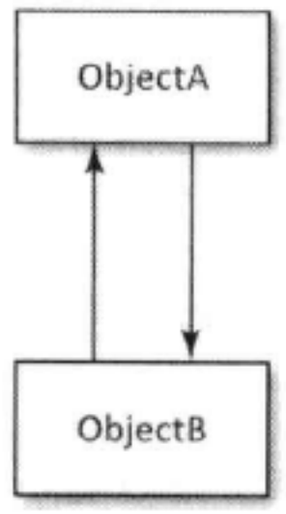

图 5-4 两个对象通过彼此之间的强引用而构成保留环

看代码很容易就能发现其中可能出现的保留环：如果把 EOCClassA 实例的 other 属性设置成某个 EOCClassB 实例，而把那个 EOCClassB 实例的 other 属性又设置成这个 EOCClassA 实例，那么就会出现图 5-5 中的保留环。

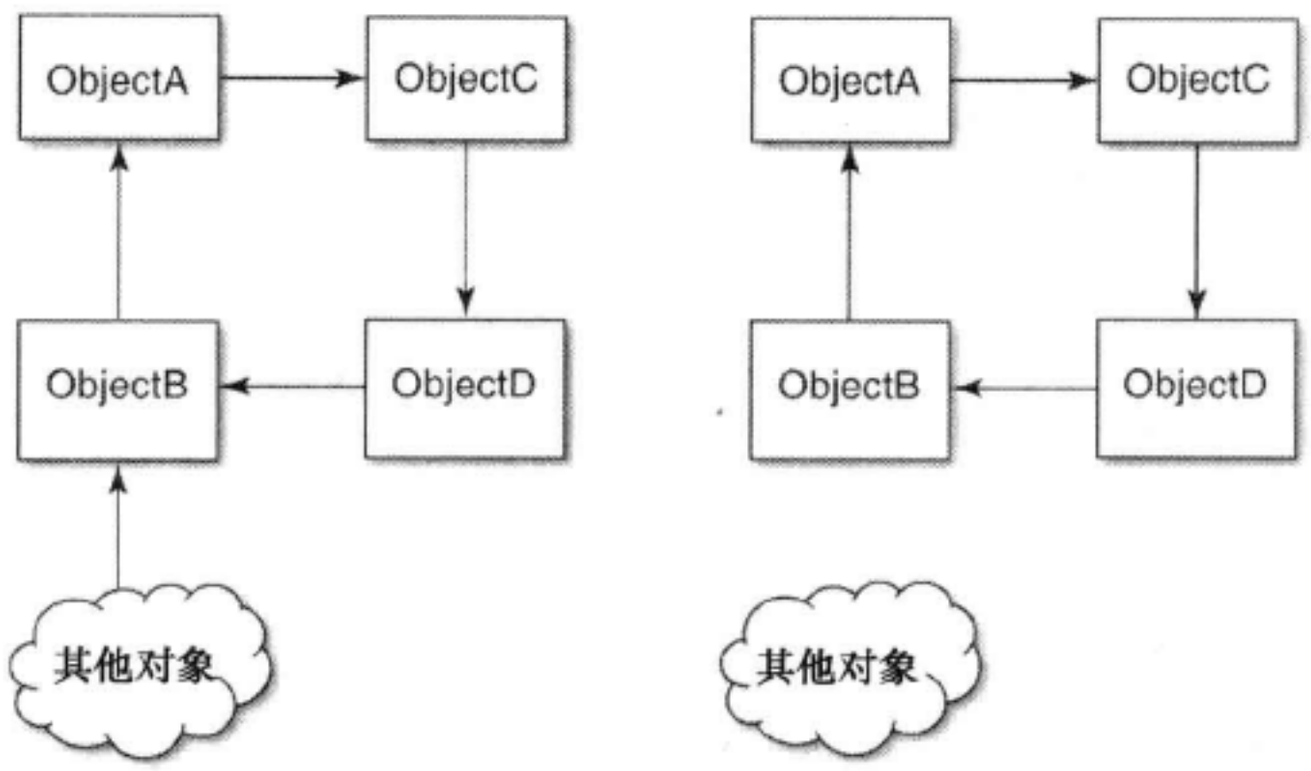

图 5-5 保留环中只剩一个对象还为对象图里的其他对象所引用，移除此引用后整个保留环就泄漏了

保留环会导致内存泄漏。如果只剩一个引用还指向保留环中的实例，而现在又把这个引用移除，那么整个保留环就泄漏了。也就是说，没办法再访问其中的对象了。图 5-5 所示的保留环更复杂一些，其中有四个对象，只有 ObjectB 还为外界所引用，把仅有的这个引用移除之后，四者所占内存就泄漏了。

Mac OS X 平台的 Objective-C 程序有个选项，可以启用垃圾收集器（garbage collector），它会检测保留环，若发现外界不再引用其中的对象，则将之回收。但是，从 Mac OS X 10.8 开始，垃圾收集机制就废弃了，而且 iOS 系统从未支持过这项功能。因此，从一开始编码时就要注意别出现保留环。

避免保留环的最佳方式就是弱引用。这种引用经常用来表示“非拥有关系”（nonowning relationship）。将属性声明为 unsafe_unretained 即可。修改刚才那段范例代码，将其属性声明如下：

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

@class EOCClassA;
@class EOCClassB;

@interface EOCClassA : NSObject
@property (nonatomic, strong) EOCClassB *other;
@end

@interface EOCClassB : NSObject
@property (nonatomic, unsafe_unretained) EOCClassA *other;
@end
```

修改之后，EOCClassB 实例就不再通过 other 属性来拥有 EOCClassA 实例了。属性特质（attribute）中的 unsafe_unretained 一词表明，属性值可能不安全，而且不归此实例所拥有。如果系统已经把属性所指的那个对象回收了，那么在其上调用方法可能会使应用程序崩溃。由于本对象并不保留属性对象，因此其有可能为系统所回收。

用 unsafe_unretained 修饰的属性特质，其语义同 assign 特质等价（参见第 6 条）。然而，assign 通常只用于“整体类型”（int、float、结构体等），unsafe_unretained 则多用于对象类型。这个词本身就表明其所修饰的属性可能无法安全使用（unsafe）。

Objective-C 中还有一项与 ARC 相伴的运行期特性，可以令开发者安全使用弱引用：这就是 weak 属性特质，它与 unsafe_unretained 的作用完全相同。然而，只要系统把属性回收，属性值就会自动设为 nil。在刚才那段代码中，EOCClassB 的 other 属性可修改如下：

```
@property (nonatomic, weak) EOCClassA *other;
```

图 5-6 演示了 unsafe_unretained 与 weak 属性的区别。

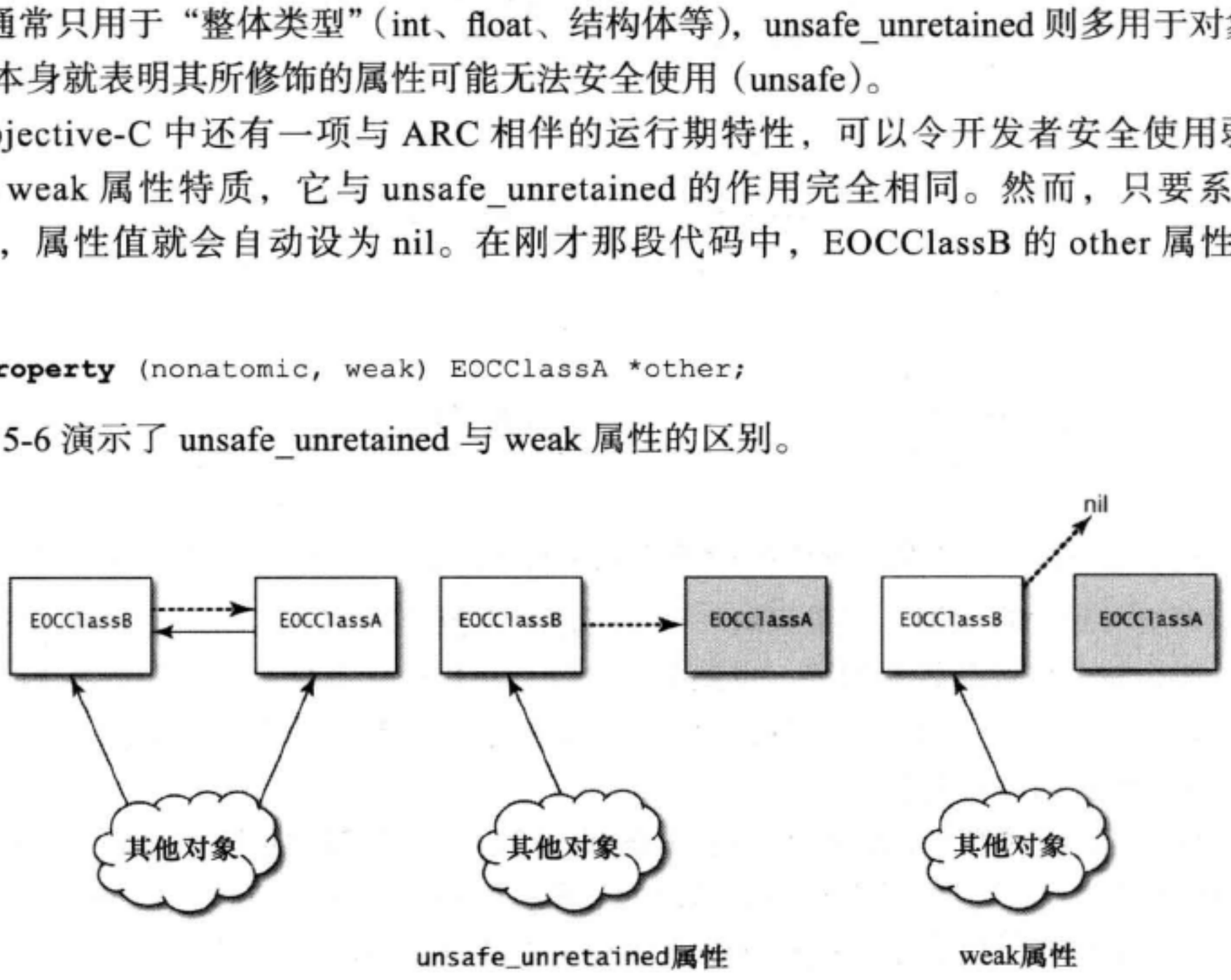

图 5-6 unsafe_unretained 与 weak 属性，在其所指的对象回收以后表现出来的行为不同

当指向 EOCClassA 实例的引用移除后，unsafe_unretained 属性仍然指向那个已经回收的实例，而 weak 属性则指向 nil。

但是，使用 weak 属性并不是偷懒的借口。在刚才那个例子中，如果在 EOCClassA 对象已经回收之后，引用它的 EOCClassB 实例仍然存活，那么就是编程错误。发生这种情况，就是 bug。开发者应确保程序中不出现此类问题。然而，使用 weak 而非 unsafe_unretained 引用可以令代码更安全。应用程序也许会显示出错误的数据，但不会直接崩溃。这么做显然比令终端用户直接看到程序退出要好。不过无论如何，只要在所指对象已经彻底销毁后还继续使用弱引用，那就依然是个 bug。比方说，用户界面中的某个元素会把数据源设置给某个属性，并通过它来查询将要显示的数据。这种属性通常是弱引用（参见第 23 条）。假如还未等界面元素查询完数据源对象就已经回收，那么，继续使用弱引用虽不致程序崩溃，但却无法再查到数据了。

一般来说，如果不拥有某对象，那就不要保留它。这条规则对 collection 例外，collection 虽然并不直接拥有其内容，但是它要代表自己所属的那个对象来保留这些元素。有时，对象中的引用会指向另外一个并不归自己所拥有的对象，比如 Delegate 模式就是这样（参见第 23 条）。

要点

- 将某些引用设为 weak，可避免出现“保留环”。
- weak 引用可以自动清空，也可以不自动清空。自动清空（autonilling）是随着 ARC 而引入的新特性，由运行期系统来实现。在具备自动清空功能的弱引用上，可以随意读取其数据，因为这种引用不会指向已经回收过的对象。

第 34 条：以“自动释放池块”降低内存峰值

Objective-C 对象的生命期取决于其引用计数（参见第 29 条）。在 Objective-C 的引用计数架构中，有一项特性叫做“自动释放池”（autorelease pool）。释放对象有两种方式：一种是调用 release 方法，使其保留计数立即递减；另一种是调用 autorelease 方法，将其加入“自动释放池”中。自动释放池用于存放那些需要在稍后某个时刻释放的对象。清空（drain）自动释放池时，系统会向其中的对象发送 release 消息。

创建自动释放池所用语法如下：

```
@autoreleasepool {
    // ...
}
```

如果在没有创建自动释放池的情况下给对象发送 autorelease 消息，那么控制台会输出这样一条信息：

```
Object 0xabcd0123 of class __NSCFString autoreleased
with no pool in place - just leaking - break on objc_
```

```
autoreleaseNoPool() to debug
```

然而，一般情况下无须担心自动释放池的创建问题。Mac OS X 与 iOS 应用程序分别运行于 Cocoa 及 Cocoa Touch 环境中。系统会自动创建一些线程，比如说主线程或是“大中枢派发”(Grand Central Dispatch，GCD)[⊖]机制中的线程，这些线程默认都有自动释放池，每次执行“事件循环”(event loop)时，就会将其清空。因此，不需要自己来创建“自动释放池块”。通常只有一个地方需要创建自动释放池，那就是在 main 函数里，我们用自动释放池来包裹应用程序的主入口点（main application entry point)。比方说，iOS 程序的 main 函数经常这样写：

```
int main(int argc, char *argv[]) {
    @autoreleasepool {
        return UIApplicationMain(argc,
                                 argv,
                                 nil,
                                 @"EOCAppDelegate");
    }
}
```

从技术角度看，不是非得有个“自动释放池块”才行。因为块的末尾恰好就是应用程序的终止处，而此时操作系统会把程序所占的全部内存都释放掉。虽说如此，但是如果不写这个块的话，那么由 UIApplicationMain 函数所自动释放的那些对象，就没有自动释放池可以容纳了，于是系统会发出警告信息来表明这一情况。所以说，这个池可以理解成最外围捕捉全部自动释放对象所用的池。

下面这段代码中的花括号定义了自动释放池的范围。自动释放池于左花括号处创建，并于对应的右花括号处自动清空。位于自动释放池范围内的对象，将在此范围末尾处收到 release 消息。自动释放池可以嵌套。系统在自动释放对象时，会把它放到最内层的池里。比方说：

```
@autoreleasepool {
    NSString *string = [NSString stringWithFormat:@"1 = %i", 1];
    @autoreleasepool {
        NSNumber *number = [NSNumber numberWithInt:1];
    }
}
```

本例中有两个对象，它们都由类的工厂方法所创建，这样创建出来的对象会自动释放(参见第 30 条)。NSString 对象放在外围的自动释放池中，而 NSNumber 对象则放在里层的自动释放池中。将自动释放池嵌套用的好处是，可以借此控制应用程序的内存峰值，使其不致过高。

考虑下面这段代码：

⊖ 也称为“大中央调度”。——译者注

```
for (int i = 0; i <100000; i++) {
    [self doSomethingWithInt:i];
}
```

如果“doSomethingWithInt:”方法要创建临时对象，那么这些对象很可能会放在自动释放池里。比方说，它们可能是一些临时字符串。但是，即便这些对象在调用完方法之后就不再使用了，它们也依然处于存活状态，因为目前还在自动释放池里，等待系统稍后将其释放并回收。然而，自动释放池要等线程执行下一次事件循环时才会清空。这就意味着在执行 for 循环时，会持续有新对象创建出来，并加入自动释放池中。所有这种对象都要等 for 循环执行完才会释放。这样一来，在执行 for 循环时，应用程序所占内存量就会持续上涨，而等到所有临时对象都释放后，内存用量又会突然下降。

这种情况不甚理想，尤其当循环长度无法预知，必须取决于用户输入时更是如此。比方说，要从数据库中读出许多对象。代码可能会这么写：

```
NSArray *databaseRecords = /* ... */;
NSMutableArray *people = [NSMutableArray new];
for (NSDictionary *record in databaseRecords) {
    EOCPerson *person = [[EOCPerson alloc]
                                initWithRecord:record];
    [people addObject:person];
}
```

EOCPerson 的初始化函数也许会像上例那样，再创建出一些临时对象。若记录有很多条，则内存中也会有很多不必要的临时对象，它们本来应该提早回收的。增加一个自动释放池即可解决此问题。如果把循环内的代码包裹在“自动释放池块”中，那么在循环中自动释放的对象就会放在这个池，而不是线程的主池里面。例如：

```
NSArray *databaseRecords = /* ... */;
NSMutableArray *people = [NSMutableArray new];
for (NSDictionary *record in databaseRecords) {
    @autoreleasepool {
        EOCPerson *person =
            [[EOCPerson alloc] initWithRecord:record];
        [people addObject:person];
    }
}
```

加上这个自动释放池之后，应用程序在执行循环时的内存峰值就会降低，不再像原来那么高了。内存峰值（high-memory waterline）是指应用程序在某个特定时段内的最大内存用量（highest memory footprint）。新增的自动释放池块可以减少这个峰值，因为系统会在块的末尾把某些对象回收掉。而刚才提到的那种临时对象，就在回收之列。

自动释放池机制就像“栈”（stack）一样。系统创建好自动释放池之后，就将其推入栈中，而清空自动释放池，则相当于将其从栈中弹出。在对象上执行自动释放操作，就等于将其放入栈顶的那个池里。

是否应该用池来优化效率，完全取决于具体的应用程序。首先得监控内存用量，判断其中有没有需要解决的问题，如果没完成这一步，那就别急着优化。尽管自动释放池块的开销不太大，但毕竟还是有的，所以尽量不要建立额外的自动释放池。

如果在 ARC 出现之前就写过 Objective-C 程序，那么可能还记得有种老式写法，就是使用 NSAutoreleasePool 对象。这个特殊的对象与普通对象不同，它专门用来表示自动释放池，就像新语法中的自动释放池块一样。但是这种写法并不会在每次执行 for 循环时都清空池，此对象更为“重量级”(heavyweight)，通常用来创建那种偶尔需要清空的池，比方说：

```
NSArray *databaseRecords = /* ... */;
NSMutableArray *people = [NSMutableArray new];
int i = 0;

NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
for (NSDictionary *record in databaseRecords) {
    EOCPerson *person = [[EOCPerson alloc]
                                initWithRecord:record];
    [people addObject:person];

    // Drain the pool only every 10 cycles
    if (++i == 10) {
        [pool drain];
        i = 0;
    }
}

// Also drain at the end in case the loop is not a multiple of 10
[pool drain];
```

现在不需要再这样写代码了。采用随着 ARC 所引入的新语法，可以创建出更为“轻量级”（lightweight）的自动释放池。原来所写的代码可能会每执行 *n* 次循环清空一次自动释放池，现在可以改用自动释放池块把 for 循环中的语句包起来，这样的话，每次执行循环时都会建立并清空自动释放池。

@autoreleasepool 语法还有个好处：每个自动释放池均有其范围，可以避免无意间误用了那些在清空池后已为系统所回收的对象。比方说，考虑下面这段采用旧式写法的代码：

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
id object = [self createObject];
[pool drain];
[self useObject:object];
```

这样写虽然稍显夸张，但却能说明问题。调用“ useObject:”方法时所传入的那个对象，可能已经为系统所回收了。同样的代码改用新式写法就变成了：

```
@autoreleasepool {
    id object = [self createObject];
}
```

```
[self useObject:object];
```

这次根本就无法编译，因为 object 变量出了自动释放池块的外围后就不可用了，所以在调用“useObject:”方法时不能用它做参数。

要点

- 自动释放池排布在栈中，对象收到 autorelease 消息后，系统将其放入最顶端的池里。
- 合理运用自动释放池，可降低应用程序的内存峰值。
- @autoreleasepool 这种新式写法能创建出更为轻便的自动释放池。

第 35 条：用“僵尸对象”调试内存管理问题

调试内存管理问题很令人头疼。大家都知道，向业已回收的对象发送消息是不安全的。这么做有时可以，有时不行。具体可行与否，完全取决于对象所占内存有没有为其他内容所覆写。而这块内存有没有移作他用，又无法确定，因此，应用程序只是偶尔崩溃。在没有崩溃的情况下，那块内存可能只复用了其中一部分，所以对象中的某些二进制数据依然有效。还有一种可能，就是那块内存恰好为另外一个有效且存活的对象所占据。在这种情况下，运行期系统会把消息发到新对象那里，而此对象也许能应答，也许不能。如果能，那程序就不崩溃，可你会觉得奇怪：为什么收到消息的对象不是预想的那个呢？若新对象无法响应选择子，则程序依然会崩溃。

所幸 Cocoa 提供了“僵尸对象”（Zombie Object）这个非常方便的功能。启用这项调试功能之后，运行期系统会把所有已经回收的实例转化成特殊的“僵尸对象”，而不会真正回收它们。这种对象所在的核心内存无法重用，因此不可能遭到覆写。僵尸对象收到消息后，会抛出异常，其中准确说明了发送过来的消息，并描述了回收之前的那个对象。僵尸对象是调试内存管理问题的最佳方式。

将 NSZombieEnabled 环境变量设为 YES，即可开启此功能。比方说，在 Mac OS X 系统中用 bash 运行应用程序时，可以这么做：

```
export NSZombieEnabled="YES"
./app
```

给僵尸对象发消息后，控制台会打印消息，而应用程序则会终止。打印出来的消息就像这样：

```
*** -[CFString respondsToSelector:]: message sent to
deallocated instance 0x7ff9e9c080e0
```

也可以在 Xcode 里打开此选项，这样的话，Xcode 在运行应用程序时会自动设置环境变量。开启方法为：编辑应用程序的 Scheme，在对话框左侧选择“Run”，然后切换至“Diagnostics”分页，最后勾选“Enable Zombie Objects”选项。图 5-7 演示了 Xcode 的配置

