

objc.io | objc 中国

函数式

Swift

已对应 Swift 4

Chris Eidhof, Florian Kugler, Wouter Swiersta 著
陈聿菡, 杜欣, 王巍 译

英文版本 4.0 (2017 年 11 月), 中文版本 4.0 (2017 年 11 月)

© 2017 Kugler, Eggert und Eidhof GbR

版权所有

ObjC 中国

在中国地区独家翻译和销售授权

获取更多书籍或文章, 请访问 <http://objccn.io>
电子邮件: mail@objccn.io

1	引言	7
	译序 8	
2	函数式思想	12
	案例：Battleship 12	
	一等函数 18	
	类型驱动开发 22	
	注解 22	
3	案例研究：封装 Core Image	25
	滤镜类型 25	
	理论背景：柯里化 30	
	讨论 32	
4	Map、Filter 和 Reduce	34
	泛型介绍 34	
	Filter 38	
	Reduce 39	
	实际运用 43	
	泛型和 Any 类型 44	
	注释 46	
5	可选值	48
	案例研究：字典 48	
	玩转可选值 51	
	为什么使用可选值？ 57	
6	案例研究：QuickCheck	61
	构建 QuickCheck 62	
	缩小范围 65	

展望 71

7	不可变性的价值	74
	变量和引用 74	
	值类型与引用类型 75	
	讨论 79	
8	枚举	83
	关于枚举 83	
	关联值 85	
	添加泛型 87	
	Swift 中的错误处理 88	
	再聊聊可选值 89	
	数据类型中的代数学 90	
	为什么使用枚举? 92	
9	纯函数式数据结构	94
	二叉搜索树 94	
	基于字典树的自动补全 101	
	讨论 108	
10	案例研究：图表	110
	绘制正方形和圆 110	
	核心数据结构 113	
	讨论 123	
11	迭代器和序列	126
	迭代器 126	
	序列 132	
	案例研究：遍历二叉树 136	
	案例研究：优化 QuickCheck 的范围收缩 136	

12	案例研究：解析器组合算子	140
	解析器的类型 140	
	组合解析器 143	
	解析算术表达式 153	
	更 Swift 化的解析器类型 154	
13	案例研究：构建一个表格应用	158
	解析 158	
	求值 164	
	用户界面 169	
14	函子、适用函子与单子	172
	函子 172	
	适用函子 174	
	单子 177	
	讨论 179	
15	尾声	182
	拓展阅读 182	
	结语 183	

引言

1

为什么写这本书？关于 Swift，已经有大量来自 Apple 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 — 其实同样地，我们也很难给出面向对象编程，亦或是其它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一些特质上：

- **模块化**：相较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，每个类和对象都有它们自己的封装状态。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，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，我们将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。对于习惯写 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，或是教你创建你的第一个 Xcode 工程，但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。如果你刚刚开始学习编程，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。

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，或者是想给我们其它类型的反馈，请在我们的 [GitHub 仓库](#) 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。

我们还想要感谢测试版的读者们在本书写作过程中给予我们反馈 (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, Karel 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, and Wouter

译序

随着程序语言的发展，我们作为软件开发人员，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上世纪企业级的软件开发中大放异彩，然而随着软件行业不断发展，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的时候，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几何上升。

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热门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。对于国内的不少开发者，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，在第二章至第十章中，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。第二部分从第十一章开始，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。如果你在阅读最后表格应用章节时遇到麻烦的话，我们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

函数式思想

2

函数在 Swift 中是一等值 (first-class-values)，换句话说，函数可以作为参数被传递到其它函数，也可以作为其它函数的返回值。如果你习惯了使用像是整型，布尔型或是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。

函数和方法将会贯穿整个章节。方法是函数的一种特殊情况：它是定义在类型里的函数。

案例：Battleship

我们将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 2.1：

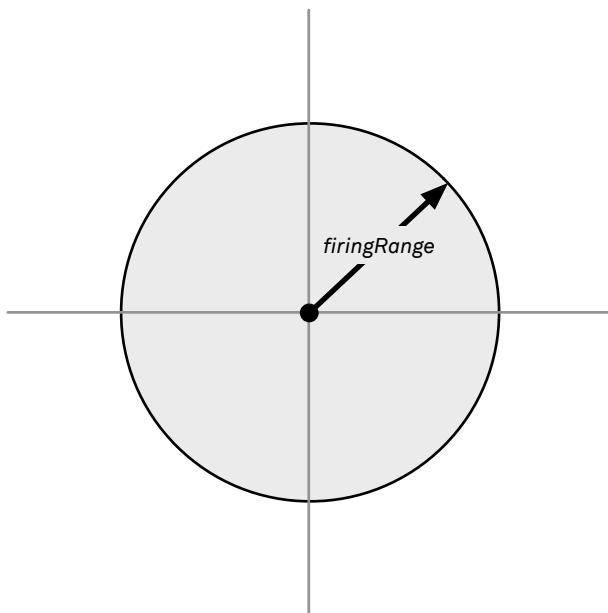


图 2.1：位于原点的船舶射程范围内的点

首先，我们定义两种类型，Distance 和 Position：

```
typealias Distance = Double

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

注意，我们使用了 Swift 的 typealias 关键字，它允许我们为已经存在的类型创建一个别名。这里我们将 Distance 定义为 Double 的别名，目的在于让 API 的语义更清晰。

然后我们在 Position 中添加一个方法 within(range:)，用于检验一个点是否在图 2.1 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 within(range: Distance) -> Bool {
        return sqrt(x * x + y * y) <= range
    }
}
```

如果假设我们总是位于原点，那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它位置出现，我们可以更新一下形象化图，如图 2.2 所示：

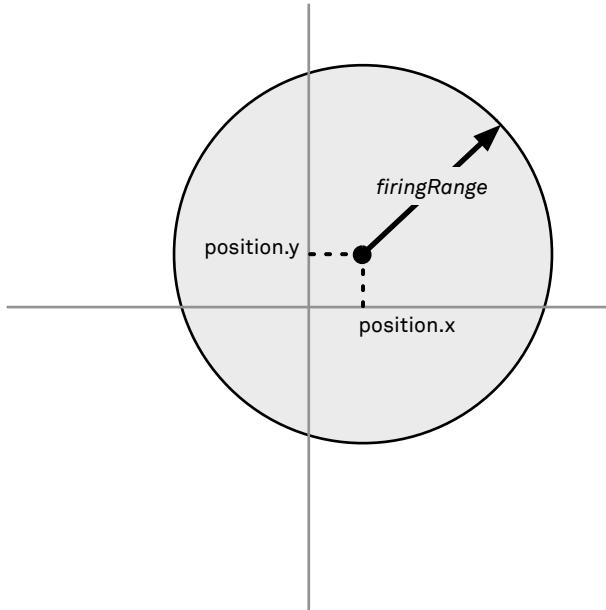


图 2.2: 允许船有它自己的位置

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。一会儿我们会回到这个问题。

我们向结构体 Ship 中添加一个 canEngage(ship:) 方法对其进行扩展，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它任何位置：

```
extension Ship {  
    func canEngage(ship 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得过近。可以用图 2.3 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：

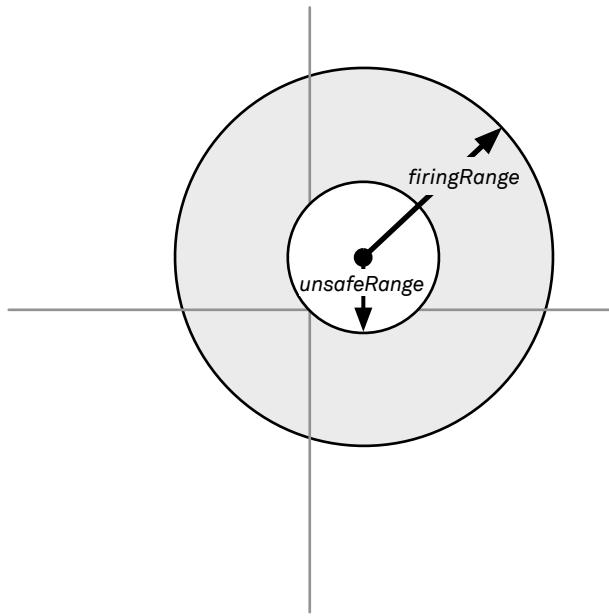


图 2.3: 避免与过近的敌方船舶交战

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 canSafelyEngage(ship 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 2.4：

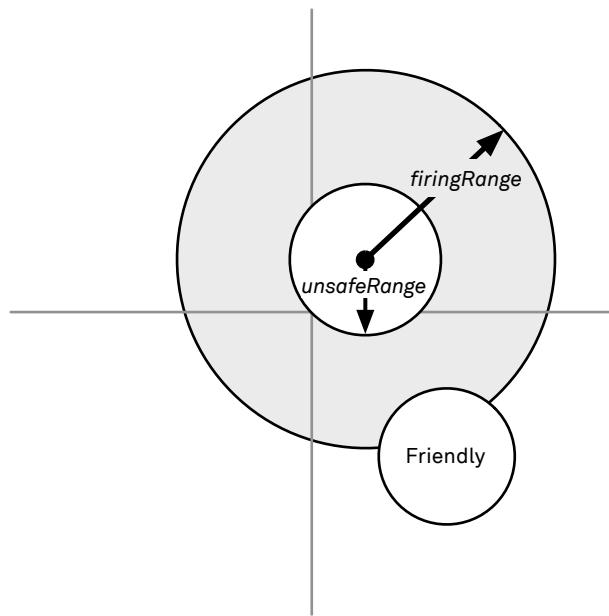


图 2.4: 避免敌方过于接近友方船舶

相应地，我们可以向 `canSafelyEngage(ship:)` 方法添加另一个参数代表友好船舶位置：

```

extension Ship {
func canSafelyEngage(ship 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
let friendlyDistance = sqrt(friendlyDx * friendlyDx +
    friendlyDy * friendlyDy)
return targetDistance <= firingRange
&& targetDistance > unsafeRange
&& (friendlyDistance > unsafeRange)
}
}

```

随着代码逐渐发展，它变得越来越难以维护。这个方法包含了一大段复杂的运算代码，不过也没有那么糟，我们可以在 `Position` 中添加一个辅助方法和一个计算属性来负责几何运算，从而让这段代码变得清晰易懂一些：

```

extension Position {
    func minus(_ p: Position) -> Position {
        return Position(x: x - p.x, y: y - p.y)
    }
    var length: Double {
        return sqrt(x * x + y * y)
    }
}

```

添加了辅助方法和属性之后，上述方法变成了下面这样：

```

extension Ship {
    func canSafelyEngage2(ship target: Ship, friendly: Ship) -> Bool {
        let targetDistance = target.position.minus(position).length
        let friendlyDistance = friendly.position.minus(target.position).length
        return targetDistance <= firingRange
            && targetDistance > unsafeRange
            && (friendlyDistance > unsafeRange)
    }
}

```

现在的代码相较之前已经易读多了，但是我们还想更进一步，用一种声明式的方法来明确现有的问题。

一等函数

在当前 `canSafelyEngage(ship:friendly)` 的方法中，主要的行为是为构成返回值的布尔条件组合进行编码。在这个简单的例子中，虽然想知道这个函数做了什么并不是太难，但我们还是想要一个更模块化的解决方案。

我们已经在 `Position` 中引入了辅助方法使几何运算的代码更清晰易懂。用同样的方式，我们现在要添加一个函数，以更加声明式的方式来判断一个区域内是否包含某个点。

原来的问题归根结底是要定义一个函数来判断一个点是否在范围内。这样一个函数的类型应该是像下面这样的：

```
func pointInRange(point: Position) -> Bool {  
    // 方法的具体实现  
}
```

由于该函数的类型将会非常重要，所以我们打算给它一个独立的名字：

```
typealias Region = (Position) -> Bool
```

从现在开始，`Region` 类型将指代把 `Position` 转化为 `Bool` 的函数。严格来说这不是必须的，但是它可以让我们更容易理解在接下来即将看到的一些类型。

我们使用一个能判断给定点是否在区域内的函数来代表一个区域，而不是定义一个对象或结构体来表示它。如果你不习惯函数式编程，这可能看起来会很奇怪，但是记住：在 Swift 中函数是一等值！我们有意识地选择了 `Region` 作为这个类型的名字，而非 `CheckInRegion` 或 `RegionBlock` 这种字里行间暗示着它们代表一种函数类型的名字。**函数式编程的核心理念就是函数是值，它和结构体、整型或是布尔型没有什么区别 —— 对函数使用另外一套命名规则会违背这一理念。**

我们现在要写几个函数来创建、控制和合并各个区域。

我们定义的第一个区域是以原点为圆心的圆 (`circle`)：

```
func circle(radius: Distance) -> Region {  
    return { point in point.length <= radius }  
}
```

值得注意的是，以半径 `r` 为参数，调用 `circle(radius: r)` **返回的是一个函数**。这里我们使用了 Swift 的闭包来构造我们期待的返回函数。给定一个表示位置的参数 `point`，然后检查该 `point` 是否在以原点为中心且给定半径的圆所限定的区域中。

当然，并不是所有圆的圆心都是原点。我们可以给 `circle` 函数添加更多的参数来解决这个问题。要得到一个圆心是任意定点的圆，我们只需要添加另一个代表圆心的参数，并确保在计算新区域时将这个参数考虑进去：

```
func circle2(radius: Distance, center: Position) -> Region {  
    return { point in point.minus(center).length <= radius }  
}
```

然而，如果我们想对更多的图形组件（例如，想象我们不仅有圆，还有矩形或其它形状）做出同样的改变，可能需要重复这些代码。更加函数式的方式是写一个**区域变换函数**。这个函数按一定的偏移量移动一个区域：

```
func shift(_ region: @escaping Region, by offset: Position) -> Region {  
    return { point in region(point.minus(offset)) }  
}
```

当需要在函数返回之后使用其参数（如：`region`）时，该参数需要被标记为 `@escaping`。即使忘记标记，编译器也将提醒我们。如果你想了解更详细的信息，请参阅 Apple 的 *The Swift Programming Language* 一书中《逃逸闭包》的部分。

调用 `shift(region, by: offset)` 函数会将区域向右上方移动，偏移量分别是 `offset.x` 和 `offset.y`。我们需要的是一个传入 `Position` 并返回 `Bool` 的函数 `Region`。为此，我们需要另写一个闭包，它接受我们要检验的点，这个点减去偏移量之后我们得到一个新的点。最后，为了检验新点是否在**原来的**区域内，我们将它作为参数传递给 `region` 函数。

这是函数式编程的核心概念之一：为了避免创建像 `circle2` 这样越来越复杂的函数，我们编写了一个 `shift(_:by:)` 函数来改变另一个函数。例如，一个圆心为 `(5, 5)` 半径为 `10` 的圆，可以用下面的方式表示：

```
let shifted = shift(circle(radius: 10), by: Position(x: 5, y: 5))
```

还有很多其它的方法可以变换已经存在的区域。例如，也许我们想要通过反转一个区域以定义另一个区域。这个新产生的区域由原区域以外的所有点组成：

```
func invert(_ region: @escaping Region) -> Region {
    return { point in !region(point) }
}
```

我们也可以写一个函数将既存区域合并为更大更复杂的区域。比如，下面两个函数分别可以计算参数中两个区域的交集和并集：

```
func intersect(_ region: @escaping Region, with other: @escaping Region)
-> Region {
    return { point in region(point) && other(point) }
}

func union(_ region: @escaping Region, with other: @escaping Region)
-> Region {
    return { point in region(point) || other(point) }
}
```

当然，我们可以利用这些函数来定义更丰富的区域。`difference` 函数接受两个区域作为参数——原来的区域和要减去的区域——然后为所有在第一个区域中且不在第二个区域中的点构建一个新的区域：

```
func subtract(_ region: @escaping Region, from original: @escaping Region)
-> Region {
    return intersect(original, with: invert(region))
}
```

这个例子告诉我们，在 Swift 中计算和传递函数的方式与整型或布尔型没有任何不同。这让我们能够写出一些基础的图形组件（比如圆），进而能以这些组件为基础，来构建一系列函数。每个函数都能修改或是合并区域，并以此创建新的区域。比起写复杂的函数来解决某个具体的问题，现在我们完全可以通过将一些小型函数装配起来，广泛地解决各种各样的问题。

现在让我们把注意力转回原来的例子。关于区域的小型函数库已经准备就绪，我们可以像下面这样重构 `canSafelyEngage(ship:friendly:)` 这个复杂的方法：

```
extension Ship {
    func canSafelyEngage(ship target: Ship, friendly: Ship) -> Bool {
        let rangeRegion = subtract(circle(radius: unsafeRange),
                                   from: circle(radius: firingRange))
        let firingRegion = shift(rangeRegion, by: position)
```

```
let friendlyRegion = shift(circle(radius: unsafeRange),
    by: friendly.position)
let resultRegion = subtract(friendlyRegion, from: firingRegion)
return resultRegion(target.position)
}
}
```

这段代码定义了两个区域：firingRegion 和 friendlyRegion。通过计算这两个区域的差集（即在 firingRegion 中且不在 friendlyRegion 中的点的集合），我们可以得到我们感兴趣的区域。将这个区域函数作用在目标船舶的位置上，我们就可以计算所需的布尔值了。

面对同一个问题，与原来的 canSafelyEngage(ship:friendly:) 方法相比，使用 Region 方法重构后的版本是更加声明式的解决方案。我们坚信后一个版本会更容易理解，因为我们的解决方案是装配式的。你可以探究组成它的每个部分，例如 firingRegion 和 friendlyRegion，看一看它们是如何被装配并解决原来的问题的。另一方面，原来庞大的方法混合了各个组成区域的描述语句和描述它们所需要的算式。通过定义我们之前提出的辅助函数将这些关注点进行分离，显著提高了复杂区域的组合性和易读性。

能做到这样，一等函数是至关重要的。虽然 Objective-C 也支持一等函数，或者说是 **block**，即使可以做到类似的事情，但难掩遗憾，在 Objective-C 中使用 block 十分繁琐。一部分原因是由于语法问题：block 的声明和 block 的类型与 Swift 的对应部分相比并不是那么简单。在后面的章节中，我们也将看到泛型如何让一等函数更加强大，远比 Objective-C 中用 blocks 实现更加容易。

我们定义 Region 类型的方法有它自身的缺点。我们选择了将 Region 类型定义为简单类型，并作为 (Position) -> Bool 函数的别名。其实，我们也可以选择将其定义为一个包含单一函数的结构体：

```
struct Region {
    let lookup: Position -> Bool
}
```

接下来我们可以用 extensions 的方式为结构体定义一些类似的方法，来代替对原来的 Region 类型进行操作的自由函数。这可以让我们能够通过对区域进行反复的函数调用来变换这个区域，直至得到需要的复杂区域，而不用像以前那样将区域作为参数传递给其他函数：

```
rangeRegion.shift(ownPosition).difference(friendlyRegion)
```

这种方法有一个优点，它需要的括号更少。再者，这种方式下 Xcode 的自动补全在装配复杂的区域时会十分有用。不过，为了便于展示，我们选择了使用简单的 typealias 以突出在 Swift 中使用高阶函数的方法。

此外，值得指出的是，现在我们不能够看到一个区域是如何被构建的：它是由更小的区域组成的吗？还是单纯只是一个以原点为圆心的圆？我们唯一能做的是检验一个给定的点是否在区域内。如果想要形象化一个区域，我们只能对足够多的点进行采样来生成（黑白）位图。

在后面的章节中，我们将使用另外一种设计，来帮助你解答这些问题。

我们在本章节，或者说是整本书中使用的命名方式稍有违背 Swift API 设计指南。Swift 指南要求在设计时考虑方法名。例如，如果 `intersect` 被定义为一个方法，那么该方法名应该为 `intersecting` 或 `intersected`，因为返回值是一个新值，而非对现有区域进行修改。尽管如此，在编写顶级函数时，我们决定使用 `intersect` 这样的基本形式。

类型驱动开发

在引言中，我们提到了函数式编程可以用规范的方式将函数作为参数装配为规模更大的程序。在本章中，我们看到了一个以这种函数式方式设计的具体例子。我们定义了一系列函数来描述区域。每一个函数单打独斗的话都并不强大。然而装配到一起时，却可以描述你绝不会想要从零开始编写的复杂区域。

解决的办法简单而优雅。这与单纯地将 `canSafelyEngage(ship:friendly:)` 方法拆分为几个独立的方法那种重构方式是完全不同的。我们确定了如何来定义区域，这是至关重要的设计决策。当我们选择了 `Region` 类型之后，其它所有的定义就都自然而然，水到渠成了。这个例子给我们的启示是，我们应该 **谨慎地选择类型**。这比其他任何事都重要，因为类型将左右开发流程。

注解

本章提供的代码受到了一个 Haskell 解决方案的启发，该方案解决了由美国高等研究计划局 (ARPA) 的 Hudak and Jones (1994) 提出的一个问题。

Objective-C 通过引入 `blocks` 实现了对一等函数的支持：你可以将函数和闭包作为参数并轻松地使用内联的方式定义它们。然而，在 Objective-C 中使用它们并不像在 Swift 中一样方便，尽管两者在语意上完全相同。

从历史上看，一等函数的理念可以追溯到 Church 的 lambda 演算 (Church 1941; Barendregt 1984)。此后，包括 Haskell, OCaml, Standard ML, Scala 和 F# 在内的大量 (函数式) 编程语言都不同程度地借鉴了这个概念。

案例研究：封装 Core Image

3

前一章介绍了**高阶函数**的概念，并展示了将函数作为参数传递给其它函数的方法。不过，使用的例子可能与你日常写的“真实”代码相去甚远。在本章中，我们将会围绕一个已经存在且面向对象的 API，展示如何使用高阶函数将其以小巧且函数式的方式进行封装。

Core Image 是一个强大的图像处理框架，但是它的 API 有时可能略显笨拙。Core Image 的 API 是弱类型的——我们通过键值编码 (KVC) 来配置图像滤镜 (filter)。在使用参数的类型或名字时，我们都使用字符串来进行表示，这十分容易出错，极有可能导致运行时错误。而我们开发的新 API 将会利用**类型**来避免这些原因导致的运行时错误，最终我们将得到一组类型安全而且高度模块化的 API。

即使你不熟悉 Core Image，或者不能完全理解本章代码片段的细节，也大可不必担心。我们的目标并不是围绕 Core Image 构建一个完整的封装，而是要说明如何把像高阶函数这样的函数式编程概念运用到实际的生产代码中。

滤镜类型

CIFilter 是 Core Image 中的核心类之一，用于创建图像滤镜。当实例化一个 CIFilter 对象时，你(几乎)总是通过 `kCIInputImageKey` 键提供输入图像，再通过 `outputImage` 属性取回处理后的图像。取回的结果可以作为下一个滤镜的输入值。

在本章即将开发的 API 中，我们会尝试封装应用这些键值对的具体细节，从而呈现给用户一个安全的强类型 API。我们将 Filter 类型定义为一个函数，该函数接受一个图像作为参数并返回一个新的图像：

```
typealias Filter = (CIImage) -> CIImage
```

我们将以这个类型为基础进行后续的构建。

构建滤镜

现在我们已经定义了 Filter 类型，接着就可以开始定义函数来构建特定的滤镜了。这些函数在接受特定滤镜所需要的参数之后，构造并返回一个 Filter 类型的值。它们的基本形态大概都是下面这样：

```
func myFilter(...) -> Filter
```

注意这里的返回值 Filter，也是一个函数。稍后我们将会借助它来组合多个滤镜以实现期待的图像效果。

模糊

让我们来定义第一个简单的滤镜——高斯模糊滤镜。定义它只需要模糊半径这一个参数：

```
func blur(radius: Double) -> Filter {
    return { image in
        let parameters: [String: Any] = [
            kCIInputRadiusKey: radius,
            kCIInputImageKey: image
        ]
        guard let filter = CIFilter(name: "CIGaussianBlur",
            withInputParameters: parameters)
        else { fatalError() }
        guard let outputImage = filter.outputImage
        else { fatalError() }
        return outputImage
    }
}
```

一切就是这么简单。blur 函数返回一个新函数，新函数接受一个 `CIImage` 类型的参数 `image`，并返回一个新图像 (`return filter.outputImage`)。因此，blur 函数的返回值满足我们之前定义的 `(CIImage) -> CIImage`，也就是 `Filter` 类型。

`CIFilter` 的初始化及 `filter` 的 `outputImage` 属性都返回可选值，我们使用了 `guard` 语句来解包这些可选值。如果这些值中有 `nil`，那么就意味着发生了程序错误的情况。比方说，如果我们向滤镜传递了错误的参数，就会导致这种情况。结合 `fatalError()` 使用 `guard` 语句而非强制解包可选值使得这种意图更明显。

这个例子仅仅只是对 `Core Image` 中一个已经存在的滤镜进行的简单封装。我们可以反复使用相同的模式来创建自己的滤镜函数。

颜色叠层

现在让我们来定义一个能够在图像上覆盖纯色叠层的滤镜。Core Image 默认不包含这样一个滤镜，但是我们完全可以用已经存在的滤镜来组成它。

我们将使用的两个基础组件：颜色生成滤镜 (CIClusterColorGenerator) 和图像覆盖合成滤镜 (CICompositeOver)。首先让我们来定义一个生成固定颜色的滤镜：

```
func generate(color: UIColor) -> Filter {
    return { _ in
        let parameters = [kCIInputColorKey: CIColor(cgColor: color.cgColor)]
        guard let filter = CIFilter(name: "CIClusterColorGenerator",
            withInputParameters: parameters)
        else { fatalError() }
        guard let outputImage = filter.outputImage
        else { fatalError() }
        return outputImage
    }
}
```

这段代码看起来和我们用来定义模糊滤镜的代码非常相似，但是有一个显著的区别：颜色生成滤镜不检查输入图像。因此，我们不需要给返回函数中的图像参数命名。取而代之，我们使用一个匿名参数 `_` 来强调滤镜的输入图像参数是被忽略的。

接下来，我们将要定义合成滤镜：

```
func compositeSourceOver(overlay: CIImage) -> Filter {
    return { image in
        let parameters = [
            kCIInputBackgroundImageKey: image,
            kCIInputImageKey: overlay
        ]
        guard let filter = CIFilter(name: "CICompositeOver",
            withInputParameters: parameters)
        else { fatalError() }
        guard let outputImage = filter.outputImage
        else { fatalError() }
        return outputImage.cropped(to: image.extent)
    }
}
```

```
}
```

在这里我们将输出图像剪裁为与输入图像一致的尺寸。严格来说，这不是必须的，完全取决于我们希望滤镜如何工作。不过，这个选择在我们即将涉及的例子中效果很好。

最后，我们通过结合两个滤镜来创建颜色叠层滤镜：

```
func overlay(color: UIColor) -> Filter {
    return { image in
        let overlay = generate(color: color)(image).cropped(to: image.extent)
        return compositeSourceOver(overlay: overlay)(image)
    }
}
```

我们再次返回了一个接受图像作为参数的函数。我们在定义该函数的整个过程中所做的一切可大致概括为：首先使用先前定义的颜色生成滤镜函数 `generate(color:)` 来生成一个新叠层。然后以颜色作为参数调用该函数，返回 `Filter` 类型值。而 `Filter` 类型本身就是一个从 `CIImage` 到 `CIImage` 的函数，因此我们还可以向 `generate(color:)` 函数传递一个 `image` 参数，最终通过计算能够得到一个 `CIImage` 类型的新叠层。

与叠层的创建相似，所构建的滤镜函数的返回值也有着相同的结构：先通过调用 `compositeSourceOver(overlay:)` 来创建一个滤镜。然后以输入图像为参数调用这个滤镜。

组合滤镜

到现在为止，我们已经定义了模糊滤镜和颜色叠层滤镜，可以把它们组合在一起使用：首先将图像模糊，然后再覆盖上一层红色叠层。让我们来载入一张图片试试看：

```
let url = URL(string: "http://via.placeholder.com/500x500")!
let image = CIImage(contentsOf: url)!
```

现在我们可以链式地将两个滤镜应用到载入的图像上：

```
let radius = 5.0
let color = UIColor.red.withAlphaComponent(0.2)
let blurredImage = blur(radius: radius)(image)
let overlaidImage = overlay(color: color)(blurredImage)
```

我们再一次通过创建滤镜来处理图像，例如先创建 blur(radius:radius) 滤镜，随后将其运用到图像上。

复合函数

当然，我们可以将上面代码里两个调用滤镜的表达式简单合为一体：

```
let result = overlay(color: color)(blur(radius: radius)(image))
```

然而，由于括号错综复杂，这些代码很快失去了可读性。更好的解决方式是构建一个可以将滤镜合二为一的函数：

```
func compose(filter filter1: @escaping Filter,  
           with filter2: @escaping Filter) -> Filter  
{  
    return { image in filter2(filter1(image)) }  
}
```

compose(filter:with:) 函数接受两个 Filter 类型的参数，并返回一个新滤镜。这个复合滤镜接受一个 CIImage 类型的图像参数，然后将该参数传递给 filter1，取得返回值之后再传递给 filter2。这里我们举一个例子，来阐述我们可以如何使用 compose(filter:with:) 定义自己的复合滤镜：

```
let blurAndOverlay = compose(filter: blur(radius: radius),  
                           with: overlay(color: color))  
let result1 = blurAndOverlay(image)
```

为了让代码更具可读性，我们还可以再进一步，为组合滤镜引入自定义运算符。诚然，随意自定义运算符并不一定对提升代码可读性有帮助。不过，在图像处理库中，滤镜的组合是一个反复被讨论的问题。一旦你知道了这个运算符，滤镜的定义也会随之变得易读：

infix operator >>>

```
func >>>(filter1: @escaping Filter, filter2: @escaping Filter) -> Filter {  
    return { image in filter2(filter1(image)) }  
}
```

现在我们可以使用 >>> 运算符达到与之前使用 compose(filter:with:) 相同的目的：

```
let blurAndOverlay2 =  
    blur(radius: radius) >>> overlay(color: color)  
let result2 = blurAndOverlay2(image)
```

由于运算符 `>>>` 默认是左结合的 (left-associative)，就像 Unix 的管道一样，因此滤镜将以从左到右的顺序被应用到图像上。

我们定义的组合滤镜运算符是一个**复合函数**的例子。在数学中， f 和 g 两个函数构成的复合函数有时候被写作 $f \cdot g$ ，表示定义的新函数将输入的 x 映射到 $f(g(x))$ 。除了顺序，这恰恰也是我们的 `>>>` 运算符所做的：将一个图像参数传递给运算符操作的两个滤镜。

理论背景：柯里化

在本章中，我们已经反复写了好多次类似下面这样的代码：

```
blur(radius: radius)(image)
```

先调用一个函数，该函数返回新函数 (本例指 `Filter`)，然后传入另一个参数并调用之前返回的新函数。事实上也可以简单地传递两个参数给 `blur` 函数，然后直接返回图像，两者效果完全相同：

```
let blurredImage = blur(image: image, radius: radius)
```

写一个函数返回另一个函数，只是为了再次调用被返回的函数。为什么我们要选择这个看似更复杂的方式呢？

结合上文我们知道，有两种等效的方式能够定义一个可以接受两个 (或更多) 参数的函数。对于大多数程序员来说，应该会觉得第一种风格更熟悉：

```
func add1(_ x: Int, _ y: Int) -> Int {  
    return x + y  
}
```

`add1` 函数接受两个整型参数并返回它们的和。然而在 Swift 中，我们对该函数的定义还可以有另一个版本：

```
func add2(_ x: Int) -> ((Int) -> Int) {  
    return { y in x + y }
```

```
}
```

这里的 add2 函数接受第一个参数 `x` 之后，返回一个闭包，然后再接受第二个参数 `y`。这与滤镜函数的结构一模一样。

因为该函数的箭头**向右结合** (right-associative)，所以我们可以移除包围在结果函数类型周围的括号。从而得到本质上与 add2 完全等价的函数 add3：

```
func add3(_ x: Int) -> (Int) -> Int {  
    return { y in x + y }  
}
```

add1 与 add2 的区别在于调用方式：

```
add1(1, 2) // 3  
add2(1)(2) // 3
```

在第一种方法中，我们将两个参数同时传递给 add1；而第二种方法则首先向函数传递第一个参数 1，然后将返回的函数应用到第二个参数 2。两个版本是完全等价的：我们可以根据 add2 来定义 add1，反之亦然。

add1 和 add2 的例子向我们展示了如何将一个接受多参数的函数变换为一系列只接受单个参数的函数，这个过程被称为**柯里化** (Currying)，它得名于逻辑学家 Haskell Curry；我们将 add2 称为 add1 的柯里化版本。

那么，为什么说柯里化很有趣呢？正如迄今为止我们在本书中所看到的，在一些情况下，你可能想要将函数作为参数传递给其它函数。如果我们有像 add1 一样**未柯里化**的函数，那我们就必须同时用到它的**全部两个参数**来调用这个函数。然而，对于一个像 add2 一样被**柯里化**了的函数来说，我们有两个选择：可以使用一个**或两个参数**来调用。

在本章中为了创建滤镜而定义的函数全部都已经被柯里化了——它们都接受一个附加的图像参数。按照柯里化风格来定义滤镜，我们可以很容易地使用 `>>>` 运算符将它们进行组合。假如我们用这些**函数未柯里化**的版本来构建滤镜的话，虽然依然可以写出相同的滤镜，但是这些滤镜的类型将根据它们所接受的参数不同而略有不同。这样一来，想要为这些不同类型的滤镜定义一个统一的组合运算符就要比现在困难得多了。

讨论

这个例子再一次阐释了将复杂的代码拆解为小块的方式，而这些小块可以使用函数的方式进行重新装配，并形成完整的功能。本章的目标并不是为 Core Image 定义一个完整的 API，而是想说明在更实际的案例中如何使用高阶函数和复合函数。

为什么要这样做这么多努力呢？事实上 Core Image API 已经很成熟并能够提供几乎所有你可能需要的功能。但是尽管如此，我们相信本章所设计的 API 也有一些优点：

- **安全** — 使用我们构筑的 API 几乎不可能发生由未定义键或强制类型转换失败导致的运行时错误。
- **模块化** — 使用 `>>>` 运算符很容易将滤镜进行组合。这样你可以将复杂的滤镜拆解为更小，更简单，且可复用的组件。此外，组合滤镜与组成它的组件是完全相同的类型，所以你可以交替使用它们。
- **清晰易懂** — 即使你从未使用过 Core Image，也应该能够通过我们定义的函数来装配简单的滤镜。你完全不需要关心 `kCIIputImageKey` 或 `kCIIputRadiusKey` 这样的特定键如何进行初始化。单看类型，你几乎就能够知道如何使用 API，甚至不需要更多文档。

我们的 API 提供了一系列能够用来定义和组合滤镜的函数。使用和复用任何自定义的滤镜都是安全的。每个滤镜都能被独立测试和理解。比起原来的 Core Image API，我们的设计会更让人偏爱。如果说理由的话，相信上面几点足够让人信服。

Map、Filter 和 Reduce

4

接受其它函数作为参数的函数有时被称为**高阶函数**。在 Swift 标准库中就有几个作用于数组的高阶函数，本章中，我们将对它们进行探索。伴随这个过程，同时将介绍 Swift 的**泛型**，以及展示如何将复杂计算运用于数组。

泛型介绍

假如我们需要写一个函数，它接受一个给定的整型数组，通过计算得到并返回一个新数组，新数组各项为原数组中对应的整型数据加一。这一切，仅仅只需要使用一个 for 循环就能非常容易地实现：

```
func increment(array: [Int]) -> [Int] {
    var result: [Int] = []
    for x in array {
        result.append(x + 1)
    }
    return result
}
```

现在假设我们还需要一个函数，用于生成一个每项都为参数数组对应项两倍的新数组。这同样能很容易地使用一个 for 循环来实现：

```
func double(array: [Int]) -> [Int] {
    var result: [Int] = []
    for x in array {
        result.append(x * 2)
    }
    return result
}
```

这两个函数有大量相同的代码，我们能不能将没有区别的地方抽象出来，并单独写一个体现这种模式且更通用的函数呢？像这样的函数需要追加一个新参数来接受一个函数，这个参数能根据各个数组项计算得到新的整型数值：

```
func compute(array: [Int], transform: (Int) -> Int) -> [Int] {
    var result: [Int] = []
    for x in array {
        result.append(transform(x))
    }
}
```

```
    return result
}
```

现在，可以根据我们想如何从原数组得到一个新数组，来向函数传递不同的参数了。随之，`double` 函数和 `increment` 函数都精简为了一行调用 `compute` 的语句：

```
func double2(array: [Int]) -> [Int] {
    return compute(array: array) { $0 * 2 }
}
```

代码仍然不像想象中的那么灵活。假设我们想要得到一个布尔型的新数组，用于表示原数组中对应的数字是否是偶数。我们可能会尝试编写一些像下面这样的代码：

```
func isEven(array: [Int]) -> [Bool] {
    return compute(array: array) { $0 % 2 == 0 }
}
```

不幸的是，这段代码导致了一个类型错误。问题在于，我们的 `compute` 函数接受一个 `(Int) -> Int` 类型的参数，也就是说，该参数是一个返回整型值的函数。而在 `isEven` 函数的定义中，我们传递了一个 `(Int) -> Bool` 类型的参数，于是导致了类型错误。

我们应该如何解决这个问题呢？一种可行方案是定义新版本的 `compute(array:transform:)` 函数，接受一个 `(Int) -> Bool` 类型的函数作为参数。类似下面这样：

```
func compute(array: [Int], transform: (Int) -> Bool) -> [Bool] {
    var result: [Bool] = []
    for x in array {
        result.append(transform(x))
    }
    return result
}
```

但是，这个方案的扩展性并不好。如果接下来需要计算 `String` 类型呢？我们是否还需要定义另一个高阶函数来接受 `(Int) -> String` 类型的参数？

幸运的是，该问题有一个解决方案：我们可以使用泛型。不论参数是 `transform: (Int) -> Bool` 还是 `transform: (Int) -> Int`，`compute` 的定义是相同的；唯一的区别在于**类型签名 (type signature)**。假如我们要定义一个参数为 `transform: (Int) -> String` 的相似函数来支持 `String`

类型，其函数体也将会与先前两个函数完全一致。事实上，相同部分的代码可以用于**任何**类型。我们真正想做的是写一个能够适用于每种可能类型的泛型函数：

```
func genericCompute<T>(array: [Int], transform: (Int) -> T) -> [T] {  
    var result: [T] = []  
    for x in array {  
        result.append(transform(x))  
    }  
    return result  
}
```

关于这段代码，最有意思的是它的类型签名。理解这个类型签名有助于你将 `genericCompute<T>` 理解为一个函数族。**类型参数** `T` 的每个选择都会确定一个新函数。该函数接受一个整型数组和一个 `(Int) -> T` 类型的函数作为参数，并返回一个 `[T]` 类型的数组。

没有理由让这个函数仅能处理类型为 `[Int]` 的输入数组，让我们来进一步将它一般化：

```
func map<Element, T>(_ array: [Element], transform: (Element) -> T) -> [T] {  
    var result: [T] = []  
    for x in array {  
        result.append(transform(x))  
    }  
    return result  
}
```

这里我们写了一个 `map` 函数，它在两个维度都是通用的：对于任何 `Element` 的数组和 `transform: (Element) -> T` 函数，它都会生成一个 `T` 的新数组。这个 `map` 函数甚至比我们之前看到的 `genericCompute` 函数更通用。事实上，我们可以通过 `map` 来定义 `genericCompute`：

```
func genericCompute2<T>(array: [Int], transform: (Int) -> T) -> [T] {  
    return map(array, transform)  
}
```

同样地，上述函数的定义并没有什么太过特别之处：函数接受 `array` 和 `transform` 两个参数之后，将它们传递给 `map` 函数，然后返回结果。关于这个定义，最有意思地方还是类型定义。`genericCompute2(array:transform:)` 是 `map` 函数的一个实例，只不过它有一个更具体的类型。

实际上，比起定义一个顶层 `map` 函数，按照 Swift 的惯例将 `map` 定义为 `Array` 的扩展会更合适：

```
extension Array {  
    func map<T>(_ transform: (Element) -> T) -> [T] {  
        var result: [T] = []  
        for x in self {  
            result.append(transform(x))  
        }  
        return result  
    }  
}
```

我们在函数的 `transform` 参数中所使用的 `Element` 类型源自于 Swift 的 `Array` 中对 `Element` 所进行的泛型定义。

作为 `map(array, transform: transform)` 的替代，我们现在可以通过 `array.map(transform)` 来调用 `Array` 的 `map` 函数：

```
func genericCompute3<T>(array: [Int], transform: (Int) -> T) -> [T] {  
    return array.map(transform)  
}
```

想必你会很乐意听到其实并不需要自己像这样来定义 `map` 函数，因为它已经是 Swift 标准库的一部分了（实际上，它基于 `Sequence` 协议被定义，我们会在之后关于迭代器和序列的章节中提到）。本章的重点并不是说你应该自己定义 `map`；我们只是想要告诉你 `map` 的定义中并没有什么复杂难懂的魔法——你能够轻松地自己定义它！

顶层函数和扩展

你可能已经注意到，在本节的函数中我们使用了两种不同的方式来声明函数：顶层函数和类型扩展。在一开始创建 `map` 函数的过程中，为了简单起见，我们选择了顶层函数的版本作为例子进行展示。不过，最终我们将 `map` 的泛型版本定义为 `Array` 的扩展，这与它在 Swift 标准库中的实现方式十分相似。

在 Swift 标准库最初的版本中，顶层函数仍然是无处不在的，但伴随 Swift 2 的诞生，这种模式被彻底地从标准库中移除了。随着协议扩展（protocol extensions），当前第三方开发者有了一个强有力的工具来定义他们自己的扩展——现在我们不仅仅可以在 `Array` 这样的具体类型上进行定义，还可以在 `Sequence` 之类的协议上来定义扩展。

我们建议遵循此规则，并把处理确定类型的函数定义为该类型的扩展。这样做的优点是自动补全更完善，有歧义的命名更少，以及(通常)代码结构更清晰。

Filter

map 函数并不是 Swift 标准数组库中唯一一个使用泛型的函数。我们将在后面的部分中介绍其它几个。

假设我们有一个由字符串组成的数组，代表文件夹的内容：

```
let exampleFiles = ["README.md", "HelloWorld.swift", "FlappyBird.swift"]
```

如果我们想要一个包含所有 .swift 文件的数组，可以很容易通过简单的循环得到：

```
func getSwiftFiles(in files: [String]) -> [String] {
    var result: [String] = []
    for file in files {
        if file.hasSuffix(".swift") {
            result.append(file)
        }
    }
    return result
}
```

现在可以使用这个函数来取得 exampleFiles 数组中的 Swift 文件：

```
getSwiftFiles(in: exampleFiles) // ["HelloWorld.swift", "FlappyBird.swift"]
```

当然，我们可以将 getSwiftFiles 函数一般化。比如，相比于使用硬编码 (hardcoding) 的方式筛选扩展名为 .swift 的文件，传递一个附加的 String 参数进行比对会是更好的方法。这样我们接下来就可以使用同样的函数去比对像是 .md 这样的其他文件了。但是假如我们想查找没有扩展名的所有文件，或者是名字以字符串 "Hello" 开头的文件，那该怎么办呢？

为了进行一个这样的查找，我们可以定义一个名为 filter 的通用型函数。就像之前看到的 map 那样，filter 函数接受一个函数作为参数。filter 函数的类型是 (Element) -> Bool —— 对于数组中的所有元素，此函数都会判定它是否应该被包含在结果中：

```
extension Array {  
    func filter(_ includeElement: (Element) -> Bool) -> [Element] {  
        var result: [Element] = []  
        for x in self where includeElement(x) {  
            result.append(x)  
        }  
        return result  
    }  
}
```

根据 filter 能很容易地定义 getSwiftFiles：

```
func getSwiftFiles2(in files: [String]) -> [String] {  
    return files.filter { file in file.hasSuffix(".swift") }  
}
```

就像 map 一样，Swift 标准库中的数组类型已经有定义好的 filter 函数了。所以除非是作为练习，否则并没有必要重写它。

现在你可能会问：有没有更通用的函数，既可以用来定义 map，又可以用来定义 filter？关于这个问题，我们将在本章的最后解答。

Reduce

一如既往地，在定义一个泛型函数来体现一个更常见的模式之前，我们会先考虑一些相对简单的函数。

定义一个计算数组中所有整型值之和的函数非常简单：

```
func sum(integers: [Int]) -> Int {  
    var result: Int = 0  
    for x in integers {  
        result += x  
    }  
    return result  
}
```

下面是一个使用 `sum` 函数的例子：

```
sum(integers: [1, 2, 3, 4]) // 10
```

我们也可以使用类似 `sum` 中的 `for` 循环来定义一个 `product` 函数，用于计算所有数组项相乘之积：

```
func product(integers: [Int]) -> Int {  
    var result: Int = 1  
    for x in integers {  
        result = x * result  
    }  
    return result  
}
```

同样地，我们可能想要连接数组中的所有字符串：

```
func concatenate(strings: [String]) -> String {  
    var result: String = ""  
    for string in strings {  
        result += string  
    }  
    return result  
}
```

或者说，我们可以选择连接数组中的所有字符串，并插入一个单独的首行，以及在每一项后面追加一个换行符：

```
func prettyPrint(strings: [String]) -> String {  
    var result: String = "Entries in the array xs:\n"  
    for string in strings {  
        result = " " + result + string + "\n"  
    }  
    return result  
}
```

这些函数有什么共同点呢？它们都将变量 `result` 初始化为某个值。随后对输入数组的每一项进行遍历，最后以某种方式更新结果。为了定义一个可以体现所需类型的泛型函数，我们需要对两份信息进行抽象：赋给 `result` 变量的初始值，和用于在每一次循环中更新 `result` 的函数。

我们可以通过下述 `reduce` 函数的定义来实现这种模式：

```
extension Array {  
    func reduce<T>(_ initial:T, combine:(T, Element) -> T) -> T {  
        var result = initial  
        for x in self {  
            result = combine(result, x)  
        }  
        return result  
    }  
}
```

该函数的泛型体现在两个方面：对于任意 [Element] 类型的输入数组来说，它会计算一个类型为 T 的返回值。这么做的前提是，首先需要一个 T 类型的初始值（赋给 `result` 变量），以及一个用于更新 for 循环中变量值的函数 `combine: (T, Element) -> T`。在一些像 OCaml 和 Haskell 这样的函数式语言中，`reduce` 函数被称为 `fold` 或 `fold_left`

我们可以用 `reduce` 来定义迄今为止本章出现的所有函数。下面是几个例子：

```
func sumUsingReduce(integers: [Int]) -> Int {  
    return integers.reduce(0) { result, x in result + x }  
}
```

除了写一个闭包，我们也可以将运算符作为最后一个参数。这使得代码更短，如下面两个函数所示：

```
func productUsingReduce(integers: [Int]) -> Int {  
    return integers.reduce(1, combine: *)  
}  
  
func concatUsingReduce(strings: [String]) -> String {  
    return strings.reduce("", combine: +)  
}
```

要再一次说明，我们自定义 `reduce` 仅仅只是为了练习。Swift 的标准库已经为数组提供了 `reduce` 函数。

我们可以使用 `reduce` 来定义新的泛型函数。例如，假设有一个数组，它的每一项都是数组，而我们想将它展开为一个单一数组。可以使用 `for` 循环编写一个函数：

```
func flatten<T>(_xss: [[T]]) -> [T] {
    var result: [T] = []
    for xs in xss {
        result += xs
    }
    return result
}
```

然而，若使用 `reduce` 则可以像下面这样编写这个函数：

```
func flattenUsingReduce<T>(_xss: [[T]]) -> [T] {
    return xss.reduce([]) { result, xs in result + xs }
}
```

实际上，我们甚至可以使用 `reduce` 重新定义 `map` 和 `filter`：

```
extension Array {
    func mapUsingReduce<T>(_transform: (Element) -> T) -> [T] {
        return reduce([]) { result, x in
            return result + [_transform(x)]
        }
    }

    func filterUsingReduce(_includeElement: (Element) -> Bool) -> [Element] {
        return reduce([]) { result, x in
            return _includeElement(x) ? result + [x] : result
        }
    }
}
```

我们能够使用 `reduce` 来表示所有这些函数，这个事实说明了 `reduce` 能够通过通用的方法来体现一个相当常见的编程模式：遍历数组并计算结果。

请务必注意：尽管通过 `reduce` 来定义一切是个很有趣的练习，但是在实践中这往往不是一个什么好主意。原因在于，不出意外的话你的代码最终会在运行期间大量复制生成的数组，换句话说，它会反复分配内存，释放内存，以及复制大量内存中的内容。
比如说，用一个可变结果数组来编写 `map` 的效率显然会更高。理论上，编译器可以优

化上述代码，使其速度与可变结果数组的版本一样快，但是 Swift (目前) 并没有那么做。如果想了解更多详情，请参阅我们的另一本书——[《Swift 进阶》](#)。

实际运用

渐渐进入了本章的尾声，我们将提供一个使用了 `map`、`filter` 和 `reduce` 的小实例。

假设我们有下面这样的结构体，其定义由城市的名字和人口 (单位为千居民) 组成：

```
struct City {  
    let name: String  
    let population: Int  
}
```

我们可以定义一些示例城市：

```
let paris = City(name: "Paris", population: 2241)  
let madrid = City(name: "Madrid", population: 3165)  
let amsterdam = City(name: "Amsterdam", population: 827)  
let berlin = City(name: "Berlin", population: 3562)  
  
let cities = [paris, madrid, amsterdam, berlin]
```

假设我们现在想筛选出居民数量至少一百万的城市，并打印一份这些城市的名字及总人口数的列表。我们可以定义一个辅助函数来换算居民数量：

```
extension City {  
    func scalingPopulation() -> City {  
        return City(name: name, population: population * 1000)  
    }  
}
```

现在我们可以使用所有本章中见到的函数来编写下面的语句：

```
cities.filter { $0.population > 1000 }  
.map { $0.scalingPopulation() }  
.reduce("City: Population") { result, c in
```

```
        return result + "\n" + "\(c.name): \(c.population)"
    }
/*
City: Population
Paris: 2241000
Madrid: 3165000
Berlin: 3562000
*/

```

我们首先将居民数量少于一百万的城市过滤掉。然后将剩下的结果通过 `scalingPopulation` 函数进行 `map` 操作。最后，使用 `reduce` 函数来构建一个包含城市名字和人口数量列表的 `String`。这里我们使用了 Swift 标准库中 `Array` 类型的 `map`、`filter` 和 `reduce` 定义。于是，我们可以顺利地链式使用过滤和映射的结果。表达式 `cities.filter(..)` 的结果是一个数组，对其调用 `map`；然后这个返回值调用 `reduce` 即可得到最终结果。

泛型和 `Any` 类型

除了泛型，Swift 还支持 `Any` 类型，它能代表任何类型的值。从表面上看，这好像和泛型极其相似。`Any` 类型和泛型两者都能用于定义接受两个不同类型参数的函数。然而，理解两者之间的区别至关重要：泛型可以用于定义灵活的函数，类型检查仍然由编译器负责；而 `Any` 类型则可以避开 Swift 的类型系统（所以应该尽可能避免使用）。

让我们考虑一个最简单的例子，构想一个函数，除了返回它的参数，其它什么也不做。如果使用泛型，我们可能写为下面这样：

```
func noOp<T>(_ x:T) -> T {
    return x
}
```

而使用 `Any` 类型，则可能写为这样：

```
func noOpAny(_ x:Any) -> Any {
    return x
}
```

`noOp` 和 `noOpAny` 两者都将接受任意参数。关键的区别在于我们所知道的返回值。在 `noOp` 的定义中，我们可以清楚地看到返回值和输入值完全一样。而 `noOpAny` 的例子则不太一样，返回

值是任意类型 — 甚至可以是和原来的输入值不同的类型。我们可以给出一个 noOpAny 的错误定义，如下所示：

```
func noOpAnyWrong(_ x: Any) -> Any {  
    return 0  
}
```

使用 Any 类型可以避开 Swift 的类型系统。然而，尝试将使用泛型定义的 noOp 函数返回值设为 0 将会导致类型错误。此外，任何调用 noOpAny 的函数都不知道返回值会被转换为何种类型。而结果就是可能导致各种各样的运行时错误。

最后不得不说，泛型函数的**类型**真的是十分丰富。不妨考虑一下我们在上一章节封装 Core Image 中定义的函数组合运算符 `>>>` 的泛型版本：

```
infix operator >>>  
func >>> <A, B, C>(f: @escaping (A) -> B, g: @escaping (B) -> C) -> (A) -> C {  
    return {x in g(f(x))}  
}
```

这个函数的类型极具通用性，以至于它完全决定了**函数自身**被定义的形式。这里我们会尝试对此进行一些不太正式的说明和讨论。

我们需要得到的是一个 C 类型的值。由于我们并不知道其它任何有关 C 的信息，所以暂时没有能够返回的值。如果知道 C 是像 Int 或者 Bool 这样的具体类型的话，我们就可以返回一个该类型的值，例如分别返回 5 或 true。但是由于我们的函数必须要能处理**任意**类型的 C，所以不能轻率地返回具体值。在 `>>>` 运算符的参数中，只有 g: (B) -> C 函数提及了类型 C。因此，将 B 类型的值传递给函数 g 是我们能够触及类型 C 的唯一途径。

同样，要得到一个 B 类型值的唯一方法是将类型为 A 的值传递给 f。类型为 A 的值唯一出现的地方是在我们运算符要求返回的函数的输入参数里。因此，函数组合的定义只有这唯一一种可能，才能满足所要求的泛型类型。

我们可以用相同的方式定义一个泛型函数，该函数能够将任意的接受两个元素的元组作为输入的函数进行柯里化 (curry) 处理，从而生成相应的柯里化版本：

```
func curry<A, B, C>(_ f: @escaping (A, B) -> C) -> (A) -> (B) -> C {  
    return {x in {y in f(x, y)}}  
}
```

我们不再需要像我们在上一章中做过的那样，对同样的函数定义柯里化与未柯里化两个不同的版本。换句话说，像 curry 一样的泛型函数可以被用于函数变换 —— 由未柯里化版本变换为柯里化版本。同样地，这个函数的类型非常通用，它（几乎）给出了一个完整的函数设计：确实只有一种合理的实现方式。

使用泛型允许你无需牺牲类型安全就能够在编译器的帮助下写出灵活的函数；如果使用 Any 类型，那你就真的就孤立无援了。

注释

泛型的历史可以追溯到 Strachey (2000), Girard 的系统 F (1972) 和 Reynolds (1974)。务必注意，这些作者将泛型称为（参数）多态 (polymorphism)，该术语仍然被用在很多函数式语言中。而许多面向对象语言用多态这个术语指代子类型引起的隐式类型转换，因此泛型这个词的引入是为了消除两个概念之间的歧义。

在前面我们进行了一些非正式的讨论，说明了为什么这样的泛型类型只有一种可能的对应函数：

$(f: (A) \rightarrow B, g: (B) \rightarrow C) \rightarrow (A) \rightarrow C$

这其实可以用数学方法来进行更精确的解释。Reynolds (1983) 首先完成了这项工作，而后来 Wadler (1989) 将其称为**自然推断** (Theorems for free!) —— 它强调你可以通过泛型函数的类型来推断出函数的内容。

可选值

5

Swift 的**可选类型**可以用来表示可能缺失或是计算失败的值。本章将介绍如何有效地利用可选类型，以及它们在函数式编程范式中的使用方式。

案例研究：字典

除了数组，Swift 对**字典**也有特别的支持。字典是键值对的集合，它提供了一个有效的方法来查询与某个键关联的值。创建字典的语法与创建数组类似：

```
let cities = ["Paris": 2241, "Madrid": 3165, "Amsterdam": 827, "Berlin": 3562]
```

上述字典存储了几个欧洲城市的人口数量。在这个例子中，键 "Paris" 与值 2241 相关联；也就是说，Paris 的居民数量约为 2,241,000。

和数组一样，Dictionary 支持泛型。字典类型接受两种类型参数，分别对应所存储的键和值。在我们的例子中，城市字典的类型是 Dictionary<String, Int>。还有一种简写形式，[String: Int]。

我们可以使用下标来查询与键相关联的值：

```
let madridPopulation: Int = cities["Madrid"]  
// 结果为: Error: Value of optional type 'Int?' not unwrapped
```

然而，这个例子无法通过类型检查。问题在于 "Madrid" 键可能并不存在于 cities 字典里——当不存在时应该返回什么值呢？我们无法保证字典的查询操作**总是**为每个键返回一个 Int 值。Swift 的**可选类型**可以表达这种失败的可能性。编写这个例子的正确方式应该如下所示：

```
let madridPopulation: Int? = cities["Madrid"]
```

例子中 madridPopulation 的类型是可选类型 Int?，而非 Int。一个 Int? 类型的值是 Int 或者特殊的“缺失”值 nil。

我们可以检验查询是否成功：

```
if madridPopulation != nil {  
    print("The population of Madrid is \(madridPopulation! * 1000)")  
} else {  
    print("Unknown city: Madrid")  
}
```

如果 `madridPopulation` 不是 `nil`, 第一个分支就会被执行。这里我们使用后缀运算符 `!` 来取得 `madridPopulation` 中实际的 `Int` 值。不过要注意, 这个做法并不安全: 一旦 `madridPopulation` 是 `nil`, 这段代码就会导致程序崩溃。在所举的例子中, 我们在取值前先确保了 `madridPopulation` 不是 `nil`, 但是在后面可能稍不留意会打破这个假定。

Swift 有一个特殊的可选绑定 (optional binding) 机制, 能够让你避免写 `! 后缀`。于是我们可以将 `madridPopulation` 的定义和上面的检验语句相结合, 使它成为一个新语句:

```
if let madridPopulation = cities["Madrid"] {  
    print("The population of Madrid is \(madridPopulation * 1000)")  
} else {  
    print("Unknown city: Madrid")  
}
```

如果查询 `cities["Madrid"]` 是成功的, 我们便可以在分支中使用 `Int` 类型的变量 `madridPopulation`。值得注意的是, 我们不再需要显式地使用强制解包 (forced unwrapping) 运算符。

如果可以选择, 我们建议使用可选绑定而非强制解包。如果你有一个 `nil` 值, 强制解包可能导致崩溃; 可选绑定鼓励你显式地处理异常情况, 从而避免运行时错误。可选类型未经检验进行的强制解包, 或者 Swift 的隐式解包可选值, 都是很糟的代码异味, 它们预示着可能发生运行时错误。

Swift 还给 `!` 运算符提供了一个更安全的替代, `??` 运算符。使用这个运算符时, 需要额外提供一个默认值, 当运算符被运用于 `nil` 时, 这个默认值将被作为返回值。简单来说, 它可以定义为下面这样:

infix operator ??

```
func ??<T>(optional:T?, defaultValue:T) -> T {  
    if let x = optional {  
        return x  
    } else {  
        return defaultValue  
    }  
}
```

`??` 运算符会检验它的可选参数是否为 `nil`。如果是, 返回 `defaultValue` 参数; 否则, 返回可选值中实际的值。

上面的定义有一个问题：如果 `defaultValue` 的值是通过某个函数或者表达式计算得到的，那么无论可选值是否为 `nil`, `defaultValue` 都会被求值。通常我们并不希望这种情况发生：一个 `if-then-else` 语句应该根据各分支关联的值是否为真，只执行其中一个分支。这种行为有时候被称为短路，与 `|| and &&` 的工作原理相似。同样的道理，`??` 运算符应该只在可选值参数是 `nil` 时才对 `defaultValue` 参数进行求值。举个例子，假设我们像下面这样调用 `??`:

```
let cache = ["test.swift": 1000]
let defaultValue = 2000 //本地读取
cache["hello.swift"] ?? defaultValue
```

在这个例子中，如果可选值是非 `nil` 的话，我们真的不愿意对 `defaultValue` 进行求值——因为这可能是一个开销非常大的计算，只有绝对必要时我们才会想运行这段代码。可以按如下方式解决这个问题：

```
func ??<T>(optional: T?, defaultValue: () -> T) -> T {
    if let x = optional {
        return x
    } else {
        return defaultValue()
    }
}
```

作为 `T` 类型的替代，我们提供一个 `() -> T` 类型的默认值。现在 `defaultValue` 函数中的代码只在 `else` 分支中会被执行。美中不足的是，当调用 `??` 运算符时需要在匿名函数中对默认值进行封装。就是说，我们需要编写以下代码：

```
myOptional ?? { myDefaultValue }
```

Swift 标准库中的定义通过使用 Swift 的 `autoclosure` 类型标签来避开创建显式闭包的需求。它会在所需要的闭包中隐式地将参数封装到 `??` 运算符。这样一来，我们能够提供与最初相同的接口，但是用户无需再显式地创建闭包封装 `defaultValue` 参数。Swift 标准库中使用的定义如下：

```
infix operator ??

func ??<T>(optional: T?, defaultValue: @autoclosure () throws -> T)
    rethrows -> T
{
    if let x = optional {
        return x
    } else {
```

```
    } else {
        return try defaultValue()
    }
}
```

`??` 运算符提供了一个相较于强制可选解包更安全的替代，并且不像可选绑定一样繁琐。

玩转可选值

Swift 的可选值可以使失败的情况直截了当。虽然这在有些场合，特别是当多个可选结果组合在一起的时候，写起来会有些麻烦。但事实上，有很多技术能让可选值更易用。

可选链

首先，Swift 有一个特殊的机制，**可选链**，它用于在被嵌套的类或结构体中调用方法或访问属性。让我们来考虑一下处理客户订单的简单模型：

```
struct Order {
    let orderNumber: Int
    let person: Person?
}

struct Person {
    let name: String
    let address: Address?
}

struct Address {
    let streetName: String
    let city: String
    let state: String?
}

let order = Order(orderNumber: 42, person: nil)
```

给定一个 `Order`，如何才能知道客户地址中的 `state` 为何值呢？我们可以使用显式解包运算符：

```
order.person!.address!.state!
```

然而，如果任意中间数据缺失，这么做可能会导致运行时异常。使用可选绑定相对更安全：

```
if let person = order.person {  
    if let address = person.address {  
        if let state = address.state {  
            print("Got a state: \(state)")  
        }  
    }  
}
```

但这未免有些烦琐，而且我们至今也没有处理其他情况。若使用可选链，这个例子将会变成：

```
if let myState = order.person?.address?.state {  
    print("This order will be shipped to \(myState)")  
} else {  
    print("Unknown person, address, or state.")  
}
```

我们使用了问号运算符来尝试对可选类型进行解包，而不是强制将它们解包。访问任意属性失败时，都将会导致整条语句链返回 nil。

分支上的可选值

上面我们已经讨论了 if let 可选绑定机制，但是 Swift 还有其他两种分支语句，switch 和 guard，它们也非常适合与可选值搭配使用。

为了在一个 switch 语句中匹配可选值，我们简单地为 case 分支中的每个模式添加一个 ? 后缀：

```
switch madridPopulation {  
    case 0?: print("Nobody in Madrid")  
    case (1..<1000)?: print("Less than a million in Madrid")  
    case let x?: print("\(\(x)) people in Madrid")  
    case nil: print("We don't know about Madrid")  
}
```

`guard` 语句的设计旨在当一些条件不满足时，可以尽早退出当前作用域。可以将它和可选绑定组合在一起处理没有值存在的情况，这个使用情境非常常见。很显然，`guard` 语句后面的任何代码都需要值存在才会被执行。举个例子，我们可以重写打印给定城市居民数量的代码，如下所示：

```
func populationDescription(for city: String) -> String? {
    guard let population = cities[city] else { return nil }
    return "The population of Madrid is \(population)"
}

populationDescription(for: "Madrid")
// Optional("The population of Madrid is 3165")
```

在 `guard` 语句后面，我们有非可选的 `population` 值可以使用。以这种方式使用 `guard` 语句，会让控制流比嵌套 `if let` 语句时更简单。

可选映射

? 运算符允许我们选择性地访问可选值的方法或字段。然而，在很多其它例子中，若可选值存在，你可能会想操作它，否则返回 `nil`。参考下面的例子：

```
func increment(optional: Int?) -> Int? {
    guard let x = optional else { return nil }
    return x + 1
}
```

例子 `increment(optional:)` 的行为与 ? 运算符相似：如果可选值是 `nil`，则结果也是 `nil`；不然就执行一些计算。

我们可以将 `increment(optional:)` 函数和 ? 运算符一般化，然后为可选值定义一个 `map` 函数。这样一来，我们不仅能像 `increment(optional:)` 那样，对一个 `Int?` 类型的值做增量运算，还可以将想任何要执行的运算作为参数传递给 `map` 函数：

```
extension Optional {
    func map<U>(_ transform: (Wrapped) -> U) -> U? {
        guard let x = self else { return nil }
        return transform(x)
    }
}
```

```
}
```

map 函数接受一个类型为 (Wrapped) -> U 的 transform 函数作为参数。如果可选值不是 nil，map 将会将其作为参数来调用 transform，并返回结果；否则 map 函数将返回 nil。这个 map 函数是 Swift 标准库的一部分。

我们可以使用 map 来重写 increment(optional:)，如下所示：

```
func increment(optional: Int?) -> Int? {
    return optional.map { $0 + 1 }
}
```

当然，我们也可以使用 map 来访问和操作可选值结构体或者类中的字段或方法，就像我们使用 ? 运算符时所做的那样。

为什么将这个函数命名为 map？它和运用于数组的 map 运算有什么共同点吗？我们有充分的理由将这两个函数都称为 map，但是现在我们暂时不会展开，之后在关于函子、适用函子与单子的章节中会再次讨论这个问题。

再谈可选绑定

map 函数展示了一种操作可选值的方法，但是还有很多其它方法存在。参考下面的例子：

```
let x: Int? = 3
let y: Int? = nil
let z: Int? = x + y
```

这段程序并不被 Swift 编译器接受，你能定位到错误吗？

这里的问题是加法运算只支持 Int 值，而不支持我们这里的 Int? 值。要解决这个问题，我们可以像下面这样引入 if 嵌套语句：

```
func add(_ optionalX: Int?, _ optionalY: Int?) -> Int? {
    if let x = optionalX {
        if let y = optionalY {
            return x + y
        }
    }
}
```

```
    }
    return nil
}
```

除了层层嵌套，我们还可以同时绑定多个可选：

```
func add2(_ optionalX: Int?, _ optionalY: Int?) -> Int? {
    if let x = optionalX, let y = optionalY {
        return x + y
    }
    return nil
}
```

若还想更简短，可以使用一个 `guard` 语句，当值缺失时提前退出：

```
func add3(_ optionalX: Int?, _ optionalY: Int?) -> Int? {
    guard let x = optionalX, let y = optionalY else { return nil }
    return x + y
}
```

这个例子可能看起来很生硬，不过操作可选值确实是常常发生的事。假设我们定义了下面这个字典，国家与其首都相关联：

```
let capitals = [
    "France": "Paris",
    "Spain": "Madrid",
    "The Netherlands": "Amsterdam",
    "Belgium": "Brussels"
]
```

为了编写一个能返回给定国家首都人口数量的函数，我们将 `capitals` 字典与之前定义的的 `cities` 字典结合。对于每一次字典查询，我们必须确保它返回一个结果：

```
func populationOfCapital(country: String) -> Int? {
    guard let capital = capitals[country], let population = cities[capital]
        else { return nil }
    return population * 1000
}
```

可选链和 if let (或 guard let) 都是语言中让可选值能够更易于使用的特殊构造。不过，Swift 还提供了另一条途径来解决上述问题：那就是借力于标准库中的 flatMap 函数。多种类型中都定义了flatMap 函数，在可选值类型的情况下，它的定义是这样的：

```
extension Optional {  
    func flatMap<U>(_ transform: (Wrapped) -> U?) -> U? {  
        guard let x = self else { return nil }  
        return transform(x)  
    }  
}
```

flatMap 函数检查一个可选值是否为 nil。若不是，我们将其传递给参数函数 transform；若是，那么结果也将是 nil。

现在我们可以使用此函数，来重写前面的例子：

```
func add4(_ optionalX: Int?, _ optionalY: Int?) -> Int? {  
    return optionalX.flatMap { x in  
        optionalY.flatMap { y in  
            return x + y  
        }  
    }  
}  
  
func populationOfCapital2(country: String) -> Int? {  
    return capitals[country].flatMap { capital in  
        cities[capital].flatMap { population in  
            population * 1000  
        }  
    }  
}
```

当前我们通过嵌套的方式调用 flatMap，取而代之，也可以通过链式调用来重写 populationOfCapital2，这样能使得代码结构更浅显易懂：

```
func populationOfCapital3(country: String) -> Int? {  
    return capitals[country].flatMap { capital in  
        cities[capital]  
    }.flatMap { population in
```

```
    population * 1000
}
}
```

我们并非想要鼓吹 flatMap 是组合可选值的“正确”方法。仅仅只是希望说明 Swift 编译器内置的可选绑定并不神奇，它不过是一种你能够使用高阶函数自己实现的控制结构。

为什么使用可选值？

引入一个显式可选类型的意义是什么呢？对于习惯了 Objective-C 的程序员来说，最初使用可选类型也许会觉得奇怪。Swift 的类型系统相当严格：一旦我们有可选类型，就必须处理它可能是 nil 的问题。于是不得不编写像 map 一样的新函数来操作可选值。在 Objective-C 中，这一切更灵活。举个例子来说，将上面的例子翻译为下面的 Objective-C 代码，将不会有任何编译错误：

```
- (int)populationOfCapital:(NSString *)country
{
    return [self.cities[self.capitals[country]] intValue] * 1000;
}
```

我们可以将 nil 作为一个国家的名字传递给函数，然后得到一个结果 0。一切都很顺利。在许多语言中并没有可选，空指针是危险的来源。Objective-C 稍好一些，你可以安全地向 nil 发送消息，然后根据不同的返回值的类型，得到像是 nil、数字 0 这样的零值。为什么在 Swift 中要改变这种特性呢？

选择显式的可选类型更符合 Swift 增强静态安全的特性。强大的类型系统能在代码执行前捕获到错误，而且显式可选类型有助于避免由缺失值导致的意外崩溃。

Objective-C 采用的默认取零的做法有其弊端，可能你会想要区分失败的字典查询 (键不存在于字典) 和成功但返回 nil 的字典查询 (键存在于字典，但关联值是 nil) 两种情况。若要在 Objective-C 中做到这一点，你只能使用 NSNull。

虽然在 Objective-C 中将消息发送给 nil 是安全的，但是使用它们往往并不安全。

比方说我们想创建一个属性字符串 (attributed string)。如果我们传递 nil 作为 country 的值，那么 capital 也会是 nil，但是当我们试图将 NSAttributedString 初始化为 nil 时，将会引起崩溃：

```
- (NSAttributedString *)attributedCapital:(NSString *)country
{
    NSString *capital = self.capitals[country];
    NSDictionary *attr = @{}; // ...
    return [[NSAttributedString alloc] initWithString:capital attributes:attr];
}
```

虽然像上面那样的崩溃并不经常发生，不过几乎每个开发者都写过像这样导致崩溃的代码。大多数时候，在调试阶段这些崩溃的导火索就会被发现，不过偶尔难免不知不觉就发布了代码，一些情况下，变量还可能出乎意料的是 nil。因此，许多程序员使用断言来显式地标识这这种情况。例如，我们可以添加一个 NSParameterAssert 以确保当 country 是 nil 时就立即崩溃：

```
- (NSAttributedString *)attributedCapital:(NSString *)country
{
    NSParameterAssert(country);
    NSString *capital = self.capitals[country];
    NSDictionary *attr = @{}; // ...
    return [[NSAttributedString alloc] initWithString:capital attributes:attr];
}
```

如果我们传递了一个 country 值，但是 self.capitals 中并没有键与之匹配该怎么办呢？这种情况发生的概率很大，特别是当 country 来源于用户输入时。在这种情况下，capital 将会是 nil，我们的代码仍然会崩溃。当然，修复的方法很简单。不过，我们关注的重点是，在 Swift 中使用 nil 编写**健壮的**代码比在 Objective-C 中容易。

最后，从本质上讲，使用断言是非模块化的。假设我们要实现一个 checkCountry 方法用于确认是否支持非空 NSString *。可以很容易地在上述代码中加入该方法：

```
- (NSAttributedString *)attributedCapital:(NSString*)country
{
    NSParameterAssert(country);
    if (checkCountry(country)) {
        // ...
    }
}
```

问题来了：checkCountry 函数是否也应该断言它的参数不是 nil？一方面，它不应该——因为我们方才刚在 attributedCapital 方法中执行了检验的代码。另一方面，如果 checkCountry 函数仅适用于非 nil 值，我们还是应该复制上述断言。我们被迫在暴露不安全接口和复制断言

之间进行选择。还有一种做法是，可以给签名添加一个 `nonnull` 标注，当该方法被一个可能为 `nil` 的值调用时，它会发出警告，但这种做法在大多数 Objective-C 的代码库中并不常见。

在 Swift 中，事情要来得容易得多：函数签名可以显式地使用可选值来提示一个值可能为 `nil`。与他人协同编码时，这是十分宝贵的信息。下面的签名提供了大量信息：

```
func attributedCapital(country: String) -> NSAttributedString?
```

我们不仅被警告有失败的可能性，还知道了必须传递一个 `String` 作为参数 —— 且不能是 `nil` 值。像上面一样的崩溃将不会再发生。此外，这也是编译器要检验的信息。文档很容易过时，但是你可以永远依赖函数签名。

在 Objective-C 中处理标量值时，可选问题显得更加棘手。不妨看看下面的示例，尝试在一个字符串中查找一个特定关键词的位置：

```
NSString *someString = ...;
if ([someString rangeOfString:@"swift"].location != NSNotFound) {
    NSLog(@"Someone mentioned swift!");
}
```

看起来毫无问题：如果 `rangeOfString:` 没有找到字符串，`location` 就会被设置为 `NSNotFound`。`NSNotFound` 被定义为 `NSIntegerMax`。这段代码几乎是正确的，第一眼很难看到它的问题：若 `someString` 是 `nil`，`rangeOfString:` 将返回一个属性全为零的结构体，`location` 将返回 0。接着所做的判断结果若为真，`if` 语句中的代码将被执行。

如果有可选值的话，这一切将免于发生。如果我们想将这份代码转为 Swift，会需要做出一些结构性的改变。上面的代码会被编译器拒绝，类型系统也不会允许你在一个 `nil` 值上运行 `rangeOfString:`。因此，首先需要将它解包：

```
if let someString = ...,
    someString.rangeOfString("swift").location != NSNotFound {
    print("Found")
}
```

类型系统将有助于你捕捉难以察觉的细微错误。其中一些错误很容易在开发过程中被发现，但是其余的可能会一直留存到生产代码中去。坚持使用可选值能够从根本上杜绝这类错误。

案例研究： QuickCheck

6

近些年来，在 Objective-C 中，测试变得越来越普遍。现在有许多流行的库通过持续集成工具来进行自动化测试。XCTest 是用来写单元测试的标准框架。此外，很多第三方框架 (例如 Specta、Kiwi 和 FBSnapshotTestCase) 也已经可供使用，同时现阶段还有若干 Swift 的新框架正在开发中。

所有的这些框架都遵循一个相似的模式：测试通常由一些代码片段和预期结果组成。执行代码之后，将它的结果与测试中定义的预期结果相比较。不同的库测试的层次会有所不同——有的测试独立的方法，有的测试类，还有一些执行集成测试 (运行整个应用)。在本章中，我们将通过迭代的方法，一步一步完善并最终构建一个针对 Swift 函数进行“特性测试 (property-based testing)”的小型库。

在写单元测试的时候，输入数据是程序员定义的静态数据。比方说，在对一个加法进行单元测试时，我们可能会写一个验证 $1 + 1$ 等于 2 的测试。如果加法的实现方式破坏了这个特性，测试就会失败。不过，为了更一般化，我们可以选择测试加法的交换律——换句话说，就是验证 $a + b$ 等于 $b + a$ 。为了进行这项测试，我们可以写一个测试用例来验证 $42 + 7$ 等于 $7 + 42$ 。

QuickCheck (Claessen and Hughes 2000) 是一个用于随机测试的 Haskell 库。相较于独立的单元测试中每个部分都依赖特定输入来测试函数是否正确，QuickCheck 允许你描述函数的抽象特性并生成测试来验证这些特性。当一个特性通过了测试，就没有必要再证明它的正确性。更确切地说，QuickCheck 旨在找到证明特性错误的临界条件。在本章中，我们将用 Swift (部分地) 移植 QuickCheck 库。

这里举例说明会比较好。假设我们想要验证加法是一个满足交换律的运算。于是，首先为两个整型数值 x 和 y 写了一个函数，来检验 $x + y$ 与 $y + x$ 是否相等：

```
func plusIsCommutative(x: Int, y: Int) -> Bool {  
    return x + y == y + x  
}
```

用 QuickCheck 检验这条语句就像调用 check 函数一样简单：

```
check("Plus should be commutative", plusIsCommutative)  
// "Plus should be commutative" passed 10 tests.
```

check 函数一遍又一遍地调用 plusIsCommutative 函数且每次传递两个随机整型值作为参数，以此来完成上述检验。如果语句不为真，它将会打印出导致测试失败的输入值。这里的关键是，我们可以用返回 Bool 的函数 (如 plusIsCommutative) 来描述代码的抽象特性 (如交换律)。现在，check 函数使用这个特性来生成单元测试，这比起你自己手写的单元测试，代码覆盖率会更高。

当然，并不是所有的测试都能通过。例如，我们可以定义一个语句来描述减法满足交换律：

```
func minusIsCommutative(x: Int, y: Int) -> Bool {  
    return x - y == y - x  
}
```

现在，如果我们使用 QuickCheck 对这个函数进行测试，将会得到一个失败的测试用例：

```
check("Minus should be commutative", minusIsCommutative)  
// "Minus should be commutative" doesn't hold: (0, 1)
```

若使用 Swift 的尾随闭包 (trailing closures) 语法，我们可以直接编写测试，而无需单独定义（像 plusIsCommutative 或 minusIsCommutative 这样的）特性：

```
check("Additive identity") { (x: Int) in x + 0 == x }  
// "Additive identity" passed 10 tests.
```

当然，我们还能测试很多其它类似的标准算术特性。接下来，我们即将介绍更多有趣的测试和特性。不过在此之前，首先要给出一些关于如何实现 QuickCheck 的细节。

构建 QuickCheck

为了构建 Swift 版本的 QuickCheck，我们需要做几件事情：

- 首先，我们需要一个方法来生成不同类型的随机数。
- 有了随机数生成器之后，我们需要实现 check 函数，然后将随机数传递给它的特性参数。
- 如果一个测试失败了，我们会希望测试的输入值尽可能小。比方说，如果我们在对一个有 100 个元素的数组进行测试时失败了，我们会尝试让数组元素更少一些，然后看一看测试是否依然失败。
- 最后，我们还需要做一些额外的工作以确保检验函数适用于带有泛型的类型。

生成随机数

首先，让我们定义一个可以表达如何生成随机数的协议。这个协议只包含一个函数 —— 返回 Self 类型值的 arbitrary，返回的 Self 也就是实现了 Arbitrary 协议的这个类或结构体的实例：

```
protocol Arbitrary {
    static func arbitrary() -> Self
}
```

然后，让我们来写一个 Int 的例子。使用标准库中的 arc4random 函数并将其返回值转换为 Int。注意，这里只能生成正整数。事实上，一个完整实现的库也应能够生成负整数，不过本章中我们会尽可能让事情简单一些：

```
extension Int: Arbitrary {
    static func arbitrary() -> Int {
        return Int(arc4random())
    }
}
```

现在我们可以像下面这样生成随机整数：

```
Int.arbitrary() // 3314367665
```

为了避免随机整数越界，我们添加一个变量来对其进行约束：

```
extension Int {
    static func arbitrary(in range: CountableRange<Int>) -> Int {
        let diff = range.upperBound - range.lowerBound
        return range.lowerBound + (Int.arbitrary() % diff)
    }
}
```

如果要生成随机字符串，还需要再多做一点点工作。首先是生成随机字符：

```
extension UnicodeScalar: Arbitrary {
    static func arbitrary() -> UnicodeScalar {
        return UnicodeScalar(Int.arbitrary(in: 65..<90))!
    }
}
```

接下来我们先随机生成一个介于 0 到 40 之间的数，然后生成 randomLength 个随机字符，并将它们组合为一个字符串。注意，目前我们只随机生成大写字母：我们在这里限制了自身的原因除是，是我们希望本书的输出内容可读性更高。在实际的生产库中，应该生成包含任意字符且更长的字符串：

```
extension String: Arbitrary {  
    static func arbitrary() -> String {  
        let randomLength = Int.arbitrary(in: 0..<40)  
        let randomScalars = (0..            UnicodeScalar.arbitrary()  
        }  
        return String(UnicodeScalarView(randomScalars))  
    }  
}
```

如同我们生成随机 `Int` 类型值时所做的，我们可以用同样的方法调用 `arbitrary` 函数，唯一不同的是我们在 `String` 类上调用它：

```
String.arbitrary() // KVQMQYSNNFEJAXGPMIFCUHAKUHVIUKXLSHCE
```

实现 `check` 函数

现在，我们已经准备就绪，即将开始实现检验函数的第一个版本。`check1` 函数包含一个简单循环，每次迭代时为待检验特性生成随机的输入值，然后进行检验。一旦发现反例，就将其打印出来，并立即返回。否则 `check1` 函数将会汇报成功通过的测试数量。(注意，这里我们将函数称为 `check1` 是由于我们稍后将会编写其最终版。)

```
func check1<A: Arbitrary>(_ message: String, _ property: (A) -> Bool) -> () {  
    for _ in 0..<numberOflterations {  
        let value = A.arbitrary()  
        guard property(value) else {  
            print("\(message)\n" doesn't hold: \(value))  
            return  
        }  
    }  
    print("\(message)\n" passed \(numberOflterations) tests.)  
}
```

我们可以选择使用更函数式的风格，用 `reduce` 或 `map` 来编写这个函数，而非现在的 `for` 循环。不过，在本例中使用 `for` 循环合情合理：我们想要反复执行一个运算的次数是固定的，一旦发现反例就停止执行——对于这个过程，使用 `for` 循环十分合适。

下面是用这个函数来测试特性的方法：

```
extension CGSize {
    var area: CGFloat {
        return width * height
    }
}

extension CGSize: Arbitrary {
    static func arbitrary() -> CGSize {
        return CGSize(width: .arbitrary(), height: .arbitrary())
    }
}

check1("Area should be at least 0") { (size: CGSize) in size.area >= 0 }
/*
"Area should be at least 0" doesn't hold: CGSize(width:
-285.50077585631533, height: 9.5873849954426724)
*/
```

上面我们看到的例子充分地说明了何时 QuickCheck 会非常有用：它为我们找到了临界情况。如果尺寸有且只有一个负值，我们的 area 函数将返回一个负值。当被作为 CGRect 的一部分来使用时，CGSize 可以有负值。在编写一般的单元测试时，这种情况很容易被发现，因为尺寸通常只有正值。

缩小范围

如果我们在字符串上运行 check1 函数，可能会收到一条相当长的失败消息：

```
check1("Every string starts with Hello") { (s: String) in
    s.hasPrefix("Hello")
}

/** 打印:
"Every string starts with Hello" doesn't hold: "Very long string..."
```

理想情况下，我们希望失败的输入尽可能简单。通常，反例所处的范围越小，越容易定位到失败是由哪一段代码引起的。在上例中，反例还是相当易于理解的，但是不可能总是这种情况。想象一个复杂的状况，数组或字典因为不明原因失败了——如果有最小反例，判断为什么测试会失败将变得容易很多。原则上，用户可以尝试对失败的输入值进行缩减，并重新运行测试。然而，为了不将这个麻烦抛给用户，我们会将这个过程自动化。

首先，我们将额外定义一个名为 `Smaller` 的协议，它只做一件事——尝试缩小反例所处的范围：

```
protocol Smaller {
    func smaller() -> Self?
}
```

注意，`smaller` 函数的返回值类型被标记为了可选值。某些情况下，如何进一步缩小测试数据的范围这件事本身并不是很明确。例如，没有办法缩小空数组，这种情况我们将返回 `nil`。

在我们的例子中，对于整数，我们尝试将其除以二，直到等于零：

```
extension Int: Smaller {
    func smaller() -> Int? {
        return self == 0 ? nil : self / 2
    }
}
```

现在，让我们来测试一下上例：

```
100.smaller() // Optional(50)
```

而对于字符串，则是移除第一个字符 (除非该字符串为空)：

```
extension String: Smaller {
    func smaller() -> String? {
        return isEmpty ? nil : String(characters.dropFirst())
    }
}
```

为了在 `check` 函数中使用 `Smaller` 协议，我们需要一个方法，能够缩小 `check` 函数生成的任意测试数据的范围。于是，我们将重新定义 `Arbitrary` 协议以扩展 `Smaller` 协议：

```
protocol Arbitrary: Smaller {
    static func arbitrary() -> Self
}
```

反复缩小范围

我们现在可以重新定义 `check` 函数，来缩小任意导致失败的测试数据范围。为此，我们使用 `iterate(while:initial:next:)` 函数，它接受一个条件和一个初始值，并且只要条件成立就反复调用自身，我们用了递归的方式来实现它：

```
func iterate<A>(while condition: (A) -> Bool, initial: A, next: (A) -> A?) -> A {
    guard let x = next(initial), condition(x) else {
        return initial
    }
    return iterate(while: condition, initial: x, next: next)
}
```

通过使用 `iterate(while:initial:next:)`，我们能够反复缩小测试中发现的反例所属的范围：

```
func check2<A: Arbitrary>(_ message: String, _ property: (A) -> Bool) -> () {
    for _ in 0..
```

这个函数做了不少事情：生成随机输入值，再检验它们是否满足 `property` 参数，以及一旦发现反例，就反复缩小其范围。我们使用 `iterate(while:initial:next:)` 而非独立的循环来定义反复缩小方法，这样做的优点是能够让这段代码的控制流简单如初。

随机数组

目前，我们的 check2 函数只支持 Int 和 String。尽管我们可以自由地为其它像是 Bool 一样的类型定义新扩展，但是当我们想要生成随机数组时，这么做事情只会越加复杂。为了展开话题，让我们来编写一个函数式版本的快速排序：

```
func qsort(_ input: [Int]) -> [Int] {
    var array = input
    if array.isEmpty { return [] }
    let pivot = array.removeFirst()
    let lesser = array.filter { $0 < pivot }
    let greater = array.filter { $0 >= pivot }
    let intermediate = qsort(lesser) + [pivot]
    return intermediate + qsort(greater)
}
```

注意：不幸的是，由于 Swift 中存在一个 bug，上述的代码需要一个中间变量，详情可参考 <https://bugs.swift.org/browse/SR-1914>。

我们也可以试着编写一个特性来检验当前版本的快速排序相对于内置 sort 函数是否有区别：

```
check2("qsort should behave like sort") { (x: [Int]) in
    return qsort(x) == x.sorted()
}

/** 结果为：
Error
*/
```

然而，编译器警告我们 [Int] 没有遵循 Arbitrary 协议。在能够实现 Arbitrary 之前，我们还需要先实现 Smaller。首先，我们提供一个简单的函数来移除数组的最后一项：

```
extension Array: Smaller {
    func smaller() -> [Element]? {
        guard !isEmpty else { return nil }
        return Array(dropLast())
    }
}
```

我们也可以编写一个函数，它能够为任何遵循 Arbitrary 协议的类型生成一个随机长度的数组：

```
extension Array where Element: Arbitrary {  
    static func arbitrary() -> [Element] {  
        let randomLength = Int.arbitrary(in: 0..<50)  
        return (0..    }  
}
```

现在，我们想做的是让 Array 本身遵循 Arbitrary 协议。不过，只有数组的每一项都遵循 Arbitrary 协议，数组本身才会遵循 Arbitrary 协议。例如，为了生成一个由随机数组成的数组，我们首先需要确保能够生成随机数。理想情况下，我们将会像下面这样来表示数组的每一项都应该遵循 Arbitrary 协议：

```
extension Array: Arbitrary where Element: Arbitrary {  
    static func arbitrary() -> [Element] {  
        // ...  
    }  
}
```

很遗憾，目前还无法将这个限制表示为类型约束，并不可能编写一个让 Array 遵循 Arbitrary 协议的扩展。因此，我们选择修改 check2 函数。

check2<A> 函数的问题是它要求类型 A 遵循 Arbitrary 协议。我们将放弃这个需求，取而代之，要求必要的函数 smaller 和 arbitrary 被作为参数传入。

我们首先定义一个包含两个所需函数的辅助结构体：

```
struct ArbitraryInstance<T> {  
    let arbitrary: () -> T  
    let smaller: (T) -> T?  
}
```

现在，我们可以写一个接受 ArbitraryInstance 结构体作为参数的辅助函数。checkHelper 的定义严格参照了前面的 check2 函数。两者之间唯一的不同是 arbitrary 和 smaller 被定义的位置。在 check2 中，它们被泛型类型 <A: Arbitrary> 约束；而在 checkHelper 中，它们在 ArbitraryInstance 结构体中被显式地传递：

```

func checkHelper<A>(_ arbitraryInstance: ArbitraryInstance<A>,
    _ property: (A) -> Bool, _ message: String) -> ()
{
    for _ in 0..<numberOflterations {
        let value = arbitraryInstance.arbitrary()
        guard property(value) else {
            let smallerValue = iterate(while: { !property($0) },
                initial: value, next: arbitraryInstance.smaller)
            print("\(message)" " doesn't hold: \(smallerValue)")
            return
        }
    }
    print("\(message)" " passed \(numberOflterations) tests.")
}

```

这是一个标准方法：我们显式地将需要的信息作为参数进行传递，而非运用定义于协议中的函数。这么做，灵活性会更高。我们不再依赖 Swift 来推断需要的信息，而是完全自己来控制这一切。

我们可以使用 checkHelper 函数来重新定义 check2 函数。如果我们知道所需的 Arbitrary 定义，就可以将它们封装到 ArbitraryInstance 结构体中，然后调用 checkHelper：

```

func check<X: Arbitrary>(_ message: String, property: (X) -> Bool) -> () {
    let instance = ArbitraryInstance(arbitrary: X.arbitrary,
        smaller: { $0.smaller() })
    checkHelper(instance, property, message)
}

```

如果我们有一个类型，无法对它定义所需的 Arbitrary 实例，就像数组的情况一样，我们可以重载 check 函数并自己构造所需的 ArbitraryInstance 结构体：

```

func check<X: Arbitrary>(_ message: String, _ property: ([X]) -> Bool) -> () {
    let instance = ArbitraryInstance(arbitrary: Array.arbitrary,
        smaller: { (x: [X]) in x.smaller() })
    checkHelper(instance, property, message)
}

```

现在，我们终于可以运行 `check` 来验证我们所实现的快速排序了。大量的随机数组将会被生成并被传递给我们的测试：

```
check("qsort should behave like sort") { (x: [Int]) in
    return qsort(x) == x.sorted()
}
// "qsort should behave like sort" passed 10 tests.
```

使用 QuickCheck

也许出乎你的意料，不过有确凿的证据表明，测试技术会影响你的代码设计。依赖**测试驱动设计**的人们使用测试并不仅仅是为了验证他们的代码是否正确，他们还根据测试来指导编写测试驱动的代码，这样一来，代码的设计将会变得简单。这非常有意义——如果不需要复杂的构建流程就能够容易地为类编写测试代码的话，说明这个类的耦合度很低。

对于 QuickCheck 来说，同样的规则也是适用的。通常来看，事后向现有代码添加 QuickCheck 测试并不容易，尤其是当你有一个面向对象的架构且它很大程度依赖于其它类或使用可变状态的时候。但是，如果你从一开始就使用 QuickCheck 进行测试驱动开发，会发现它将给你的代码设计带来巨大的影响。

QuickCheck 迫使你去思考你的函数必须满足哪些抽象特性，并允许你给出一个高级规范。单元测试可以断言 $3 + 0$ 等于 $0 + 3$ ；QuickCheck 特性则更泛用地认为加法是可交换的运算。通过最初对一个高级 QuickCheck 规范的思考，你的代码会向着模块化和**引用透明** (将在下一章中提及) 的方向发展。对于有状态的函数或 API，QuickCheck 并不适用。因此，从一开始就使用 QuickCheck 来编写测试代码将有助于保持代码整洁。

展望

这个库已经很有用了，但是距离完成还有很大差距。也就是说，还有很多明显的地方可以改进：

- 缩小的方法很傻很天真。比方说，在数组的情况下，目前我们是移除数组的第一项。然而，我们完全可以选择移除其它项，或是对数组中的元素进行缩小(两者均做也可)。当前的实现方式返回一个可选且已经缩小范围的值，而我们可能想要生成一个由值组成的列表。在后面的章节中，我们将看到如何生成一个结果的惰性列表 (lazy list)，在那里我们可以使用相同的方法。

- Arbitrary 实例相当简单。为了对应不同的输入类型，我们可能想要更多复杂的 Arbitrary 实例。例如，当生成随机枚举值时，我们可以基于不同的频率来生成某种情况。我们也可以生成像是已排序的或是非空的数组这样的约束值。在编写多个 Arbitrary 实例的时候，可以定义一些辅助函数来协助我们。
- 将生成的测试数据进行分类：如果我们生成了大量长度为一的数组，可以将它们归类为“不重要的”测试数据。被参照的 Haskell 库本身是支持分类的，直接将这些理念移植过来即可。
- 我们也许会希望能更好地控制生成的随机输入值的个数。在 Haskell 版本的 QuickCheck 中，Arbitrary 协议接受一个额外的参数，用于限制随机输入值的个数；因此 check 函数一开始只测试“小”范围内的值，相当于小且快的测试。随着越来越多的测试通过，check 函数会增大输入值的范围并试图找到更大更复杂的反例。
- 我们可能想用确定的种子来初始化随机生成器，以便它能够重现测试用例所生成的值。这将会使失败的测试更容易被复现。

显然，这并不是全部；在让这个库变得完整的路上，还有很多其它大大小小的事情可以做。

目前，有很多实现了特性测试的 Swift 库。比如说 [SwiftCheck](#) 就是其中之一。如果你想在 Objective-C 上进行特性测试，可以考虑 [Fox](#)。

不可变性的价值

7

Swift 有一些机制，用于控制值的变化方式。在本章中，我们将介绍这些不同的机制是如何工作的，以及如何区别值类型和引用类型，并证明为什么限制可变状态的使用是一个良好的理念。

变量和引用

Swift 有两种初始化变量的方法，分别使用 `var` 和 `let` 关键字：

```
var x: Int = 1  
let y: Int = 2
```

两者的关键差异在于，我们可以给使用 `var` 声明的变量赋新值，而使用 `let` 创建的变量**不能被修改**：

```
x = 3 // 没问题  
y = 4 // 被编译器拒绝
```

使用 `let` 声明的变量被称为**不可变变量**；另一方面，使用 `var` 声明的变量则被叫做**可变变量**。

为什么？你可能想质问我——为什么要声明一个不可变的变量？这么做难道不会限制变量的能力吗？严格来说，一个可变变量用途更广泛。因为这个显而易见的理由，比起 `let` 我们更偏爱 `var`。不过，在本章中，我们会尝试证明事实恰恰相反。

不妨想象一下，如果要阅读一个他人编写的 Swift 类。其中有一些方法全都引用了某个名字毫无意义的实例变量，例如 `x`。如果可以选择，你会使用 `var` 还是 `let` 来声明 `x` 呢？显然，将 `x` 声明为不可变变量会更好：这样你可以通读代码而无需担心**当前** `x` 的值是什么，你可以在 `x` 的定义语句中自由地替换它的值，而不用担心给 `x` 赋一个新值时可能对类中其余部分的不可变性造成破坏。

不可变变量不能被赋以新值。因此，很容易知道不可变变量的行为。Edsger Dijkstra 在他一篇著名的论文《Go To 语句之害》(Go To Statement Considered Harmful) 中写道：

我的...观点是，以我们的智力，更适合掌控静态关系，而把随时间不断发展的过程形象化的能力相对不那么发达。

Dijkstra 接着论证了在阅读结构化代码 (使用条件语句，循环和函数调用，而非 goto 语句) 时，程序员需要具备的心智模型 (mental model) 比阅读充斥着 goto 的繁杂代码时所需的要简单。我们应该恪守这个信条，并再进一步，尽可能避开对可变变量的使用：var 是有害的。然而，就如我们将在下一章看到的一样，在 Swift 中，事情略有些微妙。

值类型与引用类型

不可变性并不只存在于变量声明中。Swift 类型分为**值类型**和**引用类型**。两者最典型的例子分别是结构体和类。为了阐明它们之间的区别，我们将定义下述结构体：

```
struct PointStruct {  
    var x: Int  
    var y: Int  
}
```

现在让我们来看看下面的代码片段：

```
var structPoint = PointStruct(x: 1, y: 2)  
var sameStructPoint = structPoint  
sameStructPoint.x = 3
```

在执行这段代码之后，很明显 sameStructPoint 等于 (x: 3, y: 2)。然而 structPoint 仍然保持原始值。这就是值类型与引用类型之间的关键区别：当被赋以一个新值或是作为参数传递给函数时，值类型会被复制。之所以给 sameStructPoint.x 赋值并不更新原来的 structPoint，正是因为先前的 sameStructPoint = structPoint 赋值过程中，发生了**值复制**。

为了进一步说明区别，我们可以声明一个点类：

```
class PointClass {  
    var x: Int  
    var y: Int  
  
    init(x: Int, y: Int) {  
        self.x = x  
        self.y = y  
    }  
}
```

然后修改上面的代码片段，使用类代替结构体：

```
var classPoint = PointClass(x: 1, y: 2)
var sameClassPoint = classPoint
sameClassPoint.x = 3
```

现在，给 `sameClassPoint.x` 赋值将会修改 `sameClassPoint` 变量所指向的那个对象，因为类是 **引用类型**。而 `classPoint` 变量指向同一个对象；因此，访问 `classPoint.x` 同样会返回新的值。

理解值类型与引用类型之间的区别极其重要，这可以让你预测赋值行为将会如何修改数据，同时确定哪一部分的代码可能受到这个改变的影响。

在调用函数的时候，值类型与引用类型之间的区别同样是显而易见的。不妨让我们看一看下面这个总是返回原点的函数：

```
func setStructToOrigin(point: PointStruct) -> PointStruct {
    var newPoint = point
    newPoint.x = 0
    newPoint.y = 0
    return newPoint
}
```

我们使用这个函数来生成一个点：

```
var structOrigin = setStructToOrigin(point: structPoint)
```

比如结构体这样的值类型，在作为函数的参数被传递时将会被复制后使用。因此，在这个例子中，调用 `setStructToOrigin` 之后，原来的 `structPoint` 并没有被修改。

现在假设我们编写了下面的函数，参数由结构体变为了类：

```
func setClassToOrigin(point: PointClass) -> PointClass {
    point.x = 0
    point.y = 0
    return point
}
```

于是下面的函数调用**将会修改** `classPoint`：

```
var classOrigin = setClassToOrigin(point: classPoint)
```

当把一个值类型赋值给新的变量，或者传递给函数时，值类型**总是**会被复制，而引用类型**并不**会被复制。对引用类型的对象来说，只有对于对象的**引用**会被复制，而不是对象本身。对于对象本身的任何修改都会在通过另一个引用访问相同对象时被反映出来：所以调用 `setClassToOrigin` 会影响 `classPoint` 和 `classOrigin`。

Swift 为结构体也提供了 `mutating` 方法，它们只能在被声明为 `var` 的结构体变量上被调用。作为例子，我们可以重写 `setStructToOrigin`：

```
extension PointStruct {  
    mutating func setStructToOrigin() {  
        x = 0  
        y = 0  
    }  
}
```

相较于对类进行操作的对应方法，结构体的 `mutating` 方法有其优势，它们不存在类似的副作用。一个 `mutating` 方法只作用于单一变量，完全不影响其它变量：

```
var myPoint = PointStruct(x: 100, y: 100)  
let otherPoint = myPoint  
myPoint.setStructToOrigin()  
otherPoint // PointStruct(x: 100, y: 100)  
myPoint // PointStruct(x: 0, y: 0)
```

正如我们所见，在上述代码中调用 `mutating` 方法十分安全，并且不影响任何其它变量。虽然这里使用 `let` 声明变量会让事情愈加简单，但是很多时候可变切实地提升了代码的可读性。在不存在全局副作用的情况下，Swift 结构体允许我们拥有局部可变性。

Andy Matuschak 在他给 [objc.io](#) 撰写的文章中对值类型与引用类型之间的区别进行讨论时，给出了一些十分有用且直观的例子。

结构体在 Swift 的标准库中遍地开花 — 例如，数组，字典，数字和布尔值全都被定义为结构体。此外，元组和枚举 (后者将在下一章介绍) 也是值类型。类 (class) 是一个例外。值类型在 Swift 标准库中的普遍用法正说明了 Swift 正在从面向对象编程进化到其他的编程范式。同样地，在 Swift 3 中，许多来自 Foundation 的类都存在一个对应的值类型。

结构体与类：究竟是否可变？

在上述例子中，我们使用 var 而非 let 将所有的 Point 类型及它们的属性声明为了可变变量。我们需要对由结构体和类等构造出的混合类型，以及它们与 var 和 let 声明的相互作用进行一些说明。

假如我们对 PointStruct 以不可变的方式进行了实例化，如下所示：

```
let immutablePoint = PointStruct(x: 0, y: 0)
```

很显然，给 immutablePoint 赋一个新值不会被接受：

```
immutablePoint = PointStruct(x: 1, y: 1) // 被拒绝
```

同样地，尝试给点的任意一个属性赋新值也会被拒绝，尽管在 PointStruct 中定义属性使用了 var，这是由于 immutablePoint 是通过 let 被定义的：

```
immutablePoint.x = 3 // 被拒绝
```

然而，如果我们将点声明为可变变量，则在初始化之后仍然可以修改它的属性：

```
var mutablePoint = PointStruct(x: 1, y: 1)  
mutablePoint.x = 3;
```

如果使用 let 关键字声明结构体中的 x 和 y 属性，那么一旦初始化，我们将再也不能修改它们，无论保存这个点实例的变量可选还是不可选的：

```
struct ImmutablePointStruct {  
    let x: Int  
    let y: Int  
}  
  
var immutablePoint2 = ImmutablePointStruct(x: 1, y: 1)  
  
immutablePoint2.x = 3 // 被拒绝!
```

当然，我们仍然可以给 immutablePoint2 赋一个新值：

```
immutablePoint2 = ImmutablePointStruct(x: 2, y: 2)
```

Objective-C

Objective-C 程序员对于可变性和不可变性的概念应该早已十分熟悉。Apple 的 Core Foundation 和 Foundation 框架提供的许多数据结构都存在不可变和可变两个版本，比如 NSArray 和 NSMutableArray，NSString 和 NSMutableString，当然并不只有这两对。大多数情况下使用不可变类型是默认选项，就像 Swift 中比起引用类型更倾向于优先选择值类型一样。

不过，相较于 Swift，Objective-C 中并没有万无一失的方法来确保变量不可变。我们可以将对象的属性声明为只读（或者为了避免可变，仅暴露一个接口），但这无法阻止我们在无意地在类型内部修改已经被初始化的值。比方说，当遇上遗留代码时，那些编译器力所不逮的可变性假定实在是太容易被打破了。在使用可变变量时，如果没有经由编译器进行检验，保证任何一种规范都将是天方夜谭。

在和框架代码打交道的时候，我们通常可以将已经存在的可变类封装到一个结构体中。不过，这里务必小心：如果我们在结构体中保存了一个对象，引用是不可变的，但是对象本身却可以改变。Swift 数组就是这样的：它们使用低层级的可变数据结构，但提供一个高效且不可变的接口。这里使用了一个被称为写入时复制 (copy-on-write) 的技术。你可以阅读我们的书籍《Swift 进阶》来了解更多关于封装已有 API 的内容。

讨论

在本章中，我们已经看到 Swift 如何区别可变值和不可变值，以及值类型和引用类型。在本章最后，我们想要解释一下为什么这些区别很重要。

在了解一款软件的时候，耦合度通常被用来描述代码各个独立部分之间彼此依赖的程度。耦合度是衡量软件构建好坏的重要因素之一。最坏的情况下，所有类和方法都错综复杂相互关联，共享大量可变变量，甚至连具体的实现细节都存在依赖关系。这样的代码难以维护和更新：你无法理解或修改一小段独立的代码片段，而是需要一直站在整体的角度来考虑整个系统。

在 Objective-C 和很多其它面向对象的语言中，对于方法而言，由于共享实例变量而产生耦合的情况十分常见。其结果就是，修改变量的同时可能会改变类中方法的行为。通常，这是一件好事——如果你改变了存储在对象中的值，它的所有方法都将使用新值。不过同时，这样的共享实例变量在类的方法之间建立了耦合关系。一旦有方法或是外部函数将这个共享状态弄错，

所有类方法都有可能表现出错误行为。由于它们彼此耦合，独立测试任意一个方法也变得十分困难。

现在，让我们来回顾一下QuickCheck 章节中我们所测试的函数。那些函数的输出值都只取决于输入值。像这样只要输入值相同则得到的输出值一定相同的函数有时被称为**引用透明**函数。根据定义，引用透明函数在它所存在环境中是松耦合的：除了函数的参数，不存在任何隐式依赖的状态或变量。因此，引用透明函数更容易单独测试和理解。此外，我们可以创建、调用和组装引用透明函数，其结果也将是引用透明的。引用透明性是模块化和可重用性的重要保证。

引用透明化在各个层面都使代码更加模块化。想象一下，你通过阅读 API 源码来试图弄清楚它是如何工作。文档可能早已过时，不再有用，但如果这个 API 没有可变状态——所有变量都通过 `let` 而非 `var` 进行声明——这将会是难以置信的有用信息。你再也不必担心初始化对象或处理命令的顺序是否正确。而只需关注函数类型和 API 定义的常量，以及考虑它们是如何被组装起来并产生想要的值的。

在 Swift 中，`var` 和 `let` 之间的区别不仅使得程序员能够区分可变和不可变数据，还可以让编译器识别这种区别。相较于 `var`，我们更倾向于 `let`，它降低了程序的复杂性——你不必再因为不知道可变变量当前的值到底是什么感到不安，而可以简单地使用它们的不可变定义。对不可变性的偏爱使得编写引用透明函数更加容易，最终还降低了耦合度。

同样，Swift 中值类型和引用类型的区别，也鼓励你在程序中对那些可能改变的对象和不会改变的数据进行区分。函数可以自由地复制、改变或共享传入的值类型——任何对它的修改仅会影响函数内部的副本。另外，尽量使用引用透明的函数，将会有助于编写耦合更松散的代码，因为任何源自共享状态或对象的依赖性都将被消除。

我们可以彻底不使用可变变量吗？像 Haskell 一样的纯函数式编程语言鼓励程序员彻底避免使用可变状态。当然，在这个世界上是存在不使用任何可变状态且庞大的 Haskell 程序的。然而在 Swift 中，教条式地不惜一切代价避开 `var` 并不见得会使你的代码更好。在不少情况下，函数会在其内部使用一些可变状态。不妨看看下面这个求所有数组元素之和的例子：

```
func sum(integers: [Int]) -> Int {
    var result = 0
    for x in integers {
        result += x
    }
    return result
}
```

`sum` 函数使用的变量 `result` 是可变的，它反复被更新。但是暴露给用户的接口却隐瞒了这个事实。`sum` 函数依然是引用透明的，甚至比一个为了避开可变变量而不惜一切代价所进行的繁琐定义更容易理解。这个例子展示了一种可变变量的**良性**使用方式。

像这样良性的可变变量运用很广泛。比方说在QuickCheck 章节定义的 `qsort` 方法：

```
func qsort(_ input: [Int]) -> [Int] {
    if input.isEmpty { return [] }
    var array = input
    let pivot = array.removeLast()
    let lesser = array.filter { $0 < pivot }
    let greater = array.filter { $0 >= pivot }
    return qsort(lesser) + [pivot] + qsort(greater)
}
```

虽然这个方法尽可能避免了可变引用的使用，但它带来了额外的内存开销，无法运行在常数量级($O(1)$)的内存中。它为组成返回值的新数组 `lesser` 和 `greater` 分配了内存。当然，通过使用可变数组，我们可以定义一个运行在常数量级内存中，且仍然是引用透明的新版本的快速排序算法。巧妙地使用可变变量有时能够提升性能和内存使用。

总而言之，Swift 提供了几种专门控制程序中使用可变状态的语法特征。虽然完全避开可选状态几乎不可能，但是仍有很多程序过度且不必要地使用可变性。学会在可能的时候避免使用可变状态和对象，将有助于降低耦合度，从而改善你的代码结构。

枚举

8

在设计和实现 Swift 应用时，**类型**扮演着非常重要的角色，这也是本书想说明的重点之一。在这一章，我们将介绍 Swift 中的**枚举**类型。借此，你可以创建更为严密的类型，来表示应用中使用的数据。

关于枚举

创建字符串时，字符编码是很重要的信息之一。在 Objective-C 中，NSString 对象会有以下几种可能的编码：

```
NS_ENUM(NSStringEncoding) {
    NSASCIIStringEncoding = 1,
    NSNEXTSTEPStringEncoding = 2,
    NSJapaneseEUCStringEncoding = 3,
    NSUTF8StringEncoding = 4,
    // ...
};
```

每一种编码都可以用一个数字来表示，NS_ENUM 关键字允许开发者为整数常量指派一些有意义的名字，以此来关联特定的字符编码。

在 Objective-C 和其他类 C 语言中，枚举的声明方式是有一些缺陷的。最需要注意的是，NSStringEncoding 作为类型来说并不够严密 —— 有些整数值，比如 16，并没有一个与之对应的合法编码。更糟糕的是，正因为所有的枚举类型实际上都是整数，它们之间是可以进行运算的，**就好像它们只是数字一样**。

```
NSAssert(NSASCIIStringEncoding + NSNEXTSTEPStringEncoding
        == NSJapaneseEUCStringEncoding, @"Adds up...");
```

谁能想到 NSASCIIStringEncoding + NSNEXTSTEPStringEncoding 会等于 NSJapaneseEUCStringEncoding 呢？这样的表达式虽然毫无意义，但 Objective-C 的编译器却对此大开方便之门。

在之前章节列举的例子中，我们已经利用 Swift 中的**类型系统**发现过类似的错误。仅仅依靠整数作为标记的枚举类型，并不满足 Swift 函数式编程中的一条核心原则：高效地利用类型排除程序缺陷。

Swift 也有一种 enum 的构造方式，不过其用法与你熟悉的 Objective-C 语法相距甚远。我们可以用下面的代码来声明我们自己的字符编码枚举类型：

```
enum Encoding {  
    case ascii  
    case nextstep  
    case japaneseEUC  
    case utf8  
}
```

新枚举只定义了我们之前在 `NSStringEncoding` 枚举中列举的前四种编码，实际上类似的字符编码还有很多，这里就不再一一列举了。毕竟，上文声明的 Swift 枚举只是为了说明问题。

`Encoding` 类型中包括四个可能值：`ascii`, `nextstep`, `japaneseEUC` 与 `utf8`。我们将这些可能的值视为枚举的**枚举值**。在很多文献中，这样的枚举有时会被称为**和类型** (*sum types*)，不过在本书中，将以苹果的术语为准。

与 Objective-C 相比而言，编译器是**不支持**以下代码的：

```
let myEncoding = Encoding.ascii + Encoding.utf8
```

不同于 Objective-C，枚举在 Swift 中创建了新的类型，与整数或者其他已经存在的类型没有任何关系。

我们可以定义一个函数，利用 `switch` 语句来计算对应的编码。比如，我们可能希望计算出枚举的成员在 `NSStringEncoding` (在导入 Swift 后为 `String.Encoding`) 中对应的值：

```
extension Encoding {  
    var nsStringEncoding: String.Encoding {  
        switch self {  
            case .ascii: return String.Encoding.ascii  
            case .nextstep: return String.Encoding.nextstep  
            case .japaneseEUC: return String.Encoding.japaneseEUC  
            case .utf8: return String.Encoding.utf8  
        }  
    }  
}
```

这里的 `nsStringEncoding` 属性映射了每一个 `Encoding` 条件下对应的 `NSStringEncoding` 值。要注意的是，以上四种不同的编码方案各自对应了一条分支。如果缺少了任意一条分支，Swift 的编译器会警告我们这个计算属性中的 `switch` 语句是不完整的。

当然，我们也可以定义一个函数实现相反的功能，即根据 `NSStringEncoding` 创建一个 `Encoding`。我们可以据此实现一个 `Encoding` 枚举的构造方法：

```
extension Encoding {
    init?(encoding: String.Encoding) {
        switch encoding {
            case String.Encoding.ascii: self = .ascii
            case String.Encoding.nextstep: self = .nextstep
            case String.Encoding.japaneseEUC: self = .japaneseEUC
            case String.Encoding.utf8: self = .utf8
            default: return nil
        }
    }
}
```

由于这个精简版的 `Encoding` 枚举并没有列举所有可能的 `NSStringEncoding` 值，所以该构造方法是可失败的。如果前四个条件都不匹配，`default` 分支就会被选中，并返回一个 `nil`。

在编写完以上的代码之后，我们在使用 `Encoding` 枚举时，就不必再使用 `switch` 语句了。比如，我们想得到某个编码的本地化名称，可以编写以下代码：

```
extension Encoding {
    var localizedName: String {
        return String.localizedName(of: nsStringEncoding)
    }
}
```

关联值

到这里，我们已经看到了 Swift 的枚举表示若干选项中的特定选项的用法。`Encoding` 枚举为不同的字符编码方案提供了一种安全、类型化的表示方式。不过，Swift 中的枚举可不止这么点用途。

回过头来看看第五章中的 `populationOfCapital` 函数。它用来查找一个国家的首都，如果找到，它会返回该城市的人口总数。这个函数的返回类型是一个整数类型的可选值：如果所有信息都被找到的话，返回人口数；否则，返回 `nil`。

使用 Swift 的可选值类型时有一个缺点：当有错误发生时，我们无法返回相关的信息，所以也无法从判定到底是哪里出了错。是国家的信息不在我们的字典里么？还是首都的人口数没有被定义？

如果可以的话，我们会更希望 `populationOfCapital` 函数返回一个 Int 或者一个 Error。利用 Swift 的枚举，就可以搞定这件事。我们可以重新定义 `populationOfCapital` 函数，使之返回一个 `PopulationResult` 枚举的成员，来代替之前的 Int?。可以像下文这样定义 `PopulationResult`：

```
enum LookupError: Error {
    case capitalNotFound
    case populationNotFound
}

enum PopulationResult {
    case success(Int)
    case error(LookupError)
}
```

与枚举 `Encoding` 相比，`PopulationResult` 的每个枚举值都具有一个关联值：枚举值 `success` 关联了一个整数，而 `error` 则关联了一个 `Error`。具体的用法，可以参考以下对枚举值 `success` 的声明：

```
let exampleSuccess: PopulationResult = .success(1000)
```

类似地，使用枚举值 `Error` 来创建一个 `PopulationResult` 时，则需要关联一个 `LookupError` 值。

现在，我们可以重写 `populationOfCapital` 函数，使之返回一个 `PopulationResult`：

```
func populationOfCapital(country: String) -> PopulationResult {
    guard let capital = capitals[country] else {
        return .error(.capitalNotFound)
    }
    guard let population = cities[capital] else {
        return .error(.populationNotFound)
    }
    return .success(population)
}
```

现在，函数会返回人口数或者一个 `LookupError` 来代替之前的 `Int` 可选值。首先，我们检查了 `capitals` 字典中是否存在对应的首都名，如果不存在，就返回一个 `.capitalNotFound` 错误。接着，我们验证了 `cities` 字典中是否存在对应的人口数，如果不存在，则返回一个 `.populationNotFound` 错误。最后，如果两次查询都找到了对应的值，便返回一个 `success`。

在调用 `populationOfCapital` 时，可以使用一个 `switch` 语句来确定函数是否成功：

```
switch populationOfCapital(country: "France") {
    case let .success(population):
        print("France's capital has \(population) thousand inhabitants")
    case let .error(error):
        print("Error: \(error)")
}
// France's capital has 2241 thousand inhabitants
```

添加泛型

有人说，我想写一个与 `populationOfCapital` 类似的函数，只不过不是查询人口，而是查询一个国家首都的市长：

```
let mayors = [
    "Paris": "Hidalgo",
    "Madrid": "Carmena",
    "Amsterdam": "van der Laan",
    "Berlin": "Müller"
]
```

通过可选值，我们可以简单的查询到一个国家的首都，然后在结果中使用 `flatMap` 找到这座城市的市长：

```
func mayorOfCapital(country: String) -> String? {
    return capitals[country].flatMap { mayors[$0] }
}
```

然而，使用可选值作为返回类型，依旧不会告诉我们为什么查询会失败。

不过，我们已经知道如何去解决这个问题了！我们的第一反应，可能是通过复用 PopulationResult 枚举来返回错误。可是在这里，并没有一个能够与枚举值 success 相关联的 Int 值，我们有的只是一个字符串。虽然我们可以将字符串转换成对应的整数，但这并不是一个好的设计：我们应该使用更为严密的类型，来避免为类似的类型转换编写额外的代码。

或者，我们可以定义一个新枚举 MayorResult，来对应两种可能的情况：

```
enum MayorResult {  
    case success(String)  
    case error(Error)  
}
```

毫无疑问，我们可以利用这个枚举来编写另一个版本的 mayorOfCapital 函数 —— 不过为每一个新函数都引入一个枚举实在是太乏味了。更何况，MayorResult 与 PopulationResult 大同小异得令人发指。两个枚举值唯一的区别就是 success 的关联值类型。所以我们定义了一个新的枚举，将泛型作为 success 的关联值：

```
enum Result<T> {  
    case success(T)  
    case error(Error)  
}
```

现在，我们可以在 populationOfCapital 与 mayorOfCapital 中使用同样的结果类型了。新的类型表达式变成了下面的样子：

```
func populationOfCapital(country: String) -> Result<Int>  
func mayorOfCapital(country: String) -> Result<String>
```

populationOfCapital 函数返回一个 Int 或者一个 LookupError，mayorOfCapital 则返回一个 String 或 Error。

Swift 中的错误处理

实际上，Swift 中内建的错误处理机制与我们在上文定义的 Result 类型十分相似。它们的不同主要有两点：Swift 强制你使用关键字 throws 注明哪些函数和方法可能抛出错误，且必须使用 try (或 try 的变体) 来调用这些代码。如果换作 Result 类型的话，我们是无法在静态环境下确保错误被处理的。另外，Swift 内建错误处理机制的局限性在于，它必须借助函数的返回类型来触

发：如果我们想构建一个函数，且提供的参数包含失败情况（比如一个回调函数），使用 `throw` 的方式来提供这个参数，会让一切都变得复杂起来。若是换用可选值或 `Result`，编写起来就没那么繁琐，处理也会更简单。

如果使用 Swift 的错误处理机制重写 `populationOfCapital`，我们可以简单地在函数声明上加入 `throws` 关键字。相应的，我们得 `throw` 一个错误，而不再是返回一个 `.error`。类似地，我们现在可以直接返回人口数而不再是一个 `.success` 了：

```
func populationOfCapital(country: String) throws -> Int {
    guard let capital = capitals[country] else {
        throw LookupError.capitalNotFound
    }
    guard let population = cities[capital] else {
        throw LookupError.populationNotFound
    }
    return population
}
```

要调用一个有 `throws` 标记的函数，我们可以将调用代码嵌入一个 `do` 执行块中，然后添加一个 `try` 的前缀。这样做的好处在于，我们可以在 `do` 执行块中编写正常的流程，然后在 `catch` 块中去处理所有可能的错误：

```
do {
    let population = try populationOfCapital(country: "France")
    print("France's population is \(population)")
} catch {
    print("Lookup error: \(error)")
}
// France's population is 2241
```

再聊聊可选值

实际上，Swift 内建的可选值类型与 `Result` 类型也很像。下面的代码片段基本上是直接从 Swift 的标准库中复制出来的：

```
enum Optional<Wrapped> {
    case none
    case some(Wrapped)
```

```
// ...  
}
```

可选值类型提供了一些语法糖，像是后缀标记 ? 以及可选值的展开机制等，使其更容易被使用。其实，你完全可以自己来定义需要的操作。

比如，我们可以在我自己的 Result 类型中定义一些用于操作可选值的函数。通过在 Result 中重新定义 ?? 运算符，我们可以对 Result 进行运算：

```
func ??<T>(result: Result<T>, handleError: (Error) -> T {  
    switch result {  
        case let .success(value):  
            return value  
        case let .error(error):  
            return handleError(error)  
    }  
}
```

值得注意的是，我们并没有（像在之前讨论可选值的相关章节中对 ?? 的定义那样）使用 autoclosure 来标记第二个参数。实际上，在这里我们会显式地要求传入一个以 Error 作为参数的函数，而该函数需要返回一个类型为 T 的值。

数据类型中的代数学

就像我们之前提到的那样，枚举常常被称为“和类型”。这可能是一个让人困惑的名字，枚举看起来与数字毫无关系。不过深挖下去的话，你可能会发现，枚举和多元组的数学结构，在计算时其实非常相似。

在分析这个结构之前，我们需要考虑一个问题：两个类型在什么时候是相同的。这个问题可能会让你觉得诧异，答案就好像 String 与 String 相同，与 Int 是不同的一样显而易见，不是么？然而，当你把泛型、枚举、结构体还有函数都放在一起考虑时，问题就变得复杂起来了。实际上，这个看似简单的问题至今仍被当做一个数学中的基础课题在研究。为了讲清楚这个小节，我们需要先理解两个类型在什么时候是同构 (isomorphic) 的。

比较直观的解释是，如果两个类型 A 和 B 在相互转换时不会丢失任何信息，那么它们就是同构的。为此，我们需要构造两个函数， $f: (A) \rightarrow B$ ，和 $g: (B) \rightarrow A$ ，使两者可以相互转换。也就是说，对任意 $x: A$ ，调用 $g(f(x))$ 方法得到的结果一定与 x 相等；类似地，对任意 $y: B$ ，调用 $f(g(y))$

的结果也等于 y 。我们可以将刚才提到的关于同构的直观说明提取成一个定义：我们可以随意地利用 f 和 g 来转换 A 和 B ，而不会丢失信息（也就是说我们可以利用 g 来撤销 f ，反之亦然）。如果仅仅针对编程，这个定义可能不够严密——一个 64 位的值既可以被用于表示整数，也可以是一个内存地址，这是两个完全不同的概念。不过，在我们研究类型的数学结构时，这个定义就派上用场了。

接着，我们来看看下面的枚举：

```
enum Add<T, U> {  
    case inLeft(T)  
    case inRight(U)  
}
```

这段代码给出了两个类型， T 和 U 。枚举 $Add<T, U>$ 由一个 T 类型或者一个 U 类型的值组成。就像命名所表达的那样， Add 枚举是 T 与 U 的枚举值相加之和：如果 T 有三个枚举值，而 U 有七个，那 $Add<T, U>$ 就会有十个可能的枚举值。以上描述渗透出来的观点，也为枚举被称为“和类型”的原因，提供了更深层次的解释。

在算术中， 0 是加法的运算单元，比如 $x + 0$ 和 x 一样，可以表示任意一个数字 x 。我们可以找到一个枚举的表示方法类似于 0 么？有趣的是，Swift 允许我们定义以下的枚举：

```
enum Zero {}
```

这个枚举是空的——它没有任何枚举值。正如我们所希望的那样，这个枚举与算术中的 0 有着极其相似的功能：对任何一个类型 T ， $Add<T, Zero>$ 和 T 是同构的。这很容易被证明。我们可以使用 `inLeft` 定义一个函数将 T 转换为 $Add<T, Zero>$ ，而反向的转换则可以通过模式匹配来完成。

在 Swift 3 中，标准库添加了一个类型 `Never`。`Never` 与 `Zero` 的定义十分相似：它可以作为一个不返回任何值的函数的返回类型。在函数式语言中，该类型有时也被称作 **底层类型** (*bottom type*)。

关于加法的部分就到此为止——我们再来看看乘法。如果有一个包含三个枚举值的枚举 T ，和另一个包含两个枚举值的枚举 U 。我们如何去定义一个混合类型 $Times<T, U>$ ，使之包含六个成员呢？如果要满足这个需求， $Times<T, U>$ 类型应该被允许同时选择一个 T 的成员和一个 U 的成员。换句话说，它应该可以代表一对类型分别为 T 和 U 的值：

```
typealias Times<T, U> = (T, U)
```

就像 Zero 可以作为一个加法的单元一样，空类型 ()，也可以作为一个 Times (乘法) 的单元：

```
typealias One = ()
```

将这些结构看作同构类型时，很多熟悉的算术规则在这里同样适用，且很容易被验证：

- Times<One, T> 与 T 是同构的
- Times<Zero, T> 与 Zero 是同构的
- Times<T, U> 与 Times<U, T> 是同构的

使用枚举和结构体定义的类型有时候也被称作代数数据类型 (*algebraic data types*)，因为它们就像自然数一样，具有代数学结构。

关于数字与类型的一致性这个话题，其实还可以挖的更深。比如函数在某种程度上就相当于幂运算。甚至，还可以为类型定义一个微分的概念！

以上的讨论可能没什么实用的价值。只不过，它告诉了我们，包括枚举在内的许多 Swift 特性，并不是什么新的发明，而是从经年累月的数学研究和编程语言设计中所萃取的精华。

为什么使用枚举？

在实际开发中，可选值可能还是会比上文定义的 Result 类型更好用，原因有很多：内建的语法糖使用起来更方便；相对于使用自己定义的枚举，依赖一些已经存在的类型，会使你定义的接口更容易被其他 Swift 开发者接受；而且有时候并不值得为 Error 专门费事去定义一个枚举。

其实，我们希望说明的问题，并不是“Result 类型是 Swift 中处理错误的最好的方案”。我们只是试图阐述，如何使用枚举去定义你自己的类型，来解决你的具体需求。通过让类型更加严密，我们可以在程序测试或运行之前，就利用 Swift 的类型检验优势，来避免许多错误。

纯函数式数据结 构

9

在前面的章节中，我们了解了如何利用枚举针对正在开发的应用来定义特定类型。而本章中，我们会定义可递归的枚举，并向大家展示，如何利用这个特性来定义一些高性能且非可变的数据结构。

所谓纯函数式数据结构 (Purely Functional Data Structures) 指的是那些具有不变性的高效的数据结构。像 C 或 C++ 这样的指令式语言中的数据结构往往是可变的，直接将这类数据结构使用在函数式语言中往往会使水土不服。通过本章，我们想向您展示函数式语言中的纯函数式数据结构的构建方式和特点。

二叉搜索树

在 Swift 发布的时候，并没有一个类似于标准库中 NSSet 的类型来处理无序集合 (Set)。尽管我们可以使用 Swift 封装 NSSet —— 就像我们曾经为 Core Image 与 String 构造方法所做的那样 —— 但这里我们还是想探究一种不太一样的方式。我们的目标，依旧不是去定义一个功能全面的类型 (Set 类型已经在 Swift 2 之后被添加到了标准库中)，而是为了演示如何用递归式枚举来定义高效的数据结构。

在这个迷你库中，我们会实现以下三种操作：

- isEmpty —— 检查一个无序集合是否为空
- contains —— 检查无序集合中是否包含某个元素
- insert —— 向无序集合中插入一个元素

我们最先想到的，可能是使用数组来表示无序集合。那么实现这三个操作，简直是探囊取物：

```
struct MySet<Element: Equatable> {  
    var storage: [Element] = []  
  
    var isEmpty: Bool {  
        return storage.isEmpty  
    }  
  
    func contains(_ element: Element) -> Bool {  
        return storage.contains(element)  
    }  
  
    func inserting(_ x: Element) -> MySet {
```

```
    return contains(x) ? self : MySet(storage: storage + [x])
}
}
```

尽管实现简单，可随之而来的痛点是，大部分操作的性能消耗与无序集合的大小是线性相关的。如果无序集合过大，这可能会导致性能问题。

想要提高性能，这里有一些可行的方式。例如，我们可以确保数组是经过排序的，然后使用二分查找来定位特定元素。或者再彻底一些，索性定义一个**二叉搜索树** (Binary Search Trees) 来表示无序集合。我们可以用传统的 C 语言风格打造一个树形结构，在每个节点持有指向子树的指针。当然，也可以利用 Swift 中的 `indirect` 关键字，直接将二叉树结构定义为一个枚举：

```
indirect enum BinarySearchTree<Element: Comparable> {
    case leaf
    case node(BinarySearchTree<Element>,
              Element, BinarySearchTree<Element>)
}
```

这个定义规定了每一棵树，要么是：

- 一个没有关联值的叶子 `leaf`，要么是
- 一个带有三个关联值的节点 `node`，关联值分别是左子树，储存在该节点的值和右子树。

在为二叉树定义函数之前，我们可以手动构造几棵树作为示例：

```
let leaf: BinarySearchTree<Int> = .leaf
let five: BinarySearchTree<Int> = .node(leaf, 5, leaf)
```

`leaf` 树是空的；`five` 树在节点上存了值 5，但两棵子树都为空。我们可以编写两个构造方法来生成这两种树：一个会创建一棵空树，而另一个则创建含有某个单独值的树：

```
extension BinarySearchTree {
    init() {
        self = .leaf
    }

    init(_ value: Element) {
        self = .node(.leaf, value, .leaf)
    }
}
```

```
    }
}
```

就像我们之前章节中看到的那样，我们可以编写一些函数，利用 `switch` 语句来处理这些树。由于 `BinarySearchTree` 枚举是支持递归的，所以理所应当，我们编写的许多基于树的函数也都会是递归的。举个例子，下面的函数用来计算一棵树中存值的个数：

```
extension BinarySearchTree {
    var count: Int {
        switch self {
            case .leaf:
                return 0
            case let .node(left, _, right):
                return 1 + left.count + right.count
        }
    }
}
```

在枚举值为基本值 `.leaf` 时，可以直接返回 0。而在值为 `.node` 时就比较有意思：我们递归地计算了两个子树储存的元素个数，然后加 1，也就是当前节点存值的个数，再将它们的总和返回。

类似地，我们可以写一个 `elements` 属性，用于计算树中所有元素组成的数组：

```
extension BinarySearchTree {
    var elements: [Element] {
        switch self {
            case .leaf:
                return []
            case let .node(left, x, right):
                return left.elements + [x] + right.elements
        }
    }
}
```

`count` 属性与 `elements` 非常相似。对于 `leaf` 的情况，会有一个基础值。而在 `node` 的情况下，它将递归地调用子节点，然后将结果与当前节点中的元素合并起来。这个被抽象出来的过程，有时候被称作 `fold` 或 `reduce`：

```
extension BinarySearchTree {
```

```

func reduce<A>(leaf leafF: A, node nodeF: (A, Element, A) -> A) -> A {
    switch self {
        case .leaf:
            return leafF
        case let .node(left, x, right):
            return nodeF(left.reduce(leaf: leafF, node: nodeF),
                         x,
                         right.reduce(leaf: leafF, node: nodeF))
    }
}
}

```

这使得我们可以以很少的代码来编写 elements 与 count:

```

extension BinarySearchTree {
    var elementsR: [Element] {
        return reduce(leaf: []) { $0 + [$1] + $2 }
    }
    var countR: Int {
        return reduce(leaf: 0) { 1 + $0 + $2 }
    }
}

```

现在，让我们回到最初的目的，即利用树型结构编写一个高效的无序集合库。对于检查一棵树是否为空，有一个现成的方案：

```

extension BinarySearchTree {
    var isEmpty: Bool {
        if case .leaf = self {
            return true
        }
        return false
    }
}

```

这里需要注意的是属性 isEmpty 在枚举值为 node 时，可以直接返回 false，而不必再去检查子树或是当前节点储存的值。

遗憾地是，当我们试着去我们编写 `insert` 和 `contains` 函数的雏形时，看起来并没有什么可以利用的特性。不过，如果为这个结构加上一个**二叉搜索树**的限制，问题就会迎刃而解。如果一棵(非空)树符合以下几点，就可以被视为一棵二叉搜索树：

- 所有储存在左子树的值都**小于**其根节点的值
- 所有储存在右子树的值都**大于**其根节点的值
- 其左右子树都是二叉搜索树

我们在本章实现的 `BinarySearchTree` 有一个缺点：因为你可以“手动”构造出任何样式的树，所以我们无法严格地将一棵树限制为二叉搜索树。在实际情况中，我们应该把枚举封装为一个私有的实现细节，以确保我们生成的树是一棵二叉搜索树。为求简单，我们在这里忽略就好。

我们可以写一个(低效率的)属性来检查 `BinarySearchTree` 实际上是不是一棵二叉搜索树：

```
extension BinarySearchTree {  
    var isBST: Bool {  
        switch self {  
            case .leaf:  
                return true  
            case let .node(left, x, right):  
                return left.elements.all { y in y < x }  
                    && right.elements.all { y in y > x }  
                    && left.isBST  
                    && right.isBST  
        }  
    }  
}
```

方法 `all` 检查了一个数组中的元素是否都符合某个条件。它被定义为一个 `Sequence` 协议的拓展：

```
extension Sequence {  
    func all(predicate: (Iterator.Element) -> Bool) -> Bool {  
        for x in self where !predicate(x) {  
            return false  
        }  
        return true  
    }  
}
```

```
}
```

二叉搜索树的关键特性在于它们支持高效的查找操作，类似于在一个数组中做二分查找。当我们需要遍历一棵树来查找某个元素是否在树中时，我们可以在每一步都排除一半元素。举例来说，这里有一个可行的 `contains` 函数定义，来查找一个元素是否在树中：

```
extension BinarySearchTree {
    func contains(_ x: Element) -> Bool {
        switch self {
            case .leaf:
                return false
            case let .node(_, y, _) where x == y:
                return true
            case let .node(left, y, right) where x < y:
                return left.contains(x)
            case let .node(_, y, right) where x > y:
                return right.contains(x)
            default:
                fatalError("The impossible occurred")
        }
    }
}
```

`contains` 函数现在被分为四种可能的情况：

- 如果树是空的，则 `x` 不在树中，返回 `false`。
- 如果树不为空，且储存在根节点的值与 `x` 相等，返回 `true`。
- 如果树不为空，且储存在根节点的值大于 `x`，那么如果 `x` 在树中的话，它一定是在左子树中，所以，我们在左子树中递归搜索 `x`。
- 类似地，如果根节点的值小于 `x`，我们就在右子树中继续搜索。

不幸地是，Swift 的编译器还没有聪明到能够发现这四种情况已经包括了所有的可能性，所以我们还得再添加一个用来安抚编译器的 `default`。

插入操作也是用同样的方式对二叉搜索树进行搜索：

```
extension BinarySearchTree {
```

```
mutating func insert(_ x: Element) {
    switch self {
        case .leaf:
            self = BinarySearchTree(x)
        case .node(var left, let y, var right):
            if x < y { left.insert(x) }
            if x > y { right.insert(x) }
            self = .node(left, y, right)
    }
}
```

不同于先检查一个元素是否已经被包括在一棵二叉搜索树中，`insert` 会找到一个合适的位置来添加新元素。如果树是空的，就构建一棵只有一个元素的树。如果元素已经存在，就返回树本身。否则，`insert` 函数持续地递归，直到找到一个合适的位置来插入新元素。

`insert` 函数被写作一个 `mutating` (可变的) 函数，然而，这与那种基于类的数据结构中的可变性有很大区别。实际的值并没有被修改，被修改的只是变量。举个例子，在执行插入操作的情况下，新的树是在旧树的分支之外构建的，分支本身并不会被修改。数据结构的这种特性通常被称作**可持久化的数据结构** (*persistent data structures*) 我们可以观察一个例子来验证这个特性：

```
let myTree: BinarySearchTree<Int> = BinarySearchTree()
var copied = myTree
copied.insert(5)
myTree.elements // []
copied.elements // [5]
```

在最坏的情况下，二叉搜索树中的 `insert` 与 `contains` 仍然是线性的 —— 毕竟，总会出现像是所有的左子树都为空这种非常不平衡的树。一些更为巧妙的实现方案，比如 2-3 树，AVL 树，或者红黑树，可以通过使每棵树都保持合理平衡来避免这种情况。另外，我们并没有编写 `delete` 操作，这个操作也需要对树进行反复地平衡。关于这些内容，各种文献中有大量被充分论证过的经典方案 —— 所以重申一下，这里的例子只是为了说明如何利用递归枚举，而并不会构建一个完整的库。

基于字典树的自动补全

在了解了二叉树之后，这一小节会演示一个更加高级的纯函数式数据结构。假设，我们现在需要自己实现一个自动补全算法——在给定一组搜索的历史记录和一个现在待搜索字符串的前缀时，计算出一个与之相匹配的补全列表。

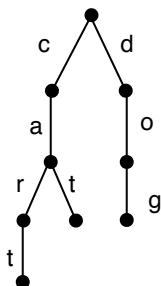
如果使用数组的话，一句代码就可以解决问题：

```
extension String {  
    func complete(history: [String]) -> [String] {  
        return history.filter { $0.hasPrefix(self) }  
    }  
}
```

遗憾地是，这个函数依旧不是很高效。在历史记录很多，或是前缀很长的情况下，运算会慢得离谱。按照之前的经验，我们可以将历史记录排序为一个数组，并对其使用某种二叉搜索来提高性能。不过这次，我们可以试试另一种解决方案，这会用到一种专门用于解决类似检索问题的自定义数据结构。

字典树 (Tries)，也被称作数字搜索树 (digital search trees)，是一种特定类型的有序树，通常被用于搜索由一连串字符组成的字符串。不同于将一组字符串储存在一棵二叉搜索树中，字典树把构成这些字符串的字符逐个分解开来，储存在了一个更高效的数据结构中，

上文中，BinarySearchTree 类型的每个节点处都存在两棵子树。对于字典树来说，则是每一个字符都（潜在地）对应着一棵子树，不过也因此，其每个节点的子树个数都是不确定的。比如，我们可以想象一棵储存着 “cat”、“car”、“cart” 和 “dog” 的字典树，如下图所示：



字典树

想知道“care”是不是在这棵字典树中，我们可以从根节点开始，沿着标记了 c、a 和 r 的路径找下去。标记着 r 的节点并没有一个子节点被标记为 e，所以字符串“care”并不在这棵树中。而字符串“cat”是在这个字典树里的，因为我们可以从根节点找到一条标记了 c、a、和 t 的路径。

我们应该如何在 Swift 中表示一棵字典树呢？最先想到的，是写一个结构体，并以一个字典作为属性，用来储存所有节点处字符与子字典树的映射关系：

```
struct Trie {  
    let children: [Character: Trie]  
}
```

在这个定义的基础上，我们可以再做两点优化。首先，我们需要在节点上添加一些额外信息。从上图的字典树中可以看出，在添加“cart”到字典树中时，“cart”的所有前缀，“c”、“ca”和“car”也会被添加进来。为了区分出这些前缀是不是也作为一个元素储存在字典树中，我们会在每个节点添加一个额外的布尔值 isElement。这个布尔值会标记截止于当前节点的字符串是否在树中。此外，我们可以定义一棵泛型字典树，去掉只能储存字符的限制。据此定义字典树如下：

```
struct Trie<Element: Hashable> {  
    let isElement: Bool  
    let children: [Element: Trie<Element>]  
}
```

接下来的文章中，我们有时会将 [Element] 类型的一组键（以下简称键组）看作字符串，而将 Element 类型的值看作字符。这种做法并不严谨——Element 可以被实例化为与字符不同的类型，而字符串实际上也不是 [Character]——不过我们希望这可以让你更直观的感受到“字典树储存了一组字符串”。

在利用字典树定义自动补全函数之前，我们可以先写一些简单的定义来练练手。比如，一棵空的字典树应该由一个字典为空的节点构成：

```
extension Trie {  
    init() {  
        isElement = false  
        children = [:]  
    }  
}
```

如果将一棵空字典树的 `isElement` 赋值为 `true` 而不是 `false`，那么空字符串就会变成空字典树的一个元素——空字典树里却有一个字符串么？别闹！

接着，我们定义一个属性，用于将字典树展平 (`flatten`) 为一个包含全部元素的数组：

```
extension Trie {
    var elements: [[Element]] {
        var result: [[Element]] = isElement ? [[]] : []
        for (key, value) in children {
            result += value.elements.map { [key] + $0 }
        }
        return result
    }
}
```

这个函数的内部实现十分精妙。首先，我们会检查当前的根节点是否被标记为一棵字典树的成员。如果是，这个字典树就包含了一个空的键，反之，`result` 变量则会被实例化为一个空的数组。接着，函数会遍历字典，计算出子树的所有元素——这是通过调用 `value.elements` 实现的。最后，每一棵子字典树对应的“character”（也就是代码中的 `key`）会被添加到子树 `elements` 的首位——这正是 `map` 函数中所做的事情。虽然我们也可以使用 `flatMap` 函数取代 `for` 循环来实现属性 `elements`，不过现在的代码让整个过程能稍微清晰一些。

接下来，我们将定义查询和插入的函数。不过在这之前，我们还需要几个辅助函数。我们已经使用了数组来表示键组，虽然我们将字典树定义为了一个（递归的）结构体，但数组却并不能递归。一个能够被遍历的数组还是很有用的，为数组添加下文中的拓展可以便捷的实现这个功能：

```
extension Array {
    var slice: ArraySlice<Element> {
        return ArraySlice(self)
    }
}

extension ArraySlice {
    var decomposed: (Element, ArraySlice<Element>)? {
        return isEmpty ? nil : (self[startIndex], self.dropFirst())
    }
}
```

属性 `decomposed` 会检查一个数组切片是否为空。如果为空，就返回一个 `nil`；反之，则会返回一个多元组，这个多元组包含该切片的第一个元素，以及去掉第一个元素之后的切片。我们可以通过重复调用 `decomposed` 递归地遍历一个数组，直到返回 `nil`，而此时数组将为空。

我们之所以为 `ArraySlice` 而不是 `Array` 定义 `decomposed`，是因为性能上的原因。`Array` 中的 `dropFirst` 方法的复杂度是 $O(n)$ ，而 `ArraySlice` 中 `dropFirst` 的复杂度则为 $O(1)$ 。因此，此处的 `decomposed` 也只具有 $O(1)$ 的复杂度。

比如，我们可以抛开 `for` 循环或是 `reduce` 函数，而使用 `decompose` 函数递归地对一个数组的元素求和：

```
func sum(_ integers: ArraySlice<Int>) -> Int {  
    guard let (head, tail) = integers.decomposed else { return 0 }  
    return head + sum(tail)  
}  
  
sum([1,2,3,4,5].slice) // 15
```

回到我们最开始的问题——我们现在可以使用 `ArraySlice` 的辅助方法 `decomposed` 来为数组切片编写一个查询函数：给定一个由一些 `Element` 组成的键组，遍历一棵字典树，来逐一确定对应的键是否储存在树中：

```
extension Trie {  
    func lookup(key: ArraySlice<Element>) -> Bool {  
        guard let (head, tail) = key.decomposed else { return isElement }  
        guard let subtrie = children[head] else { return false }  
        return subtrie.lookup(key: tail)  
    }  
}
```

查询可以被分为三种情况：

- 键组是一个空数组——在这种情况下，我们返回当前节点的 `isElement`，即字典树中用于描述这个字符串是否存在与树中的布尔值。
- 键组不为空，但是不存在对应的子树——在这种情况下，我们返回 `false` 就可以了，因为字典树中没有储存这个键组。

→ 键组不为空 —— 在这种情况下，我们会查询键组中第一个键对应的子树。如果子树存在，我们就递归地调用该函数，来查询剩余的键是否在这棵子树中。

我们也可以对 `lookup` 函数小作修改，给定一个前缀键组，使其返回一个含有所有匹配元素的子树：

```
extension Trie {  
    func lookup(key: ArraySlice<Element>) -> Trie<Element>? {  
        guard let (head, tail) = key.decomposed else { return self }  
        guard let remainder = children[head] else { return nil }  
        return remainder.lookup(key: tail)  
    }  
}
```

该函数与 `lookup` 唯一的不同在于它不再返回一个 `isElement` 的布尔值，而是将整棵子树作为返回值，其中包含了所有以参数作为前缀的元素。

终于，我们可以利用这个高性能的字典树数据结构，重新定义 `complete` 函数：

```
extension Trie {  
    func complete(key: ArraySlice<Element>) -> [[Element]] {  
        return lookup(key: key)?.elements ?? []  
    }  
}
```

要计算出字典树中与给定前缀相匹配的所有字符串，我们只需要调用 `lookup` 函数，如果结果是字典树，就将其中的元素提取出来。如果不存在与给定前缀匹配的子树，就返回一个空数组。

我们可以使用与 `decompose` 相同的方式来创建字典树。比如，下面的代码可以创建只含有一个元素的字典树：

```
extension Trie {  
    init(_ key: ArraySlice<Element>) {  
        if let (head, tail) = key.decomposed {  
            let children = [head: Trie(tail)]  
            self = Trie(isElement: false, children: children)  
        } else {  
            self = Trie(isElement: true, children: [:])  
        }  
    }  
}
```

```
    }
}
```

这里分为两种情况：

- 如果传入的键组不为空，且能够被分解为 head 与 tail，我们就用 tail 递归地创建一棵字典树。然后创建一个新的字典 children，以 head 为键存储这个刚才递归创建的字典树。最后，我们用这个字典创建一棵新的字典树。因为输入的 key 非空，这意味着当前键组尚未被全部存入，所以 isElement 应该是 false。
- 如果传入的键组为空，我们可以创建一棵没有子节点的空字典树，用于储存一个空字符串，并将 isElement 赋值为 true。

我们还可以定义下面的插入函数来填充字典树：

```
extension Trie {
    func inserting(_ key: ArraySlice<Element>) -> Trie<Element> {
        guard let (head, tail) = key.decomposed else {
            return Trie(isElement: true, children: children)
        }
        var newChildren = children
        if let nextTrie = children[head] {
            newChildren[head] = nextTrie.inserting(tail)
        } else {
            newChildren[head] = Trie(tail)
        }
        return Trie(isElement: isElement, children: newChildren)
    }
}
```

这个插入函数被分为三种情况：

- 如果键组为空，我们将 isElement 设置为 true，然后不再修改剩余的字典树。
- 如果键组不为空，且键组的 head 已经存在于当前节点的 children 字典中，我们只需要递归地调用该函数，将键组的 tail 插入到对应的子字典树中。
- 如果键组不为空，且第一个键 head 并不是该字典树中 children 字典的某条记录，就创建一棵新的字典树来储存键组中剩下的键。然后，以 head 键对应新的字典树，储存在当前节点中，完成插入操作。

作为练习，你可以试着将 `inserting` 写成一个 `mutating` 函数。

字符串字典树

为了使用我们自己的自动补全算法，现在我们可以为字符串字典树写一些简化操作的封装。首先，我们可以编写一个简单的封装，从一个单词列表来进行字典树的构建。先创建一棵空字典树，然后将单词逐个插入，直到字典树包含了所有的单词。因为我们的字典树是基于数组工作的，所以需要先将每一个字符串转换为一个以字符为元素的数组切片。或者，我们也可以另写一个 `inserting`，以支持所有遵守 `Sequence` 协议的类型：

```
extension Trie {
    static func build(words: [String]) -> Trie<Character> {
        let emptyTrie = Trie<Character>()
        return words.reduce(emptyTrie) { trie, word in
            trie.inserting(Array(word.characters).slice)
        }
    }
}
```

然后，我们通过调用之前定义的 `complete` 方法，并将结果转换回字符串，就能得到一组经过我们自动补全的单词了。注意我们在每个结果前拼接输入字符串的方式，这么做是因为 `complete` 方法的返回没有包含相同的前缀，只返回了单词剩下的部分：

```
extension String {
    func complete(_ knownWords: Trie<Character>) -> [String] {
        let chars = Array(characters).slice
        let completed = knownWords.complete(key: chars)
        return completed.map { chars in
            self + String(chars)
        }
    }
}
```

为了测试我们的函数，我们可以使用一个简单的单词列表，创建一颗字典树，然后列出自动补全的选项：

```
let contents = ["cat", "car", "cart", "dog"]
```

```
let trieOfWords = Trie<Character>.build(words: contents)
"car".complete(trieOfWords)

/** 结果:
 ["car", "cart"]
 */
```

眼下，我们的接口只允许插入数组切片。创建另一版 `inserting` 方法，以支持我们插入任意的集合类型也很容易。类型签名可能会更复杂，但是函数内部的实现其实大同小异。

```
func inserting<S: Sequence>(key: Seq) -> Trie<Element>
  where S.Iterator.Element == Element
```

使 `Trie` 数据类型遵守 `Sequence` 协议非常简单，且这会让其获得很多函数式特性：协议自动地为元素提供了一些函数比如 `contains`, `filter`, `map` 和 `reduce`。我们可以在[生成器与迭代器](#)一章中看到 `Sequence` 协议更详细的细节。

讨论

我们在本章中列举了两个例子，使用枚举和结构体编写了高性能的不可变数据类型。在 Chris Okasaki 所著的**《纯函数式数据结构》**(1999) 中，还有很多其它的范例，这本书也是该主题下的标准参考书目之一。感兴趣的读者或许也会想阅读 Ralf Hinze 与 Ross Paterson 的关于 `finger trees`(2006) 的论述，这是一个可以满足诸多场景的通用纯函数式数据结构。最后，[StackOverflow](#) 中有一份极好的清单，列出了该领域近些年来大部分的研究成果。

案例研究：图表

10

在本章中，我们会看到一种描述图表的函数式方式，并讨论如何利用 Core Graphics 来绘制它们。通过对 Core Graphic 进行一层函数式的封装，可以得到一个更简单且易于组合的 API。

译者注：本章的示例代码并非严格按照编写顺序给出。如果希望能以调试代码的方式来理解本章内容，建议您直接下载本章的示例代码。如果下载不便，则建议您在进行通篇阅读之后，再摘抄需要用到的代码。

绘制正方形和圆

想象一下该如何绘制图 10.1 中的图表。在 Core Graphic 中，可以通过以下的代码来实现：

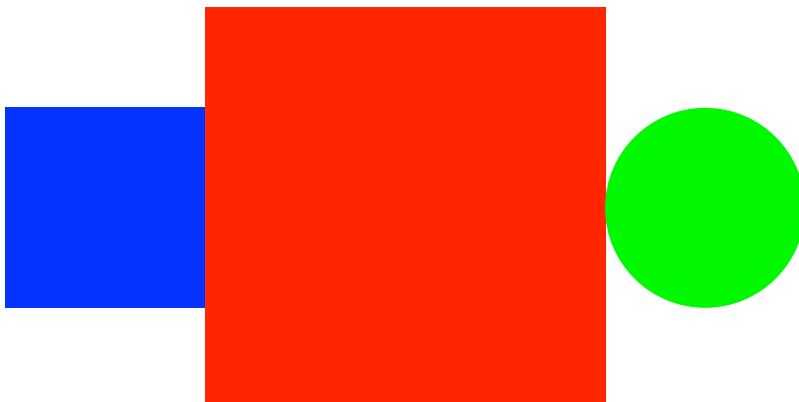


图 10.1: 简易图表

```
let bounds = CGRect(origin: .zero, size: CGSize(width: 300, height: 200))
let renderer = UIGraphicsImageRenderer(bounds: bounds)
renderer.image { context in
    UIColor.blue.setFill()
    context.fill(CGRect(x: 0.0, y: 37.5, width: 75.0, height: 75.0))
    UIColor.red.setFill()
    context.fill(CGRect(x: 75.0, y: 0.0, width: 150.0, height: 150.0))
    UIColor.green.setFill()
    context.cgContext.fillEllipse(in:
```

```
    CGRect(x: 225.0, y: 37.5, width: 75.0, height: 75.0))  
}
```

这段代码虽然短小精悍，但却有一点难以维护。比如，该如何像图 10.2 一样添加一个额外的圆进去呢？

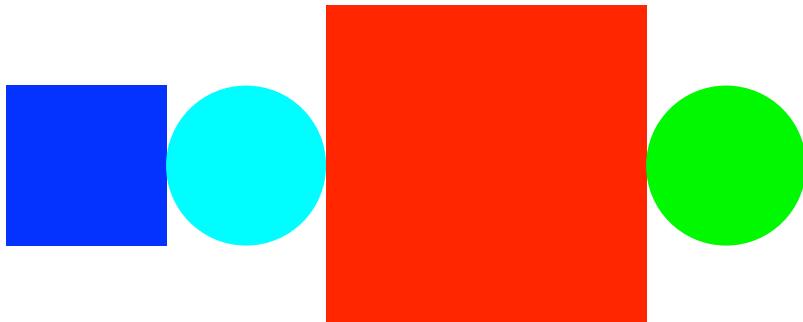


图 10.2: 添加额外的圆

我们可能得先添加一段绘制圆的代码，然后再更新位于该圆形右边其它图形的代码来移动它们。在 Core Graphic 中，我们总是在描述**如何**绘制物体。在本章中，我们将为图表构建一个库，以允许我们表达想画的是**什么**。举个例子，第一个图表可以像这样表达：

```
let blueSquare = square(side: 1).filled(.blue)  
let redSquare = square(side: 2).filled(.red)  
let greenCircle = circle(diameter: 1).filled(.green)  
let example1 = blueSquare ||| redSquare ||| greenCircle
```

添加第二个圆只需要简单地修改最后一行代码：

```
let cyanCircle = circle(diameter: 1).filled(.cyan)  
let example2 = blueSquare ||| cyanCircle ||| redSquare ||| greenCircle
```

上面的代码描述了一个相对边长为 1 的蓝色正方形。红色正方形的边长是它的两倍 (相对尺寸为 2)。通过使用运算符 `|||`，可以将相邻的正方形和圆依次排列，继而组合为图表。修改这个图表非常简单，且不用再去考虑计算边框或是移动周围的部分。这个例子描述了应该画**什么**，而不是**如何**画出来。

在函数式思想这一章中，我们已经构造了一些简易函数的组合来表示区域。尽管在讲解函数式的编程概念时这样做很有效，但之前的思路有一个硬伤：我们无法得知区域是**如何**被构造的——唯一能知道的只是一个点是否在这个区域中。

本章会更进一步：不再立刻执行绘制指令，而是构造一个中间层数据结构来对图表进行描述。这是一个非常强大的技巧：不同于之前区域的例子，这允许数据结构被检视，被修改，或将其转换成为其它的格式。

图 10.3 展示了一个更复杂的示例图表，一张由同一个库生成的柱形图：

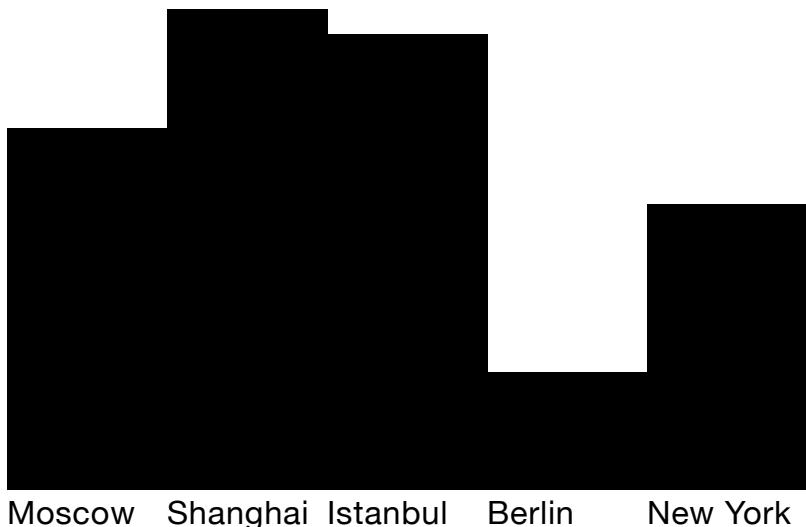


图 10.3: 柱形图

我们可以编写一个 `barGraph` 函数来处理一组由名称与值(柱形的相对高度)组成的多元组。对应每个多元组中值的部分，我们会绘制一个合适大小的矩形，然后使用 `hcat` 方法以水平方向连接这些矩形。最后，使用 `---` 运算符，将文字依次放置在柱形下方：

```
func barGraph(_ input: [(String, Double)]) -> Diagram {
    let values: [CGFloat] = input.map { CGFloat($0.1) }
    let bars = values.normalized.map { x in
        return rect(width: 1, height: 3 * x).filled(.black).aligned(to: .bottom)
```

```
.hcat
let labels = input.map { label, _ in
    return text(label, width: 1, height: 0.3).aligned(to: .top)
}.hcat
return bars --- labels
}
```

其中属性 `normalized` 用于等比规范所有的值，并确保最大值等于一。

```
extension Sequence where Iterator.Element == CGFloat {
    var normalized: [CGFloat] {
        let maxVal = reduce(0, Swift.max)
        return map { $0 / maxVal }
    }
}
```

核心数据结构

在我们的库中，将绘制三种类型的元素：椭圆、矩形与文字。利用枚举，可以为这三种情况定义一个数据类型：

```
enum Primitive {
    case ellipse
    case rectangle
    case text(String)
}
```

图表则会用另一个枚举来定义。首先，一个图表可以是一个有确定尺寸的 `Primitive`，即椭圆、矩形或者文字其中之一：

```
case primitive(CGSize, Primitive)
```

接着，可以用两个枚举成员来表示一对左右相邻（水平方向）或上下相邻（垂直方向）的图表。注意一个 `.beside` 图表是如何被递归地定义的——它包含了两个相邻的图表：

```
case beside(Diagram, Diagram)
case below(Diagram, Diagram)
```

为了能定制图表的样式，我们可以为带有样式属性的图表添加一个枚举成员。该成员使得图表的填充色可以被设置（比如，填充椭圆和矩形的颜色）。Attribute 类型会在稍后定义：

```
case attributed(Attribute, Diagram)
```

最后的枚举成员用于描述对齐方式。假设有一个小的矩形和一个大的矩形彼此相邻。默认情况下，小矩形会在垂直方向被居中，如图 10.4 所示：

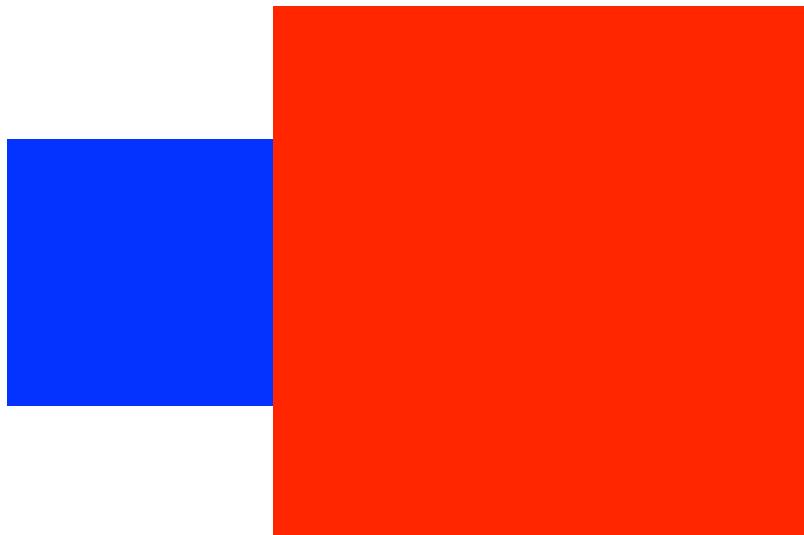


图 10.4: 垂直居中

不过，若是为对齐方式关联一个图表，我们就可以控制图表中较小部分的对齐方式了：

```
case align(CGPoint, Diagram)
```

比如，图 10.5 展示了一张顶部对齐的图表。绘制代码如下：

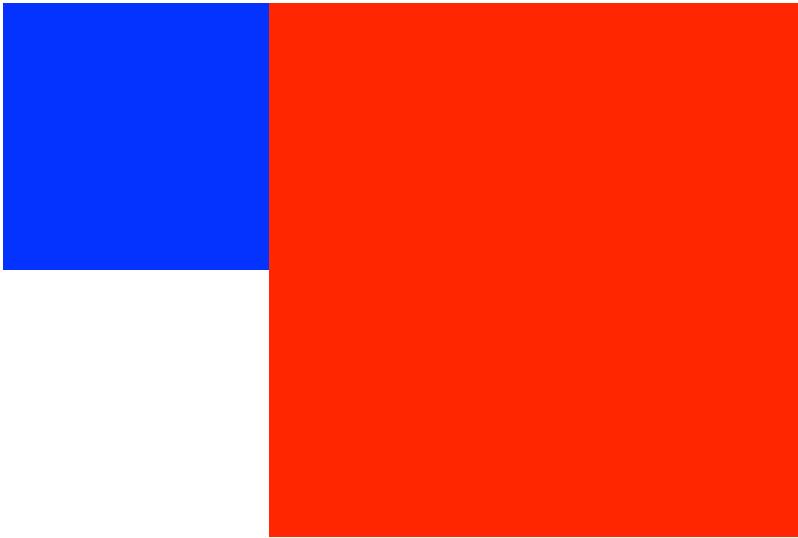


图 10.5: 垂直方向的对齐方式

```
.align(CGPoint(x: 0.5, y: 0), blueSquare) ||| redSquare
```

我们可以使用关键字 `indirect` 将 `Diagram` 定义为一个递归枚举：

```
indirect enum Diagram {  
    case primitive(CGSize, Primitive)  
    case beside(Diagram, Diagram)  
    case below(Diagram, Diagram)  
    case attributed(Attribute, Diagram)  
    case align(CGPoint, Diagram)  
}
```

前文提到的 `Attribute` 枚举是一个用来描述图表各类样式属性的数据结构。它现在只支持 `fillColor`，不过将其拓展以支持描边、渐变、文字排版属性等等的样式属性并不会很麻烦：

```
enum Attribute {  
    case fillColor(UIColor)  
}
```

计算与绘制

除了枚举值 `.beside` 与 `.below` 相对复杂些之外，计算数据类型 `Diagram` 的尺寸其实并不难。在值为 `.beside` 时，宽度等于两个(被关联的)图表宽度之和，而高度则等于左右图表中较高者的高度。`.below` 也是以类似的方式进行计算。除此之外，其它情况只需要递归地调用 `size`：

```
extension Diagram {  
    var size: CGSize {  
        switch self {  
            case .primitive(let size, _):  
                return size  
            case .attributed(_, let x):  
                return x.size  
            case let .beside(l, r):  
                let sizeL = l.size  
                let sizeR = r.size  
                return CGSize(width: sizeL.width + sizeR.width,  
                             height: max(sizeL.height, sizeR.height))  
            case let .below(l, r):  
                return CGSize(width: max(l.size.width, r.size.width),  
                             height: l.size.height + r.size.height)  
            case .align(_, let r):  
                return r.size  
        }  
    }  
}
```

在开始绘制之前，我们还需要再定义一个 `fit` 方法。这个方法会确保在某尺寸值(比如某个图表的尺寸)长宽比不变的情况下，将它依据传入的矩形进行缩放。被等比修正的尺寸值在目标矩形中的坐标值则由一个类型为 `CGPoint` 的参数 `alignment` 传入：如果该 `CGPoint` 的 `x` 为 0 表示左对齐，为 1 则表示右对齐。类似地，`y` 为 0 时表示上对齐，为 1 时则表示下对齐：

```
extension CGSize {  
    func fit(into rect: CGRect, alignment: CGPoint) -> CGRect {  
        let scale = min(rect.width / width, rect.height / height)  
        let targetSize = scale * self  
        let spacerSize = alignment.size * (rect.size - targetSize)  
        return CGRect(origin: rect.origin + spacerSize.point, size: targetSize)  
    }  
}
```

```
}
```

为了让 fit 方法中编写的计算过程更加直观，我们为 CGSize 与 CGPoint 定义了下列运算符：

```
func *(l: CGFloat, r: CGSize) -> CGSize {
    return CGSize(width: l * r.width, height: l * r.height)
}

func *(l: CGSize, r: CGSize) -> CGSize {
    return CGSize(width: l.width * r.width, height: l.height * r.height)
}

func -(l: CGSize, r: CGSize) -> CGSize {
    return CGSize(width: l.width - r.width, height: l.height - r.height)
}

func +(l: CGPoint, r: CGPoint) -> CGPoint {
    return CGPoint(x: l.x + r.x, y: l.y + r.y)
}

extension CGSize {
    var point:CGPoint {
        return CGPoint(x: self.width, y: self.height)
    }
}

extension CGPoint {
    var size:CGSize { return CGSize(width: x, height: y) }
}
```

来试试 fit 方法吧。举个例子，我们希望在一个 200x100 的矩形中适配并居中一个 1x1 的正方形，得出的结果如下：

```
let center = CGPoint(x: 0.5, y: 0.5)
let target = CGRect(x: 0, y: 0, width: 200, height: 100)
CGSize(width: 1, height: 1).fit(into: target, alignment: center)
// (50.0, 0.0, 100.0, 100.0)
```

如果要与矩形左对齐，可以这样写：

```
let topLeft = CGPoint(x: 0, y: 0)
CGSize(width: 1, height: 1).fit(into: target, alignment: topLeft)
```

```
// (0.0, 0.0, 100.0, 100.0)
```

既然我们既可以表示图表，又能计算出它们的尺寸值，将它们绘制出来也就不成问题了。鉴于绘制总是在相同的上下文中进行，所以我们在 CGContext 的扩展中定义一个方法 draw。首先，我们需要对即将被绘制的元素做一些处理。想要将元素绘制在一个 frame 内，我们只需要通过调用该方法来匹配对应的元素就可以了。在这个版本的绘制库中，所有的文本都会被设定为固定字号的系统字体。将其实现为一个样式属性，或是通过修改元素 text 使其可配置化，都不是什么难事：

```
extension CGContext {
    func draw(_ primitive: Primitive, in frame: CGRect) {
        switch primitive {
        case .rectangle:
            fill(frame)
        case .ellipse:
            fillEllipse(in: frame)
        case .text(let text):
            let font = UIFont.systemFont(ofSize: 12)
            let attributes = [NSAttributedString.Key.font: font]
            let attributedText = NSAttributedString(string: text, attributes: attributes)
            attributedText.draw(in: frame)
        }
    }
}
```

接着，我们可以对 draw(_:in:) 进行重载。另一个版本的 draw(_:in:) 会接收两个参数：一个图表，以及用于绘制该图表的矩形边界：

```
extension CGContext {
    func draw(_ diagram: Diagram, in bounds: CGRect) {
        // ...
    }
}
```

在给定边界后，图表会利用之前为 CGSzie 定义的 fit 方法基于该边界进行尺寸适配。想要绘制一个图表的话，我们就要处理 Diagram 中所有的枚举值。

第一个枚举值是 .primitive。这里我们只需要以居中的方式在边界中为元素进行尺寸适配就好：

```
case let .primitive(size, primitive):
    let bounds = size.fit(into: bounds, alignment: .center)
    draw(primitive, in: bounds)
```

在枚举值`.align`的情况下，调用`fit`方法时将该枚举值的关联值传入即可：

```
case .align(let alignment, let diagram):
    let bounds = diagram.size.fit(into: bounds, alignment: alignment)
    draw(diagram, in: bounds)
```

如果在两个图表是水平相邻，也就是枚举值`.beside`描述的情况下，我们需要计算出左侧图表宽度与合并后整体图表宽度的比值，然后根据该比值将绘制边界拆分后，分别绘制左侧与右侧的图形：

```
case let .beside(left, right):
    let (lBounds, rBounds) = bounds.split(
        ratio: left.size.width/diagram.size.width, edge: .minXEdge)
    draw(left, in: lBounds)
    draw(right, in: rBounds)
```

这里我们使用了一个`CGRect`的辅助方法，它会按照指定的比例与拆分方向，将某个矩形平行地拆分：

```
extension CGRect {
    func split(ratio: CGFloat, edge: CGRectEdge) -> (CGRect, CGRect) {
        let length = edge.isHorizontal ? width : height
        return divided(atDistance: length * ratio, from: edge)
    }
}

extension CGRectEdge {
    var isHorizontal: Bool {
        return self == .maxXEdge || self == .minXEdge;
    }
}
```

对枚举值`.below`的处理方式与`.beside`类似：

```
case .below(let top, let bottom):
```

```
let (tBounds, bBounds) = bounds.split(
    ratio: top.size.height/diagram.size.height, edge: .minYEdge)
draw(top, in: tBounds)
draw(bottom, in: bBounds)
```

我们要处理的最后一个枚举值是 `.attributed`。在本章中，我们只支持填充色这一种样式属性，不过想支持其它的样式属性也是非常容易的。要为一个图形绘制出 `fillColor`，我们需要保存当前的绘图状态，然后填充颜色，再绘制图表，最后，恢复之前的绘图状态。

绘制图表的方法 `draw(_:in:)` 完整代码看起来应该像下文这样：

```
extension CGContext {
    func draw(_ diagram: Diagram, in bounds: CGRect) {
        switch diagram {
            case let .primitive(size, primitive):
                let bounds = size.fit(into: bounds, alignment: .center)
                draw(primitive, in: bounds)
            case .align(let alignment, let diagram):
                let bounds = diagram.size.fit(into: bounds, alignment: alignment)
                draw(diagram, in: bounds)
            case let .beside(left, right):
                let (lBounds, rBounds) = bounds.split(
                    ratio: left.size.width/diagram.size.width, edge: .minXEdge)
                draw(left, in: lBounds)
                draw(right, in: rBounds)
            case .below(let top, let bottom):
                let (tBounds, bBounds) = bounds.split(
                    ratio: top.size.height/diagram.size.height, edge: .minYEdge)
                draw(top, in: tBounds)
                draw(bottom, in: bBounds)
            case let .attributed(.fillColor(color), diagram):
                saveGState()
                color.set()
                draw(diagram, in: bounds)
                restoreGState()
        }
    }
}
```

额外的组合算子

为了更容易地构建图表，添加一些额外的函数（也称作组合算子（Combinator））会是个不错的选择。这在函数式库中是一种很普遍的模式：选定一小部分核心的数据类型和函数，然后在它们之上构建一些便利函数。举个例子，对于矩形，圆形，文字，正方形，我们可以定义如下的便利函数：

```
func rect(width: CGFloat, height: CGFloat) -> Diagram {
    return .primitive(CGSize(width: width, height: height), .rectangle)
}

func circle(diameter: CGFloat) -> Diagram {
    return .primitive(CGSize(width: diameter, height: diameter), .ellipse)
}

func text(_ theText: String, width: CGFloat, height: CGFloat) -> Diagram {
    return .primitive(CGSize(width: width, height: height), .text(theText))
}

func square(side: CGFloat) -> Diagram {
    return rect(width: side, height: side)
}
```

值得注意的是我们将我们的组合算子定义为了顶层的函数。将其定义为 `Diagram` 的扩展方法或是初始化方法也是完全可行的，它们只是在调用方式上有所区分罢了。

同时，事实证明为水平或垂直的组合图表添加一个运算符是非常方便的，而这将使代码更为易读。运算符只是将 `.beside` 与 `.below` 封装了起来，鉴于我们还定义了优先级组，在合并图表时可以少写很多括号：

```
precedencegroup HorizontalCombination {
    higherThan: VerticalCombination
    associativity: left
}

infix operator |||: HorizontalCombination
func |||(l: Diagram, r: Diagram) -> Diagram {
    return .beside(l, r)
}
```

```
precedencegroup VerticalCombination {
    associativity:left
}

infix operator --- :VerticalCombination
func ---(l: Diagram, r: Diagram) -> Diagram {
    return .below(l, r)
}
```

我们还可以扩展 Diagram 类型，添加填充和对齐的方法。这些方法也可以被定义为框架的顶层函数。这只是一个风格问题，两者在功能上并没有太大区别：

```
extension Diagram {
    func filled(_ color: UIColor) -> Diagram {
        return .attributed(.fillColor(color), self)
    }

    func aligned(to position: CGPoint) -> Diagram {
        return .align(position, self)
    }
}
```

为了能更方便地调用 .aligned(to:)，我们可以为 CGPoint 定义下面这个扩展：

```
extension CGPoint {
    static let bottom = CGPoint(x: 0.5, y: 1)
    static let top = CGPoint(x: 0.5, y: 1)
    static let center = CGPoint(x: 0.5, y: 0.5)
}
```

最后，我们可以定义一个空图表和水平连接一组图表的方式。只需要使用 reduce 方法就可以实现：

```
extension Diagram {
    init() {
        self = rect(width: 0, height: 0)
    }
}

extension Sequence where Iterator.Element == Diagram {
```

```
var hcat: Diagram {  
    return reduce(Diagram(), ||)|  
}  
}
```

通过添加这些小巧的辅助函数，我们就得到了一个强大的图表绘制库。

讨论

本章代码的灵感来源于 Haskell 的图表库 (Yorgey 2012)。即便我们已经可以绘制简单的图表，本章所展示的库还是有很多可以改进和拓展的方面。有一些功能仍未添加但却易于为之。比如，添加更多的属性和风格选项应该是顺手拈来的。稍微复杂一些的，可能是添加一些转换功能 (比如旋转)，不过这也确确实实是可行的。

当我们将本章中构建的库与 第二章 中的库进行对比时，可以看到很多相似点。两者都是针对某个问题领域 (区域和图表)，并且创建了一个小巧的函数库来描述这个领域。两个库都通过函数提供了一个高度可组合的接口。这两个库都定义了一种 **领域特定语言** (*domain-specific language*，简称 DSL)，并将其嵌入在 Swift 中。每种 DSL 都具有针对性，它们是用于解决特定问题的小型编程语言。

你可能已经接触过很多 DSL，比如正则表达式，SQL，或者 HTML —— 这些语言都不是通用目的编程语言，它们不用于编写**任何**应用，而是更聚焦于解决某种特定类型的问题。正则表达式用于描述文本规则或词法分析，SQL 用于查询数据库，而 HTML 用于描述网页中的内容。

然而，我们在本书中构建的这两种 DSL 之间有一个很重要的区别：在 函数式思想一章中，我们创建的函数根据每一个位置返回一个布尔值。而为了绘制图表，我们构建了一个中间结构，那就是 Diagram 枚举。**浅嵌入**的 DSL 在像 Swift 这样的通用目的的编程语言中不会创建中间数据结构。相反，**深嵌入**则明确地创建了一个中间数据结构，就像本章中编写的 Diagram 枚举那样。

术语“嵌入”是指用于区域或图表的 DSL 是如何被嵌入进 Swift 当中的。它们都有各自的优势。一个浅嵌入 DSL 可以更容易被编写，执行开销更少，且更容易用新的函数进行拓展。而在使用深嵌入时，优点则在于我们很容易对整体结构进行分析，对它进行转换，或者为中间数据结构指定不同的含义。

如果我们想换用深嵌入重写第二章中的 DSL，可能需要定义一个枚举来表示库中不同的函数。枚举成员可以是某个基础区域，比如圆或者正方形，也可以是某个合成的区域，比如这些基础区域组合形成的交集或并集。接着，就可以使用各种方式对这些区域进行分析和计算：生成图

片，检查区域是否是一个基础组件，确定某个给定的点是否在区域中，或是在该中间数据结构上执行任意的计算。

将本章的图表库重写为浅嵌入则复杂些。中间数据结构可以被检视，修改和转换。要定义为一个浅嵌入，我们可能需要对每个希望在 DSL 中支持的操作中直接调用 Core Graphics。比起先创建一个中间结构，直接对绘制操作进行组合会困难得多，毕竟只需要一次渲染，图表就会被完全合并。

迭代器和序列

11

在本章中，我们将关注点放在了迭代器 (Iterators) 和序列 (Sequences) 上，正是它们，组成了 Swift 中 for 循环的基础体系。

迭代器

在 Objective-C 和 Swift 中，我们常常使用数据类型 Array 来表示一组有序元素。虽然数组简单而又快捷，但总会有并不适合用数组来解决的场景出现。举个例子，在总数接近无穷的时候，你可能不想对数组中所有的元素进行计算；或者你并不想使用全部的元素。在这些情况下，你可能会更希望使用一个迭代器来代替数组。

为了说明问题，我们会首先利用与数组运算相似的例子，使你觉得迭代器或许会是更好的选择。

Swift 的 for 循环可以被用于迭代数组元素：

```
for x in array {  
    // do something with x  
}
```

在这样一个 for 循环中，数组会被从头遍历到尾。不过，如果你想要用不同的顺序对数组进行遍历呢？这时，迭代器就可能派上用场。

从概念上来说，一个迭代器是每次根据请求生成数组新元素的“过程”。任何类型只要遵守以下协议，那么它就是一个迭代器：

```
protocol IteratorProtocol {  
    associatedtype Element  
    mutating func next() -> Element?  
}
```

这个协议需要一个由 IteratorProtocol 定义的关联类型：Element，以及一个用于产生新元素的 next 方法，如果新元素存在就返回元素本身，反之则返回 nil。

举个例子，下文的迭代器会从数组的末尾开始生成序列值，一直到 0。Element 的类型可以由方法 next 推断出来，我们不必显式地指定：

```
struct ReverseIndexIterator: IteratorProtocol {  
    var index: Int
```

```
init<T>(array: [T]) {
    index = array.endIndex-1
}

mutating func next() -> Int? {
    guard index >= 0 else { return nil }
    defer { index -= 1 }
    return index
}
}
```

我们声明了一个输入参数为数组的构造方法，然后使用数组的最后一个合法序列值初始化 `index`。

我们可以使用 `ReverseIndexIterator` 来倒序地遍历数组：

```
let letters = ["A", "B", "C"]

()
var iterator = ReverseIndexIterator(array: letters)
while let i = iterator.next() {
    print("Element \(i) of the array is \(letters[i])")
}
/*
Element 2 of the array is C
Element 1 of the array is B
Element 0 of the array is A
*/
```

尽管这个小例子看起来有点小题大做，可迭代器却封装了数组序列值的计算。如果你想要用另一种方式排序序列值，我们只需要更新迭代器，而不必再修改这里的代码。

在某些情况下，迭代器并不需要生成 `nil` 值。比如，我们可以定义一个迭代器用来生成“无数个”二的幂值（直到该值变为某个极大值，致使 `NSDecimalNumber` 溢出）：

```
struct PowerIterator: IteratorProtocol {
    var power: NSDecimalNumber = 1

    mutating func next() -> NSDecimalNumber? {
```

```
    power = power.multiplying(by: 2)
    return power
}
}
```

我们可以使用 PowerIterator 来检视增长中的大数组序列值——比如，在实现一个在每次迭代都为数组序列值乘以二的指数搜索算法时我们就需要这么做。

我们也可能想使用 PowerIterator 做一些完全不同的事情。假设我们希望在二的幂值中搜索一些有趣的值。下文的 find 方法接受一个 (NSDecimalNumber) -> Bool 类型的 predicate 参数并返回符合该条件的最小值：

```
extension PowerIterator {
    mutating func find(where predicate: (NSDecimalNumber) -> Bool)
        -> NSDecimalNumber? {
        while let x = next() {
            if predicate(x) {
                return x
            }
        }
        return nil
    }
}
```

我们可以使用 find 方法来计算二的幂值中大于 1000 的最小值：

```
var powerIterator = PowerIterator()
powerIterator.find { $0.intValue > 1000 } // Optional(1024)
```

迄今为止，我们看到的迭代器都用于生成数字元素，但这并不是必须的。我们也可以编写生成其他类型值的迭代器。比如，下面的迭代器会生成一组字符串，与某个文件中以行为单位的内容相对应：

```
struct FileLinesIterator: IteratorProtocol {
    let lines: [String]
    var currentLine: Int = 0

    init(filename: String) throws {
        let contents: String = try String(contentsOfFile: filename)
```

```

        lines = contents.components(separatedBy: .newlines)
    }

    mutating func next() -> String? {
        guard currentLine < lines.endIndex else { return nil }
        defer { currentLine += 1 }
        return lines[currentLine]
    }
}

```

通过这种定义迭代器的方式，我们将数据的**生成与使用**分离开来。生成过程可能会涉及到打开一个文件或是一个 URL，并且会处理过程中抛出的错误。将这些隐藏在一份简单的迭代器协议之后，可以确保用代码去操作生成的数据时，不必再考虑这些问题。我们的实现方案甚至可以让我们逐行读取文件，而且对于迭代器的使用者来说，他们的代码可以保持不变。

基于为迭代器定义的协议，我们还可以编写一些适用于所有迭代器的泛型函数。举个例子，之前的 `find` 方法的泛型版本如下：

```

extension IteratorProtocol {
    mutating func find(predicate: (Element) -> Bool) -> Element? {
        while let x = next() {
            if predicate(x) {
                return x
            }
        }
        return nil
    }
}

```

`find` 方法现在适用于任意的迭代器。最有趣的事情是它的类型签名，由于调用了 `next`，迭代器可能会被 `find` 方法修改，所以我们需要在类型声明中添加 `mutating` 标注。查询条件应当是一个可以将生成元素映射为布尔值的函数。如果需要引用迭代器中的类型，我们可以使用它的关联类型 `Element`。最后，注意我们查询一个符合条件的值是可能失败的。为此，`find` 会返回一个可选值，在迭代器被耗尽时返回 `nil`。

层级式的组合迭代器也是可行的。比如，你可能希望限制生成元素的个数，缓冲生成的值，或是编码已生成的数据。这里有个小例子，我们构建了一个迭代器转换器，它可以用参数中的 `limit` 值来限制参数迭代器所生成的结果个数：

```
struct LimitIterator<I: IteratorProtocol>: IteratorProtocol {
    var limit = 0
    var iterator: I

    init(limit: Int, iterator: I) {
        self.limit = limit
        self.iterator = iterator
    }

    mutating func next() -> I.Element? {
        guard limit > 0 else { return nil }
        limit -= 1
        return iterator.next()
    }
}
```

在填充固定大小的数组，或是以某种方式缓冲已生成的元素时，这样的迭代器可能会很有用。

在编写迭代器时，为每个迭代器引入一个新的结构体有时是一件很繁琐的事情。Swift 提供了一个简单的 AnyIterator<Element> 结构体，其中的元素类型是一个泛型。要初始化该结构体，既可以传入一个已有的迭代器，也可以传入一个 next 函数：

```
struct AnyIterator<Element>: IteratorProtocol {
    init(_ body: @escaping () -> Element?)
    // ...
}
```

我们会在稍后提供完整的 AnyIterator 定义。这里想指出的是，AnyIterator 不仅实现了 Iterator 协议，也实现了我们在下一节进行讲解的 Sequence 协议。

使用 AnyIterator 可以使我们更为简短地定义迭代器。比如，我们可以像下面的代码一样重写 ReverseIndexIterator：

```
extension Int {
    func countDown() -> AnyIterator<Int> {
        var i = self - 1
        return AnyIterator {
            guard i >= 0 else { return nil }
            defer { i -= 1 }
        }
    }
}
```

```
        return i
    }
}
}
```

我们甚至可以依据 AnyIterator 来定义能够对迭代器进行操作和组合的函数。比如，我们可以拼接两个基础元素类型相同的迭代器，代码如下：

```
func +<I: IteratorProtocol, J: IteratorProtocol>(first: I, second: J)
-> AnyIterator<I.Element> where I.Element == J.Element
{
    var i = first
    var j = second
    return AnyIterator { i.next() ?? j.next() }
}
```

返回的迭代器会先读取 first 迭代器的所有元素；在该迭代器被耗尽之后，则会从 second 迭代器中生成元素。如果两个迭代器都返回 nil，该合成迭代器也会返回 nil。

通过对第二个参数进行延迟化处理，可以改进上文的定义。详细的内容会在本章后续再做介绍。经过延迟化处理的版本会返回完全相同的结果，可如果只需要迭代器的部分结果，下面的版本会更高效：

```
func +<I: IteratorProtocol, J: IteratorProtocol>(
    first: I, second: @escaping @autoclosure () -> J)
-> AnyIterator<I.Element> where I.Element == J.Element
{
    var one = first
    var other: J? = nil
    return AnyIterator {
        if other != nil {
            return other!.next()
        } else if let result = one.next() {
            return result
        } else {
            other = second()
            return other!.next()
        }
    }
}
```

```
}
```

序列

迭代器为 Swift 另一个协议提供了基础类型，这个协议就是**序列**。迭代器提供了一个“单次触发”的机制以反复地计算出下一个元素。这种机制不支持返查或重新生成已经生成过的元素，我们想要做到这个的话就只能再创建一个新的迭代器。协议 SequenceType 则为这些功能提供了一组合适的接口：

```
protocol Sequence {
    associatedtype Iterator: IteratorProtocol
    func makeIterator() -> Iterator
    // ...
}
```

注意协议 Sequence 本身无法确保一个序列在被消费时是否会被破坏。比如，你可以将 readLine() 函数封装在一个序列中。

每一个序列都有一个关联的迭代器类型和一个创建新迭代器的方法。我们可以据此使用该迭代器来遍历序列。举个例子，我们可以使用 ReverseIndexIterator 定义一个序列，用于生成某个数组的一系列倒序序列值：

```
struct ReverseArrayIndices<T>: Sequence {
    let array: [T]

    init(array: [T]) {
        self.array = array
    }

    func makeIterator() -> ReverseIndexIterator {
        return ReverseIndexIterator(array: array)
    }
}
```

每当我们希望遍历 ReverseArrayIndices 结构体中存储的数组时，我们可以调用 makeIterator 方法来生成一个需要的迭代器。下面的例子展示了如何将上述的片段组合在一起：

```
var array = ["one", "two", "three"]
let reverseSequence = ReverseArrayIndices(array: array)
var reverseIterator = reverseSequence.makeIterator()

while let i = reverseIterator.next() {
    print("Index \(i) is \(array[i])")
}

/*
Index 2 is three
Index 1 is two
Index 0 is one
*/
```

对比之前仅仅使用迭代器的例子，同一个序列可以被第二次遍历——为此我们只需要调用 `makeIterator()` 来生成一个新的迭代器就可以了。通过在 `Sequence` 的定义中封装迭代器的创建过程，开发者在使用序列时不必再担心潜在的迭代器创建问题。这与面向对象理念中的将使用和创建进行分离的思想是一脉相承的，代码亦由此具有了更高的内聚性。

Swift 在处理序列时有一个特别的语法。不同于创建一个序列的关联迭代器，你可以编写一个 `for` 循环。比如，我们也可以将以上的代码片段写成这样：

```
for i in ReverseArrayIndices(array: array) {
    print("Index \(i) is \(array[i])")
}

/*
Index 2 is three
Index 1 is two
Index 0 is one
*/
```

实际上，Swift 所做的只是使用 `makeIterator()` 方法生成了一个迭代器，然后重复地调用其 `next` 方法直到返回 `nil`。

之前的 `ReverseArrayIndices` 中比较明显的缺点是，尽管我们可能对一个数组中关联的元素更感兴趣，但它却只生成了数字。不过好消息是，不单单是数组，序列也具有标准的 `map` 和 `filter` 方法：

```
public protocol Sequence {
    public func map<T>(
```

```
_ transform: (Iterator.Element) throws -> T)
rethrows -> [T]

public func filter(
    _ isIncluded: (Iterator.Element) throws -> Bool)
rethrows -> [Iterator.Element]
}
```

想倒序生成数组中的元素，我们可以使用 map 来映射 ReverseArrayIndices：

```
let reverseElements = ReverseArrayIndices(array: array).map { array[$0] }
for x in reverseElements {
    print("Element is \(x)")
}
/*
Element is three
Element is two
Element is one
*/
```

类似地，我们当然也会想从序列中过滤掉某些元素。

值得指出的是，这些 map 和 filter 方法**不会**返回新的序列，而是遍历序列来生成一个数组。同样地，Sequence 协议中有一个内建的方法 reversed()，就会返回一个新的数组。数学家们或许是据此才反对将这样的操作称之为 map (映射) 的，毕竟它们无法原封不动地还原之前的结构 (一个序列)。

另外，在协议 BidirectionalCollection 中的方法 reversed，可以根据序列值返回内部集合的反向视图。不过因为在 Sequence 协议中并没有序列值的概念，在计算出全部的结果之前，我们是无法给出一个高效的 reversed 序列的。

延迟化序列

在对序列进行操作时，我们可以将一些简短易懂的变换步骤组合起来。来看看下文的代码吧。这段代码给出了一个数字数组，在进行过滤之后，再将结果平方：

```
(1...10).filter { $0 % 3 == 0 }.map { $0 * $0 } // [9, 36, 81]
```

我们可以继续在这条表达式后链接其它的操作。相互独立的各个操作的易读，使得表达式的整体含义也便于理解。如果我们将这些步骤写成一个 for 循环，则会像下文这样：

```
var result: [Int] = []
for element in 1...10 {
    if element % 3 == 0 {
        result.append(element * element)
    }
}

result // [9, 36, 81]
```

利用 for 循环编写出的命令式版本会更为复杂。而我们一旦开始添加新的操作，代码会很快失控。再回头看这段代码时，也会难以理解到底在表达什么。而函数式版本则非常浅显：给定一个数组，过滤，然后映射。

不过命令式版本还是有一个好处的：执行起来更快。它只对序列进行了一次迭代，并且将过滤和映射合并为一步。同时，数组 result 也只被创建了一次。在函数式版本中，不止序列被迭代了两次（过滤与映射各一次），还生成了一个过渡数组用于将 filter 的结果传递至 map 操作。

大多数情况下，代码的可读性要比性能更重要。不过，这里我们还是可以进行一些优化。通过使用 LazySequence，我们可以在链式操作的同时，一次性计算出应用了所有操作之后的结果。通过这种方法，每个元素的 filter 与 map 操作也可以被合并为一步：

```
let lazyResult = (1...10).lazy.filter { $0 % 3 == 0 }.map { $0 * $0 }
```

在上述代码中，结果的类型略显复杂，并且新的元素实际上还没有被计算出来。鉴于该类型遵守 Sequence 协议，我们既可以使用一个 for 循环来迭代它，也可以将其转换为一个数组：

```
Array(lazyResult) // [9, 36, 81]
```

在将多个方法同时进行链接时，使用 lazy 来合并所有的循环，就可以写出一段性能足以媲美命令式版本的代码了。

案例研究：遍历二叉树

为了讲解序列和迭代器，我们来考虑定义一个二叉树的遍历工具。重新来看之前在第九章中定义的二叉树：

```
indirect enum BinarySearchTree<Element: Comparable> {
    case leaf
    case node(
        BinarySearchTree<Element>,
        Element,
        BinarySearchTree<Element>
    )
}
```

我们可以使用上文中为迭代器定义的拼接运算符 +，来生成二叉树元素的序列。比如，`makeliterator` 遍历地依次访问了左子树、根节点和右子树：

```
extension BinarySearchTree: Sequence {
    func makeliterator() -> AnyIterator<Element> {
        switch self {
            case .leaf: return AnyIterator { return nil }
            case let .node(l, element, r):
                return l.makeliterator() + CollectionOfOne(element).makeliterator() +
                    r.makeliterator()
        }
    }
}
```

如果树中没有元素，我们会返回一个空迭代器。如果树有一个节点，则使用生成器的拼接运算符，将两个递归调用与该根节点储存的值拼接起来，作为结果返回。`CollectionOfOne` 是标准库中的一个类型。注意我们使用了延迟化方式来定义 +。如果我们用前一种（非延迟的）方式来定义 +，`makeliterator` 方法就需要先访问整棵树，再返回结果了。

案例研究：优化 QuickCheck 的范围收缩

在这一节，我们准备了一个篇幅略长的序列定义案例，即改进我们在QuickCheck 中实现的 `Smaller` 协议。在之前，该协议的定义是这样的：

```
protocol Smaller {
    func smaller() -> Self?
}
```

我们之前使用 Smaller 协议去判断和收缩我们测试中发现的反例，smaller 函数会被重复地调用来生成一个更小的值；如果这个值依然无法通过测试，这意味着存在一个比之前“更好”的反例。我们为数组定义的 Smaller 实例只是尝试着重复地移除数组的第一个元素：

```
extension Array: Smaller {
    func smaller() -> [T]? {
        guard !self.isEmpty else { return nil }
        return Array(dropFirst())
    }
}
```

尽管在某些情况下，这确实有助于收缩反例，但依旧有很多不同的途径可以用于收缩数组。计算出所有可能的子数组是一个非常昂贵的操作。对于一个长度为 n 的数组，会有 2^n 个可能的子数组，它们可能是也可能不是符合条件的反例——生成和测试这些子数组并不是一个好主意。

不同于之前，我们会展示如何使用迭代器来生成一系列更小的值。接着，我们就据此修改我们的 QuickCheck 库来使用以下协议：

```
protocol Smaller {
    func smaller() -> AnyIterator<Self>
}
```

在 QuickCheck 中发现了一个反例时，我们可以在一系列更小值中重复运行我们的测试，直到我们找到足够小的反例。我们只需要再做一件事，为数组（和其他我们可能希望收缩的类型）编写一个 smaller 函数。

不同于只移除数组的第一个元素，我们将计算出一系列数组，每一个新数组都被移除了一个元素。这并不会生成所有可能的子列表，而是生成一个以数组为元素的序列，其中的每个数组都比原始数组少一个元素。利用 AnyIterator，我们可以定义这样的一个函数如下：

```
extension Array {
    func smaller() -> AnyIterator<[Element]> {
        var i = 0
        return AnyIterator {
            guard i < self.endIndex else { return nil }
```

```
    var result = self
    result.remove(at: i)
    i += 1
    return result
}
}
}
```

`smaller` 方法返回的迭代器会持续地追踪变量 `i`。当请求下一个元素时，函数会检查 `i` 是否小于数组的长度。如果小于，就计算一个新数组 `result`，并使 `i` 自增。如果已经访问到原始数组的末尾，则返回 `nil`。

在 `Array` 中有一个以单个 `Sequence` 作为参数的初始化方法。通过这个初始化方法，我们可以按照以下方式来测试我们的迭代器：

```
Array([1, 2, 3].smaller()) // [[2, 3], [1, 3], [1, 2]]
```

我们能做的还有很多。不同于只是移除元素，如果元素本身也支持 `Smaller` 协议的话，我们也可能想要尝试去缩小元素本身。

案例研究：解析 器组合算子

12

在试图将一系列符号 (通常是一组字符) 转换为结构化的数据时，解析器 (parser) 是一个非常有用的工具。有些时候，你可以使用正则表达式作为替代，来解析一串简单的字符串，但正则表达式能做的事很有限，且难以在短时间内理解其含义。一旦你知道了如何去编写一个解析器 (或是如何使用一个已有的解析器库)，它就会变成你工具箱中非常有用的工具之一。

想要创建一个解析器，通常有这样几种方式：首先，你可以手工编码解析器，通过使用类似于标准库里 Scanner 类的这样的辅助工具，或许会让整个过程变得简单一些。或者，你可以使用额外的工具来生成一个解析器，比如 [Bison](#) 或是 [YACC](#)。又或是，你可以使用一个 [解析器组合算子库](#)，在很多情况下，与之前各有优劣的两种方案相比，这种方案会更为折中。

解析器组合算子是一种函数式的解析方案。不同于对解析器的可变状态进行管理 (比如我们即将要处理的字符)，解析器组合算子使用纯函数来避免可变状态。在本章，我们将探索解析器组合算子的工作方式，同时自行编写一些解析器组合算子库的核心部件。作为本章的示例，我们将尝试解析类似 $1+2*3$ 这样简单的算术表达式。

我们的目标并非编写一个全面的组合算子库，而只是学习解析器组合算子库背后的工作原理，以便你能够毫不费力地使用它们。相对于自己编写来说，在大部分情况下，使用一个已有的库总是更好的选择。

解析器的类型

在实现这个解析器组合算子库的核心部分之前，我们需要先思考一个解析器实际上都要做些什么。通常而言，一个解析器会接收一组字符 (一个字符串) 作为输入，如果解析成功，则返回一些结果值与剩下的字符串。如果解析失败，则什么也不返回。我们可以将整个过程总结为一个像下面这样的函数类型：

```
typealias Parser<Result> = (String) -> (Result, String)?
```

当然，解析器不只是用于解析字符，它应当能解析任何被传入的符号序列。为了简单起见，在本章我们只需要专注于解析字符就好。

事实证明，直接操作字符串的性能其实会很差。所以在选择输入和剩余部分的类型时，我们会使用 `String.CharacterView` 而不是字符串。别小看这点微不足道的改动，它会使性能得到大幅提升：

```
typealias Stream = String.CharacterView  
typealias Parser<Result> = (Stream) -> (Result, Stream)?
```

接着，我们会将 Parser 类型定义为一个结构体而不是简单的类型别名。这样做能够使我们将组合算子编写为 Parser 内的方法，而不再是一些无主的函数。同时，也会让代码更容易阅读：

```
struct Parser<Result> {
    typealias Stream = String.CharacterView
    let parse: (Stream) -> (Result, Stream)?
}
```

现在可以实现我们的第一个解析器了。我们接下来会一直用到一个基本的构建单元，一个能够解析出匹配条件字符的解析器：

```
func character(matching condition: @escaping (Character) -> Bool)
-> Parser<Character> {
    // ...
}
```

character 是一个用于构造解析器的便利函数，被构造出来的解析器会尝试去解析一个匹配确定条件的字符。由于这个函数的返回类型是一个 Parser，我们以返回一个新的 Parser 作为开始来实现函数体：

```
func character(matching condition: @escaping (Character) -> Bool)
-> Parser<Character> {
    return Parser(parse: { input in
        // ...
    })
}
```

剩下要做的只是检查 input 的第一个字符是否匹配 condition 中的条件。如果匹配，我们会返回一个包含了第一个字符以及输入字符串剩余部分的多元组。如果不匹配，则直接返回 nil。此外，我们还将用到尾随闭包的语法来调用 Parser 类型的初始化方法：

```
func character(condition: @escaping (Character) -> Bool) -> Parser<Character> {
    return Parser { input in
        guard let char = input.first, condition(char) else { return nil }
        return (char, input.dropFirst())
    }
}
```

来测试下我们的字符解析器吧，试试从一串由多个数字组成的输入字符串中解析数字 "1"：

```
let one = character { $0 == "1" }
one.parse("123".characters)
/*
Optional(("1", Swift.String.CharacterView(_core:
Swift._StringCore(_baseAddress: Optional(0x00000001132fa5b9),
_countAndFlags: 2, _owner: nil), _coreOffset: 1)))
*/
```

为了能更方便的测试解析器，我们会为 Parser 添加一个 run 方法，用于将输入的字符串转化为字符视图，并将解析剩余的部分转换回一个字符串，以便于我们解读解析结果：

```
extension Parser {
    func run(_ string: String) -> (Result, String)? {
        guard let (result, remainder) = parse(string.characters) else { return nil }
        return (result, String(remainder))
    }
}

one.run("123") // Optional(("1", "23"))
```

鉴于我们想解析的是任意的数字而不只是字符 "1"。所以接下来就用之前的 character 函数来创建一个数字解析器吧。为了检查某个字符是不是十进制数字，我们要用到标准库中的 CharacterSet 类。CharacterSet 的 contains 方法希望接收的是一个类型为 UnicodeScalar 的值，但我们希望检查的值的类型却是一个 Character。为此，我们先给 CharacterSet 添加一个小巧的辅助方法：

```
extension CharacterSet {
    func contains(_ c: Character) -> Bool {
        let scalars = String(c).unicodeScalars
        guard scalars.count == 1 else { return false }
        return contains(scalars.first!)
    }
}
```

有了上文的辅助方法，定义数字的解析就简单多了：

```
let digit = character { CharacterSet.decimalDigits.contains($0) }

digit.run("456") // Optional(("4", "56"))
```

接下来，我们将探索如何将这些原子化的解析器组合为更强大的解析器。

组合解析器

我们的第一个目标是解析一个整数而不只是一个单独的数字。为此，我们需要多次执行解析器 `digit`，并将解析结果合并为一个整数值。

首先，我们会创建一个组合算子 `many`，用于多次执行某个解析器并将解析结果作为一个数组返回：

```
extension Parser {  
    var many: Parser<[Result]> {  
        // ...  
    }  
}
```

`many` 的类型是 `Parser<[Result]>`，对于如何编写该属性的内部实现来说，这给了我们一个线索：与 `character` 函数类似，我们需要返回一个新的 `Parser`。在解析器内给定 `parse` 函数的情况下，我们会持续地调用 `self.parse` 并且积聚结果直到解析失败：

```
extension Parser {  
    var many: Parser<[Result]> {  
        return Parser<[Result]> { input in  
            var result: [Result] = []  
            var remainder = input  
            while let (element, newRemainder) = self.parse(remainder) {  
                result.append(element)  
                remainder = newRemainder  
            }  
            return (result, remainder)  
        }  
    }  
}  
  
digit.many.run("123") // Optional(([1, 2, 3], ""))
```

既然我们可以将多个数字解析为一个字符串数组，那仅剩的步骤只是将字符数组转换为一个整数。为了完成这个任务，我们将为 Parser 定义一个 map 方法：

与可选值的 map 能够将一种类型的可选值转换为另一种类型的可选值类似，Parser 的 map 方法能够把返回值为某种类型的解析器，转换为返回另一种类型的解析器。在从 map 函数返回的新解析器的解析函数中，我们会先尝试调用 self.parse 来解析输入字符串，并在解析失败时直接返回 nil。如果解析成功，则对结果应用函数 transform，然后将新的结果与输入字符的剩余部分一同返回：

```
extension Parser {  
    func map<T>(_ transform: @escaping (Result) -> T) -> Parser<T> {  
        return Parser<T> { input in  
            guard let (result, remainder) = self.parse(input) else { return nil }  
            return (transform(result), remainder)  
        }  
    }  
}
```

在有了 many 和 map 后，再定义我们的整数解析器就简单多了：

```
let integer = digit.many.map { Int(String($0))! }  
  
integer.run("123") // Optional((123, ""))
```

如果我们的解析器解析了一串不止包含数字的输入字符串，从第一个不是数字的字符开始，后面的字符串都将作为剩余部分返回：

```
integer.run("123abc") // Optional((123, "abc"))
```

值得注意的是当前版本的解析器 digit.many 在解析一个空字符串时也能够解析成功，在我们尝试对 Int 初始化方法的返回值进行强制解包时，这会导致崩溃。我们会在稍后再解决这个问题。

顺序解析

迄今为止，我们能做的只是重复执行某个解析器，并将解析结果组合为一个数组。然而，我们通常更希望连续地执行多个不同的解析器。比如，想要解析一个像 "2*3" 这样的乘法表达式，我们需要解析的是一个整数，接下来的符号 "*", 以及另一个整数。

出于这个目的，我们将引入一个顺序组合算子 `followed(by:)`。就像实现 `many` 那样，我们将这个组合算子作为 `Parser` 的一个方法来实现。`followed(by:)` 接收另一个解析器作为参数，然后返回一个全新的解析器，该解析器会将前两个解析器的结果组合为一个多元组作为结果返回：

```
extension Parser {  
    func followed<A>(by other: Parser<A>) -> Parser<(Result, A)> {  
        // ...  
    }  
}
```

`followed(by:)` 有一个泛型参数 `A`，基于此，在调用 `followed(by:)` 的过程中，被调用的解析器与即将作为参数传入方法的解析器可以具有不同的类型。

与之前相似，这个方法的返回类型 `Parser<(Result, A)>`，又给了我们实现方法内部的线索。我们要返回的解析器，其 `parse` 函数的返回值类型必须为 `(Result, A)`。因此，我们先尝试为传入的参数调用 `self.parse`。如果解析成功，我们对 `self.parse` 返回的剩余部分调用 `other.parse`。如果再次解析成功，我们返回被合并的结果。在其它情况下，都会返回 `nil`：

```
extension Parser {  
    func followed<A>(by other: Parser<A>) -> Parser<(Result, A)> {  
        return Parser<(Result, A)> { input in  
            guard let (result1, remainder1) = self.parse(input) else { return nil }  
            guard let (result2, remainder2) = other.parse(remainder1)  
            else { return nil }  
            return ((result1, result2), remainder2)  
        }  
    }  
}
```

利用 `followed(by:)` 和之前为整数和字符定义的解析器，现在可以定义一个乘法表达式的解析器了：

```
let multiplication = integer  
.followed(by: character { $0 == "*" })  
.followed(by: integer)  
multiplication.run("2*3") // Optional(((2, "*"), 3), "")
```

由于越多次的调用 `followed(by:)` 会导致越深层级的多元组嵌套，上述的解析结果看起来实在复杂了些。我们会在稍后针对这种情况做一些改进，不过先让我们使用之前定义的 `map` 方法把这个乘法表达式的结果真正计算出来吧：

```
let multiplication2 = multiplication.map { $0.0 * $1 }
multiplication2.run("2*3") // Optional((6, ""))
```

通过观察传入 `map` 的转换函数，你大概已经看到嵌套数组带来的问题了：这个函数的第一个参数是一个包含了第一个整数值与运算符号的多元组，而第二个参数则是第二个整数值。以上代码的表达已经足够让人费解了，如果再组合更多的解析器，情况只会更糟。

改进顺序解析

稍微倒退一步，来分析一下问题的源头吧。每一次我们使用 `followed(by:)` 来组合两个解析器，其返回值都是一个多元组。由于两个解析器的解析结果可以有不同的类型，所以结果只能是一个多元组（或者类似的数据类型）。也所以像在组合算子 `many` 中做的那样，将结果积聚成一个数组是不可行的——至少要确保不丢失任何的类型信息。

与其先将它们包裹在一个嵌套的多元组里然后再去计算，不如依次向每个解析器的结果传入一个可执行的函数，倘若如此做，我们就可以避免这个问题。

事实上，有一种技术非常适合解决这个问题：柯里化。我们已经在封装 Core Image 一章中讨论过关于柯里化的话题。实际上，我们可以以两种方式来表示一个拥有两个甚至更多参数的函数：既可以表示为一个一次性获取所有参数的函数，也可以表示为一个每次只获取一个参数的柯里化函数。来看看对于我们的乘法函数，这意味着什么。

首先我们可以将之前传入 `map` 中的转换函数剥离出来：

```
func multiply(lhs: (Int, Character), rhs: Int) -> Int {
    return lhs.0 * rhs
}
```

鉴于无论如何都要打散嵌套的多元组，我们还可以将这个函数写得更可读一些：

```
func multiply(_ x: Int, _ op: Character, _ y: Int) -> Int {
    return x * y
}
```

同一个函数的柯里化版本会像下文这样：

```
func curriedMultiply(_ x:Int) -> (Character) -> (Int) -> Int {  
    return { op in  
        return {y in  
            return x * y  
        }  
    }  
}
```

柯里化版本函数的调用部分看起来非常不同。在给定参数 2, "*", 和 3 时，我们可以像这样来调用该函数：

```
curriedMultiply(2)("*")(3) // 6
```

至此，我们就会意识到这将帮助我们来避免多元组嵌套的问题，因为我们可以每次只将一个解析器的解析结果传入该函数。

把 `curriedMultiply` 的调用解读得再细致一些吧。如果我们只传入第一个参数 2 来调用它，返回值会是一个类型为 `(Character) -> (Int) -> Int` 的函数。同理，在传入下一个参数 "*" 来调用返回的函数后，会返回另一个函数，不过这次的类型是 `(Int) -> Int`。在对返回结果传入仅剩的参数 3 之后，最终得出了这个乘法算式的结果。

顺便一提，手动编写每一个柯里化函数可能是一件有点枯燥的事。所以，我们也可以定义一个函数，`curry`，用于将参数个数确定的非柯里化函数转化为柯里化版本。比如，对于双参函数来说，`curry` 可以这样定义：

```
func curry<A, B, C>(_ f: @escaping (A, B) -> C) -> (A) -> (B) -> C {  
    return { a in { b in f(a, b) } }  
}
```

那么如何使用 `curriedMultiply` 来简化我们的乘法解析器呢？第一个整数解析器的解析结果的类型与 `curriedMultiply` 的第一个参数类型是相同的。所以我们可以使用 `map`，对解析器 `integer` 的结果应用 `curriedMultiply`：

```
let p1 = integer.map(curriedMultiply)
```

猜猜得出的解析器会是什么样的类型？

`p1` 的结果类型现在与仅传入了第一个参数的 `curriedMultiply` 是相同的：
`(Character) -> (Int) -> Int`。到这里你可能会觉得有些困惑，不过稍微坚持一下，稍后一切都会变得明朗起来。

接下来，我们会重新使用 `followed(by:)` 来添加乘法符号的解析器：

```
let p2 = p1.followed(by: character { $0 == "✳️" })
```

`p2` 的解析结果类型现在是一个多元组：多元组的第一个元素类型是 `p1` 的解析结果类型
`(Character) -> (Int) -> Int`。第二个元素的类型则是字符解析器的结果类型 `Character`。所以我们继续使用 `map`，将多元组的第二个元素作为参数传入第一个元素中：

```
let p3 = p2.map { f, op in f(op) }
```

到这里，`p3` 的结果类型变为了 `(Int) -> Int`。所以我们只需要重复之前的过程，添加最后的整数解析器：

```
let p4 = p3.followed(by: integer)
let p5 = p4.map { f, y in f(y) }
```

现在，`p5` 的解析结果类型是 `Int` 了：

```
p5.run("2✳️3") // Optional((6, ""))
```

让我们重新将乘法算式的解析器编写为一段连贯的代码：

```
let multiplication3 =
    integer.map(curriedMultiply)
        .followed(by: character { $0 == "✳️" }).map { f, op in f(op) }
        .followed(by: integer).map { f, y in f(y) }
```

再也没有嵌套的多元组了！不过说句实话，这段代码依旧令人困惑，幸好，我们还可以再做些改进。

仔细观察上面的代码，我们会发现后两行的模式是通用的：先利用另一个解析器作为参数来调用 `followed(by:)`，然后对结果进行映射。而在每次调用 `map` 时，传入的转换函数实际上是一样的，唯一的不同只是参数名。我们可以将这种通用的模式抽象为一个顺序解析运算符 `<*>`：

```
func <*>(lhs: Parser<...>, rhs: Parser<...>) -> Parser<...> {
    return lhs.followed(by: rhs).map { f, x in f(x) }
}
```

我们只需要处理好参数与返回值的类型就可以了。在 multiplication3 中第一次调用 followed(by:) 时，由于我们知道 integer.map(curriedMultiply) 的类型，所以我们可以推测出左侧解析器的结果类型会是什么：这是一个接收单个参数并返回某个值的函数。因此，右侧解析器的类型需要与左侧的解析器的函数参数的类型相匹配。最后，被返回的解析器的结果类型需要与左侧解析器函数的返回值类型相同：

```
func <*><A, B>(lhs: Parser<(A) -> B>, rhs: Parser<A>) -> Parser<B> {
    return lhs.followed(by: rhs).map { f, x in f(x) }
}
```

为了使这个运算符能够投入使用，我们还需要将其指定为一个中缀运算符，并指定它的运算方向与优先级：

```
precedencegroup SequencePrecedence {
    associativity: left
    higherThan: AdditionPrecedence
}
```

```
infix operator <*>: SequencePrecedence
```

一旦你熟悉了这个运算符，解析器的代码会因此变得更为易读：

```
let multiplication4 =
    integer.map(curriedMultiply) <*> character { $0 == "*" } <*> integer
```

这样看起来就好多了。不过还有一点瑕疵，.map(curriedMultiply) 的调用位于第一个整数解析器与其它解析器的中间。如果我们能调换调用顺序的话，整句代码读起来会更像一个语法定义。所以这样做吧，我们定义一个应用运算符 <^>，用于将一个前置的函数传入解析器的 map 方法中：

```
infix operator <^>: SequencePrecedence
func <^><A, B>(lhs: @escaping (A) -> B, rhs: Parser<A>) -> Parser<B> {
    return rhs.map(lhs)
}
```

现在我们可以这样来编写一个乘法算式的解析器了：

```
let multiplication5 =  
    curriedMultiply <^> integer <*> character { $0 == "*" } <*> integer
```

一旦你习惯了这样的语法，解读某个解析器的具体功能就会变得非常容易。你可以像阅读一个函数调用一样来阅读运算符 <*>，curriedMultiply 被调用并传入了三个参数，分别是整数解析器、字符解析器、与另一个整数解析器的结果：

```
multiply(integer, character { $0 == "*" }, integer)
```

另一种版本的顺序解析

除了之前定义的 <*>，通常还会顺便定义出另一种版本的顺序解析运算符。这类运算符同样会合并两个解析器，但会丢弃掉第一个解析器的结果。我们在稍后会解析像是 "*" 或 "+" 尾随了一个整数这样的算术表达式，这个版本的运算符就会派上用场。在那些情况中，我们并不关注运算符的解析结果；我们只要确保被解析的符号是存在的就可以了。

我们使用符号 *> 来表示这个运算符，并参照运算符 <*> 来定义它：

```
infix operator *:> SequencePrecedence  
func *><A, B>(lhs: Parser<A>, rhs: Parser<B>) -> Parser<B> {  
    return curry({ _, y in y }) <^> lhs <*> rhs  
}
```

类似地，我们可以再定义一个运算符 <*> 来丢弃右侧解析器的解析结果。我们会在下一章中用到这个运算符：

```
infix operator <*> SequencePrecedence  
func <*><A, B>(lhs: Parser<A>, rhs: Parser<B>) -> Parser<A> {  
    return curry({ x, _ in x }) <^> lhs <*> rhs  
}
```

选择解析

通常来说，我们还会希望解析器以另一种类似于运算符 or 的方式被合并。比如，我们希望解析一个乘号 "*" 或是一个加号 "+"。

出于这个目的，我们为 Parser 添加另一个组合算子方法：

```
extension Parser {  
    func or(_ other: Parser<Result>) -> Parser<Result> {  
        return Parser<Result> { input in  
            return self.parse(input) ?? other.parse(input)  
        }  
    }  
}  
  
let star = character { $0 == "*" }  
let plus = character { $0 == "+" }  
let starOrPlus = star.or(plus)  
starOrPlus.run("+" // Optional(("+", "")))
```

类似于上文的顺序解析运算符，我们可以定义一个选择解析运算符 `<|>`：

```
infix operator <|>  
func <|><A>(lhs: Parser<A>, rhs: Parser<A>) -> Parser<A> {  
    return lhs.or(rhs)  
}
```

有了这个运算符，我们就可以将刚刚的示例编写如下：

```
(star <|> plus).run("+" // Optional(("+", "")))
```

一次或更多次解析

之前我们定义整数解析器时，它还有一个缺陷：我们之前使用的组合算子 `many`，会导致解析器 `integer` 在尝试将一个空字符串解析为一个整数时发生崩溃。这是因为在 `many` 的内部实现中，会利用循环语句多次运行解析器直到解析失败，而即便解析器在一开始就解析失败，也依然会将一个空的数组作为解析结果返回。换句话说，我们需要考虑先运行一次解析器的情况，而不是任由解析器要么因为之前的解析失败而不被运行，要么无论如何都会进入循环运行。

为了解决这个问题，我们将引入另一个组合算子 `many1`，它会先尝试单次运行一个解析器，而后再重复更多次。直到现在才介绍 `many1`，是因为现在我们可以用顺序解析运算符来实现它：

```
extension Parser {
```

```
var many1: Parser<[Result]> {
    return {x in {manyX in [x] + manyX}} <^> self <*> self.many
}
}
```

我们以 `self <*> self.many` 这种写法来满足上文中先单次运行解析器再运行更多次解析的诉求。在 `<^>` 运算符之前的匿名函数是一个柯里化函数，它将 `self` 所得到的单个结果拼接到 `self.many` 解析后所返回的数组之前。这种写法可读性并不高，不过我们可以编写一版非柯里化的函数，然后使用 `curry` 将其转化为柯里化版本：

```
extension Parser {
    var many1: Parser<[Result]> {
        return curry({[$0] + $1}) <^> self <*> self.many
    }
}
```

可选

有些情况下我们会希望表达出某些特定的字符应当被可选地解析，换句话说，无论是解析还是不解析，整个解析过程都会继续如常。我们可以以下文中的可选组合算子 `optional` 来表述这个思路：

```
extension Parser {
    var optional: Parser<Result?> {
        return Parser<Result?> { input in
            guard let (result, remainder) = self.parse(input) else { return (nil, input) }
            return (result, remainder)
        }
    }
}
```

在下一节中，我们会在构建算术表达式的解析器时让这个组合算子派上用场。

解析算术表达式

在我们开始使用我们的解析器组合算子来编写算术表达式的解析器之前，花一点时间来思考这些表达式的语法会对有助于接下来的工作。尤其是对优先级规则的掌控：比如，乘法比加法有更高的优先级。

语法看起来可能会像这样：

```
expression = minus
minus = addition ("-" addition)?
addition = division ("+" division)?
division = multiplication ("/" multiplication)?
multiplication = integer ("*" integer)?
```

优先级规则也被囊括在这个语法中。比如，一个加法表达式被定义为两个除法表达式与一个夹在其中的 "+"。而乘法表达式的结果将被优先计算。

你可能已经注意到这个语法还很简陋。除了对像是类似圆括号这样的语素有明显的缺失以外，还有更严重的问题：比如，一个加法表达式现阶段只能包含一个运算符 "+”，而我们可能会希望计算多个被加数的和。在这个语法里被定义的其它表达式也是相同的状况。要实现这些并不难，但是它会让整个例子更复杂。因此，我们会在默许这些限制的情况下实现解析器。

好消息是，在使用解析器组合算子库时，代码会与之前我们编写的语法非常相似。我们只需要将上述语法从下到上的翻译为组合算子的代码就可以：

```
let multiplication = curry({ $0 * ($1 ?? 1) }) <^>
    integer <*> (character { $0 == "*" } *> integer).optional
```

在运算符 <^> 之前的部分是一个用于计算结果的函数。在这里，它接收两个参数。由于乘法表达式的第二部分在我们的语法中被标记为可选，所以第二个参数是可选的。在运算符 <^> 之后，我们只需要写出表达式中不同部分的解析器就可以了：一个整数解析器，随后是一个符号 "*" 的字符解析器，再之后是另一个整数解析器。由于我们不需要字符解析器的结果，我们使用顺序解析运算符 *> 来丢弃该运算符左侧的解析结果。最后，我们使用组合算子 optional 来标记符号与后一个整数的解析器是可选的。

在之前列出的语法中，其它表达式的解析器也可以使用同样的方式来定义：

```
let division = curry({ $0 / ($1 ?? 1) }) <^>
    multiplication <*> (character { $0 == "/" } *> multiplication).optional
```

```
let addition = curry({ $0 + ($1 ?? 0) }) <^>
    division <*> (character { $0 == "+" } *> division).optional
let minus = curry({ $0 - ($1 ?? 0) }) <^>
    addition <*> (character { $0 == "-" } *> addition).optional
let expression = minus
```

因为所有的解析器都具有相同的结构，我们可以将这些代码重构为更精简的代码，比如编写一个函数来生成指定算术运算符的解析器。不过，在本章的简易示例中，为了避免涉及到另一个抽象层次，我们只关注现有的部分就好。

最后要做的就是测试这个 `expression` 解析器了：

```
expression.run("2*3+4*6/2-10") // Optional((8, ""))
```

在下一章，我们将基于本章的表达式解析器来实现一个简单的电子表格应用。

更 Swift 化的解析器类型

在本章开头，我们是这样定义解析器的类型的：

```
struct Parser<Result> {
    typealias Stream = String.CharacterView
    let parse: (Stream) -> (Result, Stream)?
}
```

这是一种纯函数式的定义方案：函数 `parse` 没有任何副作用，而解析结果与剩余的输入符号序列会作为一个多元组返回。

不过在 Swift 中，其实还有另一种解决方案，这需要用到关键字 `inout`：

```
struct Parser2<Result> {
    typealias Stream = String.CharacterView
    let parse: (inout Stream) -> Result?
}
```

关键字 `inout` 允许我们修改输入的符号序列，使我们能够只返回解析结果而不再是同时包含了结果和剩余符号的多元组。

值得注意的是，`inout` 的作用效果与 Objective-C 中将对某个值的引用进行传递是不同的。我们仍旧可以将该参数当做其它简单的值一样进行操作，区别在于，这个值会在函数返回的同时被复制回去。因此，在使用 `inout` 时并不会产生危及全局的副作用，因为可变性被严格限制在某个特定的变量上了。

在 `parse` 中使用 `inout` 参数可以简化一些组合算子的实现。比如，组合算子 `many` 现在可以编写如下：

```
extension Parser2 {
    var many: Parser2<[Result]> {
        return Parser2<[Result]> { input in
            var result: [Result] = []
            while let element = self.parse(&input) {
                result.append(element)
            }
            return result
        }
    }
}
```

与之前的实现相比，鉴于每次调用 `self.parse` 时可以顺手修改 `input`，所以我们不必再去管理每一次解析之后的剩余部分，

而像是 `or` 这样的组合算子在实现时会更具技巧性：

```
extension Parser2 {
    func or(_ other: Parser2<Result>) -> Parser2<Result> {
        return Parser2<Result> { input in
            let original = input
            if let result = self.parse(&input) { return result }
            input = original // reset input
            return other.parse(&input)
        }
    }
}
```

由于在调用 `self.parse` 时会修改 `input`，我们需要先将 `input` 的值复制到另一个变量中储存起来，以确保在 `self.parse` 返回 `nil` 的时候，我们可以将 `input` 恢复为解析前的值。

两种方案中编写参数的方式都很不错，以至于我们对自己的选择都有点矛盾。最终，我们还是决定在本章使用纯函数式的版本，这是因为我们认为这种方式更适合去解释解析器组合算子背后的思路。

案例研究：构建 一个表格应用

13

在本章中，我们将对上一章中用于解析算术表达式的代码进行拓展，并基于这些代码构建出一个简易的电子表格应用。我们会把这项工作分为三步：首先，我们要为希望被解析的表达式编写解析器；然后，我们会为这些表达式编写一个求值器；最后，我们会将以上代码嵌入一套简单的用户界面中。

解析

在为本章的电子表格表达式构建解析器时，我们会用到在上一章编写的解析器组合算子相关代码。如果你还没有阅读过之前的章节，建议你在阅读完毕之后再返回本章。

相较于前章中算术表达式的解析器，我们将在解析本章的电子表格表达式时引入一个额外的抽象层级。在此之前，我们编写的解析器会直接返回计算结果。比如在解析 "2*3" 这样的乘法表达式时：

```
let multiplication = curry({ $0 * ($1 ?? 1) }) <^>
    integer <*> (character { $0 == "+" } *> integer).optional
```

multiplication 的类型是 `Parser<Int>`，也就是说，在这个传入字符串 "2*3" 并执行解析器之后，会返回整数值 6。

只有在表达式中不含有对任何外部数据的依赖的时候，我们才能够直接将结果计算出来。然而，在电子表格中，我们希望可以表达类似于 A3 这样对其它单元格值的引用，或者是像 SUM(A1:A3) 这样的函数调用。

要支持这些特性，解析器需要将输入的字符串转换为一棵**抽象语法树 (abstract syntax tree, AST)**，这是一种用于描述表达式内容的中间表现形式。在转换为抽象语法树后，我们可以取得这些结构化的数据，再对其做实际的计算。

在更为复杂的脚本中 (比如解析一种编程语言)，你或许需要使用更多的中间层级：先将文本转换为一系列的符号，然后将符号转换为抽象语法树，最后再对语法树进行计算。

我们将这种中间表现形式定义为一个枚举：

```
indirect enum Expression {
    case int(Int)
    case reference(String, Int)
```

```
case infix(Expression, String, Expression)
case function(String, Expression)
}
```

枚举 Expression 包含四个枚举值：

- int 表示简单的数值。
- reference 表示对其它单元格内值的引用，比如 "A3"。其中，列引用被指定为从 "A" 开始的大写字母。而行引用则被定义为从 0 开始的数字。枚举值 reference 有两个关联值，一个字符串和一个整数，用于储存该引用的行与列。
- infix 表示类似 "A2+3" 这样位于运算符左右侧两个参数之间的一次运算。枚举值 infix 中的关联值储存了运算符左侧的表达式，运算符本身，以及右侧的表达式。
- function 表示一个类似于 "SUM(A1:A3)" 这样的函数调用。第一个关联值是函数名，第二个则是函数的参数 (在本章的示例中，函数只能够接收单个参数)。

在有了上文的 Expression 枚举之后，我们就可以为每种表达式编写一个解析器了。

从最简单的枚举值 int 开始吧。这里我们可以利用上一章中的整数解析器，然后将整数结果封装为一个 Expression 类型：

```
extension Expression {
    static var intParser: Parser<Expression> {
        return { .int($0) } <^> integer
    }
}
```

```
Expression.intParser.run("123") // Optional((Expression.int(123), ""))
```

行列坐标的解析器也很简单。我们先定义一个大写字母的解析器：

```
let capitalLetter = character { CharacterSet.uppercaseLetters.contains($0) }
```

然后将大写字母的解析器与一个整数解析器合并起来，再将解析结果封装为枚举值 .reference：

```
extension Expression {
    static var referenceParser: Parser<Expression> {
        return curry({ .reference(String($0), $1) }) <^> capitalLetter <*> integer
    }
}
```

```
    }
}

Expression.referenceParser.run("A3")
// Optional(Expression.reference("A", 3), "")
```

接下来，我们来看看枚举值 `.function`。这里我们需要解析出一个函数名（一个或更多个大写字母），以及接下来的圆括号中的表达式。在本章中，圆括号中的表达式将只支持形如 "A1:A2" 的参数类型。

这里需要先定义两个额外的辅助函数。首先，我们添加了函数 `string`，它会创建一个用于匹配特定字符串的解析器。使用已有的函数 `character` 就可以实现这个函数：

```
func string(_ string: String) -> Parser<String> {
    return Parser<String> { input in
        var remainder = input
        for c in string.characters {
            let parser = character {$0 == c}
            guard let (_, newRemainder) = parser.parse(remainder) else { return nil }
            remainder = newRemainder
        }
        return (string, remainder)
    }
}

string("SUM").run("SUM") // Optional(("SUM", ""))
```

接着，我们为 `Parser` 定义一个便利方法，用来将当前解析器封装在一个用于解析左右圆括号的解析器中：

```
extension Parser {
    var parenthesized: Parser<Result> {
        return string("(") *> self <*> string(")")
    }
}

string("SUM").parenthesized.run("(SUM)") // Optional(("SUM", ""))
```

这里我们使用了运算符 `<*>` 与 `<^>` 将当前解析器与另外两个用于检测圆括号的解析器合并。以这种方式编写时，圆括号的解析器依旧会解析成功，但它们的解析结果会被丢弃。

有了这两个辅助函数，我们就可以继续为函数表达式编写解析器了：

```
extension Expression {  
    static var functionParser: Parser<Expression> {  
        let name = { String($0) } <^> capitalLetter.many1  
        let argument = curry({ Expression.infix($0, String($1), $2) }) <^>  
            referenceParser <*> string(":") <*> referenceParser  
        return curry({ .function($0, $1) }) <^> name <*> argument.parenthesized  
    }  
}
```

在 `functionParser` 中，我们首先定义了一个用于解析函数名的解析器。接下来则为函数的参数定义了解析器，该参数是一个使用 `:` 作为运算符的 `.infix` 表达式，并以另外两个单元格的引用作为运算参数。最后，我们按照先解析函数名，再解析圆括号里函数参数的顺序，将上述解析器进行了合并。

我们还需要处理比如加号或是乘号这样的算术运算符。这实际上是本章示例中最复杂的部分。我们在上一章已经定义了一个用于解析乘法表达式的解析器，然而，之前的版本有一个显而易见的缺点（我们无法解析使用了多个 `**` 的表达式），而且它只能接受整数值作为参数。

在电子表格的表达式中，我们的实现方案必须支持以下所有情况：我们能够进行运算的不应该仅仅是整数，还可以是被引用的值，函数调用，甚至是圆括号中的子表达式。当然，我们还希望能够使用任意的常见算术运算符。

所以首先，我们依旧会定义一个只能对整数进行运算的乘法（或除法）表达式解析器，不过这个解析器可以处理多个运算符。为此，我们先定义一个解析器 `multiplier`，用于解析跟随在整数之后的 `**` 或 `/*` 符号：

```
let multiplier = curry({ ($0, $1) } <^> (string("*") <|> string("//")) <*> intParser
```

该解析器的结果是一个包含运算符与整数的多元组。接着，我们可以定义一个能够零次或多次解析输入字符串的解析器：

```
... <^> intParser <*> multiplier.many
```

这里的挑战在于我们需要编写一个用来合并右侧两个解析器结果的函数。这个函数要接收的参数类型是已知的，它们分别是一个(从 intParser 返回的) Expression，以及一个由解析器 multiplier.many 返回的元素类型为多元组 (String, Expression) 的数组：

```
func combineOperands(first: Expression, _ rest: [(String, Expression)])  
    -> Expression {  
    return rest.reduce(first) { result, pair in  
        return Expression.infix(result, pair.0, pair.1)  
    }  
}
```

该函数使用了 reduce 将第一个表达式与之后由 .infix 表达式返回的 (operator, expression) 合并。如果你不是很熟悉 reduce，可以在 [Map、Filter 和 Reduce](#) 一章中查看更详细的内容。

在有了以上代码后，我们就可以像这样编写我们的第一版 productParser 了：

```
extension Expression {  
    static var productParser: Parser<Expression> {  
        let multiplier = curry({ ($0, $1) }) <^> (string("*)") <|> string("//")) <*> intParser  
        return curry(combineOperands) <^> intParser <*> multiplier.many  
    }  
}
```

现在，我们只需要再解决乘法运算只能在整数间进行的限制就可以了。为此，我们引入了 primitiveParser，它能够解析出任意可以被算术运算符进行运算的元素：

```
extension Expression {  
    static var primitiveParser: Parser<Expression> {  
        return intParser <|> referenceParser <|> functionParser <|>  
            lazy(parser).parenthesized  
    }  
}
```

primitiveParser 使用了选择解析运算符 <|>，定义中可运算的元素可以是一个整数，一个单元格引用，一个函数调用，或者是一个被圆括号括起的子表达式。这里最重要的部分是对辅助函数 lazy 的使用。我们必须确保任意表达式的解析器 parser 仅在必要时才被求值。否则，我们将会陷入无尽的循环之中。为了实现这个功能，lazy 使用 @autoclosure 将参数封装到了一个函数中：

```
func lazy<A>(_ parser: @autoclosure @escaping () -> Parser<A>) -> Parser<A> {
    return Parser<A> { parser().parse($0) }
}
```

现在我们可以用 primitiveParser 替换 intPaser 来编写之前的 productParser 了：

```
extension Expression {
    static var productParser: Parser<Expression> {
        let multiplier = curry({ ($0, $1) }) <^> (string("*) <|> string("//")) <*>
            primitiveParser
        return curry(combineOperands) <^> primitiveParser <*> multiplier.many
    }
}
```

最后，我们仿照 productParser 对 sumParser 进行定义。用于解析算术表达式的完整代码如下：

```
extension Expression {
    static var primitiveParser: Parser<Expression> {
        return intParser <|> referenceParser <|> functionParser <|>
            lazy(parser).parenthesized
    }
}

static var productParser: Parser<Expression> {
    let multiplier = curry({ ($0, $1) }) <^> (string("*) <|> string("//")) <*>
        primitiveParser
    return curry(combineOperands) <^> primitiveParser <*> multiplier.many
}

static var sumParser: Parser<Expression> {
    let summand = curry({ ($0, $1) }) <^> (string("+" <|> string("-")) <*>
        productParser
    return curry(combineOperands) <^> productParser <*> summand.many
}

static var parser = sumParser
}
```

现在，可以用 `Expression.parser` 来解析本章中电子表格应用需求范围内的任意一个表达式了。比如：

```
print(Expression.parser.run("2+4*SUM(A1:A2)")!.0)
/*
infix(Expression.int(2), "+", Expression.infix(Expression.int(4), "*",
Expression.function("SUM", Expression.infix(Expression.reference("A",
1), ":"), Expression.reference("A", 2))))))
*/
```

下面，我们将探究如何对这些表达式的抽象语法树进行求值，来得出用户可读的结果。

求值

在求值阶段，我们的目标是将一棵 `Expression` 树转换为一个实际的结果。首先，我们会定义一个 `Result` 类型：

```
enum Result {
    case int(Int)
    case list([Result])
    case error(String)
}
```

`Result` 类型共有三个枚举值：

- `int` 用于结果为整数的表达式的求值，比如 "2*3" 或是 "SUM(A1:A3)" (假设在单元格 A1 至 A3 中的值都是合法的)。
- `list` 用于返回多个结果的表达式的求值。在本章的示例中，只有像 "A1:A3" 这样的表达式会发生这种情况，这些表达式通常为 "SUM" 或是 "MIN" 这类函数的参数。
- `error` 表示在求值过程中出现的错误，在其关联值中会储存错误的详细说明。

为了对 `Expression` 进行求值，我们会在其拓展中添加一个 `evaluate` 方法：

```
extension Expression {
    func evaluate(context: [Expression?]) -> Result {
        switch (self) {
```

```
// ...
default:
    return .error("Couldn't evaluate expression \($self)")
}
}
}
```

参数 context 是一个数组，其中包含了该电子表格中其他单元格的所有表达式，这是为了能够处理像是 A2 这样的单元格引用。简单起见，我们会将该电子表格限制为只有一列单元格，这样我们就可以使用一个简单的数组来表示所有的表达式了。

接下来，我们只需要对 Expression 中不同的枚举值依次进行匹配，然后返回对应的 Result 值就可以了。下文是 evaluate 方法的全部代码，我们会在随后逐步进行讨论：

```
extension Expression {
    func evaluate(context: [Expression?]?) -> Result {
        switch ($self) {
            case let .int(x):
                return .int(x)
            case let .reference("A", row):
                return context?[row]!.evaluate(context: context)
                    ?? .error("Invalid reference \($self)")
            case .function:
                return $self.evaluateFunction(context: context)
                    ?? .error("Invalid function call \($self)")
            case let .infix(l, op, r):
                return $self.evaluateArithmetic(context: context)
                    ?? $self.evaluateList(context: context)
                    ?? .error("Invalid operator \($op) for operands \($l, $r)")
            default:
                return .error("Couldn't evaluate expression \($self)")
        }
    }
}
```

第一个匹配条件 .int 很容易处理。.int 的关联值已经是一个整数值，我们只需要提取该值然后返回一个 .int 结果值就可以了。

第二个匹配条件 `.reference` 则稍微复杂一些。考虑到本章的电子表格 (因为上文提到的原因) 被限制为仅有 一列，我们只需要匹配对 A 列单元格的引用，并且将第二个关联值与变量 `row` 进行绑定。在获得被引用单元格的行坐标后，我们利用 `context` 参数查询该单元格的表达式，并且为其表达式递归地调用 `evaluate`。在该引用不存在的情况下，我们会将一个 `.error` 作为结果值返回。

第三个匹配条件 `.function`，只是将求值任务转移给了 `Expression` 另一个被命名为 `evaluateFunction` 的方法。虽然我们可以将这部分代码也写入 `evaluate` 方法中，但顾及到本书的排版，为了避免单个方法太过冗长，我们还是决定将这个步骤提了出来。`evaluateFunction` 的代码如下：

```
extension Expression {  
    func evaluateFunction(context: [Expression?]?) -> Result? {  
        guard  
            case let .function(name, parameter) = self,  
            case let .list(list) = parameter.evaluate(context: context)  
            else { return nil }  
        switch name {  
        case "SUM":  
            return list.reduce(.int(0), lift(+))  
        case "MIN":  
            return list.reduce(.int(Int.max), lift { min($0, $1) })  
        default:  
            return .error("Unknown function \(name)")  
        }  
    }  
}
```

`guard` 语句会先检查该方法是否确实被 `.function` 枚举值调用，然后确定表达式中函数的参数求值后的结果是否为一个 `.list`。如果不满足以上某个条件，会直接返回 `nil`。

剩下的 `switch` 语句并不复杂：我们为每个函数名实现一个匹配条件，然后利用 `reduce` 对从 `parameter` 中求得的列表进行相应的计算就可以了。

这里要提到一个重要的细节，`list` 是一个元素类型为 `Result` 的数组。因此，我们不能直接使用标准的算术运算符 (比如 `+`) 或是函数 (比如 `min`)。为此，我们会定义一个函数 `lift`，用于将任意类型为 `(Int, Int) -> Int` 的函数转换为一个类型为 `(Result, Result) -> Result` 的函数：

```
func lift(_ op: @escaping (Int, Int) -> Int) -> ((Result, Result) -> Result) {
```

```

return { lhs, rhs in
    guard case let (.int(x), .int(y)) = (lhs, rhs) else {
        return .error("Invalid operands \(lhs, rhs) for integer operator")
    }
    return .int(op(x, y))
}
}

```

在这里，我们必须先确保我们处理的确实是两个 `Result.int` 值，否则我们会返回一个 `.error` 作为结果。一旦我们获取了两个整数，就可以由参数传入的运算函数 `op` 来计算结果，并将其再封装回一个 `Result.int` 值。

`evaluate` 方法要处理的最后一个条件是枚举值 `.infix`。考虑到可读性，就像在处理 `.function` 枚举值时所做的那样，我们已经将对 `.infix` 表达式的求值代码拆分到了 `Expression` 的两个拓展中。第一个是 `evaluateArithmetic`。这个方法尝试在当 `.infix` 表达式是一个算术表达式时对其进行求值，如果求值失败则会返回 `nil`：

```

extension Expression {
    func evaluateArithmetic(context: [Expression?]) -> Result? {
        guard case let .infix(l, op, r) = self else { return nil }
        let x = l.evaluate(context: context)
        let y = r.evaluate(context: context)
        switch (op) {
            case "+": return lift(+)(x, y)
            case "-": return lift(-)(x, y)
            case "*": return lift(*)(x, y)
            case "/": return lift(/)(x, y)
            default: return nil
        }
    }
}

```

由于这个方法可以被任意的 `Expression` 调用，我们需要先检查我们是否真的在处理一个 `.infix` 表达式。我们同样使用 `guard` 语句来直接获取关联值：左侧的表达式，运算符，以及右侧的表达式。

接着，我们会为左右两侧的表达式调用 `evaluate`，然后通过 `switch` 语句匹配运算符来计算结果。这里我们又一次使用了 `lift` 函数，使像是两个 `Result` 值相加这样的运算变得可行。这里要注意我们不必再考虑 `x` 或 `y` 可能不是 `.int` 的情况，`lift` 函数会搞定的。

Expression 中第二个用于计算 .infix 表达式的方法用于处理列表运算符，比如 "A1:A3":

```
extension Expression {  
    func evaluateList(context: [Expression?]?) -> Result? {  
        guard  
            case let .infix(l, op, r) = self,  
            op == ":",  
            case let .reference("A", row1) = l,  
            case let .reference("A", row2) = r  
            else { return nil }  
        return .list((row1...row2).map  
            { Expression.reference("A", $0).evaluate(context: context) })  
    }  
}
```

再一次，我们会先检查 evaluateList 是否被适当的 Expression 调用。在 guard 语句中，我们检查了我们是否在处理一个 .infix 表达式，其中的运算符是否是一个 ":"，以及左右两个表达式是不是引用 "A" 列中某个单元格的 .reference。如果不匹配其中任何一个条件，我们会直接返回 nil。

如果所有的检查都通过，我们会对从 row1 开始，到包括 row2 在内的所有序列值进行映射，对这些单元格中的表达式依次进行求值，再将映射得到的结果封装在 .list 中作为结果返回。

以上就是我们需要在 Expression 的 evaluate 方法中实现的全部内容了。考虑到我们总是希望在对某个单元格求值时先求得 context 中另一个单元格的值(我们需要能解决对单元格的引用)，我们会添加一个额外的便利方法来对一个表达式数组进行求值：

```
func evaluate(expressions: [Expression?]?) -> [Result] {  
    return expressions.map { $0?.evaluate(context: expressions) ??  
        .error("Invalid expression \($0)") }  
}
```

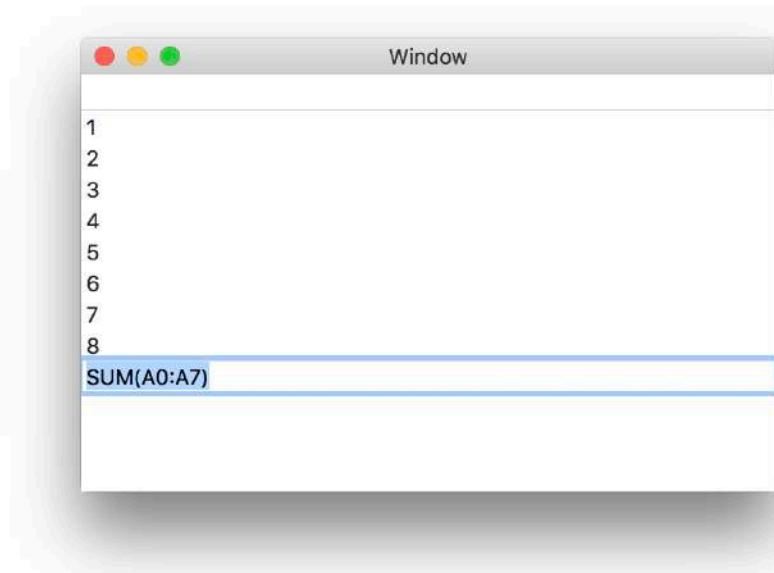
现在我们可以定义一个示例表达式的数组并尝试对它们进行求值：

```
let expressions: [Expression] = [  
    // (1+2)*3  
    .infix(.infix(.int(1), "+", .int(2)), "*", .int(3)),  
    // A0*3  
    .infix(.reference("A", 0), "*", .int(3)),
```

```
// SUM(A0:A1)
.function("SUM", .infix(.reference("A", 0), ":", .reference("A", 1)))
]

evaluate(expressions: expressions)
// [Result.int(9), Result.int(27), Result.int(36)]
```

用户界面



本章中简易电子表格应用的用户界面

我们使用标准的 Cocoa 代码构建了上图中用于嵌入解析与求值代码的用户界面。因此，我们不会在本书中再对这一部分做更多描述。如果你对这些内容感兴趣，可以在 GitHub 上查看该示例项目。

当然，这个应用还有诸多限制，代码也还有很大的优化空间。比如，我们不必因为用户修改了单个单元格的内容就对所有的单元格进行解析和求值。同时，如果 10 个单元格同时引用了一个单元格，那么这个单元格会被求值 10 次。

不过，作为一个小巧且可运行的示例，它也向我们展示出，将以函数式风格编写的代码与传统的面向对象 Cocoa 代码融合起来其实并不难。

函子、适用函子 与单子

14

在本章中，我们会解释一些函数式编程中的专用术语和一些常见模式，比如函子 (Functor)、适用函子 (Applicative Functor) 和单子 (Monad) 等。理解这些常见的模式，会有助于你设计自己的数据类型，并为你的 API 选择更合适的函数。

函子

迄今为止，我们已经遇到了几个被命名为 map 的方法，类型分别如下：

```
extension Array {  
    func map<R>(transform: (Element) -> R) -> [R]  
}  
  
extension Optional {  
    func map<R>(transform: (Wrapped) -> R) -> R?  
}  
  
extension Parser {  
    func map<T>(_ transform: @escaping (Result) -> T) -> Parser<T>  
}
```

为什么这三个截然不同的方法会拥有相同的方法名呢？要回答这个问题，我们需要先探究这三个方法的相似之处。在开始之前，将 Swift 使用的一些简写符号进行展开会帮助我们更容易理解发生了什么。像 Int? 这样的可选值也可以被显式地写作 Optional<Int>。同样地，我们也可以使用 Array<T> 来替换 [T]。如果我们按照上面的书写方式定义数组和可选值的 map 方法，相似之处就变得明显了起来：

```
extension Array {  
    func map<R>(transform: (Element) -> R) -> Array<R>  
}  
  
extension Optional {  
    func map<R>(transform: (Wrapped) -> R) -> Optional<R>  
}  
  
extension Parser {  
    func map<T>(_ transform: @escaping (Result) -> T) -> Parser<T>  
}
```

Optional 与 Array 都是需要一个泛型作为参数来构建具体类型的**类型构造体** (Type Constructor)。对于一个实例来说，Array<T> 与 Optional<Int> 是合法的类型，而 Array 本身却并不是。每个 map 方法都需要两个参数：一个即将被映射的数据结构，和一个类型为 $(T \rightarrow U)$ 的函数 transform。对于数组或可选值参数中所有类型为 T 的值，map 方法会使用 transform 将它们转换为 U。这种支持 map 运算的类型构造体 —— 比如可选值或数组 —— 有时候也被称为**函子** (Functor)。

实际上，我们之前定义的很多类型都是函子。比如，我们可以为第八章中的 Result 类型实现一个 map 方法：

```
extension Result {  
    func map<U>(f: (T) -> U) -> Result<U> {  
        switch self {  
            case let .success(value): return .success(f(value))  
            case let .error(error): return .error(error)  
        }  
    }  
}
```

类似地，我们之前看到的二叉搜索树、字典树以及解析器组合算子等类型都是函子。函子有时会被描述为一个储存特定类型值的“容器”。而 map 方法则用来对储存在容器中的值进行转换。作为一个直观的印象，这样理解或许会有帮助，但却有点过于狭隘。还记得我们在第二章中看到的 Region 类型么？

```
struct Position {  
    var x: Double  
    var y: Double  
}  
  
typealias Region = (Position) -> Bool
```

如果只使用 Region 的定义 (一个返回布尔值的函数)，我们至多能生成一些非黑即白的位图。可以将这个定义泛型化，对每一个位置关联信息的类型做一次抽象：

```
struct Region<T> {  
    let value: (Position) -> T  
}
```

通过这样的定义，我们可以为每个位置关联一个布尔值、RGB 色值、或者任意其他的信息。我们甚至还可以为这些泛型区域定义一个 map 方法。实际上，上述定义可以被归纳为一个复合函数：

```
extension Region {  
    func map<U>(transform: @escaping (T) -> U) -> Region<U> {  
        return Region<U> { pos in transform(self.value(pos)) }  
    }  
}
```

就反驳“函子是一个容器”这一直观印象来说，上文定义的 Region 算得上是一个绝佳的反例。在这里，我们用来表示区域的类型是一个函数，与“容器”可谓是风马牛不相及了。

基本上，你在 Swift 中定义的所有泛型枚举都是一个函子。如果能为这些枚举提供一个强大而又熟悉的 map 方法，我们的开发者小伙伴们一定会乐不可支的。

适用函子

除了 map，许多函子还支持其它的运算。比如第十二章中的解析器，它除了是一个函子之外，还定义了以下运算：

```
func <*><A, B>(lhs: Parser<(A) -> B>, rhs: Parser<A>)  
    -> Parser<B> {
```

运算符 `<*>` 将两个解析器顺序化合并：第一个解析器返回一个函数，而第二个解析器为这个函数返回一个参数。对该运算符的选用并非是偶然。对于任意的类型构造体，如果我们都可以为其定义恰当的 `pure` 与 `<*>` 运算，我们就可以将其称之为一个**适用函子** (Applicative Functor)。或者再严谨一些，对任意一个函子 `F`，如果能支持以下运算，该函子就是一个适用函子：

```
func pure<A>(_ value: A) -> F<A>  
func <*><A, B>(f: F<A -> B>, x: F<A>) -> F<B>
```

虽然我们并没有为 `Parser` 定义 `pure` 运算，但你自己也可以很容易的实现它。实际上，适用函子一直隐藏在本书的各个角落中。比如，上一节定义的 `Region` 也是一个适用函子：

```
precedencegroup Apply { associativity: left }  
infix operator <*>: Apply
```

```
func pure<A>(_ value: A) -> Region<A> {
    return Region { pos in value }
}

func <*><A, B>(regionF: Region<(A) -> B>, regionX: Region<A>) -> Region<B> {
    return Region { pos in regionF.value(pos)(regionX.value(pos)) }
}
```

现在，函数 `pure` 可以使任意区域都返回某个特定的值。而运算符 `<*>` 则会将结果区域的参数 `Position` 分别传入它的两个参数区域，其中一个参数会生成一个类型为 `(A) -> B` 的函数，另一个则会生成一个类型为 `A` 的值。接着，合并这两个区域的方法相信你也猜得到，将第一个参数返回的函数应用在第二个参数生成的值上。

许多为 `Region` 定义的函数都可以借由这两个基础构建模块简短地描述出来。参照第二章中的内容，这里以适用函子的形式编写了一小部分函数作为示例：

```
func everywhere() -> Region<Bool> {
    return pure(true)
}

func invert(region: Region<Bool>) -> Region<Bool> {
    return pure(!) <*> region
}

func intersection(region1: Region<Bool>, region2: Region<Bool>
    -> Region<Bool>
{
    let and: (Bool, Bool) -> Bool = { $0 && $1 }
    return pure(curry(and)) <*> region1 <*> region2
}
```

上述代码展示的，便是利用 `Region` 类型的适用实例逐个定义区域运算的方式。

适用函子并不仅限于区域和解析器。Swift 内建的可选类型就是适用函子的另一个例子。对应的定义简单明了：

```
func pure<A>(_ value: A) -> A? {
    return value
```

```
}
```

```
func <*>(A, B)(optionalTransform: ((A) -> B)?, optionalValue: A?) -> B? {
    guard let transform = optionalTransform,
        let value = optionalValue else { return nil }
    return transform(value)
}
```

pure 函数将一个值封装在可选值中。由于这个过程通常会被 Swift 做隐式的处理，我们定义的这个版本并不是很实用。而运算符 `<*>` 就会更有意思一些：传入一个（可能为 nil 的）函数和一个（可能为 nil 的）参数，它会在两个可选值同时有值时，将函数“适用”在参数上，并将结果返回。如果两个参数中任意一个为 nil，则运算结果也会是 nil。我们也可以为第八章中的 Result 类型定义类似的 pure 与 `<*>`。

这些定义单独来看并不能发挥什么特别厉害的功效，但组合起来就会很有意思。我们可以回顾一些之前的例子，还能回想起之前的 addOptionals 函数么？一个尝试计算两个可能为 nil 的整数之和的函数：

```
func addOptionals(optionalX: Int?, optionalY: Int?) -> Int? {
    guard let x = optionalX, y = optionalY else { return nil }
    return x + y
}
```

利用之前的定义，只需要一条 return 语句，我们就可以定义出一版更为简洁的 addOptionals：

```
func addOptionals(optionalX: Int?, optionalY: Int?) -> Int? {
    return pure(curry(+)) <*> optionalX <*> optionalY
}
```

一旦你理解了 `<*>` 以及类似运算符中所包含的控制流，以上述方式去组织一些复杂的计算就变得轻而易举了。

在可选值章节中还有一个值得回顾的例子：

```
func populationOfCapital(country: String) -> Int? {
    guard let capital = capitals[country], population = cities[capital]
        else { return nil }
    return population * 1000
}
```

在这里，我们需要查阅一个字典 `capitals`，来检索特定国家的首都城市。接着查阅另一个字典 `cities` 来确定该城市的人口数量。尽管与前例中 `addOptionals` 的运算过程差不多，可这个函数却无法以适用的方式来编写。如果我们尝试这样做的话，会发生以下的情况：

```
func populationOfCapital(country: String) -> Int? {  
    return { pop in pop * 1000 } <*> capitals[country] <*> cities[...]  
}
```

这里的问题在于，之前的版本中，第一次查询的结果被绑定在了变量 `capital` 上，而第二次查询需要调用这个变量。如果只使用适用方式来运算，就会被卡在这里：一个适用运算的结果（上例中的 `capitals[country]`）是无法影响另一个适用运算（在 `cities` 中进行查询）的。要解决这个问题，我们还需要其它的接口。

单子

在第五章中，我们给出了 `populationOfCapital` 的另一种定义方式：

```
func populationOfCapital3(country: String) -> Int? {  
    return capitals[country].flatMap { capital in  
        return cities[capital]  
    }.flatMap { population in  
        return population * 1000  
    }  
}
```

在这里，我们使用了内建的 `flatMap` 方法来组合可选值的计算过程。这与适用接口的不同点在哪里呢？答案是，类型。在适用运算符 `<*>` 中，两个参数都是可选值，反观 `flatMap` 方法，第二个参数却是一个返回可选值的函数。也正是因此，我们才得以将第一个字典的检索结果传递给第二个字典来进行查阅。

只在适用函数之间进行组合是无法定义出 `flatMap` 方法的。实际上，`flatMap` 是**单子**结构支持的两个函数之一。更通俗地说，如果一个类型构造体 `F` 定义了下面两个函数，它就是一个**单子**（`Monad`）：

```
func pure<A>(_ value: A) -> F<A>  
  
func flatMap<A, B>(x: F<A>)(_ f: (A) -> F<B>) -> F<B>
```

`flatMap` 函数有时会被定义为一个运算符 `>>=`。鉴于它将第一个参数的计算结果绑定到第二个参数的输入上去，这个运算符也被称为“绑定 (bind)”运算。

除了 Swift 的可选值类型之外，[第八章](#)中定义的枚举 `Result` 也是一个单子。按照这个思路，将一些可能返回 `Error` 的运算连接在一起就变得可行了。比如，我们可以定义一个用来将一个文件的内容复制到另一个文件的函数，如下例：

```
func copyFile(sourcePath: String, targetPath: String, encoding: Encoding)
    -> Result<()
{
    return readFile(sourcePath, encoding).flatMap { contents in
        writeFile(contents, targetPath, encoding)
    }
}
```

如果对 `readFile` 与 `writeFile` 中任何一个方法的调用失败，一个 `Error` 就会被抛出。虽然上例不如 Swift 的可选值绑定机制那么好用，但也十分接近了。

除了处理错误之外，单子还有很多其它的用武之地。比如，数组也是一个单子。在标准库中，数组的 `flatMap` 方法已经被定义了，不过你也可以像这样来实现：

```
func pure<A>(_ value: A) -> [A] {
    return [value]
}

extension Array {
    func flatMap<B>(_ f: (Element) -> [B]) -> [B] {
        return map(f).reduce([]) { result, xs in result + xs }
    }
}
```

我们能从这些定义中得到什么呢？作为一个单子结构，数组提供了一种便利的方式来定义各种各样的可组合函数，又或是解决搜索相关的问题。举个例子，假设我们需要计算两个数组 `xs` 与 `ys` 的笛卡尔积。笛卡尔积是一个由多元组组成的新数组，每个多元组的第一部分都从 `xs` 中抽取，第二部分则从 `ys` 中抽取。直接使用 `for` 循环的话，我们可能会这样写：

```
func cartesianProduct1<A, B>(xs: [A], ys: [B]) -> [(A, B)] {
    var result: [(A, B)] = []
    for x in xs {
        for y in ys {
            result.append((x, y))
        }
    }
    return result
}
```

```
    for y in ys {
        result += [(x, y)]
    }
}
return result
}
```

我们现在可以使用 flatMap 替换 for 循环来重写 cartesianProduct：

```
func cartesianProduct2<A, B>(xs: [A], ys: [B]) -> [(A, B)] {
    return xs.flatMap { x in ys.flatMap { y in [(x, y)] } }
}
```

方法 flatMap 允许我们从第一个数组 xs 中提取一个元素 x，然后在 ys 中提取一个元素 y。对于每一组 x 与 y，我们都会返回一个数组 [(x, y)]。最后，flatMap 方法会将所有这些数组合并为一个结果集。

尽管这个例子看起来有点勉强，flatMap 方法在数组中还是用很多很重要的应用场景。像是 Haskell 与 Python 这样的语言，会专门提供一些语法糖来定义列表，这被称作**列表推导式** (*list comprehensions*)。这些列表推导式允许你从已经存在的列表中抽取元素并检查这些元素是否符合某些指定的规则。所有列表推导式语法糖都可以脱糖为 map、filter 与 flatMap 的组合。列表推导式与 Swift 中的可选值绑定很相似，只不过它们作用于列表而不是可选值。

讨论

为什么要关注这些概念呢？知道某些类型是不是一个适用函子或一个单子真的重要么？我们认为，是的。

回忆一下第十二章中的解析器组合算子吧。定义一种合适的方式来使两个解析器顺序解析不容易：这需要对解析器的工作原理有一些相对深入的了解。在我们的库中，这是必不可少的一个环节，否则，我们连最简单的解析器都写不出来。如果你能发觉我们的解析器被构造为了一个适用函子，你就会意识到之前定义的运算符 `<*>` 为顺序合并两个解析器提供了最优的解决方案。在寻求如此复杂的定义时，对自定义类型所支持的抽象运算有所了解会很有帮助。

像函子这样的抽象概念，也为我们提供了很重要的词汇。如果你偶然遇到了一个名为 map 的函数，你或许能把这个函数的功能猜个八九不离十。若是没有使用精确的术语来描述类似于函子

这样的通用结构的话，你可能就要花时间从各种拍脑门的函数名中发觉，这其实就是一个 map 函数。

另外，你也可以在设计自己的 API 时以这些结构作为参考。如果你定义了一个泛型枚举或是结构体，且恰好支持 map 运算。这是你希望暴露给用户的接口吗？你的数据结构是不是也是一个适用函子呢？它是一个单子么？这个操作会做些什么？一旦你熟悉了这些抽象结构，问题就会一个接一个的蹦出来。

尽管在 Swift 中比在 Haskell 中要难一些，但你依旧可以为任意适用函子定义合适的泛型函数。就好像解析器运算符 `</>` 这样的函数，可以借助适用函数 `pure` 与 `<*>` 来定义一般。此外，我们可能会想为解析器以外的**其它**适用函子重新定义这些函数。若是如此，我们便可以利用这些抽象的结构来编写一些程序，继而在编写过程中接触到一些通用的模式；而这些模式本身，也会在很多场合下派上用场。

在函数式编程的世界中，单子还有一段趣的发展史。最开始，单子是在数学领域里一个被称作**范畴论**的分支中被发展起来的。其与计算机科学的联系通常被归结为 Moggi (1991) 的发现，随后由 Wadler (1992a; 1992b) 将其发扬光大。从那时起，他们就已经开始在 Haskell 这样的函数式语言中使用单子，并对单子的副作用与 I/O (Peyton Jones 2001) 做了控制。而适用函子虽然首先由 McBride and Paterson (2008) 提出，但在当时，其实已经有了很多已知的范例。关于本章中提到的许多抽象概念之间的关系，可以在 Typeclassopedia (Yorgey 2009) 中找到一篇完整的概述。

尾声

15

那么，函数式编程到底是什么呢？许多人（错误地）认为函数式编程只是使用像 `map` 与 `filter` 这样的高阶函数进行编程，这可能有点管中窥豹，只见一斑。

我们在引言中提到过，一段被精心设计的 Swift 函数式程序所应该具有三种特性：模块化、对可变状态的谨慎处理，以及选择合适的类型。而在后续的章节中，这三个概念也被一再地提及。

无论是第三章中的 `Filter` 类型，还是第二章的 `Region`，高阶函数都是一柄定义抽象概念的利器。不过，这只是开始，而非全部。我们为 `Core Image` 库定义的函数式封装提供了一个类型安全且模块化的方法来组织复杂的图像滤镜。而第十一章的生成器和序列则帮助我们对循环迭代进行了抽象。

Swift 先进的类型系统甚至可以在运行代码之前就捕获到许多错误。比如第五章中讲述的可选值类型会对可能为 `nil` 的值做不可信标记；而泛型不仅使代码复用变得简单，更使得类型安全能够被可靠地执行，这些内容都在第四章中有所提及。在第八章与第九章中介绍的枚举和结构体，则为你在自己的代码中精确地构建数据模型时，提供了基本的构建单元。

引用透明的函数更易于被推导和测试。我们在第六章中实现的 `QuickCheck` 库展示了使用高阶函数为引用透明的函数生成随机单元测试的方式。第七章则告诉我们，Swift 对于值类型的谨慎处理使我们得以在程序中自由地共享数据，而无需担心那些无心之失或是预料之外的变化。

译者注：“引用透明的函数”指那些不会产生副作用（side effect），或者说不会改变程序运行状态和修改函数外变量的函数。纯函数式的函数因为不会对函数外的变量产生作用，因此具有引用透明性。

我们可以汇总以上所有的思路来构建一个强大的特定领域的语言。在第十章中构建的图表库与第十二章中的解析器组合算子都定义了一个小的函数集，它们提供的模块化构建单元，足以错综复杂的问题组合出可行的解决方案。而第十三章中的最后一个案例研究，则展示了如何将这些特定领域语言应用到一个完整的程序中去。

最后，我们在本书中见到的许多类型都承担着相似的功能。在第十四章中，我们展示了如何将它们分类，也揭示了其彼此间的关联。

拓展阅读

想更好地磨练函数式编程技能，一种方式是学习 Haskell。其实函数式语言还有很多种，比如 F#，OCaml，Standard ML，Scala，以及 Racket。无论是哪种语言，作为对 Swift 语言的学习

补充，都是很不错的选择。然而，Haskell 却是最能够考验编程思想的语言。随着对 Haskell 理解的深入，你编写 Swift 的方式也将会随之改变。

如今，优秀的 Haskell 书籍与课程遍地生花。Graham Hutton 所著的《**Programming in Haskell**》(2007) 作为一本优秀的入门教程，可以让你熟悉语言基础。《**Learn You a Haskell for Great Good!**》是一本囊括了许多高级话题的免费在线读物。《**Real World Haskell**》中讲解了一些大型的案例研究，以及大量在其它书中难以见到的技术讲解，内容遍及语言特性，调试与自动化测试。Richard Bird 以他的“Functional Pearl”而闻名——这是一些优雅且具有指导性的函数式编程范例，你可以在他的著作《**Pearls of Functional Algorithm Design**》(2010) 以及在线版中看到这些内容。《**The Fun of Programming**》则集合了 Haskell 中嵌入的领域特定语言，涵盖的领域从金融合约到硬件设计(Gibbons and de Moor 2003)。

如果你希望学习更多关于泛型编程语言设计的内容，Benjamin Pierce 的《**Types and Programming Languages**》(2002) 不失为明智之选。Bob Harper 的《**Practical Foundations for Programming Languages**》(2012) 虽然发布较新且更为严谨，可如果你的计算机科学与数学功底不够扎实，可能会觉得像在读天书。

不必觉得对以上资源的涉猎都是必须的，其中绝大部分其实都不是你的菜。不过你需要知道的是，编程语言设计、函数式编程与数学的演进在很大程度上直接影响了 Swift 的设计。

如果你希望继续提高自己的 Swift 技艺——且不只是函数式的部分——我们已经编写了一整本关于 Swift 进阶主题的书，选题从低级编程到高级抽象均有涉及。

结语

对 Swift 来说，现在是最令人激动的时代。这门语言仍然在蹒跚学步。与 Objective-C 相比，它有许多借鉴于其它已存在的函数式编程语言的全新特性，这预示着我们为 iOS 与 OS X 编写程序的方式将产生颠覆性的变化。

与此同时，Swift 的社区发展尚不明朗。人们会接受这些特性么？又或是写着与 Objective-C 相同的代码，只不过少个分号？时间终会给我们答案。在此，我们仅希望能够通过编写此书，使

你对一些函数式编程的概念有所了解。而是否要将本书的内容加以实践，可能要取决于你对 Swift 抱有何种期待。言尽于此，愿我们可以一同续写 Swift 的未来！

Barendregt, H.P. 1984. *The Lambda Calculus, Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. Elsevier.

Bird, Richard. 2010. *Pearls of Functional Algorithm Design*. Cambridge University Press.

Church, Alonzo. 1941. *The Calculi of Lambda-Conversion*. Princeton University Press.

Claessen, Koen, and John Hughes. 2000. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs.” In *ACM Sigplan Notices*, 268–79. ACM Press.
doi:[10.1145/357766.351266](https://doi.org/10.1145/357766.351266).

Gibbons, Jeremy, and Oege de Moor, eds. 2003. *The Fun of Programming*. Palgrave Macmillan.

Girard, Jean-Yves. 1972. “Interprétation Fonctionnelle et élimination Des Coupures de L’arithmétique d’ordre Supérieur.” PhD thesis, Université Paris VII.

Harper, Robert. 2012. *Practical Foundations for Programming Languages*. Cambridge University Press.

Hinze, Ralf, and Ross Paterson. 2006. “Finger Trees: A Simple General-Purpose Data Structure.” *Journal of Functional Programming* 16 (02). Cambridge Univ Press: 197–217.
doi:[10.1017/S0956796805005769](https://doi.org/10.1017/S0956796805005769).

Hudak, P., and M.P. Jones. 1994. “Haskell Vs. Ada Vs. C++ Vs. Awk Vs. ... an Experiment in Software Prototyping Productivity.” Research Report YALEU/DCS/RR-1049. New Haven, CT: Department of Computer Science, Yale University.

Hutton, Graham. 2007. *Programming in Haskell*. Cambridge University Press.

McBride, Conor, and Ross Paterson. 2008. “Applicative Programming with Effects.” *Journal of Functional Programming* 18 (01). Cambridge Univ Press: 1–13.

Moggi, Eugenio. 1991. “Notions of Computation and Monads.” *Information and Computation* 93 (1). Elsevier: 55–92.

Okasaki, C. 1999. *Purely Functional Data Structures*. Cambridge University Press.

Peyton Jones, Simon. 2001. “Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell.” In *Engineering Theories*

of Software Construction, edited by Tony Hoare, Manfred Broy, and Ralf Steinbruggen, 180:47. IOS Press.

Pierce, Benjamin C. 2002. *Types and Programming Languages*. MIT press.

Reynolds, John C. 1974. "Towards a Theory of Type Structure." In *Programming Symposium*, edited by B.Robinet, 19:408–25. Lecture Notes in Computer Science. Springer.

———. 1983. "Types, Abstraction and Parametric Polymorphism." *Information Processing*.

Strachey, Christopher. 2000. "Fundamental Concepts in Programming Languages." *Higher-Order and Symbolic Computation* 13 (1-2). Springer: 11–49.

Wadler, Philip. 1989. "Theorems for Free!" In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, 347–59.

———. 1992a. "Comprehending Monads." *Mathematical Structures in Computer Science* 2 (04). Cambridge Univ Press: 461–93.

———. 1992b. "The Essence of Functional Programming." In *POPL '92: Conference Record of the Nineteenth Annual Acm Sigplan-Sigact Symposium on Principles of Programming Languages*, 1–14. ACM.

Yorgey, Brent. 2009. "The Typeclassopedia." *The Monad. Reader* 13: 17.

Yorgey, Brent A. 2012. "Monoids: Theme and Variations (Functional Pearl)." In *Proceedings of the 2012 Haskell Symposium*, 105–16. Haskell '12. Copenhagen, Denmark. doi:[10.1145/2364506.2364520](https://doi.org/10.1145/2364506.2364520).