

objc.io | objc 中国

函数式 Swift

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

英文版本 2.0 (2015 年 12 月), 中文版本 1.0 (2016 年 4 月)

© 2015 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

译序 6

2 函数式思想

案例： Battleship 9

一等函数 15

类型驱动开发 19

注解 19

3 案例研究：封装 Core Image

滤镜类型 21

构建滤镜 21

组合滤镜 24

理论背景：柯里化 25

讨论 27

4 Map、Filter 和 Reduce

泛型介绍 29

Filter 33

Reduce 34

实际运用 38

泛型和 Any 类型 39

注释 41

5 可选值

案例研究：字典 43

玩转可选值 46

为什么使用可选值？ 52

6 案例研究：QuickCheck

构建 QuickCheck 57

缩小范围 61

随机数组 63

使用 QuickCheck 66

展望 67

7 不可变性的价值

变量和引用 69

值类型与引用类型 70

讨论 73

8 枚举

关于枚举 77

关联值 79

添加泛型 81

Swift 中的错误处理 82

再聊聊可选值 83

数据类型中的代数学 84

为什么使用枚举？ 86

9 纯函数式数据结构

二叉搜索树 88

基于字典树的自动补全 94

讨论 101

10 案例研究：图表

绘制正方形和圆 103

核心数据结构 106

计算与绘制 109

创建视图与 PDF 113

额外的组合算子 114

讨论 116

11 生成器和序列

生成器 119

序列 124

案例研究：遍历二叉树 127

案例研究：优化 QuickCheck 的范围收缩 128

不止是 Map 与 Filter 131

12 案例研究：解析器组合算子

核心部分 136

选择 139

顺序解析 140

便利组合算子 146

一个简单的计算器 150

13 案例研究：构建一个表格应用

示例代码 157

解析器 157

求值器 167

GUI 171

14 函子、适用函子与单子

函子 176

适用函子 177

单子 180

讨论 183

15 尾声

拓展阅读 185

结语 186

参考文献

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

很难给出函数式的准确定义 — 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的一些**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，或者教你建立你的第一个 Xcode 工程，不过我们会尝试在适当的时候引用现有的 Apple 文档。你应当能自如地阅读 Swift 程序，并且熟悉常见的编程概念，如类，方法和变量等。如果你只是刚刚开始学习编程，这本书可能并不适合你。

在这本书中，我们希望让函数式编程易于理解，并消除人们对它的一些偏见。使用这些理念去改善你的代码并不需要你拥有数学的博士学位！函数式编程并不是 Swift 编程的**唯一**方式。但

是我们相信学习函数式编程会为你的工具箱添加一件重要的新工具，不论你使用那种语言，这件工具都会让你成为一个更好的开发者。

示例代码

你可以在我们的 [GitHub 仓库](#) 中找到这本书里所有的示例代码。这个仓库包括一些章节的 playgrounds，以及其它章节的 Swift 文件和 OS X 工程。

书籍更新

随着 Swift 的发展，我们会继续更新和改进这本书。如果你遇到任何错误，或者是想给我们其它类型的反馈，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。

我们想要感谢测试版读者在本书的写作过程中给我们的反馈 (按字母顺序排列)：

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

Chris, Florian, 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。

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：

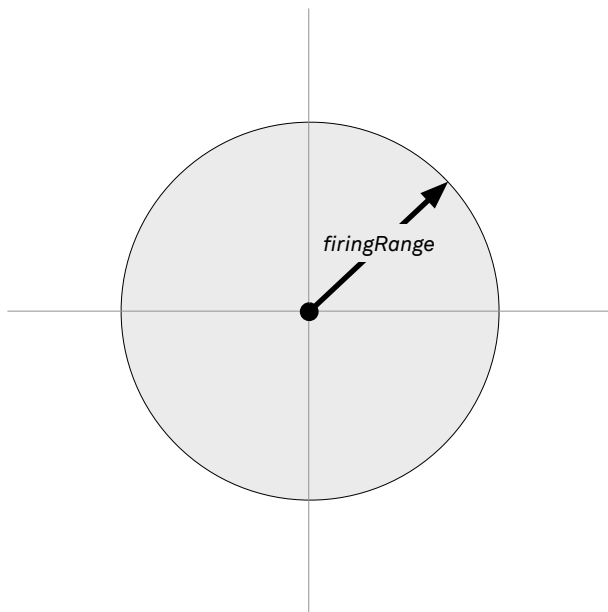


Figure 2.1: 位于原点的船舶射程范围内的点

首先，我们定义两种类型，Distance 和 Position：

```
typealias Distance = Double
```

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

然后我们在 Position 中添加一个函数 inRange(_:)，用于检验一个点是否在图 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 inRange(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：

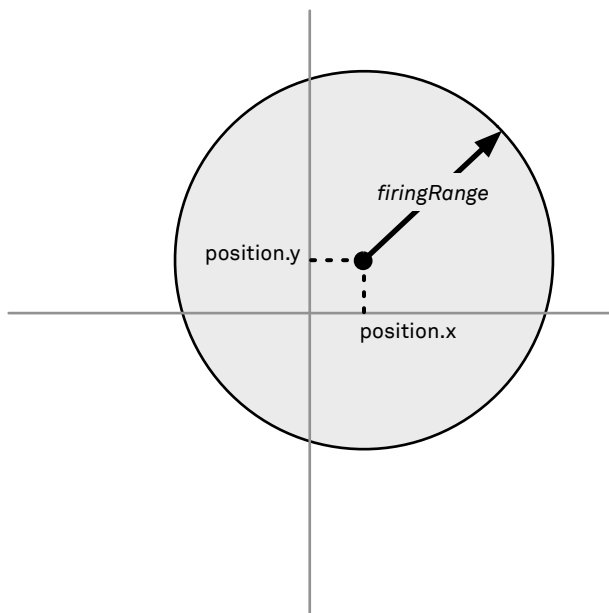


Figure 2.2: 允许船有它自己的位置

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 `canEngageShip(_)` 函数对其进行扩展，这个函数允许我们检验是否有另一艘船在范围内，不论我们是位于原点还是其它任何位置：

```
extension Ship {  
    func canEngageShip(target: Ship) -> Bool {  
        let dx = target.position.x - position.x
```

```

    let dy = target.position.y - position.y
    let targetDistance = sqrt(dx * dx + dy * dy)
    return targetDistance <= firingRange
  }
}

```

也许现在你已经意识到，我们同时还想避免目标船舶离你过近。我们可以用图 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：

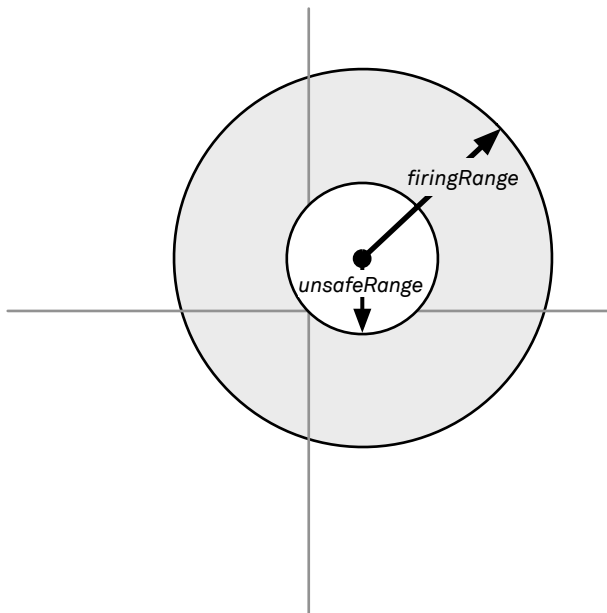


Figure 2.3: 避免与过近的敌方船舶交战

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 canSafelyEngageShip(target: Ship) -> Bool {
    let dx = target.position.x - position.x

```



```

    let dy = target.position.y - position.y
    let targetDistance = sqrt(dx * dx + dy * dy)
    return targetDistance <= firingRange && targetDistance > unsafeRange
  }
}

```

最后，我们还需要避免目标船舶过于靠近我方的任意一艘船。我们再一次将其形象化，见图 2.4：

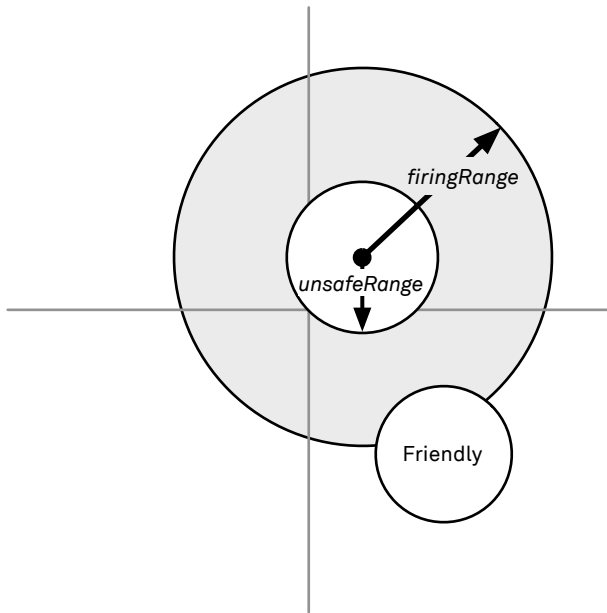


Figure 2.4: 避免敌方过于接近友方船舶

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

```

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

```

```

    let friendlyDx = friendly.position.x - target.position.x
    let friendlyDy = friendly.position.y - target.position.y
    let friendlyDistance = sqrt(friendlyDx * friendlyDx +
        friendlyDy * friendlyDy)
    return targetDistance <= firingRange
        && targetDistance > unsafeRange
        && (friendlyDistance > unsafeRange)
}
}

```

随着代码的发展，它变得越来越难维护。这个函数包含了一大段复杂的计算的代码。我们可以在 `Position` 中添加两个负责几何运算的辅助函数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)
    }
}
}

```

添加了辅助函数之后，函数变成了下面这样：

```

extension Ship {
    func canSafelyEngageShip2(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)
    }
}
}

```

现在的代码已经更易读了，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 `canSafelyEngageShip` 的函数体中，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` 中引入了辅助函数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会像是下面这样的：

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

这个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 }  
}
```

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: Region, offset: Position) -> Region {  
    return { point in region(point.minus(offset)) }  
}
```

调用 `shift(region, offset: 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(_:offset:)` 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：

```
shift(circle(10), offset: 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: 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 intersection(region1: Region, _ region2: Region) -> Region {  
    return { point in region1(point) && region2(point) }
```

```
}
```

```
func union(region1: Region, _ region2: Region) -> Region {  
    return { point in region1(point) || region2(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 difference(region: Region, minus: Region) -> Region {  
    return intersection(region, invert(minus))  
}
```

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 `canSafelyEngageShip(_:friendly:)` 这个复杂的函数：

```
extension Ship {  
    func canSafelyEngageShip(target: Ship, friendly: Ship) -> Bool {  
        let rangeRegion = difference(circle(firingRange),  
                                   minus: circle(unsafeRange))  
        let firingRegion = shift(rangeRegion, offset: position)  
        let friendlyRegion = shift(circle(unsafeRange),  
                                   offset: friendly.position)  
        let resultRegion = difference(firingRegion, minus: friendlyRegion)  
        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 `canSafelyEngageShip1(_:friendly:)` 函数相比，使用 `Region` 函数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，原来庞大的函数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**，也可以做到类似的事情，但遗憾的是，在 Objective-C 中使用 **block** 十分繁琐。一部分原因是因为语法问题：**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函数，来代替对原来的 `Region` 类型进行操作的那些函数。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类型别名以突出在 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。

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 `canSafelyEngageShip1(_:friendly:)` 函数写成一些分开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 kCIOutputImageKey 键取回处理后的图像。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 = CImage -> CImage
```

我们将在这个类型的基础上进行后续的构建。

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(/* parameters */) -> Filter
```

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 = [
            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 CImage 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 CImage -> CImage，也就是 Filter 类型。

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 (CIColorGenerator) 和图像覆盖合成滤镜 (CISourceOverCompositing)。首先让我们来定义一个生成固定颜色的滤镜：

```
func colorGenerator(color: NSColor) -> Filter {
    return { _ in
        guard let c = CIColor(color: color) else { fatalError() }
        let parameters = [kCIInputColorKey: c]
        guard let filter = CIFilter(name: "CIColorGenerator",
            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: UIImage) -> Filter {
    return { image in
        let parameters = [
            kCIInputBackgroundImageKey: image,
            kCIInputImageKey: overlay
        ]
        guard let filter = CIFilter(name: "CISourceOverCompositing",
            withInputParameters: parameters) else { fatalError() }
        guard let outputImage = filter.outputImage else { fatalError() }
        let cropRect = image.extent
        return outputImage.imageByCroppingToRect(cropRect)
    }
}

```

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，而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 colorOverlay(color: NSColor) -> Filter {
    return { image in
        let overlay = colorGenerator(color)(image)
        return compositeSourceOver(overlay)(image)
    }
}

```

我们再次返回了一个接受图像作为参数的函数。`colorOverlay` 函数首先调用了 `colorGenerator` 滤镜。`colorGenerator` 滤镜需要一个 `color` 作为参数，然后返回一个新的滤镜，因此代码片段

`colorGenerator(color)` 是 `Filter` 类型。而 `Filter` 类型本身就是一个从 `CImage` 到 `CImage` 的函数；因此我们可以向 `colorGenerator(color)` 函数传递一个附加的 `CImage` 类型的参数，最终我们能够得到一个 `CImage` 类型的新叠层。这就是我们在定义 `overlay` 的过程中所发生的全部事情，可以大致概括为——首先使用 `colorGenerator` 函数创建一个滤镜，接着向这个滤镜传递一个 `image` 参数来创建新图像。与之类似，返回值 `compositeSourceOver(overlay)(image)` 由一个通过 `compositeSourceOver(overlay)` 函数构建的滤镜和随即被作为参数的 `image` 组成。

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 = NSURL(string: "http://www.objc.io/images/covers/16.jpg")!
let image = CImage(contentsOfURL: 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 blurRadius = 5.0
let overlayColor = NSColor.redColor().colorWithAlphaComponent(0.2)
let blurredImage = blur(blurRadius)(image)
let overlaidImage = colorOverlay(overlayColor)(blurredImage)
```

我们再一次通过创建滤镜来处理图像，例如先创建了 `blur(blurRadius)` 滤镜，然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 = colorOverlay(overlayColor)(blur(blurRadius)(image))
```

然而，由于括号错综复杂，这些代码很快失去了可读性。更好的解决方式是自定义一个运算符来组合滤镜。为了定义该运算符，首先我们要定义一个用于组合滤镜的函数：

```
func composeFilters(filter1: Filter, _ filter2: Filter) -> Filter {
```

```
    return { image in filter2 ( filter1 (image)) }  
}
```

composeFilters 函数接受两个 Filter 类型的参数，并返回一个新定义的滤镜。这个复合滤镜接受一个 CImage 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。我们可以使用复合函数来定义复合滤镜，就像下面这样：

```
let myFilter1 = composeFilters(blur(blurRadius), colorOverlay(overlayColor))  
let result1 = myFilter1(image)
```

为了让代码更具可读性，我们可以再进一步，为组合滤镜引入运算符。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，所以引入运算符极有意义：

```
infix operator >>> { associativity left }  
  
func >>> (filter1: Filter, filter2: Filter) -> Filter {  
    return { image in filter2 ( filter1 (image)) }  
}
```

与之前使用 composeFilters 的方法相同，现在我们可以使用 >>> 运算符达到目的：

```
let myFilter2 = blur(blurRadius) >>> colorOverlay(overlayColor)  
let result2 = myFilter2(image)
```

由于已经定义的运算符 >>> 是左结合的 (left-associative)，就像 Unix 的管道一样，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，我们已经看到了用于定义接受两个参数的函数的两种方法。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 return x + y }
}
```

这里的 add2 函数接受第一个参数 x 之后，返回一个**闭包**，然后等待第二个参数 y。这两个 add 函数的调用方法自然也是不同的：

```
add1(1, 2)
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。

在 Swift 中，我们甚至可以省略 add2 函数的一个 return 关键字和返回类型中的某些括号，然后写为下面这样：

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

函数中的箭头 -> 向右结合。这也就是说，你可以将 A -> B -> C 理解为 A -> (B -> C)。然而在本书中，我们通常会为函数类型引入一个类型别名 (像我们对 Region 和 Filter 类型做的处理一样)，或者是显式地写括号来提升代码的可读性。

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。要得到结果，你并不需要知道像 `kCIOutputImageKey` 一样特殊的字典键，也不需要关心对 `kCIInputImageKey` 或 `kCIInputRadiusKey` 这样的键进行初始化。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**。本章中，我们将在一些来自 Swift 标准库中作用于数组的高阶函数中漫游。伴随这个过程，我们将介绍 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 incrementArray(xs: [Int]) -> [Int] {
    var result: [Int] = []
    for x in xs {
        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 doubleArray1(xs: [Int]) -> [Int] {
    var result: [Int] = []
    for x in xs {
        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 computeIntArray(xs: [Int], transform: Int -> Int) -> [Int] {
    var result: [Int] = []
    for x in xs {
        result.append(transform(x))
    }
}
```

```
    return result
}
```

现在，取决于我们想如何根据原数组得到一个新数组，我们可以向函数传递不同的参数。`doubleArray` 函数和 `incrementArray` 函数都精简为了一行调用 `computeIntArray` 的语句：