对话框，以及启用僵尸对象所需勾选的选项。

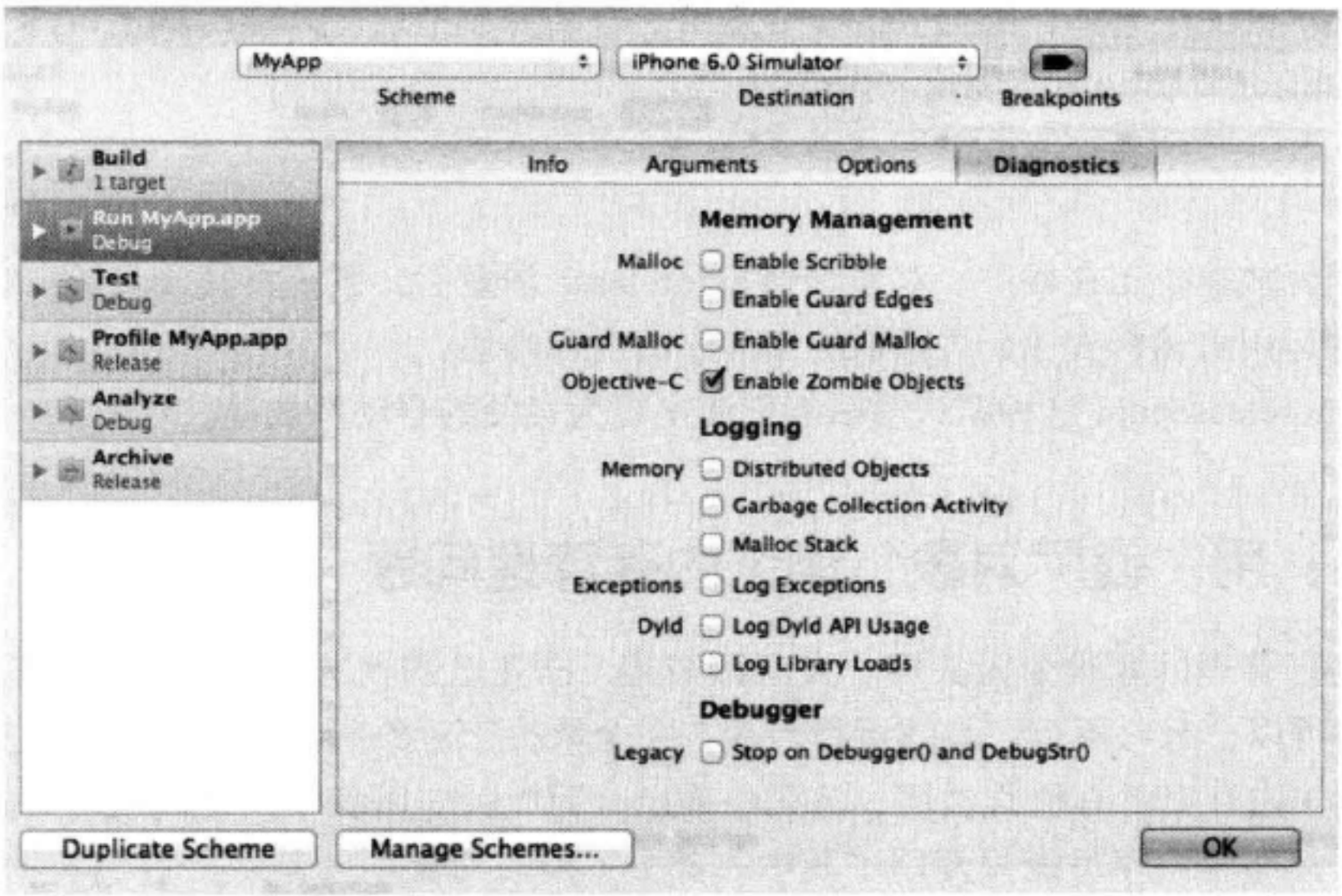

图 5-7 在 Xcode 的 Scheme 编辑器中启用僵尸对象

那么，僵尸对象的工作原理是什么呢？它的实现代码深植于 Objective-C 的运行期程序库、Foundation 框架及 CoreFoundation 框架中。系统在即将回收对象时，如果发现通过环境变量启用了僵尸对象功能，那么还将执行一个附加步骤。这一步就是把对象转化为僵尸对象，而不彻底回收。

下列代码有助于理解这一步所执行的操作：

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

@interface EOCClass : NSObject
@end

@implementation EOCClass
@end

void PrintClassInfo(id obj) {
    Class cls = object_getClass(obj);
    Class superCls = class_getSuperclass(cls);
    NSLog(@"=== %s : %s ===",
          class_getName(cls), class_getName(superCls));
}

int main(int argc, char *argv[]) {
    EOCClass *obj = [[EOCClass alloc] init];
    NSLog(@"Before release:");
```

```
    PrintClassInfo(obj);

    [obj release];
    NSLog(@"After release:");
    PrintClassInfo(obj);
}
```

为了便于演示普通对象转化为僵尸对象的过程，这段代码采用了手动引用计数。因为假如使用 ARC 的话，str 对象就会根据代码需要，尽可能多存活一段时间，于是在这个简单的例子中，就不可能变成僵尸对象了。这并不是说对象在 ARC 下绝对不可能转化为僵尸对象。即便用了 ARC，也依然会出现这种内存 bug，只不过一般要通过稍微复杂些的代码才能表现出来。

范例代码中有个函数，可以根据给定的对象打印出所属的类及其超类名称。此函数没有直接给对象发送 Objective-C 的 class 消息，而是调用了运行期库里的 object_getClass() 函数。因为如果参数已经是僵尸对象了，那么给其发送 Objective-C 消息后，控制台会打印错误消息，而且应用程序会崩溃。范例代码将输出下面这种消息：

```
Before release:
=== EOCClass : NSObject ===
After release:
=== _NSZombie_EOCClass : nil ===
```

对象所属的类已由 EOCClass 变为 _NSZombie_EOCClass。但是，这个新类是从哪里来的呢？代码中没有定义过这样一个类。而且，在启用僵尸对象后，如果编译器每看到一种可能变成僵尸的对象，就创建一个与之对应的类，那也太低效了。_NSZombie_EOCClass 实际上是在运行期生成的，当首次碰到 EOCClass 类的对象要变成僵尸对象时，就会创建这么一个类。创建过程中用到了运行期程序库里的函数，它们的功能很强大，可以操作类列表（class list）。

僵尸类（zombie class）是从名为 _NSZombie_ 的模板类里复制出来的。这些僵尸类没有多少事情可做，只是充当一个标记。接下来介绍它们是怎样充当标记的。首先来看下面这段伪代码，其中演示了系统如何根据需要创建出僵尸类，而僵尸类又如何把待回收的对象转化成僵尸对象。

```
// Obtain the class of the object being deallocated
Class cls = object_getClass(self);

// Get the class's name
const char *clsName = class_getName(cls);

// Prepend _NSZombie_ to the class name
const char *zombieClsName = "_NSZombie_" + clsName;

// See if the specific zombie class exists
Class zombieCls = objc_lookUpClass(zombieClsName);

// If the specific zombie class doesn't exist,
```

```
// then it needs to be created
if (!zombieCls) {
    // Obtain the template zombie class called _NSZombie_
    Class baseZombieCls = objc_lookUpClass("_NSZombie_");

    // Duplicate the base zombie class, where the new class's
    // name is the prepended string from above
    zombieCls = objc_duplicateClass(baseZombieCls,
                                    zombieClsName, 0);
}

// Perform normal destruction of the object being deallocated
objc_destructInstance(self);

// Set the class of the object being deallocated
// to the zombie class
objc_setClass(self, zombieCls);

// The class of 'self' is now _NSZombie_OriginalClass
```

这个过程其实就是NSObject的dealloc方法所做的事。运行期系统如果发现NSZombieEnabled环境变量已设置，那么就把dealloc方法“调配”（swizzle，参见第13条）成一个会执行上述代码的版本。执行到程序末尾时，对象所属的类已经变为_NSZombie_OriginalClass了，其中OriginalClass指的是原类名。

代码中的关键之处在于：对象所占内存没有（通过调用free()方法）释放，因此，这块内存不可复用。虽说内存泄漏了，但这只是个调试手段，制作正式发行的应用程序时不会把这项功能打开，所以这种泄漏问题无关紧要。

但是，系统为何要给每个变为僵尸的类都创建一个对应的新类呢？这是因为，给僵尸对象发消息后，系统可由此知道该对象原来所属的类。假如把所有僵尸对象都归到_NSZombie_类里，那原来的类名就丢了。创建新类的工作由运行期函数objc_duplicateClass()来完成，它会把整个_NSZombie_类结构拷贝一份，并赋予其新的名字。副本类的超类、实例变量及方法都和复制前相同。还有种做法也能保留旧类名，那就是不拷贝_NSZombie_，而是创建继承自_NSZombie_的新类，但是用相应的函数完成此功能，其效率不如直接拷贝高。

僵尸类的作用会在消息转发例程（参见第12条）中体现出来。_NSZombie_类（以及所有从该类拷贝出来的类）并未实现任何方法。此类没有超类，因此和NSObject一样，也是个“根类”，该类只有一个实例变量，叫做isa，所有Objective-C的根类都必须有此变量。由于这个轻量级的类没有实现任何方法，所以发给它的全部消息都要经过“完整的消息转发机制”（full forwarding mechanism，参见第12条）。

在完整的消息转发机制中，___forwarding___是核心，调试程序时，大家可能在栈回溯消息里看见过这个函数。它首先要做的事情就包括检查接收消息的对象所属的类名。若名称前缀为_NSZombie_，则表明消息接收者是僵尸对象，需要特殊处理。此时会打印一条消息

（本条目开头曾列出），其中指明了僵尸对象所收到的消息及原来所属的类，然后应用程序就终止了。在僵尸类名中嵌入原始类名的好处，这时就可以看出来了。只要把 _NSZombie_ 从僵尸类名的开头拿掉，剩下的就是原始类名。下列伪代码演示了这一过程：

```
// Obtain the object's class
Class cls = object_getClass(self);

// Get the class's name
const char *clsName = class_getName(cls);

// Check if the class is prefixed with _NSZombie_
if (string_has_prefix(clsName, "_NSZombie_") {
    // If so, this object is a zombie

    // Get the original class name by skipping past the
    // _NSZombie_, i.e. taking the substring from character 10
    const char *originalClsName = substring_from(clsName, 10);

    // Get the selector name of the message
    const char *selectorName = sel_getName(_cmd);

    // Log a message to indicate which selector is
    // being sent to which zombie
    Log("*** -[%s %s]: message sent to deallocated instance %p",
        originalClsName, selectorName, self);

// Kill the application
abort();
}
```

把本节开头那个范例扩充一下，试着给变成僵尸的 EOCClass 对象发送 description 消息：

```
EOCClass *obj = [[EOCClass alloc] init];
NSLog(@"Before release:");
PrintClassInfo(obj);

[obj release];
NSLog(@"After release:");
PrintClassInfo(obj);

NSString *desc = [obj description];
```

若是开启了僵尸对象功能，那么控制台会输出下列消息：

```
Before release:
=== EOCClass : NSObject ===
After release:
=== _NSZombie_EOCClass : nil ===
*** -[EOCClass description]: message sent to deallocated
instance 0x7fc821c02a00
```

大家可以看到，这段消息明确指出了僵尸对象所收到的选择子及其原来所属的类，其中还包含接收消息的僵尸对象所对应的“指针值”(pointer value)。在调试器中深入分析程序时，也许会用到此消息，而且若能与适当的工具（比如 Xcode 自带的 Instruments）相搭配，则效果甚佳。

要点

- 系统在回收对象时，可以不将其真的回收，而是把它转化为僵尸对象。通过环境变量 NSZombieEnabled 可开启此功能。
- 系统会修改对象的 isa 指针，令其指向特殊的僵尸类，从而使该对象变为僵尸对象。僵尸类能够响应所有的选择子，响应方式为：打印一条包含消息内容及其接收者的消息，然后终止应用程序。

第 36 条：不要使用 retainCount

Objective-C 通过引用计数来管理内存（参见第 29 条）。每个对象都有一个计数器，其值表明还有多少个其他对象想令此对象继续存活。对象创建好之后，其保留计数大于 0。保留与释放操作分别会使该计数递增及递减。当计数变为 0 时，对象就为系统所回收并摧毁了。

NSObject 协议中定义了下列方法，用于查询对象当前的保留计数：

```
- (NSUInteger)retainCount
```

然而 ARC 已经将此方法废弃了。实际上，如果在 ARC 中调用，编译器就会报错，这和在 ARC 中调用 retain、release、autorelease 方法时的情况一样。虽然此方法已经正式废弃了，但还是经常有人误解它，其实这个方法根本就不应该调用。若在不启用 ARC 的环境下编程（说真的，还是在 ARC 下编程比较好），那么仍可调用此方法，而编译器不会报错。所以，还是必须讲清楚为何不应使用此方法。

这个方法看上去似乎挺合理、挺有用的。它毕竟返回了保留计数，而此值对每个对象来说显然都很重要。但问题在于，保留计数的绝对数值一般都与开发者所应留意的事情完全无关。即便只在调试时才调用此方法，通常也还是无所助益的。

此方法之所以无用，其首要原因在于：它所返回的保留计数只是某个给定时间点上的值。该方法并未考虑到系统会稍后把自动释放池清空（参见第 34 条），因而不会将后续的释放操作从返回值里减去，这样的话，此值就未必能真实反映实际的保留计数了。因此，下面这种写法非常糟糕：

```
while ([object retainCount]) {
    [object release];
}
```

这种写法的第一个错误是：它没考虑到后续的自动释放操作，只是不停地通过释放操作

来降低保留计数，直至对象为系统所回收。假如此对象也在自动释放池里，那么稍后系统清空池子时还要把它再释放一次，而这将导致程序崩溃。

第二个错误在于：retainCount 可能永远不返回 0，因为有时系统会优化对象的释放行为，在保留计数还是 1 的时候就把它回收了。只有在系统不打算这么优化时，计数值才会递减至 0。因此，保留计数可能永远都不会完全归零。所以说，这段代码就算有时能正常运行，也多半是凭运气，而非理性判断。对象回收之后，如果 while 循环仍在运行，那么目前的运行期系统一般会直接令应用程序崩溃。

从来都不需要编写这种代码。这段代码所要实现的操作，应该通过内存管理来解决。开发者在期望系统于某处回收对象时，应该确保没有尚未抵消的保留操作，也就是不要令保留计数大于期望值。在这种情况下，如果发现某对象的内存泄漏了，那么应该检查还有谁仍然保留这个对象，并查明其为何没有释放此对象。

读者可能还是想看一看保留计数的具体值，然而看过之后你就会觉得奇怪了：它的值为何那么大呢？比方说，有下面这段代码：

```
NSString *string = @"Some string";
NSLog(@"string retainCount = %lu", [string retainCount]);

NSNumber *numberI = @1;
NSLog(@"numberI retainCount = %lu", [numberI retainCount]);

NSNumber *numberF = @3.141f;
NSLog(@"numberF retainCount = %lu", [numberF retainCount]);
```

在 64 位 Mac OS X 10.8.2 系统中，用 Clang 4.1 编译后，这段代码输出的消息如下：

```
string retainCount = 18446744073709551615
numberI retainCount = 9223372036854775807
numberF retainCount = 1
```

第一个对象的保留计数是 $2^{64}-1$，第二个对象的保留计数是 $2^{63}-1$。由于二者皆为“单例对象”(singleton object)，所以其保留计数都很大。系统会尽可能把 NSString 实现成单例对象。如果字符串像本例所举的这样，是个编译期常量（compile-time constant），那么就可以这样来实现了。在这种情况下，编译器会把 NSString 对象所表示的数据放到应用程序的二进制文件里，这样的话，运行程序时就可以直接用了，无须再创建 NSString 对象。NSNumber 也类似，它使用了一种叫做“标签指针”（tagged pointer）的概念来标注特定类型的数值。这种做法不使用 NSNumber 对象，而是把与数值有关的全部消息都放在指针值里面。运行期系统会在消息派发（参见第 11 条）期间检测到这种标签指针，并对它执行相应操作，使其行为看上去和真正的 NSNumber 对象一样。这种优化只在某些场合使用，比如范例中的浮点数对象就没有优化，所以其保留计数就是 1。

另外，像刚才所说的那种单例对象，其保留计数绝对不会变。这种对象的保留及释放操作都是“空操作”（no-op）。可以看到，即便两个单例对象之间，其保留计数也各不相同，系

统对其保留计数的这种处理方式再一次表明：我们不应该总是依赖保留计数的具体值来编码。假如你根据 NSNumber 对象的具体保留计数来增减其值，而系统却以标签指针来实现此对象，那么编出来的代码就错了。

那么，只为了调试而使用 retainCount 方法行不行呢？即便只为调试，此方法也不是很有用。由于对象可能处在自动释放池中，所以其保留计数未必如想象般精确。而且其他程序库也有可能自行保留或释放对象，这都会扰乱保留计数的具体取值。看了具体的计数值之后，你可能还误以为是自己的代码修改了它，殊不知其实是由深埋在另外一个程序库中的某段代码所改的。以下列代码为例：

```
id object = [self createObject];
[opaqueObject doSomethingWithObject:object];
NSLog(@"retainCount = %lu", [object retainCount]);
```

object 的保留计数是多少呢？这个计数可以是任意值。“doSomethingWithObject:”方法也许会将对象加到多个 collection 中，而这些 collection 均会保留此对象。这个方法还可能会多次保留并自动释放此对象，而其中某些自动释放操作要留待系统稍后清空自动释放池时才执行。因此，保留计数的实际值就不是那么有用了。

那到底何时才应该用 retainCount 呢？最佳答案是：绝对不要用，尤其考虑到苹果公司在引入 ARC 之后已正式将其废弃，就更不应该用了。

要点

- 对象的保留计数看似有用，实则不然，因为任何给定时间点上的“绝对保留计数”（absolute retain count）都无法反映对象生命期的全貌。
- 引入 ARC 之后，retainCount 方法就正式废止了，在 ARC 下调用该方法会导致编译器报错。

第6章

块与大中枢派发

当前在开发应用程序时，每位程序员都应留意多线程问题。你可能会说自己要开发的应用程序用不到多线程，即便如此，它也很可能依然是多线程的，因为系统框架通常会在UI线程之外再使用一些线程来执行任务。开发应用程序时，最糟糕的事莫过于程序因UI线程阻塞而挂起了。在Mac OS X系统中，这将使鼠标指针一直呈现令人焦急的旋转彩球状；而在iOS系统中，阻塞过久的程序可能会终止执行。

所幸苹果公司以全新方式设计了多线程。当前多线程编程的核心就是“块”(block)与“大中枢派发”（Grand Central Dispatch，GCD)。这虽然是两种不同的技术，但它们是一并引入的。“块”是一种可在C、C++及Objective-C代码中使用的“词法闭包”(lexical closure)，它极为有用，这主要是因为借由此机制，开发者可将代码像对象一样传递，令其在不同环境（context）下运行。还有个关键的地方是，在定义“块”的范围内，它可以访问到其中的全部变量。

GCD是一种与块有关的技术，它提供了对线程的抽象，而这种抽象则基于“派发队列”(dispatch queue)[⊖]。开发者可将块排入队列中，由GCD负责处理所有调度事宜。GCD会根据系统资源情况，适时地创建、复用、摧毁后台线程（background thread)，以便处理每个队列。此外，使用GCD还可以方便地完成常见编程任务，比如编写“只执行一次的线程安全代码”(thread-safe single-code execution)，或者根据可用的系统资源来并发执行多个操作。

块与GCD都是当前Objective-C编程的基石。因此，必须理解其工作原理及功能。

第37条：理解“块”这一概念

块可以实现闭包。这项语言特性是作为“扩展”(extension）而加入GCC编译器中的，在近期版本的Clang中都可以使用（Clang是开发Mac OS X及iOS程序所用的编译器)。10.4版及其后的Mac OS X系统，与4.0版及其后的iOS系统中，都含有正常执行块所需的运行期组件。从技术上讲，这是个位于C语言层面的特性，因此，只要有支持此特性的编译器，以及能执行块的运行期组件，就可以在C、C++、Objective-C、Objective-C++代码中使用它。

⊖ 亦称“调度队列”。——译者注

块的基础知识

块与函数类似，只不过是直接定义在另一个函数里的，和定义它的那个函数共享同一个范围内的东西。块用“^”符号[⊖]来表示，后面跟着一对花括号，括号里面是块的实现代码。例如，下面就是个简单的块：

```
^{
    //Block implementation here
}
```

块其实就是个值，而且自有其相关类型。与 int、float 或 Objective-C 对象一样，也可以把块赋给变量，然后像使用其他变量那样使用它。块类型的语法与函数指针近似。下面列出的这个块很简单，没有参数，也不返回值：

```
void (^someBlock)() = ^{
      //Block implementation here
};
```

这段代码定义了一个名为 someBlock 的变量。由于变量名写在正中间，所以看上去也许有点怪，不过一旦理解了语法，很容易就能读懂。块类型的语法结构如下：

```
return_type (^block_name)(parameters)
```

下面这种写法所定义的块，返回 int 值，并且接受两个 int 做参数：

```
int (^addBlock)(int a, int b) = ^(int a, int b){
    return a + b;
};
```

定义好之后，就可以像函数那样使用了。比方说，addBlock 块可以这样用：

```
int add = addBlock(2, 5); //< add = 7
```

块的强大之处是：在声明它的范围里，所有变量都可以为其所捕获。这也就是说，那个范围里的全部变量，在块里依然可用。比如，下面这段代码所定义的块，就使用了块以外的变量：

```
int additional = 5;
int (^addBlock)(int a, int b) = ^(int a, int b){
    return a + b + additional;
};
int add = addBlock(2, 5); //< add = 12
```

默认情况下，为块所捕获的变量，是不可以在块里修改的。在本例中，假如块内的代码改动了 additional 变量的值，那么编译器就会报错。不过，声明变量的时候可以加上 __block 修饰符，这样就可以在块内修改了。例如，可以用下面这个块来枚举数组中的元素（参见第 48 条），以判断其中有多少个小于 2 的数：

⊖ caret，可称为脱字符或插入符。——译者注

```
NSArray *array = @[@0, @1, @2, @3, @4, @5];
__block NSInteger count = 0;
[array enumerateObjectsUsingBlock:
    ^(NSNumber *number, NSUInteger idx, BOOL *stop){
        if ([number compare:@2] == NSOrderedAscending) {
            count++;
        }
    }];
//count = 2
```

这段范例代码也演示了“内联块”（inline block）的用法。传给“numerateObjects-UsingBlock:”方法的块并未先赋给局部变量，而是直接内联在函数调用里了。由这种常见的编码习惯也可以看出块为何如此有用。在 Objective-C 语言引入块这一特性之前，想要编出与刚才那段代码相同的功能，就必须传入函数指针或选择子的名称，以供枚举方法调用。状态必须手工传入与传出，这一般通过“不透明的 void 指针”（opaque void pointer）实现，如此一来，就得再写几行代码了，而且还会令方法变得有些松散。与之相反，若声明内联形式的块，则可把所有业务逻辑都放在一处。

如果块所捕获的变量是对象类型，那么就会自动保留它。系统在释放这个块的时候，也会将其一并释放。这就引出了一个与块有关的重要问题。块本身可视为对象。实际上，在其他 Objective-C 对象所能响应的选择子中，有很多是块也可以响应的。而最重要之处则在于，块本身也和其他对象一样，有引用计数。当最后一个指向块的引用移走之后，块就回收了。回收时也会释放块所捕获的变量，以便平衡捕获时所执行的保留操作。

如果将块定义在 Objective-C 类的实例方法中，那么除了可以访问类的所有实例变量之外，还可以使用 self 变量。块总能修改实例变量，所以在声明时无须加 __block。不过，如果通过读取或写入操作捕获了实例变量，那么也会自动把 self 变量一并捕获了，因为实例变量是与 self 所指代的实例关联在一起的。例如，下面这个块声明在 EOCClass 类的方法中：

```
@interface EOCClass

- (void)anInstanceMethod {
    //...
    void (^someBlock)() = ^{
        _anInstanceVariable = @"Something";
        NSLog(@"_anInstanceVariable = %@", _anInstanceVariable);
    };
    //...
}

@end
```

如果某个 EOCClass 实例正在执行 anInstanceMethod 方法，那么 self 变量就指向此实例。由于块里没有明确使用 self 变量，所以很容易就会忘记 self 变量其实也为块所捕获了。直接访问实例变量和通过 self 来访问是等效的：

```
self->_anInstanceVariable = @"Something";
```

之所以要捕获 self 变量，原因正在于此。我们经常通过属性（参见第 6 条）访问实例变

量，在这种情况下，就要指明 self 了：

