

# objc.io | objc 中国

# Swift

# 进阶

已对应 Swift 5.6

Chris Eidhof, Ole Begemann, Airspeed Velocity 著  
王巍, 崔轶, 范子君 译

英文版本 5.0 (2022 年 3 月), 中文版本 5.0 (2022 年 4 月)

© 2017 Kugler, Eggert und Eidhof GbR

版权所有

ObjC 中国

在中国地区独家翻译和销售授权

获取更多书籍或文章, 请访问 <https://objccn.io>

电子邮件: mail@objccn.io

<b>1</b>	<b>介绍</b>	<b>13</b>
	本书所面向的读者	14
	主题	15
	术语	18
	Swift 风格指南	23
	修订历史	24
	译者简介	27
<b>2</b>	<b>内建集合类型</b>	<b>28</b>
	数组	29
	数组和可变性	29
	数组索引	31
	数组变形	32
	数组切片	46
	字典	48
	可变性	49
	一些有用的字典方法	50
	键的 <code>Hashable</code> 要求	51
	Set	53
	集合代数	53
	在闭包中使用集合	54
	Range	55
	可数范围	56
	部分范围	58
	范围表达式	58
	RangeSet	59
	回顾	60
<b>3</b>	<b>可选值</b>	<b>62</b>
	哨岗值	63

通过枚举解决魔法数的问题	65
可选值概览	67
if let	67
while let	69
双重可选值	70
if var and while var	72
解包后可选值的作用域	73
可选链	77
nil 合并运算符	81
在字符串插值中使用可选值	83
可选值 map	85
可选值 flatMap	87
使用 compactMap 过滤 nil	89
可选值判等	90
可选值比较	93
强制解包的时机	93
改进强制解包的错误信息	95
在调试版本中进行断言	95
隐式解包可选值	97
隐式可选值行为	98
回顾	99
<b>4 函数</b>	<b>100</b>
综述	101
函数的灵活性	108
函数作为数据	111
函数作为代理	114
Cocoa 风格的代理	114
使用函数，而非代理	116
inout 参数和可变方法	119
嵌套函数和 inout	121
& 不意味 inout 的情况	122

下标	123
自定义下标操作	124
下标进阶	124
自动闭包	126
@escaping 标注	129
withoutActuallyEscaping	131
Result Builder	132
Block 和表达式	134
重载 Builder 方法	137
条件语句	138
循环	143
其他的构建方法	144
不支持的语句	145
回顾	145
<b>5 属性</b>	<b>147</b>
变更观察者	149
延迟存储属性	150
属性包装	152
使用方法	154
属性包装的来龙去脉	161
键路径	163
Key Paths Can Be Modeled with Functions	165
可写键路径	166
键路径层级	166
对比 Objective-C 的键路径	167
未来的方向	168
回顾	168
<b>6 结构体和类</b>	<b>169</b>
值类型和引用类型	170

可变性	174
可变方法	178
inout 参数	179
生命周期	180
循环引用	181
闭包和循环引用	185
在 unowned 引用和弱引用之间做选择	187
在结构体和类之间做抉择	188
具有值语义的类	189
具有引用语义的结构体	190
写时复制优化	192
写时复制的权衡	193
实现写时复制	194
willSet 对写时复制的破坏	199
回顾	201
<b>7 枚举</b>	<b>202</b>
概述	203
枚举是值类型	204
总和类型和乘积类型	206
模式匹配	208
在其他上下文中的模式匹配	212
使用枚举进行设计	214
Switch 语句的完备性	215
不可能产生非法的状态	217
使用枚举来实现 Model 状态	220
在枚举和结构体之间做选择	225
枚举和协议之间的相似之处	227
使用枚举实现递归数据结构	230
原始值 (Raw Value)	234
RawRepresentable 协议	234
手动实现 RawRepresentable	235

让结构体和类来实现 RawRepresentable	237
原始值的内部表示	238
列举枚举值	238
手动实现 Caselterable	240
固定和非固定枚举	241
提示和窍门	243
回顾	249
<b>8   字符串</b>	<b>250</b>
Unicode, 而非固定宽度	251
字位簇和标准等价	254
合并标记	254
颜文字	257
字符串和集合	260
双向索引, 而非随机访问	261
范围可替换, 而非可变	262
字符串索引	263
字符串解析	265
子字符串	268
StringProtocol	270
编码单元视图	272
共享索引	275
字符串和 Foundation	276
其他基于字符串的 Foundation API	277
字符范围	281
CharacterSet	283
Unicode 属性	283
String 和 Character 的内部结构	285
字符串字面量	286
字符串插值	288
定制字符串描述	291
LosslessStringConvertible	293

文本输出流	293
回顾	296
<b>9 泛型</b>	<b>298</b>
泛型类型	299
扩展泛型类型	301
泛型和 Any	303
基于泛型的设计	305
泛型的静态派发	308
泛型的工作方式	310
泛型特化	313
回顾	315
<b>10 协议</b>	<b>316</b>
协议目击者	319
条件化协议实现 (Conditional Conformance)	321
协议继承	322
使用协议进行设计	324
协议扩展	325
自定义协议扩展	326
协议组合	329
协议继承	330
关联类型	331
Self	332
例子：状态恢复	333
基于关联类型的条件化协议实现	334
回溯满足协议	336
存在体	336
对比存在体和泛型	338
存在体和关联类型	339
存在体无法遵守协议	341

不要过早使用存在体	342
不透明类型	343
信息隐藏	343
不透明类型的规则	347
类型消除器	349
使用解锁后的存在体手动实现类型消除	352
回顾	354
<b>11 集合类型协议</b>	<b>355</b>
序列	357
迭代器	358
基于函数的迭代器和序列	365
单次遍历序列	367
序列和迭代器之间的关系	368
集合类型	369
自定义的集合类型	370
数组字面量	376
关联类型	377
索引	378
索引失效	380
索引步进	381
自定义集合索引	382
子序列	385
切片	388
切片与原集合共享索引	389
专门的集合类型	391
BidirectionalCollection	391
RandomAccessCollection	393
MutableCollection	394
RangeReplaceableCollection	395
延迟序列	397
集合的延迟处理	400

回顾	401
<b>12 并发</b>	<b>402</b>
Async/Await	404
异步函数是如何执行的	407
和 Completion Handler 对接	409
结构化并发	411
任务	411
async let	413
任务组	416
Sendable 类型和函数	419
取消	422
非结构化并发	428
Actor	431
资源隔离	432
可重入	435
Actor 性能	437
Main Actor	438
推断执行上下文	440
回顾	441
<b>13 错误处理</b>	<b>442</b>
错误分类	443
Result 类型	445
抛出和捕获	447
具体类型错误和无类型错误	449
不可忽略的错误	452
错误转换	453
在 throws 和 Optionals 之间转换	453
在 throws 和 Result 之间转换	455
错误链	456

throws 链	456
Result 链	457
错误和回调	459
使用 defer 进行清理	461
Rethrows	463
将错误桥接到 Objective-C	465
回顾	467
<b>14 编码和解码</b>	<b>469</b>
一个最小的例子	471
自动遵循协议	471
Encoding	473
Decoding	474
自定义编码格式	475
编码过程	477
容器	478
值是如何对自己编码的	480
合成的代码	481
Coding Keys	481
encode(to:) 方法	482
init(from:) 初始化方法	483
枚举和原始表示	483
手动遵守协议	486
自定义 Coding Keys	486
自定义的 encode(to:) 和 init(from:) 实现	487
常见的编码任务	491
让其他人的代码满足 Codable	491
让类满足 Codable	495
解码多态集合	499
回顾	500

<b>15 互用性</b>	<b>502</b>
封装一个用 C 编写的程序库	503
设置包管理器	503
封装 CommonMark 程序库	507
包装 cmark_node 类型	508
一个更安全的接口	513
底层类型概览	520
指针	521
把闭包用作 C 的回调函数	525
泛化对 C 函数的转换	527
回顾	531
<b>16 写在最后</b>	<b>532</b>

# 介绍

1

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

和其他大多数编程语言一样，Swift 也是一门复杂的语言。但是它将这些复杂的细节隐藏得很好。你可以使用 Swift 迅速上手开发应用，而不必知晓泛型，重载或者写时复制在底层的工作机制。你可能永远都不需要去调用 C 语言的代码，或者实现自定义的集合类型。但是随着时间的推移，无论是想要提升你的代码性能，还是想让程序更加优雅清晰，亦或只是为了完成某项开发任务，你都有可能要逐渐接触到这些事情。

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

## 本书所面向的读者

本书面向的是有经验的程序员，当然你不需要是程序开发的专家，不过你应该已经是 Apple 平台的开发者。本书也面向那些从其他比如 Java 或者 C++ 这样的语言转行过来，并想把 Swift 相关知识技能提升到和原来已经熟知的其他语言同一水平线上的人。另外本书也适合那些已经开始学习 Swift，对这门语言基础有一定了解，并且渴望再上一个层次的新程序员们。

这本书不是一本介绍 Swift 基础的书籍，我们假定你已经熟悉这门语言的语法和结构。如果你需要完整地学习 Swift 的基础知识，最好的资源是官方的 Swift 相关书籍(可以在 [docs.swift.org](https://docs.swift.org) 上找到)。如果你很有把握，你可以尝试同时阅读我们的这本书和官方的 Swift 书籍。

这也不是一本教你如何为 macOS 或者 iOS 编程的书籍。不可否认，Swift 现在主要用于 Apple 的平台，我们会尽量包含一些实践中使用的例子，但是我们认为这本书可以对非 Apple 平台的程序员也有所帮助。本书中绝大部分的例子应该可以无缝运行在其他操作系统中。那些不能运行的代码，要么是由于它们是彻底与 Apple 平台绑定的(比如它们使用了 iOS 的框架或者依赖于 Objective-C 运行时)，要么可以通过很小的更改就能运行。从个人经验来说，对于在 Linux 上写服务端程序这件事来说 Swift 是一门优秀的语言，并且在过去几年中，生态和社区的发展使得这成为一个可行的方案。

# 主题

我们按照每章覆盖一个特定主题的方式来组织本书，其中有一些深入像是可选值和字符串这样基本概念的章节，也有对于像是 C 语言互用性方面的主题。不过纵观全书，有些主题也描绘了 Swift 给人的总体印象：

**Swift 跨越多个抽象层次。**Swift 是一门高级语言 - 它允许你用 map 或者 reduce 来写出十分类似于 Ruby 和 Python 的代码，你也可以很容易地创建自己的高阶函数。Swift 也让你有能力写出直接编译为原生的二进制可执行文件的高效代码，这使得性能上可以与 C 代码编写的程序相媲美。

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

不过，为了最大化利用这些特性，有一些知识是你需要掌握的。如果你能对结构体和类的区别有深刻理解，或者对动态和静态方法派发的不同了然于胸的话，你就能从中获益，我们将在之后更深入地介绍这些话题。另外，即使你需要降到一个更低的抽象层来直接操作指针的话，Swift 也支持你这样做。

**Swift 是一门多范式的语言。**对那些来自其他语言的开发者来说，Swift 总会有和他们所喜爱的语言非常相似的特性。你可以用 Swift 来编写面向对象的代码，也可以使用不可变值来写纯函数式的程序，你甚至还能使用指针运算来写命令式的 C 代码。

这是一把双刃剑。好的一面，在 Swift 中你将有很多可用工具，你也不会被限制在单一的代码写法里。但是这也让你身临险境，因为可能你实际上会毫不思索地把你熟悉的 Java 或者 C 或者 Objective-C 的模式移植到 Swift 里，而不去寻找更符合习惯的替代方式。Swift 的类型系统提供了很多旧时代编程语言中无法使用的能力，为常见的任务提供了(而且通常是被鼓励使用的)新的解决方案。

Erik Meijer 是一位著名的程序语言专家，他在 [2015 年 10 月发推说道](#)：

现在，相比 Haskell，Swift 可能是更好，更有价值，也更适合用来学习函数式编程的语言。

Swift 拥有泛型，协议，值类型以及闭包等特性，这些特性是对函数式风格的很好的介绍。我们甚至可以将运算符和函数结合起来使用。即便如此，当在结合那些源自函数式语言的模式时，在 Swift 社区中的多数人倾向于一种更加命令式的风格。Swift 对于值类型的新颖的可变性 (mutability) 模型，以及它的错误处理模型，这些都是这门语言在友好的命令式语法背后隐藏函数式概念的例子。

**Swift 十分灵活。**在 On Lisp 这本书的介绍中，Paul Graham 写到：

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

Swift 与 Lisp 是非常不同的，不过，我们仍能强烈感受到 Swift 也鼓励从下向上的编程方式。这让我们能轻而易举地编写一些通用可重用组件，然后你可以将它们组合起来实现更强大的特性，最后用它们来解决你的实际问题。Swift 非常适合用来构建这些组件，你可以使它们看起来就像是语言自身的一部分。一个很好的例子就是 Swift 的标准库，许多你能想到的基本组件 - 像是可选值和基本的运算符等 - 其实都不是直接在语言层中定义的，相反，它们是在标准库中被实现的。尾随闭包使你能够像用内置特性一样来扩展这门语言。

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

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

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

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

C# 的主要设计者之一的 Eric Lippert 在他关于创造 C# 的 10 件后悔的事情中总结了一些经验教训：

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

说这段话时，Eric 特别所指的是 C# 中的终止方法 (finalizer)，它和 C++ 中的析构函数 (destructor) 类似。但是不同于析构函数，终止方法的运行是不确定的，它受命于垃圾回收器，并且运行在垃圾回收器所在的线程上。更糟糕的是，很可能终止方法甚至完全不会被调用到。在 Swift 中，因为采用的是引用计数，deinit 方法的调用是可以确定和预测的（虽然一个对象到底会在哪个时间点被释放这件事，还是会由于编译器优化而有所不同）。

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

当你真正需要的时候，也有不少“不安全”的方式，比如 unsafeBitcast 函数，或者是 UnsafeMutablePointer 类型。但是强大能力所伴随着的是巨大的未定义行为的风险。比如下面的代码：

```
var someArray = [1,2,3]
let uhOh = someArray.withUnsafeBufferPointer { ptr in
```

```
    return ptr  
}  
// 稍后...  
print(uhOh[10])
```

这段代码可以编译，但是天知道它最后会做什么。ptr 变量只在闭包中有效，把它返回给调用者是非法的。但是没有什么能阻止你让这个变量成为野指针。然而你不能说没人阻止过你。

**Swift 是一门独断的语言。**关于“正确的”Swift 编码方法，作为本书作者，我们有着自己坚定的看法。你会在本书中看到很多这方面的内容，有时候我们会把这些看法作为事实来对待。但是，归根结底，这只是我们的看法，你完全可以反对我们的观点。不论你在读什么资料，最重要的事情是你应当亲自尝试，去检验它们的行为，并且去体会这些用法。

**Swift 在持续进化中。**Swift 1.0 是在 2014 年发布的。每年语法发生重大变化的时期已经离我们远去了，Swift 的每个版本也越来越“完整”。但是语言中的一些重要的部分还未完全定稿（比如并发、字符串的 API 和泛型系统），有的部分还在剧烈变化（比如反射和所有权）。

## 术语

“我用一个词，总是同我想要说的恰如其分，既不重，也不轻。”矮胖子相当傲慢地说。

— 爱丽丝镜中奇遇记，刘易斯·卡罗尔

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

在 Swift 中，我们需要对值，变量，引用以及常量加以区分。

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

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

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

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

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

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

类 (**class**) 和参与者 (**actor**) 是引用类型 (**reference type**)。你不能在一个变量里直接持有一个类的实例 (我们偶尔可能会把这个实例称作对象 (**object**)，这个术语经常被滥用，会让人困惑)。对于一个类的实例，变量会持有对它的引用，然后通过这个引用来访问它。

引用类型具有同一性 (**identity**)，也就是说，你可以使用 `==` 来检查两个变量是否确实引用了同一个对象。如果相应类型的 `=` 运算符被实现了的话，你也可以用 `=` 来判断两个对象是否相等。两个不同的对象按照定义也是可能相等的。

值类型不存在同一性的问题。比如你不能对某个变量判定它是否和另一个变量持有“相同”的数字 2。你只能检查它们都包含了 2 这个值。`==` 运算符实际做的是询问“这两个变量是不是持有同样的引用”。在程序语言的论文里，`=` 有时候被称为结构相等，而 `==` 则被称为指针相等或者引用相等。

Swift 中，class 和 actor 引用不是唯一的引用类型。Swift 中依然有指针，比如使用 `withUnsafeMutablePointer` 和类似方法所得到的就是指针。不过 class 和 actor 是使用起来最简单的引用类型，这与它们的引用特性被部分隐藏在语法糖之后是不无关系的。你不需要像在其他一些语言中那样显式地对指针做“解引用”。(我们会在稍后的互用性章节中详细提及其他种类的引用。)

一个引用变量也可以用 `let` 来声明，这样做会使引用变为常量。换句话说，这会使变量不能被改变为引用其他东西，不过很重要的是，这并不意味着这个变量所引用的对象本身不能被改变。所以，当用常量的方式来引用变量的时候要格外小心，只有指向关系被常量化了，而对象本身还是可变的。(如果前面这几句话看起来有些不明不白的话，不要担心，我们在结构体和类还会详细解释)。这一点造成的问题是，就算在一个声明变量的地方看到 `let`，你也不能一下子就知道声明的东西是不是完全不可变的。想要做出正确的判断，你必须先知道这个变量持有的是值类型还是引用类型。

复制值类型时，它通常执行深复制，也就是说，它包含的所有值会被递归地复制。这种复制可能是在赋值新变量时就发生的，也可能会延迟到变量内容发生变更的时候再发生。执行深复制的类型被称作具有值语义 (**value semantics**)。

这里我们会遇到另一件复杂的事情。如果我们的结构体中包含有引用类型，在将结构体赋值给一个新变量时所发生的复制行为中，这些引用类型的内容是不会被自动复制一份的，只有引用本身会被复制。这种复制的行为被称作浅复制 (**shallow copy**)。

举个例子，Foundation 框架中的 Data 结构体是对储存实际字节数据的 class 的封装。这意味着这些字节数据在 Data 传递时其实自身并不会被复制。不过，为了维持值语义，Data 的作者采取了额外的步骤，来保证当 Data 结构体发生变化的时候对其中的 class 对象进行深复制。它使用一种名为写时复制 (**copy-on-write**) 的技术来保证操作的高效，我们在结构体和类里详细介绍这种机制。现在我们需要重点知道的是，这种写时复制的特性并不是自动的。如果你想要让你自己的类型使用写时复制，你需要自己实现它。

在标准库中，像是数组这样的集合类型也都是对引用类型的封装，它们同样使用了写时复制的方式来在提供值语义的同时保持高效。不过，如果集合类型的元素是引用类型(比如一个含有对象的数组)的话，对象本身将不会被复制，只有对它的引用会被复制。也就是说，Swift 的数组只有当其中的元素满足值语义时，数组本身才具有值语义。

有些类是完全不可变的，也就是说，从被创建以后，它们就不提供任何方法来改变它们的内部状态。这意味着即使它们是类，它们依然具有值语义（因为它们就算被到处使用也从不会改变）。但是要注意的是，只有那些标记为 `final` 的类能够保证不被子类化，也不会被添加可变状态。

在 Swift 中，函数也是可以传递的值。你可以将一个函数赋值给一个变量，也可以创建一个包含函数的数组，或者调用变量所持有的函数。如果一个函数接受别的函数作为参数（比如 `map` 函数接受一个转换函数，并将其应用到数组中的每个元素上），或者一个函数的返回值是函数，那么这样的函数就叫做高阶函数 (**higher-order function**)。

函数不需要被声明在最高层级 — 你可以在一个函数内部声明另一个函数，也可以在一个 `do` 作用域或者其他作用域中声明函数。如果一个函数被定义在外层作用域中，但是被传递出这个作用域（比如这个函数被作为其他函数的返回值返回时），它将能够“捕获”局部变量。这些局部变量将存在于函数中，不会随着局部作用域的结束而消亡，函数也将持有它们的状态。这种行为的变量被称为“闭合变量”，我们把这样的函数叫做闭包 (**closure**)。

函数可以通过 `func` 关键字来定义，也可以通过 {} 这样的简短的闭包表达式 (**closure expression**) 来定义，有时简称为闭包，不过不要让这种叫法蒙蔽了你的双眼。实际上使用 `func` 关键字定义的函数，如果它包含了外部的变量，那么它也是一个闭包。

函数是引用类型。也就是说，将一个捕获了状态的函数赋值给另一个变量，并不会导致这些状态被复制。和对象引用类似，这些状态会被共享。换句话说，当两个闭包持有同样的局部变量时，它们是共享这个变量以及它的状态的。这可能会让你有点儿惊讶，我们将在 函数 一章中涉及这方面的更多内容。

定义在类或者协议中的函数就是方法 (**method**)，它们有一个隐式的 `self` 参数。有时候我们会把那些不是方法的函数叫做自由函数 (**free function**)，这可以将它们与方法区分开来。

类似地，在一个类型中的变量或者常数被叫做属性 (**property**)。那些为一个值定义了内存位置的属性或者变量，被称为存储属性或存储变量 (**stored property, stored variable**)。相比之下，计算属性/变量 (**computed property/variable**) 则不对应存储空间：它实际上是不带有参数的方法/函数的另一种写法。计算属性要么是只读的，要么它可以获取或者设置其他的值。

在 Swift 中，一个完整的函数名字不仅仅只包括函数的基本名（括号前面的部分），也包括它的参数标签 (`argument label`)。举例来说，将一个集合中的索引移动给定步数的函数的全名是

`index(_:offsetBy:)`, 表示该函数接受两个参数(由两个冒号表示), 其中第一个参数没有标签(用下划线表示)。在本书中, 如果我们所提及的函数处于清晰的上下文中的话, 我们通常会把标签省略掉(编译器也允许你这么做)。

自由函数和那些在结构体和枚举上调用的方法是**静态派发 (statically dispatched)** 的。对于这些函数的调用, 在编译的时候就已经确定了。对于静态派发的调用, 编译器可能会实施**内联 (inline)** 优化, 也就是说, 完全不去做函数调用, 而是将函数调用替换为函数中需要执行的代码。优化器还能够帮助丢弃或者简化那些在编译时就能确定不会被实际执行的代码。

类或者协议上的方法可能是**动态派发 (dynamically dispatched)** 的。编译器在编译时不需要知道哪个函数将被调用。在 Swift 中, 这种动态特性要么由 `ytable` 来完成, 要么通过 `selector` 和 `objc_msgSend` 来完成, 前者的处理方式和 Java 或是 C++ 中类似, 而后者只针对那些 Objective-C 运行时中用 `@objc` 修饰的类和协议上的方法。

子类型和方法重写 (**overriding**) 是实现多态 (**polymorphic**) 特性的一种手段, 也就是说, 根据类型的不同, 同样的方法会呈现出不同的行为。第二种方式是函数重载 (**overloading**), 它是指为不同的类型多次写同一个函数的行为(注意不要把重写和重载弄混了, 它们是完全不同的)。实现多态的第三种方法是通过泛型, 也就是一次性地编写能够接受任意类型的函数或者方法, 不过这些方法的实现会各有不同。与方法重写不同的是, 函数重载和泛型中的方法在编译期间就是可以确定的。我们会在泛型章节中提及关于这方面的更多内容。

Swift 的代码会被按照模块 (**module**) 进行组织。要获取另一个模块中所声明的内容, 你需要 `import` 这个模块。标准库是一个叫做 Swift 的模块, 它在每个源文件中都被自动导入了。当“whole module”优化开启时, 编译器会把同一个模块中的所有源文件一起编译。这可以解锁像是泛型特化 (**generics specialization**) 或者内联 (**inlining**) 等重要的优化策略, 来改善同一模块中代码的调用。而跨模块的调用通常优化起来就比较困难。

Swift Packager Manager 使用目标 (**target**) 这个术语来代替模块。目标可以定义模块, 但它们之间有一些边界定义上的不同: 一个 target 也可以包括 C/C++/Objective-C 的代码以及非代码的资源文件。一个产品 (**product**) 代表了作者想要提供给外部客户的功能单元。通常, 一个 product 会包含单个 target, 但是你也可以把多个 target 打包到一个 product 中, 或者让一些内部的 target 不属于任何 product。一个包 (**package**) 是把一个或多个 product 一同进行版本标记的产物。包的用户把这个包声明为包依赖 (**package dependency**), 然后从这个包中把一个或多个 product 作为目标依赖 (**target dependency**) 加载到他们自己的 target 中。

# Swift 风格指南

当我们编写这本书，或者在我们自己的项目中使用 Swift 代码时，我们尽量遵循如下的原则：

- 对于命名，在使用时能清晰表意是最重要的。因为 API 被使用的次数要远远多于被声明的次数，所以我们应当从使用者的角度来考虑它们的名字。尽快熟悉 Swift API 设计准则，并且在你自己的代码中坚持使用这些准则。
- 简洁经常有助于代码清晰，但是简洁本身不应该独自成为我们编码的目标。
- 在设计你的 API 时，尽量按照积极引导用户去做“正确的事情”的方式来进行 (Xiaodi Wu)。不要让程序员有“自掘坟墓”的机会。
- 务必为函数添加文档注释 — 特别是泛型函数。
- 类型使用大写字母开头，函数、变量和枚举成员使用小写字母开头，两者都使用驼峰式命名法。
- 使用类型推断。省略掉显而易见的类型会有助于提高可读性。
- 如果存在歧义或者在进行定义契约（比如 func 就需要显式地指定返回类型）的时候不要使用类型推断。
- 优先选择结构体，只在确实需要使用到类特有的特性或者是引用语义时才使用类。
- 除非你的设计就是希望某个类被继承使用，否则都应该将它们标记为 final。如果你允许这个类被模块内部继承，但不允许外部的用户进行子类化，那么标记这个类为 public，而不是 open。
- 除非一个闭包后面立即跟随有左花括号（比如在 if 条件中），否则都应该使用尾随闭包（trailing closure）的语法。
- 使用 guard 来提早退出方法。
- 避免对可选值进行强制解包和隐式强制解包。它们偶尔有用，但是经常需要使用它们的话往往意味着有其他不妥的地方。
- 不要写重复的代码。如果你发现你写了好几次类似的代码片段的话，试着将它们提取到一个函数里，并且考虑将这个函数转化为协议扩展的可能性。

- 尝试去使用 map 和 reduce，但这不是强制的。当合适的时候，使用 for 循环也无可厚非。高阶函数的意义是让代码可读性更高。但是如果使用 reduce 的场景难以理解的话，强行使用往往事与愿违，这种时候简单的 for 循环可能会更清晰。
- 尝试去使用不可变值：除非你需要改变某个值，否则都应该使用 let 来声明变量。不过如果能让代码更加清晰高效的话，也可以选择使用可变的版本。同样这也不是强制的，在结构体上用可变方法通常比返回一个全新的结构体更加惯用及高效。
- 结构体的属性通常可以设计为可变的，因为 API 的用户可以通过把结构体变量标记为 let 或 var 的方式来控制可变性。
- 除非你确实需要，否则不要使用 self.。不过在闭包表达式中，self 是被强制使用的，这是一个清晰的信号，表明闭包将会捕获 self.
- 尽可能地对现有的类型和协议进行扩展，而不是写一些自由函数。这有助于提高可读性，让别人更容易发现你的代码。
- 当有意义时，去扩展已有（标准库中的）类型，不必犹豫。

最后，关于整本书中的示例代码我们还有一点补充说明：为了节省空间并且专注于重要的部分，我们通常会省略 import 语句，这往往会导致代码无法编译。如果你想要自己尝试运行这些示例代码，而编译器告诉你它不认识某个特定的符号的话，请尝试添加 import Foundation 或者 import UIKit 这样的语句。

## 修订历史

第五版 (2022 年 4 月)

- 所有章节按照 Swift 5.6 进行修订。
- 新章节：
  - 并发
  - 属性 (之前是函数一章中的部分内容)
- 新内容：
  - 内建集合类型：

→ RangeSet

→ 函数:

→ Result Builder

→ 属性:

→ 属性包装

→ 结构体和类

→ willSet 对写时复制的破坏

→ 泛型:

→ 泛型是静态派发的

→ 泛型的工作方式

→ 协议:

→ 回溯满足协议

→ 存在体

→ 不透明类型

→ 大幅修改:

→ 函数

→ 编码和解码

## 第四版 (2019 年 5 月)

→ 所有章节按照 Swift 5 进行修订。

→ 新章节: 枚举

→ 重写一些章节:

→ 结构体和类

→ 泛型

→ 协议

→ 重大修改并添加新的内容：

→ 字符串

→ 集合类型协议

→ 错误处理

→ 重排章节的顺序；把集合类型协议移到书的后半部分，为读者带来更平滑的学习曲线。

→ Florian Kugler 作为联合作者加入。

### 第三版 (2017 年 10 月)

→ 所有章节按照 Swift 4 进行修订。

→ 新章节：编码和解码

→ 重大修改并添加新的内容：

→ 内建集合类型

→ 集合类型协议

→ 函数 (new section on key paths)

→ 字符串 (more than 40 percent longer)

→ 互用性

→ 全文可用于 Xcode 的 playgrounds 上显示。

### 第二版 (2016 年 9 月)

→ 所有章节按照 Swift 3 进行修订。

→ 把集合类型这一章划分成内建集合类型和集合类型协议两章。

→ 对整本书做重大修改并添加新的内容，尤其是在：

→ 集合类型协议

→ 函数

→ 泛型

→ 全文可用在 iPad 上的 Swift playground app 上显示。

→ Ole Begemann 作为联合作者加入。

### 第一版 (2016 年 3 月)

→ 首次发行，覆盖 Swift 2.2。

## 译者简介

本书在翻译过程中受到了很多社区朋友的无私帮助，没有大家的努力和反馈，我们将很难顺利完成这些工作。现对第四版更新中的联合译者进行介绍：

### 崔轶

泊学 创始人。2016 年，独自开发了泊学网站的前台和服务端，并于 2016 至 2019 年间，独自录制并发布了泊学网站上关于 Swift & iOS, PHP & Laravel, Nginx & Docker 等技术领域的 500+ 技术视频。目前，正致力 Swift 语言的深入研究，以及泊学 App 客户端和服务端的开发和分享。

### 茆子君

iOS 码农一枚。因为在 iPhone 4 发布的时候被这么酷的一个设备给震惊到了，就特别想在这个设备上开发各种有意思 app，以此为契机从后端转向了 iOS 开发。两年前来到日本加入了 LINE，目前在 SmartNews 参与同名 app 的开发。对算法，数据结构一直保持着兴趣，对 Swift 和 iOS 的动画充满了热情。一直追求写出更漂亮的代码。

# 内建集合类型

2

在所有的编程语言中，元素的集合都是最重要的数据类型。编程语言对不同集合类型的良好支持，是决定编程效率和幸福指数的重要因素。因此，Swift 在序列和集合这方面进行了特别的强调，标准库的开发者对于该部分所投入的精力远超其他部分，甚至让我们觉得标准库几乎就是用来专门处理集合类型的。正是有了这样的努力，我们能够使用到非常强大的集合模型，它比你所习惯的其他语言的集合拥有更好的可扩展性，不过同时它也相当复杂。

在本章中，我们将会讨论 Swift 中内建的几种主要集合类型，并重点研究如何以符合语言习惯的方式高效地使用它们。在集合类型协议中，我们会沿着抽象的阶梯蜿蜒而上，去探究标准库中集合协议的工作原理。

## 数组

在 Swift 中最常用的集合类型非数组莫属。数组是一个容器，它以有序的方式存储相同类型的元素，并且允许随机访问每个元素。举个例子，要创建一个元素是数字的数组，可以这样：

```
// 斐波那契数列  
let fibs = [0, 1, 1, 2, 3, 5]
```

## 数组和可变性

要是我们使用像是 `append(_:)` 这样的方法修改上面定义的数组，那么会得到一个编译错误。这是因为 `fibs` 是用 `let` 定义为常量的。在很多情况下，这是正确的做法，因为它可以避免我们不小心修改数组。如果我们想让数组是一个变量，那么必须用 `var` 来定义它：

```
var mutableFibs = [0, 1, 1, 2, 3, 5]
```

现在我们就能很容易地为数组添加单个或是一系列元素了：

```
mutableFibs.append(8)  
mutableFibs.append(contentsOf: [13, 21])  
mutableFibs // [0, 1, 1, 2, 3, 5, 8, 13, 21]
```

区别使用 `var` 和 `let` 可以给我们带来不少好处。因为具有不变性，所以使用 `let` 定义的常量更容易被理解。当你读到类似 `let fibs = ...` 这样的声明时，你可以确定 `fibs` 的值将永远不变，这一

点是由编译器强制保证的。这在阅读代码的时候会很有帮助。不过，要注意这只针对那些具有值语义的类型。使用 let 定义一个指向类实例的引用类型时，它保证的是这个引用永远不会发生变化，也就是说，你不能再给这个引用赋一个新的值，但是这个引用所指向的对象却是可以改变的。我们将在结构体和类中更加详尽地介绍两者的区别。

和标准库中所有集合类型一样，数组是具有值语义的。当你把一个已经存在的数组赋值给另一个变量时，这个数组的内容会被复制。举个例子，在下面的代码中，x 将不会被更改：

```
var x = [1,2,3]
var y = x
y.append(4)
y // [1, 2, 3, 4]
x // [1, 2, 3]
```

var y = x 语句复制了 x，所以在将 4 添加到 y 后，x 并不会发生改变，它的值依然是 [1,2,3]。当你把一个数组传递给一个函数时，会发生同样的事情；函数将得到这个数组的一份本地复制，所有对它的改变都不会影响调用者所持有的数组。

对比一下 Foundation 框架中 NSArray 在可变特性上的处理方法。NSArray 中没有更改方法，想要更改一个数组，你必须使用 NSMutableArray。但是，就算你拥有的是一个不可变的 NSArray，但是它的引用特性并不能保证这个数组不会被改变：

对比一下很多其他语言 (比如 JavaScript, Java 和使用 Foundation 框架中的 NSArray 的 Objective-C) 中对可变特性上处理的方法。这些语言中的数组使用引用语义：通过一个变量对数组的内容进行变更，会隐式地改变其他所有引用了这个数组的变量，因为他们全都指向了同一个存储。比如在 JavaScript 中的这个例子：

```
// 'const' 让**变量** a 和 b 不可变。
const a = [1,2,3];
const b = a;
// 但是它们**引用**的对象依旧是可变的。
b.push(4);
console.log(b); // [ 1, 2, 3, 4 ]
console.log(a); // [ 1, 2, 3, 4 ]
```

正确的方式是在赋值时，先手动进行复制：

```
const c = [1,2,3];
// 进行明确复制。
const d = c.slice();
d.push(4);
console.log(d); // [ 1, 2, 3, 4 ]
console.log(c); // [ 1, 2, 3 ]
```

这个操作非常容易被忘记，也非常容易出错。比如，如果一个对象从它的内部状态中返回了一个数组，但却没有将它进行复制，那么当调用者改变这个数组时，这个对象会突然发现状态被破坏了。Swift 通过将值语义赋予集合类型来避免这一问题。

在每次赋值时都创建一份复制有可能造成性能问题，不过实际上 Swift 标准库中的所有集合类型都使用了“写时复制”这一技术，它能够保证只在必要的时候对数据进行复制。在我们的例子中，直到 `y.append` 被调用的之前，`x` 和 `y` 都将共享内部的存储。在结构体和类中我们也将仔细研究值语义，并告诉你如何为自己的类型实现写时复制特性。

## 数组索引

Swift 数组提供了你能想到的所有常规操作方法，像是 `isEmpty` 和 `count`。数组也允许通过下标直接访问指定索引上的元素，像是 `fibs[3]`。不过要牢记在使用下标获取元素之前，要确保索引值没有超出范围。比如使用索引值 3，你需要保证数组中至少有 4 个元素。否则，程序将会崩溃。

Swift 也有很多无需计算索引就能操作数组的方法：

- 想要迭代数组？`for x in array`
- 想要迭代除了第一个元素以外的数组其余部分？`for x in array.dropFirst()`
- 想要迭代除了最后 5 个元素以外的数组？`for x in array.dropLast(5)`
- 想要为数组中的所有元素编号？`for (num, element) in collection.enumerated()`
- 想要列举下标和元素？`for (index, element) in zip(array.indices, array)`

→ 想要寻找一个指定元素的位置?

```
if let idx = array.firstIndex { someMatchingLogic($0) }
```

→ 想要对数组中的所有元素进行变形? `array.map { someTransformation($0) }`

→ 想要筛选出符合某个特定标准的元素? `array.filter { someCriteria($0) }`

Swift 不鼓励你去做索引计算的另一个标志, 传统的 C 风格 for 循环在这门语言中的缺席证明了这一点。手动计算和使用索引值往往会造成很多潜在的 bug, 所以最好避免这么做。

但是有些时候你仍然不得不使用索引。对于数组索引来说, 当你这么做时, 应该已经对索引计算背后的逻辑进行过认真思考。在这个前提下, 如果每次都要对获取的结果进行解包的话就显得多余了, 因为这意味着你不信任你的代码。但实际上你是信任自己的代码的, 你知道这些下标都是有效的, 所以你可能会选择将结果进行强制解包。一方面这十分麻烦, 另一方面也是一个坏习惯。当强制解包变成一种习惯后, 最终很可能你会不小心强制解包了本来不应该解包的东西。所以, 为了避免这个行为变成习惯, 数组根本没有给你这个选项。

无效的下标操作会造成可控的崩溃, 有时候这种行为可能会被叫做不安全, 但这只是安全性的一个方面。从内存安全的角度上说, 下标操作是完全安全的, 标准库中的集合总是会执行边界检查, 并禁止那些越界索引对内存的访问。在 Swift 中, “安全”这个术语一般指的是内存安全, 避免发生未定义的行为。

其他操作的行为略有不同。`first` 和 `last` 属性返回一个可选值, 当数组为空时, 它们返回 `nil`。`first` 相当于 `isEmpty ? nil : self[0]`。类似地, 如果数组为空时调用 `removeLast`, 那么将会导致崩溃; 然而 `popLast` 在数组不为空时删除最后一个元素并返回它, 在数组为空时, 它将不执行任何操作, 直接返回 `nil`。你应该根据自己的需要来选取到底使用哪个方法: 当把数组用做栈时, 你可能总是想要将 `empty` 检查和移除最后元素组合起来使用; 而另一方面, 如果你已经知道数组是否为空, 那再去处理可选值就没有必要了。

我们会在本章后面讨论字典的时候再次遇到关于这部分的权衡。除此之外, 关于可选值我们会有一整章的内容对它进行讨论。

## 数组变形

## Map

对数组中的每个值执行转换操作是一个很常见的任务。每个程序员可能都写过上百次这样的代码：创建一个新数组，对已有数组中的元素进行循环依次取出其中元素，对取出的元素进行操作，并把操作的结果添加到新数组。比如，下面的代码计算了一个整数数组里的元素的平方：

```
var squared: [Int] = []
for fib in fibs {
    squared.append(fib * fib)
}
squared // [0, 1, 1, 4, 9, 25]
```

Swift 数组拥有 map 方法，这个方法来自函数式编程的世界。下面的例子使用了 map 来完成同样的操作：

```
let squares = fibs.map { fib in fib * fib }
squares // [0, 1, 1, 4, 9, 25]
```

这个版本有三大优势。首先，它很短。长度短一般意味着错误少，不过更重要的是，它比原来更清晰。所有无关的内容都被移除了，一旦你习惯了 map 满天飞的世界，你就会发现 map 就像是一个信号，一旦你看到它，就会知道即将有一个函数被作用在数组的每个元素上，并返回另一个数组，它将包含所有被转换后的结果。

其次，squared 将由 map 的结果得到，并且我们不会去改变它的值，所以也就不再需要用 var 来进行声明了。结果将会在完全构建后再由 map 传递出来，所以我们可以将其声明为 let。另外，由于返回的数组元素的类型可以从传递给 map 的函数中推断出来，我们也不再需要为 squared 显式地指明类型了。

最后，创造 map 函数并不难，你只需要把 for 循环中的代码模板部分，用一个泛型函数封装起来就可以了。下面是一种可能的实现方式 (在 Swift 中，它实际上是 Sequence 协议的一个扩展，我们将在集合类型协议里讨论它)：

```
extension Array {
    func map<T>(_ transform: (Element) -> T) -> [T] {
```

```
var result: [T] = []
result.reserveCapacity(count)
for x in self {
    result.append(transform(x))
}
return result
}
```

Element 是数组中包含的元素类型的占位符，T 是元素转换之后的类型占位符。map 函数本身并不关心 Element 和 T 究竟是什么，它们可以是任意类型。T 的具体类型由调用者传入给 map 的 transform 方法的返回值类型来决定。有关泛型参数的细节，请参阅泛型。

实际上，这个函数的签名应该是

```
func map<T>(_ transform: (Element) throws -> T) rethrows -> [T]
```

也就是说，对于可能抛出错误的变形函数，map 会把错误转发给调用者。我们会在错误处理一章里覆盖这个细节。在这里，我们选择去掉错误处理的修饰，这样看起来会更简单一些。如果你感兴趣，可以看看 GitHub 上 Swift 仓库的 Sequence.map 的源码实现。

## 使用函数将行为参数化

即使你已经很熟悉 map 了，也请花一点时间来想一想 map 的实现。是什么让它如此通用而且有用？

map 设法将模板代码分离出来，这些模板代码并不会随着每次调用发生变动，发生变动的是那些功能代码——也就是如何变换每个元素的逻辑。map 通过把调用者所提供的变换函数作为参数来做到这一点。

纵观标准库，这种把行为参数化的设计模式有很多。例如，在 Array 以及其他集合类型中，有不下十多个方法接受一个函数作为参数，来自定义它们的行为：

- **map** 和 **flatMap** — 对元素进行变换。
- **filter** — 只包含特定的元素。
- **allSatisfy** — 针对一个条件测试所有元素。
- **reduce** — 将元素聚合成一个值。
- **forEach** — 访问每个元素。
- **sort(by:)**, **sorted(by:)**, **lexicographicallyPrecedes(\_:by:)**, 和 **partition(by:)** — 重排元素。
- **firstIndex(where:)**, **lastIndex(where:)**, **first(where:)**, **last(where:)**, 和 **contains(where:)** — 一个元素是否存在?
- **min(by:)** 和 **max(by:)** — 找到所有元素中的最小或最大值。
- **elementsEqual(\_:by:)** 和 **starts(with:by:)** — 将元素与另一个数组进行比较。
- **split(whereSeparator:)** — 把所有元素分成多个数组。
- **prefix(while:)** — 从头取元素直到条件不成立。
- **drop(while:)** — 当条件为真时, 丢弃元素; 一旦不为真, 返回其余的元素 (和 **prefix** 类似, 不过返回相反的集合)。
- **removeAll(where:)** — 删除所有符合条件的元素。

所有这些函数的目的都是为了摆脱代码中那些杂乱无趣的部分, 比如像是新建数组, 或对源数据进行 for 循环之类的事情。这些部分都被一个单独的, 描述做什么的单词替代了。这就可以重点突出那些程序员真正想要表达的逻辑代码。

这些函数中有一些拥有默认行为。除非你进行过指定, 否则 **sort** 默认将会把可以作比较的元素按照升序排列。**contains** 对于可以判等的元素, 会直接检查两个元素是否相等。这些默认行为让代码变得更加易读。升序排列非常自然, 因此 `array.sort()` 的意义也很符合直觉。而对于 `array.index(of: "foo")` 这样的表达方式, 也要比 `array.index { $0 == "foo" }` 更清晰。

不过在上面的例子中, 它们都只是对于常规情况的简写。集合中的元素并不一定需要可以作比较, 也不一定需要可以判等, 甚至你不需要对整个元素进行操作, 比如, 对一个包含 Person 对象的数组, 你可以通过他们的年龄进行排序 (`people.sort { $0.age < $1.age }`), 或者是检查集合

中有没有包含未成年人 (`people.contains { $0.age < 18 }`)。你也可以对转变后的元素进行比较，比如通过 `people.sort { $0.name.uppercased() < $1.name.uppercased() }` 来进行忽略大小写的排序，虽然这么做的效率不会很高。

还有一些类似的很有用的函数，可以接受一个函数来指定行为。虽然它们并不在标准库中，但自己实现出来也很简单，我们建议你亲自动手试一下：

- **accumulate** — 累加，和 `reduce` 类似，不过是将所有元素合并到一个数组中，并保留合并时每一步的值。
- **count(where:)** — 计算满足条件的元素的个数 (它原本应该在 Swift 5.0 的时候被加入，但由于和 `count` 属性的名字冲突而延迟了，它可能会在后续版本中被重新引入)。
- **indices(where:)** — 返回一个包含满足某个条件的所有元素的索引列表，和 `index(where:)` 类似，但是不会在遇到首个元素时就停止。

如果在你的代码中，发现多个地方都有遍历一个数组并做相同或类似的事情时，可以考虑给 `Array` 写一个扩展。比如，下面的代码将数组中的元素按照相邻且相等的方式拆分开：

```
let array: [Int] = [1, 2, 2, 2, 3, 4, 4]
var result: [[Int]] = array.isEmpty ? [] : [[array[0]]]
for (previous, current) in zip(array, array.dropFirst()) {
    if previous == current {
        result[result.endIndex-1].append(current)
    } else {
        result.append([current])
    }
}
result // [[1], [2, 2, 2], [3], [4, 4]]
```

我们可以从逻辑中提取出遍历数组相邻元素的代码，来正式定义这个算法。因为不同应用之间的区别就是决定在哪里拆分数组，所以我们通过一个函数参数让调用者自定义这部分逻辑：

```
extension Array {
    func split(where condition: (Element, Element) -> Bool) -> [[Element]] {
```

```
var result: [[Element]] = self.isEmpty ? [] : [[self[0]]]
for (previous, current) in zip(self, self.dropFirst()) {
    if condition(previous, current) {
        result.append([current])
    } else {
        result[result.endIndex-1].append(current)
    }
}
return result
}
```

这样，就可以用下面的代码替换掉 for 循环：

```
let parts = array.split { $0 != $1 }
parts // [[1], [2, 2, 2], [3], [4, 4]]
```

在这种特定情况下，甚至可以写成这样：

```
let parts2 = array.split(where: !=)
```

这么做的好处和我们在介绍 map 时所描述的一样的，相较 for 循环，split(where:) 版本的可读性更好。虽然 for 循环也很简单，但是在你的头脑里始终还是要去做个循环，这加重了理解的负担。使用 split(where:) 可以减少出错的可能性，而且它允许你使用 let 而不是 var 来声明结果变量。

这个 split(where:) 操作是 Apple 的 [Swift Algorithms](#) 包中的一部分，它被叫做了 [chunked\(by:\)](#)。[Swift Algorithms](#) 是一个实现了高质量的常用集合类型操作的开源库。它在 Swift 社区中作为一个低阻力的场所，用来迭代各种算法，而不必着急让它们立刻达到能添加到标准库的高度。那些被证明有用的功能可以在之后再迁移到标准库中。

我们在本书后面会进一步涉及扩展集合类型和使用函数的相关内容。

## 可变和带有状态的闭包

当遍历一个数组的时候，你可以使用 map 来执行一些带副作用的操作（比如将元素插入到一个查找表中）。我们不推荐这么做，来看看下面这个例子：

```
array.map { item in
    table.insert(item)
}
```

这将副作用（改变了查找表）隐藏在了一个看起来只是对数组变形的操作中。如果你看到类似上面这样的代码，使用简单的 for 循环显然是比使用 map 这样的函数更好的选择。在这种情况下，forEach 方法也比 map 更合适，但是 forEach 本身存在一些问题，我们一会儿会详细讨论。

这样带有副作用的做法，和故意给闭包一个局部状态有本质不同，后者是一种非常有用的技术。闭包是指那些可以捕获和修改自身作用域之外的变量的函数，当它和高阶函数结合时也就成为了一种强大的工具。举个例子，方才我们提到的 accumulate 函数就可以用 map 结合一个带有状态的闭包来实现：

```
extension Array {
    func accumulate<Result>(_ initialResult: Result,
                           _ nextPartialResult: (Result, Element) -> Result) -> [Result] {
        var running = initialResult
        return map { next in
            running = nextPartialResult(running, next)
            return running
        }
    }
}
```

这个函数创建了一个中间变量来存储每一步的值，然后使用 map 来从这个中间值逐步计算结果数组：

```
[1,2,3,4].accumulate(0, +) // [1, 3, 6, 10]
```

## filter

另一个常见操作是检查一个数组，然后将这个数组中符合一定条件的元素过滤出来并用它们创建一个新的数组。对数组进行循环并且根据条件过滤其中元素的模式可以用 filter 方法表示：

```
let nums = [1,2,3,4,5,6,7,8,9,10]
nums.filter { num in num % 2 == 0 } // [2, 4, 6, 8, 10]
```

对于闭包表达式的参数，我们可以使用简写方式来使代码更加简短。我们可以不用写出 num 参数，而将上面的代码重写为：

```
nums.filter { $0 % 2 == 0 } // [2, 4, 6, 8, 10]
```

对于很短的闭包来说，这样做有助于提高可读性。但是如果闭包比较复杂的话，更好的做法应该是像我们之前那样，显式地把参数名字写出来。不过这更多的是一种个人选择，使用一眼看上去更易读的版本就好。一个不错的经验是，如果闭包可以很好地写在一行里的话，那么使用简写会更合适。

通过组合使用 map 和 filter，我们现在可以轻易完成很多数组操作，而不需要引入中间变量。这会使得最终的代码变得更短更易读。比如，为了寻找所有 100 以内的偶平方数，我们可以对 0..<10 进行 map 来得到所有平方数，然后再过滤掉所有奇数：

```
(1..<10).map { $0 * $0 }.filter { $0 % 2 == 0 } // [4, 16, 36, 64]
```

filter 的实现看起来和 map 很类似：

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

```
}
```

这里有两个关于性能的小提示：首先，要注意像这样链式使用 `map` 和 `filter` 会创建一个中间数组（也就是 `map` 操作的结果），这个数组会被立即抛弃。对于我们这个小例子来说这不会有什么问题，但是对于大的集合类型或者更长的链式调用，这种额外的内存申请会对性能造成影响。我们可以通过在链开始时加入 `.lazy`，来让所有的变形延迟发生，来避免这些中间数组。只有在最后，我们才需要将这个延迟集合类型转换回一般的数组：

```
let lazyFilter = (1..<10)
    .lazy.map { $0 * $0 }.filter { $0 % 2 == 0 }
let filtered = Array(lazyFilter) // [4, 16, 36, 64]
```

我们会在集合类型协议中对延迟序列进行更多讨论。

第二个提示是，如果你正在写下面这样的代码，请不要这么做！

```
bigArray.filter { someCondition }.count > 0
```

`filter` 会创建一个全新的数组，并且会对数组中的每个元素都进行操作。然而在上面这段代码中，这显然是不必要的。上面的代码只需要检查是否至少有一个元素满足条件，在这个情景下，使用 `contains(where:)` 更为合适：

```
bigArray.contains { someCondition }
```

这种做法会快得多，主要因为两个方面：它不会仅仅为了计数而去创建一个全新的数组，并且一旦找到了第一个匹配的元素，它就将提前退出。一般来说，你只应该在需要所有结果时才去选择使用 `filter`。

## reduce

`map` 和 `filter` 都作用在一个数组上，并产生另一个新的、经过修改的数组。不过有时候，你可能会想把所有元素合并为一个新的单一的值。比如，要是我们想将元素的值全部加起来，可以这样写：

```
let fibs = [0, 1, 1, 2, 3, 5]
var total = 0
for num in fibs {
    total = total + num
}
total // 12
```

reduce 方法采用这种模式，并抽象出两部分：一个初始值（在这里是 0），以及一个将中间值（total）与序列中的元素（num）进行合并的函数。使用 reduce，我们可以将上面的例子重写为这样：

```
let sum = fibs.reduce(0) { total, num in total + num } // 12
```

运算符也是函数，所以我们也可以把上面的例子写成这样：

```
fibs.reduce(0, +) // 12
```

reduce 的输出值的类型不必和元素的类型相同。举个例子，如果我们想将一个整数列表转换为一个字符串，并且每个数字后面跟一个逗号和空格，那么可以这样：

```
fibs.reduce("") { str, num in str + "\\"(num), " } // 0, 1, 1, 2, 3, 5,
```

reduce 的实现是这样的：

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

```
}
```

另一个关于性能的小提示：reduce 相当灵活，所以在构建数组或者是执行其他操作时看到 reduce 的话不足为奇。比如，你可以单纯靠使用 reduce 去实现 map 和 filter：

```
extension Array {  
    func map2<T>(_ transform: (Element) -> T) -> [T] {  
        return reduce([]) {  
            $0 + [transform($1)]  
        }  
    }  
  
    func filter2(_ isIncluded: (Element) -> Bool) -> [Element] {  
        return reduce([]) {  
            isIncluded($1) ? $0 + [$1] : $0  
        }  
    }  
}
```

这样的实现符合美学，并且不再需要那些啰嗦的命令式 for 循环。但 Swift 不是 Haskell，Swift 的数组并不是列表（list）。在这里，当每次通过追加一个变换过的或符合条件的元素到上一次的结果，来执行合并操作时，就会创建一个全新的数组。这意味着上面两个实现的复杂度是  $O(n^2)$ ，而不是  $O(n)$ 。随着数组长度的增加，执行这些函数所消耗的时间将以平方关系增加。

reduce 还有另外一个版本，它的类型稍有不同。负责将中间结果和一个元素合并的 reducer 函数，也可以接受一个 inout 的 Result 作为参数：

```
public func reduce<Result>(into initialResult: Result,  
    _ updateAccumulatingResult:  
        (_ partialResult: inout Result, Element) throws -> ()  
) rethrows -> Result
```

我们会在函数以及结构体和类的相关章节中探究 inout 参数的细节，现在的话，你可以把 inout Result 看作是一个可变的参数：我们可以在函数内部更改它。这让我们可以用一种高效得多的方式实现 filter：

```
extension Array {  
    func filter3(_ isIncluded: (Element) -> Bool) -> [Element] {  
        return reduce(into: []) { result, element in  
            if isIncluded(element) {  
                result.append(element)  
            }  
        }  
    }  
}
```

当使用 inout 时，编译器不会每次都创建一个新的数组，这样一来，这个版本的 filter 时间复杂度再次回到了  $O(n)$ 。当 reduce(into:\_:) 的调用被编译器内联时，生成的代码通常会和使用 for 循环所得到的代码是一致的。

## 一个展平的 map

有时候我们想对一个数组进行 map，但这个变形函数返回的是另一个数组，而不是单独的元素。

举个例子，假如我们有个叫做 extractLinks 的函数，它会读取一些 Markdown 文本，并返回一个包含文本中所有链接的 URL 数组。这个函数的签名是这样的：

```
func extractLinks(markdownFile: String) -> [URL]
```

如果我们有一堆 Markdown 文件，并且想将这些文件中所有的链接都提取到一个单独的数组中的话，可以尝试使用 markdownFiles.map(extractLinks)。不过问题是这个方法返回的是一个包含了 URL 的数组的数组，其中每个元素都是一个文件中的 URL 的数组。现在你可以在执行 map 得到了一个数组的数组之后，调用 joined 把这个二维数组展平 (flatten) 成一个一维数组：

```
let markdownFiles: [String] = // ...  
let nestedLinks = markdownFiles.map(extractLinks)
```

```
let links = nestedLinks.joined()
```

flatMap 方法将变换和展平这两个操作合并为一个步骤。markdownFiles.flatMap(links) 把所有 Markdown 文件中的 URL 放到一个数组里返回。

flatMap 的函数签名看起来和 map 基本一致，只是它的变换函数返回的是一个数组。在实现中，它使用 append(contentsOf:) 代替了 append(\_:)，这样返回的数组是展平的了：

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

flatMap 的另一个常见使用情景是将不同数组里的元素进行合并。为了得到两个数组中元素的所有配对组合，我们可以对其中一个数组进行 flatMap，然后在变换函数中对另一个数组进行 map 操作：

```
let suits = ["♠", "♥", "♣", "♦"]  
let ranks = ["J", "Q", "K", "A"]  
let result = suits.flatMap { suit in  
    ranks.map { rank in  
        (suit, rank)  
    }  
}  
/*  
[("♠", "J"), ("♠", "Q"), ("♠", "K"), ("♠", "A"), ("♥", "J"), ("♥",  
"Q"), ("♥", "K"), ("♥", "A"), ("♣", "J"), ("♣", "Q"), ("♣", "K"),  
("♣", "A"), ("♦", "J"), ("♦", "Q"), ("♦", "K"), ("♦", "A")]
```

```
*/
```

## 使用 forEach 进行迭代

我们最后要讨论的操作是 `forEach`。它和 `for` 循环的工作方式非常类似：把传入的函数对序列中的每个元素执行一次。和 `map` 不同，`forEach` 不返回任何值，这一点特别适合执行那些带副作用的操作。让我们不暇思索地从用 `forEach` 替换 `for` 循环开始：

```
for element in [1,2,3] {  
    print(element)  
}  
  
[1,2,3].forEach { element in  
    print(element)  
}
```

这没什么特别之处，不过如果你想要对集合中的每个元素都调用一个函数的话，使用 `forEach` 会比较合适。相比于传递一个闭包表达式来说，传递一个函数名给 `forEach` 可以使代码更加简洁和紧凑。比如你正在 iOS 上实现一个 `view controller`，然后想把一个数组中的视图都加到当前 `view` 上的话，只需要写 `theViews.forEach(view.addSubview)` 就足够了。

不过，`for` 循环和 `forEach` 有些细微的不同，值得我们注意。比如，当一个 `for` 循环中有 `return` 语句时，将它重写为 `forEach` 会造成代码行为上的极大区别。让我们举个例子，下面的代码是通过结合使用 `where` 限定条件和 `for` 循环完成的：

```
extension Array where Element: Equatable {  
    func firstIndex(of element: Element) -> Int? {  
        for idx in self.indices where self[idx] == element {  
            return idx  
        }  
        return nil  
    }  
}
```

我们不能直接将 where 语句加入到 forEach 中，所以我们可能会用 filter 来重写这段代码（实际上这段代码是错误的）：

```
extension Array where Element: Equatable {  
    func firstIndex_foreach(of element: Element) -> Int? {  
        self.indices.filter { idx in  
            self[idx] == element  
        }.forEach { idx in  
            return idx  
        }  
        return nil  
    }  
}
```

在 forEach 中的 return 并不能让外部函数返回，它仅仅只是让闭包本身返回。在这种情况下，编译器会发现 return 语句的参数没有被使用，从而给出警告，所以我们可以找到问题所在。但我们不应该将找到所有这类错误的希望寄托在编译器上。

再思考一下下面这个简单的例子：

```
(1..  
10).forEach { number in  
    print(number)  
    if number > 2 { return }  
}
```

你可能一开始还没反应过来，其实这段代码将会把输入的数字全部打印出来。return 语句并不会终止循环，它做的仅仅是将从闭包中返回，因此在 forEach 的实现中会开始下一个循环的迭代。

在某些情况下，比如上面的 addSubview 的例子里，forEach 可能会比 for 循环更好。不过，因为 return 在其中的行为不太明确，我们建议大多数其他情况下不要用 forEach。这种时候，使用常规的 for 循环可能会更好。

## 数组切片

除了通过单独的下标来访问数组中的单个元素 (比如 `fibs[0]`)，我们还可以通过下标来获取某个范围中的元素。比如，想要得到数组中除了首个元素以外的其他元素，可以这么做：

```
let slice = fibs[1...]
slice // [1, 1, 2, 3, 5]
type(of: slice) // ArraySlice<Int>
```

它将返回数组的一个切片 (`slice`)，其中包含了原数组中从第二个元素开始的所有部分。得到的结果的类型是 `ArraySlice`，而不是 `Array`。切片类型只是数组的一种表示方式，它背后的数据仍然是原来的数组，只不过是用切片的方式来进行表示。因为数组的元素不会被复制，所以创建一个切片的代价是很小的。

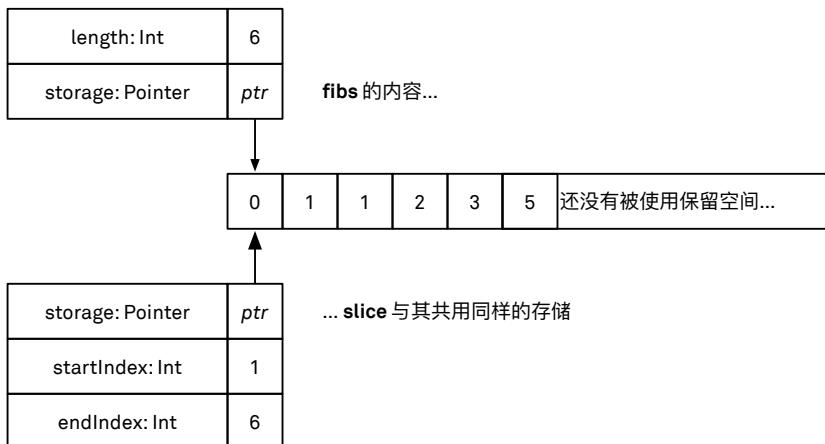


Figure 2.1: 数组切片

因为 `ArraySlice` 和 `Array` 都满足了相同的协议 (当中最重要的就是 Collection 协议)，所以两者具有的方法是一致的，因此你可以把切片当做数组来进行处理。如果你需要将切片转换为数组的话，可以通过把它传递给 `Array` 的构建方法来完成：

```
let newArray = Array(slice)
```

```
type(of: newArray) // Array<Int>
```

需要谨记的是切片和它背后的数组总是使用相同的索引来引用元素的。因此，切片索引不需要从零开始。例如，在上面我们用 `fibs[1...]` 创建的切片的第一个元素的索引是 1，因此错误地访问 `slice[0]` 元素会使我们的程序因越界而崩溃。如果你操作切片的话，我们建议你总是基于 `startIndex` 和 `endIndex` 属性做索引计算。即使你正在处理的是一个这两个属性分别是 0 和 `count-1` 的普通数组，也请做到这一点，因为这个隐含的假设很容易被打破。我们会在集合类型协议里对切片上的这个属性做更多讨论。

## 字典

另一个关键的数据结构是 `Dictionary`。字典包含键以及它们所对应的值，其中每个键是唯一的。通过键来获取值所花费的平均时间是常数量级的，作为对比，在数组中搜寻一个特定元素所花的时间将与数组尺寸成正比。和数组有所不同，字典是无序的，使用 `for` 循环来枚举字典中的键值对时，顺序是不确定的。

在下面的例子中，我们虚构一个 app 的设置界面，并使用字典作为模型数据层。这个界面由一系列的设置项构成，每一个设置项都有自己的名字(也就是我们字典中的键)和值。值可以是文本，数字或者布尔值之中的一种。我们使用一个带有关联值的 `enum` 来表示：

```
enum Setting {  
    case text(String)  
    case int(Int)  
    case bool(Bool)  
}
```

```
let defaultSettings: [String: Setting] = [  
    "Airplane Mode": .bool(false),  
    "Name": .text("My iPhone"),  
]  
  
defaultSettings["Name"] // Optional(setting.text("My iPhone"))
```

我们使用下标来得到某个设置的值。字典查找总是返回一个可选值，当指定的键不存在时，它就返回 nil。这点和数组有所不同，在数组中，使用越界下标进行访问将会导致程序崩溃。

与通常使用的以键为参数的下标方法形成对比的是，作为实现 Collection 协议的一部分，字典也有一个接受索引的下标方法。就像数组一样，当用一个无效的索引调用这个方法时，程序会崩溃。不过除了在泛型集合类型的算法外，这个下标方法几乎不会被用到。

从理论上来说，这个区别的原因是数组索引和字典的键的使用方式有很大不同。我们已经讨论过，对数组来说，你很少需要直接使用数组的索引。即使你用到索引，这个索引也一般是通过某些方式计算得来的（比如从 `0..array.count` 这样的范围内获取到）。也就是说，使用一个无效索引一般都是程序员的失误。而另一方面，字典的键往往是从其他渠道得来的，从字典本身获取键反而十分少见。

与数组不同，字典是一种稀疏结构。比如，在“name”键下存在某个值这件事，对确定“address”键下是否有值毫无帮助。

## 可变性

和数组一样，使用 `let` 定义的字典是不可变的：你不能向其中添加、删除或者修改条目。同样我们可以用 `var` 定义一个可变字典。想要从字典中移除一个值的话，要么用下标将对应的值设为 `nil`，要么调用 `removeValue(forKey:)`。后一种方法还会将被删除的值返回（如果待删除的键不存在，则返回 `nil`）。对于一个不可变的字典，想要进行修改的话，我们需要进行复制：

```
var userSettings = defaultSettings
userSettings["Name"] = .text("Jared's iPhone")
userSettings["Do Not Disturb"] = .bool(true)
```

注意，`defaultSettings` 的值同样没有改变。和键的移除类似，除了使用下标之外，还有一种方法可以更新字典内容，那就是 `updateValue(_:_forKey:)`。如果之前键已经存在的话，这个方法会返回更新前的值：

```
let oldName = userSettings
```

```
.updateValue(.text("Jane's iPhone"), forKey: "Name")
userSettings["Name"] // Optional(Setting.text("Jane's iPhone"))
oldName // Optional(Setting.text("Jared's iPhone"))
```

## 一些有用的字典方法

如果我们想要将一个默认的设置字典和某个用户更改过的自定义设置字典合并，应该怎么做呢？自定义的设置应该要覆盖默认设置，同时得到的字典中应当依然含有那些没有被自定义的键值。换句话说，我们需要合并两个字典，用来做合并的字典需要覆盖重复的键。

Dictionary 有一个 `merge(_:uniquingKeysWith:)`，它接受两个参数，第一个是要进行合并的键值对，第二个是定义如何合并相同键的两个值的函数。我们可以使用这个方法将一个字典合并至另一个字典中去，如下例所示：

```
var settings = defaultSettings
let overriddenSettings: [String:Setting] = ["Name":.text("Jane's iPhone")]
settings.merge(overriddenSettings, uniquingKeysWith: { $1 })
settings
// ["Name": Setting.text("Jane's iPhone"), "Airplane Mode": Setting.bool(false)]
```

在上面的例子中，我们使用了 `{ $1 }` 来作为合并两个值的策略。也就是说，如果某个键同时存在于 `settings` 和 `overriddenSettings` 中时，我们使用 `overriddenSettings` 中的值。

我们还可以从一个 `(Key,Value)` 键值对的序列中构建一个新的字典。如果我们能够保证键是唯一的，那么就可以使用 `Dictionary(uniqueKeysWithValues:)`。不过，对于一个序列中某个键可能存在多次的情况，就和上面一样，我们需要提供一个函数来对相同键对应的两个值进行合并。比如，要计算序列中某个元素出现的次数，我们可以对每个元素进行映射，将它们和 1 对应起来，然后从得到的 `(元素, 次数)` 的键值对序列中创建字典。如果我们遇到相同键下的两个值（也就是说，我们看到了同样地元素若干次），我们只需要将次数用 + 累加起来就行了：

```
extension Sequence where Element: Hashable {
    var frequencies: [Element:Int] {
        let frequencyPairs = self.map { ($0, 1) }
        return Dictionary(frequencyPairs, uniquingKeysWith: +)
    }
}
```

```
    }
}

let frequencies = "hello".frequencies // ["h": 1, "e": 1, "l": 2, "o": 1]
frequencies.filter { $0.value > 1 } // ["l": 2]
```

另一个有用的方法是对字典的值做映射。因为 Dictionary 是一个实现了 Sequence 的类型，所以它已经有一个 map 方法来产生数组。不过我们有时候想要保持字典的结构，只对其中的值进行变换。mapValues 方法就是做这件事的：

```
let settingsAsStrings = settings.mapValues { setting -> String in
    switch setting {
        case .text(let text): return text
        case .int(let number): return String(number)
        case .bool(let value): return String(value)
    }
}
settingsAsStrings // ["Name": "Jane's iPhone", "Airplane Mode": "false"]
```

## 键的 Hashable 要求

字典其实是哈希表。字典通过键的哈希值来为每个键在其底层作为存储的数组上指定一个位置。这也就是 Dictionary 要求它的 Key 类型需要遵守 Hashable 协议的原因。标准库中所有的基本数据类型都是遵守 Hashable 协议的，它们包括字符串，整数，浮点数以及布尔值。另外，像是数组，集合和可选值这些类型，如果它们的元素都是可哈希的，那么它们自动成为可哈希的。

为了保证性能，哈希表要求存储在其中的类型提供一个良好的哈希函数，也就是说这个函数不会产生太多的冲突。实现一个在整个整数范围内均匀分布其输入的良好的哈希函数不容易。不过幸运的是我们几乎不需要自己实现这个函数。在很多情况下，编译器可以生成 Hashable 的实现，即使它不适用于某个特定类型，标准库也带有让自定义类型可以挂钩的内置哈希函数。

对于结构体和枚举，只要这些类型是由可哈希的类型组成的，那么 Swift 就可以帮助我们自动合成 Hashable 协议所需要的实现。如果一个结构体的所有存储属性都是可哈希的，那么我们不用手动实现 Hashable 协议，结构体就已经实现这个协议了。类似的，只要枚举包含可哈希的

关联值，那么就可以自动实现这个协议；对于那些没有关联值的枚举，甚至都不用显式声明要实现 `Hashable` 协议。这不仅可以节省一开始实现的工作量，还可以在添加或者删除属性时自动更新实现。

如果你不能利用自动 `Hashable` 合成（要么因为你正在实现一个类；要么出于哈希的目的，在你自定义结构体中有几个存储属性需要被忽略），那么首先需要让类型实现 `Equatable` 协议，然后你可以实现 `hash(into:)` 方法来满足 `Hashable` 协议。这个方法接受一个 `Hasher` 类型参数，这个类型封装了一个通用的哈希函数，并在使用者向其提供数据时，捕获哈希函数的状态。该参数有一个接受任何可哈希值的 `combine` 方法。实质上构成类型的那些属性是关键组件，你通常需要排除那些可以重新延时创建或者对用户来说不可见的那些暂态属性。比如说，`Array` 会在内部存储其缓冲区的可用容量，这个值代表了在重新申请新的内存之前数组可以容纳的最大元素数量。但是如果两个数组只有最大容量不同的话，是不应该被认为是不同数组的：容量在比较 `Array` 类型时并不是关键组件。

你应该通过调用 `combine` 方法的方式将类型的所有基本组件逐个传递给 `hasher`。基本组件是那些构成类型实质的属性，你通常会想要排除那些可以被惰性重建的临时属性。

你应该使用相同的基本组件来进行相等性检查，因为必须遵守以下的不变原则：两个同样的实例（由你实现的 `==` 定义相同），必须拥有同样的哈希值。不过反过来不必为真：两个相同哈希值的实例不一定需要相等。不同的哈希值的数量是有限的，然而很多可以被哈希的类型（比如字符串）的个数是无穷的。

标准库的通用哈希函数使用一个随机种子作为其输入之一。也就是说，字符串 "abc" 的哈希值在每次程序执行时都会是不同的。随机种子是一种用来防止有针对性的哈希洪泛式拒绝服务攻击的安全措施。因为字典和集合是按照存储在哈希表中的顺序来迭代它们的元素，并且由于这个顺序是由哈希值决定的，所以这意味着相同的代码在每次执行时会产生不同的迭代顺序。如果你需要哈希值每次都一样，例如为了测试，那么可以通过设置环境变量 `SWIFT_DETERMINISTIC_HASHING=1` 来禁用随机种子，但是你不应该在正式环境中这么做。

最后，当你使用不具有值语义的类型（比如可变的对象）作为字典的键时，需要特别小心。如果你在将一个对象用作字典键后，改变了它的内容，它的哈希值和/或相等特性往往也会发生改变。这时候你将无法再在字典中找到它。这时字典会在错误的位置存储对象，这将导致字典内

部存储的错误。对于值类型来说，因为字典中的键不会和复制的值共用存储，因此它也不会被从外部改变，所以不存在这个问题。

## Set

标准库中第三种主要的集合类型是集合 Set (虽然听起来有些别扭)。集合是一组无序的元素，每个元素只会出现一次。你可以将集合想像为一个只存储了键而没有存储值的字典。和 Dictionary 一样，Set 也是通过哈希表实现的，并拥有类似的性能特性和要求。测试集合中是否包含某个元素是一个常数时间的操作，和字典中的键一样，集合中的元素也必须满足 `Hashable`。

如果你需要高效地测试某个元素是否存在于序列中并且元素的顺序不重要时，使用集合是更好的选择 (同样的操作在数组中的复杂度是  $O(n)$ )。另外，当你需要保证序列中不出现重复元素时，也可以使用集合。

Set 遵守 `ExpressibleByArrayLiteral` 协议，也就是说，我们可以用数组字面量的方式初始化一个集合：

```
let naturals: Set = [1, 2, 3, 2]
naturals // [1, 3, 2]
naturals.contains(3) // true
naturals.contains(0) // false
```

注意数字 2 在集合中只出现了一次，重复的数字并没有被插入到集合中。

和其他集合类型一样，集合也支持我们已经见过的基本操作：你可以用 `for` 循环进行迭代，对它进行 `map` 或 `filter` 操作，或者做其他各种事情。

## 集合代数

正如其名，集合 Set 和数学概念上的集合有着紧密关系；它支持你在数学课中学到的基本集合操作。比如，我们可以在一个集合中求另一个集合的补集：

```
let iPods: Set = ["iPod touch", "iPod nano", "iPod mini",
```

```
"iPod shuffle", "iPod Classic"]  
let discontinuedIPods: Set = ["iPod mini", "iPod Classic",  
    "iPod nano", "iPod shuffle"]  
let currentIPods = iPods.subtracting(discontinuedIPods) // ["iPod touch"]
```

我们也可以求两个集合的交集，换言之，找出两个集合中都含有的元素：

```
let touchscreen: Set = ["iPhone", "iPad", "iPod touch", "iPod nano"]  
let iPodsWithTouch = iPods.intersection(touchscreen)  
// ["iPod nano", "iPod touch"]
```

或者，我们能求两个集合的并集，也就是将两个集合合并为一个（当然，移除那些重复多余的）：

```
var discontinued: Set = ["iBook", "Powerbook", "Power Mac"]  
discontinued.formUnion(discontinuedIPods)  
discontinued  
/*  
["iPod nano", "Power Mac", "iPod Classic", "iPod shuffle",  
"Powerbook", "iPod mini", "iBook"]  
*/
```

这里我们使用了可变版本的 `formUnion` 来改变原来的集合（正因如此，我们需要将原来的集合用 `var` 声明）。几乎所有的集合操作都有不可变版本以及可变版本的形式，后一种都以 `form` 开头。想要了解更多的集合操作，可以看看 `SetAlgebra` 协议。

## 在闭包中使用集合

就算不暴露给函数的调用者，字典和集合在函数中也会是非常好用的数据结构。我们如果想要为 `Sequence` 写一个扩展，来获取序列中所有的唯一元素，我们只需要将这些元素放到一个 `Set` 里，然后返回这个集合的内容就行了。不过，因为集合并没有定义顺序，所以这么做是不稳定的，输入的元素的顺序在结果中可能会不一致。为了解决这个问题，我们可以创建一个扩展来解决这个问题，在扩展方法内部我们还是使用 `Set` 来验证唯一性：

```
extension Sequence where Element: Hashable {
```

```
func unique() -> [Element] {
    var seen: Set<Element> = []
    return filter { element in
        if seen.contains(element) {
            return false
        } else {
            seen.insert(element)
            return true
        }
    }
}
```

```
[1,2,3,12,1,3,4,5,6,4,6].unique() // [1, 2, 3, 12, 4, 5, 6]
```

上面这个方法让我们可以找到序列中的所有不重复的元素，并且通过元素必须满足 `Hashable` 这个约束来维持它们原来的顺序。在我们传递给 `filter` 的闭包中，我们使用了一个外部的 `seen` 变量，我们可以在闭包里访问和修改它的值。我们会在函数一章中详细讨论它的细节。

## Range

范围代表的是两个值的区间，它由上下边界进行定义。你可以通过 `.. 来创建一个不包含上边界 的半开范围，或者使用 ... 创建同时包含上下边界的闭合范围：`

```
// 0 到 9, 不包含 10
let singleDigitNumbers = 0..10
Array(singleDigitNumbers) // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
// 包含 "z"
let lowercaseLetters = Character("a")...Character("z")
```

这些操作符还有一些前缀和后缀的变型版本，用来表示单边的范围：

```
let fromZero = 0...
```

```
let upToZ = ..<Character("z")
```

一共有五种不同的具体类型可以用来表示范围，每种类型都代表了对值的不同的约束。最常用的两种类型是 Range (由 ..< 创建的半开范围) 和 ClosedRange (由 ... 创建的闭合范围)。两者都有一个 Bound 的泛型参数：对于 Bound 的唯一的要求是它必须遵守 Comparable 协议。举例来说，上面的 lowercaseLetters 表达式的类型是 ClosedRange<Character>。

对范围最基本的操作是检测它是否包含了某些元素：

```
singleDigitNumbers.contains(9) // true  
lowercaseLetters.overlaps("c" ..<"f") // true
```

半开范围和闭合范围各有所用：

- 只有半开范围能表达空间隔 (也就是下界和上界相等的情况，比如 5..<5)。
- 只有闭合范围能包括其元素类型所能表达的最大值 (比如 0...Int.max)。而半开范围则要求范围上界是一个比自身所包含的最大值还要大 1 的值。

## 可数范围

范围看起来很自然地会是一个序列或者集合类型。并且你确实可以遍历一个整数范围，或像集合类型那样对待它：

```
for i in 0..<10 {  
    print("\(i)", terminator: " ")  
} // 0 1 2 3 4 5 6 7 8 9
```

```
singleDigitNumbers.last // Optional(9)
```

但并不是所有的范围都可以使用这种方式。比如，编译器不允许我们遍历一个 Character 的范围：

```
// 错误: 'Character' 类型没有实现 'Strideable' 协议。
```

```
for c in lowercaseLetters {
```

```
...  
}
```

(不能直接遍历字符范围的原因是和 Unicode 有关。我们会在字符串这章中讨论这个问题)

这是怎么回事呢？让 Range 满足集合类型协议是有条件的，条件是它的元素需要满足 Strideable 协议（你可以通过增加偏移来从一个元素移动到另一个），并且步长 (stride step) 是整数：

```
extension Range<Sequence>  
    where Bound : Strideable, Bound.Stride : SignedInteger { /* ... */ }
```

```
extension Range<Collection, BidirectionalCollection,  
    RandomAccessCollection>  
    where Bound : Strideable, Bound.Stride : SignedInteger { /* ... */ }
```

(我们会在集合类型协议中详细讨论 Sequence, Collection, BidirectionalCollection 和 RandomAccessCollection 这些协议)

换句话说，为了能遍历范围，一个 range 必须是可数的。对于可数范围（满足了那些约束），因为对于 Stride 有整数类型这样一个约束，所以有效的边界包括整数和指针类型，但不能是浮点数类型。如果你想要对连续的浮点数值进行迭代的话，你可以通过使用 stride(from:to:by) 和 stride(from:through:by) 方法来创建序列用以迭代。

在条件化实现协议被 Swift 4.1 和 4.2 引入之前，标准库为了区分可数和不可数范围，包含了两个具体类型，分别名为 CountableRange 和 CountableClosedRange。为了向后兼容，这两个名字还是作为类型别名而存在着。就像标准库中注释所说，你也可以把它们作为对于冗长的范围加上一堆约束的声明的一种简写：

```
// 注意：这不仅仅是为了兼容性，它还是有用的简写。  
public typealias CountableRange<Bound : Strideable> = Range<Bound>  
where Bound.Stride : SignedInteger
```

## 部分范围

部分范围指的是那些以 ... 或 ..< 作为前置或后置操作符所构建的范围。之所以把这类范围称为部分范围，是因为它们缺少一个边界。比如说，0... 描述了一个由 0 开始，但是没有上界的范围。部分范围有三种类型：

```
let fromA: PartialRangeFrom<Character> = Character("a")...
let throughZ: PartialRangeThrough<Character> = ...Character("z")
let upto10: PartialRangeUpTo<Int> = ..<10
```

和 CountableRange 是带有 Strideable 类型的 Range 类型别名类似，CountablePartialRangeFrom 也是 PartialRangeFrom 的类型别名，只不过它的约束更加严格。

当我们对一个可数的 PartialRangeFrom 进行迭代时，它会从 lowerBound 开始，并重复调用 advanced(by: 1)。如果你在 for 循环里使用这样一个范围，你必须自己添加 break 条件，否则你会得到一个无限循环 (或者在计数器溢出时崩溃)。对 PartialRangeThrough 和 PartialRangeUpTo 来说，由于缺少下界，不管它们元素类型是否可以步进，它们都是不能进行迭代的。

## 范围表达式

所有这五种范围都满足 RangeExpression 协议。这个协议内容很简单，所以我们可以将它列举到书里。首先，它允许我们询问某个元素是否被包括在该范围内。其次，给定一个集合类型，它能够计算出表达式所指定的完整的 Range：

```
public protocol RangeExpression {
    associatedtype Bound: Comparable
    func contains(_ element: Bound) -> Bool
    func relative<C>(to collection: C) -> Range<Bound>
    where C: Collection, Self.Bound == C.Index
}
```

对于下界缺失的部分范围，`relative(to:)` 方法会把集合类型的 `startIndex` 作为范围下界。对于上界缺失的部分范围，同样，它会使用 `endIndex` 作为上界。这样一来，部分范围就能使集合切片的语法变得相当紧凑：

```
let numbers = [1,2,3,4]
numbers[2...] // [3, 4]
numbers[..<1] // [1]
numbers[1...2] // [2, 3]
```

这种写法能够正常工作，是因为 Collection 协议里对应的下标操作符声明中，所接收的是一个实现了 RangeExpression 的类型，而不是上述五个具体的范围类型中的某一个。你甚至还可以将两个边界都省略掉，这样将会得到表示整个集合的一个切片：

```
numbers[...] // [1, 2, 3, 4]
type(of: numbers[...]) // ArraySlice<Int>
```

(这其实是 Swift 标准库中的一个特殊实现，这种无界范围还不是有效的 RangeExpression 类型，不过它应该会在今后遵守 RangeExpression 协议。)

如果可能的话，尝试复制标准库的方案，然后让你自己的函数接受一个 RangeExpression 类型的参数，而不是一个具体的范围类型。因为这个协议不允许访问范围的界限，所以除非你在一个集合类型的上下文中，否则这个方案并不总是可行的。但是如果可以的话，你将为你 API 的使用者提供更大的自由来让他们传递任何种类的范围表达式。

## RangeSet

RangeSet 是包含了拥有相同元素类型的一组 range。它的主要用途，是让处理非连续的集合索引这项工作变得简单和高效。你当然可以使用 `Set<Index>` 来完成这项任务，但是 RangeSet 在存储上更有效率，因为它可以把相邻的值合并成一个单一的范围。比如说你有一个 1,000 个元素的 table view，现在你想用一个集合来管理用户选中的各行。根据被选中行的数量，`Set<Int>` 最多会需要存储 1,000 个元素。但如果 RangeSet，它会存储连续的范围，如果列表中的前 500 行被选中，那么只需要存储两个整数 (选中范围的下界和上界) 就可以了。

RangeSet 上有一个 `ranges` 属性，它为 RangeSet 所表示的范围暴露出一个集合类型的接口：

```
var indices = RangeSet(1..<5)
indices.insert(contentsOf: 11..<15)
/*show*/ Array(indices.ranges)
```

不管范围是按照什么顺序插入到 RangeSet 中的，ranges 都会按照升序返回它们，这些范围绝不会重叠或者毗邻。如果你需要的是独立的各个元素组成的集合，可以使用 flatMap：

```
/*show*/ Array(indices.ranges.flatMap { $0 })
/*show*/ let evenIndices = indices.ranges
    .flatMap { $0 }
    .filter { $0 % 2 == 0 }
```

RangeSet 并不遵守 SetAlgebra 协议，但是它还是实现了 SetAlgebra 的子集，以方便进行合集或者交集这样的操作。

目前，RangeSet 类型还不是标准库的一部分，但是它已经作为提案 [SE-0270](#) 通过了 Swift Evolution 流程。它现在是标准库预览包的一部分，这个包是新的标准库功能的“实验室”，它允许开发者进行更快地使用这些新功能，并在现实世界中测试它们，同时保留引入破坏性变更的可能。

## 回顾

在本章中，我们讨论了一系列不同的集合类型：Array、Dictionary、Set 和 Range 等。我们同时研究了这些集合所拥有的一系列操作方法，以及如何通过组合这些操作创建强力算法的方式。我们看到，Swift 内建的集合类型允许你使用 let 和 var 来控制集合的可变性。另外，我们也对各种不同的 Range 类型进行了介绍。

和其他语言相比，Swift 的标准库所提供的通用的集合类型是相对较少的。如果你想要某种数据结构，可以去看看 [Swift Collections](#) 包，在那边标准库的成员们和其余 Swift 社区的开发者为常见的数据结构创建了一系列优质的实现，比如双端队列或者有序集合和有序字典等。

字符串也是集合类型。我们在后面有单独的一章讨论字符串，所以在这里我们没有涉及这个话题。

我们将在集合类型协议中重新讨论本章的主题，在那里，我们将深入讨论那些在 Swift 内建集合类型上的协议。

# 可选值

3

# 哨岗值

在编程世界中有一种非常通用的模式，那就是某个操作是否要返回一个有效值。

当你在读取文件并读到文件末尾时，也许期望的是不返回值，就像下面的 C 代码这样：

```
int ch;
while ((ch = getchar()) != EOF) {
    printf("Read character %c\n", ch);
}
printf("Reached end-of-file\n");
```

EOF 只是对于 -1 的一个 #define。如果文件中还有其他字符，getchar 将会返回它们。一旦到达文件末尾，getchar 将返回 -1。

又或者，返回空值意味着“未找到”，就像下面这段 C++ 代码一样：

```
auto vec = {1, 2, 3};
auto iterator = std::find(vec.begin(), vec.end(), someValue);
if (iterator != vec.end()) {
    std::cout << "vec contains " << *iterator << std::endl;
}
```

在这里，vec.end() 返回的迭代器表示容器最后一个元素的下一个位置。这是一个特殊的迭代器，你可以用它来检查容器末尾，但是和 Swift 集合类型中的 endIndex 类似，你不能实际用它来获取一个值。find 使用它来表达容器中没有这样的值。

再或者，是因为函数处理过程中发生了某些错误，而导致没有值能被返回。其中，最臭名昭著的例子大概就是空指针了。下面这句看起来人畜无害的 Java 代码就将抛出一个 NullPointerException：

```
int i = Integer.getInteger("123")
```

因为实际上 Integer.getInteger 做的事情并不是将字符串解析为整数，它实际上会去尝试获取一个叫做“123”的系统属性的整数值。因为系统中并不存在这样的属性，所以 getInteger 返回的是 null。当 null 被拆箱成一个 int 时，Java 将抛出异常。

这里还有一个 Objective-C 的例子：

```
[[NSString alloc] initWithContentsOfURL:url  
encoding:NSUTF8StringEncoding error:&error];
```

这个初始化方法有可能返回 nil，只有在这种情况下，你才应该去检查 error 指针。如果返回的是非 nil 的话，error 并不一定是个有效的指针。

在上面所有例子中，这些函数都返回了一个“魔法”数来表示其并没有返回真实的值。这样的值被称为“哨岗值 (sentinel values)”。

不过这种策略是有问题的。因为返回的结果不管从哪个角度看都很像一个真实值。-1 也是一个有效的整数，但你却不想将它打印出来。v.end() 也是一个迭代器，但你读取这个位置的值时，结果却是未定义的。另外，当你的 Java 程序抛出一个 NullPointerException 时，所有人都想看的是栈转储信息 (stack dump)。

和 Java 不同，Objective-C 允许我们向 nil 发送消息。这种行为是“安全”的，因为 Objective-C 的运行时会保证向 nil 发送消息时，返回值总是等于 0。也就是说，根据消息的返回值是对象或数值，运行时会分别使用 nil 和 0 表示，以此类推。如果消息返回的是一个结构体，那么它的所有属性都将被初始化为零。记住这一点后，让我们来看看下面这个查找子字符串的例子：

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

如果 someString 是 nil，那么 rangeOfString: 消息将返回一个值都为零的 NSRange。也就是说，.location 将为零，而 NSNotFound 被定义为 NSIntegerMax。这样一来，当 someString 是 nil 时，if 语句的内容将被执行，而其实这并不应该发生。

Tony Hoare 在 1965 年设计了 null 引用，他对此设计表示痛心疾首，并将这个问题称为“价值十亿美元的错误”：

那时候，我正在为一门面向对象语言 (ALGOL W) 设计第一个全面的引用类型系统。我的目标是在编译器自动执行的检查的保证下，确保对于引用的所有使用都是安全的。但是我没能抵挡住引入 null 引用的诱惑，因为它太容易实现了。这导致了不计其数的错误，漏洞以及系统崩溃。这个问题可能在过去四十年里造成了有十亿美元的损失。

哨岗值的另一个问题是想要正确使用它们还需要一些前置知识。比如像是 C++ 的 end 迭代器，或 Objective-C 中错误处理这样约定俗成的用法。如果没有这些约定，或你不知道它们，那你只能依赖文档进行开发了。另外，一个函数也没有办法来表明自己不会失败。也就是说，当一个函数返回指针时，这个指针有可能绝对不会是 nil。但是除了阅读文档之外，你并没有办法能知道这个事实。更甚者有可能文档本身就是错的。

## 通过枚举解决魔法数的问题

当然，每个有经验的程序员都知道使用魔法数并不好。大多数语言都支持某种形式的枚举类型，用它表达某个类型可能包含的所有值是一种更为安全的做法。

Swift 更进一步，它在枚举中引入了“关联值”的概念。也就是说，枚举可以在它们的成员中包含另外的关联的值。我们会在枚举这一章中探究它的细节。现在，只要知道 Optional 也是通过枚举实现的就好了：

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

获取枚举关联值的唯一方法是通过模式匹配，就像在 switch 或 if case let 中使用的匹配方法一样。将“缺失的值”的状态编码到类型系统中，可以让 API 调用者一眼就看出他们是否必须要处理这个 case，这让代码更具有表达力。而且，和哨岗值不同，除非你显式地检查并解包，否则是不可能意外地获取到一个 Optional 包装值的。

因此，Swift 中与 find 等效的方法 firstIndex(of:) 返回的不是一个哨岗值，而是一个 Optional<Index>。以下是一个类似的实现：

```
extension Collection where Element: Equatable {  
    func firstIndex(of element: Element) -> Optional<Index> {  
        var idx = startIndex  
        while idx != endIndex {  
            if self[idx] == element {  
                return .some(idx)  
            }  
            formIndex(after: &idx)  
        }  
        // 没有找到, 返回 .none  
        return .none  
    }  
}
```

因为可选值是 Swift 中非常重要和基础的类型，所以有很多让它用起来更简洁的语法：Optional<Index> 可以写成 Index?；可选值遵守 ExpressibleByNilLiteral 协议，因此你可以用 nil 替代 .none；像上面 idx 这样的非可选值将在需要的时候自动“升级”为可选值，这样你就可以直接写 return idx，而不用 return .some(idx) 了。

这个语法糖实际上掩盖了 Optional 类型的真正本质。请时刻牢记，可选值并不是什么魔法，它就是一个普通的枚举值。即便 Swift 没有提供它，你也完全可以自己定义一个。

现在，用户就不会在没有检查的情况下，错误地使用一个值了：

```
var array = ["one", "two", "three"]  
let idx = array.firstIndex(of: "four") // 返回 Optional<Int>。  
// 编译错误: remove(at:) 接受 Int, 而不是 Optional<Int>。  
array.remove(at: idx)
```

相反，假设得到的结果不是 .none，为了使用包装在可选值中的索引，你必须对其进行“解包”：

```
var array = ["one", "two", "three"]
```

```
switch array.firstIndex(of: "four") {  
    case .some(let idx):  
        array.remove(at: idx)  
    case .none:  
        break // 什么都不做  
}
```

在这个 `switch` 语句中我们使用了匹配普通枚举的语法来处理可选值，例如，提取 `.some` 关联值的用法。这种做法非常安全，但是读写起来都不是很顺畅。一种更简明的写法是使用 `?`  作为在 `switch` 中对 `some` 进行匹配时的模式后缀，另外，你还可以使用 `nil` 字面量来匹配 `none`：

```
switch array.firstIndex(of: "four") {  
    case let idx?:  
        array.remove(at: idx)  
    case nil:  
        break // 什么都不做  
}
```

但是这还是有点太笨重。我们接下来会看看其他一些简短而又清晰的处理可选值的方式，你可以根据你的使用情景酌情选择。

## 可选值概览

可选值在 Swift 中有很多来自语言内建的支持。如果你已经使用了 Swift 一段时间，下面这些例子看起来可能会很简单，但是请务必确认你已经理解了它们要表达的概念，因为我们在整本书中不断地使用它们。

### if let

使用 `if let` 来进行可选值绑定 (optional binding) 要比上面使用 `switch` 语句要稍好一些。`if let` 语句会检查可选值是否为 `nil`，如果不是 `nil`，便会解包可选值。`idx` 的类型就是 `Int` (而不再是可选值)，并且 `idx` 也只在这个 `if let` 语句的作用域中有效：

```
var array = ["one", "two", "three", "four"]
if let idx = array.firstIndex(of: "four") {
    array.remove(at: idx)
}
```

你可以把布尔限定语句与 if let 搭配在一起使用。因此，如果要实现“查找到的位置是数组的第一个元素，就不删除它”这样的逻辑，可以这样：

```
if let idx = array.firstIndex(of: "four"), idx != array.startIndex {
    array.remove(at: idx)
}
```

你也可以在同一个 if 语句中绑定多个值。更赞的是，在后面的绑定中可以使用之前成功解包出来的结果。当你要连续调用多个返回可选值的函数时，这个功能就特别有用了。比如，下面的 URL 和 UIImage 的构造方法都是“可失败的 (failable)”，也就是说，要是你的 URL 是无效的，或者数据不是一个图片数据，这些方法都会返回 nil。而 Data 的初始化方法会抛出错误，你可以通过 try? 来把它转变为一个可选值。它们三者的调用可以通过这样的方式串联起来：

```
let urlString = "https://www.objc.io/logo.png"
if let url = URL(string: urlString),
    let data = try? Data(contentsOf: url),
    let image = UIImage(data: data)
{
    let view = UIImageView(image: image)
    PlaygroundPage.current.liveView = view
}
```

同步的 Data(contentsOfURL:) 初始化方法会在下载期间阻塞它所在的线程。对于快速举例来说还好，但是不推荐把它用在产品代码里。你应该始终选择异步的网络 API，比如 URLSession.data(from:delegate:)。

多个 let 的任意部分也能拥有布尔值限定的语句：

```
if let url = URL(string: urlString), url.pathExtension == "png",
    let data = try? Data(contentsOf: url),
    let image = UIImage(data: data)
{
    let view = UIImageView(image: image)
}
```

最后，你可以在同一个 if 中将可选值绑定，布尔语句和 case let 用任意的方式组合在一起使用。

## while let

while let 语句和 if let 非常相似，它表示当一个条件返回 nil 时便终止循环。

标准库中的 `readLine` 函数从标准输入中读取内容，并返回一个可选字符串。当到达输入末尾时，这个函数将返回 nil。所以，你可以使用 while let 来实现一个非常基础的和 Unix 中 `cat` 命令等价的功能：

```
while let line = readLine() {
    print(line)
}
```

和 if let 一样，你可以在可选绑定后面添加一个布尔值语句。如果你想在遇到 EOF 或者空行的时候终止循环的话，只需要加一个判断空字符串的语句就行了。要注意，一旦条件为 false，循环就会停止（也许你错误地认为 where 条件会像 filter 那样工作，其实不然）。

```
while let line = readLine(), !line.isEmpty {
    print(line)
}
```

我们会在集合类型协议一章中看到，`for x in seq` 这样的循环语句需要 `seq` 遵守 Sequence 协议。该协议提供 `makeIterator` 方法来创建迭代器，而迭代器中的 `next` 方法将不断返回序列中的值，并在序列中的值被耗尽的时候，返回 nil。while let 非常适合用在这个场景中：

```
let array = [1, 2, 3]
```

```
var iterator = array.makeIterator()
while let i = iterator.next() {
    print(i, terminator: " ")
} // 1 2 3
```

因为一个 for 循环其实就是 while 循环，这样一来，for 循环也支持布尔语句就是情理之中的事了，只是，我们要在布尔语句之前，使用 where 关键字：

```
for i in 0..<10 where i % 2 == 0 {
    print(i, terminator: " ")
} // 0 2 4 6 8
```

注意上面的 where 语句和 while 循环中的布尔语句工作方式有所不同。在 while 循环中，一旦值为 false 时，迭代就将停止。而在 for 循环里，它的工作方式就和 filter 相似了。如果我们将上面的 for 循环用 while 重写的话，看起来是这样的：

```
var iterator2 = (0..<10).makeIterator()
while let i = iterator2.next() {
    guard i % 2 == 0 else { continue }
    print(i)
}
```

## 双重可选值

说到这，是时候来看看一个可选值的包装类型也是一个可选值的情况了，这会导致可选值的嵌套。不过，这既不是一个奇怪的边界情况，编译器也不应该自动合并嵌套的可选值类型。为了了解这种应用场景，假设有一个表示数字的字符串数组，为了把它转换成整数数组，你可能会使用 map 进行转换：

```
let stringNumbers = ["1", "2", "three"]
let maybeInts = stringNumbers.map { Int($0) } // [Optional(1), Optional(2), nil]
```

你现在得到了一个元素类型为 `Optional<Int>` (也就是 `Int?`) 的数组，这是因为 `Int.init(String)` 是可能失败的，只要字符串无法转换成整数。我们的例子中，最后一个元素就将是 `nil`，因为字符串 "three" 无法转换成一个整数。

于是，当使用 `for` 遍历 `maybeInts` 的时候，自然访问到的每个元素都是 `Int?` 了：

```
for maybeInt in maybeInts {  
    // maybeInt 是一个 Int? 值  
    // 得到两个整数值和一个 `nil`  
}
```

我们已经知道 `for...in` 是 `while` 循环加上一个迭代器的简写方式。由于 `next` 方法会把序列中的每个元素包装成可选值，所以 `iterator.next()` 函数返回的其实是一个 `Optional<Optional<Int>>` 值，或者说是一个 `Int??`。而 `while let` 会解包并检查这个值是不是 `nil`，如果不是，则绑定解包的值并运行循环体部分：

```
var iterator = maybeInts.makeIterator()  
while let maybeInt = iterator.next() {  
    print(maybeInt, terminator: " ")  
}  
// Optional(1) Optional(2) nil
```

当循环到达最后一个值，也就是从 "three" 转换而来的 `nil` 时，从 `next` 返回的其实是一个非 `nil` 的值，这个值是 `.some(nil)`。`while let` 将这个值解包，并将解包结果 (也就是 `nil`) 绑定到 `maybeInt` 上。如果没有嵌套可选值的话，这个操作将无法完成。

顺带一提，如果你只想对非 `nil` 的值做 `for` 循环的话，可以使用 `case` 来进行模式匹配：

```
for case let i? in maybeInts {  
    // i 将是 Int 值，而不是 Int?  
    print(i, terminator: " ")  
}  
// 1 2
```

```
// 或者只对 nil 值进行循环
for case nil in maybeInts {
    // 将对每个 nil 执行一次
    print("No value")
}
// No value
```

这里使用了 `x?` 这个模式，它只会匹配那些非 `nil` 的值。这个语法是 `.some(x)` 的简写形式，所以该循环还可以被写为：

```
for case let .some(i) in maybeInts {
    print(i)
}
```

基于 `case` 的模式匹配可以让我们把在 `switch` 的匹配中用到的规则同样地应用到 `if`、`for` 和 `while` 上去。最有用的场景是结合可选值，但是也有一些其他的使用方式，比如：

```
let j = 5
if case 0..<10 = j {
    print("\u{1f60a}j 在范围内")
} // 5 在范围内
```

我们会在枚举这一章中详细讨论模式匹配。

## if var and while var

除了 `let` 以外，你还可以使用 `var` 来搭配 `if`、`while` 和 `for`。这让你可以在语句块中改变变量：

```
let number = "1"
if var i = Int(number) {
    i += 1
    print(i)
} // 2
```

不过注意，`i` 会是一个本地的复制。任何对 `i` 的改变将不会影响到原来的可选值。可选值是值类型，解包一个可选值做的事情是将它里面的值复制出来。

## 解包后可选值的作用域

有时候只能在 `if` 块的内部访问被解包的变量确实让人感到是一种限制。举个例子，数组有个 `first` 方法，它会返回数组首个元素的可选值，如果数组为空的话，则返回 `nil`。它是下面这段代码的简写：

```
let array = [1,2,3]
if !array.isEmpty {
    print(array[0])
}
// if 块的外部，编译器无法保证 a[0] 的有效性
```

推荐使用 `first` 属性，因为如果想要使用它的话，你必须先解包，你不会因为不小心忘了这件事而造成问题：

```
if let firstElement = array.first {
    print(firstElement)
}
// if 块的外部，不能使用 firstElement
```

解包后的值只能在 `if let` 代码块中使用，这在绝大多数情况下都很好。但当 `if` 语句的目的是在某些条件不满足时提前退出函数的话，这个特性就不太实用了。提前退出 (early exit) 可以帮助我们在这个函数稍后的部分避免嵌套或者重复的检查。例如，你可能会编写下面这样的代码：

```
func doStuff(withArray a: [Int]) {
    if a.isEmpty {
        return
    }
    // 现在就可以安全地使用 a[0] 或 a.first! 了
}
```

这里，`if let` 无法继续帮助我们实现期望的功能了，因为在 `if` 语句块之后，绑定的值就离开它的作用域了。不过，你还是可以确定数组中至少包含一个元素，所以即使语法上看上去有些恼人，但对第一个元素强制解包还是安全的。

为了在作用域外使用解包后的可选值，一个可行的方案是利用 Swift 的延迟初始化 (deferred initialization) 的能力。看看下面这个例子，它重新实现了 `URL` 和 `NSString` 的 `pathExtension` 属性的一部分功能：

```
extension String {
    var fileExtension: String? {
        let period: String.Index
        if let idx = lastIndex(of: ".") {
            period = idx
        } else {
            return nil
        }
        let extensionStart = index(after: period)
        return String(self[extensionStart...])
    }
}

"hello.txt".fileExtension // Optional("txt")
```

编译器会检查并确保你的代码只有两条执行路径：一条是没有找到 `"."` 就提早返回；另一条则是 `period` 被正确初始化。因此，在 `if` 语句之后，`period` 既不可能为 `nil` (因为 `period` 就不是一个可选值)，也不可能未被初始化 (Swift 不允许你使用未初始化的变量)，你也就完全可以不用在代码中考虑可选值的问题了。

但是，上面这两个例子看起来都很丑。我们在这里真正需要的其实只是一个类似 `if not let` 的语句，而这正是 `guard let` 所做的事情。

```
func doStuff(withArray a: [Int]) {
    guard let firstElement = a.first else {
        return
```

```
}

// firstElement 在这里已经被解包了

}
```

第二个例子也变得清晰得多了：

```
extension String {

    var fileExtension: String? {
        guard let period = lastIndex(of: ".") else {
            return nil
        }

        let extensionStart = index(after: period)
        return String(self[extensionStart...])
    }
}
```

在 `guard` 的 `else` 代码块中，你可以执行任意代码，例如，在其中包含一个完整的 `if...else` 语句也没问题。它唯一的要求是必须离开当前的作用域，通常这意味着一条 `return` 语句，或抛出一个错误，亦或调用 `fatalError` (或者其他返回 `Never` 的方法)。如果你是在循环中使用 `guard` 的话，那么最后也可以是 `break` 或者 `continue`。

一个返回 `Never` 的函数用于通知编译器：它绝对不会返回。有两类常见的函数会这么做：一种是像 `fatalError` 那样表示程序失败的函数，另一种是像 `dispatchMain` 那样运行在整个程序生命周期的函数。编译器会使用这个信息来检查和确保控制流正确。举例来说，`guard` 语句的 `else` 路径必须退出当前域或者调用一个不会返回的函数。

`Never` 又被叫做无人类型 (uninhabited type)。这种类型没有有效值，因此也不能够被构建。一个声明为返回无人类型的函数绝对不可能正常返回。

在 Swift 中，无人类型是通过一个不包含任意成员的 `enum` 实现的：

```
public enum Never { }
```

一般来说，你不会自己定义返回 Never 的方法，除非你在为 fatalError 或者 preconditionFailure 写封装。一个很有意思的应用场景是，当你要创建一个很复杂的 switch 语句，在逐条编写每个 case 的过程中，编译器就会用空的 case 语句或者是没有返回值这样的错误一直轰炸你，而你又想先集中精力处理某一个 case 语句的逻辑。这时，放几个 fatalError() 就能让编译器闭嘴。你还可以写一个 unimplemented() 方法，这样能够更好地表达这些调用是暂时没有实现的意思：

```
func unimplemented() -> Never {  
    fatalError("This code path is not implemented yet.")  
}
```

在泛型环境里，把 Never 和 Result 或者某些类似的类型结合在一起使用是非常常见的。例如，在像是 Apple 的 Combine 这样的响应式编程框架中，Combine 把事件流按照 Publisher 协议进行建模，它有 Output 和 Failure 两个关联类型。Output 描述了所发出事件的类型，而 Failure 则代表了错误情况的类型。一个文本框对象可能会提供 AnyPublisher<String, Never>，在用户每次编辑文本时，它发布一个事件。使用 Never 作为失败类型向程序员和编译器进行暗示，告诉他们这个 publisher 永远不会发送失败事件，因为 Never 值是永远无法被构建出来的。

Swift 在区分各种“无”类型上非常严密。除了 nil 和 Never，还有 Void，Void 是空元组 (tuple) 的另一种写法：

```
public typealias Void = ()
```

Void 或者 () 最常见的用法是作为那些不返回任何东西的函数的返回值，不过它也还有其他使用场景。举例来说，一个按钮对象的 publisher 会在用户点击按钮时发送一次事件，但是这个事件并不包含附加的其他内容，所以它的事件流类型将会 AnyPublisher<(), Never>。

正如 David Smith 所指出的，Swift 对“东西不存在”(nil)，“存在且为空”(Void) 以及“不可能发生”(Never) 这几个概念进行了仔细的区分。

当然，guard 并不局限于用在绑定上。guard 能够接受任何在普通的 if 语句中能接受的条件。比如上面的空数组的例子可以用 guard 重写为：

```
func doStuff2(withArray a: [Int]) {  
    guard !a.isEmpty else { return }  
    // 现在可以安全地使用 a[0] 或 a.first! 了  
}
```

和可选值绑定的情况不同，单单使用 `guard` 并没有太多好处。实际上它还要比原来的版本稍微啰嗦一些。不过用这种方式来提前退出还是有其可取之处的，比如有时候（但不是像我们的这个例子这样）使用反向的布尔条件会让事情更清楚一些。另外，在阅读代码时，`guard` 是一个明确的信号，它暗示我们“只在条件成立的情况下继续”。最后 Swift 编译器还会检查你是否确实在 `guard` 块中退出了当前作用域，如果没有的话，你会得到一个编译错误。因为可以得到编译器帮助，所以我们建议尽量选择使用 `guard`，即便 `if` 也可以正常工作。