```
func doubleArray2(xs: [Int]) -> [Int] {
    return computeIntArray(xs) { x in x * 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 isEvenArray(xs: [Int]) -> [Bool] {
    computeIntArray(xs) { x in x % 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 `computeIntArray` 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 `isEvenArray` 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 `computeIntArray` 函数，接受一个 `Int -> Bool` 类型的函数作为参数。类似下面这样：

```
func computeBoolArray(xs: [Int], transform: Int -> Bool) -> [Bool] {
    var result: [Bool] = []
    for x in xs {
        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。`computeBoolArray` 和 `computeIntArray` 的定义是相同的；唯一的区别在于类型签名 (type signature)。假如我们定义一个相似的函数 `computeStringArray` 来支持 `String` 类型，其函数体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 genericComputeArray1<T>(xs: [Int], transform: Int -> T) -> [T] {
    var result: [T] = []
    for x in xs {
        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 `genericComputeArray<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。

我们仍能进一步将这个函数一般化。没有理由让它仅能对类型为 `[Int]` 的输入数组进行处理。将数组类型进行抽象，能得到下面这样的类型签名：

```
func map<Element, T>(xs: [Element], transform: Element -> T) -> [T] {
    var result: [T] = []
    for x in xs {
        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` 函数甚至比之前看到的 `genericComputeArray` 函数更通用。事实上，我们可以通过 `map` 来定义 `genericComputeArray`：

```
func genericComputeArray2<T>(xs: [Int], transform: Int -> T) -> [T] {
    return map(xs, transform: transform)
}
```

同样的，上述函数的定义并没有什么太过特别之处：函数接受 `xs` 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，最有趣非类型莫属。`genericComputeArray(_:transform:)` 是 `map` 函数的一个实例，只是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(xs, transform) 的替代，我们现在可以通过 xs.map(transform) 来调用 Array 的 map 函数：

```
func genericComputeArray<T>(xs: [Int], transform: Int -> T) -> [T] {
    return xs.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 SequenceType 协议被定义，我们会在之后关于生成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 2.0 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 `SequenceType` 一样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充更完善，暧昧的命名更少，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"]
```

现在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(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(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。我们接下来可以使用同样的函数去比对 `.swift` 或 `.md` 文件。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(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

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

```
}
```

可以像下面一样使用 `sum` 函数：

```
sum([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(xs: [Int]) -> Int {  
    var result: Int = 1  
    for x in xs {  
        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(xs: [String]) -> String {  
    var result: String = ""  
    for x in xs {  
        result += x  
    }  
    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 prettyPrintArray(xs: [String]) -> String {  
    var result: String = "Entries in the array xs:\n"  
    for x in xs {  
        result = "  " + result + x + "\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 `xs` 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。

考虑到这一点，我们得出了能够匹配此模式的 `reduce` 函数定义，如下所示：

```
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  
    }  
}
```

这个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，首先需要有一个 `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 一样的函数式语言中，`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(xs: [Int]) -> Int {  
    return xs.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(xs: [Int]) -> Int {  
    return xs.reduce(1, combine: *)  
}  
  
func concatUsingReduce(xs: [String]) -> String {  
    return xs.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，它不得不反复分配内存，释放内存，以及复制大量内存中的内容。像我们之前做的一样，用一个可变结果数组定义 `map` 的效率显然会更高。理论上，编译器可以优化代码，使其速度与可变结果数组的版本一样快，但是 Swift 2.0 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，我们将给出一个使用了 `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 cityByScalingPopulation() -> 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.cityByScalingPopulation() }
    .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 `cityByScalingPopulation` 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 类型可以避免 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 >>> { associativity left }  
func >>> <A, B, C>(f: A -> B, g: 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 A -> C 类型的函数。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。然而函数必须能处理**任意**类型的 C，所以我们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: (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]。

我们可以使用与数组索引相同的形式来查询与键相关联的值：

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

然而，这个例子无法通过类型检查。问题是 **"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。我们写 `madridPopulation!` 是为了获取可选值中实际的 `Int` 值。后缀运算符 `!` 强制将一个可选类型转换为一个不可选类型。为了计算 Madrid 的总人口数，我们强制将可选的 `madridPopulation` 转换为 `Int` 并与 1000 相乘。

Swift 有一个特殊的**可选绑定** (optional binding) 机制，可以让你避免写 `!` 后缀。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 `default` 的值是通过某个函数或者表达式得到的，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。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 `??`：

`optional ?? defaultValue`

在这个例子中，如果 `optional` 变量是非 `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` 闭包中的代码仅当我们对它进行调用时才会执行。在这样的定义下，代码会如预期一样，只在 `else` 分支中被执行。美中不足的是，当调用 `??` 运算符时需要为默认值创建一个显式闭包。例如，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 标准库中使用的实际定义如下：

```
infix operator ?? { associativity right precedence 110 }
```

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

可选值链

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对方法或属性进行选择。考虑下面处理客户订单的模型的代码片段：

```
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?  
}
```

给定一个 Order，如何才能查找到客户的状态呢？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 myPerson = order.person {
    if let myAddress = myPerson.address {
        if let myState = myAddress.state {
            // ...
        }
    }
}

```

但这未免有些烦琐。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。当任意一个组成项失败时，整条语句链将返回 `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，我们简单地给 `case` 分支中的每个模式添加一个 `?` 后缀。如果我们对一个特定值没有兴趣，也可以直接匹配 `Optional` 的 `None` 值或 `Some` 值：

```

switch madridPopulation {
    case 0?: print("Nobody in Madrid")
    case (1..<1000)??: print("Less than a million in Madrid")
    case .Some(let x): print("\(x) people in Madrid")
    case .None: 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。没有值存在时就提早退出，是一个很常见的使用情境。将它和可选绑定组合在一起可以很好地处理 `None` 的情况。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 populationDescriptionForCity(city: String) -> String? {
    guard let population = cities[city] else { return nil }
}

```

```

    return "The population of Madrid is \((population * 1000)"
}

populationDescriptionForCity("Madrid")

/** 结果为:
    Optional("The population of Madrid is 3165000")
*/

```

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

```

例子 `incrementOptional` 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 `incrementOptional` 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 `incrementOptional` 那样，对一个 `Int?` 类型的值做增量运算，还可以将想要执行的任何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。

我们使用 map 来重写 `incrementOptional`，如下所示：

```
func incrementOptional2(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 函数、适用函数与单子的章节 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 addOptionals(optionalX: Int?, optionalY: Int?) -> Int? {  
    if let x = optionalX {  
        if let y = optionalY {  
            return x + y  
        }  
    }  
    return nil  
}
```

不过，除了层层嵌套，我们还可以同时绑定多个可选：

```
func addOptionals(optionalX: Int?, optionalY: Int?) -> Int? {
    if let x = optionalX, 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 addOptionals(optionalX: Int?, optionalY: Int?) -> Int? {
    guard let x = optionalX, 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 `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], 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>(f: Wrapped -> U?) -> U? {
        guard let x = self else { return nil }
        return f(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 f；若是 **nil**，那么结果也将是 **nil**。

现在可以使用此函数，来重写我们的例子：

```

func addOptionals2(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
            return 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，这样使代码结构更浅显易懂：

```

func populationOfCapital3(country: String) -> Int? {
    return capitals[country].flatMap { capital in
        return cities[capital]
    }.flatMap { population in
        return population * 1000
    }
}

```

我们并不想鼓吹 flatMap 是组合可选值的“正确”方法。恰恰相反，我们希望说明的是 Swift 编译器内置的可选绑定并不神奇，它只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 采用的默认近似零的做法有其弊端，可能你会想要区分失败（键不存在于字典）和成功返回 `nil`（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示这种情况。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，从本质上看，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)) {
        // ...
    }
}

```

现在的问题是：`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 = ... {
    if 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)
```

```
/** 打印：  
    "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)
```

```
/** 打印:  
    "Minus should be commutative" doesn't hold: (3, 2)  
*/
```

使用 Swift 的尾随闭包 (trailing closures) 语法, 我们也可以直接编写测试, 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 }
```

```
/** 打印:  
    "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()
```

```
/** 结果为：  
    994416812  
*/
```

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 Character: Arbitrary {  
    static func arbitrary() -> Character {  
        return Character(UnicodeScalar(Int.random(from: 65, to: 90)))  
    }  
}
```

```
}
```

然后，我们使用下面定义的 `random` 函数，随机生成一个介于 0 到 40 之间的数 `x` 作为字符串长度。接着再生成 `x` 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。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：

```
func tabulate<A>(times: Int, transform: Int -> A) -> [A] {  
    return (0..times).map(transform)  
}
```

```
extension Int {  
    static func random(from from: Int, to: Int) -> Int {  
        return from + (Int(arc4random()) % (to - from))  
    }  
}
```

```
extension String: Arbitrary {  
    static func arbitrary() -> String {  
        let randomLength = Int.random(from: 0, to: 40)  
        let randomCharacters = tabulate(randomLength) { _ in  
            Character.arbitrary()  
        }  
        return String(randomCharacters)  
    }  
}
```

我们通过调用 `tabulate` 函数，对 0 到 `times-1` 的数字使用 `map` 函数，生成一个由 `f(0)`, `f(1)`, ..., `f(times-1)` 组成的数组。`String` 的扩展 `arbitrary` 使用了 `tabulate` 函数来填充一个随机字符数组。

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()`

```
/** 结果为：  
    WTGYFDTCMQCLSKPJULMLHTVWMEUVWMMG  
*/
```

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..
```

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: CGFloat.arbitrary(),
            height: CGFloat.arbitrary())
    }
}
```

```
check1("Area should be at least 0") { (size: CGSize) in size.area >= 0 }
```



```
/** 打印:
    "Area should be at least 0" doesn't hold: (293.39088483094, -230.171975034795)
*/
```

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:
    PMRVWLLVLBRJXMTUFUBCDBVBBE
*/
```

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，我们会将这个过程的自动化。

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?
}
```

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()
```

```
/** 结果为：  
    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 iterateWhile 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：

```
func iterateWhile<A>(condition: A -> Bool, initial : A, next: A -> A?) -> A {  
  if let x = next(initial) where condition(x) {
```

```

        return iterateWhile(condition, initial : x, next: next)
    }
    return initial
}

```

通过使用 `iterateWhile`，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 `iterateWhile` 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(var array: [Int]) -> [Int] {
    if array.isEmpty { return [] }
    let pivot = array.removeAtIndex(0)
    let lesser = array.filter { $0 < pivot }
    let greater = array.filter { $0 >= pivot }
    return qsort(lesser) + [pivot] + qsort(greater)
}

```

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.sort(<)
}
```

然而，编译器警告我们 `[Int]` 不遵循 `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第一项：

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

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(arc4random() % 50)
    return tabulate(randomLength) { _ in Element.arbitrary() }
  }
}
```

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。两者之间唯一的区别是 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..  
numberOfIterations {  
        let value = arbitraryInstance.arbitrary()  
        guard property(value) else {  
            let smallerValue = iterateWhile({ !property($0) },  
                initial : value, next: arbitraryInstance.smaller)  
            print("\(message)" doesn't hold: \(smallerValue)")  
            return  
        }  
    }  
    print("\(message)" passed \(numberOfIterations) 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: X.smaller)
```

```

        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.sort(<)
}

```

```

/** 打印:
    "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。QuickCheck 并不适用于有状态的函数或 APIs。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，以使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。

不可变性的价值

7

Swift 有几个用于控制值的变化方式的机制。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。

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, 正是因是先前的 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 赋值之后，既修改了 sameClassPoint，也修改了 classPoint，原因在于类是引用类型。理解值类型与引用类型之间的区别极其重要——由此你可以预测赋值行为将会如何修改数据，同时哪些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(var point: PointStruct) -> PointStruct {
    point.x = 0
    point.y = 0
    return point
}
```

我们使用这个函数来生成一个点：

```
var structOrigin: PointStruct = setStructToOrigin(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(classPoint)
```

当被赋以一个新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。引用类型顾名思义，我们只是使用了对已经存在的对象或实例的引用，任何对该引用的改变都会修改原来的对象或实例。

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。

在 Swift 中，结构体并不是唯一的值类型。事实上，Swift 几乎所有类型都是值类型，包括数组，字典，数值，布尔值，元组和枚举(将在下一章介绍)。只有类(class)是一个例外。这正说明了 Swift 正在从“面向对象编程”进化到其他的编程范式。

稍后我们会就此对类和结构体加以对比；在此之前，我们先简要地探讨一下迄今为止所看到的的不同形式可变性之间的相互作用。

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

大多 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（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(xs: [Int]) -> Int {
    var result = 0
    for x in xs {
        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(var array: [Int]) -> [Int] {
    if array.isEmpty { return [] }
    let pivot = array.removeAtIndex(0)
    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：

```
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，**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**成员值**，也可以简称为**成员**。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` 中对应的值：

```
extension Encoding {
    var nsStringEncoding: NSStringEncoding {
        switch self {
            case .ASCII: return NSASCIIStringEncoding
            case .NEXTSTEP: return NSNEXTSTEPStringEncoding
            case .JapaneseEUC: return NSJapaneseEUCStringEncoding
            case .UTF8: return NSUTF8StringEncoding
        }
    }
}
```

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?(enc: NSStringEncoding) {
        switch enc {
            case NSASCIIStringEncoding: self = .ASCII
            case NSNEXTSTEPStringEncoding: self = .NEXTSTEP
            case NSJapaneseEUCStringEncoding: self = .JapaneseEUC
            case NSUTF8StringEncoding: 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：

```
func localizedEncodingName(encoding: Encoding) -> String {
    return .localizedNameOfStringEncoding(encoding.rawValue)
}
```

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 `ErrorType`。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: ErrorType {  
    case CapitalNotFound  
    case PopulationNotFound  
}
```

```
enum PopulationResult {  
    case Success(Int)  
    case Error(LookupError)  
}
```

与 `Encoding` 枚举相比，`PopulationResult` 的成员是带有关联值的。它只有两个可能的成员值：`Success` 和 `Error`，每一个成员值都携带了额外的信息：`Success` 关联了一个整数值，对应着国家首都的人口数；而 `Error` 则关联了一个 `ErrorType`。为了方便说明，我们可以像下文这样声明一个 `Success`：

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

类似地，使用 `Error` 成员来创建一个 `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("France") {
  case let .Success(population):
    print("France's capital has \((population) thousand inhabitants")
  case let .Error(error):
    print("Error: \((error)")
}
```

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。不过，与 `mayorOfCapital` 成功时的返回值相冲突的是，之前 `Success` 的关联值并不是一个字符串，而是一个整数。虽然我们可以将返回的整数再转换成对应的字符串，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(ErrorType)  
}
```

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。两个枚举唯一的区别就是 `Success` 和 `Error` 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(ErrorType)  
}
```

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 `ErrorType`。

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 强制你注明哪些代码可能抛出错误，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 populationOfCapital1(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 populationOfCapital1("France")
    print("France's population is \(population)")
} catch {
    print("Lookup error: \(error)")
}
```

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从 Swift 的标准库中复制出来的：

```
enum Optional<T> {
    case None
    case Some(T)
    // ...
}
```

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，我们可以定义我们自己的 `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: ErrorType -> T) -> T {  
    switch result {  
        case let .Success(value):  
            return value  
        case let .Error(error):  
            return handleError(error)  
    }  
}
```

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，不是吗？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 -> B`，和 `g: B -> 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 $\text{Add}\langle T, U \rangle$ 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 的成员相加之和：如果 T 有三个成员，而 U 有七个，那 $\text{Add}\langle T, U \rangle$ 就会有十个可能的成员。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 —— 它没有任何成员。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 ， $\text{Add}\langle T, \text{Zero} \rangle$ 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 $\text{Add}\langle T, \text{Zero} \rangle$ ，而反向的转换则可以通过模式匹配来完成。

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成员的枚举 T ，和另一个包含两个成员的枚举 U 。我们如何去定义一个混合类型 $\text{Times}\langle T, U \rangle$ ，使之包含六个成员呢？如果要满足这个需求， $\text{Times}\langle T, U \rangle$ 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：

```
struct Times<T, U> {  
    let fst: T  
    let snd: 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：

- $\text{Times}\langle \text{One}, T \rangle$ 与 T 是同构的
- $\text{Times}\langle \text{Zero}, T \rangle$ 与 Zero 是同构的
- $\text{Times}\langle T, U \rangle$ 与 $\text{Times}\langle U, T \rangle$ 是同构的

使用枚举和多元组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 `ErrorType` 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 Objective-C 中 `NSSet` 的库来处理无序集合 (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在 Swift 中处理无序集合的完备库，而是为了演示如何用递归式枚举来定义高效的数据结构。

在我们的迷你库中，我们会实现以下四种操作：

- `empty` —— 返回一个空的无序集合
- `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。那么实现这四个操作，简直是探囊取物：

```
func empty<Element>() -> [Element] {
    return []
}

func isEmpty<Element>(set: [Element]) -> Bool {
    return set.isEmpty
}

func contains<Element: Equatable>(x: Element, _ set: [Element]) -> Bool {
    return set.contains(x)
}
```

```
func insert<Element: Equatable>(x: Element, _ set:[Element]) -> [Element] {
    return contains(x, set) ? set : [x] + set
}
```

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
        }
    }
}

```

在树是 `.Leaf` 的基本情况下，可以直接返回 0。而在 `.Node` 的情况下就比较有意思：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
        }
    }
}

```

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
}
}

```

遗憾的是，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 where Element: Comparable {
    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 `SequenceType` 协议的拓展：

```

extension SequenceType {
  func all(predicate: Generator.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, _) 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 (mutation) 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。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)
```

```
/** 结果为：
    ([], [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：

