Broadview®
www.broadview.com.cn

objc.io | ObjC 中国

Swift进阶

Advanced Swift

[德] Chris Eidhof　Ole Begemann　Airspeed Velocity　著
王巍　译

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

Swift进阶

Advanced Swift

[德] Chris Eidhof　Ole Begemann　Airspeed Velocity　著
王　巍　译

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

内容简介

本书涵盖了关于 Swift 程序设计的进阶话题。如果你已经通读 Apple 的《Swift 编程指南》，并且想要深入探索关于这门语言的更多内容，那么这本书正适合你！

Swift 非常适合用来进行系统编程，同时它也能被用于书写高层级的代码。我们在书中既会研究像泛型、协议这样的高层级抽象的内容，也会涉足像封装 C 代码以及字符串内部实现这样的低层级话题。本书将帮助你进一步完善知识体系，带领你从 Swift 的入门或中级水平迈入 Swift 高级开发者的大门。

图书在版编目（CIP）数据

Swift 进阶/（德）克里斯 · 安道夫（Chris Eidhof），（德）奥勒 · 毕格曼（Ole Begemann），德国空速网站著；王巍译.—北京：电子工业出版社，2017.5

ISBN 978-7-121-31200-7

I. ①S…II. ①克…②奥…③德…④王…III. ①程序语言－程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字（2017）第 065891 号

策划编辑：张春雨

责任编辑：王　静

印　　刷：三河市良远印务有限公司

装　　订：三河市良远印务有限公司

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：18.75　　字数：410 千字

版　　次：2017 年 5 月第 1 版

印　　次：2017 年 5 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）88254888，88258888。

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。

本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

译者序

我经常会收到 Swift 学习者和使用者发来的电子邮件，问我应该怎么进一步提高自己的 Swift 水平，而在这种时候，我都会向他们推荐您手中的这本书——《Swift 进阶》。

在 2017 年 3 月的 TIOBE 最受欢迎编程语言排行榜中，Swift 首次进入前十名，已经将它的“前辈” Objective-C 远远抛在脑后；而 Swift 3.0 的开源及跨平台，也为这门语言的使用范围和持续发展带来了前所未有的机遇和希望。

在 Swift 高速发展的今天，越来越多的 Swift 开发者不仅仅满足于完成开发任务，他们更希望能知道如何写出优秀的代码，如何优雅高效地实现功能，以及如何更灵活地使用这门语言来应对改变。想要做到这些，我们就需要做到知其然，并知其所以然。《Swift 进阶》正是一本向您介绍 Swift 的种种语言特性“所以然”的书籍。

本书的英文版由 objc.io 的创始人 Chris Eidhof[1]、著名科技编辑和博主 Ole Begemann[2]以及一直关注 Swift 的博客 Airspeed Velocity[3] 的幕后人员一同联合编写。本书原版一经公布，就引起了国外 Swift 社区的极大关注，可以说是国外高级 Swift 开发者几乎人手一本的必读读物。书中深入浅出地剖析了 Swift 里深层次的实现细节以及设计思路。对于包括诸如内建集合类型的底层实现、泛型和协议的设计、Swift 字符串的原理和使用逻辑、值类型和引用类型的适用场景和特点等话题，书中都进行了详细的分析。

本书通过这些细致和系统的解释，为我们揭示了 Swift 的设计哲学，让我们在学习 Swift 的过程中，从“身在此山”变为“高屋建瓴”。虽然在技术精进的道路上没有捷径，但若将前人的经验和总结的精华作为基础，确实能让我们事半功倍。

技术书籍总会面临版本变动和更新的问题。本书的英文原版是在 2015 年 Swift 2 时发布的，其实该书的翻译工作也早在 2015 年年中就完成了。但是在 Swift 3 中，Apple 对这门语言进行了大幅的重塑和调整，本着对读者负责的态度，我们并没有急于推出本书的过时版本，而是在等待 Swift 趋于稳定后，直接以对应最新版本的形式进行发布。在能预见的未来中，

[1] *https://twitter.com/chriseidhof*
[2] *https://twitter.com/olebegemann*
[3] *http://airspeedvelocity.net/*