## 可选链

在 Objective-C 中，对 `nil` 发消息什么都不会发生。Swift 里，我们可以通过“可选链 (optional chaining)”来达到同样的效果：

```
delegate?.callback()
```

但和 Objective-C 不同的是，Swift 编译器会强制要求你声明消息的接受者可能为 `nil`。这里的问号对代码的读者来说是一个清晰地信号，表示方法可能会不被调用。

当你通过调用可选链得到一个返回值时，这个返回值本身也会是可选值。看看下面的代码你就知道为什么需要这么设定了：

```
let str: String? = "Never say never"  
// 我们希望 upper 是大写的字符串  
  
let upper: String  
if str != nil {  
    upper = str!.uppercase()  
} else {  
    // 这里没有合理的处理方法  
    fatalError("no idea what to do now...")  
}
```

如果 str 不等于 nil, upper 就会有我们想要的值。但如果 str 等于 nil, upper 就没有办法设置一个值了。因此使用可选链的时候, 下面的 upper2 只能是可选值, 因为它需要考虑 str 可能为 nil 的情况:

```
let upper2 = str?.uppercased() // Optional("NEVER SAY NEVER")
```

正如同可选链名字所暗示的那样, 你可以将可选值的调用链接起来:

```
let lower = str?.uppercased()?.lowercased() // Optional("never say never")
```

这看起来有点出乎意料。我们不是刚刚才说过可选链调用的结果是一个可选值么? 所以为什么在 uppercased() 后面不需要加上问号呢? 这是因为可选链是一个“展平”操作。

str?.uppercased() 返回了一个可选值, 如果我们再对它调用 ?.lowercased() 的话, 逻辑上来说将得到一个可选值的可选值。不过其实我们想要得到的是一个普通的可选值, 所以我们在写链上第二个调用时不需要包含可选的问号, 因为可选的特性已经在之前就被捕获了。

另一方面, 如果 uppercased 方法本身也返回一个可选值的话, 我们就需要在它后面加上 ? 来表示我们正在链接这个可选值。比如, 让我们对 Int 类型进行扩展, 添加一个计算属性 half, 这个属性将把整数值除以二并返回结果。但是如果数字不够大的话, 比如当数字小于 2 时, 函数将返回 nil:

```
extension Int {  
    var half:Int? {  
        guard self < -1 || self > 1 else { return nil }  
        return self / 2  
    }  
}
```

因为调用 half 返回一个可选结果, 因此当我们重复调用它时, 需要一直添加问号。因为函数的每一步都有可能返回 nil:

```
20.half?.half?.half // Optional(2)
```

编译器非常聪明，它能为我们展平结果类型。上面的表达式的类型正是我们期待的 Int?，而不是 Int??。后一种类型可以给我们更多的信息，比如说可选链是在哪个部分解包失败的，但是这也会让结果非常难以处理，从而让可选链一开始时给我们带来的便利性损失殆尽。

到现在为止，我们已经看到了方法调用和属性获取时的可选链的用法。对于下标也同样适用，比如：

```
let dictOfArrays = ["nine": [0, 1, 2, 3]]  
dictOfArrays["nine"]?[3] // Optional(3)
```

另外，还有这样的情况：

```
let dictOfFunctions: [String: (Int, Int) -> Int] = [  
    "add":(+),  
    "subtract":(-)  
]  
dictOfFunctions["add"]?(1, 1) // Optional(2)
```

设想一个类在某个事件发生时，要通过调用存储在其中的回调函数来通知其所有者，上面的特性就会非常有用。比如有一个 TextField 类：

```
class TextField {  
    private(set) var text = ""  
    var didChange: ((String) -> ())?  
  
    // 被框架调用的事件处理方法。  
    func textDidChange(newText: String) {  
        text = newText  
        // 如果不是 nil 的话，触发回调。  
        didChange?(text)  
    }  
}
```

`didChange` 属性存储了一个回调函数，每当用户编辑文本时，这个函数都会被调用。因为文本框的所有者并不一定需要注册这个回调，所以该属性是可选值，它的初始值为 `nil`。当这个回调被调用的时候（在上面的 `textDidChange` 方法中），可选链的写法就非常简洁了。

如果你不赋值，可选值的 `var`（而非 `let`）会隐性地被初始化为 `nil`。Swift 要求严格的明确初始化，而这里是唯一一个例外。这里的根本原因在于便利性，而且可选值拥有“明显”的默认值。就算这样，Swift 团队里的部分成员还是对这个设计决定感到后悔，如果不是因为会破坏已有代码的话，他们更希望改变这个行为。说来很怪，这个隐式的初始化只对那些使用 `...?` 作为类型标记的变量有效。如果我们把上面写成 `var didChange: Optional<(String) -> ()>` 的话，编译器就会向我们抱怨说这个属性没有被初始化。

## 可选链和赋值

你还可以通过可选链来进行赋值。假设你有一个可选值变量，如果它不是 `nil` 的话，你想要更新它的一个属性：

```
struct Person {  
    var name: String  
    var age: Int  
}  
  
var optionalLisa: Person? = Person(name: "Lisa Simpson", age: 8)  
// 如果不是 nil，则增加 age  
if optionalLisa != nil {  
    optionalLisa!.age += 1  
}
```

这种写法非常繁琐，也很丑陋。特别注意，在这种情况下你不能使用可选绑定。因为 `Person` 是一个结构体，所以它是一个值类型，绑定后的值只是原来值的局部作用域的复制，对这个复制进行变更，并不会影响原来的值：

```
if var lisa = optionalLisa {
```

```
// 对 lisa 的变更不会改变 optionalLisa 中的属性  
lisa.age += 1  
}
```

如果 Person 是类的话，这么做是可行的。我们会在结构体和类中进一步讨论值类型和引用类型的差异。就算能用可选绑定，这么写还是太过复杂了。其实，你可以使用可选值链来进行赋值，如果它不是 nil 的话，赋值操作将会成功：

```
optionalLisa?.age += 1
```

一个有点古怪 (但逻辑上合理) 的边界情况是你完全可以像下面这样直接给 Optional 赋值：

```
var a: Int? = 5  
a? = 10  
a // Optional(10)
```

```
var b: Int? = nil  
b? = 10  
b // nil
```

请注意 a = 10 和 a? = 10 的细微不同。前一种写法无条件地将一个新值赋给变量，而后一种写法只在 a 的值在赋值发生前不是 nil 的时候才生效。

## nil 合并运算符

很多时候，你都会想在解包可选值的同时，为 nil 的情况设置一个默认值。而这正是 nil 合并运算符的功能：

```
let stringteger = "1"  
let number = Int(stringteger) ?? 0
```

如果字符串可以被转换为一个整数的话，number 将会是那个解包后的整数值。如果不能的话，Int.init 将返回 nil，默认值 0 会被用来赋值给 number。也就是说 lhs ?? rhs 做的事情类似于 lhs != nil ? lhs! : rhs。

如果你之前使用过其他语言，那么可能会觉得 nil 合并操作符类似于 `a ? b : c` 这样的三元操作符。例如，要从可能为空的数组中获取第一个元素，你可以这样：

```
let array = [1,2,3]
!array.isEmpty ? array[0] : 0
```

因为 Swift 数组中有一个 `first` 属性，当数组为空时，它将为 `nil`。这样，你就可以直接使用 nil 合并操作符来完成这件事：

```
array.first ?? 1
```

这个解决方法漂亮且清晰，“从数组中获取第一个元素”这个意图被放在最前面，之后是通过 `??` 操作符连接的使用默认值的语句，它代表“这是一个默认值”。对比一下上面的，首先进行检查，然后取值，最后指定默认值的三元操作符的版本。老版本中，检查的时候使用了很不方便的否定形式（如果使用相反的判断逻辑的话，就要把默认值放在中间，把实际的值放到最后）。另外，在可选值的例子中，由于有编译器的保证，你不可能会忘记 `first` 是可选值并且在没有检查的情况下不小心使用它。

当你发现在检查某个条件来确保语句有效的时候，往往意味着使用可选值会是一个更好的选择。假设你要做的不是对空数组判定，而是要检查一个索引值是否在数组边界内：

```
array.count > 5 ? array[5] : 0 // 0
```

不像 `first` 和 `last`，通过索引值从数组中获取元素不会返回 `Optional`。不过我们可以对 `Array` 进行扩展来包含这个功能：

```
extension Array {
    subscript(guarded idx: Int) -> Element? {
        guard (startIndex..
```

现在你就可以这样写：

```
array[guarded: 5] ?? 0 // 0
```

合并操作也能够进行链接 — 如果你有多个可选值，并且想要选择第一个非 nil 的值，你可以将它们按顺序合并：

```
let i: Int? = nil
let j: Int? = nil
let k: Int? = 42
i ?? j ?? k ?? 0 // 42
```

正因为如此，所以如果你要处理的是双重嵌套的可选值，并且想使用 ?? 操作符的话，需要特别小心区分 a ?? b ?? c 和 (a ?? b) ?? c。前者是合并操作的链接，而后者是先解括号内的内容，然后再处理外层：

```
let s1: String?? = nil // nil
(s1 ?? "inner") ?? "outer" // inner
let s2: String?? = .some(nil) // Optional(nil)
(s2 ?? "inner") ?? "outer" // outer
```

如果你将 ?? 操作符看作是和 “or” 语句类似的话，那么可以把多个并列的 if let 语句视作 “and” 语句：

```
if let n = i, let m = j {}
// 和 if i != nil && j != nil 类似
```

和 || 操作符一样， ?? 操作符使用短路求值 (short circuiting)。当我们用 l ?? r 时，只有当 l 为 nil 时，r 的部分才会被求值。这是因为在操作符的函数声明中，对第二个参数使用了 @autoclosure。我们会在 函数 中详细讨论自动闭包 (autoclosure) 的工作原理。

## 在字符串插值中使用可选值

可能你已经注意到了，当你尝试打印一个可选值或者将一个可选值用在字符串插值表达式中时，编译器会给出警告：

```
let bodyTemperature: Double? = 37.0
let bloodGlucose: Double? = nil
print(bodyTemperature) // Optional(37.0)
// 警告：表达式被隐式强制从 'Double?' 转换为 Any
print("Blood glucose level: \(bloodGlucose)") // Blood glucose level: nil
// 警告：字符串插值将使用调试时的可选值描述,
// 请确认这确实是你要做的。
```

很多时候这个警告很有用，它可以防止我们把 "Optional(...)" 或者 "nil" 这样的东西不小心弄到我们想要显示给用户的文本里。你需要确保避免直接在面向用户的字符串中使用可选值，请一定记住先对它们进行解包。因为所有类型都允许放在字符串插值中（包括了 Optional），所以编译器不能将内嵌可选值当作一个错误，给出警告是它能做出的最好的选择。

有时候你确实会想要在字符串插值中使用可选值，比如想要在调试的时候将它的值打印出来，在这种情况下，警告就很烦人了。编译器为我们提供了几种修正这个警告的方式：显式地用 as Any 进行转换，使用 ! 对值进行强制解包（如果你能确定该值不为 nil 时），使用 String(describing: ...) 对它进行包装，或者用 nil 合并运算符提供一个默认值。

最后一种做法通常是比较快捷和优雅的方式，但是它有一点不足：在 ?? 表达式两侧的类型必须匹配，也就是说，你为一个 Double? 类型提供的默认值必须是 Double。因为我们最终的目标是将表达式转换为一个字符串，所以如果我们能够一开始就提供一个字符串作为默认值的话，就会特别方便。

Swift 的 ?? 运算符不支持这种类型不匹配的操作，确实，它无法决定当表达式两侧不共享同样的基础类型时，到底应该使用哪一个类型。不过，只是为了在字符串插值中使用可选值这一特殊目的的话，添加一个为我们自己量身定制的运算符也很简单。让我们把它叫做 ???：

**infix operator ???: NilCoalescingPrecedence**

```
public func ???<T>(optional: T?, defaultValue: @autoclosure () -> String)
-> String
{
    switch optional {
        case let value?: return String(describing: value)
```

```
case nil: return defaultValue()
}
}
```

这个函数接受左侧的可选值 `T?` 和右侧的字符串。如果可选值不是 `nil`，我们将它解包，然后返回它的字符串描述。否则，我们将传入的默认字符串返回。`@autoclosure` 标注确保了只有当需要的时候，我们才会对第二个表达式进行求值。在[函数](#)的相关章节中，我们会深入这部分内容。

现在，下面这样的代码不会触发任何编译器警告了：

```
print("Body temperature: \(bodyTemperature ??? "n/a")")
// Body temperature: 37.0
print("Blood glucose level: \(bloodGlucose ??? "n/a")")
// Blood glucose level: n/a
```

## 可选值 map

假设，我们要把一个字符数组的第一个元素转换成字符串：

```
let characters: [Character] = ["a", "b", "c"]
String(characters[0]) // a
```

如果 `characters` 可能为空的话，我们可以用 `if let`，在数组不为空的时候创建字符串：

```
var firstCharAsString: String? = nil
if let char = characters.first {
    firstCharAsString = String(char)
}
```

这样一来，`firstCharAsString` 就只在数组不为空的时候才包含对应元素的字符串，否则，它就是 `nil`。

这种只在可选值不为 nil 的时候才进行转换的模式十分常见。因此，可选值提供了一个 map 方法专门处理这个问题。它接受一个转换可选值内容的函数作为参数。把刚才转换字符数组的功能用 map 来实现，就是这样的：

```
let firstChar = characters.first.map { String($0) } // Optional("a")
```

显然，这个 map 和数组以及其他序列里的 map 方法非常类似。但是与序列中操作一系列值所不同的是，可选值的 map 方法只会操作一个值，那就是该可选值中的那个可能存在的值。你可以把可选值当作一个包含零个或者一个值的集合，这样 map 要么在零个值的情况下不做处理，要么在有值的时候会对其进行转换。

以下是在可选值上实现 map 的一种方式：

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

当你想要的就是一个可选值结果时，Optional.map 就非常有用。假设你要为数组实现一个变种的 reduce 方法，这个方法不接受初始值，而是直接使用数组中的首个元素作为初始值（在一些语言中，这个函数可能被叫做 reduce1，但是 Swift 里我们有重载，所以也将它叫做 reduce 就行了）。

因为数组可能会是空的，这种情况下没有初始值，结果只能是可选值。你可能会这样来实现它：

```
extension Array {  
    func reduce(_ nextPartialResult: (Element, Element) -> Element) -> Element? {  
        // 如果数组为空，first 将是 nil  
        guard let fst = first else { return nil }  
        return dropFirst().reduce(fst, nextPartialResult)  
    }  
}
```

你可以这样来使用它：

```
[1, 2, 3, 4].reduce(+) // Optional(10)
```

因为可选值为 nil 时，可选值 map 也会返回 nil，所以我们可以不使用 guard，而就用一个 return 来重写我们的这个 reduce：

```
extension Array {  
    func reduce_alt(_ nextPartialResult: (Element, Element) -> Element)  
        -> Element?  
    {  
        first.map {  
            dropFirst().reduce($0, nextPartialResult)  
        }  
    }  
}
```

## 可选值 flatMap

我们在内建集合中已经看到，在集合上运行 map 并给定一个变换函数可以获取新的集合，但是一般来说我们想要的结果会是一个单一的数组，而不是数组的数组。

类似地，如果你对一个可选值调用 map，但是你的转换函数本身也返回可选值的话，最终结果将是一个双重嵌套的可选值。举个例子，比如你想要获取数组的第一个字符串元素，并将它转换为数字。首先你使用数组上的 first，然后用 map 将它转换为数字：

```
let stringNumbers = ["1", "2", "3", "foo"]  
let x = stringNumbers.first.map { Int($0) } // Optional(Optional(1))
```

问题在于，map 返回可选值（因为 first 可能会是 nil），并且 Int(String) 也返回可选值（字符串可能不是一个整数），最后 x 的结果将会是 Int??。

flatMap 可以把结果展平为单个可选值。这样一来，y 的类型将会是 Int?:

```
let y = stringNumbers.first.flatMap { Int($0) } // Optional(1)
```

对于这个例子，你还可能通过 if let 实现，因为后面被绑定的值可以从前面绑定的值计算出来：

```
if let a = stringNumbers.first, let b = Int(a) {  
    print(b)  
} // 1
```

这显示了 flatMap 和 if let 是非常相似的。在本章早些时候，我们已经看过使用多个 if-let 语句的例子了，我们也可以用 map 和 flatMap 进行重写：

```
let urlString = "https://www.objc.io/logo.png"  
let view = URL(string: urlString)  
.flatMap { try? Data(contentsOf: $0) }  
.flatMap { UIImage(data: $0) }  
.map { UIImageView(image: $0) }  
  
if let view = view {  
    PlaygroundPage.current.liveView = view  
}
```

可选链也和 flatMap 很相似：`i?.advance(by: 1)` 实际上和 `i.flatMap { $0.advance(by: 1) }` 是等价的。

我们已经看到多个 if-let 语句等价于 flatMap，所以我们可以用这种方式来实现它：至此，就像我们已经看到的，既然多个 if let 语句连接在一起的用法和 flatMap 是等价的，我们就可以像下面这样，用前者来实现后者：

```
extension Optional {  
    func flatMap<U>(transform: (Wrapped) -> U?) -> U? {  
        if let value = self, let transformed = transform(value) {  
            return transformed  
        }  
        return nil  
    }  
}
```

## 使用 compactMap 过滤 nil

如果你的序列中包含可选值，可能你会只对那些非 nil 值感兴趣。实际上，你可能只想忽略掉它们。

设想在一个字符串数组中你只想处理数字。在 for 循环中，用可选值模式匹配可以很简单地就实现：

```
let numbers = ["1", "2", "3", "foo"]
var sum = 0
for case let i? in numbers.map({ Int($0) }) {
    sum += i
}
sum // 6
```

你可能也会想用 ?? 来把 nil 替换成 0：

```
numbers.map { Int($0) }.reduce(0) { $0 + ($1 ?? 0) } // 6
```

实际上，你想要的版本应该是一个可以将那些 nil 过滤出去并将非 nil 值进行解包的 map。标准库中序列的 compactMap 正是你想要的：

```
numbers.compactMap { Int($0) }.reduce(0, +) // 6
```

为了实现我们自己的 compactMap，先转换整个数组，然后过滤出非 nil 值，最后解包每个被过滤出来的元素：

```
extension Sequence {
    func compactMap<B>(_ transform: (Element) -> B?) -> [B] {
        return lazy.map(transform).filter { $0 != nil }.map { $0! }
    }
}
```

在这个实现中，使用了 `lazy` 来将数组的创建推迟到了使用前的最后一刻。这可能只是一个小的优化，但在处理较大的序列时还是值得的。使用 `lazy` 可以避免多个作为中间结果的数组的内存分配。不过标准库中的 `compactMap` 实现却不是这么做的。在 [集合类型协议](#) 中，我们会探究延迟序列和延迟集合的细节。

## 可选值判等

通常，你不关心一个值是不是 `nil`，只想检查它是否包含某个特定值而已：

```
let regex = "^Hello$"  
// ...  
if regex.first == "^" {  
    // 只匹配字符串开头  
}
```

在这种情况下，值是否是 `nil` 并不关键。如果字符串是空，它的第一个字符肯定不是插入符号 `^`，所以你不会进入到相应的代码块中。但是你肯定还是会想要 `first` 为你带来的安全保障和简洁。如果用替代的写法，将会是 `if !regex.isEmpty && regex[regex.startIndex] == "^"`，这太可怕了。

上面的代码之所以能工作主要基于两点。首先，只有当 `Optional` 的 `Wrapped` 类型实现了 `Equatable` 协议时，这个 `Optional` 才会也实现 `Equatable` 协议：

```
extension Optional: Equatable where Wrapped: Equatable {  
    static func ==(lhs: Wrapped?, rhs: Wrapped?) -> Bool {  
        switch (lhs, rhs) {  
            case (nil, nil): return true  
            case let (x?, y?): return x == y  
            case (_, nil), (nil, _?): return false  
        }  
    }  
}
```

当比较两个可选值时，会有四种组合的可能性：两者都是 nil；两者都有值；两者中有一个有值，另一个是 nil。switch 语句完成了对这四种组合的遍历，所以这里并不需要 default 语句。两个 nil 的情况被定义为相等，而 nil 永远不可能等于非 nil，两个非 nil 的值将通过解包后的值是否相等来进行判断。

第二点，要注意一下，我们并不一定要写这样的代码：

```
if regex.first == Optional("^") { // 或者: == .some("^")
    // 只匹配字符串开头
}
```

这是因为当你在使用一个非可选值的时候，如果需要匹配成可选值类型，Swift 总是会将它“升级”为一个可选值。

这个隐式的转换对于写出清晰紧凑的代码特别有帮助。设想要是没有这样的转换，但是还是希望调用者在使用的时候比较容易的话，你需要 == 可以同时作用于可选值和非可选值。这样一来，会需要实现三个分开的版本：

```
// 两者都可选
func == <T: Equatable>(lhs: T?, rhs: T?) -> Bool
// lhs 非可选
func == <T: Equatable>(lhs: T, rhs: T?) -> Bool
// rhs 非可选
func == <T: Equatable>(lhs: T?, rhs: T) -> Bool
```

不过事实是我们只需要第一个版本，编译器会帮助我们将值在需要时转变为可选值。

实际上，我们在整本书中都依赖这个隐式的转换。比方说，当我们在实现 Optional.map 时，我们将内部的实际值进行转换并返回。但是我们知道 map 的返回值其实是个 Optional 值。编译器自动帮我们完成了转换，得益于此，我们不需要写 return Optional(transform(value)) 这样的代码。

Swift 代码也一直依赖这个隐式转换。例如，使用键作为下标在字典中查找时，因为键有可能不存在，所以返回值是可选值。对于用下标读取和写入时，所需要的类型是相同的。也就是说，

在使用下标进行赋值时，我们其实需要传入一个可选值。如果没有隐式转换，你就必须写像是 `myDict["someKey"] = Optional(someValue)` 这样的代码。

附带提一句，如果你想知道当使用下标操作为字典的某个键赋值 `nil` 会发什么的话，答案就是这个键会从字典中移除。有时候这会很有用，但是这也意味着你在使用字典来存储可选值类型时需要小心一些。看看这个字典：

```
var dictWithNils: [String: Int?] = [
    "one": 1,
    "two": 2,
    "none": nil
]
```

这个字典有三个键，其中一个的值是 `nil`。如果我们想要把 `"two"` 的键也设置为 `nil` 的话，下面的代码是做不到的：

```
dictWithNils["two"] = nil
dictWithNils // ["one": Optional(1), "none": nil]
```

它将会把 `"two"` 这个键移除。

为了改变这个键的值，你可以使用下面中的任意一个。它们都可以正常工作，所以选择一个你觉得清晰的就可：

```
dictWithNils["two"] = Optional(nil)
dictWithNils["two"] = .some(nil)
dictWithNils["two"]? = nil
dictWithNils // ["one": Optional(1), "none": nil, "two": nil]
```

注意上面的第三个版本和其他两个稍有不同。它之所以能够工作，是因为 `"two"` 这个键已经存在于字典中了，所以它使用了可选链的方式来在获取成功后对值进行设置。现在来看看对于不存在的键进行设置会怎么样：

```
dictWithNils["three"]? = nil
dictWithNils.index(forKey: "three") // nil
```

你可以看到并没有值被更新或者插入。

## 可选值比较

和 `==` 类似，可选值曾经也是拥有 `<`、`>`、`<=` 和 `>=` 这些操作符的。在 Swift 3.0 中，这些操作符从可选值中被移除了，因为它们容易导致意外的结果。

比如说，`nil < .some(_)` 会返回 `true`。在与高阶函数或者可选绑定结合起来使用的时候，会产生很多意外。考虑下面（这个已经过时了）的例子：

```
let temps = ["-459.67", "98.6", "0", "warm"]
let belowFreezing = temps.filter { Double($0) < 0 }
```

因为 `Double("warm")` 将会返回 `nil`，而 `nil` 被定义为了小于 0，所以它将被包含在 `belowFreezing` 温度中，这显然是不合情理的。

如果你想要在可选值之间进行除了相等之外的关系比较的话，现在你需要先对它们进行解包，然后明确地决定 `nil` 要如何处理。

## 强制解包的时机

上面提到的例子都用了很干净的方式来解包可选值，那什么时候你应该用感叹号 (!) 这个强制解包运算符呢？在网上散布着各种说法，比如“绝不使用”，“当可以让代码变清晰时使用”，或者“在不可避免的时候使用”。我们提出了下面这个规则，它概括了大多数的场景：

当你能确定你的某个值不可能是 `nil` 时可以使用叹号，你应当会希望如果它意外是 `nil` 的话，程序应当直接挂掉。

举个例子，看看之前 `compactMap` 的实现：

```
extension Sequence {
    func compactMap<B>(_ transform: (Element) -> B?) -> [B] {
        return lazy.map(transform).filter {$0 != nil }.map {$0! }
```

```
    }  
}
```

在这里，因为 filter 的阶段已经把所有 nil 元素过滤出去了，所以 map 的时候没有任何可能会出现 \$0! 碰到 nil 的情况。当然可以把强制解包运算符从这个函数中消除掉，通过使用循环的方法一个一个检查数组中的元素，并将那些非 nil 的元素添加到一个数组中。但是 filter / map 结合起来的版本更加简洁和清晰，所以这里使用 ! 是完全没有问题的。

不过这些情况还是很罕见的。如果你完全掌握了本章中提到的解包的知识，一般应该可以找到比强制解包更好的方法。每当你发现需要使用 ! 时，可以回头看看是不是真的别无他法了。

第二个例子，下面这段代码会根据特定的条件从字典中找到满足这个条件的值所对应的所有的键：

```
let ages = [  
  "Tim": 53, "Angela": 54, "Craig": 44,  
  "Jony": 47, "Chris": 37, "Michael": 34,  
]  
  
ages.keys  
.filter { name in ages[name]! < 50 }  
.sorted()  
// ["Chris", "Craig", "Jony", "Michael"]
```

同样，这里使用 ! 是非常安全的 — 因为所有的键都来源于字典，所以在字典中找不到这个键是不可能的。

不过你也可以通过一些手段重写这几句代码，来把强制解包移除掉。利用字典本身是一个键值对的序列这一特性，你可以对序列先进行过滤，然后再通过映射来返回键的序列：

```
ages.filter { (_, age) in age < 50 }  
.map { (name, _) in name }  
.sorted()  
// ["Chris", "Craig", "Jony", "Michael"]
```

这样的写法可能还会额外带来一些性能上的收益，它避免了不必要的用键进行的查找。

尽管如此，有时还是会造化弄人，你有一个可选值，并且确实知道它不可能是 nil。在这种情况下，当你碰到一个 nil 值的时候，你肯定会选择让程序挂掉而不是让它继续运行，因为这意味着在你的逻辑中有一个非常严重的 bug。此时，终止程序而不是让它继续运行会是更好的抉择，这里 ! 这一个符号就实现了“解包”和“报错”两种功能的结合。相比于使用 nil 可选链或者合并运算符来在背后清除这种理论上不可能存在的方法，直接强制解包的处理方式通常要好一些。

## 改进强制解包的错误信息

就算你要对一个可选值进行强制解包，除了使用 ! 操作符以外，你还有其他的选择。当程序发生错误时，你从输出的 log 中无法通过描述知道原因是什么。

其实，你可能会留一个注释来提醒为什么这里要使用强制解包。那为什么不把这个注释直接作为错误信息呢？这里我们加了一个 !! 操作符，它将强制解包和一个更具有描述性质的错误信息结合在一起，当程序意外退出时，这个信息也会被打印出来：

```
infix operator !!
```

```
func !!<T>(wrapped: T?, failureText: @autoclosure () -> String) -> T {  
    if let x = wrapped { return x }  
    fatalError(failureText())  
}
```

现在你可以写出更能描述问题的错误信息了，它还包括了你期望的被解包的值：

```
let s = "foo"  
let i = Int(s) !! "Expecting integer, got \"\$(s)\""
```

## 在调试版本中进行断言

说实话，选择在发布版中让应用崩溃还是很大胆的行为。通常，你可能会选择在调试版本或者测试版本中进行断言，让程序崩溃，但是在最终产品中，你可能会把它替换成像是零或者空数组这样的默认值。

我们可以实现一个疑问感叹号 !? 操作符来代表这个行为。我们将这个操作符定义为对失败的解包进行断言，并且在断言不触发的发布版本中将值替换为默认值：

### infix operator !?

```
func !?<T: ExpressibleByIntegerLiteral>
(wrapped: T?, failureText: @autoclosure () -> String) -> T
{
    assert(wrapped != nil, failureText())
    return wrapped ?? 0
}
```

现在，下面的代码将在调试时触发断言，但在发布版本中打印 0：

```
let s = "20"
let i = Int(s) !? "Expecting integer, got \"\$(s)\""
```

对其他字面量转换协议进行重载，可以覆盖不少能够有默认值的类型：

```
func !?<T: ExpressibleByArrayLiteral>
(wrapped: T?, failureText: @autoclosure () -> String) -> T
{
    assert(wrapped != nil, failureText())
    return wrapped ?? []
}

func !?<T: ExpressibleByStringLiteral>
(wrapped: T?, failureText: @autoclosure () -> String) -> T
{
    assert(wrapped != nil, failureText())
    return wrapped ?? ""
```

如果你想要显式地提供一个不同的默认值，或者是为非标准的类型提供这个操作符，我们可以定义一个接受元组为参数的版本，元组包含默认值和错误信息：

```
func !?<T>(wrapped: T?,  
    nilDefault: @autoclosure () -> (value:T, text: String)) -> T  
{  
    assert(wrapped != nil, nilDefault().text)  
    return wrapped ?? nilDefault().value  
}  
  
// 调试版本中断言，发布版本中返回 5  
Int(s) !?(5, "Expected integer")
```

因为对于返回 Void 的函数，使用可选链进行调用时将返回 Void?，所以利用这一点，你也可以写一个非泛型的版本来检测一个可选链调用碰到 nil，且无操作的情况：

```
func !?(wrapped: ()?, failureText: @autoclosure () -> String) {  
    assert(wrapped != nil, failureText())  
}  
var output: String? = nil  
output?.write("something") !? "Wasn't expecting chained nil here"
```

想要挂起一个操作我们有三种方式。首先，fatalError 将接受一条信息，并且无条件地停止操作。第二种选择，使用 assert 来检查条件，当条件结果为 false 时，停止执行并输出信息。在发布版本中，assert 会被移除掉，也就是说条件不会被检测，操作也永远不会挂起。第三种方式是使用 precondition，它和 assert 有一样的接口，但是在发布版本中不会被移除，也就是说，只要条件被判定为 false，执行就会被停止。

## 隐式解包可选值

可不要弄错了，像 UIView! 这种在类型的后面加一个感叹号的隐式解包可选值，尽管不论何时你使用它们的时候都会自动强制解包，但它们仍然只是可选值。现在我们已经知道了当可选值

是 nil 的时候强制解包会造成应用崩溃，那你到底为什么要用到隐式可选值呢？好吧，实际上有两个原因：

原因 1：暂时来说，你可能还需要到 Objective-C 里去调用那些没有检查返回是否存在的代码；或者你会调用一个没有针对 Swift 做注解的 C 语言的库。

隐式解包可选值还存在的唯一原因其实是为了能更容易地和 Objective-C 与 C 一起使用。在早期我们刚开始通过 Swift 来使用已经存在的那些 Objective-C 代码时，所有返回引用的 Objective-C 方法都被转换为了返回一个隐式的可选值。因为其实 Objective-C 中表示引用是否可以为空的语法是最近才被引入的，以前除了假设返回的引用可能是 nil 引用以外，也没有什么好办法。但是只有很少的 Objective-C API 会真的返回一个空引用，所以将它们自动在 Swift 里暴露为普通的可选值是一件很烦人的事情。因为所有人都已经习惯了 Objective-C 世界中对象“可能为空”的设定，因此把这样的返回值作为隐式解包可选值来使用是可以说得过去的。

所以你会在有些没有经过检查的 Objective-C 桥接代码中看到它们的身影。但你不应该看见一个纯的原生 Swift API 返回一个隐式可选值，也不应该把它们传入回调中。

原因 2：因为一个值只是很短暂地为 nil，在一段时间后，它就再也不会是 nil。

最常见的情况就是两阶段初始化 (two-phase initialization)。当你的类准备好被使用时，所有的隐式解包可选值都将有一个值。这就是 Xcode 和 Interface Builder 在 view controller 的生命周期中使用它们的方式：在 Cocoa 和 Cocoa Touch 中，view controller 会延时创建他们的 view，所以在 view controller 自身已经被初始化，但是它的 view 还没有被加载的这段时间窗口内，被 view controller 所引用的 view 对象还没有被创建。

## 隐式可选值行为

虽然隐式解包的可选值在行为上就好像是非可选值一样，不过你依然可以对它们使用可选链，nil 合并，if let，map 或者将它们与 nil 比较，所有的这些操作都是一样的：

```
var s: String! = "Hello"  
s?.isEmpty // Optional(false)  
if let s = s { print(s) } // Hello  
s = nil
```

```
s ?? "Goodbye" // Goodbye
```

## 回顾

可选值是 Swift 的一大卖点，它是让开发者得以书写更安全的代码的最大特性之一，而且我们完全同意这个说法。但仔细想想，其实真正带来变化的不是可选值，而是非可选值。几乎所有的主流语言都有类似“null”或者“nil”的概念；它们中的大多数所缺乏的是把一个值声明为“从不为 nil”的能力。或者，反过来想，有一些类型（比如 Objective-C 或 Java 中的非 class 类型）“总是不为 nil”，这又让开发者们必须用某个魔法数来表示缺少一个值的情况。

设计 API 的过程中，根据实际需要让输入和输出包含精心设计过的可选值，不仅会让调用函数的代码更具表现力，这些函数用起来也会更简单。因为通过签名可以传递更多的信息，开发者也就不用总是诉诸文档了。

我们在本章中呈现的所有解包可选值的方法，都是 Swift 对于如何在可选值与非可选值的两个世界之间搭建一座尽可能方便的桥梁的尝试。你想要使用哪种方法，最后往往取决于你的个人选择。

---

# 函数

4

# 综述

开始本章之前，让我们先来回顾一些关于函数的重要概念。如果你已经很熟悉头等函数（first-class function）的话，就可以直接跳到下一节了。但如果你对此还有些懵懵懂懂的话，请仔细阅读以下内容。

要理解 Swift 中的函数和闭包，你需要切实弄明白三件事情，我们把这三件事按照重要程度进行了大致排序：

0. 函数可以像 Int 或者 String 那样被赋值给变量，也可以作为另一个函数的输入参数，或者另一个函数的返回值来使用。
1. 函数能够捕获存在于其局部作用域之外的变量。
2. 有两种方法可以创建函数，一种是使用 func 关键字，另一种是 {}。在 Swift 中，后一种被称为闭包表达式。

有时候，新接触闭包的人会认为上面这三点的重要顺序是反过来的，并且会忽略其中的某一点，或把闭包和闭包表达式混为一谈。尽管这些概念确实容易引起困惑，但这三点却是鼎足而立，互为补充的，如果你忽视其中任何一条，终究会在函数的应用上，狠狠地摔上一跤。

## 1. 函数可以被赋值给变量，也能够作为函数的输入和输出

Swift 和很多现代化编程语言相同，都把函数视为“头等对象”。你既可以将函数赋值给变量，也可以将它作为其他函数的参数或返回值。

这一点是我们需要理解的最重要的东西。在函数式编程中明白这一点，就和在 C 语言中明白指针的概念一样。如果你没有牢牢掌握这部分的话，其他所有东西都将是镜花水月。

让我们从一个打印整数的函数开始：

```
func printInt(i: Int) {  
    print("You passed \(i).")  
}
```

要将函数赋值给一个变量，比如 `funVar`，我们可以把函数名字作为值。注意在函数名后面没有括号：

```
let funVar = printInt
```

现在，我们可以使用 `funVar` 变量来调用 `printInt` 函数。注意在变量名后面需要使用括号：

```
funVar(2) // You passed 2.
```

这里值得注意的是，我们不能在 `funVar` 调用时包含参数标签，而在 `printInt` 的调用 (像是 `printInt(i: 2)`) 却要求有参数标签。Swift 只允许在函数声明中包含标签，这些标签不是函数类型的一部分。也就是说，现在你不能将参数标签赋值给一个类型是函数的变量，不过这在未来 Swift 版本中可能会有改变。

我们也能够写出一个接受函数作为参数的函数：

```
func useFunction(function: (Int) -> () {
    function(3)
}

useFunction(function: printInt) // You passed 3.
useFunction(function: funVar) // You passed 3.
```

为什么函数可以作为变量使用的这种能力如此关键呢？因为它让你很容易写出“高阶”函数，高阶函数将函数作为参数的能力使得它们在很多方面都非常有用，我们已经在内建集合类型中看到过它的威力了。

函数也可以返回其他函数：

```
func returnFunc() -> (Int) -> String {
    func innerFunc(i: Int) -> String {
        return "you passed \(i)"
    }
    return innerFunc
}
```

```
let myFunc = returnFunc()  
myFunc(3) // you passed 3
```

## 2. 函数可以捕获存在于它们作用域之外的变量

当函数引用了在其作用域之外的变量时，这个变量就被捕获了，它们将会继续存在，而不是在超过作用域后被摧毁。

为了研究这一点，让我们修改一下 `returnFunc` 函数。这次我们添加一个计数器，每次调用这个函数时，计数器将会增加：

```
func makeCounter() -> (Int) -> String {  
    var counter = 0  
    func innerFunc(i: Int) -> String {  
        counter += i // counter 被捕获  
        return "Running total: \(counter)"  
    }  
    return innerFunc  
}
```

一般来说，因为 `counter` 是 `makeCounter` 的局部变量，它在 `return` 语句执行之后就应该离开作用域并被摧毁。但因为 `innerFunc` 捕获了它，所以 Swift 运行时将一直保证它的存在，直到捕获它的函数被销毁为止。我们可以多次调用 `innerFunc`，并且看到 `running total` 的输出在增加：

```
let f = makeCounter()  
f(3) // Running total: 3  
f(4) // Running total: 7
```

如果我们再次调用 `makeCounter()` 函数，将会生成并捕获一个新的 `counter` 变量：

```
let g = makeCounter()  
g(2) // Running total: 2  
g(2) // Running total: 4
```

这并不影响我们的第一个函数，它拥有属于自己的 counter：

```
f(2) // Running total: 9
```

你可以将这些函数以及它们所捕获的变量想象为一个类的实例，这个类拥有一个单一的方法（也就是这里的函数）以及一些成员变量（这里的被捕获的变量）。

在编程术语里，一个函数和它所捕获的变量环境组合起来被称为闭包。上面 f 和 g 都是闭包的例子，因为它们捕获并使用了一个在它们作用域之外声明的非局部变量 counter。

### 3. 函数可以使用 {} 来声明为闭包表达式

在 Swift 中，定义函数的方法有两种。一种是使用 func 关键字。另一种方法是使用闭包表达式。下面这个简单的函数将会把数字翻倍：

```
func doubler(i: Int) -> Int {  
    return i * 2  
}  
[1, 2, 3, 4].map(doubler) // [2, 4, 6, 8]
```

使用闭包表达式的语法来写相同的函数，像之前那样将它传给 map：

```
let doublerAlt = { (i: Int) -> Int in return i*2 }  
[1, 2, 3, 4].map(doublerAlt) // [2, 4, 6, 8]
```

使用闭包表达式来定义的函数可以被想成函数的字面量 (**function literals**)，就像 1 是整数字面量，"hello" 是字符串字面量那样。与 func 相比，它的区别在于闭包表达式是匿名的，它们没有被赋予一个名字。使用它们的方式只能是在它们被创建时将其赋值给一个变量（就像我们这里对 doubler 进行的赋值一样），或者是将它们传递给另一个函数或方法。

其实还有第三种使用匿名函数的方法：你可以在定义一个表达式的同时，对它进行调用。这个方法在定义那些初始化时代码多于一行的属性时会很有用。我们将在属性章节中看到一个例子。

使用闭包表达式声明的 `doubler`, 和之前使用 `func` 关键字声明的函数, 除了在参数标签上的处理上略有不同以外, 其实是完全等价的。它们甚至存在于同一个“命名空间”中, 这一点和有些编程语言有所不同。

那么 {} 语法有什么用呢? 为什么不每次都使用 `func` 呢? 因为闭包表达式可以简洁得多, 特别是在像是 `map` 这样的将一个快速实现的函数传递给另一个函数时, 这个特点更为明显。这里, 我们将 `doubler map` 的例子用更短的形式进行了重写:

```
[1, 2, 3].map { $0 * 2 } // [2, 4, 6]
```

之所以看起来和原来很不同, 是因为这里使用了 Swift 中的一些可以让代码更加简洁的特性, 我们来一一看下它们:

0. 如果你将闭包作为参数传递, 并且你不再用这个闭包做其他事情的话, 就没有必要先将它存储到一个局部变量中。可以想象一下比如 `5*i` 这样的数值表达式, 你可以把它直接传递给一个接受 `Int` 的函数, 而不必先将它计算并存储到变量里。
1. 如果编译器可以从上下文中推断出类型的话, 你就不需要指明它了。在我们的例子中, 从数组元素的类型可以推断出传递给 `map` 的函数接受 `Int` 作为参数, 从闭包内的乘法结果的类型可以推断出闭包返回的也是 `Int`。
2. 如果闭包表达式的主体部分只包括一个单一的表达式的话, 它将自动返回这个表达式的结果, 你可以不写 `return`。
3. Swift 会自动为函数的参数提供简写形式, `$0` 代表第一个参数, `$1` 代表第二个参数, 以此类推。
4. 如果函数的最后一个参数是闭包表达式的话, 你可以将这个闭包表达式移到函数调用的圆括号的外部。这样的尾随闭包语法 (**trailing closure syntax**) 在多行的闭包表达式中表现非常好, 因为它看起来更接近于装配了一个普通的函数定义, 或者是像 `if (expr) {}` 这样的执行块的表达形式。从 Swift 5.3 开始, 多个尾随闭包也被支持了。
5. 最后, 如果一个函数除了闭包表达式外没有别的参数, 那么调用的时候在方法名后面的圆括号也可以一并省略。

依次将上面的规则应用到最初的表达式里, 我们就可以逐步得到最后的结果了:

```
/* */ [1, 2, 3].map({ (i: Int) -> Int in return i * 2 })
/* */ [1, 2, 3].map({ i in return i * 2 })
/* */ [1, 2, 3].map({ i in i * 2 })
/* */ [1, 2, 3].map({ $0 * 2 })
/* */ [1, 2, 3].map() { $0 * 2 }
/* */ [1, 2, 3].map { $0 * 2 }
```

如果你刚接触 Swift 的语法，或者刚接触函数式编程的话，这些精简的函数表达第一眼看起来可能让你丧失信心。但是一旦你习惯了这样的语法以及函数式编程风格的话，它们很快就会看起来很自然，移除这些杂乱的表达，可以让你对代码实际做的事情看得更加清晰，你一定会为语言中有这样的特性而心存感激。一旦你习惯了阅读这样的代码，你一眼就能看出这段代码做了什么，而想在一个等效的 for 循环中做到这一点则要困难得多。

Swift 在进行类型推断的时候，可能需要你提供一些帮助。还有些情况下，你可能会遇到问题并且类型和你认为的也不一样。如果你在尝试提供闭包表达式时遇到一些谜一样的错误的话，将闭包表达式写成上面例子中的第一种包括类型的完整形式，往往会是个好主意。在很多情况下，这有助于厘清错误到底在哪儿。一旦完整版本可以编译通过，你就可以逐渐将类型移除，直到编译无法通过。如果造成错误的是你的代码的话，在这个过程中相信你已经修复好这些代码了。

还有一些时候，Swift 会要求你用更明确的方式进行调用。假设你要得到一个随机数数组，一种快速的方法就是通过 Range.map 方法，并在 map 的函数中生成并返回随机数。这里，无论如何你都要为 map 的函数提供一个参数。或者明确使用 \_ 告诉编译器你承认这里有一个参数，但并不关心它究竟是什么：

```
(0..<3>.map { _ in Int.random(in: 1..<100) } // [5, 64, 70]
```

当你需要显式地指定变量类型时，你不一定要在闭包表达式内部来设定。比如，让我们来定义一个 isEven，它不指定任何类型：

```
let isEven = { $0 % 2 == 0 }
```

在上面，isEven 被推断为 `Int -> Bool`。这和 `let i = 1` 被推断为 `Int` 是一个道理，因为 `Int` 是整数字面量的默认类型。

这是因为标准库中的 `IntegerLiteralType` 有一个类型别名：

```
protocol ExpressibleByIntegerLiteral {  
    associatedtype IntegerLiteralType  
    /// 用 `value` 创建一个实例。  
    init(integerLiteral value: Self.IntegerLiteralType)  
}  
  
/// 一个没有其余类型限制的整数字面量的默认类型。  
typealias IntegerLiteralType = Int
```

如果你想要定义你自己的类型别名，你可以重写默认值来改变这一行为：

```
typealias IntegerLiteralType = UInt32  
let i = 1 // i 的类型为 UInt32.
```

显然，这不是一个什么好主意。

不过，如果你需要 `isEven` 是别的类型的话，也可以在闭包表达式中为参数和返回值指定类型：

```
let isEvenAlt = { (i: Int8) -> Bool in i % 2 == 0 }
```

你也可以在闭包外部的上下文里提供这些信息：

```
let isEvenAlt2: (Int8) -> Bool = { $0 % 2 == 0 }  
let isEvenAlt3 = { $0 % 2 == 0 } as (Int8) -> Bool
```

因为闭包表达式最常见的使用情景就是在一些已经存在输入或者输出类型的上下文中，所以这种写法并不是经常需要，不过知道它还是会很有用。

当然了，如果能定义一个对所有整数类型都适用的 `isEven` 的泛用版本的计算属性会更好：

```
extension BinaryInteger {  
    var isEven: Bool { return self % 2 == 0 }
```

```
}
```

或者，我们也可以选择为所有的 Integer 类型定义一个全局函数：

```
func isEven<T: BinaryInteger>(_ i:T) -> Bool {  
    return i % 2 == 0  
}
```

要把这个全局函数赋值给变量的话，你需要先决定它的参数类型。变量不能持有泛型函数，它只能持有一个类型具体化之后的版本：

```
let int8IsEven: (Int8) -> Bool = isEven
```

最后要说明的是关于命名的问题。要清楚，那些使用 func 声明的函数也可以是闭包，就和用 {} 声明的一样的。记住，闭包指的是一个函数以及被它所捕获的所有变量的组合。而使用 {} 来创建的函数被称为 **闭包表达式**，人们常常会把这种语法简单地叫做闭包。但是不要因此就认为使用闭包表达式语法声明的函数和其他方法声明的函数有什么不同。它们都是一样的，它们都是函数，也都可以是闭包。

## 函数的灵活性

在内建集合类型一章中，我们谈到过使用函数将行为参数化。现在让我们来看一个另外的例子：排序。

在 Swift 中为集合排序很简单：

```
let myArray = [3, 1, 2]  
myArray.sorted() // [1, 2, 3]
```

一共有四个排序的方法：不可变版本的 sorted(by:) 和可变的 sort(by:)，以及两者在待排序对象遵守 Comparable 时进行升序排序的无参数版本。对于最常见的情况，这个无参数的 sorted() 就是你所需要的。如果你需要用不同于默认升序的顺序进行排序的话，只需提供一个排序函数：

```
myArray.sorted(by: >) // [3, 2, 1]
```

就算待排序的元素不遵守 Comparable，但是只要有 `<` 操作符，你就可以使用这个方法来进行排序，比如多元组 (tuple) 就是一个例子：

```
var numberStrings = [(2, "two"), (1, "one"), (3, "three")]
numberStrings.sort(by: <)
numberStrings // [(1, "one"), (2, "two"), (3, "three")]
```

(提案 [SE-0283](#) 提出让多元组自动满足像是 Comparable 这样的标准协议。这个提案已经在 2020 年被采纳了，但是直到 Swift 5.5 它还没有被实现。)

或者，你也可以使用一个更复杂的函数，来按照任意你需要的计算标准进行排序：

```
let animals = ["elephant", "zebra", "dog"]
animals.sorted { lhs, rhs in
    let l = lhs.reversed()
    let r = rhs.reversed()
    return l.lexicographicallyPrecedes(r)
}
// ["zebra", "dog", "elephant"]
```

最后，Swift 的排序还有一个能力，它可以使用任意的比较函数来对集合进行排序。这使得 Swift 排序非常强大。

不过，如果我们想要按照多个特征进行排序的话，要怎么做呢？举例来说，如果有一个 Person 结构体，我们想要先按姓来排序，如果姓相同，那么按姓名排序。在 Objective-C 中，这可以通过 NSSortDescriptor 来完成。虽然 NSSortDescriptor (以及它当前的变体 SortDescriptor) 非常灵活且强大，但它只能针对 NSObject 排序。这个限制是由于它使用了 Objective-C 的运行时系统。在本章中，我们会使用高阶函数来重新实现我们自己的 SortDescriptor，并让它同样灵活和强大。

我们首先定义一个 Person 类型：

```
struct Person {
    let first: String
    let last: String
```

```
let yearOfBirth:Int  
}
```

接下来我们定义一个数组，其中包含了不同名字和出生年份的人：

```
let people = [  
    Person(first: "Emily", last: "Young", yearOfBirth: 2002),  
    Person(first: "David", last: "Gray", yearOfBirth: 1991),  
    Person(first: "Robert", last: "Barnes", yearOfBirth: 1985),  
    Person(first: "Ava", last: "Barnes", yearOfBirth: 2000),  
    Person(first: "Joanne", last: "Miller", yearOfBirth: 1994),  
    Person(first: "Ava", last: "Barnes", yearOfBirth: 1998),  
]
```

我们想要对这个数组进行排序，规则是先按照姓排序，再按照名排序，最后是出生年份。排序应该遵照用户的区域设置。一开始，我们先用一个键来排序，比如姓：

```
/* */people.sorted { p1, p2 in  
    p1.last.localizedStandardCompare(p2.last) == .orderedAscending  
}
```

如果我们想要先比较姓，然后再比较名，情况就已经变得比较复杂了：

```
/* */people.sorted { p1, p2 in  
    switch p1.last.localizedStandardCompare(p2.last) {  
        case .orderedAscending:  
            return true  
        case .orderedDescending:  
            return false  
        case .orderedSame:  
            return p1.first.localizedStandardCompare(p2.first) == .orderedAscending  
    }  
}
```

## 函数作为数据

我们不会再去写一个包含出生年分的更加复杂的函数了，我们先往回一步，来看看能不能引入一层抽象，用它来描述值的排序。标准库中的 `sort(by:)` 和 `sorted(by:)` 方法使用一个比较函数来进行排序，这个函数接受两个对象，并当它们在顺序正确时返回 `true`。我们可以通过定义一个泛型的类型别名，把它称为排序描述符 (`sort descriptor`)：

```
typealias SortDescriptor<Root> = (Root, Root) -> Bool
```

另一种替代方案是创建一个结构体来包装这些比较函数。将函数包装在结构体中的好处，是我们可以定义多个初始化方法和实例方法，使用者可以通过代码补全更容易地发现它们。是否要把一个函数包装到结构体中，最主要还是取决于个人的选择，但是在这个情况下，它使得最终的 API 感觉上更贴近于 Swift 的习惯：

```
struct SortDescriptor<Root> {
    var areInIncreasingOrder: (Root, Root) -> Bool
}
```

举个例子，我们可以定义排序描述符，来通过出生年份或者是姓的顺序来比较两个 `Person`：

```
let sortByYear: SortDescriptor<Person> = .init { $0.yearOfBirth < $1.yearOfBirth }
let sortByLastName: SortDescriptor<Person> = .init {
    $0.last.localizedStandardCompare($1.last) == .orderedAscending
}
```

相比于手写排序描述符，我们可以通过一个函数来生成它们。在一个函数里写两次相同的属性是不太好的：比如在 `sortByLastName` 里，我们可能很容易不小心让 `$0.last` 和 `$1.first` 进行比较，从而导致错误。而且，书写这些排序描述符其实是很单调的；想要按照 `first name` (名) 排序，很可能最简单的方法就是把 `sortByLastName` 复制粘贴一下，然后进行一些修改。所以首先，让我们来创建一个排序描述符，让它可以接受任意满足 `Comparable` 的属性，这可以让整个过程简单些：

```
extension SortDescriptor {
```

```
init<Value: Comparable>(_ key: @escaping (Root) -> Value) {  
    self.areInIncreasingOrder = { key($0) < key($1) }  
}  
}
```

key 函数描述了如何深入一个 Root 类型的元素，并提取出一个和特定排序步骤相关的 Value 类型的值。因为它和 Swift 的键路径 (key paths) 有很多相同之处，所以我们从 KeyPath 类型那边借鉴了 Root 和 Value，来作为泛型参数的名字。

现在，我们就可以用这个新的初始化方法来定义 sortByYear 了：

```
let sortByYearAlt: SortDescriptor<Person> = .init { $0.yearOfBirth }
```

类似地，对于那些形如 localizedStandardCompare 以及 Foundation 中其他一些返回 ComparisonResult 的情况，我们也可以为它们创建初始化方法。如果我们把 String 的部分作为泛型，这个初始化方法会是：

```
extension SortDescriptor {  
    init<Value>(_ key: @escaping (Root) -> Value,  
                 by compare: @escaping (Value) -> (Value) -> ComparisonResult) {  
        self.areInIncreasingOrder = {  
            compare(key($0))(key($1)) == .orderedAscending  
        }  
    }  
}
```

如果你查看表达式 String.localizedStandardCompare 的类型，你会注意到它是 `(String) -> (String) -> ComparisonResult`。发生了什么？在底层，实例方法会被处理为这样一个函数：如果给定某个实例，它将返回另一个可以在该实例上进行操作的函数。`someString.localizedStandardCompare` 实际上是 `String.localizedStandardCompare(someString)` 的另一种写法，两个表达式都返回

的是类型为 `(String) -> ComparisonResult` 的函数，该函数是捕获了 `someString` 的闭包。

这可以让我们用一种很简洁的方式写出 `sortByFirstName`：

```
let sortByFirstName: SortDescriptor<Person> =  
.init({ $0.first }, by: String.localizedStandardCompare)
```

当我们想要按照多个属性进行排序时，我们可以将两个排序描述符进行合并。我们可以首先使用主要的排序描述符作比较，如果两个值既不是升序，也不是降序，那么就使用第二个排序描述符的结果：

```
extension SortDescriptor {  
    func then(_ other: SortDescriptor<Root>) -> SortDescriptor<Root> {  
        SortDescriptor { x, y in  
            if areInIncreasingOrder(x,y) { return true }  
            if areInIncreasingOrder(y,x) { return false }  
            return other.areInIncreasingOrder(x,y)  
        }  
    }  
}
```

我们就可以把我们这三个排序描述符连接起来，合并形成一个单独的排序描述符了：

```
let combined = sortByLastName.then(sortByFirstName).then(sortByYear)  
people.sorted(by: combined.areInIncreasingOrder)  
/*  
[Person(first: "Ava", last: "Barnes", yearOfBirth: 1998),  
 Person(first: "Ava", last: "Barnes", yearOfBirth: 2000),  
 Person(first: "Robert", last: "Barnes", yearOfBirth: 1985),  
 Person(first: "David", last: "Gray", yearOfBirth: 1991),  
 Person(first: "Joanne", last: "Miller", yearOfBirth: 1994),  
 Person(first: "Emily", last: "Young", yearOfBirth: 2002)]  
*/
```

虽然我们的解决方法的表达能力还不如 Foundation 中的排序描述符，但是它可以针对 Swift 中的所有值，而不仅仅只是 NSObject。另外，由于我们并不依赖运行时编程，编译器将能够为我们提供更多的优化。

基于函数的方式有一个不足，那就是函数是不透明的。我们可以获取一个 NSSortDescriptor 并将它打印到控制台，我们也能从排序描述符中获得一些信息，比如键路径，selector 的名字，以及排序的升降序设定等。我们甚至可以用 NSSecureCoding 来把一个 NSSortDescriptor 进行序列化和反序列化。但是在基于函数的方式中，这些都无法做到。

把函数作为数据使用的这种方式（例如：在运行时构建包含排序函数的数组），把语言的动态行为带到了一个新的高度。这使得像 Swift 这种需要编译的静态语言也可以实现诸如 Objective-C 或 Ruby 中的一部分动态特性。

我们也看到了合并其他函数的函数的用武之地，它也是函数式编程的构建模块之一。例如，我们的 then 方法接受两个排序描述符，并将它们合并成单个的排序描述符。在很多不同的应用场景下，这项技术都非常强大。

## 函数作为代理

代理无处不在，它反复出现在 Objective-C（以及 Java）程序员的脑海中：使用协议（或者在 Java 中的接口）来做回调。你通过定义一个协议，然后让代理的所有者实现这个协议，最后将它本身注册为代理，这样你就能获得那些回调。

如果代理协议只包含单个方法，那么你肯定可以将那个存储代理对象的属性替换为直接存储回调函数的属性。不过，在这里你必须进行一些权衡。

## Cocoa 风格的代理

就像 Cocoa 中不计其数的代理协议一样，我们以仿照同样的风格创建一个协议作为开始。大部分来自 Objective-C 的程序员肯定已经写过很多遍这样的代码了：

```
protocol AlertViewDelegate: AnyObject {
    func buttonTapped(atIndex: Int)
}
```

AlertViewDelegate 是一个只有类才能实现的协议 (它继承自 AnyObject)，因为我们希望在 AlertView 中持有一个代理的弱引用。这样一来，就不用担心引用循环的问题了。AlertView 不会强引用它的代理，所以即使是代理 (直接或者间接) 强引用了 alert view，也不会出现什么问题。如果代理被析构了，delegate 属性也会自动变为 nil：

```
class AlertView {  
    var buttons: [String]  
    weak var delegate: AlertViewDelegate?  
  
    init(buttons: [String] = ["OK", "Cancel"]) {  
        self.buttons = buttons  
    }  
  
    func fire() {  
        delegate?.buttonTapped(atIndex: 1)  
    }  
}
```

这种模式在处理类的时候非常好用。假设我们有一个 ViewController 类，它将初始化 alert view，并把自己设为这个视图的代理。因为代理被标记为 weak，我们不需要担心引用循环：

```
class ViewController: AlertViewDelegate {  
    let alert: AlertView  
  
    init() {  
        alert = AlertView(buttons: ["OK", "Cancel"])  
        alert.delegate = self  
    }  
  
    func buttonTapped(atIndex index: Int) {  
        print("Button tapped: \(index)")  
    }  
}
```

将代理属性标记为 `weak` 在实践中非常常见，这个约定让内存管理变得很容易。实现代理协议的类不需要担心引入引用循环的问题。

## 使用函数，而非代理

如果代理协议中只定义了一个函数的话，我们可以用一个存储回调函数的属性来替换原来的代理属性。在我们的案例中，这可以用一个可选的 `buttonTapped` 属性来做到，默认情况下这个属性是 `nil`：

```
class AlertView {  
    var buttons: [String]  
    var buttonTapped: ((_ buttonIndex: Int) -> ())?  
  
    init(buttons: [String] = ["OK", "Cancel"]) {  
        self.buttons = buttons  
    }  
  
    func fire() {  
        buttonTapped?(1)  
    }  
}
```

对于 `(_ buttonIndex: Int) -> ()` 这个函数类型，看起来可能会有一点奇怪，因为 `buttonIndex` 这个内部名字并没有出现在代码的其他任何地方。我们在上面提到过，很遗憾目前函数类型不能有参数标签，但是它们能有一个明确写出的空类型标签，配合上一个内部的参数名字。它能让我们给函数类型的参数添加标签，用来作为文档说明。在 Swift 支持更好的方式之前，这是官方所认可的变通方式。

我们现在可以创建一个 `logger` 结构体，一个 `alert view` 实例以及一个 `logger` 变量：

```
struct TapLogger {  
    var taps: [Int] = []  
  
    mutating func logTap(index: Int) {
```

```
taps.append(index)
}
}
```

```
let alert = AlertView()
var logger = TapLogger()
```

不过，我们不能简单地将 `logTap` 方法赋值给 `buttonTapped` 属性。Swift 编译器会告诉我们“不允许部分应用 ‘可变’ 方法”：

```
alert.buttonTapped = logger.logTap // 错误
```

在上面的代码中，这个赋值的结果不明确。是 `logger` 需要复制一份呢，还是 `buttonTapped` 需要改变它原来的状态（即 `logger` 被捕获）呢？

要修正这个错误，我们需要将赋值的右侧用一个闭包封装起来。这让代码变得十分清楚，我们是想要捕获原来的 `logger` 变量（不是其中的值），然后我们将改变它：

```
alert.buttonTapped = { logger.logTap(index: $0) }
```

这么做还有一个额外的好处，就是命名现在解耦了：回调属性的名字是 `buttonTapped`，而实现它的函数叫做 `logTap`。除了使用方法以外，我们也可以指定一个匿名函数：

```
alert.buttonTapped = { print("Button \"\$0\" was tapped") }
```

当将回调和类合在一起使用时，我们有一些忠告。让我们回到我们的 `view controller` 例子中，在它的初始化方法里，`view controller` 现在可以将它的 `buttonTapped` 赋值给 `alert view` 的回调，而不是将自身赋值为 `alert view` 的代理：

```
class ViewController {
    let alert: AlertView

    init() {
        alert = AlertView(buttons: ["OK", "Cancel"])
    }
}
```

```
    alert.buttonTapped = self.buttonTapped(atIndex:  
}  
  
func buttonTapped(atIndex index: Int) {  
    print("Button tapped: \(index)")  
}  
}
```

alert.buttonTapped = self.buttonTapped(atIndex:) 这行代码看起来是个无害的赋值语句，但是小心：我们刚刚创建了一个引用循环！所有指向某个对象的实例方法的引用（比如这个例子中的 self.buttonTapped）都会在背后捕获这个对象。要理解为什么一定要这样做，可以站在 alert view 的视角来考虑问题：当 alert view 要调用存储在它的 buttonTapped 属性中的回调函数时，这个函数必须“知道”它到底需要调用哪个对象的实例方法 - 因此不光要存储一个指向 ViewController.buttonTapped(atIndex:) 的引用，还需要存储实例本身。

我们可以把 self.buttonTapped(atIndex:) 简写为 self.buttonTapped 或者单写 buttonTapped；所有三种写法都说的是同一个函数。只要没有产生歧义，参数标签就可以被省略。

想要避免强引用，通常我们需要将方法调用包装在另一个闭包中，这个闭包通过弱引用的方式捕获对象：

```
alert.buttonTapped = { [weak self] index in  
    self?.buttonTapped(atIndex: index)  
}
```

这样一来，alert view 就不会强引用 view controller 了。如果我们能保证 alert view 的生命周期和 view controller 绑定的话，另一个选项是使用 unowned 来替代 weak。使用 weak 时，当 alert view 的生命周期超过 view controller 时，当函数被调用时，闭包里的 self 将为 nil。

正如我们所看到的，在使用协议和回调函数之间，一定是存在权衡的。协议的方式比较啰嗦，但是一个只针对类的协议配合 weak 代理可以让我们完全不用担心引用循环。将代理用函数替

代可以带来更多的灵活性，让我们可以使用结构体和匿名函数。不过，当处理类的时候，你需要特别小心，避免引入引用循环。

另外，当你需要多个紧密相连的回调函数（比如，为一个 table view 提供数据）的时候，最好将它们组织在一个协议里，而不是去使用单个回调。另一方面，当使用协议的时候，遵守协议的类型需要实现其中的所有方法。

要注销一个代理或者函数回调，我们可以简单地将它设为 nil。但如果我们的类型是用一个数组来存储代理或者回调呢？对于基于类的代理，我们可以直接将它从代理列表中移除；不过对于回调函数，就没那么简单了，因为函数不能被比较，所以我们需要添加额外的逻辑去进行移除。

## inout 参数和可变方法

如果你有些 C 或 C++ 的背景，Swift 中用在 inout 参数前面的 & 可能会给你一种这是在传递引用的错觉。但事实并非如此，inout 做的事情是传值，然后复制回来，并不是传递引用。引用官方《[Swift 编程语言](#)》中的话：

一个 in-out 参数持有一个传入函数的值，函数可以改变这个值，然后从函数中传出并替换掉原来的值。

想要了解什么样的表达式可以作为 inout 参数，我们需要区分 lvalue 和 rvalue。lvalue 描述的是一个内存地址，它是“左值(left value)”的缩写，因为 lvalues 是可以存在于赋值语句左侧的表达式。举例来说，array[0] 是一个 lvalue，因为它描述了数组中第一个元素所在的内存位置。而 rvalue 描述的是一个值。2 + 2 是一个 rvalue，它描述的是 4 这个值。你不能把 2 + 2 或者 4 放到赋值语句的左侧。

对于 inout 参数，你只能传递左值，因为右值是不能被修改的。当你在普通的函数或者方法中使用 inout 时，需要显式地将它们传入：即在每个左值前面加上 & 符号。例如，当调用 increment 时（它有一个 inout int 参数），我们就要在传入的变量前添加 &：

```
func increment(value: inout Int) {
    value += 1
}
```

```
var i = 0
increment(value: &i)
```

如果用 `let` 定义一个变量的话，它就不能被用作一个 `lvalue` 了。因为我们不能改变 `let` 变量，所以将它用作 `inout` 也是没有意义的，我们只能使用那些“可更改”的 `lvalue`：

```
let y: Int = 0
increment(value: &y) // 错误
```

除了变量，还有不少东西都是 `lvalue`。举个例子，如果数组是用 `var` 定义的话，我们可以传入数组下标：

```
var array = [0, 1, 2]
increment(value: &array[0])
array // [1, 1, 2]
```

实际上，所有的下标操作符（包括那些你自定义的），只要它们同时定义了 `get` 和 `set` 方法，也都可以作为 `inout` 参数。类似的，只要同时定义了 `get` 和 `set` 方法，属性也可以作为左值：

```
struct Point {
    var x: Int
    var y: Int
}
var point = Point(x: 0, y: 0)
increment(value: &point.x)
point // Point(x: 1, y: 0)
```

如果一个属性是只读的（也就是说，只有 `get` 可用），我们将不能将其用于 `inout` 参数：

```
extension Point {
    var squaredDistance: Int {
        return x*x + y*y
    }
}
```

```
increment(value: &point.squaredDistance) // 错误
```

运算符也可以接受 inout 值，但是为了简化，在调用时我们不需要加上 & 符号，简单地使用 lvalue 就可以了。比如，自增运算符在 Swift 3 中被移除了，不过我们可以自己把它加回来：

```
postfix func ++(x: inout Int) {
    x += 1
}
point.x++
point // Point(x: 2, y: 0)
```

可变运算符甚至还可以与可选链一起使用。这里，我们将自增操作连接到字典下标访问后：

```
var dictionary = ["one": 1]
dictionary["one"]?++
dictionary["one"] // Optional(2)
```

注意，在字典查找返回 nil 的时候，++ 操作符不会被执行。

编译器可能会把 inout 变量优化成引用传递，而非传入和传出时的复制。不过，文档已经明确指出我们不应该依赖这个行为。

我们会在后面结构体和类一章中回到 inout 上，在那里我们将看看 mutating 方法和接受 inout 参数的函数之间的相似之处。

## 嵌套函数和 inout

你可以在嵌套函数中使用 inout 参数，Swift 会保证你的使用是安全的。比如说，你可以定义一个嵌套函数（使用 func 或者使用闭包表达式），然后安全地改变一个 inout 的参数：

```
func incrementTenTimes(value: inout Int) {
    func inc() {
        value += 1
    }
}
```

```
for _ in 0..<10 {  
    inc()  
}  
}  
  
var x = 0  
incrementTenTimes(value: &x)  
x // 10
```

不过，你不能够让这个 `inout` 参数逃逸（我们会在本章稍后详细讨论逃逸函数的内容）：

```
func escapeIncrement(value: inout Int) -> () -> () {  
    func inc() {  
        value += 1  
    }  
    // error: 嵌套函数不能捕获 inout 参数然后让其逃逸  
    return inc  
}
```

可以这么理解，因为 `inout` 的值会在函数返回之前复制回去。那么要是我们可以在函数返回之后再去改变它，会发生什么呢？是说值应该在某个时间点再复制回去吗？要是调用源已经不存在了怎么办？编译器必须对此进行验证，因为这对保证安全十分关键。

## & 不意味 `inout` 的情况

说到不安全 (`unsafe`) 的函数，你应该小心 `&` 的另一种含义：把一个函数参数转换为一个不安全指针。

如果一个函数接受 `UnsafeMutablePointer` 作为参数，你可以用和 `inout` 参数类似的方法，在一个 `var` 变量前面加上 `&` 传递给它。在这种情况下，你确实在传递引用，更确切地说，是在传递指针。

这里是一个没有使用 `inout`，而是接收不安全的可变指针作为参数的 `increment` 函数的例子：

```
func incref(pointer: UnsafeMutablePointer<Int>) -> () -> Int {  
    // 将指针的的复制存储在闭包中  
    return {  
        pointer.pointee += 1  
        return pointer.pointee  
    }  
}
```

我们会在后面的章节介绍，Swift 的数组可以无缝地隐式退化为指针，这使得将 Swift 和 C 一起使用的时候非常方便。现在，假设在调用这个函数之前，你传入的数组已经离开其作用域了：

```
let fun: () -> Int  
do {  
    var array = [0]  
    fun = incref(pointer: &array)  
}  
/*_*/fun()
```

这个操作为我们打开了充满“惊喜”的未知世界的大门。在测试的时候，每次运行上面的代码都将打印出不同的值，有时候是 0，有时候是 1，有时候是 140362397107840，有时候就直接崩溃了。

这个例子告诉我们的就是：了解并确定你正在传递的参数。当你使用 `&` 时，它代表的既有可能是 Swift 安全且优秀的 `inout` 语义，但也可能把你的变量带到不安全指针的荒蛮之地。当处理不安全的指针时，你需要非常小心变量的生命周期。我们会在 [互用性](#) 章节里深入这方面的内容。

## 下标

在标准库中，我们已经看到过一些下标用法了，例如：用 `dictionary[key]` 这样的方式在字典查找元素。这些下标很像函数和属性的混合体，只不过它们使用了特殊的语法。之所以像函数，是因为它们也可以接受参数；之所以像计算属性，是因为它们要么是只读的（只提供 `get`），要么是可读写的（同时提供 `get` 和 `set`）。和普通的函数类似，我们可以通过重载提供不同类型的下标

操作符。比如，数组默认有两个下标操作，一个用来访问单个元素，另一个用来返回一个切片（更精确地说，它们是被定义在 Collection 协议中的）：

```
let fibs = [0, 1, 1, 2, 3, 5]
let first = fibs[0] // 0
fibs[1..<3] // [1, 1]
```

## 自定义下标操作

我们可以为自己的类型添加下标支持，也可以为已经存在的类型添加新的下标重载。举个例子，让我们给 Collection 添加一个接受索引列表为参数的下标方法，它返回一个包含这些索引位置上的元素的数组：

```
extension Collection {
    subscript(indices indexList: Index...) -> [Element] {
        var result: [Element] = []
        for index in indexList {
            result.append(self[index])
        }
        return result
    }
}
```

请注意我们是如何使用一个显式的参数标签，来将我们的下标方法和标准库中的方法区分开来的。三个点表示 indexList 是一个可变长度参数 (**variadic parameter**)。调用者可以传入零个或多个以逗号分隔的指定类型的值（在这里是 Collection 的 Index 类型）。在函数中，这个参数将被作为数组来使用。

我们可以像这样使用这个新的下标操作符：

```
Array("abcdefghijklmnopqrstuvwxyz")[indices: 7, 4, 11, 11, 14]
// ["h", "e", "l", "l", "o"]
```

## 下标进阶

下标并不局限于单个参数。我们已经看到过多个参数的下标例子了：在字典中，下标还可以接受一个键和一个默认值。如果你对此感兴趣，可以看看 Swift 源码中[它的实现](#)。

下标还可以在参数或者返回类型上使用泛型。考虑下面这个类型为 [String: Any] 的异值字典：

```
var japan: [String: Any] = [
    "name": "Japan",
    "capital": "Tokyo",
    "population": 126_740_000,
    "coordinates": [
        "latitude": 35.0,
        "longitude": 139.0
    ]
]
```

如果你想要更改字典中某个嵌套值，比如上例中 coordinates 里的 latitude 的话，你会发现这其实没那么简单：

```
// 错误：类型 'Any' 没有下标成员
japan["coordinate"]?["latitude"] = 36.0
```

这可以理解，因为 japan["coordinate"] 的类型为 Any?，所以你可能会去尝试在使用嵌套下标之前先将它的类型转为字典：

```
// 错误：不能对不可变表达式赋值
(japan["coordinates"] as? [String: Double])?["coordinate"] = 36.0
```

啊啦，不仅代码很快变得很丑，而且它也依然无法工作。问题在于你不能修改一个类型转换后的变量 - japan["coordinates"] as? [String: Double] 这个表达式已经不再是一个左值了。你需要先将这个嵌套的字典存储到一个局部变量中，修改它，然后再把这个变量赋值回顶层的键。

我们可以通过为 Dictionary 提供一个泛型下标的扩展，来更好地完成这件事。这个下标方法的第二个参数接受目标类型，并且在下标实现中进行类型转换的尝试：

```
extension Dictionary {
```

```
subscript<Result>(key: Key, as type: Result.Type) -> Result? {
    get {
        return self[key] as? Result
    }
    set {
        // 如果传入 nil, 就删除现存的值。
        guard let value = newValue else {
            self[key] = nil
            return
        }
        // 如果类型不匹配, 就忽略掉。
        guard let value2 = value as? Value else {
            return
        }
        self[key] = value2
    }
}
```

因为我们不再需要将下标返回的值做向下的类型转换了, 所以更改操作可以直接在顶层字典变量中进行:

```
japan["coordinates", as: [String: Double].self]?["latitude"] = 36.0
japan["coordinates"] // Optional([{"latitude": 36.0, "longitude": 139.0}])
```

泛型下标方法为类型系统填上了一个大洞。不过, 你可能会觉得这个例子最终的语法还是有些丑陋。基本上来说, Swift 并不适合用来处理像上面字典这样的异值集合。在大多数情况下, 可能为你的数据定义一个自定义类型(比如这里可以定义一个 Country 结构体), 然后让这些类型满足 Codable 协议, 来在值和数据交换格式之间进行转换, 会是更好的选择。

## 自动闭包

我们都对“逻辑与”，也就是 `&&` 操作符如何对其参数求值很熟悉了：它会先对左边的操作数求值，如果左边的求值为 `false` 时，则直接返回。只有当左侧值为 `true` 时，右边的操作数才会被求值。这是因为，一旦左边的结果是 `false` 的话，整个表达式就不可能是 `true` 了。这种行为又被叫做短路求值。举个例子，如果我们想要检查数组的第一个元素是否满足某个要求，我们可以这样做：

```
let evens = [2,4,6]
if !evens.isEmpty && evens[0] > 10 {
    // 执行操作
}
```

在上面的代码中，我们依赖了短路求值：对于数组的访问仅仅发生在第一个条件满足时。如果没有条件短路的话，代码将会在空数组的时候发生崩溃。

对这个特定的例子，使用 `if let` 绑定是更好的写法：

```
if let first = evens.first, first > 10 {
    // 执行操作
}
```

这是另一种形式的短路求值：第二个条件只有在第一个条件成功后，才会进行判断。

在几乎所有的语言中，对于 `&&` 和 `||` 操作符的短路求值都是内建在语言中的。想要定义一个你自己的带有短路逻辑的操作符或者方法往往是不可能的。如果一门语言支持头等函数，我们就可以通过提供匿名函数而非值的方式，来模拟短路求值操作。比如，我们想在 Swift 中定义一个和 `&&` 操作符具有相同功能的 `and` 函数：

```
func and(_ l: Bool, _ r: () -> Bool) -> Bool {
    guard l else { return false }
    return r()
}
```

上面的函数首先对 `l` 进行检查，如果 `l` 的值为 `false` 的话，就直接返回 `false`。只有当 `l` 是 `true` 的时候，才会返回闭包 `r` 的求值结果。但它要比 `&&` 操作符的使用复杂一些，因为右边的操作数现在必须是一个函数：

```
if and(!evens.isEmpty, { evens[0] > 10 }) {
    // 执行操作
}
```

在 Swift 中有一个很好的特性，能让代码更漂亮。我们可以使用 `@autoclosure` 标注来告诉编译器它应该将一个特定的参数用闭包表达式包装起来。通过这种方式构建的 `and` 的定义和上面几乎一样，除了在 `r` 参数前加上了 `@autoclosure` 标注：

```
func and(_ l: Bool, _ r: @autoclosure () -> Bool) -> Bool {
    guard l else { return false }
    return r()
}
```

`and` 的使用现在就要简单得多了，因为我们不再需要将第二个参数封装到闭包中了。我们只需要像使用普通的 `Bool` 参数那样来调用它，编译器将“透明地”把参数包装到闭包表达式中：

```
if and(!evens.isEmpty, evens[0] > 10) {
    // 执行操作
}
```

这让我们可以使用短路操作的行为定义我们自己的函数和运算符。比方说，像是 `??` 和 `!?` (我们在可选值一章中定义过这个运算符) 这样的运算符现在可以直接编码实现。在标准库中，像是 `assert` 和 `precondition` 这样的函数，为了只在确实需要时才对参数进行求值，也使用了自动闭包。对断言条件进行求值的时机，从调用时被推迟到了 `assert` 函数的内部，这可以将耗时的操作完全从优化后的版本中移除掉，因为我们并不需要它们出现在最终的发布版中。

自动闭包在实现日志函数的时候也很有用。比如，下面是一个只在条件为 `true` 的时候才会对日志消息进行求值的 `log` 函数：

```
func log(ifFalse condition: Bool,
         message: @autoclosure () -> (String),
```

```
file: String = #fileID, function: String = #function, line: Int = #line)
{
    guard !condition else { return }
    print("Assertion failed:\(message()), \(file):\(\(function)) (\(line) \(\(line)))")
}
```

这意味着你可以在传入的表达式中进行昂贵的计算，而不必担心在这个值没有使用时所带来的开销。这个 log 函数使用了像是 #fileID, #function 和 #line 这样的调试标识符。当被用作一个函数的默认参数时，它们代表的值分别是调用者所在的文件名、函数名以及行号，这会非常有用。

不过请谨慎使用自动闭包特性。它们的行为与一般的期望有冲突 - 比如，要是某个表达式被自动闭包包装的话，它有可能不被执行，而导致其中的某些副作用没有生效。引用 Apple 的 Swift 书中的一段：

过度使用自动闭包可能会让你的代码难以理解。使用时的上下文和函数名应该清晰地指出实际求值会被推迟。

## @escaping 标注

你有可能已经注意到了，在一些闭包表达式中，编译器要求你显式地访问 self，而在另外一些表达式中却不需要这样。例如，我们需要在一个网络请求的完成回调中显式地使用 self，但是在传递给 map 或 filter 的闭包中却不需要这样做。两者的不同在于是否会为了稍后的使用而把闭包保存下来（正如网络请求），或者说闭包是否只在函数的作用域中被同步调用（就像 map 和 filter 这样）。

一个被保存在某个地方（比如一个属性中）等待稍后再调用的闭包就叫做逃逸闭包。相对的，永远不会离开一个函数的局部作用域的闭包就是非逃逸闭包。对于逃逸闭包，编译器强制我们在闭包表达式中显式地使用 self，因为无意中对于 self 的强引用，是发生引用循环的最常见原因之一。当一个函数返回的时候，非逃逸闭包会自动销毁，所以它不会创建一个固定的引用循环。

闭包参数默认是非逃逸的。如果你想要保存一个闭包稍后再用，你需要将闭包参数标记为 @escaping。编译器将会对此进行验证，如果你没有将闭包标记为 @escaping，编译器将不允许你保存这个闭包（或者比如将它返回给调用者）。

在排序描述符的例子中，我们已经看到过几个必须使用 @escaping 的函数参数了：

```
extension SortDescriptor {  
    init<Value: Comparable>(_ key: @escaping (Root) -> Value) {  
        self.areInIncreasingOrder = { key($0) < key($1) }  
    }  
}
```

注意默认非逃逸的规则只对函数参数，以及那些直接参数位置 (immediate parameter position) 的函数类型有效。也就是说，如果一个存储属性的类型是函数的话，那么它将会是逃逸的（这很正常）。出乎意料的是，对于那些使用闭包作为参数的函数，如果闭包被封装到像是元组或者可选值等类型的话，这个闭包参数也是逃逸的。因为在这种情况下闭包不是直接参数，它将自动变为逃逸闭包。这样的结果是，你不能写出一个函数，使它接受的函数参数同时满足可选值和非逃逸。很多情况下，你可以通过为闭包提供一个默认值来避免可选值。如果这样做行不通的话，可以通过重载函数，提供一个包含可选值（逃逸）的函数，以及一个不是可选值，非逃逸的函数来绕过这个限制：

```
func transform(_ input: Int, with f: ((Int) -> Int)?) -> Int {  
    print("使用可选值重载")  
    guard let f = f else { return input }  
    return f(input)  
}  
  
func transform(_ input: Int, with f: (Int) -> Int) -> Int {  
    print("使用非可选值重载")  
    return f(input)  
}
```

这样一来，如果用 nil 参数（或者一个可选值类型的变量）来调用函数，将使用可选值变种，而如果使用闭包字面量的调用将使用非逃逸和非可选值的重载方法：

```
_ = transform(10, with: nil) // 使用可选值重载  
_ = transform(10) { $0 * $0 } // 使用非可选值重载
```

## withoutActuallyEscaping

可能你会遇到这种情况：你确实知道一个闭包不会逃逸，但是编译器无法证明这点，所以它会强制你添加 `@escaping` 标注。为了说明这点，让我们看一个标准库文档中的例子。我们在 `Array` 上实现一个自定义的 `allSatisfy` 方法，这个方法在内部使用一个数组的 `延迟视图` (`lazy view`) (不要和我们在 属性 一章中要讨论的 `延迟属性` 搞混了)。然后我们在这个 `延迟视图` 上应用一个 `filter` 来检查是否有任何的元素满足 `filter` 的条件 (也就是说至少有一个元素不满足断言)。我们的首次尝试导致了一个编译错误：

```
extension Array {  
    func allSatisfy2(_ predicate: (Element) -> Bool) -> Bool {  
        // 错误: 使用非逃逸参数 'predicate' 的闭包有可能允许它逃逸。  
        return self.lazy.filter({ !predicate($0) }).isEmpty  
    }  
}
```

我们会在集合类型协议中再讨论 `延迟集合 API`。对于现在来说，你只需要知道 `延迟视图` 为了之后使用后续的转换 (比如说这里传递给 `filter` 的闭包) 会把它们保存到一个内部的属性中去。这就要求传入的任何闭包都是逃逸的，这也是这个错误的原因，因为我们的 `predicate` 参数是非逃逸的。

我们可以通过把参数标注成 `@escaping` 来解决这个问题，但在这个情况下，我们知道这个闭包是不会逃逸的，因为这个 `延迟集合视图` 的生命周期是同这个函数的生命周期绑定的。对于类似这样的情况，Swift 提供了一个 `withoutActuallyEscaping` 函数来作为一种“安全出口”。这个函数允许你对一个接受逃逸闭包的函数，传入一个非逃逸的闭包。这下可以编译通过并正常工作了：

```
extension Array {  
    func allSatisfy2(_ predicate: (Element) -> Bool) -> Bool {  
        return withoutActuallyEscaping(predicate) { escapablePredicate in
```

```
self.lazy.filter { !escapablePredicate($0) }.isEmpty  
}  
}  
}  
  
let areAllEven = [1,2,3,4].allSatisfy2 { $0 % 2 == 0 } // false  
let areAllOneDigit = [1,2,3,4].allSatisfy2 { $0 < 10 } // true
```

请注意，上面这个使用延迟的实现，并不比标准库的 allSatisfy 实现更高效，标准库的实现是使用了一个简单的 for 循环，并不需要使用 withoutActuallyEscaping。使用延迟的实现仅用于演示这种情况，也就是我们知道闭包不会逃逸但编译器无法证明。

注意，使用 withoutActuallyEscaping 后，你就进入了 Swift 中不安全的领域。让闭包的复制从 withoutActuallyEscaping 调用的结果中逃逸的话，会造成不确定的行为。

## Result Builder

Result builder 是一种特殊的函数，它允许我们通过简洁和富有表达力的多个语句构建出结果值。

最显著的例子，也可以说这个 Swift 特性被引入的动机，是 SwiftUI 的 view builder 语法。使用 view builder，你可以像下面的水平 stack view 这样来定义内容：

```
HStack {  
    Text("Finish the Advanced Swift Update")  
    Spacer()  
    Button("Complete") { /* ... */ }  
}
```

虽然没有以任何特殊方式进行标记，不过 HStack 的尾随闭包就是一个 result builder 函数。我们可以在 HStack 的初始化方法中找到这个 @ViewBuilder 标记。我们在这里为了举例，稍微对它进行了一些简化：

```
struct HStack<Content>: View where Content: View {
```

```
public init(  
    alignment: VerticalAlignment = .center,  
    spacing: CGFloat? = nil,  
    @ViewBuilder content: () -> Content)  
// ...  
}
```

在上面的 view builder 中，我们在三行里写了三个表达式。在普通的 Swift 代码里，这些代码没有任何意义：我们没有对这些表达式的值做任何事情，它们也没有任何副作用。但是，在一个 result builder 函数中，编译器会重写这些代码，用函数中的所有语句来创建一个合成的值。为了做到这一点，编译器会使用定义在对应 result builder 类型（比如本例中的 ViewBuilder）上的一些静态方法。

重写上面的例子时，编译器将使用 ViewBuilder 的静态 buildBlock 方法，它接受三个满足 View 协议的参数：

```
@resultBuilder public struct ViewBuilder {  
    // ...  
    public static func buildBlock<C0, C1, C2>(_ c0: C0, _ c1: C1, _ c2: C2)  
        -> TupleView<(C0, C1, C2)>  
    where C0: View, C1: View, C2: View  
    // ...  
}
```

重写后的代码将不再依赖于 result builder，类似这样：

```
HStack {  
    return ViewBuilder.buildBlock(  
        Text("Finish the Advanced Swift Update"),  
        Spacer(),  
        Button("Complete") { /* ... */ }  
    )  
}
```

Result builder 类型被 @resultBuilder 标记，它可以实现一系列不同的静态 build... 方法，我们在本节剩余的部分将会检查更多细节。哪些方法被实现，将决定在 result builder 函数中可以使用哪些类型的表达式和语句。

## Block 和表达式

最基本的构建方法是 buildBlock 和 buildExpression。和所有构建方法一样，它们是静态方法。Builder 类型只是作为命名空间来使用，它们从不会被实例化。

要实现一个 @resultBuilder 类型，只有唯一一个要求，那就是要至少实现一个 buildBlock 方法。因为构建方法允许把多个中间结果合并成一个结果值，所以大部分时候，参数的个数以及它们的类型可能会不同，你会需要实现不止一个的 buildBlock 变体。例如，view builder 的 buildBlock 方法就有从零至十个参数的变体，这让 view builder 函数可以接受从零至十的 view，并把它们合并成一个 TupleView。

SwiftUI 的 buildBlock 方法只接受 View 的值作为参数，但是 buildBlock 并没有限制自身要求只能接受一种类型的参数。我们可以按照不同的参数类型写出多种重载，不过要在 builder 函数中支持多种类型的话，实现 buildExpression 通常是更优雅的方式。

buildExpression 并不是 builder 类型所要求实现的，但是它在支持不同类型的表达式时非常有用。当我们实现了这个方式的一个或多个变体时，Swift 会首先将 buildExpression 应用到 builder 函数的每个表达式上，然后再把中间结果传递给 buildBlock。

buildExpression 函数接受一个参数，但是你可以实现多种变体来支持不同的参数类型。返回值的类型只要能被某个 buildBlock 方法所支持的话，就可以是任意类型。

在我们研究 result builder 类型上可以实现的其他 build... 方法之前，我们先来用 buildBlock 和 buildExpression 创建我们自己的 result builder 类型。作为例子，我们会实现一个构建字符串的 result builder。这个字符串 builder 类型的最简单的形式看起来是这样的：

```
@resultBuilder
struct StringBuilder {
    static func buildBlock() -> String {
        ""
    }
}
```

```
    }  
}
```

通过实现不带参数的 `buildBlock`, 我们支持了空字符串的构建函数, 比如:

```
@StringBuiler func build() -> String {  
}
```

```
build() //
```

执行这个函数的结果是一个空字符串。在底层, 编译器将 `build` 函数重写为:

```
func build_rewritten() -> String {  
    StringBuilder.buildBlock()  
}
```

为了支持从多个字符串构建一个字符串, 我们需要添加一个接受可变数量字符串参数的 `buildBlock` 变体 (可变参数也对应了 0 个参数的情况, 因此我们可以把上面的无参数方法替换掉):

```
static func buildBlock(_ strings: String...) -> String {  
    strings.joined()  
}
```

可变参数用一个方法就涵盖了从 0 到任意数量的字符串参数。在我们的例子中, 因为所有的参数都是相同类型的, 所以可以用这样的方式进行处理。对比 SwiftUI 的 `view builder`, 它为不同的参数数量分别提供了 `buildBlock` 的重载, 所以它被人为地限制在了十个 `view` (因为 Apple 决定到此为止)。SwiftUI 之所以这么做, 是因为每个参数都可能是不同的类型 (它们都满足 `View`) 而框架希望保留这些类型信息。我们希望未来的 Swift 版本能够支持可变泛型参数, 这个特性将会让 SwiftUI (以及其他框架) 可以以可变参数重载的方式提供一个接受泛型参数的 `buildBlock`。

现在, 我们可以这样来写字符串 `builder` 的函数了:

```
@StringBuilder func greeting() -> String {  
    "Hello,"  
    "World!"  
}
```

`greeting()` // Hello, World!

Swift 将这个字符串 builder 函数翻译成下面的代码：

```
func greeting_rewritten() -> String {  
    StringBuilder.buildBlock(  
        "Hello,",  
        "World!"  
    )  
}
```

接下来，我们可以通过实现 `buildExpression` 来为其他类型添加支持，比如整型数：

```
static func buildExpression(_ s:String) -> String {  
    s  
}  
  
static func buildExpression(_ x:Int) -> String {  
    "\u{1d64}"  
}
```

注意，就算 `buildBlock` 中有可用的对应版本的方法，我们还是必须为我们想要支持的所有表达式类型实现 `buildExpression`。现在我们可以无缝地将字符串和整数混合起来构建一个字符串结果了：

```
let planets = [  
    "Mercury", "Venus", "Earth", "Mars", "Jupiter",  
    "Saturn", "Uranus", "Neptune"  
]
```

```
@StringBuilder func greetEarth() -> String {  
    "Hello, Planet "  
    planets.firstIndex(of: "Earth")!  
    "!"  
}  
  
greetEarth() // Hello, Planet 2!
```

下面是这个例子重写后的代码：

```
func greetEarth_rewritten() -> String {  
    StringBuilder.buildBlock(  
        StringBuilder.buildExpression("Hello, Planet "),  
        StringBuilder.buildExpression(planets.firstIndex(of: "Earth")!),  
        StringBuilder.buildExpression("!")  
    )  
}
```

## 重载 Builder 方法

在上面我们已经看到了，我们可以为 `buildBlock` 或 `buildExpression` 写出多个重载方法。大多数情况下，我们在 `builder` 函数中提供重载来支持不同的类型。比如，我们用不同的参数类型实现了多个 `buildExpression` 方法，让它在字符串 `builder` 中同时支持字符串和整数；SwiftUI 实现了多个变体的 `buildBlock` 来支持将不同数量的 `view` 进行组合。然而，重载也能被用来启用其他一些有用的特性。

例如，在默认情况下，是不可能在 `builder` 函数中使用 `fatalError` 语句的，因为它会被视为一个 `Never` 类型的表达式。我们可以通过实现一个 `Never` 类型的 `buildExpression` 变体来修正这个问题（不过不幸的是，这个方法不太实用，因为编译器会把这个实现标记为“永远不会执行”，并给出一个无法关闭的警告，你肯定不会想要在你的代码里一直看到这个警告）：

```
static func buildExpression(_ x: Never) -> String {  
    fatalError()  
}
```

同样地，我们不能在 `builder` 函数中使用 `print` 语句，因为 `print` 的返回类型是 `Void`。要允许使用 `print` 语句，我们可以添加一个接受 `void` 参数的 `buildExpression` 变体，它不影响最后构建出的结果：

```
static func buildExpression(_ x: Void) -> String {  
    """  
}  
}
```

此外，`buildExpression` 的重载也可以用来提供更加清晰的编译诊断信息。比如，我们可以添加第二个更泛用的 `buildExpression` 变体，让它为不支持的类型值提供一个清晰的错误信息：

```
@available(*, unavailable,  
    message: "String Builders only support string and integer values")  
static func buildExpression<A>(_ expression: A) -> String {  
    """  
}  
}
```

因为已经存在的 `String` 和 `Int` 重载方法相比起这个泛型函数来说更加具体，编译器只会在输入为我们不想提供支持的那些类型时才选用这个“错误”的变体。在本例里，`@available` 标志将这个实现标记为不可用，并附带一条自定义的错误信息。

## 条件语句

到目前为止，我们的字符串 `builder` 几乎就只是字符串插值（我们会在字符串一章里讨论）的另一套语法。通过向 `builder` 类型添加 `buildIf` 和 `buildEither` 方法，我们可以把它的能力进行拓展，让它支持 `if`, `if...else` 和 `switch` 语句。

我们从实现 `buildIf` 方法开始，它可以支持不带 `else` 分支的简单 `if` 语句。`buildIf` 的参数是一个可选值，如果条件不满足的话，这个参数会是 `nil`:

```
static func buildIf(_ s: String?) -> String {  
    s ?? ""  
}
```

这可以让我们把上面的例子重写为：

```
@StringBuilder func greet(planet: String) -> String {  
    "Hello, Planet"  
    if let idx = planets.firstIndex(of: planet) {  
        " "  
        idx  
    }  
    "  
}  
  
greet(planet: "Earth") // Hello, Planet 2!  
greet(planet: "Sun") // Hello, Planet!
```

现在我们向 builder 函数中引入了条件，重写的版本开始变得更有意思了：

```
func greet_rewritten(planet: String) -> String {  
    let v0 = "Hello, Planet"  
    var v1: String?  
    if let idx = planets.firstIndex(of: planet) {  
        v1 = StringBuilder.buildBlock(  
            StringBuilder.buildExpression(" "),  
            StringBuilder.buildExpression(idx)  
        )  
    }  
    let v2 = StringBuilder.buildIf(v1)  
    return StringBuilder.buildBlock(v0, v2)  
}
```

首先，为了暂时存储中间结果，编译器声明了一个可选值变量（像是 v0 和 v1 这样的变量名字都是我们为了在例子中为了简化说明而选取的通用名）。Swift 接下来用我们曾经看到过的方式来把条件中的语句进行重写：buildExpression 会为每个表达式进行调用，接着所有由 buildExpression 产生的中间结果会被 buildBlock 调用。最后，条件部分的中间结果被用来调用 buildIf。

想要更好地处理 greeting 函数中输入无效的情况，我们可以添加 if ... else 支持。为此，我们必须实现 buildEither(first:) 和 buildEither(second:) 这两个 builder 方法。

buildEither(first:) 会被第一个分支的结果调用，而 buildEither(second:) 将被第二个分支的结果调用。如果有多个链式的 if ... else 语句，编译器会把它们呈现为内嵌的 buildEither 调用。比如，下面这样的条件语句：

```
if x == 0 {  
    // ...  
} else if x < 10 {  
    // ...  
} else {  
    // ...  
}
```

我们可以像下面这样写出同样的语句：

```
if x == 0 {  
    // ...  
} else {  
    if x < 10 {  
        // ...  
    } else {  
        // ...  
    }  
}
```

现在两个 if 语句都有一个 else 分支了，它们将会被重写成 buildEither 的 first 和 second 变体。

对于字符串 builder 的例子，buildEither 的实现非常简单，我们只需要从第一个或者第二个分支中返回中间结果就可以了：

```
static func buildEither(first component: String) -> String {
```

```
    component
}

static func buildEither(second component: String) -> String {
    component
}
```

在其他使用情景中，这两个变体可以用来区分结果是如何在 if 或 else 分支中被变形的。比如，可以保留下关于是哪个分支被执行了的信息。

通过增加对 if ... else 的支持 (同时 switch 也支持了)，我们可以扩展我们的例子了：

```
@StringBuilder func greet2(planet: String) -> String {
    "Hello,"
    if let idx = planets.firstIndex(of: planet) {
        switch idx {
            case 2:
                "World"
            case 1, 3:
                "Neighbor"
            default:
                "planet "
                idx + 1
        }
    } else {
        "unknown planet"
    }
    "!"
}

greet2(planet: "Earth") // Hello, World!
greet2(planet: "Mars") // Hello, Neighbor!
greet2(planet: "Jupiter") // Hello, planet 5!
```

```
greet2(planet: "Pluto") // Hello, unknown planet!
```

由 Swift 编译器为上面的 greeting 函数重写的代码已经变得相当复杂了：

```
func greet2_rewritten(planet: String) -> String {
    let v0 = StringBuilder.buildExpression("Hello, ")
    let v1: String
    if let idx = planets.firstIndex(of: planet) {
        let v1_0: String
        switch idx {
            case 2:
                v1_0 = StringBuilder.buildEither(first:
                    StringBuilder.buildBlock(StringBuilder.buildExpression("World")))
            )
            case 1, 3:
                v1_0 = StringBuilder.buildEither(second:
                    StringBuilder.buildEither(first:
                        StringBuilder.buildBlock(StringBuilder.buildExpression("Neighbor")))
                )
            default:
                let v1_0_0 = StringBuilder.buildExpression("planet")
                let v1_0_1 = StringBuilder.buildExpression(idx + 1)
                v1_0 = StringBuilder.buildEither(second:
                    StringBuilder.buildEither(second:
                        StringBuilder.buildBlock(v1_0_0, v1_0_1)
                    )
                )
            }
        v1 = StringBuilder.buildEither(first: v1_0)
    } else {
        v1 = StringBuilder.buildEither(second:
            StringBuilder.buildBlock(

```

```
        StringBuilder.buildExpression("unknown planet")
    )
)
}
let v2 = StringBuilder.buildExpression("!")
return StringBuilder.buildBlock(v0, v1, v2)
}
```

## 循环

在 result builder 函数中，还有一种我们可以使用的语句，那就是 for...in 循环。为了使用 for 循环，我们需要为 builder 类型添加 buildArray 方法：

```
static func buildArray(_ components: [String]) -> String {
    components.joined(separator: "")
}
```

这可以让我们编写这样的循环：

```
@StringBuilder func greet3(planet: String?) -> String {
    "Hello"
    if let p = planet {
        p
    } else {
        for p in planets.dropLast() {
            "\((p), "
        }
        "and \((planets.last!)!"
    }
}

greet3(planet: nil)
// Hello Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, and Neptune!
```

Swift 会获取循环中每次迭代的中间结果，并将它们收集在一个数列中。这个数列之后被传递给 buildArray 用来为整个循环构建一个中间结果：

```
func greet3_rewritten(planet: String?) -> String {
    let v0 = StringBuilder.buildExpression("Hello ")
    let v1: String
    if let p = planet {
        v1 = StringBuilder.buildBlock(StringBuilder.buildExpression(p))
    } else {
        var v1_0: [String] = []
        for p in planets.dropLast() {
            let v1_0_0 = StringBuilder.buildBlock(
                StringBuilder.buildExpression("\\"(p), ")
            )
            v1_0.append(v1_0_0)
        }
        let v1_1 = StringBuilder.buildArray(v1_0)
        let v1_2 = "and \\(planets.last!)!"
        v1 = StringBuilder.buildBlock(v1_1, v1_2)
    }
    return StringBuilder.buildBlock(v0, v1)
}
```

## 其他的构建方法

Result builder 类型还可以实现另外的两个方法，我们可以用来把已经构建的结果进行变形转换。第一个是 buildLimitedAvailability，它可以用来转换带有可用性限定的上下文（比如 if #available(...)）中的结果类型。

举例来说，SwiftUI 在它的 view builder 类型中使用了这个方法，来把从可用性限定上下文中获取的中间结果包装到一个擦除了类型的 AnyView 中。因为可用性限定上下文中的结果类型，可能包含上下文之外不可用的类型，所以这么做是必要的。

Result builder 类型上最后一个变形方法是 buildFinalResult：顾名思义，这个方法会对整个 result builder 函数的结果调用一次。这个方法的一个应用场景，是用来隐藏那些被用来构建中间结果的内部类型，让它们不必暴露到外界。

比如，字符串 builder 可以使用 [String] 作为中间类型来构建结果，然后在 buildFinalResult 中将这个数组转换为一个字符串。这意味着除了 buildFinalResult 这个最终返回字符串的方法之外，其他所有的构建方法都可以使用 [String] 作为结果类型：

```
@resultBuilder
struct StringBuilder {
    static func buildBlock(_ x: [String]...) -> [String] { x.flatMap { $0 } }
    static func buildIf(_ x: [String]?) -> [String] { x ?? [] }
    static func buildExpression(_ x: String) -> [String] { [x] }
    static func buildExpression(_ x: Int) -> [String] { ["\(x)"] }
    static func buildExpression(_ x: Never) -> [String] {}
    static func buildExpression(_ x: Void) -> [String] { [] }
    static func buildArray(_ x: [[String]]) -> [String] { x.flatMap { $0 } }
    static func buildEither(first x: [String]) -> [String] { x }
    static func buildEither(second x: [String]) -> [String] { x }
    static func buildFinalResult(_ x: [String]) -> String { x.joined() }
}
```

## 不支持的语句

在上面我们已经看到了如何在 result builder 函数中启用某些类型的语句，比如 if/if let, if ... else, switch 和 for ... in。在本书书写时，几乎其他所有语句，包括 guard, defer, do ... catch, break 和 continue 都是不支持的。

## 回顾

函数是 Swift 中的头类对象。将函数视作数据可以让我们的代码更加灵活。我们已经看到了如何使用简单的函数来替代运行时编程，还比较了几种实现代理的方式。我们也研究了 mutating 函数，inout 参数，下标（实际上它是一种特殊的函数）以及 @autoclosure 和 @escaping 标注。

最后我们介绍了 `result builder` 这一类特殊的函数，它让我们可以用极具表达力和简洁的语法来构建返回值。

在下面的章节中，我们会探讨一个相关的话题：属性。在泛型和协议两章中，我们还将看到关于如何使用 Swift 中的函数来获取额外的灵活性的更多内容。

# 属性

5

Swift 中的属性有两种变体：存储属性和计算属性。存储属性可以存储实际的值，而计算属性则和函数更相似：它们不提供存储，而只提供一种对值的获取和（可选的）设置方法。从这方面考虑，你可以把计算属性想象成一种拥有不同语法的方法。

你可以认为属性就是定义在某个类型上的变量。我们在本章中提到的大部分内容，也适用于本地和全局变量。变量可以被存储或者计算，它们支持变更观察，也可以使用属性包装器。我们认为属性是变量的一种“特殊形式”，而不是反过来。

有两个重要的特性是在属性的基础上构建的：键路径 (key path) 和属性包装 (property wrapper)。键路径是一种在不需要引用值的前提下引用属性的路径的方式。越来越多的库都采用了键路径的方式来让代码变得简洁和通用，我们会在本章看到一些这方面的例子。属性包装则允许你使用极简的语法来改变属性的行为。属性包装在 SwiftUI 的轻量级语法中起到了重要作用。

让我们来看看定义属性的各种方式。我们以表示 GPS 追踪信息的结构体作为开始，它在一个叫做 record 的数组中存储了所有的记录点，它是一个存储属性：

```
import CoreLocation
```

```
struct GPSTrack {  
    var record: [(CLLocation, Date)] = []  
}
```

如果我们想要将 record 属性作为外部只读，内部可读写的话，我们可以使用 `private(set)` 或者 `fileprivate(set)` 修饰符：

```
struct GPSTrack {  
    private(set) var record: [(CLLocation, Date)] = []  
}
```

想要获取 GPS 追踪中所有记录的时间戳，我们可以创建一个计算属性：

```
extension GPSTrack {  
    /// 返回 GPS 追踪的所有时间戳  
    /// - 复杂度：O(n)，n 是记录点的数量。  
}
```

```
var timestamps: [Date] {
    return record.map {$0.1}
}
```

因为我们没有指定 `setter`, 所以 `timestamps` 属性是只读的。它的结果不会被缓存, 每次你访问这个属性时, 结果都要被计算一遍。[Swift API 设计指南](#)推荐你对所有复杂度不是  $O(1)$  的计算属性都应该在文档中写明, 因为调用者可能会假设访问一个属性并不会有什么性能损耗。

## 变更观察者

我们也可以为属性和变量实现 `willSet` 和 `didSet` 方法, 每次当一个属性被设置时(就算它的值没有发生变化), 这两个方法都会被调用。它们会分别在设置前和设置后被立即调用。在 `view` 由于需要基于某个特定属而重新布局时, 这个特性会非常有用。通过在 `didSet` 中调用 `setNeedsLayout`, 我们就可以确信布局一定会发生了(在关于属性包装的部分, 我们会看到一种更短的方法)。

```
class MyView: UIView {
    var pageSize: CGSize = CGSize(width: 800, height: 600) {
        didSet {
            self.setNeedsLayout()
        }
    }
}
```

属性观察者必须在声明一个属性的时候就被定义, 你无法在扩展里进行追加。所以, 这不是一个提供给类型用户的工具, 它是专门提供给类型的设计者的。`willSet` 和 `didSet` 本质上是一对属性的简写: 一个是存储值的私有存储属性; 另一个是读取值的公开计算属性, 这个计算属性的 `setter` 会在将值存储到私有存储属性之前和/或之后, 进行额外的工作。这和 Foundation 中的[键值观察 \(KVO, key-value observing\)](#)有本质的不同, 键值观察通常是对象的消费者来观察对象内部变化的手段, 而与类的设计者是否希望如此无关。

不过, 你可以在子类中重写一个属性, 来添加观察者。下面就是一个例子:

```
class Robot {  
    enum State {  
        case stopped, movingForward, turningRight, turningLeft  
    }  
    var state = State.stopped  
}  
  
class ObservableRobot: Robot {  
    override var state: State {  
        willSet {  
            print("Transitioning from \(state) to \(newValue)")  
        }  
    }  
}  
  
var robot = ObservableRobot()  
robot.state = .movingForward // Transitioning from stopped to movingForward
```

这种做法和“改变观察者是一个类型的内部特性”并不矛盾。即便这种做法不被允许，子类也还是可以通过使用一个计算属性的 `setter` 对父类存储属性进行重写，并完成那些额外的工作。

使用上的差异被反应在这些特性的实现中。KVO 使用 Objective-C 的运行时特性，动态地在类的 `setter` 中添加观察者，这在现在的 Swift 中，特别是对值类型来说，是无法实现的。Swift 的属性观察是一个纯粹的编译时特性。

