SWIFTER

Swift 开发者必备 Tips

第四版

王巍 (@onevcat)

Table of Contents

介绍

> 虽然我们都希望能尽快开始在 Swift 的世界里遨游，但是我觉得仍然有必要花一些时间将本书的写作目的和适合哪些读者进行必要说明。我不喜欢自吹自擂，也无法承担“骗子”的骂名。在知识这件严肃的事情上，我并不希望对读者产生任何的误导。作为读者，您一定想要找的是一本合适自己的书；而作为作者，我也希望找到自己的伯乐和子期。

为什么要写这本书

中文的科技书籍太少了，内容也太浅了。这是国内市场尴尬的现状：真正有技术的大牛不在少数，但他们很多并不太愿意通过出书的方式来分享他们的知识：一方面是回报率实在太低，另一方面是出版的流程过于繁琐。这就导致了市面上充斥了一些习惯于出版业务，但是却丝毫无视质量和素质的流氓作者和图书。

特别是对于 Swift 语言来说，这个问题尤其严重。iOS 开发不可谓不火热，每天都有大量的开发者涌入这个平台。而 Swift 的发布更使得原本高温的市场更上一层楼。但是市面上随处可见的都是各种《开发指南》《权威指南》或者《21天学会XXX》系列的中文资料。这些图书大致都是对官方文档的翻译，并没有什么实质的见解，可以说内容单一，索然无味。作为读者，很难理解作者写作的重心和目的 (其实说实话，大部分情况下这类书的作者自己都不知道写作的重心和目的是什么)，这样的“为了出版而出版”的图书可以说除了增加世界的熵以外，几乎毫无价值。

如果想要入门 Swift 语言，阅读 Apple 官方教程和文档无论从条理性和权威性来说，都是更好的选择。而中国的 Cocoa 开发者社区也以令人惊叹的速度完成了对文档的高品质翻译，这在其他任何国家都是让人眼红的一件事情。因此，如果您是初学程序设计或者 Swift 语言，相比起那些泯灭良心 (抱歉我用了这个词，希望大家不要对号入座) 的“入门书籍”，我更推荐您看这份翻译后的官方文档，这是非常珍惜和宝贵的资源。

说到这里，可以谈谈这本《Swifter - Swift 必备 tips》的写作目的了。很多 Swift 的学习者 -- 包括新接触 Cocoa/Cocoa Touch 开发的朋友，以及之前就使用 Objective-C 的朋友 -- 所共同面临的一个问题是，入门以后应该如何进一步提高。也许你也有过这样的感受：在阅读完 Apple 的教程后，觉得自己已经学会了 Swift 的语法和使用方式，你满怀信心地打开 Xcode，新建了一个 Swift 项目，想写点什么，却发现实际上满不是那么回事。你需要联想 Optional 应该在什么时候使用，你可能发现本已熟知 API 突然不太确定要怎么表达，你可能遇到怎么也编译不了的问题但却不知如何改正。这些现象都非常正常，因为教程是为了展示某个语法点而写的，而几乎不涉及实际项目中应该如何使用的范例。本书的目的就是为广大已经入门了 Swift 的开发者提供一些参考，以期能迅速提升他们在实践中的能力。因为这部分的中级内容是我自己力所能及，有自信心能写好的；也是现在广大 Swift 学习者所急缺和需要的。

这本书是什么

本书是 Swift 语言的知识点的集合。我自己是赴美参加了 Apple 的 WWDC 14 的，也正是在这届开发者大会上，Swift 横空出世。毫不夸张地说，从 Swift 正式诞生的第一分钟开始，我就在学习这门语言。虽然天资驽钝，不得其所，但是在这段集中学习和实践的时间里，也还算总结了一些心得，而我把这些总结加以整理和示例，以一个个的小技巧和知识点的形式，编写成了这本书。本书中每一节都是一个相对独立的主题，涵盖了一个中高级开发人员需要知道的 Swift 语言的方方面面。

这本书非常适合用作官方文档的参考和补充，也会是中级开发人员很喜爱的 Swift 进阶读本。具体每个章节的内容，可以参看本书的目录。

这本书不是什么

这本书**不是** Swift 的入门教程，也**不会**通过具体的完整实例引导你用 Swift 开发出一个像是计算器或者记事本这样的 app。这本书的目的十分纯粹，就是探索那些不太被人注意，但是又在每天的开发中可能经常用到的 Swift 特性。这本书并不会系统地介绍 Swift 的语法和特性，因为基于本书的写作目的和内容特点，采用松散的模式和非线性的组织方式会更加适合。

换言之，如果你是想找一本 Swift 从零开始的书籍，那这本书不应该是你的选择。你可以在阅读 Apple 文档后再考虑回来看这本书。

组织形式和推荐的阅读方式

本书的每个章节的内容是相对独立的，也就是说你没有必要从头开始看，随手翻开到任何一节都是没问题的。当然，按顺序看会是比较理想的阅读方式，因为在写作时我特别注意了让靠前的章节不涉及后面章节的内容；另一方面，位置靠后的章节如果涉及到之前章节内容的话，我添加了跳转到相关章节的链接，这可以帮助迅速复习和回顾之前的内容。我始终坚信不断的重复和巩固，是真正掌握知识的唯一途径。

本书的电子版的目录是可以点击跳转的，您可以通过目录快速地在不同章节之间导航。如果遇到您不感兴趣或者已经熟知的章节，您也完全可以暂时先跳过去，这不会影响您对本书的阅读和理解。

代码运行环境

建议您一边阅读本书时一边开启 Xcode 环境并且对每一章节中的代码进行验证，这有利于您真正理解代码示例想表达的意思，也有利于记忆的形成。随本书所附的 Playground 文件中有大部分章节的示例代码，以供参考。每一段代码示例都不太长，但却是经过精心准备，能很好地说明章节内容的，希望您能在每一章里都能通过代码和我进行心灵上的“对话”。您也可以在已有的基础上进行自己的探索，用来加深对讨论内容的理解。

书中每一章基本都配有代码示例的说明。这些代码一般来说包括 Objective-C 或者 Swift 的代码。理论上来说所有代码都可以在 Swift 4 (也就是 Xcode 9) 版本环境下运行。当然 因为 Swift 版本变化很快，可能部分代码需要微调或者结合一定的上下文环境才能运行，但我相信这种调整是显而易见的。如果您发现明显的代码错误和无法运行的情况，欢迎到本书的 issue 页面 上提出，我将尽快修正。