```
self.aProperty = @" Something";
```

然而，一定要记住：self 也是个对象，因而块在捕获它时也会将其保留。如果 self 所指代的那个对象同时也保留了块，那么这种情况通常就会导致“保留环”。更多内容请参阅第 40 条。

块的内部结构

每个 Objective-C 对象都占据着某个内存区域。因为实例变量的个数及对象所包含的关联数据互不相同，所以每个对象所占的内存区域也有大有小。块本身也是对象，在存放块对象的内存区域中，首个变量是指向 Class 对象的指针，该指针叫做 isa（参见第 14 条）。其余内存里含有块对象正常运转所需的各种信息。图 6-1 详细描述了块对象的内存布局。

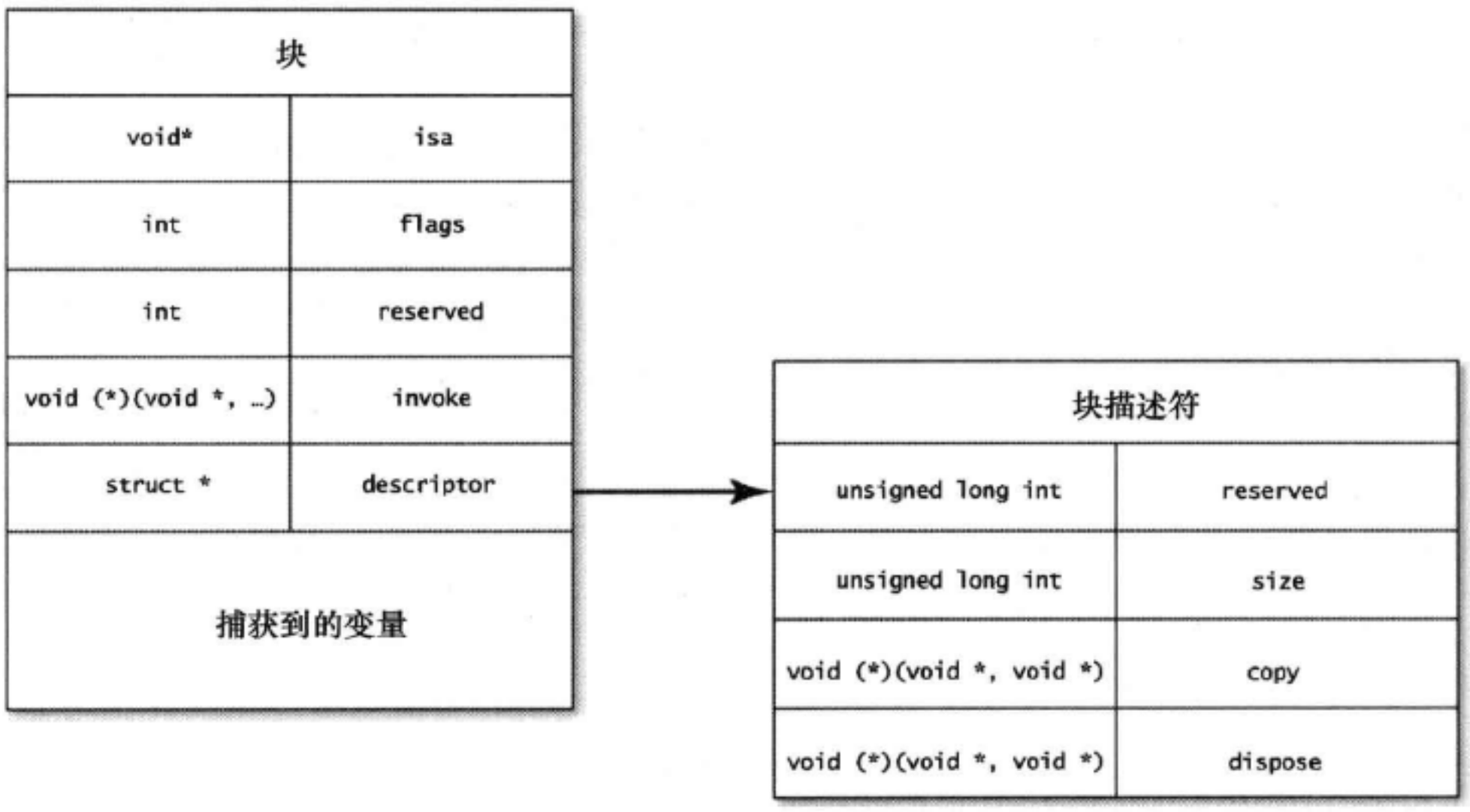

图 6-1 块对象的内存布局

在内存布局中，最重要的就是 invoke 变量，这是个函数指针，指向块的实现代码。函数原型至少要接受一个 void* 型的参数，此参数代表块。刚才说过，块其实就是一种代替函数指针的语法结构，原来使用函数指针时，需要用“不透明的 void 指针”来传递状态。而改用块之后，则可以把原来用标准 C 语言特性所编写的代码封装成简明且易用的接口。

descriptor 变量是指向结构体的指针，每个块里都包含此结构体，其中声明了块对象的总体大小，还声明了 copy 与 dispose 这两个辅助函数所对应的函数指针。辅助函数在拷贝及丢弃块对象时运行，其中会执行一些操作，比方说，前者要保留捕获的对象，而后者则将之释放。

块还会把它所捕获的所有变量都拷贝一份。这些拷贝放在 descriptor 变量后面，捕获了多少个变量，就要占据多少内存空间。请注意，拷贝的并不是对象本身，而是指向这些对象的指针变量。invoke 函数为何需要把块对象作为参数传进来呢？原因就在于，执行块时，要

从内存中把这些捕获到的变量读出来。

全局块、栈块及堆块

定义块的时候，其所占的内存区域是分配在栈中的。这就是说，块只在定义它的那个范围内有效。例如，下面这段代码就有危险：

```
void (^block)();
if ( /* some condition */ ) {
    block = ^{
        NSLog(@"Block A");
    };
} else {
    block = ^{
        NSLog(@"Block B");
    };
}
block();
```

定义在 if 及 else 语句中的两个块都分配在栈内存中。编译器会给每个块分配好栈内存，然而等离开了相应的范围之后，编译器有可能把分配给块的内存覆写掉。于是，这两个块只能保证在对应的 if 或 else 语句范围内有效。这样写出来的代码可以编译，但是运行起来时而正确，时而错误。若编译器未覆写待执行的块，则程序照常运行，若覆写，则程序崩溃。

为解决此问题，可给块对象发送 copy 消息以拷贝之。这样的话，就可以把块从栈复制到堆了。拷贝后的块，可以在定义它的那个范围之外使用。而且，一旦复制到堆上，块就成了带引用计数的对象了。后续的复制操作都不会真的执行复制，只是递增块对象的引用计数。如果不再使用这个块，那就应将其释放，在 ARC 环境下会自动释放，而手动管理引用计数时则需要自己来调用 release 方法。当引用计数降为 0 后，“分配在堆上的块”（heap block）会像其他对象一样，为系统所回收。而“分配在栈上的块”（stack block）则无须明确释放，因为栈内存本来就会自动回收，刚才那段范例代码之所以有危险，原因也在于此。

明白这一点后，我们只需给代码加上两个 copy 方法调用，就可令其变得安全了：

```
void (^block)();
if ( /* some condition */ ) {
    block = [^{
        NSLog(@"Block A");
    } copy];
} else {
    block = [^{
        NSLog(@"Block B");
    } copy];
}
block();
```

现在代码就安全了。如果手动管理引用计数，那么在用完块之后还需将其释放。

除了“栈块”和“堆块”之外，还有一类块叫做“全局块”（global block）。这种块不会捕捉任何状态（比如外围的变量等），运行时也无须有状态来参与。块所使用的整个内存区域，在编译期已经完全确定了，因此，全局块可以声明在全局内存里，而不需要在每次用到的时候于栈中创建。另外，全局块的拷贝操作是个空操作，因为全局块决不可能为系统所回收。这种块实际上相当于单例。下面就是个全局块：

```
void (^block)() = ^{
    NSLog(@"This is a block");
};
```

由于运行该块所需的全部信息都能在编译期确定，所以可把它做成全局块。这完全是种优化技术：若把如此简单的块当成复杂的块来处理，那就会在复制及丢弃该块时执行一些无谓的操作。

要点

- 块是C、C++、Objective-C中的词法闭包。
- 块可接受参数，也可返回值。
- 块可以分配在栈或堆上，也可以是全局的。分配在栈上的块可拷贝到堆里，这样的话，就和标准的Objective-C对象一样，具备引用计数了。

第38条：为常用的块类型创建typedef

每个块都具备其“固有类型”（inherent type），因而可将其赋给适当类型的变量。这个类型由块所接受的参数及其返回值组成。例如有下面这个块：

```
^(BOOL flag, int value){
    if (flag) {
        return value * 5;
    } else {
        return value * 10;
    }
}
```

此块接受两个类型分别为BOOL及int的参数，并返回类型为int的值。如果想把它赋给变量，则需注意其类型。变量类型及相关赋值语句如下：

```
int (^variableName)(BOOL flag, int value) =
    ^(BOOL flag, int value){
        // Implementation
        return someInt;
    }
```

这个类型似乎和普通的类型大不相同，然而如果习惯函数指针的话，那么看上去就会觉得眼熟了。块类型的语法结构如下：

```
return_type (^block_name)(parameters)
```

与其他类型的变量不同，在定义块变量时，要把变量名放在类型之中，而不要放在右侧。这种语法非常难记，也非常难读。鉴于此，我们应该为常用的块类型起个别名，尤其是打算把代码发布成 API 供他人使用时，更应这样做。开发者可以起个更为易读的名字来表示块的用途，而把块的类型隐藏在其后面。

为了隐藏复杂的块类型，需要用到 C 语言中名为“类型定义”（type definition）的特性。typedef 关键字用于给类型起个易读的别名。比方说，想定义新类型，用以表示接受 BOOL 及 int 参数并返回 int 值的块，可通过下列语句来做：

```
typedef int(^EOCSomeBlock)(BOOL flag, int value);
```

声明变量时，要把名称放在类型中间，并在前面加上“^”符号，而定义新类型时也得这么做。上面这条语句向系统中新增了一个名为 EOCSomeBlock 的类型。此后，不用再以复杂的块类型来创建变量了，直接使用新类型即可：

```
EOCSomeBlock block = ^(BOOL flag, int value){
    // Implementation
};
```

这次代码读起来就顺畅多了：与定义其他变量时一样，变量类型在左边，变量名在右边。

通过这项特性，可以把使用块的 API 做得更为易用些。类里面有些方法可能需要用块来做参数，比如执行异步任务时所用的“completion handler”（任务完成后所执行的处理程序）参数就是块，凡遇到这种情况，都可以通过定义别名使代码变得更为易读。比方说，类里有个方法可以启动任务，它接受一个块作为处理程序，在完成任务之后执行这个块。若不定义别名，则方法签名会像下面这样：

```
- (void)startWithCompletionHandler:
            (void(^)(NSData *data, NSError *error))completion;
```

注意，定义方法参数所用的块类型语法，又和定义变量时不同。若能把方法签名中的参数类型写成一个词，那读起来就顺口多了。于是，可以给参数类型起个别名，然后使用此名称来定义：

```
typedef void(^EOCCompletionHandler)
                            (NSData *data, NSError *error);
- (void)startWithCompletionHandler:
            (EOCCompletionHandler)completion;
```

现在参数看上去就简单多了，而且易于理解。当前，优秀的集成开发环境（Integrated Development Environment，IDE）都可以自动把类型定义展开，所以 typedef 这个功能变得很实用。

使用类型定义还有个好处，就是当你打算重构块的类型签名时会很方便。比方说，要给原来的 completion handler 块再加一个参数，用以表示完成任务所花的时间，那么只需修改类

型定义语句即可：

```
typedef void(^EOCCompletionHandler)
        (NSData *data, NSTimeInterval duration, NSError*error);
```

修改之后，凡是使用了这个类型定义的地方，比如方法签名等处，都会无法编译，而且报的是同一种错误，于是开发者可据此逐个修复。若不用类型定义，而直接写块类型，那么代码中要修改的地方就更多了。开发者很容易忘掉其中一两处，从而引发难于排查的 bug。

最好在使用块类型的类中定义这些 typedef，而且还应该把这个类的名字加在由 typedef 所定义的新类型名前面，这样可以阐明块的用途。还可以用 typedef 给同一个块签名类型创建数个别名。在这件事上，多多益善。

Mac OS X 与 iOS 的 Accounts 框架就是个例子。在该框架中可以找到下面这两个类型定义语句：

```
typedef void(^ACAccountStoreSaveCompletionHandler)
                                    (BOOL success, NSError*error);
typedef void(^ACAccountStoreRequestAccessCompletionHandler)
                                    (BOOL granted, NSError*error);
```

这两个类型定义的签名相同，但用在不同的地方。开发者看到类型别名及签名中的参数之后，很容易就能理解此类型的用途。它们本来也可以合并成一个 typedef，比如叫做 ACAccountStoreBooleanCompletionHandler，使用那两个别名的地方，都可以统一使用此名称。然而，这么做之后，块与参数的用途看上去就不那么明显了。

与此相似，如果有好几个类都要执行相似但各有区别的异步任务，而这几个类又不能放入同一个继承体系，那么，每个类就应该有自己的 completion handler 类型。这几个 completion handler 的签名也许完全相同，但最好还是在每个类里都各自定义一个别名，而不要共用同一个名称。反之，若这些类能纳入同一个继承中，则应该将类型定义语句放在超类中，以供各子类使用。

要点

- 以 typedef 重新定义块类型，可令块变量用起来更加简单。
- 定义新类型时应遵从现有的命名习惯，勿使其名称与别的类型相冲突。
- 不妨为同一个块签名定义多个类型别名。如果要重构的代码使用了块类型的某个别名，那么只需修改相应 typedef 中的块签名即可，无须改动其他 typedef。

第 39 条：用 handler 块降低代码分散程度

为用户界面编码时，一种常用的范式就是“异步执行任务”（perform task asynchronously）。这种范式的好处在于：处理用户界面的显示及触摸操作所用的线程，不会因为要执行 I/O 或网络通信这类耗时的任务而阻塞。这个线程通常称为主线程（main thread）。假设

把执行异步任务的方法做成同步的，那么在执行任务时，用户界面就变得无法响应用户输入了。某些情况下，如果应用程序在一定时间内无响应，那么就会自动终止。iOS 系统上的应用程序就是如此，“系统监控器”(system watchdog）在发现某个应用程序的主线程已经阻塞了一段时间之后，就会令其终止。

异步方法在执行完任务之后，需要以某种手段通知相关代码。实现此功能有很多办法。常用的技巧是设计一个委托协议（参见第 23 条），令关注此事件的对象遵从该协议。对象成为 delegate 之后，就可以在相关事件发生时（例如某个异步任务执行完毕时）得到通知了。比方说，要写一个从 URL 中获取数据的类。使用委托模式设计出来的类会是这个样子：

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

@class EOCNetworkFetcher;
@protocol EOCNetworkFetcherDelegate <NSObject>
- (void)networkFetcher:(EOCNetworkFetcher*)networkFetcher
     didFinishWithData:(NSData*)data;
@end

@interface EOCNetworkFetcher : NSObject
@property (nonatomic, weak)
                        id <EOCNetworkFetcherDelegate> delegate;
- (id)initWithURL:(NSURL*)url;
- (void)start;
@end
```

而其他类则可像下面这样使用此类所提供的 API：

```
- (void)fetchFooData {
    NSURL *url = [[NSURL alloc] initWithString:
                  @"http://www.example.com/foo.dat"];
    EOCNetworkFetcher *fetcher =
        [[EOCNetworkFetcher alloc] initWithURL:url];
    fetcher.delegate = self;
    [fetcher start];
}

// ...

- (void)networkFetcher:(EOCNetworkFetcher*)networkFetcher
     didFinishWithData:(NSData*)data
{
    _fetchedFooData = data;
}
```

这种做法确实可行，而且没有什么错误。然而如果改用块来写的话，代码会更清晰。块可以令这种 API 变得更紧致，同时也令开发者调用起来更加方便。办法就是：把 completion handler 定义为块类型，将其当作参数直接传给 start 方法：

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

```
typedef void(^EOCNetworkFetcherCompletionHandler)(NSData *data);

@interface EOCNetworkFetcher : NSObject
- (id)initWithURL:(NSURL*)url;
- (void)startWithCompletionHandler:
            (EOCNetworkFetcherCompletionHandler)handler;
@end
```

这和使用委托协议很像，不过多了个好处，就是可以在调用 start 方法时直接以内联形式定义 completion handler，以此方式来使用“网络数据获取器”（network fetcher），可以令代码比原先易懂很多。例如，下面这个类就以块的形式来定义 completion handler，并以此为参数调用 API：

```
- (void)fetchFooData {
    NSURL *url = [[NSURL alloc] initWithString:
                  @"http://www.example.com/foo.dat"];
    EOCNetworkFetcher *fetcher =
        [[EOCNetworkFetcher alloc] initWithURL:url];
    [fetcher startWithCompletionHandler:^(NSData *data){
        _fetchedFooData = data;
    }];
}
```

与使用委托模式的代码相比，用块写出来的代码显然更为整洁。异步任务执行完毕后所需运行的业务逻辑，和启动异步任务所用的代码放在了一起。而且，由于块声明在创建获取器的范围里，所以它可以访问此范围内的全部变量。本例比较简单，体现不出这一点，然而在更为复杂的场景中，会大有裨益。

委托模式有个缺点：如果类要分别使用多个获取器下载不同数据，那么就得在 delegate 回调方法里根据传入的获取器参数来切换。这种代码的写法如下：

```
- (void)fetchFooData {
    NSURL *url = [[NSURLalloc] initWithString:
                  @"http://www.example.com/foo.dat"];
    _fooFetcher = [[EOCNetworkFetcheralloc]initWithURL:url];
    _fooFetcher.delegate = self;
    [_fooFetcher start];
}

- (void)fetchBarData {
    NSURL *url = [[NSURLalloc] initWithString:
                  @"http://www.example.com/bar.dat"];
    _barFetcher = [[EOCNetworkFetcheralloc]initWithURL:url];
    _barFetcher.delegate = self;
    [_barFetcher start];
}

- (void)networkFetcher:(EOCNetworkFetcher*)networkFetcher
```

```
 didFinishWithData:(NSData*)data
{
    if (networkFetcher == _fooFetcher) {
        _fetchedFooData = data;
        _fooFetcher = nil;
    } else if (networkFetcher == _barFetcher) {
        _fetchedBarData = data;
        _barFetcher = nil;
    }
    //etc.
}
```

这么写代码，不仅会令 delegate 回调方法变得很长，而且还要把网络数据获取器对象保存为实例变量，以便在判断语句中使用。这么做可能有其他原因，比如稍后要根据情况解除监听等，然而这种写法有副作用，通常很快就会使类的代码激增。改用块来写的好处是：无须保存获取器，也无须在回调方法里切换。每个 completion handler 的业务逻辑，都是和相关的获取器对象一起来定义的：

```
- (void)fetchFooData {
    NSURL *url = [[NSURL alloc] initWithString:
                  @"http://www.example.com/foo.dat"];
    EOCNetworkFetcher *fetcher =
        [[EOCNetworkFetcher alloc] initWithURL:url];
    [fetcher startWithCompletionHandler:^(NSData *data){
        _fetchedFooData = data;
    }];
}

- (void)fetchBarData {
    NSURL *url = [[NSURL alloc] initWithString:
                  @"http://www.example.com/bar.dat"];
    EOCNetworkFetcher *fetcher =
        [[EOCNetworkFetcher alloc] initWithURL:url];
    [fetcher startWithCompletionHandler:^(NSData *data){
        _fetchedBarData = data;
    }];
}
```

这种写法还有其他用途，比如，现在很多基于块的 API 都使用块来处理错误。这又分为两种办法。可以分别用两个处理程序来处理操作失败的情况和操作成功的情况。也可以把处理失败情况所需的代码，与处理正常情况所用的代码，都封装到同一个 completion handler 块里。如果想采用两个独立的处理程序，那么可以这样设计 API：

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

@class EOCNetworkFetcher;
typedef void(^EOCNetworkFetcherCompletionHandler)(NSData *data);
typedef void(^EOCNetworkFetcherErrorHandler)(NSError *error);
```

```
@interface EOCNetworkFetcher : NSObject
- (id)initWithURL:(NSURL*)url;
- (void)startWithCompletionHandler:
            (EOCNetworkFetcherCompletionHandler)completion
        failureHandler:
            (EOCNetworkFetcherErrorHandler)failure;
@end
```

依照此风格设计出来的 API，其调用方式如下：

```
EOCNetworkFetcher *fetcher =
    [[EOCNetworkFetcher alloc] initWithURL:url];
[fetcher startWithCompletionHander:^(NSData *data){
    //Handle success
}
                    failureHandler:^(NSError *error){
    //Handle failure
}];
```

这种 API 设计风格很好，由于成功和失败的情况要分别处理，所以调用此 API 的代码也就会按照逻辑，把应对成功和失败情况的代码分开来写，这将令代码更易读懂。而且，若有需要，还可以把处理失败情况或成功情况所用的代码省略。

另一种风格则是像下面这样，把处理成功情况和失败情况所用的代码全放在一个块里：

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

@class EOCNetworkFetcher;
typedef void(^EOCNetworkFetcherCompletionHandler)
                                (NSData *data, NSError *error);

@interface EOCNetworkFetcher : NSObject
- (id)initWithURL:(NSURL*)url;
- (void)startWithCompletionHandler:
                (EOCNetworkFetcherCompletionHandler)completion;
@end
```

此种 API 的调用方式如下：

```
EOCNetworkFetcher *fetcher =
    [[EOCNetworkFetcher alloc] initWithURL:url];
[fetcher startWithCompletionHander:
    ^(NSData *data, NSError *error){
        if (error) {
            //Handle failure
        } else {
            //Handle success
        }
    }];
```

这种方式需要在块代码中检测传入的 error 变量，并且要把所有逻辑代码都放在一处。

这种写法的缺点是：由于全部逻辑都写在一起，所以会令块变得比较长，且比较复杂。然而只用一个块的写法也有好处，那就是更为灵活。比方说，在传入错误信息时，可以把数据也传进来。有时数据正下载到一半，突然网络故障了。在这种情况下，可以把数据及相关的错误都回传给块。这样的话，completion handler 就能据此判断问题并适当处理了，而且还可利用已下载好的这部分数据做些事情。

把成功情况和失败情况放在同一个块中，还有个优点：调用 API 的代码可能会在处理成功响应的过程中发现错误。比方说，返回的数据可能太短了。这种情况需要和网络数据获取器所认定的失败情况按同一方式处理。此时，如果采用单一块的写法，那么就能把这种情况和获取器所认定的失败情况统一处理了。要是把成功情况和失败情况交给两个不同的处理程序来负责，那么就没办法共享同一份错误处理代码了，除非把这段代码单独放在一个方法里，而这又违背了我们想把全部逻辑代码都放在一处的初衷。

总体来说，笔者建议使用同一个块来处理成功与失败情况，苹果公司似乎也是这样设计其 API 的。例如，Twitter 框架中的 TWRequest 及 MapKit 框架中的 MKLocalSearch 都只使用一个 handler 块。

有时需要在相关时间点执行回调操作，这种情况也可以使用 handler 块。比方说，调用网络数据获取器的代码，也许想在每次有下载进度时都得到通知。这可以通过委托模式实现。不过也可以使用本节讲的 handler 块，把处理下载进度的 handler 定义成块类型，并新增一个此类型的属性：

```
typedef void(^EOCNetworkFetcherProgressHandler)
                                        (float progress);
@property (nonatomic, copy)
        EOCNetworkFetcherProgressHandler progressHandler;
```