## 延迟存储属性

延迟初始化一个值在 Swift 中是一种常见的模式，为了定义一个延迟初始化的属性，Swift 提供了一个专用的关键字 `lazy`。关键字来定义一个延迟属性 (`lazy property`)。要注意的是，延迟属性只能用 `var` 定义，因为在初始化方法完成后，它的初始值可能仍旧是未设置的。而 Swift 对 `let` 常量则有着严格的规则，它必须在实例的初始化方法完成之前就拥有值。延迟修饰符是编程记忆化的一种具体的表现形式。

例如，在一个显示 GPSTrack 的 view controller 上，我们可能会想展示一张追踪路径的预览图像。通过延迟加载，我们可以将耗时的图像生成工作推迟到属性被首次访问的时候：

```
class GPSTrackViewController: UIViewController {  
    var track: GPSTrack = GPSTrack()  
  
    lazy var preview: UIImage = {  
        for point in track.record {  
            // Do some expensive computation.  
        }  
        return UIImage(/* ... */)  
    }  
}
```

注意我们是如何定义延迟属性的：它是一个返回存储值(在我们的例子中，就是一张图片)的闭包表达式。当属性第一次被访问时，闭包将被执行(注意闭包后面的括号)，它的返回值将被存储在属性中。对于需要多行代码来初始化的延迟属性来说，这种闭包方式是很常见的。

因为延迟属性需要存储，所以我们需要在 GPSTrackViewController 的定义中来加入这个延迟属性。和计算属性不同，存储属性和需要存储的延迟属性不能被定义在扩展中。同样地，和计算属性不一样，存储属性和延迟存储属性不会在这些属性每次被访问时去重新计算。比如，当 track 被改变时，preview 不会重新计算新的值。

让我们用一个更简单的例子来看看发生了什么。我们有一个 Point 结构体，并且用延迟的方式存储了 distanceFromOrigin：

```
struct Point {  
    var x: Double  
    var y: Double  
    private(set) lazy var distanceFromOrigin: Double  
        = (x*x + y*y).squareRoot()  
  
    init(x: Double, y: Double) {  
        self.x = x  
    }  
}
```

```
self.y = y  
}  
}
```

当我们创建一个点后，可以访问 `distanceFromOrigin` 属性，这将会计算出值，并存储起来等待重用。不过，如果我们之后改变了 `x` 的值，这个变化将不会反应在 `distanceFromOrigin` 中：

```
var point = Point(x: 3, y: 4)  
point.distanceFromOrigin // 5.0  
point.x += 10  
point.distanceFromOrigin // 5.0
```

这需要特别注意。一种解决的办法是在 `x` 和 `y` 的 `didSet` 中重新计算 `distanceFromOrigin`，不过这样一来 `distanceFromOrigin` 就不是真正的延迟属性了，在每次 `x` 或者 `y` 变化的时候它都将被重新计算。当然，在这个例子中，更好的解决方式是，我们一开始就将 `distanceFromOrigin` 设置为一个普通的（非延迟）计算属性。

访问一个延迟属性是 `mutating` 操作，因为这个属性的初始值会在第一次访问时被设置。当结构体包含一个延迟属性时，这个结构体的所有者如果想要访问该延迟属性的话，也需要将结构体声明为可变量，因为访问这个属性的同时，也会潜在地对这个属性的容器进行改变。所以，下面的代码是不被允许的：

```
let immutablePoint = Point(x: 3, y: 4)  
immutablePoint.distanceFromOrigin  
// 错误：不能在一个不可变量上使用可变 getter
```

让想访问这个延迟属性的所有 `Point` 用户都使用 `var` 是非常不方便的事情，所以在结构体中使用延迟属性通常不是一个好主意。

另外需要注意，`lazy` 关键字不会进行任何线程同步。如果在一个延迟属性完成计算之前，多个线程同时尝试访问它的话，计算有可能进行多次，计算过程中的各种副作用也会发生多次。

## 属性包装

把属性包装 (property wrapper) 加入到 Swift 的最直接驱动力毫无疑问就是为了 SwiftUI。不过，这个话题在 SwiftUI 之前就被讨论过了，而且它们的使用范围超过了 SwiftUI：Apple 和社区都写出了适用于很多范围的属性包装。实际上，属性包装的动机之一，是让我们可以把延迟属性写成库的形式，而不需要使用编译器的内建能力。

本质上，属性包装让我们可以改变属性声明的行为。比如说，在下面的 SwiftUI view 中：

```
struct Toggle: View {  
    @Binding var isOn: Bool  
    // ...  
    var body: some View {  
        if isOn {  
            // ...  
        }  
    }  
}
```

在上面的代码中，`@Binding` 是一个属性包装。你可以就像普通的 `Bool` 属性那样来使用 `isOn` 属性：你可以读取和设置它的值。不过行为上会有区别：实际的内存被保存在了 `Toggle` 值之外，它对 `view` 来说是不透明的。你也可以在非 `mutating` 的方法中更改这个值。属性包装在 SwiftUI 中被广泛使用，大多数时候被用来管理状态。

你可以把属性包装用在类和结构体的属性上，也可以用在本地变量（但是不能用在全局变量上）或方法参数上。在本节的最后，我们会更详细讨论属性包装的一些限制。

属性包装的另一个用例，是在 UIKit 和 AppKit 中，某个属性需要把 `view` 的某个部分标记为失效时。回顾一下本章早一些的代码，我们使用了变更观察的方法，来在某个属性发生改变时使 `view` 的布局失效：

```
class MyView: UIView {  
    var pageSize: CGSize = CGSize(width: 800, height: 600) {  
        didSet {  
            self.setNeedsLayout()  
        }  
    }
```

```
    }  
}
```

使用 iOS 15 (以及 macOS 12) 添加的 `@Invalidating` 属性包装，你现在可以写出下面这样的代码了：

```
class MyView: UIView {  
    @Invalidating(.layout) var pageSize: CGSize =  
        CGSize(width: 800, height: 600)  
}
```

`@Invalidating` 属性包装在内部存储了尺寸，一旦这个尺寸发生变化，它就会为你调用 `setNeedsLayout`。如果你有很多属性，它们需要让你的 `view` 的不同部分失效 (比如布局，约束，显示等)，使用这个属性包装能够大幅简化你的代码。你依然可以在属性包装的基础上使用 `willSet` 和 `didSet`；它们的工作方式就和普通属性上的变更观察者是一样的。

除了 `view` 以外，属性包装也非常有用。社区里写了一些属性包装，比如为类型的编解码提供自定义方式的 Codable，为 `User Defaults` 提供包装的 `UserDefaults`，让响应式编程更加简单的属性包装 (比如 `Combine` 中 `@Published`) 等。

## 使用方法

属性包装是一种语法特性：就算不用属性包装，你也可以使用 `Binding` 和 `Invalidating` 这些类型。这里是上面的 `Binding` 例子在不使用属性包装时重写的样子：

```
struct Toggle: View {  
    var isOn: Binding<Bool>  
    // ...  
    var body: some View {  
        if isOn.wrappedValue {  
            // ...  
        }  
    }  
}
```

事实上，当你使用属性包装时，编译器也会以类似的方式把你的代码进行变形。为了说明，我们来构建一个简单的属性包装，它负责把值存放在 `box` 中：

```
@propertyWrapper  
class Box<A> {  
    var wrappedValue: A  
  
    init(wrappedValue: A) {  
        self.wrappedValue = wrappedValue  
    }  
}
```

`Box` 类型用途广泛：当你需要一个可以被共享的可变变量时（可以把同一个 `Box` 实例传递到不同地方），或者当你需要在一个不允许变更的上下文中拥有一个可变值（比如，就算在一个不可变方法中，你也可以修改 `Box` 内的值）时，`Box` 类型都有用武之地。上面的定义让我们可以方便地使用 `Box`：

```
struct Checkbox {  
    @Box var isOn: Bool = false  
  
    func didTap() {  
        isOn.toggle()  
    }  
}
```

想要理解发生了什么，让我们来看看这段相同代码在编译器转换之后的样子：

```
struct Checkbox {  
    private var _isOn: Box<Bool> = Box(wrappedValue: false)  
    var isOn: Bool {  
        get { _isOn.wrappedValue }  
        nonmutating set { _isOn.wrappedValue = newValue }  
    }  
}
```

```
func didTap() {
    isOn.toggle()
}

}
```

对于每个被标记为属性包装的属性，编译器都会为它生成一个带有下划线前缀的实际存储属性。另外，编译器还会生成一个计算属性，来访问这个背后的属性包装的 `wrappedValue`。如果这个属性初始化时带有一个值（在上例中，`isOn` 在初始化时带有 `false` 值），这个属性包装将通过 `.init(wrappedValue:)` 进行初始化。

当定义一个属性包装时，你至少需要为 `wrappedValue` 提供一个 `getter`。`setter` 是可选的，根据属性包装中 `setter` 是否存在，编译器会决定是否为计算属性生成一个 `setter`。在上面的用例中，因为定义了 `setter`，所以计算属性的 `setter` 将被生成。由于 `Box` 是一个 `class`，所以生成的 `setter` 是 `nonmutating` 的。和 `setter` 一样，`init(wrappedValue:)` 也是可选的。因为我们在 `Box` 中提供了这个初始化方法，我们可以通过 `Bool` 来初始化一个 `Box<Bool>`。

## 投影值

在 SwiftUI 中，像是 `@State` 和 `@ObservedObject` 这样的属性包装被用来定义值的所有权以及存储。比如，`@State var x: Int` 定义了一个可变的 `Int` 变量的存储。这个存储本身是被 SwiftUI 管理的。然后，很多组件并不关心某个值是在哪里被存储的，它们需要的只是一个它们可以操作的值。举例来说，`Toggle` 需要一个能够读写的 `Bool` 值，`TextField` 需要一个可变的 `String` 值。这些值通过 `Binding` 属性包装被提供给 SwiftUI，本质上这个属性包装所提供的只是一个值的 `getter` 和 `setter`，而这个值实际是存储在绑定之外的。

SwiftUI 允许你使用属性包装中一个被称为 **投影值 (projected value)** 的特殊特性来从一个 `@State`（或者像是 `@ObservedObject` 之类的其他属性包装）中创建绑定。它的工作方式如下：当你实现 `projectedValue` 属性时，编译器会生成一个和属性自身具有相同名字，但是在定义上前缀一个 `$` 的额外的计算属性来访问它。换句话说，如果你用属性包装定义了一个 `foo` 属性，那么 `$foo` 就相当于 `foo.projectedValue`。

来实际看看，我们可以将 `Box` 进行扩展，来创建一个可以引用 `Box` 的部分值的类型。我们会创建一个和 SwiftUI 中 `Binding` 的工作方式几乎相同的类型。例如，如果我们的 `Box` 中有一个 `Person` 结构体，那我们就可以拥有一个对 `name` 属性的引用。`Box` 依然是 `Person` 值得存储，

但是对 name 的引用允许我们对这部分的值进行读写操作。第一步，我们来定义一个 Reference 属性包装，它负责存储获取和设置一个值的方法：

```
@propertyWrapper
class Reference<A> {
    private var _get: () -> A
    private var _set: (A) -> ()

    var wrappedValue: A {
        get { _get() }
        set { _set(newValue) }
    }

    init(get: @escaping () -> A, set: @escaping (A) -> ()) {
        _get = get
        _set = set
    }
}
```

现在，第二步，我们可以扩展 Box，让它拥有 projectedValue，这个属性从 Box<A> 中创建一个 Reference<A>：

```
extension Box {
    var projectedValue: Reference<A> {
        Reference<A>(get: { self.wrappedValue }, set: { self.wrappedValue = $0 })
    }
}
```

因为我们实现了 projectedValue，现在我们可以创建一个持有 Person 值的 Box，使用 \$ 前缀基于这个 Box 值创建一个引用，然后把这个引用传递给一个另外的函数或者初始化方法了。在 PersonEditor 中，对于 person 的任意变动，都将改变 Box 中背后的值：

```
struct Person {
    var name: String
```

```
}
```

```
struct PersonEditor {  
    @Reference var person: Person  
}  
  
func makeEditor() -> PersonEditor {  
    @Box var person = Person(name: "Chris")  
    PersonEditor(person: $person)  
}
```

投影值在和基于键路径的动态成员查找组合使用时，会非常有用。举个例子，相比起把 Person 传递给 PersonEditor，我们可能更希望只把 person 的 name 传递给一个 TextEditor。在上面的例子中，我们不能只写 \$person.name，因为 \$person 的类型是 Reference<Person>，而不是 Person。我们可以通过为 Reference 类型添加动态成员查找的方式进行修正：

```
@propertyWrapper  
@dynamicMemberLookup  
class Reference<A> {  
    // ...  
  
    subscript<B>(dynamicMember keyPath: WritableKeyPath<A, B>)  
        -> Reference<B> {  
        Reference<B>(get: {  
            self.wrappedValue[keyPath: keyPath]  
        }) {  
            self.wrappedValue[keyPath: keyPath] = $0  
        }  
    }  
}
```

这将允许我们通过 \$person.name 来创建一个 Reference<String>。\$foo.prop1.prop2 这样的语法则会被变形为 foo.projectedValue[dynamicMember: \.prop1.prop2]。

## Self 封装

有些属性包装只有在当它们能访问到它的封装 (enclosing) 对象 (也就是属性包装值的“宿主”) 时，才会有用。比方说，Apple 的 Combine 框架中的 @Published 属性包装，就需要能访问到包含有这个 @Published 属性的对象上的 objectWillChange 属性。在下面这个例子中，对于 a.city 的变更，需要通过 a.objectWillChange 这个 publisher 才能把事件发送出去 (因此，@Published 需要能访问 a: Address 的 objectWillChange)：

```
class Address: ObservableObject {  
    // ...  
    @Published var city: String = "Berlin"  
}  
  
let a = Address()  
a.city = "New York"
```

类似地，在本节介绍部分中的 @Invalidateing 里，对任意被标记为 @Invalidateing 属性的变更，将会使封装 view 无效化。作为展示工作原理的例子，我们会重新实现一小部分的 @Invalidateing。想要获取到封装实例，我们需要用到一个非公开的 API。如果以后这部分内容公开的话，官方的 API 可能会和本节中描述的有所不同。当需要负责封装的对象时，我们不再去实现 wrappedValue，而是需要实现一个静态的下标方法。这个下标方法会在参数中接收到封装对象，以及构成属性包装的两个属性键路径。这里是一个例子：

```
@propertyWrapper  
struct InvalidateingLayout<A> {  
    private var _value: A  
    // ...  
    static subscript<T: UIView>(  
        _enclosingInstance object: T,  
        wrapped _: ReferenceWritableKeyPath<T, A>,  
        storage storage: ReferenceWritableKeyPath<T, Self>) -> A {
```

```
get {
    object[keyPath: storage]._value
}
set {
    object[keyPath: storage]._value = newValue
    object.setNeedsLayout()
}
}
```

在上面的代码中，属性包装将使用下标方法，而非 `wrappedValue`。在下标方法中，我们可以（通过 `object`）获取到 `view`，以及（通过结合使用封装对象和 `storage` 键路径）获取到属性包装自身。在使用时，这个属性包装就和普通的属性包装并无二致：

```
class AView: UIView {
    @InvalidateLayout var x = 100
}
```

根据上面的定义，编译器会为我们生成如下代码：

```
class AView: UIView {
    var _x = InvalidateLayout(wrappedValue: 100)
    var x: Int {
        get {
            InvalidateLayout[_enclosingInstance: self, wrapped: \.x, storage: \._x]
        }
        set {
            InvalidateLayout[_enclosingInstance: self, wrapped: \.x, storage: \._x]
                = newValue
        }
    }
}
```

我们关于 `@InvalidateLayout` 属性的定义还不完整。除了下标之外，我们还需要提供一个 `init(wrappedValue:)` 来允许使用一个初始值进行初始化。我们还被要求实现 `wrappedValue`，由于我们确实需要 `object`，所以我们确实没什么办法写出一个有意义的 `setter` 实现。不过，我们可以把这个属性标记为弃用：

```
struct InvalidatingLayout<A> {  
    // ...  
  
    @available(*, unavailable,  
        message: "@InvalidateLayout is only available on UIView subclasses")  
    var wrappedValue: A {  
        get { fatalError() }  
        set { fatalError() }  
    }  
  
    init(wrappedValue: A) {  
        _value = wrappedValue  
    }  
  
    // ...  
}
```

这个 `wrappedValue` 上的标注可以确保我们的属性包装只被用在 `UIView` 的子类上；任何其他的使用方式都会造成编译错误。

在 SwiftUI 中，你可能会觉得 `@State` 也是通过查找自身的封装来工作的。但事实并非如此，因为进行封装的并不是一个对象，而是一个值类型。实际上，`State` 值是 SwiftUI 内部通过 `view` 在 `view` 层级中的位置来进行管理的。这一行为通过内省 (introspection) 编程的方式实现。

## 属性包装的来龙去脉

属性包装可以修饰结构体或类中的属性，但是它们无法在枚举中工作。因为枚举在 `case` 之外并没有存储空间，而属性包装总是需要生成一个存储属性，所以无法使用是合情合理的。你可以在本地变量 (比如说在一个函数的函数体内) 使用属性包装，但是你不能在全局变量中使用。

更进一步，通过属性包装定义的属性或者变量，是不能被 `unowned`, `weak`, `lazy` 或 `@NSCopying` 修饰的。

从 Swift 5.5 开始，函数也能够接受属性包装作为参数了。例如，下面这个函数：

```
func takesBox(@Box foo: String) {  
    // ...  
}
```

要调用这个函数，你需要提供 `String` 而非 `Box<String>` 作为参数。编译器会把上面的函数转换成这样的形式：

```
func takesBox(foo initialValue: String) {  
    var _foo: Box<String> = Box(wrappedValue: initialValue)  
    var foo: String {  
        get { _foo.wrappedValue }  
        nonmutating set { _foo.wrappedValue = newValue }  
    }  
}
```

注意，在上面的例子中，属性包装 `Box` 类型本身并不是 `takesBox` 这个 API 的一部分。然而，如果属性包装拥有一个名为 `init(projectedValue:)` 的初始化方法，那么编译器就还会为这个函数生成接受 `projectedValue` 作为参数的另一个版本。比如，如果我们在 `Box` 上实现了 `init(projectedValue:)` 的话，`takesBox` 的第二个版本看起来会是这样的：

```
func takesBox($foo initialValue: Reference<String>) {  
    var _foo: Box<String> = Box(projectedValue: initialValue)  
    var foo: String {  
        get { _foo.wrappedValue }  
        nonmutating set { foo.wrappedValue = newValue }  
    }  
}
```

换言之，当一个带有 `init(projectedValue:)` 的属性包装被用在函数中时，这个包装类型（上面的 Reference）将会变为这个函数 API 的一部分。在结构体中被使用属性包装定义的属性，同样也是逐一成员初始化方法 (memberwise initializer)的一部分。举例来说，下面这个结构体定义：

```
struct Test {  
    @Box var name: String  
    @Reference var street: String  
}
```

编译器为它生成的成员初始化方法将会是 `init(name: String, street: Reference<String>)`。如果这个属性包装上存在 `init(wrappedValue:)` 初始化方法（就像 `@Box` 那样），那么成员初始化方法将会接受一个被包装的值（比如上例中的 `String`），并将它用于设置该属性。如果属性包装中并不存在这样的初始化方法（比如 `@Reference`），那么成员初始化方法中对应的参数就需要包含属性包装，当然，你仍然可以按照需要去创建自己的初始化方法，来接受被包装值或者没有被包装的值。

属性包装也能够被嵌套使用。比如，要对一个属性创建双层 box，可以这样写：

`@Box @Box var name: String`。不过，这种方法看起来（现在还）无法适用于需要封装自身实例的那些属性包装。另外，那些基于内省方式的属性包装（比如 SwiftUI 中的 `@State`）通常只有在它们是最外层包装时，才能正常工作。

## 键路径

键路径是一个指向属性的未调用的引用，它和对某个方法的未使用的引用很类似。键路径表达式以一个反斜杠开头，比如 `\String.count`。反斜杠是为了将键路径和同名的类型属性区分开来（假如 `String` 也有一个 `static count` 属性的话，`String.count` 返回的就是这个属性值了）。类型推断对键路径也是有效的，在上下文中如果编译器可以推断出类型的话，你可以将类型名省略，只留下 `\.count`。

正如其名，键路径描述了一个从根开始的类型层级路径。举例来说，在下面的 `Person` 和 `Address` 类型中，`\Person.address.street` 表达了一个人的街道住址的键路径：

```
struct Address {  
    var street: String
```

```
var city: String  
var zipCode: Int  
}  
  
struct Person {  
    let name: String  
    var address: Address  
}  
  
let streetKeyPath = \Person.address.street  
// Swift.WritableKeyPath<Person, Swift.String>  
let nameKeyPath = \Person.name // Swift.KeyPath<Person, Swift.String>
```

键路径可以由任意的存储和计算属性组合而成，其中还可以包括可选链操作符。编译器会自动为所有类型生成 [keyPath:] 的下标方法。你通过这个方法来“调用”某个键路径。对键路径的调用，也就是在某个实例上访问由键路径所描述的属性。所以，“Hello”[keyPath: \.count] 等效于 “Hello”.count。或者在我们现在的例子中：

```
let simpsonResidence = Address(street: "1094 Evergreen Terrace",  
    city: "Springfield", zipCode: 97475)  
var lisa = Person(name: "Lisa Simpson", address: simpsonResidence)  
lisa[keyPath: nameKeyPath] // Lisa Simpson
```

如果你检查上面两个键路径变量的类型，你会注意到 nameKeyPath 的类型是 KeyPath<Person, String>。这个键路径是强类型的，它表示该键路径可以作用于 Person，并返回一个 String。而 streetKeyPath 是一个 WritableKeyPath，这是因为构成这个键路径的所有属性都是可变的，所以这个可写键路径本身允许其中的值发生变化：

```
lisa[keyPath: streetKeyPath] = "742 Evergreen Terrace"
```

对 nameKeyPath 做同样的操作会造成错误，因为它背后的属性不是可变的。

键路径不仅可以对属性做引用，我们也可以用它们来描述下标操作。例如，可以用下面这样的语法提取数组里第二个 person 对象的 name 属性：

```
var bart = Person(name: "Bart Simpson", address: simpsonResidence)
let people = [lisa, bart]
people[keyPath: \.[1].name] // Bart Simpson
```

同样的语法也可用于在键路径中包含字典下标。

## Key Paths Can Be Modeled with Functions

一个将基础类型 Root 映射为类型为 Value 的属性的键路径，和一个具有 (Root) -> Value 类型的函数十分类似。而对于可写的键路径来说，则对应着一对获取和设置值的函数。相对于这样的函数，键路径除了在语法上更简洁外，最大的优势在于它们是值。你可以测试键路径是否相等，也可以将它们用作字典的键（因为它们遵守 Hashable）。另外，不像函数，键路径是不包含状态的，所以它也不会捕获可变的状态。如果使用普通的函数的话，这些都是无法做到的。

编译器可以自动把一个键路径表达式转换为一个函数。比如说，下面的代码其实是 `people.map { $0.name }` 的一种简写：

```
people.map(\.name) // ["Lisa Simpson", "Bart Simpson"]
```

注意，这只对键路径表达式有效。比如，下面的代码用键路径表达式去定义了一个键路径，但是它无法编译：

```
/*show*/
let keyPath = \Person.name
people.map(keyPath)
```

对于一个键路径表达式，编译器在推断它的类型时，有两种可能的选择：`\Person.name` 可以被推断为 `KeyPath<Person, String>`，也可以是 `(Person) -> String`。编译器会优先尝试把它推断为键路径类型，但是如果不行的话，它会再尝试函数类型。

键路径还可以通过将一个键路径附加到另一个键路径的方式来生成。这么做时，类型必须要匹配；如果你有一个从 A 到 B 的键路径，那么你要附加的键路径的根类型必须为 B，得到的将会是一个从 A 到 C 的键路径，其中 C 是所附加的键路径的值的类型：

```
// KeyPath<Person, String> + KeyPath<String, Int> = KeyPath<Person, Int>
let nameCountKeyPath = nameKeyPath.appending(path: \.count)
// Swift.KeyPath<Person, Swift.Int>
```

## 可写键路径

可写键路径比较特殊：你可以用它来读取或者写入一个值。因此，它和一对函数等效：一个负责获取属性值 ((Root) -> Value)，另一个负责设置属性值 ((inout Root, Value) -> Void)。可选键路径将很多代码包括在了简洁的语法中。把 streetKeyPath 与等效的 getter 和 setter 对进行比较：

```
let streetKeyPath = \Person.address.street
let getStreet: (Person) -> String = { person in
    return person.address.street
}
let setStreet: (inout Person, String) -> () = { person, newValue in
    person.address.street = newValue
}

// 使用 setter
lisa[keyPath: streetKeyPath] = "1234 Evergreen Terrace"
setStreet(&lisa, "1234 Evergreen Terrace")
```

可写键路径有两种形式：WritableKeyPath 和 ReferenceWritableKeyPath。第二种形式是和那些具有引用语义的值配合使用的，而第一种形式则适用于其他所有类型。使用上的区别在于，WritableKeyPath 要求原值是可变的，而 ReferenceWritableKeyPath 则没有这个要求。

可写键路径在像是 SwiftUI 这样的框架中被广泛使用。举个例子，在 SwiftUI 中，在 view 层级间会有一个从外到内传递“环境”值。这个值被框架管理和传递，但是你可以通过一个特殊的函数用 WritableKeyPath 来改变这个值的一部分。另外，我们在投影值的部分也看到过，可写键路径在结合动态成员查找时，也会非常有用。

## 键路径层级

There are five different types for key paths, each adding more precision and functionality to the previous one:

键路径有五种不同的类型，每种类型都在前一种上添加了更加精确的描述及功能：

- AnyKeyPath 和 (Any) -> Any? 类型的函数相似。
- PartialKeyPath<Source> 和 (Source) -> Any? 函数相似。
- KeyPath<Source, Target> 和 (Source) -> Target 函数相似。
- WritableKeyPath<Source, Target> 和 (Source) -> Target 与 (inout Source, Target) -> () 这一对函数相似。
- ReferenceWritableKeyPath<Source, Target> 和 (Source) -> Target 与 (Source, Target) -> () 这一对函数相似。第二个函数可以用 Target 来更新 Source 的值，且要求 Source 是一个引用类型。对 WritableKeyPath 和 ReferenceWritableKeyPath 进行区分是必要的，前一个类型的 setter 要求它的参数是 inout 的。

这几种键路径的层级结构现在是通过类的继承来实现的。理想状态下，这些特性应该由协议来完成，但是 Swift 的泛型系统还缺少一些使之可行的特性。这种类的层级有意地保持了对外不可见，这样以便于未来在更新时，现有的代码也不会被破坏。

我们前面也提到，键路径不同于函数，它们是满足 Hashable 的，而且在将来它们很有可能还会满足 Codable。这也是为什么我们强调 AnyKeyPath 和 (Any) -> Any 类型只是相似的原因。虽然我们能够将一个键路径转换为对应的函数，但是我们无法做相反的操作。

## 对比 Objective-C 的键路径

在 Foundation 和 Objective-C 中，键路径是通过字符串来建模的（我们会将它们称为 Foundation 键路径，以区别 Swift 的键路径）。由于 Foundation 键路径是字符串，它们不含有任何的类型信息。从这个角度看，它们和 AnyKeyPath 类似。如果一个 Foundation 键路径拼写错误、或者没有正确生成、或者它的类型不匹配的话，程序可能会崩溃。（Swift 中的 #keyPath 指令对拼写错误的问题进行了一些改善，编译器可以检查特定名字所对应的属性是否存在。）Swift 的 KeyPath、WritableKeypath 和 ReferenceWritableKeyPath 从构造开始就是正确的：它们不可能被拼错，也不会有类型错误。

很多 Cocoa API 在原本用函数会更好的地方使用了 (Foundation) 键路径。这其中有一部分是历史原因：匿名函数 (或者在 Objective-C 中所谓的 block) 其实是相对最近才添加的特性，而键路径的存在则要长久得多。在 block 被引入 Objective-C 之前，想要在不用键路径 "address.street" 的条件下，表达类似 { \$0.address.street } 这样的函数是很困难的。

## 未来的方向

键路径还在活跃的讨论中，而且可能在未来它会具有更多功能。一个可能的特性是通过 Codable 进行序列化。这将允许我们把键路径存储到磁盘上，或者是通过网络进行传递。一旦我们可以访问到键路径的结构，我们就可以进行内省。比如，我们可以用键路径的结构来构建带有类型的数据库查询语句 (其实已经有开源项目在做这个了，但是它们所依赖的内部实现可能会在未来发生改变)。如果类型能够自动提供它们的属性的键路径数组的话，就可以作为运行时反射 API 的基础了。

当前，相比起直接访问属性来说，键路径是很慢的。这是一个已知问题。在绝大多数情况下，键路径带来的明确表达要比它们的速度更重要，但是如果我要写一些性能敏感的代码的话，你还是需要注意这一点 (比如不要在一个密集的循环里去生成键路径)。

## 回顾

属性和变量是 Swift 中的基础内容。虽然存储属性和计算属性在使用时拥有相同的语法，但它们有着非常大的区别：存储属性用来存储数据，而计算属性则更接近于函数。

在很大程度上，属性包装和键路径都是语法糖。换句话说，它们让你可以用短得多的语法来表达你之前就能够写出来的东西。不过，不要因为这个原因就否定它们。干净的语法对写出清晰的代码是非常重要的。

属性包装和其他任何一种抽象一样，会让正在发生的事情变得更难被理解。不过，一个精心设计的属性包装则可以清理常见的模式，这会让额外的抽象层物超所值。SwiftUI 中对属性包装的使用就是一个很好的例子。

# 结构体和类

6

当我们设计自己的数据类型时，Swift 提供了两个乍看之下非常类似的选项：结构体和类。它们都能拥有存储属性和计算属性，都可以定义方法。不止如此，它们不仅都有初始化方法，而且都可以进行扩展，以及都可以实现各种协议。有时，就算我们把代码中的 `class` 和 `struct` 关键字做一个互换，代码也能编译通过。但别被这种表面的相似性给骗到了，结构体和类在行为上有着天壤之别。

结构体是值类型，而类是引用类型。就算不知道这两个术语，在每天的工作中，我们都已经熟悉了值和引用的行为。下一节中，我们将阐明这种潜在认知，从整体上对值类型和引用类型之间的区别，更确切说，对结构体和类的区别，进行正式的区分。

## 值类型和引用类型

让我们从一个最简单的类型开始：整数。对于下面的代码来说：

```
var a:Int = 3
var b = a
b += 1
```

现在，`a` 的值是什么呢？毋庸置疑，即使把 `b` 增加到 4，我们还是会期望 `a` 的值是 3。除此之外的结果都是令人惊讶的。而这种期望确实也是正确的：

```
a // 3
b // 4
```

这种行为揭示了值类型的本质：赋值意味着按值拷贝。也就是说，每个值类型变量所持有的值都是独立的。具有这种行为特征的类型被称为具有值语义 (**value semantics**)。

看一下标准库中 `Int` 的定义，可以看到它的确就是一个结构体，因此也就具有值语义：

```
public struct Int: FixedWidthInteger, SignedInteger {
    ...
}
```

在继续讨论接下来的内容之前，让我们先停一下，从一个更底层的角度来看看这种行为。

如何定义“变量”这个词呢？我们可以说变量是一个名字，这个名字表示内存中的一个位置，而这个位置包含了某个特定类型的值。在上面例子中，我们使用 `a` 这个名字来指向内存中的一个位置，这个位置包含了一个 `Int` 类型的值 `3`。第二个变量 `b` 是指向内存中另外一个位置的名字，在初始化赋值之后，包含了和 `a` 同样的类型和值。接下来的 `b += 1` 语句则读取 `b` 所指向的内存中的值，再加1，然后写回到相同的位置。因此，现在 `b` 的值为 `4`。因为这个语句只会修改变量 `b` 的值，所以 `a` 并不受影响。

Memory	
...	
Variable <code>a</code> : Int	3
Variable <code>b</code> : Int	4
...	

Figure 6.1: 一个值类型变量是一个指向内存中某个位置的名字，这个位置保存了一个值

值类型的特征就是变量和值之间的这种直接关系：在变量的背后，值（也可以说是值类型的实例）直接保存在变量指向的内存位置。这不但适用于像整数这样简单的值类型，同样也适用于更复杂的类型。例如，包含多个属性的自定义结构体（在机器码的层面，因为编译器的优化，这个说法不一定成立。但这些对开发人员来说是不可见的，因此至少在语义上，我们的说法是准确的）。

接下来，让我们来看一个 `UIView` 例子，它一个典型的引用类型：

```
var view1 = UIView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))
var view2 = view1
view2.frame.origin = CGPoint(x: 50, y: 50)
view1.frame.origin // (50, 50)
view2.frame.origin // (50, 50)
```

尽管只对 view2.frame.origin 赋值了一个新的 CGPoint 值，但我们自然希望 view1 的 frame 属性也会被修改。事实上，我们认为 view1 和 view2 在某种意义上是相同的，它们是我们在屏幕上看到的同一个 view。可以这么说，因为 UIView 是一个引用类型，所以 view1 和 view2 这两个变量包含的引用，背后都指向内存中同一个 UIView 实例。

我们可以这样对 view2 重新赋值：

```
view2 = UILabel()
```

当我们这么做时，虽然现在 view2 指向了一个新创建的 UILabel 实例，但 view1 仍旧指向之前创建的 UIView 实例。换言之，重新赋值这个操作改变了 view2 所指向的实例（或者说是对象）。

这就是引用类型的本质：变量不含有“事物”本身（例如，UIView 或 URLSession 的实例），而是持有一个对“事物”的引用。其他变量也可以含有对同一个实例的引用，并可以通过任意一个指向它的变量对其做修改。具有这些特性的类型被称为具有引用语义（reference semantics）。

相比于值类型，这里多了一个间接层在发挥作用。一个值类型变量含有的是值本身，而一个引用类型变量含有的是一个指向其他某个地方的值的引用。这种间接性允许我们可以在程序的不同部分访问同一个对象。

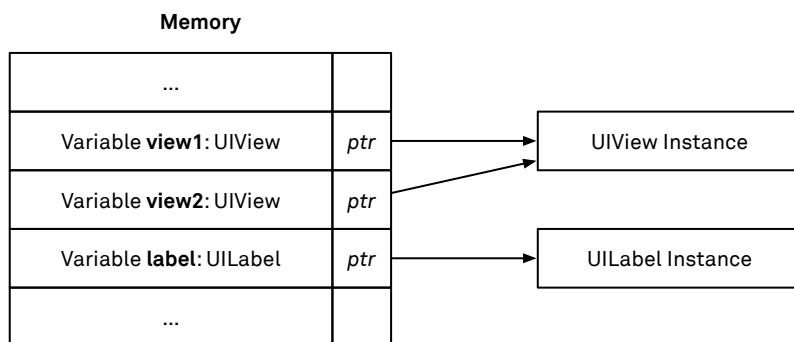


Figure 6.2: 一个引用类型变量含有一个指针，指向在内存中某个地方真正的实例

让我们通过创建一个自定义类型，来观察下值类型和引用类型在行为上的差别。先从一个类开始（这个类记录主队和客队的得分）：

```
class ScoreClass {  
    var home: Int  
    var guest: Int  
    init(home: Int, guest: Int) {  
        self.home = home  
        self.guest = guest  
    }  
}  
  
var score1 = ScoreClass(home: 0, guest: 0)  
var score2 = score1  
score2.guest += 1  
score1.guest // 1
```

score1 和 score2 这两个变量，背后都指向同一个 Score 实例。因此，用 score2 变量修改的 guest 值，也能通过读取 score1 的 guest 属性而得到。我们也能把 score2 传递给一个函数，并在函数中执行一些修改：

```
func scoreGuest(_ score: ScoreClass) {  
    score.guest += 1  
}  
scoreGuest(score1)  
score1.guest // 2  
score2.guest // 2
```

如果我们把这个类型改成结构体的话，行为则会发生变化：

```
struct ScoreStruct {  
    var home: Int  
    var guest: Int  
} // 编译器会生成成员初始化方法 (Memberwise initializer)。
```

```
}
```

```
var score3 = ScoreStruct(home: 0, guest: 0)
```

```
var score4 = score3
```

```
score4.guest += 1
```

```
score3.guest // 0
```

和我们之前看到的整数的例子一样，将一个结构体赋值给另外一个变量，会创建一个该结构体的拷贝。因此，通过 `score4` 变量改变 `guest` 的得分，并不会影响 `score3` 上 `guest` 的得分。

使用结构体版本的 `Score`，我们就不能像之前用类的时候那样，定义 `scoreGuest` 函数了。首先，就像给变量赋值一样，把值类型作为参数传递给函数时，也会创建一份这个值的拷贝。其次，在函数里这个参数是不可变的（就像一个声明为 `let` 的变量），我们不能改变它的属性。为了创建一个类似的函数，我们必须使用一个 `inout` 参数，我们会在下一节讨论这个问题。

虽然结构体和类在语法上有相似性，并且有些特性是相同的，但我们希望通过对于两者的这种初步概述，能突出它们的不同特质。在本章接下来的部分，我们将探讨在结构体和类之间如何做权衡。类是更为强大的工具，但能力越大代价也越大。另一方面，虽然结构体有很多限制，但这些限制也能带来好处。

## 可变性

在控制可变性方面，结构体和类有着天壤之别。一开始，这种差别可能有违直觉，但把它摆在值类型和引用类型的区别面前，就显得非常合理了（毕竟，结构体是值类型，类是引用类型）。我们还是用之前的 `ScoreClass` 和 `ScoreStruct` 举例：

```
class ScoreClass {
```

```
    var home: Int
```

```
    var guest: Int
```

```
    init(home: Int, guest: Int) {
```

```
        self.home = home
```

```
        self.guest = guest
```

```
    }
```

```
}
```

```
struct ScoreStruct {  
    var home: Int  
    var guest: Int  
    // 编译器生成成员初始化方法。  
}
```

这两个类型都用 `var` 关键字声明了 `home` 和 `guest` 属性。如果我们各创建一个实例，并把它们保存到用 `var` 声明的变量中，我们就可以随意修改这些属性：

```
var scoreClass = ScoreClass(home: 0, guest: 0)  
var scoreStruct = ScoreStruct(home: 0, guest: 0)  
scoreClass.home += 1  
scoreStruct.guest += 1
```

但这里，对类和结构体的修改有一个非常重要的区别：对于结构体的修改永远都只是修改局部变量，也就是说，我们只修改了局部变量 `scoreStruct` 的值。而对于类实例的修改可能会产生全局性的影响：这个修改会影响其他那些，持有指向同一个实例的引用的对象。

如果我们将两个实例保存到用 `let` 定义的变量中，类实例还是可以修改的，但结构体实例则不行了：

```
let scoreClass = ScoreClass(home: 0, guest: 0)  
let scoreStruct = ScoreStruct(home: 0, guest: 0)  
scoreClass.home += 1 // 可以工作  
scoreStruct.guest += 1  
// 错误：可变操作符的左侧是不可变类型  
// 'scoreStruct' 是一个 `let` 常量
```

用 `let` 声明一个变量，意味着在初始化之后，就不能改变它的值了。由于 `scoreClass` 变量的值是一个指向 `ScoreClass` 实例的引用，所以这意味着我们只是不能再给 `scoreClass` 赋值另外一

个引用而已。但修改我们所创建的 ScoreClass 实例的属性时，并不需要改变 scoreClass 变量的值。我们只需要通过引用得到实例，然后在实例上修改那些我们用 var 所声明的属性。

对于结构体的情况，这个行为是非常不同的。因为结构体是值类型，所以 scoreStruct 变量不包含一个对其他某个地方的实例的引用，而是 ScoreStruct 实例本身。由于用 let 声明的变量的值，在初始化之后就不能再被修改，所以即使在结构体里一个属性是用 var 声明的，我们也不能修改它。会有这种行为的原因是，其实修改一个结构体的属性这件事，在语义上是等同于对变量赋值一个新的结构体实例。所以，之前例子中的代码：

```
scoreStruct.guest += 1
```

等同于：

```
scoreStruct = ScoreStruct(home: scoreStruct.home,  
    guest: scoreStruct.guest + 1)
```

这个情况不仅发生在直接修改一个结构体实例的属性时，还适用于修改任何的嵌套属性。例如，修改一个矩形的原点 x 坐标，在语义上等同于对变量赋值一个全新的矩形值：

```
var rect = CGRect(origin: .zero, size: CGSize(width: 100, height: 100))  
rect.origin.x = 10 // 这等同于下一行代码  
rect = CGRect(origin: CGPoint(x: 10, y: 0), size: rect.size)
```

那如果我们把属性都声明成 let，而把 scoreClass 以及 scoreStruct 这两个变量声明成 var 的话，会发生些什么呢？

```
class ScoreClass {  
    let home: Int  
    let guest: Int  
    init(home: Int, guest: Int) {  
        self.home = home  
        self.guest = guest  
    }  
}
```

```
}
```

```
struct ScoreStruct {  
    let home: Int  
    let guest: Int  
}  
  
var scoreClass = ScoreClass(home: 0, guest: 0)  
var scoreStruct = ScoreStruct(home: 0, guest: 0)  
scoreClass.home += 1  
// 错误：可变操作符的左侧是不可变类型：  
// 'home' 是一个 `let` 常量  
scoreStruct.guest += 1  
// 错误：可变操作符的左侧是不可变类型：  
// 'guest' 是一个 `let` 常量
```

即使 `scoreClass` 被声明为 `var`，我们还是不能修改它的属性。把变量声明为 `var`，只意味着我们可以改变变量本身的价值。在类的情况下，变量的值是一个指向实例的引用，所以我们可以改变的是变量持有的引用：

```
scoreClass = ScoreClass(home: 2, guest: 1) // 没问题
```

然而，因为 `home` 属性被定义为了 `let`，所以我们没办法更改 `scoreClass` 所指向的实例的这个属性。

我们同样也不能修改结构体的属性：属性都被定义成了 `let`，所以即使 `scoreStruct` 被定义成 `var`，也不能修改它的属性了。不过，我们还是可以给 `scoreStruct` 变量赋值一个新的结构体实例：

```
scoreStruct = ScoreStruct(home: 2, guest: 1) // 可以工作
```

最后，如果我们把属性和变量都定义为 `let` 的话，编译器会禁止任何形式的修改：即不能把一个新的实例赋值给 `someClass` 或 `someStruct`，也不能修改实例的任何属性。

我们推荐默认使用 var 来定义结构体的属性。这允许通过在变量这一层上，使用 var 或 let 来控制结构体实例的可变性，从而为你提供了更大的灵活性。相比于类，在结构体上使用 var 定义属性，并不会引入潜在的全局可变状态，因为修改一个结构体的属性，实际上只是创建一份带着修改值的结构体的拷贝。我们应该谨慎地使用 let，只应该把那些即使实例被保存在一个 var 变量，但在初始化之后确实就不应该再改变的属性（例如，因为改变了某个属性，就会使结构体进入一种无效的状态）声明为常量。

要理解在属性和变量上用 let 和 var 的所有不同组合的关键是要记住两点：

- 类型为类的变量的值，是一个指向实例的引用；而类型为结构体的变量的值，是结构体实例本身。
- 修改一个结构体的属性，即使修改的是多层的嵌套属性，都等同于给变量赋值一个全新的结构体实例。

## 可变方法

在结构体上用 func 关键字定义的普通方法，是不能修改结构体的任何属性的。这是因为被隐式传入的 self 参数，默认是不可变的。我们必须明确地使用 mutating func 关键字来创建一个可变方法：

```
extension ScoreStruct {  
    mutating func scoreGuest() {  
        self.guest += 1  
    }  
}  
  
var scoreStruct2 = ScoreStruct(home: 0, guest: 0)  
scoreStruct2.scoreGuest()  
scoreStruct2.guest // 1
```

在可变方法中，我们可以认为 self 是一个用 var 声明的变量，所以也就可以修改那些在 self 中，用 var 声明的属性。

编译器把 `mutating` 关键字作为一个标记，以此来决定哪些方法是不能在 `let` 常量上被调用的。我们只能对用 `var` 声明的变量调用可变方法，因为调用一个可变方法，就等于对变量赋值一个新的值（事实上，在可变方法中，还允许对 `self` 赋值一个全新的值）。如果尝试在一个 `let` 变量上调用可变方法的话，编译器会显示错误。而且即使可变方法实际上并不会改变 `self`，`mutating` 标注也足以禁止此方法在 `let` 变量上被调用。

属性和下标的 `setter` 都是隐式的可变方法。在极少数的情况下，你会希望使用一个不可变的 `setter` 来实现计算属性，例如，你的结构体封装了一个全局资源，而相应属性的 `setter` 只是去修改这个全局状态。这个时候，你可以用 `nonmutating set` 来标注相应的 `setter`。编译器允许你在一个 `let` 常量上调用此类 `setter`。

类并没有也不需要可变方法，正如我们上面所见，即使一个类变量被声明为 `let`，我们依然可以改变实例的属性。和结构体的普通方法类似，在类的方法中，`self` 表现得就像一个用 `let` 声明的变量。不过虽然我们不能对 `self` 重新赋值，但可以通过使用 `self` 来修改其所指向的实例的属性，只要该属性被声明为 `var` 就可以。

## inout 参数

我们在上面提到过，因为在结构体的可变方法中，访问的是一个可变的 `self`，所以它能改变这个 `self` 中任何一个被声明为 `var` 的属性。但除了利用 `self` 之外，我们还可以通过 `inout` 参数编写可以直接修改参数的函数。例如，之前的 `scoreGuest` 可变方法还可以像这样写成一个全局函数：

```
func scoreGuest(_ score: ScoreStruct) {
    score.guest += 1
    // 错误：可变操作符的左边是不可变类型：
    // 'score' 是一个 'let' 常量。
}
```

函数的参数默认就像 `let` 变量一样，是不可变的。我们虽然可以把参数复制并赋值给一个局部的 `var` 变量，但对这个变量的修改，并不会对传入的原来的值产生影响。为了解决这个问题，我们可以在参数类型之前加上 `inout` 关键字：

```
func scoreGuest(_ score: inout ScoreStruct) {  
    score.guest += 1  
}  
  
var scoreStruct3 = ScoreStruct(home: 0, guest: 0)  
scoreGuest(&scoreStruct3)  
scoreStruct3.guest // 1
```

为了向 `scoreGuest` 函数传递 `inout` 参数，我们必须做两件事：首先，作为 `inout` 参数传递的变量必须是用 `var` 定义的；其次，当把这个变量传递给函数时，必须在变量名前加上 `&` 符号。站在调用者的角度，`&` 符号可以清楚的表明，这个函数会修改传入的变量的值。

虽然 `&` 符号可能会让你想起 C 和 Objective-c 中的取址操作符，或者是 C++ 中的引用传递操作符，但在 Swift 中，其作用是不一样的。就像对待普通的参数一样，Swift 还是会复制传入的 `inout` 参数，但当函数返回时，会用这些参数的值覆盖原来的值。也就是说，即使在函数中对一个 `inout` 参数做多次修改，但对调用者来说只会注意到一次修改的发生，也就是在用新的值覆盖原有值的时候。同理，即使函数完全没有对 `inout` 参数做任何的修改，调用者也还是会注意到一次修改 (`willSet` 和 `didSet` 这两个观察者方法都会被调用)。正如我们在[函数](#)一章中解释过的那样，编译器会在安全的时候，把复制操作优化掉，转而变成传递引用。

## 生命周期

在生命周期管理方面，结构体和类是非常不同的。相比类，结构体要简单得多，因为它们不会有多个所有者，它们的生命周期，是和含有结构体实例的变量的生命周期绑定的。当变量离开作用域时，其内存将被释放，结构体实例也会被销毁。

与此相反，一个类的实例可以被多个所有者引用，这就需要一种更精细的内存管理模型。Swift 使用自动引用计数 (ARC) 来追踪一个实例的引用计数。当引用计数降至 0 时 (例如所有包含引用的变量都离开了作用域，或被设置成了 `nil`)，Swift 运行时会调用对象的 `deinit` 方法并释放内存。因此，对那些在最终被释放时需要执行清理工作的共享对象，是可以用类来实现的。这类例子包括像是文件句柄 (必须在某个时间点关闭底层的文件描述符)，以及 `view controller` 可能需要做各种清理工作，例如，注销观察者等。

## 循环引用

当两个或多个对象互相之间有强引用的时候，就会产生循环引用，它会让这些对象都无法被释放（除非开发者显式地打破这种循环）。这会造成内存泄漏，并让那些潜在的清理任务无法执行。

因为结构体是值类型，所以在结构体之间是不会产生循环引用的（因为不存在对结构体的引用）。这即是优势又是限制：一方面我们可以少担心一件事，但同时也意味着无法用结构体实现循环数据结构（cyclical data structure）。类的情况正好相反：因为一个实例可以有多个所有者，所以可以使用类来实现循环数据结构，但必须要小心，不要产生循环引用了。

产生循环引用的情况可以有多种：从两个对象互相之间强引用，到由许多对象组成的复杂循环，以及在闭包中捕获对象。让我们先来看个由 window 以及它的 rootView 属性构成的简单例子：

```
// 第一版

class Window {
    var rootView: View?
}

class View {
    var window: Window
    init(window: Window) {
        self.window = window
    }
}

var window: Window? = Window() // 引用计数: 1
window = nil // 引用计数: 0, 销毁
```

第一行代码被执行之后，window 的引用计数成为了1。当我们把它设置为 nil 时，Window 实例的引用计数就成为了0，然后实例就被销毁了。然而，如果我们创建一个 view 并把它赋值给 window 的 rootView 属性的话，引用计数就再也不会变成0了。让我们一行行地来追踪一下它的引用计数。

首先，window 被创建，此时，它的引用计数是1：

```
var window: Window? = Window() // window: 1
```

接下来，创建一个 view，并且它有一个对 window 的强引用，此时 window 和 view 的引用计数分别为 2 和 1：

```
var view: View? = View(window: window!) // window: 2, view: 1
```

把 view 赋值给 window 的 rootView 属性，这会让 view 的引用计数加1。现在，view 和 window 的引用计数都变成了 2：

```
window?.rootView = view // window: 2, view: 2
```

在把两个变量都设置成 nil 之后，两个实例的引用计数仍然为 1：

```
view = nil // window: 2, view: 1
```

```
window = nil // window: 1, view: 1
```

即使已经无法通过变量访问到实例，但他们还是互相强引用着对方。这被称为**循环引用**，当处理类似这种彼此有引用关系的数据结构时，我们要特别留意由于循环引用而造成内存泄漏的可能性。因为循环引用的关系，在程序的生命周期中，这两个对象永远不会被销毁。

## 弱引用

为了打破循环引用，我们需要使其中一个引用变为弱引用或 unowned 引用。把一个对象赋值给一个弱引用变量，并不会改变实例的引用计数。在 Swift 里，弱引用变量是**归零 (zeroing)**的：一旦所指向的对象被销毁，变量会自动被设置成 nil。这也是为什么弱引用变量必须是可选值的原因。

为了修复上面的例子，我们会把 window 的 rootView 属性改为弱引用，这意味着此属性不会强引用 view，并且一旦 view 被销毁，它就自动变成 nil。为了看一下会发生什么，我们可以在 Window 的析构函数中添加一些 print 语句。在一个类被销毁之前，会调用 deinit 方法：

```
// 第二个版本
class Window {
```

```
weak var rootView: View?  
deinit {  
    print("Deinit Window")  
}  
}  
  
class View {  
    var window: Window  
    init(window: Window) {  
        self.window = window  
    }  
    deinit {  
        print("Deinit View")  
    }  
}
```

在下面代码中，我们仍旧各创建一个 window 和 view。和之前一样，view 还是强引用 window；但因为现在 window 的 rootView 属性被声明为了 weak，所以 window 不再强引用 view 了。这样，我们就消除了循环引用。当把变量设置为 nil 后，这些对象都会被销毁：

```
var window: Window? = Window()  
var view: View? = View(window: window!)  
window?.rootView = view  
window = nil  
view = nil  
/*  
Deinit View  
Deinit Window  
*/
```

当使用代理 (delegate) 时，弱引用是非常有用的，并且这在 Cocoa 中很常见。代理对象 (例如，一个 table view) 需要一个指向它的代理的引用，但它不应该拥有代理，否则就可能会产生一个

循环引用。因此，指向代理的引用通常都是弱引用，而另一个对象（例如，一个 view controller）的职责就是确保代理对象在需要的时候确实存在。

## Unowned 引用

但有时候，我们希望一个引用既是弱引用，但同时又不是一个可选值。例如，也许我们知道，view 永远都会拥有一个 window（所以这个属性不应该是可选值），但又不希望 view 强引用 window。对于这种情况，可以使用 unowned 关键字：

```
// 第三版

class Window {
    var rootView: View?
    deinit {
        print("Deinit Window")
    }
}

class View {
    unowned var window: Window
    init(window: Window) {
        self.window = window
    }
    deinit {
        print("Deinit View")
    }
}
```

在下面的代码中，我们可以看到，两个对象都被销毁了，效果就和上一个用弱引用的例子一样：

```
var window: Window? = Window()
var view: View? = View(window: window!)
window?.rootView = view
view = nil
window = nil
```

```
/*
Deinit Window
Deinit View
*/
```

对于 `unowned` 引用，我们的责任是，确保“被引用者”的生命周期比“引用者”要长。在这个例子中，我们必须确保 `window` 的生命周期比 `view` 长。如果 `window` 在 `view` 之前被销毁，并且之后再访问这个 `unowned` 变量的话，程序就会崩溃。

要注意的是，这里的崩溃与未定义行为是不同的。在对象中，Swift 运行时使用另外一个引用计数来追踪 `unowned` 引用。当对象没有任何强引用的时候，会释放所有资源（例如，对其他对象的引用）。然而，只要对象还有 `unowned` 引用存在，其自身所占用的内存就不会被回收。这块内存会被标记为无效，有时也称作僵尸内存（zombie memory）。被标记为僵尸内存之后，只要我们尝试访问这个 `unowned` 引用，就会发生一个运行时错误。

但我们也可以通过 `unowned(unsafe)` 来绕过这个保护机制，当访问一个被标记为 `unowned(unsafe)` 的无效引用时，行为是未定义的。

## 闭包和循环引用

在 Swift 中，类不是唯一的引用类型。我们在稍后并发一章中要提到的 actor，以及函数（也包括闭包表达式和方法）同样也是引用类型。如果一个闭包捕获了一个引用类型的变量，那么在闭包中会持有一个对这个变量的强引用。这是继之前的例子之后，另一种把循环引用引入到你代码中的主要方式。

通常的模式是这样的：对象 A 引用对象 B，对象 B 保存了一个闭包，这个闭包又引用对象 A（实际上，循环引用可能会涉及多个中间对象和闭包）。例如，我们添加一个名为 `onRotate` 的回调到上面的 `Window` 类中，这个回调是一个可选值：

```
class Window {
    weak var rootView: View?
    var onRotate: (() -> ())? = nil
}
```

如果我们设置 `onRotate` 回调，并在闭包中使用 `view` 的话，我们就产生了一个循环引用：

```
var window: Window? = Window()
var view: View? = View(window: window!)
window?.onRotate = {
    print("We now also need to update the view:\`(\String(describing: view))\"")
}
```

`view` 引用了 `window`, `window` 引用 `onRotate` 回调，这个回调引用了 `view`:

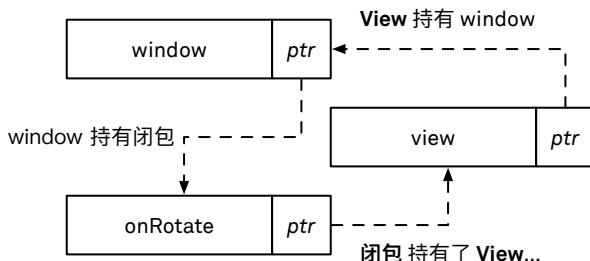


Figure 6.3: 在 `view`, `window` 和回调之间的一个循环引用

我们可以从三个地方打破这个循环引用(每个都对应上图中的一个箭头):

- 我们可以把 `view` 对 `window` 的引用改为弱引用。但不幸的是，因为没有其他强引用存在，`window` 会在创建之后立即被销毁。
- 我们可能会希望把 `onRotate` 属性标记为 `weak`，但 Swift 并不允许将类型是函数的属性标记为 `weak`。
- 我们可以使用一个捕获列表 (capture list)，并在捕获列表中弱引用 `view`，这样就能确保闭包不会强引用 `view`。在这个例子中，这是唯一正确的做法。

```
window?.onRotate = { [weak view] in
    print("We now also need to update the view: \(String(describing: view))")
}
```

捕获列表可以做的不仅仅只是将变量标记为 weak 或 unowned。比如，如果我们想有一个指向 window 的弱引用变量的话，可以直接在捕获列表中初始化这样一个变量；或者甚至可以在其中定义完全不相关的变量，像以下这样：

```
window?.onRotate = { [weak view, weak myWindow=window, x=5*5] in
    print("We now also need to update the view: \(String(describing: view))")
    print("window: \(String(describing: myWindow)), x: \(x)")
}
```

这几乎与在闭包语句之前定义变量完全相同，除了定义的地方是在捕获列表中。捕获列表中变量的作用域就在闭包之内，这些变量在闭包作用域之外是不可用的。

## 在 unowned 引用和弱引用之间做选择

在自己设计的 API 中，如何在这两种引用中做选择呢？说到底，这个问题的答案取决于对象的生命周期。如果对象具有独立的生命周期（也就是说，你不能保证哪一个对象存在的时间会比另一个长），那么弱引用是唯一安全的选择。

另一方面，如果你可以保证，非强引用的对象与持有这个引用的对象的生命周期是一样的，甚至于更长的话，unowned 引用通常是更方便的。因为它的类型不需要是可选值，并可以被声明为 let，而弱引用则必须是用 var 声明的可选值。生命周期相同的情况是很常见的，特别当两个对象之间是父子关系时。当父对象使用强引用来控制其子对象的生命周期，并且我们可以保证没有其他对象知道子对象存在的话，子对象对父对象的引用就可以是 unowned。

相比弱引用，unowned 引用的开销也小一点，通过它访问属性或调用方法的速度会快一点点。不过，应该只在对效率非常敏感的代码路径中，才把这个优点作为考虑的因素之一。

使用 unowned 引用的缺点也很明显，如果你错误地判断了对象的生命周期，你的程序可能会崩溃。就个人而言，即使可以用 unowned 引用，我们也经常发现自己更喜欢用弱引用，因为在

每个使用弱引用的地方，它都强迫我们必须检查引用是否仍然有效。特别是当重构时，`unowned` 引用很容易打破之前对于对象生命周期的设想，并引入会使程序崩溃的 bug。

不过，关于是否应该总是明确地使用对应的修饰符，来表达你所期望的代码的生命周期特性的  
问题，还是有很多争论。如果你或者其他人在稍后对代码更改中，破坏了这些假设的话，在  
测试和寻找 bug 的时候，一个强制崩溃可以让你对这个问题产生足够的重视。

## 在结构体和类之间做抉择

当设计一个类型时，我们必须考虑的是，是否会在我们程序的不同部分之间共享这个类型实例的所有权；或者，是否存在只要多个实例表示相同的值时，就可以被交换使用的情况。要共享一个实例的所有权的话，我们必须使用类。否则，我们可以使用结构体。

例如，因为 URL 是一个结构体，所以它的实例就不能被共享。每次我们把一个 URL 实例赋值给一个变量，或传递给一个函数，编译器都会生成一份它的拷贝。然而，这并不是一个问题，因为如果两个 URL 实例表示同一个 URL 的话，我们就认为它们是可交换的。这同样适用于像是整数，布尔值，字符串这样的其他结构体：我们不在意两个整数或两个字符串的背后是否共享同一块内存；我们在意的是它们是否表示相同的值。

相反，我们不会认为两个 UIView 实例是可互换的。即使所有属性都是一样的，在视图层次的不同位置，它们还是表示在屏幕上的不同“对象”。因此，UIView 是用类来实现的，以便我们可以把一个指向特定实例的引用传递给我们程序的多个部分：一个 view 被它的父视图引用，但也被它的子视图作为父视图而引用。另外，我们可以把对 view 的引用保存在其他地方，例如，保存在一个 view controller 里。可以通过所有引用修改同一个 view 实例，并且这些修改会自动反映到其他所有引用中。

话虽这么说，但当我们设计一个不需要共享所有权的类型时，其实不一定非要使用结构体。我们还是可以用类来实现，并提供一个不可变的 API，这样这个类本质上具有值语义。从这个意义上讲，我们可以只使用类，而不必大幅改变我们设计程序的方式。当然，我们会损失一些编译期的检查，并可能承担额外的引用计数的操作所带来的开销。但至少我们可以使它工作。

另一方面，如果我们没有类（或者说引用类型）可供使用的话，我们会失去整个共享所有权的概念，并且将不得不从上到下，重新架构我们的程序（这里假定的是我们之前依赖的是类）。所以虽然我们可以基于一些权衡来用类实现一个结构体，但反过来却并不一定如此。

在我们的武器库中，结构体作为一种工具，它被有意设计得不如类那么强大。作为回报，结构体提供了简洁性：没有引用，没有生命周期，没有子类。这意味着我们不必担心很多问题，仅举几个例子：循环引用，副作用，通过共享引用而产生的竞态条件以及继承规则等问题都将不复存在。

另外，结构体提供了更好的性能，特别是对简单的类型。例如，如果 Int 是一个类的话，元素类型为 Int 的数组会占用更多的内存，因为除了实例本身需要的内存之外，数组中要保存指向实际实例的引用（指针），以及每个实例需要的额外开销（例如，保存它的引用计数）。更重要的是，迭代这样一个数据会慢得多，因为对于每个元素，访问它的代码都必须经由额外的间接层，因此可能无法有效地利用 CPU 缓存。特别是如果数组中的 Int 实例被分配在内存中的位置都相距很远的话，情况就更糟糕。

## 具有值语义的类

在上面，我们已经概述了结构体具有值语义（即每个变量包含一个独立的值），而类具有引用语义（即多个变量背后可以都指向同一个类实例）的特点。虽然这没错，但我们可以实现不可变的类，让其行为上更像一个值类型；而且我们也能实现，至少第一眼看上去，不像值类型的结构体。

当实现一个类时，我们可以着眼于一个点，就是让引用语义不再对其行为产生影响。首先我们把所有的属性都声明为 let，使它们都变成不可变。其次，为了避免因为子类而重新引入任何可变行为，我们把类标记为 final 来禁止子类化：

```
final class ScoreClass {  
    let home: Int  
    let guest: Int  
    init(home: Int, guest: Int) {  
        self.home = home  
        self.guest = guest  
    }  
}  
  
let score1 = ScoreClass(home: 0, guest: 0)  
let score2 = score1
```

score1 和 score2 这两个变量持有的引用，背后还是都指向同一个 ScoreClass 实例，毕竟这是类的工作方式。但总而言之，因为背后的实例完全是不可变的，所以我们可以当它们包含的是独立的值，来使用它们。

这方面的一个例子是 Foundation 中的 NSArray 类。NSArray 本身没有暴露任何可变操作的 API，所以基本上它的实例可以当作值类型来用。但现实情况有点复杂，因为 NSArray 有一个可变的子类 NSMutableArray，所以除非一个 NSArray 的实例是你亲手创建的，否则我们就无法假定确实是在处理这样一个类型的实例。这就是为什么在上面，要把我们的类声明为 final，并且这也是为什么从一个你无法控制的 API 得到一个 NSArray 后，在做下一步之前，我们建议你先对得到的结果做一个 copy 操作。

## 具有引用语义的结构体

反过来，结构体包含引用类型属性的话，也会表现出令人惊讶的行为。让我们扩展 ScoreStruct 类型，引入一个计算属性 pretty，这个属性会根据当前的比分，返回一个漂亮的格式化过的字符串：

```
struct ScoreStruct {
    var home: Int
    var guest: Int
    let scoreFormatter: NumberFormatter

    init(home: Int, guest: Int) {
        self.home = home
        self.guest = guest
        scoreFormatter = NumberFormatter()
        scoreFormatter.minimumIntegerDigits = 2
    }

    var pretty: String {
        let h = scoreFormatter.string(from: home as NSNumber)!
        let g = scoreFormatter.string(from: guest as NSNumber)!
        return "\(h) - \(g)"
    }
}
```

```
    }  
}  
  
let score1 = ScoreStruct(home: 2, guest: 1)  
score1.pretty // 02 - 01
```

在初始化方法中，我们创建了一个 NumberFormatter 实例，并把它设置为即使分数小于 10，也至少显示两位数字。在 pretty 属性中，我们使用它来产生格式化输出。

现在，我们来生成一份 score1 的拷贝，然后在这份拷贝上设置 scoreFormatter 属性：

```
let score2 = score1  
score2.scoreFormatter.minimumIntegerDigits = 3
```

虽然我们是在 score2 上做的修改，但 score1.pretty 的输出也发生了改变：

```
score1.pretty // 002 - 001
```

会发生这种情况的原因是，NumberFormatter 是一个类，也就是说，在结构体中的 scoreFormatter 属性，包含的是指向一个 NumberFormatter 实例的引用。当我们把 score1 赋值给 score2 变量时，产生了一份 score1 的拷贝。虽然一个结构体会拷贝它所有属性值，但因为 scoreFormatter 的值只是一个引用，所以 score2 和 score1 中持有的引用，背后都指向同一个 NumberFormatter 实例。

理论上说，ScoreStruct 还是具有值语义：当你把一个实例赋值给另一个变量，或你把它作为函数的参数传递时，程序都会生成一份完整的拷贝。但是，是否具有值语义这件事，取决于我们对于值类型的认知是什么。如果我们有意要存储一个引用来作为结构体的属性之一，也就是说，我们把引用本身视为值的话，那么上面的结构体就表现的完全符合预期了。但我们可能是希望结构体包含了 NumberFormatter 实例本身，以便每个拷贝都有自己的格式化方式。对于这种情况，上面结构体的行为就不正确了。

为了避免上面例子中那种不符合预期的行为，我们要么可以把类型修改为类（这样使用这个类型的用户就不会期望它具有值语义了），要么我们可以把这个 NumberFormatter 类型的属性变为

一个私有属性，这样它就不能被外部修改了。但后面那种方案并不完美：我们还是可以（无意中）在这个类型上暴露其他的公有方法，并在这些方法中修改内部的这个 NumberFormatter 属性。

我们建议在结构体中存储引用时要非常小心，因为这样做通常都会导致意外的行为。但在某些情况下，是有意这么做的，并且也是你所需要的效果。我们将在下一节中介绍这样一个例子，其中还会涉及到写时复制 (copy-on-write) 的内容。

## 写时复制优化

对于值类型，因为赋值或作为函数的参数传递，都会产生一份拷贝，所以会有大量的复制操作。虽然编译器试图更智能地去对待是否要复制这件事，让它可以在能证明即使不复制也是安全的时候来避免产生复制操作，但对于一个值类型的实现者来说，有另外一种优化的方式来解决这个问题，那就是使用一种名为写时复制的技术来实现该类型。这对于那些持有大量数据的类型尤其重要，像是标准库中的集合类型 (Array, Dictionary, Set 和 String)，它们都在实现中使用了写时复制。

写时复制的意思是，在结构体中的数据，一开始是在多个变量之间共享的：只有在其中一个变量修改了它的数据时，才会产生对数据的复制操作。因为数组是用写时复制实现的，所以如果我们创建一个数组，并把它赋值给另外一个变量的话，数组的数据实际上并不会被复制：

```
var x = [1, 2, 3]
var y = x
```

从内部来看，`x` 和 `y` 持有的数组都包含一个指向同一块内存缓冲区的引用。此缓冲区是保存数组中实际元素的地方。当我们修改 `x`（或者 `y`）时，因为数组监测到有多个变量共享一块缓冲区，所以在做修改之前，会先产生一份这个缓冲区的拷贝。这意味着我们可以独立地修改这两个变量，并且那些昂贵的复制元素的操作，都只会发生在它必须发生的时候：

```
x.append(5)
y.removeLast()
x // [1, 2, 3, 5]
y // [1, 2]
```

对于我们自己的类型，写时复制的行为不是可以免费获得的：我们必须自己实现它，就像标准库在集合类型上实现它一样。实际上，因为标准库已经提供了处理大量数据时所需的一些最常见的类型，所以只有在极少数的情况下，才需要为自定义的结构体实现写时复制。即使我们定义了一个可以包含大量数据的结构体，在其内部我们也经常使用内建集合来表示这些数据，因此，也就能从这些内建集合的写时复制优化中受益。

然而，了解如何实现写时复制，有助于理解 Swift 中集合类型的行为以及我们应该注意的一些边界情况。

## 写时复制的权衡

在实现写时复制之前，我们要注意使用它时需要做的一些权衡。值类型的一个优点就是它们不会产生引用计数方面的开销。但是，因为实现写时复制的结构体，依赖于保存在内部的一个引用，所以这个结构体每产生一份拷贝都会增加这个内部引用的引用计数。实际上，我们是放弃了值类型不需要引用计数的这个优点，来减轻值类型的复制语义这个特性所可能带来的成本。

增加或减少一个引用计数，都是一个相对较慢的操作 (这里的慢，比较的是把一些字节复制到栈上另一个位置之类的操作)。因为这样一个操作必须是线程安全的，因此就会有锁的开销。由于标准库中所有可变长度的类型 (数组，字典，集合，字符串)，内部都依赖于写时复制，所以对于含有这些类型的属性的结构体，每次复制也都会带来操作引用计数的开销。甚至当含有多个这种类型的属性时，这种开销会很多次。这里有一个例外情况，对于长度最多只有 15 个 UTF-8 编码单元 (code unit) 的短字符串，Swift 实现了一种优化来避免为其分配一个缓冲区。

SwiftNIO 项目中就有一个实际的例子：在这个项目中，一个 HTTP 请求就是用结构体来实现的，它包含了多个属性，像是 HTTP 方法和头。当这样一个结构体被复制时，不仅要复制它所有的字段，而且所有内部的数组，字典和字符串的引用计数也都会增加。当传递这种类型的值时 (这种操作很常见)，这种开销会导致性能的显著下降，而用类实现它的话，性能会好很多，因为一个对类的引用，要比 HTTP 请求结构体的所有字段更小，并且也只需要更新这一个引用的引用计数。

接下来，我们会看一下在这种特殊情况下，如何使用写时复制技术来结合这两种类型的优点：具有值语义的同时，兼具使用类的性能优势。在 dotSwift 2019 上，SwiftNIO 团队的 Johannes Weiss 对此也有一个很棒的分享。

## 实现写时复制

我们从一个极其简单的版本开始，用结构体实现 `HTTPRequest`:

```
struct HTTPRequest {  
    var path: String  
    var headers: [String: String]  
    // 忽略其他字段。。。  
}
```

为了最小化在上面提到的引用计数的开销，首先，我们会把结构体所有属性都封装到一个私有的 `Storage` 类中：

```
struct HTTPRequest {  
    fileprivate class Storage {  
        var path: String  
        var headers: [String: String]  
        init(path: String, headers: [String: String]) {  
            self.path = path  
            self.headers = headers  
        }  
    }  
  
    private var storage: Storage  
  
    init(path: String, headers: [String: String]) {  
        storage = Storage(path: path, headers: headers)  
    }  
}
```

这样做的话，`HTTPRequest` 结构体只会包含 `storage` 这一个属性，并在复制时，只需要增加这一个内部的 `Storage` 实例的引用计数。现在，为了把私有的 `Storage` 实例的 `path` 和 `headers` 属性暴露给外部，我们在结构体上增加相应的计算属性：

```
extension HTTPRequest {
    var path: String {
        get { storage.path }
        set {/* to do */}
    }
    var headers: [String: String] {
        get { storage.headers }
        set {/* to do */}
    }
}
```

在这个实现中，属性的 setter 是最重要的部分：因为存储在内部的 Storage 实例有可能被多个变量所共享，所以在这些 setter 中，我们不能只是简单的在 Storage 实例上设置新的值。由于，把请求的相关数据都保存在一个类的实例中，这样的实现细节不应该被暴露出去，所以我们必须保证这个基于类的结构体的行为，和原始的纯结构体版本是一致的。这意味着，修改 HTTPRequest 类型变量的一个属性时，受到改变影响的应该只是这个变量而已。

首先，每次属性的 setter 被调用的时候，我们就生成一份内部 Storage 类实例的拷贝。为了生成拷贝，在 Storage 上我们添加一个 copy 方法：

```
// Shadow Swift.print to capture print output for testing.
var printedLines: [String] = []
func print(_ value: Any) {
    var output = ""
    Swift.print(value, terminator: "", to: &output)
    printedLines.append(output)
    Swift.print(output)
}

extension HTTPRequest.Storage {
    func copy() -> HTTPRequest.Storage {
        print("Making a copy...") // 调试语句
        return HTTPRequest.Storage(path: path, headers: headers)
    }
}
```

```
}
```

然后，在设置新的值之前，我们产生一份拷贝并赋值给 storage 属性：

```
extension HTTPRequest {  
    var path: String {  
        get {  
            storage.path  
        }  
        set {  
            storage = storage.copy()  
            storage.path = newValue  
        }  
    }  
  
    var headers: [String: String] {  
        get {  
            storage.headers  
        }  
        set {  
            storage = storage.copy()  
            storage.headers = newValue  
        }  
    }  
}
```

虽然现在 `HTTPRequest` 结构体的背后完全由一个类实例所支撑，但它仍然展现出了值语义，它的所有属性就好像是直接定义在结构体本身一样：

```
let req1 = HTTPRequest(path: "/home", headers: [:])  
var req2 = req1  
req2.path = "/users"  
assert(req1.path == "/home") // 通过
```

但当前的实现还不够高效。无论是否有其他变量引用同一个 Storage 实例，只要我们修改属性，就会创建一份内部 Storage 实例的拷贝：

```
var req = HTTPRequest(path: "/home", headers: [:])
for x in 0..<5 {
    req.headers["X-RequestId"] = "\((x)"
}
/*
Making a copy...
*/
```

每次我们修改 `request`，就会生成一份 Storage 的拷贝。但所有这些拷贝都是不需要的，因为只有 `req` 这一个变量持有指向 Storage 实例的引用。

为了实现一个高效的写时复制，我们需要知道，一个对象（在我们的例子中是 Storage 实例）是否被唯一引用，也就是说，它是否只有一个所有者。如果是的话，我们可以直接修改对象，否则，我们在修改之前，创建一份对象的拷贝。

我们可以使用 `isKnownUniquelyReferenced` 函数来检查一个引用类型的实例是否只有一个所有者。如果你把一个 Swift 的类实例传递给此函数，并且这个实例没有其他强引用的话，这个函数就返回 `true`，反之，如果有其他强引用存在，此函数返回 `false`。

使用 `isKnownUniquelyReferenced` 时，请务必牢记以下这些细微的地方：

虽然这个函数是线程安全的，但是，你必须保证传入的变量不会被另外一个线程所访问，这个限制不单单只是针对 `isKnownUniquelyReferenced`，它适用于所有的 `inout` 参数。换言之，`isKnownUniquelyReferenced` 不能防止竞争条件（race condition）。以下的代码就是不安全的，因为两个队列会同时修改同一个变量：

```
var numbers = [1, 2, 3]
queue1.async { numbers.append(4) }
queue2.async { numbers.append(5) }
```

`isKnownUniquelyReferenced` 的参数是一个 `inout` 参数，因为在 Swift 中，这是在函数参数的上下文中引用一个变量的唯一方式。如果是普通（非 `inout`）参数的话，当调用函数时，编译器就会产生一份参数的拷贝，也就是说，`isKnownUniquelyReferenced` 这个函数肯定会返回 `false`，因为在函数中，被测试对象的引用，不可能是唯一的。

`unowned` 引用和弱引用并不被计算在内，即我们不能把此类变量作为参数传入这个函数，否则函数总是会返回 `false`。

这个函数并不能作用于 Objective-C 的类。如果想绕过这个限制的话，我们可以把一个 Objective-C 类的实例封装在一个 Swift 类中。

利用这一点，现在我们可以继续对 `HTTPRequest` 做出改进：在更改 `storage` 之前，先检查其引用是否唯一。为了避免在每个属性的 `setter` 中都写一遍这个检查的语句，我们将把这部分的逻辑封装到一个 `storageForWriting` 属性中：

```
extension HTTPRequest {
    private var storageForWriting: HTTPRequest.Storage {
        mutating get {
            if !isKnownUniquelyReferenced(&storage) {
                self.storage = storage.copy()
            }
            return storage
        }
    }

    var path: String {
        get { storage.path }
        set { storageForWriting.path = newValue }
    }
}
```

```
var headers: [String: String] {
    get { storage.headers }
    set { storageForWriting.headers = newValue }
}
```

为了测试我们的代码，让我们再写一遍之前的循环：

```
var req = HTTPRequest(path: "/home", headers: [:])
var copy = req
for x in 0..<5 {
    req.headers["X-RequestId"] = "\\"(x)"
} // Making a copy...
```

调试的语句只在我们第一次修改 `req` 的时候被打印了一次。在之后的迭代中，因为引用都是唯一的，所以没有产生复制。结合编译器的优化，写时复制可以避免大多数不必要的复制值类型的操作。

## willSet 对写时复制的破坏

这里有一个性能上的陷阱需要注意：在写时复制类型属性上的 `willSet` 将会破坏写时复制的优化。这是因为 `willSet` 将会在它的函数体中用到 `newValue`，这导致编译器必须去为这个值创建一个临时的复制。这样一来，这个值就不再是唯一引用的了，任何对该属性的变更，将会触发一次复制。

为了看到这一点，我们来定义三个 `HTTPRequest` 的属性，并为它们附上不同的变更观察者：

```
struct Wrapper {
    var reqWithNoObservers = HTTPRequest(path: "/", headers: [:])
    var reqWithWillSet = HTTPRequest(path: "/", headers: [:]) {
        willSet {
            print("willSet")
        }
    }
}
```

```
var reqWithDidSet = HTTPRequest(path: "/", headers: [:]) {  
    didSet {  
        print("didSet")  
    }  
}  
}
```

这些属性都是独立的值，它们中的每一个都拥有自己单独的引用存储。所以变更这些值应该不会触发写时复制。对于不加修饰的属性以及 didSet 属性来说，确实如此：

```
var wrapper = Wrapper()  
wrapper.reqWithNoObservers.path = "/about"  
wrapper.reqWithDidSet.path = "/forum"  
// didSet
```

然后，标记了 willSet 却在变更时触发了写时复制：

```
wrapper.reqWithWillSet.path = "/blog"  
/*  
Making a copy...  
willSet  
*/
```

要理解这个行为，我们必须考虑这些操作的顺序。在 willSet 被调用之前，已存在的值会被复制一次。然后这个复制被改变，随后被改变后的复制值作为 newValue 去调用 willSet，最后属性的存储发生变更。对于 didSet 来说，行为是类似的：在变更之前，现有值也会被复制一份，然后这个复制会被作为 oldValue 来使用。但是，在上面这个例子中，编译器会检测到我们在 didSet 中并没有使用到 oldValue，它会将这个复制给优化掉。但对于 willSet，编译器并不会对它做类似的事情。

假设一下，要是 Swift 也对 willSet 做了这个优化，那么是否使用了 newValue 将会改变执行的顺序：如果你没有使用 newValue 的话，编译器将会需要在实际变更之前调用 willSet，那么在 willSet 中的任意副作用将会比属性变更所导致的副作用更早发生。然而现在，属性变更中的副

作用是早于 willSet 的。现在来改变这种行为可能会导致依赖这些操作顺序的程序被破坏。对于 didSet 来说并没有这个问题，所以 Swift 团队可以改变它的语义。

由于 ObservableObject 所使用的 @Published 属性在底层使用了 willSet，所以 willSet 的语义可能会带来 SwiftUI 代码的性能问题。比如，下面这个作为一个很长的 list view 的数据源的数组：

```
class ViewModel: ObservableObject {  
    @Published var cities: [String]  
}
```

每次 cities 属性的变更，都会导致底层数组的存储被复制一次。

## 回顾

在这一章，我们看到了尽管结构体和类之间有一些共同点，但因为一个是值类型，另一个是引用类型，所以它们之间的行为差别很大。一个值类型直接持有值本身，并且在赋值给其他变量，或传递给一个函数的时候，会创建一个这个值的拷贝。相比之下，一个引用类型持有的是一个指向实例的引用。把它赋值给另一个变量，或传递给一个函数的话，只会创建一个对于引用的拷贝，而并不会对背后所指向的实例做复制操作。

我们还讨论了如何使用 let 和 var 来控制可变性，mutating 关键字是如何工作的，以及如何使用 inout 参数。最后，我们还展示了写时复制是如何工作的，以及如何在你自己的结构体上实现这个特性。

# 枚举

7

我们在前一章讨论的结构体和类都是记录类型 (record type)。一个记录由零个或多个具有类型的字段 (属性) 组成。元组也属于记录类型：实际上它是一个功能较少的轻量级的匿名结构体。语言支持记录类型是一件理所应当的事。几乎所有语言都允许你来定义这类组合类型 (早期的 BASIC 和最初的 Lisp，或许是最著名的例外了)。即使汇编程序员得不到语言层面的支持，他们也总是使用记录的概念来组织内存中的数据。

Swift 的枚举属于一个完全不同的类别，有时称它为标签联合或变体类型 (variant type)。尽管它和记录一样强大，但在主流编程语言中对其支持却不那么普遍。然而它在函数式语言中却是司空见惯的，并已在 Rust 这样较新的语言中流行起来。在我们看来，枚举是 Swift 最好的特性之一。

## 概述

一个枚举由零个或多个成员 (case) 组成，每个成员都可以有一个元组样式的关联值 (associated value) 列表。在这一章中，当我们讨论单个成员的关联类型时，有时我们会忽略具体的数量。一个成员可以有多个关联值，但你可以把这些值视为单个元组。

下面是一个表示段落对齐方式的枚举，这些成员都没有关联值：

```
enum TextAlignement {  
    case left  
    case center  
    case right  
}
```

我们在可选值这一章看到过，Optional 是一个具有 none 和 some 这两个成员的泛型枚举。some 有一个关联值用来表示当前的装箱值 (boxed value)：

```
@frozen enum Optional<Wrapped> {  
    /// 没有值。  
    case none  
    /// 存在一个值，保存为 `Wrapped`。  
    case some(Wrapped)  
}
```

(现在先忽略 @frozen 属性。我们会在稍后固定和非固定枚举这一节中讨论它。)

Result 类型则用来表示一个操作的成功或失败，它的结构和可选值类似，区别是它增加了一个泛型参数，并把这个参数设置为 failure 的关联值，以便可以获得详细的错误信息：

```
@frozen enum Result<Success, Failure: Error> {  
    /// 成功, 保存一个 `Success` 值。  
    case success(Success)  
    /// 失败, 保存一个 `Failure` 值。  
    case failure(Failure)  
}
```

在错误处理这一章中我们会讨论 Result，并在很多例子中使用它。

你可以通过指定一个成员以及它的关联值(如果有的话)来创建一个枚举值：

```
let alignment = TextAlignment.left  
let download: Result<String, NetworkError> = .success("<p>Hello world!</p>")
```

注意上面第二行，我们必须提供完整的类型标注，包括所有的泛型参数。除非编译器可以从上下文中推断出另一个 Failure 泛型参数的具体类型，否则像 Result.success(htmlText) 这样的表达式就会产生错误。一旦指定了完整类型，我们就可以依靠着类型推断来使用前导点 (leading-dot) 语法了(这里忽略了 NetworkError 的定义)。

## 枚举是值类型

就像结构体一样，枚举也是值类型。它的能力几乎和结构体相同：

- 枚举可以有方法，计算属性和下标操作。
- 方法可以被声明为可变或不可变。
- 你可以为枚举实现扩展。
- 枚举可以实现各种协议。

但是，枚举不能拥有存储属性。一个枚举的状态完全由它的成员和成员的关联值组合起来表示。对于某个特定的成员，可以将关联值视为其存储属性。

枚举和结构体中的可变方法的工作方式是一样的。在结构体和类这一章中我们看到过，在一个可变方法中，因为 `self` 是 `inout` 参数，所以它是可变的。由于枚举没有存储属性，并且没有办法直接改变一个成员的关联值，所以修改一个枚举的方式就是直接分配一个新的值给 `self`。

因为通常会用一个枚举的成员来初始化枚举类型的变量，所以枚举不太需要一个显式的初始化方法。但也有可能会需要在类型定义或扩展中添加另外的简便初始化方法 (convenience initializer)。例如，使用 Foundation 的 Locale API 时，我们可以为 `TextAlignment` 枚举添加一个初始化方法，这个方法根据传入的 `locale` 参数来设置一个默认的文本对齐方式：

```
extension TextAlignment {
    init(defaultFor locale: Locale) {
        guard let language = locale.languageCode else {
            // 以下是没能获得当前语言时的默认值。
            self = .left
            return
        }
        switch Locale.characterDirection(forLanguage: language) {
        case .rightToLeft:
            self = .right
            // Left 是对于其他情况的默认值。
        case .leftToRight, .topToBottom, .bottomToTop, .unknown:
            self = .left
        @unknown default:
            self = .left
        }
    }
}

let english = Locale(identifier: "en_AU")
TextAlignment(defaultFor: english) // left
let arabic = Locale(identifier: "ar_EG")
```

```
TextAlignment(defaultFor: arabic) // right
```

(我们会在固定和非固定枚举这一节中讨论 @unknown default 分支)

## 总和类型和乘积类型

一个枚举值只会包含一个枚举成员 (如果这个成员有关联值的话，则再加上关联值)。事实上，在 Swift 早期的日子里 (在第一个公开的版本之前)，枚举曾经被称为“之一”(oneof)，后来又被称为“联合”(union)。具体来说，一个 Result 类型变量包含的值要么是 success 要么是 failure，但不会两者都包含 (也不会什么都不包含)。相对的，一个记录类型的实例包含了它所有字段的值：一个 (String, Int) 类型的元组包含一个字符串和一个整型。注意，在这里我们讨论的是具有多个字段的复合记录，所以虽然 UInt8 也是一个结构体，并且你也许会说，它就将实例的值限制为 0 到 255 之间的一个数，但这和我们在这里讨论的不是一个意思。

这种实现”或”关系的能力是相当特别的，它使得枚举变得非常有用。通常在无法用结构体、元组或类来清晰地表达逻辑的情况下，枚举允许我们充分利用强类型来写出更安全，更清晰的代码。

我们之所以说是“相当特别的”，是因为尽管权衡和应用会有所不同，但也可以用协议和子类达到同样的目的。一个协议类型的变量 (也被称为existential) 可以是实现这个协议的任意类型之一。类似地，在iOS里，一个 UIView 类型的对象也可以指向任意一个 UIView 的直接或间接子类，例如 UILabel 或 UIButton。当操作这样一个对象时，我们要么使用定义在基类的公共接口 (等同于调用定义在枚举上的方法)，要么为了访问某个具体子类中特有的数据，而尝试把实例向下转换成这个子类 (等同于转换一个枚举值)。

通过协议和类的公共接口做动态派发，或是使用枚举进行 switch，这两种方法的区别在于哪种更通用，以及两种结构所特有的功能和限制。比如，一个枚举中所有的成员是固定的，无法在除了声明之外的地方扩展它，但你总是可以让多个类型满足同一个协议，或添加另外一个子类 (除非你声明一个类为 open，否则跨模块的子类化是禁止的)。这种自由是否可取，甚至于说是否需要，则取决于要解决的问题。作为值类型，枚举通常是更轻量，更适合于实现 POD (plain old values) 的。

这两种类型（“或”，“和”）和数学概念中加法与乘法之间，有着一一对应的关系。对于想成为一名优秀的 Swift 程序员来说，了解它并不是必须的，但我们发现，当在设计自定义类型时，它提供了一个有用的思路。

术语“类型”的定义可能有多种。这里给出一个定义：一个类型，是它的实例所能表示的所有可能的值的集合，值也被称为居留元 (inhabitant)。Bool 有两个居留元，false 和 true。UInt8 有  $2^8$ （也就是 256）个居留元。Int64 有  $2^{64}$ （也就是大约 1.84 千京，千京等于  $10^{19}$ ）个居留元。像是字符串这种类型，至少在填满内存之前，会有无穷多的居留元，因为你总是可以通过追加一个字符来创建另一个字符串。

现在考虑一个有两个布尔字段的元组：(Bool, Bool)。这个类型有多少个居留元呢？答案是 4：(false, false), (true, false), (false, true) 以及 (true, true)。除了这 4 个之外，不可能用其他值来构造这个类型。如果我们再增加一个 Bool 字段，使它成为 (Bool, Bool, Bool) 的话，居留元的数量会变为多少呢？因为之前 4 个居留元中的每一个，都可以分别与 false 和 true 相结合，所以居留元的数量翻倍为 8。当然这不仅只适用于字段都是 Bool 的情况。一个类型为 (Bool, UInt8) 的元组，因为 UInt8 有 256 个居留元，并且每个居留元又都可以与两个布尔值相结合，所以这个元组的居留元数量就是  $2 \times 256$ , 512 个。

通常来说，一个元组（或者结构体，类）的居留元数量，等于其成员的居留元数量的乘积。因此，结构体，类和元组也被称为乘积类型 (Product Types)。

将这种类型与枚举进行比较。下面是一个有三个成员的枚举：

```
enum PrimaryColor {  
    case red  
    case yellow  
    case blue  
}
```

这个类型有三个居留元，每个对应一个枚举的成员。除了 .red, .yellow 或 .blue 之外，不可能用其他值来构建它。那如果我们往这个类型中增加一个带关联值的成员，居留元数量会发生什么变化呢？让我们在其中添加第四个成员，这个成员允许我们在 0 (黑色) 到 255 (白色) 之间，指定一个灰度值：

```
enum ExtendedColor {  
    case red  
    case yellow  
    case blue  
    case gray(brightness: UInt8)  
}
```

单就 `.gray` 来说，可能的值的数量是 256，所以整个枚举的居留元数量就为  $3 + 256 = 259$  个。一般而言，一个枚举的居留元数量，等于它所有成员的居留元数量的总和。这也是为什么称枚举为 **总和类型 (Sum Types)** 的原因。

向结构体增加一个字段会增加可能状态的数量，而且这个数量通常会非常大。而往枚举中添加一个成员却只会另外增加一个居留元 (如果这个成员有关联值的话，增加的数量就是关联值的居留元数量)。对于实现安全的代码，这是一个非常有用的特性。在稍后的 使用枚举进行设计 这一节中，会涵盖关于如何在我们自己的代码中利用这个特性的内容。

## 模式匹配

为了用枚举值来做点有用的事，通常我们必须检查枚举成员并提取它的关联值。以可选值为例：涉及它的每个操作，像是 `if let` 绑定，可选链或调用 `Optional.map`，其实都是对于解包 `some` 成员的关联值并进一步处理这个值的一种简写。如果发现值为 `none` 的话，通常就会中止这些操作。

检查一个枚举值最常用方法就是使用 `switch` 语句，它允许我们在单个语句中，将值与多个候选值 (`candidates`) 进行比较。使用 `switch` 语句还有一个额外的好处，就是它有一个方便的语法，可以一口气做完值与成员的比较和提取关联值这两件事。这个机制被称为 **模式匹配**。模式匹配不是 `switch` 所独有的，但却是最明显的用例。

模式匹配是有用的，因为它让我们可以通过结构而不是内容，来解构一个数据结构。将纯匹配与值绑定相结合的能力使其特别的强大。

一个 `switch` 语句的每个分支都由一个或多个和输入值相匹配的模式组成。一个模式描述了值的结构。例如，在下面的例子中，模式 `.success((42, _))` 匹配枚举中带有一对关联值的

success 成员，并且关联值的第一个元素等于 42。这个模式中的下划线是一个通配符模式，它表示第二个元素可以是任意值。除了这种普通匹配，我们还能提取关联值的某个部分，并把它们和变量进行绑定。还是下面这个例子，模式 .failure(let error) 匹配 failure 成员，并且把关联值绑定到一个新的局部常量 error 中：

```
let result: Result<(Int, String), Error> = ...
switch result {
    case .success((42, _)):
        print("Found the magic number!")
    case .success(_):
        print("Found another number")
    case .failure(let error):
        print("Error: \(error)")
}
```

让我们看一下Swift 支持的模式类型：

**通配符模式** - 符号为下划线：`_`。它匹配任意值并忽略这个值。当匹配到关联值的一部分，并想忽略另一部分的时候，就可以使用它了。在上面的代码中，我们已经在 `.success((42, _))` 模式中看到了用通配符的例子。在 `switch` 语句中，`case _` 是等同于 `default` 关键字的：两者都匹配任意值，并且把它们作为最后一个分支才是合理的。

**元组模式** - 使用一个用逗号分割的子模式 (subpattern) 列表来匹配元组。例如，`(let x, 0, _)` 匹配的是一个含有三个元素的元组，其中第二个元素为 0，然后把第一个元素与 `x` 做绑定。元组模式本身只匹配元组的结构，也就是说，只匹配在括号内用逗号分割的元素数量。对于元组内容的匹配则是通过各个子模式来做的。`(let x, 0, _)` 中就有三个子模式：值绑定模式 (value-binding pattern)，表达式模式 (expression pattern) 和通配符模式。当需要在单个 `switch` 语句中转换多个值时，这个模式是非常有用的。

**枚举成员模式** - 匹配指定的枚举成员。它可以包含子模式来处理关联值，像是等式检查 (`.success(42)`) 或值绑定 (`.failure(let error)`) 这种。为了忽略一个关联值，可以在子模式中使用下划线，或删除整个子模式来达到目的，例如，`.success(_)` 和 `.success` 就是等价的。

如果你想提取一个成员的关联值，或是只想匹配成员而忽略掉关联值的话，这个模式是唯一的方法。为了与某个含有特定关联值的特定成员做比较，只要这个枚举实现了 Equatable 协议，你就可以在 if 语句中使用 == 操作符来做比较。

**值绑定模式** - 把一个匹配值的部分或全部，绑定到一个新的常量或变量上。语法是 let someIdentifier 或 var someIdentifier 这样。绑定变量的作用域就在其声明的那个 case 语句块中。

作为在单个模式中绑定多个值的一种简写方式，你不需要在每个绑定变量前重复的写 let，只需要在模式前加一个 let 前缀就可以了。所以模式 let (x, y) 和 (let x, let y) 是一样的。请注意在单个模式中同时使用值绑定和等式匹配时，两者的细微差别：例如，模式 (let x, y) 中把元组的第一个元素和一个新的常量做绑定，但对于第二个元素，模式只是拿它与一个现有的变量 y 做比较。

为了把值绑定和其他一些绑定值所必须满足的条件相结合，你可以使用 where 子句来扩展一个值绑定模式。例如，模式 .success(let httpStatus) where 200..<300 ~ = httpStatus 只会匹配 success 且关联值在指定范围内的值。非常重要的一点是，因为 where 子句是在值绑定之后执行的，所以我们才能在子句中使用绑定的值（有关模式匹配操作符 ~= 的更多信息，请参阅下面“表达式模式”中的内容）。

如果你在单个分支中包含了多个模式，那么各个模式中值绑定模式的数量，以及每个值绑定所用的变量名和类型都必须相同。假设你要处理以下这个枚举：

```
enum Shape {  
    case line(from: Point, to: Point)  
    case rectangle(origin: Point, width: Double, height: Double)  
    case circle(center: Point, radius: Double)  
}
```

我们可以注意到，虽然每个成员的关联值都包含了所对应形状的原点，但其他参数则根据形状的类型不同而有所不同。尽管如此，还是可以用一个包含三个模式的 case 语句来提取形状的原点：

```
switch shape {
```

```
case .line(let origin, _),
.rectangle(let origin, _, _),
.circle(let origin, _):
    print("Origin point:", origin)
}
```

在这种情况下，你就不能再引入像是对于 circle 的 radius 关联值这一类的绑定了。因为当匹配到其中某个模式时，编译器要保证每个绑定的变量都含有一个有效的值，因此编译器必须能够给每个变量赋值一个有效值，而如果 shape 是一个 line 或 rectangle 的话，就不能对 radius 做这个操作了。

**可选值模式** - 通过使用我们所熟悉的问号语法，为匹配及解包可选值这两个操作，提供了一个语法糖。模式 let value? 等价于 .some(let value)，也就是说，它匹配一个不为 nil 的可选值，并把解包出来的值和一个常量绑定。

正如在可选值这一章所见，我们也能使用 nil 来匹配一个可选值的 none 成员。这个简写方式并没有使用任何编译器的黑魔法，标准库为了和 nil 做比较而重载了 ~= 操作符，让其作为一个普通的表达式模式（对于这个模式，请看下面的详细介绍）。

**类型转换模式** - 模式 is SomeType 匹配成功的条件是，一个值的运行时类型必须是 SomeType 或是其子类。模式 let value as SomeType 会执行同样的检查，并且另外还会将匹配的值转换为指定的类型，而 is 只会检查类型：

```
let input: Any = ...
switch input {
    case let integer as Int: ... // integer 的类型是 Int。
    case let string as String: ... // string 的类型是 String。
    default: fatalError("Unexpected runtime type: \(type(of: input))")
}
```

**表达式模式** - 通过把输入值和模式作为参数传递给定义在标准库中的模式匹配操作符 (~=) 来匹配表达式。对于实现 Equatable 协议的类型，~= 的默认实现就是转发到 == 中，这也是在模式中简单的等式检查的工作方式。

标准库还为范围 (ranges) 提供了 `~=` 的重载。特别是和单边范围 (one-sided ranges) 相结合的时候，这个重载为检查一个值是否在一个范围内提供了漂亮的语法。下面的 `switch` 语句会检查一个数字是否是正数、负数或是零：

```
let randomNumber = Int8.random(in: .min...( .max))  
switch randomNumber {  
    case ..<0: print("\(randomNumber) is negative")  
    case 0: print("\(randomNumber) is zero")  
    case 1...: print("\(randomNumber) is positive")  
    default: fatalError("Can never happen")  
}
```

需要注意的是，因为编译器无法确定这三个具体的分支是否能覆盖所有可能的输入（即使它们确实是可能覆盖的），所以它还是会强制我们添加了一个 `default` 分支。`switch` 语句必须始终是完备的 (exhaustive)。我们会在 使用枚举进行设计 这一节中，讨论更多关于完备性检查的内容。

我们可以通过重载自定义类型的 `~=` 操作符来扩展模式匹配系统。实现 `~=` 的函数必须具有以下的格式：

```
func ~= (pattern: ???, value: ???) -> Bool
```

可以自由选择参数的类型，它们甚至不必是一样的。编译器会选择对于输入值类型最精确的重载版本。对于编译器碰到的每个表达式模式，它会执行 `pattern ~= value` 这个表达式，其中的 `value` 就是我们在 `switch` 语句中操作的值，而 `pattern` 就是每个 `case` 语句中的模式。如果表达式返回 `true` 则表示匹配成功。

应该注意到，除了这些为了说明原理而举的例子之外，我们从未发现需要用这种方式来扩展模式匹配系统。标准库已经很好的覆盖了所有的基本情况，并且任何想超出这些基本情况的东西，都会因为无法将基于自定义 `~=` 操作符的模式匹配与值绑定和通配符模式相结合而受到很大影响。

## 在其他上下文中的模式匹配

虽然模式匹配是从枚举中提取关联值的唯一方式，但它不是专属于枚举或 switch 语句的。事实上，即使像 let `x = 1` 这样的赋值语句，也能被视为是一个值绑定模式：用赋值操作符左边的变量来匹配右边的表达式。另外一些模式匹配的例子包括：

**在赋值时解构元组**，例如，`let (word, pi) = ("Hello", 3.1415)` 或迭代时的 `for (key, value) in dictionary { ... }`。请注意在 for 循环的例子中，我们没有使用 let 来指明这是一个值绑定。因为在这种情况下，默认所有标识符都是值绑定。for 循环也支持 where 子句，比如，`for n in 1...10 where n.isMultiple(of: 3) { ... }` 只会在 `n` 为 3, 6, 及 9 的情况下才会执行循环体。

在解构嵌套元组时，可以把元组模式嵌入解构值中。例如：

```
for (num, (key: k, value: v)) in dictionary.enumerated() { ... }.
```

**使用通配符来忽略我们不感兴趣的值**，例如，`for _ in 1...3` 会执行 3 次循环而不为循环计数创建一个变量，或当我们要执行一个有副作用的函数时，`_ = someFunction()` 会避免编译器产生“未使用结果”的警告。

**在一个 catch 子句中捕获错误**：例如，`do { ... } catch let error as NetworkError { ... }` 这样。关于这个的详细信息，请参阅 [错误处理这一章](#)。

**if case 和 guard case 语句**类似于只有单个分支的 switch 语句。尽管在许多情况下，我们为了利用编译器的完备性检查而更喜欢用 switch 语句，但因为这两个语句所需的行数要少于 switch，所以它们偶尔还是有用的。

对于 Swift 的新手来说，if/guard case [let] 的语法经常是一个很大的阻碍。我们认为会造成这种情况的原因是，它使用赋值操作符来进行基本的比较操作，并且可以不包含值绑定。例如，以下的代码用来测试一个枚举值是否等于一个特定的成员，但同时又忽略其关联值：

```
let color: ExtendedColor = ...
if case .gray = color {
    print("Some shade of gray")
}
```

你可以把这里的赋值操作符视为“在操作符右边的值和左边的模式之间做一个模式匹配”。当你引入值绑定时，这个语法会变得更清晰，语法是相同的你只需要在之前的语句中加上 let 或 var 就可以了：

```
if case .gray(let brightness) = color {  
    print("Gray with brightness \(brightness)")  
}
```

这和我们熟悉的对可选值使用的 if let  $x = x$  语法并没有什么大的区别。实际上，Swift 的创造者 Chris Lattner 对 Swift 开发者选择添加 if case [let] 表达过后悔。如果针对可选值的 if let 语法能使用包括问号的 (if let  $x? = x$ ) 真正的可选值模式匹配的话，这门语言就可以在 if 条件中接受任意有效的模式，而不再需要 if case 语句了。

**for case** 和 **while case** 循环的工作方式类似于 if case。它们允许你仅在模式匹配成功时才执行循环体。有关示例请参阅[可选值这一章](#)。

最后，有时闭包表达式的参数列表看起来像是模式，因为它们也支持一种元组解构。例如，对于字典的 map 方法，即使传入的参数被指定为单个 Element，但对字典执行 map 操作时，我们可以在执行转换的闭包中使用一个 (key, value) 参数列表 (Dictionary.Element 的类型是一个 (Key, Value) 元组)。

```
dictionary.map { (key, value) in  
    ...  
}
```

这里的 (key, value) 看上去像一个元组，但实际上它只是一个拥有两个元素的参数列表。我们之所以能在这里把元组解包成参数列表，是由于编译器对此有特殊处理的缘故，和模式匹配没有关系。没有这个特性的话，我们将不得不使用类似 { element in ... } 这样的单项参数列表，然后在闭包中用两行代码把 element (现在它真正是一个元组了) 解构成 key 和 value。

## 使用枚举进行设计

因为相比结构体和类，枚举属于不同的类别，所以适用于它的设计模式也是不同的。又由于在主流的编程语言中，真正的总和类型是一种相对不常见的（如果语言快速发展的话）特性，因此相比传统的面向对象方法，你可能会不习惯于使用它。

那么让我们看一下在代码中，可以使用的一些充分利用了枚举各种特性的模式。我们将它们分为六个主要方面：

0. Switch 语句的完备性
1. 不可能产生非法的状态
2. 使用枚举来实现 Model 状态
3. 在枚举和结构体之间做选择
4. 枚举和协议之间的相似之处
5. 使用枚举实现递归数据结构

## Switch 语句的完备性

多数情况下，switch 只是对于带有多个 else if case 条件的 if case 语句的一种更方便的语法而已。除了语法差异之外，它们之间还有一个重要的区别：一个 switch 语句必须是完备的，也就是说，它的分支必须覆盖所有可能的输入值。编译器也会强制执行这个完备性。

对于实现安全的代码及在程序修改时保持代码正确来说，完备性检查是一种重要工具。每次你增加一个成员到一个现有的枚举时，编译器会在所有对这个枚举使用 switch 语句的地方发出警告，提醒你需要处理这个新加的成员。if 语句并不会执行完备性检查，它也不会作用于一个包含 default 分支的 switch 语句 - 因为 default 可以匹配任意值，所以这样一种 switch 语句是永远不会有完备的。

因此我们建议你尽可能避免在 switch 语句中使用 default 分支。当然了你是无法完全避免它的，因为编译器有时并不足够聪明，无法确定一组分支是否真的是完备的。我们已经在之前的例子中看到过这样的情况了，当 switch 一个 Int8 类型的变量时，尽管我们的模式覆盖了所有可能的值，但还是需要一个 default 分支。编译器只会在安全性方面出错，也就是说，它永远不会将一组非完备的模式报告为完备的。

然而当 switch 枚举时，是不会发生漏判 (False negative) 的。对于以下的类型，完备性检查是完全可以信赖的：

- 布尔值
- 枚举，只要任何关联值可以被检测出是完备的，或者你用某个模式来匹配任意的关联值（例如，通配符或值绑定）
- 元组，只要它的成员类型可以被检测出是完备的

让我们看一个例子。以下的代码中，我们 switch 之前已经定义过的 Shape 枚举：

```
let shape: Shape = ...
switch shape {
    case let .line(from, to) where from.y == to.y:
        print("Horizontal line")
    case let .line(from, to) where from.x == to.x:
        print("Vertical line")
    case .line(_, _):
        print("Oblique line")
    case .rectangle, .circle:
        print("Rectangle or circle")
}
```

我们加了两个 where 子句来处理水平 (y 坐标相等) 和垂直 (x 坐标相等) 这两种特定的情况。但这两种情况并不能覆盖 .line 成员的所有情况，因此我们需要添加另一个分支来捕获其他所有线段类型。尽管在这里我们并不对 .rectangle 和 .circle 这两个成员的区别感兴趣，但相比使用一个 default 分支，我们更喜欢显式地把剩下的成员都列出来，因为这样可以让我们利用到完备性检查。

另外，编译器还会验证一个 switch 语句中各个模式的权重。如果一个模式之前的所有模式已经可以覆盖所有情况的话，编译器就能证明这个模式永远不会被匹配到，从而针对这个情况发出一个警告。

完备性检查的最大好处体现在如果你想让枚举和使用它的代码是同步演进的时候，也就是说，每次给枚举增加一个新的成员时，所有 switch 这个枚举代码都可以被同时更新。如果你可以访问程序的依赖项的源代码，并且程序和依赖项是一起编译的话，确实可以利用到完备性检查的各个好处。但当一个库是以二进制形式发布，并且使用这个库的程序必须做好编译之后会使用更新版本的准备，情况就会变得更复杂了。在这种情况下，即使分支已经覆盖了现有的所有情况，但还是需要始终包含一个 default 分支的。我们会在本章后面的固定和非固定枚举中再回到这个问题。

## 不可能产生非法的状态

有太多正面的理由来解释为什么要使用像是 Swift 这种静态类型语言。性能是其中之一：编译器对于程序中变量的类型知道的越多，通常就越能产生更快的代码。

另一个同样重要的理由是，类型系统可以指导开发人员应该如何使用 API。如果你把一个错误的类型传递给函数，编译器马上就会报错。我们可以将它称为**编译器驱动开发 - 不要把编译器看作必须打倒的敌人，而是把它作为一种工具**，通过使用类型信息，它几乎可以神奇般地引导你找到正确地解决方案：