如果没有特别说明，这些代码在 Playground 和项目中都应该可以运行，并拥有同样表现的。但是也存在一些代码只能在 Playground **或者**项目文件中才能正确工作的情况，这主要是因为平台限制的因素，如果出现这种情况，我都会在相关章节中特别加以说明。

勘误和反馈

Swift 仍然在高速发展和不断变化中，本书最早版本基于 Swift 1.0，当前版本基于 Swift 4。随着 Swift 的新特性引入以及错误修正，本书难免会存在部分错误，其中包括为对应的更新纰漏或者部分内容过时的情况。虽然我会随着 Swift 的发展继续不断完善和修正这本书，但是这个过程亦需要时间，请您谅解。

另外由于作者水平有限，书中也难免会出现一些错误，如果您在阅读时发现了任何问题，可以到这本书 issue 页面进行反馈。我将尽快确认和修正。得益于电子书的优势，本书的读者都可以在本书更新时免费获得所有的新内容。每次更新的变更内容将会写在本书的更新一节中，您也可以在更新内容页面上找到同样的列表。

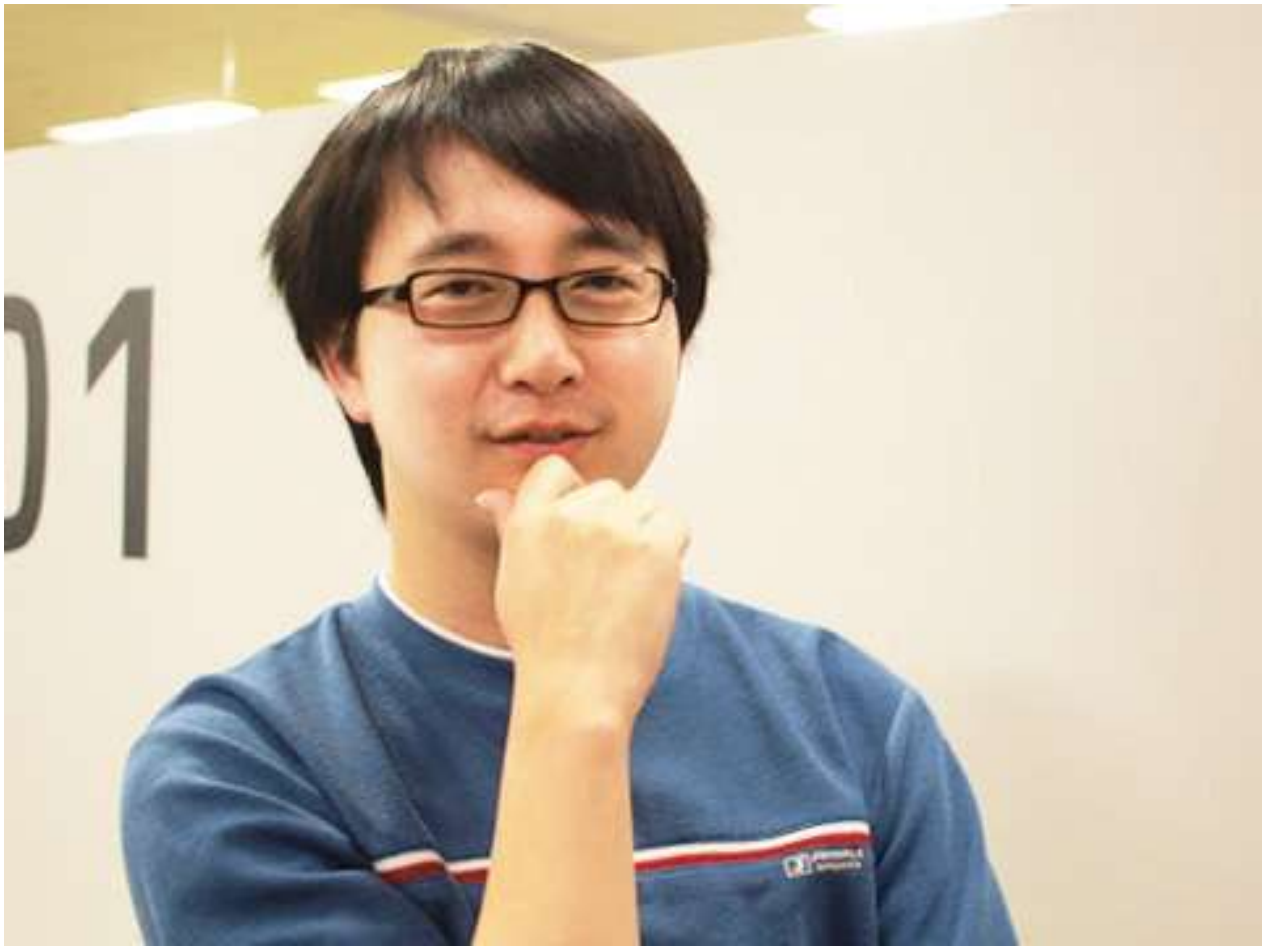
版权问题

为了方便读者使用和良好的阅读体验，本书不包含任何 DRM 保护。首先我在此想感谢您购买了这本书，在国内知识产权保护不足的现状下，我自知出版这样一本没有任何保护措施的电子书可能无异于飞蛾扑火。我其实是怀着忐忑的心情写下这些文字的，小心翼翼地希望没有触动到太多人。如果您不是通过 Gumroad，Leanpub 或者是 SelfStore 购买，而拿到这本书的话，您应该是盗版图书的受害者。这本书所提供的知识和之后的服务我想应该是超过它的售价 (大约是一杯星巴克咖啡的价格) 的，在阅读前还请您再三考虑。您的支持将是我继续更新和完善本书的动力，也对我继续前进是很大的鼓励。

另外，这本书也有纸质版本，不过暂时是面向 Swift 2.0 的。如果您有意阅读，可以搜索“Swifter:100个Swift开发必备Tip 第二版”来获取相关信息。

最后一部分是我的个人简介，您可以跳过不看，而直接开始阅读正文内容。

作者简介



王巍 (onevcat) 是来自中国的一线 iOS 开发者，毕业于清华大学。在校期间就开始进行 iOS 开发，拥有丰富的 Cocoa 和 Objective-C 开发经验，另外他也活跃于使用 C# 的 Unity3D 游戏开发界。曾经开发了《小熊推金币》，《Pomo Do》等一系列优秀的 iOS 游戏和应用。在业余时间，王巍会在 OneV's Den 撰写博客，分享他在开发中的一些心得和体会。另外，王巍还是翻译项目 objc 中国 的组织者和管理者，为中国的 Objective-C 社区的发展做出了贡献。同时，他也很喜欢为开源社区贡献代码，是著名的 Xcode 插件 VVDocumenter 和开源库 Kingfisher 的作者。