这种写法很好，因为它还是能把所有业务逻辑都放在一起：也就是把创建网络数据获取器和定义 progress handler 所用的代码写在一处。

基于 handler 来设计 API 还有个原因，就是某些代码必须运行在特定的线程上。比方说，Cocoa 与 Cocoa Touch 中的 UI 操作必须在主线程上执行。这就相当于 GCD 中的“主队列”（main queue）。因此，最好能由调用 API 的人来决定 handler 应该运行在哪个线程上。NSNotificationCenter 就属于这种 API，它提供了一个方法，调用者可以经由此方法来注册想要接收的通知，等到相关事件发生时，通知中心就会执行注册好的那个块。调用者可以指定某个块应该安排在哪个执行队列里，然而这不是必需的。若没有指定队列，则按默认方式执行，也就是说，将由投递通知的那个线程来执行。下列方法可用来新增观察者（observer）：

```
- (id)addObserverForName:(NSString*)name
                  object:(id)object
                   queue:(NSOperationQueue*)queue
              usingBlock:(void(^)(NSNotification*))block
```

此处传入的 NSOperationQueue 参数就表示触发通知时用来执行块代码的那个队列。这是个“操作队列”（operation queue），而非“底层 GCD 队列”（low-level GCD queue），不过两

者语义相同。(第 43 条详细对比了 GCD 队列与其他方式的区别。)

你也可以照此设计自己的 API，根据 API 所处的细节层次，可选用操作队列甚至 GCD 队列来作为参数。

要点

- 在创建对象时，可以使用内联的 handler 块将相关业务逻辑一并声明。
- 在有多个实例需要监控时，如果采用委托模式，那么经常需要根据传入的对象来切换，而若改用 handler 块来实现，则可直接将块与相关对象放在一起。
- 设计 API 时如果用到了 handler 块，那么可以增加一个参数，使调用者可通过此参数来决定应该把块安排在哪个队列上执行。

第 40 条：用块引用其所属对象时不要出现保留环

使用块时，若不仔细思量，则很容易导致“保留环”（retain cycle)。比方说，下面这个类就提供了一套接口，调用者可由此从某个 URL 中下载数据。在启动获取器时，可设置 completion handler，这个块会在下载结束之后以回调方式执行。为了能在下载完成后通过 p_requestCompleted 方法执行调用者所指定的块，这段代码需要把 completion handler 保存到实例变量里面。

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

typede  f void(^EOCNetworkFetcherCompletionHandler)(NSData *data);

@interface EOCNetworkFetcher : NSObject
@property (nonatomic, strong, readonly) NSURL *url;
- (id)initWithURL:(NSURL*)url;
- (void)startWithCompletionHandler:
            (EOCNetworkFetcherCompletionHandler)completion;
@end

//EOCNetworkFetcher.m
#import "EOCNetworkFetcher.h"

@interface EOCNetworkFetcher ()
@property (nonatomic, strong, readwrite) NSURL *url;
@property (nonatomic, copy)
          EOCNetworkFetcherCompletionHandler completionHandler;
@property (nonatomic, strong) NSData *downloadedData;
@end

@implementation EOCNetworkFetcher

- (id)initWithURL:(NSURL*)url {
```

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

- (void)startWithCompletionHandler:
        (EOCNetworkFetcherCompletionHandler)completion
{
    self.completionHandler = completion;
    //Start the request
    //Request sets downloadedData property
    //When request is finished, p_requestCompleted is called
}

- (void)p_requestCompleted {
    if (_completionHandler) {
_completionHandler(_downloadedData);
    }
}

@end
```

某个类可能会创建这种网络数据获取器对象，并用其从 URL 中下载数据：

```
@implementation EOCClass {
    EOCNetworkFetcher *_networkFetcher;
    NSData *_fetchedData;
}

- (void)downloadData {
    NSURL *url = [[NSURL alloc] initWithString:
                  @"http://www.example.com/something.dat"];
    _networkFetcher =
        [[EOCNetworkFetcher alloc] initWithURL:url];
    [_networkFetcher startWithCompletionHandler:^(NSData *data){
        NSLog(@"Request URL %@ finished", _networkFetcher.url);
        _fetchedData = data;
    }];
}

@end
```

这段代码看上去没什么问题。但你可能没发现其中有个保留环。因为 completion handler 块要设置 _fetchedData 实例变量，所以它必须捕获 self 变量（变量捕获问题详见第 37 条）。这就是说，handler 块保留了创建网络数据获取器的那个 EOCClass 实例。而 EOCClass 实例则通过 strong 实例变量保留了获取器，最后，获取器对象又保留了 handler 块。图 6-2 描述了这个保留环。

要打破保留环也很容易：要么令 _networkFetcher 实例变量不再引用获取器，要么令获

取器的 completionHandler 属性不再持有 handler 块。在网络数据获取器这个例子中，应该等 completion handler 块执行完毕后，再去打破保留环，以便使获取器对象在 handler 块执行期间保持存活状态。比方说，completion handler 块的代码可以这么修改：

```
[_networkFetcher startWithCompletionHandler:^(NSData *data){
    NSLog(@"Request for URL %@ finished", _networkFetcher.url);
    _fetchedData = data;
    _networkFetcher = nil;
}
```

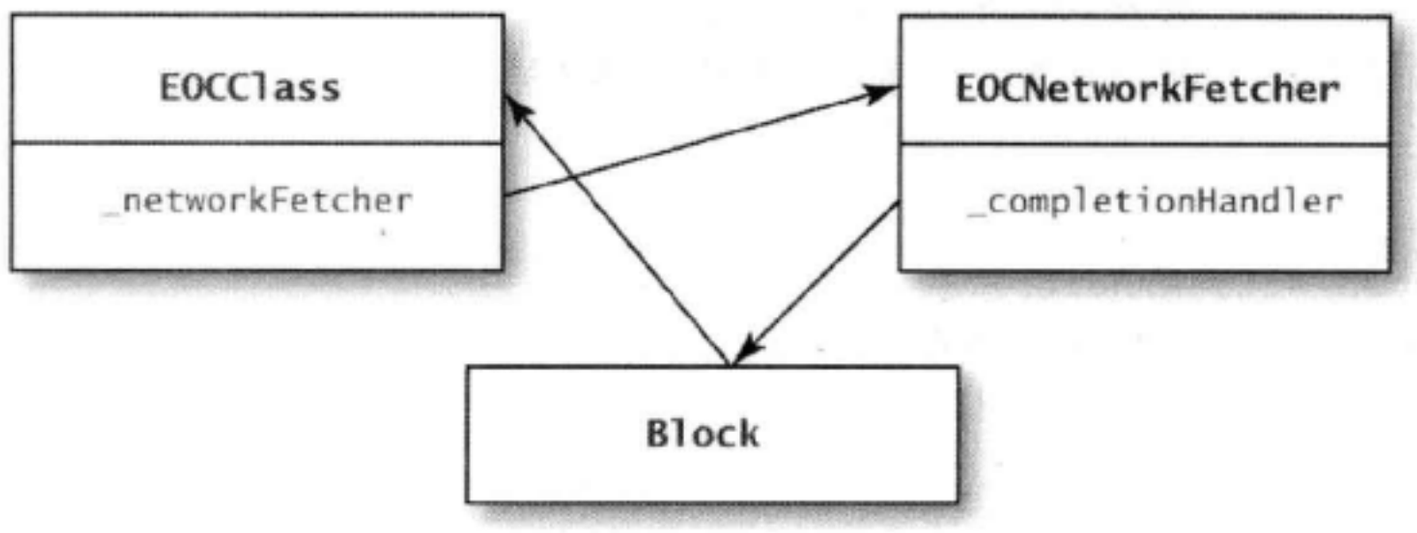

图 6-2　网络数据获取器和拥有它的 EOCClass 类实例之间构成了保留环

如果设计 API 时用到了 completion handler 这样的回调块，那么很容易形成保留环，所以必须意识到这个重要问题。一般来说，只要适时清理掉环中的某个引用，即可解决此问题，然而，未必总有这种机会。在本例中，唯有 completion handler 运行过后，方能解除保留环。若是 completion handler 一直不运行，那么保留环就无法打破，于是内存就会泄漏。

像 completion handler 块这种写法，还可能引入另外一种形式的保留环。如果 completion handler 块所引用的对象最终又引用了这个块本身，那么就会出现保留环。比方说，我们修改一下前面那个例子，使调用 API 的那段代码无须在执行期间保留指向网络数据获取器的引用，而是设定一套机制，令获取器对象自己设法保持存活。要想保持存活，获取器对象可以在启动任务时把自己加到全局的 collection 中（比如用 set 来实现这个 collection），待任务完成后，再移除。而调用方则需将其代码修改如下：

```
- (void)downloadData {
    NSURL *url = [[NSURL alloc] initWithString:
                  @"http://www.example.com/something.dat"];
    EOCNetworkFetcher *networkFetcher =
        [[EOCNetworkFetcher alloc] initWithURL:url];
    [networkFetcher startWithCompletionHandler:^(NSData *data){
        NSLog(@"Request URL %@ finished", networkFetcher.url);
        _fetchedData = data;
    }];
}
```

大部分网络通信库都采用这种办法，因为假如令调用者自己来将获取器对象保持

存话的话，他们会觉得麻烦。Twitter 框架的 TWRequest 对象也用这个办法。然而，就 EOCNetworkFetcher 的现有代码来看，此做法会引入保留环。而这次比刚才那个例子更难于发觉，completion handler 块其实要通过获取器对象来引用其中的 URL。于是，块就要保留获取器，而获取器反过来又经由其 completionHandler 属性保留了这个块。所幸要修复这个问题也不难。回想一下，获取器对象之所以要把 completion handler 块保存在属性里面，其唯一目的就是想稍后使用这个块。可是，获取器一旦运行过 completion handler 之后，就没有必要再保留它了。所以，只需将 p_requestCompleted 方法按如下方式修改即可：

```
- (void)p_requestCompleted {
    if (_completionHandler) {
        _completionHandler(_downloadedData);
    }
    self.completionHandler = nil;
}
```

这样一来，只要下载请求执行完毕，保留环就解除了，而获取器对象也将会在必要时为系统所回收。请注意，之所以要在 start 方法中把 completion handler 作为参数传进去，这也是一条重要原因。假如把 completion handler 暴露为获取器对象的公共属性，那么就不便在执行完下载请求之后直接将其清理掉了，因为既然已经把 handler 作为属性公布了，那就意味着调用者可以自由使用它，若是此时又在内部将其清理掉的话，则会破坏“封装语义”（encapsulation semantic）。在这种情况下要想打破保留环，只有一个办法可用，那就是强迫调用者在 handler 代码里自己把 completionHandler 属性清理干净。可这并不是十分合理，因为你无法假定调用者一定会这么做，他们反过来会抱怨你没把内存泄漏问题处理好。

这两种保留环都很容易发生。使用块来编程时，一不小心就会出现这种 bug，反过来说，只要小心谨慎，这种问题也很容易解决。关键在于，要想清楚块可能会捕获并保留哪些对象。如果这些对象又直接或间接保留了块，那么就要考虑怎样在适当的时机解除保留环。

要点

- 如果块所捕获的对象直接或间接地保留了块本身，那么就得当心保留环问题。
- 一定要找个适当的时机解除保留环，而不能把责任推给 API 的调用者。

第 41 条：多用派发队列，少用同步锁

在 Objective-C 中，如果有多个线程要执行同一份代码，那么有时可能会出问题。这种情况下，通常要使用锁来实现某种同步机制。在 GCD 出现之前，有两种办法，第一种是采用内置的“同步块”（synchronization block）：

```
- (void)synchronizedMethod {
    @synchronized(self) {
        // Safe
    }
}
```

这种写法会根据给定的对象，自动创建一个锁，并等待块中的代码执行完毕。执行到这段代码结尾处，锁就释放了。在本例中，同步行为所针对的对象是 self。这么写通常没错，因为它可以保证每个对象实例都能不受干扰地运行其 synchronizedMethod 方法。然而，滥用 @synchronized（self）则会降低代码效率，因为共用同一个锁的那些同步块，都必须按顺序执行。若是在 self 对象上频繁加锁，那么程序可能要等另一段与此无关的代码执行完毕，才能继续执行当前代码，这样做其实并没有必要。

另一个办法是直接使用 NSLock 对象：

```
_lock = [[NSLockalloc] init];

- (void)synchronizedMethod {
    [_lock lock];
    // Safe
    [_lock unlock];
}
```

也可以使用 NSRecursiveLock 这种“递归锁”(recursive lock)[⊖]，线程能够多次持有该锁，而不会出现死锁（deadlock）现象。

这两种方法都很好，不过也有其缺陷。比方说，在极端情况下，同步块会导致死锁，另外，其效率也不见得很高，而如果直接使用锁对象的话，一旦遇到死锁，就会非常麻烦。

替代方案就是使用 GCD，它能以更简单、更高效的形式为代码加锁。比方说，属性就是开发者经常需要同步的地方，这种属性需要做成“原子的”。用 atomic 特质来修饰属性，即可实现这一点（参见第 6 条）。而开发者如果想自己来编写访问方法的话，那么通常会这样写：

```
- (NSString*)someString {
    @synchronized(self) {
        return _someString;
    }
}

- (void)setSomeString:(NSString*)someString {
    @synchronized(self) {
        _someString = someString;
    }
}
```

刚才说过，滥用 @synchronized（self）会很危险，因为所有同步块都会彼此抢夺同一个锁。要是有很多个属性都这么写的话，那么每个属性的同步块都要等其他所有同步块执行完毕才能执行，这也许并不是开发者想要的效果。我们只是想令每个属性各自独立地同步。

顺便说一下，这么做虽然能提供某种程度的“线程安全”（thread safety），但却无法保证访问该对象时绝对是线程安全的。当然，访问属性的操作确实是“原子的”。使用属性时，必定能从中获取到有效值，然而在同一个线程上多次调用获取方法（getter），每次获取到的

⊖ 也称“重入锁”。——译者注

结果却未必相同。在两次访问操作之间，其他线程可能会写入新的属性值。

有种简单而高效的办法可以代替同步块或锁对象，那就是使用“串行同步队列”（serial synchronization queue）。将读取操作及写入操作都安排在同一个队列里，即可保证数据同步。其用法如下：

```
_syncQueue =
dispatch_queue_create("com.effectiveobjectivec.syncQueue", NULL);

- (NSString*)someString {
    __block NSString *localSomeString;
    dispatch_sync(_syncQueue, ^{
        localSomeString = _someString;
    });
    return localSomeString;
}

- (void)setSomeString:(NSString*)someString {
    dispatch_sync(_syncQueue, ^{
        _someString = someString;
    });
}
```

此模式的思路是：把设置操作与获取操作都安排在序列化的队列里执行，这样的话，所有针对属性的访问操作就都同步了。为了使块代码能够设置局部变量，获取方法中用到了__block 语法，若是抛开这一点，那么这种写法要比前面那些更为整洁。全部加锁任务都在 GCD 中处理，而 GCD 是在相当深的底层来实现的，于是能够做许多优化。因此，开发者无须担心那些事，只要专心把访问方法写好就行。

然而还可以进一步优化。设置方法并不一定非得是同步的。设置实例变量所用的块，并不需要向设置方法返回什么值。也就是说，设置方法的代码可以改成下面这样：

```
- (void)setSomeString:(NSString*)someString {
        dispatch_async(_syncQueue, ^{
            _someString = someString;
    });
}
```

这次只是把同步派发改成了异步派发，从调用者的角度来看，这个小改动可以提升设置方法的执行速度，而读取操作与写入操作依然会按顺序执行。但这么改有个坏处：如果你测一下程序性能，那么可能会发现这种写法比原来慢，因为执行异步派发时，需要拷贝块。若拷贝块所用的时间明显超过执行块所花的时间，则这种做法将比原来更慢。由于本书所举的这个例子很简单，所以改完之后很可能会变慢。然而，若是派发给队列的块要执行更为繁重的任务，那么仍然可以考虑这种备选方案。

多个获取方法可以并发执行，而获取方法与设置方法之间不能并发执行，利用这个特点，还能写出更快一些的代码来。此时正可以体现出 GCD 写法的好处。用同步块或锁对象，是无

法轻易实现出下面这种方案的。这次不用串行队列，而改用并发队列（concurrent queue）：

```
_syncQueue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

- (NSString*)someString {
    __block NSString *localSomeString;
    dispatch_sync(_syncQueue, ^{
        localSomeString = _someString;
    });
    return localSomeString;
}

- (void)setSomeString:(NSString*)someString {
    dispatch_async(_syncQueue, ^{
        _someString = someString;
    });
}
```

像现在这样写代码，还无法正确实现同步。所有读取操作与写入操作都会在同一个队列上执行，不过由于是并发队列，所以读取与写入操作可以随时执行。而我们恰恰不想让这些操作随意执行。此问题用一个简单的GCD功能即可解决，它就是栅栏（barrier）。下列函数可以向队列中派发块，将其作为栅栏使用：

```
void dispatch_barrier_async(dispatch_queue_t queue,
                            dispatch_block_t block);
void dispatch_barrier_sync(dispatch_queue_t queue,
                           dispatch_block_t block);
```

在队列中，栅栏块必须单独执行，不能与其他块并行。这只对并发队列有意义，因为串行队列中的块总是按顺序逐个来执行的。并发队列如果发现接下来要处理的块是个栅栏块（barrier block）[⊖]，那么就一直要等当前所有并发块都执行完毕，才会单独执行这个栅栏块。待栅栏块执行过后，再按正常方式继续向下处理。

在本例中，可以用栅栏块来实现属性的设置方法。在设置方法中使用了栅栏块之后，对属性的读取操作依然可以并发执行，但是写入操作却必须单独执行了。在图6-3所演示的这个队列中，有许多读取操作，而且还有一个写入操作。

实现代码很简单：

```
_syncQueue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,0);

- (NSString*)someString {
    __block NSString *localSomeString;
    dispatch_sync(_syncQueue, ^{
        localSomeString = _someString;
    });
```

⊖ “barrier”一词也称“阻断器”、“障碍”、“屏障”。——译者注

```
    return localSomeString;
}

- (void)setSomeString:(NSString*)someString {
    dispatch_barrier_async(_syncQueue, ^{
        _someString = someString;
    });
}
```

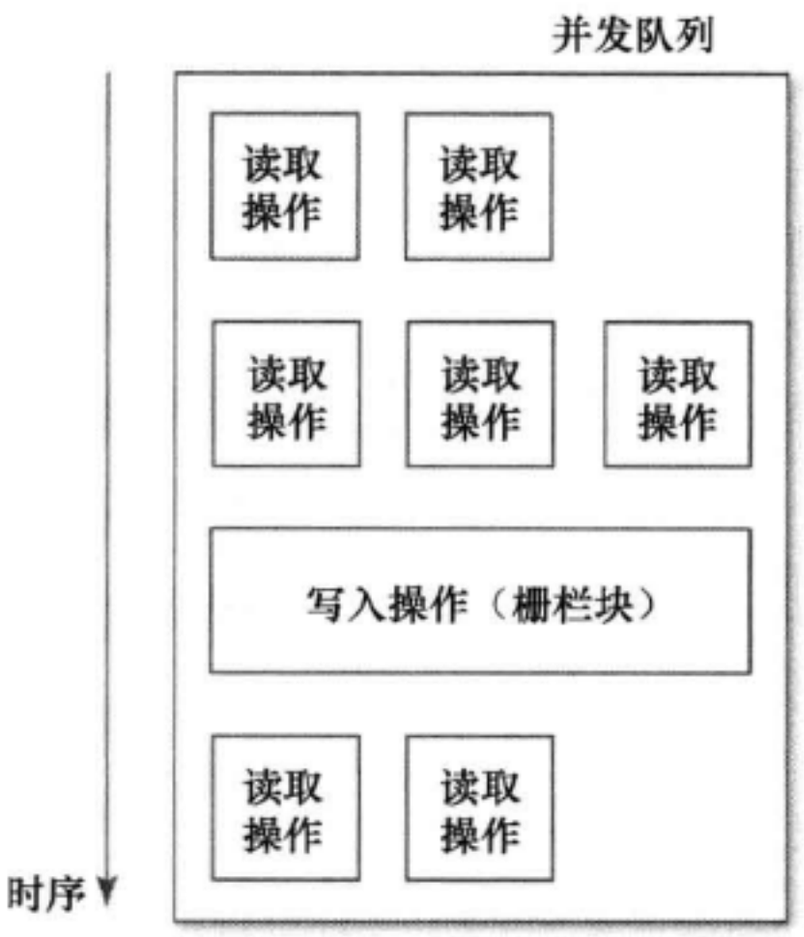

图 6-3　在这个并发队列中，读取操作是用普通的块来实现的，而写入操作则是用栅栏块来实现的。读取操作可以并行，但写入操作必须单独执行，因为它是栅栏块

测试一下性能，你就会发现，这种做法肯定比使用串行队列要快。注意，设置函数也可以改用同步的栅栏块（synchronous barrier）来实现，那样做可能会更高效，其原因刚才已经解释过了。最好还是测一测每种做法的性能，然后从中选出最适合当前场景的方案。

要点

- 派发队列可用来表述同步语义（synchronization semantic），这种做法要比使用 @synchronized 块或 NSLock 对象更简单。
- 将同步与异步派发结合起来，可以实现与普通加锁机制一样的同步行为，而这么做却不会阻塞执行异步派发的线程。
- 使用同步队列及栅栏块，可以令同步行为更加高效。

第 42 条：多用 GCD，少用 performSelector 系列方法

Objective-C 本质上是一门非常动态的语言（参见第 11 条），NSObject 定义了几个方法，令开发者可以随意调用任何方法。这几个方法可以推迟执行方法调用，也可以指定运行方法

所用的线程。这些功能原来很有用，但是在出现了大中枢派发及块这样的新技术之后，就显得不那么必要了。虽说有些代码还是会经常用到它们，但笔者劝你还是避开为妙。

这其中最简单的是“performSelector：”。该方法的签名如下，它接受一个参数，就是要执行的那个选择子：

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

该方法与直接调用选择子等效。所以下面两行代码的执行效果相同：

```
[object performSelector:@selector(selectorName)];
[object selectorName];
```

这种方式看上去似乎多余。如果某个方法只是这么来调用的话，那么此方式确实多余。然而，如果选择子是在运行期决定的，那么就能体现出此方式的强大之处了。这就等于在动态绑定之上再次使用动态绑定，因而可以实现出下面这种功能：

```
SEL selector;
if ( /* some condition */ ) {
    selector = @selector(foo);
} else if ( /* some other condition */ ) {
    selector = @selector(bar);
} else {
    selector = @selector(baz);
}
[object performSelector:selector];
```

这种编程方式极为灵活，经常可用来简化复杂的代码。还有一种用法，就是先把选择子保存起来，等某个事件发生之后再调用。不管哪种用法，编译器都不知道要执行的选择子是什么，这必须到了运行期才能确定。然而，使用此特性的代价是，如果在 ARC 下编译代码，那么编译器会发出如下警示信息：

```
warning: performSelector may cause a leak because its selector
is unknown [-Warc-performSelector-leaks]
```

你可能没料到会出现这种警告。要是早就料到了，那么你也许已经知道使用这些方法时为何要小心了。这条消息看上去可能比较奇怪，而且令人纳闷：为什么其中会提到内存泄漏问题呢？只不过是用“performSelector：”调用了一个方法。原因在于，编译器并不知道将要调用的选择子是什么，因此，也就不了解其方法签名及返回值，甚至连是否有返回值都不清楚。而且，由于编译器不知道方法名，所以就没办法运用 ARC 的内存管理规则来判定返回值是不是应该释放。鉴于此，ARC 采用了比较谨慎的做法，就是不添加释放操作。然而这么做可能导致内存泄漏，因为方法在返回对象时可能已经将其保留了。