- 精心选择一个函数的输入和输出类型的话，可以减少误用这个函数的机会，因为类型为函数的行为建立了一个“上限”。例如，如果你正在实现一个参数类型是非可选值对象的函数的话，那么在函数体中，则可以确保该参数对象永远不会是 nil。这个方式能工作到多好，完全取决于我们如何能精细地约束类型使之只接受有效的值。当要精确定义允许值的范围时，枚举就经常是一个完美的工具。
- 静态类型检查可以完全防止某些类别的错误；如果代码违反了类型系统设置的约束的话，这些代码就不会被编译成功，也永远不会再运行时才去处理这些违反约束的情况。
- 类型就像是永远不会失效的文档。在这点上它和注释是不一样的，随着代码的更新，人们有可能会忘记更新相应的注释，但因为类型是程序的一个组成部分，所以它永远都是最新的。

当然了，类型系统是不能表达程序的所有方面的。例如，Swift 本身没办法告诉你一个函数是否是纯函数（即函数是没有副作用的）或这个函数的性能特征。这就是我们还需要文档的原因，并且开发者在更新现有代码时，必须注意不要违反那些记录在文档中所需要遵守的事。但显而易见的是，你从编译器得到的帮助是和类型系统所提供的功能成正比的（我们应该注意到，对于一

个编程语言，当然有可能会过度使用类型系统。虽然通常它们很有帮助，但为编译器提供的信息越多，开发者需要做的事也越多，这有时会妨碍解决实际的任务。此外，对一个特定的用例，你调整其类型的粒度越细，你就需要编写更多的代码来转换类型之间的值。我们不认为 Swift 已经达到了这一点，但毕竟没有免费的午餐)。

以下是我们设计自定义类型时，为了从编译器获得最大程度帮助的建议：**使用类型使其无法表示非法状态**。在之前总和类型和乘积类型这一节中已经看到过，在为一个枚举增加一个成员后，这个类型就只会增加一个可能的值。除此之外你的粒度不会更细了，对于此目的来说枚举是非常有用的。

典型的例子就是 Optional，通过 none 成员以及添加一个泛型参数作为封装类型，就在不依靠哨岗值的情况下，已经可以精确地表达出缺少一个值这件事了。我们已经在可选值这一章中讨论过哨岗值的问题了。

让我们看一个 API 的例子，因为不符合上述的准则，所以它比想象中还要难用。在 Apple 的 iOS SDK 中，异步操作(比如执行一个网络请求)的常见模式是将一个完成时处理的方法(回调函数)传递给你调用的异步操作。当任务完成时，此方法会用任务的结果为参数来调用之前传入的回调函数。因为大多数异步操作都存在失败的可能，所以通常情况下，任务的结果要么是某个表示成功的值(例如，服务器的响应)，要么就是一个错误。在未来，我们也许可以看到更少的接受回调的方法，取而代之我们可以看到更多的 async 方法。不过现在它们还是很常见的。

看一下在 Apple 的 Core Location 框架中的地理解码 API。你把一个表示某个地址的字符串和一个回调函数传递给这个 API。这个 API 会去请求服务器返回匹配这个地址的所有地标对象(placemark object)。然后它要么用得到的地标对象列表，要么用一个错误来调用传入的回调函数：

```
class CLGeocoder {  
    func geocodeAddressString(_ addressString: String,  
        completionHandler: @escaping ([CLPlacemark]?, Error?) -> Void)  
    // ...  
}
```

观察一下回调函数的类型，([CLPlacemark]?, Error?) -> Void。它的两个参数都是可选值。这意味着此函数可以呈现给调用者四种可能的状态：(.some, .none), (.none, .some),

(`.some`, `.some`), 或 (`.none`, `.none`) (这是一个简化的视角; 因为实际上 `some` 的状态是有无限多个可能的值, 但这里我们只关心它们是不是非空的)。四个合法的状态所造成的问题在于, 在实践中只有前两个状态是有意义的。如果开发人员同时接收到一个地标的列表和一个错误的话, 他们应该做什么? 更糟糕的是, 如果两个值返回的都是 `nil` 的话, 又该怎么办? 因为类型不够精确, 所以在这里编译器无法帮助到你。

目前为止, 因为 Apple 可能都在小心翼翼地实现这个方法, 让其永远不会返回这些无效状态中的任何一个, 所以在实践中前面说的问题永远不会发生。但 API 的使用者是无法对此吃下定心丸的, 即使今天没问题, 但谁也不能保证在下一个版本的 SDK 中这种情况就不会发生。

如果把这两个可选值替换为一个 `Result<[CLPlacemark], Error>` 类型的话, 这个地理解码的 API 将会变得对开发者更友好:

```
extension CLGeocoder {  
    func geocodeAddressString(_ addressString: String,  
        completionHandler: @escaping (Result<[CLPlacemark], Error>) -> Void) {  
        // ...  
    }  
}
```

`Result` 类型表示操作要么成功要么失败, 但这两个状态不可能同时存在, 也不可能都不存在。通过使用一个无法表达无效状态的类型, API 变得更易使用, 并且因为编译器禁止了很多情况, 所以一系列潜在的错误也就自然不会发生了。由于许多 Apple 的 iOS API 都是用 Objective-C 实现的, 所以它们也就无法充分利用 Swift 的类型系统, 毕竟 Objective-C 的枚举是没有像关联值这种概念的东西的。但这并不意味着我们不能用 Swift 做得更好。

从 iOS 15 和 macOS 12 开始, `CLGeocoder` 也提供了返回 `[CLPlacemark]` 或抛出错误的 `async` API。这个新的 API 比我们上面重写的例子更加简洁, 因为它确保了方法始终只会返回一个值或者抛出一个错误, 而接受 `completionHandler` 的版本, 这个回调可能会不被调用, 调用一次, 或者甚至被多次调用。

当实现一个函数时，请仔细考虑参数和返回值的类型。你对输入和输出值的有效集合所做的约束越严格，那么无论是对实现函数的人，还是对调用这个 API 的用户来说，编译器给予的帮助就越多。

另外，迄今为止我们忽略了地理解码 API 中回调函数可能存在的另一个有趣的状态：如果返回的地标对象数组是空的话，那怎么办呢？文档上似乎说这种情况永远不会发生，也就是说，对于输入的字符串，如果服务器没有找到任何一个匹配的地标，那么会返回一个错误。但这里存在另一种可能的解释：一个空数组可以表示请求本身是成功的（没有网络错误之类的），但没有找到匹配的地标。仅仅通过查看回调函数的参数类型，我们是不能确定哪个解释是正确的。如果想在类型系统中实现第一种解释的话，我们就会需要一个能在编译期提供永不为空保证的数组类型。虽然标准库没有提供这样的类型，但我们可以自己实现一个。基本的思路是实现一个有两个属性的结构体：一个表示数组中第一个元素（称之为 head），另一个表示的是除了第一个元素之外剩余的元素（称之为 tail），其类型就是普通的数组（它可以为空）。

```
struct NonEmptyArray<Element> {  
    private var head: Element  
    private var tail: [Element]  
}
```

由于 head 不是可选值，因此不可能创建一个什么元素都不包含的 NonEmptyArray 值。在 NonEmptyArray 的完整实现中，它实现的协议应该和 Array 是相同的，其中最重要的协议就是 Collection。这会使得它用起来和普通的数组一样方便，甚至有时会更方便，因为我们可以重载一些 Collection API，例如，重载 first 和 last 来返回一个非可选值。如果你想继续研究这个类型，可以看一下由 Brandon Williams 和 Stephen Celis 两人实现的 [NonEmpty 库](#)。它是对于这种模式的一种实现，在 NonEmpty 的泛型参数上有 Collection 协议的约束（因此你也可以拥有一个不为空的字符串）。有关对于 Swift 集合类型协议的深入讨论，请参阅 [集合类型协议](#) 这一章。

## 使用枚举来实现 Model 状态

如何在我们的程序中实现状态，并且不让状态出现非法情况，是程序设计的另一个主要方面。在一个给定的时间点上，程序的 状态 包括所有变量的内容加上（隐式地）其当前的执行状态，即哪些线程正在运行以及它们正在执行哪条指令。一个状态需要“记住”很多事，像是一个程序所处的模式，正在显示的数据，当前正在处理的用户交互等。除了最简单的程序之外，所有程序都是有状态的：一个特定的指令被执行时，接下来会发生什么是取决于系统所处的当前状态

(HTTP 是一个无状态协议的例子，这意味着对于同一个客户端，服务端必须在处理当前请求时不考虑先前的请求。在多个请求之间，Web 开发者必须使用像是 cookies 之类的功能来记住状态。即使 HTTP 是无状态的，但一个用来处理 HTTP 请求的程序仍然是有状态的，因为它需要维护内部的状态)。

当程序运行时，它会改变状态以响应像是用户交互或从网络传入数据等这种外部事件。这可以是隐式地发生的，而开发者不用过多的考虑它 - 毕竟，状态改变是始终发生的。但随着程序变得越来越复杂，最好有意识地定义程序 (或其某个子程序) 可能存在的状态，以及状态之间的合法转换。一个系统可以存在的状态集合也称为其状态空间。

**尝试使你程序的状态空间尽可能的小。**状态空间越小，你作为开发者所做的工作就变得越简单 - 一个较小的状态空间减少了代码所需要处理的情况的数量。因为枚举的状态数量是有限的，所以非常适用于实现状态以及状态之间的转换。并且因为每个枚举状态，或者说每个成员，都带有自己的数据 (以关联值的形式)，所以很容易禁止表达非法状态的组合，正如我们在上一节中看到的那样。

我们应该注意到，理论上你程序的状态空间很可能是无限大的，特别是如果你的程序接收的用户输入形式是像文本或上传图像这种。这些数据类型的值自然是接近无限的。但就像之前我们只关心一个值是否为 nil 一样，这通常不是一个问题。多数系统的状态的基本部分都是有限的，并且数量通常很少；否则，我们就不能在代码中实现它们了。

让我们来看一个例子。假设我们正在实现一个聊天的程序。当用户打开一个聊天频道后，程序应该在从网络请求消息列表的期间显示一个 spinner。当网络请求完成时，UI 要么就变成显示接收到的消息列表，要么如果网络失败的话就显示一个错误。让我们先考虑一下如何在没有枚举的情况下，以传统的方式来实现程序的状态 (技术上说，因为我们会用可选值，所以仍然使用的是枚举，但你明白我们这里的意思)。我们可以用三个变量，其中一个为布尔值，我们把这个值设为 true 来表示当前正处于网络请求的过程中，另外两个都为可选值，分别用于消息列表和错误：

```
struct StateStruct {  
    var isLoading: Bool  
    var messages: [Message]?  
    var error: Error?  
}
```

```
// 设置初始状态。  
var structState = StateStruct(isLoading: true, messages: nil, error: nil)
```

当加载消息列表时，messages 和 error 这两个变量都该为 nil，然后当网络请求完成时，它们中的一个应该被赋值。在同一时刻这两个变量应该永远不可能都是非 nil 的，当它们中的任何一个不为 nil 时，isLoading 的值也不应该是 true。

回想一下在总和类型和乘积类型中，我们关于如何确定一个类型所能拥有居留元数量的讨论。StateStruct 结构体是一个拥有  $2 \times 2 \times 2 = 8$  个可能状态的乘积类型：布尔的 true 或 false 以及两个可选值中任意一个的 none 或 some 所组成的任意组合（再一次，因为和这里讨论的内容无关，所以我们忽略了 some 所拥有的无限多状态）。这其实就是一个问题，因为我们的程序只需要处理这八种状态中的三种：加载，显示一个消息列表或显示一个错误。如果我们正确实现了程序的话，另外五个状态都应该是不会发生的无效组合，但我们无法指望编译器能在我们创建了一个无效的状态时给出警报。

现在，让我们用一个自定义枚举来实现我们的状态，这个枚举有三个值，loading, loaded 和 failed：

```
enum StateEnum {  
    case loading  
    case loaded([Message])  
    case failed(Error)  
}
```

```
// 设置初始状态  
var enumState = StateEnum.loading
```

你会马上注意到，因为我们不必再关心和初始状态无关的属性，所以设置初始状态的代码变得更清晰了。此外，我们完全消除了转换到一个无效状态的可能性。因为每个状态都带有自己的关联数据，所以 loaded 和 failed 的关联值的类型也就不必是可选值了。因此，除非在代码中我们确实是有一个 Error 值，否则就不可能转换到 failed 状态（对于 loaded 状态，因为你总是可以给它赋值一个空数组，所以情况会有点不太清晰，但这不是你会不小心做的事）。当程序处

于某个特定状态时，我们可以确信该状态所需要的数据也已经是可用的了。我们的 StateEnum 枚举可以作为一个状态机 (state machine) 的基础。

枚举不是完整的有限状态机 (finite-state machines)，因为它缺乏指定非法状态转换的能力 - 例如，在我们这么简单的例子中，应该不可能在 loaded 和 failed 之间做转换。实际上，除非你拥有所有关联数据的有效值，否则让其无法实例化一个状态也几乎算是一个好的方案了。但在一个设计良好的程序中，你不太可能在代码中找到许多地方都存在对一个状态的所有关联数据都可用，但转换到该状态仍然是无效的操作。

每次需要访问一些依赖状态的数据 (例如，消息数组) 时，我们现在都 switch 枚举来提取关联值。因为 switch 语法是非常严格的，所以有时会让人觉得不方便。但这是一个重要的安全性方面的特性，因为它强迫我们总是处理每个可能的状态 - 至少会检查我们是否在 switch 语句中使用了 default 分支。

另外，你可能已经发觉了，我们开始使用的结构体和之后替换它的枚举都不是实现这个状态的唯一方式。事实上，StateStruct.isLoading 属性是多余的，因为在我们的设计中，isLoading 应该只在 messages 和 error 都为 nil 的时候才为 true。我们可以在不损失任何东西的前提下让 isLoading 是一个计算属性：

```
struct StateStruct2 {  
    var messages: [Message]?  
    var error: Error?  
  
    var isLoading: Bool {  
        get { return messages == nil && error == nil }  
        set {  
            messages = nil  
            error = nil  
        }  
    }  
}
```

这使得可能的状态的数量由八个减少为四个，从而只剩下一个无效状态（当 `messages` 和 `error` 都为非 `nil` 时） - 这并不完美，但要比我们一开始的那个结构体版本要好。通常很难注意到像这样的冗余属性，但这种一个类型的居留元数量和代数之间的联系是真正可以帮助到我们的。例如在这个例子中，我们发现自定义类型有  $2 \times 2 \times 2$  个居留元，但其中只有三个是有效的，很容易看出其中一个因数是多余的： $2 \times 2$  就足够容纳这三个有效状态了，所以肯定有删除一个因数的可能性。

这种具有两个互斥的可选值的模式，可能会让你想起我们在前一节中使用 `Result<[CLPlacemark], Error>` 替换 `([CLPlacemark]?, Error?)` 的例子。对我们现在这个例子使用相同方式的话就会变成 `Result<[Message], Error>`，但请注意这两种情况并不完全相同；聊天程序需要第三个状态 - “加载”，这种状态下，`messages` 和 `error` 都为 `nil`。把 `Result` 变成一个可选值就可以实现这一点（回想一下，把一个类型封装为一个可选值的话，总是会为这个类型增加一个居留元），所以用以下另一种方式来表示我们的状态：

```
/// nil 意味着状态为 "加载"。  
typealias State2 = Result<[Message], Error>?
```

这和我们所自定义的枚举是等价的，也就是说，两者在状态的数量和每个状态的有效负载上是一样的（`Result<[Message]?, Error>` 是另一个等价的版本）。但从语义上讲，这可以说是一个较差的解决方案，因为它不能立即清晰地表明出，当值为 `nil` 时就表示“加载”状态这件事。

我们的例子仅用枚举实现了程序中单个子系统的状态。但你可以进一步推广这种模式并将整个程序的状态用一个枚举来实现 - 通常会用许多嵌入的枚举和结构体来把这个枚举状态分解成各个子系统。思路就是拥有一个可以捕获整个程序的状态的变量。所有状态的改变都通过这个变量，然后你可以观察这个变量（例如，通过使用  `didSet`），以便在一个状态改变发生时，更新你程序的 UI。这个设计还可以轻松地将整个程序的状态写入磁盘，并在下次启动时将其读回，从根本上为你提供了免费的状态恢复功能。如果你想了解更多关于这个设计模式方面的内容，请查看由 Chris 和 Florian 及 Matt Gallagher 同一编写的 [App 架构](#) 一书。

虽然可以用枚举来实现你整个程序的状态，但使用枚举表示状态（enums-as-state）的好处是你不必全部投入其中就可以从中受益。可以从转换一个子系统（例如，app 里的某一屏）开始，并看它是如何工作的。然后通过把各个子系统的状态枚举分别封装到同一个枚举（这个枚举对于每个子系统都有一个成员）中的方式来逐步把这个模式扩展出去。

总而言之，枚举是实现状态的绝佳选择。它可以在很大程度上防止无效状态，并将子系统（或者甚至是整个程序）的整个状态都放在一个变量中，从而使状态转换更不容易出错。此外，switch语句的完备性允许编译器能在你添加了一个新的状态，或改变了现有状态的关联值时，指出需要更新的代码路径。

## 在枚举和结构体之间做选择

在本章前面一点的部分，我们讨论了枚举和结构体之间如何具有非常不同的特性：一个枚举值精确地表示所有成员中的一个（加上它的关联值），但一个结构体的值表示的是它所有属性的值。尽管存在这些差异，但我们也经常会遇到既可以用枚举又可以用结构体来解决的问题。

下面这个例子受到 [Matt Diephouse](#) 发表的博客中的内容的启发，我们会用枚举和结构体各实现一个用来分析事件的数据类型。以下是枚举的版本：

```
enum AnalyticsEvent {  
    case loginFailed(reason: LoginFailureReason)  
    case loginSucceeded  
    ... // 更多的枚举值。  
}
```

通过增加数个计算属性来扩展这个枚举，在这些计算属性中，switch枚举并返回用户所需的数据，即实际上应发送到服务器的字符串和字典：

```
extension AnalyticsEvent {  
    var name: String {  
        switch self {  
            case .loginSucceeded:  
                return "loginSucceeded"  
            case .loginFailed:  
                return "loginFailed"  
            // ... more cases.  
        }  
  
        var metadata: [String: String] {
```

```
switch self {  
    // ...  
}  
}  
}  
}
```

另一种选择是我们可以用结构体来实现相同的功能，将其名字和元数据 (metadata) 保存在两个属性中。我们提供一些静态方法 (分别对应于上面各个枚举值) 来为特定的事件创建实例：

```
struct AnalyticsEvent {  
    let name: String  
    let metadata: [String : String]  
  
    private init(name: String, metadata: [String: String] = [:]) {  
        self.name = name  
        self.metadata = metadata  
    }  
  
    static func loginFailed(reason: LoginFailureReason) -> AnalyticsEvent {  
        return AnalyticsEvent(  
            name: "loginFailed"  
            metadata: ["reason" : String(describing: reason)]  
        )  
    }  
    static let loginSucceeded = AnalyticsEvent(name: "loginSucceeded")  
    // ...  
}
```

由于我们把初始化方法声明为了私有，所以暴露出去的公有接口是和枚举版本相同的：枚举暴露了一些成员，像是 `.loginFailed(reason:)` 或 `.loginSucceeded` 这种，而结构体暴露的则是静态方法和属性。`name` 和 `metadata` 在两个版本中都可用的，只是在枚举中是计算属性而在结构体中是存储属性。

但是，每个版本的 AnalyticsEvent 类型都有其独特的特性，这些特性可以成为优点也可以成为缺点，具体取决于你的需求是什么：

- 如果我们让结构体的初始化方法的访问级别为 internal 或 public 的话，则可以在其他文件或者甚至其他模块中通过添加静态方法或属性来扩展这个结构体，从而添加新的分析事件到 API 中。枚举的版本是无法实现这一点的：你不能在其他地方添加新的成员到枚举中。
- 枚举可以更精确地实现数据类型；它只能表示预定义成员中的一个，但结构体因为这两个属性而可能表示无限多的值。如果你想对事件做额外的处理（例如，合并事件序列），则枚举的精确性和安全性会派上用场。
- 结构体可以有私有“成员”（也就是说，对所有使用者都不可见的静态方法或静态属性），而枚举中成员的可见性始终和枚举本身保持一致。
- 你可以对枚举使用 switch 语句，并利用语句的完备性来确保不会错过任何一个事件的类型。但由于这种严格性，所以向枚举添加一个新的事件类型就可能会破坏使用这个 API 用户的源代码，但你可以为新的事件类型往结构体中添加静态方法，而不用担心会影响其他代码。

## 枚举和协议之间的相似之处

乍看之下，枚举和协议似乎没有太多共同之处。但实际上这两者之间有一些有趣的相似之处。在总和类型和乘积类型这一节中，我们提到过枚举不是唯一可以表示“之一”关系的结构；协议也可用于此目的。在这一节中，我们会看一个这样的例子，并讨论这两种方式之间的差异。

从我们在本章早先用过例子开始，在一个画图程序中用枚举表示多个不同的形状的：

```
enum Shape {  
    case line(from: Point, to: Point)  
    case rectangle(origin: Point, width: Double, height: Double)  
    case circle(center: Point, radius: Double)  
}
```

一个形状可以是线段，矩形或圆形中的一种。我们在其扩展中添加一个渲染方法来把这些形状渲染到 Core Graphics 的上下文中。在这个方法的实现中必须 `switch self`，并在每个分支中执行适当的绘图命令：

```
extension Shape {  
    func render(into context: CGContext) {  
        switch self {  
            case let .line(from, to): // ...  
            case let .rectangle(origin, width, height): // ...  
            case let .circle(center, radius): // ...  
        }  
    }  
}
```

另一种方式就是我们可以定义一个名为 `Shape` 的协议，任何实现这个协议的类型都可以把自身渲染到一个 Core Graphics 上下文中：

```
protocol Shape {  
    func render(into context: CGContext)  
}
```

我们之前用成员表示的各个形状类型，现在都变成了实现这个 `Shape` 协议的具体类型。每个类型都实现了属于自己的 `render(into:)` 方法：

```
struct Line: Shape {  
    var from: Point  
    var to: Point  
  
    func render(into context: CGContext) { /* ... */}  
}  
  
struct Rectangle: Shape {  
    var origin: Point  
    var width: Double
```

```
var height: Double  
  
func render(into context: CGContext) { /* ... */}  
}  
  
// 忽略了 `Circle` 类型
```

虽然功能上是等价的，但考虑以下两件事就会觉得很有意思：枚举和协议这两种方式是如何组织代码的，以及如何为了新功能来扩展它们。基于枚举的实现是按方法来分组的：所有形状类型的基于 `CGContext` 的渲染代码都在 `render(into:)` 方法中的单个 `switch` 语句中。另一方面，基于协议的实现是按“成员”来分组的：每个具体的类型都实现自己的 `render(into:)` 方法，该方法中包含了每个形状特定的渲染代码。

这在扩展性方面具有重要的影响：在枚举的版本中，我们可以在之后的 `Shape` 扩展中轻松地添加新的渲染方法 - 例如，渲染成一个 SVG 文件，就是在不同模块中也可以如此。然而，除非我们可以控制含有枚举声明的源代码，否则我们就无法在枚举中添加新的形状。并且即使我们可以修改枚举的定义，添加一个新的成员这件事，是对所有在实现中 `switch` 这个枚举的方法的一种破坏源代码的修改。

另一方面，在协议的版本中我们可以轻松地添加新的形状：只需创建一个新的结构体，并让其实现 `Shape` 协议即可。但是，如果不修改现有的 `Shape` 协议，我们就无法添加新的渲染方法，因为我们不能在协议声明之外添加新的协议要求（我们是可以在协议的扩展中添加新的方法的，但正如会在协议这一章中看到的一样，扩展方法通常不适合于向协议添加新功能的这个需求，因为这些方法不是动态派发的）。

事实证明，在这种情况下，枚举和协议具有互补的优势和劣势。每个解决方案在一个维度上是可扩展的，而在另一个维度上就缺乏灵活性。如果 API 的声明和使用都发生在同一个模块中的话，枚举和协议之间这些扩展性的差异就不那么重要了。但是，如果你在实现一个库中的代码的话，那么你应该考虑哪个维度上的扩展性更重要：是添加新的成员重要，还是添加新的方法更重要。

如果你对跨模块边界的扩展性这一特定问题感兴趣的话，请查看我们同 Brandon Kase 一起录制的关于这个主题的Swift Talk 视频。在这些视频中，探讨了一种允许我们同时在这两个维度上获得扩展性的技术。

## 使用枚举实现递归数据结构

枚举非常适合用来实现递归数据结构，即“包含”自身的数据结构。想象一下树结构：一个树有多个分支，每个分支其实又是另一个分成多个子树的树，以此类推，直到到达树叶。许多常见的数据格式都是树结构，例如，HTML, XML, 和 JSON。

作为递归数据结构的一个例子，让我们实现一个 XML 的小的子集。对于成熟的实现，你可以参考 [Swim](#) 或者 [swift-html](#) 库。我们会创建一个 Node 枚举，它要么是一个文本节点，要么是一个元素，要么一个片段(也就是多个节点)。这种“要么...要么...”的选择关系，强烈暗示了一个总和类型(即枚举)是非常适合用来定义该数据结构的：

```
enum Node: Hashable {  
    case text(String)  
    indirect case element(  
        name: String,  
        attributes: [String: String] = [:],  
        children: Node = .fragment([]))  
    case fragment([Node])  
}
```

请注意 `indirect` 关键字，这是使代码能编译通过所必需的。`indirect` 告诉编译器把 `element` 成员表示为一个引用，从而使递归起作用。如果 `element` 没有成为一个引用，那么这个枚举就将会有无限大尺寸。我们会在下一节关于 `indirect` 的部分进行更多说明。

上面的定义就是我们所需要用来构建一棵简单的节点树的所有内容了，像是这样的 HTML 代码 `<h1>Hello <em>World</em></h1>`，可以被表示为：

```
let header: Node = .element(name: "h1", children: .fragment([  
    .text("Hello "),  
    .element(name: "em", children: .text("World"))  
]))
```

现在，我们添加一些简便方法，来让 `Node` 的构建变得更简单一些。首先我们可以让这个类型实现 `ExpressibleByArrayLiteral`，这样在构建片段 (`fragment`) 的时候会更容易：

```
extension Node: ExpressibleByArrayLiteral {  
    init(arrayLiteral elements: Node...) {  
        self = .fragment(elements)  
    }  
}
```

类似的，我们可以通过实现 ExpressibleByStringLiteral 来让创建 .text 节点更简单：

```
extension Node: ExpressibleByStringLiteral {  
    init(stringLiteral value: String) {  
        self = .text(value)  
    }  
}
```

最后，让我们通过添加一个扩展方法，来简化将节点包装到元素 (element) 的过程。wrapped 方法将会负责把自身包装到一个 element 节点中并返回它。

```
extension Node {  
    func wrapped(  
        in elementName: String,  
        attributes: [String: String] = [:]  
    ) -> Node {  
        .element(name: elementName, attributes: attributes, children: self)  
    }  
}
```

有了这三个扩展方法，现在我们能把上面的例子用一种短得多的方式写出来了：

```
let contents: Node = [  
    "Hello ",  
    ("World" as Node).wrapped(in: "em")  
]  
let headerAlt = contents.wrapped(in: "h1")
```

想要让这些代码更具有 Swift 的风格，我们可以使用result builder，来让语法看起来更像 SwiftUI。

在本章开头，我们提到过枚举也可以拥有 mutating 的方法。比如，我们可以写一个可变版本的 wrapped 方法，让它原地对 self 进行更改：

```
extension Node {  
    mutating func wrap(in elementName: String, attributes: [String: String] = [:]) {  
        self = .element(name: elementName, attributes: attributes, children: self)  
    }  
}  
  
var greeting: Node = "Hello"  
greeting.wrap(in: "strong")
```

和结构体一样，mutating 并不会改变值本身：它只是改变了变量所指向的是哪个值。

Swift 中的枚举还可以用来为更多的抽象数据结构建模：比方说，用它来构建一个链表、二叉树、或者甚至可持久化数据结构。当这样做时，最好听我一句劝：对于绝大多数使用场景，性能上想要超过像是数组、字典或者 Set 集合这些 Swift 的内建数据类型，是非常困难的。

## Indirect

为了理解为什么我们要将递归的 Node 枚举声明为 indirect，先回想一下枚举是值类型这件事。值类型是不能包含自身的，因为如果允许这样的话，在计算类型大小的时候，就会创建一个无限递归。编译器必须能够为每种类型确定一个固定且有限的尺寸。将需要递归的成员作为一个引用是可以解决这个问题的，因为引用类型在其中增加了一个间接层；并且编译器知道任何引用的存储大小（在一个 64 位的系统上）总是为 8 个字节。

注意，虽然 fragment 中也递归使用了 Node，但我们并没有在它前写 indirect。这是因为这个关联值是一个数组，而数组这个结构体具有固定尺寸（数组中元素使用的实际内存被存储在一个缓冲区中）。

indirect 语法只能用在枚举上。如果没有这个关键字，或者我们想要构建一个递归的结构体的话，我们就只能通过把递归值打包到一个 class 里，通过手动创建间接值来重现同样的行为。

对于 Node 枚举，如果我们不希望允许一个顶层的片段成员，我们还可以提供一种替代的定义方式。这种方式就不需要被标记为 `indirect` 了，因为它依赖的是数组提供的间接特性：

```
enum NodeAlt {  
    case text(String)  
    case element(name: String, attributes: [String: String], children: [Node])  
}
```

我们也可以把 `indirect` 加到枚举定义本身上去，比如 `indirect enum Node { ... }`。这是一个为所有带有关联值的成员开启间接特性的简便写法 (`indirect` 只会被作用在关联值上；对于那些枚举用来区别成员的标签位 (tag bits)，是不存在间接特性的)。如果 `indirect` 的成员拥有多个关联值，那么引用将会被放在一个把这些关联值合并起来的值上。

一个枚举类型的内存尺寸，是其中最大的成员的尺寸，再加上存储标签 (也就是成员的呈现方式) 所需要的空间。举个例子，下面这个枚举的尺寸是 17 字节，它包括了占用空间最大的成员 (16 字节) 以及一个存储标签的字节 (1 字节)：

```
enum TwoInts {  
    case nothing  
    case int(Int, Int)  
}  
MemoryLayout<TwoInts>.size // 17
```

如果我们吧 `int` 成员标记为 `indirect`，枚举尺寸会变成 8，而非 9。这是因为虽然引用的尺寸是 8 个字节，但是引用有一些没有使用的字节位可以分给标签位使用，标签位被内联存储在了引用中。

有的时候，当你在处理一个拥有很大尺寸成员的枚举时，你可能会需要把这个成员标记为 `indirect` 来减少枚举的尺寸。当然了，这么做的代价是引入了间接行为。

理论上说，编译器应该可以推断一个枚举是否应该被标记为 `indirect`。一些其他语言 (比如 Haskell) 确实这么做了。Swift 的设计者选择不这么做，是因为他们想要让开发者能明确地控制比如把一个大的成员标记为 `indirect` 这种行为。另外，实际上编译器并不能完全可靠地推断

出是否应该标记 `indirect`: 对一个泛型枚举来说, 它是否应该是 `indirect` 枚举, 有可能会取决于泛型参数到底是什么类型。

## 原始值 (Raw Value)

有时需要将枚举的每个成员同一个数字或其他某个类型的值相关联。C 或 Objective-C 中枚举的工作方式默认就是如此 - 实际上在底层它们就只是一些整数。Swift 的枚举不能与任意整数互换, 但我们可以选择性地在枚举的成员和所谓的 **原始值** 之间声明一个 1 对 1 的映射。这对于与 C API 进行相互操作, 或把一个枚举值编码成像是 JSON 这种数据格式来说都是非常有用的(我们会在编码和解码这一章中讨论 `Codable` 系统, 它可以使用原始值来为枚举合成实现 `Codable` 的代码)。

给枚举指定一个原始值需要在枚举名字后面加上原始值的类型并用冒号分隔开。然后我们使用赋值语法给每个成员赋值一个原始值。以下是一个原始值类型为 `Int` 的枚举的例子, 用这个枚举来表示 HTTP 状态:

```
enum HTTPStatus: Int {  
    case ok = 200  
    case created = 201  
    // ...  
    case movedPermanently = 301  
    // ...  
    case notFound = 404  
    // ...  
}
```

每个成员的原始值必须唯一。如果我们不为一个或多个成员提供原始值的话, 编译器会尝试选择合理的默认值。在这个例子中, 我们其实可以不用为 `created` 成员显式地分配一个原始值; 编译器会通过递增前一个成员的原始值来给 `created` 选择一个和现在相同的值 - 201。

## RawRepresentable 协议

一个实现 RawRepresentable 协议的类型会获得两个新的 API：一个 rawValue 属性和一个可失败的初始化方法 (init?(rawValue:))。这两个 API 都被声明在 RawRepresentable 协议中（编译器自动为具有原始值的枚举实现这个协议）：

```
/// 一个可以同相关原始值做转换的类型。
```

```
protocol RawRepresentable {  
    /// 原始值的类型, 例如 Int 或 String。  
    associatedtype RawValue  
  
    init?(rawValue: RawValue)  
    var rawValue: RawValue { get }  
}
```

因为对于每个 RawValue 类型的值，有可能会存在对于实现这个协议的类型来说无效的值，所以初始化方法是可失败的。例如，只有一些整数是有效的 HTTP 状态码；对于其他所有的输入，HTTPStatus.init?(rawValue:) 必须返回 nil：

```
HTTPStatus(rawValue: 404) // Optional(HTTPStatus.notFound)  
HTTPStatus(rawValue: 1000) // nil  
HTTPStatus.created.rawValue // 201
```

## 手动实现 RawRepresentable

上面那种把原始值赋值给一个枚举的语法，只作用于有限的一组类型：类型可以是 String, Character, 任意整数或浮点类型。这覆盖了一些用例，但并不意味着类型只能是这些。因为上面的语法只是一个实现 RawRepresentable 的语法糖，所以如果你需要更多的灵活性的话，总是可以选择手动实现这个协议。

以下的例子中定义了一个枚举，它表示在一个逻辑坐标系统中的一些点，每个点的 x 和 y 坐标都在 -1 (左/下) 和 1 (右/上) 之间。这个坐标系统有点类似于 Apple 的 Core Animation 框架中 CALayer 的 anchorPoint 属性。我们使用一对整数来作为原始值的类型，并且由于自动合成 RawRepresentable 的语法糖并不支持元组类型，所以我们需手动实现 RawRepresentable：

```
enum AnchorPoint {
```

```
case center
case topLeft
case topRight
case bottomLeft
case bottomRight
}

extension AnchorPoint: RawRepresentable {
    typealias RawValue = (x: Int, y: Int)

    var rawValue: (x: Int, y: Int) {
        switch self {
            case .center: return (0, 0)
            case .topLeft: return (-1, 1)
            case .topRight: return (1, 1)
            case .bottomLeft: return (-1, -1)
            case .bottomRight: return (1, -1)
        }
    }

    init?(rawValue: (x: Int, y: Int)) {
        switch rawValue {
            case (0, 0): self = .center
            case (-1, 1): self = .topLeft
            case (1, 1): self = .topRight
            case (-1, -1): self = .bottomLeft
            case (1, -1): self = .bottomRight
            default: return nil
        }
    }
}
```

这需要多写点代码，但并不难。这些代码正是编译器在自动合成 RawRepresentable 时为我们生成的代码。毫无疑问，对于使用这个枚举的用户来说，在这两种情况下行为都是相同的：

```
AnchorPoint.topLeft.rawValue // (x: -1, y: 1)  
AnchorPoint(rawValue: (x: 0, y: 0)) // Optional(AnchorPoint.center)  
AnchorPoint(rawValue: (x: 2, y: 1)) // nil
```

在手动实现 RawRepresentable 时要注意的一件事是用重复的原始值来赋值。自动合成的语法要求原始值是唯一的 - 重复的话会引发一个编译错误。但在手动实现中，编译器不会阻止你从多个成员中返回相同的原始值。可能有充分的理由来使用重复的原始值（例如，当多个成员互相是同义时，也可能为了向后兼容），但它应该是例外情况。Switch 一个枚举时，总是与成员而不是原始值来进行匹配的。换句话说，即使两个成员具有相同的原始值，你也不能用一个来匹配另外一个。

## 让结构体和类来实现 RawRepresentable

另外，RawRepresentable 不仅限于枚举；你同样可以让一个结构体或类来实现这个协议。对于为了保护类型安全而引入的简单的封装类型而言，实现 RawRepresentable 协议通常是一个不错的选择。例如，一个程序可能在内部使用字符串来表示用户的 ID。不要直接使用 String 类型，而最好定义一个新的 UserID 类型来防止不小心和其他字符串变量混淆了。还有可能会需要用一个字符串来初始化一个 UserID 实例，以及提取它的字符串值；而 RawRepresentable 非常适合这些需求：

```
struct UserID: RawRepresentable {  
    var rawValue: String  
}
```

这里的 rawValue 属性满足了实现 RawRepresentable 协议的两个要求之一，但是第二个要求（初始化方法）的实现在哪里呢？它由 Swift 结构体自动生成的成员初始化方法这个特性所提供。编译器足够聪明，让其把一个不会失败的 init(rawValue:) 方法的实现视作协议所需的那个可失败的初始化方法的实现。这有一个很好的副作用，在用字符串创建一个 UserID 实例时，我们不必处理可选值了。如果我们想对输入的字符串进行验证的话（也许不是所有的字符串都是有效的用户 ID），就必须为 init?(rawValue:) 提供我们自己的实现。

## 原始值的内部表示

除了添加 RawRepresentable API 和自动 Codable 的合成规则有所区别之外，实际上具有原始值的枚举与所有其他枚举并没有什么不同。特别是，具有原始值的枚举保持了其完整的类型标识。和 C 语言中你可以将任意整数值赋值给一个枚举类型的变量所不同，一个具有 Int 原始值的 Swift 枚举是不会“成为”一个整数的。一个枚举类型的实例所能拥有的值只能是其成员中的一个。获取原始值的唯一方法就是通过调用 rawValue 和 init?(rawValue:) 这两个 API。

拥有原始值也不会改变枚举在内存中的表示方式。我们可以定义一个具有 String 类型原始值的枚举，并通过查看其类型尺寸来验证这一点：

```
enum MenuItem: String {  
    case undo = "Undo"  
    case cut = "Cut"  
    case copy = "Copy"  
    case paste = "Paste"  
}  
  
MemoryLayout<MenuItem>.size // 1
```

MenuItem 类型的大小只有 1 个字节。这就告诉了我们一个 MenuItem 实例没有在内部存储原始值 - 如果它这样做的话，其大小肯定至少有 16 个字节 (在 64 位平台上 String 的大小)。编译器生成的 rawValue 实现就像一个计算属性一样，类似于我们在上面展示的 AnchorPoint 的实现。

## 列举枚举值

在之前总和类型和乘积类型这一节中，我们已经讨论过什么是一个类型的居留元：一个类型的实例可以拥有的所有可能的值的集合。把这些值作为一个集合进行操作的这个需求通常是很常用的，例如，迭代或计数它们。Caseltable 协议通过添加一个静态属性 allCases 来实现这个功能 (也就是说，不是在实例上，而是在类型上调用此属性)：

```
/// 一个提供其所有值集合的类型
```

```
protocol Caselterable {  
    associatedtype AllCases: Collection  
    where AllCases.Element == Self  
  
    static var allCases: AllCases { get }  
}
```

对于没有关联值的枚举，编译器会自动生成实现 Caselterable 的代码；我们所要做的就只是在声明的时候把协议加上就可以了。让我们在前一节中的 MenuItem 类型上实现一下这个功能：

```
enum MenuItem: String, Caselterable {  
    case undo = "Undo"  
    case cut = "Cut"  
    case copy = "Copy"  
    case paste = "Paste"  
}
```

因为 allCases 属性的类型是 Collection，所以它具有你从数组和其他集合类型中，所知的所有常用属性和功能。在下面的示例中，我们使用 allCases 来得到所有菜单项的数量，并把它们转换为适合在用户界面中显示的字符串（为了简单起见，我们直接使用原始值作为菜单项的标题；在一个真实的 app 中，会把原始值作为一个键，用在被存储的本地化标题的查找表上）：

```
MenuItem.allCases  
// [MenuItem.undo, MenuItem.cut, MenuItem.copy, MenuItem.paste]  
MenuItem.allCases.count // 4  
MenuItem.allCases.map { $0.rawValue } // ["Undo", "Cut", "Copy", "Paste"]
```

和其他像是 Equatable 和 Hashable 这一类，编译器会自动合成实现的协议类似，Caselterable 自动合成的代码的最大好处，并不是代码本身的难度有多高（手动实现该协议是很简单的），而是编译器生成的代码始终会是最新的 - 手动实现的话，每次添加或删除成员时都必须手动更新你的实现，这是很容易忘记的。

Caselterable 协议没有规定 allCases 返回的集合中的值的特定顺序，但 Caselterable 的文档中则保证集合中的值的顺序是和它们在声明时的顺序所一致的。

## 手动实现 Caselterable

对于没有关联值的普通枚举来说，`Caselterable` 是特别有用的，并且自动编译器合成也只支持这一种类型。这是合理的，因为向一个枚举添加关联值可能会使这个枚举的居留元数量变成无限。但只要我们能实现一个方法来生成一个所有居留元的集合，那么我们总是可以手动实现这个协议的。事实上，这个协议并不只限于枚举。虽然 `Caselterable` 和 `allCases` 这两个名字都暗示了此功能主要用于枚举（没有其他类型有成员这个概念），但编译器对一个实现了此协议的结构体或类也是没有意见的。

以下的代码中，在最简单的类型之一 `Bool` 上手动实现 `Caselterable`：

```
extension Bool: Caselterable {  
    public static var allCases: [Bool] {  
        return [false, true]  
    }  
}
```

```
Bool.allCases // [false, true]
```

一些整数类型同样也是好的选择。请注意，`allCases` 的返回类型不必一定是数组 - 它可以是任何一个实现了 `Collection` 的类型。当一个范围可以用更少的内存来表示相同的集合时，生成一个包含所有可能的整数的数组就显得非常浪费了：

```
extension UInt8: Caselterable {  
    public static var allCases: ClosedRange<UInt8> {  
        return .min ... .max  
    }  
}
```

```
UInt8.allCases.count // 256
```

```
UInt8.allCases.prefix(3) + UInt8.allCases.suffix(3) // [0, 1, 2, 253, 254, 255]
```

同理，如果你想为一个居留元数量很多的类型实现 `Caselterable`，或者生成一个类型的值是非常昂贵的话，请考虑返回一个 `LazyCollection`，以便不用提前执行一些不必要的操作。我们会在集合类型协议这一章中讨论这个类型。

请注意，这两个例子都忽略了一个通用原则，即不要让你所不拥有的类型去实现你不拥有的协议。在你在生产环境的代码中破坏这条原则之前，请考虑与此相关的各种权衡。有关详细信息，请参阅协议这一章。

## 固定和非固定枚举

在本章中我们反复强调了，枚举的最佳优点之一就是在 `switch` 它的时候所展现出来的完备性。只有在编译期间，编译器知道了一个枚举所可能拥有的全部成员的情况下，才能执行完备性检查。当枚举的声明和 `switch` 它的语句都是在同一个模块时，这点是很容易做到的。如果枚举的声明是在另一个库，但这个库是和我们的代码一起编译的话（每次添加或删除一个成员时，都会重新编译枚举的声明和我们自己的代码，这允许编译器重新检查所有相关的 `switch` 语句），这点也是容易做到的。

然而在某些情况下，我们用到的枚举是在一个以二进制形式链接到我们程序的库中。标准库就是最明显的例子：尽管标准库的源代码已经开源了，但我们通常使用的都是由 Swift 发行版或操作系统所附带的二进制文件。Swift 所附带的一些其他库也是如此，包括 Foundation 和 Dispatch。最后，Apple 和其他公司都希望以二进制形式来发布 Swift 的库。

就拿标准库中的类型来做例子，假设在代码中我们要处理一个 `DecodingError` 实例。它是一个枚举，在 Swift 5.5 中，它有四个成员来表示不同的错误条件：

```
let error: DecodingError = ...
// 在编译时做完完备性检查，而可能不在运行时做检查。
switch error {
    case .typeMismatch: ...
    case .valueNotFound: ...
    case .keyNotFound: ...
    case .dataCorrupted: ...
}
```

随着 Codable 系统的扩展，未来的 Swift 版本中很有可能会增加另外的成员到这个枚举。但如果我们构建的 app 包含了上述的代码，并且把 app 发给了用户，那么那些用户最终可能会在一个较新的操作系统上运行发送给他们的可执行文件，并且系统附带的是包含了一个新的 DecodingError 成员的较新的 Swift 版本。在这种情况下，我们的程序会崩溃，因为它遇到了一个无法处理的错误条件。

可能会在未来添加新成员的枚举，称之为**非固定**。为了让程序能够防范这种非固定枚举的修改，在一个模块中 switch 另一个模块中的非固定枚举的话，必须始终包含一个 default 子句，以便能够处理将来会发生的这种情况。在 Swift 5.5 中，如果你忽略了 default 分支的话，编译器只会发出警告（而不是错误），但这只是为了让你能方便地迁移现有代码所做的权宜之计。在未来的版本中警告会变成错误。

如果你让编译器帮你修复这个警告的话，你会注意到它帮你在 default 分支之前加了一个 @unknown 属性：

```
switch error {  
    ...  
    case .dataCorrupted: ...  
    @unknown default:  
        // Handle unknown cases.  
    ...  
}
```

在运行时，@unknown default 的行为就像一个普通的 default 子句，但它也是对于编译器的一个信号，用来告诉编译器这个 default 分支只是为了处理在编译时无法知道的成员的情况。如果 default 分支匹配了一个在编译时就知道的成员的话，我们还是会得到一个相应的警告。这意味着针对未来一个新的库的接口，当重新编译程序时我们还是可以从完备性检查中获益。如果自上次更新后有新的成员被添加到库的 API 中的话，就会得到一个警告，让我们更新所有相关的 switch 语句来显式地处理新的成员。@unknown default 为你提供了两全其美的方案：编译时的完备性检查和运行时的安全性。

固定枚举和非固定枚举的区别，只有在模块以库进化模式 (library evolution mode) 编译时才会显现，而这个模式默认是关闭的，只有在添加了 -enable-library-evolution 编译器标志，它才会被开启。这些开启了进化模式的库也被叫做**弹性库** (resilient libraries)。从设计上来说，

它们要在维持稳定的 ABI 的同时，允许库的作者进行一些 API 的修改。向枚举中添加一个成员就是这样的修改。标准库以及所有的 Apple SDK 中的框架都是弹性库。在弹性库（或者在两次发布之间的库）中的枚举默认都是非固定的。Swift 中存在一个 @frozen 标志，它可以用来把某个指定的枚举声明为固定枚举。一旦使用了这个标志，就意味着库的开发者承诺永远不会再向这个枚举中添加新的成员，否则这种改变就会破坏二进制兼容。

在标准库中，固定枚举的例子包括有 Optional 和 Result；如果它们不是固定的话，switch 它们时就总是会需要一个 default 子句，这是一个很大的烦恼。

在非弹性库（包括了像是通过 Swift Package Manager 引入的开源依赖这样的，你直接在你的代码里进行编译和链接的那些模块）中的枚举都会被认为是固定枚举，所以它们在 switch 时不需要 @unknown default。因为这些模块的二进制接口会永远和源码同步，所以这么做是没问题的。

## 提示和窍门

我们会用一些提示和窍门来结束本章。

**尽量避免使用嵌套 switch 语句。**你可以使用元组一次性匹配多个值。例如，假设你要根据两个布尔类型的值来设置一个变量的话，一个接一个的匹配就会需要一个嵌入的 switch 语句，这很快会让代码变得难看：

```
let isImportant: Bool = ...
```

```
let isUrgent: Bool = ...
```

```
let priority: Int
```

```
switch isImportant {
```

```
case true:
```

```
    switch isUrgent {
```

```
        case true: priority = 3
```

```
        case false: priority = 2
```

```
    }
```

```
case false:
```

```
    switch isUrgent {
```

```
case true: priority = 1  
case false: priority = 0  
}  
}
```

把两个 Bool 值放到一个元组中去匹配则会让代码更短，并提高可读性：

```
let priority2: Int  
switch (isImportant, isUrgent) {  
case (true, true): priority2 = 3  
case (true, false): priority2 = 2  
case (false, true): priority2 = 1  
case (false, false): priority2 = 0  
}
```

**利用明确初始化检查 (definite initialization check)**。再看一眼上面的例子。这里有一个模式，我们在 switch 之前声明了一个 let 常量，但未初始化它，然后在 switch 的每个分支中再初始化它，在这里我们利用了编译器的明确初始化检查。编译器会在首次使用一个变量之前去验证这个变量是否已完全初始化了 - 如果我们在一条或多条代码路径中忘记初始化的话，编译器会产生一个错误。这种风格要比单纯地用 var 声明 priority 并对其赋值两次（一次在声明时，一次在 switch 分支中）的方案更安全。

就像 if 一样，switch 是一个语句，不是一个表达式，尽管我们常常希望它是后者。Swift 没有方便的语法把 switch 一个枚举的结果赋值给一个变量。在 switch 语句之前声明一个常量，然后在每个分支中设置它，是我们能做到最好的程度了。

**避免用 none 或 some 来命名成员**。这两个名字都是对命名成员这件事来说有吸引力的；但我们建议你避免使用它们，因为在模式匹配的上下文中，它们可能与 Optional 的成员发生冲突。以下是一个有问题的枚举的定义：

```
enum Selection {  
case none  
case some  
case all
```

```
}
```

假设我们有一个 Selection? 类型 (即一个可选值) 的变量，希望将它与一个模式进行匹配：

```
var optionalSelection: Selection? = ...  
  
if case .some = optionalSelection {  
    // 哪个 some 被匹配到了?  
}
```

这里是匹配了 Selection.some 还是匹配了 Optional.some (即任何非 nil 的值)？答案是后者，但这非常容易出错，特别是考虑到 Swift 喜欢隐式地将非可选值提升为可选值。(编译器确实会对 if case .none = optionalSelection 发出警告，来指出可能存在歧义，但是对 if case .some =... 的形式并不会警告。)

对那些用保留的关键字来命名的成员使用反引号 (backtick)。如果你使用某些关键字来作为成员名字的话 (例如， default)，类型检查器会因为无法解析代码而产生错误。你可以用反引号把名字括起来使用它：

```
enum Strategy {  
    case custom  
    case `default` // 需要反引号。  
}
```

这样做的好处是，在类型检查器可以消除歧义的地方，都不需要反引号了。以下代码是完全有效的：

```
let strategy = Strategy.default
```

可以像工厂方法一样使用成员。如果一个成员拥有关联值的话，这个枚举值就单独地形成了一个签名名为 (AssocValue) -> Enum 的函数。以下枚举用来表示在两个颜色空间之一 (RGB 或灰阶) 的一个颜色：

```
enum OpaqueColor {
```

```
case rgb(red: Float, green: Float, blue: Float)
case gray(intensity: Float)
}
```

OpaqueColor.rgb 是一个有着三个 Float 类型的参数和返回类型是 OpaqueColor 的函数：

```
OpaqueColor.rgb // (Float, Float, Float) -> OpaqueColor
```

我们也可以将这些函数传递给例如 map 这样的高阶函数。以下的代码中，我们把成员作为一个工厂方法传递给 map，然后创建从黑到白的渐变灰度颜色：

```
let gradient = stride(from: 0.0, through: 1.0, by: 0.25).map(OpaqueColor.gray)
/*
[OpaqueColor.gray(intensity: 0.0), OpaqueColor.gray(intensity: 0.25),
OpaqueColor.gray(intensity: 0.5), OpaqueColor.gray(intensity: 0.75),
OpaqueColor.gray(intensity: 1.0)]
*/
```

只要满足同样的签名，枚举的成员甚至可以满足协议要求。这里，协议要求一个静态方法，它直接可以被映射到相同名字的枚举成员上去，所以这个枚举类型不需要额外工作就可以满足协议了：

```
protocol ColorProtocol {
    static func rgb(red: Float, green: Float, blue: Float) -> Self
}

// No code required.

extension OpaqueColor: ColorProtocol {}
```

**不要使用关联值来模拟存储属性。请改用结构体。** 枚举不能拥有存储属性。这听起来像是一个重大的限制，但事实并非如此。请你思考一下，实际上添加一个类型为 T 的存储属性与为每个成员添加相同类型的关联值是没有什么不同的。例如，让我们为上面那个 OpaqueColor 类型添加一个透明通道，在每个成员中都添加一个对应的关联值：

```
enum AlphaColor {
```

```
case rgba(red: Float, green: Float, blue: Float, alpha: Float)
case gray(intensity: Float, alpha: Float)
}
```

这是可以工作的，但从一个 AlphaColor 实例中提取 alpha 组件不是很方便 - 即使我们知道每个 AlphaColor 实例都有一个 alpha 组件，但还是必须 switch 实例并在每个分支中提取这个值。虽然我们可以将这个逻辑封装到一个计算属性中，但更好的解决方案可能是一开始就避免这个问题 - 让我们把之前的 OpaqueColor 枚举封装到一个结构体中，并把 alpha 作为结构体的一个存储属性：

```
struct Color {
    var color: OpaqueColor
    var alpha: Float
}
```

这是一个通用模式：当你发现一个枚举中每个成员的关联值都有一部分是相同的，请考虑把这个枚举封装到一个结构体中，并把公共部分提取出来。这会改变结果类型的样子，但不会改变其基本性质。这和在数学等式中提取公因子是一样的： $a \times b + a \times c = a \times (b + c)$ 。这种与代数的对应关系也解释了为什么把总和类型和乘积类型总称为“代数数据类型”。

**不要过度使用关联值组件。**在本章中我们大量使用了多个元组式组件来表示关联值，像是 OpaqueColor.rgb(red:green:blue:) 这种。这对简短的例子来说很方便，但在生产环境的代码中，通常来说为每个成员实现一个自定义的结构体是更好的选择。比较以下两个在之前模式匹配这一节中用过的 Shape 类型的版本。首先是元组风格的版本：

```
enum Shape {
    case line(from: Point, to: Point)
    case rectangle(origin: Point, width: Double, height: Double)
    case circle(center: Point, radius: Double)
}
```

其次是为每个成员都实现一个自定义结构体的版本：

```
struct Line {
```

```
var from: Point  
var to: Point  
}  
  
struct Rectangle {  
    var origin: Point  
    var width: Double  
    var height: Double  
}  
  
struct Circle {  
    var center: Point  
    var radius: Double  
}  
  
enum Shape2 {  
    case line(Line)  
    case rectangle(Rectangle)  
    case circle(Circle)  
}
```

后一个例子一开始需要写的代码会多一点，但它会让枚举的声明和 switch 语句中模式的代码变得清晰。此外，这些结构体还具有自己的特征：我们可以扩展它们，并让它们实现各种协议。

**把空枚举作为命名空间。**除了由模块形成的隐式命名空间之外，Swift 没有内置的命名空间。但我们可以用枚举来“模拟”命名空间。由于类型定义是可以嵌套的，因此外部类型可以充当其包含的所有声明的命名空间。正如我们在[可选值](#)这一章看到的那样，像是 Never 这样的空枚举是不能被实例化的。这使得空枚举是定义自定义命名空间的最佳选择。标准库也是这么做的，例如 Unicode 命名空间：

```
/// 一个含有 Unicode 实用方法的命名空间。
```

```
public enum Unicode {  
    public struct Scalar {
```

```
internal var _value: UInt32  
// ...  
}  
// ...  
}
```

不幸的是，空枚举并不是对于缺乏适当的命名空间的完美解决方法：协议不能被嵌入到其他声明中，这就是为什么相关的标准库协议被命名为 `UnicodeCodec` 而不是 `Unicode.Codec` 的原因。

## 回顾

枚举是总和类型。当实现自定义类型时，枚举是一种重要的工具，它可以避免基于纯乘积类型而设计后出现的特性，即不需要的那些状态的组合数会爆炸。仔细想一下，一个类型的居留元有助于我们做出更好的设计决策。如果你需要的类型是针对你尝试解决的问题而量身定制的（比如要实现你的程序中的状态），那么一个枚举，或者一个嵌套枚举和结构体的组合通常是最佳选择。

与更熟悉的记录类型相比，枚举会有不同的设计模式。你的目标应该是让非法的程序状态无法在你的类型中表示出来。这减少了必须准备要处理的状态集合的代码数量，并使编译器在你编写新代码时能够指导你。不论何时，都请尽可能利用编译器来进行完备性检查。

# 字符串

8

所有的现代编程语言都支持 Unicode 字符串，但是通常这只意味着原生的字符串类型可以存储 Unicode 数据 - 它没有保证像是获取字符串长度这类简单操作会返回“恰当”的结果。实际上，大部分语言，以及用这些语言所写的对字符串进行操作的代码，都在某种程度上展现出了对 Unicode 内在复杂度的抗拒。这可能会造成一些令人不快 bug。

Swift 在字符串实现上做出了英勇的努力，它力求尽可能做到 Unicode 正确。Swift 中的 `String` 是 `Character` 值的集合，而 `Character` 是人类在阅读文字时所理解的单个字符，这与该字符由多少个 Unicode 标量 (Unicode scalars) 组成无关。这样一来，像是 `count` 或者 `prefix(5)` 在内的所有标准的 `Collection` 操作都会在用户所理解的字符这个层级上工作。

这对于正确性来说非常重要，但是也有所代价。大部分代价来源于开发者对这套规则的不熟悉。如果你在其他语言中用整数作为索引操作过字符串，那么 Swift 的设计一开始可能看起来非常笨重。你可能会想，为什么我不能用 `str[999]` 来获取字符串的第一千个字符？为什么 `str[idx+1]` 不能访问到下一个字符？为什么我不能在 "a"..."z" 这样的 `Character` 的值所构成的范围内进行循环？同样，还有一些性能上的影响：`String` 不支持随机访问，也就是说，跳到字符串中某个随机的字符不是一个  $O(1)$  操作。当字符拥有可变宽度时，字符串并不知道第  $n$  个字符到底存储在哪儿，它必须查看这个字符前面的所有字符，才能最终确定对象字符的存储位置，所以这不可能是一个  $O(1)$  操作。

在本章中，我们会深入讨论字符串的架构，我们也会涉及如何发挥 Swift 字符串的功能，以及保持良好性能的话题。不过，我们需要先从了解一些 Unicode 术语的总览开始。

## Unicode, 而非固定宽度

事情原本很简单。ASCII 字符串就是由 0 到 127 之间的整数组成的序列。如果你把这种整数放到一个 8 比特的字节里，你甚至还能省出一个比特。由于每个字符宽度都是固定的，所以 ASCII 字符串可以被随机存取。

但是对于非英语的文字，或者受众不是美国人的时候，ASCII 编码就够了。其他国家和语言需要不一样的字符（就连同样说英语的英国人都需要一个表示英镑的 £ 符号），其中绝大多数需要的字符用七个比特是放不下的。ISO/IEC 8859 使用了额外的第八个比特，并且在 ASCII 范围外又定义了 16 种不同的编码。比如第 1 部分 (ISO/IEC 8859-1，又叫 Latin-1)，涵盖多种西欧语言；以及第 5 部分，涵盖那些使用西里尔 (俄语) 字母的语言。

但是这样依然很受限。如果你想按照 ISO/IEC 8859 来用土耳其语书写关于古希腊的内容，那你就不能怎么走运了。因为你只能在第 7 部分 (Latin/Greek) 或者第 9 部分 (Turkish) 中选一种。另外，八个比特对于许多语言的编码来说依然是不够的。比如第 6 部分 (Latin/Arabic) 没有包括书写乌尔都语或者波斯语这样的阿拉伯字母语言所需要的字符。同时，在从 ASCII 的下半区替换了少量字符后，我们才能用八比特去编码基于拉丁字母但同时又有大量变音符组合的越南语。而其他东亚语言则完全不能被放入八个比特中。

当固定宽度的编码空间被用完后，你有两种选择：选择增加宽度或者切换到可变长的编码。最初的时候，Unicode 被定义成两个字节固定宽度的格式，这种格式现在被称为 UCS-2。不过这已经是现实问题出现之前的决定了，而且大家都承认其实两个字节还是不够用，四个字节的话在大多数情况下又太低效。所以今天的 Unicode 是一个可变长格式。它的可变长特性有两种不同的意义：

- 一个 Unicode 字符，也叫做**扩展字位簇 (extended grapheme cluster)**，由一个或多个 Unicode 标量 (Unicode scalar) 组成。
- 一个 Unicode 标量可以被编码成一个或多个**编码单元 (code units)**。

为了理解为什么设计成这样，我们先要搞清楚刚才提到的这些名词的含义。

Unicode 中最基础的原件叫做**编码点 (code point)**：它是一个位于 Unicode 编码空间 (从 0 到 0x10FFFF，也就是十进制的 1,114,111) 中的整数。Unicode 中的每个字符或其它语系单位 (译注：例如接下来会看到的颜文字) 都有一个唯一的编码点。2021 年 9 月发布的 Unicode 14 标准，在整个编码空间提供的 110 万个数值里，只使用了其中大约 14.5 万个。因此，还有大量的空间可以用于诸如存储颜文字 (Emoji) 这样的东西。通常，编码点都会写成带有 U+ 前缀的十六进制数。例如，欧元符号可以写成 code point U+20AC (或者十进制数 8364)。

Unicode 标量和刚才提到的编码点，在绝大多数情况下，是同一个东西。或者说，除了编码点中 0xD800 - 0xDFFF 之外的值，都可以叫做 Unicode 标量。而 0xD800 - 0xDFFF 这 2048 个值则叫做**代理编码点 (surrogate code points)**，它们在 UTF-16 编码中用于表示那些值大于 65535 的字符。在 Swift 里，Unicode 标量用形如 \u{xxxx} 这样的字符串表示，其中 xxxx 就是标量对应的十六进制数。因此，刚才提到的欧元符号就可以表示成 "\u{20AC}"，或者 "€"。和这些字符的值对应的 Swift 类型是 `Unicode.Scalar`，它是一个 struct，里面用 `UInt32` 包装了对应的标量值。

同样的 Unicode 数据 (例如：一个 Unicode 标量序列) 可以用多种不同的编码方式进行编码，其中最普遍使用的是 8 比特 (UTF-8) 和 16 比特 (UTF-16)。编码方式中使用的最小实体叫做编码单元，也就是说 UTF-8 编码单元的宽度是 8 比特，而 UTF-16 编码单元的宽度是 16 比特。因此，UTF-8 提供的一个额外的好处就是为使用 8 比特的 ASCII 编码提供了向后兼容，正是这个特性，才让 UTF-8 接过了 ASCII 大旗，成为了现如今 Web 和文件格式中最为流行的编码方式。要注意的是，编码单元和编码点 (或者说 Unicode 标量) 并不相同，一个 Unicode 标量通常都会编码成多个编码单元。由于可能存在 100 万以上个潜在的编码点，UTF-8 会使用 1 至 4 个编码单元 (也就是 1 至 4 个字节) 来编码单个 Unicode 标量，类似的，UTF-16 则会使用 1 至 2 个编码单元 (也就是 2 或 4 个字节)。在 Swift 里，UTF-8 和 UTF-16 使用的编码单元的值分别用 UInt8 和 UInt16 表示 (它们还有两个别名，分别是 Unicode.UTF8.CodeUnit 和 Unicode.UTF16.CodeUnit)。

如果要用单个编码单元来对应一个 Unicode 标量，你就需要一个 21 位的编码系统 (通常它会被向上“取整”到 32 位，也就是 UTF-32)。这个值在 Swift 里用 Unicode.Scalar 表示。但即便如此，你还是无法得到一个定宽的编码方案。提及到“字符”这个概念，Unicode 仍就是一个可变宽度的格式。因为用户在屏幕上看到的“单个字符”，可能是由多个 Unicode 标量组合起来的。对于这种用户感知到的“单个字符”，Unicode 中有一个术语，叫做 **(扩展) 字位簇**，对应的英文叫做 **(extended) grapheme cluster**。

因此，Unicode 标量形成字位簇的规则，决定了文本的分段方式。比如说，当按下键盘上的退格键时，你期望的是文本编辑器删除掉一个字位簇。这个字位簇表示的单个“字符”有可能是由多个 Unicode 标量组成的，而每个标量在表示文本的内存中，又可能使用不同数量的编码单元。在 Swift 里，字位簇是通过 Character 表示的。无论一个用户感知到的单个字符由多少 Unicode 标量组合而成，Character 都可以正确地把它们当作单个字符处理。

下面的图表显示了字符串 “AB •□” 的不同表示方式。按照你看待它的方式不同，这个字符串会由四个 Character，五个 Unicode 标量，或者是十二个 UTF-8 编码单元组成：

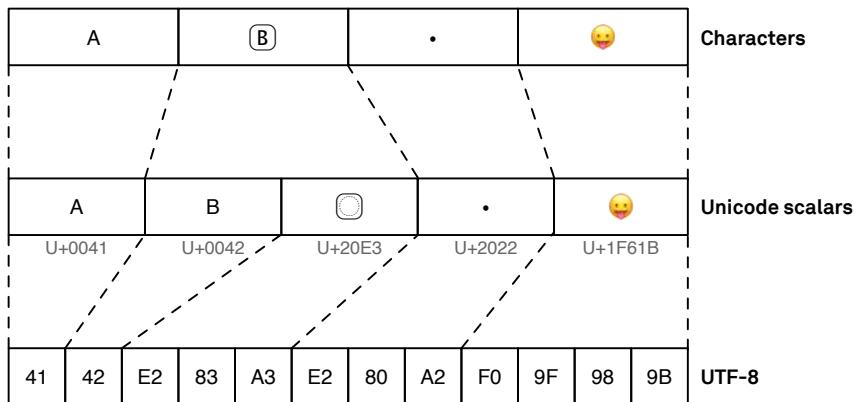


Figure 8.1: 同一字符串的不同表示

在下一节，我们会看到更多的例子，以及 Swift 处理由此引出的复杂度的方式。

## 字位簇和标准等价

### 合并标记

一种快速考察 String 是如何处理 Unicode 数据的方法是研究 “é” 的两种不同写法。Unicode 将 U+00E9 (带尖音符的小写拉丁字母 e) 定义成一个单一值。不过你也可以用一个普通的字母 “e” 后面跟一个 U+0301 (组合尖音符) 来表达它。这两种写法都显示为 é，而且用户也不会去关心这里的两个 “é” 是由哪种方式生成的，对他们来说，两个都显示为 “résumé”的字符串彼此相等，且含有六个字符，是非常合理的预期。Unicode 规范将此称作 **标准等价** (canonically equivalent)。

而这正是你将看到的 Swift 的运作方式：

```
let single = "Pok\u{00E9}mon" // Pokémon
let double = "Poke\u{0301}mon" // Pokémon
```

它们的显示一模一样：

```
(single, double) // ("Pokémon", "Pokémon")
```

并且两者有着相等的字符数：

```
single.count // 7  
double.count // 7
```

对它们进行比较，结果也是相等的：

```
single == double // true
```

只有当你深入观察它们的底层表现形式的时候，才能发现其中的不同：

```
single.unicodeScalars.count // 7  
double.unicodeScalars.count // 8
```

将上述字符串的表现和 Foundation 中的 NSString 对比就会发现：NSString 版本的两个字符串并不相等，并且 length 属性也给出了不同的结果（麻烦的是，许多 Objective-C 程序员还可能会用此属性计算显示在屏幕上的字符数）：

```
let nssingle = single as NSString  
nssingle.length // 7  
let nsdouble = double as NSString  
nsdouble.length // 8  
nssingle == nsdouble // false
```

这里 == 比较的，是两个 NSObject 对象：

```
extension NSObject: Equatable {  
    static func ==(lhs: NSObject, rhs: NSObject) -> Bool {  
        return lhs isEqual(rhs)  
    }  
}
```

就 `NSString` 而言，这会在 UTF-16 编码单元的层面上，按字面量做一次比较，而不会将不同字符组合起来的等价性纳入考虑。其他语言的大部分字符串 API 也都是这么做的。如果你真要按照标准等价的方式进行比较两个 `NSString`，就得使用 `NSString.compare(_:)` 方法。

当然，只比较编码单元也有一个很大的好处：它更快！在 Swift 里，可以通过比较字符串的 `utf8` 视图达到同样的效果：

```
single.utf8.elementsEqual(double.utf8) // false
```

Unicode 支持用多种表现形式表示同一个字符，到底为什么要这么做呢？因为，正是这些预先已经组合好的字符，才使得开放区间的 Unicode 编码点可以和已经拥有 “é” 和 “ñ” 这类字符的 Latin-1 兼容。这使得两者之间的转换快速而简单，尽管处理它们还是挺痛苦的。

并且，即便抛弃这些预先组合好的字符也不会解决 Unicode 字符有多种表现形式的问题。因为字符的组合并不只有成对的情况；你可以把一个以上的变音符号组合在一起。比如约鲁巴语有一个字符 ो，它可以用三种不同的形式来书写：通过组合 ो 和一个点；通过组合 ो 和一个尖音符；或者是通过组合 ो 和一个点与一个尖音符。而对于最后这种形式，两个变音符号的顺序甚至可以调换！所以下面这些全是相等的：

```
let chars: [Character] = [
    "\u{1ECD}\u{300}", // ꝑ
    "\u{F2}\u{323}", // Ꝓ
    "\u{6F}\u{323}\u{300}", // ꝓ
    "\u{6F}\u{300}\u{323}" // Ꝕ
]
let allEqual = chars.dropFirst().allSatisfy { $0 == chars.first } // true
```

实际上，某些变音符号可以被无限地添加。例如，下面这个著名的网红字符就很好地诠释了这一点：

```
let zalgo = "soon"
```

zalgo.count //4

```
zalgo.utf16.count // 36
```

在上面，`zalgo.count` 可以正确地返回 4，而 `zalgo.utf8.count` 则返回 68。如果你的代码不能正确处理这些网红字符，那还有什么用呢？

就算你要处理的所有字符串都是纯 ASCII 的，Unicode 的字位分隔规则还是会产生影响。例如：`CR+LF` 字符表示回车 (carriage return) 和换行 (line feed) 的组合，但它在 Windows 系统上通常被当作单个换行字符来使用，它是单个的字位簇：

```
// CR+LF 是单个字符
```

```
let crlf = "\r\n"
```

```
crlf.count // 1
```

## 颜文字

在很多其他语言中，含有颜文字的字符串也令人有些惊讶。很多颜文字的 Unicode 标量都无法通过单个 UTF-16 编码单元来表示，比如在 Java 或者 C# 里，会认为 "😂" 是两个“字符”长。Swift 则能正确处理这种情况：

```
let oneEmoji = "😂" // U+1F602
```

```
oneEmoji.count // 1
```

注意这里重要的是，字符串是如何呈现在程序中的，而不是它是如何存储在内存中的。在内部，Swift 使用了 UTF-8 作为编码方式，但是这都是实现细节。公有的 API 是基于字位簇的。

有些颜文字还可以由多个 Unicode 标量组合而成。例如：一个颜文字的国旗是由两个代表 ISO 国家码的区域表示字母 (regional indicator symbols) 所组成的。Swift 也能将它们正确地识别为一个 Character：

```
let flags = "🇧🇷 🇦🇺"
```

```
flags.count // 2
```

要观察组成字符串的 Unicode 标量，我们可以使用字符串的 `unicodeScalars` 视图。这里，我们将标量值格式化为编码点常用的十六进制格式：

```
flags.unicodeScalars.map {  
    "U+\($0.value, radix: 16, uppercase: true))"  
}  
// ["U+1F1E7", "U+1F1F7", "U+1F1F3", "U+1F1FF"]
```

把五种肤色修饰符 (比如 ，或者其他四种肤色修饰符之一) 和一个像是  的基础角色组合起来，就可以得到类似  这样的带有肤色的角色。再一次，Swift 能正确对其处理：

```
let skinTone = "👨‍🦰" // 🤷‍♂ +   
skinTone.count // 1
```

而那些表达家庭或情侣的颜文字，例如  和 ，则对 Unicode 标准提出了进一步的挑战。对于这种表达人群的颜文字，无论是性别还是人数都存在着数不清的组合，为其中每一种组合都单独定义一个编码点非常容易出问题。如果再把这些组合考虑上肤色的维度，让每种情况都有对应的编码点简直就成了一件不可能的事情。对此，Unicode 的解决方案是把这种复杂的颜文字表示成一个简单颜文字的序列，序列中的颜文字则通过一个标量值为 U+200D 的不可见零宽连接字符 (zero-width joiner, ZWJ) 连接。因此，家庭颜文字  是由男人  + ZWJ + 女人  + ZWJ + 女孩  + ZWJ + 男孩  构成的。ZWJ 的存在，是对操作系统的提示，表明如果可能的话，把 ZWJ 连接的字符当成一个字形符号 (glyph) 处理。

你可以通过下面的代码来确认 Unicode 的这种工作方式：

```
let family1 = "👨‍🦰👩‍🦰"  
let family2 = "👨‍🦰\u{200D}👩‍🦰\u{200D}👧‍🦰\u{200D}👦‍🦰"  
family1 == family2 // true
```

再一次，Swift 非常聪明，它能把这样的序列识别为单个 Character：

```
family1.count // 1  
family2.count // 1
```

代表职业的颜文字也是 ZWJ 序列。比如，一个女性消防员 🚒 是女人 👩 + ZWJ + 消防水车 🚒 的组合。而男性医护工作者 🚑 则是男人 👨 + ZWJ + 医疗之神阿斯克勒庇俄斯的权杖 🏊 的序列。

将这些序列渲染为单个字形符号，是操作系统的任务。在 2019 年的 Apple 平台上，操作系统所包括的字形符号是 Unicode 标准所列出的一般交换所推荐支持的 (recommended for general interchange, RGI) 的序列的子集。换句话说，这个列表中的颜文字“可以被认为是在多个平台被广泛支持的”。当一个语法上有效的序列，系统中没有对应可用的字形符号时，系统的字符渲染系统将会进行回退，将其中每个部分渲染为单独的字形。这样一来，用户视角的字符数和 Swift 所看到的单个字位簇就可能“在另一个方向上”发生不匹配了；到目前为止的例子都是编程语言认为字符数较实际要多，而现在我们就会看到相反的情况。比如，带有肤色的家庭颜文字序列现在并不在 RGI 子集里。尽管操作系统会将它渲染为多个字形，但 Swift 依然将它算作单个的 Character，因为 Unicode 的文本分段规则并没有将渲染的问题考虑进去：

```
// 在 2022 年，带有肤色的家庭颜文字在大多数平台上会被渲染为多个字形符号
```

```
let family3 = "👩\u{200D}👨\u{200D}骺\u{200D}骺\u{200D}"
```

// 但是 Swift 依然认为它是单个字符

```
family3.count // 1
```

不论一个字符串 API 被如何精心设计，文本问题实在是太过复杂，永远都会有没有注意到的边界情况存在。

Swift 使用的是系统 ICU 库中的字位切割算法。这样一来，只要你的用户升级了它们的系统，你的程序就将自动适配新的 Unicode 规则。不过，这也意味着你不能保证用户所看到的和你在开发中所看到东西一定是完全一致的。例如，在 Linux 上部署 Swift 服务端代码的时候，你的程序就可能有不同的表现，因为 Linux 发行版使用的 ICU 库的版本可能和你的开发机并不相同。

在本节中我们讨论的例子里，对于语言没有完全考虑 Unicode 的复杂性的情况，我们只用了字符串长度是否正确作为窗口来进行研究。随便想一想，要是一门编程语言没有按照字位簇来处理字符串的话，对那些包含字符序列的字符串，进行像是翻转字符串这样操作的话，就会带来奇怪的结果。这并不是一个新问题，随着颜文字爆炸性地流行，就算你的用户主要都使用英文，这种不严谨的文本处理所造成的诸多问题还是会迅速浮上台面。另外，错误的幅度也日益增加，

在以前，一个变音符号所带来的错误可能只是误差一个字符，但是现代颜文字通常会带来十个或更多的“字符”。比如，一个四人家庭颜文字在 UTF-16 中有 11 个编码单元长，而在 UTF-8 中这个数字则达到了 25：

```
family1.count // 1  
family1.utf16.count // 11  
family1.utf8.count // 25
```

并不是说其他语言就完全没有 Unicode 正确的 API，大部分其实都有。比如，`NSString` 有一个 `enumerateSubstrings` 方法，能被用来以字位簇的方式枚举字符串。但是默认行为十分重要，Swift 认为默认情况下行为正确具有更高的优先级。如果你想要下降到一个更低层级的抽象中，`String` 也提供了直接操作 Unicode 标量和编码单元的字符串视图。我们会在下面对它们进行更多讨论。

## 字符串和集合

我们已经提到过，`String` 是 `Character` 值的集合。在 Swift 存在的头三年里，`String` 在满足和不满足 `Collection` 协议之间来回摇摆了几次。不添加 `Collection` 的观点认为，如果支持了 `Collection`，开发者会认为所有一般化的集合处理算法在处理字符串时也是绝对安全和 Unicode 正确的，而这在某些边界情况下并非事实。

举个简单的例子，在将两个集合连接的时候，你可能会假设所得到的集合的长度是两个用来连接的集合长度之和。但是对于字符串来说，如果第一个集合的末尾和第二个集合的开头能够形成一个字位簇的话，它们就不再相等：

```
let flagLetterC = "🇨"  
  
let flagLetterN = "🇳"  
  
let flag = flagLetterC + flagLetterN // 🇨🇳  
  
flag.count // 1  
  
flag.count == flagLetterC.count + flagLetterN.count // false
```

考虑到这一点，在 Swift 2 和 Swift 3 中 `String` 本身并非 `Collection`。另外，由字符组成的集合也被移动到了 `characters` 属性里，它和 `unicodeScalars`, `utf8` 以及 `utf16` 等其他集合视图类

似，是一种字符串的表现形式。选取某一个特定的字符串视图可以提醒你进入了“集合处理”的模式，你需要自行考虑这对即将运行的算法会有什么影响。

在实践中，这个改动带来的非常稀有的边界情况下的正确性的提升，相比起在易用性上的损失和学习难度的增加，实在是得不偿失。所以在 Swift 4 里，String 又成为了 Collection。

## 双向索引，而非随机访问

不过，String 并不是一个可以随机访问的集合，我们从本章里见过的例子中已经清楚地理解了这个设计的原因。就算知道给定字符串中第  $n$  个字符的位置，也并不会对计算这个字符之前有多少个 Unicode 标量有任何帮助。所以，String 只实现了 BidirectionalCollection。你可以从字符串的头或者尾开始，向后或者向前移动，代码会察看毗邻字符的组合，跳过正确的字节数。不管怎样，你每次只能迭代一个字符。

当你在书写一些字符串处理的代码时，需要将这个性能影响时刻牢记在心。那些需要随机访问才能维持其性能保证的算法对于 Unicode 字符串来说并不是一个好的选择。假设我们要扩展 String，来生成一个包含字符串所有前缀子字符串的数组。我们可以先生成从 0 开始到字符串长度的范围，然后对这个范围进行映射，来为每个长度值创建前缀字符串：

```
extension String {  
    var allPrefixes1: [Substring] {  
        return (0...count).map(prefix)  
    }  
}  
  
let hello = "Hello"  
hello.allPrefixes1 // ["", "H", "He", "Hel", "Hell", "Hello"]
```

这段代码看上去简单，但是它非常低效。首先，它会遍历一次字符串，来计算其长度，这没什么大问题。但是，之后  $n + 1$  次对 prefix 的调用中，每一次都是一个  $O(n)$  操作，这是因为 prefix 总是要从头开始工作，然后在字符串上经过所需要的字符个数。在一个线性复杂度的处理中运行另一个线性复杂度的操作，意味着算法复杂度将会是  $O(n^2)$ 。随着字符串长度的增长，这个算法所花费的时间将以平方的方式增加。

如果可能的话，一个高效的字符串算法应该只对字符串进行一次遍历，而且它应该操作字符串的索引，用索引来表示感兴趣的子字符串。这里是相同算法的另一个版本：

```
extension String {  
    var allPrefixes2: [Substring] {  
        return [""] + indices.map { index in self[...index] }  
    }  
}
```

```
hello.allPrefixes2 // ["", "H", "He", "Hel", "Hell", "Hello"]
```

上面的代码依然需要迭代一次字符串，以获取索引的集合 `indices`。不过，一旦这个过程完成，`map` 中的下标操作就是  $O(1)$  复杂度的。这使得整个算法的复杂度得以保持在  $O(n)$ 。

## 范围可替换，而非可变

`String` 还满足 `RangeReplaceableCollection` 协议。下面的例子中，展示了如何首先找到字符串索引中一个恰当的范围，然后通过调用 `replaceSubrange` 来完成字符串替换。用于替换的字符串可以有不同的长度，或者甚至可以是一个空字符串 (这时相当于调用了 `removeSubrange`)：

```
var greeting = "Hello, world!"  
if let comma = greeting.firstIndex(of: ",") {  
    greeting[..    greeting.replaceSubrange(comma..., with: " again.")  
}  
greeting // Hello again.
```

和之前一样，要注意用于替换的字符串有可能与原字符串相邻的字符形成新的字位簇。

`MutableCollection` 是一个集合的经典特性，然而字符串并没有实现这个协议。除了 `get` 方法以外，`MutableCollection` 协议还为集合添加了对单个元素进行 `set` 的下标方法。这并不是说字符串是不可变的，我们刚才已经看到过，字符串拥有一系列可变方法。但是你无法做到通过下标操作对一个字符进行替换。究其原因，又回到可变长度的字符上。大部分人直觉上会认为，就像在 `Array` 中那样，单个元素的下标更新会在常数时间内完成。但是由于字符串中的字符可

能是可变长度，改变其中一个元素的宽度将意味着要把后面元素在内存中的位置上下移动。不  
止如此，在被插入的索引位置之后的所有索引值也会由于内存未知的改动而失效，这同样并不  
直观。由于这些原因，就算你想要更改的元素只有一个，你也必须使用 `replaceSubrange`。

## 字符串索引

大部分编程语言使用整数值对字符串进行下标操作，比如 `str[5]` 将会返回 `str` 中的第六个“字  
符”（这里的“字符”的概念由所操作的编程语言进行定义）。Swift 不允许这么做。为什么？答案  
可能现在你已经很耳熟了：因为整数的下标访问无法在常数时间内完成（对于 Collection 协议  
来说这也是一个直观要求），而且查找第 `n` 个 `Character` 的操作也必须要对它之前的所有字节进  
行检查。

`String.Index` 是 `String` 和它的视图所使用的索引类型，它本质上是一个存储了从字符串开头的  
字节偏移量的不透明值。如果你想计算第 `n` 个字符所对应的索引，你依然从字符串的开头或结  
尾开始，并花费  $O(n)$  的时间。但是一旦你拥有了有效的索引，就可以通过索引下标以  $O(1)$  的  
时间对字符串进行访问了。至关重要的是，通过一个已有索引来寻找下一个索引也是很快的，  
因为你可以从这个已有索引的字节偏移量开始进行查找，而不需要从头开始。正是由于这个原  
因，按顺序（前向或者后向）对字符串中的字符进行迭代是一个高效操作。

操作字符串索引的 API 与你在遇到其他任何集合时使用的索引操作是一样的，它们都基于  
Collection 协议。我们之所以容易忽略索引操作的等效性，是因为我们最经常使用的数组的索  
引就是整数类型，于是我们往往通过简单的算数，而非正式的索引操作 API，来对数组索引进  
行操作。`index(after:)` 方法将返回下一个字符的索引：

```
let s = "abcdef"  
let second = s.index(after: s.startIndex)  
s[second] // b
```

你可以通过 `index(_:offsetBy:)` 方法来一次性地自动对多个字符进行迭代：

```
// 步进 4 个字符  
let sixth = s.index(second, offsetBy: 4)  
s[sixth] // f
```

如果存在超过字符串末尾的风险，你可以加上 `limitedBy:` 参数。如果这个方法在达到目标索引之前就先触发了限制条件的话，它将返回 `nil`：

```
let safelidx = s.index(s.startIndex, offsetBy: 400, limitedBy: s.endIndex)  
safelidx // nil
```

毫无疑问，这比简单的整数索引需要更多的代码，但是再一次，Swift 就是这样设计的。如果 Swift 允许使用整数下标索引来访问字符串，会大大增加意外地写出性能相当糟糕的代码的可能性（比如，在一个循环中使用了整数下标）。

确实，对习惯于处理固定长度字符的人来说，起初操作 Swift 字符串看上去颇具挑战性。不通过整数索引你要怎么浏览字符呢？确实，很多看起来很简单的任务，比如说要提取字符串的前四个字符，实现看起来都会有些奇怪：

```
s[..
```

不过令人欣慰的是，我们可以通过 `Collection` 的接口来访问字符串，也就是说你能按照需求使用很多有用的技术。许多操作 `Array` 的函数一样可以操作 `String`。使用 `prefix` 方法，同样的事情就清楚多了：

```
s.prefix(4) // abcd
```

（注意，两个表达式返回的都是 `Substring`；你可以通过将其传递给 `String.init` 将它转换回 `String`。我们会在下一节里再谈到子字符串的话题。）

再来看个略显复杂的例子，我们可以不借助任何字符串下标操作，就从一个表示日期的字符串中提取出表示月份的部分：

```
let date = "2019-09-01"  
date.split(separator: "-")[1] // 09  
date.dropFirst(5).prefix(2) // 09
```

为了找到某个特定的字符，你可以用 `firstIndex(of:)` 方法：

```
var hello = "Hello!"
```

```
if let idx = hello.firstIndex(of: "!) {
    hello.insert(contentsOf: ", world", at: idx)
}
hello // Hello, world!
```

`insert(contentsOf:)` 方法将会把另一个具有相同元素类型 (对于字符串来说就是 `Character`) 的集合插入到给定索引之前。这个集合并不需要是另一个 `String`, 你也可以很容易地将一个字符组成的数组插入到字符串中。

虽然使用不透明索引来操作字符串已经是既成事实了, Swift 团体也意识到有时候人们确实只是想做一些“简单”的事情。因此, 他们进行了提议, 为包括字符串在内的所有集合类型添加一个整数偏移量的下标语法。核心团队最终决定把这个提案打回修改, 以纳入审查期间收集到的反馈意见, 但是这个修改一直没有进行, 也许是因为有其他事情更加紧迫。

## 字符串解析

当然, 还有一些字符串操作的任务是无法通过 Collection API 完成的: 解析 CSV 文件就是一个很好的例子。我们不能简单地用逗号分隔每一行的内容, 因为逗号还可能作为 CSV 中的值, 用引号包围起来。为了解决这个问题, 我们可以一个字符一个字符地遍历整个字符串, 并记录一些跟踪状态。本质上来说, 这就是编写一个简单的解析器:

```
func parse(csv: String) -> [[String]] {
    var result: [[String]] = []
    var currentField = ""
    var inQuotes = false
    for c in csv {
        switch (c, inQuotes) {
            // ...
        }
    }
    return result
}
```

首先，我们创建了一个元素是字符串数组类型的数组 `result`，这个数组中的每个元素，对应 CSV 文件中的一行内容。而参数 `csv` 则包含了 CSV 文件的多行内容。接下来，`currentField` 作为迭代字符串过程中，收集每个字段所有字符的缓冲区。最后，布尔变量 `inQuotes` 用于跟踪当前我们是否在遍历字段值时遇到了引号，它是这个简单的解析器中唯一的一个状态。

完成后，我们就可以根据遍历结果填充 `switch` 语句中的各种情况了：

- `("","", false)` - 引号外的逗号，表示当前字段的结束。
- `("\\n", false)` - 引号外的换行符号，表示结束当前行。
- `("\"\"", _)` - 读到引号，用于切换 `inQuotes` 状态。
- `default` - 除了上述情况之外的其他情况，我们把遍历到的字符添加到 `currentField` 就好了。

```
func parse(csv: String) -> [[String]] {  
    // ...  
    for c in csv {  
        switch (c, inQuotes) {  
            case ("","", false):  
                result[result.endIndex-1].append(currentField)  
                currentField.removeAll()  
            case ("\n", false):  
                result[result.endIndex-1].append(currentField)  
                currentField.removeAll()  
                result.append([])  
            case ("\\"", _):  
                inQuotes = !inQuotes  
            default:  
                currentField.append(c)  
        }  
    }  
    result[result.endIndex-1].append(currentField)  
    return result
```

```
}
```

(这里，我们创建了一个临时的 tuple 在 case 语句中同时匹配两个值。你应该记得在枚举这章讲到的这种技术。)

for 循环结束后，在返回结果之前，我们还要把 currentField 中的内容添加到 result 末尾，因为 CSV 内容的最后一行可能并不包含换行符。

完成后，可以用下面的例子试一下：

```
let csv = #"""
  "Values in quotes","can contain , characters"
  "Values without quotes work as well:",42
"""

parse(csv: csv)
/*
[[ "Values in quotes", "can contain , characters" ],
 [ "Values without quotes work as well:", "42" ]]
*/
```

上面例子中的字符串字面量使用了 扩展分隔符语法 (extended delimiters syntax)，也就是把字符串用 # 包围起来。这样就可以在字符串中直接使用引号而不用转义了。

写这样一个小型解析器可以显著提高你的字符串处理技巧。要完成这种任务，通过 Collection API 或者正则表达式都很困难，甚至是不可能的。但使用 String 提供的各种 API，读和写都会变得更简单。

刚刚实现的这个 CSV 解析器尽管还不完整，但它已经已然颇有用处。它的结构很精简，因为我们无需跟踪很多内部状态，仅仅用了一个布尔变量。通过一点额外的工作，我们还可以忽略空行、忽略引号周围的空格，并支持在字段的引号中通过转义的方式继续使用引号。为此，我们可以使用一个枚举来精确区分和解析这些内容需要的状态，而无需为每一种情况都使用一个单独的布尔变量。

但是，解析器中使用的状态越多，就越容易在实现中出现错误。因此，这种通过单次循环实现解析的方法只适用于规模较小的任务。如果需要跟踪更多的状态，我们只能改变这种在单一循环中跟踪状态的做法，把解析器的实现打散成多个函数。

## 子字符串

和所有集合类型一样，String有一个特定的SubSequence类型，叫做Substring。Substring和ArraySlice很相似：它是一个以原始字符串内容为基础，用不同起始和结束位置标记的视图。子字符串和原字符串共享文本存储，这带来的巨大的好处，就是让对字符串切片成为了高效操作。在下面的例子中，创建firstWord并不会导致昂贵的复制操作或者内存申请：

```
let sentence = "The quick brown fox jumped over the lazy dog."
let firstSpace = sentence.firstIndex(of: " ") ?? sentence.endIndex
let firstWord = sentence[..
```

在你对一个(可能会很长的)字符串进行迭代并提取它的各个部分的循环中，切片的高效特性就非常重要了。这类任务可能包括在文本中寻找某个单词出现的所有位置，或者就像我们上面做的那样，解析一个CSV文件等。在这里，字符串分割是一个很有用的操作。Collection定义了一个split方法，它会返回一个子序列的数组(也就是[Substring])。最常用的一种形式是：

```
extension Collection where Element: Equatable {
    public func split(separator: Element, maxSplits: Int = Int.max,
                      omittingEmptySubsequences: Bool = true) -> [SubSequence]
}
```

你可以这样来使用：

```
let poem = """
Over the wintry
forest, winds howl in rage
with no leaves to blow.

"""

let lines = poem.split(separator: "\n")
```

```
// ["Over the wintry", "forest, winds howl in rage", "with no leaves to blow."]
type(of:lines) // Array<Substring>
```

这个函数和 String 从 NSString 继承来的 components(separatedBy:) 很类似，不过还多加了一个决定是否要丢弃空值的选项。再一次，整个过程中没有发生对输入字符串的复制。另外 split 还有一种形式可以接受闭包作为参数，所以除了单纯的字符比较以外，它还能做更多事情。这里有一个简单的按词折行算法的例子，其中闭包里捕获了当前行中的字符数：

```
extension String {
    func wrapped(after maxLength: Int = 70) -> String {
        var lineLength = 0
        let lines = self.split(omittingEmptySubsequences: false) {
            character in
            if character.isWhitespace && lineLength >= maxLength {
                lineLength = 0
                return true
            } else {
                lineLength += 1
                return false
            }
        }
        return lines.joined(separator: "\n")
    }
}

sentence.wrapped(after: 15)
/*
The quick brown
fox jumped over
the lazy dog.
*/
```

又或者，考虑写一个接受含有多个分隔符的序列作为参数的版本：

```
extension Collection where Element: Equatable {  
    func split<S: Sequence>(separators: S) -> [SubSequence]  
        where Element == S.Element  
    {  
        return split { separators.contains($0) }  
    }  
}
```

现在，你就可以写出下列语句了：

```
"Hello, world!".split(separators: ", ") // ["Hello", "world"]
```

## StringProtocol

Substring 和 String 的接口几乎完全一样。这是通过一个叫做 StringProtocol 的通用协议来达到的，String 和 Substring 都遵守这个协议。因为几乎所有的字符串 API 都被定义在 StringProtocol 上，对于 Substring，你完全可以假装将它看作就是一个 String，并完成各项操作。不过，在某些时候，你还是需要将子字符串转回 String 实例；和所有的切片一样，子字符串的设计意图是用于短期存储，以避免在操作过程中发生昂贵的复制。当这个操作结束，你想将结果保存起来，或是传递给下一个子系统，这时你应该通过初始化方法从 Substring 创建一个新的 String，如下例所示：

```
func lastWord(in input: String) -> String? {  
    // 处理输入，操作子字符串  
    let words = input.split(separators: [" ", "\n"])  
    guard let lastWord = words.last else { return nil }  
    // 转换为字符串并返回  
    return String(lastWord)  
}  
  
lastWord(in: "one, two, three, four, five") // Optional("five")
```

不鼓励长期存储子字符串的根本原因在于，子字符串会一直持有整个原始字符串。如果有一个巨大的字符串，它的一个只表示单个字符的子字符串将会在内存中持有整个字符串。即使当原

字符串的生命周期本应该结束时，只要子字符串还存在，这部分内存就无法释放。长期存储子字符串实际上会造成内存泄漏，由于原字符串还必须被持有在内存中，但是它们却不能再被访问。

通过在一个操作内部使用子字符串，而只在结束时创建新字符串，我们将复制操作推迟到最后一刻，这可以保证由这些复制操作所带来的开销是实际需要的。在上面的例子中，我们将这个（可能很长）的字符串分割为子字符串，但是付出的开销只是在最后复制了一个短的子字符串（虽然算法本身不够高效，但我们现在先忽略这块；从后向前进行迭代，直到我们找到第一个分隔符，会是更好的策略）。

接受 `Substring` 的函数非常罕见，大多数的函数要么接受 `String`，要么接受任意满足 `StringProtocol` 协议的类型。但是如果你确实需要传递 `Substring` 的话，最快的方式是用不指定任何边界的范围操作符 ... 通过下标方式访问字符串：

```
// 包含整个字符串的子串。  
let substring = sentence[...]
```

我们已经在集合类型协议一章中定义 `Words` 集合的时候看到过这样的例子了。

你可能会经不住 `StringProtocol` 种种优点的诱惑，将你的所有 API 从接受 `String` 实例转换为遵守 `StringProtocol` 的类型。但是 Swift 团队给我们的建议是不要这么做。

一般来说，我们的建议是坚持使用 `String`。在大部分 API 中只使用 `String`，而不是将它换为泛型（其实泛型本身也会带来开销），会更加简单和清晰。用户在有限的几个场合对 `String` 进行转换，也不会带来太大的负担。

但那些极有可能处理子字符串，同时又无法进一步泛型化成 `Sequence` 或 `Collection` 一般操作的 API，则不适用上述规则。标准库中的 `joined` 方法就是这样的例子。对于元素类型遵守了 `StringProtocol` 的序列，标准库就专门定义了一个重载的版本：

```
extension Sequence where Element: StringProtocol {  
    /// 将一个序列中的元素使用给定的分隔符拼接起为新的字符串，并返回  
    public func joined(separator: String = "") -> String  
}
```

这让你可以直接在(比如可能通过 `split` 得到的)子字符串的数组上调用 `joined`, 而不需要对这个数组进行 `map` 操作将每个子字符串转换为一个新的字符串。这要方便很多, 而且也快很多。

那些接受字符串并将它们转换为数字的数字类型初始化方法在 Swift 4 中也接受 `StringProtocol`。同样, 这在你想要处理一个子字符串数组中会特别有用:

```
let commaSeparatedNumbers = "1,2,3,4,5"  
let numbers = commaSeparatedNumbers.split(separator: ",")  
.compactMap { Int($0) }  
numbers // [1, 2, 3, 4, 5]
```

因为子字符串应当是短时存在的, 一般不建议一个函数返回子字符串, 除非你要处理的本来就是 `Sequence` 或 `Collection` 中返回切片的 API。如果你正在写一个只对字符串有效的类似的函数的话, 返回子字符串就意味着告诉读者没有发生复制。然而, 像是 `uppercased()` 这样包含内存申请以及创建新的字符串函数, 应该总是返回 `String`。

如果你想要扩展 `String` 为其添加新的功能, 将这个扩展放在 `StringProtocol` 会是一个好主意, 这可以保持 `String` 和 `Substring` API 的统一性。`StringProtocol` 设计之初就是为了在你想要对 `String` 扩展时来使用的。如果你想要将已有的扩展从 `String` 移动到 `StringProtocol` 的话, 唯一需要做的改动是将传入其他 API 的 `self` 通过 `String(self)` 换为具体的 `String` 类型实例。

不过需要记住, `StringProtocol` 并不是一个你想要构建自己的字符串类型时所应该实现的目标协议。文档中明确警告了这一点:

不要声明任何新的遵守 `StringProtocol` 协议的类型。只有标准库中的 `String` 和 `Substring` 是有效的适配类型。

## 编码单元视图

有时候字位簇无法满足需要时, 我们还可以向下到比如 Unicode 标量或者编码单元这样更低的层次中进行查看和操作。`String` 为此提供了三种视图: `unicodeScalars`, `utf16` 和 `utf8`。和 `String` 一样, 它们是双向索引的集合, 并且支持所有我们已经熟悉了的操作。和子字符串相似, 视图将与字符串本身共享存储; 它们只是简单地以另一种方式呈现底层的字节。

至于为什么你会想要对某个视图进行访问和操作，我们总结了几个常见原因。首先，也许你确实需要编码单元，也许为了在一个 UTF-8 编码的网页中进行渲染，或者和某个只接受某种特定编码的非 Swift API 进行交互，或者你需要字符串某种特定格式下的信息等。

比如，假设你正在开发一个 Twitter 客户端。虽然 Twitter 的 API 接受 UTF-8 编码的字符串，但 Twitter 的字符计算算法是基于 NFC 归一化 (NFC-normalized) 标量的（至少曾经如此，最近几年 Twitter 的字符计算方法已经更加复杂。但为了理解接下来的例子，我们仍然使用了之前的字符计数方法）。所以如果你想要为你的用户显示还可以输入多少字符，可以这么做：

```
let tweet = "Having ☕ in a cafe\u{301} in 🇺🇸 and enjoying the ☀️!"  
let characterCount = tweet.precomposedStringWithCanonicalMapping  
    .unicodeScalars.count  
/*end*/
```

NFC 归一可以对基础字母及合并标记进行转换，比如 "cafe\u{301}" 中的 e 和变音符可以被正确预组起来。`precomposedStringWithCanonicalMapping` 属性是定义在 Foundation 中的。

UTF-8 是用来存储或者在网络上发送文本的事实标准。因为 `utf8` 视图是一个集合，你可以用它来将字符串的 UTF-8 字节传递给任一接受一串字节的其他 API，例如 `Data` 或者 `Array` 的初始化方法：

```
let utf8Bytes = Data(tweet.utf8)  
utf8Bytes.count // 62
```

UTF-8 视图是 `String` 所有编码单元视图中，系统开销最低的。因为它是 Swift 字符串在内存中的原生存储格式。

要注意的是，`utf8` 集合不包含字符串尾部的 `null` 字节。如果你需要用 `null` 表示结尾的话，可以使用 `String` 的 `withCString` 方法或者 `utf8CString` 属性。后者会返回一个字节的数组：

```
let nullTerminatedUTF8 = tweet.utf8CString  
nullTerminatedUTF8.count // 63
```

而 `withCString` 则会调用一个你传递的函数。这个函数接受一个以 `null` 结尾的 UTF-8 字符串指针作为参数。可以用它调用那些接受 `char *` 作为参数的 C 函数。但在很多情况下，你却无需显式调用 `withCString`，为了函数可以被调用，调用编译器会帮你把一个 Swift 字符串转换成一个 C 字符串。例如，调用 C 标准库的 `strlen` 函数就可以这样：

```
strlen(tweet) // 62
```

我们将会在 互用性 这一章看到更多相关的例子。在多数情况下（如果字符串的底层存储已经是 UTF-8 编码的），Swift 和 C 的字符串转换不会带来任何额外的系统开销，因为 Swift 可以直接把指向字符串内部存储的指针传递给 C（因为存储实际上包含了一个尾部的 `null` 字节）。如果字符串在内存中使用了不同的编码，编译器将会自动插入对内容进行转码的代码，并把转换过的内容存放到一个临时的缓冲区。

`utf16` 视图的意义十分特殊，因为传统的 Foundation 的 API 会将字符串看作 UTF-16 的编码单元的集合。虽然 `NSString` 的接口是以透明的方式桥接到 `Swift.String` 的，但这背后 Swift 为你进行了转换的处理。其他一些像是 `NSRegularExpression` 或 `NSAttributedString` 的 Foundation API 通常接受的也是 UTF-16 形式的输入数据。我们会在 字符串和 Foundation 这一节中看到一个这方面的例子。

使用编码单元视图的第二个原因是，相比于操作完整的字符来说，对编码单元进行操作会更快一些。这是因为 Unicode 的字位分割算法相对还是复杂的，想要确定下一个字位簇的开头，需要额外向前进行查看。近些年来，以 `Character` 集合的形式遍历 `String` 的性能已经提升了许多。所以，在你冒着失去 Unicode 正确性的风险，用通过遍历编码单元视图的方式去获取（相对很小）的性能提升之前，请先进行性能的测量，避免得不偿失。一旦你使用了某一个编码单元的视图，你必须确保你的特定的算法可以在此基础上正确工作。例如：使用 UTF-8 视图来解析 JSON 应该没有问题，因为解析器所感兴趣的字符（像是逗号，引号或者括号等）都可以由单个的编码单元表示；JSON 数据中可能含有复杂的颜文字序列，但是这并不会对 JSON 解析产生影响。另一方面，如果你想要寻找字符串中某个单词出现的位置，而且希望搜索算法能够在字符串中找到想要的所有不同的归一化形式的话，使用特定编码单元视图可能就不管用了。

这些视图都没有提供我们想要的随机访问特性。这样造成的后果是，那些要求随机访问的算法将不能很好地运行在 `String` 和它的视图上。大部分的字符串处理任务都可以通过对字符串的顺序遍历完成，同时，算法可以存储一个子字符串，这样就可以在常数时间内再次访问字符串的这个部分。如果你真的需要随机存储的话，你依然可以把字符串自身或者它的视图转换为数组，

例如：Array(str) 或 Array(str.utf8)，然后对它们进行操作。如果想牺牲安全性来达到最大的性能，我们可以使用 withUTF8(str) { buffer in...}，它会为我们提供一个指向字符串存储的临时指针。

## 共享索引

字符串和它们的视图共享同样的索引类型，String.Index。也就是说，你可以从字符串中获取一个索引，然后将它用在某个视图的下标访问中。在下面的例子里，我们在字符串里搜索 "é" (这个字符包含了两个标量，字母 e 以及组合变音符号)。得到的索引指向的是 Unicode 标量视图中的第一个标量：

```
let pokemon = "Poke\u{301}mon" // Pokémon
if let index = pokemon.firstIndex(of: "é") {
    let scalar = pokemon.unicodeScalars[index] // e
    String(scalar) // e
}
```

只要是你从上往下进行，也就是在从字符，到标量，再到 UTF-16 或 UTF-8 编码单元这个方向上的话，这么做不会有什麼问题。但是另一个方向的话就不一定正确了，因为并不是每个编码单元视图中的有效索引都会在 Character 的边界上。下面的例子中，通过 UTF-16 编码单元上的索引访问字符串时，将发生崩溃：

```
let flag = "国旗" // [N] + [Z]
// 这个初始化方法基于 UTF-16 编译量创建索引
let someUTF16Index = String.Index(utf16Offset: 2, in: flag)
flag[someUTF16Index] // [Z]
```

String.Index 有一系列方法 (比如 samePosition(in:)) 和可失败的初始化方法 (String.Index.init?(\_:within:)) 来在不同视图中进行索引转换。如果输入的索引在给定的视图中没有对应的位置，这些方法将返回 nil。比如，尝试将标量视图中的变音符的位置转换为字符串的有效索引时，由于变音符在字符串中没有自己的位置，所以这个操作将会失败：

```
if let accentIndex = pokemon.unicodeScalars.firstIndex(of: "\u{301}") {
```

```
accentIndex.samePosition(in:pokemon) // nil  
}
```

## 字符串和 Foundation

Swift 的 String 类型和 Foundation 的 NSString 有着非常密切的关系。任意的 String 实例都可以通过 as 操作桥接转换为 NSString，而且那些接受或者返回 NSString 的 Objective-C API 也会把类型自动转换为 String。在 Swift 5.5 中，String 依然缺少很多 NSString 中所拥有的功能。不过因为字符串是非常基础的类型，而且每次都要将 String 的类型转为 NSString 实在是有些烦人，所以 String 受到了编译器的特殊对待：当你引入 Foundation 后，NSString 的成员就都可以在 String 实例上进行访问了，这让 Swift 的字符串变得比它们原本要强大得多。

拥有额外的特性毫无疑问是件好事儿，但是这也可能让字符串操作变得有些让人迷惑。比如说，如果你忘了引入 Foundation，你可以会奇怪为什么有些方法不可用。Foundation 历史上曾是 Objective-C 的框架，就算只考虑命名规则上的区别，也让 NSString 的 API 在和标准库一起使用时显得有些出戏。最后，也是很重要的一点是，两个库有一些重叠的特性，有时候会有两个名字完全不同的 API，但是它们做的事情却几乎一样。如果你在 Swift 问世之前就已经是一个长期的 Cocoa 开发者，并且已经学习过 NSString 的 API 的话，这不会带来很大的麻烦，但是对于新人来说，这会让他们十分迷惑。

我们已经看到过一个例子了，那就是标准库中的 `split` 方法和 Foundation 里的 `components(separatedBy:)`。另外还有很多其他不匹配的地方：Foundation 使用 `ComparisonResult` 来表示比较断言的结果，而标准库是围绕布尔值来设计断言的；像是 `trimmingCharacters(in:)` 和 `components(separatedBy:)` 接受一个 `CharacterSet` 作为参数，而很不幸，`CharacterSet` 这个类型的名字在 Swift 中是相当不恰当的（我们稍后再说）。`enumerateSubstrings(in:options:_)` 这个使用字符串和范围来对输入字符串按照字位簇、单词、句子或者段落进行迭代的超级强力的方法，在 Swift 中对应的 API 使用的是子字符串。（标准库中通过延迟序列暴露了相同功能的 API，这一点非常棒。）Foundation 的 API 通常都是对应了本地化的，它们可以根据本地化的特定标准来比较或列举字符串。标准库有意忽略了文本处理的这个重要方面。