现在王巍旅居日本，并就职于即时通讯软件公司 LINE，从事 iOS 开发工作，致力于为全世界带来更好体验和功能的应用。如果您需要进一步了解作者的话，可以访问他的资料页面。

1. Swift 新元素

柯里化 (Currying)

Swift 里可以将方法进行柯里化 (Currying)，这是也就是把接受多个参数的方法进行一些变形，使其更加灵活的方法。函数式的编程思想贯穿于 Swift 中，而函数的柯里化正是这门语言函数式特点的重要表现。

举个例子，下面的函数简单地将输入的数字加 1：

```
func addOne(num: Int) -> Int {
    return num + 1
}
```

这个函数所表达的内容非常有限，如果我们之后还需要一个将输入数字加 2，或者加 3 的函数，可能不得不类似地去定义返回为 `num + 2` 或者 `num + 3` 的版本。有没有更通用的方法呢？我们其实可以定义一个通用的函数，它将接受需要与输入数字相加的数，并返回一个函数。返回的函数将接受输入数字本身，然后进行操作：

```
func addTo(_ adder: Int) -> (Int) -> Int {
    return {
        num in
        return num + adder
    }
}
```

有了 `addTo`，我们现在就能轻易写出像是 `addOne` 或者 `addTwo` 这样的函数了：

```
let addTwo = addTo(2)    // addTwo: Int -> Int
let result = addTwo(6)   // result = 8
```

再举一个例子，我们可以创建一个比较大小的函数：

```
func greaterThan(_ comparer: Int) -> (Int) -> Bool {
    return { $0 > comparer }
}

let greaterThan10 = greaterThan(10);

greaterThan10(13)    // => true
greaterThan10(9)     // => false
```

柯里化是一种量产相似方法的好办法，可以通过柯里化一个方法模板来避免写出很多重复代码，也方便了今后维护。

举一个实际应用时候的例子，在 Selector 一节中，我们提到了在 Swift 中 `Selector` 只能使用字符串在生成。这面临一个很严重的问题，就是难以重构，并且无法在编译期间进行检查，其实这是十分危险的行为。但是 target-action 又是 Cocoa 中如此重要的一种设计模式，无论如何我们都想安全地使用的话，应该怎么办呢？一种可能的解决方式就是利用方法的柯里化。Ole Begemann 在这篇帖子里提到了一种很好封装，这为我们如何借助柯里化，安全地改造和利用 target-action 提供了不少思路。

```
protocol TargetAction {
    func performAction()
}

struct TargetActionWrapper<T: AnyObject>:
                            TargetAction {
    weak var target: T?
    let action: (T) -> () -> ()

    func performAction() -> () {
        if let t = target {
            action(t)()
        }
    }
}

enum ControlEvent {
    case TouchUpInside
    case ValueChanged
    // ...
}


class Control {
    var actions = [ControlEvent: TargetAction]()

    func setTarget<T: AnyObject>(target: T,
                   action: @escaping (T) -> () -> (),
                controlEvent: ControlEvent) {

        actions[controlEvent] = TargetActionWrapper(
            target: target, action: action)
    }

    func removeTargetForControlEvent(controlEvent: ControlEvent) {
        actions[controlEvent] = nil
    }

    func performActionForControlEvent(controlEvent: ControlEvent) {
        actions[controlEvent]?.performAction()
    }
}
```

模式匹配

在之前的正则表达式中，我们实现了 `=~` 操作符来完成简单的正则匹配。虽然在 Swift 中没有内置的正则表达式支持，但是一个和正则匹配有些相似的特性其实是内置于 Swift 中的，那就是模式匹配。

当然，从概念上来说正则匹配只是模式匹配的一个子集，但是在 Swift 里现在的模式匹配还很初级，也很简单，只能支持最简单的相等匹配和范围匹配。在 Swift 中，使用 `~=` 来表示模式匹配的操作符。如果我们看看 API 的话，可以看到这个操作符有下面几种版本：

```
func ~=<T : Equatable>(a: T, b: T) -> Bool

func ~=<T>(lhs: _OptionalNilComparisonType, rhs: T?) -> Bool

func ~=<I : IntervalType>(pattern: I, value: I.Bound) -> Bool
```

从上至下在操作符左右两边分别接收可以判等的类型，可以与 `nil` 比较的类型，以及一个范围输入和某个特定值，返回值很明了，都是是否匹配成功的 `Bool` 值。你是否有想起些什么呢..没错，就是 Swift 中非常强大的 switch，我们来看看 switch 的几种常见用法吧：

1. 可以判等的类型的判断

```
let password = "akfuv(3"
switch password {
    case "akfuv(3": print("密码通过")
    default:        print("验证失败")
}
```

2. 对 Optional 的判断

```
let num: Int? = nil
switch num {
    case nil: print("没值")
    default:  print("\(num!)")
}
```

3. 对范围的判断

```
let x = 0.5
switch x {
    case -1.0...1.0: print("区间内")
    default: print("区间外")
}
```

这并不是巧合。没错，Swift 的 `switch` 就是使用了 `~=` 操作符进行模式匹配，`case` 指定的模式作为左参数输入，而等待匹配的被 `switch` 的元素作为操作符的右侧参数。只不过这个调用是由 Swift 隐式地完成的。于是我们可以发挥想象的地方就很多了，比如在 `switch` 中做 `case` 判断的时候，我们完全可以使用我们自定义的模式匹配方法来进行判断，有时候这会让代码变得非常简洁，具有条理。我们只需要按照需求重载 `~=` 操作符就行了，接下来我们通过一个使用正则表达式做匹配的例子加以说明。

首先我们要做的是重载 `~=` 操作符，让它接受一个 `NSRegularExpression` 作为模式，去匹配输入的 `String`：

```
func ~=(pattern: NSRegularExpression, input: String) -> Bool {
    return pattern.numberOfMatches(in: input,
        options: [],
        range: NSRange(location: 0, length: input.count)) > 0
}
```

然后为了简便起见，我们再添加一个将字符串转换为 `NSRegularExpression` 的操作符 (当然也可以使用 `ExpressibleByStringLiteral`，但是它不是这个 tip 的主题，在此就先不使用它了)：

```
prefix operator ~/

prefix func ~/(pattern: String) -> NSRegularExpression {
    return NSRegularExpression(pattern: pattern, options: nil, error: nil)
}
```

