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

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

版权所有 · 侵权必究
封底无防伪标均为盗版
本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2013- 4807

Authorized translation from the English language edition, entitled *Effective Objective-C 2.0: 52 Specific Ways to Improve Your iOS and OS X Programs*, 9780321917010 by Matt Galloway, published by Pearson Education, Inc., Copyright © 2013.

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

Chinese Simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2014.

本书中文简体字版由 Pearson Education（培生教育出版集团）授权机械工业出版社在中华人民共和国境内（不包括中国台湾地区和中国香港、澳门特别行政区）独家出版发行。未经出版者书面许可，不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

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。

图書馆

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，

试读结束：需要全本请在线购买： www.ertongbook.com

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

译者序

前言

致谢

第 1 章　熟悉 Objective-C …… 1

第 1 条：了解 Objective-C 语言的起源 …… 1

第 2 条：在类的头文件中尽量少引入其他头文件 …… 4

第 3 条：多用字面量语法，少用与之等价的方法 …… 7

第 4 条：多用类型常量，少用 #define 预处理指令 …… 11

第 5 条：用枚举表示状态、选项、状态码 …… 14

第 2 章　对象、消息、运行期 …… 21

第 6 条：理解“属性”这一概念 …… 21

第 7 条：在对象内部尽量直接访问实例变量 …… 28

第 8 条：理解“对象等同性”这一概念 …… 30

第 9 条：以“类族模式”隐藏实现细节 …… 35

第 10 条：在既有类中使用关联对象存放自定义数据 …… 39

第 11 条：理解 objc_msgSend 的作用 …… 42

第 12 条：理解消息转发机制 …… 46

第 13 条：用“方法调配技术”调试“黑盒方法” …… 52

第 14 条：理解“类对象”的用意 …… 56

第 3 章　接口与 API 设计 …… 60

第 15 条：用前缀避免命名空间冲突 …… 60

第 16 条：提供“全能初始化方法” …… 64

第 17 条：实现 description 方法 …… 69

第 18 条：尽量使用不可变对象 …… 73

第 19 条：使用清晰而协调的命名方式 …… 78

第 20 条：为私有方法名加前缀 …… 83

第 21 条：理解 Objective-C 错误模型 …… 85

第 22 条：理解 NSCopying 协议 …… 89

第 4 章　协议与分类 …… 94

第 23 条：通过委托与数据源协议进行对象间通信 …… 94

第 24 条：将类的实现代码分散到便于管理的数个分类之中 …… 101

第 25 条：总是为第三方类的分类名称加前缀 ········· 104
第 26 条：勿在分类中声明属性 ········· 106
第 27 条：使用“class-continuation 分类”隐藏实现细节 ········· 108
第 28 条：通过协议提供匿名对象 ········· 114

第 5 章　内存管理 ········· 117

第 29 条：理解引用计数 ········· 117
第 30 条：以 ARC 简化引用计数 ········· 122
第 31 条：在 dealloc 方法中只释放引用并解除监听 ········· 130
第 32 条：编写“异常安全代码”时留意内存管理问题 ········· 132
第 33 条：以弱引用避免保留环 ········· 134
第 34 条：以“自动释放池块”降低内存峰值 ········· 137
第 35 条：用“僵尸对象”调试内存管理问题 ········· 141
第 36 条：不要使用 retainCount ········· 146

第 6 章　块与大中枢派发 ········· 149

第 37 条：理解“块”这一概念 ········· 149
第 38 条：为常用的块类型创建 typedef ········· 154
第 39 条：用 handler 块降低代码分散程度 ········· 156
第 40 条：用块引用其所属对象时不要出现保留环 ········· 162
第 41 条：多用派发队列，少用同步锁 ········· 165
第 42 条：多用 GCD，少用 performSelector 系列方法 ········· 169
第 43 条：掌握 GCD 及操作队列的使用时机 ········· 173
第 44 条：通过 Dispatch Group 机制，根据系统资源状况来执行任务 ········· 175
第 45 条：使用 dispatch_once 来执行只需运行一次的线程安全代码 ········· 179
第 46 条：不要使用 dispatch_get_current_queue ········· 180

第 7 章　系统框架 ········· 185

第 47 条：熟悉系统框架 ········· 185
第 48 条：多用块枚举，少用 for 循环 ········· 187
第 49 条：对自定义其内存管理语义的 collection 使用无缝桥接 ········· 193
第 50 条：构建缓存时选用 NSCache 而非 NSDictionary ········· 197
第 51 条：精简 initialize 与 load 的实现代码 ········· 200
第 52 条：别忘了 NSTimer 会保留其目标对象 ········· 205

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

试读结束：需要全本请在线购买： www.ertongbook.com

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。

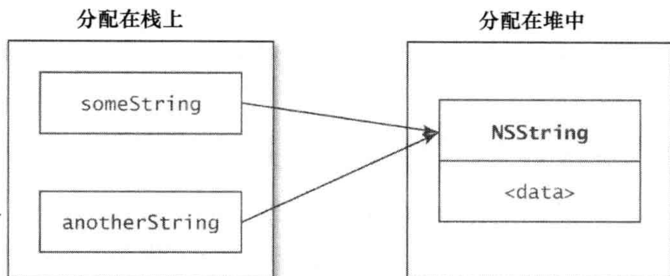

图 1-1 此内存布局图演示了一个分配在堆中的 NSString 实例，有两个分配在栈上的指针指向该实例

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;
```

试读结束：需要全本请在线购买： www.ertongbook.com