下面的例子将按照单词来迭代字符串，对每个找到的单词，回调闭包都会被调用一次：

```
let sentence = """
    The quick brown fox jumped \
    over the lazy dog.
"""

var words: [String] = []
sentence.enumerateSubstrings(in: sentence.startIndex..., options: .byWords)
{ (word, range, _, _) in
    guard let word = word else { return }
    words.append(word)
}
words
// ["The", "quick", "brown", "fox", "jumped", "over", "the", "lazy", "dog"]
```

所有导入到 String 的 NSString 成员可以在 Foundation 源码仓库的 [NSStringAPI.swift](#) 文件中找到。

由于 Swift 字符串在内存中的原生编码是 UTF-8，而 NSString 是 UTF-16，这种差异会导致 Swift 字符串桥接到 NSString 时会有一些额外的性能开销。这就意味着，给诸如 enumerateSubstrings(in:options:using:) 这样的 Foundation API 传递 Swift 原生字符串可能不如直接传递 NSString 执行快。毕竟，对于 NSString，在以 UTF-16 计算的偏移上移动位置消耗的是常量时间，而这对于 Swift 字符串来说，是一个花费线性时间的操作。为了减少这种性能上的差异，Swift 实现了一种非常复杂却高效的索引缓存方法，使得这些原本线性时间的操作可以达到均摊常量时间 (amortized constant time) 的性能。

## 其他基于字符串的 Foundation API

说了这么多，其实原生的 NSString API 对于 Swift 字符串来说是使用起来最方便的 API 了，因为编译器为你完成了大部分的桥接工作。有些其他 Foundation 中处理字符串的 API 使用起来就需要更多的技巧，因为 Apple (还？) 没有为它们创造特殊的 Swift 封装层。比如 Foundation 中用来代表正则表达式的 NSRegularExpression 就是一例。要在 Swift 中成功使用这个类型，你必须注意下面这些：

- 虽然所有原来接受 `NSString` 的 `NSRegularExpression` API 现在都接受 `Swift.String` 了，不过整个 API 的基础还是 `NSString` 的 UTF-16 编码单元集合的概念。Foundation 使用 `NSRange` 类型作为范围来表示“子字符串”，这些都是以 UTF-16 编码单元来完成工作的。
- 不这么重要的一点，频繁地在 `String` 和 `NSString` 之间发生桥接可能会带来意外的性能开销。关于详细说明，可以参考 [WWDC 2018 中 session 229](#)。

比如说，`rangeOfFirstMatch(in:options:range:)` 方法将会把第一个匹配正则表达式的位置，通过 `NSRange` 返回回来，这里并不会使用 `Range<String.Index>`。除此之外，因为 Objective-C 没有可选值的概念，这个方法会返回一个哨岗值 `NSRange(location: NSNotFound, length: 0)` 来代表“没有找到”。

`NSRange` 是一个包含两个整数字段 `location` 和 `length` 的结构体：

```
public struct NSRange {  
    public var location: Int  
    public var length: Int  
}
```

在字符串中，这两个字段可以指定 UTF-16 编码单元的一段位置。Swift 提供了在 `Range<String.Index>` 和 `NSRange` 之间相互转换的初始化方法。相比起每次都要记住深入到 UTF-16 的形式进行操作，使用 `NSRange` 更不容易出错。不过，这并没有让来回转换所需要的代码变得更少。下面是一个创建正则表达式和使用一个搜索字符串去匹配它的例子：

```
extension NSRegularExpression {  
    func firstMatch(in input: String) -> Substring? {  
        // NSRegularExpression 需要一个“子字符串”作为 NSRange 来搜索。  
        var inputRange = NSRange(input.startIndex..., in: input)  
        let utf16Length = inputRange.length  
        while true {  
            let matchNSRange = self.rangeOfFirstMatch(in: input, range: inputRange)  
            guard matchNSRange != NSRange(location: NSNotFound, length: 0) else {  
                // 模式没有找到  
                return nil  
            }
```

```
}

// 将 NSRange 转换为 Swift Range.

guard let matchRange = Range(matchNSRange, in: input) else {
    // 匹配没有落在字符边界 → 再来一次。
    inputRange.location =
        matchNSRange.location + matchNSRange.length
    inputRange.length = utf16Length - inputRange.location
    continue
}

return input[matchRange]
}

}

}

let input = "My favorite numbers are -9, 27, and 81."
let regex = try! NSRegularExpression(pattern: "-?[0-9]+")
if let match = regex.firstMatch(in: input) {
    print("Found: \(match)")
} else {
    print("Not found")
}
// Found: -9
```

let regex = try!... 这行代码，是一个正确使用强制解包(或者说，和它等效的错误处理方式)的好例子。我们通过字符串字面量初始化了 NSRegularExpression。如果在开发过程中这个语句因为我们传递了无效的表达式而崩溃，我们可以很容易地修正它，而且它肯定不会在最终产品中崩溃。想要了解更多这方面的话题，可以参看可选值的有关部分。

你可能会好奇在上面的代码中我们为什么需要循环。NSRegularExpression 会按照 Unicode 标量的方式来检查字符串。这意味着它进行匹配所找到的标量，可能是一个字位簇(也就是 Character)的一部分。下面就是一个例子，一个国旗 emoji 由两个标量组成，我们用其中之一进行查找：

```
let flag = "🇪🇸"
```

```
let regex2 = try! NSRegularExpression(pattern: "[B] ")  
regex2.rangeOfFirstMatch(in: flag, range: NSRange(flag.startIndex..., in: flag))  
// {0, 2}
```

只考虑 `NSRegularExpression` 的话，这个匹配是有效的，但是这大概率不是 Swift 程序员所期望和想要看到的结果。另外，因为这个匹配落在了 `Character` 的一个组成部分上，它在 Swift 字符串中不构成一个有效的范围 (range)。当我们无法把匹配得到的 `NSRange` 转换回 `Range<String.Index>` 的时候，我们就将这个匹配忽略掉，然后从匹配位置再次开始新一轮匹配。

总结一下，我们通过下面的方法为原来的 API 添加一层优雅而安全的封装：

- 隐藏所有 `NSRange` 的使用
- 在 `String` 和 `Foundation` 处理 `Unicode` 的不同方式之间进行转换
- 使用可选值而非哨岗值来表达“没有匹配”的情况
- 返回子字符串而非一个范围对象

我们已经有了寻找第一个匹配的方法，现在来用同样的方式从一个字符串中把所有的匹配都找出来。`NSRegularExpression` 中等价的 API 是 `matches(in:options:range:)`，它会返回一个装有 `NSTextCheckingResult` 对象的数组。这是一个相对复杂的类，不过对于我们的目的来说，只需要知道这个类代表了一个匹配项，并提供相应的 `NSRange`。最终的代码实际上要比上面的 `firstMatch(in:)` 简单一些，因为我们可以把循环的部分去掉：

```
extension NSRegularExpression {  
    func matches(in input: String) -> [Substring] {  
        let inputRange = NSRange(input.startIndex..., in: input)  
        let matches = self.matches(in: input, range: inputRange)  
        return matches.compactMap { match in  
            guard match.range != NSRange(location: NSNotFound, length: 0) else {  
                // NSTextChecking 的 range 永不为 "nil"。  
                fatalError("Should never happen")  
            }  
        }  
    }  
}
```

```
guard let matchRange = Range(match.range, in: input) else {
    // 匹配没有落在字符边界
    return nil
}
return input[matchRange]
}
}
}
regex.matches(in: input) // [-9, "27", "81"]
```

你可能也发现了，把这些 Foundation 的 API 进行改造，让它们更符合 Swift 的习惯，是一件很费力的事。而且不得不使用处理 Unicode 的 API 来以另一种方式操作字符串，本身就是很方便的，这也更容易引入难以发现的 bug。在不远的将来，Swift 可能会得到原生的正则表达式语法支持，但在那之前，利用 Foundation API 依然还是最合适的方式。

除了 NSRegularExpression，用来表示富文本的 NSAttributedString 是另一个有着类似的不匹配特质的 Foundation 类。Apple 引入了一个名为 struct AttributedString 的更现代化的版本，让它能在 Swift 中更加易用。如果你的目标部署版本允许，我们建议你使用这个类型。

## 字符范围

说到范围，你可能已经尝试过对字符的范围进行迭代，不过它无法工作：

```
let lowercaseLetters = ("a" as Character)..."z"
for c in lowercaseLetters { // 错误
    ...
}
```

(这里将“a”转换为 Character 是必要的，否则字符串字面量的默认类型将是 String；我们需要告诉类型检查器这里我们想要一个 Character 范围。)

在内建集合类型中我们已经解释过这个操作失败的原因了：Character 并没有实现 Strideable 协议，而只有实现了这个协议的范围才是可数的集合。对于一个字符范围，唯一能够进行的操作是将它和其他字符进行比较，比如，我们可以检查某个字符是否在一个范围内：

```
lowercaseLetters.contains("A") // false
lowercaseLetters.contains("c") // true
lowercaseLetters.contains("é") // false
```

而对 Unicode.Scalar 类型来说，至少当你保持在 ASCII 或者其他一些有很好排序的 Unicode 类别的子集时，可数范围的概念就有意义了。Unicode 标量的顺序是通过它们的代码点的值进行定义的，所以在两个边界之间，一定存在的有限个数的标量。默认情况下 Unicode.Scalar 不遵守 Strideable，但是我们可以为它追加满足这个协议：

```
extension Unicode.Scalar: Strideable {
    public typealias Stride = Int

    public func distance(to other: Unicode.Scalar) -> Int {
        return Int(other.value) - Int(self.value)
    }

    public func advanced(by n: Int) -> Unicode.Scalar {
        return Unicode.Scalar(UInt32(Int(value) + n))!
    }
}
```

(我们这里忽略了 0xD800 和 0xDFFF 之间的代理编码点不是有效的 Unicode 标量值这个事实。构建一个与这个区域有重合的范围，被认为是程序员的错误。)

让一个不属于你的类型实现某个协议是一种有问题的做法，一般来说，不建议这样。如果你使用的其它程序库添加了同样的扩展，或者原始类型的供应商提供了对同样协议的支持 (但实现的方式很可能和你使用的不同)，就会发生代码冲突。一个更好的做法，是为这种类型创建一个包装类，把要实现的协议添加到这个包装类上。我们将会在 [协议](#) 这一节再回来讨论这个话题。

有了这个扩展，我们就可以创建一个可数的 Unicode 标量范围。通过它，可以方便地生成一个字符数组：

```
let lowercase = ("a" as Unicode.Scalar)..."z"  
Array(lowercase.map(Character.init))  
/*  
["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n",  
"o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"]  
*/
```

## CharacterSet

让我们来看看最后一个有意思的 Foundation 类型，它是 CharacterSet。CharacterSet 是一种存储一组 Unicode 码点的有效手段。它经常被用来检查一个特定的字符串是否包含像是 alphanumerics (英数字符) 或者 decimalDigits (数字) 这类某个特定的字符子集中的字符。CharacterSet 这个名字是从 Objective-C 中导入进来的，其实 CharacterSet 和 Swift 的 Character 完全不是一回事。可能 UnicodeScalarSet 会是一个更好的名字。

我们可以通过创建一些复杂颜文字的集合来说明这一点。在下面的例子中，虽然看上去我们只放入了两个颜文字，但是使用第三个颜文字来测试是否是集合的成员时，返回的确是 true。这是因为女消防员的颜文字实际上是女人 + ZWJ + 消防车的组合：

```
let favoriteEmoji = CharacterSet("👩🚒").unicodeScalars  
// 错误！还是正确？  
  
favoriteEmoji.contains("🚒") // true
```

CharacterSet 提供了一些工厂初始化方法，比如 .alphanumerics 或者 .whitespacesAndNewlines。它们大部分对应着 Unicode 字符分类 (每个编码点都被赋予了一个分类，比如“字母”或者“非空格标记”)。这些分类覆盖了所有的文字，不单单是 ASCII 或者 Latin-1，所以通常来说这些预先定义的集合中的成员个数都非常庞大。这个类型遵守 SetAlgebra 协议，该协议中包含一些集合操作，比如检查元素是否在集合内，或者构建并集或交集等。

## Unicode 属性

在 Swift 5 里, CharacterSet 的部分功能被移植到了 Unicode.Scalar。我们不再需要 Foundation 中的类型来测试一个标量是否属于某个官方的 Unicode 分类了, 现在只要直接访问 Unicode.Scalar 中的某个属性就好了, 例如: isEmoji 或者 isWhiteSpace。为了避免在 Unicode.Scalar 中塞入过多的成员, 所有 Unicode 属性都放在了 properties 这个名字空间里:

```
("\u{1f600}" as Unicode.Scalar).properties.isEmoji // true  
("\u{ff1f}" as Unicode.Scalar).properties.isMath // true
```

你可以在 [Unicode.Scalar.Properties](#) 找到完整的属性列表。其中, 大部分都是布尔类型, 不过也有例外。例如: age (该标量被引入时 Unicode 的版本), name (官方 Unicode 字符名称), numericValue(用于少部分有自己编码点却表示数值概念的非拉丁语系字符, 例如, 字符 \u{00d7} 的 numericValue 是 4.0, 字符 \u{00b7} 的 numericValue 是 0.2, 对于没有数值概念的字符, 这个属性是 nil) 和 generalCategory (一个表示 Unicode 标量最常用分类的枚举)。

来看个例子, 现在列出字符串中每一个标量的编码点、名称和一般分类只需要对字符串做一点格式化就行了:

```
"I'm a \u{1f469}.".unicodeScalars.map { scalar -> String in  
    let codePoint = "U+\\" + String(scalar.value, radix: 16, uppercase: true) + "\"  
    let name = scalar.properties.name ?? "(no name)"  
    return "\u{(" + codePoint + ")}: \u{(" + name + ")} - \u{(" + scalar.properties.generalCategory + ")}"  
}.joined(separator: "\n")  
  
/*  
U+49: LATIN CAPITAL LETTER I – uppercaseLetter  
U+2019: RIGHT SINGLE QUOTATION MARK – finalPunctuation  
U+6D: LATIN SMALL LETTER M – lowercaseLetter  
U+20: SPACE – spaceSeparator  
U+61: LATIN SMALL LETTER A – lowercaseLetter  
U+20: SPACE – spaceSeparator  
U+1F469: WOMAN – otherSymbol  
U+1F3FD: EMOJI MODIFIER FITZPATRICK TYPE-4 – modifierSymbol  
U+200D: ZERO WIDTH JOINER – format  
U+1F692: FIRE ENGINE – otherSymbol
```

*U+2E: FULL STOP – otherPunctuation*

\*/

Unicode 标量的这些属性非常底层，它们主要是为表达 Unicode 中那些不为人熟知的术语而定义的。如果在更为常用的 Character 这个层面也提供一些类似的分类，用起来会更加方便。为此，Swift 5 也为 Character 添加了一系列表达字符类型的属性：

```
Character("4").isNumber // true  
Character("$").isCurrencySymbol // true  
Character("\n").isNewline // true
```

但这些定义在 Character 上分类和 Unicode 标量属性不同，它们并不是 Unicode 官方规格中的一部分，因为 Unicode 仅根据标量值进行分类，而不是扩展字位族。Swift 标准库的目标则是为一个字符的特性提供最为合理的信息。但由于支持的语系数量实在过于庞大，并且，Unicode 还具备几乎无限种标量的组合形式，这些分类中总有一些不那么准确，或者和其它编程语言或工具提供的信息并不一致。但无论是这些不匹配的问题，还是编程语言自身，也都会随着时间不断地改进和进步。

## String 和 Character 的内部结构

和标准库中的其他集合类型一样，字符串也是一个实现了写时复制的值语义类型。一个 String 实例存储了一个对缓冲区的引用，实际的字符数据被存放在那里。当你（通过赋值或者将它传递给一个函数）创建一个字符串的复制，或者创建一个子字符串时，所有这些实例都共享同样的缓冲区。字符数据只有当与另外一个或多个实例共享缓冲区，且某个实例被改变时，才会被复制。关于写时复制的更多内容，请参看[结构体和类一章](#)。

Swift 原生字符串（相对于从 Objective-C 接收的字符串）在内存中是用 UTF-8 格式表示的。如果你了解了这一点，就能通过它获取理论上字符串处理的最佳性能，因为遍历 UTF-8 视图要比遍历 UTF-16 视图或 Unicode 标量视图更快。并且，UTF-8 也是大部分字符串处理工作使用的格式，因为通过文件或网络等数据源获取的内容，也大多使用了 UTF-8 编码。

而从 Objective-C 接收到的字符串则是通过一个 `NSString` 表示的。在这种时候，为了让桥接尽可能高效，`NSString` 直接扮演了 Swift 字符串的缓冲区。一个基于 `NSString` 的 `String` 在被改变时，将被转换为原生的 Swift 字符串。

对于那些小于 16 个（在 32 位平台上是小于 11 个）UTF-8 编码单元的小型字符串，作为特别优化，Swift 并不会为其创建专门的存储缓冲区。由于字符串最多只有 16 字节，这些编码单元可以用内连的方式存储。尽管 15 个 UTF-8 编码单元看上去并不怎么显眼，但它能表达的字符串却比我们想象的要多很多。对于 JSON 等一些机器可以读懂的格式，很多 key 和值（例如数值和布尔值）就都在这个长度以内，特别是用在 JSON 里的，通常都只是 ASCII 字符。

另外，这种小字符串的优化方法也用在了 `Character` 类型的内部表示上。下面是来自 [Swift 标准库源代码](#) 中简化出来的一段实现：

```
public struct Character {  
    internal var _str: String  
  
    internal init(unchecked str: String) {  
        self._str = str  
        // ...  
    }  
}
```

一个字符在内部被表示成长度为 1 的字符串。由于大部分的字符长度都在 15 个 UTF-8 字节之内（最常见的例外应该是超长的 emoji 序列），小字符串优化意味着绝大多数的 `Character` 值都不需要额外的堆空间或者引用计数。

## 字符串字面量

这一章内容至此，我们一直都在用近乎等价的方式交换使用 `String("Hello")` 和 `"Hello"`，但这两者是不同的。就如在 [集合协议](#) 一章中涉及的数组字面量一样，`""` 是字符串字面量。你可以通过实现 `ExpressibleByStringLiteral` 协议让你自己的类型支持通过字符串字面量进行初始化。

然而，字符串字面量还要比数组字面量稍复杂一些，因为它是下面这三个协议架构的一部分：`ExpressibleByStringLiteral`, `ExpressibleByExtendedGraphemeClusterLiteral` 和

ExpressibleByUnicodeScalarLiteral。其中，每个协议都约束了一个用它们各自表示的字面量创建对象的 init 方法。但除非你真的需要根据 Unicode 标量还是字位族仔细调整初始化逻辑，否则，只需要实现字符串的版本就好了。Swift 会基于你提供的字符串版本为其他版本的 init 提供默认实现。

来看一个让自定义类型支持 ExpressibleByStringLiteral 的例子，我们定义了一个 SafeHTML 类型。它仅仅是字符串类型的包装，但提供了额外的类型安全性。当使用 SafeHTML 值的时候，我们可以确保它表示的字符串中，所有有潜在风险的 HTML 标签都已经被转义了，这可以避免招致一些安全问题：

```
extension String {
    var htmlEscaped: String {
        // 替换所有开闭尖括号
        // 实际的实现应该会更加精细
        return replacingOccurrences(of: "<", with: "&lt;")
            .replacingOccurrences(of: ">", with: "&gt;")
    }
}

struct SafeHTML {
    private(set) var value: String

    init(unsafe html: String) {
        self.value = html.htmlEscaped
    }
}
```

通过 SafeHTML，就可以让处理我们 view 的 API 只接受转义过的字符串。但这样做的一个缺点就是需要在调用这些 API 之前写很多包装字符串的代码。不过幸运的是，这个工作可以通过让 SafeHTML 实现 ExpressibleByStringLiteral 自动完成：

```
extension SafeHTML: ExpressibleByStringLiteral {
    public init(stringLiteral value: StringLiteralType) {
        self.value = value
    }
}
```

```
    }  
}
```

这就确保了代码中所有的字符串字面量都是安全的(这种假设是合理的,毕竟字面量,指的是我们硬编码在代码里的字符串):

```
let safe:SafeHTML = "<p>Angle brackets in literals are not escaped</p>"  
// SafeHTML(value: "<p>Angle brackets in literals are not escaped</p>")
```

在上面的代码中,必须显式定义 `safe` 的类型,否则,`safe` 就被推导成 `String` 了。但是,如果编译器可以通过上下文明确需要一个 `SafeHTML` 对象,例如给属性赋值或函数调用,我们就能使用类型自动推导了。

## 字符串插值

字符串插值(String interpolation)是从 Swift 发布之初就存在的语法特性。它可以让我们在字符串字面量中插入表达式(例如:`"a * b = \(a * b)"`)。Swift 5 则进一步开放了公共 API,可以支持在构建自定义类型时使用字符串插值。这些 API 正好可以用来改进之前实现的 `SafeHTML`。我们经常要基于用户输入内容创建包含 HTML 标签的字面量:

```
let input = ... // 这部分由用户输入, 不安全!  
let html = "<li>Username: \(input)</li>"
```

`input` 中的内容必须被转义后使用,因为它的来源并不安全。但 `html` 变量中字面量的分段不应发生变化,因为我们在那里就是要写入带有 HTML 标签的值。为了实现这个逻辑,我们可以给 `SafeHTML` 创建一个自定义的字符串插值规则。

Swift 的字符串插值 API 由两个协议组成: `ExpressibleByStringInterpolation` 和 `StringInterpolationProtocol`。前者由那些需要以字符串插值的形式构建的自定义类型实现,后者也可以由相同的类型或者其它相关的类型实现,它包含了若干一步步构建(创建目标对象需要的)插值的方法。

`ExpressibleByStringInterpolation` 继承自 `ExpressibleByStringLiteral`。之前,我们已经让 `SafeHTML` 实现了后者,所以,只要再给 `SafeHTML` 实现一个前者要求的初始化方法,它就是

一个实现了 ExpressibleByStringInterpolation 的类型了。这个初始化方法，接受一个实现了 StringInterpolationProtocol 协议的参数。例如，我们同样给它传递 SafeHTML：

```
extension SafeHTML: ExpressibleByStringInterpolation {  
    init(stringInterpolation: SafeHTML) {  
        self.value = stringInterpolation.value  
    }  
}
```

但这样做，就要求 SafeHTML 是一个实现了 StringInterpolationProtocol 的类型。这个协议有三个约束，分别是：一个初始化方法，一个 appendLiteral 方法，以及若干 appendInterpolation 方法。Swift 有一个实现了 StringInterpolationProtocol 协议的默认类型 DefaultStringInterpolation，正是它让我们从标准库中免费获得了通过字符串插值生成新字符串的能力。而我们需要做的，则是自定义一个可以从字符串插值中转义出 SafeHTML 对象的 appendInterpolation 方法：

```
extension SafeHTML: StringInterpolationProtocol {  
    init(literalCapacity: Int, interpolationCount: Int) {  
        self.value = ""  
        value.reserveCapacity(literalCapacity)  
    }  
  
    mutating func appendLiteral(_ literal: String) {  
        value += literal  
    }  
  
    mutating func appendInterpolation<T>(_ x: T) {  
        self.value += String(describing: x).htmlEscaped  
    }  
}
```

这里，初始化方法通知了插值类型大约需要多少空间存储所有要合并的字面量，以及期望的插值数量。我们原本可以忽略这两个参数，直接把 value 初始化成空字符串。不过，我们最好至少把 literalCapacity 传递给 String.reserveCapacity。这会告诉字符串预先去申请所期望的存

储空间，以此避免在拼接过程中进行多次昂贵的内存分配操作。因为我们无法完全预先知道被插入的内容到底有多大，所以无法做到完美，但至少聊胜于无。我们可以进一步对它进行改进，比如为每一个插入段添加一个预估尺寸（比如 10 个字节）。

appendLiteral 方法的实现直接在 value 属性后面追加新的内容就行了，因为我们可以默许插入的字面量是安全的（就像之前实现 ExpressibleByStringLiteral 一样）。而 appendInterpolation( \_: ) 方法则接受一个任意类型的参数并使用 String(describing:) 方法把参数转换成一个字符串。在把这个字符串添加到 value 之前，要先对它进行转义。

由于 appendInterpolation( \_: ) 方法的参数不需要名称，我们可以像 Swift 默认的字符串插值一样使用它：

```
let unsafeInput = "<script>alert('Oops!')</script>"  
let safe: SafeHTML = "<li>Username: \($0)</li>"  
  
safe  
/*  
SafeHTML(value: "<li>Username:  
&lt;script&gt;alert('Oops!')&lt;/script&gt;</li>"  
*/
```

编译器会把包含插值的字符串翻译成一系列 appendLiteral 和 appendInterpolation 调用，这些调用对应着我们自己实现的 StringInterpolationProtocol 中的方法。通过这种方式，我们就可以用合适的方式保存字符串中每一部分的值。当所有的字面量和插入的值都处理完了，最终的结果就会传递给 init(stringInterpolation:) 方法，一个 SafeHTML 对象就创建出来了。

在这个例子中，我们选择了让同一个类型同时实现了 ExpressibleByStringInterpolation 和 StringInterpolationProtocol，因为它们的实现共享同一个结构（都只需要设置一个字符串属性）。但是，当构建字符串插值使用的数据结构和通过字符串插值要创建的对象结构不同时，可以用不同类型实现这两个协议就很有用了。

除此之外，字符串插值还有很多其它的用法。本质上，\(...) 这种语法就是 appendInterpolation 的方法调用，因此，我们还能创建一些带有参数名称版本的 appendInterpolation 方法。用这种方法，就可以让 SafeHTML 通过指定 “raw” 这种方式，支持插入非转义字符串的功能了：

```
extension SafeHTML {  
    mutating func appendInterpolation<T>(raw x:T) {  
        self.value += String(describing:x)  
    }  
}  
  
let star = "<sup>*</sup>"  
let safe2:SafeHTML = "<li>Username\(raw: star):\\"unsafeInput</li>"  
safe2  
/*  
SafeHTML(value: "<li>Username<sup>*</sup>:  
&lt;script&gt;alert('Oops!')&lt;/script&gt;</li>"  
*/
```

## 定制字符串描述

像 print, String(describing:) 这些函数以及字符串插值这种用法，我们可以传递给它们任意类型作为参数。即使没有对传给它们的类型做任何定制，你可能仍旧会得到一个可以接受的结果，因为结构体会默认打印它们的属性：

```
let safe:SafeHTML = "<p>Hello, World!</p>"  
print(safe)  
// 这里会打印 SafeHTML(value: "<p>Hello, World!</p>")
```

除此之外，你可能还希望得到的结果更美观，或者你的类型中可能还包含一些不需要显示的私有变量。别担心！只需要一分钟，就可以让你的自定义类型满足 CustomStringConvertible 协议，这样当它传到 print 的时候，就能打印出完全符合你需要的结果：

```
extension SafeHTML: CustomStringConvertible {  
    var description: String {  
        return value  
    }  
}
```

现在，如果有人尝试把 SafeHTML 转换成一个字符串，无论用什么方式，把它传递给 print 也好，把它传递给 String(describing:) 也好，或者把 SafeHTML 用在字符串插值里，就会直接得到它的 value 属性值了：

```
print(safe) // <p>Hello, World!</p>
```

另外，还有一个 CustomDebugStringConvertible。为了调试方便，它允许我们为对象的输出定义另外一种格式。当调用 String(reflecting:) 的时候，就会得到这个调试版的结果了：

```
extension SafeHTML: CustomDebugStringConvertible {  
    var debugDescription: String {  
        return "SafeHTML: \(value)"  
    }  
}
```

但如果你没有实现 CustomDebugStringConvertible，String(reflecting:) 就会选择使用 CustomStringConvertible 提供的结果，反之也是。如果你的类型没有实现 CustomStringConvertible，String(describing:) 会选择使用 CustomDebugStringConvertible 提供的结果。所以，如果你的自定义类型比简单，就没必要实现 CustomDebugStringConvertible。但是，如果你的定义类型是个容器，让它实现 CustomDebugStringConvertible 则是一种更为友好的行为。通过它，可以打印容器中每个元素在调试模式的信息。或者，当你为了调试打印结果之后还要做一些特别的处理，也应该通过实现 CustomDebugStringConvertible 完成。但是，如果你为 description 和 debugDescription 提供的结果相同，那么二者选其一去实现就好了。

顺便说一下，Array 总是会打印它包含的元素的调试版信息，即使你把它传递给 String(describing:)。这是因为数组的普通字符串描述永远都不应该对用户呈现。举个例子，对于空字符串 "", String.description 会忽略包围字符串的引号。因此，如果定义一个包含空字符串的数组，打印这个数组的时候如果使用字符串的普通描述，就会得到类似 [,,] 这样的结果，这看上去就跟发生了 bug 一样。因此，数组总是会使用元素的调试版描述。

既然实现了 CustomStringConvertible 就意味着一个类型可以为自己提供一份漂亮的 print 结果，你可能会想着写一个类似下面这样的泛型函数：

```
func doSomethingAttractive<T:CustomStringConvertible>(with value: T) {  
    //在这里打印参数 value, 因为你知道类型 T 一定可以提供一份  
    //让人满意的打印结果。  
}
```

但你不应该用这种方式使用 `CustomStringConvertible`。你应该总是坚持使用 `String(describing:)`，而不是用这种甄别类型的方式去试探它是否提供了 `description` 属性。即便有些类型的打印结果很糟糕，你也应该认了，那是属于类型自己的事情，但这样做至少对任何类型都不会失败。同时，这也是一个督促你为复杂的自定义类型实现 `CustomStringConvertible` 的理由，通常，几行代码就能搞定了。

## LosslessStringConvertible

要介绍的最后一个 `...StringConvertible` 协议是 `LosslessStringConvertible`。它构建在 `CustomStringConvertible` 上，添加了一个初始化方法，用来将字符串转换回你的自定义类型。

```
public protocol LosslessStringConvertible: CustomStringConvertible {  
    /// 从字符串表述实例化一个满足协议的类型的实例。  
    init?(description: String)  
}
```

正如其名，这个字符串描述必须是一个对原来值的无损 (`lossless`) 的，保留了全部值信息的表示，我们可以通过它返回到原值。在标准库中，数字类型、字符串类型和 `Bool` 类型满足了 `LosslessStringConvertible`。这个协议很少被使用。如果你需要的不是简单地把单个值序列化为字符串，那么 Swift 的 `Codable` 系统会更加灵活。关于这个话题的详细内容，请参看 [编码和解码的章节](#)。

## 文本输出流

标准库中的 `print` 和 `dump` 函数会把文本记录到 [标准输出](#) 中。这两个函数的默认实现调用了 `print(_:to:)` 和 `dump(_:to:)`。`to` 参数就是输出的目标，它可以是任何实现了 `TextOutputStream` 协议的类型：

```
public func print<Target:TextOutputStream>
(_ items: Any..., separator: String = " ",
terminator: String = "\n", to output: inout Target)
```

标准库维护了一个内部的文本输出流，这个流将所有输入的内容写到标准输出中。你还能将文本写到其他什么地方吗？嗯，String 是标准库中唯一的和输出流紧密相关的类型 (Substring 和 DefaultStringInterpolation 也是输出流，但是你通常很少对它们进行写入)：

```
var s = ""
let numbers = [1,2,3,4]
print(numbers, to: &s)
s // [1, 2, 3, 4]
```

这在你想要将 print 和 dump 的输出重新定向到一个字符串的时候会很有用。顺带一提，标准库也利用了输出流，来让 Xcode 获取所有的标准输出。你可以  
在标准库中找到这样的全局变量声明：

```
public var _playgroundPrintHook: ((String) -> Void)?
```

如果这个变量不是 nil，print 就将用一个特殊的输出流来将所有打印的内容同时传递给标准输出和这个函数。这个声明甚至是公开的，所以你可以用它来做很多有意思的事情：

```
var printCapture = ""
_playgroundPrintHook = { text in
    printCapture += text
}
print("这本应该只打印到标准输出")
printCapture // 这本应该只打印到标准输出
```

不过不要依赖它！这个 API 并没有出现在文档里，我们也不知道当你给它重新赋值时 Xcode 的相关功能会不会出问题。

我们还可以创建自己的输出流。TextOutputStream 协议只有一个要求，就是一个接受字符串，并将它写到流中的 write 方法。比如，这个输出流将输入写到一个缓冲数组里：

```
struct ArrayStream: TextOutputStream {  
    var buffer: [String] = []  
    mutating func write(_ string: String) {  
        buffer.append(string)  
    }  
}  
  
var stream = ArrayStream()  
print("Hello", to: &stream)  
print("World", to: &stream)  
stream.buffer // ["", "Hello", "\n", "", "World", "\n"]
```

文档明确允许那些将输出写到输出流的函数在每次写操作时可以多次调用 `write(_)`。这就是上面例子中包含有换行分隔元素和一些空字符串的原因。这是 `print` 函数的一个实现细节，它可能会在未来的版本中发生改变。

另一个可能的方式是扩展 `Data` 类型，让它接受流输入，并输出 UTF-8 编码的结果：

```
extension Data: TextOutputStream {  
    mutating public func write(_ string: String) {  
        self.append(contentsOf: string.utf8)  
    }  
}  
  
var utf8Data = Data()  
var string = "café"  
utf8Data.write(string)  
Array(utf8Data) // [99, 97, 102, 195, 169]
```

输出流的源可以是实现了 `TextOutputStreamable` 协议的任意类型。这个协议需要 `write(to:)` 这个泛型方法，它可以接受满足 `TextOutputStream` 的任意类型作为输入，并将 `self` 写到这个输出流中。在标准库中，`String`, `Substring`, `Character` 和 `Unicode.Scalar` 都满足 `TextOutputStreamable`，不过你也为你的自定义类型添加 `TextOutputStreamable` 支持。

我们已经知道，`print` 函数在内部使用了一些满足 `TextOutputStream` 的东西来封装标准输出。你也可以为标准错误写一些类似的东西，比如：

```
struct StdErr: TextOutputStream {  
    mutating func write(_ string: String) {  
        guard !string.isEmpty else { return }  
  
        // Swift 字符串可以直接传递给那些接受 const char *  
        // 作为参数的 C 函数。参阅互用性一章获取更多信息！  
        fputs(string, stderr)  
    }  
}  
  
var standarderror = StdErr()  
print("oops!", to: &standarderror)
```

## 回顾

Swift 的字符串和其他所有主流编程语言中的字符串都很不同。如果你已经习惯了将字符串作为编码单元数组进行处理的话，你可能会需要一点时间来切换你的思维方式，相比于简洁，Swift 中的字符串以 Unicode 正确性为第一优先。

最终，我们认为 Swift 做出了正确的选择。其他编程语言假装 Unicode 文本没有那么复杂，其实这不是真相。在长远看来，严格的 Swift 字符串可以让你避免写出一些本来会出现的 bug，这可以节省下很多时间。与之相比，努力去忘却整数索引所花费的时间完全可以忽略不计。

我们曾经很习惯进行“字符”的随机访问，却不曾意识到，在字符串处理的代码中，这几乎是个用不到的特性。希望本章中的例子能够说服你，让你也认为简单的按顺序遍历对绝大多数普通操作是完全可用的。强制你显式地指出你想要使用的字符串表示是字位簇、Unicode 标量、UTF-16 还是 UTF-8 编码单元，这样做可以添加额外的安全措施，你的代码的读者也会因此心存感激。

当 Chris Lattner 在 2016 年 7 月展望 Swift 的字符串实现的目标时，他这样结尾：

我们的目标是在字符串处理上做得比 Perl 更好!

Swift 5.5 距离这个目标还有距离，还有不少大家希望的特性是缺失的，比如把更多的字符串 API 从 Foundation 移动到标准库，比如原生语言级别的正则表达式支持，比如字符串格式化和解析的 API，以及更加强大的字符串插值等。不过好消息是，Swift 团队表达了在今后版本中处理所有这些话题的兴趣。

泛型

9

泛型编程是一种可以保持类型安全性的代码重用技术。例如，标准库通过泛型编程让 `sort` 方法可以接受一个自定义的比较操作符，并且，让这个比较操作符的参数和进行排序的序列中元素的类型相同。类似的，为了以类型安全的方式提供访问和修改数组的 API，`Array` 也是一个泛型类型。

当谈论 Swift 中的泛型编程时，我们通常指的是对类型泛化后的编程（一个显著的语法特征就是要使用尖括号具像化这些泛化的类型，例如：`Array<Int>`）。不过，泛型这个概念却远不止泛化类型。我们可以认为泛型是多态 (**polymorphism**) 的一种形式，而多态则是指一个接口或名称可以通过多个类型进行访问的现象。

至少有四种不同的概念，可以归纳到多态编程这个范畴里：

- 我们可以定义多个同名但是类型不同的函数。例如，在 函数 这一章，我们定义了三个不同的 `sortDescriptor` 方法，它们每个都有不同的参数类型。这种用法叫做重载 (**overloading**)，或者更技术地说，这是一种（为了解决排序这个问题而特别设置的）专属多态 (**ad hoc polymorphism**)。
- 当一个函数或方法接受类 C 作为参数的时候，我们也可以给它传递 C 的派生类，这种用法叫做子类型多态 (**subtype polymorphism**)。
- 当一个函数（通过尖括号语法）接受泛型参数的时候，我们管这个函数叫做泛型函数 (**generic function**)（这和泛型类型类似）。这种用法叫做参数化多态 (**parametric polymorphism**)。这些泛型化的参数，也被叫做泛型 (**generics**)。
- 我们可以定义一个协议并让多个类型实现它。这是另外一种更加结构化的专属多态。我们将在 协议 这一章深入讨论这种用法。

用上面哪种形式解决问题只是编程风格的问题。在这一章，我们主要讨论第三种技术，也就是参数化的多态。泛型通常都会和协议成对出现，因为 Swift 要通过协议明确约束泛型参数的行为，在 协议 这一章我们会看到更多例子，这一章，我们还是把重点放在泛型本身上。

## 泛型类型

我们能编写的一个最普通的函数，就是恒等函数 (**identity function**)。例如，一个原封不动返回参数的函数：

```
func identity<A>(_ value: A) -> A {  
    return value  
}
```

这个恒等函数有一个**泛型参数 (generic parameter)**: 对于任何类型 A, 这个函数的类型就是  $(A) \rightarrow A$ 。但是, 这个函数却有无数多个**具体类型 (concrete type)**, 也就是不带泛型参数的类型, 例如 Int, Bool 或 String。例如, A 是 Int, 这个函数的具体类型就是  $(Int) \rightarrow Int$ , 如果我们让 A 是  $(String \rightarrow Bool)$ , 对应的具体类型就是  $((String) \rightarrow Bool) \rightarrow (String) \rightarrow Bool$ 。

函数和方法并不是唯一的泛型类型。我们还可以有泛型结构体, 泛型类和泛型枚举。例如, 下面就是 Optional 的定义:

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

我们说 Optional 是一个泛型类型。当我们为 Wrapped 选择了一个值, 就得到了一个具体类型, 例如:  $\text{Optional}<\text{Int}>$  或  $\text{Optional}<\text{UIView}>$ 。我们可以把 Optional (不带尖括号的) 当作一个**类型构建器 (type constructor)**: 给它传递一个具体类型 (例如: Int), 它就会创建一个新的具体类型 (例如:  $\text{Optional}<\text{Int}>$ )。

当我们去浏览 Swift 标准库的时候, 就会看到其中包含了很多具体类型, 但也有很多泛型类型 (例如: Array, Dictionary 和 Result)。Array 有一个泛型参数 Element, 这就让我们可以使用任何一个具体类型来创建数组。我们还可以创建自己的泛型类型。例如, 这是一个描述二叉树的枚举:

```
enum BinaryTree<Element> {  
    case leaf  
    indirect case node(Element, l: BinaryTree<Element>, r: BinaryTree<Element>)  
}
```

BinaryTree 是个单泛型参数的泛型类型。为了创建一个具体类型, 我们必须为 Element 设置一个具体类型, 例如 Int:

```
let tree: BinaryTree<Int> = .node(5, l: .leaf, r: .leaf)
```

当我们把泛型类型转换成具体类型的时候，一个泛型参数只能对应一个具体类型。你可能早已经通过创建 Array 熟悉这种做法了。例如，当创建空数组的时候，我们必须提供明确的数组类型，否则 Swift 编译器会抱怨说无法确认数组元素的具体类型：

```
// 必须明确定义变量类型
var emptyArray: [String] = []
```

只要所有的成员都拥有相同的父类型，你就可以把不同具体类型的值放到同一个数组中去。对于持有对象的数组，编译器会自动把元素类型推断为数组中对象们最具体的父类型：

```
// 类型被推断为 [UIView].
let subviews = [UILabel(), UISwitch()]
```

在其他情况下，Swift 要求你明确地指出你想要创建的就是一个 Any 数组：

```
// 类型标注是必须的
let multipleTypes: [Any] = [1, "foo", true]
```

我们会在泛型函数这一节详细讨论 Any。

## 扩展泛型类型

在 BinaryTree 的作用域里，泛型参数 Element 都是可用的。例如，当为 BinaryTree 编写扩展的时候，也可以像使用一个具体类型一样来使用 Element。我们可以给 BinaryTree 添加一个使用 Element 作为参数的便利初始化方法 (convenience initializer)：

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

接下来，是一个把树中所有节点值递归保存为数组并返回的计算属性：

```
extension BinaryTree {  
    var values: [Element] {  
        switch self {  
            case .leaf:  
                return []  
            case let .node(el, left, right):  
                return left.values + [el] + right.values  
        }  
    }  
}
```

当我们访问 `BinaryTree<Int>` 中的 `values` 时，得到的就是一个整数数组：

```
tree.values // [5]
```

我们还可以定义泛型方法。例如，给 `BinaryTree` 添加一个 `map` 方法。这个方法有一个额外的泛型参数 `T`，表示转换方法的返回值，也就是新的 `BinaryTree` 中节点的值的类型。由于这个方法也定义在 `BinaryTree` 的扩展里，我们仍旧可以使用 `Element`：

```
extension BinaryTree {  
    func map<T>(_ transform: (Element) -> T) -> BinaryTree<T> {  
        switch self {  
            case .leaf:  
                return .leaf  
            case let .node(el, left, right):  
                return .node(transform(el),  
                            l: left.map(transform),  
                            r: right.map(transform))  
        }  
    }  
}
```

由于 Element 和 T 都没有协议约束，这里，调用者可以使用任意类型。甚至，我们可以为两个泛型参数选取同一个具体的类型：

```
let incremented: BinaryTree<Int> = tree.map { $0 + 1 }

// node(6, l: BinaryTree<Swift.Int>.leaf, r: BinaryTree<Swift.Int>.leaf)
```

当然，让它们是不同的类型也可以。在下面这个例子中，Element 是 Int，T 是 String：

```
let stringValues: [String] = tree.map { "\($0)" }.values // ["5"]
```

在 Swift 里，很多集合类型都是泛型类型（例如：Array，Set 和 Dictionary）。但是，泛型这项技术本身可不仅仅用在表达集合类型上。它的应用几乎贯穿了整个 Swift 标准库的实现，例如：

- Optional 用泛型参数抽象它包装的类型。
- Result 有两个泛型参数：分别表示成功和失败这两种结果对应的值的类型。
- Unsafe[Mutable]Pointer 用泛型参数表示指针指向的内存的类型。
- Key paths 中使用了泛型表示它们的根类型以及路径的值类型。
- 各种表示范围的类型，使用了泛型表达它们的元素类型 (Bound)。

## 泛型和 Any

泛型和 Any 可以被看作类似的用途，但它们有截然不同的表现。在没有泛型的编程语言里，Any 通常用来实现和泛型同样的效果，但是却缺少了类型安全性。这通常意味着要使用一些运行时特性，例如内省 (introspection) 或动态类型转换，把 Any 这种不确定的类型变成一个确定的具体类型。而泛型不仅能解决绝大部分同样的问题，还能带来编译期类型检查以及提高运行时性能等额外的好处。

阅读代码的时候，泛型可以帮助我们理解一个函数或方法执行的任务。例如，对于下面这个处理数组的 reduce 方法（在这个例子中，我们忽视掉 reduce 是被定义在 Sequence 协议上这一事实）：

```
extension Array {
```

```
func reduce<Result>(_ initial: Result,  
    _ combine: (Result, Element) -> Result  
)
```

无需查看它的实现，我们从它的签名中就能大致描述出它的功能(前提是 `reduce` 有一个合理的实现)：

- 首先，我们知道 `Result` 是 `reduce` 的泛型参数，也是它的返回值类型(这里 `Result` 只是泛型参数的名称。别把它和标准库中的 `Result` 枚举搞混了)。
- 其次，来看参数。`reduce` 接受一个 `Result` 类型的值，以及一个把 `Result` 和 `Element` 合成新的 `Result` 值的方法作为参数。
- 由于返回值的类型是 `Result`，`reduce` 的返回值只能是 `initial` 或者调用 `combine` 得到的结果。我们之所以知道这一点，是因为这个方法并没有构建任意 `Result` 值的能力。
- 如果数组是空的，我们就没有用于合成的 `Element`，这时能返回的只有 `initial`。
- 如果数组不为空，`reduce` 的类型就给它的实现提供了自由：它即可直接返回 `initial` 而根本不去合并数组中的元素(虽然这已经游走在不合理实现的边缘了)，也可以只合并数组中的特定元素(例如，只合并第一个或最后一个元素)，或者，和我们期待的常规实现一样，按顺序合并数组的每一个元素。

要注意的是，`reduce` 是可以有无数种实现方式的。例如，只针对数组中的某些特定元素调用 `combine`。它还可以使用一些运行时类型自省特性，修改一些全局状态，或者发起某些网络请求。但是，这些都不算在合理实现的范畴。并且，由于标准库已经实现了 `reduce` 方法，我们能从它的实现确认什么才属于合理实现的范畴。

现在，我们再来看 `Any` 版本的 `reduce`：

```
extension Array {  
    func reduce(_ initial: Any, _ combine: (Any, Any) -> Any)  
}
```

即使我们假设 `reduce` 有一个合理的实现，单从类型签名看，这个声明传递出来的信息还是太少了。我们根本无法从这个签名中了解到第一个参数和方法返回值的关系，也无法从 `combine`

中了解到它的两个参数究竟是如何进行合并的。其实，我们甚至都不知道 `combine` 合并的是上一次的累计结果和数组中的下一个元素。

从某种意义上说，一个函数或方法的泛型程度越高，它能做的事情就越少。还记得一开始我们提到的那个恒等函数么？这种泛型到没有任何类型约束的函数唯一能做的事情，就是返回它的参数：

```
func identity<A>(_ value: A) -> A {  
    return value  
}
```

在我们的经验里，泛型类型对于源码阅读有很大帮助。更确切地说，只要看到形如 `reduce` 或 `map` 这样的函数或方法，我们不用去猜它们的功能，单就签名中的泛型类型，就已经约束了可能的实现方法。

## 基于泛型的设计

泛型在程序的设计之初就可以派上用场，它们可以在众多类型中剥离开共享的逻辑并生成模板代码。在这一节，我们将重构一个非泛型的网络代码，使用泛型提取出其中公共的功能。

我们将使用扩展给 `URLSession` 添加一个方法：`loadUser`，它从一个 `web` 服务中获取用户信息，并解析成 `User` 对象。首先，我们构建一个 URL 并开始一个网络请求，然后，我们通过 `Codable` 架构（在 编码和解码 这一章我们会讨论到）对下载的数据解码：

```
extension URLSession {  
    func loadUser() async throws -> User {  
        let userURL = webserviceURL.appendingPathComponent("/profile")  
        let (data, _) = try await data(from: userURL)  
        return try JSONDecoder().decode(User.self, from: data)  
    }  
}
```

如果，要用这个方法创建另外一个类型的对象（例如：`BlogPost`），实现的逻辑几乎是一样的。我们可以复制代码并做三处修改：返回类型、访问的 URL，以及 `JSONDecoder.decode(_:from:)`

调用中 `User.self` 类型。但我们有更好的做法，只要把 `User` 替换成一个泛型参数，`load` 方法就能在保留几乎相同实现逻辑的同时，摆脱掉对具体类型的依赖。于此同时，我们还可以替换掉直接进行 JSON 解码的代码，给 `load` 添加另外一个参数：`parse`，让它负责解析 web service 返回的数据，并创建一个 `A` 类型的对象（我们很快就会看到，这样做可以带来极大的灵活性）：

```
extension URLSession {  
    func load<A>(url: URL, parse: (Data) throws -> A)  
        async throws -> A  
    {  
        let (data, _) = try await data(from: url)  
        return try parse(data)  
    }  
}
```

这个新的 `load` 函数接受 `URL` 和 `parse` 函数作为参数，因为它们依赖要访问的目标以及加载的内容。这个重构的策略，和 `map` 以及 内建集合类型 中提到的其它标准库方法使用的，是一样的：

0. 确定一个任务的公共执行模式（从一个 HTTP URL 中加载数据并解析返回结果）。
1. 把执行这个任务的代码模版抽象成一个泛型方法。
2. 允许方法的调用者通过泛型参数和函数参数，在每次调用时泛型方法时，注入需要自定义的内容（在我们的例子中，也就是选择加载特定的 URL 并传递解析对应的返回结果的函数）。

现在，就可以用 `load` 请求不同的 URL 了，通过泛型重构，我们去掉了所有重复的代码：

```
let profileURL = webserviceURL.appendingPathComponent("profile")  
let user = try await URLSession.shared.load(url: profileURL, parse: {  
    try JSONDecoder().decode(User.self, from: $0)  
})  
print(user)  
let postURL = webserviceURL.appendingPathComponent("blog")  
let post = try await URLSession.shared.load(url: postURL, parse: {  
    try JSONDecoder().decode(BlogPost.self, from: $0)  
})
```

```
)  
print(post)
```

由于 URL 和进行数据解码的 parse 函数总是成对出现的，一个更好的做法是把它们封装到一个泛型 Resource 结构体中：

```
struct Resource<A> {  
    let url: URL  
    let parse: (Data) throws -> A  
}
```

于是，之前访问的两个不同的 URL 就可以像下面这样更生动地表示成两个 Resource：

```
let profile = Resource<User>(url: profileURL, parse: {  
    try JSONDecoder().decode(User.self, from: $0)  
})  
let post = Resource<BlogPost>(url: postURL, parse: {  
    try JSONDecoder().decode(BlogPost.self, from: $0)  
})
```

由于 Resource 中的 parse 方法对 JSON 解码这件事情一无所知，我们还可以创建表示图片或者 XML 等其它类型资源的 Resource 对象。但通用性的另一面则意味着，对于每一个表示 JSON 的资源，我们都要在 parse 函数里编写解码的逻辑。为了避免这种重复，当 A 实现了 Decodable 协议的时候（我们将在下一章详细讨论这种条件化协议实现 (conditional conformance) 的细节），我们可以给 Resource 添加一个便利初始化方法。这个初始化方法可以直接使用泛型参数 A 来确定要解码的类型：

```
extension Resource where A: Decodable {  
    init(json url: URL) {  
        self.url = url  
        self.parse = { data in  
            try JSONDecoder().decode(A.self, from: data)  
        }  
    }  
}
```

```
}
```

有了这个方法之后，就可以更精炼的定义表示 JSON 的资源：

```
let profile = Resource<User>(json: profileURL)
let blogPost = Resource<BlogPost>(json: postURL)
```

最后，就可以定义接受 Resource 版本的 load 方法了：

```
extension URLSession {
    func load<A>(_ r: Resource<A>) async throws -> A {
        let (data, _) = try await data(from: r.url)
        return try r.parse(data)
    }
}
```

现在，我们就可以用 profile 资源来调用 load 了：

```
let user = try await URLSession.shared.load(profile)
print(user)
```

创建泛型 Resource 的另外一个良性副作用，是可以让代码更容易测试，因为我们想要测试的逻辑(对象解析 parse 的部分)现在是完全同步的，而且也没有任何依赖。URLSession 上的 load 方法仍旧不好测试，因为它是异步的，而且它对于 URLSession 的依赖使得测试环境变得很复杂。但是至少，我们把异步代码限制在了单一的方法里，而不是让它遍布于每一个获取网络资源的代码中。

## 泛型的静态派发

Swift 支持函数重载，也就是说，可以存在多个同样名字但参数和/或返回类型不同的函数。编译器决定调用哪个函数的过程，被称为**重载解析 (overload resolution)**。对于每个调用点，编译器都遵循一套规则，这套规则可以被归纳为“为给定的输入和结果类型挑选最具体的那个重载”。重载解析总是在编译期间静态地发生；动态运行时的类型信息在其中不扮演任何角色。

下面是一个拥有两个重载的简单的格式化函数：其中一个是对所有 A 的泛型类型，另一个只针对 Int：

```
// 对所有值都有效的泛型变体
func format<A>(_ value: A) -> String {
    String(describing: value)
}

// 对 Int 进行自定义行为的重载
func format(_ value: Int) -> String {
    "+\((value))+"
}
```

并不意外，当我们使用 Int 调用 format 时，Swift 会使用指定 Int 的变体。对其他所有的参数类型，只能选取泛型的版本，因为这是唯一一个类型匹配的版本：

```
format("Hello") // Hello
format(42) // +42+
```

现在，让我们添加另一个泛型函数，它的实现中有调用 format 的部分：

```
func process<B>(_ input: B) -> String {
    let formatted = format(input)
    return "-\((formatted))-"
}
```

正如预期，使用字符串调用 process 时，会使用到泛型的 format 重载。但是当我们使用 Int 调用 process 时，它依然会调用到更不具体的 format 泛型重载：

```
process("Hello") // -Hello-
// !
process(42) // -42-
```

很多人觉得这很奇怪。编译器不是应该已经获取到了所有的信息，并能指出有一个更具体的函数来处理 Int 类型么？对于这个例子中，调用者和被调用函数被作为同一模块的一部分一起编译时，这个假设是正确的。但是对于更一般的情况则不是如此：如果 `process(42)` 的调用发生在实现 `process` 的模块之外，那这个函数就已经是被编译过的了，因为重载解析总是发生在编译期间，所以到底哪个重载应该被调用的决定已经做出了。

Swift 只会使用调用侧本地可用的信息来决定重载解析，这些信息包括了泛型类型和它们的约束。编译器不会依据本地作用域之外的信息来选择重载。

另一种看法，从语义上来说，对每个泛型函数或者类型，只存在唯一一个版本，而这个版本必须能处理所有有效的类型参数。这与 C++ 中的模板有着显著不同，C++ 模板会为每个特定的类型编译单独的实例。但我们强调“语义上来说”，是因为 Swift 实际上为了优化性能，也是将泛型声明编译为专门的特定版本的，但是这是一个实现细节，它并不影响代码的行为。

那么，如果我们想要在运行时基于参数的动态类型，去动态地派发调用 `format` 时，应该怎么做呢？一种可能的方案是手动在调用方法中把参数动态地用 `as?` 转换还原成具体类型：

```
func process2<B>(_ input: B) -> String {  
    let formatted: String  
    if let int = input as? Int {  
        formatted = format(int)  
    } else {  
        formatted = format(input)  
    }  
    return "-\\"(formatted)-"  
}
```

实际上我们在这里实现了我们自己的动态派发机制。这对我们的例子是可用的，但是它很笨拙，而且随着重载的增加，需要手动维护。更好的方法是使 `format` 函数成为一个协议的要求，并将泛型参数约束到这个协议中。这样的方法会使用动态派发，因为协议要求是动态派发的。我们会在下一章的协议中看到更多内容。

## 泛型的工作方式

泛型在编译器里是怎么实现的？为了回答这个问题，让我们先来仔细看看标准库中的 min 函数（这个例子是我们从 Apple 2015 年 WWDC 的优化 Swift 性能中获取的）：

```
func min<T: Comparable>(_ x: T, _ y: T) -> T {  
    return y < x ? y : x  
}
```

对于 min 函数，参数和返回值唯一的约束是所有三个值都必须拥有相同的类型 T，而且 T 必须满足 Comparable。只要满足了这些，T 就可以是任意值，比如 Int、Float、String 或者甚至是定义在其他模块中，编译期间编译器都还不知道的类型。这意味着，编译器无法为这个函数生成代码，因为它缺少了三个必要的信息：

- 类型 T 的值的大小（包括参数和返回值）。
- 如何复制和销毁类型 T 的值（比如，它们是否需要引用计数）。
- 需要调用的指定重载的 < 函数的地址。

Swift 通过向泛型代码引入一层间接层来解决这些问题：

- 使用指针来传递大小未知的函数参数，返回值以及变量。
- 对每个泛型类型参数 T，编译器会把 T 的类型元数据（**type metadata**）传递到函数中去。类型元数据记录包含了类型的\*\*值目击表（value witness table, VWT）\*\*和其他一些东西。VWT 提供了对类型 T 的值的进行基本操作的函数，比如如何复制，移动，或者销毁一个值。对于一些像是 Int 遮掩的个简单值类型，它们可能是无操作或者内存复制，但对于引用类型来说，可能就会包含引用计数的逻辑。VWT 还对类型的内存布局（大小和对齐）进行了记录。
- 对于 T 上的每个约束，编译器都会将一个协议目击表（**protocol witness table, PWT**）传递给函数。PWT 是一个记录有协议要求方法的 vtable。它用来在运行时将函数调用动态地派发到合适的实现去。在我们的例子中，对于满足了 Comparable 的 T，PWT 会提供正确的 < 函数。

在伪代码里，编译器为 min 函数生成的指令看起来类似这样：

```
void func min(_ x: OpaquePointer, _ y: OpaquePointer,
```

```
returnValue: OpaquePointer, // caller-allocated storage
meta_T: TypeMetadata, T_is_Comparable: VTable)
{
    let result = T_is_Comparable.lessThan(y, x, meta_T, T_is_Comparable)
    if result {
        metadata_T.vwt.copyWithInit(returnValue, from: y, meta_T)
    } else {
        metadata_T.vwt.copyWithInit(returnValue, from: x, meta_T)
    }
}
```

因为 T 的内存布局在编译期间是未知的，因此包括返回值在内，所有的值都要通过指针传递。编译器通过 PWT 来调用  $y < x$ ，然后通过 VWT 来把 y 或者 x 复制到存储中并最后返回。

PWT 提供了一种映射关系，来把泛型参数所实现的协议（编译器在静态时就能通过约束得知）和特定类型中实现了这个协议的函数（这只有在运行时才知道）联系起来。实际上，调用协议的方法的唯一途径，就只有通过 PWT。我们无法定义一个没有约束条件的  $\langle T \rangle$  版本的 min 方法，如果没有满足 Comparable 这一条件，就算对应类确实实现了  $<$  函数，编译也无法通过。缺少了 PWT 的信息，编译器将无法定位正确的  $<$  实现，也就无法容许这样的代码。这也是泛型和协议紧密相连的原因：除非你只是在写一些容器类的泛型（比如  $\text{Array}\langle \text{Element} \rangle$  或  $\text{Optional}\langle \text{Wrapped} \rangle$ ），否则无约束的泛型能做的事情确实寥寥无几。我们会在下一章讨论 协议 时再来回顾这个话题。

上面使用指针来传递泛型值的伪代码可能会让你觉得 Swift 会使用装包（boxing）的方式来代表尺寸未知的值，但其实不是这样的。指针的存在，是因为编译器无法为可变大小的值申请 CPU 寄存器的空间，但是这些指针所指向值依然可以是直接生存在栈上的，这不会带来额外的开销。这种做法比 boxing 要高效得多，特别是在处理集合类型的时候：一个泛型值的数组 [T]（当  $T == \text{Int}$  时）所拥有的运行时内存布局和具体类型的数组 [Int] 完全一样。泛型代码使用间接操作元素的方法来使用这个数组，但数组中元素自身依然是紧凑排列，在内存中连续的。这个设计为 CPU 缓存的 缓存局部性优化 提供了最大的可能性。

如果你还想学习更多关于泛型系统工作方式的知识，Swift 编译器工程师 Slava Pestov 和 John McCall 在 2017 年 LLVM 开发者会议上关于这个话题做过一次演讲。

## 泛型特化

相比于非泛型的代码，Swift 对泛型的处理模型添加了无法绕过的间接层代码，这显然会带来运行时的性能损耗。在你只考虑单个函数的调用时，这个损耗并不太需要担心，但是泛型在 Swift 中很普及，把它们全加起来情况可能就会有所不同。标准库到处都在用泛型，包括像是比较两个值这样需要尽可能快速完成的操作。

幸运的是，在许多情况下 Swift 编译器可以使用一种叫做泛型特化 (**generic specialization**) 或者称作单态 (**monomorphization**) 的方式，来完全去除掉这个开销。特化指的是编译器用 Int 这样的具体类型作为泛型参数类型，去把泛型类型或者函数 (比如 `min<T>`) 复制一份。然后，这个特化后的函数可以专门为 Int 进行优化，并去掉所有的间接层了。使用 Int 为 `min<T>` 进行特化后的版本看起来就像这样：

```
func min(_ x: Int, _ y: Int) -> Int {  
    return y < x ? y : x  
}
```

这正是我们会为一个特定的、非泛型的 `min` 函数手写的实现。泛型特化不仅仅节省了虚拟派发的开销，它也开启了更多优化的可能。比如内联 (`inline`)，如果没有特化的话，间接层将会是一个很大的障碍。

当你在编译代码时开启优化 (命令行中使用 `swiftc -O` 或者 SwiftPM 中使用 `swift package build -c release`) 时，优化器将会针对它所能看到的信息，为你的泛型类型和函数创建特化版本。如果你的代码使用了 `Int` 和 `Float` 参数进行 `min` 的调用，那么这两个特化版本将最终存在于二进制中 (而且编译器会在调用侧使用这些特化后的版本)。`min<T>` 这个没有特化的版本也会被保留下来，那些输入类型在编译期间无法确认的调用，将会继续使用泛型版本。

Swift 实际上并没有对编译器会生成什么样的泛型特化代码作出承诺。在 Swift 5.5 中，规则很直接：当优化被打开时，对所有可以看到的调用侧的类型进行特化；否则，就不进行任何特化生成。在今后，优化器可能会引入启发式算法，把一个函数的被

某个特定输入类型进行调用的频率也纳入考虑，以此达到更加平衡的运行时性能、代码尺寸和编译时间优化。

关于特化要记住的是，只有当优化器在编译调用侧时可以看到泛型类型或者函数的完整定义的情况下，特化才会发生。当调用者和被调用者在同一个文件时，这种情况是必然发生的。如果不是，那么你有两种选项来帮助优化器：

0. 打开“全模块优化”(whole module optimization)。在这个编译模式下，当前模块的所有文件都会在一起进行优化。除了泛型特化，全模块优化也会开启其他重要的优化策略。比如说，优化器会识别出那些在整个模块中没有子类的 internal class。因为 internal 修饰符确保了这个类不会被模块外看到，这也就意味着编译器可以把这个类的所有方法由动态派发替换成静态派发。
1. 将泛型函数标记为 @inlinable 以便其他模块使用。这个标记会把函数体也作为模块接口的一部分进行导出，其他模块引用它时，优化器就能看到这些代码。在这种情况下，当调用者进行编译时，虽然包含泛型函数的模块已经被编译过了，但是优化器仍然能生成一个该函数的特化版本并把它放到调用模块中。我们的 min 函数就拥有 @inlinable 标签：

#### @inlinable

```
func min<T: Comparable>(_ x: T, _ y: T) -> T {  
    return y < x ? y : x  
}
```

包括标准库和其他一些像是 SwiftNIO, Swift Collections 和 Swift Algorithms 这样的低层级包，大量地使用了 @inlinable 来确保 API 的特化(以及促成一些其他的优化)。还在试验期间的 -cross-module-optimization 编译器标记会通过把模块中的所有公开 API 都标记为可内联(inlinable)来尝试让这一过程自动化。在 Swift 5.5 中，它还不是一个官方特性，但是对于静态链接到单个二进制文件的模块来说，这种做法几乎没什么缺点。将某个东西进行内联，对于像标准库这样的 ABI 稳定的库来说，是一个巨大的承诺，因为这会让更改实现或者修复 bug 变得不可能。但是如果对于总是会一同进行编译或者重新编译的文件来说，是不需要考虑这些的。

Swift 对泛型的编译模型非常特殊，它作为桥梁，填充了两类不同语言之间的空隙：一头是 C++ 和 Rust 这样的，把所有东西都进行特化的语言；另一头是像 Java 这样，只把泛型作为类

型检查，而在运行时通过包装将它抹去的语言。抛开性能方面的考量，Swift 的方式允许对泛型函数或类型在声明侧和客户端的使用侧进行分别编译。这对 Apple 来说是一个重要的功能，因为它允许 Apple 在它的 SDK 中以二进制的方式搭载 Swift 书写的框架。

## 回顾

通过这一章的内容，我们了解了如何通过泛型实现参数化多态。泛型可以应用在很多地方：我们可以编写泛型类型、泛型函数和泛型下标操作符。我们在整本书中都看到和用到了很多泛型，这是因为标准库中也到处都用到了它们。类似的，在我们自己的代码中，也可以用泛型抽象任何与类型无关的逻辑细节，并以此划分出更加清晰的责任边界。

最后，像是 `Array` 和 `Optional` 这类没有约束条件的泛型已经很有用了，但是将泛型和协议约束一起使用，往往会为我们打开新世界的大门。我们现在就进入下一章，协议。

协议

10

当我们使用泛型类型的时候，通常都会希望对泛型参数有一些约束。协议恰好可以满足我们的这个需要，下面就是一些最常见的例子：

- 通过协议，你可以构建一个依赖数字（而不是诸如 Int, Double 等某个具体的数值类型）或集合类型的算法。这样一来，所有实现了这个协议的类型就都具备了这个新算法提供的能力。
- 通过协议还可以抽象代码接口背后的实现细节，你可以针对协议进行编程，然后让不同的类型实现这个协议。例如，一个使用了 Drawable 协议的画图程序既可以使用 SVG 来渲染图形，也可以使用 Core Graphics。类似的，跨平台的代码可以使用一个 Platform 协议，然后由类似 Linux, macOS 或 iOS 这样的类型来提供具体的实现。
- 你还可以使用协议让代码更具可测试性。更具体地说，当你基于协议而不是一个具体类型来实现某个功能的时候，你可以在产品代码和测试中使用不同的具体实现。

在 Swift 里，一个协议表示一组正式提出的要求 (**requirements**)。例如，Equatable 协议要求实现的类型提供 == 操作符：

```
public protocol Equatable {  
    static func == (lhs: Self, rhs: Self) -> Bool  
}
```

这些要求可以是普通方法、初始化方法、关联类型、属性和继承的协议。大部分协议还有一些无法用 Swift 类型系统表达的要求，例如，Collection 协议就要求切片 (slice) 应该使用和原集合类型相同的索引去访问一个特性元素。语义上的要求也可以包含对于性能的承诺，比如 RandomAccessCollection 就保证要在常数时间内能在任意索引之间跳转。一个类型如果没办法满足协议的语义，那么它就不应该被认为满足这个协议。由于针对协议所编写的算法会依赖这些语义，所以这一点是非常重要的。

让我们先过一遍 Swift 协议的主要特性。然后，这一章里，我们会深入讨论这些特性。

Swift 的协议可以通过扩展的方式，来提供超过它所要求的功能。最简单的例子就是 Equatable：它要求实现的类型提供 == 操作符，然后，它会根据 == 的实现提供 != 操作符的功能：

```
extension Equatable {
```

```
public static func != (lhs: Self, rhs: Self) -> Bool {  
    return !(lhs == rhs)  
}  
}
```

类似地，Sequence 协议要求的方法并不多（它只要求提供一个产生迭代器的方法），但它却可以通过扩展，为自己加入大量可供使用的方法。

只有协议中被要求的方法会被动态派发。也就是说，当你调用变量上的一个要求的方法时，具体哪一个方法会被使用，是在运行时基于变量的动态类型来决定的。协议扩展中的那些非要求的方法，始终会基于变量的静态类型进行静态派发。我们会在下面自定义协议扩展的部分来看看为什么这个区别非常重要。

协议可以通过条件化扩展 (conditional extensions) 添加需要额外约束的 API。例如，在 Sequence 中，只有 Element 实现了 Comparable 的时候，才提供了 max() 方法：

```
extension Sequence where Element: Comparable {  
    /// 返回序列中最大的元素。  
    public func max() -> Element? {  
        return self.max(by: <)  
    }  
}
```

协议可以继承其它协议。例如，Hashable 要求实现的类型必须同时实现 Equatable 协议。类似的，RangeReplaceableCollection 继承自 Collection，而 Collection 继承自 Sequence。换句话说，我们可以构建一个协议层次结构。

另外，协议还可以被组合起来形成新的协议。例如，标准库中的 Codable 就是 Encodable 和 Decodable 协议组合之后的别名。这种组合叫做协议合成 (protocol composition)。

有时，某个协议的实现还依赖于其它协议的实现。例如，当且仅当数组中 Element 类型实现了 Equatable 的时候，对应的数组类型才实现了 Equatable。这叫做条件化实现 (conditional conformance)：Array 实现 Equatable 的条件，就是 Element 实现了 Equatable：

```
extension Array: Equatable where Element: Equatable { ... }
```

协议还可以声明一个或多个关联类型，这些类型充当占位符的角色，相关类型在之后会被用来为协议定义其他的要求。实现了这个协议的类型需要为每一个关联类型指定对应的具体类型。例如，Sequence 定义了一个关联类型 Element，每一个实现了 Sequence 的类型都要定义自己的 Element 类型具体是什么。String 的 Element 类型是 Character；Data 的是 UInt8。

## 协议目击者

在这一节中，我们想展示一下要是没有协议，Swift 看起来会是什么样。反过来，我们希望这个例子能帮你更好地直观感受到协议的工作方式。举个例子，假设 Swift 中没有协议这个特性，如果要给 Array 添加一个判断元素是否全部相等的方法，没有 Equatable 协议的话，我们就只能给这个方法传递一个用于比较的函数：

```
extension Array {
    func allEqual(_ compare: (Element, Element) -> Bool) -> Bool {
        guard let f = first else { return true }
        for el in dropFirst() {
            guard compare(f, el) else { return false }
        }
        return true
    }
}
```

为了让事情更正式一些，我们可以基于 allEqual 的参数创建一个封装，让它更明确的表达相等比较的含义：

```
struct Eq<A> {
    let eq: (A, A) -> Bool
}
```

现在，我们就可以为不同的具体类型（例如：Int）创建不同的 Eq 实例了。我们管这些实例，叫做表示相等判断的显式目击者（**explicit witnesses**）：

```
let eqInt: Eq<Int> = Eq { $0 == $1 }
```

接下来，就可以用 Eq 改造之前的 allEqual 实现了。要注意的是，我们使用了数组中的泛型类型 Element 来表达要比较的所有元素的类型：

```
extension Array {  
    func allEqual(_ compare: Eq<Element>) -> Bool {  
        guard let f = first else { return true }  
        for el in dropFirst() {  
            guard compare.eq(f, el) else { return false }  
        }  
        return true  
    }  
}
```

尽管 Eq 放在这里看上去有点儿晦涩，但正是它为我们呈现了协议在背后的工作方式：当你为一个泛型类型添加了 Equatable 约束之后，只要创建一个对应的具体类型的实例，就会有一个协议目击者传递给它。在 Equatable 的例子中，这个目击者携带的，正是用于比较两个值的 == 操作符。基于要创建的具体类型，编译器会自动传入协议目击者。下面，则是通过协议取代了显式目击者 (explicit witnesses) 的 allEqual 实现：

```
extension Array where Element: Equatable {  
    func allEqual() -> Bool {  
        guard let f = first else { return true }  
        for el in dropFirst() {  
            guard f == el else { return false }  
        }  
        return true  
    }  
}
```

对于满足一个协议的值或者类型(或者对于一个满足协议要求的方法)来说,“目击”这个术语来源于数理逻辑。你可以把它想象成,Int: Equatable通过它的存在,目击了Int满足这个协议的事实(否则就无法编译了)。

我们还可以给Eq添加一个扩展。只要定义了比较两个元素是否相等的方法,就可以实现一个判断两个元素不等的方法:

```
extension Eq {  
    func notEqual(_ l: A, _ r: A) -> Bool {  
        return !eq(l,r)  
    }  
}
```

这和通过扩展给协议添加功能是类似的:由于eq方法是肯定存在的,我们就能基于这个方法构建更多功能。于是,给Equatable添加同样功能的扩展和上面这个notEqual的实现,几乎就是一回事儿。只不过,相比泛型参数A,我们可以使用Self,在协议中它表示实现了协议的类型:

```
extension Equatable {  
    static func notEqual(_ l: Self, _ r: Self) -> Bool {  
        return !(l == r)  
    }  
}
```

而这,正是标准库为Equatable实现!=操作符的方法。

## 条件化协议实现 (Conditional Conformance)

为了实现比较数组的Eq,我们需要一种比较数组中两个元素的方法。这次,我们把eqArray定义成函数,然后把显式目击者传递给它:

```
func eqArray<El>(_ eqElement: Eq<El>) -> Eq<[El]> {  
    return Eq { arr1, arr2 in  
        guard arr1.count == arr2.count else { return false }  
        for i in 0..  
    }
```

```
for (l, r) in zip(arr1, arr2) {  
    guard eqElement.eq(l, r) else { return false }  
}  
return true  
}  
}
```

再一次，eqArray 为我们诠释了 Swift 中条件化协议实现的工作方式。例如，下面是标准库中 Array 对 Equatable 的实现：

```
extension Array: Equatable where Element: Equatable {  
    static func ==(lhs: [Element], rhs: [Element]) -> Bool {  
        ...  
    }  
}
```

这里，给 Element 添加 Equatable 约束，和之前把 eqElement 传递给 eqArray 函数本质上是一样的。在 Array 的扩展里，我们就可以直接使用 == 操作符比较两个元素的值了。而这两种方法最大的区别就是，使用协议约束类型，编译器会自动传递一个协议目击者。

## 协议继承

Swift 还支持协议的继承。例如，实现 Comparable 的类型也一定实现了 Equatable。这叫做提炼 (refining)，Comparable 提炼了 Equatable：

```
public protocol Comparable : Equatable {  
    static func <(lhs: Self, rhs: Self) -> Bool  
    // ...  
}
```

在之前假想的没有协议特性的 Swift 版本里，我们也可以表达这种协议提炼的想法。为此，先为 Comparable 创建一个显式目击者，让它包含 Equatable 的目击者和一个 lessThan 函数：

```
struct Comp<A> {
```

```
let equatable: Eq<A>
let lessThan: (A, A) -> Bool
}
```

同样地，在一个协议继承其他协议时，目击者的工作方式是类似的。在 `Comp` 的扩展里，我们现在可以同时使用 `Eq` 和 `lessThan` 了：

```
extension Comp {
    func greaterThanOrEqual(_ l: A, _ r: A) -> Bool {
        return lessThan(r, l) || equatable.eq(l, r)
    }
}
```

这种传递显式目击者的模式对于我们理解编译器内部对协议的支持很有帮助。而这，也有助于在用协议解决问题卡壳的时候，帮我们找到思路。

但是，（显式传递目击者和使用协议约束类型）这两种做法并不完全相同。同一种类型可以有无数多个显式目击者，但一个类型只能对协议约束的方法提供一份实现。并且，不像显式目击者可以通过参数手动传递，协议目击者的传递是自动的。

如果允许为一个协议提供多份实现，编译器就需要一些方法找到当前环境里最合适的实现。如果这个过程再加上条件化协议实现，就会更加复杂。为了避免这种复杂性，Swift 不允许我们这样做。

(译注：为什么同一种类型可以有无数多个显式目击者呢？举个例子：对于 `Eq<String>` 来说，我们可以有不同的比较字符串相等的逻辑。例如，按字位族比较，按不同的编码比较，或者是其它脑洞大开的比较方法。这些方法，只要创建不同的 `Eq<String>` 对象，并提供不同的函数定义就好了。因此，对 `String` 来说，`Eq` 代表的显式目击者的个数是无限的。并且，我们也可以把不同版本的 `Eq<String>` 作为参数同时传递给一个函数。而对于通过协议约束的类型，则不具有这样的性质，一个实现了 `Equatable` 的类型只能提供一份 `==` 操作符的实现，并且这个实现会被编译器作为协议目击者自动插入到需要 `==` 操作符的地方。)

# 使用协议进行设计

这一节，我们来看个绘图协议的例子。有两个具体类型会实现这个协议：我们可以把图形绘制成 SVG 或渲染到 Apple 自家 Core Graphics 框架的图形上下文 (graphics context) 里。让我们从定义一个要求实现绘制椭圆和矩形接口的协议开始：

```
protocol DrawingContext {  
    mutating func addEllipse(rect: CGRect, fill: UIColor)  
    mutating func addRectangle(rect: CGRect, fill: UIColor)  
}
```

让 CGContext 实现这个协议是易如反掌的事情，直接设置填充颜色并填充对应的区域就好了：

```
extension CGContext: DrawingContext {  
    func addEllipse(rect: CGRect, fill fillColor: UIColor) {  
        setFillColor(fillColor.cgColor)  
        fillEllipse(in: rect)  
    }  
  
    func addRectangle(rect: CGRect, fill fillColor: UIColor) {  
        setFillColor(fillColor.cgColor)  
        fill(rect)  
    }  
}
```

类似的，让 SVG 实现这个协议也不太复杂。我们把矩形转换成一系列 XML 属性，并把 UIColor 转换成一个十六进制字符串：

```
extension SVG: DrawingContext {  
    mutating func addEllipse(rect: CGRect, fill: UIColor) {  
        var attributes: [String:String] = rect.svgEllipseAttributes  
        attributes["fill"] = String(hexColor: fill)  
        append(Node(tag: "ellipse", attributes: attributes))  
    }  
}
```

```
}

mutating func addRectangle(rect: CGRect, fill: UIColor) {
    var attributes: [String:String] = rect.svgAttributes
    attributes["fill"] = String(hexColor: fill)
    append(Node(tag: "rect", attributes: attributes))
}
}
```

(我们没有列出 SVG, CGRect.svgAttributes, CGRect.svgEllipseAttributes 和 String.init(hexColor:) 的定义, 它们在这个例子中并不重要。)

## 协议扩展

Swift 协议中的一个关键特性就是协议扩展 (**protocol extension**)。只要知道了如何绘制椭圆, 就可以添加一个扩展来以某点为圆心绘制圆形。例如, 我们给 DrawingContext 添加下面这样的扩展:

```
extension DrawingContext {
    mutating func addCircle(center: CGPoint, radius: CGFloat, fill: UIColor) {
        let diameter = radius * 2
        let origin = CGPoint(x: center.x - radius, y: center.y - radius)
        let size = CGSize(width: diameter, height: diameter)
        let rect = CGRect(origin: origin, size: size)
        addEllipse(rect: rect.integral, fill: fill)
    }
}
```

为了使用它, 可以给 DrawingContext 再创建一个扩展, 给它添加一个在黄色方块中绘制蓝色圆形的方法:

```
extension DrawingContext {
    mutating func drawSomething() {
        let rect = CGRect(x: 0, y: 0, width: 100, height: 100)
```

```
addRectangle(rect: rect, fill: .yellow)  
let center = CGPoint(x: rect.midX, y: rect.midY)  
addCircle(center: center, radius: 25, fill: .blue)  
}  
}
```

把这个方法定义在 DrawingContext 的扩展里，我们就能通过 SVG 或 CGContext 实例调用它。这是一种贯穿 Swift 标准库实现的做法：只要你实现协议要求的几个少数方法，就可以“免费”收获这个协议通过扩展得到的所有功能。

## 自定义协议扩展

通过扩展给协议添加的方法，并不是协议要求的一部分。在某些情况下，这会导致出乎意料的结果，因为只有被要求的函数会进行动态派发。我们来看看这是什么意思。回到之前的例子中，我们希望使用 SVG 对 circle 内建的支持，也就是说：圆形在 SVG 中就应该被渲染为 `<circle>`，而不是 `<ellipse>`。我们为 SVG 的实现添加一个 `addCircle` 方法：

```
extension SVG {  
    mutating func addCircle(center: CGPoint, radius: CGFloat, fill: UIColor) {  
        let attributes = [  
            "cx": "\((center.x)",  
            "cy": "\((center.y)",  
            "r": "\((radius)",  
            "fill": String(hexColor: fill),  
        ]  
        append(Node(tag: "circle", attributes: attributes))  
    }  
}
```

当我们创建一个 SVG 变量并调用 `addCircle` 方法的时候，商谜爱的呢方法会被调用：

```
var circle = SVG()  
circle.addCircle(center: .zero, radius: 20, fill: .red)  
circle
```

```
/*
<svg>
<circle cx="0.0" cy="0.0" fill="#ff0000" r="20.0"/>
</svg>
*/
```

这里符合预期。但是当我们调用定义在 Drawing 上的 drawSomething() (这个方法里有调用 addCircle) 时，为 SVG 扩展的 addCircle 并不会被调用。在下面的结果里可以看到，SVG 语法中包含的是 <ellipse> 标签而不是我们期望的 <circle>：

```
var drawing = SVG()
drawing.drawSomething()
drawing
/*
<svg>
<rect fill="#ffff00" height="100.0" width="100.0" x="0.0" y="0.0"/>
<ellipse cx="50.0" cy="50.0" fill="#0000ff" rx="25.0" ry="25.0"/>
</svg>
*/
```

这种行为实在是让人惊讶。为了了解发生了什么，我们先把 drawSomething 扩展写成一个泛型的全局函数。它表达的语意和协议扩展中的实现是完全一样的：

```
func drawSomething<D: DrawingContext>(context: inout D) {
    let rect = CGRect(x: 0, y: 0, width: 100, height: 100)
    context.addRectangle(rect: rect, fill: .yellow)
    let center = CGPoint(x: rect.midX, y: rect.midY)
    context.addCircle(center: center, radius: 25, fill: .blue)
}
```

这里，泛型参数 D 是一个实现了 DrawingContext 的类型。这就意味着调用 drawSomething 的时候，编译期就会自动传递一个 DrawingContext 的协议目击者。这个目击者只带有协议约束的所有方法，也就是 addRectangle 和 addEllipse。由于 addCircle 仅是一个定义在扩展里的方法，它并不是这个协议约束的一部分，因此也就不在目击者里了。

这个问题的关键就是只有协议目击者中的方法才能被动态派发到一个具体类型对应的实现，因为只有目击者中的信息在运行时是可用的。在泛型上下文环境中，调用协议中的非约束方法总是会被静态派发到协议扩展中的实现。

结果就是，当从 drawSomething 中调用 addCircle 的时候，调用总是会静态派发到协议扩展中的实现。编译器无法生成必要的动态派发的代码去调用我们给 SVG 扩展中添加的实现。

为了获得动态派发的行为，我们应该让 addCircle 成为协议约束的一部分：

```
protocol DrawingContext {  
    mutating func addEllipse(rect: CGRect, fill: UIColor)  
    mutating func addRectangle(rect: CGRect, fill: UIColor)  
    mutating func addCircle(center: CGPoint, radius: CGFloat, fill: UIColor)  
}
```

这样，在协议扩展中 addCircle 的实现就变成了协议约束的默认实现。有了这个默认实现，之前实现了 DrawingContext 的代码无需任何修改，仍旧可以通过编译。现在，addCircle 成为了协议的一部分之后，它也就成为了协议目击者中的一员，当我们再调用 SVG 对象的 drawSomething 方法时，就会调用到预期的 addCircle 实现了：

```
var drawing2 = SVG()  
drawing2.drawSomething()  
drawing2  
/*  
<svg>  
<rect fill="#ffff00" height="100.0" width="100.0" x="0.0" y="0.0"/>  
<circle cx="50.0" cy="50.0" fill="#0000ff" r="25.0"/>  
</svg>  
*/
```

带有默认实现的协议方法在 Swift 社区中有时也叫做定制点 (**customization point**)。实现协议的类型会收到一份方法的默认实现，并有权决定是否要对其进行覆盖。标准库中这种定制点随处可见，它们被用来提供可以被覆盖的共通默认行为，这类覆盖通常是因为性能原因。举个例子，计算集合中两个元素之间距离的 distance(from:to:) 方法。这个方法默认实现的时间复杂

度是  $O(n)$ ，因为它要遍历两个元素之间的所有位置。由于 `distance(from:to:)` 也是一个定制点，对于像是 `Array` 这样可以提供更有效率实现的类型，就可以重写默认的实现了。

## 协议组合

协议可以被组合在一起，形成一个合成的协议。标准库中的 `Codable` 就是一个合成的协议，它是 `Encodable & Decodable` 的别名：

```
typealias Codable = Decodable & Encodable
```

这就意味着编写下面的函数，我们就能在它的实现里，通过 `value` 同时使用这两个协议约束的方法了：

```
func use Codable<C: Codable>(value: C) {  
    // ...  
}
```

在之前绘图的例子中，我们可能希望渲染一些带有属性的字符串（这些字符串会包含一些表示格式的子区间，例如：粗体、字体和颜色等）。但是，SVG 文件格式并没有提供属性字符串的原生支持（Core Graphics 是可以的）。相比给 `DrawingContext` 添加一个新的方法，我们创建了一个新的协议：

```
protocol AttributedDrawingContext {  
    mutating func draw(_ str: AttributedString, at point: CGPoint)  
}
```

这样，就可以只让 `CGContext` 实现这个协议，而无需给 SVG 添加同样的支持。并且，我们还可以把这两个协议合在一起。例如，给 `DrawContext` 添加一个扩展，要求实现它的类型同样实现了 `AttributedDrawingContext`：

```
extension DrawingContext where Self: AttributedDrawingContext {  
    mutating func drawSomething2() {  
        let size = CGSize(width: 200, height: 100)  
        addRectangle(rect: .init(origin: .zero, size: size), fill: .red)
```

```
var text = AttributedString("Hello")
text.font = UIFont.systemFont(ofSize: 48)
draw(text, at: CGPoint(x: 20, y: 20))
}
}
```

或者，我们也可以写一个带有泛型参数约束的函数。和之前一样，这个函数和扩展中的方法语意上是一样的：

```
func drawSomething2<C: DrawingContext & AttributedDrawingContext>(
    _ c: inout C)
{
    // ...
}
```

所以，协议组合是非常强大的语法工具，通过它，可以给协议添加一些不是所有实现了该协议的类型都支持的操作。

## 协议继承

除了像上一节那样把协议组合起来，协议之间还可以是继承关系。例如，之前定义的 AttributedDrawingContext 还可以写成这样：

```
protocol AttributedDrawingContext: DrawingContext {
    mutating func draw(_ str: AttributedString, at point: CGPoint)
}
```

这个定义就要求实现了 AttributedDrawingContext 的类型，必须同时实现 DrawingContext。

协议继承和协议组合有它们各自的应用场景。例如：Comparable 协议就继承自 Equatable。这意味着我们只要让实现 Comparable 的类型实现 < 操作符，它就可以自动添加诸如 >= 和 <= 操作符的定义了。而在 Codable 的例子中，让 Encodable 继承自 Decodable，或者反之，都是没道理的。但是，定义一个叫做 Codable 的新协议，让它同时继承自 Encodable 和 Decodable 则完全没问题。实际上，typealias Codable = Encodable & Decodable 这种写法

在语法上，和 protocol Codable: Encodable, Decodable {} 是完全一样的。只是别名的写法看上去稍微简洁了一点，它更明确地告诉我们： Codable 仅仅是这两个协议的组合，并没有在组合的结果里添加任何新的方法。

## 关联类型

有些协议需要约束的不仅仅是方法、属性和初始化方法，它们还希望和它相关的一些类型满足特定的条件。这就可以通过关联类型 (associated type) 来实现。

在我们自己的代码里，关联类型并不常用，但标准库中却随处可见。其中，一个最简短的例子就是标准库中的 IteratorProtocol 协议。它有一个关联类型表示迭代的元素，以及一个访问下一个元素的方法：

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

注意 next 在它的返回类型中使用了关联类型。当你要满足这个协议时，你需要为 Element 指定一个类型。下面是一个定义了迭代 Int 元素的例子：

```
struct Counter: IteratorProtocol {  
    typealias Element = Int  
    func next() -> Int? { ... }  
}
```

类型别名 (typealias) 明确地设置了关联类型。我们其实可以把它去掉，编译器也可以从 next 方法的返回值为我们推断出关联类型。这个小例子展示了协议的作者可以依靠关联类型，使用一个或者多个有关的类型来定义协议所要求的方法。具体要使用什么类型，则是到了协议被实现的时候才知道。

关联类型可以有默认值，在关联类型声明的地方用等号就可以定义默认值。Collection 有五个关联类型，而它们之中很多都拥有默认值。比如，关联类型 SubSequence 就拥有一个默认的 Slice<Self>：

```
public protocol Collection: Sequence {  
    associatedtype Iterator = IndexingIterator<Self>  
    associatedtype SubSequence: Collection = Slice<Self>  
    ...  
}
```

带有默认值的关联类型成为了另一个定制点，就像拥有默认实现的方法一样：当一个类型实现 Collection 的时候，你可以选择使用默认实现来减少开发的工作量。而那些为了性能或使用更加方便的集合类型，通常也会覆盖这个类型（例如：String 使用 Substring 作为 SubSequence 类型）。我们在集合类型协议会讨论集合中的所有这些关联类型。

关联类型扮演着占位符的角色，它们在晚一些的时候会被填充上，从这点上来说，关联类型和泛型参数有些类似。它和泛型参数不同的地方在于，到底是谁在哪里来填充这些占位符。泛型参数是由类型的用户在他们的代码中使用这个类型时所提供的，而关联类型则是类型的作者（或者那些正在扩展一个已有类型的人）在实现协议的时候决定的。比方说，数组类型的用户可以选择这个数组的元素类型（例如 Array<Int> 或者 Array<String>），这是因为标准库的作者把它暴露成了一个泛型参数。但用户无法自定义一个数组的迭代器或者自序列的类型；这些类型在标准库中对 Collection 进行实现的时候就已经确定了。

## Self

在协议的定义或者扩展中，Self 指的就是那些实现了这个协议的类型。你可以把 Self 想象成一个隐式的一直存在的关联类型。稍早之前，我们看到过 Equatable 是如何使用 Self 来定义 == 操作符的。下面是来自 BinaryInteger 协议的另一个例子：

```
public protocol BinaryInteger: ... {  
    func quotientAndRemainder(dividingBy rhs: Self)  
        -> (quotient: Self, remainder: Self)  
}
```

这个声明在实现该协议的类型、这个方法的参数的类型、以及返回值中的两个类型之间建立起一种联系，让它们都具有相同的类型。

## 例子：状态恢复

作为本节中的主要例子，我们通过使用带有关联类型的协议，来重新实现一个小型版本的 UIKit 状态恢复机制。在 UIKit 里，状态恢复需要读取视图控制器以及视图的架构，并在 app 挂起的时候将它们的状态序列化。当 App 下一次加载的时候，UIKit 会尝试恢复应用程序的状态。

我们不使用类继承的关系，而是使用协议来表示视图控制器。在真实的实现中，`ViewController` 协议可能会包含很多方法，但为了简单起见，我们让它是一个空的协议：

```
protocol ViewController {}
```

为了恢复一个特定的视图控制器，我们需要能够读写它的状态，我们还希望这个状态实现了 `Codable` 以便进行编码和解码。由于这个状态和具体的视图控制器相关，它就可以定义成一个关联类型：

```
protocol Restorable {
    associatedtype State: Codable
    var state: State { get set }
}
```

为了演示，我们创建一个显示消息的视图控制器。这个视图控制器的状态由一个消息数组以及当前的滚动位置构成，我们把它定义成一个实现了 `Codable` 的内嵌类型：

```
class MessagesVC: ViewController, Restorable {
    typealias State = MessagesState

    struct MessagesState: Codable {
        var messages: [String] = []
        var scrollPosition: CGFloat = 0
    }
    var state: MessagesState = MessagesState()
}
```

实际上，在实现 Restorable 的代码里，我们无需声明 typealias State。编译器足够聪明，它可以通过 state 属性推断出 State 的类型。我们也可以把 MessagesState 重命名成 State，一切仍旧可以正常工作。

## 基于关联类型的条件化协议实现

有些类型只在特定条件下才会实现一个协议。就像之前在条件化协议实现这一节中看到的，只有当数组中元素的类型实现了 Equatable 的时候，Array 才是个 Equatable 的类型。在约束协议实现的条件中，我们也可以使用关联类型的信息。例如，Range 有一个泛型参数 Bound。当且仅当 Bound 实现了 Strideable 协议，并且 Bound 中的 Stride (这是 Strideable 的一个关联类型) 是一个实现了 SignedInteger 协议的时候，Range 才是一个实现了 Sequence 的类型：

```
extension Range: Sequence
where Bound: Strideable, Bound.Stride: SignedInteger
```

要说明的是，编写这么复杂的约束关系，即便是在标准库的实现里，都只是一个例外情况。

在之前假想的 UI 框架里，我们还定义了 SplitViewController，它用两个泛型参数表示它的两个子视图控制器：

```
class SplitViewController<Master: ViewController, Detail: ViewController> {
    var master: Master
    var detail: Detail
    init(master: Master, detail: Detail) {
        self.master = master
        self.detail = detail
    }
}
```

假设分割视图控制器没有它自己的状态，我们就可以把它的两个子视图控制器的状态合并起来作为分割视图控制器的状态。为此，可能我们最自然想到的就是这样：

var state: (Master.State, Detail.State)。但遗憾的是，元组类型没有实现 Codable，也无法通过条件化协议实现为它添加 Codable 支持 (实际上，元组无法实现任何协议；有一个已经被接受了的提案希望让所有元组自动实现 Equatable、Hashable 和 Comparable，但是它还没有被

实现)。因此，我们只能自己编写一个泛型结构体：Pair，然后给它添加 Codable 的条件化协议实现：

```
struct Pair<A, B>: Codable where A: Codable, B: Codable {
    var left: A
    var right: B
    init(_ left: A, _ right: B) {
        self.left = left
        self.right = right
    }
}
```

最后，为了让 SplitViewController 实现 Restorable，我们必须要求 Master 和 Detail 也是实现了 Restorable 的类型。相对于在 SplitViewController 中单独保存一份组合的状态，我们可以直接从它的两个子视图控制器中计算出来。省去了这个局部变量，我们就把状态的修改立即传递到了两个子控制器：

```
extension SplitViewController: Restorable
    where Master: Restorable, Detail: Restorable
{
    var state: Pair<Master.State, Detail.State> {
        get {
            return Pair(master.state, detail.state)
        }
        set {
            master.state = newValue.left
            detail.state = newValue.right
        }
    }
}
```

就像之前在条件化协议实现这一节中提到的，任何类型都只能实现协议一次。这就意味着我们不能再添加诸如 Master 实现了 Restorable，但是 Detail 没有(或者反之)，这样的协议实现条件了。

## 回溯满足协议

Swift 协议的一个主要特性是，它支持以回溯的方式让一个类型满足某个协议。比如，在上面我们让 CGContext 满足了我们的 Drawable 协议。这允许程序员对定义在其他模块中的类型进行功能扩展，并使这些类型可以使用那些依据协议中的约束所衍生出的算法。

然而，有时候这种设计会太过自由。当让一个类型满足某个协议时，我们应该始终确保我们至少是这个类型或者这个协议的所有者（或者同时是两者的所有者）。我们建议你不要让一个不属于你的类型去满足一个不属于你的协议。

例如，在写这篇文章的时候，来自 Core Location 框架的 CLLocationCoordinate2D 并不满足 Codable 协议。尽管我们很容易自己为它添加 Codable 适配，但是如果 Apple 决定让 CLLocationCoordinate2D 实现 Codable 的话，我们的实现就被破坏了。在这种情况下，Apple 可能会选择一种其他的实现方式，结果就是，我们不再能对已有的文件格式进行反序列化。

满足协议的冲突也可能发生在两个分别的包里，比如它们同时让一个类型满足了相同的协议。这种问题曾经发生在 SourceKit-LSP 和 SwiftPM 中，两者同时为 Range 按照不同的约束添加了 Codable 协议的适配。（在 Swift 5 中，标准库让 Range 满足了 Codable。）

想要解决这类潜在问题，我们通常使用一个包装类型，然后在包装类型上添加条件适配。例如，我们可以创建一个包装有 CLLocationCoordinate2D 的结构体，然后让这个包装满足 Codable。我们会在编码和解码一章中看到一个具体例子。

## 存在体

严格来说，在 Swift 中是不能把协议当作一个具体类型来使用的，它们只能用来约束泛型参数。不过，下面的代码还是可以通过编译的（我们使用了上面例子中的 DrawingContext 协议）：

```
let context: DrawingContext = SVG()
```

当我们把协议当作具体类型使用的时候，编译器会在背后为协议创建一个包装类型，叫做**存在体 (existential)**。和目击以及其他一些通用的类型理论一样，这个术语来自于逻辑学：DrawingContext 断言了确实存在某个类型，它满足所需要的条件。（相比之下，泛型创造了

“对于所有”关系：泛型类型 `Array<Element>` 断言，对于所有类型 `Element`，都会有一个相应的 `Array<Element>` 类型)。

事实上，Swift 在处理协议 (和约束) 以及存在体类型时，选择使用了同一语法这一事实，在很大程度上造成了 Swift 程序员在区分它们时所面临的疑惑。特别是存在体其实有一些不太直观的限制，我们会在接下来的部分看到这些限制。“相同或相似的概念应该拥有同样的名字，而不同的东西应该有不同的名字”，这是设计出良好 API 和语言的基本规则，但是这个规则在这里被违反了。另外，存在体这个特性所拥有的轻量级的语法，错误地吸引了开发者对它进行使用，而对这个特性的使用，除非你真的特别需要灵活性，否则其实是不被鼓励的。像是泛型或者不透明类型这类语法上更复杂一些的替代，才是首选的解决方式。

为此，Swift 5.6 为存在体引入了新的 `any P` 语法。旧的使用方式现在依然有效，但是它会在未来的语言版本中被弃用或者移除。在本章的剩余部分，我们都会使用 `any P` 语法来作为提醒：

```
let context: any DrawingContext = SVG()
```

我们可以把 `any DrawingContext` 看作是类似 `Any<DrawingContext>` 的另一种写法 (假若 `Any` 是一个泛型类型的话)，也就是一个带有额外约束的 `Any` 值。当编译器看到 `any DrawingContext` 时，它会创建一个 (四个字长，也就是 64 位系统中的 32 字节的) `Any` 盒子，并在其中为类型实现的每个协议添加一个 8 字节的协议目击者。我们可以通过下面的代码来验证这个结果：

```
MemoryLayout<Any>.size // 32  
MemoryLayout<any DrawingContext>.size // 40
```

这个为协议类型的值创建的盒子，也被叫做**存在体容器 (existential container)**。这是编译器必须要做的事情，因为它需要在编译期确认类型的大小。不同的尺寸的各种类型都有可能满足某个协议，因此将协议包装到一个存在体容器中，就可以创建一个固定尺寸的类型，这样编译器就可以在内存中确定它的布局了。`Any` 容器的四个字中的三个，会用来直接内联存放一些小的值。如果被包装的值大于三个字长，那么编译器就会把它放到堆上，并在盒子里存储一个指针。第四个字会被用来存放一个指针，它指向被包装类型的类型元数据记录。这个类型元数据包含了值目击表，一些诸如创建、销毁或者复制值的基本的操作，就被包括在这个表内。

我们可以看到存在体容器的大小会随着类型实现协议的增多而增长。例如，`Codable` 是 `Encodable` 和 `Decodable` 的组合，所以，我们可以预期 `any Codable` 存在体的大小是 32 字节的 `Any` 容器再加上两个 8 字节的协议目击者：

```
MemoryLayout<any Codable>.size // 48
```

当我们创建一个 `any Codable` 数组的时候，无论数组中元素的具体类型是什么，编译器都可以确认，每个元素的大小是 48 字节。例如，下面这个包含了三个元素的数组，将会占用 144 字节空间：

```
let codables: [any Codable] = [Int(42), Double(42), "fourtytwo"]
```

对于 `codables` 数组中的元素，我们唯一能做的事情，就是调用 `Encodable` 和 `Decodable` 中的 API (这是指不用 `as`, `as?` 或 `is` 等运行时类型转换的条件下)。因为元素的具体类型已经被存在体容器隐藏起来了。绝大多数情况下，这个容器对程序员是不可见的。比如，调用 `type(of:)` 将会返回被包装值的类型，而不是盒子包装本身：

```
type(of: codables[0]) // Int
```

## 对比存在体和泛型

有时，存在体和带有类型约束的泛型参数是可以交换使用的。来看下面这两个函数：

```
func encode1(x: any Encodable) { /* ... */}  
func encode2<E: Encodable>(x: E) { /* ... */}
```

尽管这两个函数都可以用一个实现了 `Encodable` 的类型调用，但它们并不完全相同。对于 `encode1` 来说，编译器会把参数包装到 `any Encodable` 的存在体容器里。这个包装不仅会带来一些性能开销，如果要包装的值过大以至于无法直接存放到存在体里，就还需要开辟额外的内存空间。可能最重要的是，这还会阻止编译器的进一步优化，因为对被包装类型的所有方法调用都只能经过存在体中的协议目击者表完成。

而对于泛型函数，编译器可以为部分或者所有传递给 `encode2` 的参数类型生成一个特化的版本。这些特化版本的性能，和我们手工去针对每个类型书写专门的函数是完全一样的。而相比

encode1，泛型方式实现的缺点，则是更长的编译时间以及更大的二进制程序。要更多地了解泛型特化，可以参考泛型一章。

对大多数代码来说，存在体带来的性能开销不是问题，但当你编写一些性能关键的代码时，就要把这个影响考虑进来。如果你在一个循环里调用上千次上面这两个 encode 函数，就会发现 encode2 要快的多得多。(在简单的情况下，优化器实际上可以把存在体的函数重写为对应的泛型函数，但是这些性能观点对一般情况依然是成立的。)

## 存在体和关联类型

在 Swift 5.6 中，存在体的使用被限制在那些既没有关联类型又不针对 Self 进行要求 (除了 Self 被用在返回类型的情况以外) 的协议里。Swift 开发者们应该对这个长久以来都存在的限制很熟悉了，当看到编译器抱怨 “Protocol ‘P’ can only be used as a generic constraint because it has Self or associated type requirements.” (带有 Self 或关联类型要求的协议 ‘P’ 只能用作泛型约束) 这句话时，你遇到的就是这个限制。这个错误最初的原因是由于编译器实现中缺少了一项功能，现在它已经被修复了。因此，在 Swift 5.7 中，这个限制会被去掉，并且任意协议都可以被当作存在体使用了。

但是还有一个更基本的限制，就算是最好的编译器也无法绕开：有一些协议的要求无论如何都不可能与存在体一同工作。考虑下面的例子：

```
let a: any Equatable = "Alice" // Error in Swift 5.5, legal in Swift 5.7
let b: any Equatable = "Bob"

a == b
// Swift 5.7 error: binary operator '==' cannot be applied
// to two 'Equatable' operands.
```

上面的代码在 Swift 5.7 中也不能编译，因为 == 操作符在它的参数中引用的是 Self：

```
public protocol Equatable {
    static func == (lhs: Self, rhs: Self) -> Bool
}
```

`==` 函数会期望两个参数拥有完全一样的类型。但存在体无法满足这个要求，因为它们会把类型信息抹消掉，这样编译器就不再知道 `a` 和 `b` 其实是同样类型这一事实了。这个例子也说明了其实 `Self` 或者关联类型的限制并没有（也不可能）完全消失，它只是从协议层级向下移动到了协议中独立的每一个协议要求这一层级去。所以，一些特定的协议要求的方法，对存在体类型的值来说依然不可用。

有意思的是，这并不适用于那些在协变位置（比如，作为 `return` 类型）上所使用的 `Self` 或关联类型。下面这段使用了 `FloatingPoint` 存在体的代码在 Swift 5.7 中也是正确的：

```
let number: any FloatingPoint = 1.0
print(number.nextUp) // 1.0000000000000002
```

虽然 `FloatingPoint.nextUp` 返回的是 `Self`，但是它依然可以在一个存在体上调用。这是因为当 `Self` 被用作返回类型时，编译器知道可以把结果再次打包到另一个存在体容器中。表达式 `number.nextUp` 的静态类型是 `any FloatingPoint`。

我们所看到的对于 `Self` 的规则，也同样适用于那些引用了关联类型的协议要求方法。在参数中使用了关联类型的函数，同样不能被用于存在体上：

```
let anyCollection: any Collection = [1, 2, 3] // Legal in Swift 5.7
anyCollection.index(after: 0)
// Swift 5.7 error: member 'index' cannot be used on value of protocol
// type 'Collection'; use a generic constraint instead
```

同时，那些返回一个关联类型的协议要求依然可用：

```
anyCollection.first // 1 (type is Optional<Any>)
```

`Collection.first` 的返回类型是 `Optional<Element>`。由于存在体把具体的元素类型抹除了，所以编译器用 `Any` 对它进行了替换。`anyCollection.first` 的静态类型变成了 `Optional<Any>`。

这只所以能奏效，是因为编译器会把 `Optional<T>` 看作是 `Optional<Any>` 的子类型。编译器对于如何将 `Optional<T>` 解包，又如何将包装里的值重新打包成 `Optional<Any>`

有特别的处理。换句话说，Optional 对它的泛型类型来说是一个协变值。这个协变特性是写死在编译器里的，Optional 是少数几个支持这一特性的类型之一。其他几个类型包括 Array 和 Dictionary (的 Value 类型)。元组在考虑其中的元素类型时，也可以被认为是协变的。Swift 并没有对任意泛型类型支持一般性的协变，比如，`SomeStruct<T>` 并不是 `SomeStruct<Any>` 的子类型。在一般情况下允许这样做会使类型系统变得不健全。

如果我们想要在存在体集合中保持具体的元素类型，应该怎么做呢？假如 Swift 能在存在体上支持 `where` 约束，那就会很容易了：

```
let anyIntCollection: any Collection where Element == Int  
// 不合法语法
```

这还没有被支持，不过很有可能类似的东西会在未来出现。

在 Swift 5.7 中，我们可以通过一些小手段达到类似的效果。我们可以引入一个继承自原协议的新的协议，并为它添加一个我们想要固定的关联类型：

```
protocol IntCollection: Collection where Element == Int {}  
extension Array: IntCollection where Element == Int {}  
let anyIntCollection: any IntCollection = [1, 2, 3]  
anyIntCollection.first // 1 (type is Optional<Int>)
```

我们实际上是把 `where` 约束从存在体上移动到了一个新的编码了同样约束的新协议上。这给了编译器足够的信息，让它知道协议上任意使用 `Element` 的 API 的静态类型肯定会是 `Int`。

## 存在体无法遵守协议

“协议不能遵守它们自己” 在 Swift 社区中被奉为金科玉律。现在，我们已经知道协议和隐式的由编译器生成的协议类型 (也就是存在体) 是不同的东西了。我们可以把这句话说得更精确一些：一个存在体类型不能遵守和“它的”名字相同的那个协议。换句话说，你不能把一个存在体 `any P` 传递给 `func f<T: P>(_ x: P)` 这样的泛型函数。