现在，我们在 `case` 语句里使用正则表达式的话，就可以去匹配被 `switch` 的字符串了：

```
let contact = ("http://onevcat.com", "onev@onevcat.com")

let mailRegex: NSRegularExpression
let siteRegex: NSRegularExpression

mailRegex =
    try ~/"^([a-z0-9_\\.-]+)@([\\da-z\\.-]+)\\.([a-z\\.]{2,6})$"
siteRegex =
    try ~/"^(https?:\\/\\/)?([\\da-z\\.-]+)\\.([a-z\\.]{2,6})([\\/\\w \\.-]*)*\\/?$"

switch contact {
    case (siteRegex, mailRegex): print("同时拥有有效的网站和邮箱")
    case (_, mailRegex): print("只拥有有效的邮箱")
    case (siteRegex, _): print("只拥有有效的网站")
    default: print("嘛都没有")
}

// 输出
// 同时拥有网站和邮箱
```

2. 从 Objective-C/C 到 Swift

Selector

`@selector` 是 Objective-C 时代的一个关键字，它可以将一个方法转换并赋值给一个 `SEL` 类型，它的表现很类似一个动态的函数指针。在 Objective-C 时 selector 非常常用，从设定 target-action，到自举询问是否响应某个方法，再到指定接受通知时需要调用的方法等等，都是由 selector 来负责的。在 Objective-C 里生成一个 selector 的方法一般是这个样子的：

```
-(void) callMe {
    //...
}

-(void) callMeWithParam:(id)obj {
    //...
}

SEL someMethod = @selector(callMe);
SEL anotherMethod = @selector(callMeWithParam:);

// 或者也可以使用 NSSelectorFromString
// SEL someMethod = NSSelectorFromString(@"callMe");
// SEL anotherMethod = NSSelectorFromString(@"callMeWithParam:");
```

一般为了方便，很多人会选择使用 `@selector`，但是如果要追求灵活的话，可能会更愿意使用 `NSSelectorFromString` 的版本 -- 因为我们可以在运行时动态生成字符串，从而通过方法的名字来调用到对应的方法。

在 Swift 中没有 `@selector` 了，取而代之，从 Swift 2.2 开始我们使用 `#selector` 来从暴露给 Objective-C 的代码中获取一个 selector。类似地，在 Swift 里对应原来 `SEL` 的类型是一个叫做 `Selector` 的结构体。像上面的两个例子在 Swift 中等效的写法是：

```
@objc func callMe() {
    //...
}

@objc func callMeWithParam(obj: AnyObject!) {
    //...
}

let someMethod = #selector(callMe)
let anotherMethod = #selector(callMeWithParam(obj:))
```

和 Objective-C 时一样，记得在 `callMeWithParam` 后面加上冒号和参数名 (:)，这才是完整的方法名字。多个参数的方法名也和原来类似，是这个样子：

```
@objc func turn(by angle: Int, speed: Float) {
    //...
}

let method = #selector(turn(by:speed:))
```

最后需要注意的是，selector 其实是 Objective-C runtime 的概念。在 Swift 4 中，默认情况下所有的 Swift 方法在 Objective-C 中都是不可见的，所以你需要在这类方法前面加上 `@objc` 关键字，将这个方法暴露给 Objective-C，才能进行使用。

在 Swift 3 和之前的版本中，Apple 为了更好的 Objective-C 兼容性，会自动对 `NSObject` 的子类的非私有方法进行推断并为在幕后为它们自动加上 `@objc` 。但是这需要每次 Swift 代码变动时都重新生成 Objective-C 所使用的头文件，这将造成 Swift 与 Objective-C 混编时速度大幅恶化。 另外，即使在 Swift 3 中，私有方法也只在 Swift 中可见，在调用这个 selector 时你会遇到一个 unrecognized selector 错误：

> ## 这是错误代码
>
> ```
> // In Swift 3
> private func callMe() {
>     //...
> }
>
> NSTimer.scheduledTimerWithTimeInterval(1, target: self,
>             selector:#selector(callMe), userInfo: nil, repeats: true)
> ```

正确的做法是在 `private` 前面加上 `@objc` 关键字，这样运行时就能找到对应的方法了。

```
@objc private func callMe() {
    //...
}

NSTimer.scheduledTimerWithTimeInterval(1, target: self,
            selector:#selector(callMe), userInfo: nil, repeats: true)
```

> 同理，现在如果你想要 Objective-C 能使用 Swift 的类型或者方法的话，也需要进行相应的标记。对于单个方法，在前面添加 `@objc` 。如果想让整个类型在 Objective-C 可用，可以在类型前添加 `@objcMembers` 。

最后，值得一提的是，如果方法名字在方法所在域内是唯一的话，我们可以简单地只是用方法的名字来作为 `#selector` 的内容。相比于前面带有冒号的完整的形式来说，这么写起来会方便一些：

```
let someMethod = #selector(callMe)
let anotherMethod = #selector(callMeWithParam)
let method = #selector(turn)
```

但是，如果在同一个作用域中存在同样名字的两个方法，即使它们的函数签名不相同，Swift 编译器也不允许编译通过：

```
@objc func commonFunc() {

}

@objc func commonFunc(input: Int) -> Int {
    return input
}

let method = #selector(commonFunc)
// 编译错误，`commonFunc` 有歧义
```

对于这种问题，我们可以通过将方法进行强制转换来使用：

```
let method1 = #selector(commonFunc as ()->())
let method2 = #selector(commonFunc as (Int)->Int)
```

3. Swift 与开发环境及一些实践

print 和 debugPrint

在定义和实现一个类型的时候，Swift 中的一种非常常见，也是非常先进的做法是先定义最简单的类型结构，然后再通过扩展 (extension) 的方式来实现为数众多的协议和各种各样的功能。这种按照特性进行分离的设计理念对于功能的可扩展性的提升很有帮助。虽然在 Objective-C 中我们也可以通过类似的 protocol + category 的形式完成类似的事情，但 Swift 相比于原来的方式更加简单快捷。

`CustomStringConvertible` 和 `CustomDebugStringConvertible` 这两个协议就是很好的例子。对于一个普通的对象，我们在调用 `print` 对其进行打印时只能打印出它的类型：

```
class MyClass {
    var num: Int
    init() {
        num = 1
    }
}

let obj = MyClass()
print(obj)
// MyClass
```

