Broadview®
www.broadview.com.cn

PEARSON

· 原味精品书系 ·

Effective Objective-C 2.0（英文版）

编写高质量iOS与OS X代码的52个有效方法

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

[美] Matt Galloway 著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

· 原味精品书系 ·

Effective Objective-C 2.0（英文版）

编写高质量iOS与OS X代码的52个有效方法

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

[美] Matt Galloway 著

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

内容简介

本书不是在讲 Objective-C 语言的基础知识，而是要讲如何高效运用这门语言。全书共分 7 章，从 7 个方面总结和探讨了 Objective-C 编程中 52 个鲜为人知又容易被忽视的特性。第 1 章从整体上讲解 Objective-C 的核心概念；第 2 章讲述了与面向对象语言的重要特征（对象、消息和运行期）相关的行为；第 3 章介绍了如何编写适合与 Objective-C 搭配的类；第 4 章讲述协议与分类相关的技巧；第 5 章介绍内存管理的注意事项；第 6 章介绍块与大中枢派发（Grand Central Dispatch）相关的技巧；第 7 章总览了 Cocoa 和 Cocoa Touch 系统框架，并深入研究了其中的某些类。

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 =Effective Objective-C 2.0: 52 Specific Ways to Improve Your iOS and OS X Programs：英文 /（美）加洛韦（Galloway,M.）著. — 北京：电子工业出版社，2016.4
（原味精品书系）
ISBN 978-7-121-27263-9
I. ① E…II. ① 加…III. ① C 语言 – 程序设计 – 英文 IV. ① TP312
中国版本图书馆 CIP 数据核字（2015）第 227473 号

策划编辑：张春雨
责任编辑：徐津平
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980 1/16　　印张：19　　字数：456 千字
版　　次：2016 年 4 月第 1 版
印　　次：2016 年 4 月第 1 次印刷
定　　价：75.00 元
凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。
服务热线：（010）88258888。

献给 Rosie

前言

经常听到有人抱怨 Objective-C 这门语言的冗长、笨拙、别扭，但我却认为它优雅、灵活又漂亮。然而，为了领略这些优点，我们不仅要掌握基础知识，还要理解该语言的特性、陷阱及繁难之处。这正是本书要讲述的内容。

关于本书

本书假定读者已经熟悉了 Objective-C 的语法，所以不再对其赘述。本书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 Objective-C 源自 Smalltalk，所以它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，则要由“运行时”（runtime）执行。于是，某个函数即使在测试环境下能正常运行，到了工作环境中也可能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，本书中的许多话题与 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，后者所用的框架则叫作 Cocoa Touch。

随着 iOS 的兴起，许多开发者都加入了 Objective-C 开发的队伍。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。无论是哪种，为了能高效运用 Objective-C，你都必须花时间研究这门语言，才可能写出更高效、更易于维护、更少 bug 的代码来。

尽管本书的内容只花了 6 个月时间就写好了，但是其酝酿过程却长达数年。几年前的一天，我一时兴起，买了个 iPod Touch；然后等到第一版 SDK 发布之后，我决定试着开发一个程序。我做的第一个“应用程序”叫“Subnet Calc”，其下载量比预想中要多。那时候我感觉到自己以后要和这个美妙的语言结缘了。从此我就一直研究 Objective-C，并定期在自己的网站 www.galloway.me.uk 上发表博客。我对该语言的内部工作原理，如块（block）、自动引用计数（Auto Reference Count，ARC）等特别感兴趣。于是，在有机会写一本讲解 Objective-C 的书时，我自然就抓住了机会。

为提升本书的阅读效果，我鼓励大家跳跃阅读，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。我将各类相关技巧归并成章，所以读者可以根据各章标题快速找到关于某个语言特性的技巧。

本书目标读者

本书面向那些有志于深入研究 Objective-C 的开发者，帮助其编写更便于维护、更高效且更少 bug 的代码。如果你目前还不是 Objective-C 程序员，但是会使用 Java 或 C++ 等其他面向对象的语言，那么你仍可阅读此书，不过你需要先了解一下 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，你可以在许多教材和参考资料中找到这些内容。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有逻辑地组织为如下各章。

第 1 章：熟悉 Objective-C（Accustoming Yourself to Objective-C）

从整体上讲解该语言的核心概念。

第 2 章：对象、消息和运行时（Objects, Messaging, and the Runtime）

面向对象语言的一个重要特征是，对象之间能够关联与交互。本章讲述了这些特征，并深入研究代码在运行时的行为。

第 3 章：接口与 API 设计（Interface and API Design）

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。本章讲解如何编写适合与 Objective-C 搭配的类。

第 4 章：协议与分类（Protocols and Categories）

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（Memory Management）

Objective-C 语言以引用计数来管理内存——许多初学者对此感觉很别扭；如果之前使用的语言以垃圾收集器（garbage collector）来管理内存，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（Blocks and Grand Central Dispatch）