这一限制的基本原因和前一节中的相同：并非所有的协议方法都在存在体中可用。除了对 Self 或者关联类型有引用的协议要求方法以外，这一规则也对初始化方法和静态方法适用，因为这些方法可以被看作是接受 Self 作为第一个参数的一类特殊函数。

以 Decodable 协议为例，JSONDecoder 类型提供了解码 JSON 值的方法：

```
extension JSONDecoder {  
    func decode<T: Decodable>(_ type: T.Type, from data: Data) throws -> T  
}
```

decode 方法最终会调用 T.init(from:)，它是 Decodable 协议中对于初始化方法的所要求的部分。如果我们允许把存在体类型 (any Decodable).self 传递给 decode，编译器就不知道要初始化什么类型了。下面的代码无法编译，是合情合理的：

```
let json = #"{ "email": "alice@example.com" }"#  
let jsonData = Data(json.utf8) // 32 bytes  
let decoded = try JSONDecoder().decode((any Decodable).self, from: jsonData)  
// 错误: protocol 'Decodable' as a type cannot conform to the protocol itself.
```

这和协议类型变量的限制有所不同，对于变量来说，我们刚才看到可以把这些限制从整个协议的层级下移到单个限制的层级，但是存在体“自我遵守”的限制会始终对整个协议都有效。只有 Error 协议是一个例外。

Error 存在体确实是遵守 Error 协议的。这个特例是写死在编译器里的，它允许 any Error (作为存在体) 可以用作泛型参数的类型，来让泛型参数满足 Error (作为协议) 的约束。如果没有这一特性，那么一些很常见的类型，比如 Result<Int, any Error> (在 Swift 5.6 之前被写作 Result<Int, Error>) 将会导致错误。假如这样，你就必须要指定一个具体的错误类型，比如 Result<Int, URLError>，才能进行编译了。这个编译器魔法之所以能够生效，是因为 Error 协议是一个纯粹的标记协议 (marker protocol)，它没有任何要求，所以不会有无法满足协议要求的问题。

## 不要过早使用存在体

一般来说，除非你需要变量装箱所带来的灵活性（比如在一个集合类型中存储类型相异的值），否则都应该尽量选择使用泛型。存在体会抹去类型信息，而泛型则会保留它。频繁地把值装箱到存在体容器中，再频繁地把它从容器中取出，本身就对性能不利；此外，过早地抹去类型将不可避免地阻止一些编译器优化。

另外，我们已经看到，就算未来的 Swift 版本中能把现在仍存在的对协议类型的限制解除掉，但依然会存在一些基本的限制，让 API 永远无法用于存在体类型值上。为所有的协议解锁存在体，可以消除几乎所有 Swift 开发者都会遇到的困惑。尽管它不可避免地会引入一些全新的让人费解的问题（比如，我们为什么能调用协议的这个方法，而不能调用另一个？），但我们还是认为解锁存在体的使用是一件好事。真正的风险在于，没有经验的开发者会因为存在体的语法非常方便，从而开始到处使用它，最后却在好几个小时或者好几天后才发现实际上应该使用的是带有协议约束的泛型。担心开发者搬石头砸脚，是 Swift 团队在解除存在体限制上犹豫的最大因素。

使用一门类型系统强如 Swift 的静态类型语言的最主要原因之一，就是能给予编译器尽可能多的信息来帮助它完成任务。而不必要的类型消除，恰恰是反其道而行。

## 不透明类型

不透明类型是 API 作者用来在不借助存在体（它会抹消掉类型信息）的前提下把具体返回类型隐藏起来的一种手段。不透明类型对应的语法是 `some MyProtocol`，它代表“某个满足了所列举出的约束的具体类型”的意思。对于用户来说，底层的具体类型被隐藏了，用户只能通过协议所声明的能力（这里是 `MyProtocol`）来操作这个值。到这里为止听起来和存在体很相似，但是和存在体不同的是，类型系统会保留不透明类型（所隐藏）的类型信息。这允许用户在使用不透明类型时能做到一些在存在体上无法做到的事情，比如说使用带有 `Self` 或者关联类型约束的 API。另外，`some MyProtocol` 这个不透明类型是满足了协议的，而我们已经看到过，存在体是不满足协议的。相比于存在体，不透明类型总体来说更利于编译器进行优化。

## 信息隐藏

Apple 在 SwiftUI 中大量使用了不透明类型。如果没有这个特性，书写和阅读 SwiftUI 代码会变得非常不方便，SwiftUI API 的设计也会看起来和现在非常不同。不透明类型在信息隐藏方面有两个作用：(a) 让深层嵌套的泛型更容易使用，以及 (b) 将实现细节隐藏起来。我们会通过一个更大一些的例子来仔细看看这两方面。

让我们来为文本格式化构建一个像 Markdown 那样的修饰语法。我们想要定义一组 API，来把格式属性应用到文本上，并生成修饰后的结果。我们通过定义一个协议作为开始，它表示那些能渲染成修饰文本的富文本：

```
protocol RichText {  
    func render() -> String  
}
```

我们在这里只添加\*\*\*\*粗体\*\*\*和\_斜体 \_ \*文本的支持，但是通过协议，用户可以把这套系统按照他们自己的修饰语法进行扩展。

对于没有任何格式的纯文本，我们可以让 String 满足 RichText：

```
extension String: RichText {  
    func render() -> String { self }  
}
```

接下来，我们创建一个代表粗体文本的类型。这个结构体存储一个 RichText 值，并在渲染时为它添加正确的修饰语法：

```
struct Bold<Text: RichText>: RichText {  
    var text: Text  
    func render() -> String {  
        "***\\(text.render())***"  
    }  
}
```

注意，我们在这里选择了让结构体根据它的输入文本进行泛型。这并不是唯一的选择（我们甚至可以使用存在体来代替），但考虑到避免过早类型消除等一般规则，这是合适的做法。

表示斜体文本的类型看起来几乎一样：

```
struct Italic<Text: RichText>: RichText {  
    var text: Text  
    func render() -> String {
```

```
"_\\(text.render())_"
}
}
```

目前，我们可以这样来使用这些类型：

```
Bold(text: "bold").render() // **bold**
Bold(text: Italic(text: "bold and italic")).render() // **_bold and italic_**
```

为了给具有不同格式的文本建模，我们需要一种方法来组合多个文本片段。我们可以定义一个结构体 `Concat`，来把两个 `RichText` 值连接起来：

```
struct Concat<Text1: RichText, Text2: RichText>: RichText {
    var a: Text1
    var b: Text2
    func render() -> String {
        "(a.render())\\(b.render())"
    }
}
```

这可以奏效，但是语法难以操作：想要连接多于两个片段的话，就需要进行嵌套。通过向协议添加一些 SwiftUI 风格的“修饰符”(modifiers)，可以让 API 更流畅：

```
extension RichText {
    func bold() -> Bold<Self> {
        Bold(text: self)
    }
    func italic() -> Italic<Self> {
        Italic(text: self)
    }
    func appending<Other: RichText>(_ other: Other) -> Concat<Self, Other> {
        Concat(a: self, b: other)
    }
}
```

这是新 API 的使用方式：

```
let text = "Hello,"  
.bold()  
.Appending(" ")  
.Appending(  
    "world!".italic()  
)  
text.render() // **Hello,** _world!
```

很好！如果你看过 SwiftUI 代码，这些对你来说应该很熟悉了。然而美中不足的是，`text` 值的类型是 `Concat<Concat<Bold<String>, String>, Italic<String>>`。这揭示了深度嵌套的泛型类型的两个常见问题：

0. **\*\*嵌套的泛型很快会变得无法维护。\*\***类型推断在一定程度上可以帮助解决这个问题，但是函数声明却需要明确的返回类型。想象一下一个构建一些格式化文本并将它返回的函数，你不仅需要在函数签名里写出完整的让用户难以理解的返回类型，而且每次你修改返回值的结构时（比如添加另一个文本片段），你都会需要更新这个类型。
1. **\*\*类型暴露了实现细节。\*\***—但你把一个类型签名当作公开 API 发布，你想要再进行修改，就会破坏客户端的实现。如果一个库在每次更新一些文本格式时，都要强制它的用户也去更新自己的代码，那这个库应该不会受到欢迎。此外，一个只希望用户通过 `RichText` 接口进行交互的文本标记库本身就完全不应该把像是 `Bold` 或者 `Concat` 这样的类型泄漏给调用者。

不透明类型通过让我们把一个像 `Concat<Self, Other>` 的具体类型，替换成 `some RichText` 来解决这些问题。让我们来为刚才写的文本修饰器这么做吧（实现的方式保持不变）：

```
extension RichText {  
    func bold() -> some RichText {  
        Bold(text: self)  
    }  
    func italic() -> some RichText {  
        Italic(text: self)  
    }  
}
```

```
func appending<Other: RichText>(_ other: Other) -> some RichText {  
    Concat(a: self, b: other)  
}  
}
```

调用侧也不发生改变：

```
let text = "Hello,"  
.bold()  
.appending(" ")  
.appending("world!".italic())  
text.render() // **Hello,** _world!
```

不同之处在于，`text` 的具体类型现在被隐藏了，就好像 API 返回的是一个存在体一样。新的函数签名更好地反映出我们作为 `RichText` 库的作者，对所提供的内容向用户所做出的约定。因为调用者现在只能访问不透明类型所指定的能力（这里是 `RichText`），用户也不需要知道内部类型的细节了。就算底层的具体类型发生了变化，也没有人需要去更新他们的代码了。

## 不透明类型的规则

不透明类型的正式的规则概述如下：

1. 不透明类型可以出现在函数的返回类型，属性/变量，或者下标中。因为它们是专门关于输出类型的，所以它们也被叫做不透明的结果类型。

Swift 5.7 将会把 `some P` 的语法扩展到函数参数中去，让它作为一个轻量级的替代泛型参数的方式。不过，“不透明函数参数”和“不透明结果类型”除了名字不同外，还有一个很重要的语义上的区别。当 `some P` 出现在函数参数中时，是调用侧来决定究竟把什么类型传递给函数，而隐藏在不透明结果类型之后的具体类型，则是由函数的作者来选定的。

2. 通常来说约束是某个协议，但它也可以是一个类（如 `some UIView` 表示任意 `UIView` 的子类）或者所个约束的组合（`some AnyObject & Encodable`）。

\*\*3. 不透明类型的函数必须在所有代码路径返回相同的类型。\*\*比如，下面这个 appending 的替代实现会在添加文本为空时简单返回 self，它是无法编译的：

```
extension RichText {  
    // 错误：函数声明了一个不透明类型，但是函数体的返回语句底层类型不匹配。  
    func appending2<Other: RichText>(_ other: Other) -> some RichText {  
        if other.render().isEmpty {  
            return self  
        } else {  
            return Concat(a: self, b: other)  
        }  
    }  
}
```

编译器会保证所有的代码路径都返回相同的类型。虽然在函数体外部，具体的类型是不可知的，但是类型系统会把“appending2 的返回类型”作为单个永不改变的符号。这和处理存在体的方式存在很大不同。

你可能会奇怪，那 SwiftUI 是如何允许在条件语句的不同分支中返回不同 view 类型的呢？比如，这个例子是完全合法的：

```
struct ContentView: View {  
    var rounded: Bool  
    var body: some View {  
        if rounded {  
            Circle()  
        } else {  
            Rectangle()  
        }  
    }  
}
```

这里能工作的原因，是 body 属性可以隐性地变成一个 result builder 函数。注意我们没有（也不能）在 if/else 分支里写 return 语句。这个不透明的 some View 的具体类型既不是 Circle 也

不是 Rectangle，而是 \_ConditionalContent<Circle, Rectangle>。如果想要更多地了解 result builder，可以参看方法一章中的相关部分。

\*\*4. 不透明类型的函数必须在每次调用时都返回相同的具体类型。\*\*编译器“知晓”而且它也会利用这一点。比如，我们可以多次调用一个返回 some BinaryInteger 的函数，并把返回值都加起来：

```
func randomNumber() -> some BinaryInteger {  
    Int16.random(in: 1...20)  
}  
  
let a = randomNumber() // 16  
let b = randomNumber() // 10  
a + b // 26
```

如果 randomNumber 返回的是 any BinaryInteger 存在体的话，上面的代码将导致错误，因为 + 操作符只被定义在了相同类型的操作数上，而存在体可以把任意满足协议的类型装箱。

\*\*5. 你可以通过动态转换还原出具体类型。\*\*不透明类型只被静态类型系统所知，在运行时它们不做任何表达。在一个不透明值上调用 type(of:) 会返回它实际的具体类型。要测试一个不透明值是否是某个具体类型，你可以使用 is 或者 as? 操作符：

```
if let d = randomNumber() as? Int16 {  
    print("\(d) is an Int16")  
} else {  
    print("It's some other type")  
} /*end*/  
// 3 is an Int16
```

## 类型消除器

尽管我们无法为带有 Self 或关联类型约束的协议创建存在体，但我们可以编写一个执行类似功能的函数，叫做：类型消除器 (type erasers)。

我们已经看到存在体就是一种类型消除的形式了。由于目前的限制，存在体只对那些没有 Self 或关联类型要求的协议起作用，有时候我们会想要编写一个类型，它能够执行类型消除的任务，同时不受到存在体那样的使用限制。我们把这些类型叫做(手动)类型消除器。就算是 Swift 5.7 中更加灵活的存在体，也不能覆盖所有的使用场景(即那些需要保留一个或多个关联类型的类型消除)，所以至少在一段时间里，编写手动的类型消除依然是有价值的。

例如，考虑下面这个表达式：

```
let seq = [1, 2, 3].lazy.filter { $0 > 1 }.map { $0 * 2 }
```

它的类型是 LazyMapSequence<LazyFilterSequence<[Int>, Int>。随着串联更多的操作，这个声明就会更加复杂。有时，我们会想要消除掉结果中类型的细节，只需要一个 Sequence，且它包含 Int 元素。通过不透明类型(some Sequence)或者(Swift 5.7 中的)存在体，我们可以表达前一个约束(也就是 Sequence)，但是无法表达后一个(Int 元素)。不过很方便，AnySequence 可以让我们隐藏掉底层类型：

```
let anySeq = AnySequence(seq)
```

anySeq 的类型就是 AnySequence<Int>。尽管这看上去简单多了，并且用起来也和一个序列一样，但这样做也是有代价的：AnySequence 引入了额外的一层间接性，它比直接使用被隐藏的原始类型慢得多。

标准库为很多协议都提供了类型消除器，例如：AnyCollection 和 AnyHashable。这一节接下来的内容里，我们给这一章之前定义的 Restorable 协议实现一个简单的类型消除器。

Restorable 的定义如下：

```
protocol Restorable {
    associatedtype State: Codable
    var state: State { get set }
}
```

作为第一次尝试，我们可能会写一个像下面这样的 AnyRestorable。但它并不能完成任务。因为泛型参数 R 还是直接暴露了要隐藏的协议，这个版本的 AnyRestorable 就跟形同虚设的一样：

```
struct AnyRestorable<R: Restorable> {
```

```
var restorable: R  
}
```

实际上，我们希望 AnyRestorable 的泛型参数反映的应该是 State。在类型消除器中，这是一个常见模式：它们将具体的满足协议的类型消除掉，而保留下对类型消除器的用户来说很关键的关联类型。举例来说，AnyCollection<Element> 会保留集合的元素类型，因为大部分用户需要这个类型来进行其他操作。而其他的各种 Collection 的实现类型，包括其他的关联类型，都被消除了。

为了让 AnyRestorable 实现 Restorable，我们还需要提供 state 属性。为此，我们将使用和标准库同样的实现方法：它使用了三个类来实现一个类型消除器。首先，我们创建一个实现了 Restorable 的类：AnyRestorableBoxBase，它仅仅只是 State 的泛型，而不对 Restorable 进行泛型约束。我们通过将 state 实现为 fatalError 来满足 Restorable。这个类是对实现是私有的，它永远不会被实例化：

```
class AnyRestorableBoxBase<State: Codable>: Restorable {  
    internal init() {}  
    public var state: State {  
        get { fatalError() }  
        set { fatalError() }  
    }  
}
```

接下来，创建一个 AnyRestorableBoxBase 的派生类，让它带有一个实现了 Restorable 的泛型参数 R。这里，让类型消除器得以工作的伎俩，就是限制了 AnyRestorableBoxBase 的泛型参数，让它和基类的 R.State 是同一个类型：

```
class AnyRestorableBox<R: Restorable>: AnyRestorableBoxBase<R.State> {  
    var r: R  
    init(_ r: R) {  
        self.r = r  
    }  
  
    override var state: R.State {
```

```
get { r.state }
set { r.state = newValue }
}

}
```

这种继承关系意味着，我们可以创建一个 AnyRestorableBox 实例，但把它当成一个 AnyRestorableBoxBase 来用，而后者仅仅只暴露了泛型的 State 参数。由于 AnyRestorableBoxBase 实现了 Restorable，进而，它又可以直接当成 Restorable 来用。最后，我们创建一个包装结构体 AnyRestorable，用它把 AnyRestorableBox 藏起来：

```
struct AnyRestorable<State: Codable>: Restorable {
    private let box: AnyRestorableBoxBase<State>
    init<R>(_ r: R) where R: Restorable, R.State == State {
        self.box = AnyRestorableBox(r)
    }

    var state: State {
        get { return box.state }
        set { box.state = newValue }
    }
}
```

这个结构体是整个类型消除器唯一公开可见的部分。它和标准库中的 AnySequence、AnyIterator 等类型是直接对应的。这个结构体只对 State 做了泛型，它的初始化方法则限制了 R: Restorable 泛型。这个初始化方法保证了泛型 State 参数和即将被消除的 Restorable 的 State 类型具有相等关系。

总体来说，当编写一个类型消除器的时候，我们要确保它包含了协议约束的所有方法。尽管编译器可以在这件事情上帮我们一把，但它会放过那些带有默认实现的协议方法。在类型消除器里，我们不能依赖这些默认实现，而是要始终把方法调用转发到被隐藏的原始类型上，因为它们都是有可能被定制的。

## 使用解锁后的存在体手动实现类型消除

虽然 Swift 5.7 里更加强大的存在体不会让手动类型消除的方法立刻过时，但它提供了一种新的方式来编写这些包装，让类型消除更简洁，也更容易让编译器去优化它们。我们从编写一个直接的 AnyRestorable 包装类型开始，现在它只是直接存储一个存在体：

```
struct AnyRestorable<State: Codable> {
    private var _value: any Restorable
    init<R: Restorable>(_ value: R) where R.State == State {
        self._value = value
    }
}
```

我们会在为 Restorable 提供实现的时候遇到问题。因为我们已经把 State 类型和 Restorable 存在体之间的关系抹消掉了，所以这里就没有简单的方法能实现那些参照了 State 的协议要求了。这里可以取巧用一个协议扩展作为桥梁，来连接有类型的世界和类型抹消后的世界：

```
private extension Restorable {
    // 用存在体可以访问的方式转发协议的要求
    func _getStateThunk<_State>() -> _State {
        assert(_State.self == State.self)
        return unsafeBitCast(state, to: _State.self)
    }
    mutating func _setStateThunk<_State>(newValue: _State) {
        assert(_State.self == State.self)
        state = unsafeBitCast(newValue, to: State.self)
    }
}
```

一般来说，每有一个存在体不能直接访问到的协议要求，你就需要在这个扩展中包括一个对应的方法。对于协议中的可变属性要求，我们还需要拆分成 getter 和 setter 两个方法，因为这些方法必须是泛型的，但属性并没有泛型参数。现在，协议可以通过转发给这些新的辅助方法来进行实现了：

```
extension AnyRestorable: Restorable {
    var state: State {
```

```
get { _value._getStateThunk() }
set { _value._setStateThunk(newValue: newValue) }
}
}
```

这段代码的聪明之处在于，它利用了协议扩展中协议的关联类型（以及 `Self`）是可用的，以及所有引用这些类型的 API 都是可以调用的这一事实。在 `State`（关联类型）和 `_State`（泛型参数）之间的按位转换（bitcasting）是安全的，因为我们的类型消除包装的初始化方法保证了只有符合要求的类型会被使用。

## 回顾

Swift 的协议可以和泛型一起，让我们编写出可重用、可扩展的代码。而诸如协议扩展、条件化协议实现以及关联类型等进阶特性则可以让我们构建更复杂的接口。

如果你仔细想想，标准库和其他一些库中使用协议的目的，和大多数 app 开发者是不同的。Collection 协议的继承层级是被精心设计过的，它提供了一层抽象，让程序员能写出有意义的泛型算法。每个协议都必须通过以下方式证明它的存在：(a) 具有实现一类新算法的协议要求和语义；(b) 与其他协议很好地结合，以实现更多的算法；(c) 允许尽可能多的类型来实现它，而不能要求过于严苛。但并不是每个协议都要这样。用一个简单的协议来抽象出依赖关系，使你的代码更易测试，这也没有错。

另外，需要牢记在心，和其他抽象方式一样，协议可以简化代码，但是它们也可以有相反的效果：我们肯定见过（也写过）由于过度使用协议而变得难以理解的代码。在很多情况下，可以用值或者函数以更简单的方式编写代码。想要找到正确的平衡需要一定经验。当要把常规函数基于协议重写的时候，明确地使用协议目击可以作为第一步。

# 集合类型协议

11

在内建集合类型这一章，我们提到了 Swift 中的集合类型，包括：Array, Dictionary 和 Set。它们并非空中楼阁，而是建立在一系列由 Swift 标准库提供的用来处理元素序列的抽象之上。这一章我们将讨论 Sequence 和 Collection 协议，它们构成了这套集合类型模型的基石。我们会研究这些协议是如何工作的，它们为什么要这样工作，以及如何自定义序列和集合类型等话题。

为了更好地理解集合类型协议，下面这张图展示了它们之间的继承关系：

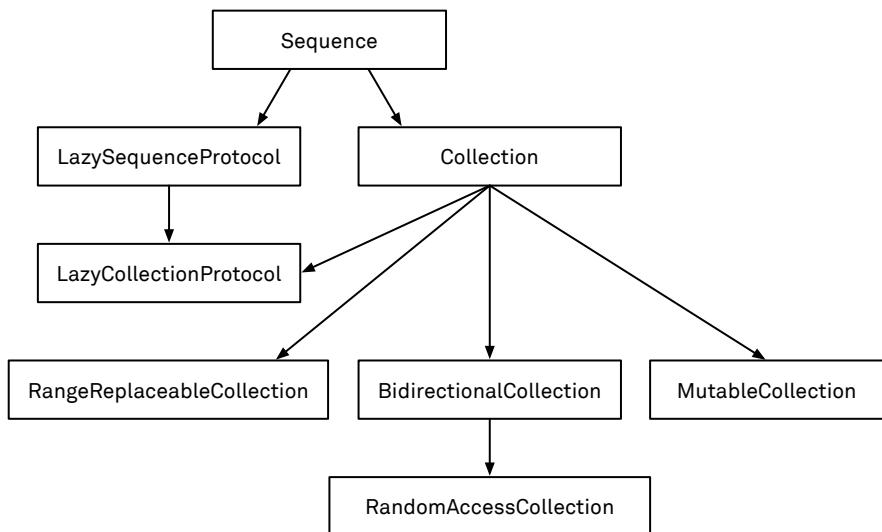


Figure 11.1: 标准库中的集合协议架构

- **Sequence** 提供了迭代的能力。它允许你创建一个迭代器，但对于序列是否只能单次遍历（例如读取标准输入的内容）或者支持多次遍历（例如遍历一个数组），则不提供任何保障。
- **Collection** 扩展了 Sequence。它不仅是一个可以多次遍历的序列，还允许你通过索引访问其中的元素。并且，Collection 还通过 SubSequence 提供了集合切片的能力，而这个切片自身也是一个集合。

- **MutableCollection** 提供了可以在常数时间内，通过下标修改集合元素的能力。但它不允许向集合中添加删除元素。
- **RangeReplaceableCollection** 提供了替换集合中一个连续区间的元素的能力。通过扩展，这个能力还衍生出了诸如 append 和 remove 等方法。很多可变集合类型 (mutable collections) 都可以进行区间内容替换，但其中也有例外。例如，最常用的 Set 和 Dictionary 就不支持这个操作，而 Array 和 String 则没问题。
- **BidirectionalCollection** 添加了从集合尾部向集合头部遍历的能力。显然，我们无法像这样遍历一个 Dictionary，但“逆着”遍历一个字符串则完全没问题。对于某些算法来说，逆向遍历是一个至关重要的操作。
- **RandomAccessCollection** 扩展了 **BidirectionalCollection**，添加了更有效率的索引计算能力：它要求计算索引之间的距离或移动索引位置都是常数时间的操作。例如：Array 就是一个随机访问集合，但字符串就不是，因为计算两个字符之间距离是一个线性时间的操作。
- **LazySequenceProtocol** 定义了一个只有在开始遍历时才计算其中元素的序列。在函数式风格编写的算法里，它很常用：你可以接受一个无穷序列，从中筛选元素，然后读取结果中的前几个记录。这个过程并不会因为需要计算结果集之外的无穷多个元素而耗尽资源。
- **LazyCollectionProtocol** 和 **LazySequenceProtocol** 是类似的，只是它用于定义有相同行为特性的集合类型。

在这一章，我们将深入探索这些协议类型的细节。当你自己实现某些算法的时候，应该时刻把上面协议继承的架构关系考虑在内：只要你的算法基于这个架构之中的一些协议实现，就会有更多的类型可以和这个算法搭配在一起工作。

## 序列

Sequence 协议是集合类型结构中的基础。一个序列 (sequence) 代表的是一系列类型相同的值，你可以对这些值进行迭代。遍历一个序列最简单的方式是使用 for 循环：

```
for element in someSequence {  
    doSomething(with: element)
```

```
}
```

实际上，Sequence 依赖于上面这种步进式迭代元素的能力，为实现它的类型提供了大量有用的方法。在上一章，我们已经看到过不少例子了，每当遇到一个需要顺序访问一系列值的常用操作，你都应该考虑通过 Sequence 来实现它。

满足 Sequence 协议的要求十分简单，唯一要做的就是提供一个返回迭代器 (iterator) 的 `makeIterator()` 方法：

```
protocol Sequence {
    associatedtype Element
    associatedtype Iterator: IteratorProtocol

    func makeIterator() -> Iterator
    // ...
}
```

从这个 (简化后的) Sequence 定义中，我们能发现两件事情：一个是 Sequence 有一个关联类型 Element，另一个是它有一个创建迭代器的方法。所以，我们先来仔细看看迭代器是什么。

## 迭代器

序列通过创建一个迭代器来提供对元素的访问。迭代器每次产生序列中的一个值，并对遍历状态进行管理。在 `IteratorProtocol` 中，唯一的一个方法是 `next()`，这个方法需要在每次被调用时返回序列中的下一个值。当序列被耗尽时，`next()` 应该返回 `nil`：

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

大部分协议的名称不需要用 `Protocol` 结尾，但标准库中有一些例外：  
[Async]IteratorProtocol, StringProtocol, Keyed[En|De]CodingContainerProtocol

以及 Lazy[Collection|Sequence]Protocol。这主要是为了避免与协议的关联类型或实现了协议的具体类型发生命名冲突，因为这些类型也会使用不带有后缀的名称。

API 设计指南 建议：根据协议承担的角色，协议名称应该是个名词，或带有 -able, -ible 或 -ing 后缀。

关联类型 Element 指定了迭代器产生的值的类型。比如 String 的迭代器的元素类型是 Character。通过扩展，迭代器同时也定义了它对应的序列的元素类型。这是通过给 Sequence 的关联类型 Iterator 定义了一个类型约束实现的：Iterator.Element == Element，它确保了序列和迭代器中的元素类型是一致的：

```
protocol Sequence {  
    associatedtype Element  
    associatedtype Iterator: IteratorProtocol  
    where Iterator.Element == Element  
    // ...  
}
```

一般来说，只有在实现一个自定义序列类型的时候，才需要关心迭代器。除此之外，你几乎不会直接去使用它，因为 for 循环才是我们遍历序列常用的方式。实际上，for 正是通过迭代器实现的：编译器会为序列创建一个新的迭代器，并且不断调用迭代器的 next 方法，直到它返回 nil 为止。从本质上说，我们在这一章开始看到的 for 循环，其实是下面这段代码的一种简写形式：

```
var iterator = someSequence.makeIterator()  
while let element = iterator.next() {  
    doSomething(with: element)  
}
```

迭代器是单向结构，它只能按照增加的方向前进，而不能倒退或者重置。为了重新迭代，你只能新建一个迭代器（实际上，这也正是 Sequence 允许通过 makeIterator() 完成的操作）。虽然大部分的迭代器的 next() 都只产生有限数量的元素，并最终会返回 nil，但是你也完全可以创建一个无限的序列。实际上，除了那种一上来就返回 nil 的迭代器，最简单的情况应该是一个不断返回同样值的迭代器了：

```
struct ConstantIterator: IteratorProtocol {
```

```
typealias Element = Int
mutating func next() -> Int? {
    1
}
}
```

其实，在上面的代码里，我们并不用显式使用 typealias 指定 Element 的类型（不过通常可以把这种做法看作文档，它有助于理解代码，特别是大型协议中这点尤为明显）。即便去掉它，编译器也可以从 next() 的返回值中推断出 Element 的类型：

```
struct ConstantIterator: IteratorProtocol {
    mutating func next() -> Int? {
        1
    }
}
```

注意这里 next() 被标记为了 mutating。对于这个简单的例子来说，我们的迭代器不包含任何可变状态，所以它并不是必须的。不过在实践中，迭代器的本质是存在状态的。几乎所有有意义的迭代器都会要求可变状态，这样它们才能够管理在序列中的当前位置。

现在，我们可以创建一个 ConstantIterator 的实例，并使用 while 循环来对它产生的序列进行迭代，这将会打印无穷的数字 1：

```
var iterator = ConstantIterator()
while let x = iterator.next() {
    print(x)
}
```

来看个更有意义的例子。FibsIterator 可以产生一个斐波那契序列。它记录接下来的两个数字，并作为状态存储。而 next 方法则返回第一个数，并在接下来的调用中更新存储的状态。和之前的例子一样，这个迭代器也将产生一个“无限”序列，它将持续累加数字，直到程序因为所得到的数字发生类型溢出而崩溃：

```
struct FibsIterator: IteratorProtocol {
```

```
var state = (0, 1)

mutating func next() -> Int? {
    let upcomingNumber = state.0
    state = (state.1, state.0 + state.1)
    return upcomingNumber
}

}
```

## 遵守序列协议

关于(有限)序列, HTML 文档中所有节点的序列会是一个更有用的例子。在枚举一章中, 我们定义了一个表示 HTML 节点的枚举类型:

```
enum Node: Hashable {
    case text(String)
    indirect case element(
        name: String,
        attributes: [String: String] = [:],
        children: Node = .fragment([]))
    case fragment([Node])
}
```

下面是 Node 的一个例子(在枚举章节中, 我们还定义了一系列辅助方法来创建节点, 不过为了简单, 这里就都省略了):

```
let header: Node = .element(name: "h1", children: .fragment([
    .text("Hello "),
    .element(name: "em", children: .text("World"))
]))
```

我们可以通过创建一个自定义的迭代器来让 Node 类型遵守 Sequence。在迭代器中, 我们需要追踪所有的“剩余”节点。我们首先会把我们的 HTML 树的根节点加进来。然后迭代器移除并返回剩余节点数组中的第一个元素。如果这个元素拥有子节点, 那么它会把这些子节点添加到剩余数组中:

```
struct Nodelterator: IteratorProtocol {  
    var remaining: [Node]  
  
    mutating func next() -> Node? {  
        guard !remaining.isEmpty else { return nil }  
        let result = remaining.removeFirst()  
        switch result {  
            case .text(_):  
                break  
            case .element(name: _, attributes: _, children: let children):  
                remaining.append(children)  
            case .fragment(let elements):  
                remaining.append(contentsOf: elements)  
        }  
        return result  
    }  
}
```

这并不是迭代一棵节点树的唯一方式。这个迭代器执行的是对树的广度优先遍历，但你也可以使用任何其他的树的遍历算法。不过，你只能让 `Node` 满足 `Sequence` 一次，所以你需要选择最合适的做法。

让 `Node` 满足 `Sequence` 现在就很简单了，只需要创建一个 `Nodelterator`:

```
extension Node: Sequence {  
    func makeIterator() -> Nodelterator {  
        Nodelterator(remaining: [self])  
    }  
}
```

仅只是满足 `Sequence`，就已经为 `Node` 带来非常多方便的方法了。比如，我们可以使用 `contains(where:)` 来检查文档是否含有强调文本 (`<em>` 标签) 的节点了：

```
header.contains(where: { node in
    guard case .element(name: "em", _, _) = node else { return false }
    return true
})
// true
```

或者，我们可以使用 compactMap 来从文档中把所有文本提取出来：

```
header.compactMap { node -> String? in
    guard case let .text(t) = node else { return nil }
    return t
}
// ["Hello ", "World"]
```

还有其他很多有用的操作：我们可以使用 Array 的初始化方法来创建一个所有节点的数组，用 allSatisfy 来检查某个条件是否对所有元素都适用，或者仅仅只是用 for 循环来列举文档中的所有节点。

通过同样的方式，我们可以为 ConstantIterator 和 FibsIterator 创建对应的序列。这里就不进行展示了，你可以自己尝试一下。不同之处在于这些迭代器会创建出无穷序列，你可以使用像是 for i in fibsSequence.prefix(10) 的方式来截取其中有限的一段进行测试。

## 迭代器和值语义

至今为止，我们看到的迭代器都具有值语义。和我们期待的结果一样，如果复制一份，迭代器的所有状态也都会被复制，这两个迭代器将分别在自己的范围内工作。也就是说，标准库中的大部分迭代器也都具有值语义，不过也有例外存在。

为了说明值语义和引用语义的不同，我们先来看个 StrideIterator 的例子。它是 stride(from:to:by:) 方法返回的序列所使用的迭代器。我们可以创建一个 StrideIterator 并试着调用几次 next 方法：

```
// 一个从 0 到 9 的序列
let seq = stride(from: 0, to: 10, by: 1)
```

```
var i1 = seq.makeIterator()  
i1.next() // Optional(0)  
i1.next() // Optional(1)
```

现在，`i1` 已经准备好返回 2 了。接下来，我们对它进行复制：

```
var i2 = i1
```

这样，原有的迭代器和新复制的迭代器就是分开且独立的了。在下两次 `next` 时，它们都会分别返回 2 和 3：

```
i1.next() // Optional(2)  
i1.next() // Optional(3)  
i2.next() // Optional(2)  
i2.next() // Optional(3)
```

这是因为 `StrideIterator` 是一个很简单的结构体，它的实现和上面的斐波纳契迭代器没有太大不同，也具有值语义。

接下来，我们再来看一个不具有值语义的迭代器的例子。`AnyIterator` 是一个对别的迭代器进行封装的迭代器，它可以将原始迭代器的具体类型“抹消”掉。比如你在创建公有 API 时想要将一个很复杂的迭代器的具体类型隐藏起来，而不暴露它的具体实现的时候，就可以使用这种迭代器。`AnyIterator` 进行封装的做法是将另外的迭代器包装到一个内部的盒子对象中，而这个对象是引用类型（如果你想要了解具体到底做了什么，可以看看协议一章中关于类型消除器部分的内容）。

为了了解为什么 `AnyIterator` 的行为和其内部使用的盒子对象有关，我们创建一个包装了 `i1` 的 `AnyIterator`，然后进行复制：

```
var i3 = AnyIterator(i1)  
var i4 = i3
```

在这种情况下，原来的迭代器和它的副本就不再是彼此独立的了，因为它们不再是一个结构体，`AnyIterator` 并不具有值语义。`AnyIterator` 中用来存储原始迭代器的盒子对象是一个类实例，

当我们将 `i3` 赋值给 `i4` 的时候，只有对这个盒子的引用被复制了。盒子里存储的对象将被两个迭代器所共享。所以，任何对 `i3` 或者 `i4` 进行的 `next` 调用，实际上都作用于底层那个相同的原始迭代器：

```
i3.next() // Optional(4)  
i4.next() // Optional(5)  
i3.next() // Optional(6)
```

显然，这可能会造成一些 bug，不过在实践中你可能很少会遇到这种问题，因为迭代器通常不会在代码里被传来传去。基本上你只是会在本地创建迭代器，用它来循环元素，然后就将其抛弃。这种行为有时是显式进行的，但更多的时候是通过使用 `for` 循环隐式完成的。如果你发现要与其他对象共享迭代器，可以考虑将它封装到序列中，而不是直接传递它。

## 基于函数的迭代器和序列

`AnyIterator` 还有另外一个接受 `next` 函数作为参数的初始化方法。把它和对应的 `AnySequence` 类型结合起来使用，就可以让我们在不定义任何新类型的情况下，创建迭代器和序列。举个例子，我们可以通过一个返回 `AnyIterator` 的函数来定义斐波纳契迭代器：

```
func fibsIterator() -> AnyIterator<Int> {  
    var state = (0, 1)  
    return AnyIterator {  
        let upcomingNumber = state.0  
        state = (state.1, state.0 + state.1)  
        return upcomingNumber  
    }  
}
```

通过将 `state` 放到迭代器的 `next` 函数外面，并在闭包中将其进行捕获，闭包就可以在每次被调用时对其进行更新。这里的定义和上面使用自定义类型的斐波纳契迭代器只有一个功能上的不同，那就是自定义的结构体具有值语义，而使用 `AnyIterator` 定义的没有。

通过这种方式创建序列甚至更简单，因为 `AnySequence` 提供了一个接受返回迭代器的函数作为参数的初始化方法：

```
let fibsSequence = AnySequence(fibsIterator)
Array(fibsSequence.prefix(10)) // [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

另一种方法是使用 `sequence` 函数，这个函数有两个版本。第一个版本，`sequence(first:next:)` 它使用第一个参数的值作为序列的首个元素，并使用 `next` 参数传入的闭包生成序列的后续元素，最后返回生成的序列。另一个版本是 `sequence(state:next:)`，因为可以在两次 `next` 闭包被调用之间保存任意的可变状态，所以它更强大一些。通过它，我们可以只进行一次方法调用就构建出斐波纳契序列：

```
let fibsSequence2 = sequence(state: (0, 1)) { state -> Int? in
    let upcomingNumber = state.0
    state = (state.1, state.0 + state.1)
    return upcomingNumber
}
```

```
Array(fibsSequence2.prefix(10)) // [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

`sequence(first:next:)` 和 `sequence(state:next:)` 的返回值类型是 `UnfoldSequence`。这个类型的名称来自函数式编程，在函数式编程中，这种操作被称为 **展开 (unfold)**。`sequence` 是和 `reduce` 对应的，在函数式编程中 `reduce` 又常被叫做 **折叠 (fold)**。`reduce` 将一个序列缩减（或者说折叠）为一个单一的返回值，而 `sequence` 则将一个单一的值扩展（或者说展开）形成一个序列。

尽管 `AnyIterator` 要比那些有一长串复杂名称的迭代器看上去有好一些，标准库还是由于性能方面的原因更倾向于使用专门定制的迭代器类型。使用 `AnyIterator` 会导致编译器难以对这部分代码进行优化，有时，这会带来上百倍的性能损失。Ben 在 Swift 论坛上发表过一篇 [详细讨论这个问题的帖子](#)。

就像我们至今为止看到的迭代器一样，`sequence` 对于 `next` 闭包的使用是被延迟的。也就是说，序列的下一个值不会被预先计算，它只在调用者需要的时候生成。这使得我们可以使用类似 `fibsSequence2.prefix(10)` 这样的语句，因为 `prefix(10)` 只会向序列请求它的前十个元素，然后停止。如果序列是主动计算它的所有值的话，因为序列是无限的，程序将会在有机会执行下一步之前就因为整数溢出的问题而发生崩溃。

对于序列和集合来说，它们之间的一个重要区别就是序列可以是无限的，而集合则不行。

## 单次遍历序列

序列并不只限于像是数组或者列表这样的传统集合数据类型。像是网络流，磁盘上的文件，UI事件的流，以及其他很多类型的数据都可以使用序列进行建模。但它们之中，并不是所有类型的序列都和数组一样，可以让你反复遍历其中的元素。

斐波纳契序列确实不会因为遍历其中的元素而发生改变，你可以从 0 开始再次进行遍历，但是像是网络包流这样的序列则只能遍历一次。就算再次对其进行迭代，它也不会产生同样的值。但斐波纳契序列和网络包流都是有效的序列，因此，[Sequence 的文档非常明确地指出了序列并不保证可以被多次遍历](#)：

Sequence 协议并不关心遵守该协议的类型是否会在迭代后将序列的元素销毁。也就是说，请不要假设对一个序列进行多次的 for-in 循环将继续之前的迭代或是从头开始：

```
for element in sequence {  
    if ... some condition { break }  
}  
  
for element in sequence {  
    // 未定义行为  
}
```

一个非集合的序列可能会在第二次 for-in 循环时产生随机的序列元素。

这也解释了为什么我们只有在集合类型上能见到 first 这个看起来很简单的属性，而序列中却没有。调用一个属性的 getter 方法应该是没有任何副作用的，但只有 Collection 协议能保证多次进行迭代是安全的。

举一个只能单次遍历序列的例子：我们有一个对 readLine 函数的封装，它会从标准输入中读取一行：

```
let standardIn = AnySequence {  
    AnyIterator {  
        readLine()  
    }  
}
```

现在，你可以使用 Sequence 的各种扩展来进行操作了，比如你可以这样来写一个带有行号的类似 Unix 中 cat 命令的函数：

```
let numberedStdIn = standardIn.enumerated()  
for (i, line) in numberedStdIn {  
    print("\(i+1):\n(line)")  
}
```

enumerated 将一个序列转化为另一个带有（从0开始）递增数字的新序列。和 readLine 进行的封装一样，这里的元素也是延迟生成的。对原序列的消耗只在你通过迭代器读取序列下一个元素的时候发生，而不是在序列被创建时发生的。因此，如果你在命令行中运行上面的代码，会看到程序在 for 循环中进行等待。当你输入一行内容并按下回车的时候，程序才会打印出相应的内容。当按下 control-D 结束输入的时候，程序会停止等待。不过无论如何，每次 enumerated 从 standardIn 中获取一行时，它都会消耗掉标准输入中的一行，你没有办法将这个序列迭代两次并获得相同的结果。

如果只是写一个 Sequence 扩展，你并不需要考虑这个序列是否只能单次遍历，但你也应该尽可能基于单次遍历的序列实现你的算法。如果你是一个序列方法的调用者，则应该时刻提醒自己注意正在使用的序列是否只能单次遍历。

如果一个序列遵守 Collection 协议的话，那它肯定可以被反复遍历，因为 Collection 在这方面进行了保证。但反过来却不一定，标准库中就有些序列可以安全地多次遍历，但它们并不是集合类型。例如： stride(from:to:by:) 返回的 StrideTo，以及 stride(from:through:by:) 返回的 StrideThrough。

## 序列和迭代器之间的关系

序列和迭代器非常相似，你可能会问，为什么它们会被分为不同的类型？为什么不能直接把 `IteratorProtocol` 的功能包含到 `Sequence` 中呢？对于单次遍历的序列（例如之前标准输入的例子），这么做确实没有问题。这类序列自己持有迭代状态，并且会随着遍历而发生改变。

然而，对于像斐波纳契序列这样的可多次遍历序列来说，它的值不能随着 `for` 循环而改变，它们需要独立的遍历状态，这就是迭代器所存在的意义（当然还需要遍历的逻辑，不过这部分是序列的内容）。`makeliterator` 方法的目的就是创建这样一个遍历状态。

其实，可以把每一个迭代器都看作是由它们返回的元素所组成的单次遍历序列。实际上，只要你声明迭代器实现了 `Sequence` 协议，它就可以名正言顺地成为一个序列类型。因为 `Sequence` 为迭代器类型提供了一个默认的 `makeliterator` 实现，这个方法就是返回 `self` 本身。

标准库中的大部分迭代器都实现了 `Sequence` 协议。

## 集合类型

集合类型 (`Collection`) 指的是那些可以被多次遍历且保持一致的序列。除了线性遍历以外，集合中的元素也可以通过下标索引的方式访问。下标索引通常是整数，至少在数组中是这样。不过我们马上会看到，索引也可以是一些不透明值（比如在字典或者字符串中），它们用起来有时就不如整数那样直观。集合的索引值可以构成一个有限的范围，包含了定义好的开始和结束位置。也就是说，和序列不同，集合类型不能是无限的。每一个集合类型还有一个关联类型 `SubSequence`，它表示集合中一段连续内容的切片。

`Collection` 协议是建立在 `Sequence` 协议上的。除了从 `Sequence` 继承了全部方法以外，得益于可以获取指定位置的元素以及稳定迭代的保证，集合还获取了一些新的能力。比如 `count` 属性（如果对只能单次遍历的序列进行计数，将会消耗序列中的元素，这显然不是我们的目的）。

即使用不到集合类型提供的这些特性，你依旧可以让自定义序列满足 `Collection` 协议，这将告诉你的序列类型的使用者该序列是有限的，而且可以进行多次迭代。不过如果你只是想说明序列可以被多次迭代，但却必须选择一个合适的索引类型，确实会显得比较奇怪。特别是在实现 `Collection` 协议时，最难的部分就是选取一个合适的索引类型来表达集合类型中的位置。这样设计的一个目的是，Swift 团队希望避免引入一个专门的可多次遍历序列的协议，因为它和 `Sequence` 拥有同样的要求，但是语义却不一致，这容易让用户感到迷惑。

集合类型在标准库中运用广泛。除了 Array, Dictionary 和 Set 之外, String 和它的各种视图, [Closed]Range 以及 UnsafeBufferPointer 都是集合类型。在标准库之外, Foundation 框架中的 Data 是使用最频繁的集合类型。[Swift Collections](#) 包则提供了像是有序 Set 和字典这类附加的数据结构。

## 自定义的集合类型

为了展示 Swift 中集合类型的工作方式, 我们将实现一个自己的集合类型。“队列”可能是在 Swift 标准库中没有被实现, 但是却最有用的容器类型了(不过在 [Swift Collections](#) 包中, 有一个双端队列了)。Swift 数组可以很容易地被当作栈来使用, 我们可以用 append 来入栈, 用 popLast 出栈。但是对于队列来说, 就不那么理想。你可以结合使用 push 和 remove(at: 0) 来实现, 但是因为数组是在连续的内存中持有元素的, 所以移除数组中首个元素时, 其他每个元素都需要移动去填补空白, 这个操作的复杂度会是  $O(n)$  (而出栈最后一个元素只需要常数时间就能完成)。

下面是一个简单的先进先出队列, 我们基于两个数组实现了它的 enqueue 和 dequeue 方法:

```
/// 一个高效的 FIFO 队列, 其中元素类型为 `Element`  
struct FIFOQueue<Element> {  
    private var left: [Element] = []  
    private var right: [Element] = []  
  
    /// 将元素添加到队列最后  
    /// - 复杂度: O(1)  
    mutating func enqueue(_ newElement: Element) {  
        right.append(newElement)  
    }  
  
    /// 从队列前端移除一个元素  
    /// 当队列为空时, 返回 nil  
    /// - 复杂度: 平摊 O(1)  
    mutating func dequeue() -> Element? {  
        if left.isEmpty {  
            left = right.reversed()  
        }  
        return left.popFirst()  
    }  
}
```

```
    right.removeAll()  
}  
return left.popLast()  
}  
}
```

这个实现使用两个栈(两个常规的数组)来模拟队列的行为。当元素入队时，它们被添加到“右”栈中。当元素出队时，它们从“右”栈的反序数组，也就是“左”栈中被弹出。当左栈变为空时，再将右栈反序后设置为左栈。

你可能会对 `dequeue` 操作被声明为  $O(1)$  感到有点奇怪。确实，它包含了一个复杂度为  $O(n)$  的 `reverse` 操作。对于单个的操作来说可能耗时会长一些，不过对于非常多的 `push` 和 `pop` 操作来说，取出一个元素的平摊耗时是一个常数。

理解这个复杂度的关键在于理解反向操作发生的频率以及发生在多少个元素上。我们可以使用“银行家理论”来分析平摊复杂度。想象一下，你每次将一个元素放入队列，就相当于你在银行存了一块钱。接下来，你把右侧的栈的内容转移到左侧去，因为对应每个已经入队的元素，你在银行里都相当于有一块钱。你可以用这些钱来支付反转。你的账户永远不会负债，你也从来不会花费比你付出的更多的东西。

这个理论可以用来解释一个操作的消耗在时间上进行平摊的情况，即便其中的某次调用可能不是常数，但平摊下来以后这个耗时依然是常数。Swift 中向数组后面添加一个元素的操作是常数时间复杂度，这也可以用同样的理论进行解释。当数组存储空间耗尽时，它需要申请更大的空间，并且把所有已经存在于数组中的元素复制到新的存储中去。但是因为每次申请空间都会使存储空间翻倍，“添加元素，支付一块钱，数组尺寸翻倍，最多耗费所有钱来进行复制”这个理论依然是有效的。

现在我们拥有一个可以出队和入队元素的容器了。下一步是为 `FIFOQueue` 添加 `Collection` 协议的支持。不幸的是，在 Swift 中，想要找出要实现某个协议所需要提供的最少实现有时候不容易。

在本章写作的时候，`Collection` 协议有五个关联类型，五个属性，六个实例方法，以及两个下标操作符：

```
protocol Collection: Sequence {
    associatedtype Element // 从 Sequence 继承而来
    associatedtype Index: Comparable
    //省去了 where ... 语句

    var startIndex: Index { get }
    var endIndex: Index { get }

    associatedtype Iterator = IndexingIterator<Self>
    associatedtype SubSequence: Collection = Slice<Self>
    where Element == SubSequence.Element,
          SubSequence == SubSequence.SubSequence

    subscript(position: Index) -> Element { get }
    subscript(bounds: Range<Index>) -> SubSequence { get }

    associatedtype Indices: Collection =
        DefaultIndices<Self> where Indices == Indices.SubSequence

    var indices: Indices { get }
    var isEmpty: Bool { get }
    var count: Int { get }

    func makeIterator() -> Iterator // 从 Sequence 继承而来
    func index(_ i: Index, offsetBy distance: Int) -> Index
    func index(_ i: Index, offsetBy distance: Int, limitedBy limit: Index) -> Index?
    func distance(from start: Index, to end: Index) -> Int
    func index(after i: Index) -> Index
    func formIndex(after i: inout Index)
}
```

关联类型 SubSequence 使用了递归约束表明 SubSequence 自身也是一个集合类型。这个约束确保了 SubSequence 中元素的类型和原始集合中的元素类型是相同的，并且，

SubSequence 中的 SubSequence 类型和它自身相同。例如：String 的 SubSequence 类型是 Substring，而 Substring 中的 SubSequence 类型仍旧是 Substring。

关联类型 Indices 也是一个集合类型。为了突出重点，我们省去了Collection中对 Index 类型的一长串约束。简单来说，就是 Collection 中的 Index 是这个 Indices 集合中的元素类型、索引类型以及 Indices.SubSequence 集合中的索引类型。

考虑到上面的要求如此繁杂，遵守 Collection 看起来让人望而却步。不过，实际上事情并没有那么糟糕。除了 Index 和 Element 以外，其他的关联类型都有默认值，除非自定义类型有特殊要求，否则你都不必去关心它们。对于大部分方法、属性和下标，情况同样如此：Collection 的协议扩展也为我们提供了默认实现。不过，这些扩展中，有些包含对关联类型的约束，它们要求关联类型是协议中的默认类型；比如，Collection 只为 Iterator 为 IndexingIterator<Self> 的类型提供了 makeliterator 方法的默认实现：

```
extension Collection where Iterator == IndexingIterator<Self> {  
    func makeliterator() -> IndexingIterator<Self>  
}
```

如果你的类型需要一个不同的迭代器类型，你就需要自己实现上面这个方法。

为了实现一个协议，找出哪些方法必须实现，哪些存在默认实现，这件事情本身并不十分困难，不过它需要你花时间亲自去找，而且一旦你不小心，就会进入到根据编译器的结果进行猜猜猜的游戏里。其中最恼人的部分在于，其实编译器本来知道实现一个协议必备的所有条件，但它提供给我们的信息却不是十分有用。

结果就是，现如今，你最应该给予希望的是在文档中去寻找满足一个协议的需要实现的最小要求。好在，这方面 Collection 不会让你失望：

... 要让一个类型实现 Collection，你必须声明以下内容：

- startIndex 和 endIndex 属性。
- 至少能以只读方式访问集合元素的下标操作符。

→ 用来在集合索引之间进行步进的 index(after:) 方法。

于是最后，我们需要实现的有：

```
protocol Collection: Sequence {
    /// 一个表示序列中元素的类型
    associatedtype Element

    /// 一个表示集合中位置的类型
    associatedtype Index: Comparable
    /// 一个非空集合中首个元素的位置
    var startIndex: Index { get }
    /// 集合中超过末位的位置 — 也就是比最后一个有效下标值大 1 的位置
    var endIndex: Index { get }
    /// 返回在给定索引之后的位置
    func index(after i: Index) -> Index
    /// 访问特定位置的元素
    subscript(position: Index) -> Element { get }
}
```

至此，我们就能让 FIFOQueue 满足 Collection 了：

```
if position < left.endIndex {  
    return left[left.count - position - 1]  
} else {  
    return right[position - left.count]  
}  
}  
}  
}
```

这里，我们使用了 Int 作为 FIFOQueue 的 Index 类型。但并没有显式地提供这个关联类型，和 Element 一样，Swift 可以帮我们从方法和属性的定义中将它推断出来。注意索引是从集合开头开始返回元素的，所以 FIFOQueue.first 将会返回下一个即将被出队的元素（你可以将它当做 peek 来使用）。

通过这几行代码，我们的队列就已经拥有超过 40 个方法和属性可供使用了。比如，我们可以迭代队列：

```
var queue = FIFOQueue<String>()  
for x in ["1", "2", "foo", "3"] {  
    queue.enqueue(x)  
}  
  
for s in queue {  
    print(s, terminator: " ")  
} // 1 2 foo 3
```

我们可以将队列传递给接受序列的方法：

```
var a = Array(queue) // ["1", "2", "foo", "3"]  
a.append(contentsOf: queue[2...3])  
a // ["1", "2", "foo", "3", "foo", "3"]
```

我们可以调用那些 Sequence 的扩展方法和属性：

```
queue.map {$0.uppercased()} // ["1", "2", "FOO", "3"]
```

```
queue.compactMap { Int($0) } // [1, 2, 3]
queue.filter { $0.count > 1 } // ["foo"]
queue.sorted() // ["1", "2", "3", "foo"]
queue.joined(separator: " ") // 1 2 foo 3
```

我们也可以调用 Collection 的扩展方法和属性：

```
queue.isEmpty // false
queue.count // 4
queue.first // Optional("1")
```

## 数组字面量

当实现一个类似 FIFOQueue 这样的集合类型时，最好也去实现一下 ExpressibleByArrayLiteral。这可以让用户能够以他们所熟知的 [value1, value2, etc] 语法创建一个队列。而这个协议只要求我们实现一个下面这样的初始化方法就好了：

```
extension FIFOQueue: ExpressibleByArrayLiteral {
    public init(arrayLiteral elements: Element...) {
        self.init(left: elements.reversed(), right: [])
    }
}
```

对 FIFOQueue 的逻辑来说，我们希望元素已经在左侧的缓冲区准备好待用。当然，我们也可以将这些元素直接放在右侧数组里，但因为在出队时迟早要将它们复制到左侧去，所以直接先逆序将它们复制过去会更高效一些。

现在我们就可以用数组字面量来创建一个队列了：

```
let queue2: FIFOQueue = [1, 2, 3] // FIFOQueue<Int>(left: [3, 2, 1], right: [])
```

在这里需要特别注意 Swift 中字面量和类型的区别。这里的 [1, 2, 3] 并不是一个数组，它只是一个“数组字面量”，是一种写法，我们可以用它创建任意实现了 ExpressibleByArrayLiteral

的类型。在这个字面量里还包括了其他的字面量类型，比如能够创建任意遵守 ExpressibleByIntegerLiteral 类型的整数型字面量。

这些字面量有“默认”的类型，如果你不指明类型，那 Swift 将假设你想要的就是默认的类型。正如你所料，数组字面量的默认类型是 Array，整数字面量的默认类型是 Int，浮点数字面量默认为 Double，而字符串字面量则对应 String。但这只发生在你没有指定类型的情况下，举个例子，上面声明了一个类型为 Int 的队列类型，但是如果你指定了其他整数类型的话，你也可以声明一个其他类型的队列：

```
let byteQueue: FIFOQueue<UInt8> = [1,2,3]
// FIFOQueue<UInt8>(left: [3, 2, 1], right: [])
```

通常来说，字面量的类型可以从上下文中推断出来。举个例子，下面这个函数可以接受一个从字面量创建的参数，而调用时所传递的字面量的类型，可以根据函数参数的类型被推断出来：

```
func takesSetOfFloats(floats: Set<Float>) {
    ...
}
```

takesSetOfFloats(floats: [1,2,3])

这个字面量被推断为 Set<Float>，而不是 Array<Int>。

## 关联类型

我们已经看到 Collection 为除了 Index 和 Element 以外的关联类型都提供了默认值。虽然你不需要太多关心其他的关联类型，不过为了更好地理解这些类型的用途，我们还是逐个过一遍它们。

**Iterator** - 这是从 Sequence 继承来的关联类型。我们已经在关于序列的篇幅中详细看过迭代器的相关内容了。集合类型中的默认迭代器类型是 IndexingIterator<Self>，这是个很简单的结构体，它对集合进行了封装，并用集合本身的索引来迭代每个元素。

标准库中大多数集合类型都使用 `IndexingIterator` 作为它们的迭代器。你几乎没有理由需要为一个自定义的集合类型单独定义迭代器。

**SubSequence** - 是表示集合中一段连续内容切片的类型。`SubSequence` 自身也是集合类型。它的默认值是 `Slice<Self>`，这是对原始集合的封装，并存储了切片相对于原始集合的起始和终止索引。

为一个集合类型自定义它的 `SubSequence` 是很常见的情况，特别是当 `SubSequence` 就是 `Self` 时（也就是说，一个集合的切片拥有和集合本身相同的类型）。Foundation 框架中的 `Data` 就是一个这种集合的例子。我们将在本章稍后的子序列部分再回到这个话题。

**Indices** - 集合的 `indices` 属性的类型。它是集合中所有有效索引按升序排列组成的集合。注意 `endIndex` 并不包含在其中，因为 `endIndex` 代表的是最后一个有效索引之后的那个索引，它不是有效的下标参数。

`Indices` 的默认类型是 `DefaultIndices<Self>`。和 `Slice` 一样，它是对于原来的集合类型的简单封装，并包含起始和结束索引。它需要保持对原集合的引用，这样才能够对索引进行步进。当用户在迭代索引的同时改变集合的内容的时候，可能会造成意想不到的性能问题：如果集合是以写时复制（就像标准库中的所有集合类型所做的一样）来实现的话，这个对于集合的额外引用将触发不必要的复制。因此，如果你可以为自定义集合类型提供一个不需要引用原始序列的 `Indices` 类型，这绝对是一个值得尝试的优化。实际上，只要索引位置的计算不依赖于集合自身，例如数组或之前实现的 `FIFOQueue`，就都应该如此。如果你的索引是整数类型，可以直接使用 `Range<Index>`：

```
extension FIFOQueue: Collection {  
    // ...  
    typealias Indices = Range<Int>  
    var indices: Range<Int> {  
        startIndex..endIndex  
    }  
}
```

## 索引

索引表示了集合中的位置。每个集合都有两个特殊的索引值：`startIndex` 和 `endIndex`。`startIndex` 指定集合中第一个元素，`endIndex` 是集合中最后一个元素的下一个位置。所以 `endIndex` 并不是一个有效的下标索引，你可以用它创建索引的范围 (`someIndex..<endIndex`)，或者将它与别的索引进行比较，例如：控制循环的退出条件 (`while someIndex < endIndex`)。

到现在为止，我们都使用整数作为集合的索引。Array 如此，我们的 `FIFOQueue` 类型在经过一些操作之后亦是如此。整数索引很直观，但它并不是唯一选项。集合类型的 `Index` 的唯一要求是，它必须实现 `Comparable`，换句话说，索引必须要有确定的顺序。

用 `Dictionary` 作为例子，因为我们使用字典的键来定位字典中的值，所以可能使用键来作为字典的索引看上去会是很自然的选择。但这是行不通的，因为你无法（像计算整数索引位置一样）移动一个键，你不能给出某个键之后的索引值应该是什么。另外，使用索引进行下标访问应该立即返回获取的元素，而不是再去搜索或者计算哈希值。

所以，字典的索引是 `DictionaryIndex` 类型，它是一个指向字典内部存储缓冲区的不透明值。事实上这个类型只是一个 `Int` 偏移值的封装，不过它包含了字典类型使用者所不需要关心的实现细节（其实事情要比这里描述的复杂得多。一方面，字典的索引还包含了一个计数器，字典通过这个计数器检测不可用索引的访问；另一方面，字典还可能被传递到 Objective-C 的 API，或者是由 Objective-C 的 API 获取到的，为了效率，它们会使用 `NSDictionary` 作为背后的存储，这样的字典的索引类型又有所不同。不过意思是一样的）。

这也说明了为什么通过索引下标访问 `Dictionary` 时返回的值，不像用字典键下标访问时那样是一个可选值。我们平时用键访问的 `subscript(_ key: Key)` 方法是直接定义在 `Dictionary` 上的下标方法的一个重载，它返回可选的 `Value`：

```
struct Dictionary {  
    ...  
    subscript(key: Key) -> Value?  
}
```

而通过索引访问的方法是 `Collection` 协议所定义的，它总是返回非可选值。因为使用（像是数组里的越界索引这样的）无效的下标被认为是程序员犯的错误，即便程序“崩了”也是理所当然的事情：

```
protocol Collection {  
    subscript(position: Index) -> Element { get }  
}
```

注意，`subscript` 的返回类型是 `Element`。`Element` 类型是一个多元组：`(key: Key, value: Value)`，因此对 `Dictionary`，下标访问返回的是一个键值对，而非单个的 `Value`。这也是通过 `for` 循环遍历字典会得到键值对的原因。

在内建集合类型的数组索引部分，我们讨论过为什么像 Swift 这样一门“注重操作安全的”语言不把所有可能失败的操作用可选值或者错误包装起来的原因。“如果所有 API 都可能失败，那你就没法儿写代码了。你需要有一个基本盘的东西可以依赖，并且信任这些操作的正确性”，否则你的代码将会深陷安全检查的囹圄。

## 索引失效

当集合发生改变时，索引可能会失效。失效有两层含义，它可能意味着虽然索引本身仍是有效的，但是它现在指向了一个另外的元素；或者有可能索引本身就已经无效了，通过它对集合访问将造成崩溃。在你用数组进行思考时，这个问题会比较直观。当你添加一个元素后，所有已有的索引依然有效。但是当你移除第一个元素后，指向最后一个元素的索引就无效了。同时，那些较小的索引依然有效，但是它们所指向的元素都发生了改变。

字典的索引不会随着新键值对的加入而失效，直到字典的尺寸增大到触发重新的内存分配。这是因为一般情况下字典存储的缓冲区内的元素位置是不会改变的，而在元素超过缓冲区尺寸时，缓冲区会被重新分配，其中的所有元素的哈希值也会被重新计算。从字典中移除元素总是会使索引失效。

索引应该是一个只存储包含描述元素位置所需最小信息的简单值。在尽可能的情况下，索引不应该持有对它们集合的引用。类似地，一个集合通常也无法区分它“自己的”索引和一个来自同样类型集合的索引。用数组来举例子就再清楚不过了。显然，你可以用从一个数组中获取到的整数索引值来访问另一个数组的内容：

```
let numbers = [1,2,3,4]  
let squares = numbers.map { $0 * $0 }  
let numbersIndex = numbers.firstIndex(of: 4)! // 3
```

```
squares[numbersIndex] // 16
```

这对那些不透明的索引类型，例如：String.Index，也是适用的。在这个例子中，我们使用一个字符串的 startIndex 来访问另一个字符串的首字符：

```
let hello = "Hello"  
let world = "World"  
let helloIdx = hello.startIndex  
world[helloIdx] // W
```

不过，能这么做并不意味着这么做是个好主意。如果我们用这个索引作为下标访问一个空字符串的话，程序将因为索引越界而发生崩溃。另外，由于字符长度是可变的，指向一个字符串中第  $n$  个字符的索引在另一个字符串中不一定指向的是一个正确的字符边界。在 [字符串这一章中](#)，我们已经详细讨论了这个话题。

另外，在集合之间共享索引也是有其合理用法的，在切片上我们会大量使用这种方式。Collection 协议要求原集合的索引必须在切片上也命中同样的元素，这样一来，在切片之间共享索引就总是一个安全的操作。

## 索引步进

Swift 3 在集合处理索引遍历的方面引入了一个大的变化。现在，向前或者向后移动索引（或者说，从一个给定索引计算出新的索引）的任务是由集合来负责的了，而在 Swift 3 之前，索引是可以自行移动的。你之前可能会用 someIndex.successor() 来获取下一个索引，现在你需要写 collection.index(after: someIndex)。

为什么 Swift 团队要做这样的变化？简单说，为了性能。通常，从一个索引获取到另一个索引会涉及到集合内部的信息。数组中的索引没有这个顾虑，因为在数组里索引的步进就是简单的加法运算。但是对于字符串索引，因为字符在 Swift 中是尺寸可变的，所以计算索引时就需要考虑字符的实际数据到底是什么。

在之前可以自行移动的索引模型中，索引必须保存一个对集合存储的引用。这个额外的引用会在集合被迭代修改时造成不必要的复制，它带来的开销足以抵消标准库中集合的写时复制特性带来的性能改善。

新的模型通过将索引保持为简单值，就可以解决这个问题。它的概念更容易理解，也让你可以更简单地实现自定义的索引类型。在大多数情况下，索引可以由一个或者两个高效地对集合底层存储元素位置进行编码的整数值所表示。

新的索引模型的缺点是，它的语法则显得有一些啰嗦。

## 自定义集合索引

作为非整数索引集合的例子，我们将会构建一种在字符串中迭代单词的方式。当你想要将一个字符串分割成一个个的单词，最简单的方法就是使用

`split(separator:maxSplits:omittingEmptySubsequences:)`(当然，这是对英文而言)，这个方法将使用提供的分割元素进行分割，把一个集合转变为其 SubSequence 的数组：

```
var str = "Still I see monsters"  
str.split(separator: " ") // ["Still", "I", "see", "monsters"]
```

返回数组中的每个单词的类型都是 SubString，这正是 String 所关联的 SubSequence 类型。当你想要分割一个集合类型时，`split` 方法往往是最适合的工具。不过，它有一个缺点：这个方法将会热心地为你计算出整个数组。如果你的字符串非常大，而且你只需要前几个词的话，这么做是相当低效的。为了提高性能，我们要构建一个 Words 集合，它能够让我们不一次性地计算出所有单词，而是可以用延迟加载的方式进行迭代。[\(Swift Algorithms 包提供了一个延迟版本的split 以及其他一些很有用的集合算法。在产品代码中，相比起我们这里的实现，你应该去使用那个包里方法\)。](#)

我们从在 SubString 中寻找第一个单词的范围开始。这里，我们直接使用空格作为单词的边界，当然，作为练习你也可以将它写为可配置的值。一个子字符串可能由若干个空格作为开始，我们将它们跳过。`start` 是所有前置空格都被移除了的子字符串。接下来，我们寻找下一个空格；如果找到空格，就以它作为单词的结束边界。如果无法找到另一个空格，则使用 `endIndex`：

```
extension Substring {  
    var nextWordRange: Range<Index> {  
        let start = drop(while: { $0 == " " })  
        let end = start.firstIndex(where: { $0 == " " }) ?? endIndex  
        return start.startIndex..
```

```
    }  
}
```

注意 Range 是一个半开范围，也就是说，上边界 end 是不包含在单词范围里的。

逻辑上来说，Words 集合类型的索引类型的第一选择应该是 Int：某个整数索引 i 代表的是集合类型中的第 i 个单词。不过，通过索引下标访问某个元素应该是一个  $O(1)$  操作，而为了找到第 i 个单词，我们需要对整个字符串进行处理（这会是一个  $O(n)$  的操作）。

对于索引类型，另一个选择是使用 String.Index。string.startIndex 将成为集合类型的 startIndex，之后的索引则成为下一个单词的开始索引，以此类推。不过不幸的是，这种下标实现也存在一样的问题：寻找最后的单词依然是  $O(n)$  复杂度。

我们采取的办法是将 Range<Substring.Index> 进行包装，然后用它来作为索引类型。因为集合类型的索引需要满足 Comparable，我们还需要为我们的自定义索引类型实现这个协议。在比较两个索引时，我们只比较范围的下边界。虽然在一般的情况下，这并不能完成对于 Range 类型的比较，但是在我们的这个例子中，比较范围下界就足够了。另外，因为 Comparable 继承自 Equatable，所以我们的索引类型还需要实现 ==，不过编译器可以为我们生成这个方法。

通过将 range 属性和它的初始化方法标记为 fileprivate，我们可以把 WordsIndex 当作一个不透明类型来使用；我们的集合类型的使用者不知道它的内部结构，想要创建一个索引，唯一的方式是通过集合的接口：

```
struct WordsIndex: Comparable {  
    fileprivate let range: Range<Substring.Index>  
    fileprivate init(_ value: Range<Substring.Index>) {  
        self.range = value  
    }  
  
    static func <(lhs: Words.Index, rhs: Words.Index) -> Bool {  
        lhs.range.lowerBound < rhs.range.lowerBound  
    }  
}
```

现在，我们就可以来构建 Words 集合类型了。它在底层将 String 作为 SubString 存储(我们会在切片的部分看到原因)并提供了一个起始索引和一个结束索引。Collection 协议要求 startIndex 的复杂度为  $O(1)$ 。不过，现在计算它的开销是  $O(n)$ ， $n$  是字符串开头的空格的数量。因此，我们将会在初始化方法中对它进行计算和存储，而不是将其定义为一个计算属性。对于结束索引，我们使用在底层字符串边界以外的一个空范围来表示：

```
struct Words {  
    let string: Substring  
    let startIndex: WordsIndex  
  
    init(_ s: String) {  
        self.init(s[...])  
    }  
  
    private init(_ s: Substring) {  
        self.string = s  
        self.startIndex = WordsIndex(string.nextWordRange)  
    }  
  
    public var endIndex: WordsIndex {  
        let e = string.endIndex  
        return WordsIndex(e..)  
    }  
}
```

集合类型需要我们提供 subscript 下标方法来获取元素。这里我们直接使用索引中的范围值。使用单词的范围作为索引，可以使下标的实现满足  $O(1)$  复杂度：

```
extension Words {  
    subscript(index: WordsIndex) -> Substring {  
        string[index.range]  
    }  
}
```

Collection 的最后一个要求是给定某个索引时，能够计算出下一个索引。由于索引中范围上界表示的是当前单词的结束的下一个位置。所以，只要上界的值不是字符串的 endIndex (这表示我们已经遍历到了字符串末尾)，我们就可以从这个上界开始，继续在剩余的子字符串中寻找下一个单词的范围：

```
extension Words: Collection {  
    public func index(after i: WordsIndex) -> WordsIndex {  
        guard i.range.upperBound < string.endIndex else { return endIndex }  
        let remainder = string[i.range.upperBound...]  
        return WordsIndex(remainder.nextWordRange)  
    }  
}
```

```
Array(Words(" hello world test ").prefix(2)) // ["hello", "world"]
```

再经过一些完善的话，Words 集合类型将能够可以处理一些更为普遍的问题。首先，我们可以对单词的边界进行设置：相对于现在直接使用空格，我们可以传入一个 isWordBoundary: (Character) -> Bool 的函数来确定单词边界。其次，这些代码并不是只针对字符串的：我们完全可以将 String 替换成任意集合类型。比如，我们可以用同样的算法来将 Data 延迟分割为可处理的小数据块。再强调一次，Swift Algorithms 包已经在 split 的延迟重载中将这些都做好了。

## 子序列

Collection 协议还有一个关联类型，叫做 SubSequence。它表示集合中一个连续的子区间：

```
extension Collection {  
    associatedtype SubSequence: Collection = Slice<Self>  
    where Element == SubSequence.Element,  
          SubSequence == SubSequence.SubSequence  
}
```

SubSequence 用于那些返回原始集合类型切片的操作：

- **prefix** 和 **suffix** — 获取开头或结尾的  $n$  个元素。
- **prefix(while:)** - 从集合开始，获取满足 `while` 指定条件的操作。
- **dropFirst** 和 **dropLast** — 返回移除掉前  $n$  个或后  $n$  个元素的子序列。
- **drop(while:)** - 移除元素，直到条件不再为真，然后返回剩余元素。
- **split** — 将一个序列通过指定的分隔元素截断，返回子序列的数组。

另外，基于范围的下标操作符也会以切片的形式返回这个范围内的索引在原始集合中标记的区间。这样做，相比于直接返回一个包含所有子序列元素的新集合（例如数组）的好处就是不会招致额外的内存分配，因为子序列和它们原始的集合类型是共享内部存储的。

术语“子序列”有它自己的历史背景。之前，`SubSequence` 是 `Sequence` 协议的关联类型。但定义在 `Sequence` 中，使得它无法真正发挥作用，还会招致一些性能问题。于是，在 Swift 5 中，它才被“提升”到 `Collection` 协议。之所以还保留了原有的名称，是为了和已有代码保持兼容。但在集合的上下文里，术语“切片”更为贴切，并且，它和“子序列”是可以互换使用的。

默认情况下，`Collection` 会把 `Slice<Self>` 作为自己的 `SubSequence` 类型。但很多具体类型都有它们自己定制的实现：例如，`String` 的 `SubSequence` 类型是 `Substring`，`Array` 的 `SubSequence` 类型是 `ArraySlice`。

在有些时候，让子序列和原始集合的类型一致，也就是 `SubSequence == Self`，会非常方便。因为这样，只要能传递原始集合类型的地方，你也都能够使用切片类型。`Foundation` 中的 `Data` 就是这么做的，所以当你见到一个 `Data` 实例的时候，你无法分辨这究竟是一块儿完整的数据 (`startIndex == 0` 并且 `endIndex == count`)，还是一大块数据的切片（拥有不是基于 0 开始的索引）。但标准库中的集合却拥有不一样的切片类型，这么做的主要考量是为了防止不经意的内存“泄漏”，一个非常小的切片将会持有它原来的集合类型（有可能非常大）。在切片的生命周期比预期要长得多的时候，这可能造成一些问题。使用它们自己的类型来表示切片，可以比较容易将它们的生命周期绑定到局部作用域中。

对于这一节开始列出的那些标准库中返回切片的方法来说，把 `SubSequence` 和 `[SubSequence]` 作为返回值的类型的确是一个不错的选择。下面这个例子，就以每  $n$  个元素切

割集合。在它的实现里，我们只要迭代集合索引，每步进  $n$ ，就创建一个切片，并把这些切片保存到一个子序列数组里。如果集合中元素个数不是  $n$  的倍数，最后一个切片就会小一些：

```
extension Collection {  
    public func split(batchSize: Int) -> [SubSequence] {  
        var result: [SubSequence] = []  
        var batchStart = startIndex  
        while batchStart < endIndex {  
            let batchEnd = index(batchStart, offsetBy: batchSize,  
                limitedBy: endIndex) ?? endIndex  
            let batch = self[batchStart ..< batchEnd]  
            result.append(batch)  
            batchStart = batchEnd  
        }  
        return result  
    }  
}  
  
let letters = "abcdefg"  
let batches = letters.split(batchSize: 3) // ["abc", "def", "g"]
```

Swift Algorithms 以 chunks(ofCount:) 为名提供了这个操作。

由于子序列也是一个同样以 Element 作为元素类型的集合，我们完全可以用处理集合的方法处理与之对应的切片类型。例如，我们来验证下，刚才实现的 `split` 返回的所有切片中元素的个数和原始集合中元素的个数是相同的：

```
let batchesCount = batches.map { $0.count }.reduce(0, +) // 7  
batchesCount == letters.count // true
```

鉴于它们的内存开销很低，子序列非常适合表达中间结果。但是，为了避免小块切片意外地把整个原始序列保持在内存里，通常不建议长时间保存子序列（例如：定义一个子序列属性），或者把它传递给有可能造成这种结果的方法。为了切断子序列和原始集合类型之间的关系，我们可以用子序列创建一个新集合，例如：`String(substring)` 或 `Array(arraySlice)`。

## 切片

所有的集合类型都有切片操作的默认实现，并有一个接受 Range<Index> 作为参数的 subscript 方法。下面的操作等价于 words.dropFirst():

```
let words: Words = Words("one two three")
let onePastStart = words.index(after: words.startIndex)
let firstDropped = words[onePastStart..
```

因为像是 words[somewhere..

```
let firstDropped2 = words.suffix(from: onePastStart)
// 或者
let firstDropped3 = words[onePastStart...]
```

默认情况下，firstDropped 的类型不是 Words，而是 Slice<Words>。Slice 是基于任意集合类型的一个轻量级封装、它的实现看上去会是这样的：

```
struct Slice<Base: Collection>: Collection {
    typealias Index = Base.Index
    let collection: Base

    var startIndex: Index
    var endIndex: Index

    init(base: Base, bounds: Range<Index>) {
        collection = base
        startIndex = bounds.lowerBound
        endIndex = bounds.upperBound
    }
}
```

```
func index(after i: Index) -> Index {  
    return collection.index(after: i)  
}  
  
subscript(position: Index) -> Base.Element {  
    return collection[position]  
}  
  
subscript(bounds: Range<Index>) -> Slice<Base> {  
    return Slice(base: collection, bounds: bounds)  
}  
}
```

Slice 非常适合作为默认的子序列类型，不过当你创建一个自定义集合类型时，最好还是考虑下是否能将集合类型本身当作它的 SubSequence 使用。对于 Words 来说，这非常容易：

```
extension Words {  
    subscript(range: Range<WordsIndex>) -> Words {  
        let start = range.lowerBound.range.lowerBound  
        let end = range.upperBound.range.upperBound  
        return Words(string[start..<end])  
    }  
}
```

编译器会通过基于范围的下标操作符返回的类型，来推断出 SubSequence 的类型。

将集合类型的 SubSequence 定义为和集合类型相同的类型，可以减轻集合类型使用者的负担，因为他们只需要理解单个类型就可以了，而不需要学习两个类型。不过另一面，让原始集合和它的切片使用不同的类型，有助于避免意外的内存“泄漏”，这也是标准库中使用 ArraySlice 和 Substring 来作为 Array 和 String 的子序列的原因。

## 切片与原集合共享索引

Collection 协议还有另一个正式的要求，那就是切片的索引可以和原集合的索引互换使用。

Swift 文档是这样陈述这个要求的：

集合类型和它的切片拥有相同的索引。只要集合和它的切片在切片被创建后没有改变，切片中某个索引位置上的元素，应当也存在于原集合中同样的索引位置上。

这种模型带来了一个很重要的暗示，那就是即使在使用整数索引时，一个集合的索引也并不需要从 0 开始。这里是一个数组切片的开始索引和结束索引的例子：

```
let cities = ["New York", "Rio", "London", "Berlin",
    "Rome", "Beijing", "Tokyo", "Sydney"]
let slice = cities[2...4]
cities.startIndex // 0
cities.endIndex // 8
slice.startIndex // 2
slice.endIndex // 5
```

这种情况下，如果不小心访问了 slice[0]，你的程序将会崩溃。这也是我们应当尽可能始终选择 for x in collection 的形式，而不去手动计算索引的一个原因。

而我们之前提到过的 Data 在集合的这个特性下，用起来就比较危险了。Data 用整数作为索引，所以很自然的就会想到用数字 0 表示 Data 的起始位置。然而事实并非总是如此，因为 Data 的 SubSequence 类型也是它自己。

如果你确实需要访问索引，使用 for index in collection.indices 在绝大多数时候都比手工计算更可靠。但这也有一个例外，如果你通过索引进行迭代的同时修改了集合，indices 持有的原始集合的强引用会破坏写时复制的优化机制，并引发本不需要的额外拷贝。取决于集合自身的大小，这个拷贝可能会带来非常严重的性能影响（并不是所有集合使用的 Indices 类型都会持有原始集合的强引用，但由于标准库中默认的 DefaultIndices 就是这样做的，所以绝大多数集合类型都有这样的性质）。

要避免这件事情发生，你可以将 for 循环替换为 while 循环，然后手动在每次迭代的时候增加索引值，这样你就不会用到 indices 属性。当你这么做的时候，一定要记住要从 collection.startIndex 开始进行循环，而不要把 0 作为开始。

## 专门的集合类型

和所有精心设计的协议一样，Collection 努力将它的需求控制在最小。为了使更多的类型能满足 Collection，对于实现它的类型来说，应该只需要提供完成 Collection 核心功能所必备的方法就好了。

对于 Collection 来说，有两个特别有意思的限制：一个是它无法往回移动索引，另一个是，它也没有提供像是插入，移除或者替换元素这样改变集合内容的功能。当然，这并不是说满足 Collection 的类型不能拥有这些能力，只是 Collection 协议自身没有把它们当作必须要实现的功能。

有些算法会对集合支持的操作有额外的要求，这使得只有部分集合可以和它们搭配在一起工作。如果可以把这些额外的要求抽象出来形成一些通用的集合变体，用起来就会更加方便。为此，标准库提供了四个专门的集合类型，每一个都用特定的方式给 Collection 追加了新的功能（以下这些内容引用自标准库文档）：

- **BidirectionalCollection** — “一个既支持前向又支持后向遍历的集合。”
- **RandomAccessCollection** — “一个支持高效随机访问索引进行遍历的集合。”
- **MutableCollection** — “一个支持下标赋值的集合。”
- **RangeReplaceableCollection** — “一个支持将任意子范围的元素用别的集合中的元素进行替换的集合。”

让我们一个个来进行讨论。

### BidirectionalCollection

BidirectionalCollection 给集合添加了一个关键的能力，就是通过 index(before:) 方法把索引往回移动一个位置。有了这个方法，就可以对应 first，给出默认的 last 属性的实现了：

```
extension BidirectionalCollection {  
    /// 集合中的最后一个元素。  
    public var last: Element?  
        return isEmpty ? nil : self[index(before: endIndex)]  
    }  
}
```

当然了，Collection 本身也能提供 last 属性，但是这么做不太好。在一个只能前向进行索引的集合类型中，想要获取最后一个元素，你就得一路从头迭代到尾，而这是一个  $O(n)$  操作。通过一个看起来人畜无害的简单的 last 属性来获取最后一个元素，可能会带来问题。比如一个含有数百万条数据的单链表中，为了获取最后一个元素，你会花费很长时间。

标准库中的 String 就是双向索引集合的一个例子。在[字符串](#)这一章我们已经详细讨论过，由于 Unicode 相关的原因，一个字符集合不能含有随机存取的索引，你只能从后往前一个一个对字符进行遍历。

受益于这种可以向前遍历集合的能力，BidirectionalCollection 还实现了一些可以高效执行的方法，比如 suffix，removeLast 和 reversed。其中，reversed 不会直接将集合反转，而是返回一个延时加载的视图：

```
extension BidirectionalCollection {  
    /// 返回一个表达集合中元素逆序排列的视图  
    /// - 复杂度: O(1)  
    public func reversed() -> ReversedCollection<Self> {  
        return ReversedCollection(_base: self)  
    }  
}
```

和上面 Sequence 的 enumerated 封装类似，ReverseCollection 并不会真的把元素逆序排列，而是会持有原来的集合，并使用一个定制的索引类型实现逆序遍历。另外，ReverseCollection 还逆向了所有索引遍历方法，这样一来，向前移动它的索引就相当于在原始序列中向后移动索引，反之一样。

这种做法之所以可行，很大部分是依赖了值语义的特性。在构建时，这个封装将原集合“复制”到它自己的存储中，这样对原序列的子序列进行改变，也不会改变 ReverseCollection 中持有的复制。也就是说，ReverseCollection 可观测到的行为和通过 reversed 返回的数组的行为是一致的。

标准库中的大部分类型都在实现 Collection 的同时，实现了 BidirectionalCollection。然而，像是 Dictionary 和 Set 这样的类型，它们本身就是无序的集合类型，所以对于它们来说，讨论前向迭代还是后向迭代，几乎都是没有意义的。

## RandomAccessCollection

RandomAccessCollection 提供了最高效的元素存取方式，它能够在常数时间内跳转到任意索引。要做到这一点，满足该协议的类型必须能够 (a) 以任意距离移动一个索引，以及 (b) 测量任意两个索引之间的距离，两者都需要是  $O(1)$  时间的常数操作。RandomAccessCollection 以更严格的约束重新声明了关联的 Indices 和 SubSequence 类型，这两个类型自身也必须是可以进行随机存取的。除此之外，相比于 BidirectionalCollection，RandomAccessCollection 并没有更多的要求。不过，满足协议的类型必须确保满足文档所要求的  $O(1)$  复杂度。你可以通过提供 index(\_:offsetBy:) 和 distance(from:to:) 方法，或者是使用一个满足 Strideable 的 Index 类型 (像是 Int)，就可以做到这一点。

一开始，你可能觉得 RandomAccessCollection 没什么大不了的。即使是像我们的 Words 这样的简单前向遍历的集合的索引也能够以任意距离进行前进。但这其中有很大区别，对于 Collection 和 BidirectionalCollection，index(\_:offsetBy:) 操作通过渐进的方式访问下一个索引，直到到达目标索引为止。这显然是一个线性复杂度的操作，随着距离的增加，完成操作需要消耗的时间也线性增长。而随机存取索引则完全不同，它可以直接在两个索引间进行移动。

这个能力是很多算法的关键。例如，当你实现一个泛型二分搜索算法的时候，限制这个算法只能搭配随机存取集合使用是很关键的。因为如果没有这个保证的话，进行二分搜索将会比从头到尾遍历集合还要慢很多。

随机存取的集合类型可以在常数时间内计算 startIndex 和 endIndex 之间的距离，这意味着该集合同样能在常数时间内计算出 count。

## MutableCollection

可变集合支持原地的元素更改。相比于 Collection，MutableCollection 只增加了一个要求，那就是单个元素的下标访问方法 subscript 现在必须提供一个 setter：

```
extension FIFOQueue: MutableCollection {  
    public var startIndex: Int { return 0 }  
    public var endIndex: Int { return left.count + right.count }  
  
    public func index(after i: Int) -> Int {  
        i + 1  
    }  
  
    public subscript(position: Int) -> Element {  
        get {  
            precondition((0..            if position < left.endIndex {  
                return left[left.count - position - 1]  
            } else {  
                return right[position - left.count]  
            }  
        }  
        set {  
            precondition((0..            if position < left.endIndex {  
                left[left.count - position - 1] = newValue  
            } else {  
                return right[position - left.count] = newValue  
            }  
        }  
    }  
}
```

注意编译器不让我们向一个已经存在的 Collection 中通过扩展添加下标 setter 方法。一方面，编译器不允许只提供 setter 而不提供 getter，另一方面，我们也无法重新定义已经存在的 getter 方法。所以我们只能替换掉已经存在的满足 Collection 的扩展，并提供一个新的扩展方法来满足 MutableCollection。现在这个队列可以通过下标进行更改了：

```
var playlist:FIFOQueue = ["Shake It Off", "Blank Space", "Style"]
playlist.first // Optional("Shake It Off")
playlist[0] = "You Belong With Me"
playlist.first // Optional("You Belong With Me")
```

很多对集合进行原地修改的算法都要求对应的集合类型满足 MutableCollection 协议。例如标准库中的原地排序、逆序以及 swapAt 方法。

相对来说，标准库中满足 MutableCollection 的类型并不多。在三个主要的集合类型中，只有 Array 满足这个协议。MutableCollection 允许改变集合中的元素值，但是它不允许改变集合的长度或者元素的顺序。正是这个不可改变元素顺序的要求，解释了为什么 Dictionary 和 Set 虽然是可变的数据结构，却不满足 MutableCollection 的原因。

字典和集合都是无序的集合类型，两者中元素的顺序对于使用这两个集合类型的代码来说是没有定义的。不过，即使是这些集合类型，在内部实现上说，它的元素顺序也是唯一确定的。当你想要通过下标赋值的 MutableCollection 来改变一个元素时，被改变的元素的索引必须保持不变，也就是说，这个索引在 indices 中的位置必须不能改变。Dictionary 和 Set 无法保证这一点，因为它们的索引表示的是元素在内部存储中的位置，而一旦元素发生变化，这个位置也会发生改变。

字符串也不是一个 MutableCollection 类型，因为字符在字符串缓冲区中的大小是不固定的。结果就是，替换字符串中的一个字符需要线性时间完成，缓冲区的尾部也可能需要根据替换的字符适当延长或缩短几个字节。另外，修改了一个字符之后，新字符还可能和相邻的字符形成新的字位族，进而改变字符串的 count 属性。

## RangeReplaceableCollection

对于需要添加或者移除元素的操作，可以使用 RangeReplaceableCollection 协议。这个协议有两个要求：

- 一个空的初始化方法 — 在泛型函数中这很有用，因为它允许一个函数创建相同类型的空集合。
- 一个replaceSubrange(\_:with:) 方法 — 它接受一个要替换的范围以及一个用来进行替换的集合。

RangeReplaceableCollection 是展示协议扩展强大能力的绝佳例子。你只需要实现一个灵活的 replaceSubrange 方法，协议扩展就可以为你引申出一系列有用的方法：

- **append(\_:) 和 append(contentsOf:)** — 将 endIndex..- **remove(at:) 和 removeSubrange(\_:)** — 将 i...i 或者 subrange 替换为空集合。
- **insert(at:) 和 insert(contentsOf:at:)** — 将 i..*- **removeAll** — 将 startIndex..*

这些方法，也是协议要求的一部分。也就是说，特定的集合类型可以根据自己的情况为这些方法提供更高效的实现。这些实现在使用时将比协议扩展中的默认实现具有更高的优先级。

这里，我们选择为之前编写的 FIFOQueue 提供一个简单的不那么高效的实现。在一开始定义这个数据类型的时候，左侧的栈是以逆序持有元素的。为了实现起来简单一些，我们会将这些元素再做一次逆序，然后合并到右侧的数组中，这样我们就可以一次性地对整个范围进行替换了：

```
extension FIFOQueue: RangeReplaceableCollection {  
    mutating func replaceSubrange<C: Collection>(  
        _ subrange: Range<Int>,  
        with newElements: C) where C.Element == Element  
    {  
        right = left.reversed() + right  
        left.removeAll()  
        right.replaceSubrange(subrange, with: newElements)  
    }  
}
```

你可能会想要尝试一些更加高效的实现，比如说先检查要替换的范围是不是确实跨越了左侧栈和右侧栈。在这个例子中，我们没有必要去实现一个空的 init 方法，因为 FIFOQueue 默认就有这个方法了。

和 RandomAccessCollection 扩展了 BidirectionalCollection 不同， RangeReplaceableCollection 并不是对 MutableCollection 的扩展，它们拥有各自独立的继承关系。在标准库中， String 就是一个实现了 RangeReplaceableCollection 但是却没有实现 MutableCollection 的例子。究其原因，是因为我们刚才说过的索引必须在任何一个元素改变时保持稳定，而 String 不能保证这一点。在 字符串 这一章中我们对这个话题有更多探讨。

## 组合能力

这些专门的集合协议还可以被很好地组合起来，作为一组约束，来匹配每个特定算法的要求。举例来说，标准库中有个 sort 方法，可以原地对一个集合进行排序（而不像它的不可变版本 sorted 那样，返回一个排序后的数组）。原地排序需要集合是可变的，如果你想要排序保持迅速，你还需要随机存取。最后，当然你还需要能够比较集合中元素的大小。

将这些要求组合起来， sort 方法被定义在了 MutableCollection 的扩展里，并使用 RandomAccessCollection 和 Element: Comparable 分别约束了集合自身以及集合元素的类型特征：

```
extension MutableCollection
    where Self: RandomAccessCollection, Element: Comparable {
    /// 原地对集合进行排序
    public mutating func sort() { ... }
}
```

## 延迟序列

标准库为支持延迟求值 (lazy evaluation) 提供了两个协议：LazySequenceProtocol 和 LazyCollectionProtocol。延迟求值意味着结果只有在真正需要的时候才会计算出来，这是相较于立即求值 (eager evaluation) 而提出的概念，而后者也是 Swift 默认的编程方式。一个延迟序列 (lazy sequence) 可以在序列的生成和使用之间创建一个隔离：你不必预先创建出整个

序列。相反，只有当使用者真正需要序列的下一个元素时，延迟序列才产生这个元素。这种方式不仅仅是一种编程风格，还可以带来性能上的收益。

在这一章里，我们看到过很多延迟序列的例子了。我们展示了三种不同的构建斐波那契数列的方法：使用定制的迭代器，使用 AnySequence，以及使用 sequence(first:next:) 方法。这些方法都会产生一个包含无穷多个元素的数值流，只是这个流中的元素都是延迟生成出来的，下一个值只有在使用者通过 next 方法获取的时候，才会计算出来。这就让使用者可以决定究竟要在序列中生成多少个元素。例如，我们可以只获取这个序列的第一个元素，或者某个元素的前缀。

延迟计算序列的一个问题就是有些变换操作需要对整个序列进行计算。例如，对于之前我们定义的 standardIn，它从标准输入逐行读取输入的字符串，这是一个只能单次遍历的序列。我们可能希望只打印某些符合特定条件的行。通过函数式风格表达的话，我们很自然地就会想到用 filter 来实现。但 Sequence 中 filter 的返回值是个数组（这也就意味着它不再具有延迟计算的特性了），因此这个实现只有在处理完所有从标准输入读取的内容之后，才会返回结果。换句话说，这个结果已经不是延迟计算出来的了。下面的代码会打印输入中所有多于三个字符的行，但结果只有在标准输入收到了文件末尾 (end-of-file, EOF) 信号之后才会显示出来：

```
let filtered = standardIn.filter {
    $0.split(separator: " ").count > 3
}
for line in filtered { print(line) }
```

但显然，我们希望每读到一行符合条件的结果，就打印一个消息。一个可行的解决方案就是采用下面命令式编程的风格：

```
for line in standardIn {
    guard line.split(separator: " ").count > 3 else { continue }
    print(line)
}
```

如果我们希望使用函数式的风格，用方法的串联替代掉 for 循环，我们必须在获取到值的时候就生成筛选后的结果，而不是等读取了所有的输入之后再处理。换句话说，我们需要一个延迟序列。标准库为 Sequence 提供了一个 .lazy 属性帮助我们实现这样的行为。因此之前的例子可以改成这样：

```
let filtered = standardIn.lazy.filter {
    $0.split(separator: " ").count > 3
}
for line in filtered { print(line) }
```

.lazy 的返回值类型是 LazySequence<Self>。这也就意味着 standardIn.lazy 的类型是 LazySequence<AnySequence<String>>。LazySequence 会在内部存储原始的序列，但它不会对其进行额外的处理。然后，LazySequence 的 filter 方法会返回一个 LazyFilterSequence<AnySequence<String>>。它会在内部存储原始的序列以及 filter 的谓词函数。

这也就意味着，在上面的例子中，随着我们遍历 filtered，从标准输入中读到的每一行都会立即处理，我们可以在控制台立即看到符合条件的输入，而不是等到 EOF 信号之后统一处理。这样一来，也就不会创建保存所有输入结果的临时数组了。

另外，当需求发生变化时，观察这种编程方式下代码的变化也很有意思。例如，让我们只打印第一个输入大于三个字符的行。使用 lazy，我们完全不需要修改 filtered 的定义，只要访问它的 first 属性就好了：

```
let filtered = standardIn.lazy.filter {
    $0.split(separator: " ").count > 3
}
print(filtered.first ?? "<none>")
```

如果没有 lazy，上面的代码就会从标准输入读入所有的行，直到它收到 EOF 信号。然后再从中找到第一个符合条件的元素。使用 lazy，它就只会读到第一个满足条件的输入，之后的标准输入就不再处理了。如果要读取前两个符合条件的输入，我们可以把 first 改成 prefix(2)。相比于修改命令式编程的代码，修改这些函数式风格的代码通常都要更简单和清晰。例如，下面就是同一个功能两种实现方式的比较，一个是 lazy 的版本，另一个则是命令式编程的版本：

```
// 函数式
let result = Array(standardIn.lazy.filter {
    $0.split(separator: " ").count > 3
}.prefix(2))
```

```
// 命令式

var result: [String] = []
for line in standardIn {
    guard line.split(separator: " ").count > 3 else { continue }

    result.append(line)
    if result.count == 2 { break }
}
```

## 集合的延迟处理

当在一个常规集合类型（例如：Array）上串联多个操作的时候，可以延迟处理的序列会更有效率。如果你更倾向于函数式编程，你可能会写出下面这样的代码：

```
/*_*/(1..<100).map { $0 * $0 }.filter { $0 > 10 }.map { "\($0)"}
```

习惯函数式编程之后，上面的代码就会显得很清晰并易于理解。但是，它却存在着效率不佳的问题：每一次调用 `map` 和 `filter` 都会新建一个包含中间结果的数组，并且，这个数组在函数返回的时候就被销毁了。通过在这个调用链的开始插入 `.lazy`，就不会产生任何保存中间结果的数组，代码的执行就会更有效率：

```
/*_*/(1..<100).lazy.map { $0 * $0 }.filter { $0 > 10 }.map { "\($0)"}
```

在 Swift 5.0 中，不启用编译器优化的条件下，使用了 `.lazy` 的代码可以比之前的版本快三倍，而使用 `-O` 开启优化之后，性能可以提升八倍。

`LazyCollectionProtocol` 扩展了 `LazySequence`, 它要求实现它的类型也是一个实现了 `Collection` 的类型。在一个延迟加载的序列里, 我们只能“按需”逐个生成序列中的每个元素。但在一个集合里, 我们可以直接“按需”生成指定的某个元素。例如, 当你对一个延迟加载的集合应用 `map` 方法之后, 通过下标访问其中的某个元素, `map` 就只会对你访问的那个结果执行变换(在这里例子中, 就是第 51 个元素):

```
let allNumbers = 1..1_000_000  
let allSquares = allNumbers.lazy.map { $0 * $0 }
```

```
print(allSquares[50]) // 2500
```

但是，延迟加载的集合也不总是能在性能方面胜出，有些情况你就需要特别注意。当使用下标访问元素的时候，每一次结果都是计算出来的。根据集合在获取结果时进行的计算，下标操作符可能就不是一个  $O(1)$  操作了。例如，考虑下面这个例子：

```
let largeSquares = allNumbers.lazy.filter { $0 > 1000 }.map { $0 * $0 }
print(largeSquares[50]) // 2500
```

在打印语句执行前，filter 和 map 不会进行任何实质性的工作。由于对 allNumbers 进行了过滤，因此，largeSquares 中的第 51 个元素和 allNumbers 中是不一样的。为了找到正确的元素，每次像这样 [50] 访问的时候，filter 都必须计算出原始序列中的前 51 个元素，显然，这不是一个常量时间的操作。

## 回顾

Sequence 和 Collection 协议构成了 Swift 集合类型的基础。它们提供了很多通用的操作，并且可以作为你自己的泛型函数的约束。而 MutableCollection 或 RandomAccessCollection 这样专门的集合类型，则为你按照需求和性能要求实现自己的算法时，提供了细粒度级别的控制。

高层级的抽象会使模型变得复杂，这是正常现象，所以如果你无法立即明白所有东西，也不要气馁。想要适应严格的类型系统，需要大量的练习。理解编译器想要告诉你的信息会是一门艺术，你需要仔细阅读每一行提示。作为回报，你得到的是一个非常灵活的系统，它可以处理任何东西，从一个指向内存缓冲区的指针到一组可以被消耗的网络封包流，都能从类型系统中获益。

这种灵活性的意义还在于，一旦你抓住了模型的本质，在未来很多代码一眼看上去就会非常熟悉。因为它们是基于相同抽象构建的，并支持相同的操作。你在创建自定义类型时，如果这个类型适用于 Sequence 或者 Collection 的框架，那你就应该考虑添加这些支持。这在之后会让你和你的同事都倍感轻松。

# 并发

12

Swift 5.5 中，基本的并发特性被添加到了语言中。它允许我们在编译器的强健支援下编写并发代码，这让一整类 bug 不再可能发生。

**async/await** 是最重要的变化，它让我们能够对异步代码使用像是内建错误处理这样的 Swift 结构化编程技术，从而让我们好像是在写同步代码一样。和 completion handler 相比，**async/await** 要更加简单，也更容易理解。

在本章中，我们将以加载来源于网络的 [Swift Talk](#) 的每一集的剧集 (episode)、合集 (collection) 和相关数据作为例子。我们预先定义了遵守 Codable 和 Identifiable 的 Episode 和 Collection 结构体。比如，Episode 类型如下：

```
struct Episode: Identifiable, Codable {  
    var id: String  
    var poster_url: URL  
    var collection: String  
    // ...  
  
    static let url = URL(string: "https://talk.objc.io/episodes.json")!  
}
```

使用 **async/await** 从网络加载剧集数据并将它们作为 JSON 解析的方式如下：

```
func loadEpisodes() async throws -> [Episode] {  
    let session = URLSession.shared  
    let (data, _) = try await session.data(from: Episode.url)  
    return try JSONDecoder().decode([Episode].self, from: data)  
}
```

同样的例子，不使用 **async/await**，而是使用 completion handler 的话：

```
func loadEpisodesCont(  
    _ completion: @escaping (Result<[Episode], Error>) -> ())  
{  
    let session = URLSession.shared
```

```
let task = session.dataTask(with: Episode.url) { data, _, err in
    completion(Result {
        guard let d = data else {
            throw (err ?? UnknownError())
        }
        return try JSONDecoder().decode([Episode].self, from: d)
    })
    task.resume()
}
```

注意，在第一个 (使用 `async/await` 的) 例子中，执行顺序是自上而下的，这和普通的结构化编程是一样的。在数据从网络加载的过程中，这个函数被挂起 (suspend)了。然而，在第二个 (使用 `completion handler` 的) 例子中，执行顺序进行了分叉：一部分代码会在 `task.resume()` 之后继续执行，而另一部分代码 (`completion handler` 中的部分) 则在网络请求完成 (或者失败) 的时候异步地运行。观察这两个函数的类型，可以看到第一个函数一定会返回一个数组或者抛出一个错误。而 `completion handler`，我们只能依靠约定来假设 `completion handler` 会且仅会被调用一次。

一个 `completion handler` 也可以被称为一个续体 (continuation)。这个名字描述了 `completion handler` 的目的：它是在 `loadEpisodesCont` 完成工作后，让程序继续执行的点。类似地，`session.dataTask` 也不会返回数据，它也会调用提供的续体 (在这里，是闭包表达式所提供的)。在底层，`async/await` 也使用续体。在上例中，`await` 后面的部分就是续体。

Swift 中的并发编程系统还仅处于起步阶段。在未来会有更多功能加入，现有的特性也会得到完善和改良。此外，一些编译期间的检查也还没有完全实现。为了确保你的代码在未来也可用，可以添加下面的 Swift 编译器标志：`-Xfrontend -warn-concurrency` 和 `-Xfrontend -enable-actor-data-race-checks`。当你使用了不安全的构建方式时，编译器会给你有用的警告。

## Async/Await

在 `async/await` 以前，我们通过 `completion handler` (续体) 和 `delegate` 来编写非阻塞的代码。这在 Apple 的生态系统中已经是多年的标准实践。但是，这种模型可能会导致深层嵌套，难以追踪的续体。更糟糕的是，我们无法使用像是 Swift 内建的错误处理和 `defer` 等这些标准的结构化编程的组件。

当我们对比上面两个代码例子时，有几件事情是一目了然的。首先，`async/await` 的代码要更短 (三行，对比 `completion handler` 方法的十行)。第二，`completion handler` 更难阅读；代码执行的顺序不是从上倒下，续体部分会在函数返回之后再执行。最后，当使用 `completion handler` 写代码时，更容易犯错误，比如我们很容易会在错误发生时忘记调用 `completion handler`。只看函数的类型的话，并不清楚 `completion handler` 会以怎样的频率调用；你必须查阅文档，并且相信文档没有过时。

本质上说，`async/await` 的版本更短，也更容易懂。它同时也更加清晰；通过查看类型，我们知道这个方法要么返回一个错误，要么返回一个剧集的数组，并且它会且只会返回一次。

将 `async/await` 的代码进行组合，要比组合 `completion handler` 的代码容易得多，这个差异在需要考虑错误处理的时候更加明显。比如说，让我们把前面的例子扩展一下，让它下载第一集的海报图片。我们需要做的唯一的事情就是在函数中添加另一行：

```
func loadFirstPoster() async throws -> Data {
    let session = URLSession.shared
    let (data, _) = try await session.data(from: Episode.url)
    let episodes = try JSONDecoder().decode([Episode].self, from: data)
    let (imageData, _) = try await session.data(from: episodes[0].poster_url)
    return imageData
}
```

对比一下，使用 `completion handler` 来向代码添加加载第一张海报图片要困难得多。我们现在必须非常小心地对错误处理进行处理，以确保它的正确。如果我们无法加载一开始的数据，或者出现解析错误，我们需要保证立刻用这个错误调用 `completion handler`。如果没有错误，我们就可以继续加载海报图片，并在完成时调用 `completion handler`。

使用 completion handler 的实现要复杂得多：这个实现的正确性并不明显（我们是不是在所有不同情况下都只调用了一次 completion handler?），而且追踪控制流也困难得多。如果我们确实犯了错误（比如没有调用 completion handler），编译器并不会对此向我们发出警告：

```
func loadFirstPosterCont(_ completion: @escaping (Result<Data, Error>) -> ()) {
    let session = URLSession.shared
    let task = session.dataTask(with: Episode.url) { data, _, err in
        do {
            guard let d = data else {
                throw (err ?? UnknownError())
            }
            let episodes = try JSONDecoder().decode([Episode].self, from: d)
            let inner = session.dataTask(with: episodes[0].poster_url) { data, _, err in
                completion(Result {
                    guard let d = data else {
                        throw (err ?? UnknownError())
                    }
                    return d
                })
            }
            inner.resume()
        } catch {
            completion(.failure(error))
        }
    }
    task.resume()
}
```

很明显，async/await 的代码不论在实现上还是接口上，都要比使用 completion handler 的代码简单得多，编译器可以把整类 bug（虽然不是全部）都消除掉。和 completion handler 相比，async/await 还有一个好处，我们可以使用 defer 来执行那些想要在退出当前函数作用域时完成的操作。不过，要注意 defer 语句中你是不能使用并发代码的。比如，你不能在 defer 语句里异步地更新一个模型对象。

## 异步函数是如何执行的

想要理解异步函数的执行方式，我们可以把一个异步函数分割成不同的部分，其中每个部分都由一个潜在的暂停点 (**suspension point**) 划定。换言之，我们在每个 `await` 语句后把函数拆分开。比如之前的函数，现在可以标记为三个部分：

```
func loadFirstPoster() async throws -> Data {  
    // 第一部分  
    let session = URLSession.shared  
    let (data, _) = try await session.data(from: Episode.url)  
    // 第二部分  
    let episodes = try JSONDecoder().decode([Episode].self, from: data)  
    let imageData = try await session.data(from: episodes[0].poster_url).0  
    // 第三部分  
    return imageData  
}
```