对于 struct 来说，情况好一些。打印一个 struct 实例的话，会列举出它所有成员的名字和值：比如我们有一个日历应用存储了一些会议预约，model 类型包括会议的地点，位置和参与者的名字：

```
struct Meeting {
    var date: NSDate
    var place: String
    var attendeeName: String
}

let meeting = Meeting(date: NSDate(timeIntervalSinceNow: 86400),
                      place: "会议室B1",
                 attendeeName: "小明")
print(meeting)
// 输出:
// Meeting(date: 2015-08-10 03:15:55 +0000,
//          place: "会议室B1", attendeeName: "小明")
```

直接这样进行输出对了解对象的信息很有帮助，但也会存在问题。首先如果实例很复杂，我们将很难在其中找到想要的结果；其次，对于 `class` 的对象来说，只能得到类型名字，可以说是毫无帮助。我们可以对输出进行一些修饰，让它看起来好一些，比如使用格式化输出的方式：

```
print("于 \(meeting.date) 在 \(meeting.place) 与 \(meeting.attendeeName) 进行会议")
// 输出:
// 于 2014-08-25 11:05:28 +0000 在 会议室B1 与 小明 进行会议
```

但是如果每次输出的时候，我们都去写这么一大串东西的话，显然是不可接受的。正确的做法应该是使用 `CustomStringConvertible` 协议，这个协议定义了将该类型实例输出时所用的字符串。相对于直接在原来的类型定义中进行更改，我们更应该倾向于使用一个 `extension`，这样不会使原来的核心部分的代码变乱变脏，是一种很好的代码组织的形式：

```
extension Meeting: CustomStringConvertible {
    var description: String {
        return "于 \(self.date) 在 \(self.place) 与 \(self.attendeeName) 进行会议"
    }
}
```

这样，再当我们使用 `print` 时，就不再需要去做格式化，而是简单地将实例进行打印就可以了：

```
print(meeting)
// 输出:
// 于 2015-08-10 03:33:34 +0000 在 会议室B1 与 小明 进行会议
```

`CustomDebugStringConvertible` 与 `CustomStringConvertible` 的作用很类似，但是仅发生在调试中使用 debugger 来进行打印的时候的输出。对于实现了 `CustomDebugStringConvertible` 协议的类型，我们可以在给 `meeting` 赋值后设置断点并在控制台使用类似 `po meeting` 的命令进行打印，控制台输出将为 `CustomDebugStringConvertible` 中定义的 `debugDescription` 返回的字符串。

后记及致谢

其实写这本书纯属心血来潮。从产生想法到做出决定花了一炷香的时间，而从开始下笔到这篇后记花了一个月的时间。

> 这么点儿时间，确实是不够写出一本好书的

这是我到现在，所得到的第一个教训。

虽然在博客上已经坚持写了两三年，但是写书来说，这还是自己的第一次。从刚下笔时的诚惶诚恐，到中途的渐入佳境，挥挥洒洒，再到最后绞尽脑汁，也算是在这一个月里把种种酸甜苦辣尝了个遍。诚然，不会有哪一本书能完美，大家也不必指望能得到什么武林秘籍帮你一夜功成。知识的积累从来都只能依靠日常点滴，而我也尽了自己的努力，尝试将我的积累分享出来，仅此而已。

所谓靡不有初，鲜克有终，我看过太多的雄心壮志，也见过许多的半途而废。如果您看到了这个后记，那么大概您真的是耐着性子把这本有些枯燥书都看完了。我很感谢您的坚持和对我的忍耐，并希望这些积累能够在您自己的道路上起到一些帮助。当然也可能您是喜欢“直接翻看后记”的剧透党，但我依然想要进行感谢，这个世界总会因为感谢而温暖和谐。

这本书在写作过程中参考了许多资料，包括但不限于 Apple 关于 Swift 的官方文档，Apple 开发者论坛上关于 Swift 的讨论，Stackoverflow 的 Swift 标签的所有问题，NSHipster，NSBlog 的周五问答，Airspeed Velocity，猫·仁波切 以及其他一些由于篇幅限制而没有列出参考博客。在此对这些社区的贡献者们表示衷心感谢。

在我写作的过程中，国内的许多开发者朋友们忍受了我的各种莫名其妙的低级疑问，他们的热心和细致的解答，对这本书的深入和准确性起到了很大的帮助。而本书的预售工作也在大家的捧场和宣传下顺利地进行并完成，在这里我想一并向他们表示感谢，正是你们的坚持和努力，让国内的开发者社区如此充满活力。

最后，感谢我的家人在这一个月时间内对我的照顾，让我可以不用承担和思考太多写书以外的事情。随着这本书暂时告一段落，我想是时候回归到每天洗碗和拖地的日常劳动中去了。

我爱这个世界，愿程序让这个世界变得更美好！

版本更新

4.0.0 (2017 年 8 月)

- 全书根据 Swift 4.0 的改动进行了更新，Playground 代码现在可以运行在 Swift 4 环境下。具体的增删修改内容请参见下文列表。

新增

- 《Selector》 Swift 4 中默认关闭了 @objc 推断，因此按照默认行为重写了部分说明，并对新版本中的对应方式进行解释。
- 《KeyPath 和 KVO》 KeyPath 在 Swift 4 中得到了原生支持，同时使用 KeyPath 可以大幅简化 KVO 的使用方式。以对比的方式添加了 Swift 4 中使用 KeyPath 的内容，并对非 `NSObject` 类型的 KVO 进行了简单讨论。
- 《属性访问控制》 Swift 4 中更改了部分 private 和 fileprivate 的含义。新增了例子对这部分内容进行说明。
- 《JSON 和 Codable》 使用 `Codable` 协议来处理 JSON 输入输出。同时对 `Codable` 进行了简单讨论。移除了 SwiftJSON 相关部分的内容。

修改

- 《模式匹配》 中一处 `ExpressibleByStringLiteral` 使用错误的问题。
- 《协议和类方法中的 Self》 调整部分语句，使其理解更加容易。
- 《Lock》 原示例代码的说明可能存在不同理解。修正了代码注释可能造成的误解，增加了实际使用例来避免歧义。
- String 相关：现在 `String` 实现了 `Collection` 协议，因此所有与 `String` 的集合特性相关的部分都使用简化的方式进行了重写。
- 一些文章中示例代码没有对应最新版本 Swift 的问题，以及修正原代码在新版本编译环境下的警告。

3.0.0 (2016 年 9 月)

总体

- 对应 Swift 3.0。几乎所有章节和相关实例代码均使用 Swift 3.0 的语法进行了重写。对于只有语法细节上的变化章节，在更新文档中将不再提及。

新增

- 《@escaping》 Swift 3 中 `@noescaping` 成为默认选项，而需要对可能发生逃逸的闭包添加

`@escaping`，对这可能引发的问题进行了一些讨论。
- 《associatedtype》介绍了如何在协议中引入类型占位，来实现类似泛型协议的特性。针对 `associatedtype` 使用上可能面临和困难和情景进行了介绍。
- 《Playground 可视化开发》介绍了使用 PlaygroundPage 的 `liveView` 来在 Xcode 集成环境中快速进行 UI 开发的方法。

修改或删除

- 《Sequence》Swift 3.0 中相关协议名称和方法发生了重要变化。
- 《操作符》现在操作符定义时需要指定优先级组，而不是直接进行定义。
- 《func 的参数修饰》因为已经完全弃用，因此移除了 C 语言风格的自增例子。增加了对于 `inout` 背后机制的一些说明。
- 《字面量表达》字面量转换 (LiteralConvertible) 已经被重命名为字面量表达 (Expressible)，因此本章相关的协议名称都进行了更新。
- 《Any 和 AnyObject》Swift 3 中 Any 和 AnyObject 的使用发生了很大变化，本节内容不再有效，因此删除。
- 《typealias》由于泛型协议的内容得到了强化，并且关键字也进行了分离，因此将这部分内容进行重新梳理，独立为《associatedtype》一节。
- 《protocol 组合》本节内容由于 `Any` 变化，不再有必要单独成文，因此将必要的内容合并到了《typealias》中。
- 《where 和模式匹配》现在 `where` 语句不再能够使用在可选绑定中，因此移除了可选绑定部分的内容，并进行了说明。
- 《条件编译》OS X 被重命名为 macOS，另外由于 Swift 现在是开源跨平台语言，额外增加了 Linux，Windows 和 Android 等系统平台选项。
- 《内存管理，weak 和 unowned》为了绕过 Playground 中一处编译器的限制，使用 `NSObject` 作为例子说明。
- 《GCD 和延时调用》使用 Swift 3 中的基于类型的 GCD 调用方式进行重写。
- 《Swizzle》由于更多地是 Objective-C 的概念，而且 Swizzle 的方法和 Swift 的编程思想没有太大关联，为了避免误导，将本节删除。
- 《调用 C 动态库》使用更先进的方式重写了本节的示例代码。
- 《类型编码 @encode》现在不允许隐式的类型转换了，因此在定义示例变量时使用了显式的类型转换。
- 《sizeof 和 sizeofValue》sizeof 和 sizeOfValue 现在被行为更加准备和统一的 `MemoryLayout` 取代了。因为 `MemoryLayout` 本身足够简单，因此本节也没有存在的必要，故删除。
- 《错误和异常处理》对于 `rethrows` 的定义和目的进行了更进一步的说明。另外添加了额外的错误定义例子。
- 《fatalError》对于父类虚函数现在可以使用协议扩展的方式来进行抽象，因此更新了相关部分的解释。
- 《Playground 延时运行》XCPlayground 已经被完全弃用，将所有代码更新为使用 PlaygroundSupport 框架的版本。
- 《Playground 可视化》XCPlayground 中值捕获的内容被完全移除，通过值捕获来进行可视化的技术已经成为历史，因此相关内容被删除。

- 《文档注释》 Xcode 8 中已经集成了生成文档注释的功能，因此对相关部分的说明进行了更改。

2.2.0 (2016 年 4 月 12 日)

- 对应 Swift 2.2。
- 《Selector》 Swift 2.2 中 `Selector` 已经可以使用安全的方式 (`#selector`) 进行定义，因此全书中 selector 相关的内容全部改为 Swift 2.2 中的语法进行。另外，根据 Swift 2.2 的特点，增加了重名 selector 的说明和相应问题的解决方法。
- 《func 的参数修饰》 Swift 2.2 中使用 `var` 修饰输入参数已被标记为弃用，按照 Swift 2.2 的写法重写了相关示例代码。
- 《Playground 延时运行》： `XCPlayground` 框架中大部分顶层方法都已经被弃用，使用 `XCPlaygroundPage` 中对应的方法重写了相关示例代码。
- 《Swizzle》：使用 `#selector` 重写了 Selector 声明部分的代码。
- 《柯里化》 多括号的柯里化写法已经被弃用，将文中的柯里化写法更新为多重返回的方式。
- 《Protocol Extension》 修正一处方法名称和文中名称没有对应的问题。
- 《@autoclosure 和 ??》 一节中关于 Swift 1.2 中 `@autoclosure` 的叙述已经过时，为了避免造成误解，将过时的注解删除。
- 《溢出》 修正了“溢出求模”的符号错误。
- 《多元组》 为了避免误导，删除了本节中使用 `NSError` 作为例子的说明。
- 《Sequence》 更新了本节中的示例代码，使其更接近于实际项目中会使用的代码。
- 《Log 输出》 现在使用类似 `#file` 的编译标记来代替原来的 `__FILE__`，对应进行了更改。
- 《Optional Map》 添加了关于函子 (Functor) 和函数性特性的简单说明及参考链接。
- 《协议和类方法中的 Self》 添加了有关使用 `final` 进行标记来满足 `init` 方法编译器需求的方式的说明。

2.1.0 (2015 年 11 月 24 日)

- 对应 Swift 2.1。
- 添加《安全的资源组织方式》一节，介绍使用 enum 或者 struct 等在编译时进行资源组织以保证安全性。
- 《单例》 修改了单例的写法，使用了更加简洁的形式，并且添加了对私有 init 的说明。
- 由于编译器的进步，struct mutating 现在有更好的错误提示了，因此删除了《Struct Mutable 的方法》一节。
- 《@autoclosure 和 ??》 一节中关于默认值的说明不太清晰，追加了说明。
- 修正了一些错别字。

2.0.2 (2015 年 9 月 19 日)

- 调整了在 Kindle 下的显示效果，加深了字体颜色
- 《@objc 和 dynamic》 Swift 2.0 中使用 `@objc` 来重命名类型名的技巧只适用于 ASCII 字符了，对此添加了一些说明和讨论。
- 《多类型和容器》 加入了 `Set` 类型的说明。

- 《哈希》 加入了关于哈希用途和使用时需要注意的说明。
- 《Protocol Extension》 重新审校，修正了一些语句和用词问题。
- 《AnyClass，元类型和 .self》 修正了一处错字。
- 《JSON》 修正了一处代码中的警告。

2.0.1 (2015 年 8 月 25 日)

根据 Xcode 7 beta 6 的改动对部分内容进行了更新。

- 《多元组》 删掉了最后的说明，因为在 Swift 2.0 中已经不再适用。
- 《可变参数函数》 Xcode 7 beta 6 中可变参数可以在函数中的任意位置，而不一定要是最后一个参数位置了。另外增加了相应的示例代码。
- 《String 还是 NSString》 修正了一处代码错误，Xcode 7 beta 6 中 `advance` 方法已经被 `advancedBy` 替代了。
- 《... 和 ..<》 中 `print` 方法的签名发生了变化，更新了对应的示例代码。
- 《lazy 修饰符和 lazy 方法》 `lazy` 在 Xcode 7 beta 6 中已经是一个 protocol extension 方法而不再是全局方法了，因此调整了代码使其符合要求。
- 《闭包歧义》 中省略闭包参数类型的写法现在一律被推断为 `Any`，这应该是一个编译器的 bug。增加了相关说明，我之后会对此继续关注并再次进行更新。
- 《错误和异常处理》 添加了关于 `try!`， `try?` 和 `rethrows` 的讨论以及对应的示例代码。
- 修正了一些章节的错别字。

2.0.0 (2015 年 8 月 24 日)

2.0.0 版本是本书的一个重大更新。根据 Swift 2.0 的内容重新修订了本书，包括对新内容的扩展和过时内容的删除。新版本中对原来的 tips 进行了归类整理，将全书大致分为三个部分，以方便读者阅读查阅。另外，为了阅读效果，对全书的排版和字体等进行了大幅调整。

这一年来随着 Swift 的逐渐变化，书中有不少示例代码的用法已经和现在版本的 Swift 有所区别，因此在这次修订中对所有的代码都在 Swift 2.0 环境下进行了再次的验证和修改。现在本书的所有代码例子已经附在 Swifter.playground 文件中，一并提供给读者进行参考。

新增条目

- 《Protocol Extension》
- 《where 和模式匹配》
- 《错误和异常处理》：代替了原来的错误处理一节。因为 Swift 2 中引入了异常处理的机制，因此现在对于发生错误后如何获取错误信息以及从错误中恢复有了新的方式。原来的内容已经不再适合新版本，因此用新的一节来替代。
- 《indirect 和嵌套 enum》
- 《尾递归》

修改和删除

- 为了表意明确，有特别的理由的个例除外，将其他所有返回 `()` 的闭包的改写为了返回 `Void`。
- 《@autoclosure 和 ??》 修正了一处笔误。
- 《操作符》 有几处 `Vector2` 应为 `Vector2D`，修正用词错误。
- 《typealias 和泛型协议》 修正一处表述问题，使得句子读起来更通顺。
- 《Sequence》 中 Sequence 相关的全局方法现在已经被写为协议扩展，因此对说明也进行了相应地更改。
- 《多元组》 中有关错误处理的代码已经被异常机制替代，因此选取了一个新的例子来说明如何使用多元组。
- 《方法参数名称省略》 Swift 2 中已经统一了各 scope 中的方法名称中的参数规则，因此本节已经没有存在的必要，故删去。
- 《Swift 命令行工具》 编译器的命令行工具在输出文件时的参数发生了变化，对此进行修正。另外在运行命令时省去了已经不必要的 xcrun。
- 《下标》 `Array` 现在的泛型类型占位符变为了 `Element`，另外原来的 `Slice` 被 `ArraySlice` 取代了。更新了代码使其能在新版本下正常工作。
- 《初始化返回 nil》 因为 `Int` 已经有了内建的从 `String` 进行初始化的方法，因此改变了本节的例程。现在使用一个中文到数字的转换初始化方法来进行说明。
- 《protocol 组合》 修正了示例代码中的错误。
- 《static 和 class》 Swift 2 中已经可以用 static 作为通用修饰，因此修改了一些过时的内容。
- 《可选协议》 中加入了关于使用协议扩展来实现协议方法可选的技巧。因为加入了其他内容，这一节也更名为《可选协议和协议扩展》。
- 《内存管理，weak 和 unowned》 新版本中 Playground 也能正确地反应内存状况以及与 ARC 协同工作了，因此去除了必须在项目中运行的条件，另外修改了代码使它们能在 Playground 中正常工作。
- 《default 参数》 中与参数 # 修饰符相关的内容已经过时，删除。`NSLocalizedString` 的补全现在也已经改进，所以不再需要说明。
- 《正则表达式》 `NSRegularExpression` 的初始化方法现在有可能直接抛出异常，使用异常机制重写了本节的示例代码。
- 《模式匹配》 同《正则表达式》，使用异常机制重写了示例代码。
- 《COpaquePointer 和 C convention》 `CFunctionPointer` 在 Swift 2.0 中被删除，现在 C 方法指针可以直接由 Swift 闭包进行无缝转换。重写了该部分内容，添加了关于 `@convention` 标注的说明。
- 《Foundation 框架》在 Swift 2.0 中 `String` 和 `NSString` 的转换已经有了明确的界限，因此本节内容已经过时，故删去。
- 《GCD 和延时调用》 重新说明了 iOS 8 中对 block 的改进。另外由于 Swift 2 中重新引入了 `performSelector`，对相关内容进行了小幅调整。
- 《获取对象类型》 Swift.Type 现在对于 `print` 和 `debugPrint` 中有了新的实现，进行了补充说明和代码修正。
- 《类型转换》 因为 Objective-C 中对 collection 加入了泛型的支持，现在在 Swift 中使用 Cocoa API 时基本已经不太需要类型转换，故删去。
- 《局部 scope》 添加了关于 `do` 的说明。Swift 2.0 中加入了 `do` 关键字，可以作为局部作用域来使用。