考虑下面这段代码：

```
SEL selector;
if ( /* some condition */ ) {
    selector = @selector(newObject);
```

```
} else if ( /* some other condition */ ) {
    selector = @selector(copy);
} else {
    selector = @selector(someProperty);
}
id ret = [object performSelector:selector];
```

此代码与刚才那个例子稍有不同，以便展示问题所在。如果调用的是两个选择子之一，那么 ret 对象应由这段代码来释放，而如果是第三个选择子，则无须释放。不仅在 ARC 环境下应该如此，而且在非 ARC 环境下也应该这么做，这样才算严格遵循了方法的命名规范。如果不使用 ARC（此时编译器也就不发警告信息了），那么在前两种情况下需要手动释放 ret 对象，而在后一种情况下则不需要释放。这个问题很容易忽视，而且就算用静态分析器，也很难侦测到随后的内存泄漏。performSelector 系列的方法之所以要谨慎使用，这就是其中一个原因。

这些方法不甚理想，另一个原因在于：返回值只能是 void 或对象类型。尽管所要执行的选择子也可以返回 void，但是 performSelector 方法的返回值类型毕竟是 id。如果想返回整数或浮点数等类型的值，那么就需要执行一些复杂的转换操作了，而这种转换很容易出错。由于 id 类型表示指向任意 Objective-C 对象的指针，所以从技术上来讲，只要返回值的大小和指针所占大小相同就行，也就是说：在 32 位架构的计算机上，可以返回任意 32 位大小的类型；而在 64 位架构的计算机上，则可返回任意 64 位大小的类型。若返回值的类型为 C 语言结构体，则不可使用 performSelector 方法。

performSelector 还有如下几个版本，可以在发消息时顺便传递参数：

```
- (id)performSelector:(SEL)selector
           withObject:(id)object
- (id)performSelector:(SEL)selector
           withObject:(id)objectA
           withObject:(id)objectB
```

比方说，可以用下面这个版本来设置对象中名为 value 的属性值：

```
id object = /* an object with a property called value */;
id newValue = /* new value for the property */;
[object performSelector:@selector(setValue:)
             withObject:newValue];
```

这些方法貌似有用，但其实局限颇多。由于参数类型是 id，所以传入的参数必须是对象才行。如果选择子所接受的参数是整数或浮点数，那就不能采用这些方法了。此外，选择子最多只能接受两个参数，也就是调用“performSelector：withObject：withObject：”这个版本。而在参数不止两个的情况下，则没有对应的 performSelector 方法能够执行此种选择子。

performSelector 系列方法还有个功能，就是可以延后执行选择子，或将其放在另一个线程上执行。下面列出了此方法中一些更为常用的版本：

```
- (void)performSelector:(SEL)selector
```

```
                withObject:(id)argument
                afterDelay:(NSTimeInterval)delay
- (void)performSelector:(SEL)selector
                onThread:(NSThread*)thread
                withObject:(id)argument
            waitUntilDone:(BOOL)wait
- (void)performSelectorOnMainThread:(SEL)selector
                            withObject:(id)argument
                        waitUntilDone:(BOOL)wait
```

然而很快就会发觉，这些方法太过局限了。例如，具备延后执行功能的那些方法都无法处理带有两个参数的选择子。而能够指定执行线程的那些方法，则与之类似，所以也不是特别通用。如果要用这些方法，就得把许多参数都打包到字典中，然后在受调用的方法里将其提取出来，这样会增加开销，而且还可能出 bug。

如果改用其他替代方案，那就不受这些限制了。最主要的替代方案就是使用块（参见第37条）。而且，performSelector 系列方法所提供的线程功能，都可以通过在大中枢派发机制中使用块来实现。延后执行可以用 dispatch_after 来实现，在另一个线程上执行任务则可通过 dispatch_sync 及 dispatch_async 来实现。

例如，要延后执行某项任务，可以有下面两种实现方式，而我们应该优先考虑第二种：

```
//Using performSelector:withObject:afterDelay:
[self performSelector:@selector(doSomething)
           withObject:nil
           afterDelay:5.0];

//Using dispatch_after
dispatch_time_t time = dispatch_time(DISPATCH_TIME_NOW,
                                     (int64_t)(5.0 * NSEC_PER_SEC));
dispatch_after(time, dispatch_get_main_queue(), ^(void){
    [self doSomething];
});
```

想把任务放在主线程上执行，也可以有下面两种方式，而我们还是应该优选后者：

```
//Using performSelectorOnMainThread:withObject:waitUntilDone:
[self performSelectorOnMainThread:@selector(doSomething)
                       withObject:nil
                    waitUntilDone:NO];

//Using dispatch_async
//(or if waitUntilDone is YES, then dispatch_sync)
dispatch_async(dispatch_get_main_queue(), ^{
    [self doSomething];
});
```

要点

- performSelector 系列方法在内存管理方面容易有疏失。它无法确定将要执行的选择子

具体是什么，因而 ARC 编译器也就无法插入适当的内存管理方法。

- performSelector 系列方法所能处理的选择子太过局限了，选择子的返回值类型及发送给方法的参数个数都受到限制。
- 如果想把任务放在另一个线程上执行，那么最好不要用 performSelector 系列方法，而是应该把任务封装到块里，然后调用大中枢派发机制的相关方法来实现。

第 43 条：掌握 GCD 及操作队列的使用时机

GCD 技术确实很棒，不过有时候采用标准系统库的组件，效果会更好。一定要了解每项技巧的使用时机，如果选错了工具，那么编出来的代码就会难于维护。

很少有其他技术能与 GCD 的同步机制（参见第 41 条）相媲美。对于那些只需执行一次的代码来说，也是如此，使用 GCD 的 dispatch_once（参见第 45 条）最为方便。然而，在执行后台任务时，GCD 并不一定是最佳方式。还有一种技术叫做 NSOperationQueue，它虽然与 GCD 不同，但是却与之相关，开发者可以把操作以 NSOperation 子类的形式放在队列中，而这些操作也能够并发执行。其与 GCD 派发队列有相似之处，这并非巧合。“操作队列”（operation queue）在 GCD 之前就有了，其中某些设计原理因操作队列而流行，GCD 就是基于这些原理构建的。实际上，从 iOS 4 与 Mac OSX 10.6 开始，操作队列在底层是用 GCD 来实现的。

在两者的诸多差别中，首先要注意：GCD 是纯 C 的 API，而操作队列则是 Objective-C 的对象。在 GCD 中，任务用块来表示，而块是个轻量级数据结构（参见第 37 条）。与之相反，“操作”（operation）则是个更为重量级的 Objective-C 对象。虽说如此，但 GCD 并不总是最佳方案。有时候采用对象所带来的开销微乎其微，使用完整对象所带来的好处反而大大超过其缺点。

用 NSOperationQueue 类的“addOperationWithBlock：”方法搭配 NSBlockOperation 类来使用操作队列，其语法与纯 GCD 方式非常类似。使用 NSOperation 及 NSOperationQueue 的好处如下：

- 取消某个操作。如果使用操作队列，那么想要取消操作是很容易的。运行任务之前，可以在 NSOperation 对象上调用 cancel 方法，该方法会设置对象内的标志位，用以表明此任务不需执行，不过，已经启动的任务无法取消。若是不使用操作队列，而是把块安排到 GCD 队列，那就无法取消了。那套架构是“安排好任务之后就不管了”（fire and forget）。开发者可以在应用程序层自己来实现取消功能，不过这样做需要编写很多代码，而那些代码其实已经由操作队列实现好了。
- 指定操作间的依赖关系。一个操作可以依赖其他多个操作。开发者能够指定操作之间的依赖体系，使特定的操作必须在另外一个操作顺利执行完毕后方可执行。比方说，从服务器端下载并处理文件的动作，可以用操作来表示，而在处理其他文件之前，必须先下载“清单文件”（manifest file）。后续的下载操作，都要依赖于先下载清单文件

这一操作。如果操作队列允许并发的话，那么后续的多个下载操作就可以同时执行，但前提是它们所依赖的那个清单文件下载操作已经执行完毕。

- 通过键值观测机制监控 NSOperation 对象的属性。NSOperation 对象有许多属性都适合通过键值观测机制（简称 KVO）来监听，比如可以通过 isCancelled 属性来判断任务是否已取消，又比如可以通过 isFinished 属性来判断任务是否已完成。如果想在某个任务变更其状态时得到通知，或是想用比 GCD 更为精细的方式来控制所要执行的任务，那么键值观测机制会很有用。
- 指定操作的优先级。操作的优先级表示此操作与队列中其他操作之间的优先关系。优先级高的操作先执行，优先级低的后执行。操作队列的调度算法（scheduling algorithm）虽“不透明”（opaque），但必然是经过一番深思熟虑才写成的。反之，GCD 则没有直接实现此功能的办法。GCD 的队列确实有优先级，不过那是针对整个队列来说的，而不是针对每个块来说的。而令开发者在 GCD 之上自己来编写调度算法，又不太合适。因此，在优先级这一点上，操作队列所提供的功能要比 GCD 更为便利。

NSOperation 对象也有“线程优先级”（thread priority），这决定了运行此操作的线程处在何种优先级上。用 GCD 也可以实现此功能，然而采用操作队列更简单，只需设置一个属性。

- 重用 NSOperation 对象。系统内置了一些 NSOperation 的子类（比如 NSBlockOperation）供开发者调用，要是不想用这些固有子类的话，那就得自己来创建了。这些类就是普通的 Objective-C 对象，能够存放任何信息。对象在执行时可以充分利用存于其中的信息，而且还可以随意调用定义在类中的方法。这就比派发队列中那些简单的块要强大许多。这些 NSOperation 类可以在代码中多次使用，它们符合软件开发中的“不重复”（Don't Repeat Yourself，DRY）原则[⊖]。

正如大家所见，操作队列有很多地方胜过派发队列。操作队列提供了多种执行任务的方式，而且都是写好了的，直接就能使用。开发者不用再编写复杂的调度器，也不用自己来实现取消操作或指定操作优先级的功能，这些事情操作队列都已经实现好了。

有一个 API 选用了操作队列而非派发队列，这就是 NSNotificationCenter，开发者可通过其中的方法来注册监听器，以便在发生相关事件时得到通知，而这个方法接受的参数是块，不是选择子。方法原型如下：

```
- (id)addObserverForName:(NSString*)name
                  object:(id)object
                   queue:(NSOperationQueue*)queue
              usingBlock:(void(^)(NSNotification*))block
```

本来这个方法也可以不使用操作队列，而是把处理通知事件所用的块安排在派发队列里。但实际上并没有这样做，其设计者显然使用了高层的 Objective-C API。在这种情况下，两套方案的运行效率没多大差距。设计这个方法的人可能不想使用派发队列，因为那样做将依赖于 GCD，而这种依赖没有必要，前面说过，块本身和 GCD 无关，所以如果仅使用块的

⊖ 详情参阅：https://zh.wikipedia.org/wiki/ 不要重复你自己。——译者注

话，就不会引入对 GCD 的依赖了。也有可能是编写这个方法的人想全部用 Objective-C 来描述，而不想使用纯 C 的东西。

经常会有人说：应该尽可能选用高层 API，只在确有必要时才求助于底层。笔者也同意这个说法，但我并不盲从。某些功能确实可以用高层的 Objective-C 方法来做，但这并不等于说它就一定比底层实现方案好。要想确定哪种方案更佳，最好还是测试一下性能。

要点

- 在解决多线程与任务管理问题时，派发队列并非唯一方案。
- 操作队列提供了一套高层的 Objective-C API，能实现纯 GCD 所具备的绝大部分功能，而且还能完成一些更为复杂的操作，那些操作若改用 GCD 来实现，则需另外编写代码。

第 44 条：通过 Dispatch Group 机制，根据系统资源状况来执行任务

dispatch group[⊖] 是 GCD 的一项特性，能够把任务分组。调用者可以等待这组任务执行完毕，也可以在提供回调函数之后继续往下执行，这组任务完成时，调用者会得到通知。这个功能有许多用途，其中最重要、最值得注意的用法，就是把将要并发执行的多个任务合为一组，于是调用者就可以知道这些任务何时才能全部执行完毕。比方说，可以把压缩一系列文件的任务表示成 dispatch group。

下面这个函数可以创建 dispatch group：

```
dispatch_group_tdispatch_group_create();
```

dispatch group 就是个简单的数据结构，这种结构彼此之间没什么区别，它不像派发队列，后者还有个用来区别身份的标识符。想把任务编组，有两种办法。第一种是用下面这个函数：

```
void dispatch_group_async(dispatch_group_t group,
                          dispatch_queue_t queue,
                          dispatch_block_t block);
```

它是普通 dispatch_async 函数的变体，比原来多一个参数，用于表示待执行的块所归属的组。还有种办法能够指定任务所属的 dispatch group，那就是使用下面这一对函数：

```
void dispatch_group_enter(dispatch_group_t group);
void dispatch_group_leave(dispatch_group_t group);
```

前者能够使分组里正要执行的任务数递增，而后者则使之递减。由此可知，调用了 dispatch_group_enter 以后，必须有与之对应的 dispatch_group_leave 才行。这与引用计数（参见第 29 条）相似，要使用引用计数，就必须令保留操作与释放操作彼此对应，以防内存泄

⊖ 意为“派发分组”或“调度组”。——译者注

漏。而在使用 dispatch group 时，如果调用 enter 之后，没有相应的 leave 操作，那么这一组任务就永远执行不完。

下面这个函数可用于等待 dispatch group 执行完毕：

```
long dispatch_group_wait(dispatch_group_t group,
                         dispatch_time_t timeout);
```

此函数接受两个参数，一个是要等待的 group，另一个是代表等待时间的 timeout 值。timeout 参数表示函数在等待 dispatch group 执行完毕时，应该阻塞多久。如果执行 dispatch group 所需的时间小于 timeout，则返回 0，否则返回非 0 值。此参数也可以取常量 DISPATCH_TIME_FOREVER，这表示函数会一直等着 dispatch group 执行完，而不会超时（time out）。

除了可以用上面那个函数等待 dispatch group 执行完毕之外，也可以换个办法，使用下列函数：

```
void dispatch_group_notify(dispatch_group_t group,
                           dispatch_queue_t queue,
                           dispatch_block_t block);
```

与 wait 函数略有不同的是：开发者可以向此函数传入块，等 dispatch group 执行完毕之后，块会在特定的线程上执行。假如当前线程不应阻塞，而开发者又想在那些任务全部完成时得到通知，那么此做法就很有必要了。比方说，在 Mac OS X 与 iOS 系统中，都不应阻塞主线程，因为所有 UI 绘制及事件处理都要在主线程上执行。

如果想令数组中的每个对象都执行某项任务，并且想等待所有任务执行完毕，那么就可以使用这个 GCD 特性来实现。代码如下：

```
dispatch_queue_t queue =
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_group_t dispatchGroup = dispatch_group_create();
for (id object in collection) {
    dispatch_group_async(dispatchGroup,
                         queue,
                         ^{ [object performTask]; });
}

dispatch_group_wait(dispatchGroup, DISPATCH_TIME_FOREVER);
//Continue processing after completing tasks
```

若当前线程不应阻塞，则可用 notify 函数来取代 wait：

```
dispatch_queue_t notifyQueue = dispatch_get_main_queue();
dispatch_group_notify(dispatchGroup,
                      notifyQueue,
                      ^{
                     //Continue processing after completing tasks
                       });
```

notify 回调时所选用的队列，完全应该根据具体情况来定。笔者在范例代码中使用了主

队列，这是种常见写法。也可以用自定义的串行队列或全局并发队列。

在本例中，所有任务都派发到同一个队列之中。但实际上未必一定要这样做。也可以把某些任务放在优先级高的线程上执行，同时仍然把所有任务都归入同一个 dispatch group，并在执行完毕时获得通知：

```
dispatch_queue_t lowPriorityQueue =
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_LOW, 0);
dispatch_queue_t highPriorityQueue =
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0);
dispatch_group_t dispatchGroup = dispatch_group_create();

for (id object in lowPriorityObjects) {
    dispatch_group_async(dispatchGroup,
                         lowPriorityQueue,
                         ^{ [object performTask]; });
}

for (id object in highPriorityObjects) {
    dispatch_group_async(dispatchGroup,
                         highPriorityQueue,
                         ^{ [object performTask]; });
}

dispatch_queue_t notifyQueue = dispatch_get_main_queue();
dispatch_group_notify(dispatchGroup,
                      notifyQueue,
                      ^{
                    //Continue processing after completing tasks
                       });
```

除了像上面这样把任务提交到并发队列之外，也可以把任务提交至各个串行队列中，并用 dispatch group 跟踪其执行状况。然而，如果所有任务都排在同一个串行队列里面，那么 dispatch group 就用处不大了。因为此时任务总要逐个执行，所以只需在提交完全部任务之后再提交一个块即可，这样做与通过 notify 函数等待 dispatch group 执行完毕然后再回调块是等效的：

```
dispatch_queue_t queue =
dispatch_queue_create("com.effectiveobjectivec.queue", NULL);

for (id object in collection) {
    dispatch_async(queue,
                   ^{ [object performTask]; });
}

dispatch_async(queue,
               ^{
                    //Continue processing after completing tasks
                });
```

上面这段代码表明，开发者未必总需要使用 dispatch group。有时候采用单个队列搭配标准的异步派发，也可实现同样效果。

笔者为何要在标题中谈到“根据系统资源状况来执行任务”呢？回头看看向并发队列派发任务的那个例子，就会明白了。为了执行队列中的块，GCD 会在适当的时机自动创建新线程或复用旧线程。如果使用并发队列，那么其中有可能会有多个线程，这也就意味着多个块可以并发执行。在并发队列中，执行任务所用的并发线程数量，取决于各种因素，而 GCD 主要是根据系统资源状况来判定这些因素的。假如 CPU 有多个核心，并且队列中有大量任务等待执行，那么 GCD 就可能会给该队列配备多个线程。通过 dispatch group 所提供的这种简便方式，既可以并发执行一系列给定的任务，又能在全部任务结束时得到通知。由于 GCD 有并发队列机制，所以能够根据可用的系统资源状况来并发执行任务。而开发者则可以专注于业务逻辑代码，无须再为了处理并发任务而编写复杂的调度器。

在前面的范例代码中，我们遍历某个 collection，并在其每个元素上执行任务，而这也可以用另外一个 GCD 函数来实现：

```
void dispatch_apply(size_t iterations,
                    dispatch_queue_t queue,
                    void(^block)(size_t));
```

此函数会将块反复执行一定的次数，每次传给块的参数值都会递增，从 0 开始，直至“iterations-1”。其用法如下：

```
dispatch_queue_t queue =
  dispatch_queue_create("com.effectiveobjectivec.queue", NULL);
dispatch_apply(10, queue, ^(size_t i){
    // Perform task
});
```

采用简单的 for 循环，从 0 递增至 9，也能实现同样效果：

```
for (int i = 0; i <10; i++) {
    // Perform task
}
```

有一件事要注意：dispatch_apply 所用的队列可以是并发队列。如果采用并发队列，那么系统就可以根据资源状况来并行执行这些块了，这与使用 dispatch group 的那段范例代码一样。上面这个 for 循环要处理的 collection 若是数组，则可用 dispatch_apply 改写如下：

```
dispatch_queue_t queue =
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_apply(array.count, queue, ^(size_t i){
    id object = array[i];
    [object performTask];
});
```

这个例子再次表明：未必总要使用 dispatch group。然而，dispatch_apply 会持续阻塞，

直到所有任务都执行完毕为止。由此可见：假如把块派给了当前队列（或者体系中高于当前队列的某个串行队列），就将导致死锁。若想在后台执行任务，则应使用 dispatch group。

要点

- 一系列任务可归入一个 dispatch group 之中。开发者可以在这组任务执行完毕时获得通知。
- 通过 dispatch group，可以在并发式派发队列里同时执行多项任务。此时 GCD 会根据系统资源状况来调度这些并发执行的任务。开发者若自己来实现此功能，则需编写大量代码。

第 45 条：使用 dispatch_once 来执行只需运行一次的线程安全代码

单例模式（singleton）对 Objective-C 开发者来说并不陌生，常见的实现方式为：在类中编写名为 sharedInstance 的方法，该方法只会返回全类共用的单例实例，而不会在每次调用时都创建新的实例。假设有个类叫做 EOCClass，那么这个共享实例的方法一般都会这样写：

```
@implementation EOCClass

+ (id)sharedInstance {
    static EOCClass *sharedInstance = nil;
    @synchronized(self) {
        if (!sharedInstance) {
            sharedInstance = [[self alloc] init];
        }
    }
    return sharedInstance;
}

@end
```

笔者发现单例模式容易引起激烈争论，Objective-C 的单例尤其如此。线程安全是大家争论的主要问题。为保证线程安全，上述代码将创建单例实例的代码包裹在同步块里。无论是好是坏，反正这种实现方式很常用，这样的代码随处可见。

不过，GCD 引入了一项特性，能使单例实现起来更为容易。所用的函数是：

```
void dispatch_once(dispatch_once_t *token,
                   dispatch_block_t block);
```

此函数接受类型为 dispatch_once_t 的特殊参数，笔者称其为“标记”（token），此外还接受块参数。对于给定的标记来说，该函数保证相关的块必定会执行，且仅执行一次。首次调用该函数时，必然会执行块中的代码，最重要的一点在于，此操作完全是线程安全的。请注意，对于只需执行一次的块来说，每次调用函数时传入的标记都必须完全相同。因此，开发者通常将标记变量声明在 static 或 global 作用域里。

刚才实现单例模式所用的 sharedInstance 方法，可以用此函数来改写：

```
+ (id)sharedInstance {
    static EOCClass *sharedInstance = nil;
    stat icdispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedInstance = [[self alloc] init];
    });
    return sharedInstance;
}
```

使用 dispatch_once 可以简化代码并且彻底保证线程安全，开发者根本无须担心加锁或同步。所有问题都由 GCD 在底层处理。由于每次调用时都必须使用完全相同的标记，所以标记要声明成 static。把该变量定义在 static 作用域中，可以保证编译器在每次执行 sharedInstance 方法时都会复用这个变量，而不会创建新变量。

此外，dispatch_once 更高效。它没有使用重量级的同步机制，若是那样做的话，每次运行代码前都要获取锁，相反，此函数采用“原子访问”（atomic access）来查询标记，以判断其所对应的代码原来是否已经执行过。笔者在自己装有 64 位 Mac OS X 10.8.2 系统的电脑上简单测试了性能，分别采用 @synchronized 方式及 dispatch_once 方式来实现 sharedInstance 方法，结果显示，后者的速度几乎是前者的两倍。

要点

- 经常需要编写“只需执行一次的线程安全代码”（thread-safe single-code execution）。通过 GCD 所提供的 dispatch_once 函数，很容易就能实现此功能。
- 标记应该声明在 static 或 global 作用域中，这样的话，在把只需执行一次的块传给 dispatch_once 函数时，传进去的标记也是相同的。

第46条：不要使用 dispatch_get_current_queue

使用 GCD 时，经常需要判断当前代码正在哪个队列上执行，向多个队列派发任务时，更是如此。例如，Mac OS X 与 iOS 的 UI 事务都需要在主线程上执行，而这个线程就相当于 GCD 中的主队列。有时似乎需要判断出当前代码是不是在主队列上执行。阅读开发文档时，大家会发现下面这个函数：

```
dispatch_queue_t dispatch_get_current_queue()
```