如果我们使用 `completion handler` 来重写我们的函数，那么这些部分就对应原来函数中的每个同步代码块。在底层，Swift 将包含暂停点的每个异步函数重写为续体。上面函数的第一部分将正常运行，但是第二和第三部分都是续体。

当执行上面的代码时，“第一部分”将会同步执行。一个工作单元被叫做一项作业 (**job**)。然后这个函数被暂停，它会等待从网络中加载数据。这个加载会在另一个单独的作业中完成。一旦数据被加载，`loadFirstPoster` 继续，一个新的作业会被调度运行，并负责同步地执行标记为“第二部分”的代码。为了加载图片的数据，这个函数被再次暂停，并进行一个新的作业。当这个作业结束时，我们的函数将会继续并返回它的值。

Swift 的并发模型被叫做协同式多任务 (**cooperative multitasking**)。简言之，这意味着函数永远不应该阻塞当前的线程，而是应该自愿地暂停。函数只能在 (被 `await` 标记的) 潜在暂停点才能暂停。当函数被暂停时，并不意味着当前的线程被阻塞了：相反，此时控制权会被交还给负责线程安排的调度器，在此期间，(对应于其他任务的) 其他作业可以运行在这个线程上。在稍后的时间点上，调度器通过调用函数的续体，来让该函数恢复运行。比如，在上面的函数中，函数在“第二部分”之前暂停，数据被从网络加载进来。在此期间，其他的作业可以被执行，一旦数据可用，这个函数就将继续。

注意，一个被暂停的函数并不承诺在继续时会使用它原来的线程。Swift 运行时为异步任务维护了一个线程池，当这些线程池中的线程可用时，作业会被添加到其中并得以运行。一个异步函数具体运行的线程不应该很重要（通常也不重要），除非我们谈论的是主线程，而主线程依然需要特殊对待的。

作业之间发生的线程跳动 (hopping) 意味着我们不能假定线程本地值在暂停点的前后会是一致的。因此，使用线程本地值的 API，比如 Foundation 中的 `Progress class`，是无法和异步函数兼容的。在并发中，线程本地存储的替代品是任务本地存储 (`task-local storage`)。我们在本章中不会讨论这个内容，不过你可以在相应的 [Swift 进化提案](#) 中阅读这方面的细节。

不过，“当前执行线程”并不是在暂停点会发生变化的唯一事项。更一般地，你必须假设所有非本地状态都发生了变化，因为其他代码有机会在函数被暂停期间执行（非本地状态包含了全局变量以及当 `self` 没有值语义时它上面的属性）。这可能会造成一些难以发现，且编译器也无法捕获的 bug。我们会在 [Actor 重入](#) 的部分再回到这个话题。

并发模型被称为协作式的原因，是因为它依赖单独的函数协同工作：不能有函数在等待昂贵的 I/O 操作或者执行长时间任务时去阻塞线程。函数需要通过其他针对 I/O 的异步函数来让它们自身暂停，或者在长耗时工作之间通过调用 `Task.yield()` 来让其他作业得到执行的机会。

`async/await` 最适合进行 I/O 相关的等待，因为 Swift 的并发系统是针对高效的任务切换来设计的。相比起切换到另一个线程或者创建一个新线程，将一个函数暂停，然后在空闲出来的线程上继续另一个函数的做法要快得多。如果一个函数能够在等待慢速 I/O 的时候不占用线程，那么系统就能在几乎没有额外开销的情况下让 CPU 核心保持忙碌。

并发系统也支持取消，这也是协作式的。我们会在 [本章稍后](#) 讨论关于取消的内容。

你可能会注意到在我们的例子中，`JSONDecoder().decode` 的调用并非异步的，这是因为这个 API 还没有异步的变体版本。如果我们想要处理大量的数据，解码步骤有可能会耗时数百毫秒甚至更长，在这里段时间内，我们的函数在并发世界中就表现不佳了，因为它并没有让其他任务有机会运行。

说到这里，短暂地占用线程并不是世界末日，特别是当 CPU 正在做一些有用的事情的时候（比如 JSON 解码）。除非所有的函数在同一时间“行为错乱”（占用线程），否则调度器依然可以把作业分配给其他的 CPU 内核：协同式线程池就是按照使用大致和 CPU 核心数相同的线程数量来设计的。不过，请记住，调度器并不会为了保持程序响应而去创建新线程，因为这有可能导致线程爆炸（**thread explosion**），而它正是 GCD 中性能问题的来源之一。反过来说，过多的非协作函数可能会造成整个线程池被挂起，因此我们应该尽可能避免阻塞。

理论上，在两个暂停点之间的每个函数部分，都将会被作为一个独立的作业同步执行。每次对异步函数的调用也会创建一个独立的作业，它有可能被执行在一个不同的 actor 上。然而，有时候多个作业会被“融合”（fusion）到一起，作为一个较大的不中断的作业进行执行。

比如，Foundation 在 `FileHandle` 上有一个 `bytes` 属性，可以用来从一个文件中以异步方式读取字节。它的实现使用了 `AsyncBytes` 类型。`AsyncBytes` 满足 `AsyncSequence` 协议（它是 `Sequence` 协议的异步变体），它使用 `next` 方法返回单个字节。`next` 方法的大致实现如下：

```
mutating func next() async {
    if !buffer.isEmpty {
        return buffer.removeFirst()
    } else {
        return await reloadBufferAndNext()
    }
}
```

因为有作业融合，只要在缓冲区非空的情况下，对 `next` 方法的调用将等效于普通的同步调用；在缓冲区被读空的时候，这个方法才会停止，并只在读取到更多数据后才会返回。虽然我们在调用 `next` 的时候必须写 `await`，但是它真的只是一个潜在暂停点：在大多数情况下，缓冲区都不为空，所以当前的任务也就不会暂停。

## 和 Completion Handler 对接

在已有项目中，你可能会有一些代码已经使用了 completion handler。这可以使你自己的代码，第三方的代码，或者是 Apple 自己的还没有更新到 `async/await` 的代码。使用

withCheckedContinuation (或者三个近似相关的变体其中之一)，你可以把任意的带有 completion handler 的函数包装成异步函数。

比如，让我们假设存在这样一个带有 completion handler 的函数，用来加载某一集：

```
func loadEpisodeCont(id: Episode.ID,  
    _ completion: (Result<Episode, Error>) -> () {  
    // ...  
}
```

要把这个方法转成异步 API，我们可以创建一个异步函数，并将对原来函数的调用包装到 withCheckedThrowingContinuation 中去。后者会立即执行它的函数体，然后暂停，直到我们在函数体中调用 cont.resume 才会继续：

```
func loadEpisode(id: Episode.ID) async throws -> Episode {  
    try await withCheckedThrowingContinuation { cont in  
        loadEpisodeCont(id: id) {  
            cont.resume(with: $0)  
        }  
    }  
}
```

因为我们原来 loadEpisodeCont 方法中的 completion handler 可能会接收到错误值 (也就是 Result 的 .failure 成员)，我们需要把这个异步包装方法标记为 throws，并使用 withCheckedThrowingContinuation 而非 withCheckedContinuation。“Checked”这个单词表明这两个函数都会进行一些运行时的检查，来确保我们对 resume 的调用进行且仅进行了一次。对它进行多次调用会产生一个运行时错误。如果我们在调用它之间就把这个续体丢弃了的话，我们也会在运行时收到警告。

withUnsafeContinuation 和 withUnsafeThrowingContinuation 时另外两种变体，它们在运行时要稍微高效一些，因为相比起 checked 的版本，它们跳过了这些安全检查：你必须确保续体恰好被调用了一次。没有调用续体将导致任务永远无法继续，而调用超过一次则进入未定义行为。在编写代码时，最好使用带有检查的版本，如果你真的需要这点性能，也请确保在转换成不安全版本之前，进行仔细的检查。

除了将已有的基于 completion handler 的代码进行对接之外，with[...]Continuation 函数在其他方面也很有用。本质上来说，它允许你手动暂停一个任务，然后在稍后的时间点恢复它。根据我们的经验，除开最简单的场景之外，你需要特别小心，才能以正确的方式使用它。举个例子，标准库中的 AsyncStream 类型就使用了 withUnsafeContinuation 来在等待更多项目产生的时候，将 stream 流的消费者进行暂停。

除了把 loadEpisode 写作异步函数以外，我们也可以把它写成一个异步的 init：

```
extension Episode {  
    init(id: Episode.ID) async throws {  
        self = try await withCheckedThrowingContinuation { cont in  
            loadEpisodeCont(id: id, cont.resume(with:))  
        }  
    }  
}
```

当你想要从 Objective-C 中访问这些异步方法时，会存在一些限制。你可以把异步方法标记成 @objc 来使它们在 Objective-C 中以 completion handler 的形式可见。但是你不能把 init 标记成 @objc 变体，因为异步的初始化方法是没有桥接的。Swift 中还有一个更一般的限制，那就是所有的全局函数都不能被标记为 @objc，所以它们也必须作为一个已经对 Objective-C 可见的类的方法，才能被暴露出去。

## 结构化并发

在上面我们已经看到 async/await 是如何让异步代码结构化的了，这让我们可以使用 Swift 内建的控制流构件：条件、循环、错误处理，以及 defer 这些语句，都和同步代码中的工作方式一样。然后，async/await 本身其实并没有引入并发，也就是说，多个任务并不能同时进行。为了实现并发，我们需要能够创建任务的方式。

## 任务

任务 (task) 是 Swift 并发模型的基本执行上下文。每一个异步函数都是在一个任务中执行的（那些被异步函数调用的同步函数也是如此）。任务的作用，大致上和传统多线程代码中的线程是

一样的。和一个线程类似，一个任务自身是没有并发的，它每次只能运行一个函数。当运行中的任务遇到了 `await` 时，它会暂停当前的运行，放弃自己的线程，并把控制权交回给 Swift 运行时的调度器。之后调度器可以在相同线程上运行另一个任务。当轮到第一个任务继续运行时，这个任务会精准地从它离开的地方继续开始（有可能在不同的线程上）。

## 子任务和非结构化任务

当我们使用 `await` 调用一个异步函数时，这个被调用的函数将会和调用者运行在同一个任务中。想要创建一个新任务，必须要进行明确的动作。我们可以创建的任务分为两类：

- **子任务** — 这类任务构成了结构化并发的基础。我们使用 `async let` 或者任务组这两种方式其中之一，来创建子任务。子任务被按照树形结构来组织，并且有作用域和生命周期。我们会在下面详细对子任务进行讨论。
- **非结构化任务** — 这些是单独的任务，它们会成为一个一棵独立的任务树的根节点。我们通过调用 `Task` 的初始化方法或者 `Task.detached` 工厂方法来创建一个非结构化的任务。非结构化任务的生命周期和当前任务的生命周期是独立的。我们会在非结构化并发部分讨论非结构化的任务。

## 任务树

结构化并发的核心思想，是把结构化编程的思想（清晰的控制流以及作用域规定的生命周期）扩展到并发运行的多个任务中去。要达到这一点，Swift 并发的做法是将任务按照树形结构进行组织，以及为这些任务制定生命周期规则：

- **子任务相互之间可以并发运行** — 当前任务可以创建多个并行运行的子任务，这个当前任务将成为子任务们的父任务。父任务和子任务们也是并发运行的。
- **子任务的寿命不能超过父任务** — 和本地变量在定义它的作用域结束时就会无效一样，每个子任务的生命周期也都被限制在了创建它的作用域里。父任务只有在等待所有子任务都完成后，才能退出它的作用域，就像一个同步函数只有在它所调用的所有函数都返回以后它本身才能返回一样。
- **取消会从父任务向下传递给子任务** — 这确保了单次取消就能够取消它下方的完整的任务树，而无论这棵树有多深。

这些规则的结果是，一个异步函数可以暂时地分支成多个并发执行的子任务，同时又将这些并发限制在当前作用域中。反过来，函数的作者可以确信，当函数返回时，将不会再有悬空的资源（也就是运行中的任务）会被保留下来。

子任务也会继承其父任务的任务优先级（除非明确地重写了这个优先级）以及任务本地值。由于任务树中各个任务之间的依赖关系是已知的，如果有一个高优先级的父任务正在等待它的子任务，调度器可以调高这些子任务（以及可能存在的子任务的子任务）的优先级。

## async let

async let 是创建一个子任务的最快捷的语法。我们继续 [Async/Await](#) 中的例子，下面的函数会从网络并发地加载剧集和合集：

```
// loadCollections has the same structure as loadEpisodes above.
```

```
func loadCollections() async throws -> [Collection] { ... }
```

```
func loadEpisodesAndCollections() async throws -> ([Episode], [Collection]) {
    async let episodes = loadEpisodes()
    async let collections = loadCollections()
    return try await (episodes, collections)
}
```

async let episodes = loadEpisodes() 语句创建了一个异步绑定（**asynchronous binding**）。当执行到这一行时，运行时会创建一个新的子任务并在这个任务中执行 loadEpisodes() 的调用。子任务会立即开始运行。同时，父任务也继续运行 — 注意，虽然 loadEpisodes 是一个异步函数，我们也不需要写 await。在下一行，我们开始了第二个子任务，用它来运行 loadCollections()。

接下来，父任务会等待异步绑定并从这些子任务中收集结果。在我们的例子中，这就是 try await (episodes, collections) 表达式，编译器要求我们在这里写上 await（以及如果合适的话 try）来承认父任务可能会在子任务完成之前进行暂停。一般来说，因为父任务和子任务是并发运行的，在开始等待子任务之前，父任务还可以执行其他（同步或者异步）的工作。在我们的例子中，父任务没有其他事情要做了，所以它立即开始了对子任务的等待。

我们来总结一下异步绑定最重要的一些规则：

\*\*第二次访问一个 `async let` 值并不会让它再次暂停。\*\*一旦一个父任务等待过某个异步绑定，后续的访问将立即得到值。再次读取一个异步绑定值并不会让子任务再次开始，也不会让父任务暂停（虽然你还是必须在两个地方都写上 `await`）。比如，下面的代码只会暂停一次，而两次打印出的数字是相同的：

```
// 慢速 IO 的随机数发生器，所以使用异步。  
func requestRandomNumber() async -> Int { ... }  
  
async let rand = requestRandomNumber()  
await print(rand) // 暂停  
await print(rand) // 不暂停，不重跑
```

\*\*异步绑定在类型系统中并不拥有独自的表述。\*\*和其他很多语言不一样，Swift 并没有把 `async let` 绑定建模为一个 Future 或者 Promise 值。`async let episodes` 的类型是 `[Episode]`，而不是 `Future<[Episode]>` 或者像是 `episodes: async [Episode]` 这样的东西。这是一个 内部限制：如果子任务返回的是 `Future` 值，那么程序员就将可以把这些 `Future` 传来传去，从而导致它们存在逃逸出当前作用域的可能性，这会打破结构化并发所做出的关于生命周期的保证。

\*\*没有被 `await` 的 `async let` 将会在作用域结束后被隐式地取消并等待。\*\*对没有明确 `await` 的子任务进行等待是必要的，这是为了满足结构化并发子任务不能在父任务之外存活的承诺。父任务在退出作用域之前，必须要允许它所创建的任何尚未完成的子任务有机会完成。下面的例子是一段根据用户输入在两个页面之间进行导航的程序。为了最小化等待时间，我们创建两个子任务来预先加载两个页面的内容，而同时我们在第三个子任务中等待用户的行为。当用户输入到达时，我们只需要等待两个子任务中的一个：

```
func waitForUserInput() async -> NavigationAction { ... }  
func loadPage(at pageIndex: Int) async -> Page { ... }  
func navigate(currentPage: Page) async -> Page {  
    async let nextAction = waitForUserInput()  
    async let nextPage = loadPage(at: currentPage.index + 1)
```

```
async let previousPage = loadPage(at: currentPage.index - 1)
switch await nextAction {
    case .nextPage:
        return await nextPage
        // 隐式取消和等待 previousPage
    case .previousPage:
        return await previousPage
        // 隐式取消和等待 nextPage
}
}
```

在 switch 语句的每个分支中，没有被等待的那个子任务将在函数返回之前被隐式取消以及等待。注意，取消并不意味着子任务会迅速完成，因为和暂停一样，取消在 Swift 的并发模型中也是协作式的。在本章后面我们还会谈到取消特性。顺带一提，任务组 (task group) 的方式 (我们在下一节会介绍) 在退出作用域时在取消上展现出略微不同的行为：任务组在正常退出时 (也就是非抛出状态) 是不会隐式地取消那些未等待的子任务的。

就算一个异步绑定中包含多次异步调用，它也只会创建一个任务。在等号右边的整个表达式将会被包装成一个任务。比如，下面的代码创建了一个子任务，其中包含了两次按顺序进行的异步函数调用：

```
// 只有一个子任务
async let (num1, num2) = (requestRandomNumber(), requestRandomNumber())
await print(num1) // 等待 num1 和 num2。
await print(num2) // num2 已经可用。
```

就算我们的代码只等待了子任务中的一个部分，系统也总是会将整个子任务作为单一单元进行等待。也就是说，上面代码中的 await num1，即使 num1 会更早可用，但整个函数还是会在子任务全部完成之前处于暂停状态。正因如此，await num1 其实也让 num2 可用了，所以接下来的 await num2 就不再需要暂停了。

和前面的代码形成对比，下面的代码将并行执行两次函数调用，因为它们存在于不同的子任务中：

```
// Two concurrent child tasks
async let num3 = requestRandomNumber()
async let num4 = requestRandomNumber()
await print(num3) // Awaits only num3.
await print(num4) // Awaits only num4.
```

对于在编译期间就已经能确定子任务数量的情况来说，`async let` 是一种方便且轻量的语法。对于动态数量的子任务，我们需要另一种工具：任务组。

## 任务组

任务组提供了动态数量的并发，也就是说，子任务的数量在运行时才能确定。下面的例子中，这个函数负责并行地按照一个剧集数组来下载海报图片：

```
func loadPosterImages(for episodes: [Episode]) async throws
    -> [Episode.ID: Data]
{
    let session = URLSession.shared
    return try await withThrowingTaskGroup(of: (id: Episode.ID, image: Data).self)
    { group in
        for episode in episodes {
            group.addTask {
                let (imageData, _) = try await session.data(from: episode.poster_url)
                return (episode.id, imageData)
            }
        }
        return try await group.reduce(into: [:]) { dict, pair in
            dict[pair.id] = pair.image
        }
    }
}
```

我们可以通过 `withTaskGroup(of:returning:body:)` 或者 `withThrowingTaskGroup` 来创建一个任务组，它们的区别在于子任务是否具有可抛出的行为（Swift 的类型系统无法适用单一的函数来表达这两种情况）。`withTaskGroup` 函数同时接受子任务的结果类型以及一个为我们提供 `TaskGroup` 实例的闭包作为参数。注意，调用 `withTaskGroup` 并不会创建子任务，它的闭包依然是运行在父任务里的。

之后我们通常会在一个循环里为每个想要创建的子任务调用 `TaskGroup.addTask {...}` (或者 `addTaskUnlessCancelled`)。子任务将立即开始以任意顺序执行。我们传递给 `addTask` 的闭包，需要返回一个值，来作为这个子任务的结果。但是注意，我们并不会立即获得这个结果。任务组会把所有子任务的结果收集起来，并把它们作为一个 `AsyncSequence` 提供给我们。在我们的例子中，我们为任务组上调用 `reduce` 来把结果存储到一个字典里，不过我们也可以用异步的 `for` 循环来做这件事：

```
return try await withThrowingTaskGroup(of: (id: Episode.ID, image: Data).self)
{ group in
    // ...
    var result: [Episode.ID: Data] = [:]
    for try await (id, imageData) in group {
        result[id] = imageData
    }
    return result
}
```

这种使用一个循环发起子任务，然后跟着另一个循环来收集或者处理结果的任务组闭包结构非常典型。它类似于常见的 MapReduce 算法：一个父节点把工作分成小块并将这些小块分配给独立的工作节点。这些工作节点传回它们的结果，然后父节点再把它们合并成一个总的结果。我们可以认识到，结构化的并发模式是有选择地引入并发，并将其影响包含在其中。

想一想如果使用 `completion handler` 来实现 `loadPosterImages` 的难度吧。我们在这里就不展示它的代码了，但是我们将我们的尝试放到了 [GitHub 上](#)。这个版本想要正确实现，有两个主要的难点。首先，我们需要找到一种方法来收集并发运行的 `URLSession` 任务的结果。当这些“子任务”结束并调用它们各自的 `completion handler` 时，我们把它们的结果添加到一个共享的本地数组中去。我们必须通过一个串行的派发队列（或者一个锁）来保护数组，以免两个 `completion handler` 同时尝试访问数组时所造成的数据竞争（我们在会Actor一节中谈到更多

有关数据竞争的话题)。第二，我们需要等到所有的网络任务都完成后，再延迟地调用函数的 completion handler。我们通过使用一个 DispatchGroup 来追踪还有多少“子任务”仍然在运行中。我们不能进行同步等待，因为这将会阻塞当前线程，所以我们只能把等待行为派发到另一个独立的派发队列里去。而在结构化并发的版本中，任务组会为我们处理所有这些需要手动进行的记录和同步。

接下来，和 async let 一样，我们会谈到任务组中最重要的一些规则：

\*\*子任务的结果是按照完成顺序传递的，而非提交顺序。\*\*这也是为什么在我们的例子里，返回的是一个从剧集 ID 到图片数据的映射字典，而不是一个简单数组的原因。当然，这个函数也可以将任务组送来的子任务结果先重新排序后再返回。

\*\*子任务的结果类型必须全部相同。\*\*因为 TaskGroup 在传递结果时必须拥有单一的元素类型，所以这也成为了一个要求。如果你需要运行不同的具体结果类型的子任务，有三种选择：

0. 如果子任务的数量是固定的，那么使用 async let。
1. 对于不同元素类型的子任务，使用多个任务组。
2. 定义一个通用的可以表示所有结果的类型，比如每个成员都代表一种子任务类型的枚举。

\*\*添加到任务组中的子任务存活时间不能超过任务组闭包的范围。\*\*如果在退出任务组闭包时依然有未经等待的子任务，运行时会在继续前隐式等待这些任务直到它们结束(并且把结果丢弃掉)。和之前一样，这遵循了结构化并发中子任务的生命周期和它的作用域绑定的规则。

\*\*和 async let 不同，任务组不会在退出时取消那些尚未等待的子任务(除非任务组是以抛出错误的方式终结的)。\*\*如果我们不想要这个行为，就必须调用 TaskGroup.cancelAll() 来手动取消所有剩余的任务。默认情况下不取消，可以让那些返回为 Void 的任务使用起来更简单：

```
await withThrowingTaskGroup(of: Void.self) { group in
    for document in modifiedDocuments {
        group.addTask { try await document.save() }
    }
    // 隐式等待所有子任务。
}
```

这个任务组并不去收集子任务的结果 (而且结果类型本身就是 Void)，但任务组闭包在全部子任务完全前依然不会退出。注意，任务组会默默地把子任务抛出的错误丢弃掉，所以对于可能会失败的任务，这不是很好的做法。下面这个修改后的例子对错误进行了检查，不过从语义上来说它也明显不同了：

```
try await withThrowingTaskGroup(of: Void.self) { group in
    for document in modifiedDocuments {
        group.addTask { try await document.save() }
    }
    // Explicitly await child tasks, catch errors.
    for try await _ in group {
        // Do nothing.
    }
}
```

for try await 循环的目的是捕获子任务抛出的错误，并把它向上传递。当任务组通过抛出而退出时，它将会像 async let 那样，隐式地取消所有剩余的任务。如果这不是我们想要的行为，我们可以通过在子任务的闭包里处理这些错误，而不让它们传递到任务组中的方式来阻止取消；单纯在任务组闭包中处理它们是不够的，因为 AsyncSequence 在迭代时会在遇到第一个错误时就停下，这样我们就可能会错过余下的其他错误。

\*\*任务组不会去限制并发的数量。\*\*如果我们用一个包含 500 个剧集的数组调用 loadPosterImages 函数，它会产生 500 个子任务，从而导致可能会有 500 个网络请求在同时运行。为了性能着想，最好还是把能同时并发进行的子任务数量限制在一个较小值会比较好。但现在 TaskGroup 中并没有提供能限制它的并发“宽度”的 API。不过我们可以自己来实现它：从比较少的一些子任务开始，创建后立即等待它们。之后，每当任务组收到一个结果，我们就开始另一个子任务，直到全部输入都被处理完。在任务组正在产生结果时向其中添加新任务是被可行而且已经被预想过做法。

## Sendable 类型和函数

传递给 TaskGroup.addTask 的闭包类型是

@escaping @Sendable () async -> ChildTaskResult。这里的 @Sendable 标志表述的是这个

函数是会穿越并发执行上下文的，它必须做到并发安全（在这里，函数穿越了父任务和子任务的执行上下文）。Sendable 闭包有一些编译器负责检查的限制，当它捕获状态时，必须保证这个行为不会引起数据竞争：

→ 任何被捕获的值它们自己也必须是 Sendable 的。Sendable 是一个不含有任何要求的空协议，我们把这类协议称为标记协议（marker protocol）。某些类型可以声明它们满足 Sendable。那些非 public 的结构体和枚举，只要它们所有的组件都满足 Sendable 的话，这些类型本身也会隐式满足 Sendable。Actor 在默认情况下也是 Sendable 的，因为它们就是被设计出来在并发访问期间保护自己的状态的。

对于可变的或者没有标记为 final 的 class，编译器无法检查在不同并发域之间共享状态是否安全，所以编译器拒绝让它们满足一个简单的 Sendable 协议。不过，我们依然可以通过编写 class C: @unchecked Sendable 这样的代码来让一个 class 成为 Sendable，这样就可以跳过编译器的检查。不过，现在就变成由我们来确保这个 class 是线程安全的，比如通过锁来保护所有的状态访问。

→ 虽然 Swift 默认情况下会把捕获的状态当作引用，但这里所有的捕获都必须按值来处理。对于声明为 let 的不可变值的捕获默认就是按照值处理的；对于其他所有的捕获，必须明确作为值写进捕获列表里。

第二条规则其实排除了所有的可变状态捕获。比如，我们的任务组代码如果采用下面这种构想，就会因为子任务闭包捕获了父任务的可变状态，而无法编译通过：

```
return await withThrowingTaskGroup(of: (id: Episode.ID, image: Data).self)
{ group in
    var result: [Episode.ID: Data] = [:]
    for episode in episodes {
        group.addTask {
            let (imageData, _) = try await session.data(from: episode.poster_url)
            // 错误：捕获的 var 'result' 在并发执行的代码中发生改变。
            result[episode.id] = imageData
        }
    }
    return result
}
```

这里我们尝试在多个子任务中并发修改 `result` 字典，这会导致数据竞争。这个问题和 `completion handler` 版本的 `loadPosterImages` 函数中的问题是一样的。不过，和之前不同，闭包上的 `@Sendable` 标注会让编译器立刻捕获到这个错误。我们已经看到过正确的编写方式了：对子任务的结果进行第二次循环迭代，因为这个循环是运行在父任务上的，所以它可以自由地修改本地状态。

同样的 `@Sendable` 限制对 `async let` 也起作用。你可以把一个 `async let` 赋值语句的右侧看作是被 `@Sendable` 所包装并运行在另一个上下文的代码，因此它不能访问那些非 `sendable` 的状态，也不能对被捕获的变量进行修改。

Swift 5.6 中，编译器还不能捕获全部 `Sendable` 可能被违反的情况。这是因为（包括 Apple 的框架在内的）已经存在的库，还没有针对并发安全进行检查和标注，所以编译器无法知道哪些类型可以跨越并发域进行安全传递。如果仅仅是因为 Foundation 中没有标注，而使得在 `sendable` 函数中只是使用像是 `Date` 或者 `Data` 这样的类型就导致错误的话，开发者几乎就没有办法去适配 `async/await` 了。

我们在本章开头介绍过 `-Xfrontend -warn-concurrency` 这个编译器标记，它会开启对于违反 `Sendable` 的更严格的编译器检查。这些额外的警告可以帮助理解异步函数何时会运行在一个不同的并发上下文，以及编译器是如何阻止数据竞争的。我们认为开启这个警告会是好主意，就算你最终决定关掉它来减少噪声，但至少可以在编写异步代码的时候打开它。通过一个不做检查的扩展，为第三方的类型追加 `sendable` 也是可能的，比如：

```
// 因为 Date 就是个值，所以应该是安全的。  
extension Date: @unchecked Sendable {}
```

但是要注意，这种做法本质上是不安全的，因为我们告诉了编译器对这个类型禁用并发检查。不过如果你想让编译器保持沉默，这也是一个临时的解决方案。

在写作本文时，关于 `Sendable` 检查的整个话题都还处于不断变动中。最终，在 Swift 6，违反 `Sendable` 的地方会报错。在转换期间（因为我们需要等所有库都检查过，所以这可能需要数年），可以遵循迁移路径，它允许模块的作者让他们的代码既兼容并发前，也兼容并发后的世界；同时让用户可以选择性地按照模块来禁用警告。用户侧可以使用 `@preconcurrency import A` 指令来禁用模块 A 中类型的并发警告。编译器十分聪明，一旦模块 A 使用 `Sendable` 标注进行了更新，这些警告就会重新浮出水面。

## 取消

拥有取消一个异步操作的能力，在几乎所有程序中都是很常见的需求。再一次，和错误处理一样，在传统的基于 completion handler 的代码中，想要实现取消是很困难的。用 Async/Await 部分的 `loadFirstPosterCont` 函数为例，想要支持取消，我们所需要的变更有：

- 为调用者提供一种方法，来让它能够发出信号表示操作已经取消。我们可以通过返回一个线程安全的“取消令牌”来做到这一点，它提供一个公开的 `cancel` 方法，并且允许我们查询取消状态。`URLSessionDataTask` 本身并不适合，它确实拥有一个 `cancel` 方法，但是一旦网络任务完成，它就不再能被取消了，所以我们无法使用它来可靠地取消内部的下载任务。
- 当取消令牌接受到取消请求时，取消掉 `URL Session` 的操作。这通常是通过传递一个在取消时运行的闭包给取消令牌来完成的。需要额外的记录来传递内部下载任务给这个取消处理程序。
- 注意不要在任务已经被取消后去开始新的任务。在我们的例子中，我们应该在（可能会很耗时的）JSON 解码步骤后插入一个手动的对取消状态的检查，并在需要的时候提早返回。
- 确保在取消时向调用者返回一个合适的错误码。`URLSession` 在取消时会给出 `URLError.Code.cancelled` 作为错误码；我们必须为手动的取消检查开发出我们自己的错误表示方式，并可能要将 URL 错误转换成更适合我们需求的形式。

结果得到的代码非常杂乱，因为代码中很大篇幅都是模板化的处理取消和错误的内容，所以想要理解代码的意图会非常困难。

对应的异步代码天生就支持取消，可以对比看看：

```
func loadFirstPoster() async throws -> Data {  
    let session = URLSession.shared  
    let (data, _) = try await session.data(from: Episode.url)  
    let episodes = try JSONDecoder().decode([Episode].self, from: data)  
    return try await session.data(from: episodes[0].poster_url).0  
}
```

当这个函数所在的任务被取消时，这个函数也会被错误打断。我们下面马上来仔细看看它的工作原理。

## 取消是协作式的

早先我们已经看到 Swift 的并发系统是协作式的，这一点对取消也同样适用：除非在任务中运行的代码周期性地检查取消状态，并在需要时提前结束，否则取消一个任务将不会带来任何影响。对于这套系统来说，一个很重要的设计目标是为函数提供机会，让它们能在取消时进行清理。这也是系统不能单纯地中断一个被取消的任务的原因。

取消始终是以任务为单位进行的，你不能取消一个任务中的某一个特定函数。如果你能访问到一个非结构化任务的 Task 对象，那么你就能通过调用它的 cancel 方法来取消这个任务。你也可以用这样的代码来取消当前任务：

```
withUnsafeCurrentTask { task in
    task?.cancel()
}
```

在最基本的层面上，取消一个任务所做的事情，仅只是设置任务元数据上的一个标志。某个函数如果想要支持取消，它就需要偶尔对 Task.isCancelled 进行检查，或者去调用 try Task.checkCancellation()，后者会在 isCancelled 是 true 时直接抛出一个何时的错误。为我们的示例函数加上这些检查后，代码如下：

```
func loadFirstPoster2() async throws -> Data {
    try Task.checkCancellation()
    let session = URLSession.shared
    let (data, _) = try await session.data(from: Episode.url)
    try Task.checkCancellation()
    let episodes = try JSONDecoder().decode([Episode].self, from: data)
    try Task.checkCancellation()
    return try await session.data(from: episodes[0].poster_url).0
}
```

这并不是太坏，但是在我们的例子中其实不太必要，因为我们调用的 URLSession API 已经在内部进行了类似的取消检查。如果任务被取消了，那么当前活跃的下载将会停止并抛出错误，这个错误会被我们的函数传递给调用者。取决于我们所期望下载的数据模型有多大，在两次网络请求之间的 JSON 解码的步骤可能会成为问题。因为 JSONDecoder 现在还不支持取消，所以如果 JSON 解码花费太久，比如 500 毫秒，在接收到取消信号后，我们的函数依旧会持续运行这么长的时间，这一点并不理想。为了解决这个问题，我们可能需要使用一个带有取消支持的 JSON 解码器，或者自己编写一个。

值得注意的是，就算 JSONDecoder 是一个完全同步的 API，它也是可以支持取消的。同步函数也能用同样的 Task.isCancelled 或 try Task.checkCancellation() 调用来检查取消，如果同步函数是内嵌在一个异步上下文中时，这些方式将会按正确的方式运行；而如果在同步上下文中的话，它们会返回默认值（比如 false）。

总之，在常见情况下，我们编写的函数大部分时间其实都在调用一些已经处理了取消的（系统）API。只要我们把取消错误传递给调用者，基本上可以什么都不用做，就能获取对于取消的支持。不过，如果我们在执行长时间运算，或者按顺序调用一些无法取消的 API，我们就应该为代码加上手动的取消检查。例如，下面的函数可能会计算大量的随机数。我们在循环中每迭代一千次就添加一个取消检查：

```
func makeRandomNumbers(in range: ClosedRange<Int>, count: Int) throws
    -> [Int]
{
    var result: [Int] = []
    result.reserveCapacity(count)
    for i in 1...count {
        // Check for cancellation periodically.
        if i.isMultiple(of: 1000) {
            try Task.checkCancellation()
        }
        result.append(Int.random(in: range))
    }
    return result
}
```

如果是在异步函数中，我们也会想要以同样的方式定期调用 `Task.yield()`，以便让运行时有机会在我们的线程上安排其他任务运行。

## 使用错误路径

按照惯例，一个函数如果检测到自己被取消了，它应该抛出 `CancellationError` 这个标准库中的新类型。调用者可以通过检查这个错误来把取消操作和其他错误区分开来。使用这个取消错误的好处是，它不会引入新的控制流路径，但是要注意，对这个特定错误的使用只是一个惯例：调用者不能完全依赖这个类型。比如，`URLSession` 在被取消时会继续按照以前的方式抛出 `URLError.Code.cancelled` 错误。如果我们不想把这个 `URLSession` 的行为暴露给调用者的话，我们可以捕获这个“出错”的错误类型，把它替换成 `CancellationError`：

```
func loadFirstPoster3() async throws -> Data {
    do {
        let session = URLSession.shared
        let (data, _) = try await session.data(from: Episode.url)
        let episodes = try JSONDecoder().decode([Episode].self, from: data)
        return try await session.data(from: episodes[0].poster_url).0
    } catch URLError.Code.cancelled {
        // Replace URLError.Code.cancelled.
        throw CancellationError()
    } catch {
        // Propagate all other errors.
        throw error
    }
}
```

非抛出的函数也可以参与到取消中来。当然了，它们不能抛出错误，所以函数的作者必须要选择一个恰当的“空”返回值，比如 `nil` 就是一种选择。在大多数情况下，让函数可被抛出也许是更好的选择。取消的函数甚至也可以返回部分结果：比如，`makeRandomNumbers` 在被终止时，也可以返回直到它侦测到取消时为止的那部分被填充的数组。如果你决定这样做，请确保清楚地把这个行为写进文档，因为对调用者来说，接收到部分结果可能是出乎意料的事情。

## 取消和结构化并发

到目前为止，我们已经看到了单个任务中的取消，但是取消系统的真正威力，在于它和结构化并发的整合。取消信号会自动沿着任务树向下流动：当一个父任务被取消时，它的子任务会看到它们的 `isCancelled` 标志被切换为了 `true`。同样地，这不会让子任务立即停止，每个任务都要负责自己执行定期的取消检查。此外，取消必须尊重结构化并发的基本规则，也就是子任务不能超出父任务的生命周期，被取消的父任务会继续保持运行，直到它的所有子任务都完成为止。

取消会沿任务树向下传递，意味着取消一个顶层任务（比如对用户操作的响应）时，我们不需要任何代码，也不论为此工作的子任务被嵌套得有多深，它们都可以被终止。这是非常强大的特性，不过只有当我们使用结构化并发并且我们在设计函数时考虑了取消，才能发挥作用。对于非结构化的任务，我们只能手动取消。

我们用另一个例子来说明为取消设计代码的重要性，让我们编写一个函数，来运行一个带有超时处理的异步函数：

```
func asyncWithTimeout<R>(  
    nanoseconds timeout: UInt64,  
    do work: @escaping () async throws -> R  
) async throws -> R {  
    return try await withThrowingTaskGroup(of: R.self) { group in  
        // 开始实际工作  
        group.addTask { try await work() }  
        // 开始超时计时  
        group.addTask {  
            try await Task.sleep(nanoseconds: timeout)  
            // 超时  
            throw CancellationError()  
        }  
        // 第一个到达的胜出，取消其他任务  
        let result = try await group.next()  
        group.cancelAll()  
        return result  
    }  
}
```

```
}
```

这里是一个使用例子，它以一秒为超时界限加载剧集图片。如果下载任务用时超出一秒，这个函数将以 CancellationError 终止：

```
let imageData = try await asyncWithTimeout(nanoseconds: 1_000_000_000,  
do: loadFirstPoster3)
```

在 `asyncWithTimeout` 中，我们使用了一个任务组来开始两个并发子任务：一个进行实际工作，另一个按照需要的超时时间进行休眠。然后我们开始等待第一个完成的子任务。如果负责实际工作的任务“获胜”的话，我们就把它的结果返回，并取消掉超时任务。如果超时任务更快完成，那么 `group.next()` 将会抛出 `CancellationError`。因为任务组由于错误退出了，它将会隐式地将未完成工作任务取消掉。

在获取第一个结果后的 `group.cancelAll()` 调用十分重要，这是因为任务组并不会在成功完成时隐式取消掉还没有等待的子任务。如果没有 `cancelAll()`，那么任务组就会在返回前默默等待负责超时的任务执行到结束。忘掉这一行是一个非常常见的错误，因为它并不会改变你的程序的输出结果，而只会影响性能。实际上，Swift Evolution 中结构化并发提案的作者就犯了好几次这个错误。

`asyncWithTimeout` 同样为我们展示了函数支持协作式取消模型的重要性。如果超时“胜出”了，那么任务组就会把工作任务取消掉，不过之后它会等待工作任务返回。如果工作任务并没有检查取消（就像上面 `JSONDecoder` 例子那样），而且要需要很长时间才能完成的话，`asyncWithTimeout` 所完成的时间就会比指定的超时时间长得多了。

## 取消处理程序

`withTaskCancellationHandler(operation:onCancel:)` 函数让我们可以为当前任务指定一个取消时运行的处理程序（cancellation handler）。这个函数接受两个闭包：一个是运行在当前任务中的异步操作（工作）；另一个则是在当前任务被取消时会被立即调用的取消处理程序。这段处理程序的目的，是为了在取消时执行一些清理代码，比如说释放资源或者把取消信号传递给那些和 Swift 并发系统尚未整合的其他对象。

比如，Foundation 框架中很可能就用了取消处理程序来实现基于 `async/await` 的 `URLSession` API。在这个例子中，`onCancel` 闭包的任务会负责把取消事件转发给 `URLSession` 系统，这样网络请求就可以被取消掉。

要实际实现这一点，可能会比听起来更困难些，因为取消处理程序和被取消的任务是运行在不同并发上下文的。特别地，取消处理程序和主操作是无法访问共享的可变状态的 (`onCancel` 的闭包被特地用 `@Sendable` 进行了标注，来提醒编译器)。我们需要用一个带有手动同步机制（锁或者派发队列）的类来在两个上下文之间以线程安全的方式共享 `URLSessionDataTask` 实例。这里我们也不能使用 `actor`，因为不管是取消处理程序还是续体闭包，都不是 `async` 的。代码太长这里放不下，不过你可以在 [GitHub 上查看它们](#)。

## 非结构化并发

有时，限制在一个范围内的结构化并发的特征反而会变成一种障碍。我们需要另一种工具，来让我们可以退出结构化并发，并开启一个能够存活超过当前作用范围的新任务。。这个工具就是 `Task.init(priority:operation:)` 初始化方法。我们通常使用尾随闭包的语法来调用这个初始化方法：

```
let task = Task {  
    return try await loadEpisodesAndCollections()  
}
```

调用 `Task.init` 会开启一个新的独立任务，它的生命周期不和当前作用范围绑定。这个任务会立即开始，并和原来的任务并发运行。新任务会变成另一棵任务树的根节点，一直运行到任务结束。由于这个新任务和原来的任务并没有父子关系，所以我们将这类并发叫做**非结构化并发**。

非结构化的任务不会随着原任务的取消而取消。初始化方法所返回的 `Task` 实例，可以让我们手动进行取消，或者使用 `[try] await task.value` 来获取任务的结果（只有当任务闭包可以 `throw` 时才需要 `try`）。因为当我们尝试获取结果时，任务有可能还没有完成，所以 `value` 属性是 `async` 的。如果你对其他支持异步的编程语言有所了解的话，`Task` 实例就是 Swift 版本的 `future` 或者 `promise` 对象。

注意，在一个结构化的并发上下文中（比如在一个任务组里）创建非结构化任务，通常来说是反模式的行为，因为它让一些操作能够逃离结构化并发的世界。但这并不是说创建一个非结构化

的任务一定是不好的：对于那些需要超过当前范围的并发工作来说，这个特性显然是必要的。但是失去自动取消，错误传递以及寿命管理的代价是巨大的。此外，非结构化并发由于任务层级和代码结构的关联不再明显，因而失去了清晰度，我们也需要为此买单。结构化并发所施加的约束是有其设计目的的，它让我们的代码更容易理解；除非我们绝对需要，否则不应该破坏它们。

Swift 运行时会在所有任务完成前都持有一个对它的强引用，所以我们不需要仅仅只是为了存活，去把任务变量存储在某个地方。另外，传递给 Task.init 的闭包在强引用 self 时并不需要显式地写明 self.。这是通过一个新的非官方编译器属性 \_implicitSelfCapture 来实现的。之所以不要求 self，是因为大部分任务不会创建永久的引用环，因为这些任务最终是会完成的，那时这些被捕获的变量也会被释放。如果我们创建的是一个有可能永远不会完成的任务的话（比如一个将接收到的通知作为 AsyncSequence 发布出去的 NotificationCenter 观察者），我们还是应该小心翼翼地把那些可能会导致引用环的捕获变量标记为 weak。

## 非结构化任务和游离任务

通过 Task { ... } 启动的非结构化任务，会继承原来上下文的优先级（除非明确进行了覆盖）、任务本地值、以及最重要的，actor 隔离域。所以，如果一个开启任务的函数正运行在某个指定的 actor 上的话，新任务也会被隔离到那个 actor 中。通常这就是想要的效果，因为这可以让非结构化任务访问到它周围的被 actor 隔离的状态，就像下面例子中这样：

```
@MainActor class ViewModel: ObservableObject {  
    @Published var counter = 0  
  
    func incrementAsync() {  
        Task {  
            // Task 可以访问到 actor 隔离的状态  
            counter += 1  
        }  
    }  
}
```

对于 actor 隔离，我们会在本章稍后的 [Actors](#) 一节中谈论更多内容。注意如果一个任务闭包调用了其他 actor 隔离（比如从 @MainActor 中调用了其他没有 @MainActor 标注的 actor）中的

异步函数，那这个任务也会转到其他 actor 中去。“普通”的没有任何 actor 所属的异步函数通常就在当前的 actor 上下文中运行，但是这个行为还并没有被确定。在书写本章时，Swift 团队进行正式规则的确立，以便我们总能静态地就知道函数会在哪个上下文中执行。

如果我们不想要新任务运行在当前的执行上下文中的话，我们可以转而用 `Task.detached { ... }` 发起一个游离任务。游离任务在很大程度上与非结构化任务很像，它们拥有相同的 API，但是它不会继承当前任务的优先级、任务本地值以及 actor 隔离。

你可能会疑惑，相比于非结构化任务，到底什么时候应该使用游离任务呢？我们的建议是，把 `Task { ... }` 作为默认选项，只有在特殊原因想要离开当前 actor 上下文时，才选用 `Task.detached`。比如其中一个场景是，如果正在 `@MainActor` 上（稍后会更多介绍），而且想要启动一个不应该占用主线程的任务时，应该选用 `Task.detached`。

在本章剩余的部分，我们会使用“非结构化任务”笼统地指代非结构化和游离任务。除非我们特别地对 actor 隔离进行讨论，否则可以忽略它们之间的不同。

## Starting Async Work from Synchronous Contexts

### 在同步上下文中开始异步工作

非结构化任务可以在任意上下文中开启，这也包括那些非异步的函数。实际上，`Task { ... }` 和 `Task.detached { ... }` 是在一个同步上下文调用异步函数的唯一方法。我们有时候会说 `async` 标记具有“传染性”，因为将一个函数标记为 `async`，会要求它的所有调用者也变成 `async`。

所有被执行的 `async` 函数，在当前任务树的根部必定有一个非结构化的任务节点。这个任务可能隐藏在第三方库或者 Swift 运行时中，但是它就在那里。下面的例子是我们能写出的最简单的 `async/await` 程序之一：

```
@main struct Program {
    static func main() async throws {
        print("Sleeping for one second")
        try await Task.sleep(nanoseconds: 1_000_000_000)
        print("Done")
    }
}
```

```
}
```

当我们把可执行的 `main()` 函数声明为 `async` 时，编译器会生成一个从 Swift 运行时进行的 `_runAsyncMain` 调用。如果你查看[这个函数的源代码](#)，会发现在那里实际调用的是 `Task.detached`。

## 非结构化任务的错误处理

如同我们在[错误处理](#)一章中谈到的那样，让忽视错误变得困难，是 Swift 的基础设计目标之一：我们必须当场处理错误，或者明确地将它转发给我们的调用者，然后让调用者处理或传递它。不幸的是，非结构化并发的 API 为错误处理打开了一类新的漏洞，让我们在某些情况下非常容易忽略错误。作为例子，考虑下面的非结构化任务，它负责将程序的当前状态保存到磁盘中去：

```
Task {  
    try await save()  
}
```

如果 `save` 抛出错误，这个错误将会被默默丢弃掉，因为没有任何代码等待了这个任务的结果。理想情况下，编译器应该可以检测到这个问题，并强制我们处理错误的情况。但我们甚至连个警告都不会收到，这是因为 `Task` 的初始化方法被声明为了 `@discardableResult`，它允许我们丢弃掉返回值。如果强制写成 `_ = Task { ... }` 的话会稍好一些，至少它让任务的“触发后就不用再管”的特性能够变得明确。

## Actor

不论你使用的是老式的线程操作，还是 GCD 队列，又或者 Swift 任务，并发编程都必须要处理共享资源的潜在冲突这一固有问题。只要代码会被并发执行（以时间共享的方式或者并行的方式），你就必须保证保护共享资源，比如对象上的属性、一般性质的内存、文件或者数据库连接等。

传统上，我们为此使用过各种各样的锁相关的 API，而最近，GCD 串行队列已经成为确保共享资源无冲突访问的一种流行方式。随着 Swift 原生的并发模型被引入，这门语言获得了一种全新的资源隔离机制：actor。

## 资源隔离

actor 是引用类型，它通过只允许在 `self` 中对可变属性进行直接访问（对于 `let` 声明的常量，可以安全地跨越 actor 边界进行访问），以及把它的函数部分用一个串行执行上下文进行隔离，来保护状态。有一点需要特别记住，actor 并不是把整个方法的执行进行隔离：一般规则是，只有那些暂停点之间的函数的部分被视为原子操作。对于没有使用 `await` 调用其他异步函数的方法这种特殊情况，这个方法会被作为单独一个作业执行，整个方法成为一个原子操作（我们在可重入的部分谈论更多这方面的话题）。

actor 的访问模型提供了资源隔离，用来避免数据竞争。如果两个线程在同一时间尝试访问同一位置的内存，并且其中之一是写操作的话，就会发生数据竞争。例如，如果读取线程在另一个线程的写入操作进行到一半时，从相关内存中读取数据的话，可能得到不合理的数据。更糟糕的是，如果两个线程同时向同一个内存位置写，就会造成数据损坏。

actor 为它们的属性提供免于数据竞争的保护。不论是读取还是写入，上面所描述的 actor 的访问模型都会严格地限制两个线程同时对一个属性进行访问的可能性。不过，要注意的是，使用 actor 并不会让你的并发代码神奇地自动正确，在竞态条件 (race condition) 这个更广泛的范畴中，数据竞争只是其中一种类型。

当我们的代码需要依赖多个线程顺序执行时，竞态条件就会发生。在 actor 的上下文中，我们可以说，假如 actor 的正确行为需要依赖多个被隔离函数的执行顺序的话，那么就会发生竞态条件。

虽然上面提到的每个函数部分的隔离特性能够避免数据竞争，但是我们无法控制多个函数部分在暂停点的执行顺序。举个例子，在 `await` 之前修改 actor 上的一个属性，然后在 `await` 之后依然依赖该属性拥有同样的值，就是一个竞态条件。竞态条件的后果取决于具体情况，但是它可能会造成意料之外的结果，或者甚至数据损坏（这个数据损坏发生在一个更高层级上，而不是像数据竞争的情况那样发生在实际的内存位置中）。

由竞态条件造成的 bug，常常会被叫做海森堡bug (heisenbugs) (谐音梗，来源于“海森堡测不准”以及“不确定性原理”)，由于它们的不确定性，这些 bug 极难被修复。因此，就算在 actor 中，也要小心设计以避开竞态条件的可能性。我们会在可重入的部分解释为什么 actor 不选择保护你免受竞态条件困扰的原因。

有了关于 actor 能够提供以及不能提供的安全特性的基本了解，我们就可以来看看这个例子了。设想我们要实现一个网络爬虫的队列：

```
class CrawlerQueue {  
    var pending: [URL] = []  
    var finished: [URL: String] = [:]  
}
```

让我们假设网络爬虫使用若干个并发的 worker 来处理处于等待状态的 URL，我们想要在所有的 worker 之间共享这个爬虫队列。一个 worker 应该能够从队列中获取到一个等待爬取的 URL，把它新发现的 URL 添加到队列里，然后把处理过的 URL 的爬取结果存储起来。这正对应了 CrawlerQueue 实例的两个属性 pending 和 finished，在多个并发 worker 之间共享的资源，必须要进行免于数据竞争的保护。

使用派发队列的话，我们可以这样来实现所需要的同步机制：

```
class CrawlerQueue {  
    private var pending: Set<URL> = []  
    private var finished: [URL: String] = [:]  
  
    private var queue = DispatchQueue(label: "crawler-queue")  
  
    public func getURL() -> URL? {  
        queue.sync {  
            self.pending.popFirst()  
        }  
    }  
  
    public func enqueue(_ url: URL) {
```

```
queue.async {
    guard self.finished[url] == nil else { return }
    self.pending.insert(url)
}

}

public func store(_ contents: String, for url: URL) {
    queue.async {
        self.finished[url] = contents
    }
}
}
```

所有的属性现在都是队列所私有的，它们只能通过公开的方法访问到。而公开方法都是用了一个私有的串行派发队列来保护 pending 和 finished 属性，让它们免于数据竞争。

这个队列用 actor 的实现看起来很相似，只是去掉了那些将操作派发到私有串行队列的代码：

```
actor CrawlerQueue {
    var pending: Set<URL> = []
    var finished: [URL: String] = [:]

    public func getURL() -> URL? {
        pending.popFirst()
    }

    public func enqueue(_ url: URL) {
        guard finished[url] == nil else { return }
        pending.insert(url)
    }

    public func store(_ contents: String, for url: URL) {
        finished[url] = contents
    }
}
```

```
    }  
}
```

`pending` 和 `finished` 属性默认就可以被 `actor` 的方法访问到 (属性是被隔离在 `actor` 的执行上下文中的), `actor` 的每个方法也都运行在 `actor` 的串行执行器中, 这确保了我们不会在属性访问上遇到数据竞争。

## 可重入

与其在 `CrawlerQueue` 上提供单独的方法来获取 URL、把新 URL 加入队列、以及为某个 URL 存储结果, 不如让我们来提供一个统一的 API, 这样我们就不会不小心忘记调用其中的某一个了:

```
actor CrawlerQueue {  
    var pending: Set<URL> = []  
    var finished: [URL: String] = [:]  
  
    func process(_ handler: (URL) async -> (contents: String, links: [URL])) async {  
        guard let url = pending.popFirst() else { return }  
        let (contents, links) = await handler(url)  
        finished[url] = contents  
        for link in links {  
            guard finished[link] == nil else { continue }  
            pending.insert(link)  
        }  
    }  
}
```

`process` 方法在被调用时会接受一个异步函数作为参数。这个函数接收要处理的 URL, 然后以字符串和一个要加入队列的新 URL 数组的方式返回结果。

`process` 被明确标记为 `async`, 要是不这么做, 我们就不能使用 `await` 来调用这段异步处理 `handler` 了。当我们调用 `await handler(url)` 时, 关于如何处理, Swift 有两种并不完美的处理

方式：要么它可以在等待 handler 结果的同时，让其他代码能够执行；要么它可以在 handler 没有返回之前，阻止 actor 上其他代码的执行。

对于后一种情况，我们会很容易就进到死锁 (deadlock) 里。比如，只要我们在 handler 函数里调用了 CrawlerQueue actor 上的任何一个其他方法，执行就将被挂起。由于这个原因，Swift 选择了可重入 (reentrancy) 作为 actor 的默认行为：一旦当前作业被暂停 (当我们调用 await handler(url) 时)，其他代码可以在作业继续前运行在这个 actor 上。这样就算 handler 函数在内部对 actor 进行另一次调用，代码的执行依然可以继续。

由于可重入是默认的，所以对 actor 的状态在暂停点前后相同这一假设是不可靠的，我们要确保不依赖这个假设。同样，我们也不能假定一个异步方法会被作为原子操作来执行。比如，当 process 在等待 handler 完成时，我们的作业队列会处于一种特殊状态：正在被处理的 URL 已经被从 pending URL 中移除了，但是它的结果还没有被添加到已完成作业的字典中。因此，会有一段窗口期，导致虽然这个 URL 已经正在被处理了，但同样的 URL 被别的作业再一次添加到 pending URL 里。

只有在 actor 的 process 方法可重入的时候，这个问题才可能表现出来。如果 actor 的方法不可重入，那么在 process 方法执行的时候，actor 上不会有其他代码能干涉到 actor 的状态，我们也就不用担心在暂停点等待时 actor 的状态会发生改变了。

为了纠正这个问题，我们可以向队列添加第三个属性，来追踪正在被处理的 URL：

```
actor CrawlerQueue {
    var pending: Set<URL> = []
    var inProcess: Set<URL> = []
    var finished: [URL: String] = [:]

    func process(_ handler: (URL) async -> (String, [URL])) async {
        guard let url = pending.popFirst() else { return }
        inProcess.insert(url)
        let (contents, links) = await handler(url)
        finished[url] = contents
        inProcess.remove(url)
        for link in links {
```

```
guard finished[link] == nil, !inProcess.contains(link) else { continue }
    pending.insert(link)
}
}
}
```

一旦 URL 进入到队列中，它就总会被以某种方式追踪到：要么在 pending 里，要么正在被处理，要么已经结束。这样，process 就可以以可重入的方式安全执行了。

## Actor 性能

对外部世界来说，actor 的所有公开方法都是异步方法：它们必须通过 await 进行调用，以便让 actor 能切换到它自己的执行上下文中。比如，下面的函数连续调用了两个 actor。我们把这称为任务在 actor 之间进行了跳跃 (hop)：

```
// Counter 是 actor
let counter1 = Counter()
let counter2 = Counter()

func incrementAll() async {
    await counter1.increment()
    await counter2.increment()
}
```

actor 跳跃不可避免地存在一些开销，但 Swift 的并发系统经过设计，以确保 actor 跳跃要比切换线程或者队列派发快得多，所以你不必太担心这里的性能损耗。因为调用 actor 方法的任务（在我们的例子中，就是执行 incrementAll 函数的任务）必须等待 actor 方法完成，所以 actor 可以跑在任务的当前线程上。如果 actor 在调用发生时没有在进行其他工作，那么 actor 跳转就只是在 actor 上设置一个标记，把自己“锁定”起来，让别的调用者不能使用，然后执行一个普通的函数调用。当方法返回时，actor 又被“解锁”。但在竞争存在时（比如 actor 正在执行其他任务），上下文的切换就会变得更昂贵。在这种情况下，我们的任务必须在等待 actor 可用期间，暂停并放弃当前线程。如果你能够把你的程序设计成大多数时候 actor 都免于竞争的话，你就能得到最好的性能特性。

当向 main actor 切换或者从 main actor 切换出来时，actor 跳跃的开销也会变大。这是因为 main actor 是运行在主线程上的，从另一个执行上下文(通常是运行在协同式线程池的一个线程上)切换到 main actor，或者反过来从 main actor 切换到一个一般的 actor 时，会涉及到相对昂贵的线程切换。

## Main Actor

有时候你并不需要为了同步某些状态的访问而去创建自己的 actor。特别是对 GUI 程序来说，通常你会想要某个特定的方法或者属性仅在主线程被调用和访问。由于主线程(或者在 GCD 术语中的主队列)在 Apple 平台上一直扮演着特殊的角色，它现在也以全局 actor 的方式被暴露出来了，也就是 MainActor。在底层，main actor 使用主线程作为它的串行执行上下文。

全局 actor 的特殊之处在于它可以被作为标签使用，来标记其他的类型、属性或者函数。比如，不需要把 CrawlerQueue 自身定义为一个 actor，我们也可以使用系统定义的 @MainActor 标签来确保它的方法和属性访问都只发生在主线程上：

```
@MainActor
final class CrawlerQueue {
    var pending: Set<URL> = []
    var inProcess: Set<URL> = []
    var finished: [URL: String] = [:]

    func process(_ handler: (URL) async -> (String, [URL])) async {
        // ...
    }
}
```

当然，和基于 actor 的版本相比，这个爬虫队列的实现会在主线程上做更多的工作，但特别是如果要把变更发布到 UI 上的话，这么做是非常有用的。

将整个类型标记为 @MainActor，相当于把它上面的所有属性个方法一个个单独进行标记。上面的例子在语义上等效于下面这样：

```
final class CrawlerQueue {
```

```
@MainActor var pending: Set<URL> = []
@MainActor var inProcess: Set<URL> = []
@MainActor var finished: [URL: String] = [:]

@MainActor func process(_ handler: (URL) async -> (String, [URL])) async {
    // ...
}
```

有时候你会遇到这样的情况，在整个类型中，你想要把除了一两个成员之外的部分都标记为 @MainActor。除了完全手动一个个进行标注之外，我们也可以采取相反的做法：把整个类型标记为 @MainActor，然后用 nonisolated 关键字为特定的方法或者属性取消这个标记。比如，我们可以使用这项技术来为其余部分都被 MainActor 隔离的爬虫队列添加一个没有被隔离的初始化方法：

```
@MainActor
final class CrawlerQueue {
    nonisolated init(_ initialURLs: [URL]) {
        self.pending = Set(initialURLs)
    }
    // ...
}
```

大多数情况下，@MainActor 标签做的事情都符合你的预期：它确保被标记的方法或者属性只会在主线程上被访问。不过，有一些微妙的边缘情况值得思考。具体来说，有三种不同情况我们需要考虑：使用 @MainActor 标注 `async` 方法，使用它标注非 `async` 方法，以及使用它标注属性（我们可以把它想象为非 `async` 的 `getter` 和 `setter`）。

0. 异步的 @MainActor 方法：当把一个 `async` 方法标注为 @MainActor 时，我们可以肯定这个方法中的代码会运行在主线程的。编译器会强制我们在调用这个 `async` 方法时使用 `await`，这给方法一个机会切换到 main actor 的执行上下文中。就算是该方法是从 Objective-C 代码进行的调用，由于 `async` 方法在 Objective-C 中会被暴露成带有 `completion handler` 的形式，运行时也可以进行执行上下文的切换。

1. **非异步的 @MainActor 方法。**当我们用 `@MainActor` 标记一个非 `async` 的方法时，情况会稍为复杂一些。编译器依然会强制我们只能在 `main actor` 的执行上下文直接调用这个方法，从其他上下文进行调用的话需要加上 `await`。然而，因为方法是同步的，它不能在运行时再被派发到正确的执行上下文中。所有的检查都必须在编译期间完成，而这种检查可能会在 Objective-C 代码中失效。

这个问题的一个常见例子，发生在我们让一个 `class` 遵守某个协议，并将协议的方法标记为 `@MainActor` 时。我们可能会假设这能够确保该方法始终运行在 `main actor` 上，但事实并非如此。比如，就算你将一个原本应该在 `URLSession` 的私有队列上调用的 `URLSessionDelegate` 里的协议方法标记为 `@MainActor`，也并不会改变它所运行的线程，而且编译器无法对此做出任何警告。

2. **@MainActor 属性：**对于属性进行标记的 `@MainActor` 执行的也是编译期间的检查。如果我们尝试从一个其他执行上下文访问 `main actor` 隔离的属性，编译器会阻止我们这么做。和 `main actor` 隔离的同步方法调用不同，我们不能在属性访问前插入 `await` 来运行上下文切换。我们必须确保在访问这些属性时已经处于 `main actor` 的执行上下文中；否则，编译器就会给出错误。和同步方法一样，需要记住编译器对于 `main actor` 隔离的属性的检查是有可能失效的。

## 推断执行上下文

当你在阅读那些用 GCD 实现并发的 Swift 代码时，你基本上是通过“在脑海中”执行这些代码，并在运行时追踪它们的派发路径，来推断哪些代码是执行在哪个队列的。此外，你会需要查看 Apple 或者第三方 API 提供的文档，来确认 `completion handler` 或者是 `delegate` 方法究竟是在主队列还是其他什么队列被执行的。

在这个模型中，你当前所在的队列是在运行时被决定的。如果你只是看某一小块特定代码（如果你看的代码在此刻并没有自己进行派发的话），你就无法断定这些代码被执行时会运行在哪个队列上。

有了 Swift 的 `actor` 隔离函数模型，这个推理过程就不再只是单纯的运行时行为了。你可以通过检查一个 `async` 函数所在的词法范围，来确定这个函数会运行在哪个 `actor` 执行上下文中。如果一个方法存在于 `actor` 中，你就知道不论在运行时这些代码处在什么样的环境里，它们都会在这个 `actor` 的执行上下文中以隔离的方式运行。同样，任何一个被（像是 `@MainActor` 标签这样）`actor` 隔离 `async` 函数，都可以确保是在指定的 `actor` 中隔离运行的。如果 `async` 函数没

有明确地以这两种方式之一进行 actor 隔离，那么它最终就会运行在一个没有关联任何 actor 的泛用执行器上，[这个提案](#)对此进行了描述。

不过，如果你处理的是一个同步函数，那么执行上下文依然是运行时决定的：你必须要追溯可能的调用栈，找到最近的一个 `async` 函数，才能知道它所在的执行上下文到底是哪个。

## 回顾

并发模型是从 Swift 2 引入协议扩展以来这门语言中最大的改变。`async/await` 让我们可以用与同步代码相同的风格和正常的语言控制流结构来编写异步代码。结构化并发则将已经拥有 60 年历史的结构化变成方式应用到并发代码中。`actor` 可以通过串行访问内存或其他资源的方式，来防止数据竞争。

这些概念本身都不算新颖，但 Swift 把它们整合成了条理清晰的并发模型，并且将它们深深融入到语言中。这使得编译器可以检查出许多常见的并发错误，以此带来更少的 bug 和更安全的程序。

当我们写这些内容时，Swift 5.5 才刚出来几个月。有些东西还在变化之中，我们自然也还没有太多在实际产品中使用这个新的并发模型的经验。在我们书写和介绍本章中新特性的同时，我们也在持续地学到新东西。我们可以合理假设，至少还需要一两年的真实世界的使用经验，我们才能真正理解这个新模型。我们对本章中所写内容的正确性和相关建议的可靠程度有着坚实的自信，但同时我们也确信我们的理解将会继续发展和深入。

# 错误处理

13

作为程序员，我们总要是处理一些可能发生错误的事情，例如：网络连接可能会断开，访问的文件可能不存在等等。是否能够很好地处理失败情况是区分软件质量优劣的一个无形的指标。但通常在开发过程中，我们却倾向于把错误处理作为一个低优先级的任务，总是把它放在：“稍后再来实现”这样的地位（结果就是错误处理相关的工作通常会由于项目工期的问题最终被砍掉）。

我们知道：编写错误处理相关的代码会很杂乱，而实现正确逻辑的代码则有趣得多。因此，编程语言自身可以为程序员提供一个良好的模型来支持这项任务，就显得更为重要了。以下是我们认为的，带有 `throw`、`try` 和 `catch` 的 Swift 内建错误处理架构所应该具备的一些特征：

- **安全 (Safety)**: Swift 让程序员无法出于意外的原因而忽略错误处理。
- **简洁 (Conciseness)**: 不应该让逻辑正确的代码淹没在抛出和捕获错误的代码里。
- **通用 (Universality)**: 错误抛出和处理的机制可以用于包括异步代码在内的所有场景。  
新的 `async/await` 异步函数模式完全支持基于 `throw/try` 的错误处理方式。（在 `async/await` 之前，为了支持并发经常使用的回调方式，是完全无法和语言本身的错误处理机制相结合的。）
- **可传递 (Propagation)**: 错误不一定要在发生的原地进行处理，因为通常从一个错误中恢复的逻辑，会远离错误发生的地方。Swift 的错误处理架构可以很容易让错误沿着调用栈向上传递到合适的位置。中间函数（指的是那些自身不抛出错误也不处理错误的函数，但它们的实现里，调用了会抛出错误的其它函数）在不引入复杂语法变更的情况下也可以进行错误的传递。
- **文档 (Documentation)**: 编译器强制要求可抛出函数和它们的调用侧都需要进行标注，这让程序员可以很容易看到哪里可能会发生错误。不过，类型系统并不会把一个函数所能抛出的错误类型暴露出来。

在本章中，我们会再回顾这些要点。

## 错误分类

术语“错误 (Error)”和“失败 (Failure)”可以表达各种类型的问题。让我们先根据通常在代码中处理它们的方式，为“可能发生错误的事件”进行一些归类：

**预期中的错误：**这些指的是那些在常规操作中，可以被程序员预见的失败情况。例如：网络连接问题（网络连接永远都不可能是 100% 可靠的），或用户输入内容不合法问题等。根据造成失败原因的复杂度，我们可以进一步把预期中的错误分成下面几类。

- **可以忽略细节的错误 (Trivial errors)：**有些操作只有一个可以预期的失败情况。例如，查询字典的时候，结果只能是键值存在（操作成功）或不存在（操作失败）。在 Swift 里，对于“不存在”或“不合法输入”这种简单且明确的日常错误条件，我们倾向于让函数返回可选值。因为返回一个包含详细信息的错误，并不会比可选值提供更多有效的信息。  
在错误原因对函数调用者非常明确的时候，可选值在简洁性（这要得益于一部分可选值的语法糖），安全性（在使用可选值之前，必须对其解包），文档（可选值是包含在函数签名里的），可传递性（可选值支持串联）以及通用性（可选值的应用在 Swift 中无处不在）方面均表现优异。
- **需要提供详细信息的错误 (Rich errors)：**对于网络和文件系统操作，它们应该提供关于失败情况的更多实质性问题描述，而不能只是一句简单的“有些东西工作不正常”而已。在这些场景里，造成失败的因素很多，程序员会根据不同的因素采用不同的处理方法（例如：连接超时可能会重新发起请求，URL 不存在则会直接给用户错误提示）。这类错误是这一章接下来，我们主要关注的内容。

尽管大部分标准库 API 都返回可以忽略细节的错误（通常就是返回可选值），但 Codable 系统使用了这种需要提供详细信息的错误。编码和解码包含了很多触发错误的条件，因此，精确的错误信息对于指出发生问题的环节非常有价值。并且，编码和解码方法的声明中也都包含了 throws 关键字，提示它们的调用者要为错误处理做好准备。

- **非预期错误：**指的是那些在程序员预料之外的条件下导致的错误，这类错误通常会导致程序难以继续执行。通常这意味着程序员假定的一些（“绝对不会发生的”）条件被破坏了。例如，标准库就假设访问数组的下标不会越界，一个范围的上界不可能小于下界，整数不会溢出，以及除数不会为 0 等等。

Swift 中，处理这类错误的方式通常就是让程序崩溃，因为让程序在一个不确定的状态下执行并不安全。更多的是，这些情况都被认为是 **程序员自身的错误**，它们应该在测试中被检测出来并修复，而不是采用什么“恰当”的方式（例如：给用户显示一个错误提示）去处理。

在代码中，我们使用各种类型的断言（例如：assert, precondition, 或者 fatalError）来确认期望的结果，并且在不满足条件的时候，让程序中断。在可选值章节中，我们见到过这些函数。断言是一个定位代码中 bug 的很棒的工具。正确使用它，一旦你的程序进入到了非预期的状态，它就能帮你定位到最早触发这个问题的位置。另外，它们还是一个很棒的文档工具：每个 assert 或 precondition 调用都是方法的作者对程序状态作出的一个（隐性的）假设，这些假设通过断言可以呈现给代码的读者。

断言不应该用来提示预期中的错误，因为程序无法从断言中恢复执行，这也就意味着你失去了妥善处理这些错误的机会。同理，你也不应该使用可选值或可抛出异常的函数来应对属于程序员自身的错误。对于这类错误，最好就是将它当场抓获，而不是让它还有机会流窜到程序其它的位置。

## Result 类型

在继续深入 Swift 内建的错误处理之前，让我们先来讨论下 Result 类型。Result 是从 Swift 5 开始被加入到标准库的，但它各种形式的变体却早在 Swift 首次发布之后就在社区中流行开了。理解 Result 表达错误的方式，可以帮助我们去掉语法糖包装之后，看清 Swift 的错误处理方法。

在枚举这一章中，我们提到过，Result 是一个和 Optional 结构类似的枚举，它也包含两个成员，分别是：success 和 failure，它们的功能和 Optional 中的 some 和 none 是相同的。不同的是，Result.failure 也带有关联值，因此，Result 可以表达那些需要提供详细信息的错误：

```
enum Result<Success, Failure: Error> {
    case success(Success)
    case failure(Failure)
}
```

要注意的是，Result 给表达失败情况的泛型参数添加了 Error 约束，表示这个 case 只用于表达错误。

Optional 和 Result 的区别可以提示你究竟是否可以忽略错误的细节，还是必须要为错误提供额外的信息。这不是个巧合，当需要表达具体错误信息的时候，我们就可以用 Optional 表达简单错误的方式，来使用 Result。

假设我们正在写一个从磁盘读取文件的函数。一开始时，我们使用可选值来定义接口。因为读取一个文件可能会失败，在这种情况下，我们想要返回 nil：

```
func contentsOrNil(ofFile filename: String) -> String?
```

上面这个函数签名非常简单，但它没有告诉我们读取文件失败的具体原因。是因为文件不存在吗？还是说我们没有读取它的正确权限？显然，告诉调用者失败的原因是有必要的。因此，让我们定义一个 enum 来表明可能出现的错误情况：

```
enum FileError: Error {
    case fileDoesNotExist
    case noPermission
}
```

这样，我们就可以让函数返回一个 Result，它要么表示一个字符串（操作成功），要么表示一个 FileError（操作失败）：

```
func contents(ofFile filename: String) -> Result<String, FileError>
```

现在，函数的调用者就可以对调用结果情况进行判断，并且基于错误的类型作出不同的响应了。在下面的代码中，我们尝试读取文件，并在成功时打印文档内容。而失败的时候，根据可能发生的错误，打印一条为其特别定制的错误消息：

```
let result = contents(ofFile: "input.txt")
switch result {
    case let .success(contents):
        print(contents)
    case let .failure(error):
        switch error {
            case .fileDoesNotExist:
                print("File not found")
            case .noPermission:
                print("No permission")
        }
}
```

```
}
```

注意，上面这两个 switch 语句都不需要包含 default 分支——编译器可以从代码中确认我们已经包含了 switch 表达式中所有可能的值。这是由于 `FileError` 也是一个枚举。如果用 `Result<String, Error>` 作为函数的返回值，我们就得为判断错误的 switch 添加处理 default 的代码了。

## 抛出和捕获

Swift 内建的错误处理方式在很多方面都借鉴了上一节提到的 `Result` 的用法，只不过使用了不同的语法而已。Swift 没有使用返回 `Result` 的方式来表示失败，而是将方法标记为 `throws`。对于每个可以抛出错误的函数调用，编译器都会验证调用者有没有捕获错误，或者把这个错误向上传递给它调用者。把之前实现的 `contents(ofFile:)` 用 `throws` 语法表达出来是这样的：

```
func contents(ofFile filename: String) throws -> String
```

现在，所有对 `contents(ofFile:)` 的调用都必须用关键字 `try` 标记，否则代码将无法编译。而这个 `try` 关键字无论对编译器还是代码的读者来说，都是一个信号，表明这个函数可能会抛出错误。

调用一个可抛出错误的函数还会迫使我们决定如何处理错误。我们可以选择使用 `do/catch` 直接处理，或者把当前函数标记为 `throws` 将错误传递给调用栈上层的调用者。如果使用 `catch` 的话，可能会存在多条 `catch` 语句，我们可以用模式匹配来捕获某个特定的错误类型。在下面的例子中，我们显式地捕获了 `fileDoesNotExist` 的情况，并在最后的 `catch-all` 语句中处理其他所有错误。另外，在 `catch-all` 里，编译器还会自动生成一个 `error` 变量（这一点和属性的 `willSet` 中的 `newValue` 很像）：

```
do {
    let result = try contents(ofFile: "input.txt")
    print(result)
} catch FileError.fileDoesNotExist {
    print("File not found")
} catch {
    print(error)
```

```
// 处理其它错误。  
}
```

你也许会觉得 Swift 中的错误处理的语法看起来很眼熟。很多其他语言都在处理异常时使用相同的 try, catch 和 throw 关键字。除开这些类似点以外, Swift 的异常机制并不会像很多语言那样带来额外的运行时开销。编译器会认为 throw 是一个普通的返回, 这样一来, 普通的代码路径和异常的代码路径速度都会很快。

如果要在错误中给出更多信息, 我们可以使用带有关联值的枚举。例如, 一个文件解析器的程序库可以像下面这样建模可能的错误条件:

```
enum ParseError: Error {  
    case wrongEncoding  
    case warning(line: Int, message: String)  
}
```

注意, 我们也可以把一个结构体或者类作为错误类型来使用; 任何遵守 Error 协议的类型都可以被函数作为错误抛出。而且由于 Error 协议中其实并没有任何要求, 所以任何类型都可以声明遵守它, 而并不需要添加任何额外的实现。

为了快速测试某些代码, 或者编写一些简单原型的时候, 我们发现, 让 String 实现 Error 有时会很有帮助。只需要一行代码就可以搞定了: extension String: Error {}。这样, 就可以直接把表达错误消息的字符串作为可抛出的错误值使用, 例如: throw "File not found"。在生产环境的代码里, 我们并不推荐如此, 因为让一个不属于你的类型实现某个协议并不是值得推荐的做法(关于这方面更详细的讨论, 参考[协议](#)这一章的内容)。但在一个 REPL 会话或者类似的环境里, 这就是个很讨人喜欢的小技巧了。

有了 ParseError 之后, 我们的解析函数看起来是这样的:

```
func parse(text: String) throws -> [String]
```

现在，如果要解析一个字符串，同样，我们可以用模式匹配的方式来区分不同的错误。对于 warning 的情况，我们可以把行号和警告消息绑定到变量上，就像在 switch 的 case 语句中绑定变量一样：

```
do {
    let result = try parse(text: "{\"message\": \"We come in peace\" }")
    print(result)
} catch ParseError.wrongEncoding {
    print("Wrong encoding")
} catch let ParseError.warning(line, message) {
    print("Warning at line \(line): \(message)")
} catch {
    preconditionFailure("Unexpected error: \(error)")
}
```

其实，对于 Result 和 Swift 原生错误处理机制来说，无论是把成功和失败的情况分成不同的区域处理，还是在错误处理中，通过模式匹配区分执行路径以及绑定变量，这两种方法用起来的套路是非常相似的。当然，这并不是偶然，Swift 的错误处理方法本质上只是一层更漂亮的语法封装，本质上它要做的事情，还是创建并解包 Result。

## 具体类型错误和无类型错误

上节代码中的 do/catch 有些地方乍一看并不是非常合适。即使我们非常确定所有可能发生的错误都是 ParseError 类型的，并且，我们也逐一处理了它的每一个 case，编译器还是需要我们在最后写一个捕获所有异常的 catch 以确认所有可能的错误都被处理了。

这是因为 Swift 原生的错误处理机制使用了无类型错误 (untyped errors)。我们只能用 throws 声明函数会抛出错误，但无法指定它究竟会抛出哪些具体的错误。因此，为了从语言层面确保所有错误都可以被处理，编译器才总是要求我们编写一个 catchall 语句。在错误处理系统中使用无类型错误是 Swift 核心团队刻意为之的。原因是在大部分情况下，巨细无遗的错误处理是不现实也没必要的。通常，你可能只关心一两个特定的错误情况，然后给其它错误在 catchall 语句中提供一个通用的处理方法就好了。

而之前介绍过的 Result，则属于具体类型错误 (typed errors)。Result 带有两个泛型参数，Success 和 Failure，而后者指定了错误的具体类型。正是这种结构，才让我们最早实现 contents(ofFile:) 的时候，可以通过遍历 Result<String, FileNotFoundError> 中的 FileNotFoundError，处理每一种错误。再来看个例子，下面是 parse(text:) 方法的一个变体，我们把 throws 替换成了 Result<[String], ParseError>。于是，parse 可能发生的错误就被限定成了 ParseError，我们也就只能只处理它的每一种情况了 (无须像 do/catch 一样提供一个“多余”的 catchall 语句)：

```
func parse(text: String) -> Result<[String], ParseError>
```

通过上面这两个例子不难发现，使用 Swift 内建错误处理机制中的无类型错误，和使用 Result 这种带有类型的错误在行为模式上是不一样的。Swift 核心团队为什么决定接受这种差异呢？毕竟，核心团队还提供了一个持有无类型错误的 Result 变体，也就是说 case failure 的关联值是任何一个实现了 Error 的类型。正因为如此，Result 实际上是一个同时支持两种错误处理范式的混合体。如果你不希望指定具体的错误类型，就用 Result<..., Error> 作为返回值就行了。

所以，Result 为我们在使用无类型错误和具体错误之间提供了选择。只是用 Result 表示无类型错误的时候，要多写一些代码，因为我们要在泛型参数列表中，写上 Error。如果你觉得这样很烦，为这种 Result 定义一个别名就好了：

```
typealias UResult<Success> = Result<Success, Error>
```

顺带说一下，之所以可以定义 Result<Success, Error>，实际上是编译器特别为 Error 协议开了后门。刚才看到过，在 Result 中 Failure 是一个实现了 Error 的类型：

```
enum Result<Success, Failure: Error>
```

在 Swift 中的协议一般来说并不会遵守它们自己，因此 Result<..., Error> 这种写法中，Error 并不是一个满足类型约束要求的表达方式。为了允许用这样的形式表达无类型错误，Swift 团队为编译器加入了对 Error 的特别处理，让它是一个可以“自我实现”的协议，而其它协议均不具有这样的性质。我们在协议章节中详细讨论了相关内容。

既然现在我们有了通过 Result 表示具体类型错误的方法，很有可能在未来，这种用法也会整合到 Swift 原生错误处理机制中。在那之前，为了避免编译器总是提示我们要通过 catchall 处理所有错误，使用 Result 让它包含一个具体的错误类型都是个很好的选择。如果我们到处得到的

都是具体类型错误，可以指定抛出的具体错误类型就必然应该成为函数的一个可选功能，但也不必要求每个抛出错误的函数都必须如此。因为具体类型错误也有它自己严重的问题：

- 具体错误类型使得组合会抛出异常的函数，以及聚合这些函数抛出的错误都非常困难。  
如果一个函数调用了多个可能抛出错误的函数，要么它就会向调用栈上层传递多种错误，要么它就得为调用栈下层定义一个包含这些错误的全新错误类型。这种做法很快就会超出我们的控制。在错误链这一节，我们会继续讨论这个话题。
- 严格的错误类型会限制程序库的可扩展性。例如，每次为函数添加新的错误条件，对于那些需要捕获完整错误列表的调用代码来说，都是一次破坏性的更新。为了在不同版本的程序库之间维持二进制兼容性，每一处 do/catch 代码都要加上可以捕获所有异常的默认处理语句，这和处理非固定枚举时采取的方式是类似的。光是这一点就足以解释为什么像 Cocoa 这样的框架不太可能包含具体类型错误了。
- 和必须完整遍历枚举的所有成员不同，要求处理所有的错误情况通常都是没必要也不现实的。想象当你发起一个网络请求的时候，有多少种原因可能导致请求失败？对程序员来说，给每一种原因都提供一个合理的解决方案几乎是不可能的。大部分程序只会有针对性的处理几种常见的情况，并为其它原因提供一个通用的解决方案，例如记录一条日志，或给用户显示一个错误提示。

由于错误是无类型的，在文档中记录函数有可能抛出的错误就显得尤为重要了。Xcode 支持在代码注释中使用 Throws 关键字 标记函数抛出的错误。下面就是个例子：

```
/// Opens a text file and returns its contents.  
///  
/// - Parameter filename: The name of the file to read.  
/// - Returns: The file contents, interpreted as UTF-8.  
/// - Throws: `FileError` if the file does not exist or  
///           the process doesn't have read permissions.  
func contents(ofFile filename: String) throws -> String
```

这样，当你按住 Option 点击函数名的时候，在弹出的快速帮助面板中，就会出现一块专门用于显示抛出错误的区域了。

# 不可忽略的错误

在这一章的介绍中，我们把安全性作为了判断一个优质错误处理系统的因素。使用内建错误处理的一个很大的好处，就是当你调用一个可能会抛出错误的方法时，编译器不会让你忽略掉这些错误。但使用 Result，情况就不会总是如此了。

例如，考虑 Foundation 中的 Data.write(to:options:) 方法 (向文件中写入若干字节) 或 FileManager.removeItem(at:) 方法 (删除指定文件)：

```
extension Data {  
    func write(to url: URL, options: Data.WritingOptions = []) throws  
}  
  
extension FileManager {  
    func removeItem(at URL: URL) throws  
}
```

如果这些方法使用基于 Result 的错误处理方式，它们的声明看上去可能是这样的：

```
extension Data {  
    func write(to url: URL, options: Data.WritingOptions = [])  
        -> Result<(), Error>  
}  
  
extension FileManager {  
    func removeItem(at URL: URL) -> Result<(), Error>  
}
```

这些方法的特别之处就是我们是为了它们的副作用而调用它们的，而不是为了返回值。实际上，除了表示操作是否成功之外，这两个方法都没有一个真正有意义的返回值。所以，上面这两个 Result 的版本，无论是否是故意的，对于程序员来说，都太容易忽略掉任何失败的情况了，他们可能会直接写出下面的代码：

```
_ = FileManager.default.removeItem(at: url)
```

而调用 `throws` 版本的时候，编译器会强制我们在调用前面使用 `try` 前缀。编译器还会要求我们要么把调用嵌套在一个 `do/catch` 代码块里，要么把错误传递到调用栈的上层。无论是对写这段代码的程序员，还是这段代码的读者，这都是一个明确清晰的提示：当前调用的函数是有可能执行失败的，编译器会强制要求我们处理相关的错误。

对函数返回值来说，尽管 `Result<(), Error>` 不是一个好的选择，但对于那些不用什么具体值表示操作成功的异步代码，把它用在回调函数中返回错误则是十分常见的用法（因为 `throws` 在这种场景里是不可用的，我们在错误和回调这一节还会提到这个话题）。空的元组，或者说 `Void`，只有一个可能的值：`()`（这很容易让人困惑，因为一个类型和这个类型的值使用了同样的拼写方式）。因此，让 `Result.Success` 是 `()`，就表示“除了操作成功之外，没有任何其他信息了”。

## 错误转换

### 在 `throws` 和 `Optionals` 之间转换

错误和可选值，都是函数要对外报告有些功能不正确时的常用方式。在这章一开始的介绍中，我们提供了一些为自定义的函数选择报错方式的建议。但最终，你可能还是会同时和这两种方式打交道。当把一个函数的结果传递给其它 API 时，在可抛出错误的函数和返回可选值的函数之间，难免我们还要在这两种表达错误的形式之间来回转换。

`try?` 关键字允许我们忽略函数抛出的错误，并把函数的返回值变成包含原始返回值的 `Optional`。这个 `Optional` 可以告诉我们函数是否执行成功了：

```
if let result = try? parse(text: input) {  
    print(result)  
}
```

使用 `try?` 意味着我们将比之前获得更少的错误信息，我们唯一知道的，就是函数究竟是执行成功了，还是发生了某些错误，至于和错误相关的信息，则已经被丢掉无法找回了。类似的，

为了把一个返回 Optional 的函数变成一个抛出错误的函数，我们得为 nil 提供相应的错误值。

下面是个 Optional 的扩展，它会对自己解包，并且当自身为 nil 时，抛出一个错误：

```
extension Optional {  
    /// 如果是非 `nil` 值，就对 `self` 解包。  
    /// 如果 `self` 是 `nil`，就抛出错误。  
    func or(error: Error) throws -> Wrapped {  
        switch self {  
            case let x?: return x  
            case nil: throw error  
        }  
    }  
}
```

这个扩展用起来，是这样的：

```
do {  
    let int = try Int("42").or(error: ReadIntError.couldNotRead)  
} catch {  
    print(error)  
}
```

当多个 try 语句结合起来，或者当你在处理一个已经标记为 throws 的函数时，这个扩展就会非常有用。另外，在单元测试中，这也是个不错的模式，甚至 XCTest 框架都为它提供了一个便捷 API。XCTest 可以在测试方法抛出错误时，自动将这个测试标记为失败（当然你必须为这个方法标记上 throws 以确保能编译）。如果你的测试用例需要依赖一个 Optional 不为 nil 才可以继续，你可以调用 let nonOptional = try XCTUnwrap(someOptional) 来尝试解包这个可选值，当 Optional 为 nil 的时候，测试将会失败。而这些逻辑，只需要一行代码就能搞定了。

try? 关键字的存在可能会引起一些争议，毕竟它和 Swift 不允许忽略错误的哲学相违背。但说到底，你还是要明确使用 try? 关键字，这也可以看作是编译器强制你对错误进行的某种响应，而代码的读者，也可以通过 try? 明确你的意图。因此，当你对错误信息完全不感兴趣的时候，try? 是一种合理的选择。

try 还有第三种形式: try!。只有你确认函数绝对不可能发生错误的时候, 才应该使用这种形式。和强制解包一个 nil 的 Optional 值类似, 如果在运行时你的假设发生了错误, 那么使用 try! 就会造成程序崩溃。

## 在 throws 和 Result 之间转换

我们已经看到过了, Result 和 throws 关键字, 其实只是 Swift 错误处理机制的两种不同的呈现方式。把具体类型错误和无类型错误的区别先放一边, 你可以把 Result 看成是对可抛出错误的函数的返回值进行的改良。正是因为这种双重身份, 标准库中提供了在这两种表现形式之间进行转换的方法就一点儿也不奇怪了。

为了调用一个可抛出错误的函数, 并把它的返回值包装成一个 Result, 可以使用 init(catching:) 初始化方法, 它接受一个可抛出错误的函数作为参数, 并把这个函数的返回值包装成Result 对象。它的实现是这样的:

```
extension Result where Failure == Swift.Error {
    /// 通过评估一个可抛出错误的函数的返回值创建一个新的 `Result` 对象,
    /// 把成功的返回结果包装在 `case success` 里, 而失败时抛出的错误
    /// 则包装在 `case failure` 里。
    init(catching body: () throws -> Success) {
        do {
            self = .success(try body())
        } catch {
            self = .failure(error)
        }
    }
}
```

这个初始化方法用起来是这样的:

```
let encoder = JSONEncoder()
let encodingResult = Result { try encoder.encode([1, 2]) } // success(5 bytes)
type(of: encodingResult) // Result<Data, Error>
```

如果你想延迟处理错误，或者把函数的返回结果发送给其它函数，这个方法就会非常有用。

和 init(catching:) 相反的方法叫做 Result.get()。它会评估 Result 的结果，并把 failure 中的值作为错误抛出。它的实现[在这里](#)：

```
extension Result {  
    public func get() throws -> Success {  
        switch self {  
            case let .success(success):  
                return success  
            case let .failure(failure):  
                throw failure  
        }  
    }  
}
```

## 错误链

接连调用多个可能抛出错误的函数是很普遍的情况。例如，一个操作可能被分成多个子任务，每一个子任务的输出都是下一个子任务的输入。如果每个子任务都可能执行失败，一旦其中一个抛出错误，整个操作就应该立即退出。

### throws 链

并不是所有的错误处理系统都可以很好地处理上面这种情况，但这在 Swift 内建错误处理机制之下就很简单了，我们不需要使用嵌套的 if 语句或者类似的结构来保证代码运行，只要简单地将这些函数调用放到一个 do/catch 代码块中（或者封装到一个被标记为 throws 的函数中）就好了。当遇到第一个错误时，调用链将结束，代码将被切换到 catch 块中，或者传递到上层调用者去。

下面这个例子，是个拥有三个子任务的操作：

```
func complexOperation(filename: String) throws -> [String] {  
    let text = try contents(ofFile: filename)
```

```
let segments = try parse(text: text)
return try process(segments: segments)
}
```

## Result 链

让我们把基于 try 的例子和与之等价的使用 Result 的代码进行一次对比。将多个返回 Result 的函数手动链接起来需要很多努力。我们需要调用第一个函数，解包它的输出，如果遇到的是 .success，则将值传递给下一个函数重新开始这个过程。一旦函数返回了 .failure，则需要将链打断，放弃接下来的所有调用，并将这个失败返回给调用者：

```
func complexOperation1(filename: String) -> Result<[String], Error> {
    let result1 = contents(ofFile: filename)
    switch result1 {
        case .success(let text):
            let result2 = parse(text: text)
            switch result2 {
                case .success(let segments):
                    return process(segments: segments)
                    .mapError { $0 as Error }
                case .failure(let error):
                    return .failure(error as Error)
            }
        case .failure(let error):
            return .failure(error as Error)
    }
}
```

很快，这种用法就会让代码变得一团糟，因为每串联一个返回 Result 的函数，就需要一层额外的 switch 语句嵌套，并且，还要重复一遍相同的错误处理语句。

在重构这段代码之前，我们先来看看在所有 failure 语句中是如何处理错误的。以下是这些方法的签名：

```
func contents(ofFile filename: String) -> Result<String, FileError>
func parse(text: String) -> Result<[String], ParseError>
func process(segments: [String]) -> Result<[String], ProcessError>
```

每个函数都有个不同的错误类型：FileError，ParseError 和 ProcessError。因此，沿着这些子任务的调用链，我们不仅要转换每一步成功之后的结果（从 String 到 [String] 再到 [String]），还必须要把每一步可能发生的错误转换成一个聚合类型，在上面的代码中，这个聚合类型就是 Error，当然也可以是其它具体的类型。而这种错误类型的转换，一共发生了三次：

- 前两个 return .failure(error as Error) 把 error 从具体类型转换成了一个遵守 Error 的类型。我们也可以忽略 as Error 的部分，编译器会帮我们进行隐式类型转换。但明确写出来可以表明这里真正完成的工作。
- 在调用链的最后一步，我们不能简单 return process(segments: segments)，因为 process 的返回值类型和 Result<[String], Error> 并不兼容。我们必须用 mapError（这是 Result 提供的方法）对错误类型再进行一次转换。

先把由于严格的错误类型导致的复杂性放在一边，无论如何我们都应该重构这一坨嵌套的 switch 语句。幸运的是，Result 已经提供了解决办法。Result.flatMap 方法封装了这种根据 Result 结果决定是要继续向下一个环节传递成功值，还是由于失败必须退出调用链的模式。它的结构和我们在可选值这一章中提到的 flatMap 是一样的。

用 flatMap 替换掉嵌套的 switch 语句之后，代码立即就清爽了很多。实际上，最终的结果是非常优雅的，尽管和 throws 的实现方案比还差了那么一点儿：

```
func complexOperation2(filename: String) -> Result<[String], Error> {
    return contents(ofFile: filename).mapError { $0 as Error }
        .flatMap { text in
            parse(text: text).mapError { $0 as Error }
        }
        .flatMap { segments in
            process(segments: segments).mapError { $0 as Error }
        }
}
```

要注意，我们还是得处理不兼容的错误类型。Result.flatMap 只会转换执行成功的结果，并保持 failure 的情况不变。因此串联多个 map 或者 flatMap 就要求 Result 中的 Failure 类型是相同的。在我们的例子中，这是通过不断调用 mapError 完成的，它们的任务就是把具体的错误类型泛化成“一个实现了 Error 的类型”。

这一节，我们讨论的例子很好地展示了严格错误类型系统可能带来的问题，通常它们带来麻烦要比收益更多。如果可以在 complexOperation2 的实现中去掉 mapError，显然代码的可读性会更好。退一步说，其实在整个调用链最后，complexOperation2 的返回值中，也已经不带有具体的错误类型了，所以，整个调用栈的上层方法也完全享受不到具体的错误类型带来的好处。

## 错误和回调

Swift 的内建错误处理和使用 `async/await` 模型的异步 API 有着紧密的结合：到目前为止我们所讨论过的内容，都可以等效地运用在 `async` 函数和方法上。不过内建的错误处理机制无法和通过基于回调的异步 API 搭配在一起工作。让我们看个异步大数计算的例子，它通过回调函数在计算完成后通知结果：

```
func compute(callback: (Int) -> ())
```

我们需要给 `compute` 提供一个回调函数，这个回调函数把计算结果作为它唯一的参数：

```
compute { number in
    print(number)
}
```

在这种模式下，应该如何集成错误呢？如果可选值足以表达要传递的错误，我们可以让回调函数接受 `Int?` 作为参数就好了。这样，只要回调函数收到 `nil`，就表示计算失败了：

```
func computeOptional(callback: (Int?) -> ())
```

现在，在我们的回调函数里，就必须通过某种方式对参数进行解包了，例如，使用 `??` 操作符：

```
computeOptional { numberOrNil in
```

```
    print(numberOrNil ?? -1)
}
```

如果我们想给回调函数传递给多错误信息该怎么办呢？下面这个函数签名看上去是个很自然的做法：

```
func computeThrows(callback: (Int) throws -> ())
```

但是它和我们想象的并不一样，这个签名有着完全不同的含义。它并不表明计算大数的方法会执行失败，而是表示回调函数自身可能发生错误。我们把上面的回调函数用返回 `Result` 的形式写出来，问题就清楚多了：

```
func computeResult(callback: (Int) -> Result<(), Error>)
```

当然，上面两种签名没有一个是对的。我们需要的是把计算得到的 `Int` 包装在 `Result` 里，而不是用 `Result` 包装回调函数的返回值。

最终，我们想出了下面这样的做法：

```
func computeResult(callback: (Result<Int, Error>) -> ())
```

Swift 内建错误处理机制对异步 API 的不兼容，体现了使用 `throws` 和使用 `Optional` 或 `Result` 处理错误时的一个关键区别。只有后者才可以自由地传递错误信息，而 `throws` 反而不那么灵活。对此，[Joshua Emmons](#) 非常形象地解释了这个区别：

看，`throw` 和 `return` 很像，它只能沿着一个方向工作，也就是向上传递消息。我们可以把错误抛给函数的调用者，但不能把错误向下作为参数抛给接下来会调用的其它函数。

这种可以把错误抛给接下来会执行的函数的能力，正是我们在基于回调的异步代码环境中进行错误处理所需要的。但不幸的是，现在还没有一种非常清晰的方式表明应该如何把 `throws` 用在异步的环境里。我们能做的，只有把 `Int` 包装在一个可能抛出错误的函数里，但这只能让 `compute` 的签名更加复杂：

```
func compute(callback: () throws -> Int) -> ()
```

并且，compute 用起来也会更加麻烦。为了得到计算的整数，我们得在回调函数里调用这个会抛出错误的函数。这也就意味着，我们要在回调函数里，进行错误处理：

```
compute { (resultFunc: () throws -> Int) in
    do {
        let result = try resultFunc()
        print(result)
    } catch {
        print("An error occurred: \(error)")
    }
}
```

虽然这样可行，但绝不是 Swift 提倡的编程方式。对于回调中的错误处理，使用 Result 是更好的方式。从 Swift 5.5 开始，compute 函数更应该被写成 async 的形式，这会极大简化错误处理：

```
func compute() async throws -> Int

do {
    print(try await compute())
} catch {
    print("An error occurred: \(error)")
}
```

## 使用 defer 进行清理

很多编程语言都有 try/finally 这样的结构，当函数返回的时候，无论是否发生错误，finally 指定的代码块总是会被执行。Swift 中的 defer 关键字功能和它类似，但具体做法却稍有不同。和 finally 类似的是，当离开当前作用域的时候，defer 指定的代码块也总是会被执行，无论是因为执行成功返回，还是因为发生了某些错误，亦或是其它原因。这使得 defer 代码块成为了执

行清理工作的首选场所。和 `finally` 不同的是，`defer` 不需要前置的 `try` 或 `do` 代码块，你可以把它部署到代码中的任何地方。

让我们回到本章开头的 `contents(ofFile:)` 函数，来看看基于 `defer` 的一种可能的实现：

```
func contents(ofFile filename: String) throws -> String {  
    let file = open(filename, O_RDONLY)  
    defer { close(file) }  
    return try load(file: file)  
}
```

无论 `contents` 执行成功或者抛出了错误，第二行的 `defer` 代码块都可以确保文件在函数返回的时候可以被关闭。

虽然 `defer` 经常会被和错误处理一同使用，但在其他上下文中，这个关键字也很有用。例如，当你希望把资源的初始化和清理代码放在一起的时候（例如打开和关闭文件）。把这类代码放在一起可以极大地提高代码的可读性，尤其是在较长的函数里。

如果相同的作用域中有多个 `defer` 代码块，它们将按照定义的顺序逆序执行。你可以把这些 `defer` 想象成一个栈。起初，你可能会觉得逆序执行 `defer` 很奇怪，不过，如果看看下面这个执行数据库查询的例子，你就会觉得这种做法非常合理了：

```
let database = try openDatabase(...)  
defer { closeDatabase(database) }  
let connection = try openConnection(database)  
defer { closeConnection(connection) }  
let result = try runQuery(connection, ...)
```

在执行查询之前，我们必须打开数据库并创建一个数据库连接。接下来，如果在执行 `runQuery` 的时候抛出错误了，资源的清理工作当然应该按照和代码正常执行相反的顺序完成。我们应该先关闭数据库连接，再关闭数据库自身。由于 `defer` 语句就是逆序执行的，因此这正是我们期望的结果。

一个 `defer` 代码块会在程序离开 `defer` 定义的作用域时被执行。甚至 `return` 语句的评估都会在同作用域的 `defer` 被执行之前完成。你可以利用这个特性在返回某个变量之后再修改某个变量

的值。在接下来的例子中，increment 函数在返回 counter 之后，使用 defer 代码块递增了捕获到的 counter：

```
var counter = 0

func increment() -> Int {
    defer { counter += 1 }
    return counter
}

increment() // 0
counter // 1
```

如果去看看标准库中的代码，就会发现这样的用法比比皆是。如果没有 defer，编写同样的逻辑就只能声明一个临时变量存储 counter 的值了。

当然，也有一些 defer 语句不会执行的情况，例如：当程序发生段错误的时候，或者触发了致命错误的时候（使用 fatalError 函数或者强制解包 nil），这时所有代码执行都会立即终止。

## Rethrows

由于函数可以抛出错误，这给那些接受函数作为参数的函数（例如 map 或 filter）带来了一个问题。在内建集合类型这一章，我们讨论了一个基于 Array 实现的 filter（实际的 filter 是定义在 Sequence 中的，它更复杂一些）：

```
func filter(_ isIncluded: (Element) -> Bool) -> [Element]
```

这个定义没问题，但它有个缺陷：编译器不会接受一个可抛出错误的函数作为谓词，因为 isIncluded 参数没有标记为 throws。

接下来，我们再看个例子。这次 isIncluded 参数所呈现的缺陷会升级成为一个较为棘手的问题。我们从编写一个检查文件某种可用性的函数开始（至于检查的究竟是什么在这个例子中并不重要）。checkFile 可以返回一个布尔值（true 表示可用，false 不可用），或者抛出一个检查文件过程中发生的错误：

```
func checkFile(filename: String) throws -> Bool
```

假设我们有一个文件名数组，要从中筛选出不可用的文件。很自然地，我们会选择使用 filter 方法，但是编译器不会让我们这么干，因为 checkFile 是一个可能抛出错误的函数：

```
let filenames: [String] = ...
// Error: Call can throw but is not marked with 'try'.
let validFiles = filenames.filter(checkFile)
```

作为一种解决方案，我们可以在 filter 的谓词函数中进行错误处理：

```
let validFiles = filenames.filter { filename in
    do {
        return try checkFile(filename: filename)
    } catch {
        return false
    }
}
```

但这样很不方便，甚至这都不是我们想要的效果——上面的代码用 false 遮掩了 checkFile 所有可能抛出的错误。但如果我想在发生错误的时候就中断所有的操作该怎么办呢？

一种解决方案就是让标准库在 filter 的签名中，用 throws 修饰自己的谓词函数：

```
func filter(_ isIncluded: (Element) throws -> Bool) throws -> [Element]
```

这样做可行，但它同样会带来不便。因为现在每个 filter 调用都必须用 try (或者 try!) 来修饰。进而不难想象，这会导致标准库中所有高阶函数都必须用 try 调用。显然，这就违背了 try 关键字的设计初衷，它本来是帮助代码读者快速识别可抛出错误函数的。

另外一种实现方案是实现两个版本的 filter，分别接受普通的和可抛出错误的谓词函数。除了要用 try 调用谓词函数之外，这两个版本的 filter 实现，是完全一样的。我们可以依赖编译器根据函数重载的规则自动选择正确的版本。这样做看上去更好一些，至少不同版本的调用很清晰，但是这还是太浪费了。

幸运的是，Swift 通过 `rethrows` 关键字提供了一个更好的方案。用 `rethrows` 标记一个函数就相当于告诉编译器：这个函数只有它的参数抛出错误的时候，它才会抛出错误。因此，`filter` 方法最终的签名是这样的：

```
func filter(_ isIncluded: (Element) throws -> Bool) rethrows -> [Element]
```

谓词函数仍旧被标记成了 `throws` 函数，表示调用者可能会传递一个可抛出错误的函数。在 `filter` 的实现里，必须使用 `try` 调用谓词函数。而 `rethrows` 则确保了 `filter` 会把谓词函数中的错误沿着调用栈向上传递，但 `filter` 自身不会抛出任何错误。因此，当传递的谓词函数不会抛出错误时，编译器就不会要求使用 `try` 调用 `filter` 了。

在标准库里，几乎所有序列和集合类型中，带有函数类型参数的方法都是用 `rethrows` 进行了标记。其中只有一个例外，就是延迟加载的集合方法，我们在集合类型协议这一章会详细讨论这个类型。这主要是由于延迟加载的集合会把变形函数存储起来之后再用。在这个上下文中想要支持抛出，就需要把所有可能变成延迟集合的 Collection API 调用标记为 `try`，这再一次违反了 `try` 作为实际抛出调用标记的目的。

## 将错误桥接到 Objective-C

在 Objective-C 里，并没有像 `throws` 和 `try` 这样的机制。(虽然 Objective-C 中确实有一套相同的关键字用来处理异常，但 Objective-C 中的异常应该只被用来表达程序员的错误。你很少会在一个普通的 app 里去用 Objective-C 异常)

Cocoa 的通用做法是在发生错误时返回 `NO` 或者 `nil`。另外，可能发生错误的方法还会接受一个 `NSError` 指针的引用作为额外的参数。它们使用这个指针给函数的调用者传递错误的详细信息。例如，Objective-C 版本的 `contentsOfFile:` 写出来可能是这样的：

```
- (NSString *)contentsOfFile:(NSString *)filename error:(NSError **)error;
```

Swift 会自动把遵循这个规则 (译注：这个规则指的是接受 `NSError **` 作为参数) 的方法转换为 `throws` 语法的版本。因为不再需要参数传递错误了，它会从声明中删除。于是，`contentsOfFile` 被导入到 Swift 就会变成这样：

```
func contentsOfFile(filename: String) throws -> String
```

(译者注：签名的转换过程除了会删除 NSError \*\* 类型的参数之外，对于那些返回 BOOL 来表示操作成功与否的函数，例如 `createDirectoryAtPath`，转换到 Swift 之后也会变成一个 throws 函数，也就不再需要使用 BOOL 表示调用结果了，因此，BOOL 会变成 Void。)

使用了这种结构的所有 Objective-C 方法都会自动完成同样的签名转换。其他的 NSError 参数，比如异步 API 的 completion 回调中，回传给调用者的错误，将被桥接到 Error 协议，所以一般来说你不再需要直接和 NSError 打交道了。

如果你把一个 Swift 错误传递给 Objective-C 的方法，类似地，它将被桥接为 NSError。因为所有的 NSError 对象都必须有一个 domain 字符串和一个整数的错误代码 code，运行时将在必要的时候提供默认值，它会使用类型名作为 domain，使用从 0 开始的枚举的序号作为错误代码。如果有需要，你也可以让你的错误类型遵守 CustomNSError 协议来提供自定义的值。

例如，我们可以像这样扩展 ParseError：

```
extension ParseError: CustomNSError {
    static let errorDomain = "io.objc.parseError"

    var errorCode: Int {
        switch self {
            case .wrongEncoding: return 100
            case .warning(_, _): return 200
        }
    }

    var userInfo: [String: Any] {
        return [:]
    }
}
```

类似地，你还可以实现下面这两个协议，来让你的错误拥有更有意义的描述，并且更好地遵循 Cocoa 的习惯：

- **LocalizedError** — 提供一个本地化的信息，来表示错误为什么发生 (`failureReason`)，从错误中恢复的提示 (`recoverySuggestion`) 以及额外的帮助文本 (`helpAnchor`)。
- **RecoverableError** — 描述一个用户可以恢复的错误，展示一个或多个 `recoveryOptions`，并在用户要求的时候执行恢复。这多用于使用了 AppKit 的 macOS 应用。

就算没有实现 `LocalizedError` 协议，所有实现了 `Error` 的类型也会有一个可以重写的 `localizedDescription` 属性。

实现 `Error` 的那些类型也可以定义自己的 `localizedDescription`。不过，因为这并不是 `Error` 协议所要求的，这个属性也不支持动态派发。除非你也遵守了 `LocalizedError`，否则在 Objective-C 的 API 或者 `Error` 的存在体容器 (existential container) 里，你将不能使用这个自定义的 `localizedDescription`。当编写 Cocoa 应用的时候，你应该总是让传递给 Cocoa API 的错误类型实现 `LocalizedError` 协议。关于动态派发和存在体的更多详细信息，可以参考[协议这一章](#)的内容。

## 回顾

Swift 给了我们不少选择来处理代码中的意外情况。当我们不可能继续时，可以使用 `fatalError` 进行断言。当我们对错误的类型不感兴趣，或者只有一种可能的错误的话，我们使用可选值。当我们多于一种错误，或者想要提供额外信息的话，我们可以使用 Swift 原生的错误处理模型或者 `Result` 类型。

当 Apple 在 Swift 2.0 中引入错误处理机制的时候，社区中很多人都持怀疑态度。在 Swift 严格的类型系统中，让 `throws` 使用无类型错误被认为是一个偏离语言设计初衷的举措。起初，我们也对此持怀疑态度，但事后来看，我们认为 Swift 团队作出了正确的选择，因为细粒度的错误处理通常是不必要的。现在，我们有了一个表达泛型错误类型的 `Result`，这是一个好的契机，在未来，强类型错误处理也可能会作为语言的特性被加入进来。

错误处理是 Swift 作为一门务实语言的好例子，开发团队选择首先针对最常用的情况进行优化。相比基于 `Result` 和 `flatMap` 这种“更纯函数式”的风格，让语法对习惯了 C 风格的程序员感到熟悉是个更为重要的目标，当然现在标准库也支持了函数式编程。错误处理模型的设计也遵循了 Swift 的一贯风格：它的目标是把安全、实用的理念包装到友好并紧凑的语法中 (另一个例子

是值类型的可修改模型)。对 `async/await` 风格的并发模型的引入，以及它们同样支持 `throws` 进行错误处理，是朝这个目标方向迈进的坚实一步。

# 编码和解码

14

将程序内部的数据结构序列化为一些可交换的数据格式，以及反过来将通用的数据格式反序列化为内部使用的数据结构，这在编程中是一项非常常见的任务。Swift 将这些操作称为**编码 (encoding)** 和**解码 (decoding)**。

Codable 系统 (以这个系统提供的协议命名，而这个协议实际上只是一个别名) 定义了一套编码和解码数据的标准方法，所有自定义类型都能选择使用这套方法。它的设计主要围绕三个核心目标：

- **普遍性** - 对结构体、枚举和类都适用。
- **类型安全** - 像 JSON 这样的可交换格式通常都是弱类型的，而你的代码应该使用强类型数据结构。
- **减少模板代码** - 当自定义类型加入这套系统时，应该尽可能减少开发者需要编写的“适配代码”，编译器应该可以自动生成它们。

一个类型通过声明自己遵守 Encodable 和/或 Decodable 协议，来表明可以被序列化和/或反序列化。这两个协议都只约束了一个方法，其中：Encodable 约束了 encode(to:)，它定义了一个类型如何对自身进行编码；而 Decodable 则约束了一个初始化方法，用来从序列化的数据中创建实例：

```
/// 一个类型可以将自身编码为某种外部表示形式。  
public protocol Encodable {  
    /// 将值编码到给定的 encoder 中。  
    public func encode(to encoder: Encoder) throws  
}  
  
/// 一个类型可以从某种外部表示形式中解码得到自身。  
public protocol Decodable {  
    /// 从给定的 decoder 中解码来创建新的实例。  
    public init(from decoder: Decoder) throws  
}
```

因为大多数实现了其中一个协议的类型，也会实现另一个，所以标准库中还提供了 Codable 类型别名，它是这两个协议组合后的简写：

```
public typealias Codable = Decodable & Encodable
```

标准库中的所有基本类型，包括 Bool，数值类型和 String，都是实现了 Codable 的类型。另外，如果数组，字典，Set 以及 Range 中包含的元素实现了 Codable，那么这些类型自身也是实现了 Codable 的类型。另外，包括 Data，Date，URL，CGPoint 和 CGRect 在内的许多 Apple 框架中的常用数据类型，也已经适配了 Codable。最后，如果结构体、类和枚举的所有属性或关联值都满足 Codable，那么 Swift 编译器就可以为它们自动生成 Codable 的实现。

依靠所有这些类型的内建 Codable 和编译器生成的 Codable 支持，也有不方便的一面，那就是缺乏对序列化数据格式的控制。如果你只是需要一种简易的方法来为你的数据进行序列化（和反序列化），而并不对序列化后的数据究竟表现为怎样格式有特殊要求的话， Codable 系统将恰如其分。如果你想要和像是你无法控制的 JSON API 这样的外部格式进行对接的话，如果外部格式和 Swift 的默认值只有细微差别的话， Codable 依然会是不错的选择。如果数据结构和 Codable 系统的默认值有太多不兼容的地方，在 Codable 架构之上构建你自己的序列化代码依然是可能的，不过想要达成这个目的，你必须要编写很多编码和解码的代码。

下面，我们会先看看 Codable 系统在直接使用进行序列化时的行为。然后，我们将探讨如何使用属性包装来选择性地自定义序列化的数据格式。最后，我们将深入研究编码和解码的过程。

## 一个最小的例子

让我们从一个最小的例子开始，它使用 Codable 系统将一个自定义类型的实例编码为 JSON。

### 自动遵循协议

只要让你的类型满足 Codable 协议，它就能变为可编解码的类型。如果类型中所有的存储属性都是可编解码的，那么 Swift 编译器会自动帮你生成实现 Encodable 和 Decodable 协议的代码。下面的 Coordinate 存储了一个 GPS 位置信息：

```
struct Coordinate: Codable {  
    var latitude: Double  
    var longitude: Double  
    // 不需要实现
```

```
}
```

因为 Coordinate 的两个存储属性都已经是可编解码的类型，所以只要声明 Coordinate 实现了 Codable，就完全可以满足编译器的需要了。现在，我们可以定义一个 Placemark 结构体，由于 Coordinate 已经实现了 Codable，它也就自动成为一个满足 Codable 的类型了：

```
struct Placemark: Codable {  
    var name: String  
    var coordinate: Coordinate  
}
```

如果枚举不包含关联值，或者它们的关联值也遵守 Codable，那么这个枚举也可以同样地通过代码生成满足 Codable。比如说，我们可以定义下面这样带有关联值的 Surrounding 枚举，把它添加到另一个版本的 Placemark 中，我们什么额外的事情都不需要做，就可以让它们都满足 Codable：

```
enum Surrounding: Codable {  
    case land  
    case inlandWater(name: String)  
    case ocean(name: String)  
}  
  
struct Placemark2: Codable {  
    var name: String  
    var coordinate: Coordinate  
    var surrounding: Surrounding  
}
```

编译器为一个类型自动合成 Codable 协议的代码是不可见的，不过在本章稍后，我们会一点点地对其进行剖析。现在，可以将这些生成的代码看作是标准库对该协议的默认实现，就像 Sequence.drop(while:) 那样。你可以免费获得默认的行为，也可以提供自己的实现。

让编译器生成代码和一个常规的默认实现的唯一一个实质性的区别在于，后者意味着这部分代码是标准库的一部分。但现在，合成 Codable 实现的逻辑还属于编译器的职责。要把这部分职

责迁移到标准库，需要 Swift 拥有比现如今更强大的类型反射能力。不过即便这些能力存在，运行时类型反射也有自己的额外开销（通常，反射机制会比编译器在内部直接处理慢一些）。

无论如何，尽可能将语言定义从编译器移动到标准库中，一直是 Swift 的一个明确目标。在未来，我们可能会得到一个足够强力的宏系统，将整个 Codable 移动到标准库中，不过那至少是好几年以后的事情了。在那之前，依靠编译器合成代码还是解决这个问题更为实用的方式。并且，这种设计在 Swift 其它语言特性的实现上也得到了应用，例如：为结构体和枚举自动合成实现 Equatable 和 Hashable 的代码，以及为枚举合成实现 Caselterable 的代码。

## Encoding

Swift 自带两个编码器，分别是 JSONEncoder 和 PropertyListEncoder（它们定义在 Foundation 中，而不是在标准库里）。另外，实现了 Codable 的类型和 Cocoa 中的 NSKeyedArchiver 也是兼容的。接下来，我们会集中研究 JSONEncoder，因为 JSON 是网络上最常见的数据交换格式。

我们可以像这样把一个 Placemark 数组编码为 JSON：

```
let places = [
    Placemark(name: "Berlin", coordinate:
        Coordinate(latitude: 52, longitude: 13)),
    Placemark(name: "Cape Town", coordinate:
        Coordinate(latitude: -34, longitude: 18))
]

do {
    let encoder = JSONEncoder()
    let jsonData = try encoder.encode(places) // 129 bytes
    let jsonString = String(decoding: jsonData, as: UTF8.self)

    /*
    [{"name": "Berlin", "coordinate": {"longitude": 13, "latitude": 52}},
     {"name": "Cape Town", "coordinate": {"longitude": 18, "latitude": -34}}]
    */
} catch {
```

```
    print(error.localizedDescription)
}
```

实际的编码步骤非常简单：创建并且配置编码器，然后将值传递给它进行编码。JSON 编码器通过 Data 实例的方式返回一个字节的集合，这里为了显示，我们将它转为了字符串。

除了通过一个属性来设定输出格式 (带有缩进的易读格式和/或按词典对键进行排序) 以外，`JSONEncoder` 还支持对日期的表达方式 (包括 ISO 8601 或者 Unix epoch 时间戳)，`Data` 值的形式 (比如进行 Base64 编码) 以及异常浮点数的处理方法 (例如，无穷或者 `NaN`) 进行自定义。我们甚至可以使用编码器的 `keyEncodingStrategy` 选项让 JSON 中的键采用蛇形命名方式 (`snake case`)，或者自定义生成键的函数。这些选项对所有值的编码是通用的，也就是说，你不能指定 `Date` 在不同的类型中，采用不同的编码配置。如果需要这种粒度上的控制，你只能对受影响的类型编写自定义的 `Codable` 实现，或者使用一个属性包装来为制定的属性进行编码自定义 (我们在下面会展示一个这么做的例子)。

值得注意的是，所有这里提到的配置项都是针对 `JSONEncoder` 说的。其他的编码器会有不同的选项 (或者没有选项)。而且 `encode(_:)` 方法也是随着编码器不同而不同的，它并没有被定义在任何协议里。其他的编码器可能会返回一个 `String` 或者甚至是被编码后文件的 URL，而不像 `JSONEncoder` 那样返回一个 `Data` 值。

实际上，`JSONEncoder` 甚至都没有实现 `Encoder` 协议，它只是一个实现了 `Encoder` 协议的私有类的封装，这个私有类会负责进行实际的编码工作。之所以这样做，是因为顶层编码器 (译注：这里指 `JSONEncoder`) 应该提供的 API (这个 API 通常只用于启动编码过程)，和在编码过程中传递给可编码类型的 `Encoder` 对象 (译注：这里指被封装的私有类) 是截然不同的。将这些任务清晰地分开，意味着在任意给定的情景下，使用编码器的一方只能访问到适当的 API。例如，一个 `Codable` 类型不能在编码过程中重新配置编码器，因为公开的配置 API 只暴露在顶层编码器的定义里。之后，Apple 在 Combine 框架中正式确定了顶层编码器和解码器的概念，它们包括 `TopLevelEncoder` 和 `TopLevelDecoder` 协议。有人建议把这些内容移动到标准库中，但是现在这件事还没有发生。

## Decoding

JSONEncoder 的解码器版本是 JSONDecoder。解码和编码遵循同样的模式：创建一个解码器，然后将 JSON 数据传递给它进行解码。JSONDecoder 接受一个 Data 实例，这个 Data 应该包含 UTF-8 编码的 JSON 文本。不过和编码器一样，其他类型的解码器也可能会有不同的接口：

```
do {
    let decoder = JSONDecoder()
    let decoded = try decoder.decode([Placemark].self, from: jsonData)
    // [Berlin (lat: 52.0, lon: 13.0), Cape Town (lat: -34.0, lon: 18.0)]
    type(of: decoded) // Array<Placemark>
    decoded == places // true
} catch {
    print(error.localizedDescription)
}
```

注意 decoder.decode(\_:from:) 接受两个参数。除了输入的数据，我们还需要指定解码的目标类型 (这里是 [Placemark].self)。这让代码在编译期间能够类型安全。而 JSON 中的弱类型数据到我们代码中的具体数据类型的转换，这个乏味冗长的过程则是在后台自动完成的。

将解码的目标类型明确地作为解码方法的参数，是一个有意的设计选择。这其实不是严格必须的，因为编译器其实可以在很多情况下推断出正确的类型。但 Swift 团队认为增加 API 的明确性和避免产生歧义，要比最大化精简代码更重要。

和编码过程比起来，解码过程中的错误处理是非常重要的。有太多的事情能导致解码失败 - 比如数据缺失 (JSON 中缺少某个必要的字段)、类型错误 (服务器不小心将数字编码为了字符串) 以及数据完全损坏等。你可以通过 [DecodingError](#) 类型的文档来查看可能会遇到的完整错误列表。

## 自定义编码格式

有时候，内建的编码器和解码器所使用的数据格式只需要一点点小调整就能满足像是和特定 JSON API 通讯的需求。我们在上面已经看到，JSONEncoder 为数据在 JSON 格式中的常见表现方式，提供了多种配置选项。如果这些选项没有提供你所需要的表现，你可以考虑用一个属性包装来对特定属性的序列化格式进行自定义。

比如，我们也许会想要把 Coordinate 中的 Double 值表示为字符串。为了做到这一点，我们会实现一个 CodedAsString 属性包装，并“手动”让它满足 Codable。也就是说，我们会自己实现 init(from:) 初始化方法和 encode(to:) 方法：

```
@propertyWrapper
struct CodedAsString: Codable {
    var wrappedValue: Double
    init(wrappedValue: Double) {
        self.wrappedValue = wrappedValue
    }

    init(from decoder: Decoder) throws {
        let container = try decoder.singleValueContainer()
        let str = try container.decode(String.self)
        guard let value = Double(str) else {
            let error = EncodingError.Context(
                codingPath: container.codingPath,
                debugDescription: "Invalid string representation of double value"
            )
            throw EncodingError.invalidValue(str, error)
        }
        wrappedValue = value
    }

    func encode(to encoder: Encoder) throws {
        var container = encoder.singleValueContainer()
        try container.encode(String(wrappedValue))
    }
}
```

虽然一开始看起来这里有很多代码，但是它们大部分都是模板代码或者错误处理。其中只有两行 (init 里对 decode 的调用以及 encode(to:) 里对 encode 的调用) 是与将数据在 Double 和字符串之间转换相关的。在下一部分，我们会看到编码和解码的具体过程，并且解释如何使用编解码容器。

把这个属性包装应用到 Coordinate 结构体的 latitude 和 longitude 值上就很简单了：

```
struct Coordinate: Codable {  
    @CodedAsString var latitude: Double  
    @CodedAsString var longitude: Double  
}  
  
let jsonData = try encoder.encode(places)  
let jsonString = String(decoding: jsonData, as: UTF8.self)  
/*  
[{"name": "Berlin", "coordinate": {"longitude": "13.0",  
"latitude": "52.0"}}, {"name": "Cape Town",  
"coordinate": {"longitude": "18.0", "latitude": "-34.0"}}]  
*/
```

使用属性包装来为特定的属性自定义编码和解码过程有两个巨大优势：首先，默认的编解码行为依然对其他所有属性生效（在我们的 Coordinate 例子中，已经没有其他属性了，但是在真实世界里，你通常只会需要对好多属性中的一个进行自定义）。第二点，我们可以很容易地在其他地方重用像是 CodedAsString 这样的变形。

相比起单个属性上的属性包装所能提供的行为，如果你需要更多的自定义功能，你就必须手动实现 init(from:) 和 encode(to:) 了，我们将在下一节详细介绍它们。

## 编码过程

如果你感兴趣的只是使用 Codable 系统，而且默认行为已经满足了你的需求的话，你就可以不用继续读下去了。不过如果你想了解如何自定义类型编码方式的话，我们还需要再进行一些深入挖掘。编码过程是如何工作的？在我们声明一个类型遵从 Codable 时，编译器为我们生成的代码到底是什么？

当你开始编码过程时，编码器会调用正在被编码的值上的 encode(to: Encoder) 方法，并将自身作为参数传递给它。接下来，如何用编码器进行正确的编码，就是值自身的责任了。

在上面的例子中，我们将 Placemark 数组传递到 JSON 编码器中：

```
let jsonData = try encoder.encode(places)
```

编码器(或者说是实际满足了 Encoder 的私有类)将会调用 places.encode(to: self)。那么，数组又是如何将自己编码为编码器可以理解的格式的呢？

## 容器

让我们来看看 Encoder 协议，这是编码器暴露给被编码值的接口：

```
/// 一个可以把值编码成某种外部表现形式的类型。  
public protocol Encoder {  
    /// 编码到当前位置的编码键 (coding key) 路径  
    var codingPath: [CodingKey] { get }  
    /// 用户为编码设置的上下文信息。  
    var userInfo: [CodingUserInfoKey : Any] { get }  
    /// 返回一个容器，用于存放多个由给定键索引的值。  
    func container<Key: CodingKey>(keyedBy type: Key.Type)  
        -> KeyedEncodingContainer<Key>  
    /// 返回一个容器，用于存放多个没有键索引的值。  
    func unkeyedContainer() -> UnkeyedEncodingContainer  
    /// 返回一个适合存放单一值的编码容器。  
    func singleValueContainer() -> SingleValueEncodingContainer  
}
```

现在先忽略 codingPath 和 userInfo，显然 Encoder 的核心功能就是提供一个**编码容器**(encoding container)。一个容器就是编码器内部存储的一种沙盒视图。通过为每个要编码的值创建一个新的容器，编码器能够确保每个值都不会覆盖彼此的数据。

容器有三种类型：

- **键容器 (Keyed Container)** — 它们用于编码键值对。可以把键容器想像为一个特殊的字典，这是到目前为止，应用最普遍的容器。

键容器内部使用的键是强类型的，这为我们提供了类型安全和自动补全的特性。编码器最终会在写入目标格式（比如 JSON）时，将键转换为字符串（或者数字），不过这对开发者来说是隐藏的。修改编码后的键名是最简单的一种自定义编码方式的操作，我们将会在下面看到一些相关的例子。

- **无键容器 (Unkeyed Container)** — 它们用于编码一系列值，但不需要对应的键，可以将它想像成保存编码结果的数组。因为没有对应的键来确定某个值，所以对无键容器中的值进行解码的时候，需要遵守和编码时同样的顺序。
- **单值容器 (Single-value Container)** — 它们对单一值进行编码。你可以用它来处理只由单个属性定义的那些类型。例如：Int 这样的原始类型，或以原始类型实现了 RawRepresentable 协议的枚举。

对于这三种容器，它们每个都对应了一个协议，来约束容器应该如何接收一个值并进行编码。

下面是 SingleValueEncodingContainer 的定义：

```
/// 支持存储和直接编码无索引单一值的容器。  
public protocol SingleValueEncodingContainer {  
    /// 编码到当前位置的编码键路径。  
    var codingPath: [CodingKey] { get }  
  
    /// 编码空值。  
    mutating func encodeNil() throws  
  
    /// 编码原始类型的方法  
    mutating func encode(_ value: Bool) throws  
    mutating func encode(_ value: Int) throws  
    mutating func encode(_ value: Int8) throws  
    mutating func encode(_ value: Int16) throws  
    mutating func encode(_ value: Int32) throws  
    mutating func encode(_ value: Int64) throws  
    mutating func encode(_ value: UInt) throws  
    mutating func encode(_ value: UInt8) throws  
    mutating func encode(_ value: UInt16) throws  
    mutating func encode(_ value: UInt32) throws
```

```
mutating func encode(_ value: UInt64) throws
mutating func encode(_ value: Float) throws
mutating func encode(_ value: Double) throws
mutating func encode(_ value: String) throws

mutating func encode<T: Encodable>(_ value: T) throws
}
```

可以看到，这个协议主要对 Bool, String, 各种整数以及浮点数声明了一系列 encode(::\_) 重载方法。另外，还有一个专门对 null 值进行编码的方法。所有的编码器和解码器都必须支持这些原始类型，而且所有的 Encodable 类型从根本上来说，也都必须归结到这些类型。[Swift 进化提案](#)中在介绍 Codable 系统的时候说道：

这些重载为编码提供了静态强类型的保证，这可以避免意外地编码不可用的类型。同时它们也为用户提供了一份可以依靠的常用原始类型列表，所有的编码器和解码器都支持对这些类型的编解码操作。

其他不属于原始类型的值，最后都会落到泛型的 encode<T: Encodable> 重载中。在这个方法里，容器最终会调用参数的 encode(to: Encoder) 方法，这使得整个过程会下降一个层级并重新开始，最终到达只剩下原始类型的情况。不过容器可以对不同的类型有不同的特殊要求。例如，这时候 JSONEncoder 在编码 Data 时会检查编码策略，比如是否编码成 Base64 字符串 (Data 默认的编码行为是把它自己编码到一个 UInt8 字节的无键容器里)。

UnkeyedEncodingContainer 和 KeyedEncodingContainerProtocol 拥有和 SingleValueEncodingContainer 相同的结构，不过它们具备更多的能力，比如可以创建嵌套的容器。如果你想要为其它数据格式创建编码器或解码器，那么最重要的部分就是实现这些容器。

## 值是如何对自己编码的

回到之前的例子，我们要编码的顶层类型是 Array<Placemark>。而无键容器是保存数组编码结果的绝佳场所 (因为数组说白了就是一串值的序列)。因此，数组将会向编码器请求一个无键容器。然后，对自身的元素进行迭代，并告诉容器对这些元素一一进行编码。把这个过程用代码表示出来，是这样的：

```
extension Array: Encodable where Element: Encodable {
    public func encode(to encoder: Encoder) throws {
        var container = encoder.unkeyedContainer()
        for element in self {
            try container.encode(element)
        }
    }
}
```

数组中的元素是 Placemark 实例。之前我们已经说过，对于非原始类型的值，容器将继续调用这个值的 encode(to:) 方法。

## 合成的代码

要继续研究，我们就需要知道为 Placemark 结构体实现 Codable 协议的时候，编译器究竟合成了哪些代码。让我们一步步来。

### Coding Keys

首先，在 Placemark 里，编译器会生成一个叫做 CodingKeys 的私有枚举类型：

```
struct Placemark {
    // ...

    private enum CodingKeys: CodingKey {
        case name
        case coordinate
    }
}
```

这个枚举包含的成员与结构体中的存储属性一一对应。而枚举值即为键容器编码对象时使用的键。和字符串形式的键相比，因为编译器会检查拼写错误，所以这些强类型的键要更加安全和

方便。不过，编码器最后为了存储需要，还是必须要能将这些键转为字符串或者整数值。而完成这个转换任务的，就是 CodingKey 协议：

```
/// 该类型作为编码和解码时使用的键
public protocol CodingKey {
    /// 在一个命名集合 (例如：以字符串作为键的字典) 中的字符串值。
    var stringValue: String { get }
    /// 在一个整数索引的集合 (一个整数作为键的字典) 中使用的值。
    var intValue: Int? { get }
    init?(stringValue: String)
    init?(intValue: Int)
}
```

所有键都必须可以用字符串的形式表示，另外，一个键类型也可以提供和整数互相转换的能力。如果使用整数更高效，编码器会选择整数形式的键。但它们也可以完全忽略掉这个特性而坚持使用字符串键，而 JSONEncoder 就是这么做的。因此，编译器合成的默认代码也只包含了字符串键。

## encode(to:) 方法

下面是编译器为 Placemark 结构体生成的 encode(to:) 方法：

```
struct Placemark: Codable {
    // ...
    func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: CodingKeys.self)
        try container.encode(name, forKey: .name)
        try container.encode(coordinate, forKey: .coordinate)
    }
}
```

和编码 Placemark 数组时的主要区别是，Placemark 会将自己编码到一个键容器中。对于那些拥有多个属性的复合数据类型（例如结构体和类），使用键容器是正确的选择（这里有一个例外，就是 Range，它 使用无键容器来编码上下边界）。注意代码中，Placemark 从编码器申请键容器

时，是如何通过 `CodingKeys.self` 指定容器中的键值的。接下来的所有编码命令都必须使用与之相同的类型。由于键类型通常都是被编码类型私有的，因此，当实现 `encode(to:)` 方法时，不小心使用了其它类型的编码键几乎是不可能发生的事情。

编码过程的结果，最终是一棵嵌套的容器树。JSON 编码器可以根据树中节点的类型把这个结果转换成对应的目标格式：键容器会变成 JSON 对象 ({ ... }), 无键容器变成 JSON 数组 ([ ... ]), 单值容器则按照它们的数据类型，被转换为数字，布尔值，字符串或者 `null`。

## init(from:) 初始化方法

当我们调用 `try decoder.decode([Placemark].self, from: jsonData)` 时，解码器会按照我们传入的类型 (这里是 `[Placemark]`)，使用 `Decodable` 中定义的初始化方法创建一个该类型的实例。和编码器类似，解码器也管理一棵由解码容器 (**decoding containers**) 构成的树，树中所包含的容器我们已经很熟悉了，它们还是键容器，无键容器，以及单值容器。

每个被解码的值会以递归方式向下访问容器的层级，并且使用从容器中解码出来的值初始化对应的属性。如果某个步骤发生了错误 (比如由于类型不匹配或者值不存在)，那么整个过程都会失败，并抛出错误。

因此，编译器为 `Placemark` 生成的解码初始化方法看上去是这样的：

```
struct Placemark: Codable {  
    // ...  
    init(from decoder: Decoder) throws {  
        let container = try decoder.container(keyedBy: CodingKeys.self)  
        name = try container.decode(String.self, forKey: .name)  
        coordinate = try container.decode(Coordinate.self, forKey: .coordinate)  
    }  
}
```

## 枚举和原始表示

在上面，我们看到了 Swift 的代码合成为结构体生成 `Codable` 实现的方式 (对于 `class` 也一样)。不过，有一个例外：如果一个类型满足 `RawRepresentable` 协议并且它的 `RawValue` 是可编码

的原初类型(也就是 Bool、String、Float、Double 或者任意整数类型)之一的话，这个原始值将会被直接编码到一个单值容器内。

对于原始表示类型，一种常见情况是你会想要将枚举中的原始值进行编码。枚举甚至有特殊的语法，来让你在不需要手动实现初始化方法和 rawValue 属性的前提下(参看枚举一章了解更多内容)，就能让枚举可以被原始值表示。让我们来看一个上面所编码的 Surrounding 枚举简化后的原始表示版本：

```
enum Surrounding2: String, Codable {
    case land
    case inlandWater
    case ocean
}

struct Placemark2: Codable {
    var name: String
    var coordinate: Coordinate
    var surrounding: Surrounding2
}

let berlin = Placemark2(
    name: "Berlin",
    coordinate: Coordinate(latitude: 52, longitude: 13),
    surrounding: .land
)

let data = try JSONEncoder().encode(berlin)
String(decoding: data, as: UTF8.self)
/*
{"name":"Berlin","coordinate":{"longitude":13,"latitude":52},
"surrounding":"land"}
*/
```

Surrounding2 值的原始值(默认情况下就是枚举成员的名字)被一字不差地用来作为 JSON 字典中的 surrounding 键所对应的值。编译器也对原始表示的结构体或类使用相同的编码格式。

不过，要注意并非所有的枚举都是这样编码的：如果枚举不是原始表示值，那么它的值就会被编码到键容器中。这个键就是枚举成员的名字，它的值是一个包含了枚举成员的关联值的字典：

```
enum Surrounding3:Codable {
    case land
    case inlandWater(name: String)
    case ocean(name: String)
}

struct Placemark3:Codable {
    var name: String
    var coordinate: Coordinate
    var surrounding: Surrounding3
}

let greatBlueHole = Placemark3(
    name: "Great Blue Hole",
    coordinate: Coordinate(latitude: 17.32278, longitude: -87.534444),
    surrounding: .ocean(name: "Caribbean Sea")
)

let data2 = try JSONEncoder().encode(greatBlueHole)
String(decoding: data2, as: UTF8.self)

/*
{"name":"Great Blue Hole",
"coordinate":{"longitude":-87.53444399999999,
"latitude":17.322780000000002},
"surrounding":{"ocean":{"name":"Caribbean Sea"}}}
*/
```

和结构体一样，你可以为枚举定义你自己的编码键或者跳过这个成员。此外，你也可以定义一个 `<CaseName>CodingKeys` 类型来自定义关联值的标签（或者如果这个关联值有一个默认值的话，也可以跳过它）。例如，我们可以为 `Surrounding3` 添加一个 `OceanCodingKeys` 枚举来把关联值的标签从 `"name"` 重命名为其他东西。

# 手动遵守协议

如果你的类型有特殊要求，可以通过手动实现 `Encodable` 和 `Decodable` 协议来进行满足。好的地方在于，自动代码合成不是一件一锤子买卖的事儿，你可以选择想要覆盖的部分，然后依然把剩下的事情交给编译器来做。

## 自定义 Coding Keys

控制一个类型如何编码自己最简单的方式，是为它创建自定义的 `CodingKeys` 枚举（顺带一提，虽然自动合成的代码也使用枚举实现了 `CodingKey` 协议，但这个类型实际上也可以不是枚举）。它可以让我们用一种快速且声明式的方法，改变类型的编码方式。在这个枚举中，我们可以：

- 在编码后的输出中，用明确指定的字符串值重命名字段。
- 将某个键从枚举中移除，以此跳过与之对应字段。

想要设置一个不同的名字，我们需要明确将枚举的底层类型设置为 `String`。例如，下面的代码会把 `name` 在 JSON 中映射为 `"label"`，但保持 `coordinate` 的名字不变：

```
struct Placemark2: Codable {  
    var name: String  
    var coordinate: Coordinate  
  
    private enum CodingKeys: String, CodingKey {  
        case name = "label"  
        case coordinate  
    }  
  
    // 编译器合成的 encode 和 decode 方法将使用覆盖后的 CodingKeys。  
}
```

在下面的实现中，枚举里没有包含 `name` 键，因此编码时地图标记的名字将会被跳过，只有 GPS 坐标信息会被编码：

```
struct Placemark3: Codable {  
    var name: String = "(Unknown)"  
    var coordinate: Coordinate  
  
    private enum CodingKeys: CodingKey {  
        case coordinate  
    }  
}
```

注意我们给 `name` 属性赋了一个默认值。如果没有这个默认值，为 `Decodable` 生成的代码将会编译失败，因为编译器会发现在初始化方法中它无法给 `name` 属性正确赋值。

在编码阶段跳过一些暂时值有时候会很有用，因为它们很容易被重新计算过或者根本没必要保存，例如缓存，或者保存的某些繁重计算的结果。编译器可以自己过滤出标记为 `lazy` 的属性，但如果你想把普通的存储属性作为暂时值的话，就需要像上面这样自定义类型的编码键。

## 自定义的 `encode(to:)` 和 `init(from:)` 实现

如果你需要更多的控制，自己实现 `encode(to:)` 和/或 `init(from:)` 总是一个可行的办法。作为例子，我们来自定义 `Placemark` 类型在解码时处理缺失值的方式。下面是 `Placemark` 类型的另外一种定义，`coordinate` 属性现在是可选值：

```
struct Placemark4: Codable {  
    var name: String  
    var coordinate: Coordinate?  
}
```

默认情况下，如果输入数据中没有对应的值存在，`JSONDecoder` 将会用 `nil` 来初始化目标中的这个可选值属性。因此，现在我们的服务器发送的 JSON 数据中 "coordinate" 字段是可以不存在的：

```
let validJSONInput = """  
[  
    { "name": "Berlin" },  
    { "name": "Paris" }  
]"""
```

```
{ "name" : "Cape Town" }  
]  
"""
```

当我们让 `JSONDecoder` 将这个输入解码为 `Placemark4` 值的数组时，解码器将自动把 `coordinate` 设为 `nil`，一切都很好。现在假设服务器的配置是发送一个空的 JSON 对象来表示某个可选值空缺的情况，于是，发送的 JSON 就会变为这样：

```
let invalidJSONInput = """  
[  
 {  
   "name" : "Berlin",  
   "coordinate": {}  
 }  
]  
"""
```

当我们尝试解码这个输入时，解码器本来期待 `"latitude"` 和 `"longitude"` 字段存在于 `coordinate` 中，但是由于这两个字段实际并不存在，所以这会触发 `.keyNotFound` 错误：

```
do {  
    let inputData = invalidJSONInput.data(using: .utf8)!  
    let decoder = JSONDecoder()  
    _ = try decoder.decode([Placemark4].self, from: inputData)  
} catch {  
    print(error.localizedDescription)  
    // The data couldn't be read because it is missing.  
}
```

要让这些代码工作，我们可以重载 `Decodable` 的初始化方法，明确地捕获我们所期待的错误：

```
struct Placemark4: Codable {  
    var name: String  
    var coordinate: Coordinate?
```

```
// encode(to:) 依然由编译器合成

init(from decoder: Decoder) throws {
    let container = try decoder.container(keyedBy: CodingKeys.self)
    self.name = try container.decode(String.self, forKey: .name)
    do {
        self.coordinate = try container.decodeIfPresent(Coordinate.self,
            forKey: .coordinate)
    } catch DecodingError.keyNotFound {
        self.coordinate = nil
    }
}
```

现在，解码器就可以成功地解码这个错误的 JSON 了：

```
do {
    let inputData = invalidJSONInput.data(using: .utf8)!
    let decoder = JSONDecoder()
    let decoded = try decoder.decode([Placemark4].self, from: inputData)
    decoded // [Berlin (nil)]
} catch {
    print(error.localizedDescription)
}
```

当遇到其他错误，比如输入数据完全损坏，或者在 name 字段上发生任何问题时，解码过程依然会抛出异常。

在只有一两个类型需要处理时，这种自定义方式是不错的选择，但是它很难大规模运用。如果一个类型有很多属性的话，就算你只想要自定义其中一个，你也将会需要对每个字段都手写代码。让上面的例子变得非常棘手的原因，是 coordinate 属性存在了两种 JSON 变体会导致解析为 nil：要么在 JSON 中 "coordinate" 键直接不存在，要么 "coordinate" 的值是一个空的 JSON 对象。

因为 "coordinate" 字段可能会整个缺失，我们无法用属性包装的方法来提供一种更加优雅和可重用的方式，去接受空 JSON 对象并把它看作是 nil：就算我们的属性包装的 wrappedValue 是可选值，但属性包装本身肯定不是可选的，生成的解码代码会期望这个属性对应的键在 JSON 中存在。

如果我们稍微放宽要求，并假设在 JSON 数据中 "coordinate" 一定会存在，我们在把“空对象处理成 nil 时，”就可以不用在 Coordinate 上手动实现 CodingKeys 和 init(from:) 了。如果有这个假设，那么我们就能构建一个 NilWhenKeyNotFound 属性包装，让它满足 Decodable：

```
@propertyWrapper
struct NilWhenKeyNotFound<Value: Decodable>: Decodable {
    var wrappedValue: Value?
    init(wrappedValue: Value?) {
        self.wrappedValue = wrappedValue
    }
    init(from decoder: Decoder) throws {
        do {
            let container = try decoder.singleValueContainer()
            self.wrappedValue = try container.decode(Value.self)
        } catch DecodingError.keyNotFound {
            self.wrappedValue = nil
        }
    }
}
```

现在，我们可以把这个属性包装加到 coordinate 属性上了。如果 coordinate 的解码抛出了 keyNotFound 错误，那么 coordinate 就会被解码为 nil：

```
struct Placemark5: Decodable {
    var name: String
    @NilWhenKeyNotFound var coordinate: Coordinate?
}
```

注意我们只让 Placemark5 满足了 Decodable，这是因为我们的 `NilWhenKeyNotFound` 属性包装现在还只支持解码。不过，添加 Encodable 支持只不过是实现一个 `encode(to:)`，来在 `wrappedValue` 是 `nil` 时创建一个空的键容器就可以了。

想要了解更多有关在 Codable 系统的范围内处理杂乱数据的技巧，你可以阅读 Dave Lyon 关于这个话题的文章。Dave 对于这个问题，给出了一种基于协议的泛型解决方案。不过如果你可以控制输入的话，最好还是在问题的源头进行修正（让服务器返回有效的 JSON），而不是在之后的阶段再去对奇怪的数据进行处理。

## 常见的编码任务

在这一节，我们讨论一些你可能希望用 Codable 系统解决的常见任务，以及在使用 Codable 时可能遇到的潜在问题。

### 让其他人的代码满足 Codable

假设我们要把 Coordinate 换成 Core Location 框架中的 `CLLocationCoordinate2D`，`CLLocationCoordinate2D` 和 Coordinate 的结构完全一样，所以我们应该尽量避免重复造轮子。

不过问题是，`CLLocationCoordinate2D` 并不满足 Codable 协议。所以，编译器现在会（正确地）抱怨说它无法为 Placemark5 自动生成实现 Codable 的代码，因为它的 `coordinate` 属性不再是遵从 Codable 的类型了：

```
import CoreLocation

struct Placemark5: Codable {
    var name: String
    var coordinate: CLLocationCoordinate2D
}

// 错误：无法自动合成 'Decodable'/'Encodable' 的适配代码,
// 因为 'CLLocationCoordinate2D' 不遵守相关协议
```

就算它定义在其它模块里，我们可以让 CLLocationCoordinate2D 也遵守 Codable 吗？在扩展中给类型添加协议支持会造成一个错误：

```
extension CLLocationCoordinate2D: Codable {}  
// 错误: 不能在类型定义的文件之外通过扩展自动合成实现 'Encodable' 的代码。
```

Swift 只在两种情况下会自动合成协议实现的代码，分别是直接添加在类型定义上的协议，以及定义在同一个文件的类型扩展上的协议。因此，在我们的例子中，只能自己手工添加实现代码。不过即使这个限制不存在，通过扩展让一个不属于我们的类型适配 Codable 也并不是一个好主意。要是 Apple 决定在今后的 SDK 版本中自己来满足协议的话，怎么办？很可能 Apple 的实现与你自己的实现不兼容。也就是说，用我们自己的实现进行编码的结果，很可能在 Apple 的代码中无法解码，反之也是如此。而这就是问题所在了，因为解码器不知道自己到底应该使用哪个实现 - 它看到的只有这个值应该被解码为 CLLocationCoordinate2D 而已。

Apple 的工程师 Itai Ferber 写了很多关于 Codable 系统的东西，他给出了这样的建议：

实际上我会更进一步，并且建议在当你想要扩展别人的类型，使其满足 Encodable 或 Decodable 时，你几乎总是应该考虑写一个结构体把它封装起来，除非你有理由能够确信这个类型自己绝对不会去遵循这些协议。

在下一节，我们会看到一个用结构体进行封装的例子。而对于当前的问题，让我们先通过一个略有不同（但同样安全）的方案来解决：我们会为 Placemark5 提供我们自己的 Codable 实现，在那里直接对纬度和经度进行编码。这么做可以有效地对编码器和解码器隐藏 CLLocationCoordinate2D 的存在；从它们的角度来看，纬度和经度就好像是直接定义在 Placemark5 里的一样：

```
extension Placemark5 {  
    private enum CodingKeys: String, CodingKey {  
        case name  
        case latitude = "lat"  
        case longitude = "lon"  
    }  
}
```

```
func encode(to encoder: Encoder) throws {
    var container = encoder.container(keyedBy: CodingKeys.self)
    try container.encode(name, forKey: .name)
    // 分别编码纬度和经度
    try container.encode(coordinate.latitude, forKey: .latitude)
    try container.encode(coordinate.longitude, forKey: .longitude)
}

init(from decoder: Decoder) throws {
    let container = try decoder.container(keyedBy: CodingKeys.self)
    self.name = try container.decode(String.self, forKey: .name)
    // 从纬度和经度重新构建 CLLocationCoordinate2D
    self.coordinate = CLLocationCoordinate2D(
        latitude: try container.decode(Double.self, forKey: .latitude),
        longitude: try container.decode(Double.self, forKey: .longitude)
    )
}
}
```

上面这个例子就很好地展示了当编译器无法自动生成 Codable 实现时 (当然，这种情况下，编译器也不会为我们合成实现 CodingKey 协议的代码)，我们不得不为每个类型手工编写的代码模板。

当编译器不能为我们生成上面的例子中的这些模板代码时，我们就必须为每个类型都自己写出这些代码 (为满足 CodingKey 协议的代码生成在这里也没有进行)。

另一种方案是使用嵌套容器来编码经纬度。KeyedDecodingContainer 有一个叫做 nestedContainer(forKey:) 的方法，它可以在 forKey 指定的键上，新建一个嵌套的键容器 (译注：想象一下，原本这个键对应的应该是在原始容器中保存的编码结果)，这个嵌套键容器使用 keyedBy 参数指定的另一套编码键。于是，我们只要再定义一个实现了 CodingKeys 的枚举，用它作为键，在嵌套的键容器中编码纬度和精度就好了 (这里我们只给出了 Encodable 的实现；Decodable 也遵循同样的模式)：

```
struct Placemark6: Encodable {
```

```
var name: String
var coordinate: CLLocationCoordinate2D

private enum CodingKeys: CodingKey {
    case name
    case coordinate
}

// 嵌套容器的编码键
private enum CoordinateCodingKeys: CodingKey {
    case latitude
    case longitude
}

func encode(to encoder: Encoder) throws {
    var container = encoder.container(keyedBy: CodingKeys.self)
    try container.encode(name, forKey: .name)
    var coordinateContainer = container.nestedContainer(
        keyedBy: CoordinateCodingKeys.self, forKey: .coordinate)
    try coordinateContainer.encode(coordinate.latitude, forKey: .latitude)
    try coordinateContainer.encode(coordinate.longitude, forKey: .longitude)
}
}
```

这样，我们就在 Placemark 结构体里，有效地重建了 Coordinate 类型的编码方式，而没有向 Codable 系统暴露这个内嵌的类型。当然，无论是直接编码、还是使用嵌套容器，这两种方式生成的 JSON 结果是完全相同的。

可以看到，无论上面哪种情况，我们都要写很多代码。对这个特定的例子，我们推荐一种不同的策略。这次，在 Placemark 里，我们定义一个 Coordinate 类型的私有属性 \_coordinate，用它存储位置信息。然后，给用户暴露一个 CLLocationCoordinate2D 类型的计算属性 coordinate。这次，由于 Coordinate 已经实现了 Codable，因此整个 Placemark 类型就自动是一个实现 Codable 的类型了。所以，我们唯一要做的事情，就是在 CodingKeys 枚举中，重

命名 `_coordinate` 对应的键，让它和暴露给用户的属性同名。这样，用户仍旧可以像之前一样使用 `coordinate`，而 `Codable` 系统则会完全忽略它，因为它只是一个计算属性：

```
struct Placemark7:Codable {
    var name: String
    private var _coordinate: Coordinate
    var coordinate: CLLocationCoordinate2D {
        get {
            return CLLocationCoordinate2D(latitude: _coordinate.latitude,
                longitude: _coordinate.longitude)
        }
        set {
            _coordinate = Coordinate(latitude: newValue.latitude,
                longitude: newValue.longitude)
        }
    }

    private enum CodingKeys: String, CodingKey {
        case name
        case _coordinate = "coordinate"
    }
}
```

这种方式之所以可以良好工作，是因为 `CLLocationCoordinate2D` 是一个很简单的类型，而且它与我们自定义类型的相互转换也非常容易。

## 让类满足 `Codable`

在之前的章节里我们已经看到了，对于任意的值类型，我们都可以（但是不建议）通过追加的方式让其满足 `Codable`。不过对于非 `final` 的类来说，情况就不是这样了。

作为一般性的原则，`Codable` 系统也能用在类上，但由于还可能存在子类，事情会因此变得更加复杂。打个比方，如果我们试图让 `UIColor` 满足 `Decodable` 的话（我们这里先暂时忽略

Encodable, 因为它和这个讨论无关), 会发生什么呢? 下面的例子取自 Jordan Rose 在 Swift 论坛上关于进化方向讨论的一个帖子。

UIColor 的一个自定义的 Decodable 实现看起来可能是这样的:

```
extension UIColor: Decodable {
    private enum CodingKeys: CodingKey {
        case red
        case green
        case blue
        case alpha
    }

    // 错误: 在一个可继承的类里, `Decodable` 约束的 'init(from:)'
    // 初始化方法只能通过在类定义中的一个`必须的`初始化方法满足。
    public init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        let red = try container.decode(CGFloat.self, forKey: .red)
        let green = try container.decode(CGFloat.self, forKey: .green)
        let blue = try container.decode(CGFloat.self, forKey: .blue)
        let alpha = try container.decode(CGFloat.self, forKey: .alpha)
        self.init(red: red, green: green, blue: blue, alpha: alpha)
    }
}
```

上面的代码无法编译, 它有好几个错误, 最终它们可以归结到一个不可解决的冲突: 只有必须的初始化方法 (**required initializers**) 才能满足协议的要求, 而这类方法不能添加在扩展里, 它们必须直接添加在类定义中。

一个必须的初始化方法 (通过 **required** 关键字标记) 表示所有的子类都必须实现这个初始化方法。定义在协议中的初始化方法必须都是 **required** 的, 和协议的所有要求一样, 这能够保证对该初始化方法的调用都能动态地作用在子类上。编译器必须保证类似这样的代码能够正确工作:

```
func decodeDynamic(_ colorType: UIColor.Type,
```

```
from decoder: Decoder) throws -> UIColor {
    return try colorType.init(from: decoder)
}

let color = decodeDynamic(SomeUIColorSubclass.self, from: someDecoder)
```

要让这个动态派发正确工作，编译器需要在类的派发表中为 Decodable 约束的初始化方法创建一条记录。但这个表是在类定义被编译的时候创建的，它的大小是固定的，不能通过扩展再向其中添加新的记录。这就是为什么 required 初始化方法只能在类定义中存在的原因。

长话短说，我们不能为一个非 final 的类用扩展的方式事后追加 Codable 特性。在我们上面提到的帖子中，Jordan Rose 讨论了一系列场景，来说明 Swift 今后如何能让这些代码工作：比如可以允许 required 初始化方法是 final 的（这样它就不需要在派发表中有一个条目），比如可以添加运行时的检查，如果子类没有提供 required 初始化方法所调用的指定初始化方法（designated initializer），则让程序中断。

不过即使这样，为不属于你的类型添加 Codable 还是问题重重，我们必须要面对这个事实。上一节说过，推荐的方式是写一个结构体来封装 UIColor，并且对这个结构体进行编解码。

一种方法是把 UIColor 的值存在一个包装结构体的属性中，然后为它实现一个自定义的 Codable 实现，为颜色的红、绿、蓝和透明度值定义 CodingKey。不过，我们也可以写一个含有颜色组件作为存储属性的结构体，然后在需要时从这些组件中创建出一个 UIColor。这种做法的好处是我们可以完全依靠编译器为这个包装结构体生成代码：

```
@propertyWrapper
struct CodedAsRGBA: Codable {
    private var red: CGFloat = 0
    private var green: CGFloat = 0
    private var blue: CGFloat = 0
    private var alpha: CGFloat = 0

    var wrappedValue: UIColor {
        get { UIColor(red: red, green: green, blue: blue, alpha: alpha) }
        set { store(newValue) }
    }
}
```

```
init(wrappedValue: UIColor) {
    store(wrappedValue)
}

mutating func store(_ color: UIColor) {
    let success: Bool =
        color.getRed(&red, green: &green, blue: &blue, alpha: &alpha)
    if !success {
        fatalError("无效的颜色格式")
    }
}
```

注意，这个简单的包装只支持能够被表示为 RGBA 的 UIColor 实例。这个包装器的一个更完整的版本必须检查颜色空间，并将其与和颜色空间相关的组件一起进行存储。

通过将这个包装结构体正式定义成属性包装，我们就可以通过对外透明的方式来使用它：

```
struct ColoredRect: Codable {
    var rect: CGRect
    @CodedAsRGBA var color: UIColor
}
```

现在，编码一个 ColorRect 数组得到的 JSON 结果就是这样的：

```
let rects = [ColoredRect(rect: CGRect(x: 10, y: 20, width: 100, height: 200),
    color: .yellow)]
do {
    let encoder = JSONEncoder()
    let jsonData = try encoder.encode(rects)
    let jsonString = String(decoding: jsonData, as: UTF8.self)
// [{"color":{"red":1,"alpha":1,"blue":0,"green":1},"rect":[[10,20],[100,200]]}]
} catch {
```

```
    print(error.localizedDescription)
}
```

## 解码多态集合

我们已经看到过了，解码器要求我们为要解码的值传入具体的类型。直觉上这很合理：解码器需要知道具体的类型才能调用合适的初始化方法，而且由于被编码的数据一般不含有类型信息，所以类型必须由调用者来提供。这种对强类型的强调导致了一个结果，那就是在解码步骤中不存在多态的可能。

例如，我们想编码一个 UIView 的数组，数组中的元素则是 UILabel 或 UIImageView 这样的 UIView 的子类：

```
let views: [UIView] = [label, imageView, button]
```

(让我们假设现在 UIView 和它的子类现在都满足 Codable，尽管现在它们并不是这样的类型。)

如果我们编码这个数组，再对它进行解码，就会发现得到的结果和原来 views 并不相同 - 数组中元素的具体类型在解码回来后都消失了。解码器能还原回来的只是普通的 UIView 对象，因为它对被解码数据类型的全部了解就是 [UIView].self。

那么，我们应该如何编码这样的多态对象集合呢？最好的方式就是创建一个枚举，让它的每个 case 对应我们要支持的子类，而 case 的关联值则是对应的子类对象：

```
enum View: Codable {
    case view(UIView)
    case labelUILabel)
    case imageViewUIImageView)
    // ...
}
```

我们应该还需要编写两个便利函数，来把 UIView 包装到一个 View 值里，以及反过来从 View 里解出 UIView。这样一来，将源数组传递给编码器以及从解码器中取出它时，都只需要一个 map 就行了。

注意，这并不是一个动态的解决方案；每次我们想要支持新的子类时，都需要手动更新 View 枚举。这不是很方便，但是却情有可原，因为我们必须明确地告诉解码器代码中所能接受的每个类型的名字。其他方式可能会带来潜在的安全威胁，因为攻击者很可能通过操作程序包来初始化一些我们程序里未知的对象。

## 回顾

Swift 具有使用很少的代码就能在程序原生类型和常用数据格式之间进行无缝转换的能力，至少在大多数情况下如此，这为我们节省了成吨的编写胶水代码的工作。如果你可以同时在客户端和服务器上使用 Swift 的话， Codable 系统会更加强大，因为在所有平台上使用相同的类型可以确保编码格式的兼容性。而当你需要处理那些并非由你定义的不支持 Codable 的类型时，尽管有时会遇到一些不方便，但终究你还是可以重载它们默认的编码和解码行为。

本章中我们只讨论了传统的数据归档任务，不过你完全可以跳出这个把值和原始数据相互转换的思维局限，基于这个转换过程执行的标准化方法，去探索一些其他方面的应用。例如，你可以使用 Decodable 系统取代反射，从可解码的值中生成 SQL 查询。或者你还可以写一个为每种原始数据类型产生随机值的解码器，并让它在单元测试中生成随机测试数据。

这就是说，当你把 Codable 应用在以类型安全的方式处理已知格式的统一数据 (译注：指的是数据格式的每一个部分都有对应的 Swift 类型) 时，这套系统将在这个被设计之初就用来针对的场景中熠熠生辉。除了常见情况之外，当试图自定义 Codable 系统以满足你的需求时，会存在一个平衡点，可能导致自定义需要的工作量反而大于节省的工作量。Swift 核心团队在 2021 年 3 月的论坛帖子 中也承认了这一点：

核心团队认为有必要分享一下：虽然 Codable 在 Swift 生态系统中发挥了关键作用，但核心团队并没有将 Codable 视为 Swift 中序列化的最终状态。在设计上， Codable 只解决了序列化需求的一个子集，而 Swift 需要获得更多的能力来拥有一个更完整的序列化解决方案。

核心团队希望与社区开展对话，以收集需求并讨论未来的设计及其权衡，这包括对 Codable 的改进以及额外的工具和 API。我们的目标是利用在这个帖子中收集到的信息，来为未来的提案提供参考，从而为 Swift 带来更完整的序列化解决方案。

让我们对这些讨论能达成怎样的结果拭目以待吧。

# 互用性

15

Swift 的一个强大之处在于，当你把它和 Objective-C 以及 C 混用的时候，阻力非常小。Swift 可以自动桥接 Objective-C 的类型，它甚至可以桥接很多 C 的类型。这让我们可以使用现有的代码库，并且在其基础上提供一个漂亮的 API 接口。

在本章中，我们将用 Swift 封装一个用 C 实现的程序库：[CommonMark](#)。CommonMark 是 Markdown 的一种正式规范。如果你曾经在 GitHub 或者 Stack Overflow 上写过东西的话，那你应该已经用过 Markdown 了，它是一种很流行的对纯文本进行格式化的语法。在这个实际的例子之后，我们会研究一下标准库中提供的操作内存的工具，以及如何使用它们与 C 代码进行交互。

## 封装一个用 C 编写的程序库

Swift 调用 C 代码的能力让我们可以很容易地使用大量已经存在的 C 的代码库。C 提供的 API 通常都比较笨重，其使用的内存管理方式也较为复杂。但无论如何，用 Swift 对一个程序的接口进行封装，通常还是比重新发明轮子要简单得多，工作量也少得多。同时，封装得当的话，用户将不会感觉这个封装和用 Swift 原生实现的 API 在类型安全以及易用性上有任何区别。而整个封装工作，只需要一个动态库和它的 C 语言头文件，就可以开始了。

我们要封装的这个 C 语言的 CommonMark 库，是一个 CommonMark 标准的参考实现，这个实现非常高效，测试也很齐全。在这个教程中，为了通过 Swift 访问它，我们将采用层层递进的方式进行封装。首先，围绕库暴露给外界的不透明类型 (opaque type)，我们创建一个简易的 Swift 类。然后，我们会将这个类包装成 Swift 枚举，以此提供更符合 Swift 风格的 API。

## 设置包管理器

相比之前，为导入 C 程序库设置一个 Swift 包管理器项目已经不是什么难事儿了，不过还是有不少步骤要完成，下面就是一份各种准备工作的流水账。

首先，通过 macOS 上的包管理软件 [Homebrew](#) 来安装 [cmark](#) 程序库。打开终端，然后执行下面的命令：

```
$ brew install cmark
```

如果你正在使用其它操作系统，用你系统上的包管理器安装 cmark 也就是了。在写作这本书的时候，cmark 的最新版本是 0.30.2。

然后，设置一个新的 SwiftPM 项目。切换到要保存代码的目录，执行下面的命令创建一个生成可执行程序的 SwiftPM 包，这会为我们的项目生成一个叫做 CommonMarkExample 的子目录：

```
$ mkdir CommonMarkExample  
$ cd CommonMarkExample  
$ swift package init --type executable
```

至此，你可以执行下 `swift run` 来检查是否一切工作正常。Swift 包管理器应该可以构建并执行这个应用程序，并在控制台上打印出 "Hello, world!"。

现在你需要让 Swift 能找到这个用 C 编写的 cmark 库，这样，才能通过 Swift 调用它。在 C 里，可以通过 `#include` 一个或多个程序库的头文件，让这些程序库中的签名在代码中可见。但 Swift 无法直接处理 C 的头文件，它的依赖关系是基于模块 (**modules**) 的。为了让一个用 C 或 Objective-C 编写的程序库对 Swift 编译器可见，它们必须按照 Clang 模块的格式提供一份 模块地图 (**module map**)。这份地图最重要的功能，就是列出构成模块的头文件。

因为 cmark 程序库并没有提供模块地图，所以你的下一个任务就是在 SwiftPM 里定义一个专门生成地图文件的目标 (target)。这个 target 不包含任何代码，它唯一的作用就是把 cmark 程序库包装成一个模块。

为此，打开 `Package.swift`，并添加下面的内容：

```
// swift-tools-version:5.5  
  
import PackageDescription  
  
  
let package = Package(  
    name: "CommonMarkExample",  
    dependencies: [],  
    targets: [  
        .executableTarget(  
            name: "CommonMarkExample",
```

```
dependencies: ["Ccmark"]),
.systemLibrary(
    name: "Ccmark",
    pkgConfig: "libcmark",
    providers: [
        .brew(["cmark"]),
        .apt(["cmark"]),
    ],
)
```

(为了看起来简单，我们去掉了文件中的所有注释以及 SwiftPM 为了执行测试而创建的 target。当然，你完全可以保留它们。)

你在包配置清单中，给 cmark 添加了一个系统程序库目标 (**system library target**)。在 SwiftPM 眼中，所谓的系统程序库指的是那些由操作系统这个级别的包管理器安装的程序库，例如：Homebrew，或者 Linux 上的 APT。任何指向这样的库的 SwiftPM target 都是系统程序库目标。作为惯例，这种纯包装类的模块名都应该使用 C 前缀，这就是为什么把 target 名字设置成 Ccmark 的原因。

pkgConfig 参数指定了 config 文件的名字，包管理器可以通过它找到要导入的库的头文件和库搜索路径。providers 指令是可选的。它可以在目标库没有被安装时，为包管理器提供一些用于显示的安装步骤的提示。

注意，在包配置清单中，你还要在主应用程序 target 中把 "Ccmark" target 作为依赖关系。这是通过 dependencies: ["Ccmark"] 完成的。

接下来，为系统程序库 target 创建一个保存代码的目录，这是保存模块地图的地方：

```
$ mkdir Sources/Ccmark
```

在你编写模块地图前，先在这个目录中创建一个名为 shim.h 的 C 头文件。它应该只包含下面这行代码：

```
#include <cmark.h>
```

最后，`module.modulemap` 看上去应该是这样的：

```
module Ccmark [system] {
    header "shim.h"
    link "cmark"
    export *
}
```

`shim` 头文件的作用是绕过模块地图中必须包含绝对路径的限制。如果你想要去掉 `shim` 文件的话，就得在模块地图中直接指定 `cmark` 头文件，比如 `header "/usr/local/include/cmark.h"`。这样的话，`cmark.h` 的路径将被硬编码到模块地图中。通过使用 `shim`，包管理器将会从 `pkg-config` 文件中读取正确的头文件搜索路径，并将它添加到编译器的调用中去。

现在，你应该就可以 `import Ccmark` 并调用其中任意的 API 了。为了快速确认是否一切正常，在 `main.swift` 中加入下面的代码：

```
import Ccmark

let markdown = "*Hello World*"
let cString = cmark_markdown_to_html(markdown, markdown.utf8.count, 0)!
defer { free(cString) }
let html = String(cString: cString)
print(html)
```

回到终端，运行程序：

```
$ swift run
```

如果你能看到 `<p><em>Hello World</em></p>` 这样的输出，则意味着你已经成功在 Swift 中调用了 C 函数！现在你已经有了一个可以工作的环境，接下来，就可以开始构建 Swift 封装了。（如果要使用 Xcode 编辑和运行你的代码，你可以直接在 Xcode 里打开这个包的目录。）

你也可以通过创建一个 C 语言 target 的方式把 C 代码嵌入到包里。根据库的不同，想要正确构建可能要花很多工夫，但是它确实有自己的好处，那就是包的用户不必再安装动态库。当你面向的是 iOS 这样的平台，或者甚至是在写一个 macOS app 时，你不能假设像是 CommonMark 这样的库是被安装了的，所以创建一个 C 语言 target 会使更好的选择。Apple 在它自己的 [cmark 分叉项目](#) 中就这么做了，它形成了 [Swift Markdown](#) 包的基础。

## 封装 CommonMark 程序库

现在所有东西都准备就绪了，作为开始，让我们把一个单独的函数封装到更好的接口里。`cmark_markdown_to_html` 接受 Markdown 格式的文本，并返回对应的 HTML 代码。在 C 中，它的签名是这样的：

```
/// 将 'text' (假设是 UTF-8 编码的字符串，且长度为 'len')  
/// 从 CommonMark Markdown 转换为 HTML，  
/// 返回一个以 null 结尾的 UTF-8 编码的字符串。  
/// 调用者负责对返回的缓冲区进行释放。  
char *cmark_markdown_to_html(const char *text, int len, int options);
```

当 Swift 导入这个声明的时候，它会把第一个表达 C 字符串的参数定义成 `UnsafePointer`，也就是一个指向若干 `CChar` (根据目标平台，这个类型时 `Int8` 或者 `UInt8` 的别名) 值的指针。通过文档我们知道，这些值都应该是 UTF-8 的编码单元。而 `len` 则是字符串的长度：

```
// Swift 中的函数接口。  
func cmark_markdown_to_html  
(_ text: UnsafePointer<CChar>!, _ len: Int, _ options: Int32)  
-> UnsafeMutablePointer<CChar>!
```

当然了，我们希望包装过的函数可以接受 Swift 字符串，所以你可能会想，我们需要把 Swift 字符串转换为一个 `CChar` 指针，然后再传递给 `cmark_markdown_to_html`。不过，桥接 Swift 字符串和 C 字符串是一个非常常见的操作，所以 Swift 会自动完成这件事情。而我们真正要小心的，是参数 `len`。它是 UTF-8 编码的字符串的字节数，而并不是字符串中的字符数。我们可以通过 Swift 字符串的 `utf8` 视图得到正确的值。最后，对于参数 `options`，我们传入 0 就可以了：

```
func markdownToHtml(input: String) -> String {  
    let outString = cmark_markdown_to_html(input, input.utf8.count, 0)!  
    defer { free(outString) }  
    return String(cString: outString)  
}
```

在上面的实现中，我们强制解包了函数的返回值。因为我们知道 `cmark_markdown_to_html` 肯定会返回一个有效的字符串，所以这么做是安全的。通过在方法内部进行强制解包，代码库的用户就可以在调用 `markdownToHTML` 的时候不必关心可选值的问题了，返回的结果一定不会为 `nil`。这是编译器无法自动为我们完成的事情，因为 C 和 Objective-C 的指针如果没有 nullable 标记，它们就总是会以隐式解包可选值 (`Implicitly Unwrapped Optional`, `IUO`) 的形式导入到 Swift 里。

注意，当 Swift 在原生字符串和 C 字符串之间自动完成桥接转换时，Swift 会假设了你调用的 C 函数接受的是 UTF-8 编码的字符串。这在绝大多数情况下是正确的，但也有些 C API 需要不同的字符串编码，这时你就不能用自动桥接了。不过，通常来说构建替代格式也很简单。比如，假设 C API 需要一个 UTF-16 编码点的数组的话，可以使用 `Array(string.utf16)`。只要元素类型匹配，Swift 编译器就会自动把 Swift 数组桥接为所需要的 C 数组。

另外还可以注意到，我们在 `markdownToHTML` 内部调用了 `free` 来释放 `cmark_markdown_to_html` 为了返回结果申请的内存。当与 C API 进行交互的时候，遵守 C 程序库的内存管理规则也是我们的义务，在这方面，编译器无法为我们提供任何帮助。

## 包装 `cmark_node` 类型

除了直接输出 HTML 之外，`cmark` 还提供了一种使用方式，可以将 Markdown 文本解析为结构化的节点树。举例来说，一串简单的文本可以被转换为一个基于文本块的节点列表，这些文本块可以是段落，引用，列表，代码块，标题等等。有些层级的文本块可以包含其他层级的元素（例如：引用可以包含多个段落），而有些则只能包含内联元素（例如：标题可以包含要突出显示的文本）。但一个节点不能同时包含文本块和内联元素（例如：一个列表条目中的内联元素一定是被包装在段落元素里的）。

C 代码库的实现用 `cmark_node` 这个单一的数据类型来表示节点。它是不透明的，也就是说，库的作者选择将它的定义隐藏起来。我们在头文件中能看到的只有对这个节点的操作，或返回 `cmark_node` 指针的函数。Swift 将这些指针导入为 `OpaquePointer`。(我们会在本章后面的部分仔细研究标准库中各种指针类型，例如 `OpaquePointer` 和 `UnsafeMutablePointer`，之间到底有什么区别。)

现在让我们将一个节点包装成一个 Swift 原生类型，这样用起来会更简单一些。我们在结构体和类中提到过，当创建一个自定义类型时，我们需要考虑存储的语义：这个类型是一个值么？还是说把它当作具有同一性的实例会更合适？如果是前者的话，我们应该使用结构体或者枚举；如果是后者，则应该使用类。但我们这里的情况很有意思：一方面，一个 Markdown 文档的节点应该是个值，因为两个具有相同类型和内容的节点不应该被认为是不同的东西，所以它们不应该拥有同一性；而另一方面，因为我们对 `cmark_node` 内部的信息一无所知，所以没有直接的方法可以复制一个节点，我们也因此无法保证它的值语义。因为这个原因，我们会先使用类来实现。稍后，我们将会在类的基础上再添加一层接口，来实现值语义。

我们的类只是简单地存储这个不透明指针，然后在 `deinit` 中释放 `cmark_node` 的内存，以保证这个类的实例不再拥有对节点的引用。我们只在整个文档的层级上释放内存，因为如果不这么做的话，我们可能会错误地释放那些还在使用的节点的内存。将文档进行释放也会造成所有的子节点自动被释放。通过这样的方式包装不透明指针将会让我们直接从自动引用计数中受益：

```
public class Node {
    let node: OpaquePointer

    init(node: OpaquePointer) {
        self.node = node
    }

    deinit {
        guard type == CMARK_NODE_DOCUMENT else { return }
        cmark_node_free(node)
    }
}
```

下一步是封装 `cmark_parse_document` 函数，这个函数会将 Markdown 解析为一个文档根节点。它接受的参数和 `cmark_markdown_to_html` 函数相同：要解析的 markdown 字符串，字符串的长度，以及一个代表解析选项的整数值。在 Swift 中，这个函数的返回类型是 `OpaquePointer`，它代表了这个文档节点：

```
func cmark_parse_document
(_ buffer: UnsafePointer<Int8>!, _ len: Int, _ options: Int32)
-> OpaquePointer!
```

我们将这个函数转换为类的初始化方法：

```
public init(markdown: String) {
    let node = cmark_parse_document(markdown, markdown.utf8.count, 0)!
    self.node = node
}
```

这次，我们同样强制解包了 `cmark_parse_document` 返回的指针，因为我们知道这个函数是不会失败的，任何字符串都是一个合法的 Markdown。和很多 C APIs 一样，`cmark` 程序库没有使用 `nullability` 标记指针，也没有明确的文档说明哪些指针可能为 NULL。如果你想确定一个指针是否可能为空，查看程序库的 C 代码应该是帮你解惑最合适的方法了。

上面提到过，`cmark` 有很多有意思的函数可以操作节点。比如说，有个函数可以返回节点的类型，用它可以判断一个节点是否是段落或者标题：

```
cmark_node_type cmark_node_get_type(cmark_node *node);
```

在 Swift 中，它被导入为：

```
func cmark_node_get_type(_ node: OpaquePointer!) -> cmark_node_type
```

`cmark_node_type` 是一个 C 的枚举，它包括了由 Markdown 定义的不同文本块和内联元素，以及一个表示错误的成员：

```
typedef enum {
    /* 错误状态 */
```

```
CMARK_NODE_NONE,  
  
/* 文本块 */  
CMARK_NODE_DOCUMENT,  
CMARK_NODE_BLOCK_QUOTE,  
...  
  
/* 内联元素 */  
CMARK_NODE_TEXT,  
CMARK_NODE_EMPH,  
...  
} cmark_node_type;
```

Swift 将 C 枚举导入为一个包含单个 UInt32 属性的结构体。除此之外，对原来枚举中的每个成员，Swift 还会为它生成一个全局常量：

```
struct cmark_node_type: RawRepresentable, Equatable {  
    public init(_ rawValue: UInt32)  
    public init(rawValue: UInt32)  
    public var rawValue: UInt32  
}  
  
var CMARK_NODE_NONE:cmark_node_type { get }  
var CMARK_NODE_DOCUMENT:cmark_node_type { get }  
...
```

C 枚举被导入成结构体是因为 C 中的枚举实际上就是整数。Swift 不得不假设一个 C 中的枚举变量值可以是任意整数值，但处理这种情况这并不是 Swift 原生枚举的设计意图。只有那些被 Apple 在 Objective-C 框架中用 NS\_ENUM 宏标记的枚举才会导入成 Swift 原生的枚举类型。

在 Swift 中，节点的类型应该是 Node 数据类型的一个属性，所以我们将 cmark\_node\_get\_type 函数转变为 Node 类中的一个计算属性：

```
var type:cmark_node_type {
```

```
cmark_node_get_type(node)
}
```

现在，我们就可以用 `node.type` 来获取一个元素的类型了。

我们还可以访问更多的节点属性。例如：对节点列表来说，它就会有一个属性表示列表的类型，包括：“无序列表”和“有序列表”。而所有非列表节点的列表类型则统一表示成“无列表”。同样，Swift 会将表示列表类型的 C 枚举映射为一个结构体，其中的每种列表类型都定义成一个顶层变量，因此，我们也可以用类似的方法将它们包装成 Swift 计算属性。这里，我们还为这个属性提供了 `setter`，在本章后面的部分我们会使用到它：

```
var listType: cmark_list_type {
    get { return cmark_node_get_list_type(node) }
    set { cmark_node_set_list_type(node, newValue) }
}
```

对于还没有提到过的其它节点属性，cmark 程序库都提供了类似的函数（例如标题的层级，代码块的信息，以及链接的 URL 和文字等）。但这些属性通常只对特定类型的节点有意义，在 Swift 里，我们可以选择使用可选值（比如对于链接的 URL）或者是默认值（比如对于标题来说默认层级为 0）为这些属性提供更好的建模。但由于缺乏类型安全的特性，这使得 C 版本的 API 在处理这些问题时表现力就要弱的多。我们会在下面继续讨论这个话题。

在 cmark 里，有些节点还可以拥有子节点，为了对这些子节点进行遍历，CommonMark 库提供了 `cmark_node_first_child` 和 `cmark_node_next` 函数。我们想要在 `Node` 类中提供一个子节点的数组。要生成这个数组，我们从第一个子节点开始，不断向数组中加入子节点，直到 `cmark_node_first_child` 或 `cmark_node_next` 返回 `nil` 为止，这表示我们已经遍历到了列表的结尾。要注意的是，`cmark_node_next` 返回的指针会被自动转换成一个 Swift 可选值，其中 `nil` 代表了 C 中空指针（`null`）的情况：

```
var children: [Node] {
    var result: [Node] = []
    var child = cmark_node_first_child(node)
    while let unwrapped = child {
        result.append(Node(node: unwrapped))
    }
}
```

```
    child = cmark_node_next(child)
}
return result
}
```

我们还可以选择返回一个 `lazy` 序列 (比如使用 `sequence` 函数或 `AnySequence` 类型) 而不是一个数组。不过，这里存在一个问题：就是在创建和开始使用序列之间，节点的结构可能发生变  
化。这样一来，使用迭代器寻找下一个节点的操作可能会返回不正确的值，或者更糟糕的话，  
会导致程序崩溃。根据你的使用场景，返回一个延迟创建的序列可能正是你需要的，但是如果  
你的数据结构会发生改变，返回一个数组才是更安全的选择。

有了这个包装节点的类，现在通过 Swift 访问 CommonMark 库生成的抽象语法树就方便多了。  
我们不再需要调用像是 `cmark_node_get_list_type` 这样的函数，而只需要通过访问  
`node.listType` 属性就可以了。除了简洁明了，这还给我们带来了自动补全和类型安全等诸多好  
处。不过，我们仍然有改进的余地。虽然现在使用 `Node` 类已经比原来的 C 函数要好很多了，  
但是 Swift 可以让我们以一种更加自然和安全的方式来表达这些节点，那就是使用带有关联值  
的枚举。

## 一个更安全的接口

我们上面提到过，有很多节点属性只在特定的上下文中有效。比如，访问一个列表节点的  
`headerLevel` 或者访问一个代码块的 `listType` 都是没有意义的。就像我们在枚举这一章看到过  
的，带有关联值的枚举可以让我们指定对于每种特定情况下哪些元数据是有意义的。因此，我  
们可以创建两个枚举，一个表示所有的内联元素，另一个表示所有的文本块。这两个枚举将成  
为 Swift 版程序库的公开接口，而 `Node` 类将成为程序库内部的实现细节。

通过这么做，我们就能强制结构化 CommonMark 的文档。举例来说，一个纯文本元素只能存  
储一个 `String`，而表示强调的节点则包含了一个其他内联元素的数组，但它无法包含其它文本  
块级别的子元素。以下是表示内联元素的枚举：

```
public enum Inline {
    case text(text: String)
    case softBreak
    case lineBreak
```

```
case code(text: String)
case html(text: String)
case emphasis(children: [Inline])
case strong(children: [Inline])
case custom(literal: String)
case link(children: [Inline], title: String?, url: String)
case image(children: [Inline], title: String?, url: String)
}
```

而对于文本块元素，哪些元素可以包含什么特定的子元素也有对应的规则。例如：段落和标题只能包含内联元素，而引用则可以包含其它的文本块元素。我们的 Block 类型是这样建模这些约束的：

```
public enum Block {
    case list(items: [[Block]], type: ListType)
    case blockQuote(items: [Block])
    case codeBlock(text: String, language: String?)
    case html(text: String)
    case paragraph(text: [Inline])
    case heading(text: [Inline], level: Int)
    case custom(literal: String)
    case thematicBreak
}
```

可以看到，一个列表 (list) 是通过列表项的数组 ([[Block]]) 定义的，每一个列表项则是通过一个文本块数组 ([Block]) 定义的。而 ListType 则是一个简单的枚举，它用来区别一个列表到底是有序列表还是无序列表：

```
public enum ListType {
    case unordered
    case ordered
}
```

因为枚举是值类型，通过把 cmark 中的节点转换成它们各自的枚举表现形式之后，我们就能把这些节点也当作值来看待了。

通过之前给不透明指针类型创建的包装类是无法实现这一点的。在接下来的开发中，我们将遵循 API 设计准则 的建议，通过初始化方法进行类型转换。我们创建了两对初始化方法：其中一对从 Node 类型中创建 Block 和 Inline 枚举值，另一对则依据枚举值重新构建对应的 Node。这让我们可以创建和操作 Inline 或者 Block 值，之后再根据这些值构建出来的 Node 对象重新组成 CommonMark 文档，并使用 C 代码库将其渲染为 HTML 或者转换回 Markdown 文本。

我们先来实现把 Node 转换为 Inline 的初始化方法。先使用 switch 对节点的类型进行选择，然后构建出对应的 Inline 值。比如说，如果遇到一个文本节点，我们就访问 cmark 库中节点的 literal 属性读出文本。这时，对 literal 进行强制解包是安全的，因为这种类型的节点一定是包含文本的。但访问其它节点的 literal 属性则有可能得到 nil。例如：斜体强调和粗体节点只包含子节点而没有 literal 值。为了解析这些节点，我们就得遍历它们的子节点，根据这些子节点的类型递归调用相应的初始化方法完成转换。为了避免重复的代码，我们可以创建一个会被按需调用的内联函数：inlineChildren。在匹配节点类型的 switch 语句中，应该永远不会执行到 default 分支，因此如果发生了这种情况，我们就选择杀死程序。之所以不选择返回可选值或抛出错误，是因为按照 Swift 约定，这些都是用来表达可预期错误的，而这里显然是一个程序员造成的错误：

```
extension Inline {  
    init(_ node: Node) {  
        let inlineChildren = { node.children.map(Inline.init) }  
        switch node.type {  
            case CMARK_NODE_TEXT:  
                self = .text(text: node.literal!)  
            case CMARK_NODE_STRONG:  
                self = .strong(children: inlineChildren())  
            case CMARK_NODE_IMAGE:  
                self = .image(children: inlineChildren(),  
                             title: node.title, url: node.urlString)  
            // ... (more cases)  
            default:  
                fatalError("Unrecognized node: \(node.typeString)")  
        }  
    }  
}
```

```
    }
}
}
```

我们可以使用同样的方式转换文本块级别的元素。不过需要注意的是，根据节点类型的不同，一个文本块级别的元素可以包含内联元素，列表条目或者其他文本块级别的元素。在 cmark\_node 语法树中，列表条目是被包装到一个额外的节点中的。在 Node 的 listItem 属性中，我们移除了这层包装，并且直接返回一个由文本块级别的元素组成的数组：

```
extension Block {
    init(_ node: Node) {
        let parseInlineChildren = { node.children.map(Inline.init) }
        switch node.type {
            case CMARK_NODE_PARAGRAPH:
                self = .paragraph(text: parseInlineChildren())
            case CMARK_NODE_LIST:
                let type: ListType = node.listType == CMARK_BULLET_LIST ?
                    .unordered : .ordered
                self = .list(items: node.children.map { $0.listItem }, type: type)
            case CMARK_NODE_HEADING:
                self = .heading(text: parseInlineChildren(), level: node.headerLevel)
            // ... (more cases)
            default:
                fatalError("Unrecognized node: \(node.typeString)")
        }
    }
}
```

现在，只要有一个文档级别的 Node，我们就可以把它转换为一个 Block 数组了：

```
extension Node {
    public var elements: [Block] {
        children.map(Block.init)
    }
}
```

```
}
```

其中的 Block 元素是值：我们可以随意地复制或者改变它们，而不需要担心会破坏原来的引用。这在操作节点的时候是非常强大的特性。因为按照值类型的特性，它们并不在意是如何被创建的，我们可以通过代码直接从头开始创建 Markdown 的语法树，而完全不必使用 CommonMark 库。节点的类型现在也更加清晰了，你不会再意外地编写一些不合理的代码，例如访问一个列表的标题之类的，因为编译器现在不允许你这么做了。除了让你的代码更加安全以外，这样的写法自身也是一种更加稳定的文档的形式。只需要看一眼类型，你就能知道一个 CommonMark 是如何被构建的。和注释不同，编译器将会保证这种形式永不过时。

现在，对我们的新的数据类型进行操作就易如反掌了。比如，我们想要从 Markdown 文档中构建一个包含所有一级标题和二级标题的数据作为目录，我们只需要对所有子节点进行循环，然后找出它们是不是标题，以及级别是否满足要求就可以了：

```
func tableOfContents(document: String) -> [Block] {
    let blocks = Node(markdown: document)?.children.map(Block.init) ?? []
    return blocks.filter {
        switch $0 {
        case .heading(_, let level) where level < 3: return true
        default: return false
        }
    }
}
```

但在继续更多操作之前，让我们先实现逆向转换，也就是把一个 Block 转换回 Node。之所以要实现这个转换，是因为如果想要直接使用 CommonMark 来从我们构建或者操作过的 Markdown 语法树中生成 HTML 或者是其他的文本格式时，cmark 程序库的 C API 只接受 cmark\_node\_type 类型的输入。

我们要做的是为 Node 添加两个初始化方法：一个负责将 Inline 值转换为节点，另一个负责处理 Block 元素。我们先将 Node 进行扩展，为它添加一个初始化方法，根据指定的类型和子节点从头开始创建一个新的 cmark\_node。还记得我们写过一个释放内存的 deinit 吗？它释放了以该节点为根的节点树（包括其下的子节点们）的占用的内存。这个 deinit 会保证我们在这里初始化的内容也可以被正确地释放：

```
extension Node {  
    convenience init(type: cmark_node_type, children: [Node] = []) {  
        self.init(node: cmark_node_new(type))  
        for child in children {  
            cmark_node_append_child(node, child.node)  
        }  
    }  
}
```

我们经常会创建只包含文本的节点，或者有一系列子节点的节点。所以，让我们写三个简便初始化方法来对应这些情况：

```
extension Node {  
    convenience init(type: cmark_node_type, literal: String) {  
        self.init(type: type)  
        self.literal = literal  
    }  
    convenience init(type: cmark_node_type, blocks: [Block]) {  
        self.init(type: type, children: blocks.map(Node.init))  
    }  
    convenience init(type: cmark_node_type, elements: [Inline]) {  
        self.init(type: type, children: elements.map(Node.init))  
    }  
}
```

现在，就可以用刚才编写的这些简便初始化方法来实现负责转换的初始化方法了。我们先根据输入判断出元素的类型，然后再创建对应类型的节点。以下是转换内联元素的实现：

```
extension Node {  
    convenience init(element: Inline) {  
        switch element {  
        case .text(let text):  
            self.init(type: CMARK_NODE_TEXT, literal: text)  
        case .emphasis(let children):  
        }
```

```
self.init(type: CMARK_NODE_EMPH, elements: children)
case .html(let text):
    self.init(type: CMARK_NODE_HTML_INLINE, literal: text)
case .custom(let literal):
    self.init(type: CMARK_NODE_CUSTOM_INLINE, literal: literal)
case let .link(children, title, url):
    self.init(type: CMARK_NODE_LINK, elements: children)
    self.title = title
    self.urlString = url
// ... (more cases)
}
}
}
```

为文本块级别的元素创建节点也是一样的。唯一的小区别在于列表的情况要复杂一些。希望你还记得，在上面将 Node 转换为 Block 的函数中，我们把 CommonMark 库里用来代表列表的额外的节点移除了，所以这里我们需要把这一层节点加回来：

```
extension Node {
convenience init(block: Block) {
switch block {
case .paragraph(let children):
    self.init(type: CMARK_NODE_PARAGRAPH, elements: children)
case let .list(items, type):
    let listItems = items.map { Node(type: CMARK_NODE_ITEM, blocks: $0) }
    self.init(type: CMARK_NODE_LIST, children: listItems)
    listType = type == .unordered
        ? CMARK_BULLET_LIST
        : CMARK_ORDERED_LIST
case .blockQuote(let items):
    self.init(type: CMARK_NODE_BLOCK_QUOTE, blocks: items)
case let .codeBlock(text, language):
    self.init(type: CMARK_NODE_CODE_BLOCK, literal: text)
}}
```

```
fenceInfo = language
// ... (more cases)
}
}
}
```

最后，为了给用户提供一个良好的接口，我们定义了一个公开的初始化方法，它接受一个文本块级别的元素组成的数组，并生成一个文档节点。稍后，我们将可以把这个节点渲染成不同的输出格式：

```
extension Node {
    public convenience init(blocks: [Block]) {
        self.init(type: CMARK_NODE_DOCUMENT, blocks: blocks)
    }
}
```

现在，我们就可以在两个方向自由穿梭了：我们可以加载一个文档，将它转换为 [Block] 元素，更改这些元素，然后再将它们转换回一个 Node。这让我们能够编写程序从 Markdown 中提出信息，或者甚至动态地改变这个 Markdown 的内容。

通过首先创建一个 C 库的简单封装层 (Node 类)，我们抽象了底层 C API 到 Swift 的转换过程。这让我们可以专注于提供符合 Swift 语言习惯的接口。你可以在 [GitHub](#) 上找到完整的项目。

## 底层类型概览

在标准库中有不少底层类型可以让我们直接访问到内存。这些类型的不仅数量繁多，而且还有诸如 UnsafeMutableRawBufferPointer 这样让人畏惧的名字。但好消息是，它们的命名方式都很统一，每个类型的目的都能够从它的名字中推断出来。下面是命名中几个最重要的部分：

- 含有 **managed** 的类型代表内存是自动管理的。编译器将负责为你申请，初始化并且释放内存。

- 含有 **unsafe** 的类型摒弃了 Swift 常规的安全特性，例如边界检查以及变量使用前必须初始化的保证。它也不提供自动化的内存管理 (这和 **managed** 正好相反)。你需要明确地进行内存申请，初始化，销毁和回收。
- 含有 **buffer** 类型作用于在连续存储空间上的多个元素，而非一个单独的元素上。因此，它也提供了 Collection 的接口。
- 含有 **pointer** 的类型拥有指针的语义 (和 C 中的指针是类似的)。
- 含有 **raw** 的类型包含无类型的原始数据，它和 C 的 **void\*** 是等价的。名称中不含有 **raw** 的类型包含的都是具体类型的数据，这些具体类型都是通过各自的泛型参数指定的。
- 含有 **mutable** 的类型允许修改它指向的内存。

如果你需要直接访问内存，并且不需要与 C 交互，你可以使用 `ManagedBuffer` 来申请内存。这和 Swift 标准库中集合类型在底层管理内存的方式是类似的。它由一个单独的 `header` 值 (用来存储像是元素个数这样的数据) 和一段连续的内存空间 (用来存储元素) 组成。它还有一个 `capacity` 属性，但这个属性并不等于实际保存的元素个数：例如，一个 `count` 是 17 的数组可能会有一个 `capacity` 为 32 的缓冲区，也就是说，数组在必须申请更多内存之前，还可以再容纳 15 个元素。这种类型还有一个变种，叫做 `ManagedBufferPointer`，但是它在标准库之外并没有太多应用场景，而且在 未来可能会被移除。

有时候你需要手动进行内存管理。比如，你可能需要将一个 Swift 对象传给某个 C 函数，完成处理之后，再把它取回来。为了绕开 C 没有闭包的限制，使用回调函数 (函数指针) 的 C API 通常还会接受一个指向上下文环境的参数 (通常是一个无类型指针，或者说 `void*`)，并在每次调用回调函数时，将这个参数回传。当你从 Swift 调用这样的函数时，如果能将一个原生的 Swift 对象作为上下文环境传递的话，会非常方便，不过 C 无法直接处理 Swift 对象。这时，就可以让 `Unmanaged` 类型出场了。它是一个类实例的包装，我们可以通过它获取一个指向其自身的原始指针，并把这个指针传递给 C API。由于包装在 `Unmanaged` 里的对象活动于 Swift 内存管理系统之外，我们只能自己留心 `retain` 和 `release` 的调用平衡。下一节，我们就会看到这样的例子。

## 指针

除了我们已经看到过的 `OpaquePointer` 类型以外，Swift 中还有另外八种指针类型，用来映射 C 中不同的指针。

UnsafePointer 是最基础的指针类型，它与 C 中的 `const` 指针类似，但是泛化了其指向数据的类型。所以，`UnsafePointer<Int>` 对应的就是 `const int*`。

注意，C 中 `const int*` (一个指向不可变数据的可变指针，你不能修改指针指向的数据) 和 `int* const` (一个不可变指针，或者说，你不能改变这个指针指向的位置) 是不一样的。

UnsafePointer 和前者是等效的。和其它的类型一样，你通过将指针变量声明为 `var` 或者 `let` 来控制指针本身的可变性。

## 针对函数参数的自动指针转换

你可以使用其他类型的指针，把它们传递给特定的初始化方法来创建 UnsafePointer 对象。对于那些接受 `UnsafePointer` 参数的函数，Swift 还提供了一种特殊的调用语法，只要在任意类型正确的可变变量前面加上 `&` 符号把它变成 `in-out` 表达式就好了：

```
var x = 5
func fetch(p: UnsafePointer<Int>) -> Int {
    p.pointee
}
fetch(p: &x) // 5
```

这看起来和我们在函数一章中提到的 `inout` 参数完全一样，而且它们的行为也是类似的。不过在这里，因为指针不是可变的，所以不会有任何东西通过这个指针传回给调用者。Swift 在幕后创建和传递给函数的指针只保证在函数调用期间是有效的。切记不要尝试返回这个指针，或者在函数返回之后再去访问它，这么做的结果是未定义的。

还有一个可变版本的指针，那就是 `UnsafeMutablePointer`。这个结构体的行为和普通的 C 指针很像，你可以对指针进行解引用，并修改内存的值，这些改动将通过 `in-out` 表达式的方式传回给调用者：

```
func increment(p: UnsafeMutablePointer<Int>) {
    p.pointee += 1
}
var y = 0
increment(p: &y)
```

```
y // 1
```

## 指针的生命周期

除了使用 `in-out` 表达式，你也可以通过直接申请内存的方式来使用 `UnsafeMutablePointer`。Swift 中申请内存的规则和 C 很相似：申请内存后，你必须初始化之后才能使用它。一旦你不再需要这个指针，你需要释放内存：

```
// 申请两个 Int 的内存，并初始化它们
let z = UnsafeMutablePointer<Int>.allocate(capacity: 2)
z.initialize(repeating: 42, count: 2)
z.pointee // 42
// 指针计算：
(z+1).pointee = 43
// 下标：
z[1] // 43
// 销毁内存
z.deallocate()
// 不要在 deallocate 之后再访问被指向的值
```

如果指针的 `Pointee` 类型（也就是指针指向的数据类型）是一个需要内存管理的非简单类型（例如在一个类或结构体中包含了其它类），你必须在调用 `deallocate` 之前调用 `deinitialize`。`initialize` 和 `deinitialize` 方法用于管理 ARC 中的引用计数。忘记调用 `deinitialize` 可能会引起内存泄漏。更糟的是，如果忘记调用 `initialize`，例如直接用下标操作符给指向一片未初始化内存的指针赋值，可以引发各种未定义的问题甚至让程序崩溃。

## 原始指针

在 C API 中，单纯指向一个字节序列的指针（比如 `void*` 或者 `const void*`）是很常见的，这些指针不指向任何具体的数据类型。在 Swift 中与之等价的是 `UnsafeMutableRawPointer` 和 `UnsafeRawPointer` 类型。C API 中的 `void*` 或 `const void*` 将会被导入为这样的类型。除非你真的就要对字节直接操作，通常，你可以把这些类型用 `load(fromByteOffset:as:)` 等方法转换成 `Unsafe[Mutable]Pointer` 或者其他确定类型的指针。

## 用可选值代表可空指针

和 C 不同，Swift 使用可选值来区分可能为空的指针和不可能为空的指针。只有那些包含指针的可选值才能表示一个空指针。在底层，`UnsafePointer<T>` 和 `Optional<UnsafePointer<T>>` 的内存结构相同；编译器会将 `Optional.none` 映射为一个所有位全为零的空指针。

## 不透明指针

有时 C API 拥有一个不透明指针类型。举例来说，在 `cmark` 库中，我们看到过 `cmark_node*` 被导入为 `OpaquePointer`。`cmark_node` 的定义并没有在头文件中暴露，所以，我们不能访问到指针指向的内存。你可以通过初始化方法将不透明指针转换为其他的指针。

Swift 中的 `OpaquePointer` 实际上在类型安全上还不如对应的 C 代码，因为从 C 中引入的不透明指针失去了它们的类型信息。在 C 中，`cmark_node*` 和 `cmark_iter*` 是不同的类型，编译器会在你把其中一个传递给希望接受另一个作为参数的函数时，会对你进行警告。Swift 则把两个类型都引入为 `OpaquePointer`，并把它们等同视之。理想情况下，`OpaquePointer` 应该是一个泛型，这样 `OpaquePointer<cmark_node>` 和 `OpaquePointer<cmark_iter>` 就能在类型系统上区分开了。

## 缓冲区指针

在 Swift 中，我们经常会使用 `Array` 来连续存储一系列值。在 C 里，数组通常用指向第一个元素的指针以及元素的个数来表示。如果要将 C 中的数组作为 Swift 中的集合类型使用，我们可以把它转换成 `Array`，但这会导致元素被复制。通常来说这是件好事（因为一旦这些元素存在于 `Array` 中，它们的内存就将由 Swift 运行时管理）。然而，有时候你却不想复制每个元素。对于那些情况，我们可以使用 `Unsafe[Mutable]BufferPointer`，它通过一个指向起始元素的指针和元素的个数来进行初始化。完成后，你就拥有一个可以随机访问的集合了。缓冲区指针让 Swift 与 C 的集合协同工作变得容易很多。

`Array` 还提供了一个 `withUnsafe[Mutable]BufferPointer` 方法，通过缓冲区指针，它让我们可以直接访问到数组的内部存储（这个操作甚至是可写的）。有了这些 API，我们就可以跳过数组的边界检查，直接对其中的元素进行批量复制或写入。这可以提高数组循环操作的性能。在 Swift

5 里，这些方法还有了泛型的版本：withContiguous[Mutable]StorageIfAvailable。但要时刻警惕，当你使用了这些方法的时候，Swift 对集合类型实施的所有常规安全检查就都失效了。

最后，是 Unsafe[Mutable]RawBufferPointer，它让我们可以直接把原始内存数据当作集合来处理（它们在底层与 Foundation 中的 Data 是等价的）。

## 把闭包用作 C 的回调函数

先用一个使用函数指针的 C API 看个具体的例子。我们的目标是用 Swift 封装 C 标准库中的 qsort 排序函数。这个函数被导入 Swift Darwin（或者如果你在用 Linux 的话，则是 Glibc）模块时，签名是这样的：

```
public func qsort(  
    __base: UnsafeMutableRawPointer!, // 要排序的数组  
    __nel: Int, // 数组中元素的个数  
    __width: Int, // 每个元素的大小  
    __compar: @escaping @convention(c) (UnsafeRawPointer?,  
        UnsafeRawPointer?) // 执行比较的函数  
    -> Int32)
```

qsort 的 man 页面 (man qsort) 描述了如何使用这个函数：

qsort() 和 heapsort() 函数可以对一个有 nel 个元素的数组进行排序，base 指针指向数组中第一个成员。数组中每个对象的尺寸由 width 规定。

base 数组的内容将基于 compar 指向的比较方法的结果进行升序排列，这个方法接受两个待比较的对象作为参数。

这里是使用 qsort 来排序 Swift 字符串数组的封装方法：

```
extension Array where Element == String {  
    mutating func quickSort() {  
        qsort(&self, count, MemoryLayout<String>.stride) { a, b in
```

```
let l = a!.assumingMemoryBound(to: String.self).pointee
let r = b!.assumingMemoryBound(to: String.self).pointee

if r > l { return -1 }
else if r == l { return 0 }
else { return 1 }

}
```

让我们研究一下传入 `qsort` 的每个参数的含义：

- 第一个参数是指向数组首个元素的指针。当你将 Swift 数组传递给一个接受 `UnsafePointer` 的函数时，它们会被自动转换为 C 风格的首元素指针。因为这个指针的类型实际上是 `UnsafeMutableRawPointer` (在 C 声明中，它是 `void *base`)，所以我们需要添加 `&` 前缀。如果 C 函数不会修改它的输入，并且参数在 C 中的声明是 `const void *base` 的话，导入到 Swift 之后，就可以不写 `&` 前缀。这和使用 Swift 函数的 `inout` 参数的规则是一样的，普通参数不用 `&` 前缀，`inout` 参数则需要使用 `&` 前缀。
- 第二个参数是元素的个数。这个很容易，我们只需要使用数组的 `count` 属性就可以了。
- 第三个参数中，我们使用了 `MemoryLayout.stride` 而非 `MemoryLayout.size` 来获取每个元素的宽度。在 Swift 中，`MemoryLayout.size` 返回的是一个类型的真实尺寸，但是对于那些在内存中的元素，平台的内存对齐规则可能会导致相邻元素之间存在空隙。`stride` 获取的是这个类型的尺寸，再加上空隙的宽度 (这个宽度可能为 0)。对于字符串来说，`size` 和 `stride` 取到的值在 Apple 的平台上恰好相同，但是这并不是说对于其他类型都会是相同的，比如 `MemoryLayout<(Int32, Bool)>.size` 是 5，而 `MemoryLayout<(Int32, Bool)>.stride` 的值是 8。当你将代码从 C 转换为 Swift 时，对于 C 中的 `sizeof`，在 Swift 中你应该使用 `MemoryLayout.stride`。
- 最后一个参数是一个指向 C 函数的指针，这个 C 函数用来比较数组中的两个元素。Swift 可以自动把一个函数类型桥接到 C 函数指针，所以我们只需要传递一个签名符合要求的函数就行了。不过，要特别提醒的是，C 函数指针仅仅只是单纯的指针，它们不能捕获任何值。因为这个原因，Swift 编译器只允许传递没有捕获任何外部状态 (例如：

没有局部变量也没有泛型参数) 的函数。在 Swift 签名中的 @convention(c) 属性就是用来保证这个前提的。

compar 函数接受两个 raw 指针。UnsafeRawPointer 这样的指针可以指向任何东西。我们之所以要处理 UnsafeRawPointer，而不是 UnsafePointer<String>，是因为 C 中没有泛型。不过，我们自己知道传递的是一个 String，所以我们可以将它解释为指向 String 的指针。这里我们还知道这些指针永远不会是 nil，所以我们能安全地将它们强制解包。最后，这个函数返回的是一个 Int32 值：正值表示第一个元素比第二个元素大，0 表示它们相等，而负数表示第一个元素比第二个元素小。

## 泛化对 C 函数的转换

为另外一种类型的 C 数组创建一个 qsort 包装也很容易，我们只需要复制粘贴这些代码，然后把 String 换成其他类型就可以了。但是我们真正需要的是通用的代码。但当我们尝试用泛型改造 quickSort 排序的时候，会遇到一个 C 函数指针带来的限制。下面的代码无法通过编译，因为这个用于比较的函数已经变成了一个闭包，它捕获了 Element 类型的比较和相等操作符，它们对于每种具体类型来说都是不同的。我们对此似乎无能为力，“不能使用闭包”是 C 语言自身带来的限制：

```
extension Array where Element: Comparable {
    mutating func quickSort() {
        // 错误：一个 C 函数指针不能由捕获了泛型参数的闭包形成
        qsort(&self, self.count, MemoryLayout<Element>.stride) { a, b in
            let l = a!.assumingMemoryBound(to: Element.self).pointee
            let r = b!.assumingMemoryBound(to: Element.self).pointee
            if r > l { return -1 }
            else if r == l { return 0 }
            else { return 1 }
        }
    }
}
```

我们也可以从编译器的视角去理解这个限制。C 函数指针存储的只是内存中的一个地址，这个地址指向的是对应的代码块。对于没有什么上下文依赖的函数来说，这个地址是静态的，并且在编译的时候就已经确定了。然而，对于泛型函数来说，其实我们还传入了一个额外的参数表示泛型类型。当把泛型参数的类型具体化之后，得到的函数是没有固定地址的。闭包的情况与此类似。即便编译器可以重写闭包，让它可以作为一个函数指针传递，和闭包相关的内存管理也无法自动完成，我们根本无法知道应该在什么时候释放这个闭包。

实际使用的时候，这也是很多 C 程序员面临的问题。在 macOS 上，有一个 `qsort` 的变种，叫做 `qsort_b`，它接受一个 `block`，也就是闭包，而不是函数指针，作为最后一个参数。如果我们把上面代码中的 `qsort` 用 `qsort_b` 替换掉的话，它就可以正常地编译和运行了。

不过，由于 `block` 并不是 C 标准的一部分，`qsort_b` 在大多数平台上都是不可用的。另外，除了 `qsort` 以外，其他函数可能也没有基于 `block` 的版本。大多数和回调相关的 C API 都提供另外一种解决方式：它们接受一个额外的 `UnsafeRawPointer` 指针作为参数，并且在调用回调函数时将这个指针再回传给调用者。这样一来，API 的用户可以在每次调用这个带有回调的函数时传递一小段随机数据进去，然后在回调中就可以判别调用者究竟是谁。`qsort` 的另一个变种 `qsort_r` 做的就是这件事情，它的函数签名中包含了一个额外的参数 `thunk`，它是一个 `UnsafeMutableRawPointer` 指针。注意这个参数也被加入到了比较函数中，因为 `qsort_r` 会在每次调用这个比较函数时将 `thunk` 传递过来：

```
public func qsort_r(  
    __base: UnsafeMutableRawPointer!,  
    __nel: Int,  
    __width: Int,  
    __thunk: UnsafeMutableRawPointer!,  
    __compar: @escaping @convention(c)  
        (UnsafeMutableRawPointer?, UnsafeRawPointer?, UnsafeRawPointer?)  
        -> Int32  
)
```

如果 `qsort_b` 在我们的目标平台不存在的话，我们可以用 `qsort_r` 在 Swift 中重新构建它。使用 `thunk` 参数可以传递任何东西，唯一的限制是我们需要将它转换为一个 `UnsafeRawPointer`

对象。在我们的例子中，我们想要传递的是执行比较的闭包。还记得之前提到过的 `Unmanaged` 吗？它就可以在原生 Swift 对象和纯指针 (raw pointer) 之间进行转换，而这正是我们期望的。由于 `Unmanaged` 只能包装 class，我们还得将闭包包装到一个类里。我们可以重用属性一章中的 `Box` 类来达到这个目的：

```
@propertyWrapper  
class Box<A> {  
    var wrappedValue: A  
  
    init(wrappedValue: A) {  
        self.wrappedValue = wrappedValue  
    }  
}
```

有了上面的代码，我们就可以编写自己的 `qsort_b` 变体了。为了和 C 命名方式保持一致，我们把它定义成 `qsort_block`：

```
typealias Comparator = (UnsafeRawPointer?, UnsafeRawPointer?) -> Int32  
func qsort_block(_ array: UnsafeMutableRawPointer, _ count: Int,  
                 _ width: Int, _ compare: @escaping Comparator)  
{  
    let box = Box(wrappedValue: compare) // 1  
    let unmanaged = Unmanaged.passRetained(box) // 2  
    defer {  
        unmanaged.release() // 6  
    }  
    qsort_r(array, count, width, unmanaged.toOpaque()) {  
        (ctx, p1, p2) -> Int32 in // 3  
        let innerUnmanaged =  
            Unmanaged<Box<Comparator>>.fromOpaque(ctx!) // 4  
        let comparator =  
            innerUnmanaged.takeUnretainedValue().wrappedValue // 4  
        return comparator(p1, p2) // 5  
    }  
}
```

```
}
```

上面的代码执行了以下步骤：

0. 我们先把用于比较的闭包“打包”成了一个 Box 实例。
1. 然后，把这个实例用 Unmanaged 进行了包装。passRetained 方法可以把 Box 实例保持在内存里，避免它被过早地释放（要记住，保持包装在 Unmanaged 中的对象存活是自己的责任）。
2. 接下来，调用 qsort\_r，并给它传递指向 Unmanaged 对象的指针（Unmanaged.toOpaque 返回一个指向其自身的纯指针）作为 thunk 参数。
3. 在 qsort\_r 的回调函数里，先把传入的纯指针转换回一个 Unmanaged 对象，从中得到它包装的 Box 对象，再从这个 Box 对象中“开箱”得到用于比较的 closure。在这个过程中，注意不要修改 Unmanaged 包装的对象的引用计数。这和前三步打包并传递闭包的过程是相反的，fromOpaque 用于返回 Unmanaged 对象，takeUnretainedValue 用于提取包装的 Box 实例，wrappedValue 用于“开箱”得到闭包。
4. 完成后，调用 comparator 比较数组中的两个元素。
5. 最后，在 qsort\_r 返回之后，box 对象就没用了。在 defer 代码块里，我们通过 release 方法释放它。这可以平衡掉我们在第二步为保持对象增加的引用计数。

现在，就可以用 qsort\_block 修改我们的 qsortWrapper 函数，并为 C 标准库中的 qsort 算法提供一个漂亮的泛型接口了：

```
extension Array where Element: Comparable {  
    mutating func quickSort() {  
        qsort_block(&self, self.count, MemoryLayout<Element>.stride) { a, b in  
            let l = a!.assumingMemoryBound(to: Element.self).pointee  
            let r = b!.assumingMemoryBound(to: Element.self).pointee  
            if r > l { return -1 }  
            else if r == l { return 0 }  
            else { return 1 }  
        }  
    }  
}
```

```
    }  
}  
  
var numbers = [3, 1, 4, 2]  
numbers.quickSort()  
numbers // [1, 2, 3, 4]
```

看起来为了使用 C 标准库中的排序算法，我们大费周章。这好像很不值得，毕竟 Swift 中内建的 sort 函数要易用得多，而且在大多数情况下也更快。不过，除了排序以外，还有很多有意思的 C API。而将它们以类型安全和泛型接口的方式进行封装所用到的技巧，与我们上面的例子是一致的。

## 回顾

从头开始将一个已经存在的 C 的库用 Swift 重写会是一件很有意思的事情，但是将你的时间花在这上面未免有些可惜（如果你是以学习为目的的话，那就很棒）。这个世界上已经有很多经过良好测试的 C 代码了，把它们都抛弃的话会是巨大的浪费。Swift 可以和 C 代码进行很好的交互，所以为什么不直接利用它们呢？话虽如此，不过毫无疑问，大部分的 C API 在 Swift 中看起来会格格不入。而且，让 C 中像是指针和手动内存管理这样的结构在你的整个代码库中蔓延，并不是什么好主意。

正如我们本章中对这个 Markdown 库所做的那样，我们可以在内部对这些不安全的部分进行封装，并暴露一个符合 Swift 语言习惯的接口。这样一来，我们既不需要重新发明轮子（也就是说，写一个完整的 Markdown 解析器），又可以让使用我们 API 的使用者感到这是百分百的原生 Swift API。

# 写在最后

16

希望你能享受这段和我们一起徜徉在 Swift 中的旅程。

虽然 Swift 还很年轻，但是它已经是一门复杂的语言了。想要在一本书里覆盖到它的方方面面，几乎是不可能的；而想让读者都能将它们记住，更是难上加难。但即使你无法马上学以致用，我们也坚信，更深入地了解你使用的编程语言可以让你成为更加优秀的程序员。

如果你只能从这本书中学到一点，我们希望让你记住 Swift 有很多高级特性可以帮助你编写更好、更安全以及更具表现力的代码。当然，你也可以用 Swift 写出和 Objective-C, Java 或 C# 感觉完全相同的代码。但我们希望这本书已经说服你了：Swift 中诸如枚举、泛型、具有一等公民身份的函数和结构化并发等特性，可以显著提高你的代码质量。

原生的并发模型也许是 Swift 最近一段时间内最大的变化了，但这并不是说这门语言就不会在继续进步了。在今后的几年中，我们期待 Swift 在下面这些领域还会有显著的增强：

- **并发模型的扩展**。编译时检查将逐渐加强，以便库的开发者有时间审核他们的代码对并发的支持。Swift 团队也已经在研究分布式 actor，这可以将 actor 模型扩展到多个进程或机器中去。
- **泛型系统的增强**。我们在本书中提到过一些这方面的增强，比如不透明类型的 where 约束，以及解除存在体的限制等。
- **明确的内存管理和所有**。其目的是为编译器提供所有需要的信息，以避免在向函数传值时进行不必要的复制，从而使 Swift 更适合于编写有严格性能要求的底层代码。
- **更强大的内省**。编译器将大量的关于类型及其属性的元数据植入二进制文件中。这些信息已经被调试工具所使用，但还没有任何公共的 API 可以访问它。这些数据的存在为更强大的反射和内省功能打开了大门，这些能力远远超过了当前的 Mirror 类型所能做到的事情。
- **字符串 API 的重大改进**。Swift 团队正在致力于正则表达式的支持，他们也在研究如何以声明式字符串处理的名义，基于 result builder 的语法来编写解析器。

如果你对这些增强或者其他特性会对 Swift 造成怎样的影响感兴趣的话，不要忘了 Swift 的开发是开放的。你可以加入到 Swift 论坛，在那里参加讨论，并发表你的观点。

最后，我们想要鼓励你好好利用 Swift 开源这一优势。当你遇到问题，但是文档无法回答的时候，通常源码会为你提供答案。如果你已经看到这里了，相信你已经可以自己找

到标准库的源码文件了。能够对事情是如何被实现的进行确认，在我们写作本书时，也起到了莫大的帮助。