Swift 4 及后续版本并不会发生像前面版本那样的大规模改动，因此我们认为学习和进阶 Swift 的时机已经成熟。《Swift 进阶》一书在探讨问题时也对版本之间的差异进行了说明，让读者可以了解到技术变革的来龙去脉，并为未来的知识更新提前做好准备。

我们必须承认，在国内当前 Swift 的接受度和使用范围，已经与国外产生了一些差距。由此导致了 Swift 程序开发的平均水平也稍有落后。但我们相信这只是暂时的，随着 Swift 社区的日益强大，国内使用 Swift 的机会和应用场合，都会发生爆发式的增长。让更多的中国开发者有机会接触和了解 Swift 开发更深层次的内容，正是本书目的所在。

王巍

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），您即可享受以下服务。

- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **与作者交流**：在页面下方 读者评论 处留下您的疑问或观点，与作者和其他读者一同学习交流。

页面入口：*http://www.broadview.com.cn/31200*

目录

第1章 介绍

《**Swift 进阶**》对一本书来说是一个很大胆的标题，所以我想我们应该先解释一下它意味着什么。

当我们开始本书第一版的写作时，Swift 才刚刚一岁。我们推测这门语言会在进入第二个年头的时候继续高速地发展，不过尽管我们十分犹豫，但还是决定在 Swift 2.0 测试版发布以前就开始写作。几乎没有别的语言能够在如此短的时间里就能吸引这么多的开发者前来使用。

但是这留给了我们一个问题：如何写出“符合语言习惯”的 Swift 代码？对于某一个任务，有正确的做法吗？标准库给了我们一些提示，但是我们知道，即使是标准库本身也会随时间发生变化，它常常抛弃一切约定，又去遵守另一些约定。不过，在过去两年里，Swift 高速进化着，而优秀的 Swift 代码标准也日益明确。

对于从其他语言迁移过来的开发者，Swift 可能看起来很像你原来使用的语言，特别是它可能拥有你原来使用的语言中你最喜欢的那一部分。它可以像 C 语言一样进行低层级的位操作，但又可以避免许多未定义行为的陷阱。Ruby 的教徒可以在像是 `map` 或 `filter` 的轻量级的尾随闭包中感受到宾至如归。Swift 的泛型和 C++ 的模板如出一辙，但是额外的类型约束能保证泛型方法在被定义时就是正确的，而不必等到使用的时候再进行判定。灵活的高阶函数和运算符重载让你能够以 Haskell 或者 F# 那样的风格进行编码。最后 `@objc` 关键字允许你像在 Objective-C 中那样使用 selector 和各种运行时的动态特性。

有了这些相似点，Swift 可以去适应其他语言的风格。比如，Objective-C 的项目可以自动地导入 Swift 中，很多 Java 或者 C# 的设计模式也可以直接照搬过来使用。在 Swift 发布的前几个月，一大波关于单子 (monad) 的教程和博客也纷至沓来。

但是失望也接踵而至。为什么我们不能像 Java 中的接口那样将协议扩展 (protocol extension) 和关联类型 (associated type) 结合起来使用？为什么数组不具有我们预想那样的协变 (covariant) 特性？为什么我们无法写出一个“函子”(functor)？有时候这些问题的答案是 Swift 还没有来得及实现这部分功能，但是更多时候，这是因为在 Swift 中有其他更适合这门语言的方式来完成这些任务，或者是因为 Swift 中这些你认为等价的特性其实和你原来的想象大有不同。

> 译者注：数组的协变特性指的是，包含有子类型对象的数组，可以直接赋值给包含有父类型对象的数组的变量。比如在 Java 和 C# 中，string 是 object 的子类型，而对应的数组类型 string[] 可以直接赋值给声明为 object[] 类型的变量。但是在 Swift 中，Array<Parent> 和 Array<Child> 之间并没有这样的关系。

和其他大多数编程语言一样，Swift 也是一门复杂的语言。但是它将这些复杂的细节隐藏得很好。你可以使用 Swift 迅速上手开发应用，而不必知晓泛型、重载或者是静态调用和动态派发之间的区别等这些知识。你可能永远都不会需要去调用 C 语言的代码，或者实现自定义的集合类型。但是随着时间的推移，无论是想要提升你的代码的性能，还是想让程序更加优雅清晰，抑或是只是为了完成某项开发任务，你都有可能要逐渐接触到这些事情。

带你深入地学习这些特性就是这本书的写作目的。我们在书中尝试回答了很多“这个要怎么做”以及“为什么在 Swift 中会是这个结果”这样的问题，这种问题遍布各个论坛。我们希望你一旦阅读过本书，就能把握这些语言基础的知识，并且了解很多 Swift 的进阶特性，从而对 Swift 是如何工作的有一个更好的理解。本书中的知识点可以说是一个高级 Swift 程序员所必须了解和熟悉的内容。

1.1 本书所面向的读者

本书面向的是有经验的程序员，你不需要是程序开发的专家，不过你应该已经是 Apple 平台的开发者，或者是想要从其他比如 Java 或者 C++ 这样的语言转行过来的程序员。如果你想要把你的 Swift 相关知识技能提升到和你原来已经熟知的 Objective-C 或者其他语言的同一水平线上，那么这本书会非常适合你。本书也适合那些已经开始学习 Swift，对这门语言基础有一定了解，并且渴望再上一个层次的新程序员们。

这本书不是一本介绍 Swift 基础的书籍，我们假定你已经熟悉这门语言的语法和结构。如果你需要完整地学习 Swift 的基础知识，最好的资源是 Apple 的 Swift 相关书籍 (在 iBooks[1] 以及 Apple 开发者网站[2] 上均可以下载)。如果你很有把握，你可以尝试同时阅读我们的这本书和 Apple 的 Swift 书籍。

这也不是一本教你如何为 macOS 或者 iOS 编程的书籍。不可否认，Swift 现在主要用于 Apple 的平台，我们会尽量包含一些实践中使用的例子，但是我们更希望这本书可以对非 Apple 平台的程序员也有所帮助。

[1] *https://itunes.apple.com/us/book/swift-programming-language/id1002622538*
[2] *https://developer.apple.com/swift/resources/*

1.2 主题

我们按照基本概念的主题来组织本书，其中有一些深入像可选值和字符串这样基本概念的章节，也有对于像 C 语言互用性方面的主题。不过纵观全书，有一些主题可以描绘出 Swift 给人的总体印象：

Swift 既是一门高层级语言，又是一门低层级语言。 你可以在 Swift 中用 `map` 或者 `reduce` 来写出十分类似于 Ruby 和 Python 的代码，你也可以很容易地创建自己的高阶函数。Swift 让你有能力快速完成代码编写，并将它们直接编译为原生的二进制可执行文件，这使得其在性能上可以与用 C 语言编写的程序相媲美。

Swift 真正激动人心，以及令人赞叹的是，我们可以**兼顾**高低两个层级。将一个数组通过闭包表达式映射到另一个数组所编译得到的汇编码，与直接对一块连续内存进行循环所得到的结果是一致的。

不过，为了最大化地利用这些特性，有一些知识你需要掌握。如果你能对结构体和类的区别有深刻理解，或者对动态和静态方法派发的不同了然于胸，那么你就能从中获益。我们会在之后更深入地介绍这些内容。

Swift 是一门多范式的语言。 你可以用 Swift 来编写面向对象的代码，也可以使用不变量的值来写纯函数式的程序，在必要的时候，你甚至还能使用指针运算来写和 C 类似的代码。

这是一把双刃剑。好的一面，在 Swift 中你将有很多可用的工具，你也不会被限制在一种代码写法里。但是这也让你身临险境，因为实际上你可能会变成使用 Swift 语言来书写 Java 或者 C 或者 Objective-C 的代码。

Swift 仍然可以使用大部分 Objective-C 的功能，包括消息发送，运行时的类型判定，以及 KVO 等。但是 Swift 还引入了很多 Objective-C 中不具备的特性。

Erik Meijer 是一位著名的程序语言专家，他在 2015 年 10 月“发推”[1]说道：

> 现在，相比 Haskell，Swift 可能是更好，更有价值，也更合适用来的学习函数式编程的语言。

Swift 拥有泛型、协议、值类型以及闭包等特性，这些特性是对函数式风格的很好的介绍。我们甚至可以将运算符和函数结合起来使用。在 Swift 早期，这门语言为世界带来了很多关于单子 (monad) 的博客。不过等到 Swift 2.0 发布并引入协议扩展的时候，大家研究的趋势也随之发生了变化。

[1] *https://twitter.com/headinthebox/status/655407294969196544*

Swift 十分灵活。在 On Lisp[1] 这本书的介绍中，Paul Graham 写道：

> 富有经验的 Lisp 程序员将他们的程序拆分成不同的部分。除了自上而下的设计原则，他们还遵循一种可以被称为自下而上的设计，他们可以将语言进行改造，让它更适合解决当前的问题。在 Lisp 中，你并不只是使用这门语言来编写程序，在开发过程中，你同时也在构建这门语言。当你编写代码的时候，你可能会想“要是 Lisp 有这个或者这个运算符就好了”，之后你就真的可以去实现一个这样的运算符。事后来看，你会意识到使用新的运算符可以简化程序的某些部分的设计，语言和程序就这样相互影响，发展进化。

Swift 的出现比 Lisp 要晚得多，不过，我们能强烈感受到 Swift 也鼓励从下向上的编程方式。这让我们能轻而易举地编写一些通用可重用组件，然后可以将它们组合起来实现更强大的的特性，最后用它们来解决实际问题。Swift 非常适合用来构建这些组件，你可以使它们看起来就像是语言自身的一部分。一个很好的例子就是 Swift 的标准库，许多你能想到的基本组件——像可选值和基本的运算符等——其实都不是直接在语言本身中定义的，相反，它们是在标准库中被实现的。

Swift 代码可以做到紧凑、精确，同时保持清晰。Swift 使用相对简洁的代码，但这并不意味着其单纯地减少代码的输入量，还标志了一个更深层次的目标。Swift 的观点是，通过抛弃你经常在其他语言中见到的模板代码，而使得代码更容易被理解和阅读。这些模板代码往往会成为理解程序的障碍，而非助力。

举个例子，有了类型推断，在上下文很明显的时候我们就不再需要乱七八糟的类型声明了；那些几乎没有意义的分号和括号也都被移除了；泛型和协议扩展让你免于重复，并且把通用的操作封装到可以复用的方法中。这些特性最终的目的都是为了能够让代码看上去一目了然。

一开始，这可能会对你造成一些困扰。如果你以前从来没有用像是 `map`、`filter` 和 `reduce` 这样的函数，那么它们可能看起来比简单的 `for` 循环要难理解。但是我们相信这个学习过程会很短，并且作为回报，你会发现这样的代码你第一眼看上去就能更准确地判断出它“显然正确”。

除非你有意为之，否则 Swift 在实践中总是安全的。Swift 和 C 或者 C++ 这样的语言不同，在那些语言中，你只要忘了做某件事情，你的代码很可能就不是安全的了。它和 Haskell 或者 Java 也不一样，在后两者中有时候不论你是否需要，它们都“过于”安全。

[1] *http://www.paulgraham.com/onlisp.html*

C# 的主要设计者之一 Eric Lippert 在他关于创造 C# 的 10 件后悔的事情中总结了[1]一些经验教训：

> 有时候你需要为那些构建架构的专家实现一些特性，这些特性应当被清晰地标记为危险——它们往往并不能很好地对应其他语言中某些有用的特性。

说这段话时，Eric 特别所指的是 C# 中的终止方法 (finalizer)，它和 C++ 中的析构函数 (destructor) 比较类似。但是不同于析构函数，终止方法的运行是不确定的，它受命于垃圾回收器，并且运行在垃圾回收的线程上。更糟糕的是，很可能终止方法甚至完全不会被调用。但是，在 Swift 中，因为采用的是引用计数，`deinit` 方法的调用是可以确定和预测的。

Swift 的这个特点在其他方面也有体现。未定义的和不安全的行为默认是被屏蔽的。比如，一个变量在被初始化之前是不能使用的，使用越界下标访问数组将会抛出异常，而不是继续使用一个可能取到的错误值。

当你真正需要的时候，也有不少“不安全”的方式，比如 `unsafeBitcast` 函数，或者是 `UnsafeMutablePointer` 类型。但是强大能力的背后是更大的未定义行为的风险。比如下面的代码：

```
var someArray = [1,2,3]
let uhOh = someArray.withUnsafeBufferPointer { ptr in
    // ptr 只在这个 block 中有效
    // 不过你完全可以将它返回给外部世界：
    return ptr
}
// 稍后...
print(uhOh[10])
```

这段代码可以编译，但是天知道它最后会做什么。方法名里已经警告了你这是不安全的，所以对此你需要自己负责。

Swift 是一门独断的语言。关于“正确的”Swift 编码方法，作为本书作者，我们有着坚定的自己的看法。你会在本书中看到很多这方面的内容，有时候我们会把这些看法作为事实来对待。但是，归根结底，这只是我们的看法，你完全可以反对我们的观点。Swift 还是一门年轻的语言，许多事情还未成定局。更糟糕的是，很多博客或者文章是不正确的，或者已经过时 (包括我们曾经写过的一些内容，特别是早期就完成了的内容)。不论你在读什么资料，最重要的事情是你应当亲自尝试，去检验它们的行为，并且去体会这些用法。带着批判的眼光去审视和思考，并且警惕那些已经过时的信息。

[1] *http://www.informit.com/articles/article.aspx?p=2425867*

1.3 术语

> 你用，或是不用，术语就在那里，不多不少。你懂，或是不懂，定义就在那里，不偏不倚。

程序员总是喜欢说行话[1]。为了避免困扰，接下来我们会介绍一些贯穿于本书的术语定义。我们会尽可能遵守 Swift 官方文档中的术语用法，使用被 Swift 社区所广泛接受的定义。这些定义大多都会在接下来的章节中被详细介绍，所以就算一开始你对它们一头雾水，也大可不必在意。如果你已经对这些术语非常了解，那么我们也还是建议你再浏览一下它们，并且确定你能接受我们的表述。

在 Swift 中，我们需要对值、变量、引用以及常量加以区分。

值 (value) 是不变的，永久的，它从不会改变。比如，`1`、`true` 和 `[1,2,3]` 都是值。这些是**字面量 (literal)** 的例子，值也可以是运行代码时生成的。当你计算 5 的平方时，你得到的数字也是一个值。

当我们使用 `var x = [1,2]` 来将一个值进行命名的时候，我们实际上创建了一个名为 `x` 的**变量 (variable)** 来持有 `[1,2]` 这个值。通过像执行 `x.append(3)` 这样的操作来改变 `x` 时，我们并没有改变原来的值。相反，我们所做的是使用 `[1,2,3]` 这个新的值来替代原来 `x` 中的内容。可能实际上它的内部实现真的只是在某段内存的后面添加上一个条目，并不是进行全体替换，但是至少从**逻辑**上来说此值是全新的。我们将这个过程称为变量的**改变** (mutating)。

我们还可以使用 `let` 而不是 `var` 来声明一个**常量变量 (constant variables)**，或者简称为常量。一旦常量被赋予一个值，它就不能再次被赋一个新的值了。

我们不需要在一个变量被声明的时候就立即为它赋值。我们可以先对变量进行声明 (`let x: Int`)，然后稍后再给它赋值 (`x = 1`)。Swift 是强调安全的语言，它将检查所有可能的代码路径，并确保变量在被读取之前一定是完成了赋值的。在 Swift 中变量不会存在未定义状态。当然，如果一个变量是用 `let` 声明的，那么它只能被赋值一次。

结构体 (struct) 和枚举 (enum) 是**值类型 (value type)**。当你把一个结构体变量赋值给另一个变量，那么这两个变量将会包含同样的值。你可以将它理解为内容被复制了一遍，但是更精确地描述，则是被赋值的变量与另外的那个变量包含了同样的值。

引用 (reference) 是一种特殊类型的值：它是一个“指向”另一个值的值。两个引用可能会指向同一个值，这引入了一种可能性，那就是这个值可能会被程序的两个不同的部分所改变。

[1] *https://zh.wikipedia.org/wiki/行話*