文档中说，此函数返回当前正在执行代码的队列。确实是这样，不过用的时候要小心。实际上，iOS 系统从 6.0 版本起，已经正式弃用此函数了。不过 Mac OS X 系统直到 10.8 版本也尚未将其废弃。虽说如此，但在 Mac OS X 系统里还是要避免使用它。

该函数有种典型的错误用法（antipattern，“反模式”），就是用它检测当前队列是不是某个特定的队列，试图以此来避免执行同步派发时可能遭遇的死锁问题。考虑下面这两个存取方法，其代码用队列来保证对实例变量的访问操作是同步的（参见第 41 条）：

```
- (NSString*)someString {
    __block NSString *localSomeString;
    dispatch_sync(_syncQueue, ^{
        localSomeString = _someString;
    });
    return localSomeString;
}

- (void)setSomeString:(NSString*)someString {
    dispatch_async(_syncQueue, ^{
        _someString = someString;
    });
}
```

这种写法的问题在于，获取方法（getter）可能会死锁，假如调用获取方法的队列恰好是同步操作所针对的队列（本例中是 _syncQueue），那么 dispatch_sync 就一直不会返回，直到块执行完毕为止。可是，应该执行块的那个目标队列却是当前队列，而当前队列的 dispatch_sync 又一直阻塞着，它在等待目标队列把这个块执行完，这样一来，块就永远没机会执行了。像 someString 这种方法，就是“不可重入的”。

看了 dispatch_get_current_queue 的文档后，你也许觉得可以用它改写这个方法，令其变得“可重入”，只需检测当前队列是否为同步操作所针对的队列，如果是，就不派发了，直接执行块即可：

```
- (NSString*)someString {
    __block NSString *localSomeString;
    dispatch_block_t accessorBlock = ^{
        localSomeString = _someString;
    };

    if (dispatch_get_current_queue() == _syncQueue) {
        accessorBlock();
    } else {
        dispatch_sync(_syncQueue, accessorBlock);
    }

    return localSomeString;
}
```

这种做法可以处理一些简单情况。不过仍然有死锁的危险。为说明其原因，请读者考虑下面这段代码，其中有两个串行派发队列：

```
dispatch_queue_t queueA =
  dispatch_queue_create("com.effectiveobjectivec.queueA", NULL);
dispatch_queue_t queueB =
  dispatch_queue_create("com.effectiveobjectivec.queueB", NULL);

dispatch_sync(queueA, ^{
```

```
    dispatch_sync(queueB, ^{
        dispatch_sync(queueA, ^{
            // Deadlock
        });
    });
});
```

这段代码执行到最内层的派发操作时，总会死锁，因为此操作是针对 queueA 队列的，所以必须等最外层的 dispatch_sync 执行完毕才行[⊖]，而最外层的那个 dispatch_sync 又不可能执行完毕，因为它要等最内层的 dispatch_sync 执行完，于是就死锁了。现在按照刚才的办法，使用 dispatch_get_current_queue 来检测：

```
dispatch_sync(queueA, ^{
    dispatch_sync(queueB, ^{
        dispatch_block_t block = ^{ /* ... */ };
        if (dispatch_get_current_queue() == queueA) {
            block();
        } else {
            dispatch_sync(queueA, block);
        }
    });
});
```

然而这样做依然死锁，因为 dispatch_get_current_queue 返回的是当前队列，在本例中就是 queueB。这样的话，针对 queueA 的同步派发操作依然会执行，于是和刚才一样，还是死锁了。

在这种情况下，正确做法是：不要把存取方法做成可重入的，而是应该确保同步操作所用的队列绝不会访问属性，也就是绝对不会调用 someString 方法。这种队列只应该用来同步属性。由于派发队列是一种极为轻量的机制，所以，为了确保每项属性都有专用的同步队列，我们不妨创建多个队列。

刚才那个例子似乎稍显做作，但是使用队列时还要注意另外一个问题，而那个问题会在你意想不到的地方导致死锁。队列之间会形成一套层级体系，这意味着排在某条队列中的块，会在其上级队列（parent queue，也叫“父队列”）里执行。层级里地位最高的那个队列总是“全局并发队列”（global concurrent queue）。图 6-4 描绘了一套简单的队列体系。

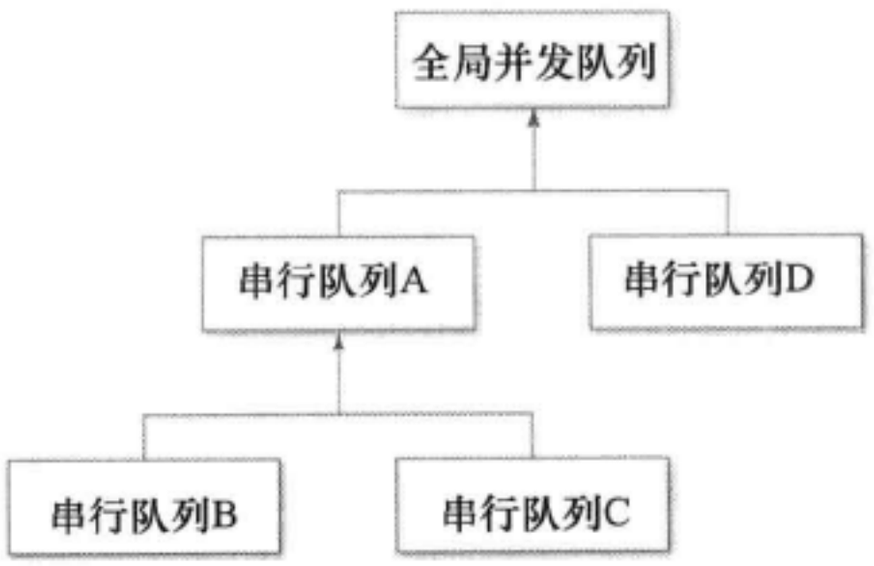

图 6-4 派发队列层级体系

排在队列 B 或队列 C 中的块，稍后会在队列 A 里依序执行。于是，排在队列 A、B、C 中的块总是要彼此错开执行。然而，安排在队列 D 中的块，则有可能与队列 A 里的块（也包括队列 B 与 C 里的块）并行，因为 A 与 D 的目标队列是个并发队列。若有必要，并发队列可以用多个线程并行执行多个块，而是否会这样做，则需根据 CPU 的核心数量等系统资源状况来定。

⊖ 因为最外层的派发操作与最内层一样，也是针对 queueA 的。——译者注

由于队列间有层级关系，所以“检查当前队列是否为执行同步派发所用的队列”这种办法，并不总是奏效。比方说，排在队列 C 里的块，会认为当前队列就是队列 C，而开发者可能会据此认定：在队列 A 上能够安全地执行同步派发操作。但实际上，这么做依然会像前面那样导致死锁。

有的 API 可令开发者指定运行回调块时所用的队列，但实际上却会把回调块安排在内部的串行同步队列上，而内部队列的目标队列又是开发者所提供的那个队列，在此情况下，也许就要出现刚才说的那种问题了。使用这种 API 的开发者可能误以为：在回调块里调用 dispatch_get_current_queue 所返回的“当前队列”，总是其调用 API 时指定的那个。但实际上返回的却是 API 内部的那个同步队列。

要解决这个问题，最好的办法就是通过 GCD 所提供的功能来设定“队列特有数据”（queue-specific data），此功能可以把任意数据以键值对的形式关联到队列里。最重要之处在于，假如根据指定的键获取不到关联数据，那么系统就会沿着层级体系向上查找，直至找到数据或到达根队列为止。笔者这么说，大家也许还不太明白其用法，所以来看下面这个例子：

```
dispatch_queue_t queueA =
  dispatch_queue_create("com.effectiveobjectivec.queueA", NULL);
dispatch_queue_t queueB =
  dispatch_queue_create("com.effectiveobjectivec.queueB", NULL);
dispatch_set_target_queue(queueB, queueA);

static int kQueueSpecific;
CFStringRef queueSpecificValue = CFSTR("queueA");
dispatch_queue_set_specific(queueA,
                            &kQueueSpecific,
                            (void*)queueSpecificValue,
                            (dispatch_function_t)CFRelease);

dispatch_sync(queueB, ^{
    dispatch_block_t block = ^{ NSLog(@"No deadlock!"); };

    CFStringRef retrievedValue =
                    dispatch_get_specific(&kQueueSpecific);
    if (retrievedValue) {
        block();
    } else {
        dispatch_sync(queueA, block);
    }
});
```

本例创建了两个队列。代码中将队列 B 的目标队列设为队列 A，而队列 A 的目标队列仍然是默认优先级的全局并发队列。然后使用下列函数，在队列 A 上设置“队列特定值”：

```
void dispatch_queue_set_specific(dispatch_queue_t queue,
                                 const void *key,
                                 void *context,
                                 dispatch_function_t destructor);
```

此函数的首个参数表示待设置数据的队列，其后两个参数是键与值。键与值都是不透明的 void 指针。对于键来说，有个问题一定要注意：函数是按指针值来比较键的，而不是按照其内容。所以，“队列特定数据”的行为与 NSDictionary 对象不同，后者是比较键的“对象等同性”。“队列特定数据”更像是“关联引用”（associated reference，参见第 10 条）。值（在函数原型里叫做“context”[⊖]）也是不透明的 void 指针，于是可以在其中存放任意数据。然而，必须管理该对象的内存。这使得在 ARC 环境下很难使用 Objective-C 对象作为值。范例代码使用 CoreFoundation 字符串作为值，因为 ARC 并不会自动管理 CoreFoundation 对象的内存。所以说，这种对象非常适合充当“队列特定数据”，它们可以根据需要与相关的 Objective-C Foundation 类无缝衔接。

函数的最后一个参数是“析构函数”（destructor function），对于给定的键来说，当队列所占内存为系统所回收，或者有新的值与键相关联时，原有的值对象就会移除，而析构函数也会于此时运行。dispatch_function_t 类型的定义如下：

```
typedef void (*dispatch_function_t)(void*)
```

由此可知，析构函数只能带有一个指针参数且返回值必须为 void。范例代码采用 CFRelease 做析构函数，此函数符合要求，不过也可以采用开发者自定义的函数，在其中调用 CFRelease 以清理旧值，并完成其他必要的清理工作。

于是，“队列特定数据”所提供的这套简单易用的机制，就避免了使用 dispatch_get_current_queue 时经常遭遇的一个陷阱。此外，调试程序时也许会经常用到 dispatch_get_current_queue。在此情况下，可以放心使用这个已经废弃的方法，只是别把它编译到发行版的程序里就行。如果对“访问当前队列”这项操作有特殊需求，而现有函数又无法满足，那么最好还是联系苹果公司，请求其加入此功能。

要点

- dispatch_get_current_queue 函数的行为常常与开发者所预期的不同。此函数已经废弃，只应做调试之用。
- 由于派发队列是按层级来组织的，所以无法单用某个队列对象来描述“当前队列”这一概念。
- dispatch_get_current_queue 函数用于解决由不可重入的代码所引发的死锁，然而能用此函数解决的问题，通常也能改用“队列特定数据”来解决。

⊖ 中文称为“上下文”、“语境”、“环境参数”等。——译者注

第7章

系统框架

虽说不使用系统框架也能编写 Objective-C 代码，但几乎没人这么做。即便是 NSObject 这个标准的根类，也属于 Foundation 框架，而非语言本身。若不使用 Foundation，就必须自己编写根类，同时还要自己来写 collection、事件循环，以及其他会用到的类。此外，若不用系统框架，也就无法使用 Objective-C 来开发 Mac OS X 及 iOS 应用程序了。系统框架很强大，不过它是历经多年研发才成了今天这个样子的。因此，里面也许会有不合时宜而且用起来很蹩脚的地方，但也会有遗失的珍宝藏于其间。

第 47 条：熟悉系统框架

编写 Objective-C 应用程序时几乎都会用到系统框架，其中提供了许多编程中经常使用的类，比如 collection。若是不了解系统框架所提供的内容，那么就可能会把其中已经实现过的东西又重写一遍。用户升级操作系统后，你所开发的应用程序也可以使用最新版的系统库了。所以说，如果直接使用这些框架中的类，那么应用程序就可以得益于新版系统库所带来的改进，而开发者也就无须手动更新其代码了。

将一系列代码封装为动态库（dynamic library），并在其中放入描述其接口的头文件，这样做出来的东西就叫框架。有时为 iOS 平台构建的第三方框架所使用的是静态库（static library），这是因为 iOS 应用程序不允许在其中包含动态库。这些东西严格来讲并不是真正的框架，然而也经常视为框架。不过，所有 iOS 平台的系统框架仍然使用动态库。

在为 Mac OS X 或 iOS 系统开发“带图形界面的应用程序”（graphical application）时，会用到名为 Cocoa 的框架，在 iOS 上称为 Cocoa Touch。其实 Cocoa 本身并不是框架，但是里面集成了一批创建应用程序时经常会用到的框架。

开发者会碰到的主要框架就是 Foundation，像是 NSObject、NSArray、NSDictionary 等类都在其中。Foundation 框架中的类，使用 NS 这个前缀，此前缀是在 Objective-C 语言用作 NeXTSTEP 操作系统[㊀] 的编程语言时首度确定的。Foundation 框架真可谓所有 Objective-C 应

㊀ 二十世纪八九十年代的操作系统，后来成为 Mac OS X 系统的基础。详情参见：https://en.wikipedia.org/wiki/NeXTSTEP。——译者注

用程序的“基础”，若是没有它，那么本书大部分内容就不知所云了。

Foundation 框架不仅提供了 collection 等基础核心功能，而且还提供了字符串处理这样的复杂功能。比方说，NSLinguisticTagger 可以解析字符串并找到其中的全部名词、动词、代词等。简言之，Foundation 所提供的功能远远不止那几个基础类。

还有个与 Foundation 相伴的框架，叫做 CoreFoundation。虽然从技术上讲，CoreFoundation 框架不是 Objective-C 框架，但它却是编写 Objective-C 应用程序时所应熟悉的重要框架，Foundation 框架中的许多功能，都可以在此框架中找到对应的 C 语言 API。CoreFoundation 与 Foundation 不仅名字相似，而且还有更为紧密的联系。有个功能叫做“无缝桥接”（toll-free bridging）[⊖]，可以把 CoreFoundation 中的 C 语言数据结构平滑转换为 Foundation 中的 Objective-C 对象，也可以反向转换。比方说，Foundation 框架中的字符串是 NSString，而它可以转换为 CoreFoundation 里与之等效的 CFString 对象。无缝桥接技术是用某些相当复杂的代码实现出来的，这些代码可以使运行期系统把 CoreFoundation 框架中的对象视为普通的 Objective-C 对象。但是，像无缝桥接这么复杂的技术，想自己编写代码实现它，可不太容易。开发程序时可以使用此功能，但若决定以手工编码的方式来复刻这套机制，则需认真审视自己的想法了。

除了 Foundation 与 CoreFoundation 之外，还有很多系统库，其中包括但不限于下面列出的这些：

- **CFNetwork** 此框架提供了 C 语言级别的网络通信能力，它将“BSD 套接字”（BSD socket）抽象成易于使用的网络接口。而 Foundation 则将该框架里的部分内容封装为 Objective-C 语言的接口，以便进行网络通信，例如可以用 NSURLConnection 从 URL 中下载数据。
- **CoreAudio** 该框架所提供的 C 语言 API 可用来操作设备上的音频硬件。这个框架属于比较难用的那种，因为音频处理本身就很复杂。所幸由这套 API 可以抽象出另外一套 Objective-C 式 API，用后者来处理音频问题会更简单些。
- **AVFoundation** 此框架所提供的 Objective-C 对象可用来回放并录制音频及视频，比如能够在 UI 视图类里播放视频。
- **CoreData** 此框架所提供的 Objective-C 接口可将对象放入数据库，便于持久保存[⊜]。CoreData 会处理数据的获取及存储事宜，而且可以跨越 Mac OS X 及 iOS 平台。
- **CoreText** 此框架提供的 C 语言接口可以高效执行文字排版及渲染操作。

除此之外，还有别的框架，然而通过此处列出的这几个框架，可以看出 Objective-C 编程的一项重要特点，那就是：经常需要使用底层的 C 语言级 API。用 C 语言来实现 API 的好处是，可以绕过 Objective-C 的运行期系统，从而提升执行速度。当然，由于 ARC 只负责 Objective-C 的对象（参见第 30 条），所以使用这些 API 时尤其需要注意内存管理问题。若想

⊖ 直译为“免费桥接”。——译者注

⊜ 也称“将对象持久化至数据库中”。——译者注

使用这种框架，一定得熟悉 C 语言基础才行。

读者可能会编写使用 UI 框架的 Mac OS X 或 iOS 应用程序。这两个平台的核心 UI 框架分别叫做 AppKit 及 UIKit，它们都提供了构建在 Foundation 与 CoreFoundation 之上的 Objective-C 类。框架里含有 UI 元素，也含有粘合机制，令开发者可将所有相关内容组装为应用程序。在这些主要的 UI 框架之下，是 CoreAnimation 与 CoreGraphics 框架。

CoreAnimation 是用 Objective-C 语言写成的，它提供了一些工具，而 UI 框架则用这些工具来渲染图形并播放动画。开发者编程时可能从来不会深入到这种级别，不过知道该框架总是好的。CoreAnimation 本身并不是框架，它是 QuartzCore 框架的一部分。然而在框架的国度里，CoreAnimation 仍应算作“一等公民”(first-class citizen)。

CoreGraphics 框架以 C 语言写成，其中提供了 2D 渲染所必备的数据结构与函数。例如，其中定义了 CGPoint、CGSize、CGRect 等数据结构，而 UIKit 框架中的 UIView 类在确定视图控件之间的相对位置时，这些数据结构都要用到。

还有很多框架构建在 UI 框架之上，比方说 MapKit 框架，它可以为 iOS 程序提供地图功能。又比如 Social 框架，它为 Mac OS X 及 iOS 程序提供了社交网络（social networking）功能。开发者通常会将这些框架与操作系统平台所对应的核心 UI 框架结合起来使用。

总的来说，许多框架都是安装 Mac OS X 与 iOS 系统时的标准配置。所以，在打算编写新的工具类之前，最好在系统框架里搜一下，通常都有写好的类可供直接使用。

要点

- 许多系统框架都可以直接使用。其中最重要的是 Foundation 与 CoreFoundation，这两个框架提供了构建应用程序所需的许多核心功能。
- 很多常见任务都能用框架来做，例如音频与视频处理、网络通信、数据管理等。
- 请记住：用纯 C 写成的框架与用 Objective-C 写成的一样重要，若想成为优秀的 Objective-C 开发者，应该掌握 C 语言的核心概念。

第 48 条：多用块枚举，少用 for 循环

在编程中经常需要列举 collection 中的元素，当前的 Objective-C 语言有多种办法实现此功能，可以用标准的 C 语言循环，也可以用 Objective-C 1.0 的 NSEnumerator 以及 Objective-C 2.0 的快速遍历（fast enumeration）。语言中引入“块”这一特性后，又多出来几种新的遍历方式，而这几种方式容易为开发者所忽视。采用这几种新方式遍历 collection 时，可以传入块，而 collection 中的每个元素都可能会放在块里运行一遍，这种做法通常会大幅度简化编码过程，笔者下面将会详细说明。

本条所讲的 collection 包含 NSArray、NSDictionary、NSSet 这几个频繁使用的类型。此外，这里所说的遍历技巧也适用于自定义的 collection，但是具体做法并不在本条范围内。

for 循环

遍历数组的第一种办法就是采用老式的 for 循环，这令人想起：在作为 Objective-C 根基的 C 语言里，就已经有此特性了。这是个很基本的办法，因而功能非常有限。通常会这样写代码：

```
NSArray *anArray = /* ... */;
for (int i = 0; i < anArray.count; i++) {
    id object = anArray[i];
    // Do something with 'object'
}
```

这么写还好，不过若要遍历字典或 set，就要复杂一些了：

```
// Dictionary
NSDictionary *aDictionary = /* ... */;
NSArray *keys = [aDictionary allKeys];
for (int i = 0; i < keys.count; i++) {
    id key = keys[i];
    id value = aDictionary[key];
    // Do something with 'key' and 'value'
}

// Set
NSSet *aSet = /* ... */;
NSArray *objects = [aSet allObjects];
for (int i = 0; i < objects.count; i++) {
    id object = objects[i];
    // Do something with 'object'
}
```

根据定义，字典与 set 都是“无序的”（unordered），所以无法根据特定的整数下标来直接访问其中的值。于是，就需要先获取字典里的所有键或是 set 里的所有对象，这两种情况下，都可以在获取到的有序数组上遍历，以便借此访问原字典及原 set 中的值。创建这个附加数组会有额外开销，而且还会多创建一个数组对象，它会保留 collection 中的所有元素对象。当然了，释放数组时这些附加对象也要释放，可是要调用本来不需执行的方法。其他各种遍历方式都无须创建这种中介数组。

for 循环也可以实现反向遍历，计数器的值从“元素个数减 1”开始，每次迭代时递减，直到 0 为止。执行反向遍历时，使用 for 循环会比其他方式简单许多。

使用 Objective-C 1.0 的 NSEnumerator 来遍历

NSEnumerator 是个抽象基类，其中只定义了两个方法，供其具体子类（concrete subclass）来实现：

```
- (NSArray*)allObjects
- (id)nextObject
```

其中关键的方法是 nextObject，它返回枚举里的下个对象。每次调用该方法时，其内部数据结构都会更新，使得下次调用方法时能返回下个对象。等到枚举中的全部对象都已返回之后，再调用就将返回 nil，这表示达到枚举末端了。

Foundation 框架中内建的 collection 类都实现了这种遍历方式。例如，想遍历数组，可以这样写代码：

```
NSArray *anArray = /* ... */;
NSEnumerator *enumerator = [anArray objectEnumerator];
id object;
while ((object = [enumerator nextObject]) != nil) {
    // Do something with 'object'
}
```

这种写法的功能与标准的 for 循环相似，但是代码却多了一些。其真正优势在于：不论遍历哪种 collection，都可以采用这套相似的语法。比方说，遍历字典及 set 时也可以按照这种写法来做：

```
// Dictionary
NSDictionary *aDictionary = /* ... */;
NSEnumerator *enumerator = [aDictionary keyEnumerator];
id key;
while ((key = [enumerator nextObject]) != nil) {
    id value = aDictionary[key];
    // Do something with 'key' and 'value'
}

// Set
NSSet *aSet = /* ... */;
NSEnumerator *enumerator = [aSet objectEnumerator];
id object;
while ((object = [enumerator nextObject]) != nil) {
    // Do something with 'object'
}
```

遍历字典的方式与数组和 set 略有不同，因为字典里既有键也有值，所以要根据给定的键把对应的值提取出来。使用 NSEnumerator 还有个好处，就是有多种“枚举器”(enumerator)可供使用。比方说，有反向遍历数组所用的枚举器，如果拿它来遍历，就可以按反方向来迭代 collection 中的元素了。例如：

```
NSArray *anArray = /* ... */;
NSEnumerator *enumerator = [anArray reverseObjectEnumerator];
id object;
while ((object = [enumerator nextObject]) != nil) {
    // Do something with 'object'
}
```

与采用 for 循环的等效写法相比，上面这段代码读起来更顺畅。

快速遍历

Objective-C 2.0引入了快速遍历这一功能。快速遍历与使用NSEnumerator来遍历差不多，然而语法更简洁，它为for循环开设了in关键字。这个关键字大幅简化了遍历collection所需的语法，比方说要遍历数组，就可以这么写：

```
NSArray *anArray = /* ... */;
for (id object in anArray) {
    // Do something with 'object'
}
```