```
func autocomplete(history: [String], textEntered: String) -> [String] {  
    return history.filter { $0.hasPrefix(textEntered) }  
}
```

遗憾的是，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：

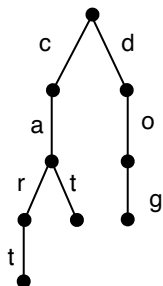


Figure 9.1: 字典树

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地感受到“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。虽然也可以使用 `flatMap` 函数来替换 `for` 循环，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 decompose: (Element, [Element])? {
        return isEmpty ? nil : (self[startIndex], Array(self.dropFirst()))
    }
}
```

decompose 函数会检查一个数组是否为空。如果为空，就返回一个 **nil**；反之，则会返回一个多元组，这个多元组包含该数组的第一个元素，以及去掉第一个元素之后由该数组其余元素组成的新数组。我们可以通过重复调用 decompose 递归地遍历一个数组，直到返回 **nil**，而此时数组将为空。

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(xs: [Int]) -> Int {
    guard let (head, tail) = xs.decompose else { return 0 }
    return head + sum(tail)
}
```

递归式的实现还有另一个不太容易被想到的例子，即使用 decompose 来重写[第六章](#)的 qsort 函数：

```
func qsort(input: [Int]) -> [Int] {
    guard let (pivot, rest) = input.decompose else { return [] }
    let lesser = rest.filter { $0 < pivot }
    let greater = rest.filter { $0 >= pivot }
    return qsort(lesser) + [pivot] + qsort(greater)
}
```

回到我们最开始的问题 —— 我们现在可以使用 decompose 来为数组编写一个查询函数：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: [Element]) -> Bool {
        guard let (head, tail) = key.decompose else { return isElement }
        guard let subtrie = children[head] else { return false }
        return subtrie.lookup(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于树中的布尔值。
- 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 withPrefix(prefix: [Element]) -> Trie<Element>? {  
    guard let (head, tail) = prefix.decompose else { return self }  
    guard let remainder = children[head] else { return nil }  
    return remainder.withPrefix(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 `autocomplete` 函数：

```
extension Trie {  
  func autocomplete(key: [Element]) -> [[Element]] {  
    return withPrefix(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 `withPrefix` 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: [Element]) {
    if let (head, tail) = key.decompose {
      let children = [head: Trie(tail)]
      self = Trie(isElement: false, children: children)
    } else {
      self = Trie(isElement: true, children: [])
    }
  }
}

```

与之前一样，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 insert(key: [Element]) -> Trie<Element> {
    guard let (head, tail) = key.decompose else {
      return Trie(isElement: true, children: children)
    }
    var newChildren = children
    if let nextTrie = children[head] {
      newChildren[head] = nextTrie.insert(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 `insert` 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组字符。或者，我们也可以另写一个 `insert`，以支持所有遵守 `SequenceType` 协议的类型：

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

然后，我们通过调用之前定义的 `autocomplete` 函数，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 `autocomplete` 函数的返回没有包含相同的前缀，只返回了单词剩下的部分：

```
func autocompleteString(knownWords: Trie<Character>, word: String) -> [String] {  
    let chars = Array(word.characters)  
    let completed = knownWords.autocomplete(chars)  
    return completed.map { chars in  
        word + 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 = buildStringTrie(contents)
autocompleteString(trieOfWords, word: "car")
```

```
/** 结果为：
    ["car", "cart"]
*/
```

眼下，我们的接口只允许添加数组。创建另一版 insert 函数，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 insert<Seq: CollectionType where Seq.Generator.Element == Element>
    (key: Seq -> Trie<Element>
```

本章还提供了一个示例项目 (可以在 [GitHub](#) 上找到)，项目会加载 /usr/share/dict/words 路径下所有的单词并通过上文的 buildStringTrie 将它们构造为一棵字典树。构造一个拥有 250,000 个单词的字典树需要耗费相当一段时间。不过我们可以利用这个单词列表已被排序的特点，来重写构建函数对其进行优化。而且，这个操作是高度可并行的；将单词表分为从 a 到 m 与从 n 到 z 两部分，并行构建，然后合并结果，也是可以的。

使 Trie 数据类型遵守 SequenceType 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 SequenceType 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。

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：

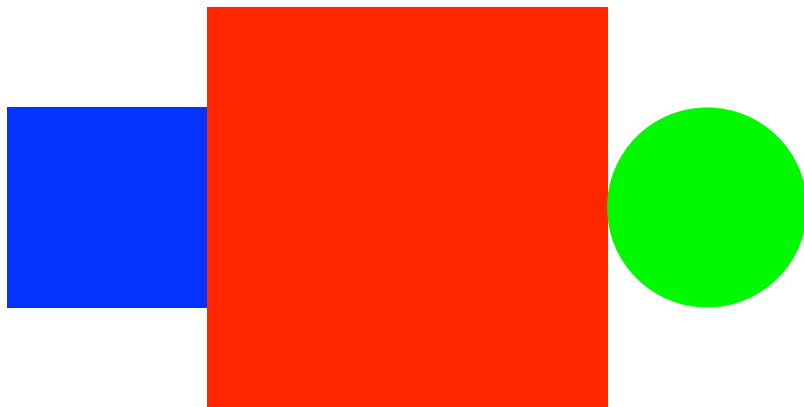


Figure 10.1: 简易图表

```
NSColor.blueColor().setFill()
CGContextFillRect(context, CGRectMake(0.0, 37.5, 75.0, 75.0))
NSColor.redColor().setFill()
CGContextFillRect(context, CGRectMake(75.0, 0.0, 150.0, 150.0))
NSColor.greenColor().setFill()
CGContextFillEllipseInRect(context, CGRectMake(225.0, 37.5, 75.0, 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？

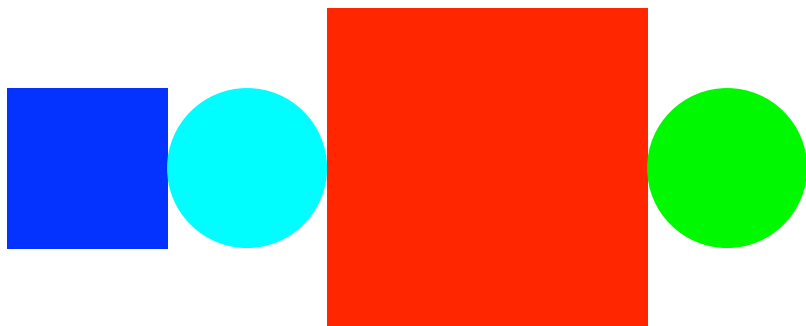


Figure 10.2: 添加额外的圆

我们可能需要添加一段绘制圆的代码，还要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).fill(. blueColor())
let redSquare = square(side: 2).fill(. redColor())
let greenCircle = circle(diameter: 1). fill (. greenColor())
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). fill (. cyanColor())
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，但这个思路有一个硬伤：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：不同于立刻执行绘制指令，我们会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；与之前区域的例子相反，这允许数据结构被检视，被修改，或将其转换成为不同的格式。

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：

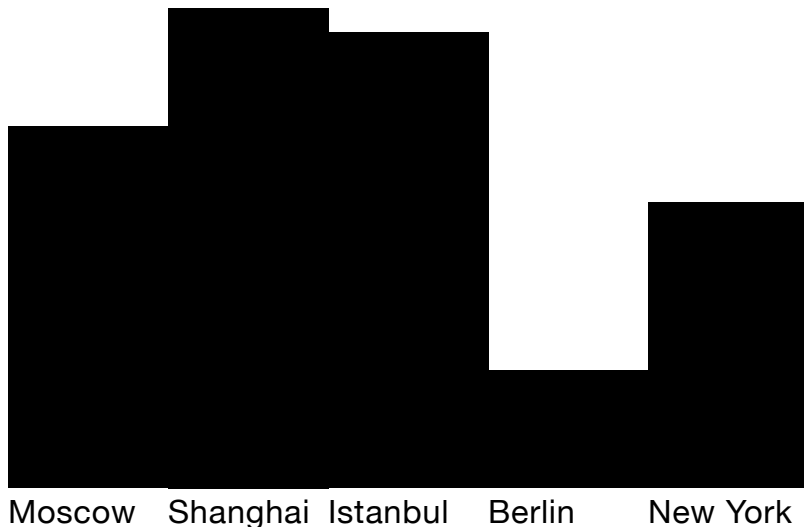


Figure 10.3: 柱形图

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` 函数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：

```
extension SequenceType where Generator.Element == CGFloat {  
  func normalize() -> [CGFloat] {  
    let maxVal = self.reduce(0) { max($0, $1) }  
    return self.map { $0 / maxVal }  
  }  
}
```

```
func barGraph(input: [(String, Double)]) -> Diagram {
```

```

let values: [CGFloat] = input.map { CGFloat($0.1) }
let nValues = values.normalize()
let bars = hcat(nValues.map { (x: CGFloat) -> Diagram in
    return rect(width: 1, height: 3 * x) . fill (. blackColor()).alignBottom()
})
let labels = hcat(input.map { x in
    return text(x.0, width: 1, height: 0.3).alignTop()
})
return bars --- labels
}

let cities = [
    ("Shanghai", 14.01),
    ("Istanbul", 13.3),
    ("Moscow", 10.56),
    ("New York", 8.33),
    ("Berlin", 3.43)
]

let example3 = barGraph(cities)

```

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

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)
}

```

图表也会利用一个枚举来定义。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。注意，之所以将其写作 `Prim`，是因为在本书编写的时候，编译器还不允许某个枚举成员的名称与另一个枚举的类型名称相同：

case Prim(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：

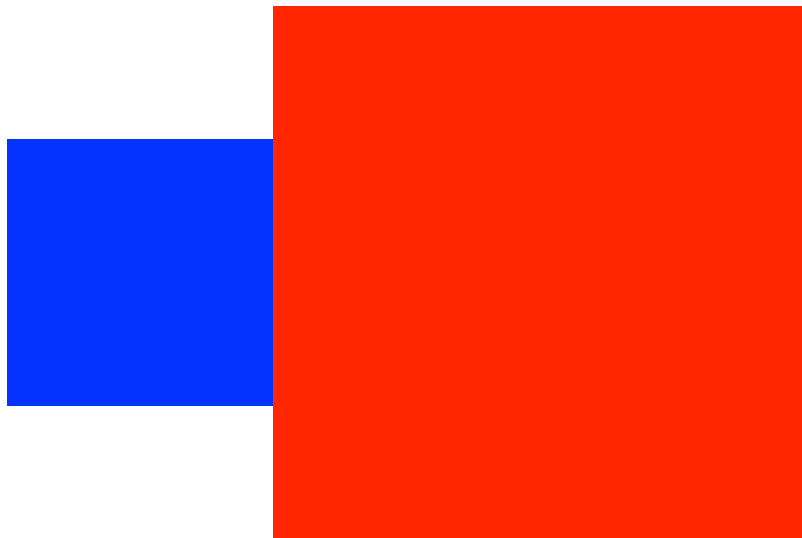


Figure 10.4: 垂直居中

不过通过为对齐方式添加枚举成员，我们可以控制图表中较小部分的对齐方式：

case Align(CGVector, Diagram)

比如，图 10.5 展示了一张顶部对齐的图表。绘制代码如下：

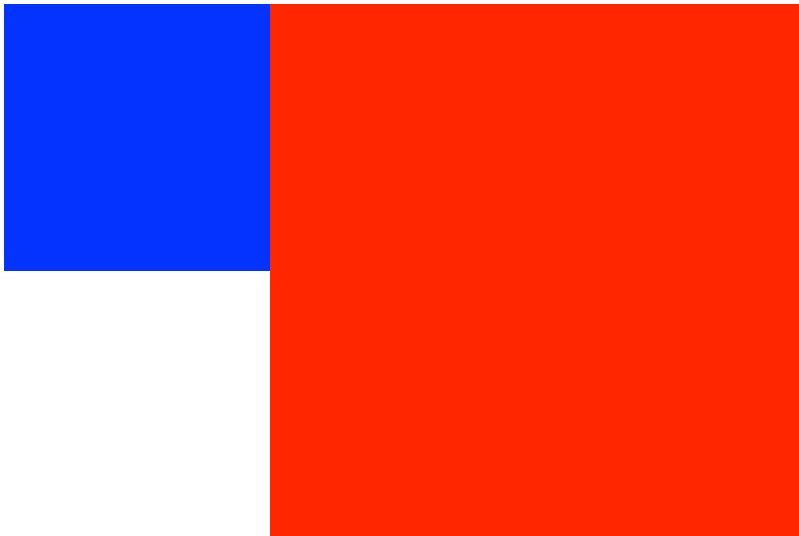


Figure 10.5: 垂直方向的对齐方式

```
Diagram.Align(CGVector(dx: 0.5, dy: 1), blueSquare) ||| redSquare
```

我们可以将 Diagram 定义为一个递归枚举。就像对枚举 Tree 所做得那样，需要将这个枚举标记为 **indirect**：

```
indirect enum Diagram {  
    case Prim(CGSize, Primitive)  
    case Beside(Diagram, Diagram)  
    case Below(Diagram, Diagram)  
    case Attributed(Attribute, Diagram)  
    case Align(CGVector, Diagram)  
}
```


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(NSColor)  
}
```

计算与绘制

计算 Diagram 数据类型的尺寸并不难。只是在 Beside 与 Below 条件下计算可能相对复杂些。在 Beside 情况下，宽度等于两个图表宽度之和，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 .Prim(let size, _):  
                return size  
            case .Attributed(_, let x):  
                return x.size  
            case .Beside(let l, let r):  
                let sizeL = l.size  
                let sizeR = r.size  
                return CGSizeMake(sizeL.width + sizeR.width,  
                                   max(sizeL.height, sizeR.height))  
            case .Below(let l, let r):  
                return CGSizeMake(max(l.size.width, r.size.width),  
                                   l.size.height + r.size.height)  
            case .Align(_, let r):  
                return r.size  
        }  
    }  
}
```

在我们开始绘制之前，还需要再定义一个函数。fit 函数作用于一个给定的尺寸值（比如某个图表的尺寸），参数包括一个对齐用的矢量（也即 Diagram 的 Align 成员的关联值），和一个希望将该尺寸值适配在内的矩形。这里的尺寸值是图表中其它的元素相对而言的。函数会等地地放大给定尺寸，同时保持其长宽比：

```

extension CGSize {
    func fit (vector: CGVector, _ rect: CGRect) -> CGRect {
        let scaleSize = rect.size / self
        let scale = min(scaleSize.width, scaleSize.height)
        let size = scale * self
        let space = vector.size * (size - rect.size)
        return CGRect(origin: rect.origin - space.point, size: size)
    }
}

```

为了让 fit 函数中编写的计算过程更加直观，我们为 CGSize，CGPoint，和 CGVector 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: 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 CGVector {
    var point: CGPoint { return CGPoint(x: dx, y: dy) }
    var size: CGSize { return CGSize(width: dx, height: dy) }
}

```

```
}
```

来试试 `fit` 函数吧。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：

```
CGSize(width: 1, height: 1).fit (
    CGVector(dx: 0.5, dy: 0.5), CGRect(x: 0, y: 0, width: 200, height: 100))
```

```
/** 结果为：
    (50.0, 0.0, 100.0, 100.0)
*/
```

如果要与矩形左对齐，可以这样写：

```
CGSize(width: 1, height: 1).fit (
    CGVector(dx: 0, dy: 0.5), CGRect(x: 0, y: 0, width: 200, height: 100))
```

```
/** 结果为：
    (0.0, 0.0, 100.0, 100.0)
*/
```

既然我们已经可以描述图表并计算出它们的尺寸，也就做好了绘制出它们的准备。使用模式匹配可以很容易地得知要画什么。因为进行绘制的上下文总是相同的，由此可以为 `CGContextRef` 定义一个扩展。`draw` 函数需要两个参数：绘制的边界，与实际的图表。在给定边界后，图表会试着利用之前定义的 `fit` 函数将自己适配在边界内。举个例子，当绘制一个椭圆的时候，我们使其居中并与边界相切：

```
extension CGContextRef {
    func draw(bounds: CGRect, _ diagram: Diagram) {
        switch diagram {
        case .Prim(let size, .Ellipse):
            let frame = size.fit(CGVector(dx: 0.5, dy: 0.5), bounds)
            CGContextFillEllipseInRect(self, frame)
```

矩形的处理方式也差不多，只不过需要调用不同的 Core Graphics 函数。你可能会发现计算矩形 `frame` 的方式与椭圆是相同的。虽然可以将这个步骤提取出来，再嵌套一层 `switch` 语句，不过我们觉得下面的代码在图书形式下会更易读：

```
case .Prim(let size, .Rectangle):
```

```
let frame = size.fit(CGVector(dx: 0.5, dy: 0.5), bounds)
CGContextFillRect(self, frame)
```

在这个库的当前版本中，所有的文字都是固定大小的系统字体。如果想要对其进行配置的话，不论是添加一个属性，还是修改 Text 图元，都是可行的选择。不过在当前版本下，绘制文字的方式如下：

```
case .Prim(let size, .Text(let text)):
    let frame = size.fit(CGVector(dx: 0.5, dy: 0.5), bounds)
    let font = NSFont.systemFontOfSize(12)
    let attributes = [NSFontAttributeName: font]
    let attributedText = NSAttributedString(string: text, attributes: attributes)
    attributedText.drawInRect(frame)
```

我们唯一支持的属性是填充色。为额外的属性提供支持很容易，为求简短，在此不作考虑。要绘制一个带有 FillColor 属性的图表，需要保存当前的图形状态，设置填充色，绘制出图表，然后恢复图形状态：

```
case .Attributed(.FillColor(let color), let d):
    CGContextSaveGState(self)
    color.set()
    draw(bounds, d)
    CGContextRestoreGState(self)
```

绘制两个相邻的图表，需要先计算出它们各自的 frame。我们为 CGRect 增加了一个 split 函数，用于根据某个比值 (在这里，我们选择用左侧图表的相对尺寸) 来分割一个 CGRect。然后根据两个图表的 frame 对它们进行绘制：

```
case .Beside(let left, let right):
    let (lFrame, rFrame) = bounds.split(
        left.size.width/diagram.size.width, edge: .MinXEdge)
    draw(lFrame, left)
    draw(rFrame, right)
```

split 函数定义如下：

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