- 《print 和 debugPrint》 现在 `Printable` 和 `DebugPrintable` 协议的名称分别改为了 `CustomStringConvertible` 和 `CustomDebugStringConvertible`。
- 《Playground 限制》 随着 Apple 对 Playground 的改进和修复，原来的一些限制 (特别是内存管理上的限制) 现在已经不复存在。这一节已经没有太大意义，故删去。
- 《Swizzle》 `+load` 方法在 Swift 1.2 中已经不能直接被覆盖使用，另外使用 Swift 实现的 `+load` 方法在运行时也不再被调用，因此需要换为使用 `+initialize` 来实现方法的交换。改写了代码以使其正常工作，另外加入了关于交换方法选择的说明。
- 《find》因为引入了 protocol extension，像类似 `find` 一类作用在 collection 上的全局方法都已经使用 protocol extension 实现了，因此本节移除。
- 《Reflection 和 Mirror》 `reflect` 方法和 `MirrorType` 类型现在已经变为 Swift 标准库的私有类型，现在我们需要使用 `Mirror` 来获取和使用对象的反射。重写了本节的内容以符合 Swift 2.0 中的反射特性和使用方式。另外，为了避免误导，对反射的使用场合也进行了说明。
- 《文档注释》 Swift 2.0 中文档注释的格式发生了变化，因此对本节内容进行了修改已符合新版本的格式要求。
- 《Options》 原来的 `RawOptionSetType` 在 Swift 2.0 中已经被新的 `OptionSetType` 替代，现在 Options 有了更简洁的表示方法和运算逻辑。另外加入了 Options 集合运算的内容，以及更新了生成 Options 的代码片段。
- 《Associated Object》 作为 Key 值的变量需要是 Optional 类型，因此对原来不正确的示例代码进行了修改。
- 《Swift 中的测试》 Swift 2 中导入了 `@testable`，可以让测试 target 访问到导入的 target 的 internal 代码，因此本节的一些讨论过时了。根据 Swift 2.0 的测试方式重写了本节内容。
- [其他]：修正了一些用词上的不妥和错别字。

1.2.1 (2015 年 2 月 25 日)

- 《@autoclosure 和 ??》，以及其他出现 `@autoclosure` 的章节中，将 `@autoclosure` 的位置进行了调整。现在 `@autoclosure` 作为参数名的修饰，而非参数类型的修饰。
- 《闭包歧义》 Swift 1.2 中闭包歧义的使用将由编译器给出错误。在保留 Swift 1.1 及之前的讨论的前提下，补充说明了 Swift 1.2 版本以后的闭包歧义的处理和避免策略。
- 《获取对象类型》删除了过时内容和已经无效的黑科技，补充了 `dynamicType` 对内建 Swift 类型的用法说明。
- 《单例》 现在可以直接使用类常量/变量，因此更新了推荐的单例写法。
- 《static 和 class》中更新了类常/变量的用法。
- 《@UIApplicationMain》中 `C_ARGC` 和 `C_ARGV` 分别被 `Process.argc` 和 `Process.unsafeArgv` 替代。
- 《COpaquePointer 和 CFunctionPointer》更新了一处 API 的参数名。
- 《类型转换》使用意义明确的 Swift 1.2 版本的 `as!` 进行强制转换。
- 《数学和数字》作为补充，添加了 `Darwin` 中判定 NAN 的方法 `isnan`。
- 《Swift 命令行工具》中新增了新版本中不需要 `xcrun` 的说明。
- 《方法参数名称省略》中的一处 API 的 unwrap 更新。
- 《C 代码调用和 @asmname》 修正了一个头文件引入的错误。

1.2.0 (2015 年 2 月 10 日)

- 《static 和 class》一节针对 Swift 1.2 进行了更新。Swift 1.2 中 protocol 中定义的“类方法”需要使用 static 而非 class。
- 《多类型和容器》中的错别字，“不知名”应该为“不指明”。
- 《内存管理，weak 和 unowned》 中标注例子中标注错误，标注中的逗号应该是冒号。
- 《Reflection 和 MirrorType》一节代码中遗漏了一个引号。
- 《Any 和 AnyObject》 修正了一处赋值时的代码警告。

1.1.2 (2014 年 12 月 2 日)

- 《default 参数》中的错别字，“常亮”应该为“常量”。
- 《Selector》中示例代码 `func aMethod(external paramName: String) { ... }` 中 `String` 应为 `AnyObject!`，否则会导致程序崩溃（因为 Swift 的原生 `String` 并没有实现 `NSCopying`）。
- 《获取对象类型》 中由于 API 变更，使用 `objc_getClass` 和 objc runtime 获取对象类型的方法已经不再有效。新的 API 需要与 `UnsafePointer` 有关，已经超出章节内容，故将该部分内容删除。
- 《多类型和容器》 中由于 Swift 类型推断和字面量转换的改善，原来的陷阱基本都已经消除，因此本节进行了一些简化，去掉了过时的内容。

1.1.1 (2014 年 11 月 7 日)

- 《闭包歧义》中的内容在 Swift 1.1 中已经发生了变化，因此重写了这一节。
- 将代码示例中的 `toRaw()` 和 `fromRaw()` 按照 Swift 1.1 的语法改为了 `rawValue` 和对应的 `init` 方法。

1.1.0 (2014 年 10 月 21 日)

- 《属性访问控制》中“Swift 中的 swift 和其他大部分语言不太一样” 应为 “Swift 中的 private 和其他大部分语言不太一样”
- 《代码组织和 Framework》中 “开发中我们所使以的第三方框” 改为 “开发中我们所使用的第三方框”
- 《方法名称参数省略》中 “使用自动覆盖的方式” 应为 “使用原子写入的方式”
- 《内存管理，weak 和 unowned》中对 unowed 的表述不当，进行了修正
- 《字面量转换》的内容完全重写，更新为适应 Swift 1.1 版本。
- 《Any 和 AnyObject》一节作出修正，Any 现在可以支持方法类型了。
- 《初始化返回 nil》一节作出修正，在本书 1.0.1 版本的基础上，进一步更改了部分内容的表述，使其更适应 Swift 1.1 中关于可失败的初始化方法的修改。

1.0.1 (2014 年 9 月 25 日)

- 《多类型和容器》中在 `let mixed: [Printable]` 处有个字面量转换的小陷阱，在是否导入 `Foundation` 时存在一个有趣的差别，对这部分进行了一定补充说明；

- 《将 protocol 的方法声明为 mutating》中 `blueColor()!` -> `blueColor()` 。 `blueColor()` 返回的已经是 `UIColor` 而不是 `UIColor?`；
- 《typealias 和泛型协议》中有误字 “wei”，删除；
- 《Optional Chaining 文字错误》 它们的等价的 -> 它们是等价的；
- 《Swift 命令行工具》中示例代码 xcrun swift MyClass.swift main.swift 应当为 xcrun swiftc MyClass.swift main.swift；
- 《内存管理，weak和unowned》笔误，“变量一定需要时 Optional 值” 中 “时” 应当为 “是”。

1.0.0 (2014 年 9 月 19 日)

《Swifter - Swift 必备 tips》首发