这样写简单多了。如果某个类的对象支持快速遍历，那么就可以宣称自己遵从名为NSFastEnumeration的协议，从而令开发者可以采用此语法来迭代该对象。此协议只定义了一个方法：

```
- (NSUInteger)countByEnumeratingWithState:
                                (NSFastEnumerationState*)state
                                    objects:(id*)stackbuffer
                                      count:(NSUInteger)length
```

该方法的工作原理不在本条目所述范围内。不过网上能找到一些优秀的教程，它们会把这个问题解释得很清楚。其要点在于：该方法允许类实例同时返回多个对象，这就使得循环遍历操作更为高效了。

遍历字典与set也很简单：

```
// Dictionary
NSDictionary *aDictionary = /* ... */;
for (id key in aDictionary) {
    id value = aDictionary[key];
    // Do something with 'key' and 'value'
}

// Set
NSSet *aSet = /* ... */;
for (id object in aSet) {
    // Do something with 'object'
}
```

由于NSEnumerator对象也实现了NSFastEnumeration协议，所以能用来执行反向遍历。若要反向遍历数组，可采用下面这种写法：

```
NSArray *anArray = /* ... */;
for (id object in [anArray reverseObjectEnumerator]) {
    // Do something with 'object'
}
```

在目前所介绍的遍历方式中，这种办法是语法最简单且效率最高的，然而如果在遍历字典时需要同时获取键与值，那么会多出来一步。而且，与传统的for循环不同，这种遍历方式无法

轻松获取当前遍历操作所针对的下标。遍历时通常会用到这个下标，比如很多算法都需要它。

基于块的遍历方式

在当前的 Objective-C 语言中，最新引入的一种做法就是基于块来遍历。NSArray 中定义了下面这个方法，它可以实现最基本的遍历功能：

```
- (void)enumerateObjectsUsingBlock:
    (void(^)(id object, NSUInteger idx, BOOL *stop))block
```

除此之外，还有一系列类似的遍历方法，它们可以接受各种选项，以控制遍历操作，稍后将会讨论那些方法。

在遍历数组及 set 时，每次迭代都要执行由 block 参数所传入的块，这个块有三个参数，分别是当前迭代所针对的对象、所针对的下标，以及指向布尔值的指针。前两个参数的含义不言而喻。而通过第三个参数所提供的机制，开发者可以终止遍历操作。

例如，下面这段代码用此方法来遍历数组：

```
NSArray *anArray = /* ... */;
[anArray enumerateObjectsUsingBlock:
    ^(id object, NSUInteger idx, BOOL *stop){
        // Do something with 'object'
        if (shouldStop) {
            *stop = YES;
        }
    }];
```

这种写法稍微多了几行代码，不过依然明晰，而且遍历时既能获取对象，也能知道其下标。此方法还提供了一种优雅的机制，用于终止遍历操作，开发者可以通过设定 stop 变量值来实现，当然，使用其他几种遍历方式时，也可以通过 break 来终止循环，那样做也很好。

此方式不仅可用来遍历数组。NSSet 里面也有同样的块枚举方法，NSDictionary 也是这样，只是略有不同：

```
- (void)enumerateKeysAndObjectsUsingBlock:
                    (void(^)(id key, id object, BOOL *stop))block
```

因此，遍历字典与 set 也同样简单：

```
// Dictionary
NSDictionary *aDictionary = /* ... */;
[aDictionary enumerateKeysAndObjectsUsingBlock:
    ^(id key, id object, BOOL *stop){
        // Do something with 'key' and 'object'
        if (shouldStop) {
            *stop = YES;
        }
    }];

// Set
```

```
NSSet *aSet = /* ... */;
[aSet enumerateObjectsUsingBlock:
    ^(id object, BOOL *stop){
        // Do something with 'object'
        if (shouldStop) {
            *stop = YES;
        }
    }];
```

此方式大大胜过其他方式的地方在于：遍历时可以直接从块里获取更多信息。在遍历数组时，可以知道当前所针对的下标。遍历有序 set（NSOrderedSet）时也一样。而在遍历字典时，无须额外编码，即可同时获取键与值，因而省去了根据给定键来获取对应值这一步。用这种方式遍历字典，可以同时得知键与值，这很可能比其他方式快很多，因为在字典内部的数据结构中，键与值本来就是存储在一起的。

另外一个好处是，能够修改块的方法签名，以免进行类型转换操作，从效果上讲，相当于把本来需要执行的类型转换操作交给块方法签名来做。比方说，要用“快速遍历法”来遍历字典。若已知字典中的对象必为字符串，则可以这样编码：

```
for (NSString *key in aDictionary) {
    NSString *object = (NSString*)aDictionary[key];
    // Do something with 'key' and 'object'
}
```

如果改用基于块的方式来遍历，那么就可以在块方法签名中直接转换：

```
NSDictionary *aDictionary = /* ... */;
[aDictionary enumerateKeysAndObjectsUsingBlock:
    ^(NSString *key, NSString *obj, BOOL *stop){
        // Do something with 'key' and 'obj'
    }];
```

之所以能如此，是因为 id 类型相当特殊，它可以像本例这样，为其他类型所覆写。要是原来的块签名把键与值都定义成 NSObject*，那这么写就不行了。此技巧初看不甚显眼，实则相当有用。指定对象的精确类型之后，编译器就可以检测出开发者是否调用了该对象所不具备的方法，并在发现这种问题时报错。如果能够确知某 collection 里的对象是什么类型，那就应该使用这种方法指明其类型。

用此方式也可以执行反向遍历。数组、字典、set 都实现了前述方法的另一个版本，使开发者可向其传入“选项掩码”（option mask）：

```
- (void)enumerateObjectsWithOptions:
            (NSEnumerationOptions)options
                    usingBlock:
            (void(^)(id obj, NSUInteger idx, BOOL *stop))block
- (void)enumerateKeysAndObjectsWithOptions:
            (NSEnumerationOptions)options
                        usingBlock:
            (void(^)(id key, id obj, BOOL *stop))block
```

NSEnumerationOptions 类型是个 enum，其各种取值可用“按位或”（bitwise OR）连接，用以表明遍历方式。例如，开发者可以请求以并发方式执行各轮迭代，也就是说，如果当前系统资源状况允许，那么执行每次迭代所用的块就可以并行执行了。通过 NSEnumerationConcurrent 选项即可开启此功能。如果使用此选项，那么底层会通过 GCD 来处理并发执行事宜，具体实现时很可能会用到 dispatch group（参见第 44 条）。不过，到底如何来实现，不是本条所要讨论的内容。反向遍历是通过 NSEnumerationReverse 选项来实现的。要注意：只有在遍历数组或有序 set 等有顺序的 collection 时，这么做才有意义。

总体来看，块枚举法拥有其他遍历方式都具备的优势，而且还能带来更多好处。与快速遍历法相比，它要多用一些代码，可是却能提供遍历时所针对的下标，在遍历字典时也能同时提供键与值，而且还有选项可以开启并发迭代功能，所以多写这点代码还是值得的。

要点

- 遍历 collection 有四种方式。最基本的办法是 for 循环，其次是 NSEnumerator 遍历法及快速遍历法，最新、最先进的方式则是“块枚举法”。
- “块枚举法”本身就能通过 GCD 来并发执行遍历操作，无须另行编写代码。而采用其他遍历方式则无法轻易实现这一点。
- 若提前知道待遍历的 collection 含有何种对象，则应修改块签名，指出对象的具体类型。

第 49 条：对自定义其内存管理语义的 collection 使用无缝桥接

Objective-C 的系统库包含相当多的 collection 类，其中有各种数组、各种字典、各种 set。Foundation 框架定义了这些 collection 及其他各种 collection 所对应的 Objective-C 类。与之相似，CoreFoundation 框架也定义了一套 C 语言 API，用于操作表示这些 collection 及其他各种 collection 的数据结构。例如，NSArray 是 Foundation 框架中表示数组的 Objective-C 类，而 CFArray 则是 CoreFoundation 框架中的等价物。这两种创建数组的方式也许有区别，然而有项强大的功能可在这两个类型之间平滑转换，它就是“无缝桥接”(toll-free bridging)。

使用“无缝桥接”技术，可以在定义于 Foundation 框架中的 Objective-C 类和定义于 CoreFoundation 框架中的 C 数据结构之间互相转换。笔者将 C 语言级别的 API 称为数据结构，而没有称其为类或对象，这是因为它们与 Objective-C 中的类或对象并不相同。例如，CFArray 要通过 CFArrayRef 来引用，而这是指向 struct__CFArray 的指针。CFArrayGetCount 这种函数则可以操作此 struct，以获取数组大小。这和 Objective-C 中的对应物不同，在 Objective-C 中，可以创建 NSArray 对象，并在该对象上调用 count 方法，以获取数组大小。

下列代码演示了简单的无缝桥接：

```
NSArray *anNSArray = @[@1, @2, @3, @4, @5];
CFArrayRef aCFArray = (__bridge CFArrayRef)anNSArray;
NSLog(@"Size of array = %li", CFArrayGetCount(aCFArray));
//Output: Size of array = 5
```

转换操作中的 __bridge 告诉 ARC（参见第 30 条）如何处理转换所涉及的 Objective-C 对象。__bridge 本身的意思是：ARC 仍然具备这个 Objective-C 对象的所有权。而 __bridge_retained 则与之相反，意味着 ARC 将交出对象的所有权。若是前面那段代码改用它来实现，那么用完数组之后就要加上 CFRelease（aCFArray）以释放其内存。与之相似，反向转换可通过 __bridge_transfer 来实现。比方说，想把 CFArrayRef 转换为 NSArray*，并且想令 ARC 获得对象所有权，那么就可以采用此种转换方式。这三种转换方式称为“桥式转换”（bridged cast）。

可是，你也许会问：以纯 Objective-C 来编写应用程序时，为何要用到这种功能呢？这是因为：Foundation 框架中的 Objective-C 类所具备的某些功能，是 CoreFoundation 框架中的 C 语言数据结构所不具备的，反之亦然。在使用 Foundation 框架中的字典对象时会遇到一个大问题，那就是其键的内存管理语义为“拷贝”，而值的语义却是“保留”。除非使用强大的无缝桥接技术，否则无法改变其语义。

CoreFoundation 框架中的字典类型叫做 CFDictionary。其可变版本称为 CFMutableDictionary。创建 CFMutableDictionary 时，可以通过下列方法来指定键和值的内存管理语义：

```
CFMutableDictionaryRef CFDictionaryCreateMutable(
    CFAllocatorRef allocator,
    CFIndex capacity,
    const CFDictionaryKeyCallBacks *keyCallBacks,
    const CFDictionaryValueCallBacks *valueCallBacks
)
```

首个参数表示将要使用的内存分配器（allocator）[⊖]。如果你大部分时间都在编写 Objective-C 代码，那么也许会对 CoreFoundation 框架中的这部分稍感陌生。CoreFoundation 对象里的数据结构需要占用内存，而分配器负责分配及回收这些内存。开发者通常为这个参数传入 NULL，表示采用默认的分配器。

第二个参数定义了字典的初始大小。它并不会限制字典的最大容量，只是向分配器提示了一开始应该分配多少内存。假如要创建的字典含有 10 个对象，那就向该参数传入 10。

最后两个参数值得注意。它们定义了许多回调函数，用于指示字典中的键和值在遇到各种事件时应该执行何种操作。这两个参数都是指向结构体的指针，二者所对应的结构体如下：

```
struct CFDictionaryKeyCallBacks {
    CFIndex version;
    CFDictionaryRetainCallBack retain;
    CFDictionaryReleaseCallBack release;
    CFDictionaryCopyDescriptionCallBack copyDescription;
    CFDictionaryEqualCallBack equal;
    CFDictionaryHashCallBack hash;
};

struct CFDictionaryValueCallBacks {
```

⊖ 也称“配置器”。——译者注

```
    CFIndex version;
    CFDictionaryRetainCallBack retain;
    CFDictionaryReleaseCallBack release;
    CFDictionaryCopyDescriptionCallBack copyDescription;
    CFDictionaryEqualCallBack equal;
};
```

version 参数目前应设为 0。当前编程时总是取这个值，不过将来苹果公司也许会修改此结构体，所以要预留该值以表示版本号。这个参数可以用于检测新版与旧版数据结构之间是否兼容。结构体中的其余成员都是函数指针，它们定义了当各种事件发生时应该采用哪个函数来执行相关任务。比方说，如果字典中加入了新的键与值，那么就会调用 retain 函数。此参数的类型定义如下：

```
typedef const void* (*CFDictionaryRetainCallBack) (
    CFAllocatorRef allocator,
    const void *value
);
```

由此可见，retain 是个函数指针，其所指向的函数接受两个参数，其类型分别是 CFAllocatorRef 与 const void*。传给此函数的 value 参数表示即将加入字典中的键或值。而返回的 void* 则表示要加到字典里的最终值。开发者可以用下列代码来实现这个回调函数：

```
const void* CustomCallback(CFAllocatorRef allocator,
                           const void *value)
{
    return value;
}
```

这么写只是把即将加入字典中的值照原样返回。于是，如果用它充当 retain 回调函数来创建字典，那么该字典就不会“保留”键与值了。将此种写法与无缝桥接搭配起来，就可以创建出特殊的 NSDictionary 对象，而其行为与用 Objective-C 创建出来的普通字典不同。

下列范例代码完整演示了这种字典的创建步骤：

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

const void* EOCRetainCallback(CFAllocatorRef allocator,
                              const void *value)
{
    return CFRetain(value);
}

void EOCReleaseCallback(CFAllocatorRef allocator,
                        const void *value)
{
    CFRelease(value);
}

CFDictionaryKeyCallBacks keyCallbacks = {
```

```
        0,
        EOCRetainCallback,
        EOCReleaseCallback,
        NULL,
        CFEqual,
        CFHash
    };

    CFDictionaryValueCallBacks valueCallbacks = {
        0,
        EOCRetainCallback,
        EOCReleaseCallback,
        NULL,
        CFEqual
    };

    CFMutableDictionaryRef aCFDictionary =
        CFDictionaryCreateMutable(NULL,
                                  0,
                                  &keyCallbacks,
                                  &valueCallbacks);

    NSMutableDictionary *anNSDictionary =
        (__bridge_transfer NSMutableDictionary*)aCFDictionary;
```

在设定回调函数时，copyDescription取值为NULL，因为采用默认实现就很好。而equal与hash回调函数分别设为CFEqual与CFHash，因为这二者所采用的做法与NSMutableDictionary的默认实现相同。CFEqual最终会调用NSObject的“isEqual：”方法，而CFHash则会调用hash方法。由此可以看出无缝桥接技术更为强大的一面。

键与值所对应的retain与release回调函数指针分别指向EOCRetainCallback与EOCRelease Callback函数。为什么要这么做呢？回想一下，前面说过，在向NSMutableDictionary中加入键和值时，字典会自动“拷贝”键并“保留”值。如果用作键的对象不支持拷贝操作，那会如何呢？此时就不能使用普通的NSMutableDictionary了，假如用了，会导致下面这种运行期错误：

```
*** Terminating app due to uncaught exception
'NSInvalidArgumentException', reason: '-[EOCClass
copyWithZone:]: unrecognized selector sent to instance
0x7fd069c080b0
```

该错误表明，对象所属的类不支持NSCopying协议，因为“copyWithZone：”方法未实现。开发者可以直接在CoreFoundation层创建字典，于是就能修改内存管理语义，对键执行“保留”而非“拷贝”操作了。

通过类似手段，也可创建出不保留其元素对象的数组或set。这么做或许有用，因为有时如果令数组保留对象的话，那么可能会引入“保留环”。不过要注意，这个问题可以改用

更好的办法来解决[⊖]。不保留其元素对象的那种数组，很容易出错。要是数组中的某个对象已为系统所回收，而应用程序又去访问该对象的话，那很可能就崩溃了。

要点

- 通过无缝桥接技术，可以在 Foundation 框架中的 Objective-C 对象与 CoreFoundation 框架中的 C 语言数据结构之间来回转换。
- 在 CoreFoundation 层面创建 collection 时，可以指定许多回调函数，这些函数表示此 collection 应如何处理其元素。然后，可运用无缝桥接技术，将其转换成具备特殊内存管理语义的 Objective-C collection。

第 50 条：构建缓存时选用 NSCache 而非 NSDictionary

开发 Mac OS X 或 iOS 应用程序时，经常会遇到一个问题，那就是从因特网下载的图片应如何来缓存。首先能想到的好办法就是把内存中的图片保存到字典里，这样的话，稍后使用时就无须再次下载了。有些程序员会不假思索，直接使用 NSDictionary 来做（准确来说，是使用其可变版本），因为这个类很常用。其实，NSCache 类更好，它是 Foundation 框架专为处理这种任务而设计的。

NSCache 胜过 NSDictionary 之处在于，当系统资源将要耗尽时，它可以自动删减缓存。如果采用普通的字典，那么就要自己编写挂钩，在系统发出“低内存”（low memory）通知时手工删减缓存。而 NSCache 则会自动删减，由于其是 Foundation 框架的一部分，所以与开发者相比，它能在更深的层面上插入挂钩。此外，NSCache 还会先行删减“最久未使用的”（lease recently used）对象。若想自己编写代码来为字典添加此功能，则会十分复杂。

NSCache 并不会“拷贝”键，而是会“保留”它。此行为用 NSDictionary 也可以实现，然而需要编写相当复杂的代码（参见第 49 条）。NSCache 对象不拷贝键的原因在于：很多时候，键都是由不支持拷贝操作的对象来充当的。因此，NSCache 不会自动拷贝键，所以说，在键不支持拷贝操作的情况下，该类用起来比字典更方便。另外，NSCache 是线程安全的。而 NSDictionary 则绝对不具备此优势，意思就是：在开发者自己不编写加锁代码的前提下，多个线程便可以同时访问 NSCache。对缓存来说，线程安全通常很重要，因为开发者可能要在某个线程中读取数据，此时如果发现缓存里找不到指定的键，那么就要下载该键所对应的数据了。而下载完数据之后所要执行的回调函数，有可能会放在背景线程中运行，这样的话，就等于是用另外一个线程来写入缓存了。

开发者可以操控缓存删减其内容的时机。有两个与系统资源相关的尺度可供调整，其一是缓存中的对象总数，其二是所有对象的“总开销”（overall cost）。开发者在将对象加入缓存时，可为其指定“开销值”。当对象总数或总开销超过上限时，缓存就可能会删减其中的对

⊖ 参见本书第 33 条。——译者注

象了，在可用的系统资源趋于紧张时，也会这么做。然而要注意，“可能”会删减某个对象，并不意味着“一定”会删减这个对象。删减对象时所遵照的顺序，由具体实现来定。这尤其说明：想通过调整“开销值”来迫使缓存优先删减某对象，不是个好主意。

向缓存中添加对象时，只有在能很快计算出“开销值”的情况下，才应该考虑采用这个尺度。若计算过程很复杂，那么照这种方式来使用缓存就达不到最佳效果了，因为每次向缓存中放入对象时，还要专门花时间来计算这个附加因素的值。而缓存的本意则是要增加应用程序响应用户操作的速度。比方说，如果计算“开销值”时必须访问磁盘才能确定文件大小，或是必须访问数据库才能决定具体取值，那就不太好了。然而，如果要加入缓存中的是NSData对象，那么就不妨指定“开销值”了，可以把数据大小当作“开销值”来用。因为NSData对象的数据大小是已知的，所以计算“开销值”的过程只不过是读取一项属性。

下面这段代码演示了缓存的用法：

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

//Network fetcher class
typedef void(^EOCNetworkFetcherCompletionHandler)(NSData *data);
@interface EOCNetworkFetcher : NSObject
- (id)initWithURL:(NSURL*)url;
- (void)startWithCompletionHandler:
                    (EOCNetworkFetcherCompletionHandler)handler;
@end

//Class that uses the network fetcher and caches results
@interface EOCClass : NSObject
@end

@implementation EOCClass {
    NSCache *_cache;
}

- (id)init {
    if ((self = [super init])) {
        _cache = [NSCache new];

        //Cache a maximum of 100 URLs
        _cache.countLimit = 100;

        /**
         * The size in bytes of data is used as the cost,
         * so this sets a cost limit of 5MB.
         */
        _cache.totalCostLimit = 5 * 1024 * 1024;
    }
    returnself;
}

- (void)downloadDataForURL:(NSURL*)url {
    NSData *cachedData = [_cache objectForKey:url];
    if (cachedData) {
```

```
        //Cache hit
        [self useData:cachedData];
    } else {
        //Cache miss
        EOCNetworkFetcher *fetcher =
            [[EOCNetworkFetcher alloc] initWithURL:url];
        [fetcher startWithCompletionHandler:^(NSData *data){
            [_cache setObject:data forKey:url cost:data.length];
            [self useData:data];
        }];
    }
}
@end
```

在本例中，下载数据所用的 URL，就是缓存的键。若缓存未命中（cache miss）[⊖]，则下载数据并将其放入缓存。而数据的“开销值”则设为其长度。创建 NSCache 时，将其中可缓存的总对象数目上限设为 100，将“总开销”上限设为 5MB，不过，由于“开销值”以“字节”为单位，所以要通过算式将 MB 换算成字节。

还有个类叫做 NSPurgeableData，和 NSCache 搭配起来用，效果很好，此类是 NSMutableData 的子类，而且实现了 NSDiscardableContent 协议。如果某个对象所占的内存能够根据需要随时丢弃，那么就可以实现该协议所定义的接口。这就是说，当系统资源紧张时，可以把保存 NSPurgeableData 对象的那块内存释放掉。NSDiscardableContent 协议里定义了名为 isContentDiscarded 的方法，可用来查询相关内存是否已释放。

如果需要访问某个 NSPurgeableData 对象，可以调用其 beginContentAccess 方法，告诉它现在还不应丢弃自己所占据的内存。用完之后，调用 endContentAccess 方法，告诉它在必要时可以丢弃自己所占据的内存了。这些调用可以嵌套，所以说，它们就像递增与递减引用计数所用的方法那样。只有对象的“引用计数”为 0 时才可以丢弃。

如果将 NSPurgeableData 对象加入 NSCache，那么当该对象为系统所丢弃时，也会自动从缓存中移除。通过 NSCache 的 evictsObjectsWithDiscardedContent 属性，可以开启或关闭此功能。

刚才那个例子可用 NSPurgeableData 改写如下：

```
- (void)downloadDataForURL:(NSURL*)url {
    NSPurgeableData *cachedData = [_cache objectForKey:url];
    if (cachedData) {
        //Stop the data being purged
        [cacheData beginContentAccess];

        //Use the cached data
        [self useData:cachedData];

        //Mark that the data may be purged again
```

⊖ 意思是指缓存中没有访问者所需的数据。——译者注

```
        [cacheData endContentAccess];
    } else {
        //Cache miss
        EOCNetworkFetcher *fetcher =
            [[EOCNetworkFetcher alloc] initWithURL:url];
        [fetcher startWithCompletionHandler:^(NSData *data){
            NSPurgeableData *purgeableData =
                [NSPurgeableData dataWithData:data];
            [_cache setObject:purgeableData
                       forKey:url
                         cost:purgeableData.length];

            //Don't need to beginContentAccess as it begins
            //with access already marked

            //Use the retrieved data
            [self useData:data];

            //Mark that the data may be purged now
            [purgeableData endContentAccess];
        }];
    }
}
```

注意，创建好 NSPurgeableData 对象之后，其“purge 引用计数”会多 1，所以无须再调用 beginContentAccess 了，然而其后必须调用 endContentAccess，将多出来的这个“1”抵消掉。