```

        return divide(length * ratio, fromEdge: edge)
    }
}

extension CGRectEdge {
    var isHorizontal: Bool {
        return self == .MaxXEdge || self == .MinXEdge;
    }
}

```

Below case 分支也一样，只不过不再是水平地分割 CGRect，而是垂直方向。这些代码是为了运行在 Mac 上而编写的，也所以，其绘制顺序是先 bottom 后 top (不同于 UIKit, Cocoa 坐标系的原点在左下方)：

```

case .Below(let top, let bottom):
    let (lFrame, rFrame) = bounds.split(
        bottom.size.height/diagram.size.height, edge: .MinYEdge)
    draw(lFrame, bottom)
    draw(rFrame, top)

```

最后一个 case 分支是对齐图表。在这里，我们可以复用之前定义的 fit 函数来计算新的边界来适配图表：

```

case .Align(let vec, let diagram):
    let frame = diagram.size.fit(vec, bounds)
    draw(frame, diagram)
}
}

```

现在，我们已经定义出了这个库的核心部分。所有其它的元素都可以基于以上这些图元来构建。

创建视图与 PDF

我们可以创建一个 NSView 的子类用于实现绘制，在使用 playground，或是你想要在 Mac 应用中绘制这些图表时，这个子类会很实用：

```

class Draw: NSView {

```

```

let diagram: Diagram

init (frame frameRect: NSRect, diagram: Diagram) {
    self.diagram = diagram
    super.init(frame:frameRect)
}

required init(coder: NSCoder) {
    fatalError("NSCoding not supported")
}

override func drawRect(dirtyRect: NSRect) {
    guard let context = NSGraphicsContext.currentContext() else { return }
    context.cgContext.draw(self.bounds, diagram)
}
}

```

既然我们已经有了一个 `NSView`，那为这些图表输出一份 PDF 也会很容易。计算尺寸之后，只需要使用 `NSView` 的 `dataWithPDFInsideRect` 方法，就可以得到 PDF 的数据。作为一个例子，上述代码很好的展示了如何将已有的面向对象代码封装在一个函数式层中：

```

extension Diagram {
    func pdf(width: CGFloat) -> NSData {
        let height = width * (size.height / size.width)
        let v = Draw(frame: NSMakeRect(0, 0, width, height), diagram: self)
        return v.dataWithPDFInsideRect(v.bounds)
    }
}

```

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 width: CGFloat, height: CGFloat) -> Diagram {
    return .Prim(CGSizeMake(width, height), .Rectangle)
}

```

```

}

func circle(diameter diameter: CGFloat) -> Diagram {
    return .Prim(CGSizeMake(diameter, diameter), .Ellipse)
}

func text(theText: String, width: CGFloat, height: CGFloat) -> Diagram {
    return .Prim(CGSizeMake(width, height), .Text(theText))
}

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

```

事实证明，根据这种思路定义的运算符，会使水平或垂直地组合图表变得异常方便，且让代码更加易读。它们只是对 Beside 和 Below 的封装：

```

infix operator ||| { associativity left }
func ||| (l: Diagram, r: Diagram) -> Diagram {
    return Diagram.Beside(l, r)
}

infix operator --- { associativity left }
func --- (l: Diagram, r: Diagram) -> Diagram {
    return Diagram.Below(l, r)
}

```

我们还可以拓展 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 (top-level) 函数。这是一个风格问题：两者在功能上并没有太大区别：

```

extension Diagram {
    func fill (color: NSColor) -> Diagram {
        return .Attributed(.FillColor(color), self)
    }

    func alignTop() -> Diagram {
        return .Align(CGVector(dx: 0.5, dy: 1), self)
    }
}

```

```
func alignBottom() -> Diagram {
    return .Align(CGVector(dx: 0.5, dy: 0), self)
}
}
```

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。只需要使用数组的 `reduce` 函数就可以实现：

```
let empty: Diagram = rect(width: 0, height: 0)

func hcat(diagrams: [Diagram]) -> Diagram {
    return diagrams.reduce(empty, combine: ||)
}
```

通过添加这些小巧的辅助函数，我们拥有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。

生成器和序列

11

在本章中，我们将关注点放在了生成器 (Generators) 和序列 (Sequences) 上。它们组成了 Swift 中 **for** 循环的基础体系，同时也是我们在接下来章节中展示的解析库的基础部分。

生成器

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，然而总有些问题并不适合用数组来解决。举个例子，在总数接近无穷的时候，你可能不想对数组对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**生成器**。

为了说明问题，我们会首先利用与数组运算相似的例子，使你觉得生成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 xs {  
    // 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，生成器就可能派上用场。

从概念上来说，一个生成器是每次根据请求生成数组新元素的“过程”。一个生成器可以是遵守以下协议的任何类型：

```
protocol GeneratorType {  
    typealias Element  
    func next() -> Element?  
}
```

这个协议需要一个由 GeneratorType 定义的**关联类型**：Element。还有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，下文的生成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 next 函数推断出来，我们不必显式地指定：

```
class CountdownGenerator: GeneratorType {  
    var element: Int
```

```

init<T>(array: [T]) {
    self.element = array.count - 1
}

func next() -> Int? {
    return self.element < 0 ? nil : element--
}
}

```

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 (index) 初始化 element。

我们可以使用 CountdownGenerator 来倒序地遍历数组：

```

let xs = ["A", "B", "C"]

let generator = CountdownGenerator(array: xs)
while let i = generator.next() {
    print("Element \(i) of the array is \(xs[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，可生成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，我们只需要更新生成器，而不必再修改这里的代码。

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

```

class PowerGenerator: GeneratorType {
    var power: NSDecimalNumber = 1
    let two: NSDecimalNumber = 2

    func next() -> NSDecimalNumber? {
        power = power.decimalNumberByMultiplyingBy(two)
        return power
    }
}

```

```
}
```

我们可以使用 `PowerGenerator` 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 `PowerGenerator` 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。`findPower` 函数带有一个 `NSDecimalNumber -> Bool` 类型的 `predicate` 参数并返回符合该条件的最小值：

```
extension PowerGenerator {  
    func findPower(predicate: NSDecimalNumber -> Bool) -> NSDecimalNumber {  
        while let x = next() {  
            if predicate(x) {  
                return x  
            }  
        }  
        return 0  
    }  
}
```

我们可以使用 `findPower` 函数来计算二的幂值中大于 1000 的最小值：

```
PowerGenerator().findPower { $0.integerValue > 1000 }
```

```
/** 结果为：  
    1024  
*/
```

迄今为止，我们看到的生成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生成器。比如，下面的生成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：

```
class FileLinesGenerator: GeneratorType {  
    typealias Element = String  
  
    var lines: [String] = []  
  
    init(filename: String) throws {  
        let contents: String = try String(contentsOfFile: filename)
```

```

    let newLine = NSString.newlineCharacterSet()
    lines = contents.componentsSeparatedByCharactersInSet(newLine)
}

func next() -> Element? {
    guard !lines.isEmpty else { return nil }
    let nextLine = lines.removeAtIndex(0)
    return nextLine
}
}

```

通过这种定义生成器的方式，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生成器协议之后，可以确保代码在操作被生成的数据时不必再去考虑这些问题。

基于为生成器定义的协议，我们也编写了一些适用于所有生成器的泛型函数。举个例子，之前的 `findPower` 函数的泛型版本如下：

```

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

```

`find` 函数现在适用于任意的生成器。最有趣的事情是它的类型签名，由于调用了 `next`，生成器可能会被这个查找函数修改，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 `Bool` 值的函数。在 `find` 的类型签名中，我们可以引用生成器的关联类型 `Element`。最后，注意我们查询一个符合条件的值是不可能失败的。为此，`find` 会返回一个可选值，在生成器被耗尽时返回 `nil`。

层级式的组合生成器也是可行的。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，我们构建了一个生成器转换器，它可以用参数中的 `limit` 值来限制只获取对应个数的生成器 `G` 所生成的结果：

```

class LimitGenerator<G: GeneratorType>: GeneratorType {
    var limit = 0
    var generator: G

    init(limit: Int, generator: G) {
        self.limit = limit
        self.generator = generator
    }

    func next() -> G.Element? {
        guard limit >= 0 else { return nil }
        limit--
        return generator.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，这样的生成器可能会很有用。

在编写生成器时，为每个生成器引入一个新的类有时是一件很繁琐的事情。Swift 提供了一个简单的 `AnyGenerator<Element>` 类，其中的元素类型是一个泛型。该类可以通过传入一个 `next` 函数来进行初始化：

```

class AnyGenerator<Element>: GeneratorType, SequenceType {
    init(next: () -> Element?)
    ...
}

```

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

使用 `AnyGenerator` 可以使我们的代码更为简短地定义生成器。比如，我们可以像下面的代码一样重写 `CountdownGenerator`：

```

extension Int {
    func countDown() -> AnyGenerator<Int> {
        var i = self
        return anyGenerator { i < 0 ? nil : i-- }
    }
}

```

我们甚至可以依据 AnyGenerator 来定义能够对生成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生成器，代码如下：

```
func +<G: GeneratorType, H: GeneratorType where G.Element == H.Element>
    (var first: G, var second: H) -> AnyGenerator<G.Element>
{
    return anyGenerator { first.next() ?? second.next() }
}
```

返回的生成器会先读取 first 生成器的所有元素；在该生成器被耗尽之后，则会从 second 生成器中生成元素。如果两个生成器都返回 nil，该合成生成器也会返回 nil。

序列

生成器为 Swift 另一个协议提供了基础类型，这个协议就是**序列**。生成器提供了一个“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生成器。协议 SequenceType 则为这些功能提供了一组合适的接口：

```
protocol SequenceType {
    typealias Generator: GeneratorType
    func generate() -> Generator
}
```

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

```
struct ReverseSequence<T>: SequenceType {
    var array: [T]

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

    func generate() -> CountdownGenerator {
        return CountdownGenerator(array: array)
    }
}
```



```
}
```

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

```
let reverseSequence = ReverseSequence(array: xs)
let reverseGenerator = reverseSequence.generate()
```

```
while let i = reverseGenerator.next() {
    print("Index \(i) is \(xs[i])")
}
```

```
/** 打印:
    Index 2 is C
    Index 1 is B
    Index 0 is A
*/
```

对比之前仅仅使用生成器的例子，**同一个序列**可以被第二次遍历 —— 为此我们只需要调用 `generate` 来生成一个新的生成器就可以了。通过在 `SequenceType` 的定义中封装生成器的创建过程，开发者在使用序列时不必再担心潜在的生成器创建问题。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生成器，你可以编写一个 `for-in` 循环。比如，我们也可以将以上的代码片段写成这样：

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

```
/** 打印:
    Index 2 is C
    Index 1 is B
    Index 0 is A
*/
```

实际上，Swift 做的只是使用 `generate` 方法生成了一个生成器，然后重复地调用其 `next` 函数直到返回 `nil`。

之前的 CountdownGenerator 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，在序列上也有标准的 map 和 filter 函数：

```
public protocol SequenceType {
    public func map<T>(<
        @noescape transform: (Self.Generator.Element) throws -> T)
        rethrows -> [T]

    public func filter (<
        @noescape includeElement: (Self.Generator.Element) throws -> Bool)
        rethrows -> [Self.Generator.Element]
}
```

想倒序生成数组中的**元素**，我们可以使用 map 来映射 ReverseSequence：

```
let reverseElements = ReverseSequence(array: xs).map { xs[$0] }
for x in reverseElements {
    print("Element is \(x)")
}

/** 打印:
    Element is C
    Element is B
    Element is A
*/
```

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 函数**不会**返回新的序列，而是遍历序列来生成一个数组。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)。用于生成序列的 map 和 filter 也是存在的。它们被定义为 LazySequence 类的拓展。LazySequence 是对标准序列的封装，可以通过序列的一个 **lazy** 属性获得：

```
extension SequenceType {
    public var lazy: LazySequence<Self> { get }
}
```

如果你需要映射或过滤一个序列，而该序列可能会产生无穷多个结果，或是包含很多你并不感兴趣的元素时，请务必使用 `LazySequence` 而不是 `Sequence`。否则可能会导致你的程序无法完结，或是比你想像的要写的多得多。

案例研究：遍历二叉树

为了讲解序列和生成器，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>)
}
```

在定义一个生成树中元素的生成器之前，我们需要先定义一个辅助函数。在 Swift 的标准库中，有一个 `GeneratorOfOne` 结构体可以用于将某个可选值封装为一个生成器：

```
struct GeneratorOfOne<Element>: GeneratorType, SequenceType {
    init(_ element: Element?)
    // ...
}
```

给定一个可选值元素，它会生成一个基于该元素的序列 (仅当该元素不为 `nil` 时)：

```
let three: [Int] = Array(GeneratorOfOne(3))
let empty: [Int] = Array(GeneratorOfOne(nil))
```

为了方便，我们会为 `GeneratorOfOne` 构建一个小巧的封装函数：

```
func one<T>(x: T?) -> AnyGenerator<T> {
    return anyGenerator(GeneratorOfOne(x))
}
```

我们可以使用函数 `one` 与之前拼接生成器的运算符 `+` 一起来生成二叉树元素的序列。比如，遍历工具 `inOrder` 依次访问了左子树、根节点和右子树：

```
extension BinarySearchTree {
    var inOrder: AnyGenerator<Element> {
```

```

switch self {
case .Leaf:
    return anyGenerator { return nil }
case .Node(let left, let x, let right):
    return left.inOrder + one(x) + right.inOrder
}
}
}

```

如果树中没有元素，我们会返回一个空生成器。如果树有一个节点，我们会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。

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，这个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生成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() -> AnyGenerator<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。

首先，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 AnyGenerator，我们可以定义这样的一个函数如下：

```
extension Array {
    func generateSmallerByOne() -> AnyGenerator<[Element]> {
        var i = 0
        return anyGenerator {
            guard i < self.count else { return nil }
            var result = self
            result.removeAtIndex(i)
            i++
            return result
        }
    }
}
```

generateSmallerByOne 函数持续地追踪变量 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。

现在，我们可以看到这个函数返回所有可能的数组，每一个数组都比原始数组少一个元素：

```
[1, 2, 3].generateSmallerByOne()
```

不幸地是，这个调用并没有生成期望的结果 —— 返回值被定义为一个 AnyGenerator<[Int]>，而我们想看到的是一个以数组为元素的数组。好消息是，Array 有个构造方法带有一个 SequenceType 类型的参数。利用这个构造方法，我们可以测试生成器如下：

```
Array([1, 2, 3].generateSmallerByOne())
```

```
/** 结果为：
    [[2, 3], [1, 3], [1, 2]]
*/
```

利用 (在第九章定义的) `decompose` 函数，我们可以重新为数组定义 `smaller` 函数。如果想为之前 `generateSmallerByOne` 函数的计算结果设计一个递归的伪代码定义，我们需要做到以下几点：

- 如果数组为空，返回 `nil`
- 如果数组可以被分割为 `head` 与 `tail` 两部分，我们可以遵守以下规则并递归地计算剩余子数组：
 - 后一部分是一个子数组
 - 如果我们为 `tail` 的所有子数组首位拼接 `head`，可以计算出原始数组的子数组

我们可以根据已经定义的函数将这些规则直接翻译为 Swift：

```
extension Array {
    func smaller1() -> AnyGenerator<Element> {
        guard let (head, tail) = self.decompose else { return one(nil) }
        return one(tail) + Array<Element>(tail.smaller1()).map { smallerTail in
            [head] + smallerTail
        }.generate()
    }
}
```

我们现在已经做好了测试新版函数的准备，并且验证可知，它与 `generateSmallerByOne` 的返回值是相同的：

```
Array([1, 2, 3].smaller1())
```

```
/** 结果为：
    [[2, 3], [1, 3], [1, 2]]
*/
```

还有最后一个值得做的改进：那就是另一种可以判断和减小 QuickCheck 所找到的反例的方式。不同于只是移除元素，我们也会希望尝试让元素自己进行缩小。想实现这个功能，我们需要多处理一种情况，即 `Element` 也遵守 `Smaller` 协议：

```

extension Array where Element: Smaller {
  func smaller() -> AnyGenerator<Element> {
    guard let (head, tail) = self.decompose else { return one(nil) }
    let gen1 = one(tail).generate()
    let gen2 = Array<Element>(tail.smaller()).map { xs in
      [head] + xs
    }.generate()
    let gen3 = Array<Element>(head.smaller()).map { x in
      [x] + tail
    }.generate()
    return gen1 + gen2 + gen3
  }
}

```

可以检查新 `smaller` 函数的结果：

```
Array([1, 2, 3].smaller())
```

*/** 结果为：*

```

    [[2, 3], [1, 3], [1, 2], [1, 2, 2], [1, 1, 3], [0, 2, 3]]
*/

```

除了生成子列表之外，新版的 `smaller` 函数也可以用于生成元素值更小的数组。

不止是 Map 与 Filter

在未来的章节里，我们会需要更多基于序列和生成器的运算。为了定义这些运算，我们需要定义一个与前文中 `AnyGenerator` 类似的结构体 `AnySequence`。本质上，它封装了一个用于返回一个序列中生成器的函数。它的定义大概如下：

```

struct AnySequence<Element>: SequenceType {
  init<G: GeneratorType where G.Element == Element>
    (_ makeUnderlyingGenerator: () -> G)

  func generate() -> AnyGenerator<Element>
}

```

我们已经利用运算符 + 为生成器定义了拼接运算。一开始定义序列的拼接方法，可能会像下文中的定义：

```
func <A>(l: AnySequence<A>, r: AnySequence<A>) -> AnySequence<A> {  
    return AnySequence(l.generate() + r.generate())  
}
```

这个定义调用了两个参数序列的 generate 方法，然后将经过拼接得到的生成器指定给序列。不幸地是，实际效果并不符合预期。来看看下面的例子：