苹果公司引入了“块”（block）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之前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，取决于系统资源状况，这些任务也许能并发执行。本章将教读者如何充分运用这两项核心技术。

第 7 章：系统框架（The System Frameworks）

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，并深入研究其中的某些类。

致谢

在被人问是否愿意写一本关于 Objective-C 的书时，我立刻兴奋了起来。读过了 Effective 系列的其他书籍后，我意识到要想写好这本 Objective-C 的书籍可真是个挑战。然而在众人协助之下，这本书终于和大家见面了。

本书中的很多灵感都源自许多专门论述 Objective-C 的精彩博文。所以首先要感谢博文作者 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发了我的思路。还要感谢苹果公司提供的极为有用的开发文档。

在供职于 MX Telecom 期间，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，他给了我机会，让我以一套成熟的 C++ 代码库为基础，开发出了公司的第一个 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一种帮助。

感谢培生集团的编辑团队，与你们的合作相当愉快。Trina MacDonald、Olivia Basegio、Scott Meyers 及 Chris Zahn 都在需要时给予我帮助与鼓励。诸位为我提供了专心写书的工具，并回答了必要的问题。

还要感谢技术编辑们，与你们的合作也非常融洽，你们给了我莫大的帮助。严格的审校帮助本书内容臻于完美。诸位在检查书稿时认真细致的态度也非常令人称赞。

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 Rosie 降生了！ Helen 与 Rosie 伴我顺利写完这本书，你们俩真棒！

关于作者

Matt Galloway 是英国伦敦的一名 iOS 开发人员。他在 2007 年毕业于剑桥大学彭布鲁克学院，获得工学硕士学位，研究方向是电子信息科学。自那时起，他一直从事编程，主要使用 Objective-C。从 iOS 发布第一个 SDK 开始，他一直在 iOS 上进行开发。他的 Twitter 账号是 @mattjgalloway，常常在 Stack Overflow（http://stackoverflow.com）上回答问题。

目录

1 Accustoming Yourself to Objective-C

Objective-C brings object-oriented features to C through an entirely new syntax. Often described as verbose, Objective-C syntax makes use of a lot of square brackets and isn't shy about using extremely long method names. The resulting source code is very readable but is often difficult for C++ or Java developers to master.

Writing Objective-C can be learned quickly but has many intricacies to be aware of and features that are often overlooked. Similarly, some features are abused or not fully understood, yielding code that is difficult to maintain or to debug. This chapter covers fundamental topics; subsequent chapters cover specific areas of the language and associated frameworks.

Item 1: Familiarize Yourself with Objective-C's Roots

Objective-C is similar to other object-oriented languages, such as C++ and Java, but also differs in many ways. If you have experience in another object-oriented language, you'll understand many of the paradigms and patterns used. However, the syntax may appear alien because it uses a messaging structure rather than function calling. Objective-C evolved from Smalltalk, the origin of messaging. The difference between messaging and function calling looks like this:

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

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

The key difference is that in the messaging structure, the runtime decides which code gets executed. With function calling, the compiler decides which code will be executed. When polymorphism is introduced to the function-calling example, a form of runtime lookup is involved through what is known as a virtual table. But with messaging, the lookup is always at runtime. In fact, the compiler doesn't even care about the type of the object being messaged. That is looked up at runtime as well, through a process known as dynamic binding, covered in more detail in Item 11.

The Objective-C runtime component, rather than the compiler, does most of the heavy lifting. The runtime contains all the data structures and functions that are required for the object-oriented features of Objective-C to work. For example, the runtime includes all the memory-management methods. Essentially, the runtime is the set of code that glues together all your code and comes in the form of a dynamic library to which your code is linked. Thus, whenever the runtime is updated, your application benefits from the performance improvements. A language that does more work at compile time needs to be recompiled to benefit from such performance improvements.

Objective-C is a superset of C, so all the features in the C language are available when writing Objective-C. Therefore, to write effective Objective-C, you need to understand the core concepts of both C and Objective-C. In particular, understanding the memory model of C will help you to understand the memory model of Objective-C and why reference counting works the way it does. This involves understanding that a pointer is used to denote an object in Objective-C. When you declare a variable that is to hold a reference to an object, the syntax looks like this:

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

This syntax, mostly lifted straight from C, declares a variable called `someString` whose type is `NSString*`. This means that it is a pointer to an `NSString`. All Objective-C objects must be declared in this way because the memory for objects is always allocated in heap space and never on the stack. It is illegal to declare a stack-allocated Objective-C object:

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

The `someString` variable points to some memory, allocated in the heap, containing an `NSString` object. This means that creating another variable pointing to the same location does not create a copy but rather yields two variables pointing to the same object:

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

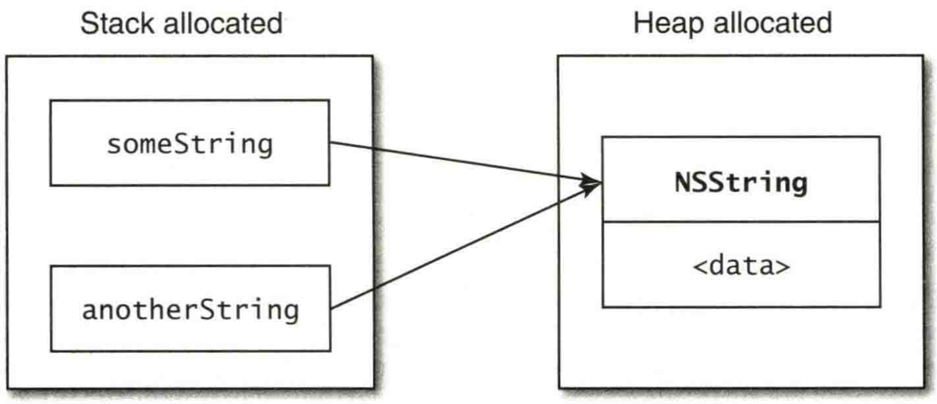

Figure 1.1 Memory layout showing a heap-allocated `NSString` instance and two stack-allocated pointers to it

There is only one `NSString` instance here, but two variables are pointing to the same instance. These two variables are of type `NSString*`, meaning that the current stack frame has allocated 2 bits of memory the size of a pointer (4 bytes for a 32-bit architecture, 8 bytes for a 64-bit architecture). These bits of memory will contain the same value: the memory address of the `NSString` instance.

Figure 1.1 illustrates this layout. The data stored for the `NSString` instance includes the bytes needed to represent the actual string.

The memory allocated in the heap has to be managed directly, whereas the stack-allocated memory to hold the variables is automatically cleaned up when the stack frame on which they are allocated is popped.

Memory management of the heap memory is abstracted away by Objective-C. You do not need to use `malloc` and `free` to allocate and deallocate the memory for objects. The Objective-C runtime abstracts this out of the way through a memory-management architecture known as reference counting (see Item 29).

Sometimes in Objective-C, you will encounter variables that don't have a * in the definition and might use stack space. These variables are not holding Objective-C objects. An example is `CGRect`, from the CoreGraphics framework:

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

A CGRect is a C structure, defined like so:

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

These types of structures are used throughout the system frameworks, where the overhead of using Objective-C objects could affect performance. Creating objects incurs overhead that using structures does not, such as allocating and deallocating heap memory. When nonobject types (int, float, double, char, etc.) are the only data to be held, a structure, such as CGRect, is usually used.

Before embarking on writing anything in Objective-C, I encourage you to read texts about the C language and become familiar with the syntax. If you dive straight into Objective-C, you may find certain parts of the syntax confusing.

Things to Remember

- ✦ Objective-C is a superset of C, adding object-oriented features. Objective-C uses a messaging structure with dynamic binding, meaning that the type of an object is discovered at runtime. The runtime, rather than the compiler, works out what code to run for a given message.
- ✦ Understanding the core concepts of C will help you write effective Objective-C. In particular, you need to understand the memory model and pointers.

Item 2: Minimize Importing Headers in Headers

Objective-C, just like C and C++, makes use of header files and implementation files. When a class is written in Objective-C, the standard approach is to create one of each of these files named after the class, suffixed with .h for the header file and .m for the implementation file. When you create a class, it might end up looking like this:

```
// 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
```

The importing of Foundation.h is required pretty much for all classes you will ever make in Objective-C. Either that, or you will import the base header file for the framework in which the class's superclass lives. For example, if you were creating an iOS application, you would subclass UIViewController often. These classes' header files will import UIKit.h.

As it stands, this class is fine. It imports the entirety of Foundation, but that doesn't matter. Given that this class inherits from a class that's part of Foundation, it's likely that a large proportion of it will be used by consumers of EOCPerson. The same goes for a class that inherits from UIViewController. Its consumers will make use of most of UIKit.

As time goes on, you may create a new class called EOCEmployer. Then you decide that an EOCPerson instance should have one of those. So you go ahead and add a property for it:

```
// 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
```

A problem with this, though, is that the EOCEmployer class is not visible when compiling anything that imports EOCPerson.h. It would be wrong to mandate that anyone importing EOCPerson.h must also import EOCEmployer.h. So the common thing to do is to add the following at the top of EOCPerson.h:

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

This would work, but it's bad practice. To compile anything that uses EOCPerson, you don't need to know the full details about what an EOCEmployer is. All you need to know is that a class called EOCEmployer exists. Fortunately, there is a way to tell the compiler this: