Broadview®
www.broadview.com.cn

objc.io | ObjC 中国

函数式Swift

Functional Swift

[德] Chris Eidhof　Florian Kugler　Wouter Swierstra　著

陈聿菡　杜　欣　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

函数式Swift

Functional Swift

[德] Chris Eidhof　Florian Kugler　Wouter Swierstra　著

陈聿菡　杜　欣　王　巍　译

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

内容简介

Swift 是一门有着合适的语言特性来适配函数式编程方法的优秀语言，对国内的大部分开发者来说，Swift 可能是我们第一次真正有机会去接触和使用的一门函数式特性语言。Swift 在语法上更加优雅灵活，语言本身也遵循了函数式的设计模式。

本书是一本引领你进入 Swift 函数式编程世界的优秀读物，它让更多的中国开发者有机会接触并了解 Swift 语言函数式的一面，是广大程序开发者不可多得的工具书。

图书在版编目（CIP）数据

函数式 Swift /（德）克里斯·安道夫（Chris Eidhof），（德）弗洛莱恩·库格勒（Florian Kugler），（德）沃特·斯维（Wouter Swierstra）著；陈聿菡，杜欣，王巍译.—北京：电子工业出版社，2016.9

ISBN 978-7-121-29357-3

I. ①函…II. ①克…②弗…③沃…④陈…⑤杜…⑥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（2016）第 157831 号

策划编辑：张春雨

责任编辑：王　静

印　　刷：三河市鑫金马印装有限公司

装　　订：三河市鑫金马印装有限公司

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：12　　字数：250 千字

版　　次：2016 年 9 月第 1 版

印　　次：2016 年 9 月第 1 次印刷

印　　数：3000 册　　定价：65.00 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。

译序

随着程序语言的发展，软件开发人员所熟知和使用的工具也在不断进化。以 Java 和 C++ 为代表的面向对象编程的编程方式在 20 世纪企业级的软件开发中大放异彩，然而随着软件行业的不断发展，开发者们发现了面向对象范式的诸多不足。面向对象强调的是将与某数据类型相关的一系列操作都封装到该数据类型中去，因此，在数据类型中难免存在大量状态，以及相关的行为。虽然这很符合人类的逻辑直觉，但是当类型关系变得错综复杂时，类型中状态的改变和类型之间彼此的继承和依赖将使程序的复杂度呈几何级数上升。

避免使用程序状态和可变对象，是降低程序复杂度的有效方式之一，而这也正是函数式编程的精髓。函数式编程强调执行的结果，而非执行的过程。我们先构建一系列简单却具有一定功能的小函数，然后再将这些函数进行组装以实现完整的逻辑和复杂的运算，这是函数式编程的基本思想。

正如上面引言所述，Swift 是一门有着合适的语言特性来适配函数式编程方法的优秀语言。这个世界上最纯粹的函数式编程语言非 Haskell 莫属，但是由于我国程序开发的起步和走热相对西方世界要晚一些，使用 Haskell 的开发者可谓寥寥无几，因此 Haskell 在国内的影响力也十分有限。对国内的不少开发者，特别是 iOS / OS X 的开发者来说，Swift 可能是我们第一次真正有机会去接触和使用的一门函数式特性语言。相比于很多已有的函数式编程语言，Swift 在语法上更加优雅灵活，语言本身也遵循了函数式的设计模式。作为函数式编程的入门语言，可以说 Swift 是非常理想的选择。而本书正是一本引领你进入 Swift 函数式编程世界的优秀读物，让更多的中国开发者有机会接触并了解 Swift 语言函数式的一面，正是我们翻译本书的目的所在。

本书大致上可以分为两个部分。首先，在第 1 章至第 9 章中，我们会介绍 Swift 函数式编程特性的一些基本概念，包括高阶函数的使用方法、不可变量的必要性、可选值的存在价值、枚举在函数式编程中的意义，以及纯函数式数据结构的优势等内容。这些都是函数式编程中的基本概念，也构成了 Swift 函数式特性甚至是这门语言的基础。当然，在这些概念讲解中我们也穿插了不少研究案例，以帮助读者真正理解这些基本概念，并对在何时使用它们以及使用它们为程序设计带来的改善形成直观印象。第二部分从第 10 章开始，相比于前面

的章节，这部分属于本书的进阶内容。我们将从构建最基本的生成器和序列开始，利用解析器组合算子构建一个解析器库，并最终实现一个相对复杂的公式解析器和函数式的表格应用。这部分内容环环相扣，因为内容抽象度较高，所以理解起来也可能比较困难。如果你在阅读第 12 章时遇到麻烦，则强烈建议你下载对应的完整源码进行研究，并且折回头去再次阅读第二部分的相关章节。随着你对各个函数式算子的深入理解，函数式编程的概念和思想将自然而然进入你的血液，这将丰富你的知识体系，并会对之后的开发工作大有裨益。

本书原版的三位作者都是富有经验的函数式编程方法的使用者或教师，他们将自己对于函数式编程的理解和 Swift 中的相关特性进行了对应和总结，并将这些联系揭示了出来。而中文版的三位译者花费了大量时间和精力，试图将这些规律以更易于理解的组织方式和语言，带给国内的开发者们。不过不论是原作者还是译者，其实和各位读者一样，都只不过是普通开发者中的一员，所以本书出现谬漏可能在所难免。如果您在阅读时发现了问题，可以给我们发邮件，或是在本书 issue 页面提出，我们将及时研究并加以改进。

事不宜迟，现在就让我们开始在函数式的世界中遨游一番吧！

陈聿菡
杜欣
王巍

前言

为什么写这本书？关于 Swift，已经有大量来自苹果的现成文档，而且还有更多的书正在编写中。为什么世界上依然需要关于这种编程语言的另一本书呢？

这本书尝试让你学会以**函数式**的方式进行思考。我们认为 Swift 有着合适的语言特性来适配**函数式**的编程方法。然而是什么使得程序具有函数式特性？又为何要一开始就学习关于函数式的内容呢？

很难给出函数式的准确定义——同样地，我们也很难给出面向对象编程，抑或是其他编程范式的准确定义。因此，我们会尽量把重点放在我们认为设计良好的 Swift 函数式程序应该具有的一些**特质**上：

- **模块化**：相比于把程序认为是一系列赋值和方法调用，函数式开发者更倾向于强调每个程序都能够被反复分解为越来越小的模块单元，而所有这些块可以通过函数装配起来，以定义一个完整的程序。当然，只有当我们能够避免在两个独立组件之间共享状态时，才能将一个大型程序分解为更小的单元。这引出我们的下一个关注特质。
- **对可变状态的谨慎处理**：函数式编程有时候（被半开玩笑地）称为“面向值编程”。面向对象编程专注于类和对象的设计，每个类和对象都有自己的封装状态。然而，函数式编程强调基于值编程的重要性，这能使我们免受可变状态或其他一些副作用的困扰。通过避免可变状态，函数式程序比其对应的命令式或者面向对象的程序更容易组合。
- **类型**：最后，一个设计良好的函数式程序在使用**类型**时应该相当谨慎。精心选择你的数据和函数的类型，将会有助于构建你的代码，这比其他东西都重要。Swift 有一个强大的类型系统，使用得当的话，则它能够让你的代码更加安全和健壮。

我们认为这些特质是 Swift 程序员可能从函数式编程社区学习到的精华点。在这本书中，我们会通过许多实例和学习案例说明以上几点。

根据我们的经验，学习用函数式的方式思考并不容易。它挑战了我们既有的熟练解决问题的方式。对习惯写 for 循环的程序员来说，递归可能让我们倍感迷惑；赋值语句和全局状态的缺失让我们寸步难行；更不用提闭包、泛型、高阶函数和单子 (Monad)，这些东西简直让人痛不欲生。

在这本书中，我们假定你以前有过 Objective-C (或其他一些面向对象的语言) 的编程经验。书中不会涵盖 Swift 的基础知识，或者教你建立你的第一个 Xcode 工程，不过我们会尝试在适当的时候引用现有的 Apple 文档。你应当能自如地阅读 Swift 程序，并且熟悉常见的编程概念，如类、方法和变量等。如果你只是刚刚开始学习编程，则这本书可能并不适合你。

在这本书中，我们希望让函数式编程易于理解，并消除人们对它的一些偏见。使用这些理念去改善你的代码并不需要你拥有数学的博士学位！函数式编程并不是 Swift 编程的**唯一**方式。但是我们相信学习函数式编程会为你的工具箱添加一件重要的新工具，不论你使用那种语言，这件工具都会让你成为一个更好的开发者。

示例代码

你可以在我们的 GitHub 仓库[1]中找到本书里所有的示例代码。这个仓库包括一些章节的 playgrounds，以及其他章节的 Swift 文件和 OS X 工程。

书籍更新

随着 Swift 的发展，我们会继续更新和改进这本书。如果你遇到任何错误，或者是想给我们其他类型的反馈，请在我们的 GitHub 仓库[2]中创建一个 issue。

致谢

我们想要感谢众多帮助我们塑造了这本书的人。在此我们想要特别提及其中几位：

Natalye Childress 是我们的出版编辑。她给了我们很多宝贵的反馈意见，不仅保证了语言的正确性和一致性，而且确保了本书清晰易懂。

Sarah Lincoln 设计了本书的封面和布局。

Wouter 想要感谢**乌得勒支大学**允许他能够在这本书上投入时间进行编写。

[1] *https://github.com/objcio/functional-swift*
[2] *https://github.com/objcio/functional-swift*

我们想要感谢测试版读者在本书的写作过程中给我们的反馈(按字母顺序排列):

Adrian Kosmaczewski, Alexander Altman, Andrew Halls, Bang Jun-young, Daniel Eggert, Daniel Steinberg, David Hart, David Owens II, Eugene Dorfman, f-dz-v, Henry Stamerjohann, J Bucaran, Jamie Forrest, Jaromir Siska, Jason Larsen, Jesse Armand, John Gallagher, Kaan Dedeoglu, Kare Morstol, Kiel Gillard, Kristopher Johnson, Matteo Piombo, Nicholas Outram, Ole Begemann, Rob Napier, Ronald Mannak, Sam Isaacson, Ssu Jen Lu, Stephen Horne, TJ, Terry Lewis, Tim Brooks, Vadim Shpakovski.

Chris
Florian
Wouter

目录

函数式 Swift 基础

第1章　函数式思想

函数在 Swift 中是**一等值** (first-class-value)，换句话说，函数可以作为参数被传递到其他函数，也可以作为其他函数的返回值。如果你习惯了使用像整型、布尔型或结构体这样的简单类型来编程，那么这个理念可能看来非常奇怪。在本章中，我们会尽可能清晰地解释为什么一等函数是很有用的语言特性，并实际地提供本书的第一个函数式编程案例。

1.1　案例：Battleship

下面我们会用一个小案例来引出一等函数：这个例子是你在编写战舰类游戏时可能需要实现的一个核心函数。我们把将要看到的问题归结为，判断一个给定的点是否在射程范围内，并且距离友方船舶和我们自身都不太近。

首先，你可能会写一个很简单的函数来检验一个点是否在范围内。为了简明易懂，我们假定我们的船位于原点。这样一来，我们就可以将想要描述的区域形象化，如图 1.1所示。

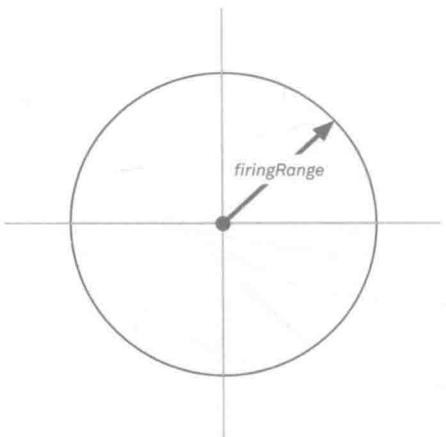

图 1.1　位于原点的船舶射程范围内的点

首先，我们定义两种类型——Distance 和 Position：

```
typealias Distance = Double

struct Position {
    var x: Double
    var y: Double
}
```

然后我们在 Position 中添加一个函数 inRange(_:)，用于检验一个点是否在图 1.1中的灰色区域里。使用一些基础的几何知识，我们可以像下面这样定义这个函数：

```
extension Position {
    func inRange(range: Distance) -> Bool {
        return sqrt(x * x + y * y) <= range
    }
}
```

如果假设我们总是位于原点，那么现在这样就可以正常工作了。但是船舶还可能在原点以外的其他位置出现，我们可以更新一下形象化图，如图 1.2所示。

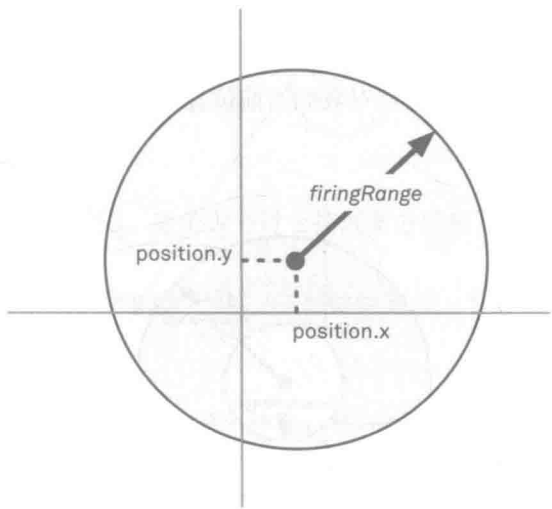

图 1.2　允许船有它自己的位置

考虑到这一点，我们引入一个结构体 Ship，它有一个属性为 position：

```
struct Ship {
    var position: Position
    var firingRange: Distance
    var unsafeRange: Distance
}
```

目前，姑且先忽略附加属性 unsafeRange。后面我们会回到这个问题。

我们向结构体 Ship 中添加一个 canEngageShip(_:) 函数对其进行扩展，这个函数允许我们检验是否有另一艘船在范围内，不论我们是位于原点还是其他任何位置：

```
extension Ship {
    func canEngageShip(target: Ship) -> Bool {
        let dx = target.position.x - position.x
        let dy = target.position.y - position.y
        let targetDistance = sqrt(dx * dx + dy * dy)
        return targetDistance <= firingRange
    }
}
```

也许现在你已经意识到，我们同时还想避免目标船舶离你过近。我们可以用图 1.3来说明新情况，我们想要瞄准的仅仅只有那些对我们当前位置而言在 unsafeRange（不安全范围）外的敌人：

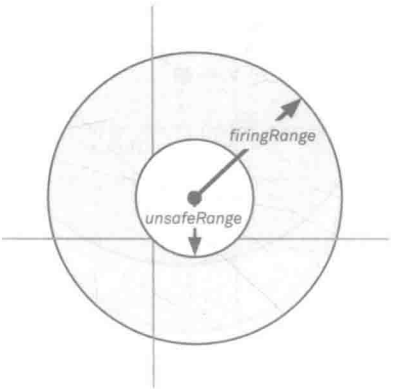

图 1.3　避免与过近的敌方船舶交战

这样一来，我们需要再一次修改代码，使 unsafeRange 属性能够发挥作用：

```
extension Ship {
    func canSafelyEngageShip(target: Ship) -> Bool {
        let dx = target.position.x - position.x
        let dy = target.position.y - position.y
        let targetDistance = sqrt(dx * dx + dy * dy)
        return targetDistance <= firingRange && targetDistance > unsafeRange
    }
}
```

最后，我们还需要避免目标船舶过于靠近我方的任意一艘船。我们再一次将其形象化，见图 1.4。

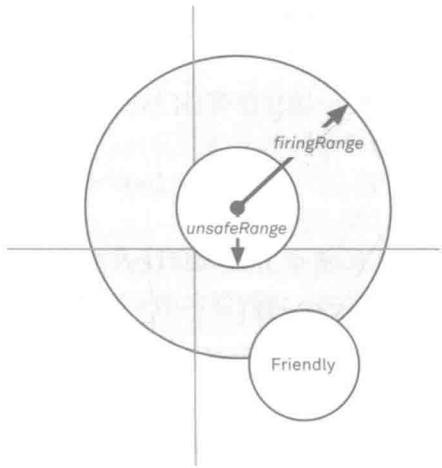

图 1.4　避免敌方过于接近友方船舶

相应地，我们可以向 canSafelyEngageShip(_:) 函数添加另一个参数代表友好船舶位置：

```
extension Ship {
    func canSafelyEngageShip1(target: Ship, friendly: Ship) -> Bool {
        let dx = target.position.x - position.x
        let dy = target.position.y - position.y
        let targetDistance = sqrt(dx * dx + dy * dy)
        let friendlyDx = friendly.position.x - target.position.x
        let friendlyDy = friendly.position.y - target.position.y
```