```
let s = AnySequence([1, 2, 3]) + AnySequence([4, 5, 6])  
print("First pass: ")  
for x in s {  
    print(x)  
}  
print("Second pass:")  
for x in s {  
    print(x)  
}
```

我们构建了一个包含元素 [1, 2, 3, 4, 5, 6] 的序列，然后遍历两次，打印出我们即将得到的元素。可能会感到一点点讶异，这段代码生成了如下的输出：

第一次输出：123456

第二次输出：

第二个 for 循环没有输出任何东西 — 发生什么了？其实问题出在对于序列进行连接的定义上。我们将两个生成器进行了组装 l.generate() + r.generate()。这个新的生成器在上面例子的第一次循环中将会生成所有我们需要的元素。然而，在所合成的序列进行第二次循环时，并没有生成一个新的生成器，所使用的依然是之前那个耗尽状态的生成器。

不过还好，这个问题很容易被修复。我们需要确保拼接操作的结果可以生成新的生成器。要做到这点，我们要向 AnySequence 的构造方法中传入一个可以生成生成器的函数，而不是一个固定的生成器：

```
func <A>(l: AnySequence<A>, r: AnySequence<A>) -> AnySequence<A> {  
    return AnySequence { l.generate() + r.generate() }  
}
```


现在，我们可以多次迭代同一个序列。在编写我们自己的序列组合方法时，要确保每次调用 `generate()` 都会生成一个新的生成器以忽略之前的遍历操作，这点非常重要。

到目前为止，我们可以拼接两个序列。将一个序列展开为一组序列又该如何呢？在处理序列之前，让我们试着编写一个 `join` 操作：给定一个 `AnyGenerator<AnyGenerator<A>>`，生成一个 `AnyGenerator<A>`：

```
struct JoinedGenerator<Element>: GeneratorType {
    var generators: AnyGenerator<AnyGenerator<Element>>
    var current: AnyGenerator<Element>?

    init<G: GeneratorType where G.Element: GeneratorType,
        G.Element.Element == Element>(var _ g: G)
    {
        generators = g.map(anyGenerator)
        current = generators.next()
    }

    mutating func next() -> Element? {
        guard let c = current else { return nil }
        if let x = c.next() {
            return x
        } else {
            current = generators.next()
            return next()
        }
    }
}
```

`JoinedGenerator` 维护了两部分可变状态：一个可选的当前生成器 `current`，和剩余的生成器 `generators`。在请求生成下一个元素的时候，如果当前生成器存在，则调用该生成器的 `next` 函数。如果不存在，则更新当前生成器 `current`，并再次递归地调用 `next`。仅当所有生成器都被耗尽后，`next` 函数返回 `nil`。

接着，我们使用 `JoinedGenerator` 来连接一组序列：

```
extension SequenceType where Generator.Element: SequenceType {
    typealias NestedElement = Generator.Element.Generator.Element
}
```

```

func join () -> AnySequence<NestedElement> {
    return AnySequence { () -> JoinedGenerator<NestedElement> in
        var generator = self.generate()
        return JoinedGenerator(generator.map { $0.generate() })
    }
}

```

我们先为嵌套序列的元素定义一个类型别名 `NestedElement`。返回类型也由此修改为 `AnySequence<NestedElement>`。要创建这样一个实例，需要利用一个返回 `JoinedGenerator<NestedElement>` 类型值的函数来初始化 `AnySequence`。为了创建嵌套的生成器，我们调用 `self` 的 `generate()`，返回一个 `AnyGenerator<AnySequence<A>>` 类型的值。最后要做的，则是为所有的序列进行一次映射，并调用每个序列的 `generate()`，这恰好可以依靠调用 `map` 来完成。

最后，我们也可以将 `join` 与 `map` 结合起来编写下面的 `flatMap` 函数：

```

extension AnySequence {
    func flatMap<T, Seq: SequenceType where Seq.Generator.Element == T>
        (f: Element -> Seq) -> AnySequence<T>
    {
        return AnySequence<Seq>(self.map(f)).join()
    }
}

```

给定一个以 `Element` 为元素类型的序列，以及一个函数 `f`。`f` 以一个 `Element` 类型的值为参数，生成一个元素类型为 `T` 的序列。只要符合以上条件，我们就可以构建一个 `T` 元素的单层序列。实现的方式很简单，对当前序列做一次 `f` 的映射，构建一个 `AnySequence<AnySequence<T>>`，然后对其调用 `join`，就可以获得期望的 `AnySequence<T>`。

现在我们已经充分掌握了序列和它支持的运算，我们可以继续进行下一个案例研究了：编写一个组合算子解析库。

案例研究：解析 器组合算子

12

解析器 (Parser) 是非常有用的工具：它们接收一组符号 (通常是一组字符) 并将它们转换为某种结构。通常，解析器是由像 [Bison](#) 或 [YACC](#) 这样的外部工具生成的。不过在本章，不同于引入外部工具，我们将构建一个解析器库，它将被用于在后面的章节中生成我们自己的解析器。将这个任务交由函数式语言来完成再合适不过。

编写一个解析库有几种方案。在这里，我们会构建一个解析器组合算子 (Parser Combinators) 库。每个解析器组合算子都是一个高阶函数，它接受若干 (在本章中被定义为函数的) 解析器作为参数，并返回一个新的解析器作为结果。我们即将构建的库基本上是对一个 Haskell 工具库 (2009) 的移植，当然，也做了一些微调。

我们将从定义一些核心的组合算子开始。接着，会在此之上定义一些额外的便利函数。在结尾处，我们会展示一个例子，解析比如 $1+3*3$ 这样的算数表达式，并将结果计算出来。

核心部分

在这个库中，我们会重度使用[序列](#)与[数组切片](#)。

译者注：也因此，在本文中会出现一些未详加提及却加以引用的拓展方法来简化使用，建议您下载本书 [GitHub 仓库](#)中[本章的示例代码](#)对照阅读。

我们将解析器定义为一个函数，其参数是一个符号数组的切片，函数会处理这些符号中的一部分，并返回一个序列，其元素是一系列包含处理结果与剩余符号的多元组。为了让以后的编码更为简单，我们将这样的函数封装在一个结构体里 (否则，我们可能得一直把这个函数完整的类型签名写出来)。我们使解析器支持两种泛型：Token 和 Result：

```
struct Parser<Token, Result> {  
    let p: ArraySlice<Token> -> AnySequence<(Result, ArraySlice<Token>)>  
}
```

如果不是因为不支持泛型，我们会更愿意为解析器类型定义一个类型别名。所以这个案例里，我们只能使用结构体来作为载体。

让我们从最简单的解析器开始，先定义一个用于解析单个字母 "a" 的解析器。为此，我们编写了一个返回字符 "a" 解析器的函数：

```
func parseA() -> Parser<Character, Character>
```

要返回单个结果，我们可以使用函数 `one` 构建一个只包含单个元素的序列。该函数的定义与前一章中一个返回单个 `Generator` 的函数 `one` 类似：

```
func one<A>(x: A) -> AnySequence<A> {  
    return AnySequence(GeneratorOfOne(x))  
}
```

如果第一个字符不是 “a”，解析器将返回一个空序列 `none()` 来表示失败。完整的 `parseA` 函数如下：

```
func none<T>() -> AnySequence<T> {  
    return AnySequence(anyGenerator { nil })  
}  
  
func parseA() -> Parser<Character, Character> {  
    let a: Character = "a"  
    return Parser { x in  
        guard let (head, tail) = x.decompose where head == a else {  
            return none()  
        }  
        return one((a, tail))  
    }  
}
```

可以使用一个名为 `testParser` 的函数来测试解析器：

```
func testParser<A>(parser: Parser<Character, A>, _ input: String) -> String {  
    var result: [String] = []  
    for (x, s) in parser.p(input.slice) {  
        result += ["Success, found \(x), remainder: \(Array(s))"]  
    }  
    return result.isEmpty ? "Parsing failed." : result.joinWithSeparator("\n")  
}
```

这个函数会使用作为第一个参数传入的解析器，来解析第二个参数 `input` 对应的字符串。解析器将可能的结果生成为一个序列，由 `testParser` 函数获取并返回一个说明结果的字符串：

```
testParser(parseA(), "abcd")
```

```
/** 结果为：  
    Success, found a, remainder: ["b", "c", "d"]  
*/
```

在上文的例子中，如果我们对一个不包括“a”的字符串运行解析器，就会得到一个失败的提示：

```
testParser(parseA(), "test")
```

```
/** 结果为：  
    Parsing failed.  
*/
```

对函数 `parseA` 做一次抽象，使其返回一个支持任意字符的解析器并不难。我们可以向函数传入一个需要被解析的字符参数，仅当被解析字符串的第一个字符与参数字符相同时返回结果：

```
func parseCharacter(character: Character) -> Parser<Character, Character> {  
    return Parser { x in  
        guard let (head, tail) = x.decompose where head == character else {  
            return none()  
        }  
        return one((character, tail))  
    }  
}
```

可以对 `parseCharacter` 做如下调用：

```
testParser(parseCharacter("t"), "test")
```

```
/** 结果为：  
    Success, found t, remainder: ["e", "s", "t"]  
*/
```

我们可以再对这个方法做一次抽象，将符号类型定义为泛型。不同于检查符号是否相等，这次我们会传入一个类型为 `Token -> Bool` 的函数，当该函数在处理输入流中首个 token 的结果为 `true` 时，我们将解析的结果返回：

```
func satisfy<Token>(condition: Token -> Bool) -> Parser<Token, Token> {  
    return Parser { x in
```

```

guard let (head, tail) = x.decompose where condition(head) else {
    return none()
}
return one((head, tail))
}
}

```

现在我们可以定义一个与 `parseCharacter` 功能类似的函数 `token`，不同之处只在于该函数适用于任何遵守 `Equatable` 协议的数据类型：

```

func token<Token: Equatable>(t: Token) -> Parser<Token, Token> {
    return satisfy { $0 == t }
}

```

选择

解析单独的符号可能会稍显鸡肋，所以我们会添加一些可以合并两个解析器的函数。我们要介绍的第一个函数是选择运算符 `<|>`。该运算符可以同时使用左边与右边的运算对象进行解析。

选择运算符实现起来很简单：假定传入一个字符串，运算符运行左侧的解析器，提供一个可能结果的序列。然后运行右侧解析器，也提供一个可能结果的序列，再将两个序列拼接起来就可以了。要注意的是：这两个序列既可能同时为空，也可能同时包含大量的元素。只不过因为它们实际的运算时机是被延迟的，所以并不会有什么影响：

```

infix operator <|> { associativity right precedence 130 }
func <|> <Token, A>(l: Parser<Token, A>, r: Parser<Token, A>)
    -> Parser<Token, A>
{
    return Parser { l.p($0) + r.p($0) }
}

```

我们可以构建一个同时解析 `"a"` 和 `"b"` 的解析器来测试我们的新运算符：

```

let a: Character = "a"
let b: Character = "b"

testParser(token(a) <|> token(b), "bcd")

```

```
/** 结果为：  
    Success, found b, remainder: ["c", "d"]  
*/
```

顺序解析

在实现了对两个解析器的合并之后，我们想实现一个更大胆的方案，并在之后将其拓展为更为方便和强大的工具。首先，我们编写一个 `sequence` 函数：

```
func sequence<Token, A, B>(l: Parser<Token, A>, _ r: Parser<Token, B>)  
    -> Parser<Token, (A, B)>
```

该函数返回的解析器会先使用左解析器来解析类型为 `A` 的某个值。假设我们希望解析字符串 “xyz” 中一个 “x” 紧接着一个 “y” 的部分。左侧 (查找 “x” 的) 解析器会生成一个序列，这个序列中只包含一个 `(result, remainder)` 的多元组，就像这样：

```
[ ("x", "yz") ]
```

左解析器返回的结果多元组中，剩余部分为 “yz”，对剩余部分使用右解析器，会得到另一个包含单个多元组的序列：

```
[ ("y", "z") ]
```

接着，将 “x” 与 “y” 组合为一个新的多元组 (“x”, “y”)：

```
[ ((("x", "y"), "z")) ]
```

在左侧解析器返回序列中的每一个多元组都经历了这个过程之后，我们会得到一个嵌套序列：

```
[ [ ((("x", "y"), "z")) ] ]
```

最后，我们将这个结构展平为一个单层序列，其元素是类型为 `((A, B), ArraySlice<Token>)` 的多元组。完整的 `sequence` 函数代码如下：

```
func sequence<Token, A, B>(l: Parser<Token, A>, _ r: Parser<Token, B>)  
    -> Parser<Token, (A, B)>  
{  
    return Parser { input in
```



```

    let leftResults = l.p(input)
    return leftResults.flatMap {
        (a, leftRest) -> [[[A, B), ArraySlice<Token>]] in
        let rightResults = r.p(leftRest)
        return rightResults.map { b, rightRest in
            ((a, b), rightRest)
        }
    }
}
}
}
}

```

我们可以试着依次解析 "x" 与 "y" 来测试我们的解析器：

```

let x: Character = "x"
let y: Character = "y"

```

```

let p: Parser<Character, (Character, Character)> = sequence(token(x), token(y))
testParser(p, "xyz")

```

```

/** 结果为：
    Success, found ("x", "y"), remainder: ["z"]
*/

```

改进

为了合并多个解析器并使之依次生效，最初的思路正如我们在上文中编写的 `sequence` 函数。假设我们仍旧想解析上文的字符串 "xyz"，不过这次是要依次对 "x"、"y" 与 "z" 进行解析。先来试着参照之前的方式调用 `sequence` 函数合并三个解析器：

```

let z: Character = "z"

```

```

let p2 = sequence(sequence(token(x), token(y)), token(z))
testParser(p2, "xyz")

```

```

/** 结果为：
    Success, found (("x", "y"), "z"), remainder: []
*/

```

这个方法并不可行，返回的结果是一个多元组 `(("x", "y"), "z")`，而不是预期的被展平的 `("x", "y", "z")`。要解决这个问题，可以对函数 `sequence` 进行一点拓展，增加参数的个数，编写一个合并三个解析器的函数 `sequence3`：

```
func sequence3<Token, A, B, C>(p1: Parser<Token, A>, _ p2: Parser<Token, B>,
    _ p3: Parser<Token, C>) -> Parser<Token, (A, B, C)>
{
    typealias Result = ((A, B, C), ArraySlice<Token>)
    typealias Results = [Result]

    return Parser { input in
        let p1Results = p1.p(input)
        return p1Results.flatMap { a, p1Rest -> Results in
            let p2Results = p2.p(p1Rest)
            return p2Results.flatMap { b, p2Rest -> Results in
                let p3Results = p3.p(p2Rest)
                return p3Results.map { (c, p3Rest) -> Result in
                    ((a, b, c), p3Rest)
                }
            }
        }
    }
}
```

```
let p3 = sequence3(token(x), token(y), token(z))
testParser(p3, "xyz")
```

```
/** 结果为：
    Success, found ("x", "y", "z"), remainder: []
*/
```

虽然返回了预期的结果，可这种方式既不灵活，也不易被拓展。其实，还有另一种更为简便的方法来依次合并多个解析器。

首先，我们需要创建一个不会消耗 `token` 的解析器，并使其返回一个形如 `A -> B` 的函数。这个函数用于将其余的若干解析器结果转换为我们指定的形式。下面的小例子就是一个这样的解析器：

```
func integerParser<Token>() -> Parser<Token, Character -> Int> {
```

```

    return Parser { input in
        return one(({ x in Int(String(x))! }, input))
    }
}

```

这个解析器并没有消耗任何符号，它返回了一个将字符转换为整数的函数。传入一个非常简单的字符串 "3" 作为例子。对该传入值使用 `integerParser` 会返回以下序列：

```
[ (A -> B, "3") ]
```

再使用另一个 (用于解析符号的) 解析器来解析剩余的符号 "3" (由于 `integerParser` 并没有消耗任何符号，剩余的符号与原始的符号是相同的)：

```
[ ("3", "") ]
```

现在我们只需要创建一个函数来合并这两个解析器，并返回一个新解析器，就可以使 `integerParser` 返回的函数被应用在 (前文提到的) 符号解析器返回的字符 "3" 上。这个函数看起来和函数 `sequence` 非常相似 —— 函数在第一个解析器所返回的序列上调用了 `flatMap` 得到转换函数，第二个解析器会对剩余部分进行解析并得到另一个序列，最后在这个序列上使用转换函数进行映射就能得到最终结果。

最关键的不同点在于闭包内部并没有像 `sequence` 那样，将两个解析器的结果都返回，而是将第一个解析器生成的函数应用在了第二个解析器的解析结果上：

```

func combinator<Token, A, B>(l: Parser<Token, A -> B>, _ r: Parser<Token, A>)
    -> Parser<Token, B>
{
    typealias Result = (B, ArraySlice<Token>)
    typealias Results = [Result]
    return Parser { input in
        let leftResults = l.p(input)
        return leftResults.flatMap { f, leftRemainder -> Results in
            let rightResults = r.p(leftRemainder)
            return rightResults.map { x, rightRemainder -> Result in
                (f(x), rightRemainder)
            }
        }
    }
}

```

将上述过程结合在一起，解析结果会像这样：

```
let three: Character = "3"
```

```
testParser(combinator(integerParser(), token(three)), "3")
```

```
/** 结果为：  
    Success, found 3, remainder: []  
*/
```

到这里，我们已经做好了准备来建立一个真正优雅的解析器合并机制。

我们要做的第一件事是将 `integerParser` 重构为一个泛型函数，它会接收一个参数，并返回一个总是成功的解析器，这个解析器不会消耗符号，且会将我们向函数中传入的参数作为解析结果返回：