要点

- 实现缓存时应选用 NSCache 而非 NSDictionary 对象。因为 NSCache 可以提供优雅的自动删减功能，而且是“线程安全的”，此外，它与字典不同，并不会拷贝键。
- 可以给 NSCache 对象设置上限，用以限制缓存中的对象总个数及“总成本”，而这些尺度则定义了缓存删减其中对象的时机。但是绝对不要把这些尺度当成可靠的“硬限制”(hard limit)，它们仅对 NSCache 起指导作用。
- 将 NSPurgeableData 与 NSCache 搭配使用，可实现自动清除数据的功能，也就是说，当 NSPurgeableData 对象所占内存为系统所丢弃时，该对象自身也会从缓存中移除。
- 如果缓存使用得当，那么应用程序的响应速度就能提高。只有那种“重新计算起来很费事的”数据，才值得放入缓存，比如那些需要从网络获取或从磁盘读取的数据。

第 51 条：精简 initialize 与 load 的实现代码

有时候，类必须先执行某些初始化操作，然后才能正常使用。在 Objective-C 中，绝大多数类都继承自 NSObject 这个根类，而该类有两个方法，可用来实现这种初始化操作。

首先要讲的是 load 方法，其原型如下：

```
+ (void)load
```

对于加入运行期系统中的每个类（class）及分类（category）来说，必定会调用此方法，而且仅调用一次。当包含类或分类的程序库载入系统时，就会执行此方法，而这通常就是指应用程序启动的时候，若程序是为 iOS 平台设计的，则肯定会在此时执行。Mac OS X 应用程序更自由一些，它们可以使用“动态加载”（dynamic loading）之类的特性，等应用程序启动好之后再去加载程序库。如果分类和其所属的类都定义了 load 方法，则先调用类里的，再调用分类里的。

load 方法的问题在于，执行该方法时，运行期系统处于“脆弱状态”（fragile state）。在执行子类的 load 方法之前，必定会先执行所有超类的 load 方法，而如果代码还依赖了其他程序库，那么程序库里相关类的 load 方法也必定会先执行。然而，根据某个给定的程序库，却无法判断出其中各个类的载入顺序。因此，在 load 方法中使用其他类是不安全的。比方说，有下面这段代码：

```
#import <Foundation/Foundation.h>
#import "EOCClassA.h"//< From the same library

@interface EOCClassB : NSObject
@end

@implementation EOCClassB
+ (void)load {
    NSLog(@"Loading EOCClassB");
    EOCClassA *object = [EOCClassAnew];
    //Use 'object'
}
@end
```

此处使用 NSLog 没问题，而且相关字符串也会照常记录，因为 Foundation 框架肯定在运行 load 方法之前就已经载入系统了。但是，在 EOCClassB 的 load 方法里使用 EOCClassA 却不太安全，因为无法确定在执行 EOCClassB 的 load 方法之前，EOCClassA 是不是已经加载好了。可以想见：EOCClassA 这个类，也许会在其 load 方法中执行某些重要操作，只有执行完这些操作之后，该类实例才能正常使用。

有个重要的事情需注意，那就是 load 方法并不像普通的方法那样，它并不遵从那套继承规则。如果某个类本身没实现 load 方法，那么不管其各级超类是否实现此方法，系统都不会调用。此外，分类和其所属的类里，都可能出现 load 方法。此时两种实现代码都会调用，类的实现要比分类的实现先执行。

而且 load 方法务必实现得精简一些，也就是要尽量减少其所执行的操作，因为整个应用程序在执行 load 方法时都会阻塞。如果 load 方法中包含繁杂的代码，那么应用程序在执行期间就会变得无响应。不要在里面等待锁，也不要调用可能会加锁的方法。总之，能不做的事情就别做。实际上，凡是想通过 load 在类加载之前执行某些任务的，基本都做得不太对。其真正用途仅在于调试程序，比如可以在分类里编写此方法，用来判断该分类是否已经正确

载入系统中。也许此方法一度很有用处，但现在完全可以说：时下编写 Objective-C 代码时，不需要用它。

想执行与类相关的初始化操作，还有个办法，就是覆写下列方法：

```
+ (void)initialize
```

对于每个类来说，该方法会在程序首次用该类之前调用，且只调用一次。它是由运行期系统来调用的，绝不应该通过代码直接调用。其虽与 load 相似，但却有几个非常重要的微妙区别。首先，它是“惰性调用的”，也就是说，只有当程序用到了相关的类时，才会调用。因此，如果某个类一直都没有使用，那么其 initialize 方法就一直不会运行。这也就等于说，应用程序无须先把每个类的 initialize 都执行一遍，这与 load 方法不同，对于 load 来说，应用程序必须阻塞并等着所有类的 load 都执行完，才能继续。

此方法与 load 还有个区别，就是运行期系统在执行该方法时，是处于正常状态的，因此，从运行期系统完整度上来讲，此时可以安全使用并调用任意类中的任意方法。而且，运行期系统也能确保 initialize 方法一定会在“线程安全的环境”（thread-safe environment）中执行，这就是说，只有执行 initialize 的那个线程可以操作类或类实例。其他线程都要先阻塞，等着 initialize 执行完。

最后一个区别是：initialize 方法与其他消息一样，如果某个类未实现它，而其超类实现了，那么就会运行超类的实现代码。这听起来并不稀奇，但却经常为开发者所忽视。比方说有下面这两个类：

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

@interface EOCBaseClass : NSObject
@end

@implementation EOCBaseClass
+ (void)initialize {
    NSLog(@"%@ initialize", self);
}
@end

@interface EOCSubClass : EOCBaseClass
@end

@implementation EOCSubClass
@end
```

即便 EOCSubClass 类没有实现 initialize 方法，它也会收到这条消息。由各级超类所实现的 initialize 也会先行调用。所以，首次使用 EOCSubClass 时，控制台会输出如下消息：

```
EOCBaseClass initialize
EOCSubClass initialize
```

你可能认为输出的内容有些奇怪，不过这完全符合规则。与其他方法（除去 load）一样，initialize 也遵循通常的继承规则，所以，当初始化基类 EOCBaseClass 时，EOCBaseClass 中定义的 initialize 方法要运行一遍，而当初始化子类 EOCSubClass 时，由于该类并未覆写此方法，因而还要把父类的实现代码再运行一遍。鉴于此，通常都会这么来实现 initialize 方法：

```
+ (void)initialize {
    if (self == [EOCBaseClassclass]) {
        NSLog(@"%@ initialized", self);
    }
}
```

加上这条检测语句之后，只有当开发者所期望的那个类载入系统时，才会执行相关的初始化操作。如果把刚才的例子照此改写，那就不会打印出两条记录信息了，这次只输出一条：

```
EOCBaseClass initialize
```

看过 load 与 initialize 方法的这些特性之后，又回到了早前提过的那个主要问题上，也就是这两个方法的实现代码要尽量精简。在里面设置一些状态，使本类能够正常运作就可以了，不要执行那种耗时太久或需要加锁的任务。对于 load 方法来说，其原因已在前面解释过了，而 initialize 方法要保持精简的原因，也与之相似。首先，大家都不想看到应用程序“挂起”（hang）。对于某个类来说，任何线程都可能成为初次用到它的那个线程，并导致其初始化。如果这个线程碰巧是 UI 线程，那么初始化期间就会一直阻塞，导致应用程序无响应。有时很难预测到底哪个线程会先用到这个类，强令某线程去初始化该类，显然不是好办法。

其二，开发者无法控制类的初始化时机。类在首次使用之前，肯定要初始化，但编写程序时不能令代码依赖特定的时间点，否则会很危险。运行期系统将来更新了之后，可能会略微改变类的初始化方式，这样的话，开发者原来如果假设某个类必定会在某个具体时间点初始化，那么现在这条假设可能就不成立了。

最后一个原因是，如果某个类的实现代码很复杂，那么其中可能会直接或间接用到其他类。若那些类尚未初始化，则系统会迫使其初始化。然而，本类的初始化方法此时尚未运行完毕。其他类在运行其 initialize 方法时，有可能会依赖本类中的某些数据，而这些数据此时也许还未初始化好。例如：

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

static id EOCClassAInternalData;
@interface EOCClassA : NSObject
@end

staticid EOCClassBInternalData;
@interface EOCClassB : NSObject
@end

@implementation EOCClassA

+ (void)initialize {
    if (self == [EOCClassAclass]) {
        [EOCClassBdoSomethingThatUsesItsInternalData];
        EOCClassAInternalData = [self setupInternalData];
    }
}

@end
```

```
@implementation EOCClassB

+ (void)initialize {
    if (self == [EOCClassBclass]) {
        [EOCClassAdoSomethingThatUsesItsInternalData];
        EOCClassBInternalData = [self setupInternalData];
    }
}
@end
```

若是 EOCClassA 先初始化，那么 EOCClassB 随后也会初始化，它会在自己的初始化方法中调用 EOCClassA 的 doSomethingThatUsesItsInternalData，而此时 EOCClassA 内部的数据还没准备好。在实际编码工作中，问题不可能像此处说的那样明显，而且牵涉的类可能也不止两个。因此，当代码无法正常运行时，想要找出错误就更难了。

所以说，initialize 方法只应该用来设置内部数据。不应该在其中调用其他方法，即便是本类自己的方法，也最好别调用。因为稍后可能还要给那些方法里添加更多功能，如果在初始化过程中调用它们，那么还是有可能导致刚才说的那个问题。若某个全局状态无法在编译期初始化，则可以放在 initialize 里来做。下列代码演示了这种用法：

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

@interface EOCClass : NSObject
@end

//EOCClass.m
#import "EOCClass.h"

static const int kInterval = 10;
static NSMutableArray *kSomeObjects;

@implementation EOCClass

+ (void)initialize {
    if (self == [EOCClassclass]) {
        kSomeObjects = [NSMutableArraynew];
    }
}

@end
```

整数可以在编译期定义，然而可变数组不行，因为它是个 Objective-C 对象，所以创建实例之前必须先激活运行期系统。注意，某些 Objective-C 对象也可以在编译期创建，例如 NSString 实例。然而，创建下面这种对象会令编译器报错：

```
static NSMutableArray *kSomeObjects = [NSMutableArray new];
```

编写 load 或 initialize 方法时，一定要留心这些注意事项。把代码实现得简单一些，能节省很多调试时间。除了初始化全局状态之外，如果还有其他事情要做，那么可以专门创建一

个方法来执行这些操作，并要求该类的使用者必须在使用本类之前调用此方法。比如说，如果“单例类”（singleton class）在首次使用之前必须执行一些操作，那就可以采用这个办法。

要点

- 在加载阶段，如果类实现了 load 方法，那么系统就会调用它。分类里也可以定义此方法，类的 load 方法要比分类中的先调用。与其他方法不同，load 方法不参与覆写机制。
- 首次使用某个类之前，系统会向其发送 initialize 消息。由于此方法遵从普通的覆写规则，所以通常应该在里面判断当前要初始化的是哪个类。
- load 与 initialize 方法都应该实现得精简一些，这有助于保持应用程序的响应能力，也能减少引入“依赖环”（interdependency cycle）[⊖] 的几率。
- 无法在编译期设定的全局常量，可以放在 initialize 方法里初始化。

第 52 条：别忘了 NSTimer 会保留其目标对象

计时器是一种很方便也很有用的对象。Foundation 框架中有个类叫做 NSTimer，开发者可以指定绝对的日期与时间，以便到时执行任务，也可以指定执行任务的相对延迟时间。计时器还可以重复运行任务，有个与之相关联的“间隔值”（interval）可用来指定任务的触发频率。比方说，可以每 5 秒轮询某个资源。

计时器要和“运行循环”（run loop）相关联，运行循环到时候会触发任务。创建 NSTimer 时，可以将其“预先安排”在当前的运行循环中，也可以先创建好，然后由开发者自己来调度。无论采用哪种方式，只有把计时器放在运行循环里，它才能正常触发任务。例如，下面这个方法可以创建计时器，并将其预先安排在当前运行循环中：

```
+ (NSTimer *)scheduledTimerWithTimeInterval:
                              (NSTimeInterval)seconds
                                    target:(id)target
                                  selector:(SEL)selector
                                  userInfo:(id)userInfo
                                   repeats:(BOOL)repeats
```

用此方法创建出来的计时器，会在指定的间隔时间之后执行任务。也可以令其反复执行任务，直到开发者稍后将其手动关闭为止。target 与 selector 参数表示计时器将在哪个对象上调用哪个方法。计时器会保留其目标对象，等到自身“失效”时再释放此对象。调用 invalidate 方法可令计时器失效；执行完相关任务之后，一次性的计时器也会失效。开发者若将计时器设置成重复执行模式，那么必须自己调用 invalidate 方法，才能令其停止。

由于计时器会保留其目标对象，所以反复执行任务通常会导致应用程序出问题。也就是说，设置成重复执行模式的那种计时器，很容易引入“保留环”。要想知道其中缘由，请看下列代码：

⊖ 也称“环状依赖”。——译者注

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

@interface EOCClass : NSObject
- (void)startPolling;
- (void)stopPolling;
@end

@implementation EOCClass {
    NSTimer *_pollTimer;
}

- (id)init {
    return [super init];
}

- (void)dealloc {
    [_pollTimer invalidate];
}

- (void)stopPolling {
    [_pollTimer invalidate];
    _pollTimer = nil;
}

- (void)startPolling {
    _pollTimer =
    [NSTimerscheduledTimerWithTimeInterval:5.0
                                    target:self
                                  selector:@selector(p_doPoll)
                                  userInfo:nil
                                   repeats:YES];
}

- (void)p_doPoll {
// Poll the resource
}

@end
```

能看出问题吗？如果创建了本类的实例，并调用其 startPolling 方法，那会如何呢？创建计时器的时候，由于目标对象是 self，所以要保留此实例。然而，因为计时器是用实例变量存放的，所以实例也保留了计时器。（回想一下，第 30 条说过，在 ARC 环境中，这种情况将执行保留操作。）于是，就产生了“保留环”，如果此环能在某一时刻打破，那就不会出什么问题。然而要想打破保留环，只能改变实例变量或令计时器无效。所以说，要么调用 stopPolling，要么令系统将此实例回收，只有这样才能打破保留环。除非使用该类的所有代码均在你的掌控之中，否则无法确保 stopPolling 一定会调用。而且即便能满足此条件，这种通过调用某方法来避免内存泄漏的做法，也不是个好主意。另外，如果想在系统回收本类实例的过程中令计时器无效，从而打破保留环，那又会陷入死结。因为在计时器对象尚且有效

时，EOCClass 实例的保留计数绝不会降为 0，因此系统也绝不会将其回收。而现在又没人来调用 invalidate 方法，所以计时器将一直处于有效状态。图 7-1 演示了此情况。

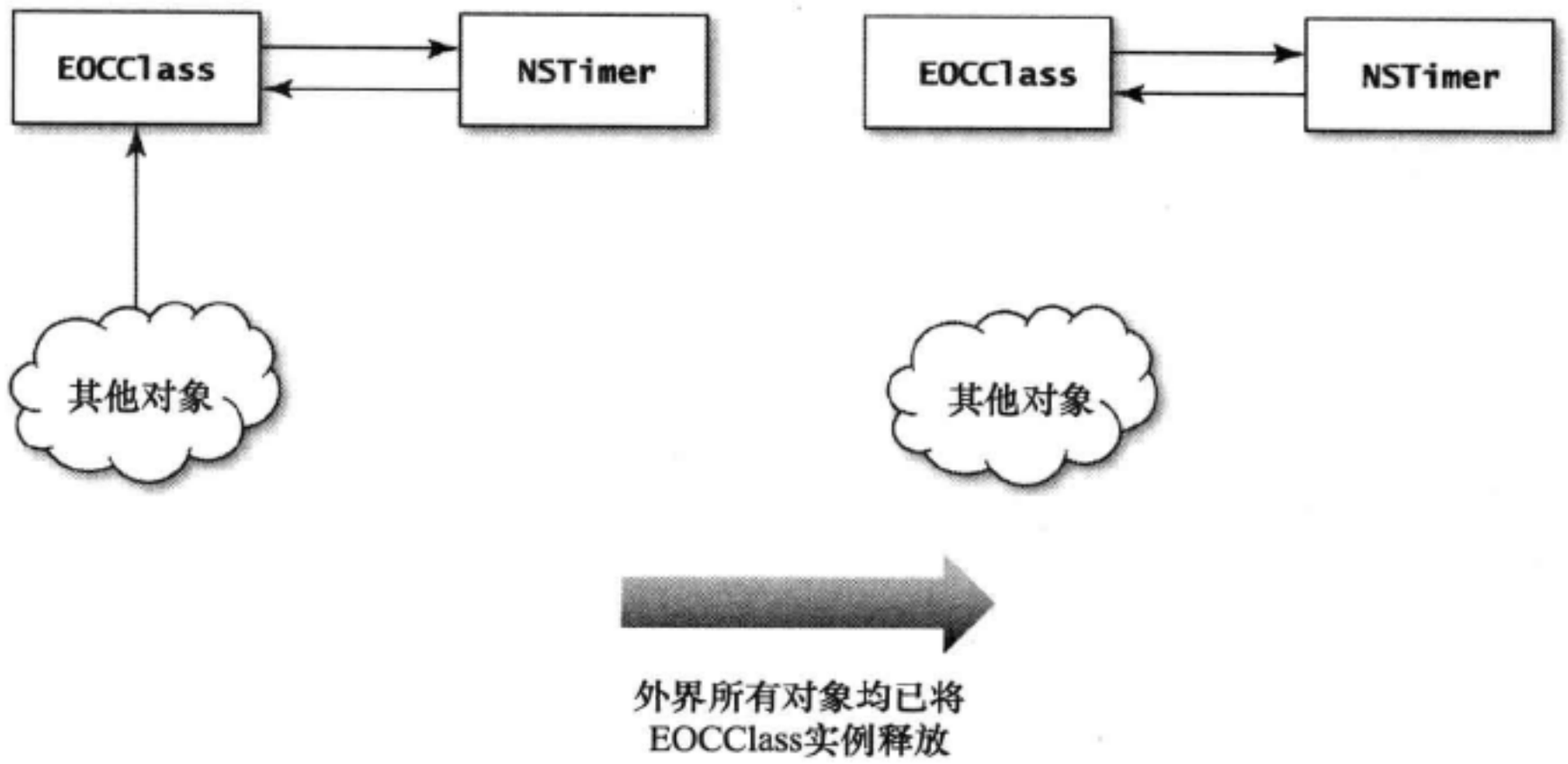

图 7-1　由于计时器保留其目标对象，而对象又保留计时器，所以导致保留环

当指向 EOCClass 实例的最后一个外部引用移走之后，该实例仍然会继续存活，因为计时器还保留着它。而计时器对象也不可能为系统所释放，因为实例中还有个强引用正在指向它。更糟糕的是：除了计时器之外，已经没有别的引用再指向这个实例了，于是该实例就永远“丢失”了。而除了该实例之外，又没有其他引用指向计时器。于是，内存就泄漏了。这种内存泄漏问题尤为严重，因为计时器还将继续反复地执行轮询任务。要是每次轮询时都得联网下载数据的话，那么程序就会一直下载数据，这又更容易导致其他内存泄漏问题。

单从计时器本身入手，很难解决这个问题。可以要求外界对象在释放最后一个指向本实例的引用之前，必须先调用 stopPolling 方法。然而这种情况无法通过代码检测出来，此外，假如该类随着某套公开的 API 对外发布给其他开发者，那么无法保证他们一定会调用此方法。

这个问题可通过“块”来解决。虽然计时器当前并不直接支持块，但是可以用下面这段代码为其添加此功能：

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

@interface NSTimer (EOCBlocksSupport)

+ (NSTimer*)eoc_scheduledTimerWithTimeInterval:
                              (NSTimeInterval)interval
                                       block:(void(^)())block
                                     repeats:(BOOL)repeats;

@end

@implementation NSTimer (EOCBlocksSupport)

+ (NSTimer*)eoc_scheduledTimerWithTimeInterval:
```

```
                                    (NSTimeInterval)interval
                                              block:(void(^)())block
                                            repeats:(BOOL)repeats
{
    return [self scheduledTimerWithTimeInterval:interval
                                         target:self
                                       selector:@selector(eoc_blockInvoke:)
                                       userInfo:[block copy]
                                        repeats:repeats];
}

+ (void)eoc_blockInvoke:(NSTimer*)timer {
    void (^block)() = timer.userInfo;
    if (block) {
        block();
    }
}

@end
```

这个办法为何能解决“保留环”问题呢？大家马上就会明白。这段代码将计时器所应执行的任务封装成“块”，在调用计时器函数时，把它作为 userInfo 参数传进去。该参数可用来存放“不透明值”(opaque value)[⊖]，只要计时器还有效，就会一直保留着它。传入参数时要通过 copy 方法将 block 拷贝到“堆”上（参见第 37 条），否则等到稍后要执行它的时候，该块可能已经无效了。计时器现在的 target 是 NSTimer 类对象，这是个单例，因此计时器是否会保留它，其实都无所谓。此处依然有保留环，然而因为类对象（class object）无须回收，所以不用担心。

这套方案本身并不能解决问题，但它提供了解决问题所需的工具。修改刚才那段有问题的范例代码，使用新分类中的 eoc_scheduledTimerWithTimeInterval 方法来创建计时器：

```
- (void)startPolling {
    _pollTimer =
    [NSTimereoc_scheduledTimerWithTimeInterval:5.0
                                         block:^{
                                             [self p_doPoll];
                                         }
                                       repeats:YES];
}
```

仔细看看代码，就会发现还是有保留环。因为块捕获了 self 变量，所以块要保留实例。而计时器又通过 userInfo 参数保留了块。最后，实例本身还要保留计时器。不过，只要改用 weak 引用（参见第 33 条），即可打破保留环：

```
- (void)startPolling {
    __weak EOCClass *weakSelf = self;
```

⊖ 不指明具体用途的值，可理解为“万能值”。——译者注

```
    _pollTimer =
    [NSTimer eoc_scheduledTimerWithTimeInterval:5.0
                                          block:^{
                                  EOCClass *strongSelf = weakSelf;
                                  [strongSelf p_doPoll];
                                              }
                                        repeats:YES];
}
```

这段代码采用了一种很有效的写法，它先定义了一个弱引用，令其指向 self，然后使块捕获这个引用，而不直接去捕获普通的 self 变量。也就是说，self 不会为计时器所保留。当块开始执行时，立刻生成 strong 引用，以保证实例在执行期间持续存活。

采用这种写法之后，如果外界指向 EOCClass 实例的最后一个引用将其释放，则该实例就可为系统所回收了。回收过程中还会调用计时器的 invalidate 方法（请读者参看原来那段范例代码），这样的话，计时器就不会再执行任务了。此处使用 weak 引用还能令程序更加安全，因为有时开发者可能在编写 dealloc 时忘了调用计时器的 invalidate 方法，从而导致计时器再次运行，若发生此类情况，则块里的 weakSelf 会变成 nil。

要点

- NSTimer 对象会保留其目标，直到计时器本身失效为止，调用 invalidate 方法可令计时器失效，另外，一次性的计时器在触发完任务之后也会失效。
- 反复执行任务的计时器（repeating timer），很容易引入保留环，如果这种计时器的目标对象又保留了计时器本身，那肯定会导致保留环。这种环状保留关系，可能是直接发生的，也可能是通过对象图里的其他对象间接发生的。
- 可以扩充 NSTimer 的功能，用“块”来打破保留环。不过，除非 NSTimer 将来在公共接口里提供此功能，否则必须创建分类，将相关实现代码加入其中。