```
func pure<Token, A>(value: A) -> Parser<Token, A> {  
    return Parser { one((value, $0)) }  
}
```

有了这个函数，我们可以像这样重写前一个例子：

```
func toInteger(c: Character) -> Int {  
    return Int(String(c))!  
}  
testParser(combinator(pure(toInteger), token(three)), "3")
```

```
/** 结果为：  
    Success, found 3, remainder: []  
*/
```

利用这个机制来合并多个解析器的技巧得益于柯里化这一概念。从第一个解析器中返回一个柯里化函数，可以让我们基于这个柯里化函数的参数个数，多次地执行合并过程。就像这样：

```
func toInteger2(c1: Character) -> Character -> Int {  
    return { c2 in  
        let combined = String(c1) + String(c2)  
        return Int(combined)!  
    }  
}
```

```

}

testParser(combinator(combinator(pure(toInteger2), token(three)),
    token(three)), "33")

/** 结果为:
    Success, found 33, remainder: []
*/

```

鉴于在代码中调用太多的 `combinator` 会使可读性下降，我们为此定义了一个运算符：

```

infix operator <*> { associativity left precedence 150 }
func <*>(Token, A, B) (l: Parser<Token, A -> B>, r: Parser<Token, A>)
    -> Parser<Token, B>
{
    typealias Result = (B, ArraySlice<Token>)
    typealias Results = [Result]
    return Parser { input in
        let leftResults = l.p(input)
        return leftResults.flatMap { (f, leftRemainder) -> Results in
            let rightResults = r.p(leftRemainder)
            return rightResults.map { (x, y) -> Result in (f(x), y) }
        }
    }
}

```

现在我们将之前的示例改写如下：

```

testParser(pure(toInteger2) <*> token(three) <*> token(three), "33")

/** 结果为:
    Success, found 33, remainder: []
*/

```

需要注意的是，我们刚刚定义的运算符 `<*>` 是左向优先的。这意味着运算符会先对左侧的两个解析器进行运算，之后才会将运算所得的结果与右侧的解析器结合。换句话说，这个行为与我们之前嵌套地调用 `combinator` 函数是完全相同的。

还有另一个运用该运算符的例子，通过将多个字符合并为字符串，也可以来构造一个字符串的解析器：

```
let aOrB = token(a) <|> token(b)

func combine(a: Character)-> Character -> Character -> String {
    return { b in {c in String([a, b, c]) } }
}

let parser = pure(combine) <*> aOrB <*> aOrB <*> token(b)
testParser(parser, "abb")

/** 结果为：
    Success, found abb, remainder: []
*/
```

在第三章中我们曾经定义过一个对双参函数进行柯里化的函数 `curry`。我们可以通过定义多个版本的 `curry` 函数，来支持不同数量参数的函数。举个例子，可以定义一个支持三参函数的版本，使我们能够以另一种方式编写上例中的解析器：

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

let parser2 = pure(curry { String([ $0, $1, $2]) })
    <*> aOrB <*> aOrB <*> token(b)
testParser(parser2, "abb")

/** 结果为：
    Success, found abb, remainder: []
*/
```

便利组合算子

利用上文中的组合算子，我们已经可以解析很多有意思的语言。只不过，以上文种的方式编写解析器还是有些枯燥。幸运地是，我们可以定义一些额外的函数来简化编码。首先我们将定义一个返回解析器的函数，该解析器可以从一个 `NSCharacterSet` 中解析出一个字符。这可以用于创建一个解析十进制数字的解析器，如下例：

```
func characterFromSet(set: NSCharacterSet) -> Parser<Character, Character> {
    return satisfy(set.member)
}
```

```
let decimals = NSCharacterSet.decimalDigitCharacterSet()
let decimalDigit = characterFromSet(decimals)
```

想验证解析器 `decimalDigit` 的作用，我们可以用它来解析下例中被传入的字符串：

```
testParser(decimalDigit, "012")
```

```
/** 结果为：
    Success, found 0, remainder: ["1", "2"]
*/
```

接下来要编写的便利组合算子是一个 `zeroOrMore` 函数，它可以选择将一个解析器执行多次，或是不执行：

```
func zeroOrMore<Token, A>(p: Parser<Token, A>) -> Parser<Token, [A]> {
    return (pure(prepended) <*> p <*> zeroOrMore(p)) <|> pure([])
}
```

示例中的 `prepend` 函数可以将类型为 `A` 的值与一个形如 `[A]` 的数组合并为一个新数组：

```
func prepend<A>(l: A)->([A]) -> [A] {
    return {r in [l] + r}
}
```

然而，如果我们尝试使用函数 `zeroOrMore` 时，我们会陷入一个死循环中。这是因为我们在返回语句中递归地调用了 `zeroOrMore`。

不过还好，我们可以将 `zeroOrMore` 的递归调用延迟到真正需要的时候再做执行，并利用这种方式将死循环中断。要实现这个功能，需要先定义一个辅助函数 `lazy`。该函数返回一个仅在必要的时候被执行一次的解析器：

```
func lazy<Token, A>(f: () -> Parser<Token, A>) -> Parser<Token, A> {
    return Parser { f (). p($0) }
}
```

现在我们可以用这个函数封装对 `zeroOrMore` 的递归调用：

```
func zeroOrMore<Token, A>(p: Parser<Token, A>) -> Parser<Token, [A]> {  
    return pure(prepend) <*> p <*> lazy { zeroOrMore(p) } <|> pure([])  
}
```

我们来测试下组合算子 `zeroOrMore`，看看它是否会生成多个结果吧。就像即将在本章看到的那样，我们通常只使用解析器的第一个成功结果，而剩余的结果由于是被延迟解析的，所以它们将不会被实际使用：

```
testParser(zeroOrMore(decimalDigit), "12345")
```

*/** 结果为：*

```
Success, found ["1", "2", "3", "4", "5"], remainder: []  
Success, found ["1", "2", "3", "4"], remainder: ["5"]  
Success, found ["1", "2", "3"], remainder: ["4", "5"]  
Success, found ["1", "2"], remainder: ["3", "4", "5"]  
Success, found ["1"], remainder: ["2", "3", "4", "5"]  
Success, found [], remainder: ["1", "2", "3", "4", "5"]
```

**/*

另一个有用的组合算子是 `oneOrMore`，这是一个将某解析器执行至少一次的函数。可以借助 `zeroOrMore` 来定义该函数：

```
func oneOrMore<Token, A>(p: Parser<Token, A>) -> Parser<Token, [A]> {  
    return pure(prepend) <*> p <*> zeroOrMore(p)  
}
```

如果解析出至少一个数字，我们会得到一个 `[Character]` 类型的数组，其中每个元素都是一个数字字符。要将结果数组转换为一个整数，我们可以先将元素类型为 `Character` 的数组转换为一个字符串，然后据此构造一个 `Int` 类型的值。尽管 `Int` 的该构造方法被标记为可选值，可因为我们知道构造一定会成功，所以我们可以使用运算符 `!` 强制解值：

```
let number = pure { Int(String($0))! } <*> oneOrMore(decimalDigit)
```

```
testParser(number, "205")
```

*/** 结果为：*

```
Success, found 205, remainder: []
```



```

    Success, found 20, remainder: ["5"]
    Success, found 2, remainder: ["0", "5"]
*/

```

在我们已经编写的所有代码里，可以看到一个重复出现的模式：

```
pure(x) <*> y
```

实际上，由于对这种模式的使用太过普遍，以至于有必要为其定义一个额外的运算符。如果将着眼点放在类型上，我们可以看到这个模式与 `map` 函数非常相似——该模式接收一个类型为 `A -> B` 的函数与一个结果类型为 `A` 的解析器，并返回一个结果类型为 `B` 的解析器：

```

infix operator </> { precedence 170 }
func </> <Token, A, B>(l: A -> B, r: Parser<Token, A>) -> Parser<Token, B> {
    return pure(l) <*> r
}

```

现在我们已经定义了许多有用的函数，是时候开始将这些函数（所返回的解析器）合并为真正的解析器了。举个例子，如果我们希望创建一个解析器来计算两个整数的和，可以以下文的方式来编写这个解析器：

```

let plus: Character = "+"
func add(x: Int)-> Character -> Int -> Int {
    return { _ in { y in x + y } }
}
let parseAddition = add </> number <*> token(plus) <*> number

```

再一次，我们来验证解析器的功能：

```
testParser(parseAddition, "41+1")
```

```

/** 结果为：
    Success, found 42, remainder: []
*/

```

有时候会出现这么一种情况，我们希望在解析一些东西的同时将解析结果忽略掉。就像上文的加法符号一样，我们只需要知道它被成功解析，却并不关注解析器返回了什么样的结果。可以定义另一个与运算符 `<*>` 功能十分相似的运算符 `<*>`，只不过该运算符会将右侧解析器的解析结

果丢弃 (也所以在运算符定义时去掉了右侧的尖括号)。类似的, 我们也定义了运算符 `>` 来丢弃左侧解析器的解析结果:

```
infix operator <* { associativity left precedence 150 }
func <* > <Token, A, B>(p: Parser<Token, A>, q: Parser<Token, B>)
  -> Parser<Token, A>
{
  return { x in { _ in x } } </> p <*> q
}
```

```
infix operator *> { associativity left precedence 150 }
func *> <Token, A, B>(p: Parser<Token, A>, q: Parser<Token, B>)
  -> Parser<Token, B>
{
  return { _ in { y in y } } </> p <*> q
}
```

让我们再为乘法编写一个解析器。内容与 `parseAddition` 函数基本相同, 只不过这次使用了新运算符 `<*>`, 在解析过 `"*"` 后会将结果丢弃:

```
let multiply: Character = "*"
let parseMultiplication = curry(*) </> number <*> token(multiply) <*> number
testParser(parseMultiplication, "8*8")
```

```
/** 结果为:
    Success, found 64, remainder: []
*/
```

一个简单的计算器

接下来, 我们会拓展之前的示例, 来解析类似于 `10+4*3` 的算术表达式。在计算结果时, 最需要注意的地方在于, 乘法具有比加法更高的优先级。这是因为数学 (与程序) 中有一个被称为**运算符优先级**的规则。在我们的解析器中描述这个规则非常自然。在这里具有最高优先级的操作是解析作为运算单元的数字, 让我们从它开始:

```
typealias Calculator = Parser<Character, Int>
```

```
func operator0(character: Character,
```

```

        _ evaluate: (Int, Int) -> Int,
        _ operand: Calculator -> Calculator
    {
        return curry { evaluate($0, $1) } </> operand <* token(character) <*> operand
    }

func pAtom0() -> Calculator { return number }
func pMultiply0() -> Calculator { return operator0("*", *, pAtom0()) }
func pAdd0() -> Calculator { return operator0("+", +, pMultiply0()) }
func pExpression0() -> Calculator { return pAdd0() }

testParser(pExpression0(), "1+3*3")

/** 结果为:
    Parsing failed.
*/

```

解析为什么会失败呢？

首先被解析的是一个加法表达式。加法表达式被上文的 pAdd0() 函数定义为由一个乘法表达式，一个 "+" 与另一个乘法表达式依次组合而成。而实际上，3*3 是一个乘法表达式，但 1 却只是一个数字。要修复这个问题，我们可以改写 operator 函数，使其既能解析一个形如“运算对象 运算符 运算对象”的表达式，也能解析一个只包括单个运算对象的表达式：

```

func operator1(character: Character,
    _ evaluate: (Int, Int) -> Int,
    _ operand: Calculator -> Calculator
{
    let withOperator = curry { evaluate($0, $1) } </> operand
    <* token(character) <*> operand
    return withOperator <|> operand
}

```

现在，我们终于有了一个具有实际功能的版本：

```

func pAtom1() -> Calculator { return number }
func pMultiply1() -> Calculator { return operator1("*", *, pAtom1()) }
func pAdd1() -> Calculator { return operator1("+", +, pMultiply1()) }
func pExpression1() -> Calculator { return pAdd1() }

```

```
testParser(pExpression1(), "1+3*3")
```

```
/** 结果为:
```

```
    Success, found 10, remainder: []
```

```
    Success, found 4, remainder: ["*", "3"]
```

```
    Success, found 1, remainder: ["+", "3", "*", "3"]
```

```
*/
```

如果想添加更多的运算符，并将解析过程再做一次抽象，我们可以创建一个多元组的数组，在每个多元组中包含一个运算符字符和一个实现该运算的函数，然后使用 `reduce` 函数将它们合并到一个解析器里：

```
typealias Op = (Character, (Int, Int) -> Int)
```

```
let operatorTable: [Op] = [("*", *), ("/", /), ("+", +), ("-", -)]
```

```
func pExpression2() -> Calculator {
```

```
    return operatorTable.reduce(number) { (next: Calculator, op: Op) in  
        operator1(op.0, op.1, next)
```

```
    }
```

```
}
```

```
testParser(pExpression2(), "1+3*3")
```

```
/** 结果为:
```

```
    Success, found 10, remainder: []
```

```
    Success, found 4, remainder: ["*", "3"]
```

```
    Success, found 1, remainder: ["+", "3", "*", "3"]
```

```
*/
```

然而，我们的解析器随着运算符的不断增加时会明显地变慢。这是因为解析器需要不断地回溯：解析器在解析的时候，如果失败，就需要接着尝试替代方案。比如，在尝试去解析 `"1+3*3"` 时，会先去尝试 `"-"` 运算符（根据之前的 `reduce` 的顺序，减法表达式应该由加法表达式、运算符 `"-"` 以及另一个加法表达式依次组成）。式中的第一个加法表达式会被解析成功，但接下来却找不到字符 `"-"`，因此解析器会去尝试另一种可能性：即只有一个加法表达式作为运算对象的情况。如果按照这样的方式进行解析，会有大量不必要的步骤被执行。

编写一个类似于上文的解析器非常简单。不过，却并不高效。如果我们回退一步来看，之前利用解析器组合算子定义的语法，其实可以像下文一样（使用伪语法描述语言）写出来（下例中，

符号 | 表示**或**，即先对左侧进行运算。如左侧失败，运算右侧，如也失败，则整个表达式返回失败，否则，返回第一次成功的结果)：

```
expression = min
min = add "-" add | add
add = div "+" div | div
div = mul "/" mul | mul
mul = num "*" num | num
```

为了移除大量重复表达，我们可以将语法重构为下文的样式(下例中，符号?用于修饰左侧括号内的表达式可能出现一次)：

```
expression = min
min = add ("-" add)?
add = div ("+" div)?
div = mul ("/" mul)?
mul = num ("*" num)?
```

在我们定义新的运算符函数之前，我们先额外定义一版运算符 </>。新版的运算符同样会使右侧的解析器进行解析，但会将解析结果丢弃，从而将左侧的运算过程保留至下一次合并运算中。我们将新运算符定义为 </>：

```
infix operator </ { precedence 170 }
func </ <Token, A, B>(l: A, r: Parser<Token, B>) -> Parser<Token, A> {
    return pure(l) <* r
}
```

同样地，我们将定义一个函数 optionallyFollowed，解析可能伴随着额外部分的左侧运算对象：

```
func optionallyFollowed<A>(l: Parser<Character, A>,
    _ r: Parser<Character, A -> A>) -> Parser<Character, A>
{
    let apply: A -> (A -> A) -> A = { x in { f in f(x) } }
    return apply </> l <*> (r </> pure { $0 })
}
```

最后，我们可以定义我们的运算符函数 op 了。它会解析类型为 Calculator 的参数 operand，以及可能随之而来的一个运算符和另一个 operand 参数。注意，这里并不能直接执行 evaluate，而是要先使用 flip 函数将其翻转(交换参数的顺序)。对于一些运算符来说，这并不

是必须的 (比如 $a + b$ 与 $b + a$ 的运算结果是相同的), 但对于另外一些来说, 却是必须的 (比如在 b 不等于零时, $a - b$ 与 $b - a$ 的结果是不同的):

```
func op(character: Character, _ evaluate: (Int, Int) -> Int,
    _ operand: Calculator) -> Calculator
{
    let withOperator = curry(flip(evaluate)) </ token(character) <*> operand
    return optionallyFollowed(operand, withOperator)
}
```

flip 函数的定义如下:

```
func flip<A, B, C>(f: (B, A) -> C) -> (A, B) -> C {
    return { (x, y) in f(y, x) }
}
```

我们现在已经为再一次定义完整的解析器做完了铺垫, 不过这次的方式会更为高效:

```
func eof<A>() -> Parser<A, ()> {
    return Parser { stream in
        if (stream.isEmpty) {
            return one (((), stream))
        }
        return none()
    }
}

func pExpression() -> Calculator {
    return operatorTable.reduce(number) { next, inOp in
        op(inOp.0, inOp.1, next)
    }
}
```

```
testParser(pExpression() <*> eof(), "10-3*2")
```

/** 结果为:

Success, found 4, remainder: []

*/

我们在以上示例中构建的计算器依然有明显的缺陷，最致命的问题在于每个运算符只能被使用一次。我们会在下一章中解决这个问题，并使用我们的解析器库构建一个小型的表格应用。

案例研究：构建 一个表格应用

13

在本章的研究中，我们会着手为一个简易的表格应用构建一个解析器、一个求值器以及一个 GUI (图形化用户界面)。表格是由一些按照行与列进行排列的单元格组成的。单元格可以包含像是 $10*2$ 这样的公式。在对公式进行解析后，我们会构造一棵被称为**抽象语法树**的树形结构来描述公式，并将其作为结果返回。随后，我们会对这些公式语法树进行求值，计算出每一个单元格的结果，再将其展示在 GUI 的表格视图中。

在我们的表格中，会使用从 0 开始的数字作为行坐标，而列坐标则使用从 A 到 Z 的字母来命名。比如，表达式 C10 用于表示位于第 10 行第 C 列的单元格。而形如 A0:A10 的表达式会用来表示一个单元格的列表。在本章的案例中，即是从第 A 列的第一个单元格开始，一直到行坐标为 10 的单元格为止 (该单元格也被包含在内)。

在实现这个表格应用的过程中，会用到前一章中构建的解析器库。在之前，我们就已经使用该库实现过一个简单的计算器。而在这次的表格应用中，我们会拓展该库来解析一些更复杂的公式：除了简单的数学计算，它还需要支持对单元格与函数的解析，比如 `SUM(A0:A10)`。

最后，在具备了解析器与求值器的函数式内核之后，我们会引入标准的 Cocoa 框架，来向大家展示如何将纯粹的函数式内核与面向对象的用户界面结合起来。

示例代码

不同于其它章节的 Playground 示例，本章会引入一个示例项目。这是因为项目中会附带一个简易的 GUI，仅在 Playground 中处理的话，例子中所涉及的内容会相当繁琐。在接下来的介绍中，请务必打开“Spreadsheet”项目来查阅完整的示例。

解析器

我们将把解析阶段分为两个步骤：符号化与解析。符号化阶段 (也被称为词法或词法分析) 会将输入的字字符串转化为一个符号序列。在此期间，我们还会移除空白符、并对运算符与括号进行解析，如此一来，就不必再担心它们可能会出现在接下来的解析过程中了。

而接下来的解析阶段中，则会对符号生成器返回的符号序列进行操作，将其转化为一个抽象语法树：一个用于表示公式表达式的树形结构。

符号化

一般而言，我们可以使用苹果 Foundation 框架中的 `NSScanner` 类来生成所需要的符号列表。可若是希望能有一个好用的 Swift API 来实现功能，可能得先对这个类进行封装。此外，我们还需要关掉该类自动跳过空白符的功能，并自行编写处理空白符的逻辑。因此在本书中，我们选择了更简单且有具示范性的做法：不使用 `NSScanner`，而是利用我们构建的解析器库编写一个扫描器。

和之前以函数式解决问题的方式相同，我们会先定义一个数据类型 `Token`。这是一个包含五个成员的枚举：数字、运算符、单元格的行列坐标 (references, 比如 A10 或 C42)、标点符号 (punctuation, 在本例中，只有圆括号)，以及函数名 (比如 SUM 或 AVG)：

```
enum Token: Equatable {
    case Number(Int)
    case Operator(String)
    case Reference(String, Int)
    case Punctuation(String)
    case FunctionName(String)
}
```

接下来，我们会为每个成员各定义一个解析器，解析器被用于从输入字符串读取字符，并返回一个由对应符号与字符串剩余部分组成的多元组。举个例子，对于公式字符串 `10+2`，数字成员 (`Number`) 对应的解析器函数应当返回一个 (`Token.Number(10), "+2"`)。

不过在此之前，让我们先定义一组可以使代码更为简洁的辅助函数。这些函数本身可能并没有太大的意义，不过还请拭目以待，它们在稍后必有用武之地。

第一个辅助函数被命名为 `const`，实现并不复杂：

```
func const<A, B>(x: A) -> B -> A {
    return { _ in x }
}
```

向 `const` 传入一个类型为 `A` 的值，它将返回一个形如 `B -> A` 的常量函数。也就是说，无论你向返回的函数中传入 (类型为 `B` 的) 什么值，它总是会返回 `const` 函数第一次接收的那个 (类型为 `A` 的) 参数。

另一个有用的辅助函数是 `tokens`。该函数会根据你传入的数组 `input` 构建一个解析器并将其返回。返回的解析器将消耗 `input` 中的那些元素，并返回一个由传入数组与剩余元素组成的多元组：

```
func tokens<A: Equatable>(input: [A]) -> Parser<A, [A]> {  
    guard let (head, tail) = input.decompose else { return pure([]) }  
    return prepend </> token(head) <*> tokens(tail)  
}
```

如果以上内容对你来说宛若天书，建议你先阅读[解析器组合算子](#)章节，该章节讲解了解析器库中所有的基本构建单元，比如运算符 `</>` 与 `<*>`。

`tokens` 函数会被递归地应用在传入的数组上：先将数组拆分为 `head` 部分 (第一个元素) 与 `tail` 部分 (包含剩余元素的数组)。接着使用[上一章](#)中定义的 `token` 函数来解析元素 `head`，并为 `tail` 递归地调用 `tokens`，将这二者依序合并之后，便会得到最终的解析器。

在具备了这个函数后，我们可以很容易地创建一个函数 `string` 来构造一个用于解析特定字符串的解析器：

```
func string(string: String) -> Parser<Character, String> {  
    return const(string) </> tokens(Array(string.characters))  
}
```

最后要介绍的辅助函数是 `oneOf`，一个以相互独立的方式结合多个解析器的函数 (比如，我们希望对 `+`、`-`、`/` 与 `*` 中的某一个运算符进行解析)：

```
func fail<Token, Result>() -> Parser<Token, Result> {  
    return Parser { _ in none() }  
}  
  
func oneOf<Token, A>(parsers: [Parser<Token, A>]) -> Parser<Token, A> {  
    return parsers.reduce(fail(), combine: <|>)  
}
```

函数中用到的 `fail` 辅助函数会直接返回一个总是解析失败的解析器，无论传入的是什么。

所有需要的辅助函数都已经准备就绪，让我们从数字成员 `Number` 的解析器开始吧。为了实现它，我们可以定义一个解析自然数的解析器，对字符输入流中至少一个十进制的数字进行消费，然后藉由[reduce](#) 函数的功能，将这些数字合并为一个整数：

```
let pDigit = oneOf(Array(0...9).map { const($0) </> string("\\($0)") })
```

```
func toNaturalNumber(digits: [Int]) -> Int {  
    return digits.reduce(0) { $0 * 10 + $1 }  
}
```

```
let naturalNumber = toNaturalNumber </> oneOrMore(pDigit)
```

现在，我们就可以定义函数 `tNumber` 了，该函数所做的就是解析出一个自然数，并通过将解析结果封装在 `Number` 中来生成一个 `Token`：

```
let tNumber = { Token.Number($0) } </> naturalNumber
```

想测试这个数字号生成器，我们可以传入 `"42"` 来运行这个解析器：

```
parse(tNumber, "42")
```

*/** 结果为：*

```
    Optional(main.Token.Number(42))
```

**/*

接下来是对运算符的解析：我们需要解析出对应运算符的字符串并将它封装为 `Token` 的 `.Operator` 成员中去。通过使用 `string` 函数，我们会将数组内定义每个运算符都转换为一个解析器，然后利用 `oneOf` 函数将这些单个运算符的解析器结合起来：

```
let operatorParsers = ["*", "/", "+", "-", ";"].map { string($0) }  
let tOperator = { Token.Operator($0) } </> oneOf (operatorParsers)
```

对于行列坐标 `Reference`，我们需要解析出一个大写字母以及尾随其后的自然数。首先我们利用上一章中的辅助函数 `characterFromSet` 构建一个解析器 `capital`：

```
let capitalSet = NSCharacterSet.uppercaseLetterCharacterSet()  
let capital = characterFromSet(capitalSet)
```

现在我们可以将解析器 `capital` 与解析器 `naturalNumber` 拼接起来以解析行列坐标。由于是对两个解析器进行合并，这里还需要将构造行列坐标的函数柯里化：

```
let tReference = curry { Token.Reference(String($0), $1) }  
    </> capital <*> naturalNumber
```

标点符号的解析器也非常简单：将左右圆括号的字符封装在 Punctuation 中即可：

```
let punctuationParsers = ["(", ")"].map { string($0) }  
let tPunctuation = { Token.Punctuation($0) } </> oneOf(punctuationParsers)
```

最后，由至少一个大写字母组成的函数名 (如 SUM) 可以被视为一个字符串，再将其封装在 FunctionName 中：

```
let tName = { Token.FunctionName(String($0)) } </> oneOrMore(capital)
```

到这里，为公式表达式生成一串符号输入流所需要的函数就已经基本准备完毕了。不过考虑到我们还需要忽略掉表达式中的各种空白符，这里我们额外定义一个辅助函数 ignoreLeadingWhitespace，把所有符号间的空白符“吃掉”：

```
let whitespaceSet = NSCharacterSet.whitespaceAndNewlineCharacterSet()  
let whitespace = characterFromSet(whitespaceSet)
```

```
func ignoreLeadingWhitespace<A>(p: Parser<Character, A>)  
    -> Parser<Character, A>  
{  
    return zeroOrMore(whitespace) *> p  
}
```

至此，一个完整的表达式符号生成器就可以被定义出来了。利用 oneOf 函数将以上所有的解析器结合在一起，再在外层封装一个 ignoreLeadingWhitespace 函数来清除空白符，最后将这些内容传入一个 zeroOrMore，就可以得到一个符号的列表：

```
func tokenize() -> Parser<Character, [Token]> {  
    let tokenParsers = [tNumber, tOperator, tReference, tPunctuation, tName]  
    return zeroOrMore(ignoreLeadingWhitespace(oneOf(tokenParsers)))  
}
```

关于符号生成器的内容就这么多，现在可以试着解析一个表达式的例子：

```
parse(tokenize(), "1+2*3+SUM(A4:A6)")
```

*/** 结果为：*

```
Optional([main.Token.Number(1), main.Token.Operator("+"), main.Token.  
    Number(2), main.Token.Operator("*"), main.Token.Number(3), main.Token.
```

```

Operator("+"), main.Token.FunctionName("SUM"), main.Token.
Punctuation("("), main.Token.Reference("A", 4), main.Token.Operator(":"),
main.Token.Reference("A", 6), main.Token.Punctuation(")"))
*/

```

解析

接下来，我们要从符号生成器生成的符号列表中创建一个表达式。表达式可以是数字、单元格的行列坐标、带有一个运算符的双目运算表达式，也可以是对某个函数的调用。在下文的递归枚举中可以看到以上描述的所有情况，这便是该表格应用所支持表达式的抽象语法树了：

```

indirect enum Expression {
    case Number(Int)
    case Reference(String, Int)
    case BinaryExpression(String, Expression, Expression)
    case FunctionCall(String, Expression)
}

```

到目前为止，我们构造出来的解析器只能处理一些字符串 (更确切地说，是一个字符列表)。而实际上，之前定义解析器的方式是支持任意类型的符号的，自然也包括上一小节中定义的 `Token` 类型。

在最开始，我们会为接下来的解析器定义一个简洁的类型别名，用以声明该解析器会从一连串类型为 `Token` 的值中，解析出 `Expression` 类型的值作为结果：

```

typealias ExpressionParser = Parser<Token, Expression>

```

先从解析数字开始吧。在试图将一个符号解析为一个数字表达式时，可能会发生两件事：要么在符号为数字的情况下成功，要么在其它所有情况下失败。为了构造数字解析器，我们还需要定义一个额外的辅助函数 `optionalTransform`。借助该函数，我们可以将被解析的符号转换为一个表达式 `Expression`，或是在符号无法被转为对应表达式时返回一个 `nil`：

```

func optionalTransform<A, T>(f: T -> A?) -> Parser<T, A> {
    return { f($0)! } </> satisfy { f($0) != nil }
}

```

现在，我们就可以定义数字表达式的解析器了。要注意的是 `Number` 在这里被使用了两次：第一个实例其实是 `Token.Number`，而第二个实例才是一个 `Expression.Number`：

```
let pNumber: ExpressionParser = optionalTransform {
  guard case let .Number(number) = $0 else { return nil }
  return Expression.Number(number)
}
```

对单元格坐标的解析也是大同小异：

```
let pReference: ExpressionParser = optionalTransform {
  guard case let .Reference(column, row) = $0 else { return nil }
  return Expression.Reference(column, row)
}
```

再将这两个解析器合并在一起：

```
let pNumberOrReference = pNumber <|> pReference
```

现在，我们可以使用这个解析器分别对数字和行列坐标进行解析，来测试它是否按照预期工作：

```
parse(pNumberOrReference, parse(tokenize(), "42"))!
```

*/** 结果为：*

```
Optional(main.Expression.Number(42))
```

**/*

```
parse(pNumberOrReference, parse(tokenize(), "A5"))!
```

*/** 结果为：*

```
Optional(main.Expression.Reference("A", 5))
```

**/*

接下来，我们来看看类似于 SUM(...) 的函数调用。为了实现这个功能，我们先定义一个解析器，与上文中数字和行列坐标的解析器类似，该解析器会解析一个函数名符号，或者在符号不是 Token.FunctionName 时返回 nil：

```
let pFunctionName: Parser<Token, String> = optionalTransform {
  guard case let .FunctionName(name) = $0 else { return nil }
  return name
}
```

在本案例中，调用函数时传入的参数总是形如 A1:A3 的一个以单元格行列坐标为元素的列表。而列表解析器则需要解析由运算符符号：分割开来的两个行列坐标符号：

```
func makeList(l: Expression, _ r: Expression) -> Expression {  
    return Expression.BinaryExpression(":", l, r)  
}
```

```
let pList: ExpressionParser = curry(makeList)  
    </> pReference <* op(":",) <*> pReference
```

上文中的 op 是一个简单的辅助函数，它负责接收一个运算符字符串并返回一个解析器。该解析器能够解析对应的运算符符号，同时将传入的运算符字符串作为解析结果返回：

```
func op(opString: String) -> Parser<Token, String> {  
    return const(opString) </> token(Token.Operator(opString))  
}
```

要想将以上功能结合起来对某个函数的调用进行解析，我们还需要一个用来解析括号表达式的函数：

```
func parenthesized<A>(p: Parser<Token, A>) -> Parser<Token, A> {  
    return token(Token.Punctuation("(")) *> p <*> token(Token.Punctuation(" "))  
}
```

这个函数以任意解析器 p 作为参数，然后将左右圆括号的解析器分别合并在该解析器左右两侧。由于使用了组合算子运算符 * 与 < 对括号解析器的解析结果进行丢弃，parenthesized 与 p 的解析结果是完全相同的。

现在我们可以将以上各个要素合并在一起，来解析一个完整的函数调用：

```
func makeFunctionCall(name: String, _ arg: Expression) -> Expression {  
    return Expression.FunctionCall(name, arg)  
}
```

```
let pFunctionCall = curry(makeFunctionCall)  
    </> pFunctionName <*> parenthesized(pList)
```

做个测试吧：


```
parse(pFunctionCall, parse(tokenize(), "SUM(A1:A3)"))!
```

*/** 结果为:*

```
Optional(main.Expression.FunctionCall("SUM", main.Expression.  
BinaryExpression(":", main.Expression.Reference("A", 1), main.Expression.  
Reference("A", 3))))
```

**/*

在解析完整的公式时，我们需要从表达式所包含的最小组成单元开始解析，并选择合适的方式实现运算符的优先级。举个例子，我们希望在解析时，乘法拥有比加法更高的优先级。在这里，我们先为公式的基本元素定义一个解析器：

```
let pParenthesizedExpression = parenthesized(lazy { expression() })  
let pPrimitive = pNumberOrReference <|> pFunctionCall  
    <|> pParenthesizedExpression
```

公式的基本元素可以是一个数字、一个单元格的行列坐标，一个函数调用，也可以是另一个包含在括号中的表达式。对于后者而言，我们需要使用辅助函数 **lazy**，以确保 **expression** 函数仅在真正需要的时候被调用，否则我们会陷入一个死循环中。(这里的 **expression** 函数应当返回另一个完整表达式的解析器，我们会在稍后实现。)

我们已经可以解析出基本元素了，现在可以将以上内容放在一起来解析乘法表达式 (或是除法——在这里，我们可以认为两种运算具有相同的运算优先级，所以会被同等对待)。一个乘法表达式可能只是一个因数，也可能还包括尾随在因数之后的多个像 ***** 被乘数 或 **/** 被除数 这样配对的运算符与运算对象。这些成对的表达式可以被定义成如下的样式：

```
let pMultiplier = curry { ($0, $1) } </> (op("**") <|> op("/")) <*> pPrimitive
```

解析器 **pMultiplier** 的结果是一个由运算符字符串 (如 ******) 与其右侧表达式组成的多元组。鉴于此，完整的乘法解析器看起来会像这样：

```
let pProduct = curry(combineOperands) </> pPrimitive <*> zeroOrMore(pMultiplier)
```

这里的关键是 **combineOperands** 函数，它可以根据单个基本元素和可能存在的多个 **pMultiplier** 结果多元组，构建出一棵表达式树。在构建这棵语法树时，我们需要确保运算符的计算是符合左向优先的 (不用担心 *****，因为乘法运算满足交换律，但 **/** 则不然)。幸运地是，**reduce** 函数就是以左向优先的方式生效的，也就是说，一个类似于 1, 2, 3, 4 的序列会被 **reduce** 函数以 ((1, 2), 3), 4) 的方式处理。我们可以利用这个特性来将一个表达式里的乘法运算对象进行组合：

```

func combineOperands(first: Expression, _ rest: [(String, Expression)])
    -> Expression
{
    return rest.reduce(first) { result, pair in
        let (op, exp) = pair
        return Expression.BinaryExpression(op, result, exp)
    }
}

```

到这里，我们就可以对基本元素和由它们组成的乘法表达式进行解析了。最后一个没有解决的问题是一个或多个乘法表达式之间的加法运算。鉴于这与乘法表达式的解析模式如出一辙，我们在这里就不多啰嗦了：

```

let pSummand = curry { ($0, $1) } </> (op("-") </> op("+")) <*> pProduct
let pSum = curry(combineOperands) </> pProduct <*> zeroOrMore(pSummand)

```

完整的公式表达式现在可以用解析器 pSum 来解析。我们为此解析器起一个别名，也就是上文中已经引用过的 expression：

```

func expression() -> ExpressionParser {
    return pSum
}

```

最后，可以在函数 parseExpression 中对符号生成器和解析器进行合并。先将输入的字符串符号化，如果符号化成功，就解析该表达式：

```

func parseExpression(input: String) -> Expression? {
    return parse(tokenize(), input).flatMap { parse(expression(), $0) }
}

```

只需要将该函数应用在一个完整的公式上，就可以生成我们预期的表达式语法树：

```

parseExpression("1 + 2*3 - MIN(A5:A9)")

```

*/** 结果为：*

```

Optional(main.Expression.BinaryExpression("-", main.Expression.
BinaryExpression("+", main.Expression.Number(1), main.Expression.
BinaryExpression("*", main.Expression.Number(2), main.Expression.
Number(3))), main.Expression.FunctionCall("MIN", main.Expression.

```

```
BinaryExpression(":", main.Expression.Reference("A", 5), main.Expression.  
Reference("A", 9))))  
*/
```

求值器

我们得到了一个 (使用 `Expression` 类型值来表示的) 抽象语法树, 现在要做的是将这个表达式的结果计算出来。在这个小例子中, 我们会假设我们的表格是一维的 (即只有一列)。将示例拓展为一个二维表格并不会太难, 不过出于演示的目的, 只关注一个维度会让问题更简单一些。

为了让代码更易读, 我们会做另一个简化: 不对单元格中表达式的计算结果进行缓存。在一个“真正的”表格应用中, 定义运算的顺序是非常重要的。举个例子, 假设单元格 A2 需要用到 A1 的值, 通常会先计算 A1 并将结果缓存, 以便我们在计算 A2 的结果时能够使用该缓存的结果。添加这个特性并不难, 但在这里, 为了让整个讲解更为清晰, 这个功能的实现就留给读者作为练习了。

让我们从定义 `Result` 枚举开始吧。在计算一个表达式时, 存在三种可能的结果。如果遇到类似 `10+10` 这种简单的算术表达式, 结果会是一个由 `IntResult` 成员来表示的数字。如果是形如 `A0:A10` 的表达式, 则会得到一个计算结果的列表, 这可以用 `ListResult` 来表示。最后, 在计算时很有可能出现一个错误, 比如一个表达式无法被解析, 或是使用了一个不合法的表达式。在这种情况下, 我们会希望显示一个错误, 我们用 `EvaluationError` 来代表这种情况:

```
enum Result {  
    case IntResult(Int)  
    case StringResult(String)  
    case ListResult([Result])  
    case EvaluationError(String)  
}
```

在继续进行求值之前, 我们先来定义一个名为 `lift` 的辅助函数。它可以使平常进行整数计算时所使用的函数对 `Result` 枚举也生效。比如, 运算符 `+` 是一个将两个整数参数求和的函数。借助 `lift`, 我们可以将运算符 `+` 提升为一个接收两个 `Result` 型参数的函数, 如果两个参数都是 `IntResult`, 则将两个参数合并为一个新的 `IntResult`。如果某一个 `Result` 参数并不是一个 `IntResult`, 该函数则返回一个求值错误:

```
typealias IntegerOperator = (Int, Int) -> Int
```

```
func lift (f: IntegerOperator) -> ((Result, Result) -> Result) {
```

```

    return { l, r in
      guard case let (.IntResult(x), .IntResult(y)) = (l, r) else {
        return .EvaluationError("Couldn't evaluate \{l, r}")
      }
      return .IntResult(f(x, y))
    }
  }
}

```

虽然符号生成器支持运算符 `"+"`、`"/"`、`"*"` 与 `"-"`，然而，在表达式枚举的 `BinaryExpression` 成员中运算符是作为一个 `String` 类型的值被存储的。因此，我们需要一种方式将这些字符串映射为一个可以合并两个 `Int` 参数的函数。这里定义了一个字典 `integerOperators` 来解决这个问题：

```

let integerOperators: [String: IntegerOperator] = [
  "+": op(+),
  "/": op(/),
  "*": op(*),
  "-": op(-)
]

```

现在，我们可以编写一个名为 `evaluateIntegerOperator` 的函数，给定一个运算符字符串 `op` 以及两个表达式，它将返回一个可选的 `Result`。我们将函数 `evaluate` 作为额外的参数传入，用来对单个表达式进行求值：

```

func evaluateIntegerOperator(op: String, _ l: Expression, _ r: Expression,
  _ evaluate: Expression? -> Result) -> Result?
{
  return integerOperators[op].map {
    lift($0)(evaluate(l), evaluate(r))
  }
}

```

运算符字符串 `op` 被用于在 `integerOperators` 字典中进行查询，它返回一个类型为 `(Int, Int) -> Int` 的可选值函数。我们对该可选值函数进行映射，然后使用 `evaluate` 参数对左右两个参数 `l` 与 `r` 进行求值，得到两个计算结果。最后使用 `lift` 函数将两个结果结合为一个新的结果。这里要注意，如果运算符无法被识别，会返回一个 `nil`：

为了对一个列表运算符 (比如 `A1:A5`) 进行求值，还要定义函数 `evaluateListOperator`，与上文中 `evaluateIntegerOperator` 类似：

```

func evaluateListOperator(op: String, _ l: Expression, _ r: Expression,
    _ evaluate: Expression? -> Result) -> Result?
{
    switch (op, l, r) {
    case (":", .Reference("A", let row1), .Reference("A", let row2))
        where row1 <= row2:
        return Result.ListResult(Array(row1...row2).map {
            evaluate(Expression.Reference("A", $0))
        })
    default:
        return nil
    }
}

```

我们先检查运算符字符串是否为列表运算符 `:`，同时由于这里的示例仅支持单列的表格，我们需要确保 `l` 与 `r` 都是行列坐标，且 `row2` 大于或等于 `row1`。注意，在这里我们使用一个 `switch` 条件分支就可以同时检查所有满足条件的情况。在其它所有情况下，我们都只需要返回 `nil`。如果所有前提条件都满足的话，我们就可以映射各个单元格并进行求值了。

到这里，我们就可以对任意双目运算符进行求值了。首先，我们会尝试整数运算符是否能求值成功。如果返回了 `nil`，我们会尝试去对列表运算符进行求值。如果都不成功，我们则返回一个错误：

```

func evaluateBinary(op: String, _ l: Expression, _ r: Expression,
    _ evaluate: Expression? -> Result) -> Result
{
    return evaluateIntegerOperator(op, l, r, evaluate)
        ?? evaluateListOperator(op, l, r, evaluate)
        ?? .EvaluationError("Couldn't find operator \(op)")
}

```

接下来，我们将为列表添加两个函数，`SUM` 和 `MIN`，它们分别用于计算某个列表的总和以及最小值。给定一个函数名和一个类型为 `Result` 的参数（我们现在仅支持单参函数），如果参数是一个 `ListResult`，就利用 `switch` 语句对所有可能的函数名进行检测。如果匹配成功，就可以对参数列表的总和或者最小值进行求值了。因为这些列表的元素并不是 `Int`，而是 `Result`，所以还需要使用 `lift` 函数对运算进行升级：

```

func evaluateFunction(functionName: String, _ parameter: Result) -> Result {
    switch (functionName, parameter) {

```

```

    case ("SUM", .ListResult(let list )):
        return list .reduce(Result.IntResult(0), combine: lift(+))
    case ("MIN", .ListResult(let list )):
        return list .reduce(Result.IntResult(Int.max),
            combine: lift { min($0, $1) })
    default:
        return .EvaluationError("Couldn't evaluate function")
}
}

```

将这些函数集合到一起，就可以对单个表达式进行求值了。不过在真正求值时，我们还需要所有其它的表达式的信息 (因为表达式可能包含对另一个单元格的引用)。因此，函数 `evaluateExpression` 将接收一个元素类型为 `Expression?` 的数组，作为参数 `context`：数组中的每个元素对应了每一个单元格的表达式 (数组的第一个元素是 A0 的表达式，第二个是 A1 的，以此类推)。在解析失败的情况下，可选 `Expression` 会是一个 `nil`。`evaluateExpression` 所需做的只是对表达式进行匹配并调用对应的求值函数：

```

func evaluateExpression(context: [Expression?]) -> Expression? -> Result {
    return { (e: Expression?) in
        e.map { expression in
            let recurse = evaluateExpression(context)
            switch (expression) {
            case let .Number(x):
                return Result.IntResult(x)
            case let .Reference("A", idx):
                return recurse(context[idx])
            case let .BinaryExpression(s, l, r):
                return evaluateBinary(s, l, r, recurse)
            case let .FunctionCall(f, p):
                return evaluateFunction(f, recurse(p))
            default:
                return .EvaluationError("Couldn't evaluate expression")
            }
        } ?? .EvaluationError("Couldn't parse expression")
    }
}

```

最后，我们可以定义一个便利函数 `evaluateExpressions`，它接收一个由可选 `Expression` 组成的列表，通过 `evaluateExpression` 对列表进行映射，生成一组 `Result` 的列表：

```
func evaluateExpressions(expressions: [Expression?]) -> [Result] {
    return expressions.map(evaluateExpression(expressions))
}
```

如同本章简介和通篇中所说明的那样，当前的解析器和求值器还有很大的局限性。更好的错误信息，对单元格的依赖性分析，循环检测，大规模的优化，类似这样可以做的事情还有很多。不过值得注意的是，我们只用了大约 200 行代码，就已经为这个表格应用定义了完整的模型层，解析器与求值器。这正是函数式编程的强大之处：在考虑了类型与数据结构之后，许多的函数编写起来都非常简单。而代码只要被写出来，就总会有办法进行改进和优化。

另外，我们在表达式解析和求值的编写过程中复用了很多的标准库函数。组合算子 `lift` 让复用任意的双整数参数的函数变得格外简单。像 `SUM` 和 `MIN` 这样的函数可以使用 `reduce` 来计算。而且，在具备了前一章的解析库之后，我们很快就编写出了词法分析器与解析器。

GUI

现在，我们已经具备了公式的解析器和求值器，就可以围绕着它们来构建一个简单的 GUI (图形化用户界面) 了。由于基本上所有的 Cocoa 框架都是面向对象的，我们将不得不在接下来的工作中将面向对象编程与函数式编程进行结合。幸好，Swift 使这项工作变得非常简单。

我们会创建一个包含窗口的 XIB 文件，窗口中只包含一个单列的 `NSTableView`。`NSTableView` 由一个数据源来提供数据，并由代理来处理编辑操作。数据源与代理是两个不同的对象。数据源会存储公式以及它们的结果，而代理则告诉数据源当前情况下，正在被编辑的是哪一行。处于编辑状态的一行会展示公式而不是结果。

数据源

数据源是真正意义上结合函数式与面向对象编程的部分。该对象遵守

`NSTableViewDataSource` 协议，并为 `NSTableView` 提供数据。在我们的案例中，数据就是指单元格。我们会检查一个单元格是否正处于编辑状态，如果处于编辑状态的话，我们会显示该单元格的公式，否则显示计算结果。在这个类中，我们需要做的是保存像是公式字符串、计算得出的结果、以及正处于编辑状态的行的行标 (它可能为 `nil`) 这些信息：

```
class SpreadsheetDatasource: NSObject, NSTableViewDataSource, EditedRow {
    var formulas: [String]
    var results: [Result]
```

```

    var editedRow: Int? = nil
    // ...
}

```

在这个类的构造方法中，我们会将公式初始化为从一到十的数字字符串。并将结果初始化为上述公式对应的结果值：

```

override init() {
    let initialValues = Array(1.. $10$ )
    formulas = initialValues.map { "\($0)" }
    results = initialValues.map { Result.IntResult($0) }
}

```

在这里，我们需要实现 `NSTableViewDataSource` 协议中的两个方法：第一个方法返回表格的行数（只需要返回数组 `formulas` 的元素个数）。第二个方法返回每一个单元格的内容。在本案例中，我们还要关注每一行的状态。如果该行正处于编辑状态，我们会返回公式。否则，则返回运算结果：

```

func numberOfRowsInTableView(tableView: NSTableView) -> Int {
    return formulas.count
}

func tableView(aTableView: NSTableView,
    objectValueForTableColumn: NSTableColumn?, row: Int) -> AnyObject?
{
    return editedRow == row ? formulas[row] : results[row].description
}

```

当用户编辑完一个单元格时，我们会重新计算所有的公式：

```

func tableView(aTableView: NSTableView, setObjectValue: AnyObject?,
    forTableColumn: NSTableColumn?, row: Int)
{
    let string = setObjectValue as! String
    formulas[row] = string
    calculateExpressions()
    aTableView.reloadData()
}

```


为了计算表达式，我们会使用之前定义的函数 `parseExpression`，对每一条公式进行映射，将其符号化并解析。映射会返回一组元素为 `Expression?` 的数组（符号化和解析有可能会失败）。然后，使用函数 `evaluateExpressions` 来计算一组对应的结果：

```
func calculateExpressions() {  
    results = evaluateExpressions(formulas.map(parseExpression))  
}
```

代理

代理唯一的任务是告诉数据源，当前处于编辑状态的单元格是哪一个。由于我们希望两个类之间是松耦合的，所以定义了一个额外的协议 `EditedRow`。协议要求实现一个名为 `editedRow` 的属性，在某些情况下会包含正处于编辑状态的行标（你可能已经在 `SpreadsheetDataSource` 类的类型声明中注意到了这个协议）：

```
protocol EditedRow {  
    var editedRow: Int? { get set }  
}
```

现在，当某行处于编辑状态的时候，我们只需要为数据源的该属性赋值就可以了：

```
class SpreadsheetDelegate: NSObject, NSTableViewDelegate {  
    var editedRowDelegate: EditedRow?  
  
    func tableView(aTableView: NSTableView,  
        shouldEditTableColumn aTableColumn: NSTableColumn?,  
        row rowIndex: Int) -> Bool  
    {  
        editedRowDelegate?.editedRow = rowIndex  
        return true  
    }  
}
```

窗口控制器

最后一个部分，我们会在窗口控制器中将所有元素连接在一起。（如果你是一名 iOS 的开发者，窗口控制器 `NSWindowController` 与视图控制器 `UIViewController` 的作用如出一辙）。在窗口

控制器中，有三个属性被声明为 IBOutlet，它们分别是一个 NSTableView，一个数据源以及一个代理。所有这些对象都会由 nib 文件进行实例化：

```
class SheetWindowController: NSWindowController {  
  
    @IBOutlet var tableView: NSTableView! = nil  
    @IBOutlet var dataSource: SpreadsheetDatasource?  
    @IBOutlet var delegate: SpreadsheetDelegate?
```

当窗口被载入时，我们将代理与数据源链接起来，以确保代理能够通知数据源某一行的编辑状态。另外，我们会注册一个观察者以得知编辑在何时结束。在编辑结束的通知发送之后，我们会将数据源的 editedRow 属性置为 nil：

```
    override func windowDidLoad() {  
        delegate?.editedRowDelegate = dataSource  
        NotificationCenter.defaultCenter().addObserver(self,  
            selector: NSSelectorFromString("endEditing:"),  
            name: NSControlTextDidEndEditingNotification, object: nil)  
    }  
  
    func endEditing(note: NSNotification) {  
        if note.object as! NSObject === tableView {  
            dataSource?.editedRow = nil  
        }  
    }  
}
```

大功告成。现在我们拥有了一个单列表格应用的高度可用的原型。在[示例工程](#)中可以查阅本案例的完整示例。

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，我们已经遇到了三个被命名为 `map` 的函数，类型签名分别如下 (为求简洁，以下签名都做了柯里化处理)：

```
func map<T, U>(xs: [T]) -> (T -> U) -> [U]
func map<T, U>(optional: T?) -> (T -> U) -> U?
func map<T, U>(result: Result<T>) -> (T -> U) -> Result<U>
```

为什么这三个截然不同的函数会拥有相同的函数名呢？要回答这个问题，我们需要先探究这三个函数的相似之处。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` 函数，相似之处就变得明显了起来：

```
extension Optional {
    func map<U>(transform: Wrapped -> U) -> Optional<U>
}

extension Array {
    func map<U>(transform: Element -> U) -> Array<U>
}
```

`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` 函数都需要两个参数：一个即将被映射的数据结构，和一个类型为 `T -> 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` 函数会使用 `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 —— 有时候也被称作**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` 函数：

```
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 函数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？

```

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
}

extension Region {
    func map<U>(transform: 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 函数，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，还定义了以下两种运算：

```
func pure<Token, A>(value: A) -> Parser<Token, A>
```

```
func <*><Token, A, B>(l: Parser<Token, A -> B>, r: Parser<Token, A>)  
    -> Parser<Token, B>
```

函数 `pure` 阐明了如何将任意值转化为一个返回该值的解析器。同时，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。对于任意类型构造体，如果我们可以为它定义恰当的 `pure` 与 `<*>` 运算，我们就可以将其称之为一个**适用函数** (Applicative Functor)。或者再严谨一些，对任意一个函数 `F`，如果能支持以下运算，该函数就是一个适用函数：

```
func pure<A>(value: A) -> F<A>
```

```
func <*><A, B>(f: F<A -> B>, x: F<A>) -> F<B>
```

实际上，适用函数一直隐藏在本书的各个角落中。比如，上一节定义的 `Region` 也是一个适用函数：

```
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，这里以适用函数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 = { x in { y in x && y } }
    return pure(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。

适用函数并不仅限于区域和解析器。Swift 内建的可选类型就是适用函数的另一个例子。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, 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 函数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 函数，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 函数的。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>) -> (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 ErrorType 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 NSError 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 self.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 += [(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)] } }
}

```

函数 `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` 函数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` 函数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？我们觉得，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并不容易：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，以及大量在其它书目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，可如果你的计算机可续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——且并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！

参考文献

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.

Swierstra, S Doaitse. 2009. "Combinator Parsing: A Short Tutorial." In *Language Engineering and Rigorous Software Development*, 252–300. Springer. doi:[10.1.1.184.7953](https://doi.org/10.1.1.184.7953).

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).